| Activity No. 10 | |
|---|---|
| **GRAPHS** | |
| **Course Code:** CPE010 | **Program:** Computer Engineering |
| **Course Title:** Data Structures and Algorithms | **Date Performed:** 13/11/2024 |
| **Section:** CPE21S4 | **Date Submitted:** 13/11/2024 |
| **Name(s):** Mamaril, Justin Kenneth I,<br>San Juan, Edson<br>San Jose, Alexander<br>Reyes, Alexzander<br>Titong, Lee Ivan | **Instructor:** Prof. Maria Rizette Sayo |
| **A. Output(s) and Observation(s)** | |

**ILO A: Create C++ code for graph implementation utilizing adjacency matrix and adjacency list**

```
#include <iostream>
#include <vector>
#include <list>

class Graph {
private:
    int numVertices; // Number of vertices
    std::vector<std::vector<int>> adjacencyMatrix; // Adjacency matrix representation
    std::vector<std::list<int>> adjacencyList; // Adjacency list representation

public:
    // Constructor
    Graph(int vertices) : numVertices(vertices) {
        // Initialize the adjacency matrix
        adjacencyMatrix.resize(numVertices, std::vector<int>(numVertices, 0));

        // Initialize the adjacency list
        adjacencyList.resize(numVertices);
    }

    // Add an edge to the graph (undirected)
    void addEdge(int u, int v) {
        // Update adjacency matrix
        adjacencyMatrix[u][v] = 1;
        adjacencyMatrix[v][u] = 1; // For undirected graph

        // Update adjacency list
        adjacencyList[u].push_back(v);
        adjacencyList[v].push_back(u); // For undirected graph
    }

    // Display the adjacency matrix
    void displayAdjacencyMatrix() {
        std::cout << "Adjacency Matrix:\n";
        for (int i = 0; i < numVertices; ++i) {
            for (int j = 0; j < numVertices; ++j) {
                std::cout << adjacencyMatrix[i][j] << " ";
            }
            std::cout << std::endl;
```

```cpp
        }
    }

    // Display the adjacency list
    void displayAdjacencyList() {
        std::cout << "Adjacency List:\n";
        for (int i = 0; i < numVertices; ++i) {
            std::cout << "Vertex " << i << ":";
            for (int v : adjacencyList[i]) {
                std::cout << " -> " << v;
            }
            std::cout << std::endl;
        }
    }
};

int main() {
    // Create a graph with 5 vertices
    Graph graph(5);

    // Add edges
    graph.addEdge(0, 1);
    graph.addEdge(0, 4);
    graph.addEdge(1, 2);
    graph.addEdge(1, 3);
    graph.addEdge(1, 4);
    graph.addEdge(2, 3);
    graph.addEdge(3, 4);

    // Display the graph
    graph.displayAdjacencyMatrix();
    graph.displayAdjacencyList();

    return 0;
}
```

Explanation:
Graph Class: This class contains both an adjacency matrix and an adjacency list to represent the graph.

numVertices: Stores the number of vertices in the graph.
adjacencyMatrix: A 2D vector to store the adjacency matrix.
adjacencyList: A vector of lists to store the adjacency list.
Constructor: Initializes the adjacency matrix and list based on the number of vertices.

addEdge Method: Adds an edge between two vertices. It updates both the adjacency matrix and the adjacency list. The graph is undirected, so we update both directions.

displayAdjacencyMatrix Method: Displays the adjacency matrix representation of the graph.

displayAdjacencyList Method: Displays the adjacency list representation of the graph.

Main Function: Creates an instance of the graph, adds edges, and displays both representations.

Usage:
You can compile and run this code in any C++ environment. The output will show the adjacency matrix and adjacency list of the graph created. Adjust the number of vertices and edges as needed to explore different graph structures.

