Contents

1 Minor Grievances

1. Do not use `NULL`. Use `nullptr` instead. For whatever reason, Shahadat has them mixed up and has been telling us that using `nullptr` will create errors, when it is actually `NULL` that will lead to more problems. `nullptr` was added in C++11 and is better in virtually every possible circumstance, except when making C compatible C++ code.

2. Do not use `free()` on objects that were not allocated memory with `malloc()`. I'm betting you don't even know what `malloc()` is, so just don't use `free()`. You should only be using `delete` or `delete[]`. `free()` does not call destructors, and I shouldn't have to explain why not calling an object's destructor is bad.

3. The implementation of singly linked lists that is given in this book is stupid. `append()` will always add something to the end of the list, even if that makes it unsorted. On its own, that isn't a problem, but the `insert()` makes it stupid, since `insert()` will try to insert elements in a sorted order with how the linked list is implemented here. A final complaint is that I also just don't like the naming practices they use for the various member functions. There is literally no point in having `'Node'` at the end of every member function, except to distinguish it from `delete`.

2 Topic List

2.1 Chapter 16

2.1.1 Exceptions

Vocabulary

A. **`throw` Keyword**: A keyword that allows you to throw an error. It must be provided an argument to be passed to it to do anything useful.

B. **Throw Point**: The line containing the throw statement.

C. **Exception Handler**: The part of the program that is executed once a throw statement executes.

D. **Try/Catch Construct (a.k.a. Try/Catch Block)**: The part of the program that will handle any exceptions that are thrown. It is composed of a try block and as many catch blocks as necessary. The try block is where any functions that may throw errors are run, and the catch block "catches" the exceptions that are thrown by the try block.

E. **Exception Parameter**: The argument that is given with a throw statement.

F. **Exception Class**: A class that is defined within another class and is designed to be used to throw particular errors in the program. Sometimes they are empty classes, while other times they will have data that can be extracted to know more about the nature of the error that occurred. When the class is not empty, it is populated with members just like any other class, but usually only has members that are especially relevant for that type of error.

G. **Unwinding the Stack**: The process that occurs when nested functions throw exceptions that are not caught. The process continues until the program reaches the try block. If the member function of a a class object throws an error, then that object will have its destructor called. Any objects that were created in the try block will also have their destructors called.

H. **Rethrowing Exceptions**: When an error is thrown by a catch block after it was already thrown by something in the try block. This process is necessary when multiple catch blocks need to perform their operations when a particular error is thrown.

Important Concepts

A. Exceptions signal errors or unexpected events that occur while a program is running.

B. Returning pre-determined values to signal that an error has occurred has drawbacks, particularly when such a value is a possible valid result of the function.

C. When a function throws an exception, it is automatically aborted.

D. If an exception is not caught, whether that be because no catch blocks match the exception parameter or an exception is thrown outside of a try block, the entire program will stop immediately.

E. In order to handle multiple possible exceptions, multiple catch blocks must be declared that will handle every different exception that can be thrown.

F. Try/Catch constructs can allow the program to recover from errors that would have terminated it previously.

G. It is possible for try blocks to be nested. When this is done, it is sometimes necessary to rethrow exceptions, depending on the nature of the error.

2.1.2 Function Templates

Vocabulary

A. **Function Template**: A "generic" function that is able to work with any data type. Parameters are used in place of explicit data types. The true data type of the parameters will automatically be filled in by the compiler whenever it encounters a call to the function.

B. **Template Function**: The code that is generated by the compiler when function templates are used to create multiple versions of the same function for different data types.

C. **Template Prefix**: The beginning of a function template that begins with with the `'template'` keyword and is followed by angled brackets with class and a parameter name for the generic data type. If multiple generic parameters are used, the process is repeated inside the angled brackets and a different name is assigned to the parameter.

D. **Type Parameter**: The thing that is used to specify the generic data type.

Important Concepts

A. Function templates allow you to avoid having to overload functions when writing functions perform the exact same operations on different data types.

