Name-Abhi Mehta

Reg no.-221080001

Experiment-10

Aim: To write a Java code for Implementing Graph Data structure and Graph ADT.

Theory: Graphs are versatile data structures used to model relationships between entities. The representation of a graph can significantly impact the efficiency of various operations performed on it. Here, we discuss four representations of a graph: Edge List, Adjacency List, Adjacency Map, and Adjacency Matrix, along with the corresponding methods in the Graph ADT.

1. Edge List:

• Description: In the edge list representation, all edges are stored in an unordered list. This simple representation allows for easy addition and removal of edges.

• Pros: Easy to implement, suitable for sparse graphs.

• Cons: Inefficient for finding specific edges or edges incident to a vertex.

2. Adjacency List:

• Description: In the adjacency list representation, each vertex maintains a separate list containing edges incident to it. This allows for efficient retrieval of all edges incident to a given vertex.

• Pros: Space-efficient for sparse graphs, efficient for finding incident edges for a vertex.

• Cons: Less efficient for dense graphs.

3. Adjacency Map:

• Description: Similar to the adjacency list, but the secondary container for edges incident to a vertex is organized as a map, where the adjacent vertex serves as the key. This allows for O(1) expected time access to a specific edge through hashing.

• Pros: Efficient access to specific edges, especially for sparse graphs.

• Cons: Slightly more memory overhead due to hashing.

4. Adjacency Matrix:

• Description: An adjacency matrix is implemented as an n × n matrix (for a graph with n vertices), where each slot stores a reference to the edge (u, v) for a particular pair of vertices. If no such edge exists, the slot stores null.

• Pros: Provides O(1) worst-case access to specific edges. Suitable for dense graphs.

• Cons: Inefficient for sparse graphs due to increased memory usage.

Graph ADT Methods:

numVertices(): Returns number of vertices of the graph

vertices(): Returns an iterator of all vertices of the graph

numEdges(): Returns number of edges of the graph

edges(): Returns an iterator of all edges of the graph

getEdge(u,v): returns the edge from vertex u to vertex v

endVertices(e): Returns an array of two endpoint vertices of edge e

opposite(v,e): For edge e incident to vertex v, return other incident vertex

outDegree(v): Returns number of outgoing edges incident on v

inDegree(v): Returns number of incoming edges incident on v

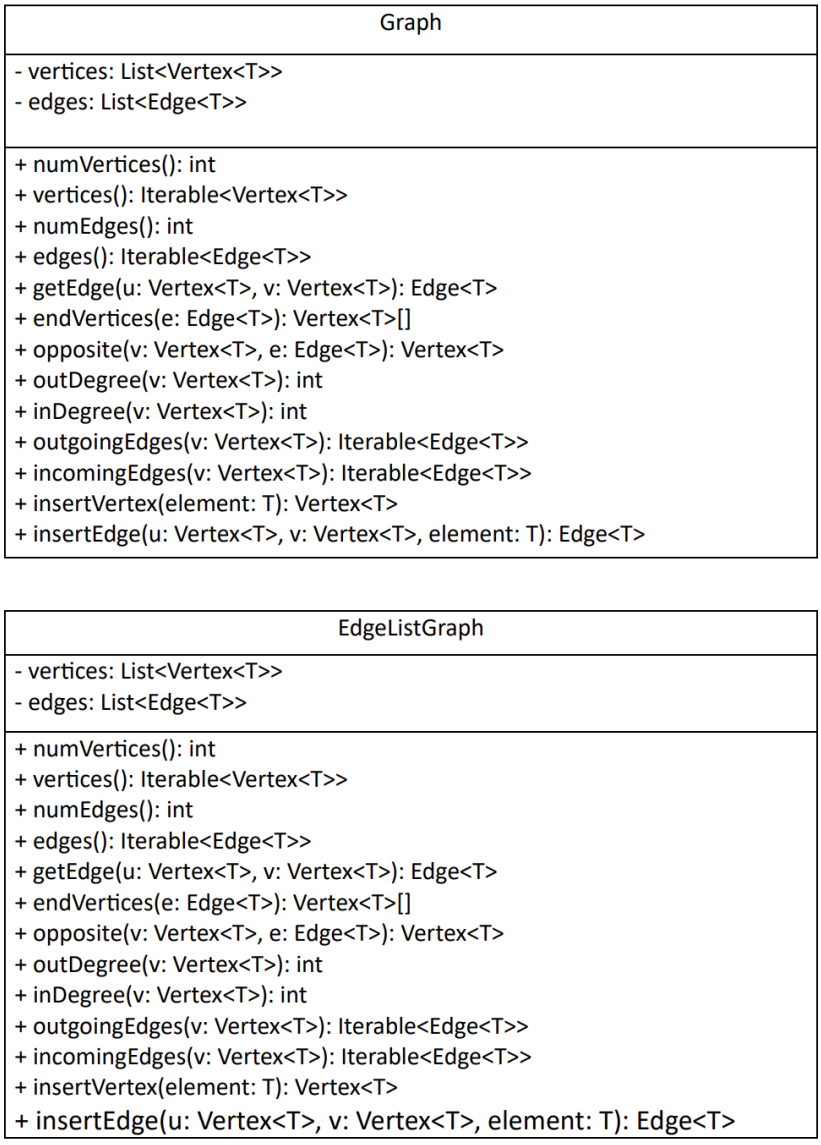
outGoingEdges(v) : Returns an iterator of outgoing edges incident on v

