CptS 122 Spring 2025 Final Review (Solutions)

Question 1

Examine the following code snippet and identify any errors present:

```
class A {
   int foo() const {
      return x = 10;
   }

   int bar() {
      return x = 49;
   }

   void reset() {
      x = -1;
   }

   int x;
};
```

Indicate the issues you found in the code and provide suggestions for how to fix them.

Two critical issues: (1) x is modified in a constant function; (2) x is uninitialized (no constructor or default value).

1. Refactor code to which no modification is needed in foo, or remove the const qualifier (based on specifications)
2. Ensure there is a constructor of which x is initialized, or provide a default value in the class declaration.

Two readability issues: (1) x is non-descript; (2) value directly returned from assignment.

1. To address this, we should name x as testInt or something which describes it is the focus.
2. Return statements should not consist of an assignment operation. Move the assignment to above the return and simply return x.

Question 2

What is the worst-case time complexity in Big-O notation for inserting an element into a Binary Search Tree (BST)?

Worst-case time is O(n). There is no guarantee a BST will be balanced, but there is a guarantee any node to the left of a given parent node will be less than the parent node, and the right is greater than.

Suppose the numbers inserted must be in the range $0 < x < 1000$. As such, if we insert 999 first, everything will be to the left. If 1-999 were inserted in descending order, the tree will look identical to a singly linked list. Therefore, it is possible to achieve O(n) time.

Question 3

What is the worst-case time complexity in Big-O notation for the **enqueue** operation in a Queue?

*Assume the Queue is implemented using pointers: Node * mpHead and Node *mpTail for the Queue.*

The worst-case time complexity of enqueue **when we maintain a pointer to the tail** is O(1). The operation is simply composed of a few operations independent of input size.

Note: different implementations of a queue may result in different time complexities.

Question 4

What is the worst-case time complexity in Big-O notation for the **dequeue** operation in a Queue?

*Assume the Queue is implemented using pointers: Node * mpHead and Node *mpTail for the Queue.*

The worst-case time complexity of dequeue is O(1) **when we use a linked list**. The operation is simply composed of a few operations independent of input size.

Note: Different underlying data structures of Queue may affect the time complexity and result in O(n) time.

Question 5

Determine the Big-O time complexity for the following functions, assuming a singly-linked list is used as the underlying data structure:

1. BST::search(for: T) O(n)
    a. Search can traverse the entire tree if it is unbalanced and nearly resembles a linear data structure.
2. List::insertAt(index: Int) O(n)
    a. index could be anywhere in the List, including the end.
3. Stack::push(data: T) O(1)
    a. We always push to the end, nothing will ever shift.
4. Stack::pop(store: T&) O(1)
    a. O(n) We always pop from the end. Nothing needs to be shifted.
5. Stack::isEmpty() O(1)
    a. This is a single comparison (or up to a few). Regardless, it is irrespective of the input size.
6. BST::delete(item: T) O(n)

   a. While nothing needs to be shifted, we may have to traverse the whole list to not find item.

7. List::insertAtEnd(item: T) (Assume we have pHead and pTail) O(1)
   a. There is no traversal which occurs since we already have a pointer to the tail.
8. List::insertAtEnd(item: T) (Assume we have pHead) O(n)
   a. As we do not have the tail, we need to traverse the entire list to insert item.
9. List::removeFromFront(item: T) (Implemented using a C-style array) O(n)
   a. Since we are removing an element from the front of a C-style array, everything must shift to the left. Without the shift, data at the end may be lost.

Question 6

Consider the following operations on a List. For each operation, identify the best underlying data structure. Write down any assumptions you have made.

a) **Read** an item at position n

A primitive array and std::vector produce the lowest time complexity of O(1) for read operation. Whether a primitive array or std::vector should be used depends on the runtime constraints. If it is preferred to have a dynamic size, then std::vector is best. If we are guaranteed to have no more than x elements, then a primitive array would best.

b) **Insert** an item at position n

A singly or doubly linked list would be best as primitive arrays and std::vector require shifting of elements to insert at position n (unless n is the end).

