Version 6.0

The LEDA User Manual

Algorithmic Solutions
## Contents

1 Preface 3

2 Basics 5
   2.1 Getting Started ................................................. 5
   2.2 The LEDA Manual Page (the type specification) ............... 6
   2.3 User Defined Parameter Types ................................ 8
      2.3.1 Linear Orders ................................................... 9
      2.3.2 Hashed Types .................................................... 12
      2.3.3 Implementation Parameters ................................ 13
   2.4 Arguments .......................................................... 13
   2.5 Items ............................................................... 14
   2.6 Iteration ............................................................. 16

3 Modules 19

4 Simple Data Types and Basic Support Operations 21
   4.1 Strings (string) .................................................. 21
   4.2 File Input Streams (file_istream) ................................. 25
   4.3 File Output Streams (file_ostream) ............................... 25
   4.4 String Input Streams (string_istream) ......................... 25
   4.5 String Output Streams (string_ostream) ....................... 25
   4.6 Random Sources (random_source) ............................... 27
   4.7 Random Variates (random_variate) .............................. 29
   4.8 Dynamic Random Variates (dynamic_random_variate) ......... 29
   4.9 Memory Management ............................................... 31
   4.10 Memory Allocator (leda_allocator) ............................. 32
   4.11 Error Handling (error) .......................................... 34
   4.12 Files and Directories (file) ................................... 36
   4.13 Sockets (leda_socket) ............................................ 38
## CONTENTS

4.14 Socket Streambuffer (socket_streambuf) .......................... 41
4.15 Some Useful Functions (misc) ........................................ 44
4.16 Timer (timer) .......................................................... 46
4.17 Counter (counter) ...................................................... 49
4.18 Two Tuples (two_tuple) ............................................... 51
4.19 Three Tuples (three_tuple) ........................................... 52
4.20 Four Tuples (four_tuple) .............................................. 53
4.21 A date interface (date) ............................................... 56

5 Alignment Data Types .................................................. 63
5.1 Alphabet (alphabet) .................................................... 63
5.2 String Matching Algorithms (string_matching) .................... 65
5.3 Score Matrix (score_matrix) .......................................... 68
5.4 String Distance Function (distance) .................................. 70
5.5 Alignment Functions (alignment) .................................... 72

6 Number Types and Linear Algebra ..................................... 75
6.1 Integers of Arbitrary Length (integer) ............................... 75
6.2 Rational Numbers (rational) .......................................... 78
6.3 The data type bigfloat (bigfloat) ..................................... 80
6.4 The data type real (real) .............................................. 85
6.5 Interval Arithmetic in LEDA (interval) ............................... 92
6.6 Modular Arithmetic in LEDA (residual) ............................... 95
6.7 The mod kernel of type residual (residual) .......................... 96
6.8 The smod kernel of type residual (residual) ........................ 97
6.9 A Floating Point Filter (floatf) ...................................... 100
6.10 Double-Valued Vectors (vector) ..................................... 102
6.11 Double-Valued Matrices (matrix) ................................... 105
6.12 Vectors with Integer Entries (integer_vector) ....................... 108
6.13 Matrices with Integer Entries (integer_matrix) ..................... 110
6.14 Rational Vectors (rat_vector) ....................................... 115
6.15 Real-Valued Vectors (real_vector) .................................. 120
6.16 Real-Valued Matrices (real_matrix) .................................. 123
6.17 Numerical Analysis Functions (numerical_analysis) .............. 125
   6.17.1 Minima and Maxima .............................................. 125
   6.17.2 Integration ..................................................... 126
   6.17.3 Useful Numerical Functions ................................... 126
   6.17.4 Root Finding .................................................. 126
## 7 Basic Data Types

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>One Dimensional Arrays (array)</td>
<td>127</td>
</tr>
<tr>
<td>7.2</td>
<td>Two Dimensional Arrays (array2)</td>
<td>131</td>
</tr>
<tr>
<td>7.3</td>
<td>Stacks (stack)</td>
<td>132</td>
</tr>
<tr>
<td>7.4</td>
<td>Queues (queue)</td>
<td>133</td>
</tr>
<tr>
<td>7.5</td>
<td>Bounded Stacks (b_stack)</td>
<td>134</td>
</tr>
<tr>
<td>7.6</td>
<td>Bounded Queues (b_queue)</td>
<td>135</td>
</tr>
<tr>
<td>7.7</td>
<td>Linear Lists (list)</td>
<td>137</td>
</tr>
<tr>
<td>7.8</td>
<td>Singly Linked Lists (slist)</td>
<td>145</td>
</tr>
<tr>
<td>7.9</td>
<td>Sets (set)</td>
<td>147</td>
</tr>
<tr>
<td>7.10</td>
<td>Integer Sets (int_set)</td>
<td>150</td>
</tr>
<tr>
<td>7.11</td>
<td>Dynamic Integer Sets (d_int_set)</td>
<td>152</td>
</tr>
<tr>
<td>7.12</td>
<td>Partitions (partition)</td>
<td>155</td>
</tr>
<tr>
<td>7.13</td>
<td>Parameterized Partitions (Partition)</td>
<td>157</td>
</tr>
<tr>
<td>7.14</td>
<td>Dynamic Trees (dynamic_trees)</td>
<td>159</td>
</tr>
<tr>
<td>7.15</td>
<td>Dynamic Collections of Trees (tree_collection)</td>
<td>161</td>
</tr>
</tbody>
</table>

## 8 Dictionary Types

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1</td>
<td>Dictionaries (dictionary)</td>
<td>163</td>
</tr>
<tr>
<td>8.2</td>
<td>Dictionary Arrays (d_array)</td>
<td>166</td>
</tr>
<tr>
<td>8.3</td>
<td>Hashing Arrays (h_array)</td>
<td>169</td>
</tr>
<tr>
<td>8.4</td>
<td>Maps (map)</td>
<td>171</td>
</tr>
<tr>
<td>8.5</td>
<td>Two-Dimensional Maps (map2)</td>
<td>173</td>
</tr>
<tr>
<td>8.6</td>
<td>Persistent Dictionaries (p_dictionary)</td>
<td>175</td>
</tr>
<tr>
<td>8.7</td>
<td>Partially Persistent Dictionaries (pp_dictionary)</td>
<td>177</td>
</tr>
<tr>
<td>8.8</td>
<td>Sorted Sequences (sortseq)</td>
<td>179</td>
</tr>
</tbody>
</table>

## 9 Priority Queues

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1</td>
<td>Priority Queues (p_queue)</td>
<td>187</td>
</tr>
<tr>
<td>9.2</td>
<td>Bounded Priority Queues (b_priority_queue)</td>
<td>190</td>
</tr>
</tbody>
</table>

## 10 Lossless Compression

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1</td>
<td>Adaptive Arithmetic Coder (A0Coder)</td>
<td>196</td>
</tr>
<tr>
<td>10.2</td>
<td>Static Arithmetic Coder (A0sCoder)</td>
<td>199</td>
</tr>
<tr>
<td>10.3</td>
<td>Prediction by Partial Matching (PPMIIICoder)</td>
<td>201</td>
</tr>
<tr>
<td>10.4</td>
<td>Deflation/Inflation Coder (DeflateCoder)</td>
<td>204</td>
</tr>
<tr>
<td>10.5</td>
<td>Dictionary Based Coder (DictCoder)</td>
<td>207</td>
</tr>
</tbody>
</table>
10.6 Static Huffman Coder (HuffmanCoder) ........................................ 209
10.7 Adaptive Huffman Coder (AdaptiveHuffmanCoder) ...................... 211
10.8 Run-Length Coder (RLECoder) .................................................. 213
10.9 Burrows-Wheeler Transform (BWTCoder) .................................... 215
10.10 Move-To-Front Coder (MTFCoder) ........................................... 218
10.11 Move-To-Front Coder II (MTF2Coder) ....................................... 220
10.12 RLE for Runs of Zero (RLE0Coder) ........................................... 222
10.13 Checksummers (checksummer_base) ......................................... 224
10.14 CRC32 Checksum (CRC32Coder) ............................................. 226
10.15 CRC-CCITT Checksum (CCITTCoder) ........................................ 226
10.16 Adler32 Checksum (Adler32Coder) ........................................... 226
10.17 MD5 Checksum (MD5SumCoder) ................................................ 227
10.18 SHA-1 Checksum (SHACoder) ................................................ 227
10.19 Encoding Output Stream (encoding_ostream) ............................... 228
10.20 Decoding Input Stream (decoding_istream) ............................... 228
10.21 Encoding File Stream (encoding_ofstream) ................................ 229
10.22 Decoding File Stream (decoding_ifstream) ................................ 230
10.23 Coder Pipes (CoderPipe2) ...................................................... 232
10.24 Automatic Decoder (AutoDecoder) ............................................ 235
10.25 Block Coder (BlockCoder) ...................................................... 237
10.26 Memory Streambuffer (memory_streambuf) ................................. 241

11 Symmetric Key Cryptography ...................................................... 243
  11.1 Secure Byte String (CryptByteString) ....................................... 248
  11.2 Key for Cryptography (CryptKey) ............................................ 249
  11.3 Encryption and Decryption with Ciphers .................................. 252
  11.4 Example for a Stream-Cipher (CBCCoder) .................................. 254
  11.5 Authentication (OMACCoder) .................................................. 257
  11.6 Automatic Decoder supporting Cryptography (CryptAutoDecoder) ... 260
  11.7 Secure Socket Streambuffer (secure_socket_streambuf) ................ 263

12 Graphs and Related Data Types .................................................. 267
  12.1 Graphs (graph) ................................................................. 267
  12.2 Parameterized Graphs (GRAPH) ............................................. 282
  12.3 Static Graphs (static_graph) ................................................ 286
  12.4 Undirected Graphs (ugraph) .................................................. 292
  12.5 Parameterized Ugraph (UGRAPH) .......................................... 292
12.6 Planar Maps (planar_map) ............................................. 294
12.7 Parameterized Planar Maps (PLANAR_MAP) ...................... 296
12.8 Node Arrays (node_array) ........................................... 298
12.9 Edge Arrays (edge_array) .......................................... 300
12.10 Face Arrays (face_array) ........................................... 302
12.11 Node Maps (node_map) .............................................. 304
12.12 Edge Maps (edge_map) .............................................. 306
12.13 Face Maps (face_map) .............................................. 308
12.14 Two Dimensional Node Arrays (node_matrix) .................... 310
12.15 Two-Dimensional Node Maps (node_map2) ....................... 312
12.16 Sets of Nodes (node_set) ........................................... 314
12.17 Sets of Edges (edge_set) ........................................... 315
12.18 Lists of Nodes (node_list) .......................................... 316
12.19 Node Partitions (node_partition) .................................. 318
12.20 Node Priority Queues (node_pq) .................................. 319
12.21 Bounded Node Priority Queues (b_node_pq) ....................... 321
12.22 Graph Generators (graph_gen) ..................................... 323
12.23 Miscellaneous Graph Functions (graph_m misc) .................. 328
12.24 Markov Chains (markov_chain) .................................... 332
12.25 Dynamic Markov Chains (dynamic_markov_chain) .............. 333
12.26 GML Parser for Graphs (gml_graph) ............................. 334
12.27 The LEDA graph input/output format ............................. 339

13 Graph Algorithms ......................................................... 341
13.1 Basic Graph Algorithms (basic_graph_alg) ....................... 342
13.2 Shortest Path Algorithms (shortest_path) ......................... 345
13.3 Maximum Flow (max_flow) ............................................ 349
13.4 Min Cost Flow Algorithms (min_cost_flow) ....................... 353
13.5 Minimum Cut (min_cut) ............................................... 354
13.6 Maximum Cardinality Matchings in Bipartite Graphs (mc b_matching) 355
13.7 Bipartite Weighted Matchings and Assignments (mwb_matching) 356
13.8 Maximum Cardinality Matchings in General Graphs (mc_matching) 360
13.9 General Weighted Matchings (mw_matching) ...................... 361
13.10 Stable Matching (stable_matching) ................................ 368
13.11 Minimum Spanning Trees (min_span) .............................. 370
13.12 Euler Tours (euler_tour) ............................................ 371
13.13 Algorithms for Planar Graphs (plane_graph_alg) ............... 372
13.14 Graph Drawing Algorithms (graph_draw) ......................... 375
13.15 Graph Morphism Algorithms (graph_morphism) .................. 378
13.16 Graph Morphism Algorithm Functionality (graph_morphism_algorithm) 379
14 Graphs and Iterators 387
14.1 Introduction .................................................... 387
14.1.1 Iterators ...................................................... 387
14.1.2 Handles and Iterators ........................................ 388
14.1.3 STL Iterators .................................................. 388
14.1.4 Circulators ..................................................... 389
14.1.5 Data Accessors ............................................... 389
14.1.6 Graphiterator Algorithms ................................. 391
14.2 Node Iterators (NodeIt) ........................................ 393
14.3 Edge Iterators (EdgeIt) ........................................ 395
14.4 Face Iterators (FaceIt) ......................................... 396
14.5 Adjacency Iterators for leaving edges (OutAdjIt) .......... 398
14.6 Adjacency Iterators for incoming edges (InAdjIt) ........ 401
14.7 Adjacency Iterators (AdjIt) ................................... 403
14.8 Face Circulators (FaceCirc) .................................... 406
14.9 Filter Node Iterator (FilterNodeIt) .......................... 408
14.10 Comparison Predicate (CompPred) .......................... 409
14.11 Observer Node Iterator (ObserverNodeIt) .................. 411
14.12 STL Iterator Wrapper (STLNodeIt) ........................ 413
14.13 Node Array Data Accessor (node_array_da) ............... 415
14.14 Constant Accessors (constant_da) .......................... 417
14.15 Node Member Accessors (node_member_da) ............... 417
14.16 Node Attribute Accessors (node_attribute_da) ........... 419
14.17 Breadth First Search (flexible) (GIT_BFS) ................ 420
14.18 Depth First Search (flexible) (GIT_DFS) ................. 422
14.19 Topological Sort (flexible) (GIT_TOPOSORT) ............ 424
14.20 Strongly Connected Components (flexible) (GIT_SCC) ... 426
14.21 Dijkstra (flexible) (GIT_DIJKSTRA) ....................... 428

15 Basic Data Types for Two-Dimensional Geometry 431
15.1 Points (point) .................................................. 432
15.2 Segments (segment) ............................................ 437
15.3 Straight Rays (ray) ............................................. 441
15.4 Straight Lines (line) ............................................ 444
15.5 Circles (circle) .................................................. 448
15.6 Polygons (POLYGON) ........................................... 452
## 15.7 Generalized Polygons (GEN_POLYGON) ............................................ 458

## 15.8 Triangles (triangle) ................................................................. 465

## 15.9 Iso-oriented Rectangles (rectangle) ............................................. 468

## 15.10 Rational Points (rat_point) ....................................................... 471

## 15.11 Rational Segments (rat_segment) ............................................... 476

## 15.12 Rational Rays (rat_ray) ............................................................ 481

## 15.13 Straight Rational Lines (rat_line) ............................................... 484

## 15.14 Rational Circles (rat_circle) ...................................................... 488

## 15.15 Rational Triangles (rat_triangle) ............................................... 491

## 15.16 Iso-oriented Rational Rectangles (rat_rectangle) .......................... 494

## 15.17 Real Points (real_point) ............................................................. 498

## 15.18 Real Segments (real_segment) .................................................... 503

## 15.19 Real Rays (real_ray) ............................................................... 507

## 15.20 Straight Real Lines (real_line) ................................................... 510

## 15.21 Real Circles (real_circle) .......................................................... 514

## 15.22 Real Triangles (real_triangle) .................................................... 518

## 15.23 Iso-oriented Real Rectangles (real_rectangle) ............................... 521

## 15.24 Geometry Algorithms (geo_alg) ................................................ 525

## 15.25 Transformation (TRANSFORM) ................................................... 536

## 15.26 Point Generators (point generators) ........................................... 539

## 15.27 Point on Rational Circle (r_circle_point) ..................................... 543

## 15.28 Segment of Rational Circle (r_circle_segment) ............................... 545

## 15.29 Polygons with circular edges (r_circle_polygon) ............................ 550

## 15.30 Generalized polygons with circular edges (r_circle_gen_polygon) ...... 556

## 15.31 Parser for well known binary format (wkb_io) ............................... 564

### 16 Advanced Data Types for Two-Dimensional Geometry

#### 16.1 Two-Dimensional Dictionaries (d2_dictionary) ............................ 565

#### 16.2 Point Sets and Delaunay Triangulations (POINT_SET) .................. 568

#### 16.3 Sets of Intervals (interval_set) ................................................ 574

#### 16.4 Sets of Parallel Segments (segment_set) .................................... 576

#### 16.5 Sets of Parallel Rational Segments (rat_segment_set) ................... 578

#### 16.6 Planar Subdivisions (subdivision) ............................................. 580

### 17 Basic Data Types for Three-Dimensional Geometry

#### 17.1 Points in 3D-Space (d3_point) ................................................ 582

#### 17.2 Straight Rays in 3D-Space (d3_ray) .......................................... 587
CONTENTS

17.3 Segments in 3D-Space (d3_segment) .............................................. 589
17.4 Straight Lines in 3D-Space (d3_line) .............................................. 591
17.5 Planes (d3_plane) .......................................................... 593
17.6 Spheres in 3D-Space (d3_sphere) ................................................. 596
17.7 Simplices in 3D-Space (d3_simplex) .............................................. 598
17.8 Rational Points in 3D-Space (d3_rat_point) ...................................... 600
17.9 Straight Rational Rays in 3D-Space (d3_rat_ray) ............................... 609
17.10 Rational Lines in 3D-Space (d3_rat_line) ....................................... 611
17.11 Rational Segments in 3D-Space (d3_rat_segment) ............................. 614
17.12 Rational Planes (d3_rat_plane) ................................................... 617
17.13 Rational Spheres (d3_rat_sphere) ................................................. 620
17.14 Rational Simplices (d3_rat_simplex) ............................................. 622
17.15 3D Convex Hull Algorithms (d3_hull) ........................................... 624
17.16 3D Triangulation and Voronoi Diagram Algorithms (d3_delaunay) ....... 625

18 Graphics .............................................................................. 627
18.1 Colors (color) ................................................................. 627
18.2 Windows (window) .......................................................... 629
18.3 Panels (panel) ................................................................. 664
18.4 Menues (menu) ................................................................. 665
18.5 Postscript Files (ps_file) ...................................................... 667
18.6 Graph Windows (GraphWin) ..................................................... 668
18.7 The GraphWin (GW) File Format .................................................. 686
18.7.1 A complete example ......................................................... 691
18.8 Geometry Windows (GeoWin) ...................................................... 693
18.9 Windows for 3d visualization (d3_window) ....................................... 729

19 Implementations ...................................................................... 733
19.1 List of data structures .......................................................... 733
19.1.1 Dictionaries ........................................................................ 733
19.1.2 Priority Queues ..................................................................... 733
19.1.3 Geometry ............................................................................. 733
19.2 User Implementations .............................................................. 734
19.2.1 Dictionaries ........................................................................ 734
19.2.2 Priority Queues ..................................................................... 736
19.2.3 Sorted Sequences ................................................................. 737
CONTENTS

A  Technical Information 739
   A.1 LEDA Library and Packages ........................................... 739
   A.2 Contents of a LEDA Source Code Package ............................ 739
   A.3 Source Code on UNIX Platforms ...................................... 740
   A.4 Source Code on Windows with MS Visual C++ ...................... 741
   A.5 Compiling LEDA for Multithreading .................................. 742
   A.6 Usage of Header Files ................................................ 742
   A.7 Object Code on UNIX .................................................. 742
   A.8 Static Libraries for MS Visual C++ .NET ............................ 744
   A.9 DLL’s for MS Visual C++ .NET ......................................... 749
   A.10 Namespaces and Interaction with other Libraries .................. 754
   A.11 Platforms ............................................................ 754

B  The golden LEDA rules 755
   B.1 The LEDA rules in detail .............................................. 755
   B.2 Code examples for the LEDA rules .................................... 757

C  The LEDA Tools for Manual Production and Documentation 761
License Terms and Availability

Any use of LEDA requires a license which is distributed by

Algorithmic Solutions Software GmbH
Schützenstrasse 3–5
66123 Saarbrücken
Germany

phone: +49 681 876470
fax: +49 681 8764729
email: leda@algorithmic-solutions.com

For the current price list, license terms, available packages, supported platforms, and other information please visit

http://www.algorithmic-solutions.com

and follow the corresponding link.
Chapter 1

Preface

One of the major differences between combinatorial computing and other areas of computing such as statistics, numerical analysis and linear programming is the use of complex data types. Whilst the built-in types, such as integers, reals, vectors, and matrices, usually suffice in the other areas, combinatorial computing relies heavily on types like stacks, queues, dictionaries, sequences, sorted sequences, priority queues, graphs, points, segments, ... In the fall of 1988, we started a project (called LEDA for Library of Efficient Data types and Algorithms) to build a small, but growing library of data types and algorithms in a form which allows them to be used by non-experts. We hope that the system will narrow the gap between algorithms research, teaching, and implementation. The main features of LEDA are:

1. LEDA provides a sizable collection of data types and algorithms in a form which allows them to be used by non-experts. This collection includes most of the data types and algorithms described in the text books of the area.

2. LEDA gives a precise and readable specification for each of the data types and algorithms mentioned above. The specifications are short (typically, not more than a page), general (so as to allow several implementations), and abstract (so as to hide all details of the implementation).

3. For many efficient data structures access by position is important. In LEDA, we use an item concept to cast positions into an abstract form. We mention that most of the specifications given in the LEDA manual use this concept, i.e., the concept is adequate for the description of many data types.

4. LEDA contains efficient implementations for each of the data types, e.g., Fibonacci heaps for priority queues, skip lists and dynamic perfect hashing for dictionaries, ...

5. LEDA contains a comfortable data type graph. It offers the standard iterations such as “for all nodes v of a graph G do” or “for all neighbors w of v do”, it allows to add and delete vertices and edges and it offers arrays and matrices indexed by nodes and edges,... The data type graph allows to write programs for graph problems in a form close to the typical text book presentation.

6. LEDA is implemented by a C++ class library. It can be used with almost any C++ compiler that supports templates.

This manual contains the specifications of all data types and algorithms currently available in LEDA. Users should be familiar with the C++ programming language (see [83] or [56]).

The manual is structured as follows: In Chapter 2, which is a prerequisite for all other chapters, we discuss the basic concepts and notations used in LEDA. New users of LEDA should carefully read Section 2.3 to avoid problems when plugging in self defined parameter types. If you want to get information about the LEDA documentation scheme please read Section C. For technical information concerning the installation and usage of LEDA users should refer to Chapter A. There is also a section describing namespaces and the interaction with other software libraries (Section A.10). The other chapters define the data types and algorithms available in LEDA and give examples of their use. These chapters can be consulted independently from one another.

More information about LEDA can be found on the LEDA web page:
http://www.algorithmic-solutions.com/leda/

Finally there’s a tool called x1man which allows online help and demonstration on all unix platforms having a \LaTeX{} package installed.

**New in Version 6.0**

Please read the CHANGES and FIXES files in the LEDA root directory for more information.
Chapter 2

Basics

An extended version of this chapter is available as chapter Foundations of [64]

2.1 Getting Started

Please use your favourite text editor to create a file prog.c with the following program:

```c
#include <LEDA/core/d_array.h>
#include <LEDA/core/string.h>
#include <iostream>

using std::cin;
using std::cout;
using std::endl;
using leda::string;
using leda::d_array;

int main()
{
    d_array<string,int> N(0);
    string s;
    while (cin >> s) N[s]++;
    forall_defined (s,N)
        cout << s << " " << N[s] << endl;

    return 0;
}
```

If you followed the installation guidelines (see Chapter A ff.), you can compile and link it with LEDA’s library libleda (cf. Section A.1). For example, on a Unix machine where g++ is installed you can type

```
g++ -o prog prog.c -lleda -lX11 -lm
```
When executed it reads a sequence of strings from the standard input and then prints the number of occurrences of each string on the standard output. More examples of LEDA programs can be found throughout this manual. The program above uses the parameterized data type dictionary array \(d\_array\langle I,E\rangle\) from the library. This is expressed by the include statement (cf. Section A.6 for more details). The specification of the data type \(d\_array\) can be found in Section 8.2. We use it also as a running example to discuss the principles underlying LEDA in the following sections.

### 2.2 The LEDA Manual Page (the type specification)

In general the specification of a LEDA data type consists of five parts: a definition of the set of objects comprising the (parameterized) abstract data type, a list of all local types of the data type, a description of how to create an object of the data type, the definition of the operations available on the objects of the data type, and finally, information about the implementation. The five parts appear under the headers definition, types, creation, operations, and implementation, respectively. Sometimes there is also a fifth part showing an example.

- **Definition**
  This part of the specification defines the objects (also called instances or elements) comprising the data type using standard mathematical concepts and notation.

  **Example**
  The generic data type dictionary array:

  An object \(a\) of type \(d\_array\langle I,E\rangle\) is an injective function from the data type \(I\) to the set of variables of data type \(E\). The types \(I\) and \(E\) are called the index and the element type, respectively. \(a\) is called a dictionary array from \(I\) to \(E\).

  Note that the types \(I\) and \(E\) are parameters in the definition above. Any built-in, pointer, item, or user-defined class type \(T\) can be used as actual type parameter of a parameterized data type. Class types however have to provide several operations listed in Chapter 2.3.

- **Types**
  This section gives the list of all local types of the data type. For example,

  \[
  \begin{align*}
  d\_array\langle I,E\rangle::item & \quad \text{the item type.} \\
  d\_array\langle I,E\rangle::index\_type & \quad \text{the index type.} \\
  d\_array\langle I,E\rangle::element\_type & \quad \text{the element type.}
  \end{align*}
  \]

- **Creation**
  A variable of a data type is introduced by a C++ variable declaration. For all LEDA data types variables are initialized at the time of declaration. In many cases the user has to provide arguments used for the initialization of the variable. In general a declaration
XYZ< t1, ... , tk> y(x1, ... , xt);

introduces a variable \( y \) of the data type \( \text{XYZ< t1, ... , tk> } \) and uses the arguments \( x1, ... , xt \) to initialize it. For example,

\[
\text{h_array<string,int> A(0);}
\]

introduces \( A \) as a dictionary array from strings to integers, and initializes \( A \) as follows: an injective function \( a \) from \text{string} to the set of unused variables of type \text{int} is constructed, and is assigned to \( A \). Moreover, all variables in the range of \( a \) are initialized to 0. The reader may wonder how LEDA handles an array of infinite size. The solution is, of course, that only that part of \( A \) is explicitly stored which has been accessed already.

For all data types, the assignment operator (=) is available for variables of that type. Note however that assignment is in general not a constant time operation, e.g., if \( L1 \) and \( L2 \) are variables of type \text{list<T>} then the assignment \( L1 = L2 \) takes time proportional to the length of the list \( L2 \) times the time required for copying an object of type \( T \).

**Remark:** For most of the complex data types of LEDA, e.g., dictionaries, lists, and priority queues, it is convenient to interpret a variable name as the name for an object of the data type which evolves over time by means of the operations applied to it. This is appropriate, whenever the operations on a data type only “modify” the values of variables, e.g., it is more natural to say an operation on a dictionary \( D \) modifies \( D \) than to say that it takes the old value of \( D \), constructs a new dictionary out of it, and assigns the new value to \( D \). Of course, both interpretations are equivalent. From this more object-oriented point of view, a variable declaration, e.g., \text{dictionary<string,int> D}; hence the name “Creation” for this part of a specification.

### Operations

In this section the operations of the data types are described. For each operation the description consists of two parts

1. The interface of the operation is defined using the C++ function declaration syntax. In this syntax the result type of the operation (\textit{void} if there is no result) is followed by the operation name and an argument list specifying the type of each argument. For example,

\[
\text{list_item L.insert (E x, list_item it, int dir = leda::after)}
\]

defines the interface of the insert operation on a list \( L \) of elements of type \( E \) (cf. Section 7.7). Insert takes as arguments an element \( x \) of type \( E \), a \text{list_item it} and an optional relative position argument \( dir \). It returns a \text{list_item} as result.

\[
\text{E& A[I x]}
\]
CHAPTER 2. BASICS

defines the interface of the access operation on a dictionary array \( A \). It takes
an element \( x \) of type \( I \) as an argument and returns a variable of type \( E \).

2. The effect of the operation is defined. Often the arguments have to fulfill
certain preconditions. If such a condition is violated the effect of the operation
is undefined. Some, but not all, of these cases result in error messages and
abnormal termination of the program (see also Section 4.11). For the insert
operation on lists this definition reads:

A new item with contents \( x \) is inserted after (if \( \text{dir} = \text{leda::after} \)) or
before (if \( \text{dir} = \text{leda::before} \)) item \( \text{it} \) into \( L \). The new item is returned.

Precondition: item \( \text{it} \) must be in \( L \).

For the access operation on dictionary arrays the definition reads:
returns the variable \( A(x) \).

- Implementation

The implementation section lists the (default) data structures used to implement the
data type and gives the time bounds for the operations and the space requirement.
For example,

Dictionary arrays are implemented by randomized search trees ([2]). Access opera-
tions \( A[x] \) take time \( O(\log \text{dom}(A)) \). The space requirement is \( O(\text{dom}(A)) \).

2.3 User Defined Parameter Types

If a user defined class type \( T \) shall be used as actual type parameter in a container class,
it has to provide the following operations:

a) a constructor taking no arguments \( T :: T() \)
b) a copy constructor \( T :: T(const T&) \)
c) an assignment operator \( T& T :: \text{operator} = (const T&) \)
d) an input operator \( \text{istream}& \text{operator} >> (\text{istream}&, T&) \)
e) an output operator \( \text{ostream}& \text{operator} << (\text{ostream}&, const T&) \)

and if required by the parameterized data type

f) a compare function \( \text{int compare}(const T&, const T&) \)
g) a hash function \( \text{int Hash}(const T&) \)

Notice: Starting with version 4.4 of LEDA, the operations "compare" and
"Hash" for a user defined type need to be defined inside the "namespace leda"!

In the following two subsections we explain the background of the required compare and
hash function. Section 2.3.3 concerns a very special parameter type, namely implemen-
tation parameters.
2.3. USER DEFINED PARAMETER TYPES

2.3.1 Linear Orders

Many data types, such as dictionaries, priority queues, and sorted sequences require linearly ordered parameter types. Whenever a type $T$ is used in such a situation, e.g. in dictionary<$T,...>$ the function

```cpp
int compare(const T&, const T&)
```

must be declared and must define a linear order on the data type $T$.

A binary relation $\text{rel}$ on a set $T$ is called a linear order on $T$ if for all $x,y,z$ in $T$:

1) $x \text{ rel } x$
2) $x \text{ rel } y$ and $y \text{ rel } z$ implies $x \text{ rel } z$
3) $x \text{ rel } y$ or $y \text{ rel } x$
4) $x \text{ rel } y$ and $y \text{ rel } x$ implies $x=y$

A function $\text{int compare}(\text{const }T\&\text{, const }T\&)$ defines the linear order $\text{rel}$ on $T$ if

```cpp
\text{compare}(x, y) \begin{cases} < 0, & \text{if } x \text{ rel } y \text{ and } x \neq y \\ = 0, & \text{if } x = y \\ > 0, & \text{if } y \text{ rel } x \text{ and } x \neq y \end{cases}
```

For each of the data types $\text{char}$, $\text{short}$, $\text{int}$, $\text{long}$, $\text{float}$, $\text{double}$, $\text{integer}$, $\text{rational}$, $\text{bigfloat}$, $\text{real}$, $\text{string}$, and $\text{point}$ a function $\text{compare}$ is predefined and defines the so-called default ordering on that type. The default ordering is the usual $\leq$ - order for the built-in numerical types, the lexicographic ordering for $\text{string}$, and for $\text{point}$ the lexicographic ordering of the cartesian coordinates. For all other types $T$ there is no default ordering, and the user has to provide a $\text{compare}$ function whenever a linear order on $T$ is required.

**Example:** Suppose pairs of double numbers shall be used as keys in a dictionary with the lexicographic order of their components. First we declare class $\text{pair}$ as the type of pairs of double numbers, then we define the I/O operations operator$\gg$ and operator$\ll$ and the lexicographic order on $\text{pair}$ by writing an appropriate $\text{compare}$ function.

```cpp
class pair {
    double  x;
    double  y;

public:
    pair() { x = y = 0; }
    pair(const pair& p) { x = p.x; y = p.y; }
    pair& operator=(const pair& p)
    {
        if(this != &p)
            { x = p.x; y = p.y; }
        return *this;
    }
```
double get_x() {return x;}
double get_y() {return y;}

friend istream& operator>>(istream& is, pair& p)
{ is >> p.x >> p.y; return is; }
friend ostream& operator<<(ostream& os, const pair& p)
{ os << p.x << " " << p.y; return os; }
}

namespace leda {

int compare(const pair& p, const pair& q)
{
    if (p.get_x() < q.get_x()) return -1;
    if (p.get_x() > q.get_x()) return 1;
    if (p.get_y() < q.get_y()) return -1;
    if (p.get_y() > q.get_y()) return 1;
    return 0;
}
}

Now we can use dictionaries with key type pair, e.g.,

dictionary<pair,int> D;

Sometimes, a user may need additional linear orders on a data type T which are different from the order defined by compare. In the following example a user wants to order points in the plane by the lexicographic ordering of their cartesian coordinates and by their polar coordinates. The former ordering is the default ordering for points. The user can introduce an alternative ordering on the data type point (cf. Section 15) by defining an appropriate compare function (in namespace leda)

int pol_cmp(const point& x, const point& y)
{ /* lexicographic ordering on polar coordinates */ }

Now she has several possibilities:

1. First she can call the macro

    DEFINE_LINEAR_ORDER(point, pol_cmp, pol_point)

After this call pol_point is a new data type which is equivalent to the data type point, with the only exception that if pol_point is used as an actual parameter e.g. in dictionary<pol_point,...>, the resulting data type is based on the linear order defined by pol_cmp. Now, dictionaries based on either ordering can be used.
2.3. USER DEFINED PARAMETER TYPES

\[
\begin{align*}
dictionary<point,int> & \quad D0; \quad // \text{default ordering} \\
dictionary<pol\_point,int> & \quad D1; \quad // \text{polar ordering}
\end{align*}
\]

In general the macro call

\[
\text{DEFINE\_LINEAR\_ORDER}(T, \text{cmp}, T1)
\]

introduces a new type \(T1\) equivalent to type \(T\) with the linear order defined by the compare function \(\text{cmp}\).

2. As a new feature all order based data types like dictionaries, priority queues, and sorted sequences offer a constructor which allows a user to set the internally used ordering at construction time.

\[
\begin{align*}
dictionary<point,int> & \quad D0; \quad // \text{default ordering} \\
dictionary<point,int> & \quad D1(\text{pol\_cmp}); \quad // \text{polar ordering}
\end{align*}
\]

This alternative handles the cases where two or more different orderings are needed more elegantly.

3. Instead of passing a compare function \(\text{cmp}(\text{const } T&, \text{const } T&)\) to the sorted type one can also pass an object (a so-called compare object) of a class that is derived from the class \textit{leda\_cmp\_base} and that overloads the function-call operator \(\text{int operator\_}(\text{const } T&, \text{const } T&)\) to define a linear order for \(T\). This variant is helpful when the compare function depends on a global parameter. We give an example. More examples can be found in several sections of the LEDA book [64]. Assume that we want to compare edges of a graph \(\textit{GRAPH}<\textit{point}, \textit{int}>\) (in this type every node has an associated point in the plane; the point associated with a node \(v\) is accessed as \(G[v]\)) according to the distance of their endpoints. We write

\[
\begin{align*}
\text{using namespace leda}; \\
\text{class cmp\_edges\_by\_length: public leda\_cmp\_base\_edge} & \{ \\
\text{\quad const GRAPH<point,int>& G;} \\
\text{\quad public:} \\
\text{\quad \quad cmp\_edges\_by\_length(const GRAPH<point,int>& g): G(g)\{} \\
\text{\quad \quad int operator\_}(\text{const edge}& e, \text{const edge}& f) \text{ const} \\
\text{\quad \quad \quad \{} \text{ point pe } = \text{G[G.source(e)]}; \text{ point qe } = \text{G[G.target(e)]}; \\
\text{\quad \quad \quad \quad \quad \quad \quad point pf } = \text{G[G.source(f)]}; \text{ point qf } = \text{G[G.target(f)]}; \\
\text{\quad \quad \quad \quad \quad \quad \quad \text{ return compare(pe.sqr\_dist(qe),pf.sqr\_dist(qf));} \\
\text{\quad \quad \}\}; \\
\text{\quad \};} \\
\text{\};} \\
\text{\};}
\end{align*}
\]

\[
\begin{align*}
\text{int main()}\{ \\
\text{\quad GRAPH<point,int> G;} \\
\end{align*}
\]
The class `cmp_edges_by_length` has a function operator that takes two edges `e` and `f` of a graph `G` and compares them according to their length. The graph `G` is a parameter of the constructor. In the main program we define `cmp(G)` as an instance of `cmp_edges_by_length` and then pass `cmp` as the compare object to the sort function of `list<edge>`. In the implementation of the sort function a comparison between two edges is made by writing `cmp(e, f)`, i.e., for the body of the sort function there is no difference whether a function or a compare object is passed to it.

### 2.3.2 Hashed Types

LEDA also contains parameterized data types requiring a hash function and an equality test (operator==) for the actual type parameters. Examples are dictionaries implemented by hashing with chaining (`dictionary<K,I,ch_hashing>`) or hashing arrays (`h_array<I,E>`). Whenever a type `T` is used in such a context, e.g., in `h_array<T,...>` there must be defined

1. a hash function `int Hash(const T&)`
2. the equality test `bool operator == (const T&, const T&)`

Hash maps the elements of type `T` to integers. It is not required that `Hash` is a perfect hash function, i.e., it has not to be injective. However, the performance of the underlying implementations very strongly depends on the ability of the function to keep different elements of `T` apart by assigning them different integers. Typically, a search operation in a hashing implementation takes time linear in the maximal size of any subset whose elements are assigned the same hash value. For each of the simple numerical data types char, short, int, long there is a predefined `Hash` function: the identity function.

We demonstrate the use of `Hash` and a data type based on hashing by extending the example from the previous section. Suppose we want to associate information with values of the `pair` class by using a hashing array `h_array<pair,int> A`. We first define a hash function that assigns each pair `(x, y)` the integral part of the first component `x`

```cpp
namespace leda {
    int Hash(const pair& p) { return int(p.get_x()); }
};
```

and then we can use a hashing array with index type `pair`

```cpp
h_array<pair, int> A;
```
2.3.3 Implementation Parameters

Many of the parameterized data type templates (e.g., dictionary, priority queue, d_array, and sortseq) take an optional data structure parameter for choosing a particular implementation (e.g., d_array<I,E,impl>). We can easily modify the example program from Section 2.1 to use a dictionary array implemented by a particular data structure, e.g., skip lists, instead of the default data structure (cf. Section 19.1).

```cpp
#include <LEDA/core/d_array.h>
#include <LEDA/core/impl/skiplist.h>

using namespace leda;

using std::cin;
using std::cout;
using std::endl;

int main()
{
    d_array<string, int, skiplist> N(0);
    string s;

    while (cin >> s) N[s]++;

    forall_defined(s,N)
        cout << s << " " << N[s] << endl;

    return 0;
}
```

LEDA offers several implementations for each of the data types. For instance, skip lists, randomized search trees, and red-black trees for dictionary arrays. Users can also provide their own implementation. A data structure xyz_impl can be used as actual implementation parameter for a data type XYZ if it provides a certain set of operations and uses certain virtual functions for type dependent operations (e.g., compare, initialize, copy, ...). Chapter 19 lists all data structures contained in the current version and gives the exact requirements for implementations of dictionaries, priority queues, sorted sequences and dictionary arrays.

2.4 Arguments

• Optional Arguments

The trailing arguments in the argument list of an operation may be optional. If these trailing arguments are missing in a call of an operation the default argument values given in the specification are used. For example, if the relative position argument in the list insert operation is missing it is assumed to have the value leda::after, i.e., L.insert(it, y) will insert the item <y> after item it into L.
• **Argument Passing**

There are two kinds of argument passing in C++, by value and by reference. An argument $x$ of type `type` specified by "`type` x" in the argument list of an operation or user defined function will be passed by value, i.e., the operation or function is provided with a copy of $x$. The syntax for specifying an argument passed by reference is "`type&` x". In this case the operation or function works directly on $x$ (the variable $x$ is passed not its value).

Passing by reference must always be used if the operation is to change the value of the argument. It should always be used for passing large objects such as lists, arrays, graphs and other LEDA data types to functions. Otherwise a complete copy of the actual argument is made, which takes time proportional to its size, whereas passing by reference always takes constant time.

• **Functions as Arguments**

Some operations take functions as arguments. For instance the bucket sort operation on lists requires a function which maps the elements of the list into an interval of integers. We use the C++ syntax to define the type of a function argument $f$:

$$T \ (\ast f)(T_1, T_2, \ldots, T_k)$$

declares $f$ to be a function taking $k$ arguments of the data types $T_1, \ldots, T_k$, respectively, and returning a result of type $T$, i.e,

$$f : T_1 \times \ldots \times T_k \rightarrow T$$

### 2.5 Items

Many of the advanced data types in LEDA (dictionaries, priority queues, graphs, ...), are defined in terms of so-called items. An item is a container which can hold an object relevant for the data type. For example, in the case of dictionaries a `dic_item` contains a pair consisting of a key and an information. A general definition of items is given at the end of this section.

**Remark:** Item types are, like all other types, functions, constants, ..., defined in the "namespace leda" in LEDA-4.5.

We now discuss the role of items for the dictionary example in some detail. A popular specification of dictionaries defines a dictionary as a partial function from some type $K$ to some other type $I$, or alternatively, as a set of pairs from $K \times I$, i.e., as the graph of the function. In an implementation each pair $(k, i)$ in the dictionary is stored in some location of the memory. Efficiency dictates that the pair $(k, i)$ cannot only be accessed through the key $k$ but sometimes also through the location where it is stored, e.g., we might want to lookup the information $i$ associated with key $k$ (this involves a search in the data structure), then compute with the value $i$ a new value $i'$, and finally associate the new value with $k$. This either involves another search in the data structure or, if the
lookup returned the location where the pair \((k, i)\) is stored, can be done by direct access. Of course, the second solution is more efficient and we therefore wanted to provide it in LEDA.

In LEDA items play the role of positions or locations in data structures. Thus an object of type `dictionary<K,I>`, where \(K\) and \(I\) are types, is defined as a collection of items (type `dic_item`) where each item contains a pair in \(K \times I\). We use \(<k, i>\) to denote an item with key \(k\) and information \(i\) and require that for each \(k\) in \(K\) there is at most one \(i\) in \(I\) such that \(<k, i>\) is in the dictionary. In mathematical terms this definition may be rephrased as follows: A dictionary \(d\) is a partial function from the set `dic_item` to the set \(K \times I\). Moreover, for each \(k\) in \(K\) there is at most one \(i\) in \(I\) such that the pair \((k, i)\) is in \(d\).

The functionality of the operations

\[
\text{dic_item D.lookup(K k)}
\]

\[
\text{I D.inf(dic_item it)}
\]

\[
\text{void D.change_inf(dic_item it, I i')}
\]

is now as follows: \(D.lookup(K k)\) returns an item \(it\) with contents \((k, i)\), \(D.inf(it)\) extracts \(i\) from \(it\), and a new value \(i'\) can be associated with \(k\) by \(D.change_inf(it, i')\).

Let us have a look at the insert operation for dictionaries next:

\[
\text{dic_item D.insert(K k, I i)}
\]

There are two cases to consider. If \(D\) contains an item \(it\) with contents \((k, i')\) then \(i'\) is replaced by \(i\) and \(it\) is returned. If \(D\) contains no such item, then a new item, i.e., an item which is not contained in any dictionary, is added to \(D\), this item is made to contain \((k, i)\) and is returned. In this manual (cf. Section 8.1) all of this is abbreviated to

\[
\text{dic_item D.insert(K k, I i)}
\]

associates the information \(i\) with the key \(k\). If there is an item \(<k, j>\) in \(D\) then \(j\) is replaced by \(i\), else a new item \(<k, i>\) is added to \(D\). In both cases the item is returned.

We now turn to a general discussion. With some LEDA types \(XYZ\) there is an associated type `XYZ_item` of items. Nothing is known about the objects of type `XYZ_item` except that there are infinitely many of them. The only operations available on `XYZ_items` besides the one defined in the specification of type `XYZ` is the equality predicate `"=="` and the assignment operator `"="`. The objects of type `XYZ` are defined as sets or sequences of `XYZ_items` containing objects of some other type \(Z\). In this situation an `XYZ_item` containing an object \(z\) in \(Z\) is denoted by `<z>`. A new or unused `XYZ_item` is any `XYZ_item` which is not part of any object of type `XYZ`.

**Remark:** For some readers it may be useful to interpret a `dic_item` as a pointer to a variable of type \(K \times I\). The differences are that the assignment to the variable contained in a `dic_item` is restricted, e.g., the \(K\)-component cannot be changed, and that in return for this restriction the access to `dic_items` is more flexible than for ordinary variables, e.g., access through the value of the \(K\)-component is possible.
2.6 Iteration

For many (container) types LEDA provides iteration macros. These macros can be used to iterate over the elements of lists, sets and dictionaries or the nodes and edges of a graph. Iteration macros can be used similarly to the C++ for statement. Examples are

- for all item based data types:
  
  ```cpp
  forall_items(it, D) { the items of D are successively assigned to variable it }
  forall_rev_items(it, D) { the items of D are assigned to it in reverse order }
  ```

- for lists and sets:
  
  ```cpp
  forall(x, L) { the elements of L are successively assigned to x}
  forall_rev(x, L) { the elements of L are assigned to x in reverse order}
  ```

- for graphs:
  
  ```cpp
  forall_nodes(v, G) { the nodes of G are successively assigned to v}
  forall_edges(e, G) { the edges of G are successively assigned to e}
  forall_adj_edges(e, v) { all edges adjacent to v are successively assigned to e}
  ```

PLEASE NOTE:

Inside the body of a forall loop insertions into or deletions from the corresponding container are not allowed, with one exception, the current item or object of the iteration may be removed, as in

```cpp
forall_edges(e,G) {
  if (source(e) == target(e)) G.del_edge(e);
} // remove selfloops
```

The item based data types `list`, `array`, and `dictionary` provide now also an STL compatible iteration scheme. The following example shows STL iteration on lists. Note that not all LEDA supported compilers allow the usage of this feature.

```cpp
using namespace leda;
using std::cin;
using std::cout;
using std::endl;

list<int> L;
// fill list somehow
list<int>::iterator it;
for ( it = L.begin(); it != L.end(); it++ )
  cout << *it << endl;
```
2.6. \textit{ITERATION}

\texttt{list<int>::iterator} defines the iterator type, \texttt{begin()} delivers access to the first list item via an iterator. \texttt{end()} is the past the end iterator and serves as an end marker. The increment operator \texttt{++} moves the iterator one position to the next item, and \texttt{*it} delivers the content of the item to which the iterator is pointing. For more information on STL please refer to the standard literature about STL.

For a more flexible access to the LEDA graph data type there are graph iterators which extent the STL paradigm to more complex container types. To make use of these features please refer to 14.
Chapter 3

Modules

During the last years, LEDA’s main include directory has grown to more than 400 include files. As a result, the include directory was simply too complex so that new features were hard to identify. We therefore introduced modules to better organize LEDA’s include structure. Starting from version 5.0 LEDA consists of the several modules:

- **core** (LEDA/incl/core/)
  Module core stores all basic data types (array, list, set, partition, etc.), all dictionary types (dictionary, d_array, h_array sortseq, etc.), all priority queues, and basic algorithms like sorting.

- **numbers** (LEDA/incl/numbers/)
  Module numbers stores all LEDA number types (integer, real, rational, bigfloat, polynomial, etc.) as well as data types related to linear algebra (vector, matrix, etc.) and all additional data types and functions related to numerical computation (fpu, numerical analysis, etc.)

- **graph** (LEDA/incl/graph/)
  Module graph stores all graph data types, all types related to graphs and all graph algorithms.

- **geo** (LEDA/incl/geo/)
  Module geo stores all geometric data types and all geometric algorithms.

- **graphics** (LEDA/incl/graphics/)
  Module graphics stores all include files and data types related to our graphical user interfaces, i.e. window, graphwin and geowin.

- **coding** (LEDA/incl/coding/)
  Module codings contains all data types and algorithms relating to compression and cryptography.

- **system** (LEDA/incl/system/)
  Module system contains all data types that offer system-related functionality like date, time, stream, error handling and memory management.
• *internal* (LEDA/incl/internal/)
  Module internal contains include files that are needed for LEDA’s maintenance or for people who want to implement extension packages.

• *beta* (LEDA/incl/beta/)
  Module beta contains data types that are not fully tested.

We will still maintain the old include structure for a fixed period of time. Please note that we will however turn off the old include structure in one of the next releases. The old include structure is available under $(LEDAROOT)/incl_old/LEDA/$.
Chapter 4

Simple Data Types and Basic Support Operations

This section describes simple data types like strings, streams and gives some information about error handling, memory management and file system access. The stream data types described in this section are all derived from the C++ stream types istream and ostream. They can be used in any program that includes the <LEDA/stream.h> header file. Some of these types may be obsolete in combination with the latest versions of the standard C++ I/O library.

4.1 Strings (string)

1. Definition

An instance \(s\) of the data type \(string\) is a sequence of characters (type \(char\)). The number of characters in the sequence is called the length of \(s\). A string of length zero is called the empty string. Strings can be used wherever a C++ \(const\ char^{*}\) string can be used.

Strings differ from the C++ type \(char^{*}\) in several aspects: parameter passing by value and assignment works properly (i.e., the value is passed or assigned and not a pointer to the value) and strings offer many additional operations.

\#include <LEDA/core/string.h>

2. Types

\(string::size\_type\) the size type.

3. Creation

\(string\ s;\) introduces a variable \(s\) of type \(string\). \(s\) is initialized with the empty string.
string s(const char *p); introduces a variable s of type string. s is initialized with a copy of the C++ string p.

string s(char c); introduces a variable s of type string. s is initialized with the one-character string “c”.

string s(const char *format, ...); introduces a variable s of type string. s is initialized with the string produced by printf(format, ...).

4. Operations

int s.length() returns the length of string s.

bool s.empty() returns whether s is the empty string.

char& s[int i] returns the character at position i.
Precondition: 0 ≤ i ≤ s.length()-1.

string s(int i, int j) returns the substring of s starting at position max(0, i) and ending at position min(j, s.length()-1).
If min(j, s.length()-1) < max(0, i) then the empty string is returned.

string s.head(int i) returns the first i characters of s.

string s.tail(int i) returns the last i characters of s.

int s.pos(string s1, int i) returns the minimum j such that j ≥ i and s1 is a substring of s starting at position j (returns -1 if no such j exists).

int s.pos(const string& s1) returns pos(s1, 0).

bool s.contains(const string& s1) true iff s1 is a substring of s.

string s.insert(int i, string s1) returns s(0, i - 1) + s1 + s(i, s.length()-1).

string s.replace(const string& s1, const string& s2, int i = 1)
returns the string created from s by replacing the i-th occurrence of s1 in s by s2.
Remark: The occurrences of s1 in s are counted in a non-overlapping manner, for instance the string sasas contains only one occurrence of the string sas.
4.1. STRINGS (STRING)

```cpp
string s.replace(int i, int j, const string &s1)
returns the string created from s by replacing s(i, j) by s1.
Precondition: i ≤ j.
```

```cpp
string s.replace(int i, const string &s1)
returns the string created from s by replacing s[i] by s1.
```

```cpp
string s.replace_all(const string &s1, const string &s2)
returns the string created from s by replacing all occurrences of s1 in s by s2.
Precondition: The occurrences of s1 in s do not overlap (it’s hard to say what the function returns if the precondition is violated.).
```

```cpp
string s.del(const string &s1, int i = 1)
returns s.replace(s1, "", i).
```

```cpp
string s.del(int i, int j)
returns s.replace(i, j, "").
```

```cpp
string s.del(int i)
returns s.replace(i, "").
```

```cpp
string s.del_all(const string &s1)
returns s.replace_all(s1, "").
```

```cpp
void s.read(istream &I, char delim = ' ')
reads characters from input stream I into s until the first occurrence of character delim. (If delim is \n it is extracted from the stream, otherwise it remains there.)
```

```cpp
void s.read(char delim = ' ')
same as s.read(cin, delim).
```

```cpp
void s.read_line(istream &I)
same as s.read(I, \n).
```

```cpp
void s.read_line()
same as s.read_line(cin).
```

```cpp
void s.read_file(istream &I)
same as s.read(I, 'EOF').
```

```cpp
void s.read_file()
same as s.read_file(cin).
```

```cpp
string & s += const string & x
appends x to s and returns a reference to s.
```

```cpp
string const string & x + const string & y
returns the concatenation of x and y.
```

```cpp
bool const string & x == const string & y
true iff x and y are equal.
```
bool const string& x != const string& y
true iff x and y are not equal.

bool const string& x < const string& y
true iff x is lexicographically smaller than y.

bool const string& x > const string& y
true iff x is lexicographically greater than y.

bool const string& x ≤ const string& y
returns (x < y) \| (x == y).

bool const string& x ≥ const string& y
returns (x > y) \| (x == y).

istream& istream& I >> string& s
same as s.read(I, ‘ ’).

ostream& ostream& O << const string& s
writes string s to the output stream O.

5. Implementation

Strings are implemented by C++ character vectors. All operations involving the search for a pattern s1 in a string s take time $O(s.length() \ast s1.length())$, [] takes constant time and all other operations on a string s take time $O(s.length())$. 
4.2 File Input Streams (file_istream)

1. Definition

The data type file_istream is equivalent to the ifstream type of C++.

#include <LEDA/system/stream.h>

4.3 File Output Streams (file_ostream)

1. Definition

The data type file_ostream is equivalent to the ofstream type of C++.

#include <LEDA/system/stream.h>

4.4 String Input Streams (string_istream)

1. Definition

An instance \( I \) of the data type string_istream is an C++istream connected to a string \( s \), i.e., all input operations or operators applied to \( I \) read from \( s \).

#include <LEDA/system/stream.h>

2. Creation

\[
\text{string_istream } I(\text{const char } * s);
\]

creates an instance \( I \) of type string_istream connected to the string \( s \).

3. Operations

All operations and operators (\( >> \)) defined for C++istreams can be applied to string input streams as well.

4.5 String Output Streams (string_ostream)

1. Definition

An instance \( O \) of the data type string_ostream is an C++ostream connected to an internal
string buffer, i.e., all output operations or operators applied to \( O \) write into this internal buffer. The current value of the buffer is called the contents of \( O \).

```c
#include <LEDA/system/stream.h>
```

2. Creation

```c
string ostream O;  // creates an instance \( O \) of type string ostream.
```

3. Operations

```c
string O.str();  // returns the current contents of \( O \).
```

All operations and operators (<<) defined for C++ streams can be applied to string output streams as well.
4.6 Random Sources (random_source)

1. Definition

An instance of type random_source is a random source. It allows to generate uniformly distributed random bits, characters, integers, and doubles. It can be in either of two modes: In bit mode it generates a random bit string of some given length $p$ ($1 \leq p \leq 31$) and in integer mode it generates a random integer in some given range $[\text{low}..\text{high}]$ ($\text{low} \leq \text{high} < \text{low} + 2^{31}$). The mode can be changed any time, either globally or for a single operation. The output of the random source can be converted to a number of formats (using standard conversions).

```c
#include <LEDA/core/random_source.h>
```

2. Creation

```c
random_source S; // creates an instance S of type random_source, puts it into bit mode, and sets the precision to 31.
random_source S(int p); // creates an instance S of type random_source, puts it into bit mode, and sets the precision to $p$ ($1 \leq p \leq 31$).
random_source S(int low, int high); // creates an instance S of type random_source, puts it into integer mode, and sets the range to $[\text{low}..\text{high}]$. 
```

3. Operations

```c
unsigned long S.get() // returns a random unsigned integer of maximal precision (32 bits on 32-bit systems and 64 bits on 64-bit systems).
void S.set_seed(int s) // resets the seed of the random number generator to $s$.
int S.reinit_seed() // generates and sets a new seed $s$. The return value is $s$.
void S.set_range(int low, int high) // sets the mode to integer mode and changes the range to $[\text{low}..\text{high}]$.
int S.set_precision(int p) // sets the mode to bit mode, changes the precision to $p$ bits and returns previous precision.
int S.get_precision() // returns current precision of S.
```
random_source& S >> char& x extracts a character $x$ of default precision or range and returns $S$, i.e., it first generates an unsigned integer of the desired precision or in the desired range and then converts it to a character (by standard conversion).

random_source& S >> unsigned char& x extracts an unsigned character $x$ of default precision or range and returns $S$.

random_source& S >> int& x extracts an integer $x$ of default precision or range and returns $S$.

random_source& S >> long& x extracts a long integer $x$ of default precision or range and returns $S$.

random_source& S >> unsigned int& x extracts an unsigned integer $x$ of default precision or range and returns $S$.

random_source& S >> unsigned long& x extracts a long unsigned integer $x$ of default precision or range and returns $S$.

random_source& S >> double& x extracts a double precision floating point number $x$ in $[0, 1]$, i.e., $u/(2^{31} - 1)$ where $u$ is a random integer in $[0..2^{31} - 1]$, and returns $S$.

random_source& S >> float& x extracts a single precision floating point number $x$ in $[0, 1]$, i.e., $u/(2^{31} - 1)$ where $u$ is a random integer in $[0..2^{31} - 1]$, and returns $S$.

random_source& S >> bool& b extracts a random boolean value (true or false).

int S( ) returns an integer of default precision or range.

int S(int prec) returns an integer of supplied precision $prec$.

int S(int low, int high) returns an integer from the supplied range $[low..high]$. 
4.7 Random Variates (random_variate)

1. Definition

An instance $R$ of the data type `random_variate` is a non-uniform random number generator. The generation process is governed by an `array<int>` $w$. Let $[l..r]$ be the index range of $w$ and let $W = \sum_i w[i]$ be the total weight. Then any integer $i \in [l..h]$ is generated with probability $w[i]/W$. The weight function $w$ must be non-negative and $W$ must be non-zero.

```cpp
#include <LEDA/core/random_variate.h>
```

2. Creation

`random_variate` $R$(`const array<int>& w`);

creates an instance $R$ of type `random_variate`.

3. Operations

```cpp
int R.generate() generates $i \in [l..h]$ with probability $w[i]/W$.
```

4.8 Dynamic Random Variates (dynamic_random_variate)

1. Definition

An instance $R$ of the data type `dynamic_random_variate` is a non-uniform random number generator. The generation process is governed by an `array<int>` $w$. Let $[l..r]$ be the index range of $w$ and let $W = \sum_i w[i]$ be the total weight. Then any integer $i \in [l..h]$ is generated with probability $w[i]/W$. The weight function $w$ must be non-negative and $W$ must be non-zero. The weight function can be changed dynamically.

```cpp
#include <LEDA/core/random_variate.h>
```

2. Creation

`dynamic_random_variate` $R$(`const array<int>& w`);

creates an instance $R$ of type `dynamic_random_variate`.

3. Operations

```cpp
int R.generate() generates $i \in [l..h]$ with probability $w[i]/W$.
```
\textbf{int} \quad \texttt{R.set\_weight}(\textit{int} \; i, \; \textit{int} \; g)

sets $w[i]$ to $g$ and returns the old value of $w[i]$.  
\textit{Precondition:} $i \in [l..h]$. 
4.9 Memory Management

LEDA offers an efficient memory management system that is used internally for all node, edge and item types. This system can easily be customized for user defined classes by the “LEDA_MEMORY” macro. You simply have to add the macro call “LEDA_MEMORY(T)” to the declaration of a class T. This redefines new and delete operators for type T, such that they allocate and deallocate memory using LEDA’s internal memory manager.

```cpp
struct pair {
    double x;
    double y;

    pair() { x = y = 0; }
    pair(const pair& p) { x = p.x; y = p.y; }

    friend ostream& operator<<(ostream&, const pair&); {
    friend istream& operator>>(istream&, pair&); {
    friend int compare(const pair& p, const pair& q); {

    LEDA_MEMORY(pair)

};

dictionary<pair,int> D;
```

The LEDA memory manager only frees memory at its time of destruction (program end or unload of library) as this allows for much faster memory allocation requests. As a result, memory that was deallocated by a call to the redefined delete operator still resides in the LEDA memory management system and is not returned to the system memory manager. This might lead to memory shortages. To avoid those shortages, it is possible to return unused memory of LEDA’s memory management system to the system memory manager by calling

```cpp
leda::std_memory_mgr.clear();
```
4.10 Memory Allocator (leda_allocator)

1. Definition

An instance $A$ of the data type $\text{leda_allocator}<T>$ is a memory allocator according to the C++ standard. $\text{leda_allocator}<T>$ is the standard compliant interface to the LEDA memory management.

```c
#include <LEDA/system/allocator.h>
```

2. Types

Local types are $\text{size_type}$, $\text{difference_type}$, $\text{value_type}$, $\text{pointer}$, $\text{reference}$, $\text{const_pointer}$, and $\text{const_reference}$.

```c
template <class T1>
\text{leda_allocator}<T>::\text{rebind} \quad \text{allows the construction of a derived allocator:}
\text{leda_allocator}<T>::\text{template rebind}<T1>::\text{other}
\text{is the type } \text{leda_allocator}<T1>.
```

3. Creation

$\text{leda_allocator}<T> \ A; \quad \text{introduces a variable } A \text{ of type } \text{leda_allocator}<T>$.

4. Operations

```c
\text{pointer} \quad A.\text{allocate}(\text{size_type } n, \text{const_pointer } = 0)
\quad \text{returns a pointer to a newly allocated memory range of size } n \ast \text{sizeof}(T).

\text{void} \quad A.\text{deallocate}(\text{pointer } p, \text{size_type } n)
\quad \text{deallocates a memory range of } n \ast \text{sizeof}(T) \text{ starting at } p. \quad \text{Precondition: the memory range was obtained via }\text{allocate}(n).

\text{pointer} \quad A.\text{address}(\text{reference } r)
\quad \text{returns } &r.

\text{const_pointer} \quad A.\text{address}(\text{const_reference } r)
\quad \text{returns } &r.

\text{void} \quad A.\text{construct}(\text{pointer } p, \text{const_reference } r)
\quad \text{makes an inplace new } \text{new}((\text{void}*)p) \ T(r).

\text{void} \quad A.\text{destroy}(\text{pointer } p)
\quad \text{destroys the object referenced via } p \text{ by calling } p \rightarrow ~T( ).
```
size_type A.max_size() the largest value \( n \) for which the call allocate\((n, 0)\) might succeed.

5. Implementation

Note that the above class template uses all kinds of modern compiler technology like member templates, partial specialization etc. It runs only on a subset of LEDA’s general supported platforms like \texttt{g++ \textgreater 2.95}, \texttt{SGI CC \textgreater 7.3}. 
4.11 Error Handling (error)

LEDA tests the preconditions of many (not all!) operations. Preconditions are never tested, if the test takes more than constant time. If the test of a precondition fails an error handling routine is called. It takes an integer error number $i$ and a char* error message string $s$ as arguments. The default error handler writes $s$ to the diagnostic output ($\text{cerr}$) and terminates the program abnormally if $i \neq 0$. Users can provide their own error handling function $\text{handler}$ by calling

\[
\text{set_error_handler}(\text{handler})
\]

After this function call $\text{handler}$ is used instead of the default error handler. $\text{handler}$ must be a function of type $\text{void handler}(\text{int}, \text{const char}*)$. The parameters are replaced by the error number and the error message respectively.

New:
Starting with version 4.3 LEDA provides an exception error handler

\[
\text{void exception_error_handler}(\text{int num}, \text{const char} * \text{msg})
\]

This handler uses the C++exception mechanism and throws an exception of type $\text{leda_exception}$ instead of terminating the program. An object of type $\text{leda_exception}$ stores a pair consisting of an error number and an error message. Operations $\text{e.get_msg()}$ and $\text{e.get_num()}$ can be called to retrieve the corresponding values from an exception object $e$.

1. Operations

\[
\text{#include } < \text{LEDA/system/error.h} >
\]

\[
\text{void error_handler(} \text{int err_no, const char} * \text{msg})
\]

reports error messages by passing $\text{err_no}$ and $\text{msg}$ to the default error handler.

\[
\text{LedaErrorHandler set_error_handler(} \text{void (} *\text{err_handler})(\text{int, const char}*))
\]

sets the default error handler to function $\text{err_handler}$. Returns a pointer to the previous error handler.

\[
\text{LedaErrorHandler get_error_handler( )}
\]

returns a pointer to the current default error handler.

\[
\text{void catch_system_errors(} \text{bool b = true})
\]

after a call to this function system errors (e.g. bus errors and segmentation faults) are handled by LEDA’s error handler.
bool leda_assert(bool cond, const char *errmsg, int err_no = 0)
calls error_handler(err_no, errmsg) if cond = false and returns cond.
4.12 Files and Directories (file)

The following functions are declared in `<LEDA/file.h>`.

- `string set_directory(string new_dir)`
  sets the current working directory to `new_dir` and returns the name of the old cwd.
- `string get_directory()` returns the name of the current working directory.
- `string get_home_directory()` returns the name of the user’s home directory.
- `string get_directory_delimiter()` returns the character that delimits directory names in a path (i.e. “\” on Windows and “/” on Unix).
- `void append_directory_delimiter(string& dir)` appends the directory delimiter to `dir` if `dir` does not already end with the delimiter.
- `void remove_trailing_directory_delimiter(string& dir)` removes the directory delimiter from `dir` if `dir` ends with it.
- `list<string> get_directories(string dir)` returns the list of names of all sub-directories in directory `dir`.
- `list<string> get_files(string dir)` returns the list of names of all regular files in directory `dir`.
- `list<string> get_files(string dir, string pattern)` returns the list of names of all regular files in directory `dir` matching pattern.
- `list<string> get_entries(string dir)` returns the list of all entries (directory and files) of directory `dir`.
- `bool create_directory(string fname)` creates a directory with name `fname`, returns true on success.
- `bool is_directory(string fname)` returns true if `fname` is the path name of a directory and false otherwise.
- `bool is_file(string fname)` returns true if `fname` is the path name of a regular file and false otherwise.
bool is_link(string fname) returns true if fname is the path name of a symbolic link and false otherwise.

int size_of_file(string fname) returns the size of file fname in bytes.

string tmp_file_name() returns a unique name for a temporary file.

bool delete_file(string fname) deletes file fname returns true on success and false otherwise.

string first_file_in_path(string fname, string path, char sep = ' : ') searches all directories in string path (separated by sep) for the first directory dir that contains a file with name fname and returns dir/fname (the empty string if no such directory is contained in path).
4.13 Sockets (leda_socket)

1. Definition

A data packet consists of a sequence of bytes (in C of type unsigned char) $c_0, c_1, c_2, c_3, x_1, \ldots, x_n$. The first four bytes encode the number $n$ of the following bytes such that $n = c_0 + c_1 \cdot 2^8 + c_2 \cdot 2^{16} + c_3 \cdot 2^{24}$. The LEDA data type leda_socket offers, in addition to the operations for establishing a socket connection, functions for sending and receiving packets across such a connection. It is also possible to set a receive limit; if such a receive limit is set, messages longer than the limit will be refused. If the limit is negative (default), no messages will be refused.

In particular, the following operations are available:

#include <LEDA/system/socket.h>

2. Creation

leda_socket $S$(const char *host, int port);

creates an instance $S$ of type leda_socket associated with host name host and port number port.

leda_socket $S$(const char *host);

creates an instance $S$ of type leda_socket associated with host name host.

leda_socket $S$;

creates an instance $S$ of type leda_socket.

3. Operations

void $S$.set_host(const char *host)

sets the host name to host.

void $S$.set_port(int port)

sets the port number to port.

int $S$.get_limit()

returns the receive limit parameter.

void $S$.set_limit(int limit)

sets the receive limit parameter to limit. If a negative limit is set, the limit parameter will be ignored.

void $S$.set_qlength(int len)

sets the queue length to len.

void $S$.set_timeout(int sec)

sets the timeout interval to sec seconds.

void $S$.set_LEDA_EXCEPTION(void (*)(leda_socket&, string))

sets the error handler to function $f$. 
4.13. SOCKETS (LEDA_SOCKET)

void $S$.set_receive_handler(void (*f)(leda_socket&, int, int))
sets the receive handler to function $f$.

void $S$.set_send_handler(void (*f)(leda_socket&, int, int))
sets the send handler to function $f$.

string $S$.get_host() returns the host name.

int $S$.get_port() returns the port number.

int $S$.get_timeout() returns the timeout interval length in seconds.

int $S$.get qlength() returns the queue length.

string $S$.get_ip() returns the ip address.

bool $S$.connect(int sec = 10)
tries to establish a connection from a client to a server. If the connection can be established within $sec$ seconds, the operation returns true and false otherwise.

bool $S$.listen() creates a socket endpoint on the server, performs address binding and signals readiness of a server to receive data.

bool $S$.accept() the server takes a request from the queue.

void $S$.disconnect() ends a connection.

Sending and receiving packets

void $S$.send_file(string fname)
sends the contents of file $fname$.

void $S$.send_bytes(char *buf, int numb)
sends $numb$ bytes starting at address $buf$.

void $S$.send_string(string msg)
sends string $msg$.

void $S$.send_int(int x) sends (a text representation of) integer $x$.

bool $S$.receive_file(string fname)
receives data and writes it to file $fname$.

char* $S$.receive_bytes(int& numb)
receives $numb$ bytes. The function allocates memory and returns the first address of the allocated memory. $numb$ is used as the return parameter for the number of received bytes.
int \quad S.receive\_bytes(\textit{char} \ast \textit{buf}, \textit{int} \textit{buf}\_sz)

receives at most \textit{buf}\_sz bytes and writes them into the buffer \textit{buf}. It returns the number of bytes supplied by the sender (maybe more than \textit{buf}\_sz), or -1 in case of an error.

bool \quad S.receive\_string(\textit{string}\& \textit{s})

receives string \textit{s}.

bool \quad S.receive\_int(\textit{int}\& \textit{x})

receives (a text representation of) an integer and stores its value in \textit{x}.

bool \quad S.wait(\textit{string} \textit{s})

returns \textit{true}, if \textit{s} is received, \textit{false} otherwise.

The following template functions can be used to send/receive objects supporting input and output operators for iostreams.

\textbf{template} <\textit{class} \textit{T}>

\textbf{void} \quad \textit{socket\_send\_object}(\textit{const} \textit{T}\& \textit{obj}, \textit{leda\_socket}\& \textit{sock})

sends \textit{obj} to the connection partner of \textit{sock}.

\textbf{template} <\textit{class} \textit{T}>

\textbf{void} \quad \textit{socket\_receive\_object}(\textit{T}\& \textit{obj}, \textit{leda\_socket}\& \textit{sock})

receives \textit{obj} from the connection partner of \textit{sock}.
4.14 Socket Streambuffer (socket_streambuf)

1. Definition

An instance $sb$ of class `socket_streambuf` can be used as an adapter: It turns a `leda_socket` $s$ into a C++ streambuf object. This object can be used in standard C++ `ostreams` and `istreams` to make the communication through the socket easier. A `socket_streambuf` can also be applied within an `encoding_ostream` or a `decoding_istream` provided by LEDA to send compressed or encrypted data over a socket connection (see Sections 10.19 and 10.20). Observe that `socket_streambuf` itself does not apply cryptography to secure the transmitted data. All data is sent as it is. If you want to secure your data, consider using the class `secure_socket_streambuf` (see Section 11.7).

If two parties want to use the class `socket_streambuf` to exchange data they have to do the following. First they establish a connection through `leda_sockets`. Then each party constructs an instance of the class `socket_streambuf` which is attached to its socket. This is shown in an example at the end of this manual page.

Every instance $sb$ has an out-buffer where outgoing data is buffered before it is sent through the socket over the internet. In addition it has an in-buffer to store data that has been received through the socket. The user may specify the maximum size of each buffer. The actual size of each buffer is determined in a negotiation between the server and the client at the beginning of the communication. The size of outgoing packets from the server is set to the minimum of the out-buffer size of the server and the in-buffer size of the client. The size of the incoming packets is determined in an analogous way.

```c
#include <LEDA/coding/socket_streambuf.h>
```

2. Creation

```c
socket_streambuf sb(leda_socket& s, uint32 out_buf_sz = DefaultBufferSize,
                    uint32 in_buf_sz = DefaultBufferSize,
                    bool send_acknowledge = false);
```

creates a $sb$ and attaches it to the socket $s$. The parameters `out_buf_sz` and `in_buf_sz` determine the maximum size of the out-buffer and the in-buffer. `send_acknowledge` specifies whether an acknowledgement is sent for every received packet.

**Precondition:** The connection between the server and the client must have been established when $sb$ is created.

3. Operations

The class `socket_streambuf` inherits most of its operations from the class `streambuf` that belongs to the C++ standard library. Usually there is no need to call these operations explicitly. (You can find documentation for `streambuf` at http://www.cplusplus.com)
bool sb.failed( )
returns whether a failure has occurred.

string sb.get_error( )
returns an error message (if available).

void sb.sputEOF( )
signals the end of the transmission to the receiving socket, so that it does not wait for further data. (This function is called automatically in the destructor unless it has been called explicitly by the user. If sb is not immediately destroyed after the end of the transmission then you should call sputEOF explicitly, otherwise the receiving party might incur a timeout error.)

bool sb.has_put_EOF( )
returns whether EOF has already been sent.

bool sb.has_get_EOF( )
returns whether EOF has already been received.

leda_socket& sb.get_socket( )
returns the socket to which sb is attached.

uint32 sb.get_outgoing_packet_size( )
returns the (actual) outgoing packet size.

uint32 sb.get_incoming_packet_size( )
returns the (actual) incoming packet size.

bool sb.waits_for_acknowledge( )
returns whether sb expects an acknowledgement for outgoing packets.

bool sb.sends_acknowledge( )
returns whether sb sends an acknowledgement for incoming packets.

4. Example

The following example shows how the usage of sb from the server and from the client side.
In our example the server sends a string, which is received by the client. (Note that it would also be possible that the client sends and the server receives data.)
In order to add compression to the example simply replace “ostream” by “encoding_ostream<Coder>” and “istream” by “decoding_istream” as indicated in the comments.

void socket_example_server(int port)
{
    leda_socket sock;
    sock.set_port(port);

    // open port
if (! sock.listen()) {
    cerr << sock.get_error() << endl; return;
}
for (; ;) {
    // establish connection
    if (! sock.accept()) {
        cerr << sock.get_error() << endl; continue;
    }

    // send data
    {
        socket_streambuf sb(sock);
        ostream out(&sb);
        // or: encoding_ostream<PPMIICoder> out(&sb);
        out << "Hello world!" << endl;
    } // destroys sb and calls sb.sputEOF() automatically
}

void socket_example_client(int port, string host)
{
    leda_socket sock;
    sock.set_host(host);
    sock.set_port(port);

    // establish connection
    if (! sock.connect()) {
        cerr << sock.get_error() << endl; return;
    }

    // receive data
    socket_streambuf sb(sock);
    istream in(&sb);
    // or: decoding_istream<PPMIICoder> in(&sb);
    string str;
    str.read_line(in);
    cout << "received: " << str << endl;
}
4.15 Some Useful Functions (misc)

The following functions and macros are defined in `<LEDA/system/basic.h>`.

- **int** `read_int(string s)` prints `s` and reads an integer from `cin`.
- **double** `read_real(string s)` prints `s` and reads a real number from `cin`.
- **string** `read_string(string s)` prints `s` and reads a line from `cin`.
- **char** `read_char(string s)` prints `s` and reads a character from `cin`.
- **int** `Yes(string s)` returns `(read_char(s) == 'y')`.
- **bool** `get_environment(string var)` returns `true` if variable `var` is defined in the current environment and `false` otherwise.
- **bool** `get_environment(string var, string& val)` if variable `var` is defined in the current environment its value is assigned to `val` and the result is `true`. Otherwise, the result is `false`.
- **float** `used_time()` returns the currently used cpu time in seconds. (The class `timer` in Section 4.16 provides a nicer interface for time measurements.)
- **float** `used_time(float& T)` returns the cpu time used by the program from time `T` up to this moment and assigns the current time to `T`.
- **float** `elapsed_time()` returns the current daytime time in seconds.
- **float** `elapsed_time(float& T)` returns the elapsed time since time `T` and assigns the current elapsed time to `T`.
- **void** `print_statistics()` prints a summary of the currently used memory, which is used by LEDA’s internal memory manager. This only reports on memory usage of LEDA’s internal types and user-defined types that implement the LEDA_MEMORY macro (see Section 4.9).
- **bool** `is_space(char c)` returns `true` if `c` is a white space character.
- **void** `wait(double sec)` suspends execution for `sec` seconds.
double truncate(double x, int k = 10)

returns a double whose mantissa is truncated after \( k - 1 \) bits after the binary point, i.e., if \( x \neq 0 \) then
the binary representation of the mantissa of the result
has the form \( d.ddddddd \), where the number of \( d \)'s
is equal to \( k \). There is a corresponding function for
integers; it has no effect.

template <class T>

const T& min(const T& a, const T& b)

returns the minimum of \( a \) and \( b \).

template <class T>

const T& max(const T& a, const T& b)

returns the maximum of \( a \) and \( b \).

template <class T>

void swap(T& a, T& b)

swaps values of \( a \) and \( b \).
4.16 Timer (timer)

1. Definition

The class timer facilitates time measurements. An instance \( t \) has two states: *running* or *stopped*. It measures the time which elapses while it is in the state *running*. The state depends on a (non-negative) internal counter, which is incremented by every *start* operation and decremented by every *stop* operation. The timer is *running* iff the counter is not zero. The use of a counter (instead of a boolean flag) to determine the state is helpful when a recursive function \( f \) is measured, which is shown in the example below:

```c
#include <LEDA/system/timer.h>
leda::timer f_timer;

void f()
{
  f_timer.start();

  // do something ...
  f(); // recursive call
  // do something else ...

  f_timer.stop(); // timer is stopped when top-level call returns
}

int main()
{
  f();
  std::cout << "time spent in f " << f_timer << "\n"; return 0;
}
```

Let us analyze this example. When \( f \) is called in main, the timer is in the state *stopped*. The first *start* operation (in the top-level call) increments the counter from zero to one and puts the timer into the state *running*. In a recursive call the counter is incremented at the beginning and decremented upon termination, but the timer remains in the state *running*. Only when the top-level call of \( f \) terminates and returns to main, the counter is decremented from one to zero, which puts the timer into the state *stopped*. So the timer measures the total running time of \( f \) (including recursive calls).

```c
#include <LEDA/system/timer.h>
```

2. Types

\texttt{timer::measure} auxiliary class to facilitate measurements (see example below).
3. Creation

\texttt{timer \ t(const string& name, bool report\_on\_destruction = true);};

creates an instance \( t \) with the given \texttt{name}. If \texttt{report\_on\_destruction} is true, then the timer reports upon its destruction how long it has been running in total. The initial state of the timer is \textit{stopped}.

\texttt{timer \ t;} creates an unnamed instance \( t \) and sets the \texttt{report\_on\_destruction} flag to false. The initial state of the timer is \textit{stopped}.

4. Operations

\texttt{void \ t.reset( \ )} sets the internal counter and the total elapsed time to zero.

\texttt{void \ t.start( \ )} increments the internal counter.

\texttt{void \ t.stop( \ )} decrements the internal counter. (If the counter is already zero, nothing happens.)

\texttt{void \ t.restart( \ )} short-hand for \( t.reset(\ ) + t.start(\ ) \).

\texttt{void \ t.halt( \ )} sets the counter to zero, which forces the timer into the state \textit{stopped} no matter how many \textit{start} operations have been executed before.

\texttt{bool \ t.is\_running( \ )} returns if \( t \) is currently in the state \textit{running}.

\texttt{float \ t.elapsed\_time( \ )} returns how long (in seconds) \( t \) has been in the state \textit{running} (since the last \textit{reset}).

\texttt{void \ t.set\_name(const string& name)}

sets the name of \( t \).

\texttt{string \ t.get\_name( \ )} returns the name of \( t \).

\texttt{void \ t.report\_on\_destruction(bool do\_report = true)}

sets the flag \texttt{report\_on\_destruction} to \texttt{do\_report}.

\texttt{bool \ t.will\_report\_on\_destruction( \ )} returns whether \( t \) will issue a report upon its destruction.

5. Example

We give an example demonstrating the use of the class \textit{measure}. Note that the function below has several \texttt{return} statements, so it would be tedious to stop the timer “by hand”.

\texttt{#include <LEDA/system/timer.h>
unsigned fibonacci(unsigned n)
{
  static leda::timer t("fibonacci");
  // report total time upon destruction of t

  leda::timer::measure m(t);
  // starts the timer t when m is constructed, and stops t
  // when m is destroyed, i.e. when the function returns

  if (n < 1) return 0;
  else if (n == 1) return 1;
  else return fibonacci(n-1) + fibonacci(n-2);
}

int main()
{
  std::cout << fibonacci(40) << "\n";
  return 0; // reports "Timer(fibonacci): X.XX s" upon termination
}
4.17 Counter (counter)

1. Definition

The class counter can be used during profiling to count how often certain code is executed. An example is given below.

```c
#include <LEDA/system/counter.h>
```

2. Creation

```c
counter c(const string& name, bool report_on_destruction = true);
```

creates an instance c with the given name. If report_on_destruction is true, then the counter reports its value upon destruction. The initial value of the counter is zero.

```c
counter c;
```

creates an unnamed instance c and sets the report_on_destruction flag to false. The initial value of the counter is zero.

3. Operations

```c
void c.reset() sets the value of c to zero.
void c.set_value(const unsigned long val) sets the value of c to val.
const unsigned long c.get_value() returns the current value of c.
const unsigned long c.increment() increments c and returns its new value. (We also provide the operator ++.)
void c.set_name(const string& name) sets the name of c.
string c.get_name() returns the name of c.
void c.report_on_destruction(bool do_report = true) sets the flag report_on_destruction to do_report.
bool c.will_report_on_destruction() returns whether c will issue a report upon its destruction.
```

4. Example

In the example below we count how often the function fibonacci is executed.

```c
#include <LEDA/system/counter.h>
```
unsigned fibonacci(unsigned n)
{
    static leda::counter cnt("fibonacci");
    // report upon destruction of cnt
    ++cnt;

    if (n < 1) return 0;
    else if (n == 1) return 1;
    else return fibonacci(n-1) + fibonacci(n-2);
}

int main()
{
    std::cout << fibonacci(40) << "\n";
    return 0; // reports "Counter(fibonacci) = 331160281" upon termination
}
4.18 Two Tuples (two_tuple)

1. Definition

An instance $p$ of type $\text{two_tuple}<A,B>$ is a two-tuple $(a,b)$ of variables of types $A$, and $B$, respectively.

Related types are $\text{two_tuple}$, $\text{three_tuple}$, and $\text{four_tuple}$.

```cpp
#include <LEDA/core/tuple.h>
```

2. Types

$\text{two_tuple}<A,B>::\text{first_type}$ the type of the first component.

$\text{two_tuple}<A,B>::\text{second_type}$

the type of the second component.

3. Creation

$\text{two_tuple}<A,B>\ p; \quad$ creates an instance $p$ of type $\text{two_tuple}<A,B>$. All components are initialized to their default value.

$\text{two_tuple}<A,B>\ p(\text{const} \ A\& \ u, \text{const} \ B\& \ v);$
creates an instance $p$ of type $\text{two_tuple}<A,B>$ and initializes it with the value $(u,v)$.

4. Operations

$A\& \quad p.\text{first}() \quad$ returns the $A$-component of $p$. If $p$ is a const-object the return type is $A$.

$B\& \quad p.\text{second}() \quad$ returns the $B$-component of $p$. If $p$ is a const-object the return type is $B$.

```cpp
template <class A, class B>
bool const two_tuple<A,B>& p == const two_tuple<A,B>& q
```
equality test for $\text{two_tuples}$. Each of the component types must have an equality operator.

```cpp
template <class A, class B>
int compare(const two_tuple<A,B>& p, const two_tuple<A,B>& q)
```
lexicographic ordering for $\text{two_tuples}$. Each of the component types must have a compare function.

```cpp
template <class A, class B>
```
CHAPTER 4. SIMPLE DATA TYPES AND BASIC SUPPORT OPERATIONS

int Hash(const two_tuple<A, B>& p)

hash function for two_tuples. Each of the component
types must have a Hash function.

5. Implementation

The obvious implementation is used.

4.19 Three Tuples ( three_tuple )

1. Definition

An instance \( p \) of type \( \text{three_tuple}<A, B, C> \) is a three-tuple \( (a, b, c) \) of variables of types \( A, B, \) and \( C, \) respectively.

Related types are \( \text{two_tuple}, \text{three_tuple}, \) and \( \text{four_tuple}. \)

```
#include <LEDA/core/tuple.h>
```

2. Types

\( \text{three_tuple}<A, B, C>::\text{first}_\text{type} \)
the type of the first component.

\( \text{three_tuple}<A, B, C>::\text{second}_\text{type} \)
the type of the second component.

\( \text{three_tuple}<A, B, C>::\text{third}_\text{type} \)
the type of the third component.

3. Creation

\( \text{three_tuple}<A, B, C> \ p; \) creates an instance \( p \) of type \( \text{three_tuple}<A, B, C> \). All com-
ponents are initialized to their default value.

\( \text{three_tuple}<A, B, C> \ p(\text{const } A& \ u, \text{const } B& \ v, \text{const } C& \ w); \)
creates an instance \( p \) of type \( \text{three_tuple}<A, B, C> \) and initial-
izes it with the value \( (u, v, w) \).

4. Operations

\( A& \ p.\text{first}() \) returns the \( A \)-component of \( p \). If \( p \) is a const-object the
return type is \( A \).

\( B& \ p.\text{second}() \) returns the \( B \)-component of \( p \). If \( p \) is a const-object the
return type is \( B \).
4.20. FOUR TUPLES (FOUR_TUPLE)

1. Definition

An instance \( p \) of type `four_tuple<A, B, C, D>` is a four-tuple \((a, b, c, d)\) of variables of types \( A, B, C, \) and \( D, \) respectively.

Related types are `two_tuple`, `three_tuple`, and `four_tuple`.

```cpp
#include <LEDA/core/tuple.h>
```

2. Types

`four_tuple<A, B, C, D>::first_type`

the type of the first component.

`four_tuple<A, B, C, D>::second_type`

the type of the second component.

`four_tuple<A, B, C, D>::third_type`

the type of the third component.

`four_tuple<A, B, C, D>::fourth_type`

the type of the fourth component.
3. Creation

\( \text{four_tuple}<A, B, C, D \> \ p; \) creates an instance \( p \) of type \( \text{four_tuple}<A, B, C, D \> \). All components are initialized to their default value.

\( \text{four_tuple}<A, B, C, D \> \ p(const \ A \& \ u, const \ B \& \ v, const \ C \& \ w, const \ D \& \ x); \)

creates an instance \( p \) of type \( \text{four_tuple}<A, B, C, D \> \) and initializes it with the value \( (u, v, w, x) \).

4. Operations

\( A \& p.\text{first()} \) returns the \( A \)-component of \( p \). If \( p \) is a const-object the return type is \( A \).

\( B \& p.\text{second()} \) returns the \( B \)-component of \( p \). If \( p \) is a const-object the return type is \( B \).

\( C \& p.\text{third()} \) returns the \( C \)-component of \( p \). If \( p \) is a const-object the return type is \( C \).

\( D \& p.\text{fourth()} \) returns the \( D \)-component of \( p \). If \( p \) is a const-object the return type is \( D \).

**template** \( <\text{class A, class B, class C, class D}> \)

\( \text{bool} \ const \ p = \text{four_tuple}<A, B, C, D \& q \)

equality test for \( \text{four_tupl}es \). Each of the component types must have an equality operator.

**template** \( <\text{class A, class B, class C, class D}> \)

\( \text{int} \ \text{compare}(const \ p = \text{four_tuple}<A, B, C, D \& q) \)

lexicographic ordering for \( \text{four_tupl}es \). Each of the component types must have a compare function.

**template** \( <\text{class A, class B, class C, class D}> \)

\( \text{int} \ \text{Hash}(\text{four_tuple}<A, B, C, D \& p) \)

hash function for \( \text{four_tupl}es \). Each of the component types must have a Hash function.

5. Implementation

The obvious implementation is used.

6. Example

We customize \( \text{four_tupl}es \) and define a \( \text{h_array} \) for them.

\#define prio() first()
\#define inf() second()
#define pq_item() third()
#define part_item() fourth()
typedef four_tuple<int,int,int,int> my_qu;

my_qu q;
my_qu q1(2,2,0,0);
q.prio() = 5;

h_array<my_qu,int> M;
M[my_qu(2,2,nil,nil)] = 5;
CHAPTER 4. SIMPLE DATA TYPES AND BASIC SUPPORT OPERATIONS

4.21 A date interface (date)

1. Definition

An instance of the data type date represents a date consisting of a day \( d \), a month \( m \) and year \( y \). It will be denoted by \( d.m.y \). Valid dates range from 1.1.1 to 31.12.9999. A date is valid if it lies in the range and is correct according to the gregorian calendar, i.e. a year \( y \) is considered to be a leap year iff \( y \) is divisible by 4 but not by 100 or \( y \) is divisible by 400. The year part \( y \) is always a four digit number, so that each date in the valid range has an unambiguous representation.

With the date class there is associated an input and an output format, each is described by a string which determines how instances of type date are read from streams and how they are printed to streams. Printing the date 4.11.1973 using the format string ”dd.mm.yy” will result in ”04.11.73”, whereas printing the same date using ”mm/dd/yyyy” will produce ”11/04/1973”. The date type provides some predefined formats, it also allows user-defined formats and supports different languages (for month names and weekday names). A format string consists of tokens, not all tokens are valid for both input and output formats. But any sequence of valid tokens forms a valid format string, the only exception to this rule is the delim token (see the table below). In order to avoid ambiguities when parsing a format string the longest prefix rule is applied, which ensures that dd is parsed as a single token and not as twice the token d.

An input format does not have to refer to all the three parts (day, month and year) of a date; the parts which do not appear in the format are left unchanged when the format is used in an update operation. Applying the format ”d.m.”, for example, changes the day and the month part but not the year part. (The result of using input formats referring twice to the same part as in ”m M” is undefined.) Please see table 4.1 for an overview of all possible tokens.

```
#include <LEDA/system/date.h>
```

2. Types

\texttt{date::month} \{ Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec \}

The enumeration above allows to specify months by their name. Of course, one can also specify months by their number writing \texttt{date::month(m)}.

\texttt{date::language} \{ user_def_lang, local, english, german, french \}

When the language is set to local, the month names and weekday names are read from the local environment; the other identifiers are self-explanatory.
### Table 4.1: Token Overview

<table>
<thead>
<tr>
<th>token</th>
<th>input</th>
<th>output</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>yes</td>
<td>yes</td>
<td>day with 1 or 2 digits</td>
</tr>
<tr>
<td>dd</td>
<td>yes</td>
<td>yes</td>
<td>day with 2 digits (possibly with leading zero)</td>
</tr>
<tr>
<td>dth</td>
<td>yes</td>
<td>yes</td>
<td>day as abbreviated english ordinal number (1st, 2nd, 3rd, 4th, . . .)</td>
</tr>
<tr>
<td>m</td>
<td>yes</td>
<td>yes</td>
<td>month with 1 or 2 digits</td>
</tr>
<tr>
<td>mm</td>
<td>yes</td>
<td>yes</td>
<td>month with 2 digits (possibly with leading zero)</td>
</tr>
<tr>
<td>M</td>
<td>yes</td>
<td>yes</td>
<td>month name (when used in an input format this token must be followed by a single char (c) which does not belong to any month name, (c) is used to determine the end of the name. e.g.: &quot;(d.M.yy)&quot;&quot;)</td>
</tr>
<tr>
<td>M:l</td>
<td>yes</td>
<td>yes</td>
<td>the first (l) characters of the month name ((l) must be a single digit)</td>
</tr>
<tr>
<td>yy</td>
<td>yes</td>
<td>yes</td>
<td>year with 2 digits ((yy) is considered to represent a year in ([1950;2049]))</td>
</tr>
<tr>
<td>yyyy</td>
<td>yes</td>
<td>yes</td>
<td>year with 4 digits</td>
</tr>
<tr>
<td>[yy]yy</td>
<td>yes</td>
<td>yes</td>
<td>input: year with 2 or 4 digits / output: same as yyyy</td>
</tr>
<tr>
<td>w</td>
<td>no</td>
<td>yes</td>
<td>calendar week (in the range ([1;53])) (see <code>get_week()</code> for details)</td>
</tr>
<tr>
<td>diy</td>
<td>no</td>
<td>yes</td>
<td>day in the year (in the range ([1,366]))</td>
</tr>
<tr>
<td>dow</td>
<td>no</td>
<td>yes</td>
<td>day of the week (1=Monday, . . . , 7=Sunday)</td>
</tr>
<tr>
<td>DOW</td>
<td>no</td>
<td>yes</td>
<td>name of the weekday</td>
</tr>
<tr>
<td>DOW:l</td>
<td>no</td>
<td>yes</td>
<td>the first (l) characters of the weekday name ((l) must be a single digit)</td>
</tr>
<tr>
<td>&quot;txt&quot;</td>
<td>yes</td>
<td>yes</td>
<td>matches/prints <code>txt</code> (<code>txt</code> must not contain a double quote)</td>
</tr>
<tr>
<td>'txt'</td>
<td>yes</td>
<td>yes</td>
<td>matches/prints <code>txt</code> (<code>txt</code> must not contain a single quote)</td>
</tr>
<tr>
<td>c</td>
<td>yes</td>
<td>yes</td>
<td>matches/prints (c) ((c) $\notin {d, m, M, ?, *, ;}))</td>
</tr>
<tr>
<td>?</td>
<td>yes</td>
<td>no</td>
<td>matches a single arbitrary character</td>
</tr>
<tr>
<td>*c</td>
<td>yes</td>
<td>no</td>
<td>separates different formats, e.g. &quot;(d.M.yy;dd.mm.yy)&quot;</td>
</tr>
<tr>
<td>;</td>
<td>yes</td>
<td>yes</td>
<td>input: the first format that matches the input is used output: all but the first format is ignored</td>
</tr>
<tr>
<td>delim:c</td>
<td>yes</td>
<td>no</td>
<td>(c) serves as delimiter when reading input from streams (If this token is used, it must be the first in the format string.) When you use &quot;(\text{delim:}c;\text{d.M.yy})&quot; as input format to read a date from a stream, everything until the first occurrence of &quot;(c)&quot; is read and then the format &quot;(\text{d.M.yy})&quot; is applied.</td>
</tr>
</tbody>
</table>

### date::format

```plaintext
{ user_def_fmt, US_standard, german_standard, colons, hyphens }
```

The format `US_standard` is an abbreviation for `mm/dd/[yy]yy`, the format `german_standard` is the same as `dd.mm.[yy]yy`, the other formats are the same as the latter except that the periods are replaced by colons/hyphens.
3. Creation

date D;
creates an instance $D$ of type $date$ and initializes it to the current date.

date D(int d, month m, int y);
creates an instance $D$ of type $date$ and initializes it to $d.m.y$.
Precondition: $d.m.y$ represents a valid date.

date D(string date_str, bool swallow = true);
creates an instance $D$ of type $date$ and initializes it to date given in $date_str$.
If $swallow$ is true, then the format "m/d/[yy]yy; d?m?[yy]yy" is used to parse $date_str$, otherwise the current input format is applied.
Precondition: $date_str$ represents a valid date.

4. Operations

4.1 Languages and Input/Output Formats

void date::set_language(language l)
sets the language to $l$, which means that the month names and the weekday names are set according to the language.
Precondition: $l \neq user\_def\_lang$

void date::set_month_names(const char * names[])
sets the names for the months and changes the language to $user\_def\_lang$.
Precondition: $names[0..11]$ contains the names for the months from January to December.

void date::set_dow_names(const char * names[])
sets the names for the weekdays and changes the language to $user\_def\_lang$.
Precondition: $names[0..6]$ contains the names for the weekdays from Monday to Sunday.

language date::get_language() returns the current language.

void date::set_input_format(format f)
sets the input format to $f$.
Precondition: $f \neq user\_def\_fmt$

void date::set_input_format(string f)
sets the input format to the user-defined format in $f$.
Precondition: $f$ is a valid format string
4.2.1. A DATE INTERFACE (DATE)

**format**

```
date::get_input_format()
```
returns the current input format.

**string**

```
date::get_input_format_str()
```
returns the current input format string.

**void**

```
date::set_output_format(format f)
```
sets the output format to \( f \).

*Precondition: \( f \neq \text{user_def_fmt} \)*

```
date::set_output_format(string f)
```
sets the output format to the user-defined format in \( f \).

*Precondition: \( f \) is a valid format string*

**format**

```
date::get_output_format()
```
returns the current output format.

**string**

```
date::get_output_format_str()
```
returns the current output format string.

### 4.2 Access and Update Operations

All update operations which may fail have in common that the date is changed and \( \text{true} \) is returned if the new date is valid, otherwise \( \text{false} \) is returned and the date is left unchanged. (Note that the functions `add_to_day`, `add_to_month` and `add_to_year` can only fail if the valid range \((1.1.1 - 31.12.9999)\) is exceeded.)

**void**

```
D.set_to_current_date()
```
sets \( D \) to the current date.

**bool**

```
D.set_date(int d, month m, int y)
```
\( D \) is set to \( d.m.y \) (if \( d.m.y \) is valid).

**bool**

```
D.set_date(const string date_str, bool swallow = true)
```
\( D \) is set to the date contained in \( date_str \). If \( swallow \) is \( \text{true} \), then the format ”\( m/d/[yy]yy; d?m?[yy]yy \)” is used to parse \( date_str \), otherwise the current input format is applied.

**string**

```
D.get_date()
```
returns a string representation of \( D \) in the current output format.

**int**

```
D.get_day()
```
returns the day part of \( D \), i.e. if \( D \) is \( d.m.y \) then \( d \) is returned.

**month**

```
D.get_month()
```
returns the month part of \( D \).
string $D$.get_month_name() returns the name of the month of $D$ in the current language.

int $D$.get_year() returns the year part of $D$.

bool $D$.set_day(int $d$) sets the day part of $D$ to $d$, i.e. if $D$ is $d.m.y$ then $D$ is set to $d.m.y$.

bool $D$.add_to_day(int $d$) adds $d$ days to $D$ (cf. arithmetic operations).

bool $D$.set_month(month $m$) sets the month part of $D$ to $m$.

bool $D$.add_to_month(int $m$) adds $m$ months to the month part of $D$.

Let $D$ be $d.m'.y$, then it is set to $d.(m'+m).y$. If this produces an overflow (i.e. $m'+m > 12$) then the month part is repeatedly decremented by 12 and the year part is simultaneously incremented by 1, until the month part is valid. (An underflow (i.e. $m'+m < 1$) is treated analogously.) The day part of the result is set to the minimum of $d$ and the number of days in the resulting month.

bool $D$.set_year(int $y$) sets the year part of $D$ to $y$.

bool $D$.add_to_year(int $y$) adds $y$ years to the year part of $D$.

(If $D$ has the form $29.2.y'$ and $y'+y$ is no leap year, then $D$ is set to $28.2.(y'+y)$.)

int $D$.get_day_of_week() returns the day of the week of $D$. 
(1=Monday, 2=Tuesday, ..., 7=Sunday)

string $D$.get_dow_name() returns the name of the weekday of $D$ in the current language.

int $D$.get_week() returns the number of the calendar week of $D$ (range $[1,53]$). 
A week always ends with a Sunday. Every week belongs to the year which covers most of its days. (If the first Sunday of a year occurs before the fourth day of the year, then all days up to this Sunday belong to the last week of the preceding year. Similarly, if there are less than 4 days left after the last Sunday of a year, then these days belong to the first week of the succeeding year.)

int $D$.get_day_in_year() returns the number of the day in the year of $D$ (range $[1;366]$).
4.3 Arithmetic Operations

\[ date \; D + int \; d \] returns the date \( d \) days after \( D \).

\[ date \; D - int \; d \] returns the date \( d \) days before \( D \).

The related operators \( ++, --, +/, = \) and all comparison operators are also provided.

\[ int \; D - const \; date & \; D2 \] returns the difference between \( D \) and \( D2 \) in days.

\[ int \; D.\text{days}_{\text{until}}(const \; date & \; D2) \]
returns \( D2 - D \).

\[ int \; D.\text{months}_{\text{until}}(const \; date & \; D2) \]
if \( D2 \geq D \) then \( \max \{ m : D.\text{add}_{\text{to}}_{\text{month}}(m) \leq D2 \} \) is returned; otherwise the result is \( -D2.\text{months}_{\text{until}}(D) \).

\[ int \; D.\text{years}_{\text{until}}(const \; date & \; D2) \]
if \( D2 \geq D \) then \( \max \{ y : D.\text{add}_{\text{to}}_{\text{year}}(y) \leq D2 \} \) is returned; otherwise the result is \( -D2.\text{years}_{\text{until}}(D) \).

4.4 Miscellaneous Predicates

\[ bool \; date::\text{is}\text{\_valid}(int \; d, \; month \; m, \; int \; y) \]
returns \( true \) iff \( d.m.y \) represents a valid date.

\[ bool \; date::\text{is}\text{\_valid}(\text{string} \; d, \; bool \; swallow = true) \]
returns \( true \) iff \( d \) represents a valid date. If \( swallow \) is \( true \) the swallow format (cf. \( \text{set}	ext{\_date} \)) is used, otherwise the current input format is tried.

\[ bool \; date::\text{is}\text{\_leap}\text{\_year}(int \; y) \]
returns \( true \) iff \( y \) is a leap year.

\[ bool \; D.\text{is}\text{\_last}\text{\_day}\text{\_in}\text{\_month}() \]
let \( D \) be \( d.m.y \); the function return \( true \) iff \( d \) is the last day in the month \( m \) of the year \( y \).

5. Example

We count the number of Sundays in the days from now to 1.1.2010 using the following code chunk:

```c++
int number_of_Sundays = 0;
for (date D; D<=date(1,date::Jan,2010); ++D)
    if (D.get_day_of_week() == 7) ++number_of_Sundays;
```
Now we show an example in which different output formats are used:

```cpp
date D(2, date::month(11), 1973);
date::set_output_format(date::german_standard);
cout << D << endl; // prints "02.11.1973"
date::set_language(date::english);
date::set_output_format("dth M yyyy");
cout << D << endl; // prints "2nd November 1973"
```

Finally, we give an example for the usage of a multi-format. One can choose among 3 different formats:

1. If one enters only day and month, then the year part is set to the current year.

2. If one enters day, month and year providing only 2 digits for the year, the year is considered to be in the range \([1950, 2049]\). (Note that the date 1.1.10 must be written as "1.1.0010".)

3. One may also specify the date in full detail by entering 4 digits for the year.

The code to read the date in one of the formats described above looks like this:

```cpp
D.set_to_current_date(); // set year part to current year
date::set_input_format("delim:\n;d.m.\n;d.m.[yy]yy\n");
cin >> D; cout << D << endl;
```
Chapter 5

Alignment Data Types

This section describes several data types being used in the area of bioinformatics. The data types `alphabet` and `score_matrix` are just ingredients needed for the `alignment` type. `distance`, `string_matching`, and `alignment` are classes providing comparison and alignment functionality for a pair of strings. This chapter should be considered as the beginning of a new module. Users are encouraged to send their expectations with respect to future functionality to support@algorithmic-solutions.com.

5.1 Alphabet (alphabet)

1. Definition

An instance $A$ of the data type `alphabet` is an object containing a collection of letters. During a creation phase, letters can be added to or removed from the alphabet. If the final set of letters has been created, the alphabet has to be finished. Afterwards no further changes are accepted. The letters are lexicographically ordered in the alphabet.

Remark: This data type is experimental. It can significantly change in future versions.

```c
#include <LEDA/core/alphabet.h>
```

2. Creation

```c
alphabet A;
```
creates an instance $A$ of type `alphabet`.

```c
alphabet A(const string seq);
```
creates an instance $A$ of type `alphabet` and initializes $A$ to be the set of all different letters in `seq`. The alphabet is set to be finished by this constructor.

3. Operations

```c
void A.setStandard();
```
sets the standard alphabet consisting of the uppercase letters A-Z and finishes the alphabet construction phase.
\textbf{bool} \texttt{A.isStandard( )} \quad \text{returns true if the alphabet consists of the upper case letters A-Z, false otherwise.}

\textbf{void} \texttt{A.setDNA( )} \quad \text{sets the alphabet to the set consisting of the letters A,C,T, and G and finishes the alphabet construction phase.}

\textbf{bool} \texttt{A.isDNA( )} \quad \text{returns true if the alphabet consists of the letters A,C,T, and G.}

\textbf{void} \texttt{A.addLetter(char ch)} \quad \text{adds letter \texttt{ch} to \texttt{A}.}  
\hspace*{1em} \textit{Precondition:} \texttt{A} is not yet finished. If it is finished, an exception is thrown.

\textbf{void} \texttt{A.delLetter(char ch)} \quad \text{deletes letter \texttt{ch} from \texttt{A} if contained; otherwise, nothing happens.}  
\hspace*{1em} \textit{Precondition:} \texttt{A} is not yet finished. If it is finished, an exception is thrown.

\textbf{char} \texttt{A.getLetter(unsigned i)} \quad \text{returns the letter at position \texttt{i} of the lexicographically ordered set of letters, starting with position 0. If \texttt{i} is too big, an exception is thrown.}  
\hspace*{1em} \textit{Precondition:} \texttt{A} is finished. Otherwise an exception is thrown.

\textbf{void} \texttt{A.setFinished( )} \quad \text{finishes the alphabet.}

\textbf{bool} \texttt{A.getFinished( )} \quad \text{returns \texttt{true} if \texttt{A} is finished, \texttt{false} otherwise.}

\textbf{bool} \texttt{A.isFinished( )} \quad \text{returns \texttt{true} if \texttt{A} is finished, \texttt{false} otherwise.}

\textbf{string} \texttt{A.getLetters( )} \quad \text{returns a string containing all letters of \texttt{A} in lexicographical order.}  
\hspace*{1em} \textit{Precondition:} \texttt{A} is finished. Otherwise an exception is thrown.

\textbf{int} \texttt{A.getLength( )} \quad \text{returns the size of \texttt{A}, that is the number of different letters in \texttt{A}.}  
\hspace*{1em} \textit{Precondition:} \texttt{A} is finished. Otherwise an exception is thrown.

\textbf{int} \texttt{A.getSize( )} \quad \text{same as \texttt{getLength( ).}}
5.2 String Matching Algorithms (string_matching)

1. Definition

An instance \( M \) of the data type \textit{string_matching} is an object maintaining a pattern and a string. It provides a collection of different algorithms for computation of the exact string matching problem. Each function computes a list of all starting positions of occurrences of the pattern in the string. The algorithms can be called explicitly by their name or implicitly by making an algorithm the "current" algorithm (specified by the \textit{setAlgorithm}-method) and using functions \textit{getFirstOccurrence()}, \textit{getNextOccurrence()}, and \textit{getAllOccurences()}.

Here is the list of the available parameters to specify the algorithm in the \textit{setAlgorithm(alg)}-method. For each algorithm, the enumeration index gives the corresponding \textit{int} that can be used in the \textit{setAlgorithm(int)}-method in order to specify the algorithm.

1. \textit{KARP_RABIN}
2. \textit{MORRIS_PRATT}
3. \textit{KNUTH_MORRIS_PRATT}
4. \textit{GALIL_SEIFERAS}
5. \textit{BOYER_MOORE}
6. \textit{TURBO_BOYER_MOORE}
7. \textit{TUNED_BOYER_MOORE}
8. \textit{ZHU_TAKAOKA}
9. \textit{BERRY_RAVINDRAN}
10. \textit{SMITH}
11. \textit{RAITA}
12. \textit{HORSPOOL}
13. \textit{BRUTE_FORCE}
14. \textit{SHIFT_OR}
15. \textit{REVERSE_COLUMUSI}
16. \textit{QUICK_SEARCH}
17. \textit{REVERSE\_FACTOR}
18. \textit{TURBO\_REVERSE\_FACTOR}
19. \textit{OPTIMAL\_MISMATCH}

Remark: This data type is experimental. It can significantly change in future versions.

\#include < LEDA/core/string\_matching.h >

2. Creation

\texttt{string\_matching } M; \textit{ creates an instance } M \textit{ of type } string\_matching.

\texttt{string\_matching } M(\texttt{const string}& s1, \texttt{const string}& s2);
\textit{ creates an instance } M \textit{ of type } string\_matching \textit{ and initializes the pattern to } s1 \textit{ and the string to } s2.

3. Operations

\texttt{void } M.\texttt{setPattern(}const \texttt{string}& \texttt{str})
\texttt{sets the pattern to } str.

\texttt{void } M.\texttt{setString(}const \texttt{string}& \texttt{str})
\texttt{sets the string to } str.

\texttt{string } M.\texttt{getPattern( )} \texttt{returns the pattern.}

\texttt{string } M.\texttt{getString( )} \texttt{returns the string.}

\texttt{void } M.\texttt{setStrings(}const \texttt{string}& \texttt{str1}, \texttt{const string}& \texttt{str2})
\texttt{sets the pattern to } str1 \texttt{ and the string to } str2.

\texttt{void } M.\texttt{setAlgorithm(}alg \texttt{A}) \texttt{sets the current algorithm to } A \texttt{ (see table above).}

\texttt{void } M.\texttt{setAlgorithm(}int \texttt{A}) \texttt{sets the current algorithm to } A \texttt{ by index (see table above).}

\texttt{list<int> } M.\texttt{getAllOccurences( )} \texttt{executes the current algorithm reporting all occurrences of the pattern in the string.}

\texttt{int } M.\texttt{getFirstOccurence( )} \texttt{executes the current algorithm reporting the first occurrence of the pattern in the string. If there is none, -1 is returned.}

\texttt{int } M.\texttt{getNextOccurence(}int \texttt{i}) \texttt{executes the current algorithm reporting the first occurrence after position } i \texttt{ of the pattern in the string. If there is none, -1 is returned. String positions start with 0.}
5.2. STRING MATCHING ALGORITHMS (STRING_MATCHING)

```c
list<int> M.getResult() returns the list of results after a computation.
void M.karp_rabin() applies the algorithm of Karp and Rabin.
void M.morris_pratt() applies the algorithm of Morris and Pratt.
void M.knuth_morris_pratt() applies the algorithm of Knuth, Morris and Pratt.
void M.gali_seiferas() applies the algorithm of Galil and Seiferas.
void M.boyer_moore() applies the algorithm of Boyer and Moore.
void M.turbo_boyer_moore() applies the turbo algorithm of Boyer and Moore.
void M.tuned_boyer_moore() applies the tuned algorithm of Boyer and Moore.
void M.zhu_takaoka() applies the algorithm of Zhu and Takaoka.
void M.berry_ravindran() applies the algorithm of Berry and Ravindran.
void M.smith() applies the algorithm of Smith.
void M.raita() applies the algorithm of Raita.
void M.horspool() applies the algorithm of Horspool.
void M.brute_force() applies the naive brute force algorithm.
void M.shift_or() applies the shift-or algorithm.
void M.reverse_colussi() applies the reverse colussi algorithm.
void M.quick_search() applies the quick search algorithm.
void M.reverse_factor() applies the reverse factor algorithm.
void M.turbo_reverse_factor() applies the turbo reverse colussi algorithm.
void M.optimal_mismatch() applies the optimal mismatch algorithm.
```
5.3 Score Matrix (score_matrix)

1. Definition

An instance $S$ of the data type score_matrix is an object required by alignment computations. The object contains an alphabet representing residues and maintains a table that assigns a score to any pair of aligned residues. In addition it maintains the gap penalty values.

Remark: This data type is experimental. It can significantly change in future versions.

```cpp
#include <LEDA/core/score_matrix.h>
```

2. Creation

```cpp
score_matrix S(int defmatch = 1, int defmismatch = -1);
```

creates an instance $S$ of type score_matrix. The alphabet is set to the standard alphabet consisting of the capital letters A - Z. The scores are initialized to defmatch if the letters are identical (1 by default), to defmismatch otherwise (-1 by default).

```cpp
score_matrix S(alphabet& alt, int defmatch = 1, int defmismatch = -1);
```

creates an instance $S$ of type score_matrix. The alphabet is set to alt. The scores are initialized to defmatch if the letters are identical (1 by default), to defmismatch otherwise (-1 by default).

3. Operations

```cpp
void S.setScore(char a, char b, int sc)
```

sets the score for $a$ and $b$ to $sc$.

**Precondition:** $a$ and $b$ are in the score alphabet. Otherwise an exception is thrown.

```cpp
int S.getScore(char a, char b)
```

returns the score for $a$ and $b$.

**Precondition:** $a$ and $b$ are in the score alphabet. Otherwise an exception is thrown.

```cpp
bool S.hasScore(char a, char b)
```

returns `true` if score is defined for $a$ and $b$, otherwise `false`.

```cpp
void S.setScoreMatch(int sc)
```

sets the score for all pairs $a$ and $a$ to $sc$.

```cpp
void S.setScoreMismatch(int sc)
```

sets the score for all pairs $a$ and $b$ with $a \neq b$ to $sc$.

```cpp
void S.setGapPenalty(int pen)
```

sets the penalty for starting a new gap to $pen$. 
5.3. SCORE MATRIX (SCORE MATRIX)

\begin{verbatim}
void S.setGapIncPenalty(int incpen)
    sets the penalty for extending a gap by one position to incpen.

void S.setGapPenalty(int pen, int incpen)
    sets the penalty for starting a new gap to pen and for extending a gap by 1 position to incpen.

int S.getGapPenalty()
    returns the penalty for starting a new gap.

int S.getGapIncPenalty()
    returns the penalty for extending a gap by one position.
\end{verbatim}
5.4 String Distance Function (distance)

1. Definition

An instance $D$ of the data type $distance$ is an object maintaining two strings and providing several string distance functions.

The Hamming distance between two strings of equal length is the number of positions for which the corresponding symbols are different.

The Levenshtein distance or edit distance between two strings is given by the minimum number of operations needed to transform one string into the other, where an operation is an insertion, deletion, or substitution of a single character.

The General Levenshtein distance is given by the minimum sum of the costs over a sequence of operations needed to transform one string into the other, where operation is an insertion, deletion, or substitution and a cost is assigned to each of the operations.

The Damerau-Levenshtein distance is an extension of Levenshtein distance that counts transposition as a single edit operation. Strictly speaking, the Damerau-Levenshtein distance is equal to the minimal number of insertions, deletions, substitutions and transpositions needed to transform one string into the other.

The algorithms can be called explicitly by their name or implicitly by making an algorithm the ”current” algorithm (method $setAlgorithm(.)$ and using the $run()$-method.

Here is the list of the available parameters to specify the algorithm in the $setAlgorithm(alg)$-method in order to call it by the $run()$-method. For each algorithm, the enumeration index gives the corresponding $int$ that can be used in the $setAlgorithm(int)$-method in order to specify the algorithm.

1. HAMMING
2. LEVENSHTEIN
3. DAMERAU_LEVENSHTEIN
4. GENERAL_LEVENSHTEIN

Remark: This data type is experimental. It can significantly change in future versions.

#include <LEDA/core/distance.h>

2. Creation

distance $D$; creates an instance $D$ of type $distance$.

distance $D$(const string& $s1$, const string& $s2$);
creates an instance $D$ of type $distance$ and initializes the two strings to $s1$ resp. $s2$. 
3. Operations

`int D.hamming()` returns the Hamming distance of two strings. If the strings have different lengths, -1 is returned.

`int D.levenshtein()` returns the Levenshtein distance of two strings.

`int D.damerau_levenshtein()` returns the Damerau Levenshtein distance of two strings.

`unsigned int D.general_levenshtein()` returns the General Levenshtein distance of two strings.

`void D.setAlgorithm(alg A)` sets the current algorithm to A (see table above).

`void D.setAlgorithm(int A)` sets the current algorithm to A by index (see table above).

`int D.run()` executes the current algorithm.

`void D.setFirstString(const string& str)` assigns str to the first string.

`void D.setSecondString(const string& str)` assigns str to the second string.

`string D.getFirstString()` returns first string.

`string D.getSecondString()` returns second string.
5.5 Alignment Functions (alignment)

1. Definition

An instance $A$ of the data type alignment is an object maintaining two strings, a score_matrix, a penalty value for starting a gap and a penalty value for extending an already started gap. It provides several functions for global and local alignment computation. The result of an alignment computation is stored in the object.

Remark: This data type is experimental. It can significantly change in future versions.

```c
#include <LEDA/core/alignment.h>
```

2. Creation

- `alignment A;` creates an instance $A$ of type alignment.
- `alignment A(const score_matrix& score);` creates an instance $A$ of type alignment and initializes the score matrix to score.
- `alignment A(const score_matrix& score, const string& s1, const string& s2);` creates an instance $A$ of type alignment and initializes the score matrix to score, and the strings to $s_1$ and $s_2$.
- `alignment A(const string& s1, const string& s2);` creates an instance $A$ of type alignment and initializes the strings to $s_1$ and $s_2$.

3. Operations

- `void A.setStringA(const string& x)` sets the first string to $x$.
- `void A.setStringB(const string& x)` sets the second string to $x$.
- `string A.getStringA()` returns the first string.
- `string A.getStringB()` returns the second string.
- `string A.getAlA()` returns the first alignment string, that is the first string with its gaps after an alignment computation.
- `string A.getAlB()` returns the second alignment string, that is the second string with its gaps after an alignment computation.
5.5. ALIGNMENT FUNCTIONS (ALIGNMENT)

`string A.getAllInfo()` shows the correspondence information for each position in the two alignment strings. Its i-th position is "/", if the letters are identical, "X" if the are different but they are matched and " " if there is a gap in one of the alignment strings.

`void A.setGapPenalty(int x)` sets the penalty for a gap to x.

`void A.setGapIncPenalty(int x)` sets the penalty for extending an existing gap by one position to x.

`int A.getGapPenalty()` returns the penalty for a gap.

`int A.getGapIncPenalty()` returns the penalty for a gap extension of one position.

`void A.global_nw()` runs a global alignment algorithm based on Needleman and Wunsch, where there is a penalty x for each gap proportional to its length. x can be set by `setGapPenalty`. The algorithm requires quadratic space.

`void A.local_sw()` runs a local alignment algorithm based on Smith and Waterman, where there is a penalty x for each gap proportional to its length. x can be set by `setGapPenalty`. The algorithm requires quadratic space.

`void A.global_aff_nw()` runs a global alignment algorithm based on Needleman and Wunsch, where there is a penalty $x + k \times y$ for each gap, where k is the length of the gap. x can be set by `setGapPenalty` and x by `setGapIncPenalty`. The algorithm requires quadratic space.

`void A.local_aff_sw()` runs a local alignment algorithm based on Smith and Waterman, where there is a penalty $x + k \times y$ for each gap, where k is the length of the gap. x can be set by `setGapPenalty` and x by `setGapIncPenalty`. The algorithm requires quadratic space.

`unsigned int A.longest_common_substring()` returns the length of the longest common substring of the two strings.

`unsigned int A.longest_common_subsequence()` returns the length of the longest common subsequence of the two strings and stores them aligned in the alignment strings.
Chapter 6

Number Types and Linear Algebra

6.1 Integers of Arbitrary Length (integer)

1. Definition

An instance of the data type integer is an integer number of arbitrary length. The internal representation of an integer consists of a vector of so-called digits and a sign bit. A digit is an unsigned long integer (type unsigned long).

#include <LEDA/numbers/integer.h>

2. Creation

integer a; creates an instance a of type integer and initializes it with zero.

integer a(int n); creates an instance a of type integer and initializes it with the value of n.

integer a(unsigned int i); creates an instance a of type integer and initializes it with the value of i.

integer a(long l); creates an instance a of type integer and initializes it with the value of l.

integer a(unsigned long i); creates an instance a of type integer and initializes it with the value of i.

integer a(double x); creates an instance a of type integer and initializes it with the integral part of x.

integer a(unsigned int sz, const digit * vec, int sign = 1); creates an instance a of type integer and initializes it with the value represented by the first sz digits vec and the sign.


integer a(const char * s);

    a creates an instance a of type integer from its decimal representation given by the string s.

integer a(const string & s);

    a creates an instance a of type integer from its decimal representation given by the string s.

3. Operations

The arithmetic operations +, −, *, /, +=, −=, *=, /=, −(unary), ++, --, the modulus operation (%, %=), bitwise AND (&, &=), bitwise OR (|, |=), the complement (~), the shift operations (<<, >>), the comparison operations <, <=, >, >=, ==, != and the stream operations all are available.

int a.sign() returns the sign of a.

int a.length() returns the number of bits of the representation of a.

bool a.is_long() returns whether a fits in the data type long.

long a.to_long() returns a long number which is initialized with the value of a. Precondition: a.is_long() is true.

double a.to_double() returns a double floating point approximation of a.

double a.to_double(bool & is_double) as above, but also returns in is_double whether the conversion was exact.

double a.to_float() as above.

string a.to_string() returns the decimal representation of a.

integer & a.from_string(string s) sets a to the number that has decimal representation s.

sz_t a.used_words() returns the length of the digit vector that represents a.

digit a.highword() returns the most significant digit of a.

digit a.contents(int i) returns the i-th digit of a (the first digit is a.contents(0)).

void a.hex_print(ostream & o) prints the digit vector that represents a in hex format to the output stream o.

bool a.iszero() returns whether a is equal to zero.
Non-member functions

\begin{itemize}
\item \texttt{double to\_double(const integer& a)} returns a double floating point approximation of \texttt{a}.
\item \texttt{integer sqrt(const integer& a)} returns the largest \texttt{integer} which is not larger than the square root of \texttt{a}.
\item \texttt{integer abs(const integer& a)} returns the absolute value of \texttt{a}.
\item \texttt{integer factorial(const integer& n)} returns \texttt{n}!.
\item \texttt{integer gcd(const integer& a, const integer& b)} returns the greatest common divisor of \texttt{a} and \texttt{b}.
\item \texttt{int log(const integer& a)} returns the logarithm of \texttt{a} to the basis 2 (rounded down).
\item \texttt{int log2\_abs(const integer& a)} returns the logarithm of \texttt{|a|} to the basis 2 (rounded up).
\item \texttt{int sign(const integer& a)} returns the sign of \texttt{a}.
\item \texttt{integer sqrt(const integer& a)} returns \texttt{a}^2.
\item \texttt{double double\_quotient(const integer& a, const integer& b)} returns a the best possible floating-point approximation of \texttt{a/b}.
\item \texttt{integer integer::random(int n)} returns a random integer of length \texttt{n} bits.
\end{itemize}

4. Implementation

An \texttt{integer} is essentially implemented by a vector \texttt{vec} of \texttt{unsigned long} numbers. The sign and the size are stored in extra variables. Some time critical functions are also implemented in assembler code.
6.2 Rational Numbers (rational)

1. Definition

An instance \( q \) of type \( \text{rational} \) is a rational number where the numerator and the denominator are both of type \( \text{integer} \).

```
#include <LEDA/numbers/rational.h>
```

2. Creation

- \( \text{rational} \ q; \) creates an instance \( q \) of type \( \text{rational} \).
- \( \text{rational} \ q(\text{integer} \ n); \) creates an instance \( q \) of type \( \text{rational} \) and initializes it with the integer \( n \).
- \( \text{rational} \ q(\text{integer} \ n, \text{integer} \ d); \) creates an instance \( q \) of type \( \text{rational} \) and initializes it to the rational number \( n/d \).
- \( \text{rational} \ q(\text{double} \ x); \) creates an instance \( q \) of type \( \text{rational} \) and initializes it with the value of \( x \).

3. Operations

The arithmetic operations +, −, *, /, +=, −=, *=, /=, -(unary), ++, −−, the comparison operations <, <=, >, >=, ==, != and the stream operations are all available.

- \( \text{void} \ q.\text{negate}(\ ) \) negates \( q \).
- \( \text{void} \ q.\text{invert}(\ ) \) inverts \( q \).
- \( \text{rational} \ q.\text{inverse}(\ ) \) returns the inverse of \( q \).
- \( \text{integer} \ q.\text{numerator}(\ ) \) returns the numerator of \( q \).
- \( \text{integer} \ q.\text{denominator}(\ ) \) returns the denominator of \( q \).
- \( \text{rational} & q.\text{simplify}(\text{const integer}& \ a) \) simplifies \( q \) by \( a \).  
  \textbf{Precondition:} \( a \) divides the numerator and the denominator of \( q \).
- \( \text{rational} & q.\text{normalize}(\ ) \) normalizes \( q \).
double to_float() returns a double floating point approximation of \( q \). If the \( q \) is approximable by a normalized, finite floating point number, the error is 3ulps, i.e., three units in the last place.

string \( q\).to_string() returns a string representation of \( q \).

**Non-member functions**

int sign(const rational& \( q \)) returns the sign of \( q \).

rational abs(const rational& \( q \)) returns the absolute value of \( q \).

rational sqr(const rational& \( q \)) returns the square of \( q \).

integer trunc(const rational& \( q \)) returns the \texttt{integer} with the next smaller absolute value.

rational pow(const rational& \( q \), int \( n \)) returns the \( n \)-th power of \( q \).

rational pow(const rational& \( q \), integer \( a \)) returns the \( a \)-th power of \( q \).

integer floor(const rational& \( q \)) returns the next smaller \texttt{integer}.

integer ceil(const rational& \( q \)) returns the next bigger \texttt{integer}.

integer round(const rational& \( q \)) rounds \( q \) to the nearest \texttt{integer}.

rational small_rational_between(const rational& \( p \), const rational& \( q \)) returns a rational number between \( p \) and \( q \) whose denominator is as small as possible.

rational small_rational_near(const rational& \( p \), rational \( eps \)) returns a rational number between \( p - eps \) and \( p + eps \) whose denominator is as small as possible.

**4. Implementation**

A \texttt{rational} is implemented by two \texttt{integer} numbers which represent the numerator and the denominator. The sign is represented by the sign of the numerator.
6.3 The data type bigfloat (bigfloat)

1. Definition

In general a bigfloat is given by two integers $s$ and $e$ where $s$ is the significant and $e$ is the exponent. The tuple $(s, e)$ represents the real number $s \cdot 2^e$.

In addition, there are the special bigfloat values NaN (not a number), pZero, nZero ($= +0, -0$), and pInf, nInf ($= +\infty, -\infty$). These special values behave as defined by the IEEE floating point standard. In particular, $\frac{5}{10} = \infty$, $\frac{-5}{10} = -\infty$, $\infty + 1 = \infty$, $\frac{5}{\infty} = +0$, $+\infty + (-\infty) = NaN$ and $0 \cdot \infty = NaN$.

Arithmetic on bigfloats uses two parameters: The precision $\text{prec}$ of the result (in number of binary digits) and the rounding mode $\text{mode}$. Possible rounding modes are:

- TO_NEAREST: round to the closest representable value
- TO_ZERO: round towards zero
- TO_INF: round away from zero
- TO_P_INF: round towards $+\infty$
- TO_N_INF: round towards $-\infty$
- EXACT: compute exactly for $+,-,\ast$ and round to nearest otherwise

Operations $+, -, \ast$ work as follows. First, the exact result $z$ is computed. If the rounding mode is EXACT then $z$ is the result of the operation. Otherwise, let $s$ be the significant of the result; $s$ is rounded to $\text{prec}$ binary places as dictated by $\text{mode}$. Operations $/$ and $\sqrt{}$ work accordingly except that EXACT is treated as TO_NEAREST.

The parameters $\text{prec}$ and $\text{mode}$ are either set directly for a single operation or else they are set globally for every operation to follow. The default values are 53 for $\text{prec}$ and TO_NEAREST for $\text{mode}$. 

```cpp
#include <LEDA/numbers/bigfloat.h>
```

2. Creation

A bigfloat may be constructed from data types double, long, int and integer, without loss of accuracy. In addition, an instance of type bigfloat can be created as follows.

```cpp
bigfloat x(const integer& s, const integer& e);
```

introduces a variable $x$ of type bigfloat and initializes it to $s \cdot 2^e$.
6.3. **THE DATA TYPE BIGFLOAT (BIGFLOAT)**

\[ \text{double } x.\text{to\_double( ) returns the double value next to } x \text{ (i.e. rounding mode is always } \text{TO\_NEAREST}). \]

\[ \text{double } x.\text{to\_double(bool\& is\_double)} \]

\[ \text{as above, but also returns in } \text{is\_double} \text{ whether the conversion was exact.} \]

\[ \text{double } x.\text{to\_double(double\& abs\_err, rounding\_modes m = } \text{TO\_NEAREST)} \]

\[ \text{as above, but with more flexibility: The parameter } m \text{ specifies the rounding mode. For the returned value } d, \text{ we have } |x - d| \leq \text{abs\_err}. \text{ (abs\_err is zero iff the conversion is exact and the returned value is finite.)} \]

\[ \text{double } x.\text{to\_double(rounding\_modes m)} \]

\[ \text{as above, but does not return an error bound.} \]

\[ \text{rational } x.\text{to\_rational( ) converts } x \text{ into a number of type rational.} \]

\[ \text{sz\_t } x.\text{get\_significant\_length(void)} \]

\[ \text{returns the length of the significant of } x. \]

\[ \text{sz\_t } x.\text{get\_effective\_significant\_length(void)} \]

\[ \text{returns the length of the significant of } x \text{ without trailing zeros.} \]

\[ \text{integer } x.\text{get\_exponent(void)} \]

\[ \text{returns the exponent of } x. \]

\[ \text{integer } x.\text{get\_significant(void)} \]

\[ \text{returns the significant of } x. \]

\[ \text{sz\_t } \text{bigfloat::set\_precision(sz\_t p)} \]

\[ \text{sets the global arithmetic precision to } p \text{ binary digits and returns the old value} \]

\[ \text{sz\_t } \text{bigfloat::get\_precision( )} \]

\[ \text{returns the currently active global arithmetic precision.} \]

\[ \text{sz\_t } \text{bigfloat::set\_output\_precision(sz\_t d)} \]

\[ \text{sets the precision of } \text{bigfloat output to } d \text{ decimal digits and returns the old value} \]

\[ \text{sz\_t } \text{bigfloat::set\_input\_precision(sz\_t p)} \]

\[ \text{sets the precision of } \text{bigfloat input to } p \text{ binary digits and returns the old value} \]

\[ \text{rounding\_modes } \text{bigfloat::set\_rounding\_mode(rounding\_modes m)} \]

\[ \text{sets the global rounding mode to } m \text{ and returns the old rounding mode.} \]
rounding
modes bigfloat::get_rounding_mode()
returns the currently active global rounding mode

output
modes bigfloat::set_output_mode(output
modes o_mode)
sets the output mode to o_mode and returns the old
output mode

A bigfloat x can be rounded by the call round(x, prec, mode, is_exact). The optional
boolean variable is_exact is set to true if and only if the rounding operation did not
change the value of x.

integer to_integer(rounding
modes rmode = TO_NEAREST,
bool& is_exact = bigfloat::dbool)
returns the integer value next to x (in the given round-
ing mode)

integer to_integer(const bigfloat& x, rounding
modes rmode, bool& is_exact)
returns x.to_integer(…).

3. Operations

The arithmetical operators +, −, ∗, /, +, −, ∗, /, sqrt, the comparison operators
<, ≤, >, ≥ , =, ≠ and the stream operators are available. Addition, subtraction, mul-
tiplication, division, square root and power are implemented by the functions add, sub,
mul, div, sqrt and power respectively. For example, the call

add(x, y, prec, mode, is_exact)

computes the sum of bigfloats x and y with prec binary digits, in rounding mode mode,
and returns it. The optional last parameter is_exact is a boolean variable that is set
to true if and only if the returned bigfloat exactly equals the sum of x and y. The
parameters prec and mode are also optional and have the global default values global_prec
and round_mode respectively, that is, the three calls add(x, y, global_prec, round_mode),
add(x, y, global_prec), and add(x, y) are all equivalent. The syntax for functions sub,
mul, div, and sqrt is analogous.

The operators +, −, ∗, and / are implemented by their counterparts among the functions
add, sub, mul and div. For example, the call x + y is equivalent to add(x, y).

bool isnan(const bigfloat& x)
returns true if and only if x is in special state NaN

bool isnInf(const bigfloat& x)
returns true if and only if x is in special state nInf

bool ispInf(const bigfloat& x)
returns true if and only if x is in special state pInf
6.3. THE DATA TYPE BIGFLOAT (BIGFLOAT)

**bool isnZero(const bigfloat& x)**
returns `true` if and only if `x` is in special state `nZero`

**bool ispZero(const bigfloat& x)**
returns `true` if and only if `x` is in special state `pZero`

**bool isZero(const bigfloat& x)**
returns `true` if and only if `ispZero(x)` or `isnZero(x)`

**bool isInf(const bigfloat& x)**
returns `true` if and only if `ispInf(x)` or `isnInf(x)`

**bool isSpecial(const bigfloat& x)**
returns `true` if and only if `x` is in a special state

**int sign(const bigfloat& x)**
returns the sign of `x`.

**bigfloat abs(const bigfloat& x)**
returns the absolute value of `x`

**bigfloat ipow2(const integer& p)**
returns $2^p$

**integer ilog2(const bigfloat& x)**
returns the binary logarithm of `abs(x)`, rounded up to the
next integer. **Precondition:** $x \neq 0$

**integer ceil(const bigfloat& x)**
returns `x`, rounded up to the next integer

**integer floor(const bigfloat& x)**
returns `x`, rounded down to the next integer

**bigfloat sqrt_d(const bigfloat& x, sz_t p, int d)**
returns $\sqrt{\text{x}}$, with relative error $\leq 2^{-p}$ but not necessarily
exactly rounded to `p` binary digits

**string x.to_string(sz_t dec_prec = global_output_prec)**
returns the decimal representation of `x`, rounded to a decimal
precision of `dec_prec` decimal places.

**bigfloat& x.from_string(string s, sz_t bin_prec = global_input_prec)**
returns an approximation of the decimal number given by the
string `s` by a `bigfloat` that is accurate up to `bin_prec` binary
digits

**ostream& ostream& os << const bigfloat& x**
writes `x` to output stream `os`
$\texttt{istream} \& \texttt{istream} \& \texttt{is} \, \Rightarrow \, \texttt{bigfloat} \& \texttt{x}$

reads $x$ from input stream $\texttt{is}$ in decimal format
6.4  The data type real ( real )

1. Definition

An instance $x$ of the data type real is a real algebraic number. There are many ways to construct a real: either by conversion from double, bigfloat, integer or rational, by applying one of the arithmetic operators $+, -, *, /$ or $\sqrt{}$ to real numbers or by using the $\diamond$-operator to define a real root of a polynomial over real numbers. One may test the sign of a real number or compare two real numbers by any of the comparison relations $=, \neq, <, \leq, >, \geq$. The outcome of such a test is mathematically exact. We give an example expression to clarify this:

$$x := (\sqrt{17} - \sqrt{12}) \ast (\sqrt{17} + \sqrt{12}) - 5$$

Clearly, the value of $x$ is zero. But if you evaluate $x$ using double arithmetic you obtain a tiny non-zero value due to rounding errors. If the data type real is used to compute $x$ then $\text{sign}(x)$ yields zero. There is also a non–standard version of the sign function: the call $x.\text{sign}(\text{integer } q)$ computes the sign of $x$ under the precondition that $|x| \leq 2^{-q}$ implies $x = 0$. This version of the sign function allows the user to assist the data type in the computation of the sign of $x$, see the example below.

There are several functions to compute approximations of reals. The calls $x.\text{to}\text{bigfloat}( )$ and $x.\text{get}\text{bigfloat}\text{error}( )$ return bigfloats $\text{xnum}$ and $\text{xerr}$ such that $|\text{xnum} - x| \leq \text{xerr}$. The user may set a bound on $\text{xerr}$. More precisely, after the call $x.\text{improve}\text{approximation}\text{to}(\text{integer } q)$ the data type guarantees $\text{xerr} \leq 2^{-q}$. One can also ask for double approximations of a real number $x$. The calls $x.\text{to}\text{double}( )$ and $x.\text{get}\text{double}\text{error}( )$ return doubles $\text{xnum}$ and $\text{xerr}$ such that $|\text{xnum} - x| \leq \text{xerr}$. Note that $\text{xerr} = \infty$ is possible.

```c
#include <LEDA/numbers/real.h>
```

2. Types

typedef polynomial<real> Polynomial  // the polynomial type.

3. Creation

reals may be constructed from data types double, bigfloat, long, int and integer. The default constructor real( ) initializes the real to zero.

4. Operations

`double x.to_double( )` returns the current double approximation of $x$.

`double x.to_double(double& error)`

as above, but also computes a bound on the absolute error.
bigfloat $x$.to_bigfloat() returns the current bigfloat approximation of $x$.

double $x$.get_double_error() returns the absolute error of the current double approximation of $x$, i.e., $|x - x$.to_double()| $\leq x$.get_double_error().

bigfloat $x$.get_bigfloat_error() returns the absolute error of the current bigfloat approximation of $x$, i.e., $|x - x$.to_bigfloat()| $\leq x$.get_bigfloat_error().

bigfloat $x$.get_lower_bound() returns the lower bound of the current interval approximation of $x$.

bigfloat $x$.get_upper_bound() returns the upper bound of the current interval approximation of $x$.

rational $x$.high() returns a rational upper bound of the current interval approximation of $x$.

rational $x$.low() returns a rational lower bound of the current interval approximation of $x$.

double $x$.get_double_lower_bound() returns a double lower bound of $x$.

double $x$.get_double_upper_bound() returns a double upper bound of $x$.

bool $x$.possible_zero() returns true if 0 is in the current interval approximation of $x$.

integer $x$.separation_bound() returns the separation bound of $x$.

integer $x$.sep_bfms() returns the k-ary BFMSS separation bound of $x$.

integer $x$.sep_degree_measure() returns the degree measure separation bound of $x$.

integer $x$.sep_li_yap() returns the Li / Yap separation bound of $x$.

void $x$.print_separation_bounds() prints the different separation bounds of $x$.

bool $x$.is_general() returns true if the expression defining $x$ contains a $\diamond$-operator, false otherwise.

bool $x$.is_rational() returns true if the expression is rational, false otherwise.
6.4. THE DATA TYPE REAL (REAL)

\[ \text{rational } \text{x.to_rational()} \text{ returns the rational number given by the expression.} \]
\[ \text{Precondition: is_rational()} \text{ has is true.} \]

\[ \text{int } \text{x.compare(const real } \& \text{ y)} \]
\[ \text{returns the sign of x-y.} \]

\[ \text{int } \text{compare_all(const growing_array<real>& } \text{ R, int } \& \text{ j)} \]
\[ \text{compares all elements in } \text{R. It returns } i \text{ such that } \text{R}[i] = \text{R}[j] \]
\[ \text{and } i \neq j. \text{ Precondition: Only two of the elements in } \text{R} \text{ are equal. [Experimental]} \]

\[ \text{int } \text{x.sign()} \]
\[ \text{returns the sign of (the exact value of) } \text{x.} \]

\[ \text{int } \text{x.sign(const integer } \& \text{ q)} \]
\[ \text{as above. Precondition: The user guarantees that } |x| \leq 2^{-q} \]
\[ \text{is only possible if } x = 0. \text{ This advanced version of the sign function should be applied only by the experienced user. It gives an improvement over the plain sign function only in some cases.} \]

\[ \text{void } \text{x.improve_approximation_to(const integer } \& \text{ q)} \]
\[ \text{recomputes the approximation of } \text{x if necessary; the resulting error of the bigfloat approximation satisfies} \]
\[ \text{x.get_bigfloat_error()} \leq 2^{-q} \]

\[ \text{void } \text{x.compute_with_precision(long } k) \]
\[ \text{recomputes the bigfloat approximation of } \text{x, if necessary; each numerical operation is carried out with a mantissa length of } k. \]
\[ \text{Note that here the size of the resulting } \text{x.get_bigfloat_error()} \text{ cannot be predicted in general.} \]

\[ \text{void } \text{x.guarantee_relative_error(long } k) \]
\[ \text{recomputes an approximation of } \text{x, if necessary; the relative error of the resulting bigfloat approximation is less than } 2^{-k}, \]
\[ \text{i.e., } \text{x.get_bigfloat_error()} \leq |x| \cdot 2^{-k}. \]

\[ \text{ostream} \& \text{ ostream} \& \text{ O } \ll \text{ const real } \& \text{ x} \]
\[ \text{writes the closest interval that is known to contain } x \text{ to the output stream } O. \text{ Note that the exact representation of } x \text{ is lost in the stream output.} \]

\[ \text{istream} \& \text{ istream} \& \text{ I } \gg \text{ real } \& \text{ x} \]
\[ \text{reads } x \text{ number } x \text{ from the output stream } I \text{ in double format. Note that stream input is currently impossible for more general types of reals.} \]
real $\sqrt{\text{const real } x}$

real $\sqrt[d]{\text{const real } x}$, precondition: $d \geq 2$

**Note:** The functions real roots and diamond below are all experimental if they are applied to a polynomial which is not square-free.

$$\text{real roots(}\text{const Polynomial& } P, \text{ list<real>& roots, algorithm_type algorithm, bool is_squarefree})$$

returns all real roots of the polynomial $P$.

$$\text{real roots(}\text{const Polynomial& } P, \text{ growing_array<real>& roots, algorithm_type algorithm, bool is_squarefree})$$

same as above.

$$\text{real roots(}\text{const int_Polynomial& } iP, \text{ list<real>& roots, algorithm_type algorithm = isolating_algorithm, bool is_squarefree = true})$$

returns all real roots of the polynomial $iP$.

$$\text{real diamond(}\text{int } j, \text{ const Polynomial& } P, \text{ algorithm_type algorithm, bool is_squarefree})$$

returns the $j$-th smallest real root of the polynomial $P$.

$$\text{real diamond(}\text{rational } l, \text{ rational } u, \text{ const Polynomial& } P, \text{ algorithm_type algorithm, bool is_squarefree})$$

returns the real root of the polynomial $P$ which is in the isolating interval $[l,u]$.

$$\text{real diamond_short(}\text{rational } l, \text{ rational } u, \text{ const Polynomial& } P, \text{ algorithm_type algorithm, bool is_squarefree})$$

returns the real root of the polynomial $P$ which is in the isolating interval $[l,u]$.  
**Precondition:** $(u - l) < 1/4$

$$\text{real diamond(}\text{int } j, \text{ const int_Polynomial& } iP, \text{ algorithm_type algorithm = isolating_algorithm, bool is_squarefree = true})$$

returns the $j$-th smallest real root of the polynomial $iP$.

$$\text{real diamond(}\text{rational } l, \text{ rational } u, \text{ const int_Polynomial& } iP, \text{ algorithm_type algorithm = isolating_algorithm, bool is_squarefree = true})$$

returns the real root of the polynomial $iP$ which is in the isolating interval $[l,u]$.

$$\text{real abs(}\text{const real& } x)$$

absolute value of $x$
6.4. THE DATA TYPE REAL (REAL)

\begin{verbatim}
real  sqr(const real& x) square of x
real  dist(const real& x, const real& y)
      euclidean distance of point (x,y) to the origin
real  powi(const real& x, int n)
      x^n, i.e., n.th power of x
integer  floor(const real& x)
      returns the largest integer smaller than or equal to x.
integer  ceil(const real& x)
      returns the smallest integer greater than or equal to x.
rational  small_rational_between(const real& x, const real& y)
      returns a rational number between x and y with the smallest available denominator. Note that the denominator does not need to be strictly minimal over all possible rationals.
rational  small_rational_near(const real& x, double eps)
      returns small_rational_between(x - eps, x + eps).
\end{verbatim}

5. Implementation

A real is represented by the expression which defines it and an interval inclusion \( I \) that contains the exact value of the real. The arithmetic operators \(+, -, *, \sqrt{\ }\) take constant time. When the sign of a real number needs to be determined, the data type first computes a number \( q \), if not already given as an argument to \textit{sign}, such that \(|x| \leq 2^{-q}\) implies \( x = 0 \). The bound \( q \) is computed as described in [79]. Using \textit{bigfloat} arithmetic, the data type then computes an interval \( I \) of maximal length \( 2^{-q} \) that contains \( x \). If \( I \) contains zero, then \( x \) itself is equal to zero. Otherwise, the sign of any point in \( I \) is returned as the sign of \( x \).

Two shortcuts are used to speed up the computation of the sign. Firstly, if the initial \textit{interval} approximation already suffices to determine the sign, then no \textit{bigfloat} approximation is computed at all. Secondly, the \textit{bigfloat} approximation is first computed only with small precision. The precision is then roughly doubled until either the sign can be decided (i.e., if the current approximation interval does not contain zero) or the full precision \( 2^{-q} \) is reached. This procedure makes the sign computation of a real number \( x \) adaptive in the sense that the running time of the sign computation depends on the complexity of \( x \).

6. Example

We give two typical examples for the use of the data type real that arise in Computational geometry. We admit that a certain knowledge about Computational geometry is required for their full understanding. The examples deal with the Voronoi diagram of line segments and the intersection of line segments, respectively.
The following incircle test is used in the computation of Voronoi diagrams of line segments [17, 14]. For \( i, 1 \leq i \leq 3 \), let \( l_i : a_i x + b_i y + c_i = 0 \) be a line in two–dimensional space and let \( p = (0,0) \) be the origin. In general, there are two circles passing through \( p \) and touching \( l_1 \) and \( l_2 \). The centers of these circles have homogeneuos coordinates \((x_v, y_v, z_v)\), where

\[
\begin{align*}
x_v &= a_1 c_2 + a_2 c_1 \pm \text{sign}(s) \sqrt{2c_1 c_2(\sqrt{N} + D)} \\
y_v &= b_1 c_2 + b_2 c_1 \pm \text{sign}(r) \sqrt{2c_1 c_2(\sqrt{N} - D)} \\
z_v &= \sqrt{N} - a_1 a_2 - b_1 b_2
\end{align*}
\]

and

\[
\begin{align*}
s &= b_1 D_2 - b_2 D_1, \quad N &= (a_1^2 + b_1^2)(a_2^2 + b_2^2) \\
r &= a_1 D_2 - a_2 D_1, \quad D &= a_1 a_2 - b_1 b_2.
\end{align*}
\]

Let us concentrate on one of these (say, we take the plus sign in both cases). The test whether \( l_3 \) intersects, touches or misses the circle amounts to determining the sign of

\[
E := \text{dist}^2(v, l_3) - \text{dist}^2(v, p) = \frac{(a_3 x_v + b_3 y_v + c_3)^2}{a_3^2 + b_3^2} - (x_v^2 + y_v^2).
\]

The following program computes the sign of \( \tilde{E} := (a_3^2 + b_3^2) \cdot E \) using our data type real.

```c
int incircle( real a1, real b1, real c1, real a2, real b2, real c2, real a3, real b3, real c3 )
{
    real RN = sqrt((a1 * a1 + b1 * b1) * (a2 * a2 + b2 * b2));
    real RN1 = sqrt(a1 * a1 + b1 * b1);
    real RN2 = sqrt(a2 * a2 + b2 * b2);
    real A = a1 * c2 + a2 * c1;
    real B = b1 * c2 + b2 * c1;
    real C = 2 * c1 * c2;
    real D = a1 * a2 - b1 * b2;
    real s = b1 * RN2 - b2 * RN1;
    real r = a1 * RN2 - a2 * RN1;
    int signx = sign(s);
    int signy = sign(r);
    real xv = A + signx * sqrt(C * (RN + D));
    real yv = B - signy * sqrt(C * (RN - D));
    real zv = RN - (a1 * a2 + b1 * b2);
    real P = a3 * xv + b3 * yv + c3 * zv;
    real D3 = a3 * a3 + b3 * b3;
    real R2 = xv * xv + yv * yv;
    real E = P * P - D3 * R2;
    return sign(E);
}
```
6.4. THE DATA TYPE REAL (REAL)

We can make the above program more efficient if all coefficients \(a_i, b_i\) and \(c_i\), \(1 \leq i \leq 3\), are \(k\) bit integers, i.e., integers whose absolute value is bounded by \(2^k - 1\). In [17, 14] we showed that for \(\tilde{E} \neq 0\) we have \(|\tilde{E}| \geq 2^{-24k-26}\). Hence we may add a parameter \(\text{int } k\) in the above program and replace the last line by

\[
\text{return } E.\text{sign}(24 \ast k + 26).
\]

Without this assistance, \(\text{reals}\) automatically compute a weaker bound of \(|\tilde{E}| \geq 2^{-56k-161}\) for \(\tilde{E} \neq 0\) by [15].

We turn to the line segment intersection problem next. Assume that all endpoints have \(k\)-bit integer homogeneous coordinates. This implies that the intersection points have homogeneous coordinates \((X, Y, W)\) where \(X, Y\) and \(W\) are \((4k + 3)\) - bit integers. The Bentley–Ottmann plane sweep algorithm for segment intersection [65] needs to sort points by their \(x\)-coordinates, i.e., to compare fractions \(X_1/W_1\) and \(X_2/W_2\) where \(X_1, X_2, W_1, W_2\) are as above. This boils down to determining the sign of the \(8k + 7\) bit integer \(X_1 \ast W_2 - X_2 \ast W_1\). If all variables \(X_i, W_i\) are declared \(\text{real}\) then their sign test will be performed quite efficiently. First, an \(\text{interval}\) approximation is computed and then, if necessary, \(\text{bigfloat}\) approximations of increasing precision. In many cases, the \(\text{interval}\) approximation already determines the sign. In this way, the user of the data type \(\text{real}\) gets nearly the efficiency of a hand-coded floating point filter [35, 66] without any work on his side. This is in marked contrast to [35, 66] and will be incorporated into [65].
6.5 Interval Arithmetic in LEDA (interval)

1. Definition

An instance of the data type *interval* represents a real interval \( I = [a, b] \). The basic interval operations \(+, -, *, /, \sqrt{\cdot}\) are available. Type *interval* can be used to approximate exact real arithmetic operations by inexact interval operations, as follows. Each input number \( x_i \) is converted into the interval \( \{x_i\} \) and all real operations are replaced by interval operations. If \( x \) is the result of the exact real calculation and \( I \) the interval computed by type *interval*, it is guaranteed that \( I \) contains \( x \). \( I \) can be seen as a more or less accurate approximation of \( x \). In many cases the computed interval \( I \) is small enough to provide a useful approximation of \( x \) and the exact sign of \( x \).

There are four different implementations of *intervals* (consult the implementation section below for details):

- Class *interval_bound_absolute*
- Class *interval_bound_relative*
- Class *interval_round_inside*
- Class *interval_round_outside*, which is usually the fastest but requires that the IEEE754 rounding mode *ieee_positive* is activated, e.g. by using the LEDA class *fpu*.

The interface of all *interval* variants are identical. However, note that the types *interval_round_inside* and *interval_round_outside* are only available on some explicitly supported UNIX platforms, currently including SPARC, MIPS, i386 (PC’s compatible to 80386 or higher), and ALPHA. For all platforms, the name *interval* stands for the default implementation *interval_bound_absolute*.

```c
#include <LEDA/numbers/interval.h>

interval x;             // creates an instance x of type interval and initializes it with the interval \{0\}
interval x(VOLATILE_I double a);          // creates an instance x of type interval and initializes it with \{a\}
interval x(int a);            // creates an instance x of type interval and initializes it with \{a\}
interval x(long a);          // creates an instance x of type interval and initializes it with \{a\}
interval x(const integer& a);                        // creates an instance x of type interval and initializes it with the smallest possible interval containing a
```
interval \( x(const \text{bigfloat}\& a); \)

creates an instance \( x \) of type \textit{interval} and initializes it with the smallest possible interval containing \( a \)

interval \( x(const \text{real}\& a); \)

creates an instance \( x \) of type \textit{interval} and initializes it with the smallest possible interval containing \( a \)

interval \( x(const \text{rational}\& a); \)

creates an instance \( x \) of type \textit{interval} and initializes it with the smallest possible interval containing \( a \)

\section{Operations}

The arithmetic operations \(+, -, *, /, \text{sqrt}, ++, --, \ast, /=\) and the stream operators are all available. \textbf{Important:} If the advanced implementation \textit{interval\_round\_outside} is used, the user has to guarantee that for each \textit{interval} operation the IEEE754 rounding mode "towards \(+\infty\)" is active. This can be achieved by calling the function \textit{fpu::round\_up( )}. To avoid side effects with library functions that require the default IEEE754 rounding mode \textit{to\_nearest}, the function \textit{fpu::round\_nearest( )} can be used to reset the rounding mode.

\begin{itemize}
  \item \texttt{double \quad x.to\_double( )} \quad returns the midpoint of the interval \( x \) as an approximation for the exact real number represented by \( x \).
  \item \texttt{double \quad x.get\_double\_error( )} \quad returns the diameter of the interval \( x \) which is the maximal error of the approximation \( x.to\_double( ) \) of the exact real number represented by \( x \).
  \item \texttt{bool \quad x.is\_a\_point( )} \quad returns true if and only if the interval \( x \) consists of a single point.
  \item \texttt{bool \quad x.is\_finite( )} \quad returns true if and only if the interval \( x \) is a finite interval.
  \item \texttt{bool \quad x.contains(double \( x \))} \quad returns true if and only if the interval \( x \) contains the number \( x \)
  \item \texttt{double \quad x.upper\_bound( )} \quad returns the upper bound of the interval \( x \).
  \item \texttt{double \quad x.lower\_bound( )} \quad returns the lower bound of the interval \( x \).
  \item \texttt{void \quad x.set\_range(VOLATILE\_I double x, VOLATILE\_I double y)} \quad sets the current interval to \([x, y]\).  
\end{itemize}
void x.set_midpoint(VOLATILE_I double num, VOLATILE_I double error)
sets the current interval to a superset of \([num - error, num + error]\), i.e., to an interval with midpoint \(num\) and radius \(error\).

bool x.sign_is_known() returns true if and only if all numbers in the interval \(x\) have the same sign.

int x.sign() returns the sign of all numbers in the interval \(x\) if this sign is unique; aborts with an error message if \(x\).sign_is_known() gives false.

3. Implementation

The types \(\text{interval\_round\_inside}\) and \(\text{interval\_round\_outside}\) represent intervals directly by (the negative of) its lower bound and its upper bound as \(\text{doubles}\). Here all arithmetic operations require that the IEEE754 rounding mode "towards +\(\infty\)" is active. For type \(\text{interval\_round\_inside}\) this is done inside each operation, and for type \(\text{interval\_round\_outside}\) the user has to do this manually "from outside the operations" by an explicit call of \(\text{fpu\::round\_up()}\).

The types \(\text{interval\_bound\_absolute}\) and \(\text{interval\_bound\_relative}\) represent intervals by their \(\text{double}\) midpoint \(\text{NUM}\) and diameter \(\text{ERROR}\). The interpretation is that \(\text{NUM}\) is the numerical approximation of a real number and \(\text{ERROR}\) is a bound for the absolute, respectively relative error of \(\text{NUM}\).
6.6 Modular Arithmetic in LEDA ( residual )

1. Definition

The data type residual provides an implementation of exact integer arithmetic using modular computation. In contrast to the LEDA type integer which offers similar functionality as residual, the user of residual has to specify for each calculation the maximal bit length \( b \) of the integers she wants to be exactly representable by residuals. This is done by a call of \texttt{residual::set_maximal_bit_length(b)} preceding the calculation. The set of integers in the interval \([-2^b, 2^b]\) is called the current range of numbers representable by residuals.

A residual number \( x \) that is outside the current range is said to overflow. As an effect of its overflow, certain operations cannot be applied to \( x \) and the result is undefined. These critical operations include e.g. all kinds of conversion, sign testing and comparisons. It is important to realize that for an integer \( x \) given by a division-free expression it only matters whether the final result \( x \) does not overflow. This is sometimes useful and hence overflow is not always checked by default.

Division is available for residuals, but with certain restrictions. Namely, for each division \( x/y \) the user has to guarantee at least one of the following two conditions:

- \( y \) is invertible is true
- \( x/y \) is integral and \( x \) and \( y \) do not overflow.

If the first condition is satisfied, there is an alternative way to do the division \( x/y \). Introducing the residual variable \( z = y \) inverse(), the call \( x/y \) is equivalent to the call \( x \ast z \). The latter form is advantageous if several divisions have the same divisor \( y \) because here the time-consuming inversion of \( y \), which is implicit in the division \( x/y \), has to be performed only once.

If the result of an operation is not integral, the computation will usually proceed without warning. In such cases the computation produces a nonsensical result that is likely to overflow but otherwise is a perfect residual. However, the operations mentioned above check for overflow. Note that the implemented overflow checks are not rigorous, detecting invalidity only with empirically high probability. Overflow checking can be switched off by calling \texttt{set_maximal_bit_length} with a second, optional parameter \texttt{residual::no_overflow_check}.

\texttt{#include <LEDA/numbers/residual.h>
6.7 The mod kernel of type residual ( residual )

1. Definition

Type residual::mod provides the basic modular arithmetic modulo primes of maximal size $2^{26}$. Here numbers modulo the prime $p$ are represented by integral doubles in $[0, \cdots p-1]$. This type cannot be instantiated, so there are only static functions and no constructors. The following functions have the common precondition that $p$ is a prime between 2 and $2^{26}$.

```
#include <LEDA/numbers/residual.h>
```

2. Operations

```cpp
double residual::reduce_of_positive(double a, double p)
    returns $a$ modulo $p$ for nonnegative integral $0 \leq a < 2^{54}$

double residual::reduce(double a, double p)
    returns $a$ modulo $p$ for any integral $a$ with $|a| < 2^{54}$

double residual::add(double a, double b, double p)
    returns $(a + b) \mod p$ where $a$, $b$ are integral with $|a|, |b| < 2^{52}$

double residual::sub(double a, double b, double p)
    returns $(a - b) \mod p$ where $a$, $b$ are integral with $|a|, |b| < 2^{52}$

double residual::mul(double a, double b, double p)
    returns $(a \cdot b) \mod p$ where $a$, $b$ are integral with $|a \cdot b| < 2^{53}$

double residual::div(double a, double b, double p)
    returns $(a \cdot b^{-1}) \mod p$ where $a$, $b$ are integral with $|a| < 2^{26}$ and $b \neq 0 \mod p$

double residual::negate(double a, double p)
    returns $-a \mod p$ for nonnegative $a < p$

double residual::inverse(double a, double p)
    returns the inverse of $a$ modulo $p$ for integral $0 \leq a < p < 2^{32}$
```
6.8 The smod kernel of type residual (residual)

1. Definition

Type residual::smod is a variant of class residual::mod that uses a signed representation. Here numbers modulo $p$ are represented by integral doubles in $(-p/2, +p/2)$. All functions have the common precondition that $p$ is a prime between 3 and $2^{26}$. The functions of type residual::mod are also provided for class residual::smod and have the same meaning, so we do not list them separately here.

```
#include <LEDA/numbers/residual.h>
```

2. Operations

```
double residual::frac(double a)
```
returns $a + z$ where $z$ is the unique integer such that $a + z \in [-1/2, 1/2)$

3. Creation

```
residual x;
```
creates an instance $x$ of type residual and initializes it with zero.

```
residual x(long a);
```
creates an instance $x$ of type residual and initializes it with the value of $a$.

```
residual x(int a);
```
creates an instance $x$ of type residual and initializes it with the value of $a$.

```
residual x(double a);
```
creates an instance $x$ of type residual and initializes it with the integral part of $x$.

```
residual x(const integer& a);
```
creates an instance $x$ of type residual and initializes it with the value of $a$.

4. Operations

```
int residual::set_maximal_bit_length(int b, bool with_check = do_overflow_check)
```
sets the maximal bit size of the representable numbers to $b$ and returns the previous maximal bit size

```
int residual::get_maximal_bit_length()
```
returns the maximal bit size of the representable numbers
int residual::required_prime_table_size(int b)
returns the number of primes required to represent
signed numbers up to bit length b

The following functions have the common precondition that the residual objects a, x are integral and do not overflow.

integer x.to_integer() returns the integer equal to x.
long x.length() returns the length of the binary representation of
the integer represented by x.
bool x.is_long() returns true if and only if x fits in the data format long.
long x.to_long() returns a long number which is initialized with the
value of x. Precondition: x.is_long() is true.
double x.to_double() returns a double floating point approximation of x.
double x.to_float() as above.
bool x.is_zero() returns true if and only if x is equal to zero.
bool x.is_invertible() returns true if and only if x is nonzero and the cur-
rent modular representation of x allows to invert x
without loss of information.
int x.sign() returns the sign of x.
int x.lagrange_sign() returns the sign of x using Lagrange’s formula.
int x.garner_sign() returns the sign of x using Garner’s formula.
string x.to_string() returns the decimal representation of x.
residual abs(const residual & a) returns the absolute value of a
void x.absolute(const residual & a)
sets x to the absolute value of a.

The remaining functions do not have implicit preconditions. Although not explicitly
mentioned, the arithmetic operations +, −, *, /, +=, −=, *=, /=, ++, −−, the shift
operations, the comparison operations <, ≤, >, ≥, ==, != and the stream operations are
available.
residual sqr(const residual & a) returns a * a
residual det2x2(const residual & a, const residual & b, const residual & c,
const residual & d) returns a * d − b * c
6.8. THE SMOD KERNEL OF TYPE RESIDUAL ( RESIDUAL )

void x.add(const residual& a, const residual& b) sets x to \( a + b \).

void x.sub(const residual& a, const residual& b) sets x to \( a - b \).

void x.mul(const residual& a, const residual& b) sets x to \( a \ast b \).

void x.div(const residual& a, const residual& b) sets x to \( a/b \).

void x.det2x2(const residual& a, const residual& b, const residual& c, const residual& d) sets x to \( a \ast d - b \ast c \).

void x.inverse(const residual& a) sets x to the modular inverse of a. Precondition: x.in_invertible is true.

void x.negate(const residual& a) sets x to \(-a\).

The following functions provide direct read-only access to the internal representation of residual objects. They should only be used by the experienced user after reading the full documentation of type residual.

residual_sequence residual::get_primetab( ) returns a copy of the currently used primetable

residual_sequence residual::get_garnertab( ) returns a copy of the currently used table of Garner’s constants

residual_sequence get_representation( ) returns a copy of the residual sequence representing x
6.9 A Floating Point Filter (floatf)

1. Definition

The type floatf provides a clean and efficient way to approximately compute with large integers. Consider an expression $E$ with integer operands and operators $+,-, \text{ and } \ast$, and suppose that we want to determine the sign of $E$. In general, the integer arithmetic provided by our machines does not suffice to evaluate $E$ since intermediate results might overflow. Resorting to arbitrary precision integer arithmetic is a costly process. An alternative is to evaluate the expression using floating point arithmetic, i.e., to convert the operands to doubles and to use floating-point addition, subtraction, and multiplication.

Of course, only an approximation $E'$ of the true value $E$ is computed. However, $E'$ might still be able to tell us something about the sign of $E$. If $E'$ is far away from zero (the forward error analysis carried out in the next section gives a precise meaning to “far away”) then the signs of $E'$ and $E$ agree and if $E'$ is zero then we may be able to conclude under certain circumstances that $E$ is zero. Again, forward error analysis can be used to say what ‘certain circumstances’ are.

The type floatf encapsulates this kind of approximate integer arithmetic. Any integer (= object of type integer) can be converted to a floatf; floatfs can be added, subtracted, multiplied, and their sign can be computed: for any floatf $x$ the function $\text{Sign}(x)$ returns either the sign of $x$ ($-1$ if $x < 0$, $0$ if $x = 0$, and $+1$ if $x > 0$) or the special value $\text{NO\_IDEA}$. If $x$ approximates $X$, i.e., $X$ is the integer value obtained by an exact computation, then $\text{Sign}(x)! = \text{NO\_IDEA}$ implies that $\text{Sign}(x)$ is actually the sign of $X$ if $\text{Sign}(x) = \text{NO\_IDEA}$ then no claim is made about the sign of $X$.

#include <LEDA/numbers/floatf.h>

2. Creation

floatf $x$; introduces a variable $x$ of type floatf and initializes it with zero.

floatf $x($integer $i$); introduces a variable $x$ of type floatf and initializes it with integer $i$.

3. Operations

floatf $\text{const floatf} \& a + \text{const floatf} \& b$

Addition.

floatf $\text{const floatf} \& a - \text{const floatf} \& b$

Subtraction.

floatf $\text{const floatf} \& a \ast \text{const floatf} \& b$

Multiplication.
int Sign(const float& f)

as described above.

4. Implementation

A floatf is represented by a double (its value) and an error bound. An operation on floatfs performs the corresponding operation on the values and also computes the error bound for the result. For this reason the cost of a floatf operation is about four times the cost of the corresponding operation on doubles. The rules used to compute the error bounds are described in ([65]).

5. Example

see [65] for an application in a sweep line algorithm.
6.10 Double-Valued Vectors (vector)

1. Definition

An instance of data type vector is a vector of variables of type double.

```c
#include <LEDA/numbers/vector.h>
```

2. Creation

- `vector v;` creates an instance `v` of type `vector`; `v` is initialized to the zero-dimensional vector.
- `vector v(int d);` creates an instance `v` of type `vector`; `v` is initialized to the zero vector of dimension `d`.
- `vector v(double a, double b);` creates an instance `v` of type `vector`; `v` is initialized to the two-dimensional vector `(a, b)`.
- `vector v(double a, double b, double c);` creates an instance `v` of type `vector`; `v` is initialized to the three-dimensional vector `(a, b, c)`.
- `vector v(const vector& w, int prec);` creates an instance `v` of type `vector`; `v` is initialized to a copy of `w`. The second argument is for compatibility with `rat_vector`.

