ALWAYS PASS BY CONST REFERENCE A LIST, STRING,

Classes:
Class Name{
  public:
  private:
};
**Implementations include Name:: before all member functions, constructors included.
Templated classes:
template class <T> className{};

Hash Collision Resolution:
1. Store a linked list of values in each bucket.
2. Open Addressing:
  linear probing:
    (i+1)%n, (i+2)%n, ...
  quadratic probing:
    (i+1)%n, (i+2*2)%n, (i+3*3)%n, ...
  These two methods must mark buckets as previously occupied when/if their contents are deleted.

Exceptions:
#include <exception>, std::exception
STL base exceptions use the .what()
1. When error is encountered (perhaps by if statement), then throw something: throw 20; or throw std::string("OHNO");
2. Throw abandons the rest of the code block.
3. try/catch can be tied together:
try{
  //code block
} catch (specific type or ...){
  //do stuff
}
4. catches can be chained to capture multiple throws from the same try block, and ... will allow the catch to grab anything- but the thrown object won't be useable.
5. Functions can specify throws in their prototypes:
int foo(int a, int b) throw(double, bool);
**Exceptions are the only way for constructors to fail!

Useful Includes:
  #include <iostream>
  #include <fstream>
-ofstream -ifstream
  #include <iomanip>
-setprecision(num) -setw (num) -setfill(char)
  #include <cstdlib>
  #include <cassert>

Erase for two children:
- Find the location to be erased
- Find the node with the greatest value in the left subtree or the node with the smallest value in the right subtree
- The value in this node may be safely moved into the current node because of tree ordering
- Then recursively apply erase to that node which can have at most one child

Stop and Copy:
1. Split memory in half (working and copy).
2. When out of working memory, stop and begin garbage collection.
(a) Scan and free pointers at the start of the copy memory.
(b) Copy root to copy memory, incrementing free. Whenever a node is copied from working memory, leave a forwarding address to its new location in copy memory in the left address slot of its old location.
(c) Starting at the scan pointer, process the left and right pointers of each node. Look for their locations in working memory. If the node has already been copied (i.e., it has a forwarding address), update the reference. Otherwise, copy the location (as before) and update the reference.
(d) Repeat until scan == free.
(e) Swap the roles of the working and copy memory

Ways to overload operators:
1. As a non-member function
2. As a member function
3. As a friend function

Binary Search Trees:
  Right: larger, Left: smaller
  Leaves have no children.
  Nodes must fulfill all inherited requirements

Red-Black Binary Trees:
1. reds have only black children
2. leaves have black children and are black
3. All paths from a node to a null child pointer have the same # of black nodes
4. THE ROOT IS ALWAYS BLACK

All dynamically allocated classes require a destructor, a copy constructor, and an assignment operator:
  ~ClassName();
  ClassName(const ClassName & c);
  ClassName operator=(const ClassName & c);

Reference Counting:
1. Attach a counter to each Node in memory.
2. When a new pointer is connected to that Node, increment the counter
3. When a pointer is removed, decrement the counter
4. Any Node with counter == 0 is garbage and is available for reuse.

Sorting: #include <algorithm>
  sort(vector.begin(), vector.end(), myfunction);
  sort(vector.begin(), vector.end()); or list.sort(function;)
  **myfunction is a bool function that accepts two items by const reference.
  **By convention/default, items are sorted smallest to largest
  **You can pass iterators that aren't at either end to sort only part of the structure

Binary Tree Traversals:
  In order: recursively go left, then root, then right (1, 2, 3)
  Pre order: parents are listed before their children (2, 1, 3)
  Post order: children are listed before their parents (1, 3, 2)
Traversal Sample code:
void print(ostream& ostr, const Node<T>*p){
  if (p != NULL){
    1. print(ostr, p->left);
    2. ostr << p->value;
    3. print(ostr, p->right);
  }
}

Mark-Sweep:
1. Add a mark bit to each location in memory.
2. Keep a free pointer to the head of the free list.
3. When memory runs out, stop computation, clear the mark bits and begin garbage collection.
4. Mark:
(a) Start at the root and follow the accessible structure (keeping a stack of where you still need to go).
(b) Mark every node you visit.
(c) Stop when you see a marked node, so you don't go into a cycle.
5. Sweep:
(a) Start at the end of memory, and build a new free list.
(b) If a node is unmarked, then it's garbage, so hook it into the free list by chaining the left pointers.

Functors:
class a_class_name{
public:
  <insert generic class stuff here>
  return_type operator() (<args>);
private:
  <Insert representation here>
};
called as:
a_class_name hashFunc;
int location = hashFunc();
**provided return_type is an int

Class Polymorphism:
-Is-A, Has-A, As-A implementations.
-Constructors of a derived class call the base constructor first--YOU MUST DO THIS EXPLICITLY! For destructors, the reverse is true.
-Destructor must be marked virtual in base classes.
-Public inheritance: doesn't change public, private, or protected designations of inherited class aspects.
-Protected inheritance: public members become protected. Nothing else changes.
-Private inheritance: all inherited members become private.
-Protected designation means that the members are accessible by child classes, but not outside of the class.
-Private designation means that the members aren't accessible by children or from outside.
-Virtual allows a function to be overwritten lower down in the inheritance tree.

