```
CS32 Final Notes

Include things that I have written for this class


==========================================================================================
                                  Data Structures
==========================================================================================
 - LinkedLists
 - Stacks
 - Queues
 - Priority Queues
 - Vectors
 - --Sorted v. Unsorted
 - Sets
 - Unordered Sets
 - Unordered Multiset
 - Hash Table (map)
 - Unordered Map
 - Unordered Multimap
 - Binary Search Trees (w/ w/o balancing)
 - Heaps
 - --Binary Heaps: Minheap and Maxheap
                                ------------------------
                                |      Linked Lists    |
                                ========================

C++ STL:
#include <list>
list<itemType> l;

ACCESSORS
    l.size();   Return current number of elements.  =       =       =   =       =   O(1)
    l.empty();  Return true if list is empty.   -       -       -       -       -   O(1)
    l.begin();  Return bidirectional iterator to start. =   =       =   =   =   =   O(1)
    l.end();    Return bidirectional iterator to end. -     -   -   -       -       O(1)
    l.front();  Return the first element.   -       -       -       -       -     - O(1)
    l.back();   Return the last element.        -       -       -       -       -   O(1)

MODIFIERS
    l.push_front(value);    Add value to front.         =           =       =       O(1)
    l.push_back(value);     Add value to end.   -           -           -       -   O(1)
    l.insert(iterator, value);  Insert value after position indexed by iterator.    O(1)
    l.pop_front();  Remove value from front. =      =           =   =       =       O(1)
    l.pop_back();   Remove value from end.      -       -       -       -       -   O(1)
    l.erase(iterator);  Erase value indexed by iterator.    =       =       =       O(1)
    l.erase(begin, end);    Erase the elements from begin to end.       -       -   O(1)
    l.remove(value);    Remove all occurrences of value.    -       -       -       O(n)
    l.remove_if(test);  Remove all element that satisfy test.   =           =       O(n)
    l.reverse();    Reverse the list.   =-      =-      =-      =-      =-      =-  O(n)
    l.sort();   Sort the list.      =-      =-          =-                      =-  O(n log n)
    l.sort(comparison);     Sort with comparison function.                          O(n logn)
    l.merge(l2);    Merge sorted lists.                                             O(n)



//ADVANTAGES OF LINKED LISTS
    Linked Lists make it much easier to insert things into arrays/lists. Arrays have to shift all the objects down/deal with them in some way, but linked lists ya kinda just stick em in:
        -add an item
        -adjust some pointers
    Removing things from a linked lists:
        -adjust the pointers
        -delete the node

//DISADVANTAGE OF LINKED LISTS:
    Don't have immediate access to an arbitrary element of the list. The only way to
```

```
    get to an item is to follow the chain of pointers.


struct Node //nodes form the building blocks of linked lists
{
    int data; //values of the list are stored in this->data
    Node* next; //the definition for Node contains a pointer to a node
}
Node* head; //the head of the linked list. This can act as a dummy node or the actual first element

//Linked List Advice
    -draw pictures!
    -set a node's pointer members before changing other pointers
    -order matters
    -any time you write p->, make sure:
        --p has previously been given a value
        --p is not nullptr

class LinkedList //a sample linked list
{
public:
    LinkedList();
    void addToFront(string v);
    void addToRear(string v);
    void deleteItem(string v);
    bool findItem(string v);
    void printItems();
    ~LinkedList();
private:
    Node* head;
    struct Node //nodes can act as a private member struct
    {
        string value;
        Node *next;
        Node *prev;
    }
}

//allocating new nodes
    Node *p = new Node;
    Node *q = new Node;

//change/access node p's value
    p->value = "blah";
    cout << p->value;

//make p link to another node at address q
    p->next = q;

//get the address of the node after p
    Node *r = p->next

//make node q a terminal node
    q->next = nullptr;

delete p;
delete q;

void deleteItem(string v) //function to delete an arbitrary element of a singly linked list
