Assignment overview: GRAPHS!

This week it's the beginning of our fun with graphs! Here is a summary of the program you will write. More details on all the parts are below.

1. (3 pts) Write a class `Graph` that implements a graph structure using an adjacency matrix, including methods addEdge() and toString().
2. (3 pts) Write a test program that uses your `Graph` class to create and print some graphs.
3. (1 pt) Add a `degree()` method to your `Graph` class.
4. (1 pt) Modify `Graph` to support directed graphs.
5. (2 pts) Add a method to `Graph` that performs Depth First Search (DFS) on the graph.
6. (2 pts) Add a method to `Graph` that performs Breadth First Search (BFS) on the graph.

The lab is worth 15 points. There are 3 points leftover for general good design and programming style.

Submission information

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

What to submit:

1. Java file containing your Graph class.
2. Java file containing your Main class testing and performing various methods of Graph. **Note**: You will be adding code successively to both of these files as you progress through the parts of this lab. *Please submit the Java files just one time after you have finished all the parts.*

Please do not zip or compress your submissions.

More details

Question 1:

Write a class `Graph` 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 `Graph(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
```

Note: We have not made provision for any names/labels for the vertices. Think of them as simply numbered from 0 to V-1. The above output signifies that the graph has vertices V = {0, 1, 2}, and edges E = {(0, 1), (0, 2)}.

Question 2:

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

In your main() you can hardcode adding 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

Note: Your graph output need not include the (imaginary) vertex labels. For now it is enough to print the contents of the matrix, unlabeled. For example (the first graph above):

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

Question 3:

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

You must also modify addEdge() so that it works correctly whether the graph is directed or undirected, and add two new methods `int inDegree(int v)` and `int outDegree(int v)`.

Here is a sample directed 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 designated as directed, and include some calls to inDegree() and outDegree() on several of the vertices.

The methods degree(), inDegree() and outDegree() should all return -1 if they are called on the wrong "type" of graph (directed/undirected).

Question 5:

Add a method `void DFS()` to your `Graph` 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, and your implementation should have both of these. When selecting vertices to visit, break ties by choosing the vertex that comes first in numerical 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 6:

Add a method `void BFS()` to your `Graph` 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 increasing numerical order.

Print the same style of output for BFS as you did for DFS in Question 5. For the sample graph in Question 5, 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
```