B. Any objects that are passed to functions that use a given operator must overload the operators that are needed by the function, or else the template function will thrown an error.

C. Function templates can be overloaded as long as the two functions have different parameter lists.

2.1.3 Class Templates

Vocabulary

A. **Specialized Templates**: A template that is designed to work with a specific data type. In the declaration of the template, the actual data type is used instead of a type parameter.

Important Concepts

A. Templates can be used to create generic classes and abstract data types without having to write duplicated code for handling multiple data types.

B. Class templates are declared similarly to function templates, where `template <class T>` is placed before the class declaration. The parameter `T` (or whatever the identifier is) can then be used like any other data type in the class.

C. In order to define objects with class templates, you need to use `ClassName<TypeParameter> objectName`

D. Class templates can be used in conjunction with inheritance by passing the type parameter to the base class that uses the class template.

2.2 Chapter 17

2.2.1 Introduction to the STL

Vocabulary

A. **Container**: An object that holds a collection of values or other objects

B. **Iterator**: An object that is used to iterate over the items in a collection, providing access to them.

Important Concepts

A. **STL**: The Standard Template Library. Contains an extensive library of templates to use.

B. **Container**: A type of class template for objects to that store and organize data.

C. **Iterator**: A type of class template for objects that behave like pointers and are used to access the individual elements in a container.

D. **Algorithm**: A type of function template that is used to perform operations on elements of containers.

2.2.2 STL Container and Iterator Fundamentals

Vocabulary

A. **Sequence container**: A type of container that stores its data in memory sequentially, similarly to an array.

B. **Associative container**: A type of container that stores its data in a non-sequential manner that makes it faster to access the elements therein.

C. Sequence Containers:

   (a) **`array`**: A fixed-size sequence container that is similar to an array.

(b) **deque**: A double-ended queue. Like a vector but designed such that values can be quickly added or removed from either end.

(c) **forward_list**: A single linked list of data elements. Values may be inserted to or removed from any position.

(d) **list**: A doubly linked list of data elements. Values may be inserted to or removed from any position.

(e) **vector**: Similar to an array, except its size may be changed to accommodate any number of elements

D. Associative Containers:

(a) **set**: A set of unique, sorted values that does not allow duplicates.

(b) **multiset**: A set of of unique, sorted values that allows duplicates.

(c) **map**: A set of key-value pairs that is sorted by its keys and does not allow duplicates.

(d) **multimap**: A set of key-value pairs that is sorted by its keys and allows duplicates.

(e) **unordered_set**: Like a set, but elements are unsorted.

(f) **unordered_multiset**: Like a multiset, but elements are unsorted.

(g) **unordered_map**: Like a map, but elements are unsorted.

(h) **unordered_multimap**: Like a map, but elements are unsorted.

E. Adapter Classes:

(a) **stack**: Stores elements in a deque by default. It is a last-in, first-out container.

(b) **queue**: Stores elements in a deque by default. It is a first-in, first-out container.

(c) **priority_queue**: Stores elements in a vector by default. Retrieved elements are always those with the greatest value.

F. Types of Iterators:

   (a) **Forward**: Moves forward in a container (uses `'++'`)
   (b) **Bidirectional**: Moves forward or backward in a container (uses `'++'` and `'--'`)
   (c) **Random Access**: Can move forward, backward, or jump to a specific element.
   (d) **Input**: Can be used with an input stream to read data.
   (e) **Output**: Can be used with an output stream to write data.

G. **`const_iterator`**: provides read-only access to the elements in a constant container. Only a `const_iterator` can be used with a constant container. They can be obtained with the `'c'` prefixed iterator member functions, if applicable.

H. **`reverse_iterator`**: is either bidirectional or random-access and works in reverse. The first element is considered the last element and the last element is considered the first element. The ++ operator moves this iterator backward and the – operator moves it forward. Array, deque, list, map, multimap, set, and vector support reverse iterators, and they can be obtained with the `'r'` prefixed iterator member functions.

