Lab #13

Task#1:

#include <iostream>

using namespace std;

class Graph

{

private:

struct Node

{

char vertex; //attributes

Node\* next;

Node(char v)

{

vertex=v;

next = nullptr;

}

};

struct Queue

{

Node\* front;

Node\* rear;

Queue() //constructor

{

front=nullptr;

rear = nullptr;

}

void enqueue(char v)

{

Node\* newNode = new Node(v);

if (!rear)

{

front = rear = newNode;

return;

}

rear->next = newNode;

rear = newNode;

}

char dequeue()

{

if (!front)

return '\0';

char vertex = front->vertex;

Node\* temp = front;

front = front->next;

if (!front)

rear = nullptr;

delete temp;

return vertex;

}

bool isEmpty()

{

return (front == nullptr);

}

};

struct Stack

{

Node\* top;

Stack() : top(nullptr) {}

void push(char v) {

Node\* newNode = new Node(v);

newNode->next = top;

top = newNode;

}

char pop()

{

if (!top)

return '\0';

char vertex = top->vertex;

Node\* temp = top;

top = top->next;

delete temp;

return vertex;

}

bool isEmpty() {

return (top == nullptr);

}

};

Node\*\* adjList;

public:

Graph() {

adjList = new Node\*[26];

for (int i = 0; i < 26; i++)

adjList[i] = nullptr;

}

void addEdge(char a, char b)

{

int srcIndex = a - 'A';

int destIndex = b - 'A';

Node\* newNode = new Node(b);

newNode->next = adjList[srcIndex];

adjList[srcIndex] = newNode;

newNode = new Node(a);

newNode->next = adjList[destIndex];

adjList[destIndex] = newNode;

}

bool BFS(char start, char target, char parent[])

{

bool visited[26] = { false };

Queue q;

int startIndex = start - 'A';

int targetIndex = target - 'A';

visited[startIndex] = true;

q.enqueue(start);

while (!q.isEmpty()) {

char curr = q.dequeue();

int currIndex = curr - 'A';

for (Node\* neighbor = adjList[currIndex]; neighbor; neighbor = neighbor->next) {

int neighborIndex = neighbor->vertex - 'A';

if (!visited[neighborIndex]) {

q.enqueue(neighbor->vertex);

parent[neighborIndex] = curr;

visited[neighborIndex] = true;

if (neighborIndex == targetIndex)

return true;

}

}

}

return false;

}

bool DFS(char start, char target, char parent[]) {

bool visited[26] = { false };

Stack st;

int startIndex = start - 'A';

int targetIndex = target - 'A';

visited[startIndex] = true;

st.push(start);

while (!st.isEmpty()) {

char curr = st.pop();

int currIndex = curr - 'A';

if (currIndex == targetIndex)

return true;

for (Node\* neighbor = adjList[currIndex]; neighbor; neighbor = neighbor->next) {

int neighborIndex = neighbor->vertex - 'A';

if (!visited[neighborIndex]) {

st.push(neighbor->vertex);

parent[neighborIndex] = curr;

visited[neighborIndex] = true;

}

}

}

return false;

}

void display(char parent[], char end) {

cout << "Path from A to G: ";

char curr = end;

while (curr != '\0') {

cout << curr;

if (parent[curr - 'A'] != '\0')

cout << " -> ";

curr = parent[curr - 'A'];

}

cout << endl;

}

};

int main() {

Graph obj;

// Adding edges to the graph

obj.addEdge('A', 'B');

obj.addEdge('A', 'C');

obj.addEdge('B', 'D');

obj.addEdge('B', 'E');

obj.addEdge('C', 'F');

obj.addEdge('E', 'G');

char start = 'A';

char target = 'G';

char parentBFS[26] = { 0 };

char parentDFS[26] = { 0 };

cout << "The BFS TRAVERSAL IS =" << endl;

// Using BFS to find the path

if (obj.BFS(start, target, parentBFS)) {

obj.display(parentBFS, target);

}

else {

cout << "No path found from A to G using BFS.\n";

}

cout << "THE DFS TRAVERSAL IS = " << endl;

// Using DFS to find the path

if (obj.DFS(start, target, parentDFS))

{

obj.display(parentDFS, target);

}

else {

cout << "No path found from A to G using DFS.\n";

}

system("pause");

return 0;

}

Output:

