National University of Computer and Emerging Sciences



Name: Muhammad Suleman

Roll #: 22F-3350

Section: BCS-4B

Lab # 13

# Problem 01

#include <iostream>

using namespace std;

int \*\*createAdjacencyMatrix(int vertices, int edges, int arr[][2])

{

int \*\*adjMatrix = new int \*[vertices];

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

{

adjMatrix[i] = new int[vertices];

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

{

adjMatrix[i][j] = 0;

}

}

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

{

int x = arr[i][0];

int y = arr[i][1];

adjMatrix[x - 1][y - 1] = 1;

adjMatrix[y - 1][x - 1] = 1;

}

return adjMatrix;

}

void displayAdjacencyMatrix(int \*\*adjMatrix, int N)

{

cout << "\nDisplaying Adjacency Matrix of " << N << " \* " << N << endl;

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

{

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

{

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

}

cout << endl;

}

cout << endl;

}

int main()

{

int vertices = 8;

int edges = 7;

int arr[][2] = {{1, 2}, {2, 3}, {4, 5}, {1, 5}, {6, 1}, {7, 4}, {3, 8}};

cout << "Input: "

<< "Vertices: " << vertices << ", Edges: " << edges << endl;

cout << "Array arr[][]: ";

cout << "{";

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

{

cout << " { " << arr[i][0] << " , " << arr[i][1] << " } ";

}

cout << "}" << endl;

int \*\*adjMatrix = createAdjacencyMatrix(vertices, edges, arr);

displayAdjacencyMatrix(adjMatrix, vertices);

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

{

delete[] adjMatrix[i];

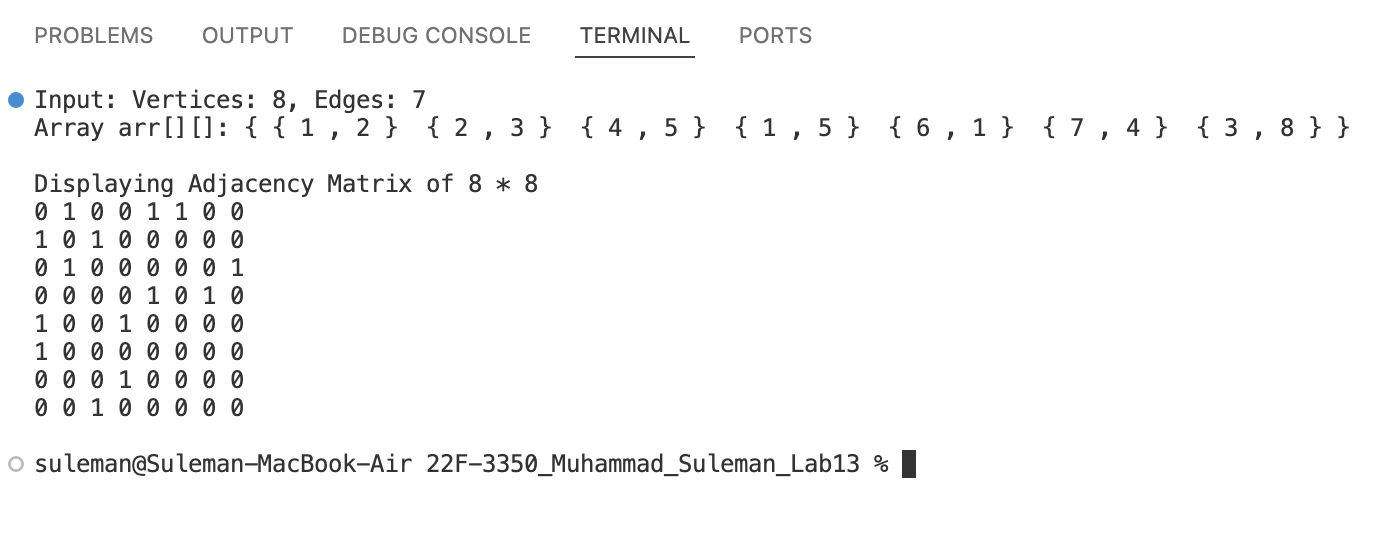
}

delete[] adjMatrix;

return 0;

}

## Output



# Problem 02

#include <iostream>

using namespace std;

class Node

{

public:

int data;

Node \*next;

Node()

: data(0), next(nullptr) {}

};

class Graph

{

int numVertices;

Node \*\*adjList;

public:

Graph(int vertices)

: numVertices(vertices)

{

adjList = new Node \*[numVertices];

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

{

adjList[i] = nullptr;

}

}

void addEdge(int v1, int v2)

{

Node \*newNode = new Node;

newNode->data = v2;

newNode->next = adjList[v1];

adjList[v1] = newNode;

newNode = new Node;

newNode->data = v1;

newNode->next = adjList[v2];

adjList[v2] = newNode;

}

void printGraph()

{

cout << endl;

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

{

cout << "Adjacency list of vertex " << i << ": ";

Node \*temp = adjList[i];

if (temp == nullptr)

