**Kathmandu University**

**Department of Computer Science and Engineering**

**Dhulikhel, Kavre**



**COMP 202**

**Lab Report 1**

**Submitted by**

Aarush Timalsina

Roll No: 61

CE 2nd Year/1st Sem

**Submitted To**

Rajani Chulyadyo

Department of Computer Science and Engineering

**Linked List**

**LinkedList.h**

#ifndef LINKEDLIST\_H

#define LINKEDLIST\_H

class Node

{

public:

int info;

Node \*next;

};

class LinkedList

{

private:

Node \*HEAD;

Node \*TAIL;

public:

LinkedList();

~LinkedList();

bool isEmpty();

void addToHead(int data);

void addToTail(int data);

void add(int data, Node \*predecessor);

void removeFromHead();

void remove(int data);

Node \*retrieve(int data, Node \*outputNodePointer);

bool search(int data);

void traverse();

int getFromHead();

int getFromTail();

};

#endif

**LinkedList.cpp**

#include <iostream>

#include "LinkedList.h"

//Singly linked list implementation

LinkedList::LinkedList()

{

HEAD = NULL;

TAIL = NULL;

}

LinkedList::~LinkedList() {}

//Check if the list is empty and return integer value 0 for false and 1 for true

bool LinkedList::isEmpty()

{

return (this->HEAD == NULL);

}

//Add new data infront of the head

void LinkedList::addToHead(int data)

{

Node \*newNode = new Node();

newNode->info = data;

newNode->next = HEAD;

HEAD = newNode;

if (TAIL == NULL)

{

TAIL = HEAD;

}

}

//add new data at the tail end

void LinkedList::addToTail(int data)

{

Node \*newNode = new Node();

newNode->info = data;

newNode->next = NULL;

TAIL->next = newNode;

TAIL = newNode;

}

//add item betweeen the nodes

void LinkedList::add(int data, Node \*pre)

{

Node \*node = new Node();

node->info = data;

node->next = pre->next;

pre->next = node;

}

//delete data from the head

void LinkedList ::removeFromHead()

{

Node \*nodeToDelet = new Node();

nodeToDelet = HEAD;

HEAD = nodeToDelet->next;

delete nodeToDelet;

}

//delete data from node

void LinkedList::remove(int data)

{

//check if list is empty

if (this->isEmpty())

{

std::cout << "List was empty";

}

else

{

//check if the data present in head

if (this->HEAD->info == data)

{

this->removeFromHead();

if (this->HEAD == NULL)

this->TAIL == NULL;

}

else

{

Node \*temp = this->HEAD->next;

Node \*prev = this->HEAD;

while (temp != NULL)

{

if (temp->info == data)

{

//remove temp node

prev->next = temp->next;

delete temp;

if (prev->next == NULL)

this->TAIL = prev;

}

else

{

prev = prev->next;

temp = temp->next;

}

}

}

}

}

// (g) retrieve(data, outputNodePointer): Returns the pointer to the node with the requested data

Node \*LinkedList::retrieve(int data, Node \*opPtr)

{

//check if list is empty

if (this->isEmpty())

{

std::cout << "List was empty";

return 0;

}

else

{

//check if the data present in head

if (this->HEAD->info == data)

{

\*opPtr = \*this->HEAD;

return opPtr;

}

else

{

Node \*temp = this->HEAD->next;

while (temp != NULL)

{

if (temp->info == data)

{

\*opPtr = \*temp;

return opPtr;

}

else

{

temp = temp->next;

}

}

opPtr->info = -1;

return opPtr;

}

}

}

// (h) search(data): Returns true if the data exists in the list, and false otherwise

bool LinkedList::search(int data)

{

//check if list is empty

if (this->isEmpty())

{

std::cout << "List was empty";

return 0;

}

else

{

//check if the data present in head

if (this->HEAD->info == data)

{

return true;

}

else

{

Node \*temp = this->HEAD->next;

while (temp != NULL)

{

if (temp->info == data)

{

return true;

}

else

{

temp = temp->next;

}

}

return false;

}

}

}

void LinkedList::traverse()

{

//check if list is empty

if (this->isEmpty())

{

std::cout << "List was empty";

}

else

{

Node \*temp = this->HEAD;

while (temp != NULL)

{

std::cout << temp->info << std::endl;

temp = temp->next;

