Note: Some comments in code may be incorrect. This is to simulate an employee who thinks the code does X, but it does Y.
Note: Unless specified, there may be more than one correct answer
You will have two hours to complete this exam.

1. Suppose we implement a queue.
    a. What is the worst-case time to dequeue?
        i. $O(n^2)$
        ii. O(n)
        iii. O(1) // We perform some operations *independent of the input size*. Therefore, we can perform it in constant time. Even if we had 1000 lines to execute, O(1000) is simplified to O(1).
        iv. $O(2^n)$
    b. What is the worst-case time to enqueue?
        i. $O(n^2)$
        ii. O(n)
        iii. O(1) // We perform some operations *independent of the input size*. Therefore, we can perform it in constant time. Even if we had 1000 lines to execute, O(1000) is simplified to O(1).
        iv. $O(2^n)$
    c. What is the worst-case time to peek?
        i. $O(n^2)$
        ii. O(n)
        iii. O(1) // We perform some operations *independent of the input size*. Therefore, we can perform it in constant time. Even if we had 1000 lines to execute, O(1000) is simplified to O(1).
        iv. $O(2^n)$
    d. What is the worst-case time of isEmpty?
        i. $O(n^2)$
        ii. O(n)
        iii. O(1) // We perform some operations *independent of the input size*. Hopefully this is one or two instructions, thus O(2).
        iv. $O(2^n)$
2. Suppose we implement a Stack.
    a. What is the worst-case time to dequeue?
        i. $O(n^2)$
        ii. O(n)

iii. **O(1) // We perform some operations *independent of the input size*. Therefore, we can perform it in constant time. Even if we had 1000 lines to execute, O(1000) is simplified to O(1).**
iv. $O(2^n)$

b. What is the worst-case time to enqueue?
i. $O(n^2)$
ii. O(n)
iii. **O(1) // We perform some operations *independent of the input size*. Therefore, we can perform it in constant time. Even if we had 1000 lines to execute, O(1000) is simplified to O(1).**
iv. $O(2^n)$

c. What is the worst-case time to peek?
i. $O(n^2)$
ii. O(n)
iii. **O(1) // We perform some operations *independent of the input size*. Therefore, we can perform it in constant time. Even if we had 1000 lines to execute, O(1000) is simplified to O(1).**
iv. $O(2^n)$

d. What is the worst-case time of isEmpty?
i. $O(n^2)$
ii. O(n)
iii. **O(1) // We perform some operations *independent of the input size*. Therefore, we can perform it in constant time. Even if we had 1000 lines to execute, O(1000) is simplified to O(1).**
iv. $O(2^n)$

3. Suppose we have a word W and the length L. We want to see if W is a palindrome (radar backwards is radar). Which of the following is the correct implementation?
    a. A
    b. B
    c. C

B & C are valid options. B is the most efficient, using linear time to start at the beginning and end of the list. They will converge to the center at which point we determine if it is a palindrome or not. C will go through and push the first half of the word onto the stack then iterate over second half, all of which in reverse. It would be easier to traverse both sections straight. *A is incorrect because you push everything to the queue, then traverse both the word and stack in the same direction. Everything will be considered a palindrome.*

A)

```
bool isPalindromeA(string word){
  queue<char> q;

  // Put all of the word in the queue, reversed
  for (char& c : word){
    q.push(c);
  }

  char checkC;

  // Check the reversed word agaist the orig
word
  for (char& c : word){
    checkC = q.front();
    q.pop();

    if (c != checkC){
      return false;
    }
  }


  return true;
}
```

B)

```
bool isPalindromeB(string word){
  // Go through the word, in linear time
  for (size_t i = word.size() - 1, j = 0;
    i >= j;
    i--, j++){
   if (word[i] != word[j]){
     return false;
   }
  }

  return true;
}
```

C)

```
bool isPalindromeC(string word) {
    stack<char> s;
    for (int i = word.size()/2;
         i >= 0;
         i--) {
        s.push(word[i]);
    }

    for (auto i = word.size() - 1;
         i >= word.size()/2;
         i--) {
        if (s.top() != word[i]) {
            return false;
        }
        s.pop();
    }
    return true;
}
```

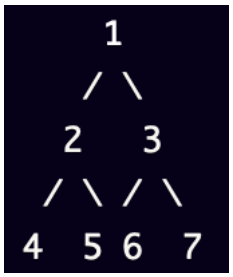
4. What is wrong with the following tree?