{
    Node *p = head;
    while (p != nullptr)
    {
        if (p->next != nullptr && p->next->value == v)
            break; //if you find the value, break - p points to the node above

        p = p->next; //don't find the value, go down one
```

```
    }
    if (p != nullptr) //when you find the value, delete it
    {
        Node *killMe = p->next;
        p->next = killMe->next;
        delete killMe;
    }
}

bool search(Node* head, string v) //function to find a string in a singly linked list
{
    for (node *p = head; p != nullptr; p = p->next)
    {
        if (p->value == v)
            return true;
    }
    return false;
}

:/Doubly linked list:
    //the next + previous pointers contained in each nodes
    //this->next points to the next item, this->prev points to the previous item
:/Circular doubly linked list
    //same as a DLL, but the last node in the array points to the first one.

:/Dummy Node == the first node
    //value isn't part of the list, and it's not initialized
    //first item of the list is at head->m_next;, last one's at head->m_prev

int cmpr(Node* head, int* arr, int arr_size) //function to compare an array and linked list and return the number of consecutive elements they share
{
    Node* p = head->next;
    for (int sim = 0; sim < arr_size; sim++)
    {
        if ((p->value == nullptr) || (arr[sim] != p->value))
            break;
        p = p->next;
    }
    return sim - 1;
}



                                --------------------
                                |      STACKS      |
                                ====================

,/Big O: Everything is functionally constant time

//items are always added to and removed from the TOP of the stack
//ALL interfaces allow these interactions with stacks:
    ,/create an empty stack - stack<ItemType> s;

    ,/push an item onto the stack - s.push(item);

    ,/pop an item off the stack - s.pop();

    ,/look at the top item of the stack - s.top();

    ,/is the stack empty? - s.empty();
//some interfaces let you do these:
    how many items are on the stack?
    look at any item on the stack

#include <stack>
using namespace std;

//Maze function written for HW3, using stacks
```

```
class Coord //coord class, used to store the coordinates checked in the maze
{
public:
    Coord(int rr, int cc) : m_r(rr), m_c(cc) {}
    int r() const { return m_r; }
    int c() const { return m_c; }
private:
    int m_r;
    int m_c;
};

bool pathExists(char maze[][10], int sr, int sc, int er, int ec)
{
    stack<Coord> coordStack; //The working stack of Coords
    Coord start(sr, sc); //starting position of the player
    Coord end(er, ec); //desired spot to reach

    coordStack.push(start);
    maze[start.r()][start.c()] = '#'; //set checked locations to a value that won't get checked again

    while (!coordStack.empty())
    {
        Coord loc = coordStack.top();
        coordStack.pop(); //get rid of the item we're looking at from the stack - don't need it again

        if ((loc.r() == end.r()) && (loc.c() == end.c())) //if it finds it, returns true
            return true;

        if (maze[loc.r() + SOUTH][loc.c()] == '.')
        //code puts all empty spaces around the current location onto the stack, and eventually checks them all, resulting in every accessible maze location being checked
        {
            Coord newloc(loc.r() + SOUTH, loc.c());
            maze[newloc.r()][newloc.c()] = '#';
            coordStack.push(newloc);
        }
        if (maze[loc.r()][loc.c() + WEST] == '.')
        {
            Coord newloc(loc.r(), loc.c() + WEST);
            maze[newloc.r()][newloc.c()] = '#';
            coordStack.push(newloc);
        }
        if (maze[loc.r() + NORTH][loc.c()] == '.')
        {
            Coord newloc(loc.r() + NORTH, loc.c());
            maze[newloc.r()][newloc.c()] = '#';
            coordStack.push(newloc);
        }
        if (maze[loc.r()][loc.c() + EAST] == '.')
        {
            Coord newloc(loc.r(), loc.c() + EAST);
            maze[newloc.r()][newloc.c()] = '#';
            coordStack.push(newloc);
        }
    }
    return false; //reaches here after checking every available coordinate, yet failing to find the desired one
}
```

```
                                --------------------
                                |      QUEUES      |
                                ====================
```

