CS32 Final Overview:

//

Data Structures:

ITERATORS:

vector<int>::iterator it; ==> Syntax for an iterator of type int that operates on a vector

> Vectors: Can iterate through using brackets (like an array), or using an iterator
> Lists: Can't use brackets so must use iterators

Iterating Through Items:

> Use an iterator variable to enumerate through the contents of the container
> Similar to a pointer variable, but it is used on STL containers
> Start by pointing the iterator to the front of the container
  - Use begin() to point the iterator to the front of the container
> Similar to a pointer, you can increment and decrement the iterator up and down a container
  - Advance down the container using it++
> Use (*it) to access the values
  - Dereferencing operator
  - You could also dereference a value using the arrow operator (->)
> To point to the last time, you use the end() method
  - The end() method points the iterator to the item just after the container

Ex:

```
vector<int> myVec;
myVec.push_back(1234);
myVec.push_back(12);
myVec.push_back(34);
vector<int>::iterator it = myVec.begin();
while (it != myVec.end())
{
	cout << (*it);
	it++;
}
```

Constant Iterators:

> If you pass a constant reference of an object to a function, then you will need to use a constant iterator
  ==> vector<int>::const_iterator it;

Iterator Properties:

> An iterator works like a pointer, but it is not truly a pointer
> An iterator is an object that knows three things:
  1. What element it points to
  2. How to find the previous element in the container
  3. How to find the next element in the container

Iterator Problems:

> Let's say you point an iterator to an item in a vector
> If you add or erase an item from the same vector, then the iterator would become invalidated
  - Leaves old pointer pointing to some random spot in memory
> NOTE:
  - This problem only occurs with vectors ==> IT DOESN'T OCCUR WITH LISTS, MAPS, OR SETS
    > That is unless you delete the item that the iterator is pointing to (but that is intuitive)

Ex:

main()
{
vector<string> x;
x.push_back("Alex");
x.push_back("Carey");
x.push_back("David");
vector<string>::iterator it;
it = x.begin();
x.push_back("Ben");
cout << *it; ==> ERROR!! WRONG! ==> Whatever the iterator was pointing to is invalidated if an item was added/erased
}

==

VECTORS:
#include <vector>
vector<int> isVector;

> A data structure that is used to store data in contiguous memory locations (similar to an array)
> A vector is a resizable array
- The vector grows/shrinks automatically when you add/delete items from the vector

----

Constructing Vector Objects:
> To create an empty vector ==> vector<int> isVector;
> To create a vector with a given number of elements (but left initialized to 0) ==> vector<int> isVector(10);
> To create a vector with a given number of elements that have the same value ==> vector<int> isVector(10, 3);

Operations:
push_back(int v); ==> To add items to the end of the vector
pop_back(); ==> To remove items from the end of the vector
empty(); ==> Returns a boolean value if the vector is empty or not
size(); ==> Returns an int value of the size of a vector
begin(); ==> Set the iterator here to point to the front of the vector
end(); ==> Set the iterator here to point to the position just after the last item ofthe vector
front(); ==> Returns the value of the first item
back(); ==> Returns the value of the last item

Change Items:
vector<int> vals(3); ==> Create a vector called vals with 3 elements that have values of 0
vector[0] = 42; ==> Use brackets to change the value of the first element from 0 to 42
cout << vals[2];
vals[4] = 1971; ==> ILLEGAL!! ERROR! ==> There is no item here, so you can't change its value

----

Front and Back:
cout << vals.front(); ==> Outputs 42 (based on the previous example)
cout << vals.end(); ==> Outputs 0 (based onn the previous example)

==

LIST (STL):
#include <list>
list<int> isList;

> STL list is a class that works like a linked list
> Lists can add/remove items from the front and the end of the list
> Cannot use square brackets to access elements ==> Must use an iterator

----

Operations:
push_back(int v); ==> Add an item to the end of the list
push_front(int v); ==> Add an item to the front of the list
pop_back(); ==> Remove the last item in the list
pop_front(); ==> Remove the first item inthe list
size(); ==> Returns an int value of the size of the list
empty(); ==> Returns a boolean value if the list is empty or not
front(); ==> Returns the value of the first item in the list
back(); ==> Returns the value of the last item in the list
begin(); ==> Set the iterator to point here to start at the beginning of the list
end(); ==> Set the iterator to point here to point to one after the last item in the list

----

Vectors or Lists?
> Vectors (Based on Dynamic Arrays)
- Pro: Can access any element using brackets quickly
Con: Adds new items slowly

> Lists (Based on Linked Lists)
- Pro: Fast insertion/deletion
- Con: Slow access to middle items

==

STACKS:
#include <stack>
stack<int> isStack;

> A data structure that holds a collection of elements that are always added to one end (the top)
> LIFO (Last In First Out) ==> The newest element added to the stack is the first one that is removed

----

Operations:
push(int v); ==> Items are inserted at the top of the stack
pop(); ==> Items are removed from the top of the stack (the most recent item added is the one removed)
top(); ==> Returns the value of the top item of the stack
empty(); ==> Returns a boolean value of whether or not the stack is empty
size(); ==> Returns an int value that is the size of the stack

----

Uses:
- Soring undo items in the word processor
- Evaluating expressions
- Converting infix expressions to postfix
- Solving mazes
- Keeping track of the runtime stack (useful for recursion tracing)

Maze Algorithm:
Input:
10x10 maze in a 2D array
Starting coordinates
End coordinates
Output:
True if the maze is solvable, and false otherwise

```
class Coord
{
	public:
		Coord(int x, int y) {}
		int r() const {return m_r;}
		int c() const {return m_c;}
	private:
		int m_r;
		int m_c;
};

bool pathExists(string maze[], int nRows, int nCols, int sr, int sc, int er, int ec)
{
	stack<Coord> coordStack;
	coordStack.push(Coord(sr,sc));
	maze[sr][sc] = '@';
	while (coordStack.empty() == false)
	{
		Coord start = coordStack.top();
		int r = start.r();
		int c = start.c();
		coordStack.pop();
		if (r == er and c == ec)
			return true;
		if (maze[r - 1][c] == '.')
		{
			coordStack.push(Coord(r - 1, c));
			maze[r - 1][c] = '@';
		}
		if (maze[r][c + 1] == '.')
		{
			coordStack.push(Coord(r, c + 1));
			maze[r][c + 1] = '@';
		}
		if (maze[r + 1][c] == '.')
		{
			coordStack.push(Coord(r + 1, c));
			maze[r + 1][c] = '@';
		}
		if (maze[r][c - 1] == '.')
		{
			coordStack.push(Coord(r, c - 1));
			maze[r][c - 1] = '@';
		}
	}
	return false;
}
```

==

QUEUES:
#include <queue>
queue<int> isQueue;

> A data strucutre that holds a collection of elements that are added at one end, and removed from the other
> FIFO (First In First Out) ==> The element that has been in the queue the longest, is the one that is first removed

----

Operations:
push(int v); ==> Insert an item at the back of the queue
pop(); ==> Remove an item from the front of the queue
front(); ==> Returns the value that is at the front of the queue (the first item added to the queue)
empty(); ==> Returns a boolean value of whether or not the queue is empty

size(); ==> Returns an int value that is the size of the queue

----

Uses:
- Solving a maze (unlike a stack, the queue will exmaine the oldest (x,y) location)

==

PRIORITY QUEUES:

> A priority queue is a special type of queue that removes items from the queue with the highest priority
> Priority Queues are implemented using heaps (more specifically ==> a maxheap)

Operations:
> Insert
> Get value of the highest priority item
> Remove the highest priority item from the queue
> Note:
- When you define a priority queue, you must specify how to determine the priority of each item in the queue

----

How to Implement?
> Use a special type of binary tree called a heap
> Preferably a maxheap (which stores the largest item at the root of the tree)

==

MAP:
#include <map>
map<string, int> isMap; ==> Associates string keys to int values

> Allows you to associate 2 related values
Ex: Names to phone numbers (strings to ints)
- Look up strings to see what ints are associated with it
- To Associate a string to a phone number ==> isMap["Carey"] = 8185552121
> Maps only associate in one diretion
- So ==> isMap[8185552121] = "Carey"; ==> WRONG!! ERROR!
> To efficiently search in both directions, use two maps

How does the map class work?
> Basically stores associations in a struct variable

```
struct Pair
{
    string first;
    int second;
};
```

> To access these variables, you will use first and second to refer to the given object's string or int values

----

How to search a map?
> First define an iterator ==> map<string, int>::iterator it;
> Then call map's find function to locate an assocation
- Note: You can only search by the left-hand side
it = isMap.find("Carey");
it = isMap.find(8185552121); ==> ERROR! WRONG! ==> Incorrect association (ints were on the right-hand side)
> Then you can look at the pair of values of the given object that the iterator is pointing to:
- cout << (*it).first;
- cout << (*it).second;

> If the find method can't locate the item, then it returns an iterator that points past the end of the map

```
it = isMap.find("Ziggy");
if (it == isMap.end())
        cout << "Couldn't find.";
else
        cout << (*it).second;
```

How to iterate through a map?