**OUTPUT:**

```
C:\Users\TIPQC\Desktop\MAMARIL_SORTING ALGORITHMS BUBBLE, SELECTION, AND I
Adjacency Matrix:
0 1 0 0 1
1 0 1 1 1
0 1 0 1 0
0 1 1 0 1
1 1 0 1 0
Adjacency List:
Vertex 0: -> 1 -> 4
Vertex 1: -> 0 -> 2 -> 3 -> 4
Vertex 2: -> 1 -> 3
Vertex 3: -> 1 -> 2 -> 4
Vertex 4: -> 0 -> 1 -> 3

--------------------------------
Process exited after 0.02958 seconds with return value 0
Press any key to continue . . .
```

**ILO B: Create C++ code for implementing graph traversal algorithms such as Breadth-First and Depth-First Search**

**B.1. Depth-First Search**

```
#include <string>
#include <vector>
#include <iostream>
#include <set>
#include <map>
#include <stack>
template <typename T>
class Graph;

template <typename T>
struct Edge
{
	size_t src;
	size_t dest;
	T weight;
	// To compare edges, only compare their weights,
	// and not the source/destination vertices
	inline bool operator<(const Edge<T> &e) const
	{
		return this->weight < e.weight;
```

```
	}
	inline bool operator>(const Edge<T> &e) const
	{
		return this->weight > e.weight;
	}
};

template <typename T>
std::ostream &operator<<(std::ostream &os, const Graph<T> &G)
{

	for (auto i = 1; i < G.vertices(); i++)
	{
		os << i << ":\t";
		auto edges = G.outgoing_edges(i);
		for (auto &e : edges)
			os << "{" << e.dest << ": " << e.weight << "}, ";
		os << std::endl;
	}
	return os;
}

template <typename T>
class Graph
{
public:
	// Initialize the graph with N vertices
	Graph(size_t N) : V(N)
	{
	}
	// Return number of vertices in the graph
	auto vertices() const
	{
		return V;
	}
	// Return all edges in the graph
	auto &edges() const
	{
		return edge_list;
	}

	void add_edge(Edge<T> &&e)
	{
		// Check if the source and destination vertices are within range
		if (e.src >= 1 && e.src <= V &&
			e.dest >= 1 && e.dest <= V)
			edge_list.emplace_back(e);
		else
			std::cerr << "Vertex out of bounds" << std::endl;
}
// Returns all outgoing edges from vertex v
```

```
auto outgoing_edges(size_t v) const
{
			std::vector<Edge<T>> edges_from_v;
			for (auto &e : edge_list)
{
					if (e.src == v)
						edges_from_v.emplace_back(e);
		}
		return edges_from_v;
}
// Overloads the << operator so a graph be written directly to a stream
// Can be used as std::cout << obj << std::endl;
template <typename U>
friend std::ostream &operator<<(std::ostream &os, const Graph<U> &G);
private:
		size_t V; // Stores number of vertices in graph
		std::vector<Edge<T>> edge_list;
};

template <typename T>
auto depth_first_search(const Graph<T> &G, size_t dest)
{
		std::stack<size_t> stack;
		std::vector<size_t> visit_order;
		std::set<size_t> visited;
		stack.push(1); // Assume that DFS always starts from vertex ID 1
		while (!stack.empty())
		{
			auto current_vertex = stack.top();
			stack.pop();
			// If the current vertex hasn't been visited in the past
			if (visited.find(current_vertex) == visited.end())
			{
				visited.insert(current_vertex);
				visit_order.push_back(current_vertex);
				for (auto e : G.outgoing_edges(current_vertex))
				{
					// If the vertex hasn't been visited, insert it in the stack.
					if(visited.find(e.dest) == visited.end())
					{
					stack.push(e.dest);
					}
				}
			}
		}
		return visit_order;
}

template <typename T>
auto create_reference_graph()
{
```

