Topics

- Arrays and lists, comparison.
- Stacks and queues, implementation.
- Trees, rooted trees, binary trees, traversal.
- Disjoint sets.
- Heaps.
- (All diagrams will be provided during the lecture.)

Arrays

- Mathematically, an array is a (finite) sequence of elements, each of which can be accessed directly.
- Arrays are important because many data structures/abstract data types can be implemented in arrays: stacks, queues, sets, graphs, priority queues, etc.

Arrays: Programming

- How is an array declared and initialized?
- How is an array element accessed?
- Can we use an array without declaring one? DIY arrays?

Array Declaration, Initialization, and Use

- Declare array attributes:

  name, dimensions, array element type, and range of indices.

- C arrays can be initialized at declaration time, unlike Pascal arrays.

- An array element is accessed by the "index operator" [ ]:

$$a[i], b[i][j], c[i, j]$$

C DIY "Arrays" (Size Unknown in Advance)

```
#define N       20
main() {
  int *x, i, s;

  x = (int *) calloc( N, sizeof(int) );
  for( i = 0;  i < N;  i++ ) x[i] = i*i;
  s = 0;
  for( i = 0;  i < N;  i++ ) s += x[i];
  printf( "%d\n", s );
}
$ a.out
2470
```

Many C Compilers Support Dynamic Array Size

```
#define N       20
void y( int n ) {
  int x[n], i, s;
  for( i = 0;  i < n;  i++ ) x[i] = i*i;
  s = 0;
  for( i = 0;  i < n;  i++ ) s += x[i];
  printf( "%d\n", s );
}

main() { y( N ); }
$ a.out
2470
```

Dynamic Multi-Dimensinal Array Sizes

```
void x( int m, int n ) {
  int y[m][n], i, j, s;

  for( i = 0;  i < m;  i++ )
    for( j = 0;  j < n;  j++ )
      y[i][j] = i+j;
  s = 0;
  for( i = 0;  i < m;  i++ )
    for( j = 0;  j < n;  j++ )
      s += y[i][j];

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

```
main( int ac, char *av[] ) {
  x( atoi( av[1] ), atoi( av[2] ) );
}
```

```
$ cc 3.c
$ a.out 151 29
389731
```

```
$ gcc 3.c
$ a.out 151 29
389731
```

Why DIY Arrays?

- Maximum array sizes are not known in advance.
- The compiler may not support dynamic array size.
- When an array is declared inside a function, the array is deallocated from the stack when the function exits.

C DIY Arrays

```
#define X(i,j)  x[m*(i)+j] /* blocks of m elements */
#define Y(i,j)  x[n*(i)+j] /* blocks of n elements */

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

  m = atoi( av[1] );  n = atoi( av[2] );
  x = (int *) calloc( m*n, sizeof(int) );
```

```
#define X(i,j)  x[m*(i)+j]
#define Y(i,j)  x[n*(i)+j]

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

  m = atoi( av[1] );  n = atoi( av[2] );
  x = (int *) calloc( m*n, sizeof(int) );

  for( i = 0;  i < n;  i++ )
    for( j = 0;  j < m;  j++ )
      X(i,j) = i+j;

  for( s = i = 0;  i < n;  i++ )
    for( j = 0;  j < m;  j++ )
      s += X(i,j);
```

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

  for( s = i = 0;  i < m;  i++ )
    for( j = 0;  j < n;  j++ )
      s += Y(i,j);  // Note that Y(i,j) != i+j
  printf( "%d\n", s );
}
```

```
$ a.out 151 29
389731
389731

$ a.out 29 151
389731
389731
```

Link Lists

- Mathematically, a link list is a (finite) sequence of elements, each of which can be accessed serially.
- By serially we mean to access an element, we have to access its predecessor first (singly linked) or its successor first (doubly linked).
- An element of a linked list is implemented as a record.
- One (or two if doubly linked) field of the record is a pointer, which gives the address of the successor record.
- Like arrays, other data structures such as stacks, queues, sets, graphs, can be implemented using linked lists.

