Midterm contains a total of 120 points.
Time allotted is 2 hours (normal class length)
Please state any assumptions and show your work.
(Note, you do not need to speak or address the characters below 😊)

Section A

1. **Michelangelo** raises his hand in his Comp 15 class and asks “what are some tradeoffs one makes when creating a data structure?”
Describe to **Michelangelo** some of the tradeoffs one must make when creating a data structure? (5 points)

- Space complexity
- time complexity
- efficiency (startup time, allocations, destruction, power used etc.)
- elegance
- ease of implementation
- security
- expressiveness (think about lab 3 and using ints versus Jumbo0
- convenience of using API (array versus linked list for n-dimensional data for example)

2. **Raphael** is stubborn and says he likes to only code in the ‘C’ language. **Donatello** tells **Raphael** that he should start using C++, because Object-Oriented Programming can be very useful.
What are some of the reasons **Donatello** can state that motivate **Raphael** to use Object-Oriented Programming? (5 points)

- Encapsulation
- Security
- Hiding implementation details
- Inheritance
- Simplicity
- Consistency of API

3. **April** will win $1-million dollars if she can correctly identify the asymptotic complexity of the following algorithms.
**Given**: \( f(n) = 2n^2 + \log(n) \), what can April say about \( f(n) \)? (Circle one or more of the following) (5 points)

a.) \( \Omega(\log(n)) \)  

b.) \( \Theta(n^2) \)  

c.) \( O(n^2) \)  

d.) \( O(n^3) \)

(See solutions attached at end for how to prove formally)

**Given**: \( f(n) = 2n + 2n\log(n) \), what can April say about \( f(n) \)? (Circle one or more of the following) (5 points)

a.) \( \Omega(\log(n)) \)  

b.) \( \Theta(n) \)  

c.) \( O(n\log n) \)  

d.) \( O(n^3) \)
Yes, this is the first step.

$$\log(n) = n$$

$$2^{n-1} \cdot \log(n) = 3n^2$$

$$c = 100$$

Let's simplify the left-hand side.

We must show:

$$c \cdot n^2 \leq 2^{n-1} \cdot \log(n) \leq c \cdot n^2$$

For that, we must show that $2n^{2\log(n)}$ can be simplified further.

Let $n^2 = f(n)$, we can see that

$$f(n) = 2^{n-1} \log(n) \cdot n \log(n)$$

So far, our function gets $c = 1$, not constant.

The function, i.e., it will never get from better than $2 \log(n)$

Some constant, c, are still not clear.

Eventually, the whole series become more say the left hand side (see

$$\frac{\log(n)}{2n^2 + \log(n)}$$

So we need to find a $c$ such that

such that $g(n) = 2 \log(n)$ for all $n \geq n_0$.

For each such a positive real constant $c$, and a positive integer $n_0$,

Formal definition:

Let's now prove $2n^{2\log(n)}$ is $O(n^2)$.
Let $c = 2$.

So we try to show $c = 2$.

Thus, it is not the while that each other.

$2n^2 \log_2(n) \geq 2n$

Every $n \times 2k$.

Then each poison real animals $c$ such that $0 < a \leq c < b$.

Given: $f(x) = 2n \log_2(n)$

Remembers: $\log_a n > n > \log_{a^2} n$.

If $c$ is an integer.
Let \( c^2 = 50,000 \)
\[
\sqrt{c^2} = 5000
\]
\( c \in 2n + 2n \log(n) \Rightarrow c^2 \in 4n^2 + 4n^2 \log^2(n) \)
\( n \geq 100 \Rightarrow 4n^2 + 4n^2 \log^2(n) \leq 500,000 \)
\( n \leq 21.1819 \Rightarrow 3n^2 + 2n \log(n) \leq 500,000 \)
\( \left\lfloor \frac{1}{3} \right\rfloor = 3 \)