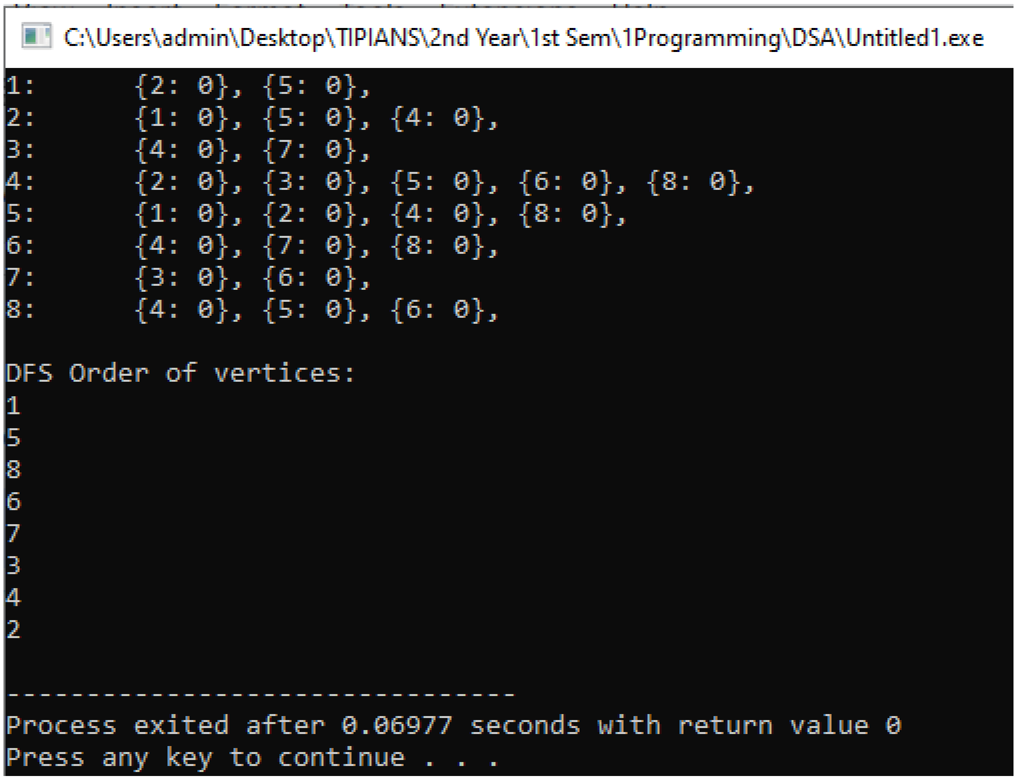
```
	Graph<T> G(9);
	std::map<unsigned, std::vector<std::pair<size_t, T>>> edges;
	edges[1] = {{2, 0}, {5, 0}};
	edges[2] = {{1, 0}, {5, 0}, {4, 0}};
	edges[3] = {{4, 0}, {7, 0}};
	edges[4] = {{2, 0}, {3, 0}, {5, 0}, {6, 0}, {8, 0}};
	edges[5] = {{1, 0}, {2, 0}, {4, 0}, {8, 0}};
	edges[6] = {{4, 0}, {7, 0}, {8, 0}};
	edges[7] = {{3, 0}, {6, 0}};
	edges[8] = {{4, 0}, {5, 0}, {6, 0}};
	for (auto &i : edges)
		for (auto &j : i.second)
			G.add_edge(Edge<T>{i.first, j.first, j.second});
	return G;
}

template <typename T>
void test_DFS()
{
	// Create an instance of and print the graph
	auto G = create_reference_graph<unsigned>();
	std::cout << G << std::endl;
	// Run DFS starting from vertex ID 1 and print the order
	// in which vertices are visited.
	std::cout << "DFS Order of vertices: " << std::endl;
	auto dfs_visit_order = depth_first_search(G, 1);
	for (auto v : dfs_visit_order)
		std::cout << v << std::endl;
}
int main()
{
	using T = unsigned;
	test_DFS<T>();
	return 0;
}
```

**Explanation:**

- The Edge struct represents an edge in the graph with a source vertex (src), a destination vertex (dest), and a weight (weight). It overloads the < and > operators to compare edges based on their weights.
- The Graph class template represents a graph with a specified number of vertices (N). It maintains a vector of Edge objects to store the edges of the graph. It provides methods to add edges, get outgoing edges from a vertex, and retrieve the number of vertices.
- The add_edge method adds an edge to the graph if the source and destination vertices are within valid bounds (1 to V).
- The outgoing_edges returns all outgoing edges from a specified vertex.
- The depth_first_search function implements the DFS algorithm using a stack. It maintains a set of visited vertices to avoid cycles and a vector to record the order of visits.
- The test_DFS function creates a graph, prints it, and performs DFS starting from vertex 1, printing the order of visited vertices.
- The main function serves as the entry point of the program, calling the test_DFS function.