> Simply use a for/while loop to iterate through, just like with vectors and lists

Ex:

```
main()
{
        map<string, int> isMap;
        map<string, int>::iterator it;
        it = isMap.begin();
        while (it != isMap.end())
        {
                cout << (*it).first;
                cout << (*it).second;
                it++;
        }
```

Maps with complex data types:

> You can associate maps with user-defined data types, or classes/structs

> If you have a user-defined class on the left-hand side, then you must define an operator< method to compare the left-hand side values

- If the user-defined class is on the right-hand side, then you don't have to define a comparison operator

Ex:

```
struct Student
{
        string name;
        string idNum;
        bool operator< (const Student& other)
        {
                return (name < other.name);

                or

                return (idNum < other.idNum);
        }
};
main()
{
        map<Student, float> studentToGPA;
        Student d;
        d.name = "David Smallberg";
        d.idNum = 916451243;
        studentToGPA[d] = 1.3;
}
```

How are maps implemented?

> Maps are implemented using a binary search tree

> Because of this, maps always maintain items in alphabetical order (no sorting required)

==

SET:
```
#include <set>
set<int> isSet;
```

> A container that keeps track of unique values (no duplicates in the set)

Ex:
```
main()
{
    set<int> a; ==> Create a set
    a.insert(2); ==> Insert items
    a.insert(3);
    a.insert(4);
    a.insert(3); ==> If you insert a duplicate, it is ignored because the item is already present
    a.erase(2); ==> Erase items
    cout << a.size() << endl; ==> Returns the size of the set
}
```

Searching/Iterating through a set:

> You can iterate through a set, just like you can with a map

> Create an iterator, and use find() method, or use iterator arithmetic (with begin() and end() to iterate)

Ex:
```
main()
{
    set<int> a;
    a.insert(3);
    a.insert(4);
    a.insert(5);
    set<int>::iterator it;
    it = a.begin();
    while (it != it.end())
    {
        cout << (*it);
        it++;
    }
    it = a.find(2);
    if (it == a.end())
        cout << "Couldn't find.";
    else
        cout << "I found " << (*it);
}
```

Sets with complex data types:

> Just like with maps, you can have sets that hold complex data types (structs/classes)

> Just like with maps, if your set has a complex data type, then you must define an operator<

Ex:
```
struct Course
{
    string name;
    int units;
    bool operator< (const Course& other)
    {
        return name < other.name;

        or

        return units < other.units;
    }
};
main()
{
    set<Course> courseSet;
```

```
        Course lec1;
        lec1.name = "CS31";
        lec1.units = 4;
        courseSet.insert(lec1);
    }
```

How are sets implemented?

> Sets are implemented using a binary search tree
> Because of this, sets always maintain items in alphabetical order (no sorting required)

==

TREE:

> A speical linked-list based data structure

Applications:

> To organize hierarchal data (e.g. family trees)
> To make info easily searchable (e.g. binary search trees)
> To simplify evaluating mathematical expressions

Basic Facts:

1. Trees are made up of nodes
2. Each tree node can have 2 or more next pointers

```
struct Node
{
    int value;
    Node *left, *right;
};
Node* rootptr;
```

3. Every tree has a root pointer (similar to a linked list's head pointer)
4. The top node of a tree is called the "root" node
5. Every node may have 0 or more "children" nodes
6. A tree with no nodes is called an "empty tree" ==> The root pointer is nullptr

----

Tree Nodes can have many children:

> A tree node can have more than just two children

Ex:

```
struct Node
{
    int value;
    Node* pChildrren[26];
};
```

>However, a binary tree has a maximum of 2 children nodes per node

==

BINARY TREE:

> A special form of a tree
> Every node has at most 2 children nodes ==> left child and right child

```
struct BinaryTree
{
    int value;
    Node *right, *left;
};
```

Binary Tree Subtrees:

> You can pick any node in the tree and then focus on it's "subtree" ==> It and all the nodes below it

----

Full Binary Trees:

> A binary tree in which every leaf node has the same depth, and every non-leaf node has exactly 2-children
- By same depth ==> This means that the very bottom of the tree all has leaf nodes at the same level of the tree

----

Operations on Binary Trees:

> Iterate through
> Search for an item
> Add new items
> Delete an item
> Delete the entire tree (destruction)
> Remove a whole section of the tree (pruning)
> Add a whole new section to a tree (graphing)

Iterating through each item (traversing):

> Binary tree Traversals:
- Every traversal begins with the root node:
- 4 Methods of Traversal:
    1. Preorder
    2. Inorder
    3. Postorder
    4. Levelorder
- By processing the current node, we mean...
    1. Print the current node's value out
    2. Search the current node to see if its value matches the one you're searching for
    3. Add the current node's value to a total for the tree

> Traversals involve recursive solutions

Preorder:

1. Process the current node
2. Process the left subtree
3. Process the right subtree

Code:

```
void preorder(Node* curr)
{
	if (curr == nullptr)
		return;

	cout << curr->value << " "; ==> Process the current node
	preorder(curr->left); ==> Process the left subtree
	preorder(curr->right); ==> Process the right subtree
}
```

Inorder:

1. Process the left subtree
2. Process the current node
3. Process the right subtree

Code:

```
void inorder(Node* curr)
{
```

```
        if (curr == nullptr)
            return;

        preorder(curr->left); ==> Process the left subtree
        cout << curr->value << " "; ==> Process the current node
        preorder(curr->right); ==> Process the right subtree
    }
```

Postorder:

1. Process the left subtree
2. Process the right subtree
3. Process the current node

Code:

```
void postorder(Node* curr)
{
    if (curr == nullptr)
        return;

    postorder(curr->left); ==> Process the left subtree
    postorder(curr->right); ==> Process the right subtree
    cout << curr->value << " ";
}
```

Levelorder:

> Visit each level's nodes, from left to right, before visiting nodes in the next level

Algorithm:

1. Use a temp ptr variable and a queue of Node pointers
2. Insert the root node ptr into the queue
3. While the queue isn't empty:
    a. Dequeue the front node ptr and put it in temp
    b. Process the node
    c. Add the node's children to the queue if they are not null

Code:

```
void levelorder(Node* curr)
{
    Node* temp;
    queue<Node*> levelQueue;
    levelQueue.push(curr);
    while(! levelQueue.empty())
    {
        temp = levelQueue.front();
        levelQueue.pop();
        cout << temp->value << " ";
        if (temp->left != nullptr)
            levelQueue.push(temp->left);
        if (temp->right != nullptr)
            levelQueue.push(temp->right);
    }
}
```

Big-O of Traversals:

> Efficiency: O(N)
> Each traversal must visit each node exactly once
> And since there are N nodes in the tree ==> Big-O is O(N)

Expression Evaluation:

> A binary tree can represent arithmetic expressions using a binary tree

Algorithm:
> Start by passing in a pointer to the root of the tree
1. If current node is a number, return its value
2. Recursively evaluate the left subtree and get result
3. Recursively evaluate the right subtree and get result
4. Apply the operator at the current node to the left and right results, return the result

----

Full Binary Tree:
> All nodes that are at level less than the height h (the bottom level) have 2 children each
> Each node in a full binary tree have left and right subtrees of the same height
> A full binary tree has as many leaves as possible, and they are all at the bottom level
- No missing nodes

Complete Binary Tree:
> Of height h is a binary tree that is fully grown to h - 1 (all levels, but the last level)
> A binary tree is complete if:
1. All nodes at level h - 2 and above have 2 children each
2. When a node at level h - 1 has children, all nodes to its left at the same level have 2 children each
3. When a node at h - 1 has a child, it is a left child

Balanced Binary Tree:
> A binary tree is height balanced, or simply balanced, if the height of any node's right subtree differs from the height of that node's left tree by no more than 1
> A complete binary tree is balanced, and therefore, a full binary tree is balanced

===

BINARY SEARCH TREE:

> A type of Binary Tree with specific properties that make them very efficient to search for a value

Definition:
> A binary search tree is a binary tree with the following two properties:
1. Given any node in the BST, all nodes in the left subtree must be less than the node's value
2. Given any node in the BST, all nodes in the right subtree must be greater than the node's value