Here is a binary search tree (BST) that can be constructed with the input numbers 12, 5, 3, 85, 39, and 90:

Credit: GPT-3.5 Turbo

a. **Nothing // This is the correct insertion using a BST. First number is always root, then values less than root will go to the left. Values more root will go to the right. This process will continue until there is no spot.**
b. 85 and 90 should be swapped
c. The root is incorrect
d. Something else (describe below)

5. What input would produce the following tree?

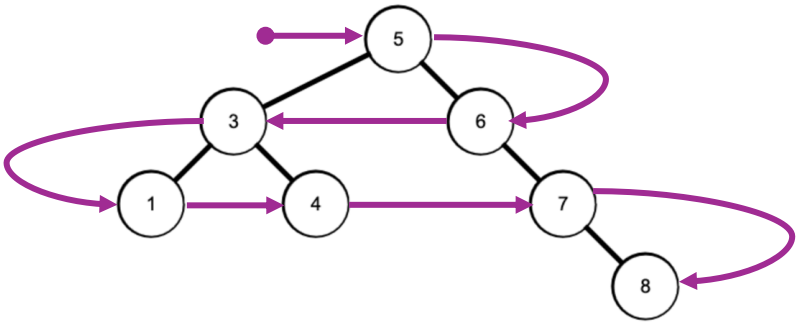


Credit: CPT-3.5 Turbo

a. I cannot know without more details
b. 7 6 5 4 3 2 1
c. 1 2 3 4 5 6 7
d. 3 2 1 7 6 5 4
e. **Something else (describe below)**

We cannot possibly have a BST with values on the left side which are greater than the root. We see this violation for 2, 3, and 1. Regardless of knowing more details, a BST cannot form this graph using the functions we have used. To achieve this tree, we would manually need to set left and right values.

Use the tree below for 6 & 7.



Credit: https://wintercore.github.io/bst-visualizer/

6. What is the insert order to obtain this tree?
    a. 8 7 4 1 3 6 5
    b. 5 3 6 1 4 7 8
    c. 5 6 3 7 4 1 8
    d. 6 5 7 8 4 3 1
    e. Something else (Describe to the right)

How do we determine a possible input? We draw a zig-zag down the tree until we read the base. For each level, the order of the numbers can be in any order. But, the levels must remain in the same order. That is, we have the levels:

5 Combos: 5
3, 6 Combos: 3, 6 **OR** 6, 3
1, 4, 7 Combos: 1, 4, 7 **OR** 1, 7, 4 **OR** 4, 1, 7 **OR** 4, 7, 1 **OR** 7, 1, 4 **OR** 7, 4, 1
8 Combos: 8

7. What insert order can we use to obtain a balanced tree?
    a. No such order exists
    b. The order: 5, 3, 7, 1, 4, 6, 8 // There are many correct answers. If your numbers are in the same color group, it is correct.
8. Why would we want to write a private and public version of the same function?
    a. We do not, it wastes time
    b. If we are working with time-critical programs where 1ns is critical
    c. To increase security
    d. To hide pointer information // By using a public interface, we can handle all pointer information within the class itself. This way, pointers are never mishandled.
    e. To convert a const parameter to non-const parameter
    f. Because our boss tells us to
9. What should every class contain? [Select exactly 3]
    a. Constructor // Sometimes you will need multiple of these. For example, you need one to accept no parameters (default), one that

only accepts a string, another that only accepts an int. You can use default parameters to reduce the number of bodies used.