Graph Representation: The graph is represented using an adjacency list via the Edge structure and the Graph class.
DFS Implementation: The DFS algorithm is implemented using a stack to traverse the graph
**Output:**



```
C:\Users\admin\Desktop\TIPIANS\2nd Year\1st Sem\1Programming\DSA\Untitled1.exe
1:      {2: 0}, {5: 0},
2:      {1: 0}, {5: 0}, {4: 0},
3:      {4: 0}, {7: 0},
4:      {2: 0}, {3: 0}, {5: 0}, {6: 0}, {8: 0},
5:      {1: 0}, {2: 0}, {4: 0}, {8: 0},
6:      {4: 0}, {7: 0}, {8: 0},
7:      {3: 0}, {6: 0},
8:      {4: 0}, {5: 0}, {6: 0},

DFS Order of vertices:
1
5
8
6
7
3
4
2

--------------------------------
Process exited after 0.06977 seconds with return value 0
Press any key to continue . . .
```

**Explanation:**
Edge List Interpretation
Vertex 1 is connected to vertices 2 and 5.
Vertex 2 is connected to vertices 1, 4, and 5.
Vertex 3 is connected to vertices 4 and 7.
Vertex 4 is connected to vertices 2, 3, 5, 6, and 8.
Vertex 5 is connected to vertices 1, 2, 4, and 8.
Vertex 6 is connected to vertices 4, 7, and 8.
Vertex 7 is connected to vertices 3 and 6.
Vertex 8 is connected to vertices 4, 5, and 6.

- The overloaded << operator for the Graph class is used to print the graph. The output displays each vertex and its outgoing edges.
- DFS Order of Vertices: After printing the graph, the program performs a depth-first search starting from vertex 1. The order in which vertices are visited during the DFS traversal will be printed. The exact order can vary based on how the edges are stored and accessed, but it will generally follow the depth-first strategy.

Explanation of DFS Order
Starting from vertex 1, it goes to vertex 2 (the first outgoing edge).
From 2, it goes to 4 (the first outgoing edge from 2).
From 4, it can go to 3 (the first outgoing edge from 4).
From 3, it goes to 7 (the only outgoing edge).
From 7, it goes to 6 (the only outgoing edge).
From 6, it can go to 8 (the only outgoing edge).
Finally, it goes back to 4 and checks for other unvisited edges, but all are visited, so it backtracks to 2 and then visits 5.