----

Operations on a BST:
> Determine if BST is empty
> Search a BST for a value
> Insert
> Delete
> Find height
> Find the number of nodes and leaves in the BST
> Traverse BST
> Free memory used by BST (destructor)

Searching a BST:
> Input: A value v to search for
> Output: True if found, false otherwise
Algorithm:
- Start at the root of the tree
> If v is equal to curr node's value, then found
> If v is less than curr node's value, then go left
> If v is greater than curr node's value, then go right

- If we hit a nullptr, then not found

Code:

```
bool searchBST(Node* curr, int v)
{
	if (curr == nullptr)
		return false;
	if (curr->value == v)
		return true;
	else if (v < curr->value)
		searchBST(curr->left);
	else if (v > curr->value)
		searchBST(curr->right);
}
```

Big-O of Searching a BST:

> In the average case with N values ==> O(log N)
> In the worst case with N values ==> O(N)

Inserting New Value into BST:

> Input: A value v to insert

Algorithm:

- If tree is empty:
  > Allocate new node and put v in it
  > Point root ptr to new node ==> DONE!
- Start at root of the tree
- While we aren't done...
  > If v is equal to curr node's value ==> DONE!
  > If v is less than curr node's value
    - If there is a left child, go left
    - Else allocate new node with v in it, and set curr node's left ptr to new node ==> DONE!
  > If v is greater than curr node's value
    - If there is a right child, go right
    - Else allocate new node with v in it, and set curr node's right ptr to new node ==> DONE!

Code:

```
void insertBST(Node* curr, int v)
{
	if (curr == nullptr)
	{
		curr = new Node;
		curr->value = v;
	}
	for ( ; ;)
	{
		if (v == curr->value)
			return;
		else if (v < curr->value)
		{
			if (curr->left != nullptr)
				insertBST(curr->left);
			else
			{
				curr = new Node;
				curr->value = v;
				return;
			}
		}
		else if (v > curr->value)
		{
			if (curr->right != nullptr)
				insertBST(curr->right);
```

```
                    else
                    {
                        curr = new Node;
                        curr->value = v;
                        return;
                    }
                }
            }
        }
```

Big-O of Insertion to BST:
> O(log N)
> Use binary search to find where to insert the value ==> O(log N)
> Once found, insert new node ==> O(1) ==> Constant Time

Finding Min/Max of BST:
> Min value is located in the left-most node
> Max value is located in the right-most node

Code:

```
int getMin(Node* curr)
{
    if (curr == nullptr)
        return -1;
    if (curr->left == nullptr)
        return curr->value;
    return getMin(curr->left);
}
```

```
int getMax(Node* curr)
{
    if (curr == nullptr)
        return -1;
    if (curr->right == nullptr)
        return curr->value;
    return getMax(curr->right);
}
```

Big-O of Finding Max/Min of BST:
> O(log N)
> Only searching half of the elements

Print BST in Alphabetical Order:
> Inorder Traversal

Code:

```
void inorder(Node* curr)
{
    if (curr == nullptr)
        return;

    inorder(curr->left);
    cout << curr->value << " ";
    inorder(curr-right);
}
```

Big-O of Printing in Alphabetical Order:
> O(N)
> Must visit every node in the tree

Constructing a Binary Tree (Contructor):

```
class BST
{
    public:
        BST() {m_root = nullptr;}
    private:
```

```
                Node* m_root;
        }
```

Freeing the Whole Tree (Destructor):

> Postorder

Code:

```
void freetree(Node* curr)
{
        if (curr == nullptr)
                return;

        freetree(curr->left);
        freetree(curr->right);
        delete curr;
}
```

Big-O of Freeing Whole Tree:

> O(N)
> Again, must visit every node in the tree

Deleting a Node from a BST:

> Algorithm:
- Given a value v to delete from the tree:
  1. Find the value v in the tree, with a modified BST search:
     > Use 2 pointers: curr pointer and parent pointer
  2. If the node was found, delete it from the tree ==> Make sure to preserve the BST
     > Three cases:
       1. Node is a leaf
          1. Target node isn't root node
             1. Unlink the parent node from the target node ==> set parent node to nullptr
             2. Delete curr
          2. Target node is the root node
             1. Set root pointer to nullptr
             2. Delete curr
       2. Node has one child
          1. Target node isn't root node
             1. Relink parent node to curr's only child
             2. Delete curr
          2. Target node is the root node
             1. Relink parent node to curr's only child
             2. Delete curr
       3. Node has two children

       Background:
          > We don't actually delete the node itself
          > Instead, we replace it's value with an appropriate one
             1. Replace with curr's left subtree's largest value
             2. Replace with curr's right subtree's smallest value

     Rule:
     > Pick one, copy value up, delete node
       - After you copy the value, then you have to delete by Case 1 or 2 (no children or 1 children deletion)
       - Guaranteed to have 0 or 1 child (by picking smallest in right or largest in left)

Code:

```
void deletenode(Node* curr, Node* parent, int v)
{
        searchnode(curr, parent, v); ==> Curr will be pointing to the node to delete, parent will be pointing one above it
```

```
				if (curr->left == nullptr && curr->right == nullptr) ==> If node is a leaf
				{
					if (parent == nullptr) ==> Node is the root
					{
						delete curr;
						curr = nullptr;
					}
					else
					{
						delete curr;
						parent = nullptr;
					}
				}
				else if ((curr->left != nullptr && curr->right == nullptr) || (curr->left == nullptr && curr->right != nullptr)) ==> Node has one child
				{
					if (curr->left != nullptr) ==> Relink parent node to curr's child (either left or right)
					{
						parent = curr->left;
						delete curr;
					}
					else
					{
						parent = curr->right;
						delete curr;
					}
				}
				else ==> Node has two children
				{
					Replace curr's value with left subtree's largest value
					Delete curr by either (no children rule or 1 child rule)
				}

			}
			void searchnode(Node* curr, Node* parent, int v)
			{
				if (curr == nullptr)
					return;
				parent = nullptr; //curr = root
				while (curr != nullptr)
				{
					if (v == curr->value)
						return;
					else if (v < curr->value)
					{
						parent = curr;
						curr = curr->left;
					}
					else if (v > curr->value)
					{
						parent = curr;
						curr = curr->right;
					}
				}
			}

		----------------------------------------------------------------
```

Where are BSTs used?

> STL maps and sets use a BST (they both store their items in alphabetical order)

STL Map Example:

```
main()
{
	map<string, float> stud2GPA;
	stud2GPA["Carey"] = 3.62; ==> BST Insert
	cout << stud2GPA["David"]; ==> BST Search
}
```

STL Set Example:

```
main()
{
	set<int> a; ==> Construct a BST
	a.insert(2); ==> Insert into BST
	a.erase(2); ==? Delete from BST
}
```

===

BALANCED SEARCH TREE:

> A binary tree is "perfectly balanced" if for each node, the number of nodes in its left and right subrees differ by at most 1 level

> Perfectly balanced search trees have a max height of log N

> Three popular approaches to building balanced search trees

===

AVL TREES:

> A Balanced Search Tree in which the height of the left and right subtrees of each node differ by at most 1

Implement AVL Tree:

> Each node has a balanced value

- 0 if node's left/right subtrees have the same height
- -1 if the node's left subtree is 1 higher than the right
- 1 if the node's right subtree is 1 higher than the left

```
struct AVLNode
{
	int value;
	int balance;
	Node *left, *right;
};
```

> Note:

- When you insert, you have to update the balanced values

Inserting into an AVL Tree:

> Case 1:

- After you insert the new node, all nodes in the tree still have balance of -1, 0, 1
    1. Insert new node normally (Binary Search Tree equivalent)
    2. Update the balances of all parent nodes, from new leaf to root
    3. If all balances are -1, 0, 1 ==> DONE!