Sets:

s.begin(), s.end(), s.cbegin(), s.cend()	returns begin and end iterators, respectively (iterator, const_iterator)	Const time
s.rbegin(), s.rend(), s.crbegin(), s.crend()	returns reverse itr to last value and first value, respectively (reverse_iterator, const_reverse_iterator)	Const time
s.find(value), s.count(value)	return iterator to value or end() *(s.count(value) returns 1 or 0)	Log(n)
s.insert(value)	returns pair of iterator, bool	Log(n)
s.insert(s::iterator, value)	returns just an iterator	Log(n)
s.insert(s::iterator, s::iterator)	inserts from first up to (not including) last. returns ??	??
s.erase(s::iterator)	no return	Const time
s.erase(value)	returns # elements erased	Log(n)
s.swap(otherset)	no return, swaps the contents of the two containers	Const time
s.clear()	clears the set	O(n)
s.empty(), s.size()	returns bool true if empty, returns int size of set	Const time

Maps:

Usage Form	Return Value	Order Notation
m.insert(make_pair(key,val))	Returns pair of iterator and bool – false if unsuccessful	Log(n)
m::iterator	Pair of first, second	n/a
m.find()	Returns m::iterator	Log(n)
m.erase(m::iterator)	No return	Const time
m.erase(key value)	Returns 1 or 0	Log(n)
m.erase(itr1, itr2)	No return	n
m[]	Returns reference to value associated with key	Log(n)
m.clear()	No return	n

**MAKE SURE TO IMPLEMENT < operator if using custom class

```
void percolate_down(std::vector<T>& heap){
    int i = 0;
    while ((2*i+1) < heap.size()){// While there is a left child
        //If there is a right child and it is smaller than the left
        if ((2*i+2) < heap.size() && heap[2*i+2] < heap[2*i+1])
            i = 2*i+2;
        else
            i = 2*i+1;
        if(heap[i] < heap[(i-1)/2]){
            T temp = heap[i];
            heap[i] = heap[(i-1)/2];
            heap[(i-1)/2] = temp;
        }else break;
    }
}
```

Types of Smart Pointers:
auto_ptr:
When copied (copy constructor), new object takes ownership, old object is now empty.
unique_ptr:
Cannot be copied (copy constructor not public). Can only be moved to transfer ownership.
scoped_ptr (Boost):
deletes things when they go out of scope. Alternate to auto_ptr. Cannot be copied.
shared_ptr:
Reference counted ownership of pointer. Unfortunately, circular references are still a problem.
weak_ptr:
Use with shared_ptr. Memory is destroyed when no more shared_ptrs are pointing to object.

Concurrency:
Atomic operations are guaranteed to complete without interruption.
Concurrent systems should produce the same result that would occur in the original sequential system.
To avoid certain concurrent permutations of operations that are undesirable, operations can be serialized with a mutex. (#include <mutex>)
For example, a class can have a mutex member variable. Access is controlled with m.lock() and m.unlock().
Threads can be added with #include <thread>

```
For a binary heap of n values:
indices begin at 0
The parent to a node is (i-1)/2
The left child is 2i+1
The right child is 2i+2
The last leaf is at n-1
The last non-leaf is at (n-1)/2
```

Order Notation:

-- O(1), a.k.a. CONSTANT: The number of operations is independent of the size of the problem. e.g., compute quadratic root.
-- O(log n), a.k.a. LOGARITHMIC. e.g., dictionary lookup, binary search.
-- O(n), a.k.a. LINEAR. e.g., sum up a list.
-- O(n log n), e.g., sorting.
-- O(n^(1/2)), O(n^3), O(n^k), a.k.a. POLYNOMIAL, find the closest pair
-- O(2^n), O(kn), a.k.a. EXPONENTIAL. e.g., Fibonacci, playing chess.
-- O(N * M), nested for loops.

Lists: (No [] opeators, navigate with iters)

Same l.begin, l.end, etc	Returns appropriate iterator	O(1)
l.push_front(val), l.push_back(val)	Insert element at beginning/end of list, none.	O(1)
l.pop_front(), l.pop_back()	Remove the first/last element in list, none.	O(1)
l.insert(iter, val)	Insert val before iter, returns iter to new item.	O(1)
l.erase(iter) || l.erase(iter_begin, iter_end)	Erases iters, returns item after last erased	O(1)
l.unique()	Removes duplicates from list, returns none.	O(n)
l.reverse()	Reverses elements in list, returns none.	O(n)

Queue
Implemented as hobbled linked list, where you can only pop from the front and push to the back. Member functions in STL are q.front(), q.back(), q.push(value), and q.pop(). pop and push have no return values.