InComingEdges(v): Returns an iterator of incoming edges incident on v

insetVertex(x): Create and return a new Vertex storing element

insertEdge(u, v, x) : Create and return a new edge x from u to v

Class Diagram:



Code:

```
// Vertex class to store element\
import java.util.*;
class Vertex<T> {
    T element;

    Vertex(T element) {
        this.element = element;
```

```
    }

    public T getElement() {
        return element;
    }
}

class Edge<T> {

    private Vertex<T> u;
    private Vertex<T> v;
    private T element;

    public Edge(Vertex<T> u, Vertex<T> v, T element) {
        this.u = u;
        this.v = v;
        this.element = element;
    }

    public boolean hasVertices(Vertex<T> vertex1, Vertex<T> vertex2) {
        return (u.equals(vertex1) && v.equals(vertex2)) || (u.equals(vertex2) && v.equals(vertex1));
    }

    public Vertex<T>[] getVertices() {
        return new Vertex[]{u, v};
    }

    public Vertex<T> opposite(Vertex<T> vertex) {
        if (vertex.equals(u)) {
            return v;
        } else if (vertex.equals(v)) {
            return u;
        } else {
            return null;
        }
    }

    public Vertex<T> getSource() {
        return u;
    }

    public Vertex<T> getTarget() {
        return v;
    }
```

```
    public T getElement() {
        return element;
    }

    // Other methods and fields as needed
}


//Graph interface
interface Graph<T> {

    // Get the number of vertices in the graph
    int numVertices();

    // Get an iterable collection of all vertices in the graph
    Iterable<Vertex<T>> vertices();

    // Get the number of edges in the graph
    int numEdges();

    // Get an iterable collection of all edges in the graph
    Iterable<Edge<T>> edges();

    // Get the edge between two vertices
    Edge<T> getEdge(Vertex<T> u, Vertex<T> v);

    // Get the vertices at the ends of a given edge
    Vertex<T>[] endVertices(Edge<T> e);

    // Get the opposite vertex of a given vertex along a given edge
    Vertex<T> opposite(Vertex<T> v, Edge<T> e);

    // Get the out-degree of a given vertex
    int outDegree(Vertex<T> v);

    // Get the in-degree of a given vertex
    int inDegree(Vertex<T> v);

    // Get an iterable collection of outgoing edges from a given vertex
    Iterable<Edge<T>> outgoingEdges(Vertex<T> v);

    // Get an iterable collection of incoming edges to a given vertex
    Iterable<Edge<T>> incomingEdges(Vertex<T> v);
```

```
    // Insert a new vertex with the given element into the graph
    Vertex<T> insertVertex(T element);

    // Insert a new edge between two vertices with the given element into the graph
    Edge<T> insertEdge(Vertex<T> u, Vertex<T> v, T element);
}

// import java.util.*;

class EdgeListGraph<T> implements Graph<T> {

    private final List<Vertex<T>> vertices;
    private final List<Edge<T>> edges;

    public EdgeListGraph() {
        vertices = new ArrayList<>();
        edges = new ArrayList<>();
    }

    @Override
    public int numVertices() {
        return vertices.size();
    }

    @Override
    public Iterable<Vertex<T>> vertices() {
        return vertices;
    }

    @Override
    public int numEdges() {
        return edges.size();
    }

    @Override
    public Iterable<Edge<T>> edges() {
        return edges;
    }

    @Override
    public Edge<T> getEdge(Vertex<T> u, Vertex<T> v) {
        for (Edge<T> edge : edges) {
            if ((edge.hasVertices(u, v))) {
```

```
            return edge;
        }
    }
    return null;
}

@Override
public Vertex<T>[] endVertices(Edge<T> e) {
    return e.getVertices();
}

@Override
public Vertex<T> opposite(Vertex<T> v, Edge<T> e) {
    return e.opposite(v);
}

@Override
public int outDegree(Vertex<T> v) {
    return (int) edges.stream().filter(edge -> edge.getSource() == v).count();
}

@Override
public int inDegree(Vertex<T> v) {
    return (int) edges.stream().filter(edge -> edge.getTarget() == v).count();
}

@Override
public Iterable<Edge<T>> outgoingEdges(Vertex<T> v) {
    List<Edge<T>> outgoing = new ArrayList<>();
    edges.stream().filter(edge -> edge.getSource() == v).forEach(outgoing::add);
    return outgoing;
}

@Override
public Iterable<Edge<T>> incomingEdges(Vertex<T> v) {
    List<Edge<T>> incoming = new ArrayList<>();
    edges.stream().filter(edge -> edge.getTarget() == v).forEach(incoming::add);
    return incoming;
}

@Override
public Vertex<T> insertVertex(T element) {
    Vertex<T> v = new Vertex<>(element);
    vertices.add(v);
```

```
        return v;
    }

    @Override
    public Edge<T> insertEdge(Vertex<T> u, Vertex<T> v, T element) {
        Edge<T> e = new Edge<>(u, v, element);
        edges.add(e);
        return e;
    }
}

//TestInterface

import java.util.*;
public class TestInterface    {
    public static void main(String[] args) {
        EdgeListGraph<String> graph = new EdgeListGraph<>();
        Scanner scanner = new Scanner(System.in);

        int choice;
        do {
            System.out.println("Graph Operations:");
            System.out.println("1. Add Vertex");
            System.out.println("2. Add Edge");
            System.out.println("3. Display Vertices");
            System.out.println("4. Display Edges");
            System.out.println("5. Get Edge");
            System.out.println("6. End Vertices");
            System.out.println("7. Opposite");
            System.out.println("8. Out Degree");
            System.out.println("9. In Degree");
            System.out.println("10. Outgoing Edges");
            System.out.println("11. Incoming Edges");
            System.out.println("0. Exit");
            System.out.print("Enter your choice: ");
            choice = scanner.nextInt();
            scanner.nextLine(); // Consume the newline character

            switch (choice) {
                case 1:
                    System.out.print("Enter vertex element: ");
                    String vertexElement = scanner.nextLine();
                    graph.insertVertex(vertexElement);
                    System.out.println("Vertex added.");
```

```
            break;
        case 2:
            System.out.print("Enter source vertex element: ");
            String sourceElement = scanner.nextLine();
            System.out.print("Enter target vertex element: ");
            String targetElement = scanner.nextLine();
            System.out.print("Enter edge element: ");
            String edgeElement = scanner.nextLine();

            Vertex<String> sourceVertex = findVertex(graph, sourceElement);
            Vertex<String> targetVertex = findVertex(graph, targetElement);

            if (sourceVertex != null && targetVertex != null) {
                graph.insertEdge(sourceVertex, targetVertex, edgeElement);
                System.out.println("Edge added.");
            } else {
                System.out.println("Invalid vertices. Edge not added.");
            }
            break;
        case 3:
            displayVertices(graph);
            break;
        case 4:
            displayEdges(graph);
            break;
        case 5:
            getEdge(graph, scanner);
            break;
        case 6:
            endVertices(graph, scanner);
            break;
        case 7:
            opposite(graph, scanner);
            break;
        case 8:
            outDegree(graph, scanner);
            break;
        case 9:
            inDegree(graph, scanner);
            break;
        case 10:
            outgoingEdges(graph, scanner);
            break;
        case 11:
```

```
                incomingEdges(graph, scanner);
                break;
            case 0:
                System.out.println("Exiting...");
                break;
            default:
                System.out.println("Invalid choice. Please try again.");
        }
    } while (choice != 0);
}

private static Vertex<String> findVertex(EdgeListGraph<String> graph, String element) {
    for (Vertex<String> vertex : graph.vertices()) {
        if (vertex.getElement().equals(element)) {
            return vertex;
        }
    }
    return null;
}

private static void displayVertices(EdgeListGraph<String> graph) {
    System.out.println("Vertices:");
    for (Vertex<String> vertex : graph.vertices()) {
        System.out.println(vertex.getElement());
    }
}

private static void displayEdges(EdgeListGraph<String> graph) {
    System.out.println("Edges:");
    for (Edge<String> edge : graph.edges()) {
        System.out.println(edge.getVertices()[0].getElement() + " -- " +
                edge.getElement() + " -- " +
                edge.getVertices()[1].getElement());
    }
}

private static void getEdge(EdgeListGraph<String> graph, Scanner scanner) {
    System.out.print("Enter source vertex element: ");
    String sourceElement = scanner.nextLine();
    System.out.print("Enter target vertex element: ");
    String targetElement = scanner.nextLine();

    Vertex<String> sourceVertex = findVertex(graph, sourceElement);
    Vertex<String> targetVertex = findVertex(graph, targetElement);
```

```
        if (sourceVertex != null && targetVertex != null) {
            Edge<String> edge = graph.getEdge(sourceVertex, targetVertex);
            if (edge != null) {
                System.out.println("Edge found: " + edge.getElement());
            } else {
                System.out.println("Edge not found between the specified vertices.");
            }
        } else {
            System.out.println("Invalid vertices.");
        }
    }

    private static void endVertices(EdgeListGraph<String> graph, Scanner scanner) {