,/Big O: Everything is functionally constant time

```
//like a stack but backwards:
```

```
//items are always added to and removed from the BACK of the queue

//all interfaces allow these interactions with queues
    ,/create an empty queue - queue<ItemType> q;

    ,/enqueue an item onto the queue  - q.push();

    ,/dequeue an item from the queue - q.pop();

    ,/look at the front item of the queue - q.front();

    ,/is the queue empty? - q.empty();
//some interfaces let you do these
    how many items are in the queue?
    look at the back item of the queue
    look at any item in the queue

#include <queue>

Works VERY similarly to stacks, but in reverse order. Oftentimes one will be better than the other, depending on what you want your code to do. Queues add to the back, so don't check the most recent thing added until N time has passed, where N is the number of items on in the queue at the time of addition.

---------------------------
      Priority Queues
---------------------------
Similar to a queue, but the first element is always the larger than all others
Implemented as a ,/container adaptor - uses another type of structure as its underlying container, and provides functions to access specific elements
the priority queues basic operations will be implemented using the underlying container's operations.
If no underlying container is specified, a vector is used

Priority queues are neither first-in-first-out nor last-in-first-out. You push objects onto the priority queue. The top element is always the "biggest" of the elements currently in the priority queue. Biggest is determined by the comparison predicate you give the priority queue constructor.

    If that predicate is a "less than" type predicate, then biggest means largest.

    If it is a "greater than" type predicate, then biggest means smallest.

#include <queue>
priority_queue<T, container<T>, comparison<T> > q;

//Member functions of priority_queue
    q.top();    Return the "biggest" element.               O(1)
    q.size();   Return current number of elements.          O(1)
    q.empty();  Return true if priority queue is empty.     O(1)
    q.push(value);  Add value to priority queue.            O(log n)
    q.pop();    Remove biggest value.                       O(log n)

                                ----------------------------
                                    |      VECTORS     |
                                ============================
    "bekutodajzu" - Hiro in Ling 102

Vectors are sequence containers representing arrays that can change in size.
"Arrays that can change in size"

Can be accessed using regularly offset pointers to their elements, just like arrays
Unlike arrays, the size can change dynamically

//Creating a vector
#include <vector>

vector<itemType> v; //create a vector of itemType named v
```

```
-------------------------
 Public Member Functions
=========================

ITERATORS - all O(1)
    begin - Return iterator to beginning (public member function )
    end - Return iterator to end (public member function )
    rbegin - Return reverse iterator to reverse beginning (public member function )
    rend - Return reverse iterator to reverse end (public member function )

CAPACITY
    O(1) size - Return size (public member function )
    O(1) max_size - Return maximum size (public member function )
    O(N) resize - Change size (public member function )
    O(1) capacity - Return size of allocated storage capacity (public member function )
    O( ) empty - Test whether vector is empty (public member function )
    O(N) reserve - Request a change in capacity (public member function ) N is vector size
    O(N) shrink_to_fit - Shrink to fit (public member function ) N is container size

ELEMENT ACCESS - all O(1)
    operator[] - access element
    at - access element
    front - get first element
    back - get last element

MODIFIERS
    O(N) assign - Assign vector content (public member function )
    O(1) push_back - Add element at the end (public member function )
    O(1) pop_back - Delete last element (public member function )
    O(N) insert(position, value) - Insert elements (public member function )
    O(N) erase - Erase elements (public member function )
    O(1) swap - Swap content (public member function )
    O(N) clear - Clear content (public member function )



Iterating through a vector:

for (std::vector<itemType>::iterator iter = v.begin(); iter != v.end(); iter++)
{
    if (*iter == desiredValue)
        break;
}


funky cases with erase() - calling erase dereferences any iterators used to go through the vector
can be fixed by passing a modified value: v.erase(i--);
or by setting the iterator to the return of the function: i = v.erase(i);


Searching through an unordered vector: N time
Searching through an ordered vector: log(N) time (possible to use binary search, since items can be accessed in constant time)



                            ----------------------------
                                |      Sets          |
                            ============================

Sets store objects, automatically keep them sorted to allow for easy access
Sets can only contain unique elements - no duplicates!

<multiset>s

#include <set>
```

```
Constructors
    set< type, compare > s;     Make an empty set. compare should be a binary predicate for ordering the set. It's optional and will default to a function that uses operator<.     O(1)
    set< type, compare > s(begin, end);     Make a set and copy the values from begin to end.    O(n log n)

Accessors
    s.find(key)     Return an iterator pointing to an occurrence of key in s, or s.end() if key is not in s.    O(log n)
