Arrays, Lists, Stacks, Queues, Trees

Chionh Eng Wee
Computer Science Department
School of Computing
chionhew@comp.nus.edu.sg

May 22, 2004

Abstract

In C and Pascal, arrays and records are the simplest aggregating mechanisms. More complicated data structures such as lists, stacks, queues, and trees can be built on top of (1) arrays whose elements can be records, or (2) records with pointers, or (3) both arrays and records.

This talk is mainly about implementation and is based on C.

Arrays

- The simplest aggregating mechanism in C and Pascal.
- An array has two parts: indices and elements.
- The disadvantage of arrays.

  An array has a fixed size (number of elements).
- The advantage of arrays.

  Its elements can be accessed directly and simply when the index value of an element is known.

Using Arrays

When using arrays in a programming language, we need to know the following:

- What are the allowable types for the index and the element?
- How is an array defined and initialized?
- Can the size of an array be determined at execution time?
- Are out of range indices caught at run-time?

C Arrays

- The size must be either declared/allocated explicitly or implied by initialization.
- The index expression must evaluate to an integer and is expected to be in the range 0..size-1 inclusively.
- Define using either [] or dynamic allocation.
- Some C compilers support variable-size arrays defined with [] but this is not ANSI. Find out using "gcc -ansi -pedantic".
- No range checking during both compile time and run time.

Example: Variable-Size Arrays with []

```
$ cat 3.c
see( int s )
{
  char a,  b[s],  c;

  printf( "%4d %08x %08x %08x\n", s, &a, &b[0], &c );
}

main( int ac, char *av[] )
{
  see( 1 );  see( 10 );  see( 100 );  see( 1000 );
}
```

Example: Compile and Execute

```
$ gcc -ansi -pedantic 3.c
3.c: In function ‘see’:
3.c:3: warning: ANSI C forbids variable-size array ‘b’

$ a.out
   1 ffbef7c7 ffbef7b8 ffbef7c6
  10 ffbef7c7 ffbef7b0 ffbef7c6
 100 ffbef7c7 ffbef758 ffbef7c6
1000 ffbef7c7 ffbef3d8 ffbef7c6
```

Multi-Dimensional Variable-Size C Arrays

- Use `#define` for multiple indices.
- Use `calloc()` to allocate storage and initialize it to zero.
- We still think in terms of the indices, how the indices affect the actual storage location does not affect the application logic.

Example: 2-Dimensional Variable-Size Arrays

```
$ cat 4.c
#include <stdlib.h>

#define IDX(i,j)        ((i)*n + (j))

main( int ac, char *av[] )
{
  int  *x, m, n, i, j, k;

  if( ac < 3 ) {
    printf( "a.out m n\n" );
    return -1;
  }
```

```
  m = atoi( av[1] );
  n = atoi( av[2] );

  x = calloc( m*n, sizeof(int) );

  for( k = i = 0;  i < m;  i++ )
    for( j = 0;  j < n;  j++ )
      x[IDX(i,j)] = k++;

  for( k = 0;  k < m*n;  k++ ) {
    if( !(k % n) ) printf( "\n" );
    printf( " %d", x[k] );
  }
  printf( "\n" );
}
```

```
$ cc 4.c
$ a.out 2 3

 0 1 2
 3 4 5
$ a.out 3 2

 0 1
 2 3
 4 5
```

Another Example

```
$ cat 4a.c
#include <stdlib.h>

#define IDX(i,j)        ((i) + (j)*m)

main( int ac, char *av[] )
{
  int  *x, m, n, i, j, k;

  if( ac < 3 ) {
    printf( "a.out m n\n" );
    return -1;
  }
```

```
  m = atoi( av[1] );
  n = atoi( av[2] );

  x = calloc( m*n, sizeof(int) );

  for( k = i = 0;  i < m;  i++ )
    for( j = 0;  j < n;  j++ )
      x[IDX(i,j)] = k++;

  for( k = 0;  k < m*n;  k++ ) {
    if( !(k % m) ) printf( "\n" );
    printf( " %d", x[k] );
  }
  printf( "\n" );
}
```

```
$ cc 4a.c
$ a.out 2 3

 0 3
 1 4
 2 5
$ a.out 3 2

 0 2 4
 1 3 5
```