        System.out.print("Enter source vertex element: ");
        String sourceElement = scanner.nextLine();
        System.out.print("Enter target vertex element: ");
        String targetElement = scanner.nextLine();

        Vertex<String> sourceVertex = findVertex(graph, sourceElement);
        Vertex<String> targetVertex = findVertex(graph, targetElement);

        if (sourceVertex != null && targetVertex != null) {
            Edge<String> edge = graph.getEdge(sourceVertex, targetVertex);
            if (edge != null) {
                Vertex<String>[] endVertices = graph.endVertices(edge);
                System.out.println("End Vertices: " + endVertices[0].getElement() + ", " +
endVertices[1].getElement());
            } else {
                System.out.println("Edge not found between the specified vertices.");
            }
        } else {
            System.out.println("Invalid vertices.");
        }
    }

    private static void opposite(EdgeListGraph<String> graph, Scanner scanner) {
        System.out.print("Enter vertex element: ");
        String vertexElement = scanner.nextLine();
        Vertex<String> vertex = findVertex(graph, vertexElement);

        if (vertex != null) {
            System.out.print("Enter source vertex element: ");
            String sourceElement = scanner.nextLine();
```

```
        System.out.print("Enter target vertex element: ");
        String targetElement = scanner.nextLine();

        Vertex<String> sourceVertex = findVertex(graph, sourceElement);
        Vertex<String> targetVertex = findVertex(graph, targetElement);

        if (sourceVertex != null && targetVertex != null) {
            Edge<String> edge = graph.getEdge(sourceVertex, targetVertex);
            if (edge != null) {
                Vertex<String> oppositeVertex = graph.opposite(vertex, edge);
                if (oppositeVertex != null) {
                    System.out.println("Opposite Vertex: " + oppositeVertex.getElement());
                } else {
                    System.out.println("Specified vertex is not an end vertex of the edge.");
                }
            } else {
                System.out.println("Edge not found between the specified vertices.");
            }
        } else {
            System.out.println("Invalid vertices.");
        }
    } else {
        System.out.println("Invalid vertex.");
    }
}

private static void outDegree(EdgeListGraph<String> graph, Scanner scanner) {
    System.out.print("Enter vertex element: ");
    String vertexElement = scanner.nextLine();
    Vertex<String> vertex = findVertex(graph, vertexElement);

    if (vertex != null) {
        int outDegree = graph.outDegree(vertex);
        System.out.println("Out Degree of " + vertex.getElement() + ": " + outDegree);
    } else {
        System.out.println("Invalid vertex.");
    }
}

private static void inDegree(EdgeListGraph<String> graph, Scanner scanner) {
    System.out.print("Enter vertex element: ");
    String vertexElement = scanner.nextLine();
    Vertex<String> vertex = findVertex(graph, vertexElement);
```

```
        if (vertex != null) {
            int inDegree = graph.inDegree(vertex);
            System.out.println("In Degree of " + vertex.getElement() + ": " + inDegree);
        } else {
            System.out.println("Invalid vertex.");
        }
    }

    private static void outgoingEdges(EdgeListGraph<String> graph, Scanner scanner) {
        System.out.print("Enter vertex element: ");
        String vertexElement = scanner.nextLine();
        Vertex<String> vertex = findVertex(graph, vertexElement);

        if (vertex != null) {
            System.out.println("Outgoing Edges of " + vertex.getElement() + ":");
            for (Edge<String> edge : graph.outgoingEdges(vertex)) {
                System.out.println(edge.getVertices()[0].getElement() + " -- " +
                        edge.getElement() + " -- " +
                        edge.getVertices()[1].getElement());
            }
        } else {
            System.out.println("Invalid vertex.");
        }
    }

    private static void incomingEdges(EdgeListGraph<String> graph, Scanner scanner) {
        System.out.print("Enter vertex element: ");
        String vertexElement = scanner.nextLine();
        Vertex<String> vertex = findVertex(graph, vertexElement);

        if (vertex != null) {
            System.out.println("Incoming Edges of " + vertex.getElement() + ":");
            for (Edge<String> edge : graph.incomingEdges(vertex)) {
                System.out.println(edge.getVertices()[0].getElement() + " -- " +
                        edge.getElement() + " -- " +
                        edge.getVertices()[1].getElement());
            }
        } else {
            System.out.println("Invalid vertex.");
        }
    }
}
```