}

std::cout << std::endl;

}

}

int LinkedList::getFromHead() { return this->HEAD->info; }

int LinkedList::getFromTail() { return this->TAIL->info; }

//end of singly linked list implementation

int main() {

LinkedList l;

std::cout << l.isEmpty() << std::endl; //evaluate to 1(true)

l.addToHead(10);

l.addToHead(20);

l.addToTail(50);

std::cout << l.isEmpty() << std::endl;//evaluate to 0(false)

std::cout << l.getFromHead() << " -----> Head data" << std::endl;

std::cout << l.getFromTail() << " -----> tail data" << std::endl;

std::cout << l.search(20) << std::endl;

std::cout << l.search(230) << std::endl;

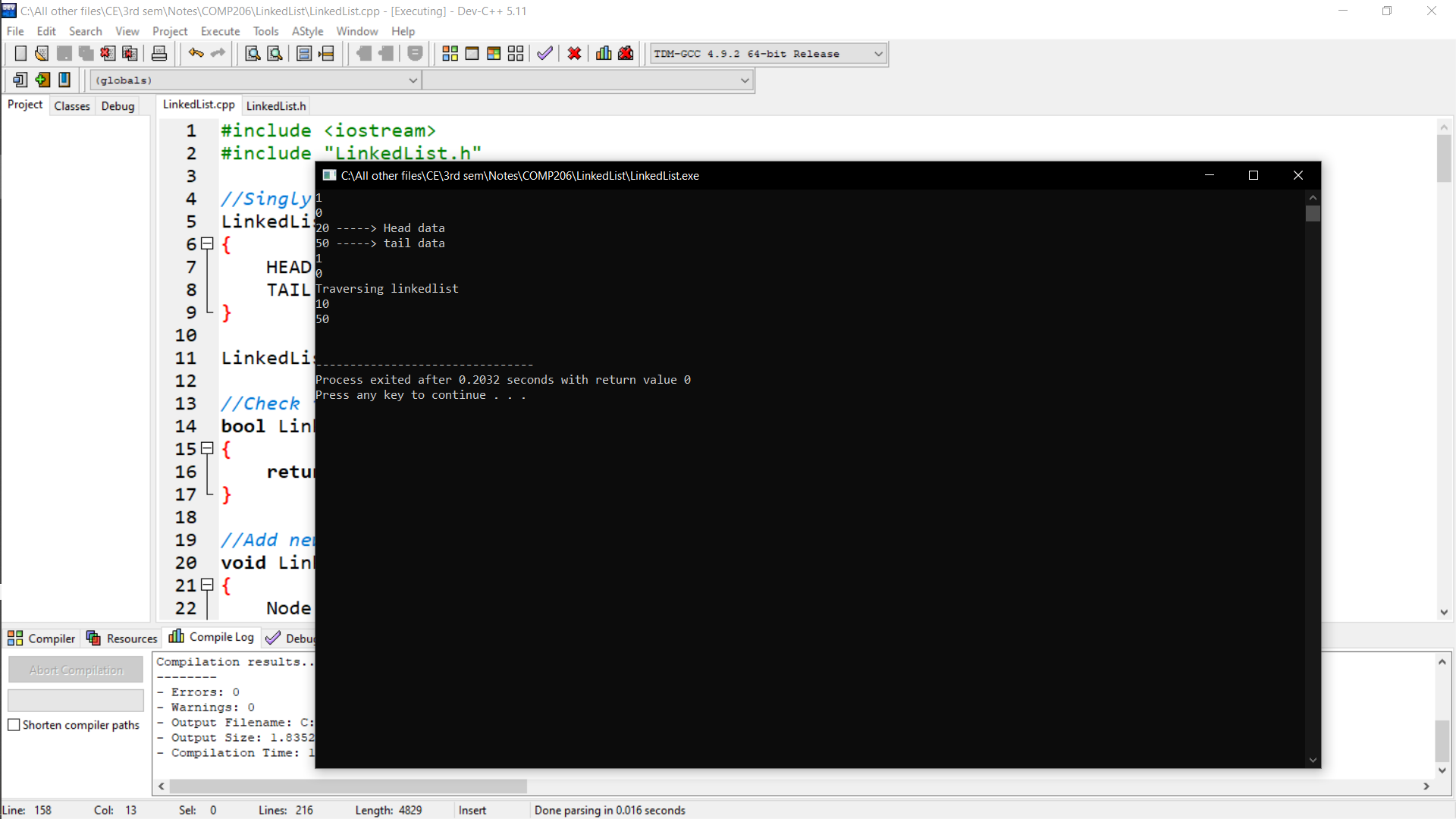
l.removeFromHead();

std::cout << "Traversing linkedlist\n";

l.traverse();

