# Department of Computing

# School of Electrical Engineering and Computer Science

**CS-250: Data Structure and Algorithms**

**Class: BSCS 11AC**

# 

# Lab 03-Part(B) : Doubly Linked Lists

**Date: 09th September, 2022**

**Time: 10:00 am – 12:50 pm   
&  
 02:00 pm – 4:50 pm**

# Instructor: Dr. Yasir Faheem

# Lab Engineer: Hadaiq Ahmad

# 

## Linked List.h

//  
// Created by hasib on 9/27/22.  
//  
  
#ifndef DOUBLE\_LINKED\_LIST\_LINKED\_LIST\_H  
#define DOUBLE\_LINKED\_LIST\_LINKED\_LIST\_H  
  
#include "node.h"  
class linked\_list {  
public:  
 node \*start;  
 node \*last;  
 node \*loc;  
 node \*ploc;  
linked\_list();  
~linked\_list();  
  
  
void PrintReverse();  
void GroupEvenOdd();  
void SwapWhere(int,int);  
void ReverseDoubleLinkedList();  
void SwapNodes();  
bool isEmpty();  
void printlist();  
void Search(int);  
bool isPresent(int);  
void InsertSorted(int);  
void InsertAtFront(int);  
void DeleteFront();  
void DeleteLast();  
void DeleteValue(int);  
void DestroyList();  
void InsertAtLast(int);  
  
  
};  
  
  
#endif //DOUBLE\_LINKED\_LIST\_LINKED\_LIST\_H

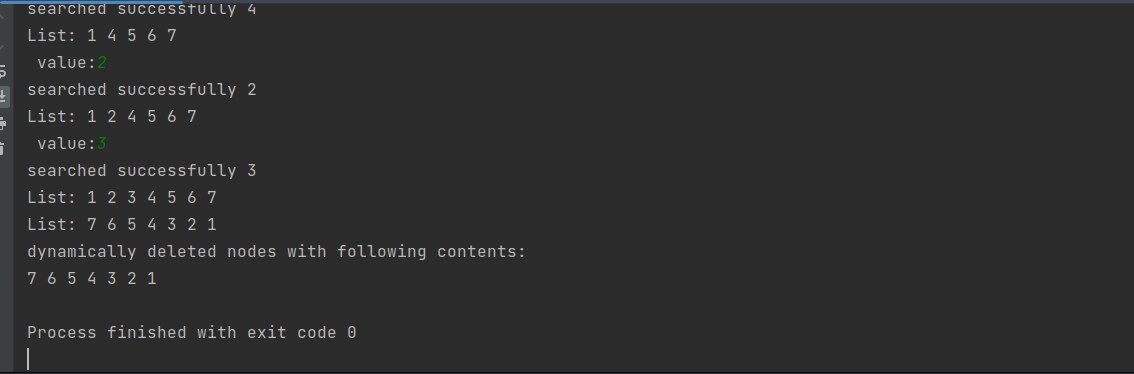
## Linked List.cpp

//  
// Created by hasib on 9/27/22.  
//  
  
#include "linked\_list.h"  
#include "node.h"  
#include <iostream>  
linked\_list ::linked\_list() {  
 start = nullptr;  
 last = nullptr;  
 loc = nullptr;  
 ploc = nullptr;  
  
}  
  
  
  
void linked\_list ::Search(int value) {  
 if (!isEmpty()) {  
 loc = start;  
 ploc = nullptr;  
 while (loc != nullptr && loc->data < value) {  
 ploc = loc;  
 loc = loc->next;  
 }  
  
 if(ploc!=last)  
 {  
 if (loc->data != value)  
 {  
 loc = nullptr;  
 }  
 }  
  
  
  
 }  
 std::cout<<"searched successfully "<<value<<std::endl;  
}  
void linked\_list ::InsertSorted(int value) {  
  
 if (!isEmpty()) {  
 Search(value);  
 if (loc == nullptr) {  
 if (ploc != nullptr) {  
 if (ploc == last) {  
 InsertAtLast(value);  
 }  
 else {  
 node\* nn = new node();  
 nn->data=value;  
 nn->next = ploc->next;  
 nn->previous = ploc;  
 ploc->next->previous= nn;  
 ploc->next = nn;  
  
 }  
 }  
 else {  
 InsertAtFront(value);  
 }  
 }  
 }  
 else {  
 InsertAtFront(value);  
 }  
}  
  
  
bool linked\_list ::isPresent(int value) {  
 Search(value);  
  
 if (loc!= nullptr){  
 if (loc->data == value) {  
 return true;  
 }  
 else{  
 return false;  
 }  
  