Variable-Size 3-Dimensional Arrays

Let the indices be i, j, k and

$$
\begin{aligned}
i &= 0, \cdots, l-1 \\
j &= 0, \cdots, m-1 \\
k &= 0, \cdots, n-1
\end{aligned}
$$

The storage for an element indexed by (i, j, k) within an area for lmn elements can be allocated as any of:

$$
\begin{gathered}
imn + jn + k \\
imn + j + km \\
in + jln + k \\
i + jln + kl \\
i + jl + klm \\
im + j + klm
\end{gathered}
$$

Example: Variable-Size 3-Dimensional Arrays

```
#include <stdlib.h>

#define Xijk(i,j,k) ((i)*m*n+(j)*n+(k))
#define Xikj(i,j,k) ((i)*m*n+(k)*m+(j))
#define Xjik(i,j,k) ((j)*l*n+(i)*n+(k))
#define Xjki(i,j,k) ((j)*l*n+(k)*l+(i))
#define Xkij(i,j,k) ((k)*l*m+(i)*m+(j))
#define Xkji(i,j,k) ((k)*l*m+(j)*l+(i))
```

```
main( int ac, char *av[] )
{
  int  i, j, k, l, m, n, *A, cnt;

  if( ac < 4 ) {
    printf( "a.out l m n\n" );
    return -1;
  }

  l = atoi( av[1] );
  m = atoi( av[2] );
  n = atoi( av[3] );

  A = calloc( l*m*n, sizeof(int) );
```

```
cnt = 0;
for( i = 0;  i < l;  i++ )
for( j = 0;  j < m;  j++ )
for( k = 0;  k < n;  k++ )
  A[Xijk(i,j,k)] = cnt++;
for(cnt = 0; cnt < l*m*n; cnt++) printf( " %d", A[cnt] );
printf( "\n" );

cnt = 0;
for( i = 0;  i < l;  i++ )
for( j = 0;  j < m;  j++ )
for( k = 0;  k < n;  k++ )
  A[Xikj(i,j,k)] = cnt++;
for(cnt = 0; cnt < l*m*n; cnt++) printf( " %d", A[cnt] );
printf( "\n" );
```

```
cnt = 0;
for( i = 0;  i < l;  i++ )
for( j = 0;  j < m;  j++ )
for( k = 0;  k < n;  k++ )
  A[Xjik(i,j,k)] = cnt++;
for(cnt = 0; cnt < l*m*n; cnt++) printf( " %d", A[cnt] );
printf( "\n" );

cnt = 0;
for( i = 0;  i < l;  i++ )
for( j = 0;  j < m;  j++ )
for( k = 0;  k < n;  k++ )
  A[Xjki(i,j,k)] = cnt++;
for(cnt = 0; cnt < l*m*n; cnt++) printf( " %d", A[cnt] );
printf( "\n" );
```

```
  cnt = 0;
  for( i = 0;  i < l;  i++ )
  for( j = 0;  j < m;  j++ )
  for( k = 0;  k < n;  k++ )
    A[Xkij(i,j,k)] = cnt++;
  for(cnt = 0; cnt < l*m*n; cnt++) printf( " %d", A[cnt] );
  printf( "\n" );

  cnt = 0;
  for( i = 0;  i < l;  i++ )
  for( j = 0;  j < m;  j++ )
  for( k = 0;  k < n;  k++ )
    A[Xkji(i,j,k)] = cnt++;
  for(cnt = 0; cnt < l*m*n; cnt++) printf( " %d", A[cnt] );
  printf( "\n" );
}
```

```
$ a.out 2 2 3
 0 1 2 3 4 5 6 7 8 9 10 11
 0 3 1 4 2 5 6 9 7 10 8 11
 0 1 2 6 7 8 3 4 5 9 10 11
 0 6 1 7 2 8 3 9 4 10 5 11
 0 3 6 9 1 4 7 10 2 5 8 11
 0 6 3 9 1 7 4 10 2 8 5 11
```

Records

- We may say an array is a homogeneous record (treat array elements as fields within a record) and a record is a heterogeneous array (treat record fields as array elements).
- Record structure is perhaps the next simplest aggregating mechanism in C and Pascal.
- In C, the keyword **struct** defines a record.
- The member operator "·" or the pointer operator "->" are used to access the fields of a record.
- An element of an array and a field of a record can be accessed directly. But unlike arrays, there is no way to step through the fields of a record in a loop.