}

**Output:**



**Linked List - Queue**

**LLQueue.h**

#ifndef LLQUEUE\_H

#define LLQUEUE\_H

class Node

{

public:

int data;

Node \*next;

};

class QueueL

{

private:

Node \*front;

Node \*rear;

Node \*temp;

public:

QueueL();

~QueueL();

void enqueue();

void dequeue();

void display();

};

#endif

**LLQueue.cpp**

#include <iostream>

#include <cstdlib>

#include "LLQueue.h"

using namespace std;

QueueL::QueueL() {

front = NULL;

rear = NULL;

}

QueueL::~QueueL() {}

void QueueL::enqueue() {

int val;

cout<<"Insert the element in queue : "<<endl;

cin>>val;

if (rear == NULL) {

rear = (Node\*)malloc(sizeof(Node));

rear->next = NULL;

rear->data = val;

front = rear;

} else {

temp=(Node\*)malloc(sizeof(Node));

rear->next = temp;

temp->data = val;

temp->next = NULL;

rear = temp;

}

}

void QueueL::dequeue() {

temp = front;

if (front == NULL) {

cout<<"Underflow"<<endl;

return;

}

else

if (temp->next != NULL) {

temp = temp->next;

cout<<"Element deleted from queue is : "<<front->data<<endl;

free(front);

front = temp;

} else {

cout<<"Element deleted from queue is : "<<front->data<<endl;

free(front);

front = NULL;

rear = NULL;

}

}

void QueueL::display() {

temp = front;

if ((front == NULL) && (rear == NULL)) {

cout<<"Queue is empty"<<endl;

return;

}

cout<<"Queue elements are: ";

while (temp != NULL) {

cout<<temp->data<<" ";

temp = temp->next;

}

cout<<endl;

}

int main() {

QueueL q;

int ch;

cout<<"1) Insert element to queue"<<endl;

cout<<"2) Delete element from queue"<<endl;

cout<<"3) Display all the elements of queue"<<endl;

cout<<"4) Exit"<<endl;

do {

cout<<"Enter your choice : "<<endl;

cin>>ch;

switch (ch) {

case 1: q.enqueue();

break;

case 2: q.dequeue();

break;

case 3: q.display();

break;

case 4: cout<<"Exit"<<endl;

break;

default: cout<<"Invalid choice"<<endl;