**B.2. Breadth-First Search**

```
#include <string>
#include <vector>
#include <iostream>
#include <set>
#include <map>
#include <queue>

template <typename T>
class Graph;

template <typename T>
struct Edge
{
    size_t src;
    size_t dest;
    T weight;
    inline bool operator<(const Edge<T> &e) const
    {
        return this->weight < e.weight;
    }
    inline bool operator>(const Edge<T> &e) const
    {
        return this->weight > e.weight;
    }
};

template <typename T>
std::ostream &operator<<(std::ostream &os, const Graph<T> &G)
{
    for (auto i = 1; i <= G.vertices(); i++) // Change < to <=
    {
        os << i << ":\t";

        auto edges = G.outgoing_edges(i);
        for (auto &e : edges)
            os << "{" << e.dest << ": " << e.weight << "}, ";
        os << std::endl;
    }
    return os; // Add this line
}

template <typename T>
class Graph
{
public:
    Graph(size_t N) : V(N) {}

    auto vertices() const
    {
```

```
        return V;
    }

    auto &edges() const
    {
        return edge_list;
    }

    void add_edge(Edge<T> &&e)
    {
        if (e.src >= 1 && e.src <= V &&
            e.dest >= 1 && e.dest <= V)
            edge_list.emplace_back(e);
        else
            std::cerr << "Vertex out of bounds" << std::endl;
    }

    auto outgoing_edges(size_t v) const
    {
        std::vector<Edge<T>> edges_from_v;
        for (auto &e : edge_list)
        {
            if (e.src == v)
            {
                edges_from_v.emplace_back(e);
            }
        }
        return edges_from_v;
    }

    template <typename U> // Changed from T to U to avoid shadowing
    friend std::ostream &operator<<(std::ostream &os, const Graph<U> &G);

private:
    size_t V;
    std::vector<Edge<T>> edge_list;
};

template <typename T>
auto create_reference_graph()
{
            Graph<T> G(9);

            std::map<unsigned, std::vector<std::pair<size_t, T>>> edges;
            edges[1] = {{2, 2}, {5, 3}};
            edges[2] = {{1, 2}, {5, 5}, {4, 1}};
            edges[3] = {{4, 2}, {7, 3}};
            edges[4] = {{2, 1}, {3, 2}, {5, 2}, {6, 4}, {8, 5}};
            edges[5] = {{1, 3}, {2, 5}, {4, 2}, {8, 3}};
            edges[6] = {{4, 4}, {7, 4}, {8, 1}};
            edges[7] = {{3, 3}, {6, 4}};
```

```
	edges[8] = {{4, 5}, {5, 3}, {6, 1}};

	for (auto &i : edges)
		for (auto &j : i.second)
			G.add_edge(Edge<T>{i.first, j.first, j.second});
	return G;
}

template <typename T>
auto breadth_first_search(const Graph<T> &G, size_t dest)
{
	std::queue<size_t> queue;
	std::vector<size_t> visit_order;
	std::set<size_t> visited;
	queue.push(1); // Assume that BFS always starts from vertex ID 1
	while (!queue.empty())
	{
		auto current_vertex = queue.front();
		queue.pop();
		// If the current vertex hasn't been visited in the past
		if (visited.find(current_vertex) == visited.end())
		{
			visited.insert(current_vertex);
			visit_order.push_back(current_vertex);
			for (auto e : G.outgoing_edges(current_vertex))
				queue.push(e.dest);
		}
	}
	return visit_order;
}

template <typename T>
void test_BFS()
{
	// Create an instance of and print the graph
	auto G = create_reference_graph<unsigned>();
	std::cout << G << std::endl;
	// Run BFS starting from vertex ID 1 and print the order
	// in which vertices are visited.
	std::cout << "BFS Order of vertices: " << std::endl;
	auto bfs_visit_order = breadth_first_search(G, 1);
	for (auto v : bfs_visit_order)
		std::cout << v << std::endl;
}

int main()
{
	using T = unsigned;
	test_BFS<T>();
	return 0;
}
```

**Explanation of the code**
The provided code defines a graph data structure in C++ using templates, allowing for the representation of graphs with weighted edges. It includes functionalities to add edges, perform breadth-first search (BFS), and display the graph.

Key Methods in Graph Class
add_edge: This method adds an edge to the graph if the source and destination vertices are within valid bounds.
outgoing_edges: This method retrieves all edges that originate from a given vertex.
vertices: This method returns the total number of vertices in the graph.

Operator Overloading
The operator<< is overloaded to allow for easy printing of the graph's adjacency list format. It prints each vertex followed by its outgoing edges.

Graph Creation
create_reference_graph:
This function creates a specific graph with 9 vertices and predefined edges. It uses a map to define edges and then adds them to the graph using the add_edge method.

Breadth-First Search (BFS)
breadth_first_search:
This function implements the BFS algorithm starting from vertex ID 1. It uses a queue to explore the graph level by level. It maintains a set of visited vertices to avoid processing the same vertex multiple times and records the order in which vertices are visited.

Testing the BFS
test_BFS:
This function creates an instance of the graph, prints it, and performs a BFS starting from vertex 1, printing the order of visited vertices.

Main Function
The main function is the entry point of the program. It calls test_BFS to demonstrate the functionality of the graph and BFS traversal.

Summary
Overall, this code defines a simple directed graph structure and implements BFS to explore it. The graph is represented using an adjacency list format, and the BFS algorithm is designed to visit all reachable vertices from a starting point, demonstrating basic graph traversal techniques. The use of templates allows the graph to work with different data types for edge weights.