Linked List: Example 1

```
struct node {
  int key;
  struct node *nxt;
};

main() {
  struct node x, y, z, *l;

  x.key = 0;  y.key = 1;  z.key = 2;
  x.nxt = &y;  y.nxt = &z;  z.nxt = 0;

  for( l = &x;  l;  l = l->nxt ) printf( "%d\n", l->key );
}
```

```
$ a.out
0
1
2
```

Example 2

```
struct node {
  int key;
  struct node *nxt;
};

struct node *x;

print( struct node *l ) {
  for( ;  l;  l = l->nxt ) printf( "%d\n", l->key );
}
```

```
void insert( struct node **p, int v ) {
  struct node *l;

  while( *p && (*p)->key < v ) p = &((*p)->nxt);

  l = *p;
  *p = (struct node *) calloc( 1, sizeof(struct node) );
  (*p)->key = v;
  (*p)->nxt = l;
}
```

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

  for( i = 1;  i < ac;  i++ ) {
    insert( &x, atoi( av[i] ) );
  }

  print( x );
}
```

```
$ a.out 13 1 2 3 9 8 7 5 4 3
1
2
3
3
4
5
7
8
9
13
```

Arrays versus Linked Lists

Arrays	Linked Lists
Fixed number of elements	Shrinks and grows
Direct access by index	Serial access by chasing pointers
Insert at prefix involves shifting	Insert anywhere without shifting
Delete at prefix involves shifting	Delete anywhere without shifting
Trivial to find successor	Trivial to find successor
Trivial to find predecessor	To find predecessor needs doubly linked

Stacks

- Mathematically, a stack is a (finite) sequence that provides the last-in/first-out insertion/deletion discipline.
- A stack is supposed to be used with two operations: $push(v)$, $pop()$.
- It can be easily implemented using an array and a variable (for the top of the stack) if the maximum stack depth can be determined in advance.

Stacks by Arrays

```
#define MAX 100
int s[MAX], top;

init() { top = -1; }
is_empty() { return top == -1; }

push( int v ) {
  if(++top >= MAX) error("overflow"); else s[top]=v;
}

pop() {
  if(is_empty()) error("underflow"); else return s[top--];
}
```

Stack Example

```
#define MAX     100

int s[MAX], top = -1;
```

```
void push( int v ) {
  if( ++top >= MAX ) {
    printf( "stack overflow\n" );
    exit( -1 );
  }
  s[top] = v;
}

int pop() {
  if( top < 0 ) {
    printf( "stack underflow\n" );
    exit( -1 );
  }
  return s[top--];
}
```

```
main( int ac, char *av[] ) {
  int i;
  for( i = 1;  i < ac;  i++ ) {
    if( av[i][0] == ']' ) pop();
    else
    if( av[i][0] == '[' ) push( 0 );
    else {
      printf( "unexpected input\n" );
      exit( -1 );
    }
  }
  if( top == -1 )
    printf( "balanced\n" );
  else
    printf( "not balanced\n" );
}
```

```
$ a.out [ [ ]
not balanced

$ a.out [ [ [ ] [ [ ] ] ] ]
balanced

$ a.out [
not balanced

$ a.out ]
stack underflow
```

Queues

- Mathematically, a queue is a (finite) sequence that provides the first-in/first-out insertion/deletion discipline.
- A queue is supposed to be used with two operations: $enqueue(v)$, $dequeue()$.
- It can be easily implemented using an array and two variables (for head and tail) if the maximum queue size (or maximum queue size plus 1) can be determined in advance.

Queues by Arrays: Example

```
#define MAX     3

int q[MAX], head, tail = 0;

init() { head = tail = 0; }

is_empty() { return head == tail; }
```

```
enqueue( int v ) {
  if( (tail+1)%MAX == head ) {
    printf( "overflow\n" );
    exit( -1 );
  }
  q[tail++] = v;
  if( tail >= MAX ) tail = 0;
}
```

