Doubly Linked List

This program is for a templated doubly linked list that uses smart pointers to reduce possibilities of memory leaks. See: <https://en.cppreference.com/w/cpp/memory/shared_ptr>

It has the potential to display all values in the list, insert data to the end of the list, locate given data inside the list and delete data from the list.

To begin use make sure to #include “DoublyLinkedList.h”.

**Node Class Attributes**

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Definition** |
| data | genericType | Holds node data |
| nextNode | shared\_ptr<Node<genericType>> | Holds memory location of next Node |
| previousNode | shared\_ptr<Node<genericType>> | Holds memory location of previous Node |

**Node Class Methods**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Return type** | **Parameter(s)** | **Definition** |
| Node() +2 overloads | Constructor | -  + genericType data  + shared\_ptr nextNode, shared\_ptr previousNode | Used during the construction of object |
| ~Node() | Deconstructor | - | Used during deletion of object |
| getData() | genericType |  | Return data in Node |
| setData() | void | genericType Data | Set data in node |
| getNext() | genericType |  | Return pointer to next node |
| setNext() | Void | Shared\_ptr<Node> | Set pointer to next node |
| getPrev() | genericType |  | Return pointer to previous node |
| setPrev() | void | Shared\_ptr<Node> | Set pointer to previous node |

**Doubly Linked List Class Attributes**

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Definition** |
| head | shared\_ptr<Node<genericType>> | Holds memory location of head sentinel node |
| tail | shared\_ptr<Node<genericType>> | Holds memory location of tail sentinel node |

**Doubly Linked List Class Methods**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Return type** | **Parameter(s)** | **Definition** |
| DoublyLinkedList() | Constructor | - | Used in construction of the list. This creates a head and tail sentinel node and sets pointers to each other as well as pointers to null and sets data to null. |
| Find() | Shared\_ptr  <Node<genericType>> | genericType dataToFind | Find a node that contains the data we send as a parameter and return its pointer. If none can be found return a nullptr. |
| insertNode() | Void | genericType dataToFind | Dynamically create an object and set its data to the data passed in our parameter. Next add this to our list by assigning pointers to it and assigning its pointers to the respective surrounding nodes |
| deleteNode | Void | genericType dataToFind | Search for the data we are trying using the find function. Use the pointer to get access to the node we are trying to delete. Assign the next and previous nodes to each other respectively. Reset the shared\_ptr to clear ownership and delete the object.  NOTE: This will not allow you to access nullptr or delete the head/tail sentinel nodes. |
| displayList() | Void | - | Output to console the data of every element in our linked list. |

|  |
| --- |
| DoublyLinkedList<Int> MyLinkedList;  MyLinkedList.insertNode(5);  MyLinkedList.insertNode(4);  MyLinkedList.displayList();  // Outputs 54, ignoring head and tail sentinel nodes  Cout << \*MyLinkedList.find(5);  // Returns pointer to the first node that holds 5 in its data.  MyLinkedList.deleteNode(5);  // Deletes node from list that holds 5 in its data  MyLinkedList.displayList();  // outputs 4, ignoring head and tail sentinel nodes |

Output:

|  |
| --- |
| 5 // from displayList  4 // from displayList  5 // from cout << \*MyLinkList.find(5)  4 // from displayList after removing node that contains 5 |