Disadvantages of Arrays

Arrays are fixed size. Even with variable-size arrays (not supported in ANSI), once created, the size cannot be changed.

Another disadvantage is that elements cannot be inserted or deleted easily. To insert or delete an element, other elements following the element have to be moved.

Advantages of Lists

- Lists are a very flexible structure:
- A list grows and shrinks on demand.
- Elements can be inserted or deleted at any position within a list without moving other list elements.
- But unlike arrays, a list element cannot be accessed directly even when you know where it is within a list.

Lists

Mathematically, a list is a sequence of elements

$$e_1, e_2, \cdots, e_n$$

that supports the following operations, among others:

- Insert an element x at certain list position:

$$e_1, \cdots, e_i, x, e_{i+1}, \cdots, e_n$$

- Delete an element:

$$e_1, \cdots, e_{i-1}, e_{i+1}, \cdots, e_n$$

- Find an element e_i in the list based on some properties of e_i.
- Print some attributes of each list element in their list order.

More List Operations

The above operations act on a single list.

For two lists, concatenation of one list to the end of another is also an important operation.

Example: Lists

```
#include <stdio.h>

struct node {
  int  value;
  struct node *next;
};
```

```
struct node * printList( struct node * l )
{
  struct node *t = l;

  while( l ) {
    printf( " %d", l->value );
    l = l->next;
  }
  printf( "\n" );

  return t;
}
```

```
struct node * concatList( struct node * f, struct node * b )
{
  struct node * h = f;

  if( !f ) return b;
  if( !b ) return f;

  while( f->next ) f = f->next;
  f->next = b;

  return h;
}
```

```
struct node * deleteList( struct node * l, int v )
{
  struct node *p, *n;

  if( !l ) return l;

  if( l->value == v ) { n = l->next;  free( l );  return n; }

  n = l;
  while( n && n->value != v ) { p = n;  n = n->next; }
  if( n && n->value == v ) {
    p->next = n->next;  free( n );
  }
  return l;
}
```

```
main( int ac, char *av[] )
{
  struct node *a, *b, *t;

  /* a single element list: 1 */
  a = (struct node *) calloc( 1, sizeof(struct node) );
  a->value = 1;

  /* another single element list: 9 */
  b = (struct node *) calloc( 1, sizeof(struct node) );
  b->value = 9;

  printList( a );
```

```
/* a 2-element list: 1, 9 */
a->next = b;

printList( a );

b = (struct node *) calloc( 1, sizeof(struct node) );
b->value = 5;

/* a 3-element list: 1, 5, 9 */
t = a->next;
a->next = b;
b->next = t;
printList( a );
```

```
  /* a 2-element list: 13, 19 */
  b = (struct node *) calloc( 1, sizeof(struct node) );
  b->value = 13;
  b->next = (struct node *) calloc(1,sizeof(struct node));
  b->next->value = 19;
  printList( b );

  /* a 5-element list: 1, 5, 9, 13, 19 */
  a = concatList( a, b );
  printList( a );

  a = printList( deleteList( a, 13 ) );
  a = printList( deleteList( deleteList( a, 1 ), 19 ) );
  a = printList( deleteList( a, 5 ) );
}
```

```
$ cc 5.c

$ a.out
 1
 1 9
 1 5 9
 13 19
 1 5 9 13 19
 1 5 9 19
 5 9
 9
```

List Variants

The above list is known as a singly linked list.

A singly link list can also be given by two pointers, head and tail, so that both the first and the last elements can be accessed immediately.

If it is important to traverse the list both forward and backward, the doubly linked list should be used.

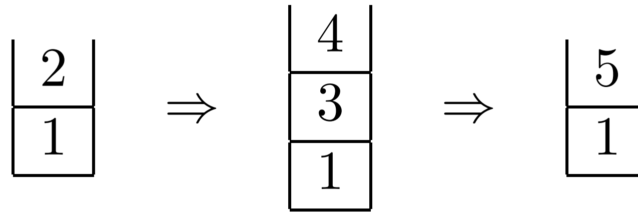
Stacks

Mathematically a stack is a list with a restricted insert (push) and a restricted delete (pop). An element can only be inserted to become the first (top) element and only the first (top) element can be deleted. Thus a stack has only one end, the top of the stack, and enforces a last-in-first-out discipline.

