Part I: Short Answer (12 questions, 65 points total)

Answer all of the following questions. READ EACH QUESTION CAREFULLY. Answer each question in the space provided on these pages. Budget your time so you spend enough on the programming questions at the end.

Keep your answers short and to the point. Good luck.

---

1. (7 points) Consider the following class and function definition (the code for the class member functions is included in their definitions to save space).

```
#include <iostream>
using namespace std;

class Mumble {
public:
    Mumble( )        { value = 0; cout << "default constructor " << value << endl; }
    Mumble(int val) { value = val; cout << "constructor " << value << endl; }
    ~Mumble( )       { cout << "destructor" << endl; }
private:
    int value;
};

void g( ) {
    Mumble mmm;
    for (int x = 0; x < 3; x++) {
        Mumble * rhubarbRhubarb = new Mumble(x);
    }
}
```

(a) (5 points) What output is produced when function g() is called?

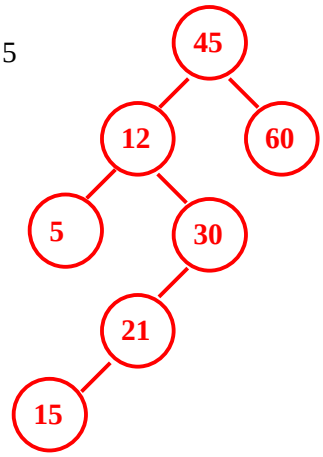
default constructor 0
constructor 0
constructor 1
constructor 2
destructor

(b) (2 points) Are there any errors or problems in the code? If so, what's wrong?

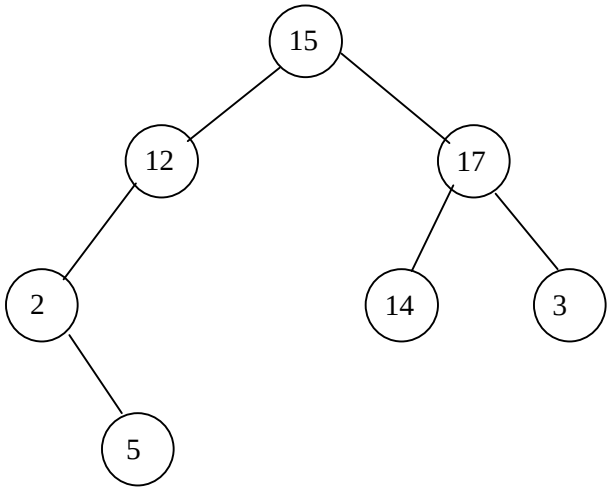
Yes – a memory leak. Three Mumble objects are allocated in the loop and never deleted.

2. (5 points) Draw the Binary Search Tree that is produced when these numbers are added to an initially empty tree ***in this order.***

45 60 12 30 21 15 5



3. (6 points) Consider the following tree:



(a) (2 points) Circle the correct answer. This data structure is:

(i) A Binary Search Tree.

(ii) A binary tree, but not a binary search tree.

(iii) A tree, but not a binary tree and not a binary search tree.

(iv) Not a tree.

(b) (4 points) Write down the numbers in the nodes in the order they would be encountered in a postorder traversal of the tree.

5 2 12 14 3 17 15

4. (4 points) Suppose we have a class that contains the following member function declaration.

```
class C {
public:
    …
    bool doSomething(const Thing &t) const;
    …
};
```

(a) (b)

(a) (2 points) What is the significance of the keyword const that is labeled (a)?

doSomething will not change parameter t.

(b) (2 points) What is the significance of the keyword const that is labeled (b)?

doSomething will not change the instance of class C associated with the current call (i.e., will not change the "this" object).

5. (3 points) A class V contains a private integer variable named value, along with functions to give that variable a new value:

```
class V {
public:
    void setValue(int value);
    …
private:
    int value;
};
```

Your colleague, A. Hacker, is trying to implement this function and has this code in file v.cpp.

```
// set value
void V::setValue(int value) {
    value = value;
}
```

Something's wrong – the variable value in the object isn't getting updated when this function is called.

What change could you make ***to the body of the function only*** that would fix the problem?

this -> value = value;

[Answers that changed the function heading by renaming the parameter received partial credit.]

[It is also technically possible to use the scope resolution operator to solve the problem, but that is pretty obscure (V::value = value;).]

6. (3 points) Suppose the following functions have been declared

```
void exam(int n);                 // 1
void exam(double x, int n);       // 2
void exam(int n, double x);       // 3
void exam(double x, double y);    // 4
```

For each of the following function calls, indicate which overloaded function (1, 2, 3, or 4) is actually called. If the call cannot be resolved to a unique function, describe why this can't be done.

a) exam(3.14, 1); **2**

b) exam(3.14, 1.0); **4**

c) exam(3, 1); **Ambiguous – 2 and 3 are equally good matches**

7. (4 points) An important part of creating a hash table is picking a good *hash function.*

(a) (2 points) What is a hash function?