b. Setter // Use these sparingly – if an external entity NEEDS to change the value, then provide a setter. Otherwise, do not exposure variables that are meant to be internal.
c. Getter // Use these sparingly – if an external entity NEEDS to get the value, then provide a getter. Otherwise, do not exposure values. Also be careful returning references and pointers as they can allow for direct manipulation after return.
d. Destructor // There should 1 and one only. It can be empty, but any memory that was (possibly) allocated within the class must be deallocated here. The form must be `~CoolClass()`
e. Overloaded stream operators (<< and/or >>) // Not in a class
f. Overloaded assignment operator // Use as needed
g. Inheritance // Use to simplify classes which have similar attributes
h. Copy constructor // There is a single form and a single form only. For some class Cool, `Cool(Cool& copy)`. If you do not have the Cool& as the parameter type, then it is not a copy constructor. If there is no & or *, then it is a compile-time error since you need to invoke the copy constructor to make that operation possible, but it is not defined. If we have Cool*, then it is simply a constructor which accepts a pointer to another instance.
i. Private members // Use to hide data from other classes whether they inherit or instantiate the class.
j. Virtual functions // Can be useful in Inheritance
k. Virtual constructors // Not a thing

10. Suppose we have a queue with 100 elements.
    a. When we dequeue,
        i. Only the front changes // Dequeue always occurs at the front, therefore we need to remove the front element (and update the reference).
        ii. Only the rear changes
        iii. Nothing changes
        iv. Other/Conditionally I, II, III
    b. When we enqueue,
        i. Only the front changes

ii. **Only the rear changes // Enqueue always occurs at the back, therefore we need to add to the back element (and update the reference).**
iii. Nothing changes
iv. Other/Conditionally I, II, III

c. When we peek at the front or back,
i. Only the front changes
ii. Only the rear changes
iii. **Nothing changes // Usually, we will look at the front using a function `front()`. We also could look at the back of the queue, but it does not necessarily make sense since we are operating in FIFO**
iv. Other/Conditionally I, II, III

11. What is wrong with the following code?

```
// The base class with x, y, and z visible
class BaseClass {
protected:
    int x; int y; int z;
};

// The mid class with x, y, z, a, b, and c visible
class MidClass: BaseClass {
protected:
    int a; int b; int c;
};

// The upper class with x, y, z, a, b, c, d, f, and g visible
class UpperClass: MidClass {
    UpperClass() {
        x = d; y = f; z = g;
    }
protected:
    int d; int f; int g;
};
```

Write what is wrong with the code here (also provide solution):

We are directly accessing a private member of BaseClass. Since we inherit BaseClass here, we are doing so using private inheritance. This makes all visible members of BaseClass private members of MidClass. *If there are private members of BaseClass, MidClass still cannot see them.* When we inherit MidClass in UpperClass, x, y, and z are private members of MidClass. Therefore, we cannot see them in UpperClass. Another potential issue is the location of UpperClass' constructor. If we wanted it to be only accessible as private, then nothing needs

to change. If we wanted someone to create an instance of UpperClass, then the definition given is not visible to them.

To resolve this, we can either add a *protected* getter/setter in MidClass which will allow classes who inherit from MidClass (such as UpperClass) to modify the value of x, y, and z. If we only handle those variables in the constructor, we could invoke a constructor of MidClass which will set the variables. If we need reoccur access, we can use the getter/setter method as described or inherit BaseClass by protected. The downside of this would be any protected/public members of BaseClass will then be protected members of MidClass. As such, we may be forced to use the getter/setter method to resolve the access issue.

