Part I: 20 Multiple choice questions (2 points each)

Answer all of the following questions. READ EACH QUESTION CAREFULLY. Fill the correct bubble on your mark-sense sheet. Each correct question is worth 2 points. Choose the one BEST answer for each question. Assume that all given C++ code is syntactically correct unless a possibility to the contrary is suggested in the question.

In code fragments, YOU SHOULD ASSUME THAT ANY NECESSARY HEADER FILES HAVE BEEN INCLUDED. For example, if a program does input or output, you should assume that header files for the appropriate stream library have been #included, and that "using namespace std;" has been provided, even if it is not shown in a question.

Remember not to devote too much time to any single question, and good luck!

1. **Given a binary search tree, which traversal type would print the values in the nodes in sorted order?**

   A. Preorder
   B. Postorder
   C. Inorder
   D. None of the above

2. **What is the running time of the following code fragment?**

```
for(int i=0; i<10; i++)
   for(int j=0; j<N; j++)
      for(int k=N-2; k<N+2; k++)
          cout << i << " " << j << endl;
```

   A. O(log N)
   B. O(N log N)
   C. O(N)
   D. O(N^2)
   E. O(N^3)

3. **Suppose we're debugging a quicksort implementation that is supposed to sort an array in ascending order. After the first partition step has been completed, the contents of the array are in the following order:**

   3 9 1 14 17 24 22 20

   Which of the following statements is correct about the partition step?

   A. The pivot could have been either 14 or 17
   B. The pivot could have been 14, but could not have been 17
   C. The pivot could have been 17, but could not have been 14
   D. Neither 14 nor 17 could have been the pivot

4. Which of the following statements about binary trees is NOT true?

A. Every binary tree has at least one node.
B. Every non-empty tree has exactly one root node.
C. Every node has at most two children.
D. Every non-root node has exactly one parent.

5. Which of the following will be the likely result of failing properly to fill in your name, student ID, section number, and *EXAM VERSION* on your scantron form?

A. A score of 0 will be recorded for the multiple choice portion of the final exam, regardless of how many questions you answer correctly.
B. Your grade in the course will be lower than it might otherwise be since a 0 will be recorded for the multiple choice portion of the final exam.
C. You will earn the gratitude of your classmates by helping to lower the curve, since a 0 will be recorded for the multiple choice portion of the final exam.
D. You will need to do exceptionally well on the programming portion of this exam to help offset the 0 that you will earn for the multiple choice portion.
E. All of the above

6. Suppose we have two classes, one of which extends the other:

```
class Base { … };

class Derived: public Base { … };
```

Now suppose we execute the following program:

```
line 1   int main( ) {
line 2     Base* b;
line 3     Derived* d = new Derived;
line 4     b = d;
line 5     delete d;
line 6     return 0;
line 7   }
```

What is the *static type* of variable b after line 4 has been executed and before line 5 is executed?

A. Base *
B. Base &
C. Derived *
D. Derived &
E. Derived

[The dynamic type of a variable can change as the result of an assignment, but the static type is fixed – it's the one given in the variable's declaration.]

7. **How many times is the symbol '#' printed by the call foo(4)?**

```
void foo (int i) {
  if (i > 1) {
    foo (i/2);
    foo (i/2);
  }
  cout << "#";
}
```

A. 3
B. 4
C. **7**
D. 8
E. Something else

8. **A class template in C++ has the following structure**

```
template <class T> class TemplatedClass {
  …
};
```

What is the meaning of T in the above declaration?

A. It is a placeholder for a pointer value
B. **It is a placeholder for a type name**
C. It is a string variable
D. It must be an integer constant
E. It must be a function name

9. **Are there any dynamic memory management errors in the following code?**

```
int *p = new int;
int *q = new int;
int *r;
*p = 17;
r = q;
*q = 42;
p = q;
delete r;
```

A. No, there are no errors
B. **Yes, a memory leak**
C. Yes, misuse of a dangling pointer
D. Yes, both a memory leak and misuse of a dangling pointer
E. Yes, a dangling chad

10. Suppose we have the following pair of class definitions, along with implementations of some functions of the derived class.

```
class Odd {
public:
  virtual void setName(string name);
  virtual void setID(int id) = 0;
protected:
  int id;
private:
  string name;
};

class Deranged: public Odd {
public:
  virtual void setName(string name);
  virtual void setID(int id);
};

Deranged::setName(string name) {
  this->name = name;              // this is statement A
}

Deranged::setID(int id) {
  this->id = id;                  // this is statement B
}
```

The question concerns whether the two statements labeled A and B are legal (i.e., will be accepted by a standard C++ compiler).