    s.lower_bound(key)  Return an iterator pointing to the first occurrence of an item in s not less than key, or s.end() if no such item is found.    O(log n)
    s.upper_bound(key)  Return an iterator pointing to the first occurrence of an item greater than key in s, or s.end() if no such item is found.    O(log n)
    s.equal_range(key)  Returns pair<lower_bound(key), upper_bound(key)>.           O(log n)
    s.count(key)    Returns the number of items equal to key in s.                  O(log n)
    s.size();   Return current number of elements.                                  O(1)
    s.empty();  Return true if set is empty.                                        O(1)
    s.begin()   Return an iterator pointing to the first element.                   O(1)
    s.end()     Return an iterator pointing one past the last element.              O(1)

Modifiers
    s.insert(iterator, key)     Inserts key into s. iterator is taken as a "hint" but key will go in the correct position no matter what. Returns an iterator pointing to where key went.              O(log n)
    s.insert(key)   Inserts key into s and returns a pair<iterator, bool>, where iterator is where key went and bool is true if key was actually inserted, i.e., was not already in the set.         O(log n)





                                 ----------------------------
                                 |       Maps/Multimap      |
                                 ============================

Maps are kinda like generalized vectors. They allow map[key] = value for any kind of key, not just integers. Maps are often called associative tables in other languages, and are incredibly useful. They're even useful when the keys are integers, if you have very sparse arrays, i.e., arrays where almost all elements are one value, usually 0.

Maps are implemented with balanced binary search trees, typically red-black trees. Thus, they provide logarithmic storage and retrieval times. Because they use search trees, maps need a comparison predicate to sort the keys. operator<() will be used by default if none is specified a construction time.

Maps store <key, value> pair's. That's what map iterators will return when dereferenced. To get the value pointed to by an iterator, you need to say

(*mapIter).second

Usually though you can just use map[key] to get the value directly.

    Warning: map[key] creates a dummy entry for key if one wasn't in the map before. Use map.find(key) if you don't want this to happen.

multimaps are like map except that they allow duplicate keys. map[key] is not defined for multimaps. Instead you use lower_bound() and upper_bound(), or equal_range(), to get the iterators for the beginning and end of the range of values stored for the key. To insert a new entry, use map.insert(pair<key_type, value_type>(key, value)).


#include <map>
```

```
Constructors
    map< key_type, value_type, key_compare > m;     Make an empty map. key_compare should be a binary predicate for ordering the keys. It's optional and will default to a function that uses operator<.    O(1)
    map< key_type, value_type, key_compare > m(begin, end);     Make a map and copy the values from begin to end.   O(n log n)

Accessors
    m[key]  Return the value stored for key. This adds a default value if key not in map .   O(log n)
    m.find(key)     Return an iterator pointing to a key-value pair, or m.end() if key is not in map.   O(log n)
    m.lower_bound(key)  Return an iterator pointing to the first pair containing key, or m.end() if key is not in map.  O(log n)
    m.upper_bound(key)  Return an iterator pointing one past the last pair containing key, or m.end() if key is not in map.     O(log n)
    m.equal_range(key)  Return a pair containing the lower and upper bounds for key. This may be more efficient than calling those functions separately.    O(log n)
    m.size();   Return current number of elements.  O(1)
    m.empty();  Return true if map is empty.    O(1)
    m.begin()   Return an iterator pointing to the first pair.  O(1)
    m.end()     Return an iterator pointing one past the last pair.     O(1)

Modifiers
    m[key] = value;     Store value under key in map.   O(log n)
    m.insert(pair)  Inserts the <key, value> pair into the map. Equivalent to the above operation.  O(log n)




                            ----------------------------
                            |       Hash Tables        |
                            ============================

Hashing is an improvement over Direct Access Table. The idea is to use hash function that converts a given phone number or any other key to a smaller number and uses the small number as index in a table called hash table.

Hash Function: A function that converts a given big phone number to a small practical integer value. The mapped integer value is used as an index in hash table. In simple terms, a hash function maps a big number or string to a small integer that can be used as index in hash table.
    A good hash function should have following properties
        1) Efficiently computable.
        2) Should uniformly distribute the keys (Each table position equally likely for each key)

For example for phone numbers a bad hash function is to take first three digits. A better function is consider last three digits. Please note that this may not be the best hash function. There may be better ways.

Hash Table: An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry.

Collision Handling: Since a hash function gets us a small number for a big key, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique. (say, chaining)

class Hash //a hash class i made for project four
{
private:
    int BUCKET;
public:
    //this probably isn't good practice, but works exceedingly well for this setup
    list<Node> *table;