{

cout << "None";

}

while (temp != nullptr)

{

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

temp = temp->next;

}

cout << endl;

}

cout << endl;

}

};

int main()

{

int vertices, edges;

cout << "Enter the number of vertices: ";

cin >> vertices;

cout << "Enter the number of edges: ";

cin >> edges;

Graph graph(vertices);

cout << "Enter the edges (u v):" << endl;

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

{

int v1, v2;

cin >> v1 >> v2;

graph.addEdge(v1, v2);

}

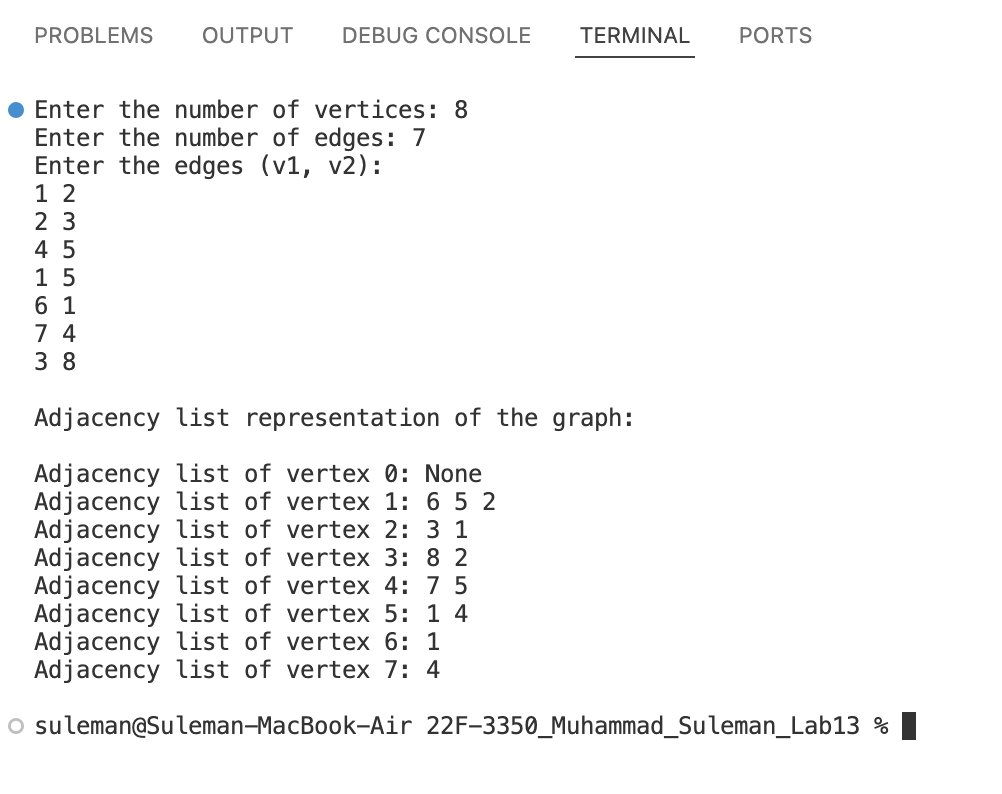
cout << "\nAdjacency list representation of the graph:" << endl;

graph.printGraph();

return 0;

}

## Output



# Problem 03

#include <iostream>

#include <queue>

#include <stack>

#include <string>

using namespace std;

// Edge class to represent edges

class Edge

{

private:

int from;

int to;

int dist;

public:

Edge(int from, int to, int dist)

: from(from), to(to), dist(dist) {}

// Getters

int getFrom()

{

return from;

}

int getTo()

{

return to;

}

int getDist()

{

return dist;

}

};

// Vertex class to represent vertices

class Vertex

{

private:

int id;

string name;

public:

Vertex() {}

Vertex(int id, string name)

: id(id), name(name) {}

// Getters

int getId()

{

return id;

}

string getName()

{

return name;

}

// Setters

void setId(int id)

{

this->id = id;

}

void setName(string name)

{

this->name = name;

}

};

// Graph class to represent the graph structure

class Graph

{

private:

Vertex \*verts;

Edge \*\*\*adjMat;

int vCount;

int maxVerts;

public:

Graph(int maxVerts)

: maxVerts(maxVerts), vCount(0)

{

verts = new Vertex[maxVerts];

adjMat = new Edge \*\*[maxVerts];

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

{

adjMat[i] = new Edge \*[maxVerts];

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

{

adjMat[i][j] = nullptr;

}

}

}

~Graph()

{

delete[] verts;

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

{

delete[] adjMat[i];

}

delete[] adjMat;

}

bool vertexExistsById(int id)

{

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

{

if (verts[i].getId() == id)

{

return true;

}

}

return false;

}

void addVertex(Vertex newVertex)

{

verts[vCount++] = newVertex;

}

Vertex getVertexById(int vid)

{

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

{

if (verts[i].getId() == vid)

{

return verts[i];

}

}

return Vertex();

}

bool edgeExistsById(int from, int to)

