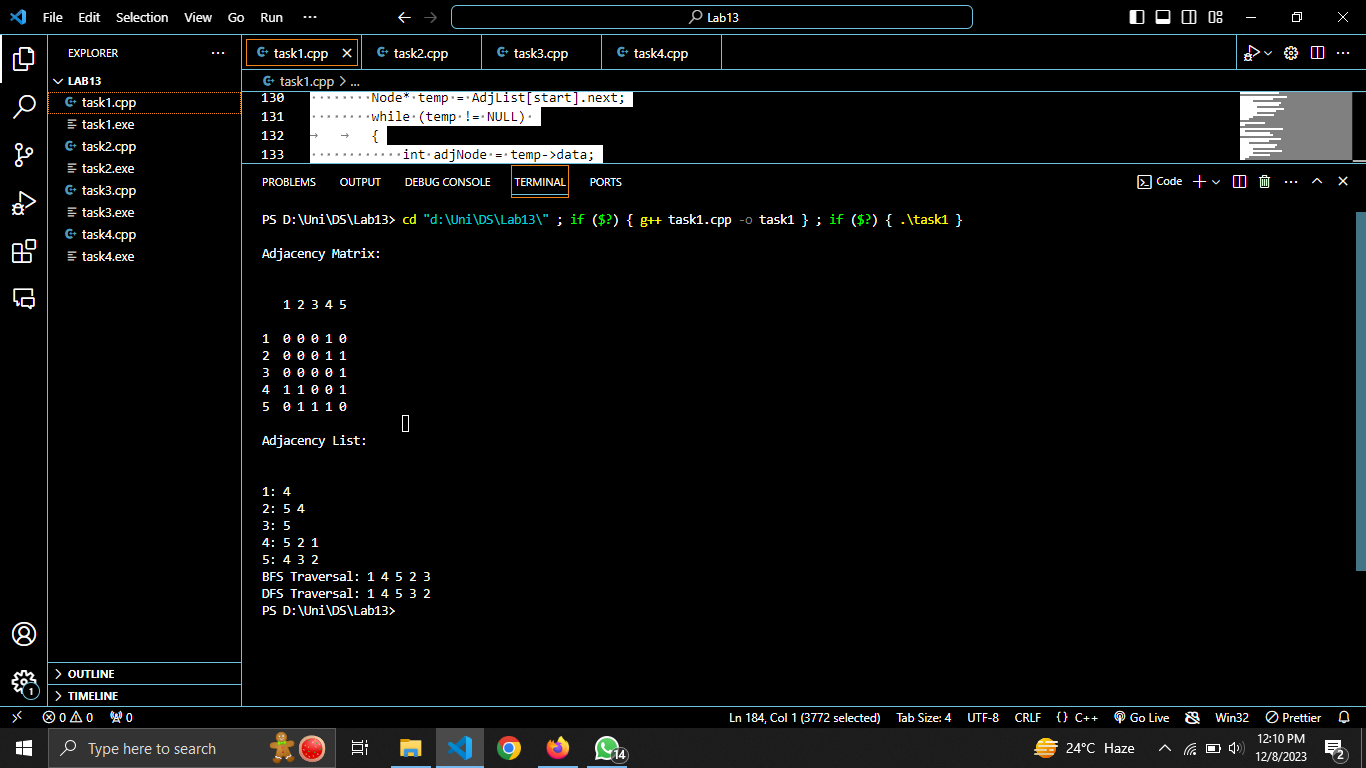
**LAB 12 DS**  
**K226007**

Task 1:



#include <iostream>

#include <list>

#include <queue>

using namespace std;

class Node

{

public:

int data;

Node\* next;

Node() : data(0), next(NULL) {}

Node(int d) : data(d), next(NULL) {}

};

class Graph

{

private:

bool\*\* adj;

int vertices;

Node\* AdjList;

public:

Graph(int v) {

this->vertices = v;

adj = new bool\*[vertices];

for (int i = 0; i < vertices; i++) {

adj[i] = new bool[vertices];

for (int j = 0; j < vertices; j++) {

adj[i][j] = false;