 }  
 else{  
 return false;  
 }  
}  
bool linked\_list :: isEmpty(){  
 return start == nullptr;  
}  
void linked\_list ::printlist() {  
  
 node \*temp = start;  
 std::cout<<"List: ";  
 while(temp!= nullptr){  
 std::cout<<temp->data<<" ";  
 temp = temp->next;  
 }  
 std::cout<<std::endl;  
}  
  
void linked\_list::InsertAtFront(int value) {  
 if (!isPresent(value)) {  
 node \*new\_node = new node();  
 new\_node->data = value;  
 new\_node->next = start;  
 new\_node->previous = nullptr;  
  
 if (isEmpty()) {  
 last = new\_node;  
 }  
 else{  
 start->previous = new\_node;  
  
 }  
 start = new\_node;  
 }  
}  
  
  
void linked\_list ::InsertAtLast(int value) {  
 if(!isPresent(value)) {  
 node \*new\_node = new node();  
 new\_node->data = value;  
 new\_node->next = nullptr;  
 new\_node->previous = last;  
  
  
 if (isEmpty()) {  
 start = new\_node;  
 } else {  
 last->next = new\_node;  
 }  
 last = new\_node;  
 }  
}  
  
void linked\_list ::DeleteFront() {  
 if(!isEmpty())  
 {  
 node \*temp = start;  
 if (start->next == nullptr)  
 {  
 start = nullptr;  
 last = nullptr;  
 }  
 else  
 {  
 start = start->next; // moving the start node to second position;  
 start->previous = nullptr;// removing value of to be deleted first node in modified start node  
 }  
 delete temp;  
 }  
 else{  
 std::cout<<"Error:the list is empty"<<std::endl;  
  
 }  
}  
  
void linked\_list ::DeleteLast() {  
 if(!isEmpty())  
 {  
 node \*temp = last;  
 if (last->previous == nullptr)  
 {  
 start = nullptr;  
 last = nullptr;  
 }  
 else  
 {  
 last = last->previous; // moving the last node to second last position;  
 last->next = nullptr;// removing value of to be deleted last node in modified last node  
 }  
 delete temp;  
 }  
 else{  
 std::cout<<"Error:the list is empty"<<std::endl;  
  
 }  
  
}  
void linked\_list ::DeleteValue(int value) {  
 if(isPresent(value)) // Search has already been called in isPresent Function  
 {  
 if (loc == start){  
 DeleteFront();  
 }  
 else if (loc == last){  
 DeleteLast();  
 }  
 else{  
 node \*temp = loc;  
  
 //targeted node's next is pointing to temp->next  
 ploc->next = temp->next;  
 //successor of the targeted node's(temp->next) previous is pointing to temp  
 temp->next->previous = ploc;  
 }  
 }  
 else{  
 std::cout<<"Error: "<<value<<" is not present in the list"<<std::endl;  
 }  
  
  
  
}  
  
void linked\_list::ReverseDoubleLinkedList(){  
 loc = start; // adress of current node  
 // ploc as a swapper  
 while (loc!= nullptr){  
 ploc = loc->next;  
 loc->next = loc->previous;  
 loc->previous = ploc;  
  
 loc = loc->previous;  
 }  
  
 loc = start;  
 start = last;  
 last = loc;  
  
  
}  
  
  
  
void linked\_list ::SwapWhere(int x, int y)  
{  
 //x should be the start  
  
 if(x>y){  
 int temp = x;  
 x = y;  
 y = temp;  
 }  
  
 Search(y);  
 node \* loc\_2 = loc;  
 node \* ploc\_2= ploc;  
 loc = nullptr;  
 ploc= nullptr;  
 Search(x);  
  
 if (loc!= nullptr && loc\_2!=nullptr) {  
  
 //if we need to swap both the first and last node  
 if ((loc == start && loc\_2 == last) || (loc\_2 == start && loc == last)) {  
  
 if (start->next == last) // if we have only two nodes  
 {  
 last->previous = nullptr;  
 last->next = start;  
 start->previous = last;  
 start->next = nullptr;  
 start = last;  
 last = loc;  
 }  
 // if there are other nodes as well in between first and last  
 else {  
  
  
 ploc\_2 = last->previous;  
 last->previous = nullptr;  
 last->next = start->next;  
 start->next->previous = last;  
 ploc\_2->next = start;  
 start->previous = ploc\_2;  
 start->next = nullptr;  
  
 loc = start;  
 start = last;  
 last = loc;  
  
