Advanced Object-Oriented Programming

Lab Assignment 1: Breadth First Search (BFS)

Queues

Learning Objectives

By completing this lab assignment, you will:

- Understand and implement the Breadth First Search (BFS) algorithm for graph traversal
- Design an object-oriented representation of graphs using adjacency lists
- Apply encapsulation principles by keeping data structures private and exposing behavior through public methods
- Implement proper exception handling for invalid inputs
- Develop comprehensive test cases to validate algorithm correctness

Background

What is a Graph?

A graph is a fundamental data structure in computer science that models relationships between objects. It consists of:

- **Vertices (or Nodes):** Represent entities such as cities, people, web pages, or any other objects.
- **Edges:** Represent connections or relationships between these entities. In an undirected graph, edges have no direction (e.g., friendship relationships). In a directed graph, edges have a direction (e.g., following someone on social media).

Breadth First Search (BFS)

Breadth First Search is a graph traversal algorithm that explores the graph level by level. Starting from a source vertex, BFS:

- Visits the source vertex
- Explores all neighbors of the source vertex (vertices at distance 1)
- Then explores all vertices at distance 2 from the source
- Continues this pattern until all reachable vertices are visited

Key Characteristics:

- **Data Structure:** Uses a queue (FIFO - First In First Out) to maintain the order of vertices to visit

- **Time Complexity:** O(V + E) where V is the number of vertices and E is the number of edges
- **Space Complexity:** O(V) for the queue and visited array
- **Applications:** Shortest path in unweighted graphs, finding connected components, web crawlers, social network analysis

Problem Statement

You are given an undirected graph with N vertices labeled from 0 to N-1. Your task is to:

- Design an object-oriented representation of the graph using an adjacency list
- Implement the Breadth First Search (BFS) algorithm starting from a given source vertex
- Print the order in which vertices are visited during the traversal
- Handle edge cases and validate inputs properly

Design Requirements

You must follow these object-oriented design principles:

1. Graph Representation

- Create a Graph class that encapsulates all graph-related functionality
- Use an adjacency list representation (ArrayList of ArrayLists) to store the graph
- Do NOT use static methods for BFS - make it an instance method of the Graph class

2. Encapsulation

- All data structures (adjacency list, visited array, queue) must be private
- Provide public methods for adding edges and performing BFS
- Use getter methods if external access to graph properties is needed

3. Constructor Design

- The Graph constructor should accept the number of vertices as a parameter
- Initialize the adjacency list with empty lists for each vertex
- Validate that the number of vertices is positive

Implementation Tasks

Task 1: Implement the Graph Class

Create a file named Graph.java with the following structure:

- **Constructor:** Graph(int numVertices) - Initializes the graph with the specified number of vertices
- **Method:** void addEdge(int v1, int v2) - Adds an undirected edge between vertices v1 and v2
- **Method:** void bfs(int startVertex) - Performs BFS starting from startVertex and prints the traversal order

Implementation Guidelines:

- Use java.util.ArrayList for the adjacency list
- Use java.util.LinkedList as a Queue for BFS
- Use a boolean array to track visited vertices
- Print each vertex as it is visited, separated by spaces

Task 2: Implement BFS Algorithm

The BFS algorithm should follow these steps:

- Create a boolean array to track visited vertices, initially all false
- Create a queue and enqueue the starting vertex
- Mark the starting vertex as visited
- While the queue is not empty:

a. Dequeue a vertex and print it
b. For each unvisited neighbor of this vertex:
  i. Mark it as visited
  ii. Enqueue it

Task 3: Create Driver Class (BFSTest)

Create a separate class BFSTest.java with a main method that:

- Creates a graph with an appropriate number of vertices
- Adds edges to construct the graph
- Calls the bfs() method with different starting vertices
- Tests edge cases (invalid vertices, disconnected graphs, etc.)

Task 4: Input Validation and Exception Handling

Enhance your Graph class to handle invalid inputs gracefully:

- Throw IllegalArgumentException if the number of vertices in the constructor is less than 1
- Throw IllegalArgumentException in addEdge() if either vertex is invalid (negative or >= numVertices)
- Throw IllegalArgumentException in bfs() if startVertex is invalid
- Include meaningful error messages in all exceptions

Expected Behavior and Examples

Example 1: Simple Connected Graph

Consider the following undirected graph:

```
0 -- 1
|    |
2 -- 3
```

Code:

```
Graph g = new Graph(4);
g.addEdge(0, 1);
g.addEdge(0, 2);
g.addEdge(1, 3);
g.addEdge(2, 3);
g.bfs(0);
```

Expected Output:

```
0 1 2 3
```

Explanation: Starting from vertex 0, BFS first visits 0, then its neighbors 1 and 2 (at distance 1), and finally vertex 3 (at distance 2).

Example 2: Linear Graph

Consider the following chain:

```
0 -- 1 -- 2 -- 3 -- 4
```

Code:

```
Graph g = new Graph(5);
g.addEdge(0, 1);
g.addEdge(1, 2);
g.addEdge(2, 3);
g.addEdge(3, 4);
g.bfs(2);
```

Expected Output:

```
2 1 3 0 4
```

Example 3: Disconnected Graph

Consider the following disconnected graph:

```
0 -- 1      3 -- 4
|           |
2           5
```

Starting BFS from vertex 0:

Expected Output:

0 1 2

Note: BFS only visits vertices reachable from the starting vertex. Vertices 3, 4, and 5 are not visited because they are in a separate connected component.

Test Cases

Your BFSTest class should include comprehensive test cases:

- **Normal Case:** Test with the graph shown in Example 1
- **Single Vertex:** Test a graph with only one vertex
- **Disconnected Graph:** Test BFS on a disconnected graph (Example 3)
- **Complete Graph:** Test a graph where every vertex is connected to every other vertex
- **Exception Handling:** Test invalid vertex numbers (negative, out of bounds)
- **Different Starting Points:** Test BFS from different vertices on the same graph

Conceptual Questions

Answer the following questions in comments within your code or in a separate text file:

- **Why does BFS require a queue instead of a stack?** What would happen if we used a stack instead? What traversal pattern would that produce?
- **What happens if we don't track visited vertices?** Explain the consequences with an example of a graph with a cycle.
- **How would BFS change if the graph was directed?** Would the algorithm logic need to be modified? Would addEdge() need changes?
- **Where is encapsulation used in your design?** Identify at least three examples of encapsulation in your Graph class and explain why each is important.
- **What is the time complexity of your BFS implementation?** Explain in terms of V (vertices) and E (edges). What is the space complexity?

Submission Guidelines

- Create a folder named Lab1_YourRollNumber (e.g., Lab1_2024001)
- Include the following files:
  a. Graph.java - Your Graph class implementation
  b. BFSTest.java - Your test driver class

Academic Integrity

This is an individual assignment. You may discuss concepts with classmates, but **all code must be your own work**. Any instances of copying code from online sources, other students, or AI tools (unless explicitly permitted for CLO7) will result in an F grade for the course as per the Unfair Means Policy.

Good luck and happy coding!