12. Write the purpose of the following data structures:
    a. Tree
        i. It allows for efficient searching, traversal, and organization. They can be used for purposes such as database indexing, AI, file systems (Unix file system is a tree), sorting a list, parsing, computing RPN equations. This is a non-linear data structure. Worst case search/insert will be O(N); best/average case is O(logn) .
    b. Queue
        i. It allows for a way to hold data in a linear fashion, reading from one end and writing to the other (FIFO). This can be used for a printing queue, process queue, represent a prize claim system (user X won, if they do not respond in 2 min, Y wins, etc.). These can operate in O(1) time for enqueue/dequeue if we have a reference to the back and front elements. Everything in-between will be linked to each other.
    c. List
        i. It is a linear data structure. Operations can be done at any index. A grocery list for example. We can add grapes to a lot of other items, then remove apples from the middle of the list once we grab them. Queues and Stacks do not allow for this any index operation. These operations come at a cost of O(M) where M is the index, worst case, they will be O(N) where N is the length of the list.
    d. Stack
        i. It is also a linear data structure. Operations can only be done to the top, first in last out. We see this in code all the time – call stack.

When we call a function, we must exist that prior to returning to main. While we can technically jump, we should remove one frame at a time (aka, do not touch program counter). All operations should be O(1) since we work on the same reference. We could also use this if we want to check if something is a reverse of another. It can also be used for reversing something (becomes $O(N^2)$ operation).

13. Virtual functions must be pure virtual functions if there is at least one pure virtual function.
    a. True
    b. False // There can a pure virtual and virtual function, they do not all have to be pure virtual.
14. We use the @Override attribute to override (not overload) a function.
    a. True
    b. False // We use the override keyword. Attributes are not used in that way for C++. Other langs such as Java use that notation.
15. Inserting at the end of a singly linked list is *always* slower than inserting at the end of an array.
    a. True
    b. False // If we have a reference to the end, then the operation will become O(1). If we do not have a reference to the end, then the operation will be O(N). Inserting at the end of an array is O(1). So, in some cases it is slower to use a linked list, but it is not guaranteed.
16. We use UML to graphically describe the functionality of a system.
    a. True // We can use UML to model anything relating to a system (how a user interacts with it, how the system communicates to other components within, how the classes interact (e.g., inheritance, class relationships), and many more)
    b. False
17. UML can only describe logical, class, and other code-related relationships.
    a. True
    b. False // Same reason as above. Me interacting with the system has nothing to do with code, logic, or class structure. It may influence them, but UML describes a lot more.

18. You attempt to allocate memory, but it fails. An exception (`std::bad_alloc`) is thrown. But you are a good programmer and anticipated this being thrown, write the code below. **You want to print out the debug info using .what().**

```
int * myArr;
try {
     myArr = new int[1000000000];
} catch (std::bad_alloc& e) {
     cout << “Something went wrong: ” << e.what << endl;
}
```

For reference, lets say we wanted to throw an exception bad_alloc:

```
int * getMallocArrayOfSize(size_t size) {
     // Deny anything over 10 mb or more.
     if (size >= 10240000) {
          throw new std::bad_alloc();
     }
     return new int[size];
}
```

See https://en.cppreference.com/w/cpp/memory/new/bad_alloc for more info.

19. FIFO is the same as FILO
    a. True
    b. False // First In *First* Out is not First in *Last* Out
20. FIFO is comparable as FCFS
    a. True // First in First Out is the same as First Come First Serve
    b. False
21. We can use a BST to sort, reverse, and maintain order of an input vector.
    a. We can only maintain the order // No way to guarantee the order is maintained. There are special cases where this will occur.
    b. We can only reverse it, the original order is lost // No way to guarantee we can reverse it. The only special case this will is occur is: the input array is reversed, then we call in order traversal. *NOTE: There is a way to get the reversed array (RIGHT DATA LEFT), but we did not cover it in this class.*
    c. <u>We can only sort it, the original order is lost</u> // This is the only guaranteed option. We would insert everything from the array into the tree. Then we call in order traversal. This will give it to us in order.