I. **`const_reverse_iterator`**: is a read-only version of `reverse_iterator`. They can be used with any container that supports reverse iterators and can be obtained with the `'cr'` prefixed member functions.

Important Concepts

A. Container adapter classes adapts another container to a specific use.

B. The `[]` operator of the `array` class does not perform bounds checking, while the `at()` member function does.

C. `array`, `vector`, and `deque` use random-access iterators.

D. `list`, `set`, `multiset`, `map`, and `multimap` use bidirectional iterators.

E. `forward_list` and unordered containers use forward iterators.

F. `auto` can often be used to simplify the declaration of an iterator.

2.2.3 The `vector` Class

Vocabulary

A. **Default Constructor**: `vector<type> name`; Creates an empty vector.

B. **Fill Constructor**: `vector<type> name(size)`; Creates an vector of size *size*. If the elements are objects, they are initialized via their default constructor. Otherwise, they are initialized with 0.

C. **Fill Constructor**: `vector<type> name(size, value)`; Creates a vector of size argument and fills every index with the value argument.

D. **Range Constructor**: `vector<type> name(iterator1, iterator2)`; Creates a vector and populates it with a range of elements from another container based on iterator1 to iterator2.

E. **Copy Constructor**: `vector<type> name(vector2)`; Creates a vector and populates it with the elements from vector2.

Important Concepts

A. Vector overloads `[]` so you can use that to access elements, but it will not perform bounds checking. The `at()` member function does the same, but performs bounds checking.

B. `insert()` and `push_back()` create temporary objects in memory while insertion is taking place. For very large objects, this creates problems. This is solved with the `emplace()` and `emplace_back()` member functions.

C. When adding an object to a vector with `insert()` or `push_back()`, the object must either already exist or you must pass the a constructor as the argument to the function (i.e., `insert(0, ClassName(param1, param2))`). The emplace versions use in-place construction, where you pass the arguments that would otherwise be passed to the constructor directly in the emplace function: `emplace(iterator, param1, param2)`

D. Vectors use a dynamically allocated array internally. Because an array's capacity can be exceeded, vectors typically allocate more memory than is "needed", leading to a vector having two sizes: the number of elements (obtained with `size()`) and its capacity (obtained with `capacity()`). The capacity will always be greater than or equal to the size.

2.2.4 The `map`, `multimap`, and `unordered_map` Classes

Vocabulary

A. **Map**: An associative container that stores its elements as key-value pairs, where the key is used to access the value associated with it. They can be initialized with an initialization list where each element is initialized as the {key, value}. All keys must be unique.

B. **`pair`**: An object that stores a key and value pair within a `map`.

Important Concepts

A. If a value is assigned to an existing key, the new value will replace the old value.

B. Because the elements of a map are stored as a pair, the key is accessed as `element.first` and the value as `element.second` when iterating over it with a range-based for loop.

C. When adding elements with the `insert()` member function, you must pass the key-value pair as arguments in the `make_pair(key, value)` member function of the `pair` class. The emplace version will create the pair for you: `emplace(key, value)`.

D. In order to store an object as a value in a map, it must have a default constructor.

E. In order to use user-defined objects as keys of a map, you must overload the `'<'` operator to allow the class to order the keys.

F. `map` is a binary tree, while `unordered_map` is a hash table.

2.2.5 The `set`, `multiset`, and `unordered_set` Classes

Vocabulary

A. **Set**: An associative container that stores a sorted set of unique values.

B. **Default Constructor**: `set<type> name`; Creates an empty set object.

C. **Range Constructor**: `set<type> name(iterator1, iterator2)`; Creates a set object based on a range of values from another set object that starts at iterator1 and ends at iterator2.

D. **Copy Constructor**: `set<type> name(set2)`; Creates a set object from another set.

Important Concepts

A. Sets use bidirectional iterators.

B. The `count()` or `find()` member functions can be used to determine if a value exists in a set. Count will return 0 or 1 for a set, and find will return an iterator to the element if it exists.

