Name: Saransh Kalra

R Number: R11528524

LAB section: 502

**DoublyLinkedList.hpp**

#ifndef DoublyLinkedList\_hpp

#define DoublyLinkedList\_hpp

#include <stdio.h>

*//creating the node class for the node of the linked list*

**class** Node

{

**private**:

*//variable info for storing node info*

**int** info;

**public**:

*//pointer of class node, pointing to the next node of the linked list*

Node\* next;

*//pointer of class node, pointing to the prev node of the linked list*

Node\* prev;

*//accessor for the info stored in node*

**int** getInfo();

*//mutator for the info stored in node*

**void** setInfo(**int** value);

};

*//class for linked list*

**class** DoublyLinkedList

{

**private**:

*//pointers to point of the head and tail of the list*

Node \*head, \*tail;

*//to keep the track of the size of the DoublyLinkedList*

**int** size;

**public**:

DoublyLinkedList(); *//constructor to initialize head and tail to 0 or NULL*

~DoublyLinkedList(); *//deconstructor to free the memory borrowed from heap*

*//member functions to insert a node at the head, the tail and at a particular position of the linked list*

**void** insertAtHead(**int** value);

**void** insertAtTail(**int** value);

**void** insertAtPos(**int** pos, **int** value);

*//Print the info value before deletion of the node and then deleting the node from the head, the tail and at a particular position of the linked list*

**void** deleteAtHead();

**void** deleteAtTail();

**void** deleteAtPos(**int** pos);

*//search the given info found at which node of a linked list*

**void** search(**int** value);

*//member function to display the list*

**void** displayList();

};

#endif */\* DoublyLinkedList\_hpp \*/*

**DoublyLinkedList.cpp**

#include "DoublyLinkedList.hpp"

#include <stdio.h>

#include <iostream>

**using** **namespace** std;

*//mutator for setting the info of the node*

**void** Node::setInfo(**int** value){

**this**->info=value;

}

*//accessor for getting the info from the node*

**int** Node::getInfo(){

**return** info;

}

*//Initialising the head and tail pointers to NULL*

DoublyLinkedList::DoublyLinkedList(){

**this**->head=**this**->tail=**NULL**;

**this**->size=0;

}

*//deconstructor to free the memory occupied by linked list*

DoublyLinkedList::~DoublyLinkedList(){

*//while the head doesn't reach till node, traverse through the list and free the memory occupied by each node*

**while**(**this**->head!=**NULL**)

{

Node \*ptr;

ptr = **this**->head;

**this**->head = **this**->head->next;

**delete** ptr;

}

}

*//member function to insert a node at the head of the linked list*

**void** DoublyLinkedList::insertAtHead(**int** value){

*//if head is NULL, means the list is empty, put the info in the first node and make the tail as head too as single node is both the head and tail*

**if**(**this**->head==**NULL**){

*//initialise a pointer to store the new info to be added*

Node \*ptr = **new** Node();

*//the new node's previous is set to NULL*

ptr->prev=**NULL**;

ptr->setInfo(value);

*//the new pointer's next node is the old head*

ptr->next=**this**->head;

*//the updated head points to the new pointer now*

**this**->head=ptr;

*//making the tail to be the head*

**this**->tail=**this**->head;

*//since there is just one record so previous is NULL*

**this**->tail->prev = **this**->head->prev = **NULL**;

*//increase size by 1, when one more pointer is added*

**this**->size++;

}

**else**{

*//same as above just dont make the head the tail this time as this is not the first node*

Node \*ptr = **new** Node();

ptr->setInfo(value);

ptr->prev=**NULL**;

ptr->next=**this**->head;

*//the new added ptr take the place of prev of old head*

**this**->head->prev = ptr;

**this**->head=ptr;

**this**->size++;

}

}

*//member function to insert a node at the tail of the linked list*

**void** DoublyLinkedList::insertAtTail(**int** value){

*//if tail is NULL, means the list is empty, put the info in the first node and make the head as tail too as single node is both the head and tail*

**if**(**this**->tail==**NULL**){

*//initialise a pointer to store the new info to be added*

Node \*ptr = **new** Node();

ptr->setInfo(value);

*//the old tail's pointer to next node is the new pointer*

**this**->tail->next=ptr;

*//the new tail's prev pointer points to the old tail*

ptr->prev=**this**->tail;

