Assignment overview: GRAPHS!

Part I:

1. (2 pts) Write a class `AdjGraph` that implements a graph structure as an adjacency matrix.
2. (1 pt) Write a test program that uses your `AdjGraph` class to create and print some graphs.
3. (1 pt) Add a `degree()` method to your `AdjGraph` class.
4. (1 pt) Modify `Adjgraph` to support directed graphs.

Part II:

1. (2 pts) Add a method to `AdjGraph` that performs DFS on the graph.
2. (1 pt) Add a method to `AdjGraph` that tests if the graph has a cycle.
3. (2 pts) Add a method to `AdjGraph` that performs BFS on the graph.

This lab requires Java programming. You may (and should!) discuss the lab and coding techniques with your classmates, but all of the code you submit to Learning Hub must be your own.

Submission information

Due date: As shown on Learning Hub. Late assignments will not be graded.

What to submit:

1. Java file containing your AdjGraph class.
2. Java file containing your main() class testing and performing various methods of AdjGraph.
   Note: You will be adding code to both of these files as you progress through the parts of this lab. *Submit the Java files just one time after you have finished all the parts.*
3. Screenshots of output for all Parts/Questions of the lab. Please label your output so that it is clear for each portion of output which question is applicable. E.g. your code could include a statement such as “Here are the answers for Question 37:”.
   To make life easier for marking, it is helpful if you can combine all of the output screenshots into a single document (combined image, Word file, PDF, …)

Please do not zip or compress your submissions.

Part I Details

Question 1:

Write a class `AdjGraph` that implements a graph using an adjacency matrix. Initially your class will have methods addEdge() and toString() plus a constructor. You must use primitive array types. *Do not use ArrayList!*

The constructor `AdjGraph(int V)` allocates space for a graph with V vertices (i.e. it creates a VxV matrix) and zero edges.

The addEdge() method (`void addEdge(int u, int v)`) adds an edge to the graph from vertex u to vertex v, where u and v are integers in the range 0..V-1.

The toString() method (`String toString()`) returns a String that represents the adjacency matrix with one line of output for each vertex (each line separated by newlines). For example, here is the resulting string for a graph with three vertices and two edges:

```
0 1 1
1 0 0
1 0 0
```

Question 2:

Write a test program that uses your `AdjGraph` class to create and then print the three sample graphs shown below.

In your main() you can hardcode the edges; e.g. you might have `myGraph.addEdge(1,2)` to create an edge from vertex 1 to vertex 2.

	0	1	2	3	4
0	0	1	0	1	1
1	1	0	1	0	1
2	0	1	0	1	0
3	1	0	1	0	1
4	1	1	0	1	0

	0	1	2	3
0	0	1	0	0
1	1	0	1	0
2	0	1	0	1
3	0	0	1	0

	0	1	2	3	4	5
0	0	0	1	0	1	0
1	0	0	0	1	0	1
2	1	0	0	0	1	0
3	0	1	0	0	0	1
4	1	0	1	0	0	0
5	0	1	0	1	0	0

Question 3:

Modify your `AdjGraph` class to contain a method `int degree(int v)` that returns the degree of a vertex.

Add some tests to your main() to make several calls to degree() on vertices of the above sample graphs to make sure this function works correctly.

Question 4:

Modify your `AdjGraph` class to support directed graphs. To do this you will add a variable `directed` which can be set from your test program, and a method `boolean isDirected()` that returns the current value of this variable.

You will need to modify addEdge() to work correctly for both directed and undirected graphs, and add two new methods `int inDegree(int v)` and `int outDegree(int v)`.

Here is a sample graph

	0	1	2	3	4
0	1	0	0	0	1
1	0	0	1	0	1
2	1	0	0	1	0
3	0	1	1	0	0
4	0	0	0	1	0

Add this graph to your main(), be sure it is marked as directed, and include some calls to inDegree() and outDegree() on several of the vertices.

Part II Details

In this part you will implement both DFS and BFS in your AdjGraph class, and you will use them to solve some problems.

Question 1:

Add a method `void DFS()` to your `AdjGraph` class that performs depth-first search.

Model your code after the pseudocode for DFS in the textbook and the lecture notes from Week 8. The pseudocode for DFS() uses a recursive "helper" function dfs(), and your implementation should have both of these. When selecting vertices to visit, break ties by choosing the vertex that is next in lexicographic order.

Test your DFS by having it print the vertex labels in the order that they are visited. For example, if your input graph is the one shown here:

	0	1	2	3	4	5	6	7
0	0	1	1	0	1	0	0	0
1	1	0	0	1	0	1	0	0
2	1	0	0	1	0	0	1	0
3	0	1	1	0	0	0	0	1
4	1	0	0	0	0	1	1	0
5	0	1	0	0	1	0	0	1
6	0	0	1	0	1	0	0	1
7	0	0	0	1	0	1	1	0

Then your DFS would produce the following output:

```
visiting vertex 0
visiting vertex 1
visiting vertex 3
visiting vertex 2
visiting vertex 6
visiting vertex 4
visiting vertex 5
visiting vertex 7
```

Question 2:

Apply your DFS algorithm to the problem of checking for cycles in a graph. Add a method `boolean hasCycle()` to your AdjGraph class. The method should return true if there is a cycle in the graph, false otherwise.

Test your method with a couple of sample graphs (at least one with a cycle, one without). Be sure to include the results of these tests in your output. And be sure it's clear which graphs you are testing.

Question 3:

Add a method `void BFS()` to your `AdjGraph` class that performs breadth-first search on the graph. Model your code after the pseudocode for BFS in the textbook and the lecture notes from Week 8. When visiting the (unvisited) neighbours of one particular vertex, visit them in lexicographic order.

Print the same style of output for BFS as you did for DFS in Question 1. For the sample graph in Question 1, you would see this:

```
visiting vertex 0
visiting vertex 1
visiting vertex 2
visiting vertex 4
visiting vertex 3
visiting vertex 5
visiting vertex 6
visiting vertex 7
```