Topics List for Exam 1

- Material from 3110
    - Classes:
        - **Accessors and mutators (aka getters and setters)**

          setter→ void setLastName (string last_name);

          getter→ string getLastName()const;
        - **Private / public**
        - **What the =operator and copy constructors do by default and why this causes a problem with classes that have pointers in them.**

          ```
          // Copy constructor
             Point(const Point &p2) {x = p2.x; y = p2.y; }
          ```

          We need to define our own copy constructor only if an object has pointers or any run time allocation of resource like file handle, a network connection..etc.

```
    //copy constructor, creates an el on the left hand side
     Array(const Array& rhs)
     {
          numbers = new int[rhs.numbers];
          numEl= rhs.numEl;
          for (int i=0; i<numEl; i++)
                    numbers[i]=rsh.numbers[i];
      }
//programmer defined copy constructor
NumberArray::NumberArray(const NumberArray &obj)
{
     arraySize=obj.arraySize;
     aPtr=new double[arraySize];
     for(int index=0;index<arraySize;index++)
            aPtr[index]=obj.aPtr[index];
}

//assignment operator
NumberArray& NumberArray: :operator =(const NumberArray &right )
{
       if (this != &right )
      {
         i f (arraySize > 0)  { delete[] aPtr;  }
         arraySize = right.arraySize ;
         aPtr = new double [arraySize ];
         for (int index = 0; index< arraySize; index ++}
                aPtr[index] = right.aPtr[index];
      }
      return *this;
}
```

- **Know how to implement constructors using member initialization lists.**

```
class Array{
        int* numbers;
        int numEl;
 public:
```

```
        Array(int size): numbers(new int[size]), numEl(size){}
};
```

```
 // With Initializer List
 class MyClass {
    Type variable;
 public:
    MyClass(Type a):variable(a) {   // Assume that Type is an already
                // declared class and it has appropriate
                // constructors and operators
    }
```

▪▪ **Know when you would need a destructor.**

→ program uses dynamic memory allocation

The **rule of three** (also known as the Law of The Big Three or The Big Three)

- destructor
- copy constructor
- copy assignment operator

▪▪ **What pass by value, pass by reference, and pass by const reference do, and why you would use/not use each one.**

Pass by Value - does member wise assignment/copying (one by one)
- e.g void printPlayer (Player P);

When an argument is passed by **value**, the argument's **value** is copied into the function's parameter.

Pass by Reference - allows modification of the original value (no copies)
- e.g. void printPlayer (Player& p);

Pass-by-reference means to pass the reference of an argument in the calling function to the corresponding formal parameter of the called function. The called function can modify the value of the argument by using its reference passed in.

Pass by Const Reference - doesn't allow modification and avoids making a copy
- e.g. void printPlayer (const Player& p);

When you pass by **const reference**, you take the argument in by reference (avoiding making any copies of it), but cannot make any changes to the original object (much as would happen when you would take the parameters in by value).

▪▪Operator overloading

- **Know how to overload the Boolean operators (==, !=, >, <, >=, <=)**

```
bool operator ==(Length a, Length b)
{
        return a .len _inches == b.len _inches;
}

bool operator< (Length a, Length b)
{
        return a .len _inches < b.len _inches;
}
```

- **Know how to implement the [] operator**

```
int operator [ ] (string num_str)
{
 for (int k = 0; k < numerals .size(); k++) {
            if (numeral s[k ] == num _str)  { return k; }
 }
 return -1 ;
 }
```

- **<u>Know how to implement the << operator</u>**

```
ostream &operator<<(ostream& out , Length a)
{
          out<< a.getFeet() <<" feet , "<< a.getInches() <<" inches";
          return out;
 }
```

- **<u>Know how to implement the + operator</u>**

```
Length operator+ (Length a , Length b)
{
       return Length(a.len _inches + b.len _inches);
}
```

- o Recursion
  - **<u>How to write a simple recursive algorithm</u>**

```
int factorial(int n) {
          if (n == 0)                    // this if statement serves as the function's base case
               return 1;
          return n* factorial (n - 1);
}
```

  - How to trace through a recursive algorithm

- Vectors
  - o STL vectors

<u>VECTOR FUNCTIONS</u>: built in: pop_back(); push_back(); pop_front(); push_front(); size(); empty(); [], at();
not built in: erase(), insert();

```
void insert(int index, int value)
{
   if (allocLen==logicLen)
       grow();
  for (int i=logicLen; i>=index+1; i--){      or       for (i=logicLen-1; i>=index; i--)
          data[i]=data[i-1];                                  data[i+1]=data[i];
          data[index]=value;
          logicLen++; }
}

void erase(const Contact& contact)
{
   int index = findContact(contact);
   if(index==-1) return;
   else
   {
       vec[index] = vec.back();
       vec.pop_back();
   }
}
```