3. Operations

- `int v.dim()` returns the dimension of `v`.
- `double& v[int i]` returns `i`-th component of `v`. 
  **Precondition:** `0 ≤ i ≤ v.dim()−1`.
- `double v.hcoord(int i)` for compatibility with `rat_vector`.
- `double v.coord(int i)` for compatibility with `rat_vector`.
- `double v.sqr_length()` returns the square of the Euclidean length of `v`.
- `double v.length()` returns the Euclidean length of `v`.
- `vector v.norm()` returns `v` normalized.
- `double v.angle(const vector& w)` returns the angle between `v` and `w`. 
vector \ v.\rotate90(\text{int } i = 1) \quad \text{returns } v \text{ by an angle of } i \times 90 \text{ degrees. If } i > 0 \text{ the rotation is counter-clockwise otherwise it is clockwise. } \textbf{Precondition: } v.\text{dim}() = 2

vector \ v.\rotate(\text{double } a) \quad \text{returns the } v \text{ rotated counterclockwise by an angle of } a \text{ (in radian).} \quad \textbf{Precondition: } v.\text{dim}() = 2

\begin{align*}
\text{vector}& \quad v += \text{ const vector}& v1 \quad \text{Addition and assign.} \\
\text{vector}& \quad v -= \text{ const vector}& v1 \quad \text{Subtraction and assign.} \\
\text{vector} \quad v + \text{ const vector}& v1 \quad \text{Addition.} \\
\text{vector} \quad v - \text{ const vector}& v1 \quad \text{Subtraction.} \\
\text{double} \quad v * \text{ const vector}& v1 \quad \text{Scalar multiplication.} \\
\text{vector} \quad v * \text{ double } r \quad \text{Componentwise multiplication with double } r. \\
\text{vector}& \quad v *= \text{ double } r \quad \text{multiplies all coordinates by } r. \\
\text{vector} \quad v / \text{ double } r \quad \text{Componentwise division which double } r. \\
\text{bool} \quad v == \text{ const vector}& w \quad \text{Test for equality.} \\
\text{bool} \quad v != \text{ const vector}& w \quad \text{Test for inequality.} \\
\text{void} \quad v.\text{print}(\text{ostream}& O) \quad \text{prints } v \text{ componentwise to ostream } O. \\
\text{void} \quad v.\text{print}( ) \quad \text{prints } v \text{ to } cout. \\
\text{void} \quad v.\text{read}(\text{istream}& I) \quad \text{reads } d = v.\text{dim}( ) \text{ numbers from input stream } I \text{ and writes them into } v[0] \ldots v[d - 1]. \\
\text{void} \quad v.\text{read}( ) \quad \text{reads } v \text{ from } cin. \\
\text{ostream}& \quad \text{ostream}& O \ll \text{ const vector}& v \quad \text{writes } v \text{ componentwise to the output stream } O. \\
\text{istream}& \quad \text{istream}& I \gg \text{ vector}& v \quad \text{reads } v \text{ componentwise from the input stream } I.
\end{align*}

\textbf{Additional Operations for vectors in two and three-dimensional space}

\begin{align*}
\text{double} \quad v.\text{xcoord}( ) \quad \text{returns the zero-th cartesian coordinate of } v.
\end{align*}
**double** \( v.ycoord() \) returns the first cartesian coordinate of \( v \).

**double** \( v.zcoord() \) returns the second cartesian coordinate of \( v \).

**int** \( \text{compare\_by\_angle(const vector& v1, const vector& v2)} \)

For a non-zero vector \( v \) let \( \alpha(v) \) be the angle by which the positive \( x \)-axis has to be turned counterclockwise until it aligns with \( v \). The function compares the angles defined by \( v1 \) and \( v2 \), respectively. The zero-vector precedes all non-zero vectors in the angle-order.

**vector** \( \text{cross\_product(const vector& v1, const vector& v2)} \)

returns the cross product of the three-dimensional vectors \( v1 \) and \( v2 \).

### 4. Implementation

Vectors are implemented by arrays of real numbers. All operations on a vector \( v \) take time \( O(v.\text{dim}()) \), except for \( \text{dim} \) and \( [\cdot] \) which take constant time. The space requirement is \( O(v.\text{dim}()) \).

Be aware that the operations on vectors and matrices incur rounding errors and hence are not completely reliable. For example, if \( M \) is a matrix, \( b \) is a vector, and \( x \) is computed by \( x = M.\text{solve}(b) \) it is not necessarily true that the test \( b == M * x \) evaluates to true. The types \text{integer\_vector} and \text{integer\_matrix} provide exact linear algebra.
6.11 Double-Valued Matrices (matrix)

1. Definition

An instance of the data type matrix is a matrix of variables of type double.

```c
#include <LEDA/numbers/matrix.h>
```

2. Creation

```c
matrix M(int n = 0, int m = 0);
```

creates an instance \( M \) of type matrix, \( M \) is initialized to the \( n \times m \)-zero matrix.

```c
matrix M(int n, int m, double *D);
```

creates the \( n \times m \) matrix \( M \) with \( M(i, j) = D[i \times m + j] \) for \( 0 \leq i \leq n - 1 \) and \( 0 \leq j \leq m - 1 \). **Precondition**: \( D \) points to an array of at least \( n \times m \) numbers of type double.

3. Operations

```c
int M.dim1() returns n, the number of rows of M.
int M.dim2() returns m, the number of columns of M.
vector& M.row(int i) returns the i-th row of M (an m-vector).
    Precondition: 0 \leq i \leq n - 1.
vector M.col(int i) returns the i-th column of M (an n-vector).
    Precondition: 0 \leq i \leq m - 1.
matrix M.trans() returns \( M^T \) (\( m \times n \) - matrix).
matrix M.inv() returns the inverse matrix of M.
    Precondition: M is quadratic and M.det() \neq 0.
double M.det() returns the determinant of M.
    Precondition: M is quadratic.
vector M.solve(const vector& b) returns vector \( x \) with \( M \cdot x = b \).
    Precondition: M.dim1() == M.dim2() = = b.dim() and M.det() \neq 0.
double& M(int i, int j) returns \( M_{i,j} \).
    Precondition: 0 \leq i \leq n - 1 and 0 \leq j \leq m - 1.
```
matrix \( M + \text{const matrix} \& M1 \)
Addition.
Precondition: \( M.\text{dim1}() == M1.\text{dim1}() \) and \( M.\text{dim2}() == M1.\text{dim2}() \).

matrix \( M - \text{const matrix} \& M1 \)
Subtraction.
Precondition: \( M.\text{dim1}() == M1.\text{dim1}() \) and \( M.\text{dim2}() == M1.\text{dim2}() \).

matrix \( M \times \text{const matrix} \& M1 \)
Multiplication.
Precondition: \( M.\text{dim2}() == M1.\text{dim1}() \).

vector \( M \times \text{const vector} \& \text{vec} \)
Multiplication with vector.
Precondition: \( M.\text{dim2}() == \text{vec}.\text{dim}() \).

matrix \( M \times \text{double } x \)
Multiplication with double x.

void \( M.\text{print}(\text{ostream} \& O) \)
prints \( M \) row by row to ostream \( O \).

void \( M.\text{print}( ) \)
prints \( M \) cout.

void \( M.\text{read}(\text{istream} \& I) \)
reads \( M.\text{dim1}( ) \times M.\text{dim2}( ) \) numbers from input stream \( I \) and writes them row by row into matrix \( M \).

void \( M.\text{read}( ) \)
prints \( M \) from cin.

ostream\& \( \text{ostream} \& O \ll \text{const matrix} \& M \)
writes matrix \( M \) row by row to the output stream \( O \).

istream\& \( \text{istream} \& I \gg \text{matrix} \& M \)
reads a matrix row by row from the input stream \( I \) and assigns it to \( M \).

4. Implementation

Data type \textit{matrix} is implemented by two-dimensional arrays of double numbers. Operations \textit{det}, \textit{solve}, and \textit{inv} take time \( O(n^3) \), \textit{dim1}, \textit{dim2}, \textit{row}, and \textit{col} take constant time, all other operations take time \( O(nm) \). The space requirement is \( O(nm) \).

Be aware that the operations on vectors and matrices incur rounding error and hence are not completely reliable. For example, if \( M \) is a matrix, \( b \) is a vector, and \( x \) is computed
by \( x = M.\text{solve}(b) \) it is not necessarily true that the test \( b == M \ast b \) evaluates to true. The types \texttt{integer\_vector} and \texttt{integer\_matrix} provide exact linear algebra.
6.12 Vectors with Integer Entries (integer_vector)

1. Definition

An instance of data type integer_vector is a vector of variables of type integer, the so called ring type. Together with the type integer_matrix it realizes the basic operations of linear algebra. Internal correctness tests are executed if compiled with the flag LA_SELFTEST.

#include <LEDA/numbers/integer_vector.h>

2. Creation

integer_vector v; creates an instance $v$ of type integer_vector. $v$ is initialized to the zero-dimensional vector.

integer_vector v(int d); creates an instance $v$ of type integer_vector. $v$ is initialized to a vector of dimension $d$.

integer_vector v(const integer& a, const integer& b); creates an instance $v$ of type integer_vector. $v$ is initialized to the two-dimensional vector $(a, b)$.

integer_vector v(const integer& a, const integer& b, const integer& c); creates an instance $v$ of type integer_vector. $v$ is initialized to the three-dimensional vector $(a, b, c)$.

integer_vector v(const integer& a, const integer& b, const integer& c, const integer& d); creates an instance $v$ of type integer_vector; $v$ is initialized to the four-dimensional vector $(a, b, c, d)$.

3. Operations

int $v$.dim() returns the dimension of $v$.

integer& $v$[int $i$] returns $i$-th component of $v$.  
Precondition: $0 \leq i \leq v$.dim( ) - 1.

integer_vector& $v$ += const integer_vector& $v1$ 
Addition plus assignment.  
Precondition: $v$.dim( ) == $v1$.dim( ).

integer_vector& $v$ -= const integer_vector& $v1$ 
Subtraction plus assignment.  
Precondition: $v$.dim( ) == $v1$.dim( ).
integer_vector $v + \text{const integer_vector} & v1$

Addition.
Precondition: $v.\text{dim}( ) == v1.\text{dim}( )$.

integer_vector $v - \text{const integer_vector} & v1$

Subtraction.
Precondition: $v.\text{dim}( ) == v1.\text{dim}( )$.

integer $v * \text{const integer_vector} & v1$

Inner Product.
Precondition: $v.\text{dim}( ) == v1.\text{dim}( )$.

integer_vector $\text{const integer} & r * \text{const integer_vector} & v$

Componentwise multiplication with number $r$.

integer_vector $\text{const integer_vector} & v * \text{const integer} & r$

Componentwise multiplication with number $r$.

ostream& ostream& $O \ll \text{const integer_vector} & v$

writes $v$ componentwise to the output stream $O$.

istream& istream& $I \gg \text{integer_vector} & v$

reads $v$ componentwise from the input stream $I$.

4. Implementation

Vectors are implemented by arrays of type integer. All operations on a vector $v$ take time $O(v.\text{dim}( ))$, except for dimension and [ ] which take constant time. The space requirement is $O(v.\text{dim}( ))$. 
6.13 Matrices with Integer Entries (integer_matrix)

1. Definition

An instance of data type integer_matrix is a matrix of variables of type integer, the so-called ring type. The arithmetic type integer is required to behave like integers in the mathematical sense.

The types integer_matrix and integer_vector together realize many functions of basic linear algebra. All functions on integer matrices compute the exact result, i.e., there is no rounding error. Most functions of linear algebra are checkable, i.e., the programs can be asked for a proof that their output is correct. For example, if the linear system solver declares a linear system $Ax = b$ unsolvable it also returns a vector $c$ such that $c^T A = 0$ and $c^T b \neq 0$. All internal correctness checks can be switched on by the flag LA_SELFTEST. Preconditions are checked by default and can be switched off by the compile flag LEDA_CHECKING_OFF.

```c
#include <LEDA/numbers/integer_matrix.h>
```

2. Creation

```c
integer_matrix M(int n, int m);
```
creates an instance $M$ of type integer_matrix of dimension $n \times m$.

```c
integer_matrix M(int n = 0);
```
creates an instance $M$ of type integer_matrix of dimension $n \times n$.

```c
integer_matrix M(const array<integer_vector> &A);
```
creates an instance $M$ of type integer_matrix. Let $A$ be an array of $m$ column-vectors of common dimension $n$. $M$ is initialized to an $n \times m$ matrix with the columns as specified by $A$.

```c
integer_matrix integer_matrix::identity(int n)
```
returns an identity matrix of dimension $n$.

3. Operations

```c
int M.dim1() returns $n$, the number of rows of $M$.
```

```c
int M.dim2() returns $m$, the number of columns of $M$.
```

```c
integer_vector& M.row(int i) returns the $i$-th row of $M$ (an $m$-vector).
```
**Precondition:** $0 \leq i \leq n - 1$. 

integer_vector \hspace{1em} M\.col\(\text{int } i\) returns the \(i\)-th column of \(M\) (an \(n\) - vector).  
\textit{Precondition:} \(0 \leq i \leq m - 1\).

integer& \hspace{1em} M\(\text{int } i, \text{int } j\) returns \(M_{i,j}\).  
\textit{Precondition:} \(0 \leq i \leq n - 1\) and \(0 \leq j \leq m - 1\).

**Arithmetic Operators**

integer_matrix \hspace{1em} M + \text{const integer_matrix} & M1  
\text{Addition.}  
\textit{Precondition:} \(M\.\text{dim1()} == M1\.\text{dim1()}\) and \(M\.\text{dim2()} == M1\.\text{dim2()}\).

integer_matrix \hspace{1em} M − \text{const integer_matrix} & M1  
\text{Subtraction.}  
\textit{Precondition:} \(M\.\text{dim1()} == M1\.\text{dim1()}\) and \(M\.\text{dim2()} == M1\.\text{dim2()}\).

integer_matrix \hspace{1em} M * \text{const integer_matrix} & M1  
\text{Multiplication.}  
\textit{Precondition:} \(M\.\text{dim2()} == M1\.\text{dim1()}\).

integer_vector \hspace{1em} M * \text{const integer_vector} & vec  
\text{Multiplication with vector.}  
\textit{Precondition:} \(M\.\text{dim2()} == \text{vec}\.\text{dim()}\).

integer_matrix \hspace{1em} \text{const integer_matrix} & M * \text{const integer} & x  
\text{Multiplication of every entry with integer } x.

integer_matrix \hspace{1em} \text{const integer} & x * \text{const integer_matrix} & M  
\text{Multiplication of every entry with integer } x.

**Non-Member Functions**

integer_matrix \hspace{1em} transpose(const integer_matrix & M)  
returns \(M^T\) (\(m \times n\) - matrix).
integer_matrix inverse(const integer_matrix& M, integer& D)
returns the inverse matrix of \( M \). More precisely, 
\( \frac{1}{D} \) times the matrix returned is the inverse of 
\( M \).

Precondition: determinant\( (M) \neq 0 \).

bool inverse(const integer_matrix& M, integer_matrix& inverse,
integer& D, integer_vector& c)
determines whether \( M \) has an inverse. It also computes either 
the inverse as \( \left( \frac{1}{D} \right) \cdot \text{inverse} \) or a vector \( c \) such that \( c^T \cdot M = 0 \).

integer determinant(const integer_matrix& M, integer_matrix& L,
integer_matrix& U, array<int>& q, integer_vector& c)
returns the determinant \( D \) of \( M \) and sufficient information to 
verify that the value of the determinant is correct. If the de-
terminant is zero then \( c \) is a vector such that \( c^T \cdot M = 0 \). If 
the determinant is non-zero then \( L \) and \( U \) are lower and upper 
diagonal matrices respectively and \( q \) encodes a permutation ma-
trix \( Q \) with \( Q(i,j) = 1 \) iff \( i = q(j) \) such that \( L \cdot M \cdot Q = U, 
L(0,0) = 1, L(i,i) = U(i-1,i-1) \) for all \( i, 1 \leq i < n \), and 
\( D = s \cdot U(n-1,n-1) \) where \( s \) is the determinant of \( Q \).

Precondition: \( M \) is quadratic.

bool verify_determinant(const integer_matrix& M, integer D,
integer_matrix& L, integer_matrix& U,
array<int> q, integer_vector& c)
verifies the conditions stated above.

integer determinant(const integer_matrix& M)
returns the determinant of \( M \).

Precondition: \( M \) is quadratic.

int sign_of_determinant(const integer_matrix& M)
returns the sign of the determinant of \( M \).

Precondition: \( M \) is quadratic.

bool linear_solver(const integer_matrix& M, const integer_vector& b,
integer_vector& x, integer& D,
integer_matrix& spanning_vectors, integer_vector& c)
determines the complete solution space of the linear system \( M \cdot x = b \). If the system is unsolvable then \( c^T \cdot M = 0 \) and \( c^T \cdot b \neq 0 \). If 
the system is solvable then \( \left( \frac{1}{D} \right) \) \( x \) is a solution, and the columns 
of \( \text{spanning_vectors} \) are a maximal set of linearly independent 
solutions to the corresponding homogeneous system.

Precondition: \( M \cdot \text{dim1}() == b \cdot \text{dim}() \).
6.13. MATRICES WITH INTEGER ENTRIES (INTEGER_MATRIX)

bool linear_solver(const integer_matrix& M, const integer_vector& b, integer_vector& x, integer& D, integer_vector& c)

determines whether the linear system $M \cdot x = b$ is solvable. If yes, then $(1/D)x$ is a solution, if not then $c^T \cdot M = 0$ and $c^T \cdot b \neq 0$.
Precondition: $M$.dim1() == b.dim().

bool linear_solver(const integer_matrix& M, const integer_vector& b, integer_vector& x, integer& D)

as above, but without the witness $c$
Precondition: $M$.dim1() == b.dim().

bool is_solvable(const integer_matrix& M, const integer_vector& b)

determines whether the system $M \cdot x = b$ is solvable
Precondition: $M$.dim1() == b.dim().

bool homogeneous_linear_solver(const integer_matrix& M, integer_vector& x)

determines whether the homogeneous linear system $M \cdot x = 0$ has a non-trivial solution. If yes, then $x$ is such a solution.

int homogeneous_linear_solver(const integer_matrix& M, integer_matrix& spanning_vecs)

determines the solution space of the homogeneous linear system $M \cdot x = 0$. It returns the dimension of the solution space. Moreover the columns of spanning_vecs span the solution space.

void independent_columns(const integer_matrix& M, array<int>& columns)

returns the indices of a maximal subset of independent columns of $M$. The index range of columns starts at 0.

int rank(const integer_matrix& M)

returns the rank of matrix $M$

ostream& ostream& O << const integer_matrix& M

writes matrix $M$ row by row to the output stream $O$.

istream& istream& I >> integer_matrix& M

reads matrix $M$ row by row from the input stream $I$.

4. Implementation

The datatype integer_matrix is implemented by two-dimensional arrays of variables of type integer. Operations determinant, inverse, linear_solver, and rank take time $O(n^3)$, column takes time $O(n)$, row, dim1, dim2, take constant time, and all other operations take time $O(nm)$. The space requirement is $O(nm)$.

All functions on integer matrices compute the exact result, i.e., there is no rounding error. The implementation follows a proposal of J. Edmonds (J. Edmonds, Systems of
distinct representatives and linear algebra, Journal of Research of the Bureau of National Standards, (B), 71, 241 - 245). Most functions of linear algebra are checkable, i.e., the programs can be asked for a proof that their output is correct. For example, if the linear system solver declares a linear system $Ax = b$ unsolvable it also returns a vector $c$ such that $c^T A = 0$ and $c^T b \neq 0$. 
6.14 Rational Vectors (rat_vector)

1. Definition

An instance of data type rat_vector is a vector of rational numbers. A $d$-dimensional vector $\mathbf{r} = (r_0, \ldots, r_{d-1})$ is represented in homogeneous coordinates $(h_0, \ldots, h_d)$, where $r_i = h_i / h_d$ and the $h_i$'s are of type integer. We call the $r_i$'s the cartesian coordinates of the vector. The homogenizing coordinate $h_d$ is positive.

This data type is meant for use in computational geometry. It realizes free vectors as opposed to position vectors (type rat_point). The main difference between position vectors and free vectors is their behavior under affine transformations, e.g., free vectors are invariant under translations.

rat_vector is an item type.

#include <LEDA/numbers/rat_vector.h>

2. Creation

rat_vector $v(int d = 2)$; introduces a variable $v$ of type rat_vector initialized to the zero vector of dimension $d$.

rat_vector $v(integer a, integer b, integer D)$; introduces a variable $v$ of type rat_vector initialized to the two-dimensional vector with homogeneous representation $(a, b, D)$ if $D$ is positive and representation $(-a, -b, -D)$ if $D$ is negative.
Precondition: $D$ is non-zero.

rat_vector $v(rational x, rational y)$; introduces a variable $v$ of type rat_vector initialized to the two-dimensional vector with homogeneous representation $(a, b, D)$, where $x = a/D$ and $y = b/D$.

rat_vector $v(integer a, integer b, integer c, integer D)$; introduces a variable $v$ of type rat_vector initialized to the three-dimensional vector with homogeneous representation $(a, b, c, D)$ if $D$ is positive and representation $(-a, -b, -c, -D)$ if $D$ is negative.
Precondition: $D$ is non-zero.

rat_vector $v(rational x, rational y, rational z)$; introduces a variable $v$ of type rat_vector initialized to the three-dimensional vector with homogeneous representation $(a, b, c, D)$, where $x = a/D$, $y = b/D$ and $z = c/D$. 
rat_vector v(const array<rational>& A);

introduces a variable v of type rat_vector initialized to the d-dimensional vector with homogeneous coordinates $(\pm c_0, \ldots, \pm c_{d-1}, \pm D)$, where $d = A.size()$ and $A[i] = c_i/D$, for $i = 0, \ldots, d - 1$.

rat_vector v(integer a, integer b);

introduces a variable v of type rat_vector initialized to the two-dimensional vector with homogeneous representation $(a, b, 1)$.

rat_vector v(const integer_vector& c, integer D);

introduces a variable v of type rat_vector initialized to the vector with homogeneous coordinates $(\pm c_0, \ldots, \pm c_{d-1}, \pm D)$, where $d$ is the dimension of c and the sign chosen is the sign of D.

Precondition: D is non-zero.

rat_vector v(const integer_vector& c);

introduces a variable v of type rat_vector initialized to the direction with homogeneous coordinate vector $\pm c$, where the sign chosen is the sign of the last component of c.

Precondition: The last component of c is non-zero.

rat_vector v(const vector& w, int prec);

introduces a variable v of type rat_vector initialized to $(\lfloor P*w_0\rfloor, \ldots, \lfloor P*w_{d-1}\rfloor, P)$, where $d$ is the dimension of w and $P = 2^{\text{prec}}$.

3. Operations

3.1 Initialization, Access and Conversions

rat_vector rat_vector::d2(integer a, integer b, integer D)

returns a rat_vector of dimension 2 initialized to a vector with homogeneous representation $(a, b, D)$ if $D$ is positive and representation $(-a, -b, -D)$ if $D$ is negative.

Precondition: D is non-zero.
rat_vector :: d3(integer a, integer b, integer c, integer D)
returns a rat_vector of dimension 3 initialized to a vector with homogeneous representation 
\((a, b, c, D)\) if \(D\) is positive and representation 
\((-a, -b, -c, -D)\) if \(D\) is negative.
Precondition: \(D\) is non-zero.

rat_vector :: unit(int i, int d = 2)
returns a rat_vector of dimension \(d\) initialized to the \(i\)-th unit vector.
Precondition: \(0 \leq i < d\).

rat_vector :: zero(int d = 2) returns the zero vector in \(d\)-dimensional space.

int v.dim() returns the dimension of \(v\).

integer v.hcoord(int i) returns the \(i\)-th homogeneous coordinate of \(v\).

rational v.coord(int i) returns the \(i\)-th cartesian coordinate of \(v\).

rational v[int i] returns the \(i\)-th cartesian coordinate of \(v\).

rational v.sqr_length() returns the square of the length of \(v\).

vector v.to_float() returns a floating point approximation of \(v\).

Additional Operations for vectors in two and three-dimensional space

rational v.xcoord() returns the zero-th cartesian coordinate of \(v\).

rational v.ycoord() returns the first cartesian coordinate of \(v\).

rational v.zcoord() returns the second cartesian coordinate of \(v\).

integer v.X() returns the zero-th homogeneous coordinate of \(v\).

integer v.Y() returns the first homogeneous coordinate of \(v\).

integer v.Z() returns the second homogeneous coordinate of \(v\).

integer v.W() returns the homogenizing coordinate of \(v\).

rat_vector v.rotate90(int i = 1) returns \(v\) by an angle of \(i \times 90\) degrees. If \(i > 0\) the rotation is counter-clockwise otherwise it is clockwise. Precondition: \(v\).dim() == 2.
For a non-zero vector $v$ let $\alpha(v)$ be the angle by which the positive $x$-axis has to be turned counter-clockwise until it aligns with $v$. The function compares the angles defined by $v_1$ and $v_2$, respectively. The zero-vector precedes all non-zero vectors in the angle-order.

returns the cross product of the three-dimensional vectors $v_1$ and $v_2$.

Test for equality.

Test for inequality.

multiplies all cartesian coordinates by $n$.

multiplies all cartesian coordinates by $r$.

multiplies all cartesian coordinates by $n$.

multiplies all cartesian coordinates by $r$.

divides all cartesian coordinates by $n$.

divides all cartesian coordinates by $r$.

divides all cartesian coordinates by $n$.

divides all cartesian coordinates by $r$.

scalar product, i.e., $\sum_{0 \leq i < d} v_iw_i$, where $v_i$ and $w_i$ are the cartesian coordinates of $v$ and $w$ respectively.

adds cartesian coordinates.
6.14. RATIONAL VECTORS (RAT_VECTOR)

\(\text{rat\_vector}& \quad v += \text{const rat\_vector}& w\) addition plus assignment.

\(\text{rat\_vector} \quad \text{const rat\_vector}& v = \text{const rat\_vector}& w\)
subtracts cartesian coordinates.

\(\text{rat\_vector}& \quad v -= \text{const rat\_vector}& w\) subtraction plus assignment.

\(\text{rat\_vector} \quad -v\) returns \(-v\).

3.4 Input and Output

\(\text{ostream}& \quad \text{ostream}& O \ll \text{const rat\_vector}& v\)
writes \(v\)'s homogeneous coordinates componentwise to the output stream \(O\).

\(\text{istream}& \quad \text{istream}& I \gg \text{rat\_vector}& v\)
reads \(v\)'s homogeneous coordinates componentwise from the input stream \(I\). The operator uses the current dimension of \(v\).

3.5 Linear Hull, Dependence and Rank

\(\text{bool} \quad \text{contained\_in\_linear\_hull}(\text{const array<rat\_vector>>& A,\)
\text{const rat\_vector}& x)\)
determines whether \(x\) is contained in the linear hull of the vectors in \(A\).

\(\text{int} \quad \text{linear\_rank}(\text{const array<rat\_vector>>& A)\)
computes the linear rank of the vectors in \(A\).

\(\text{bool} \quad \text{linearly\_independent}(\text{const array<rat\_vector>>& A)\)
decides whether the vectors in \(A\) are linearly independent.

\(\text{array<rat\_vector>} \quad \text{linear\_base}(\text{const array<rat\_vector>>& A)\)
computes a basis of the linear space spanned by the vectors in \(A\).

4. Implementation

Vectors are implemented by arrays of integers as an item type. All operations like creation, initialization, tests, vector arithmetic, input and output on an vector \(v\) take time \(O(v.\text{dim}( ))\). \(\text{dim}( )\), coordinate access and conversions take constant time. The operations for linear hull, rank and independence have the cubic costs of the used matrix operations. The space requirement is \(O(v.\text{dim}( ))\).
6.15 Real-Valued Vectors ( real_vector )

1. Definition

An instance of data type real_vector is a vector of variables of type real.

#include <LEDA/numbers/real_vector.h>

2. Creation

real_vector v; creates an instance v of type real_vector; v is initialized to the zero-dimensional vector.

real_vector v(int d); creates an instance v of type real_vector; v is initialized to the zero vector of dimension d.

real_vector v(real a, real b);
        creates an instance v of type real_vector; v is initialized to the two-dimensional vector (a, b).

real_vector v(real a, real b, real c);
        creates an instance v of type real_vector; v is initialized to the three-dimensional vector (a, b, c).

real_vector v(double a, double b);
        creates an instance v of type real_vector; v is initialized to the two-dimensional vector (a, b).

real_vector v(double a, double b, double c);
        creates an instance v of type real_vector; v is initialized to the three-dimensional vector (a, b, c).

3. Operations

int v.dim( ) returns the dimension of v.

real& v[int i] returns i-th component of v.
        Precondition: 0 ≤ i ≤ v.dim()−1.

real v.hcoord(int i) for compatibility with rat_vector.

real v.coord(int i) for compatibility with rat_vector.

real v.sqr_length( ) returns the square of the Euclidean length of v.

real v.length( ) returns the Euclidean length of v.
6.15. REAL-VALUED VECTORS (REAL_VECTOR)

`real_vector v.norm()` returns $v$ normalized.

`real_vector v.rotate90(int i = 1)` returns $v$ by an angle of $i \times 90$ degrees. If $i > 0$ the rotation is counter-clockwise otherwise it is clockwise. **Precondition:** $v$.dim() = 2

`real_vector v + const real_vector& v1` Addition. **Precondition:** $v$.dim() = $v1$.dim().

`real_vector v - const real_vector& v1` Subtraction. **Precondition:** $v$.dim() = $v1$.dim().

`real v * const real_vector& v1` Scalar multiplication. **Precondition:** $v$.dim() = $v1$.dim().

`real_vector& v *= real r` multiplies all coordinates by $r$.

`real_vector v * real r` Componentwise multiplication with real $r$.

`bool v == const real_vector& w` Test for equality.

`bool v != const real_vector& w` Test for inequality.

`void v.print(ostream& O)` prints $v$ componentwise to stream $O$.

`void v.print()` prints $v$ to cout.

`void v.read(istream& I)` reads $d = v$.dim() numbers from input stream $I$ and writes them into $v[0] \ldots v[d-1]$.

`void v.read()` reads $v$ from cin.

`ostream& ostream& O << const real_vector& v` writes $v$ componentwise to the output stream $O$.

`istream& istream& I >> real_vector& v` reads $v$ componentwise from the input stream $I$.

`vector v.to_float()` returns a floating point approximation of $v$.

**Additional Operations for vectors in two and three-dimensional space**

`real v.xcoord()` returns the zero-th cartesian coordinate of $v$.

`real v.ycoord()` returns the first cartesian coordinate of $v$.

`real v.zcoord()` returns the second cartesian coordinate of $v$. 
int compare_by_angle(const real_vector& v1, const real_vector& v2)

For a non-zero vector $v$ let $\alpha(v)$ be the angle by which the positive $x$-axis has to be turned counterclockwise until it aligns with $v$. The function compares the angles defined by $v1$ and $v2$, respectively. The zero-vector precedes all non-zero vectors in the angle-order.

real_vector cross_product(const real_vector& v1, const real_vector& v2)

returns the cross product of the three-dimensional vectors $v1$ and $v2$.

4. Implementation

Vectors are implemented by arrays of real numbers. All operations on a vector $v$ take $O(v \cdot \text{dim}())$ real-number operations, except for dim and $[$ $]$ which take constant time. The space requirement depends on the size of the representations of the coordinates.
6.16 Real-Valued Matrices (\texttt{real_matrix})

1. Definition

An instance of the data type \texttt{real_matrix} is a matrix of variables of type \texttt{real}.

\texttt{#include <LEDA/numbers/real_matrix.h>}

2. Creation

\texttt{real_matrix M(int n = 0, int m = 0);}  
creates an instance \texttt{M} of type \texttt{real_matrix}, \texttt{M} is initialized to the \texttt{n} \times \texttt{m} - zero matrix.

\texttt{real_matrix M(int n, int m, real *D);}  
creates the \texttt{n} \times \texttt{m} matrix \texttt{M} with \texttt{M(i, j) = D[i \times m + j]} for \texttt{0 \leq i \leq n - 1} and \texttt{0 \leq j \leq m - 1}. \texttt{Precondition: D} points to an array of at least \texttt{n} \times \texttt{m} numbers of type \texttt{real}.

3. Operations

\texttt{int M.dim1( )}  
returns \texttt{n}, the number of rows of \texttt{M}.

\texttt{int M.dim2( )}  
returns \texttt{m}, the number of columns of \texttt{M}.

\texttt{real\_vector\& M.row(int i)}  
returns the \texttt{i}-th row of \texttt{M} (an \texttt{m}-vector).  
\texttt{Precondition: 0 \leq i \leq n - 1}.

\texttt{real\_vector M.col(int i)}  
returns the \texttt{i}-th column of \texttt{M} (an \texttt{n}-vector).  
\texttt{Precondition: 0 \leq i \leq m - 1}.

\texttt{real\_matrix M.trans( )}  
returns \texttt{M}^T (\texttt{m} \times \texttt{n} - matrix).

\texttt{real\_matrix M.inv( )}  
returns the inverse matrix of \texttt{M}.  
\texttt{Precondition: M} is quadratic and \texttt{M.det()} \neq 0.

\texttt{real M.det( )}  
returns the determinant of \texttt{M}.  
\texttt{Precondition: M} is quadratic.

\texttt{real\_vector M.solve(const real\_vector\& b)}  
returns vector \texttt{x} with \texttt{M \cdot x = b}.  
\texttt{Precondition: M.dim1()} == \texttt{M.dim2()} == \texttt{b.dim()} and \texttt{M.det()} \neq 0.

\texttt{real\& M(int i, int j)}  
returns \texttt{M}_{i,j}.  
\texttt{Precondition: 0 \leq i \leq n - 1} and \texttt{0 \leq j \leq m - 1}.
real_matrix M + const real_matrix & M1
Addition.
Precondition: M.dim1() == M1.dim1() and M.dim2() == M1.dim2().

real_matrix M - const real_matrix & M1
Subtraction.
Precondition: M.dim1() == M1.dim1() and M.dim2() == M1.dim2().

real_matrix M * const real_matrix & M1
Multiplication.
Precondition: M.dim2() == M1.dim1().

real_vector M * const real_vector & vec
Multiplication with vector.
Precondition: M.dim2() == vec.dim().

real_matrix M * real x
Multiplication with real x.

void M.print(ostream & O)
prints M row by row to ostream O.

void M.print()
prints M cout.

void M.read(istream & I)
reads M.dim1( ) \times M.dim2( ) numbers from input stream I and writes them row by row into matrix M.

void M.read()
prints M from cin.

ostream & ostream & O << const real_matrix & M
writes matrix M row by row to the output stream O.

istream & istream & I >> real_matrix & M
reads a matrix row by row from the input stream I and assigns it to M.

4. Implementation

Data type real_matrix is implemented by two-dimensional arrays of real numbers. Operations det, solve, and inv take time \(O(n^3)\) operations on reals, dim1, dim2, row, and col take constant time, all other operations perform \(O(nm)\) operations on reals. The space requirement is \(O(nm)\) plus the space for the \(nm\) entries of type real.

The functions in this section become available by including `numerical_analysis.h`.

### 6.17.1 Minima and Maxima

```c
double minimize_function(double (*f)(double), double& xmin, double tol = 1.0e-10)
```

finds a local minimum of the function $f$ of one argument. The minimizing argument is returned in $xmin$ and the minimal function value is returned as the result of the function. $xmin$ is determined with tolerance $tol$, i.e., the true value of the minimizing argument is contained in the interval $[xmin(1-\epsilon), xmin(1+\epsilon)]$, where $\epsilon = \max(1, xmin) \cdot tol$. Please do not choose $tol$ smaller than $10^{-15}$.

*Precondition:* If $+\infty$ or $-\infty$ is a local minimum of $f$, then the call of `minimize_function` may not terminate.

The algorithm is implemented as follows: First three arguments are determined such that $a < b < c$ (or $a > b > c$) and $f(a) \geq f(b) \leq f(c)$, i.e., $a$ and $c$ bracket a minimum. The interval is found by first taking two arbitrary arguments and comparing their function values. The argument with the larger function value is taken as $a$. Then steps of larger and larger size starting at $b$ are taken until a function value larger than $f(b)$ is found. Once the bracketing interval is found, golden-ratio search is applied to it.

```c
template <class F>
double minimize_function(const F& f, double& xmin, double tol = 1.0e-10)
```

a more flexible version of the above. It is assumed that class $F$ offers the operator

```c
double operator()(double x).
```

This operator is taken as the function $f$. 
6.17.2 Integration

```cpp
double integrate_function(double (*f)(double), double l, double r, double delta = 1.0e−2)
```

Computes the integral of \( f \) in the interval \([l, r]\) by forming the sum \( \delta \sum_{0 \leq i < K} f(l + i \cdot \delta) \), where \( K = (r - l) / \delta \).

**Precondition:** \( l \leq r \) and \( \delta > 0 \).

```cpp
template <class F>
double integrate_function(const F& f, double l, double r, double delta = 1.0e−2)
```

a more flexible version of the above. It is assumed that class \( F \) offers the operator

```cpp
double operator()(double x).
```

This operator is taken as the function \( f \).

6.17.3 Useful Numerical Functions

```cpp
double binary_entropy(double x)
```

returns the binary entropy of \( x \), i.e., \(-x \cdot \log x - (1 - x) \cdot \log(1 - x)\).

**Precondition:** \( 0 \leq x \leq 1 \).

6.17.4 Root Finding

```cpp
double zero_of_function(double (*f)(double), double l, double r, double tol = 1.0e−10)
```

returns a zero \( x \) of \( f \). We have either \( |f(x)| \leq 10^{-10} \) or there is an interval \([x_0, x_1]\) containing \( x \) such that \( f(x_0) \cdot f(x_1) \leq 0 \) and \( x_1 - x_0 \leq tol \cdot \max(1, |x_1| + |x_1|) \).

**Precondition:** \( l \leq r \) and \( f(l) \cdot f(r) \leq 0 \).

```cpp
template <class F>
double zero_of_function(const F& f, double l, double r, double tol = 1.0e−10)
```

a more flexible version of the above. It is assumed that class \( F \) offers the operator

```cpp
double operator()(double x).
```

This operator is taken as the function \( f \).
Chapter 7
Basic Data Types

7.1 One Dimensional Arrays (array)

1. Definition
An instance $A$ of the parameterized data type $\text{array}<E>$ is a mapping from an interval $I = [a..b]$ of integers, the index set of $A$, to the set of variables of data type $E$, the element type of $A$. $A(i)$ is called the element at position $i$. The array access operator ($A[i]$) checks its precondition ($a \leq i \leq b$). The check can be turned off by compiling with the flag `-DLEDA_CHECKING_OFF`.

```cpp
#include <LEDA/core/array.h>
```

2. Types

$\text{array}<E>::\text{item}$ the item type.

$\text{array}<E>::\text{value_type}$ the value type.

3. Creation

$\text{array}<E> A(\text{int} low, \text{int} high)$;
creates an instance $A$ of type $\text{array}<E>$ with index set $[low..high]$.

$\text{array}<E> A(\text{int} n)$;
creates an instance $A$ of type $\text{array}<E>$ with index set $[0..n - 1]$.

$\text{array}<E> A$; creates an instance $A$ of type $\text{array}<E>$ with empty index set.

Special Constructors

$\text{array}<E> A(\text{int} low, \text{const} E& x, \text{const} E& y)$;
creates an instance $A$ of type $\text{array}<E>$ with index set $[low, low + 1]$ initialized to $[x, y]$.
array\langle E \rangle \ A(int \ low, \ const \ E \& \ x, \ const \ E \& \ y, \ const \ E \& \ w);  
creates an instance $A$ of type $\text{array}\langle E \rangle$ with index set $[\text{low}, \text{low} + 2]$ initialized to $[x, y, w]$.

array\langle E \rangle \ A(int \ low, \ const \ E \& \ x, \ const \ E \& \ y, \ const \ E \& \ z, \ const \ E \& \ w);  
creates an instance $A$ of type $\text{array}\langle E \rangle$ with index set $[\text{low}, \text{low} + 3]$ initialized to $[x, y, z, w]$.

4. Operations

**Basic Operations**

$E\& \ A[int \ x]$  
returns $A(x)$.  
*Precondition: $a \leq x \leq b$.*

$E\& \ A.get(int \ x)$  
returns $A(x)$.  
*Precondition: $a \leq x \leq b$.*

$\text{void} \ A.set(int \ x, \ const \ E\& \ e)$  
sets $A(x) = e$.  
*Precondition: $a \leq x \leq b$.*

$\text{void} \ A.swap(int \ i, \ int \ j)$  
swaps the values of $A[i]$ and $A[j]$.

$\text{void} \ A.copy(int \ x, \ int \ y)$  
sets $A(x) = A(y)$.  
*Precondition: $a \leq x \leq b$ and $\text{low()} \leq y \leq \text{high()}$.*

$\text{void} \ A.copy(int \ x, \ const \ \text{array}\langle E\& \rangle\& \ B, \ int \ y)$  
sets $A(x) = B(y)$.  
*Precondition: $a \leq x \leq b$ and $B.\text{low()} \leq y \leq B.\text{high()}$.*

$\text{void} \ A.resize(int \ low, \ int \ high)$  
sets the index set of $A$ to $[a..b]$ such that for all $i \in [a..b]$ which are not contained in the old index set $A(i)$ is set to the default value of type $E$.

$\text{void} \ A.resize(int \ n)$  
same as $A.resize(0, n - 1)$.

$\text{int} \ A.\text{low()}$  
returns the minimal index $a$ of $A$.

$\text{int} \ A.\text{high()}$  
returns the maximal index $b$ of $A$.

$\text{int} \ A.\text{size()}$  
returns the size $(b - a + 1)$ of $A$.

$\text{void} \ A.\text{init}(\text{const} \ E\& \ x)$  
assigns $x$ to $A[i]$ for every $i \in \{a...b\}$.

$\text{bool} \ A.\text{C-style()}$  
returns $true$ if the array has “C-style”, i.e., the index set is $[0..\text{size} - 1]$.
void A.permute()  
the elements of A are randomly permuted.

void A.permute(int low, int high)  
the elements of A[low..high] are randomly permuted.

### Sorting and Searching

void A.sort(int (*cmp)(const E&, const E&)) 
sorts the elements of A using function cmp to compare two elements, i.e., if \((in_a,\ldots,in_b)\) and \((out_a,\ldots,out_b)\) denote the values of the variables \((A(a),\ldots,A(b))\) before and after the call of sort, then \(cmp(out_i, out_j) \leq 0\) for \(i \leq j\) and there is a permutation \(\pi\) of \([a..b]\) such that \(out_i = in_{\pi(i)}\) for \(a \leq i \leq b\).

void A.sort()  
sorts the elements of A according to the linear order of the element type E. **Precondition:** A linear order on E must have been defined by \(\text{compare(\text{const E}&, \text{const E}&)}\) if E is a user-defined type (see Section 2.3).

void A.sort(int (*cmp)(const E&, const E&), int low, int high)  
sorts sub-array A[low..high] using compare function \(\text{cmp}\).

void A.sort(int low, int high)  
sorts sub-array A[low..high] using the linear order on E. If E is a user-defined type, you have to define the linear order by providing the compare function (see Section 2.3).

int A.unique()  
removes duplicates from A by copying the unique elements of A to \(A[A.low()],\ldots,A[h]\) and returns \(h\) \((A.low() - 1\) if A is empty). **Precondition:** A is sorted increasingly according to the default ordering of type E. If E is a user-defined type, you have to define the linear order by providing the compare function (see Section 2.3).

int A.binary_search(int (*cmp)(const E&, const E&), const E& x)  
performs a binary search for x. Returns an i with \(A[i] = x\) if x in A, \(A.low() - 1\) otherwise. Function \(\text{cmp}\) is used to compare two elements. **Precondition:** A must be sorted according to \(\text{cmp}\).

int A.binary_search(const E& x)  
as above but uses the default linear order on E. If E is a user-defined type, you have to define the linear order by providing the compare function (see Section 2.3).
int A.binary_locate(int (*cmp)(const E&, const E&), const E& x)

Returns the maximal i with A[i] \leq x or A.low()-1 if x < A[low]. Function cmp is used to compare elements. **Precondition:** A must be sorted according to cmp.

int A.binary_locate(const E& x)

as above but uses the default linear order on E. If E is a user-defined type, you have to define the linear order by providing the compare function (see Section 2.3).

**Input and Output**

void A.read(istream& I) reads b–a+1 objects of type E from the input stream I into the array A using the operator >> (istream&, E&).

void A.read() calls A.read(cin) to read A from the standard input stream cin.

void A.read(string s) As above, uses string s as prompt.

void A.print(ostream& O, char space = ' ') prints the contents of array A to the output stream O using operator << (ostream&, const E&) to print each element. The elements are separated by character space.

void A.print(char space = ' ') calls A.print(cout, space) to print A on the standard output stream cout.

void A.print(string s, char space = ' ') As above, uses string s as header.

ostream& ostream& out << const array<E>& A

same as A.print(out); returns out.

istream& istream& in >> array<E>& A

same as A.read(in)); returns in.

**Iteration**

STL compatible iterators are provided when compiled with `-DLEDA_STL_ITERATORS` (see LEDAROOT/demo/stl/array.c for an example).

**5. Implementation**

Arrays are implemented by C++vectors. The access operation takes time $O(1)$, the sorting is realized by quicksort (time $O(n \log n)$) and the binary search operation takes time $O(\log n)$, where $n = b - a + 1$. The space requirement is $O(n \times sizeof(E))$. 
7.2 Two Dimensional Arrays (array2)

1. Definition

An instance $A$ of the parameterized data type $\text{array2}\langle E \rangle$ is a mapping from a set of pairs $I = [a..b] \times [c..d]$, called the index set of $A$, to the set of variables of data type $E$, called the element type of $A$, for two fixed intervals of integers $[a..b]$ and $[c..d]$. $A(i, j)$ is called the element at position $(i, j)$.

```c
#include <LEDA/core/array2.h>
```

2. Creation

```c
\text{array2}\langle E \rangle \ A(\text{int} \ a, \ \text{int} \ b, \ \text{int} \ c, \ \text{int} \ d);
```
creates an instance $A$ of type $\text{array2}\langle E \rangle$ with index set $[a..b] \times [c..d]$.

```c
\text{array2}\langle E \rangle \ A(\text{int} \ n, \ \text{int} \ m);
```
creates an instance $A$ of type $\text{array2}\langle E \rangle$ with index set $[0..n-1] \times [0..m-1]$.

3. Operations

```c
\text{E} & \ A(\text{int} \ i, \ \text{int} \ j) \quad \text{returns} \ A(i, j).
```
Precondition: $a \leq i \leq b$ and $c \leq j \leq d$.

```c
\text{int} \ A.\text{low1}(\ ) \quad \text{returns} \ a.
```

```c
\text{int} \ A.\text{high1}(\ ) \quad \text{returns} \ b.
```

```c
\text{int} \ A.\text{low2}(\ ) \quad \text{returns} \ c.
```

```c
\text{int} \ A.\text{high2}(\ ) \quad \text{returns} \ d.
```

4. Implementation

Two dimensional arrays are implemented by C++ vectors. All operations take time $O(1)$, the space requirement is $O(I \times \text{sizeof}(E))$. 
7.3 Stacks (stack)

1. Definition

An instance \( S \) of the parameterized data type \( \text{stack}\langle E \rangle \) is a sequence of elements of data type \( E \), called the element type of \( S \). Insertions or deletions of elements take place only at one end of the sequence, called the top of \( S \). The size of \( S \) is the length of the sequence, a stack of size zero is called the empty stack.

```cpp
#include <LEDA/core/stack.h>
```

2. Creation

\[ \text{stack}\langle E \rangle \ S; \] creates an instance \( S \) of type \( \text{stack}\langle E \rangle \). \( S \) is initialized with the empty stack.

3. Operations

\begin{align*}
\text{const } E\& \quad & S.\text{top( )} & \text{returns the top element of } S. \\
\text{Precondition: } S & \text{ is not empty.} \\
\text{void} \quad & S.\text{push( const } E\& \ x) & \text{adds } x \text{ as new top element to } S. \\
\text{E} \quad & S.\text{pop( )} & \text{deletes and returns the top element of } S. \\
\text{Precondition: } S & \text{ is not empty.} \\
\text{int} \quad & S.\text{size( )} & \text{returns the size of } S. \\
\text{bool} \quad & S.\text{empty( )} & \text{returns true if } S \text{ is empty, false otherwise.} \\
\text{void} \quad & S.\text{clear( )} & \text{makes } S \text{ the empty stack.}
\end{align*}

4. Implementation

Stacks are implemented by singly linked linear lists. All operations take time \( O(1) \), except clear which takes time \( O(n) \), where \( n \) is the size of the stack.
7.4 Queues (queue)

1. Definition

An instance \( Q \) of the parameterized data type \( \text{queue}<E> \) is a sequence of elements of data type \( E \), called the element type of \( Q \). Elements are inserted at one end (the rear) and deleted at the other end (the front) of \( Q \). The size of \( Q \) is the length of the sequence; a queue of size zero is called the empty queue.

```
#include <LEDA/core/queue.h>
```

2. Types

\( \text{queue}<E>::\text{value\_type} \) the value type.

3. Creation

\( \text{queue}<E> \ Q; \) creates an instance \( Q \) of type \( \text{queue}<E> \). \( Q \) is initialized with the empty queue.

4. Operations

- \( \text{const } E& \ Q.\text{top}(\) \) returns the front element of \( Q \).
  
  **Precondition:** \( Q \) is not empty.

- \( E \ Q.\text{pop}(\) \) deletes and returns the front element of \( Q \).
  
  **Precondition:** \( Q \) is not empty.

- \( \text{void } Q.\text{append}(\text{const } E& \ x) \) appends \( x \) to the rear end of \( Q \).

- \( \text{void } Q.\text{push}(\text{const } E& \ x) \) inserts \( x \) at the front end of \( Q \).

- \( \text{int } Q.\text{size}(\) \) returns the size of \( Q \).

- \( \text{int } Q.\text{length}(\) \) returns the size of \( Q \).

- \( \text{bool } Q.\text{empty}(\) \) returns true if \( Q \) is empty, false otherwise.

- \( \text{void } Q.\text{clear}(\) \) makes \( Q \) the empty queue.

**Iteration**

```
forall(x, Q) { “the elements of Q are successively assigned to x” }
```

5. Implementation

Queues are implemented by singly linked linear lists. All operations take time \( O(1) \), except \text{clear} which takes time \( O(n) \), where \( n \) is the size of the queue.
7.5 Bounded Stacks (b_stack)

1. Definition

An instance \( S \) of the parameterized data type \( b\text{\_stack}<E> \) is a stack (see section 7.3) of bounded size.

```c
#include <LEDA/core/b_stack.h>
```

2. Creation

\[
b\text{\_stack}<E> \ S(int \ n);
\]

creates an instance \( S \) of type \( b\text{\_stack}<E> \) that can hold up to \( n \) elements. \( S \) is initialized with the empty stack.

3. Operations

\[
\text{const } E \& \ S\text{.top( )} \quad \text{returns the top element of } S.
\]

\[
\text{Precondition: } S \text{ is not empty.}
\]

\[
\text{const } E \& \ S\text{.pop( )} \quad \text{deletes and returns the top element of } S.
\]

\[
\text{Precondition: } S \text{ is not empty.}
\]

\[
\text{void } S\text{.push(\text{const } E \& x)} \quad \text{adds } x \text{ as new top element to } S.
\]

\[
\text{Precondition: } S\text{.size()} < n.
\]

\[
\text{void } S\text{.clear( )} \quad \text{makes } S \text{ the empty stack.}
\]

\[
\text{int } S\text{.size( )} \quad \text{returns the size of } S.
\]

\[
\text{bool } S\text{.empty( )} \quad \text{returns true if } S \text{ is empty, false otherwise.}
\]

4. Implementation

Bounded stacks are implemented by C++vectors. All operations take time \( O(1) \). The space requirement is \( O(n) \).
7.6 Bounded Queues (b_queue)

1. Definition

An instance $Q$ of the parameterized data type $b_queue<E>$ is a (double ended) queue (see section 7.4) of bounded size.

```cpp
#include <LEDA/core/b_queue.h>
```

2. Creation

$b_queue<E> Q(int n);$ creates an instance $Q$ of type $b_queue<E>$ that can hold up to $n$ elements. $Q$ is initialized with the empty queue.

3. Operations

- `const E& Q.front()` returns the first element of $Q$.  
  **Precondition:** $Q$ is not empty.
- `const E& Q.back()` returns the last element of $Q$.  
  **Precondition:** $Q$ is not empty.
- `const E& Q.pop_front()` deletes and returns the first element of $Q$.  
  **Precondition:** $Q$ is not empty.
- `const E& Q.pop_back(const E& x)` deletes and returns the last element of $Q$.  
  **Precondition:** $Q$ is not empty.
- `void Q.push_front(const E& x)` inserts $x$ at the beginning of $Q$.  
  **Precondition:** $Q.size()<n$.
- `void Q.push_back(const E& x)` inserts $x$ at the end of $Q$.  
  **Precondition:** $Q.size()<n$.
- `void Q.append(const E& x)` same as $Q.push_back( )$.
- `void Q.clear()` makes $Q$ the empty queue.
- `int Q.size()` returns the size of $Q$.
- `int Q.length()` same as $Q.size( )$.
- `bool Q.empty()` returns true if $Q$ is empty, false otherwise.

**Stack Operations**

- `const E& Q.top()` same as $Q.front( )$. 
const E& Q.pop() same as Q.pop_front().

void Q.push(const E& x) same as Q.push_front().

Iteration

forall(x, Q) { “the elements of Q are successively assigned to x” }

4. Implementation

Bounded queues are implemented by circular arrays. All operations take time $O(1)$. The space requirement is $O(n)$. 
7.7 Linear Lists (list)

1. Definition

An instance \( L \) of the parameterized data type \( \text{list}<E> \) is a sequence of items \( (\text{list}<E>::\text{item}) \). Each item in \( L \) contains an element of data type \( E \), called the element or value type of \( L \). The number of items in \( L \) is called the length of \( L \). If \( L \) has length zero it is called the empty list. In the sequel \( (x) \) is used to denote a list item containing the element \( x \) and \( L[i] \) is used to denote the contents of list item \( i \) in \( L \).

```c#
#include <LEDA/core/list.h>
```

2. Types

\( \text{list}<E>::\text{item} \) the item type.
\( \text{list}<E>::\text{value}\_\text{type} \) the value type.

3. Creation

\( \text{list}<E> L; \) creates an instance \( L \) of type \( \text{list}<E> \) and initializes it to the empty list.

4. Operations

Access Operations

\( \text{int} \ L.\text{length}(\) \) returns the length of \( L \).
\( \text{int} \ L.\text{size}(\) \) returns \( L.\text{length}() \).
\( \text{bool} \ L.\text{empty}(\) \) returns true if \( L \) is empty, false otherwise.
\( \text{list}\_\text{item} \ L.\text{first}(\) \) returns the first item of \( L \) (nil if \( L \) is empty).
\( \text{list}\_\text{item} \ L.\text{last}(\) \) returns the last item of \( L \). (nil if \( L \) is empty)
\( \text{list}\_\text{item} \ L.\text{get}\_\text{item}(\) \text{int \ i} \) returns the item at position \( i \) (the first position is 0).
Precondition: \( 0 \leq i < L.\text{length}() \). \textbf{Note} that this takes time linear in \( i \).
\( \text{list}\_\text{item} \ L.\text{succ}(\) \text{list}\_\text{item \ it} \) returns the successor item of item \( it \), nil if \( it = L.\text{last}() \).
Precondition: \( it \) is an item in \( L \).
\( \text{list}\_\text{item} \ L.\text{pred}(\) \text{list}\_\text{item \ it} \) returns the predecessor item of item \( it \), nil if \( it = L.\text{first}() \).
Precondition: \( it \) is an item in \( L \).
list_item \( L.cyclic\_succ(list\_item\ it) \) returns the cyclic successor of item \( it \), i.e., \( L.first() \) if \( it = L.last() \), \( L.succ(it) \) otherwise.

cyclic

list_item \( L.cyclic\_pred(list\_item\ it) \) returns the cyclic predecessor of item \( it \), i.e., \( L.last() \) if \( it = L.first() \), \( L.pred(it) \) otherwise.

const E& \( L.contents(list\_item\ it) \) returns the contents \( L[it] \) of item \( it \).

Precondition: it is an item in \( L \).

cyclic

const E& \( L.inf(list\_item\ it) \) returns \( L.contents(it) \).

cyclic

const E& \( L.front() \) returns the first element of \( L \), i.e. the contents of \( L.first() \).

Precondition: \( L \) is not empty.

cyclic

const E& \( L.head() \) same as \( L.front() \).

cyclic

const E& \( L.back() \) returns the last element of \( L \), i.e. the contents of \( L.last() \).

Precondition: \( L \) is not empty.

cyclic

const E& \( L.tail() \) same as \( L.back() \).

ncyclic

int \( L.rank(const\ E\ &\ x) \) returns the rank of \( x \) in \( L \), i.e. its first position in \( L \) as an integer from \([1..|L|]\) (0 if \( x \) is not in \( L \)). Note that this takes time linear in \( rank(x) \).

Precondition: operator== has to be defined for type \( E \).

Update Operations

list_item \( L.push(const\ E\ &\ x) \) adds a new item \( \langle x \rangle \) at the front of \( L \) and returns it \((L.insert(x, L.first(), leda::before))\).

list_item \( L.push\_front(const\ E\ &\ x) \) same as \( L.push(x) \).

list_item \( L.append(const\ E\ &\ x) \) appends a new item \( \langle x \rangle \) to \( L \) and returns it \((L.insert(x, L.last(), leda::behind))\).

list_item \( L.push\_back(const\ E\ &\ x) \) same as \( L.append(x) \).

list_item \( L.insert(const\ E\ &\ x, list\_item\ pos, int\ dir = leda::behind) \) inserts a new item \( \langle x \rangle \) behind (if \( dir = leda::behind \)) or in front of (if \( dir = leda::before \)) item \( pos \) into \( L \) and returns it (here \( leda::behind \) and \( leda::before \) are predefined constants).

Precondition: it is an item in \( L \).

cyclic

E \( L.pop() \) deletes the first item from \( L \) and returns its contents.

Precondition: \( L \) is not empty.
7.7. LINEAR LISTS (LIST)

\[
\begin{align*}
E & \quad L.pop_front() \quad \text{same as } L.pop(). \\
E & \quad L.pop_back() \quad \text{deletes the last item from } L \text{ and returns its contents.} \\
& \quad \text{Precondition: } L \text{ is not empty.} \\
E & \quad L.Pop() \quad \text{same as } L.pop_back(). \\
E & \quad L.del_item(list_item it) \quad \text{deletes the item } it \text{ from } L \text{ and returns its contents } L[it]. \\
& \quad \text{Precondition: } it \text{ is an item in } L. \\
E & \quad L.del(list_item it) \quad \text{same as } L.del_item(it). \\
void & \quad L.erase(list_item it) \quad \text{deletes the item } it \text{ from } L. \\
& \quad \text{Precondition: } it \text{ is an item in } L. \\
void & \quad L.remove(const E& x) \quad \text{removes all items with contents } x \text{ from } L. \\
& \quad \text{Precondition: } \text{operator}== \text{ has to be defined for type } E. \\
void & \quad L.move_to_front(list_item it) \quad \text{moves } it \text{ to the front end of } L. \\
void & \quad L.move_to_rear(list_item it) \quad \text{moves } it \text{ to the rear end of } L. \\
void & \quad L.move_to_back(list_item it) \quad \text{same as } L.move_to_rear(it). \\
void & \quad L.assign(list_item it, const E& x) \quad \text{makes } x \text{ the contents of item } it. \\
& \quad \text{Precondition: } it \text{ is an item in } L. \\
void & \quad L.concat(list<E>& L1, int dir = leda::behind) \quad \text{appends } (dir = leda::behind \text{ or prepends } (dir = leda::before) \text{ list } L1 \text{ to list } L \text{ and makes } L1 \text{ the empty list.} \\
& \quad \text{Precondition: } : L \neq L1. \\
void & \quad L.swap(list<E>& L1) \quad \text{swaps lists of items of } L \text{ and } L1; \\
void & \quad L.split(list_item it, list<E>& L1, list<E>& L2) \quad \text{splits } L \text{ at item } it \text{ into lists } L1 \text{ and } L2. \quad \text{More precisely, if } it \neq \text{nil and } \\
& \quad L = x_1, \ldots, x_{k-1}, it, x_{k+1}, \ldots, x_n \text{ then } \\
& \quad L1 = x_1, \ldots, x_{k-1} \text{ and } L2 = it, x_{k+1}, \ldots, x_n. \quad \text{If } it = \text{nil then } L1 \text{ is made empty and } L2 \text{ a copy of } L. \quad \text{Finally } L \text{ is made empty if it is not identical} \\
& \quad \text{to } L1 \text{ or } L2. \\
& \quad \text{Precondition: } it \text{ is an item of } L \text{ or } \text{nil.}
\end{align*}
split: L.split(list_item it, list<> & L1, list<> & L2, int dir)

splits L at item it into lists L1 and L2. Item it becomes the first item of L2 if dir == leda::before and the last item of L1 if dir = leda::behind.

Precondition: it is an item of L.

extract: L.extract(list_item it1, list_item it2, list<> & L1, bool inclusive = true)

extracts a sublist L1 from L. More precisely, if L = x1, ..., xp, it1, ..., it2, xs, ..., xn then L1 = it1, ..., it2 and L = x1, ..., xp, xs, ..., xn. (If inclusive is false then it1 and it2 remain in L.)

Precondition: it1 and it2 are items of L or nil.

apply: L.apply(void (*)(E & x)) for all items ⟨x⟩ in L function f is called with argument x (passed by reference).

reverse: L.reverse()

reverses the sequence of entries of L.

reverse: L.reverse(list_item it1, list_item it2)

reverses the sub-sequence it1, ..., it2 of items of L.

Precondition: it1 = it2 or it1 appears before it2 in L.

permute: L.permute()

randomly permutes the items of L.

permute: L.permute(list_item *I)

permutes the items of L into the same order as stored in the array I.

clear: L.clear()

makes L the empty list.
Sorting and Searching

void L.sort(int (*cmp)(const E&, const E&))

sorts the items of L using the ordering defined by the compare function \( cmp : E \times E \rightarrow \mathbb{Z} \), with

\[
\begin{align*}
    \text{cmp}(a,b) &= \begin{cases} 
        < 0, & \text{if } a < b \\
        = 0, & \text{if } a = b \\
        > 0, & \text{if } a > b
    \end{cases}
\end{align*}
\]

More precisely, if \((in_1, \ldots, in_n)\) and \((out_1, \ldots, out_n)\) denote the values of L before and after the call of sort, then \( \text{cmp}(L[out_j], L[out_{j+1}]) \leq 0 \) for \( 1 \leq j < n \) and there is a permutation \( \pi \) of \([1..n]\) such that \( out_i = in_{\pi_i} \) for \( 1 \leq i \leq n \).

void L.sort()

sorts the items of L using the default ordering of type E, i.e., the linear order defined by function \text{int compare(const E&, const E&)}. If E is a user-defined type, you have to provide the compare function (see Section 2.3).

void L.merge_sort(int (*cmp)(const E&, const E&))

sorts the items of L using merge sort and the ordering defined by \( cmp \). The sort is stable, i.e., if \( x = y \) and \( \langle x \rangle \) is before \( \langle y \rangle \) in L then \( \langle x \rangle \) is before \( \langle y \rangle \) after the sort. \( L\text{.merge\_sort()} \) is more efficient than \( L\text{.sort()} \) if L contains large pre-sorted intervals.

void L.merge_sort()

as above, but uses the default ordering of type E. If E is a user-defined type, you have to provide the compare function (see Section 2.3).

void L.bucket_sort(int i, int j, int (*b)(const E&))

sorts the items of L using bucket sort, where \( b \) maps every element \( x \) of L to a bucket \( b(x) \in [i..j] \). If \( b(x) < b(y) \) then \( \langle x \rangle \) appears before \( \langle y \rangle \) after the sort. If \( b(x) = b(y) \), the relative order of \( x \) and \( y \) before the sort is retained, thus the sort is stable.

void L.bucket_sort(int (*b)(const E&))

sorts list\(<E>\) into increasing order as prescribed by \( b \). Precondition: \( b \) is an integer-valued function on \( E \).
merges the items of \( L \) and \( L_1 \) using the ordering defined by \( cmp \). The result is assigned to \( L \) and \( L_1 \) is made empty.

**Precondition:** \( L \) and \( L_1 \) are sorted increasingly according to the linear order defined by \( cmp \).

```cpp
void L.merge(list<E>& L1) { /* merges the items of L and L1 using the default linear order of type E. If E is a user-defined type, you have to define the linear order by providing the compare function (see Section 2.3). */ }
```

```cpp
void L.unique(int (*cmp)(const E&, const E&)) { /* removes duplicates from L. */
    Preconditions: L is sorted increasingly according to the ordering defined by cmp.
```

```cpp
void L.unique() { /* removes duplicates from L. */
    Preconditions: L is sorted increasingly according to the default ordering of type E and operator== is defined for E. If E is a user-defined type, you have to define the linear order by providing the compare function (see Section 2.3).
```

```cpp
list_item L.search(const E& x) { /* returns the first item of L that contains x, nil if x is not an element of L. */
    Preconditions: operator== has to be defined for type E.
```

```cpp
list_item L.min(const leda_cmp_base<E>& cmp) { /* returns the item with the minimal contents with respect to the linear order defined by compare function cmp. */
```

```cpp
list_item L.min() { /* returns the item with the minimal contents with respect to the default linear order of type E. */
```

```cpp
list_item L.max(const leda_cmp_base<E>& cmp) { /* returns the item with the maximal contents with respect to the linear order defined by compare function cmp. */
```

```cpp
list_item L.max() { /* returns the item with the maximal contents with respect to the default linear order of type E. */
```

**Input and Output**

```cpp
void L.read(istream& I) { /* reads a sequence of objects of type E from the input stream I using operator >> (istream&, E&). L is made a list of appropriate length and the sequence is stored in L. */
```
void L.read(istream& I, char delim)
    as above but stops reading as soon as the first occurrence of character delim is encountered.

void L.read(char delim = 'n')
calls L.read(cin, delim) to read L from the standard input stream cin.

void L.read(string prompt, char delim = 'n')
As above, but first writes string prompt to cout.

void L.print(ostream& O, char space = ' ')
prints the contents of list L to the output stream O using operator ≪ (ostream&, const E&) to print each element. The elements are separated by character space.

void L.print(char space = ' ')
calls L.print(cout, space) to print L on the standard output stream cout.

void L.print(string header, char space = ' ')
As above, but first outputs string header.

Operators

list<E>& L = const list<E>& L1 The assignment operator makes L a copy of list L1. More precisely if L1 is the sequence of items x1, x2, ..., xn then L is made a sequence of items y1, y2, ..., yn with L[yi] = L1[xi] for 1 ≤ i ≤ n.

E& L[list_item it] returns a reference to the contents of it.

list_item L[int i] an abbreviation for L.get_item(i).

list_item L += const E& x same as L.append(x); returns the new item.

ostream& ostream& out ≪ const list<E>& L
same as L.print(out); returns out.

istream& istream& in >> list<E>& L
same as L.read(in)); returns in.

Iteration

forall_items(it, L) { “the items of L are successively assigned to it” }

forall(x, L) { “the elements of L are successively assigned to x” }
STL compatible iterators are provided when compiled with \texttt{-DLEDA\_STL\_ITERATORS} (see LEDAROOT/demo/stl/list.c for an example).

5. Implementation

The data type list is realized by doubly linked linear lists. Let $c$ be the time complexity of the compare function and let $d$ be the time needed to copy an object of type \texttt{list$<$E$>$}. All operations take constant time except of the following operations: search, reverse, items, permute and rank take linear time $O(n)$, item$(i)$ takes time $O(i)$, min, max, and unique take time $O(c \cdot n)$, merge takes time $O(c \cdot (n1 + n2))$, operator=, apply, reverse, read, and print take time $O(d \cdot n)$, sort and merge_sort take time $O(n \cdot c \cdot \log n)$, and bucket_sort takes time $O(e \cdot n + j - i)$, where $e$ is the time complexity of $f$. $n$ is always the current length of the list.
7.8 Singly Linked Lists ( slist )

1. Definition
An instance \( L \) of the parameterized data type \( \text{slist}<E> \) is a sequence of items
(\( \text{slist}<E>::\text{item} \)). Each item in \( L \) contains an element of data type \( E \), called the ele-
ment or value type of \( L \). The number of items in \( L \) is called the length of \( L \). If \( L \) has
length zero it is called the empty list. In the sequel \( \langle x \rangle \) is used to denote a list item
containing the element \( x \) and \( L[i] \) is used to denote the contents of list item \( i \) in \( L \).

\#include < LEDA/core/slist.h >

2. Types
\( \text{slist}<E>::\text{item} \) the item type.
\( \text{slist}<E>::\text{value_type} \) the value type.

3. Creation
\( \text{slist}<E> \ L; \) creates an instance \( L \) of type \( \text{slist}<E> \) and initializes it to the
empty list.

\( \text{slist}<E> \ L(\text{const } E& \ x); \) creates an instance \( L \) of type \( \text{slist}<E> \) and initializes it to the
one-element list \( \langle x \rangle \).

4. Operations
\( \text{int } \ L.\text{length(); } \) returns the length of \( L \).
\( \text{int } \ L.\text{size(); } \) returns \( L.\text{length(); } \)
\( \text{bool } \ L.\text{empty(); } \) returns true if \( L \) is empty, false otherwise.
\( \text{item } \ L.\text{first(); } \) returns the first item of \( L \).
\( \text{item } \ L.\text{last(); } \) returns the last item of \( L \).
\( \text{item } \ L.\text{succ(item it);} \) returns the successor item of item \( it \), nil if \( it = L.\text{last(); } \)
Precondition: \( it \) is an item in \( L \).
\( \text{item } \ L.\text{cyclic}\_\text{succ(item l);} \) returns the cyclic successor of item \( it \), i.e., \( L.\text{first(); } \)
if \( it = L.\text{last(); } \), \( L.\text{succ(it);} \) otherwise.
\( \text{const } E& \ L.\text{contents(item it);} \) returns the contents \( L[it] \) of item \( it \).
Precondition: \( it \) is an item in \( L \).
\( \text{const } E& \ L.\text{inf(item it);} \) returns \( L.\text{contents(it);} \).
Precondition: \( it \) is an item in \( L \).
const E& L.head() returns the first element of L, i.e. the contents of L.first().
Precondition: L is not empty.

const E& L.tail() returns the last element of L, i.e. the contents of L.last().
Precondition: L is not empty.

item L.push(const E& x) adds a new item ⟨x⟩ at the front of L and returns it.

item L.append(const E& x) appends a new item ⟨x⟩ to L and returns it.

item L.insert(const E& x, item pos) inserts a new item ⟨x⟩ after item pos into L and returns it.
Precondition: it is an item in L.

E L.pop() deletes the first item from L and returns its contents.
Precondition: L is not empty.

void L.del_succ_item(item it) deletes the successor of item it from L.
Precondition: it is an item in L and has a successor.

void L.conc(slist<E>& L) appends list L₁ to list L and makes L₁ the empty list.
Precondition: L₁ = L₁.

void L.clear() makes L the empty list.

E& L[item it] returns a reference to the contents of it.

item L += const E& x appends a new item ⟨x⟩ to L and returns it.
7.9 Sets (set)

1. Definition

An instance $S$ of the parameterized data type set<$E$> is a collection of elements of the linearly ordered type $E$, called the element type of $S$. The size of $S$ is the number of elements in $S$, a set of size zero is called the empty set.

#include <LEDA/core/set.h>

2. Creation

set<$E$> $S$; creates an instance $S$ of type set<$E$> and initializes it to the empty set.

3. Operations

void $S$.insert(const $E$& $x$) adds $x$ to $S$.

void $S$.del(const $E$& $x$) deletes $x$ from $S$.

bool $S$.member(const $E$& $x$) returns true if $x$ in $S$, false otherwise.

const $E$& $S$.choose() returns an element of $S$.
Precondition: $S$ is not empty.

set<$E$, set_impl>$S$.join(const set<$E$, set_impl>& $T$)
returns $S \cup T$.

set<$E$, set_impl>$S$.diff(const set<$E$, set_impl>& $T$)
returns $S - T$.

set<$E$, set_impl>$S$.intersect(const set<$E$, set_impl>& $T$)
returns $S \cap T$.

set<$E$, set_impl>$S$.symdiff(const set<$E$, set_impl>& $T$)
returns the symmetric difference of $S$ and $T$.

set<$E$, set_impl>$S + \text{const } set<$E$, set_impl>& $T$
returns $S$.join($T$).

set<$E$, set_impl>$S - \text{const } set<$E$, set_impl>& $T$
returns $S$.diff($T$).

set<$E$, set_impl>$S \& \text{const } set<$E$, set_impl>& $T$
returns $S$.intersect($T$).
set<E, set_impl>& S % const set<E, set_impl>& T  
returns S.symdiff(T).

set<E, set_impl>& S += const set<E, set_impl>&& T  
assigns S.join(T) to S and returns S.

set<E, set_impl>& S -= const set<E, set_impl>&& T  
assigns S.diff(T) to S and returns S.

set<E, set_impl>& S &= const set<E, set_impl>&& T  
assigns S.intersect(T) to S and returns S.

set<E, set_impl>& S %= const set<E, set_impl>&& T  
assigns S.symdiff(T) to S and returns S.

bool S ≤ const set<E, set_impl>& T  
returns true if S ⊆ T, false otherwise.

bool S ≥ const set<E, set_impl>& T  
returns true if T ⊆ S, false otherwise.

bool S == const set<E, set_impl>& T  
returns true if S = T, false otherwise.

bool S != const set<E, set_impl>& T  
returns true if S ≠ T, false otherwise.

bool S < const set<E, set_impl>& T  
returns true if S ⊂ T, false otherwise.

bool S > const set<E, set_impl>& T  
returns true if T ⊂ S, false otherwise.

bool S.empty()  
returns true if S is empty, false otherwise.

int S.size()  
returns the size of S.

void S.clear()  
makes S the empty set.

Iteration

forall(x, S) { "the elements of S are successively assigned to x" }
4. Implementation

Sets are implemented by randomized search trees [2]. Operations insert, del, member take time $O(\log n)$, empty, size take time $O(1)$, and clear takes time $O(n)$, where $n$ is the current size of the set.

The operations join, intersect, and diff have the following running times: Let $S_1$ and $S_2$ be a two sets of type $T$ with $|S_1| = n_1$ and $|S_2| = n_2$. Then $S_1\text{.join}(S_2)$ and $S_1\text{.diff}(S_2)$ need time $O(n_2 \log(n_1 + n_2))$, $S_1\text{.intersect}(S_2)$ needs time $O(n_1 \log(n_1 + n_2))$. 
7.10 Integer Sets (int_set)

1. Definition

An instance $S$ of the data type int_set is a subset of a fixed interval $[a..b]$ of the integers, called the range of $S$.

```c
#include <LEDA/core/int_set.h>
```

2. Creation

```c
int_set S(int a, int b);
```
creates an instance $S$ of type int_set for elements from $[a..b]$ and initializes it to the empty set.

```c
int_set S(int n);
```
creates an instance $S$ of type int_set for elements from $[0..n-1]$ and initializes it to the empty set.

3. Operations

```c
void S.insert(int x) adds $x$ to $S$.
Precondition: $a \leq x \leq b$.
```

```c
void S.del(int x) deletes $x$ from $S$.
Precondition: $a \leq x \leq b$.
```

```c
bool S.member(int x) returns true if $x$ in $S$, false otherwise.
Precondition: $a \leq x \leq b$.
```

```c
int S.min() returns the minimal integer in the range of $S$.
```

```c
int S.max() returns the maximal integer in the range of $S$.
```

```c
void S.clear() makes $S$ the empty set.
```

In any binary operation below, $S$ and $T$ must have the same range:

```c
int_set& S.join(const int_set& T) replaces $S$ by $S \cup T$ and returns it.
```

```c
int_set& S.intersect(const int_set& T) replaces $S$ by $S \cap T$ and returns it.
```

```c
int_set& S.diff(const int_set& T) replaces $S$ by $S \setminus T$ and returns it.
```

```c
int_set& S.symdiff(const int_set& T) replaces $S$ by $(S \setminus T) \cup (T \setminus S)$ and returns it.
```

```c
int_set& S.complement() replaces $S$ by $[a..b] \setminus S$ and returns it.
```
7.10. INTEGER SETS (INT_SET)

\begin{align*}
\text{int_set} & \quad S \mid \text{const int_set}&: T \quad \text{returns the union of } S \text{ and } T. \\
\text{int_set} & \quad S \& \text{ const int_set}&: T \quad \text{returns the intersection of } S \text{ and } T. \\
\text{int_set} & \quad S - \text{ const int_set}&: T \quad \text{returns the set difference of } S \text{ and } T. \\
\text{int_set} & \quad S \% \text{ const int_set}&: T \quad \text{returns the symmetric difference of } S \text{ and } T. \\
\text{int_set} & \quad \sim S \quad \text{returns the complement of } S, \text{ i.e. } [a..b] \setminus S.
\end{align*}

4. Implementation

Integer sets are implemented by bit vectors. Operations insert, delete, member, min and max take constant time. All other operations take time $O(b - a + 1)$. 
7.11 Dynamic Integer Sets (\texttt{d\_int\_set})

1. Definition

An instance $S$ of the data type \texttt{d\_int\_set} is a subset of the integers.

```cpp
#include <LEDA/core/d_int_set.h>
```

2. Creation

\texttt{d\_int\_set \ S;} creates an instance $S$ of type \texttt{d\_int\_set} initializes it to the empty set.

3. Operations

- \texttt{int S.min( )} returns the smallest element in $S$.
  \textit{Precondition:} $S$ is not empty.

- \texttt{int S.max( )} returns the largest element in $S$.
  \textit{Precondition:} $S$ is not empty.

- \texttt{void S.insert(int x)} adds $x$ to $S$. As the sets range is expanding dynamically during insertion for the range $[S.min( ), S.max( )]$ inserting the extrema early saves repeated reallocation time.

- \texttt{void S.del(int x)} deletes $x$ from $S$.

- \texttt{bool S.member(int x)} returns true if $x$ in $S$, false otherwise.

- \texttt{int S.choose( )} returns a random element of $S$.
  \textit{Precondition:} $S$ is not empty.

- \texttt{bool S.empty( )} returns true if $S$ is empty, false otherwise.

- \texttt{int S.size( )} returns the size of $S$.

- \texttt{void S.clear( )} makes $S$ the empty set.

- \texttt{d\_int\_set S.join(const d\_int\_set& T)} returns $S \cup T$.

- \texttt{d\_int\_set S.intersect(const d\_int\_set& T)} returns $S \cap T$.

- \texttt{d\_int\_set S.diff(const d\_int\_set& T)} returns $S \setminus T$.

- \texttt{d\_int\_set S.symdiff(const d\_int\_set& T)} returns the symmetric difference of $S$ and $T$.

- \texttt{d\_int\_set S + const d\_int\_set& T} returns the union $S.join(T)$.
7.11. Dynamic Integer Sets (D_INT_SET)

- \texttt{d\_int\_set \ S - const \ d\_int\_set \& \ T} returns the difference \( S \text{.diff}(T) \).
- \texttt{d\_int\_set \ S \& const \ d\_int\_set \& \ T} returns the intersection of \( S \) and \( T \).
- \texttt{d\_int\_set \ S \mid \ const \ d\_int\_set \& \ T} returns the union \( S \text{.join}(T) \).
- \texttt{d\_int\_set \ S \% \ const \ d\_int\_set \& \ T} returns the symmetric difference \( S \text{.symdiff}(T) \).
- \texttt{d\_int\_set \& S += \ const \ d\_int\_set \& \ T} assigns \( S \text{.join}(T) \) to \( S \) and returns \( S \).
- \texttt{d\_int\_set \& S -= \ const \ d\_int\_set \& \ T} assigns \( S \text{.diff}(T) \) to \( S \) and returns \( S \).
- \texttt{d\_int\_set \& S \&= \ const \ d\_int\_set \& \ T} assigns \( S \text{.intersect}(T) \) to \( S \) and returns \( S \).
- \texttt{d\_int\_set \& S |= \ const \ d\_int\_set \& \ T} assigns \( S \text{.join}(T) \) to \( S \) and returns \( S \).
- \texttt{d\_int\_set \& S \% = \ const \ d\_int\_set \& \ T} assigns \( S \text{.symdiff}(T) \) to \( S \) and returns \( S \).
- \texttt{bool \ S != \ const \ d\_int\_set \& \ T} returns true if \( S \neq T \), false otherwise.
- \texttt{bool \ S == \ const \ d\_int\_set \& \ T} returns true if \( S = T \), false otherwise.
- \texttt{bool \ S \geq \ const \ d\_int\_set \& \ T} returns true if \( T \subseteq S \), false otherwise.
- \texttt{bool \ S \leq \ const \ d\_int\_set \& \ T} returns true if \( S \subseteq T \), false otherwise.
- \texttt{bool \ S < \ const \ d\_int\_set \& \ T} returns true if \( S \subset T \), false otherwise.
- \texttt{bool \ S > \ const \ d\_int\_set \& \ T} returns true if \( T \subset S \), false otherwise.
- \texttt{void \ S.get\_element\_list(list<\textless int}>\& \ L)} fills \( L \) with all elements stored in the set in increasing order.

**Iteration**

\texttt{forall\_elements(x, S) \{ \"the elements of \( S \) are successively assigned to \( x \)\} \}}

**4. Implementation**

Dynamic integer sets are implemented by (dynamic) bit vectors. Operations member, empty, size, min and max take constant time. The operations clear, intersection, union and complement take time \( O(b-a+1) \), where \( a = \text{max}( ) \) and \( b = \text{min}( ) \). The operations
insert and del also take time $O(b - a + 1)$, if the bit vector has to be reallocated. Otherwise they take constant time. Iterating over all elements (with the iteration macro) requires time $O(b - a + 1)$ plus the time spent in the body of the loop.
7.12 Partitions ( partition )

1. Definition
An instance $P$ of the data type partition consists of a finite set of items (partition_item) and a partition of this set into blocks.

```c
#include <LEDA/core/partition.h>
```

2. Creation

```c
partition P; creates an instance P of type partition and initializes it to the empty partition.
```

3. Operations

```c
partition_item P.make_block() returns a new partition_item it and adds the block it to partition P.
```

```c
partition_item P.find(partition_item p) returns a canonical item of the block that contains item p, i.e., iff P.same_block(p, q) then P.find(p) and P.find(q) return the same item.
Precondition: p is an item in P.
```

```c
int P.size(partition_item p) returns the size of the block containing p.
```

```c
int P.number_of_blocks() returns the number of blocks in P.
```

```c
bool P.same_block(partition_item p, partition_item q) returns true if p and q belong to the same block of partition P.
Precondition: p and q are items in P.
```

```c
void P.union_blocks(partition_item p, partition_item q) unites the blocks of partition P containing items p and q.
Precondition: p and q are items in P.
```

```c
void P.split(const list<partition_item>& L) turns all items in L to singleton blocks.
Precondition: L is a union of blocks.
```

4. Implementation
Partitions are implemented by the union find algorithm with weighted union and path compression (cf. [86]). Any sequence of $n$ make_block and $m \geq n$ other operations (except
for \textit{split} takes time $O(m \alpha(m,n))$. The cost of a split is proportional to the size of the blocks dismantled.

5. Example

Spanning Tree Algorithms (cf. section 13).
7.13 Parameterized Partitions (Partition)

1. Definition

An instance $P$ of the data type $\text{Partition}<E>$ consists of a finite set of items ($\text{partition\_item}$) and a partition of this set into blocks. Each item has an associated information of type $E$.

```plaintext
#include <LEDA/core/partition.h>
```

2. Creation

$\text{Partition}<E>\; P; \; \text{creates an instance } P \text{ of type } \text{Partition}<E> \text{ and initializes it to the empty partition.}$

3. Operations

```plaintext
\text{partition\_item \; } P.\text{make\_block(\text{const \; } E\& \; x)} \; \text{return a new \text{partition\_item it}, adds the block it to partition } P \text{, and associates } x \text{ with it.}

\text{partition\_item \; } P.\text{find(\text{partition\_item } p)} \; \text{return a canonical item of the block that contains item } p \text{, i.e., iff } P.\text{same\_block}(p, q) \text{ then } P.\text{find}(p) \text{ and } P.\text{find}(q) \text{ return the same item.}
\text{Precondition: } p \text{ is an item in } P.

\text{int \; } P.\text{size(\text{partition\_item } p)} \; \text{return the size of the block containing } p.

\text{int \; } P.\text{number\_of\_blocks()} \; \text{return the number of blocks in } P.

\text{bool \; } P.\text{same\_block(\text{partition\_item } p, \text{partition\_item } q)} \; \text{return true if } p \text{ and } q \text{ belong to the same block of partition } P.
\text{Precondition: } p \text{ and } q \text{ are items in } P.

\text{void \; } P.\text{union\_blocks(\text{partition\_item } p, \text{partition\_item } q)} \; \text{unites the blocks of partition } P \text{ containing items } p \text{ and } q.
\text{Precondition: } p \text{ and } q \text{ are items in } P.

\text{void \; } P.\text{split(\text{const list}\langle\text{partition\_item}\rangle\& \; L)} \; \text{turns all items in } L \text{ to singleton blocks.}
\text{Precondition: } L \text{ is a union of blocks}

\text{const } E\& \; P.\text{inf(\text{partition\_item it})} \; \text{return the information associated with } it.$
void change_inf(partition_item it, const E& x)
  changes the information associates with it to x.
7.14  Dynamic Trees ( dynamic_trees )

1. Definition

An instance $D$ of the data type $\text{dynamic_trees}$ is a set of dynamically changing rooted trees. Each edge is directed towards the root and has a weight. Optionally, user defined information can be stored at the vertices and at the edges.

Remark Dynamic trees are very similar to the data type $\text{tree_collection}$. The main difference is that the former use edge weights instead of the node weights used by the latter.

```c
#include <LEDA/core/dynamic_trees.h>
```

2. Creation

```c
dynamic_trees D;  // creates an empty instance D of type dynamic_trees.
```

3. Operations

- `vertex D.make(void * x = nil)` creates a tree containing a single node $v$ which is returned. The additional user defined information $x$ is stored at $v$.
- `void* D.vertex_inf(vertex v)` returns the additional user defined information at $v$.
- `void* D.edge_inf(vertex v)` returns the additional user defined information at $(v, \text{parent}(v))$ or $\text{nil}$, if $v$ has no parent.
- `vertex D.parent(vertex v)` returns the parent of $v$ in the tree or $\text{nil}$.
- `vertex D.root(vertex v)` returns the root of the tree containing $v$.
- `double D.cost(vertex v)` returns the cost of $(v, \text{parent}(v))$.
  **Precondition:** $v$ is not a tree root.
- `vertex D.mincost(vertex v)` returns vertex $w$ closest to $\text{root}(v)$ s.t. $(w, \text{parent}(w))$ has minimal cost on the path $v \rightarrow \text{root}(v)$.
  **Precondition:** $v$ is not a tree root.
- `void D.update(vertex v, double x)` adds $x$ to each edge on the path $v \rightarrow \text{root}(v)$.
- `void D.link(vertex v, vertex w, double x, void * e_inf = nil)` links the tree root $v$ (prec.) to the vertex $w$ in a different tree (prec.). The edge $e = (v, w)$ gets weight $x$, and the additional user defined information $e_{inf}$ is stored at $e$.
- `double D.cut(vertex v)` deletes the edge $(v, \text{parent}(v))$ and returns its weight.
  **Precondition:** $v$ is not a tree root.
void D.evert(vertex v) makes v the new root of its tree.

vertex D.lca(vertex v, vertex w) returns the lowest common ancestor of v and w or nil.  
Precondition: v and w are not nil.

4. Implementation

Dynamic Trees are implemented using binary trees with the randomized balancing scheme by Aragon and Seidel. Each operation takes $O(\log^2 n)$ amortized expected time except for make which takes constant time. $n$ is the current number of nodes.
7.15 Dynamic Collections of Trees (tree_collection)

1. Definition

An instance $D$ of the parameterized data type $\text{tree_collection}<I>$ is a collection of vertex disjoint rooted trees, each of whose vertices has a double-valued cost and contains an information of type $I$, called the information type of $D$.

```c
#include <LEDA/core/tree_collection.h>
```

2. Creation

```c
tree_collection<I> D;
```

creates an instance $D$ of type $\text{tree_collection}<I>$, initialized with the empty collection.

3. Operations

- $D._\text{vertex} \rightarrow \text{D.maketree}(\text{const } I& \ x)$ adds a new tree to $D$ containing a single vertex $v$ with cost zero and information $x$, and returns $v$.

- $\text{const } I& \rightarrow \text{D.inf}(\text{d_vertex } v)$ returns the information of vertex $v$.

- $d\_\text{vertex} \rightarrow \text{D.findroot}(d\_\text{vertex } v)$ returns the root of the tree containing $v$.

- $d\_\text{vertex} \rightarrow \text{D.findcost}(d\_\text{vertex } v, \text{double }& \ x)$ sets $x$ to the minimum cost of a vertex on the tree path from $v$ to $\text{findroot}(v)$ and returns the last vertex (closest to the root) on this path of cost $x$.

- $d\_\text{vertex} \rightarrow \text{D.parent}(d\_\text{vertex } v)$ returns the parent vertex of $v$.

- $\text{void} \rightarrow \text{D.addcost}(d\_\text{vertex } v, \text{double } x)$ adds double number $x$ to the cost of every vertex on the tree path from $v$ to $\text{findroot}(v)$.

- $\text{void} \rightarrow \text{D.link}(d\_\text{vertex } v, d\_\text{vertex } w)$ combines the trees containing vertices $v$ and $w$ by adding the edge $(v, w)$. (We regard tree edges as directed from child to parent.)

  Precondition: $v$ and $w$ are in different trees and $v$ is a root.

- $\text{void} \rightarrow \text{D.cut}(d\_\text{vertex } v)$ divides the tree containing vertex $v$ into two trees by deleting the edge out of $v$.

  Precondition: $v$ is not a tree root.
4. Implementation

Dynamic collections of trees are implemented by partitioning the trees into vertex disjoint paths and representing each path by a self-adjusting binary tree (see [86]). All operations take amortized time $O(\log n)$ where $n$ is the number of maketree operations.
Chapter 8

Dictionary Types

8.1 Dictionaries (dictionary)

1. Definition

An instance $D$ of the parameterized data type $\text{dictionary}<K,I>$ is a collection of items ($\text{dic} \_ \text{item}$). Every item in $D$ contains a key from the linearly ordered data type $K$, called the key type of $D$, and an information from the data type $I$, called the information type of $D$. IF $K$ is a user-defined type, you have to provide a compare function (see Section 2.3). The number of items in $D$ is called the size of $D$. A dictionary of size zero is called the empty dictionary. We use $(k,i)$ to denote an item with key $k$ and information $i$ ($i$ is said to be the information associated with key $k$). For each $k \in K$ there is at most one $i \in I$ with $(k,i) \in D$.

#include <LEDA/core/dictionary.h>

2. Types

$\text{dictionary}<K,I>::\text{item}$ the item type.

$\text{dictionary}<K,I>::\text{key} \_ \text{type}$ the key type.

$\text{dictionary}<K,I>::\text{inf} \_ \text{type}$ the information type.

3. Creation

$\text{dictionary}<K,I> \ D;$

creates an instance $D$ of type $\text{dictionary}<K,I>$ based on the linear order defined by the global compare function and initializes it with the empty dictionary.
dictionary\textless K, I\textgreater{} \ D(\text{int} (*\text{cmp})(\text{const} K \&, \text{const} K \&));

creates an instance $D$ of type dictionary\textless K, I\textgreater{} based on the linear order defined by the compare function $\text{cmp}$ and initializes it with the empty dictionary.

4. Operations

\begin{itemize}
\item \texttt{const K\&} $D.\text{key}(\text{dic\_item it})$ returns the key of item $it$.
  \textit{Precondition:} it is an item in $D$.
\item \texttt{const I\&} $D.\text{inf}(\text{dic\_item it})$ returns the information of item $it$.
  \textit{Precondition:} it is an item in $D$.
\item \texttt{I\&} $D[\text{dic\_item it}]$ returns a reference to the information of item $it$.
  \textit{Precondition:} it is an item in $D$.
\item \texttt{dic\_item} $D.\text{insert}(\text{const K\&} k, \text{const I\&} i)$ associates the information $i$ with the key $k$. If there is an item $<k, j>$ in $D$ then $j$ is replaced by $i$, else a new item $(k, i)$ is added to $D$. In both cases the item is returned.
\item \texttt{dic\_item} $D.\text{lookup}(\text{const K\&} k)$ returns the item with key $k$ (nil if no such item exists in $D$).
\item \texttt{I} $D.\text{access}(\text{const K\&} k)$ returns the information associated with key $k$.
  \textit{Precondition:} there is an item with key $k$ in $D$.
\item \texttt{void} $D.\text{del}(\text{const K\&} k)$ deletes the item with key $k$ from $D$ (null operation, if no such item exists).
\item \texttt{void} $D.\text{del\_item}(\text{dic\_item it})$ removes item $it$ from $D$.
  \textit{Precondition:} it is an item in $D$.
\item \texttt{bool} $D.\text{defined}(\text{const K\&} k)$ returns true if there is an item with key $k$ in $D$, false otherwise.
\item \texttt{void} $D.\text{undefine}(\text{const K\&} k)$ deletes the item with key $k$ from $D$ (null operation, if no such item exists).
\item \texttt{void} $D.\text{change\_inf}(\text{dic\_item it}, \text{const I\&} i)$ makes $i$ the information of item $it$.
  \textit{Precondition:} it is an item in $D$.
\item \texttt{void} $D.\text{clear}()$ makes $D$ the empty dictionary.
\end{itemize}
8.1. DICTIONARIES (DICTIONARY)

int $D$.size() returns the size of $D$.

bool $D$.empty() returns true if $D$ is empty, false otherwise.

Iteration

forall_items($it$, $D$) { "the items of $D$ are successively assigned to $it"$ }

forall_rev_items($it$, $D$) { "the items of $D$ are successively assigned to $it$ in reverse order" }

forall($i$, $D$) { "the informations of all items of $D$ are successively assigned to $i"$ }

forall_defined($k$, $D$) { "the keys of all items of $D$ are successively assigned to $k"$ }

STL compatible iterators are provided when compiled with -DLEDA_STL_ITERATORS (see LEDAROOT/demo/stl/dic.c for an example).

5. Implementation

Dictionaries are implemented by (2, 4)-trees. Operations insert, lookup, del_item, del take time $O(\log n)$, key, inf, empty, size, change_inf take time $O(1)$, and clear takes time $O(n)$. Here $n$ is the current size of the dictionary. The space requirement is $O(n)$.

6. Example

We count the number of occurrences of each string in a sequence of strings.

```c
#include <LEDA/core/dictionary.h>

main()
{
dictionary<string,int> D;
    string s;
    dic_item it;

    while (cin >> s)
    {
        it = D.lookup(s);
        if (it==nil) D.insert(s,1);
        else D.change_inf(it,D.inf(it)+1);
    }

   forall_items(it,D) cout << D.key(it) << " : " << D.inf(it) << endl;
}
```
8.2 Dictionary Arrays (d_array)

1. Definition

An instance \( A \) of the parameterized data type \( d\text{\_array}<I,E> \) (dictionary array) is an injective mapping from the linearly ordered data type \( I \), called the index type of \( A \), to the set of variables of data type \( E \), called the element type of \( A \). We use \( A(i) \) to denote the variable with index \( i \) and we use \( \text{dom}(A) \) to denote the set of “used indices”. This set is empty at the time of creation and is modified by array accesses. Each dictionary array has an associated default value \( x\text{\_def} \). The variable \( A(i) \) has value \( x\text{\_def} \) for all \( i \notin \text{dom}(A) \). If \( I \) is a user-defined type, you have to provide a compare function (see Section 2.3).

Related data types are \( h\text{\_arrays} \), maps, and dictionaries.

```cpp
#include <LEDA/core/d_array.h>
```

2. Types

- \( d\text{\_array}<I,E>::\text{item} \) the item type.
- \( d\text{\_array}<I,E>::\text{index\_type} \) the index type.
- \( d\text{\_array}<I,E>::\text{element\_type} \) the element type.

3. Creation

- \( d\text{\_array}<I,E> \quad A; \) creates an injective function \( a \) from \( I \) to the set of unused variables of type \( E \), sets \( x\text{\_def} \) to the default value of type \( E \) (if \( E \) has no default value then \( x\text{\_def} \) stays undefined) and \( \text{dom}(A) \) to the empty set, and initializes \( A \) with \( a \).
- \( d\text{\_array}<I,E> \quad A(E \ x); \) creates an injective function \( a \) from \( I \) to the set of unused variables of type \( E \), sets \( x\text{\_def} \) to \( x \) and \( \text{dom}(A) \) to the empty set, and initializes \( A \) with \( a \).

4. Operations

- \( E\& \quad A[\text{\_const} \ I\& \ i] \) returns the variable \( A(i) \).
- \( \text{bool} \quad A.\text{defined}(\text{\_const} \ I\& \ i) \) returns true if \( i \in \text{dom}(A) \) and false otherwise.
- \( \text{void} \quad A.\text{undefine}(\text{\_const} \ I\& \ i) \) removes \( i \) from \( \text{dom}(A) \) and sets \( A(i) \) to \( x\text{\_def} \).
- \( \text{void} \quad A.\text{clear()} \) makes \( \text{dom}(A) \) empty.
8.2. DICTIONARY ARRAYS (D_ARRAY)

int A.size() returns |dom(A)|.

void A.set_default_value(const E& x)
    sets xdef to x.

Iteration

forall_defined(i, A) { “the elements from dom(A) are successively assigned to i” }
forall(x, A) { “for all i ∈ dom(A) the entries A[i] are successively assigned to x” }

5. Implementation

Dictionary arrays are implemented by (2,4)-trees [58]. Access operations A[i] take time $O(\log \text{dom}(A))$. The space requirement is $O(\text{dom}(A))$.

6. Example

Program 1:

We use a dictionary array to count the number of occurrences of the elements in a sequence of strings.

```cpp
#include <LEDA/core/d_array.h>

main()
{
    d_array<string,int> N(0);
    string s;

    while (cin >> s) N[s]++;

    forall_defined(s,N) cout << s << " " << N[s] << endl;
}
```

Program 2:

We use a $\text{d_array}<\text{string}, \text{string}>$ to realize an english/german dictionary.

```cpp
#include <LEDA/core/d_array.h>

main()
```
```cpp
{  
    d_array<string,string> dic;

    dic["hello"] = "hallo";
    dic["world"] = "Welt";
    dic["book"] = "Buch";
    dic["key"] = "Schluessel";

    string s;
    forall_defined(s,dic) cout << s << " " << dic[s] << endl;
}
```
8.3 Hashing Arrays (h_array)

1. Definition

An instance $A$ of the parameterized data type $h\text{\_array}<I, E>$ (hashing array) is an injective mapping from a hashed data type $I$, called the index type of $A$, to the set of variables of arbitrary type $E$, called the element type of $A$. We use $A(i)$ to denote the variable indexed by $i$ and we use $\text{dom}(A)$ to denote the set of “used indices”. This set is empty at the time of creation and is modified by array accesses. Each hashing array has an associated default value $x\text{def}$. The variable $A(i)$ has value $x\text{def}$ for all $i \notin \text{dom}(A)$. If $I$ is a user-defined type, you have to provide a Hash function (see Section 2.3).

Related data types are $d\text{\_arrays}$, maps, and dictionaries.

#include <LEDA/core/h_array.h>

2. Creation

$h\text{\_array}<I, E> A;$ creates an injective function $a$ from $I$ to the set of unused variables of type $E$, sets $x\text{def}$ to the default value of type $E$ (if $E$ has no default value then $x\text{def}$ stays undefined) and $\text{dom}(A)$ to the empty set, and initializes $A$ with $a$.

$h\text{\_array}<I, E> A(E \ x);$ creates an injective function $a$ from $I$ to the set of unused variables of type $E$, sets $x\text{def}$ to $x$ and $\text{dom}(A)$ to the empty set, and initializes $A$ with $a$.

$h\text{\_array}<I, E> A(E \ x, \text{int} \ \text{table\_sz});$ as above, but uses an initial table size of $\text{table\_sz}$ instead of the default size 1.

3. Operations

$E\& \hspace{1cm} A[\text{const} \ I\& \ i]$ returns the variable $A(i)$.

$bool \hspace{1cm} A.\text{defined}(\text{const} \ I\& \ i)$ returns true if $i \in \text{dom}(A)$ and false otherwise.

$void \hspace{1cm} A.\text{undefine}(\text{const} \ I\& \ i)$ removes $i$ from $\text{dom}(A)$ and sets $A(i)$ to $x\text{def}$.

$void \hspace{1cm} A.\text{clear}()$ makes $\text{dom}(A)$ empty.

$void \hspace{1cm} A.\text{clear}(\text{const} \ E\& \ x)$ makes $\text{dom}(A)$ empty and sets $x\text{def}$ to $x$.

$int \hspace{1cm} A.\text{size}()$ returns $|\text{dom}(A)|$. 
bool A.empty() returns true if \( A \) is empty, false otherwise.

\[\text{void A.set_default_value(const E & x)}\]
\hspace{1cm} sets \( x \text{def} \) to \( x \).

**Iteration**

forall\_defined\((i, A)\) \{ “the elements from \( \text{dom}(A) \) are successively assigned to \( i \)” \}

Remark: the current element may not be deleted resp. declared undefined during execution of the loop.

forall\((x, A)\) \{ “for all \( i \in \text{dom}(A) \) the entries \( A[i] \) are successively assigned to \( x \)” \}.

**4. Implementation**

Hashing arrays are implemented by hashing with chaining. Access operations take expected time \( O(1) \). In many cases, hashing arrays are more efficient than dictionary arrays (cf. 8.2).
8.4 Maps (map)

1. Definition
An instance \( M \) of the parameterized data type \( \text{map}<I, E> \) is an injective mapping from
the data type \( I \), called the index type of \( M \), to the set of variables of data type \( E \), called
the element type of \( M \). \( I \) must be a pointer, item, or handle type or the type int. We use
\( M(i) \) to denote the variable indexed by \( i \). All variables are initialized to \( xdef \), an element
of \( E \) that is specified in the definition of \( M \). A subset of \( I \) is designated as the domain of
\( M \). Elements are added to \( \text{dom}(M) \) by the subscript operator; however, the domain may
also contain indices for which the access operator was never executed.

Related data types are \( d\text{arrays}, h\text{arrays}, \) and dictionaries.

```c
#include <LEDA/core/map.h>
```

2. Types
\( \text{map}<I, E> ::= \text{item} \) the item type.
\( \text{map}<I, E> ::= \text{index}\_\text{type} \) the index type.
\( \text{map}<I, E> ::= \text{element}\_\text{type} \) the element type.

3. Creation
\( \text{map}<I, E> M; \) creates an injective function \( m \) from \( I \) to the set of unused
variables of type \( E \), sets \( xdef \) to the default value of type \( E \)
(if \( E \) has no default value then \( xdef \) is set to an unspecified
element of \( E \)), and initializes \( M \) with \( m \).

\( \text{map}<I, E> M(E\ x); \) creates an injective function \( m \) from \( I \) to the set of unused
variables of type \( E \), sets \( xdef \) to \( x \), and initializes \( M \) with \( m \).

\( \text{map}<I, E> M(E\ x, \text{int }\table\_\text{sz}); \)
as above, but uses an initial table size of \( \table\_\text{sz} \) instead of
the default size 1.

4. Operations
\( E& \quad M[\text{const }I&\ i] \) returns the variable \( M(i) \) and adds \( i \) to \( \text{dom}(M) \).
If \( M \) is a const-object then \( M(i) \) is read-only and
\( i \) is not added to \( \text{dom}(M) \).

\( \text{bool} \quad M.\text{defined(\text{const }I&\ i)} \) returns true if \( i \in \text{dom}(M) \).

\( \text{void} \quad M.\text{clear( )} \) makes \( M \) empty.

\( \text{void} \quad M.\text{clear(\text{const }E&\ x)} \) makes \( M \) empty and sets \( xdef \) to \( x \).

\( \text{void} \quad M.\text{set}\_\text{default}\_\text{value(\text{const }E&\ x)} \)
sets \( xdef \) to \( x \).

\( E \quad M.\text{get}\_\text{default}\_\text{value( )} \) returns the default value \( xdef \).

Iteration:
\( \text{forall}(x, M) \{ \text{"the entries } M[i] \text{ with } i \in \text{dom}(M) \text{ are successively assigned to } x" \} \)
Note that it is *not* possible to iterate over the indices in $dom(M)$. If you need this feature use the type $h\_array$ instead.

5. Implementation
Maps are implemented by hashing with chaining and table doubling. Access operations $M[i]$ take expected time $O(1)$. 
8.5 Two-Dimensional Maps (map2)

1. Definition

An instance $M$ of the parameterized data type $\text{map}2<\mathcal{I}_1, \mathcal{I}_2, E>$ is an injective mapping from the pairs in $\mathcal{I}_1 \times \mathcal{I}_2$, called the index type of $M$, to the set of variables of data type $E$, called the element type of $M$. $I$ must be a pointer, item, or handle type or the type int. We use $M(i, j)$ to denote the variable indexed by $(i, j)$ and we use $\text{dom}(M)$ to denote the set of “used indices”. This set is empty at the time of creation and is modified by $\text{map2}$ accesses.

Related data types are $\text{map}$, $\text{d_arrays}$, $\text{h_arrays}$, and $\text{dictionaries}$.

```cpp
#include <LEDA/core/map2.h>
```

2. Types

$\text{map}2<\mathcal{I}_1, \mathcal{I}_2, E>::\text{item}$ the item type.

$\text{map}2<\mathcal{I}_1, \mathcal{I}_2, E>::\text{index}\_\text{type1}$ the first index type.

$\text{map}2<\mathcal{I}_1, \mathcal{I}_2, E>::\text{index}\_\text{type2}$ the second index type.

$\text{map}2<\mathcal{I}_1, \mathcal{I}_2, E>::\text{element}\_\text{type}$ the element type.

3. Creation

$\text{map}2<\mathcal{I}_1, \mathcal{I}_2, E>\ M;$ creates an injective function $m$ from $\mathcal{I}_1 \times \mathcal{I}_2$ to the set of unused variables of type $E$, sets $x\text{def}$ to the default value of type $E$ (if $E$ has no default value then $x\text{def}$ stays undefined) and $\text{dom}(M)$ to the empty set, and initializes $M$ with $m$.

$\text{map}2<\mathcal{I}_1, \mathcal{I}_2, E>\ M(E\ x);$ creates an injective function $m$ from $\mathcal{I}_1 \times \mathcal{I}_2$ to the set of unused variables of type $E$, sets $x\text{def}$ to $x$ and $\text{dom}(M)$ to the empty set, and initializes $M$ with $m$.

4. Operations

$E& \quad M(\text{const}\ \mathcal{I}_1&\ i, \ \text{const}\ \mathcal{I}_2&\ j)$

returns the variable $M(i)$.

$\text{bool} \quad M.\text{defined(}\text{const}\ \mathcal{I}_1&\ i, \ \text{const}\ \mathcal{I}_2&\ j)$

returns true if $i \in \text{dom}(M)$ and false otherwise.


void M.clear() clears $M$ by making $\text{dom}(M)$ the empty set.

5. Implementation

Maps are implemented by hashing with chaining and table doubling. Access operations $M(i,j)$ take expected time $O(1)$.
8.6 Persistent Dictionaries (p_dictionary)

1. Definition

An instance $D$ of the parameterized data type $\textit{p_dictionary}<K,I>$ is a set of items (type $\textit{p_dic_item}$). Every item in $D$ contains a key from the linearly ordered data type $K$, called the key type of $D$, and an information from data type $I$, called the information type of $D$. If $K$ is a user-defined type, you have to provide a compare function (see Section 2.3). The number of items in $D$ is called the size of $D$. A dictionary of size zero is called empty. We use $\langle k, i \rangle$ to denote an item with key $k$ and information $i$ ($i$ is said to be the information associated with key $k$). For each $k \in K$ there is at most one item $\langle k, i \rangle \in D$.

The difference between dictionaries (cf. section 8.1) and persistent dictionaries lies in the fact that update operations performed on a persistent dictionary $D$ do not change $D$ but create and return a new dictionary $D'$. For example, $D\textit{.del}(k)$ returns the dictionary $D'$ containing all items $it$ of $D$ with $\text{key}(it) \neq k$. Also, an assignment $D1 = D2$ does not assign a copy of $D2$ (with new items) to $D1$ but the value of $D2$ itself.

```c
#include <LEDA/core/p_dictionary.h>
```

2. Creation

$\textit{p_dictionary}<K,I> \ D;$

creates an instance $D$ of type $\textit{p_dictionary}<K,I>$ and initializes $D$ to an empty persistent dictionary.

3. Operations

- $K$ $\textit{D.key(p_dic_item it)}$ returns the key of item $it$.  
  \textit{Precondition:} $it \in D$.

- $I$ $\textit{D.inf(p_dic_item it)}$ returns the information of item $it$.  
  \textit{Precondition:} $it \in D$.

- $\textit{p_dic_item}$ $\textit{D.locate(const K& k)}$ returns the item with key $k$ (nil if no such item exists in $D$).

- $\textit{p_dictionary}<K,I>$ $\textit{D.del(const K& k)}$ returns $\{x \in D | \text{key}(x) \neq k\}$.

- $\textit{p_dictionary}<K,I>$ $\textit{D.del_item(p_dic_item it)}$ returns $\{x \in D | x \neq it\}$.

- $\textit{p_dictionary}<K,I>$ $\textit{D.insert(const K& k, const I& i)}$ returns $D.\textit{del}(k) \cup \{\langle k, i \rangle\}$.
p\_dictionary<K, I> \quad D.\text{change\_inf}(p\_dic\_item\ it, \ const\ I\&\ i) \quad \text{returns } D.\text{del\_item}(it) \cup \{\langle k, i \rangle \}, \text{ where } k = key(it). \quad \text{Precondition: } it \in D.

\begin{align*}
\text{int} \quad &D.\text{size}(\) \quad \text{returns the size of } D. \\
\text{bool} \quad &D.\text{empty}(\) \quad \text{returns true if } D \text{ is empty, false otherwise.}
\end{align*}

4. Implementation

Persistent dictionaries are implemented by leaf oriented persistent red black trees. Operations insert, lookup, del\_item, del take time $O(\log^2 n)$, key, inf, empty, size and change\_inf take time $O(1)$. The space requirement is $O(1)$ for each update operation.
8.7 Partially Persistent Dictionaries (pp_dictionary)

1. Definition

A partially persistent dictionary stores the history of a set of items \(<k, i>\) (type pp_dic_item). Here \(k\) is a key from a linearly ordered type \(K\), and \(i\) is an information from the data type \(I\). An instance \(D\) of the parameterized data type \(pp\_dictionary<K, I, CMP>\) references a version of the set in the history.

An update operation may only be performed on an instance \(D\) containing the newest version \(v\). The operation changes the set of items and creates a new version \(v'\) which represents the new state of the set. After the operation, \(D\) references version \(v'\), and all other instances which have referenced version \(v\) remain unchanged. An access operation may be applied to any instance \(D\). Let \(v\) denote the version referenced by \(D\). Then the operation is carried out on the set which existed when \(v\) was created.

The main advantage of a partially persistent dictionary compared to a dictionary is that copy and assignment operations use only constant time and space, because only the versions and not the whole sets are assigned.

The linear order for the data type \(K\) is given by a compare object of type \(CMP\). It must provide a function call operator which has the same syntax and semantics as a LEDA compare function (see Section 2.3). The data type \(pp\_dictionary\) may be instantiated without specifying the third template argument, i.e. as type \(pp\_dictionary<K, I>\). This data type uses the linear order which is defined by the compare function for the type \(K\).

```cpp
#include <LEDA/core/pp_dictionary.h>
```

2. Creation

\[pp\_dictionary<K, I, CMP>\ D(\text{const } CMP & cmp = CMP( ));\]

creates an instance \(D\) of type \(pp\_dictionary<K, I, CMP>\) and initializes \(D\) to an empty persistent dictionary.

When the argument \(cmp\) is provided, the object is used as compare function object which defines the linear order for the data type \(K\).

3. Operations

3.1 Update operations

\[pp\_dic\_item\ D.insert(\text{const } K & k, \text{const } I & i)\]

associates the information \(i\) with the key \(k\).

If there is an item \(<k, j>\) in the set then it is deleted. A new item \(<k, i>\) is added and returned.
CHAPTER 8. DICTIONARY TYPES

void D.del(const K& k) deletes the item with key k from the set (null operation, if no such item exists).

void D.del_item(pp_dic_item it) removes item it from the set.
Precondition: it is an item in the set.

pp_dic_item D.change_inf(pp_dic_item it, const I& i) makes i the information of item it. This may create a new item; the item <key(it), i> in the new set is returned.
Precondition: it is an item in the set.

void D.clear() makes D the empty dictionary.

3.2 Access operations

const K& D.key(pp_dic_item it) returns the key of item it.
Precondition: it ∈ D.

const I& D.inf(pp_dic_item it) returns the information of item it.
Precondition: it ∈ D.

pp_dic_item D.lookup(const K& k) returns the item with key k (nil if no such item exists in D).

pp_dic_item D.locate_succ(const K& k) returns the item x in D so that key(x) = min{key(y)| y ∈ D ∧ key(y) ≥ k}.

pp_dic_item D.locate_pred(const K& k) returns the item x in D so that key(x) = max{key(y)| y ∈ D ∧ key(y) ≤ k}.

int D.size() returns the size of D.

bool D.empty() returns true if D is empty, false otherwise.

4. Implementation

Partially persistent dictionaries are implemented by leaf oriented partially persistent red-black trees. Operations insert, del, del_item and lookup take expected time $O(\log n)$, the expected time for key, inf, empty, size and change_inf is $O(1)$, and clear takes expected time $O(n)$. Here $n$ is the number of items in the version of the dictionary to which the operation is applied. The expected space requirement is $O(1)$ for each update operation.
8.8 Sorted Sequences (sortseq)

1. Definition

An instance $S$ of the parameterized data type $\text{sortseq}<K, I>$ is a sequence of items ($\text{seq\_item}$). Every item contains a key from a linearly ordered data type $K$, called the key type of $S$, and an information from a data type $I$, called the information type of $S$. If $K$ is a user-defined type, you have to provide a compare function (see Section 2.3). The number of items in $S$ is called the size of $S$. A sorted sequence of size zero is called empty. We use $\langle k, i \rangle$ to denote a $\text{seq\_item}$ with key $k$ and information $i$ (called the information associated with key $k$). For each $k$ in $K$ there is at most one item $\langle k, i \rangle$ in $S$ and if item $\langle k_1, i_1 \rangle$ precedes item $\langle k_2, i_2 \rangle$ in $S$ then $k_1 < k_2$.

Sorted sequences are a very powerful data type. They can do everything that dictionaries and priority queues can do. They also support many other operations, in particular finger searches and operations concat, split, merge, reverse\_items, and delete\_subsequence.

The key type $K$ must be linearly ordered. The linear order on $K$ may change over time subject to the condition that the order of the elements that are currently in the sorted sequence remains stable. More precisely, whenever an operation (except for reverse\_items) is applied to a sorted sequence $S$, the keys of $S$ must form an increasing sequence according to the currently valid linear order on $K$. For operation reverse\_items this must hold after the execution of the operation. An application of sorted sequences where the linear order on the keys evolves over time is the plane sweep algorithm for line segment intersection. This algorithm sweeps an arrangement of segments by a vertical sweep line and keeps the intersected segments in a sorted sequence sorted according to the $y$-coordinates of their intersections with the sweep line. For intersecting segments this order depends on the position of the sweep line.

Sorted sequences support finger searches. A finger search takes an item $it$ in a sorted sequence and a key $k$ and searches for the key in the sorted sequence containing the item. The cost of a finger search is proportional to the logarithm of the distance of the key from the start of the search. A finger search does not need to know the sequence containing the item. We use $IT$ to denote the sequence containing $it$. In a call $S$.finger\_search($it$, $k$) the types of $S$ and $IT$ must agree but $S$ may or may not be the sequence containing $it$.

```c
#include <LEDA/core/sortseq.h>
```

2. Types

$\text{sortseq}<K, I>:: \text{item}$ the item type $\text{seq\_item}$.

$\text{sortseq}<K, I>:: \text{key\_type}$ the key type $K$.

$\text{sortseq}<K, I>:: \text{inf\_type}$ the information type $I$. 
3. Creation

\texttt{sortseq<K, I> S;}

creates an instance \( S \) of type \( \text{sortseq}<K, I> \) based on the linear order defined by the global \texttt{compare} function and and initializes it to the empty sorted sequence.

\( \text{sortseq}<K, I> S(\text{int } (*\text{cmp})(\text{const } K\&, \text{const } K\&))\);

creates an instance \( S \) of type \( \text{sortseq}<K, I> \) based on the linear order defined by the compare function \( \text{cmp} \) and initializes it with the empty sorted sequence.

4. Operations

\texttt{const K\& S.key(seq_item it)} returns the key of item \( it \).

\texttt{const I\& S.inf(seq_item it)} returns the information of item \( it \).

\texttt{I\& S[seq_item it]} returns a reference to the information of item \( it \).

\texttt{Precondition: it is an item in S.}

\texttt{seq_item S.lookup(const K\& k)} returns the item with key \( k \) (\textit{nil} if there is no such item).

\texttt{seq_item S.finger\_lookup(const K\& k)} equivalent to \( S.\text{lookup}(k) \)

\texttt{seq_item S.finger\_lookup\_from\_front(const K\& k)} equivalent to \( S.\text{lookup}(k) \)

\texttt{seq_item S.finger\_lookup\_from\_rear(const K\& k)} equivalent to \( S.\text{lookup}(k) \)

\texttt{seq_item S.locate(const K\& k)} returns the item \( \langle k1, i \rangle \) in \( S \) such that \( k1 \geq k \) (\textit{nil} if no such item exists).

\texttt{seq_item S.finger\_locate(const K\& k)} equivalent to \( S.\text{locate}(k) \)

\texttt{seq_item S.finger\_locate\_from\_front(const K\& k)} equivalent to \( S.\text{locate}(k) \)

\texttt{seq_item S.finger\_locate\_from\_rear(const K\& k)} equivalent to \( S.\text{locate}(k) \)

\texttt{seq_item S.locate\_succ(const K\& k)} equivalent to \( S.\text{locate}(k) \)
**seq_item** \( S.\text{succ}(\text{const } K \& k) \) equivalent to \( S.\text{locate}(k) \)

**seq_item** \( S.\text{finger}\_\text{locate}\_\text{succ}(\text{const } K \& k) \) equivalent to \( S.\text{locate}(k) \)

**seq_item** \( S.\text{finger}\_\text{locate}\_\text{succ}\_\text{from}\_\text{front}(\text{const } K \& k) \) equivalent to \( S.\text{locate}(k) \)

**seq_item** \( S.\text{finger}\_\text{locate}\_\text{succ}\_\text{from}\_\text{rear}(\text{const } K \& k) \) equivalent to \( S.\text{locate}(k) \)

**seq_item** \( S.\text{locate}\_\text{pred}(\text{const } K \& k) \) returns the item \( \langle k1, i \rangle \) in \( S \) such that \( k1 \) is maximal with \( k1 \leq k \) (\( \text{nil} \) if no such item exists).

**seq_item** \( S.\text{pred}(\text{const } K \& k) \) equivalent to \( S.\text{locate}\_\text{pred}(k) \)

**seq_item** \( S.\text{finger}\_\text{locate}\_\text{pred}(\text{const } K \& k) \) equivalent to \( S.\text{locate}\_\text{pred}(k) \)

**seq_item** \( S.\text{finger}\_\text{locate}\_\text{pred}\_\text{from}\_\text{front}(\text{const } K \& k) \) equivalent to \( S.\text{locate}\_\text{pred}(k) \)

**seq_item** \( S.\text{finger}\_\text{locate}\_\text{pred}\_\text{from}\_\text{rear}(\text{const } K \& k) \) equivalent to \( S.\text{locate}\_\text{pred}(k) \)

**seq_item** \( S.\text{finger}\_\text{lookup}(\text{seq_item } it, \text{const } K \& k) \) equivalent to \( IT.\text{lookup}(k) \) where \( IT \) is the sorted sequence containing \( it \).

Precondition: \( S \) and \( IT \) must have the same type

**seq_item** \( S.\text{finger}\_\text{locate}(\text{seq_item } it, \text{const } K \& k) \) equivalent to \( IT.\text{locate}(k) \) where \( IT \) is the sorted sequence containing \( it \).

Precondition: \( S \) and \( IT \) must have the same type.

**seq_item** \( S.\text{finger}\_\text{locate}\_\text{succ}(\text{seq_item } it, \text{const } K \& k) \) equivalent to \( IT.\text{locate}\_\text{succ}(k) \) where \( IT \) is the sorted sequence containing \( it \).

Precondition: \( S \) and \( IT \) must have the same type.

**seq_item** \( S.\text{finger}\_\text{locate}\_\text{pred}(\text{seq_item } it, \text{const } K \& k) \) equivalent to \( IT.\text{locate}\_\text{pred}(k) \) where \( IT \) is the sorted sequence containing \( it \).

Precondition: \( S \) and \( IT \) must have the same type.

**seq_item** \( S.\text{min}\_\text{item}() \) returns the item with minimal key (\( \text{nil} \) if \( S \) is empty).
**seq_item S.max_item( )**
returns the item with maximal key (*nil* if *S* is empty).

**seq_item S.succ(seq_item it)**
returns the successor item of *it* in the sequence containing *it* (*nil* if there is no such item).

**seq_item S.pred(seq_item x)**
returns the predecessor item of *it* in the sequence containing *it* (*nil* if there is no such item).

**seq_item S.insert(const K& k, const I& i)**
associates information *i* with key *k*: If there is an item ⟨*k, j*⟩ in *S* then *j* is replaced by *i*, else a new item ⟨*k, i*⟩ is added to *S*. In both cases the item is returned.

**seq_item S.insert_at(seq_item it, const K& k, const I& i)**
Like *IT.insert(k, i)* where *IT* is the sequence containing item *it*.
**Precondition:** *it* is an item in *IT* with
*key(it)* is maximal with *key(it) < k* or
*key(it)* is minimal with *key(it) > k* or
if *key(it) = k* then inf(*it*) is replaced by *i*. *S* and *IT* have the same type.

**seq_item S.insert_at(seq_item it, const K& k, const I& i, int dir)**
Like *IT.insert(k, i)* where *IT* is the sequence containing item *it*.
**Precondition:** *it* is an item in *IT* with
*key(it)* is maximal with *key(it) < k* if *dir* = *leda::before* or
*key(it)* is minimal with *k < key(it)* if *dir* = *leda::behind* or
if *key(it) = k* then inf(*it*) is replaced by *i*. *S* and *IT* have the same type.

**int S.size( )**
returns the size of *S*.

**bool S.empty( )**
returns true if *S* is empty, false otherwise.

**void S.clear( )**
makes *S* the empty sorted sequence.

**void S.reverse_items(seq_item a, seq_item b)**
the subsequence of *IT* from *a* to *b* is reversed, where
*IT* is the sequence containing *a* and *b*.
**Precondition:** *a* appears before *b* in *IT*.

**void S.flip_items(seq_item a, seq_item b)**
equivalent to *S.reverse_items(a, b).*
void S.del(const K& k)
removes the item with key k from S (null operation if no such item
exists).

void S.del_item(seq_item it)
removes the item it from the sequence containing it.

void S.change_inf(seq_item it, const I& i)
makes i the information of item it.

void S.split(seq_item it, sortseq<K, I, seq_impl>& S1, sortseq<K, I, seq_impl>& S2,
int dir = leda::behind)
splits IT at item it, where IT is the sequence containing it, into se-
quences S1 and S2 and makes IT empty (if distinct from S1 and
S2). More precisely, if IT = x1, . . . , xk−1, it, xk+1, . . . , xn and dir =
leda::behind then S1 = x1, . . . , xk−1, it and S2 = xk+1, . . . , xn. If
dir = leda::before then S2 starts with it after the split.

void S.delete_subsequence(seq_item a, seq_item b, sortseq<K, I, seq_impl>& S1)
deletes the subsequence starting at a and ending at b from the sequence
IT containing both and assigns the subsequence to S1.
Precondition: a and b belong to the same sequence IT, a is equal to or
before b and IT and S1 have the same type.

sortseq<K, I, seq_impl>& S.conc(sortseq<K, I, seq_impl>& S1, int dir = leda::behind)
appends S1 at the front (dir = leda::before) or rear (dir =
leda::behind) end of S, makes S1 empty and returns S.
Precondition: S.key(S.max_item( )) < S1.key(S1.min_item( )) if dir =
leda::behind and S1.key(S1.max_item( )) < S.key(S.min_item( ) if
dir = leda::before.

void S.merge(sortseq<K, I, seq_impl>& S1)
merges the sequence S1 into sequence S and makes S1 empty.
Precondition: all keys are distinct.

void S.print(ostream& out, string s, char c = ' ')
prints s and all elements of S separated by c onto stream out.

void S.print(string s, char c = ' ')
equivalent to S.print(cout, s, c).

bool S == const sortseq<K, I, seq_impl>& S1
returns true if S agrees with S1 componentwise and false otherwise.

sortseq<K, I, seq_impl>* sortseq<K, I>::my_sortseq(seq_item it)
returns a pointer to the sortseq containing it.
Precondition: The type of the sortseq containing it must be
sortseq<K, I>.
Iteration

forall_items(it, S) { “the items of S are successively assigned to it” }

forall_rev_items(it, S) { “the items of S are successively assigned to it in reverse order” }

forall(i, S) { “the informations of all items of S are successively assigned to i” }

forall_defined(k, S) { “the keys of all items of S are successively assigned to k” }

5. Implementation

Sorted sequences are implemented by skiplists [77]. Let $n$ denote the current size of the sequence. Operations insert, locate, lookup and del take time $O(\log n)$, operations succ, pred, max, min_item, key, inf, insert_at and del_item take time $O(1)$. clear takes time $O(n)$ and reverse_items $O(l)$, where $l$ is the length of the reversed subsequence. Finger_lookup$(x)$ and finger_locate$(x)$ take time $O(\log \min(d, n - d))$ if $x$ is the $d$-th item in $S$. Finger_lookup_from_front$(x)$ and finger_locate_from_front$(x)$ take time $O(\log d)$ if $x$ is the $d$-th item in $S$. Finger_lookup_from_rear$(x)$ and finger_locate_from_rear$(x)$ take time $O(\log \min(d, n - d))$ if $x$ is the $n-d$-th item in $S$. Finger_lookup$(it, x)$ and finger_locate$(it, x)$ take time $O(\log \min(d, n - d))$ where $d$ is the number of items between $it$ and the item containing $x$. Note that $\min(d, n - d)$ is the smaller of the distances from $it$ to $x$ if sequences are viewed as circularly closed. Split, delete_subsequence and conc take time $O(\log \min(n_1, n_2))$ where $n_1$ and $n_2$ are the sizes of the results of split and delete_subsequence and the arguments of conc respectively. Merge takes time $O(\log ((n_1 + n_2)/n_1))$ where $n_1$ and $n_2$ are the sizes of the two arguments. The space requirement of sorted sequences is linear in the length of the sequence (about $25.5n$ Bytes for a sequence of size $n$ plus the space for the keys and the informations.).

6. Example

We use a sorted sequence to list all elements in a sequence of strings lying lexicographically between two given search strings.

```c++
#include <LEDA/core/sortseq.h>
#include <iostream>

using leda::sortseq;
using leda::string;
using leda::seq_item;
using std::cin;
using std::cout;
```
int main()
{
    sortseq<string, int> S;
    string s1, s2;

    cout << "Input a sequence of strings terminated by 'STOP'\n";
    while (cin >> s1 && s1 != "STOP")
        S.insert(s1, 0);

    while (true) {
        cout << "\n\nInput a pair of strings: \n";
        cin >> s1 >> s2;
        cout << "All strings s with " << s1 << " <= s <= " << s2 << " :\n";
        if (s2 < s1) continue;
        seq_item last = S.locate_pred(s2);
        seq_item first = S.locate(s1);
        if (!first || !last || first == S.succ(last)) continue;
        seq_item it = first;
        while (true) {
            cout << "\n" << S.key(it);
            if (it == last) break;
            it = S.succ(it);
        }
    }
}

Further examples can be found in section Sorted Sequences of [64].
Chapter 9

Priority Queues

9.1 Priority Queues (p_queue)

1. Definition

An instance $Q$ of the parameterized data type $\text{p_queue}<P,I>$ is a collection of items (type $\text{pq\_item}$). Every item contains a priority from a linearly ordered type $P$ and an information from an arbitrary type $I$. $P$ is called the priority type of $Q$ and $I$ is called the information type of $Q$. If $P$ is a user-defined type, you have to define the linear order by providing the compare function (see Section 2.3). The number of items in $Q$ is called the size of $Q$. If $Q$ has size zero it is called the empty priority queue. We use $(p,i)$ to denote a $\text{pq\_item}$ with priority $p$ and information $i$.

Remark: Iteration over the elements of $Q$ using iteration macros such as forall is not supported.

#include <LEDACore/p_queue.h>

2. Types

$p\_queue<P,I>::\text{item}$ the item type.

$p\_queue<P,I>::\text{prio\_type}$ the priority type.

$p\_queue<P,I>::\text{inf\_type}$ the information type.

3. Creation

$p\_queue<P,I> Q;$ creates an instance $Q$ of type $p\_queue<P,I>$ based on the linear order defined by the global compare function $\text{compare}(\text{const } P\&, \text{ const } P\&)$ and initializes it with the empty priority queue.
CHAPTER 9. PRIORITY QUEUES

\[
p\text{-queue}<P, I> \quad Q(int (*cmp)(const P&, const P&));
\]
creates an instance \( Q \) of type \( p\text{-queue}<P, I> \) based on the linear order defined by the compare function \( cmp \) and initializes it with the empty priority queue. **Precondition:** \( cmp \) must define a linear order on \( P \).

4. Operations

- \( \text{const } P& \quad Q.\text{prio}(pq\text{\_item } it) \) returns the priority of item \( it \).
  **Precondition:** \( it \) is an item in \( Q \).

- \( \text{const } I& \quad Q.\text{inf}(pq\text{\_item } it) \) returns the information of item \( it \).
  **Precondition:** \( it \) is an item in \( Q \).

- \( I& \quad Q[pq\text{\_item } it] \) returns a reference to the information of item \( it \).
  **Precondition:** \( it \) is an item in \( Q \).

- \( pq\text{\_item} \quad Q.\text{insert}(\text{const } P& x, \text{const } I& i) \) adds a new item \(<x, i>\) to \( Q \) and returns it.

- \( pq\text{\_item} \quad Q.\text{find\_min}() \) returns an item with minimal priority (nil if \( Q \) is empty).

- \( P \quad Q.\text{del\_min}() \) removes the item \( it = Q.\text{find\_min}() \) from \( Q \) and returns the priority of it.
  **Precondition:** \( Q \) is not empty.

- \( \text{void} \quad Q.\text{del\_item}(pq\text{\_item } it) \) removes the item \( it \) from \( Q \).
  **Precondition:** \( it \) is an item in \( Q \).

- \( \text{void} \quad Q.\text{change\_inf}(pq\text{\_item } it, \text{const } I& i) \) makes \( i \) the new information of item \( it \).
  **Precondition:** \( it \) is an item in \( Q \).

- \( \text{void} \quad Q.\text{decrease\_p}(pq\text{\_item } it, \text{const } P& x) \) makes \( x \) the new priority of item \( it \).
  **Precondition:** \( it \) is an item in \( Q \) and \( x \) is not larger then \( \text{prio}(it) \).

- \( \text{int} \quad Q.\text{size}() \) returns the size of \( Q \).

- \( \text{bool} \quad Q.\text{empty}() \) returns true, if \( Q \) is empty, false otherwise.

- \( \text{void} \quad Q.\text{clear}() \) makes \( Q \) the empty priority queue.

5. Implementation

Priority queues are implemented by binary heaps [91]. Operations \( \text{insert}, \text{del\_item}, \text{del\_min} \) take time \( O(\log n) \), \( \text{find\_min}, \text{decrease\_p}, \text{prio}, \text{inf}, \text{empty} \) take time \( O(1) \) and \( \text{clear} \) takes time \( O(n) \), where \( n \) is the size of \( Q \). The space requirement is \( O(n) \).
6. Example

Dijkstra’s Algorithm (cf. section 13)
9.2 Bounded Priority Queues (b\_priority\_queue)

1. Definition

An instance \( Q \) of the parameterized data type \( b\_priority\_queue\langle I \rangle \) is a collection of items (type \( b\_pq\_item \)). Every item contains a priority from a fixed interval \([a..b]\) of integers (type \( int \)) and an information from an arbitrary type \( I \). The number of items in \( Q \) is called the size of \( Q \). If \( Q \) has size zero it is called the empty priority queue. We use \( \langle p, i \rangle \) to denote a \( b\_pq\_item \) with priority \( p \in [a..b] \) and information \( i \).

Remark: Iteration over the elements of \( Q \) using iteration macros such as \( \text{forall} \) is not supported.

\#include &lt;LEDA/core/b\_prio.h &gt;

2. Creation

\scriptsize
\hl
b\_priority\_queue\langle I \rangle &amp; Q(int a, int b);
\hl

creates an instance \( Q \) of type \( b\_priority\_queue\langle I \rangle \) with key type \([a..b]\) and initializes it with the empty priority queue.

3. Operations

\scriptsize
\hl
b\_pq\_item &amp; Q.insert(int key, const I&amp; inf)
\hl

adds a new item \( \langle key, inf \rangle \) to \( Q \) and returns it.
\scriptsize
\hl
Precondition: key \( \in [a..b] \)
\hl

\hl
void Q.decrease\_key(b\_pq\_item it, int newkey)
\hl

makes \( newkey \) the new priority of item \( it \).
\scriptsize
\hl
Precondition: it is an item in \( Q \), newkey \( \in [a..b] \) and newkey is not larger than \( \text{prio(it)} \).
\hl

\hl
void Q.delete\_item(b\_pq\_item x)
\hl

 deletes item \( it \) from \( Q \).
\scriptsize
\hl
Precondition: it is an item in \( Q \).
\hl

\hl
int Q.prio(b\_pq\_item x)
\hl

returns the priority of item \( i \).
\scriptsize
\hl
Precondition: it is an item in \( Q \).
\hl

\hl
const I&amp; Q.inf(b\_pq\_item x)
\hl

returns the information of item \( i \).
\scriptsize
\hl
Precondition: it is an item in \( Q \).
\hl

\hl
b\_pq\_item Q.find\_min()
\hl

returns an item with minimal priority (\( nil \) if \( Q \) is empty).
\hl

I Q.delete\_min()
\hl

deletes the item \( it = Q.find\_min() \) from \( Q \) and returns the information of \( it \).
\scriptsize
\hl
Precondition: \( Q \) is not empty.
\hl

void Q.clear()

makes \( Q \) the empty bounded priority queue.
9.2. **BOUNDED PRIORITY QUEUES (B\_PRIORITY\_QUEUE)**

```plaintext
int Q.size()   returns the size of Q.
bool Q.empty() returns true if Q is empty, false otherwise.
int Q.lower_bound() returns the lower bound of the priority interval [a..b].
int Q.upper_bound() returns the upper bound of the priority interval [a..b].
```

4. **Implementation**

Bounded priority queues are implemented by arrays of linear lists. Operations insert, find_min, del_item, decrease_key, key, inf, and empty take time $O(1)$, del_min (= del_item for the minimal element) takes time $O(d)$, where $d$ is the distance of the minimal element to the next bigger element in the queue (= $O(b - a)$ in the worst case). clear takes time $O(b - a + n)$ and the space requirement is $O(b - a + n)$, where $n$ is the current size of the queue.
Chapter 10

Lossless Compression

Before we go into the details of the classes belonging to this module we want to give an overview of the different components and how they interact. We start with an example. Suppose you want to write a string to a compressed file named “foo” and read it back from the file. Then you can use the following program:

```cpp
#include <LEDA/basics/string.h>
#include <LEDA/coding/compress.h> // contains all compression classes

using namespace leda;

typedef HuffmanCoder Coder;

int main()
{
    string str = "Hello World";

    encoding_ofstream<Coder> out("foo");
    out << str << "\n";
    out.close();
    if (out.fail()) std::cout << "error writing foo" << "\n";

    decoding_ifstream<Coder> in("foo");
    str.read_line(in);
    in.close();
    if (in.fail()) std::cout << "error reading foo" << "\n";

    std::cout << "decoded string: " << str << "\n";

    return 0;
}
```

In the example above we used the classes `encoding_ofstream` and `decoding_ifstream` with LEDA datatypes only. We want to emphasize that they work together with user-defined types as well. All operations and operators (<< and >>) defined for C++ streams can be applied to them, too.
Assume that you want to send the file “foo” to a friend over the internet and you want to make sure that its contents do not get corrupted. Then you can easily add a checksum to your file. All you have to do is to replace the coder in the `typedef`-statement by `CoderPipe2<MD5SumCoder, HuffmanCoder>`. The class `CoderPipe2` combines the two LEDA coders `MD5SumCoder` (the checksummer) and `HuffmanCoder` into a single coder. If the pipe is used for encoding, then the `MD5SumCoder` is used first and the `HuffmanCoder` is applied to its output. In decoding mode the situation is reversed.

The standard behaviour of a checksummer like `MD5SumCoder` is as follows: In encoding mode it reads the input stream and computes a checksum; the output data basically consists of the input data with the checksum appended. In decoding mode the checksum is stripped from the input data and verified. If the input is corrupted the failure flag of the coder is set to signal this.

Suppose further that your friend has received the encoded file “foo” and wants to decode it but he does not know which combination of coders you have used for encoding. This is not a problem because LEDA provides a class called `AutoDecoder` which can be used to decode any stream that has been encoded by LEDA. The complete code for this extended example is depicted below:

```c++
#include <LEDA/basics/string.h>
#include <LEDA/coding/compress.h>

using namespace leda;

typedef CoderPipe2<MD5SumCoder, HuffmanCoder> Coder;

int main()
{
    string str = "Hello World";

    // your code ...
    encoding_ofstream<Coder> out("foo");
    out << str << "\n";
    out.close();
    if (out.fail()) std::cout << "error writing foo" << "\n";

    // your friend's code ...
    autodecoding_ifstream in("foo");
    str.read_line(in);
    // autodecoding_ifstream = decoding_istream<AutoDecoder>
    in.finish(); // read till the end before closing (~ verify checksum)
    if (in.fail()) std::cout << "decoding error, foo corrupted" << "\n";
    std::cout << "decoded string: " << str << "\n";

    return 0;
}
```

This example shows how easy it is to add compression to existing applications: You include the header “`LEDA/coding/compress.h`”, which makes all classes in the com-
pression module available. Then you simply replace every occurrence of ofstream by encoding_ofstream<Coder> and every occurrence of ifstream by autodecoding_ifstream.

Of course, you can also use the LEDA coders in file mode. This means you can encode a file "foo" into a file "bar" and decode "bar" again. The example below shows how. We also demonstrate a nice feature of the AutoDecoder: If you query a description after the decoding the object tells you which combination has been used for encoding the input.

```cpp
#include <LEDA/coding/compress.h>

using namespace leda;

typedef CoderPipe2<MD5SumCoder, HuffmanCoder> Coder;

int main()
{
    Coder coder("foo", "bar");
    coder.encode();
    if (coder.fail()) std::cout << "error encoding foo" << "\n";

    AutoDecoder auto("bar", "foo");
    auto.decode();
    if (auto.fail()) std::cout << "error decoding bar" << "\n";

    std::cout << "Decoding info: " << auto.get_description() << "\n";

    return 0;
}
```

More examples can be found in $LEDAROOT/test/compression. There we show in particular how the user can build a LEDA compliant coder which integrates seamlessly with the AutoDecoder.

Below we give a few suggestions about when to use which coder:

- maximum compression needed, speed and memory usage not so important:
  use the PPMIIICoder.

- high compression needed, main memory < 2MB, speed less relevant:
  use a coder from the BWT-family:
  ```cpp
  CoderPipe4<BWTCoder, MTF2Coder, RLE0Coder, A0Coder>
  CoderPipe3<BWTCoder, MTFCoder, A0Coder>.
  ```

- maximum speed and good compression needed:
  use the DeflateCoder.
10.1 Adaptive Arithmetic Coder (A0Coder)

1. Definition

An instance $C$ of $A0Coder$ is an adaptive arithmetic coder of order zero (which explains the “0” in the class name). When the coder encodes a stream it counts the frequencies of the characters that have occurred so far in order to obtain a model of the probabilities for the future characters. The model is called order zero because it does not take into account that the probability of a character $c$ may depend on the characters that precede it. (E.g., in an English text the probability that the next character is a “u” is very high if the current character is a “q”. The $PPMIICoder$ in Section 10.3 takes this into account.) The advantage of arithmetic coding [92] is that a character in the source stream can be encoded by a fractional number of bits in the target stream, which leads to good results in practice.

The coder is called adaptive because the model evolves gradually while the coder scans its input. In contrast to that a static coder reads the whole input once before it generates a model. During the actual encoding (in a second scan) the model remains fixed. The adaptive coder resets and/or scales its model when the number of characters read since the last reset/scaling exceeds a certain threshold. Both thresholds can be controlled by the user.

```c
#include <LEDA/coding/arithmetic_coding.h>
```

2. Creation

$A0Coder$ $C(\text{streambuf } * \text{src\_stream} = 0, \text{streambuf } * \text{tgt\_stream} = 0,$
\begin{verbatim}
 bool own\_streams = false);
\end{verbatim}
creates an instance $C$ which uses the given source and target streams. If $own\_streams$ is set, then $C$ is responsible for the destruction of the streams, otherwise the pointers $src\_stream$ and $tgt\_stream$ must be valid during the life-time of $C$.

$A0Coder$ $C(\text{const char } * \text{src\_file\_name}, \text{const char } * \text{tgt\_file\_name});$
creates an instance $C$ which uses file-streams for input and output.

3. Operations

Standard Operations

```c
void $C$.encode( ) encodes the source stream and writes the output to the target stream.
```

```c
void $C$.decode( ) decodes the source stream and writes the output to the target stream.
```
10.1. ADAPTIVE ARITHMETIC CODER (A0CODER)

```c
uint32 C.encode_memory_chunk(const char *in_buf, uint32 in_len, char *out_buf, uint32 out_len)
```

encodes the memory chunk starting at `in_buf` with size `in_len` into the buffer starting at `out_buf` with size `out_len`. The function returns actual length of the encoded chunk which may be smaller than `out_len`. If the output buffer is too small for the encoded data the failure flag will be set (see below).

```c
uint32 C.decode_memory_chunk(const char *in_buf, uint32 in_len, char *out_buf, uint32 out_len)
```

decodes a memory chunk. The meaning of the parameters and the return value is the same as in the previous function.

```c
streambuf* C.get_src_stream()
```
returns the current source stream.

```c
void C.set_src_stream(streambuf *src_stream, bool own_stream = false)
```
sets the source stream (cf. constructor).

```c
void C.set_src_file(const char *file_name)
```
sets a file as source stream.

```c
streambuf* C.get_tgt_stream()
```
returns the current target stream.

```c
void C.set_tgt_stream(streambuf *tgt_stream, bool own_Stream = false)
```
sets the target stream (cf. constructor).

```c
void C.set_tgt_file(const char *file_name)
```
sets a file as target stream.

```c
void C.reset(bool keep_thresholds = true)
```
puts `C` in the same state as the default constructor. If `keep_thresholds` is false the thresholds are set to their default values.

```c
bool C.failed()
```
returns `true` if an error occurred.

```c
bool C.finished()
```
returns `true` if the coding is finished.

```c
string C.get_description()
```
provides a description for `C`.

### Additional Operations

```c
uint32 C.get_reset_threshold()
```
returns the current threshold for resetting the model.

```c
void C.set_reset_threshold(uint32 t)
```
sets the threshold for resetting the model.
```plaintext
void C.set_scale_threshold(uint32 t)
  sets the threshold for scaling the model.
  Precondition: \(512 \leq t \leq S\)
```
10.2 Static Arithmetic Coder (A0sCoder)

1. Definition

An instance $C$ of $A0sCoder$ is a static arithmetic coder of order zero. (We assume that the reader is familiar with the definition of the class $A0Coder$ from the previous section.) $A0sCoder$ is called static because it uses a fixed model for encoding. The input is scanned twice. In the first scan the frequencies of the characters are counted. This information is used to precompute the model. In the second scan the actual coding takes place. Since the model is fixed the static coder is usually faster than the adaptive coder (despite the two scans) but the compression rates are often slightly inferior.

#include <LEDA/coding/arithmetic_coding.h>

2. Creation

$A0sCoder \ C(streambuf*src\_stream=0,streambuf\_get\_tgt\_stream=0,$
bool own\_streams=false);

creates an instance $C$ which uses the given source and target streams. If own\_streams is set, then $C$ is responsible for the destruction of the streams, otherwise the pointers src\_stream and tgt\_stream must be valid during the life-time of $C$.

$A0sCoder \ C(const\ char*src\_file\_name,const\ char\_tgt\_file\_name);$  
creates an instance $C$ which uses file-streams for input and output.

3. Operations

Standard Operations

void $C$.encode() encodes the source stream and writes the output to the target stream.

void $C$.decode() decodes the source stream and writes the output to the target stream.

uint32 $C$.encode\_memory\_chunk(const char*in\_buf,uint32 in\_len,char*out\_buf,
uint32 out\_len)  
encodes the memory chunk starting at in\_buf with size in\_len into the buffer starting at out\_buf with size out\_len. The function returns actual length of the encoded chunk which may be smaller than out\_len. If the output buffer is too small for the encoded data the failure flag will be set (see below).
\begin{verbatim}
uint32 C.decode_memory_chunk(const char * in_buf, uint32 in_len, char * out_buf, uint32 out_len)

decodes a memory chunk. The meaning of the parameters and the return value is the same as in the previous function.

streambuf* C.get_src_stream() returns the current source stream.

void C.set_src_stream(streambuf* src_stream, bool own_stream = false)

sets the source stream (cf. constructor).

void C.set_src_file(const char* file_name)

sets a file as source stream.

streambuf* C.get_tgt_stream() returns the current target stream.

void C.set_tgt_stream(streambuf* tgt_stream, bool own_stream = false)

sets the target stream (cf. constructor).

void C.set_tgt_file(const char* file_name)

sets a file as target stream.

void C.reset() puts C in exactly the same state as the default constructor.

bool C.failed() returns true if an error occurred.

bool C.finished() returns true if the coding is finished.

string C.get_description() provides a description for C.
\end{verbatim}
10.3 Prediction by Partial Matching (PPMIIICoder)

1. Definition

The *PPMIIICoder* is based on the compression scheme “Prediction by Partial Matching with Information Inheritance” by D. Shkarin [81].

This coder works as follows: Suppose we have processed the first \( n - 1 \) symbols \( x_1 \ldots x_{n-1} \) of the stream. Before reading the next symbol \( x_n \) we try to guess it, i.e. for every symbol \( s \) we estimate the probability \( p(s) \) for the event “\( x_n = s \)”. This probability distribution determines how the next symbol is encoded: The higher \( p(s) \), the fewer bits are used for encoding \( s \). If our estimation is good, which means that \( p(x_n) \) is high, then we obtain a good compression rate.

In order to predict the probability distribution for the \( n \)th symbol the PPM approach considers the preceding \( k \) symbols \( x_{n-k} \ldots x_{n-1} \). We call these symbols the context of \( x_n \) and \( k \) the order of the model. (For \( k = 0 \) we obtain the order-0 model from the previous section.) E.g., if the current context is “req”, then we should predict the letter “u” as next symbol with high probability.

PPMII is a variant of PPM which usually achieves very accurate estimations.

The *PPMIIICoder* combines very good compression rates with acceptable speed. (Shkarin [81] reports that his coder outperforms ZIP and BZIP2 with respect to compression rates and speed.) The only disadvantage of this coder is that it needs a fair amount of main memory to store the model. However, the user can set an upper bound on the memory usage. And he can specify which model restoration method the coder shall apply when it runs out of memory:

- **mr_restart** (default):
  The model is deleted completely and rebuilt from scratch. This method is fast.

- **mr_cut_off**:
  Parts of the model are freed to gain memory. This method is optimal for so-called quasistationary sources. It usually gives better compression but it is slower.

- **mr_freeze**:
  The model is not extended any more. This method is optimal for so-called stationary sources. (We want to point out that data streams arising in practical applications usually do not behave like a stationary source.)

```cpp
#include <LEDA/coding/PPMII.h>
```

2. Types

*PPMIIICoder::methode { mr_restart, mr_cut_off, mr_freeze }*

the different model restoration modes.
3. Creation

PPMIICoder \( C(\text{streambuf} * \text{src\_stream} = 0, \text{streambuf} * \text{tgt\_stream} = 0, \)
\( \text{bool own\_streams} = \text{false}); \)

creates an instance \( C \) which uses the given source and target
streams. If \( \text{own\_streams} \) is set, then \( C \) is responsible for the destruc-
tion of the streams, otherwise the pointers \( \text{src\_stream} \) and \( \text{tgt\_stream} \)
must be valid during the life-time of \( C \).

PPMIICoder \( C(\text{const char} * \text{src\_file\_name}, \text{const char} * \text{tgt\_file\_name}); \)

creates an instance \( C \) which uses file-streams for input and output.

4. Operations

Standard Operations

\( \text{void} \) \( C.\text{encode}() \) encodes the source stream and writes the output to
the target stream.

\( \text{void} \) \( C.\text{decode}() \) decodes the source stream and writes the output to
the target stream.

\( \text{uint32} \) \( C.\text{encode\_memory\_chunk}(\text{const char} * \text{in\_buf}, \text{uint32 in\_len}, \text{char} * \text{out\_buf}, \)
\( \text{uint32 out\_len} \) encodes the memory chunk starting at \( \text{in\_buf} \) with
size \( \text{in\_len} \) into the buffer starting at \( \text{out\_buf} \) with size
\( \text{out\_len} \). The function returns actual length of the
encoded chunk which may be smaller than \( \text{out\_len} \). If
the output buffer is too small for the encoded data
the failure flag will be set (see below).

\( \text{uint32} \) \( C.\text{decode\_memory\_chunk}(\text{const char} * \text{in\_buf}, \text{uint32 in\_len}, \text{char} * \text{out\_buf}, \)
\( \text{uint32 out\_len} \) decodes a memory chunk. The meaning of the pa-
rameters and the return value is the same as in the
previous function.

\( \text{streambuf} * \) \( C.\text{get\_src\_stream}() \) returns the current source stream.

\( \text{void} \) \( C.\text{set\_src\_stream}(\text{streambuf} * \text{src\_stream}, \text{bool own\_stream} = \text{false}) \)
sets the source stream (cf. constructor).

\( \text{void} \) \( C.\text{set\_src\_file}(\text{const char} * \text{file\_name}) \)
sets a file as source stream.

\( \text{streambuf} * \) \( C.\text{get\_tgt\_stream}() \) returns the current target stream.

\( \text{void} \) \( C.\text{set\_tgt\_stream}(\text{streambuf} * \text{tgt\_stream}, \text{bool own\_Stream} = \text{false}) \)
sets the target stream (cf. constructor).
void C.set_tgt_file(const char *file_name)
    sets a file as target stream.

void C.reset(bool keep_parameters = true)
    puts C in the same state as the default constructor.
    If keep_parameters is false the parameters are set to
    their default values.

bool C.failed() returns true if an error occurred.

bool C.finished() returns true if the coding is finished.

string C.get_description() provides a description for C.

Additional Operations

uint16 C.get_model_memory_bound()
    returns how much memory (in MB) C is allowed to
    use for storing the model.

void C.set_model_memory_bound(uint16 max_mem)
    determines the amount of memory available for the
    model (in MB).

byte C.get_model_order() returns the order of the model.

void C.set_model_order(byte order)
    sets the order of the model.
    Precondition: order ∈ [2..16].

mr_method C.get_model_restoration_method()
    returns the model restoration method.

void C.set_model_restoration_method(mr_method method)
    sets the model restoration method.

5. Implementation

Our implementation encapsulates the code by D. Shkarin (the implementation of the
PPMII model) and by D. Subbotin (the implementation of the range coder).
10.4 Deflation/Inflation Coder (DeflateCoder)

1. Definition

The class DeflateCoder encapsulates the algorithms “deflation” and “inflation” from the zlib-library by J.-L. Gilly and M. Adler (see www.zlib.org).

Both algorithms use two buffers (for encoded and decoded data). The user can control the size of each buffer. In addition he can fine-tune some parameters of the algorithm, which gives him a trade-off between compression rate and speed.

This coder is fast, gives good compression rates and has moderate memory requirements.

#include <LEDA/coding/deflate.h>

2. Creation

DeflateCoder C(streambuf * src_stream = 0, streambuf * tgt_stream = 0,
bool ownStreams = false, uint32 enc_buffer_sz = DefaultBufferSize,
uint32 dec_buffer_sz = DefaultBufferSize);

creates an instance C which uses the given source and target streams. If ownStreams is set, then C is responsible for the destruction of the streams, otherwise the pointers src_stream and tgt_stream must be valid during the life-time of C. (In addition the sizes of the two buffers may be specified.)

DeflateCoder C(const char * src_file_name, const char * tgt_file_name,
uint32 enc_buffer_sz = DefaultBufferSize,
uint32 dec_buffer_sz = DefaultBufferSize);

creates an instance C which uses file-streams for input and output. (In addition the sizes of the two buffers may be specified.)

3. Operations

Standard Operations

void C.encode() encodes the source stream and writes the output to the target stream.

void C.decode() decodes the source stream and writes the output to the target stream.
10.4. DEFLATION/INFLATION CODER (DEFLATECODER)

\texttt{uint32 C.encode\_memory\_chunk(const char * in\_buf, uint32 in\_len, char * out\_buf, uint32 out\_len)}

generates the memory chunk starting at \texttt{in\_buf} with size \texttt{in\_len} into the buffer starting at \texttt{out\_buf} with size \texttt{out\_len}. The function returns actual length of the encoded chunk which may be smaller than \texttt{out\_len}. If the output buffer is too small for the encoded data the failure flag will be set (see below).

\texttt{uint32 C.decode\_memory\_chunk(const char * in\_buf, uint32 in\_len, char * out\_buf, uint32 out\_len)}

decodes a memory chunk. The meaning of the parameters and the return value is the same as in the previous function.

\texttt{streambuf* C.get\_src\_stream()} returns the current source stream.

\texttt{void C.set\_src\_stream(streambuf* src\_stream, bool own\_stream = false)}

sets the source stream (cf. constructor).

\texttt{void C.set\_src\_file(const char* file\_name)}

sets a file as source stream.

\texttt{streambuf* C.get\_tgt\_stream()} returns the current target stream.

\texttt{void C.set\_tgt\_stream(streambuf* tgt\_stream, bool own\_stream = false)}

sets the target stream (cf. constructor).

\texttt{void C.set\_tgt\_file(const char* file\_name)}

sets a file as target stream.

\texttt{void C.reset(bool keep\_parameters = true)}

puts \texttt{C} in the same state as the default constructor. If \texttt{keep\_parameters} is false the parameters are set to their default values.

\texttt{bool C.failed()} returns \texttt{true} if an error occurred.

\texttt{bool C.finished()} returns \texttt{true} if the coding is finished.

\texttt{string C.get\_description()} provides a description for \texttt{C}.

**Additional Operations**

\texttt{int C.get\_compression\_level()} returns the current compression level.
void C.set_compression_level(int level)
sets the compression level (between 0 and 9), whereas
0 means no compression, 1 gives the best speed and 9
yields the highest compression.

int C.get_window_size() returns the base two logarithm of the size of the sliding
window used by the deflation algorithm.

void C.set_window_size(int log2_of_sz)
sets the window size (log2_of_sz must be between 8 and 15).

int C.get_memory_level() returns the current memory level.

void C.set_memory_level(int level)
sets the memory level (between 1 and 9). The higher
this number, the higher the memory consumption.

int C.get_strategy() returns the current strategy for the deflation algo-

void C.set_strategy(int strategy)
sets the compression strategy (between 0 and 3):
0=default, 1=optimization for filtered data, i.e. small
values with random distribution, 2=Huffman only (no
string matching), 3=RLE (for image data).

void C.get_memory_consumption(uint32 & mem_for_enc, uint32 & mem_for_dec)
returns the (approximate) memory consumption (in
bytes) needed for encoding and decoding repectively.

uint32 C.get_enc_buffer_size() returns the size of the buffer for encoded data.

void C.set_enc_buffer_size(uint32 buffer_size)
sets the size of the buffer for encoded data.

uint32 C.get_dec_buffer_size() returns the size of the buffer for decoded data.

void C.set_dec_buffer_size(uint32 buffer_size)
sets the size of the buffer for decoded data.
10.5 Dictionary Based Coder (DictCoder)

1. Definition

The class DictCoder provides a dictionary based coder. The algorithms for encoding and decoding follow the ideas of Lempel and Ziv [93, 94].

```
#include <LEDA/coding/dict_coder.h>
```

2. Creation

```
DictCoder C(streambuf * src_stream = 0, streambuf * tgt_stream = 0,
             bool own_streams = false);
```

creates an instance \( C \) which uses the given source and target streams. If \( own\_streams \) is set, then \( C \) is responsible for the destruction of the streams, otherwise the pointers \( src\_stream \) and \( tgt\_stream \) must be valid during the life-time of \( C \).

```
DictCoder C(const char * src_file_name, const char * tgt_file_name);
```

creates an instance \( C \) which uses file-streams for input and output.

3. Operations

Standard Operations

```
void C.encode() encodes the source stream and writes the output to the target stream.
```

```
void C.decode() decodes the source stream and writes the output to the target stream.
```

```
uint32 C.encode_memory_chunk(const char * in_buf, uint32 in_len, char * out_buf,
                              uint32 out_len)
```

encodes the memory chunk starting at \( in\_buf \) with size \( in\_len \) into the buffer starting at \( out\_buf \) with size \( out\_len \). The function returns actual length of the encoded chunk which may be smaller than \( out\_len \). If the output buffer is too small for the encoded data the failure flag will be set (see below).

```
uint32 C.decode_memory_chunk(const char * in_buf, uint32 in_len, char * out_buf,
                              uint32 out_len)
```

decodes a memory chunk. The meaning of the parameters and the return value is the same as in the previous function.

```
streambuf* C.get_src_stream() returns the current source stream.
```
void C.set_src_stream(streambuf * src_stream, bool own_stream = false) sets the source stream (cf. constructor).

void C.set_src_file(const char * file_name) sets a file as source stream.

streambuf* C.get_tgt_stream() returns the current target stream.

void C.set_tgt_stream(streambuf * tgt_stream, bool own_Stream = false) sets the target stream (cf. constructor).

void C.set_tgt_file(const char * file_name) sets a file as target stream.

void C.reset() puts C in exactly the same state as the default constructor.

bool C.failed() returns true if an error occurred.

bool C.finished() returns true if the coding is finished.

string C.get_description() provides a description for C.
10.6 Static Huffman Coder (HuffmanCoder)

1. Definition

The class HuffmanCoder provides a static coder based on the algorithm by Huffman [48]. The term static means that a fixed model is used for encoding. The input is scanned twice. In the first scan the frequencies of the characters are counted. This information is used to precompute the Huffman-trees. The actual coding takes place in a second scan.

```
#include <LEDA/coding/huffman.h>
```

2. Creation

```
HuffmanCoder C(streambuf * src_stream = 0, streambuf * tgt_stream = 0,
bool own_streams = false);
```

creates an instance C which uses the given source and target streams. If own_streams is set, then C is responsible for the destruction of the streams, otherwise the pointers src_stream and tgt_stream must be valid during the life-time of C.

```
HuffmanCoder C(const char * src_file_name, const char * tgt_file_name);
```

creates an instance C which uses file-streams for input and output.

3. Operations

**Standard Operations**

- `void C.encode()` encodes the source stream and writes the output to the target stream.
- `void C.decode()` decodes the source stream and writes the output to the target stream.
- `uint32 C.encode_memory_chunk(const char * in_buf, uint32 in_len, char * out_buf,
uint32 out_len)` encodes the memory chunk starting at in_buf with size in_len into the buffer starting at out_buf with size out_len. The function returns actual length of the encoded chunk which may be smaller than out_len. If the output buffer is too small for the encoded data the failure flag will be set (see below).
- `uint32 C.decode_memory_chunk(const char * in_buf, uint32 in_len, char * out_buf,
uint32 out_len)` decodes a memory chunk. The meaning of the parameters and the return value is the same as in the previous function.
streambuf* C.get_src_stream() returns the current source stream.

void C.set_src_stream(streambuf* src_stream, bool own_stream = false)
sets the source stream (cf. constructor).

void C.set_src_file(const char* file_name)
sets a file as source stream.

streambuf* C.get_tgt_stream() returns the current target stream.

void C.set_tgt_stream(streambuf* tgt_stream, bool own_stream = false)
sets the target stream (cf. constructor).

void C.set_tgt_file(const char* file_name)
sets a file as target stream.

void C.reset() puts C in exactly the same state as the default constructor.

bool C.failed() returns true if an error occurred.

bool C.finished() returns true if the coding is finished.

string C.get_description() provides a description for C.
10.7 Adaptive Huffman Coder (AdaptiveHuffmanCoder)

1. Definition

The class AdaptiveHuffmanCoder provides an adaptive coder based on the algorithm by J.S. Vitter [87]. Vitter has proven that his algorithm achieves the best compression among all one-pass Huffman schemes.