How about \( \Theta(n) \), so we want to show

(Coffee some proof is a very hard part)

tight bound

prove \( O(n) \), we know \( \Theta(n) \) \( \leq \Theta(n) \), so also free, but not a

\( \Theta(100) \leq \Theta(n) \)

You're log(n) is a lower bound for \( 2n + 2n \log(n) \)
\[
\log(n) \Rightarrow 2n + 2n \log(n)
\]
\( \log(n) \leq 2n + 2n \log(n) \)
\[
\text{From: } \forall (\log(n))
\]
\[
\text{with: } \forall (2n + 2n \log(n))
\]
What are the tradeoffs of using a linked list versus using an array to store and lookup data? (5 points)

<table>
<thead>
<tr>
<th>Linked list</th>
<th>Arrays</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pros:</strong></td>
<td><strong>Pros:</strong></td>
</tr>
<tr>
<td>1.) Expandable, don’t need to know exact amount of data needed allocated</td>
<td>O(1) lookup time, can perform compile-time error checking or other analysis easily.</td>
</tr>
<tr>
<td>2.) Can restructure data (reverse, traverse, copy, delete, insert in the middle, detect cycles, perhaps other interesting operations?)</td>
<td></td>
</tr>
<tr>
<td>3.)</td>
<td></td>
</tr>
<tr>
<td><strong>Cons:</strong></td>
<td><strong>Cons:</strong></td>
</tr>
<tr>
<td>O(n) lookup time, can have cycles or other errors</td>
<td>Difficult and expensive to resize an array.</td>
</tr>
</tbody>
</table>

5. *Harry* is impatient and thinks writing the word “*const*” is not necessary, but *Dumbeldore* insists that it is important.  
Discuss the different usages of const keyword in C++. (5 points)

Basic usage – much better than #define(which does a pure text replacement), because compiler can understand the actual item and do some error checking. Also is used to make a variable value ‘constant’

Function return values – ensure that we cannot alter return values after compilation e.g. const char *Funct() return “text”;

OOP – Stop methods from modifying memover variables e.g. class_method() const

Parameter passing – void foo(int &param) // parameter can be modified here

            void foo(int const & param) // passes variable without making a copy and allows us to maintain the original value
6. Hermione and Ron are studying sorting algorithms, and Ron gets confused about the term ‘in-place.’
What does it mean if a sorting algorithm occurs ‘in place’? Name at least 2 sorting algorithms that sort in-place. (10 points)

In-place means without allocating any extra memory (or using only a small fixed amount).

Insertion sort and selection sort are in-place. I was very lenient, because you can implement almost every sorting algorithm using pointers as an in-place algorithm using points and additional space. For example, you could have an in-place ‘merge-sort’ implementation (although it would cost more space for the original merge-sort algorithm).
Section B

1. Harry and his friends are studying Comp 15 together, and have a good laugh when they discover recursion is like magic. After practicing some problems though, they understand that it is very mathematical and quite elegant for solving problems like the Towers of Hanoi.

Write a recursive function (in a C++ routine or a pseudo-code) that determines whether a string of characters is a palindrome. A palindrome is a word or phrase that reads the same when you reverse it.

Examples:
- ‘dad’ reversed is still ‘dad’
- ‘aa’ reversed is still ‘aa’
- ‘a’ reversed is still ‘a’
- The empty string reversed is still the empty string

(20 points)

You needed the correct return type (bool), a function that took in a string, the correct base case, and somewhere to return true or false.