C. In order to store objects in a set, the class must overload the < operator.

D. The `equal_range()` member function will at most return a range with one element when used on a regular set. With a `multiset`, it can return multiple elements. Likewise, the `count()` member function can return values greater than one when used on an `multiset`, but only 0 or 1 on a regular set.

2.2.6 Algorithms

Important Concepts

A. The `sort(it1, it2)` function will sort a range from `it1` to `it2` in ascending order.

B. The `binary_search(it1, it2, value)` function will search for *value* in the range from `it1` to `it2`.

C. In order to use `sort()` and `binary_search()` on a container of objects, the objects' class must overload the < operator.

2.2.7 Introduction to Function Objects and Lambda Expression

Vocabulary

A. **Function Object (Functor)**: An object that looks like a function. They can accept arguments and can be called. In order to create a function object, you need to overload the `()`.

B. **Anonymous Function Object**: Function objects that are created and used without being given a name.

C. **Predicate**: A function or function object that returns a boolean value.

D. **Unary Predicate**: A predicate that takes one argument.

E. **Binary Predicate**: A predicate that takes two arguments.

F. **Lambda Expression**: A way to create a function object without having to write a class declaration. The compiler only generates the function object in memory when it encounters it in code. The basic syntax is `[](`*`params`*`) {` *`function body`* `}`.

2.3 Chapter 18

2.3.1 Introduction to the Linked List ADT

Vocabulary

A. **Linked List**: A dynamically allocated data structure that stores data by connecting nodes together with a series of pointers. Because linked lists are stored non-contiguously, you can add and remove nodes at any point, without having to reallocate memory or copy the list to a new structure.

B. **Self-referential Data Structure**: A data structure that contains a pointer to an object of the same type as is being declared.

Diagrams

I am not making these stupid ass diagrams in LaTeX, so these PNGs will have to suffice.

Important Concepts

A.

2.3.2 Linked List Operations

Vocabulary

A. There are five basic operations that can be performed on a linked list: traversal, appendage, insertion, deletion, and destruction.

Important Concepts

A.

2.3.3 Variations of the Linked List

Vocabulary

A. **Doubly Linked List**: A variation of the regular linked list where each node has two pointers instead of one: one points to the previous node, while the other points to the next node.

B. **Circularly Linked List**: A variation of the regular linked list where each node still only has one pointer, but the last pointer in the chain points to the first node in the list.

3 Textbook Questions

3.1 Chapter 16

3.1.1 Short Answer

1. What is a throw point?
2. What is an exception handler?
3. Explain the difference between a try block and a catch block.
4. What happens if an exception is thrown, but not caught?
5. What is ”unwinding the stack”?
6. What happens if an exception is thrown by a class’ member function?
7. How do you prevent a program from halting when the new operator fails to allocate memory? (Covered by the section on `bad_alloc`. We did not cover this topic.)
8. Why is it more convenient to write a function template than a series of overloaded functions?
9. Why must you be careful when writing a function that uses operators such as `[]` with its parameters?

3.1.2 Fill in the Blank

10. The line containing a throw statement is known as the ________
11. The ________ block contains code that may directly or indirectly cause an exception to be thrown.
12. The ________ block handles an exception
13. When writing function or class templates, you use a(n) ________ to specify a generic data type.

14. The beginning of a template is marked by a(n) ________.
15. When defining objects of class templates, the ________ you wish to pass into the type parameter must be specified.
16. A(n) ________ template works with a specific data type.

3.1.3 True or False

23. Data may be passed with an exception by storing it in members of an exception class.
24. Once an exception has been thrown, it is not possible for the program to jump back to the throw point.
25. All type parameters defined in a function template must appear at least once in the function parameter list.
26. The compiler creates an instance of a function template in memory as soon as it encounters the template.
27. A class object passed to a function template must overload any operators used on the class object by the template.
28. Only one generic type may be used with a template.
29. In the function template definition, it is not necessary to use each type parameter declared in the template prefix.
30. It is possible to overload two function templates.
31. It is possible to overload a function template and an ordinary (non-template) function.
32. A class template may not be derived from another class template.
33. A class template may not be used as a base class.
34. Specialized templates work with a specific data type.