▪ ▪ **<u>How to create a vector with an initial size</u>** → **vector<int> numbers(10);**

```
 vector<int> first;                          // empty vector of ints
 vector<int> second (4,100);                     // four ints with value 100
 vector<int> third (second.begin(),second.end());  // iterating through second
 vector<int> fourth (third);                      // a copy of third
// the iterator constructor can also be used to construct from arrays:
 int myints[] = {16,2,77,29};
 vector<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );

 cout << "The contents of fifth are:";
 for (vector<int>::iterator it = fifth.begin(); it != fifth.end(); ++it)
   cout << ' ' << *it;
```

<u>Output:</u>

```
The contents of fifth are: 16 2 77 29
```

▪ ▪ **<u>How to create an empty vector</u>**

<u>vector<int> first;</u>

▪ ▪ **<u>How to initialize a vector using an array.</u>** (You'll need to recognize this, but you won't need to write it)

int arr [] = {1, 2, 3, 69};
vector<int> numbers(array, array+4);
*(Type *begin, Type *end) constructor.*Last position is meant to be out of bounds

▪ ▪ **<u>What the operations push_back(), pop_back(), back(), size(), empty(), and the [] operator do.</u>**

push_back() - Adds a new element at the end of the <u>vector</u>, after its current last element. Also adds one to the size and changes capacity of vector. vector.back() changes as well.
pop_back() - Removes an element from the end of the vector, effectively subtracting one from vector.size() and changing the element from vector.back() (destroys the element)
[]- Returns a reference to the element at position *n* in the <u>vector</u> container.
A similar member function, <u>vector::at</u>, has the same behavior as this operator function, except that <u>vector::at</u> is bound-checked and signals if the requested position is *out of range* by throwing an <u>out_of_range</u> exception.
back() - Returns a reference to the last element in the <u>vector</u>.
Unlike member <u>vector::end</u>, which returns an iterator just past this element, this function returns a direct reference.
Calling this function on an <u>empty</u> container causes undefined behavior.
empty() - Returns whether the <u>vector</u> is empty (i.e. whether its <u>size</u> is 0).
This function does not modify the container in any way. To clear the content of a <u>vector</u>, see <u>vector::clear</u>.
size() - Returns the number of elements in the <u>vector</u>.
This is the number of actual objects held in the <u>vector</u>, which is not necessarily equal to its storage <u>capacity</u>.

▪ ▪ **How to write functions that go through a vector, performing some operation (e.g. printing out a whole vector, summing up a vector, whatever).**

```
//to print out a vector
void printVector (const vector<int>& aVec)     O(n)
{
   for( int i=0; i<aVec.size(); i++)
        cout<<aVec[i];
}

//to sum a vector
int fib(int n)
{
   vector<int> answers(n+1);
   answers[1]= answers[2]=1;
   for(int i=3; i<n; i++)
      answers[i]= answers[i-1]+ answers[i-2];
   return answers[n];
}
```