For example, the stack is initially empty, after the following operations:

$$push(1), push(2), pop(), push(3), push(4), pop(), pop(), push(5)$$

the stack contents are



Postfix Expressions

A postfix expression has the wonderful property that its order of evaluation is completely determined by the expression itself without the need of parentheses and rules of precedence and association.

Formally, postfix expressions are expressions generated by the rules:

$$
\begin{aligned}
E &\rightarrow EEO \\
E &\rightarrow number \\
O &\rightarrow + \mid - \mid * \mid /
\end{aligned}
$$

For example,

$$
\begin{aligned}
E_0 &\overset{*}{\Rightarrow} E_1 E_2 + \\
&\overset{*}{\Rightarrow} E_1 E_5 E_6 * + \\
&\overset{*}{\Rightarrow} E_3 E_4 + E_5 E_6 * + \\
&\overset{*}{\Rightarrow} E_7 E_8 * E_4 + E_5 E_6 * +
\end{aligned}
$$

Evaluating Postfix Expressions

Postfix expressions can be easily evaluated by a stack:

1. Read next symbol X.
2. If X is EOF then return pop().
3. If X is an operator then L = pop(), R = pop(), push(L X R); else push(X).
4. Goto 1.

Example: Evaluating Postfix Expressions

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>

struct node {
  int value;
  struct node *next;
};
```

```
struct node * push( struct node **stack, int value )
{
  struct node *s;

  s = (struct node *)
    calloc( 1, sizeof(struct node) );
  s->value = value;
  s->next = *stack;
  *stack = s;
  return s;
}
```

```
int pop( struct node ** stack )
{
  int v;
  struct node *s;

  if( ! *stack ) {
    printf( "poping empty stack\n" );
    exit( -1 );
  }

  s= *stack;
  v = s->value;
  *stack = s->next;
  free(s);
  return v;
}
```

```
void print( struct node * stack )
{
  while( stack ) {
    printf( " %d", stack->value );
    stack = stack->next;
  }
  printf( "\n" );
}
```

```
main( int ac, char *av[] )
{
  struct node *s = NULL;
  int i, result, left, rite;
  char *c;

  if( ac < 2 ) {
    printf( "a.out strings\n" );
    return -1;
  }
```

```
for( i = 1;  i <= ac-1;  i++ ) {

  if( strcmp( av[i], "+" ) == 0 ) {
    push( &s, pop(&s) + pop(&s) );
    print( s );
    continue;
  }

  if( strcmp( av[i], "x" ) == 0 ) {
    push( &s, pop(&s) * pop(&s) );
    print( s );
    continue;
  }
```

```
if( strcmp( av[i], "-" ) == 0 ) {
  rite = pop( &s );
  left = pop( &s );
  push( &s, left - rite );
  print( s );
  continue;
}

if( strcmp( av[i], "/" ) == 0 ) {
  rite = pop( &s );
  left = pop( &s );
  push( &s, left / rite );
  print( s );
  continue;
}
```

```
    push( &s, atoi( av[i] ) );
    print( s );

  }
  printf( "%d\n", pop(&s) );
}
```

```
$ cc 7.c

$ a.out 6 5 2 3 + 8 x + 3 + x
 6
 5 6
 2 5 6
 3 2 5 6
 5 5 6
 8 5 5 6
 40 5 6
 45 6
 3 45 6
 48 6
 288
288
```

Implementing Stacks Using Arrays

Depending on the problem, sometimes it is very convenient to implement stacks using arrays. In order to do this the array must be large enough to hold the highest possible stack.

The following example checks if a sequence of `(`, `)`, `[`, `]`, `{`, `}`, is balanced. Such a sequence is balanced if after repeatedly canceling pairs of `( )`, `[ ]`, `{ }`, there are no symbols left.

Example: Stacks by Arrays

```
char *Stack;
int Top = -1;

void push( char value )
{
  Stack[++Top] = value;
}

char pop()
{
  if( Top < 0 ) return -1;
  return Stack[Top--];
}
```

```
main( int ac, char *av[] )
{
  int i;
  char c;

  if( ac < 2 ) return 0;

  Stack = (char*)calloc(ac-1,sizeof(char));
  Top = -1;
```

```
for( i = 1;  i <= ac-1;  i++ ) {
  c = av[i][0];
  switch( c ) {
  case '(':  case '[':  case '{':
    push( c ); break;
  case ')':
    if( '(' != pop() ) return -1;
    break;
  case ']':
    if( '[' != pop() ) return -1;
    break;
  case '}':
    if( '{' != pop() ) return -1;
    break;
  default:
```

