#ifndef PERSON\_H\_

#define PERSON\_H\_

#include <iostream>

#include <string>

namespace std{

class Person {

private:

int id;

string name, surname;

public:

Person(int a=0, string n="", string sn="" ){

id = a;

name = n;

surname = sn;

}

bool operator==(const Person &other){

return (id == other.id);

}

friend ostream& operator<<(ostream &out, const Person &p){

out << "(" << p.name << " " << p.surname <<","<< p.id <<")";

return out;

}

int getId() const {

return id;

}

void setId(int age) {

this->id = age;

}

const string& getName() const {

return name;

}

void setName(const string &name) {

this->name = name;

}

const string& getSurname() const {

return surname;

}

void setSurname(const string &surname) {

this->surname = surname;

}

};

}

#endif /\* PERSON\_H\_ \*/

------------------------------------

#ifndef LINKEDLIST\_H\_

#define LINKEDLIST\_H\_

#include <iostream>

#include <cassert>

namespace std {

template<typename T>

struct LLNode{

T data;

LLNode<T>\* next;

};

template<typename T>

class LinkedList {

protected:

LLNode<T>\* head;

LLNode<T>\* tail;

int count;

public:

LinkedList();

virtual ~LinkedList();

LinkedList(const LinkedList &other);

LinkedList& operator=(const LinkedList &other);

bool isEmpty() const;

int length() const;

LLNode<T>\* search(const T& val);

void insertFirst(const T &val);

void insertLast(const T &val);

void insertAt(const int idx, const T &val);

void insertAfter(const T &sVal, const T &val);

void insertBefore(const T &sVal, const T &val);

void deleteNode(const T &val);

void deleteFirst();

void clearList();

T front() const;

T back() const;

template <typename S>

friend ostream& operator<<(ostream& out, const LinkedList<S> &list);

void reversePrint();

protected:

bool searchAndFindPrev(const T &val, LLNode<T>\*\* sNode, LLNode<T>\*\* pNode);

void recursiveReversePrint(LLNode<T> \*p);

};

template <typename T>

LinkedList<T>::LinkedList() {

head = tail = NULL;

count = 0;

}

template <typename T>

LinkedList<T>::~LinkedList() {

clearList();

}

template <typename T>

LinkedList<T>::LinkedList(const LinkedList &other) {

head = tail = NULL;

LLNode<T> \*p = other.head;

while (p != NULL){

insertLast(p->data);

p = p->next;

}

count = other.count;

}

template <typename T>

LinkedList<T>& LinkedList<T>::operator=(const LinkedList &other) {

clearList();

if (other.count == 0)

return \*this;

LLNode<T> \*p = other.head;

while(p != NULL){

insertLast(p->data);

p = p->next;

}

return \*this;

}

template <typename T>

bool LinkedList<T>::isEmpty() const{

return (count == 0);

}

template <typename T>

int LinkedList<T>::length() const{

return count;

}

template <typename T>

LLNode<T>\* LinkedList<T>::search(const T &val){

bool found = false;

LLNode<T> \*p = head;

while ((p != NULL) && !found){

if (p->data == val)

found = true;

else

p = p->next;

}

return p;

}

template <typename T>

void LinkedList<T>::insertFirst(const T &val){

LLNode<T> \*p = new LLNode<T>;

p->data= val;

p->next = head;

head = p;

if (tail == NULL)

tail = p;

count++;

}

template <typename T>

void LinkedList<T>::insertLast(const T &val){

LLNode<T> \*p = new LLNode<T>;

p->data = val;

p->next = NULL;

if (head != NULL){

tail->next = p;

tail = p;

}

else{

head = tail = p;

}

count++;

}

template <typename T>

void LinkedList<T>::insertAt(const int idx, const T &val){

if (idx > count){

// cout << "Given index is greater than the number of element, so appending to list"<<endl;

insertLast(val);

}

else if (idx <= 1){

// cout << "Given index value is less than or equal to 1, so inserting as the first element"<<endl;

insertFirst(val);

}

else{ // insert at given position

LLNode<T> \*prevNode = head;

int cnt=2;

while (cnt < idx){

prevNode = prevNode->next;

} // we found (idx-1)'th node

LLNode<T> \*newNode = new LLNode<T>;

newNode->data = val;

newNode->next = prevNode->next;

prevNode->next = newNode;

}

count++;

}

template <typename T>

void LinkedList<T>::insertAfter(const T &sVal, const T &val){

LLNode<T> \*searchItem = search(sVal); // search for sVal in the list

if (searchItem == NULL){

cout << "The value "<<sVal<<" is not found in the list, so nothing will be inserted."<<endl;

}

else{ // we found searched item

LLNode<T> \*newNode = new LLNode<T>;

newNode->data = val;

newNode->next = searchItem->next;

searchItem->next = newNode;

count++;

}

}

template <typename T>

void LinkedList<T>::insertBefore(const T &searchValue, const T &val){

LLNode<T> \*searchedNode=NULL;

LLNode<T> \*prevNode=NULL;

// search for searchValue in the list, assign searchedNode and prevNode if found

bool found = searchAndFindPrev(searchValue, &searchedNode, &prevNode);

if (!found){

cout << "The value "<<searchValue<<" is not found in the list, so nothing will be inserted."<<endl;