Singly and doubly linked lists allow us simply relink nodes, leading to O(1) time.

c) **Remove** an item from the front

Again, singly and doubly linked lists will be the best option as we update head to be the second node.
However, if we are using circular linked lists, then a doubly link list is the best option as we need to iterate over the entire list to update the next pointer. Doubly linked would permit us to access the last node and update the next ptr of the end node with O(1) time. At the same time, the definition of head and tail become gray as it is circular.

d) **Add** an item at the end

Primitive array and std::vector are traditionally the best option for this as they achieve O(1) time. There could be some overhead with using std::vector if it needs to grow, but we still consider this O(1) complexity.
If we are using circular doubly linked lists, then doubly linked lists can perform O(1) complexity. At the same time, the definition of head and tail become gray as it is circular.

We have the following options:

1. Singly linked ~~nodes~~ lists
2. Doubly linked ~~nodes~~ lists
3. Primitive array
4. std::vector

Question 7

Examine the following pseudocode and describe its functionality. Check your answer by writing C++ code.

```
Init stack with 200 numbers
Init empty list
While ( the stack is not empty ) do
        Pop element from the stack and put at the **FRONT** of the list.
Done
While ( the list is not empty) do
        Remove element from the **END** of the list and push to the stack
Done
```

This will reverse a stack.

Output:

```
PRE: 5 -> 4 -> 3 -> 2 -> 1 ->
POST: 1 -> 2 -> 3 -> 4 -> 5 ->
```

Code:

```
int main(void) {
   stack<int> s;
   list<int> l;
  // Push some data to s here
   while (s.empty() == false) {
      l.push_front(s.top());
      s.pop();
   }

   while (l.empty() == false) {
      s.push(l.back());
      l.pop_back();
   }
}
```

Question 8

Examine the following pseudocode and describe its functionality. Check your answer by writing C++ code.

```
Init stack with 200 numbers
Init empty list
While ( the stack is not empty ) do
        Pop element from the stack and put at the **FRONT** of the list.
Done
While ( the list is not empty) do
        Remove element from the **FRONT** of the list and push to the stack
Done
```

This stack will remain unmodified.

Output:

```
PRE: 5 -> 4 -> 3 -> 2 -> 1 ->
POST: 5 -> 4 -> 3 -> 2 -> 1 ->
```

Code:

```
int main(void) {
  stack<int> s;
  list<int> l;
 // Push some data to s here
  while (s.empty() == false) {
    l.push_front(s.top());
    s.pop();
  }

  while (l.empty() == false) {
    s.push(l.front());
    l.pop_front();
  }
}
```

Question 9

What pointer(s) **will** be modified when the Queue::dequeue() is called on a **non-empty** queue?

1. Tail
2. Head
3. None
4. Head & Tail
5. None of the above (*Please describe what you believe the answer is*)

5 is correct. There is no guarantee 2, 3, or 4 will always be true. I assume dequeue uses defensive programming. With this assumption, it is possible that no pointers change, which makes it possible for 3 to occur. 2 can be true; this occurs when the queue has two or more elements. 4 is only true when there is a single element in the queue and we need to set both to null.

Question 10

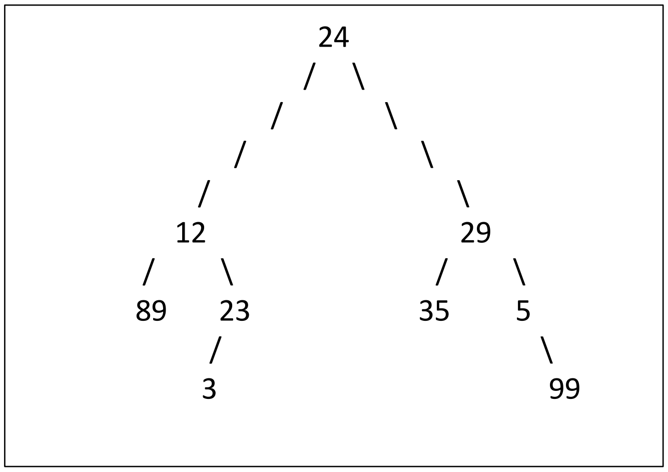
What is the difference between a binary tree and a binary search tree?