```
      return -1;
    }
  }

  if( Stack[Top] != 0 ) return -1;
  else return 0;
}
```

```
$ cc 8.c

$ a.out [
$ echo $?
255

$ a.out [ ]
$ echo $?
0

$ a.out [ [ { } ] ] { { ] ]
$ echo $?
255

$ a.out [ [ { } ] ] { [ ] }
$ echo $?
```

```
0

$ a.out [ { ] }
$ echo $?
255
```

Queues

Like stacks, mathematically a queue is a list with a restricted insert (enqueue) and a restriced delete (dequeue). An element can only be inserted to become the last element and only the first element can be deleted. Thus a queue has two ends front and rear (also called head and tail) and enforces a first-in-first-out discipline.

For example, with an initially empty queue, after the operations

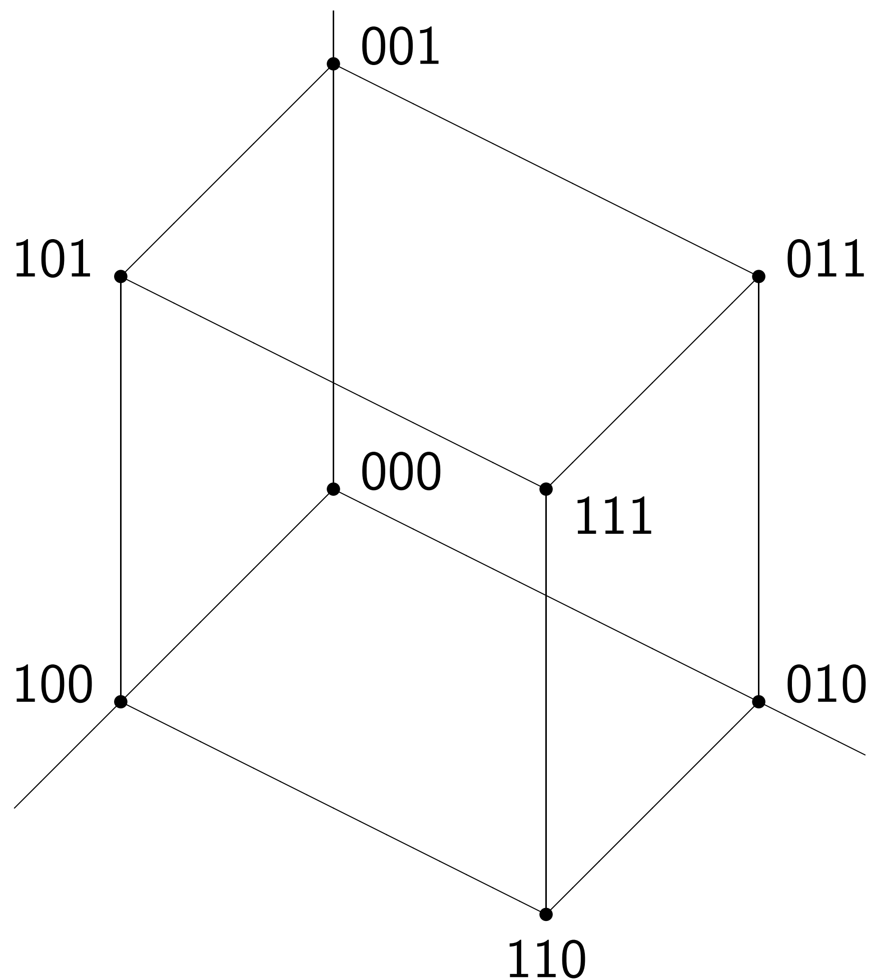
$$enq(1), enq(4), enq(5), deq(), enq(3), deq(), enq(6);$$

the queue contents are

$$5, 3, 6.$$

Example: A Queue Application

Consider the following cube:



List vertices connected to a given vertex by "distance". (Breadth first search in disguise.)

The Program