```
int dequeue() {
  int v;
  if( head == tail ) {
    printf( "underflow\n" );
    exit( -1 );
  }
  v = q[head++];
  if( head >= MAX ) head = 0;
  return v;
}
```

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

  for( i = 1;  i < ac;  i++ ) {
    if( (v = atoi(av[i])) < 0 )
      printf( "%d\n", dequeue() );
    else
      enqueue( v );
  }
}
```

```
$ a.out 1 2 -1 3 -1 4 -1 -1 5 6 7
1
2
3
4
overflow

$ a.out 1 2 -1 3 -1 4 -1 -1 5 6
1
2
3
4
```

```
$ a.out 1 2 -1 3 -1 4 -1 -1 5 6 -1 -1
1
2
3
4
5
6
```

Graphs

- Mathematically, a (undirected) graph G consists of a vertex set V and an edge set E:

$$G = (V, E).$$

- For example, the following is a (simple) graph:

$$G = (\{1, 2, 3, 4, 5, 6\}, \{\{1, 2\}, \{1, 3\}, \{3, 4\}, \{5, 6\}\}).$$

- Intuitively, a graph is simply some points with lines connecting some of the points.

Trees

- A tree is a connected acyclic (undirected) graph.
- A graph is connected if one can move from one vertex v of the graph to any other vertex (except v) of the graph through the edges of the graph.
- A graph is acyclic (cycle free) if one cannot move from one vertex to itself through one or more edges of the graph.
- Examples.

Rooted Trees

- To impose some hierarchy on the vertices of a tree, we may designate a vertex of a tree as the root of the tree.
- A tree of n vertices becomes n different rooted trees (if children of a node are not ordered).
- Examples.

Intuitive Rooted Tree Terminology

- The root, internal nodes, and leaves.
- Ancestors and descendants.
- Parents and children.
- Siblings.
- Levels and the height.

Binary Trees

- A binary tree is a rooted tree such that every internal node has either 1 child or 2 children.
- A child is either a left child or a right child.
- Left/right distinction is significant. For example, there are only 1 rooted tree with 2 vertices but there are 2 binary trees with 2 vertcies due to the distinction of left and right child.

Binary Search Trees, Binary Tree Traversal: Example

```
struct node {
  int key;
  struct node *left, *rite;
};

struct node *bst;
```

```
insert( struct node **t, int v ) {
  if( *t == 0 ) {
    *t = (struct node *) calloc( 1, sizeof(struct node) );
    (*t)->key = v;
  } else {
    if( (*t)->key > v )
      insert( &((*t)->left), v );
    else
      insert( &((*t)->rite), v );
  }
}
```

```
inorder( struct node *t ) {
  if( t != 0 ) {
    inorder( t->left );
    printf( "%d\n", t->key );
    inorder( t->rite );
  }
}
```

```
preorder( struct node *t ) {
  if( t != 0 ) {
    printf( "%d\n", t->key );
    preorder( t->left );
    preorder( t->rite );
  }
}
```

```
postorder( struct node *t ) {
  if( t != 0 ) {
    postorder( t->left );
    postorder( t->rite );
    printf( "%d\n", t->key );
  }
}
```

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

  for( i = 1;  i < ac;  i++ ) insert( &bst, atoi(av[i]) );
  printf( "inorder:\n" );  inorder( bst );
  printf( "preorder:\n" );  preorder( bst );
  printf( "postorder:\n" );  postorder( bst );
}
```

```
$ a.out 3 5 6 4 1 2 0
inorder:
0
1
2
3
4
5
6
preorder:
3
1
0
2
5
4
```

```
6
postorder:
0
2
1
4
6
5
3
```

Disjoint Sets

- Very often we have to represent a collection of disjoint sets and perform union and find operations efficiently.
- This can be done easily with the disjoint set data structure.
- Examples.

Disjoint Sets as an Array

- A universal set containing n elements can be represented by a size n array.
- Members of a set can be represented as a tree.
- Thus set membership can be represented by treating this size n array as a parent array.
- That is,

$$parent[i] == parent[j]$$

means elements i and j are in the same set.

Disjoint Sets Operations