A function that converts key values to integer bucket numbers (typically integer values between 0 and *B*-1, when there are *B* buckets in the hash table).

(b) (2 points) What does it mean for a hash function to be "good", and why does this matter?

A good hash function minimizes the number of keys that hash to any single bucket number (i.e., distributes the keys evenly among the buckets). That minimizes the amount of time needed to insert, delete, and look for items in the table.

8. (5 points) Use the definition of big $O(\ )$ notation to prove that $6n + 12n^2 + 105$ is $O(n^2)$.

To show that $6n + 12n^2 + 105$ is $O(n^2)$ we need to find a constant c such that

$$6n + 12n^2 + 105 \leq cn^2$$

for all sufficiently large n. In this case, we can pick c to be any number larger than 12 (13, for example).

[Answers that argued that $6n + 12n^2 + 105$ is $O(n) + O(n^2) + O(1)$, which is $O(n^2)$, received partial credit.]

9. (6 points) What is the running time of each of the following code fragments expressed using $O(\ )$ notation as a function of n? You should give an answer that is the smallest (fastest) time class (e.g., all of these are $O(2^n)$, but that does not answer the question).

Assume that all variables are declared to have type int.

(a)

```
for (outer = n; outer > 0; outer--) {
    inner = 1;
    while (inner < n) {
        inner = 2 * inner;
    }
}
```

$O(\ \mathbf{n \log n}\ )$

(b)

```
for (outer = 1; outer < n; outer++) {
    inner = 1;
    while (inner < outer) {
        inner = 2 + inner;
    }
}
```

$O(\ \mathbf{n^2}\ )$

(c)

```
for (x = 0; x < 10; x++) {
    y = 0;
    while (y < x) {
        y ++;
    }
}
```

$O(\ \mathbf{1}\ )$

10. (7 points) (a) (4 points) Fill in the blanks below with the expected (average case) and worst case running times for quicksort and mergesort using *O*( ) notation.

Quicksort: average *O*(***n log n***) , worst case *O*($\boldsymbol{n^2}$)

Mergesort: average *O*(***n log n***) , worst case *O*(***n log n***)

(b) (3 points) If the worst cases are the same, say so. If they are different, explain why. Use appropriate examples in your answer to show that you understand the conditions under which the worst cases could be different, if that is so (but please be reasonably brief).

At each recursive step, mergesort always partitions the array section being sorted into two equal-sized halves. That means that the depth of the recursive calls will always be log *n*.

Quicksort is *n* log *n* provided that the partition function divides the array sections into approximately equal-sized halves. If the partition function consistently picks the largest or smallest element in the array section, then each recursive call will only process that one element, and the depth of the recursion will be *n*.

11. (7 points) (a) (5 points) What is the **average** (expected) **time**, in *O*( ) notation, needed to determine whether a particular number appears in the following data structures, assuming that the data structure contains *n* numbers, and that an appropriate, fast algorithm is used.

i) Linked list ***O*(*n*)**

ii) Sorted array ***O*(log *n*)**

iii) Unsorted array ***O*(*n*)**

iv) Binary tree (not Binary Search Tree) ***O*(*n*)**

v) Binary Search Tree ***O*(log *n*)**

(b) (2 points) Is the **worst-case** time needed to search for a value different from the average (expected) time for any of the data structures in part (a)? If so, which ones, and what is (are) the worst-case time(s) for each one?

A binary search tree (v) can require *O*(*n*) time to search if it is badly unbalanced.

12. (8 points) The nodes of an integer Binary Search Tree can be represented by the following C++ data structure.

```
struct BSTNode {
    int item;
    BSTNode *left;
    BSTNode *right;
};
```

Complete the definition of the **recursive** function nPositive so it returns the number of nodes whose item field is a positive integer (i.e., greater than 0). For full credit, you should not search the entire tree if you don't have to.

```
// number of positive nodes in BST with root r
int nPositive(BSTNode *r) {

    if (r == null) {
        return 0;
    }

    if (r -> item <= 0)
        return nPositive(r -> right);
    else
        return 1 + nPostive(r -> left) + nPositive(r -> right);


}
```

Part II. Programming Problem (1 question, 50 points total)

In this question, you are to implement a queue data structure to model queues of customers waiting in line at a bank. This question involves implementing a complete C++ queue class, including constructors, destructors, assignment, and queue operations, using a linked list to represent the queue. Parts of the implementation are provided for you; you must use these as given. You need to complete the definitions of several functions; if you find it helpful to define additional, helper functions, you are free to do so.