o Our own implementation

▪ ▪ **How did we implement a vector? (You don't have to implement one from scratch on the test, but you should know what we did)**

```
//h file
#ifndef MYVECTOR_H
#define MYVECTOR_H
#include <iostream>

class MyVector {
   int* arr;
   int capacity;
   int numElements;

   //This function will be called if the array runs out of space and needs to be resized.
   void grow();

public:
   //Constructors
   MyVector();
   MyVector(int n, int initialValue = 0);

   //Destructor
   ~MyVector();

   //returns the number of elements in the vector
   int size();

   //returns whether or not the vector is empty
   bool empty();
   //returns the element at the end of the vector
   int& back();

   //returns the element of the vector at position "index," and throws an exception
```

```
    //if the index is out of bounds
    int& at(int index);

    //same as at() but doesn't do any bounds checking
    int& operator[] (int index);

    //adds "number" to the end of the vector.
    void push_back(int number);

    //removes the element from the end of the vector.
    void pop_back();

    //printing function

    friend std::ostream& operator<<(std::ostream& os, MyVector& rhs);
};

//cpp file
#include <iostream>
#include <string>
#include "MyVector.h"
#include "IndexOutOfBoundsException.h"
#include "EmptyVectorException.h"

//Constructors
MyVector::MyVector(): numElements(0), capacity(10), arr(new int[10]) {}

MyVector::MyVector(int n, int initialValue):numElements(n), capacity(n), arr(new int[n]) {
    for(int i=0; i<n; i++)
        arr[i] = initialValue;
}

//Destructor
MyVector::~MyVector() { delete [] arr; }


//returns the number of elements in the vector
int MyVector::size() { return numElements; }

//returns whether or not the vector is empty
bool MyVector::empty() { return numElements==0; }


//returns the element at the end of the vector
int& MyVector::back() {  return arr[numElements-1]; }

//returns the element of the vector at position "index," and throws an exception
//if the index os out of bounds
int& MyVector::at(int index) {
    if(index < 0 || index >=numElements) {
        std::string message = "In MyVector::at(), index supplied is out of bounds.";
        throw IndexOutOfBoundsException(message);
    }
    return arr[index];
}

//same as at() but doesn't do any bounds checking
int& MyVector::operator[] (int index){ return arr[index]; }

//adds "number" to the end of the vector.
void MyVector::push_back(int number) {
    if(numElements == capacity)
        grow();
    arr[numElements] = number;
    numElements++;
}

//removes the element from the end of the vector.
void MyVector::pop_back() {
    if(empty()) {
        std::string message = "MyVector::pop_back() was called on an empty vector. ";
        throw EmptyVectorException(message);
    }
    numElements--;
}

//This function will be called if the array runs out of space and needs to be resized.
void MyVector::grow() {
    int* temp = new int [capacity*2];
    for(int i=0; i<numElements; i++)
        temp[i] = arr[i];
    capacity*=2;
    delete[] arr;
    arr = temp;
}

//printing function
std::ostream& operator<<(std::ostream& os, MyVector& rhs) {
    os<<"[ ";
    for(int i=0; i<rhs.numElements; i++)
        os<<rhs.arr[i]<<" ";
    os<<"]";
    return os;
}
```

- **What is the running time of push_back(), pop_back(), back(), size(), empty(), and the [] operator?** O(1)

- Lists
  - **STL lists built-in functions**→ built in: pop_back(); push_back(); pop_front(); push_front(); size(); empty();back();front(); erase(), insert();
    for iterators: begin(); end();

representing a function **advance(it,2)**//moves to a 3rd el in a list
*it=10; //{0,0,10,0,0}

- **How to create a list with an initial size.**

list<type> name(int);   //MUST be an int of course, as this represents the number of nodes in this linked list. You CAN'T half 2 and half nodes!

- **How to create an empty list.**

list<type>name;

- **How to initialize a list using an array.** (You'll need to just recognize this.)

```
struct Test { Test() : set { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 } { }; int set[10]; };
```

int arr [] = {1, 2, 3, 69};
list<int> numbers(array, array+4);

- **What the operations push_back(), pop_back(), back(), size(), empty(), push_front(), and pop_front() do.**

push_back() - adds a new element 'g' at the end of the list
- exampleList.push_back(g);   O(n) find the end (with tail O(1))

pop_back() - removes the last element of the list, and reduces the size of the list by 1
- exampleList.pop_back();   O(n)

back() - returns reference to the last element in the list
- exampleList.back();   O(1)

front() - returns reference to the first element in the list   O(1)
size() - returns the number of elements in the list
- exampleList.size();   O(1)

empty() - returns whether the list is empty(1) or not(0)
- exampleList.empty();   O(1)

push_front() - adds a new element 'g' at the beginning of the list
- exampleList.push_front(g);   O(n), but would like it to be O(1)

pop_front() - removes the first element of the list, and reduces size of the list by 1
- exampleList.pop_front();   O(1)

- **How to write functions that go through an entire list using iterators, performing some operation.**

```
list<int> numbers(5, 10);
list<int>::iterator it;
it = numbers.begin();
for (it=numbers.begin(); it!=numbers.end(); it++)
        cout<<*it;
```

