The purpose of this assignment is to practice pointers. To this end, we will study a data structure, *Linked Lists*. When studying arrays, we found some limitations on arrays:

 Arrays have fixed size. Once an array is created, the size cannot be changed.

 Assuming an array has to be sorted, inserting a new element required shifting, in the worst case, n elements, where n is the number of elements in the array, i.e.:

**The figure below shows an array. A request is received to insert 1:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 5 | 10 | 15 | 18 | 20 |  |

**Since array needs to be sorted, all elements must be shifted to make room for the new element:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 5 | 10 | 15 | 18 | 20 |

**Once the first spot is cleared, then 1 can be inserted:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | 5 | 10 | 15 | 18 | 20 |

 Removing an element from an array having n elements requires, in the worst case, shifting n-1 elements assuming the array is sorted.

A *Linked List* overcomes these limitations. Conceptually, a Linked List is a collection of nodes where each nodes points to the next node in the list. The following figure depicts a linked list:



In order to insert a new element to the linked list, it is not necessary shifting the elements. It suffices to rearrange a few pointers. For instance, suppose that we need to insert 12 into the above linked list:



To insert 12 into the list, we need have to change the pointer of 10 to make it point to 12 and make 12 point to 15:



The shape is irrelevant. The above linked list represents the same linked list as:



Therefore, inserting an element into a linked list requires changing two pointers provided it is known the preceding node.

From the above figures, we can identify the classes needed to implement a linked list. A class *Node* is needed. A node contains the value and a pointer to the next node. In order to make this class reusable, it will be implemented as a template:

#pragma once

template<class T>

class Node

{

public:

 // Creates a new instance of class Node.

 // parameter nodeValue: The value of the node.

 Node(T nodeValue): value(nodeValue), next(NULL) { }

 // Creates a new instance of class Node.

 // parameter nodeValue: The value of the node.

 // parameter nextNode: The node this instance points to.

 Node(T nodeValue, Node\* nextNode): value(nodeValue), next(nextNode) { }

 // Disposes the instance.

 ~Node(void) { }

 // The node referenced by this instance.

 Node\* next;

 // The value stored in this instance.

 T value;

};

The constructors show an alternate syntax to initialize data members. For instance, the second constructor is equivalent to:

 Node(T nodeValue, Node\* nextNode)

 {

 value = nodeValue;

 next = nextNode;

 }

The following source code shows the interface and the implementation of one of the methods of class LinkedList:

#pragma once

#include "Node.h"

template<class T>

class LinkedList

{

public:

 // Creates a new instance of class LinkedList.

 LinkedList(void): \_front(NULL), \_back(NULL), \_size(0) {};

 // Disposes this instance.

 ~LinkedList(void) { }

 // Adds the specified element to the front of the linked list.

 // parameter T: The value of the node to be added.

 void addToFront(T);

 // Adds the specified element to the back of the linked list.

 // parameter T: The value of the node to be added.

 void addToBack(T);

 // Adds the specified element after the specified node.

 // parameter T: The value of the node to be added.

 // parameter Node<T>\*: The node that will reference the new node

 // created in this function.

 void add(T, Node<T>\*);

 // Finds the first node containing the specified value.

 // parameter T: The value to find in the list.

 Node<T>\* find(T);

 // Removes the first node containing the specified list.

 void remove(T);

 // Returns the size of the linked list.

 int getSize();

 // Displays to the screen the all the elements of the list.

 void displayList();

private:

 // The first of the list

 Node<T>\* \_front;

 // The last element of the list

 Node<T>\* \_back;

 // The size of the list.

 int \_size;

};

template<class T>

void LinkedList<T>::addToFront(T value)

{

 Node<T>\* newFront = new Node<T>(value, \_front);

 // Adding the first element

 if(\_back == NULL)

 {

 \_back = newFront;

 }

 \_size++;

 \_front = newFront;

}

// .... Rest of function members

The shown function, addToFront, is performing the following logic:

Initially there is a linked list:



The next picture is the new state of our linked list after executing the line

Node<T>\* newFront = new Node<T>(value, \_front);



And finally, after executing

\_front = newFront;



Implement the rest of the functions in class LinkedList. The following main function displays some test cases and the expected output. Note the arrow syntax used to access members from an object. This syntax is used when objects are created using the *new* keyword.

int main()

{

 LinkedList<int>\* list = new LinkedList<int>;

 // Size: 0, list: x

 displayInfo(list);

 // Size: 1, list: 10 -> x

 list->addToFront(10);

 displayInfo(list);

 // Size: 2, list: 10 -> 20 -> x

 list->addToBack(20);

 displayInfo(list);

 // Size: 3, list: 10 -> 20 -> 30 -> x

 list->addToBack(30);

 displayInfo(list);

 // Size: 4, list: 10 -> 20 -> 25 -> 30 -> x

 Node<int>\* nodeWith20 = list->find(20);

 list->add(25, nodeWith20);

 displayInfo(list);

 // Size: 5, list: 10 -> 15 -> 20 -> 25 -> 30 -> x

 Node<int>\* nodeWith10 = list->find(10);

 list->add(15, nodeWith10);

 displayInfo(list);

 // Size: 4, list: 10 -> 15 -> 25 -> 30 -> x

 list->remove(20);

 displayInfo(list);

 // Size: 3, list: 10 -> 15 -> 25 -> x

 list->remove(30);

 displayInfo(list);

 system("pause");

 return 0;

}

In your assignment, make sure to include a few more test cases showing the correctness of your linked list.