**Linked List Project: Code Documentation**

Brandon Michelsen

**Introduction:**

A linked list is a very important tool for a programmer. It allows for dynamic data storage of generic data types when implemented correctly, and it allows developers to implement extra functionality for manipulating and operating on the stored data. It can be used as a list data structure. For example, as an appointment list or a student list. Linked lists also serve as the building block for data structures such as a stack or a queue. A linked list also serves as a starting point for understanding other abstract data types in the realm of computer science. In this example, we will focus on a doubly-linked list.

A linked list is actually made from a collection of objects of another fundamental data structure: nodes. A node is an object that stores data and encapsulates it with the ability to link to other nodes. A node, in our case, has three data elements tied to it: a pointer to the previous node in the list, a pointer to the next node in the list, and the data that the node stores. Using this fundamental building block, we can build a linked list. By using the pointers that each node has, we can link nodes together to form a linked list. A linked list has two main nodes we need to be concerned with: a head node and a tail node. The head node indicates the beginning of the list, while the tail node indicates the end. We can use these two nodes to traverse the linked list when we want to see the data that is stored in the list.

There are also two main functions that a linked list object must have: an insert function and a delete function. The insert function is meant to insert a node into the list at a specified location. While an insert function can be implemented so that it can insert a node in the middle of the list, for the sake of simplicity, we look at an insert function that only inserts a node at the end of the list. When initially insert a new node (the first element in the list), the new node becomes the head node and the tail node. The node’s previous node pointer and the next node pointer are set to point to null. Now, let’s insert another node. This new node first is set up so that its previous node pointer points to the previous tail node. The previous tail node’s next node pointer is then set to point to the new node. Finally, the tail node is reset to be the newly inserted node. This process can go on for all new nodes inserted at the tail end of the list.

The deletion function works by first specifying a node to delete, resetting the linkages in the list around that node, and then deleting the node. There are four main cases that the deletion function needs to handle: deletion of the head node, deletion of the tail node, deletion in the middle of the list, and deletion of a node that does not exist. In the first case – deletion of the head node – the head node has to be reset. To do this, the head node is set to be the node after the previous head node (the second node in the list). Then, the head node can be deleted. In the case of the tail node, the tail node needs to be reset. To do this, the tail is set to be the node directly before the previous tail node (the second to last node). Then, the previous tail node can be deleted. Next, there is deletion of a node in the middle of the list. This case is a bit more complex than the first two cases, as both sides of the node to be deleted need to be reset. In this case, we first need to access the node before the node we are deleting and reset its next node pointer to point to the node directly after the node we are deleting. Next, we need to access the node after the node we are deleting and reset its previous node pointer to point to the node before the node we are deleting. Then, we can delete the specified node. The final case is where the node we want to delete does not exist. If this is the case, we do not want any attempt at deletion to occur. The delete function should simply return.

On top of this structure, we can add in other functionality. For example, we can add in a find function that looks for a specific value in the list and then returns the node location of that data element. In this function, the head and tail nodes are used as sentry values, and the function searches through the list until it has found the element we are searching for, or until it reaches the end of the list. There are other functions that can be added on top of this, making a linked list a fairly versatile data structure.

**Programmer’s Guide**

In this section, the programmer’s guide for both the Node Class and the Linked List Class are given.

**Node Class:**

|  |
| --- |
| **Class: Node** |
| template <class T> |
| **Member Variables:** |
| T data  Node<T>\* nextPointer  Node<T>\* prevPointer |
| **Member Functions:** |
| Node()  Node(T newData)  void setNextNode(Node<T>\* next)  Node<T>\* getNextNode()  void setPrevNode(Node<T>\* previous)  Node<T>\* getPrevNode()  void setData(T newData)  T getData() |

The Node Class is a data structure that stores a data element of type “T” (generic data type) along with pointers to adjacent nodes (both a previous node and a next node). Access to the two pointers and the stored data is accomplished through the use of getter and setter functions. The constructor can either initialize the node with data, or simply use the default of null.