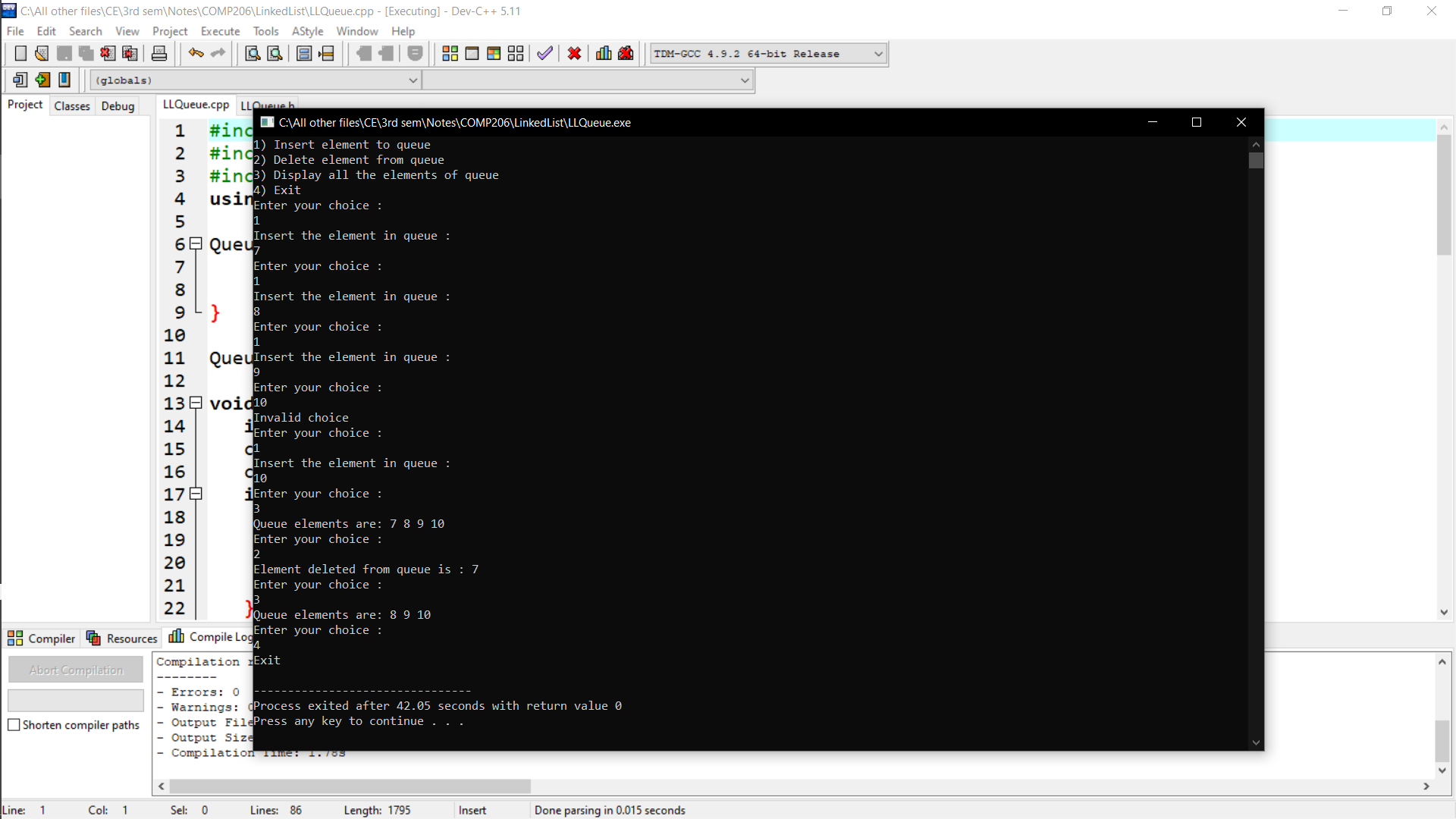
}

} while(ch!=4);

return 0;

}

**Output:**



**Linked List - Stack**

**LLStack.h**

#ifndef LSTACK\_H

#define LSTACK\_H

class Node

{

public:

int data;

Node \*next;

};

class StackL

{

private:

Node \*top;

public:

StackL();

~StackL();

void push(int);

void pop();

void display();

};

#endif

**LLStack.cpp**

#include <iostream>

#include "LLStack.h"

using namespace std;

StackL::StackL()

{

top = NULL;

}

StackL::~StackL() {}

void StackL::push(int val)

{

Node \*newnode = new Node();

newnode->data = val;

newnode->next = top;

top = newnode;

}

void StackL::pop()

{

if (top == NULL)

cout << "Stack Underflow" << endl;

else

{

cout << "The popped element is " << top->data << endl;

top = top->next;

}

}

void StackL::display()

{

struct Node \*ptr;

if (top == NULL)

cout << "stack is empty";

else

{

ptr = top;

cout << "Stack elements are: ";

while (ptr != NULL)

{

cout << "\n" << ptr->data;

ptr = ptr->next;

}

}

cout << endl;

}

int main()

{

StackL s;

int ch, val;

cout << "1) Push in stack" << endl;

cout << "2) Pop from stack" << endl;

cout << "3) Display stack" << endl;

cout << "4) Exit" << endl;

do

{

cout << "Enter choice: " << endl;

cin >> ch;

switch (ch)

{

case 1:

{

cout << "Enter value to be pushed:" << endl;

cin >> val;

s.push(val);

break;

}

case 2:

{

s.pop();

break;

}

case 3:

{

s.display();

break;

}

case 4:

{

cout << "Exit" << endl;

break;

}

default:

{

cout << "Invalid Choice" << endl;

}

}

} while (ch != 4);

return 0;

}

**Output:**