Please read the question through before you start. The rest of this page contains a quick summary of the classes used to represent customer data. You ***do not*** need to implement any of these classes. This is supplied only to help you understand the kind of data to be stored on the queue. The description of the queue class, and the functions you are to implement are given on the next page.

Customer Data. Banks have several kinds of customers. These are represented using a class hierarchy, outlined here. Each of the classes contains a complete set of constructors, destructors, and assignment. In addition, each class contains a version of function duplicate that returns a pointer to a new duplicate (copy) of the customer object. You ***do not*** need to implement these classes. **Repeat: Don't implement these classes. Just use them.**

```
// generic customer (base class)
class Customer {
public:
    Customer();                                     // default constructor
    Customer(const Customer &other);                // copy constructor
    virtual ~Customer();                            // destructor
    Customer &operator=(const Customer &other);     // assignment

    // return a pointer to a new,  exact copy of this Customer with the correct dynamic type
    virtual Customer *duplicate( ) const;

    // other member functions as necessary
    …
private:
    …       // representation omitted
};

// retail customer
class RetailCustomer: public Customer {
public:
    // RetailCustomer versions of functions declared in Customer (overriding functions)
    …
};

// commercial customer
class CommercialCustomer: public Customer {
public:
    // CommercialCustomer versions of functions declared in Customer (overriding functions)
    …
};
```

Here is the specification of the CustomerQueue class that you are to implement.

```
class CustomerQueue {
public:
    // constructor, create an empty queue
    CustomerQueue();

    // copy constructor
    CustomerQueue(const CustomerQueue& other);

    // destructor – delete this CustomerQueue and all of the Customer objects it refers to
    ~CustomerQueue();

    // assignment operator
    CustomerQueue& operator=(const CustomerQueue& other);

    // Add (enqueue) customer object c to the rear of the queue.
    void add(Customer *c);

    // Remove (dequeue) the customer on the front of the queue and return a pointer
    // to that customer object.  Do nothing and return NULL if the queue is empty.
    Customer *remove( );

private:
    // private helper functions

    // Delete all data (nodes and customers) associated with this CustomerQueue
    void deleteAll( );

    // Store a copy of CustomerQueue other in this CustomerQueue
    // Precondition: this CustomerQueue contains no data (any nodes or objects have
    // already been deleted)
    void copy(const CustomerQueue &other);
```

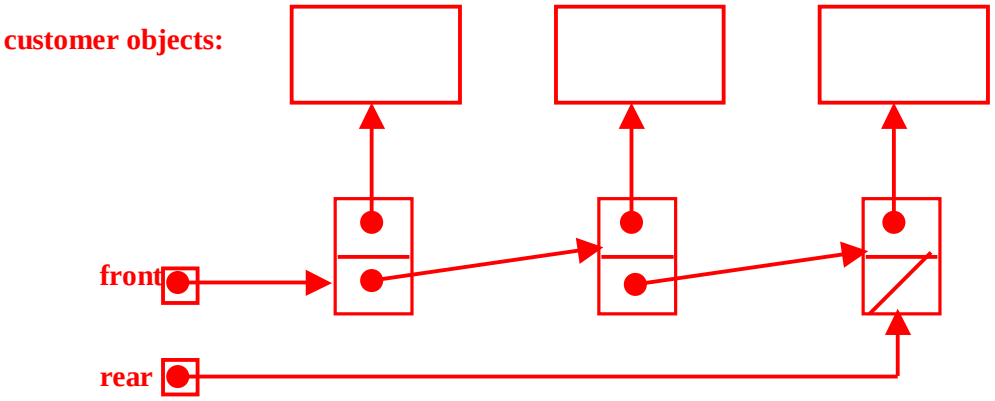
Write part (b) answers below (data representation plus declarations of any other functions or variables you need; see next page)

```
// individual queue node
struct CQNode {
    Customer * item;      // -> customer object
    CQNode * next;        // -> node inserted in the queue after this one, or NULL
                          //       if this node is at the rear of the queue
};


// CustomerQueue representation
CQNode * front;           // front node or NULL if queue is empty
CQNode * rear;            // last node inserted in the queue, or NULL if is queue empty
```

[Note: It is possible to represent an empty queue by using, for example, the convention that if front is NULL, then the queue is empty and the value of rear is irrelevant. In this solution we adopt the convention that both front and rear are NULL if the queue is empty.]