A. Both statements are legal

B. A is legal, B is not

C. B is legal, A is not

D. Neither statement is legal

11. Suppose we have the following class whose underlying data structure is a linked list of ListNodes.

```
class List {
public:
        // other public functions
   ~List();
private:
   struct ListNode {
           int item;
           ListNode *next;
   };

   ListNode *head;
};
```

Which of the following sequences of code could be used in the destructor ~List() to correctly delete all of the nodes in the list? (Which ones are legal, even if the style is atrocious?)

```
I.      for (ListNode *n = head; head != NULL; head = n) {
           n = head->next;
           delete head;
        }

II.     for (ListNode *n = head; n != NULL; n->next) {
           delete  n;
        }

III.    ListNode* n;
        while (head != NULL) {
           n = head->next;
           delete head;
           head = n;
        }
```

A I and II only

B. II and III only

C. I and III only

D. III only

E. I, II, and III

12. Suppose you were implementing a data structure to store information about the paintings on display at an art dealer's showroom. Of the following data structures, which one is the <u>right</u> one to use?

A. Unordered array
B. Sorted array
C. Linked list
D. Binary search tree
E. It depends

[This was a recurring theme in our discussions of data structures. You have to know how the data structure will be used and which operations need to be efficient to make an intelligent choice. The question doesn't provide enough information to do this without additional guesses or assumptions.]

13. In lecture we defined a class IntStack to implement a stack of integers:

```
class IntStack {
public:
  IntStack( );
  bool isEmpty( );
  void push(int item);
  int pop( );
  int top( );
}
```

What happens if we execute the following statements?

```
IntStack s;
int n1, n2, n3;

s.push(17);
s.push(143);
s.push(42);
n1 = s.pop( );
n2 = s.top( );
s.push(n1);
n3 = s.pop( );
n1 = s.top( );
```

A. Stack contains 143 (top), 17 (bottom); n1=42, n2=42, n3=42
B. Stack contains 42 (top), 42, 143, 17 (bottom); n1=42, n2=42; n3=42
C. Stack is empty; n1=17, n2=143, n3=42
D. Stack contains 42 (top), 17 (bottom); n1=42, n2=143, n3=143
E. Stack contains 143 (top), 17 (bottom); n1=143, n2=143, n3=42

14. Is this a binary search tree?

```
            55
          /    \
        17      60
       /  \    /  \
      5   20  42  105
     / \        \
    3   9        55
```

A. Yes

[✓ **B. No**]

15. What is the <u>expected</u> time required to search for a value in a binary search tree containing n nodes? (You should make reasonable assumptions about the structure of the tree.)

A. O(1)

[✓ **B. O(log n)**]

C. O(n)

D. O(n log n)

E. O(n^2)

16. If we use mergesort to sort an array with n elements, what is the <u>worst case</u> time required for the sort?

A. O(1)

B. O(log n)

C. O(n)

[✓ **D. O(n log n)**]

E. O(n^2)

17. There are several factors that affect the efficiency of lookup operations in a hash table. Which of the following is <u>not</u> one of those factors?

A. Number of elements stored in the hash table

[✓ **B. Size of elements stored in the hash table**]

C. Number of buckets in the hash table

D. Quality of the hash function

E. All of the above factors affect the efficiency of hash table lookups

18. What is the complexity of the following code expressed in O( ) notation? If more than one answer is correct, choose the smallest one.

```
for (int j = n; j > 0; j--) {
  for (int k = 1; k < j; k = k+k) {
    cout << j+k << " ";
  }
  cout << endl;
}
```

A. O(log n)
B. O(n)
C. O(n log n)
D. O(n^2)
E. O(2^n)

19. What is the infix version of the following postfix expression?

x 12 + z 17 y + 42 * / +

Hint: Try using a stack to evaluate the postfix expression and see what happens.

A. (x + 12 + z) / (17 + y * 42)
B. x + 12 + z / 17 + y * 42
C. x + 12 + z / (17 + y) * 42
D. x + 12 + z / ((17 + y) * 42)
E. x + (12 + z) / (17 + y * 42)

20. What is printed when we execute the following program?

```
class FloridaBallot {
public:
        FloridaBallot( )  { cout << "Welcome to Florida" << endl; }
        virtual  ~FloridaBallot( ) { cout << "Come back soon!" << endl; }
        virtual void vote(string who);
private:
        …
};

class PalmBeachBallot: public FloridaBallot {
public:
        PalmBeachBallot( )   { cout << "Entering Palm Beach" << endl; }
        ~ PalmBeachBallot( ) { cout << "Leaving Palm Beach" << endl; }
};

int main( ) {
        FloridaBallot* b = new PalmBeachBallot;
        …
        b->vote("Mickey Mouse");
        …
        delete b;
        return 0;
}
```

A. Welcome to Florida
Come back soon!
Entering Palm Beach
Leaving Palm Beach

B. Welcome to Florida
Entering Palm Beach
Come back soon!
Leaving Palm Beach

C. Welcome to Florida
Entering Palm Beach
Leaving Palm Beach
Come back soon!

D. Entering Palm Beach
Welcome to Florida
Come back soon!
Leaving Palm Beach

E. Entering Palm Beach
Welcome to Florida
Leaving Palm Beach
Come back soon!

Part II: 2 Short Answer Questions (10 points)

[Remember to answer question 20 on the back of the previous page. Don’t skip it!]

