**Kathmandu University**

**Department of Computer Science and Engineering**

**Dhulikhel, Kavre**



**A**

**Lab Report**

**On**

**“Graph”**

**[Code No.: COMP 202]**

**Submitted by:**

Sushma Acharya(03)

Aashish Adhikari(04)

CE: UNG (II/I)

**Submitted to:**

Dr. Rajani Chulyadyo

Department of Computer Science and Engineering

**Submission Date:** 13/07/2024

**Lab Report 5**

**Graph**

Graphs are non-linear data structures consisting of a finite set of vertices (or nodes) and a set of edges connecting these vertices. Each edge connects a pair of vertices, which can represent a wide variety of relationships in different domains such as social networks, transportation systems, and more. This report aims to demonstrate the functionality and applications of graph data structures through code demonstrations.

**Operations**

The major operations performed in a graph data structure are as follows:

1. **isEmpty()**: Returns true if the graph is empty, and false otherwise.
2. **isDirected()**: Returns true if the graph is directed, and false otherwise.
3. **addVertex(newVertex)**: Inserts a new vertex to the graph.
4. **addEdge(vertex1, vertex2)**: Adds an edge from vertex1 to vertex2.
5. **removeVertex(vertexToRemove)**: Removes a vertex from the graph.
6. **removeEdge(vertex1, vertex2)**: Removes an edge from the graph.
7. **numVertices()**: Returns the number of vertices in the graph.
8. **numEdges()**: Returns the number of edges in the graph.
9. **indegree(vertex)**: Returns the indegree of a vertex.
10. **outdegree(vertex)**: Returns the outdegree of a vertex.
11. **degree(vertex)**: Returns the degree of a vertex.
12. **neighbours(vertex)**: Returns the neighbours of a vertex.
13. **neighbour(vertex1, vertex2)**: Returns true if vertex2 is a neighbour of vertex1.

**Github Link to Code**

The GitHub link to the repository for the implementation of graph data structure is:

<https://github.com/suslabok/lab5>

**Functions and Outputs:**

The outputs for each operation performed on the singly linked list are displayed below:

**Class Graph and its constructor:**

class Graph {

private:

    std::vector<std::vector<int>> adjMatrix;

    int vertexCount;

    bool directed;

public:

    Graph(bool isDirected = false);

    bool isEmpty() const;

    bool isDirected() const;

    void addVertex();

    void addEdge(int vertex1, int vertex2);

    void removeVertex(int vertex);

    void removeEdge(int vertex1, int vertex2);

    int numVertices() const;

    int numEdges() const;

    int indegree(int vertex) const;

    int outdegree(int vertex) const;

    int degree(int vertex) const;

    std::vector<int> neighbours(int vertex) const;

    bool neighbour(int vertex1, int vertex2) const;

};

Graph::Graph(bool isDirected) : directed(isDirected), vertexCount(0) {}

1. **isEmpty():**

bool Graph::isEmpty() const {

    return vertexCount == 0;

}

1. **isDirected():**

bool Graph::isDirected() const {

    return directed;

}

1. **addVertex(newVertex):**

void Graph::addVertex() {

    vertexCount++;

    for (auto &row : adjMatrix) {

        row.push\_back(0);

    }

    adjMatrix.push\_back(std::vector<int>(vertexCount, 0));

}

1. **addEdge(vertex1, vertex2):**

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

    if (vertex1 >= vertexCount || vertex2 >= vertexCount) return;

    adjMatrix[vertex1][vertex2] = 1;

    if (!directed) {

        adjMatrix[vertex2][vertex1] = 1;

    }

}

1. **removeVertex(vertexToRemove):**

void Graph::removeVertex(int vertex) {

    if (vertex >= vertexCount) return;

    adjMatrix.erase(adjMatrix.begin() + vertex);

    for (auto &row : adjMatrix) {

        row.erase(row.begin() + vertex);

    }

    vertexCount--;

}

1. **removeEdge(vertex1, vertex2):**

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

    if (vertex1 >= vertexCount || vertex2 >= vertexCount) return;

    adjMatrix[vertex1][vertex2] = 0;

    if (!directed) {

        adjMatrix[vertex2][vertex1] = 0;

    }

}

1. **numVertices():**

int Graph::numVertices() const {

    return vertexCount;

}

1. **numEdges():**

int Graph::numEdges() const {

    int count = 0;

    for (const auto &row : adjMatrix) {

        for (int edge : row) {

            count += edge;

        }

    }

    return directed ? count : count / 2;

}

1. **indegree(vertex):**

int Graph::indegree(int vertex) const {

    if (vertex >= vertexCount) return -1;

    int count = 0;

    for (const auto &row : adjMatrix) {

        count += row[vertex];

    }

    return count;

}

1. **outdegree(vertex):**

int Graph::outdegree(int vertex) const {

    if (vertex >= vertexCount) return -1;

    int count = 0;

    for (int edge : adjMatrix[vertex]) {

        count += edge;