```

```
    // Constructor
    Hash(int b) : BUCKET(b) { table = new list<Node>[BUCKET]; }


    ~Hash() { delete[] table; }


    unsigned int hashFunction(Node item) //node-parameter overload of hashFunction
    {
        string word = item.value;
        return hashFunction(word);
    }

    unsigned int hashFunction(string word)  //the location of items in the string is dependent on the STRING in the node - not the offset. Makes searching faster
    {
        int sum = 0;
        for (unsigned int k = 0; k < word.length(); k++)
            sum = sum + int(word[k]);
        return  sum % BUCKET;
    }

    // insert a key into hash table
    void insertItem(Node key)
    {
        int index = hashFunction(key);
        string value = key.value;
        table[index].push_back(key);
    }

    int checkValue(string val) //1-argument overload of checkValue
    {
        int index = hashFunction(val);

        return checkValue(val, index);
    }
};
```

```
----------------------------
|    Binary Search Trees   |
============================
```

Binary Search Tree, is a node-based binary tree data structure which has the following properties:

    The left subtree of a node contains only nodes with keys lesser than the node’s key.
    The right subtree of a node contains only nodes with keys greater than the node’s key.
    The left and right subtree each must also be a binary search tree.
    There must be no duplicate nodes.

To search a given key in Binary Search Tree, we first compare it with root, if the key is present at root, we return root. If key is greater than root’s key, we recur for right subtree of root node. Otherwise we recur for left subtree.

```
         100                             100
        /   \        Insert 40          /    \
      20     500    --------->        20     500
     /  \                            /  \
    10   30                         10   30
                                           \
                                            40
```

```
// C program to demonstrate insert operation in binary search tree
#include<stdio.h>
#include<stdlib.h>

struct node
```

```c
{
    int key;
    struct node *left, *right;
};

// A utility function to create a new BST node
struct node *newNode(int item)
{
    struct node *temp =  (struct node *)malloc(sizeof(struct node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to do inorder traversal of BST
void inorder(struct node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("%d \n", root->key);
        inorder(root->right);
    }
}
struct node* insert(struct node* node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL) return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left  = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    /* return the (unchanged) node pointer */
    return node;
}
```

Heaps

A complete binary tree is a binary tree where all levels are filled, except possibly the bottom level, which is filled from left to right

A (max) heap is a complete binary tree in which the value at every note is >= all the values in that node's subtrees

A (min) heap is a complete binary tree in which the value at every node is <= all the values in that node's subtrees

```text
Remove a node:
    Delete the root
    make the bottom rightmost item the new root
    trickle down until it's in its proper place
```

A Binary Heap is a Complete Binary Tree. A binary heap is typically represented as array. The representation is done as:

```text
    The root element will be at Arr[0].
    Below table shows indexes of other nodes for the ith node, i.e., Arr[i]:
    Arr[(i-1)/2]    Returns the parent node
    Arr[(2*i)+1]    Returns the left child node
    Arr[(2*i)+2]    Returns the right child node

    The traversal method use to achieve Array representation is Level Order
```

```
class MinHeap
{
    int *harr; // pointer to array of elements in heap
    int capacity; // maximum possible size of min heap
    int heap_size; // Current number of elements in min heap
public:
    // Constructor
    MinHeap(int capacity);

    // to heapify a subtree with the root at given index
    void MinHeapify(int );

    int parent(int i) { return (i-1)/2; }

    // to get index of left child of node at index i
    int left(int i) { return (2*i + 1); }

    // to get index of right child of node at index i
    int right(int i) { return (2*i + 2); }

    // to extract the root which is the minimum element
    int extractMin();

    // Decreases key value of key at index i to new_val
    void decreaseKey(int i, int new_val);

    // Returns the minimum key (key at root) from min heap
    int getMin() { return harr[0]; }

    // Deletes a key stored at index i
    void deleteKey(int i);

    // Inserts a new key 'k'
    void insertKey(int k);
};

// Constructor: Builds a heap from a given array a[] of given size
MinHeap::MinHeap(int cap)
{
    heap_size = 0;
    capacity = cap;
    harr = new int[cap];
}

// Inserts a new key 'k'
void MinHeap::insertKey(int k)
{
    if (heap_size == capacity)
    {
        cout << "\nOverflow: Could not insertKey\n";
        return;
    }

    // First insert the new key at the end
    heap_size++;
    int i = heap_size - 1;
    harr[i] = k;

    // Fix the min heap property if it is violated
    while (i != 0 && harr[parent(i)] > harr[i])
    {
       swap(&harr[i], &harr[parent(i)]);
       i = parent(i);
    }
}

// Decreases value of key at index 'i' to new_val.  It is assumed that
// new_val is smaller than harr[i].
```

```
void MinHeap::decreaseKey(int i, int new_val)
{
    harr[i] = new_val;
    while (i != 0 && harr[parent(i)] > harr[i])
    {
       swap(&harr[i], &harr[parent(i)]);
       i = parent(i);
    }
}

// Method to remove minimum element (or root) from min heap
int MinHeap::extractMin()
{
    if (heap_size <= 0)
        return INT_MAX;
    if (heap_size == 1)
    {
        heap_size--;
        return harr[0];
    }

    // Store the minimum value, and remove it from heap
    int root = harr[0];
    harr[0] = harr[heap_size-1];
    heap_size--;
    MinHeapify(0);

    return root;
}


// This function deletes key at index i. It first reduced value to minus
// infinite, then calls extractMin()
void MinHeap::deleteKey(int i)
{
    decreaseKey(i, INT_MIN);
    extractMin();
}

// A recursive method to heapify a subtree with the root at given index
// This method assumes that the subtrees are already heapified
void MinHeap::MinHeapify(int i)
{
    int l = left(i);
    int r = right(i);
    int smallest = i;
    if (l < heap_size && harr[l] < harr[i])
        smallest = l;
    if (r < heap_size && harr[r] < harr[smallest])
        smallest = r;
    if (smallest != i)
    {
        swap(&harr[i], &harr[smallest]);
        MinHeapify(smallest);
    }
}

// A utility function to swap two elements
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}


================================================================================
                Sorting Algorithms
================================================================================
```

