Assignment overview: GRAPHS!

This week it's the beginning of our fun with graphs! Here is a summary of the program you will write. There are full details in the sections below.

NOTE: There are lots of details in this lab. Be sure to read all the sections thoroughly and carefully so that you don't overlook anything!

1. (4 pts) Write a class `Graph` that implements a graph structure using an adjacency matrix, including methods addEdge() and toString().
2. (4 pts) Write a main/test program that uses your `Graph` class to create and print some graphs. *You will add more code to main() incrementally as you do the following parts.*
3. (2 pts) Add a `degree()` method to your `Graph` class.
4. (2 pts) 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.

This assignment is worth 20 points. There are 4 points allocated for comments and coding style.

How, what, and when to submit:

Please submit the following to the dropbox on Learning Hub:

- Java file containing your Graph class
- Java file containing your Main class testing and performing various methods of Graph. Note: You will be adding code incrementally to both of these files as you progress through the parts of this lab.

Please do not zip or compress your files, or send entire project directories. (I know it's tempting).

This lab is worth 20 points.

It is due at midnight next Wednesday, 3 November.

Detailed requirements

Part 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 a primitive 2D array for the adjacency matrix. (It is OK to use ArrayList for other things, if you have any other arrays/lists of things that you need to store for any reason.)

The constructor `Graph(int V)` allocates space for a graph with V vertices (i.e. it creates a VxV matrix) and zero edges. Note that V here is being used just as the *number of vertices*. It is not a set of vertices. (Or, you could think of it as secretly designating a hidden meaning of the set of integers from 0 to V-1, but you are not actually *storing* these numbers in the class.)

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 each line of output indicating the neighbours for one vertex. 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 allowance for any names/labels for the vertices. We know the vertices are secretly the numbers 0 to V-1, but we have not stored them anywhere. The above adjacency matrix signifies that the graph has 3 vertices V = {0, 1, 2}, and 2 edges E = {(0, 1), (0, 2)}.

Part 2:

Create a separate driver class with a main() and any helper functions you wish that uses your `Graph` class to create and then print the three sample graphs shown below.

In your main() (and/or helpers) you can hardcode adding the edges in these graphs; e.g. you would use `Graph1.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 toString() graph method does **not** have to produce the vertex labels as shown above (but you may if you wish). For this lab it is enough to display the contents of the matrix. 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
```

Part 3:

Modify your `Graph` class to add 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.

Part 4:

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

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

Create this graph in your driver program, be sure it is designated as directed, and perform some calls to inDegree() and outDegree() on some 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).

Part 5:

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

You may model your code after the pseudocode for DFS in the textbook or the lecture notes from Week 7. 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*. This is the so-called "DFS order" from the lecture notes. 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
```

Your DFS method does not need to store or return the list of vertices in the visited order. It is enough to simply modify the DFS algorithm so that it prints the vertices while the traversal is being performed. (Note: You *may* have your DFS method save/return the list of vertices so that you can print it all together later. But you don't have to.)

Part 6:

Add a method `void BFS()` to your `Graph` class that performs breadth-first search on the graph. You may model your code after the pseudocode for BFS in the textbook and the lecture notes from Week 7. 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 Part 5. For the sample graph in Part 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
```

As with DFS, your BFS method does not need to store or return the list of vertices; printing them during the traversal is sufficient.

Output summary

This is an overview of all the output that your program should produce. *All output must include appropriate descriptive messages.* For example "Here is the adjacency matrix for Graph 1:"

For Part 1, there is no output.

For Part 2:

- The toString() output for all three of the required graphs.

For Part 3:

- Results of *at least 5 calls* to your degree() method.

For Part 4:

- The adjacency matrix of the directed graph shown.
- Results of at least 2 calls to inDegree() and outDegree() on the directed graph.
- Results of *invalid* calls to all three of the degree methods (i.e., call them on the wrong graphs).

For Part 5:

- The adjacency matrix of the sample graph shown.
- Results of the DFS traversal as shown.

For Part 6:

- Results of the BFS traversal of the same graph you used in Part 5.

More information, tips, etc.

What algorithm

It is not strictly required for you to follow my own pseudocode for DFS or BFS. There are other versions available.

However, you must follow the *signature* that is specified for the DFS/BFS function: these should be public methods in your Graph class that take no arguments and return no value (type is `void`). They will perform the search on the object on which they are being called, i.e. your main/test class will call the DFS method by doing something like `myGraph.DFS()`. Their only "result" is to display output as they execute.

Another sample graph

Here is one more sample graph that you can use for testing purposes. You don't have to include this in your final output, but it's fine if you do. It is the graph that appears in the lecture notes for "DFS example" and "BFS example" with vertices a..h. Shown here as Java code for easy copy/pasting if it's helpful:

```
G.addEdge(0, 1); //ab
G.addEdge(0, 4); //ae
G.addEdge(0, 5); //af
G.addEdge(1, 5); //bf
G.addEdge(1, 6); //bg
G.addEdge(2, 3); //cd
G.addEdge(2, 6); //cg
G.addEdge(3, 7); //dh
G.addEdge(4, 5); //ef
G.addEdge(6, 7); //gh
```

Here are the DFS and BFS results for this graph (note how the numbers 0..7 correspond with a..h in the lecture notes):

```
DFS traversal of graph:
Visiting vertex 0
Visiting vertex 1
Visiting vertex 5
Visiting vertex 4
Visiting vertex 6
Visiting vertex 2
Visiting vertex 3
Visiting vertex 7

BFS traversal of graph:
BFS visiting vertex 0
BFS visiting vertex 1
BFS visiting vertex 4
BFS visiting vertex 5
BFS visiting vertex 6
BFS visiting vertex 2
BFS visiting vertex 7
BFS visiting vertex 3
```