*//the updated tail points to the new pointer now*

**this**->tail=ptr;

*//making the head to be the tail*

**this**->head=**this**->tail;

*//increase size by 1, when one more pointer is added*

**this**->size++;

}

**else**{

*//same as above just dont make the tail the head this time as this is not the first node*

Node \*ptr = **new** Node();

ptr->setInfo(value);

ptr->next=**NULL**;

**this**->tail->next=ptr;

ptr->prev=**this**->tail;

**this**->tail=ptr;

**this**->size++;

}

}

**void** DoublyLinkedList::insertAtPos(**int** pos, **int** value){

*//out of bounds test*

**if**(pos > **this**->size+1||pos<0){

cout<<"index out of bounds"<<endl;

}

*//set a value zero to indicate add at head*

**else** **if**(pos==0){

DoublyLinkedList::insertAtHead(value);

}

*//set a value size+1 to indicate add at head*

**else** **if**(pos==**this**->size+1){

DoublyLinkedList::insertAtTail(value);

}

**else**{

Node \*ptr = **new** Node();

ptr->setInfo(value);

Node \*ptr1 = **new** Node();

ptr1 = **this**->head;

**int** count=1;

*//found the position of the record where it is to be added*

**while**(count!=pos){

ptr1=ptr1->next;

count++;

}

*//if it says add at pos 1, add after head*

**if**(ptr1==**this**->head){

ptr->next=ptr1->next;

ptr1->next->prev=ptr;

ptr->prev=ptr1;

ptr1->next=ptr;

**this**->size++;

cout<<value<<" added to the list"<<endl;

}

*//if it says add at pos (size of list), add before tail*

**else** **if**(ptr1==**this**->tail){

ptr->prev=ptr1->prev;

ptr1->prev->next=ptr;

ptr->next=ptr1;

ptr1->prev=ptr;

**this**->size++;

cout<<value<<" added to the list"<<endl;

}

**else**{

ptr->next=ptr1;

ptr->prev=ptr1->prev;

ptr1->prev->next=ptr;

ptr1->prev=ptr;

**this**->size++;

cout<<value<<" added to the list"<<endl;

}

}

}

*//Print the info value before deletion of the node and then deleting the node at the head of the linked list*

**void** DoublyLinkedList::deleteAtHead(){

*//if the head points to NULL, means the list is empty, therefore error message printed*

**if**(**this**->head==**NULL**){

cout<<"there is nothing to be deleted, the list is empty"<<endl;

}

*//if the head's next points to NULL, means there is just one record so print deleted info and make both the head and tail to be NULL, making the list to be empty*

**else** **if**(**this**->head->next==**NULL**){

cout<<"The value being deleted is: "<<**this**->head->getInfo()<<endl;

**this**->head=**this**->tail=**NULL**;

**this**->size--;

}

*//else just make the head to be its next and print the old head info before that*

**else**{

cout<<"The value being deleted is: "<<**this**->head->getInfo()<<endl;

**this**->head=**this**->head->next;

**this**->head->prev=**NULL**;

**this**->size--;

}

}

*//Print the info value before deletion of the node and then deleting the node at the tail of the linked list*

**void** DoublyLinkedList::deleteAtTail(){

*//if the head points to NULL, means the list is empty, therefore error message printed*

**if**(**this**->head==**NULL**){

cout<<"there is nothing to be deleted, the list is empty"<<endl;

}

*//if the head's next points to NULL, means there is just one record so print deleted info and make both the head and tail to be NULL, making the list to be empty*

**else** **if**(**this**->head->next==**NULL**){

cout<<"The value being deleted is: "<<**this**->tail->getInfo()<<endl;

**this**->head=**this**->tail=**NULL**;

**this**->size--;

}

*//else just make the tail's next to be NULL, and the old tail to be the previous record of tail*

**else**{

cout<<"The value being deleted is: "<<**this**->tail->getInfo()<<endl;

**this**->tail=**this**->tail->prev;

**this**->tail->next=**NULL**;

**this**->size--;

}

}

**void** DoublyLinkedList::deleteAtPos(**int** pos){

*//out of bounds test*

**if**(pos<0||pos>**this**->size+1){

cout<<"index out of bounds"<<endl;

}

*//if the head points to NULL, means the list is empty, therefore error message printed*

**else** **if**(**this**->head==**NULL**){

cout<<"there is nothing to be deleted, the list is empty"<<endl;