3.2 Chapter 17

3.2.1 Short Answer

1. What two emplacement member functions are provided by the vector class? How are these member functions different from the `insert()` and `push_back()` member functions?

2. What is the difference between the vector class's `size()` member function and the `capacity()` member function?

3. If you want to store objects of a class that you have written as values in a map, what requirement must the class meet?

4. If you want to store objects of a class that you have written as keys in a map, what requirement must the class meet?

5. What is the difference between a `map` and an `unordered_map`?

6. What is the difference between a `map` and a `multimap`?

7. What happens if you use the `insert()` member function to insert a value into a set and that value already exists in the set?

8. If you want to store objects of a class that you have written as keys in a `set`, what requirement must the class meet?

9. What is the difference between a `set` and a `multiset`?

10. How does the behavior of the `count()` member function differ between the `set` class and the `multiset` class?

11. How does the behavior of the `equal_range()` member function differ between the set class and the `multiset` class?

12. What are two differences between the `set` and `unordered_set` classes?

13. When using one of the STL algorithm function templates, you typically work with a range of elements that are denoted by two iterators. To what does the first iterator point? To what does the second iterator point?

14. You have written a class and plan to store objects of that class in a vector. If you plan to use the `sort()` and/or `binary_search()` functions on the vector's elements, what operator must the class overload?

15. What is a function object?

16. If you want to create function objects from a class, what must the class overload?

17. What is an anonymous function object?

18. What is a lambda expression?

3.2.2 Fill in the Blank

19. There are two types of container classes in the STL: ________ and ________.

20. A(n) ________ container organizes data in a sequential fashion similar to an array.

21. A container ________ class is not itself a container, but a class that adapts a container to a specific use.

22. A(n) container stores data in a nonsequential way that makes it faster to locate elements.

23. ________ are pointer-like objects that are used to access data stored in a container.

24. Each element that is stored in a map has two parts: a ________ and a ________.

25. ________ is a map container that allows duplicate keys.

26. the ________ class is an associative container that stores a collection of unique, sorted values.

27. The ________ header defines several function templates that implement useful algorithms.

28. A ________ is a pointer to a function's executable code.

29. A ________ object is an object that can be called, like a function.

30. A ________ is a function or function object that returns a Boolean value.

31. A ________ is a predicate that takes one argument.

32. A ________ is a predicate that takes two arguments.

33. A ________ is a compact way of creating a function object without having to write a class declaration.

3.2.3 True or False

34. The `array` class is a fixed-size container.

35. The `vector` class is a fixed-size container.

36. You can use the `*` operator to dereference an iterator.

37. You can use the `++` operator to increment an iterator.

38. A container's `end()` member function returns an iterator pointing to the last element in the container.

39. A container's `rbegin()` member function returns a reverse iterator pointing to the first element in a container.

40. You do not have to declare the size of a `vector` when you define one.

41. A `vector` uses an array internally to store its elements.

42. A `map` is a sequence container.

43. You can store duplicate keys in a `map` container.

44. The `multimap` class' `erase()` member function erases only one element at a time. If you want to erase multiple elements with the same key, you have to call `erase()` multiple times.

45. All the elements in a `set` must be unique.

46. The elements in a `set` are sorted in ascending order.

47. If the same value appears more than once in the initialization list of a `set` definition, an exception will occur at runtime.

48. The `unordered_set` container has better performance than the `set` container.

49. If two iterators denote a range of elements that will be processed by an STL algorithm function, the element pointed to by the second iterator is not included in the range.

50. You must sort a range of elements before searching it with the `binary_search()` function.

51. Any class that will be used to create function objects must overload the `operator[]` member function.

52. Writing a lambda expression usually requires more code than writing a function object's class declaration and then instantiating the class.