21. (6 points) Quicksort is claimed to have an expected running time of O(n log n), but it could be as slow as O(n^2).

(a) Briefly explain why Quicksort could use O(n^2) time instead of always running in time O(n log n).

Quicksort will use O(n^2) time if the partition function always picks as the pivot the largest or smallest element of the array section being sorted. That will cause the depth of the recursion to be O(n) instead of O(log n).

(b) How can you fix Quicksort so the expected time is O(n log n), if it can be done? You should give a specific suggestion (don’t just say something like “be clever and careful”). Explain why your solution will change the expected time to O(n log n).

Use a partition algorithm that partitions the array section into two equal-sized halves. One way to do this is to pick the pivot element randomly; another is to use an algorithm that estimates the median value in the array by, say, picking the median of the first, last, and a few intermediate elements of the array section. This will limit the depth of the recursive calls to O(log n) and, since each level does O(n) work, the total time will be O(n log n).

22. (4 points) One of your colleagues is trying to debug a list class that uses an array to hold the list elements. Something is wrong somewhere in this code. Where are the bug(s)?

Circle the bug(s) in the code and give a brief explanation of what's wrong.

To save space, the implementations of the functions are given in the class definition, not in a separate implementation file.

```
const int MAXSIZE = 1234;        // max # elements that can be stored in a list

class List {
public:
        // construct empty list
        List( ) { size = 0; }

        // insert item at end of list; do nothing if list is already full
        void Insert(string item) {
          if (size < MAXSIZE) {
            size++;                      These statements are in the wrong order.  size++
            data[size] = item;           needs to be executed after the assignment, or
          }                              something similar needs to be done
        }

        // = # of elements in this list
        int sizeOf( ) {
          return size;
        }

        // = location of item in the list, or –1 if not found
        int find(string item) {
          int k = 0;
          while (data[k] != item && k < size)   This needs to test k<size before accessing
            k++;                                 data[k].  Reverse the condition so it is
          if (k < size) return k; else return –1;  k < size && data[k] != item.  Otherwise
        }                                        searching for an element not in the list
                                                 will reference data[size], which is out of
private:                                         bounds
        int      size;                   // # of elements currently stored in this list
        string data[MAXSIZE];            // list elements are stored in data[0..size-1]
};
                                                       ↑
                                                 key to both bugs
```

Part III: Programming Questions (50 points)

23. (10 points) The nodes of a binary tree containing strings can be represented in C++ as follows:

```
struct BTNode {          // binary tree node
  string data;           // node data
  BTNode * left;         // left subtree
  BTNode * right;        // right subtree
};
```

An *internal node* in a binary tree is one that is not a leaf, i.e., an internal node is one that has at least one child.

Complete the following function so it returns the number of internal nodes in a binary tree.

Hint: Recursion is your friend.

```
// = # of internal nodes in tree with given root
int nInternal(BTNode *root) {

  // handle empty tree or leaf node
  if ((root == NULL) || (root->left == NULL && root->right == NULL))
    return 0;

  // internal node
  return 1 + nInternal(root->left) + nInternal(root->right);

}
```

A couple of CSE grad students (including one of the TAs) were getting a bit punchy late one night and came up with the following solution. The style is definitely something to be avoided, unless you're entering the Obfuscated C coding contest.

```
// = # of internal nodes in tree with root r
int nInternal(BTNode *r) {
  return (r) ? (r->left || r->right) + nInternal(r->left) + nInternal(r->right) : 0;
}
```

24. (25 points) Recall from lecture that a fixed-size queue can be implemented using an array, a variable holding the number of items currently in the queue, and two variables indicating where in the array the front and rear of the queue can be found.

The main complication is that the front and rear indicators need to wrap around to the beginning of the array after they run off the end.