}

**else**{

*//if pos is 0 or 1 the record will be deleted at head*

**if**(pos==0||pos==1){

deleteAtHead();

}

*//if pos is size or size+1 the record will be deleted at tail*

**else** **if**(pos==**this**->size||pos==**this**->size+1){

deleteAtTail();

}

**else**{

Node \*ptr1 = **new** Node();

ptr1 = **this**->head;

**int** count=1;

*//finding the record to be deleted*

**while**(count!=pos){

ptr1=ptr1->next;

count++;

}

*//display info to be deleted*

cout<<"the info being deleted is : "<<ptr1->getInfo()<<endl;

ptr1->next->prev=ptr1->prev;

ptr1->prev->next=ptr1->next;

**this**->size--;

}

}

}

*//member function to search the given info found at which node of a linked list*

**void** DoublyLinkedList::search(**int** value){

*//counter variable to count at which position given info was found*

**int** count=0;

*//flag variable to indicate whether value was found or not*

**int** flag=0;

*//if head points to NULL, means the list is empty*

**if**(**this**->head==**NULL**)

cout<<"The whole list is empty, so the record was not found"<<endl;

*//else traverse the whole linked list to find if some node's info matches with the given info*

**else**{

Node \*ptr = **this**->head;

**while**(ptr->getInfo()!=value && ptr!=**this**->tail){

ptr=ptr->next;

count++;

**if**(ptr->getInfo()==value){

flag=1;

}

}

*//error message if info was not found*

**if**(flag==0){

cout<<"The given value "<<value<<" was not found in the list!"<<endl;

}

**else**{

cout<<"The value "<<value<<" was found and its the no. "<<count+1<<" record."<<endl;

}

}

}

*//member function to display the linked list*

**void** DoublyLinkedList::displayList(){

*//if head points to NULL, the list is empty*

**if**(**this**->head==**NULL**){

cout<<"The whole list is empty, nothing to display."<<endl;

}

*//else traverse through the list displaying each node's info followed by an arrow*

**else**{

Node \*ptr;

ptr = **this**->head;

**while**(ptr!=**NULL**){

cout<<ptr->getInfo()<<"->";

ptr = ptr->next;

}

cout<<endl<<"The size of the list is: "<<**this**->size<<endl;

*/\**

*//to check the working of the prev pointer*

*cout<<"reverse order"<<endl;*

*ptr = this->tail;*

*while(ptr!=NULL){*

*cout<<ptr->getInfo()<<"->";*

*ptr = ptr->prev;*

*}*

*cout<<endl;*

*\*/*

}

}

**Main.cpp**

#include <iostream>

#include "DoublyLinkedList.hpp"

**using** **namespace** std;

**int** main() {

*//initialising instance variable of linked list*

DoublyLinkedList LinkedList1;

*//call to member function to delete at tail*

LinkedList1.deleteAtTail();

*//call to member function to delete at head*

LinkedList1.deleteAtHead();

*//call to insert 1 at head*

LinkedList1.insertAtHead(1);

*//display the list function called*

LinkedList1.displayList();

*//call to insert 2 at tail*

LinkedList1.insertAtTail(2);

*//display the list function called*

LinkedList1.displayList();

*//call to insert 3 at tail*

LinkedList1.insertAtTail(3);

*//display the list function called*

LinkedList1.displayList();

*//call to insert 4 at tail*

LinkedList1.insertAtTail(4);

*//display the list function called*

LinkedList1.displayList();

*//search 2 in the linked list*

LinkedList1.search(2);

*//search 4 in the linked list*

LinkedList1.search(4);

*//delete the value at tail*

LinkedList1.deleteAtTail();

*//search 4 in the linked list*

LinkedList1.search(4);

*//delete the value at head*

LinkedList1.deleteAtHead();

*//display the list*

LinkedList1.displayList();

*//testing the insert function*

LinkedList1.insertAtPos(1, 6);

*//display the list*

LinkedList1.displayList();

*//testing the insert function*

LinkedList1.insertAtPos(3, 5);

*//display the list*

LinkedList1.displayList();

*//testing the delete function*

LinkedList1.deleteAtPos(2);

*//display the list*

LinkedList1.displayList();

*//testing the delete function*

LinkedList1.deleteAtPos(3);

*//display the list*

LinkedList1.displayList();

**return** 0;

}

**OUTPUT**

****