}

else{ // we found searched item

LLNode<T> \*newNode = new LLNode<T>;

newNode->data = val;

if (searchedNode == head) { // insert as first element

newNode->next = head;

head = newNode;

} else {

//Node<T> \*prevNode = getPrevNodeOf(searchItem);

newNode->next = prevNode->next;

prevNode->next = newNode;

}

count++;

}

}

template <typename T>

void LinkedList<T>::deleteNode(const T &val){

LLNode<T> \*delNode=NULL;

LLNode<T> \*prevNode=NULL;

bool found = searchAndFindPrev(val, &delNode, &prevNode); // search for val in the list

if (!found){ // not found in the list

cout << "Item "<< val << " is not in the list, nothing to delete!"<<endl;

}

else{ // found in the list and delItem points to that node

assert(delNode != NULL);

// depending on the position, delete appropriately

if (delNode == head){ // delete first node

head = delNode->next; // or head = head->next; both are same

}

else {

assert (prevNode != NULL);

if (delNode == tail) { // if last item is to be deleted

prevNode->next = NULL;

tail = prevNode;

} else {

prevNode->next = delNode->next;

}

}

delete delNode;

count--;

}

}

template <typename T>

void LinkedList<T>::deleteFirst(){

if (!isEmpty()){

LLNode<T> \*p = head;

head = head->next;

delete p;

count--;

}

}

template <typename T>

void LinkedList<T>::clearList() {

LLNode<T> \*p;

while (head != NULL){

p = head;

head = head->next;

delete p;

}

tail = NULL;

count = 0;

}

template <typename T>

T LinkedList<T>::front() const{

assert(head != NULL);

return head->data;

}

template <typename T>

T LinkedList<T>::back() const{

assert(tail != NULL);

return tail->data;

}

template <typename T>

ostream& operator<<(ostream& out, const LinkedList<T> &list){

LLNode<T> \*p = list.head;

while(p != NULL){

cout << p->data << " ";

p = p->next;

}

return out;

}

template <typename T>

bool LinkedList<T>::searchAndFindPrev(const T &val, LLNode<T>\*\* sNode, LLNode<T>\*\* pNode){

// search for val in the list, if found sNode points to searched node, and

// pNode points to previous of that node

// return true if found, else return false

bool found = false;

\*sNode = head; // searched Node

\*pNode = NULL; // previous node

while ((\*sNode != NULL) && !found){

if ((\*sNode)->data == val){

found = true;

}

else{

\*pNode = \*sNode;

\*sNode = (\*sNode)->next;

}

}

return found;

}

template<class T>

void LinkedList<T>::reversePrint() {

recursiveReversePrint(head);

}

template<class T>

void LinkedList<T>::recursiveReversePrint(LLNode<T> \*p) {

if (p != NULL) {

recursiveReversePrint(p->next); //recursive call

cout << p->data << " ";

}

}

} /\* namespace std \*/

#endif /\* LINKEDLIST\_H\_ \*/

------------------------------------------------------------------------------------------

#include <iostream>

#include "Person.h"

#include "LinkedList.h"

using namespace std;

//

//int main() {

// LinkedList<Person> myList;

//

// Person p(28, "Ziya", "Karakaya");

//

// myList.insertLast(p);

// cout << myList << endl;

//

// myList.insertLast(Person(30, "Ä°brahim", "Cereci"));

// cout << "First list: " << myList << endl;

//

// LinkedList<Person> secondList = myList;

//

// cout << "Second List : "<< secondList << endl;

//

// myList.deleteNode(Person(21, "Ziya","Karakaya"));

// cout << "First list: " << myList << endl;

//

// myList.insertAfter(Person(30), Person(49, "Ali","Veli"));

// cout << "First list: " << myList << endl;

//

// LinkedList<Person> thirdList;

// thirdList = myList;

// cout << "Third list: " << thirdList << endl;

//

// myList.insertBefore(Person(49), Person(17, "Ali", "Deli"));

// cout << "First list: " << myList << endl;

//

// myList.insertFirst(Person(51, "Erdem", "Korkmaz"));

// cout << "First list: " << myList << endl;

//

// myList.insertBefore(p, Person(50, "Veli","Deli"));

// cout << "First list: " << myList << endl;

//

// myList.insertBefore(Person(51), Person(15, "Simay","Alkan"));

// cout << "First list: " << myList << endl;

//

// myList.insertAt(2, Person(70, "Ozalp", "Tozan"));

// cout << "First list: " << myList << endl;

//

// myList.deleteNode(p);

// cout << "First list: " << myList << endl;

//

// myList.deleteNode(Person(70));

// cout << "First list: " << myList << endl;

//

// myList.deleteNode(Person(49));

// cout << "First list: " << myList << endl;

//

// myList.deleteNode(Person(50));

// cout << "First list: " << myList << endl;

// cout << "First list in reverse: ";

// myList.reversePrint();

// cout << endl;

//

// cout << "Second List : "<< secondList << endl;

// cout << "Second List in Reverse: ";

// secondList.reversePrint();

//

// return 0;

//}