Priority Queue
Implemented as complete min heaps.
Remember heaps are actually stored in arrays.
A complete heap has no “holes” in it.
All vector slots are filled except for the last leaves.
Every node has the max number of children possible, except the bottom layer. Leaf nodes are full and fill left to right in the heap.
Min value is the root, max is going to be a leaf node.

Binary Heap
A binary tree that:

Is complete; that is, all levels of the tree, except possibly the last one (deepest) are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right.
All nodes are *either* [greater than or equal to] *or* [less than or equal to] each of its children, according to a comparison predicate defined for the heap.
Heaps with a mathematical "greater than or equal to" (≥) comparison predicate are called *max-heaps*; those with a mathematical "less than or equal to" (≤) comparison predicate are called *min-heaps*.

Min-heaps = Priority Queue

INSERT

To add an element to a heap we must perform an *up-heap* operation (also known as *bubble-up*, *percolate-up*, *shift-up*, *trickle up*, *heapify-up*, or *cascade-up*), by following this algorithm:

1. Add the element to the bottom level of the heap.
2. Compare the added element with its parent; if they are correctly ordered, stop.
3. If not, swap the element with its parent and return to the previous step.

DELETE

The procedure for deleting the root from the heap (effectively extracting the maximum element in a max-heap or the minimum element in a min-heap) and restoring the properties is called *down-heap* (also known as *bubble-down*, *percolate-down*, *shift-down*, *trickle down*, *heapify-down*, *cascade-down* and *extract-min/max*).

1. Replace the root of the heap with the last element on the last level.
2. Compare the new root with its children; if they are in the correct order, stop.
3. If not, swap the element with one of its children and return to the previous step. (Swap with its smaller child in a min-heap and its larger child in a max-heap.)

```
void push( const T& entry ){
    m_heap.push_back(entry);
    // Now percolate up
    int i = m_heap.size()-1;          // Begin iterator integer at newest entry
    while ((i-1)/2 > -1){             // While there is a parent
        if (m_heap[i] < m_heap[(i-1)/2]){ // If the parent is greater than the value, swap them
            T temp = m_heap[i];
            m_heap[i] = m_heap[(i-1)/2];
            m_heap[(i-1)/2] = temp;
            i = (i-1)/2;                  // Move along
        }else break;
    }
}
```

```
//Copy Constructor Prototype:
Table(const Table& t) { copy(t); }
//Assignment Operator Prototype:
const Table& operator=(const Table& t);

//Assgn 1 Tble 2 another, avoid self-assgnment
template <class T> const Table<T>& Table<T>::operator=(const Table<T>& v) {
  if (this != &v) {
    destroy(); //destructs this
    this->copy(v); //Copy is below
  }
  return *this;
}
//Create the Tble as a copy of the given Tble
template <class T> void Table<T>::copy(const Table<T>& v) {
  this->create(v.rows,v.cols);
  for (unsigned int i = 0; i < rows; i++) {
    for (unsigned int j = 0; j < cols; j++) {
      values[i][j] = v.values[i][j];
    }
  }
}
```

20.17 Leftist Heaps — Mathematical Background

- **Definition:** The *null path length* (NPL) of a tree node is the length of the shortest path to a node with 0 children or 1 child. The NPL of a leaf is 0. The NPL of a NULL pointer is -1.
- **Definition:** A *leftist tree* is a binary tree where at each node the null path length of the left child is greater than or equal to the null path length of the right child.
- **Definition:** The *right path* of a node (e.g. the root) is obtained by following right children until a NULL child is reached.
  - In a leftist tree, the right path of a node is at least as short as any other path to a NULL child.
- **Theorem:** A leftist tree with $r > 0$ nodes on its right path has at least $2^r - 1$ nodes.
  - This can be proven by induction on r.
- **Corollary:** A leftist tree with n nodes has a right path length of at most $\lfloor \log(n+1) \rfloor = O(\log n)$ nodes.
- **Definition:** A *leftist heap* is a leftist tree where the value stored at any node is less than or equal to the value stored at either of its children.

20.18 Leftist Heap Operations

- The `insert` and `delete_min` operations will depend on the `merge` operation.
- Here is the fundamental idea behind the merge operation. Given two leftist heaps, with `h1` and `h2` pointers to their root nodes, and suppose `h1->value <= h2->value`. Recursively merge `h1->right` with `h2`, making the resulting heap `h1->right`.
- When the leftist property is violated at a tree node involved in the merge, the left and right children of this node are swapped. This is enough to guarantee the leftist property of the resulting tree.
- `Merge` requires $O(\log n + \log m)$ time, where m and n are the numbers of nodes stored in the two heaps, because it works on the right path at all times.