```c
#include <LEDA/coding/huffman.h>
```

2. Creation

AdaptiveHuffmanCoder C(streambuf * src_stream = 0, streambuf * tgt_stream = 0, bool own_streams = false);

creates an instance C which uses the given source and target streams. If own_streams is set, then C is responsible for the destruction of the streams, otherwise the pointers src_stream and tgt_stream must be valid during the life-time of C.

AdaptiveHuffmanCoder C(const char * src_file_name, const char * tgt_file_name);

creates an instance C which uses file-streams for input and output.

3. Operations

Standard Operations

```c
void C.encode() encodes the source stream and writes the output to the target stream.

void C.decode() decodes the source stream and writes the output to the target stream.

uint32 C.encode_memory_chunk(const char * in_buf, uint32 in_len, char * out_buf, uint32 out_len)
encodes the memory chunk starting at in_buf with size in_len into the buffer starting at out_buf with size out_len. The function returns actual length of the encoded chunk which may be smaller than out_len. If the output buffer is too small for the encoded data the failure flag will be set (see below).

uint32 C.decode_memory_chunk(const char * in_buf, uint32 in_len, char * out_buf, uint32 out_len)
decodes a memory chunk. The meaning of the parameters and the return value is the same as in the previous function.
```
**CHAPTER 10. LOSSLESS COMPRESSION**

```c
streambuf* C.get_src_stream() returns the current source stream.

void C.set_src_stream(streambuf* src_stream, bool own_stream = false)
    sets the source stream (cf. constructor).

void C.set_src_file(const char* file_name)
    sets a file as source stream.

streambuf* C.get_tgt_stream() returns the current target stream.

void C.set_tgt_stream(streambuf* tgt_stream, bool own_stream = false)
    sets the target stream (cf. constructor).

void C.set_tgt_file(const char* file_name)
    sets a file as target stream.

void C.reset() puts C in exactly the same state as the default constructor.

bool C.failed() returns true if an error occurred.

bool C.finished() returns true if the coding is finished.

string C.get_description() provides a description for C.
```
10.8 Run-Length Coder (RLECoder)

1. Definition
A Run-Length coder is used for compressing inputs which contain many long sequences of repeating characters. The basic idea is as follows: A sequence aaaaaaa of 7 consecutive a’s is replaced by something like a#7.
We provide two schemes for the actual encoding of the compressed sequence, which allows the user to select between fast or high compression.

#include <LEDA/coding/RLE.h>

2. Creation
RLECoder C(streambuf * src_stream = 0, streambuf * tgt_stream = 0, bool own_streams = false, bool fast_mode = true);
creates an instance C which uses the given source and target streams. If ownStreams is set, then C is responsible for the destruction of the streams, otherwise the pointers src_stream and tgt_stream must be valid during the life-time of C. The parameter fast_mode allows to select fast or high compression mode.

RLECoder C(const char * src_file_name, const char * tgt_file_name, bool fast_mode = true);
creates an instance C which uses file-streams for input and output. The parameter fast_mode allows to select fast or high compression mode.

3. Operations
Standard Operations

void C.encode() encodes the source stream and writes the output to the target stream.

void C.decode() decodes the source stream and writes the output to the target stream.

uint32 C.encode_memory_chunk(const char * in_buf, uint32 in_len, char * out_buf, uint32 out_len)
encodes the memory chunk starting at in_buf with size in_len into the buffer starting at out_buf with size out_len. The function returns actual length of the encoded chunk which may be smaller than out_len. If the output buffer is too small for the encoded data the failure flag will be set (see below).


\begin{verbatim}
uint32 C.decode_memory_chunk(const char * in_buf, uint32 in_len, char * out_buf, uint32 out_len)
decodes a memory chunk. The meaning of the parameters and the return value is the same as in the previous function.

streambuf* C.get_src_stream( ) returns the current source stream.

void C.set_src_stream(streambuf* src_stream, bool own_stream = false)
sets the source stream (cf. constructor).

void C.set_src_file(const char* file_name)
sets a file as source stream.

streambuf* C.get_tgt_stream( ) returns the current target stream.

void C.set_tgt_stream(streambuf* tgt_stream, bool own_Stream = false)
sets the target stream (cf. constructor).

void C.set_tgt_file(const char* file_name)
sets a file as target stream.

void C.reset(bool keep_mode = true)
puts C in the same state as the default constructor. If keep_mode is false the compression mode is set to the default value.

bool C.failed( ) returns true if an error occurred.

bool C.finished( ) returns true if the coding is finished.

string C.get_description( ) provides a description for C.

Additional Operations

bool C.is_in_fast_mode( ) returns true if the coder is in fast mode, and false if it is in high compression mode.

void C.set_fast_compression_mode( )
selects fast compression mode.

void C.set_high_compression_mode( )
selects high compression mode.
\end{verbatim}
10.9 Burrows-Wheeler Transform (BWTCoder)

1. Definition

The class `BWTCoder` applies the Burrows-Wheeler transform [18] (in encoding mode) or its reversal (in decoding mode) to the input stream. To be more precise, in encoding mode the input stream is partitioned in blocks and the transformation is applied to each block. The size of the blocks, i.e. the number of characters per block, can be specified by the user.

This transformation does not compress the input but it can be used as a preprocessing step for other coders. Applying `BWTCoder` first often improves the overall compression rates. The following combinations yield very good compression results:

- `CoderPipe3<BWTCoder, MTCoder, A0Coder>`
- `CoderPipe4<BWTCoder, MTCoder, RLE0Coder, A0Coder>`

```cpp
#include <LEDA/coding/BWT.h>
```

2. Creation

`BWTCoder` `C(streambuf * src_stream = 0, streambuf * tgt_stream = 0, bool own_streams = false, uint32 block_size = DefaultBlockSize);`

creates an instance `C` which uses the given source and target streams. If `own_streams` is set, then `C` is responsible for the destruction of the streams, otherwise the pointers `src_stream` and `tgt_stream` must be valid during the life-time of `C`. The block size can be controlled via `block_size`.

`BWTCoder` `C(const char * src_file_name, const char * tgt_file_name, uint32 block_size = DefaultBlockSize);`

creates an instance `C` which uses file-streams for input and output. The block size can be controlled via `block_size`.

3. Operations

**Standard Operations**

```cpp
void C.encode() encodes the source stream and writes the output to the target stream.

void C.decode() decodes the source stream and writes the output to the target stream.
```
CHAPTER 10. LOSSLESS COMPRESSION

```
uint32 C.encode_memory_chunk(const char *in_buf, uint32 in_len, char *out_buf, uint32 out_len)

encodes the memory chunk starting at in_buf with size in_len into the buffer starting at out_buf with size out_len. The function returns actual length of the encoded chunk which may be smaller than out_len. If the output buffer is too small for the encoded data the failure flag will be set (see below).

uint32 C.decode_memory_chunk(const char *in_buf, uint32 in_len, char *out_buf, uint32 out_len)
decodes a memory chunk. The meaning of the parameters and the return value is the same as in the previous function.
```

```
streambuf* C.get_src_stream()
returns the current source stream.

void C.set_src_stream(streambuf* src_stream, bool own_stream = false)
sets the source stream (cf. constructor).

void C.set_src_file(const char* file_name)
sets a file as source stream.

streambuf* C.get_tgt_stream()
returns the current target stream.

void C.set_tgt_stream(streambuf* tgt_stream, bool own_Stream = false)
sets the target stream (cf. constructor).

void C.set_tgt_file(const char* file_name)
sets a file as target stream.

void C.reset(bool keep_block_size = true)
puts C in the same state as the default constructor. If keep_block_size is false the block size is set to the default value.

bool C.failed()
returns true if an error occurred.

bool C.finished()
returns true if the coding is finished.

string C.get_description()
provides a description for C.
```

**Additional Operations**

```
uint32 C.get_block_size()
returns the current block size (for encoding).

void C.set_block_size(uint32 block_size)
sets the block size.
```
4. Implementation

Our implementation is based on code for suffix sorting by P. Ferragina and G. Manzini.
10.10 Move-To-Front Coder (MTFCoder)

1. Definition

A Move-To-Front coder is used for preprocessing the input before it is fed to the actual compressor. Encoding works as follows: The coder maintains a list \( L \) containing all the 256 characters that can appear in the input. Whenever it receives an input character \( c \) it looks up the position \( i \) of \( c \) in \( L \), outputs \( i \) and moves \( c \) to the front of \( L \).

Let us look at an example. Assume that the position of every character in the initial list is equal to its ASCII-code. When we encode the sequence \( \text{aaabbbbaab} \) we obtain the output \( 97, 0, 0, 98, 0, 0, 1, 0, 1 \). So a MTFCoder has the following properties:

- If the input contains sequences where one character is repeated many times then the output will have a lot of consecutive zeros.
- A sequence in the input where a small set of different characters is used corresponds to a sequence in the output with (mainly) small integers.

Therefore this coder is often used as an intermediate coding step between a Burrows-Wheeler transform (see Section 10.9) and a compressing coder.

#include <LEDA/coding/MTF.h>

2. Creation

\[ \text{MTFCoder } C(\text{streambuf} * \text{src\_stream} = 0, \text{streambuf} * \text{tgt\_stream} = 0, \]
\[ \text{bool own\_streams} = \text{false}); \]

creates an instance \( C \) which uses the given source and target streams. If \( \text{own\_streams} \) is set, then \( C \) is responsible for the destruction of the streams, otherwise the pointers \( \text{src\_stream} \) and \( \text{tgt\_stream} \) must be valid during the life-time of \( C \).

\[ \text{MTFCoder } C(\text{const char} * \text{src\_file\_name}, \text{const char} * \text{tgt\_file\_name}); \]

creates an instance \( C \) which uses file-streams for input and output.

3. Operations

Standard Operations

\[ \text{void } C.\text{encode() } \]
encodes the source stream and writes the output to the target stream.

\[ \text{void } C.\text{decode() } \]
decodes the source stream and writes the output to the target stream.
\texttt{uint32 \hspace{1em} C.encode\_memory\_chunk(const char* in\_buf, uint32 in\_len, char* out\_buf, uint32 out\_len)}

encodes the memory chunk starting at \texttt{in\_buf} with size \texttt{in\_len} into the buffer starting at \texttt{out\_buf} with size \texttt{out\_len}. The function returns actual length of the encoded chunk which may be smaller than \texttt{out\_len}. If the output buffer is too small for the encoded data the failure flag will be set (see below).

\texttt{uint32 \hspace{1em} C.decode\_memory\_chunk(const char* in\_buf, uint32 in\_len, char* out\_buf, uint32 out\_len)}

decodes a memory chunk. The meaning of the parameters and the return value is the same as in the previous function.

\texttt{streambuf* \hspace{1em} C.get\_src\_stream() \hspace{1em} returns the current source stream.}

\texttt{void \hspace{1em} C.set\_src\_stream(streambuf* src\_stream, bool own\_stream = false)}

sets the source stream (cf. constructor).

\texttt{void \hspace{1em} C.set\_src\_file(const char* file\_name)}

sets a file as source stream.

\texttt{streambuf* \hspace{1em} C.get\_tgt\_stream() \hspace{1em} returns the current target stream.}

\texttt{void \hspace{1em} C.set\_tgt\_stream(streambuf* tgt\_stream, bool own\_Stream = false)}

sets the target stream (cf. constructor).

\texttt{void \hspace{1em} C.set\_tgt\_file(const char* file\_name)}

sets a file as target stream.

\texttt{void \hspace{1em} C.reset()} \hspace{1em} puts \texttt{C} in exactly the same state as the default constructor.

\texttt{bool \hspace{1em} C.failed()} \hspace{1em} returns \texttt{true} if an error occurred.

\texttt{bool \hspace{1em} C.finished()} \hspace{1em} returns \texttt{true} if the coding is finished.

\texttt{string \hspace{1em} C.get\_description()} \hspace{1em} provides a description for \texttt{C}.
10.11 Move-To-Front Coder II (MTF2Coder)

1. Definition

This is a variant of the Move-To-Front coder from the previous section. It was proposed by Balkenhol and Shtarkov [7] and sometimes yields better compression than MTFCoder (when used with BWTCoder as preprocessor and A0Coder as postprocessor). The idea is not to exchange the symbol at the head of MTF list $L$ whenever a new symbol is encountered.

The details are as follows: Let us call the position of a symbol $c$ in $L$ the rank of $c$ where the rank of the head symbol is 0. Whenever a symbol of rank greater than 1 appears we move it to position 1, i.e. behind the head. When a symbol of rank 1 is encountered it is moved to position 0 only if the previous symbol was not the head of $L$. Otherwise it remains at position 1.

#include <LEDA/coding/MTF2.h>

2. Creation

MTF2Coder $C$(streambuf * src_stream = 0, streambuf * tgt_stream = 0, bool own_streams = false);

creates an instance $C$ which uses the given source and target streams. If own_streams is set, then $C$ is responsible for the destruction of the streams, otherwise the pointers src_stream and tgt_stream must be valid during the life-time of $C$.

MTF2Coder $C$(const char * src_file_name, const char * tgt_file_name);

creates an instance $C$ which uses file-streams for input and output.

3. Operations

Standard Operations

void $C$.encode( ) encodes the source stream and writes the output to the target stream.

void $C$.decode( ) decodes the source stream and writes the output to the target stream.

uint32 $C$.encode_memory_chunk(const char * in_buf, uint32 in_len, char * out_buf, uint32 out_len)

encodes the memory chunk starting at in_buf with size in_len into the buffer starting at out_buf with size out_len. The function returns actual length of the encoded chunk which may be smaller than out_len. If the output buffer is too small for the encoded data the failure flag will be set (see below).
10.11. MOVE-TO-FRONT CODER II (MTF2CODER)

\[ \text{uint32 } \ C. \text{decode_memory_chunk(} \text{const char } \ast \text{in_buf, uint32 in_len, char } \ast \text{out_buf, uint32 out_len) } \]

decodes a memory chunk. The meaning of the parameters and the return value is the same as in the previous function.

\[ \text{streambuf } \ast \ C. \text{get_src_stream( ) returns the current source stream.} \]

\[ \text{void } \ C. \text{set_src_stream(} \text{streambuf } \ast \text{src_stream, bool own_stream } = \text{false) } \]

sets the source stream (cf. constructor).

\[ \text{void } \ C. \text{set_src_file(} \text{const char } \ast \text{file_name) } \]

sets a file as source stream.

\[ \text{streambuf } \ast \ C. \text{get_tgt_stream( ) returns the current target stream.} \]

\[ \text{void } \ C. \text{set_tgt_stream(} \text{streambuf } \ast \text{tgt_stream, bool own_Stream } = \text{false) } \]

sets the target stream (cf. constructor).

\[ \text{void } \ C. \text{set_tgt_file(} \text{const char } \ast \text{file_name) } \]

sets a file as target stream.

\[ \text{void } \ C. \text{reset( ) puts } C \text{ in exactly the same state as the default constructor.} \]

\[ \text{bool } \ C. \text{failed( ) returns true if an error occured.} \]

\[ \text{bool } \ C. \text{finished( ) returns true if the coding is finished.} \]

\[ \text{string } \ C. \text{get_description( ) provides a description for } C. \]
10.12 RLE for Runs of Zero ( RLE0Coder )

1. Definition

The class RLE0Coder is designed for compressing inputs which contain many runs of the
class 0x00. This coder is useful as a preprocessing stage for a Move-To-Front coder.
(This stage was suggested by D.J. Wheeler and described by P. Fenwick [32].)
The following combinations of coders often yield very good compression results:

- CoderPipe4<BWTCoder, MTF2Coder, RLE0Coder, A0Coder>
- CoderPipe4<BWTCoder, MTF2Coder, RLE0Coder, A0Coder>

#include <LEDAC/coding/RLE0.h>

2. Creation

RLE0Coder C(streambuf * src_stream = 0, streambuf * tgt_stream = 0,
bool own_streams = false);
creates an instance C which uses the given source and target
streams. If own_streams is set, then C is responsible for the destruction
of the streams, otherwise the pointers src_stream and tgt_stream
must be valid during the life-time of C.

RLE0Coder C(const char * src_file_name, const char * tgt_file_name);
creates an instance C which uses file-streams for input and output.

3. Operations

Standard Operations

void C.encode() encodes the source stream and writes the output to
the target stream.

void C.decode() decodes the source stream and writes the output to
the target stream.

uint32 C.encode_memory_chunk(const char * in_buf, uint32 in_len, char * out_buf,
uint32 out_len)
encodes the memory chunk starting at in_buf with
size in_len into the buffer starting at out_buf with size
out_len. The function returns actual length of the
encoded chunk which may be smaller than out_len. If
the output buffer is too small for the encoded data
the failure flag will be set (see below).
uint32 C.decode_memory_chunk(const char *in_buf, uint32 in_len, char *out_buf, uint32 out_len)
decodes a memory chunk. The meaning of the parameters and the return value is the same as in the
previous function.

streambuf* C.get_src_stream() returns the current source stream.

void C.set_src_stream(streambuf *src_stream, bool own_stream = false)
sets the source stream (cf. constructor).

void C.set_src_file(const char *file_name)
sets a file as source stream.

streambuf* C.get_tgt_stream() returns the current target stream.

void C.set_tgt_stream(streambuf *tgt_stream, bool own_Stream = false)
sets the target stream (cf. constructor).

void C.set_tgt_file(const char *file_name)
sets a file as target stream.

void C.reset() puts C in exactly the same state as the default constructor.

bool C.failed() returns true if an error occurred.

bool C.finished() returns true if the coding is finished.

string C.get_description() provides a description for C.
10.13 Checksummers (checksummer_base)

1. Definition

The class\textit{checksummer\_base} is the base class for all LEDA checksummers. It cannot be instantiated. In order to avoid repeating this documentation for each derived class below we will discuss its members here.

When a checksummer is used in encoding mode it can add checksums to the output stream. This is controlled by the\textit{checksums\_in\_stream} flag. If this flag is set and the block size is zero (default setting) then one checksum is appended at the end of the stream. If the flag is switched on and the block size $b$ is positive then a checksum is written for every block of $b$ characters.

When the checksummer is used in decoding mode this flag specifies whether the source stream contains checksums. If so, they will be compared against the computed checksum for the stream (or for the respective block if the block size is positive).

If you use a checksummer in a coder pipe (cf. Section 10.23) then it should be the first coder in the pipe. This ensures that the checksum is computed for the original input. Finally, we want to point out that all checksummers provide fast seek operations (cf. Section 10.22).

#include <LEDA/coding/checksum.h>

2. Creation

\begin{verbatim}
checksummer_base C(streambuf * src_stream = 0, streambuf * tgt_stream = 0, bool own_streams = false);
\end{verbatim}

creates an instance $C$ which uses the given source and target streams. If \textit{own\_streams} is set, then $C$ is responsible for the destruction of the streams, otherwise the pointers \textit{src\_stream} and \textit{tgt\_stream} must be valid during the life-time of $C$.

\begin{verbatim}
checksummer_base C(const char * src_file_name, const char * tgt_file_name = 0);
\end{verbatim}

creates an instance $C$ which uses file-streams for input and output.

3. Operations

Standard Operations

\begin{verbatim}
void C.encode( ) encodes the source stream and writes the output to the target stream.
\end{verbatim}

\begin{verbatim}
void C.decode( ) decodes the source stream and writes the output to the target stream.
\end{verbatim}
uint32 C.encode_memory_chunk(const char *in_buf, uint32 in_len, char *out_buf, uint32 out_len)

encodes the memory chunk starting at in_buf with size in_len into the buffer starting at out_buf with size out_len. The function returns actual length of the encoded chunk which may be smaller than out_len. If the output buffer is too small for the encoded data the failure flag will be set (see below).

uint32 C.decode_memory_chunk(const char *in_buf, uint32 in_len, char *out_buf, uint32 out_len)

decodes a memory chunk. The meaning of the parameters and the return value is the same as in the previous function.

streambuf * C.get_src_stream() returns the current source stream.

void C.set_src_stream(streambuf *src_stream, bool own_stream = false)

sets the source stream (cf. constructor).

void C.set_src_file(const char *file_name)

sets a file as source stream.

streambuf * C.get_tgt_stream() returns the current target stream.

void C.set_tgt_stream(streambuf *tgt_stream, bool own_stream = false)

sets the target stream (cf. constructor).

void C.set_tgt_file(const char *file_name)

sets a file as target stream.

void C.reset(bool keep_parameters = true)

puts C in the same state as the default constructor. If keep_parameters is false the parameters are set to their default values.

bool C.failed() returns true if an error occurred or a checksum in the stream does not match the computed checksum.

bool C.finished() returns true if the coding is finished.

string C.get_description() provides a description for C.

Additional Operations

string C.check() checks the source stream and returns the computed checksum. No target stream needs to be set. (If a target stream is set, check coincides with decode).
bool C.checksum_is_valid() returns whether C has a valid checksum.

string C.get_checksum_as_hex_string() returns the checksum for the stream that has been processed last as hexadecimal string.

string C.check(string str) computes a checksum for a string. (Sets source and target stream to nil.)

void C.set_checkums_in_stream_flag(bool sums_in_stream, uint32 blocksize = 0) sets the checksums_in_stream flag and the block size.

bool C.get_checkums_in_stream_flag() returns the checksums_in_stream flag.

uint32 C.get_block_size() returns the current block size.

10.14 CRC32 Checksum (CRC32Coder)

1. Definition

The class CRC32Coder computes CRC32 checksums (see ISO 3309 and ITU-T V.42 for a formal specification).

#include <LEDA/coding/checksum.h>

10.15 CRC-CCITT Checksum (CCITTCoder)

1. Definition

The class CCITTCoder computes CRC-CCITT checksums (see ISO 3309).

#include <LEDA/coding/checksum.h>

10.16 Adler32 Checksum (Adler32Coder)

1. Definition

The class Adler32Coder computes Adler32 checksums (see RFC 1950, ZLIB specification version 3.3).

#include <LEDA/coding/checksum.h>
10.17 MD5 Checksum (MD5SumCoder)

1. Definition

The class MD5SumCoder computes MD5 checksums (see RFC 1321, The MD5 Message-Digest Algorithm).

#include <LEDA/coding/checksum.h>

10.18 SHA-1 Checksum (SHACoder)

1. Definition

The class SHACoder computes SHA-1 checksums (see FIPS PUB 180-1, Secure Hash Standard).

#include <LEDA/coding/checksum.h>
10.19 Encoding Output Stream (encoding_ostream)

1. Definition

The type \textit{encoding_ostream}<$\text{Coder}$> is the encoding counterpart of the type \textit{ostream} from the C++ iostream library. Each instance \textit{os} of type \textit{encoding_ostream}<$\text{Coder}$> is attached to a \textit{streambuf} \textit{sb}. Every data that is written to \textit{os} is encoded on-the-fly by an instance of type \textit{Coder} and then written to \textit{sb}. All operations and all operators (\texttt{<<}) defined for C++ \textit{ostreams} can be applied to \textit{encoding_ostream} as well.

\begin{verbatim}
#include <LEDA/coding/coder_util.h>
\end{verbatim}

2. Creation

\textit{encoding_ostream}<$\text{Coder}$> \textit{os}(\textit{streambuf} * \textit{encoded_stream}, \textit{bool} \textit{own_stream} = \texttt{false});

creates an instance \textit{os} and attaches it to the given \textit{streambuf} object.

3. Operations

\begin{verbatim}
Coder* \textit{os}.get_coder() \quad \text{returns the instance of \textit{Coder} which is used for encoding.}

void \textit{os}.close() \quad \text{detaches \textit{os} from its \textit{streambuf} object.}
\end{verbatim}

10.20 Decoding Input Stream (decoding_istream)

1. Definition

The type \textit{decoding_istream}<$\text{Coder}$> is the decoding counterpart of the type \textit{istream} from the C++ iostream library. Each instance \textit{is} of type \textit{decoding_istream}<$\text{Coder}$> is attached to a \textit{streambuf} \textit{sb}. Whenever data is requested from \textit{is}, it reads some data from \textit{sb} and then decodes it on-the-fly with an instance of type \textit{Coder}. All operations and all operators (\texttt{>>}) defined for C++ \textit{istreams} can be applied to \textit{decoding_istream} as well.

\begin{verbatim}
#include <LEDA/coding/coder_util.h>
\end{verbatim}

2. Creation

\textit{decoding_istream}<$\text{Coder}$> \textit{is}(\textit{streambuf} * \textit{encoded_stream}, \textit{bool} \textit{own_stream} = \texttt{false});

creates an instance \textit{is} and attaches it to the given \textit{streambuf} object.
3. Operations

- Coder* is.get_coder() returns the instance of Coder which is used for encoding.
- void is.close() detaches is from its streambuf object.
- void is.finish() reads till the end of the stream and then closes is. (This is useful if Coder is a checksummer; the checksum is only verified after EOF has been read.)

10.21 Encoding File Stream (encoding_ofstream)

1. Definition

The type encoding_ofstream<Coder> is the encoding counterpart of the type ofstream from the C++ iostream library. Each instance os of type encoding_ofstream<Coder> is associated with a file. Every data that is written to os is encoded on-the-fly by an instance of type Coder and then written to the associated file. All operations and all operators (<<) defined for C++ ostream can be applied to encoding_ofstream as well.

```cpp
#include <LEDA/coding/coder-util.h>
```

2. Creation

encoding_ofstream<Coder> os(const char* file_name = 0, ios::openmode mode = ios::openmode(ios::out | ios::trunc));

creates an instance os. If file_name is specified, the stream is attached to the file with the given name.

3. Operations

- bool os.is_open() returns if os is attached to an open file.
- void os.open(const char* file_name = 0, ios::openmode mode = ios::openmode(ios::out | ios::trunc)) opens a file and attaches os to it.
- void os.close() closes the attached file and detaches os from it.
- streampos os.tellp() queries the position of the (internal) put pointer. This pointer determines the position in the (original) stream whereto the next character is written.
- ostream& os.seekp(streampos pos) sets the position of the put pointer to pos. pos must be greater than the current put pointer. The skipped characters are assumed to be zero.
Chapter 10. Lossless Compression

\[
\begin{align*}
\text{ostream} & \quad \text{os.seekp(steamoff off, ios::seekdir dir)} \\
& \quad \text{moves the put pointer by off relative to the position}
& \quad \text{determined by dir. dir can be ios::beg (beginning),}
& \quad \text{ios::cur (current position) or ios::end (last position).}
& \quad \text{The new position of the put pointer has to be greater}
& \quad \text{than its current position. The skipped characters are}
& \quad \text{assumed to be zero.}
\end{align*}
\]

\[
\begin{align*}
\text{Coder} & \quad \text{os.get_coder( )} \\
& \quad \text{returns the instance of Coder which is used for encoding.}
\end{align*}
\]

10.22 Decoding File Stream (decoding_ifstream)

1. Definition

The type \texttt{decoding_ifstream} is the decoding counterpart of the type \texttt{ifstream} from the C++ iostream library. Each instance \textit{is} of type \texttt{decoding_ifstream} is associated with a file. Whenever data is requested from an instance \textit{is} of type \texttt{decoding_ifstream}, \textit{is} reads some data from the associated file and then decodes it on-the-fly with an instance of type \textit{Coder}. All operations and all operators (\texttt{>>}) defined for C++ istreams can be applied to \texttt{decoding_ifstream} as well.

\texttt{decoding_ifstream} supports random access read operations by providing seek operations. For this purpose it maintains a so-called \textit{get pointer}. This pointer specifies the position in the decoded data stream from which the next read operation will extract its data. After the operation the get pointer refers to the position immediately behind the last extracted character. E.g., if you are not interested in the first 5000 characters and you want to extract the 10 characters in positions 5000 – 5009, you can perform a seek operation which moves the get pointer to position 5000 and then read 10 characters. (After that the get pointer will refer to position 5010.) This seek operation will usually be faster than skipping 5000 characters by hand.

Seek operations are supported no matter which coder is plugged into \texttt{decoding_ifstream}. However, some coders (like checksummers) provide fast seek operations. This manual mentions these coders explicitly. Moreover, we provide a class called \texttt{BlockCoder} (in Section 10.25) which allows you to speed up seeks for any coder.

\#include \texttt{<LEDA/coding/coder_util.h>}

2. Creation

\texttt{decoding_ifstream} \texttt{\langle Codger\rangle} \texttt{\langle const char * file_name = 0,}
& \texttt{ios::openmode mode = ios::in\rangle; }
\texttt{creates an instance is. If file_name is specified, the stream is attached to the file with the given name.}
3. Operations

**bool** `is.is_open()` returns if `is` is attached to an open file.

**void** `is.open(const char *file_name = 0, ios::openmode mode = ios::in)`
opens a file and attaches `is` to it.

**void** `is.close()` closes the attached file and detaches `is` from it.

**void** `is.finish()` reads till the end of the stream and then closes `is`.
(This is useful if `Coder` is a checksummer; the checksum is only verified after EOF has been read.)

**streampos** `is.tellg()` queries the position of the (internal) get pointer. This pointer determines the position in the (decoded) stream from which the next character is read.

**istream&** `is.seekg(streampos pos)`
sets the position of the get pointer to `pos`.

**istream&** `is.seekg(streamoff off, ios::seekdir dir)`
moves the get pointer by `off` relative to the position determined by `dir`. `dir` can be `ios::beg` (beginning), `ios::cur` (current position) or `ios::end` (last position).

**Coder** `is.get_coder()` returns the instance of `Coder` which is used for decoding.
10.23 Coder Pipes (CoderPipe2)

1. Definition

The type \textit{CoderPipe2< Coder1, Coder2 >} can be used to combine two coders of type \textit{Coder1} and \textit{Coder2} into one single coder. This works in an analogous way as a pipe of the operating system: In encoding mode the original input is processed by \textit{Coder1}. Its output is fed into \textit{Coder2}. The output of \textit{Coder2} becomes the output of the pipe. In decoding mode the situation is reversed: The data is sent through \textit{Coder2} first and then through \textit{Coder1}.

We also provide pipes for combining more than two coders (up to six): \textit{CoderPipe3}, \ldots, \textit{CoderPipe6}. (Since these classes have a similar interface as \textit{CoderPipe2}, we do not include manual pages for them.)

\begin{verbatim}
#include <LEDA/coding/coder_util.h>
\end{verbatim}

2. Types

\textit{CoderPipe2< Coder1, Coder2 >::coder1}

the type \textit{Coder1}.

\textit{CoderPipe2< Coder1, Coder2 >::coder2}

the type \textit{Coder2}.

3. Creation

\textit{CoderPipe2< Coder1, Coder2 > C(streambuf * src_stream = 0,
path streambuf * tgt_stream = 0,
bool own_streams = false);}

creates an instance \textit{C} which uses the given source and target streams. If \textit{own_streams} is set, then \textit{C} is responsible for the destruction of the streams, otherwise the pointers \textit{src_stream} and \textit{tgt_stream} must be valid during the life-time of \textit{C}.

\textit{CoderPipe2< Coder1, Coder2 > C(const char * src_file_name,
const char * tgt_file_name);}

creates an instance \textit{C} which uses file-streams for input and output.

4. Operations

Standard Operations

\begin{verbatim}
void C.encode() encodes the source stream and writes the output to the target stream.
void C.decode() decodes the source stream and writes the output to the target stream.
\end{verbatim}
`uint32 C.encode_memory_chunk(const char *in_buf, uint32 in_len, char *out_buf, uint32 out_len)`

encodes the memory chunk starting at `in_buf` with size `in_len` into the buffer starting at `out_buf` with size `out_len`. The function returns actual length of the encoded chunk which may be smaller than `out_len`. If the output buffer is too small for the encoded data the failure flag will be set (see below).

`uint32 C.decode_memory_chunk(const char *in_buf, uint32 in_len, char *out_buf, uint32 out_len)`

decodes a memory chunk. The meaning of the parameters and the return value is the same as in the previous function.

`streambuf * C.get_src_stream()` returns the current source stream.

`void C.set_src_stream(streambuf *src_stream, bool own_stream = false)`

sets the source stream (cf. constructor).

`void C.set_src_file(const char *file_name)`

sets a file as source stream.

`streambuf * C.get_tgt_stream()` returns the current target stream.

`void C.set_tgt_stream(streambuf *tgt_stream, bool own_Stream = false)`

sets the target stream (cf. constructor).

`void C.set_tgt_file(const char *file_name)`

sets a file as target stream.

`void C.reset(bool keep_parameters = true)`

puts `C` in the same state as the default constructor. If `keep_parameters` is false the parameters of all coders in the pipe are set to their default values.

`bool C.failed()` returns `true` if an error occurred or a checksum in the stream does not match the computed checksum.

`bool C.finished()` returns `true` if the coding is finished.

`string C.get_description()` provides a description for `C`.

**Additional Operations**

`Coder1 * C.get_coder1()` returns the currently used instance of `Coder1`.
\textit{void} \hspace{1em} \textit{C.set\_coder1(\textit{Coder1} * \textit{c1}, bool \textit{own\_coder} = \textit{false})}

sets the instance of \textit{Coder1}. If \textit{own\_coder} is \textit{true}, then the \textit{c1} is deleted by \textit{C}. Otherwise \textit{c1} is not deleted by \textit{C} and the pointer \textit{c1} must be valid during the life-time of \textit{C}.

\textit{Coder2*} \hspace{1em} \textit{C.get\_coder2( )} \hspace{1em} \text{returns the currently used instance of} \hspace{1em} \textit{Coder2}.

\textit{void} \hspace{1em} \textit{C.set\_coder2(\textit{Coder2} * \textit{c2}, bool \textit{own\_coder} = \textit{false})}

sets the instance of \textit{Coder2}. The parameter \textit{own\_coder} is explained above.
10.24 Automatic Decoder (AutoDecoder)

1. Definition

An instance \( C \) of AutoDecoder can be used for decoding any stream that has been encoded with a LEDA coder or a pipe of LEDA coders. Thus this class is useful if you want to decode a stream and have forgotten which combination of coders you have used to encode it. This class is also helpful if the encoding method is not known at compile-time. Then the decoding method cannot be fixed at compile-time either but it has to be determined at run-time.

```cpp
#include <LEDA/coding/auto_decoder.h>
```

2. Types

In order to facilitate the usage of AutoDecoder with `decoding_istream` we provide the typedef `autodecoding_istream` as a shorthand for `decoding_istream<AutoDecoder>`.

3. Creation

```cpp
AutoDecoder C(streambuf *src_stream = 0, streambuf *tgt_stream = 0,
                bool own_streams = false);
```
creates an instance \( C \) which uses the given source and target streams. If `own_streams` is set, then \( C \) is responsible for the destruction of the streams, otherwise the pointers `src_stream` and `tgt_stream` must be valid during the life-time of \( C \).

```cpp
AutoDecoder C(const char *src_file_name, const char *tgt_file_name);
```
creates an instance \( C \) which uses file-streams for input and output.

4. Operations

Standard Operations

```cpp
void C.decode() decodes the source stream and writes the output to the target stream.

streambuf* C.get_src_stream() returns the current source stream.

uint32 C.decode_memory_chunk(const char *in_buf, uint32 in_len, char *out_buf, uint32 out_len)
```
decodes the memory chunk starting at `in_buf` with size `in_len` into the buffer starting at `out_buf` with size `out_len`. The function returns actual length of the encoded chunk which may be smaller than `out_len`. If the output buffer is too small for the decoded data the failure flag will be set (see below).
void \texttt{C.set.src.stream(streambuf \ast src\_stream, bool own\_stream = false)}
\hspace{1em} sets the source stream (cf. constructor).

void \texttt{C.set.src.file(const char \ast file\_name)}
\hspace{1em} sets a file as source stream.

\texttt{streambuf\ast C.get.tgt.stream( )} \hspace{1em} returns the current target stream.

void \texttt{C.set.tgt.stream(streambuf \ast tgt\_stream, bool own\_Stream = false)}
\hspace{1em} sets the target stream (cf. constructor).

void \texttt{C.set.tgt.file(const char \ast file\_name)}
\hspace{1em} sets a file as target stream.

void \texttt{C.reset( )} \hspace{1em} puts \texttt{C} in exactly the same state as the default constructor.

bool \texttt{C.failed( )} \hspace{1em} returns \texttt{true} if an error occurred.

bool \texttt{C.finished( )} \hspace{1em} returns \texttt{true} if decoding is finished.

\texttt{string C.get.description( )} \hspace{1em} provides a description for \texttt{C}. After decoding this includes a description of the coder that has been used for encoding the stream.

**Additional Operations**

\texttt{coder\_base C.get.coder( )} \hspace{1em} after decoding this function returns the coder that has actually been used to decode the stream. E.g., if the original source stream has been encoded with an instance of type \texttt{A0Coder} then the function returns an instance of \texttt{A0Coder}.
10.25 Block Coder (BlockCoder)

1. Definition

An instance of type `BlockCoder` encodes a source stream as follows: It divides the stream into equally sized blocks and encodes each block separately with an instance of type `Coder`. If you later want to decode only parts of the encoded data then you do not have to decode the entire data but only the appropriate blocks, which is usually much faster. (The block size can be specified by the user; please note that the main memory of your computer has to be large enough to hold one block.)

The class `BlockCoder` has been designed to speed up the seek operations of the class `decoding_ifstream` (see Section 10.22). In order to gain the speed up simply replace the template parameter `Coder` in `decoding_ifstream` by `BlockCoder` (see also the example below).

There is an important precondition for `BlockCoder`: In encoding mode the target stream must support seek operations which query the current position (i.e. `seek(0, ios::cur, ios::out)`). In decoding mode the source stream must be capable of random seek operations. These conditions are surely fullfilled if the respective stream is a file. Moreover, if `BlockCoder` is used within a coder pipe the pipe will make sure that it operates properly. However, this configuration only makes sense if all coders in the pipe optimize seek operations. Then the whole pipe will offer fast seek operations.

```
#include <LEDA/coding/block_coder.h>
```

2. Creation

```
BlockCoder C(streambuf * src_stream = 0, streambuf * tgt_stream = 0,
bool own_streams = false,
uint32 block_size = DefaultBlockSize);
```

creates an instance `C` which uses the given source and target streams. If `own_streams` is set, then `C` is responsible for the destruction of the streams, otherwise the pointers `src_stream` and `tgt_stream` must be valid during the life-time of `C`. The initial block size is `block_size`.

```
BlockCoder C(const char * src_file_name, const char * tgt_file_name,
uint32 block_size = DefaultBlockSize);
```

creates an instance `C` which uses file-streams for input and output. The block size is set to `block_size`. 
3. Operations

Standard Operations

```c
void C.encode() encodes the source stream and writes the output to the target stream.

void C.decode() decodes the source stream and writes the output to the target stream.

uint32 C.encode_memory_chunk(const char * in_buf, uint32 in_len, char * out_buf, uint32 out_len)
    encodes the memory chunk starting at in_buf with size in_len into the buffer starting at out_buf with size out_len. The function returns actual length of the encoded chunk which may be smaller than out_len. If the output buffer is too small for the encoded data the failure flag will be set (see below).

uint32 C.decode_memory_chunk(const char * in_buf, uint32 in_len, char * out_buf, uint32 out_len)
    decodes a memory chunk. The meaning of the parameters and the return value is the same as in the previous function.

streambuf* C.get_src_stream() returns the current source stream.

void C.set_src_stream(streambuf* src_stream, bool own_stream = false)
    sets the source stream (cf. constructor).

void C.set_src_file(const char * file_name)
    sets a file as source stream.

streambuf* C.get_tgt_stream() returns the current target stream.

void C.set_tgt_stream(streambuf* tgt_stream, bool own_Stream = false)
    sets the target stream (cf. constructor).

void C.set_tgt_file(const char * file_name)
    sets a file as target stream.

void C.reset(bool keep_block_size = true)
    puts C in the same state as the default constructor. If keep_block_size is false, the block size is set to the default value.

bool C.failed() returns true if an error occurred.

bool C.finished() returns true if the coding is finished.
```
**Additional Operations**

- `uint32 C.get_block_size( )` returns the current block size.
- `void C.set_block_size(uint32 block_size)` sets the block size.
- `Coder* C.getCoder( )` returns the instance of Coder which is actually used for encoding and decoding.

### 4. Example

The following example shows how BlockCoder can be used to speed up seek operations. We encode a file of about 3.85 MB, once with A0Coder and once with BlockCoder<A0Coder>. We feed each encoded file into an autodecoding_ifstream and perform random seeks. The file encoded with BlockCoder is slightly longer but seek operations become much faster.

```cpp
#include <LEDA/coding/compress.h>
#include <LEDA/core/random_source.h>
#include <LEDA/system/file.h>
#include <LEDA/system/timer.h>
using namespace LEDA;

void test_random_seek(istream& is)
{
    const int buffer_sz = 100; char buffer[buffer_sz];
    is.seekg(0, ios::end); streamoff size = (streamoff) is.tellg();
    // yields the size of the original file

    random_source rs;
    rs.set_seed(0); rs.set_range(0, size - buffer_sz - 1);

    timer t("seek"); t.start();
    for (int i = 0; i < 100; ++i) {
        is.seekg(rs()); is.read(buffer, buffer_sz); // random seek
    }
    t.stop();
}

int main()
{
    typedef A0Coder Coder;
    string in_file = "D:\ctemp\bible.txt"; // 3,85MB
```
string out_file1 = tmp_file_name(), out_file2 = tmp_file_name();

Coder c1(in_file, out_file1);
c1.encode(); c1.reset();
cout << "encoded size: " << size_of_file(out_file1) << endl;
autodecoding_ifstream is1(out_file1);
test_random_seek(is1); is1.close();
// output on my machine: enc. sz: 2201723 / Timer(seek): 51.36 s

BlockCoder<Coder> c2(in_file, out_file2);
c2.encode(); c2.reset();
cout << "encoded size: " << size_of_file(out_file2) << endl;
autodecoding_ifstream is2(out_file2);
test_random_seek(is2); is2.close();
// output on my machine: enc. sz: 2207892 / Timer(seek): 1.91 s

delete_file(out_file1); delete_file(out_file2); return 0;
10.26 Memory Streambuffer (memory_streambuf)

1. Definition

An object \( mb \) of the class \( \text{memory}\_\text{streambuf} \) can be used as source or target stream for a coder and allows encoding and decoding in memory (i.e. without file accesses). Every read or write operation is forwarded to a buffer in memory. This buffer is directly accessible by the user. The length of this buffer is called the capacity of \( mb \). The size of \( mb \) is the number of characters that can be read from or written to the stream before an underflow/overflow occurs. (In that case the corresponding stream operation returns EOF (end-of-file).) Thus the capacity is the maximum size that \( mb \) can have without reallocating its buffer.

```c
#include <LEDA/coding/memory_streambuf.h>
```

2. Creation

```c
memory_streambuf \( mb \)(streamsize buf\_sz = 0, bool wipe\_buf = false);
```
creates an object \( mb \) with capacity and size \( buf\_sz \). The parameter \( wipe\_buf \) determines whether \( \text{wipe}\_\text{buffer} \) is called when the buffer is not needed any more.

```c
memory_streambuf \( mb \)(char *buf, streamsize buf\_sz, bool own\_buf = false,
    bool wipe\_buf = false);
```
creates an object \( mb \) with the memory buffer \( buf \) and sets the capacity and the size to \( buf\_sz \). If \( own\_buf/wipe\_buf \) is true, then \( buf \) is deleted/wiped by \( mb \) when it is not used anymore. Note that \( buf \) is not copied but it is used as the internal buffer of \( mb \).

3. Operations

```c
void \( mb\).reset() moves the internal get and put pointers to the beginning of the buffer and clears the underflow/overflow flags.
```

```c
streamsize \( mb\).get\_capacity() returns the current capacity.
```

```c
streamsize \( mb\).get\_size() returns the current size.
```

```c
void \( mb\).set\_size(streamsize n)
```
changes the size to \( n \) and calls \( \text{reset} \). (If \( n \) exceeds the capacity then a new buffer with sufficient capacity is allocated. The contents of the old buffer are not copied.)
void \texttt{mb.truncate(streamsize n)}

also changes the size to \( n \), but never allocates a new buffer.

\textit{Precondition:} \( n \leq \) current capacity.

\texttt{char* mb.get\_buffer()}

returns the memory buffer of \texttt{mb}.

\texttt{void mb.set\_buffer(char* buf, streamsize buf\_sz, bool own\_buf = false, bool wipe\_buf = false)}

makes \( buf \) the new memory buffer and changes capacity and size to \( buf\_sz \). (The meaning of \texttt{own\_buf} and \texttt{wipe\_buf} is the same as in the second constructor.)

void \texttt{mb.wipe\_buffer()}

like reset, in addition all bytes in the memory buffer are set to zero.

\texttt{streamsize mb.in\_avail()}

returns how many characters can be read from \texttt{mb} before an underflow occurs.

\texttt{streamsize mb.in\_count()}

returns how many characters have been read from \texttt{mb} since the last reset.

\texttt{streamsize mb.out\_avail()}

returns how many characters can be written to \texttt{mb} before an overflow occurs.

\texttt{streamsize mb.out\_count()}

returns how many characters have been written to \texttt{mb} since the last reset.

\texttt{bool mb.had\_underflow()}

returns whether an underflow has occurred since the last reset.

\texttt{bool mb.had\_overflow()}

returns whether an overflow has occurred since the last reset.
Chapter 11
Symmetric Key Cryptography

LEDAs covers the following aspects of symmetric key cryptography:

- encryption (= enciphering) and decryption (= deciphering) of a stream:
  Encryption takes as input some plaintext (a stream of data) and a key (a small piece of secret data) and outputs some ciphertext. Decryption is the reverse operation: its input consists of the ciphertext and a key, and its output is the original plaintext. The idea is that the plaintext cannot be recovered from the ciphertext without knowing the correct decryption key. This property is called secrecy. It allows you to send the ciphertext through an insecure channel or to store it on an insecure filesystem. There are algorithms where the decryption key differs from the corresponding encryption key. An algorithm which uses the same key for both operations is called symmetric. (All algorithms provided by LEDA have this property.)

- authentication of a stream:
  Authentication takes as input a message and a key and outputs a so-called Message Authentication Code (MAC). A MAC can be seen as a key-dependent checksum. The idea is that nobody can generate a valid MAC for a message without knowing the key. So if message and MAC are sent through an insecure channel then the receiver – given that he knows the key – can verify that the text has not been tampered with (integrity) and that it originates from someone knowing the key (authentication). MACs are also useful if you want to make sure that no intruder (in particular no virus) can alter a file on your system. (If ordinary checksums were used the intruder could change the file and then simply update corresponding checksum.)

- key generation:
  The security of a cryptographic system depends heavily on the strength of its keys. If an attacker can obtain your keys he can decrypt your messages or fake MACs no matter how good the encryption and authentication algorithms are. In order to support the user LEDA provides secure methods to generate a key from a human-readable passphrase. Of course, the user has the responsibility to chose a good passphrase, i.e. a phrase that cannot be guessed easily by an attacker.

All these topics will be discussed in detail later. We want to point out that secrecy and authentication are two orthogonal concepts. If you use only authentication, nobody will
be able to change your message without being detected but everybody can read it. On the other hand, if you use only encryption then nobody can read your message but an attacker could change it. At first sight this does not seem to make much sense because the attacker can only alter ciphertext and he will not be able to predict what the corresponding plaintext will look like. However, this may still cause damage. Imagine for example a satellite that is remote controlled via encrypted but unauthenticated commands. If an attacker manages to make the satellite listen to his commands, he cannot exploit the device but he might be able to make it leave its orbit and crash.

Hence, sometimes secrecy and authentication must be used together to achieve the desired security. LEDA makes it easy to combine these different aspects. The example below will illustrate this. The program is similar to the opening example of Chapter 10. It consists of three parts: key generation, encoding and decoding. Each part will be explained below.

```cpp
#include <LEDA/coding/crypt.h> // contains all cryptography classes

using namespace leda;

void generate_keys(CryptKey& auth_key, CryptKey& cipher_key)
{
    // key generation (two keys)
    CryptByteString passphrase = CryptKey::read_passphrase("Passphrase: ");
    CryptByteString salt(1);
    salt[0] = 'a'; // for authentication
    auth_key = CryptKey::generate_key(128/8, passphrase, salt);
    salt[0] = 'c'; // for enciphering/deciphering
    cipher_key = CryptKey::generate_key(128/8, passphrase, salt);
}

int main()
{
    string str = "Hello World";
    CryptKey auth_key, cipher_key;

    // encode: MAC -> compress -> encipher
    typedef CoderPipe3<OMACCoder<>, PPMIICoder, CBCCoder<> > CryptCoder;
    encoding_ofstream<CryptCoder> out("foo");
generate_keys(auth_key, cipher_key);
    out.get_coder()->get_coder1()->set_key(auth_key);
    out.get_coder()->get_coder3()->set_key(cipher_key);
    out << str << "\n";
    out.close();
    if (out.fail()) std::cout << "error writing foo" << "\n";

    // decode: decipher -> decompress -> MAC
    decoding_ifstream<CryptCoder> in("foo");
generate_keys(auth_key, cipher_key);
in.get_coder()->get_coder1()->set_key(auth_key);
```

in.get_coder()->get_coder3()->set_key(cipher_key);
str.read_line(in);
in.finish(); // read till EOF is reached and then close "in"
if (in.fail()) std::cout << "error authenticating foo" << "\n";

std::cout << "decoded string: " << str << "\n";

return 0;
}

In the first part of the program (function generate_keys) two keys are generated. The input for each generation is a passphrase and a salt. The passphrase is a human-readable and easy-to-remember string which must be kept secret. The salt is an array of bytes (in the example just one byte) which should be unique but it can be made public without endangering security. It allows to generate different keys from a single passphrase. In the example we generate one key for authentication and one key for encryption. More information on key generation can be found in Section 11.2.

In the second part (beginning of main) we encode a message. We use a coder pipe CryptCoder which first authenticates (OMACCoder), then compresses (PPMIIICoder) and finally encrypts (CBCCoder) its source stream. Observe that the output of the OMACCoder (in encoding mode) is the original input plus the MAC for this input. (In decoding mode this MAC will be verified automatically.) In order to use the CryptCoder we construct an encodingofstream called out. After setting the keys of the cryptographic coders we can use out just like an ordinary ofstream.

We want to point out that the order of the coders in the pipe is not arbitrary. It makes sense to put the authentication at the very beginning because you want to authenticate the original plaintext and not some compressed or encrypted version of it. Moreover, it is important to place compression before encryption for two reasons: Since compression usually destroys patterns and structures in the plaintext it can be seen as a way of obscuring the plaintext, which can be a helpful preprocessing step for encryption. But the following is even more important: As encryption transforms its input into a seemingly random stream applying compression after encryption usually increases the length of stream. (Some cryptographers even warn not to trust an encryption algorithm if its output can be compressed well [80].)

Now we describe the decoding process (at the end of main). It looks very similar to the encoding process but instead of encoding ofstream we use its counterpart decoding ifstream. After the key set up the stream in can be used as a usual C++ input stream. However, there is one subtle point that we want to highlight: When we are done with in we do not call the close but the finish method. This reads the stream till its end before closing it. This ensures that the OMACCoder verifies the MAC and reports an error if the computed MAC and the MAC found in the stream differ. (If in were closed before its end is reached then no authentication would be performed and no error would be signaled. The reader is invited to try this out.)

We want to discuss a feature of LEDA that can simplify the decoding process. Suppose a friend of yours has sent you an authenticated and encrypted file over the internet, and he has provided you the authentication and the encryption key (via a secure channel). But unfortunately he did not tell you which coders he actually used for encoding his message.
Then you can use the *CryptAutoDecoder* to decode his message. This class is able to automatically reconstruct the coder (or the coder pipe) that was used for encoding the stream:

```cpp
decoding_ifstream<CryptAutoDecoder> in("foo");
in.get_coder()->add_key(auth_key);
in.get_coder()->add_key(cipher_key);
```

Of course, LEDA also supports encryption and decryption of files:

```cpp
// encryption
CoderPipe3<OMACCoder>, PPMICoder, CBCCoder> coder;
coder.set_src_file("input.txt"); coder.set_tgt_file("output");
coder.get_coder1()->set_key(auth_key);
coder.get_coder3()->set_key(cipher_key);
coder.encode();

// decryption
CryptAutoDecoder auto("output", "input.txt");
auto->add_key(auth_key); auto->add_key(cipher_key);
auto.decode();
```

We want to discuss some security assumptions made by LEDA, i.e. some preconditions that must be fulfilled in order to ensure the security of your data:

- **Your passphrases and your keys must be safe.**
  That means that the user is responsible for choosing passphrases that cannot be guessed easily by an attacker. One way to “generate” such a passphrase is to take a sentence and to deliberately add some typos like “LEDA’s crytografic algorhythms are very good!!”. (If you get tired of such long phrases you could use a phrase that consists basically of the first letter of every word: “L’scaa++!”) If passphrases or keys are stored on a disc this disc must be protected against an attacker. (Also take care of temporary copies, they must be thoroughly wiped out. Simply deleting a file does not suffice because its contents can still be recovered with certain tools. Some specialists are even capable of recovering data on a hard disc that has been overwritten once.)

- **The main memory of your computer must be safe.**
  While cryptographic operations are performed passphrases and keys are stored in plaintext in the main memory of the computer. Therefore the user must make sure that no attacker can inspect any memory used by his process (in particular while the process is running). Some features of LEDA support the user in making this kind of attack harder: The class *CryptByteString* and the derived class *CryptKey* overwrite their memory before they free it and return it to the system. We want to point out that this precaution can be foiled by the operating system. If it swaps some security sensitive part of the memory to a swap file on a hard disc then this data may remain there after the termination of your process. (On some platforms LEDA prevents this swapping, see Section 11.2.)
We want to address another important issue. Some readers might be worried about the fact that a potential attacker can purchase a copy of the source code of LEDA. One may wonder if this is a security risk. Fortunately, this is not the case. The cryptographic algorithms in LEDA have been designed under the assumption that a potential attacker knows the algorithm (but not the key). All the algorithms have appeared in scientific publications and successfully passed thorough cryptographic analysis. The fact that the source code can be purchased allows you to verify that the developers of LEDA did not put any trap-doors into the code which allow them to decrypt data without knowing the key. (This means: If you loose your passphrase or your key then we are not able to help you to recover your encrypted data.)
CHAPTER 11. SYMMETRIC KEY CRYPTOGRAPHY

11.1 Secure Byte String (CryptByteString)

1. Definition

An instance \( s \) of the class CryptByteString is basically a string of bytes. When \( s \) is not used anymore its memory is wiped out (by overwriting it a couple of times) before the memory is freed and returned to the system. The goal is to prevent an attacker from reading security sensitive data after your process has terminated. We want to point out that this mechanism can be foiled by the operating system: If it swaps the memory occupied by \( s \) to a swap file on a hard disc then the data will not be erased by \( s \). (Some platforms offer to lock certain parts of the memory against swapping. CryptByteString uses this feature on Windows NT/2000/XP to protect its memory.)

As we have stated above \( s \) can be used like a string or an array of bytes. The size \( n \) of \( s \) is the number of bytes in \( s \), they are indexed from 0 to \( n - 1 \).

**Important:** If you create a CryptByteString \( s \) from a C-style array or a string, or if you convert \( s \) to a string, then only the memory of \( s \) will be wiped out but not the memory of the array or the string.

```c
#include <LEDA/coding/crypt_key.h>
```

2. Creation

CryptByteString \( s \); creates an empty string.

CryptByteString \( s \)(uint16 size);

creates a string of the given size. All bytes in \( s \) are set to zero.

CryptByteString \( s \)(const byte *bytes, uint16 num_bytes);

creates a copy of the array \( bytes \) of size \( num_bytes \).

CryptByteString \( s \)(const char *str);

creates a copy of the C-style string \( str \). (The \'\0\' character at the end is not copied.)

3. Operations

\( \text{uint16} \quad s.\text{get\_size()} \quad \text{returns the size of } s. \)

\( \text{bool} \quad s.\text{is\_empty()} \quad \text{returns true iff } s \text{ is empty.} \)

\( \text{void} \quad s.\text{clear()} \quad \text{makes } s \text{ the empty string.} \)

\( \text{const byte*} \quad s.\text{get\_bytes()} \quad \text{returns the internal byte-array of } s. \)

\( \text{byte}& \quad s[\text{uint16} \ idx] \quad \text{returns the byte at position } idx. \)

*Precondition:* \( 0 \leq idx \leq s.\text{get\_size()} - 1. \)
11.2 Key for Cryptography (CryptKey)

1. Definition

Instances of the class CryptKey store keys for cryptographic algorithms. CryptKey is derived from CryptByteString and hence, its instances also wipe out their memory upon destruction. Apart from the operations of its base class CryptKey provides some static operations which can be helpful for key generation.

```cpp
#include <LEDA/coding/crypt_key.h>
```

2. Creation

```cpp
CryptKey k;              // creates an empty key.
CryptKey k(uint16 size); // creates a key of the given size. All bytes in k are set to zero.
```
CryptKey k(const byte * bytes, uint16 num_key_bytes);
initializes k with a copy of the array bytes of size num_bytes.

CryptKey k(const CryptByteString& byte_str);
initializes k with byte_str.

CryptKey k(const char * hex_str);
initializes k with the hexadecimal representation in hex_str.

Key generation

A key generation algorithm takes as input a (partially) secret seed s of arbitrary length and outputs a key k of the desired size. The gist of this process is to apply a secure hash function H to the seed and to use the returned hash value as the key, i.e. \( k = H(s) \). In our implementation we use the checksum of the SHACoder, which is 20 bytes long. Since we want to be able to generate a key of size \( n \neq 20 \) we use the following approach: We divide the key into \( p = \lceil n/20 \rceil \) portions \( k_1, \ldots, k_p \), where the size of \( k_1, \ldots, k_{p-1} \) is 20 and the size of \( k_p \) is at most 20. We set \( k_i = H(s \circ n \circ i) \). (Here \( "x \circ y" \) denotes the concatenation of \( x \) and \( y \), and \( k_p \) is possibly a proper suffix of the returned hash value.) Then we have \( k = k_1 \circ \ldots \circ k_p \).

The question is now what to use as seed for the key generation. The first idea that comes to mind is to use the passphrase supplied by the user. In that case the process would be vulnerable to the so-called dictionary attack. This works as follows: Since many users tend to choose their passphrases carelessly, the attacker builds a dictionary of common phrases and their corresponding keys. This gives him a set of “likely” keys which he tries first to break the system. One way to make this attack harder is to apply the hash function several times, i.e. instead of \( k = H(s) \) we set \( k = H(H(\ldots H(s))) \). Even if we apply \( H \) some thousand times it does not slow down the generation of a single key noticeably. However, it should slow the generation of a whole dictionary considerably.

An even more effective counter measure is to use something that is called salt. This is a string which is appended to the passphrase to form the actual seed for the key generation. The salt does not have to be kept secret, the important point is that each key generation uses a different salt. So it is impossible to reuse a dictionary built for a specific salt. The salt is also useful if you want to reuse a passphrase. E.g., if you want to authenticate and to encrypt a file you might refrain from remembering two different phrases. But since it is not a good idea to use the same key twice you could generate two different keys by using two different salts.

CryptKey CryptKey::generate_key(uint16 key_size,
const CryptByteString& seed,
uint32 num_iterations)
generates a key of size key_size by applying a hash function num_iterations times to the given seed.
11.2. KEY FOR CRYPTOGRAPHY (CRYPTKEY)

**CryptKey**

CryptKey::generate_key(uint16 key_size,  
const CryptByteString& passphrase,  
const CryptByteString& salt =  
CryptByteString(),  
uint32 num_iterations = 4096)
generates a key from a passphrase and a salt. The seed for the generation is simply passphrase ◦ salt.

**CryptByteString**

CryptKey::generate_salt(uint16 salt_size)
generates a salt as follows: If salt_size is at least size of(date) + 4 then a representation of the current date is stored in the last size of(date) bytes of the salt. The remaining bytes are filled with pseudo-random numbers from a generator which is initialized with the current time.

**CryptKey**

CryptKey::generate_key_and_salt(uint16 key_size, uint16 salt_size,  
const CryptByteString& passphrase,  
CryptByteString& salt,  
uint32 num_iterations = 4096)
first some salt is generated (see above); then a key is generated from this salt and the given passphrase.

**CryptByteString**

CryptKey::read_passphrase(const string& prompt,  
uint16 min_length = 6)
writes the prompt to stdout and then reads a passphrase from stdin until a phrase with the specified minimum length is entered. (While the phrase is read stdin is put into unbuffered mode.)
11.3 Encryption and Decryption with Ciphers

A *stream-cipher* is a coder that encrypts or decrypts streams of data. An example for such a coder is the class `CBCCoder` that we have already seen in the sample program at the beginning of this chapter. Every stream-cipher in LEDA uses a block-cipher as building block. A *block-cipher* operates on “small” fixed-size blocks of plaintext or ciphertext – usually 64 or 128 bits. Mathematically, a block-cipher can be seen as pair of two functions $E_K$ and $D_K$ that depend on a key $K$. $E_K$ takes as input a block $B$ of size $b$ and returns the encrypted block (also of size $b$), $D_K$ describes the decryption operation. We have $D_K(E_K(B)) = B$ for all blocks $B$ of size $b$ and every admissible key $K$.

A stream-cipher is usually built by combining a block-cipher and some sort of feedback via some simple operations. The most simple stream-cipher is the *electronic codebook* (ECB) mode: No chaining is used. The plaintext is partitioned into $n$ blocks $P_1, \ldots, P_n$ of size $b$. (If the last block $P_n$ is shorter than $b$ it is padded appropriately.) The ciphertext blocks $C_1, \ldots, C_n$ are obtained by applying the block-cipher to each block of the plaintext. More formally, $C_i = E_K(P_i)$. Decryption is simple, too: $P_i = D_K(C_i)$. This simplicity has an important drawback: If a block $B$ appears several times in the plaintext each occurrence encrypts to the same ciphertext block. This can be exploited by an attacker: Suppose that the attacker knows $P_1$ (maybe because every plaintext starts with a fixed header) then he is able to detect every occurrence of $P_1$ in the plaintext without knowing the key!

To overcome this serious drawback stream-ciphers with feedback have been invented. They hide patterns in the plaintext in the ciphertext. In *cipher block chaining* (CBC) mode the current plaintext block is XORRed with the previous ciphertext block (as feedback) before encryption. In short, $C_i = E_K(P_i \oplus C_{i-1})$. And decryption works as follows: $P_i = D_K(C_i \oplus C_{i-1})$. In order to compute $C_1$ we need a so-called initialization vector (IV) $C_0$. The IV does not have to be secret but it should be unique for every plaintext stream that is encrypted with a particular key $K$. The reason is the following: If you encrypt two streams with the same key and the same IV they will encrypt to same ciphertext up to the first position where the two plaintexts differ.

Another feedback mode is the *cipher feedback* (CFB) mode. It also uses an initialization vector $C_0$. Encryption and decryption work as follows: $C_i = P_i \oplus E_K(C_{i-1})$ and $P_i = C_i \oplus E_K(C_{i-1})$. (Observe that the block-cipher is only used in encryption mode.) The IV for CFB **must** be unique (for CBC it **should** be unique), otherwise a cryptoanalyst can recover the plaintext (see [80, Chapter 9.6]).

Finally, we want to describe the *output feedback* (OFB) mode. It computes a feedback stream $S_0, S_1, \ldots, S_n$, where $S_0$ is an IV that should be unique. Despite the name of this mode, the feedback stream is completely internal it depends neither on the plaintext nor on the ciphertext: $S_i = E_K(S_{i-1})$. This mode looks as follows: $C_i = P_i \oplus S_i$ and $P_i = C_i \oplus S_i$.

Let us compare the four stream-ciphers from above. In terms of efficiency they are almost identical. The number of ciphertext blocks equals the number of plaintext blocks (not counting the IV). ECB is the fastest mode but the total running time is dominated by the time consumed by the block-cipher, the XOR operations used in the other modes are negligible. With respect to fault tolerance OFB is the best: One bit error in the ciphertext affects only one bit in the decrypted plaintext. In ECB mode one bit in the ciphertext affects the complete corresponding plaintext block. For CBC it affects the full
11.3. ENCRYPTION AND DECRYPTION WITH CIPHERS

corresponding block in the plaintext and one bit in the next block. And in CFB mode
the corresponding bit in the plaintext is affected as well as the complete next block. As
to security ECB is clearly the worst because patterns in the plaintext are not hidden.
The other three modes hide these patterns and they are almost comparable in terms of
security. CBC has some small advantages (cf. [80, Chapter 9.11]). So if you are in doubt
use CBC as stream-cipher.
The following table summarizes some facts about the stream-ciphers in LEDA:

<table>
<thead>
<tr>
<th>mode</th>
<th>encryption</th>
<th>decryption</th>
<th>IV</th>
<th>security</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECB</td>
<td>$C_i = E_K(P_i)$</td>
<td>$P_i = D_K(C_i)$</td>
<td>none</td>
<td>−</td>
</tr>
<tr>
<td>CBC</td>
<td>$C_i = E_K(P_i \oplus C_{i-1})$</td>
<td>$P_i = D_K(C_i \oplus C_{i-1})$</td>
<td>unique</td>
<td>(+)</td>
</tr>
<tr>
<td>CFB</td>
<td>$C_i = P_i \oplus E_K(C_{i-1})$</td>
<td>$P_i = C_i \oplus E_K(C_{i-1})$</td>
<td>unique!</td>
<td>+</td>
</tr>
<tr>
<td>OFB</td>
<td>$C_i = P_i \oplus S_i$, $S_i = E_K(S_{i-1})$</td>
<td>$P_i = C_i \oplus S_i$</td>
<td>unique!</td>
<td>+</td>
</tr>
</tbody>
</table>

As stated above a block-cipher is needed to make the stream-ciphers work. The following
block-ciphers are part of LEDA:

<table>
<thead>
<tr>
<th>block-cipher</th>
<th>block size / bits</th>
<th>key-size / bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blowfish</td>
<td>64</td>
<td>32 – 448</td>
</tr>
<tr>
<td>Twofish</td>
<td>128</td>
<td>128 – 256</td>
</tr>
<tr>
<td>Rijndael (AES)</td>
<td>128</td>
<td>128 – 256</td>
</tr>
</tbody>
</table>

Comparing the ciphers in terms of performance Blowfish comes out worst (probably be-
cause of the smaller block size). Our implementation of Twofish is slightly faster than
that of Rijndael. In terms of security it is hard to rank the three ciphers. All of them have
been intensively cryptoanalyzed and no weaknesses have been found so far. Blowfish is
used in the ssh command of Unix, Rijndael won the contest for the Advanced Encryption
Standard (AES), and Twofish reached the final round of that contest.
11.4 Example for a Stream-Cipher (CBCCoder)

1. Definition

A stream-cipher is a coder that encrypts and decrypts data streams. In this section we discuss the interface of such a coder by an example: \texttt{CBCCoder< BlkCipher >}. Every stream-cipher in LEDA uses a block-cipher \texttt{BlkCipher} which is specified as a template parameter. (More information about stream-ciphers and block-ciphers can be found in Section 11.3.) The following stream-ciphers are available: \texttt{ECBCoder}, \texttt{CBCCoder}, \texttt{CFBCoder} and \texttt{OFBCoder}. All of them, except for \texttt{OFBCoder}, support fast seek operations (see Section 10.22). The available block-ciphers are: \texttt{Blowfish}, \texttt{Twofish} and \texttt{Rijndael}.

\#include <LEDA/coding/stream_ciphers.h>

2. Creation

\texttt{CBCCoder< BlkCipher > C(streambuf \ast src\_stream = 0, streambuf \ast tgt\_stream = 0, bool own\_streams = false);}  
creates an instance \texttt{C} which uses the given source and target streams. If \texttt{own\_streams} is set, then \texttt{C} is responsible for the destruction of the streams, otherwise the pointers \texttt{src\_stream} and \texttt{tgt\_stream} must be valid during the life-time of \texttt{C}.

\texttt{CBCCoder< BlkCipher > C(const char \ast src\_filename, const char \ast tgt\_filename);}  
creates an instance \texttt{C} which uses file-streams for input and output.

3. Operations

Standard Operations

\texttt{void C.encode()} encodes the source stream and writes the output to the target stream.

\texttt{void C.decode()} decodes the source stream and writes the output to the target stream.

\texttt{uint32 C.calculate\_length\_of\_encoded\_data(uint32 input\_length)}  
calculates the length (in bytes) of the output when encoding some input of the given length with the current settings. (This function is helpful for encoding memory chunks (see below).)
**11.4. EXAMPLE FOR A STREAM-CIPHER ( CBCCODER )**

```c
uint32 C.encode_memory_chunk(const char * in buf, uint32 in_len, char * out_buf, uint32 out_len)
```

encodes the memory chunk starting at `in_buf` with size `in_len` into the buffer starting at `out_buf` with size `out_len`. The function returns actual length of the encoded chunk which may be smaller than `out_len`. If the output buffer is too small for the encoded data the failure flag will be set (see below). **Note:** The output data is slightly longer than the input data due to padding and header data.

```c
uint32 C.decode_memory_chunk(const char * in buf, uint32 in_len, char * out_buf, uint32 out_len)
```

decodes a memory chunk. The meaning of the parameters and the return value is the same as in the previous function.

```c
streambuf* C.get_src_stream() returns the current source stream.
void C.set_src_stream(streambuf* src_stream, bool own_stream = false)
sets the source stream (cf. constructor).
void C.set_src_file(const char* file_name)
sets a file as source stream.
```

```c
streambuf* C.get_tgt_stream() returns the current target stream.
void C.set_tgt_stream(streambuf* tgt_stream, bool own_Stream = false)
sets the target stream (cf. constructor).
void C.set_tgt_file(const char* file_name)
sets a file as target stream.
void C.reset(bool keep_key = true)
puts C in the same state as the default constructor. If `keep_key` is false the key is set to the empty key. In any case the initialization vector is set to `nil`.
bool C.failed() returns true if an error occured.
bool C.finished() returns true if the coding is finished.
string C.get_description() provides a description for C.
```

**Additional Operations**

```c
uint16 C.get_default_key_size() returns the default key size.
```
CHAPTER 11. SYMMETRIC KEY CRYPTOGRAPHY

\texttt{uint16} \quad \texttt{C.get\_accepted\_key\_size(\texttt{uint16\& min}, \texttt{uint16\& max})}

provides the minimum and maximum key size (in bytes). The return value is the default key size.

\texttt{void} \quad \texttt{C.set\_key(const CryptKey& key)}

sets the key.

\texttt{bool} \quad \texttt{C.has\_key()}

queries whether a key has been set.

\texttt{CryptKey} \quad \texttt{C.get\_key()}

returns the current key.

\texttt{BlkCipher*} \quad \texttt{C.get\_block\_cipher()}

returns the underlying block-cipher.

\texttt{void} \quad \texttt{C.set\_initialization\_vector(const byte* iv)}

sets a user defined initialization vector (IV). The size of \texttt{iv} must be at least the block size of \texttt{BlkCipher}. (If no IV is provided by the user then \texttt{C} generates its own.)

\texttt{const byte*} \quad \texttt{C.get\_initialization\_vector()}

returns the user defined IV. (If no IV has been specified then \texttt{nil} is returned.)
11.5 Authentication (OMACCoder)

1. Definition

The class OMACCoder can be used for authentication. It implements the One-Key CBC MAC algorithm by Iwata and Kurosawa [49]. A MAC (= message authentication code) is a kind of checksum that is generated for a message \( m \). In contrast to a usual checksum a MAC does not only depend on \( m \) but also on a secret key \( k \). A MAC can be used if a sender wants to transfer \( m \) through an insecure channel to a receiver. If the sender also transmits the MAC then the receiver can verify that \( m \) has not been altered and that it originates from someone who knows \( k \).

**Important:** OMACCoder does not encipher or decipher any data, it only computes and verifies MACs!

The behaviour of this coder depends on the `MAC.in_stream` flag. If it is false then encoding and decoding are equivalent: Both methods copy the source stream to the target stream and compute a MAC. If the flag is true then in encoding mode the source stream is copied to the target stream, a MAC is computed and appended to the target stream. In decoding mode the MAC at the end of source stream is removed and the original data is copied to the target stream, a MAC is computed and compared with the MAC found in the source stream. If the two MACs differ an error is signaled. The method `MAC.is_valid` may be used to check whether the source stream is authentic.

The class OMACCoder<BlkCipher> is parameterized with a block-cipher BlkCipher, which can be one of the following: Rijndael (default), Blowfish, Twofish (see Section 11.3 for more information). OMACCoder also supports fast seek operations (see Section 10.22).

```cpp
#include "LEDA/coding/authentication.h"
```

2. Creation

OMACCoder<BlkCipher> \( C(\text{streambuf} \ast \text{src\_stream} = 0, \text{streambuf} \ast \text{tgt\_stream} = 0, \text{bool own\_streams} = \text{false}) \);

creates an instance \( C \) which uses the given source and target streams. If `own_streams` is set, then \( C \) is responsible for the destruction of the streams, otherwise the pointers `src_stream` and `tgt_stream` must be valid during the life-time of \( C \).

OMACCoder<BlkCipher> \( C(\text{const char} \ast \text{src\_file\_name}, \text{const char} \ast \text{tgt\_file\_name}) \);

creates an instance \( C \) which uses file-streams for input and output.
3. Operations

Standard Operations

```c
void C.encode() encodes the source stream and writes the output to the target stream.

void C.decode() decodes the source stream and writes the output to the target stream.

uint32 C.calculate_length_of_encoded_data(uint32 input_length)
calculates the length (in bytes) of the output when encoding some input of the given length with the current settings. (This function is helpful for encoding memory chunks (see below).)

uint32 C.encode_memory_chunk(const char *in_buf, uint32 in_len, char *out_buf, uint32 out_len)
encodes the memory chunk starting at in_buf with size in_len into the buffer starting at out_buf with size out_len. The function returns actual length of the encoded chunk which may be smaller than out_len. If the output buffer is too small for the encoded data the failure flag will be set (see below). Note: The output data is slightly longer than the input data due to padding and header data.

uint32 C.decode_memory_chunk(const char *in_buf, uint32 in_len, char *out_buf, uint32 out_len)
decodes a memory chunk. The meaning of the parameters and the return value is the same as in the previous function.
```

```c
streambuf * C.get_src_stream() returns the current source stream.

void C.set_src_stream(streambuf *src_stream, bool own_stream = false)
sets the source stream (cf. constructor).

void C.set_src_file(const char *file_name)
sets a file as source stream.

streambuf * C.get_tgt_stream() returns the current target stream.

void C.set_tgt_stream(streambuf *tgt_stream, bool own_Stream = false)
sets the target stream (cf. constructor).

void C.set_tgt_file(const char *file_name)
sets a file as target stream.
```
11.5. AUTHENTICATION (OMACCODER)

void C.reset(bool keep_parameters = true)

puts C in the same state as the default constructor. If keep_parameters is false the parameters are set to their default values and the key is set to the empty key.

bool C.failed()

returns true if an error occurred.

bool C.finished()

returns true if the coding is finished.

string C.get_description()

provides a description for C.

Additional Operations

uint16 C.get_default_key_size()

returns the default key size.

uint16 C.get_accepted_key_size(uint16& min, uint16& max)

provides the minimum and maximum key size (in bytes). The return value is the default key size.

void C.set_key(const CryptKey& key)

sets the key.

bool C.has_key()

queries whether a key has been set.

CryptKey C.get_key()

returns the current key.

const byte* C.check()

checks the source stream and returns the computed MAC. No target stream needs to be set. (If a target stream is set check coincides with decode).

bool C.MAC_is_valid()

returns whether C has a valid MAC.

string C.get_MAC_as_hex_string()

returns the MAC for the stream that has been processed last as hexadecimal string.

const byte* C.compute_MAC(string str)

computes a MAC for a string. (Sets source and target stream to nil.)

void C.set_MAC_in_stream_flag(bool mac_in_stream)

sets the MAC_in_stream flag.

bool C.get_MAC_in_stream_flag()

returns the MAC_in_stream flag.
11.6 Automatic Decoder supporting Cryptography (CryptAutoDecoder)

1. Definition

The class CryptAutoDecoder is an extension of AutoDecoder. It can be used for decoding encrypted streams (in particular if the coder used for encoding is unknown). In order to decrypt a stream a key must be provided by the user because the key is – of course – not stored in the encrypted stream. There are three possibilities to specify a key: as a CryptKey, as a key stream or as a key file. Sometimes it may be necessary to provide several keys to CryptAutoDecoder. E.g., if you use CoderPipe2<OMACCoder, OFBCoder> to encode a stream two keys are needed for decoding. Therefore, CryptAutoDecoder maintains a list of keys for decoding.

```
#include <LEDA/coding/crypt_auto_decoder.h>
```

2. Creation

CryptAutoDecoder C(streambuf * src_stream = 0, streambuf * tgt_stream = 0, bool own_streams = false);
creates an instance C which uses the given source and target streams. If own_streams is set, then C is responsible for the destruction of the streams, otherwise the pointers src_stream and tgt_stream must be valid during the life-time of C.

CryptAutoDecoder C(const char * src_file_name, const char * tgt_file_name);
creates an instance C which uses file-streams for input and output.

3. Operations

Standard Operations

`void` C.decode() decodes the source stream and writes the output to the target stream.

`uint32` C.decode_memory_chunk(const char * in_buf, uint32 in_len, char * out_buf, uint32 out_len) decodes the memory chunk starting at in_buf with size in_len into the buffer starting at out_buf with size out_len. The function returns actual length of the encoded chunk which may be smaller than out_len. If the output buffer is too small for the decoded data the failure flag will be set (see below).

`streambuf*` C.get_src_stream() returns the current source stream.
11.6. AUTOMATIC DECODER SUPPORTING CRYPTOGRAPHY (CRYPTAUTODECODER)

```c
void C.set_src_stream(streambuf * src_stream, bool own_stream = false)
sets the source stream (cf. constructor).
```

```c
void C.set_src_file(const char * file_name)
sets a file as source stream.
```

```c
streambuf* C.get_tgt_stream() returns the current target stream.
```

```c
void C.set_tgt_stream(streambuf * tgt_stream, bool own_Stream = false)
sets the target stream (cf. constructor).
```

```c
void C.set_tgt_file(const char * file_name)
sets a file as target stream.
```

```c
void C.reset() puts C in exactly the same state as the default constructor.
```

```c
bool C.failed() returns true if an error occurred.
```

```c
bool C.finished() returns true if decoding is finished.
```

```c
string C.get_description()
provides a description for C. After decoding this includes a description of the coder that has been used for encoding the stream.
```

**Additional Operations**

```c
coder_base* C.get_coder() after decoding this function returns the coder that has actually been used to decode the stream.
```

```c
bool C.has_key() returns whether the key list is not empty.
```

```c
list<CryptKey> C.get_keys() returns the current list of keys.
```

```c
void C.set_key(const CryptKey& key)
sets the key that should be used for decoding.
```

```c
void C.add_key(const CryptKey& key)
adds a key to the key list.
```

```c
void C.set_keys_in_stream(streambuf * key_stream)
makes the key list equal to the key(s) in key_stream.
```

```c
void C.add_keys_in_stream(streambuf * key_stream)
adds the key(s) in key_stream to the key list.
```

```c
void C.set_keys_in_file(const char * file_name)
makes the key list equal to the key(s) in the given file.
```
void C.add_keys_in_file(const char *file_name)
    adds the key(s) in the given file to the key list.

void C.clear_keys()    removes all keys.

4. Example

Example on how to use CryptAutoDecoder with a key file:

typedef OFBCoder<> Cipher;

    // generate key ...
    CryptByteString passphrase("secret phrase");
    CryptByteString dummy_salt; // will be ignored
    CryptKey key
        = CryptKey::generate_key_and_salt(16, 16, passphrase, dummy_salt, true);
        // 8*16 = 128 bit salt and key

    // write key
    ofstream key_out("secret.key");
    key_out << key;
    key_out.close();

    // encipher data
    encoding_ofstream<Cipher> data_out("data");
    data_out.get_coder()->set_key(key);
    data_out << "my secret text ..." << endl;
    data_out.close();

    // much later ...

    // decipher data
    decoding_ifstream<CryptAutoDecoder> data_in("data");
    data_in.get_coder()->set_keys_in_file("secret.key");
    string data; data.read_line(data_in); cout << data << endl;
    data_in.close();
11.7 Secure Socket Streambuffer (secure_socket_streambuf)

1. Definition

An instance $sb$ of class `secure_socket_streambuf` can be used as an adapter: It turns a `leda_socket` $s$ into a C++-streambuf object. This object can be used in standard C++ `ostreams` and `istreams`, which makes the communication through the socket easier. Moreover, `secure_socket_streambuf` uses cryptography to secure the communication. Every piece of data is authenticated, (possibly) compressed and encrypted before it is sent.

If two parties want to use the class `secure_socket_streambuf` to exchange data they have to do the following. First they establish a connection through `leda_sockets`. Then each party constructs an instance of the class `secure_socket_streambuf` which is attached to its socket. (An example showing how to this can be found at the end of Section 4.14; simply replace “`socket_streambuf`” by “`secure_socket_streambuf`” and add the passphrase(s).)

The communication between two instances of the class `secure_socket_streambuf` can be divided in two phases. In the first phase the two parties negotiate session parameters like packet sizes and they also agree on a so-called session-seed which will be explained later. In the second phase the actual user data is exchanged. Each phases is protected with cryptography. The authentication and encryption keys for the first phase are generated in a deterministic way from the user-supplied passphrase(s). They are called master keys because they remain the same as long as the passphrases are not changed. In order to protect the master keys we use them only during the first phase, which consists of two messages from each party. After that we use the random session seed (in addition to the passphrases) to compute session keys which are used during the second phase.

```c
#include <LEDA/coding/secure_socket_streambuf.h>
```

2. Creation

```c
secure_socket_streambuf $sb$ =
(leda_socket& $s$, const CryptByteString& $auth_passphrase$,
 const CryptByteString& $cipher_passphrase$,
 uint32 $out_buf_sz = DefaultBufferSize$,
 uint32 $in_buf_sz = DefaultBufferSize$,
 bool $enable_compression = false$,
 bool $send_acknowledge = false$);
```

creates a $sb$ and attaches it to the socket $s$. The two passphrases are used to generate the keys for authentication and encryption. The parameters $out_buf_sz$ and $in_buf_sz$ determine the maximum size of the out-buffer and the in-buffer. The parameter $enable_compression$ determines whether compression is used. $send_acknowledge$ specifies whether an acknowledgement is sent for every received packet. **Precondition:** The connection between the server and the client must have been established when $sb$ is created.
secure_socket_streambuf sb(leda_socket& s, const CryptByteString& passphrase,
    uint32 out_buf_sz = DefaultBufferSize,
    uint32 in_buf_sz = DefaultBufferSize,
    bool enable_compression = false,
    bool send_acknowledge = false);

as above, but the same passphrase is used for generating both the authentication and the encryption key. However, it is still guaranteed that two different keys are used.

3. Operations

The class secure_socket_streambuf inherits most of its operations from the class streambuf that belongs to the C++ standard library. Usually there is no need to call these operations explicitly. (You can find documentation for streambuf at http://www.cplusplus.com)

bool sb.failed() returns whether a failure has occurred.

string sb.get_error() returns an error message (if available).

void sb.sputEOF() signals the end of the transmission to the receiving socket, so that it does not wait for further data. (This function is called automatically in the destructor unless it has been called explicitly by the user. If sb is not immediately destroyed after the end of the transmission then you should call sputEOF explicitly, otherwise the receiving party might incur a timeout error.)

bool sb.has_put_EOF() returns whether EOF has already been sent.

bool sb.has_get_EOF() returns whether EOF has already been received.

leda_socket& sb.get_socket() returns the socket to which sb is attached.

uint32 sb.get_outgoing_packet_size() returns the (actual) outgoing packet size.

uint32 sb.get_incoming_packet_size() returns the (actual) incoming packet size.

bool sb.uses_compression() returns whether sb compresses outgoing packets before sending.

bool sb.waits_for_acknowledge() returns whether sb expects an acknowledgement for outgoing packets.
bool sb.sends_acks

returns whether `sb` sends an acknowledgement for incoming packets.
Chapter 12

Graphs and Related Data Types

12.1 Graphs (graph)

1. Definition

An instance $G$ of the data type graph consists of a list $V$ of nodes and a list $E$ of edges (node and edge are item types). Distinct graph have disjoint node and edge lists. The value of a variable of type node is either the node of some graph, or the special value nil (which is distinct from all nodes), or is undefined (before the first assignment to the variable). A corresponding statement is true for the variables of type edge.

A graph with empty node list is called empty. A pair of nodes $(v, w) \in V \times V$ is associated with every edge $e \in E$; $v$ is called the source of $e$ and $w$ is called the target of $e$, and $v$ and $w$ are called endpoints of $e$. The edge $e$ is said to be incident to its endpoints.

A graph is either directed or undirected. The difference between directed and undirected graph is the way the edges incident to a node are stored and how the concept adjacent is defined.

In directed graph two lists of edges are associated with every node $v$: $\text{adj\_edges}(v) = \{e \in E \mid v = \text{source}(e)\}$, i.e., the list of edges starting in $v$, and $\text{in\_edges}(v) = \{e \in E \mid v = \text{target}(e)\}$, i.e., the list of edges ending in $v$. The list $\text{adj\_edges}(v)$ is called the adjacency list of node $v$ and the edges in $\text{adj\_edges}(v)$ are called the edges adjacent to node $v$. For directed graph we often use $\text{out\_edges}(v)$ as a synonym for $\text{adj\_edges}(v)$.

In undirected graph only the list $\text{adj\_edges}(v)$ is defined for every every node $v$. Here it contains all edges incident to $v$, i.e., $\text{adj\_edges}(v) = \{e \in E \mid v \in \{\text{source}(e), \text{target}(e)\}\}$. An undirected graph may not contain selfloops, i.e., it may not contain an edge whose source is equal to its target.

In a directed graph an edge is adjacent to its source and in an undirected graph it is adjacent to its source and target. In a directed graph a node $w$ is adjacent to a node $v$ if
there is an edge \((v, w) \in E\); in an undirected graph \(w\) is adjacent to \(v\) if there is an edge \((v, w)\) or \((w, v)\) in the graph.

A directed graph can be made undirected and vice versa: \(G\).make_undirected() makes the directed graph \(G\) undirected by appending for each node \(v\) the list \(\text{in}\_edges(v)\) to the list \(\text{adj}\_edges(v)\) (removing selfloops). Conversely, \(G\).make_directed() makes the undirected graph \(G\) directed by splitting for each node \(v\) the list \(\text{adj}\_edges(v)\) into the lists \(\text{out}\_edges(v)\) and \(\text{in}\_edges(v)\). Note that these two operations are not exactly inverse to each other. The data type \text{ugraph}\ (\text{cf. section 12.4}) can only represent undirected graph.

**Reversal Information, Maps and Faces**

The reversal information of an edge \(e\) is accessed through \(G\).reversal(e), it has type \text{edge} and may or may not be defined (= \text{nil}). Assume that \(G\).reversal(e) is defined and let \(e' = G\).reversal(e). Then \(e = (v, w)\) and \(e' = (w, v)\) for some nodes \(v\) and \(w\), \(G\).reversal(e') is defined and \(e = G\).reversal(e'). In addition, \(e \neq e'\). In other words, \text{reversal} deserves its name.

We call a directed graph \text{bidirected} if the reversal information can be properly defined for all edges in \(G\), resp. if there exists a bijective function \(\text{rev} : E \to E\) with the properties of \text{reversal} as described above and we call a bidirected graph a \text{map} if all edges have their reversal information defined. Maps are the data structure of choice for embedded graph. For an edge \(e\) of a map \(G\) let \(\text{face}\_cycle\_succ(e) = \text{cyclic}\_\text{adj}\_\text{pred}(\text{reversal}(e))\) and consider the sequence \(e, \text{face}\_cycle\_succ(e), \text{face}\_cycle\_succ(\text{face}\_cycle\_succ(e)), \ldots\). The first edge to repeat in this sequence is \(e\) (why?) and the set of edges appearing in this sequence is called the \text{face cycle} containing \(e\). Each edge is contained in some face cycle and face cycles are pairwise disjoint. Let \(f\) be the number of face cycles, \(n\) be the number of (non-isolated) nodes, \(m\) be the number of edges, and let \(c\) be the number of (non-singleton) connected components. Then \(g = (m/2 - n - f)/2 + c\) is called the \text{genus} of the map [89] (note that \(m/2\) is the number of edges in the underlying undirected graph). The genus is zero if and only if the map is planar, i.e., there is an embedding of \(G\) into the plane such that for every node \(v\) the counter-clockwise ordering of the edges around \(v\) agrees with the cyclic ordering of \(v\)'s adjacency list. (In order to check whether a map is planar, you may use the function \text{IsPlaneMap}( ) in 12.23.)

If a graph \(G\) is a map the faces of \(G\) can be constructed explicitly by \(G\).compute_faces( ). Afterwards, the faces of \(G\) can be traversed by different iterators, e.g., \textit{forall}\_faces\((f, G)\) iterates over all faces, \textit{forall}\_\text{adj}\_faces\((v)\) iterates over all faces adjacent to node \(v\). By using face maps or arrays (data types \text{face}\_map and \text{face}\_array) additional information can be associated with the faces of a graph. Note that any update operation performed on \(G\) invalidates the list of faces. See the section on face operations for a complete list of available operations for faces.

```cpp
#include <LEDA/graph/graph.h>
```
2. Creation

```
graph G;  // creates an object G of type graph and initializes it to the empty directed graph.
```

```
graph G(int n_slots, int e_slots);  // this constructor specifies the numbers of free data slots in the nodes and edges of G that can be used for storing the entries of node and edge arrays. See also the description of the use_node_data( ) and use_edge_data( ) operations in 12.8 and 12.9.
```

3. Operations

```
void G.init(int n, int m)  // this operation has to be called for semi-dynamic graph (if compiled with -DGRAPH_REP = 2) immediately after the constructor to specify upper bounds n and m for the number of nodes and edges respectively. This operation has no effect if called for the (fully-dynamic) standard graph representation.
```

a) Access operations

```
int G.outdeg(node v)  // returns the number of edges adjacent to node v (|adj_edges(v)|).

int G.indeg(node v)  // returns the number of edges ending at v (|in_edges(v)|) if G is directed and zero if G is undirected).

int G.degree(node v)  // returns outdeg(v) + indeg(v).

node G.source(edge e)  // returns the source node of edge e.

node G.target(edge e)  // returns the target node of edge e.

node G.opposite(node v, edge e)  // returns target(e) if v = source(e) and source(e) otherwise.

node G.opposite(edge e, node v)  // same as above.

int G.number_of_nodes()  // returns the number of nodes in G.

int G.number_of_edges()  // returns the number of edges in G.

const list<node>& G.all_nodes()  // returns the list V of all nodes of G.

const list<edge>& G.all_edges()  // returns the list E of all edges of G.
```
\textit{list<edge>} G.\texttt{adj}\_\texttt{edges}(\texttt{node} v) \quad \textit{returns} \textit{adj\_edges}(v).

\textit{list<edge>} G.\texttt{out}\_\texttt{edges}(\texttt{node} v) \quad \textit{returns} \textit{adj\_edges}(v) \textit{if} \textit{G} \textit{is directed} \textit{and} \textit{the empty list otherwise.}

\textit{list<edge>} G.\texttt{in}\_\texttt{edges}(\texttt{node} v) \quad \textit{returns} \textit{in\_edges}(v) \textit{if} \textit{G} \textit{is directed} \textit{and} \textit{the empty list otherwise.}

\textit{list<node>} G.\texttt{adj}\_\texttt{nodes}(\texttt{node} v) \quad \textit{returns} \textit{the list of all nodes adjacent to} \textit{v}.

\textit{node} G.\texttt{first}\_\texttt{node}( ) \quad \textit{returns} \textit{the first node in} \textit{V}.

\textit{node} G.\texttt{last}\_\texttt{node}( ) \quad \textit{returns} \textit{the last node in} \textit{V}.

\textit{node} G.\texttt{choose}\_\texttt{node}( ) \quad \textit{returns} \textit{a random node of} \textit{G} (\textit{nil} \textit{if} \textit{G} \textit{is empty}).

\textit{node} G.\texttt{succ}\_\texttt{node}(\texttt{node} v) \quad \textit{returns} \textit{the successor of} \textit{node} \textit{v} \textit{in} \textit{V} (\textit{nil} \textit{if} \textit{it does not exist}).

\textit{node} G.\texttt{pred}\_\texttt{node}(\texttt{node} v) \quad \textit{returns} \textit{the predecessor of} \textit{node} \textit{v} \textit{in} \textit{V} (\textit{nil} \textit{if} \textit{it does not exist}).

\textit{edge} G.\texttt{first}\_\texttt{edge}( ) \quad \textit{returns} \textit{the first edge in} \textit{E}.

\textit{edge} G.\texttt{last}\_\texttt{edge}( ) \quad \textit{returns} \textit{the last edge in} \textit{E}.

\textit{edge} G.\texttt{choose}\_\texttt{edge}( ) \quad \textit{returns} \textit{a random edge of} \textit{G} (\textit{nil} \textit{if} \textit{G} \textit{is empty}).

\textit{edge} G.\texttt{succ}\_\texttt{edge}(\texttt{edge} e) \quad \textit{returns} \textit{the successor of} \textit{edge} \textit{e} \textit{in} \textit{E} (\textit{nil} \textit{if} \textit{it does not exist}).

\textit{edge} G.\texttt{pred}\_\texttt{edge}(\texttt{edge} e) \quad \textit{returns} \textit{the predecessor of} \textit{edge} \textit{e} \textit{in} \textit{E} (\textit{nil} \textit{if} \textit{it does not exist}).

\textit{edge} G.\texttt{first}\_\texttt{adj}\_\texttt{edge}(\texttt{node} v) \quad \textit{returns} \textit{the first edge in the adjacency list of} \textit{v} (\textit{nil} \textit{if} \textit{this list is empty}).

\textit{edge} G.\texttt{last}\_\texttt{adj}\_\texttt{edge}(\texttt{node} v) \quad \textit{returns} \textit{the last edge in the adjacency list of} \textit{v} (\textit{nil} \textit{if} \textit{this list is empty}).

\textit{edge} G.\texttt{adj}\_\texttt{succ}(\texttt{edge} e) \quad \textit{returns} \textit{the successor of} \textit{edge} \textit{e} \textit{in the adjacency list of node} \textit{source}(e) (\textit{nil} \textit{if} \textit{it does not exist}).

\textit{edge} G.\texttt{adj}\_\texttt{pred}(\texttt{edge} e) \quad \textit{returns} \textit{the predecessor of} \textit{edge} \textit{e} \textit{in the adjacency list of node} \textit{source}(e) (\textit{nil} \textit{if} \textit{it does not exist}).

\textit{edge} G.\texttt{cyclic}\_\texttt{adj}\_\texttt{succ}(\texttt{edge} e) \quad \textit{returns} \textit{the cyclic successor of} \textit{edge} \textit{e} \textit{in the adjacency list of node} \textit{source}(e).

\textit{edge} G.\texttt{cyclic}\_\texttt{adj}\_\texttt{pred}(\texttt{edge} e) \quad \textit{returns} \textit{the cyclic predecessor of} \textit{edge} \textit{e} \textit{in the adjacency list of node} \textit{source}(e).
12.1. GRAPHS (GRAPH)

\[\text{edge } G.\text{first in edge(node } v) \text{ returns the first edge of } \text{in}_\text{edges}(v) \text{ (nil if this list is empty).}\]

\[\text{edge } G.\text{last in edge(node } v) \text{ returns the last edge of } \text{in}_\text{edges}(v) \text{ (nil if this list is empty).}\]

\[\text{edge } G.\text{in succ(edge } e) \text{ returns the successor of edge } e \text{ in } \text{in}_\text{edges}(\text{target}(e)) \text{ (nil if it does not exist).}\]

\[\text{edge } G.\text{in pred(edge } e) \text{ returns the predecessor of edge } e \text{ in } \text{in}_\text{edges}(\text{target}(e)) \text{ (nil if it does not exist).}\]

\[\text{edge } G.\text{cyclic in succ(edge } e) \text{ returns the cyclic successor of edge } e \text{ in } \text{in}_\text{edges}(\text{target}(e)) \text{ (nil if it does not exist).}\]

\[\text{edge } G.\text{cyclic in pred(edge } e) \text{ returns the cyclic predecessor of edge } e \text{ in } \text{in}_\text{edges}(\text{target}(e)) \text{ (nil if it does not exist).}\]

\[\text{bool } G.\text{is directed( ) returns true iff } G \text{ is directed.}\]

\[\text{bool } G.\text{is undirected( ) returns true iff } G \text{ is undirected.}\]

\[\text{bool } G.\text{empty( ) returns true iff } G \text{ is empty.}\]

b) Update operations

\[\text{node } G.\text{new node( ) adds a new node to } G \text{ and returns it.}\]

\[\text{node } G.\text{new node(node } u, \text{ int dir) adds a new node } v \text{ to } G \text{ and returns it. } v \text{ is inserted in front of } (\text{dir } = \text{leda:: before}) \text{ or behind } (\text{dir } = \text{leda:: behind}) \text{ node } u \text{ into the list of all nodes.}\]

\[\text{edge } G.\text{new edge(node } v, \text{ node } w) \text{ adds a new edge } (v,w) \text{ to } G \text{ by appending it to } \text{adj}_\text{edges}(v) \text{ and to } \text{in}_\text{edges}(w) \text{ (if } G \text{ is directed) or } \text{adj}_\text{edges}(w) \text{ (if } G \text{ is undirected), and returns it.}\]

\[\text{edge } G.\text{new edge(edge } e, \text{ node } w, \text{ int dir } = \text{leda:: behind}) \text{ adds a new edge } x = (\text{source}(e),w) \text{ to } G. \text{ x is inserted in front of } (\text{dir } = \text{leda:: before}) \text{ or behind } (\text{dir } = \text{leda:: behind}) \text{ edge } e \text{ into } \text{adj}_\text{edges(source}(e)) \text{ and appended to } \text{in}_\text{edges}(w) \text{ (if } G \text{ is directed) or } \text{adj}_\text{edges}(w) \text{ (if } G \text{ is undirected). Here } \text{leda:: before} \text{ and } \text{leda:: behind} \text{ are pre-defined constants. The operation returns the new edge } x.\]

\[\text{Precondition: source}(e) \neq w \text{ if } G \text{ is undirected.}\]
add a new edge \( x = (v, \text{target}(e)) \) to \( G \). \( x \) is appended to \( \text{adj
dges}(v) \) and inserted in front of (\( \text{dir} = \text{leda::before} \)) or behind (\( \text{dir} = \text{leda::behind} \)) edge \( e \) into \( \text{in_edges}(:,:,target(e)) \) (if \( G \) is directed) or \( \text{adj
dges}(target(e)) \) (if \( G \) is undirected). The operation returns the new edge \( x \).

**Precondition:** \( \text{target}(e) \neq v \) if \( G \) is undirected.

add a new edge \( x = (\text{source}(e1), \text{target}(e2)) \) to \( G \). \( x \) is inserted in front of (if \( d1 = \text{leda::before} \)) or behind (if \( d1 = \text{leda::behind} \)) edge \( e1 \) into \( \text{adj
dges}(\text{source}(e1)) \) and in front of (if \( d2 = \text{leda::before} \)) or behind (if \( d2 = \text{leda::behind} \)) edge \( e2 \) into \( \text{in_edges}(:,:,target(e2)) \) (if \( G \) is directed) or \( \text{adj
dges}(target(e2)) \) (if \( G \) is undirected). The operation returns the new edge \( x \).

removes edge \( e \) temporarily from \( G \) until restored by \( G\text{-restore-edge}(e) \).

hides all edges in \( el \).

returns \( \text{true} \) if \( e \) is hidden and \( \text{false} \) otherwise.

returns the list of all hidden edges of \( G \).

restores \( e \) by appending it to \( \text{adj
dges}(\text{source}(e)) \) and to \( \text{in_edges}(:,:,target(e)) \) (\( \text{adj
dges}(target(e)) \) if \( G \) is undirected). **Precondition:** \( e \) is hidden and ne-
ither \( \text{source}(e) \) nor \( \text{target}(e) \) is hidden.

restores all edges in \( el \).

restores all hidden edges.
12.1. GRAPHS (GRAPH)

void G.hide_node(node v) removes node v temporarily from G until restored by G.restore_node(v). All non-hidden edges in adj_edges(v) and in_edges(v) are hidden too.

void G.hide_node(node v, list<edge>& h_edges) as above, in addition, the list of leaving or entering edges which are hidden as a result of hiding v are appended to h_edges.

bool G.is_hidden(node v) returns true if v is hidden and false otherwise.

list<node> G.hidden_nodes() returns the list of all hidden nodes of G.

void G.restore_node(node v) restores v by appending it to the list of all nodes. Note that no edge adjacent to v that was hidden by G.hide_node(v) is restored by this operation.

void G.restore_all_nodes() restores all hidden nodes.

void G.del_node(node v) deletes v and all edges incident to v from G.

void G.del_edge(edge e) deletes the edge e from G.

void G.del_nodes(const list<node>& L) deletes all nodes in L from G.

void G.del_edges(const list<edge>& L) deletes all edges in L from G.

void G.del_all_nodes() deletes all nodes from G.

void G.del_all_edges() deletes all edges from G.

void G.del_all_faces() deletes all faces from G.

void G.move_edge(edge e, node v, node w) moves edge e to source v and target w by appending it to adj_edges(v) and to in_edges(w) (if G is directed) or adj_edges(w) (if G is undirected).

void G.move_edge(edge e, edge e1, node w, int d = leda::behind) moves edge e to source source(e1) and target w by inserting it in front of (if d = leda::before) or behind (if d = leda::behind) edge e1 into adj_edges(source(e1)) and by appending it to in_edges(w) (if G is directed) or adj_edges(w) (if G is undirected).
void G.move_edge(edge e, node v, edge e2, int d = leda::behind)

moves edge $e$ to source $v$ and target $\text{target}(e2)$ by appending it to $\text{adj_edges}(v)$ and inserting it in front of (if $d = \text{leda::before}$) or behind (if $d = \text{leda::behind}$) edge $e2$ into $\text{in_edges}(\text{target}(e2))$ (if $G$ is directed) or $\text{adj_edges}(\text{target}(e2))$ (if $G$ is undirected).

void G.move_edge(edge e, edge e1, edge e2, int d1 = leda::behind, int d2 = leda::behind)

moves edge $e$ to source $\text{source}(e1)$ and target $\text{target}(e2)$ by inserting it in front of (if $d1 = \text{leda::before}$) or behind (if $d1 = \text{leda::behind}$) edge $e1$ into $\text{adj_edges}(\text{source}(e1))$ and in front of (if $d2 = \text{leda::before}$) or behind (if $d2 = \text{leda::behind}$) edge $e2$ into $\text{in_edges}(\text{target}(e2))$ (if $G$ is directed) or $\text{adj_edges}(\text{target}(e2))$ (if $G$ is undirected).

dedge G.rev_edge(edge e) reverses $e$ ($\text{move_edge}(e, \text{target}(e), \text{source}(e))$).

void G.rev_all_edges() reverses all edges of $G$.

void G.sort_nodes(int (*cmp)(const node&, const node&))

the nodes of $G$ are sorted according to the ordering defined by the comparing function $\text{cmp}$. Subsequent executions of forall_nodes step through the nodes in this order. (cf. TOPSORT1 in section 13).

void G.sort_edges(int (*cmp)(const edge&, const edge&))

the edges of $G$ and all adjacency lists are sorted according to the ordering defined by the comparing function $\text{cmp}$. Subsequent executions of forall_edges step through the edges in this order. (cf. TOPSORT1 in section 13).

void G.sort_nodes(const node_array<T>& A)

the nodes of $G$ are sorted according to the entries of node_array $A$ (cf. section 12.8).

Precondition: $T$ must be numerical, i.e., number type int, float, double, integer, rational or real.

void G.sort_edges(const edge_array<T>& A)

the edges of $G$ are sorted according to the entries of edge_array $A$ (cf. section 12.9).

Precondition: $T$ must be numerical, i.e., number type int, float, double, integer, rational or real.
void G.bucket_sort_nodes(int l, int h, int (*ord)(const node&))
sorts the nodes of G using bucket sort
Precondition: \(l \leq \text{ord}(v) \leq h\) for all nodes \(v\).

void G.bucket_sort_edges(int l, int h, int (*ord)(const edge&))
sorts the edges of G using bucket sort
Precondition: \(l \leq \text{ord}(e) \leq h\) for all edges \(e\).

void G.bucket_sort_nodes(int (*ord)(const node&))
same as G.bucket_sort_nodes(l, h, ord) with \(l(h)\) equal to the minimal (maximal) value of ord(v).

void G.bucket_sort_edges(int (*ord)(const edge&))
same as G.bucket_sort_edges(l, h, ord) with \(l(h)\) equal to the minimal (maximal) value of ord(e).

void G.bucket_sort_nodes(const node_array<int>& A)
same as G.bucket_sort_nodes(ord) with ord(v) = A[v] for all nodes \(v\) of G.

void G.bucket_sort_edges(const edge_array<int>& A)
same as G.bucket_sort_edges(ord) with ord(e) = A[e] for all edges \(e\) of G.

void G.set_node_position(node v, node p)
moves node \(v\) in the list \(V\) of all nodes such that \(p\) becomes the predecessor of \(v\). If \(p = \text{nil}\) then \(v\) is moved to the front of \(V\).

void G.set_edge_position(edge e, edge p)
moves edge \(e\) in the list \(E\) of all edges such that \(p\) becomes the predecessor of \(e\). If \(p = \text{nil}\) then \(e\) is moved to the front of \(E\).

void G.permute_edges()
the edges of G and all adjacency lists are randomly permuted.

list<edge> G.insert_reverse_edges()
for every edge \((v, w)\) in G the reverse edge \((w, v)\) is inserted into G. Returns the list of all inserted edges.
Remark: the reversal information is not set by this function.

void G.make_undirected()
makes G undirected by appending in_edges(v) to adj_edges(v) for all nodes \(v\).

void G.make_directed()
makes G directed by splitting adj_edges(v) into out_edges(v) and in_edges(v).

void G.clear()
makes G the empty graph.
void G.join(graph& H) merges H into G by moving all objects (nodes, edges, and faces) from H to G. H is empty afterwards.

c) Reversal Edges and Maps

void G.make_bidirected() makes G bidirected by inserting missing reversal edges.

void G.make_bidirected(list<edge>& R) makes G bidirected by inserting missing reversal edges. Appends all inserted edges to list R.

bool G.is_bidirected() returns true if every edge has a reversal and false otherwise.

bool G.make_map() sets the reversal information of a maximal number of edges of G. Returns true if G is bidirected and false otherwise.

void G.make_map(list<edge>& R) makes G bidirected by inserting missing reversal edges and then turns it into a map setting the reversals for all edges. Appends all inserted edges to list R.

bool G.is_map() tests whether G is a map.

edge G.reversal(edge e) returns the reversal information of edge e (nil if not defined).

void G.set_reversal(edge e, edge r) makes r the reversal of e and vice versa. If the reversal information of e was defined prior to the operation, say as e’, the reversal information of e’ is set to nil. The same holds for r.
Precondition: e = (v, w) and r = (w, v) for some nodes v and w.

edge G.face_cycle_succ(edge e) returns the cyclic adjacency predecessor of reversal(e).
Precondition: reversal(e) is defined.

edge G.face_cycle_pred(edge e) returns the reversal of the cyclic adjacency successor s of e.
Precondition: reversal(s) is defined.

edge G.split_map_edge(edge e) splits edge e = (v, w) and its reversal r = (w, v) into edges (v, u), (u, w), (w, u), and (u, v). Returns the edge (u, w).
12.1. GRAPHS ( GRAPH )

**edge** $G$.new_map.edge(edge e1, edge e2)

inserts a new edge $e = (source(e1), source(e2))$ after $e1$ into the adjacency list of $source(e1)$ and an edge $r$ reversal to $e$ after $e2$ into the adjacency list of $source(e2)$.

**list<edge>** $G$.triangulate_map()

triangulates the map $G$ by inserting additional edges. The list of inserted edges is returned.

*Precondition:* $G$ must be connected.

The algorithm ([47]) has running time $O(|V| + |E|)$.

**void** $G$.dual_map(graph& D)

constructs the dual of $G$ in $D$. The algorithm has linear running time.

*Precondition:* $G$ must be a map.

**For backward compatibility**

**edge** $G$.reverse(edge e)

returns reversal$(e)$ (historical).

**edge** $G$.succ_face_edge(edge e)

returns face_cycle_succ$(e)$ (historical).

**edge** $G$.next_face_edge(edge e)

returns face_cycle_succ$(e)$ (historical).

**edge** $G$.pred_face_edge(edge e)

returns face_cycle_pred$(e)$ (historical).

d) Faces and Planar Maps

**void** $G$.compute_faces()

constructs the list of face cycles of $G$.

*Precondition:* $G$ is a map.

**face** $G$.face_of(edge e)

returns the face of $G$ to the left of edge $e$.

**face** $G$.adj_face(edge e)

returns $G$.face_of$(e)$.

**void** $G$.print_face(face f)

prints face $f$.

**int** $G$.number_of_faces()

returns the number of faces of $G$.

**face** $G$.first_face()

returns the first face of $G$.

(nil if empty).

**face** $G$.last_face()

returns the last face of $G$.

**face** $G$.choose_face()

returns a random face of $G$ (nil if $G$ is empty).

**face** $G$.succ_face(face f)

returns the successor of face $f$ in the face list of $G$

(nil if it does not exist).

**face** $G$.pred_face(face f)

returns the predecessor of face $f$ in the face list of $G$

(nil if it does not exist).
const list<face>& G.all_faces() returns the list of all faces of G.

list<face> G.adj_faces(node v) returns the list of all faces of G adjacent to node v in counter-clockwise order.

list<node> G.adj_nodes(face f) returns the list of all nodes of G adjacent to face f in counter-clockwise order.

list<edge> G.adj_edges(face) returns the list of all edges of G bounding face f in counter-clockwise order.

int G.size(face f) returns the number of edges bounding face f.

edge G.first_face_edge(face f) returns the first edge of face f in G.

edge G.split_face(edge e1, edge e2) inserts the edge e = (source(e1), source(e2)) and its reversal into G and returns e.  
Precondition: e1 and e2 are bounding the same face F.  
The operation splits F into two new faces.

face G.join_faces(edge e) deletes edge e and its reversal r and updates the list of faces accordingly. The function returns a face that is affected by the operations (see the LEDA book for details).

void G.make_planar_map() makes G a planar map by reordering the edges such that for every node v the ordering of the edges in the adjacency list of v corresponds to the counter-clockwise ordering of these edges around v for some planar embedding of G and constructs the list of faces.  
Precondition: G is a planar bidirected graph (map).

list<edge> G.triangulate_planar_map() triangulates planar map G and recomputes its list of faces.

e) Operations for undirected graph
edge \text{G.new\_edge(node } v, \text{ edge } e_1, \text{ node } w, \text{ edge } e_2, \text{ int } d_1 = \text{leda::behind, int } d_2 = \text{leda::behind)}

adds a new edge \((v, w)\) to \(G\) by inserting it in front of (if \(d_1 = \text{leda::before}\)) or behind (if \(d_1 = \text{leda::behind}\)) edge \(e_1\) into \(\text{adj\_edges}(v)\) and in front of (if \(d_2 = \text{leda::before}\)) or behind (if \(d_2 = \text{leda::behind}\)) edge \(e_2\) into \(\text{adj\_edges}(w)\), and returns it.

\textit{Precondition:} \(e_1\) is incident to \(v\) and \(e_2\) is incident to \(w\) and \(v \neq w\).

edge \text{G.new\_edge(node } v, \text{ edge } e, \text{ node } w, \text{ int } d = \text{leda::behind)}

adds a new edge \((v, w)\) to \(G\) by inserting it in front of (if \(d = \text{leda::before}\)) or behind (if \(d = \text{leda::behind}\)) edge \(e\) into \(\text{adj\_edges}(v)\) and appending it to \(\text{adj\_edges}(w)\), and returns it.

\textit{Precondition:} \(e\) is incident to \(v\) and \(v \neq w\).

edge \text{G.new\_edge(node } v, \text{ node } w, \text{ edge } e, \text{ int } d = \text{leda::behind)}

adds a new edge \((v, w)\) to \(G\) by appending it to \(\text{adj\_edges}(v)\), and by inserting it in front of (if \(d = \text{leda::before}\)) or behind (if \(d = \text{leda::behind}\)) edge \(e\) into \(\text{adj\_edges}(w)\), and returns it.

\textit{Precondition:} \(e\) is incident to \(w\) and \(v \neq w\).

\textsc{f) I/O Operations}

\textit{edge G.adj\_succ(edge } e, \text{ node } v)}

returns the successor of edge \(e\) in the adjacency list of \(v\).

\textit{Precondition:} \(e\) is incident to \(v\).

\textit{edge G.adj\_pred(edge } e, \text{ node } v)}

returns the predecessor of edge \(e\) in the adjacency list of \(v\).

\textit{Precondition:} \(e\) is incident to \(v\).

\textit{edge G.cyclic\_adj\_succ(edge } e, \text{ node } v)}

returns the cyclic successor of edge \(e\) in the adjacency list of \(v\).

\textit{Precondition:} \(e\) is incident to \(v\).

\textit{edge G.cyclic\_adj\_pred(edge } e, \text{ node } v)}

returns the cyclic predecessor of edge \(e\) in the adjacency list of \(v\).

\textit{Precondition:} \(e\) is incident to \(v\).
void $G$.write(ostream& $O$ = cout)
writes $G$ to the output stream $O$.

void $G$.write(string $s$)
writes $G$ to the file with name $s$.

int $G$.read(istream& $I$ = cin)
reads a graph from the input stream $I$ and assigns it to $G$.

int $G$.read(string $s$)
reads a graph from the file with name $s$ and assigns it to $G$. Returns 1 if file $s$ does not exist, 2 if the edge and node parameter types of *this and the graph in the file $s$ do not match, 3 if file $s$ does not contain a graph, and 0 otherwise.

bool $G$.write_gml(ostream& $O$ = cout, void (*node_cb)(ostream&, const graph*, const node), void (*edge_cb)(ostream&, const graph*, const edge), 0)
writes $G$ to the output stream $O$ in GML format ([46]). If node_cb is not equal to 0, it is called while writing a node $v$ with output stream $O$, the graph and $v$ as parameters. It can be used to write additional user defined node data. The output should conform with GML format (see manual page gml_graph). edge_cb is called while writing edges. If the operation fails, false is returned.

bool $G$.write_gml(string $s$, void (*node_cb)(ostream&, const graph*, const node), void (*edge_cb)(ostream&, const graph*, const edge), 0)
writes $G$ to the file with name $s$ in GML format. For a description of node_cb and edge_cb, see above. If the operation fails, false is returned.

bool $G$.read_gml(string $s$)
reads a graph in GML format from the file with name $s$ and assigns it to $G$.

bool $G$.read_gml(istream& $I$ = cin)
reads a graph in GML format from the input stream $I$ and assigns it to $G$.

void $G$.print_node(node $v$, ostream& $O$ = cout)
prints node $v$ on the output stream $O$.

void $G$.print_edge(edge $e$, ostream& $O$ = cout)
prints edge $e$ on the output stream $O$. If $G$ is directed $e$ is represented by an arrow pointing from source to target. If $G$ is undirected $e$ is printed as an undirected line segment.
void G.print(string s, ostream& O = cout)  
    prints G with header line s on the output stream O.
void G.print(ostream& O = cout)  
    prints G on the output stream O.

**g) Non-Member Functions**

*node* source(edge e)  
    returns the source node of edge e.
*node* target(edge e)  
    returns the target node of edge e.
*graph* graph_of(node v)  
    returns a pointer to the graph that v belongs to.
*graph* graph_of(edge e)  
    returns a pointer to the graph that e belongs to.
*graph* graph_of(face f)  
    returns a pointer to the graph that f belongs to.
*face* face_of(edge e)  
    returns the face of edge e.

**h) Iteration**

All iteration macros listed in this section traverse the corresponding node and edge lists of the graph, i.e. they visit nodes and edges in the order in which they are stored in these lists.

forall_nodes(v, G)  
{ “the nodes of G are successively assigned to v” }
forall_edges(e, G)  
{ “the edges of G are successively assigned to e” }
forall_rev_nodes(v, G)  
{ “the nodes of G are successively assigned to v in reverse order” }
forall_rev_edges(e, G)  
{ “the edges of G are successively assigned to e in reverse order” }
forall_adj_edges(e, w)  
{ “the edges adjacent to node w are successively assigned to e” }
forall_out_edges(e, w) a faster version of forall_adj_edges for directed graph.
forall_in_edges(e, w)  
{ “the edges of in_edges(w) are successively assigned to e” }
forall_inout_edges(e, w)  
{ “the edges of out_edges(w) and in_edges(w) are successively assigned to e” }
forall adj_nodes(v, w)
{ “the nodes adjacent to node w are successively assigned to v” }

Faces

Before using any of the following face iterators the list of faces has to be computed by calling G.compute_faces(). Note, that any update operation invalidates this list.

forall_faces(f, M)
{ “the faces of M are successively assigned to f” }

forall_face_edges(e, f)
{ “the edges of face f are successively assigned to e” }

forall_adj_faces(f, v)
{ “the faces adjacent to node v are successively assigned to f” }

4. Implementation

Graphs are implemented by doubly linked lists of nodes and edges. Most operations take constant time, except for all_nodes, all_edges, del_all_nodes, del_all_edges, make_map, make_planar_map, compute_faces, all_faces, make_map, clear, write, and read which take time $O(n + m)$, and adj_edges, adj_nodes, out_edges, in_edges, and adj_faces which take time $O(output size)$ where $n$ is the current number of nodes and $m$ is the current number of edges. The space requirement is $O(n + m)$.

12.2 Parameterized Graphs (GRAPH)

1. Definition

A parameterized graph $G$ is a graph whose nodes and edges contain additional (user defined) data. Every node contains an element of a data type $vtype$, called the node type of $G$ and every edge contains an element of a data type $etype$ called the edge type of $G$. We use $<v, w, y>$ to denote an edge $(v, w)$ with information $y$ and $<x>$ to denote a node with information $x$.

All operations defined for the basic graph type graph are also defined on instances of any parameterized graph type GRAPH$<vtype, etype>$. For parameterized graph there are additional operations to access or update the information associated with its nodes and edges. Instances of a parameterized graph type can be used wherever an instance of the data type graph can be used, e.g., in assignments and as arguments to functions with formal parameters of type graph&. If a function $f(graph& G)$ is called with an argument $Q$ of type GRAPH$<vtype, etype>$ then inside $f$ only the basic graph structure of $Q$ can be accessed. The node and edge entries are hidden. This allows the design of generic
graph algorithms, i.e., algorithms accepting instances of any parametrized graph type as argument.

```cpp
#include <LEDA/graph/graph.h>
```

### 2. Types

```cpp
GRAPH<vtype, etype>::node_value_type
```
the type of node data (\(vtype\)).

```cpp
GRAPH<vtype, etype>::edge_value_type
```
the type of edge data (\(etyp\)).

### 3. Creation

```cpp
GRAPH<vtype, etype> G; creates an instance \(G\) of type \(GRAPH<vtype, etype>\) and initializes it to the empty graph.
```

### 4. Operations

```cpp
const vtype& G.inf(node v) returns the information of node \(v\).
const vtype& G[node v] returns a reference to \(G.inf(v)\).
const etype& G.inf(edge e) returns the information of edge \(e\).
const etype& G[edge e] returns a reference to \(G.inf(e)\).
```

```cpp
node_array<vtype>& G.node_data() makes the information associated with the nodes of \(G\) available as a node array of type \(node_array<vtype>\).
edge_array<etype>& G.edge_data() makes the information associated with the edges of \(G\) available as an edge array of type \(edge_array<etype>\).
```

```cpp
void G.assign(node v, const vtype& x) makes \(x\) the information of node \(v\).
void G.assign(edge e, const etype& x) makes \(x\) the information of edge \(e\).
```

```cpp
node G.new_node(const vtype& x) adds a new node \(<x>\) to \(G\) and returns it.
node G.new_node(node u, const vtype& x, int dir) adds a new node \(<x>\) to \(G\) and returns it. \(v\) is inserted in front of \((dir = leda::before)\) or behind \((dir = leda::behind)\) node \(u\) into the list of all nodes.
**edge**  
\[ G.\text{new_edge}(\text{node } v, \text{ node } w, \text{ const etype}& x) \]

adds a new edge \(<v, w, x>\) to \(G\) by appending it to \(\text{adj}\_\text{edges}(v)\) and to \(\text{in}\_\text{edges}(w)\) and returns it.

**edge**  
\[ G.\text{new_edge}(\text{edge } e, \text{ node } w, \text{ const etype}& x, \text{ int } \text{dir } = \text{leda}::\text{behind}) \]

adds a new edge \(<\text{source}(e), w, x>\) to \(G\) by inserting it behind \((\text{dir } = \text{leda}::\text{behind})\) or in front of \((\text{dir } = \text{leda}::\text{before})\) edge \(e\) into \(\text{adj}\_\text{edges}(\text{source}(e))\) and appending it to \(\text{in}\_\text{edges}(w)\). Returns the new edge.

**edge**  
\[ G.\text{new_edge}(\text{node } v, \text{ edge } e, \text{ const etype}& x, \text{ int } \text{dir } = \text{leda}::\text{behind}) \]

adds a new edge \(<v, \text{target}(e), x>\) to \(G\) by inserting it behind \((\text{dir } = \text{leda}::\text{behind})\) or in front of \((\text{dir } = \text{leda}::\text{before})\) edge \(e\) into \(\text{in}\_\text{edges}(\text{target}(e))\) and appending it to \(\text{adj}\_\text{edges}(v)\). Returns the new edge.

**edge**  
\[ G.\text{new_edge}(\text{edge } e1, \text{ edge } e2, \text{ const etype}& x, \text{ int } \text{d1 } = \text{leda}::\text{behind}, \text{ int } \text{d2 } = \text{leda}::\text{behind}) \]

adds a new edge \(x = (\text{source}(e1), \text{target}(e2), x)\) to \(G\). \(x\) is inserted in front of \((\text{if } \text{d1 } = \text{leda}::\text{before})\) or behind \((\text{if } \text{d1 } = \text{leda}::\text{behind})\) edge \(e1\) into \(\text{adj}\_\text{edges}(\text{source}(e1))\) and in front of \((\text{if } \text{d2 } = \text{leda}::\text{before})\) or behind \((\text{if } \text{d2 } = \text{leda}::\text{behind})\) edge \(e2\) into \(\text{in}\_\text{edges}(\text{target}(e2))\) (if \(G\) is directed) or \(\text{adj}\_\text{edges}(\text{target}(e2))\) (if \(G\) is undirected). The operation returns the new edge \(x\).

**edge**  
\[ G.\text{new_edge}(\text{node } v, \text{ edge } e1, \text{ node } w, \text{ edge } e2, \text{ const etype}& x, \text{ int } \text{d1 } = \text{leda}::\text{behind}, \text{ int } \text{d2 } = \text{leda}::\text{behind}) \]

adds a new edge \((v, w, x)\) to \(G\) by inserting it in front of \((\text{if } \text{d1 } = \text{leda}::\text{before})\) or behind \((\text{if } \text{d1 } = \text{leda}::\text{behind})\) edge \(e1\) into \(\text{adj}\_\text{edges}(v)\) and in front of \((\text{if } \text{d2 } = \text{leda}::\text{before})\) or behind \((\text{if } \text{d2 } = \text{leda}::\text{behind})\) edge \(e2\) into \(\text{adj}\_\text{edges}(w)\), and returns it.

**Precondition:** \(G\) is undirected, \(v \neq w\), \(e1\) is incident to \(v\), and \(e2\) is incident to \(w\).

**edge**  
\[ G.\text{new_edge}(\text{node } v, \text{ edge } e, \text{ node } w, \text{ const etype}& x, \text{ int } \text{d } = \text{leda}::\text{behind}) \]

adds a new edge \((v, w, x)\) to \(G\) by inserting it in front of \((\text{if } d = \text{leda}::\text{before})\) or behind \((\text{if } d = \text{leda}::\text{behind})\) edge \(e\) into \(\text{adj}\_\text{edges}(v)\) and appending it to \(\text{adj}\_\text{edges}(w)\), and returns it.

**Precondition:** \(G\) is undirected, \(v \neq w\), \(e1\) is incident to \(v\), and \(e\) is incident to \(v\).
void G.sort_nodes(const list<node>& vl)
makes vl the node list of G.
Precondition: vl contains exactly the nodes of G.

void G.sort_edges(const list<edge>& el)
makes el the edge list of G.
Precondition: el contains exactly the edges of G.

void G.sort_nodes()
the nodes of G are sorted increasingly according to their contents.
Precondition: vtype is linearly ordered.

void G.sort_edges()
the edges of G are sorted increasingly according to their contents.
Precondition: etype is linearly ordered.

void G.write(string fname)
writes G to the file with name fname. The output operators operator ≪ (ostream&, const vtype&) and operator ≪ (ostream&, const etype&) (cf. section 1.6) must be defined.

int G.read(string fname)
reads G from the file with name fname. The input operators operator ≫ (istream&, vtype&) and operator ≫ (istream&, etype&) (cf. section 1.6) must be defined. Returns error code
1 if file fname does not exist
2 if graph is not of type GRAPH<vtype, etype>
3 if file fname does not contain a graph
0 if reading was successful.

5. Implementation

Parameterized graph are derived from directed graph. All additional operations for manipulating the node and edge entries take constant time.
12.3 Static Graphs (static_graph)

1. Definition

1.1 Motivation. The data type static_graph representing static graph is the result of two observations:

First, most graph algorithms do not change the underlying graph, they work on a constant or static graph and second, different algorithms are based on different models (we call them categories) of graph.

The LEDA data type graph represents all types of graph used in the library, such as directed, undirected, and bidirected graph, networks, planar maps, and geometric graph. It provides the operations for all of these graph in one fat interface. For efficiency reasons it makes sense to provide special graph data types for special purposes. The template data type static_graph, which is parameterized with the graph category, provides specialized implementations for some of these graph types.

1.2 Static Graphs. A static graph consists of a fixed sequence of nodes and edges. The parameterized data type static_graph<category, node_data, edge_data> is used to represent static graph. The first template parameter category defines the graph category and is taken from {directed_graph, bidirectional_graph, opposite_graph} (see 1.3 for the details). The last two parameters are optional and can be used to define user-defined data structures to be included into the node and edge objects (see 1.4 for the details). An instance G of the parameterized data type static_graph contains a sequence V of nodes and a sequence E of edges. New nodes or edges can be appended only in a construction phase which has to be started by calling G.start_construction() and terminated by G.finish_construction(). For every node or edge x we define index(x) to be equal to the rank of x in its sequence. During the construction phase, the sequence of the source node index of all inserted edges must be non-decreasing. After the construction phase both sequences V and E are fixed.

1.3 Graph Categories. We distinguish between five categories where currently only the first three are supported by static_graph:

- Directed Graphs (directed_graph) represent the concept of a directed graph by providing the ability to iterate over all edges incident to a given node v and to ask for the target node of a given edge e.

- Bidirectional Graphs (bidirectional_graph) extend directed graph by supporting in addition iterations over all incoming edges at a given node v and to ask for the source node of a given edge e.
Opposite Graphs (\textit{opposite\_graph}) are a variant of the bidirectional graph category. They do not support the computation of the source or target node of a given edge but allow to walk from one terminal \( v \) of an edge \( e \) to the other \textit{opposite one}.

Not yet implemented are bidirected and undirected graph.

1.4 Node and Edge Data. Static graph support several efficient ways - efficient compared to using \textit{node\_arrays}, \textit{edge\_arrays}, \textit{node\_maps}, and \textit{edge\_maps} - to associate data with the edges and nodes of the graph.

1.4.1 Dynamic Slot Assignment: It is possible to attach two optional template parameters \textit{data\_slots\<\textit{int}\>\>} at compile time:

\begin{verbatim}
static_graph<directed_graph, data_slots<3>, data_slots<1> > G;
\end{verbatim}

specifies a static directed graph \( G \) with three additional node slots and one additional edge slot. Node and edge arrays can use these data slots, instead of allocating an external array. This method is also supported for the standard LEDA data type \textit{graph}. Please see the manual page for \textit{node\_array} resp. \textit{edge\_array} (esp. the operations \textit{use\_node\_data} resp. \textit{use\_edge\_data}) for the details.

The method is called \textit{dynamic slot assignment} since the concrete arrays are assigned during runtime to the slots.

1.4.2 Static Slot Assignment: This method is even more efficient. A variant of the node and edge arrays, the so-called \textit{node\_slot} and \textit{edge\_slot} data types, are assigned to the slots during compilation time. These types take three parameters: the element type of the array, an integer slot number, and the type of the graph:

\begin{verbatim}
node\_slot\<E, graph_t, slot\>;
edge\_slot\<E, graph_t, slot\>;
\end{verbatim}

Here is an example for the use of static slot assignment in a maxflow graph algorithm. It uses three node slots for storing distance, excess, and a successor node, and two edge slots for storing the flow and capacity.

\begin{verbatim}
typedef static_graph<opposite_graph, data_slots<3>, data_slots<2> > maxflow_graph;
node\_slot\<node, maxflow_graph, 0\> succ;
node\_slot\<int, maxflow_graph, 1\> dist;
node\_slot\<edge, maxflow_graph, 2\> excess;
edge\_slot\<int, maxflow_graph, 0\> flow;
edge\_slot\<int, maxflow_graph, 1\> cap;
\end{verbatim}
When using the data types node_slot resp. edge_slot one has to include the files LEDA/graph/edge_slot.h.

1.4.3 Customizable Node and Edge Types: It is also possible to pass any structure derived from data_slots<int> as second or third parameter. Thereby the nodes and edges are extended by named data members. These are added in addition to the data slots specified in the base type. In the example

```c
struct flow_node:public data_slots<1>
{
  int excess;
  int level;
}

struct flow_edge:public data_slots<2>
{
  int flow;
  int cap;
}

typedef static_graph<bidirectional_graph, flow_node, flow_edge> flow_graph;
```

there are three data slots (one of them unnamed) associated with each node and four data slots (two of them unnamed) associated with each edge of a flow_graph.

The named slots can be used as follows:

```c
flow_graph::node v;
forall_nodes(v, G) v->excess = 0;
```

```c
#include <LEDA/graph/static_graph.h>
```

2. Creation

```c
static_graph<category, node_data = data_slots < 0 >, edge_data = data_slots < 0 > > G;
```

creates an empty static graph G. category is either directed_graph, or bidirectional_graph, or opposite_graph. The use of the other parameters is explained in the section Node and Edge Data given above.

3. Types

```c
static_graph::node the node type. Note: It is different from graph::node.
static_graph::edge the edge type. Note: It is different from graph::edge.
```
4. Operations

The interface consists of two parts. The first part - the basic interface - is independent from the actual graph category, the specified operations are common to all graphs. The second part of the interface is different for every category and contains macros to iterate over incident edges or adjacent nodes and methods for traversing a given edge.

\[
\text{void } G.\text{start}construction(int n, int m)
\]

starts the construction phase for a graph with up to \(n\) nodes and \(m\) edges.

\[
\text{node } G.\text{new}node()
\]

creates a new node, appends it to \(V\), and returns it. The operation may only be called during construction phase and at most \(n\) times.

\[
\text{edge } G.\text{new}edge(node v, node w)
\]

creates the edge \((v, w)\), appends it to \(E\), and returns it. The operation may only be called during construction phase and at most \(m\) times.

Precondition: All edges \((u, v)\) of \(G\) with \(\text{index}(u) < \text{index}(v)\) have been created before.

\[
\text{void } G.\text{finish}construction()
\]

terminates the construction phase.

\[
\text{int } \forall\text{nodes}(v, G) \quad v \text{ iterates over the node sequence.}
\]

\[
\text{int } \forall\text{edges}(e, G) \quad e \text{ iterates over the edge sequence.}
\]

Static Directed Graphs (\textit{static\_graph<directed\_graph>})

For this category the basic interface of \textit{static\_graph} is extended by the operations:

\[
\text{node } G.\text{target}(edge e) \quad \text{returns the target node of } e.
\]

\[
\text{node } G.\text{outdeg}(node v) \quad \text{returns the number of outgoing edges of } v.
\]

\[
\text{int } \forall\text{out\_edges}(e, v) \quad e \text{ iterates over all edges with } \text{source}(e) = v.
\]

Static Bidirectional Graphs (\textit{static\_graph<bidirectional\_graph>})

For this category the basic interface of \textit{static\_graph} is extended by the operations:

\[
\text{node } G.\text{target}(edge e) \quad \text{returns the target node of } e.
\]

\[
\text{node } G.\text{source}(edge e) \quad \text{returns the source node of } e.
\]
CHAPTER 12. GRAPHS AND RELATED DATA TYPES

**node** $G$.outdeg(node $v$) returns the number of outgoing edges of $v$.

**node** $G$.indeg(node $v$) returns the number of incoming edges of $v$.

**int** forall_out_edges($e$, $v$) $e$ iterates over all edges with $source(e) = v$.

**int** forall_in_edges($e$, $v$) $e$ iterates over all edges with $target(e) = v$.

**Static Opposite Graphs** (static\_graph<opposite\_graph>)

For this category the basic interface of static\_graph is extended by the operations:

**node** $G$.opposite(edge $e$, node $v$) returns the opposite to $v$ along $e$.

**node** $G$.outdeg(node $v$) returns the number of outgoing edges of $v$.

**node** $G$.indeg(node $v$) returns the number of incoming edges of $v$.

**int** forall_out_edges($e$, $v$) $e$ iterates over all edges with $source(e) = v$.

**int** forall_in_edges($e$, $v$) $e$ iterates over all edges with $target(e) = v$.

**5. Example**

The simple example illustrates how to create a small graph and assign some values. To see how static graph can be used in a max flow algorithm - please see the source file mfs.c in the directory test/flow.

```c
#include <LEDA/graph/graph.h>
#include <LEDA/graph/node_slot.h>
#include <LEDA/graph/edge_slot.h>
#include <LEDA/core/array.h>

using namespace leda;

struct node_weight:public data_slots<0>
{ int weight; }

struct edge_cap:public data_slots<0>
{ int cap; }

typedef static_graph<opposite_graph, node_weight, edge_cap> static_graph;
typedef static_graph::node st_node;
typedef static_graph::edge st_edge;
```
int main ()
{
    static_graph G;
    array<st_node> v(4);
    array<st_edge> e(4);
    G.start_construction(4,4);

    for(int i =0; i < 4; i++) v[i] = G.new_node();
    e[0] = G.new_edge(v[0], v[1]);
    e[1] = G.new_edge(v[0], v[2]);
    e[2] = G.new_edge(v[1], v[2]);
    e[3] = G.new_edge(v[3], v[1]);
    G.finish_construction();
    st_node v;
    st_edge e;
    forall_nodes(v, G) v->weight = 1;
    forall_edges(e, G) e->cap = 10;

    return 0;
}
12.4 Undirected Graphs (ugraph)

1. Definition
An instance $U$ of the data type ugraph is an undirected graph as defined in section 12.1.

```c
#include <LEDA/graph/ugraph.h>
```

2. Creation

```c
ugraph U;
```
creates an instance $U$ of type ugraph and initializes it to the empty undirected graph.

```c
ugraph U(const graph& G);
```
creates an instance $U$ of type ugraph and initializes it with an undirected copy of $G$.

3. Operations

see section 12.1.

4. Implementation

see section 12.1.

12.5 Parameterized Ugraph (UGRAPH)

1. Definition
A parameterized undirected graph $G$ is an undirected graph whose nodes and contain additional (user defined) data (cf. 12.2). Every node contains an element of a data type vtype, called the node type of $G$ and every edge contains an element of a data type etype called the edge type of $G$.

```c
#include <LEDA/graph/ugraph.h>
```

```c
UGRAPH<vtype, etype> U;
```
creates an instance $U$ of type ugraph and initializes it to the empty undirected graph.

2. Operations

see section 12.2.
3. Implementation

see section 12.2.
12.6 Planar Maps (planar_map)

1. Definition

An instance $M$ of the data type planar_map is the combinatorial embedding of a planar graph, i.e., $M$ is bidirected (for every edge $(v, w)$ of $M$ the reverse edge $(w, v)$ is also in $M$) and there is a planar embedding of $M$ such that for every node $v$ the ordering of the edges in the adjacency list of $v$ corresponds to the counter-clockwise ordering of these edges around $v$ in the embedding.

```c
#include <LEDA/graph/planar_map.h>
```

2. Creation

```c
planar_map M(const graph& G);
```

creates an instance $M$ of type planar_map and initializes it to the planar map represented by the directed graph $G$.

*Precondition:* $G$ represents a bidirected planar map, i.e. for every edge $(v, w)$ in $G$ the reverse edge $(w, v)$ is also in $G$ and there is a planar embedding of $G$ such that for every node $v$ the ordering of the edges in the adjacency list of $v$ corresponds to the counter-clockwise ordering of these edges around $v$ in the embedding.

3. Operations

```c
edge M.new_edge(edge e1, edge e2)
```

inserts the edge $e = (source(e_1), source(e_2))$ and its reversal into $M$ and returns $e$.

*Precondition:* $e_1$ and $e_2$ are bounding the same face $F$.

The operation splits $F$ into two new faces.

```c
face M.del_edge(edge e)
```

deletes the edge $e$ and its reversal from $M$. The two faces adjacent to $e$ are united to one new face which is returned.

```c
edge M.split_edge(edge e)
```

splits edge $e = (v, w)$ and its reversal $r = (w, v)$ into edges $(v, u)$, $(u, w)$, $(w, u)$, and $(u, v)$. Returns the edge $(u, w)$.

```c
node M.new_node(const list<edge>& el)
```

splits the face bounded by the edges in $el$ by inserting a new node $u$ and connecting it to all source nodes of edges in $el$.

*Precondition:* all edges in $el$ bound the same face.

```c
node M.new_node(face f)
```

splits face $f$ into triangles by inserting a new node $u$ and connecting it to all nodes of $f$. Returns $u$. 
12.6. PLANAR MAPS (PLANAR_MAP)

list<edge> M.triangulate() triangulates all faces of \( M \) by inserting new edges. The list of inserted edges is returned.

4. Implementation

Planar maps are implemented by parameterized directed graph. All operations take constant time, except for new_edge and del_edge which take time \( O(f) \) where \( f \) is the number of edges in the created faces and triangulate and straight_line_embedding which take time \( O(n) \) where \( n \) is the current size (number of edges) of the planar map.
12.7 Parameterized Planar Maps (PLANAR_MAP)

1. Definition

A parameterized planar map $M$ is a planar map whose nodes, edges and faces contain additional (user defined) data. Every node contains an element of a data type $vtype$, called the node type of $M$; every edge contains an element of a data type $etype$, called the edge type of $M$; and every face contains an element of a data type $ftype$ called the face type of $M$. All operations of the data type planar_map are also defined for instances of any parameterized planar_map type. For parameterized planar maps there are additional operations to access or update the node and face entries.

```c
#include <LEDA/graph/planar_map.h>
```

2. Creation

```c
PLANAR_MAP<vtype, etype, ftype>
M(const GRAPH<vtype, etype>& G);
```

creates an instance $M$ of type PLANAR_MAP<vtype, etype, ftype> and initializes it to the planar map represented by the parameterized directed graph $G$. The node and edge entries of $G$ are copied into the corresponding nodes and edges of $M$. Every face $f$ of $M$ is assigned the default value of type $ftype$.

Precondition: $G$ represents a planar map.

3. Operations

```c
const vtype& M.inf(node v) returns the information of node $v$.
const etype& M.inf(edge e) returns the information of node $v$.
const ftype& M.inf(face f) returns the information of face $f$.
vtype& M[node v] returns a reference to the information of node $v$.
etype& M[edge e] returns a reference to the information of edge $e$.
ftype& M[face f] returns a reference to the information of face $f$.
void M.assign(node v, const vtype& x) makes $x$ the information of node $v$.
void M.assign(edge e, const etype& x) makes $x$ the information of node $v$.
```
12.7.  PARAMETERIZED PLANAR MAPS (PLANAR_MAP)  

```cpp
void M.assign(face f, const ftype& x)
makes x the information of face f.
```

```cpp
edge M.new_edge(edge e1, edge e2, const ftype& y)
inserts the edge \( e = (\text{source}(e_1), \text{source}(e_2)) \) and its
reversal edge \( e' \) into \( M \).
Precondition: \( e_1 \) and \( e_2 \) are bounding the same face \( F \).
The operation splits \( F \) into two new faces \( f, \) adjacent
to edge \( e, \) and \( f', \) adjacent to edge \( e', \) with \( \inf(f) = \inf(F) \) and \( \inf(f') = y. \)
```

```cpp
edge M.split_edge(edge e, const vtype& x)
splits edge \( e = (v, w) \) and its reversal \( r = (w, v) \) into
edges \( (v, u), (u, w), (w, u), \) and \( (u, v) \). Assigns inform-
ation \( x \) to the created node \( u \) and returns the edge
\( (u, w) \).
```

```cpp
node M.new_node(list<edge>& el, const vtype& x)
splits the face bounded by the edges in \( el \) by inserting
a new node \( u \) and connecting it to all source nodes of
edges in \( el \). Assigns information \( x \) to \( u \) and returns
\( u \).
Precondition: all edges in \( el \) bound the same face.
```

```cpp
node M.new_node(face f, const vtype& x)
splits face \( f \) into triangles by inserting a new node \( u \)
with information \( x \) and connecting it to all nodes of
\( f \). Returns \( u \).
```

4. Implementation

Parameterized planar maps are derived from planar maps. All additional operations for
manipulating the node and edge contents take constant time.
12.8 Node Arrays (node_array)

1. Definition

An instance $A$ of the parameterized data type $\text{node_array}<E>$ is a partial mapping from the node set of a graph $G$ to the set of variables of type $E$, called the element type of the array. The domain $I$ of $A$ is called the index set of $A$ and $A(v)$ is called the element at position $v$. $A$ is said to be valid for all nodes in $I$. The array access operator $A[v]$ checks its precondition ($A$ must be valid for $v$). The check can be turned off by compiling with the flag \texttt{-DLEDA\_CHECKING\_OFF}.

```c
#include <LEDA/graph/node_array.h>
```

2. Creation

```c
node_array<E> $A$; creates an instance $A$ of type node_array\textit{<E>} with empty index set.

node_array<E> $A$(const graph_t& $G$); creates an instance $A$ of type node_array\textit{<E>} and initializes the index set of $A$ to the current node set of graph $G$.

node_array<E> $A$(const graph_t& $G$, E $x$); creates an instance $A$ of type node_array\textit{<E>}, sets the index set of $A$ to the current node set of graph $G$ and initializes $A(v)$ with $x$ for all nodes $v$ of $G$.

node_array<E> $A$(const graph_t& $G$, int $n$, E $x$); creates an instance $A$ of type node_array\textit{<E>} valid for up to $n$ nodes of graph $G$ and initializes $A(v)$ with $x$ for all nodes $v$ of $G.

Precondition: $n \geq |V|$.

$A$ is also valid for the next $n - |V|$ nodes added to $G$.

3. Operations

```c
const graph_t& $A$.get_graph() returns a reference to the graph of $A$.

E& $A[node \ v]$ returns the variable $A(v)$.

Precondition: $A$ must be valid for $v$.

void $A$.init(const graph_t& $G$) sets the index set $I$ of $A$ to the node set of $G$, i.e., makes $A$ valid for all nodes of $G$.

void $A$.init(const graph_t& $G$, E $x$) makes $A$ valid for all nodes of $G$ and sets $A(v) = x$ for all nodes $v$ of $G$.
12.8. NODE ARRAYS ( NODE_ARRAY )

\[ A.\text{init}(const \ graph_t& G, \ int \ n, \ E \ x) \]

makes \( A \) valid for at most \( n \) nodes of \( G \) and sets \( A(v) = x \) for all nodes \( v \) of \( G \).

\textit{Precondition:} \( n \geq |V| \).

\( A \) is also valid for the next \( n - |V| \) nodes added to \( G \).

\[ A.\text{use_node_data}(const \ graph_t& G) \]

use free data slots in the nodes of \( G \) (if available) for storing the entries of \( A \). If no free data slot is available in \( G \), an ordinary \textit{node_array}\(<E>\) is created. The number of additional data slots in the nodes and edges of a graph can be specified in the \textit{graph::graph}(\int \ n_{slots}, \ int \ e_{slots}) constructor. The result is \textit{true} if a free slot is available and \textit{false} otherwise.

\[ A.\text{use_node_data}(const \ graph_t& G, \ E \ x) \]

use free data slots in the nodes of \( G \) (if available) for storing the entries of \( A \) and initializes \( A(v) = x \) for all nodes \( v \) of \( G \). If no free data slot is available in \( G \), an ordinary \textit{node_array}\(<E>\) is created. The number of additional data slots in the nodes and edges of a graph can be specified in the \textit{graph::graph}(\int \ n_{slots}, \ int \ e_{slots}) constructor. The result is \textit{true} if a free slot is available and \textit{false} otherwise.

4. Implementation

Node arrays for a graph \( G \) are implemented by C++vectors and an internal numbering of the nodes and edges of \( G \). The access operation takes constant time, \textit{init} takes time \( O(n) \), where \( n \) is the number of nodes in \( G \). The space requirement is \( O(n) \).

\textbf{Remark:} A node array is only valid for a bounded number of nodes of \( G \). This number is either the number of nodes of \( G \) at the moment of creation of the array or it is explicitly set by the user. Dynamic node arrays can be realized by node maps (cf. section 12.11).
12.9 Edge Arrays (edge_array)

1. Definition

An instance $A$ of the parameterized data type $\text{edge}\_\text{array}<E>$ is a partial mapping from the edge set of a graph $G$ to the set of variables of type $E$, called the element type of the array. The domain $I$ of $A$ is called the index set of $A$ and $A(e)$ is called the element at position $e$. $A$ is said to be valid for all edges in $I$. The array access operator $A[e]$ checks its precondition ($A$ must be valid for $e$). The check can be turned off by compiling with the flag `-DLEDA_CHECKING_OFF`.

```c
#include <LEDA/graph/edge_array.h>
```

2. Creation

- `edge_array<E> A;` creates an instance $A$ of type $\text{edge}\_\text{array}<E>$ with empty index set.
- `edge_array<E> A(const graph_t& G);` creates an instance $A$ of type $\text{edge}\_\text{array}<E>$ and initializes the index set of $A$ to be the current edge set of graph $G$.
- `edge_array<E> A(const graph_t& G, E x);` creates an instance $A$ of type $\text{edge}\_\text{array}<E>$, sets the index set of $A$ to the current edge set of graph $G$ and initializes $A(v)$ with $x$ for all edges $v$ of $G$.
- `edge_array<E> A(const graph_t& G, int n, E x);` creates an instance $A$ of type $\text{edge}\_\text{array}<E>$ valid for up to $n$ edges of graph $G$ and initializes $A(e)$ with $x$ for all edges $e$ of $G$. 
  "Precondition: $n \geq |E|$. 
  $A$ is also valid for the next $n - |E|$ edges added to $G$."

3. Operations

- `const graph_t& A.get_graph()` returns a reference to the graph of $A$.
- `E& A[\text{edge } e]` returns the variable $A(e)$. 
  "Precondition: $A$ must be valid for $e$."
- `void A.init(const graph_t& G)` sets the index set $I$ of $A$ to the edge set of $G$, i.e., makes $A$ valid for all edges of $G$.
- `void A.init(const graph_t& G, E x)` makes $A$ valid for all edges of $G$ and sets $A(e) = x$ for all edges $e$ of $G$. 
void A.init(const graph_t& G, int n, E x)

makes A valid for at most n edges of G and sets \( A(e) = x \)
for all edges \( e \) of G.

Precondition: \( n \geq |E| \).

A is also valid for the next \( n - |E| \) edges added to G.

bool A.use_edge_data(const graph_t& G, E x)

use free data slots in the edges of G (if available) for
storing the entries of A. The number of additional data
slots in the nodes and edges of a graph can be specified
in the graph::graph(int n_slots, int e_slots) constructor.
The result is true if a free slot is available and false
otherwise.

4. Implementation

Edge arrays for a graph \( G \) are implemented by C++vectors and an internal numbering
of the nodes and edges of \( G \). The access operation takes constant time, init takes time
\( O(n) \), where \( n \) is the number of edges in \( G \). The space requirement is \( O(n) \).

Remark: An edge array is only valid for a bounded number of edges of \( G \). This number
is either the number of edges of \( G \) at the moment of creation of the array or it is explicitly
set by the user. Dynamic edge arrays can be realized by edge maps (cf. section 12.12).
12.10 Face Arrays ( face_array )

1. Definition

An instance $A$ of the parameterized data type face_array$<E>$ is a partial mapping from the face set of a graph $G$ to the set of variables of type $E$, called the element type of the array. The domain $I$ of $A$ is called the index set of $A$ and $A(f)$ is called the element at position $f$. $A$ is said to be valid for all faces in $I$. The array access operator $A[f]$ checks its precondition ($A$ must be valid for $f$). The check can be turned off by compiling with the flag `-DLEDA_CHECKING_OFF`.

```cpp
#include <LEDA/graph/face_array.h>
```

2. Creation

```cpp
face_array$<E>$ A;  // creates an instance $A$ of type face_array$<E>$ with empty index set.

face_array$<E>$ A(const graph_t& G);  // creates an instance $A$ of type face_array$<E>$ and initializes the index set of $A$ to the current face set of graph $G$.

face_array$<E>$ A(const graph_t& G, E x);  // creates an instance $A$ of type face_array$<E>$, sets the index set of $A$ to the current face set of graph $G$ and initializes $A(f)$ with $x$ for all faces $f$ of $G$.

face_array$<E>$ A(const graph_t& G, int n, E x);  // creates an instance $A$ of type face_array$<E>$ valid for up to $n$ faces of graph $G$ and initializes $A(f)$ with $x$ for all faces $f$ of $G$.  
Precondition: $n \geq |V|$.
$A$ is also valid for the next $n - |V|$ faces added to $G$.
```

3. Operations

```cpp
const graph_t& A.get_graph()  // returns a reference to the graph of $A$.

E& A[face f]  // returns the variable $A(f)$.
Precondition: $A$ must be valid for $f$.

void A.init(const graph_t& G)  // sets the index set $I$ of $A$ to the face set of $G$, i.e., makes $A$ valid for all faces of $G$.

void A.init(const graph_t& G, E x)  // makes $A$ valid for all faces of $G$ and sets $A(f) = x$ for all faces $f$ of $G$.
```
void A.init(const graph_t& G, int n, E x)
    makes A valid for at most n faces of G and sets A(f) = x
    for all faces f of G.
    Precondition: n ≥ |V|.
    A is also valid for the next n − |V| faces added to G.

bool A.use_face_data(const graph_t& G, E x)
    use free data slots in the faces of G (if available) for
    storing the entries of A. The number of additional data
    slots in the nodes and edges of a graph can be specified
    in the graph::graph(int n_slots, int e_slots) constructor.
    The result is true if a free slot is available and false
    otherwise.

4. Implementation

Node arrays for a graph G are implemented by C++vectors and an internal numbering of
the faces and edges of G. The access operation takes constant time, init takes time O(n),
where n is the number of faces in G. The space requirement is O(n).

Remark: A face array is only valid for a bounded number of faces of G. This number is
either the number of faces of G at the moment of creation of the array or it is explicitely
set by the user. Dynamic face arrays can be realized by face maps (cf. section 12.11).
12.11 Node Maps (node_map)

1. Definition
An instance of the data type node_map<\textit{E}> is a map for the nodes of a graph \(G\), i.e., equivalent to map<node, E> (cf. 8.4). It can be used as a dynamic variant of the data type node_array (cf. 12.8). \textbf{New:} Since node_map<\textit{E}> is derived from node_array<\textit{E}> node maps can be passed (by reference) to functions with node array parameters. In particular, all LEDA graph algorithms expecting a node_array<\textit{E}>& argument can be passed a node_map<\textit{E}> instead.

```cpp
#include <LEDA/graph/node_map.h>
```

2. Creation

node_map<\textit{E}> \(M\); \hspace{1em} introduces a variable \(M\) of type node_map<\textit{E}> and initializes it to the map with empty domain.

node_map<\textit{E}> \(M(\text{const graph_t}& G)\);
\hspace{1em} introduces a variable \(M\) of type node_map<\textit{E}> and initializes it with a mapping \(m\) from the set of all nodes of \(G\) into the set of variables of type \(E\). The variables in the range of \(m\) are initialized by a call of the default constructor of type \(E\).

node_map<\textit{E}> \(M(\text{const graph_t}& G, \text{E} x)\);
\hspace{1em} introduces a variable \(M\) of type node_map<\textit{E}> and initializes it with a mapping \(m\) from the set of all nodes of \(G\) into the set of variables of type \(E\). The variables in the range of \(m\) are initialized with a copy of \(x\).

3. Operations

\textit{const graph_t}& \(M.get\_graph()\) returns a reference to the graph of \(M\).

\textit{void} \(M.init()\) \hspace{1em} makes \(M\) a node map with empty domain.

\textit{void} \(M.init(\text{const graph_t}& G)\)
\hspace{1em} makes \(M\) a mapping \(m\) from the set of all nodes of \(G\) into the set of variables of type \(E\). The variables in the range of \(m\) are initialized by a call of the default constructor of type \(E\).

\textit{void} \(M.init(\text{const graph_t}& G, \text{E} x)\)
\hspace{1em} makes \(M\) a mapping \(m\) from the set of all nodes of \(G\) into the set of variables of type \(E\). The variables in the range of \(m\) are initialized with a copy of \(x\).
bool M.use_node_data(const graph_t& G, E x)
use free data slots in the nodes of G (if available) for storing the entries of A. The number of additional data slots in the nodes and edges of a graph can be specified in the `graph::graph(int n_slots, int e_slots)` constructor. The result is `true` if a free slot is available and `false` otherwise.

E& M[node v] returns the variable M(v).

4. Implementation

Node maps either use free node slots or they are implemented by an efficient hashing method based on the internal numbering of the nodes or they use. In each case an access operation takes expected time O(1).
12.12 Edge Maps (edge_map)

1. Definition

An instance of the data type `edge_map<E>` is a map for the edges of a graph $G$, i.e., equivalent to `map<edge,E>` (cf. 8.4). It can be used as a dynamic variant of the data type `edge_array` (cf. 12.9). New: Since `edge_map<E>` is derived from `edge_array<E>` edge maps can be passed (by reference) to functions with edge array parameters. In particular, all LEDA graph algorithms expecting an `edge_array<E>&` argument can be passed an `edge_map<E>&` instead.

```cpp
#include <LEDA/graph/edge_map.h>
```

2. Creation

```cpp
edge_map<E> M;  // introduces a variable M of type edge_map<E> and initializes it to the map with empty domain.
```

```cpp
edge_map<E> M(const graph_t& G);  // introduces a variable M of type edge_map<E> and initializes it with a mapping m from the set of all edges of G into the set of variables of type E. The variables in the range of m are initialized by a call of the default constructor of type E.
```

```cpp
edge_map<E> M(const graph_t& G, E x);  // introduces a variable M of type edge_map<E> and initializes it with a mapping m from the set of all edges of G into the set of variables of type E. The variables in the range of m are initialized with a copy of x.
```

3. Operations

```cpp
const graph_t& M.get_graph() returns a reference to the graph of M.
```

```cpp
void M.init() makes M a edge map with empty domain.
```

```cpp
void M.init(const graph_t& G) makes M a mapping m from the set of all edges of G into the set of variables of type E. The variables in the range of m are initialized by a call of the default constructor of type E.
```

```cpp
void M.init(const graph_t& G, E x) makes M a mapping m from the set of all edges of G into the set of variables of type E. The variables in the range of m are initialized with a copy of x.
```
12.12. **EDGE MAPS (EDGE_MAP)**

bool \( M.\text{use\_edge\_data(const graph\_t\& G, E x)} \)

use free data slots in the edges of \( G \) (if available) for storing the entries of \( A \). The number of additional data slots in the nodes and edges of a graph can be specified in the `graph::graph(int n\_slots, int e\_slots)` constructor. The result is `true` if a free slot is available and `false` otherwise.

\[ E\& \quad M[\text{edge } e] \] returns the variable \( M(v) \).

4. Implementation

Edge maps are implemented by an efficient hashing method based on the internal numbering of the edges. An access operation takes expected time \( O(1) \).
12.13 Face Maps ( face_map )

1. Definition

An instance of the data type face_map<$E>$ is a map for the faces of a graph $G$, i.e., equivalent to map<face, $E$> (cf. 8.4). It can be used as a dynamic variant of the data type face_array (cf. 12.10). New: Since face_map<$E>$ is derived from face_array<$E$> face maps can be passed (by reference) to functions with face array parameters. In particular, all LEDA graph algorithms expecting a face_array<$E$>& argument can be passed a face_map<$E$> instead.

#include <LEDA/graph/face_map.h>

2. Creation

face_map<$E$> $M$; introduces a variable $M$ of type face_map<$E$> and initializes it to the map with empty domain.

face_map<$E$> $M$(const graph_t& $G$);
introduces a variable $M$ of type face_map<$E$> and initializes it with a mapping $m$ from the set of all faces of $G$ into the set of variables of type $E$. The variables in the range of $m$ are initialized by a call of the default constructor of type $E$.

face_map<$E$> $M$(const graph_t& $G$, $E$ $x$);
introduces a variable $M$ of type face_map<$E$> and initializes it with a mapping $m$ from the set of all faces of $G$ into the set of variables of type $E$. The variables in the range of $m$ are initialized with a copy of $x$.

3. Operations

cnst graph_t& $M$.get_graph() returns a reference to the graph of $M$.

void $M$.init() makes $M$ a face map with empty domain.

void $M$.init(const graph_t& $G$)
makes $M$ a mapping $m$ from the set of all faces of $G$ into the set of variables of type $E$. The variables in the range of $m$ are initialized by a call of the default constructor of type $E$.

void $M$.init(const graph_t& $G$, $E$ $x$)
makes $M$ a mapping $m$ from the set of all faces of $G$ into the set of variables of type $E$. The variables in the range of $m$ are initialized with a copy of $x$. 
4. Implementation

Face maps are implemented by an efficient hashing method based on the internal numbering of the faces. An access operation takes expected time $O(1)$. 

$E & \quad M[\text{face } f] \quad \text{returns the variable } M(v)$. 
12.14 Two Dimensional Node Arrays (node_matrix)

1. Definition

An instance \( M \) of the parameterized data type \( \text{node_matrix}<E> \) is a partial mapping from the set of node pairs \( V \times V \) of a graph to the set of variables of data type \( E \), called the element type of \( M \). The domain \( I \) of \( M \) is called the index set of \( M \). \( M \) is said to be valid for all node pairs in \( I \). A node matrix can also be viewed as a node array with element type \( \text{node_array}<E> \) (\( \text{node_array}<\text{node_array}<E>> \)).

\[ \text{#include <LEDA/graph/node_matrix.h}> \]

2. Creation

\[ \text{node_matrix}<E> M; \] creates an instance \( M \) of type \( \text{node_matrix}<E> \) and initializes the index set of \( M \) to the empty set.

\[ \text{node_matrix}<E> M(\text{const graph_t}& G); \]
creates an instance \( M \) of type \( \text{node_matrix}<E> \) and initializes the index set to be the set of all node pairs of graph \( G \), i.e., \( M \) is made valid for all pairs in \( V \times V \) where \( V \) is the set of nodes currently contained in \( G \).

\[ \text{node_matrix}<E> M(\text{const graph_t}& G, E x); \]
creates an instance \( M \) of type \( \text{node_matrix}<E> \) and initializes the index set of \( M \) to be the set of all node pairs of graph \( G \), i.e., \( M \) is made valid for all pairs in \( V \times V \) where \( V \) is the set of nodes currently contained in \( G \). In addition, \( M(v, w) \) is initialized with \( x \) for all nodes \( v, w \in V \).

3. Operations

\[ \text{void} \quad M.\text{init}(\text{const graph_t}& G) \]
sets the index set of \( M \) to \( V \times V \), where \( V \) is the set of all nodes of \( G \).

\[ \text{void} \quad M.\text{init}(\text{const graph_t}& G, E x) \]
sets the index set of \( M \) to \( V \times V \), where \( V \) is the set of all nodes of \( G \) and initializes \( M(v, w) \) to \( x \) for all \( v, w \in V \).

\[ \text{const node_array}<E>& M[\text{node } v] \]
returns the node_array \( M(v) \).

\[ \text{const } E& \quad M(\text{node } v, \text{node } w) \]
returns the variable \( M(v, w) \).

\[ \text{Precondition: } M \text{ must be valid for } v \text{ and } w. \]
4. Implementation

Node matrices for a graph $G$ are implemented by vectors of node arrays and an internal numbering of the nodes of $G$. The access operation takes constant time, the init operation takes time $O(n^2)$, where $n$ is the number of nodes currently contained in $G$. The space requirement is $O(n^2)$. Note that a node matrix is only valid for the nodes contained in $G$ at the moment of the matrix declaration or initialization ($init$). Access operations for later added nodes are not allowed.
CHAPTER 12. GRAPHS AND RELATED DATA TYPES

12.15 Two-Dimensional Node Maps (node_map2)

1. Definition

An instance of the data type node_map2\langle E \rangle is a map2 for the pairs of nodes of a graph \( G \), i.e., equivalent to map2<node, node, E> (cf. 8.5). It can be used as a dynamic variant of the data type node_matrix (cf. 12.14).

```cpp
#include <LEDA/graph/node_map2.h>
```

2. Creation

```cpp
node_map2\langle E \rangle M;  \quad \text{introduces a variable } M \text{ of type node_map2}\langle E \rangle \text{ and initializes it to the map2 with empty domain.}
```

```cpp
node_map2\langle E \rangle M(const graph t& G);
```

introduces a variable \( M \) of type node_map2\langle E \rangle and initializes it with a mapping \( m \) from the set of all nodes of \( G \) into the set of variables of type \( E \). The variables in the range of \( m \) are initialized by a call of the default constructor of type \( E \).

```cpp
node_map2\langle E \rangle M(const graph t& G, E x);
```

introduces a variable \( M \) of type node_map2\langle E \rangle and initializes it with a mapping \( m \) from the set of all nodes of \( G \) into the set of variables of type \( E \). The variables in the range of \( m \) are initialized with a copy of \( x \).

3. Operations

```cpp
void M.init() \quad \text{makes } M \text{ a node map2 with empty domain.}
```

```cpp
void M.init(const graph t& G)
```

makes \( M \) to a mapping \( m \) from the set of all nodes of \( G \) into the set of variables of type \( E \). The variables in the range of \( m \) are initialized by a call of the default constructor of type \( E \).

```cpp
void M.init(const graph t& G, E x)
```

makes \( M \) to a mapping \( m \) from the set of all nodes of \( G \) into the set of variables of type \( E \). The variables in the range of \( m \) are initialized with a copy of \( x \).

```cpp
E& M(node v, node w) \quad \text{returns the variable } M(v, w).
```

```cpp
bool M.defined(node v, node w)
```

returns true if \( (v, w) \in \text{dom}(m) \) and false otherwise.
4. Implementation

Node maps are implemented by an efficient hashing method based on the internal numbering of the nodes. An access operation takes expected time $O(1)$. 


12.16 Sets of Nodes (node_set)

1. Definition

An instance $S$ of the data type node_set is a subset of the nodes of a graph $G$. $S$ is said to be valid for the nodes of $G$.

```
#include <LEDA/graph/node_set.h>
```

2. Creation

```
node_set $S$(const graph& $G$);
```

creates an instance $S$ of type node_set valid for all nodes currently contained in graph $G$ and initializes it to the empty set.

3. Operations

```
void $S$.insert($node$ $x$) adds node $x$ to $S$.
void $S$.del($node$ $x$) removes node $x$ from $S$.
bool $S$.member($node$ $x$) returns true if $x$ in $S$, false otherwise.
$node$ $S$.choose() returns a node of $S$.
int $S$.size() returns the size of $S$.
bool $S$.empty() returns true iff $S$ is the empty set.
void $S$.clear() makes $S$ the empty set.
```

4. Implementation

A node set $S$ for a graph $G$ is implemented by a combination of a list $L$ of nodes and a node array of list_items associating with each node its position in $L$. All operations take constant time, except for clear which takes time $O(S)$. The space requirement is $O(n)$, where $n$ is the number of nodes of $G$. 

12.17 Sets of Edges (edge_set)

1. Definition
An instance $S$ of the data type edge_set is a subset of the edges of a graph $G$. $S$ is said to be valid for the edges of $G$.

#include <LEDA/graph/edge_set.h>

2. Creation
$edge_set\ S(const\ graph&\ G);$

creates an instance $S$ of type edge_set valid for all edges currently in graph $G$ and initializes it to the empty set.

3. Operations

- void $S$.insert(edge $x$) adds edge $x$ to $S$.
- void $S$.del(edge $x$) removes edge $x$ from $S$.
- bool $S$.member(edge $x$) returns true if $x$ in $S$, false otherwise.
- edge $S$.choose() returns an edge of $S$.
- int $S$.size() returns the size of $S$.
- bool $S$.empty() returns true iff $S$ is the empty set.
- void $S$.clear() makes $S$ the empty set.

4. Implementation
An edge set $S$ for a graph $G$ is implemented by a combination of a list $L$ of edges and an edge array of list_items associating with each edge its position in $L$. All operations take constant time, except for clear which takes time $O(S)$. The space requirement is $O(n)$, where $n$ is the number of edges of $G$. 
12.18  Lists of Nodes ( node_list )

1. Definition

An instance of the data type node_list is a doubly linked list of nodes. It is implemented more efficiently than the general list type list<node> (7.7). However, it can only be used with the restriction that every node is contained in at most one node_list. Also many operations supported by list<node> (for instance size) are not supported by node_list.

#include <LEDA/graph/node_list.h>

2. Creation

node_list L;       introduces a variable L of type node_list and initializes it with the empty list.

3. Operations

void L.append(node v)   appends v to list L.
void L.push(node v)     adds v at the front of L.
void L.insert(node v, node w)  inserts v after w into L.
    Precondition: w ∈ L.
node L.pop()            deletes the first node from L and returns it.
    Precondition: L is not empty.
node L.pop_back()       deletes the last node from L and returns it.
    Precondition: L is not empty.
void L.del(node v)      deletes v from L.
    Precondition: v ∈ L.
bool L.member(node v)   returns true if v ∈ L and false otherwise.
bool L(node v)          returns true if v ∈ L and false otherwise.
node L.head()          returns the first node in L (nil if L is empty).
node L.tail()           returns the last node in L (nil if L is empty).
node L.succ(node v)    returns the successor of v in L.
    Precondition: v ∈ L.
node L.pred(node v)    returns the predecessor of v in L.
    Precondition: v ∈ L.
node L.cyclic_succ(node v) returns the cyclic successor of v in L.
    Precondition: v ∈ L.
node $L.cyclic\_pred(node\ v)$ returns the cyclic predecessor of $v$ in $L$.
Precondition: $v \in L$.

bool $L.empty( )$ returns true if $L$ is empty and false otherwise.

void $L.clear( )$ makes $L$ the empty list.

forall(x, L) { “the elements of $L$ are successively assigned to $x$” }
12.19 Node Partitions (node_partition)

1. Definition
An instance $P$ of the data type node_partition is a partition of the nodes of a graph $G$.

```
#include <LEDA/graph/node_partition.h>
```

2. Creation

```
node_partition $P$(const graph& $G$);
```
creates a node_partition $P$ containing for every node $v$ in $G$ a block $\{v\}$.

3. Operations

```
int $P$.same_block(node $v$, node $w$)
returns true if $v$ and $w$ belong to the same block of $P$,
false otherwise.

void $P$.union_blocks(node $v$, node $w$)
unites the blocks of $P$ containing nodes $v$ and $w$.

void $P$.split(const list<node>& $L$)
makes all nodes in $L$ to singleton blocks.
Precondition: $L$ is a union of blocks.

void $P$.make_rep(node $v$)
makes $v$ the canonical representative of the block containing $v$.

node $P$.find(node $v$)
returns a canonical representative node of the block that contains node $v$.

int $P$.size(node $v$)
returns the size of the block that contains node $v$.

int $P$.number_of_blocks()
returns the number of blocks of $P$.

node $P$(node $v$)
returns $P$.find($v$).
```

4. Implementation

A node partition for a graph $G$ is implemented by a combination of a partition $P$ and
a node array of partition_item associating with each node in $G$ a partition item in $P$.
Initialization takes linear time, union_blocks takes time $O(1)$ (worst-case), and same_block
and find take time $O(\alpha(n))$ (amortized). The cost of a split is proportional to the cost of
the blocks dismantled. The space requirement is $O(n)$, where $n$ is the number of nodes
of $G$. 
12.20 Node Priority Queues (node_pq)

1. Definition

An instance $Q$ of the parameterized data type `node_pq<P>` is a partial function from the nodes of a graph $G$ to a linearly ordered type $P$ of priorities. The priority of a node is sometimes called the information of the node. For every graph $G$ only one `node_pq<P>` may be used and every node of $G$ may be contained in the queue at most once (cf. section 9.1 for general priority queues).

```c++
#include <LEDA/graph/node_pq.h>
```

2. Creation

```c++
node_pq<P> Q(const graph_t& G);
```

creates an instance $Q$ of type `node_pq<P>` for the nodes of graph $G$ with $\text{dom}(Q) = \emptyset$.

3. Operations

```c++
void Q.insert(node v, const P& x)
```

adds the node $v$ with priority $x$ to $Q$.

Precondition: $v \notin \text{dom}(Q)$.

```c++
const P& Q.prio(node v)
```

returns the priority of node $v$.

Precondition: $v \in \text{dom}(Q)$.

```c++
bool Q.member(node v)
```

returns true if $v$ in $Q$, false otherwise.

```c++
void Q.decrease_p(node v, const P& x)
```

makes $x$ the new priority of node $v$.

Precondition: $x \leq Q.prio(v)$.

```c++
node Q.find_min()
```

returns a node with minimal priority (nil if $Q$ is empty).

```c++
void Q.del(node v)
```

removes the node $v$ from $Q$.

```c++
node Q.del_min()
```

removes a node with minimal priority from $Q$ and returns it (nil if $Q$ is empty).

```c++
node Q.del_min(P& x)
```

as above, in addition the priority of the removed node is assigned to $x$.

```c++
void Q.clear()
```

makes $Q$ the empty node priority queue.

```c++
int Q.size()
```

returns $|\text{dom}(Q)|$.

```c++
int Q.empty()
```

returns true if $Q$ is the empty node priority queue, false otherwise.
const $P \& Q.\inf(node \ v)$ returns the priority of node $v$.

4. Implementation

Node priority queues are implemented by binary heaps and node arrays. Operations insert, del_node, del_min, decrease_p take time $O(\log m)$, find_min and empty take time $O(1)$ and clear takes time $O(m)$, where $m$ is the size of $Q$. The space requirement is $O(n)$, where $n$ is the number of nodes of $G$. 
12.21 Bounded Node Priority Queues (b_node_pq)

1. Definition

An instance of the data type \( \text{b_node_pq}<N> \) is a priority queue of nodes with integer priorities with the restriction that the size of the minimal interval containing all priorities in the queue is bounded by \( N \), the minimum priority is never decreasing, and every node is contained in at most one queue. When applied to the empty queue the del_min - operation returns a special default minimum node defined in the constructor of the queue.

#include <LEDA/graph/b_node_pq.h>

2. Creation

\( \text{b_node_pq}<N> \) \( PQ \); introduces a variable \( PQ \) of type \( \text{b_node_pq}<N> \) and initializes it with the empty queue with default minimum node nil.

\( \text{b_node_pq}<N> \) \( PQ(\text{node } v) \);

introduces a variable \( PQ \) of type \( \text{b_node_pq}<N> \) and initializes it with the empty queue with default minimum node \( v \).

3. Operations

\( \text{node } PQ.\text{del_min}() \) removes the node with minimal priority from \( PQ \) and returns it (the default minimum node if \( PQ \) is empty).

\( \text{void } PQ.\text{insert}(\text{node } w, \text{int } p) \) adds node \( w \) with priority \( p \) to \( PQ \).

\( \text{void } PQ.\text{del}(\text{node } w, \text{int } = 0) \) deletes node \( w \) from \( PQ \).

4. Implementation

Bounded node priority queues are implemented by cyclic arrays of doubly linked node lists.