**Member Variables**

|  |  |
| --- | --- |
| Variable | Description |
| Data | Data to be stored in the node (of type class T) |
| nextPointer | Pointer to the next adjacent node (of type Node<T>\*) |
| prevPointer | Pointer to the previous adjacent node (of type Node<T>\*) |

**Member Functions**

|  |  |  |  |
| --- | --- | --- | --- |
| Function | Description | Parameters | Return Type |
| (constructor) | Constructs the node object (all data is defaulted to null) | None | None |
| (constructor) | Constructs the node object with initial value for data | T newData | None |
| setNextNode | Sets the value of nextPointer | Node<T>\* next | Void |
| getNextNode | Returns the value of the nextPointer | None | Node<T>\* |
| setPrevNode | Sets the value of prevPointer | Node<T>\* previous | Void |
| getPrevNode | Returns the value of prevPointer | None | Node<T>\* |
| setData | Sets the value of data | T newData | Void |
| getData | Returns the value of data | None | T |

**Linked List Class:**

|  |
| --- |
| **Class: Linked List** |
| template<class T> |
| **Member Variables:** |
| Node<T>\* head  Node<T>\* tail |
| **Member Functions:** |
| LinkedList()  void setHeadPtr(Node<T>\* headPtr)  Node<T>\* getHeadPtr()  void setTailPtr(Node<T>\* tailPtr)  Node<T>\* getTailPtr()  void insertNode(T newData)  void deleteNode(Node<T>\* delNode)  Node<T>\* find(T dataToFind)  Node<T>\* min()  Node<T>\* max() |

The Linked List class is a data structure that stores data of type “T” (generic data type). Each element in the list is stored in a node (an object of type Node<T>, from Class: Node). The implementation of this Linked List is in the form of a doubly-linked list, meaning that each node points to the adjacent node before it and the adjacent node after it. Insertion of a new node always occurs at the tail of the list. Deletion of a node can happen anywhere in the list.

The find, min, and max functions each have linear time complexity. All other functions have constant time complexity.

**Member Variables**

|  |  |
| --- | --- |
| Variable | Description |
| Head | Points to the head (first node) in the list (of type Node<T>\*) |
| Tail | Points to the tail (last node) in the list (of type Node<T>\*) |

**Member Functions**

|  |  |  |  |
| --- | --- | --- | --- |
| Function | Description | Parameters | Return Type |
| (constructor) | Constructs the LinkedList object | None | None |
| setHeadPtr | Sets the value of head | Node<T>\* headPtr | Void |
| getHeadPtr | Returns the value of head | None | Node<T>\* |
| setTailPtr | Sets the value of tail | Node<T>\* tailPtr | Void |
| getTailPtr | Returns the value of tail | None | Node<T>\* |
| insertNode | Inserts a new node with data of type “T” at the tail of the list | T newData | Void |
| deleteNode | Deletes a node at the specified memory location | Node<T>\* delNode | Void |
| Find | Searches for the node that contains the given data, returns a pointer to that node | T dataToFind | Node<T>\* |
| Min | Finds the minimum data value in the list, returns a pointer to the node that contains it | None | Node<T>\* |
| Max | Finds the maximum data value in the list, returns a pointer to the node that contains it | None | Node<T>\* |

**Implementation Notes:**

For the find, min, and max functions, each returns a pointer to a node (Node<T>\*). It is up to the programmer who is implementing this code to properly deal with null pointer values. As attempting to access member functions of a null pointer is prohibited, the programmer must check if the value returned from the specified functions is a null pointer before using it.

**Example Usage:**

Below are some usage examples for the Linked List Class.

Note: in all use cases, know where you have stored the header file for the Linked List class.

**Getters and Setters**



**Insertion Function**



**Deletion Function**



**Find Function**



**Max Function**



**Min Function**