,/Selection Sort

Suppose A is an array of N values. We want to sort A in ascending order. That is, A[0] should be the smallest and A[N-1] should be the largest.

The idea of Selection Sort is that we repeatedly find the smallest element in the unsorted part of the array and swap it with the first element in the unsorted part of the array.

```
For I = 0 to N-1 do:
  Smallsub = I
  For J = I + 1 to N-1 do:
    If A(J) < A(Smallsub)
      Smallsub = J
    End-If
  End-For
  Temp = A(I)
  A(I) = A(Smallsub)
  A(Smallsub) = Temp
End-For
```

A refinement of the above pseudocode would be to avoid swapping an element with itself.

An alternate way to sort in ascending order is to find the largest value and swap with the last element in the unsorted part of the array.

Selection Sort does roughly N**2 / 2 comparisons and does N swaps.

,/Insertion Sort

Insertion Sort

Suppose A is an array of N values. We want to sort A in ascending order.

Insertion Sort is an algorithm to do this as follows: We traverse the array and insert each element into the sorted part of the list where it belongs. This usually involves pushing down the larger elements in the sorted part.

```
For I = 1 to N-1
  J = I
  Do while (J > 0) and (A(J) < A(J - 1)
    Temp = A(J)
    A(J) = A(J - 1)
    A(J - 1) = Temp
    J = J - 1
  End-Do
End-For
```

Insertion Sort does roughly N**2 / 2 comparisons and does up to N - 1 swaps.

,/Bubble Sort - the worst one

Suppose A is an array of N values. We want to sort A in ascending order.

Bubble Sort is a simple-minded algorithm based on the idea that we look at the list, and wherever we find two consecutive elements out of order, we swap them. We do this as follows: We repeatedly traverse the unsorted part of the array, comparing consecutive elements, and we interchange them when they are out of order. The name of the algorithm refers to the fact that the largest element "sinks" to the bottom and the smaller elements "float" to the top.

```
For I = 0 to N - 2
  For J = 0 to N - 2
    If (A(J) > A(J + 1)
      Temp = A(J)
      A(J) = A(J + 1)
      A(J + 1) = Temp
    End-If
  End-For
```

```
            End-For

        Bubble Sort does roughly N**2 / 2 comparisons and does up to N**2 / 2 swaps.

    ,/MergeSort(arr[], l,  r)
    If r > l
         1. Find the middle point to divide the array into two halves:
                 middle m = (l+r)/2
         2. Call mergeSort for first half:
                 Call mergeSort(arr, l, m)
         3. Call mergeSort for second half:
                 Call mergeSort(arr, m+1, r)
         4. Merge the two halves sorted in step 2 and 3:
                 Call merge(arr, l, m, r)


    ,/Quicksort
    Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

    partitioning:
        Step 1 − Choose the highest index value has pivot
        Step 2 − Take two variables to point left and right of the list excluding pivot
        Step 3 − left points to the low index
        Step 4 − right points to the high
        Step 5 − while value at left is less than pivot move right
        Step 6 − while value at right is greater than pivot move left
        Step 7 − if both step 5 and step 6 does not match swap left and right
        Step 8 − if left ≥ right, the point where they met is new pivot

    Actually Sorting
        Step 1 − Make the right-most index value pivot
        Step 2 − partition the array using pivot value
        Step 3 − quicksort left partition recursively
        Step 4 − quicksort right partition recursively

    /* low  --> Starting index,  high  --> Ending index */
    quickSort(arr[], low, high)
    {
        if (low < high)
        {
            /* pi is partitioning index, arr[pi] is now
               at right place */
            pi = partition(arr, low, high);

            quickSort(arr, low, pi - 1);  // Before pi
            quickSort(arr, pi + 1, high); // After pi
        }
    }



    ,/Heapsort

    Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

    Why array based representation for Binary Heap?
        Since a Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index I, the left child can be calculated by 2 * I + 1 and right child by 2 * I + 2 (assuming the indexing starts at 0).

    Heap Sort Algorithm for sorting in increasing order:
        1. Build a max heap from the input data.
        2. At this point, the largest item is stored at the root of the heap. Replace it
```