```
#define IX(i,j) ((i)*Vertex_Count+(j))

int  Vertex_Count;
char *Visited;
char *Adjacency;

struct node {
  int  value;
  struct node *next;
};

struct node *Head = 0, *Tail = 0;
```

```
int empty()
{
  return (Head == 0);
}
```

```
struct node * enqueue( int v )
{
  struct node *n;

  n = (struct node *)
      calloc( 1, sizeof(struct node) );
  n->value = v;
  if( ! Tail )
    Head = Tail = n;
  else {
    Tail->next = n;
    Tail = n;
  }
}
```

```
int dequeue() {
  int value;
  struct node *h;

  if( !Head ) {
    printf( "dequeue empty queue\n" );
    exit( -1 );
  }
  h = Head;
  value = h->value;
  Head = h->next;
  if( ! Head ) Tail = Head;
  free( h );
  return value;
}
```

```
void breadthFirstSearch( int v )
{
  int  x, y;

  Visited[v] = 1;
  enqueue( v );

  while( ! empty() ) {
    x = dequeue();
    for( y = 0;  y < Vertex_Count;  y++ ) {
      if( Adjacency[IX(x,y)] ) {
        if( ! Visited[y] ) {
          Visited[y] = 1;
          enqueue( y );
          printf( "%d %d\n", x, y );
        }
```

```
            }
        }
    }
}
```

```
main( int ac, char *av[] )
{
  int  e, i, x, y;

  scanf( "%d", &Vertex_Count );

  Visited = (char *)
    calloc(Vertex_Count,sizeof(char));
  Adjacency = (char *)
    calloc( Vertex_Count*Vertex_Count,
      sizeof(char) );

  scanf( "%d", &e );
  for( i = 0;  i < e;  i++ ) {
    scanf( "%d %d", &x, &y );
```

```
    Adjacency[IX(x,y)] =
    Adjacency[IX(y,x)] = 1;
  }

  breadthFirstSearch( atoi( av[1] ) );
}
```

Cube Data

```
8
12
0 1
0 2
0 4
1 3
1 5
2 3
2 6
3 7
4 5
4 6
5 7
6 7
```

```
$ for i in 0 1 2 3 4 5 6 7
> do
> echo start at vertex $i
> a.out $i < cube
> done
```

Output

```
start at vertex 0
0 1
0 2
0 4
1 3
1 5
2 6
3 7
start at vertex 1
1 0
1 3
1 5
0 2
0 4
3 7
2 6
```

```
start at vertex 2
2 0
2 3
2 6
0 1
0 4
3 7
1 5
start at vertex 3
3 1
3 2
3 7
1 0
1 5
2 6
0 4
```

```
start at vertex 4
4 0
4 5
4 6
0 1
0 2
5 7
1 3
start at vertex 5
5 1
5 4
5 7
1 0
1 3
4 6
0 2
```