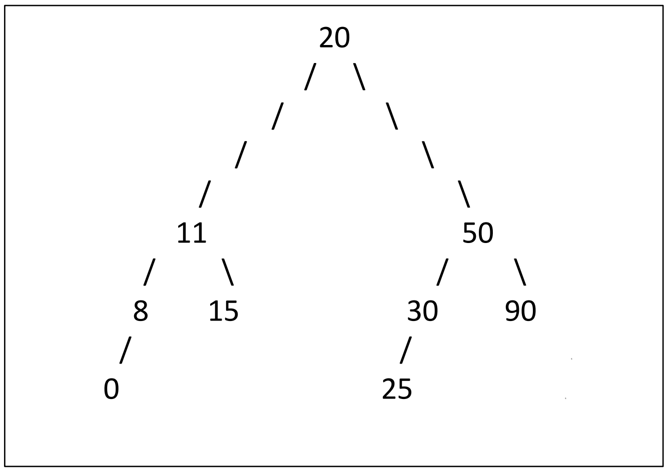
A **binary search tree** guarantees order. It ensures for any given node, all values in the left subtree are less than the node's value, while all values in the right subtree are greater than or equal to the node's value. This property allows for efficient searching, insertion, and deletion operations, as the tree maintains a sorted structure.

A **binary tree** has at most two child nodes. There is no guarantee of order among the nodes. While all binary search trees are binary trees, not all binary trees are binary search trees.

Question 11

For the given diagrams, identify the pre-order, post-order, and in-order traversals. Additionally, specify the names of these trees.





BINARY TREE | **BINARY SEARCH TREE**

PRE: 24 12 89 23 3 29 35 5 99 | PRE: 20 11 8 0 15 50 30 25 90
POST: 89 3 23 12 35 99 5 29 24 | POST: 0 8 15 11 25 30 90 50 20
IN: 89 12 3 23 24 35 29 5 99 | IN: 0 8 11 15 20 25 30 50 90

Important note: While in-order traversal is a common method used for binary search trees, it is not traditionally applied to general binary trees, as they do not have a guaranteed order.

Question 12

Briefly describe each of the following traversals for a Binary Search Tree (BST). For each traversal, please provide a use case:

1. Pre-order
   a. Process the current node, traverse left, then traverse right.
   b. Use-case: Serialization of data for storage. Simply put, this allows for the tree to be stored in an order which can rebuilt from a file.
2. Post-order
   a. Traverse left, traverse right, process the current node.

   b. Use-case: Tree destruction. This ensures we do not use any deallocated data. The left and right subtrees are guaranteed to be deallocated prior to the current node.
3. In-order
   a. Traverse left, process the current node, traverse right.
   b. Use-case: We want to find a specific data member or sort a list.

Question 13

Explain how memory is allocated for pointers and arrays in C++. Additionally, describe the process for deallocating memory in C++.

In C++, memory for pointers and arrays is allocated using the new operator. For non-array types, memory is allocated by using new followed by the type, which also calls the constructor (if necessary). For example, Data* myData = new Data(); allocates memory for a single Data object. To allocate memory for an array, the syntax consists of new, square brackets enclosing the number of elements, and the type. Data* myArr = new Data[10]; allocates memory for an array of 10 Data objects.

Deallocation requires the use of delete. For non-array allocations, use delete[] myArr. For array allocations, use delete myData.

There are two type of "smart" pointers which automatically handle this for developers. They are shared_ptr and unqiue_ptr.

Question 14

Complete the following protection mode matrix. The rows represent the inheritance mode in the derived class, while the columns represent the declared member's protection level in the base class.

Protection Mode	public	protected	private
public	public	protected	INACCESSIBLE
protected	protected	protected	INACCESSIBLE
private	private	private	INACCESSIBLE

Question 15

From the following options, identify the "Big Three" components in C++:

1. Constructor
2. Virtual destructor
3. **Copy Constructor**
4. Getters/setters
5. **Destructor**
6. Default Constructor
7. Virtual functions
8. **Overloaded assignment operator**
9. Encapsulation

Question 16

What are some reasons a data structure might provide two different implementations of operations such as insert, enqueue, search, etc.? Please explain your reasoning.