Output:

```
Graph Operations:
1. Add Vertex
2. Add Edge
3. Display Vertices
4. Display Edges
5. Get Edge
6. End Vertices
7. Opposite
8. Out Degree
9. In Degree
10. Outgoing Edges
11. Incoming Edges
0. Exit
Enter your choice: 1
Enter vertex element: 1
Vertex added.
Graph Operations:
1. Add Vertex
2. Add Edge
3. Display Vertices
4. Display Edges
5. Get Edge
6. End Vertices
7. Opposite
8. Out Degree
9. In Degree
10. Outgoing Edges
11. Incoming Edges
0. Exit
Enter your choice: 1
Enter vertex element: 2
Vertex added.
Graph Operations:
1. Add Vertex
2. Add Edge
3. Display Vertices
4. Display Edges
5. Get Edge
6. End Vertices
7. Opposite
8. Out Degree
9. In Degree
```

```
10. Outgoing Edges
11. Incoming Edges
0. Exit
Enter your choice: 2
Enter source vertex element: 1
Enter target vertex element: 2
Enter edge element: a
Edge added.
Graph Operations:
1. Add Vertex
2. Add Edge
3. Display Vertices
4. Display Edges
5. Get Edge
6. End Vertices
7. Opposite
8. Out Degree
9. In Degree
10. Outgoing Edges
11. Incoming Edges
0. Exit
Enter your choice: 3
Vertices:
1
2
Graph Operations:
1. Add Vertex
2. Add Edge
3. Display Vertices
4. Display Edges
5. Get Edge
6. End Vertices
7. Opposite
8. Out Degree
9. In Degree
10. Outgoing Edges
11. Incoming Edges
0. Exit
Enter your choice: 4
Edges:
1 -- a -- 2
Graph Operations:
1. Add Vertex
2. Add Edge
```

```
3. Display Vertices
4. Display Edges
5. Get Edge
6. End Vertices
7. Opposite
8. Out Degree
9. In Degree
10. Outgoing Edges
11. Incoming Edges
0. Exit
Enter your choice: 6
Enter source vertex element: 1
Enter target vertex element: 2
End Vertices: 1, 2
Graph Operations:
1. Add Vertex
2. Add Edge
3. Display Vertices
4. Display Edges
5. Get Edge
6. End Vertices
7. Opposite
8. Out Degree
9. In Degree
10. Outgoing Edges
11. Incoming Edges
0. Exit
Enter your choice: 8
Enter vertex element: 1
Out Degree of 1: 1
Graph Operations:
1. Add Vertex
2. Add Edge
3. Display Vertices
4. Display Edges
5. Get Edge
6. End Vertices
7. Opposite
8. Out Degree
9. In Degree
10. Outgoing Edges
11. Incoming Edges
0. Exit
Enter your choice: 0
```