with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
3. Repeat above steps while size of heap is greater than 1.

How to build the heap?
Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom up order.

```
    //make an array into a heap
void heapify(int arr[], int n, int i)
{
    int largest = i; // Initialize largest as root
    int l = 2*i + 1; // left = 2*i + 1
    int r = 2*i + 2; // right = 2*i + 2

    // If left child is larger than root
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i)
    {
        swap(arr[i], arr[largest]);

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

// main function to do heap sort
void heapSort(int arr[], int n)
{
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // One by one extract an element from heap
    for (int i=n-1; i>=0; i--)
    {
        // Move current root to end
        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}
```

==

Class Structure

==

STEPS OF CONSTRUCTING OBJECTS: THE FIRST STEP
1. Construct the Base Part
2. Construct the Data Members
3. Execute the body of the constructor

STEPS OF DESTRUCTION
1. Execute the body of the destructor
2. Destroy the data members
3. Destroy the base part //new third step

member variables are constructed before the constructor is called -> inside out
member variables are destructed after the desctructor is called -> outside in

if member variables are dynamically allocated, they will not be deleted automatically when/after the destructor is called - gotta add that to the destructor yourself to get it to happen

default copy constructor just copies member variables from the existing object to the new one

you need your own copy constructor when:

- the object contains a pointer to something in memory. otherwise, the pointers will point to the same block of memory, instead of just being a copy. This means that when you modify one the other changes too.
  (^ dynamic memory allocation)

Member Initialization Lists : value(passedvalue), okay(passedvalue2)

===========================

Templates

===========================

used to create functions that can be called with lots of different types of data

```cpp
template <typename T>
class Sequence
{
  public:
    Sequence();           // Create an empty sequence (i.e., one whose size() is 0).
    bool empty() const;  // Return true if the sequence is empty, otherwise false.
    int size() const;    // Return the number of items in the sequence.

    int insert(int pos, const T& value);
      // Insert value into the sequence so that it becomes the item at
      // position pos.  The original item at position pos and those that
      // follow it end up at positions one higher than they were at before.
      // Return pos if 0 <= pos <= size() and the value could be
      // inserted.  (It might not be, if the sequence has a fixed capacity,
      // e.g., because it's implemented using a fixed-size array.)  Otherwise,
      // leave the sequence unchanged and return -1.  Notice that
      // if pos is equal to size(), the value is inserted at the end.
        ~Sequence();
    Sequence(const Sequence& other);
    Sequence& operator=(const Sequence& rhs);

  private:
      // Representation:
      //   a circular doubly-linked list with a dummy node.
      //   m_head points to the dummy node.
      //   m_head->m_prev->m_next == m_head and m_head->m_next->m_prev == m_head
      //   m_size == 0  iff  m_head->m_next == m_head->m_prev == m_head
      //   if m_size > 0
      //       m_head->next points to the node at position 0.
      //       m_head->prev points to the node at position m_size-1.

    struct Node
    {
        T m_value;
        Node*    m_next;
        Node*    m_prev;
    };

    Node* m_head;
    int   m_size;

    void createEmpty();
      // Create an empty list.  (Should be called only by constructors.)

    void insertBefore(Node* p, const T& value);
      // Insert value in a new Node before Node p, incrementing m_size.

    Node* doErase(Node* p);
```

```
        // Remove the Node p, decrementing m_size.  Return the Node that
        // followed p.

    Node* nodeAtPos(int pos) const;
        // Return pointer to Node at position pos.  If pos == m_size, return
        // m_head.  (Will be called only when 0 <= pos <= size().)
};

// Template implementations

template <class T>
int Sequence<T>::size() const
{
    return m_size;
}


template <class T>
bool Sequence<T>::empty() const
{
    return size() == 0;
}

template <class T>
Sequence<T>::~Sequence()
{
    // Delete all Nodes from first non-dummy up to but not including
    // the dummy

    while (m_head->m_next != m_head)
        doErase(m_head->m_next);

    // Delete the dummy

    delete m_head;
}

template <class T>
Sequence<T>& Sequence<T>::operator=(const Sequence& rhs)
{
    if (this != &rhs)
    {
        Sequence temp(rhs);
        swap(temp);
    }
    return *this;
}


======================================
    Inheritance / Virtual Functions
======================================
//if a class is designed to be a base class, declare a virtual destructor and implement it

class Device //device is the base class for several other things.
{
    public:
    virtual ~Device(){}
    virtual void open() = 0;
    virtual void write(char c) = 0;
    virtual void close() = 0;
};

class BannerDisplay : public Device
{
public:
    virtual void open(); //implemented somewhere
    virtual void write(char c); //implemented somewhere
```

```
    virtual void close(); //implemented somewhere
private:
    ...
};

class Modem : public Device
{
public:
    virtual void open(); //implemented somewhere
    virtual void write(char c); //implemented somewhere
    virtual void close(); //implemented somewhere
private:
    ...
};

void writeString(Device& d, string msg)
{
    for (int k = 0; k != msg.size(); k++)
        d.write(msg[k]);
}
```