```
start at vertex 6
6 2
6 4
6 7
2 0
2 3
4 5
0 1
start at vertex 7
7 3
7 5
7 6
3 1
3 2
5 4
1 0
```

Trees

Arrays, lists, stacks, queues represent linear structures. To represent non-linear hierarchical structures, trees are used.

In computing, a tree is a collection of nodes with some parent-child relationship. This relationship can be defined recursively as follows:

1. A single node is a tree with itself as the root of the tree.

2. If n is a node and $T_1, \cdots, T_k$ are trees with roots $n_1, \cdots, n_k$ respectively. A new tree can be constructed with n as the root and $T_1, \cdots, T_k$ as subtrees of the root. In this new tree, nodes $n_1, \cdots, n_k$ are called the children of node n. Node n is known as the parent of nodes $n_1, \cdots, n_k$.

Representing Trees

- By one parent array.

  n nodes are identified with integers $0, \cdots, n-1$. The parent of i is given by the array $P[i]$.

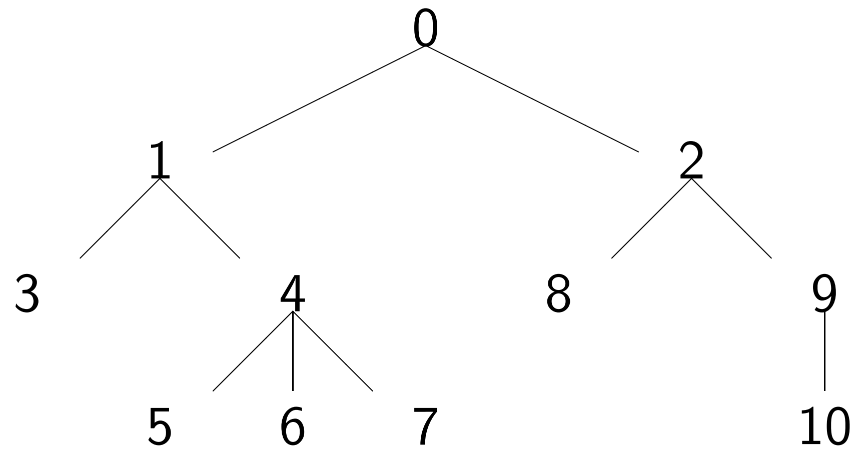
- By child/sibling lists.

  Each node is represented as a record which includes two fields FirstChild and NextSibling.

- By an array of child lists.

  n nodes are identified with integers $0, \cdots, n-1$. Array element $C[i]$ is a list of the children of node i

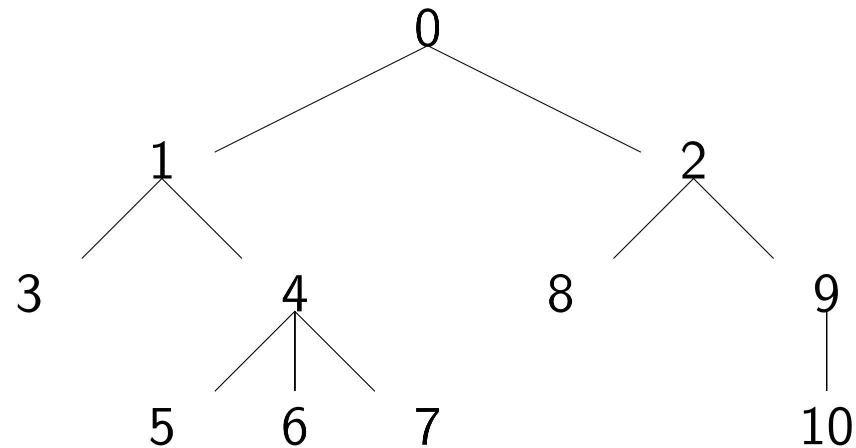
Example: Representing Trees



The parent-of array:

-1	0	0	1	1	4	4	4	2	2	9

Example: Representing Trees



The children-of array/lists:

L_0	L_1	L_2	0	L_4	0	0	0	0	L_9	0

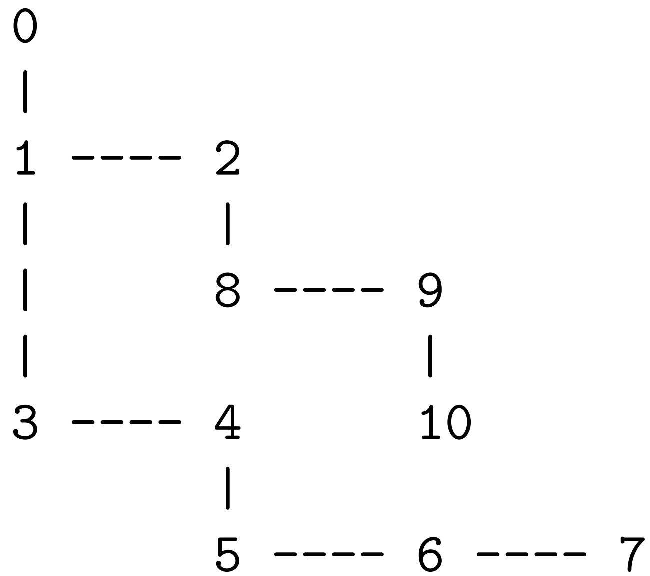
$$L_0 \to 1, 2. \quad L_1 \to 4, 5. \quad L_2 \to 8, 9. \quad L_4 \to 6, 7, 8.$$

Example: Representing Trees

A possible list node structure:

```
struct FirstSibling {
  int  element;
  struct FirstSibling  *first;
  struct FirstSibling  *sibling;
};
```

Use horizontal line and vertical line to represent first and sibling list respectively, the tree in the previous example is



Tree Traversal

One of the important tree operations is tree traversal. There are three ways (orders) to traverse a tree: preorder, inorder, postorder. These orderings are defined recursively as follows:

1. If a tree has only the root, the root by itself is the preorder, inorder, postorder traversal of the tree.

2. If a tree has root n and subtrees T_1, $\cdots$, T_k.

   The preorder traversal of the tree is

   $$n, preorder(T_1), \cdots, preorder(T_k)$$

   The inorder traversal of the tree is

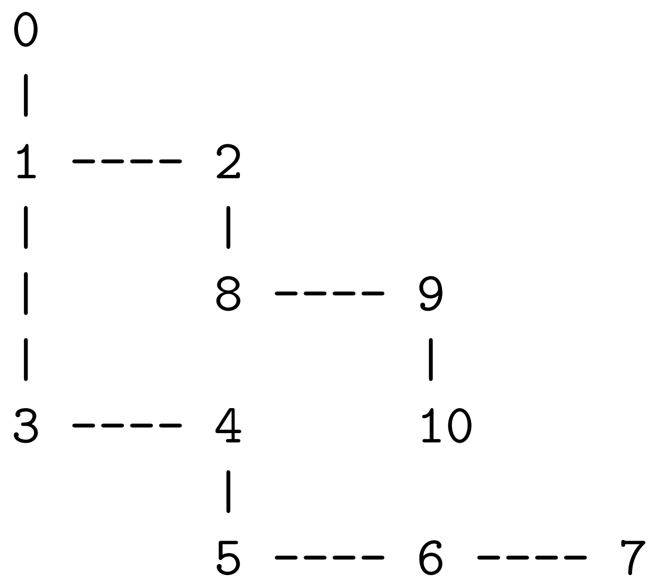
   $$inorder(T_1), n, inorder(T_2), \cdots, inorder(T_k)$$

   The postorder traversal of the tree is

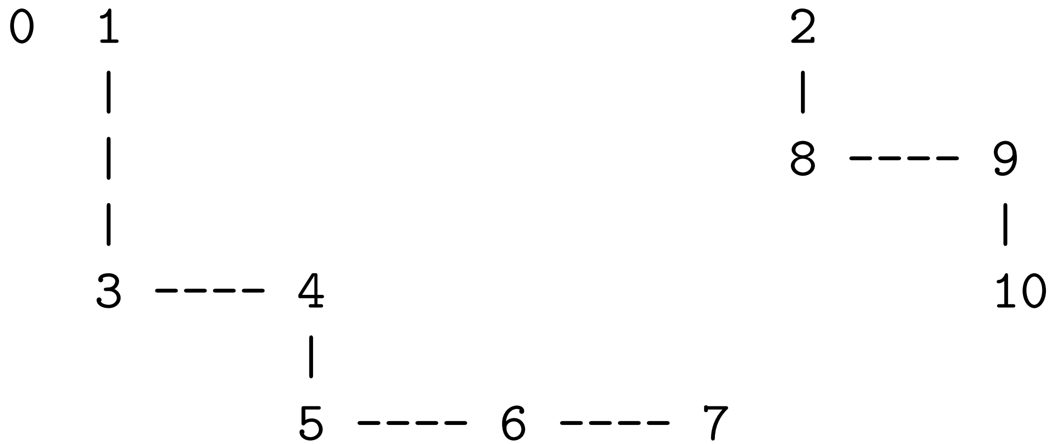
   $$postorder(T_1), \cdots, postorder(T_k), n$$

Example: Preorder Traversal

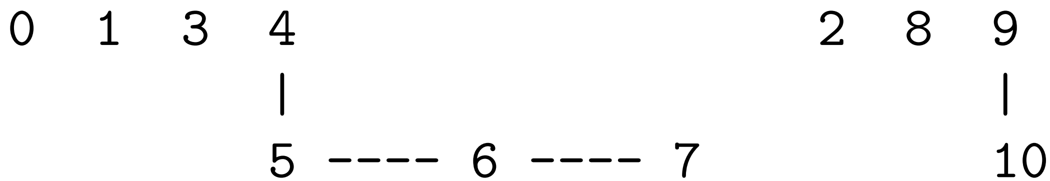
- The given tree:



- 0, subtree with root 1, subtree with root 2:



- 0, 1, subtree with roots 3, 4; 2, subtrees of roots 8, 9:



- The inorder traversal of the given tree:

0 1 3 4 5 6 7 2 8 9 10

Example: Inorder and Postorder

Similarly, the inorder traversal of the given tree is:

3 1 5 4 6 7 0 8 2 10 9

The postorder traversal of the given tree is:

3 5 6 7 4 1 8 10 9 2 0