5. Example

Using a \( \text{b_node_pq} \) in Dijkstra’s shortest paths algorithm.

```c
int dijkstra(const GRAPH<int,int>& g, node s, node t)
{
    node_array<int> dist(g,MAXINT);
    b_node_pq<100> PQ(t); // on empty queue del_min returns t
    dist[s] = 0;

    for (node v = s; v != t; v = PQ.del_min() )
```
{ int dv = dist[v];
    edge e;
    forall_adj_edges(e,v)
    { node w = g.opposite(v,e);
      int d = dv + g.inf(e);
      if (d < dist[w])
      { if (dist[w] != MAXINT) PQ.del(w);
        dist[w] = d;
        PQ.insert(w,d);
      }
    }
    return dist[t];
}
12.22 Graph Generators (graph_gen)

```c
void complete_graph(graph& G, int n)
    creates a complete graph G with n nodes.

void complete_ugraph(graph& G, int n)
    creates a complete undirected graph G with n nodes.

void random_graph_noncompact(graph& G, int n, int m)
    generates a random graph with n nodes and m edges.
    No attempt is made to store all edges in the same adjacency list consecutively.
    This function is only included for pedagogical reasons.

void random_graph(graph& G, int n, int m, bool no_anti_parallel_edges,
    bool loopfree, bool no_parallel_edges)
    generates a random graph with n nodes and m edges.
    All edges in the same adjacency list are stored consecutively.
    If no_parallel_edges is true then no parallel edges are generated,
    if loopfree is true then no self loops are generated,
    and if no_anti_parallel_edges is true then no anti parallel edges are generated.

void random_graph(graph& G, int n, int m)
    same as random_graph(G, n, m, false, false, false).

void random_simple_graph(graph& G, int n, int m)
    same as random_graph(G, n, m, false, false, true).

void random_simple_loopfree_graph(graph& G, int n, int m)
    same as random_graph(G, n, m, false, true, true).

void random_simple_undirected_graph(graph& G, int n, int m)
    same as random_graph(G, n, m, true, true, true).

void random_graph(graph& G, int n, double p)
    generates a random graph with n nodes. Each edge of the complete graph with n nodes is
    included with probability p.

void test_graph(graph& G)
    creates interactively a user defined graph G.

void complete_bigraph(graph& G, int a, int b, list<node>& A, list<node>& B)
    creates a complete bipartite graph G with a nodes on side A and b nodes on side B.
    All edges are directed from A to B.
```
void random_bigraph(graph& G, int a, int b, int m, list<node>& A, list<node>& B, int k = 1)
    creates a random bipartite graph G with a nodes on side A, b nodes on side B, and m edges. All edges are directed from A to B. If k > 1 then A and B are divided into k groups of about equal size and the nodes in the i-th group of A have their edges to nodes in the i−1-th and i+1-th group in B. All indices are modulo k.

void test_bigraph(graph& G, list<node>& A, list<node>& B)
    creates interactively a user defined bipartite graph G with sides A and B. All edges are directed from A to B.

void grid_graph(graph& G, int n)
    creates a grid graph G with n × n nodes.

void grid_graph(graph& G, node_array<double>& xcoord, node_array<double>& ycoord, int n)
    creates a grid graph G of size n × n embedded into the unit square. The embedding is given by xcoord[v] and ycoord[v] for every node v of G.

void d3_grid_graph(graph& G, int n)
    creates a three-dimensional grid graph G with n×n×n nodes.

void d3_grid_graph(graph& G, node_array<double>& xcoord, node_array<double>& ycoord, node_array<double>& zcoord, int n)
    creates a three-dimensional grid graph G of size n × n × n embedded into the unit cube. The embedding is given by xcoord[v], ycoord[v], and zcoord[v] for every node v of G.

void cmdline_graph(graph& G, int argc, char **argv)
    builds graph G as specified by the command line arguments:
    prog   →  test_graph()
    prog n →  complete_graph(n)
    prog n m →  test_graph(n, m)
    prog file →  G.read_graph(file)

Planar graph: Combinatorial Constructions

A maximal planar map with n nodes, n ≥ 3, has 3n − 6 uedges. It is constructed iteratively. For n = 1, the graph consists of a single isolated node, for n = 2, the graph consists of two nodes and one edge, for n = 3 the graph consists of three nodes and three
uedges. For $n > 3$, a random maximal planar map with $n - 1$ nodes is constructed first and then an additional node is put into a random face.

The generator with the additional parameter $m$ first generates a maximal planar map and then deletes all but $m$ edges.

The generators with the word map replaced by graph, first generate a map and then delete one edge from each uedge.

```cpp
void maximal_planar_map(graph& G, int n)
creates a maximal planar map $G$ with $n$ nodes.

void random_planar_map(graph& G, int n, int m)
creates a random planar map $G$ with $n$ nodes and $m$
edges.

void maximal_planar_graph(graph& G, int n)
creates a maximal planar graph $G$ with $n$ nodes.

void random_planar_graph(graph& G, int n, int m)
creates a random planar graph $G$ with $n$ nodes and $m$
edges.
```

**Planar graph: Geometric Constructions**

We have two kinds of geometric constructions: triangulations of point sets and intersection graph of line segments. The functions $\text{triangulation\_map}$ choose points in the unit square and compute a triangulation of them and the functions $\text{random\_planar\_graph}$ construct the intersection graph of segments.

The generators with the word map replaced by graph, first generate a map and then delete one edge from each uedge.

```cpp
void triangulation_map(graph& G, node_array<double>& xcoord,
node_array<double>& ycoord, int n)
 chooses $n$ random points in the unit square and returns their triangulation as a plane map in $G$. The coordinates of node $v$ are returned as $xcoord[v]$ and $ycoord[v]$. The coordinates are random number of the form $x/K$ where $K = 2^{20}$ and $x$ is a random integer between 0 (inclusive) and $K$ (exclusive).

void triangulation_map(graph& G, list<node>& outer_face,
node_array<double>& xcoord,
node_array<double>& ycoord, int n)
 as above, in addition the list of nodes of the outer face
( convex hull) is returned in $outer\_face$ in clockwise order.
```
void triangulation_map(graph & G, int n)
    as above, but only the map is returned.

void random_planar_map(graph & G, node_array<double> & xcoord,
    node_array<double> & ycoord, int n, int m)
    chooses n random points in the unit square and computes their triangulation as a plane map in G. It then keeps all but m uedges. The coordinates of node v are returned as xcoord[v] and ycoord[v].

void triangulation_graph(graph & G, node_array<double> & xcoord,
    node_array<double> & ycoord, int n)
    calls triangulation_map and keeps only one of the edges comprising a uedge.

void triangulation_graph(graph & G, list<node> & outer_face,
    node_array<double> & xcoord,
    node_array<double> & ycoord, int n)
    calls triangulation_map and keeps only one of the edges comprising a uedge.

void triangulation_graph(graph & G, int n)
    calls triangulation_map and keeps only one of the edges comprising a uedge.

void random_planar_graph(graph & G, node_array<double> & xcoord,
    node_array<double> & ycoord, int n, int m)
    calls random_planar_map and keeps only one of the edges comprising a uedge.

void triangulated_planar_graph(graph & G, int n)
    old name for triangulation_graph.

void triangulated_planar_graph(graph & G, node_array<double> & xcoord,
    node_array<double> & ycoord, int n)
    old name for triangulation_graph.

void triangulated_planar_graph(graph & G, list<node> & outer_face,
    node_array<double> & xcoord,
    node_array<double> & ycoord, int n)
    old name for triangulation_graph.
void random_planar_graph(graph& G, node_array<double>& xcoord, node_array<double>& ycoord, int n)

creates a random planar graph $G$ with $n$ nodes embedded into the unit square. The embedding is given by $xcoord[v]$ and $ycoord[v]$ for every node $v$ of $G$. The generator chooses $n$ segments whose endpoints have random coordinates of the form $x/K$, where $K$ is the smallest power of two greater or equal to $n$, and $x$ is a random integer in 0 to $K - 1$. It then constructs the arrangement defined by the segments and keeps the $n$ nodes with the smallest $x$-coordinates. Finally, it adds edges to make the graph connected.

void random_planar_graph(graph& G, int n)

creates a random planar graph $G$ with $n$ nodes. Uses the preceding function.

Series-Parallel Graphs

void random_sp_graph(graph& G, int n, int m)

creates a random series-parallel graph $G$ with $n$ nodes and $m$ edges.
12.23 Miscellaneous Graph Functions ( graph_misc )

1. Operations

```
#include <LEDA/graph/graph_misc.h>

void CopyGraph(graph& H, const graph& G) constructs a copy H of graph G.

void CopyGraph(GRAPH<node, edge>& H, const graph& G) constructs a copy H of graph G such that H[v] is the node of G that corresponds to v and H[e] is the edge of G that corresponds to e.

void CopyGraph(GRAPH<node, edge>& H, const graph& G, const list<node>& V, const list<edge>& E) constructs a copy H of the subgraph (V, E) of G such that H[v] is the node of G that corresponds to v and H[e] is the edge of G that corresponds to e. Precondition: V is a subset of the nodes of G and E is a subset of V × V.

void CopyGraph(GRAPH<node, edge>& H, const graph& G, const list<edge>& E) constructs a copy H of the subgraph of G induced by the edges in E.

bool Is_Simple(const graph& G) returns true if G is simple, i.e., has no parallel edges, false otherwise.

bool Is_Simple(const graph& G, list<edge>& el) as above, but returns in addition the list of all edges sorted lexicographically by source and target node, i.e, all parallel edges appear consecutively in el.

bool Is_Loopfree(const graph& G) returns true if G is loopfree, i.e., has no edge whose source is equal to its target.

bool Is_Simple_Loopfree(const graph& G) returns true if G is simple and loopfree.

bool Is_Undirected_Simple(const graph& G) returns true if G viewed as an undirected graph is simple, i.e., G is loopfree, simple, and has no anti-parallel edges.
```
**bool** Is_Bidirected(const graph& G)
returns true if every edge has a reversal and false otherwise.

**bool** Is_Bidirected(const graph& G, edge_array<edge>& rev)
computes for every edge $e = (v, w)$ in $G$ its reversal $rev[e] = (w, v)$ in $G$ (nil if not present). Returns true if every edge has a reversal and false otherwise.

**bool** Is_Map(const graph& G)
tests whether $G$ is a map.

**int** Genus(const graph& G)
computes the genus of $G$.
**Precondition:** $G$ must be a map.

**bool** Is_Plane_Map(const graph& G)
tests whether $G$ is a plane map, i.e., whether $G$ is a map of genus zero.

**bool** Is_Planar_Map(const graph& G)
old name for Is_Plane_Map

**bool** Is_Acyclic(const graph& G)
returns true if the directed $G$ is acyclic and false otherwise.

**bool** Is_Acyclic(const graph& G, list<edge>& L)
as above; in addition, constructs a list of edges $L$ whose deletion makes $G$ acyclic.

**bool** Is_Connected(const graph& G)
returns true if the undirected graph underlying $G$ is connected and false otherwise.

**bool** Is_Biconnected(const graph& G)
returns true if the undirected graph underlying $G$ is biconnected and false otherwise.

**bool** Is_Biconnected(const graph& G, node& s)
as above, computes a split vertex $s$ if the result is false.

**bool** Is_Triconnected(const graph& G)
returns true if the undirected graph underlying $G$ is triconnected and false otherwise. The running time is $O(n(n + m))$. 

---

12.23. MISCELLANEOUS GRAPH FUNCTIONS (GRAPH_MISC) 329
bool IsTriconnected(const graph& G, node& s1, node& s2)
    as above, computes a split pair s1, s2 if the result is false.

bool IsBipartite(const graph& G)
    returns true if G is bipartite and false otherwise.

bool IsBipartite(const graph& G, list<node>& A, list<node>& B)
    returns true if G is bipartite and false otherwise. If G is bipartite the two sides are returned in A and B, respectively. If G is not bipartite the node sequence of an odd-length circle is returned in A..

bool IsPlanar(const graph& G) returns true if G is planar and false otherwise.

bool IsSeriesParallel(const graph& G)
    returns true if G is series-parallel and false otherwise.

void MakeAcyclic(graph& G) makes G acyclic by removing all DFS back edges.

list<edge> MakeSimple(graph& G) makes G simple by removing all but one from each set of parallel edges. Returns the list of remaining edges with parallel edges in the original graph.

void MakeBidirected(graph& G, list<edge>& L)
    makes G bidirected by inserting missing reversal edges. Appends all inserted edges to list L.

list<edge> MakeBidirected(graph& G)
    makes G bidirected by inserting missing reversal edges. Returns the list of inserted edges.

void MakeConnected(graph& G, list<edge>& L)
    makes G connected; appends all inserted edges to list L.

list<edge> MakeConnected(graph& G)
    makes G connected; returns the list of inserted edges.

void MakeBiconnected(graph& G, list<edge>& L)
    makes G biconnected; appends all inserted edges to list L.

list<edge> MakeBiconnected(graph& G)
    makes G biconnected; returns the list of inserted edges.
list\textless node\textgreater  \textit{Delete\_Loops}(graph\& \ G) returns the list of nodes with self-loops and deletes all self-loops.
12.24 Markov Chains (markov_chain)

1. Definition

We consider a Markov Chain to be a graph $G$ in which each edge has an associated non-negative integer weight $w[e]$. For every node (with at least one outgoing edge) the total weight of the outgoing edges must be positive. A random walk in a Markov chain starts at some node $s$ and then performs steps according to the following rule:

Initially, $s$ is the current node. Suppose node $v$ is the current node and that $e_0, \ldots, e_{d-1}$ are the edges out of $v$. If $v$ has no outgoing edge no further step can be taken. Otherwise, the walk follows edge $e_i$ with probability proportional to $w[e_i]$ for all $i$, $0 \leq i < d$. The target node of the chosen edge becomes the new current node.

#include <LEDA/graph/markov_chain.h>

2. Creation

markov_chain $M$(const graph& $G$, const edge_array<int>& $w$, node $s = \text{nil}$);
creates a Markov chain for the graph $G$ with edge weights $w$. The node $s$ is taken as the start vertex ($G.$first_node() if $s$ is nil).

3. Operations

void $M$.step(int $T = 1$) performs $T$ steps of the Markov chain.

node $M$.current_node() returns current vertex.

int $M$.current_outdeg() returns the outdegree of the current vertex.

int $M$.number_of_steps() returns number of steps performed.

int $M$.number_of_visits(node $v$) returns number of visits to node $v$.

double $M$.rel_freq_of_visit(node $v$) returns number of visits divided by the total number of steps.
12.25 Dynamic Markov Chains (dynamic_markov_chain)

1. Definition

A Markov Chain is a graph $G$ in which each edge has an associated non-negative integer weight $w[e]$. For every node (with at least one outgoing edge) the total weight of the outgoing edges must be positive. A random walk in a Markov chain starts at some node $s$ and then performs steps according to the following rule:

Initially, $s$ is the current node. Suppose node $v$ is the current node and that $e_0, \ldots, e_{d-1}$ are the edges out of $v$. If $v$ has no outgoing edge no further step can be taken. Otherwise, the walk follows edge $e_i$ with probability proportional to $w[e_i]$ for all $i$, $0 \leq i < d$. The target node of the chosen edge becomes the new current node.

```cpp
#include <LEDA/graph/markov_chain.h>
```

2. Creation

`dynamic_markov_chain M(const graph& G, const edge_array<int>& w, node s = nil);`

creates a Markov chain for the graph $G$ with edge weights $w$. The node $s$ is taken as the start vertex ($G.first_node()$ if $s$ is nil).

3. Operations

```cpp
void M.step(int T = 1) performs T steps of the Markov chain.
node M.current_node() returns current vertex.
int M.current_outdeg() returns the outdegree of the current vertex.
int M.number_of_steps() returns number of steps performed.
int M.number_of_visits(node v) returns number of visits to node v.
double M.rel_freq_of_visit(node v) returns number of visits divided by the total number of steps.
int M.set_weight(edge e, int g) changes the weight of edge $e$ to $g$ and returns the old weight of $e$
```
12.26 GML Parser for Graphs (gml_graph)

1. Definition

An instance parser of the data type gml_graph is a parser for graph in GML format [46]. It is possible to extend the parser by user defined rules. This parser is used by the read_gml of class graph. The following is a small example graph (a triangle) in GML format.

```gml
# This is a comment.
graph [ # Lists start with '['.
directed 1 # This is a directed graph (0 for undirected).

# The following is an object of type string.
# It will be ignored unless you specify a rule for graph.text.
text "This is a string object."
	node [ id 1 ] # This defines a node with id 1.
	node [ id 2 ]
	node [ id 3 ]

edge [ # This defines an edge leading from node 1 to node 2.
	source 1

target 2
]
edge [ source 2
target 3 ]
edge [ source 3
target 1 ]
] # Lists end with ']'.
```

An input in GML format is a list of GML objects. Each object consists of a key word and a value. A value may have one out of four possible types, an integer (type gml_int), a double (type gml_double), a string (type gml_string), or a list of GML objects (type gml_list). Since a value can be a list of objects, we get a tree structure on the input. We can describe a class C of objects being in the same list and having the same key word by the so-called path. The path is the list of key words leading to an object in the class C.

In principle, every data structure can be expressed in GML format. This parser specializes on graphs. A graph is represented by an object with key word graph and type gml_list. The nodes of the graph are objects with path graph.node and type gml_list. Each node has a unique identifier, which is represented by an object of type gml_int with path graph.node.id. An edge is an object of type gml_list with the path graph.edge. Each edge
has a source and a target. These are objects of type \texttt{gml\_int} with path \texttt{graph.edge.source} and \texttt{graph.edge.target}, respectively. The integer values of \texttt{source} and \texttt{target} refer to node identifiers. There are some global graph attributes, too. An object of type \texttt{gml\_int} with path \texttt{graph.directed} determines whether the graph is undirected (value 0) or directed (every other integer). The type of node parameters and edge parameters in parameterized graph (see manual page \texttt{GRAPH}) can be given by objects of type \texttt{gml\_string} with path \texttt{graph.nodeType} and \texttt{graph.edgeType}, respectively. Parameters of nodes and edges are represented by objects of type \texttt{gml\_string} with path \texttt{graph.node.parameter} and \texttt{graph.edge.parameter}, respectively.

No list has to be in a specific order, e.g., you can freely mix \texttt{node} and \texttt{edge} objects in the \texttt{graph} list. If there are several objects in a class where just one object is required like \texttt{graph.node.id}, only the last such object is taken into account.

Objects in classes with no predefined rules are simply ignored. This means that an application \texttt{A} might add specific objects to a graph description in GML format and this description is still readable for another application \texttt{B} which simply does not care about the objects which are specific for \texttt{A}.

This parser supports reading user defined objects by providing a mechanism for dealing with those objects by means of callback functions. You can specify a rule for, e.g., objects with path \texttt{graph.node.weight} and type \texttt{gml\_double} like in the following code fragment.

```c
bool get_node_weight(const gml_object* gobj, graph* G, node v)
{
    double w = gobj->get_double();
    do something with \( w \), the graph and the corresponding node \( v \)
    return true; or false if the operation failed
}
...
main()
{
    char* filename;
    ...
    graph G;
    gml_graph parser(G);
    parser.append("graph"); parser.append("node");
    parser.append("weight");
    parser.add_node_rule_for_cur_path(get_node_weight,gml_double);
    // or short parser.add_node_rule(get_node_weight,gml_double,"weight");
    bool parsing_ok = parser.parse(filename);
    ...
}
```

You can add rules for the graph, for nodes, and for edges. The difference between them is the type. The type of node rules is as in the example above
bool (*gml_node_rule)(const gml_object*, graph*, node), the type for edge rules
is bool (*gml_edge_rule)(const gml_object*, graph*, edge), and the type for
graph rules is bool (*gml_graph_rule)(const gml_object*, graph*). A GML
object is represented by an instance of class gml_object. You can get its
value by using double gml_object::get_double(), int gml_object::get_int() or
char* gml_object::get_string(). If one of your rules returns false during parsing,
then parsing fails and the graph is cleared.

#include <LEDA/graph/gml_graph.h>

2. Creation

gml_graph parser(graph& G);
   creates an instance parser of type gml_graph and initializes it for
   graph G.

gml_graph parser(graph& G, const char* filename);
   creates an instance parser of type gml_graph and reads graph G
   from the file filename.

gml_graph parser(graph& G, istream& ins);
   creates an instance parser of type gml_graph and reads graph G
   from the input stream ins.

3. Operations

3.1 Parsing

bool parser.parse(const char* filename)
   parses the input taken from the file filename using the current set of rules. The graph specified in the constructor is
   set up accordingly. This operation returns false and clears the graph, if syntax or parse errors occur. Otherwise true is
   returned.

bool parser.parse(istream& ins)
   parses the input taken from the input stream ins.

bool parser.parse_string(string s)
   parses the input taken from string s.

3.2 Path Manipulation

void parser.reset_path(); resets the current path to the empty path.
void parser.append(const char * key)
    appends key to the current path.

void parser.goback() removes the last key word from the current path. If the current path is empty this operation has no effect.

3.3 User Defined Rules

void parser.add_graph_rule_for_cur_path(gml_graph_rule f, gml_value_type t)
    adds graph rule f for value type t and for the current path.

void parser.add_node_rule_for_cur_path(gml_node_rule f, gml_value_type t)
    adds node rule f for value type t and for the current path.

void parser.add_edge_rule_for_cur_path(gml_edge_rule f, gml_value_type t)
    adds edge rule f for value type t and for the current path.

void parser.add_graph_rule(gml_graph_rule f, gml_value_type t, char * key = 0)
    adds graph rule f for value type t and path graph.key to parser, if key is specified. Otherwise, f is added for the current path.

void parser.add_node_rule(gml_node_rule f, gml_value_type t, char * key = 0)
    adds node rule f for path graph.node.key (or the current path, if no key is specified) and value type t to parser.

void parser.add_edge_rule(gml_edge_rule f, gml_value_type t, char * key = 0)
    adds edge rule f for path graph.edge.key (or the current path, if no key is specified) and value type t to parser.

void parser.add_new_graph_rule(gml_graph_rule f)
    adds graph rule f to parser. During parsing f is called whenever an object o with path graph and type gml_list is encountered. f is called before objects in the list of o are parsed.

void parser.add_new_node_rule(gml_node_rule f)
    adds node rule f for path graph.node and value type gml_list to parser. f is called before objects in the corresponding list are parsed.

void parser.add_new_edge_rule(gml_edge_rule f)
    adds edge rule f for path graph.edge and value type gml_list to parser. f is called before objects in the corresponding list are parsed.
void parser.add_graph_done_rule(gml_graph_rule f)

adds graph rule \( f \) to parser. During parsing \( f \) is called whenever an object \( o \) with path \( graph \) and type \( gml\_list \) is encountered. \( f \) is called after all objects in the list of \( o \) are parsed.

void parser.add_node_done_rule(gml_node_rule f)

adds node rule \( f \) to parser for path \( graph..node \) and value type \( gml\_list \). \( f \) is called after all objects in the corresponding list are parsed.

void parser.add_edge_done_rule(gml_edge_rule f)

adds edge rule \( f \) to parser for path \( graph..edge \) and value type \( gml\_list \). \( f \) is called after all objects in the corresponding list are parsed.

4. Implementation

The data type gml_graph is realized using lists and maps. It inherits from gml_parser which uses gml_object, gml_objecttree, and gml_pattern. gml_pattern uses dictionaries.
12.27 The LEDA graph input/output format

The following passage describes the format of the output produced by the function

\texttt{graph::write(ostream \& out)}. The output consists of several lines which are separated by endl. Comment-lines have a \# character in the first column and are ignored. The output can be partitioned in three sections:

**Header Section**
The first line always contains the string \texttt{LEDA\_GRAPH}. If the graph type is not parameterized, i.e. graph or ugraph, the following two lines both contain the string \texttt{void}. In case the graph is parameterized, i.e. \texttt{GRAPH} or \texttt{UGRAPH}, these lines contain a description of the node type and the edge type, which is obtained by calling the macro \texttt{LEDA\_TYPE\_NAME}. The fourth line specifies if the graph is either directed (-1) or undirected (-2).

**Nodes Section**
The first line contains \texttt{n}, the number of nodes in the graph. The nodes are ordered and numbered according to their position in the node list of the graph. Each of the following \texttt{n} lines contains the information which is associated with the respective node of the graph. When the information of a node (or an edge) is sent to an output stream, it is always enclosed by the strings \texttt{|{ and }|}. If the graph is not parameterized, then the string between these parantheses is empty, so that all the \texttt{n} lines contain the string \texttt{|{}}.

**Edges Section**
The first line contains \texttt{m}, the number of edges in the graph. The edges of the graph are ordered by two criteria: first according to the number of their source node and second according to their position in the adjacency list of the source node. Each of the next \texttt{m} lines contains the description of an edge which consists of four space-separated parts:

(a) the number of the source node
(b) the number of the target node
(c) the number of the reversal edge or 0, if no such edge is set
(d) the information associated with the edge (cf. nodes section)

**Note:** For the data type \texttt{planar\_map} the order of the edges is important, because the ordering of the edges in the adjacency list of a node corresponds to the counter-clockwise ordering of these edges around the node in the planar embedding. And the information about reversal edges is also vital for this data type.
Chapter 13

Graph Algorithms

This chapter gives a summary of the graph algorithms contained in LEDA, basic graph algorithms for reachability problems, shortest path algorithms, matching algorithms, flow algorithms, . . . .

All graph algorithms are generic, i.e., they accept instances of any user defined parameterized graph type $GRAPH<vtype, etype>$ as arguments.

All graph algorithms are available by including the header file $<LEDA/graph/graph_alg.h>$. Alternatively, one may include a more specific header file.

An important subclass of graph algorithms are network algorithms. The input to most network algorithms is a graph whose edges or nodes are labeled with numbers, e.g., shortest path algorithms get edge costs, network flow algorithms get edge capacities, and min cost flow algorithms get edge capacities and edge costs. We use $NT$ to denote the number type used for the edge and node labels.

Most network algorithms come in three kinds: A templated version in which $NT$ is a template parameter, and reinstatiated and precompiled versions for the number types $int$ (always) and $double$ (except for a small number of functions). The function name of the templated version ends in $T$. Thus $MAX\_FLOW\_T$ is the name of the templated version of the max flow algorithm and $MAX\_FLOW$ is the name of the instantiated version.

In order to use the templated version a file $<LEDA/graph/templates/XXX.h>$ must be included, e.g., in order to use the templated version of the maxflow algorithm, one must include $<LEDA/graph/templates/max\_flow.h>$

Special care should be taken when using network algorithms with a number type $NT$ that can incur rounding error, e.g., the type $double$. The functions perform correctly if the arithmetic is exact. This is the case if all numerical values in the input are integers (albeit stored as a number of type $NT$), if none of the intermediate results exceeds the maximal integer representable by the number type ($2^{52}$ in the case of $doubles$), and if no round-off errors occur during the computation. We give more specific information on the
arithmetic demand for each function below. If the arithmetic incurs rounding error, the
computation may fail in two ways: give a wrong answer or run forever.

13.1 Basic Graph Algorithms (basic_graph_alg)

bool TOPSORT(const graph& G, node_array<int>& ord)

TOPSORT takes as argument a directed graph $G(V, E)$. It sorts $G$ topo-
logically (if $G$ is acyclic) by computing for every node $v \in V$ an integer
$ord[v]$ such that $1 \leq ord[v] \leq |V|$ and $ord[v] < ord[w]$ for all edges
$(v, w) \in E$. TOPSORT returns true if $G$ is acyclic and false otherwise.
The algorithm ([50]) has running time $O(|V| + |E|)$.

bool TOPSORT1(graph& G)

a variant of TOPSORT that rearranges nodes and edges of $G$ in topo-
logical order (edges are sorted by the topological number of their target
nodes).

list<node> DFS(const graph& G, node s, node_array<bool>& reached)

DFS takes as argument a directed graph $G(V, E)$, a node $s$ of $G$ and a
node_array reached of boolean values. It performs a depth first search
starting at $s$ visiting all reachable nodes $v$ with reached[$v$] = false. For
every visited node $v$ reached[$v$] is changed to true. DFS returns the list
of all reached nodes. The algorithm ([85]) has running time $O(|V| + |E|)$.

list<edge> DFS_NUM(const graph& G, node_array<int>& dfsnun,
node_array<int>& compnum)

DFS_NUM takes as argument a directed graph $G(V, E)$. It performs a
depth first search of $G$ numbering the nodes of $G$ in two different ways.
dfsnum is a numbering with respect to the calling time and compnum
a numbering with respect to the completion time of the recursive calls.
DFS_NUM returns a depth first search forest of $G$ (list of tree edges).
The algorithm ([85]) has running time $O(|V| + |E|)$. 
13.1. BASIC GRAPH ALGORITHMS (BASIC_GRAPH_ALG)

`list<node> BFS(const graph& G, node s, node_array<int>& dist)`

BFS takes as argument a directed graph $G(V,E)$, a node $s$ of $G$ and a node array $dist$ of integers. It performs a breadth first search starting at $s$ visiting all nodes $v$ with $dist[v] = -1$ reachable from $s$. The $dist$ value of every visited node is replaced by its distance to $s$. BFS returns the list of all visited nodes. The algorithm ([58]) has running time $O(|V| + |E|)$.

`list<node> BFS(const graph& G, node s, node_array<int>& dist, node_array<edge>& pred)`

performs a breadth first search as described above and computes for every node $v$ the predecessor edge $pred[v]$ in the BFS shortest path tree. (You can use the function `COMPUTE_SHORTEST_PATH` to extract paths from the tree (cf. Section 13.2).)

`int COMPONENTS(const graph& G, node_array<int>& compnum)`

COMPONENTS takes a graph $G(V,E)$ as argument and computes the connected components of the underlying undirected graph, i.e., for every node $v \in V$ an integer $compnum[v]$ from $[0 \ldots c-1]$ where $c$ is the number of connected components of $G$ and $v$ belongs to the $i$-th connected component iff $compnum[v] = i$. COMPONENTS returns $c$. The algorithm ([58]) has running time $O(|V| + |E|)$.

`int STRONG_COMPONENTS(const graph& G, node_array<int>& compnum)`

STRONG_COMPONENTS takes a directed graph $G(V,E)$ as argument and computes for every node $v \in V$ an integer $compnum[v]$ from $[0 \ldots c-1]$ where $c$ is the number of strongly connected components of $G$ and $v$ belongs to the $i$-th strongly connected component iff $compnum[v] = i$. STRONG_COMPONENTS returns $c$. The algorithm ([58]) has running time $O(|V| + |E|)$. 
int \textsc{biconnected}\_components(const graph\& G, edge_array<int>& compnum)

\textsc{biconnected}\_components computes the biconnected components of the undirected version of G. A biconnected component of an undirected graph is a maximal biconnected subgraph and a biconnected graph is a graph which cannot be disconnected by removing one of its nodes. A graph having only one node is biconnected.

Let \( c \) be the number of biconnected components and let \( c' \) be the number of biconnected components containing at least one edge, \( c - c' \) is the number of isolated nodes in \( G \), where a node \( v \) is isolated if is not connected to a node different from \( v \) (it may be incident to self-loops). The function returns \( c \) and labels each edge of \( G \) (which is not a self-loop) by an integer in \([0 \ldots c' - 1]\). Two edges receive the same label iff they belong to the same biconnected component. The edge labels are returned in \textit{compnum}.

Be aware that self-loops receive no label since self-loops are ignored when interpreting a graph as an undirected graph.

The algorithm ([21]) has running time \( O(|V| + |E|) \).

\textbf{GRAPH<node, edge> \textsc{transitive}\_closure(const graph\& G)}

\textsc{transitive}\_closure takes a directed graph \( G = (V, E) \) as argument and computes the transitive closure of \( G \). It returns a directed graph \( G' = (V', E') \) such that \( G'.\text{inf}(.) \) is a bijective mapping from \( V' \) to \( V \) and \( (v, w) \in E' \iff \) there is a path from \( G'.\text{inf}(v) \) to \( G'.\text{inf}(w) \) in \( G \). (The edge information of \( G' \) is undefined.) The algorithm ([40]) has running time \( O(|V| \cdot |E|) \).

\textbf{GRAPH<node, edge> \textsc{transitive}\_reduction(const graph\& G)}

\textsc{transitive}\_reduction takes a directed graph \( G = (V, E) \) as argument and computes the transitive reduction of \( G \). It returns a directed graph \( G' = (V', E') \). The function \( G'.\text{inf}(.) \) is a bijective mapping from \( V' \) to \( V \). The graph \( G \) and \( G' \) have the same reachability relation, i.e. there is a path from \( v' \) to \( w' \) in \( G' \iff \) there is a path from \( G'.\text{inf}(v') \) to \( G'.\text{inf}(w') \) in \( G \). And there is no graph with the previous property and less edges than \( G' \). (The edge information of \( G' \) is undefined.) The algorithm ([40]) has running time \( O(|V| \cdot |E|) \).

\textbf{void make}\_transitively\_closed(graph\& G)

\textit{make}\_transitively\_closed transforms \( G \) into its transitive closure by adding edges.

\textbf{void make}\_transitively\_reduced(graph\& G)

\textit{make}\_transitively\_reduced transforms \( G \) into its transitive reduction by removing edges.
13.2 Shortest Path Algorithms (shortest_path)

Let $G$ be a graph, $s$ a node in $G$, and $c$ a cost function on the edges of $G$. Edge costs may be positive or negative. For a node $v$ let $\mu(v)$ be the length of a shortest path from $s$ to $v$ (more precisely, the infimum of the lengths of all paths from $s$ to $v$). If $v$ is not reachable from $s$ then $\mu(v) = +\infty$ and if $v$ is reachable from $s$ through a cycle of negative cost then $\mu(v) = -\infty$. Let $V^+$, $V^f$, and $V^-$ be the set of nodes $v$ with $\mu(v) = +\infty$, $-\infty < \mu(v) < +\infty$, and $\mu(v) = -\infty$, respectively.

The solution to a single source shortest path problem $(G, s, c)$ is a pair $(\text{dist}, \text{pred})$ where $\text{dist}$ is a node_array$<\text{NT}>$ and $\text{pred}$ is a node_array$<\text{edge}>$ with the following properties.

- $v \in V^+$ iff $v \neq s$ and $\text{pred}[v] = \text{nil}$ and $v \in V^f \cup V^-$ iff $v = s$ or $\text{pred}[v] \neq \text{nil}$.
- $s \in V^f$ if $\text{pred}[s] = \text{nil}$ and $s \in V^-$ otherwise.
- $v \in V^f$ if $v$ is reachable from $s$ by a $P$-path and $s \in V^f$. $P$ restricted to $V^f$ forms a shortest path tree and $\text{dist}[v] = \mu(s, v)$ for $v \in V^f$.
- All $P$-cycles have negative cost and $v \in V^-$ iff $v$ lies on a $P$-cycle or is reachable from a $P$-cycle by a $P$-path.

Most functions in this section are template functions. The template parameter $\text{NT}$ can be instantiated with any number type. In order to use the template version of the function the .h-file

\texttt{#include <LEDA/graph/templates/shortest_path.h>}

must be included. The functions are pre-instantiated with $\text{int}$ and $\text{double}$. The function names of the pre-instantiated versions are without the suffix _T.

Special care should be taken when using the functions with a number type $\text{NT}$ that can incur rounding error, e.g., the type $\text{double}$. The functions perform correctly if all arithmetic performed is without rounding error. This is the case if all numerical values in the input are integers (albeit stored as a number of type $\text{NT}$) and if none of the intermediate results exceeds the maximal integer representable by the number type ($2^{52}$ in the case of $\text{doubles}$). All intermediate results are sums and differences of input values, in particular, the algorithms do not use divisions and multiplications. All intermediate values are bounded by $nC$ where $n$ is the number of nodes and $C$ is the maximal absolute value of any edge cost.
template <class NT>
bool SHORTEST_PATH_T(const graph& G, node s, const edge_array<NT>& c,
node_array<NT>& dist, node_array<edge>& pred)
SHORTEST_PATH solves the single source shortest path problem in the graph \( G(V,E) \) with respect to the source node \( s \) and the cost-function given by the edge_array \( c \). The procedure returns false if there is a negative cycle in \( G \) that is reachable from \( s \) and returns true otherwise. It runs in linear time on acyclic graph, in time \( O(m + n \log n) \) if all edge costs are non-negative, and runs in time \( O(\min(D,n)m) \) otherwise. Here \( D \) is the maximal number of edges on any shortest path.

list<edge> COMPUTE_SHORTEST_PATH(const graph& G, node s, node t,
const node_array<edge>& pred)
computes a shortest path from \( s \) to \( t \) assuming that \( \text{pred} \) stores a valid shortest path tree with root \( s \) (as it can be computed with the previous function). The returned list contains the edges on a shortest path from \( s \) to \( t \). The running time is linear in the length of the path.

template <class NT>
node_array<int> CHECK_SP_T(const graph& G, node s, const edge_array<NT>& c,
const node_array<NT>& dist,
const node_array<edge>& pred)
checks whether the pair \((\text{dist}, \text{pred})\) is a correct solution to the shortest path problem \((G,s,c)\) and returns a node_array<int> \( \text{label} \) with \( \text{label}[v] < 0 \) if \( v \) has distance \(-\infty\) \((-2\) for nodes lying on a negative cycle and \(-1\) for a node reachable from a negative cycle), \( \text{label}[v] = 0 \) if \( v \) has finite distance, and \( \text{label}[v] > 0 \) if \( v \) has distance \(+\infty\). The program aborts if the check fails. The algorithm takes linear time.

template <class NT>
void ACYCLIC_SHORTEST_PATH_T(const graph& G, node s,
const edge_array<NT>& c,
node_array<NT>& dist, node_array<edge>& pred)
solves the single source shortest path problem with respect to source \( s \). The algorithm takes linear time.
Precondition: \( G \) must be acyclic.

template <class NT>
void DIJKSTRA_T(const graph& G, node s, const edge_array<NT>& cost,
node_array<NT>& dist, node_array<edge>& pred)
solves the shortest path problem in a graph with non-negative edges weights.
Precondition: The costs of all edges are non-negative.
13.2. SHORTEST PATH ALGORITHMS (SHORTEST_PATH)

template <class NT>
void DIJKSTRA_T(const graph& G, node s, const edge_array<NT>& cost,
node_array<NT>& dist)

as above, but pred is not computed.

template <class NT>
NT DIJKSTRA_T(const graph& G, node s, node t, const edge_array<NT>& c,
node_array<edge>& pred)

computes a shortest path from s to t and returns its length. The cost of all edges must be non-negative. The return value is unspecified if there is no path from s to t. The array pred records a shortest path from s to t in reverse order, i.e., pred[t] is the last edge on the path. If there is no path from s to t or if s = t then pred[t] = nil. The worst case running time is $O(m + n \log n)$, but frequently much better.

template <class NT>
bool BELLMAN_FORD_B_T(const graph& G, node s, const edge_array<NT>& c,
node_array<NT>& dist, node_array<edge>& pred)

BELLMAN_FORD_B solves the single source shortest path problem in the graph $G(V, E)$ with respect to the source node $s$ and the cost-function given by the edge_array $c$. BELLMAN_FORD_B returns false if there is a negative cycle in $G$ that is reachable from $s$ and returns true otherwise. The algorithm ([10]) has running time $O(\min(D, n)m)$ where $D$ is the maximal number of edges on any shortest path. The algorithm is only included for pedagogical purposes.

void BF_GEN(GRAPH<int, int>& G, int n, int m, bool non_negative = true)

generates a graph with at most $n$ nodes and at most $m$ edges. The edge costs are stored as edge data. The running time of BELLMAN_FORD_B on this graph is $\Omega(nm)$. The edge weights are non-negative if non_negative is true and are arbitrary otherwise.

Precondition: $m \geq 2n$ and $m \leq n^2/2$.

template <class NT>
bool BELLMAN_FORD_T(const graph& G, node s, const edge_array<NT>& c,
node_array<NT>& dist, node_array<edge>& pred)

BELLMAN_FORD_T solves the single source shortest path problem in the graph $G(V, E)$ with respect to the source node $s$ and the cost-function given by the edge_array $c$. BELLMAN_FORD_T returns false if there is a negative cycle in $G$ that is reachable from $s$ and returns true otherwise. The algorithm ([10]) has running time $O(\min(D, n)m)$ where $D$ is the maximal number of edges in any shortest path. The algorithm is never significantly slower than BELLMAN_FORD_B and frequently much faster.
template <class NT>
bool ALL_PAIRS_SHORTEST_PATHS_T(graph& G, const edge_array<NT>& c, 
    node_matrix<NT>& DIST)
returns true if $G$ has no negative cycle and returns false otherwise. In the latter case all values returned in $DIST$ are unspecified. In the former case the following holds for all $v$ and $w$: if $\mu(v,w) < \infty$ then $DIST(v,w) = \mu(v,w)$ and if $\mu(v,w) = \infty$ then the value of $DIST(v,w)$ is arbitrary. The procedure runs in time $O(nm + n^2 \log n)$.

rational MINIMUM_RATIO_CYCLE(graph& G, const edge_array<int>& c, 
    const edge_array<int>& p, list<edge>& C_star)
Returns a minimum cost to profit ratio cycle $C_{\text{star}}$ and the ratio of the cycle. For a cycle $C$ let $c(C)$ be the sum of the $c$-values of the edges on the cycle and let $p(C)$ be the sum of the $p$-values of the edges on the cycle. The cost to profit ratio of the cycle is the quotient $c(C)/p(C)$. The cycle $C_{\text{star}}$ realizes the minimum ratio for any cycle $C$. The procedure runs in time $O(nm \log(n \cdot C \cdot P))$ where $C$ and $P$ are the maximum cost and profit of any edge, respectively. The program returns zero if there is no cycle in $G$.

Precondition: There are no cycles of cost zero or less with respect to either $c$ or $p$. 
13.3 Maximum Flow (max_flow)

Let $G = (V, E)$ be a directed graph, let $s$ and $t$ be distinct vertices in $G$ and let $cap : E \rightarrow \mathbb{R}_{\geq 0}$ be a non-negative function on the edges of $G$. For an edge $e$, we call $cap(e)$ the capacity of $e$. An $(s, t)$-flow or simply flow is a function $f : E \rightarrow \mathbb{R}_{\geq 0}$ satisfying the capacity constraints and the flow conservation constraints:

1. $0 \leq f(e) \leq cap(e)$ for every edge $e \in E$
2. $\sum_{e; source(e)=v} f(e) = \sum_{e; target(e)=v} f(e)$ for every node $v \in V\{s, t\}$

The value of the flow is the net flow into $t$ (equivalently, the net flow out of $s$). The net flow into $t$ is the flow into $t$ minus the flow out of $t$. A flow is maximum if its value is at least as large as the value of any other flow.

All max flow implementations are template functions. The template parameter $NT$ can be instantiated with any number type. In order to use the template version of the function the files

```cpp
#include <LEDA/graph/graph_alg.h>
#include <LEDA/graph/templates/max_flow.h>
```

must be included.

There are pre-instantiations for the number types `int` and `double`. The pre-instantiated versions have the same function names except for the suffix `_T`. In order to use them either

```cpp
#include <LEDA/graph/max_flow.h>
```

or

```cpp
#include <LEDA/graph/graph_alg.h>
```

has to be included (the latter file includes the former). The connection between template functions and pre-instantiated functions is discussed in detail in the section “Templates for Network Algorithms” of the LEDA book.

Special care should be taken when using the template functions with a number type $NT$ that can incur rounding error, e.g., the type `double`. The section “Algorithms on Weighted Graphs and Arithmetic Demand” of the LEDA book contains a general discussion of this issue. The template functions are only guaranteed to perform correctly if all arithmetic performed is without rounding error. This is the case if all numerical values in the input are integers (albeit stored as a number of type $NT$) and if none of the intermediate results exceeds the maximal integer representable by the number type ($2^{53} - 1$ in the case of `doubles`). All intermediate results are sums and differences of input values, in particular, the algorithms do not use divisions and multiplications.
The algorithms have the following arithmetic demands. Let $C$ be the maximal absolute value of any edge capacity. If all capacities are integral then all intermediate values are bounded by $d \cdot C$, where $d$ is the out-degree of the source.

The pre-instantiations for number type `double` compute the maximum flow for a modified capacity function $cap_1$, where for every edge $e$

$$cap_1[e] = \text{sign}(cap[e]) \lfloor |cap[e]| \cdot S \rfloor / S$$

and $S$ is the largest power of two such that $S < 2^{53} / (d \cdot C)$.

The value of the maximum flow for the modified capacity function and the value of the maximum flow for the original capacity function differ by at most $m \cdot d \cdot C \cdot 2^{-52}$.

The following functions are available:

```cpp
template <class NT>
_inline NT MAX_FLOW_T(const graph& G, node s, node t,
const edge_array<NT>& cap, edge_array<NT>& f)
computes a maximum $(s, t)$-flow $f$ in the network $(G, s, t, cap)$ and returns the value of the flow.
The implementation uses the preflow-push method of Goldberg and Tarjan [43] with the local and global relabeling heuristic and the gap heuristic. The highest level rule is used to select active nodes. The section on maximum flow of the LEDA book gives full information.
```

```cpp
template <class NT>
_inline NT MAX_FLOW_T(const graph& G, node s, node t,
const edge_array<NT>& cap, edge_array<NT>& f,
list<node>& st_cut)
as above, also computes a minimum $s - t$ cut in $G$.
```

```cpp
template <class NT>
_inline NT MAX_FLOW_T(const graph& G, node s, node t,
const edge_array<NT>& cap, edge_array<NT>& f,
int& num_pushes, int& num_edge_inspections,
int& num_relabels, int& num_global_relabels, int& num_gaps,
float h)
as above;
The additional parameter report some statistics of the execution (the number of pushes, the number of edge inspections, the number of relabels, the number of global relabels, and the number of nodes moved by the gap heuristic. The parameter $h$ controls the global relabeling heuristic. The global relabeling heuristic is called every $h \cdot m$ edge inspections. The choice $h = 5$ seems reasonable.
```

```cpp
template <class NT>
```
13.3. MAXIMUM FLOW (MAX_FLOW)

```cpp
inline bool CHECK_MAX_FLOW_T(const graph& G, node s, node t,
const edge_array<NT>& cap,
const edge_array<NT>& f)
```
checks whether \( f \) is a maximum flow in the network \((G, s, t, cap)\). The function returns false if this is not the case.

```cpp
bool MAX_FLOW_SCALE_CAPS(const graph& G, node s, edge_array<double>& cap)
```
replaces \( cap[e] \) by \( cap1[e] \) for every edge \( e \), where \( cap1[e] \) is as defined above. The function returns false if the scaling changed some capacity, and returns true otherwise.

```cpp
template <class NT>
inline NT MAX_FLOW_T(graph& G, node s, node t, const edge_array<NT>& lcap,
const edge_array<NT>& ucap, edge_array<NT>& f)
```
computes a maximum \((s, t)\)-flow \( f \) in the network \((G, s, t, ucap)\) s.th. \( f(e) \leq lcap[e] \) for every edge \( e \). If a feasible flow exists, its value returned; otherwise the return value is -1.

The following functions are only available in template form. They allow to study the effect of different heuristics and of different selection rules on the preflow push method. The class SET must provide the following operations: construction, destruction, del, insert, insert0, and clear; see the LEDA book for details. Three implementations are part of the distribution: fifo_set, hl_set, and mfifo_set.

```cpp
template <class NT, class SET>
NT MAX_FLOW_BASIC_T(const graph& G, node s, node t,
const edge_array<NT>& cap, edge_array<NT>& flow,
SET& U, int& num_pushes, int& num_edge_inspections,
int& num_relabels)
```
The basic version of the preflow push algorithm: No heuristic is used.

```cpp
template <class NT, class SET>
inline NT MAX_FLOW_LH_T(const graph& G, node s, node t,
const edge_array<NT>& cap, edge_array<NT>& flow,
SET& U, int& num_pushes, int& num_edge_inspections,
int& num_relabels)
```
The preflow push method with the distinction between low and high nodes.

```cpp
template <class NT, class SET>
inline NT MAX_FLOW_LRH_T(const graph& G, node s, node t,
const edge_array<NT>& cap, edge_array<NT>& flow,
SET& U, int& num_pushes, int& num_edge_inspections,
int& num_relabels)
```
The preflow push method with the distinction between low and high nodes and the local relabeling heuristic.
The preflow push method with the distinction between low and high nodes, the local relabeling heuristic, the global relabeling heuristic, and the two-phase approach. The global relabeling heuristic is applied every $h \cdot m$ edge inspections.

The following generators can be used to generate max-flow problems with integer capacities. The generators return a \texttt{GRAPH\ <int, int> G} and nodes \texttt{s} and \texttt{t}. The edge capacities are available as edge data, i.e., the capacity of edge \texttt{e} is available as \texttt{G[e]} and the edge array of capacities is passed as \texttt{G.edge_data( )}. For detailed information about the generators we refer to the LEDA book.

\begin{verbatim}
void max_flow_gen_rand(GRAPH<int, int>& G, node& s, node& t, int n, int m)
    A random graph with \texttt{n} nodes, \texttt{m} edges, and random edge capacities in \texttt{[2,11]} for the edges out of \texttt{s} and in \texttt{[1,10]} for all other edges.

void max_flow_gen_CG1(GRAPH<int, int>& G, node& s, node& t, int n)
    A generator suggested by Cherkassky and Goldberg.

void max_flow_gen_CG2(GRAPH<int, int>& G, node& s, node& t, int n)
    Another generator suggested by Cherkassky and Goldberg.

void max_flow_gen_AMO(GRAPH<int, int>& G, node& s, node& t, int n)
    A generator suggested by Ahuja, Magnanti, and Orlin.
\end{verbatim}
13.4 Min Cost Flow Algorithms (min_cost_flow)

```cpp
bool MIN_COST_FLOW(graph& G, const edge_array<int>& lcap,
                   const edge_array<int>& ucap,
                   const edge_array<int>& cost,
                   const node_array<int>& supply,
                   edge_array<int>& flow)
```

This variant of MIN_COST_FLOW assumes that \( lcap[e] = 0 \) for every edge \( e \in E \).

```cpp
int MIN_COST_MAX_FLOW(graph& G, node s, node t,
                       const edge_array<int>& cap,
                       const edge_array<int>& cost,
                       edge_array<int>& flow)
```

MIN_COST_MAX_FLOW takes as arguments a directed graph \( G(V, E) \), a source node \( s \), a sink node \( t \), an edge_array \( cap \) giving for each edge in \( G \) a capacity, and an edge_array \( cost \) specifying for each edge an integer cost. It computes for every edge \( e \) in \( G \) a flow \( flow[e] \) such that the total flow from \( s \) to \( t \) is maximal, the total cost of the flow is minimal, and \( 0 \leq flow[e] \leq cap[e] \) for all edges \( e \). MIN_COST_MAX_FLOW returns the total flow from \( s \) to \( t \).


13.5 Minimum Cut (min_cut)

A cut \( C \) in a network is a set \( S \) of nodes that is neither empty nor the entire set of nodes. The weight of a cut is the sum of the weights of the edges having exactly one endpoint in \( S \).

```cpp
int MIN_CUT(const graph& G, const edge_array<int>& weight,
             list<node>& C, bool use_heuristic = true)
MIN_CUT takes a graph \( G \) and an edge_array \( \text{weight} \) that gives for each edge a non-negative integer weight. The algorithm ([82]) computes a cut of minimum weight. A cut of minimum weight is returned in \( C \) and the value of the cut is the return value of the function. The running time is \( O(nm + n^2 \log n) \). The function uses a heuristic to speed up its computation. Precondition: The edge weights are non-negative.
```

```cpp
list<node> MIN_CUT(const graph& G, const edge_array<int>& weight)
as above, but the cut \( C \) is returned.
```

```cpp
int CUT_VALUE(const graph& G, const edge_array<int>& weight,
               const list<node>& C)
returns the value of the cut \( C \).
```
13.6 Maximum Cardinality Matchings in Bipartite Graphs (mcb_matching)

A matching in a graph $G$ is a subset $M$ of the edges of $G$ such that no two share an endpoint. A node cover is a set of nodes $NC$ such that every edge has at least one endpoint in $NC$. The maximum cardinality of a matching is at most the minimum cardinality of a node cover. In bipartite graph, the two quantities are equal.

```
list<edge> MAX_CARD_BIPARTITE_MATCHING(graph& G)
    returns a maximum cardinality matching.
    Precondition: $G$ must be bipartite.
```

```
list<edge> MAX_CARD_BIPARTITE_MATCHING(graph& G, node_array<bool>& NC)
    returns a maximum cardinality matching and a minimum cardinality node cover $NC$. The node cover has the same cardinality as the matching and hence proves the optimality of the matching. Precondition: $G$ must be bipartite.
```

```
bool CHECK_MCB(const graph& G, const list<edge>& M, const node_array<bool>& NC)
    checks that $M$ is a matching in $G$, i.e., that at most one edge in $M$ is incident to any node of $G$, that $NC$ is a node cover, i.e., for every edge of $G$ at least one endpoint is in $NC$ and that $M$ and $NC$ have the same cardinality. The function writes diagnostic output to cerr, if one of the conditions is violated.
```

```
list<edge> MAX_CARD_BIPARTITE_MATCHING(graph& G, const list<node>& A, const list<node>& B)
    returns a maximum cardinality matching. Precondition: $G$ must be bipartite. The bipartition of $G$ is given by $A$ and $B$. All edges of $G$ must be directed from $A$ to $B$.
```

```
list<edge> MAX_CARD_BIPARTITE_MATCHING(graph& G, const list<node>& A, const list<node>& B, node_array<bool>& NC)
    returns a maximum cardinality matching. A minimal node cover is returned in $NC$. The node cover has the same cardinality as the matching and hence proves the maximality of the matching. Precondition: $G$ must be bipartite. The bipartition of $G$ is given by $A$ and $B$. All edges of $G$ must be directed from $A$ to $B$.
```

We offer several implementations of bipartite matching algorithms. All of them require
that the bipartition \((A, B)\) is given and that all edges are directed from \(A\) to \(B\); all of them return a maximum cardinality matching and a minimum cardinality node cover. The initial characters of the inventors are used to distinguish between the algorithms. The common interface is

\[
\text{list<edge> MAX_CARD_BIPARTITE_MATCHING_XX(graph& G, const list<node>& A, const list<node>& B, node_array<bool>& NC, bool use_heuristic = true);}
\]

where XX is to be replaced by either HK, ABMP, FF, or FFB. All algorithms can be asked to use a heuristic to find an initial matching. This is the default.

HK stands for the algorithm due to Hopcroft and Karp [44]. It has running time \(O(\sqrt{nm})\).

ABMP stands for algorithm due to Alt, Blum, Mehlhorn, and Paul [1]. The algorithm has running time \(O(\sqrt{nm})\). The algorithm consists of two major phases. In the first phase all augmenting paths of length less than \(L_{max}\) are found, and in the second phase the remaining augmenting paths are determined. The default value of \(L_{max}\) is \(0.1\sqrt{n}\). \(L_{max}\) is an additional optional parameter of the procedure.

FF stands for the algorithm due to Ford and Fulkerson [34]. The algorithm has running time \(O(nm)\) and FFB stands for a simple and slow version of FF. The algorithm FF has an additional optional parameter \(use\_bfs\) of type \(bool\). If set to true, breadth-first-search is used in the search for augmenting paths, and if set to false, depth-first-search is used.

Be aware that the algorithms_XX change the graph \(G\). They leave the graph structure unchanged but reorder adjacency lists (and hence change the embedding). If this is undesirable you must restore the original order of the adjacency lists as follows.

\[
\text{edge_array<int> edge_number(G); int i = 0;}
\]
\[
\text{forall_nodes(v,G)}
\]
\[
\text{forall_adj_edges(e,G) edge_number[e] = i++;}
\]
\[
\text{call matching algorithm;}
\]
\[
\text{G.sort_edges(edge_number);}
\]

13.7 Bipartite Weighted Matchings and Assignments (mwb_matching)

We give functions
13.7. BIPARTITE WEIGHTED MATCHINGS AND ASSIGNMENTS (MWB_MATCHING)

- to compute maximum and minimum weighted matchings in bipartite graph,
- to check the optimality of matchings, and
- to scale edge weights, so as to avoid round-off errors in computations with the number type double.

All functions for computing maximum or minimum weighted matchings provide a proof of optimality in the form of a potential function pot; see the chapter on bipartite weighted matchings of the LEDA book for a discussion of potential functions.

The functions in this section are template functions. The template parameter NT can be instantiated with any number type. In order to use the template version of the function the appropriate .h-file must be included.

```c
#include <LEDA/graph/templates/mwb_matching.h>
```

There are pre-instantiations for the number types int and double. The pre-instantiated versions have the same function names except for the suffix _T. In order to use them either

```c
#include <LEDA/graph/mwb_matching.h>
```

or

```c
#include <LEDA/graph/graph_alg.h>
```

has to be included (the latter file includes the former). The connection between template functions and pre-instantiated functions is discussed in detail in the section “Templates for Network Algorithms” of the LEDA book. The function names of the pre-instantiated versions and the template versions only differ by an additional suffix _T in the names of the latter ones.

Special care should be taken when using the template functions with a number type NT that can incur rounding error, e.g., the type double. The section “Algorithms on Weighted Graphs and Arithmetic Demand” of the LEDA book contains a general discussion of this issue. The template functions are only guaranteed to perform correctly if all arithmetic performed is without rounding error. This is the case if all numerical values in the input are integers (albeit stored as a number of type NT) and if none of the intermediate results exceeds the maximal integer representable by the number type \(2^{53} - 1\) in the case of doubles. All intermediate results are sums and differences of input values, in particular, the algorithms do not use divisions and multiplications.

The algorithms have the following arithmetic demands. Let \(C\) be the maximal absolute value of any edge cost. If all weights are integral then all intermediate values are bounded by \(3C\) in the case of maximum weight matchings and by \(4nC\) in the case of the other matching algorithms. Let \(f = 3\) in the former case and let \(f = 4n\) in the latter case.
The pre-instantiations for number type `double` compute the optimal matching for a modified weight function \( c1 \), where for every edge \( e \)

\[
c1[e] = \text{sign}(c[e]) |c[e]| \cdot S / S
\]

and \( S \) is the largest power of two such that \( S < 2^{53}/(f \cdot C) \).

The weight of the optimal matching for the modified weight function and the weight of the optimal matching for the original weight function differ by at most \( n \cdot f \cdot C \cdot 2^{-52} \).

template `<class NT>`

`list<edge> MAX_WEIGHT_BIPARTITE_MATCHING_T(graph & G, const edge_array<NT>& c, node_array<NT>&& pot)`

computes a matching of maximal cost and a potential function \( pot \) that is tight with respect to \( M \). The running time of the algorithm is \( O(n \cdot (m + n \log n)) \). The argument \( pot \) is optional. 

*Precondition*: \( G \) must be bipartite.

template `<class NT>`

`list<edge> MAX_WEIGHT_BIPARTITE_MATCHING_T(graph & G, const list<node>& A, const list<node>& B, const edge_array<NT>& c, node_array<NT>&& pot)`

As above. It is assumed that the partition \((A, B)\) witnesses that \( G \) is bipartite and that all edges of \( G \) are directed from \( A \) to \( B \). If \( A \) and \( B \) have different sizes then is is advisable that \( A \) is the smaller set; in general, this leads to smaller running time. The argument \( pot \) is optional.

template `<class NT>`

`bool CHECK_MWBM_T(const graph& G, const edge_array<NT>& c, const list<edge>& M, const node_array<NT>&& pot)`

checks that \( pot \) is a tight feasible potential function with respect to \( M \) and that \( M \) is a matching. Tightness of \( pot \) implies that \( M \) is a maximum weighted matching.

template `<class NT>`

`list<edge> MAX_WEIGHT_ASSIGNMENT_T(graph & G, const edge_array<NT>& c, node_array<NT>&& pot)`

computes a perfect matching of maximal cost and a potential function \( pot \) that is tight with respect to \( M \). The running time of the algorithm is \( O(n \cdot (m + n \log n)) \). If \( G \) contains no perfect matching the empty set of edges is returned. The argument \( pot \) is optional.

*Precondition*: \( G \) must be bipartite.
13.7. BIPARTITE WEIGHTED MATCHINGS AND ASSIGNMENTS (MWB_MATCHING)

template <class NT>
list<edge> MAX_WEIGHT_ASSIGNMENT_T(graph& G, const list<node>& A,
  const list<node>& B,
  const edge_array<NT>& c,
  node_array<NT>& pot)

As above. It is assumed that the partition (A, B) witnesses that G is
bipartite and that all edges of G are directed from A to B. The argument
pot is optional.

template <class NT>
bool CHECK_MAX_WEIGHT_ASSIGNMENT_T(const graph& G,
  const edge_array<NT>& c,
  const list<edge>& M,
  const node_array<NT>& pot)

checks that pot is a tight feasible potential function with respect to M
and that M is a perfect matching. Tightness of pot implies that M is a
maximum cost assignment.

template <class NT>
list<edge> MIN_WEIGHT_ASSIGNMENT_T(graph& G, const edge_array<NT>& c,
  node_array<NT>& pot)

computes a perfect matching of minimal cost and a potential function
pot that is tight with respect to M. The running time of the algorithm
is $O(n \cdot (m + n \log n))$. If G contains no perfect matching the empty set
of edges is returned. The argument pot is optional.
Precondition: G must be bipartite.

template <class NT>
list<edge> MIN_WEIGHT_ASSIGNMENT_T(graph& G, const list<node>& A,
  const list<node>& B,
  const edge_array<NT>& c,
  node_array<NT>& pot)

As above. It is assumed that the partition (A, B) witnesses that G is
bipartite and that all edges of G are directed from A to B. The argument
pot is optional.


**CHAPTER 13. GRAPH ALGORITHMS**

```cpp
template <class NT>
bool CHECK_MIN_WEIGHT_ASSIGNMENT_T(const graph& G,
const edge_array<NT>& c,
const list<edge>& M,
const node_array<NT>& pot)
checks that pot is a tight feasible potential function with respect to M
and that M is a perfect matching. Tightness of pot implies that M is a
minimum cost assignment.
```

```cpp
template <class NT>
list<edge> MWMCB_MATCHING_T(graph& G, const list<node>& A,
const list<node>& B, const edge_array<NT>& c,
node_array<NT>& pot)
Returns a maximum weight matching among the matching of maximum
cardinality. The potential function pot is tight with respect to a modified
cost function which increases the cost of every edge by \( L = 1 + 2kC \) where
\( C \) is the maximum absolute value of any weight and \( k = \min(|A|, |B|) \).
It is assumed that the partition \((A, B)\) witnesses that \(G\) is bipartite and
that all edges of \(G\) are directed from \(A\) to \(B\). If \(A\) and \(B\) have different
sizes, it is advisable that \(A\) is the smaller set; in general, this leads to
smaller running time. The argument pot is optional.
```

```cpp
bool MWBM_SCALE_WEIGHTS(const graph& G, edge_array<double>& c)
replaces \(c[e]\) by \(c1[e]\) for every edge \(e\), where \(c1[e]\) was defined above
and \(f = 3\). This scaling function is appropriate for the maximum weight
matching algorithm. The function returns false if the scaling changed
some weight, and returns true otherwise.
```

```cpp
bool MWA_SCALE_WEIGHTS(const graph& G, edge_array<double>& c)
replaces \(c[e]\) by \(c1[e]\) for every edge \(e\), where \(c1[e]\) was defined above
and \(f = 4n\). This scaling function should be used for the algorithm
that compute minimum of maximum weight assignments or maximum
weighted matchings of maximum cardinality. The function returns false
if the scaling changed some weight, and returns true otherwise.
```

### 13.8 Maximum Cardinality Matchings in General Graphs (mc_matching)

A matching in a graph \(G\) is a subset \(M\) of the edges of \(G\) such that no two share an
endpoint.

An odd-set cover \(OSC\) of \(G\) is a labeling of the nodes of \(G\) with non-negative integers
such that every edge of \(G\) (which is not a self-loop) is either incident to a node labeled 1
or connects two nodes labeled with the same $i$, $i \geq 2$.

Let $n_i$ be the number of nodes labeled $i$ and consider any matching $N$. For $i$, $i \geq 2$, let $N_i$ be the edges in $N$ that connect two nodes labeled $i$. Let $N_1$ be the remaining edges in $N$. Then $|N_i| \leq \lfloor n_i/2 \rfloor$ and $|N_1| \leq n_1$ and hence

$$|N| \leq n_1 + \sum_{i \geq 2} \lfloor n_i/2 \rfloor$$

for any matching $N$ and any odd-set cover $OSC$.

It can be shown that for a maximum cardinality matching $M$ there is always an odd-set cover $OSC$ with

$$|M| = n_1 + \sum_{i \geq 2} \lfloor n_i/2 \rfloor,$$

thus proving the optimality of $M$. In such a cover all $n_i$ with $i \geq 2$ are odd, hence the name.

```
list<edge> MAX_CARD_MATCHING(const graph& G, node_array<int>& OSC, int heur = 0)
```

computes a maximum cardinality matching $M$ in $G$ and returns it as a list of edges. The algorithm ([26], [38]) has running time $O(nm \cdot \alpha(n, m))$. With $heur = 1$ the algorithm uses a greedy heuristic to find an initial matching. This seems to have little effect on the running time of the algorithm.

An odd-set cover that proves the maximality of $M$ is returned in $OSC$.

```
list<edge> MAX_CARD_MATCHING(const graph& G, int heur = 0)
```

as above, but no proof of optimality is returned.

```
bool CHECK_MAX_CARD_MATCHING(const graph& G, const list<edge>& M, node_array<int>& OSC)
```

checks whether $M$ is a maximum cardinality matching in $G$ and $OSC$ is a proof of optimality. Aborts if this is not the case.

### 13.9 General Weighted Matchings (mw_matching)

We give functions

- to compute maximum-weight matchings,
- to compute maximum-weight or minimum-weight perfect matchings, and
- to check the optimality of weighted matchings
in general graph.

You may skip the following subsections and restrict on reading the function signatures and the corresponding comments in order to use these functions. If you are interested in technical details, or if you would like to ensure that the input data is well chosen, or if you would like to know the exact meaning of all output parameters, you should continue reading.

The functions in this section are template functions. It is intended that in the near future the template parameter $NT$ can be instantiated with any number type. **Please note that for the time being the template functions are only guaranteed to perform correctly for the number type int.** In order to use the template version of the function the appropriate .h-file must be included.

```c
#include <LEDA/graph/templates/mw_matching.h>
```

There are pre-instantiations for the number types int. In order to use them either

```c
#include <LEDA/graph/mw_matching.h>
```

or

```c
#include <LEDA/graph/graph_alg.h>
```

has to be included (the latter file includes the former). The connection between template functions and pre-instantiated functions is discussed in detail in the section “Templates for Network Algorithms” of the LEDA book. The function names of the pre-instantiated versions and the template versions only differ by an additional suffix _T in the names of the latter ones.

**Proof of Optimality.** Most of the functions for computing maximum or minimum weighted matchings provide a proof of optimality in the form of a dual solution represented by $pot$, $BT$ and $b$. We briefly discuss their semantics: Each node is associated with a potential which is stored in the node array $pot$. The array $BT$ (type `array<two_tuple<NT, int>>`) is used to represent the nested family of odd cardinality sets which is constructed during the course of the algorithm. For each (non-trivial) blossom $B$, a two tuple $(z_B, p_B)$ is stored in $BT$, where $z_B$ is the potential and $p_B$ is the parent index of $B$. The parent index $p_B$ is set to $-1$ if $B$ is a surface blossom. Otherwise, $p_B$ stores the index of the entry in $BT$ corresponding to the immediate super-blossom of $B$. The index range of $BT$ is $[0, \ldots, k - 1]$, where $k$ denotes the number of (non-trivial) blossoms. Let $B'$ be a sub-blossom of $B$ and let the corresponding index of $B'$ and $B$ in $BT$ be denoted by $i'$ and $i$, respectively. Then, $i' < i$. In $b$ (type `node_array<int>>`) the parent index for each node $u$ is stored ($-1$ if $u$ is not contained in any blossom).
Heuristics for Initial Matching Constructions. Each function can be asked to start with either an empty matching \((heur = 0)\), a greedy matching \((heur = 1)\) or an (adapted) fractional matching \((heur = 2)\); by default, the fractional matching heuristic is used.

Graph Structure. All functions assume the underlying graph \((\text{type } graph)\) to be connected, simple, loopfree and undirected (i.e., no anti-parallel edges).

Edge Weight Restrictions. The algorithms use divisions. In order to avoid rounding errors for the number type \(\text{int}\), please make sure that all edge weights are multiples of 4; the algorithm will automatically multiply all edge weights by 4 if this condition is not met. (Then, however, the returned dual solution is valid only with respect to the modified weight function.) Moreover, in the maximum-weight (non-perfect) matching case all edge weights are assumed to be non-negative.

Arithmetic Demand. The arithmetic demand for integer edge weights is as follows. Let \(C\) denote the maximal absolute value of any edge weight and let \(n\) be the number of nodes of the graph.
In the perfect weighted matching case we have for a potential \(pot[u]\) of a node \(u\) and for a potential \(z_B\) of a blossom \(B\):
\[
-nC/2 \leq pot[u] \leq (n + 1)C/2 \quad \text{and} \quad -nC \leq z_B \leq nC.
\]
In the non-perfect matching case we have for a potential \(pot[u]\) of a node \(u\) and for a potential \(z_B\) of a blossom \(B\):
\[
0 \leq pot[u] \leq C \quad \text{and} \quad 0 \leq z_B \leq C.
\]
The function \(\text{CHECK\_WEIGHTS}\) may be used to test whether the edge weights are feasible or not. It is automatically called at the beginning of each of the algorithms provided in this chapter.

Single Tree vs. Multiple Tree Approach: All functions can either run a single tree approach or a multiple tree approach. In the single tree approach, one alternating tree is grown from a free node at a time. In the multiple tree approach, multiple alternating trees are grown simultaneously from all free nodes. On large instances, the multiple tree approach is significantly faster and therefore is used by default. If \#define \_SST\_APPROACH is defined before the template file is included all functions will run the single tree approach.
**Worst-Case Running Time:** All functions for computing maximum or minimum weighted (perfect or non-perfect) matchings guarantee a running time of $O(nm \log n)$, where $n$ and $m$ denote the number of nodes and edges, respectively.

```cpp
template <class NT>
list<edge> MAX_WEIGHT_MATCHING_T(const graph& G, const edge_array<NT>& w,
        bool check = true, int heur = 2)
    computes a maximum-weight matching $M$ of the undirected graph $G$ with
    weight function $w$. If check is set to true, the optimality of $M$ is checked
    internally. The heuristic used for the construction of an initial matching
    is determined by heur.
Precondition: All edge weights must be non-negative.

template <class NT>
list<edge> MAX_WEIGHT_MATCHING_T(const graph& G, const edge_array<NT>& w,
        node_array<NT>& pot, array<two_tuple<NT, int> >& BT,
        node_array<int>& b,
        bool check = true, int heur = 2)
    computes a maximum-weight matching $M$ of the undirected graph $G$ with
    weight function $w$. The function provides a proof of optimality in
    the form of a dual solution given by pot, BT and $b$. If check is set to
    true, the optimality of $M$ is checked internally. The heuristic used for
    the construction of an initial matching is determined by heur.
Precondition: All edge weights must be non-negative.

template <class NT>
bool CHECK_MAX_WEIGHT_MATCHING_T(const graph& G,
        const edge_array<NT>& w,
        const list<edge>& M,
        const node_array<NT>& pot,
        const array<two_tuple<NT, int> >& BT,
        const node_array<int>& b)
    checks if $M$ together with the dual solution represented by pot, BT and
    $b$ are optimal. The function returns true if $M$ is a maximum-weight
    matching of $G$ with weight function $w$.
```
computes a maximum-weight perfect matching \( M \) of the undirected graph \( G \) and weight function \( w \). If \( G \) contains no perfect matching the empty set of edges is returned. If \( check \) is set to \( true \), the optimality of \( M \) is checked internally. The heuristic used for the construction of an initial matching is determined by \( heur \).

computes a maximum-weight perfect matching \( M \) of the undirected graph \( G \) with weight function \( w \). If \( G \) contains no perfect matching the empty set of edges is returned. The function provides a proof of optimality in the form of a dual solution given by \( pot \), \( BT \) and \( b \). If \( check \) is set to \( true \), the optimality of \( M \) is checked internally. The heuristic used for the construction of an initial matching is determined by \( heur \).

checks if \( M \) together with the dual solution represented by \( pot \), \( BT \) and \( b \) are optimal. The function returns \( true \) iff \( M \) is a maximum-weight perfect matching of \( G \) with weight function \( w \).
list<edge> MIN_WEIGHT_PERFECT_MATCHING_T(const graph& G,
    const edge_array<NT>& w,
    bool check = true,
    int heur = 2)
computes a minimum-weight perfect matching \( M \) of the undirected graph \( G \) with weight function \( w \). If \( G \) contains no perfect matching the empty set of edges is returned. If \( check \) is set to \( true \), the optimality of \( M \) is checked internally. The heuristic used for the construction of an initial matching is determined by \( heur \).

template <class NT>
list<edge> MIN_WEIGHT_PERFECT_MATCHING_T(const graph& G,
    const edge_array<NT>& w,
    node_array<NT>& pot,
    array<two_tuple<NT, int>>& BT,
    node_array<int>& b,
    bool check = true,
    int heur = 2)
computes a minimum-weight perfect matching \( M \) of the undirected graph \( G \) with weight function \( w \). If \( G \) contains no perfect matching the empty set of edges is returned. The function provides a proof of optimality in the form of a dual solution given by \( pot \), \( BT \) and \( b \). If \( check \) is set to \( true \), the optimality of \( M \) is checked internally. The heuristic used for the construction of an initial matching is determined by \( heur \).

template <class NT>
bool CHECK_MIN_WEIGHT_PERFECT_MATCHING_T(const graph& G,
    const edge_array<NT>& w,
    const list<edge>& M,
    const node_array<NT>& pot,
    const array<two_tuple<NT, int>>& BT,
    const node_array<int>& b)
checks if \( M \) together with the dual solution represented by \( pot \), \( BT \) and \( b \) are optimal. The function returns \( true \) iff \( M \) is a minimum-weight matching of \( G \) with weight function \( w \).
bool CHECK_WEIGHTS_T(const graph& G, edge_array<NT>& w, bool perfect)
returns true, if \( w \) is a feasible weight function for \( G \); false otherwise. 
\textit{perfect} must be set to \textit{true} in the perfect matching case; otherwise it 
must be set to \textit{false}. If the edge weights are not multiplicative of 4 all 
edge weights will be scaled by a factor of 4. The modified weight function 
is returned in \( w \) then. This function is automatically called by each of 
the maximum weighted matching algorithms provided in this chapter, the 
user does not have to take care of it.
13.10 Stable Matching (stable_matching)

We are given a bipartite graph $G = (A \cup B, E)$ in which the edges incident to every vertex are linearly ordered. The order expresses preferences. A matching $M$ in $G$ is stable if there is no pair $(a, b) \in E \setminus M$ such that (1) $a$ is unmatched or prefers $b$ over its partner in $M$ and (2) $b$ is unmatched or prefers $a$ over its partner in $M$. In such a situation $a$ has the intention to switch to $b$ and $b$ has the intention to switch to $a$, i.e., the pairing is unstable.

We provide a function to compute a correct input graph from the preference data, a function that computes the stable matching when the graph is given and a function that checks whether a given matching is stable.

```cpp
void StableMatching(const graph& G, const list<node>& A,
                    const list<node>& B, list<edge>& M)
    // The function takes a bipartite graph G with sides A and B and computes a maximal stable matching M
    // which is A-optimal. The graph is assumed to be bidirected, i.e., for each (a, b) \in E we also have (b, a) \in E.
    // It is assumed that adjacency lists record the preferences of the vertices. The running time is \( O(n + m) \).
    // Precondition: The graph G is bidirected and a map. Sets A and B only contain nodes of graph G. In addition they are disjoint from each other.

bool CheckStableMatching(const graph& G, const list<node>& A,
                         const list<node>& B, const list<edge>& M)
    // returns true if M is a stable matching in G. The running time is \( O(n + m) \).
    // Precondition: A and B only contain nodes from G. The graph G is bipartite with respect to lists A and B.
```
The function takes a list of objects $\text{InputA}$ and a list of objects $\text{InputB}$. The objects are represented by integer numbers, multiple occurrences of the same number in the same list are ignored. The maps $\text{preferencesA}$ and $\text{preferencesB}$ give for each object $i$ the list of partner candidates with respect to a matching. The lists are decreasingly ordered according to the preferences. The function computes the input data $\text{G}$, $\text{A}$ and $\text{B}$ for calling the function $\text{StableMatching(const\,graph&, ...)}$. The maps $\text{nodes_a}$ and $\text{nodes_b}$ provide the objects in $\text{A}$ and $\text{B}$ corresponding to the nodes in the graph.

Precondition: The entries in the lists in the preference maps only contain elements from $\text{InputB}$ resp. $\text{InputA}$.

There are no multiple occurrences of an element in the same such list.
13.11 Minimum Spanning Trees (min-span)

\[
\text{list<edge> SPANNING_TREE(const graph& } G)\\
\text{SPANNING_TREE takes as argument a graph } G(V,E). \text{ It computes a spanning tree } T \text{ of the underlying undirected graph, SPANNING_TREE returns the list of edges of } T. \text{ The algorithm ([58]) has running time } O(|V| + |E|).\\
\]

\[
\text{void SPANNING_TREE1(graph& } G)\\
\text{SPANNING_TREE takes as argument a graph } G(V,E). \text{ It computes a spanning tree } T \text{ of the underlying undirected graph by deleting the edges in } G \text{ that do not belong to } T. \text{ The algorithm ([58]) has running time } O(|V| + |E|).\\
\]

\[
\text{list<edge> MIN_SPANNING_TREE(const graph& } G, \text{ const edge_array<int>& cost})\\
\text{MIN_SPANNING_TREE takes as argument an undirected graph } G(V,E) \text{ and an edge_array cost giving for each edge an integer cost. It computes a minimum spanning tree } T \text{ of } G, \text{ i.e., a spanning tree such that the sum of all edge costs is minimal. MIN_SPANNING_TREE returns the list of edges of } T. \text{ The algorithm ([52]) has running time } O(|E| \log |V|).\\
\]

\[
\text{list<edge> MIN_SPANNING_TREE(const graph& } G, \text{ const leda_cmp_base<edge>& cmp)\\}
\text{A variant using a compare object to compare edge costs.}\\
\text{list<edge> MIN_SPANNING_TREE(const graph& } G, \text{ int (*cmp)(const edge& , const edge& )})\\
\text{A variant using a compare function to compare edge costs.}\\
\]
13.12 Euler Tours ( euler.tour )

An Euler tour in an undirected graph $G$ is a cycle using every edge of $G$ exactly once. A graph has an Euler tour if it is connected and the degree of every vertex is even.

```cpp
bool Euler_Tour(const graph& G, list<two_tuple<edge, int>> & T)

The function returns true if the undirected version of
G has an Euler tour. The Euler tour is returned in
T. The items in T are of the form $(e, \pm 1)$, where
the second component indicates the traversal direction
d of the edge. If $d = +1$, the edge is traversed in
forward direction, and if $d = -1$, the edge is traversed
in reverse direction. The running time is $O(n + m)$.

bool Check_Euler_Tour(const graph& G, const list<two_tuple<edge, int>> & T)
returns true if T is an Euler tour in G. The running
time is $O(n + m)$.

bool Euler_Tour(graph& G, list<edge>& T)
The function returns true if the undirected version of
G has an Euler tour. G is reoriented such that every
node has indegree equal to its outdegree and an Euler
tour (of the reoriented graph) is returned in T. The
running time is $O(n + m)$.

bool Check_Euler_Tour(const graph& G, const list<edge>& T)
returns true if T is an Euler tour in the directed graph
G. The running time is $O(n + m)$.
13.13 Algorithms for Planar Graphs (plane_graph_alg)

**node**

```cpp
ST_NUMBERING(const graph& G, node_array<int>& stnum,
              list<node>& stlist, edge e_st = nil)
```

ST_NUMBERING computes an st-numbering of G. If e_st is nil then t is set to some arbitrary node of G. The node s is set to a neighbor of t and is returned. If e_st is not nil then s is set to the source of e_st and t is set to its target. The nodes of G are numbered such that s has number 1, t has number n, and every node v different from s and t has a smaller and a larger numbered neighbor. The ordered list of nodes is returned in stlist. If G has no nodes then nil is returned and if G has exactly one node then this node is returned and given number one.

**Precondition:** G is biconnected.

**bool**

```cpp
PLANAR(graph&, bool embed = false)
```

PLANAR takes as input a directed graph G(V, E) and performs a planarity test for it. G must not contain selfloops. If the second argument embed has value true and G is a planar graph it is transformed into a planar map (a combinatorial embedding such that the edges in all adjacency lists are in clockwise ordering). PLANAR returns true if G is planar and false otherwise. The algorithm ([45]) has running time \( O(|V| + |E|) \).

```cpp
PLANAR(graph& G, list<edge>& el, bool embed = false)
```

PLANAR takes as input a directed graph G(V, E) and performs a planarity test for G. PLANAR returns true if G is planar and false otherwise. If G is not planar a Kuratowsky-Subgraph is computed and returned in el.

```cpp
CHECK_KURATOWSKI(const graph& G, const list<edge>& el)
```

returns true if all edges in el are edges of G and if the edges in el form a Kuratowski subgraph of G, returns false otherwise. Writes diagnostic output to cerr.
KURATOWSKI computes a Kuratowski subdivision $K$ of $G$ as follows. $V$ is the list of all nodes and subdivision points of $K$. For all $v \in V$ the degree $\text{deg}(v)$ is equal to 2 for subdivision points, 4 for all other nodes if $K$ is a $K_5$, and -3 (+3) for the nodes of the left (right) side if $K$ is a $K_{3,3}$. $E$ is the list of all edges in the Kuratowski subdivision.

TRIANGULATE_PLANAR_MAP takes a directed graph $G$ representing a planar map. It triangulates the faces of $G$ by inserting additional edges. The list of inserted edges is returned. 

Precondition: $G$ must be connected.

The algorithm ([47]) has running time $O(|V| + |E|)$.

FIVE_COLOR colors the nodes of $G$ using 5 colors, more precisely, computes for every node $v$ a color $C[v] \in \{0, \ldots, 4\}$, such that $C[\text{source}(e)] \neq C[\text{target}(e)]$ for every edge $e$. Precondition: $G$ is planar. Remark: works also for many (sparse?) non-planar graph.

INDEPENDENT_SET determines an independent set of nodes $I$ in $G$. Every node in $I$ has degree at most 9. If $G$ is planar and has no parallel edges then $I$ contains at least $n/6$ nodes.

Is_CCW_Ordered checks whether the cyclic adjacency list of any node $v$ agrees with the counter-clockwise ordering of the neighbors of $v$ around $v$ defined by their geometric positions.

SORT_EDGES reorders all adjacency lists such the cyclic adjacency list of any node $v$ agrees with the counter-clockwise order of $v$'s neighbors around $v$ defined by their geometric positions. The function returns true if $G$ is a plane map after the call.
bool Is_CCW_Ordered(const graph& G, const edge_array<int>& dx, const edge_array<int>& dy)
    checks whether the cyclic adjacency list of any node v agrees with the counter-clockwise ordering of the neighbors of v around v. The direction of edge e is given by the vector (dx(e), dy(e)).

bool SORT_EDGES(graph& G, const edge_array<int>& dx, const edge_array<int>& dy)
    reorders all adjacency lists such the cyclic adjacency list of any node v agrees with the counter-clockwise order of v’s neighbors around v. The direction of edge e is given by the vector (dx(e), dy(e)). The function returns true if G is a plane map after the call.
This section gives a summary of the graph drawing algorithms contained in LEDA. Before using them the header file `<LEDA/graph/graph_draw.h>` has to be included.

```c
int STRAIGHT_LINE_EMBED_MAP(graph& G, node_array<int>& xcoord, node_array<int>& ycoord)

STRAIGHT_LINE_EMBED_MAP takes as argument a graph $G$ representing a planar map. It computes a straight line embedding of $G$ by assigning non-negative integer coordinates ($xcoord$ and $ycoord$) in the range $0..2(n−1)$ to the nodes. STRAIGHT_LINE_EMBED_MAP returns the maximal coordinate. The algorithm ([31]) has running time $O(|V|^2)$.
```

```c
int STRAIGHT_LINE_EMBEDDING(graph& G, node_array<int>& xc, node_array<int>& yc)

STRAIGHT_LINE_EMBEDDING takes as argument a planar graph $G$ and computes a straight line embedding of $G$ by assigning non-negative integer coordinates ($xcoord$ and $ycoord$) in the range $0..2(n−1)$ to the nodes. The algorithm returns the maximal coordinate and has running time $O(|V|^2)$.
```

```c
bool VISIBILITY_REPRESENTATION(graph& G, node_array<double>& x_pos, node_array<double>& y_pos, node_array<double>& x_rad, node_array<double>& y_rad, edge_array<double>& x_sanch, edge_array<double>& y_sanch, edge_array<double>& x_tanch, edge_array<double>& y_tanch)

computes a visibility representation of the graph $G$, i.e., each node is represented by a horizontal segment (or box) and each edge is represented by a vertical segment.

**Precondition**: $G$ must be planar and has to contain at least three nodes.
bool TUTTE_EMBEDDING(const graph& G, const list<node>& fixed_nodes, node_array<double>& xpos, node_array<double>& ypos)
computes a convex drawing of the graph G if possible. The list fixed_nodes contains nodes with prescribed coordinates already given in xpos and ypos. The computed node positions of the other nodes are stored in xpos and ypos, too. If the operation is successful, true is returned.

void SPRING_EMBEDDING(const graph& G, node_array<double>& xpos, node_array<double>& ypos, double xleft, double xright, double ybottom, double ytop, int iterations = 250)
computes a straight-line spring embedding of G in the given rectangular region. The coordinates of the computed node positions are returned in xpos and ypos.

void SPRING_EMBEDDING(const graph& G, const list<node>& fixed, node_array<double>& xpos, node_array<double>& ypos, double xleft, double xright, double ybottom, double ytop, int iterations = 250)
as above, however, the positions of all nodes in the fixed list is not changed.

void D3_SPRING_EMBEDDING(const graph& G, node_array<double>& xpos, node_array<double>& ypos, node_array<double>& zpos, double xmin, double xmax, double ymin, double ymax, double zmin, double zmax, int iterations = 250)
computes a straight-line spring embedding of G in the 3-dimensional space. The coordinates of the computed node positions are returned in xpos, ypos, and zpos.

int ORTHO_EMBEDDING(const graph& G, const edge_array<int>& maxbends, node_array<int>& xcoord, node_array<int>& ycoord, edge_array<list<int>>& xbends, edge_array<list<int>>& ybends)
Produces an orthogonal (Tamassia) embedding such that each edge e has at most maxbends[e] bends. Returns true if such an embedding exists and false otherwise. Precondition: G must be a planar 4-graph.
int ORTHO_EMBEDDING(const graph& G, node_array<int>& xpos,
                      node_array<int>& ypos, edge_array<list<int>>
                      >& xbends, edge_array<list<int>>& ybends)
    as above, but with unbounded number of edge bends.

bool ORTHO_DRAW(const graph& G0, node_array<double>& xpos,
                 node_array<double>& ypos, node_array<double>& xrad,
                 node_array<double>& yrad, edge_array<list<double>>
                 >& xbends, edge_array<list<double>>& ybends,
                 edge_array<double>& xsanch,
                 edge_array<double>& ysanch,
                 edge_array<double>& xtanch,
                 edge_array<double>& ytanch)
    computes a orthogonal drawing of an arbitrary planar
    graph (nodes of degree larger than 4 are allowed) in the
    so-called Giotto-Model, i.e. high-degree vertices (of
    degree greater than 4) will be represented by larger
    rectangles.

bool SP_EMBEDDING(graph& G, node_array<double>& x_coord,
                  node_array<double>& y_coord,
                  node_array<double>& x_radius,
                  node_array<double>& y_radius, edge_array<list<double>>
                  >& x_bends, edge_array<list<double>>& y_bends,
                  edge_array<double>& xsanch,
                  edge_array<double>& ysanch,
                  edge_array<double>& xtanch,
                  edge_array<double>& ytanch)
    computes a series-parallel drawing of G.
    
    Precondition: G must be a series-parallel graph.
13.15 Graph Morphism Algorithms (graph_morphism)

1. Definition

An instance alg of the parameterized data type graph_morphism<graph_t, impl> is an algorithm object that supports finding graph isomorphisms, subgraph isomorphisms, graph monomorphisms and graph automorphisms. The first parameter type parametrizes the input graphs’ types. It defaults to graph. The second parameter type determines the actual algorithm implementation to use. There are two implementations available so far which work differently well for certain types of graphs. More details can be found in the report Graph Isomorphism Implementation for LEDA by Johannes Singler. It is available from our homepage. You can also contact our support team to get it: support@algorithmic-solutions.com resp. support@quappa.com.

#include <LEDA/graph/graph_morphism.h>

2. Implementation

Allowed implementations parameters are vf2<graph_t> and conauto<graph_t, ord_t>.

3. Example

#include <LEDA/graph/graph_morphism.h>

// declare the input graphs.
graph g1, g2;

// In order to use node compatibility, declare associated node maps for the
// attributes and a corresponding node compatibility function
// (exemplary, see above for the definition of identity_compatibility).
node_map<int> nm1(g1), nm2(g2);
identity_compatibility<int> ic(nm1, nm2);

// do something useful to build up the graphs and the attributes

// instantiate the algorithm object
graph_morphism<graph, conauto<graph>> alg;

// declare the node and edge mapping arrays
node_array<node> node_mapping(g2);
edge_array<edge> edge_mapping(g2);

// prepare a graph morphism data structure for the first graph.
13.16. GRAPH MORPHISM ALGORITHM FUNCTIONALITY (GRAPH_MORPHISM_ALGORITHM)

```cpp
graph_morphism_algorithm<>::prep_graph pg1 = alg.prepare_graph(g1, ic);

// find the graph isomorphism.
bool isomorphic = alg.find_iso(pg1, g2, &node_mapping, &edge_mapping, ic);

// delete the prepared graph data structure again.
alg.delete_prepared_graph(pg1);
```

Please see `demo/graph_iso/gw_isomorphism.cpp` for an interactive demo program.

### 13.16 Graph Morphism Algorithm Functionality (graph_morphism_algorithm)

#### 1. Types

```cpp
#include <LEDA/graph/graph_morphism_algorithm.h>

graph_morphism_algorithm< graph_t >::node
    the type of an input graph node

graph_morphism_algorithm< graph_t >::edge
    the type of an input graph edge

graph_morphism_algorithm< graph_t >::node_morphism
    the type for a found node mapping

graph_morphism_algorithm< graph_t >::edge_morphism
    the type for a found edge mapping

graph_morphism_algorithm< graph_t >::node_compat
    the type for a node compatibility functor

graph_morphism_algorithm< graph_t >::edge_compat
    the type for an edge compatibility functor

graph_morphism_algorithm< graph_t >::morphism
    the type for a found node and edge mapping

graph_morphism_algorithm< graph_t >::morphism_list
    the type of a list of all found morphisms

graph_morphism_algorithm< graph_t >::callback
    the type for the callback functor

graph_morphism_algorithm< graph_t >::cardinality_t
    the number type of the returned cardinality
```
2. Operations

\texttt{prep\_graph} \hspace{1em} \texttt{alg\_prepare\_graph(const graph\_t\& g, const node\_compat\& node\_comp = DEFAULT\_NODE\_CMP, const edge\_compat\& edge\_comp = DEFAULT\_EDGE\_CMP)}

prepares a data structures of a graph to be used as input to subsequent morphism search calls. This may speed up computation if the same graph is used several times.

\texttt{void} \hspace{1em} \texttt{alg\_delete\_prepared\_graph(prep\_graph pg)}

frees the memory allocated to a prepared graph data structure constructed before.

\texttt{cardinality\_t} \hspace{1em} \texttt{alg\_get\_num\_calls(\)}

return the number of recursive calls the algorithm has made so far.

\texttt{void} \hspace{1em} \texttt{alg\_reset\_num\_calls(\)}

resets the number of recursive calls to 0.

\texttt{bool} \hspace{1em} \texttt{alg\_find\_iso(const graph\_t\& g1, const graph\_t\& g2, node\_morphism\* node\_morph = NULL, edge\_morphism\* edge\_morph = NULL, const node\_compat\& node\_comp = DEFAULT\_NODE\_CMP, const edge\_compat\& edge\_comp = DEFAULT\_EDGE\_CMP)}

searches for a graph isomorphism between \texttt{g1} and \texttt{g2} and returns it through \texttt{node\_morph} and \texttt{edge\_morph} if a non-NULL pointer to a node map and a non-NULL pointer to an edge map are passed respectively. Those must be initialized to \texttt{g2} and will therefore carry references to the mapped node or edge in \texttt{g1}. The possible mappings can be restricted by the node and edge compatibility functors \texttt{node\_comp} and \texttt{edge\_comp}. This method can be called with prepared graph data structures as input for either graph, too.
13.16. GRAPH MORPHISM ALGORITHM FUNCTIONALITY (GRAPH_MORPHISM_ALGORITHM)

`cardinality_t alg.cardinality_iso(const graph_t& g1, const graph_t& g2,
const node_compat& _node_comp = DEFAULT_NODE_CMP,
const edge_compat& _edge_comp = DEFAULT_EDGE_CMP)`

searches for a graph isomorphism between `g1` and `g2` and returns its cardinality. The possible mappings can be restricted by the node and edge compatibility functors `node_comp` and `edge_comp`. This method can be called with prepared graph data structures as input for either graph, too.

`cardinality_t alg.find_all_iso(const graph_t& g1, const graph_t& g2,
list<morphism*>& _isomorphisms,
const node_compat& _node_comp = DEFAULT_NODE_CMP,
const edge_compat& _edge_comp = DEFAULT_EDGE_CMP)`

searches for all graph isomorphisms between `g1` and `g2` and returns them through `_isomorphisms`. The possible mappings can be restricted by the node and edge compatibility functors `node_comp` and `edge_comp`. This method can be called with prepared graph data structures as input for either graph, too.

`cardinality_t alg.enumerate_iso(const graph_t& g1, const graph_t& g2,
leda_callback_base<morphism>& _callback,
const node_compat& _node_comp =
DEFAULT_NODE_CMP,
const edge_compat& _edge_comp =
DEFAULT_EDGE_CMP)`

searches for all graph isomorphisms between `g1` and `g2` and calls the callback functor `callback` for each one. The possible mappings can be restricted by the node and edge compatibility functors `node_comp` and `edge_comp`. This method can be called with prepared graph data structures as input for either graph, too.
bool alg.find_sub(const graph_t& g1, const graph_t& g2,
    node_morphism *node_morph = NULL,
    edge_morphism *edge_morph = NULL,
    const node_compat_t& node_comp = DEFAULT_NODE_CMP,
    const edge_compat_t& edge_comp = DEFAULT_EDGE_CMP)
searches for a subgraph isomorphism from g2 to g1
and returns it through node_morph and edge_morph if
a non-NULL pointer to a node map and a non-NULL
pointer to an edge map are passed respectively. Those
must be initialized to g2 and will therefore carry re-
ferences to the mapped node or edge in g1. g2 must
not have more nodes or more edges than g1 to make
a mapping possible. The possible mappings can be
restricted by the node and edge compatibility func-
tors node_comp and edge_comp. This method can be
called with prepared graph data structures as input
for either graph, too.

cardinality_t alg.cardinality_sub(const graph_t& g1, const graph_t& g2,
    const node_compat_t& node_comp = DEFAULT_NODE_CMP,
    const edge_compat_t& edge_comp = DEFAULT_EDGE_CMP)
searches for a subgraph isomorphism from g2 to g1
and returns its cardinality. g2 must not have more
nodes or more edges than g1 to make a mapping pos-
sible. The possible mappings can be restricted by the
node and edge compatibility functors node_comp and
e edge_comp. This method can be called with prepared
graph data structures as input for either graph, too.

cardinality_t alg.find_all_sub(const graph_t& g1, const graph_t& g2,
    list<morphism*>& _isomorphisms,
    const node_compat_t& node_comp = DEFAULT_NODE_CMP,
    const edge_compat_t& edge_comp = DEFAULT_EDGE_CMP)
searches for all subgraph isomorphisms from g2 to g1
and returns them through _isomorphisms. g2 must
not have more nodes or more edges than g1 to make
a mapping possible. The possible mappings can be
restricted by the node and edge compatibility func-
tors node_comp and edge_comp. This method can be
called with prepared graph data structures as input
for either graph, too.
13.16. GRAPH MORPHISM ALGORITHM FUNCTIONALITY (GRAPH_MORPHISM_ALGORITHM)

cardinality_t alg.enumerate_sub(const graph_t& g1, const graph_t& g2,
    leda_callback_base<morphism>& _callback,
    const node_compat& _node_comp = DEFAULT_NODE_CMP,
    const edge_compat& _edge_comp = DEFAULT_EDGE_CMP)

searches for all subgraph isomorphisms from g2 to g1 and calls the callback functor callb for each one. g2 must not have more nodes or more edges than g1 to make a mapping possible. The possible mappings can be restricted by the node and edge compatibility functors node_comp and edge_comp. This method can be called with prepared graph data structures as input for either graph, too.

bool alg.find_mono(const graph_t& g1, const graph_t& g2,
    node_morphism* _node_morph = NULL,
    edge_morphism* _edge_morph = NULL,
    const node_compat& _node_comp = DEFAULT_NODE_CMP,
    const edge_compat& _edge_comp = DEFAULT_EDGE_CMP)

searches for a graph monomorphism from g2 to g1 and returns it through node_morph and edge_morph if a non-NULL pointer to a node map and a non-NULL pointer to an edge map are passed respectively. Those must be initialized to g2 and will therefore carry references to the mapped node or edge in g1. g2 must not have more nodes or more edges than g1 to make a mapping possible. The possible mappings can be restricted by the node and edge compatibility functors node_comp and edge_comp. This method can be called with prepared graph data structures as input for either graph, too.

cardinality_t alg.cardinality_mono(const graph_t& g1, const graph_t& g2,
    const node_compat& _node_comp = DEFAULT_NODE_CMP,
    const edge_compat& _edge_comp = DEFAULT_EDGE_CMP)

searches for a graph monomorphism from g2 to g1 and returns its cardinality. g2 must not have more nodes or more edges than g1 to make a mapping possible. The possible mappings can be restricted by the node and edge compatibility functors node_comp and edge_comp. This method can be called with prepared graph data structures as input for either graph, too.
cardinality_t alg.find_all_monomorphisms(const graph_t& g1, const graph_t& g2, list<morphism*>& _isomorphisms, const node_compat& _node_comp = DEFAULT_NODE_CMP, const edge_compat& _edge_comp = DEFAULT_EDGE_CMP)

searches for all graph monomorphisms from g2 to g1 and returns them through _isomorphisms. g2 must not have more nodes or more edges than g1 to make a mapping possible. The possible mappings can be restricted by the node and edge compatibility functors node_comp and edge_comp. This method can be called with prepared graph data structures as input for either graph, too.

cardinality_t alg.enumerate_monomorphisms(const graph_t& g1, const graph_t& g2, leda_callback_base<morphism>& _callback, const node_compat& _node_comp = DEFAULT_NODE_CMP, const edge_compat& _edge_comp = DEFAULT_EDGE_CMP)

searches for all graph monomorphisms from g2 to g1 and calls the callback functor callb for each one. g2 must not have more nodes or more edges than g1 to make a mapping possible. The possible mappings can be restricted by the node and edge compatibility functors node_comp and edge_comp. This method can be called with prepared graph data structures as input for either graph, too.

bool alg.is_graph_isomorphism(const graph_t& g1, const graph_t& g2, node_morphism const* node_morph, edge_morphism const* edge_morph = NULL, const node_compat& _node_comp = DEFAULT_NODE_CMP, const edge_compat& _edge_comp = DEFAULT_EDGE_CMP)

checks whether the morphism given by node_morph and edge_morph (optional) is a valid graph isomorphisms between g1 and g2. The allowed mappings can be restricted by the node and edge compatibility functors node_comp and edge_comp.
bool alg.is_subgraph_isomorphism(const graph_t& g1, const graph_t& g2, 
    node_morphism const * node_morph, 
    edge_morphism const * edge_morph = NULL, 
    const node_compat& node_comp = DEFAULT_NODE_CMP, 
    const edge_compat& edge_comp = DEFAULT_EDGE_CMP)

checks whether the morphism given by node_morph and edge_morph (optional) is a valid subgraph isomorphisms from g2 to g2. The allowed mappings can be restricted by the node and edge compatibility functors node_comp and edge_comp.

bool alg.is_graph_monomorphism(const graph_t& g1, const graph_t& g2, 
    node_morphism const * node_morph, 
    edge_morphism const * edge_morph = NULL, 
    const node_compat& node_comp = DEFAULT_NODE_CMP, 
    const edge_compat& edge_comp = DEFAULT_EDGE_CMP)

checks whether the morphism given by node_morph and edge_morph (optional) is a valid graph monomorphisms from g2 to g1. The allowed mappings can be restricted by the node and edge compatibility functors node_comp and edge_comp.
Chapter 14

Graphs and Iterators

14.1 Introduction

14.1.1 Iterators

Iterators are a powerful technique in object-oriented programming and one of the fundamental design patterns [39]. Roughly speaking, an iterator is a small, light-weight object, which is associated with a specific kind of linear sequence. An iterator can be used to access all items in a linear sequence step-by-step. In this section, different iterator classes are introduced for traversing the nodes and the edges of a graph, and for traversing all ingoing and/or outgoing edges of a single node.

Iterators are an alternative to the iteration macros introduced in sect. 12.1.3.(i). For example, consider the following iteration pattern:

```plaintext
node v;
forall_nodes (n, G) { ... }
```

Using the class NodeIt introduced in sect. 14.2, this iteration can be re-written as follows:

```plaintext
for (NodeIt it (G); it.valid(); ++it) { ... }
```

The crucial differences are:

- Iterators provide an intuitive means of movement through the topology of a graph.
• Iterators are not bound to a loop, which means that the user has finer control over
the iteration process. For example, the continuation condition \textit{it.valid()} in the above
loop could be replaced by another condition to terminate the loop once a specific
node has been found (and the loop may be re-started at the same position later on).

• The meaning of iteration may be modified seamlessly. For example, the filter iter-
ators defined in sect. 14.9 restrict the iteration to a subset that is specified by an
arbitrary logical condition (\textit{predicate}). In other words, the nodes or edges that do
not fulfill this predicate are filtered out automatically during iteration.

• The functionality of iteration may be extended seamlessly. For example, the observer
iterators defined in sect. 14.11 can be used to record details of the iteration. A
concrete example is given in sect. 14.11: an observer iterator can be initialized such
that it records the number of iterations performed by the iterator.

• Iterator-based implementations of algorithms can be easily integrated into environ-
ments that are implemented according to the STL style \cite{69}, (this style has been
adopted for the standard C++ library). For this purpose, sect. 14.12 define adapters,
which convert graph iterators into STL iterators.

14.1.2 Handles and Iterators

Iterators can be used whenever the corresponding handle can be used. For example, node
iterators can be used where a node is requested or edge iterators can be used where an
edge is requested. For adjacency iterators, it is possible to use them whenever an edge is
requested\footnote{Since the edge of an adjacency iterator changes while the fixed node remains fixed, we decided to
focus on the edge.}.

An example shows how iterators can be used as handles:

\begin{verbatim}
NodeIt it(G);
leda::node_array<int> index(G);
leda::node v;
int i=0;
forall_nodes(v,G) index[v]=++i;
while (it.valid()) {
    cout << "current node " << index(it) << endl; }
\end{verbatim}

14.1.3 STL Iterators

Those who are more used to STL may take advantage from the following iterator classes:
\texttt{NodeIt\_n}, \texttt{EdgeIt\_e}, \texttt{AdjIt\_n}, \texttt{AdjIt\_e}, \texttt{OutAdjIt\_n}, \texttt{OutAdjIt\_e}, \texttt{InAdjIt\_n}, \texttt{InAdjIt\_e}.
The purpose of each iterator is the same as in the corresponding standard iterator classes `NodeIt, EdgeIt`... The difference is the interface, which is exactly that of the STL iterator wrapper classe (see sect. 14.12 for more information).

An example shows why these classes are useful (remember the example from the beginning):

```cpp
NodeIt_n base(G);
for(NodeIt_n::iterator it=base.begin(); it!=base.end(); ++it) {
    cout << "current node " << index(*it) << endl;
}
```

As in STL collections there are public type definitions in all STL style graph iterators. The advantage is that algorithms can be written that operate independently of the underlying type (note: NodeIt_n and NodeIt_n::iterator are equal types).

### 14.1.4 Circulators

Circulators differ from Iterators in their semantics. Instead of becoming invalid at the end of a sequence, they perform cyclic iteration. This type of "none-ending-iterator" is heavily used in the CGAL.

### 14.1.5 Data Accessors

Data accessor is a design pattern[71] that decouples data access from underlying implementation. Here, the pattern is used to decouple data access in graph algorithms from how data is actually stored outside the algorithm.

Generally, an attributed graph consists of a (directed or undirected) graph and an arbitrary number of node and edge attributes. For example, the nodes of a graph are often assigned attributes such as names, flags, and coordinates, and likewise, the edges are assigned attributes such as lengths, costs, and capacities.

More formally, an attribute $a$ of a set $S$ has a certain type $T$ and assigns a value of $T$ to every element of $S$ (in other words, $a$ may be viewed as a function $a : S \rightarrow T$). An attributed set $A = (S, a_1, \ldots, a_m)$ consists of a set $S$ and attributes $a_1, \ldots, a_m$. An attributed graph is a (directed or undirected) graph $G = (V, E)$ such that the node set $V$ and the edge set $E$ are attributed.

Basically, LEDA provides two features to define attributes for graph:
Classes \textit{GRAPH} and \textit{UGRAPH} (sects. 12.2 and 12.5) are templates with two arguments, \textit{vtype} and \textit{etype}, which are reserved for a node and an edge attribute, respectively. To attach several attributes to nodes and edges, \textit{vtype} and \textit{etype} must be instantiated by structs whose members are the attributes.

A \textit{node array} (sect. 12.8) or \textit{node map} (sect. Node Maps) represents a node attribute, and analogously, \textit{edge arrays} (sect. Edge Arrays) and \textit{edge maps} (sect. 12.12), represent edge attributes. Several attributes can be attached to nodes and edges by instantiating several arrays or maps.

Data accessors provide a uniform interface to access attributes, and the concrete organization of the attributes is hidden behind this interface. Hence, if an implementation of an algorithm does not access attributes directly, but solely in terms of data accessors, it may be applied to any organization of the attributes (in contrast, the algorithms in sect. Graph Algorithms require an organization of all attributes as node and edge arrays).

Every data accessor class \textit{DA} comes with a function template \texttt{get}:

\begin{verbatim}
T get(DA da, Iter it);
\end{verbatim}

This function returns the value of the attribute managed by the data accessor \texttt{da} for the node or edge marked by the iterator \texttt{it}. Moreover, most data accessor classes also come with a function template \texttt{set}:

\begin{verbatim}
void set(DA da, Iter it, T value);
\end{verbatim}

This function overwrites the value of the attribute managed by the data accessor \texttt{da} for the node or edge marked by the iterator \texttt{it} by \texttt{value}.

The data accessor classes that do not provide a function template \texttt{set} realize attributes in such a way that a function \texttt{set} does not make sense or is even impossible. The \textit{constant accessor} in sect. 14.14 is a concrete example: it realizes an attribute that is constant over the whole attributed set and over the whole time of the program. Hence, it does not make sense to provide a function \texttt{set}. Moreover, since the constant accessor class organizes its attribute in a non-materialized fashion, an overwriting function \texttt{set} is even impossible.

\textbf{Example:} The following trivial algorithm may serve as an example to demonstrate the usage of data accessors and their interplay with various iterator types. The first, nested loop accesses all edges once. More specifically, the outer loop iterates over all nodes of the graph, and the inner loop iterates over all edges leaving the current node of the outer loop. Hence, for each edge, the value of the attribute managed by the data accessor \texttt{da} is overwritten by \texttt{t}. In the second loop, a linear edge iterator is used to check whether the first loop has set all values correctly.
template <class T, class DA>
void set_and_check (graph& G, DA da, T t) {
  for (NodeIt nit(G); nit.valid(); ++nit)
    for (OutAdjIt oait(nit); oait.valid(); ++oait)
      set (da, eit, t);
  for (EdgeIt eit(G); eit.valid(); ++eit)
    if (get(da, it) != t) cout << "Error!" << endl;
}

To demonstrate the application of function set_and_check, we first consider the case that $G$ is an object of the class GRAPH derived from graph (sect. 12.1), that the template argument $vtype$ is instantiated by a struct type attributes, and that the int-member my_attr of attributes shall be processed by set_and_check with value 1. Then DA can be instantiated as a node_member_da:

```cpp
node_member_da<attributes,int> da (&attributes::my_attr);
set_and_check (G, da, 1);
```

Now we consider the case that the attribute to be processed is stored in an edge_array<int> named my_attr_array:

```cpp
node_array_da<int> da (my_attr_array);
set_and_check (G, da, 1);
```

Hence, all differences between these two cases are factored out into a single declaration statement.

### 14.1.6 Graph iterator Algorithms

Several basic graph algorithms were re-implemented to use only graph iterators and data accessors. Moreover they share three design decisions:

1. **algorithms are instances** of classes

2. algorithm instances have the **ability to “advance”**

3. algorithm instances provide **access to their internal states**
An example for an algorithm that supports the first two decisions is:

```cpp
class Algorithm {
  int state, endstate;
public:
  Algorithm(int max) : endstate(max), state(0) { }
  void next() { state++; }
  bool finished() { return state>=endstate; }
};
```

With this class `Algorithm` we can easily instantiate an algorithm object:

```cpp
Algorithm alg(5);
while (!alg.finished()) alg.next();
```

This small piece of code creates an algorithm object and invokes “next()” until it has reached an end state.

An advantage of this design is that we can write basic algorithms, which can be used in a standardized way and if needed, inspection of internal states and variables can be provided without writing complex code. Additionally, it makes it possible to write persistent algorithms, if the member variables are persistent.

Actually, those algorithms are quite more flexible than ordinary written algorithm functions:

```cpp
template<class Alg>
class OutputAlg {
  Alg alg;
public:
  OutputAlg(int m) : alg(m) {
    cout << "max state: " << m << endl;
  }
  void next() {
    cout << "old state: " << alg.state;
    alg.next();
    cout << " new state: " << alg.state << endl;
  }
  bool finished() { return alg.finished(); }
};
```

This wrapper algorithm can be used like this:

```cpp
OutputAlg<Algorithm> alg(5);
while (!alg.finished()) alg.next();
```
In addition to the algorithm mentioned earlier this wrapper writes the internal states to the standard output.

This is as efficient as rewriting the “Algorithm”-class with an output mechanism, but provides more flexibility.

14.2 Node Iterators (NodeIt)

1. Definition

A variable \( it \) of class \( NodeIt \) is a linear node iterator that iterates over the node set of a graph; the current node of an iterator object is said to be “marked” by this object.

```c
#include <LEDA/graph/graph_iterator.h>
```

2. Creation

\begin{verbatim}
NodeIt it; // introduces a variable \( it \) of this class associated with no graph.

NodeIt it(const leda::graph& G); // introduces a variable \( it \) of this class associated with \( G \).
\hspace{1cm} The graph is initialized by \( G \). The node is initialized by \( G.first_node( ) \).

NodeIt it(const leda::graph& G, leda::node n); // introduces a variable \( it \) of this class marked with \( n \) and associated with \( G \).
\hspace{1cm} Precondition: \( n \) is a node of \( G \).
\end{verbatim}

3. Operations

\begin{verbatim}
void it.init(const leda::graph& G) // associates \( it \) with \( G \) and marks it with \( G.first_node( ) \).

void it.init(const leda::graph& G, const leda::node& v) // associates \( it \) with \( G \) and marks it with \( v \).

void it.reset( ) // resets \( it \) to \( G.first_node( ) \), where \( G \) is the associated graph.

void it.make_invalid( ) // makes \( it \) invalid, i.e. \( it.valid( ) \) will be false afterwards and \( it \) marks no node.

void it.reset_end( ) // resets \( it \) to \( G.last_node( ) \), where \( G \) is the associated graph.
\end{verbatim}
void it.update(leda::node n)
    it marks n afterwards.

void it.insert()
    creates a new node and it marks it afterwards.

void it.del()
    deletes the marked node, i.e. it.valid() returns false afterwards.
    Precondition: it.valid() returns true.

NodeIt& it = const NodeIt& it2
    it is afterwards associated with the same graph and node as it2. This method returns a reference to it.

bool it == const NodeIt& it2
    returns true if and only if it and it2 are equal, i.e. if the marked nodes are equal.

leda::node it.get_node()
    returns the marked node or nil if it.valid() returns false.

const leda::graph& it.get_graph()
    returns the associated graph.

bool it.valid()
    returns true if and only if end of sequence not yet passed, i.e. if there is a node in the node set that was not yet passed.

bool it.eol()
    returns !it.valid() which is true if and only if there is no successor node left, i.e. if all nodes of the node set are passed (eol: end of list).

NodeIt& ++it
    performs one step forward in the list of nodes of the associated graph. If there is no successor node, it.eol() will be true afterwards. This method returns a reference to it.
    Precondition: it.valid() returns true.

NodeIt& --it
    performs one step backward in the list of nodes of the associated graph. If there is no predecessor node, it.eol() will be true afterwards. This method returns a reference to it.
    Precondition: it.valid() returns true.

4. Implementation

Creation of an iterator and all methods take constant time.
14.3 Edge Iterators (EdgeIt)

1. Definition

A variable \textit{it} of class \textit{EdgeIt} is a linear edge iterator that iterates over the edge set of a graph; the current edge of an iterator object is said to be “marked” by this object.

```cpp
#include <LEDA/graph/graph_iterator.h>
```

2. Creation

\textit{EdgeIt it;} introduces a variable \textit{it} of this class associated with no graph.

\textit{EdgeIt it(const leda::graph& G);} introduces a variable \textit{it} of this class associated with \textit{G} and marked with \textit{G.first_edge().}

\textit{EdgeIt it(const leda::graph& G, leda::edge e);} introduces a variable \textit{it} of this class marked with \textit{e} and associated with \textit{G}.

Precondition: \textit{e} is an edge of \textit{G}.

3. Operations

\textit{void it.init(const leda::graph& G)} associates \textit{it} with \textit{G} and marks it with \textit{G.first_edge().}

\textit{void it.init(const leda::graph& G, const leda::edge& e)} associates \textit{it} with \textit{G} and marks it with \textit{e}.

\textit{void it.update(leda::edge e)} it marks \textit{e} afterwards.

\textit{void it.reset()} resets \textit{it} to \textit{G.first_edge()} where \textit{G} is the associated graph.

\textit{void it.reset_end()} resets \textit{it} to \textit{G.last_edge()} where \textit{G} is the associated graph.

\textit{void it.make_invalid()} makes \textit{it} invalid, i.e. \textit{it.valid()} will be false afterwards and \textit{it} marks no node.

\textit{void it.insert(leda::node v1, leda::node v2)} creates a new edge from \textit{v1} to \textit{v2} and \textit{it} marks it afterwards.

\textit{void it.del()} deletes the marked edge, i.e. \textit{it.valid()} returns false afterwards.

Precondition: \textit{it.valid()} returns true.
CHAPTER 14. GRAPHS AND ITERATORS

EdgeIt& it = const EdgeIt& it2

assigns it2 to it. This method returns a reference to it.

bool it == const EdgeIt& it2

returns true if and only if it and it2 are equal, i.e. if the marked edges are equal.

bool it.eol()

returns !it.valid() which is true if and only if there is no successor edge left, i.e. if all edges leaving the marked node are passed (eol: end of list).

bool it.valid()

returns true if and only if end of sequence not yet passed, i.e. if there is an edge leaving the marked node that was not yet passed.

leda::edge it.get_edge()

returns the marked edge or nil if it.valid() returns false.

const leda::graph& it.get_graph()

returns the associated graph.

EdgeIt& ++it

performs one step forward in the list of edges of the associated graph. If there is no successor edge, it.eol() will be true afterwards. This method returns a reference to it.

Precondition: it.valid() returns true.

EdgeIt& --it

performs one step backward in the list of edges of the associated graph. If there is no predecessor edge, it.eol() will be true afterwards. This method returns a reference to it.

Precondition: it.valid() returns true.

4. Implementation

Creation of an iterator and all methods take constant time.

14.4 Face Iterators (FaceIt)

1. Definition

a variable it of class FaceIt is a linear face iterator that iterates over the face set of a graph; the current face of an iterator object is said to be “marked” by this object.

Precondition: Before using any face iterator the list of faces has to be computed by calling G.compute_faces(). Note, that any update operation invalidates this list.
#include <LEDA/graph/graph_iterator.h>

## 2. Creation

### FaceIt it;
introduces a variable it of this class associated with no graph.

### FaceIt it(const leda::graph& G);
introduces a variable it of this class associated with G. The graph is initialized by G. The face is initialized by G.first_face().

### FaceIt it(const leda::graph& G, leda::face n);
introduces a variable it of this class marked with n and associated with G. Precondition: n is a face of G.

## 3. Operations

### void it.init(const leda::graph& G)
associates it with G and marks it with G.first_face().

### void it.init(const leda::graph& G, const leda::face& v)
associates it with G and marks it with v.

### void it.reset()
resets it to G.first_face(), where G is the associated graph.

### void it.make_invalid()
makes it invalid, i.e. it.valid() will be false afterwards and it marks no face.

### void it.reset_end()
resets it to G.last_face(), where G is the associated graph.

### void it.update(leda::face n)
it marks n afterwards.

### FaceIt& it = const FaceIt& it2
it is afterwards associated with the same graph and face as it2. This method returns a reference to it.

### bool it == const FaceIt& it2
returns true if and only if it and it2 are equal, i.e. if the marked faces are equal.

### leda::face it.get_face()
returns the marked face or nil if it.valid() returns false.

### const leda::graph& it.get_graph()
returns the associated graph.
CHAPTER 14. GRAPHS AND ITERATORS

bool it.valid() returns true if and only if end of sequence not yet passed, i.e. if there is a face in the face set that was not yet passed.

bool it.eol() returns !it.valid() which is true if and only if there is no successor face left, i.e. if all faces of the face set are passed (eol: end of list).

FaceIt& ++it performs one step forward in the list of faces of the associated graph. If there is no successor face, it.eol() will be true afterwards. This method returns a reference to it.
Precondition: it.valid() returns true.

FaceIt& --it performs one step backward in the list of faces of the associated graph. If there is no predecessor face, it.eol() will be true afterwards. This method returns a reference to it.
Precondition: it.valid() returns true.

4. Implementation

Creation of an iterator and all methods take constant time.

14.5 Adjacency Iterators for leaving edges (OutAdjIt)

1. Definition

a variable it of class OutAdjIt is an adjacency iterator that marks a node (which is fixed in contrast to linear node iterators) and iterates over the edges that leave this node.

There is a variant of the adjacency iterators, so-called circulators which are heavily used in the CGAL\(^2\). The names of the classes are OutAdjCirc and InAdjCirc and their interfaces are completely equal to the iterator versions while they internally use e.g. cyclic_adj.succ() instead of adj.succ().

#include <LEDA/graph/graph_iterator.h>

2. Creation

OutAdjIt it; introduces a variable it of this class associated with no graph.

\(^2\)See the CGAL homepage at http://www.cs.uu.nl/CGAL/.
14.5. ADJACENCY ITERATORS FOR LEAVING EDGES (OUTADJIT)

OutAdjIt it(const leda::graph& G);

introduces a variable it of this class associated with G. The node is initialized by G.first_node() and the edge by G.first_adj_edge(n) where n is the marked node.

OutAdjIt it(const leda::graph& G, leda::node n);

introduces a variable it of this class marked with n and associated with G. The marked edge is initialized by G.first_adj_edge(n).
Precondition: n is a node of G.

OutAdjIt it(const leda::graph& G, leda::node n, leda::edge e);

introduces a variable it of this class marked with n and e and associated with G.
Precondition: n is a node and e an edge of G and source(e) = n.

3. Operations

void it.init(const leda::graph& G)

associates it with G and marks it with n' = G.first_node() and G.first_adj_edge(n').

void it.init(const leda::graph& G, const leda::node& n)

associates it with G and marks it with n and G.first_adj_edge(n).
Precondition: n is a node of G.

void it.init(const leda::graph& G, const leda::node& n, const leda::edge& e)

associates it with G and marks it with n and e.
Precondition: n is a node and e an edge of G and source(e) = n.

void it.update(leda::edge e)

it marks e afterwards.

void it.reset()

resets it to G.first_adj_edge(n) where G and n are the marked node and associated graph.

void it.insert(const OutAdjIt& other)

creates a new leaving edge from the marked node of it to the marked node of other. it is marked with the new edge afterwards. The marked node of it does not change.

void it.del()

deletes the marked leaving edge, i.e. it.valid() returns false afterwards.
Precondition: it.valid() returns true.
void it.reset_end( )  
resets it to $G.\text{last}\_\text{adj}\_\text{edge}(n)$ where $G$ and $n$ are the marked node and associated graph.

void it.make_invalid( )  
makes it invalid, i.e. it.valid( ) will be false afterwards and it marks no node.

void it.update(leda::node n)  
it marks $n$ and the first leaving edge of $n$ afterwards.

void it.update(leda::node n, leda::edge e)  
it marks $n$ and $e$ afterwards.

OutAdjIt& it = const OutAdjIt& it2  
assigns it2 to it. This method returns a reference to it.

bool it == const OutAdjIt& it2  
returns true if and only if it and it2 are equal, i.e. if the marked nodes and edges are equal.

bool it.has_node( )  
returns true if and only if it marks a node.

bool it.eol( )  
returns !it.valid( ) which is true if and only if there is no successor edge left, i.e. if all edges of the edge set are passed (eol: end of list).

bool it.valid( )  
returns true if and only if end of sequence not yet passed, i.e. if there is an edge in the edge set that was not yet passed.

leda::edge it.get_edge( )  
returns the marked edge or nil if it.valid( ) returns false.

leda::node it.get_node( )  
returns the marked node or nil if it.has_node( ) returns false.

const leda::graph& it.get_graph( )  
returns the associated graph.

OutAdjIt it.curr_adj( )  
returns a new adjacency iterator that is associated with $n' = \text{target}(e)$ and $G.\text{first}\_\text{adj}\_\text{edge}(n')$ where $G$ is the associated graph.

Precondition: it.valid( ) returns true.

OutAdjIt& ++it  
performs one step forward in the list of outgoing edges of the marked node. If there is no successor edge, it.eol( ) will be true afterwards. This method returns a reference to it.

Precondition: it.valid( ) returns true.
14.6. ADJACENCY ITERATORS FOR INCOMING EDGES (INADJIT)

OutAdjIt& --it performs one step backward in the list of outgoing edges of the marked node. If there is no predecessor edge, it.eol() will be true afterwards. This method returns a reference to it.
Precondition: it.valid() returns true.

4. Implementation

Creation of an iterator and all methods take constant time.

14.6 Adjacency Iterators for incoming edges (InAdjIt)

1. Definition

a variable it of class InAdjIt is an adjacency iterator that marks a node (which is fixed in contrast to linear node iterators) and iterates over the incoming edges of this node.

#include <LEDA/graph/graph_iterator.h>

2. Creation

InAdjIt it; introduces a variable it of this class associated with no graph.

InAdjIt it(const leda::graph& G);
introduces a variable it of this class associated with G.
The node is initialized by G.first_node() and the edge by G.first_in_edge(n) where n is the marked node.

InAdjIt it(const leda::graph& G, leda::node n);
introduces a variable it of this class marked with n and associated with G. The marked edge is initialized by G.first_in_edge(n).
Precondition: n is a node of G.

InAdjIt it(const leda::graph& G, leda::node n, leda::edge e);
introduces a variable it of this class marked with n and e and associated with G.
Precondition: n is a node and e an edge of G and target(e) = n.

3. Operations

void it.init(const leda::graph& G)
associates it with G and marks it with n' = G.first_node() and G.first_adj_edge(n').
void it.init(const leda::graph& G, const leda::node& n)
        associates it with G and marks it with n and G.first_adj_edge(n).
        Precondition: n is a node of G.

void it.init(const leda::graph& G, const leda::node& n, const leda::edge& e)
        associates it with G and marks it with n and e.
        Precondition: n is a node and e an edge of G and target(e) = n.

void it.update(leda::edge e)
        it marks e afterwards.

void it.reset()
        resets it to G.first_in_edge(n) where G and n are the
        marked node and associated graph.

void it.insert(const InAdjIt& other)
        creates a new incoming edge from the marked node of
        it to the marked node of other. it is marked with the
        new edge afterwards. The marked node of it does not
        change.

void it.del()
        deletes the marked incoming edge, i.e. it.valid( ) returns false afterwards.
        Precondition: it.valid( ) returns true.

void it.reset_end()
        resets it to G.last_in_edge(n) where G and n are the
        marked node and associated graph.

void it.make_invalid()
        makes it invalid, i.e. it.valid( ) will be false afterwards
        and it marks no node.

void it.update(leda::node n)
        it marks n and the first incoming edge of n afterwards.

void it.update(leda::node n, leda::edge e)
        it marks n and e afterwards.

InAdjIt& it = const InAdjIt& it2
        assigns it2 to it. This method returns a reference this
        method returns a reference to it.

bool it == const InAdjIt& it2
        returns true if and only if it and it2 are equal, i.e. if
        the marked nodes and edges are equal.

bool it.has_node()
        returns true if and only if it marks a node.
adjacency iterators (adjit)

bool it.eol() returns \(!it.valid()\) which is true if and only if there is no successor edge left, i.e. if all edges of the edge set are passed (eol: end of list).

bool it.valid() returns true if and only if end of sequence not yet passed, i.e. if there is an edge in the edge set that was not yet passed.

leda::edge it.get_edge() returns the marked edge or nil if it.valid() returns false.

leda::node it.get_node() returns the marked node or nil if it.has_node() returns false.

const leda::graph& it.get_graph() returns the associated graph.

InAdjIt it.curr_adj() returns a new adjacency iterator that is associated with \(n' = source(e)\) and \(G.first_out_edge(n')\) where \(G\) is the associated graph.

Precondition: it.valid() returns true.

InAdjIt& ++it performs one step forward in the list of incoming edges of the marked node. If there is no successor edge, it.eol() will be true afterwards. This method returns a reference to it.

Precondition: it.valid() returns true.

InAdjIt& --it performs one step backward in the list of incoming edges of the marked node. If there is no predecessor edge, it.eol() will be true afterwards. This method returns a reference to it.

Precondition: it.valid() returns true.

4. Implementation

Creation of an iterator and all methods take constant time.

14.7 adjacency iterators (adjit)

1. Definition

a variable it of class AdjIt is an adjacency iterator that marks a node (which is fixed in contrast to linear node iterators) and iterates over the edges that leave or enter this node. At first, all outgoing edges will be traversed.

Internally, this iterator creates two instances of OutAdjIt and InAdjIt. The iteration is a sequenced iteration over both iterators. Note that this only fits for directed graph, for undirected graph you should use OutAdjIt instead.
#include <LEDA/graph/graph_iterator.h>

2. Creation

AdjIt it;
introduces a variable it of this class associated with no graph.

AdjIt it(const leda::graph & G);
introduces a variable it of this class associated with G. The
marked node is initialized by \( n = G\text{.first_node}() \) and the edge
by \( G\text{.first_adj_edge}(n) \).

AdjIt it(const leda::graph & G, leda::node n);
introduces a variable it of this class marked with n and associated
with G. The marked edge is initialized by \( G\text{.first_adj_edge}(n) \).
Precondition: n is a node of G.

AdjIt it(const leda::graph & G, leda::node n, leda::edge e);
introduces a variable it of this class marked with n and e and as-
sociated with G.
Precondition: n is a node and e an edge of G and \( \text{source}(e) = n \).

3. Operations

void it.init(const graphtype & G)
associates it with G and marks it with \( n' = G\text{.first_node}() \) and \( G\text{.first_adj_edge}(n') \).

void it.init(const graphtype & G, const nodetype & n)
associates it with G and marks it with n and
\( G\text{.first_adj_edge}(v) \).
Precondition: n is a node of G.

void it.init(const graphtype & G, const nodetype & n, const edgetype & e)
associates it with G and marks it with n and e.
Precondition: n is a node and e an edge of G and
\( \text{source}(e) = n \).

void it.update(leda::edge e)
it marks e afterwards.

void it.reset()
resets it to \( G\text{.first_adj_edge}(n) \) where G and n are the
marked node and associated graph.

void it.insert(const AdjIt & other)
creates a new edge from the marked node of it to the
marked node of other. it is marked with the new edge
afterwards. The marked node of it does not change.
14.7. ADJACENCY ITERATORS (ADJIT)

```c
void it.del()  
    deletes the marked leaving edge, i.e. it.valid() returns false afterwards.
    Precondition: it.valid() returns true.

void it.reset_end()  
    resets it to G.last_adj_edge(n) where G and n are the marked node and associated graph.

void it.make_invalid()  
    makes it invalid, i.e. it.valid() will be false afterwards and it marks no node.

void it.update(leda::node n)  
    it marks n and the first leaving edge of n afterwards.

void it.update(leda::node n, leda::edge e)  
    it marks n and e afterwards.

AdjIt& it = const AdjIt& it2  
    assigns it2 to it. This method returns a reference to it.

bool it == const AdjIt& it2  
    returns true if and only if it and it2 are equal, i.e. if the marked nodes and edges are equal.

bool it.has_node()  
    returns true if and only if it marks a node.

bool it.eol()  
    returns !it.valid() which is true if and only if there is no successor edge left, i.e. if all edges of the edge set are passed (eol: end of list).

bool it.valid()  
    returns true if and only if end of sequence not yet passed, i.e. if there is an edge in the edge set that was not yet passed.

leda::edge it.get_edge()  
    returns the marked edge or nil if it.valid() returns false.

leda::node it.get_node()  
    returns the marked node or nil if it.has_node() returns false.

const leda::graph& it.get_graph()  
    returns the associated graph.

AdjIt it.curr_adj()  
    If the currently associated edge leaves the marked node, this method returns a new adjacency iterator that is associated with n’ = target(e) and G.first_adj_edge(n’) where G is the associated graph. Otherwise it returns a new adjacency iterator that is associated with n’ = source(e) and G.first_in_edge(n’) where G is the associated graph.
    Precondition: it.valid() returns true.
```
CHAPTER 14. GRAPHS AND ITERATORS

AdjIt& ++it performs one step forward in the list of incident edges of the marked node. If the formerly marked edge was a leaving edge and there is no successor edge, it is associated to $G.first\_in\_edge(n)$ where $G$ and $n$ are the associated graph and node. If the formerly marked edge was an incoming edge and there is no successor edge, $it.eol()$ will be true afterwards. This method returns a reference to $it$.
Precondition: $it.valid()$ returns true.

AdjIt& --it performs one step backward in the list of incident edges of the marked node. If the formerly marked edge was an incoming edge and there is no predecessor edge, it is associated to $G.last\_adj\_edge(n)$ where $G$ and $n$ are the associated graph and node. If the formerly marked edge was a leaving edge and there is no successor edge, $it.eol()$ will be true afterwards. This method returns a reference to $it$.
Precondition: $it.valid()$ returns true.

4. Implementation

Creation of an iterator and all methods take constant time.

14.8 Face Circulators (FaceCirc)

1. Definition

a variable $fc$ of class FaceCirc is a face circulator that circulates through the set of edges of a face as long as the graph is embedded combinatorically correct, i.e. the graph has to be bidirected and a map (see 12.1).

```cpp
#include <LEDA/graph/graph_iterator.h>
```

2. Creation

`FaceCirc fc;` introduces a variable $fc$ of this class associated with no graph.

`FaceCirc fc(const leda::graph& G);` introduces a variable $fc$ of this class associated with $G$. The edge is initialized to nil.

`FaceCirc fc(const leda::graph& G, leda::edge e);` introduces a variable $fc$ of this class marked with $e$ and associated with $G$.
Precondition: $e$ is an edge of $G$. 
3. Operations

```cpp
void fc.init(const leda::graph& G)  
associates fc with G.

void fc.init(const leda::graph& G, const leda::edge& e)  
associates fc with G and marks it with e.  
Precondition: e is an edge of G.

void fc.update(leda::edge e)  
fc marks e afterwards.

void fc.make_invalid()  
makes fc invalid, i.e. fc.valid() will be false afterwards  
and fc marks no edge.

FaceCirc& fc = const FaceCirc& fc2  
assigns fc2 to fc. This method returns a reference to fc.

bool fc == const FaceCirc& fc2  
returns true if and only if fc and fc2 are equal, i.e. if  
the marked edges are equal.

bool fc.has_edge()  
returns true if and only if fc marks an edge.

bool fc.eol()  
returns !fc.valid().

bool fc.valid()  
returns true if and only if the circulator is marked  
with an edge.

leda::edge fc.get_edge()  
returns the marked edge or nil if fc.valid() returns  
false.

const leda::graph& fc.get_graph()  
returns the associated graph.

FaceCirc& ++fc  
redirects the circulator to the cyclic adjacency prede-  
cessor of reversal(e), where e is the marked edge. This  
method returns a reference to fc.  
Precondition: fc.valid() returns true.

FaceCirc& --fc  
redirects the circulator to the cyclic adjacency succes-  
sor of e, where e is the marked edge. This method  
returns a reference to fc.  
Precondition: fc.valid() returns true.
```

4. Implementation

Creation of a circulator and all methods take constant time.
14.9 Filter Node Iterator (FilterNodeIt)

1. Definition

An instance $it$ of class $\text{FilterNodeIt}<\text{Predicate},\text{Iter}>$ encapsulates an object of type $\text{Iter}$ and creates a restricted view on the set of nodes over which this internal iterator iterates. More specifically, all nodes that do not fulfill the predicate defined by $\text{Predicate}$ are filtered out during this traversal.

Class $\text{FilterEdgeIt}$ and $\text{FilterAdjIt}$ are defined analogously, i.e. can be used for edge iterators or adjacency iterators, respectively.

Precondition: The template parameter $\text{Iter}$ must be a node iterator, e.g. $\text{NodeIt}$ or $\text{FilterNodeIt}<\text{pred},\text{NodeIt}>$. $\text{Predicate}$ must be a class which provides a method $\text{operator()}$ according to the following signature: $\text{bool operator()}$ ($\text{Iter}$).

# include <LEDA/graph/graph_iterator.h>

2. Creation

$\text{FilterNodeIt}<\text{Predicate},\text{Iter}> it;
\text{introduces a variable } it \text{ of this class, not bound to a predicate or iterator.}$

$\text{FilterNodeIt}<\text{Predicate},\text{Iter}> it(const\text{Predicate}& pred, const\text{Iter}& base_it);
\text{introduces a variable } it \text{ of this class bound to } pred \text{ and } base_it.$

3. Operations

$\text{void } it.$init($\text{const Predicate}& pred,\text{const Iter}& base_it)$
\text{initializes } it, \text{ which is bound to } pred \text{ and } base_it \text{ afterwards.}$
\text{Precondition: } it \text{ is not yet bound to a predicate or iterator.}$

4. Implementation

Constant overhead.

5. Example

Suppose each node has an own colour and we only want to see those with a specific colour, for example red (we use the LEDA colours). At first the data structures:

GRAPH<color,double> G;
NodeIt it(G);
We would have to write something like this:

```cpp
while(it.valid()) {
    if (G[it.get_node()]==red) do_something(it);
    ++it;
}
```

With the filter wrapper class we can add the test if the node is red to the behaviour of the iterator.

```cpp
struct RedPred {
    bool operator() (const NodeIt& it) const {
        return G[it.get_node()]==red;
    }
} redpred;
FilterNodeIt<RedPred,NodeIt> red_it(redpred,it);
```

This simplifies the loop to the following:

```cpp
while(red_it.valid()) {
    do_something(red_it);
    ++red_it;
}
```

All ingredients of the comparison are hard-wired in struct `RedPred`: the type of the compared values (color), the comparison value (red) and the binary comparison (equality). The following class `CompPred` renders these three choices flexible.

### 14.10 Comparison Predicate (CompPred)

#### 1. Definition

An instance $cp$ of class `CompPred<Iter, DA, Comp>` is a predicate comparator that produces boolean values with the given compare function and the attribute associated with an iterator.

```cpp
#include <LEDA/graph/graph_iterator.h>
```
2. Creation

CompPred<Iter, DA, Comp> cp(const DA& da, const Comp& comp,
   typename DA::value_type val);

introduces a variable cp of this class and associates it to the given
data accessor da, compare function comp and value val.

Precondition: Comp is a pointer-to-function type which takes two val-
ues of type typename DA::value_type and produces a boolean return value. Comp might also be a class with member function
bool operator()(typename DA::value_type, typename DA::value_type).

3. Example

In the following example, a node iterator for red nodes will be created. At first the basic part (see sect. 14.13 for explanation of the data accessor node_array_da):

graph G;
NodeIt it(G);
node_array<color> na_colour(G,black);
node_array_da<color> da_colour(na_colour);
assign_some_color_to_each_node();

Now follows the definition of a “red iterator” (Equal<T> yields true, if the given two values are equal):

template<class T>
class Equal {
   public:
      bool operator() (T t1, T t2) const {
         return t1==t2; }
   };

typedef CompPred<NodeIt,node_array_da<color>,Equal<color> > Predicate;
Predicate PredColour(da_colour,Equal<color>(),red);
FilterNodeIt<Predicate,NodeIt> red_it(PredColour,it);

This simplifies the loop to the following:

while(red_it.valid()) {
   do_something(red_it);
   ++red_it; }
Equal<T> is a class that compares two items of the template parameter T by means of a method bool operator()(T,T);. There are some classes available for this purpose: Equal<T>, Unequal<T>, LessThan<T>, LessEqual<T>, GreaterThan<T> and GreaterEqual<T> with obvious semantics, where T is the type of the values. Predicates of the STL can be used as well since they have the same interface.

14.11 Observer Node Iterator (ObserverNodeIt)

1. Definition

An instance it of class ObserverNodeIt<Obs, Iter> is an observer iterator. Any method call of iterators will be “observed” by an internal object of class Obs.

Class ObserverEdgeIt and ObserverAdjIt are defined analogously, i.e. can be used for edge iterators or adjacency iterators, respectively.

Precondition: The template parameter Iter must be a node iterator.

#include <LEDA/graph/graph_iterator.h>

2. Creation

ObserverNodeIt<Obs, Iter> it;

introduces a variable it of this class, not bound to an observer or iterator.

ObserverNodeIt<Obs, Iter> it(Obs& obs, const Iter& base_it);

introduces a variable it of this class bound to the observer obs and base_it.
Precondition: Obs must have methods observe_constructor(), observe_forward(), observe_update(). These three methods may have arbitrary return types (incl. void).

3. Operations

void it.init(const Obs& obs, const Iter& base_it)

initializes it, which is bound to obs and base_it afterwards.
Precondition: it is not bound to an observer or iterator.

Obs& it.get_observer()

returns a reference to the observer to which it is bound.
4. Example

First two simple observer classes. The first one is a dummy class, which ignores all notifications. The second one merely counts the number of calls to `operator++` for all iterators that share the same observer object through copy construction or assignment (of course, a real implementation should apply some kind of reference counting or other garbage collection).

In this example, the counter variable `_count` of class `SimpleCountObserver` will be initialized with the counter variable `_count` of class `DummyObserver`, i.e. the variable is created only once.

```cpp
template <class Iter>
class DummyObserver {
  int* _count;
public:
  DummyObserver() : _count(new int(0)) { }
  void notify_constructor(const Iter&) { }
  void notify_forward(const Iter&) { }
  void notify_update(const Iter&) { }
  int counter() const { return *_count; }
  int* counter_ptr() const { return _count; }
  bool operator==(const DummyObserver& D) const {
    return _count==D._count;
  }
};

template <class Iter, class Observer>
class SimpleCountObserver {
  int* _count;
public:
  SimpleCountObserver() : _count(new int(0)) { }
  SimpleCountObserver(Observer& obs) :
    _count(obs.counter_ptr()) { }
  void notify_constructor(const Iter&) { }
  void notify_forward(const Iter&) { ++(*_count); }
  void notify_update(const Iter&) { }
  int counter() const { return *_count; }
  int* counter_ptr() const { return _count; }
  bool operator==(const SimpleCountObserver& S) const {
    return _count==S._count;
  }
};
```

Next an exemplary application, which counts the number of calls to `operator++` of all adjacency iterator objects inside `dummy_algorithm`. Here the dummy observer class is...
used only as a “Trojan horse,” which carries the pointer to the counter without affecting
the code of the algorithm.

template<class Iter>
bool break_condition (const Iter&) { ... }

template<class ONodeIt, class OAdjIt>
void dummy_algorithm(ONodeIt& it, OAdjIt& it2) {
    while (it.valid()) {
        for (it2.update(it); it2.valid() && !break_condition(it2); ++it2)
            ++it;
    }
}

int write_count(graph& G) {
typedef DummyObserver<NodeIt> DummyObs;
typedef SimpleCountObserver<AdjIt,DummyObs> CountObs;
typedef ObserverNodeIt<NodeObs,NodeIt> ONodeIt;
typedef ObserverAdjIt<CountObs,AdjIt> OAdjIt;

    DummyObs observer;
    ONodeIt it(observer,NodeIt(G));
    CountObs observer2(observer);
    OAdjIt it2(observer2,AdjIt(G));
    dummy_algorithm(it,it2);
    return it2.get_observer().counter();
}

14.12  STL Iterator Wrapper ( STLNodeIt )

1. Definition

An instance \textit{it} of class \textit{STLNodeIt< DataAccessor, Iter >} is a STL iterator wrapper for
node iterators (e.g. \textit{NodeIt}, \textit{FilterNodeIt<pred,NodeIt>}). It adds all type tags and
methods that are necessary for STL conformance; see the standard draft working paper
for details. The type tag \textit{value\_type} is equal to \textit{typename DataAccessor::value\_type}
and the return value of \textit{operator*}.

Class \textit{STLEdgeIt} and \textit{STLAdjIt} are defined analogously, i.e. can be used for edge iterators
or adjacency iterators, respectively.

\textbf{Precondition:} The template parameter \textit{Iter} must be a node iterator. \textit{DataAccessor}
must be a data accessor.
Note: There are specialized versions of STL wrapper iterator classes for each kind of iterator that return different LEDA graph objects.

<table>
<thead>
<tr>
<th>class name</th>
<th>operator*() returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>NodeIt_n</td>
<td>node</td>
</tr>
<tr>
<td>EdgeIt_e</td>
<td>edge</td>
</tr>
<tr>
<td>AdjIt_n</td>
<td>node</td>
</tr>
<tr>
<td>AdjIt_e</td>
<td>edge</td>
</tr>
<tr>
<td>OutAdjIt_n</td>
<td>node</td>
</tr>
<tr>
<td>OutAdjIt_e</td>
<td>edge</td>
</tr>
<tr>
<td>InAdjIt_n</td>
<td>node</td>
</tr>
<tr>
<td>InAdjIt_e</td>
<td>edge</td>
</tr>
</tbody>
</table>

#include <LEDA/graph/graph_iterator.h>

2. Creation

STLNodeIt< DataAccessor, Iter > it(DataAccessor da, const Iter & base_it);

introduces a variable it of this class bound to da and base_it.

3. Operations

STLNodeIt<DataAccessor, Iter>& it = typename DataAccessor::value_type i

assigns the value i, i.e. set(DA, it, i) will be invoked where DA is the associated data accessor and it the associated iterator.

bool it == const STLNodeIt<DataAccessor, Iter>& it2

returns true if the associated values of it and it2 are equal, i.e. get(DA, cit) == get(DA, cit2) is true where cit is the associated iterator of it and cit2 is the associated iterator of it2 and DA is the associated data accessor.

bool it != const STLNodeIt<DataAccessor, Iter>& it2

returns false if the associated value equals the one of the given iterator.

STLNodeIt<DataAccessor, Iter>& it.begin()

resets the iterator to the beginning of the sequence.

STLNodeIt<DataAccessor, Iter>& it.last()

resets the iterator to the ending of the sequence.

STLNodeIt<DataAccessor, Iter>& it.end()

makes the iterators invalid, i.e. past-the-end-value.
typedef DataAccessor::value_type & *it
returns a reference to the associated value, which originally comes from data accessor da. If the associated iterator it is not valid, a dummy value reference is returned and should not be used.
Precondition: access(DA, it) returns a non constant reference to the data associated to it in DA. This functions is defined for all implemented data accessors (e.g. node_array_da, edge_array_da).

14.13 Node Array Data Accessor (node_array_da)

1. Definition

An instance da of class node_array_da<T> is instantiated with a LEDA node_array<T>.

The data in the node array can be accessed by the functions get(da, it) and set(da, it, value) that take as parameters an instance of node_array_da<T> and an iterator, see below.

node_array_da<T>::value_type is a type and equals T.

For node_map<T> there is the variant node_map_da<T> which is defined completely analogous to node_array_da<T>. Classes edge_array_da<T> and edge_map_da<T> are defined analogously, as well.

#include <LEDA/graph/graph_iterator.h>

2. Creation

node_array_da<T> da;
introduces a variable da of this class that is not bound.

node_array_da<T> da(leda::node_array<T>& na);
introduces a variable da of this class bound to na.

3. Operations

T get(const node_array_da<T>& da, const Iter& it)
returns the associated value of it for this accessor.

void set(node_array_da<T>& da, const Iter& it, T val)
sets the associated value of it for this accessor to the given value.
4. Implementation

Constant Overhead.

5. Example

We count the number of 'red nodes' in a parameterized graph G.

```c
int count_red(graph G, node_array<color> COL) {
    node_array_da<color> Color(COL);
    int counter=0;
    NodeIt it(G);
    while (it.valid()) {
        if (get(Color,it)==red) counter++;
        it++; }
    return counter;
}
```

Suppose we want to make this 'algorithm' flexible in the representation of colors. Then we could write this version:

```c
template<class DA>
int count_red_t(graph G, DA Color) {
    int counter=0;
    NodeIt it(G);
    while (it.valid()) {
        if (get(Color,it)==red) counter++;
        it++; }
    return counter;
}
```

With the templatized version it is easily to customize it to match the interface of the version:

```c
int count_red(graph G, node_array<color> COL) {
    node_array_da<color> Color(COL);
    return count_red_t(G,Color); }
```
14.14 Constant Accessors (constant_da)

1. Definition

An instance \(ca\) of class \(constant\_da<\text{T}>\) is bound to a specific value of type \(\text{T}\), and the function \(\text{get}(ca, it)\) simply returns this value for each iterator.

```cpp
#include <LEDA/graph/graph_iterator.h>
```

2. Creation

\[constant\_da<\text{T}> \ ca(\text{T} \ t);\]

introduces a variable \(ca\) of this class bound to the given value \(t\).

3. Operations

\[\text{T}\]

\[\text{get}(\text{const constant}\_da<\text{T}>\& \ ca, \ \text{const Iter}\& \ it)\]

returns the value to which \(ca\) is bound.

4. Example

With the template function of sect. 14.13 we can write a function that counts the number of nodes in a graph:

```cpp
int count_all(graph G) {
    constant_da<color> Color(red);
    return count_red_t(G,Color); }
```

14.15 Node Member Accessors (node_member_da)

1. Definition

An instance \(da\) of class \(node\_member\_da<\text{Str},\text{T}>\) manages the access to a node parameter that is organized as a member of a struct type, which is the first template argument of a parameterized graph \(GRAPH<\text{Str},?>\). The parameter is of type \(\text{T}\) and the struct of type \(\text{Str}\).

Classes \(edge\_member\_da<\text{Str},\text{T}>\) is defined completely analogously.

```cpp
#include <LEDA/graph/graph_iterator.h>
```
CHAPTER 14. GRAPHS AND ITERATORS

2. Creation

\texttt{node\_member\_da<Str,T> \ da;}

introduces a variable \texttt{da} of this class that is not bound.

\texttt{node\_member\_da<Str,T> \ da(Ptr \ ptr);} 

introduces a variable \texttt{da} of this class, which is bound to \texttt{ptr}.

3. Operations

\begin{verbatim}
T \ \ \ \ \ get(const node\_member\_da<Str,T>& \ ma, const Iter& \ it)
\end{verbatim}

returns the associated value of \texttt{it} for this accessor.

\begin{verbatim}
void \ \ \ \ set(node\_member\_da<Str,T>& \ ma, const Iter& \ it, T \ val)
\end{verbatim}

sets the associated value of \texttt{it} for this accessor to the given value.

4. Implementation

Constant Overhead.

The instance \texttt{da} accesses its parameter through a pointer to member of type \texttt{Ptr}, which is defined for example by \texttt{typedef T Str::*Ptr}.

5. Example

We have a parameterized graph \texttt{G} where the node information type is the following struct type \texttt{Str}:

\begin{verbatim}
struct Str {
    int x;
    color col;
};
\end{verbatim}

We want to count the number of red nodes. Since we have the template function of sect. 14.13 we can easily use it to do the computation:

\begin{verbatim}
int count_red(GRAPH<Str,double> \ G) {
    node\_member\_da<Str,color> \ Color(&Str::col);
    return count\_red\_t(G,Color); }
\end{verbatim}
14.16 Node Attribute Accessors (node_attribute_da)

1. Definition

An instance da of class node_attribute_da<T> manages the access to a node parameter with type T of a parameterized graph GRAPH<T,?>.

Classes edge_attribute_da<T> is defined completely analogously.

#include <LEDA/graph/graph_iterator.h>

2. Creation

node_attribute_da<T> da;
introduces a variable da of this class.

3. Operations

T
get(const node_attribute_da<T>& ma, const Iter& it)
returns the associated value of it for this accessor.

void
set(node_attribute_da<T>& ma, const Iter& it, T val)
sets the associated value of it for this accessor to the given value.

4. Implementation

Constant Overhead.

5. Example

Given a parameterized graph G with nodes associated with colours, we want to count the number of red nodes. Since we have the template function of sect. 14.13 we can easily use it to do the computation:

```c
int count_red(GRAPH<color,double> G) {
    node_attribute_da<color> Color;
    return count_red_t(G,Color);
}
```
14.17  Breadth First Search (flexible) ( GIT_BFS )

1. Definition

An instance algorithm of class GIT_BFS< OutAdjIt, Queuetype, Mark > is an implementation of an algorithm that traverses a graph in a breadth first order. The queue used for the search must be provided by the caller and contains the source(s) of the search.

- If the queue is only modified by appending the iterator representing the source node onto the queue, a normal breadth first search beginning at the node of the graph is performed.
- It is possible to initialize the queue with several iterators that represent different roots of breadth first trees.
- By modifying the queue while running the algorithm the behaviour of the algorithm can be changed.
- After the algorithm performed a breadth first search, one may append another iterator onto the queue to restart the algorithm.

**Iterator version:** There is an iterator version of this algorithm: BFS_It. Usage is similar to that of node iterators without the ability to go backward in the sequence.

```c
#include <LEDA/graph/graph_iterator.h>
```

2. Creation

GIT_BFS< OutAdjIt, Queuetype, Mark >
```
algorithm(const Queuetype& q, Mark& ma);
``` creates an instance algorithm of this class bound to the Queue q and data accessor ma.

**Preconditions:**

- Queuetype is a queue parameterized with items of type OutAdjIt.
- q contains the sources of the traversal (for each source node an adjacency iterator referring to it) and
- ma is a data accessor that provides read and write access to a boolean value for each node (accessed through iterators). This value is assumed to be freely usable by algorithm.
14.17. BREADTH FIRST SEARCH (FLEXIBLE) (GIT_BFS)

\[\text{GIT}_\text{BFS}<\text{OutAdjIt},\text{Queuetype},\text{Mark}>\]

\[\text{algorithm}(\text{const Queuetype}\& q, \text{Mark}\& ma, \text{const OutAdjIt}\& ai);\]

creates an instance \textit{algorithm} of this class bound to the queue \(q\),
data accessor \(ma\) and the adjacency iterator \(ai\) representing the
source node of the breadth first traversal.

3. Operations

\textit{void} \text{algorithm}.next() \quad \text{Performs one iteration of the core loop of the algo-
}\textit{OutAdjIt} \text{algorithm}.current() \quad \text{return the “current” iterator.}

\textit{void} \text{algorithm}.finish() \quad \text{executes the algorithm until \textit{finished}() is true, i.e.
\textit{bool} \text{algorithm}.finished() \quad \text{return \textit{true} if the internal Queue is empty.}

\text{Queuetype\&} \text{algorithm}.get_queue() \quad \text{gives direct access to internal Queue.}

4. Example

This example shows how to implement an algorithmic iterator for breadth first search:

\begin{verbatim}
class BFS_It {
  AdjIt       _source;
  node_array<da> _handler;
  node_array_da<bool> _mark;
  queue<AdjIt>   _q;
  GIT_BFS<AdjIt,queue<AdjIt>,node_array_da<bool>> _search;
public:
  BFS_It(graph& G) :
    _source(AdjIt(G)), _handler(G,false),
    _mark(_handler), _search(_q,_mark)
  { _search.get_queue().clear();
    _search.get_queue().append(_source);
  }
  bool valid() const { return !_search.finished(); } 
  node get_node() const { return _search.current().get_node(); }
  BFS_It& operator++() {
    _search.next(); return *this; }
};
\end{verbatim}
With this iterator you can easily iterate through a graph in breadth first fashion:

```cpp
graph G;
BFS_It it(G);
while (it.valid()) {
    // do something reasonable with 'it.get_node()'
    ++it;
}
```

5. Implementation

Each operation requires constant time. Therefore, a normal breadth-first search needs $\mathcal{O}(m + n)$ time.

14.18 Depth First Search (flexible) (GIT_DFS)

1. Definition

An instance `algorithm` of class `GIT_DFS< OutAdjIt, Stacktype, Mark >` is an implementation of an algorithm that traverses a graph in a depth first order. The stack used for the search must be provided by the caller and contains the source(s) of the search.

- If the stack is only modified by pushing the iterator representing the source node onto the stack, a normal depth first search beginning at the node of the graph is performed.
- It is possible to initialize the stack with several iterators that represent different roots of depth first trees.
- By modifying the stack while running the algorithm the behaviour of the algorithm can be changed.
- After the algorithm performed a depth first search, one may push another iterator onto the stack to restart the algorithm.

A next step may return a state which describes the last action. There are the following three possibilities:

1. `dfs_shrink`: an adjacency iterator was popped from the stack, i.e. the treewalk returns in root-direction

2. `dfs_leaf`: same as `dfs_shrink`, but a leaf occurred
3. **dfs_grow_depth**: a new adjacency iterator was appended to the stack because it was detected as not seen before, i.e. the treewalk goes in depth-direction.

4. **dfs_grow_breadth**: the former current adjacency iterator was replaced by the successor iterator, i.e. the treewalk goes in breadth-direction.

**Iterator version**: There is an iterator version of this algorithm: **DFS_It**. Usage is similar to that of node iterators without the ability to go backward in the sequence.

```cpp
#include <LEDA/graph/graph_iterator.h>
```

2. **Creation**

**GIT_DFS< OutAdjIt, Stacktype, Mark >**

```cpp
algorithm(const Stacktype& st, Mark& ma);
```

creates an instance **algorithm** of this class bound to the stack **st** and data accessor **ma**.

**Preconditions:**

- **Stacktype** is a stack parameterized with items of type **OutAdjIt**.
- **st** contains the sources of the traversal (for each source node an adjacency iterator referring to it) and
- **ma** is a data accessor that provides read and write access to a boolean value for each node (accessed through iterators). This value is assumed to be freely usable by **algorithm**.

**GIT_DFS< OutAdjIt, Stacktype, Mark >**

```cpp
algorithm(const Stacktype& st, Mark& ma, const OutAdjIt& ai);
```

creates an instance **algorithm** of this class bound to the stack **st**, data accessor **ma** and the adjacency iterator **ai** representing the source node of the depth first traversal.

3. **Operations**

```cpp
void algorithm.next_unseen() 
```

Performs one iteration of the core loop of the algorithm for one unseen node of the graph.

```cpp
dfs_return algorithm.next() 
```

Performs one iteration of the core loop of the algorithm.

```cpp
OutAdjIt algorithm.current() 
```

returns the "current" iterator.
void algorithm.finish_algo()
        executes the algorithm until \textit{finished()} is true, i.e. exactly if the stack is empty.

bool algorithm.finished() returns \textit{true} if the internal stack is empty.

void algorithm.init(OutAdjIt s)
        initializes the internal stack with \textit{s}.

\textit{Stacktype} & algorithm.get_stack() gives direct access to internal stack.

4. Implementation

Each operation requires constant time. Therefore, a normal depth-first search needs $O(m + n)$ time.

\subsection*{14.19 Topological Sort (flexible) ( \texttt{GIT\_TOPOSORT} )}

1. Definition

An instance \textit{algorithm} of class \texttt{GIT\_TOPOSORT< OutAdjIt, Indeg, Queuetype >} is an implementation of an algorithm that iterates over all nodes in some topological order, if the underlying graph is acyclic. An object of this class maintains an \textit{internal queue}, which contains all nodes (in form of adjacency iterators where the current node is equal to the fixed node) that are not yet passed, but all its predecessors have been passed.

\textbf{Iterator version:} There is an iterator version of this algorithm: \texttt{TOPO\_It}. Usage is similar to that of node iterators without the ability to go backward in the sequence and only a graph is allowed at creation time. Additionally there is \texttt{TOPO\_rev\_It} which traverses the graph in reversed topological order.

\begin{verbatim}
#include <LEDA/graph/graph_iterator.h>
\end{verbatim}

2. Creation

\begin{verbatim}
GIT\_TOPOSORT< OutAdjIt, Indeg, Queuetype >
algorithm(Indeg & indegree);
\end{verbatim}

creates an instance \textit{algorithm} of this class bound to \textit{indeg}. The internal queue of adjacency iterators is empty.

Preconditions:

\begin{itemize}
    \item \textit{Indeg} is a data accessor that must provide both read and write access
\end{itemize}
14.19. **TOPOLOGICAL SORT (FLEXIBLE) (GIT_TOPOSORT)**

- **indegree** stores for every node that corresponds to any iterator the number of incoming edges (has to be computed before)

- **Queuetype** is a queue parameterized with elements of type `OutAdjIt`

The underlying graph need not be acyclic. Whether or not it is acyclic can be tested after execution of the algorithm (function `cycle_found()`).

### 3. Operations

```cpp
void algorithm.next()      // Performs one iteration of the core loop of the algorithm. More specifically, the first element of get_queue() is removed from the queue, and every immediate successor n of this node for which currently holds get(indeg,n)==0 is inserted in get_queue().

void algorithm.finish_algo()  // executes the algorithm until finished() is true, i.e. exactly if the queue is empty.

bool algorithm.finished()   // returns true if the internal queue is empty.

OutAdjIt algorithm.current()  // returns the current adjacency iterator.

Queuetype& algorithm.get_queue()  // gives direct access to internal queue.

bool algorithm.cycle_found()  // returns true if a cycle was found.

void algorithm.reset_acyclic()  // resets the internal flag that a cycle was found.
```

### 4. Implementation

The asymptotic complexity is $O(m + n)$, where $m$ is the number of edges and $n$ the number of nodes.

### 5. Example

This algorithm performs a normal topological sort if the queue is initialized by the set of all nodes with indegree zero:

Definition of `algorithm`, where `indeg` is a data accessor that provides full data access to the number of incoming edges for each node:

```cpp
GIT_TOPOSORT<OutAdjIt,Indeg,Queuetype<Nodehandle> > algorithm(indeg);
```
Initialization of get_queue() with all nodes of type OutAdjIt::nodetype that have zero indegree, i.e. get(indeg,it)==indeg.value_null.

while ( !algorithm.finished() ) {
    // do something reasonable with algo.current()
    algo.next();
}

The source code of function toposort_count() is implemented according to this pattern and may serve as a concrete example.

14.20 Strongly Connected Components (flexible) (GIT_SCC)

1. Definition

An instance algorithm of class GIT_SCC< Out, In, It, OutSt, InSt, NSt, Mark > is an implementation of an algorithm that computes the strongly connected components.

**Iterator version:** There is an iterator version of this algorithm: SCC_It. Usage is similar to that of node iterators without the ability to go backward in the sequence and only a graph is allowed at creation time. Method compnumb() returns the component number of the current node.

```
#include <LEDA/graph/graph_iterator.h>
```

2. Creation

```
GIT_SCC< Out, In, It, OutSt, InSt, NSt, Mark >
    algorithm(OutSt ost, InSt ist, Mark ma, Out oai, const It& it, In iai);
```

creates an instance algorithm of this class bound to the stack st and data accessor ma.

**Preconditions:**

- Out is an adjacency iterator that iterates over the outgoing edges of a fixed vertex
- In is an adjacency iterator that iterates over the incoming edges of a fixed vertex
- OutSt is stack parameterized with items of type Out
- InSt is stack parameterized with items of type In
• **Mark** is a data accessor that has access to a boolean value that is associated with each node of the graph

### 3. Operations

```cpp
int algorithm.state() returns the internal state.

- **NEXT_OUT** = first phase,
- **NEXT_ORDER** = order phase,
- **NEXT_IN** = second phase,
- **NEXT_DONE** = algorithm finished

```cpp
void algorithm.finish() executes the algorithm until finished() is true.

```cpp
bool algorithm.finished() returns true if the algorithm is finished.

```cpp
InSt & algorithm.get_in_stack() gives direct access to the internal stack of incoming adjacency iterators.

```cpp
In algorithm.in_current() returns the current iterator of the internal stack of incoming adjacency iterators.

```cpp
OutSt & algorithm.get_out_stack() gives direct access to the internal stack of outgoing adjacency iterators.

```cpp
Out algorithm.out_current() returns the current iterator of the internal stack of outgoing adjacency iterators.

```cpp
itnodetype algorithm.current_node() returns the current node.

```cpp
int algorithm.compnumb() returns the component number of the fixed node of the current iterator if current state is NEXT_IN.

```cpp
int algorithm.next() Performs one iteration of the core loop of the algorithm.
```

4. **Implementation**

Each operation requires constant time. The algorithm has running time $O(|V| + |E|)$. 
14.21 Dijkstra(flexible) ( GIT_DIKSTRA )

1. Definition

An instance algorithm of this class is an implementation of Dijkstra that can be flexibly initialized, stopped after each iteration of the core loop, and continued, time and again.

**Iterator version:** There is an iterator version of this algorithm: DIJKSTRA.It. Usage is more complex and is documented in the graphiterator leda extension package.

```
#include <LEDA/graph/graph_iterator.h>
```

2. Creation

```
GIT_DIKSTRA<OutAdjIt, Length, Distance, PriorityQueue, QueueItem>
algorithm(const Length& l, Distance& d, const QueueItem& qi);
```

creates an instance algorithm of this class.

The length and distance data accessors are initialized by the parameter list. The set of sources is empty. **Length** is a read only data accessor that gives access to the length of edges and **Distance** is a read/write data accessor that stores the distance of the nodes. **PriorityQueue** is a Queue parameterized with element of type **OutAdjIt** and **QueueItem** is a data accessor gives access to elements of type **PriorityQueue::pq_item**.

**Precondition:** All edge lengths are initialized by values that are large enough to be taken as infinity.

**Remark:** This precondition is not necessary for the algorithm to have a defined behavior. In fact, it may even make sense to break this precondition deliberately. For example, if the distances have been computed before and shall only be updated after inserting new edges, it makes perfect sense to start the algorithm with these distances.

For a completely new computation, the node distances of all nodes are initialized to infinity (i.e. **distance.value_max**).

3. Operations

```
PriorityQueue& algorithm.get_queue()
```

gives direct access to internal priority queue.

```
void algorithm.init(OutAdjIt s)
```

s is added to the set of sources.

```
bool algorithm.finished( )
```

is true iff the algorithm is finished, i.e. the priority queue is empty.