> Case 2:

- After you insert the new node, it causes one or more balances to become < -1 or > 1
    > Must rebalance the tree
    > Two subcases:
        - Single rotation:

> If we add a node to the subtree that causes any node in the AVL tree to go out of balance, then rotate

- Double rotation:
  > Tricky stuff ==> Just know how to insert and rebalance (without writing code ==> trace through it)

==

HEAPS:

> Uses a special type of binary tree to hold its data
> A heap is a special type of binary tree (not BST)
> Two types of Heaps:
- Maxheaps:
  1. Quickly insert a new item into the heap
  2. Quickly retrieve the largest item from the heap
- Minheaps:
  1. Quickly insert a new item into the heap
  2. Quickly retrieve the smallest item from the heap

All Heaps Use A "Complete" Binary Tree:
> Complete Binary Tree:
- The top N - 1 levels of the tree are full
- All nodes on the bottom-level must be left-justified (no empty slots between nodes)

==

MAXHEAPS:

> A binary tree with the following rules:
1. The value contained by a node is greater than or equal to the value of its children
2. The tree is a complete binary tree
> Biggest (highest priority item) is always at the root(top) of the heap

----

Operations:
> Extracting the Biggest Item:
1. If the tree is empty, return error
2. Otherwise, the top item is the biggest value ==> Remember it for later
3. If the heap has only one node, then delete it, and return the saved value
4. Copy the value from the right-most node in the bottom-most row to the root node
5. Delete the right-most bottom-most row node
6. Repeatedly swap the new root value with the larger of its two children until the value is greater than or equal to both of its children ("sifting down")
7. Return the saved value to the user
> Adding a node:
1. If the tree is empty, create a new root node and return
2. Othewise, insert the new node in the left-most bottom-most position of the tree (so it is still a complete binary tree)
3. Compare the new value with it's parent value
4. If the new value is greater than the parent's value, then swap them
5. Repeat steps 3 and 4 until the new value rises to the proper position in the heap ("Reheapification")

Implementing the Heap:
> Visually:
- Binary Tree:

- Note: Not how you implement it, but this is how it visually looks

> Actually:
- Array
- We know that each level of the tree has 2x the number of nodes than the previous level (except for the bottom level)
- So we can copy nodes a level at a time into an array
    > Root node in array[0]
    > The next 2 values in next 2 slots
    > The next 4 values in next 4 slots
    > Keep a counter

Array-Based Tree Properties:
1. Root value is always in array[0]
2. Always find the bottom-most right-most node in array[count - 1]
3. Always find the bottom-most left-most empty spot (to insert) in array[count]
4. We can add or remove a node by setting array[count] = value and/or updating it

----

How do we find the children's position?
> leftChild(parent) = 2 * parent + 1
> rightChild(parent) = 2 * parent + 2

How do we find the parent's position?
> parent = (child - 1)/2) ==> Works for both left and right children

----

Heap Summary:
1. The root of the heap goes in array[0]
2. If the data for a node appears in array[i], then its children, if they exist, are in:
    > leftChild: array[2i + 1]
    > rightChild: array[2i + 2]
3. If the data for a non-root node is in array[i], then the parent is always in array[(i - 1)/2]

Using Array to Implement Heap:
> Operations:
- Locate the root node
- Locate (and delete) bottom-most, right most node in the tree
- Add a new node at the bottom-most, left-most empty position in the tree
- Easily locate the parent and children of any node in the tree

----

Extracting from Maxheap:
1. if count == 0 (empty tree)
    return error
2. Otherwise, array[0] holds the biggest value
    remember it for later
3. if count == 1 (only node)
    set count = 0, return saved value
4. Copy the value of the right-most, bottom-most node to the root
    array[0] = array[count - 1]
5. Delete the right-most, bottom-most node
    count = count - 1
6. Repeatedly swap just moved value with the larger value of its 2 children
    starting with i = 0, compare and swap
    array[i] with array[2i + 1] and array[2i + 2]
7. Return the saved value to the user

Adding a node to Maxheap:
1. Insert new node in bottom-most, left-most node
    array[count] = value
    count = count + 1
2. Compare the new value array[i] with its parent's value
    array[(i - 1)/2]
3. If the new value is greater than it's parent's value, then swap them
4. Repeat steps 2-3 until the new value rises to its proper place in the tree (array)

==

MINHEAPS:

> A binary tree with the following rules:
1. The value contained by a node is less than or equal to the value of its children
2. The tree is a complete binary tree

> Smallest (lowest priority item) is always at the root(top) of the heap

==

TABLES:

> In CS lingo, a group of related data is called a "record"
> Each record has a bunch of "fields" (e.g. name, phone number, birthday etc.)
> A bunch of records is called a "table"
> A field (like SSN) that has a unique value across all records is called a "key field"

Ex: We want to write a program to keep track of all BFFs...
> You want to remember all info about each BFF
> You want to quickly search for a BFF in more than one way:
- Find info on BFF "David"
- Find info on BFF whose phone number is 867-5463

Implementing Tables:
> How can you create a record?
- Use a struct or class to represent a record of data
> How can you create a table?
- Create an array or vector of your struct
> How can you let the user search for a record with a particular field?
- Write a search function iterating through the vector (in multiple ways)

Making an efficient table:
1. Still use a vector to store all records
2. Add a data structure that lets us associate each person's name with their slot number in the vector
3. Add another data structure that lets us associate each person's id number with their slot number too

Ex:
```
vector<Student> m_students;
map<int, int> m_idToSlot;
map<string, int> m_nameToSlot;

void addStudent (Student& stud)
{
    m_students.push_back(stud);
    int slot = m_students.size() - 1;
    m_nameToSlot[stud.name] = slot.
    m_idToSlot[stud.idNum] = slot;
```

}

> To insert you have to push it onto the vector, and update both BST's

> To delete, you have to remove from the vector, and update both BST's

----

Using Hash Tables to Speed up Tables:

> Now we can have O(1) searches by name

> But hash tables store data in a random order

- While a BST is slower, it does order the key fields in alphabetical order

==

HASH TABLES:

> So far, the most efficient ADT to insert and search data is the binary search tree O(log N)

> We want to be able to search in constant time (arrays??)

> A bucket is the CS lingo for a "slot" in an array

> Need a hash function to map values to their given bucket in the array

> Purpose of hash function: ==> To take a "key" and map it to a number

----

Types of Hash Tables:

> Closed (Linear Probing) Hash Table

> Open Hash Table

Difference between closed/linear probing and open hash tables?

> Closed:

- Fixed size (if you run out of room, you cant add any more)
- Unfixed load factor
- Collisions: Probe linearly until an open bucket is found
- Deletion: Realistically, you can't

> Open:

- Unfixed size
- Fixed load factor
- Collisions: bucket points to a linked list
- Deletion: You can just go to the correct bucket, and delete it from the list

----

Modulus Operator:

> % operator is used to divide 2 numbers and obtain the remainder

> % operator, when divided by a number, all of the remainders are less than that number

Ex:

0 % 5 = 0	5 % 5 = 5
1 % 5 = 1	6 % 5 = 1
2 % 5 = 2	7 % 5 = 2
3 % 5 = 3	8 % 5 = 3
4 % 5 = 4	9 % 5 = 4

> If you were to divide a bunch of numbers by 100,000 ==> The remainders would all be in the range from (0 - 99,999)

>Rule:

- When you divide by a given value N, then all remainders are guaranteed to be between 0 and N - 1

Hash Function:

> A hash function is a function that can be used to map data of arbitrary size to that of a fixed size

Ex:

```
int hashFunc(int idNum)
{
    const int ARR_SIZE = 100,000;
    int bucket = idNum % ARR_SIZE;
    return bucket;
}
```

> The function takes in an input value idNum and returns a value between 0 and ARR_SIZE - 1
> However, this hash function will return values that have the same remainder, and thus will lead to multiple values being in the same bucket ==> Collision
> What does a good hash function need to be?
1. Uniformly distribute ==> To minimize collisions
2. Compute values quickly

Collision:
> A condition where 2 or more values both "hash" to the same bucket in an array
> This causes ambiguity, and prevents us from telling what value is actually stored in the array

---

Hash Table Efficiency:
> Questions:
> How efficient is the hash table ADT?
> How long does it take to search for an item?
> How long does it take to insert an item>
> Answers:
> Depends on the type of hash table (e.g. closed vs open)
> How full the table is
> How many collisions there are in the table

> General Efficiency:
> If the table is completely (or nearly) empty...
- Max number of steps to insert?
> O(1) ==> constant time ==> After all it is an array (or linked list)
- Max number of steps to search?
> O(1) ==> constant time ==> After all it is an array (or linked list)
> If the table is completely (or nearly) full...
- Max number of steps to insert?
> O(N) ==> If the hash table is almost (or is) full, then you have to...
- Closed: Probe down linearly through a ton of buckets (assuming there are many collisions)
- Open: Traverse through N items of the linked list at the given bucket
- Max number of steps to search?
> O(N) ==> Same reasoning for insert above
> Open hash tables are almost always more efficent than closed hash tables

---

Load Factor:
> "Load" of a hash table is the max number of values, you intend to add, divided by the number of buckets in the array

$$L = \frac{\text{Max number of values to insert}}{\text{Total number of buckets}}$$

It's A Tradeoff:
> Either use a really big hash table with way too many buckets, and ensure fast searches
- Wastes a lot of memory
> Use a really small hash table and save memory

- Slower speeds

Hash Function for Strings:

> C++ built=in hash function for strings

```
#include <functional>
hash<string> str_hash;
int hashValue = str_hash(hashMe);
int bucket = hashValue % NUM_BUCKETS;
```

Hash Tables vs. Binary Search Trees:

	Hash Tables	Binary Search Trees
Speed:	O(1) regardless of number of Items	O(log N)
Simplicity:	Easy to Implement	More complex to implement
Max Size:	Closed(limited)/Open(Unlimitied)	Unlimited size
Space Efficiency:	Wastes a lot of space if large table	Only uses as much memory as needed
Ordering:	No ordering (random)	Alphabetical Order

==

CLOSED (LINEAR PROBING) HASH TABLES:

Ideas Behind the Closed Hash Table:

> The closed approach provides a means to address the collisions, by putting each value as close to the intended bucket as possible

> It is called a closed hash table because the size of the array is fixed

- Once every bucket is filled, you can't insert anymore values into the closed hash table

----

The Details:

> Each bucket is just a C++ struct

> Each bucket holds 2 items:

1. A variable to hold the value (e.g. int for student ID)
2. A used variable that holds a boolean value to indicate if the bucket in the hash table is used or not

```
struct Bucket
{
    int idNum;
    bool used; ==> If false then the bucket is empty, otherwise if true
};
```

Linear Probing Insertion:

> Use the hash function to locate the correct bucket in the array

> If the target bucket is empty, you can store the value there

- However, instead of storing a boolean value (like before), store the full original value ==> Prevents ambiguity

> If the bucket is occupied, probe down the array linearly, until we hit the first open bucket

- Put the value there

> Sometimes, you will need to insert near the end of the array (the bottom is full)
- If you run into a collision at the last bucket of the array, then wrap back up to the top of the array

Code:
```
#define NUM_BACK 10
class HashTable
{
	public:
		HashTable()
		{
			for (int i = 0; i < NUM_BACK; i++)
			{
				m_buckets[i].used = false;
			}
		}
		void insert(int idNum)
		{
		int bucket = hashFunc(idNum);	==> Step 1: Compute the starting bucket number
		for (int tries = 0; tries < NUM_BACK; tries++) ==> Step 2: Loop up to 10 times looking for an empty bucket
		{
			if (m_buckets[tries] == false)  ==> Step 3: Store the new item in the first unsed bucket
			{
				m_buckets[tries].idNum = idNum;
				m_buckets[tries].used = true;
				return;
			}
		      bucket = (bucket + 1) % NUM_BACK; ==> Step 4: If current bucket is occupied, advance to the next bucket
		}
		}
	private:
		Bucket m_buckets[NUM_BACK];
		int hashFunc(int idNum)
		{return idNum % NUM_BACK;}
};
```

Linear Probing Searching:
> To search for a value in a hash table, use a similar approach as insertion
> Compute the target bucket number with the hash function
- Look into the bucket for the given value ==> If found great! DONE!
- If you don't find it, probe linearly down the array until you...
1. Find the value
2. Hit an empty bucket ==> The value is not in the array
> As with insertion, if you end up finding a collision at the end of the array, then wrap back up to the top

Code:
```
#define NUM_BACK 10
class HashTable
{
	public:
		HashTable()
		{
			for (int i = 0; i < NUM_BACK; i++)
			{
				m_buckets[i].used = false;
			}
		}
		bool search(int idNum)
		{
			int bucket = hashFunc(idNum);
==> Step 1: Compute starting bucket
```

```
for (int tries = 0; tries < NUM_BACK; tries++)
==> Step 2: Loop through up to 10 times (checking all slots)
{
    if (m_buckets[bucket].used == false)
==> Step 3: If we encounter an empty bucket, the value isn't there
        return false;
    if (m_buckets[bucket].idNum == idNum)
==> Step 4: If we find our value, return true
        return true;
    bucket = (bucket + 1) % NUM_BACK;
==> Step 5: Advance to the next bucket
}
return false;
}
private:
    Bucket m_buckets[NUM_BACK];
    int hashFunc(int idNum)
    {return idNum % NUM_BACK;}
};
```

Closed/Linear Probing Deletion:

> Can't simply delete an item from the hash table
- If you delete an item where a collision happened, then it could lead to an error, when the target value was just a spot down
- You can't find a value and delete it (because the hash function produces many values with the same remainder)

> Important:
- Only used a closed hash table if you don't intend to delete items
Ex: Dictionary (you don't delete words, you only add them)

Problems with Closed Hash Table:
1. Difficult to delete items
2. It has a max numebr of items it can hold
3. Items aren't always in their respective buckets (could be slots below the correct bucket)

Closed Hash w/Linear Probing Efficiency:

> Given a particular load, it's easy to compute the average nuber of tries it'll take to insert/search for an item:
AVG number of tries = (1/2)[1 + (1/(1 - L))] for L < 1.0

> So, if the closed hash table has a

Load Factor of...	Search will Take
.10 (array is 10x bigger than required)	~1.05 searches ==> Array will fill 10% of the buckets
.20 (array is 5x bigger than required)	~1.12 searches
.30 (array is 3x bigger than required)	~1.21 searches
...	
.70 (array is 30% bigger than required)	~2.16 searches
.80 (array is 20% bigger than required)	~3.00 searches
.90 (array is 10% bigger than required)	~5.50 searches

==

OPEN HASH TABLES:

> Use if you plan to insert and delete a lot of values in hash table
> Instead of storing values directly in the bucket, we have a linked list at each bucket to store many values

- Each array bucket is a pointer that points to its respective linked list

> We use linked lists to deal with collision
- Therefore, all values will be found in the correct bucket position
- You could also use a binary search tree at each bucket too

> The linked list at each bucket can hold an unlimited number of values
- Therefore, open hash tables are not size-limited

Open Hash Efficiency:

> Given a particular load, it's easy to compute the average nuber of tries it'll take to insert/search for an item:

AVG number of tries = $1 + (L/2)$

> So, if the open hash table has a

Load Factor of...	Search will Take
.10 (array is 10x bigger than required)	~1.05 searches ==> Array will fill 10% of the buckets
.20 (array is 5x bigger than required)	~1.10 searches
.30 (array is 3x bigger than required)	~1.15 searches
...	
.70 (array is 30% bigger than required)	~1.35 searches
.80 (array is 20% bigger than required)	~1.40 searches
.90 (array is 10% bigger than required)	~1.45 searches

Sizing Your Hash Table:

> If you want to store 1000 items into an Open Hash, and be able to find an item in roughly 1.25 searches, how many buckets?

AVG number of tries = $1 + (L/2)$

> Step 1: Set equation above to be equal to 1.25 and solve for L:

$1 + (L/2) = 1.25$

$(L/2) = 0.25$

$L = 0.50$

> Step 2: Use load formula to solve for the required number of buckets:

$$L = \frac{\text{Max number of values to insert}}{\text{Total number of buckets}}$$

$$0.50 = \frac{1000}{\text{numBuckets}}$$

numBuckets = 2000 buckets

===

UNORDERED_MAP:

> Hash-based version of a map

```
#include <unordered_map>
using namespace std::trl == required for hash-based map
main()
{
	unordered_map<string, int> hm;
	unordered_map<string, int>::iterator it;
	hm["Carey"] = 10;
	hm["David"] = 10;
	it = hm.find("Carey");
	if (it == hm.end())
		cout << "Carey not found." << endl;
```

```
		else
		{
			cout << "When we look up " << it->first;
			cout << "We find that " << it->second;
		}
	}
```

	==

	GRAPHS:

//
Concepts:

	==

	CONSTRUCTION/DESTRUCTION/COPY CONSTRUCTOR/ASSIGNMENT OPERATOR:

Constructor:

Constructor::Constructor(); ==> Syntax

> Used to initialize a given class/struct object
> If you don't define one, then the compiler will define one for you (default constructor)
> If you do define one, then there will be no compiler-generated constructor
- So if you define a constructor with one parameter, and none with no parameters, then there will be no compiler-genereated one for you
> The default constructor (generated-compiler one) doesn't initialize scalar values (e.g. double, int, char)
- These are left uninitialized
> You are allowed to overload the constructor
- However, no two overloaded constructors should take the same type of parameters
> The constructor must have a class that takes no arguments because arrays are initialized by the no-parameter constructor multiple times

----

When is the constructor called?
> The constructor is called anytime a new instance of the object is created (whether it be from a local variable or dynamically allocated)
> The constructor is not called when you define a class pointer

	==

Destructor:

Destructor::~Destructor(); ==> Syntax

> Every class needs a destructor
> You must define your own destructor, if you dynamically allocate data (otherwise, there will be memory leaks)
> If you don't define a destructor, the compiler will generate one for you
> A class can only have one destructor

When is a destructor called?
> A destructor is called everytime, the delete keyword is used (in regards to dynamically allocated data)
> Everytime a local variable goes out of scope
> Everytime a local object reaches the end of its control
> When an array is created (the default constructor is called N times), when the function goes out of scope, the destructor is also called N times

===

Copy Constructor:

Circ::Circ(const Circ& other) {} ==> Syntax

> A copy constructor is used to create an instance of a variable from another pre-existing variable

> Classes are allowed to access private data members of the same class type

- Private only protects other class' from accesses the private data
- The parameter should be a const reference, and the return type should be the class

> Simply copy over the data from the other class' object to this class object

> If you don't define a copy constructor, the compiler will generate one for you

- However, this compiler-generated copy constructor does a "shallow" copy of the data

> You must define your own copy constructor if you are dynamically allocating data

----

When is the copy constructor called>

1. When instantiating an object from an existing object
2. When passing the object to a function by value
3. When a function returns an object by value

How to use the copy constructor?