{

return adjMat[from][to] != nullptr;

}

void addEdge(int from, int to, int weight)

{

if (from >= 0 && from < maxVerts && to >= 0 && to < maxVerts)

{

adjMat[from][to] = new Edge(from, to, weight);

}

}

// BFS traversal

void BFS(int startVertexId)

{

bool \*visited = new bool[maxVerts];

queue<int> q;

visited[startVertexId] = true;

q.push(startVertexId);

while (!q.empty())

{

int currentVertexId = q.front();

q.pop();

cout << "Visited: " << verts[currentVertexId].getName() << endl;

// Iterate through adjacent vertices

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

{

if (adjMat[currentVertexId][i] != nullptr && !visited[i])

{

visited[i] = true;

q.push(i);

}

}

}

delete[] visited;

}

// DFS traversal

void DFS(int startVertexId)

{

bool \*visited = new bool[maxVerts];

stack<int> s;

visited[startVertexId] = true;

s.push(startVertexId);

while (!s.empty())

{

int currentVertexId = s.top();

s.pop();

cout << "Visited: " << verts[currentVertexId].getName() << endl;

// Iterate through adjacent vertices

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

{

if (adjMat[currentVertexId][i] != nullptr && !visited[i])

{

visited[i] = true;

s.push(i);

}

}

}

delete[] visited;

}

};

int main()

{

Graph graph(10);

graph.addVertex(Vertex(0, "Karachi"));

graph.addVertex(Vertex(1, "Lahore"));

graph.addVertex(Vertex(2, "Islamabad"));

graph.addEdge(0, 1, 1);

graph.addEdge(0, 2, 1);

graph.addEdge(1, 2, 1);

cout << "BFS Traversal:" << endl;

graph.BFS(0);

cout << endl;

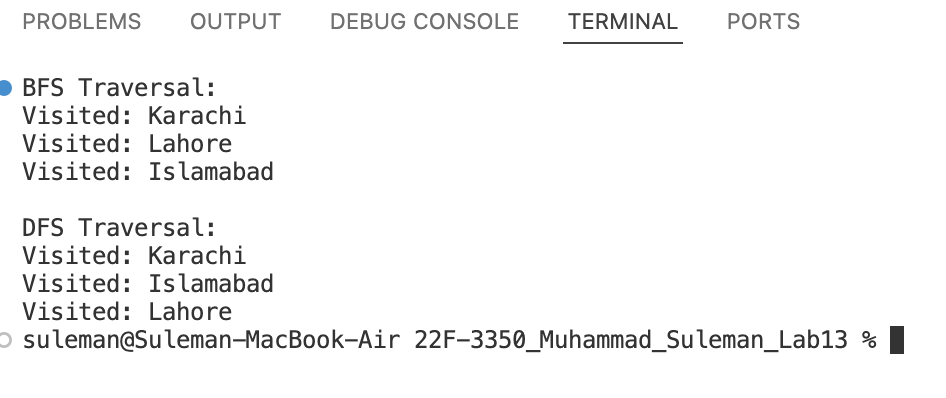
cout << "DFS Traversal:" << endl;

graph.DFS(0);

return 0;

}

## Output



# Problem 04

#include <iostream>

using namespace std;

class Graph

{

int vertices;

int \*\*adjMatrix;

bool isCyclicUtil(int v, bool visited[], int parent)

{

visited[v] = true;

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

{

if (adjMatrix[v][i])

{

if (!visited[i])

{

if (isCyclicUtil(i, visited, v))

{

return true;

}

}

else if (i != parent)

{

return true;

}

}

}

return false;

}

public:

Graph(int vertices)

: vertices(vertices)

{

adjMatrix = new int \*[vertices];

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

{

adjMatrix[i] = new int[vertices];

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

{

adjMatrix[i][j] = 0;

}

}

}

~Graph()

{

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

{

delete[] adjMatrix[i];

}

delete[] adjMatrix;

}

void addEdge(int v1, int v2)

{

adjMatrix[v1][v2] = 1;

adjMatrix[v2][v1] = 1;

}

bool isCyclic()

{

bool \*visited = new bool[vertices];

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

{

visited[i] = false;

}

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

{

if (!visited[i] && isCyclicUtil(i, visited, -1))

{

delete[] visited;

return true;

}

}

delete[] visited;

return false;

}

void displayAdjacencyMatrix()

{

cout << "\nDisplaying Adjacency Matrix of " << vertices << " \* " << vertices << endl;

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

{

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

{

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

}

cout << endl;

}

cout << endl;

}

};

int main()

{

int vertices = 5;

Graph graph(vertices);

graph.addEdge(1, 0);

graph.addEdge(0, 2);

graph.addEdge(2, 1);

graph.addEdge(0, 3);

graph.addEdge(3, 4);

graph.displayAdjacencyMatrix();

if (graph.isCyclic())

{

cout << "Graph is cyclic" << endl;

}

else

{

cout << "Graph is not cyclic" << endl;

}

return 0;

}

## Output