}

}

AdjList = new Node[vertices];

}

void addEdge(int i, int j)

{

adj[i][j] = true;

adj[j][i] = true;

}

void printMat() {

cout << "\nAdjacency Matrix: \n\n";

cout << "\n 1 2 3 4 5\n";

for (int i = 0; i < vertices; i++) {

cout << endl

<< i + 1 << " ";

for (int j = 0; j < vertices; j++) {

cout << adj[i][j] << " ";

}

}

}

void \_AdjList()

{

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

{

for (int j = 0; j < vertices; j++)

{

if (adj[i][j])

{

Node\* temp = new Node(j);

temp->next = AdjList[i].next;

AdjList[i].next = temp;

}

}

}

}

void printList()

{

cout << "\n\nAdjacency List: \n\n";

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

{

cout << endl

<< i + 1 << ": ";

Node\* temp = AdjList[i].next;

while (temp != NULL)

{

cout << temp->data + 1 << " ";

temp = temp->next;

}

}

}

void BFS(int start)

{

bool\* visited = new bool[vertices];

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

{

visited[i] = false;

}

queue<int> q;

visited[start] = true;

q.push(start);

cout << "\nBFS Traversal: ";

while (!q.empty())

{

int current = q.front();

q.pop();

cout << current + 1 << " ";

Node\* temp = AdjList[current].next;

while (temp != NULL)

{

int adjNode = temp->data;

if (!visited[adjNode])

{

visited[adjNode] = true;

q.push(adjNode);

}

temp = temp->next;

}

}

delete[] visited;

}

void DFSUtil(int start, bool\* visited)

{

visited[start] = true;

cout << start + 1 << " ";

Node\* temp = AdjList[start].next;

while (temp != NULL)

{

int adjNode = temp->data;

if (!visited[adjNode])

{

DFSUtil(adjNode, visited);

}

temp = temp->next;

}

}

void DFS(int start)

{

bool\* visited = new bool[vertices];

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

{

visited[i] = false;

}

cout << "\nDFS Traversal: ";

DFSUtil(start, visited);

delete[] visited;

}

~Graph()

{

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

delete[] adj[i];

delete[] adj;

delete[] AdjList;

}

};

int main()

{

Graph obj(5);

obj.addEdge(0, 3);

obj.addEdge(3, 4);

obj.addEdge(3, 1);

obj.addEdge(1, 4);

obj.addEdge(4, 2);

obj.printMat();

obj.\_AdjList();

obj.printList();

obj.BFS(0);

obj.DFS(0);

return 0;

}

Task 2:  
  


#include <iostream>

using namespace std;

#define MAX\_NODES 100

#define INFINITY 10000

struct Edge

{

int source;

int destination;

int weight;

};

struct Node

{

int distance;

bool visited;

};

void dijkstra(Edge edges[], int num\_edges, int num\_nodes, int start\_vertex, int distance[])

{

Node nodes[MAX\_NODES];

for (int i = 0; i < num\_nodes; i++)

{

nodes[i].distance = INFINITY;

nodes[i].visited = false;

}

nodes[start\_vertex].distance = 0;

while (true)

{

int min\_distance = INFINITY;

int min\_index = -1;

for (int i = 0; i < num\_nodes; i++)

{

if (!nodes[i].visited && nodes[i].distance < min\_distance)

{

min\_distance = nodes[i].distance;

min\_index = i;

}

}

if (min\_index == -1)

{

break;

}

nodes[min\_index].visited = true;

for (int i = 0; i < num\_nodes; i++)

{

if (edges[i].source == min\_index)

{

int neighbor = edges[i].destination;

int weight = edges[i].weight;

if (nodes[neighbor].distance > nodes[min\_index].distance + weight)

{

nodes[neighbor].distance = nodes[min\_index].distance + weight;

}

}

}

}

for (int i = 0; i < num\_nodes; i++)

{

distance[i] = nodes[i].distance;

}

}

int main()

{

int num\_nodes = 6;

int num\_edges = 6;

