At this lab section, we will learn how to implement hash tables in Java using OOP principles.

# GRAPH TRAVERSAL ALGORITHMS

Asst. Prof. Dr. Feriştah DALKILIÇ
Res. Asst. Fatih DİCLE

## PART 1 – Graph Representation Schemes

Two common implementations of the ADT graph use either an array or a list to represent the graph's edges. The array is typically a two-dimensional array called an adjacency matrix. The list is called an adjacency list. Each of these constructs represents the connections—that is, the edges—among the vertices in the graph.

### The Adjacency Matrix

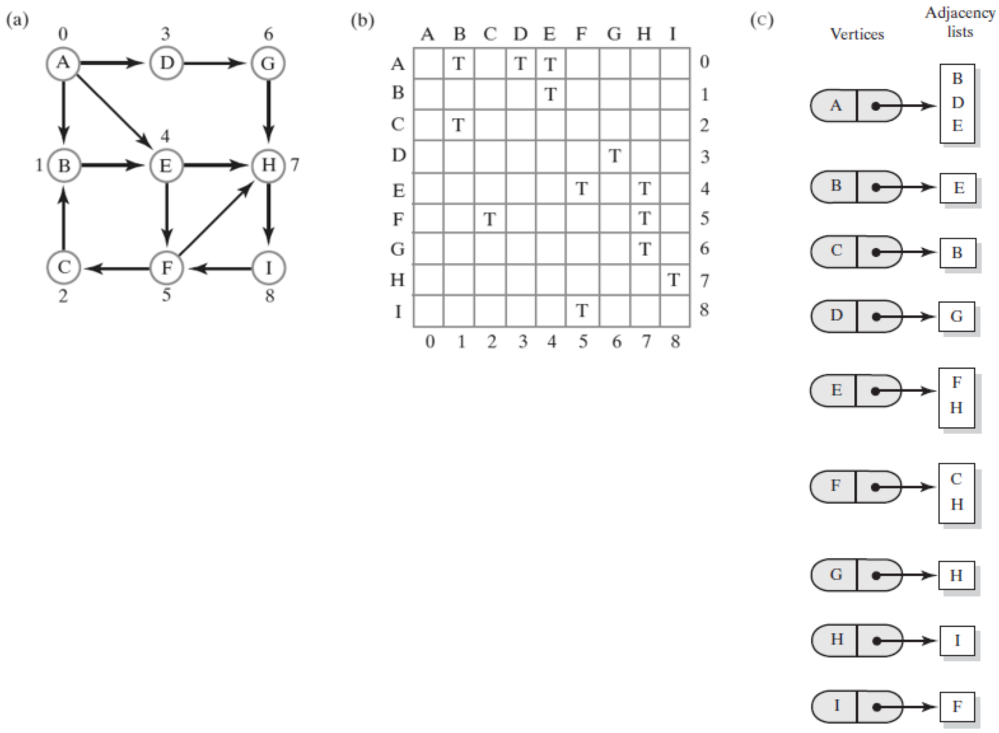
The adjacency matrix for a graph of n vertices has n rows and n columns. Each row and each column corresponds to a vertex in the graph. The vertices are numbered from 0 through n - 1 to match the row indices and the column indices. If $a_{ij}$ is the element in row `i` and column `j` of the matrix, $a_{ij}$ indicates whether an edge exists between vertex `i` and vertex `j`.

For an unweighted graph, you can use boolean values in the matrix. For a weighted graph, you can use edge weights when edges exist and a representation of infinity otherwise.

The adjacency matrix for an undirected graph is symmetric; that is, $a_{ij}$ and $a_{ji}$ have the same value. When an undirected graph has an edge from vertex `i` to vertex `j`, it also has an edge from vertex `j` to vertex `i`.

### The Adjacency List

An adjacency list for a given vertex represents only those edges that originate from the vertex. Space is not reserved for edges that do not exist. Thus, the adjacency lists use less memory than the corresponding adjacency matrix in Figure 1. For this reason, implementations of sparse graphs use adjacency lists.



|  | A | B | C | D | E | F | G | H | I |  |
|---|---|---|---|---|---|---|---|---|---|---|
| A |  | T |  | T | T |  |  |  |  | 0 |
| B |  |  |  |  | T |  |  |  |  | 1 |
| C |  | T |  |  |  |  |  |  |  | 2 |
| D |  |  |  |  |  |  | T |  |  | 3 |
| E |  |  |  |  |  | T |  | T |  | 4 |
| F |  |  | T |  |  |  |  | T |  | 5 |
| G |  |  |  |  |  |  |  | T |  | 6 |
| H |  |  |  |  |  |  |  |  | T | 7 |
| I |  |  |  |  |  | T |  |  |  | 8 |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |  |

Figure 1. (a) An unweighted, directed graph, (b) adjacency matrix, and (c) adjacency list

## Exercise – 1

In this exercise, you are expected to experiment with adjacency matrix representation of a directed and weighted graph.

### Step – 1

Create a new Java Project. Add the Graph.java class given to you in *src* folder.

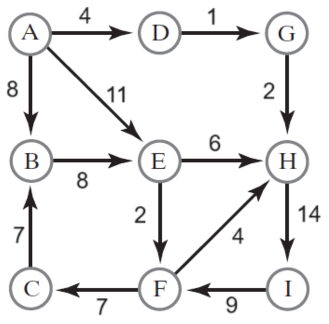
### Step – 2

Fill in the blanks in `addEdge` method of Graph.java following the adjacency matrix notation.

| **Your Code** |
|---|
| |

### Step – 3

Add a new class with the name of "Test.java". Create an instance of Graph.java and make necessary insertions to represent the graph given below:



### Step – 4

Print the final status of the graph object.

### Step – 5

Print the Breath First and Depth First Traversal order of the graph by starting at Vertex A.

| Your `Test.java` |
|---|
| |

| Your Output |
|---|
| |

## Exercise – 2

In this exercise, you are expected to experiment with adjacency list representation of a directed and weighted graph.

### Step – 1

Create a new Java Project. Add the DirectedGraph.java, Vertex.java, and Edge.java classes given to you in *src* folder.

### Step – 2

`getBreadthFirstTraversal` is already implemented in DirectedGraph.java. `getDepthFirstTraversal` will be implemented by you.

| **Your Code** |
|---|
| |

### Step – 3

Add a new class with the name of "Test.java". Create an instance of DirectedGraph.java and make necessary insertions to represent the graph in Exercise 1.

### Step – 4

Print the final status of the graph following the adjacency list notation.

### Step – 5

Print the Breath First and Depth First Traversal order of the graph by starting at Vertex A.

| Your `Test.java` |
|---|
| |

| Your Output |
|---|
| |