d. We can do all of them

22. FIFO is used in the following data structures:
    a. Tree
    b. Queue // We enqueue to the back, and dequeue from the front. On the first enqueue, these are the same. After two, it is easy to see the FIFO flow
    c. List
    d. Stack
23. FILO is used in the following data structures:
    a. Tree
    b. Queue
    c. List
    d. Stack // We push to the top and pop from the top, so the first item in will not be released until we pop everything else.
24. Find any issues with the enqueue code.

```
~~Queue~~ enqueue(T value) {
^^^^^ Should be bool or void, depending on reqs.
  ^^^^ Should be Queue:: as enqueue is a part of the Queue
class.
     Node* newNode = new Node(value); <<<< Missing ending ;
     if (rear == nullptr) {
          front = rear = newNode;
     } else {
          rear->next = newNode;
          rear = newNode;
     }
}
```

25. Find any issues with the dequeue code.

```
void dequeue(T out) {
              ^^^ We cannot return a value without a reference.
 ^^^^ Should be Queue:: as dequeue is a part of the Queue class.
     if (front == nullptr) {
          abort("front is null ptr");
          ^^^^^ No need to abort. We simply return and do not
     set out.
     }

     out = front;
     ^^^^^ We cannot possibly set Node * to type T. We should
use the data attribute of front to get T (if needed, dereference
this value).
```

```
    front = front->next;

    delete temp;
          ^^^^^ temp is undefined. We should set this below the
line out = front.
    if (front == nullptr) {
        rear = nullptr;
    }
}
```

26. What is the proper way to cast an int to a double?
    a. int x = y; // Valid, but not prefered
    b. int x = (int)y; // Valid, but not prefered
    c. int x = static_cast<int>(y); // In C++, we should use this style
    d. int x = dynamic_cast<int>(y); // Used for polymorphism scenarios, not static casts like int → double → float etc.
27. It is valid to use C code in C++
    a. True // Yes, we can use any C code in C++, although we should update some things per standards. For example, all C headers should be included as <cLIB> whereas in C it is <LIB.h>
    b. False
28. It is valid to use C++ code in C
    a. True
    b. False // No, C++ builds on C is not compatible with C compilers.
29. Which line is correct to override a pure virtual function in C++?
    a. void f() override {} // Override goes after the prototype and before the body.
    b. override void f() {} // Not valid in C++, valid in some other languages
    c. void override f() {} // Not valid in C++
    d. void f() {} override // Not valid in C++
    e. @Override void f() {} // Not valid in C++, valid in some other languages
30. What does delete do?
    a. Deallocate memory // The destructor is called, then the memory is deallocated.
    b. Allocate memory
    c. Call the constructor
    d. Call the destructor // The destructor is called, then the memory is deallocated.
    e. None of the above (describe correct answer)
31. What does new do?
    a. Deallocate memory

   b. **Allocate memory // To use dynamic memory (on the heap), we must first call new. In C, this is like malloc, calloc, or some other family functions. These calls are still valid in C++.**
   c. **Call the constructor // Called after the memory was allocated, all included with the keyword new. If we were to use a C-style malloc allocation, then the constructor is not called since malloc simply allocates space and has no idea on types that are used.**
   d. Call the destructor
   e. None of the above (describe correct answer)

32. In C++, we can inherit from 500 base classes
   a. **True // We can inherit from 500 base classes. *Unofficial* sources claim C++11 standard permits upto 1024 classes to be inherited with 16384 total when X inherits from Y inherits from Z (so X inherits from Z)**
   b. False

33. In C++, what problems do we encounter with inheritance?
   a. None
   b. Tree Problem
   c. Circular Problem
   d. **Diamond Problem // When we inherit from multiple bases, those bases can inherit from the same base.**
   e. Ruby Problem

34. What is wrong with this BST's private insert function?

Specifications:

- There must be a public interface which the developer will call insert(50). The caller should expect a Boolean value which indicates if it failed to insert. It does not matter if it failed to allocate memory or if it already exists.
- Within the private function, we want to set the left and right. It should return the same value as the public interface.
- The node contains getters and setters for the left and right node. The function to get the value is value().

/// Note: The public function is good

// Note on my changes: There is no one way to do this. I also should have created another variable holds the new node allocation. I chose to use one variable since it would save some space for the recursive calls, allowing for more calls in memory constrained systems.