Pseudo code:

Bool palindrome(string s)
If(s.empty || s.length==1)
    Return true;
Else
    If(s[pos]==s[s.length-1])
        Palindrome(s.substring(1,s.length-1))
    Else
        Return false

You likely lost points if you did not cover the empty string or strings of length one. If you returned too soon, you may have lost points. If you did not break your string on each size by 1 you lost points. If you thought strings of two were palindrome’s you likely lost points.
2. Why and when do we write a copy constructor? What happens (or what are the problems that occur) when you do not write a copy constructor? (10 points)

Why and when we need it: deep copy (when we have points and other dynamically allocated structures)

What happens: Compiler gives us a default that may not work (if we only use primitive data types it may work)

3. Draw the binary search tree with the elements 5, 3, 2, 9, 8, 6, 1, 7 inserted into their correct positions. Write down the postorder and inorder traversal of the resultant tree. How would you express the time complexity of a “find” in a Binary search tree? (15 points)
In-order: 1,2,3,5,6,7,8,9
post-order: 1,2,3,7,6,8,9,5

Worse case -> O(n)
average case -> theta(n)
Best case -> Omeag(log (n))
Enough people put “log(n)” that I gave you the benefit of the doubt. Remember, it is only log(n) in the average case if the tree is balanced (we will learn how to balance a tree later). We get the theta(n) or best case, when we have log(n) levels, thus can do a find much more quicker than O(n).
Remember, that our tree becomes essentially an inked list if we add items in already sorted order, which we know takes linear time to search.

Section C

1. Write a member function (linked_list::reverse) that reverses a linked list. You are given the following class definition for your linked list. (15 points)

```cpp
class node {
public:
    int data() const { return mData; }
    node* next() { return mNext; }
    void setNext(node* aNext) { mNext = aNext; }
    ...
private:
    int mData;
    node* mNext;
};

class linked_list {
public:
    node* head() { return mHead; }
    void reverse();
    ...
private:
    node* mHead;
};

void linked_list::reverse()
```
Recursive:

node* prevNode = NULL;
node* currNode = mHead;
while(currNode != null{
    node* nextNode = curNode->next();
    currNode->setnext(prevNode);
    prevNode = currNode;
    currNode = nextNode
}
Mhead = prevNode

Iterative:

node* firstNode = mHead;
node* rest = mHead->next();
if(rest == NULL)
    return;
    mHead = rest;
    reverse();
    rest->setNext(firstNode);
    firstNode->setNext(NULL);

If you had most of the concept right, you should have gotten most or full credit. If you forgot something critical, like setting prevNode to Null, you likely lost points. If you forgot something tiny like a star in one place somewhere, you should not have lost any points (as long as it was clear it was a node).
2. Implement (in pseudo-code) the operations enqueue and dequeue to simulate the abstract data type ‘queue’. You are given the stack data type with the member functions below, and will use two stacks in order to implement the solution. (20 points)

```cpp
#ifndef STACK_H
#define STACK_H

#include "sll.h"

/// A stack implementation using singly linked list
class stack {

public:
/// Don't need to overwrite the default constructor, destructor, 
/// copy constructor or assignment operator.

/// Push aValue into the stack
void push(int aValue)
{
    mLL.addFront(aValue);
}

/// Pop out the top value from the stack
void pop()
{
    if (isEmpty()) {
        cout << "Stack is empty. Cannot pop!" << endl;
        return;
    }
    mLL.removeFront();
}

/// Fetch the top value from the stack (without deleting it)
int top()
{
    return mLL.front();
}

/// Check whether the stack is empty
bool isEmpty()
{
    return mLL.isEmpty();
}

private:
SLinkedList mLL;

};

#endif
```
There are several approaches to solving this problem.
The simplest approach is to just push everything onto stack 1 for the enqueue (because who cares what order things go on initially, just pump it on the stack in O(1).) Then when we dequeue, we will need to push everything onto a temporary stack and pop off the last item (and then push everything back on the other stack in order to maintain order). This is a tradeoff. Because if we know we’re going to have to enqueue a lot and rarely dequeue, this might be an acceptable strategy.

The best implementations rotate which stack holds the first item to be popped off and maintain the order of the queue. This also allows us to do more interesting operations in the future (what if we wanted to insert an item in the middle of the queue, or shuffle part of the queue? By maintaining order, this allows us to do more advanced operations should we choose).