```
OutAdjIt algorithm.current( )
```

returns the current adjacency iterator.
14.21. DIJKSTRA(FLEXIBLE) (GIT_DIJKSTRA)

OutAdjIt algorithm.curr_adj() returns the an adjacency iterator that is currently adjacent to current().

bool algorithm.is_pred() returns true if the current iterator satisfies the dijkstra condition. Can be used to compute the predecessors.

void algorithm.next() performs one iteration of the core loop of the algorithm.

void algorithm.finish_algo() executes the algorithm until finished() is true, i.e. exactly if the priority queue is empty.

4. Example

Class GIT_DIJKSTRA may be used in a deeper layer in a hierarchy of classes and functions. For example, you may write a function which computes shortest path distances with given iterators and data accessors:

```cpp
template<class OutAdjIt, class Length, class Distance,
         class PriorityQueue, class QueueItem>
void GIT_dijkstra_core(OutAdjIt s, Length& length, Distance& distance,
 PriorityQueue& pq, QueueItem& qi) {
 GIT_DIJKSTRA<OutAdjIt,Length,Distance,PriorityQueue,QueueItem>
     internal_dijk(length,distance,qi);
 internal_dijk.get_queue()=pq;
 set(distance,s,distance.value_null);
 if (s.valid()) {
     internal_dijk.init(s);
     internal_dijk.finish_algo();
 }
}
```

In another layer, you would instantiate these iterators and data accessors for a graph and invoke this function.

5. Implementation

The asymptotic complexity is $O(m + n \cdot T(n))$, where $T(n)$ is the(possibly amortized) complexity of a single queue update.

For the priority queues described in Chapter 9.1, it is $T(n) = O(\log n)$. 
Chapter 15

Basic Data Types for Two-Dimensional Geometry

LEDA provides a collection of simple data types for computational geometry, such as points, vectors, directions, hyperplanes, segments, rays, lines, affine transformations, circles, polygons, and operations connecting these types.

The computational geometry kernel has evolved over time. The first kernel (types *point*, *line*, ...) was restricted to two-dimensional geometry and used floating point arithmetic as the underlying arithmetic. We found it very difficult to implement reliable geometric algorithms based on this kernel. See the chapter on computational geometry of [64] for some examples of the danger of floating point arithmetic in geometric computations. Starting with version 3.2 we therefore also provided a kernel based on exact rational arithmetic (types *rat_point*, *rat_segment* ...). (This kernel is still restricted to two dimensions.) From version 4.5 on we offer a two-dimensional kernel based on the type *real*, which also guarantees exact results. The corresponding data types are named *real_point*, *real_segment*, ...

All two-dimensional object types defined in this section support the following operations:

Equality and Identity Tests

```cpp
bool identical(object p, object q)  // Test for identity.
bool p == q                        // Test for equality.
bool p! = q                        // Test for inequality.
```

I/O Operators

```cpp
ostream& ostream& O << object x    // writes the object x to output stream O.
istream& istream& I >> object& x  // reads an object from input stream I into variable x.
```
15.1 Points (point)

1. Definition

An instance of the data type point is a point in the two-dimensional plane $\mathbb{R}^2$. We use $(x, y)$ to denote a point with first (or $x$-) coordinate $x$ and second (or $y$-) coordinate $y$.

#include <LEDA/geo/point.h>

2. Types

point::coord_type the coordinate type (double).

point::point_type the point type (point).

3. Creation

point p; introduces a variable $p$ of type point initialized to the point $(0, 0)$.

point p(double $x$, double $y$);

introduces a variable $p$ of type point initialized to the point $(x, y)$.

point p(vector $v$);

introduces a variable $p$ of type point initialized to the point $(v[0], v[1])$.

Precondition: $v$.dim() = 2.

point p(const point& $p$, int prec);

introduces a variable $p$ of type point initialized to the point with coordinates ($\lfloor P \times x \rfloor / P, \lfloor P \times y \rfloor / P$), where $p = (x, y)$ and $P = 2^{\text{prec}}$. If $\text{prec}$ is non-positive, the new point has coordinates $x$ and $y$.

4. Operations

double p.xcoord() returns the first coordinate of $p$.

double p.ycoord() returns the second coordinate of $p$.

vector p.to_vector() returns the vector $\vec{xy}$.

int p.orientation(const point& $q$, const point& $r$)

returns orientation($p, q, r$) (see below).

double p.area(const point& $q$, const point& $r$)

returns area($p, q, r$) (see below).
15.1. POINTS (POINT)

double p.sqr_dist(const point& q)
returns the square of the Euclidean distance between p and q.

int p.cmp_dist(const point& q, const point& r)
returns \( \text{compare}(p.sqr\_dist(q), p.sqr\_dist(r)) \).

double p.xdist(const point& q)
returns the horizontal distance between p and q.

double p.ydist(const point& q)
returns the vertical distance between p and q.

double p.distance(const point& q)
returns the Euclidean distance between p and q.

double p.distance() returns the Euclidean distance between p and (0, 0).

double p.distance(const point& q, const point& r)
returns the angle between \( \vec{pq} \) and \( \vec{pr} \).

point p.translate_by_angle(double alpha, double d)
returns \( p \) translated in direction \( \alpha \) by distance \( d \). The direction is given by its angle with a right oriented horizontal ray.

point p.translate(double dx, double dy)
returns \( p \) translated by vector \( (dx, dy) \).

point p.translate(const vector& v)
returns \( p + v \), i.e., \( p \) translated by vector \( v \).
Precondition: \( v \).dim() = 2.

point p + const vector& v returns \( p \) translated by vector \( v \).

point p - const vector& v returns \( p \) translated by vector \( -v \).

point p.rotate(const point& q, double a)
returns \( p \) rotated about \( q \) by angle \( a \).

point p.rotate(double a)
returns \( p \).rotate(point(0, 0), a).

point p.rotate90(const point& q, int i = 1)
returns \( p \) rotated about \( q \) by an angle of \( i \times 90 \) degrees.
If \( i > 0 \) the rotation is counter-clockwise otherwise it is clockwise.

point p.rotate90(int i = 1) returns \( p \).rotate90(point(0, 0), i).
\textit{point} \quad \textit{p}.\text{reflect(const point}& q, \textit{const point}& r) \quad \text{returns } p \text{ reflected across the straight line passing through } q \text{ and } r.

\textit{point} \quad \textit{p}.\text{reflect(const point}& q) \quad \text{returns } p \text{ reflected across point } q.

\textit{vector} \quad \textit{p} \:-\textit{const point}& q \quad \text{returns the difference vector of the coordinates.}

\textbf{Non-Member Functions}

\textit{int} \quad \text{cmp\_distances(const point}& p1, \textit{const point}& p2, \textit{const point}& p3, \\
\textit{const point}& p4) \quad \text{compares the distances } (p1, p2) \text{ and } (p3, p4). \text{ Returns } +1 (-1) \text{ if distance } (p1, p2) \text{ is larger (smaller) than distance } (p3, p4), \text{ otherwise } 0.

\textit{point} \quad \text{center(const point}& a, \textit{const point}& b) \quad \text{return} \text{ the center of } a \text{ and } b, \text{i.e. } a + \frac{ab}{2}.

\textit{point} \quad \text{midpoint(const point}& a, \textit{const point}& b) \quad \text{return} \text{ the center of } a \text{ and } b.

\textit{int} \quad \text{orientation(const point}& a, \textit{const point}& b, \textit{const point}& c) \quad \text{computes the orientation of points } a, b, \text{ and } c \text{ as the sign of the determinant}

\begin{vmatrix}
  a_x & a_y & 1 \\
  b_x & b_y & 1 \\
  c_x & c_y & 1
\end{vmatrix}

\text{i.e., it returns } +1 \text{ if point } c \text{ lies left of the directed line through } a \text{ and } b, 0 \text{ if } a, b, \text{ and } c \text{ are collinear, and } -1 \text{ otherwise.}

\textit{int} \quad \text{cmp\_signed\_dist(const point}& a, \textit{const point}& b, \textit{const point}& c, \\
\textit{const point}& d) \quad \text{compares (signed) distances of } c \text{ and } d \text{ to the straight line passing through } a \text{ and } b \text{ (directed from } a \text{ to } b). \text{ Returns } +1 (-1) \text{ if } c \text{ has larger (smaller) distance than } d
\text{ and } 0 \text{ if distances are equal.}

\textit{double} \quad \text{area(const point}& a, \textit{const point}& b, \textit{const point}& c) \quad \text{computes the signed area of the triangle determined by } a, b, c, \text{ positive if } \text{orientation}(a, b, c) > 0 \text{ and negative otherwise.}
bool collinear(const point & a, const point & b, const point & c)
returns true if points a, b, c are collinear, i.e.,
orientation(a, b, c) = 0, and false otherwise.

bool right_turn(const point & a, const point & b, const point & c)
returns true if points a, b, c form a righ turn, i.e.,
orientation(a, b, c) < 0, and false otherwise.

bool left_turn(const point & a, const point & b, const point & c)
returns true if points a, b, c form a left turn, i.e.,
orientation(a, b, c) > 0, and false otherwise.

int side_of_halfspace(const point & a, const point & b, const point & c)
returns the sign of the scalar product \((b - a) \cdot (c - a)\).
If \(b \neq a\) this amounts to: Let \(h\) be the open halfspace
orthogonal to the vector \(b - a\), containing \(b\), and having
\(a\) in its boundary. Returns +1 if \(c\) is contained in \(h\),
returns 0 is \(c\) lies on the the boundary of \(h\), and returns
−1 is \(c\) is contained in the interior of the complement of \(h\).

int side_of_circle(const point & a, const point & b, const point & c, const point & d)
returns +1 if point \(d\) lies left of the directed circle
through points \(a\), \(b\), and \(c\) if \(a,b,c,d\) are cocircular, and −1 otherwise.

bool inside_circle(const point & a, const point & b, const point & c, const point & d)
returns true if point \(d\) lies in the interior of the circle
through points \(a\), \(b\), and \(c\), and false otherwise.

bool outside_circle(const point & a, const point & b, const point & c, const point & d)
returns true if point \(d\) lies outside of the circle through
points \(a\), \(b\), and \(c\), and false otherwise.

bool on_circle(const point & a, const point & b, const point & c, const point & d)
returns true if points \(a\), \(b\), \(c\), and \(d\) are cocircular.

bool cocircular(const point & a, const point & b, const point & c, const point & d)
returns true if points \(a\), \(b\), \(c\), and \(d\) are cocircular.

int compare_by_angle(const point & a, const point & b, const point & c, const point & d)
compares vectors \(b - a\) and \(d - c\) by angle (more efficient
than calling compare_by_angle(b - a, d - x) on vectors).

bool affinely_independent(const array<point> & A)
decides whether the points in \(A\) are affinely independent.
bool contained_in_simplex(const array<point>& A, const point& p)
determines whether \( p \) is contained in the simplex spanned by the points in \( A \). \( A \) may consist of up to 3 points.

Precondition: The points in \( A \) are affinely independent.

bool contained_in_affine_hull(const array<point>& A, const point& p)
determines whether \( p \) is contained in the affine hull of the points in \( A \).
15.2 Segments ( segment )

1. Definition

An instance $s$ of the data type `segment` is a directed straight line segment in the two-dimensional plane, i.e., a straight line segment $[p, q]$ connecting two points $p, q \in \mathbb{R}^2$. $p$ is called the source or start point and $q$ is called the target or end point of $s$. The length of $s$ is the Euclidean distance between $p$ and $q$. If $p = q$ $s$ is called empty. We use `line(s)` to denote a straight line containing $s$. The angle between a right oriented horizontal ray and $s$ is called the direction of $s$.

```cpp
#include <LEDA/geo/segment.h>
```

2. Types

- `segment::coord_type` the coordinate type (double).
- `segment::point_type` the point type (point).

3. Creation

- `segment s(const point& p, const point& q);` introduces a variable $s$ of type `segment`. $s$ is initialized to the segment $[p, q]$.
- `segment s(const point& p, const vector& v);` introduces a variable $s$ of type `segment`. $s$ is initialized to the segment $[p, p + v]$.
  *Precondition: $v$dim() = 2.*
- `segment s(double x1, double y1, double x2, double y2);` introduces a variable $s$ of type `segment`. $s$ is initialized to the segment $[(x_1, y_1), (x_2, y_2)]$.
- `segment s(const point& p, double alpha, double length);` introduces a variable $s$ of type `segment`. $s$ is initialized to the segment with start point $p$, direction $alpha$, and length $length$.
- `segment s;` introduces a variable $s$ of type `segment`. $s$ is initialized to the empty segment.
- `segment s(const segment& s1, int);` introduces a variable $s$ of type `segment`. $s$ is initialized to a copy of $s_1$. 
4. Operations

**point** \( s.\text{start}() \) returns the source point of segment \( s \).

**point** \( s.\text{end}() \) returns the target point of segment \( s \).

**double** \( s.\text{xcoord1}() \) returns the x-coordinate of \( s.\text{source}() \).

**double** \( s.\text{xcoord2}() \) returns the x-coordinate of \( s.\text{target}() \).

**double** \( s.\text{ycoord1}() \) returns the y-coordinate of \( s.\text{source}() \).

**double** \( s.\text{ycoord2}() \) returns the y-coordinate of \( s.\text{target}() \).

**double** \( s.\text{dx}() \) returns the \( x\text{coord2} - x\text{coord1} \).

**double** \( s.\text{dy}() \) returns the \( y\text{coord2} - y\text{coord1} \).

**double** \( s.\text{slope}() \) returns the slope of \( s \).

*Precondition:* \( s \) is not vertical.

**double** \( s.\text{sqr_length}() \) returns the square of the length of \( s \).

**double** \( s.\text{length}() \) returns the length of \( s \).

**vector** \( s.\text{to_vector}() \) returns the vector \( s.\text{target}() - s.\text{source}() \).

**double** \( s.\text{direction}() \) returns the direction of \( s \) as an angle in the intervall \([0, 2\pi)\).

**double** \( s.\text{angle}() \) returns \( s.\text{direction}() \).

**double** \( s.\text{angle}(const \text{segment} \& t) \) returns the angle between \( s \) and \( t \), i.e., \( t.\text{direction}() - s.\text{direction}() \).

**bool** \( s.\text{is_trivial}() \) returns true if \( s \) is trivial.

**bool** \( s.\text{is_vertical}() \) returns true iff \( s \) is vertical.

**bool** \( s.\text{is_horizontal}() \) returns true iff \( s \) is horizontal.

**int** \( s.\text{orientation}(const \text{point} \& p) \) computes orientation\((s.\text{source}(), s.\text{target}(), p)\) (see below).

**double** \( s.\text{xproj}(double y) \) returns \( p.x\text{coord}(), \) where \( p \in \text{line}(s) \) with \( p.y\text{coord}() = y \).

*Precondition:* \( s \) is not horizontal.

**double** \( s.\text{yproj}(double x) \) returns \( p.y\text{coord}(), \) where \( p \in \text{line}(s) \) with \( p.x\text{coord}() = x \).

*Precondition:* \( s \) is not vertical.

**double** \( s.\text{yabs}() \) returns the y-abscissa of \( \text{line}(s) \), i.e., \( s.\text{yproj}(0) \).

*Precondition:* \( s \) is not vertical.
\begin{itemize}
  \item \texttt{bool }\texttt{s.contains(const point}\& p) \texttt{decides whether s contains p.}
  \item \texttt{bool }\texttt{s.intersection(const segment}\& t) \texttt{decides whether s and t intersect in one point.}
  \item \texttt{bool }\texttt{s.intersection(const segment}\& t, \texttt{point}\& p) \texttt{if s and t intersect in a single point this point is assigned to p and the result is true, otherwise the result is false.}
  \item \texttt{bool }\texttt{s.intersection_of_lines(const segment}\& t, \texttt{point}\& p) \texttt{if line(s) and line(t) intersect in a single point this point is assigned to p and the result is true, otherwise the result is false.}
  \item \texttt{segment }\texttt{s.translate_by_angle(double alpha, double d)} \texttt{returns s translated in direction alpha by distance d.}
  \item \texttt{segment }\texttt{s.translate(double dx, double dy)} \texttt{returns s translated by vector (dx, dy).}
  \item \texttt{segment }\texttt{s.translate(const vector}\& v) \texttt{returns s + v, i.e., s translated by vector v.}
  \texttt{Precondition: v.dim()} = 2.
  \item \texttt{segment }\texttt{s + const vector}\& v \texttt{returns s translated by vector v.}
  \item \texttt{segment }\texttt{s - const vector}\& v \texttt{returns s translated by vector -v.}
  \item \texttt{segment }\texttt{s.perpendicular(const point}\& p) \texttt{returns the segment perpendicular to s with source p and target on line(s).}
  \item \texttt{double }\texttt{s.distance(const point}\& p) \texttt{returns the Euclidean distance between p and s.}
  \item \texttt{double }\texttt{s.sqr_dist(const point}\& p) \texttt{returns the squared Euclidean distance between p and s.}
  \item \texttt{double }\texttt{s.distance()} \texttt{returns the Euclidean distance between (0,0) and s.}
  \item \texttt{segment }\texttt{s.rotate(const point}\& q, \texttt{double a)} \texttt{returns s rotated about point q by angle a.}
  \item \texttt{segment }\texttt{s.rotate(double alpha)} \texttt{returns s.rotate(s.source(), alpha).}
\end{itemize}
segment s.rotate90(const point& q, int i = 1)
returns s rotated about q by an angle of $i \times 90$ degrees.
If $i > 0$ the rotation is counter-clockwise otherwise it is
clockwise.

segment s.rotate90(int i = 1)
returns s.rotate90(s.source(), i).

segment s.reflect(const point& p, const point& q)
returns s reflected across the straight line passing through
p and q.

segment s.reflect(const point& p)
returns s reflected across point p.

segment s.reverse() returns s reversed.

Non-Member Functions

int orientation(const segment& s, const point& p)
computes orientation(s.source(), s.target(), p).

int cmp_slopes(const segment& s1, const segment& s2)
returns compare(slope(s1), slope(s2)).

int cmp_segments_at_xcoord(const segment& s1, const segment& s2,
const point& p)
compares points $l_1 \cap v$ and $l_2 \cap v$ where $l_i$ is the line under-
lying segment $s_i$ and $v$ is the vertical straight line passing
through point $p$.

bool parallel(const segment& s1, const segment& s2)
returns true if $s_1$ and $s_2$ are parallel and false otherwise.
15.3 Straight Rays (ray)

1. Definition

An instance $r$ of the data type ray is a directed straight ray in the two-dimensional plane. The angle between a right oriented horizontal ray and $r$ is called the direction of $r$.

```cpp
#include <LEDA/geo/ray.h>
```

2. Types

```cpp
ray::coord_type the coordinate type (double).
ray::point_type the point type (point).
```

3. Creation

```cpp
ray r(const point& p, const point& q);
introduces a variable $r$ of type ray. $r$ is initialized to the ray starting at point $p$ and passing through point $q$.

ray r(const segment& s); introduces a variable $r$ of type ray. $r$ is initialized to ray(s.source(), s.target()).

ray r(const point& p, const vector& v);
introduces a variable $r$ of type ray. $r$ is initialized to ray($p, p + v$).

ray r(const point& p, double alpha);
introduces a variable $r$ of type ray. $r$ is initialized to the ray starting at point $p$ with direction $alpha$.

ray r;
introduces a variable $r$ of type ray. $r$ is initialized to the ray starting at the origin with direction 0.

ray r(const ray& r1, int);
introduces a variable $r$ of type ray. $r$ is initialized to a copy of $r1$. The second argument is for compatibility with rat_ray.

4. Operations

```cpp
point r.source() returns the source of $r$.
point r.point1() returns the source of $r$.
point r.point2() returns a point on $r$ different from r.source().
```
double r.direction() returns the direction of r.

double r.angle(const ray & s) returns the angle between r and s, i.e., s.direction() - r.direction().

bool r.is_vertical() returns true iff r is vertical.

bool r.is_horizontal() returns true iff r is horizontal.

double r.slope() returns the slope of the straight line underlying r. 
Precondition: r is not vertical.

bool r.intersection(const ray & s, point & inter) if r and s intersect in a single point this point is assigned to inter and the result is true, otherwise the result is false.

bool r.intersection(const segment & s, point & inter) if r and s intersect in a single point this point is assigned to inter and the result is true, otherwise the result is false.

ray r.translate_by_angle(double a, double d) returns r translated in direction a by distance d.

ray r.translate(double dx, double dy) returns r translated by vector (dx, dy).

ray r.translate(const vector & v) returns r translated by vector v 
Precondition: v.dim() = 2.

ray r + const vector & v returns r translated by vector v.

ray r - const vector & v returns r translated by vector -v.

ray r.rotate(const point & q, double a) returns r rotated about point q by angle a.

ray r.rotate(double a) returns r.rotate(point(0,0), a).

ray r.rotate90(const point & q, int i = 1) returns r rotated about q by an angle of i × 90 degrees. If i > 0 the rotation is counter-clockwise otherwise it is clockwise.

ray r.reflect(const point & p, const point & q) returns r reflected across the straight line passing through p and q.
ray \quad r.\text{reflect}(\text{const point} & \ p) \quad \text{returns } r \text{ reflected across point } p.

ray \quad r.\text{reverse}() \quad \text{returns } r \text{ reversed.}

bool \quad r.\text{contains(\text{const point} & \ )} \quad \text{decides whether } r \text{ contains } p.

bool \quad r.\text{contains(\text{const segment} & \ )} \quad \text{decides whether } r \text{ contains } s.

\textbf{Non-Member Functions}

int \quad \text{orientation(\text{const ray} & \ r, \text{const point} & \ p)} \\
\quad \text{computes orientation}(a, b, p), \text{ where } a \neq b \text{ and } a \text{ and } b \text{ appear in this order on ray } r.

int \quad \text{cmp.slopes(\text{const ray} & \ r1, \text{const ray} & \ r2)} \\
\quad \text{returns compare(slope}(r_1), \text{ slope}(r_2)) \text{ where } \text{slope}(r_i) \text{ denotes the slope of the straight line underlying } r_i.$
15.4 Straight Lines (line)

1. Definition

An instance \( l \) of the data type line is a directed straight line in the two-dimensional plane. The angle between a right oriented horizontal line and \( l \) is called the direction of \( l \).

```c
#include <LEDA/geo/line.h>
```

2. Types

- `line::coord_type` — the coordinate type (double).
- `line::point_type` — the point type (point).

3. Creation

- `line l(const point& p, const point& q);` introduces a variable \( l \) of type line. \( l \) is initialized to the line passing through points \( p \) and \( q \) directed form \( p \) to \( q \).
- `line l(const segment& s);` introduces a variable \( l \) of type line. \( l \) is initialized to the line supporting segment \( s \).
- `line l(const ray& r);` introduces a variable \( l \) of type line. \( l \) is initialized to the line supporting ray \( r \).
- `line l(const point& p, const vector& v);` introduces a variable \( l \) of type line. \( l \) is initialized to the line passing through points \( p \) and \( p + v \).
- `line l(const point& p, double alpha);` introduces a variable \( l \) of type line. \( l \) is initialized to the line passing through point \( p \) with direction \( \alpha \).
- `line l;` introduces a variable \( l \) of type line. \( l \) is initialized to the line passing through the origin with direction 0.

4. Operations

- `point l.point1();` returns a point on \( l \).
- `point l.point2();` returns a second point on \( l \).
- `segment l.seg();` returns a segment on \( l \).
- `double Langle(const line& g)` returns the angle between \( l \) and \( g \), i.e., \( g.direction() - l.direction() \).
double L.direction() returns the direction of L.

double L.angle() returns L.direction().

bool L.is_vertical() returns true iff L is vertical.

bool L.is_horizontal() returns true iff L is horizontal.

double L.sqr_dist(const point& q) returns the square of the distance between L and q.

double L.distance(const point& q) returns the distance between L and q.

int L.orientation(const point& p) returns orientation(L.point1(), L.point2(), p).

double L.slope() returns the slope of L.
Precondition: L is not vertical.

double L.y_proj(double x) returns p.ycoord(), where p ∈ L with p.xcoord() = x.
Precondition: L is not vertical.

double L.x_proj(double y) returns p.xcoord(), where p ∈ L with p.ycoord() = y.
Precondition: L is not horizontal.

double L_y_abs() returns the y-abscissa of L (L.y_proj(0)).
Precondition: L is not vertical.

bool L.intersection(const line& g, point& p)
if L and g intersect in a single point this point is assigned to p and the result is true, otherwise the result is false.

bool L.intersection(const segment& s, point& inter)
if L and s intersect in a single point this point is assigned to p and the result is true, otherwise the result is false.

bool L.intersection(const segment& s)
returns true, if L and s intersect, false otherwise.

line L.translate_by_angle(double a, double d)
returns L translated in direction a by distance d.

line L.translate(double dx, double dy)
returns L translated by vector (dx, dy).
line \( l.\text{translate(const vector& v)} \)
returns \( l \) translated by vector \( v \).
Precondition: \( v.\text{dim()} = 2 \).

line \( l + \text{const vector& v} \)
returns \( l \) translated by vector \( v \).

line \( l - \text{const vector& v} \)
returns \( l \) translated by vector \(-v\).

line \( l.\text{rotate(const point& q, double a)} \)
returns \( l \) rotated about point \( q \) by angle \( a \).

line \( l.\text{rotate(double a)} \)
returns \( l.\text{rotate(point}(0,0), a) \).

line \( l.\text{rotate90(const point& q, int i = 1)} \)
returns \( l \) rotated about \( q \) by an angle of \( i \times 90 \) degrees. If \( i > 0 \) the rotation is counter-clockwise otherwise it is clockwise.

line \( l.\text{reflect(const point& p, const point& q)} \)
returns \( l \) reflected across the straight line passing through \( p \) and \( q \).

line \( l.\text{reverse()} \)
returns \( l \) reversed.

segment \( l.\text{perpendicular(const point& p)} \)
returns the segment perpendicular to \( l \) with source \( p \) and target on \( l \).

point \( l.\text{dual()} \)
returns the point dual to \( l \).
Precondition: \( l \) is not vertical.

int \( l.\text{side_of(const point& p)} \)
computes orientation(\( a, b, p \)), where \( a \neq b \) and \( a \) and \( b \) appear in this order on line \( l \).

bool \( l.\text{contains(const point& p)} \)
returns true if \( p \) lies on \( l \).

bool \( l.\text{clip(point p, point q, segment& s)} \)
clips \( l \) at the rectangle \( R \) defined by \( p \) and \( q \). Returns true if the intersection of \( R \) and \( l \) is non-empty and returns false otherwise. If the intersection is non-empty the intersection is assigned to \( s \); It is guaranteed that the source node of \( s \) is no larger than its target node.

**Non-Member Functions**

int \( \text{orientation(const line& l, const point& p)} \)
computes orientation(\( a, b, p \)), where \( a \neq b \) and \( a \) and \( b \) appear in this order on line \( l \).
int cmp_slopes(const line& l1, const line& l2)
returns compare(slope(l1), slope(l2)).
15.5 Circles (circle)

1. Definition

An instance $C$ of the data type circle is an oriented circle in the plane passing through three points $p_1, p_2, p_3$. The orientation of $C$ is equal to the orientation of the three defining points, i.e. $\text{orientation}(p_1, p_2, p_3)$. If $|\{p_1, p_2, p_3\}| = 1$ $C$ is the empty circle with center $p_1$. If $p_1, p_2, p_3$ are collinear $C$ is a straight line passing through $p_1, p_2$ and $p_3$ in this order and the center of $C$ is undefined.

```
#include <LEDA/geo/circle.h>
```

2. Types

```
circle::coord_type the coordinate type (double).
circle::point_type the point type (point).
```

3. Creation

```
circle $C$(const point& $a$, const point& $b$, const point& $c$);
```

introduces a variable $C$ of type circle. $C$ is initialized to the oriented circle through points $a$, $b$, and $c$.

```
circle $C$(const point& $a$, const point& $b$);
```

introduces a variable $C$ of type circle. $C$ is initialized to the counter-clockwise oriented circle with center $a$ passing through $b$.

```
circle $C$(const point& $a$);
```

introduces a variable $C$ of type circle. $C$ is initialized to the trivial circle with center $a$.

```
circle $C$;
```

introduces a variable $C$ of type circle. $C$ is initialized to the trivial circle with center $(0, 0)$.

```
circle $C$(const point& $c$, double $r$);
```

introduces a variable $C$ of type circle. $C$ is initialized to the circle with center $c$ and radius $r$ with positive (i.e. counter-clockwise) orientation.

```
circle $C$(double $x$, double $y$, double $r$);
```

introduces a variable $C$ of type circle. $C$ is initialized to the circle with center $(x, y)$ and radius $r$ with positive (i.e. counter-clockwise) orientation.
introduces a variable \( C \) of type \textit{circle}. \( C \) is initialized to a copy of \( c \). The second argument is for compatibility with \textit{rat\_circle}.

### 4. Operations

- **point** \( C\).center() returns the center of \( C \).
  
  \textit{Precondition}: The orientation of \( C \) is not 0.

- **double** \( C\).radius() returns the radius of \( C \).
  
  \textit{Precondition}: The orientation of \( C \) is not 0.

- **double** \( C\).sqr_radius() returns the squared radius of \( C \).
  
  \textit{Precondition}: The orientation of \( C \) is not 0.

- **point** \( C\).point1() returns \( p_1 \).

- **point** \( C\).point2() returns \( p_2 \).

- **point** \( C\).point3() returns \( p_3 \).

- **point** \( C\).point_on_circle(double alpha, double = 0) returns a point \( p \) on \( C \) with angle of \( \alpha \).
  
  The second argument is for compatibility with \textit{rat\_circle}.

- **bool** \( C\).is_degenerate() returns true if the defining points are collinear.

- **bool** \( C\).is_trivial() returns true if \( C \) has radius zero.

- **bool** \( C\).is_line() returns true if \( C \) is a line.

- **line** \( C\).to_line() returns line(point1(), point3()).

- **int** \( C\).orientation() returns the orientation of \( C \).

- **int** \( C\).side_of(const point& p) returns \(-1, +1, \) or \(0\) if \( p \) lies right of, left of, or on \( C \) respectively.

- **bool** \( C\).inside(const point& p) returns true iff \( p \) lies inside of \( C \).

- **bool** \( C\).outside(const point& p) returns true iff \( p \) lies outside of \( C \).

- **bool** \( C\).contains(const point& p) returns true iff \( p \) lies on \( C \).

- **circle** \( C\).translate_by_angle(double a, double d) returns \( C \) translated in direction \( a \) by distance \( d \).
circle \ C.\text{translate}(\text{double}\ dx,\ \text{double}\ dy)  
returns \( C \) translated by vector \((dx, dy)\).

circle \ C.\text{translate}(\text{const}\ \text{vector} &\ v)  
returns \( C \) translated by vector \( v \).

circle \ C + \text{const}\ \text{vector} &\ v  
returns \( C \) translated by vector \( v \).

circle \ C - \text{const}\ \text{vector} &\ v  
returns \( C \) translated by vector \(-v\).

circle \ C.\text{rotate}(\text{const}\ \text{point} &\ q, \ \text{double}\ a)  
returns \( C \) rotated about point \( q \) by angle \( a \).

circle \ C.\text{rotate}(\text{double}\ a)  
returns \( C \) rotated about the origin by angle \( a \).

circle \ C.\text{rotate90}(\text{const}\ \text{point} &\ q, \ \text{int}\ i = 1)  
returns \( C \) rotated about \( q \) by an angle of \( i \times 90 \) degrees. If \( i > 0 \) the rotation is counter-clockwise otherwise it is clockwise.

circle \ C.\text{reflect}(\text{const}\ \text{point} &\ p, \ \text{const}\ \text{point} &\ q)  
returns \( C \) reflected across the straight line passing through \( p \) and \( q \).

circle \ C.\text{reflect}(\text{const}\ \text{point} &\ p)  
returns \( C \) reflected across point \( p \).

circle \ C.\text{reverse}()  
returns \( C \) reversed.

\begin{verbatim}
list<\text{point}> C.\text{intersection}(\text{const}\ \text{circle} &\ D)  
returns \( C \cap D \) as a list of points.

list<\text{point}> C.\text{intersection}(\text{const}\ \text{line} &\ l)  
returns \( C \cap l \) as a list of (zero, one, or two) points sorted along \( l \).

list<\text{point}> C.\text{intersection}(\text{const}\ \text{segment} &\ s)  
returns \( C \cap s \) as a list of (zero, one, or two) points sorted along \( s \).
\end{verbatim}

segment \ C.\text{left_tangent}(\text{const}\ \text{point} &\ p)  
returns the line segment starting in \( p \) tangent to \( C \) and left of segment \([p, C.\text{center}()]\).

segment \ C.\text{right_tangent}(\text{const}\ \text{point} &\ p)  
returns the line segment starting in \( p \) tangent to \( C \) and right of segment \([p, C.\text{center}()]\).

double \ C.\text{distance}(\text{const}\ \text{point} &\ p)  
returns the distance between \( C \) and \( p \).
double C.sqr_dist(const point& p) returns the squared distance between C and p.

double C.distance(const line& l) returns the distance between C and l.

double C.distance(const circle& D) returns the distance between C and D.

bool radical_axis(const circle& C1, const circle& C2, line& rad_axis)
if the radical axis for C1 and C2 exists, it is assigned to rad_axis and true is returned; otherwise
the result is false.
15.6 Polygons ( POLYGON )

1. Definition

There are three instantiations of POLYGON: polygon (floating point kernel), rat_polygon (rational kernel) and real_polygon (real kernel). The respective header file name corresponds to the type name (with “.h” appended).

An instance $P$ of the data type POLYGON is a cyclic list of points (equivalently segments) in the plane. A polygon is called simple if all nodes of the graph induced by its segments have degree two and it is called weakly simple, if its segments are disjoint except for common endpoints and if the chain does not cross itself. See the LEDA book for more details.

A weakly simple polygon splits the plane into an unbounded region and one or more bounded regions. For a simple polygon there is just one bounded region. When a weakly simple polygon $P$ is traversed either the bounded region is consistently to the left of $P$ or the unbounded region is consistently to the left of $P$. We say that $P$ is positively oriented in the former case and negatively oriented in the latter case. We use $P$ to also denote the region to the left of $P$ and call this region the positive side of $P$.

The number of vertices is called the size of $P$. A polygon with empty vertex sequence is called empty.

Only the types rat_polygon and real_polygon guarantee correct results. Almost all operations listed below are available for all the three instantiations of POLYGON. There is a small number of operations that are only available for polygon, they are indicated as such.

```c
#include <LEDA/geo/generic/POLYGON.h>
```

2. Types

POLYGON::coord_type the coordinate type (e.g. rational).

POLYGON::point_type the point type (e.g. rat_point).

POLYGON::segment_type the segment type (e.g. rat_segment).

POLYGON::float_type the corresponding floating-point type (polygon).

3. Creation

POLYGON $P$; introduces a variable $P$ of type POLYGON. $P$ is initialized to the empty polygon.
15.6. POLYGONS ( POLYGON )

POLYGON P(const list<POINT>& pl, 
CHECK_TYPE check = POLYGON::WEAKLY_SIMPLE, 
RESPECT_TYPE respect_orientation = 
POLYGON::RESPECT_ORIENTATION);
introduces a variable P of type POLYGON. P is initialized
to the polygon with vertex sequence pl. If respect_orientation
is DISREGARD_ORIENTATION, the positive orientation is
chosen.
Precondition: If check is SIMPLE, pl must define a simple
polygon, and if check is WEAKLY_SIMPLE, pl must define
a weakly simple polygon. If no test is to performed, the sec-
ond argument has to be set to NO_CHECK. The constants
NO_CHECK, SIMPLE, and WEAKLY_SIMPLE are part of
a local enumeration type CHECK_TYPE.

POLYGON P(const polygon& Q, int prec = rat_point::default_precision);
introduces a variable P of type POLYGON. P is initialized
to a rational approximation of the (floating point) polygon Q
of coordinates with denominator at most prec. If prec is zero,
the implementation chooses prec large enough such that there
is no loss of precision in the conversion.

4. Operations

<table>
<thead>
<tr>
<th>Type</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>polygon</td>
<td>P.to_float()</td>
<td>returns a floating point approximation of P.</td>
</tr>
<tr>
<td>void</td>
<td>P.normalize()</td>
<td>simplifies the homogenous representation by calling p.normalize( ) for every vertex p of P.</td>
</tr>
<tr>
<td>bool</td>
<td>P.is_simple()</td>
<td>tests whether P is simple or not.</td>
</tr>
<tr>
<td>bool</td>
<td>P.is_weakly_simple()</td>
<td>tests whether P is weakly simple or not.</td>
</tr>
<tr>
<td>bool</td>
<td>P.is_weakly_simple(list&lt;POINT&gt;&amp; L)</td>
<td>as above, returns all proper points of intersection in L.</td>
</tr>
<tr>
<td>POLYGON::CHECK_TYPE</td>
<td>P.check_simplicity()</td>
<td>returns the CHECK_TYPE of P. The result can be SIMPLE, WEAKLY_SIMPLE or NOT_WEAKLY_SIMPLE.</td>
</tr>
<tr>
<td>bool</td>
<td>P.is_convex()</td>
<td>returns true if P is convex, false otherwise.</td>
</tr>
<tr>
<td>const list&lt;POINT&gt;&amp;</td>
<td>P.vertices()</td>
<td>returns the sequence of vertices of P in counterclockwise ordering.</td>
</tr>
</tbody>
</table>
const list<SEGMENT>& P.segments() returns the sequence of bounding segments of P in counterclockwise ordering.

list<POINT> P.intersection(const SEGMENT& s) returns the proper crossings between P and s as a list of points.

list<POINT> P.intersection(const LINE& l) returns the proper crossings between P and l as a list of points.

POLYGON P.intersect_halfplane(const LINE& l) returns the intersection of P with the halfspace on the positive side of l.

int P.size() returns the size of P.

bool P.empty() returns true if P is empty, false otherwise.

POLYGON P.translate(RAT_TYPE dx, RAT_TYPE dy) returns P translated by vector (dx, dy).

POLYGON P.translate(INT_TYPE dx, INT_TYPE dy, INT_TYPE dw) returns P translated by vector (dx/dw, dy/dw).

POLYGON P.translate(const VECTOR& v) returns P translated by vector v.

POLYGON P + const VECTOR& v returns P translated by vector v.

POLYGON P − const VECTOR& v returns P translated by vector −v.

POLYGON P.rotate90(const POINT& q, int i = 1) returns P rotated about q by an angle of i × 90 degrees. If i > 0 the rotation is counterclockwise otherwise it is clockwise.

POLYGON P.reflect(const POINT& p, const POINT& q) returns P reflected across the straight line passing through p and q.

POLYGON P.reflect(const POINT& p) returns P reflected across point p.

RAT_TYPE P.sqr_dist(const POINT& p) returns the square of the minimal Euclidean distance between a segment in P and p. Returns zero if P is empty.
15.6. POLYGONS (POLYGON)

**POLYGON** \( P.\text{complement}( ) \) returns the complement of \( P \).

**POLYGON** \( P.\text{eliminate\_colinear\_vertices}( ) \) returns a copy of \( P \) without colinear vertices.

**list\text{\langle\text{POLYGON}\rangle}** \( P.\text{simple\_parts}( ) \) returns the simple parts of \( P \) as a list of simple polygons.

**list\text{\langle\text{POLYGON}\rangle}** \( P.\text{split\_into\_weakly\_simple\_parts}(\text{bool strict} = \text{false}) \) splits \( P \) into a set of weakly simple polygons whose union coincides with the inner points of \( P \). If \text{strict} is true a point is considered an inner point if it is left of all surrounding segments, otherwise it is considered as an inner point if it is locally to the left of some surrounding edge. (This function is experimental.)

**GEN\_POLYGON** \( P.\text{make\_weakly\_simple}(\text{bool with\_neg\_parts} = \text{true}, \text{bool strict} = \text{false}) \) creates a weakly simple generalized polygon \( Q \) from a possibly non-simple polygon \( P \) such that \( Q \) and \( P \) have the same inner points. The flag \text{with\_neg\_parts} determines whether inner points in negatively oriented parts are taken into account, too. The meaning of the flag \text{strict} is the same as in the method above. (This function is experimental.)

**GEN\_POLYGON** \( P.\text{buffer}(\text{RAT\_TYPE} d) \) adds an exterior buffer zone to \( P \) \((d > 0)\), or removes an interior buffer zone from \( P \) \((d < 0)\). More precisely, for \( d \geq 0 \) define the buffer tube \( T \) as the set of all points in the complement of \( P \) whose distance to \( P \) is at most \( d \). Then the function returns \( P \cup T \). For \( d < 0 \) let \( T \) denote the set of all points in \( P \) whose distance to the complement is less than \(|d|\). Then the result is \( P \setminus T \). (This function is experimental.)

The functions in the following group are only available for **polygons**. They have no counterpart for **rat\_polygons**.

**polygon** \( P.\text{translate\_by\_angle}(\text{double alpha}, \text{double d}) \) returns \( P \) translated in direction \( \alpha \) by distance \( d \).
CHAPTER 15. BASIC DATA TYPES FOR TWO-DIMENSIONAL GEOMETRY

**polygon**

\[ P.\text{rotate}(\text{const point} & p, \text{double alpha}) \]

returns \( P \) rotated by \( \alpha \) degrees about \( p \).

**polygon**

\[ P.\text{rotate}(\text{double alpha}) \]

returns \( P \) rotated by \( \alpha \) degrees about the origin.

**double**

\[ P.\text{distance}(\text{const point} & p) \]

returns the Euclidean distance between \( P \) and \( p \).

**rat_polygon**

\[ P.\text{to_rational}(\text{int prec} = -1) \]

returns a representation of \( P \) with rational coordinates with precision \( \text{prec} \) (cf. Section 15.10).

All functions below assume that \( P \) is weakly simple.

**int**

\[ P.\text{side_of}(\text{const POINT} & p) \]

returns +1 if \( p \) lies to the left of \( P \), 0 if \( p \) lies on \( P \), and −1 if \( p \) lies to the right of \( P \).

**region_kind**

\[ P.\text{region_of}(\text{const POINT} & p) \]

returns BOUNDED_REGION if \( p \) lies in the bounded region of \( P \), returns ON_REGION if \( p \) lies on \( P \), and returns UNBOUNDED_REGION if \( p \) lies in the unbounded region.

**bool**

\[ P.\text{inside}(\text{const POINT} & p) \]

returns true if \( p \) lies to the left of \( P \), i.e., \( \text{side_of}(p) == +1 \).

**bool**

\[ P.\text{on_boundary}(\text{const POINT} & p) \]

returns true if \( p \) lies on \( P \), i.e., \( \text{side_of}(p) == 0 \).

**bool**

\[ P.\text{outside}(\text{const POINT} & p) \]

returns true if \( p \) lies to the right of \( P \), i.e., \( \text{side_of}(p) == -1 \).

**bool**

\[ P.\text{contains}(\text{const POINT} & p) \]

returns true if \( p \) lies to the left of or on \( P \).

**RAT_TYPE**

\[ P.\text{area}() \]

returns the signed area of the bounded region of \( P \). The sign of the area is positive if the bounded region is the positive side of \( P \).

**int**

\[ P.\text{orientation}() \]

returns the orientation of \( P \).
void \( P.bounding\_box(\text{POINT} \& \ x_{\text{min}}, \text{POINT} \& \ y_{\text{min}}, \text{POINT} \& \ x_{\text{max}}, \text{POINT} \& \ y_{\text{max}}) \)

returns the coordinates of a rectangular bounding box of \( P \).

**Iterations Macros**

\texttt{forall\_vertices}(v, P) \{ “the vertices of \( P \) are successively assigned to rat\_point \( v \)” \}

\texttt{forall\_segments}(s, P) \{ “the edges of \( P \) are successively assigned to rat\_segment \( s \)” \}

**Non-Member Functions**

\texttt{POLYGON reg\_gon(int \( n \), CIRCLE \( C \), double \( \epsilon \))}

generates a (nearly) regular \( n \)-gon whose vertices lie on the circle \( C \). The \( i \)-th point is generated by \( C.\text{point\_of\_circle}(2\pi i/n, \epsilon) \). With the rational kernel the vertices of the \( n \)-gon are guaranteed to lie on the circle, with the floating point kernel they are only guaranteed to lie near \( C \).

\texttt{POLYGON n\_gon(int \( n \), CIRCLE \( C \), double \( \epsilon \))}

generates a (nearly) regular \( n \)-gon whose vertices lie near the circle \( C \). For the floating point kernel the function is equivalent to the function above. For the rational kernel the function first generates a \( n \)-gon with floating point arithmetic and then converts the resulting \textit{polygon} to a \textit{rat\_polygon}.

\texttt{POLYGON hilbert(int \( n \), RAT\_TYPE \( x_{1} \), RAT\_TYPE \( y_{1} \), RAT\_TYPE \( x_{2} \), RAT\_TYPE \( y_{2} \))}

generates the Hilbert polygon of order \( n \) within the rectangle with boundary \((x_{1}, y_{1})\) and \((x_{2}, y_{2})\).

\textit{Precondition:} \( x_{1} < x_{2} \) and \( y_{1} < y_{2} \).
15.7 Generalized Polygons (GEN_POLYGON)

1. Definition

There are three instantiations of POLYGON: gen_polygon (floating point kernel), rat_gen_polygon (rational kernel) and real_gen_polygon (real kernel). The respective header file name corresponds to the type name (with "h" appended).

An instance $P$ of the data type GEN_POLYGON is a regular polygonal region in the plane. A regular region is an open set that is equal to the interior of its closure. A region is polygonal if its boundary consists of a finite number of line segments.

The boundary of a GEN_POLYGON consists of zero or more weakly simple closed polygonal chains. There are two regions whose boundary is empty, namely the empty region and the full region. The full region encompasses the entire plane. We call a region non-trivial if its boundary is non-empty. The boundary cycles $P_1, P_2, \ldots, P_k$ of a GEN_POLYGON are ordered such that no $P_i$ is nested in a $P_j$ with $i < j$.

Only the types rat_polygon and real_polygon guarantee correct results. Almost all operations listed below are available for all the three instantiations of POLYGON. There is a small number of operations that are only available for polygon, they are indicated as such.

A detailed discussion of polygons and generalized polygons can be found in the LEDA book.

The local enumeration type KIND consists of elements EMPTY, FULL, and NON_TRIVIAL.

```c
#include <LEDA/geo/generic/GEN_POLYGON.h>
```

2. Types

```cpp
GEN_POLYGON::coord_type
    the coordinate type (e.g. rational).
GEN_POLYGON::point_type
    the point type (e.g. rat_point).
GEN_POLYGON::segment_type
    the segment type (e.g. rat_segment).
GEN_POLYGON::polygon_type
    the polygon type (e.g. rat_polygon).
GEN_POLYGON::float_type
    the corresponding floating-point type (gen_polygon).
```
3. Creation

 GEN_POLYGON P(KIND k = GEN_POLYGON::REP::EMPTY);
 introduces a variable $P$ of type GEN_POLYGON. $P$ is initialized to the empty polygon if $k$ is EMPTY and to the full polygon if $k$ is FULL.

 GEN_POLYGON P(const POLYGON & p,
  CHECK_TYPE check = WEAKLY_SIMPLE,
  RESPECT_TYPE respect_orientation = RESPECT_ORIENTATION);
 introduces a variable $P$ of type GEN_POLYGON. $P$ is initialized to the polygonal region with boundary $p$. If $respect_orientation$ is DISREGARD_ORIENTATION, the orientation is chosen such $P$ is bounded.
 Precondition: $p$ must be a weakly simple polygon. If $check$ is set appropriately this is checked.

 GEN_POLYGON P(const list<POINT> & pl,
  CHECK_TYPE check = GEN_POLYGON::WEAKLY_SIMPLE,
  RESPECT_TYPE respect_orientation = RESPECT_ORIENTATION);
 introduces a variable $P$ of type GEN_POLYGON. $P$ is initialized to the polygon with vertex sequence $pl$. If $respect_orientation$ is DISREGARD_ORIENTATION, the orientation is chosen such that $P$ is bounded.
 Precondition: If $check$ is SIMPLE, $pl$ must define a simple polygon, and if $check$ is WEAKLY_SIMPLE, $pl$ must define a weakly simple polygon. If no test is to performed, the second argument has to be set to NO_CHECK. The three constants NO_CHECK, SIMPLE, and WEAKLY_SIMPLE are part of a local enumeration type CHECK_TYPE.

 GEN_POLYGON P(const list<POLYGON> & PL,
  CHECK_TYPE check = CHECK_REP);
 introduces a variable $P$ of type GEN_POLYGON. $P$ is initialized to the polygon with boundary representation $PL$.
 Precondition: $PL$ must be a boundary representation. This conditions is checked if $check$ is set to CHECK_REP.

 GEN_POLYGON P(const list<GEN_POLYGON> & PL);
 introduces a variable $P$ of type GEN_POLYGON. $P$ is initialized to the union of all generalized polygons in $PL$. 

CHAPTER 15. BASIC DATA TYPES FOR TWO-DIMENSIONAL GEOMETRY

GEN_POLYGON $P(const$ gen_polygon& $Q,$ int $prec = \text{rat\_point}::\text{default\_precision}$);

introduces a variable $P$ of type GEN_POLYGON. $P$ is initialized to a rational approximation of the (floating point) polygon $Q$ of coordinates with denominator at most $prec$. If $prec$ is zero, the implementation chooses $prec$ large enough such that there is no loss of precision in the conversion.

4. Operations

```
bool $P$.empty() returns true if $P$ is empty, false otherwise.
bool $P$.full() returns true if $P$ is the entire plane, false otherwise.
bool $P$.trivial() returns true if $P$ is either empty or full, false otherwise.
bool $P$.is_convex() returns true if $P$ is convex, false otherwise.
KIND $P$.kind() returns the kind of $P$.
gen_polygon $P$.to_float() returns a floating point approximation of $P$.
void $P$.normalize() simplifies the homogenous representation by calling $p$.normalize() for every vertex $p$ of $P$.
bool $P$.is_simple() returns true if the polygonal region is simple, i.e., if the graph defined by the segments in the boundary of $P$ has only vertices of degree two.
bool $GEN\_POLYGON::\text{check\_representation}$($const$ list<POLYGON>& $PL$) checks whether $PL$ is a boundary representation.
bool $P$.check_representation() tests whether the representation of $P$ is OK. This test is partial.
void $P$.canonical_rep() NOT IMPLEMENTED YET.
list<POINT> $P$.vertices() returns the concatenated vertex lists of all polygons in the boundary representation of $P$.
```
15.7. GENERALIZED POLYGONS ( GEN_POLYGON )

list<SEGMENT> P.edges() returns the concatenated edge lists of all polygons in the boundary representation of P.

Please note that it is not safe to use this function in a forall-loop. Instead of writing forall(SEGMENT s, edges())...
please write list<SEGMENT> L = edges(); forall(SEGMENT s, L)....

const list<POLYGON>& P.polygons() returns the lists of all polygons in the boundary representation of P.

list<POINT> P.intersection(const SEGMENT& s) returns the list of all proper intersections between s and the boundary of P.

list<POINT> P.intersection(const LINE& l) returns the list of all proper intersections between l and the boundary of P.

int P.size() returns the number of segments in the boundary of P.

GEN_POLYGON P.translate(RAT_TYPE dx, RAT_TYPE dy) returns P translated by vector (dx, dy).

GEN_POLYGON P.translate(INT_TYPE dx, INT_TYPE dy, INT_TYPE dw) returns P translated by vector (dx/dw, dy/dw).

GEN_POLYGON P.translate(const VECTOR& v) returns P translated by vector v.

GEN_POLYGON P + const VECTOR& v returns P translated by vector v.

GEN_POLYGON P − const VECTOR& v returns P translated by vector −v.

GEN_POLYGON P.rotate90(const POINT& q, int i = 1) returns P rotated about q by an angle of i × 90 degrees. If i > 0 the rotation is counterclockwise otherwise it is clockwise.

GEN_POLYGON P.reflect(const POINT& p, const POINT& q) returns P reflected across the straight line passing through p and q.

GEN_POLYGON P.reflect(const POINT& p) returns P reflected across point p.
RAT_TYPE  P.sqr_dist(const POINT & p)
            returns the square of the minimal Euclidean
distance between a segment in the boundary
of P and p. Returns zero is P is trivial.

GEN_POLYGON  P.make_weakly_simple(bool with_neg_parts = true,
                                      bool strict = false)
            creates a weakly simple generalized poly-
gon Q from a possibly non-simple polygon
P such that Q and P have the same inner
points. The flag with_neg_parts determines
whether inner points in negatively oriented
parts are taken into account, too. If strict is
true a point is considered an inner point if
it is left of all surrounding segments, other-
wise it is considered as an inner point if it is
locally to the left of some surrounding edge.
(This function is experimental.)

GEN_POLYGON  GEN_POLYGON::make_weakly_simple(const POLYGON & Q,
                                               bool with_neg_parts = true,
                                               bool strict = false)
            same as above but the input is a polygon Q.
            (This function is experimental.)

GEN_POLYGON  P.complement()
            returns the complement of P.

GEN_POLYGON  P.eliminate_colinear_vertices()
            returns a copy of P without colinear vertices.

int  P.side_of(const POINT & p)
            returns +1 if p lies to the left of P, 0 if p lies
on P, and −1 if p lies to the right of P.

region_kind  P.region_of(const POINT & p)
            returns BOUNDED_REGION if p lies
in the bounded region of P, returns
ON_REGION if p lies on P, and returns
UNBOUNDED_REGION if p lies in the un-
bounded region. The bounded region of the
full polygon is the entire plane.

bool  P.inside(const POINT & p)
            returns true if p lies to the left of P, i.e.,
            side_of(p) == +1.
bool \( P.\text{on\_boundary}(\text{const POINT} & p) \)
returns true if \( p \) lies on \( P \), i.e., \( \text{side\_of}(p) == 0 \).

bool \( P.\text{outside}(\text{const POINT} & p) \)
returns true if \( p \) lies to the right of \( P \), i.e., \( \text{side\_of}(p) == -1 \).

bool \( P.\text{contains}(\text{const POINT} & p) \)
returns true if \( p \) lies to the left of or on \( P \).

RAT\_TYPE \( P.\text{area}() \)
returns the signed area of the bounded region of \( P \). The sign of the area is positive if the bounded region is the positive side of \( P \). Precondition: \( P \) is not the full polygon.

int \( P.\text{orientation}() \)
returns the orientation of \( P \).

list<GEN\_POLYGON> \( P.\text{regional\_decomposition}() \)
computes a decomposition of the bounded region of \( P \) into simple connected components \( P_1, \ldots, P_n \). If \( P \) is trivial the decomposition is \( P \) itself. Otherwise, the boundary of every \( P_i \) consists of an exterior polygon and zero or more holes nested inside. But the holes do not contain any nested polygons. (Note that \( P \) may have holes containing nested polygons; they appear as separate components in the decomposition.) Every \( P_i \) has the same orientation as \( P \). If it is positive then \( P \) is the union of \( P_1, \ldots, P_n \), otherwise \( P \) is the intersection of \( P_1, \ldots, P_n \).

GEN\_POLYGON \( P.\text{buffer}(\text{RAT\_TYPE} \, d) \)
adds an exterior buffer zone to \( P \) (\( d > 0 \)), or removes an interior buffer zone from \( P \) (\( d < 0 \)). More precisely, for \( d \geq 0 \) define the buffer tube \( T \) as the set of all points in the complement of \( P \) whose distance to \( P \) is at most \( d \). Then the function returns \( P \cup T \). For \( d < 0 \) let \( T \) denote the set of all points in \( P \) whose distance to the complement is less than \( |d| \). Then the result is \( P \setminus T \). (This function is experimental.)

All binary boolean operations are regularized, i.e., the result \( R \) of the standard boolean operation is replaced by the interior of the closure of \( R \). We use \( \text{reg\_X} \) to denote the regularization of a set \( X \).
GEN_POLYGON  P.unite(const GEN_POLYGON& Q) 
returns reg(P ∪ Q).

GEN_POLYGON  P.intersection(const GEN_POLYGON& Q) 
returns reg(P ∩ Q).

GEN_POLYGON  P.diff(const GEN_POLYGON& Q) 
returns reg(P \ Q).

GEN_POLYGON  P.sym_diff(const GEN_POLYGON& Q) 
returns reg((P ∪ Q) − (P ∩ Q)).

The following functions are only available for gen_polygons. They have no counterpart for rat_gen_polygons or real_gen_polygons.

gen_polygon  P.translate_by_angle(double alpha, double d) 
returns P translated in direction alpha by distance d.

gen_polygon  P.rotate(const point& p, double alpha) 
returns P rotated by α degrees about p.

gen_polygon  P.rotate(double alpha) 
returns P rotated by α degrees about the origin.

double  P.distance(const point& p) 
returns the Euclidean distance between P and p.

rat_gen_polygon  P.to_rational(int prec = −1) 
returns a representation of P with rational coordinates with precision prec (cf. Section 15.10).

Iterations Macros

forall_polygons(p, P) { “the boundary polygons of P are successively assigned to POLYGON p” }
15.8 Triangles (triangle)

1. Definition

An instance \( t \) of the data type \( \text{triangle} \) is an oriented triangle in the two-dimensional plane. A triangle splits the plane into one bounded and one unbounded region. If the triangle is positively oriented, the bounded region is to the left of it, if it is negatively oriented, the unbounded region is to the left of it. A triangle \( t \) is called degenerate, if the 3 vertices of \( t \) are collinear.

#include <LEDAt/geo/triangle.h>

2. Types

\textit{triangle::coord\_type} the coordinate type (\textit{double}).

\textit{triangle::point\_type} the point type (\textit{point}).

3. Creation

\textit{triangle \ t; \ } introduces a variable \( t \) of type \textit{triangle}. \( t \) is initialized to the empty triangle.

\textit{triangle \ t(const point\& p, const point\& q, const point\& r);} 

\textit{introduces a variable \( t \) of type \textit{triangle}. \( t \) is initialized to the triangle \([p,q,r]\).}

\textit{triangle \ t(double x1, double y1, double x2, double y2, double x3, double y3);} 

\textit{introduces a variable \( t \) of type \textit{triangle}. \( t \) is initialized to the triangle \([(x1,y1),(x2,y2),(x3,y3)]\).}

4. Operations

\textit{point \ t.point1()} \ returns the first vertex of triangle \( t \).

\textit{point \ t.point2()} \ returns the second vertex of triangle \( t \).

\textit{point \ t.point3()} \ returns the third vertex of triangle \( t \).

\textit{point \ t[int i]} \ returns the \( i \)-th vertex of \( t \). \textit{Precondition:} \( 1 \leq i \leq 3 \).

\textit{int \ t.orientation()} \ returns the orientation of \( t \).

\textit{double \ t.area()} \ returns the signed area of \( t \) (positive, if \( \text{orientation}(a,b,c) > 0 \), negative otherwise).

\textit{bool \ t.is\_degenerate()} \ returns true if the vertices of \( t \) are collinear.
int side_of(const point& p)
returns +1 if p lies to the left of t, 0 if p lies on t and −1 if p lies to the right of t.

region_kind region_of(const point& p)
returns BOUNDED_REGION if p lies in the bounded region of t, ON_REGION if p lies on t and UNBOUNDED_REGION if p lies in the unbounded region.

bool inside(const point& p)
returns true, if p lies to the left of t.

bool outside(const point& p)
returns true, if p lies to the right of t.

bool on_boundary(const point& p)
decides whether p lies on the boundary of t.

bool contains(const point& p)
decides whether t contains p.

bool intersection(const line& l)
decides whether the bounded region or the boundary of t and l intersect.

bool intersection(const segment& s)
decides whether the bounded region or the boundary of t and s intersect.

triangle translate(double dx, double dy)
returns t translated by vector (dx, dy).

triangle translate(const vector& v)
returns t + v, i.e., t translated by vector v.
Precondition: v.dim() = 2.

triangle t + const vector& v
returns t translated by vector v.

triangle t − const vector& v
returns t translated by vector −v.

triangle rotate(const point& q, double a)
returns t rotated about point q by angle a.

triangle rotate(double alpha)
returns t.rotate(t.point1(), alpha).
**15.8. TRIANGLES ( TRIANGLE )**

```
triangle t.rotate90(const point& q, int i = 1)
     returns t rotated about q by an angle of $i \times 90$ degrees. If $i > 0$ the rotation is counter-clockwise otherwise it is clockwise.

triangle t.rotate90(int i = 1)
     returns t.rotate90(t.source(),i).
```

```
triangle t.reflect(const point& p, const point& q)
     returns t reflected across the straight line passing through p and q.
```

```
triangle t.reflect(const point& p)
     returns t reflected across point p.
```

```
triangle t.reverse()
     returns t reversed.
```
15.9 Iso-oriented Rectangles (rectangle)

1. Definition
An instance \( r \) of the data type \textit{rectangle} is an iso-oriented rectangle in the two-dimensional plane.

\#include \texttt{<LEDA/geo/rectangle.h>}

2. Creation

\textit{rectangle} \( r \)(\texttt{const point} & \( p \), \texttt{const point} & \( q \));

introduces a variable \( r \) of type \textit{rectangle}. \( r \) is initialized to the \textit{rectangle} with diagonal corners \( p \) and \( q \).

\textit{rectangle} \( r \)(\texttt{const point} & \( p \), \texttt{double} \( w \), \texttt{double} \( h \));

introduces a variable \( r \) of type \textit{rectangle}. \( r \) is initialized to the \textit{rectangle} with lower left corner \( p \), width \( w \) and height \( h \).

\textit{rectangle} \( r \)(\texttt{double} \( x1 \), \texttt{double} \( y1 \), \texttt{double} \( x2 \), \texttt{double} \( y2 \));

introduces a variable \( r \) of type \textit{rectangle}. \( r \) is initialized to the \textit{rectangle} with diagonal corners \((x1, y1)\) and \((x2, y2)\).

3. Operations

\textit{point} \( r \).upper\_left() returns the upper left corner.

\textit{point} \( r \).upper\_right() returns the upper right corner.

\textit{point} \( r \).lower\_left() returns the lower left corner.

\textit{point} \( r \).lower\_right() returns the lower right corner.

\textit{point} \( r \).center() returns the center of \( r \).

\textit{list<point>} \( r \).vertices() returns the vertices of \( r \) in counterclockwise order starting from the lower left point.

\textit{double} \( r \).xmin() returns the minimal \( x \)-coordinate of \( r \).

\textit{double} \( r \).xmax() returns the maximal \( x \)-coordinate of \( r \).

\textit{double} \( r \).ymin() returns the minimal \( y \)-coordinate of \( r \).

\textit{double} \( r \).ymax() returns the maximal \( y \)-coordinate of \( r \).

\textit{double} \( r \).width() returns the width of \( r \).
double \( r.\text{height}(\text{)} \) returns the height of \( r \).

bool \( r.\text{is\_degenerate}(\text{)} \) returns true, if \( r \) degenerates to a segment or point (the 4 corners are collinear), false otherwise.

bool \( r.\text{is\_point}(\text{)} \) returns true, if \( r \) degenerates to a point.

bool \( r.\text{is\_segment}(\text{)} \) returns true, if \( r \) degenerates to a segment.

int \( r.\text{cs\_code}(\text{const point}& p) \) returns the code for Cohen-Sutherland algorithm.

bool \( r.\text{inside}(\text{const point}& p) \) returns true, if \( p \) is inside of \( r \), false otherwise.

bool \( r.\text{outside}(\text{const point}& p) \) returns true, if \( p \) is outside of \( r \), false otherwise.

bool \( r.\text{inside\_or\_contains}(\text{const point}& p) \)
returns true, if \( p \) is inside of \( r \) or on the border, false otherwise.

bool \( r.\text{contains}(\text{const point}& p) \) returns true, if \( p \) is on the border of \( r \), false otherwise.

region\_kind \( r.\text{region\_of}(\text{const point}& p) \)
returns BOUNDED\_REGION if \( p \) lies in the bounded region of \( r \), returns ON\_REGION if \( p \) lies on \( r \), and returns UNBOUNDED\_REGION if \( p \) lies in the unbounded region.

rectangle \( r.\text{include}(\text{const point}& p) \) returns a new rectangle that includes the points of \( r \) and \( p \).

rectangle \( r.\text{include}(\text{const rectangle}& r2) \)
returns a new rectangle that includes the points of \( r \) and \( r2 \).

rectangle \( r.\text{translate}(\text{double } dx, \text{ double } dy) \)
returns a new rectangle that is the translation of \( r \) by \((dx, dy)\).

rectangle \( r.\text{translate}(\text{const vector}& v) \)
returns a new rectangle that is the translation of \( r \) by \( v \).

rectangle \( r + \text{const vector}& v \)
returns \( r \) translated by \( v \).

rectangle \( r - \text{const vector}& v \)
returns \( r \) translated by \(-v\).

point \( r[\text{int } i] \) returns the \( i \text{-th} \) vertex of \( r \). Precondition: \( 0 < i < 5 \).
CHAPTER 15. BASIC DATA TYPES FOR TWO-DIMENSIONAL GEOMETRY

rectangle    r.rotate90(const point& p, int i = 1)
returns r rotated about p by an angle of i × 90
degrees. If i > 0 the rotation is counter-clockwise
otherwise it is clockwise.

rectangle    r.rotate90(int i = 1)
returns r rotated by an angle of i × 90 degrees
about the origin.

rectangle    r.reflect(const point& p)
returns r reflected across p.

list<point>  r.intersection(const segment& s)
returns r ∩ s.

bool          r.clip(const segment& t, segment& inter)
clips t on r and returns the result in inter.

bool          r.clip(const line& l, segment& inter)
clips l on r and returns the result in inter.

bool          r.clip(const ray& ry, segment& inter)
clips ry on r and returns the result in inter.

bool          r.difference(const rectangle& q, list<rectangle>& L)
returns true iff the difference of r and q is not
empty, and false otherwise. The difference L is
returned as a partition into rectangles.

list<point>  r.intersection(const line& l)
returns r ∩ l.

list<rectangle>  r.intersection(const rectangle& s)
returns r ∩ s.

bool          r.do_intersect(const rectangle& b)
returns true iff r and b intersect, false otherwise.

double         r.area()
returns the area of r.
15.10 Rational Points ( rat_point )

1. Definition

An instance of data type rat_point is a point with rational coordinates in the two-

dimensional plane. A point with cartesian coordinates \((a, b)\) is represented by ho-

mogeneous coordinates \((x, y, w)\) of arbitrary length integers (see 6.1) such that \(a = x/w\) and \(b = y/w\) and \(w > 0\).

#include <LEDA/geo/rat_point.h>

2. Types

rat_point::coord_type the coordinate type (rational).

rat_point::point_type the point type (rat_point).

rat_point::float_type the corresponding floating-point type (point).

3. Creation

rat_point p; introduces a variable \(p\) of type rat_point initialized to the point (0, 0).

rat_point p(const rational& a, const rational& b);
introduces a variable \(p\) of type rat_point initialized to the point (\(a, b\)).

rat_point p(integer a, integer b);
introduces a variable \(p\) of type rat_point initialized to the point (\(a, b\)).

rat_point p(integer x, integer y, integer w);
introduces a variable \(p\) of type rat_point initialized to the point with homogeneous coordinates \((x, y, w)\) if \(w > 0\) and to point \((-x, -y, -w)\) if \(w < 0\).

Precondition: \(w \neq 0\).

rat_point p(const rat_vector& v);
introduces a variable \(p\) of type rat_point initialized to the point \((v[0], v[1])\).

Precondition: : v.dim( ) = 2.
rat_point p(const point& p1, int prec = rat_point::default_precision);
introduces a variable p of type rat_point initialized to the point
with homogeneous coordinates ([P*x], [P*y], P), where p1 =
(x, y) and P = 2^{prec}. If prec is non-positive, the conversion
is without loss of precision, i.e., P is chosen as a sufficiently
large power of two such that P * x and P * y are integers.

rat_point p(double x, double y, int prec = rat_point::default_precision);
see constructor above with p = (x, y).

4. Operations

point p.to_float() returns a floating point approximation of p.
rat_vector p.to_vector() returns the vector extending from the origin to p.
void p.normalize() simplifies the homogenous representation by dividing all
coordinates by gcd(X, Y, W).
integer p.X() returns the first homogeneous coordinate of p.
double p.XD() returns a floating point approximation of p.X( ).
double p.YD() returns a floating point approximation of p.Y( ).
rational p.xcoord() returns the x-coordinate of p.
rational p.ycoord() returns the y-coordinate of p.
double p.xcoordD() returns a floating point approximation of p.xcoord( ).
double p.ycoordD() returns a floating point approximation of p.ycoord( ).
rat_point p.rotate90(const rat_point& q, int i = 1)
returns p rotated by i × 90 degrees about q. If i > 0 the
rotation is counter-clockwise otherwise it is clockwise.
rat_point p.rotate90(int i = 1)
returns p rotated by i×90 degrees about the origin. If i > 0
the rotation is counter-clockwise otherwise it is clockwise.
rat_point p.reflect(const rat_point& p, const rat_point& q)
returns p reflected across the straight line passing through p and q.
Precondition: p \neq q.

rat_point p.reflect(const rat_point& q)
returns p reflected across point q.

rat_point p.translate(const rational& dx, const rational& dy)
returns p translated by vector (dx, dy).

rat_point p.translate(integer dx, integer dy, integer dw)
returns p translated by vector (dx/dw, dy/dw).

rat_point p.translate(const rat_vector& v)
returns p + v, i.e., p translated by vector v.
Precondition: v.dim() = 2.

rat_point p + const rat_vector& v
returns p translated by vector v.

rat_point p − const rat_vector& v
returns p translated by vector −v.

rational p.sqr_dist(const rat_point& q)
returns the squared distance between p and q.

int p.cmp_dist(const rat_point& q, const rat_point& r)
returns compare(p.sqr_dist(q), p.sqr_dist(r)).

rational p.xdist(const rat_point& q)
returns the horizontal distance between p and q.

rational p.ydist(const rat_point& q)
returns the vertical distance between p and q.

int p.orientation(const rat_point& q, const rat_point& r)
returns orientation(p,q,r) (see below).

rational p.area(const rat_point& q, const rat_point& r)
returns area(p,q,r) (see below).

rat_vector p − const rat_point& q
returns the difference vector of the coordinates.

Non-Member Functions
\texttt{int cmp\_signed\_dist(const rat\_point\& a, const rat\_point\& b, const rat\_point\& c, const rat\_point\& d)}

compares (signed) distances of \textit{c} and \textit{d} to the straight line passing through \textit{a} and \textit{b} (directed from \textit{a} to \textit{b}). Returns +1 (−1) if \textit{c} has larger (smaller) distance than \textit{d} and 0 if distances are equal.

\texttt{int orientation(const rat\_point\& a, const rat\_point\& b, const rat\_point\& c)}

computes the orientation of points \textit{a}, \textit{b}, \textit{c} as the sign of the determinant

\[
\begin{vmatrix}
   a_x & a_y & a_w \\
   b_x & b_y & b_w \\
   c_x & c_y & c_w \\
\end{vmatrix}
\]

i.e., it returns +1 if point \textit{c} lies left of the directed line through \textit{a} and \textit{b}, 0 if \textit{a}, \textit{b}, and \textit{c} are collinear, and −1 otherwise.

\texttt{int cmp\_distances(const rat\_point\& p1, const rat\_point\& p2, const rat\_point\& p3, const rat\_point\& p4)}

compares the distances (\textit{p1}, \textit{p2}) and (\textit{p3}, \textit{p4}). Returns +1 (−1) if distance (\textit{p1}, \textit{p2}) is larger (smaller) than distance (\textit{p3}, \textit{p4}), otherwise 0.

\texttt{rat\_point midpoint(const rat\_point\& a, const rat\_point\& b)}

returns the midpoint of \textit{a} and \textit{b}.

\texttt{rational area(const rat\_point\& a, const rat\_point\& b, const rat\_point\& c)}

computes the signed area of the triangle determined by \textit{a}, \textit{b}, \textit{c}, positive if \textit{orientation}(\textit{a}, \textit{b}, \textit{c}) > 0 and negative otherwise.

\texttt{bool collinear(const rat\_point\& a, const rat\_point\& b, const rat\_point\& c)}

returns true if points \textit{a}, \textit{b}, \textit{c} are collinear, i.e., \textit{orientation}(\textit{a}, \textit{b}, \textit{c}) = 0, and false otherwise.

\texttt{bool right\_turn(const rat\_point\& a, const rat\_point\& b, const rat\_point\& c)}

returns true if points \textit{a}, \textit{b}, \textit{c} form a right turn, i.e., \textit{orientation}(\textit{a}, \textit{b}, \textit{c}) < 0, and false otherwise.

\texttt{bool left\_turn(const rat\_point\& a, const rat\_point\& b, const rat\_point\& c)}

returns true if points \textit{a}, \textit{b}, \textit{c} form a left turn, i.e., \textit{orientation}(\textit{a}, \textit{b}, \textit{c}) > 0, and false otherwise.
int side_of_halfspace(const rat_point& a, const rat_point& b, const rat_point& c)
returns the sign of the scalar product \((b-a)\cdot(c-a)\). If \(b \neq a\) this amounts to: Let \(h\) be the open halfspace orthogonal to the vector \(b-a\), containing \(b\), and having \(a\) in its boundary. Returns +1 if \(c\) is contained in \(h\), returns 0 is \(c\) lies on the the boundary of \(h\), and returns \(-1\) is \(c\) is contained in the interior of the complement of \(h\).

int side_of_circle(const rat_point& a, const rat_point& b, const rat_point& c, const rat_point& d)
returns +1 if point \(d\) lies left of the directed circle through points \(a\), \(b\), and \(c\), 0 if \(a, b, c,\) and \(d\) are cocircular, and \(-1\) otherwise.

bool incircle(const rat_point& a, const rat_point& b, const rat_point& c, const rat_point& d)
returns true if point \(d\) lies in the interior of the circle through points \(a\), \(b\), and \(c\), and false otherwise.

bool outcircle(const rat_point& a, const rat_point& b, const rat_point& c, const rat_point& d)
returns true if point \(d\) lies outside of the circle through points \(a\), \(b\), and \(c\), and false otherwise.

bool on_circle(const rat_point& a, const rat_point& b, const rat_point& c, const rat_point& d)
returns true if points \(a\), \(b\), \(c\), and \(d\) are cocircular.

bool cocircular(const rat_point& a, const rat_point& b, const rat_point& c, const rat_point& d)
returns true if points \(a\), \(b\), \(c\), and \(d\) are cocircular.

int compare_by_angle(const rat_point& a, const rat_point& b, const rat_point& c, const rat_point& d)
compares vectors \(b-a\) and \(d-c\) by angle (more efficient than calling vector::compare_by_angle\((b-a), d-x)\) on rat_vectors).

bool affinely_independent(const array<rat_point>& A)
decides whether the points in \(A\) are affinely independent.

bool contained_in_simplex(const array<rat_point>& A, const rat_point& p)
determines whether \(p\) is contained in the simplex spanned by the points in \(A\). \(A\) may consist of up to 3 points.
Precondition: The points in \(A\) are affinely independent.

bool contained_in_affine_hull(const array<rat_point>& A, const rat_point& p)
determines whether \(p\) is contained in the affine hull of the points in \(A\).
15.11 Rational Segments (rat_segment)

1. Definition

An instance \( s \) of the data type `rat_segment` is a directed straight line segment in the two-dimensional plane, i.e., a line segment \([p, q]\) connecting two rational points \(p\) and \(q\) (cf. 15.10). \(p\) is called the source or start point and \(q\) is called the target or end point of \(s\). A segment is called trivial if its source is equal to its target.

```c
#include <LEDA/geo/rat_segment.h>
```

2. Types

- `rat_segment::coord_type`  
  the coordinate type (rational).
- `rat_segment::point_type`  
  the point type (rat_point).
- `rat_segment::float_type`  
  the corresponding floatin-point type (segment).

3. Creation

```c
rat_segment s;  
```

introduces a variable \( s \) of type `rat_segment`. \( s \) is initialized to the empty segment.

```c
rat_segment s(const rat_point& p, const rat_point& q);
```

introduces a variable \( s \) of type `rat_segment`. \( s \) is initialized to the segment \([p, q]\).

```c
rat_segment s(const rat_point& p, const rat_vector& v);
```

introduces a variable \( s \) of type `rat_segment`. \( s \) is initialized to the segment \([p, p + v]\).

**Precondition:** \( v . \text{dim}() = 2 \).

```c
rat_segment s(const rational& x1, const rational& y1, const rational& x2,
              const rational& y2);
```

introduces a variable \( s \) of type `rat_segment`. \( s \) is initialized to the segment \([(x_1, y_1), (x_2, y_2)]\).

```c
rat_segment s(const integer& x1, const integer& y1, const integer& w1,
              const integer& x2, const integer& y2, const integer& w2);
```

introduces a variable \( s \) of type `rat_segment`. \( s \) is initialized to the segment \([(x_1, y_1, w_1), (x_2, y_2, w_2)]\).

```c
rat_segment s(const integer& x1, const integer& y1, const integer& x2,
              const integer& y2);
```

introduces a variable \( s \) of type `rat_segment`. \( s \) is initialized to the segment \([(x_1, y_1), (x_2, y_2)]\).
rat_segment s(const segment& s1, int prec = rat_point::default_precision);

introduces a variable $s$ of type rat_segment. $s$ is initialized
to the segment obtained by approximating the two defining
points of $s_1$.

4. Operations

segment $s$.to_float() returns a floating point approximation of $s$.

void $s$.normalize() simplifies the homogenous representation by calling
source().normalize() and target().normalize().

rat_point $s$.start() returns the source point of $s$.

rat_point $s$.end() returns the target point of $s$.

rat_segment $s$.reversal() returns the segment (target(), source()).

rational $s$.xcoord1() returns the $x$-coordinate of the source point of $s$.

rational $s$.xcoord2() returns the $x$-coordinate of the target point of $s$.

rational $s$.ycoord1() returns the $y$-coordinate of the source point of $s$.

rational $s$.ycoord2() returns the $y$-coordinate of the target point of $s$.

double $s$.xcoord1D() returns a double precision approximation of $s$.xcoord1().

double $s$.xcoord2D() returns a double precision approximation of $s$.xcoord2().

double $s$.ycoord1D() returns a double precision approximation of $s$.ycoord1().

double $s$.ycoord2D() returns a double precision approximation of $s$.ycoord2().

integer $s$.X1() returns the first homogeneous coordinate of the source
point of $s$.

integer $s$.X2() returns the first homogeneous coordinate of the target
point of $s$.

integer $s$.Y1() returns the second homogeneous coordinate of the source
point of $s$.

integer $s$.Y2() returns the second homogeneous coordinate of the target
point of $s$.

integer $s$.W1() returns the third homogeneous coordinate of the source
point of $s$. 
integer  s.W2()  returns the third homogeneous coordinate of the target point of s.
double  s.XD1()  returns a floating point approximation of s.X1().
double  s.XD2()  returns a floating point approximation of s.X2().
double  s.YD1()  returns a floating point approximation of s.Y1().
double  s.YD2()  returns a floating point approximation of s.Y2().
double  s.WD1()  returns a floating point approximation of s.W1().
double  s.WD2()  returns a floating point approximation of s.W2().
integer  s.dx()  returns the normalized $x$-difference $X2 \cdot W1 - X1 \cdot W2$ of s.
integer  s.dy()  returns the normalized $y$-difference $Y2 \cdot W1 - Y1 \cdot W2$ of s.
double  s.dxD()  returns a floating point approximation of s.dx().
double  s.dyD()  returns a floating point approximation of s.dy().
bool  s.is_trivial()  returns true if s is trivial.
bool  s.is_vertical()  returns true if s is vertical.  
Precondition: s is non-trivial.
bool  s.is_horizontal()  returns true if s is horizontal.  
Precondition: s is non-trivial.
rational  s.slope()  returns the slope of s.  
Precondition: s is not vertical.
int  s.cmp_slope(const rat_segment& s1)  
    compares the slopes of s and s1.  
Precondition: s and s1 are non-trivial.
int  s.orientation(const rat_point& p)  
    computes orientation(a, b, p) (see below), where a $\neq$ b and a and b appear in this order on segment s.
rational  s.x.proj(rational y)  
    returns p.xcoord(), where $p \in line(s)$ with p.ycoord() = y.  
Precondition: s is not horizontal.
rational  s.y.proj(rational x)  
    returns p.ycoord(), where $p \in line(s)$ with p.xcoord() = x.  
Precondition: s is not vertical.
15.11. RATIONAL SEGMENTS ( RAT_SEGMENT )

\texttt{rational} \ s.y\_abs() \quad \text{returns the y-abscissa of \textit{line}(s), i.e., } s.y\_proj(0).

\textit{Precondition:} \ s \text{ is not vertical.}

\texttt{bool} \ s\textunderscore\texttt{contains}(\texttt{const rat\_point}& \ p) \quad \text{decides whether } s \text{ contains } p.

\texttt{bool} \ s\textunderscore\texttt{intersection}(\texttt{const rat\_segment}& \ t) \quad \text{decides whether } s \text{ and } t \text{ intersect.}

\texttt{bool} \ s\textunderscore\texttt{intersection}(\texttt{const rat\_segment}& \ t, \texttt{rat\_point}& \ p) \quad \text{decides whether } s \text{ and } t \text{ intersect. If so, some point of intersection is assigned to } p.

\texttt{bool} \ s\textunderscore\texttt{intersection}(\texttt{const rat\_segment}& \ t, \texttt{rat\_segment}& \ inter) \quad \text{decides whether } s \text{ and } t \text{ intersect. If so, the segment formed by the points of intersection is assigned to } \textit{inter}.

\texttt{bool} \ s\textunderscore\texttt{intersection\_of\_lines}(\texttt{const rat\_segment}& \ t, \texttt{rat\_point}& \ p) \quad \text{decides if the lines supporting } s \text{ and } t \text{ intersect in a single point. If so, the point of intersection is assigned to } p.

\textit{Precondition:} \ s \text{ and } t \text{ are nontrivial.}

\texttt{bool} \ s\textunderscore\texttt{overlaps}(\texttt{const rat\_segment}& \ t) \quad \text{decides whether } s \text{ and } t \text{ overlap, i.e. they have a non-trivial intersection.}

\texttt{rat\_segment} \ s\textunderscore\texttt{translate}(\texttt{const rational}& dx, \texttt{const rational}& dy) \quad \text{returns } s \text{ translated by vector } (dx, dy).

\texttt{rat\_segment} \ s\textunderscore\texttt{translate}(\texttt{const integer}& dx, \texttt{const integer}& dy, \texttt{const integer}& dw) \quad \text{returns } s \text{ translated by vector } (dx/dw, dy/dw).

\texttt{rat\_segment} \ s\textunderscore\texttt{translate}(\texttt{const rat\_vector}& v) \quad \text{returns } s + v, \text{ i.e., } s \text{ translated by vector } v.

\textit{Precondition:} \ v\textunderscore\texttt{dim}() = 2.

\texttt{rat\_segment} \ s + \texttt{const rat\_vector}& v \quad \text{returns } s \text{ translated by vector } v.

\texttt{rat\_segment} \ s - \texttt{const rat\_vector}& v \quad \text{returns } s \text{ translated by vector } -v.

\texttt{rat\_segment} \ s\textunderscore\texttt{rotate90}(\texttt{const rat\_point}& q, \texttt{int} i = 1) \quad \text{returns } s \text{ rotated about } q \text{ by an angle of } i \times 90 \text{ degrees. If } i > 0 \text{ the rotation is counter-clockwise otherwise it is clockwise.}
**CHAPTER 15. BASIC DATA TYPES FOR TWO-DIMENSIONAL GEOMETRY**

`rat_segment s.rotate90(int i = 1)`

returns $s$ rotated about the origin by an angle of $i \times 90$ degrees.

`rat_segment s.reflect(const rat_point& p, const rat_point& q)`

returns $s$ reflected across the straight line passing through $p$ and $q$.

`rat_segment s.reflect(const rat_point& p)`

returns $s$ reflected across point $p$.

`rat_segment s.reverse()`

returns $s$ reversed.

`rat_segment s.perpendicular(const rat_point& p)`

returns the segment perpendicular to $s$ with source $p$ and target on $line(s)$.

*Precondition*: $s$ is nontrivial.

`rational s.sqr_length()`

returns the square of the length of $s$.

`rational s.sqr_dist(const rat_point& p)`

returns the squared Euclidean distance between $p$ and $s$.

`rational s.sqr_dist()`

returns the squared distance between $s$ and the origin.

`rat_vector s.to_vector()`

returns the vector $s.target() - s.source()$.

`bool s == const rat_segment& t`  

returns true if $s$ and $t$ are equal as oriented segments

`int equal_as_sets(const rat_segment& s, const rat_segment& t)`

returns true if $s$ and $t$ are equal as unoriented segments

**Non-Member Functions**

`int cmp_slopes(const rat_segment& s1, const rat_segment& s2)`

returns compare(slope($s_1$), slope($s_2$)).

`int cmp_segments_at_xcoord(const rat_segment& s1, const rat_segment& s2, const rat_point& p)`

compares points $l_1 \cap v$ and $l_2 \cap v$ where $l_i$ is the line underlying segment $s_i$ and $v$ is the vertical straight line passing through point $p$.

`int orientation(const rat_segment& s, const rat_point& p)`

computes orientation($a, b, p$), where $a \neq b$ and $a$ and $b$ appear in this order on segment $s$. 
15.12 Rational Rays (rat_ray)

1. Definition

An instance $r$ of the data type `rat_ray` is a directed straight ray defined by two points with rational coordinates in the two-dimensional plane.

```cpp
#include <LEDA/geo/rat_ray.h>
```

2. Types

- `rat_ray::coord_type` the coordinate type (rational).
- `rat_ray::point_type` the point type (`rat_point`).
- `rat_ray::float_type` the corresponding floating-point type (`ray`).

3. Creation

```cpp
rat_ray r(const rat_point& p, const rat_point& q);
```
introduces a variable $r$ of type `rat_ray`. $r$ is initialized to the ray starting at point $p$ and passing through point $q$.

**Precondition:** $p \neq q$.

```cpp
rat_ray r(const rat_segment& s);
```
introduces a variable $r$ of type `rat_ray`. $r$ is initialized to the `(rat_ray(s.source()), s.target())`.

**Precondition:** $s$ is nontrivial.

```cpp
rat_ray r(const rat_point& p, const rat_vector& v);
```
introduces a variable $r$ of type `rat_ray`. $r$ is initialized to `rat_ray(p, p+v)`.

```cpp
rat_ray r;
```
introduces a variable $r$ of type `rat_ray`.

```cpp
rat_ray r(const ray& r1, int prec = rat_point::default_precision);
```
introduces a variable $r$ of type `rat_ray`. $r$ is initialized to the ray obtained by approximating the two defining points of $r_1$.

4. Operations

```cpp
ray r.to_float()
```
returns a floating point approximation of $r$.

```cpp
void r.normalize()
```
simplifies the homogenous representation by calling `point1().normalize()` and `point2().normalize()`.
rat_point  r.source( ) returns the source of r.

rat_point  r.point1( ) returns the source of r.

rat_point  r.point2( ) returns a point on r different from r.source( ).

bool  r.is_vertical( ) returns true iff r is vertical.

bool  r.is_horizontal( ) returns true iff r is horizontal.

bool  r.intersection(const rat_ray& \( s \), rat_point& \( \text{inter} \)) returns true if r and s intersect. If so, a point of intersection is returned in \( \text{inter} \).

rat_ray  r.translate(const rational& \( dx \), const rational& \( dy \)) returns r translated by vector \((dx, dy)\).

rat_ray  r.translate(integer \( dx \), integer \( dy \), integer \( dw \)) returns r translated by vector \((dx/dw, dy/dw)\).

rat_ray  r.translate(const rat_vector& \( v \)) returns \( r + v \), i.e., r translated by vector \( v \).

Precondition: \( v\.\text{dim}() = 2 \).

rat_ray  r + const rat_vector& \( v \) returns r translated by vector \( v \).

rat_ray  r - const rat_vector& \( v \) returns r translated by vector \(-v\).

rat_ray  r.rotate90(const rat_point& \( q \), int \( i = 1 \)) returns r rotated about \( q \) by an angle of \( i \times 90 \) degrees. If \( i > 0 \) the rotation is counter-clockwise otherwise it is clockwise.

rat_ray  r.reflect(const rat_point& \( p \), const rat_point& \( q \)) returns r reflected across the straight line passing through \( p \) and \( q \).

Precondition: \( p \neq q \).

rat_ray  r.reflect(const rat_point& \( p \)) returns r reflected across point \( p \).

rat_ray  r.reverse( ) returns r reversed.

bool  r.contains(const rat_point& \( p \)) decides whether r contains \( p \).
\texttt{bool \hspace{1em} r.contains(const \hspace{1em} rat\_segment& \hspace{1em} s)}
\par decides whether \( r \) contains \( s \).

\textbf{Non-Member Functions}

\texttt{int \hspace{1em} orientation(const \hspace{1em} rat\_ray& \hspace{1em} r, \hspace{1em} const \hspace{1em} rat\_point& \hspace{1em} p)}
\par computes orientation\((a, \hspace{1em} b, \hspace{1em} p)\), where \( a \neq b \) and \( a \) and \( b \) appear in this order on ray \( r \).

\texttt{int \hspace{1em} cmp\_slopes(const \hspace{1em} rat\_ray& \hspace{1em} r1, \hspace{1em} const \hspace{1em} rat\_ray& \hspace{1em} r2)}
\par returns compare\((\text{slope}(r_1), \hspace{1em} \text{slope}(r_2))\).
15.13 Straight Rational Lines (rat_line)

1. Definition

An instance $l$ of the data type `rat_line` is a directed straight line in the two-dimensional plane.

```
#include <LEDA/geo/rat_line.h>
```

2. Types

- `rat_line::coord_type` the coordinate type (rational).
- `rat_line::point_type` the point type (`rat_point`).
- `rat_line::float_type` the corresponding floating-point type (`line`).

3. Creation

- `rat_line l(const rat_point& p, const rat_point& q);` introduces a variable $l$ of type `rat_line`. $l$ is initialized to the line passing through points $p$ and $q$ directed from $p$ to $q$.
  
  **Precondition:** $p \neq q$.

- `rat_line l(const rat_segment& s);` introduces a variable $l$ of type `rat_line`. $l$ is initialized to the line supporting segment $s$.
  
  **Precondition:** $s$ is nontrivial.

- `rat_line l(const rat_point& p, const rat_vector& v);` introduces a variable $l$ of type `rat_line`. $l$ is initialized to the line passing through points $p$ and $p + v$.
  
  **Precondition:** $v$ is a nonzero vector.

- `rat_line l(const rat_ray& r);` introduces a variable $l$ of type `rat_line`. $l$ is initialized to the line supporting ray $r$.

- `rat_line l;` introduces a variable $l$ of type `rat_line`.

- `rat_line l(const line& l1, int prec = rat_point::default_precision);` introduces a variable $l$ of type `rat_line`. $l$ is initialized to the line obtained by approximating the two defining points of $l1$.

4. Operations

- `line l.to_float()` returns a floating point approximation of $l$. 
void l.normalize() simplifies the homogenous representation by calling point1().normalize() and point2().normalize().

rat_point l.point1() returns a point on l.

rat_point l.point2() returns a second point on l.

rat_segment l.seg() returns a segment on l.

bool l.is_vertical() decides whether l is vertical.

bool l.is_horizontal() decides whether l is horizontal.

rational l.slope() returns the slope of s.
Precondition: l is not vertical.

rational lx_proj(rational y) returns p.xcoord(), where p ∈ line(l) with p.ycoord() = y.
Precondition: l is not horizontal.

rational ly_proj(rational x) returns p.ycoord(), where p ∈ line(l) with p.xcoord() = x.
Precondition: l is not vertical.

rational ly_abs() returns the y-abscissa of line(l), i.e., l.y_proj(0).
Precondition: l is not vertical.

bool interception(const rat_line & g, rat_point & inter)
returns true if l and g intersect. In case of intersection a common point is returned in inter.

bool Intersection(const rat_segment & s, rat_point & inter)
returns true if l and s intersect. In case of intersection a common point is returned in inter.

bool Interception(const rat_segment & s)
returns true, if l and s intersect, false otherwise.

rat_line l.translate(const rational & dx, const rational & dy) returns l translated by vector (dx, dy).

rat_line l.translate(integer dx, integer dy, integer dw) returns l translated by vector (dx/dw, dy/dw).

rat_line l.translate(const rat_vector & v) returns l translated by vector v.
Precondition: v.dim() = 2.

rat_line l + const rat_vector & v returns l translated by vector v.
rat_line l − const rat_vector& v returns l translated by vector −v.

rat_line l.rotate90(const rat_point& q, int i = 1) returns l rotated about q by an angle of i × 90 degrees. If i > 0 the rotation is counter-clockwise otherwise it is clockwise.

rat_line l.reflect(const rat_point& p, const rat_point& q) returns l reflected across the straight line passing through p and q.

rat_line l.reflect(const rat_point& p) returns l reflected across point p.

rat_line l.reverse() returns l reversed.

rational l.sqr_dist(const rat_point& q) returns the square of the distance between l and q.

rat_segment l.perpendicular(const rat_point& p) returns the segment perpendicular to l with source p and target on l.

rat_point l.dual() returns the point dual to l. Precondition: l is not vertical.

int L.orientation(const rat_point& p) computes orientation(a, b, p), where a ≠ b and a and b appear in this order on line l.

int L.side_of(const rat_point& p) computes orientation(a, b, p), where a ≠ b and a and b appear in this order on line l.

bool L.contains(const rat_point& p) returns true if p lies on l.

bool L.clip(rat_point p, rat_point q, rat_segment& s) clips l at the rectangle R defined by p and q. Returns true if the intersection of R and l is non-empty and returns false otherwise. If the intersection is non-empty the intersection is assigned to s; It is guaranteed that the source node of s is no larger than its target node.

bool l == const rat_line& g returns true if the l and g are equal as oriented lines.
\textbf{bool} \quad \texttt{equal\_as\_sets}(\texttt{const\ rat\_line}\&\ l, \texttt{const\ rat\_line}\&\ g) \\
\hspace{1cm} returns true if the $l$ and $g$ are equal as unoriented lines.

\textbf{Non-Member Functions}

\textbf{int} \quad \texttt{orientation}(\texttt{const\ rat\_line}\&\ l, \texttt{const\ rat\_point}\&\ p) \\
\hspace{1cm} computes orientation($a$, $b$, $p$), where $a \neq b$ and $a$ and $b$ appear in this order on line $l$.

\textbf{int} \quad \texttt{cmp\_slopes}(\texttt{const\ rat\_line}\&\ l1, \texttt{const\ rat\_line}\&\ l2) \\
\hspace{1cm} returns \texttt{compare(slope(l1), slope(l2))}.

\textbf{rat\_line} \quad \texttt{p\_bisector}(\texttt{const\ rat\_point}\&\ p, \texttt{const\ rat\_point}\&\ q) \\
\hspace{1cm} returns the perpendicular bisector of $p$ and $q$. The bisector has $p$ on its left. \\
\textit{Precondition:} $p \neq q$. 
15.14 Rational Circles (rat_circle)

1. Definition

An instance $C$ of data type $\text{rat}_\text{circle}$ is an oriented circle in the plane. A circle is defined by three points $p_1$, $p_2$, $p_3$ with rational coordinates ($\text{rat}_\text{points}$). The orientation of $C$ is equal to the orientation of the three defining points, i.e., $\text{orientation}(p_1, p_2, p_3)$. Positive orientation corresponds to counterclockwise orientation and negative orientation corresponds to clockwise orientation.

Some triples of points are unsuitable for defining a circle. A triple is admissable if $|\{p_1, p_2, p_3\}| \neq 2$. Assume now that $p_1$, $p_2$, $p_3$ are admissable. If $|\{p_1, p_2, p_3\}| = 1$ they define the circle with center $p_1$ and radius zero. If $p_1$, $p_2$, and $p_3$ are collinear $C$ is a straight line passing through $p_1$, $p_2$ and $p_3$ in this order and the center of $C$ is undefined. If $p_1$, $p_2$, and $p_3$ are not collinear, $C$ is the circle passing through them.

```cpp
#include <LEDA/geo/rat_circle.h>
```

2. Types

$\text{rat}_\text{circle}::\text{coord\_type}$ the coordinate type (rational).

$\text{rat}_\text{circle}::\text{point\_type}$ the point type ($\text{rat}_\text{point}$).

$\text{rat}_\text{circle}::\text{float\_type}$ the corresponding floatin-point type (circle).

3. Creation

$\text{rat}_\text{circle} C(\text{const }\text{rat}_\text{point}& a, \text{const }\text{rat}_\text{point}& b, \text{const }\text{rat}_\text{point}& c)$;

introduces a variable $C$ of type $\text{rat}_\text{circle}$. $C$ is initialized to the circle through points $a$, $b$, and $c$. 
Precondition: $a$, $b$, and $c$ are admissable.

$\text{rat}_\text{circle} C(\text{const }\text{rat}_\text{point}& a, \text{const }\text{rat}_\text{point}& b)$;

introduces a variable $C$ of type $\text{circle}$. $C$ is initialized to the counter-clockwise oriented circle with center $a$ passing through $b$.

$\text{rat}_\text{circle} C(\text{const }\text{rat}_\text{point}& a)$;

introduces a variable $C$ of type $\text{circle}$. $C$ is initialized to the trivial circle with center $a$.

$\text{rat}_\text{circle} C$;

introduces a variable $C$ of type $\text{rat}_\text{circle}$. $C$ is initialized to the trivial circle centered at $(0,0)$. 
rat_circle  C(const circle& c, int prec = rat_point::default_precision);

introduces a variable C of type rat_circle. C is initialized to the circle obtained by approximating three defining points of c.

4. Operations

circle  C.to_float()  returns a floating point approximation of C.

void  C.normalize()  simplifies the homogenous representation by normalizing $p_1$, $p_2$, and $p_3$.

int  C.orientation()  returns the orientation of C.

rat_point  C.center()  returns the center of C.

Precondition: C has a center, i.e., is not a line.

rat_point  C.point1()  returns $p_1$.

rat_point  C.point2()  returns $p_2$.

rat_point  C.point3()  returns $p_3$.

rational  C.sqr_radius()  returns the square of the radius of C.

rat_point  C.point_on_circle(double alpha, double epsilon)

returns a point $p$ on $C$ such that the angle of $p$ differs from alpha by at most epsilon.

bool  C.is_degenerate()  returns true if the defining points are collinear.

bool  C.is_trivial()  returns true if C has radius zero.

bool  C.is_line()  returns true if C is a line.

rat_line  C.to_line()  returns line(point1(), point3()).

int  C.side_of(const rat_point& p)

returns $-1$, $+1$, or $0$ if $p$ lies right of, left of, or on $C$ respectively.

bool  C.inside(const rat_point& p)

returns true iff $p$ lies inside of $C$.

bool  C.outside(const rat_point& p)

returns true iff $p$ lies outside of $C$.

bool  C.contains(const rat_point& p)

returns true iff $p$ lies on $C$. 
rat_circle C.translate(const rational& dx, const rational& dy) returns C translated by vector (dx, dy).

rat_circle C.translate(integer dx, integer dy, integer dw) returns C translated by vector (dx/dw, dy/dw).

rat_circle C.translate(const rat_vector& v) returns C translated by vector v.

rat_circle C + const rat_vector& v returns C translated by vector v.

rat_circle C − const rat_vector& v returns C translated by vector −v.

rat_circle C.rotate90(const rat_point& q, int i = 1) returns C rotated by i × 90 degrees about q. If i > 0 the rotation is counter-clockwise otherwise it is clockwise.

rat_circle C.reflect(const rat_point& p, const rat_point& q) returns C reflected across the straight line passing through p and q.

rat_circle C.reflect(const rat_point& p) returns C reflected across point p.

rat_circle C.reverse() returns C reversed.

bool C == const rat_circle& D returns true if C and D are equal as oriented circles.

bool equal_as_sets(const rat_circle& C1, const rat_circle& C2) returns true if C1 and C2 are equal as unoriented circles.

bool radical_axis(const rat_circle& C1, const rat_circle& C2, rat_line& rad_axis) if the radical axis for C1 and C2 exists, it is assigned to rad_axis and true is returned; otherwise the result is false.

ostream& ostream& out ≪ const rat_circle& c writes the three defining points.

istream& istream& in ⇒ rat_circle& c reads three points and assigns the circle defined by them to c.
15.15 Rational Triangles (rat_triangle)

1. Definition

An instance \( t \) of the data type \( \text{rat\_triangle} \) is an oriented triangle in the two-dimensional plane with rational coordinates. A \( \text{rat\_triangle} \) \( t \) splits the plane into one bounded and one unbounded region. If \( t \) is positively oriented, the bounded region is to the left of it, if it is negatively oriented, the unbounded region is to the left of it. \( t \) is called degenerate, if the 3 vertices of \( t \) are collinear.

```cpp
#include <LEDA/geo/rat_triangle.h>
```

2. Types

- \( \text{rat\_triangle}\::\text{coord\_type} \): the coordinate type (rational).
- \( \text{rat\_triangle}\::\text{point\_type} \): the point type (rat_point).

3. Creation

- \( \text{rat\_triangle} \ t; \): introduces a variable \( t \) of type \( \text{rat\_triangle} \). \( t \) is initialized to the empty triangle.
- \( \text{rat\_triangle} \ t(\text{const rat\_point}& p, \text{const rat\_point}& q, \text{const rat\_point}& r); \): introduces a variable \( t \) of type \( \text{rat\_triangle} \). \( t \) is initialized to the triangle \([p, q, r]\).
- \( \text{rat\_triangle} \ t(\text{const rational}& x1, \text{const rational}& y1, \text{const rational}& x2, \text{const rational}& y2, \text{const rational}& x3, \text{const rational}& y3); \): introduces a variable \( t \) of type \( \text{rat\_triangle} \). \( t \) is initialized to the triangle \([(x1, y1), (x2, y2), (x3, y3)]\).
- \( \text{rat\_triangle} \ t(\text{const triangle}& t, \text{int prec = rat\_point}\::\text{default\_precision}); \): introduces a variable \( t \) of type \( \text{rat\_triangle} \). \( t \) is initialized to the triangle obtained by approximating the three defining points of \( t \).

4. Operations

- \( \text{void} \ t.\text{normalize}() \): simplifies the homogenous representation by calling \( \text{p.normalize}() \) for every vertex of \( t \).
- \( \text{rat\_point} \ t.\text{point1}() \): returns the first vertex of triangle \( t \).
- \( \text{rat\_point} \ t.\text{point2}() \): returns the second vertex of triangle \( t \).
- \( \text{rat\_point} \ t.\text{point3}() \): returns the third vertex of triangle \( t \).
rat_point \( t[i] \) returns the \( i \)-th vertex of \( t \). **Precondition:** \( 1 \leq i \leq 3 \).

int \( t.\text{orientation()} \) returns the orientation of \( t \).

rational \( t.\text{area()} \) returns the signed area of \( t \) (positive, if \( \text{orientation}(a, b, c) > 0 \), negative otherwise).

bool \( t.\text{is_degenerate()} \) returns true if the vertices of \( t \) are collinear.

int \( t.\text{side_of(const rat_point& p)} \)
returns +1 if \( p \) lies to the left of \( t \), 0 if \( p \) lies on \( t \) and −1 if \( p \) lies to the right of \( t \).

region_kind \( t.\text{region_of(const rat_point& p)} \)
returns \text{BOUNDED_REGION} if \( p \) lies in the bounded region of \( t \), \text{ON_REGION} if \( p \) lies on \( t \) and \text{UNBOUNDED_REGION} if \( p \) lies in the unbounded region.

bool \( t.\text{inside(const rat_point& p)} \)
returns true, if \( p \) lies to the left of \( t \).

bool \( t.\text{outside(const rat_point& p)} \)
returns true, if \( p \) lies to the right of \( t \).

bool \( t.\text{on_boundary(const rat_point& p)} \)
decides whether \( p \) lies on the boundary of \( t \).

bool \( t.\text{contains(const rat_point& p)} \)
decides whether \( t \) contains \( p \).

bool \( t.\text{intersection(const rat_line& l)} \)
decides whether the bounded region or the boundary of \( t \) and \( l \) intersect.

bool \( t.\text{intersection(const rat_segment& s)} \)
decides whether the bounded region or the boundary of \( t \) and \( s \) intersect.

rat_triangle \( t.\text{translate(rational dx, rational dy)} \)
returns \( t \) translated by vector \((dx, dy)\).

rat_triangle \( t.\text{translate(const rat_vector& v)} \)
returns \( t + v \), i.e., \( t \) translated by vector \( v \).
 **Precondition:** \( v.\text{dim()} = 2 \).

rat_triangle \( t + \text{const rat_vector& v} \)
returns \( t \) translated by vector \( v \).
rat_triangle t - const rat_vector& v
returns t translated by vector \(-v\).

rat_triangle t.rotate90(const rat_point& q, int i = 1)
returns t rotated about q by an angle of \(i \times 90\) degrees.
If \(i > 0\) the rotation is counter-clockwise otherwise it is clockwise.

rat_triangle t.rotate90(int i = 1)
returns t.rotate90(t.source(),i).

rat_triangle t.reflect(const rat_point& p, const rat_point& q)
returns t reflected across the straight line passing through p and q.

rat_triangle t.reflect(const rat_point& p)
returns t reflected across point p.

rat_triangle t.reverse()
returns t reversed.
15.16 Iso-oriented Rational Rectangles (rat_rectangle)

1. Definition
An instance $r$ of the data type rectangle is an iso-oriented rectangle in the two-dimensional plane with rational coordinates.

```c++
#include <LEDA/geo/rat_rectangle.h>
```

2. Creation

```c++
rat_rectangle r(const rat_point& p, const rat_point& q);
```
introduces a variable $r$ of type rat_rectangle. $r$ is initialized to the rat_rectangle with diagonal corners $p$ and $q$

```c++
rat_rectangle r(const rat_point& p, rational w, rational h);
```
introduces a variable $r$ of type rat_rectangle. $r$ is initialized to the rat_rectangle with lower left corner $p$, width $w$ and height $h$.

```c++
rat_rectangle r(rational x1, rational y1, rational x2, rational y2);
```
introduces a variable $r$ of type rat_rectangle. $r$ is initialized to the rat_rectangle with diagonal corners $(x1, y1)$ and $(x2, y2)$.

```c++
rat_rectangle r(const rectangle& r, int prec = rat_point::default_precision);
```
introduces a variable $r$ of type rat_rectangle. $r$ is initialized to the rectangle obtained by approximating the defining points of $r$.

3. Operations

```c++
rectangle r.to_float() returns a floating point approximation of $R$.
```

```c++
void r.normalize() simplifies the homogenous representation by calling p.normalize() for every vertex of $r$.
```

```c++
rat_point r.upper_left() returns the upper left corner.
```

```c++
rat_point r.upper_right() returns the upper right corner.
```

```c++
rat_point r.lower_left() returns the lower left corner.
```

```c++
rat_point r.lower_right() returns the lower right corner.
```

```c++
rat_point r.center() returns the center of $r$.
```
returns the vertices of \( r \) in counterclockwise order starting from the lower left point.

\[ r \cdot \text{vertices()} \]

returns the minimal \( x \)-coordinate of \( r \).

\[ r \cdot \text{xmin()} \]

returns the maximal \( x \)-coordinate of \( r \).

\[ r \cdot \text{xmax()} \]

returns the minimal \( y \)-coordinate of \( r \).

\[ r \cdot \text{ymin()} \]

returns the maximal \( y \)-coordinate of \( r \).

\[ r \cdot \text{ymax()} \]

returns the width of \( r \).

\[ r \cdot \text{width()} \]

returns the height of \( r \).

\[ r \cdot \text{height()} \]

returns true, if \( r \) degenerates to a segment or point (the 4 corners are collinear), false otherwise.

\[ r \cdot \text{is_degenerate()} \]

returns true, if \( r \) degenerates to a point.

\[ r \cdot \text{is_point()} \]

returns true, if \( r \) degenerates to a segment.

\[ r \cdot \text{is_segment()} \]

returns the code for Cohen-Sutherland algorithm.

\[ r \cdot \text{cs_code(const rat_point& p)} \]

returns true, if \( p \) is inside of \( r \), false otherwise.

\[ r \cdot \text{inside(const rat_point& p)} \]

returns true, if \( p \) is inside of \( r \) or on the border, false otherwise.

\[ r \cdot \text{inside_or_contains(const rat_point& p)} \]

returns true, if \( p \) is outside of \( r \), false otherwise.

\[ r \cdot \text{outside(const rat_point& p)} \]

returns true, if \( p \) is on the border of \( r \), false otherwise.

\[ r \cdot \text{contains(const rat_point& p)} \]

returns \text{BOUNDED_REGION} if \( p \) lies in the bounded region of \( r \), returns \text{ON_REGION} if \( p \) lies on \( r \), and returns \text{UNBOUNDED_REGION} if \( p \) lies in the unbounded region.

\[ r \cdot \text{region_of(const rat_point& p)} \]

returns a new \text{rat_rectangle} that includes the points of \( r \) and \( p \).

\[ r \cdot \text{include(const rat_point& p)} \]
rat_rectangle r.include(const rat_rectangle& r2)
returns a new rat_rectangle that includes the points of r and r2.

rat_rectangle r.translate(rational dx, rational dy)
returns r translated by (dx, dy).

rat_rectangle r.translate(const rat_vector& v)
returns r translated by v.

rat_rectangle r + const rat_vector& v returns r translated by v.

rat_rectangle r - const rat_vector& v returns r translated by vector -v.

rat_point r[int i]
returns the i-th vertex of r. Precondition: (0 < i < 5).

rat_rectangle r.rotate90(const rat_point& p, int i = 1)
returns r rotated about q by an angle of i × 90 degrees. If i > 0 the rotation is counter-clockwise otherwise it is clockwise.

rat_rectangle r.rotate90(int i = 1) returns r rotated by an angle of i × 90 degrees about the origin.

rat_rectangle r.reflect(const rat_point& p)
returns r reflected across p.

bool r.clip(const rat_segment& t, rat_segment& inter)
clips t on r and returns the result in inter.

bool r.clip(const rat_line& l, rat_segment& inter)
clips l on r and returns the result in inter.

bool r.clip(const rat_ray& ry, rat_segment& inter)
clips ry on r and returns the result in inter.

bool r.difference(const rat_rectangle& q, list<rat_rectangle>& L)
returns true iff the difference of r and q is not empty, and false otherwise. The difference L is returned as a partition into rectangles.

list<rat_point> r.intersection(const rat_segment& s)
returns r ∩ s.

list<rat_point> r.intersection(const rat_line& l)
returns r ∩ l.
list<rat_rectangle> r.intersection(const rat_rectangle& s)
returns r ∩ s.

bool r.do_intersect(const rat_rectangle& b)
returns true iff r and b intersect, false otherwise.

rational r.area() returns the area of r.
15.17 Real Points (real_point)

1. Definition
An instance of the data type real_point is a point in the two-dimensional plane $\mathbb{R}^2$. We use $(x, y)$ to denote a real point with first (or $x$-) coordinate $x$ and second (or $y$-) coordinate $y$.

```
#include <LEDA/geo/real_point.h>
```

2. Types
- **real_point::coord_type**  
  the coordinate type (real).
- **real_point::point_type**  
  the point type (real_point).
- **real_point::float_type**  
  the corresponding floating-point type (point).

3. Creation
- **real_point p;**  
  introduces a variable $p$ of type real_point initialized to the point $(0, 0)$.
- **real_point p(real x, real y);**  
  introduces a variable $p$ of type real_point initialized to the point $(x, y)$.
- **real_point p(const point& p1, int prec = 0);**  
  introduces a variable $p$ of type real_point initialized to the point $p_1$. (The second argument is for compatibility with rat_point.)
- **real_point p(const rat_point& p1);**  
  introduces a variable $p$ of type real_point initialized to the point $p_1$.
- **real_point p(double x, double y);**  
  introduces a variable $p$ of type real_point initialized to the real point $(x, y)$.

4. Operations
- **real p.xcoord();**  
  returns the first coordinate of $p$.
- **real p.ycoord();**  
  returns the second coordinate of $p$. 
int \hspace{0.5em} p.\text{orientation}(\text{const real point}& \hspace{0.5em} q, \text{const real point}& \hspace{0.5em} r) \\
returns \text{orientation}(p, q, r) \text{ (see below).}

real \hspace{0.5em} p.\text{area}(\text{const real point}& \hspace{0.5em} q, \text{const real point}& \hspace{0.5em} r) \\
returns \text{area}(p, q, r) \text{ (see below).}

real \hspace{0.5em} p.\text{sqr\_dist}(\text{const real point}& \hspace{0.5em} q) \\
returns \text{the square of the Euclidean distance between } p \text{ and } q.

int \hspace{0.5em} p.\text{cmp\_dist}(\text{const real point}& \hspace{0.5em} q, \text{const real point}& \hspace{0.5em} r) \\
returns \text{compare}(p.\text{sqr\_dist}(q), p.\text{sqr\_dist}(r)).

real \hspace{0.5em} p.\text{xdist}(\text{const real point}& \hspace{0.5em} q) \\
returns \text{the horizontal distance between } p \text{ and } q.

real \hspace{0.5em} p.\text{ydist}(\text{const real point}& \hspace{0.5em} q) \\
returns \text{the vertical distance between } p \text{ and } q.

real \hspace{0.5em} p.\text{distance}(\text{const real point}& \hspace{0.5em} q) \\
returns \text{the Euclidean distance between } p \text{ and } q.

real \hspace{0.5em} p.\text{distance}() \\
returns \text{the Euclidean distance between } p \text{ and } (0, 0).

real\_point \hspace{0.5em} p.\text{translate}(\text{real } dx, \text{ real } dy) \\
returns p \text{ translated by vector } (dx, dy).

real\_point \hspace{0.5em} p.\text{translate}(\text{double } dx, \text{ double } dy) \\
returns p \text{ translated by vector } (dx, dy).

real\_point \hspace{0.5em} p.\text{translate}(\text{const real vector}& \hspace{0.5em} v) \\
returns p+v, \text{i.e., } p \text{ translated by vector } v. \\
\text{Precondition: } v.\dim() = 2.

real\_point \hspace{0.5em} p + \text{const real vector}& \hspace{0.5em} v \\
returns p \text{ translated by vector } v.

real\_point \hspace{0.5em} p - \text{const real vector}& \hspace{0.5em} v \\
returns p \text{ translated by vector } -v.

real\_point \hspace{0.5em} p.\text{rotate90}(\text{const real point}& \hspace{0.5em} q, \text{int } i = 1) \\
returns p \text{ rotated about } q \text{ by an angle of } i \times 90 \text{ degrees.} \\
\text{If } i > 0 \text{ the rotation is counter-clockwise otherwise it is} \\
\text{clockwise.}

real\_point \hspace{0.5em} p.\text{rotate90}(\text{int } i = 1) \hspace{0.5em} \text{returns } p.\text{rotate90}(\text{real point}(0, 0), i).
real\_point p.reflect(const real\_point& q, const real\_point& r)
returns \( p \) reflected across the straight line passing
through \( q \) and \( r \).

real\_point p.reflect(const real\_point& q)
returns \( p \) reflected across point \( q \).

real\_vector p − const real\_point& q
returns the difference vector of the coordinates.

Non-Member Functions

int cmp\_distances(const real\_point& p1, const real\_point& p2,
const real\_point& p3, const real\_point& p4)
compares the distances \((p_1, p_2)\) and \((p_3, p_4)\). Returns
+1 (−1) if distance \((p_1, p_2)\) is larger (smaller) than dis-
tance \((p_3, p_4)\), otherwise 0.

real\_point center(const real\_point& a, const real\_point& b)
returns the center of \(a\) and \(b\), i.e. \(a + \vec{ab}/2\).

real\_point midpoint(const real\_point& a, const real\_point& b)
returns the center of \(a\) and \(b\).

int orientation(const real\_point& a, const real\_point& b,
const real\_point& c)
computes the orientation of points \(a\), \(b\), and \(c\) as the
sign of the determinant

\[
\begin{vmatrix}
  a_x & a_y & 1 \\
  b_x & b_y & 1 \\
  c_x & c_y & 1
\end{vmatrix}
\]
i.e., it returns +1 if point \(c\) lies left of the directed line
through \(a\) and \(b\), 0 if \(a,b,\) and \(c\) are collinear, and −1
otherwise.

int cmp\_signed\_dist(const real\_point& a, const real\_point& b,
const real\_point& c, const real\_point& d)
compares (signed) distances of \(c\) and \(d\) to the straight
line passing through \(a\) and \(b\) (directed from \(a\) to \(b\)). Re-
turns +1 (−1) if \(c\) has larger (smaller) distance than \(d\)
and 0 if distances are equal.
real area(const real_point& a, const real_point& b, const real_point& c) computes the signed area of the triangle determined by a, b, c, positive if orientation(a, b, c) > 0 and negative otherwise.

bool collinear(const real_point& a, const real_point& b, const real_point& c) returns true if points a, b, c are collinear, i.e., orientation(a, b, c) = 0, and false otherwise.

bool right_turn(const real_point& a, const real_point& b, const real_point& c) returns true if points a, b, c form a righ turn, i.e., orientation(a, b, c) < 0, and false otherwise.

bool left_turn(const real_point& a, const real_point& b, const real_point& c) returns true if points a, b, c form a left turn, i.e., orientation(a, b, c) > 0, and false otherwise.

int side_of_halfspace(const real_point& a, const real_point& b, const real_point& c) returns the sign of the scalar product (b - a) \cdot (c - a). If b \neq a this amounts to: Let h be the open halfspace orthogonal to the vector b - a, containing b, and having a in its boundary. Returns +1 if c is contained in h, returns 0 is c lies on the the boundary of h, and returns -1 is c is contained in the interior of the complement of h.

int side_of_circle(const real_point& a, const real_point& b, const real_point& c, const real_point& d) returns +1 if point d lies left of the directed circle through points a, b, and c, 0 if a, b, c, and d are cocircular, and -1 otherwise.

bool inside_circle(const real_point& a, const real_point& b, const real_point& c, const real_point& d) returns true if point d lies in the interior of the circle through points a, b, and c, and false otherwise.

bool outside_circle(const real_point& a, const real_point& b, const real_point& c, const real_point& d) returns true if point d lies outside of the circle through points a, b, and c, and false otherwise.

bool on_circle(const real_point& a, const real_point& b, const real_point& c, const real_point& d) returns true if points a, b, c, and d are cocircular.

bool cocircular(const real_point& a, const real_point& b, const real_point& c, const real_point& d) returns true if points a, b, c, and d are cocircular.
int compare_by_angle(const real_point& a, const real_point& b, 
const real_point& c, const real_point& d)
    compares vectors \(b - a\) and \(d - c\) by angle (more efficient 
than calling compare_by_angle\((b - a, d - x)\) on vectors).

bool affinely_independent(const array<real_point>& A)
decides whether the points in \(A\) are affinely independent.

bool contained_in_simplex(const array<real_point>& A, const real_point& p)
determines whether \(p\) is contained in the simplex 
spanned by the points in \(A\). \(A\) may consist of up to 
3 points.
    \textit{Precondition:} The points in \(A\) are affinely independent.

bool contained_in_affine_hull(const array<real_point>& A, const real_point& p)
determines whether \(p\) is contained in the affine hull of 
the points in \(A\).
15.18 Real Segments ( real_segment )

1. Definition

An instance $s$ of the data type `real_segment` is a directed straight line segment in the two-dimensional plane, i.e., a straight line segment $[p, q]$ connecting two points $p, q \in \mathbb{R}^2$. $p$ is called the source or start point and $q$ is called the target or end point of $s$. The length of $s$ is the Euclidean distance between $p$ and $q$. If $p = q$, $s$ is called empty. We use `line(s)` to denote a straight line containing $s$.

```c
#include <LEDA/geo/real_segment.h>
```

2. Types

- `real_segment::coord_type` the coordinate type (`real`).
- `real_segment::point_type` the point type (`real_point`).

3. Creation

```c
real_segment s(const real_point& p, const real_point& q);
```
introduces a variable $s$ of type `real_segment`. $s$ is initialized to the segment $[p, q]$.

```c
real_segment s(const real_point& p, const real_vector& v);
```
introduces a variable $s$ of type `real_segment`. $s$ is initialized to the segment $[p, p + v]$.

**Precondition:** $v$.dim() = 2.

```c
real_segment s(real x1, real y1, real x2, real y2);
```
introduces a variable $s$ of type `real_segment`. $s$ is initialized to the segment $[(x_1, y_1), (x_2, y_2)]$.

```c
real_segment s;
```
introduces a variable $s$ of type `real_segment`. $s$ is initialized to the empty segment.

```c
real_segment s(const segment& s1, int prec = 0);
```
introduces a variable $s$ of type `real_segment` initialized to the segment $s_1$. (The second argument is for compatibility with `rat_segment`.)

```c
real_segment s(const rat_segment& s1);
```
introduces a variable $s$ of type `real_segment` initialized to the segment $s_1$. 

```c
```
4. Operations

real_point s.start() returns the source point of segment s.
real_point s.end() returns the target point of segment s.
real s.xcoord1() returns the x-coordinate of s.source().
real s.xcoord2() returns the x-coordinate of s.target().
real s.ycoord1() returns the y-coordinate of s.source().
real s.ycoord2() returns the y-coordinate of s.target().
real s.dx() returns the $x_{coord2} - x_{coord1}$.
real s.dy() returns the $y_{coord2} - y_{coord1}$.
real s.slope() returns the slope of s.
  \textit{Precondition:} s is not vertical.
real s.sqr_length() returns the square of the length of s.
real s.length() returns the length of s.
real_vector s.to_vector() returns the vector $s.target() - s.source()$.
bool s.is_trivial() returns true if s is trivial.
bool s.is_vertical() returns true iff s is vertical.
bool s.is_horizontal() returns true iff s is horizontal.
int s.orientation(const real_point& p) computes orientation(s.source(), s.target(), p) (see below).
real s.x_proj(real y) returns $p.x_{coord}()$, where $p \in line(s)$ with $p.y_{coord}() = y$.
  \textit{Precondition:} s is not horizontal.
real s.y_proj(real x) returns $p.y_{coord}()$, where $p \in line(s)$ with $p.x_{coord}() = x$.
  \textit{Precondition:} s is not vertical.
real s.y_abs() returns the y-abscissa of line(s), i.e., s.y_proj(0).
  \textit{Precondition:} s is not vertical.
bool s.contains(const real_point& p) decides whether s contains p.
bool s.intersection(const real_segment& t) decides whether s and t intersect in one point.
bool s.intersection(const real_segment& t, real_point& p)
if s and t intersect in a single point this point is assigned
to p and the result is true, otherwise the result is false.

bool s.intersection_of_lines(const real_segment& t, real_point& p)
if line(s) and line(t) intersect in a single point this point
is assigned to p and the result is true, otherwise the result
is false.

real_segment s.translate(real dx, real dy)
returns s translated by vector (dx, dy).

real_segment s.translate(const real_vector& v)
returns s + v, i.e., s translated by vector v.
Precondition: v.dim() = 2.

real_segment s + const real_vector& v
returns s translated by vector v.

real_segment s − const real_vector& v
returns s translated by vector −v.

real_segment s.perpendicular(const real_point& p)
returns the segment perpendicular to s with source p and
target on line(s).

real s.distance(const real_point& p)
returns the Euclidean distance between p and s.

real s.sqr_dist(const real_point& p)
returns the squared Euclidean distance between p and s.

real s.distance() returns the Euclidean distance between (0,0) and s.

real_segment s.rotate90(const real_point& q, int i = 1)
returns s rotated about q by an angle of i × 90 degrees.
If i > 0 the rotation is counter-clockwise otherwise it is
clockwise.

real_segment s.rotate90(int i = 1)
returns s.rotate90(s.source(),i).

real_segment s.reflect(const real_point& p, const real_point& q)
returns s reflected across the straight line passing through
p and q.

real_segment s.reflect(const real_point& p)
returns s reflected across point p.
real_segment s.reverse() returns s reversed.

Non-Member Functions

int orientation(const real_segment& s, const real_point& p)
    computes orientation(s.source(), s.target(), p).

int cmp_slopes(const real_segment& s1, const real_segment& s2)
    returns compare(slope(s1), slope(s2)).

int cmp_segments_at_xcoord(const real_segment& s1, const real_segment& s2, const real_point& p)
    compares points $l_1 \cap v$ and $l_2 \cap v$ where $l_i$ is the line underlying segment $s_i$ and $v$ is the vertical straight line passing through point $p$.

bool parallel(const real_segment& s1, const real_segment& s2)
    returns true if $s1$ and $s2$ are parallel and false otherwise.
15.19 Real Rays (real_ray)

1. Definition
An instance $r$ of the data type real_ray is a directed straight ray in the two-dimensional plane.

```cpp
#include <LEDA/geo/real_ray.h>
```

2. Types

- `real_ray::coord_type`: the coordinate type (real).
- `real_ray::point_type`: the point type (real_point).

3. Creation

```cpp
real_ray r(const real_point& p, const real_point& q);
```

introduces a variable $r$ of type real_ray. $r$ is initialized to the ray starting at point $p$ and passing through point $q$.

```cpp
real_ray r(const real_segment& s);
```

introduces a variable $r$ of type real_ray. $r$ is initialized to `real_ray(s.source(), s.target())`.

```cpp
real_ray r(const real_point& p, const real_vector& v);
```

introduces a variable $r$ of type real_ray. $r$ is initialized to `real_ray(p, p + v)`.

```cpp
real_ray r;
```

introduces a variable $r$ of type real_ray. $r$ is initialized to the ray starting at the origin with direction 0.

```cpp
real_ray r(const ray& r1, int prec = 0);
```

introduces a variable $r$ of type real_ray initialized to the ray $r_1$. (The second argument is for compatibility with rat_ray.)

```cpp
real_ray r(const rat_ray& r1);
```

introduces a variable $r$ of type real_ray initialized to the ray $r_1$.

4. Operations

```cpp
real_point r.source();
```

returns the source of $r$.

```cpp
real_point r.point1();
```

returns the source of $r$. 
real_point r.point2() returns a point on r different from r.source().

bool r.is_vertical() returns true iff r is vertical.

bool r.is_horizontal() returns true iff r is horizontal.

real r.slope() returns the slope of the straight line underlying r.
Precondition: r is not vertical.

bool r.intersection(const real_ray& s, real_point& inter) if r and s intersect in a single point this point is assigned to inter and the result is true, otherwise the result is false.

bool r.intersection(const real_segment& s, real_point& inter) if r and s intersect in a single point this point is assigned to inter and the result is true, otherwise the result is false.

real_ray r.translate(real dx, real dy) returns r translated by vector (dx, dy).

real_ray r.translate(const real_vector& v) returns r translated by vector v
Precondition: v.dim() = 2.

real_ray r + const real_vector& v returns r translated by vector v.

real_ray r - const real_vector& v returns r translated by vector -v.

real_ray r.rotate90(const real_point& q, int i = 1) returns r rotated about q by an angle of i × 90 degrees. If i > 0 the rotation is counter-clockwise otherwise it is clockwise.

real_ray r.reflect(const real_point& p, const real_point& q) returns r reflected across the straight line passing through p and q.

real_ray r.reflect(const real_point& p) returns r reflected across point p.

real_ray r.reverse() returns r reversed.

bool r.contains(const real_point& ) decides whether r contains p.

bool r.contains(const real_segment& ) decides whether r contains s.
Non-Member Functions

\[ \texttt{int } \texttt{orientation(const real\_ray & } r, \texttt{ const real\_point & } p) \]

computes \( \text{orientation}(a, b, p) \) (see the manual page of \texttt{real\_point}), where \( a \neq b \) and \( a \) and \( b \) appear in this order on ray \( r \).

\[ \texttt{int } \texttt{cmp\_slopes(const real\_ray & } r1, \texttt{ const real\_ray & } r2) \]

returns \( \text{compare}(\text{slope}(r_1), \text{slope}(r_2)) \) where \( \text{slope}(r_i) \) denotes the slope of the straight line underlying \( r_i \).
15.20 Straight Real Lines (real_line)

1. Definition

An instance \(l\) of the data type real_line is a directed straight line in the two-dimensional plane.

#include <LEDA/geo/real_line.h>

2. Types

\texttt{real_line::coord\_type} \hspace{1cm} \text{the coordinate type (real)}.

\texttt{real_line::point\_type} \hspace{1cm} \text{the point type (real\_point)}.

3. Creation

\texttt{real\_line\ l(const real\_point& p, const real\_point& q);} \hspace{1cm} \text{introduces a variable \(l\) of type real\_line. \(l\) is initialized to the line passing through points \(p\) and \(q\) directed from \(p\) to \(q\).}

\texttt{real\_line\ l(const real\_segment& s);} \hspace{1cm} \text{introduces a variable \(l\) of type real\_line. \(l\) is initialized to the line supporting segment \(s\).}

\texttt{real\_line\ l(const real\_ray& r);} \hspace{1cm} \text{introduces a variable \(l\) of type real\_line. \(l\) is initialized to the line supporting ray \(r\).}

\texttt{real\_line\ l(const real\_point& p, const real\_vector& v);} \hspace{1cm} \text{introduces a variable \(l\) of type real\_line. \(l\) is initialized to the line passing through points \(p\) and \(p + v\).}

\texttt{real\_line\ l; } \hspace{1cm} \text{introduces a variable \(l\) of type real\_line. \(l\) is initialized to the line passing through the origin with direction 0.}

\texttt{real\_line\ l(const line& l1, int prec = 0);} \hspace{1cm} \text{introduces a variable \(l\) of type real\_line initialized to the line \(l_1\). (The second argument is for compatibility with rat\_line.)}

\texttt{real\_line\ l(const rat\_line& l1);} \hspace{1cm} \text{introduces a variable \(l\) of type real\_line initialized to the line \(l_1\).}
4. Operations

\texttt{real\_point \ l.point1()} \quad \text{returns a point on \( l \).}

\texttt{real\_point \ l.point2()} \quad \text{returns a second point on \( l \).}

\texttt{real\_segment \ l.seg()} \quad \text{returns a segment on \( l \).}

\texttt{bool \ l.is\_vertical()} \quad \text{returns true iff \( l \) is vertical.}

\texttt{bool \ l.is\_horizontal()} \quad \text{returns true iff \( l \) is horizontal.}

\texttt{real \ l.sqr\_dist(const \ real\_point\& \ q)}
\hspace{1cm} \text{returns the square of the distance between \( l \) and \( q \).}

\texttt{real \ l.distance(const \ real\_point\& \ q)}
\hspace{1cm} \text{returns the distance between \( l \) and \( q \).}

\texttt{int \ l.orientation(const \ real\_point\& \ p)}
\hspace{1cm} \text{returns orientation}(l.point1(), l.point2(), p).

\texttt{real \ l.slope()} \quad \text{returns the slope of \( l \).}
\hspace{1cm} \textbf{Precondition:} \( l \) is not vertical.

\texttt{real \ l.y\_proj(real \ x)} \quad \text{returns p.ycoord(), where \( p \in l \) with p.xcoord()} = \quad \text{returns p.xcoord(), where \( p \in l \) with p.ycoord()} = \quad \text{returns the y-abscissa of \( l \) (l.y\_proj(0)).}
\hspace{1cm} \text{Precondition:} \( l \) is not vertical.
\hspace{1cm} \text{Precondition:} \( l \) is not horizontal.

\texttt{real \ l.x\_proj(real \ y)} \quad \text{returns p.xcoord(), where \( p \in l \) with p.ycoord()} = \quad \text{returns true, if \( l \) and \( s \) intersect, false otherwise.}
\hspace{1cm} \text{Precondition:} \( l \) is not vertical.

\texttt{real\_line \ l.translate(real \ dx, real \ dy)}
\hspace{1cm} \text{returns \( l \) translated by vector \((dx, dy)\).}
real_line l.translate(const real_vector& v) returns l translated by vector v. 
Precondition: v.dim() = 2.

real_line l + const real_vector& v returns l translated by vector v.

real_line l - const real_vector& v returns l translated by vector -v.

real_line l.rotate90(const real_point& q, int i = 1) returns l rotated about q by an angle of i × 90 degrees. If i > 0 the rotation is counter-clockwise otherwise it is clockwise.

real_line l.reflect(const real_point& p, const real_point& q) returns l reflected across the straight line passing through p and q.

real_line l.reverse() returns l reversed.

real_segment l.perpendicular(const real_point& p) returns the segment perpendicular to l with source p. and target on l.

real_point l.dual() returns the point dual to l. 
Precondition: l is not vertical.

int l.side_of(const real_point& p) computes orientation(a, b, p), where a ≠ b and a and b appear in this order on line l.

bool l.contains(const real_point& p) returns true if p lies on l.

bool l.clip(real_point p, real_point q, real_segment& s) clips l at the rectangle R defined by p and q. Returns true if the intersection of R and l is non-empty and returns false otherwise. If the intersection is non-empty the intersection is assigned to s; it is guaranteed that the source node of s is no larger than its target node.

Non-Member Functions

int orientation(const real_line& l, const real_point& p) computes orientation(a, b, p) (see the manual page of real_point), where a ≠ b and a and b appear in this order on line l.
int cmp_slopes(const real_line& l1, const real_line& l2)
    returns compare(slope(l1), slope(l2)).
CHAPTER 15. BASIC DATA TYPES FOR TWO-DIMENSIONAL GEOMETRY

15.21 Real Circles ( real_circle )

1. Definition
An instance $C$ of the data type $\text{real}_\text{circle}$ is an oriented circle in the plane passing through three points $p_1, p_2, p_3$. The orientation of $C$ is equal to the orientation of the three defining points, i.e. $\text{orientation}(p_1, p_2, p_3)$. If $|\{p_1, p_2, p_3\}| = 1$ $C$ is the empty circle with center $p_1$. If $p_1, p_2, p_3$ are collinear $C$ is a straight line passing through $p_1, p_2$ and $p_3$ in this order and the center of $C$ is undefined.

#include <LEDA/geo/real_circle.h>

2. Types
$\text{real}_\text{circle}::\text{coord\_type}$ the coordinate type ($\text{real}$).
$\text{real}_\text{circle}::\text{point\_type}$ the point type ($\text{real\_point}$).

3. Creation
$\text{real}_\text{circle} C(\text{const real\_point}\& a, \text{const real\_point}\& b, \text{const real\_point}\& c);$ introduces a variable $C$ of type $\text{real}_\text{circle}$. $C$ is initialized to the oriented circle through points $a, b,$ and $c$.

$\text{real}_\text{circle} C(\text{const real\_point}\& a, \text{const real\_point}\& b);$ introduces a variable $C$ of type $\text{real}_\text{circle}$. $C$ is initialized to the counter-clockwise oriented circle with center $a$ passing through $b$.

$\text{real}_\text{circle} C(\text{const real\_point}\& a);$ introduces a variable $C$ of type $\text{real}_\text{circle}$. $C$ is initialized to the trivial circle with center $a$.

$\text{real}_\text{circle} C;$ introduces a variable $C$ of type $\text{real}_\text{circle}$. $C$ is initialized to the trivial circle with center $(0,0)$.

$\text{real}_\text{circle} C(\text{const real\_point}\& c, \text{real} r);$ introduces a variable $C$ of type $\text{real}_\text{circle}$. $C$ is initialized to the circle with center $c$ and radius $r$ with positive (i.e. counter-clockwise) orientation.

$\text{real}_\text{circle} C(\text{real} x, \text{real} y, \text{real} r);$ introduces a variable $C$ of type $\text{real}_\text{circle}$. $C$ is initialized to the circle with center $(x, y)$ and radius $r$ with positive (i.e. counter-clockwise) orientation.
real\_circle\ C(\ const\ circle&\ c,\ int\ prec\ =\ 0);\n\nintroduces a variable \emph{C} of type \textit{real\_circle} initialized to the \textit{circle} \emph{c}. (The second argument is for compatibility with \textit{rat\_circle}.)

\textit{real\_circle\ C(\ const\ rat\_circle&\ c);}\n\nintroduces a variable \emph{C} of type \textit{real\_circle} initialized to the \textit{circle} \emph{c}.  

4. Operations

\textit{real\_point\ C.\center()} \hspace{1cm} \text{returns the center of} \emph{C}.

\textit{Precondition:} The orientation of \emph{C} is not 0.

\textit{real\ C.\radius()} \hspace{1cm} \text{returns the radius of} \emph{C}.

\textit{Precondition:} The orientation of \emph{C} is not 0.

\textit{real\ C.\sqr\_radius()} \hspace{1cm} \text{returns the squared radius of} \emph{C}.

\textit{Precondition:} The orientation of \emph{C} is not 0.

\textit{real\_point\ C.\point1()} \hspace{1cm} \text{returns} \emph{p}_1.

\textit{real\_point\ C.\point2()} \hspace{1cm} \text{returns} \emph{p}_2.

\textit{real\_point\ C.\point3()} \hspace{1cm} \text{returns} \emph{p}_3.

\textit{bool\ C.\is\_degenerate()} \hspace{1cm} \text{returns true if the defining points are collinear.}

\textit{bool\ C.\is\_trivial()} \hspace{1cm} \text{returns true if} \emph{C} \text{has radius zero.}

\textit{bool\ C.\is\_line()} \hspace{1cm} \text{returns true if} \emph{C} \text{is a line.}

\textit{real\_line\ C.\to\_line()} \hspace{1cm} \text{returns} \textit{line}(\textit{point1}(\ ),\ \textit{point3}(\ )).

\textit{int\ C.\orientation()} \hspace{1cm} \text{returns the orientation of} \emph{C}.

\textit{int\ C.\side\_of(\ const\ real\_point&\ p)} \hspace{1cm} \text{returns} −1, +1, or 0 if \emph{p} \text{lies right of, left of, or on} \emph{C} \text{respectively.}

\textit{bool\ C.\inside(\ const\ real\_point&\ p)} \hspace{1cm} \text{returns true iff} \emph{p} \text{lies inside of} \emph{C}.

\textit{bool\ C.\outside(\ const\ real\_point&\ p)} \hspace{1cm} \text{returns true iff} \emph{p} \text{lies outside of} \emph{C}.

\textit{bool\ C.\contains(\ const\ real\_point&\ p)} \hspace{1cm} \text{returns true iff} \emph{p} \text{lies on} \emph{C}.
real\_circle \ C.\text{translate}(real \ dx, real \ dy) \quad \text{return}s \ C \ \text{translated by vector} \ (dx, dy).

real\_circle \ C.\text{translate}(\text{const real\_vector} \& \ v) \quad \text{return}s \ C \ \text{translated by vector} \ v.

real\_circle \ C + \text{const real\_vector} \& \ v \quad \text{return}s \ C \ \text{translated by vector} \ v.

real\_circle \ C - \text{const real\_vector} \& \ v \quad \text{return}s \ C \ \text{translated by vector} \ -v.

real\_circle \ C.\text{rotate90}(\text{const real\_point} \& \ q, \text{int} \ i = 1) \quad \text{return}s \ C \ \text{rotated about} \ q \ \text{by an angle of} \ i \times 90 \ \text{degrees}. \ \text{If} \ i > 0 \ \text{the rotation is counter-clockwise otherwise it is clockwise.}

real\_circle \ C.\text{reflect}(\text{const real\_point} \& \ p, \text{const real\_point} \& \ q) \quad \text{return}s \ C \ \text{reflected across the straight line passing through} \ p \ \text{and} \ q.

real\_circle \ C.\text{reflect}(\text{const real\_point} \& \ p) \quad \text{return}s \ C \ \text{reflected across point} \ p.

real\_circle \ C.\text{reverse}() \quad \text{return}s \ C \ \text{reversed.}

list<real\_point> \ C.\text{intersection}(\text{const real\_circle} \& \ D) \quad \text{return}s \ C \cap D \ \text{as a list of points.}

list<real\_point> \ C.\text{intersection}(\text{const real\_line} \& \ l) \quad \text{return}s \ C \cap l \ \text{as a list of (zero, one, or two) points sorted along} \ l.

list<real\_point> \ C.\text{intersection}(\text{const real\_segment} \& \ s) \quad \text{return}s \ C \cap s \ \text{as a list of (zero, one, or two) points sorted along} \ s.

real\_segment \ C.\text{left\_tangent}(\text{const real\_point} \& \ p) \quad \text{return}s \ the \ line \ segment \ starting \ in \ p \ \text{tangent to} \ C \ \text{and left of segment} \ [p, C.\text{center}()] .

real\_segment \ C.\text{right\_tangent}(\text{const real\_point} \& \ p) \quad \text{return}s \ the \ line \ segment \ starting \ in \ p \ \text{tangent to} \ C \ \text{and right of segment} \ [p, C.\text{center}()] .

real \ C.\text{distance}(\text{const real\_point} \& \ p) \quad \text{return}s \ the \ distance \ between \ C \ \text{and} \ p.

real \ C.\text{sqr\_dist}(\text{const real\_point} \& \ p) \quad \text{return}s \ the \ squared \ distance \ between \ C \ \text{and} \ p.
real \( C.\text{distance}(\text{const real}_\text{line}& \ l) \)
returns the distance between \( C \) and \( l \).

real \( C.\text{distance}(\text{const real}_\text{circle}& \ D) \)
returns the distance between \( C \) and \( D \).

bool \( \text{radical}_\text{axis}(\text{const real}_\text{circle}& \ C1, \text{const real}_\text{circle}& \ C2, \ 
\text{real}_\text{line}& \ rad\_axis) \)
if the radical axis for \( C1 \) and \( C2 \) exists, it is assigned to \( rad\_axis \) and true is returned; otherwise the result is false.
15.22 Real Triangles ( real_triangle )

1. Definition

An instance $t$ of the data type `real_triangle` is an oriented triangle in the two-dimensional plane. A triangle splits the plane into one bounded and one unbounded region. If the triangle is positively oriented, the bounded region is to the left of it, if it is negatively oriented, the unbounded region is to the left of it. A triangle $t$ is called degenerate, if the 3 vertices of $t$ are collinear.

```cpp
#include <LEDA/geo/real_triangle.h>
```

2. Types

- `real_triangle::coord_type` the coordinate type (real).
- `real_triangle::point_type` the point type (real_point).

3. Creation

- `real_triangle t;` introduces a variable $t$ of type `real_triangle`. $t$ is initialized to the empty triangle.
- `real_triangle t(const real_point& p, const real_point& q, const real_point& r);` introduces a variable $t$ of type `real_triangle`. $t$ is initialized to the triangle $[p,q,r]$.
- `real_triangle t(real x1, real y1, real x2, real y2, real x3, real y3);` introduces a variable $t$ of type `real_triangle`. $t$ is initialized to the triangle $[(x1,y1),(x2,y2),(x3,y3)]$.
- `real_triangle t(const triangle& t1, int prec = 0);` introduces a variable $t$ of type `real_triangle` initialized to the triangle $t_1$. (The second argument is for compatibility with `rat_triangle`.)
- `real_triangle t(const rat_triangle& t1);` introduces a variable $t$ of type `real_triangle` initialized to the triangle $t_1$.

4. Operations

- `real_point t.point1()` returns the first vertex of triangle $t$.
- `real_point t.point2()` returns the second vertex of triangle $t$. 
15.22. REAL TRIANGLES (REAL_TRIANGLE)

real_point t.point3() returns the third vertex of triangle t.

real_point t[int i] returns the i-th vertex of t. Precondition: 1 ≤ i ≤ 3.

int t.orientation() returns the orientation of t.

real t.area() returns the signed area of t (positive, if orientation(a, b, c) > 0, negative otherwise).

bool t.is_degenerate() returns true if the vertices of t are collinear.

int t.side_of(const real_point& p)
returns +1 if p lies to the left of t, 0 if p lies on t and −1 if p lies to the right of t.

region_kind t.region_of(const real_point& p)
returns BOUNDED_REGION if p lies in the bounded region of t, ON_REGION if p lies on t and UNBOUNDED_REGION if p lies in the unbounded region.

bool t.inside(const real_point& p)
returns true, if p lies to the left of t.

bool t.outside(const real_point& p)
returns true, if p lies to the right of t.

bool t.on_boundary(const real_point& p)
decides whether p lies on the boundary of t.

bool t.contains(const real_point& p)
decides whether t contains p.

bool t.intersection(const real_line& l)
decides whether the bounded region or the boundary of t and l intersect.

bool t.intersection(const real_segment& s)
decides whether the bounded region or the boundary of t and s intersect.

real_triangle t.translate(real dx, real dy)
returns t translated by vector (dx, dy).

real_triangle t.translate(const real_vector& v)
returns t + v, i.e., t translated by vector v. Precondition: v.dim() = 2.
real_triangle t + const real_vector& v
returns t translated by vector v.

real_triangle t − const real_vector& v
returns t translated by vector −v.

real_triangle t.rotate90(const real_point& q, int i = 1)
returns t rotated about q by an angle of \( i \times 90 \) degrees.
If \( i > 0 \) the rotation is counter-clockwise otherwise it is clockwise.

real_triangle t.rotate90(int i = 1)
returns t.rotate90(t.source(),i).

real_triangle t.reflect(const real_point& p, const real_point& q)
returns t reflected across the straight line passing through p and q.

real_triangle t.reflect(const real_point& p)
returns t reflected across point p.

real_triangle t.reverse()
returns t reversed.
15.23 Iso-oriented Real Rectangles (real_rectangle)

1. Definition
An instance \( r \) of the data type real_rectangle is an iso-oriented rectangle in the two-dimensional plane.

\[
\text{#include } < \text{LEDA/geo/real_rectangle.h }>
\]

2. Creation

\[
\text{real_rectangle } r(\text{const real_point}& p, \text{const real_point}& q);
\]
introduces a variable \( r \) of type real_rectangle. \( r \) is initialized to the real_rectangle with diagonal corners \( p \) and \( q \).

\[
\text{real_rectangle } r(\text{const real_point}& p, \text{real w, real h});
\]
introduces a variable \( r \) of type real_rectangle. \( r \) is initialized to the real_rectangle with lower left corner \( p \), width \( w \) and height \( h \).

\[
\text{real_rectangle } r(\text{real x1, real y1, real x2, real y2});
\]
introduces a variable \( r \) of type real_rectangle. \( r \) is initialized to the real_rectangle with diagonal corners \((x1, y1)\) and \((x2, y2)\).

\[
\text{real_rectangle } r(\text{const rectangle}& r1, \text{int prec} = 0);
\]
introduces a variable \( r \) of type real_rectangle initialized to the rectangle \( r1 \). (The second argument is for compatibility with rat_rectangle.)

\[
\text{real_rectangle } r(\text{const rat_rectangle}& r1);
\]
introduces a variable \( r \) of type real_rectangle initialized to the rectangle \( r1 \).

3. Operations

\[
\text{real_point } r.\text{upper_left}( )
\]
returns the upper left corner.

\[
\text{real_point } r.\text{upper_right}( )
\]
returns the upper right corner.

\[
\text{real_point } r.\text{lower_left}( )
\]
returns the lower left corner.

\[
\text{real_point } r.\text{lower_right}( )
\]
returns the lower right corner.

\[
\text{real_point } r.\text{center}( )
\]
returns the center of \( r \).

\[
\text{list<real_point>} r.\text{vertices}( )
\]
returns the vertices of \( r \) in counterclockwise order starting from the lower left point.
real $r$.xmin() returns the minimal x-coordinate of $r$.
real $r$.xmax() returns the maximal x-coordinate of $r$.
real $r$.ymin() returns the minimal y-coordinate of $r$.
real $r$.ymax() returns the maximal y-coordinate of $r$.
real $r$.width() returns the width of $r$.
real $r$.height() returns the height of $r$.
bool $r$.is_degenerate() returns true, if $r$ degenerates to a segment or point (the 4 corners are collinear), false otherwise.
bool $r$.is_point() returns true, if $r$ degenerates to a point.
bool $r$.is_segment() returns true, if $r$ degenerates to a segment.
int $r$.cs_code(const real_point& $p$) returns the code for Cohen-Sutherland algorithm.
bool $r$.inside(const real_point& $p$) returns true, if $p$ is inside of $r$, false otherwise.
bool $r$.outside(const real_point& $p$) returns true, if $p$ is outside of $r$, false otherwise.
bool $r$.inside_or_contains(const real_point& $p$) returns true, if $p$ is inside of $r$ or on the border, false otherwise.
bool $r$.contains(const real_point& $p$) returns true, if $p$ is on the border of $r$, false otherwise.
region_kind $r$.region_of(const real_point& $p$) returns BOUNDED_REGION if $p$ lies in the bounded region of $r$, returns ON_REGION if $p$ lies on $r$, and returns UNBOUNDED_REGION if $p$ lies in the unbounded region.
real_rectangle $r$.include(const real_point& $p$) returns a new rectangle that includes the points of $r$ and $p$.
real_rectangle $r$.include(const real_rectangle& $r2$) returns a new rectangle that includes the points of $r$ and $r2$. 
real_rectangle r.translate(real dx, real dy) returns a new rectangle that is the translation of r by \((dx, dy)\).

real_rectangle r.translate(const real_vector &v) returns a new rectangle that is the translation of r by v.

real_rectangle r + const real_vector &v returns r translated by v.

real_rectangle r - const real_vector &v returns r translated by \(-v\).

real_point r[int i] returns the \(i\)-th vertex of r. Precondition: \((0 < i < 5)\).

real_rectangle r.rotate90(const real_point &p, int i = 1) returns r rotated about p by an angle of \(i \times 90\) degrees. If \(i > 0\) the rotation is counter-clockwise otherwise it is clockwise.

real_rectangle r.rotate90(int i = 1) returns r rotated by an angle of \(i \times 90\) degrees about the origin.

real_rectangle r.reflect(const real_point &p) returns r reflected across p.

list<real_point> r.intersection(const real_segment &s) returns \(r \cap s\).

bool r.clip(const real_segment &t, real_segment &inter) clips t on r and returns the result in inter.

bool r.clip(const real_line &l, real_segment &inter) clips l on r and returns the result in inter.

bool r.clip(const real_ray &ry, real_segment &inter) clips ry on r and returns the result in inter.

bool r.difference(const real_rectangle &q, list<real_rectangle> &L) returns true iff the difference of r and q is not empty, and false otherwise. The difference L is returned as a partition into rectangles.

list<real_point> r.intersection(const real_line &l) returns \(r \cap l\).
\texttt{list<real\_rectangle> r.intersection(const real\_rectangle& s)}
\texttt{\hspace{1cm} returns } r \cap s.

\texttt{bool r.do\_intersect(const real\_rectangle& b)}
\texttt{\hspace{1cm} returns } true \textit{iff } r \text{ and } b \text{ intersect, false otherwise.}

\texttt{real r.area()} \texttt{\hspace{1cm} returns the area of } r.
15.24 Geometry Algorithms ( geo_alg )

All functions listed in this section work for geometric objects based on both floating-point and exact (rational) arithmetic. In particular, point can be replaced by rat_point, segment by rat_segment, and circle by rat_circle.

The floating point versions are faster but unreliable. They may produce incorrect results, abort, or run forever. Only the rational versions will produce correct results for all inputs.

The include-file for the rational version is rat_geo_alg.h, the include-file for the floating point version is float_geo_alg.h, and geo_alg.h includes both versions. Including both versions increases compile time. An alternative name for geo_alg.h is plane_alg.h.

- Convex Hulls

```
list<point> CONVEX_HULL(const list<point>& L)
CONVEX_HULL takes as argument a list of points and returns the polygon representing the convex hull of L. The cyclic order of the vertices in the result list corresponds to counter-clockwise order of the vertices on the hull. The algorithm calls our current favorite of the algorithms below.
```

```
polygon CONVEX_HULL_POLY(const list<point>& L)
CONVEX_HULL_POLY as above, but returns the convex hull of L as a polygon.
```

```
list<point> UPPER_CONVEX_HULL(const list<point>& L)
returns the upper convex hull of L.
```

```
list<point> LOWER_CONVEX_HULL(const list<point>& L)
returns the lower convex hull of L.
```

```
list<point> CONVEX_HULL_S(const list<point>& L)
CONVEX_HULL_S as above, but the algorithm is based on the sweep paradigm. Running time is \( O(n \log n) \) in the worst and in the best case.
```

```
list<point> CONVEX_HULL_IC(const list<point>& L)
CONVEX_HULL_IC as above, but the algorithm is based on incremental construction. The running time is \( O(n^2) \) worst case and is \( O(n \log n) \) expected case. The expectation is computed as the average over all permutations of L. The running time is linear in the best case.
```

```
list<point> CONVEX_HULL_RIC(const list<point>& L)
CONVEX_HULL_RIC as above. The algorithm permutes L randomly and then calls the preceding function.
```
double WIDTH(const list<point>& L, line& l1, line& l2)
returns the square of the minimum width of a stripe covering all points
in L and the two boundaries of the stripe.
Precondition: L is non-empty

• Halfplane intersections

void HALFPLANE_INTERSECTION(const list<line>& L, list<line>& Lout)
For every line \( \ell \in L \) let \( h_\ell \) be the closed halfplane lying on the postive side of \( \ell \), i.e., \( h_\ell = \{ p \in \mathbb{R}^2 \mid \text{orientation}(\ell, p) \geq 0 \} \), and let \( H = \bigcap_{\ell \in L} h_\ell \). Then HALFPLANE_INTERSECTION computes the list
of lines Lout defining the boundary of H in counter-clockwise ordering.

• Triangulations

edge TRIANGULATE_POINTS(const list<point>& L, GRAPH<point, int>& T)
computes a triangulation (planar map) T of the points in L and returns
an edge of the outer face (convex hull).

void DELAUNAY_TRIANG(const list<point>& L, GRAPH<point, int>& DT)
computes the delaunay triangulation DT of the points in L.

void DELAUNAY_DIAGRAM(const list<point>& L, GRAPH<point, int>& DD)
computes the delaunay diagram DD of the points in L.

void F_DELAUNAY_TRIANG(const list<point>& L, GRAPH<point, int>& FDT)
computes the furthest point delaunay triangulation FDT of the points in L.

void F_DELAUNAY_DIAGRAM(const list<point>& L, GRAPH<point, int>& FDD)
computes the furthest point delaunay diagram FDD of the points in L.

• Constraint Triangulations

edge TRIANGULATE_SEGMENTS(const list<segment>& L, 
GRAPH<point, segment>& G)
computes a constrained triangulation (planar map) T of the segments in
L (trivial segments representing points are allowed). The function returns
an edge of the outer face (convex hull).

edge DELAUNAY_TRIANG(const list<segment>& L, GRAPH<point, segment>& G)
computes a constrained Delaunay triangulation T of the segments in L.
The function returns an edge of the outer face (convex hull).
edge TRIANGULATE_PLANE_MAP(GRAPH<point, segment>& G)
computes a constrained triangulation $T$ of the plane map (counter-
clockwise straight-line embedded Graph) $G$. The function returns an
edge of the outer face (convex hull). *Precondition:* $G$ is simple.

edge DELAUNAY_TRIANG(GRAPH<point, segment>& G)
computes a constrained Delaunay triangulation $T$ of the plane map $G$.
The function returns an edge of the outer face (convex hull). *Precondition:*
$G$ is simple.

edge TRIANGULATE_POLYGON(const polygon& $P$, GRAPH<point, segment>& $G$,
list<edge>& inner_edges, list<edge>& outer_edges,
list<edge>& boundary_edges)
triangulates the interior and exterior of the simple polygon $P$ and stores
all edges of the inner (outer) triangulation in $inner_edges$ ($outer_edges$)
and the edges of the polygon boundary in $boundary_edges$. The function
returns an edge of the convex hull of $P$ if $P$ is simple and $nil$ otherwise.

edge TRIANGULATE_POLYGON(const gen_polygon& $GP$, 
GRAPH<point, segment>& $G$,
list<edge>& inner_edges, list<edge>& outer_edges,
list<edge>& boundary_edges, list<edge>& hole_edges)
triangulates the interior and exterior of the generalized polygon $GP$
and stores all edges of the inner (outer) triangulation in $inner_edges$ ($outer_edges$).
The function returns $nil$ if $GP$ is trivial, and an edge of the convex hull otherwise. $boundary_edges$ contains the edges of every
counter-clockwise oriented boundary cycle of $GP$, and $hole_edges$ contains the edges on every clockwise oriented boundary cycle of $GP$. Note that
the reversals of boundary and hole edges will be returned in $inner_edges$.
*Precondition:* $GP$ is simple.

edge CONVEX_COMPONENTS(const polygon& $P$, GRAPH<point, segment>& $G$,
list<edge>& inner_edges, list<edge>& boundary)
if $P$ is a bounded and non-trivial simple polygon its interior is decomposed
into convex parts. All inner edges of the constructed decomposition are
returned in $inner_edges$. $boundary_edges$ contains the edges of the polygon
boundary Note that the reversals of boundary edges will be stored in
$inner_edges$. The function returns an edge of the convex hull if $P$ is
simple and non-trivial and $nil$ otherwise.
edge CONVEX_COMPONENTS(const gen_polygon& GP,
    GRAPH<point, segment>& G,
    list<edge>& inner_edges, list<edge>& boundary_edges,
    list<edge>& hole_edges)

if GP is a bounded and non-trivial generalized polygon, its interior is
decomposed into convex parts. All inner edges of the constructed decom-
position are returned in inner_edges. boundary_edges contains the edges
of every counter-clockwise oriented boundary cycle of GP, and hole_edges
contains the edges of every clockwise oriented boundary cycle of GP.
Note that the reversals of boundary and hole edges will be stored in
inner_edges. The function returns an edge of the convex hull if GP is a
bounded and non-trivial and nil otherwise. Precondition: GP must be
simple.

list<polygon> TRIANGLE_COMPONENTS(const gen_polygon& GP)

triangulates the interior of generalized polygon GP and returns the result
of the triangulation as a list of polygons.

list<polygon> CONVEX_COMPONENTS(const gen_polygon& GP)

if GP is a bounded and non-trivial generalized polygon, its interior is
decomposed into convex parts. The function returns a list of polygons
that form the convex decomposition of GPs interior.

• Minkowski Sums

Please note that the Minkowski sums only work reliable for the rational kernel.

gen_polygon MINKOWSKILSUM(const polygon& P, const polygon& R)

computes the Minkowski sum of P and R.

gen_polygon MINKOWSKILDIFF(const polygon& P, const polygon& R)

computes the Minkowski difference of P and R, i.e. the Minkowski sum
of P and R.reflect(point(0, 0)).

gen_polygon MINKOWSKILSUM(const gen_polygon& P, const polygon& R)

computes the Minkowski sum of P and R.

gen_polygon MINKOWSKILDIFF(const gen_polygon& P, const polygon& R)

computes the Minkowski difference of P and R, i.e. the Minkowski sum
of P and R.reflect(point(0, 0)).

The following variants of the MINKOWSKI functions take two additional call-back func-
tion arguments conv_partition and conv_unite which are used by the algorithm to partition
the input polygons into convex parts and for computing the union of a list of convex poly-
gons, respectively (instead of using the default methods).
\texttt{gen\_polygon MINKOWSKISUM(const polygon\& P, const polygon\& R,}
\texttt{ void (*conv\_partition)(const gen\_polygon\& p,}
\texttt{ const polygon\& r, list\langle polygon\rangle\& lp,}
\texttt{ list\langle polygon\rangle\& lr),}
\texttt{ gen\_polygon (*conv\_unite)(const list\langle gen\_polygon\rangle\& ))}

\texttt{gen\_polygon MINKOWSKILDIFF(const polygon\& P, const polygon\& R,}
\texttt{ void (*conv\_partition)(const gen\_polygon\& p,}
\texttt{ const polygon\& r, list\langle polygon\rangle\& lp,}
\texttt{ list\langle polygon\rangle\& lr),}
\texttt{ gen\_polygon (*conv\_unite)(const list\langle gen\_polygon\rangle\& ))}

\texttt{gen\_polygon MINKOWSKISUM(const gen\_polygon\& P, const polygon\& R,}
\texttt{ void (*conv\_partition)(const gen\_polygon\& p,}
\texttt{ const polygon\& r, list\langle polygon\rangle\& lp,}
\texttt{ list\langle polygon\rangle\& lr),}
\texttt{ gen\_polygon (*conv\_unite)(const list\langle gen\_polygon\rangle\& ))}

\texttt{gen\_polygon MINKOWSKILDIFF(const gen\_polygon\& P, const polygon\& R,}
\texttt{ void (*conv\_partition)(const gen\_polygon\& p,}
\texttt{ const polygon\& r, list\langle polygon\rangle\& lp,}
\texttt{ list\langle polygon\rangle\& lr),}
\texttt{ gen\_polygon (*conv\_unite)(const list\langle gen\_polygon\rangle\& ))}

- \textbf{Euclidean Spanning Trees}

\texttt{void \textsc{min\_spanning\_tree}(const list<point>\& L, GRAPH<point, int>\& T)}

computes the Euclidian minimum spanning tree \( T \) of the points in \( L \).

- \textbf{Triangulation Checker}

\texttt{bool \textsc{is\_convex\_subdivision}(const GRAPH<point, int>\& G)}

returns true if \( G \) is a convex planar subdivision.

\texttt{bool \textsc{is\_triangulation}(const GRAPH<point, int>\& G)}

returns true if \( G \) is convex planar subdivision in which every bounded face is a triangle or if all nodes of \( G \) lie on a common line.

\texttt{bool \textsc{is\_delaunay\_triangulation}(const GRAPH<point, int>\& G,}
\texttt{ delaunay\_voronoikind kind \)}}

checks whether \( G \) is a nearest (\textit{kind} = \textit{NEAREST}) or furthest (\textit{kind} = \textit{FURTHEST}) site Delaunay triangulation of its vertex set. \( G \) is a Delaunay triangulation if it is a triangulation and all triangles have the Delaunay property. A triangle has the Delaunay property if no vertex of an adjacent triangle is contained in the interior (\textit{kind} = \textit{NEAREST}) or exterior (\textit{kind} = \textit{FURTHEST}) of the triangle.
bool IsDelaunay_Diagram(const GRAPH<point, int>& G, delaunay::voronoi::kind kind)
checks whether G is a nearest (kind = NEAREST) or furthest (kind = FURTHEST) site Delaunay diagram of its vertex set. G is a Delaunay diagram if it is a convex subdivision, if the vertices of any bounded face are co-circular, and if every triangulation of G is a Delaunay triangulation.

- Voronoi Diagrams

void VORONOI(const list<point>& L, GRAPH<circle, point>& VD)
VORONOI takes as input a list of points (sites) L. It computes a directed graph VD representing the planar subdivision defined by the Voronoi diagram of L. For each node v of VD G[v] is the corresponding Voronoi vertex (point) and for each edge e G[e] is the site (point) whose Voronoi region is bounded by e. The algorithm has running time $O(n^2)$ in the worst case and $O(n \log n)$ with high probability, where n is the number of sites.

void F_VORONOI(const list<point>& L, GRAPH<circle, point>& FVD)
computes the farthest point Voronoi Diagram FVD of the points in L.

circle LARGEST_EMPTY_CIRCLE(const list<point>& L)
computes a largest circle whose center lies inside the convex hull of L that contains no point of L in its interior. Returns the trivial circle if L is empty.

circle SMALLEST_ENCLOSING_CIRCLE(const list<point>& L)
computes a smallest circle containing all points of L in its interior.

void ALLEMPTY_CIRCLES(const list<point>& L, list<circle>& CL)
computes the list CL of all empty circles passing through three or more points of L.

void ALLENCLOSING_CIRCLES(const list<point>& L, list<circle>& CL)
computes the list CL of all enclosing circles passing through three or more points of L.

An annulus is either the region between two concentric circles or the region between two parallel lines.

bool MIN_AREA_ANNULUS(const list<point>& L, point& center, point& ipoint, point& opoint, line& ll)
computes the minimum area annulus containing the points of L. The function returns false if all points in L are collinear and returns true otherwise. In the former case a line passing through the points in L is returned in ll, and in the latter case the annulus is returned by its center and a point on the inner and the outer circle, respectively.
bool MIN_WIDTH_ANNULUS(const list<point>& L, point& center, point& ipoint, point& opoint, line& l1, line& l2)
computes the minimum width annulus containing the points of L. The function returns false if the minimum width annulus is a stripe and returns true otherwise. In the former case the boundaries of the stripes are returned in l1 and l2 and in the latter case the annulus is returned by its center and a point on the inner and the outer circle, respectively.

void CRUST(const list<point>& L0, GRAPH<point, int>& G)
takes a list L0 of points and traces to guess the curve(s) from which L0 are sampled. The algorithm is due to Amenta, Bern, and Eppstein. The algorithm is guaranteed to succeed if L0 is a sufficiently dense sample from a smooth closed curve.

bool Is_Voronoi_Diagram(const GRAPH<circle, point>& G, delaunay_voronoi_kind kind)
checks whether G represents a nearest (kind = NEAREST) or furthest (kind = FURTHEST) site Voronoi diagram. Voronoi diagrams of point sites are represented as planar maps as follows: There is a vertex for each vertex of the Voronoi diagram and, in addition, a vertex “at infinity” for each ray of the Voronoi diagram. Vertices at infinity have degree one. The edges of the graph correspond to the edges of the Voronoi diagram. The chapter on Voronoi diagrams of the LEDA-book [64] contains more details. Each edge is labeled with the site (class POINT) owning the region to its left and each vertex is labeled with a triple of points (= the three defining points of a CIRCLE). For a “finite” vertex the three points are any three sites associated with regions incident to the vertex (and hence the center of the circle is the position of the vertex in the plane) and for a vertex at infinity the three points are collinear and the first point and the third point of the triple are the sites whose regions are incident to the vertex at infinity. Let a and c be the first and third point of the triple respectively; a and c encode the geometric position of the vertex at infinity as follows: the vertex lies on the perpendicular bisector of a and c and to the left of the segment ac.

• Line Segment Intersection

void SEGMENT_INTERSECTION(const list<segment>& S, GRAPH<point, segment>& G, bool embed = false)
takes a list of segments S as input and computes the planar graph G induced by the set of straight line segments in S. The nodes of G are all endpoints and all proper intersection points of segments in S. The edges of G are the maximal relatively open subsegments of segments in S that contain no node of G. The edges are directed as the corresponding segments. If the flag embed is true, the corresponding planar map is computed. Note that for each edge e G[e] is the input segment that contains e (see the LEDA book for details).
void SWEEP_SEGMENTS(const list<segment>& S, GRAPH<point, segment>& G, bool embed = false, bool use_optimization = true)
as above.
The algorithm ([11]) runs in time \( O((n + s) \log n + m) \), where \( n \) is the number of segments, \( s \) is the number of vertices of the graph \( G \), and \( m \) is the number of edges of \( G \). If \( S \) contains no overlapping segments then \( m = O(n + s) \). If \( embed \) is true the running time increases by \( O(m \log m) \). If \( use_optimization \) is true an optimization described in the LEDA book is used.

void MULMULEY_SEGMENTS(const list<segment>& S, GRAPH<point, segment>& G, bool embed = false)
as above.
There is one additional output convention. If \( G \) is an undirected graph, the undirected planar map corresponding to \( G(s) \) is computed. The computation follows the incremental algorithm of Mulmuley ([68]) whose expected running time is \( O(M + s + n \log n) \), where \( n \) is the number of segments, \( s \) is the number of vertices of the graph \( G \), and \( m \) is the number of edges.

void SEGMENT_INTERSECTION(const list<segment>& S, void (*report)(const segment&, const segment&))
takes a list of segments \( S \) as input and executes for every pair \((s_1, s_2)\) of intersecting segments \( report(s_1, s_2) \). The algorithm ([6]) has running time \( O(n \log^2 n + k) \), where \( n \) is the number of segments and \( k \) is the number intersecting pairs of segments.

void SEGMENT_INTERSECTION(const list<segment>& S, list<point>& P)
takes a list of segments \( S \) as input, computes the set of (proper) intersection points between all segments in \( S \) and stores this set in \( P \). The algorithm ([11]) has running time \( O((|P| + |S|) \log |S|) \).

• Red-Blue Line Segment Intersection

void SEGMENT_INTERSECTION(const list<segment>& S1, const list<segment>& S2, GRAPH<point, segment>& G, bool embed = false)
takes two lists of segments \( S_1 \) and \( S_2 \) as input and computes the planar graph \( G \) induced by the set of straight line segments in \( S_1 \cup S_2 \) (as defined above). Precondition: Any pair of segments in \( S_1 \) or \( S_2 \), respectively, does not intersect in a point different from one of the endpoints of the segments, i.e. segments of \( S_1 \) or \( S_2 \) are either pairwise disjoint or have a common endpoint.

• Closest Pairs
double CLOSEST_PAIR(list<point>& L, point& r1, point& r2)

CLOSEST_PAIR takes as input a list of points L. It computes a pair of points \( r_1, r_2 \in L \) with minimal Euclidean distance and returns the squared distance between \( r_1 \) and \( r_2 \). The algorithm ([76]) has running time \( O(n \log n) \) where \( n \) is the number of input points.

- Miscellaneous Functions

void BoundingBox(const list<point>& L, point& pl, point& pb, point& pr, point& pt)

computes four points \( pl, pb, pr, pt \) from \( L \) such that \((x_{\text{left}}, y_{\text{bot}}, x_{\text{right}}, y_{\text{top}})\) with \( x_{\text{left}} = pl.xcoord() \), \( y_{\text{bot}} = pb.ycoord() \), \( x_{\text{right}} = pr.xcoord() \) and \( y_{\text{top}} = pt.ycoord() \) is the smallest iso-oriented rectangle containing all points of \( L \). **Precondition**: \( L \) is not empty.

bool IsSimplePolygon(const list<point>& L)

takes as input a list of points \( L \) and returns \text{true} if \( L \) is the vertex sequence of a simple polygon and false otherwise. The algorithms has running time \( O(n \log n) \), where \( n \) is the number of points in \( L \).

node NestingTree(const gen_polygon& P, GRAPH<polygon, int>& T)

The nesting tree \( T \) of a generalized polygon \( P \) is defined as follows. Every node \( v \) in \( T \) is labelled with a polygon \( T[v] \) from the boundary representation of \( P \), except for root \( r \) of \( T \) which is labelled with the empty polygon. The root symbolizes the whole two-dimensional plane. There is an edge \((u,v)\) (with \( u \neq r \)) in \( T \) iff the bounded region of \( T[v] \) is directly nested in \( T[u] \). The term "directly" means that there is no node \( w \) different from \( u \) and \( v \) such that \( T[v] \) is nested in \( T[w] \) and \( T[w] \) is nested in \( T[u] \). And there is an edge \((r,v)\) iff \( T[v] \) is not nested in any other polygon of \( P \). The function computes the nesting tree of \( P \) and returns its root. (The running time of the function depends on the order of the polygons in the boundary representation of \( P \). The closer directly nested polygons are, the better.)

- Properties of Geometric Graphs

We give procedures to check properties of geometric graph. We give procedures to verify properties of geometric graph. A geometric graph is a straight-line embedded map. Every node is mapped to a point in the plane and every dart is mapped to the line segment connecting its endpoints.