// While it is not specified, passing by reference is always a good idea to save size of frames in the call stack.

```
~~Node*~~ bool insertRec(Node*& root, int& value) {

    Node * next = nullptr;

    if (root == nullptr) {
        next = new Node(value);

        if (next) {
            root = next;
        }

        ~~return new Node(value);~~
    } else if (~~key~~ value < root->value()) {
        next = root->getLeft();
        if (next) {
            return insetRec(next, value);
        } else {
            next = new Node(value);
            root->setLeft(next);
        }
        ~~root->setLeft = insertRec(root->left, key);~~
    } else if (~~key~~ value > root->~~key~~ value()) {
        next = root->getRight();
        if (next) {
```

```
            return insetRec(next, value);
        } else {
            next = new Node(value);
            root->setRight(next);
        }

        root->setRight = insertRec(root->right, key);
    }

    return root next == nullptr;
}
```

Describe here:

We were using the incorrect variable names to check comparisons (using `key`, but now we use `value`). Also, to set the root, we would return the node directly. The specs we are given want us to return success. Now, we return a bool at the very end.

f the value == some value already in the tree, then no case will be entered. This Since we set next to nullptr, the return statement will return false when inserting a duplicate value.

As such, we also change the return type to bool from Node*.

35. What is the correct order for in-order transversal?
    a. **LEFT DATA RIGHT // We use in-order traversal since the minimum will be the left most item, {THE DATA WE ARE AT}, then the right of that least value. We will keep unwinding this recursion until we hit root. Then we go to the right, and start the process over again going to the least node. When we traverse on the right, we handle the data before going to the right because the current node is less than the node to the right.**
    b. LEFT RIGHT DATA
    c. RIGHT LEFT DATA
    d. RIGHT DATA LEFT
    e. DATA LEFT RIGHT
    f. DATA RIGHT LEFT
36. What is the correct order for pre-order transversal?
    a. LEFT DATA RIGHT
    b. LEFT RIGHT DATA
    c. RIGHT LEFT DATA

d. RIGHT DATA LEFT
e. **DATA LEFT RIGHT // Pre-order means we handle the data before we move. We always move to the left before the right. Useful if we wanted to preserve the data in an array to duplicate the tree at a future point in time.**
f. DATA RIGHT LEFT

37. What is the correct order for post-order transversal?
a. LEFT DATA RIGHT
b. **LEFT RIGHT DATA // We want to go all the way to the least node, then traverse to the right all the way. This ensure we are at a leaf. We process the data last. Used commonly in a tree destroy function. We do not want to delete the node we are working on. While *theoretically*, we could use in order, it would be absolute horrible practice, assuming there is nothing using memory, and assuming that there is no memory cleanup.**
c. RIGHT LEFT DATA
d. RIGHT DATA LEFT
e. DATA LEFT RIGHT
f. DATA RIGHT LEFT

38. When do we use pre-order traversal?

We use pre order to get the order of which it currently stands in the tree. Detailed info on traversals: https://en.wikipedia.org/wiki/Tree_traversal

39. When do we use post-order traversal?

When we want to delete the tree. This will ensure we do not crash. Detailed info on traversals: https://en.wikipedia.org/wiki/Tree_traversal

40. When do we use in-order traversal?

When we want to get the ordered version of the tree. Detailed info on traversals: https://en.wikipedia.org/wiki/Tree_traversal

41. We can use a queue to implement an array.
a. **True // Yes, but resizing is very inefficient, allocates space up front**
b. False

42. `this` keyword is an object.
a. True
b. **False // It is a pointer to the current object, but it does not hold an object itself.**