    }

    return count;

}

1. **degree(vertex):**

int Graph::degree(int vertex) const {

    if (directed) {

        return indegree(vertex) + outdegree(vertex);

    } else {

        return outdegree(vertex);

    }

}

1. **neighbours(vertex):**

std::vector<int> Graph::neighbours(int vertex) const {

    std::vector<int> neighbors;

    if (vertex >= vertexCount) return neighbors;

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

        if (adjMatrix[vertex][i]) {

            neighbors.push\_back(i);

        }

    }

    return neighbors;

}

1. **neighbour(vertex1, vertex2):**

bool Graph::neighbour(int vertex1, int vertex2) const {

    if (vertex1 >= vertexCount || vertex2 >= vertexCount)

return false;

    return adjMatrix[vertex1][vertex2] == 1;

}

**Main function:**

#include <iostream>

#include "graph.h"

using namespace std;

int main() {

    Graph g(true);  // Create a directed graph

    // Add vertices

    g.addVertex();

    g.addVertex();

    g.addVertex();

    g.addVertex();

    // Add edges

    g.addEdge(0, 1);

    g.addEdge(1, 2);

    g.addEdge(2, 3);

    g.addEdge(3, 0);

    // Display graph properties

    std::cout << "Graph is empty: " << g.isEmpty() << std::endl;

    std::cout << "Graph is directed: " << g.isDirected() << std::endl;

    std::cout << "Number of vertices: " << g.numVertices() << std::endl;

    std::cout << "Number of edges: " << g.numEdges() << std::endl;

    // Display degree information

    std::cout << "Indegree of vertex 1: " << g.indegree(1) << std::endl;

    std::cout << "Outdegree of vertex 1: " << g.outdegree(1) << std::endl;

    std::cout << "Degree of vertex 1: " << g.degree(1) << std::endl;

    // Display neighbors of a vertex

    std::vector<int> neighbors = g.neighbours(1);

    std::cout << "Neighbors of vertex 1: ";

    for (int v : neighbors) {

        std::cout << v << " ";

    }

    std::cout << std::endl;

    // Check if vertices are neighbors

   cout << "Vertex 1 is neighbor of vertex 0: " << g.neighbour(1, 0) << endl;

    g.removeEdge(1, 2);

    cout << "Number of edges after removing edge (1,2): " << g.numEdges() << endl;

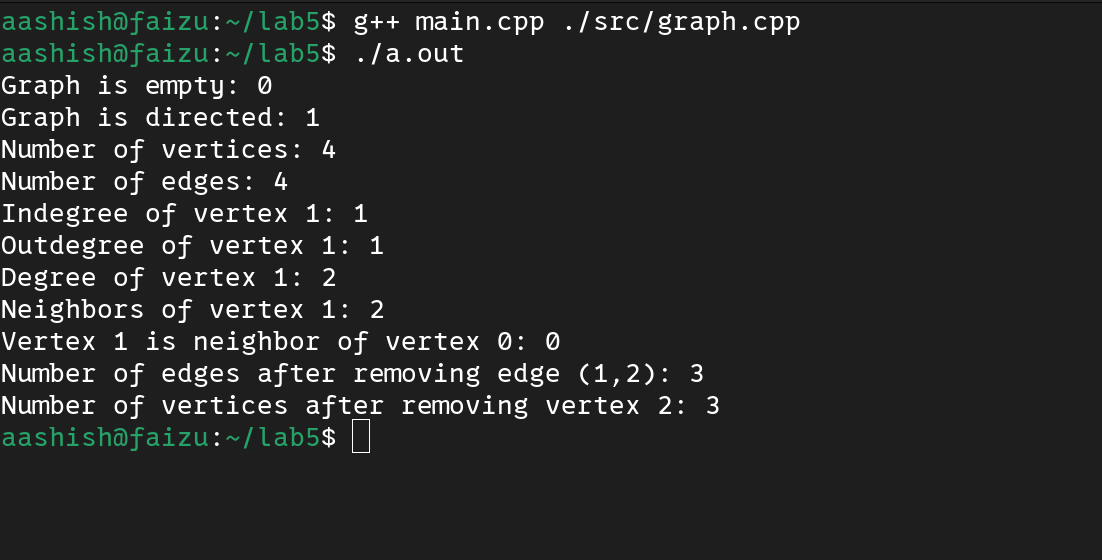
    g.removeVertex(2);

    cout << "Number of vertices after removing vertex 2:" << g.numVertices() << endl;

    return 0;

}

**Output:**

****

**Conclusion:**

Overall, graphs are versatile and powerful non-linear data structures with various operations to manage vertices and edges, allowing complex relationships and structures to be represented efficiently. They are essential in many applications due to their dynamic nature and ability to represent both sparse and dense connections. Graphs can be used to implement and model numerous real-world systems such as social networks, transportation networks, and more. With operations like adding/removing vertices and edges, and querying neighbors and degrees, graphs provide a robust framework for complex data representation and manipulation.