Errors Encountered:

Lack of Exception Handling:

- Issue: The absence of adequate exception handling for user input may result in runtime exceptions when unexpected inputs are encountered.
- Resolution: Incorporate try-catch blocks to gracefully manage errors, particularly when anticipating specific input types such as integers, ensuring a more robust program.

Unclear Separation of Graph Implementations:

- Problem: The code refers to various graph representations (e.g., AdjacencyListGraph, AdjacencyMapGraph, and AdjacencyMatrixGraph) without distinct implementations for each.
- Solution: Create separate classes or modules for each graph representation to enhance modularity and flexibility. This approach allows users to choose their preferred graph representation without impacting the overall code structure.

Absence of Garbage Collection Handling:

- Issue: The program does not explicitly manage garbage collection, potentially leading to memory leaks and inefficient resource utilization, especially with large graphs.
- Resolution: Implement effective garbage collection mechanisms to handle memory and release resources appropriately. This is crucial for optimizing program efficiency, particularly in scenarios involving extensive data structures like graphs. Utilize language-specific features or manual cleanup routines to ensure proper resource disposal.

Learnings:

Based on the provided Java code for implementing a Graph Data structure and Graph ADT, here are some key learnings:

Graph Representations Impact Efficiency:

Insight: The choice of graph representation (Edge List, Adjacency List, Adjacency Map, Adjacency Matrix) depends on the characteristics of the graph (sparse or dense). Each representation has its pros and cons, influencing the efficiency of various operations.

Edge List:

Learning: Easy addition and removal of edges make the Edge List representation suitable for sparse graphs. However, it may be inefficient for finding specific edges or edges incident to a vertex.

Adjacency List:

Learning: Efficient for sparse graphs, the Adjacency List representation organizes edges incident to each vertex separately. This facilitates the retrieval of all edges incident to a given vertex. However, it might be less efficient for dense graphs.

Adjacency Map:

Learning: Similar to the Adjacency List but with a map, the Adjacency Map allows for O(1) expected time access to specific edges through hashing. This improves efficiency for sparse graphs, but there's slightly more memory overhead due to hashing.

Adjacency Matrix:

Learning: Implemented as an n × n matrix, the Adjacency Matrix provides O(1) worst-case access to specific edges and is suitable for dense graphs. However, it becomes inefficient for sparse graphs due to increased memory usage.

Graph ADT Methods:

Learning: The Graph ADT provides essential methods for interacting with the graph, including retrieving information about vertices and edges, accessing specific edges, determining the degree of a vertex, and managing vertices and edges.

Efficiency vs. Memory Trade-off:

Learning: The choice of graph representation involves a trade-off between efficiency and memory usage. Depending on the application and characteristics of the graph, developers must choose the representation that best fits their requirements.

Versatility of Graphs:

Learning: Graphs are versatile data structures capable of modeling diverse relationships. The provided ADT methods offer a comprehensive set of tools for working with graphs, making them applicable to a wide range of real-world problems.

Understanding Edge and Vertex Relationships:

Learning: The ADT methods emphasize understanding the relationships between edges and vertices, allowing for efficient traversal, retrieval, and manipulation of graph elements.

Dynamic Graph Creation:

Learning: The ADT includes methods for dynamically creating vertices and edges, providing flexibility for adapting the graph structure during runtime. This dynamic aspect enhances the versatility of the graph data structure.

Conclusion:

In summary, the Java implementation of a Graph Data structure and Graph ADT highlights the importance of thoughtful representation choices, with each method offering insights into the efficiency-memory trade-offs. The versatility of graphs becomes evident, providing a flexible framework for modeling diverse relationships. The balance between representation efficiency and memory usage underscores the need for selecting the most suitable approach based on the specific characteristics of the graph and application requirements. This implementation serves as a foundational guide for developers, offering essential tools for creating, manipulating, and analyzing graphs in real-world scenarios.