53. You can assign a lambda expression to a variable and the use the variable to call the lambda expression's function object.

3.2.4 Find the Errors

68.

```
array<int, 5> a;
a[5] = 99;
```

69.

```
vector<string> strv = {"one", "two", "three"};
vector<string>::iterator it = strv.cbegin();
```

70.

```
vector<int> numbers(10);
for (int index = 0; index < numbers.length(); index++) {
  numbers[index] = index;
}
```

71.

```
vector<int> numbers = {1, 2, 3};
numbers.insert();
```

72.

```
map<string, string> contacts;
contact.insert("Beth Young", "555-1212);
```

73.

```
multimap<string, string> phonebook;
phonebook["Megan"] = "555-1212";
```

74.

```
vector<int> v = {6, 5, 4, 2, 3, 1};
sort(v);
if (binary_search(v, 1)) {
  cout << "The value 1 is found in the vector.\n";
} else {
  cout << "The value 1 is NOT found in the vector.\n";
}
```

75.

```
auto sum = ()[int a, int b] { return a + b; }
```

3.3 Chapter 18

3.3.1 Short Answer

1. What are some of the advantages that linked lists have over arrays?
2. What advantage does a linked list have over the STL vector?
3. What is a list head?
4. What is a self-referential data structure?
5. How is the end of a linked list usually signified?
6. Name five basic linked list operations.
7. What is the difference between appending a node and inserting a node?
8. What does "traversing the list" mean?
9. What are the two steps required to delete a node from a linked list?
10. What is the advantage of using a template to implement a linked list?
11. What is a singly linked list? What is a doubly linked list? What is a circularly linked list?
12. What type of linked list is the STL `list` container? What type of linked list is the STL `forward_list` container?

3.3.2 Fill in the Blank

13. The ________ points to the first node in a linked list.

14. A data structure that points to an object of the same type as itself is known as a(n) ________ data structure.

15. After creating a linked list's head pointer, you should make sure it points to ________ before using it in any operations.

16. ________ a node means adding it to the end of a list.

17. ________ a node means adding it to a list, but not necessarily to the end.

18. ________ a list means traveling through the list.

19. In a(n) ________ list, the last node has a pointer to the first node.

20. In a(n) ________ list, each node has a pointer to the one before it and the one after it.

3.3.3 True or False

26. The programmer must know in advance how many nodes will be needed in a linked list.

27. It is not necessary for each node in a linked list to have a self-referential pointer.

28. In physical memory, the nodes in a linked list may be scattered around.

29. When the head pointer points to `nullptr`, it signifies an empty list.s

30. Linked lists are not superior to STL vectors.

31. Deleting a node in a linked list is a simple matter of using the `delete` operator to free the node's memory.

32. A class that builds a linked list should destroy the list in the class destructor.

3.3.4 Find the Errors

33.

```
void NumberList::appendNode(double num) {
  ListNode *newNode, *nodePtr;
  // Allocate a new node and store num
  newNode = new listNode;
  newNode->value = num;
  // If there are no nodes in the list
  // make NewNode the first node.
  if (!head) {
    head = newNode;
  } else {
    // Find the last node in the list
    while (nodePtr->next) {
      nodePtr = nodePtr->next;
    }

    // Insert newNode as the last node.
    nodePtr->next = newNode;
  }
}
```

34.

```
void NumberList::deleteNode(double num) {
  ListNode *nodePtr, *previousNode;
  // If the list is empty, do nothing

  if (!head) {
    return;
  }

  // Determine if the first node is the one
  if (head->value == num) {
    delete head;
  } else {
    nodePtr = head;

    while (nodePtr->value != num) {
      previousNode = nodePtr;
      nodePtr = nodePtr->next;
    }

    previousNode->next = nodePtr->next;
    delete nodePtr;
  }
}
```

35.

```
NumberList::~NumberList() {
  ListNode *nodePtr, *nextNode;
  nodePtr = head;
  while (nodePtr != nullptr) {
    nextNode = nodePtr->next;
    nodePtr->next = nullptr;
    nodePtr = nextNode;
  }
}
```