(a) (2 points) You ***must implement*** the CustomerQueue with a single-linked list of nodes, and you ***must*** have pointers to the front and rear nodes of the queue as private data in the CustomerQueue object. Use the space below to draw a picture that illustrates the data representation of your CustomerQueue. Be sure to show how the list nodes are connected, how they refer to Customer data, and where the front and rear pointers point.



[Many solutions had links pointing from the rear to the front of the queue (i.e., next links pointing from right to left in the picture above). That requires an O(n) traversal of the list to get to the second node in the queue, but done correctly, it received full credit.]

(b) (4 points) Complete the private part of the class declaration on the previous page, adding appropriate declarations for variables and types to represent the queue linked list.

Hint: If you need to declare additional structs or classes to represent list nodes or other things, you can include them in the private part of the CustomerQueue class declaration.

(c) (4 points) Complete the definition of the CustomerQueue constructor so it creates an empty queue.

```
// Initialize this CustomerQueue to an empty queue
CustomerQueue::CustomerQueue( ) {

    front = rear = NULL;

}
```

(d) (6 points) Implement function add(c), which should add customer c to the rear of the queue.

```
// Add (enqueue) customer object c to the rear of the queue.
void CustomerQueue::add(Customer *c) {

    // create new queue node
    CQNode * p = new CQNode;
    p -> item = c;
    p -> next = NULL;

    if (front == NULL) {
        // first node in the queue
        front = p;
        rear = p;
    } else {
        // add to back of existing queue
        rear -> next = p;
        rear = p;
    }

}
```

(e) (6 points) Implement function remove(c), which should remove the customer from the front of the queue and return a pointer to it. If the queue is empty, return NULL.

```
// Remove (dequeue) the customer on the front of the queue and return a pointer
// to that customer object.  Do nothing and return NULL if the queue is empty.
Customer * CustomerQueue::remove( ) {

    if (front == NULL)
        return NULL;

    CQNode * temp = front;
    Customer * result = front -> item;
    front = front -> next;
    if (front == NULL) {
        rear = NULL;
    }
    delete temp;
    return result;

}
```

(f) (8 points) Implement function deleteAll( ), which should delete all data associated with this queue – including queue nodes and Customer records.

```
// Delete all data (nodes and customers) associated with this CustomerQueue
void CustomerQueue::deleteAll( ) {

    CQNode * p;
    while (front != NULL) {
        p = front;
        front = front -> next;
        delete p -> item;
        delete p;
    }
    rear = NULL;

}
```

Here are a couple of other solutions:

```
// Delete all data (nodes and customers) associated with this CustomerQueue
void CustomerQueue::deleteAll( ) {
    Customer * p;
    while (front != NULL) {
        p = remove( );
        delete p;
    }
    rear = NULL;
}
```

or

```
// Delete all data (nodes and customers) associated with this CustomerQueue
void CustomerQueue::deleteAll( ) {
    while (front != NULL) {
        delete remove( );
    }
    rear = NULL;
}
```

(g) (8 points) Implement function copy( ), which should make a complete copy of its CustomerQueue parameter other and store that copy in this CustomerQueue. Assume that any data previously belonging to this CustomerQueue has already been deleted.

Hint: Class Customer may contain member functions that are particularly useful here.
Hint: Class CustomerQueue may contain member functions that are particularly useful here.

```
// Store a copy of CustomerQueue other in this CustomerQueue
// Precondition: this CustomerQueue contains no data
void CustomerQueue::copy(const CustomerQueue &other) {

    front = rear = NULL;          [This could be placed at the beginning of the
                                   copy constructor, but it must be done before
                                   the following code will work properly when
                                   called from the copy constructor.]

    CQNode * p = other.front;
    while (p != NULL) {
        add( p -> item -> duplicate( ) );
        p = p -> next;
    }

}
```

(h) (4 points) Implement the copy constructor for class CustomerQueue. You ***must*** use functions copy( ) and/or deleteAll( ) if you need to copy another CustomerQueue or delete this one.

```
// copy constructor
CustomerQueue::CustomerQueue (const CustomerQueue& other) {

    copy(other);

}
```

(i) (5 points) Implement the assignment operator for class CustomerQueue. You ***must*** use functions copy( ) and/or deleteAll( ) if you need to copy another CustomerQueue or delete this one.

```
// assignment operator
CustomerQueue& CustomerQueue::operator=(const CustomerQueue& other) {

    if (&other == this)
        return *this;

    deleteAll( );
    copy(other);
    return *this;

}
```

(j) (3 points) Implement the destructor for class `CustomerQueue`. You ***must*** use functions copy() and/or deleteAll( ) if you need to copy another CustomerQueue or delete this one.

```
// destructor – delete this CustomerQueue and all of the Customer objects it refers to
CustomerQueue::~CustomerQueue() {

    deleteAll( );

}
```