**Output:**

C:\Users\admin\Desktop\TIPIANS\2nd Year\1st Sem\1Programming\DSA\Untitled2.exe

```
1:      {2: 2}, {5: 3},
2:      {1: 2}, {5: 5}, {4: 1},
3:      {4: 2}, {7: 3},
4:      {2: 1}, {3: 2}, {5: 2}, {6: 4}, {8: 5},
5:      {1: 3}, {2: 5}, {4: 2}, {8: 3},
6:      {4: 4}, {7: 4}, {8: 1},
7:      {3: 3}, {6: 4},
8:      {4: 5}, {5: 3}, {6: 1},
9:

BFS Order of vertices:
1
2
5
4
8
3
6
7

--------------------------------
Process exited after 0.06584 seconds with return value 0
Press any key to continue . . .
```

**Explanation of the output:**

1. Graph Representation

The create_reference_graph function constructs a graph with 9 vertices and specific edges defined in a map. The edges are directed and have associated weights. The edges are as follows:

Vertex 1 has edges to Vertex 2 (weight 2) and Vertex 5 (weight 3).
Vertex 2 has edges to Vertex 1 (weight 2), Vertex 5 (weight 5), and Vertex 4 (weight 1).
Vertex 3 has edges to Vertex 4 (weight 2) and Vertex 7 (weight 3).
Vertex 4 has edges to Vertex 2 (weight 1), Vertex 3 (weight 2), Vertex 5 (weight 2), Vertex 6 (weight 4), and Vertex 8 (weight 5).
Vertex 5 has edges to Vertex 1 (weight 3), Vertex 2 (weight 5), Vertex 4 (weight 2), and Vertex 8 (weight 3).
Vertex 6 has edges to Vertex 4 (weight 4), Vertex 7 (weight 4), and Vertex 8 (weight 1).
Vertex 7 has edges to Vertex 3 (weight 3) and Vertex 6 (weight 4).
Vertex 8 has edges to Vertex 4 (weight 5), Vertex 5 (weight 3), and Vertex 6 (weight 1).

2. Printing the Graph

The output shows each vertex followed by its outgoing edges, formatted as {destination: weight}.

3. BFS Order of Vertices

Next, the BFS traversal is performed starting from vertex 1. The BFS algorithm explores all vertices reachable from the starting vertex, level by level.

Given the edges, the BFS traversal starting from vertex 1 will visit the vertices in the following order:
Start at 1 (enqueue 2 and 5).
Visit 2 (enqueue 1, 5, and 4). Since 1 and 5 are already queued, only 4 is added.
Visit 5 (enqueue 1, 2, 4, and 8). Again, 1, 2, and 4 are already queued, so only 8 is added.
Visit 4 (enqueue 2, 3, 5, 6, and 8). Here, 2, 5, and 8 are already queued, so only 3 and 6 are added.
Visit 8 (enqueue 4, 5, and 6). All are already queued.
Visit 3 (enqueue 4 and 7). Only 7 are added.
Visit 6 (enqueue 4, 7, and 8). All are already queued.
Visit 7 (enqueue 3 and 6). Both are already queued.

**B. Answers to Supplementary Activity**

1. **A person wants to visit different locations indicated on a map. He starts from one location (vertex) and wants to visit every vertex until it finishes from one vertex, backtracks, and then explore other vertex from same vertex. Discuss which algorithm would be most helpful to accomplish this task.**

**Depth-First Search (DFS**) is the best algorithm for this situation.

**Reasoning**:

- **DFS** is the best algorithm for this scenario because it explores as far as possible down each branch before backtracking. This means the person will visit all possible paths from a starting location before moving back to previous locations.
- The behavior of **backtracking** is inherent in DFS, making it suitable when a person wants to explore a path fully before checking alternative routes.

**Imagine a graph where:**

- Vertex 1 is connected to vertices 2 and 3.
- Vertex 2 is connected to vertex 4.
- Vertex 3 is connected to vertex 5.
- Vertex 4 and vertex 5 have no other connections.

Using DFS, the traversal might look like this: 1 → 2 → 4 (backtrack) → 3 → 5.

2. **Identify the equivalent of DFS in traversal strategies for trees. To efficiently answer this question, provide a graphical comparison, examine pseudocode and code implementation.**

**Tree Traversal Equivalent:**

The equivalent traversal strategy of **DFS in trees is Preorder Traversal.**

**Graphical Comparison:**

- **Graph DFS:**
  - Traverses vertices by visiting deeply along each branch before moving to another.
- **Tree Preorder Traversal:**
  - Visits the root node first, then recursively visits the left subtree, followed by the right subtree.