1. Determine how much memory the old object used
2. Dynamically allocate the same amount of memory
3. Copy over the data from the old object to this object

===

Assignment Operator:

Circ& Circ::operator=(const Circ& rhs) {} ==> Syntax

> When you want to change the value of an existing object from another existing object

> Two objects are already created, and then we want to assign the value of one object over to the other

> You must define your own assignment operaot, when you dynamically allocated data

> The assignment operator uses, a combination of the destructor and the copy constructor

----

Steps:

1. Free any memory currently held by the target variable (b).
2. Determine how much memory is used by the source variable (a).
3. Allocate the same amount of memory in the target variable.
4. Copy the contents of the source variable to the target variable.
5. Return a reference to the target variable.

0. However, the assignment operator must check to make sure an object is not assigning itself to itself

- This would lead to the deletion of its data before copying it over

```
Ex: Circ& Circ::operator=(const Circ& rhs)
{
        if (this == &rhs) ==> Step 0 ==> Check to see if it is being assigned to itself
                return *this;
        delete [] m_sq; ==> Step 1 ==> Delete the already existing memory
        m_n = src.m_n;
m_sq = new int[m_n];
for (int j=0;j<m_n;j++)
m_sq[j] = src.m_sq[j];

return(*this); ==> Step 5 ==> Return a reference to the target variable
}
```

==

Class Composition:

> Is when a class object has other class data members
> Similar to inheritance, when such a class is created, the data members of the other class' variables is called first, and then the actual class itself
- The initialization of this can be achieved by the initializer list (in the same manner as with inheritance)
> Similar to inheritance, the destruction of such a class occurs in reversed order as the constructor (with the actual class members, and then the composition class' members)

==

INITIALIZER LISTS:

> Consists of the base class name, followed by its parameters in parentheses
> You can also include the derived class' members in the initializer list

==

INCLUDE GUARDS/PREPROCESSOR DIRECTIVES:

Include Etiquette:
> Never include a .cpp file in another .cpp file or .h file
> Never put a "using namespace std" in a header file
> Always include a .h file if your file is using that .h file
- Don't just assume that because a given class includes that file that you also don't have to include it

----

Preprocessor Directives:

```
#ifndef GIVEN_H ==> Syntax
#define GIVEN_H

#endif //GIVEN_H
```

> To prevent redefining a given class, and getting a compile-error, include the preprocessor directives
> The above code states that if a header file isn't already defined, then define it, if it already is defined, then ignore the redefinition

----

Header/Source File Etiquette:
> Always have separate files to define the header and source files

----

When to include?
> You should include a header file, if your header file needs to directly know about the given class:
1. You directly define that class in your class
2. You use that class' variable in any way
> You shouldn't include the header fle, if you are just using a pointer or reference to a class variable in your class
- Instead just use class theClassName vs. #include "theClassName.h"

==

ENCAPSULATION:

> The process of hiding the private implementation of the classes from the user and the programmer
> Data abstraction: Used to protect the data members from being accidentally changed or corrupted by the user
> Process of creating classes that have both private methods and member variables, as well as public methods/variables
> private:
- Used by the creator of the class
- Cannot be accessed directly by the user
- Are allowed to be indirectly accessed through public methods
> public:
- Users can directly access these methods and variables (if you so choose to include variables as private)
- A means for the user to be able to indirectly access the data (private variables)
> const keyword:
- If you want to tell the compiler that the user cannot and should not be able to change any data ==> include the const keyword
- Get and Set functions
> Get (Access): ==> functions that return values back to the caller (these methods are usually const)
> Set (Mutator): ==> functions that change the values of the data members (these methods can't be const)
==

INHERITANCE:
class Student : public Person
{};
> Forming new classes using classes that have already been defined
> "Is A" relationship ==> Then you can use inheritance
Ex: A dog is a mammal
> Inherit all of the methods/data of an existing class (you do not inherit the private variables directly)
- Enables us to define a derived class from a pre-existing class
- Avoids duplicate code
> You can have multiple instances of inheritance:
Ex: A computer science student is a student, a student is a person, a computer science student is a person
- Base Class: Person, Student
- Derived Class: Computer Science Student, Student

Three Uses of Inheritance:
1. Reuse: ==>Reduces duplication of code
> Every derived class inherits the public methods directly
> Derived classes cannot access private data members of the base class directly
- Must use the public methods
2. Extention: ==> Adds new behaviors to the derived class not present in the base class
> Adding new methods to the derived class
> Unknown to the base class (used later in polymorphism ==> a base pointer pointing to a derived class doesn't know about new functions)
3. Specialization: ==> Define existing behavior in the derived class with new behavior (overriding etc.)
> You can override existing functions that were present in the base class with new functions
- Must use the keyword virtual in the base class to do so

Virtual Keyword:
> When to use the virtual keyword?
- Only use the virtual keyword if you intend to override a function in the derived class
- If the function in the derived class does the same thing as the one in the base class, then don't need virtual
> Redefinition (overriding) a function hides the method in the base class from use
> NOTE: =====> Derived classes always use the most derived version of the method
Ex:
class Student   class NerdyStudent : public Student
{   {

public: public:
virtual void cheer() {} virtual void cheer() { cout << "Yeah" << endl; } ==> Derived version
}; void getExcided() {cheer(); }

{ Student::cheer(); } ==> Base verison
};

main()
{
NerdyStudent Katy;
Katy.getExcited(); ==> Uses the most derived version of cheer because it is virtual
==> If you want it to use the base version, then include Base::someFunction()
}

Constructon:
1. Construct the base clase members (and base class constructor)
2. Construct the derived class members in initializer list
3. Construct the body of the derived class

Destruction:
1. Destruct the body of the derived class
2. Destruct the data members of the derived class
3. Destruct the data members of the base class

Initializer List:
> If the base class has data members that need to be constructed, then the derived class must initialize the members
- This can be done in the initializer list (BaseName(type parameter1, type parameter2), derivedType parameter)
- As above, the initializer list consists of the base class named, followed by the parameters in parantheses

===

POLYMORPHISM:
WARNING::: ==> You must use a pointer or reference for polymorphism
WARNING::: ==> If you don't, this is known as chopping, and a totally new object that only consists of the base class is passed in
> If there is a function that accepts a reference or a pointer from a base class, then it also accepts (pointer or reference) of the derived class
- This is because everything that a base class can do, a derived class can do to
> Anytime you use a base pointer or reference to access a derived object, this is known as polymorphism
> When you call a virtual function, polymorphism allows the compiler to know which object the function is talking about
> If the virtual keyword is omitted, then only the base class function's will be called
> Note: If the function accepts a reference or pointer from a base class, and you pass in a derived object, then note that the function doesn't know about the methods of the derived class

----

When to use virtual?
1. When you want to override a function in the derived class
2. Always make destructors virtual
* if you don't then this is undefined behavior ==> perhaps a memory leak
3. You CAN'T make a constructor virtual

Pointers and Polymorphism:

> You can point base class pointers at derived class objects
> Never point a derived class pointer at a base class

Virtual Destructors and Polymorphism:

> Always make sure to make the destructors virtual when using polymorphism/inheritance
> Note: If you forget to make the destructor virtual, then this leads to undefined behavior

- The undefined behavior only occurs when using polymorphism (however, it is good practice, to make destructors virtual with inheritance as well)
- If virtual is forgotten, and you use a base pointer on a derived object, then when it is destructed, only the base destructor is called

----

Pure Virtual Functions:

virtual void someFunction() = 0; ==> This is a pure virtual function

> We must define functions that are common to all derived functions, however, they never need to be implemented

