# Department of Computing

# School of Electrical Engineering and Computer Science

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

**Class: BESE 13A**

# Lab 12: Graph Traversal

**Date: 15th December, 2023**

**Time: 10 am - 1 pm & 2 pm – 5 pm**

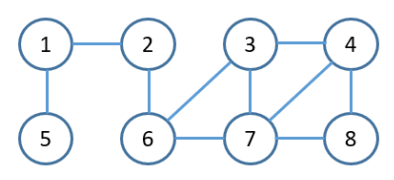
**Lab Engineer: Anum Asif**

# 

# Lab 12: Graph Traversal

**Lab Tasks**

Implement BFS traversal algorithm studied in class for a graph with 8 vertices.  
As an initial input we have the graph (G) with 8 nodes shown below with and a starting vertex 2.



**Input:**#include <iostream>

#include <queue>

using namespace std;

#define MAX\_VERTICES 8

class Graph {

    int\*\* adjacencyMatrix;

    int numVertices;

    bool\* visited;

public:

    Graph(int numVertices);

    void addEdge(int vertex1, int vertex2);

    void breadthFirstTraversal(int startVertex);

    ~Graph();

};

Graph::Graph(int numVertices) {

    numVertices++;  // Increase by 1 to account for 1-based indexing

    this->numVertices = numVertices;

    // Initialize adjacency matrix

    adjacencyMatrix = new int\*[numVertices];

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

        adjacencyMatrix[i] = new int[numVertices];

    }

    // Initialize the matrix with zeros

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

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

            adjacencyMatrix[i][j] = 0;

        }

    }

}

void Graph::addEdge(int vertex1, int vertex2) {

    if (vertex1 < numVertices && vertex2 < numVertices) {

        adjacencyMatrix[vertex1][vertex2] = 1;

        // The graph is undirected

        adjacencyMatrix[vertex2][vertex1] = 1;

    } else {

        cout << "Invalid vertices" << endl;

    }

}

void Graph::breadthFirstTraversal(int startVertex) {

    cout << "Breadth-first traversal starting from " << startVertex << " is:" << endl;

    // Initialize visited to all false

    visited = new bool[numVertices];

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

        visited[i] = false;

    }

    // Create a queue

    queue<int> q;

    int currentVertex;

    // Start from startVertex

    q.push(startVertex);

    // Go through all vertices, including disconnected ones

    while (true) {

        // Keep iterating until the queue is not empty

        while (!q.empty()) {

            // Visit the current vertex

            currentVertex = q.front();

            visited[currentVertex] = true;

            // Explore neighbors

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

                if (adjacencyMatrix[i][currentVertex] && !visited[i]) {

                    q.push(i);

                    visited[i] = true;

                }

            }

            // Print the current vertex

            cout << currentVertex << " ";

            q.pop();

        }

        currentVertex = 0;

        // Check for disconnected vertices

        for (int i = 1; i < numVertices; i++) {

            if (!visited[i]) {

                // If a disconnected vertex is found, start from there

                currentVertex = i;

                q.push(i);

                break;

            }

        }

        if (currentVertex == 0) {

            break;

        }

    }

    cout << endl;

    // Deallocate memory

    delete[] visited;

}

Graph::~Graph() {

    // Deallocate the adjacency matrix

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

        delete[] adjacencyMatrix[i];

    }

    delete[] adjacencyMatrix;

}

int main() {

    // Create the graph

    Graph myGraph(MAX\_VERTICES);

    // Add edges

    myGraph.addEdge(1, 2);

    myGraph.addEdge(1, 5);

    myGraph.addEdge(2, 6);

    myGraph.addEdge(6, 3);

    myGraph.addEdge(6, 7);

    myGraph.addEdge(3, 4);

    myGraph.addEdge(3, 7);

    myGraph.addEdge(7, 4);

    myGraph.addEdge(4, 8);

    myGraph.addEdge(7, 8);

    // Perform breadth-first traversal starting from vertex 2

    myGraph.breadthFirstTraversal(2);

    return 0;

}

**Output:**

****