o   Our Own list implementation

- **How did we implement a list?**

```
#include "MyList.h"
#include "EmptyListException.h"

#include <iostream>
```

```
#include <cstddef>      //the <cstddef> header file allows us to use the keyword NULL.
#include <string>

//Constructors
//default constructor.
//This will set the head and tail pointers to NULL, and "numElements" to 0,
//signifying that there are no elements in this list yet.
MyList::MyList():  head(NULL), tail(NULL), numElements(0) {}

//This is the second list constructor.
//I will use the push_back() function that I will write later in order to add the elements.
//the push_back() function will automatically increment the size.

MyList::MyList(int n, int initialValue): head(NULL), tail(NULL), numElements(0)
{
   for(int i=0; i<n; i++)
      push_back(initialValue);
}

//Destructor
//In our case, the destructor function must deallocate the memory associate with all of the nodes
//that are still in the list.  The easiest way to do this is using recursion.
//Unfortunately, you aren't allowed to call the destructor recursively,
//so we will write a recursive private and static function so that we can use recursion.
//(it's private because we don't want a client to call this code, and static because technically
//it isn't supposed to be called on any individual object.)
MyList::~MyList(){ removeAllElements(head); }

//returns the number of elements in the list
int MyList::size(){ return numElements; }

//returns whether or not the list is empty
bool MyList::empty(){ return numElements==0; } //or head==NULL or tail==NULL. All 3 of these work.

//returns the element at the end of the list.
//This is the "data" field of the ListNode pointed at by "tail"
//if the list is empty, we will throw an EmptyListException
int& MyList::back()
{
   if(empty()) {
      std::string message = "In MyList::back(), trying to access the back of an empty list";
      EmptyListException error(message);
      throw error;
   }
   return tail->data;
}
```

```
//returns the element at the front of the list.
//This is the "data" field of the ListNode pointed at by head.
//if the list is empty, we will throw an EmptyListException.
int& MyList::front()
 {
   if(empty()) {
      std::string message = "In MyList::front(), trying to access the front of an empty list";
      EmptyListException error(message);
      throw error;
   }
   return head->data;
}

//adds "number" to the end of the list.
void MyList::push_back(int number)
{
//no matter what, we have to create a new ListNode,
//            copy "number" into its data field
//            and set its next field to NULL.
//We also have to increment the number of elements.
   ListNode* newNode = new ListNode;
   newNode->data = number;
   newNode->next = NULL;

   //What happens next depends on a number of things.
   //we have to consider a few cases.
   //if the list is originally, empty, the new node becomes both the head and tail of the list.
   //Therefore, the node's previous field has to be set to NULL.
   if(empty()) {
      head=newNode;
      tail=newNode;
      newNode->previous = NULL;

   //suppose instead that the list already has elements.
   //Then, the tail's next field has to be set to the newNode,
   //     the newNode's previous has to be set to the tail,
   //     and the tail pointer has to be updated to point to the newNode.

   }else {
      tail->next = newNode;
      newNode->previous = tail;
      tail = newNode;
   }
   numElements++;
}

//adds "number" to the beginning of the list.
```

```
void MyList::push_front(int number)
{
  //no matter what, we have to create a new ListNode,
  //            copy "number" into its data field
  //            and set its previous field to NULL.
  //We also have to increment the number of elements.
  ListNode* newNode = new ListNode;
  newNode->data = number;
  newNode->previous = NULL;
  //What happens next depends on a number of things.
  //we have to consider a few cases.
  //if the list is originally, empty, the new node becomes both the head and tail of the list.
  //Therefore, the node's next field has to be set to NULL.
  if(empty()) {
    head=tail=newNode;
    newNode->next = NULL;
  //suppose instead that the list already has elements.
  //Then, the head's previous field has to be set to the newNode,
  //     the newNode's next has to be set to the head,
  //     and the head pointer has to be updated to point to the newNode.
  }else {
    head->previous = newNode;
    newNode->next = head;
    head = newNode;
  }
  numElements++;
}

//removes the element from the beginning of the list
void MyList::pop_front() {
  //we must consider a few cases.
  //First of all, if the list is empty, we must throw an exception.
  if(empty()) {
    std::string message = "In MyList::pop_front(), trying to pop from the front of an empty list.";
    EmptyListException error(message);
    throw error;
  }
  //Now, if the list was not empty,
  //we have to create a temporary pointer to hold onto the element after the front of the list.
  //this value will be equal to NULL if the list only has 1 element.
  ListNode* temp = head->next;
  //deallocate the memory associated with head.
  delete head;
  //we must decrement the number of elements.
  numElements--;
  //if there are no elements left in the list, we must set head and tail to NULL
  if(temp==NULL) {
        head=NULL;
        tail=NULL;
    }
    else {
        //Otherwise, we must set temp's previous field to NULL, since the head is no longer in the list
        temp->previous = NULL;
        head=temp;
    }
}

//removes the element from the end of the list.
void MyList::pop_back() {
    //we must consider a few cases.
    //First of all, if the list is empty, we must throw an exception.
    if(empty()) {
         std::string message = "In MyList::pop_back(), trying to pop from the back of an empty list.";
        EmptyListException error(message);
        throw error;
    }
    //Now, if the list was not empty,
    //we have to create a temporary pointer to hold onto the element before the end of the list.
    //this value will be equal to NULL if the list only has 1 element.
    ListNode* temp = tail->previous;
    //deallocate the memory associated with tail.
    delete tail;
    //we must decrement the number of elements.
    numElements--;
    //if there are no elements left in the list, we must set head and tail to NULL
    if(temp==NULL) {
        head=NULL;
        tail=NULL;
    }
    else {
        //Otherwise, we must now set temp's next to NULL, since the tail is no longer part of the list.
        temp->next = NULL;
        //now we have to update the tail pointer to point to where the temporary pointer points to.
        tail = temp;
    }
}

//printing function
std::ostream& operator<<(std::ostream& os, MyList& rhs) {
//let's declare a temporary pointer that will iterate through the list. This pointer will be updated
// by following the next pointer down the list.
    MyList::ListNode* temp = rhs.head;
    while(temp!=NULL) {
          os<<temp->data<<" -> ";
        temp = temp->next;
    }
    os<<"END";
    return os;
}

//this will be a recursive helper function that will allow us to easily delete
//all the dynamically allocated ListNodes. The reason that this function has to be written is because
//you aren't allowed to call the destructor recursively.
//This function is also static, because it technically isn't called on any particular MyList object.
void MyList::removeAllElements(ListNode* head) {
    //if the list is empty, there is no need to do anything
    if(head==NULL)
        return;
    //otherwise, use removeAllElements()
    //to recursively deallocate all the elements past the head of the list
    removeAllElements(head->next);
    //then deallocate the head
    delete head;
}
```