4 Code Examples

4.1 Singly Linked List (Template) Implementation

4.1.1 Class Header File

Warning: I have diverged from the names that were used in class. This is because I thought the names used were stupid. If you want, you can change them back to what we were taught to use. The entire implementation of several of these member functions is moronic, but I think if we were to change those we would almost certainly get docked points.

```
#ifndef SINGLY_LINKED_LIST
#define SINGLY_LINKED_LIST
template <class T>
class SinglyLinkedList {
  private:
```

```
  struct Node {
    T data;
    Node *next;
  };

  Node *head;

  public:
  SinglyLinkedList() { head = nullptr; }
  ~SinglyLinkedList();

  void append(const T &);
  void insert(const T &);
  void remove(const T &);
  void display() const;
};
#endif // SINGLY_LINKED_LIST
```

4.1.2 Destructor

```
template <class T>
SingleLinkedList<T>::~SingleLinkedList() {
  Node *current_node;
  Node *next_node;

  // Set current_node to the start of the list
  current_node = head;

  // Moves current_node through the list and deletes every element
  // individually
  while (current_node != nullptr) {
    next_node = current_node->next;

    delete current_node;

    current_node = next_node;
  }
}
```

4.1.3 Append

```cpp
template <class T>
void SingleLinkedList<T>::append(const T &value) {
  Node *new_node;
  Node *current_node;

  // Initialize new_node with its data and populate the next pointer with
  // nullptr
  new_node = new Node;
  new_node->data = value;
  new_node->next = nullptr;

  // If the list is empty, make new_node the head
  // Else, traverse the list until the end and append the node to the end
  if (!head) {
    head = new_node;
  } else {
    // Set the traversal node to the first element (the head)
    current_node = head;

    // While the next node in the chain isn't null, set the traversal pointer to
    // be the next node
    while (current_node->next) {
      current_node = current_node->next;
    }

    // Adds new_node to the list
    current_node->next = new_node;
  }
}
```

4.1.4 Insert

```cpp
  template <class T>
  void SingleLinkedList<T>::insert(const T &value) {
    Node *new_node;
    Node *current_node;
    Node *previous_node;

    new_node = new Node;
    new_node->data = value;

    // Make new_node the head if the list is empty
    if (!head) {
      head = new_node;
      new_node->next = nullptr;
    } else {
      current_node = head;
      previous_node = nullptr;

      // Move current_node to the correct place
      while (current_node != nullptr && current_node->data < value) {
        previous_node = current_node;
        current_node = current_node->next;
      }

      // If the new node is going to be the first in the list
      if (previous_node == nullptr) {
        head = new_node;
        new_node->next = current_node;
      } else {
        previous_node->next 7= new_node;
        new_node->next = current_node;
      }
    }
  }
```

4.1.5 Remove (Delete Node)

```
template <class T>
void SingleLinkedList<T>::remove(const T &value) {
  Node *current_node;
  Node *previous_node;

  // Exit if the list is empty
  if (!head) {
    return;
  }

  // If the head is the element to be removed
  if (head->data == value) {
    current_node = head->next;
    delete head;
    head = current_node;
  } else {
    current_node = head;

    // Move current_node to the correct node
    while (current_node != nullptr && current_node->data != value) {
      previous_node = current_node;
      current_node = current_node->next;
    }

    // Change pointers around and delete current_node
    if (current_node) {
      // Removes current_node from the list by removing the reference to it
      // within the list
      previous_node->next = current_node->next;
      delete current_node;
    }
  }
}
```

4.1.6 Display

```
template <class T>
void SingleLinkedList<T>::display() const {
  Node *current_node;

  // Set current_node to the beginning of the list
  current_node = head;

  // Traverse the list
  while (current_node) {
    // Display the current element
    cout << current_node->data << '\n';

    // Set current_node to the next node in the list
    current_node = current_node->next;
  }
}
```