**Pseudocode for Preorder Traversal (DFS Equivalent):**

```
PreorderTraversal(node):
  if node is NULL:
    return
  visit(node)
  PreorderTraversal(node.left)
  PreorderTraversal(node.right)
```

**C++ Code Implementation**:

main.cpp

```cpp
#include <iostream>
using namespace std;

struct Node {
    char data;
    Node *left, *right;
    Node(char val) : data(val), left(nullptr), right(nullptr) {}
};

// Preorder Traversal
void Preorder(Node* node) {
    if (node == nullptr) return;
    cout << node->data << " ";
    Preorder(node->left);
    Preorder(node->right);
}

// Inorder Traversal
void Inorder(Node* node) {
    if (node == nullptr) return;
    Inorder(node->left);
    cout << node->data << " ";
    Inorder(node->right);
}

// Postorder Traversal
void Postorder(Node* node) {
    if (node == nullptr) return;
    Postorder(node->left);
    Postorder(node->right);
    cout << node->data << " ";
}

int main() {
    Node* root = new Node('A');
    root->left = new Node('B');
    root->right = new Node('C');
    root->left->left = new Node('D');
    root->left->right = new Node('E');

    cout << "Preorder Traversal: ";
    Preorder(root); // Output: A B D E C
    cout << "\nInorder Traversal: ";
    Inorder(root); // Output: D B E A C
    cout << "\nPostorder Traversal: ";
    Postorder(root); // Output: D E B C A
    cout << endl;
    return 0;
}
```

Output

```
Preorder Traversal: A B D E C
Inorder Traversal: D B E A C
Postorder Traversal: D E B C A


=== Code Execution Successful ===
```

**3. In the performed code, what data structure is used to implement the Breadth First Search?**

In the performed code, **Breadth-First Search (BFS)** uses a **queue** data structure to manage the nodes.

- **Queue** works on a **First In, First Out (FIFO)** principle, making it suitable for BFS, as it explores nodes level by level.

**Reasoning:**

- In BFS, nodes are enqueued as they are discovered and dequeued when they are visited. This ensures nodes closer to the starting vertex are explored first before moving deeper into the graph.

**Example from BFS Code**:

```
queue<size_t> queue;  // Using a queue to implement BFS
```

**4. How many times can a node be visited in the BFS?**

In BFS, each node is visited exactly once.

- Once a node is dequeued and marked as visited, it is not revisited, preventing infinite loops and ensuring efficient traversal.

- The visited set keeps track of already visited nodes.

**Code Explanation:**

```
set<size_t> visited;

if (visited.find(current_vertex) == visited.end()) {

    visited.insert(current_vertex); // Mark the node as visited

}
```

**C. Conclusion & Lessons Learned**

In this activity, we deepened our understanding of graph theory, exploring key concepts such as vertices, edges, and graph traversal algorithms like Depth-First Search (DFS) and Breadth-First Search (BFS). We learned how to implement these algorithms effectively to navigate and analyze graph structures, gaining insights into their practical applications in fields such as network analysis and pathfinding. The procedure allowed us to practice constructing graphs, both as adjacency matrices and adjacency lists. By implementing DFS and BFS, we observed firsthand the differences in their traversal methods. DFS uses a stack-based approach that goes deep into the graph before backtracking, while BFS uses a queue-based approach, exploring level by level. The activity reinforced our understanding of these traversal techniques and their suitability for different types of graph problems. The supplementary activity provided additional challenges that required us to apply the learned traversal algorithms to solve real-world problems, such as detecting cycles in graphs or finding the shortest path. This exercise highlighted the importance of choosing the appropriate data structures for different graph representations and helped us practice optimizing our code for better performance .Overall, I believe we performed well in this activity, successfully implementing and understanding the core concepts of graph theory and traversal algorithms. However, there's room for improvement in optimizing our code for larger graphs and exploring more advanced graph algorithms, such as Dijkstra's and Prim's algorithms, for solving complex problems. This activity has set a strong foundation, and I look forward to further exploring the applications of graph theory in computer engineering.

**D. Assessment Rubric**

**E. External References**


- https://www.w3schools.com/dsa/dsa_theory_graphs.php
- https://www.simplilearn.com/tutorials/data-structure-tutorial/graphs-in-data-structure
- https://www.tutorialspoint.com/data_structures_algorithms/graph_data_structure.htm