Ex: When we are using shape derived classes (e.g. circle, square, triangle), we never need to create an instance of a shape

> If you don't have functions that are common to all classes, then you can't use polymorphism
> Some functions in the base class are never actually used (the derived classes all override a given function)

Ex: shape's draw function (i.e. all shapes are drawn differently)

> Therefore, this unused function in the base class can be made pure virtual

----

Abstract Base Class:

> When a class has at least one pure virtual function, then it becomes an abstract base class
> You can't create an instance of an abstract base class
> You can still use base class pointers though
> If you define an abstract base class, then the derived class must:

1. Define its own function, that was pure virtual in the base class
2. Otherwise, the derived class also becomes an abstract base class

----

Polymorphism Cheat Sheet:

> You can't access private members of the base class from the derived class
> Use an initializer list to initialize the base class members in the derived class
> Always make the destructor virtual, if you are using inheritance/polymorphism

- Otherwise, if using a base class pointer to point to a derived class object, the base destructor will be called, but not the derived one

> Use virtual keyword in the base class for any function that you plan to override in the derived class
> To call a base class method that was overriden in the derived class, simply include the base::prefix before the function

----

Polymorphism Practice:

1. If you use a base pointer to access a derived object, and call a virtual function, then the most derived version of the function is called
2. When you use a base pointer to access a derived object, and call a non-virtual function, then the base class version is called
3. When you use a base pointer to access a derived object, all function calls to virtual functions will call the derived version of the function, even if the call occurs from a function that is not virtual

==

RECURSION:

Ex:

```
void mergeSort(an array)
{
	if (array size == 1)
		return;
	mergesort(first half of array);
	mergesort(second half of array);
	merge(both array halves);
}
```

Rules of Recursion:

1. Every recursive call must have a base case
   > Must be able to solve the simplest form of the problem (without recursion)
2. A recursive function must call itself
   > Therefore, there must be some way for the function to stop calling itself (base case)
3. A recursive function must have a simplifying step
   > If every recursive call didn't have a simplifying step, then it would run infinitely
   > Every recursive call must work on a smaller version of the original problem (either 1 less, or divide and conquer)
4. Must eventually reach its base case
   > If the recursive function never reaches its base case, then it would run infinitely
5. Never use global or static variables
   > You can and should use local variables (for each frame of the recursive call) if necessary

----

How to simplify each recursive call?

1. "Divide and Conquer"
   > Divide the input in half (like mergeSort)
2. Every call operates on input that is one less than the previous recursive call

----

Recursion on an Array:

> You need at least two parameters (one for the array, and one for its size)
> The base case would be to check if the size of the array is less than or equal to zero

----

Recursion on a Linked List:

> Similar to that of the array
> However, you only need one parameter, which is the linked list
> The base case would be to check if the list is empty (head == nullptr)
> Tips:
  - Only access the current node ==> Don't try to access every other node, or skip nodes
  - Don't access nodes that came before the current node, or after it

----

Examples:

```
int factorial (int n)
{
	if (n <= 0)
		return 1;
	return n * factorial (n - 1);
}

int sumArray(const int a[], int n)
{
```

```
        if (n == 0)
            return 0;
        int total = a[0];
        return total + sumArray(a + 1, n - 1);
    }

    void printArr (const int a[], int n)
    {
        if (n == 0)
            return;
        cout << a[0] << " ";
        printArr(a + 1, n - 1);
    }

    int findBiggest(Node* curr)
    {
        if (curr == nullptr)
            return -1;
        if (curr->next == nullptr)
            return curr->val;
        int biggest = findBiggest(curr->next);
        return max(largest, curr->val);
    }
```

==
==

BIG-O (TIME COMPLEXITY):

>

==
==

SORTING ALGORITHMS:

> The process of ordering a bunch of numbers based on one or more rules
- Items: What are we sorting, and how many to sort?
  (e.g. strings, ints, objects)
- Rules:
  > Ascending vs. Descending order
  > How are we sorting?
    (e.g. radius, last name)
- Constraints?
  > Are the items in memory or on a disk
  > Is the data in a linked list or an array?

Two rules for sorting:
1. Don't choose a sort method until you fully understand the requirement for the problem
2. Always choose the simplest sorting algorithm possible

==
==

INSERTION SORT:

Efficiency ==> O(N^2) ==> Slow

Ex: Sorting Books by Height
1. Start with the size s = 2
2. While there are still books to sort:
   > If the last book is in the wrong order
     - Remove it from the shelf
     - Shift the books before it to the right, if necessary

- Insert book into proper slot
3. s = s+ 1

Big-O:
> Each round, the algorithm needs to shift books to find the right spot
- 1 step in round 1
- 2 steps in round 2
...
- N - 1 steps in last round ==> O(N^2)

Code:
```
void insertSort(int a[], int n)
{
    for (int s = 2; s < n; s++)
    {
        int sortMe = a[n - 1];
        int i = s - 2;
        while (i >= 0 && sortMe < a[i])
        {
            a[i + 1] = a[i];
            i--;
        }
        a[i + 1] = sortMe;
    }
}
```
==

SELECTION SORT:

Efficiency ==> O(N^2); ==> SLOW

Ex: Sorting Books by Height
1. Look at all N books, and find the shortest book ==> O(N)
2. Swap this with the first book ==> O(1)
3. Look at the remaining N - 1 books, find shortest==> O(N - 1)
4. Swap this book with the second book ==> O(1)
5. Look at the remaining N - 2 books, find shortest==> O(N - 2)
6. Swap this book with the third book, and so on ==> O(1)

Big-O:
n + (n - 1) + (n - 2) + ... + 2 + 1 ==> N(N - 1)/2 ==> O(N^2)

Code:
```
void selectSort(int a[], int n)
{
    for (int i = 0; i < n; i++)
    {
        int minIndex = i;
        for (int j = i + 1; j < n; j++)
        {
            if (a[j] < a[minIndex])
                minIndex = j;
        }
        swap (a[i], a[minIndex]);
    }
}
```

==

BUBBLE SORT:

Efficiency: O(N^2) ==> SLOW

Algorithm:

1. Compare the first 2 elements, if they're out of order, swap them
2. Compare the next 2 elements (a[1] and a[2]), if they are out of order, swap tem
3. Repeat the process until you hit the end of the array
4. When you reach the end, if you made at least 1 swap, then repeat the process

----

Code:

```
void bubbleSort(int a[], int n)
{
	bool atLeastOneSwap;
	do
	{
		atLeastOneSwap = false;
		for (int j = 0; j < (n - 1); j++)
		{
			if (arr[j] > arr[j + 1])
			{
				swap(arr[j], arr[j + 1]);
				atLeastOneSwap = true;
			}
		}
	} while (atLeastOneSwap == true);
}
```

==

SHELL SORT:

Efficiency: O(N^2) ==> Slow

> Based on H-sorting:

H-sorting Method:

1. Pick a value of h
2. For each element in the array:
    > If a[i] and a[i + h] are out of order, swap them
3. If you swapped once, repeat the process

Note: If you 3-swap, then every element in the array is smaller than the element in the array that is 3 positions down

----

Algorithm:

1. Select a sequence of decreasing H-values, ending with an h-value of 1 (e.g. 8, 4, 2, 1)
2. First completely 8-sort the array
    - Then 4-sort it
    - Then 2-sort it
3. Finally 1-sort the array ==> 1-sort is bubble sort

==

MERGE SORT:

Efficiency: O(N log N) ==> Faster than the 4 slow sorts

Algorithm:

1. If the array has 1 element, return (it is sorted)
2. Split the array into 2 equal sections
3. Recursively call mergeSort on the left half
4. Recursively call mergeSort on the right half
5. Merge the two halves with the merge function

===

QUICK SORT:

Efficiency: O(N log N) ==> Faster than the 4 slow sorts
Worst Case: O(N^2) ==> If the elements in the array are nearly, or are sorted
Algorithm:

1. Select an arbitrary item p from the array
2. Move items smaller than or equal to p to the left, and larger items to the right
    - p (partition) goes in-between
3. Recursively repeat quickSort for left items
4. Recursively repeat quickSort for right items

===

HEAP SORT:

Efficiency: O(N log N)

Algorithm:

> Given an array of N items that need to be sorted:

1. Convert input array into a maxheap
2. While there are still items left in the heap:
    a. Remove the biggest value from the heap
    b. Place it in the last open slot of the array

===

GENERIC PROGRAMMING:

> We use generic programming, so that we are able to build algorithms that can operate on many different types of data (e.g. int, double, string)

Part 1 ==> Generic Comparisons:

> You can define a comparison operator for a class/struct

Ex:

```
class Dog
{
	public:
		int getWeight() const {return m_weight;}
		bool operator< (const Dog& other)
		{
			if (m_weight < other.m_weight)
				return true;
			return false;
		}
	private:
		int m_weight;
};
```

> Note:

- If you define the comparison operator in the function declaration, then only use one parameter
- If you define it outside then you have to use two parameters
- If you define it outside, then you also can't use private data members, you have to use public methods to compare

- If you define it outside, then you have to use constant public methods too because we are comparing constant references

```
bool Dog::operator< (const Dog& a, const Dog& b)
{
    if (a.getWeight() < b.getWeight())
        return true;
    return false;
}
```

Part 2 ==> Writing Generic Function:

> Before we would have to write separate functions for swap (for int, for double, for string)
> With templates, you can create a "generic" function that can be used by multiple types of data
> Templates are discussed in the section below

Part 3 ==> Creating Generic Classes

> You can use templates to make entire classes generic
> Use the prefix template<typename name> before the class definition
>Everywhere that there is a need to change the previous data type with the template name, then do so

----

Part 4 ==> The Standard Template Library (STL):

> It is a collection of pre-written, tested classes that are provided by C++
> These classes were all built using templates ==> Meaning that they can be used with different types
> Such classes for this course include "Container" classes (e.g. Stack, Queue, Vector, List, Map, Set)

Part 5 ==> STL Algorithms
#include <algorithm>

> STL also provides some additional functions that work with many different types of data
> STL "find", "find_if", and "sort"

STL find:

> Works with vectors/lists (they don't have built-in find functions like maps/sets)
> Three-parameter function

- First parameter is an iterator that is pointing to the front (where you want to start searching)

- Second parameter is an iterator that is pointing to the end (just after where you want to stop searching)

- Third parameter is what you are looking for

> If you couldn't find the value that you were looking for, then it will return the 2nd parameter (end())

Ex:

```
main()
{
    list<string> names;
    names.push_back("Carey");
    names.push_back("David");
    names.push_back("Alex");
    names.push_back("Rick");
    list<string::iterator front = names.begin();
    list<string::iterator end = names.end();
    list<string::iterator itr;
    itr = names.find(front, end, "Carey");
    if (itr == end)
```

```
            cout << "Not found!" << endl;
        else
            cout << "Found: " << *itr << endl;
    }
```

> Works with arrays as well

Ex:

```
    main()
    {
        int arr[4] = {1,2,3,4};
        int* ptr;
        ptr = find(a, &a[4], 2);
        if (ptr == &a[4])
            cout << "Not found!" << endl;
        else
            cout << "Found: " << *ptr << endl;
    }
```

STL find_if:

> Loops through a container/array and passes each item to the "predicate function" that you specify
> The predicate function must return a boolean value
> The predicate function must be of the same type (the parameters) as the container holds
> Provides a convenient way to locate an item in a set/map/vector/list/array based on specific criteria

> Three parameter function:
- First parameter: Points to the beginning of where you want to start searching
- Second parametere: Points to just after where you want to stop searching
- The name of the predicate function

Operation:
> Processes each item in the container until the predicate function returns true, or runs out of items

> find_if returns an iterator pointing to the first item that triggered a true value from the predicate function

Ex:

```
    bool is_even(int n)
    {
        if (n % 2 == 0)
            return true;
        return false;
    }
    main()
    {
        int arr[4] = {1,2,3,4};
        int* ptr;
        ptr = find_if(a, &a[4], is_even);
        if (ptr == &a[4])
            cout << "Not found!" << endl;
        else
            cout << "Found: " << *ptr << endl;
    }
```

STL sort:

> Works on arrays and vectors
> To sort, pass in two iterators (or pointers)
- First parameter: Iterator(pointer) that points to the start of where you want to sort
- Second parameter: Iterator(pointer) that points to just after the last item you want to sort

> It will sort all of the items in ascending order
> Or you can order it by passing in your own criteria

Ex:

```
main()
{
    vector<string> sortVector;
    sortVector.push_back("Carey");
    sortVector.push_back("David");
    sortVector.push_back("Alex");
    sortVector.push_back("Rick");
    vector<string>::iterator front;
    vector<string>::iterator end;
    sort(front, end);

    int arr[4] = {1,2,3,4};
    sort(arr, arr + 4);
}
```

Part 6: Compound STL Data Structures:

> Using several containers in the STL together

Ex: Maintain a list of courses for each UCLA student

- Use a map between student's name and their list of courses

```
class Course
{
    string name;
    string number;
};
main()
{
    Course c1("CS", "31");
    Course c2("math", "3b");
    Course c3("english", "1");
    map<string, list<Course> > courseMap;
    courseMap["Carey"].push_back(c1);
    courseMap["Carey"].push_back(c2);
    courseMap["David"].push_back(c3);
}
```

Ex: Associate people (Person object) with a set of friends

- Map from Person to a set of friends

```
class Person
{
    public:
        string getName();
        bool operator< (const Person& other)
        {
            return m_name < other.m_name;
        }
    private:
        int m_name;
};
main()
{
    Person p1 = "Alex";
    Person p2 = "David";
    map<Person, set<Person> > friendster;
    friendster["Carey"].insert(p2);
    friendster["Carey"].insert(p1);
}
```

==

TEMPLATES:

1. Add this line above the given function ==> template<typename name> or template<class name>
2. Then use the name as your data type throughout the function:

Ex:
```
void swap(T& x, T& y)
{
    T temp = x;
    x = y;
    y = temp;
}
```

Template Rules:

> Always place templated functions in the header file (not the source file)
> Then include the header file or source files
> Each time a templated function is used, the compiler will generate a new version of the function
> You must use the data type to specify the type of at least one of the parameters
Ex:
```
template<typename Data>
Data getRandomItem(int x) ==> WRONG!! ERROR! ==> Should be Data getRandomItem(Data x)
```
> If a function has 2 or more "templated parameters", then you must pass in the same type of variable for both
Ex:
```
template<typename Data>
Data max(Data x, Data y)
{}
main()
{
    int i = 5;
    int j = 6;
    float k = 7.5;
    cout << max(i, j) << endl; ==> Correct!
    cout << max(i, k) << endl; ==> WRONG!! ERROR! ==> Both parameters are of the same templated type, so you must pass in the same type
}
```

Overriding a Template Function:

> You can override a template function with a specialized (non-templated function)
> If C++ sees a specialized version of a function, it will always choose the specialized function over the templated function

----

Problems to Avoid with Templates:

> If you use a comparison operator on templated variables... then c++ expects that all variables passed in have the comparison operator defined for that given class
> If you use a user-defined class, and a templated function uses a given comparison, then the class must have that comparison operator defined

----

Creating Templated Classes:

Ex: Stack of Dogs
```
template<typename Item>
class Stack
{
    public:
        Stack() {m_top = 0;}
```

```
            void push(Item v) {m_item[m_top++] = v;}
};
main()
{
        stack<Dog> stackOfDogs;
        Dog fido;
        stackOfDogs.push(fido);
}
```

Template Cheat Sheet:

> To templatize a non-class function called bar:

- Update the function header

```
template<typename ItemType>
ItemType bar(ItemType a)
```

- Replace the appropriate types in the function to the new ItemType:

```
{int a, float b;} ==> {ItemType a, float b;}
```

> To templatize a class called foo:

- Add template<typename ItemType> in front of the class declaration
- Update the appropriate types in the class with the new ItemType
- Update internally defined functions
- Update externally defined functions

==

INLINE METHODS:

> When you use an inline function, you are telling the compiler to directly embed the functions logic into the calling function

> By default, all methods, with their bodies, defined directly in the class (.h file) are made inline

> To make an externally-defined method (.cpp file) inline, simply add the keyword inline before the function return type

Ex:

```
inline ==> Include the inline keyword right before the function
void LinkedList::getNode() const {}
```

==

STANDARD TEMPLATE LIBRARY (STL):

> It is a collection of pre-written, tested classes that are provided by C++

> These classes were all built using templates ==> Meaning that they can be used with different types

> Such classes for this course include "Container" classes (e.g. Stack, Queue, Vector, List, Map, Set)

Deleting Items from STL Containers:

> Most STL containers have an erase() method to delete an item

> First search for the item that you want to delete, and get an iterator to it

> Then, if you found the item, use the erase() method to remove the item

Ex: Set

```
main()
{
        set<string> geeks;
        geeks.insert("Carey"):
        geeks.insert("Alex");
        geeks.insert("Rick");
        set<string::iterator it;
        it = geeks.find("Carey");
        if (it != geeks.end())
                geeks.erase(it);
```

```
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```