```
#define N       10
int parent[N];

init() { int i;  for( i=0;  i<N;  i++ ) parent[i] = -1; }

makeset( int i ) { parent[i] = i; }

int findset( int i ) {
  while( i != parent[i] ) i = parent[i];
  return i;
}

union( int i, int j ) { parent[findset(i)] = findset(j); }
```

Disjoint Sets: Example

```
#define N       10

int parent[N];

init() {
  int i;

  for( i = 0;  i < N;  i++ ) parent[i] = -1;
}
```

```
makeset( int i ) {
  parent[i] = i;
}

int findset( int i ) {
  while( i != parent[i] )
    i = parent[i];

  return i;
}

union1( int i, int j ) {
  parent[ findset(i) ] = findset(j);
}
```

```
main() {
  int i;

  init();  // not really necessary
  for( i = 0;  i < N;  i++ ) makeset(i);
  union1( 1, 4 );
  union1( 3, 5 );
  union1( 4, 2 );
  union1( 1, 5 );
  for( i = 0;  i < N;  i++ )
    printf( "%d %d %d\n", i, parent[i], findset(i) );
}
```

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

Path Compression

```
int findset( int i ) { // shorten the path traversed
  int root, j;

  root = i;
  while( root != parent[root] ) root = parent[root];

  j = parent[i];
  while( j != root ) {
    parent[i] = root;  i = j;  j = parent[i];
  }

  return root;
}
```

Union by Rank

```
/*
  Attach a short tree to a tall tree.

  Use a rank matrix to hint at the height of a tree.
*/

makeset( int i ) {
  parent[i] = i;
  rank[i] = 0;
}
```

```
union2( int i, int j ) {
  i = findset( i );
  j = findset( j );
  if( i == j ) return;
  if( rank[i] < rank[j] ) parent[i] = j;
  else
  if( rank[i] > rank[j] ) parent[j] = i;
  else {
    parent[i] = j;
    rank[j]++;
  }
}
```

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

Priority Queues

- A priority queue provides the "highest-priority-out-first" discipline.
- A priority queue can be implemented using an unsorted array with constant time enqueue and linear time dequeue.
- A priority queue can be implemented using a sorted array with linear time enqueue and constant time dequeue.
- By using a heap, a priority queue can be implemented with $\log$ time enqueue and $\log$ time dequeue.

Heap Structures

- A heap structure is a binary tree in which all levels except the last have the maximum number of nodes. On the last level, leaves are attached from left to right.

- If the nodes of a heap structure of height h is stored in an array $a[\ ]$ (with first index 1) from level 0 to level h, and within a level from left to right, then the children of node $a[i]$ are $a[2i]$ and $a[2i+1]$, and the parent of node $a[i]$ is $a[\lfloor \frac{i}{2} \rfloor]$.

Binary Minheap

- A binary minheap is a heap structure in which values are assigned to the nodes such that the value of each node is less than or equal to the values of its children (if any).

- if array $h[1..n]$ stores a binary minheap, then the smallest element can be found in constant time

```
int heap_min() {
  return h[1];
}
```

Root + Left Minheap + Right Minheap $\longrightarrow$ Heap

- If the left subtree and the right subtree of the root of a heap structure are binary minheaps, then the heap structure can be made into a binary minheap with the $siftdown()$ algorithm.

$siftdown()$

```
siftdown( int i ) { //the tree starts at h[i], ends at h[n]
  int temp, child;

  temp = h[i];
  while( 2*i <= n ) {
    child = 2*i;
    if( child < n && h[child+1] < h[child] ) child++;
    if( h[child] < temp ) v[i] = v[child]; else break;
    i = child;
  }
  h[i] = temp;
}
```

Priority Dequeue: $delete()$

```
int heap_delete() {
  h[1] = h[n--];
  siftdown( 1 );
}
```

Priority Enqueue: $heap\_insert()$

```
heap_insert( int v ) {
  int i;

  i = ++n;
  while( i > 1 && v < h[i/2] ) {
    h[i] = h[i/2];
    i = i/2;
  }
  h[i] = v;
}
```

Array $\rightarrow$ Minheap: $heapify()$

```
heapify() {
  for( i = n/2;  i >= 1;  i-- ) siftdown( i );
}
```