  
 }  
  
  
 }  
  
  
  
  
 else if (loc == start )  
 {  
 // we need to have loc == start;  
  
 node\* temp = loc->next;  
 loc->next = loc\_2->next;  
 loc->previous = ploc\_2;  
  
 loc\_2->next->previous = loc;  
 ploc\_2->next = loc;  
  
 loc\_2->next = temp;  
 temp->previous = loc\_2;  
 if (ploc!= nullptr)  
 {  
 ploc->next = loc\_2;  
 }  
 loc\_2->previous = ploc;  
  
 start = loc\_2;  
  
 }  
  
 else if (loc==last)  
 {  
 node\* temp = loc->next;  
 loc->next = loc\_2->next;  
 loc->previous = ploc\_2;  
  
 if(loc\_2->next!= nullptr)  
 {  
 loc\_2->next->previous = loc;  
 }  
 ploc\_2->next = loc;  
  
 loc\_2->next = temp;  
 temp->previous = loc\_2;  
 ploc->next = loc\_2;  
 loc\_2->previous = ploc;  
  
 last = loc\_2;  
 }  
 else  
  
  
 {  
 node\* temp = loc->next;  
 loc->next = loc\_2->next;  
 loc->previous = ploc\_2;  
  
 loc\_2->next->previous = loc;  
 ploc\_2->next = loc;  
  
 loc\_2->next = temp;  
 temp->previous = loc\_2;  
 ploc->next = loc\_2;  
 loc\_2->previous = ploc;  
  
  
  
  
  
 }  
  
  
  
  
  
  
 }  
  
  
  
}  
void linked\_list ::SwapNodes() {  
 loc = start;  
 if (start == last){  
  
 }  
 // loc will be used as the temp node  
  
 else if (start->next == last)  
 {  
 last->previous = nullptr;  
 last->next = start;  
 start->previous = last;  
 start->next = nullptr;  
 start = last;  
 last = loc;  
  
 }  
  
 else  
 {  
 //working on four nodes  
 ploc =loc; // we shall start from swapping the 1 and 2 node  
 loc=loc->next;  
 start = loc; // moving the start pointer one node ahead  
  
 node\* will\_effected\_link = nullptr;  
 while (loc!= nullptr)  
 {  
 ploc->next = loc->next;  
 loc->next = ploc;  
  
 loc->previous = ploc->previous;  
 ploc->previous = loc;  
 if(ploc->next != nullptr) {  
 ploc->next->previous = ploc;  
 }  
 if (will\_effected\_link != nullptr) {  
 will\_effected\_link->next = loc;  
 }  
 will\_effected\_link = loc;  
 loc = ploc->next;  
  
  
 }  
  
  
  
  
  
 }  
 loc = nullptr;  
 ploc = nullptr;  
  
  
}  
  
  
  
void linked\_list ::GroupEvenOdd() {  
 if (!isEmpty()){  
 if (start==last){  
 return;  
 }  
 else if(start->next == last){  
 return;  
 }  
  
 else{  
 // we will move the even ones to the last  
 node \*terminator = last;  
 loc = start; // will store the position  
 ploc = nullptr;  
 terminator = last;  
 bool shift\_start = false;  
 int remainder = 0;  
 if (start->data%2==0)  
 {  
 remainder = 1;  
 shift\_start = true;  
 }  
 // for starting with odd  
  
  
 ploc = loc;  
 loc=loc->next;  
  
 while (loc!=terminator)  
 {  
  
 if(loc->data%2==remainder) // shifting the even numbers to the end  
 {  
 ploc->next = loc->next;  
 ploc->next->previous = start;  
  
 loc->next = last->next;  
 last->next = loc;  
 loc->previous = last;  
  
  
 loc = ploc->next;  
 last = last->next;  
 }  
  
 else  
 {  
 ploc =loc;  
 loc=loc->next;  
 }  
  
 }  
  
  
 }  
  
  
  
 }  
}  
  
  
  
void linked\_list ::DestroyList() {  
 std::cout<<"dynamically deleted nodes with following contents: "<<std::endl;  
 while (start!=nullptr){  
 node \*temp = start;  
 start = start->next;  
 std::cout<<temp->data<<" ";  
 delete temp;  
 }  
 last = nullptr;  
 std::cout<<std::endl;  
}  
  
linked\_list :: ~linked\_list() {  
 DestroyList();  
  
}  
  
void linked\_list :: PrintReverse(){  
 node \*temp = last;  
 std::cout<<"List Reversed: ";  
 while(temp!= nullptr){  
 std::cout<<temp->data<<" ";  
 temp = temp->previous;  
 }  
 std::cout<<std::endl;  
  
}

## Relevent Screen Shots:

### GroupNodes ():

### ReverseOrderOfLinkedList():



### SwapWhere(10,5):