We use \textit{geo_graph} as a template parameter for geometric graph. Any instantiation of \textit{geo_graph} must provide a function

\text{VECTOR} edge\_vector(const geo\_graph& G, const edge& e)
that returns a vector from the source to the target of \( e \). In order to use any of these template functions the file `/LEDA/geo/generic/geo_check.h` must be included.

```cpp
template <class geo_graph>
bool Is_CCW_Ordered(const geo_graph& G)
    returns true if for all nodes \( v \) the neighbors of \( v \) are in increasing counter-clockwise order around \( v \).

template <class geo_graph>
bool Is_CCW_Weakly_Ordered(const geo_graph& G)
    returns true if for all nodes \( v \) the neighbors of \( v \) are in non-decreasing counter-clockwise order around \( v \).

template <class geo_graph>
bool Is_CCW_Ordered_Plane_Map(const geo_graph& G)
    Equivalent to `Is_Plane_Map(G)` and `Is_CCW_Ordered(G)`.

template <class geo_graph>
bool Is_CCW_Weakly_Ordered_Plane_Map(const geo_graph& G)
    Equivalent to `Is_Plane_Map(G)` and `Is_CCW_Weakly_Ordered(G)`.

template <class geo_graph>
void SORT_EDGES(geo_graph& G)
    Reorders the edges of \( G \) such that for every node \( v \) the edges in \( A(v) \) are in non-decreasing order by angle.

template <class geo_graph>
bool Is_CCW_Convex_Face_Cycle(const geo_graph& G, const edge& e)
    returns true if the face cycle of \( G \) containing \( e \) defines a counter-clockwise convex polygon, i.e., if the face cycle forms a cyclically increasing sequence of edges according to the compare-by-angles ordering.

template <class geo_graph>
bool Is_CCW_Weakly_Convex_Face_Cycle(const geo_graph& G, const edge& e)
    returns true if the face cycle of \( G \) containing \( e \) defines a counter-clockwise weakly convex polygon, i.e., if the face cycle forms a cyclically non-decreasing sequence of edges according to the compare-by-angles ordering.

template <class geo_graph>
bool Is_CW_Convex_Face_Cycle(const geo_graph& G, const edge& e)
    returns true if the face cycle of \( G \) containing \( e \) defines a clockwise convex polygon, i.e., if the face cycle forms a cyclically decreasing sequence of edges according to the compare-by-angles ordering.

template <class geo_graph>
```
bool Is_CW_Weakly_Convex_Face_Cycle(const geo_graph& G, const edge& e)
returns true if the face cycle of $G$ containing $e$ defines a clockwise weakly convex polygon, i.e., if the face cycle forms a cyclically non-increasing sequence of edges according to the compare-by-angles ordering.
15.25 Transformation ( TRANSFORM )

1. Definition

There are three instantiations of TRANSFORM: transform (floating point kernel), rat_transform (rational kernel) and real_transform (real kernel). The respective header file name corresponds to the type name (with “.h” appended).

An instance $T$ of type TRANSFORM is an affine transformation of two-dimensional space. It is given by a $3 \times 3$ matrix $T$ with $T_{2,0} = T_{2,1} = 0$ and $T_{2,2} \neq 0$ and maps the point $p$ with homogeneous coordinate vector $(p_x, p_y, p_w)$ to the point $T \cdot p$.

A matrix of the form
\[
\begin{pmatrix}
  w & 0 & x \\
  0 & w & y \\
  0 & 0 & w
\end{pmatrix}
\]
realizes an translation by the vector $(x/w, y/w)$ and a matrix of the form
\[
\begin{pmatrix}
  a & -b & 0 \\
  b & a & 0 \\
  0 & 0 & w
\end{pmatrix}
\]
where $a^2 + b^2 = w^2$ realizes a rotation by the angle $\alpha$ about the origin, where $\cos \alpha = a/w$ and $\sin \alpha = b/w$. Rotations are in counter-clockwise direction.

`#include <LEDA/geo/generic/TRANSFORM.h>`

2. Creation

TRANSFORM $T$; creates a variable introduces a variable $T$ of type TRANSFORM. $T$ is initialized with the identity transformation.

TRANSFORM $T$(const INT_MATRIX $t$); introduces a variable $T$ of type TRANSFORM. $T$ is initialized with the matrix $t$.
Precondition: $t$ is a $3 \times 3$ matrix with $t_{2,0} = t_{2,1} = 0$ and $t_{2,2} \neq 0$.

3. Operations

INT_MATRIX $T$.T_matrix() returns the transformation matrix

void $T$.simplify() The operation has no effect for transform. For rat_transform let $g$ be the ggT of all matrix entries. Cancels out $g$.

RAT_TYPE $T$.norm() returns the norm of the transformation
TRANSFORM $T(\text{const TRANSFORM}& \ T1)$
returns the transformation $T \circ T1$.

POINT $T(\text{const POINT}& \ p)$ returns $T(p)$.

VECTOR $T(\text{const VECTOR}& \ v)$
returns $T(v)$.

SEGMENT $T(\text{const SEGMENT}& \ s)$
returns $T(s)$.

LINE $T(\text{const LINE}& \ l)$ returns $T(l)$.

RAY $T(\text{const RAY}& \ r)$ returns $T(r)$.

CIRCLE $T(\text{const CIRCLE}& \ C)$
returns $T(C)$.

POLYGON $T(\text{const POLYGON}& \ P)$
returns $T(P)$.

GEN_POLYGON $T(\text{const GEN_POLYGON}& \ P)$
returns $T(P)$.

Non-member Functions

In any of the function below a point can be specified to the origin by replacing it by an
anonymous object of type POINT, e.g., $\text{rotation90}(\text{POINT}(\ ))$ will generate a rotation
about the origin.

TRANSFORM translation(const INT_TYPE& $dx$, const INT_TYPE& $dy$, 
const INT_TYPE& $dw$)
returns the translation by the vector $(dx/dw, dy/dw)$.

TRANSFORM translation(const RAT_TYPE& $dx$, const RAT_TYPE& $dy$)
returns the translation by the vector $(dx, dy)$.

TRANSFORM translation(const VECTOR& $v$)
returns the translation by the vector $v$.

TRANSFORM rotation(const POINT& $q$, double $alpha$, double $eps$)
returns the rotation about $q$ by an angle $alpha \pm eps$.

TRANSFORM rotation90(const POINT& $q$)
returns the rotation about $q$ by an angle of 90 degrees.
TRANSFORM reflection(const POINT& q, const POINT& r)
    returns the reflection across the straight line passing through q and r.

TRANSFORM reflection(const POINT& q)
    returns the reflection across point q.
15.26 Point Generators (point generators)

All generators are available for point, rat_point, real_point, d3_point, and d3_rat_point. We use POINT to stand for any of these classes. The corresponding header files are called random_point.h, random_rat_point.h, random_real_point.h, random_d3_point.h, and random_d3_rat_point.h, respectively. These header files are included in the corresponding kernel header files, e.g., random_rat_point.h is part of rat_kernel.h.

We use the following naming conventions: square, circle, segment, and disk refer to two-dimensional objects (even in 3d) and cube, ball, and sphere refer to full-dimensional objects, i.e., in 2d cube and square, ball and disk, and circle and sphere are synonymous.

```c
void random_point_in_square(POINT& p, int maxc)
returns a point whose x and y-coordinates are random integers in \([-maxc..maxc]\). The z-coordinate is zero.

void random_points_in_square(int n, int maxc, list<POINT>& L)
returns a list L of n points . . .

void random_point_in_unit_square(POINT& p, int D = (1 \ll 30) - 1)
returns a point whose coordinates are random ratios of the form \(i/D\) where \(i\) is a random integer in the range \([0..D]\). The default value of \(D\) is \(2^{30} - 1\).

void random_points_in_unit_square(int n, int D, list<POINT>& L)
returns a list L of n points . . . . The default value of \(D\) is used.

void random_point_in_cube(POINT& p, int maxc)
returns a point whose coordinates are random integers in \([-maxc..maxc]\). In 2d this function is equivalent to random_point_in_square.

void random_points_in_cube(int n, int maxc, list<POINT>& L)
returns a list L of n points . . .

void random_point_in_unit_cube(POINT& p, int D = (1 \ll 30) - 1)
returns a point whose coordinates are random ratios of the form \(i/D\) where \(i\) is a random integer in the range \([0..D]\). The default value of \(D\) is \(2^{30} - 1\).

void random_points_in_unit_cube(int n, int D, list<POINT>& L)
returns a list L of n points . . . .
void random_points_in_unit_cube(int n, list<POINT>& L)

as above, but the default value of D is used.

void random_point_in_disc(POINT& p, int R)

returns a random point with integer x and y-coordinates in the disc with radius R centered at the origin. The z-coordinate is zero.

Precondition: \( R \leq 2^{30}\).

void random_points_in_disc(int n, int R, list<POINT>& L)

returns a list L of n points . . .

void random_point_in_unit_disc(POINT& p, int D = (1 \ll 30) - 1)

returns a point in the unit disc whose coordinates are quotients with denominator D. The default value of D is \( 2^{30} - 1 \).

void random_points_in_unit_disc(int n, int D, list<POINT>& L)

returns a list L of n points . . . . The default value of D is used.

void random_point_in_ball(POINT& p, int R)

returns a random point with integer coordinates in the ball with radius R centered at the origin. In 2d this function is equivalent to random_point_in_disc.

Precondition: \( R \leq 2^{30}\).

void random_points_in_ball(int n, int R, list<POINT>& L)

returns a list L of n points . . .

void random_point_in_unit_ball(POINT& p, int D = (1 \ll 30) - 1)

returns a point in the unit ball whose coordinates are quotients with denominator D. The default value of D is \( 2^{30} - 1 \).

void random_points_in_unit_ball(int n, int D, list<POINT>& L)

returns a list L of n points . . .

void random_points_in_unit_ball(int n, list<POINT>& L)

returns a list L of n points . . . . The default value of D is used.

void random_point_near_circle(POINT& p, int R)

returns a random point with integer coordinates that lies close to the circle with radius R centered at the origin.
void random_points_near_circle(int n, int R, list<POINT>& L)
returns a list L of n points . . . .

void random_point_near_unit_circle(POINT& p, int D = (1 << 30) - 1)
returns a point close to the unit circle whose coordinates are quotients with denominator D. The default value of D is $2^{30} - 1$.

void random_points_near_unit_circle(int n, int D, list<POINT>& L)
returns a list L of n points . . . .

void random_points_near_unit_circle(int n, list<POINT>& L)
returns a list L of n points . . . . The default value of D is used.

void random_point_near_sphere(POINT& p, int R)
returns a point with integer coordinates close to the sphere with radius R centered at the origin.

void random_points_near_sphere(int n, int R, list<POINT>& L)
returns a list L of n points . . . .

void random_points_near_sphere(int n, int R, list<POINT>& L)
returns a list L of n points . . . . The default value of D is used.

Wit the rational kernel the functions _on_circle are guaranteed to produce points that lie precisely on the specified circle. With the floating point kernel the functions are equivalent to the _near_circle functions.

void random_point_on_circle(POINT& p, int R, int C = 1000000)
returns a random point with integer coordinates that lies on the circle with radius R centered at the origin.
The point is chosen from a set of at least C candidates.

void random_points_on_circle(int n, int R, list<POINT>& L, int C = 1000000)
returns a list L of n points . . . .

void random_point_on_unit_circle(POINT& p, int C = 1000000)
returns a point on the unit circle. The point is chosen from a set of at least C candidates.
void random_points_on_unit_circle(int n, list<POINT>& L, int C = 1000000) returns a list L of n points . . . .

void random_point_on_sphere(POINT& p, int R) same as random_point_near_sphere.

void random_points_on_sphere(int n, int R, list<POINT>& L) returns a list L of n points . . . .

void random_point_on_unit_sphere(POINT& p, int D = (1 << 30) - 1) same as random_point_near_unit_sphere.

void random_points_on_unit_sphere(int n, int D, list<POINT>& L) returns a list L of n points . . . . The default value of D is used.

void random_point_on_segment(POINT& p, SEGMENT s) generates a random point on s.

void random_points_on_segment(SEGMENT s, int n, list<POINT>& L) generates a list L of n points . . . .

void points_on_segment(SEGMENT s, int n, list<POINT>& L) generates a list L of n equally spaced points on s.

void random_point_on_paraboloid(POINT& p, int maxc) returns a point \((x, y, z)\) with \(x\) and \(y\) random integers in the range \([-\text{maxc}, \text{maxc}]\), and \(z = 0.004 * (x * x + y * y) - 1.25 * \text{maxc}\). The function does not make sense in 2d.

void random_points_on_paraboloid(int n, int maxc, list<POINT>& L) returns a list L of n points . . . .

void lattice_points(int n, int maxc, list<POINT>& L) returns a list L of approximately \(n\) points. The points have integer coordinates \(id/\text{maxc}\) for an appropriately chosen \(d\) and \(-\text{maxc/d} \leq i \leq \text{maxc/d}\).

void random_points_on_diagonal(int n, int maxc, list<POINT>& L) generates \(n\) points on the diagonal whose coordinates are random integer in the range from \(-\text{maxc}\) to \(\text{maxc}\).
15.27 Point on Rational Circle (r_circle_point)

1. Definition

An instance \( p \) of type \( r\_circle\_point \) is a point in the two-dimensional plane that can be obtained by intersecting a rational circle \( c \) and a rational line \( l \) (cf. Sections 15.14 and 15.13). Note that \( c \) and \( l \) may intersect in two points \( p_1 \) and \( p_2 \). Assume that we order these intersections along the (directed) line \( l \). Then \( p \) is uniquely determined by a triple \((c, l, \text{which})\), where \( \text{which} \) is either first or second. Observe that the coordinates of \( p \) are in general non-rational numbers (because their computation involves square roots). Therefore the class \( r\_circle\_point \) is derived from \( real\_point \) (see Section 15.17), which means that all operations of \( real\_point \) are available.

```cpp
#include <LEDA/geo/r_circle_point.h>
```

2. Types

\( r\_circle\_point::\text{tag} \{ \text{first}, \text{second} \} \)

used for selecting between the two possible intersections of a circle and a line.

3. Creation

\( r\_circle\_point \; p; \) creates an instance \( p \) initialized to the point \((0,0)\).

\( r\_circle\_point \; p(\text{const} \; \text{rat}\_\text{point}& \; \text{rat}_\text{pnt}); \)

creates an instance \( p \) initialized to the rational point \( \text{rat}_\text{pnt} \).

\( r\_circle\_point \; p(\text{const} \; \text{point}& \; \text{pnt}); \)

creates an instance \( p \) initialized to the point \( \text{pnt} \).

\( r\_circle\_point \; p(\text{const} \; \text{rat}\_\text{circle}& \; c, \text{const} \; \text{rat}\_\text{line}& \; l, \text{tag} \; \text{which}); \)

creates an instance \( p \) initialized to the point determined by \((c, l, \text{which})\) (see above).

\( r\_circle\_point \; p(\text{const} \; \text{real}\_\text{point}& \; \text{rp}, \text{const} \; \text{rat}\_\text{circle}& \; c, \text{const} \; \text{rat}\_\text{line}& \; l, \text{tag} \; \text{which}); \)

creates an instance \( p \) initialized to the real point \( \text{rp} \).

**Precondition:** \( \text{rp} \) is the point described by \((c, l, \text{which})\).

4. Operations

\textit{void} \quad p.\text{normalize}(); \quad \text{simplifies the internal representation of} \; p.

\textit{rat\_circle} \quad p.\text{supporting\_circle}(); \quad \text{returns a rational circle passing through} \; p.

\textit{rat\_line} \quad p.\text{supporting\_line}(); \quad \text{returns a rational line passing through} \; p.
tag p.intersection() returns whether \( p \) is the first or the second intersection of the supporting circle and the supporting line.

bool p.is_rat_point() returns true, if \( p \) can be converted to rat_point. (The value false means “do not know”.)

const rat_point& p.to_rat_point() converts \( p \) to a rat_point.
Precondition: is_rat_point returns true.

rat_point p.approximate_by_rat_point() approximates \( p \) by a rat_point.

r_circle_point p.round(int prec = 0) returns a rounded representation of \( p \). (experimental)

r_circle_point p.translate(rational dx, rational dy) returns \( p \) translated by vector \((dx, dy)\).

r_circle_point p.translate(const rat_vector& v) returns \( p \) translated by vector \( v \).

r_circle_point p + const rat_vector& v returns \( p \) translated by vector \( v \).

r_circle_point p - const rat_vector& v returns \( p \) translated by vector \(-v\).

r_circle_point p.rotate90(const rat_point& q, int i = 1) returns \( p \) rotated about \( q \) by an angle of \( i \times 90 \) degrees. If \( i > 0 \) the rotation is counter-clockwise otherwise it is clockwise.

r_circle_point p.reflect(const rat_point& p, const rat_point& q) returns \( p \) reflected across the straight line passing through \( p \) and \( q \).

r_circle_point p.reflect(const rat_point& p) returns \( p \) reflected across point \( p \).

bool r_circle_point::intersection(const rat_circle& c, const rat_line& l, tag which, real_point& p) checks whether \((c, l, \text{which})\) is a valid triple, if so the corresponding point is assigned to the real_point \( p \).

bool r_circle_point::intersection(const rat_circle& c, const rat_line& l, tag which, r_circle_point& p) same as above, except for the fact that \( p \) is of type r_circle_point.
15.28 Segment of Rational Circle (r_circle_segment)

1. Definition

An instance $cs$ of type $r\_circle\_segment$ is a segment of a rational circle (see Section 15.14), i.e. a circular arc. A segment is called trivial if it consists of a single point. A non-trivial instance $cs$ is defined by two points $s$ and $t$ (of type $r\_circle\_point$) and an oriented circle $c$ (of type $rat\_circle$) such that $c$ contains both $s$ and $t$. We call $s$ and $t$ the source and the target of $cs$, and $c$ is called its supporting circle. We want to point out that the circle may be a line, which means that $cs$ is a straight line segment. An instance $cs$ is called degenerate, if it is trivial or a straight line segment.

#include <LEDA/geo/r_circle_segment.h>

2. Creation

$r\_circle\_segment$ $cs$; creates a trivial instance $cs$ with source and target equal to the point $(0,0)$.

$r\_circle\_segment$ $cs$(const $r\_circle\_point$& src, const $r\_circle\_point$& tgt, const $rat\_circle$& c);
creates an instance $cs$ with source $src$, target $tgt$ and supporting circle $c$.
Precondition: $src \neq tgt$, $c$ is not trivial and contains $src$ and $tgt$.

$r\_circle\_segment$ $cs$(const $r\_circle\_point$& src, const $r\_circle\_point$& tgt, const $rat\_line$& l);
creates an instance $cs$ with source $src$, target $tgt$ and supporting line $l$.
Precondition: $src \neq tgt$, $l$ contains $src$ and $tgt$.

$r\_circle\_segment$ $cs$(const $rat\_point$& src, const $rat\_point$& middle, const $rat\_point$& tgt);
creates an instance $cs$ with source $src$ and target $tgt$ which passes through $middle$.
Precondition: the three points are distinct.

$r\_circle\_segment$ $cs$(const $r\_circle\_point$& p);
creates a trivial instance $cs$ with source and target equal to $p$.

$r\_circle\_segment$ $cs$(const $rat\_point$& rat_pnt);
creates a trivial instance $cs$ with source and target equal to $rat\_pnt$. 
r\_circle\_segment \ cs(const \ rat\_circle& \ c);
creates an instance \ cs which is equal to the full circle \ c.
Precondition: \ c is not degenerate.

r\_circle\_segment \ cs(const \ rat\_point& \ src, const \ rat\_point& \ tgt);
creates an instance \ cs which is equal to the straight line segment from \ src to \ tgt.

r\_circle\_segment \ cs(const \ rat\_segment& \ s);
creates an instance \ cs which is equal to the straight line segment \ s.

r\_circle\_segment \ cs(const \ r\_circle\_point& \ src, const \ r\_circle\_point& \ tgt);
creates an instance \ cs which is equal to the straight line segment from \ src to \ tgt.
Precondition: Both \ src and \ tgt are rat\_points.

3. Operations

\textit{void} \ cs.normalize() \quad \textit{simplifies the internal representation of} \ cs.

\textit{const r\_circle\_point&} \ cs.source() \quad \textit{returns the source of} \ cs.

\textit{const r\_circle\_point&} \ cs.target() \quad \textit{returns the target of} \ cs.

\textit{const rat\_circle&} \ cs.circle() \quad \textit{returns the supporting circle of} \ cs.

\textit{rat\_line} \ cs.supporting\_line() \quad \textit{returns a line containing} \ cs.
Precondition: \ cs is a straight line segment.

\textit{rat\_point} \ cs.center() \quad \textit{returns the center of the supporting circle of} \ cs.

\textit{int} \ cs.orientation() \quad \textit{returns the orientation (of the supporting circle) of} \ cs.

\textit{real\_point} \ cs.real\_middle() \quad \textit{returns the middle point of} \ cs, i.e. the intersection of \ cs and the bisector of its source and target.

\textit{r\_circle\_point} \ cs.middle() \quad \textit{returns a point on the circle of} \ cs, which is close to real\_middle().

\textit{bool} \ cs.is\_trivial() \quad \textit{returns true iff} \ cs is trivial.

\textit{bool} \ cs.is\_degenerate() \quad \textit{returns true iff} \ cs is degenerate.

\textit{bool} \ cs.is\_full\_circle() \quad \textit{returns true iff} \ cs is a full circle.

\textit{bool} \ cs.is\_proper\_arc() \quad \textit{returns true iff} \ cs is a proper arc, i.e. neither degenerate nor a full circle.
bool cs.is_straight_segment() returns true iff cs is a straight line segment.

bool cs.is_vertical_segment() returns true iff cs is a vertical straight line segment.

bool cs.is_rat_segment() returns true, if cs can be converted to rat_segment. (The value false means “do not know”.)

rat_segment cs.to_rat_segment() converts cs to a rat_segment.
   Precondition: is_rat_segment returns true.

bool cs.contains(const r_circle_point& p) returns true iff cs contains p.

bool cs.overlaps(const r_circle_segment& cs2) returns true iff cs (properly) overlaps cs2.

bool cs.wedge_contains(const real_point& p) returns true iff the (closed) wedge induced by cs contains p. This wedge is spanned by the rays which start at the center and pass through source and target. (Note that p belongs to cs iff p is on the supporting circle and the wedge contains p.)

r_circle_segment cs.reverse() returns the reversal of cs, i.e. source and target are swapped and the supporting circle is reversed.

r_circle_segment cs.round(int prec = 0) returns a rounded representation of cs. (experimental)

r_circle_segment cs.translate(rational dx, rational dy) returns cs translated by vector (dx, dy).

r_circle_segment cs.translate(const rat_vector& v) returns cs translated by vector v.

r_circle_segment cs + const rat_vector& v returns cs translated by vector v.

r_circle_segment cs − const rat_vector& v returns cs translated by vector −v.

r_circle_segment cs.rotate90(const rat_point& q, int i = 1) returns cs rotated about q by an angle of i×90 degrees. If i > 0 the rotation is counter-clockwise otherwise it is clockwise.
\texttt{r\_circle\_segment} \texttt{cs.reflect} (\texttt{const rat\_point}\& \texttt{p, const rat\_point}\& \texttt{q})

returns \texttt{cs} reflected across the straight line passing through \texttt{p} and \texttt{q}.

\texttt{r\_circle\_segment} \texttt{cs.reflect} (\texttt{const rat\_point}\& \texttt{p})

returns \texttt{cs} reflected across point \texttt{p}.

\texttt{list\langle r\_circle\_point\rangle} \texttt{cs.intersection} (\texttt{const rat\_line}\& \texttt{l})

computes \texttt{cs} \cap \texttt{l} (ordered along \texttt{l}).

\texttt{list\langle real\_point\rangle} \texttt{cs.intersection} (\texttt{const real\_line}\& \texttt{l})

as above.

\texttt{list\langle r\_circle\_point\rangle} \texttt{cs.intersection} (\texttt{const rat\_circle}\& \texttt{c})

computes \texttt{cs} \cap \texttt{c} (ordered lexicographically).

\texttt{list\langle r\_circle\_point\rangle} \texttt{cs.intersection} (\texttt{const r\_circle\_segment}\& \texttt{cs2})

computes \texttt{cs} \cap \texttt{cs2} (ordered lexicographically).

\texttt{real} \texttt{cs.sqr\_dist} (\texttt{const real\_point}\& \texttt{p})

computes the squared Euclidean distance between \texttt{cs} and \texttt{p}.

\texttt{real} \texttt{cs.dist} (\texttt{const real\_point}\& \texttt{p})

computes the euclidean distance between \texttt{cs} and \texttt{p}.

\texttt{real\_line} \texttt{cs.tangent\_at} (\texttt{const r\_circle\_point}\& \texttt{p})

computes the tangent to \texttt{cs} at \texttt{p}.

\texttt{Precondition:} \texttt{cs} is not trivial.

\texttt{double} \texttt{cs.approximate\_area} ()

computes the (oriented) area enclosed by the convex hull of \texttt{cs}.

\texttt{void} \texttt{cs.compute\_bounding\_box} (\texttt{real}\& \texttt{xmin, real}\& \texttt{ymin, real}\& \texttt{xmax, real}\& \texttt{ymax})

computes a tight bounding box for \texttt{cs}.

\texttt{list\langle point\rangle} \texttt{cs.approximate} (\texttt{double dist})

approximates \texttt{cs} by a sequence of points. Connecting the points with straight line segments yields a chain with the following property: The maximum distance from a point on \texttt{cs} to the chain is bounded by \texttt{dist}.

\texttt{list\langle rat\_point\rangle} \texttt{cs.approximate\_by\_rat\_points} (\texttt{double dist})

as above, returns \texttt{rat\_points} instead of \texttt{points}. 
list<rat_segment> cs.approximate_by_rat_segments(double dist)
approaches cs by a chain of rat_segments. The maximum distance from a point on cs to the chain is bounded by dist.

bool equal_as_sets(const r_circle_segment& cs1, const r_circle_segment& cs2)
returns whether cs1 and cs2 describe the same set of points.

int compare_tangent_slopes(const r_circle_segment& cs1,
const r_circle_segment& cs2,
const r_circle_point& p)
compares the slopes of the tangents to cs1 and cs2 in the point p.
Precondition: cs1 and cs2 contain p.

We provide the operator << to display an instance cs of type r_circle_segment in a window and the operator >> for reading cs from a window (see real_window.h).

void SWEEP_SEGMENTS(const list<r_circle_segment>& L,
GRAPH<r_circle_point, r_circle_segment>& G,
bool embed = true)
takes as input a list L of r_circle_segments and computes the planar graph G induced by the segments in L. The nodes of G are all endpoints and all proper intersection points of segments in L. The edges of G are the maximal relatively open subsegments of segments in L that contain no node of G. The edges are directed as the corresponding segments, if embed is false. Otherwise, the corresponding planar map is computed. Note that for each edge e G[e] is the input segment containing e.
The algorithm (a variant of [11]) runs in time O((n + s) log n) + m), where n is the number of segments, s is the number of vertices of the graph G, and m is the number of edges of G. If L contains no overlapping segments then m = O(n + s).
15.29 Polygons with circular edges (\texttt{r\_circle\_polygon})

1. Definition

An instance \(P\) of the data type \texttt{r\_circle\_polygon} is a cyclic list of \texttt{r\_circle\_segments}, i.e. straight line or circular segments. A polygon is called simple if all nodes of the graph induced by its segments have degree two and it is called weakly simple, if its segments are disjoint except for common endpoints and if the chain does not cross itself. See the LEDA book for details.

A weakly simple polygon splits the plane into an unbounded region and one or more bounded regions. For a simple polygon there is just one bounded region. When a weakly simple polygon \(P\) is traversed either the bounded region is consistently to the left of \(P\) or the unbounded region is consistently to the left of \(P\). We say that \(P\) is positively oriented in the former case and negatively oriented in the latter case. We use \(P\) to also denote the region to the left of \(P\) and call this region the positive side of \(P\).

The number of segments is called the size of \(P\). A polygon of size zero is trivial; it either describes the empty set or the full two-dimensional plane.

\`
#include <LEDA/geo/r_circle_polygon.h>
``

2. Types

\texttt{r\_circle\_polygon::coord\_type}

the coordinate type (\texttt{real}).

\texttt{r\_circle\_polygon::point\_type}

the point type (\texttt{r\_circle\_point}).

\texttt{r\_circle\_polygon::segment\_type}

the segment type (\texttt{r\_circle\_segment}).

\texttt{r\_circle\_polygon::KIND \{ EMPTY, FULL, NON_TRIVIAL \}}

describes the kind of the polygon: the empty set, the full plane or a non-trivial polygon.

\texttt{r\_circle\_polygon::CHECK\_TYPE \{ NO_CHECK, SIMPLE, WEAKLY_SIMPLE, NOT_WEAKLY_SIMPLE \}}

used to specify which checks should be applied and also describes the outcome of a simplicity check.
15.29. POLYGONS WITH CIRCULAR EDGES (R\_CIRCLE\_POLYGON)

$r\_circle\_polygon :: \text{ RESPECT\_TYPE } \{ \text{ DISREGARD\_ORIENTATION}, \text{ RESPECT\_ORIENTATION} \}$

used in constructors to specify whether to force a positive orientation for the constructed object (DISREGARD\_ORIENTATION) or to keep the orientation of the input (RESPECT\_ORIENTATION).

3. Creation

$r\_circle\_polygon P$; creates an empty polygon $P$.

$r\_circle\_polygon P(KIND k)$; creates a polygon $P$ of kind $k$, where $k$ is either EMPTY or FULL.

$r\_circle\_polygon P(\text{const list}<r\_circle\_segment>& \ chain, \ CHECK\_TYPE check = \text{WEAKLY\_SIMPLE}, \ RESPECT\_TYPE respect\_orient = \text{RESPECT\_ORIENTATION})$; creates a polygon $P$ from a closed chain of segments.

$r\_circle\_polygon P(\text{const list}<\text{rat\_point>>& L, \ CHECK\_TYPE check = \text{WEAKLY\_SIMPLE}, \ RESPECT\_TYPE respect\_orient = \text{RESPECT\_ORIENTATION})$; creates a polygon $P$ with straight line edges from a list $L$ of vertices.

$r\_circle\_polygon P(\text{const rat\_polygon}& Q, \ CHECK\_TYPE check = \text{NO\_CHECK}, \ RESPECT\_TYPE respect\_orient = \text{RESPECT\_ORIENTATION}, \ int \ prec = \text{rat\_point::default\_precision})$; converts a rat\_polygon $Q$ to an r\_circle\_polygon $P$. $P$ is initialized to a rational approximation of $Q$ of coordinates with denominator at most $prec$. If $prec$ is zero, the implementation chooses $prec$ large enough such that there is no loss of precision in the conversion.

$r\_circle\_polygon P(\text{const rat\_circle}& circ, \ RESPECT\_TYPE respect\_orient = \text{RESPECT\_ORIENTATION})$; creates a polygon $P$ whose boundary is the circle $circ$.

4. Operations

$\text{KIND} \quad P.\text{kind}() \quad \text{returns the kind of } P.$

$\text{bool} \quad P.\text{is\_trivial}() \quad \text{returns true iff } P \text{ is trivial}.$

$\text{bool} \quad P.\text{is\_empty}() \quad \text{returns true iff } P \text{ is empty}.$
CHAPTER 15. BASIC DATA TYPES FOR TWO-DIMENSIONAL GEOMETRY

\texttt{bool} \quad \texttt{P.is\_full( )} \quad \text{returns true iff} \ P \ \text{is the full plane.}

\texttt{void} \quad \texttt{P.normalize( )} \quad \text{simplifies the representation by calling} \ s.\texttt{normalize( )} \ \text{for every segment} \ s \ \text{of} \ P.

\texttt{bool} \quad \texttt{P.is\_closed\_chain( )} \quad \text{tests whether} \ P \ \text{is a closed chain.}

\texttt{bool} \quad \texttt{P.is\_simple( )} \quad \text{tests whether} \ P \ \text{is simple.}

\texttt{bool} \quad \texttt{P.is\_weakly\_simple( )} \quad \text{tests whether} \ P \ \text{is weakly simple.}

\texttt{bool} \quad \texttt{P.is\_weakly\_simple(list<r\_circle\_point>& \ crossings)} \quad \text{as above, returns all proper points of intersection in} \ \texttt{crossings}.

\texttt{CHECK\_TYPE} \ \texttt{P.check\_simplicity( )} \quad \text{checks} \ P \ \text{for simplicity. The result can be} \ \texttt{SIMPLE}, \ \texttt{WEAKLY\_SIMPLE} \ \text{or} \ \texttt{NOT\_WEAKLY\_SIMPLE}.

\texttt{bool} \quad \texttt{P.is\_convex( )} \quad \text{returns true iff} \ P \ \text{is convex.}

\texttt{int} \quad \texttt{P.size( )} \quad \text{returns the size of} \ P.

\texttt{const list<r\_circle\_segment>&} \ \texttt{P.segments( )} \quad \text{returns a chain of segments that bound} \ P. \ \text{The orientation of the chain corresponds to the orientation of} \ P.

\texttt{list<r\_circle\_point>} \ \texttt{P.vertices( )} \quad \text{returns the vertices of} \ P.

\texttt{list<r\_circle\_point>} \ \texttt{P.intersection(const r\_circle\_segment&} \ s) \quad \text{returns the list of all proper intersections between} \ s \ \text{and the boundary of} \ P.

\texttt{list<r\_circle\_point>} \ \texttt{P.intersection(const ral\_line&} \ l) \quad \text{returns the list of all proper intersections between} \ l \ \text{and the boundary of} \ P.

\texttt{r\_circle\_polygon} \ \texttt{P.intersection\_halfplane(const ral\_line&} \ l) \quad \text{clips} \ P \ \text{against the halfplane on the positive side of} \ l. \ \text{Observe that the result is only guaranteed to be weakly simple if} \ P \ \text{is convex.}

\texttt{r\_circle\_polygon} \ \texttt{P.translate(rational \ dx, rational \ dy)} \quad \text{returns} \ P \ \text{translated by vector} \ (dx, dy).

\texttt{r\_circle\_polygon} \ \texttt{P.translate(const ral\_vector&} \ v) \quad \text{returns} \ P \ \text{translated by vector} \ v.
$r_{\text{circle\_polygon}} P + \text{ const } \text{rat\_vector} & v$
returns $P$ translated by vector $v$.

$r_{\text{circle\_polygon}} P - \text{ const } \text{rat\_vector} & v$
returns $P$ translated by vector $-v$.

$r_{\text{circle\_polygon}} P.\text{rotate90} \left( \text{ const } \text{rat\_point} & q, \text{ int } i = 1 \right)$
returns $P$ rotated about $q$ by an angle of $i \times 90$ degrees. If $i > 0$ the rotation is counter-clockwise otherwise it is clockwise.

$r_{\text{circle\_polygon}} P.\text{reflect} \left( \text{ const } \text{rat\_point} & p, \text{ const } \text{rat\_point} & q \right)$
returns $P$ reflected across the straight line passing through $p$ and $q$.

$r_{\text{circle\_polygon}} P.\text{reflect} \left( \text{ const } \text{rat\_point} & p \right)$
returns $P$ reflected across point $p$.

$\text{real } P.\text{sqr\_dist} \left( \text{ const } \text{real\_point} & p \right)$
computes the squared Euclidean distance between the boundary of $P$ and $p$. (If $P$ is zero, the result is zero.)

$\text{real } P.\text{dist} \left( \text{ const } \text{real\_point} & p \right)$
computes the Euclidean distance between the boundary of $P$ and $p$. (If $P$ is zero, the result is zero.)

$\text{list}<r_{\text{circle\_polygon}}> P.\text{split\_into\_weakly\_simple\_parts} \left( \right)$
splits $P$ into a set of weakly simple polygons whose union coincides with the inner points of $P$. (This function is experimental.)

$r_{\text{circle\_gen\_polygon}} P.\text{make\_weakly\_simple} \left( \right)$
creates a weakly simple generalized polygon $Q$ from a possibly non-simple polygon $P$ such that $Q$ and $P$ have the same inner points. (This function is experimental.)

$r_{\text{circle\_polygon}} P.\text{complement} \left( \right)$
returns the complement of $P$.

$r_{\text{circle\_polygon}} P.\text{eliminate\_cocircular\_vertices} \left( \right)$
returns a copy of $P$ without cocircular vertices.

$r_{\text{circle\_polygon}} P.\text{round} \left( \text{ int } \text{prec} = 0 \right)$
returns a rounded representation of $P$. (experimental)

$\text{bool } P.\text{is\_rat\_polygon} \left( \right)$
returns whether $P$ can be converted to a $\text{rat\_polygon}$. 

CHAPTER 15. BASIC DATA TYPES FOR TWO-DIMENSIONAL GEOMETRY

rat_polygon  

\texttt{P.to\_rat\_polygon()} \quad \text{converts P to a rat\_polygon.}

\textit{Precondition: is\_rat\_polygon is true.}

rat_polygon  

\texttt{P.approximate\_by\_rat\_polygon(double dist)}

\quad \text{approximates P by a rat\_polygon. The maximum distance between a point on P and the approximation is bounded by dist.}

polygon  

\texttt{P.to\_float()} \quad \text{computes a floating point approximation of P with straight line segments.}

\textit{Precondition: is\_rat\_polygon is true.}

bool  

\texttt{P.is\_rat\_circle()} \quad \text{returns whether P can be converted to a rat\_circle.}

rat\_circle  

\texttt{P.to\_rat\_circle()} \quad \text{converts P to a rat\_circle.}

\textit{Precondition: is\_rat\_circle is true.}

void  

\texttt{P.bounding\_box(real\& xmin, real\& ymin, real\& xmax, real\& ymax)}

\quad \text{computes a tight bounding box for P.}

void  

\texttt{P.bounding\_box(double\& xmin, double\& ymin, double\& xmax, double\& ymax)}

\quad \text{computes a bounding box for P, but not necessarily a tight one.}

\textbf{All functions below assume that P is weakly simple.}

int  

\texttt{P.orientation()} \quad \text{returns the orientation of P.}

int  

\texttt{P.side\_of(const r\_circle\_point\& p)}

\quad \text{return} +1 \text{ if p lies to the left of P, 0 if p lies on P, and } -1 \text{ if p lies to the right of P.}

region\_kind  

\texttt{P.region\_of(const r\_circle\_point\& p)}

\quad \text{return} BOUNDED\_REGION \text{ if p lies in the bounded region of P, returns ON\_REGION if p lies on P, and returns UNBOUNDED\_REGION if p lies in the unbounded region.}

bool  

\texttt{P.inside(const r\_circle\_point\& p)}

\quad \text{return} \text{ true if p lies to the left of P, i.e., } side\_of(p) == +1.

bool  

\texttt{P.on\_boundary(const r\_circle\_point\& p)}

\quad \text{return} \text{ true if p lies on P, i.e., } side\_of(p) == 0.

bool  

\texttt{P.outside(const r\_circle\_point\& p)}

\quad \text{return} \text{ true if p lies to the right of P, i.e., } side\_of(p) == -1.
bool \quad P.\text{contains}(\text{const r\_circle\_point}& p) \\
\quad \text{return} \text{true if } p \text{ lies to the left of or on } P.

double \quad P.\text{approximate\_area}( ) \\
\quad \text{approximates the (oriented) area of the bounded region of } P. \\
\quad \text{Precondition: } P.\text{kind}( ) \text{ is not full.}

\text{r\_circle\_gen\_polygon buffer(double } d) \\
\quad \text{adds an exterior buffer zone to } P \text{ (if } d > 0), \text{ or removes an interior buffer zone from } P \text{ (if } d < 0). \text{ More precisely, for } d \geq 0 \text{ define the buffer tube } T \text{ as the set of all points in the complement of } P \text{ whose distance to } P \text{ is at most } d. \text{ Then the function returns } P \cup T. \text{ For } d < 0 \text{ let } T \text{ denote the set of all points in } P \text{ whose distance to the complement is less than } |d|. \text{ Then the result is } P \setminus T. \text{ Note that the result is a generalized polygon since the buffer of a connected polygon may be disconnected, i.e. consist of several parts, if } d < 0.

\text{Iterations Macros}

forall\_\text{vertices}(v, P) \{ \text{“the vertices of } P \text{ are successively assigned to r\_circle\_point } v\” \} 

forall\_\text{segments}(s, P) \{ \text{“the edges of } P \text{ are successively assigned to the segment } s\” \}
CHAPTER 15. BASIC DATA TYPES FOR TWO-DIMENSIONAL GEOMETRY

15.30 Generalized polygons with circular edges (r_circle_gen_polygon)

1. Definition

The data type \texttt{r\_circle\_polygon} is not closed under boolean operations, e.g., the set difference of a polygon \( P \) and a polygon \( Q \) nested in \( P \) is a region that contains a “hole”. Therefore we provide a generalization called \texttt{r\_circle\_gen\_polygon} which is closed under (regularized) boolean operations (see below).

A formal definition follows: An instance \( P \) of the data type \texttt{r\_circle\_gen\_polygon} is a regular polygonal region in the plane. A regular region is an open set that is equal to the interior of its closure. A region is polygonal if its boundary consists of a finite number of \texttt{r\_circle\_segments}.

The boundary of an \texttt{r\_circle\_gen\_polygon} consists of zero or more weakly simple closed polygonal chains. Each such chain is represented by an object of type \texttt{r\_circle\_polygon}. There are two regions whose boundary is empty, namely the empty region and the full region. The full region encompasses the entire plane. We call a region trivial if its boundary is empty. The boundary cycles \( P_1, P_2, \ldots, P_k \) of an \texttt{r\_circle\_gen\_polygon} are ordered such that no \( P_i \) is nested in a \( P_j \) with \( i < j \).

```c
#include <LEDA/geo/r_circle_gen_polygon.h>
```

2. Types

- \texttt{r\_circle\_gen\_polygon::coord\_type}:
  - the coordinate type (\texttt{real}).

- \texttt{r\_circle\_gen\_polygon::point\_type}:
  - the point type (\texttt{r\_circle\_point}).

- \texttt{r\_circle\_gen\_polygon::segment\_type}:
  - the segment type (\texttt{r\_circle\_segment}).

- \texttt{r\_circle\_gen\_polygon::polygon\_type}:
  - the polygon type (\texttt{r\_circle\_polygon}).

- \texttt{r\_circle\_gen\_polygon::KIND} \{ \texttt{EMPTY}, \texttt{FULL}, \texttt{NON\_TRIVIAL} \}:
  - describes the kind of the polygon: the empty set, the full plane or a non-trivial polygon.

- \texttt{r\_circle\_gen\_polygon::CHECK\_TYPE} \{ \texttt{NO\_CHECK}, \texttt{SIMPLE}, \texttt{WEAKLY\_SIMPLE}, \texttt{NOT\_WEAKLY\_SIMPLE} \}:
  - used to specify which checks should be applied and also describes the outcome of a simplicity check.
15.30. GENERALIZED POLYGONS WITH CIRCULAR EDGES (\texttt{R\_CIRCLE\_GEN\_POLYGON})

\texttt{r\_circle\_gen\_polygon}:: \textit{RESPECT\_TYPE} \{ \texttt{DISREGARD\_ORIENTATION}, \texttt{RESPECT\_ORIENTATION} \}

used in constructors to specify whether to force a positive orientation for the constructed object (\texttt{DISREGARD\_ORIENTATION}) or to keep the orientation of the input (\texttt{RESPECT\_ORIENTATION}).

3. Creation

\texttt{r\_circle\_gen\_polygon} \texttt{P};

creates an empty polygon \texttt{P}.

\texttt{r\_circle\_gen\_polygon} \texttt{P(KIND k)};

creates a polygon \texttt{P} of kind \texttt{k}, where \texttt{k} is either \texttt{EMPTY} or \texttt{FULL}.

\texttt{r\_circle\_gen\_polygon} \texttt{P(const list<r\_circle\_segment>& seg\_chain, CHECK\_TYPE check = WEAKLY\_SIMPLE, RESPECT\_TYPE respect\_orient = RESPECT\_ORIENTATION)};

creates a polygon \texttt{P} from a single closed chain of segments.

\texttt{r\_circle\_gen\_polygon} \texttt{P(const r\_circle\_polygon& Q, CHECK\_TYPE check = NO\_CHECK, RESPECT\_TYPE respect\_orient = RESPECT\_ORIENTATION)};

converts an \texttt{r\_circle\_polygon} \texttt{Q} to an \texttt{r\_circle\_gen\_polygon} \texttt{P}.

\texttt{r\_circle\_gen\_polygon} \texttt{P(const list<rat\_point>& L, CHECK\_TYPE check = NO\_CHECK, RESPECT\_TYPE respect\_orient = RESPECT\_ORIENTATION)};

creates a polygon \texttt{P} with straight line edges from a list \texttt{L} of vertices.

\texttt{r\_circle\_gen\_polygon} \texttt{P(const list<r\_circle\_polygon>& polys, CHECK\_TYPE check = NO\_CHECK, RESPECT\_TYPE respect\_orient = RESPECT\_ORIENTATION)};

introduces a variable \texttt{P} of type \texttt{r\_circle\_gen\_polygon}. \texttt{P} is initialized to the polygon with boundary representation \texttt{polys}.

Precondition: \texttt{polys} must be a boundary representation.

\texttt{r\_circle\_gen\_polygon} \texttt{P(const list<r\_circle\_gen\_polygon>& gen\_polys)};

creates a polygon \texttt{P} as the union of all the polygons in \texttt{gen\_polys}.

Precondition: Every polygon in \texttt{gen\_polys} must be weakly simple.
\( r_{\text{circle\_gen\_polygon}} \) \( P(\text{const } \text{rat\_gen\_polygon}\& Q, \)
\( \text{CHECK\_TYPE check} = \text{NO\_CHECK}, \)
\( \text{RESPECT\_TYPE respect\_orient} = \)
\( \text{RESPECT\_ORIENTATION}); \)
converts a \( \text{rat\_gen\_polygon} \) \( Q \) to an \( r_{\text{circle\_gen\_polygon}} \) \( P \).

\( r_{\text{circle\_gen\_polygon}} \) \( P(\text{const } \text{gen\_polygon}\& Q, \text{CHECK\_TYPE check} = \text{NO\_CHECK}, \)
\( \text{RESPECT\_TYPE respect\_orient} = \)
\( \text{RESPECT\_ORIENTATION}, \)
\( \text{int prec} = \text{rat\_point\::default\_precision}); \)
converts the (floating point) \( \text{gen\_polygon} \) \( Q \) to an \( r_{\text{circle\_gen\_polygon}} \). \( P \) is initialized to a rational approximation of \( Q \) of coordinates with denominator at most \( \text{prec} \). If \( \text{prec} \) is zero, the implementation chooses \( \text{prec} \) large enough such that there is no loss of precision in the conversion.

\( r_{\text{circle\_gen\_polygon}} \) \( P(\text{const } \text{rat\_circle}\& \text{circ}, \text{RESPECT\_TYPE respect\_orient} = \)
\( \text{RESPECT\_ORIENTATION}); \)
creates a polygon \( P \) whose boundary is the circle \( \text{circ} \).

4. Operations

\textit{KIND} \( P.\text{kind()} \) \hspace{1cm} returns the kind of \( P \).

\textit{bool} \( P.\text{is\_trivial()} \) \hspace{1cm} returns true iff \( P \) is trivial.

\textit{bool} \( P.\text{is\_empty()} \) \hspace{1cm} returns true iff \( P \) is empty.

\textit{bool} \( P.\text{is\_full()} \) \hspace{1cm} returns true iff \( P \) is full.

\textit{void} \( P.\text{normalize()} \) \hspace{1cm} simplifies the representation by calling \( c.\text{normalize()} \)
for every polygonal chain \( c \) of \( P \).

\textit{bool} \( P.\text{is\_simple()} \) \hspace{1cm} tests whether \( P \) is simple or not.

\textit{bool} \( P.\text{is\_weakly\_simple()} \) \hspace{1cm} tests whether \( P \) is weakly simple or not.

\textit{bool} \( P.\text{is\_weakly\_simple(list<r\_circle\_point>& crossings)} \)
\hspace{1cm} as above, returns all proper points of intersection in \( crossings \).

\textit{bool} \( r_{\text{circle\_gen\_polygon}}::\text{check\_representation(const list<r\_circle\_polygon>& polys,} \)
\( \text{CHECK\_TYPE check} = \)
\( \text{WEAKLY\_SIMPLE}) \)
checks whether \( polys \) is a boundary representation. Currently the nesting order is not checked, we check only for (weak) simplicity.
15.30. GENERALIZED POLYGONS WITH CIRCULAR EDGES (R\_CIRCLE\_GEN\_POLYGON)

**bool** P.check_representation() checks the representation of P (see above).

**bool** P.is_convex() returns true iff P is convex.

**int** P.size() returns the size of P, i.e. the number of segments in its boundary representation.

**const list<r\_circle\_polygon>&** P.polygons() returns the boundary representation of P.

**list<r\_circle\_segment>** P.edges() returns a chain of segments that bound P. The orientation of the chain corresponds to the orientation of P.

**list<r\_circle\_point>** P.vertices() returns the vertices of P.

**list<r\_circle\_point>** P.intersection(const r\_circle\_segment& s) returns the list of all proper intersections between s and the boundary of P.

**list<r\_circle\_point>** P.intersection(const rat\_line& l) returns the list of all proper intersections between l and the boundary of P.

**r\_circle\_gen\_polygon** P.translate(rational dx, rational dy) returns P translated by vector (dx, dy).

**r\_circle\_gen\_polygon** P.translate(const rat\_vector& v) returns P translated by vector v.

**r\_circle\_gen\_polygon** P + const rat\_vector& v returns P translated by vector v.

**r\_circle\_gen\_polygon** P − const rat\_vector& v returns P translated by vector −v.

**r\_circle\_gen\_polygon** P.rotate90(const rat\_point& q, int i = 1) returns P rotated about q by an angle of i×90 degrees. If i > 0 the rotation is counter-clockwise otherwise it is clockwise.

**r\_circle\_gen\_polygon** P.reflect(const rat\_point& p, const rat\_point& q) returns P reflected across the straight line passing through p and q.

**r\_circle\_gen\_polygon** P.reflect(const rat\_point& p) returns P reflected across point p.
real  \( P.\text{sqr\_dist}(\text{const real\_point} p) \)

computes the squared Euclidean distance between the boundary of \( P \) and \( p \). (If \( P \) is zero, the result is zero.)

real  \( P.\text{dist}(\text{const real\_point} p) \)

computes the Euclidean distance between the boundary of \( P \) and \( p \). (If \( P \) is zero, the result is zero.)

\( r\text{-circle\_gen\_polygon} P.\text{\_make\_weakly\_simple}() \)

creates a weakly simple generalized polygon \( Q \) from a possibly non-simple polygon \( P \) such that \( Q \) and \( P \) have the same inner points. (This function is experimental.)

\( r\text{-circle\_gen\_polygon} r\text{-circle\_gen\_polygon}::\text{\_make\_weakly\_simple}(\text{const r\text{-circle\_polygon} Q}) \)

same as above, but the input is a polygon \( Q \). (This function is experimental.)

\( r\text{-circle\_gen\_polygon} P.\text{\_complement}() \)

returns the complement of \( P \).

\( r\text{-circle\_gen\_polygon} P.\text{\_contour}() \)

returns the contour of \( P \), i.e. all holes are removed from \( P \).

\( r\text{-circle\_gen\_polygon} P.\text{\_eliminate\_cocircular\_vertices}() \)

returns a copy of \( P \) without cocircular vertices.

\( r\text{-circle\_gen\_polygon} P.\text{\_round}(\text{int prec} = 0) \)

returns a rounded representation of \( P \). (experimental)

\( \text{bool} P.\text{\_is\_r\_circle\_polygon}() \)

checks if the boundary of \( P \) consists of at most one chain.

\( r\text{-circle\_polygon} P.\text{\_to\_r\_circle\_polygon}() \)

converts \( P \) to an \( r\text{-circle\_polygon} \).

Precondition: \( \text{is\_r\_circle\_polygon} \) is true.

\( \text{bool} P.\text{\_is\_rat\_gen\_polygon}() \)

returns whether \( P \) can be converted to a \( \text{rat\_polygon} \).

\( \text{rat\_gen\_polygon} P.\text{\_to\_rat\_gen\_polygon}() \)

converts \( P \) to a \( \text{rat\_gen\_polygon} \).

Precondition: \( \text{is\_rat\_gen\_polygon} \) is true.

\( \text{rat\_gen\_polygon} P.\text{\_approximate\_by\_rat\_gen\_polygon}(\text{double dist}) \)

approximates \( P \) by a \( \text{rat\_gen\_polygon} \). The maximum distance between a point on \( P \) and the approximation is bounded by \( \text{dist} \).
**gen_polygon**  
`P.to_float()` computes a floating point approximation of $P$ with straight line segments.  
*Precondition:* `is_rat_gen_polygon` is true.

**bool**  
`P.is_rat_circle()` returns whether $P$ can be converted to a `rat_circle`.

**rat_circle**  
`P.to_rat_circle()` converts $P$ to a `rat_circle`.  
*Precondition:* `is_rat_circle` is true.

**void**  
`P.bounding_box(real & xmin, real & ymin, real & xmax, real & ymax)` computes a tight bounding box for $P$.

**void**  
`P.bounding_box(double & xmin, double & ymin, double & xmax, double & ymax)` computes a bounding box for $P$, but not necessarily a tight one.

**All functions below assume that $P$ is weakly simple.**

**int**  
`P.orientation()` returns the orientation of $P$.

**int**  
`P.side_of(const r_circle_point & p)`  
returns +1 if $p$ lies to the left of $P$, 0 if $p$ lies on $P$, and −1 if $p$ lies to the right of $P$.

**region_kind**  
`P.region_of(const r_circle_point & p)`  
returns `BOUNDED_REGION` if $p$ lies in the bounded region of $P$, returns `ON_REGION` if $p$ lies on $P$, and returns `UNBOUNDED_REGION` if $p$ lies in the unbounded region. The bounded region of the full polygon is the entire plane.

**bool**  
`P.inside(const r_circle_point & p)`  
returns true if $p$ lies to the left of $P$, i.e., `side_of(p) == +1`.

**bool**  
`P.on_boundary(const r_circle_point & p)`  
returns true if $p$ lies on $P$, i.e., `side_of(p) == 0`.

**bool**  
`P.outside(const r_circle_point & p)`  
returns true if $p$ lies to the right of $P$, i.e.,  
`side_of(p) == -1`.

**bool**  
`P.contains(const r_circle_point & p)`  
returns true if $p$ lies to the left of or on $P$. 
double P.approximate_area() approximates the (oriented) area of the bounded region of P.

Precondition: P.kind( ) is not full.

All boolean operations are regularized, i.e., the result R of the standard boolean operation is replaced by the interior of the closure of R. We use reg X to denote the regularization of a set X.

\[ r_{\text{circle_gen_polygon}} P.\text{unite}(\text{const } r_{\text{circle_gen_polygon}}& Q) \]
\[ \text{return } \text{reg}(P \cup Q). \]

\[ r_{\text{circle_gen_polygon}} P.\text{intersection}(\text{const } r_{\text{circle_gen_polygon}}& Q) \]
\[ \text{return } \text{reg}(P \cap Q). \]

\[ r_{\text{circle_gen_polygon}} P.\text{diff}(\text{const } r_{\text{circle_gen_polygon}}& Q) \]
\[ \text{return } \text{reg}(P \setminus Q). \]

\[ r_{\text{circle_gen_polygon}} P.\text{sym_diff}(\text{const } r_{\text{circle_gen_polygon}}& Q) \]
\[ \text{return } \text{reg}((P \cup Q) \setminus (P \cap Q)). \]

For optimization purposes we provide a union operation of arbitrary arity. It computes the union of a set of polygons much faster than with binary operations.

\[ r_{\text{circle_gen_polygon}} r_{\text{circle_gen_polygon}}::\text{unite}(\text{const list< } r_{\text{circle_gen_polygon}}& L) \]
\[ \text{return the (regularized) union of all polygons in } L. \]

We offer fast versions of the boolean operations which compute an approximate result. These operations work as follows: every curved segment is approximated by straight line segments, then the respective boolean operation is performed on the straight polygons. Finally, we identify those straight segments in the result that originate from a curved segment and replace them by curved segments again. (We denote the approximate computation of an operation op scheme by appr(op).) Every operation below takes a parameter dist that controls the accuracy of the approximation: dist is an upper bound on the distance of any point on an original polygon P to the approximated polygon P'.

\[ r_{\text{circle_gen_polygon}} P.\text{unite_approximate}(\text{const } r_{\text{circle_gen_polygon}}& Q, \text{double dist } = 1e^{-2}) \]
\[ \text{return appr}(P \cup Q). \]

\[ r_{\text{circle_gen_polygon}} P.\text{intersection_approximate}(\text{const } r_{\text{circle_gen_polygon}}& Q, \text{double dist } = 1e^{-2}) \]
\[ \text{return appr}(P \cap Q). \]

\[ r_{\text{circle_gen_polygon}} P.\text{diff_approximate}(\text{const } r_{\text{circle_gen_polygon}}& Q, \text{double dist } = 1e^{-2}) \]
\[ \text{return appr}(P \setminus Q). \]
15.30. GENERALIZED POLYGONS WITH CIRCULAR EDGES (\texttt{R\_CIRCLE\_GEN\_POLYGON})

\texttt{r\_circle\_gen\_polygon} \texttt{P.sym\_diff\_approximate(const r\_circle\_gen\_polygon& Q,}

\texttt{double dist = 1e-2)}

returns \texttt{appr((P \cup Q) - (P \cap Q))}.

\texttt{r\_circle\_gen\_polygon r\_circle\_gen\_polygon::unite\_approximate(const list<r\_circle\_gen\_polygon>& L,}

\texttt{double dist = 1e-2)}

returns the (approximated) union of all polygons in \texttt{L}.

\texttt{r\_circle\_gen\_polygon P.buffer(double d)}

adds an exterior buffer zone to \texttt{P(d > 0)}, or removes an interior buffer zone from \texttt{P(d < 0)}. More precisely, for \(d \geq 0\) define the buffer tube \(T\) as the set of all points in the complement of \(P\) whose distance to \(P\) is at most \(d\). Then the function returns \(P \cup T\). For \(d < 0\) let \(T\) denote the set of all points in \(P\) whose distance to the complement is less than \(|d|\). Then the result is \(P \setminus T\).

\texttt{Iterations Macros}

\texttt{forall\_polygons(p, P) \{ \texttt{“the boundary polygons of P are successively assigned to r\_circle\_polygon p”} \}}
15.31 Parser for well known binary format ( wkb_io )

1. Definition

The class \textit{wkb\_io} provides methods for reading and writing geometries in the well known binary format (wkb). Every non-trivial generalized polygon from LEDA can be written in wkb format. The method for reading supports the wkb types \textit{Polygon} and \textit{MultiPolygon}, i.e., those types that can be represented by the LEDA type \textit{gen\_polygon}.

\begin{verbatim}
#include <LEDA/beta/geo/wkb_io.h>
\end{verbatim}

2. Creation

\textit{wkb\_io} \textit{W}; creates an instance of type \textit{wkb\_io}.

3. Operations

\begin{verbatim}
bool W.read(const string& filename, gen_polygon& P)    
    reads the geometry stored in the given file and converts it to a generalized polygon \textit{P}.

bool W.write(const string& filename, const gen_polygon& P)  
    writes the generalized polygon \textit{P} to the given file.
\end{verbatim}
Chapter 16

Advanced Data Types for Two-Dimensional Geometry

16.1 Two-Dimensional Dictionaries (d2_dictionary)

1. Definition

An instance $D$ of the parameterized data type $d2\_dictionary<K1, K2, I>$ is a collection of items ($dic2\_item$). Every item in $D$ contains a key from the linearly ordered data type $K1$, a key from the linearly ordered data type $K2$, and an information from data type $I$. $K1$ and $K2$ are called the key types of $D$, and $I$ is called the information type of $D$. If $K1$ resp. $K2$ are user-defined types, you have to provide a compare function for type $K1$ resp. $K2$ (see Section 2.3). The number of items in $D$ is called the size of $D$. A two-dimensional dictionary of size zero is said to be empty. We use $<k_1, k_2, i>$ to denote the item with first key $k_1$, second key $k_2$, and information $i$. For each pair $(k_1, k_2) \in K1 \times K2$ there is at most one item $<k_1, k_2, i> \in D$. Additionally to the normal dictionary operations, the data type $d2\_dictionary$ supports rectangular range queries on $K1 \times K2$.

#include <LEDA/geo/d2_dictionary.h>

2. Creation

$d2\_dictionary<K1, K2, I> \ D;$

creates an instance $D$ of type $d2\_dictionary<K1, K2, I>$ and initializes $D$ to the empty dictionary.

3. Operations

$const K1& \quad D.key1(dic2\_item \ it) \quad$ returns the first key of item $it$.
$Precondition: \ it \ is \ an \ item \ in \ D.$

$const K2& \quad D.key2(dic2\_item \ it) \quad$ returns the second key of item $it$.
$Precondition: \ it \ is \ an \ item \ in \ D.$
const I&  D.inf(dic2_item it) returns the information of item it.
   Precondition: it is an item in D.
I&   D[dic2_item it] returns a reference to the information of item it.
     Precondition: it is an item in D.

dic2_item  D.min_key1() returns the item with minimal first key.
dic2_item  D.min_key2() returns the item with minimal second key.
dic2_item  D.max_key1() returns the item with maximal first key.
dic2_item  D.max_key2() returns the item with maximal second key.
dic2_item  D.insert(const K1& x, const K2& y, const I& i)
     associates the information i with the keys x and y. If there is an item <x, y, j> in D then
     j is replaced by i, else a new item <x, y, i> is added to D. In both cases the item is retunred.
dic2_item  D.lookup(const K1& x, const K2& y)
     returns the item with keys x and y (nil if no such item exists in D).
list<dic2_item>  D.range_search(const K1& x0, const K1& x1, const K2& y0,
                            const K2& y1)
     returns the list of all items <k1, k2, i> in D with x0 ≤ k1 ≤ x1 and y0 ≤ k2 ≤ y1.
list<dic2_item>  D.all_items()
     returns the list of all items of D.
void  D.del(const K1& x, const K2& y) deletes the item with keys x and y from D.
void  D.del_item(dic2_item it) removes item it from D.
     Precondition: it is an item in D.
void  D.change_inf(dic2_item it, const I& i)
     makes i the information of item it.
     Precondition: it is an item in D.
void  D.clear()
     makes D the empty d2_dictionary.
bool  D.empty()
     returns true if D is empty, false otherwise.
int  D.size()
     returns the size of D.
4. Implementation

Two-dimensional dictionaries are implemented by dynamic two-dimensional range trees \[ \text{[90, 57]} \] based on BB[\( \alpha \) trees. Operations insert, lookup, del_item, del take time \( O(\log^2 n) \), range_search takes time \( O(k + \log^2 n) \), where \( k \) is the size of the returned list, key, inf, empty, size, change_inf take time \( O(1) \), and clear takes time \( O(n \log n) \). Here \( n \) is the current size of the dictionary. The space requirement is \( O(n \log n) \).
16.2 Point Sets and Delaunay Triangulations (POINT_SET)

1. Definition

There are three instantiations of POINT_SET: point_set (floating point kernel), rat_point_set (rational kernel) and real_point_set (real kernel). The respective header file name corresponds to the type name (with “.h” appended).

An instance $T$ of data type POINT_SET is a planar embedded bidirected graph (map) representing the Delaunay Triangulation of its vertex set. The position of a vertex $v$ is given by $T$.pos($v$) and we use $S = \{T$.pos($v$) $| v \in T\}$ to denote the underlying point set. Each face of $T$ (except for the outer face) is a triangle whose circumscribing circle does not contain any point of $S$ in its interior. For every edge $e$, the sequence

$$e, T$.face_cycle_succ(e), T$.face_cycle_succ(T$.face_cycle_succ(e)), \ldots$$

traces the boundary of the face to the left of $e$. The edges of the outer face of $T$ form the convex hull of $S$; the trace of the convex hull is clockwise. The subgraph obtained from $T$ by removing all diagonals of co-circular quadrilaterals is called the Delaunay Diagram of $S$.

POINT_SET provides all constant graph operations, e.g., $T$.reversal($e$) returns the reversal of edge $e$, $T$.all_edges( ) returns the list of all edges of $T$, and $forall_edges(e, T)$ iterates over all edges of $T$. In addition, POINT_SET provides operations for inserting and deleting points, point location, nearest neighbor searches, and navigation in both the triangulation and the diagram.

POINT_SETs are essentially objects of type GRAPH<POINT, int>, where the node information is the position of the node and the edge information is irrelevant. For a graph $G$ of type GRAPH<POINT, int> the function Is_Delaunay($G$) tests whether $G$ is a Delaunay triangulation of its vertices.

The data type POINT_SET is illustrated by the point_set_demo in the LEDA demo directory.

Be aware that the nearest neighbor queries for a point (not for a node) and the range search queries for circles, triangles, and rectangles are non-const operations and modify the underlying graph. The set of nodes and edges is not changed; however, it is not guaranteed that the underlying Delaunay triangulation is unchanged.

```c
#include <LEDA/geo/generic/POINT_SET.h>
```

2. Creation

POINT_SET $T$; creates an empty POINT_SET $T$. 
16.2. POINT SETS AND DELAUNAY TRIANGULATIONS (POINT_SET)

POINT_SET \( T(\text{const list<POINT>& } S); \)

creates a POINT_SET \( T \) of the points in \( S \). If \( S \) contains multiple occurrences of points only the last occurrence of each point is retained.

POINT_SET \( T(\text{const GRAPH<POINT, int>& } G); \)

initializes \( T \) with a copy of \( G \).

Precondition: Is_Delaunay(\( G \)) is true.

3. Operations

void \( T.\text{init(\text{const list<POINT>& } L)} \)

makes \( T \) a POINT_SET for the points in \( S \).

POINT \( T.\text{pos(node } v) \)

returns the position of node \( v \).

POINT \( T.\text{pos_source(edge } e) \)

returns the position of source(\( e \)).

POINT \( T.\text{pos_target(edge } e) \)

returns the position of target(\( e \)).

SEGMENT \( T.\text{seg(edge } e) \)

returns the line segment corresponding to edge \( e \) (SEGMENT(\( T.\text{pos_source}(e), T.\text{pos_target}(e)) \)).

LINE \( T.\text{supporting_line(edge } e) \)

returns the supporting line of edge \( e \) (LINE(\( T.\text{pos_source}(e), T.\text{pos_target}(e)) \)).

int \( T.\text{orientation(edge } e, \text{ POINT } p) \)

returns orientation(\( T.\text{seg}(e), p \)).

int \( T.\text{dim()} \)

returns \(-1\) if \( S \) is empty, returns \( 0 \) if \( S \) consists of only one point, returns \( 1 \) if \( S \) consists of at least two points and all points in \( S \) are collinear, and returns \( 2 \) otherwise.

list<POINT> \( T.\text{points()} \)

returns \( S \).

bool \( T.\text{get_bounding_box(POINT}& \text{ lower_left, POINT}& \text{ upper_right)} \)

returns the lower left and upper right corner of the bounding box of \( T \). The operation returns true, if \( T \) is not empty; false otherwise.

list<node> \( T.\text{get_convex_hull()} \)

returns the convex hull of \( T \).

edge \( T.\text{get_hull_dart()} \)

returns a dart of the outer face of \( T \) (i.e., a dart of the convex hull).

edge \( T.\text{get_hull_edge()} \)

as above.

bool \( T.\text{is_hull_dart(edge } e) \)

returns true if \( e \) is a dart of the convex hull of \( T \), i.e., a dart on the face cycle of the outer face.
bool

T.is_hull_edge(edge e) as above.

bool

T.is_diagram_dart(edge e) returns true if e is a dart of the Delaunay diagram, i.e., either a dart on the convex hull or a dart where the incident triangles have distinct circumcircles.

bool

T.is_diagram_edge(edge e) as above.

double

T.d_face_cycle_succ(edge e) returns the face cycle successor of e in the Delaunay diagram of T. Precondition: e belongs to the Delaunay diagram.

double

T.d_face_cycle_pred(edge e) returns the face cycle predecessor of e in the Delaunay diagram of T. Precondition: e belongs to the Delaunay diagram.

bool

T.empty() decides whether T is empty.

void

T.clear() makes T empty.

double

T.locate(POINT p, edge loc_start = NULL)
returns an edge e of T that contains p or that borders the face that contains p. In the former case, a hull dart is returned if p lies on the boundary of the convex hull. In the latter case we have T.orientation(e, p) > 0 except if all points of T are collinear and p lies on the induced line. In this case target(e) is visible from p. The function returns nil if T has no edge. The optional second argument is an edge of T, where the locate operation starts searching.

double

T.locate(POINT p, const list<edge>& loc_start)
returns locate(p, e) with e in loc_start. If loc_start is empty, we return locate(p, NULL). The operation tries to choose a good starting edge for the locate operation from loc_start.
Precondition: All edges in loc_start must be edges of T.

node

T.lookup(POINT p, edge loc_start = NULL)
if T contains a node v with pos(v) = p the result is v otherwise the result is nil. The optional second argument is an edge of T, where the locate operation starts searching p.
node \( T.\text{lookup}(\text{POINT } p, \text{const } \text{list<}\text{edge}\text{>& } \text{loc}\text{\_start}) \)

returns \( \text{lookup}(p, e) \) with \( e \) in \( \text{loc}\_\text{start} \). If \( \text{loc}\_\text{start} \) is empty, we return \( \text{lookup}(p, \text{NULL}) \). The operation tries to choose a good starting edge for the \( \text{lookup} \) operation from \( \text{loc}\_\text{start} \).

Precondition: All edges in \( \text{loc}\_\text{start} \) must be edges of \( T \).

node \( T.\text{insert}(\text{POINT } p) \)

inserts point \( p \) into \( T \) and returns the corresponding node. More precisely, if there is already a node \( v \) in \( T \) positioned at \( p \) (i.e., \( \text{pos}(v) \) is equal to \( p \) then \( \text{pos}(v) \) is changed to \( p \) (i.e., \( \text{pos}(v) \) is made identical to \( p \)) and if there is no such node then a new node \( v \) with \( \text{pos}(v) = p \) is added to \( T \). In either case, \( v \) is returned.

void \( T.\text{del}(\text{node } v) \)

removes the node \( v \), i.e., makes \( T \) a Delaunay triangulation for \( S \setminus \{\text{pos}(v)\} \).

void \( T.\text{del}(\text{POINT } p) \)

removes the node \( p \), i.e., makes \( T \) a Delaunay triangulation for \( S \setminus p \).

node \( T.\text{nearest}\_\text{neighbor}(\text{POINT } p) \)

computes a node \( v \) of \( T \) that is closest to \( p \), i.e.,
\[
\text{dist}(p, \text{pos}(v)) = \min\{\text{dist}(p, \text{pos}(u)) \mid u \in T\}.
\]

This is a non-const operation.

node \( T.\text{nearest}\_\text{neighbor}(\text{node } w) \)

computes a node \( v \) of \( T \) that is closest to \( p = T[w] \), i.e.,
\[
\text{dist}(p, \text{pos}(v)) = \min\{\text{dist}(p, \text{pos}(u)) \mid u \in T\}.
\]

list<node> \( T.\text{nearest}\_\text{neighbors}(\text{POINT } p, \text{int } k) \)

returns the \( k \) nearest neighbors of \( p \), i.e., a list of the \( \min(k, |S|) \) nodes of \( T \) closest to \( p \). The list is ordered by distance from \( p \). This is a non-const operation.

list<node> \( T.\text{nearest}\_\text{neighbors}(\text{node } w, \text{int } k) \)

returns the \( k \) nearest neighbors of \( p = T[w] \), i.e., a list of the \( \min(k, |S|) \) nodes of \( T \) closest to \( p \). The list is ordered by distance from \( p \).

list<node> \( T.\text{range}\_\text{search}(\text{const } \text{CIRCLE}& C) \)

returns the list of all nodes contained in the closure of disk \( C \).

Precondition: \( C \) must be a proper circle (not a straight line). This is a non-const operation.
list<node>  \( T.\text{range\_search}(\text{node } v, \text{const } \text{POINT} & p) \)
returns the list of all nodes contained in the
closure of disk \( C \) with center \( \text{pos}[v] \) and having
\( p \) in its boundary.

list<node>  \( T.\text{range\_search}(\text{const } \text{POINT} & a, \text{const } \text{POINT} & b, \text{const } \text{POINT} & c) \)
returns the list of all nodes contained in the
closure of the triangle \((a, b, c)\).
Precondition: \( a, b, \) and \( c \) must not be collinear.
This is a non-const operation.

list<node>  \( T.\text{range\_search\_parallelogram}(\text{const } \text{POINT} & a, \text{const } \text{POINT} & b, \\
\text{const } \text{POINT} & c) \)
returns the list of all nodes contained in the
closure of the parallelogram \((a, b, c, d)\) with \( d = a + (c - b) \).
Precondition: \( a, b, \) and \( c \) must not be collinear.
This is a non-const operation.

list<node>  \( T.\text{range\_search}(\text{const } \text{POINT} & a, \text{const } \text{POINT} & b) \)
returns the list of all nodes contained in the clo-
sure of the rectangle with diagonal \((a, b)\). This
is a non-const operation.

list<node>  \( T.\text{minimum\_spanning\_tree}() \)
returns the list of edges of \( T \) that comprise a
minimum spanning tree of \( S \).

list<node>  \( T.\text{relative\_neighborhood\_graph}() \)
returns the list of edges of \( T \) that comprise a
relative neighborhood graph of \( S \).

void  \( T.\text{compute\_voronoi}(\text{GRAPH<\text{CIRCLE}, \text{POINT}> } & V) \)
computes the corresponding Voronoi diagram
\( V \). Each node of \( VD \) is labeled with its defining
circle. Each edge is labeled with the site lying
in the face to its left.

**Drawing Routines**

The functions in this section were designed to support the drawing of Delaunay triangu-
lations and Voronoi diagrams.

void  \( T.\text{draw\_nodes}() \)
\(<\text{void } (\ast\text{draw\_node})(\text{const } \text{POINT} & )) \>
calls \( \text{draw\_node}(\text{pos}(v)) \) for every node \( v \) of \( T \).
void T.draw_edge(edge e, void (*draw_diagram_edge)(const POINT&, const POINT&)),
void (*draw_triang_edge) (const POINT&, const POINT&),
void (*draw_hull_dart) (const POINT&, const POINT&))
calls draw_diagram_edge(pos_source(e), pos_target(e) if e is a diagram dart, draw_hull_dart(pos_source(e), pos_target(e)) if e is a hull dart, and draw_triang_edge(pos_source(e), pos_target(e)) if e is a non-diagram edge.

void T.draw_edges(void (*draw_diagram_edge)(const POINT&, const POINT&)),
void (*draw_triang_edge) (const POINT&, const POINT&),
void (*draw_hull_dart) (const POINT&, const POINT&))
calls the corresponding function for all edges of T.

void T.draw_edges(const list<edge>& L, void (*draw_edge)(const POINT&, const POINT&))
calls draw_edge(pos_source(e), pos_target(e) for every edge e ∈ L.

void T.draw_voro_edges(void (*draw_edge)(const POINT&, const POINT&)),
void (*draw_ray) (const POINT&, const POINT&))
calls draw_edge and draw_ray for the edges of the Voronoi diagram.

void T.draw_hull(void (*draw_poly)(const list<POINT>&))
calls draw_poly with the list of vertices of the convex hull.

void T.draw_voro(const GRAPH<CIRCLE, POINT>&,
void (*draw_node)(const POINT&),
void (*draw_edge)(const POINT&, const POINT&),
void (*draw_ray) (const POINT&, const POINT&))
calls ...

4. Implementation

The main ingredients for the implementation are Delaunay flipping, segment walking, and plane sweep.

The constructor POINT_SET(list<POINT> S) first constructs a triangulation of S by sweeping and then makes the triangulation Delaunay by a sequence of Delaunay flips.

Locate walks through the triangulation along the segment from some fixed point of T to the query point. Insert first locates the point, then updates the triangulation locally, and finally performs flips to reestablish the Delaunay property. Delete deletes the node, retriangulates the resulting face, and then performs flips. Nearest neighbor searching, circular range queries, and triangular range queries insert the query point into the triangulation, then perform an appropriate graph search on the triangulation, and finally remove the query point.

All algorithms show good expected behavior.

For details we refer the reader to the LEDA implementation report "Point Sets and Dynamic Delaunay Triangulations".
16.3 Sets of Intervals (interval_set)

1. Definition

An instance S of the parameterized data type interval_set<I> is a collection of items (is_item). Every item in S contains a closed interval of the double numbers as key and an information from data type I, called the information type of S. The number of items in S is called the size of S. An interval set of size zero is said to be empty. We use <x, y, i> to denote the item with interval [x, y] and information i; x (y) is called the left (right) boundary of the item. For each interval [x, y] there is at most one item <x, y, i> ∈ S.

#include <LEDA/geo/interval_set.h>

2. Creation

interval_set<I> S; creates an instance S of type interval_set<I> and initializes S to the empty set.

3. Operations

double S.left(is_item it) returns the left boundary of item it. Precondition: it is an item in S.

double S.right(is_item it) returns the right boundary of item it. Precondition: it is an item in S.

const I& S.inf(is_item it) returns the information of item it. Precondition: it is an item in S.

is_item S.insert(double x, double y, const I& i) associates the information i with interval [x, y]. If there is an item <x, y, j> in S then j is replaced by i, else a new item <x, y, i> is added to S. In both cases the item is returned.

is_item S.lookup(double x, double y) returns the item with interval [x, y] (nil if no such item exists in S).

list<is_item> const S.intersection(double a, double b) returns all items <x, y, i> ∈ S with [x, y] ∩ [a, b] ≠ ∅.

void S.del(double x, double y) deletes the item with interval [x, y] from S.

void S.del_item(is_item it) removes item it from S. Precondition: it is an item in S.
void S.change_inf(is_item it, const I& i)  

makes i the information of item it.  
Precondition: it is an item in S.

void S.clear() makes S the empty interval_set.

bool S.empty() returns true iff S is empty.

int S.size() returns the size of S.

4. Implementation

Interval sets are implemented by two-dimensional range trees [90, 57]. Operations insert, lookup, del_item and del take time $O(\log^2 n)$, intersection takes time $O(k + \log^2 n)$, where $k$ is the size of the returned list. Operations left, right, inf, empty, and size take time $O(1)$, and clear $O(n \log n)$. Here $n$ is always the current size of the interval set. The space requirement is $O(n \log n)$.
16.4 Sets of Parallel Segments (segment_set)

1. Definition

An instance $S$ of the parameterized data type $\text{segment\_set}<I>$ is a collection of items ($\text{seg\_item}$). Every item in $S$ contains as key a line segment with a fixed direction $\alpha$ (see data type segment) and an information from data type $I$, called the information type of $S$. $\alpha$ is called the orientation of $S$. We use $<s,i>$ to denote the item with segment $s$ and information $i$. For each segment $s$ there is at most one item $<s,i> \in S$.

```c
#include <LEDA/geo/segment_set.h>
```

2. Creation

- $\text{segment\_set}<I>$ $S$; creates an empty instance $S$ of type $\text{segment\_set}<I>$ with orientation zero, i.e., horizontal segments.
- $\text{segment\_set}<I>$ $S$(int $rot$); creates an empty instance $S$ of type $\text{segment\_set}<I>$ with orientation $rot \cdot \pi/2$.
- $\text{segment\_set}<I>$ $S$(double $a$); creates an empty instance $S$ of type $\text{segment\_set}<I>$ with orientation $a$. (Note that there may be incorrect results due to rounding errors.)

3. Operations

- $\text{segment}$ $S$.key($\text{seg\_item}$ $it$) returns the segment of item $it$.
  
  **Precondition:** it is an item in $S$.
- $\text{const I &}$ $S$.inf($\text{seg\_item}$ $it$) returns the information of item $it$.
  
  **Precondition:** it is an item in $S$.
- $\text{seg\_item}$ $S$.insert($\text{const segment &}$ $s$, $\text{const I &}$ $i$) associates the information $i$ with segment $s$. If there is an item $<s,j>$ in $S$ then $j$ is replaced by $i$, else a new item $<s,i>$ is added to $S$. In both cases the item is returned.
- $\text{seg\_item}$ $S$.lookup($\text{const segment &}$ $s$) returns the item with segment $s$ (nil if no such item exists in $S$).
- $\text{list<seg\_item>}$ $S$.intersection($\text{const segment &}$ $q$) returns all items $<s,i> \in S$ with $s \cap q \neq \emptyset$.
  
  **Precondition:** $q$ is orthogonal to the segments in $S$. 
list<seg_item> S.intersection_sorted(const segment& q)

as above, but the returned segments are ordered
as they are intersected by q if one travels from
q.source() to q.target().

list<seg_item> S.intersection(const line& l)

returns all items <s,i> ∈ S with s ∩ l ≠ ∅.
Precondition: l is orthogonal to the segments in S.

list<seg_item> S.intersection_sorted(const line& l)

as above, but the returned segments are ordered
as they are intersected by l if one travels along
l in direction l.direction().

void S.del(const segment& s) deletes the item with segment s from S.

void S.del_item(seg_item it) removes item it from S.
Precondition: it is an item in S.

void S.change_inf(seg_item it, const I& i)

makes i the information of item it.
Precondition: it is an item in S.

void S.clear() makes S the empty segment_set.

bool S.empty() returns true iff S is empty.

int S.size() returns the size of S.

4. Implementation

Segment sets are implemented by dynamic segment trees based on BB[α] trees ([90, 57])
trees. Operations key, inf, change_inf, empty, and size take time O(1), insert, lookup, del,
and del_item take time O(log^2 n) and an intersection operation takes time O(k + log^2 n),
where k is the size of the returned list. Here n is the current size of the set. The space
requirement is O(n log n).
16.5 Sets of Parallel Rational Segments (rat_segment_set)

1. Definition

An instance $S$ of the parameterized data type $\text{rat\_segment\_set}<I>$ is a collection of items ($\text{seg\_item}$). Every item in $S$ contains as key a rational line segment with a fixed direction $\alpha$ (see data type $\text{rat\_segment}$) and an information from data type $I$, called the information type of $S$. $\alpha$ is called the orientation of $S$, it must be a multiple of $\pi/2$. We use $<s,i>$ to denote the item with segment $s$ and information $i$. For each segment $s$ there is at most one item $<s,i> \in S$.

```
#include <LEDA/geo/rat_segment_set.h>
```

2. Creation

```
rat_segment_set<I> S;
```

creates an empty instance $S$ of type $\text{rat\_segment\_set}<I>$ with orientation zero, i.e., horizontal segments.

```
rat_segment_set<I> S(int rot);
```

creates an empty instance $S$ of type $\text{rat\_segment\_set}<I>$ with orientation $\text{rot} \cdot \pi/2$.

3. Operations

```
rat_segment S.key(seg_item it) returns the segment of item $it$.
Precondition: $it$ is an item in $S$.
```

```
const I& S.inf(seg_item it) returns the information of item $it$.
Precondition: $it$ is an item in $S$.
```

```
seg_item S.insert(const rat_segment& s, const I& i)
associates the information $i$ with segment $s$. If there is an item $<s,j>$ in $S$ then $j$ is replaced by $i$, else a new item $<s,i>$ is added to $S$. In both cases the item is returned.
```

```
seg_item S.lookup(const rat_segment& s)
returns the item with segment $s$ (nil if no such item exists in $S$).
```

```
list<seg_item> S.intersection(const rat_segment& q)
returns all items $<s,i> \in S$ with $s \cap q \neq \emptyset$.
Precondition: $q$ is orthogonal to the segments in $S$.
```
16.5. SETS OF PARALLEL RATIONAL SEGMENTS (RAT_SEGMENT_SET)

\[ \text{list<seg_item> S.intersection_sorted(const rat_segment& q)} \]

as above, but the returned segments are ordered as they are intersected by \( q \) if one travels from \( q.source() \) to \( q.target() \).

\[ \text{list<seg_item> S.intersection(const rat_line& l)} \]

returns all items \(<s, i> \in S \) with \( s \cap l \neq \emptyset \).

**Precondition:** \( l \) is orthogonal to the segments in \( S \).

\[ \text{list<seg_item> S.intersection_sorted(const rat_line& l)} \]

as above, but the returned segments are ordered as they are intersected by \( l \) if one travels along \( l \) in direction \( l.direction() \).

\[ \text{void S.del(const rat_segment& s)} \]

deletes the item with segment \( s \) from \( S \).

\[ \text{void S.del_item(seg_item it)} \]

removes item \( it \) from \( S \).

**Precondition:** \( it \) is an item in \( S \).

\[ \text{void S.change_inf(seg_item it, const I& i)} \]

makes \( i \) the information of item \( it \).

**Precondition:** \( it \) is an item in \( S \).

\[ \text{void S.clear()} \]

makes \( S \) the empty \( \text{rat_segment_set} \).

\[ \text{bool S.empty()} \]

returns true iff \( S \) is empty.

\[ \text{int S.size()} \]

returns the size of \( S \).

4. Implementation

Segment sets are implemented by dynamic segment trees based on BB[\( \alpha \)] trees ([90, 57]) trees. Operations key, inf, change_inf, empty, and size take time \( O(1) \), insert, lookup, del, and del_item take time \( O(\log^2 n) \) and an intersection operation takes time \( O(k + \log^2 n) \), where \( k \) is the size of the returned list. Here \( n \) is the current size of the set. The space requirement is \( O(n \log n) \).
16.6 Planar Subdivisions (subdivision)

1. Definition

An instance $S$ of the parameterized data type subdivision<$I>$ is a subdivision of the two-dimensional plane, i.e., an embedded planar graph with straight line edges (see also sections 12.6 and 12.7). With each node $v$ of $S$ is associated a point, called the position of $v$ and with each face of $S$ is associated an information from data type $I$, called the information type of $S$.

```c
#include <LEDA/geo/subdivision.h>
```

2. Creation

```c
subdivision<$I>$ S(GRAPH<point, I>& G);
```

creates an instance $S$ of type subdivision<$I>$ and initializes it to the subdivision represented by the parameterized directed graph $G$. The node entries of $G$ (of type point) define the positions of the corresponding nodes of $S$. Every face $f$ of $S$ is assigned the information of one of its bounding edges in $G$.

**Precondition:** $G$ represents a planar subdivision, i.e., a straight line embedded planar map.

3. Operations

```c
point S.position(node v) returns the position of node $v$.
const I& S.inf(face f) returns the information of face $f$.
face S.locate_point(point p) returns the face containing point $p$.
face S.outer_face() returns the outer face of $S$.
```

4. Implementation

Planar subdivisions are implemented by parameterized planar maps and an additional data structure for point location based on partially persistent search trees[25]. Operations position and inf take constant time, a locate_point operation takes (expected) time $O(\log n)$. Here $n$ is the number of nodes. The space requirement is $O(n + m)$ and the initialization time is $O(n + m \log m)$, where $m$ is the number of edges in the map.
Chapter 17

Basic Data Types for Three-Dimensional Geometry
17.1 Points in 3D-Space (d3_point)

1. Definition

An instance of the data type d3_point is a point in the three-dimensional space $\mathbb{R}^3$. We use $(x, y, z)$ to denote a point with first (or x-) coordinate $x$, second (or y-) coordinate $y$, and third (or z-) coordinate $z$.

```cpp
#include <LEDA/geo/d3_point.h>
```

2. Creation

```cpp
d3_point p; // introduces a variable p of type d3_point initialized to the point (0, 0, 0).
d3_point p(double x, double y, double z); // introduces a variable p of type d3_point initialized to the point (x, y, z).
d3_point p(vector v); // introduces a variable p of type d3_point initialized to the point (v[0], v[1], v[2]).
Precondition: v.dim() = 3.
```

3. Operations

```cpp
double p.xcoord() // returns the first coordinate of p.
double p.ycoord() // returns the second coordinate of p.
double p.zcoord() // returns the third coordinate of p.
vector p.to_vector() // returns the vector $\vec{xyz}$.
point p.project_xy() // returns $p$ projected into the xy-plane.
point p.project_yz() // returns $p$ projected into the yz-plane.
point p.project_xz() // returns $p$ projected into the xz-plane.
double p.sqr_dist(const d3_point& q) // returns the square of the Euclidean distance between $p$ and $q$.
double p.xdist(const d3_point& q) // returns the x-distance between $p$ and $q$.
```
17.1. POINTS IN 3D-SPACE (D3_POINT)

double pydist(const d3_point & q)
returns the y-distance between p and q.

double pzdist(const d3_point & q)
returns the z-distance between p and q.

double p.distance(const d3_point & q)
returns the Euclidean distance between p and q.

d3_point p.translate(double dx, double dy, double dz)
returns p translated by vector (dx, dy, dz).

d3_point p.translate(const vector & v)
returns p + v, i.e., p translated by vector v.
Precondition: v.dim() = 3.

d3_point p + const vector & v returns p translated by vector v.

d3_point p - const vector & v returns p translated by vector -v.

d3_point p.reflect(const d3_point & q, const d3_point & r, const d3_point & s)
returns p reflected across the plane passing through q, r and s.

d3_point p.reflect(const d3_point & q)
returns p reflected across point q.

vector p - const d3_point & q
returns the difference vector of the coordinates.

ostream & ostream & O << const d3_point & p
writes p to output stream O.

istream & istream & I >> d3_point & p
reads the coordinates of p (three double numbers) from input stream I.

Non-Member Functions

int cmp_distances(const d3_point & p1, const d3_point & p2, const d3_point & p3,
const d3_point & p4)
compares the distances (p1, p2) and (p3, p4). Returns +1 (-1) if distance (p1, p2) is larger (smaller) than distance (p3, p4), otherwise 0.

d3_point center(const d3_point & a, const d3_point & b)
returns the center of a and b, i.e. a + a * b / 2.
584 CHAPTER 17. BASIC DATA TYPES FOR THREE-DIMENSIONAL GEOMETRY

d3_point midpoint(const d3_point &a, const d3_point &b) returns the center of a and b.

int orientation(const d3_point &a, const d3_point &b, const d3_point &c, const d3_point &d) computes the orientation of points a, b, c, and d as the sign of the determinant

\[
\begin{vmatrix}
1 & 1 & 1 & 1 \\
a_x & b_x & c_x & d_x \\
a_y & b_y & c_y & d_y \\
a_z & b_z & c_z & d_z
\end{vmatrix}
\]

int orientation_xy(const d3_point &a, const d3_point &b, const d3_point &c) returns the orientation of the projections of a, b and c into the xy-plane.

int orientation_yz(const d3_point &a, const d3_point &b, const d3_point &c) returns the orientation of the projections of a, b and c into the yz-plane.

int orientation_xz(const d3_point &a, const d3_point &b, const d3_point &c) returns the orientation of the projections of a, b and c into the xz-plane.

double volume(const d3_point &a, const d3_point &b, const d3_point &c, const d3_point &d) computes the signed volume of the simplex determined by a, b, c, and d, positive if orientation(a, b, c, d) > 0 and negative otherwise.

bool collinear(const d3_point &a, const d3_point &b, const d3_point &c) returns true if points a, b, c are collinear and false otherwise.

bool coplanar(const d3_point &a, const d3_point &b, const d3_point &c, const d3_point &d) returns true if points a, b, c, d are coplanar and false otherwise.

int side_of_sphere(const d3_point &a, const d3_point &b, const d3_point &c, const d3_point &d, const d3_point &x) returns +1 (-1) if point x lies on the positive (negative) side of the oriented sphere through points a, b, c, and d, and 0 if x is contained in this sphere.
17.1. POINTS IN 3D-SPACE (D3_POINT)

```cpp
int region_of_sphere(const d3_point& a, const d3_point& b, const d3_point& c,
                     const d3_point& d, const d3_point& x)
    determines whether the point x lies inside (= +1),
on (= 0), or outside (= −1) the sphere through
points a, b, c, d, (equivalent to orientation(a, b, c, d) *
side_of_sphere(a, b, c, d, x))
    Precondition: orientation(A) ≠ 0
```

```cpp
bool contained_in_simplex(const d3_point& a, const d3_point& b,
                          const d3_point& c, const d3_point& d,
                          const d3_point& x)
    determines whether x is contained in the simplex
spanned by the points a, b, c, d.
    Precondition: a, b, c, d are affinely independent.
```

```cpp
bool contained_in_simplex(const array<d3_point>& A, const d3_point& x)
    determines whether x is contained in the simplex
spanned by the points in A.
    Precondition: A must have size ≤ 4 and the points in A
must be affinely independent.
```

```cpp
bool contained_in_affine_hull(const list<d3_point>& L, const d3_point& x)
    determines whether x is contained in the affine hull of
the points in L.
```

```cpp
bool contained_in_affine_hull(const array<d3_point>& A, const d3_point& x)
    determines whether x is contained in the affine hull of
the points in A.
```

```cpp
int affine_rank(const array<d3_point>& L)
    computes the affine rank of the points in L.
```

```cpp
int affine_rank(const array<d3_point>& A)
    computes the affine rank of the points in A.
```

```cpp
bool affinely_independent(const list<d3_point>& L)
    decides whether the points in A are affinely independent.
```

```cpp
bool affinely_independent(const array<d3_point>& A)
    decides whether the points in A are affinely independent.
```

```cpp
bool inside_sphere(const d3_point& a, const d3_point& b, const d3_point& c,
                   const d3_point& d, const d3_point& e)
    returns true if point e lies in the interior of the sphere
through points a, b, c, and d, and false otherwise.
```
bool outside_sphere(const d3_point& a, const d3_point& b, const d3_point& c, const d3_point& d, const d3_point& e)
returns true if point e lies in the exterior of the sphere through points a, b, c, and d, and false otherwise.

bool on_sphere(const d3_point& a, const d3_point& b, const d3_point& c, const d3_point& d, const d3_point& e)
returns true if a, b, c, d, and e lie on a common sphere.

d3_point point_on_positive_side(const d3_point& a, const d3_point& b, const d3_point& c)
returns a point d with orientation(a, b, c, d) > 0.
17.2 Straight Rays in 3D-Space (d3_ray)

1. Definition

An instance \( r \) of the data type \( d3\_ray \) is a directed straight ray in three-dimensional space.

\[
\text{#include <LEDA/geo/d3\_ray.h>}
\]

2. Creation

\[
d3\_ray \ r(\text{const d3\_point}& \ p1, \text{const d3\_point}& \ p2);
\]

introduces a variable \( r \) of type \( d3\_ray \). \( r \) is initialized to the ray starting at point \( p1 \) and going through \( p2 \).

\[
d3\_ray \ r(\text{const d3\_segment}& \ s);
\]

introduces a variable \( r \) of type \( d3\_ray \). \( r \) is initialized to \( \text{ray}(s.\text{source}(\ ), \ s.\text{target}( \ )) \).

3. Operations

\[
d3\_point \ r.\text{source}(\ ) \quad \text{returns the source of } r.
\]

\[
d3\_point \ r.\text{point1}(\ ) \quad \text{returns the source of } r.
\]

\[
d3\_point \ r.\text{point2}(\ ) \quad \text{returns a point on } r \text{ different from the source.}
\]

\[
d3\_segment \ r.\text{seg}(\ ) \quad \text{returns a segment on } r.
\]

\[
\text{bool} \quad r.\text{contains(const d3\_point}& \ p)
\]

returns true if \( p \) lies on \( r \).

\[
\text{bool} \quad r.\text{contains(const d3\_segment}& \ s)
\]

returns true if \( s \) lies on \( r \).

\[
\text{bool} \quad r.\text{intersection(const d3\_segment}& \ s, \text{d3}\_\text{point}& \ \text{inter})
\]

if \( s \) and \( r \) intersect in a single point, true is returned and the point of intersection is assigned to \( \text{inter} \). Otherwise false is returned.

\[
\text{bool} \quad r.\text{intersection(const d3\_ray}& \ r, \text{d3}\_\text{point}& \ \text{inter})
\]

if \( r \) and \( r \) intersect in a single point, true is returned and the point of intersection is assigned to \( \text{inter} \). Otherwise false is returned.

\[
\text{bool} \quad r.\text{project}\_\text{xy(ray}& \ m)
\]

if the projection of \( r \) into the xy plane is not a point, the function returns true and assigns the projection to \( m \). Otherwise false is returned.
bool r.project_xz(ray & m) if the projection of r into the xz plane is not a point, the function returns true and assigns the projection to m. Otherwise false is returned.

bool r.project_yz(ray & m) if the projection of r into the yz plane is not a point, the function returns true and assigns the projection to m. Otherwise false is returned.

bool r.project(const d3_point & p, const d3_point & q, const d3_point & v, d3_ray & m) if the projection of r into the plane through (p, q, v) is not a point, the function returns true and assigns the projection to m. Otherwise false is returned.

d3_ray r.reverse() returns a ray starting at r.source() with direction -r.to_vector().

d3_ray r.translate(const vector & v) returns r translated by vector v. Precond.: v.dim() = 3.

d3_ray r.translate(double dx, double dy, double dz) returns r translated by vector (dx, dy, dz).

d3_ray r + const vector & v returns r translated by vector v.

d3_ray r - const vector & v returns r translated by vector -v.

d3_ray r.reflect(const d3_point & p, const d3_point & q, const d3_point & v) returns r reflected across the plane through (p, q, v).

d3_ray r.reflect(const d3_point & p) returns r reflected across p.

vector r.to_vector() returns point2() - point1().
17.3 Segments in 3D-Space (d3_segment)

1. Definition

An instance \( s \) of the data type \( d3\_segment \) is a directed straight line segment in three-dimensional space, i.e., a straight line segment \([p, q]\) connecting two points \( p, q \in \mathbb{R}^3 \). \( p \) is called the source or start point and \( q \) is called the target or end point of \( s \). The length of \( s \) is the Euclidean distance between \( p \) and \( q \). A segment is called trivial if its source is equal to its target. If \( s \) is not trivial, we use \( \text{line}(s) \) to denote the straight line containing \( s \).

```cpp
#include <LEDAGeo/d3_segment.h>
```

2. Creation

```cpp
d3_segment s(const d3_point& p1, const d3_point& p2);
```

introduces a variable \( s \) of type \( d3\_segment \). \( s \) is initialized to the segment from \( p1 \) to \( p2 \).

```cpp
d3_segment s;
```

introduces a variable \( s \) of type \( d3\_segment \). \( s \) is initialized to the segment from \((0, 0, 0)\) to \((1, 0, 0)\).

3. Operations

- \( bool s\_contains(const d3\_point& p) \) decides whether \( s \) contains \( p \).
- \( d3\_point s\_source() \) returns the source point of segment \( s \).
- \( d3\_point s\_target() \) returns the target point of segment \( s \).
- \( double s\_xcoord1() \) returns the x-coordinate of \( s\_source() \).
- \( double s\_xcoord2() \) returns the x-coordinate of \( s\_target() \).
- \( double s\_ycoord1() \) returns the y-coordinate of \( s\_source() \).
- \( double s\_ycoord2() \) returns the y-coordinate of \( s\_target() \).
- \( double s\_zcoord1() \) returns the z-coordinate of \( s\_source() \).
- \( double s\_zcoord2() \) returns the z-coordinate of \( s\_target() \).
- \( double s\_dx() \) returns \( xcoord2() - xcoord1() \).
- \( double s\_dy() \) returns \( ycoord2() - ycoord1() \).
- \( double s\_dz() \) returns \( zcoord2() - zcoord1() \).
segment s.project_xy() returns the projection into the xy plane.

segment s.project_xz() returns the projection into the xz plane.

segment s.project_yz() returns the projection into the yz plane.

d3_segment s.project(const d3_point& p, const d3_point& q, const d3_point& v)
returns s projected into the plane through (p, q, v).

d3_segment s.reflect(const d3_point& p, const d3_point& q, const d3_point& v)
returns s reflected across the plane through (p, q, v).

d3_segment s.reflect(const d3_point& p)
returns s reflected across point p.

d3_segment s.reverse() returns s reversed.

vector s.to_vector() returns s.target() – s.source().

bool s.intersection(const d3_segment& t)
decides, whether s and t intersect in a single point.

bool s.intersection(const d3_segment& t, d3_point& p)
decides, whether s and t intersect in a single point. If they intersect in a single point, the point is assigned to p and the result is true, otherwise the result is false.

bool s.intersection_of_lines(const d3_segment& t, d3_point& p)
If line(s) and line(t) intersect in a single point this point is assigned to p and the result is true, otherwise the result is false.

bool s.is_trivial() returns true if s is trivial.

double s.sqr_length() returns the square of the length of s.

double s.length() returns the length of s.

d3_segment s.translate(const vector& v)
returns s translated by vector v.
Precond.: v.dim() = 3.

d3_segment s.translate(double dx, double dy, double dz)
returns s translated by vector (dx, dy, dz).

d3_segment s + const vector& v returns s translated by vector v.

d3_segment s - const vector& v returns s translated by vector -v.
17.4 Straight Lines in 3D-Space (d3_line)

1. Definition

An instance \( l \) of the data type \( d3\_line \) is a directed straight line in three-dimensional space.

```cpp
#include <LEDA/geo/d3_line.h>
```

2. Creation

- \( d3\_line l(const d3\_point& p1, const d3\_point& p2); \)
  introduces a variable \( l \) of type \( d3\_line \). \( l \) is initialized to the line through points \( p1, p2 \).
  
  *Precondition: \( p1 \neq p2 \).*

- \( d3\_line l(const d3\_segment& s); \)
  introduces a variable \( l \) of type \( d3\_line \). \( l \) is initialized to the line supporting segment \( s \).
  
  *Precondition: \( s \) is not trivial.*

- \( d3\_line l; \)
  introduces a variable \( l \) of type \( d3\_line \). \( l \) is initialized to the line through points \( (0,0,0) \) and \( (1,0,0) \).

3. Operations

- \( bool l.\text{contains}(const\ d3\_point&\ p) \)
  returns true if \( p \) lies on \( l \).

- \( d3\_point l.\text{point1}() \)
  returns a point on \( l \).

- \( d3\_point l.\text{point2}() \)
  returns a second point on \( l \).

- \( d3\_segment l.\text{seg}() \)
  returns a non-trivial segment on \( l \) with the same direction.

- \( bool l.\text{project_xy(line&}\ m) \)
  if the projection of \( l \) into the xy plane is not a point, the function returns true and assigns the projection to \( m \). Otherwise false is returned.

- \( bool l.\text{project_xz(line&}\ m) \)
  if the projection of \( l \) into the xz plane is not a point, the function returns true and assigns the projection to \( m \). Otherwise false is returned.

- \( bool l.\text{project_yz(line&}\ m) \)
  if the projection of \( l \) into the yz plane is not a point, the function returns true and assigns the projection to \( m \). Otherwise false is returned.
bool \( l \).project(const d3\_point& \( p \), const d3\_point& \( q \), const d3\_point& \( v \),
\hspace*{1cm} \text{d3\_line& \( m \)})
\hspace*{1cm} \text{if the projection of \( l \) into the plane through \((p, q, v)\) is not a point, the function returns true and assigns the projection to \( m \). Otherwise false is returned.}

d3\_line \( l \).translate(double \( dx \), double \( dy \), double \( dz \))
\hspace*{1cm} \text{returns \( l \) translated by vector \((dx, dy, dz)\).}

d3\_line \( l \).translate(const vector& \( v \))
\hspace*{1cm} \text{returns \( l \) translated by \( v \).}
\hspace*{1cm} \text{Precond.: \( v\text{.dim()} = 3 \).}

d3\_line \( l \) + const vector& \( v \) \hspace*{1cm} \text{returns \( l \) translated by vector \( v \).}

d3\_line \( l \) − const vector& \( v \) \hspace*{1cm} \text{returns \( l \) translated by vector \(-v\).}

d3\_line \( l \).reflect(const d3\_point& \( p \), const d3\_point& \( q \), const d3\_point& \( v \))
\hspace*{1cm} \text{returns \( l \) reflected across the plane through \((p, q, v)\).}

d3\_line \( l \).reflect(const d3\_point& \( p \))
\hspace*{1cm} \text{returns \( l \) reflected across point \( p \).}

d3\_line \( l \).reverse()
\hspace*{1cm} \text{returns \( l \) reversed.}

vector \( l \).to\_vector()
\hspace*{1cm} \text{returns point2() − point1().}

bool \( l \).intersection(const d3\_segment& \( s \))
\hspace*{1cm} \text{decides, whether \( l \) and \( s \) intersect in a single point.}

bool \( l \).intersection(const d3\_segment& \( s \), d3\_point& \( p \))
\hspace*{1cm} \text{decides, whether \( l \) and \( s \) intersect in a single point. If so, the point of intersection is assigned to \( p \).}

bool \( l \).intersection(const d3\_line& \( m \))
\hspace*{1cm} \text{decides, whether \( l \) and \( m \) intersect.}

bool \( l \).intersection(const d3\_line& \( m \), d3\_point& \( p \))
\hspace*{1cm} \text{decides, whether \( l \) and \( m \) intersect in a single point. If so, the point of intersection is assigned to \( p \).}

double \( l \).sqr\_dist(const d3\_point& \( p \))
\hspace*{1cm} \text{returns the square of the distance between \( l \) and \( p \).}

double \( l \).distance(const d3\_point& \( p \))
\hspace*{1cm} \text{returns the distance between \( l \) and \( p \).}
17.5 Planes ( d3-plane )

1. Definition

An instance \( P \) of the data type \( d3\_plane \) is an oriented plane in the three-dimensional space \( \mathbb{R}^3 \). It can be defined by a triple \((a,b,c)\) of non-collinear points or a single point \( a \) and a normal vector \( v \).

\[
\text{#include } < \text{LEDA/geo/d3\_plane.h}> 
\]

2. Creation

\[
d3\_plane \ p; \quad \text{introduces a variable } p \text{ of type } d3\_plane \text{ initialized to the xy-plane.}
\]

\[
d3\_plane \ p(\text{const } d3\_point & \ a, \text{const } d3\_point & \ b, \text{const } d3\_point & \ c); \quad \text{introduces a variable } p \text{ of type } d3\_plane \text{ initialized to the plane through } (a, b, c). \quad \text{Precondition: } a, b, \text{ and } c \text{ are not collinear.}
\]

\[
d3\_plane \ p(\text{const } d3\_point & \ a, \text{const } \text{vector} & \ v); \quad \text{introduces a variable } p \text{ of type } d3\_plane \text{ initialized to the plane that contains } a \text{ with normal vector } v. \quad \text{Precondition: } v.\text{dim}( ) = 3 \text{ and } v.\text{length}( ) > 0.
\]

\[
d3\_plane \ p(\text{const } d3\_point & \ a, \text{const } d3\_point & \ b); \quad \text{introduces a variable } p \text{ of type } d3\_plane \text{ initialized to the plane that contains } a \text{ with normal vector } b - a.
\]

3. Operations

\[
d3\_point \ p.\text{point1}( ) \quad \text{returns the first point of } p.
\]

\[
d3\_point \ p.\text{point2}( ) \quad \text{returns the second point of } p.
\]

\[
d3\_point \ p.\text{point3}( ) \quad \text{returns the third point of } p.
\]

\[
double \ p.A( ) \quad \text{returns the } A \text{ parameter of the plane equation.}
\]

\[
double \ p.B( ) \quad \text{returns the } B \text{ parameter of the plane equation.}
\]

\[
double \ p.C( ) \quad \text{returns the } C \text{ parameter of the plane equation.}
\]

\[
double \ p.D( ) \quad \text{returns the } D \text{ parameter of the plane equation.}
\]

\[
\text{vector } p.\text{normal}( ) \quad \text{returns a normal vector of } p.
\]
double p.sqr_dist(const d3_point& q)
returns the square of the Euclidean distance between p and q.

double p.distance(const d3_point& q)
returns the Euclidean distance between p and q.

int p.cmp_distances(const d3_point& p1, const d3_point& p2)
compares the distances of p1 and p2 to p and returns the result.

vector p.normal_project(const d3_point& q)
returns the vector pointing from q to its projection on p along the normal direction.

int p.intersection(const d3_point p1, const d3_point p2, d3_point& q)
if the line l through p1 and p2 intersects p in a single point this point is assigned to q and the result is 1, if l and p do not intersect the result is 0, and if l is contained in p the result is 2.

int p.intersection(const d3_plane& Q, d3_point& i1, d3_point& i2)
if p and plane Q intersect in a line L then (i1, i2) are assigned two different points on L and the result is 1, if p and Q do not intersect the result is 0, and if p = Q the result is 2.

d3_plane p.translate(double dx, double dy, double dz)
returns p translated by vector (dx, dy, dz).

d3_plane p.translate(const vector& v)
returns p+v, i.e., p translated by vector v.
Precondition: v.dim() == 3.

d3_plane p + const vector& v returns p translated by vector v.

d3_plane p.reflect(const d3_plane& Q)
returns p reflected across plane Q.

d3_plane p.reflect(const d3_point& q)
returns p reflected across point q.

d3_point p.reflect_point(const d3_point& q)
returns q reflected across plane p.

int p.side_of(const d3_point& q)
computes the side of p on which q lies.
17.5. PLANES (D3_PLANE)

`bool p.contains(const d3_point& q)`

returns true if point `q` lies on plane `p`, i.e., `p.side_of(q) == 0`, and false otherwise.

`bool p.parallel(const d3_plane& Q)`

returns true if planes `p` and `Q` are parallel and false otherwise.

`ostream& ostream& O ≪ const d3_plane& p`

writes `p` to output stream `O`.

`istream& istream& I >> d3_plane& p`

reads the coordinates of `p` (six `double` numbers) from input stream `I`.

**Non-Member Functions**

`int orientation(const d3_plane& p, const d3_point& q)`

computes the orientation of `p.side_of(q)`.
17.6 Spheres in 3D-Space (d3_sphere)

1. Definition

An instance of the data type d3_sphere is a directed sphere in 3d space. The sphere is defined by four points \(p_1, p_2, p_3, p_4\) (d3_points).

```cpp
#include <LEDA/geo/d3_sphere.h>
```

2. Creation

```cpp
d3_sphere S(const d3_point& p1, const d3_point& p2, const d3_point& p3,
const d3_point& p4);
```

introduces a variable \(S\) of type d3_sphere. \(S\) is initialized to the sphere through points \(p_1, p_2, p_3, p_4\).

3. Operations

- `bool S.contains(const d3_point& p)` returns true, if \(p\) is on the sphere, false otherwise.
- `bool S.inside(const d3_point& p)` returns true, if \(p\) is inside the sphere, false otherwise.
- `bool S.outside(const d3_point& p)` returns true, if \(p\) is outside the sphere, false otherwise.
- `d3_point S.point1()` returns \(p_1\).
- `d3_point S.point2()` returns \(p_2\).
- `d3_point S.point3()` returns \(p_3\).
- `d3_point S.point4()` returns \(p_4\).
- `bool S.is_degenerate()` returns true, if the 4 defining points are coplanar.
- `d3_point S.center()` returns the center of the sphere.
- `double S.sqr_radius()` returns the square of the radius.
- `double S.radius()` returns the radius.
- `double S.surface()` returns the size of the surface.
- `double S.volume()` returns the volume of the sphere.
- `d3_sphere S.translate(const vector& v)` returns \(S\) translated by vector \(v\).
\texttt{\texttt{d3\_sphere}} \quad S.\texttt{translate(}\texttt{double} \, \texttt{dx, double} \, \texttt{dy, double} \, \texttt{dz)}
\texttt{
returns \texttt{S} translated by vector \texttt{(dx, dy, dz)}.}
17.7 Simplices in 3D-Space (d3_simplex)

1. Definition

An instance of the data type d3_simplex is a simplex in 3d space. The simplex is defined by four points \( p_1, p_2, p_3, p_4 \) (d3_points). We call the simplex degenerate, if the four defining points are coplanar.

```c
#include <LEDA/geo/d3_simplex.h>
```

2. Types

- \( d3\text{\_simplex}\text{\_\_coord\_type} \) the coordinate type (double).
- \( d3\text{\_simplex}\text{\_\_point\_type} \) the point type (d3_point).

3. Creation

```c
d3\_simplex S(const d3\_point\& a, const d3\_point\& b, const d3\_point\& c, const d3\_point\& d);
```

creates the simplex \((a, b, c, d)\).

```c
d3\_simplex S;
```

creates the simplex \(((0, 0, 0), (1, 0, 0), (0, 1, 0), (0, 0, 1))\).

4. Operations

```c
d3\_point S.point1() returns \( p_1 \).
d3\_point S.point2() returns \( p_2 \).
d3\_point S.point3() returns \( p_3 \).
d3\_point S.point4() returns \( p_4 \).
d3\_point S[int i] returns \( p_i \). Precondition: \( i > 0 \) and \( i < 5 \).
int S.index(const d3\_point\& p) returns 1 if \( p == p_1 \), 2 if \( p == p_2 \), 3 if \( p == p_3 \), 4 if \( p == p_4 \), and 0 otherwise.
bool S.is_degenerate() returns true if \( S \) is degenerate and false otherwise.
d3\_sphere S.circumscribing_sphere() returns a d3\_sphere through \((p_1, p_2, p_3, p_4)\) (precondition: the d3_simplex is not degenerate).
bool S.in_simplex(const d3\_point\& p) returns true, if \( p \) is contained in the simplex.
```
bool S.insphere(const d3_point& p) returns true, if p lies in the interior of the sphere through \( p_1, p_2, p_3, p_4 \).

double S.vol() returns the signed volume of the simplex.

d3_simplex S.reflect(const d3_point& p, const d3_point& q, const d3_point& v) returns S reflected across the plane through \( (p, q, v) \).

d3_simplex S.reflect(const d3_point& p) returns S reflected across point p.

d3_simplex S.translate(const vector& v) returns S translated by vector v.  
Precond.: \( v \cdot \text{dim}( ) = 3. \)

d3_simplex S.translate(double dx, double dy, double dz) returns S translated by vector \( (dx, dy, dz) \).

d3_simplex S + const vector& v returns S translated by vector v.

d3_simplex S − const vector& v returns S translated by vector \( -v \).
17.8 Rational Points in 3D-Space (d3_rat_point)

1. Definition

An instance of data type d3_rat_point is a point with rational coordinates in the three-dimensional space. A point with cartesian coordinates \((a, b, c)\) is represented by homogeneous coordinates \((x, y, z, w)\) of arbitrary length integers (see 6.1) such that \(a = x/w, b = y/w, c = z/w\) and \(w > 0\).

```cpp
#include <LEDA/geo/d3_rat_point.h>
```

2. Creation

- `d3_rat_point p;` introduces a variable \(p\) of type \(d3\_rat\_point\) initialized to the point \((0, 0, 0)\).
- `d3_rat_point p(const rational& a, const rational& b, const rational& c);` introduces a variable \(p\) of type \(d3\_rat\_point\) initialized to the point \((a, b, c)\).
- `d3_rat_point p(integer a, integer b, integer c);` introduces a variable \(p\) of type \(d3\_rat\_point\) initialized to the point \((a, b, c)\).
- `d3_rat_point p(integer x, integer y, integer z, integer w);` introduces a variable \(p\) of type \(d3\_rat\_point\) initialized to the point with homogeneous coordinates \((x, y, z, w)\) if \(w > 0\) and to point \((-x, -y, -z, -w)\) if \(w < 0\).
  - **Precondition**: \(w \neq 0\).
- `d3_rat_point p(const rat_vector& v);` introduces a variable \(p\) of type \(d3\_rat\_point\) initialized to the point \((v[0], v[1], v[2])\).
  - **Precondition**: \(v.\text{dim}() = 3\).

3. Operations

- `d3_point p.to_float();` returns a floating point approximation of \(p\).
- `rat_vector p.to_vector();` returns the vector extending from the origin to \(p\).
- `integer p.X();` returns the first homogeneous coordinate of \(p\).
- `integer p.Y();` returns the second homogeneous coordinate of \(p\).
- `integer p.Z();` returns the third homogeneous coordinate of \(p\).
integer \hspace{1em} p.W() \hspace{1em} \text{returns the fourth homogeneous coordinate of } p.

double \hspace{1em} p.XD() \hspace{1em} \text{returns a floating point approximation of } p.X().

double \hspace{1em} p.YD() \hspace{1em} \text{returns a floating point approximation of } p.Y().

double \hspace{1em} p.ZD() \hspace{1em} \text{returns a floating point approximation of } p.Z().

double \hspace{1em} p.WD() \hspace{1em} \text{returns a floating point approximation of } p.W().

rational \hspace{1em} p.xcoord() \hspace{1em} \text{returns the } x\text{-coordinate of } p.

rational \hspace{1em} p.ycoord() \hspace{1em} \text{returns the } y\text{-coordinate of } p.

rational \hspace{1em} p.zcoord() \hspace{1em} \text{returns the } z\text{-coordinate of } p.

rational \hspace{1em} p[int i] \hspace{1em} \text{returns the } i\text{th cartesian coordinate of } p
\text{\hspace{1em} \textbf{Precondition: } } 0 \leq i \leq 2.

double \hspace{1em} p.xcoordD() \hspace{1em} \text{returns a floating point approximation of } p.xcoord().

double \hspace{1em} p.ycoordD() \hspace{1em} \text{returns a floating point approximation of } p.ycoord().

double \hspace{1em} p.zcoordD() \hspace{1em} \text{returns a floating point approximation of } p.zcoord().

integer \hspace{1em} p.hcoord(int i) \hspace{1em} \text{returns the } i\text{th homogeneous coordinate of } p
\text{\hspace{1em} \textbf{Precondition: } } 0 \leq i \leq 3.

rat_point \hspace{1em} p.project_xy() \hspace{1em} \text{returns } p \text{ projected into the xy-plane.}

rat_point \hspace{1em} p.project_yz() \hspace{1em} \text{returns } p \text{ projected into the yz-plane.}

rat_point \hspace{1em} p.project_xz() \hspace{1em} \text{returns } p \text{ projected into the xz-plane.}

d3_rat_point \hspace{1em} p.reflect(const d3_rat_point& p, const d3_rat_point& q, const d3_rat_point& r)
\hspace{1em} \text{returns } p \text{ reflected across the plane passing through } p, q \text{ and } r.
\text{\hspace{1em} \textbf{Precondition: } } p, q \text{ and } r \text{ are not collinear.}

d3_rat_point \hspace{1em} p.reflect(const d3_rat_point& q)
\hspace{1em} \text{returns } p \text{ reflected across point } q.

d3_rat_point \hspace{1em} p.translate(const rational& dx, const rational& dy, const rational& dz)
\hspace{1em} \text{returns } p \text{ translated by vector } (dx,dy,dz).

d3_rat_point \hspace{1em} p.translate(integer dx, integer dy, integer dz, integer dw)
\hspace{1em} \text{returns } p \text{ translated by vector } (dx/dw,dy/dw,dz/dw).
CHAPTER 17. BASIC DATA TYPES FOR THREE-DIMENSIONAL GEOMETRY

\[ d3\text{\_rat\_point} \quad \text{p.translate}(\text{const} \ d3\text{\_rat\_vector} & \ v) \]

returns \( p + v \), i.e., \( p \) translated by vector \( v \)

\text{Precondition: } v\text{.dim()} = 3.

\[ d3\text{\_rat\_point} \quad \text{p + const} \ d3\text{\_rat\_vector} & \ v \]

returns \( p \) translated by vector \( v \)

\text{Precondition: } v\text{.dim()} = 3.

\[ d3\text{\_rat\_point} \quad \text{p - const} \ d3\text{\_rat\_vector} & \ v \]

returns \( p \) translated by vector \( -v \)

\text{Precondition: } v\text{.dim()} = 3.

\[ \text{rational} \quad \text{p.sqr\_dist(const} \ d3\text{\_rat\_point} & \ q) \]

returns the squared distance between \( p \) and \( q \).

\[ \text{rational} \quad \text{p.xdist(const} \ d3\text{\_rat\_point} & \ q) \]

returns the x-distance between \( p \) and \( q \).

\[ \text{rational} \quad \text{p.ydist(const} \ d3\text{\_rat\_point} & \ q) \]

returns the y-distance between \( p \) and \( q \).

\[ \text{rational} \quad \text{p.zdist(const} \ d3\text{\_rat\_point} & \ q) \]

returns the z-distance between \( p \) and \( q \).

\[ \text{rat\_vector} \quad \text{p - const} \ d3\text{\_rat\_point} & \ q \]

returns the difference vector of the coordinates.

\[ \text{ostream\&} \quad \text{ostream\&} \ O \ll \text{const} \ d3\text{\_rat\_point} & \ p \]

writes the homogeneous coordinates \((x, y, z, w)\) of \( p \) to output stream \( O \).

\[ \text{istream\&} \quad \text{istream\&} \ I \gg \text{d3\text{\_rat\_point} &} \ p \]

reads the homogeneous coordinates \((x, y, z, w)\) of \( p \) from input stream \( I \).

\textbf{Non-Member Functions}
17.8. RATIONAL POINTS IN 3D-SPACE (D3_RAT_POINT)

```c
int orientation(const d3_rat_point& a, const d3_rat_point& b, const d3_rat_point& c, const d3_rat_point& d)
    computes the orientation of points a, b, c and d as the sign of the determinant
    \[
    \begin{vmatrix}
    a_w & b_w & c_w & d_w \\
    a_x & b_x & c_x & d_x \\
    a_y & b_y & c_y & d_y \\
    a_z & b_z & c_z & d_z
    \end{vmatrix}
    \]
    i.e., it returns +1 if point d lies left of the directed plane through a, b, c, 0 if a, b, c and d are coplanar, and
    −1 otherwise.

int orientation_xy(const d3_rat_point& a, const d3_rat_point& b, const d3_rat_point& c)
    returns the orientation of the projections of a, b and c into the xy-plane.

int orientation_yz(const d3_rat_point& a, const d3_rat_point& b, const d3_rat_point& c)
    returns the orientation of the projections of a, b and c into the yz-plane.

int orientation_xz(const d3_rat_point& a, const d3_rat_point& b, const d3_rat_point& c)
    returns the orientation of the projections of a, b and c into the xz-plane.

int cmp_distances(const d3_rat_point& p1, const d3_rat_point& p2, const d3_rat_point& p3, const d3_rat_point& p4)
    compares the distances (p1, p2) and (p3, p4). Returns +1 (−1) if distance (p1, p2) is larger (smaller) than
    distance (p3, p4), otherwise 0.

d3_rat_point midpoint(const d3_rat_point& a, const d3_rat_point& b)
    returns the midpoint of a and b.

rational volume(const d3_rat_point& a, const d3_rat_point& b, const d3_rat_point& c, const d3_rat_point& d)
    computes the signed volume of the simplex determined by a, b, c, and d, positive if orientation(a, b, c, d) > 0
    and negative otherwise.

bool collinear(const d3_rat_point& a, const d3_rat_point& b, const d3_rat_point& c)
    returns true if points a, b, c are collinear, and false otherwise.
```
bool coplanar(const d3_rat_point& a, const d3_rat_point& b, 
const d3_rat_point& c, const d3_rat_point& d)
returns true if points a, b, c, d are coplanar and false otherwise.

int side_of_sphere(const d3_rat_point& a, const d3_rat_point& b, 
const d3_rat_point& c, const d3_rat_point& d, 
const d3_rat_point& e)
returns +1 (−1) if point e lies on the positive (negative)
side of the oriented sphere through points a, b, c, and 
d, and 0 if e is contained in this sphere.

int region_of_sphere(const d3_rat_point& a, const d3_rat_point& b, 
const d3_rat_point& c, const d3_rat_point& d, 
const d3_rat_point& x)
determines whether the point x lies inside (= +1),
on (= 0), or outside (= −1) the sphere through
points a, b, c, d, (equivalent to orientation(a, b, c, d) * 
side_of_sphere(a, b, c, d, x))
Precondition: orientation(A) ≠ 0

bool contained_in_simplex(const d3_rat_point& a, const d3_rat_point& b, 
const d3_rat_point& c, const d3_rat_point& d, 
const d3_rat_point& x)
determines whether x is contained in the simplex
spanned by the points a, b, c, d.
Precondition: a, b, c, d are affinely independent.

bool contained_in_simplex(const array<d3_rat_point>& A, const d3_rat_point& x)
determines whether x is contained in the simplex
spanned by the points in A.
Precondition: A must have size ≤ 4 and the points in 
A must be affinely independent.

bool contained_in_affine_hull(const list<d3_rat_point>& L, const d3_rat_point& x)
determines whether x is contained in the affine hull of
the points in L.

bool contained_in_affine_hull(const array<d3_rat_point>& A, 
const d3_rat_point& x)
determines whether x is contained in the affine hull of
the points in A.

int affine_rank(const array<d3_rat_point>& L)
computes the affine rank of the points in L.

int affine_rank(const array<d3_rat_point>& A)
computes the affine rank of the points in A.
bool affinely_independent(const list<d3_rat_point>& L) 

decides whether the points in A are affinely independent.

bool affinely_independent(const array<d3_rat_point>& A) 

decides whether the points in A are affinely independent.

bool inside_sphere(const d3_rat_point& a, const d3_rat_point& b, 
const d3_rat_point& c, const d3_rat_point& d, 
const d3_rat_point& e) 

returns true if point e lies in the interior of the sphere 
through points a, b, c, and d, and false otherwise.

bool outside_sphere(const d3_rat_point& a, const d3_rat_point& b, 
const d3_rat_point& c, const d3_rat_point& d, 
const d3_rat_point& e) 

returns true if point e lies in the exterior of the sphere 
through points a, b, c, and d, and false otherwise.

bool on_sphere(const d3_rat_point& a, const d3_rat_point& b, 
const d3_rat_point& c, const d3_rat_point& d, 
const d3_rat_point& e) 

returns true if points a, b, c, d, and e lie on a common 
sphere.

d3_rat_point point_on_positive_side(const d3_rat_point& a, const d3_rat_point& b, 
const d3_rat_point& c) 

returns a point d with orientation(a, b, c, d) > 0.

Point Generators

d3_rat_point random_d3_rat_point_in_cube(int maxc) 

returns a point whose coordinates are random integers 
in [−maxc..maxc].

void random_d3_rat_points_in_cube(int n, int maxc, list<d3_rat_point>& L) 

returns a list L of n points . . . .

d3_rat_point random_d3_rat_point_in_square(int maxc) 

returns a point whose x and y-coordinates are ran-
dom integers in [−maxc..maxc]. The z-coordinate 
is zero. In 2d, this function is equivalent to 
random_rat_point_in_cube.

void random_d3_rat_points_in_square(int n, int maxc, list<d3_rat_point>& L) 

returns a list L of n points . . . .
**d3_rat_point** random_d3_rat_point_in_unit_cube(int $D = 16383$)

returns a point whose coordinates are random rationals of the form $i/D$ where $i$ is a random integer in the range $[0..D]$. The default value of $D$ is $2^{14} - 1$.

**void** random_d3_rat_points_in_unit_cube(int $n$, int $D$, list<d3_rat_point>& $L$)

returns a list $L$ of $n$ points ... .

**void** random_d3_rat_points_in_unit_cube(int $n$, list<d3_rat_point>& $L$)

as above, but the default value of $D$ is used.

**d3_rat_point** random_d3_rat_point_in_ball(int $R$)

returns a random point with integer coordinates in the ball with radius $R$ centered at the origin.

*Precondition:* $R \leq 2^{14}$.

**void** random_d3_rat_points_in_ball(int $n$, int $R$, list<d3_rat_point>& $L$)

returns a list $L$ of $n$ points ... .

**void** random_d3_rat_points_in_ball(int $n$, list<d3_rat_point>& $L$)

returns a list $L$ of $n$ points ... . The default value of $D$ is used.

**d3_rat_point** random_d3_rat_point_in_disc(int $R$)

returns a random point with integer $x$ and $y$-coordinates in the disc with radius $R$ centered at the origin. The $z$-coordinate is zero. In 2d this is the same as the function random_rat_point_in_ball.

*Precondition:* $R \leq 2^{14}$.

**void** random_d3_rat_points_in_disc(int $n$, int $R$, list<d3_rat_point>& $L$)

returns a list $L$ of $n$ points ... .

**void** random_d3_rat_points_in_disc(int $n$, list<d3_rat_point>& $L$)

returns a list $L$ of $n$ points ... .

**d3_rat_point** random_d3_rat_point_on_circle(int $R$)

returns a random point with integer coordinates that lies close to the circle with radius $R$ centered at the origin.

**void** random_d3_rat_points_on_circle(int $m$, int $R$, list<d3_rat_point>& $L$)

returns a list $L$ of $n$ points ... .
**17.8. RATIONAL POINTS IN 3D-SPACE (D3_RAT_POINT)**

**d3_rat_point** random_d3_rat_point_on_unit_circle(int \(D = 16383\))

returns a point close to the unit circle whose coordinates are random rationals of the form \(i/D\) where \(i\) is a random integer in the range \([0..D]\). The default value of \(D\) is \(2^{14} - 1\).

**void** random_d3_rat_points_on_unit_circle(int \(m, D\), list<d3_rat_point>& L)

returns a list \(L\) of \(n\) points . . . .

**void** random_d3_rat_points_on_unit_circle(int \(m\), list<d3_rat_point>& L)

returns a list \(L\) of \(n\) points . . . . The default value of \(D\) is used.

**d3_rat_point** random_d3_rat_point_on_sphere(int \(R\))

returns a point with integer coordinates close to the sphere with radius \(R\) centered at the origin.

**void** random_d3_rat_points_on_sphere(int \(m, R\), list<d3_rat_point>& L)

returns a list \(L\) of \(n\) points . . . .

**d3_rat_point** random_d3_rat_point_on_unit_sphere(int \(D = 16383\))

returns a point close to the unit sphere whose coordinates are random rationals of the form \(i/D\) where \(i\) is a random integer in the range \([0..D]\). The default value of \(D\) is \(2^{14} - 1\). In 2d this function is equivalent to point_on_unit_circle.

**void** random_d3_rat_points_on_unit_sphere(int \(m, D\), list<d3_rat_point>& L)

returns a list \(L\) of \(n\) points . . . .

**void** random_d3_rat_points_on_unit_sphere(int \(m\), list<d3_rat_point>& L)

returns a list \(L\) of \(n\) points . . . . The default value of \(D\) is used.

**d3_rat_point** random_d3_rat_point_on_paraboloid(int maxc)

returns a point \((x, y, z)\) with \(x\) and \(y\) random integers in the range \([-\text{maxc}..\text{maxc}]\), and \(z = 0.004 \ast (x \ast x + y \ast y) - 1.25 \ast \text{maxc}\). The function does not make sense in 2d.

**void** random_d3_rat_points_on_paraboloid(int \(n, maxc\), list<d3_rat_point>& L)

returns a list \(L\) of \(n\) points . . . .

**void** lattice_d3_rat_points(int \(n, maxc\), list<d3_rat_point>& L)

returns a list \(L\) of approximately \(n\) points. The points have integer coordinates \(id/maxc\) for an appropriately chosen \(d\) and \(-\text{maxc}/d \leq i \leq \text{maxc}/d\).
void random_d3_rat_points_on_segment(int n, int maxc, list<d3_rat_point>& L)
generates n points on the diagonal whose coordinates are random integer in the range from \(-maxc\) to \(maxc\).
17.9 Straight Rational Rays in 3D-Space (d3_rat_ray)

1. Definition

An instance \( r \) of the data type \( \text{d3\_rat\_ray} \) is a directed straight ray defined by two points with rational coordinates in three-dimensional space.

\[
\texttt{
#include <LEDA/geo/d3\_rat\_ray.h>
}
\]

2. Creation

\[
\text{d3\_rat\_ray} \ r(\text{const d3\_rat\_point} \& p1, \text{const d3\_rat\_point} \& p2);
\]

introduces a variable \( r \) of type \( \text{d3\_rat\_ray} \). \( r \) is initialized to the ray starting at point \( p1 \) and going through \( p2 \).

\[
\text{d3\_rat\_ray} \ r(\text{const d3\_rat\_segment} \& s);
\]

introduces a variable \( r \) of type \( \text{d3\_rat\_ray} \). \( r \) is initialized to \( \text{ray}(s.\text{source}(), s.\text{target}()) \).

3. Operations

\[
\text{d3\_rat\_point} \ r.\text{source}() \quad \text{return the source of } r.
\]

\[
\text{d3\_rat\_point} \ r.\text{point1}() \quad \text{return the source of } r.
\]

\[
\text{d3\_rat\_point} \ r.\text{point2}() \quad \text{return a point on } r \text{ different from the source.}
\]

\[
\text{d3\_rat\_segment} \ r.\text{seg}() \quad \text{return a segment on } r.
\]

\[
\text{bool} \quad r.\text{contains}(\text{const d3\_rat\_point} \& p)
\]

returns true if \( p \) lies on \( r \).

\[
\text{bool} \quad r.\text{contains}(\text{const d3\_rat\_segment} \& s)
\]

returns true if \( s \) lies on \( r \).

\[
\text{bool} \quad r.\text{intersection}(\text{const d3\_rat\_segment} \& s, \text{d3\_rat\_point} \& inter)
\]

if \( s \) and \( r \) intersect in a single point, true is returned and the point of intersection is assigned to \( \text{inter} \). Otherwise false is returned.

\[
\text{bool} \quad r.\text{intersection}(\text{const d3\_rat\_ray} \& r, \text{d3\_rat\_point} \& inter)
\]

if \( r \) and \( r \) intersect in a single point, true is returned and the point of intersection is assigned to \( \text{inter} \). Otherwise false is returned.
bool r.project_xy(rat_ray& m)  
if the projection of r into the xy plane is not a point,  
the function returns true and assigns the projection  
to m. Otherwise false is returned.

bool r.project_xz(rat_ray& m)  
if the projection of r into the xz plane is not a point,  
the function returns true and assigns the projection  
to m. Otherwise false is returned.

bool r.project_yz(rat_ray& m)  
if the projection of r into the yz plane is not a point,  
the function returns true and assigns the projection  
to m. Otherwise false is returned.

bool r.project(const d3_rat_point& p, const d3_rat_point& q,  
const d3_rat_point& v, d3_rat_ray& m)  
if the projection of r into the plane through (p, q, v)  
is not a point, the function returns true and assigns  
the projection to m. Otherwise false is returned.

d3_rat_ray r.reverse()  
returns a rat_ray starting at r.source() with direction  
-r.to_vector().

d3_rat_ray r.translate(const rat_vector& v)  
returns r translated by vector v. Precond. : v.dim( )  
= 3.

d3_rat_ray r.translate(rational dx, rational dy, rational dz)  
returns r translated by vector (dx, dy, dz).

d3_rat_ray r + const rat_vector& v  
returns r translated by vector v.

d3_rat_ray r - const rat_vector& v  
returns r translated by vector -v.

d3_rat_ray r.reflect(const d3_rat_point& p, const d3_rat_point& q,  
const d3_rat_point& v)  
returns r reflected across the plane through (p, q, v).

d3_rat_ray r.reflect(const d3_rat_point& p)  
returns r reflected across point p.

rat_vector r.to_vector()  
returns point2( ) - point1( ).
17.10 Rational Lines in 3D-Space (d3_rat_line)

1. Definition

An instance $l$ of the data type $d3\text{-}rat\text{-}line$ is a directed straight line in three-dimensional space.

```
#include <LEDA/geo/d3_rat_line.h>
```

2. Creation

```
d3\_rat\_line l(const d3\_rat\_point& p1, const d3\_rat\_point& p2);
```

introduces a variable $l$ of type $d3\_rat\_line$. $l$ is initialized to the line through points $p1,p2$.

```
d3\_rat\_line l(const d3\_rat\_segment& s);
```

introduces a variable $l$ of type $d3\_rat\_line$. $l$ is initialized to the line supporting segment $s$.

```
d3\_rat\_line l;
```

introduces a variable $l$ of type $d3\_rat\_line$. $l$ is initialized to the line through points $(0,0,0,1)$ and $(1,0,0,1)$.

3. Operations

```
d3\_line l.to\_float()
```

returns a floating point approximation of $l$.

```
bool l.contains(const d3\_rat\_point& p)
```

returns true if $p$ lies on $l$.

```
d3\_rat\_point l.point1()
```

returns a point on $l$.

```
d3\_rat\_point l.point2()
```

returns a second point on $l$.

```
d3\_rat\_segment l.seg()
```

returns a segment on $l$.

```
bool l.project\_xy(rat\_line& m)
```

if the projection of $l$ into the xy plane is not a point, the function returns true and assigns the projection to $m$. Otherwise false is returned.

```
bool l.project\_xz(rat\_line& m)
```

if the projection of $l$ into the xz plane is not a point, the function returns true and assigns the projection to $m$. Otherwise false is returned.
bool l.project_yz(rat_line& m)
if the projection of l into the yz plane is not a point, the function returns true and assigns the projection to m. Otherwise false is returned.

bool l.project(const d3_rat_point& p, const d3_rat_point& q, const d3_rat_point& v, d3_rat_line& m)
if the projection of l into the plane through (p,q,v) is not a point, the function returns true and assigns the projection to m. Otherwise false is returned.

d3_rat_line l.translate(integer dx, integer dy, integer dz, integer dw)
returns l translated by vector (dx/dw, dy/dw, dz/dw).

d3_rat_line l.translate(rat_vector v)
returns l translated by v.
Precond. : v.dim( ) = 3.

d3_rat_line l + const rat_vector& v
returns l translated by vector v.

d3_rat_line l - const rat_vector& v
returns l translated by vector -v.

d3_rat_line l.reflect(const d3_rat_point& p, const d3_rat_point& q, const d3_rat_point& v)
returns l reflected across the plane through (p,q,v).

d3_rat_line l.reflect(const d3_rat_point& p)
returns l reflected across point p.

d3_rat_line l.reverse( ) returns l reversed.

rat_vector l.to_vector( ) returns point2( ) - point1( ).

bool l.intersection(const d3_rat_segment& s)
decides, whether l and s intersect in a single point.

bool l.intersection(const d3_rat_segment& s, d3_rat_point& p)
decides, whether l and s intersect in a single point. If so, the point of intersection is assigned to p.

bool l.intersection(const d3_rat_line& m)
decides, whether l and m intersect in a single point.
\textit{bool} \quad \textit{L.intersection(} 
\textit{const d3\_rat\_line& m, d3\_rat\_point& p) } \textit{decides, whether \textit{l} and \textit{m} intersect in a single point. If so, the point of intersection is assigned to \textit{p}.}

\textit{rational} \quad \textit{L.sqr\_dist(} \textit{const d3\_rat\_point& p) } \textit{returns the square of the distance between \textit{l} and \textit{p}.}
17.11 Rational Segments in 3D-Space (d3_rat_segment)

1. Definition

An instance $s$ of the data type $d3\_rat\_segment$ is a directed straight line segment in three-dimensional space, i.e., a line segment connecting two rational points $p, q \in \mathbb{R}^3$. $p$ is called the source or start point and $q$ is called the target or end point of $s$. A segment is called trivial if its source is equal to its target. If $s$ is not trivial, we use $\text{line}(s)$ to denote the straight line containing $s$.

```c
#include <LEDA/geo/d3_rat_segment.h>
```

2. Creation

```c
d3\_rat\_segment s(const d3\_rat\_point\& p1, const d3\_rat\_point\& p2);
```
introduces a variable $S$ of type $d3\_rat\_segment$. $S$ is initialized to the segment through points $p1, p2$.

```c
d3\_rat\_segment s;
```
introduces a variable $S$ of type $d3\_rat\_segment$. $S$ is initialized to the segment through points $(0, 0, 0, 1)$ and $(1, 0, 0, 1)$.

3. Operations

```c
d3\_segment s.to_float()
```
returns a floating point approximation of $s$.

```c
bool s.contains(const d3\_rat\_point\& p)
```
decides whether $s$ contains $p$.

```c
d3\_rat\_point s.source()
```
returns the source point of segment $s$.

```c
d3\_rat\_point s.target()
```
returns the target point of segment $s$.

```c
rational s.xcoord1()
```
returns the x-coordinate of $s$.source().

```c
rational s.xcoord2()
```
returns the x-coordinate of $s$.target().

```c
rational s.ycoord1()
```
returns the y-coordinate of $s$.source().

```c
rational s.ycoord2()
```
returns the y-coordinate of $s$.target().

```c
rational s.zcoord1()
```
returns the z-coordinate of $s$.source().

```c
rational s.zcoord2()
```
returns the z-coordinate of $s$.target().

```c
rational s.dx()
```
returns $xcoord2() - xcoord1()$. 

17.11. RATIONAL SEGMENTS IN 3D-SPACE (D3_RAT_SEGMENT)

\(s.dy()\) returns \(ycoord2() - ycoord1()\).

\(s.dz()\) returns \(zcoord2() - zcoord1()\).

\(s.project_xy()\) returns the projection into the xy plane.

\(s.project_xz()\) returns the projection into the xz plane.

\(s.project_yz()\) returns the projection into the yz plane.

\(s.project(const\ d3_rat_point&\ p, const\ d3_rat_point&\ q,\ const\ d3_rat_point&\ v)\)
returns \(s\) projected into the plane through \((p,q,v)\).

\(s.reflect(const\ d3_rat_point&\ p, const\ d3_rat_point&\ q,\ const\ d3_rat_point&\ v)\)
returns \(s\) reflected across the plane through \((p,q,v)\).

\(s.reflect(const\ d3_rat_point&\ p)\)
returns \(s\) reflected across point \(p\).

\(s.reverse()\) returns \(s\) reversed.

\(s.to_vector()\) returns \(S.target() - S.source()\).

\(s.intersection(const\ d3_rat_segment&\ t)\)
decides, whether \(s\) and \(t\) intersect in a single point.

\(s.intersection(const\ d3_rat_segment&\ t, d3_rat_point&\ p)\)
decides, whether \(s\) and \(t\) intersect. If they intersect in a single point, the point is assigned to \(p\).

\(s.intersection_of_lines(const\ d3_rat_segment&\ t, d3_rat_point&\ p)\)
If line\((s)\) and line\((t)\) intersect in a single point this point is assigned to \(p\) and the result is true, otherwise the result is false.

\(s.is_trivial()\) returns true if \(s\) is trivial.

\(s.sqr_length()\) returns the square of the length of \(s\).

\(s.translate(const\ rat_vector&\ v)\)
returns \(s\) translated by vector \(v\).
\(Precond.: v.dim() = 3.\)

\(s.translate(rational\ dx, rational\ dy, rational\ dz)\)
returns \(s\) translated by vector \((dx, dy, dz)\).

\(s.translate(integer\ dx, integer\ dy, integer\ dz, integer\ dw)\)
returns \(s\) translated by vector \((dx/dw, dy/dw, dz/w)\).
\textit{d3\_rat\_segment} \( s + \text{const } \text{rat\_vector} & v \)

returns \( s \) translated by vector \( v \).

\textit{d3\_rat\_segment} \( s - \text{const } \text{rat\_vector} & v \)

returns \( s \) translated by vector \(-v\).
17.12 Rational Planes (d3_rat_plane)

1. Definition
An instance \( P \) of the data type \( \text{d3\_rat\_plane} \) is an oriented rational plane in the three-dimensional space \( \mathbb{R}^3 \). It can be defined by a triple \((a,b,c)\) of non-collinear rational points or a single rational point \( a \) and a normal vector \( v \).

```cpp
#include <LEDA/geo/d3_rat_plane.h>
```

2. Creation

\[
d3\_rat\_plane \quad p; \quad \text{introduces a variable } p \text{ of type } d3\_rat\_plane \text{ initialized to the trivial plane.}
\]

\[
d3\_rat\_plane \quad p(\text{const } d3\_rat\_point& \ a, \text{const } d3\_rat\_point& \ b, \text{const } d3\_rat\_point& \ c);\quad \text{introduces a variable } p \text{ of type } d3\_rat\_plane \text{ initialized to the plane through } (a, b, c).
\]

**Precondition:** \( a, b, \) and \( c \) are not collinear.

\[
d3\_rat\_plane \quad p(\text{const } d3\_rat\_point& \ a, \text{const } \text{rat\_vector}& \ v);\quad \text{introduces a variable } p \text{ of type } d3\_rat\_plane \text{ initialized to the plane that contains } a \text{ with normal vector } v.
\]

**Precondition:** \( v.\text{dim}( ) = 3 \) and \( v.\text{length}( ) > 0. \)

\[
d3\_rat\_plane \quad p(\text{const } d3\_rat\_point& \ a, \text{const } d3\_rat\_point& \ b);\quad \text{introduces a variable } p \text{ of type } d3\_rat\_plane \text{ initialized to the plane that contains } a \text{ with normal vector } b - a.
\]

3. Operations

\[
d3\_rat\_point \quad p.\text{point1}( ) \quad \text{returns the first point of } p.
\]

\[
d3\_rat\_point \quad p.\text{point2}( ) \quad \text{returns the second point of } p.
\]

\[
d3\_rat\_point \quad p.\text{point3}( ) \quad \text{returns the third point of } p.
\]

\[
\text{integer} \quad p.A( ) \quad \text{returns the } A \text{ parameter of the plane equation.}
\]

\[
\text{integer} \quad p.B( ) \quad \text{returns the } B \text{ parameter of the plane equation.}
\]

\[
\text{integer} \quad p.C( ) \quad \text{returns the } C \text{ parameter of the plane equation.}
\]

\[
\text{integer} \quad p.D( ) \quad \text{returns the } D \text{ parameter of the plane equation.}
\]

\[
\text{rat\_vector} \quad p.\text{normal}( ) \quad \text{returns a normal vector of } p.
\]
d3_plane p.to_float() returns a floating point approximation of p.

rational p.sqr_dist(const d3_rat_point& q) returns the square of the Euclidean distance between p and q.

rat_vector p.normal_project(const d3_rat_point& q) returns the vector pointing from q to its projection on p along the normal direction.

int p.intersection(const d3_rat_point p1, const d3_rat_point p2, d3_rat_point& q) if the line \( l \) through \( p_1 \) and \( p_2 \) intersects \( p \) in a single point this point is assigned to \( q \) and the result is 1, if \( l \) and \( p \) do not intersect the result is 0, and if \( l \) is contained in \( p \) the result is 2.

int p.intersection(const d3_rat_plane& Q, d3_rat_point& i1, d3_rat_point& i2) if \( p \) and plane \( Q \) intersect in a line \( L \) then \((i_1, i_2)\) are assigned two different points on \( L \) and the result is 1, if \( p \) and \( Q \) do not intersect the result is 0, and if \( p = Q \) the result is 2.

d3_rat_plane p.translate(const rational& dx, const rational& dy, const rational& dz) returns \( p \) translated by vector \((dx, dy, dz)\).

d3_rat_plane p.translate(integer dx, integer dy, integer dz, integer dw) returns \( p \) translated by vector \((dx/dw, dy/dw, dz/dw)\).

d3_rat_plane p.translate(const rat_vector& v) returns \( p + v \), i.e., \( p \) translated by vector \( v \). 
Precondition: \( v.d() = 3 \).

d3_rat_plane p + const rat_vector& v returns \( p \) translated by vector \( v \).

d3_rat_plane p.reflect(const d3_rat_plane& Q) returns \( p \) reflected across plane \( Q \).

d3_rat_plane p.reflect(const d3_rat_point& q) returns \( p \) reflected across point \( q \).

d3_rat_point p.reflect_point(const d3_rat_point& q) returns \( q \) reflected across plane \( p \).

int p.side_of(const d3_rat_point& q) computes the side of \( p \) on which \( q \) lies.
bool p.contains(const d3_rat_point& q)
returns true if point q lies on plane p, i.e.,
(p.side_of(q) == 0), and false otherwise.

bool p.parallel(const d3_rat_plane& Q)
returns true if planes p and Q are parallel, and false
otherwise.

ostream& ostream& O ≪ const d3_rat_plane& p
writes p to output stream O.

istream& istream& I >> d3_rat_plane& p
reads p from input stream I.

Non-Member Functions

int orientation(const d3_rat_plane& p, const d3_rat_point& q)
computes the orientation of p.sideof(q).
CHAPTER 17. BASIC DATA TYPES FOR THREE-DIMENSIONAL GEOMETRY

17.13 Rational Spheres ( d3_rat_sphere )

1. Definition

An instance of the data type d3_rat_sphere is an oriented sphere in 3d space. The sphere is defined by four points \( p_1, p_2, p_3, p_4 \) with rational coordinates (d3_rat_points).

```cpp
#include <LEDA/geo/d3_rat_sphere.h>
```

2. Creation

```cpp
d3_rat_sphere S(const d3_rat_point& p1, const d3_rat_point& p2,
const d3_rat_point& p3, const d3_rat_point& p4);
```

introduces a variable \( S \) of type d3_rat_sphere. \( S \) is initialized to the sphere through points \( p_1, p_2, p_3, p_4 \).

3. Operations

```cpp
d3_sphere S.to_float() returns a floating point approximation of \( S \).
bool S.contains(const d3_rat_point& p) returns true, if \( p \) is on the sphere, false otherwise.
bool S.inside(const d3_rat_point& p) returns true, if \( p \) is inside the sphere, false otherwise.
bool S.outside(const d3_rat_point& p) returns true, if \( p \) is outside the sphere, false otherwise.
d3_rat_point S.point1() returns \( p_1 \).
d3_rat_point S.point2() returns \( p_2 \).
d3_rat_point S.point3() returns \( p_3 \).
d3_rat_point S.point4() returns \( p_4 \).
bool S.is_degenerate() returns true, if the 4 defining points are coplanar.
d3_rat_point S.center() returns the center of the sphere.
rational S.sqr_radius() returns the square of the radius.
d3_rat_sphere S.translate(const rat_vector& v) translates the sphere by vector \( v \) and returns a new d3_rat_sphere.
```
$d3\text{\_rat\_sphere}\ S.\text{translate}(\text{const rational}\&\ r1, \text{const rational}\&\ r2, \text{const rational}\&\ r3)$

translates the sphere by vector $(r1, r2, r3)$ and returns a new $d3\text{\_rat\_sphere}$.
17.14 Rational Simplices (d3_rat_simplex)

1. Definition

An instance of the data type d3_rat_simplex is a simplex in 3d space. The simplex is defined by four points $p_1, p_2, p_3, p_4$ with rational coordinates (d3_rat_points). We call the simplex degenerate, if the four defining points are coplanar.

```
#include <LEDA/geo/d3_rat_simplex.h>
```

2. Types

- d3_rat_simplex::coord_type the coordinate type (rational).
- d3_rat_simplex::point_type the point type (d3_rat_point).

3. Creation

```
d3_rat_simplex S(const d3_rat_point& a, const d3_rat_point& b, const d3_rat_point& c, const d3_rat_point& d);
```

creates the simplex $(a, b, c, d)$.

```
d3_rat_simplex S; creates the simplex $((0, 0, 0), (1, 0, 0), (0, 1, 0), (0, 0, 1))$.
```

4. Operations

```
d3_simplex S.to_d3_simplex() returns a floating point approximation of S.
```

```
d3_rat_point S.point1() returns $p_1$.
```

```
d3_rat_point S.point2() returns $p_2$.
```

```
d3_rat_point S.point3() returns $p_3$.
```

```
d3_rat_point S.point4() returns $p_4$.
```

```
d3_rat_point S[int i] returns $p_i$. Precondition: $i > 0$ and $i < 5$.
```

```
int S.index(const d3_rat_point& p) returns 1 if $p == p_1$, 2 if $p == p_2$, 3 if $p == p_3$, 4 if $p == p_4$, 0 otherwise.
```

```
bool S.is_degenerate() returns true if $S$ is degenerate and false otherwise.
```

```
d3_rat_sphere S.circumscribing_sphere() returns a d3_rat_sphere through $(p_1, p_2, p_3, p_4)$ (precondition: the d3_rat_simplex is not degenerate).
```

**17.14. RATIONAL SIMPLICES (D3_RAT_SIMPLEX)**

- `bool S.insimplex(const d3_rat_point& p)`
  - returns true, if p is contained in the simplex.

- `bool S.insphere(const d3_rat_point& p)`
  - returns true, if p lies in the interior of the sphere through p1, p2, p3, p4.

- `rational S.vol()`
  - returns the signed volume of the simplex.

- `d3_rat_simplex S.reflect(const d3_rat_point& p, const d3_rat_point& q, const d3_rat_point& v)`
  - returns S reflected across the plane through (p, q, v).

- `d3_rat_simplex S.reflect(const d3_rat_point& p)`
  - returns S reflected across point p.

- `d3_rat_simplex S.translate(const rat_vector& v)`
  - returns S translated by vector v.
  - **Precond.:** v.dim() = 3.

- `d3_rat_simplex S.translate(rational dx, rational dy, rational dz)`
  - returns S translated by vector (dx, dy, dz).

- `d3_rat_simplex S.translate(integer dx, integer dy, integer dz, integer dw)`
  - returns S translated by vector (dx/dw, dy/dw, dz/w).

- `d3_rat_simplex S + const rat_vector& v`
  - returns S translated by vector v.

- `d3_rat_simplex S - const rat_vector& v`
  - returns S translated by vector −v.
17.15 3D Convex Hull Algorithms (d3_hull)

`void CONVEX_HULL(const list<d3_rat_point>& L, GRAPH<d3_rat_point, int>& H)`

CONVEX_HULL takes as argument a list of points and returns the (planar embedded) surface graph $H$ of the convex hull of $L$. The algorithm is based on an incremental space sweep. The running time is $O(n^2)$ in the worst case and $O(n \log n)$ for most inputs.

`bool CHECK_HULL(const GRAPH<d3_rat_point, int>& H)`

a checker for convex hulls.

`void CONVEX_HULL(const list<d3_point>& L, GRAPH<d3_point, int>& H)`

a floating point version of CONVEX_HULL.

`bool CHECK_HULL(const GRAPH<d3_point, int>& H)`

a checker for floating-point convex hulls.
17.16 3D Triangulation and Voronoi Diagram Algorithms (d3_delaunay)

void D3_TRIANG(const list<d3_rat_point>& L, GRAPH<d3_rat_point, int>& G)
computes a triangulation $G$ of the points in $L$.

void D3_DELAUNAY(const list<d3_rat_point>& L, GRAPH<d3_rat_point, int>& G)
computes a delaunay triangulation $G$ of the points in $L$.

void D3_VORONOI(const list<d3_rat_point>& L0, GRAPH<d3_rat_sphere, int>& G)
computes the voronoi diagram $G$ of the points in $L$. 
Chapter 18

Graphics

This section describes the data types color, window, panel, and menu.

18.1 Colors (color)

1. Definition

The data type color is the type of all colors available for drawing operations in windows (cf. 18.2). Each color is defined by a triple of integers \((r,g,b)\) with \(0 \leq r,g,b \leq 255\), the so-called rgb-value of the color. The number of available colors is restricted and depends on the underlying hardware. Colors can be created from rgb-values, from names in a color data base (X11), or from the 16 integer constants (enumeration in 
\(<LEDA/graphics/x\_window.h>\) black, white, red, green, blue, yellow, violet, orange; cyan, brown, pink, green2, blue2, grey1, grey2, grey3.

#include <LEDA/graphics/color.h>

2. Creation

\[\text{color \ col; \ creates a color with rgb-value } (0,0,0) \text{ (i.e. black).}\]

\[\text{color \ col(int \ r, \ int \ g, \ int \ b); \ creates a color with rgb-value } (r,g,b).\]

\[\text{color \ col(const char* \ name); \ creates a color and initializes it with the rgb-value of color name from the X11 color data base } ((0,0,0) \text{ if name does not exist}).\]

\[\text{color \ col(int \ i); \ creates a color and initializes it with one of the 16 predefined colors. Here } i \text{ is one of the 16 integer constants black, white, red, green, blue, yellow, violet, orange, cyan, brown, pink, green2, blue2, grey1, grey2, or grey3.}\]
3. Operations

```cpp
void col.set_rgb(int r, int g, int b) // sets the rgb-value of col to (r, g, b).

void col.get_rgb(int& r, int& g, int& b) // assigns the rgb-value of col to (r, g, b).

void col.set_red(int r) // changes the r-value of col to r.

void col.set_green(int g) // changes the g-value of col to g.

void col.set_blue(int b) // changes the b-value of col to b.
```
18.2 Windows ( window )

1. Definition

The data type window provides an interface for graphical input and output of basic two-dimensional geometric objects. Application programs using this data type have to be linked with libW.a and (on UNIX systems) with the X11 base library libX11.a (cf. section 1.6):

CC prog.c -lW -lP -lG -lL -lX11 -lm

An instance W of type window is an iso-oriented rectangular window in the two-dimensional plane. The default representation of W on the screen is a square of maximal possible edge length positioned in the upper right corner of the display.

In general, a window consists of two rectangular sections, a panel section in the upper part and a drawing section in the rest of the window. The panel section contains panel items such as sliders, choice fields, string items and buttons. They have to be created before the window is opened by special panel operations described in section 18.2.

The drawing section can be used for the output of geometric objects such as points, lines, segments, arrows, circles, polygons, graph, ... and for the input of all these objects using the mouse input device. All drawing and input operations in the drawing section use a coordinate system that is defined by three parameters of type double: xmin, the minimal x-coordinate, xmax, the maximal x-coordinate, and ymin, the minimal y-coordinate. The two parameters xmin and xmax define the scaling factor scaling as \( w/(xmax - xmin) \), where w is the width of the drawing section in pixels. The maximal y-coordinate ymax of the drawing section is equal to ymax + h · scaling and depends on the actual shape of the window on the screen. Here, h is the height of the drawing section in pixels.

A list of all window parameters:

1. The foreground color parameter (default black) defines the default color to be used in all drawing operations. There are 18 predefined colors (enumeration in <LEDA/graphics/x_window.h>): black, white, red, green, blue, yellow, violet, orange, cyan, brown, pink, green2, blue2, grey1, grey2, grey3 ivory, and invisible. Note that all drawing operations have an optional color argument that can be used to override the default foreground color. On monochrome systems all colors different from white are displayed as black. The color invisible can be used for invisible (transparent) objects.

2. The background color parameter (default white) defines the default background color (e.g. used by W.clear()).

3. The text font parameter defines the name of the font to be used in all text drawing operations.
4. Minimal and maximal coordinates of the drawing area $xmin$ (default 0), $xmax$ (default 100), $ymin$ (default 0).

5. The grid width parameter (default 0) defines the width of the grid that is used in the drawing area. A grid width of 0 indicates that no grid is to be used.

6. The frame label parameter defines a string to be displayed in the frame of the window.

7. The show coordinates flag (default true) determines whether the current coordinates of the mouse cursor in the drawing section are displayed in the upper right corner.

8. The flush output flag (default true) determines whether the graphics output stream is flushed after each draw action.

9. The line width parameter (default value 1 pixel) defines the width of all kinds of lines (segments, arrows, edges, circles, polygons).

10. The line style parameter defines the style of lines. Possible line styles are solid (default), dashed, and dotted.

11. The point style parameter defines the style points are drawn by the draw_point operation. Possible point styles are pixel_point, cross_point (default), plus_point, circle_point, disc_point, rect_point, and box_point.

12. The node width parameter (default value 8 pixels) defines the diameter of nodes created by the draw_node and draw_filled_node operations.

13. The text mode parameter defines how text is inserted into the window. Possible values are transparent (default) and opaque.

14. The show orientation parameter defines, whether or not the direction or orientation of segments, lines, rays, triangles, polygons and gen_polygons will be shown (default false.)

15. The drawing mode parameter defines the logical operation that is used for setting pixels in all drawing operations. Possible values are src_mode (default) and xor_mode. In src_mode pixels are set to the respective color value, in xor_mode the value is bitwise added to the current pixel value.

16. The redraw function parameter is a pointer to a function of type void ($*F$)(window*). It is called with a pointer to the corresponding window as argument to redraw (parts of) the window whenever a redrawing is necessary, e.g., if the shape of the window is changed or previously hidden parts of it become visible.

17. The window delete handler parameter is a pointer to a function of type void ($*F$)(window*). It is called with a pointer to the corresponding window as argument when the window is to be closed by the window manager (e.g. by pressing the ×-button on Windows-NT systems). The default window delete handler closes the window and terminates the program.
18. The *buttons per line* parameter (default $\infty$) defines the maximal number of buttons in one line of the panel section.

19. The *precision* parameter (default 16) defines the precision that is used for representing window coordinates, more precisely, all $x$ and $y$ coordinates generated by window input operations are doubles whose mantissa are truncated after *precision* $- 1$ bits after the binary point.

In addition to call-back (handler) functions LEDA windows now also support the usage of function objects. Function object classes have to be derived from the `window_handler` base class.

```cpp
class window_handler {
    ...
    virtual void operator()() { }

    // parameter access functions
    double get_double(int nr) const;
    int get_int() const;
    window* get_window_ptr() const;
    char* get_char_ptr() const;
};
```

Derived classes have to implement the handling function in the definition of the `operator()` method. The different `get_` methods can be called to retrieve parameters.

If both, a handler function and an object for the same action is supplied the object has higher priority.

```cpp
#include <LEDA/graphics/window.h>
```

2. Creation

```cpp
window W;       creates a squared window with maximal possible edge length (minimum of width and height of the display).

window W(const char * label);  creates a maximal squared window with frame label *label*.

window W(int w, int h);     creates a window *W* of physical size $w$ pixels $\times$ $h$ pixels.

window W(int w, int h, const char * label); creates a window *W* of physical size $w$ pixels $\times$ $h$ pixels and frame label *label*.
```
All four variants initialize the coordinates of W to xmin = 0, xmax = 100 and ymin = 0. The init operation (see below) can later be used to change the window coordinates and scaling. Please note, that a window is not displayed before the function display is called for it.

3. Operations

3.1 Initialization

```c
void W.init(double x0, double x1, double y0)
```

sets $x_{\text{min}}$ to $x_0$, $x_{\text{max}}$ to $x_1$, and $y_{\text{min}}$ to $y_0$, the scaling factor $\text{scaling}$ to $w/(x_{\text{max}}-x_{\text{min}})$, and $y_{\text{max}}$ to $y_{\text{min}}+h/\text{scaling}$. Here $w$ and $h$ are the width and height of the drawing section in pixels.

```c
void W.init_rectangle(double x0, double x1, double y0, double y1)
```

adjusts the window such that the points $(x_0, y_0)$ and $(x_1, y_1)$ are contained in the drawing section.

```c
double W.set_grid_dist(double d)
```

sets the grid distance of $W$ to $d$.

```c
grid_style W.set_grid_style(grid_style s)
```

sets the grid style of $W$ to $s$.

```c
int W.set_grid_mode(int d)
```

sets the grid distance of $W$ to $d$ pixels.

```c
int W.set_precision(int prec)
```

sets the precision of $W$ to $prec$.

```c
void W.init(double x0, double x1, double y0, int d, bool erase = true)
```

same as $W$.init($x_0, x_1, y_0$) followed by $W$.set_grid_mode($d$). If the optional flag erase is set to false the window will not be erased.