43. When we overload an operator, we can provide the precedence.
    a. True
    b. False // The precedence will always be the same, we cannot change this using an overloaded operator.
44. Within an overloaded operator, say for `+`, we must use the `+` symbol within the body, otherwise we get a compiler error.
    a. True
    b. False // No, we are simply creating an operator which we can use + in our code so that the complex addition occurs in this function. While you may need to use +, it is not required.
45. Suppose class A and B are defined. B inherits from A as such `class B : A {…};`. Now, we write `B newClass = A()`. This is valid.
    a. True
    b. False // We cannot possibly convert A into B.
46. Suppose class A and B are defined. B inherits from A as such `class B : A {…};`. Now, we write `A newClass = A()`. This is valid.
    a. True // Of course, A is an A
    b. False
47. Suppose class A and B are defined. B inherits from A as such `class B : A {…};`. Now, we write `A newBClass = B()`. This is valid.
    a. True // B is an A. In this case, we cannot use functions/variables within B without casting it to a B (dynamic_cast).
    b. False
48. Suppose class M is fully functional and has a destructor. Within M, we allocate memory that needs to be destroyed. We declare it as `M* m = new M()`. Now, we are done with m and destroy it like `free(m)`. The following occurs:
    a. Compiler Error
    b. Runtime Error
    c. Memory Leak // If there are pointers/memory allocated within m, then there could be a memory leak of that data since we no longer can access it.
    d. Dangling Pointer // We cannot tell based off the current code, so this may or may not exist. If we never set m to nullptr after the call to free, then it is a dangling pointer
    e. No Error // No error will come from it naturally, but it could cause issues as described above.
49. Suppose class M is fully functional and has a destructor. Within M, we allocate memory that needs to be destroyed. We declare it as `M* m = new M()`. Now, we are done with m and destroy it like `~(*m); free(m)`. The following occurs:

    a. **Compiler Error // This is not the proper way to call the destructor, so it will not be able to compile.**
    b. Runtime Error
    c. Memory Leak
    d. Dangling Pointer
    e. No Error

50. Suppose class M is fully functional and has a destructor. Within M, we allocate memory that needs to be destroyed. We declare it as `M* m = new M()`. Now, we are done with m and destroy it like `m->~M(); free(m)`. The following occurs:
    a. Compiler Error
    b. Runtime Error
    c. Memory Leak
    d. Dangling Pointer
    e. **No Error // This is the process that delete takes, so there are no issues. Standards state this is very poor practice, but it does work.**

51. All data structures are linear because non-linear structures are too complicated to program with efficiency.
    a. True
    b. **False // BST is an example of a non-linear data structure. However, non-linear data structures may be more complex.**

52. Polymorphism is the same as inheritance.
    a. True
    b. **False // They go hand in hand, but they are not the same concept**

53. What is required for a class to be considered abstract?
    a. It has at least one *virtual* function
    b. It has more than one *virtual* function
    c. It has all *pure virtual* functions
    d. **It has at least one *pure virtual* function // If there are 1 or more pure virtual functions, then the class is considered abstract.**
    e. It has more than one *pure virtual* function
    f. It has *virtual* constructors // Important: This is not possible!
    g. It has *virtual* destructors

54. Is there anything wrong with these constructors? If so, what is wrong?

```
virtual MyClass() {
        cout << "Setting up…" << endl;
        … set all members …
}
virtual MyClass(MyClass* copy) {
        cout << "Copying class…" << endl;
        … copy over members …
}
```

Constructors cannot be virtual in C++. Virtual means that we do not precisely know the data type. But, if we do not construct a data type, then we cannot know that data type, so we simply cannot have a virtual constructor.
https://www.geeksforgeeks.org/advanced-c-virtual-constructor/

55. What will be implicitly called when the operator is invoked?

```
operator<<(ostream& lhs, Data& rhs);
```

Note: There is a missing return type, this was to not spoil the next question.

There is nothing implicitly called, & passes everything by reference. If there were no &, then the objects would call the copy constructor. Note: for stream types, this will not compile because the copy constructor is deleted.

56. What is wrong with the implementation below?

```
ostream& operator<<(ostream& lhs, Data& rhs) {
     lhs << rhs.data() << endl;
}
```

There is no & for the return. ostream's copy constructor is deleted, see https://cplusplus.com/reference/ostream/ostream/ostream/ for more info.