I want to sort a pile of N items:

```
    if (N > 1)
    {
        split the pile about evenly into two insorted piles
        sort the left subpile
        sort the right subpile
        merge the two sorted subpiles into one sorted pile
    }
//to debug, assume a recursive function works
    void sort(int a[], int b, int e)
    {
        if (e - b >= 2)
        {
            int mid = (b + e) / 2;
            sort(a, b, mid);
            sort(a, mid, e);
            merge(a, b, mid, e);
        }
    }

    int main()
    {
        int arr[5] = {40, 30, 20, 50, 10};
        sort(arr, 0, 5);
    }
```

To prove P(N) for all N >= 0:

1. Prove: P(0)
2. Prove: If P(k) is true for all k < N, then P(N)

Constructing a program with multiple source files

```
#include "file"

#ifndef OWEN_H
#define OWEN_H


#endif
```

File I/O

```
    ostream is a base class of ofstream



=========================================================================================
                                Notation
=========================================================================================
(prefix, postfix, w/ accompanying algorithms)

infix notation:
8-6/2+1  //the same thing as the line above, can be more confusing for a human to parse sometimes?

postfix notation:
8 6 2 / - 1 +   //once again, the same thing
((8 (6 2 /) -) 1 +) //the associated groupings.
//Postfix notation doesn't need any additional specificity. To the computer, it's always unambiguous what it operates on
//postfix is actually easier to process than prefix/infix notation, runs faster than both. Common expression evaluation: given something in infix, translate it to postfix, then it's easy to evaluate


8 - 6 / 2 + 1
//  translating infix to postfix:
    Initialize postfix to empty
    Initialize the operator stack to empty
    For each character ch in the infix string
        Switch (ch)
          case operand:
            append ch to end of postfix
            break
          case '(':
            push ch onto the operator stack
            break
          case ')':
              // pop stack until matching '('
            While stack top is not '('
              append the stack top to postfix
              pop the stack
            pop the stack  // remove the '('
            break
          case operator:
            While the stack is not empty and the stack top is not '(' and precedence of ch <= precedence of stack top
                append the stack top to postfix
                pop the stack
            push ch onto the stack
            break
    While the stack is not empty
        append the stack top to postfix
            pop the stack

8 6 2 / - 1 +
//  evaluating a postfix sequence: (pseudocode)
    Initialize the operand stack to empty
    For each character ch in the postfix string
        if ch is an operand
            push the value that ch represents onto the operand stack
        else // ch is a binary operator
            set operand2 to the top of the operand stack
        pop the stack
            set operand1 to the top of the operand stack
        pop the stack
        apply the operation that ch represents to operand1 and operand2, and push the result onto the stack
    When the loop is finished, the operand stack will contain one item, the result of evaluating the expression

```