A public and private version enable procedural abstraction. It allows developers to hide the true implementation behind the scenes. The same data structure can use different implementations on the inside without affecting outside code.

It also enables extendibility since a public interface or abstract class can be defined with a set of functions. Then, the abstract class or interface can have multiple concrete implementations with different underlying data structures. The user will only know differences based on the class name.

Question 17

What is the term used to describe a class that contains one or more pure virtual functions?

Abstract class. A class with a pure virtual function *cannot* be instantiated (created in memory).

Question 18

What is the purpose of the keyword "virtual" in C++?

Enables runtime polymorphism. It also permits for functions to be overwritten in a derived class. When using pointers or references, the most specific overwritten definition of a function will be used. The most specific overwritten definition is the function which is defined in the highest-level derived class.

Suppose class C inherits from B which inherits from A. All classes have a virtual function foo(). If we have a pointer type of A, and the instantiation is of type class C, suppose foo is called on the pointer. Class C's version will be called.

Question 19

What are the different types of inheritance in C++? Provide a brief description of each type.

Public – Everybody can access the member.

Protected – Only itself and classes which inherit from the class can access the member.

Private – **Only itself** can access the member.

Question 20

Define the terms "upcasting" and "downcasting" in the context of object-oriented programming.

Upcasting – Treating a derived class as its base class. From the example in lecture, Manager is treated as an Employee. Will always succeed (assuming it inherits from the base class).

Downcasting – Treating a base class as one of its derived classes. This process can fail. Assume we have a base class Animal. We have classes Dog, Cat, Fish which inherit from Animal. If we create a Dog and store it as an Animal, then attempt to cast it to a Cat, this will fail.

Up- and down-casting come from the hierarchy shown in UML diagrams.

Question 21 (a)

We want to create a Parser class with very basic functionality. The constructor will accept a string. There will be a parse function which accepts a vector of strings. We can only parse up to 1000 characters and generate up to 500 tokens.

Declare the following members of Parser.

1. A constructor which accepts a reference to a constant string.
2. A function called parse which accepts a std::vector<string> by reference to place tokens.
3. A private data member which holds the string to parse.

```
class Parser {
public:
    Parser(const string& csvLine);

    void parse(std::vector<string>& tokens);

private:
    string line;
};
```

Important: declare means write the protypes, do not implement them. If you are instructed to declare a class in an interview and instead implement them, that could be very bad. When working (job or internship), it could waste time and lead to duplicated code.

Question 21 (b)

Define the constructor for Parser. We have defined an exception, InputLengthExceededException, which inherits from std::runtime_exception. If the input is too long, throw an exception and provide a detailed error message.

```
Parser::Parser(const string& csvLine) {
    if (csvLine.length() > 1000) {
        throw InputLengthExceededException("Input is too long to parse!");
    }

    this->line = csvLine;
}
```

Question 21 (c)

Define the parse function. We have created an exception called InputLengthExceededException, which derives from std::runtime_exception. If an excessive number of tokens are produced, throw an InputLengthExceededException with a detailed message. Assume that the input string is in CSV format and does not contain any commas within the values. *Write down assumptions*.

Hint: use s1.substr(i, c) and s1.find("str")
S1.substr(i,c) will return a string with `c` characters from index i.
S1.find("str") will return an index where the substring "str" was found. If "str" is not found, std::string::npos is returned.

Example:

Int index = S1.find(",")

S1 = "Hello, World!"

Hi = s1.substr(0,index);

World = s1.substr(index);

Cout << Hi << endl; // output: `Hello`

Cout << World << endl; // output: `, World!`

```
void Parser::parse(std::vector<string> &tokens) {
  std::size_t start = 0;
  std::size_t end = this->line.find(",", start);

  while (end != std::string::npos) {
    tokens.push_back(this->line.substr(start, end - start));

    start = end + 1;
    end = this->line.find(",", start);
  }

  if (start < this->line.length()) {
    tokens.push_back(this->line.substr(start));
    }
}
```