57. What is wrong with the following reverse function (of a singly linked list)?
Note: The members of Node are public

```
Node* reverse(Node* cur) {
    Node * last = cur;
    if (cur != nullptr) {
        last->next = nullptr;
        cur = reverse(cur);
        cur->next = last;
    }
    return cur;
}
```

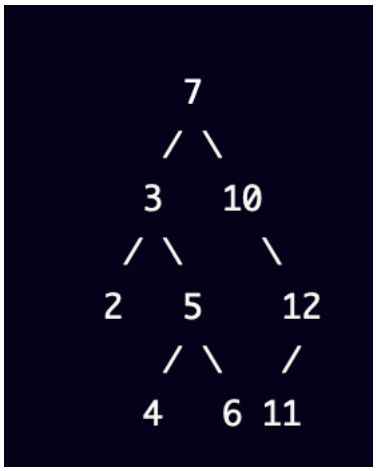
First off, this is infinite recursion. We pass cur into the call to itself and never progress it. Second off, we never have a chance to reverse the list.

We can solve this in two ways:

1. **add a param which holds the previous pointer and set the variable using the previous argument. Ultimately, we will return the head of the list. (Play around with this if you need clarification)**
2. **Add a base case which determines if the next pointer is null. If the next pointer is null, then we want to link the cur's next to last and keep returning cur. Eventually, the list will be returned.**

58. When would we want to use polymorphism?
    a. When we want to look fancy
    b. When we want to use more generic base classes
    c. When we want to use write less code to do more
59. When we inherit from multiple classes, how to we access a function in B1 instead of B2 from C `class C: B1, B2 {…};` (From the perspective of C)?
    a. dynamic_cast<B1>(this)->f1();
    b. ((B1)(this))->f1();
    c. __treat_as_base__((B1))
       this->f1();
    d. this->B1.f1(); // The only way is to use this-> and explicitly call B1's f1 since that is the class we want to call. Without the explicit B1, it would not be possible to call f1 since B1 and B2 hold that symbol. C++ makes you name the explicit class before calling the function.
    e. this->B1->f1();
    f. Something else
60. Who can access private members of class A?
    a. private, protected, public members of A
    b. private members of B (which inherits from A)
    c. private, protected, public members of B (which inherits from A)
    d. friends of A
    e. all classes which inherit via public or protected
61. Which of the following protection levels are valid in C++?
    a. private
    b. protected
    c. public
    d. fileprivate
    e. internal

62. What is the output of the code below?
    a. preOrderTraversal() => 7 3 2 5 4 6 10 12 11
    b. postOrderTraversal() => 2 4 6 5 3 11 12 10 7
    c. inOrderTraversal() => 2 3 4 5 6 7 10 11 12



Credit: GPT-3.5 Turbo

63. What is the output of the following code?

/// Node has last, next, and data

```
void foo(Node * n) {
     for (;n != nullptr; n = n->next) {
          cout << n->data() << endl;
     }
}
```

This simply prints the data of each node in the list, starting at n.

64. What is happening in the following code?

/// Node has last, next, and data

```
void bar(Node * n) {
     Node * p = nullptr;
     for (;n != nullptr; n = n->next) {
          if (p) {
               barfoo(p,n); // Assuming we are never at list end
               p = nullptr;
          } else {
               p = n;
          }
     }
}
void barfoo(Node * p, Node * n) {
     Node * pL = p->last,
          * pN = p->next,
          * nL = n->last,
```

```
            * nN = n->next;
        if (pL) {
            pL->next = n;
        }
        if (pN) {
            pN->last = n;
        }
        if (nL) {
            nL->next = p;
        }
        if (nN) {
            nN->last = p;
        }
          n->last = p->last;
          n->next = p->next;
          p->last = nL;
          p->next = nN;
}
```

We are going through the list (which n is a part of) and swapping every other node. In essence, we are calling barfoo (the function which swaps) when counter is even, if we were to add a counter.