```c
void W.display()
```

opens $W$ and displays it with its right upper corner in the upper right corner of the screen. Note that $W$.display() has to be called before all drawing operations and that all operations adding panel items to $W$ (cf. 18.2) have to be called before the first call of $W$.display().
18.2. WINDOWS ( WINDOW )

void W.display(int x, int y) opens W and displays it with its left upper corner at position (x, y). Special values for x and y are window::min, window::center, and window::max for positioning W at the minimal or maximal x or y coordinate or centering it in the x or y dimension.

void W.display(window & W0, int x, int y)

opens W and displays it with its left upper corner at position (x, y) relative to the upper left corner of window W0.

W.open... can be used as a synonym for W.display... Note, that the open operation for panels (cf. 18.3) is defined slightly different.

void W.close() closes W by removing it from the display.

void W.clear() clears W using the current background color or pixmap, i.e., if W has a background pixmap defined it is tiled with P such that the upper left corner is the tiling origin. Otherwise, it is filled with background color of W.

void W.clear(double x0, double y0, double x1, double y1)

only clears the rectangular area (x0,y0,x1,y1) of window W using the current background color or pixmap.

void W.clear(color c)

clears W with color c and sets the background color of W to c.

void W.clear(double xorig, double yorig)

clears W. If a background pixmap is defined the point (xorig,yorig) is used as the origin of tiling.

void W.redraw()

repaints the drawing area if W has a redraw function.

3.2 Setting parameters

color W.set_color(color c) sets the foreground color parameter to c and returns its previous value.

color W.set_fill_color(color c) sets the fill color parameter (used by ≪ operators) to c and returns its previous value.

color W.set_bg_color(color c) sets the background color parameter to c and returns its previous value.
\textbf{char*} \quad \texttt{W.set\_bg\_ pixmap(\textit{char* pr})}

sets the background pixmap to \textit{pr} and returns its previous value.

\textbf{int} \quad \texttt{W.set\_line\_width(\textit{int pix})}

sets the line width parameter to \textit{pix} pixels and returns its previous value.

\textbf{line\_style} \quad \texttt{W.set\_line\_style(\textit{line\_style s})}

sets the line style parameter to \textit{s} and returns its previous value.

\textbf{int} \quad \texttt{W.set\_node\_width(\textit{int pix})}

sets the node width parameter to \textit{pix} pixels and returns its previous value.

\textbf{text\_mode} \quad \texttt{W.set\_text\_mode(\textit{text\_mode m})}

sets the text mode parameter to \textit{m} and returns its previous value.

\textbf{drawing\_mode} \quad \texttt{W.set\_mode(\textit{drawing\_mode m})}

sets the drawing mode parameter to \textit{m} and returns its previous value.

\textbf{int} \quad \texttt{W.set\_cursor(\textit{int cursor\_id} = -1)}

sets the mouse cursor of \texttt{W} to \textit{cursor\_id}. Here \textit{cursor\_id} must be one of the constants predefined in \texttt{<X11/cursorfont.h>} or \texttt{-1} for the system default cursor. Returns the previous cursor.

\textbf{void} \quad \texttt{W.set\_show\_coordinates(\textit{bool b})}

sets the show coordinates flag to \textit{b}.

\textbf{bool} \quad \texttt{W.set\_show\_orientation(\textit{bool orient})}

sets the show orientation parameter to \textit{orient}.

\textbf{void} \quad \texttt{W.set\_frame\_label(\textit{string s})}

makes \textit{s} the window frame label.

\textbf{void} \quad \texttt{W.set\_icon\_label(\textit{string s})}

makes \textit{s} the window icon label.

\textbf{void} \quad \texttt{W.reset\_frame\_label(\textit{)})}

restores the standard LEDA frame label.

\textbf{void} \quad \texttt{W.set\_window\_delete\_handler(\textit{void (\textit{*F})(\textit{window*})})}

sets the window delete handler function parameter to \textit{F}. 
void W.set_window_delete_object(const window_handler& obj)
    sets the window delete object parameter to obj.

void W.set_show_coord_handler(void (*)(window*, double, double))
    sets the show coordinate handler function parameter to F.

void W.set_show_coord_object(const window_handler& obj)
    sets the show coordinate object parameter to obj.

void W.set_redraw(void (*)(window*))
    sets the redraw function parameter to F.

void W.set_redraw(const window_handler& obj)
    sets the redraw object parameter to obj.

void W.set_redraw(void (*)(window*, double, double, double, double) = 0)
    sets the redraw function parameter to F.

void W.set_redraw2(const window_handler& obj)
    sets the redraw object parameter to obj.

void W.set_bg_redraw(void (*)(window*, double, double, double, double, double) = 0)
    sets the background redraw function parameter to F.

void W.set_bg_redraw(const window_handler& obj)
    sets the background redraw object parameter to obj.

void W.start_timer(int msec, void (*)(window*))
    starts a timer that runs F every msec milliseconds with a pointer to W.

void W.start_timer(int msec, const window_handler& obj)
    starts a timer that runs the operator() of obj every msec milliseconds.

void W.stop_timer() stops the timer.

void W.set_flush(bool b) sets the flush parameter to b.

void W.set_icon_pixrect(char* pr)
    makes pr the new icon of W.
void* W.set_client_data(void *p, int i = 0) sets the i-th client data pointer of W to p and returns its previous value. Precondition: i < 16.

3.3 Reading parameters

int W.get_line_width() returns the current line width.

line_style W.get_line_style() returns the current line style.

int W.get_node_width() returns the current node width.

text_mode W.get_text_mode() returns the current text mode.

drawing_mode W.get_mode() returns the current drawing mode.

int W.get_cursor() returns the id of the current cursor, i.e, one of the constants predefined in X11/cursorfont.h or -1 for the default cursor.

double W.xmin() returns the minimal x-coordinate of the drawing area of W.

double W.ymin() returns the minimal y-coordinate of the drawing area of W.

double W.xmax() returns the maximal x-coordinate of the drawing area of W.

double W.ymax() returns the maximal y-coordinate of the drawing area of W.

double W.scale() returns the scaling factor of the drawing area of W, i.e. the number of pixels of a unit length line segment.

double W.get_grid_dist() returns the width of the current grid (zero if no grid is used).

grid_style W.get_grid_style() returns the current grid style.

int W.get_grid_mode() returns the width of the current grid in pixels (zero if no grid is used).

bool W.get_show_orientation() returns the show orientation parameter.

void* W.get_client_data(int i = 0) returns the i-th client data pointer of of W. Precondition: i < 16.
GraphWin* W.get_graphwin() returns a pointer to the GraphWin (see 18.6) that uses W as its display window or NULL if W is not used by any GraphWin.

GeoWinTypeName* W.get_geowin() returns a pointer to the GeoWin (see Section 18.8) that uses W as its display window or NULL if W is not used by any GeoWin.

int W.width() returns the width of W in pixels.

int W.height() returns the height of W in pixels.

int W.menu_bar_height() returns the height of the menu bar of W in pixels and 0 if W has no menu bar (see W.make_menu_bar( )).

int W.xpos() returns the x-coordinate of the upper left corner of the frame of W.

int W.ypos() returns the y-coordinate of the upper left corner of the frame of W.

int W.get_state() returns the state of W.

void W.set_state(int stat) sets the state of W to stat.

bool W.contains(const point& p) returns true if p lies in the drawing area.

3.4 Drawing Operations

All drawing operations have an optional color argument at the end of the parameter list. If this argument is omitted the current foreground color (cf. section 18.2) of W is used.

3.4.1 Drawing points

void W.draw_point(double x, double y, color c = window::fgcol)
        draws the point (x, y) (a cross of two short segments).

void W.draw_point(const point& p, color c = window::fgcol)
        draws point p.

void W.draw_pixel(double x, double y, color c = window::fgcol)
        sets the color of the pixel at position (x, y) to c.

void W.draw_pixel(const point& p, color c = window::fgcol)
        sets the color of the pixel at position p to c.

void W.draw_pixels(const list<point>& L, color c = window::fgcol)
        sets the color of all pixels in L to c.
void W.draw_pixels(int n, double *xcoord, double *ycoord, color c = window::fgcol)
draws all pixels (xcoord[i], ycoord[i]) for 0 ≤ i ≤ n - 1.

3.4.2 Drawing line segments

void W.draw_segment(double x1, double y1, double x2, double y2,
                    color c = window::fgcol)
draws a line segment from (x1, y1) to (x2, y2).
void W.draw_segment(const point& p, const point& q, color c = window::fgcol)
draws a line segment from point p to point q.
void W.draw_segment(const segment& s, color c = window::fgcol)
draws line segment s.
void W.draw_segment(point p, point q, line l, color c = window::fgcol)
draws the part of the line l between p and q. This version
of draw_segment should be used if p or q may lie far outside
W. Precondition: p and q lie on l or at least close to l.
void W.draw_segments(const list<segment>& L, color c = window::fgcol)
draws all segments in L.

3.4.3 Drawing lines

void W.draw_line(double x1, double y1, double x2, double y2,
                  color c = window::fgcol)
draws a straight line passing through points (x1, y1) and
(x2, y2).
void W.draw_line(const point& p, const point& q, color c = window::fgcol)
draws a straight line passing through points p and q.
void W.draw_line(const segment& s, color c = window::fgcol)
draws the line supporting segment s.
void W.draw_line(const line& l, color c = window::fgcol)
draws line l.
void W.draw_hline(double y, color c = window::fgcol)
draws a horizontal line with y-coordinate y.
void W.draw_vline(double x, color c = window::fgcol)
draws a vertical line with x-coordinate x.

3.4.4 Drawing Rays

void W.draw_ray(double x1, double y1, double x2, double y2, color c = window::fgcol)
draws a ray starting in (x1, y1) and passing through
(x2, y2).
18.2. WINDOWS ( WINDOW )

```cpp
void W::draw_ray(const point& p, const point& q, color c = window::fgcol)
    draws a ray starting in p and passing through q.

void W::draw_ray(const segment& s, color c = window::fgcol)
    draws a ray starting in s.source() containing s.

void W::draw_ray(const ray& r, color c = window::fgcol)
    draws ray r.

void W::draw_ray(point p, point q, line l, color c = window::fgcol)
    draws the part of the line l on the ray with source p and
    passing through q. This version of draw_ray should be used
    if p may lie far outside W. Precondition: p and q lie on l
    or at least close to l.
```

### 3.5.0 Drawing Arcs and Curves

```cpp
void W::draw_arc(const point& p, const point& q, const point& r,
                 color c = window::fgcol)
    draws a circular arc starting in p passing through q and
    ending in r.

void W::draw bezier(const list<point>& C, int n, color c = window::fgcol)
    draws the bezier curve with control polygon C by a poly-
    line with n points.

void W::draw_spline(const list<point>& L, int n, color c = window::fgcol)
    draws a spline curve through the points of L. Each seg-
    ment is approximated by a polyline with m points.

void W::draw_closed_spline(const list<point>& L, int n, color c = window::fgcol)
    draws a closed spline through the points of L.

void W::draw_spline(const polygon& P, int n, color c = window::fgcol)
    draws a closed spline through the vertices of P.
```

### 3.4.6 Drawing arrows

```cpp
void W::draw_arrow(double x1, double y1, double x2, double y2,
                     color c = window::fgcol)
    draws an arrow pointing from (x1, y1) to (x2, y2).

void W::draw_arrow(const point& p, const point& q, color c = window::fgcol)
    draws an arrow pointing from point p to point q.

void W::draw_arrow(const segment& s, color = window::fgcol)
    draws an arrow pointing from s.start() to s.end().
```
CHAPTER 18. GRAPHICS

```c
void W.draw_polyline_arrow(const list<point>& lp, color c = window::fgcol)
    draws a polyline arrow with vertex sequence lp.

void W.draw_arc_arrow(const point& p, const point& q, const point& r,
                      color c = window::fgcol)
    draws a circular arc arrow starting in p passing through q and ending in r.

void W.drawBezier_arrow(const list<point>& C, int n, color c = window::fgcol)
    draws the bezier curve with control polygon C by a polyline with n points, the last segment is drawn as an arrow.

void W.draw_spline_arrow(const list<point>& L, int n, color c = window::fgcol)
    draws a spline curve through the points of L. Each segment is approximated by a polyline with n points. The last segment is drawn as an arrow.

point W.drawarrow_head(const point& p, double dir, color c = window::fgcol)
    draws an arrow head at position p pointing to direction dir, where dir is an angle from [0, 2\pi].
```

### 3.4.7 Drawing circles

```c
void W.draw_circle(double x, double y, double r, color c = window::fgcol)
    draws the circle with center (x, y) and radius r.

void W.draw_circle(const point& p, double r, color c = window::fgcol)
    draws the circle with center p and radius r.

void W.draw_circle(const circle& C, color c = window::fgcol)
    draws circle C.

void W.draw_ellipse(double x, double y, double r1, double r2, color c = window::fgcol)
    draws the ellipse with center (x, y) and radii r1 and r2.

void W.draw_ellipse(const point& p, double r1, double r2, color c = window::fgcol)
    draws the ellipse with center p and radii r1 and r2.
```

### 3.4.8 Drawing discs

```c
void W.draw_disc(double x, double y, double r, color c = window::fgcol)
    draws a filled circle with center (x, y) and radius r.

void W.draw_disc(const point& p, double r, color c = window::fgcol)
    draws a filled circle with center p and radius r.

void W.draw_disc(const circle& C, color c = window::fgcol)
    draws filled circle C.
```
18.2. WINDOWS ( WINDOW )

```cpp
void W.draw_filled_circle(double x, double y, double r, color c = window::fgcol)
    draws a filled circle with center \((x, y)\) and radius \(r\).

void W.draw_filled_circle(const point& p, double r, color c = window::fgcol)
    draws a filled circle with center \(p\) and radius \(r\).

void W.draw_filled_circle(const circle& C, color c = window::fgcol)
    draws filled circle \(C\).

void W.draw_filled_ellipse(double x, double y, double r1, double r2, color c = window::fgcol)
    draws a filled ellipse with center \((x, y)\) and radii \(r_1\) and \(r_2\).

void W.draw_filled_ellipse(const point& p, double r1, double r2, color c = window::fgcol)
    draws a filled ellipse with center \(p\) and radii \(r_1\) and \(r_2\).
```

### 3.4.9 Drawing polygons

```cpp
void W.draw_polyline(const list<point>& lp, color c = window::fgcol)
    draws a polyline with vertex sequence \(lp\).

void W.draw_polyline(int n, double *xc, double *yc, color c = window::fgcol)
    draws a polyline with vertex sequence \((xc[0], yc[0]), \ldots, (xc[n-1], yc[n-1])\).

void W.draw_polygon(const list<point>& lp, color c = window::fgcol)
    draws the polygon with vertex sequence \(lp\).

void W.draw_oriented_polygon(const list<point>& lp, color c = window::fgcol)
    draws the polygon with vertex sequence \(lp\) and indicates the orientation by an arrow.

void W.draw_polygon(const polygon& P, color c = window::fgcol)
    draws polygon \(P\).

void W.draw_oriented_polygon(const polygon& P, color c = window::fgcol)
    draws polygon \(P\) and indicates the orientation by an arrow.

void W.draw_filled_polygon(const list<point>& lp, color c = window::fgcol)
    draws the filled polygon with vertex sequence \(lp\).

void W.draw_filled_polygon(const polygon& P, color c = window::fgcol)
    draws filled polygon \(P\).

void W.draw_polygon(const gen_polygon& P, color c = window::fgcol)
    draws polygon \(P\).
void W.draw_oriented_polygon(const gen_polygon& P, color c = window::fgcol)
    draws polygon P and indicates the orientation by an arrow.

void W.draw_filled_polygon(const gen_polygon& P, color c = window::fgcol)
    draws filled polygon P.

void W.draw_rectangle(double x0, double y0, double x1, double y1, 
    color = window::fgcol)
    draws a rectangle with lower left corner (x0, y0) and upper 
    right corner (x1, y1).
    Precondition: x0 < x1 and y0 < y1.

void W.draw_rectangle(point p, point q, color = window::fgcol)
    draws a rectangle with lower left corner p and upper right 
    corner q.
    Precondition: p < q.

void W.draw_rectangle(const rectangle& R, color = window::fgcol)
    draws rectangle R.

void W.draw_box(double x0, double y0, double x1, double y1, color c = window::fgcol)
    draws a filled rectangle with lower left corner (x0, y0) and upper 
    right corner (x1, y1).
    Precondition: x0 < x1 and y0 < y1.

void W.draw_filled_rectangle(point p, point q, color = window::fgcol)
    draws a filled rectangle with lower left corner p and upper right 
    corner q.
    Precondition: p < q.

void W.draw_filled_rectangle(const rectangle& R, color = window::fgcol)
    draws rectangle R.

void W.draw_box(point p, point q, color c = window::fgcol)
    same as draw_filled_rectangle(p, q, c).

void W.draw_box(const rectangle& R, color c = window::fgcol)
    same as draw_filled_rectangle(p, q, c).

void W.draw_roundrect(double x0, double y0, double x1, double y1, double rndness, 
    color col = window::fgcol)
    draws a rectangle (x0, y0, x1, y1) with round corners. The 
    rndness argument must be a real number in the interval 
    [0,1] and defines the “roundness” of the rectangle.

void W.draw_roundrect(point p, point q, double rndness, color col = window::fgcol)
    draws a round rectangle with lower left corner p, upper 
    right corner q, and roundness rndness.
void W.draw_roundbox(double x0, double y0, double x1, double y1, double rndness, color col = window::fgcol)
    draws a filled rectangle \((x_0, y_0, x_1, y_1)\) with round corners. The \textit{rndness} argument must be a real number in the interval \([0, 1]\) and defined the “roundness” of the rectangle.

void W.draw_roundbox(point p, point q, double rndness, color col = window::fgcol)
    draws a round filled rectangle with lower left corner \(p\), upper right corner \(q\), and roundness \textit{rndness}.

void W.draw_triangle(point a, point b, point c, color = window::fgcol)
    draws triangle \((a, b, c)\).

void W.draw_triangle(const triangle& T, color = window::fgcol)
    draws triangle \(T\).

void W.draw_filled_triangle(point a, point b, point c, color = window::fgcol)
    draws filled triangle \((a, b, c)\).

void W.draw_filled_triangle(const triangle& T, color = window::fgcol)
    draws filled triangle \(T\).

3.4.10 Drawing functions

void W.plot_xy(double x0, double x1, win_draw_func F, color c = window::fgcol)
    draws the graph of function \(F\) in the x-range \([x_0, x_1]\), i.e., all pixels \((x, y)\) with \(y = F(x)\) and \(x_0 \leq x \leq x_1\).

void W.plot_yx(double y0, double y1, win_draw_func F, color c = window::fgcol)
    draws the graph of function \(F\) in the y-range \([y_0, y_1]\), i.e., all pixels \((x, y)\) with \(x = F(y)\) and \(y_0 \leq y \leq y_1\).

3.4.11 Drawing text

void W.draw_text(double x, double y, string s, color c = window::fgcol)
    writes string \(s\) starting at position \((x, y)\).

void W.draw_text(const point& p, string s, color c = window::fgcol)
    writes string \(s\) starting at position \(p\).

void W.draw_ctext(double x, double y, string s, color c = window::fgcol)
    writes string \(s\) centered at position \((x, y)\).

void W.draw_ctext(const point& p, string s, color c = window::fgcol)
    writes string \(s\) centered at position \(p\).

void W.draw_ctext(string s, color c = window::fgcol)
    writes string \(s\) centered in window \(W\).
The text provides a description of how to format and write string representations into boxes using a function `W.text_box(x0, x1, y, s, draw = true)`.

`W.text_box` formats and writes string `s` into a box with its left border at `x-coordinate x0`, its right border at `x1`, and its upper border at `y-coordinate y`. Some LaTeX-like formatting commands can be used: \bf, \tt, \rm, \n, \c, \color, \ldots returns y-coordinate of lower border of box. If the optional last parameter `draw` is set to `false` no drawing takes place and only the lower y-coordinate of the box is computed.

- `void W.text_box(string s)` as above with `x0 = W.xmin( )`, `x1 = W.xmax( )`, and `y = W.ymax( )`.
- `void W.message(string s)` displays the message `s` (each call adds a new line).
- `void W.del_message( )` deletes the text written by all previous message operations.

### 3.4.12 Drawing nodes

Nodes are represented by circles of diameter `node_width`.

- `void W.draw_node(double x0, double y0, color c = window::fgcol)` draws a node at position `(x0, y0)`.
- `void W.draw_node(const point& p, color c = window::fgcol)` draws a node at position `p`.
- `void W.draw_filled_node(double x0, double y0, color c = window::bgcol)` draws a filled node at position `(x0, y0)`.
- `void W.draw_filled_node(const point& p, color c = window::bgcol)` draws a filled node at position `p`.
- `void W.draw_text_node(double x, double y, string s, color c = window::bgcol)` draws a node with label `s` at position `(x, y)`.
- `void W.draw_text_node(const point& p, string s, color c = window::bgcol)` draws a node with label `s` at position `p`.
- `void W.draw_int_node(double x, double y, int i, color c = window::bgcol)` draws a node with integer label `i` at position `(x, y)`.
- `void W.draw_int_node(const point& p, int i, color c = window::bgcol)` draws a node with integer label `i` at position `p`.

### 3.4.13 Drawing edges

Edges are drawn as straight line segments or arrows with a clearance of `node_width/2` at each end.
void W.draw_edge(double x1, double y1, double x2, double y2, color c = window::fgcol)
draws an edge from (x1, y1) to (x2, y2).

void W.draw_edge(const point& p, const point& q, color c = window::fgcol)
draws an edge from p to q.

void W.draw_edge(const segment& s, color c = window::fgcol)
draws an edge from s.start() to s.end().

void W.draw_edge_arrow(double x1, double y1, double x2, double y2,
color c = window::fgcol)
draws a directed edge from (x1, y1) to (x2, y2).

void W.draw_edge_arrow(const point& p, const point& q, color c = window::fgcol)
draws a directed edge from p to q.

void W.draw_edge_arrow(const segment& s, color c = window::fgcol)
draws a directed edge from s.start() to s.end().

3.4.14 Bitmaps and Pixrects

char* W.create_bitmap(int w, int h, unsigned char* bm_data)
creates a bitmap (monochrome pixrect) of width w, height h, from the bits in data.

cchar* W.create_pixrect(const char** xpm_str)
creates a pixrect from the xpm data string xpm_str.

cchar* W.create_pixrect(string xpm_file)
creates a pixrect from the xpm file xpm_file.

cchar* W.create_pixrect(int w, int h, unsigned char* bm_data, int fg = window::fgcol,
int bg = window::bgcol)
creates a pixrect of width w, height h, foreground color fg, and background color bg from bitmap data.

cchar* W.get_pixrect(double x1, double y1, double x2, double y2)
creates a color pixrect of width w = x2 - x1, height h = y2 - y1, and copies all pixels from the rectangular area
(x1, x2, y1, y2) of W into it.

cchar* W.get_window_pixrect()
creates a pixrect copy of the current window contents.

int W.get_width(char* pr) returns the width (number of pixels in a row) of pixrect pr.
int W.get_height(char *pr)
  returns the height (number of pixels in a column) of pixrect pr.

void W.put_pixrect(double x, double y, char *pr)
  copies the contents of pixrect pr with lower left corner at
  position (x, y) into W.

void W.put_pixrect(point p, char *pr)
  copies the contents of pixrect pr with lower left corner at
  position p into W.

void W.center_pixrect(double x, double y, char *pr)
  copies the contents of pixrect pr into W such that its center
  lies on position (x, y).

void W.put_pixrect(char *pr)
  copies pixrect pr with lower left corner at position
  (W.xmin(), W.ymin()) into W.

void W.set_pixrect(char *pr)
  copies pixrect pr with upper left corner at position (0, 0)
  into W.

void W.put_bitmap(double x, double y, char *bm, color c = window::fgcol)
  draws all pixels corresponding to 1-bits in bm with color
  c, here the lower left corner of bm corresponds to the pixel
  at position (x, y) in W.

void W.put_pixrect(double x, double y, char *pr, int x0, int y0, int w, int h)
  copies (pixel) rectangle (x0, y0, x0 + w, y0 + h) of pr with
  lower left corner at position (x, y) into W.

void W.del_bitmap(char *bm)
  destroys bitmap bm.

void W.del_pixrect(char *pr)
  destroys pixrect pr.

void W.copy_rect(double x0, double y0, double x1, double y1, double x, double y)
  copies all pixels of rectangle (x0, y0, x1, y1) into the rectan-
  gle (x, y, x + w, y + h), where w = x1 - x0 and h = y1 - y0.
void W.screenshot(string fname, bool full_color = true)
creates a screenshot of the current window. On unix systems suffix .ps is appended to fname and the output format is postscript. On windows systems the suffix .wmf is added and the format is windows metafile. If the flag full_color is set to false colors will be translated into grey scales.

3.4.15 Buffering

void W.start_buffering() starts buffering mode for W. If W has no associated buffer a buffer pixrect buf of the same size as the current drawing area of W is created. All subsequent drawing operations draw into buf instead of W until buffering mode is ended by calling W.stop_buffering().

void W.flush_buffer() copies the contents of the buffer pixrect into the drawing area of W.

void W.flush_buffer(double dx, double dy)
copies the contents of the buffer pixrect translated by vector (dx, dy) into the drawing area of W.

void W.flush_buffer(double x0, double y0, double x1, double y1)
copies the contents of rectangle (x0, y0, x1, y1) of the buffer pixrect into the corresponding rectangle of the drawing area.

void W.flush_buffer(double dx, double dy, double x0, double y0, double x1, double y1)
copies the contents of rectangle (x0, y0, x1, y1) of the buffer pixrect into the corresponding rectangle of the drawing area translated by vector (dx, dy).

void W.stop_buffering() ends buffering mode.

void W.stop_buffering(char * & prect)
ends buffering mode and returns the current buffer pixrect in prect.

3.4.16 Clipping

void W.set_clip_rectangle(double x0, double y0, double x1, double y1)
sets the clipping region of W to rectangle (x0, y0, x1, y1).

void W.set_clip_ellipse(double x0, double y0, double r1, double r2)
sets the clipping region of W to ellipse with center (x, y) and horizontal radius r1 and vertical radius r2.

void W.reset_clipping() restores the clipping region to the entire drawing area of W.
3.5 Input

The main input operation for reading positions, mouse clicks, and buttons from a window \( W \) is the operation \( W\).read_mouse(). This operation is blocking, i.e., waits for a button to be pressed which is either a “real” button on the mouse device pressed inside the drawing area of \( W \) or a button in the panel section of \( W \). In both cases, the number of the selected button is returned. Mouse buttons have pre-defined numbers \texttt{MOUSE\_BUTTON(1)} for the left button, \texttt{MOUSE\_BUTTON(2)} for the middle button, and \texttt{MOUSE\_BUTTON(3)} for the right button. The numbers of the panel buttons can be defined by the user. If the selected button has an associated action function or sub-window this function/window is executed/opened (cf. 18.2 for details).

There is also a non-blocking version \( W\).get_mouse() which returns the constant \texttt{NO\_BUTTON} if no button was pressed.

The window data type also provides two more general input operations \( W\).read_event() and \( W\).get_event() for reading events. They return the event type (enumeration in \texttt{<LEDA/graphics/x\_window.h>}), the value of the event, the position of the event in the drawing section, and a time stamp of the event.

3.5.1 Read Mouse

\begin{verbatim}
int W.read_mouse() 
  waits until a mouse button is pressed inside of the drawing area or until a button of the panel section is selected. In both cases, the number \( n \) of the button is returned which is one of the predefined constants \texttt{MOUSE\_BUTTON(i)} with \( i \in \{1,2,3\} \) for mouse buttons and a user defined value (defined when adding the button with \( W\).button()) for panel buttons. If the button has an associated action function this function is called with parameter \( n \). If the button has an associated window \( M \) it is opened and \( M\).read_mouse() is returned.

int W.read_mouse(double& x, double& y) 
  If a button is pressed inside the drawing area the current position of the cursor is assigned to \((x,y)\). The operation returns the number of the pressed button (see \( W\).read_mouse()).

int W.read_mouse(point& p) 
  as above, the current position is assigned to point \( p \).

int W.read_mouse_seg(double x0, double y0, double& x, double& y) 
  displays a line segment from \((x_0,y_0)\) to the current cursor position until a mouse button is pressed inside the drawing section of \( W \). When a button is pressed the current position is assigned to \((x,y)\) and the number of the pressed button is returned.
\end{verbatim}
int W.read_mouse_segment(const point& p, point& q)  
as above with \( x_0 = p.xcoord() \) and \( y_0 = p.ycoord() \) and  
the current position is assigned to \( q \).

int W.read_mouse_line(double \( x_0 \), double \( y_0 \), double& \( x \), double& \( y \))  
displays a line passing through \( (x_0, y_0) \) and the current  
cursor position until a mouse button is pressed inside the  
drawing section of \( W \). When a button is pressed the cur-
rent position is assigned to \( (x, y) \) and the number of the  
pressed button is returned.

int W.read_mouse_line(const point& p, point& q)  
as above with \( x_0 = p.xcoord() \) and \( y_0 = p.ycoord() \) and  
the current position is assigned to \( q \).

int W.read_mouse_ray(double \( x_0 \), double \( y_0 \), double& \( x \), double& \( y \))  
displays a ray from \( (x_0, y_0) \) passing through the current  
cursor position until a mouse button is pressed inside the  
drawing section of \( W \). When a button is pressed the cur-
rent position is assigned to \( (x, y) \) and the number of the  
pressed button is returned.

int W.read_mouse_ray(const point& p, point& q)  
as above with \( x_0 = p.xcoord() \) and \( y_0 = p.ycoord() \) and  
the current position is assigned to \( q \).

int W.read_mouse_rectangle(double \( x_0 \), double \( y_0 \), double& \( x \), double& \( y \))  
displays a rectangle with diagonal from \( (x_0, y_0) \) to the cur-
rent cursor position until a mouse button is pressed inside  
the drawing section of \( W \). When a button is pressed the  
current position is assigned to \( (x, y) \) and the number of  
the pressed button is returned.

int W.read_mouse_rectangle(const point& p, point& q)  
as above with \( x_0 = p.xcoord() \) and \( y_0 = p.ycoord() \) and  
the current position is assigned to \( q \).

int W.read_mouse_circle(double \( x_0 \), double \( y_0 \), double& \( x \), double& \( y \))  
displays a circle with center \( (x_0, y_0) \) passing through the  
current cursor position until a mouse button is pressed  
inside the drawing section of \( W \). When a button is pressed  
the current position is assigned to \( (x, y) \) and the number  
of the pressed button is returned.

int W.read_mouse_circle(const point& p, point& q)  
as above with \( x_0 = p.xcoord() \) and \( y_0 = p.ycoord() \) and  
the current position is assigned to \( q \).
3.5.2 Events

int int W.read_event(int& val, double& x, double& y, unsigned long& t)
waits for next event in window W and returns it. Assigns the button or key to val, the position in W to (x, y), and the time stamp of the event to t. Possible events are (cf. `<LEDA/graphics/x_window.h>`): key_press_event, key_release_event, button_press_event, button_release_event, configure_event, motion_event, destroy_event.

int int W.read_event(int& val, double& x, double& y, unsigned long& t, int timeout)
as above, but waits only timeout milliseconds; if no event occurred the special event no_event is returned.
18.2. WINDOWS (WINDOW)

\[
\begin{align*}
\text{int} & \quad W.\text{read\_event}(\text{int} & \text{ val, double} & \text{ x, double} & \text{ y}) \\
& \quad \text{waits for next event in window W and returns it. Assigns the button or key to val and the position in W to } (x, y). \\
\text{int} & \quad W.\text{read\_event}() \\
& \quad \text{waits for next event in window W and returns it.} \\
\text{int} & \quad W.\text{get\_event}(\text{int} & \text{ val, double} & \text{ x, double} & \text{ y}) \\
& \quad \text{if there is an event for window W in the event queue a } W.\text{read\_event} \text{ operation is performed, otherwise the integer constant } \text{no\_event} \text{ is returned.} \\
\text{bool} & \quad W.\text{shift\_key\_down}() \\
& \quad \text{returns true if a shift key was pressed during the last handled mouse button event.} \\
\text{bool} & \quad W.\text{ctrl\_key\_down}() \\
& \quad \text{returns true if a ctrl key was pressed during the last handled mouse button event.} \\
\text{bool} & \quad W.\text{alt\_key\_down}() \\
& \quad \text{returns true if an alt key was pressed during the last handled mouse button event.} \\
\text{int} & \quad W.\text{button\_press\_time}() \\
& \quad \text{returns the time-stamp (in msec) of the last button press event.} \\
\text{int} & \quad W.\text{button\_release\_time}() \\
& \quad \text{returns the time-stamp (in msec) of the last button release event.} \\
\end{align*}
\]

3.6 Panel Input

The operations listed in this section are useful for simple input of strings, numbers, and Boolean values.

\[
\begin{align*}
\text{bool} & \quad W.\text{confirm}(\text{string s}) \\
& \quad \text{displays string s and asks for confirmation. Returns true iff the answer was “yes”}. \\
\text{void} & \quad W.\text{acknowledge}(\text{string s}) \\
& \quad \text{displays string s and asks for acknowledgement.} \\
\text{int} & \quad W.\text{read\_panel}(\text{string h, int n, string* S}) \\
& \quad \text{displays a panel with header h and an array of n buttons with labels } S[0..n - 1], \text{ returns the index of the selected button.} \\
\text{int} & \quad W.\text{read\_vpanel}(\text{string h, int n, string* S}) \\
& \quad \text{like read\_panel with vertical button layout.} \\
\text{string} & \quad W.\text{read\_string}(\text{string p}) \\
& \quad \text{displays a panel with prompt p for string input, returns the input.} \\
\text{double} & \quad W.\text{read\_real}(\text{string p}) \\
& \quad \text{displays a panel with prompt p for double input returns the input.}
\end{align*}
\]
\[ \text{int } W.\text{read}_\text{int}(\text{string } p) \quad \text{displays a panel with prompt } p \text{ for integer input, returns the input.} \]

### 3.7 Input and output operators

For input and output of basic geometric objects in the plane such as points, lines, line segments, circles, and polygons the \(<<\) and \(>>\) operators can be used. Similar to C++ input streams windows have an internal state indicating whether there is more input to read or not. Its initial value is true and it is turned to false if an input sequence is terminated by clicking the right mouse button (similar to ending stream input by the \text{eof} character). In conditional statements, objects of type \textit{window} are automatically converted to boolean by returning this internal state. Thus, they can be used in conditional statements in the same way as C++ input streams. For example, to read a sequence of points terminated by a right button click, use \texttt{while (W >> p) \{ \ldots \}}.  

#### 3.7.1 Output

\begin{align*}
\text{window& } & W \ll \text{const point& } p \quad \text{like } W.\text{draw}_\text{point}(p). \\
\text{window& } & W \ll \text{const segment& } s \\
& \quad \text{like } W.\text{draw}_\text{segment}(s). \\
\text{window& } & W \ll \text{const ray& } r \quad \text{like } W.\text{draw}_\text{ray}(r). \\
\text{window& } & W \ll \text{const line& } l \quad \text{like } W.\text{draw}_\text{line}(l). \\
\text{window& } & W \ll \text{const circle& } C \quad \text{like } W.\text{draw}_\text{circle}(C). \\
\text{window& } & W \ll \text{const polygon& } P \\
& \quad \text{like } W.\text{draw}_\text{polygon}(P). \\
\text{window& } & W \ll \text{const gen_polygon& } P \\
& \quad \text{like } W.\text{draw}_\text{polygon}(P). \\
\text{window& } & W \ll \text{const rectangle& } R \\
& \quad \text{like } W.\text{draw}_\text{rectangle}(R). \\
\text{window& } & W \ll \text{const triangle& } T \\
& \quad \text{like } W.\text{draw}_\text{triangle}(T). \\
\end{align*}

#### 3.7.2 Input

\begin{align*}
\text{window& } & W \gg \text{point& } p \quad \text{reads a point } p: \text{ clicking the left button assigns the current cursor position to } p.
\end{align*}
18.2. WINDOWS (WINDOW)

\[ \text{window\\&} \quad W \gg segment\\& s \]
reads a segment \( s \): use the left button to define the start and end point of \( s \).

\[ \text{window\\&} \quad W \gg ray\\& r \]
reads a ray \( r \): use the left button to define the start point and a second point on \( r \).

\[ \text{window\\&} \quad W \gg line\\& l \]
reads a line \( l \): use the left button to define two different points on \( l \).

\[ \text{window\\&} \quad W \gg circle\\& C \]
reads a circle \( C \): use the left button to define the center of \( C \) and a point on \( C \).

\[ \text{window\\&} \quad W \gg rectangle\\& R \]
reads a rectangle \( R \): use the left button to define two opposite corners of \( R \).

\[ \text{window\\&} \quad W \gg triangle\\& T \]
reads a triangle \( T \): use the left button to define the corners of \( T \).

\[ \text{window\\&} \quad W \gg polygon\\& P \]
reads a polygon \( P \): use the left button to define the sequence of vertices of \( P \), end the sequence by clicking the right button.

\[ \text{window\\&} \quad W \gg gen\_polygon\\& P \]
reads a generalized polygon \( P \); input the polygons defining \( P \) and end the input by clicking the middle button.

\[ \text{list<point>} \quad W.read\_polygon() \]
as above, however, returns list of vertices.

As long as an input operation has not been completed the last read point can be erased by simultaneously pressing the shift key and the left mouse button.

3.8 Non-Member Functions

\[ \text{int} \quad \text{read\_mouse}(\text{window} \ast \& w, \text{double} \& x, \text{double} \& y) \]
waits for mouse input, assigns a pointer to the corresponding window to \( w \) and the position in \(*w\) to \((x, y)\) and returns the pressed button.

\[ \text{int} \quad \text{get\_mouse}(\text{window} \ast \& w, \text{double} \& x, \text{double} \& y) \]
non-blocking variant of \text{read\_mouse}; returns NO\_BUTTON if no button was pressed.

\[ \text{void} \quad \text{put\_back\_event}() \]
puts last handled event back to the event queue.

3.9 Panel Operations

The panel section of a window is used for displaying text messages and for updating the values of variables. It consists of a list of panel items and a list of buttons. The operations
in this section add panel items or buttons to the panel section of $W$. Note that they have to be called before the window is displayed the first time.

In general, a panel item consists of a string label and an associated variable of a certain type (int, bool, string, double, color). The value of this variable can be manipulated through the item. Each button has a label (displayed on the button) and an associated number. The number of a button is either defined by the user or is the rank of the button in the list of all buttons. If a button is pressed (i.e. selected by a mouse click) during a read_mouse operation its number is returned.

Action functions can be associated with buttons and some items (e.g. slider items) whenever a button with an associated action function is pressed this function is called with the number of the button as actual parameter. Action functions of items are called whenever the value of the corresponding variable is changed with the new value as actual parameter. All action functions must have the type $\text{void \ func(int)}$.

Another way to define a button is to associate another window with it. In this case the button will have a menu sign and as soon as it is pressed the attached window will open. This method can be used to implement pop-up menus. The return value of the current read_mouse operation will be the number associated with the button in the menu.

### 3.9.1 General Settings

- **void** $W$.set_panel_bg_color($\text{color c}$)
  
  sets the background color of the panel area to $\text{c}$.

- **void** $W$.buttons_per_line($\text{int n}$)
  
  defines the maximal number $\text{n}$ of buttons per line.

- **void** $W$.set_button_space($\text{int s}$)
  
  sets the space between to adjacent buttons to $\text{s}$ pixels.

- **void** $W$.set_item_height($\text{int h}$)
  
  sets the vertical size of all items to $\text{h}$ pixels.

- **void** $W$.set_item_width($\text{int w}$)
  
  sets the horizontal size of all slider and string items to $\text{w}$ pixels.

- **void** $W$.set_bitmap_colors($\text{int c}_0, \text{int c}_1$)
  
  sets the unpressed/pressed colors used for drawing the pixels in bitmap buttons to $\text{c}_0$ and $\text{c}_1$.

### 3.9.2 Simple Panel Items

- **panel_item** $W$.text_item($\text{string s}$)
  
  adds a text_item $\text{s}$ to $W$. 
18.2. WINDOWS ( WINDOW ) 655

panel_item  W.bool_item(string s, bool& x, const char * hlp = 0)

  adds a boolean item with label s and variable x to W.

panel_item  W.bool_item(string s, bool& x, void (*F)(int), const char * hlp = 0)

  as above with action function F.

panel_item  W.bool_item(string s, bool& x, const window_handler& obj,
                        const char * hlp = 0)

  as above with handler object obj.

panel_item  W.int_item(string s, int& x, const char * hlp = 0)

  adds an integer item with label s and variable x to W.

panel_item  W.string_item(string s, string& x, void (*F)(char*), const char * hlp = 0)

  as above with action function F.

panel_item  W.string_item(string s, string& x, const window_handler& obj,
                        const char * hlp = 0)

  as above with handler object obj.

panel_item  W.string_item(string s, string& x, const char * hlp = 0)

  adds a string item with label s and variable x to W.

panel_item  W.double_item(string s, double& x, const char * hlp = 0)

  adds a real item with label s and variable x to W.

panel_item  W.color_item(string s, color& x, const char * hlp = 0)

  adds a color item with label s and variable x to W.

panel_item  W.color_item(string s, color& x, void (*F)(int), const char * hlp = 0)

  as above with action function F.

panel_item  W.color_item(string s, color& x, const window_handler& obj,
                        const char * hlp = 0)

  as above with handler object obj.

panel_item  W.pstyle_item(string s, point_style& x, const char * hlp = 0)

  adds a point style item with label s and variable x to W.

panel_item  W.pstyle_item(string s, point_style& x, void (*F)(int), const char * hlp = 0)

  as above with action function F.

panel_item  W.pstyle_item(string s, point_style& x, const window_handler& obj,
                        const char * hlp = 0)

  as above with handler object obj.
panel_item  W.lstyle_item(string s, line_style& x, const char * hlp = 0)
            adds a line style item with label s and variable x to W.
panel_item  W.lstyle_item(string s, line_style& x, void(*F)(int), const char * hlp = 0)
            as above with action function F.
panel_item  W.lstyle_item(string s, line_style& x, const window_handler& obj,
                          const char * hlp = 0)
            as above with handler object obj.
panel_item  W.lwidth_item(string s, int& x, const char * hlp = 0)
            adds a line width item with label s and variable x to W.
panel_item  W.lwidth_item(string s, int& x, void(*F)(int), const char * hlp = 0)
            as above with action function F.
panel_item  W.lwidth_item(string s, int& x, const window_handler& obj,
                          const char * hlp = 0)
            as above with handler object obj.

3.9.3 Integer Choice Items

panel_item  W.int_item(string s, int& x, int l, int h, int step, const char * hlp = 0)
            adds an integer choice item with label s, variable x, range l,...,h, and step size step to W.
panel_item  W.int_item(string s, int& x, int l, int h, int step, void (*F)(int),
                          const char * hlp = 0)
            adds an integer choice item with label s, variable x, range l,...,h, and step size step to W. Function F(x) is executed whenever the value of x is changed.
panel_item  W.int_item(string s, int& x, int l, int h, int step,
                          const window_handler& obj, const char * hlp = 0)
            as above with handler object obj.
panel_item  W.int_item(string s, int& x, int l, int h, const char * hlp = 0)
            adds an integer slider item with label s, variable x, and range l,...,h to W.
panel_item  W.int_item(string s, int& x, int l, int h, void (*F)(int),
                          const char * hlp = 0)
            adds an integer slider item with label s, variable x, and range l,...,h to W. Function F(x) is executed whenever the value of x has changed by moving the slider.
3.9.4 String Menu Items

W.string_item(string s, string& x, const list<string>& L, const char* hlp = 0)

adds a string item with label s, variable x, and menu L to W.

W.string_item(string s, string& x, const list<string>& L, const window_handler& obj, const char* hlp = 0)

as above with handler object obj.

W.string_item(string s, string& x, const list<string>& L, int sz, const char* hlp = 0)

menu L is displayed in a scroll box of height sz.

W.string_item(string s, string& x, const list<string>& L, int sz, void (*F)(int), const char* hlp = 0)

as above with action function F.

W.string_item(string s, string& x, const list<string>& L, int sz, const window_handler& obj, const char* hlp = 0)

as above with handler object obj.

W.set_menu(panel_item it, const list<string>& L, int sz = 0)

replaces the menu of string menu item it by a menu for list L (table style if sz = 0 and scroll box with sz entries otherwise).

3.9.5 Choice Items

W.choice_item(string s, int& x, const list<string>& L, void (*F)(int) = 0, const char* hlp = 0)

adds an integer item with label s, variable x, and choices from L to W.

W.choice_item(string s, int& x, const list<string>& L, const window_handler& obj, const char* hlp = 0)

as above with handler object obj.

W.choice_item(string s, int& x, string s1, ... , string sk)

adds an integer item with label s, variable x, and choices s1, ..., sk to W (k ≤ 8).
panel_item  W.choice_item(string s, int& x, int n, int w, int h, unsigned char **bm, 
      const char * hlp = 0)
      adds an integer item with label s, variable x, and 
      n bitmaps bm[0], ..., bm[n - 1] each of width w 
      and height h.
panel_item  W.choice_item(string s, int& x, int n, int w, int h, unsigned char **bm, 
      void (*F)(int), const char * hlp = 0)
panel_item  W.choice_item(string s, int& x, int n, int w, int h, unsigned char **bm, 
      const window_handler& obj, const char * hlp = 0)
      as above with handler object obj.

3.9.6 Multiple Choice Items

panel_item  W.choice_mult_item(string s, int& x, const list<string>& L, 
      const char * hlp = 0)
panel_item  W.choice_mult_item(string s, int& x, const list<string>& L, 
      void (*F)(int), const char * hlp = 0)
panel_item  W.choice_mult_item(string s, int& x, const list<string>& L, 
      const window_handler& obj, const char * hlp = 0)
panel_item  W.choice_mult_item(string s, int& x, int n, int w, int h, 
      unsigned char **bm, const char * hlp = 0)
panel_item  W.choice_mult_item(string s, int& x, int n, int w, int h, 
      unsigned char **bm, void (*F)(int), 
      const char * hlp = 0)
panel_item  W.choice_mult_item(string s, int& x, int n, int w, int h, 
      unsigned char **bm, const window_handler& obj, 
      const char * hlp = 0)

3.9.7 Buttons

The first occurrence of character '&' in a button label makes the following character c an
accelerator character, i.e., the button can be selected by typing ALT-c from the keyboard.

int  W.button(string s, int n, const char * hlp = 0)
      adds a button with label s and number n to W.
int  W.fbutton(string s, int n, const char * hlp = 0)
      as above but makes this button the focus button of 
      W, i.e., this button can be selected by pressing the 
      return key.
18.2. WINDOWS ( WINDOW )

```c
int W.button(string s, const char * hlp = 0)
    adds a new button to W with label s and number equal to its position in the list of all buttons (starting with 0).

int W.fbutton(string s, const char * hlp = 0)
    as above but makes this button the focus button.

int W.button(int w, int h, unsigned char * bm, string s, int n,
             const char * hlp = 0)
    adds a button with bitmap bm, label s, and number n to W.

int W.button(char * pr1, char * pr2, string s, int n, const char * hlp = 0)
    adds a button with pixrects pr1 and pr2, label s, and number n to W.

int W.button(int w, int h, unsigned char * bm, string s, const char * hlp = 0)
    adds a new button to W with bitmap bm, label s, and number equal to its position in the list of all buttons (starting with 0).

int W.button(string s, int n, void (*F)(int), const char * hlp = 0)
    adds a button with label s, number n and action function F to W. Function F is called with actual parameter n whenever the button is pressed.

int W.button(string s, int n, const window_handler & obj,
             const char * hlp = 0)
    as above with handler object obj.

int W.fbutton(string s, int n, void (*F)(int), const char * hlp = 0)
    as above but makes this button the focus button.

int W.fbutton(string s, int n, const window_handler & obj,
              const char * hlp = 0)
    as above with handler object obj.

int W.button(int w, int h, unsigned char * bm, string s, int n,
             void (*F)(int), const char * hlp = 0)
    adds a button with bitmap bm, label s, number n and action function F to W. Function F is called with actual parameter n whenever the button is pressed.

int W.button(int w, int h, unsigned char * bm, string s, int n,
             const window_handler & obj, const char * hlp = 0)
```
int W.button(char *pr1, char *pr2, string s, int n, void (*F)(int),
const char *hlp = 0)
as above, but with pixrect pr1 and pr2.

int W.button(char *pr1, char *pr2, string s, int n,
const window_handler& obj, const char *hlp = 0)

int W.button(string s, void (*F)(int), const char *hlp = 0)
adds a button with label s, number equal to its rank and action function F to W. Function F is called with the value of the button as argument whenever the button is pressed.

int W.button(string s, const window_handler& obj, const char *hlp = 0)

int W.button(int w, int h, unsigned char *bm, string s, void (*F)(int),
const char *hlp = 0)
adds a button with bitmap bm, label s, number equal to its rank and action function F to W. Function F is called with the value of the button as argument whenever the button is pressed.

int W.button(int w, int h, unsigned char *bm, string s, int n,
const window_handler& obj, const char *hlp = 0)

int W.button(char *pr1, char *pr2, string s, void (*F)(int),
const char *hlp = 0)
as above, but with pixrect pr1 and pr2.

int W.button(char *pr1, char *pr2, string s, const window_handler& obj,
const char *hlp = 0)

int W.button(string s, int n, window& M, const char *hlp = 0)
adds a button with label s, number n and attached sub-window (menu) M to W. Window M is opened whenever the button is pressed.

int W.button(int w, int h, unsigned char *bm, string s, int n, window& M,
const char *hlp = 0)
adds a button with bitmap bm, label s, number n and attached sub-window (menu) M to W. Window M is opened whenever the button is pressed.

int W.button(char *pr1, char *pr2, string s, int n, window& M,
const char *hlp = 0)
as above, but with pixrect pr1 and pr2.
18.2. WINDOWS ( WINDOW )

`int W.button(string s, window& M, const char * hlp = 0)`

adds a button with label `s` and attached sub-window `M` to `W`. The number returned by `read_mouse` is the number of the button selected in sub-window `M`.

`int W.button(int w, int h, unsigned char * bm, string s, window& M, const char * hlp = 0)`

adds a button with bitmap `bm`, label `s` and attached sub-window `M` to `W`. The number returned by `read_mouse` is the number of the button selected in sub-window `M`.

`int W.button(char * pr1, char * pr2, string s, window& M, const char * hlp = 0)`

as above, but with pixrect `pr1` and `pr2`.

`void W.make_menu_bar()`

inserts a menu bar at the top of the panel section that contains all previously added menu buttons (buttons with a subwindow attached).

`window* window::get_call_window()`

A static function that can be called in action functions attached to panel items or buttons to retrieve a pointer to the window containing the corresponding item or button.

`panel_item window::get_call_item()`

A static function that can be called in action functions attached to panel items to retrieve the corresponding item.

`int window::get_call_button()`

A static function that can be called in action functions attached to panel buttons to retrieve the number of the corresponding button.

3.9.8. Manipulating Panel Items and Buttons

Disabling and Enabling Items or buttons

`void W.disable_item(panel_item it)`

disables panel item `it`.

`void W.enable_item(panel_item it)`

enables panel item `it`.

`bool W.is_enabled(panel_item it)`

tests whether item `it` is enabled or not.
void W.disable_button(int b) disables button b.

void W.enable_button(int b) enables button b.

bool W.is_enabled(int b) tests whether button b is enabled or not.

void W.disable_panel(bool disable_every_item = true)
disables the entire panel section of W.

void W.enable_panel() enables the entire panel section of W.

Accessing and Updating Item Data

void W.set_text(panel_item it, string s)
replaces the text of text item it by s.

panel_item W.get_item(string s) returns the item with label s and NULL if no such item exists in W.

int W.get_button(string s) returns the button with label s and −1 if no such button exists in W.

string W.get_button_label(int but) returns the label of button but.

void W.set_button_label(int but, string s) sets the label of button but to s.

void W.set_button_pixrects(int but, char *pr1, char *pr2) sets the pixrects of button but to pr1 and pr2.

window* W.get_window(int but) returns a pointer to the subwindow attached to button but (NULL if but has no subwindow)

window* W.set_window(int but, window *M) associates subwindow (menu) *M with button but.
Returns a pointer to the window previously attached to but.

void W.set_function(int but, void (*F)(int)) assign action function F to button but.

void W.set_object(int but, const window_handler& obj) assign handler object obj to button but.

3.9.9. Miscellaneous

void W.redraw_panel() redraw the panel area of W.
void W.redraw_panel(panel_item it)
       redraw item i in the panel area of W.

void W.display_help_text(string fname)
       displays the help text contained in name.hlp. The file name.hlp must exist either in the current working directory or in $LEDAROOT/incl/Help.

void W.set_tooltip(int i, double x0, double y0, double x1, double y1, string txt)
        inserts a tooltip with id i, rectangle (x0, y0, x1, y1) and text txt into the window. The text is shown when the mouse pointer enters the rectangle. The text disappears as soon as the mouse pointer leaves the rectangle.
        CAUTION: Currently the method has to be called after the call of W.display(). Setting a tooltip before the call W.display() has no effect.

void W.del_tooltip(int i) removes the tooltip with id i.

4. Example

Example programs can be found on LEDAROOT/demo/win and LEDAROOT/test/win.
18.3 Panels (panel)

1. Definition
Panels are windows consisting of a panel section only (cf. section 18.2). They are used for displaying text messages and updating the values of variables.

```c
#include <LEDA/graphics/panel.h>
```

2. Creation

```c
panel P; creates an empty panel P.

panel P(string s); creates an empty panel P with header s.

panel P(int w, int h); creates an empty panel P of width w and height h.

panel P(int w, int h, string s); creates an empty panel P of width w and height h with header s.
```

3. Operations

All window operations for displaying, reading, closing and adding panel items are available (see section 18.2). There are two additional operations for opening and reading panels.

```c
int P.open(int x = window::center, int y = window::center) 
    P.display(x, y) + P.read_mouse() + P.close().

int P.open(window& W, int x = window::center, int y = window::center) 
    P.display(W, x, y) + P.read_mouse() + P.close().
```
18.4 Menues (menu)

1. Definition

Menues are special panels consisting only of a vertical list of buttons.

```c
#include <LEDA/graphics/menu.h>
```

2. Creation

```c
menu M; creates an empty menu M.
```

3. Operations

- `int M.button(string s, int n)` adds a button with label `s` and number `n` to `M`.
- `int M.button(string s)` adds a new button to `M` with label `s` and number equal to its position in the list of all buttons (starting with 0).
- `int M.button(string s, int n, void (*F)(int))` adds a button with label `s`, number `n` and action function `F` to `M`. Function `F` is called with actual parameter `n` whenever the button is pressed.
- `int M.button(string s, int n, const window_handler& obj)` as above with handler object `obj`.
- `int M.button(string s, void (*F)(int))` adds a button with label `s`, number equal to its rank and action function `F` to `M`. Function `F` is called with the number of the button as argument whenever the button is pressed.
- `int M.button(string s, const window_handler& obj)` as above with handler object `obj`.
- `int M.button(string s, int n, window& W)` adds a button with label `s`, number `n`, and attached window `W` to `M`. Whenever the button is pressed `W` is opened.
- `int M.button(string s, window& W)` adds a button with label `s` and attached window `W` to `M`. Whenever the button is pressed `W` is opened and `W.read_mouse()` is returned.
- `void M.separator()` inserts a separator (horizontal line) at the current position.
int \ M.\text{open}(\text{window}\& \ W, \ \text{int} \ x, \ \text{int} \ y)

open and read menu \ M \ at \ position \ (x, y) \ in \ window \ W.
18.5 Postscript Files (ps_file)

1. Definition

The data type ps_file is a graphical input/output interface for the familiar LEDA drawing operations of two-dimensional geometry. Unlike the data type window, the output produced by a ps_file object is permanent, i.e., it is not lost after exiting the C++-program as it is saved in an output file.

An instance of type ps_file is (as far as the user takes notice of it) an ordinary ASCII file that contains the source code of the graphics output in the PostScript description language. After running the C++-program, the file is created in the user’s current working directory and can later be handled like any other PostScript file, i.e., it may be viewed, printed etc.

Of course, features like a panel section (as in window type instances) don’t make sense for a representation that is not supposed to be displayed on the screen and interactively worked with by the user. Therefore, only drawing operations are applicable to a ps_file instance.

ps_file was implemented by

Thomas Wahl
Lehrstuhl für Informatik I
Universität Würzburg

The complete user manual can be found in LEDAROOT/Manual/contrib.

#include <LEDAgraphics/ps_file.h>
18.6 Graph Windows (GraphWin)

1. Definition

GraphWin combines the two types graph and window and forms a bridge between the graph data types and algorithms and the graphics interface of LEDA. GraphWin can easily be used in LEDA programs for constructing, displaying and manipulating graph and for animating and debugging graph algorithms.

- The user interface of GraphWin is simple and intuitive. When clicking a mouse button inside the drawing area a corresponding default action is performed that can be redefined by users. With the initial default settings, the left mouse button is used for creating and moving objects, the middle button for selecting objects, and the right button for destroying objects. A number of menus at the top of the window give access to graph generators, modifiers, basic algorithms, embeddings, setup panels, and file input and output.

- GraphWin can display and manipulate the data associated with the nodes and edges of LEDA’s parameterized graph type \( \text{GRAPH} < \text{vtype}, \text{etype} > \). When a GraphWin is opened for such a graph the associated node and edge labels of type \( \text{vtype} \) and \( \text{etype} \) can be displayed and edited.

- Most of the actions of GraphWin can be customized by modifying or extending the menus of the main window or by defining call-back functions. So the user can define what happens if a node or edge is created, selected, moved, or deleted.

- GraphWin offers a collection of graph generators, modifiers and tests. The generators include functions for constructing random, planar, complete, bipartite, grid graph, connected graph, biconnected, graph …

There are also methods for modifying existing graph (e.g. by removing or adding a certain set of edges) to fit in one of these categories and for testing whether a given graph is planar, connected, bipartite …

- The standard menu includes a choice of fundamental graph algorithms and basic embedding algorithms.

For every node and edge of the graph GraphWin maintains a set of parameters.

With every node is associated the following list of parameters. Note that for every parameter there are corresponding set and get operations (\texttt{gw.set\_param()} and \texttt{gw.get\_param}) where \texttt{param} has to be replaced by the corresponding parameter name.

\textbf{position}: the position of the node (type \texttt{point}),
shape: the shape of the node (type `gw_node_shape`),
color: the color of the interior of the node (type `color`),
border_color: the color of the node’s border (type `color`),
label_color: the color of the node’s label (type `color`),
pixmap: the pixmap used to fill the interior of the node (`char*`),
width: the width of the node in pixels (`int`),
height: the height of the node in pixels (`int`),
radius1: the horizontal radius in real world coordinates (`double`)
radius2: the vertical radius in real world coordinates (`double`),
border_width: the width of the border in pixels (`int`),
label_type: the type of the node’s label (type `gw_label_type`),
user_label: the user label of the node (type `string`), and
label_pos: the position of the label (type `gw_position`).

With every edge is associated the following list of parameters

color: the color of the edge (type `color`),
label_color: the color of the edge label (type `color`),
shape: the shape of the edge (type `gw_edge_shape`),
style: the style of the edge (type `gw_edge_style`),
direction: the direction of the edge (type `gw_edge_dir`),
width: the width of the edge in pixels (type `int`),
label_type: the label type of the edge (type `gw_label_type`),
user_label: the user label of the edge (type `string`),
label_pos: the position of the edge’s label (type `gw_position`),
bends: the list of edge bends (type `list<point>`),
source_anchor: the source anchor of the edge (type `point`), and
target_anchor: the target anchor of the edge (type `point`).
The corresponding types are:

\[
\text{gw\_node\_shape} = \{ \text{circle\_node, ellipse\_node, square\_node, rectangle\_node} \} \\
\text{gw\_edge\_shape} = \{ \text{poly\_edge, circle\_edge, bezier\_edge, spline\_edge} \} \\
\text{gw\_position} = \{ \text{central\_pos, northwest\_pos, north\_pos, northeast\_pos, east\_pos, southeast\_pos, south\_pos, southwest\_pos, west\_pos} \} \\
\text{gw\_label\_type} = \{ \text{no\_label, user\_label, data\_label, index\_label} \} \\
\text{gw\_edge\_style} = \{ \text{solid\_edge, dashed\_edge, dotted\_edge, dashed\_dotted\_edge} \} \\
\text{gw\_edge\_dir} = \{ \text{undirected\_edge, directed\_edge, bidirected\_edge, rdirected\_edge} \};
\]

```c
#include <LEDA/graphics/graphwin.h>
```

2. Creation

\[
\text{GraphWin gw(graph\& G, int w, int h, const char * win\_label = "")};
\]

creates a graph window for graph G with a display window of size \(w\) pixels \(\times h\) pixels. If \(win\_label\) is not empty it is used as the frame label of the window, otherwise, a default frame label is used.

\[
\text{GraphWin gw(graph\& G, const char * win\_label = "")};
\]

creates a graph window for graph G with a display window of default size and frame label \(win\_label\).

\[
\text{GraphWin gw(int w, int h, const char * win\_label = "")};
\]

creates a graph window for a new empty graph with a display window of size \(w\) pixels \(\times h\) pixels, and frame label \(win\_label\).

\[
\text{GraphWin gw(const char * win\_label = "")};
\]

creates a graph window for a new empty graph with a display window of default size and frame label \(win\_label\).

\[
\text{GraphWin gw(window\& W)};
\]

as above, but \(W\) is used as display window.

\[
\text{GraphWin gw(graph\& G, window\& W)};
\]

as above, but makes \(G\) the graph of \(gw\).

3. Operations

a) Window Operations
void gw.display(int x, int y) displays gw with upper left corner at \((x, y)\). The predefined constant \texttt{window::center} can be used to center the window horizontally (if passed as \(x\)) or vertically (if passed as \(y\)).

void gw.display() displays gw at default position.

bool gw.edit() enters the edit mode of \texttt{GraphWin} that allows to change the graph interactively by operations associated with certain mouse events or by choosing operations from the windows menu bar (see section about edit-mode for a description of the available commands and operations). Edit mode is terminated by either pressing the \texttt{done} button or by selecting \texttt{exit} from the file menu. In the first case the result of the edit operation is \texttt{true} and in the latter case the result is \texttt{false}.

bool gw.open(int x, int y) displays the window at position \((x, y)\), enters edit mode and return the corresponding result.

bool gw.open() as above, but displays the window at default position.

void gw.close() closes the window.

void gw.message(const char * msg) displays the message \(msg\) at the top of the window.

string gw.get_message() returns the current message string.

void gw.del_message() deletes a previously written message.

double gw.get_xmin() returns the minimal x-coordinate of the window.

double gw.get_ymin() returns the minimal y-coordinate of the window.

double gw.get_xmax() returns the maximal x-coordinate of the window.

double gw.get_ymax() returns the maximal y-coordinate of the window.

void gw.win_init(double xmin, double xmax, double ymin) sets the coordinates of the window to \((xmin, xmax, ymin)\).

void gw.redraw() redraws the graph. If the \texttt{flush} parameter of \(gw\) is set to false (see \texttt{set_flush}) this operation can be used to display the current state of the graph after a number of update operations.
void gw.set_frame_label(const char * label)
    makes label the frame label of the window.

int gw.open_panel(panel & P)
displays panel P centered on the drawing area of
gw, disables the menu bar of gw and returns the
result of P.open().

window& gw.get_window() returns a reference to the window of gw.

void gw.finish_menu_bar() this operation has to called before additional
buttons are added to the panel section of
gw.get_window().

b) Graph Operations

node gw.new_node(const point & p)
    adds a new node at position p to gw.

void gw.del_node(node v)
deletes v and all edges incident to v from gw.

edge gw.new_edge(node v, node w)
    adds a new edge (v, w) to gw.

edge gw.new_edge(node v, node w, const list<point> & P)
    adds a new edge (v, w) with bend sequence P to
gw.

void gw.del_edge(edge e)
deletes edge e from gw.

void gw.clear_graph()
deletes all nodes and egdes.

graph& gw.get_graph() returns a reference of the graph of gw.

void gw.update_graph() this operation has to be called after any up-
date operation that has been performed directly
(not by GraphWin) on the underlying graph, e.g.,
deleting or inserting nodes or edges.

c) Node Parameters

Node parameters can be retrieved or changed by a collection of get- and set- operations.
We use param_type for the type and param for the value of the corresponding parameter.

Individual Parameters

param_type gw.get_param(node v) returns the value of parameter param for node v.
18.6. GRAPH WINDOWS (GRAPHWIN)

\texttt{param\_type gw.set\_param(node v, param\_type x)}

sets the value of parameter \texttt{param} for node \texttt{v} to \texttt{x}. and returns its previous value.

\texttt{void gw.set\_param(list<node>& L, param\_type x)}

sets the value of parameter \texttt{param} for all nodes in \texttt{L} to \texttt{x}.

Default Parameters

\texttt{param\_type gw.get\_node\_param( )} returns the current default value of parameter \texttt{param}.

\texttt{param\_type gw.set\_node\_param(param\_type x, bool apply = true)}

sets the default value of parameter \texttt{param} to \texttt{x}. and returns its previous value. If \texttt{apply == true} the parameter is changed for all existing nodes as well.

d) Edge Parameters

Individual Parameters

\texttt{param\_type gw.get\_param(edge e)} returns the value of parameter \texttt{param} for edge \texttt{e}.

\texttt{param\_type gw.set\_param(edge e, param\_type x)}

sets the value of parameter \texttt{param} for edge \texttt{e} to \texttt{x}. and returns its previous value.

\texttt{void gw.set\_param(list<edge>& L, param\_type x)}

sets the value of parameter \texttt{param} for all edges in \texttt{L} to \texttt{x}.

Default Parameters

\texttt{param\_type gw.get\_edge\_param( )} returns the current default value of parameter \texttt{param}.

\texttt{param\_type gw.set\_edge\_param(param\_type x, bool apply = true)}

sets the default value of parameter \texttt{param} to \texttt{x}. and returns its previous value. If \texttt{apply == true} the parameter is changed for all existing edges as well.

e) Global Options

\texttt{int gw.set\_gen\_nodes(int n)} sets the default number of nodes \texttt{n} for all graph generator dialog panels.
\begin{verbatim}
int gw.set_gen_edges(int m) sets the default number of edges \textit{m} for all graph generator dialog panels.

int gw.set_edge_distance(int d) sets the distance of multi-edges to \textit{d} pixels.

grid_style gw.set_grid_style(grid_style \textit{s}) sets the grid style to \textit{s}.

int gw.set_grid_dist(int \textit{d}) sets the grid distance to \textit{d}.

int gw.set_grid_size(int \textit{n}) sets the grid distance such that \textit{n} vertical grid lines lie inside the drawin area.

bool gw.set_show_status(bool \textit{b}) display a status window (\textit{b}=true) or not (\textit{b}=false).

color gw.set_bg_color(color \textit{c}) sets the window background color to \textit{c}.

char* gw.set_bg_pixmap(char* \textit{pr}, double xorig = 0, double yorig = 0) sets the window background pixmap to \textit{pr} and the tiling origin to (xorig,yorig).

void gw.set_bg_xpm(const char* \textit{xpm_data}) sets the window background pixmap to the pixmap defined by \textit{xpm_data}.

void gw.set_bg_redraw(void (*f)(window*, double, double, double, double)) sets the window background redraw function to \textit{f}.

void gw.set_node_label_font(gw_font_type \textit{t}, int \textit{sz}) sets the node label font type and size. Possible types are roman_font, bold_font, italic_font, and fixed_font.

void gw.set_node_label_font(string \textit{fn}) sets the node label font to the font with name \textit{fn}.

void gw.set_edge_label_font(gw_font_type \textit{t}, int \textit{sz}) sets the edge label font type and size. roman_font, bold_font, italic_font, and fixed_font.

void gw.set_edge_label_font(string \textit{fn}) sets the edge label font to the font with name \textit{fn}.

string gw.set_node_index_format(string \textit{s}) sets the node index format string to \textit{s}.
\end{verbatim}
18.6. GRAPH WINDOWS (GRAPHWIN)

string gw.set_edge_index_format(string s)
sets the edge index format string s.

bool gw.set_edge_border(bool b)
sets the edge border flag to b.

bool gw.enable_label_box(bool b)
enables/disables drawing of blue label boxes. Label boxes are enabled per default.

Animation and Zooming

int gw.set_animation_steps(int s)
move a node in s steps to its new position.

bool gw.set_flush(bool b)
show operations on gw instantly (b=true) or not (b=false).

double gw.set_zoom_factor(double f)
sets the zoom factor to f used when zooming from menu.

bool gw.set_zoom_objects(bool b)
resize nodes and edges when zooming (b==true) or not (b==false).

bool gw.set_zoom_labels(bool b)
resize labels when zooming (b==true) or not (b==false).

f) Node and Edge Selections

void gw.select(node v) adds v to the list of selected nodes.

void gw.select_all_nodes() selects all nodes.

void gw.deselect(node v) deletes v from the list of selected nodes.

void gw.deselect_all_nodes() clears the current node selection.

bool gw.is_selected(node v) returns true if v is selected and false otherwise.

const list<node>& gw.get_selected_nodes() returns the current node selection.

void gw.select(edge e) adds e to the list of selected edges.

void gw.select_all_edges() selects all edges.
void gw.deselect(edge e) deletes e from the list of selected edges.
void gw.deselect_all_edges() clears the current node selection.
bool gw.is_selected(edge e) returns true if e is selected and false otherwise.

const list<edge>& gw.get_selected_edges() returns the current edge selection.
void gw.deselect_all() clears node and edge selections.

g) Layout Operations

void gw.set_position(const node_array<point>& pos)
for every node v of G the position of v is set to pos[v].

void gw.set_position(const node_array<double>& x,
const node_array<double>& y)
for every node v of G the position of v is set to (x[v], y[v]).

void gw.get_position(node_array<point>& pos)
for every node v of G the position of v is assigned to pos[v].

void gw.set_layout(const node_array<point>& pos,
const node_array<double>& r1,
const node_array<double>& r2, const edge_array<list<point>>& bends, const edge_array<point>& sanch,
const edge_array<point>& tanch)
for every node v the position is set to pos[v] and radius_i is set to r_i[v]. For every edge e the list of bends is set to bends[e] and source (target) anchor is set to sanch[e] (tanch[e]).

void gw.set_layout(const node_array<point>& pos, const edge_array<list<point>>& bends, bool reset_anchors = true)
for every node v the position is set to pos[v] and for every edge e the list of bends is set to bends[e].

void gw.set_layout(const node_array<point>& pos)
for every node v the position is set to pos[v] and for every edge e the list of bends is made empty.

void gw.set_layout(const node_array<double>& x, const node_array<double>& y)
for every node v the position is set to (x[v], y[v]) and for every edge e the list of bends is made empty.
void gw.set_layout( ) same as gw.remove_bends( ).

void gw.transform_layout(node_array<double>& xpos, node_array<double>& ypos, edge_array<list<double>> &xbends, edge_array<list<double>> &ybends, double dx, double dy, double fx, double fy)
transforms the layout given by xpos, ypos, xbends, and ybends by transforming every node position or edge bend \((x, y)\) to \((dx + fx*x, dy + fy*y)\). The actual layout of the current graph is not changed by this operation.

void gw.transform_layout(node_array<double>& xpos, node_array<double>& ypos, node_array<double>& xrad, node_array<double>& yrad, edge_array<list<double>> &xbends, edge_array<list<double>> &ybends, double dx, double dy, double fx, double fy)
as above, in addition the horizontal and vertical radius of every node (given in the arrays xrad and yrad) are enlarged by a factor of \(fx\) and \(fy\), respectively.

void gw.fill_win_params(double wx0, double wy0, double wx1, double wy1, double x0, double y0, double x1, double y1, double& dx, double& dy, double& fx, double& fy)
computes parameters \(dx\), \(dy\), \(fx\), and \(fy\) for transforming rectangle \(x0,y0,x1,y1\) into (window) rectangle \(wx0,wy0,wx1,wy1\).

void gw.fill_win_params(double wx0, double wy0, double wx1, double wy1, node_array<double>& xpos, node_array<double>& ypos, edge_array<list<double>> &xbends, edge_array<list<double>> &ybends, double& dx, double& dy, double& fx, double& fy)
computes parameters \(dx\), \(dy\), \(fx\), and \(fy\) for transforming the layout given \(xpos,ypos,xbends,ybends\) to fill the (window) rectangle \(wx0,wy0,wx1,wy1\).
### CHAPTER 18. GRAPHICS

```c
void gw.fill_win_params(double wx0, double wy0, double wx1, double wy1,
node_array<double>& xpos, node_array<double>& ypos,
node_array<double>& xrad, node_array<double>& yrad,
edge_array<list<double>> &xbends,
edge_array<list<double>> &ybends, double &dx, double &dy, double &fx, double &fy)
```

computes parameters dx, dy, fx, and fy for transforming the layout given xpos, ypos, xbends, ybends, xrad, yrad to fill the (window) rectangle wx0, wy0, wx1, wy1.

```c
void gw.place_into_box(double x0, double y0, double x1, double y1)
```

moves and stretches the graph to fill the given rectangular box (x0, y0, x1, y1) by appropriate scaling and translating operations.

```c
void gw.place_into_win()
```

moves and stretches the graph to fill the entire window by appropriate scaling and translating operations.

```c
void gw.adjust_coords_to_box(node_array<double>& xpos,
node_array<double>& ypos,
edge_array<list<double>> &xbends,
edge_array<list<double>> &ybends, double x0, double y0, double x1, double y1)
```

transforms the layout given by xpos, ypos, xbends, and ybends in such way as a call of `place_into_box(x0, y0, x1, y1)` would do. However, the actual layout of the current graph is not changed by this operation.

```c
void gw.adjust_coords_to_box(node_array<double>& xpos,
node_array<double>& ypos, double x0, double y0, double x1, double y1)
```

transforms the layout given by xpos, ypos in such way as a call of `place_into_box(x0, y0, x1, y1)` would do ignoring any edge bends. The actual layout of the current graph is not changed by this operation.

```c
void gw.adjust_coords_to_win(node_array<double>& xpos,
node_array<double>& ypos,
edge_array<list<double>> &xbends,
edge_array<list<double>> &ybends)
```

same as `adjust_coords_to_box(xpos, ypos, xbends, ybends, wx0, wy0, wx1, wy1)` for the current window rectangle (wx0, wy0, wx1, wy1).
### 18.6. GRAPH WINDOWS (GRAPHWIN)

**void gw.adjust_coords_to_win(node_array<double>& xpos, node_array<double>& ypos)**

same as adjust_coords_to_box(xpos, ypos, wx0, wy0, wx1, wy1) for the current window rectangle (wx0, wy0, wx1, wy1).

**void gw.remove_bends(edge e)** removes all bends from edge e.

**void gw.remove_bends()** removes the bends of all edges of the graph.

**void gw.reset_edge_anchors()** resets all edge anchor positions to (0, 0).

**int gw.load_layout(istream& istr)**

read layout from stream istr.

**bool gw.save_layout(ostream& ostr)**

save layout to stream ostr.

**bool gw.save_layout(string fname, bool ask_override = false)**

save layout to file fname.

### h) Zooming

**void gw.zoom(double f)** zooms the window by factor f.

**void gw.zoom_area(double x0, double y0, double x1, double y1)**

performs a zoom operation for the rectangular area with current coordinates (x0, y0, x1, y1).

**void gw.zoom_graph()** performs a zoom operation, such that the graph fills the entire window.

**void gw.unzoom()** undoes last zoom operation.

### i) Operations in Edit-mode

Before entering edit mode ...  

**gw_action gw.set_action(long mask, gw_action func)**

sets action associated with condition mask to func and returns previous action for this condition. Here gw_action is the type void (*func)(GraphWin&, const point&). For func = NULL the corresponding action is deleted.

**gw_action gw.get_action(long mask)**

returns the action associated with condition mask.
void gw.reset_actions() resets all actions to their defaults.

void gw.clear_actions() deletes all actions.

void gw.add_node_menu(string label, gw_action func)
  appends action function func with label label to the context menu for nodes (opened by clicking with the right mouse button on a node).

void gw.add_edge_menu(string label, gw_action func)
  appends action function func with label label to the context menu for edges (opened by clicking with the right mouse button on an edge).

void gw.set_new_node_handler(bool (*f)(GraphWin&, const point&))
  f(gw, p) is called every time before a node is to be created at position p.

void gw.set_new_node_handler(void (*f)(GraphWin&, node) = NULL)
  f(gw, v) is called after node v has been created.

void gw.set_new_edge_handler(bool (*f)(GraphWin& , node, node))
  f(gw, v, w) is called before the edge (v, w) is to be created.

void gw.set_new_edge_handler(void (*f)(GraphWin& , edge) = NULL)
  f(gw, e) is called after the edge e has been created.

void gw.set_start_move_node_handler(bool (*f)(GraphWin& , node) = NULL)
  f(gw, v) is called before node v is to be moved.

void gw.set_move_node_handler(void (*f)(GraphWin& , node) = NULL)
  f(gw, v) is called every time node v reaches a new position during a move operation.

void gw.set_end_move_node_handler(void (*f)(GraphWin& , node))
  f(gw, v) is called after node v has been moved.

void gw.set_del_node_handler(bool (*f)(GraphWin& , node))
  f(gw, v) is called before the node v is to be deleted.

void gw.set_del_node_handler(void (*f)(GraphWin& ) = NULL)
  f(gw) is called every time after a node was deleted.

void gw.set_del_edge_handler(bool (*f)(GraphWin& , edge))
  f(gw, e) is called before the edge e is to be deleted.
void gw.set_del_edge_handler( void (*f)(GraphWin&, ) = NULL)
    f(gw) is called every time after an edge was deleted.

void gw.set_start_edge_slider_handler( void (*f)(GraphWin&, edge, double) = NULL, int sl = 0)
    f(gw, e, pos) is called before slider sl of edge e is to be moved. Here pos is the current slider position.

void gw.set_edge_slider_handler( void (*f)(GraphWin&, edge, double) = NULL, int sl = 0)
    f(gw, e, pos) is called every time slider sl of edge e reaches a new position pos during a slider move.

void gw.set_end_edge_slider_handler( void (*f)(GraphWin&, edge, double) = NULL, int sl = 0)
    f(gw, e, pos) is called after slider sl of edge e has been moved to the final position pos.

void gw.set_init_graph_handler( bool (*f)(GraphWin& ) )
    f is called every time before the entire graph is replaced, e.g. by a clear, generate, or load operation.

void gw.set_init_graph_handler( void (*f)(GraphWin& ) = NULL)
    f is called every time after the entire graph was replaced.

j) Menus

The default menu ...

void gw.set_default_menu(long mask)
    ...

void gw.add_menu(long menu_id)
    ...

void gw.del_menu(long menu_id)
    ...

Extending menus by new buttons and sub-menus ...

int gw.add_menu(string label, int menu_id = 0, char *pmap = 0)
    ...

int gw.add_simple_call( void (*func)(GraphWin& ), string label, int menu_id = 0, char *pmap = 0)
    ...
int gw.add_simple_call(void (*func)(GraphWin& ), string label, int menu_id,
    int bm_w, int bm_h, unsigned char * bm_bits)
...

int gw.add_member_call(void (GraphWin::* func)(), string label,
    int menu_id = 0, char * pmap = 0)
...

int gw.add_member_call(void (GraphWin::* func)(), string label, int menu_id,
    int bm_w, int bm_h, unsigned char * bm_bits)
...

void gw.add_separator(int menu_id)
    ...

void gw.display_help_text(string fname)
    displays the help text contained in name.hlp. The file name.hlp must exist either in the current working directory or in $LEDAROOT/incl/Help.

void gw.add_help_text(string name)
    adds the help text contained in name.hlp with label name to the help menu of the main window. The file name.hlp must exist either in the current working directory or in $LEDAROOT/incl/Help. Note that this operation must be called before gw.display().

int gw.get_menu(string label)
    returns the number of the submenu with label label or -1 if no such menu exists.

void gw.enable_call(int id)
    enable call with id id.

void gw.disable_call(int id)
    disable call with id id.

bool gw.is_call_enabled(int id)
    check if call with id is enabled.

void gw.enable_calls()
    ...

void gw.disable_calls()
    ...

k) Input/Output

int gw.read_gw(istream& in)
    reads graph in gw format from stream in.

int gw.read_gw(string fname)
    reads graph in gw format from file fname.
18.6. GRAPH WINDOWS (GRAPHWIN)

```cpp
bool gw.save_gw(ostream& out)
  writes graph in gw format to output stream out.
bool gw.save_gw(string fname, bool ask_override = false)
  saves graph in gw format to file fname.
int gw.read_gml(istream& in)
  reads graph in GML format from stream in.
int gw.read_gml(string s)
  reads graph in GML format from string s.
int gw.read_gml(string fname, bool ask_override = false)
  reads graph in GML format from file fname. Returns 1 if fname cannot be opened, 2 if a parser error occurs, and 0 on success.
bool gw.save_gml(ostream& out)
  writes graph in GML format to output stream out.
bool gw.save_gml(string fname, bool ask_override = false)
  saves graph to file fname in GML format.
bool gw.save_ps(string fname, bool ask_override = false)
  saves a postscript representation of the graph to fname.
bool gw.save_svg(string fname, bool ask_override = false)
  saves a SVG representation of the graph to fname.
bool gw.save_latex(string fname, bool ask_override = false)
  saves a postscript/latex representation of the graph to fname.
bool gw.save_wmf(string fname, bool ask_override = false)
  saves a windows metafile representation of the graph to fname.
bool gw.unsaved_changes()
  returns true if the graph has been changed after the last save (gw or gml) operation.
bool gw.save_defaults(string fname)
  saves the default attributes of nodes and edges to file fname.
bool gw.read_defaults(string fname)
  reads the default attributes of nodes and edges from file fname.
```
1) Miscellaneous

```c
void gw.set_window(window& W)  
    makes W the window of gw.

void gw.set_graph(graph& G) makes G the graph of gw.

void gw.undo_clear() empties the undo and redo stacks.

bool gw.wait()  
    waits until the done button is pressed (true returned) or exit is selected from the file menu (false returned).

bool gw.wait(const char *msg)  
    displays msg and waits until the done button is pressed (true returned) or exit is selected from the file menu (false returned).

bool gw.wait(float sec, const char *msg = "")  
    as above but waits no longer than sec seconds returns ?? if neither button was pressed within this time interval.

void gw.acknowledge(string s)  
    displays string s and asks for acknowledgement.

node gw.ask_node()  
    asks the user to select a node with the left mouse button. If a node is selected it is returned otherwise nil is returned.

edge gw.ask_edge()  
    asks the user to select an edge with the left mouse button. If an edge is selected it is returned otherwise nil is returned.

bool gw.define_area(double& x0, double& y0, double& x1, double& y1, 
                     const char *msg = "")  
    displays message msg and returns the coordinates of a rectangular area defined by clicking and dragging the mouse.

list<node> gw.get_nodes_in_area(double x0, double y0, double x1, double y1)  
    returns the list of nodes intersecting the rectangular area (x0,y0,x1,y1).

list<edge> gw.get_edges_in_area(double x0, double y0, double x1, double y1)  
    returns the list of edges intersecting the rectangular area (x0,y0,x1,y1).

void gw.save_node_attributes( )  
    ...
```
void gw.save_edge_attributes() ...

void gw.save_all_attributes() ...

void gw.restore_node_attributes() ...

void gw.restore_edge_attributes() ...

void gw.restore_all_attributes() ...

void gw.reset_nodes(long mask = N_ALL) reset node parameters to their default values.

void gw.reset_edges(long mask = E_ALL) reset edge parameters to their default values.

void gw.reset() reset node and edge parameters to their default values.

void gw.reset_defaults() resets default parameters to their original values.

node gw.get_edit_node() returns a node under the current mouse pointer position (nil if there is no node at the current position)

edge gw.get_edit_edge() returns an edge under the current mouse pointer position (nil if there is no edge at the current position).

int gw.get_edit_slider() returns the number of the slider under the current mouse pointer position (0 if there is no edge slider at the current position).

void gw.get_bounding_box(double& x0, double& y0, double& x1, double& y1) computes the coordinates (x0, y0, x1, y1) of a minimal bounding box for the current layout of the graph.

void gw.get_bounding_box(const list<node>& V, const list<edge>& E, double& x0, double& y0, double& x1, double& y1) computes the coordinates (x0, y0, x1, y1) of a minimal bounding box for the current layout of subgraph (V, E).
18.7 The GraphWin (GW) File Format

The gw-format is the external graph format of GraphWin. It extends LEDA’s graph format described in the previous section by additional parameters and attributes for describing graph drawings. Note that the gw-format was not defined to be a readable or easy to extend file format (in contrast to the GML format that is also supported by GraphWin).

Each gw file starts with a LEDA graph followed by a (possibly empty) layout section. An empty layout section indicates that no drawing of the graph is known, e.g. in the input file of a layout algorithm. If a layout section is given, it consists of three parts:

1. global parameters
2. node attributes
3. edge attributes

Global Parameters

The global parameter section consists of 7 lines (with an arbitrary number of inter-mixed comment-lines).

1. version line
   The version line specifies the version of the gw-format. It consists of the string GraphWin followed by a floating-point number (1.32 for the current version of GraphWin).

2. window parameters
   scaling wxmin wymin wxmax wymax
   This line consists of 5 floating-point numbers specifying the scaling, minimal/maximal x- and y-coordinates of the window (see the window class of LEDA).

3. node label font
   type size
   This line defines the font used for node labels. The type value of of type int. Possible values (see gw_font_type) are
   0 (roman_font)
   1 (bold_font)
   2 (italic_font)
   3 (fixed_font). The size value is of type int and defines the size of the font in points.

4. edge label font
   type size as above, but defines the font used for edge labels.

5. node index format
   format
   This line contains a printf-like format string used for constructing the index label of nodes (e.g. %d).
6. edge index format
   
   This line contains a printf-like format string used for constructing the index label of edges (e.g. `%d`).

7. multi-edge distance
   
   This line contains a floating-point parameter `dist` that defines the distance used to draw parallel edges.

We close the description of the global parameter section with an example.

```
# version
GraphWin 1.32
# window parameters
1.0 -10.0 -5.0 499.0 517.0
# node font
0 12
# edge font
0 12
# node index string
%d
# edge index string
%d
# multi-edge distance
4.0
```

**Node Attributes**

The node attribute section contains for each node of the graph a line consisting of the following attributes (separated by blanks). More precisely, the $i$-th line in this section defines the attributes of the $i$-th node of the graph (see section 12.27).

x-coordinate
an attribute of type `double` defining the x-coordinate of the center of the node.

y-coordinate
an attribute of type `double` defining the y-coordinate of the center of the node.

shape
an attribute of type `int` defining the shape of the node. Possible values are (see `gw_node_shape` of GraphWin)

0 (circle_node)
1 (ellipse_node)
2 (square_node)
3 (rectangle_node).
border color
    an attribute of type int defining the color used to draw the boundary line of the node. Possible values are (see the LEDA color type)
    -1 (invisi\textcolor{red}{l}e)
    0 (black)
    1 (white)
    2 (red)
    3 (green)
    4 (blue)
    5 (yellow)
    6 (violet)
    7 (orange)
    8 (cyan)
    9 (brown)
    10 (pink)
    11 (green2)
    12 (blue2)
    13 (grey1)
    14 (grey2)
    15 (grey3)
    16 (ivory).

border width
    an attribute of type double defining the width of the border line of the node.

radius1
    an attribute of type double defining the horizontal radius of the node

radius2
    an attribute of type double defining the vertical radius of the node

color
    an attribute of type int defining the color used to fill the interior of the node. See the LEDA color type for possible values.

label type
    an attribute of type int specifying the label type. Possible values (see gw.label.type of GraphWin) are
    0 (no.label)
    1 (user.label)
    2 (data.label)
    3 (index.label).

label color
    an attribute of type int defining the color used to draw the label of the node. See the LEDA color type for possible values.

label position
    an attribute of type int defining the label position. Possible values (see gw.position
of GraphWin) are
0 (central_pos)
1 (northwest_pos)
2 (north_pos)
3 (northeast_pos)
4 (east_pos)
5 (southeast_pos)
6 (south_pos)
7 (southwest_pos)
8 (west_pos).

user label
an attribute of type string defining the user label of the node.

We close this section with an example of a node attribute line that describes a circle node at position (189, 260) with border color black, border width 0.5, horizontal and vertical radius 12, interior color ivory, label type index_label, label position east_pos, and an empty user label.

# x y shape b-clr b-width radius1 radius2 clr l-type l-clr l-pos l-str
189.0 260.0 0 1 0.5 12.0 12.0 16 3 -1 4

Edge Attributes:

The edge attribute section contains for each edge of the graph a line consisting of the following attributes (separated by blanks). More precisely, the $i$-th line in this section defines the attributes of the $i$-th edge of the graph (see section 12.27).

width
an attribute of type double defining the width of the edge.

color
an attribute of type color defining the color of the edge.

shape
an attribute of type int defining the shape of the edge. Possible values (see gw_edge_shape of GraphWin) are
0 (poly_edge)
1 (circle_edge)
2 (bezier_edge)
3 (spline_edge).

style
an attribute of type int defining the line style of the edge. Possible values (see the LEDA line_style type) are
0 (solid)
1 (dashed)
2 (dotted)
3 (dashed_dotted).
direction
an attribute of type \textit{int} defining whether the edge is drawn as a directed or an undirected edge. Possible values (see \texttt{gw\_edge\_dir} of GraphWin) are
\begin{itemize}
  \item 0 (\texttt{undirected\_edge})
  \item 1 (\texttt{directed\_edge})
  \item 2 (\texttt{redirected\_edge})
  \item 3 (\texttt{bidirected\_edge}).
\end{itemize}

label type
an attribute of type \textit{int} defining the label type of the edge. Possible values (see \texttt{gw\_label\_type} of GraphWin) are
\begin{itemize}
  \item 0 (\texttt{no\_label})
  \item 1 (\texttt{user\_label})
  \item 2 (\texttt{data\_label})
  \item 3 (\texttt{index\_label}).
\end{itemize}

label color
an attribute of type \textit{int} defining the color of the edge label. See the LEDA \texttt{color} type for possible values.

label position
an attribute of type \textit{int} defining the position of the label. Possible values (see \texttt{gw\_position} of GraphWin) are
\begin{itemize}
  \item 0 (\texttt{central\_pos})
  \item 4 (\texttt{east\_pos})
  \item 8 (\texttt{west\_pos blue}).
\end{itemize}

polyline
an attribute of type \textit{list < point>} defining the polyline used to draw the edge. The list is represented by the number \texttt{n} of elements followed by \texttt{n} points \((x_i, y_i)\) for \(i = 1 \ldots n\). The first element of the list is the point where the edge leaves the interior of the source node, the last element is the point where the edge enters the interior of the target node. The remaining elements give the sequence of bends (or control points in case of a bezier or spline edge).

user label
an attribute of type \texttt{string} defining the user label of the edge.

We close this section with an example of an edge attribute line that describes a blue solid polygon edge of width 0.5 drawn directed from source to target, with a black user-defined label ”my label” at position \texttt{east\_pos}, centered source and target anchors, and with a bend at position (250, 265).

\begin{verbatim}
# width clr shape style dir ltype lclr lpos sanch tanch poly lstr
0.5 4 0 0 1 1 1 4 (0,0) (0,0) 3 (202.0,262.0) (250.0,265.0)
\end{verbatim}
18.7. THE GRAPHWIN (GW) FILE FORMAT

18.7.1 A complete example

LEDA.GRAPH
void
void
5
|{}|
|{}|
|{}|
|{}|
|{}|
7
1 2 0 |{}|
1 3 0 |{}|
2 3 0 |{}|
3 4 0 |{}|
3 5 0 |{}|
4 5 0 |{}|
5 1 0 |{}|
# version string
GraphWin 1.320000
# scaling wxmin wymin wxmax wymax
1.117676 -10 -5.6875 499.8828 517.6133
# node label font and size
0 13.6121
# edge label font and size
0 11.79715
# node index format
%d
# edge index format
%d
# multi-edge distance
4.537367
#
# node infos
# x y shape bclr bwidth r1 r2 clr ltype lclr lpos lstr
189.4805 260.8828 0 1 0.544484 12.70463 12.70463 16 4 -1 4
341.5508 276.0898 0 1 0.544484 12.70463 12.70463 16 4 -1 4
384.4883 175.9023 0 1 0.544484 12.70463 12.70463 16 4 -1 4
294.1406 114.1797 0 1 0.544484 12.70463 12.70463 16 4 -1 4
186.7969 114.1797 0 1 0.544484 12.70463 12.70463 16 4 -1 4
#
# edge infos
# width clr shape style dir ltype lclr lpos sanch tanch poly lstr
0.9074733 1 0 0 1 1 1 5 (0,0) (0,0) 2 (202.122,262.147) (328.9092,274.8257)
0.9074733 1 0 0 1 1 1 5 (0,0) (0,0) 2 (201.1272,255.8074) (372.8415,180.9778)
0.9074733 1 0 0 1 1 1 5 (0,0) (0,0) 2 (346.5554,264.4124) (379.4837,187.5797)
0.9074733 1 0 0 1 1 1 5 (0,0) (0,0) 2 (373.998,168.7357) (304.6309,121.3463)
0.9074733 1 0 0 1 1 1 5 (0,0) (0,0) 2 (372.361,172.116) (198.9242,117.966)
0.9074733 1 0 0 1 1 1 5 (0,0) (0,0) 2 (281.436,114.1797) (199.5015,114.1797)
0.9074733 1 0 0 1 1 1 5 (0,0) (0,0) 2 (187.0292,126.8822) (189.2481,248.1803)
18.8 Geometry Windows (GeoWin)

1. Definition

An instance of data type GeoWin is an editor for sets of geometric objects. It can be used for the visualization of result and progression of geometric algorithms. GeoWin provides an interactive interface and a programming interface to visualize and manipulate geometric objects and data structures.

Sets of geometric objects are maintained in so-called scenes.

Scenes

Scenes are instances of the various scene data types supported by GeoWin. They are used to store collections of geometric objects and attributes of the objects and collections. Furthermore the scene classes have to provide functionality for GeoWin to handle the geometric objects of a scene.

Each scene stores geometric objects in a container (a LEDA-list or STL-list). We call these geometric objects stored in a container of a scene the contents of a scene. The scenes and their contents can be manipulated by the interactive interface and the programming interface of GeoWin.

With every scene a set of attributes is associated. Most of them describe the visual representation of the scene, for instance the boundary- and fill-color of the objects, the visibility of the scene,... .

We use the type geo\_scene as the scene item type of GeoWin; it may be helpful to view it as pointers to scenes.

We distinguish the following types of scene classes:

1. Edit Scenes (type GeoEditScene<CONTAINER>)

   where CONTAINER is the type of the scene’s container storing the contents of the scene, for instance list<point>. These scenes can be edited by the user through the interactive interface of GeoWin. Note that edit scenes have some special features. An important feature is the possibility to select objects through the interactive interface. These selected objects have special attributes, see the table of scene attributes.

2. Result Scenes (type GeoResultScene<I, R>)

   These scenes are not independently editable by the user. The contents of result scenes is computed by a user-defined update function or update object executing a geometric algorithm. This recomputation of the scene contents will be done every time when another scene (this other scene we call the input scene of the result scene)
changes. The contents of the result scene is stored in a container of type $R$. The input scene must be a Basic Scene with a container of type $I$. The update function
\[\text{void \hspace{1em} (*f-update)(const I& input, R& result)}\] gets the contents of this input scene and computes the contents result of the result scene. We say that the result scene depends on its input scene.

3. Basic Scenes (type GeoBaseScene<CONTAINER>)

Edit Scenes and Result Scenes are derived from Basic Scenes. The basic scene type works on container types providing an interface as the list of the STL library. More precisely, CONTAINER has to support the following type definitions and STL-like operations:

- **value_type** - the type $T$ of the values the container holds
- **iterator**
- **operations begin** ( ) and **end** ( ) returning an iterator that can be used for begining (ending) the traversal of the container
- **void push_back(const T&)** for inserting an element at the end of the container
- **iterator insert(iterator it, const T&)** for inserting an element (before it)
- **void erase(iterator it)** for erasing an element at position it
- **operation bool empty ( )** returning true if the container is empty, false otherwise

That means, that LEDA lists can be used as well as containers.

The programming interface of GeoWin provides various operations to create Edit Scenes and Result Scenes. Basic Scenes are not created directly by the operations of the programming interface, but they are used for derivation of the other scene types, and we will find them in the programming interface, when both Edit and Result Scenes are supported by an operation.

**GeoWin - class**

We explain some important terms of the GeoWin data type. Every instance $GW$ of GeoWin can maintain a number of geo_scenes.

Visible scenes will be displayed by $GW$, non-visible scenes will not be displayed. Displayed means, that the contents of the scene will be displayed. A special case is the active scene of $GW$. Every GeoWin can have at most one active scene. The active scene is an Edit Scene with input focus. That means that this scene is currently edited by the user through the interactive interface. Note that the currently active scene will be displayed.

Another important topic is the display order of scenes. Every scene has an associated non-negative z-coordinate. When a scene is created, it gets z-coordinate 0. When $GW$ redraws a scene, the contents of this scene and the contents of its visible dependent scenes is drawn. In the redraw-operation of GeoWin the scenes with higher z-coordinates will be drawn in the background of scenes with lower z-coordinate. The scenes with z-coordinate
0 will be drawn on top in the order of their creation in its instance of GeoWin (the scene, that was created last and has z-coordinates 0 is the scene on top).

Attributes of scenes

The following attributes are associated with every scene.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>active</td>
<td>bool</td>
<td>activity status of a scene</td>
</tr>
<tr>
<td>active_line_width</td>
<td>int</td>
<td>line width used for drawing objects of active scenes</td>
</tr>
<tr>
<td>client_data</td>
<td>void*</td>
<td>some void*-pointers that can be associated with a scene</td>
</tr>
<tr>
<td>color</td>
<td>color</td>
<td>boundary color of non-selected objects</td>
</tr>
<tr>
<td>description</td>
<td>string</td>
<td>a string describing the scene</td>
</tr>
<tr>
<td>fill_color</td>
<td>color</td>
<td>fill color of objects</td>
</tr>
<tr>
<td>line_style</td>
<td>line_style</td>
<td>line style used for drawing objects</td>
</tr>
<tr>
<td>line_width</td>
<td>int</td>
<td>line width used for drawing objects of non-active scenes</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>the name of the scene</td>
</tr>
<tr>
<td>point_style</td>
<td>point_style</td>
<td>point style used for drawing objects</td>
</tr>
<tr>
<td>selection_color</td>
<td>color</td>
<td>boundary color selected objects</td>
</tr>
<tr>
<td>selection_fill_color</td>
<td>color</td>
<td>fill color of selected objects</td>
</tr>
<tr>
<td>show_orientation</td>
<td>bool</td>
<td>disables/enables the drawing of object orientations/directions</td>
</tr>
<tr>
<td>text_color</td>
<td>color</td>
<td>text label color</td>
</tr>
<tr>
<td>visible</td>
<td>bool</td>
<td>visibility of a scene in its GeoWin</td>
</tr>
<tr>
<td>z_order</td>
<td>int</td>
<td>z-coordinate of a scene in its GeoWin</td>
</tr>
</tbody>
</table>

Attributes and parameters of instances of GeoWin

Every instance of type GeoWin uses the following attributes and parameters. The parameters starting with \textit{d3}\_ are influencing the 3-d output option of GeoWin. This 3-d output option uses the LEDA-class \textit{d3\_window} for displaying geometric objects. See also the \textit{d3\_window} - Manualpages for a description of the 3-d output parameters.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>active_scene</td>
<td>geo_scene</td>
<td>the active scene</td>
</tr>
<tr>
<td>bg_color</td>
<td>color</td>
<td>window background color</td>
</tr>
<tr>
<td>bg_pixmap</td>
<td>string</td>
<td>name of the used window background pixmap</td>
</tr>
<tr>
<td>d3_elimination</td>
<td>bool</td>
<td>true - in the d3-output hidden lines will be eliminated</td>
</tr>
<tr>
<td>d3_show_edges</td>
<td>bool</td>
<td>enables/disables the redraw of edges in the d3-output</td>
</tr>
<tr>
<td>d3_solid</td>
<td>bool</td>
<td>true - in the d3-output faces will be drawn in different grey scales</td>
</tr>
<tr>
<td>grid_dist</td>
<td>double</td>
<td>width of the grid in the drawing area</td>
</tr>
<tr>
<td>grid_style</td>
<td>grid_style</td>
<td>style of the grid in the drawing area</td>
</tr>
<tr>
<td>show_grid</td>
<td>bool</td>
<td>defines if a grid should be used in the drawing area of the window</td>
</tr>
<tr>
<td>show_position</td>
<td>bool</td>
<td>true - the coordinates of the mouse cursor are displayed</td>
</tr>
</tbody>
</table>
The geometric objects

The objects stored in the containers of the scenes have to support input and output operators for streams and the LEDA window and the output operator to the \texttt{ps file}.

Manual overview

The following manual pages have this structure:

- a) Main operations (creation of scenes)
- b) Window operations (initialization of the drawing window)
- c) Scenes and scene groups (get/set - operations for changing attributes)
- d) I/O operations
- e) View operations (zooming)
- f) Parameter operations (get/set - operations for instances of type \texttt{GeoWin})
- g) Event handling
- h) Scene group operations
- i) Further operations (changing of the user interface, 3d output, ...)

\texttt{#include <LEDA/graphics/geowin.h>}

2. Creation

\texttt{GeoWin GW(const char * label = "GEOWIN");}

creates a \texttt{GeoWin GW}. \texttt{GW} is constructed with frame label \texttt{label}

\texttt{GeoWin GW(int w, int h, const char * label = "GEOWIN");}

creates a \texttt{GeoWin GW} with frame label \texttt{label} and window size \texttt{w \times h} pixels.

3. Operations

a) Main Operations

\textit{In this section you find operations for creating scenes and for starting the interactive mode of GeoWin.}

The \texttt{new scene} and \texttt{get objects} operations use member templates. If your compiler does not support member templates, you should use instead the templated functions
geowin_new_scene and geowin_get_objects with GW as an additional first parameter.

All new_scene operations can get as an optional last parameter a pointer to a function that is used to compute the three-dimensional output of the scene. The type of such a function pointer \( f \) is

\[
\text{void \(^*(f)(\text{const } T &, \text{d3_window} &, \text{GRAPH<d3_point, int> &})\)}
\]

where \( T \) is the type of the container used in the scene (for instance \( \text{list<point>} \)). The function gets a reference to the container of it’s scene, a reference to the output \( \text{d3_window} \) and to the parametrized graph describing the three-dimensional output. The function usually adds new nodes and edges to this graph. Note that every edge in the graph must have a reversal edge (and the reversal information has to be set).

Example:

```cpp
void segments_d3(const list<segment> & L, d3_window & W,
                 GRAPH<d3_point, int> & H)
{
    GRAPH<d3_point, int> G;
    segment iter;
    forall(iter,L) {
        node v1 = G.new_node(d3_point(iter.source().xcoord(),
                                iter.source().ycoord(),0));
        node v2 = G.new_node(d3_point(iter.target().xcoord(),
                                iter.target().ycoord(),0));
        edge e1 = G.new_edge(v1,v2);
        edge e2 = G.new_edge(v2,v1);
        G.set_reversal(e1,e2);
    }
    H.join(G);
}
```

In this simple example the function gets a list of segments. For every segment in the list two new nodes and two new edges are created. The reversal information is set for the two edges. At the end the local graph \( G \) is merged into \( H \).

The following templated new_scene operation can be used to create edit scenes. The CONTAINER has to be a \( \text{list<T>} \), where \( T \) is one of the following 2d LEDA kernel type

- (rat_)point
- (rat_)segment
- (rat_)line
- (rat_)circle
- (rat_)polygon
or a \texttt{d3_point} or a \texttt{d3_rat_point}. If you want to use the other 2d LEDA kernel types, you have to include \texttt{geowin_init.h} and to initialize them for usage in \texttt{GeoWin} by calling the \texttt{geowin_init_default_type} function at the beginning of \texttt{main} (before an object of data type \texttt{GW} is constructed). If you want to use the other 3d LEDA kernel types, you have to include \texttt{geowin_init_d3.h} and to initialize them for usage in \texttt{GeoWin} by calling the \texttt{geowin_init_default_type} function at the beginning of \texttt{main} (before an object of data type \texttt{GW} is constructed).

\begin{verbatim}
template <class CONTAINER>
GeoEditScene<CONTAINER>* GW.new_scene(CONTAINER& c)
creates a new edit scene and returns a pointer to the created scene. \(c\) will be the container storing the contents of the scene.

template <class CONTAINER>
GeoEditScene<CONTAINER>* GW.new_scene(CONTAINER& c, string str,
\texttt{D3_FCN} f)
creates a new edit scene and returns a pointer to the created scene. \(c\) will be the container storing the contents of the scene. The name of the scene will be set to \textit{str}.
\end{verbatim}

The following \texttt{new\_scene} operations can be used to create result scenes. Result scenes use the contents of another scene (the input scene) as input for a function (the update function). This function computes the contents of the result scene. The update function is called every time when the contents of the input scene changes. Instead of using an update function you can use an update object that encapsulates an update function. The type of this update object has to be \texttt{geowin_update\langle I, R \rangle} (\texttt{I} - type of the container in the input scene, \texttt{R} - type of the container in the result scene) or a class derived from it. A derived class should overwrite the virtual update function

\begin{verbatim}
void update(const I & in, R & out)
\end{verbatim}

of the base class to provide a user defined update function. The class \texttt{geowin_update\langle I, R \rangle} has 3 constructors getting function pointers as arguments:

\begin{verbatim}
geowin_update(void (*f)(const I & in, R & res))
geowin_update(void (*f)(const I & in, R::value_type & obj))
geowin_update(R::value_type (*f)(const I & in))
\end{verbatim}

When the update object is constructed by calling the constructor with one of these function pointers, the function \(\ast f\) will be called in the update method of the update object. The first variant is the normal update function that gets the contents \texttt{in} of the input scene and computes the contents \texttt{res} of the output scene. In the second variant the contents of the result scene will first be cleared, then the update function will be called and \texttt{obj} will
be inserted in the result scene. In the third variant the contents of the result scene will be cleared, and then the object returned by \((\ast f)\) will be inserted in the result scene. The class \textit{geowin\_update} has also the following virtual functions:

\begin{verbatim}
bool insert(const InpObject & new)
bool del(const InpObject & new)
bool change(const InpObject & old_obj, const InpObject & new_obj)
\end{verbatim}

where \textit{new} is a new inserted or deleted object and \textit{old\_obj} and \textit{new\_obj} are objects before and after a change. \textit{InpObject} is the value type of the container of the input scene. With these functions it is possible to support incremental algorithms. The functions will be called, when in the input scene new objects are added (\textit{insert}), deleted (\textit{del}) or changed when performing a move or rotate operation (\textit{change}). In the base class \textit{geowin\_update}\textit{<I, R>} these functions return \textit{false}. That means, that the standard update-function of the update object should be used. But in derived classes it is possible to overwrite these functions and provide user-defined update operations for these three incremental operations. Then the function has to return \textit{true}. That means, that the standard update function of the update object should not be used. Instead the incremental operation performs the update-operation.

It is also possible to provide user defined redraw for a scene. For this purpose we use redraw objects derived from \textit{geowin\_redraw}. The derived class has to overwrite the virtual redraw function

\begin{verbatim}
void draw(window & W, color c1, color c2, double x1, double y1, double x2, double y2)
\end{verbatim}

of the base class to provide a user defined redraw function. The first 3 parameters of this function are the redraw window and the first and second drawing color (\textit{color} and \textit{color2}) of the scene. The class \textit{geowin\_redraw} has also a virtual method

\begin{verbatim}
bool draw\_container()
\end{verbatim}

that returns \textit{false} in the base class. If you want the user defined redraw of the scene (provided by the redraw function \textit{draw}) and the execution of the ‘normal’ redraw of the scene as well (output of the objects stored in the container of the scene), you have to overwrite \textit{draw\_container} in a derived class by a function returning \textit{true}. A virtual method

\begin{verbatim}
bool write\_postscript(ps\_file & PS, color c1, color c2)
\end{verbatim}

is provided for output to a LEDA postscript file \textit{PS}. \textit{c1} and \textit{c2} are the first and second drawing color (\textit{color} and \textit{color2}) of the scene. Another class that can be used for user defined redraw is the templated class \textit{geowin\_redraw\_container\textless CONTAINER\textgreater}. This class has as well virtual functions for redraw and postscript output, but provides a slighly changed interface:

\begin{verbatim}
bool draw(const CONTAINER & c, window & w, color c1, color c2,
double, double, double, double)
\end{verbatim}
bool write_postscript(const CONTAINER& c, ps_file& ps, color c1, color c2)

The parameters of these two virtual functions are like the parameters of the members with the same name of geowin_redraw, but there is an additional first parameter. This parameter is a reference to the container of the scene that has to be redrawn.

In update- and redraw- functions and objects the following static member functions of the GeoWin class can be used:

\[
\begin{align*}
GeoWin &\ast GeoWin::get_call_geowin() \\
geo\_scene & GeoWin::get_call_scene() \\
geo\_scene & GeoWin::get_call_input\_scene()
\end{align*}
\]

The first function returns a pointer to the GeoWin of the calling scene, the second returns the calling scene and the third (only usable in update functions/objects) returns the input scene of the calling scene.

Note that \(S\) and \(R\) in the following operations are template parameters. \(S\) and \(R\) have to be a \texttt{list<T>}, where \(T\) is a 2d LEDA kernel type, a \texttt{d3\_point} or a \texttt{d3\_rat\_point}. \(S\) is the type of the contents of the input scene, \(R\) the type of the contents of the created result scene. All operations creating result scenes return a pointer to the created result scene.

This section contains three small example programs showing you the usage of the \texttt{new\_scene} operations for the creation of result scenes. All example programs compute the convex hull of a set of points stored in the container of an input scene \texttt{sc\_points} and store the computed hull in a result scene \texttt{sc\_hull}.

\[
\texttt{template } <\texttt{class S, class R}> \\
\texttt{GeoResultScene<S,R} \ast GW\texttt{.new\_scene( void }\ast f\_update\texttt{)( const S& , R& ), geo\_scene sc, } \\
\texttt{string str, D3\_FCN f }\texttt{=} \texttt{NULL})
\]

creates a new result scene with name \texttt{str}. The input scene for this new result scene will be \texttt{sc}. The update function will be \texttt{f\_update}.

The first example program shows the usage of the \texttt{new\_scene} operation taking an update function pointer. The update function computes the convex hull of the points stored in the input scene. The result polygon will be inserted in the container \(P\) of the result scene.

```cpp
#include <LEDA/graphics/geowin.h>
#include <LEDA/geo/float_geo_alg.h>

using namespace leda;

void convex_hull(const list<point>& L, list<polygon>& P) {
    P.clear(); P.append(CONVEX_HULL_POLY(L));
}

int main() {
    GeoWin gw;
    list<point> LP;
```
geo_scene sc_points = gw.new_scene(LP);
geo_scene sc_hull = gw.new_scene(convex_hull, sc_points, "Convex hull");
gw.set_color(sc_hull, blue);
gw.set_visible(sc_hull, true);

gw.edit(sc_points);
return 0;
}

template <class S, class R>
GeoResultScene<S, R>* GW.new_scene(geowin_update<S, R>& up, list<geo_scene>& infl,
    string str, D3_FCN f = NULL)
creates a new result scene scr with name str. The input scene for this new result scene will be the first scene in infl. The update object will be up. up has to be constructed by a call up(fu,0), where fu is a function of type void fu(const C0&, const C1&, ..., const Cn&, R&). infl is a list of scenes influencing the result scene. C0,...,Cn are the types of the containers of the scenes in infl. When one of the scenes in infl changes, fu will be called to update the contents of scr. Precondition: infl must not be empty.

template <class S, class R>
GeoResultScene<S, R>* GW.new_scene(geowin_update<S, R>& up, geo_scene sc_input,
    string str, D3_FCN f = NULL)
creates a new result scene with name str. The input scene for this new result scene will be sc_input. The update object will be up.

The second variant of the example program uses an update object update.
return 0;
}

template <class S, class R>
void GW.set_update(geo_scene res, geowin_update<S, R>& up)
makes up the update object of res. Precondition: res points to a scene of type GeoResultScene<S, R>.

template <class S, class R>
void GW.set_update(geo_scene res, void (*f_update)(const S&, R&))
makes f_update the update function of res. Precondition: res points to a scene of type GeoResultScene<S, R>.

GeoResultScene<S, R>* GW.new_scene(geowin_update<S, R>& up, geowin_redraw& rd,
geo_scene sc_input, string str,
D3_FCN f = NULL)
creates a new result scene with name str. The input scene for this new result scene will be sc_input. The update object will be ub. The redraw object will be rd.

The third variant of the example program uses an update and redraw object. We provide a user defined class for update and redraw of the result scene.

#include <LEDA/graphics/geowin.h>
#include <LEDA/geo/float_geo_alg.h>

using namespace leda;

class hull_update_redraw : public geowin_update<list<point>, list<polygon> >,
public geowin_redraw
{
list<polygon> polys;
public:
void update(const list<point>& L, list<polygon>& P)
{
polys.clear();
polys.append(CONVEX_HULL_POLY(L));
}

void draw(window& W, color c1, color c2, double x1, double y1, double x2, double y2)
{
polygon piter;
segment seg;
forall(piter, polys){
forall_segments(seg, piter){
W.draw_arrow(seg, c1);
}
}
```c
int main()
{
    GeoWin gw;
    list<point> LP;

    geo_scene sc_points = gw.new_scene(LP);

    hull_update_redraw up_rd;
    geo_scene sc_hull = gw.new_scene(up_rd, up_rd, sc_points, "Convex hull");
    gw.set_color(sc_hull, blue);
    gw.set_visible(sc_hull, true);

    gw.edit(sc_points);
    return 0;
}
```

```
template <class S, class R>
GeoResultScene<S, R>* GW.new_scene(geowin_update<S, R>& up,
    geowin_redraw_container<R>& rd,
    geo_scene sc_input, string str,
    D3_FCN f = NULL)
creates a new result scene with name str. The input
scene for this new result scene will be sc_input. The
update object will be ub. The redraw container object
will be rd.
```

```
template <class CONTAINER>
bool GW.get_objects(CONTAINER& c)
If the container storing the contents of the current
edit scene has type CONTAINER, then the contents
of this scene is copied to c.
```

```
template <class CONTAINER>
bool GW.get_objects(geo_scene sc, CONTAINER& c)
If the container storing the contents of scene sc has
type CONTAINER, then the contents of scene sc is
copied to c.
```

```
template <class CONTAINER>
void GW.get_selected_objects(GeoEditScene<CONTAINER>* sc,
    CONTAINER& cnt)
returns the selected objects of scene sc in container
cnt.
```

```
template <class CONTAINER>
```
void GW.set_selected_objects(GeoEditScene<CONTAINER> * sc,
    const list<typename CONTAINER::iterator>& LIT)
sselects the objects of scene sc described by the con-
tenents of container LIT.

template <class CONTAINER>
void GW.set_selected_objects(GeoEditScene<CONTAINER> * sc)
sselects all objects of scene sc.

template <class CONTAINER>
void GW.set_selected_objects(GeoEditScene<CONTAINER> * sc,
    const rectangle& R)
sselects all objects of scene sc contained in rectangle R.

void GW.edit( ) starts the interactive mode of GW. The operation re-
turns if either the DONE or Quit button was pressed.

bool GW.edit(geo_scene sc)
edits scene sc. Returns false if the Quit-Button was
pressed, true otherwise.

void GW.register_window(window& win, bool (*ev_fcn)(window * w, int event,
    int but, double x, double y))
if you enter the interactive mode of GW in an applica-
tion, but you want to handle events of other windows
as well, you can register a callback function ev_fcn
for your other window win that will be called when
events associated with win occur. The parameters of
ev_fcn are the window causing the event, the event
that occurred, the button and the x and y coordinates
of the position in win. The handler ev_fcn has to re-
turn true if the interactive mode of GeoWin has to be
stopped, false otherwise.

Simple Animations

The following operation can be used to perform simple animations. One can animate
the movement of selected objects of a scene. This can be done in the following way: select
a number of objects in an edit scene; then start the animation by calling the animate
member function. The second parameter of this member function is an object anim of
type geowin_animation, the first parameter is the scene that will be animated. The object
anim has to be derived from the abstract base class geowin_animation. The derived class
has to overwrite some methods of the base class:

class geowin_animation {
public:
    virtual void init(const GeoWin&) { }
    virtual void finish(const GeoWin&) { }
virtual bool is_running(const GeoWin&) { return true; }
virtual point get_next_point(const GeoWin&) = 0;
virtual long get_next_action(const GeoWin&)
{ return GEOWIN_STOP_MOVE_SELECTED; }
};

At the start and at the end of an animation the member functions init and finish are called. The animation is stopped if is_running returns false. The member functions get_next_point and get_next_action specify the animation. get_next_point delivers the next point of the animation path. get_next_action currently can return two values: GEOWIN_MOVE_SELECTED (moves the selected objects of the scene) and GEOWIN_STOP_MOVE_SELECTED (stops the movement of the selected objects of the scene).

bool GW.animate(geo_scene sc, geowin_animation& anim)
starts animation anim for edit scene sc.

b) Window Operations

void GW.close() closes GW.

double GW.get_xmin() returns the minimal x-coordinate of the drawing area.

double GW.get_ymin() returns the minimal y-coordinate of the drawing area.

double GW.get_xmax() returns the maximal x-coordinate of the drawing area.

double GW.get_ymax() returns the maximal y-coordinate of the drawing area.

void GW.display(int x = window::center, int y = window::center)
opens GW at (x, y).

window& GW.get_window() returns a reference to the drawing window.

void GW.init(double xmin, double xmax, double ymin)
same as window::init(xmin, xmax, ymin, g).

void GW.init(double x1, double x2, double y1, double y2,
int r = GEOWIN_MARGIN)
inializes the window so that the rectangle with lower left corner \((x1 - r, y1 - r)\) and upper right corner \((x2 + r, y2 + r)\) is visible. The window must be open. GEOWIN_MARGIN is a default value provided by GeoWin.

void GW.redraw() redraws the contents of GW (all visible scenes).

int GW.set_cursor(int cursor_id = -1)
sets the mouse cursor to cursor_id.
bool GW.get_show_status()  
    return

bool GW.set_show_status(bool b)  
    display a status window (b=true) or not (b=false).  
The operation should be called before the first display - operation of GW.

bool GW.set_show_menu(bool v)  
    sets the visibility of the menu of GW to v.

void GW.set_menu_add_fcn(void (*mfcn)(window & W))  
    This handler function can be used to add own menus to the menu bar of a GeoWin.  It is called before the menu initialization of a GeoWin.  See the demo program geowin_gui for an example.

bool GW.set_show_file_menu(bool v)  
    sets the visibility of the file menu of GW to v.

bool GW.set_show_edit_menu(bool v)  
    sets the visibility of the edit menu of GW to v.

bool GW.set_show_scenes_menu(bool v)  
    sets the visibility of the scenes menu of GW to v.

bool GW.set_show_window_menu(bool v)  
    sets the visibility of the window menu of GW to v.

bool GW.set_show_options_menu(bool v)  
    sets the visibility of the options menu of GW to v.

bool GW.set_show_algorithms_menu(bool v)  
    sets the visibility of the algorithms menu of GW to v.

bool GW.set_show_help_menu(bool v)  
    sets the visibility of the help menu of GW to v.

void GW.init_menu(window * wptr = NULL)  
    initializes the menu of GW.  Normally you don’t have to call this operation directly, but if you want to add additional graphical elements like sliders or buttons to the window of GW you have to call init_window (with no parameters).  After that add the desired elements and then call edit or display.  See the demo programs for examples.

c) Scene and scene group Operations
18.8. GEOMETRY WINDOWS ( GEOWIN )

`geo_scene`  
`GW.get_scene_with_name(string nm)`
returns the scene with name `nm` or nil if there is no scene with name `nm`.

`void`  
`GW.activate(geo_scene sc)`
makes scene `sc` the active scene of `GW`.

`int`  
`GW.get_z_order(geo_scene sc)`
returns the z-coordinate of `sc`.

`int`  
`GW.set_z_order(geo_scene sc, int n)`
sets the z-coordinate of `sc` to `n` and returns its previous value.

In front of the scenes of a GeoWin object a so-called ”user layer” can store some geometric objects illustrating scenes. The following functions let you add some of these objects.

`void`  
`GW.add_user_layer_segment(const segment& s)`
adds segment `s` to the segments of the user layer.

`void`  
`GW.add_user_layer_circle(const circle& c)`
adds circle `c` to the circles of the user layer.

`void`  
`GW.add_user_layer_point(const point& p)`
adds point `p` to the points of the user layer.

`void`  
`GW.add_user_layer_rectangle(const rectangle& r)`
adds rectangle `r` to the rectangles of the user layer.

`void`  
`GW.remove_user_layer_objects()`
removes all objects of the user layer.

`void`  
`GW.set_draw_user_layer_fcn(void (*fcn)(GeoWin*))`
this function can be used for additional user-defined redraw after drawing the objects of the user layer.

`void`  
`GW.set_postscript_user_layer_fcn(void (*fcn)(GeoWin*, ps_file& ))`

`geo_scene`  
`GW.get_active_scene()`
returns the active scene of `GW`.

`bool`  
`GW.is_active(geo_scene sc)`
returns true if `sc` is an active scene in a GeoWin.

The following `get` and `set` operations can be used for retrieving and changing scene parameters. All `set` operations return the previous value.

`string`  
`GW.get_name(geo_scene sc)`
returns the `name` of scene `sc`. 
string

\[ GW\text{.}get\_name(geo\_scenegroup gs) \]

returns the name of scene group gs.

string

\[ GW\text{.}set\_name(geo\_scene sc, string nm) \]
gives scene sc the name nm. If there is already a scene with name nm, another name is constructed based on nm and is given to sc. The operation will return the given name.

color

\[ GW\text{.}get\_color(geo\_scene sc) \]
returns the boundary drawing color of scene sc.

color

\[ GW\text{.}set\_color(geo\_scene sc, color c) \]
sets the boundary drawing color of scene sc to c.

void

\[ GW\text{.}set\_color(geo\_scenegroup gs, color c) \]
sets the boundary drawing color of all scenes in group gs to c.

color

\[ GW\text{.}get\_selection\_color(geo\_scene sc) \]
returns the boundary drawing color for selected objects of scene sc.

color

\[ GW\text{.}set\_selection\_color(geo\_scene sc, color c) \]
sets the boundary drawing color for selected objects of scene sc to c.

void

\[ GW\text{.}set\_selection\_color(geo\_scenegroup gs, color c) \]
sets the boundary drawing color for selected objects of all scenes in gs to c.

color

\[ GW\text{.}get\_selection\_fill\_color(geo\_scene sc) \]
returns the fill color for selected objects of scene sc.

color

\[ GW\text{.}set\_selection\_fill\_color(geo\_scene sc, color c) \]
sets the fill color for selected objects of scene sc to c.

line_style

\[ GW\text{.}get\_selection\_line\_style(geo\_scene sc) \]
returns the line style for selected objects of scene sc.

line_style

\[ GW\text{.}set\_selection\_line\_style(geo\_scene sc, line\_style l) \]
sets the line style for selected objects of scene sc to l.

int

\[ GW\text{.}get\_selection\_line\_width(geo\_scene sc) \]
returns the line width for selected objects of scene sc.

int

\[ GW\text{.}set\_selection\_line\_width(geo\_scene sc, int w) \]
sets the line width for selected objects of scene sc to w.
18.8. GEOMETRY WINDOWS (GEOWIN)

```
color GW.get_fill_color(geo_scene sc)
returns the fill color of sc.

color GW.set_fill_color(geo_scene sc, color c)
sets the fill color of sc to c. Use color invisible to disable filling.

void GW.set_fill_color(geo_scenegroup gs, color c)
sets the fill color of all scenes in gs to c. Use color invisible to disable filling.

color GW.get_text_color(geo_scene sc)
returns the text color of sc.

color GW.set_text_color(geo_scene sc, color c)
sets the text color of sc to c.

void GW.set_text_color(geo_scenegroup gs, color c)
sets the text color of all scenes in gs to c.

int GW.get_line_width(geo_scene sc)
returns the line width of scene sc.

int GW.get_active_line_width(geo_scene sc)
returns the active line width of sc.

int GW.set_line_width(geo_scene sc, int w)
sets the line width for scene sc to w.

void GW.set_line_width(geo_scenegroup gs, int w)
sets the line width for all scenes in gs to w.

int GW.set_active_line_width(geo_scene sc, int w)
sets the active line width of scene sc to w.

void GW.set_active_line_width(geo_scenegroup gs, int w)
sets the active line width for all scenes in gs to w.

line_style GW.get_line_style(geo_scene sc)
returns the line style of sc.

line_style GW.set_line_style(geo_scene sc, line_style l)
sets the line style of scene sc to l.

void GW.set_line_style(geo_scenegroup gs, line_style l)
sets the line style of all scenes in gs to l.

bool GW.get_visible(geo_scene sc)
returns the visible flag of scene sc.
```
bool \( GW.set\_visible(geo\_scene \textit{sc}, \textit{bool} \textit{v}) \)
sets the visible flag of scene \( \textit{sc} \) to \( \textit{v} \).

void \( GW.set\_visible(geo\_scenegroup \textit{gs}, \textit{bool} \textit{v}) \)
sets the visible flag of all scenes in \( \textit{gs} \) to \( \textit{v} \).

void \( GW.set\_all\_visible(\textit{bool} \textit{v}) \)
sets the visible flag of all scenes that are currently in \( GW \) to \( \textit{v} \).

\textit{point\_style} \( GW.get\_point\_style(geo\_scene \textit{sc}) \)
returns the point style of \( \textit{sc} \).

\textit{point\_style} \( GW.set\_point\_style(geo\_scene \textit{sc}, \textit{point\_style} \textit{p}) \)
sets the point style of \( \textit{sc} \) to \( \textit{p} \)

void \( GW.set\_point\_style(geo\_scenegroup \textit{gs}, \textit{point\_style} \textit{p}) \)
sets the point style of all scenes in \( \textit{gs} \) to \( \textit{p} \)

bool \( GW.get\_cyclic\_colors(geo\_scene \textit{sc}) \)
returns the cyclic colors flag for editable scene \( \textit{sc} \).

bool \( GW.set\_cyclic\_colors(geo\_scene \textit{sc}, \textit{bool} \textit{b}) \)
sets the cyclic colors flag for editable scene \( \textit{sc} \). If the cyclic colors flag is set, the new inserted objects of the scene get color \( \text{counter}\%16 \), where \( \text{counter} \) is the object counter of the scene.

\textit{string} \( GW.get\_description(geo\_scene \textit{sc}) \)
returns the description string of scene \( \textit{sc} \).

\textit{string} \( GW.set\_description(geo\_scene \textit{sc}, \textit{string} \textit{desc}) \)
sets the description string of scene \( \textit{sc} \) to \( \textit{desc} \). The description string has the task to describe the scene in a more detailed way than the name of the scene does.

bool \( GW.get\_show\_orientation(geo\_scene \textit{sc}) \)
returns the show orientation/direction parameter of scene \( \textit{sc} \)

bool \( GW.set\_show\_orientation(geo\_scene \textit{sc}, \textit{bool} \textit{o}) \)
sets the show orientation/direction parameter of scene \( \textit{sc} \) to \( \textit{o} \).

\textit{void*} \( GW.get\_client\_data(geo\_scene \textit{sc}, \textit{int} \textit{i} = 0) \)
returns the \( \textit{i} \)-th client data pointer of of scene \( \textit{sc} \). \textit{Pre-condition:} \( \textit{i} < 16 \).
void* GW.set_client_data(geo_scene sc, void*p, int i = 0)

sets the i-th client data pointer of scene sc to p and returns its previous value. Precondition: i < 16.

void GW.set_handle_defining_points(geo_scene sc, geowin_defining_points gdp)

sets the attribute for handling of defining points of editable scene (*sc) to gdp. Options for gdp are geowin_show (show the defining points of all objects of the scene), geowin_hide (hide the defining points of all objects of the scene) and geowin_highlight (shows only the defining points of the object under the mouse-pointer).

gewin_defining_points GW.get_handle_defining_points(geo_scene sc)

returns the attribute for handling of defining points of editable scene (*sc).

The following operations can be used for getting/setting a flag influencing the behaviour of incremental update operations in result scenes. If update_state is true (default): if the first incremental operation returns false, incremental update loop will be left false: the incremental update loop will be executed until the end You can also set an update_limit for the incremental update operations. If a number of objects bigger than this limit will be added/deleted/changed, the incremental update will not be executed. Instead the ”normal” scene update operation will be used.

bool GW.get_incremental_update_state(geo_scene sc)

returns the incremental update flag of scene sc.

bool GW.set_incremental_update_state(geo_scene sc, bool us)

sets the incremental update flag of scene sc to us.

int GW.get_incremental_update_limit(geo_scene sc)

returns the incremental update limit of scene sc.

int GW.set_incremental_update_limit(geo_scene sc, int l)

sets the incremental update limit of scene sc to l.

It is not only possible to assign (graphical) attributes to a whole scene.

The following operations can be used to set/get individual attributes of objects in scenes. All set operations return the previous value. The first parameter is the scene, where the object belongs to. The second parameter is a generic pointer to the object or an iterator pointing to the position of the object in the container of a scene. Precondition: the object belongs to the scene (is in the container of the scene).

Note that you cannot use a pointer to a copy of the object.

The following example program demonstrates the setting of individual object attributes in an update member function of an update class:
#include <LEDA/graphics/geowin.h>
#include <LEDA/geo/rat_geo_alg.h>

using namespace leda;

class attr_update : public geowin_update<list<rat_point>, list<rat_circle> >
{
    void update(const list<rat_point>& L, list<rat_circle>& C)
    {
        GeoWin* GW_ptr = GeoWin::get_call_geowin();
        GeoBaseScene<list<rat_circle>>* aec =
            (GeoBaseScene<list<rat_circle>*>* GeoWin::get_call_scene());
        C.clear();
        if (! L.empty()) {
            ALL_EMPTY_CIRCLES(L,C);

            // now set some attributes
            list<rat_circle>::iterator it = C.begin();
            int cw=0;
            for (; it!=C.end(); it++) {
                GW_ptr->set_obj_fill_color(aec,it,color(cw % 15));
                GW_ptr->set_obj_color(aec,it,color(cw % 10));
                cw++;
            }
        }
    }
};

int main()
{
    GeoWin GW("All empty circles - object attribute test");

    list<rat_point> L;
    geo_scene input = GW.new_scene(L);
    GW.set_point_style(input, disc_point);

    attr_update aec_help;
    geo_scene aec = GW.new_scene(aec_help, input, string("All empty circles"));

    GW.set_all_visible(true);
    GW.edit(input);
    return 0;
}

template <class T>
color GW.get_obj_color(GeoBaseScene<T>* sc, void* adr)
    returns the boundary color of the object at (*adr).

template <class T>
color GW.get_obj_color(GeoBaseScene<T>* sc, typename T::iterator it)
    returns the boundary color of the object it points to.
color

\[ GW.set.obj.color(\text{GeoBaseScene}<T> \ast sc, \text{void} \ast adr, \text{color} \ c) \]

sets the boundary color of the object at \((*adr)\) to \(c\).

template <class T>

\[ GW.set.obj.color(\text{GeoBaseScene}<T> \ast sc, \text{typename} T::\text{iterator} \ it, \text{color} \ c) \]

sets the boundary color of the object \(it\) points to to \(c\).

template <class T>

\[ GW.get.obj.color(\text{GeoBaseScene}<T> \ast sc, \text{typename} T::\text{value}\_\text{type} \ & \ obj, \text{color} \& \ c) \]

if there is an object \(o\) in the container of scene \(sc\) with \(o == obj\) the boundary color of \(o\) is assigned to \(c\) and \(true\) is returned. Otherwise \(false\) is returned.

template <class T>

\[ GW.set.obj.fill.color(\text{GeoBaseScene}<T> \ast sc, \text{void} \ast adr) \]

returns the interior color of the object at \((*adr)\).

template <class T>

\[ GW.set.obj.fill.color(\text{GeoBaseScene}<T> \ast sc, \text{void} \ast adr, \text{color} \ c) \]

sets the interior color of the object at \((*adr)\) to \(c\).

template <class T>

\[ GW.get.obj.fill.color(\text{GeoBaseScene}<T> \ast sc, \text{typename} T::\text{value}\_\text{type} \ & \ obj, \text{color} \& \ c, \text{bool \ all = true}) \]

if there is an object \(o\) in the container of scene \(sc\) with \(o == obj\) the interior color of \(o\) is set to \(c\) and \(true\) will be returned. Otherwise \(false\) will be returned.

template <class T>

\[ GW.get.obj.line.style(\text{GeoBaseScene}<T> \ast sc, \text{void} \ast adr) \]

returns the line style of the object at \((*adr)\).
template <class T>
line_style GW.set_obj_line_style(GeoBaseScene<T> * sc, void * adr, line_style l)
sets the line style of the object at (*adr) to l.

template <class T>
bool GW.get_obj_line_style(GeoBaseScene<T> * sc, const typename T::value_type & obj, line_style & l)
if there is an object o in the container of scene sc with o == obj the line style of o is assigned to l and true is returned. Otherwise false is returned.

template <class T>
bool GW.set_obj_line_style(GeoBaseScene<T> * sc, const typename T::value_type & obj, line_style l, bool all = true)
if there is an object o in the container of scene sc with o == obj the line style of o is set to l and true will be returned. Otherwise false will be returned.

template <class T>
int GW.get_obj_line_width(GeoBaseScene<T> * sc, void * adr)
returns the line width of the object at (*adr).

template <class T>
int GW.set_obj_line_width(GeoBaseScene<T> * sc, void * adr, int w)
sets the line width of the object at (*adr) to w.

template <class T>
bool GW.get_obj_line_width(GeoBaseScene<T> * sc, const typename T::value_type & obj, int & l)
if there is an object o in the container of scene sc with o == obj the line width of o is assigned to l and true is returned. Otherwise false is returned.

template <class T>
bool GW.set_obj_line_width(GeoBaseScene<T> * sc, const typename T::value_type & obj, int l, bool all = true)
if there is an object o in the container of scene sc with o == obj the line width of o is set to l and true will be returned. Otherwise false will be returned.

template <class T>
string GW.get_obj_label(GeoBaseScene<T> * sc, void * adr)
returns the label of the object at (*adr).

template <class T>
string GW.get_obj_label(GeoBaseScene<T> * sc, typename T::iterator it)
returns the label of the object it points to.
18.8. GEOMETRY WINDOWS (GEOWIN)

template <class T>
string GW.set_obj_label(GeoBaseScene<T> * sc, void * adr, string lb)
    sets the label of the object at (*adr) to lb.

template <class T>
string GW.set_obj_label(GeoBaseScene<T> * sc, typename T::iterator it, string lb)
    sets the label of the object it points to to lb.

Object texts
The following operations can be used to add/retrieve objects of type geowin_text to objects in scenes. The class geowin_text is used to store graphical representations of texts. It stores a string (the text) and the following attributes:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>font_type</td>
<td>geowin_font_type</td>
<td>font type</td>
</tr>
<tr>
<td>size</td>
<td>double</td>
<td>font size</td>
</tr>
<tr>
<td>text_color</td>
<td>color</td>
<td>color of the text</td>
</tr>
<tr>
<td>user_font</td>
<td>string</td>
<td>font name (if font_type = user_font)</td>
</tr>
<tr>
<td>x_offset</td>
<td>double</td>
<td>offset in x-direction to drawing position</td>
</tr>
<tr>
<td>y_offset</td>
<td>double</td>
<td>offset in y-direction to drawing position</td>
</tr>
</tbody>
</table>

The enumeration type geowin_font_type has the following set of integral constants: roman_font, bold_font, italic_font, fixed_font and user_font.
The class geowin_text has the following constructors:

geowin_text(string t, double ox, double oy, geowin_font_type ft, double sz, string uf, color c = black);
geowin_text(string t, geowin_font_type ft, double sz);
geowin_text(string t);

The arguments are: t - the text, ox, oy - the x/y offsets, ft - the font type, sz - the font size, uf - the user font and c - the text color. If a text is associated with an object, it will be drawn centered at the center of the bounding box of the object translated by the x/y - offset parameters. Note that it is also possible to add texts to a whole scene and to instances of class GeoWin. Then the x/y - offset parameters specify the position (see add_text operation).

template <class T>
bool GW.get_obj_text(GeoBaseScene<T> * sc, void * adr, geowin_text& gt)
    Gets the text associated with the object at adr in the container of scene sc and assigns it to gt. If no text is associated with the object, false will be returned, otherwise true.
template <class T>
bool GW.get_obj_text(GeoBaseScene<T> * sc, typename T::iterator it, geowin_text& gt)
        // Gets the text associated with the object it points to and assigns it to gt. If no text is associated with the object, false will be returned, otherwise true.

template <class T>
void GW.set_obj_text(GeoBaseScene<T> * sc, void * adr, const geowin_text& gt)
        // Assigns gt to the object at adr in scene sc.

template <class T>
void GW.set_obj_text(GeoBaseScene<T> * sc, typename T::iterator it, const geowin_text& gt)
        // Assigns gt to the object it points to in scene sc.

template <class T>
void GW.reset_obj_attributes(GeoBaseScene<T> * sc)
        // deletes all individual attributes of objects in scene (*sc).

d) Input and Output Operations

void GW.read(geo_scene sc, istream& is)
        // reads the contents of sc from input stream is. Before the contents of sc is cleared.

void GW.write(geo_scene sc, ostream& os)
        // writes the contents of sc to output stream os.

void GW.write_active_scene(ostream& os)
        // writes the contents of the active scene of GW to output stream os.

e) View Operations

void GW.zoom_up()
        // The visible range is reduced to the half.

void GW.zoom_down()
        // The visible range is doubled.

void GW.fill_window()
        // changes window coordinate system, so that the objects of the currently active scene fill the window.

void GW.reset_window()
        // resets the visible range to the values that where current when constructing GW.

f) Parameter Operations
The following operations allow the set and retrieve the various parameters of GeoWin.
string $GW$.get_bg_pixmap() returns the name of the current background pixmap.

string $GW$.set_bg_pixmap(string pix_name)
changes the window background pixmap to pixmap with name pix_name. Returns the name of the previous background pixmap.

color $GW$.get_bg_color() returns the current background color.

color $GW$.set_bg_color(const color& c)
sets the background color to c and returns its previous value.

color $GW$.get_user_layer_color()
returns the current color of the user layer.

color $GW$.set_user_layer_color(const color& c)
sets the user layer color to c and returns its previous value.

int $GW$.get_user_layer_line_width()
returns the current line width of the user layer.

int $GW$.set_user_layer_line_width(int lw)
sets the user layer line width to lw and returns its previous value.

bool $GW$.get_show_grid() returns true, if the grid will be shown, false otherwise.

bool $GW$.set_show_grid(bool sh)
sets the show grid flag to sh and returns the previous value.

double $GW$.get_grid_dist() returns the grid width parameter.

double $GW$.set_grid_dist(double g)
sets the grid width parameter to g and returns the previous value.

grid_style $GW$.get_grid_style() returns the grid style parameter.

grid_style $GW$.set_grid_style(grid_style g)
sets the grid style parameter to g and returns the previous value.

bool $GW$.get_show_position()
returns true, if the mouse position will be shown, false otherwise.
bool GW.set_show_position(bool sp)
sets the show position flag to sp and returns the previous value.

The following operations set or return various parameters that are used in the three-dimensional output of GeoWin. The three-dimensional output can be started by pressing the Show D3 Output button in the Window menu.

bool GW.get_d3_elimination()
returns true, if elimination of hidden lines in the 3d-output mode is enabled, false otherwise.

bool GW.set_d3_elimination(bool b)
sets the d3_elimination flag of GW to b and returns its previous value.

bool GW.get_d3_solid()
return true, if faces in the 3d-output mode have to be drawn in different grey scales, false otherwise.

bool GW.set_d3_solid(bool b)
sets the d3_solid flag of GW to b and returns its previous value.

bool GW.get_d3_show_edges()
returns true, if the redraw of edges is enabled in the 3d-output mode, false otherwise.

bool GW.set_d3_show_edges(bool b)
sets the d3_show_edges flag of GW to b and returns its previous value.

**g) Handling of events**

GeoWin provides operations for changing its default handling of events. As in GraphWin (cf. Section 18.6) the user can define what action should follow a mouse or key event. Constants are defined as in GraphWin:

- A_LEFT (left mouse-button)
- A_MIDDLE (middle mouse-button)
- A_RIGHT (right mouse-button)
- A_SHIFT (shift-key)
- A_CTRL (control-key)
- A_ALT (alt-key)
- A_DOUBLE (double click)
• A_DRAG (button not released)
• A_IMMEDIATE (do it immediately without dragging or double click check)
• A_OBJECT (editable object at mouse position).

and can be combined with OR ( | ).

```c
void GW.set_action(long mask, geo_action f = 0)
set action on condition mask to f. geo_action is a
function of type void (*)(GeoWin&, const point&).
For f == 0 the corresponding action is deleted.

geo_action GW.get_action(long mask)
get action defined for condition mask.

void GW.reset_actions() set all actions to their default values.
```

Default values are defined as follows:

• A_LEFT or A_LEFT | A_OBJECT
  read a new object at mouse position.

• A_LEFT | A_DRAG
  scrolling the window.

• A_LEFT | A_DRAG | A_OBJECT
  move the object.

• A_LEFT | A_CTRL
  pin current scene at mouse position or delete the pin point if it is currently there.

• A_MIDDLE | A_OBJECT
  toggle the selection state of the object at mouse position.

• A_MIDDLE | A_DRAG
  toggle the selection state of the objects in the dragging area.

• A_RIGHT | A_IMMEDIATE
  set the options of the currently active scene.

• A_RIGHT | A_IMMEDIATE | A_OBJECT
  opens a menu for the object at mouse position.

```c
void GW.clear_actions() clears all actions.
```

**Scene events**

The following event handling functions can be set for edit scenes:
• Pre add handler
• Pre add change handler
• Post add handler
• Pre delete handler
• Post delete handler
• Start, Pre, Post and End change handler

The add handlers will be called when a user tries to add an object to an edit scene in GeoWin, the delete handlers will be called when the user tries to delete an object and the change handlers will be called when the user tries to change an object (for instance by moving it). The templated set operations for setting handlers uses member templates. If your compiler does not support member templates, you should use instead the templated functions `geowin_set_HANDLER`, where `HANDLER` is one the following handlers. All handling functions get as the first parameter a reference to the `GeoWin`, where the scene belongs to.

```cpp
template <class T, class F>
bool GW.set_pre_add_handler(GeoEditScene<T>* sc, F handler)
sets the handler that is called before an object is added to (*sc). `handler` must have type `bool (*handler)(GeoWin&, const T::value_type&)`. `handler` gets a reference to the added object. If `handler` returns false, the object will not be added to the scene.

template <class T, class F>
bool GW.set_post_add_handler(GeoEditScene<T>* sc, F handler)
sets the handler that is called after an object is added to (*sc). `handler` must have type `void (*handler)(GeoWin&, const T::value_type&)`. `handler` gets a reference to the added object.

template <class T, class F>
bool GW.set_pre_del_handler(GeoEditScene<T>* sc, F handler)
sets the handler that is called before an object is deleted from (*sc). `handler` must have type `bool (*handler)(GeoWin&, const T::value_type&)`. `handler` gets a reference to the added object. If `handler` returns true, the object will be deleted, if `handler` returns false, the object will not be deleted.


\begin{verbatim}
18.8. GEOMETRY WINDOWS (GEOWIN)

\begin{verbatim}
template <class T, class F>
bool GW.set_post_del_handler(GeoEditScene<T> * sc, F handler)
    sets the handler that is called after an object is deleted from (*sc). handler must have type void (*handler)(GeoWin&, const T::value_type &).

template <class T, class F>
bool GW.set_start_change_handler(GeoEditScene<T> * sc, F handler)
    sets the handler that is called when a geometric object from (*sc) starts changing (for instance when you move it or rotate it). handler must have type bool (*handler)(GeoWin&, const T::value_type &). The handler function gets a reference to the object.

template <class T, class F>
bool GW.set_pre_move_handler(GeoEditScene<T> * sc, F handler)
    sets the handler that is called before every move operation. handler must have type bool (*handler)(GeoWin&, const T::value_type &, double x, double y). The handler gets as the second parameter a reference to the object, as the third parameter and fourth parameter the move vector. If the handler returns true, the change operation will be executed, if the handler returns false, it will not be executed.

template <class T, class F>
bool GW.set_post_move_handler(GeoEditScene<T> * sc, F handler)
    sets the handler that is called after every move operation. handler must have type void (*handler)(GeoWin&, const T::value_type &, double x, double y). The handler gets as the second parameter a reference to the object, as the third parameter and fourth parameter the move vector.

template <class T, class F>
bool GW.set_pre_rotate_handler(GeoEditScene<T> * sc, F handler)
    sets the handler that is called before every rotate operation. handler must have type bool (*handler)(GeoWin&, const T::value_type &, double x, double y, double a). If the handler returns true, the rotate operation will be executed, if the handler returns false, it will not be executed.

\end{verbatim}
\end{verbatim}
bool GW::set_post_rotate_handler(GeoEditScene<T> * sc, F handler)

sets the handler that is called after every rotate operation. handler must have type
void (*handler)(GeoWin&, const T::value_type&, double x, double y, double a).

template <class T, class F>
bool GW::set_end_change_handler(GeoEditScene<T> * sc, F handler)

sets the handler that is called when a geometric object from (*sc) ends changing. handler gets the object as the second parameter. handler must have type
void (*handler)(GeoWin&, const T::value_type &).

**Generator functions:** The following operation can be used to set a generator function for an edit scene. The operation uses member templates. If your compiler does not support member templates, you should use instead the templated function `geowin_set_generate_fcn`.

```cpp
template <class T>
bool GW::set_generate_fcn(GeoEditScene<T> * sc, void (*f)(GeoWin& gw, T& l))

sets the generator function for edit scene (*sc). The function gets the GeoWin where (*sc) belongs to and a reference to the container L of (*sc). The function should write the generated objects to L.
```

**Editing of objects in a scene:** It is possible to edit single objects in an editable scene. For this purpose an `edit_object`-function can be set for editable scenes. This function has type

```cpp
void (*f)(GeoWin& gw, T& obj, int nr)
```

where gw is the GeoWin-object where the scene belongs to, obj is a reference to the object that will be edited and nr is the edit mode of the scene.

```cpp
template <class T, class T2>
bool GW::set_edit_object_fcn(GeoEditScene<T> * sc, T2 f)

sets the edit object - function of scene sc to f.
```

```cpp
template <class T>
void* GW::get_edit_object_fcn(GeoEditScene<T> * sc)

returns the edit object - function of scene sc.
```

**Transformation objects:**

`GeoWin` supports affine transformations of selected objects in editable scenes for the LEDA rat- and float-kernel classes. The used transformation classes are `rat_transform` and `transform` respectively. The following class templates can be used to instantiate transformation objects. They are derived from type `geowin_transform`. 
where `KERNEL_CLASS` is a class of the LEDA rat- or float-kernel. The default is that no transformation objects are associated with editable scenes.

```cpp
template <class S, class GeoObj>
void GW.set_transform(GeoEditScene<S> * sc,
                      geowin_transform<GeoObj>& trans)
    makes `trans` the transformation object of edit scene `sc`.
```

**Input objects:** The following operation can be used to set an input object for an edit scene. The operation uses member templates. If your compiler does not support member templates, you should use instead the templated functions prefixed with `geowin`. A `GeoInputObject<GeoObj>` has the following virtual functions:

```cpp
void operator( )(GeoWin& gw, list<GeoObj>& L);
```
This virtual function is called for the input of objects. The new objects have to be returned in `L`.

```cpp
void options(GeoWin& gw);
```
This function is called for setting options for the input object.

```cpp
template <class T>
bool GW.set_input_object(GeoEditScene<T> * sc,
                          const GeoInputObject<>::value_type& obj,
                          string name)
    sets the input object `obj` for edit scene (*sc). The function gets the GeoWin where (*sc) belongs to and a reference to a list `L`. The function must write the new objects to `L`.
```

```cpp
template <class T>
bool GW.add_input_object(GeoEditScene<T> * sc,
                          const GeoInputObject<>::value_type& obj,
                          string name)
    adds the input object `obj` to the list of available input objects of edit scene (*sc) without setting `obj` as input object.
```

```cpp
template <class T>
```
void GW.set_draw_object_fcn(GeoBaseScene<T>* sc,
    window& (*fcn)(window&,
    const typename T::value_type&, int w))
sets a function fcn for scene (*sc) that will be called
for drawing the objects of scene (*sc). If no such
function is set (the default), the output operator is
used.

void GW.set_activate_handler(geo_scene sc, void (*f)(geo_scene))
sets a handler function f that is called with sc as parameter when the user activates sc.

void GW.set_edit_loop_handler(bool (*f)(const GeoWin& gw))
sets a handler function f that is called periodically in
the interactive mode. If this handler returns true, we
will leave the interactive mode.

void GW.set_quit_handler(bool (*f)(const GeoWin& gw))
sets a handler function f that is called when the user
clicks the quit menu button. f should return true for
allowing quitting, false otherwise.

void GW.set_done_handler(bool (*f)(const GeoWin& gw))
sets a handler function f that is called when the user
clicks the done menu button. f should return true for
allowing quitting, false otherwise.

int GW.set_edit_mode(geo_scene sc, int emode)
sets the edit mode of scene sc to emode.

int GW.get_edit_mode(geo_scene sc)
return the edit mode of scene sc.

h) Scene group Operations
GeoWin can manage scenes in groups. It is possible to add and remove scenes to/from
groups. Various parameters and dependences can be set for whole groups. Note that
geo_scenegroup is a pointer to a scene group.

ggeo_scenegroup GW.new_scenegroup(string name)
Creates a new scene group with name name and re-
turns a pointer to it.

ggeo_scenegroup GW.new_scenegroup(string name, const list<geo_scene>& LS)
Creates a new scene group name and adds the scenes
in LS to this group.

void GW.insert(geo_scenegroup gs, geo_scene sc)
adds sc to scene group gs.
bool GW.del(geo_scenegroup gs, geo_scene sc)
    removes sc from scene group gs and returns true, if
    the operation was succesful (false: sc was not in gs).

i) Further Operations

    int GW.set_button_width(int w)
    sets the width of the scene visibility buttons in GW
    and returns the previous value.

    int GW.set_button_height(int h)
    sets the height of the scene visibility buttons in GW
    and returns the previous value.

You can associate a) buttons with labels or b) bitmap buttons with the visibility of a
scene in GeoWin. You cannot use a) and b) at the same time. The following operations
allow you to use add such visibility buttons to GeoWin. Note that before setting bitmap
buttons with the set_bitmap operation you have to set the button width and height.

    void GW.set_label(geo_scene sc, string label)
    associates a button with label label with the visibility
    of scene sc.

    void GW.set_bitmap(geo_scene sc, unsigned char * bitmap)
    associates a button with bitmap bitmap with the vis-
    ibility of scene sc.

    void GW.add_scenes_buttons(const list<geo_scene>& Ls, const list<string>& Ln)
    add a multiple choice panel for visibility of the scenes
    in Ls to GW. The button for the n-th scene in Ls
    gets the n-th label in Ln.

    void GW.add_scenes_buttons(const list<geo_scene>& Ls, int w, int h, unsigned char ** bm)
    add a multiple choice panel for visibility of the scenes
    in Ls to GW. The button for the n-th scene in Ls
    gets the n-th bitmap in bm. The bitmaps have width
    w and height h.

    list<geo_scene> GW.get_scenes()
    returns the scenes of GW.

    list<geo_scenegroup> GW.get_scenegroups()
    returns the scene groups of GW.

    list<geo_scene> GW.get_scenes(geo_scenegroup gs)
    returns the scenes of group gs.

    list<geo_scene> GW.get_visible_scenes()
    returns the visible scenes of GW.
void GW.add_dependence(geo_scene sc1, geo_scene sc2)
   makes sc2 dependent from sc1. That means that sc2 will be updated when the contents of sc1 changes.

void GW.del_dependence(geo_scene sc1, geo_scene sc2)
   deletes the dependence of scene sc2 from sc1.

void GW.set_frame_label(const char * label)
   makes label the frame label of GW.

int GW.open_panel(panel& P)
   displays panel P centered on the drawing area of GW, disables the menu bar of GW and returns the result of P.open( ).

void GW.add_text(const geowin_text& gt)
   adds a text gt to GW.

void GW.remove_texts( ) removes all texts from GW (but not from the scenes of GW).

void GW.add_text(geo_scene sc, const geowin_text& gt)
   adds a text gt to scene sc.

void GW.remove_texts(geo_scene sc)
   removes all texts from scene sc.

void GW.enable_menus( ) enables the menus of GW.

void GW.disable_menus( ) disables the menus of GW, but not the User menu.

double GW.version( ) returns the GeoWin version number.

void GW.message(string msg)
   displays message msg on top of the drawing area. If msg is the empty string, a previously written message is deleted.

void GW.msg_open(string msg)
   displays message msg in the message window of GW. If the message window is not open, it will be opened.

void GW.msg_close( ) closes the message window.

void GW.msg_clear( ) clears the message window.
void GW.set_d3_fcn(geo_scene sc, void (*f)(geo_scene gs, d3_window & W, GRAPH<d3_point, int>& H))

sets a function for computing 3d output. The parameters of the function are the geo_scene for that it will be set and a function pointer. The function f will get the scene for that it was set and the reference to a d3_window that will be the output window.

D3_FCN GW.get_d3_fcn(geo_scene sc)

returns the function for computing 3d output that is set for scene sc. The returned function has pointer type void (*)(geo_scene, d3_window&, GRAPH<d3_point, int>&).

GeoWin can be pinned at a point in the plane. As standard behavior it is defined that moves of geometric objects will be rotations around the pin point.

bool GW.get_pin_point(point& p)

returns the pin point in p if it is set.

void GW.set_pin_point(point p)

sets the pin point to p.

void GW.del_pin_point()

deletes the pin point.

void GW.add_help_text(string name)

adds the help text contained in name.hlp with label name to the help menu of the main window. The file name.hlp must exist either in the current working directory or in $LEDAROOT/incl/Help. Note that this operation must be called before gw.display().

void GW.add_special_help_text(string name, bool auto_display = false)

adds one help text contained in name.hlp to the menu of the main window. The file name.hlp must exist either in the current working directory or in $LEDAROOT/incl/Help. Note that this operation must be called before gw.display(). If auto_display is true, this help text will be displayed, when the main window is displayed.

template <class T>
int GW.get_limit(GeoEditScene<T> * es)

returns the limit of edit scene es (a negative number will be returned, if there is no limit).

template <class T>
The templated add_user_call operation uses member templates. If your compiler does not support member templates, you should use instead the templated function geowin_add_user_call with GW as an additional first parameter.

```
template <class F>
void GW.add_user_call(string label, F f)
```

adds a menu item label to the "User" menu of GW.

The user defined function `void geo_call(GeoWin&, F, string)` is called whenever this menu button was pressed with parameters GW, f and label. This menu definition has to be finished before GW is opened.

Import- and export objects can be used to import and export the contents of scenes in various formats.

The classes `geowin_import` and `geowin_export` are used for implementing import- and export objects. The classes `geowin_import` and `geowin_export` have virtual ( ) - operators:

```
virtual void operator() (geo_scene sc, string filename)
```

This virtual operator can be overwritten in derived classes to provide import and export functionality for own formats. The first parameter is the scene `sc` that will be used as source for the output or target for the input. The second parameter `filename` is the name of the input (import objects) or output (export objects) file.

```
void GW.add_import_object(geo_scene sc, geowin_import& io, string name, string desc)
```

Adds an import object `io` to scene `sc`. The import object gets the name `name` and the description `desc`.

```
void GW.add_export_object(geo_scene sc, geowin_export& eo, string name, string desc)
```

Adds an export object `eo` to scene `sc`. The export object gets the name `name` and the description `desc`.

4. Non-Member Functions

```
GeoWin* get_geowin(geo_scene sc)
```

returns a pointer to the GeoWin of `sc`.

```
template <class CONTAINER>
bool get_objects(geo_scene sc, CONTAINER& c)
```

If the contents of scene `sc` matches type `CONTAINER`, then the contents of scene `sc` is copied to `c`. 

```
18.9 Windows for 3d visualization (d3_window)

1. Definition

The data type d3_window supports three-dimensional visualization. It uses a LEDA window to visualize and animate three-dimensional drawings of graph. For this purpose we need to assign positions in 3d space to all nodes of the graph (see init-operations and set_position-operation). The edges of the visualized graph are drawn as straight-line-segments between the 3d positions of their source and target nodes. Note all edges of the graph must have a reversal edge.

If the graph to be shown is a planar map the faces can be shaded in different grey scales (if the solid flag is true).

The graph can be drawn with the draw-operation and animated with the move-operation. The draw-operation draws a frontal projection of the graph on the output window. The move-operation starts a simple animation mode. First it draws the graph, then it rotates it (the rotation depends on the x_rotation and y_rotation flags and the mouse position) and finally returns the pressed mouse button.

Every object of type d3_window maintains a set of parameters:

- x_rotation (type bool); if true, rotation about the x-axis is enabled during a move operation
- y_rotation (type bool); if true, rotation about the y-axis is enabled during a move operation
- elim (type bool); if true, hidden lines will be eliminated
- solid (type bool); if true, faces have to be drawn in different grey scales
- draw_edges (type bool) enables/disables the redraw of edges
- message (type string) is the message that will be displayed on top of the drawing area of the output window

In addition, a d3_window stores information assigned to the nodes and edges of the visualized graph.

- color (type color) information for nodes and edges
- position (three-dimensional vectors) information for the nodes
- arrow (type bool) information for the edges (define whether or not edges have to be drawn as arrows)
#include <LEDA/graphics/d3_window.h>

2. Creation

\texttt{d3\_window} \texttt{D(window\& W, const graph\& G);}  

creates an instance \texttt{D} of the data type \texttt{d3\_window}. The output window of \texttt{D} is \texttt{W}. The visualized graph is \texttt{G}.

\texttt{d3\_window} \texttt{D(window\& W, const graph\& G, const node\_array<\texttt{vector}>\& pos);}  

creates an instance \texttt{D} of the data type \texttt{d3\_window}. The output window of \texttt{D} is \texttt{W}. The visualized graph is \texttt{G}. The positions of the nodes are given in \texttt{pos}. \textbf{Precondition}: the vectors in \texttt{pos} are three-dimensional.

\texttt{d3\_window} \texttt{D(window\& W, const graph\& G, const node\_array<\texttt{rat\_vector}>\& pos);}  

creates an instance \texttt{D} of the data type \texttt{d3\_window}. The output window of \texttt{D} is \texttt{W}. The visualized graph is \texttt{G}. The positions of the nodes are given in \texttt{pos}. \textbf{Precondition}: the vectors in \texttt{pos} are three-dimensional.

3. Operations

\texttt{void} \texttt{D.init(const node\_array<\texttt{vector}>\& pos)}  

initializes \texttt{D} by setting the node positions of the visualized graph to the positions given in \texttt{pos}. \textbf{Precondition}: the vectors in \texttt{pos} are three-dimensional.

\texttt{void} \texttt{D.init(const node\_array<\texttt{rat\_vector}>\& pos)}  

initializes \texttt{D} by setting the node positions of the visualized graph to the positions given in \texttt{pos}. \textbf{Precondition}: the vectors in \texttt{pos} are three-dimensional.

\texttt{void} \texttt{D.init(const graph\& G, const node\_array<\texttt{vector}>\& pos)}  

initializes \texttt{D} by setting the visualized graph to \texttt{G} and the node positions of the visualized graph to the positions given in \texttt{pos}. \textbf{Precondition}: the vectors in \texttt{pos} are three-dimensional.

\texttt{void} \texttt{D.draw()}  

draws the contents of \texttt{D} (see also \textit{Definition}).
18.9. WINDOWS FOR 3D VISUALIZATION (D3_WINDOW)

```c
int D.move()  
```

animates the contents of D until a button is pressed and returns the pressed mouse button. If the movement is stopped or no mouse button is pressed, `NO_BUTTON` will be returned, else the number of the pressed mouse button will be returned (see also Definition and the `get_mouse` operation of the `window` data type).

```c
int D.get_mouse()  
```

does the same as `move`.

```c
int D.read_mouse()  
```

calls `move` as long as `move` returns `NO_BUTTON`. Else the movement is stopped, and the number of the pressed mouse button is returned.

```c
void D.set_position(node v, double x, double y, double z)  
```

sets the position of node v in the visualized graph D to (x, y, z).

Get- and set-operations

The following operations can be used to get and set the parameters of D. The set-operations return the previous value of the parameter.

```c
bool D.get_x_rotation()  
```

returns `true`, if D has rotation about the x-axis enabled, `false` otherwise.

```c
bool D.get_y_rotation()  
```

returns `true`, if D has rotation about the y-axis enabled, `false` otherwise.

```c
bool D.set_x_rotation(bool b)  
```

enables (disables) rotation about the x-axis.

```c
bool D.set_y_rotation(bool b)  
```

enables (disables) rotation about the y-axis.

```c
bool D.get_elim()  
```

returns the hidden line elimination flag.

```c
bool D.set_elim(bool b)  
```

sets the hidden line elimination flag to b. If b is `true`, hidden lines will be eliminated, if b is `false`, hidden lines will be shown.

```c
bool D.get_solid()  
```

returns the `solid` flag of D.

```c
bool D.set_solid(bool b)  
```

sets the `solid` flag of D to b. If b is `true` and the current graph of D is a planar map, its faces will be painted in different grey scales, otherwise the faces will be painted white.

```c
bool D.get_draw_edges()  
```

return `true`, if edges will be drawn, `false` otherwise.

```c
bool D.set_draw_edges(bool b)  
```

enables (disables) the redraw of the edges of D.

```c
string D.get_message()  
```

returns the message that will be displayed on top of the drawing area of the window.
string $D.set_message(string msg)$ sets the message that will be displayed on top of the drawing area of the window to $msg$.

void $D.set_node_color(color c)$ sets the color of all nodes of $D$ to $c$.

void $D.set_edge_color(color c)$ sets the color of all edges of $D$ to $c$.

color $D.get_color(node v)$ returns the color of node $v$.

color $D.set_color(node v, color c)$ sets the color of node $v$ to $c$.

color $D.get_color(edge e)$ returns the color of edge $e$.

color $D.set_color(edge e, color c)$ sets the color of edge $e$ to $c$.

bool $D.get_arrow(edge e)$ returns $true$, if $e$ will be painted with an arrow, $false$ otherwise.

bool $D.set_arrow(edge e, bool ar)$ if $ar$ is $true$, $e$ will be painted with an arrow, otherwise without an arrow.

void $D.get_d2_position(node_array<point>& d2pos)$ returns the two-dimensional positions of the nodes of the graph of $D$ in $d2pos$. 
Chapter 19

Implementations

19.1 List of data structures

This section lists the data structures for dictionaries, dictionary arrays, priority queues, and geometric data types currently contained in LEDA. For each of the data structures its name and type, the list of LEDA data types it can implement, and a literature reference are given. Before using a data structures $xyz$ the corresponding header file `<LEDA/impl/xyz.h>` has to be included (cf. section 2.2 for an example).

19.1.1 Dictionaries

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Types</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>ab_tree</td>
<td>a-b tree</td>
<td>dictionary, d_array, sortseq</td>
<td>[12]</td>
</tr>
<tr>
<td>avl_tree</td>
<td>AVL tree</td>
<td>dictionary, d_array</td>
<td>[5]</td>
</tr>
<tr>
<td>bb_tree</td>
<td>BB[$\alpha$] tree</td>
<td>dictionary, d_array, sortseq</td>
<td>[13]</td>
</tr>
<tr>
<td>ch_hashing</td>
<td>hashing with chaining</td>
<td>h_array</td>
<td>[58]</td>
</tr>
<tr>
<td>dp_hashing</td>
<td>dyn. perf. hashing</td>
<td>h_array</td>
<td>[24], [88]</td>
</tr>
<tr>
<td>pers_tree</td>
<td>persistent tree</td>
<td>p_dictionary</td>
<td>[25]</td>
</tr>
<tr>
<td>rb_tree</td>
<td>red-black tree</td>
<td>dictionary, d_array, sortseq</td>
<td>[42]</td>
</tr>
<tr>
<td>rs_tree</td>
<td>rand. search tree</td>
<td>dictionary, d_array, sortseq</td>
<td>[2]</td>
</tr>
<tr>
<td>skiplist</td>
<td>skip lists</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

19.1.2 Priority Queues

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Types</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>f_heap</td>
<td>Fibonacci heap</td>
<td>priority_queue</td>
<td>[36]</td>
</tr>
<tr>
<td>p_heap</td>
<td>pairing heap</td>
<td>priority_queue</td>
<td>[84]</td>
</tr>
<tr>
<td>k_heap</td>
<td>k-nary heap</td>
<td>priority_queue</td>
<td>[58]</td>
</tr>
<tr>
<td>m_heap</td>
<td>monotonic heap</td>
<td>priority_queue</td>
<td>[58]</td>
</tr>
<tr>
<td>eb_tree</td>
<td>Emde-Boas tree</td>
<td>priority_queue</td>
<td>[29], [88]</td>
</tr>
</tbody>
</table>

19.1.3 Geometry

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Types</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>range_tree</td>
<td>range tree</td>
<td>d2_dictionary, point_set</td>
<td>[90], [57]</td>
</tr>
<tr>
<td>seg_tree</td>
<td>segment tree</td>
<td>seg_set</td>
<td>[8], [27]</td>
</tr>
<tr>
<td>ps_tree</td>
<td>priority search tree</td>
<td></td>
<td>[60]</td>
</tr>
<tr>
<td>iv_tree</td>
<td>interval tree</td>
<td>interval_set</td>
<td>[59], [27]</td>
</tr>
<tr>
<td>delaunay_tree</td>
<td>delaunay tree</td>
<td>point_set</td>
<td>[22]</td>
</tr>
</tbody>
</table>
19.2 User Implementations

In addition to the data structures listed in the previous section user-defined data structures can also be used as actual implementation parameters provided they fulfill certain requirements.

19.2.1 Dictionaries

Any class `dic_impl` that provides the following operations can be used as actual implementation parameter for the `dictionary< K,I,dic_impl>` and the `d_array< I,E,dic_impl>` data types (cf. sections 8.1 and 8.2).

```cpp
class dic_impl {
    virtual int cmp(GenPtr, GenPtr) const = 0;
    virtual int int_type() const = 0;
    virtual void clear_key(GenPtr&) const = 0;
    virtual void clear_inf(GenPtr&) const = 0;
    virtual void copy_key(GenPtr&) const = 0;
    virtual void copy_inf(GenPtr&) const = 0;

public:
    typedef ... item;
    dic_impl();
    dic_impl(const dic_impl&);
    virtual ~dic_impl();

    dic_impl& operator=(const dic_impl&);

    GenPtr key(dic_impl_item) const;
    GenPtr inf(dic_impl_item) const;
    dic_impl_item insert(GenPtr,GenPtr);
    dic_impl_item lookup(GenPtr) const;
    dic_impl_item first_item() const;
    dic_impl_item next_item(dic_impl_item) const;
    dic_impl_item item(void* p) const
    { return dic_impl_item(p); }

    void change_inf(dic_impl_item,GenPtr);
    void del_item(dic_impl_item);
    void del(GenPtr);
```
void clear();

int size() const;
};
19.2.2 Priority Queues

Any class `prio_impl` that provides the following operations can be used as actual implementation parameter for the `priority_queue< K,I,prio_impl>` data type (cf. section 9.1).

```cpp
class prio_impl {

    virtual int cmp(GenPtr, GenPtr) const = 0;
    virtual int int_type() const = 0;
    virtual void clear_key(GenPtr&) const = 0;
    virtual void clear_inf(GenPtr&) const = 0;
    virtual void copy_key(GenPtr&) const = 0;
    virtual void copy_inf(GenPtr&) const = 0;

public:

    typedef ... item;

    prio_impl();
    prio_impl(int);
    prio_impl(int,int);
    prio_impl(const prio_impl&);
    virtual ~prio_impl();

    prio_impl& operator=(const prio_impl&);

    prio_impl_item insert(GenPtr,GenPtr);
    prio_impl_item find_min() const;
    prio_impl_item first_item() const;
    prio_impl_item next_item(prio_impl_item) const;

    prio_impl_item item(void* p) const
    { return prio_impl_item(p); }

    GenPtr key(prio_impl_item) const;
    GenPtr inf(prio_impl_item) const;

    void del_min();
    void del_item(prio_impl_item);
    void decrease_key(prio_impl_item,GenPtr);
    void change_inf(prio_impl_item,GenPtr);
    void clear();

    int size() const;
};
```
19.2.3 Sorted Sequences

Any class `seq_impl` that provides the following operations can be used as actual implementation parameter for the `_sortseq<K,I,seq_impl>` data type (cf. section 8.8).

class seq_impl {
    virtual int cmp(GenPtr, GenPtr) const = 0;
    virtual int int_type() const = 0;
    virtual void clear_key(GenPtr&) const = 0;
    virtual void clear_inf(GenPtr&) const = 0;
    virtual void copy_key(GenPtr&) const = 0;
    virtual void copy_inf(GenPtr&) const = 0;

public:

typedef ... item;

    seq_impl();
    seq_impl(const seq_impl&);
    virtual ~seq_impl();

    seq_impl& operator=(const seq_impl&);
    seq_impl& conc(seq_impl&);

    seq_impl_item insert(GenPtr, GenPtr);
    seq_impl_item insert_at_item(seq_impl_item, GenPtr, GenPtr);
    seq_impl_item lookup(GenPtr) const;
    seq_impl_item locate(GenPtr) const;
    seq_impl_item locate_pred(GenPtr) const;
    seq_impl_item succ(seq_impl_item) const;
    seq_impl_item pred(seq_impl_item) const;
    seq_impl_item item(void* p) const
    { return seq_impl_item(p); }

    GenPtr key(seq_impl_item) const;
    GenPtr inf(seq_impl_item) const;

    void del(GenPtr);
    void del_item(seq_impl_item);
    void change_inf(seq_impl_item, GenPtr);
    void split_at_item(seq_impl_item, seq_impl&, seq_impl&);
    void reverse_items(seq_impl_item, seq_impl_item);
    void clear();

    int size() const;
};
Appendix A

Technical Information

This chapter provides information about installation and usage of LEDA, the interaction with other software packages, and an overview of all currently supported system platforms.

A.1 LEDA Library and Packages

The implementations of most LEDA data types and algorithms are precompiled and contained in one library libleda that can be linked with C++ application programs.

LEDA is available either as source code package or as object code package for the platforms listed in Section Platforms. Information on how to obtain LEDA can be found at http://www.algorithmic-solutions.com/enleda.html

Sections Source Contents ff. describe how to compile the LEDA libraries in the source code package for Unix (including Linux and CygWin) and Microsoft Windows. Section UnixObjectCodePackage and Section WinObjectCodePackage describe the installation and usage of the object code packages for Unix and Windows, respectively.

A.2 Contents of a LEDA Source Code Package

The main directory of the GUI source code package should contain at least the following files and subdirectories:
A.3 Source Code on UNIX Platforms

Source Code Configuration on UNIX

1. Go to the LEDA main directory.

2. Type: `lconfig <cc> [static | shared]`

   where `<cc>` is the name (or command) of your C++ compiler and the optional second parameter defines the kind of libraries to be generated. Note that in the current release shared libraries can be made only under SunOS 5.x (Solaris) and some Linux, distributions.