Edge edges[] = {

{0, 1, 8},

{0, 2, 6},

{1, 3, 9},

{2, 3, 4},

{3, 4, 14},

{3, 5, 3},

};

int distance[MAX\_NODES];

dijkstra(edges, num\_edges, num\_nodes, 1, distance);

cout << "Shortest path from node B to node E:" << endl;

cout << distance[5] << endl;

return 0;

}

Task 3:  
  


#include <iostream>

#include <list>

#include <stack>

using namespace std;

struct Vertex {

char label;

list<char> neighbors;

};

Vertex graph[26];

stack<char> topological\_order;

bool is\_valid\_vertex(char vertex) {

return vertex >= 'A' && vertex <= 'Z';

}

void add\_edge(char source, char destination) {

if (!is\_valid\_vertex(source) || !is\_valid\_vertex(destination)) {

throw invalid\_argument("Invalid vertex");

}

graph[source - 'A'].neighbors.push\_back(destination);

}

void mark\_visited(char vertex, bool visited[]) {

visited[vertex - 'A'] = true;

}

void topological\_sort\_util(char vertex, bool visited[]) {

mark\_visited(vertex, visited);

for (char neighbor : graph[vertex - 'A'].neighbors) {

if (!visited[neighbor - 'A']) {

topological\_sort\_util(neighbor, visited);

}

}

topological\_order.push(vertex);

}

void print\_topological\_order() {

bool visited[26] = {false};

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

if (!visited[i]) {

topological\_sort\_util('A' + i, visited);

}

}

while (!topological\_order.empty()) {

cout << topological\_order.top() << " ";

topological\_order.pop();

}

}

int main() {

char vertices[] = "MNOQRSTUVWXZY";

add\_edge('M', 'O');

add\_edge('M', 'N');

add\_edge('N', 'O');

add\_edge('N', 'P');

add\_edge('O', 'Q');

add\_edge('O', 'R');

add\_edge('P', 'Q');

add\_edge('P', 'S');

add\_edge('Q', 'T');

add\_edge('Q', 'U');

add\_edge('R', 'S');

add\_edge('R', 'V');

add\_edge('S', 'T');

add\_edge('S', 'U');

add\_edge('S', 'V');

add\_edge('T', 'W');

add\_edge('U', 'X');

add\_edge('V', 'X');

add\_edge('W', 'Y');

add\_edge('X', 'Z');

print\_topological\_order();

return 0;

}

Task 4:



#include <iostream>

#include <vector>

#define NUM\_NODES 6

using namespace std;

const int INF = 1e9;

int graph[NUM\_NODES][NUM\_NODES] = {

{0, 3, 4, 0, 0, 0},

{3, 0, 2, 3, 6, 0},

{4, 2, 0, 1, 0, 0},

{0, 3, 1, 0, 4, 2},

{0, 6, 0, 4, 0, 1},

{0, 0, 0, 2, 1, 0}

};

int min\_distance(vector<int>& dist, vector<bool>& spt\_set) {

int min = INF, min\_index;

for (int v = 0; v < NUM\_NODES; ++v) {

if (!spt\_set[v] && dist[v] <= min) {

min = dist[v];

min\_index = v;

}

}

return min\_index;

}

void dijkstra(int src, int dest) {

vector<int> dist(NUM\_NODES, INF);

vector<bool> spt\_set(NUM\_NODES, false);

dist[src] = 0;

for (int count = 0; count < NUM\_NODES - 1; ++count) {

int u = min\_distance(dist, spt\_set);

spt\_set[u] = true;

for (int v = 0; v < NUM\_NODES; ++v) {

if (!spt\_set[v] && graph[u][v] && dist[u] != INF && dist[u] + graph[u][v] < dist[v]) {

dist[v] = dist[u] + graph[u][v];

}

}

}

cout << "Shortest path from Node " << src + 1 << " to Node " << dest + 1 << ": ";

int node = dest;

while (node != src) {

cout << node + 1 << " <- ";

node = dist[node];

}

cout << src + 1 << endl;

cout << "Total weight of the path: " << dist[dest] << endl;

}

int main() {

int source = 0;

int destination = 5;

dijkstra(source, destination);

return 0;

}