Here is the declaration of a CircularQueue class for a queue of integers. The comments on the queue variables (data, front, and rear) are incomplete or missing – part of your task is to figure out precise descriptions of these variables.

```
const int QUEUESIZE = 100;      // max # elements in the queue
Class CircularQueue {
public:
        // construct empty CircularQueue
        CircularQueue( );

        // = “this queue is empty”
        bool isEmpty( );

        // = “this queue is full”
        bool isFull( );

        // add item to the end of the queue (enqueue)
        // do nothing if the queue is already full
        void insert(int item);

        // return item at the front of this queue and remove it (dequeue)
        // precondition: this queue is not empty
        int remove( );
private:
        int size;                // # of items currently in the queue
        int data[QUEUESIZE];     // queue elements
        int front;       // Queue elements are stored in data[front..rear-1]
        int rear;        // modulo QUEUESIZE.  data[front] is the first element
                         // in the queue if it is non-empty.  data[rear] is the next
                         // available element in the array where a new queue item
                         // can be stored, if the queue is not full.
};
```

(a) Complete the comments to the right of variables front and rear to specify exactly how they are used, i.e., which elements of array data they describe – first and last elements in the queue? Next available slot in the queue? etc. (You may find it helpful to sketch your implementation of the queue operations before you finish this part of the question.)

[There are many possible ways to define the queue pointers. Your answer did not have to use the definitions given here, but it is important that the comments make clear whether the front and rear indices point to queue elements, or the next unused array slot, or whatever. The reader should not have to examine implementations of the queue operations to figure that out. The code in your answer to part (b) should use these variables in a way that is consistent with the comments in your solution to part (a).]

(b) Complete the implementation of the CircularQueue member functions below. Your implementations *must* be consistent with the definitions of front and rear that you gave in your answer to part (a).

[You probably won't need all of the space given, particularly for the first few functions.]

```
// construct empty queue
CircularQueue::CircularQueue( ) {

   size = 0;
   front = rear = 0;

}
```

```
// = "this queue is empty"
bool CircularQueue::isEmpty( ) {

   return (size == 0);

}
```

```
// = "this queue is full"
bool CircularQueue::isFull( ) {

   return (size == QUEUESIZE);

}
```

```
// add item to the end of the queue (enqueue)
// do nothing if the queue is already full
void CircularQueue::insert(int item) {

  if ( isFull( ) )
    return;

  data[rear] = item;
  rear = (rear + 1) % QUEUESIZE;
  size++;



}



// return item at the front of this queue and remove it (dequeue)
// precondition: this queue is not empty
int CircularQueue::remove( ) {

  int item = data[front];
  front = (front + 1) % QUEUESIZE;
  size--;
  return item;



}
```

25. (15 points) Suppose we have an application where we need to store a list of words, and we need to efficiently support insert, delete, and lookup operations. Because insert and delete need to be efficient, we decide to use a linked list. Unfortunately, that means that lookup (search) operations have to scan the linked list – we can't use binary search.

However, one of your colleagues has observed that, in practice, the words on the list are not accessed randomly. In this particular application, if a word is accessed, then it is likely to be accessed again sometime soon.

That suggests a strategy for speeding up searches. When we search for a word, if we find it, we move the node containing that word to the *front* of the list. Then if we search for the same word again, it is likely to be found near the front of the list.

Example: Suppose we have the following list of words

elephant zebra aardvark platypus moose duck

and the user searches for "platypus". The lookup function should return true (the word is in the list) and should also rearrange the list so the node containing platypus has been moved to the front of the list.

platypus elephant zebra aardvark moose duck

If lookup searches for a word that is not in the list, it should return false and not change the order of the nodes in the list.

Here are the key parts of the FancyList class definition.

```
class FancyList {
public:
  …
  // = "word was found in this FancyList"
  // (side effect – the node containing word is moved to the front of the list)
  bool lookup (string word);
  …
private:
  struct LNode {            // list node
    string word;            // word stored in this node
    LNode *next;            // pointer to next node in the list or null if no next node
  };

  LNode *head;              // pointer to first node in the list, or null if list is empty
};
```

Complete the implementation of the function lookup on the next page. For *full credit*, you must rearrange the nodes in the list by updating the appropriate pointers. You MAY NOT copy or swap the string values stored in the nodes.

```
// = "word was found in this FancyList"
// (side effect – the node containing word is moved to the front of the list)
bool FancyList:: lookup(string word) {

  // if list is empty, word is not found
  if (head == NULL)
    return false;

  // if first node contains word, then we found it, and no rearrangement is needed
  if (head->word == word)
    return true;

  // General case: search for word.
  // At this point, the list is non-empty and the first node does not contain the desired word.
  LNode *prev = head;
  LNode *curr = head->next;
  while (curr != NULL && curr->word != word) {
    prev = curr;
    curr = curr ->next;
  }

  // either we found word or we reached the end of the list without finding it
  // return if we didn't find it
  if (curr == NULL)
    return false;

  // word found – move that node to the front of the list and return
  prev->next = curr->next;
  curr->next = head;
  head = curr;
  return true;

}
```

Before you leave, be sure to look through the entire test and check that you didn't skip any questions. There are 20 multiple-choice questions (do you have 20 answers on the scantron?) and 5 written ones.

Be sure that the correct test version is entered on your scantron form.

Best wishes for the Holidays from the staff of CSE 143!! See you next year!!!