   Examples: `lconfig CC, lconfig g++, lconfig sunpro shared`

   `lconfig` without arguments prints a list of known compilers.
   If your compiler is not in the list you might have to edit the `<LEDA/sys/unix.h>` header file.

LEDA Compilation on UNIX

Type `make` for building the object code library

    libleda.a (libleda.so) main library

(libleda.so if shared libraries are used)

Now follow the instructions given in Section UnixObjectCodePackage.
A.4 Source Code on Windows with MS Visual C++

Source Code Configuration for MS Visual C++

1. Setting the Environment Variables for Visual C++:
   The compiler `CL.EXE` and the linker `LINK.EXE` require that the environment variables `PATH`, `INCLUDE`, and `LIB` have been set properly.
   When Visual C++ was installed, setup created a batch file, `VCVARS32.BAT` for MS Visual Studio 6 and `VSVARS32.BAT` for MS Visual C++ .NET, containing the commands to modify `PATH`, `LIB`, and `INCLUDE`. Run `VCVARS32.BAT` (`VSVARS32.BAT`) before you compile LEDA as follows.
   
   (a) At the command prompt, change to the `\bin` subdirectory of your Visual C++ installation (e.g., `c:\programs\microsoft visual studio\vc98\bin`).
   
   (b) Type `VCVARS32` (`VSVARS32`).

2. Go to the LEDA main directory (at the command prompt used above).

3. Type: `lconfig [msc | msc6] [dll] [ ml | mld | md | mdd | mt | mt0 ]`

**Remark:** When using MSVC++ .NET you have to choose msc, when using MSVC++ 6.0 you have to choose msc6 in the command given above. When building an application with LEDA and MS Visual Studio 6 or MS Visual C++ .NET, the LEDA library you use depends on the Microsoft C runtime library you intend to link with. Your application code and LEDA both must be linked to the same Microsoft C runtime library; otherwise serious linker or runtime errors may occur. The Microsoft C runtime libraries are related to the compiler options as follows:

<table>
<thead>
<tr>
<th>C Runtime Library</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIBC.LIB</td>
<td>-ML</td>
</tr>
<tr>
<td>LIBCD.LIB</td>
<td>-MLd</td>
</tr>
<tr>
<td>LIBCMT.LIB</td>
<td>-MT</td>
</tr>
<tr>
<td>LIBCMTD.LIB</td>
<td>-MTd</td>
</tr>
<tr>
<td>MSVCRT.LIB</td>
<td>-MD</td>
</tr>
<tr>
<td>MSVCRTD.LIB</td>
<td>-MDd</td>
</tr>
</tbody>
</table>

In order to get the suitable Libs or DLL please choose the corresponding option in the call of `lconfig`.

LEDA Compilation with MS Visual C++

Type `make lib` for building the object code libraries.
APPENDIX A. TECHNICAL INFORMATION

static: 
libL.lib basic library
libG.lib graph library
libP.lib d2-geo library
libD3.lib d3-geo library
libGeoW.lib GeoWin library
libW.lib window library

dynamic: leda.dll, leda.lib
libgeow.lib

Remarks: GeoWin is currently not available as a DLL.

Now follow the instructions given in the corresponding section for the Windows object code package (Section WinObjectCodePackage ff.).

A.5 Compiling LEDA for Multithreading

There are three flags which can be set at the end of <LEDAROOT>\incl\LEDA\system.h:

LEDA_MEMORY_STD: If this is set, the (standard) LEDA memory manager, which is not thread-safe, is used.

LEDA_MEMORY_SYS: If this is set, the system memory management is used (new and delete) instead of the LEDA manager.

LEDA_MEMORY_MT: If this is set, the multithreading LEDA memory management is used.

One of these flags must be set. If LEDA_MULTI_THREAD is not defined, LEDA_MEMORY_STD is used by default. Otherwise LEDA_MEMORY_SYS is the default and LEDA_MEMORY_MT can be used alternatively.

You have to edit <LEDAROOT>\thread.h in order to include the right thread package (win32.h, posix.h, solaris.h, or cps.h).

A.6 Usage of Header Files

LEDA data types and algorithms can be used in any C++ program as described in this manual (for the general layout of a manual page please see Chapter LEDA Manual Page). The specifications (class declarations) are contained in header files. To use a specific data type its header file has to be included into the program. In general the header file for data type xyz is <LEDA/xyz.h>.

A.7 Object Code on UNIX

Files and Directories

To compile and link your programs with LEDA, the LEDA main directory should contain at least the following files and subdirectories:
Readme.txt Readme File
Install/unix.txt txt–version of this section
incl/ the LEDA include directory
libleda.a (libleda.so) the LEDA library

The static library has the extension .a. If a shared library is provided it has extension .so.

Preparations

Unpacking the LEDA distribution file LEDA-<ver>-<sys>-<cc>.tar.gz will create the LEDA root directory "LEDA-<ver>-<sys>-<cc>". You might want to rename it or move it to some different place. Let <LEDA> denote the final complete path name of the LEDA root directory.

To install and use the Unix object code of LEDA you have to modify your environment as follows:

- Set the environment variable LEDAROOT to the LEDA root directory:
  
  csh/tcsh: setenv LEDAROOT <LEDA>
  
  sh/bash: LEDAROOT=<LEDA> export LEDAROOT

- Include $LEDAROOT/Manual/cmd into your command search path (environment variable path (csh) or PATH (sh)) and call rehash (if required by your system).

- Shared Library: (for solaris, linux, irix, osf1)
  If you planning to use the shared library include $LEDAROOT into the LD_LIBRARY_PATH search path. Then go to $LEDAROOT and type make shared. This will construct the shared library from the static library.

- xlman and demos: Go to $LEDAROOT and type make xlman to compile and link LEDA’s interactive manual reader xlman. Now you can start xlman for reading and printing manual pages, starting demo programs and browsing more release notes.

Compiling and Linking Application Programs

1. Use the -I compiler flag to tell the compiler where to find the LEDA header files.

   CC (g++) -I$LEDAROOT/incl -c file.c

2. Use the -L compiler flag to tell the compiler where to find the libraries (libL.a/so, libG.a/so, ...)

   CC (g++) -L$LEDAROOT file.o -lleda -lX11 -lm
When using graphics on Solaris systems you might have to link with the system socket library and the network services library as well:

```
CC (g++) ... -lleda -lX11 -lsocket -lnsl -lm
```

Remark: The libraries must be given in the above order.

3. Compile and link simultaneously with

```
CC (g++) -I$LEDAROOT/incl -L$LEDAROOT file.c -lleda -lX11 -lm
```

You may want to ask your system administrator to install the header files and libraries in the system’s default directories. Then you no longer have to specify header and library search paths on the compiler command line.

Example programs and demos

The source code of all example and demo programs can be found in $LEDAROOT/test and $LEDAROOT/demo. Goto $LEDAROOT/test or $LEDAROOT/demo and type `make` to compile and link all test or demo programs, respectively.

A.8 Static Libraries for MS Visual C++ .NET

This section describes the installation and usage of static libraries of LEDA with Microsoft Visual C++ .NET.

Remark: This section describes the situation in a GUI package of LEDA. If you have the geometry, graph, or basic package only the corresponding libraries are provided.

Preparations

To install LEDA you only need to execute the LEDA distribution file LEDA-<ver>-<package>-win32-<compiler>.exe. During setup you can choose the name of the LEDA root directory and the parts of LEDA you want to install.

Then you have to set the environment variable LEDAROOT as follows:

1. Windows NT/2000/XP: On these platforms LEDARROOT is set automatically to the directory where you installed LEDA by the setup program.

2. Windows 95/98:

   (a) Add the line `set LEDAROOT=<LEDA>` to the file ”autoexec.bat” where `<LEDA>` is the root directory of your LEDA installation, e.g., `D:\LEDA-4.4`.

   (b) Restart Windows 95/98 for the change to take effect.
Files and Directories

To compile and link your programs with LEDA, the LEDA main directory should contain the following files and subdirectories:

- **Readme.txt**
- **Install\win32\msc70_lib.txt**
- **incl\**
- **man_html\**

and at least one of the following library sets

- `libl_ml.lib`, `libg_ml.lib`, `libp_ml.lib`, `libw_ml.lib`, `libgeow_ml.lib`, `libd3_ml.lib`
- `libl_mld.lib`, `libg_mld.lib`, `libp_mld.lib`, `libw_mld.lib`, `libgeow_mld.lib`, `libd3_mld.lib`
- `libl_md.lib`, `libg_md.lib`, `libp_md.lib`, `libw_md.lib`, `libgeow_md.lib`, `libd3_md.lib`
- `libl_mdd.lib`, `libg_mdd.lib`, `libp_mdd.lib`, `libw_mdd.lib`, `libgeow_mdd.lib`, `libd3_mdd.lib`
- `libl_mt.lib`, `libg_mt.lib`, `libp_mt.lib`, `libw_mt.lib`, `libgeow_mt.lib`, `libd3_mt.lib`
- `libl_mtd.lib`, `libg_mtd.lib`, `libp_mtd.lib`, `libw_mtd.lib`, `libgeow_mtd.lib`, `libd3_mtd.lib`

Compiling and Linking in Microsoft Visual C++ .NET

To compile and link an application program using LEDA in Microsoft Visual C++ .NET proceed as follows:

1. In the ”File” menu of Visual C++ .NET click on ”New–¿Project”.
2. Choose ”Visual C++ Projects” as **project type** and ”Win32 Project” as **template**. Enter a project name, choose a directory for the project, and click ”OK”
3. In the Win32 Application Wizard click on ”Application Settings”, choose ”Console Application” and ”Empty Project”, and click ”Finish”
4. After clicking ”OK” you have an empty project space

If you already have a source file `prog.cpp`:

5. Activate the file browser and add `prog.cpp` to the main folder of your project
(6) In the Solution Explorer of your project click on "Source Files" with the right mouse button, then click on "Add-¿ Add Existing Item" with the left mouse button.

(7) Double click on prog.cpp

If you want to enter a new source file:

(5') In the Solution Explorer of your project click on "Source Files" with the right mouse button, then click on "Add-¿ Add New Item" with the left mouse button.

(6') Choose "C++ File" in Templates, enter a name, and click "Open".

(7') Enter your code.

(8) In the Solution Explorer right click on your project and left click on "Properties".

(9) Click on "C/C++" and "Code Generation" and choose the "Run Time Library" (=compiler flag) you want to use.

The default value is "/MLd", alternatives are "/ML", "/MD", "/MDd", "/MT", and "/MTd". Notice that you have to use the LEDA libraries that correspond to the chosen flag, e.g., with option "/MD" you must use libl_md.lib, libg_md.lib, libp_md.lib, libw_md.lib, libgeow_md.lib, and libd3_md.lib. Using another set of libraries with "/MD" could lead to serious linker errors.

(10) Click on "Linker" and "Command Line" and add the name of the LEDA libraries you want to use in "Additional Options" as follows. We use <opt> to indicate the compiler option chosen in Step (9) (,e.g., <opt> is mld for "/MLd"):

- libl.<opt>.lib
  for programs using only basic data types of LEDA
- libg.<opt>.lib libl.<opt>.lib
  for programs using graph data types
- libp.<opt>.lib libg.<opt>.lib libl.<opt>.lib
  for programs using geometric data types
- libd3.<opt>.lib libp.<opt>.lib libg.<opt>.lib libl.<opt>.lib wsock32.lib
  for programs using three-dimensional data types
- libw.<opt>.lib libp.<opt>.lib libg.<opt>.lib libl.<opt>.lib wsock32.lib
  for programs using graphics data types
- libgeow.<opt>.lib libd3.<opt>.lib libw.<opt>.lib libp.<opt>.lib libg.<opt>.lib libl.<opt>.lib wsock32.lib
  for programs using GeoWin
Alternatively, you can include `<LEDA/msc/autolink_static.h>` in your program and the correct LEDA libraries are linked to your program automatically. If GeoWin is used you need to add "\_\_LINK\_\_GeoW" to the "Preprocessor definitions" in Step (9). Notice that `autolink_static.h` only works correctly for the GUI package of LEDA.

(11) Click "OK" to leave the "Settings"

(12) In the "Tools" menu click on "Options"

(13) Click on "Projects–VC++ Directories"

(14) Choose "Include Files" in "Show Directories for:" and add the directory `<LEDA>\incl` containing the LEDA include files (Double click on the small rectangle in "Directories", enter `<LEDA>\incl`, or click on the small grey rectangle on the right and choose the correct directory.)

(15) Choose "Library Files" in "Show Directories for:" and add the directory `<LEDA>` containing the LEDA libraries.

(16) Click "OK" to leave the "Options"

(17) In the "Build" menu click on "Build Project" or "Rebuild Project" to compile your program.

(18) In the "Build" menu click on "Execute prog.exe" to execute your program.

Remark: If your C++ source code files has extension .c, you need to add the option "/TP" in "Project Options" (similar to Step (9)), otherwise you will get a number of compiler errors. (Click on "C/C++" and "Command Line". Add /TP in "Additional Options" and click "Apply".)

To add LEDA to an existing Project in Microsoft Visual C++ .NET, start the Microsoft Visual Studio with your project and follow Steps (8)–(16) above.

### Compiling and Linking Application Programs in a DOS-Box

(a) Setting the Environment Variables for Visual C++:

The compiler CL.EXE and the linker LINK.EXE require that the environment variables PATH, INCLUDE, and LIB have been set properly.

When Visual C++ was installed, setup created a batch file, VSVERS32.BAT, containing commands to modify PATH, LIB, and INCLUDE. Run VSVERS32.BAT at the command prompt before you compile your application program. VSVERS32.BAT is located in the \bin subdirectory of your compiler installation, e.g., c:\programs\microsoft visual studio .NET\Vc7\bin.

To compile programs together with LEDA, the environment variables PATH, LIB, and INCLUDE must additionally contain the corresponding LEDA directories.

(b) Setting Environment Variables for LEDA:

(i) Windows NT/2000/XP:
1. In the "Start" menu, point to "Settings", then click "Control Panel".
2. In the "Control Panel", double click "System".
3. In the System Properties dialog box, click the Environment tab (for NT), respectively the Extended tab and then on "Environment Variables" (for 2000/XP).
   If a user variable \texttt{PATH}, \texttt{LIB}, or \texttt{INCLUDE} already exists, extend the current value as follows:
   \begin{itemize}
   \item extend \texttt{PATH} by \texttt{<LEDA>}
   \item extend \texttt{INCLUDE} by \texttt{<LEDA>\incl}
   \item extend \texttt{LIB} by \texttt{<LEDA>}
   \end{itemize}
   Otherwise add new a user variable \texttt{PATH}, \texttt{INCLUDE}, or \texttt{LIB} with value \texttt{<LEDA>}, respectively \texttt{<LEDA>\incl}.

(ii) Windows 95/98:

1. Change the file \texttt{autoexec.bat} as follows:
   If a variable \texttt{PATH}, \texttt{INCLUDE}, or \texttt{LIB} is already set, append the appropriate LEDA directory \texttt{<LEDA>}, respectively \texttt{<LEDA>\incl}.
   Otherwise add a corresponding line to \texttt{autoexec.bat}:
   \begin{verbatim}
   set PATH=<LEDA>
   set INCLUDE=<LEDA>\incl
   set LIB=<LEDA>
   \end{verbatim}
2. Restart Windows 95/98 for the change to take effect.

After these procedures, it suffices to call \texttt{VSVARS32.BAT} at the command prompt. In both cases the LEDA paths are automatically appended to the compiler paths.

(c) Compiling and Linking Application Programs:
After setting the environment variables, you can use the LEDA libraries as follows to compile and link programs.

Programs using basic data types:

\begin{verbatim}
cl <option> -TP prog.c <libl.lib>
\end{verbatim}

Programs using graph data types:

\begin{verbatim}
cl <option> -TP prog.c <libg.lib> <libl.lib>
\end{verbatim}

Programs using geometric data types:

\begin{verbatim}
cl <option> -TP prog.c <libp.lib> <libg.lib> <libl.lib>
\end{verbatim}

Programs using three-dimensional data types:

\begin{verbatim}
cl <option> -TP prog.c <libd3.lib> <libp.lib> <libg.lib> <libl.lib>
\end{verbatim}

Programs using graphics data types:
Example programs and demos

The source code of all example and demo programs can be found in the directory `<LEDA>\test` and `<LEDA>\demo`. Goto `<LEDA>` and type `make_test` or `make_demo` to compile and link all test or demo programs, respectively.

A.9  DLL’s for MS Visual C++ .NET

This section describes the installation and usage of LEDA Dynamic Link Libraries (DLL’s) with Microsoft Visual C++ .NET.

Remark: This section describes the situation in a GUI package of LEDA. If you have the geometry, graph, or basic package only the corresponding libraries are provided.

Preparations

To install LEDA you only need to execute the LEDA distribution file `LEDA-<ver>-<package>-win32-<compiler>.exe`. During setup you can choose the name of the LEDA root directory and the parts of LEDA you want to install.

Then you have to set the environment variable LEDAROOT as follows:

1. Windows NT/2000/XP: On these platforms LEDAROOT is set automatically to the directory where you installed LEDA by the setup program.

2. Windows 95/98:
   (a) Add the line `set LEDAROOT=<LEDA>` to the file ”autoexec.bat” where `<LEDA>` is the root directory of your LEDA installation, e.g., D:\LEDA-4.4\ 
   (b) Restart Windows 95/98 for the change to take effect.
Files and Directories

To compile and link your programs with LEDA, the LEDA main directory should contain the following files and subdirectories:

- Readme.txt: Readme File
- Install\win32\msc70_dll.txt: txt–version of this section
- incl\: the LEDA include directory
- man_html\: HTML version of the LEDA user manual

and at least one of the following dll/library sets

- leda_ml.dll, leda_ml.lib, libGeoW_ml.lib
- leda_mld.dll, leda_mld.lib, libGeoW_mld.lib
- leda_md.dll, leda_md.lib, libGeoW_md.lib
- leda_mdd.dll, leda_mdd.lib, libGeoW_mdd.lib
- leda_mt.dll, leda_mt.lib, libGeoW_mt.lib
- leda_mtd.dll, leda_mtd.lib, libGeoW_mtd.lib

Note: A DLL of GeoWin is currently not available.

Compiling and Linking in Microsoft Visual C++ .NET

You can choose between a manual and an automatic way to compile and link an application program using LEDA in Microsoft Visual C++ .NET.

Automatic Setting

Include header file LEDA/msc/autolink_dll.h in one of your project’s cpp-files. Depending on your build environment (Debug or Release build) the correct LEDA dll will be chosen automatically when you build your project. Proceed with step 18 of the manual setting.

Manual Setting

1. In the "File" menu of Visual C++ .NET click on "New-Project".

2. Choose "Visual C++ Projects" as project type and "Win32 Project" as template. Enter a project name, choose a directory for the project, and click "OK"

3. In the Win32 Application Wizard click on "Application Settings", choose "Console Application" and "Empty Project", and click "Finish"
(4) After clicking "OK" you have an empty project space

If you already have a source file prog.cpp:

(5) Activate the file browser and add prog.cpp to the main folder of your project

(6) In the Solution Explorer of your project click on "Source Files" with the right mouse button, then click on "Add Existing Item" with the left mouse button

(7) Double click on prog.cpp

If you want to enter a new source file:

(5') In the Solution Explorer of your project click on "Source Files" with the right mouse button, then click on "Add New Item" with the left mouse button.

(6') Choose "C++ File" in Templates, enter a name, and click "Open".

(7') Enter your code.

(8) In the Solution Explorer right click on your project and left click on "Properties"

(9a) Click on "C/C++" and "Code Generation" and choose the "Run Time Library" (=compiler flag) you want to use.

    The default value is "/MLd", alternatives are "/ML", "/MD", "/MDd", "/MT", and "/MTd". Notice that you have to use the LEDA libraries that correspond to the chosen flag, e.g., with option "/MD" you must use leda_md.lib and libGeoW_md.lib. Using another set of libraries with "/MD" could lead to serious linker errors.

(9b) Click on "C/C++" and "Preprocessor" and add ";LEDA_DLL" in "Preprocessor Definitions".

(10) Click on "Linker" and "Command Line" and add the name of the LEDA libraries you want to use in "Additional Options" as follows. We use <opt> to indicate the compiler option chosen in Step (9) (e.g., <opt> is mld for "/MLd”).

    • leda_<opt>.lib for programs that do not use GeoWin
    • libGeoW_<opt>.lib leda_<opt>.lib for programs using GeoWin

    Alternatively, you can include <LEDA/msc/autolink_dll.h> in your program and the correct LEDA libraries are linked to your program automatically. If GeoWin is used you need to add ";LINK_GeoW" to the "Preprocessor definitions" in Step (9).

(11) Click "OK" to leave the "Settings"

(12) In the "Tools" menu click on "Options"
(13) Click on "Projects-VC++ Directories"

(14) Choose "Include Files" in "Show Directories for:" and add the directory 
\texttt{<LEDA>\incl} containing the LEDA include files (Double click on the small rectangle in "Directories", enter \texttt{<LEDA>\incl}, or click on the small grey rectangle on the right and choose the correct directory.)

(15) Choose "Library Files" in "Show Directories for:" and add the directory \texttt{<LEDA>} containing the LEDA libraries.

(16) Click "OK" to leave the "Options"

(17) In the "Build" menu click on "Build Project" or "Rebuild Project" to compile your program.

(18) To execute the program "prog.exe" Windows needs to have \texttt{leda<opt>.dll} in its search path for DLL's. Therefore, you need to do one of the following.

- Copy \texttt{leda<opt>.dll} to the \texttt{bin} subdirectory of your compiler or the directory containing "prog.exe".
- Alternatively, you can set the environment variable \texttt{PATH} to the directory containing \texttt{leda<opt>.dll} as described below.

(19) In the "Build" menu click on "Execute prog.exe" to execute your program.

Remark: If your C++ source code files has extension .c, you need to add the option "/TP" in "Project Options" (similar to Step (9)), otherwise you will get a number of compiler errors. (Click on "C/C++" and "Command Line". Add /TP in "Additional Options" and click "Apply").

The default value is "/MLd", alternatives are "/ML", "/MD", "/MDd", "/MT", and "/MTd". Notice that you have to use the LEDA libraries that correspond to the chosen flag, e.g., with option "/MD" you must use \texttt{leda_md.lib} and \texttt{libGeoW_md.lib}. Using another set of libraries with "/MD" could lead to serious linker errors.

Examples of workspaces can be found in the directory \texttt{<LEDA>\demo\msc_70_workspaces}. To add LEDA to an existing Project in Microsoft Visual C++ .NET, start the Microsoft Visual Studio with your project and follow Steps (8)–(16) above.

\section*{Compiling and Linking Application Programs in a DOS-Box}

\textbf{a) Setting the Environment Variables for Visual C++ .NET:}
The compiler CL.EXE and the linker LINK.EXE require that the environment variables PATH, INCLUDE, and LIB have been set properly.

When Visual C++ .NET was installed, setup created a batch file, VSVARS32.BAT, containing commands to modify PATH, LIB, and INCLUDE. Run VSVARS32.BAT at the command prompt before you compile your application program. VSVARS32.BAT is located in the \texttt{bin} subdirectory of your compiler installation, e.g., \texttt{c:\programs\microsoft visual studio .NET\Vc7\bin}.

To compile programs together with LEDA, the environment variables PATH, LIB, and INCLUDE must additionally contain the corresponding LEDA directories.
(b) Setting Environment Variables for LEDA:

(i) Windows NT/2000/XP:

1. In the "Start" menu, point to "Settings", then click "Control Panel".
2. In the "Control Panel", double click "System".
3. In the System Properties dialog box, click the Environment tab (for NT), respectively the Extended tab and then on "Environment Variables" (for 2000/XP).

If a user variable PATH, LIB, or INCLUDE already exists, extend the current value as follows:
- extend PATH by <LEDA>
- extend INCLUDE by <LEDA>\incl
- extend LIB by <LEDA>

Otherwise add new a user variable PATH, INCLUDE, or LIB with value <LEDA>, respectively <LEDA>\incl.

(ii) Windows 95/98:

1. Change the file autoexec.bat as follows:

   If a variable PATH, INCLUDE, or LIB is already set, append the appropriate LEDA directory <LEDA>, respectively <LEDA>\incl.

   Otherwise add a corresponding line to autoexec.bat:

   ```
   set PATH=<LEDA>
   set INCLUDE=<LEDA>\incl
   set LIB=<LEDA>
   ```

2. Restart Windows 95/98 for the change to take effect.

After these procedures, it suffices to call VSVARS32.BAT at the command prompt. In both cases the LEDA paths are automatically appended to the compiler paths.

(c) Compiling and Linking Application Programs:

After setting the environment variables, you can use the LEDA libraries as follows to compile and link programs.

Programs that do not use GeoWin:

```c
cl <option> -DLEDA_DLL -TP prog.c <libl.lib>
```

Programs using GeoWin:

```c
cl <option> -DLEDA_DLL -TP prog.c <libGeoW.lib> <libl.lib>
```

Possible values for <option> are ",-ML", ",-MLd", ",-MD", ",-MDd", ",-MT", and ",-MTd". You have to use the LEDA libraries that correspond to the chosen <option>, e.g., with option ",-MD" you must use leda_md.lib and libGeoW_md.lib. Using another set of libraries with ",-MD" could lead to serious linker errors.

**Remark:** The option -TP is necessary if the C++ source code files have extension .c. If your files have extension .cpp this option is not necessary.
Example programs and demos

The source code of all example and demo programs can be found in the directory `<LEDA>/test` and `<LEDA>/demo`. Goto `<LEDA>` and type `make_test` or `make_demo` to compile and link all test or demo programs, respectively.

A.10 Namespaces and Interaction with other Libraries

If users want to use other software packages like STL together with LEDA in one project avoiding naming conflicts is an issue.

LEDA defines all names (types, functions, constants, ...) in the namespace `leda`. This makes the former macro–based prefixing scheme obsolete. Note, however, that the prefixed names “leda...” still can be used for backward compatibility. Application programs have to use namespace “leda” globally (by saying "using namespace leda;") or must prefix every LEDA symbol with "leda::".

The second issue of interaction concerns the data type `bool` which is part of the new C++ standard. However not all compilers currently support a bool type. LEDA offers `bool` either compiler provided or defined within LEDA if the compiler lacks the support. Some STL packages follow a similar scheme. To solve the existence conflict of two different bool type definitions we suggest to use LEDA’s bool as STL is a pure template library only provided by header files and its defined bool type can be easily replaced.

A.11 Platforms

Please visit our web pages for information about the supported platforms.
Appendix B

The golden LEDA rules

The following rules must be adhered to when programming with LEDA in order to write syntactically and semantically correct and efficient LEDA programs. The comprehension of most of the rules is eased by the categorization of the LEDA types given in section B.2. Every rule is illustrated in section B.2 by one or more code examples.

B.1 The LEDA rules in detail

1. (Definition with initialization by copying) Definition with initialization by copying is possible for every LEDA type. It initializes the defined variable with a copy of the argument of the definition. The next rule states precisely what a copy of a value is.

2. (Copy of a value) Assignment operator and copy constructor of LEDA types create copies of values. This rule defines recursively what is meant by the notion “copy of a value”.

   (a) A copy of a value of a primitive type (built-in type, pointer type, item type) is a bitwise copy of this value.

   (b) A value \( x \) of a simple-structured type is a set or a sequence of values, respectively.

      A copy of \( x \) is a componentwise copy of all constituent values of this set or this sequence, respectively.

   (c) A value \( x \) of an item-based, structured type is a structured collection of values.

      A copy of \( x \) is a collection of new values, each one of which is the copy of a value of \( x \), the original. The combinatorical structure imposed to the new values is isomorphic to the structure of \( x \), the original.

3. (Equality and identity) This rule defines when two objects \( x \) and \( y \) are considered as equal and identical, respectively.

   (a) For objects \( x \) and \( y \) of a dependent item type, the equality predicate \( x==y \) means equality between the values of these objects.
APPENDIX B. THE GOLDEN LEDA RULES

(b) For objects $x$ and $y$ of an independent item type $T$, the equality predicate $x==y$ is defined individually for each such item type. In the majority of cases it means equality between the values of $x$ and $y$, but this is not guaranteed for every type.

Provided that the identity predicate

```cpp
bool identical(const T&, const T&);
```

is defined on type $T$, it means equality between the values of these objects.

(c) For objects $x$ and $y$ of a structured type the equality predicate $x==y$ means equality between the values of these objects.

4. (Illegal access via an item) It is illegal to access a container which has been destroyed via an item, or to access a container via the item nil.

5. (Initialization of attributes of an independent item type) The attributes of an independent item type are always defined. In particular, a definition with default initialization initializes all attributes. Such a type may specify the initial values, but it need not.

6. (Specification of the structure to be traversed in forall-macros)

   The argument in a `forall`-macro which specifies the structure to be traversed should not be a function call which returns this structure, but rather an object by itself which represents this structure.

7. (Modification of objects of an item-based container type while iterating over them)

   An iteration over an object $x$ of an item-based container type must not add new elements to $x$. It may delete the element which the iterator item points to, but no other element. The values of the elements may be modified without any restrictions.

8. (Requirements for type parameters)

   Every type parameter $T$ must implement the following functions:

   | Requirement | Function
   |-------------|---------|
   | a default constructor | $T::T()$
   | a copy constructor | $T::T(const \ T&)$
   | an assignment operator | $T& T::operator = (const \ T&)$
   | an input operator | $istream& \ operator >> (istream&, \ T&)$
   | an output operator | $ostream& \ operator << (ostream&, const \ T&)$

9. (Requirements for linearly ordered types)

   In addition to the Requirements for type parameters a linearly ordered type must implement

   ```cpp
   int compare(const T&, const T&)
   ```

   Here, for the function `compare()` the following must hold:

   (a) It must be put in the namespace `leda`.

   (b) It must realize a linear order on $T$. 


B.2. CODE EXAMPLES FOR THE LEDA RULES

(c) If \( y \) is the copy of a value \( x \) of type \( T \), then \( \text{compare}(x,y) == 0 \) must hold.

10. (Requirements for hashed types) In addition to the Requirements for type parameters a hashed type must implement

<table>
<thead>
<tr>
<th>a hash function</th>
<th>( \text{int} ) ( \text{Hash}(\text{const} \ T &amp;) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>an equality operator</td>
<td>( \text{bool} ) ( \text{operator}== \text{(const} \ T &amp;, \text{const} \ T &amp;) )</td>
</tr>
</tbody>
</table>

Here, for the function \( \text{Hash()} \) the following must hold:

(a) It must be put in the namespace \text{leda}.

(b) For all objects \( x \) and \( y \) of type \( T \): If \( x == y \) holds, then so does \( \text{Hash}(x) == \text{Hash}(y) \).

For the equality operator \( \text{operator}==() \) the following must hold:

(a) It defines an equivalence relation on \( T \).

(b) If \( y \) is a copy of a value \( x \) of type \( T \), then \( x == y \) must hold.

11. (Requests for numerical types) In addition to the Requirements for type parameters a numerical type must offer the arithmetical operators \( \text{operator}+() \), \( \text{operator}-() \), and \( \text{operator}*() \), as well as the comparison operators \( \text{operator}<() \), \( \text{operator}<=() \), \( \text{operator}>() \), \( \text{operator}>=() \), \( \text{operator}==() \), and \( \text{operator}!=() \).

B.2 Code examples for the LEDA rules

1. \text{string} \ s("Jumping Jack Flash");
   \text{string} \ t(s); \quad //\text{definition with initialization by copying}
   \text{string} \ u = s; \quad //\text{definition with initialization by copying}

   \text{stack<}\text{int}>\ S;
   \quad //\ldots\text{fill} \ S \text{with some elements}
   \text{stack<}\text{int}>\ T(S); \quad //\text{definition with initialization by copying}

2. (a) \text{list\_item} \ it1, \it2;
   \quad //\ldots
   \it2 = \it1; \quad //\it2 \text{now references the same container as} \it1

(b) \text{array<}\text{int}>\ A, \ B;
   \quad //\ldots\text{fill} \ A \text{with some elements...}
   \quad B = A;

   \text{Now} \ B \text{contains the same number of integers as} \ A, \text{in the same order, with the}
   \text{same values.}

   \text{However}, \ A \text{and} \ B \text{do not contain the same objects:}

   \text{int*} \ p = A[0];
   \text{int*} \ q = B[0];
   \quad p == q; \quad //\text{false}
A and B are different objects:

\[ A == B; // \text{false} \]

(c) \text{list}\langle\text{int}\rangle \text{ L, M};
\text{list\_item it1, it2;}
L.push(42);
L.push(666);
M = L;
L and M now both contain the numbers 666 and 42. These numbers are not the same objects:

\[ it1 = \text{L.first();} \]
\[ it2 = \text{M.first();} \]
\[ it1 == it2; // \text{false} \]

L and M are different objects as well:

\[ L == M; // \text{false} \]

In the following assignment the rules c, b, and a are applied recursively (in this order):

\text{list\langle array\langle int\rangle \rangle L, M;}
// ...fill L with some array\langle int\\rangle\text{s}
// each of them filled with some elements...
M = L;

3. (a) \text{list\_item it1, it2;}
// ...
\[ \text{it2 = it1; // it2 now references the same container as it1} \]
\[ \text{it1 == it2; // true} \]

(b) \text{point p(2.0, 3.0);} 
\text{point q(2.0, 3.0);} 
\[ \text{p == q; // true (as defined for class point)} \]
\text{identical(p, q); // false} 
\text{point r;}
\[ \text{r = p;} \]
\text{identical(p, r); // true} 

(c) \text{list\langle int\rangle L, M;}
// ...fill L with some elements...
M = L;
\[ \text{L == M; // false} \]
4. `list_item it = L.first();
   L.del_item(it);
   L.contents(it); // illegal access
   it = nil;
   L.contents(it); // illegal access`

5. `point p(2.0, 3.0); // p has coordinates (2.0, 3.0)
   point q; // q has coordinates but it is not known which`

6. `edge e;
   forall(e, G.all_edges()) // dangerous!
   { ... }

   // do it like this
   list<edge> E = G.all_edges();
   forall(e, E)
   { ... }`}

7. `list_item it;
   forall(it, L) {
      L.append(1); // illegal; results in infinite loop
      if(L[it] == 5) L.del(it); // legal
      if(L[it] == 6) L.del(L.succ(it)); // illegal
      L[it]++; // legal
   }

8. `class pair {
   public:
      int x, y;

      pair() { x = y = 0; }
      pair(const pair& p) { x = p.x; y = p.y; }
      pair& operator=(const pair& p) {
         if(this != &p) { x = p.x; y = p.y; }
         return *this;
      }
   };

   std::istream& operator>>(std::istream& is, pair& p)
   { is >> p.x >> p.y; return is; }
   std::ostream& operator<<(std::ostream& os, const pair& p)
   { os << p.x << " " << p.y; return os; }

9. namespace leda {
   int compare(const pair& p, const pair& q)
   {
      if (p.x < q.x) return -1;
      if (p.x > q.x) return 1;
      if (p.y < q.y) return -1;`
if (p.y > q.y) return 1;
return 0;
};

10. namespace leda {
    int Hash(const pair& p)
    {
        return p.x ^ p.y;
    }
};

bool operator == (const pair& p, const pair& q)
{
    return (p.x == q.x && p.y == q.y) ? true : false;
}
Appendix C

The LEDA Tools for Manual Production and Documentation

The Lman and Fman utilities give online access to the manual pages of the LEDA system. Lman produces \TeX-output and Fman produces low quality ASCII-output for quick reference. The usage is

\texttt{Lman T[.h] options}  \\
\texttt{Fman T[.h] filter}

where \texttt{T} is the name of a LEDA type, e.g., list, sortseq, or point. Without options Lman produces manual pages as contained in the LEDA manual. Thus, \texttt{Lman list} produces the manual page for the LEDA type list and \texttt{Fman list} produces a low quality version of it. Please try out Lman and Fman before proceeding. If they do not work the error is very likely to be one of the following.

- You don’t have a current version of \texttt{perl} installed and accessible.
- You don’t have a current version of \texttt{latex} installed and accessible.
- The environment variable \texttt{LEDAROOT} is not set to the root directory of the LEDA system.
- \texttt{$\$LEDAROOT/Manual/cmd} is not part of your \texttt{PATH}.
- \texttt{$\$LEDAROOT/Manual/tex} is not part of your \texttt{TEXINPUTS}.

The behavior of Lman and Fman can be fine-tuned by options. A call \texttt{Lman} without arguments gives a short survey of all available options. Options are specified in assignment syntax \texttt{variable=value}. There must be no blank on either side of the equality sign. In the list of options to follow we list the default value of each option first.

\texttt{size=\{12, 11, 10\}} Determines the font size.

\texttt{constref=\{no, yes\}} Determines how const-ref parameters are displayed. With the no-option a const-ref parameter \texttt{const T& x} is displayed as a value parameter \texttt{T x} and with the yes-option it is displayed in full.
partypes={no, yes} Determines how parameters of unary and binary operators are displayed. Consider for example an operator \( + \) of a class number. With the no-option the operator \( \text{operator+}(\text{number } x, \text{number } y) \) is displayed as \( x + y \) and with the yes-option it is displayed as \( \text{number } x + \text{number } y \).

numbered={no, yes} Determines whether the headerline of the manual page is numbered. You probably want it numbered when the manual page becomes part of a larger document.

warnings={yes, no} Determines whether Lman gives warnings. We recommend to use the option warnings=yes and to deal with individual warnings as later in the chapter.

verbosity={yes, no} Determines whether Lman gives status information.

ack={no, yes} Determines whether Lman asks for acknowledgments of warnings.

usesubscripts={yes, no} Determines whether variables consisting of a single character followed by a number are displayed as subscripted variables.

filter={all, signatures, definition, creation, operations, implementation, example, opname} Determines which part of the manual page is shown. The all-option shows the complete manual page, the signature-options shows the signatures of all operations of the data type, the next five options show only the corresponding section of the manual page, and the opname-option shows only the operation with the same name. Thus \text{Fman list insert} shows the insert operation for lists and \text{Lman point filter=operator+} extracts the + operator of the point type (write \text{arrop} to inquire about the array operator and \text{funop} to inquire about the function operator).

outfile={"",string} Determines whether the \TeX-file generated is written on a temporary file (the default option) or on the file with name string. Only available for Lman.

latexruns={1, 0, 2} Determines the number of \LaTeX runs used to produce the manual page. You need to run \LaTeX twice if the manual page contains cross references.

xdvi={yes, no} Determines whether the manual page is displayed by xdvi.

Lman can be customized by putting options in a file Lman.cfg in either the home directory or the working directory. Command line options take precedence over options in the working directory which in turn take precedence over options in the home directory.

Fman is the utility for quick reference. It takes two arguments, a typename and one of the filters, and directly displays its output. For example, the call \text{Fman list signatures} will display the signatures of all operations of the list data type.

Lman and Fman are not only applicable to the header files of the LEDA system, they can be applied to any file containing (possibly among other things) C++-code suitably augmented by so-called manual comments. A manual comment is any comment of the form /*{\Mcommand \ldots}*/ where \text{Mcommand} is one of so-called manual commands. We
#ifndef LEDA_STACK_H
#define LEDA_STACK_H

#include <LEDA/basic.h>
#include <LEDA/impl/slist.h>

/*\Manpage{stack}{E}{Stacks}{S}*/
template<class E> class _CLASSTYPE stack : private SLIST {
  /*\Mdefinition
  An instance \S of the parameterized data type \E is a sequence of
  elements of data type \E, called the element type of \S. Insertions or
  deletions of elements take place only at one end of the sequence, called
  the top of \S. The size of \S is the length of the sequence, a stack
  of size zero is called the empty stack.\*/

public:
  /*\Mcreation\*/
  stack() {}
  /*\Mcreate creates an instance \Mvar of type \Mname and initializes
  it to the empty stack.\*/
  stack(const stack<E>& S) : SLIST(S) {}
  ~stack() { clear(); }
  stack<E>& operator=(const stack<E>& S)
  { return (stack<E>&)SLIST::operator=(S); }

  /*\Moperations 2.5 4\*/
  E top() const { return ACCESS(E,SLIST::head());}
  /*\Mop returns the top element of \Mvar.\precond \S is not empty.\*/
  void push(E x) { SLIST::push(Copy(x));}
  /*\Mop adds \$x\$ as new top element to \Mvar.\*/
  E pop() { E x=top(); SLIST::pop(); return x; }
  /*\Mop deletes and returns the top element of \Mvar.\precond \S is not empty.\*/

  /*\Mimplementation
  Stacks are implemented by singly linked linear lists. All operations take
time \O(1).\*/
};
#endif

Figure C.1: The header file for the stack data type.

discuss manual commands in detail below. Every manual comment produces part of the
manual page.
Stacks (stack)

1. Definition

An instance $S$ of the parameterized data type $\text{stack}<E>$ is a sequence of elements of data type $E$, called the element type of $S$. Insertions or deletions of elements take place only at one end of the sequence, called the top of $S$. The size of $S$ is the length of the sequence, a stack of size zero is called the empty stack.

2. Creation

$\text{stack}<E> \ S$; creates an instance $S$ of type $\text{stack}<E>$ and initializes it to the empty stack.

3. Operations

- $E \ S$.
  - $\text{top}(\ )$ returns the top element of $S$.
    - **Precondition:** $S$ is not empty.
- $\text{void } S$.\ $\text{push}(E \ x)$ adds $x$ as new top element to $S$.
- $E \ S$.\ $\text{pop}(\ )$ deletes and returns the top element of $S$.
  - **Precondition:** $S$ is not empty.

4. Implementation

Stacks are implemented by singly linked linear lists. All operations take time $O(1)$, except clear which takes time $O(n)$, where $n$ is the size of the stack.

Figure C.2: The manual page for the stack data type

Figure Header file of Stack shows a header file augmented by manual comments and figure A Man-Page shows the manual page produced from it. The body of a manual comment is an ordinary \LaTeX-text augmented by the possibility to *quote code*. Code is quoted by enclosing it either between vertical bars (|...|) or between double square brackets ([[[...]]]). Code quoted by double square brackets is typeset using typewriter font and code quoted by vertical bars is given a math-like appearance. So [[S.insert(E x)]] produces $\text{S.insert(E x)}$ and |S.insert(E x)| produces $\text{S.insert(E x)}$.

Lman is based on the command lextract. See figure ldel and lextract.

```
lextract infile outfile [options]
```

reads infile, processes the manual commands in it, and writes the core of a \TeX-file on outfile. All Lman options apply. Lman itself works in three phases. It first produces a temporary \TeX-file

```
\documentclass[a4paper,size pt]{article}
```
Figure C.3: The Manual Production Process

\usepackage{Lweb}
\begin{document}
output of \texttt{lextract}
\end{document}

and then applies \texttt{latex} and \texttt{xdvi} to it.

\begin{figure}[ht]
\centering
\begin{tikzpicture}
\node (LEDAROOT) at (0,0) {LEDAROOT};
\node (incl) at (-2,0) {incl};
\node (LED) at (0,0) {LED};
\node (Manual) at (2,0) {Manual};
\node (cmd) at (0,-1) {cmd};
\node (noweb) at (0,-1.5) {noweb};
\node (tex) at (0,-2) {tex};
\node (MANUAL) at (2,-2) {MANUAL};
\draw[->] (LEDAROOT) -- (incl);
\draw[->] (incl) -- (LED);
\draw[->] (LED) -- (Manual);
\draw[->] (Manual) -- (cmd);
\draw[->] (Manual) -- (noweb);
\draw[->] (Manual) -- (tex);
\draw[->] (cmd) -- (MANUAL);
\draw[->] (noweb) -- (MANUAL);
\draw[->] (tex) -- (MANUAL);
\end{tikzpicture}
\caption{Manual relevant directories}
\end{figure}

\texttt{ldel} \texttt{infile} \texttt{outfile}

reads \texttt{infile}, deletes all manual comments from it, and writes the result on \texttt{outfile}. Weave is the weave-command of the literate programming tool in use, i.e., noweave for noweb, and lweave for Lweb. Ldoc works on noweb- and Lweb-files. It first uses \texttt{lextract} to extract the manual and \texttt{ldel} to extract a file pure of manual comments, it then applies the appropriate weave command to the output of \texttt{ldel}, and it finally applies \texttt{latex} and \texttt{xdvi} to the resulting file. All Lman options apply.

For more information about the LEDA tools for manual production and documentation we refer the reader to the corresponding chapter of [64].
Bibliography


Index

Symbols

() .............................. 142
list<E> ............................ 142
\LaTeX ............................ 761

Apr  
date .............................. 56

Aug  
date .............................. 56
colons  
date .............................. 57

Dec  
date .............................. 56

english  
date .............................. 56

Feb  
date .............................. 56

french  
date .............................. 56

FULL  
\texttt{r\_circle\_gen\_polygon} .......................... 556
\texttt{r\_circle\_polygon} .......................... 550
german  
date .............................. 56
german\_standard  
date .............................. 57

hyphens  
date .............................. 57

Jul  
date .............................. 56

Jun  
date .............................. 56

local  
date .............................. 56

Mar  
date .............................. 56

May  
date .............................. 56

mr\_cut\_off  
date .............................. 56


PPMII\_Coder .......................... 201

mr\_freeze  
PPMII\_Coder .......................... 201

\texttt{NON\_TRIVIAL}  
\texttt{r\_circle\_gen\_polygon} .......................... 556
\texttt{r\_circle\_polygon} .......................... 550

\texttt{NOT\_WEAKLY\_SIMPLE}  
\texttt{r\_circle\_gen\_polygon} .......................... 556
\texttt{r\_circle\_polygon} .......................... 550

Nov  
date .............................. 56

Oct  
date .............................. 56

\texttt{RESPECT\_ORIENTATION}  
\texttt{r\_circle\_gen\_polygon} .......................... 557
\texttt{r\_circle\_polygon} .......................... 551

second  
\texttt{r\_circle\_point} .......................... 543

Sep  
date .............................. 56

\texttt{SIMPLE}  
\texttt{r\_circle\_gen\_polygon} .......................... 556
\texttt{r\_circle\_polygon} .......................... 550

US\_standard  
date .............................. 57

\texttt{WEAKLY\_SIMPLE}  
\texttt{r\_circle\_gen\_polygon} .......................... 556
\texttt{r\_circle\_polygon} .......................... 550

\texttt{\texttt{A()}}}  
\texttt{d3\_plane} .......................... 593
\texttt{d3\_rat\_plane} .......................... 617

\texttt{A0\_Coder} .......................... 196

\texttt{A0s\_Coder} .......................... 199

\texttt{abs(\ldots)} .......................... 77, 79, 83, 89, 98

\texttt{absolute(\ldots)}  
\texttt{residual} .......................... 98

\texttt{accept()}  
\texttt{leda\_socket} .......................... 39

773
INDEX

add(...)                        gml_graph .................................. 337
    dictionary<K, I> ................ 164
acknowledge(...)                add_new_graph_rule(...)  
    GraphWin ........................ 694
window ................................ 651
activate(...)                   add_new_node_rule(...)  
    GeoWin .......................... 707
ACYCLIC_SHORTEST(...(...)) ..... 346
AdaptiveHuffmanCoder ...... 211
add(...)                       gml_graph .................................. 337
    residual .......................... 96, 99
add_dependence(...)            add_node_rule(...)  
    GeoWin .......................... 726
add_edge_done_rule(...)       add_node_rule_for(...(...)
    gml_graph                      gml_graph .................................. 337
add_edge_menu(...)             add_scene_buttons(...)  
    GraphWin ........................ 680
add_edge_rule(...)            add_simple_call(...)  
    gml_graph                      GraphWin 680, 682
add_edge_rule_for(...(...)
    gml_graph                      GraphWin 337
add_export_object(...)        add_special_help...( ...)
    GeoWin .......................... 728
add_graph_done_rule(...)      add_text(...)  
    gml_graph                      GeoWin 726
add_graph_rule(...)           add_to_day(...)  
    gml_graph                      date 60
add_graph_rule_for(...(...)
    gml_graph                      date 60
add_help_text(...)            add_to_year(...)  
    GeoWin .......................... 727
add_input_object(...)         add_to_year(...)  
    GeoWin .......................... 728
add_key(...)                  add_user_call(...)  
    GeoWin .......................... 723
    CryptAutoDecoder 261          GeoWin 728
add_keys_in_file(...)         add_user_layer_ci(...)
    CryptAutoDecoder 262          GeoWin 707
add_keys_in_stream(...)       add_user_layer_point(...)  
    CryptAutoDecoder 261          GeoWin 707
add_member_call(...)          add_user_layer_re(...)
    GraphWin ........................ 682
add_menu(...)                 add_user_layer_se(...)
    GraphWin ........................ 681
add_new_edge_rule(...)        add_user_se(...)
    GeoWin .......................... 707
real_ray........................................507
dynamic_markov_chain..................333
real_segment...............................503
markov_chain..............................332
real_triangle..............................518
cut(...)..........................160
coplanar(...)..............................584, 604
dynamic_trees............................160
copy(...)..................................128
tree_collection<1>......................161
darray<E>...................................128
cyclic_adj pred(...)......................270, 287
copy_rect(...).............................646
cyclic_adj succ(...)......................270, 279
copy(...)..................................128
cyclic pred(...)............................271
create_bitmap(...).......................645
cyclic in pred(...).........................271
create_directory(...)....................36
cyclic in succ(...)........................271
create_pixrect(...).......................645
cost(...)...................................328
crypt_key...................................249
counter.....................................49
crypt_byte_string........................248
crc32_coder................................226
create_directory(...)....................36
crypt_coder................................226
create_bitmap(...).......................645
cyclic_pred(...)............................271
create_input_graph(...)................369
cyclic succ(...)............................271
cross_product(...).........................104, 118, 122
d3..............................................593
crystal .........................................531
d3_rat Plane.................................617
crypt_ascii................................248
d3_rat_rectangle..........................617
d3_rectangle................................522
d2(...).....................................116
d3_rectangle................................495
d2(...)
rectangle.....................................469
d3_rat_point................................582
curr_adj()...................................406
d3_rat_segment.............................617
d3_rat_triangle............................617
d3_grid_graph(...).......................324
d3_rat_segment.............................617
d3_rat_triangle............................617
d3_rat_triangle............................617
d3_rat_triangle............................617
d3_rat_segment.............................617
d3_rat_triangle............................617

current()...................................406
D3 delaunay..............................625
git_bfs<out_adjI>.........................421
d3 delaunay..............................625
git_dfs<out_adjI>.........................423
d3 hull.......................................624
git_dijkstra<out_adjI>..................428
d3 line.......................................591
git_toposort<out_adjI>..................425
d3 plane.....................................593
current_node()................................
d3 delaunay..............................625
dynamic_markov_chain.............333
d3 delaunay..............................625
GIT_SCC<out, in>.........................427
d3 delaunay..............................625
markov_chain..............................332
d3 delaunay..............................625
current_outdeg()...........................
d3 delaunay..............................625
d3 delaunay..............................625
INDEX

BlockCoder<Coder> ........................................ 238
BTCoder ..................................................... 216
BCoder<BlkCipher> ........................................ 255
checksumer_base ........................................... 225
CoderPipe2<Coder1,...> .................................. 233
CryptAutoDecoder ......................................... 260
DeflateCoder ............................................... 205
DictCoder ................................................... 207
HuffmanCoder ............................................ 210
MTF2Coder ................................................ 221
MTFCoder .................................................. 219
OMACCoder<BlkCipher> ................................... 258
PPMIIICoder ............................................... 202
RLE0Coder ................................................ 223
RLECoder .................................................. 214
deallocating::istream<Coder> .............................. 230
deallocating::istream<Coder> .............................. 228
decoder(...) ................................................ 475
GraphWin ................................................... 684
defined(...) ................................................. 29
degree(...) .................................................. 269
del() .......................................................... 315
del(...) ........................................................ 315
b_node_pq<N> .............................................. 321
d2_dictionary<K1, K2, I> .................................. 566
d_int_set ..................................................... 152
decoder_memory_chunk(...) ................................. 213
dictionary<K, I> ............................................ 164
dict<real> .................................................... 222
dict<real> .................................................... 222
edge_set ..................................................... 315
dict<real> .................................................... 222
GeoWin ....................................................... 725
dict<real> .................................................... 222
GeoWin ....................................................... 725
dict<real> .................................................... 222
GeoWin ....................................................... 725
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>int_set</td>
<td>150</td>
<td>GraphWin</td>
<td>671</td>
</tr>
<tr>
<td>interval_set&lt;K&gt;</td>
<td>574</td>
<td>window</td>
<td>644</td>
</tr>
<tr>
<td>list&lt;K,E&gt;</td>
<td>139</td>
<td>del_min()</td>
<td></td>
</tr>
<tr>
<td>node_list</td>
<td>316</td>
<td>b_node_pq&lt;N&gt;</td>
<td>321</td>
</tr>
<tr>
<td>node_pq&lt;K&gt;</td>
<td>319</td>
<td>b_priority_queue&lt;I&gt;</td>
<td>190</td>
</tr>
<tr>
<td>node_set</td>
<td>314</td>
<td>node_pq&lt;P&gt;</td>
<td>319</td>
</tr>
<tr>
<td>p_dictionary&lt;K,I&gt;</td>
<td>175</td>
<td>p_queue&lt;P,I&gt;</td>
<td>188</td>
</tr>
<tr>
<td>POINT_SET</td>
<td>571</td>
<td>del_min(...)</td>
<td></td>
</tr>
<tr>
<td>pp_dictionary&lt;K,I,CMP&gt;</td>
<td>178</td>
<td>node_pq&lt;P&gt;</td>
<td>319</td>
</tr>
<tr>
<td>rat_segment_set&lt;I&gt;</td>
<td>579</td>
<td>del_node(...)</td>
<td></td>
</tr>
<tr>
<td>segment_set&lt;I&gt;</td>
<td>577</td>
<td>del_node(...)</td>
<td></td>
</tr>
<tr>
<td>set&lt;E&gt;</td>
<td>147</td>
<td>GraphWin</td>
<td>672</td>
</tr>
<tr>
<td>sortseq&lt;K,I&gt;</td>
<td>183</td>
<td>del_node(...)</td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>23</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>del_all(...</td>
<td></td>
<td>del_pin_point()</td>
<td>727</td>
</tr>
<tr>
<td>del_allEdges()</td>
<td></td>
<td>del_pixrect()</td>
<td>646</td>
</tr>
<tr>
<td>graph</td>
<td>273</td>
<td>del_suc_item(...)</td>
<td>146</td>
</tr>
<tr>
<td>del_allFaces()</td>
<td></td>
<td>del_tooltip()</td>
<td>663</td>
</tr>
<tr>
<td>del_allNodes()</td>
<td></td>
<td>del_tooltip(...)</td>
<td></td>
</tr>
<tr>
<td>GraphWin</td>
<td>726</td>
<td>del_bitmap()</td>
<td></td>
</tr>
<tr>
<td>del_edge(...)</td>
<td></td>
<td>DELAUNAY_DIAGRAM(...)</td>
<td>526</td>
</tr>
<tr>
<td>GraphWin</td>
<td>672</td>
<td>DELAUNAY_TRIANG(...)</td>
<td>526, 527</td>
</tr>
<tr>
<td>del_edges(...)</td>
<td></td>
<td>delete_file(...)</td>
<td>37</td>
</tr>
<tr>
<td>del_edges(...)</td>
<td></td>
<td>Delete_Loops(...)</td>
<td>331</td>
</tr>
<tr>
<td>del_item(...</td>
<td></td>
<td>del_edges_all(...</td>
<td></td>
</tr>
<tr>
<td>b_priority_queue&lt;I&gt;</td>
<td>190</td>
<td>graph</td>
<td>380</td>
</tr>
<tr>
<td>d2_dictionary&lt;K1,K2,I&gt;</td>
<td>566</td>
<td>graph_morphism_algorithm&lt;graph_t&gt;</td>
<td></td>
</tr>
<tr>
<td>dictionary&lt;K,I&gt;</td>
<td>164</td>
<td></td>
<td></td>
</tr>
<tr>
<td>interval_set&lt;I&gt;</td>
<td>574</td>
<td></td>
<td></td>
</tr>
<tr>
<td>list&lt;E&gt;</td>
<td>139</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p_dictionary&lt;K,I&gt;</td>
<td>175</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p_queue&lt;P,I&gt;</td>
<td>188</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pp_dictionary&lt;K,I,CMP&gt;</td>
<td>178</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rat_segment_set&lt;I&gt;</td>
<td>579</td>
<td></td>
<td></td>
</tr>
<tr>
<td>segment_set&lt;I&gt;</td>
<td>577</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sortseq&lt;K,I&gt;</td>
<td>183</td>
<td></td>
<td></td>
</tr>
<tr>
<td>del_menu(...)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GraphWin</td>
<td>681</td>
<td></td>
<td></td>
</tr>
<tr>
<td>del_message()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>leda_allocator&lt;T&gt;</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>denominator()</td>
<td>64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>delselect()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>delselect_allEdges()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>delselect_allNodes()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>design_pattern</td>
<td>387</td>
<td></td>
<td></td>
</tr>
<tr>
<td>destroy()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>det()</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:** The document is a list of symbols and methods, each associated with a page number and a brief description. The content appears to be related to computer science, specifically data structures and algorithms.
<table>
<thead>
<tr>
<th>Command</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix</td>
<td>105</td>
</tr>
<tr>
<td>real_matrix</td>
<td>123</td>
</tr>
<tr>
<td>det2x2(...)</td>
<td>99</td>
</tr>
<tr>
<td>residual</td>
<td>99</td>
</tr>
<tr>
<td>determinant(...)</td>
<td>112</td>
</tr>
<tr>
<td>DFS(...)</td>
<td>342</td>
</tr>
<tr>
<td>DIJKSTRA_T(...)</td>
<td>346, 347</td>
</tr>
<tr>
<td>dim(...)</td>
<td>346</td>
</tr>
<tr>
<td>diff(...)</td>
<td>346</td>
</tr>
<tr>
<td>dictionary&lt;K, I&gt;</td>
<td>163</td>
</tr>
<tr>
<td>d_int_set</td>
<td>152</td>
</tr>
<tr>
<td>GEN_POLYGON</td>
<td>464</td>
</tr>
<tr>
<td>int_set</td>
<td>150</td>
</tr>
<tr>
<td>r_circle_gen_polygon</td>
<td>562</td>
</tr>
<tr>
<td>set&lt;E&gt;</td>
<td>147</td>
</tr>
<tr>
<td>diff_approximate(...)</td>
<td>562</td>
</tr>
<tr>
<td>r_circle_gen_polygon</td>
<td>562</td>
</tr>
<tr>
<td>difference(...)</td>
<td>496</td>
</tr>
<tr>
<td>rat_rectangle</td>
<td>523</td>
</tr>
<tr>
<td>real_rectangle</td>
<td>470</td>
</tr>
<tr>
<td>r_circle_polygon</td>
<td>562</td>
</tr>
<tr>
<td>dist(...)</td>
<td>89</td>
</tr>
<tr>
<td>DIJKSTRA_T(...)</td>
<td>346</td>
</tr>
<tr>
<td>dim()</td>
<td>346</td>
</tr>
<tr>
<td>integer_vector</td>
<td>108</td>
</tr>
<tr>
<td>POINT_SET</td>
<td>569</td>
</tr>
<tr>
<td>rat_vector</td>
<td>117</td>
</tr>
<tr>
<td>real_vector</td>
<td>120</td>
</tr>
<tr>
<td>vector</td>
<td>102</td>
</tr>
<tr>
<td>dim1()</td>
<td>110</td>
</tr>
<tr>
<td>integer_matrix</td>
<td>105</td>
</tr>
<tr>
<td>matrix</td>
<td>105</td>
</tr>
<tr>
<td>real_matrix</td>
<td>123</td>
</tr>
<tr>
<td>dim2()</td>
<td>110</td>
</tr>
<tr>
<td>integer_matrix</td>
<td>105</td>
</tr>
<tr>
<td>matrix</td>
<td>123</td>
</tr>
<tr>
<td>direction(...)</td>
<td>445</td>
</tr>
<tr>
<td>line</td>
<td>445</td>
</tr>
<tr>
<td>ray</td>
<td>442</td>
</tr>
<tr>
<td>segment</td>
<td>438</td>
</tr>
<tr>
<td>disable_button(...)</td>
<td>662</td>
</tr>
<tr>
<td>window</td>
<td>662</td>
</tr>
<tr>
<td>disable_call(...)</td>
<td>682</td>
</tr>
<tr>
<td>GraphWin</td>
<td>682</td>
</tr>
<tr>
<td>disable_calls()</td>
<td>682</td>
</tr>
<tr>
<td>display(.)</td>
<td>632</td>
</tr>
<tr>
<td>display()</td>
<td>632</td>
</tr>
<tr>
<td>GeoWin</td>
<td>705</td>
</tr>
<tr>
<td>GraphWin</td>
<td>671</td>
</tr>
<tr>
<td>window</td>
<td>633</td>
</tr>
<tr>
<td>display_help_text(.)</td>
<td>633</td>
</tr>
<tr>
<td>DISREGARD_ORIENTATION</td>
<td>633</td>
</tr>
<tr>
<td>r_circle_gen_polygon</td>
<td>557</td>
</tr>
<tr>
<td>r_circle_polygon</td>
<td>551</td>
</tr>
<tr>
<td>dist(...)</td>
<td>548</td>
</tr>
<tr>
<td>r_circle_gen_polygon</td>
<td>560</td>
</tr>
<tr>
<td>r_circle_polygon</td>
<td>553</td>
</tr>
<tr>
<td>r_circle_segment</td>
<td>439</td>
</tr>
<tr>
<td>distance(...)</td>
<td>439</td>
</tr>
<tr>
<td>circle</td>
<td>451</td>
</tr>
<tr>
<td>d3_line</td>
<td>592</td>
</tr>
<tr>
<td>d3_plane</td>
<td>594</td>
</tr>
<tr>
<td>d3_point</td>
<td>583</td>
</tr>
<tr>
<td>GEN_POLYGON</td>
<td>464</td>
</tr>
<tr>
<td>line</td>
<td>445</td>
</tr>
<tr>
<td>point</td>
<td>433</td>
</tr>
<tr>
<td>POLYGON</td>
<td>456</td>
</tr>
<tr>
<td>real_circle</td>
<td>516</td>
</tr>
<tr>
<td>real_line</td>
<td>517</td>
</tr>
<tr>
<td>real_point</td>
<td>499</td>
</tr>
<tr>
<td>real_segment</td>
<td>505</td>
</tr>
<tr>
<td>segment</td>
<td>39</td>
</tr>
<tr>
<td>div(...)</td>
<td>39</td>
</tr>
</tbody>
</table>
residual .................................................. 96, 99
\texttt{do\_intersect(...)}
\texttt{rat\_rectangle} ........................................ 497
\texttt{real\_rectangle} ........................................ 524
\texttt{rectangle} ............................................ 470
\texttt{DocTools} ............................................. 761
\texttt{double\_item(...)}
\texttt{window} ............................................... 655
\texttt{double\_quotient(...)}. ................................ 77
\texttt{draw()} .................................................. ...
\texttt{d3\_window} ............................................. 730
\texttt{draw\_arc(...)}
\texttt{window} ............................................... 639
\texttt{draw\_arc\_arrow(...)}
\texttt{window} ............................................... 640
\texttt{draw\_arrow(...)}
\texttt{window} ............................................... 639
\texttt{draw\_arrow\_head(...)}
\texttt{window} ............................................... 640
\texttt{draw\_bezier(...)}
\texttt{window} ............................................... 639
\texttt{draw\_bezier\_arrow(...)}
\texttt{window} ............................................... 640
\texttt{draw\_box(...)}
\texttt{window} ............................................... 642
\texttt{draw\_circle(...)}
\texttt{window} ............................................... 640
\texttt{draw\_closed\_spline(...)}
\texttt{window} ............................................... 639
\texttt{draw\_ctext(...)}
\texttt{window} ............................................... 640
\texttt{draw\_disc(...)}
\texttt{window} ............................................... 643
\texttt{draw\_edge(...)}
\texttt{POINT\_SET} ........................................... 573
\texttt{draw\_edge\_arrow(...)}
\texttt{window} ............................................... 645
\texttt{draw\_edges(...)}
\texttt{POINT\_SET} ........................................... 573
\texttt{draw\_ellipse(...)}
\texttt{window} ............................................... 640
\texttt{draw\_filled\_circle(...)}
\texttt{window} ............................................... 641
\texttt{draw\_filled\_ellipse(...)}
\texttt{window} ............................................... 641
\texttt{draw\_filled\_node(...)}
\texttt{window} ............................................... 641
\texttt{draw\_filled\_rectangle(...)}
\texttt{window} ............................................... 642
\texttt{draw\_filled\_triangle(...)}
\texttt{window} ............................................... 643
\texttt{draw\_hline(...)}
\texttt{window} ............................................... 638
\texttt{draw\_hull(...)}
\texttt{window} ............................................... 644
\texttt{draw\_int\_node(...)}
\texttt{window} ............................................... 644
\texttt{draw\_line(...)}
\texttt{window} ............................................... 638
\texttt{draw\_node(...)}
\texttt{window} ............................................... 638
\texttt{draw\_nodes(...)}
\texttt{POINT\_SET} ........................................... 572
\texttt{draw\_oriented\_poly(...)}
\texttt{window} ............................................... 641
\texttt{draw\_pixel(...)}
\texttt{window} ............................................... 637
\texttt{draw\_pixels(...)}
\texttt{window} ............................................... 637
\texttt{draw\_point(...)}
\texttt{window} ............................................... 638
\texttt{draw\_polygon(...)}
\texttt{window} ............................................... 637
\texttt{draw\_polyline(...)}
\texttt{window} ............................................... 641
\texttt{draw\_polyline\_arrow(...)}
\texttt{window} ............................................... 640
\texttt{draw\_ray(...)}
\texttt{window} ............................................... 640
\texttt{draw\_segment(...)}
\texttt{window} ............................................... 642
\texttt{draw\_segment\_arrow(...)}
\texttt{window} ............................................... 643
\texttt{draw\_spline(...)}
\texttt{window} ............................................... 639
\texttt{draw\_triangle(...)}
\texttt{window} ............................................... 639
\texttt{draw\_segments(...)}
\texttt{window} ............................................... 638
\texttt{draw\_roundbox(...)}
\texttt{window} ............................................... 643
\texttt{draw\_rounded\_rect(...)}
\texttt{window} ............................................... 642
\texttt{draw\_segment(...)}
\texttt{window} ............................................... 638
\texttt{draw\_segments(...)}
\texttt{window} ............................................... 638
\texttt{draw\_spline(...)}
\texttt{window} ............................................... 639
<table>
<thead>
<tr>
<th>Function / Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>draw_spline_arrow(...)</td>
<td>640</td>
</tr>
<tr>
<td>draw_text(...)</td>
<td>643</td>
</tr>
<tr>
<td>draw_text_node(...)</td>
<td>644</td>
</tr>
<tr>
<td>draw_triangle(...)</td>
<td>643</td>
</tr>
<tr>
<td>draw_vline(...)</td>
<td>638</td>
</tr>
<tr>
<td>draw_voro(...)</td>
<td>573</td>
</tr>
<tr>
<td>draw_voro_edges(...)</td>
<td>573</td>
</tr>
<tr>
<td>dual()</td>
<td>446</td>
</tr>
<tr>
<td>line</td>
<td>486</td>
</tr>
<tr>
<td>real_line</td>
<td>512</td>
</tr>
<tr>
<td>dual_map()</td>
<td>277</td>
</tr>
<tr>
<td>graph</td>
<td></td>
</tr>
<tr>
<td>dx()</td>
<td>614</td>
</tr>
<tr>
<td>d3_rat_segment</td>
<td>589</td>
</tr>
<tr>
<td>d3_segment</td>
<td>478</td>
</tr>
<tr>
<td>real_segment</td>
<td>504</td>
</tr>
<tr>
<td>segment</td>
<td>438</td>
</tr>
<tr>
<td>dxD()</td>
<td></td>
</tr>
<tr>
<td>rat_segment</td>
<td>478</td>
</tr>
<tr>
<td>dy()</td>
<td>615</td>
</tr>
<tr>
<td>d3_rat_segment</td>
<td>589</td>
</tr>
<tr>
<td>d3_segment</td>
<td>478</td>
</tr>
<tr>
<td>real_segment</td>
<td>504</td>
</tr>
<tr>
<td>segment</td>
<td>438</td>
</tr>
<tr>
<td>dyD()</td>
<td></td>
</tr>
<tr>
<td>rat_segment</td>
<td>478</td>
</tr>
<tr>
<td>dynamic_markov_chain</td>
<td>333</td>
</tr>
<tr>
<td>dynamic_random_variate</td>
<td>29</td>
</tr>
<tr>
<td>dynamic_trees</td>
<td>159</td>
</tr>
<tr>
<td>dz()</td>
<td></td>
</tr>
<tr>
<td>d3_rat_segment</td>
<td>615</td>
</tr>
<tr>
<td>d3_segment</td>
<td>590</td>
</tr>
<tr>
<td>E</td>
<td></td>
</tr>
<tr>
<td>static_graph</td>
<td>288</td>
</tr>
<tr>
<td>edge_compat</td>
<td></td>
</tr>
<tr>
<td>graph_morphism_algorithm&lt; graph_t &gt;</td>
<td>379</td>
</tr>
<tr>
<td>edge_data()</td>
<td></td>
</tr>
<tr>
<td>GRAPH&lt;vtype, e...&gt;</td>
<td>283</td>
</tr>
<tr>
<td>edge_inf()</td>
<td></td>
</tr>
<tr>
<td>dynamic_trees</td>
<td>159</td>
</tr>
<tr>
<td>edge_morphism</td>
<td></td>
</tr>
<tr>
<td>graph_morphism_algorithm&lt; graph_t &gt;</td>
<td>379</td>
</tr>
<tr>
<td>edge_value_type</td>
<td></td>
</tr>
<tr>
<td>GEN_POLYGON</td>
<td>461</td>
</tr>
<tr>
<td>edges()</td>
<td></td>
</tr>
<tr>
<td>GeoWin</td>
<td>704</td>
</tr>
<tr>
<td>GraphWin</td>
<td>671</td>
</tr>
<tr>
<td>edit()</td>
<td></td>
</tr>
<tr>
<td>elapsed_time()</td>
<td>44</td>
</tr>
<tr>
<td>elapsed_time(...)</td>
<td>44</td>
</tr>
<tr>
<td>element_type</td>
<td></td>
</tr>
<tr>
<td>d_array&lt;I, E&gt;</td>
<td>166</td>
</tr>
<tr>
<td>map2&lt;I1, I2, E&gt;</td>
<td>173</td>
</tr>
<tr>
<td>map&lt;I, E&gt;</td>
<td>171</td>
</tr>
<tr>
<td>eliminate_cocircu(...)</td>
<td></td>
</tr>
<tr>
<td>r_circle_gen_polygon</td>
<td>560</td>
</tr>
<tr>
<td>r_circle_polygon</td>
<td>553</td>
</tr>
<tr>
<td>EMPTY</td>
<td></td>
</tr>
<tr>
<td>r_circle_gen_polygon</td>
<td>556</td>
</tr>
<tr>
<td>r_circle_polygon</td>
<td>550</td>
</tr>
<tr>
<td>empty()</td>
<td></td>
</tr>
<tr>
<td>b_priority_queue&lt;I&gt;</td>
<td>191</td>
</tr>
<tr>
<td>b_queue&lt;E&gt;</td>
<td>135</td>
</tr>
<tr>
<td>b_stack&lt;E&gt;</td>
<td>134</td>
</tr>
<tr>
<td>d2_dictionary&lt;K1, K2, I&gt;</td>
<td>566</td>
</tr>
<tr>
<td>d_int_set</td>
<td>152</td>
</tr>
</tbody>
</table>
165
176
188
454
178
133
579
577
148
145
182
132
2
662
682
682
661
675
726
662
196
199
211
238
215
254
224
232
204
207
209
220
218
258
202
222
213
197
199
211
238
216
255
225
222
213
229
228
477
504
438
414
381
384
383
405
396
398
393
389
394
395
388
387
386
385
384
383
382
381
380
379
378
377
376
375
374
373
372
371
370
369
368
367
366
365
364
363
362
361
360
359
358
357
356
355
354
353
352
351
350
349
348
347
346
345
344
343
342
341
340
339
338
337
336
335
334
333
332
331
330
329
328
327
326
325
324
323
322
321
320
319
318
317
316
315
314
313
312
311
310
309
308
307
306
305
304
303
302
301
300
299
298
297
296
295
294
293
292
291
290
289
288
287
286
285
284
283
282
281
280
279
278
277
276
275
274
273
272
271
270
269
268
267
266
265
264
263
262
261
260
259
258
257
256
255
254
253
252
251
250
249
248
247
246
245
244
243
242
241
240
239
238
237
236
235
234
233
232
231
230
229
228
227
226
225
224
223
222
221
220
219
218
217
216
215
214
213
212
211
210
209
208
207
206
205
204
203
202
201
200
199
198
197
196
195
194
193
192
191
190
189
188
187
186
185
184
183
182
181
180
179
178
177
176
175
174
173
172
171
170
169
168
167
166
165
164
163
162
161
160
159
158
157
156
155
154
153
152
151
150
149
148
147
146
145
144
143
142
141
140
139
138
137
136
135
134
133
132
131
130
129
128
127
126
125
124
123
122
121
120
119
118
117
116
115
114
113
112
111
110
109
108
107
106
105
104
103
102
101
100
99
98
97
96
95
94
93
92
91
90
89
88
87
86
85
84
83
82
81
80
79
78
77
76
75
74
73
72
71
70
69
68
67
66
65
64
63
62
61
60
59
58
57
56
55
54
53
52
51
50
49
48
47
46
45
44
43
42
41
40
39
38
37
36
35
34
33
32
31
30
29
28
27
26
25
24
23
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1

encode_memory_chunk(...)
INDEX

InAdjIt ................................................. 403
NodeIt ................................................. 394
OutAdjIt ................................................. 400
equal_as_sets(...) 480, 487, 490, 549
erase(...) 480, 487, 490, 549
list<E> ................................................ 139
error ................................................ 34
error_handler(...) .................. 34
Euler_Tour(...) .................. 371
euler_tour .......................... 371
dynamic_trees ........................................ 160
extract(...) .............................................. 140

F
F_DELAUNAY_DIAGRAM(...) 526
F_DELAUNAY_TRIANG(...) 526
F_VORONOI(...) 530
face_cycle_pred(...) 276
face_cycle_succ(...) 276
face_of(...) 281
graph .............................................. 277
face_array<E> .................................. 302
face_map<E> .................................. 308
FaceIt ................................................. 396
factorial(...) ........................................ 77
failed() .............................................. 381

PPMIICoder ........................................... 203
RLE0Coder .......................................... 223
RLECoder .......................................... 214
secure_socket_streambuf 264
socket_streambuf 42
fbUTTON(...) ........................................ 659
window ............................................. 659
filter_nodeIt<predica...> 407
find(...) node_partition 318
partition 155
Partition<E> ........................................ 157
find_all_is0(...) 382

find_all_mono(...) 384
find_all_sub(...) 382
find_iso(...) 380
find_min() 381

A0Coder .............................................. 197
A0sCoder ............................................. 200
AdaptiveHuffmanCoder 212
AutoDecoder 236
BlockCoder<Coder> 238
BWTCoder 216
CBCCoder<BkCipher> 255
checksummer_base 225
CoderPipe2<Coder1,...> 233
CryptAutoDecoder 261
DeflateCoder 205
DictCoder 208
HuffmanCoder 210
MTF2Coder 221
MTFCoder 219
OMACCoder<BkCipher> 259

b_priority_queue<I> 190
nodepq<P> 319
p_queue<P, I> 188

find_mono(...) 383
find_sub(...) 382
findcost(...) 381

tree_collection<I> 161
findroot(...) 161
tree_collection<I> 161

sortseq<K, I> 180, 181
INDEX

...snipped...

G

galil_seiferas() ... 67

garner_sign() ... 98
gcd(...) ... 77

GEN_POLYGON ... 458
genera_levenshtein() ... 71
generate() ... 29

dynamic_random_variate ... 29

random_variate ... 29

generate_key(...) ... 250, 251

generate_key_and...( ... 251
generate_salt(...) ... 251

Genus(...) ... 329
geo_alg ... 525

Geometry

delaunay tree ... 733

interval tree ... 733

priority search tree ... 733

range tree ... 733

segment tree ... 733

GeoWin ... 693

get() ...

random_source ... 27

get(...)(...) ... 415, 417–419

array<E> ... 128

get_accepted_key...( ... 256

OMACCoder< BlkCipher > ... 259

get_action(...) ... 719

GeoWin ... 680

GraphWin ... 662

get_active_line(...) ... 662

d3_window ... 732

get_bg_pixel() ... 717

GeoWin ... 717

get_bigfloat_error() ...

real ... 86

get_block_cipher() ...

CBCCoder< BlkCipher > ... 256

get_block_size() ...

BlockCoder<Coder> ... 239

BWTCoder ... 216

checksummer_base ... 226

get_bounding_box(...) ...

GraphWin ... 685

POINT_SET ... 569

get_buffer() ...

memory_streambuf ... 242

get_button(...) ...

window ... 662

get_button_label(...) ...

window ... 662

get_bytes() ...

CryptoByteString ... 248

get_call_button()
window ......................................... 661
get_call_item() window ......................................... 661
get_call_window() window ......................................... 661
get_capacity() memory_streambuf ......................... 241
get_checksums() checksum_base ....................... 226
get_client_data() GeoWin ........................................ 710
window ......................................... 636
get_coder() AutoDecoder .................................. 236
BlockCoder< Coder > .................................. 239
CryptAutoDecoder .................................. 261
decoding< fstream< Coder > ......................... 231
decoding< istream< Coder > ......................... 229
encoding< ostream< Coder > ......................... 230
encoding< ostream< Coder > ......................... 228
get_coder1() CoderPipe2< Coder1, ... > ............ 233
get_coder2() CoderPipe2< Coder1, ... > ............ 234
get_color(...) d3_window .................................. 732
GeoWin ........................................... 708
get_compression(...) DeflateCoder .................... 205
get_convex_hull() POINT_SET ....................... 569
get_cursor() window ....................................... 636
get_cyclic_colors(...) GeoWin ......................... 710
get_d2_position(...) d3_window ......................... 732
get_d3_elimination() GeoWin ......................... 718
get_d3_fcn(...) GeoWin ..................................... 727
get_d3_show_edges() GeoWin ......................... 718
get_d3_solid() GeoWin ..................................... 718
get_date() date ........................................ 59
get_date() date ........................................ 59
get_day() date ........................................ 59
get_day_in_year() date .................................... 60
get_day_of_week() date .................................... 60
get_default_key_size() DeflateCoder .................. 206
get_default_value() map< I, E > ......................... 171
get_description() A0Coder ................................ 197
A0sCoder .......................................... 200
AdaptiveHuffmanCoder ............................... 212
AutoDecoder ....................................... 236
BlockCoder< Coder > .................................. 239
BWTCodec .......................................... 216
CBCCoder< BlkCipher > .............................. 255
OMACCoder< BlkCipher > ............................ 259
checksum_base ...................................... 225
DictCoder .......................................... 208
HuffmanCoder ...................................... 210
MTF2Coder .......................................... 221
MTFCoder .......................................... 219
OMACCoder< BlkCipher > ............................ 259
PPMIICoder ......................................... 203
RLE0Coder .......................................... 223
RLECoder .......................................... 214
get_description(...) GeoWin ......................... 710
get_directories() ......................... 36
get_directory() ......................... 36
get_directory_del(...) ......................... 36
get_double_error() interval ......................... 93
real .......................................... 86
get_double_lower(...) real ......................... 86
get_double_upper(...) real ......................... 86
GeoWin .................................. 711
get_initialization() ..................... 711

CBCCoder< BlkCipher > ............... 256
get_input_format() ...................... 59
get_input_format_str() ............... 59
get_ip() .................................. 59
get_item(...) ............................. 137
window .................................. 662

get_key() ................................ 256
OMACCoder< BlkCipher > .............. 259
get_keys() ................................ 261

CryptAutoDecoder ...................... 261
get_language() ..................... 58
get_limit() .................................. 58
get_limit(...) ............................. 727
GeoWin .................................. 709
get_line_style() ......... 636
GeoWin .................................. 709
get_line_style(...) .......................... 709
get_line_width() ......... 636
GeoWin .................................. 709
get_line_width(...) .......................... 709
get_lower_bound() ............. 86
real ........................................ 86
get_MACas_hex_st...( ) ................. 259
OMACCoder< BlkCipher > .............. 259
get_MACin_stream...( ) ................. 259
OMACCoder< BlkCipher > .............. 259
get_max_scale_thr...( ) ............... 198
A0Coder ................................ 198
get_maximal_bit...( ) ......... 97
get_memory_consum...( ) ............. 712, 713

DeflateCoder ....................... 206
get_memory_level() ...................... 206
DeflateCoder ....................... 206
get_menu(...) ................................ 682
GraphWin ................................ 682
get_message() ..................... 732
d3_window ....................... 732
GraphWin ................................ 671
get_mode() ..................... 636
window ................................ 636
get_model_memory...( ) ................ 203
PPMIICoder ....................... 203
get_model_order() ...................... 203
PPMIICoder ....................... 203
get_model_restora...( ) .............. 203
PPMIICoder ....................... 203
get_month() .................................. 59
get_month_name() ............ 60
get_mouse() ..................... 650
get_mouse(...). ..................... 650
counter ................................ 49
timer ................................ 47
get_name() ..................... 708
GeoWin .................................. 708
get_node() ..................... 403
AdjIt ................................ 405
InAdjIt ................................ 403
NodeIt ................................ 394
OutAdjIt ................................ 400
get_node_param() ...................... 673
get_node_width() ...................... 673
window ................................ 636
get_nodes_in_area...( ) ............. 636
GraphWin .................................. 684
get_num_calls() ......... 380

graph_morphism_algorithm< graph_t > .... 380
get_obj_color...( ) ......... 712, 713
GeoWin .................................. 712, 713
get_obj_fill_color...( ) ......... 713
GeoWin .................................. 713
get_obj_label...( ) ......... 714
GeoWin .................................. 714
get_obj_line_style...( ) ......... 714
GeoWin .................................. 714
INDEX

get\_obj\_line\_width(...) .............................. GeoWin ... 714
get\_obj\_text(...) ................................. GeoWin ... 715, 716
get\_objects(...) ................................. GeoWin ... 728
get\_observer() ............................... Observer\_NodeIt<Obs, Iter> ... 411
get\_out\_stack(...) .......................... GIT\_SCC<Out, In, ...> ... 427
get\_out\_going\_pack(...) ........................... secure\_socket\_streambuf ... 264
socket\_streambuf ............................. 42
get\_output\_format(...) ........................... date ... 59
get\_output\_format(...) ........................... date ... 59
get\_param(...) ................................. GraphWin ... 672, 673
get\_pin\_point(...) ............................ GeoWin ... 727
get\_pix\_rect(...) ............................... window ... 645
get\_point\_style(...) ............................ GeoWin ... 710
get\_port() ................................. leda\_socket ... 39
get\_position(...) .............................. GraphWin ... 676
get\_precision() ................................. bigfloat ... 81
random\_source ............................... 28
get\_primetable() ............................ residual ... 99
get\_q\_length() ................................. leda\_socket ... 39
get\_queue() ..................................... GIT\_BFS<OutAdjI...> ... 421
GIT\_DIJKSTRA<OutAdjI...> ... 428
GIT\_TOPOSORT<OutAdjI...> ... 425
get\_representation() ... 99
get\_reset\_threshold() .......................... A0Coder ... 197
get\_scale\_threshold() .......................... A0Coder ... 197
get\_scene\_with\_name(...) ............................ GeoWin ... 707
get\_scene\_groups() ............................. GeoWin ... 725
get\_scenes(...) ............................... GeoWin ... 725
get\_scenes() ................................. GeoWin ... 725
get\_selected\_edges() ............................ GraphWin ... 676
get\_selected\_nodes() ........................... GraphWin ... 675
get\_selected\_objects(...) ........................... GeoWin ... 703
get\_selection\_color(...) ............................ GeoWin ... 708
get\_selection\_fill(...) ........................... GeoWin ... 708
get\_selection\_lin(...) ........................... GeoWin ... 708
get\_show\_grid() .............................. GeoWin ... 717
get\_show\_orientation() .......................... window ... 636
get\_show\_orientation(...) ............................ GeoWin ... 710
get\_show\_position() ............................. GeoWin ... 717
get\_show\_status() .............................. GeoWin ... 706
get\_significant(...) ............................ bigfloat ... 81
get\_significant\_l(...) ... 81
get\_size() ................................. bigfloat ... 82
gt\_solid() ................................. d3\_window ... 731
gt\_src\_stream() .............................. A0Coder ... 197
A0sCoder ... 200
<table>
<thead>
<tr>
<th>Class</th>
<th>Page</th>
<th>Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdaptiveHuffmanCoder</td>
<td>212</td>
<td>get_timeout()</td>
<td></td>
</tr>
<tr>
<td>AutoDecoder</td>
<td>235</td>
<td>leda_socket</td>
<td>39</td>
</tr>
<tr>
<td>BlockCoder&lt; Coder &gt;</td>
<td>238</td>
<td>get_upper_bound()</td>
<td></td>
</tr>
<tr>
<td>BWTCoder</td>
<td>216</td>
<td>real</td>
<td>86</td>
</tr>
<tr>
<td>CBCCoder&lt; BlkCipher &gt;</td>
<td>255</td>
<td>get_user_layer_color()</td>
<td></td>
</tr>
<tr>
<td>checksum_base</td>
<td>225</td>
<td>GeoWin</td>
<td>717</td>
</tr>
<tr>
<td>CoderPipe2&lt; Coder1,...&gt;</td>
<td>233</td>
<td>get_user_layer_li...()</td>
<td></td>
</tr>
<tr>
<td>CryptAutoDecoder</td>
<td>260</td>
<td>GeoWin</td>
<td>717</td>
</tr>
<tr>
<td>DeflateCoder</td>
<td>205</td>
<td>get_value()</td>
<td></td>
</tr>
<tr>
<td>DictCoder</td>
<td>207</td>
<td>counter</td>
<td>49</td>
</tr>
<tr>
<td>HuffmanCoder</td>
<td>210</td>
<td>get_visible()</td>
<td></td>
</tr>
<tr>
<td>MTF2Coder</td>
<td>221</td>
<td>GeoWin</td>
<td>709</td>
</tr>
<tr>
<td>MTFCoder</td>
<td>219</td>
<td>get_visible_scenes()</td>
<td></td>
</tr>
<tr>
<td>OMACCoder&lt; BlkCipher &gt;</td>
<td>258</td>
<td>GeoWin</td>
<td>726</td>
</tr>
<tr>
<td>PPMIIICoder</td>
<td>202</td>
<td>get_week()</td>
<td></td>
</tr>
<tr>
<td>RLE0Coder</td>
<td>223</td>
<td>date</td>
<td>60</td>
</tr>
<tr>
<td>RLECoder</td>
<td>214</td>
<td>get_width(...</td>
<td></td>
</tr>
<tr>
<td>get_stack()</td>
<td></td>
<td>window</td>
<td>645</td>
</tr>
<tr>
<td>GIT_DFS&lt; OutAdjI,...&gt;</td>
<td>424</td>
<td>get_window()</td>
<td></td>
</tr>
<tr>
<td>get_state()</td>
<td></td>
<td>GeoWin</td>
<td>705</td>
</tr>
<tr>
<td>window</td>
<td>637</td>
<td>GraphWin</td>
<td>672</td>
</tr>
<tr>
<td>get_strategy()</td>
<td></td>
<td>get_window(...)</td>
<td></td>
</tr>
<tr>
<td>DeflateCoder</td>
<td>206</td>
<td>window</td>
<td>662</td>
</tr>
<tr>
<td>get_text_color(...)</td>
<td></td>
<td>get_window_pixrect()</td>
<td></td>
</tr>
<tr>
<td>GeoWin</td>
<td>709</td>
<td>window</td>
<td>645</td>
</tr>
<tr>
<td>get_text_mode()</td>
<td></td>
<td>get_window_size()</td>
<td></td>
</tr>
<tr>
<td>window</td>
<td>636</td>
<td>DeflateCoder</td>
<td>206</td>
</tr>
<tr>
<td>get_tgt_stream()</td>
<td></td>
<td>get_x_rotation()</td>
<td></td>
</tr>
<tr>
<td>A0Coder</td>
<td>197</td>
<td>d3_window</td>
<td>731</td>
</tr>
<tr>
<td>A0sCoder</td>
<td>200</td>
<td>get_xmax()</td>
<td></td>
</tr>
<tr>
<td>AdaptiveHuffmanCoder</td>
<td>212</td>
<td>GeoWin</td>
<td>705</td>
</tr>
<tr>
<td>AutoDecoder</td>
<td>236</td>
<td>GraphWin</td>
<td>671</td>
</tr>
<tr>
<td>BlockCoder&lt; Coder &gt;</td>
<td>238</td>
<td>get_xmin()</td>
<td></td>
</tr>
<tr>
<td>BWTCoder</td>
<td>216</td>
<td>GeoWin</td>
<td>705</td>
</tr>
<tr>
<td>CBCCoder&lt; BlkCipher &gt;</td>
<td>255</td>
<td>GraphWin</td>
<td>671</td>
</tr>
<tr>
<td>checksum_base</td>
<td>225</td>
<td>get_y_rotation()</td>
<td></td>
</tr>
<tr>
<td>CoderPipe2&lt; Coder1,...&gt;</td>
<td>233</td>
<td>d3_window</td>
<td>731</td>
</tr>
<tr>
<td>CryptAutoDecoder</td>
<td>261</td>
<td>get_year()</td>
<td></td>
</tr>
<tr>
<td>DeflateCoder</td>
<td>205</td>
<td>date</td>
<td>60</td>
</tr>
<tr>
<td>DictCoder</td>
<td>208</td>
<td>get_ymax()</td>
<td></td>
</tr>
<tr>
<td>HuffmanCoder</td>
<td>210</td>
<td>GeoWin</td>
<td>705</td>
</tr>
<tr>
<td>MTF2Coder</td>
<td>221</td>
<td>GraphWin</td>
<td>671</td>
</tr>
<tr>
<td>MTFCoder</td>
<td>219</td>
<td>get_ymin()</td>
<td></td>
</tr>
<tr>
<td>OMACCoder&lt; BlkCipher &gt;</td>
<td>258</td>
<td>GeoWin</td>
<td>705</td>
</tr>
<tr>
<td>PPMIIICoder</td>
<td>202</td>
<td>GraphWin</td>
<td>671</td>
</tr>
<tr>
<td>RLE0Coder</td>
<td>223</td>
<td>get_z_order(...)</td>
<td></td>
</tr>
<tr>
<td>RLECoder</td>
<td>214</td>
<td>GeoWin</td>
<td>707</td>
</tr>
<tr>
<td>FUNCTION/CLASS</td>
<td>PAGE(s)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------------------------</td>
<td>-----------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>secure_socket_streambuf</td>
<td>264</td>
<td></td>
<td></td>
</tr>
<tr>
<td>streambuf</td>
<td>42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hash(...)</td>
<td>52-54</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hash(...)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CRYPTBYTESTRING</td>
<td>249</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hash(...) see User defined parameter types</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hashed Types see h_array, map2, map, see User defined parameter types</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hashing see User defined parameter types</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dynamic and perfect</td>
<td>733</td>
<td></td>
<td></td>
</tr>
<tr>
<td>with chaining</td>
<td>733</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hasScore(...)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>score_matrix</td>
<td>68</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hcoord(...)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d3_rat_point</td>
<td>601</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rat_vector</td>
<td>117</td>
<td></td>
<td></td>
</tr>
<tr>
<td>real_vector</td>
<td>120</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vector</td>
<td>102</td>
<td></td>
<td></td>
</tr>
<tr>
<td>head()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>list&lt;E&gt;</td>
<td>138</td>
<td></td>
<td></td>
</tr>
<tr>
<td>node_list</td>
<td>316</td>
<td></td>
<td></td>
</tr>
<tr>
<td>slist&lt;E&gt;</td>
<td>146</td>
<td></td>
<td></td>
</tr>
<tr>
<td>head(...)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>height()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rat_rectangle</td>
<td>495</td>
<td></td>
<td></td>
</tr>
<tr>
<td>real_rectangle</td>
<td>522</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rectangle</td>
<td>469</td>
<td></td>
<td></td>
</tr>
<tr>
<td>window</td>
<td>637</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hex_print(...)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>integer</td>
<td>76</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hidden_edges()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>graph</td>
<td>272</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hidden_nodes()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>graph</td>
<td>272</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hide_edge(...)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>graph</td>
<td>272</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hide_edges(...)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>graph</td>
<td>272</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hide_node(...)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>graph</td>
<td>273</td>
<td></td>
<td></td>
</tr>
<tr>
<td>high()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>array&lt;E&gt;</td>
<td>128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>real</td>
<td>86</td>
<td></td>
<td></td>
</tr>
<tr>
<td>high1()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>array2&lt;E&gt;</td>
<td>131</td>
<td></td>
<td></td>
</tr>
<tr>
<td>high2()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>array2&lt;E&gt;</td>
<td>131</td>
<td></td>
<td></td>
</tr>
<tr>
<td>highword()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>integer</td>
<td>76</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hilbert()</td>
<td>457</td>
<td></td>
<td></td>
</tr>
<tr>
<td>homogeneous_linea(...)</td>
<td>113</td>
<td></td>
<td></td>
</tr>
<tr>
<td>horspool()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>string_matching</td>
<td>67</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HUFFMANCODER</td>
<td>209</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

I

<table>
<thead>
<tr>
<th>FUNCTION/CLASS</th>
<th>PAGE(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>identity(...)</td>
<td>110</td>
</tr>
<tr>
<td>integer_matrix</td>
<td></td>
</tr>
<tr>
<td>ilog2(...)</td>
<td>83</td>
</tr>
<tr>
<td>improve_approxima(...)</td>
<td>87</td>
</tr>
<tr>
<td>real</td>
<td></td>
</tr>
<tr>
<td>in_avail()</td>
<td></td>
</tr>
<tr>
<td>memory_streambuf</td>
<td>242</td>
</tr>
<tr>
<td>in_count()</td>
<td></td>
</tr>
<tr>
<td>memory_streambuf</td>
<td>242</td>
</tr>
<tr>
<td>in_current()</td>
<td></td>
</tr>
<tr>
<td>GIT_SCC&lt;Out, In, ...&gt;</td>
<td>427</td>
</tr>
<tr>
<td>in_edges(...)</td>
<td></td>
</tr>
<tr>
<td>graph</td>
<td>270</td>
</tr>
<tr>
<td>in_pred(...)</td>
<td></td>
</tr>
<tr>
<td>graph</td>
<td>271</td>
</tr>
<tr>
<td>in_simplex(...)</td>
<td></td>
</tr>
<tr>
<td>d3_rat_simplex</td>
<td>623</td>
</tr>
<tr>
<td>d3_simplex</td>
<td>599</td>
</tr>
<tr>
<td>in_succ(...)</td>
<td></td>
</tr>
<tr>
<td>graph</td>
<td>271</td>
</tr>
<tr>
<td>InAdjIt</td>
<td>401</td>
</tr>
<tr>
<td>incircle(...)</td>
<td></td>
</tr>
<tr>
<td>include(...)</td>
<td></td>
</tr>
<tr>
<td>rat_rectangle</td>
<td>495, 496</td>
</tr>
<tr>
<td>real_rectangle</td>
<td>522</td>
</tr>
<tr>
<td>rectangle</td>
<td>469</td>
</tr>
<tr>
<td>increment()</td>
<td></td>
</tr>
<tr>
<td>counter</td>
<td>49</td>
</tr>
<tr>
<td>indeg(...)</td>
<td></td>
</tr>
<tr>
<td>graph</td>
<td>269</td>
</tr>
<tr>
<td>static_graph</td>
<td>290</td>
</tr>
<tr>
<td>independent_columns(...)</td>
<td>113</td>
</tr>
<tr>
<td>INDEPENDENT_SET(...)</td>
<td>373</td>
</tr>
<tr>
<td>index(...)</td>
<td></td>
</tr>
<tr>
<td>d3_rat_simplex</td>
<td>622</td>
</tr>
<tr>
<td>d3_simplex</td>
<td>598</td>
</tr>
</tbody>
</table>
INDEX

index_type

d_array<1, E> .................................. 166
map<1, E> .................................. 171

dictionary

pq

set

DFS

queue

map

DIJKSTRA

int

map

map2

array

set

collection

map

set

pq

queue

queue

rectangle(...)

map

SET

dictionary

set

list

segment

array

set

array

queue

map

set

type2

queue

set

map2

dictionary

set

dictionary

MAP

init(...) .................................. 797

GeoWin .................................. 705

GIT_DFS<OutAdjI_...> .................. 424

GIT_DIJKSTRA<OutAdjI_...> ........... 428

graph .................................. 269

InAdjIt .................................. 402

node_array<E> .................. 298, 299

node_map2<E> .................. 312

node_map<E> .................. 304

node_matrix<E> .................. 310

NodeIt .................................. 393

ObserverNodeIt<Obs, Iter> ........... 411

OutAdjIt .................................. 399

POINT_SET .................................. 569

window .................................. 632

init_menu(...) .................................. 320

GeoWin .................................. 706

init_rectangle(...) .................................. 175

init(...) .................................. 188

Partition<E> .................. 158

window .................................. 632

PLANAR_MAP<type, e...> .......... 296

insert() .................................. 574

NodeIt .................................. 394

insert(...) .................................. 146

b_node_pg<N> .................. 321

b_priority_queue<I> .................. 190

d2_dictionary<K1, K2, I> .......... 566

dictionary<K, I> .................. 164

EdgeIt .................................. 395

GeoWin .................................. 724

InAdjIt .................................. 402

int_set .................................. 150

interval_set<I> .................. 574

list<E> .................. 138

list<E> .................. 146

list<E> .................. 180

list<E> .................. 580

tree_collection<I> .................. 161

segment_set<I> .................. 576

AdjIt .................................. 405

d_int_set .................. 152

d2_dictionary<K1, K2, I> .......... 566

dictionary<K, I> .................. 164

d_segment_set<I> .................. 319

d_segment_set<I> .................. 310

d_segment_set<I> .................. 411

d_set<E> .................. 578

d_segment_set<I> .................. 576

d_segment_set<I> .................. 574

d_segment_set<I> .................. 306

d_segment_set<I> .................. 308

d_segment_set<I> .................. 312

d_segment_set<I> .................. 304

init(...) .................................. 128

AdjIt .................................. 404

array<E> .................. 128

d3_window .................. 730

d3_window .................. 300, 301

d3_window .................. 306

EdgeIt .................................. 395

face_array<E> .................. 302, 303

EdgeIt .................................. 395

face_array<E> .................. 308

FaceCirc .................. 407

FaceIt .................................. 397

FilterNodeIt<Predicate...> ...... 408

GeoWin .................................. 705

INT_INDEX
segment .............................. 438  r_circle_point ............................ 544
is_hull_dart(...) .......................... 570  is_rat_polygon() ................................
POINT_SET .............................. 570  r_circle_polygon ............................ 553
is_hull_edge(...) .......................... 570  is_rat_segment() ............................ 547
POINT_SET .............................. 570  r_circle_segment ............................ 547
is_in_fast_mode() ......................... 214  is_rational() ................................ 86
 RLECoder ............................... 214  is_running() .................................. 47
is_invertible() ............................ 98  is_segment() .................................. 495
residual ................................. 98  rat_rectangle .............................. 495
is_last_day_in_month() .................... 61  real_rectangle .............................. 522
 date ................................... 61  rectangle ................................. 469
is_leap_year(...) ........................... 61  is_selected(...) .............................. 675, 676
 date ................................... 61  GraphWin .................................... 675, 676
is_line() .................................. 449  Is_Series_Parallel(...) ....................... 330
 circle .................................. 449  is_simple() .................................. 330
 rat_circle ................................ 489  Is_SIMPLE(... ................................ 328
real_circle ............................... 515  IsSimple(... ................................ 328
is_link(...) ............................... 37  Is_loopfree(... ................................ 328
 integer ................................. 76  IsSimpleLoopfree(... ......................... 328
 residual ............................... 98  IsSimplePolygon(...) ......................... 533
Is_Loopfree(...). . . . . . . . . . . . . . . . . . 328  isSolvable(...). . . . . . . . . . . . . . . . . . 113
Is_map() .................................. 276  isSpace(...) ................................. 44
graph .................................... 276  isStraightSegment() .......................... 547
Is_Map(...) ............................... 329  r_circle_segment ............................ 547
is_open() ................................ 231  is_subgraph_isomorphism(...) ............. 229
 decoding_ofstream<Coder> ............... 231  r_circle_segment ............................ 547
 encoding_ofstream<Coder> ............... 229  is_trivialial() .............................. 385
Is_Planar(...) ............................. 330  graph_morphism_algorithm< graph t > 385
Is_Planar_Map(...) ....................... 330  is_triangle(...) .............................. 330
Is_Plane_Map(...) ....................... 329  Is_Triangulation(...) ...................... 529
is_point() ................................ 495  IsTricnected(...) ......................... 329, 330
 rat_rectangle ............................ 495  is_trivialial() .............................. 385
real_rectangle ............................ 522  circle .................................. 449
triangle .................................. 469  d3_rat_segment ............................ 449
d3_segment ............................... 590  d3_rat_segment ............................ 615
r_circle_polygon .......................... 558  d3_segment ............................... 590
r_circle_polygon .......................... 558  r_circle_segment ............................ 547
r_circle_polygon .......................... 558  rat_circle ................................. 489
r_circle_polygon .......................... 558  rat_segment ............................... 478
r_circle_polygon .......................... 558  real_circle ............................... 515
r_circle_polygon .......................... 554  real_segment ............................... 504
r_circle_polygon .......................... 560  segment ................................. 438
is_rat_point() ............................ 560  is_undirected() ............................. 271
is UndirectedSimple() ............... 328
is valid(...) .......................... 61
is vertical() ........................... 445
  line .................................. 445
  real line ............................. 485
  real ray ............................. 482
  real segment ........................ 511
  real ray ............................. 508
  real segment ........................ 504
  segment ............................. 438
is vertical_segment() ............................ 547
is VoronoiDiagram(...) .......................... 531
is weakly simple() .............................. 453
  POLYGON ............................... 453
  r_circle_gen_polygon .................. 558
  r_circle_polygon ..................... 558
is weakly simple(...) .............................. 453
  POLYGON ............................... 453
  r_circle_gen_polygon .................. 558
  r_circle_polygon ..................... 558
is zero() ................................ 98
  residual .............................. 98
isDNA() .................................. 64
  alphabet ............................. 64
isFinished() ............................... 64
  alphabet ............................. 64
isInf(...) ................................ 83
isNaN(...) ................................ 82
isInf(...) ................................ 82
isZero(...) ................................ 83
ispInf(...) ................................ 83
ispZero(...) ................................ 83
isSpecial(...) ............................ 83
isStandard() .............................. 64
  alphabet ............................. 64
istream .................................. 21
iszero() ................................ 77
  integer ............................... 77
isZero(...) ................................ 83
item .................................... 14
  array<E> .............................. 127
  d_array<I, E> ......................... 166
dictionary<K, I> ......................... 163
  list<E> ................................. 137
  map2<I1, I2, E> ....................... 173
  map<I, E> .............................. 171
  p_queue<P, I> ........................... 187
  slist<E> ............................... 145
  sortseq<K, I> ......................... 179
iteration ............................... 179
  Graph iterator ........................ 17
  macros ............................... 16
  STL iterators .......................... 16
iterator ................................ 387
J
J
Jan
  date .................................. 56
join(...) ............................... 152
  d_int_set ............................. 276
  graph ............................... 276
  int_set .............................. 150
  set<E> ............................... 147
join_faces(...) .......................... 278
  graph ............................... 278
K
  karp_rabin() ........................... 67
  string_matching ........................ 67
key(...) ................................ 98
  dictionary<K, I> ...................... 164
  p_dictionary<K, I> .................... 175
  pp_dictionary<K, I, CMP> ............. 178
  rap_segment_set<I> .................... 578
  segment_set<I> ....................... 576
  sortseq<K, I> ......................... 180
key1(...) ............................... 152
  d2_dictionary<K1, K2, I> ............. 565
key2(...) ............................... 152
  d2_dictionary<K1, K2, I> ............. 566
key_type ............................... 98
  dictionary<K, I> ...................... 163
  sortseq<K, I> ......................... 179
  KIND ................................. 98
  r_circle_gen_polygon .................. 556
  r_circle_polygon ..................... 550
  kind() ............................... 98
    GEN_POLYGON ........................ 460
    r_circle_gen_polygon ................ 558
    r_circle_polygon .................... 551
knuth_morris_pratt() ..........................
INDEX

string_matching ........................................... 67
KURATOWSKI(...) ........................................... 373

L

lagrange_sign() ........................................... 98
language

date ............................................... 56
LARGEST_EMPTY_CIRCLE(...) ................................ 530
last()

  list<E> ........................................... 137
  slist<E> ........................................... 145
  STLNodeIt<DataAcc...> ......................... 414
last_adj_edge(...) ........................................ 270
last_edge()

  graph ........................................... 270
last_face()

  graph ........................................... 277
last_in_edge(...)

  graph ........................................... 271
last_node()

  graph ........................................... 270
lattice_d3_rat_po(...)

  (....) ........................................... 607
lattice_points(...)

  (....) ........................................... 542
lca(...)

  dynamic_trees .................................... 160
ldel ........................................... 764
leda_assert(...) ........................................ 35
leda_allocator<I> ........................................ 32
leda_socket ........................................... 38
left(...)

  interval_set<I> .................................... 574
left_tangent(...)

  circle ........................................... 450
  real_circle ...................................... 516
left_turn(...)

  (....) ........................................... 435, 474, 501
length()

  b_queue<E> ........................................... 135
d3_segment ........................................... 590
integer ........................................... 76
list<E> ........................................... 137
queue<E> ........................................... 133
real_segment ........................................... 504
real_vector ........................................... 120
residual ........................................... 98
segment ........................................... 438

slist<E> ........................................... 145
string ........................................... 22
vector ........................................... 102
levenshtein()

  distance ........................................... 71
lextract ........................................... 764
line ........................................... 444
Linear Orders . see dictionary, see sortseq,

  see User defined parameter types
linear_base(...) ........................................... 119
linear_rank(...) ........................................... 119
linear_solver(...) ........................................... 112, 113
linearly_independent(...) ................................ 119
link(...)

  dynamic_trees .................................... 159
tree_collection<I> ...................................... 161
list<E> ........................................... 137
listen() ........................................... 39
Lman ........................................... 761
load_layout(...) ........................................... 679
GraphWin ........................................... 679
local_aff_sw() ........................................... 73
local_sw() ........................................... 73
locate(...) ........................................... 180
locate_point(...) ........................................... 580
locate_pred(...) ........................................... 580
locate_subdivision<I> ..................................... 580
locate_subdivision<I> ..................................... 580
locate_subdivision<I> ..................................... 580
pp_dictionary<K, I> ..................................... 175
POINT_SET ........................................... 570
sortseq<K, I> ........................................... 180
pp_dictionary<K, I, CMP> ................................ 178
sortseq<K, I> ........................................... 180
log(...) ........................................... 77
log2_abs(...) ........................................... 77
longest_common_suffix() .................................. 73
lookup(...) ........................................... 180

   d2_dictionary<K1, K2, I> ................................ 566
dictionary<K, I> ........................................... 164
interval_set<I> ........................................... 574
POINT_SET ........................................... 570, 571
INDEX

pp_dictionary<K, I, CMP> ............ 178
rat_segment_set<I> ................. 578
segment_set<I> ................. 576
sortseq<K, I> .................. 180
low()                            
array<E> .......................... 128
real ................................ 86
low1()                           
array2<E> .......................... 131
low2()                           
array2<E> .......................... 131
lower_bound()                    
b_priority_queue<I> ............. 191
interval .......................... 93
LOWER_CONVEX_HULL(...) ......... 525
lower_left()                     
rat_rectangle .................... 494
real_rectangle .................... 521
rectangle ........................ 468
lower_right()                    
rat_rectangle .................... 494
real_rectangle .................... 521
rectangle ........................ 468
lstyle_item(...) ............................. 455
window ........................... 656
lwidth_item(...) ............................. 455
window ........................... 656

M
MAC_is_valid()  
OMACCoder < BlkCipher > ............ 259
make(...)                        
dynamic_trees ..................... 159
Make_Acyclic(...) ................ 330
Make_Biconnected(...) ........... 330
make_bidirected()                
graph ............................. 276
Make_Bidirected(...) ............. 330
make_bidirected(...)             
graph ............................. 276
make_block()                     
partition ........................ 155
make_block(...)                  
Partitions<E> ...................... 157
Make_Connected(...) ............. 330
make_directed()                  
graph ............................. 275
make_invalid()                   
AdjIt ................................ 405
EdgeIt ................................ 395
FaceCirc. ................................ 407
FaceIt ................................ 397
InAdjIt ................................ 402
NodeIt ................................ 393
OutAdjIt ................................ 400
make_map()                      
graph ................................ 276
make_map(...)                    
graph ................................ 276
make_menu_bar()                  
window ................................ 661
make_planar_map()                
graph ................................ 278
make_rep(...)                    
node_partition ..................... 318
Make_Simple(...) ................ 330
MAKE_TRANSITIVELY(...). .......... 344
make_undirected()                
graph ................................ 275
make_weakly_simple()             
r_circle_gen_polygon ............. 560
r_circle_polygon .................. 553
make_weakly_simple(...)          
GEN_POLYGON ....................... 462
POLYGON .......................... 455
r_circle_gen_polygon ............. 560
maketree(...)                    
tree_collection<I> .............. 161
Manual Page ........................ 6
Manual Production ................. 761
map2<I, I2, E> ..................... 173
map<I, E> ......................... 171
markov_chain ...................... 332
matrix ................................ 105
max()                           
d_int_set .......................... 152
int_set ................................ 150
list<E> ................................ 142
max(...) ............................ 45
list<E> ................................ 142
MAX_CARD_BIPARTITE(...). ...... 355
MAX_CARD_MATCHING(...) ........ 361
MAX_FLOW_BASIC_T(...) ........ 351
MAX_FLOW_GAP_T(...) ........... 352
INDEX

graph_morphism_algorithm< graph_t > 379
morphism_list

graph_morphism_algorithm< graph_t > 379

moris_pratt()

string_matching .............................. 67

move()

d3_window .................................. 731

move_edge(...)

graph ....................................... 273, 274

move_to_back(...)

list<E> ..................................... 139

move_to_front(...)

list<E> ..................................... 139

move_to_rear(...)

list<E> ..................................... 139

mr_method

PPMIIICoder ................................ 201

mr_restart

PPMIIICoder ................................ 201

msg_clear()

GeoWin ..................................... 726

msg_close()

GeoWin ..................................... 726

msg_open(...) ............................. 726

GeoWin ..................................... 726

MTF2Coder .................................. 220

MTFCoder .................................. 218

mul(...) ................................. residual .............................. 96, 99

MULMULEY_SEGMENTS(...) .............. 532

mw_matching ............................... 361

MWA_SCALE_WEIGHTS(...), 360

mwb_matching ............................. 356

MWBM_SCALE_WEIGHTS(...), 360

MWMCB_MATCHING_T(...), 360

my_sortseq(...)

sortseq<K, I> ............................... 184

N

Ngon(...) .................................. 457

nearest_neighbor(...) .......................... 571

POINT_SET .................................. 571

nearest_neighbors(...) .......................... 571

POINT_SET .................................. 571

negate() .................................. 360

negate(...) .................................. 78

residual .................................... 96, 99

Nesting_Tree(...) ........................... 533

new_edge(...) .............................. graph ............................... 271, 272, 279

GRAPH<vtetype, e...> ..................... 284

GraphWin .................................. 672

planar_map ................................ 294

PLANAR_MAP<vtetype, e...> ............. 297

static_graph ................................ 289

new_map_edge(...) ....................... graph ............................... 271

new_node() .................................. 277

static_graph ................................ 289

new_node(...) .............................. graph ............................... 271

GRAPH<vtetype, e...> ..................... 283

GraphWin .................................. 672

planar_map ................................ 294, 295

PLANAR_MAP<vtetype, e...> ............. 297

new_scene(...) ............................. GeoWin ............................ 698, 700–703

new_scenegroup(...) ..................... GeoWin ............................ 724

next() .................................. GIT_BFS<OutAdjI...> ............... 421

GIT_DFS<OutAdjI...> ....................... 423

GIT_DIJKSTRA<OutAdjI...> ............. 429

GIT_SCC<Out, In,...> ...................... 427

GIT_TOPOSORT<OutAdjI...> .......... 425

next_face_edge(...) ..................... graph ............................... 277

next_unseen() ............................. GIT_DFS<OutAdjI...> ............... 423

NO_CHECK

r_circle_gen_polygon ............... 556

r_circle_polygon ................. 550

node

graph_morphism_algorithm< graph_t > 379

static_graph ................................ 288

node_compat

graph_morphism_algorithm< graph_t > 379

node_data()
\begin{verbatim}
 GRAPH\langle\textit{vtype}, e\rangle \quad \ldots \ldots \quad 283
 node_morphism
 graph\_morphism\_algorithm< graph_t > \quad 379
 node_value_type
 GRAPH\langle\textit{vtype}, e\rangle \quad \ldots \ldots \quad 283
 node_array\langle E\rangle \quad \ldots \ldots \quad 298
 node_array\_da\langle T\rangle \quad \ldots \ldots \quad 415
 node_attribute\_da\langle T\rangle \quad \ldots \ldots \quad 418
 node_list \quad \ldots \ldots \quad 316
 node_map2\langle E\rangle \quad \ldots \ldots \quad 312
 node_map\langle E\rangle \quad \ldots \ldots \quad 304
 node_matrix\langle E\rangle \quad \ldots \ldots \quad 310
 node_member\_da\langle \textit{Str}, T\rangle \quad \ldots \ldots \quad 417
 node\_partition \quad \ldots \ldots \quad 318
 node\_pq\langle P\rangle \quad \ldots \ldots \quad 319
 node\_set \quad \ldots \ldots \quad 314
 NodeIt \quad \ldots \ldots \quad 393
 norm() 
 \quad \textit{real\_vector} \quad \ldots \ldots \quad 121
 \quad \textit{TRANSFORM} \quad \ldots \ldots \quad 536
 \quad \textit{vector} \quad \ldots \ldots \quad 102
 normal()
 \quad \textit{d3\_plane} \quad \ldots \ldots \quad 593
 \quad \textit{d3\_rat\_plane} \quad \ldots \ldots \quad 617
 normal\_project(...) 
 \quad \textit{d3\_plane} \quad \ldots \ldots \quad 594
 \quad \textit{d3\_rat\_plane} \quad \ldots \ldots \quad 618
 normalize()
 \textit{GEN\_POLYGON} \quad 460
 \textit{POLYGON} \quad 453
 \textit{r\_circle\_gen\_polygon} \quad 558
 \textit{r\_circle\_point} \quad \ldots \ldots \quad 543
 \textit{r\_circle\_polygon} \quad 552
 \textit{r\_circle\_segment} \quad \ldots \ldots \quad 546
 \textit{rat\_circle} \quad 489
 \textit{rat\_line} \quad \ldots \ldots \quad 485
 \textit{rat\_point} \quad \ldots \ldots \quad 472
 \textit{rat\_ray} \quad \ldots \ldots \quad 482
 \textit{rat\_rectangle} \quad \ldots \ldots \quad 494
 \textit{rat\_segment} \quad \ldots \ldots \quad 477
 \textit{rat\_triangle} \quad \ldots \ldots \quad 491
 \textit{rational} \quad \ldots \ldots \quad 78
 noweb \quad \ldots \ldots \quad 765
 number\_of\_blocks() 
 \quad \textit{node\_partition} \quad \ldots \ldots \quad 318
 \textit{partition} \quad \ldots \ldots \quad 155
 Partition\langle E\rangle \quad \ldots \ldots \quad 157
 number\_of\_edges() 
 \quad \textit{graph} \quad \ldots \ldots \quad 269
 number\_of\_faces() 
 \quad \textit{graph} \quad \ldots \ldots \quad 277
 number\_of\_nodes() 
 \quad \textit{graph} \quad \ldots \ldots \quad 269
 number\_of\_steps() 
 \quad \textit{dynamic\_markov\_chain} \quad 333
 \quad \textit{markov\_chain} \quad 332
 numerator() 
 \quad \textit{rational} \quad \ldots \ldots \quad 78
 numerical\_analysis \quad \ldots \ldots \quad 125
\end{verbatim}
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>list&lt;E&gt;</td>
<td>140</td>
</tr>
<tr>
<td>permute_edges()</td>
<td>275</td>
</tr>
<tr>
<td>perpendicular(...)</td>
<td></td>
</tr>
<tr>
<td>line</td>
<td>446</td>
</tr>
<tr>
<td>rat_line</td>
<td>486</td>
</tr>
<tr>
<td>rat_segment</td>
<td>480</td>
</tr>
<tr>
<td>real_line</td>
<td>512</td>
</tr>
<tr>
<td>real_segment</td>
<td>505</td>
</tr>
<tr>
<td>segment</td>
<td>439</td>
</tr>
<tr>
<td>place_into_box(...)</td>
<td></td>
</tr>
<tr>
<td>GraphWin</td>
<td>678</td>
</tr>
<tr>
<td>place_into_win()</td>
<td></td>
</tr>
<tr>
<td>GraphWin</td>
<td>678</td>
</tr>
<tr>
<td>PLANAR(...)</td>
<td>372</td>
</tr>
<tr>
<td>planar_map</td>
<td>294</td>
</tr>
<tr>
<td>PLANAR_MAP&lt;vtype, e&gt;</td>
<td>296</td>
</tr>
<tr>
<td>plane_graph_alg</td>
<td>372</td>
</tr>
<tr>
<td>plot_xy(...)</td>
<td>643</td>
</tr>
<tr>
<td>window</td>
<td></td>
</tr>
<tr>
<td>plot_yx(...)</td>
<td>643</td>
</tr>
<tr>
<td>window</td>
<td></td>
</tr>
<tr>
<td>point</td>
<td>432</td>
</tr>
<tr>
<td>point generators</td>
<td>539</td>
</tr>
<tr>
<td>point1()</td>
<td></td>
</tr>
<tr>
<td>circle</td>
<td>449</td>
</tr>
<tr>
<td>d3_line</td>
<td>591</td>
</tr>
<tr>
<td>d3_plane</td>
<td>593</td>
</tr>
<tr>
<td>d3_rat_line</td>
<td>611</td>
</tr>
<tr>
<td>d3_rat_plane</td>
<td>617</td>
</tr>
<tr>
<td>d3_rat_simplex</td>
<td>622</td>
</tr>
<tr>
<td>d3_rat_sphere</td>
<td>620</td>
</tr>
<tr>
<td>d3_simplex</td>
<td>598</td>
</tr>
<tr>
<td>d3_sphere</td>
<td>596</td>
</tr>
<tr>
<td>line</td>
<td>444</td>
</tr>
<tr>
<td>rat_circle</td>
<td>489</td>
</tr>
<tr>
<td>rat_line</td>
<td>485</td>
</tr>
<tr>
<td>rat_ray</td>
<td>482</td>
</tr>
<tr>
<td>rat_triangle</td>
<td>491</td>
</tr>
<tr>
<td>ray</td>
<td>441</td>
</tr>
<tr>
<td>real_circle</td>
<td>515</td>
</tr>
<tr>
<td>real_line</td>
<td>511</td>
</tr>
<tr>
<td>real_ray</td>
<td>507</td>
</tr>
<tr>
<td>real_triangle</td>
<td>518</td>
</tr>
<tr>
<td>triangle</td>
<td>465</td>
</tr>
<tr>
<td>point2()</td>
<td></td>
</tr>
<tr>
<td>circle</td>
<td>449</td>
</tr>
<tr>
<td>d3_line</td>
<td>591</td>
</tr>
<tr>
<td>d3_plane</td>
<td>593</td>
</tr>
<tr>
<td>d3_rat_line</td>
<td>611</td>
</tr>
<tr>
<td>d3_rat_plane</td>
<td>617</td>
</tr>
<tr>
<td>d3_rat_ray</td>
<td>609</td>
</tr>
<tr>
<td>rat_triangle</td>
<td></td>
</tr>
<tr>
<td>ray</td>
<td></td>
</tr>
<tr>
<td>real_circle</td>
<td></td>
</tr>
<tr>
<td>real_line</td>
<td></td>
</tr>
<tr>
<td>real_ray</td>
<td></td>
</tr>
<tr>
<td>real_triangle</td>
<td></td>
</tr>
<tr>
<td>triangle</td>
<td></td>
</tr>
</tbody>
</table>
INDEX

```
d3_rat_simplex .................................. 622  list<E> ........................................ 139
d3_simplex ....................................... 598  node_list ....................................... 316
GEN_POLYGON ..................................... 458  pop_back(...) .................................. 315
line ................................................ 444  b_queue<E> ....................................... 135
point .............................................. 432  pop_front() ..................................... 139
POLYGON .......................................... 452  list<E> ............................................ 316
r_circle_gen_polygon ......................... 556  firsteway<E> .................................... 317
r_circle_polygon ................................. 550  pos(...) .......................................... 320
rat_circle ....................................... 488  POINT_SET ...................................... 569
rat_line .......................................... 484  string ............................................ 22
rat_point ........................................ 471  pos_source(...) ................................. 23
rat_ray ........................................... 481  POINT_SET ...................................... 569
rat_segment ...................................... 476  pos_target(...) ................................. 24
rat_triangle ..................................... 491  print(...) ........................................ 25
ray ............................................... 441  pred(...) ......................................... 26
real_circle ....................................... 514  subdivision<I> .................................. 580
real_line ......................................... 510  possible_zero() .................................. 38
real_point ........................................ 498  real ................................................ 39
real_ray .......................................... 507  pow(...) .......................................... 40
real_segment ..................................... 503  powi(...) ........................................ 41
real_triangle .................................... 518  pp_dictionary<K, I, CMP> .................... 177
segment .......................................... 437  PPMHCoder ....................................... 201
triangle ......................................... 465  pred(...) ......................................... 21
POINT_SET ...................................... 568  list<E> ............................................ 316
points() ......................................... 569  node_list ....................................... 316
POINT_SET ...................................... 569  sortseq<K, I> ................................. 181, 182
points_on_segment(...) ....................... 542  pred_edge(...) .................................. 270
POLYGON .......................................... 452  pred_face(...) .................................. 278
polygon_type ..................................... 458  pred_face_edge(...) ......................... 277
GEN_POLYGON ................................... 458  pred_node(...) .................................. 270
r_circle_gen_polygon ......................... 556  graph .............................................. 380
polygons() ....................................... 461  graph .............................................. 380
GEN_POLYGON ................................... 461  graph_morphism_algorithm< graph_t > 380
r_circle_gen_polygon ......................... 559  prepare_graph(...) ............................ 380
Polynomial ....................................... 85
real ............................................... 85
Pop() .............................................. 85
list<E> ............................................ 139  graph_morphism_algorithm< graph_t > 380
pop() ............................................. 139  print(...) ........................................ 380
b_queue<E> ....................................... 136  prepare_graph(...) ............................ 380
b_stack<E> ....................................... 134  print(...) ........................................ 380
list<E> ............................................ 139  matrix ............................................ 106
node_list ....................................... 316  real_matrix .................................... 124
queue<E> ........................................ 133  real_vector .................................... 121
sl<E> ............................................. 146  vector ............................................ 103
stack<E> ........................................ 132  print(...) ........................................ 130
pop_back() ...................................... 132
```

INDEX

graph ............................................ 281
d3_rat_point .................................. 601
d3_rat_segment .................................. 615
d3_segment ...................................... 590
d3_line ......................................... 591
d3_rat_line ..................................... 611
d3_rat_ray ....................................... 610
d3_ray ........................................... 588
print_edge(...)................................. 281
PROJECT_XZ(....)............................... 582
print_face(...)................................. 277
d3_point ......................................... 601
d3_rat_point .................................... 615
d3_rat_segment .................................. 615
d3_segment ...................................... 590
print_node(...)................................. 280
PROJECT_XZ(....)............................... 582
print_separation(...)......................... 86
print_statistics()............................. 44
d3_line ......................................... 591
d3_rat_line ..................................... 612
d3_rat_segment .................................. 615
d3_segment ...................................... 590
prio(...)........................................ 190
b_priority_queue<1> ............................. 136
node_pg<P> ............................. 319
ps_file .................................. 667
p_queue<P,1> ......................... 188
project(...).................................... 667
pstyle_item(...)............................... 667
prio_type....................................... 187
push(...) ...................................... 655
Priority Queues ................................ 733
Emde-Boas tree ................................ 733
Fibonacci heap .................................. 733
k-nary heap ..................................... 733
monotonic heap .................................. 733
pairing heap .................................... 733
push_back(...)................................. 132
push_front(...)................................. 132
PROJECT_XY()................................. 582
PROJECT_XY(...)............................... 582
PROJECT_XZ()................................. 582
Q
queue<E> .................................. 133
quick_search() .......................... 67
R
r_circle_gen_polygon ................. 556
<table>
<thead>
<tr>
<th>Function/Concept</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_\text{circle}_\text{point} )</td>
<td>543</td>
</tr>
<tr>
<td>( r_\text{circle}_\text{polygon} )</td>
<td>550</td>
</tr>
<tr>
<td>( r_\text{circle}_\text{segment} )</td>
<td>545</td>
</tr>
<tr>
<td>\text{radical_axis}(...)</td>
<td>451, 490, 517</td>
</tr>
<tr>
<td>( \text{radius}() )</td>
<td>449</td>
</tr>
<tr>
<td>( d3_\text{sphere} )</td>
<td>596</td>
</tr>
<tr>
<td>( \text{real}_\text{circle} )</td>
<td>515</td>
</tr>
<tr>
<td>\text{raita()}</td>
<td>67</td>
</tr>
<tr>
<td>\text{random}(...)</td>
<td>77</td>
</tr>
<tr>
<td>\text{random}_\text{bigraph}(...)</td>
<td>324</td>
</tr>
<tr>
<td>\text{random}<em>d3_\text{rat}</em>\text{point}(...)</td>
<td>605–608</td>
</tr>
<tr>
<td>\text{random}_\text{graph}(...)</td>
<td>323</td>
</tr>
<tr>
<td>\text{random}<em>\text{graph}</em>\text{nonc}(...)</td>
<td>323</td>
</tr>
<tr>
<td>\text{random}<em>\text{planar}</em>\text{graph}(...)</td>
<td>325–327</td>
</tr>
<tr>
<td>\text{random}<em>\text{planar}</em>\text{map}(...)</td>
<td>325, 326</td>
</tr>
<tr>
<td>\text{random}<em>\text{point}</em>\text{in}_\text{ball}(...)</td>
<td>540</td>
</tr>
<tr>
<td>\text{random}<em>\text{point}</em>\text{in}_\text{cube}(...)</td>
<td>539</td>
</tr>
<tr>
<td>\text{random}<em>\text{point}</em>\text{in}_\text{disc}(...)</td>
<td>540</td>
</tr>
<tr>
<td>\text{random}<em>\text{point}</em>\text{in}_\text{s}(...)</td>
<td>539</td>
</tr>
<tr>
<td>\text{random}<em>\text{point}</em>\text{in}_\text{u}(...)</td>
<td>539, 540</td>
</tr>
<tr>
<td>\text{random}<em>\text{point}</em>\text{near}(...)</td>
<td>541</td>
</tr>
<tr>
<td>\text{random}<em>\text{point}</em>\text{on}_c(...)</td>
<td>541</td>
</tr>
<tr>
<td>\text{random}<em>\text{point}</em>\text{on}_p(...)</td>
<td>542</td>
</tr>
<tr>
<td>\text{random}<em>\text{point}</em>\text{on}_s(...)</td>
<td>542</td>
</tr>
<tr>
<td>\text{random}<em>\text{point}</em>\text{on}_u(...)</td>
<td>542</td>
</tr>
<tr>
<td>\text{random}<em>\text{points}</em>\text{in}(...)</td>
<td>539, 540</td>
</tr>
<tr>
<td>\text{random}<em>\text{points}</em>\text{nea}(...)</td>
<td>541</td>
</tr>
<tr>
<td>\text{random}<em>\text{points}</em>\text{on}(...)</td>
<td>541, 542</td>
</tr>
<tr>
<td>\text{random}<em>\text{simple}</em>\text{graph}(...)</td>
<td>323</td>
</tr>
<tr>
<td>\text{random}<em>\text{simple}</em>\text{loo}(...)</td>
<td>323</td>
</tr>
<tr>
<td>\text{random}<em>\text{simple}</em>\text{und}(...)</td>
<td>323</td>
</tr>
<tr>
<td>\text{random}<em>\text{sp}</em>\text{graph}(...)</td>
<td>327</td>
</tr>
<tr>
<td>\text{random}_\text{source}</td>
<td>27</td>
</tr>
<tr>
<td>\text{random}_\text{variate}</td>
<td>29</td>
</tr>
<tr>
<td>\text{range}_\text{search}(...)</td>
<td>566</td>
</tr>
<tr>
<td>( d2_\text{dictionary}&lt;K1, K2, I&gt; )</td>
<td>566</td>
</tr>
<tr>
<td>( \text{POINT_SET} )</td>
<td>571, 572</td>
</tr>
<tr>
<td>\text{range}<em>\text{search}</em>\text{para}(...)</td>
<td>572</td>
</tr>
<tr>
<td>( \text{POINT_SET} )</td>
<td>572</td>
</tr>
<tr>
<td>\text{rank}(...)</td>
<td>113</td>
</tr>
<tr>
<td>( \text{list}&lt;E&gt; )</td>
<td>138</td>
</tr>
<tr>
<td>( \text{rat}_\text{circle} )</td>
<td>488</td>
</tr>
<tr>
<td>( \text{rat}_\text{line} )</td>
<td>484</td>
</tr>
<tr>
<td>( \text{rat}_\text{point} )</td>
<td>471</td>
</tr>
</tbody>
</table>
region_of(...)
  GEN_POLYGON .................................... 462
  POLYGON .......................................... 456
  r_circle_gen_polygon .............................. 561
  r_circle_polygon ................................ 554
  rat_rectangle ...................................... 495
  rat_triangle ....................................... 492
  real_rectangle ..................................... 522
  real_triangle ...................................... 519
  rectangle .......................................... 469
  triangle ............................................ 466
region_of_sphere(...) .............................. 585, 604
region_decompos(...) .............................. 463
register_window(...) ............................... 704
reset_seed()
  random_source ..................................... 27
relative_freq_of_visit(...)
  dynamic_markov_chain .............................. 333
  markov_chain ...................................... 332
relative_neighbor(...) ............................. 572
remove(...)  ........................................ 522
  list<E> ............................................ 139
remove_bends()  ..................................... 679
reset(...)  ........................................... 543
INDEX

RLECoder ........................................ 214
graph .............................................. 273
reset_actions().................................... 719
GeoWin ............................................. 680
reset_acyclic()..................................... 425
GIT_TOPOSORT<OutAdjIt> ..................... 685
reset_clipping()................................... 647
reset_defaults()................................. 272
GraphWin .......................................... 272
reset_edge_anchors()............................. 272
GraphWin .......................................... 272
reset_edges(...) .................................. 272
GraphWin .......................................... 272
reset_end() ........................................ 272
AdjIt ............................................... 405
EdgeIt ............................................. 395
FaceIt ............................................. 397
InAdjIt ............................................ 402
node ............................................... 393mrat ............................................... 400
reset_frame_label()............................... 393
window ............................................. 647
reset_nodes(...) .................................. 273
GraphWin .......................................... 273
reset_num_calls()................................. 273
GraphWin .......................................... 273
reset_obj_attributes(...)....................... 273
GeoWin ............................................. 273
gml_graph ......................................... 277
reset_window() ..................................... 277
GeoWin ............................................. 277
reset_path() ....................................... 277
rd3_graph ......................................... 277
GeoWin ............................................. 277
reset_path() ....................................... 277
rd3_rat_line ...................................... 277
reset_frame_label()............................... 277
d3_line ........................................... 277
d3_rat_line ....................................... 277
d3_rat_line ...................................... 277
d3_rat_segment .................................. 277
d3_segment ........................................ 277
reverse() ......................................... 277
reverse(...)....................................... 277
rd3_rat_segment .................................. 277
rd3_segment ...................................... 277
rd3_line .......................................... 277
rd3_rat_segment .................................. 277
rd3_segment ...................................... 277
d3_rat_segment .................................. 277
rd3_segment ...................................... 277
circle ............................................. 446
d3_line ........................................... 592
d3_rat_line ....................................... 612
d3_rat_segment .................................. 610
d3_rat_segment .................................. 615
d3_rat_segment .................................. 588
d3_segment ........................................ 590
line ............................................... 592
list<E> ............................................ 140
rd3_rat_segment .................................. 547
real_circle ........................................ 490
rd3_circle ......................................... 486
rd3_rat_line ...................................... 482
rd3_rat_line ...................................... 480
rd3_rat_segment .................................. 493
d3_rat_triangle .................................. 493
d3_rat_segment .................................. 443
rd3_rat_triangle .................................. 443
real_circle ........................................ 516
rd3_rat_triangle .................................. 512
real_line ......................................... 516
rd3_rat_segment .................................. 508
real_segment ...................................... 506
real_segment ...................................... 520
real_triangle ..................................... 467
segment .......................................... 440
triangle ......................................... 440
real_line ......................................... 508
real_segment ...................................... 506
real_triangle ..................................... 520
real_triangle ..................................... 467
reverse(...)....................................... 140
rd3_scale ......................................... 140
realCircle ........................................ 140
triangle ......................................... 140
realsegment ...................................... 140
realtriangle ..................................... 140
r_segment ......................................... 140
real_graph ........................................ 140
reverse ........................................... 140
real_graph ........................................ 140
reverse ........................................... 140
real_graph ........................................ 140
real_graph ........................................ 140
<table>
<thead>
<tr>
<th>string_matching</th>
<th>67</th>
<th>real_circle</th>
<th>516</th>
</tr>
</thead>
<tbody>
<tr>
<td>reverse_factor()</td>
<td></td>
<td>real_line</td>
<td>512</td>
</tr>
<tr>
<td>string_matching</td>
<td>67</td>
<td>real_point</td>
<td>499</td>
</tr>
<tr>
<td>reverse_items()</td>
<td></td>
<td>real_ray</td>
<td>508</td>
</tr>
<tr>
<td>list&lt;E&gt;</td>
<td>140</td>
<td>real_rectangle</td>
<td>523</td>
</tr>
<tr>
<td>reverse_items(...)</td>
<td></td>
<td>real_segment</td>
<td>505</td>
</tr>
<tr>
<td>list&lt;E&gt;</td>
<td>140</td>
<td>real_triangle</td>
<td>520</td>
</tr>
<tr>
<td>sortseq&lt;K, I&gt;</td>
<td>182</td>
<td>real_vector</td>
<td>121</td>
</tr>
<tr>
<td>right(...)</td>
<td></td>
<td>rectangle</td>
<td>470</td>
</tr>
<tr>
<td>interval_set&lt;I&gt;</td>
<td>574</td>
<td>segment</td>
<td>440</td>
</tr>
<tr>
<td>right_tangent(...)</td>
<td></td>
<td>triangle</td>
<td>467</td>
</tr>
<tr>
<td>circle</td>
<td>450</td>
<td>vector</td>
<td>103</td>
</tr>
<tr>
<td>real_circle</td>
<td>516</td>
<td>rotation(...)</td>
<td>537</td>
</tr>
<tr>
<td>right_turn(...)</td>
<td>435, 474, 501</td>
<td>rotation90(...)</td>
<td>537</td>
</tr>
<tr>
<td>RLE0Coder</td>
<td>222</td>
<td>round(...)</td>
<td>79</td>
</tr>
<tr>
<td>RLECoder</td>
<td>213</td>
<td></td>
<td></td>
</tr>
<tr>
<td>root(...)</td>
<td>88</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dynamic_trees</td>
<td>159</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rotate(...)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>circle</td>
<td>450</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GEN_POLYGON</td>
<td>464</td>
<td></td>
<td></td>
</tr>
<tr>
<td>line</td>
<td>446</td>
<td></td>
<td></td>
</tr>
<tr>
<td>point</td>
<td>433</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POLYGON</td>
<td>456</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ray</td>
<td>442</td>
<td></td>
<td></td>
</tr>
<tr>
<td>segment</td>
<td>439, 440</td>
<td></td>
<td></td>
</tr>
<tr>
<td>triangle</td>
<td>466, 467</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vector</td>
<td>103</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rotate90(...)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>circle</td>
<td>450</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GEN_POLYGON</td>
<td>461</td>
<td></td>
<td></td>
</tr>
<tr>
<td>line</td>
<td>446</td>
<td></td>
<td></td>
</tr>
<tr>
<td>point</td>
<td>433</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POLYGON</td>
<td>454</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r_circle_gen_polygon</td>
<td>559</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r_circle_point</td>
<td>544</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r_circle_polygon</td>
<td>553</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r_circle_segment</td>
<td>548</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rat_circle</td>
<td>490</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rat_line</td>
<td>486</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rat_point</td>
<td>472</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rat_ray</td>
<td>482</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rat_rectangle</td>
<td>496</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rat_segment</td>
<td>479, 480</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rat_triangle</td>
<td>493</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rat_vector</td>
<td>117</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ray</td>
<td>442</td>
<td></td>
<td></td>
</tr>
<tr>
<td>save_all_attributes()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>save_defaults(...)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>save_gml(...)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>save_edge_attributes()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>save_indicator(...)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>same_block(...)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>node_partition</td>
<td>318</td>
<td></td>
<td></td>
</tr>
<tr>
<td>partition</td>
<td>155</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Partition&lt;E&gt;</td>
<td>157</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GraphWin</td>
<td>685</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GraphWin</td>
<td>683</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GraphWin</td>
<td>685</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GraphWin</td>
<td>683</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GraphWin</td>
<td>683</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GraphWin</td>
<td>679</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GraphWin</td>
<td>685</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
save_ps(...)           GraphWin                       683
save_svg(...)          GraphWin                       683
save_wmf(...)          GraphWin                       683
scale()                window                        636
score_matrix           68
screenshot(...)        window                        647
search(...)            list<E>                        142
second()               four_tuple<A, B, C, D>                   54
                       three_tuple<A, B, C>                      52
                       two_tuple<A, B>                          51
second_type            four_tuple<A, B, C, D>                   53
                       three_tuple<A, B, C>                      52
                       two_tuple<A, B>                          51
secure_socket_streambuf 263
seekg(...)             decoding_ofstream<Coder>                231
seekp(...)             encoding_ofstream<Coder>                230
seg()                  d3_line                         591
                       d3_rat_line                     611
                       d3_rat_ray                      609
                       d3_ray                          587
                       line                           444
                       rat_line                        485
                       real_line                       511
seg(...)               POINT_SET                      569
segment               437
SEGMENT_INTERSECTION(...) 531, 532
segment_type          GEN_POLYGON                    458
POLYGON                452
r_circle_gen_polygon   556
r_circle_polygon       550
d3_window              732
set(...                array<E>                        128
                       set<E>                         147
                       set_action(...)              147
                       set_activate_handler(...)    724
                       set_active_line_w...(...)    709
                       set_all_visible(...)         710
                       set_animation_steps(...)     675
                       set_arrow(...)               732
                       set_bg_color(...)            717
                       set(...)                      415, 418, 419
                       set(...                      709
set(...
POLYGON                454
r_circle_polygon       552
window                633
INDEX

set_edge_distance(...) .......................... 674
set_edge_index(...) ............................ 675
set_edge_label(...) ............................ 674
set_edge_param(...) ............................ 673
set_edge_position(...) .......................... 275
set_edge_slider(...) .............................. 681
set_edit_loop(...) ............................... 724
set_edit_model(...) .............................. 724
set_edit_object(...) .............................. 722
set_elim(...) .......................... 731
set_enc_buffer(...) .............................. 206
set_end_change(...) .............................. 722
set_end_move(...) ................................. 681
set_error_handler(...) ............................ 34
set_fast_compress(...) ........................... 34
set_flush(...) .......................... 675
set_frame(...) .......................... 635
set_function(...) ............................... 662
set_gen_edges(...) ............................... 674
set_gen_nodes(...) ............................... 674

GraphWin ..................................... 673
GeoWin ....................................... 722
GeoWin ....................................... 717
GeoWin ....................................... 717
GeoWin ....................................... 717

set_generate_fcn(...) ............................ 722
set_graph(...) ................................. 684
set_green(...) ................................. 628
set_grid_dist(...) ............................... 717
set_grid_mode(...) ............................... 632
set_grid_size(...) ............................... 632
set_grid_style(...) .............................. 632
set_handle(...) ................................. 711
set_high_compress(...) ........................... 711
set_icon_label(...) .............................. 634
set_icon_pixrect(...) ............................ 635
set_incremental(...) ............................. 711
set_initialization(...) .......................... 711
set_input_format(...) ............................ 58
set_input_object(...) ............................. 58
set_input_precision(...) .......................... 81
set_item_height(...) ............................... 654
set_item_width(...) ............................... 654
set_key(...) .......................... 256
set_key(...) .......................... 256
INDEX

CryptAutoDecoder .............................. 261
OMACCoder < BlkCipher > .................. 259
set_keys_in_file(...)
  CryptAutoDecoder .......................... 262
set_keys_in_stream(...)
  CryptAutoDecoder .......................... 261
set_label(...)
  GeoWin .................................... 725
set_language(...)
  date ....................................... 58
set_layout()
  GraphWin .................................. 677
set_layout(...)
  GraphWin .................................. 676, 677
set_MAC_in_stream(...)
  OMACCoder < BlkCipher > ................ 259
set_maximal_bit(...)
  residual .................................... 97
set_memory_level(...)
  DeflateCoder ............................... 206
set_menu(...)
  window .................................... 657
set_menu_add_fcn(...)
  GeoWin ..................................... 706
set_message(...)
  d3_window .................................. 732
set_midpoint(...)
  interval .................................. 94
set_model(...)
  window .................................... 634
set_model_memory(...)
  PPMIIICoder ............................... 203
set_model_order(...)
  PPMIIICoder ............................... 203
set_model_restoration(...)
  PPMIIICoder ............................... 203
set_month(...)
  date ....................................... 60
set_month_names(...)
  GeoWin ..................................... 728
set_move_node_handler(...)
  GraphWin .................................. 680
set_name(...)
  counter ................................... 49
set_new_edge_handler(...)
  GraphWin .................................. 680
set_new_node_handler(...)
  GraphWin .................................. 680
set_node_color(...)
  ledasocket .................................. 38
set_node_index_font(...)
  GraphWin .................................. 674
set_node_label_font(...)
  GraphWin .................................. 674
set_node_position(...)
  graph ...................................... 275
set_node_width(...)
  window .................................... 634
set_obj_color(...)
  GeoWin ..................................... 713
set_obj_fill_color(...)
  GeoWin ..................................... 713
set_obj_label(...)
  GeoWin ..................................... 715
set_obj_line_style(...)
  GeoWin ..................................... 714
set_obj_line_width(...)
  GeoWin ..................................... 714
set_obj_text(...)
  GeoWin ..................................... 716
set_object(...)
  window .................................... 662
set_output_format(...)
  date ....................................... 59
set_output_mode(...)
  bigfloat ................................... 82
set_output_precision(...)
  bigfloat ................................... 81
set_panel_bg_color(...)
  GeoWin ..................................... 708
  timer ...................................... 47
INDEX

set_param(...)  ........  654
set_point(...)  ... GeoWin  ........  727
set_point_style(...)  GeoWin  ........  710
set_port(...)  ...
  leda_socket  ........  38
set_position(...)  ...
  d3_window  ........  731
set_post_add_handler(...)  GeoWin  ........  720
set_post_del_handler(...)  GeoWin  ........  721
set_post_move_handler(...)  GeoWin  ........  721
set_post_rotate_handler(...)  GeoWin  ........  722
set_postscript_us(...)  GeoWin  ........  707
set_pre_add_handler(...)  GeoWin  ........  720
set_pre_del_handler(...)  GeoWin  ........  720
set_pre_move_handler(...)  GeoWin  ........  721
set_pre_rotate_handler(...)  GeoWin  ........  721
set_precision(...)  bigfloat  ........  81
  random_source  ........  27
  window  ........  632
set_qlength(...)  leda_socket  ........  38
set_quit_handler(...)  GeoWin  ........  724
set_range(...)  ...
  interval  ........  93
  random_source  ........  27
set_receive_handler(...)  leda_socket  ........  39
set_red(...)  color  ........  628
set_redraw(...)  window  ........  635
set_redraw2(...)  window  ........  635
set_reset_threshold(...)  A0Coder  ........  198
set_reversal(...)  graph  ........  276
set_rgb(...)  color  ........  628
set_rounding_mode(...)  bigfloat  ........  82
set_selected_objects(...)  GeoWin  ........  704
set_selection_color(...)  GeoWin  ........  708
set_selection_fill(...)  GeoWin  ........  708
set_selection_line(...)  GeoWin  ........  708, 709
set_send_handler(...)  leda_socket  ........  39
set_show_algorithm(...)  GeoWin  ........  706
set_show_coord(...)  window  ........  635
set_show_coord_line(...)  window  ........  635
set_show_coordinates(...)  window  ........  634
set_show_edit_menu(...)  GeoWin  ........  706
set_show_file_menu(...)  GeoWin  ........  706
set_show_grid(...)  GeoWin  ........  717
set_show_help_menu(...)  GeoWin  ........  706
set_show_menu(...)  GeoWin  ........  706
set_show_options(...)  GeoWin  ........  706
set_show_orientation(...)  GeoWin  ........  706
INDEX

GeoWin ........................................ 710
window ........................................ 634
set_show_position(...) GeoWin ......................... 718
set_show_scenes_menu(...) GeoWin ....................... 706
set_show_status(...) GeoWin ......................... 706
GraphWin ....................................... 674
set_show_window_menu(...) GeoWin ....................... 706
set_size(...) memory_streambuf .................... 241
set_solid(...) d3_window .......................... 731
set_src_file(...) A0Coder ......................... 197
A0sCoder ...................................... 200
AdaptiveHuffmanCoder ......................... 212
AutoDecoder ................................... 236
BlockCoder<Coder> ................................ 238
BWTCoder ...................................... 216
CBCCoder<BlkCipher> .......................... 255
checksum_base ................................. 225
CoderPipe2<Coder1,...> ....................... 233
CryptAutoDecoder ...................... 261
DeflateCoder ......................... 205
DictCoder ....................................... 208
HuffmanCoder ................................. 210
MTFCoder ....................................... 219
MTF2Coder ...................................... 221
OMACCoder<BlkCipher> ....................... 258
PPMIICoder ..................................... 202
RLE0Coder ...................................... 223
RLECoder ........................................ 214
set_src_stream(...) A0Coder ......................... 197
A0sCoder ...................................... 200
AdaptiveHuffmanCoder ......................... 212
AutoDecoder ................................... 236
BlockCoder<Coder> ................................ 238
BWTCoder ...................................... 216
CBCCoder<BlkCipher> ......................... 255
checksum_base ................................. 225
CoderPipe2<Coder1,...> ....................... 233
CryptAutoDecoder ...................... 261
DeflateCoder ......................... 205
DictCoder ....................................... 208
HuffmanCoder ................................. 210
MTFCoder ....................................... 219
MTF2Coder ...................................... 221
OMACCoder<BlkCipher> ....................... 258
PPMIICoder ..................................... 202
RLE0Coder ...................................... 223
RLECoder ........................................ 214
set_tgt_file(...) A0Coder ......................... 197
A0sCoder ...................................... 200
AdaptiveHuffmanCoder ......................... 212
AutoDecoder ................................... 236
BlockCoder<Coder> ................................ 238
BWTCoder ...................................... 216
CBCCoder<BlkCipher> ......................... 255
checksum_base ................................. 225
CoderPipe2<Coder1,...> ....................... 233
CryptAutoDecoder ...................... 261
DeflateCoder ......................... 205
DictCoder ....................................... 208
HuffmanCoder ................................. 210
MTFCoder ....................................... 219
MTF2Coder ...................................... 221
OMACCoder<BlkCipher> ....................... 258
PPMIICoder ..................................... 202
RLE0Coder ...................................... 223
RLECoder ........................................ 214
set_tgt_stream(...)
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHACoder</td>
<td>227</td>
</tr>
<tr>
<td>shift_key_down()</td>
<td>651</td>
</tr>
<tr>
<td>window</td>
<td>67</td>
</tr>
<tr>
<td>shift_or()</td>
<td></td>
</tr>
<tr>
<td>string_matching()</td>
<td></td>
</tr>
<tr>
<td>SHORTEST_PATH_T(...)</td>
<td>346</td>
</tr>
<tr>
<td>shortest_path()</td>
<td>345</td>
</tr>
<tr>
<td>side_of(...)</td>
<td></td>
</tr>
<tr>
<td>circle</td>
<td>449</td>
</tr>
<tr>
<td>d3_plane</td>
<td>594</td>
</tr>
<tr>
<td>d3_rat_plane</td>
<td>618</td>
</tr>
<tr>
<td>GEN_POLYGON</td>
<td>462</td>
</tr>
<tr>
<td>line</td>
<td>446</td>
</tr>
<tr>
<td>POLYGON</td>
<td>456</td>
</tr>
<tr>
<td>r_circle_gen_polygon</td>
<td>561</td>
</tr>
<tr>
<td>r_circle_polygon</td>
<td>554</td>
</tr>
<tr>
<td>rat_circle</td>
<td>489</td>
</tr>
<tr>
<td>rat_line</td>
<td>486</td>
</tr>
<tr>
<td>rat_triangle</td>
<td>492</td>
</tr>
<tr>
<td>real_circle</td>
<td>515</td>
</tr>
<tr>
<td>real_line</td>
<td>512</td>
</tr>
<tr>
<td>real_triangle</td>
<td>519</td>
</tr>
<tr>
<td>triangle</td>
<td>466</td>
</tr>
<tr>
<td>side_of_circle(...)</td>
<td>435, 475, 501</td>
</tr>
<tr>
<td>side_of_halfspace(...)</td>
<td>435, 475, 501</td>
</tr>
<tr>
<td>side_of_sphere(...)</td>
<td>584, 604</td>
</tr>
<tr>
<td>sign()</td>
<td>76</td>
</tr>
<tr>
<td>integer</td>
<td>94</td>
</tr>
<tr>
<td>interval</td>
<td>87</td>
</tr>
<tr>
<td>real</td>
<td>98</td>
</tr>
<tr>
<td>residual</td>
<td></td>
</tr>
<tr>
<td>Sign(...)</td>
<td>101</td>
</tr>
<tr>
<td>sign(...)</td>
<td>77, 79, 83</td>
</tr>
<tr>
<td>real</td>
<td>87</td>
</tr>
<tr>
<td>sign_is_known()</td>
<td></td>
</tr>
<tr>
<td>interval</td>
<td>94</td>
</tr>
<tr>
<td>sign_of_determinant(...)</td>
<td>112</td>
</tr>
<tr>
<td>simple_parts()</td>
<td></td>
</tr>
<tr>
<td>POLYGON</td>
<td>455</td>
</tr>
<tr>
<td>simplify()</td>
<td>536</td>
</tr>
<tr>
<td>TRANSFORM</td>
<td></td>
</tr>
<tr>
<td>simplify(...)</td>
<td></td>
</tr>
<tr>
<td>rational</td>
<td>78</td>
</tr>
<tr>
<td>size()</td>
<td></td>
</tr>
<tr>
<td>array&lt;E&gt;</td>
<td>128</td>
</tr>
<tr>
<td>b_priority_queue&lt;E&gt;</td>
<td>191</td>
</tr>
<tr>
<td>b_queue&lt;E&gt;</td>
<td>135</td>
</tr>
<tr>
<td>b_stack&lt;E&gt;</td>
<td>134</td>
</tr>
<tr>
<td>d2_dictionary&lt;K1, K2, I&gt;</td>
<td>566</td>
</tr>
<tr>
<td>d_array&lt;I, E&gt;</td>
<td>167</td>
</tr>
<tr>
<td>d_int_set</td>
<td>152</td>
</tr>
<tr>
<td>dictionary&lt;K, I&gt;</td>
<td>165</td>
</tr>
<tr>
<td>edge_set</td>
<td>315</td>
</tr>
<tr>
<td>GEN_POLYGON</td>
<td>461</td>
</tr>
<tr>
<td>h_array&lt;Polygon&gt;</td>
<td>170</td>
</tr>
<tr>
<td>interval_set&lt;I&gt;</td>
<td>575</td>
</tr>
<tr>
<td>list&lt;E&gt;</td>
<td>137</td>
</tr>
<tr>
<td>node_pq&lt;P&gt;</td>
<td>319</td>
</tr>
<tr>
<td>node_set</td>
<td>314</td>
</tr>
<tr>
<td>p_dictionary&lt;K, I&gt;</td>
<td>176</td>
</tr>
<tr>
<td>p_queue&lt;P, I&gt;</td>
<td>188</td>
</tr>
<tr>
<td>POLYGON</td>
<td>454</td>
</tr>
<tr>
<td>pp_dictionary&lt;K, I, CMP&gt;</td>
<td>178</td>
</tr>
<tr>
<td>queue&lt;E&gt;</td>
<td>133</td>
</tr>
<tr>
<td>r_circle_gen_polygon</td>
<td>559</td>
</tr>
<tr>
<td>r_circle_polygon</td>
<td>552</td>
</tr>
<tr>
<td>rat_segment_set&lt;I&gt;</td>
<td>579</td>
</tr>
<tr>
<td>segment_set&lt;I&gt;</td>
<td>577</td>
</tr>
<tr>
<td>set&lt;I&gt;</td>
<td>148</td>
</tr>
<tr>
<td>slist&lt;E&gt;</td>
<td>145</td>
</tr>
<tr>
<td>sortseq&lt;K, I&gt;</td>
<td>182</td>
</tr>
<tr>
<td>stack&lt;E&gt;</td>
<td>132</td>
</tr>
<tr>
<td>size()</td>
<td></td>
</tr>
<tr>
<td>graph</td>
<td>278</td>
</tr>
<tr>
<td>node_partition</td>
<td>318</td>
</tr>
<tr>
<td>partition</td>
<td>155</td>
</tr>
<tr>
<td>Partition&lt;E&gt;</td>
<td>157</td>
</tr>
<tr>
<td>size_of_file(...)</td>
<td>37</td>
</tr>
<tr>
<td>size_type</td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>21</td>
</tr>
<tr>
<td>slist&lt;E&gt;</td>
<td>145</td>
</tr>
<tr>
<td>slope()</td>
<td></td>
</tr>
<tr>
<td>line</td>
<td>445</td>
</tr>
<tr>
<td>rat_line</td>
<td>485</td>
</tr>
<tr>
<td>rat_segment</td>
<td>478</td>
</tr>
<tr>
<td>ray</td>
<td>442</td>
</tr>
<tr>
<td>real_line</td>
<td>511</td>
</tr>
<tr>
<td>real_ray</td>
<td>508</td>
</tr>
<tr>
<td>real_segment</td>
<td>504</td>
</tr>
<tr>
<td>segment</td>
<td>438</td>
</tr>
<tr>
<td>small_rational_be&lt;(...&gt;</td>
<td>79, 89</td>
</tr>
<tr>
<td>small_rational_near&lt;(...&gt;</td>
<td>79, 89</td>
</tr>
<tr>
<td>SMALLEST_ENCLOSIN&lt;(...&gt;</td>
<td>530</td>
</tr>
<tr>
<td>smith()</td>
<td></td>
</tr>
<tr>
<td>Tag</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>r_circle_point</td>
<td>543</td>
</tr>
<tr>
<td>tail()</td>
<td></td>
</tr>
<tr>
<td>list&lt;E&gt;</td>
<td>138</td>
</tr>
<tr>
<td>node_list</td>
<td>316</td>
</tr>
<tr>
<td>slist&lt;E&gt;</td>
<td>146</td>
</tr>
<tr>
<td>tail(...)</td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>22</td>
</tr>
<tr>
<td>tangent_at(...)</td>
<td></td>
</tr>
<tr>
<td>r_circle_segment</td>
<td>548</td>
</tr>
<tr>
<td>target()</td>
<td></td>
</tr>
<tr>
<td>d3_rat_segment</td>
<td>614</td>
</tr>
<tr>
<td>d3_segment</td>
<td>589</td>
</tr>
<tr>
<td>r_circle_segment</td>
<td>546</td>
</tr>
<tr>
<td>target(...)</td>
<td>281</td>
</tr>
<tr>
<td>graph</td>
<td>269</td>
</tr>
<tr>
<td>static_graph</td>
<td>289</td>
</tr>
<tr>
<td>tellg()</td>
<td></td>
</tr>
<tr>
<td>decoding_ifstream&lt;Coder&gt;</td>
<td>231</td>
</tr>
<tr>
<td>tellp()</td>
<td></td>
</tr>
<tr>
<td>encoding_ifstream&lt;Coder&gt;</td>
<td>229</td>
</tr>
<tr>
<td>test_bigraph(...)</td>
<td>324</td>
</tr>
<tr>
<td>test_graph(...)</td>
<td>323</td>
</tr>
<tr>
<td>TEXINPUTS</td>
<td>761</td>
</tr>
<tr>
<td>text_box(...)</td>
<td></td>
</tr>
<tr>
<td>window</td>
<td>644</td>
</tr>
<tr>
<td>text_item(...)</td>
<td></td>
</tr>
<tr>
<td>window</td>
<td>655</td>
</tr>
<tr>
<td>third()</td>
<td></td>
</tr>
<tr>
<td>four_tuple&lt;A, B, C, D&gt;</td>
<td>54</td>
</tr>
<tr>
<td>three_tuple&lt;A, B, C&gt;</td>
<td>53</td>
</tr>
<tr>
<td>third_type</td>
<td></td>
</tr>
<tr>
<td>four_tuple&lt;A, B, C, D&gt;</td>
<td>53</td>
</tr>
<tr>
<td>three_tuple&lt;A, B, C&gt;</td>
<td>52</td>
</tr>
<tr>
<td>three_tuple&lt;A, B, C&gt;</td>
<td>52</td>
</tr>
<tr>
<td>timer</td>
<td>46</td>
</tr>
<tr>
<td>tmp_file_name()</td>
<td>37</td>
</tr>
<tr>
<td>to_bigfloat()</td>
<td></td>
</tr>
<tr>
<td>real</td>
<td>86</td>
</tr>
<tr>
<td>to_d3simplex()</td>
<td></td>
</tr>
<tr>
<td>d3_rat_simplex</td>
<td>622</td>
</tr>
<tr>
<td>to_double()</td>
<td></td>
</tr>
<tr>
<td>bigfloat</td>
<td>81</td>
</tr>
<tr>
<td>integer</td>
<td>76</td>
</tr>
<tr>
<td>interval</td>
<td>93</td>
</tr>
<tr>
<td>real</td>
<td>85</td>
</tr>
<tr>
<td>residual</td>
<td>98</td>
</tr>
<tr>
<td>to_double(...)</td>
<td></td>
</tr>
<tr>
<td>bigfloat</td>
<td>81</td>
</tr>
<tr>
<td>integer</td>
<td>76</td>
</tr>
<tr>
<td>interval</td>
<td>93</td>
</tr>
<tr>
<td>real</td>
<td>85</td>
</tr>
<tr>
<td>residual</td>
<td>98</td>
</tr>
</tbody>
</table>
translation(...) ........................................ 537
transpose(...) ......................................... 111
Tree
   a-b ................................................. 733
   AVL .................................................. 733
   BB[α] .................................................. 733
   persistent ......................................... 733
   randomized search ................................ 733
   red-black .......................................... 733
   skiplists ........................................... 733
   tree_collection<T> .................................. 161
   triangle ............................................ 465
   TRIANGLE_COMPONENTS(...) ......................... 528
   triangulate() ......................................
      planar_map ....................................... 295
   triangulate_map() ..................................
      graph ........................................... 277
   triangulate_planar(...) ............................
      graph ........................................... 278
   TRIANGULATE_PLANAR(...) ......................... 373
   TRIANGULATE_PLANE(...) ............................ 527
   TRIANGULATE_POINTS(...) ........................... 526
   TRIANGULATE_POLYGON(...) ......................... 527
   TRIANGULATE_SEGMENTS(...) ......................... 526
   triangulated_planar(...) ...........................
   triangulation_graph(...) ...........................
   triangulation_map(...) ............................. 325, 326
   trivial() ............................................
      GEN_POLYGON ..................................... 460
   truncate(...) ...................................... 79
   truncate(...) ...................................... 45
   memory_streambuf ................................... 242
   tuned_boyer_moore() ................................
      string_matching ................................ 67
   turbo_boyer_moore() ................................
      string_matching ................................ 67
   turbo_reverse_factor() .............................
      string_matching ................................ 67
   TUTTE_EMBEDDING(...) ............................... 376
   two_tuple<A, B> .................................... 51
   U
   ugraph .............................................. 292
   UGRAPH<vtypet, e...> ................................ 292
   undefined(...) .................................... 292
   d_array<1, E> ...................................... 166
   dictionary<K, I> ................................... 164
   h_array<1, E> ...................................... 169
   undo_clear() ....................................... 292
   GraphWin ........................................... 684
   union_blocks(...) ..................................
   node_partition ...................................... 318
   partition .......................................... 155
   Partition<E> ........................................ 157
   unique() ............................................
      array<E> ......................................... 129
      list<E> ........................................... 142
   list<E> ............................................. 142
   rat_vector ......................................... 117
   unique(...) ........................................
   unit(...) ...........................................
   GraphWin ........................................... 683
   unsaved_changes() .................................. 679
   update(...) ........................................
      AdjIt ............................................ 404, 405
      dynamic_trees ................................... 159
      EdgeIt ............................................ 395
      FaceCirc ......................................... 407
      FaceIt ............................................ 397
      InAdjIt .......................................... 402
      NodeIt ............................................ 394
      OutAdjIt ......................................... 399, 400
   update_graph() ....................................
      GraphWin ........................................ 672
   upper_bound() .....................................
      h_priority_queue<1> .............................. 191
      interval ........................................... 93
   upper_CONVEX_HULL(...) ............................. 525
   upper_left() ......................................
      rat_rectangle ................................... 494
      real_rectangle .................................. 521
      rectangle ........................................ 468
   upper_right() .....................................
      rat_rectangle ................................... 494
      real_rectangle .................................. 521
      rectangle ........................................ 468
   use_edge_data(...) ................................
<table>
<thead>
<tr>
<th>Y</th>
<th>Y()</th>
</tr>
</thead>
<tbody>
<tr>
<td>d3_rat_point</td>
<td>600</td>
</tr>
<tr>
<td>rat_point</td>
<td>472</td>
</tr>
<tr>
<td>rat_vector</td>
<td>117</td>
</tr>
<tr>
<td>Y1()</td>
<td></td>
</tr>
<tr>
<td>d3_rat_point</td>
<td>601</td>
</tr>
<tr>
<td>rat_point</td>
<td>472</td>
</tr>
<tr>
<td>Y2()</td>
<td></td>
</tr>
<tr>
<td>d3_rat_point</td>
<td>601</td>
</tr>
<tr>
<td>rat_point</td>
<td>472</td>
</tr>
<tr>
<td>y_abs()</td>
<td></td>
</tr>
<tr>
<td>line</td>
<td>445</td>
</tr>
<tr>
<td>rat_line</td>
<td>485</td>
</tr>
<tr>
<td>rat_segment</td>
<td>479</td>
</tr>
<tr>
<td>real_line</td>
<td>511</td>
</tr>
<tr>
<td>real_segment</td>
<td>504</td>
</tr>
<tr>
<td>segment</td>
<td>439</td>
</tr>
<tr>
<td>y_proj(...)</td>
<td></td>
</tr>
<tr>
<td>line</td>
<td>445</td>
</tr>
<tr>
<td>rat_line</td>
<td>485</td>
</tr>
<tr>
<td>rat_segment</td>
<td>478</td>
</tr>
<tr>
<td>real_line</td>
<td>511</td>
</tr>
<tr>
<td>real_segment</td>
<td>504</td>
</tr>
<tr>
<td>segment</td>
<td>438</td>
</tr>
<tr>
<td>ycoord()</td>
<td></td>
</tr>
<tr>
<td>d3_point</td>
<td>582</td>
</tr>
<tr>
<td>d3_rat_point</td>
<td>601</td>
</tr>
<tr>
<td>point</td>
<td>432</td>
</tr>
<tr>
<td>rat_point</td>
<td>472</td>
</tr>
<tr>
<td>rat_vector</td>
<td>117</td>
</tr>
<tr>
<td>real_point</td>
<td>498</td>
</tr>
<tr>
<td>real_vector</td>
<td>121</td>
</tr>
<tr>
<td>vector</td>
<td>104</td>
</tr>
<tr>
<td>ycoord1()</td>
<td></td>
</tr>
<tr>
<td>d3_rat_segment</td>
<td>614</td>
</tr>
<tr>
<td>d3_segment</td>
<td>589</td>
</tr>
<tr>
<td>rat_segment</td>
<td>477</td>
</tr>
<tr>
<td>real_segment</td>
<td>504</td>
</tr>
<tr>
<td>segment</td>
<td>438</td>
</tr>
<tr>
<td>ycoord1D()</td>
<td></td>
</tr>
<tr>
<td>rat_segment</td>
<td>477</td>
</tr>
<tr>
<td>ycoord2()</td>
<td></td>
</tr>
<tr>
<td>d3_rat_segment</td>
<td>614</td>
</tr>
<tr>
<td>d3_segment</td>
<td>589</td>
</tr>
<tr>
<td>zcoord()</td>
<td></td>
</tr>
<tr>
<td>d3_point</td>
<td>582</td>
</tr>
<tr>
<td>d3_rat_point</td>
<td>601</td>
</tr>
<tr>
<td>real_segment</td>
<td>122</td>
</tr>
<tr>
<td>vector</td>
<td>104</td>
</tr>
<tr>
<td>zcoord1()</td>
<td></td>
</tr>
<tr>
<td>d3_rat_segment</td>
<td>614</td>
</tr>
<tr>
<td>d3_segment</td>
<td>589</td>
</tr>
<tr>
<td>zcoord2()</td>
<td></td>
</tr>
</tbody>
</table>
\begin{verbatim}
  d3_rat_segment .................. 614
  d3_segment ..................... 589
  zcoordD()
    d3_rat_point .................. 601
  ZD()
    d3_rat_point .................. 601
  zdist(....)
    d3_point ....................... 583
    d3_rat_point .................. 602
  zero(....)
    rat_vector ...................... 117
  zero_of_function(....) .......... 126
  zhu_takaoka()
    string_matching ................ 67
  zoom(....)
    GraphWin ....................... 679
  zoom_area(....)
    GraphWin ....................... 679
  zoom_down()
    GeoWin .......................... 716
  zoom_graph()
    GraphWin ....................... 679
  zoom_up()
    GeoWin .......................... 716
\end{verbatim}