- **Know how to write code to add/remove an element from the head/tail of a linked list structure. ^**
- **What is the running time of push_back(), pop_back(), back(), size(), empty(), push_front(), and pop_front() using a linked list implementation.** O(1)

- Analysis of Algorithms
    - Know the definitions for the following:
        - **Running time (or time complexity) →** # of primitive operations an algorithm performs in order to solve a problem
        - **Space complexity→** how much of an extra space an algorithm requires
        - **Algorithm→** step by step process to solve a particular problem. This process must halt.
        - **Correctness of an algorithm→** the algorithm solves the given problem for all possible inputs. Requires math proof.
    - **Know how to analyze the running time of a simple algorithm and explain why that is the correct running time. +**
    - **Know how linear search and binary search work and which is better in which situations +**

linear takes linear time O(n), binary takes algorithmic time Theta(log n)

binary→

```
int binarySearch(int arr[], int l, int r, int x)
{   while (l <= r)
    {   int m = l + (r-l)/2;
        if (arr[m] == x) return m;
        if (arr[m] < x) l = m + 1;
        else r = m - 1;
    } return -1;
}
```

linear→ `for (i=0; i<31; i++) {  if (item==arr[i]) cout << "Item is found at location " << i+1 << endl;  }`

- Stacks (last in first out)
  - **Know how to create an empty stack**

  stack <type> name; //only an empty stack can be created
  - **Know what push(), pop(), empty(), and size() do.** O(1)

  - void push(type x); //adds top of the stack
  - type&top() //stk.top(); // allows read/write top most element
  - void pop(); //removes top most element
  - bool empty(); //checks if empty
  - int size();
  - **Be able to solve a problem using a stack.+**

Questions that are going to be on the test:

1. all definitions: algorithm, time complexities, etc…….
2. draw a vector and a list
3. comment what does a list look like, what is the order of a list, what gets printed after each line. stl list vs list node (singly or doubly linked)
4. iterating through a vector, iterating through a list
5. linear vs binary search
6. algorithm, how long each one takes
7. from the previous course: smth that requires a copy constr and an assignment operator (if a program has those- why given don't work and you gotta fix it)
8. questions are based on material in class
9. code with a stack-> answer what it looks like or asked to solve a problem with (using?) stack

Mistakes made by people in the previous semester:
for a list node

- declare a list node pointer, set it to a new list node
- set data equal to the data
- set nest to null
- set tails

mistake 2: don't just memorize and use it inappropriately

INSERTING IN A LIST GIVEN THE RIGHT AND LEFT NODE TO THE INSERT SPOT

With objects passed by reference:

```
Void insertNode(ListNode& lNode, ListNode& rNode, ListNode& newNode){

        // setting the left node to point to the new one
        lNode.next = newNode;

        // setting the right one to point to the new one
        rNode.previous = newNode;

        // directing the newNode's next to the right node
        newNode.next = rNode;

        // directing the newNode's previous to point to the left
        newNode.previous = lNode;
}
```

Passed with Pointers:

```
Void insertNode(ListNode* lNode, ListNode* rNode, ListNode* newNode){

        // setting the left node to point to the new one
        *lNode -> next = newNode;

        // setting the right one to point to the new one
        *rNode -> previous = newNode;

        // directing the newNode's next to the right node
        *newNode -> next = rNode;

        // directing the newNode's previous to point to the left
        *newNode -> previous = lNode;
}
```