Graph Data-Structure

In this first assignment, you are asked to develop a set of core graph data-structures. Future labs will use these graph structures as a test-bed for more advanced algorithms and for building more practical applications. You will be developing three classes: Graph, UndirectedGraph, and DirectedGraph.

Data Representation

The internal representation of your class should be an adjacency matrix.

Implementations

You are free to use either C++ or Java (and I encourage you to try out the one you are less familiar with, but you don’t have to…). The programs need to be able to run under a Wake Forest supported Unix operating system (Linux, Solaris, or Cygwin). If you use Java, destructors are irrelevant, so you can ignore implementing those methods.

Submission

Submissions will be via Blackboard. Make sure your name is in a comment near the top of each of your files. Please zip or tar/gzip your files into an archive before uploading them. Please include a makefile which includes a target called main which, when called, compiles all of your code as well as compiles in a main.cpp or main.java program. You do not need to submit your own main.cpp or main.java, but you should write one to test out your methods.

Grading

Your implementations will be judged on a number of factors:
- Correctness and fullness of implementation of requested API (65%)
- Internal documentation and coding style (15%)
- Efficient implementations (20%)

As you will note, I have given very few details about how to implement the methods – make sure that you make operations cheap whenever possible, as these operations may be used many times over in later operations. I would suggest that time efficiency is more appropriate than space efficiency for this problem, but think about tradeoffs you make.
Graph Class Methods

You should develop the following methods for the Graph class. These may need to be overwritten in the UndirectedGraph and DirectedGraph subclasses, but should be methods callable on all Graph objects.

**Graph(???)**
- A default constructor which creates a size 0 by 0 graph.
- A constructor that takes the number of vertices to represent in the graph and which allocates appropriately sized dynamic memory to represent the graph. The graph should be initialized to the empty graph (no edges).
- A constructor that takes in a pointer to a Graph object and creates a copy of the Graph passed in as the parameter.

**~Graph()**
The destructor which de-allocates any dynamically allocated memory.

**void addEdge(int startVertex, int endVertex)**
Adds an edge to the graph, associating a weight of 1 with the edge.

**void addEdge(int startVertex, int endVertex, double weight)**
Adds an edge to the graph, associating the weight passed in as a parameter to the edge. Assume positive, non-zero weights only will be passed in, allowing us to still use zero as the no-edge present marker.

**void deleteEdge(int startVertex, int endVertex)**
Removes an edge from the graph

**double getWeight(int startVertex, int endVertex)**
Returns the weight associated with an edge in the graph, or a zero if that edge doesn’t exist.

**boolean adjacent(int vertexOne, int vertexTwo)**
Returns true if the two nodes are adjacent, false otherwise.

**int edgeCount()**
Returns the number of edges in the graph.

**int vertexCount()**
Returns the number of vertices in the graph.
UndirectedGraph Class

You should develop the following methods for the UndirectedGraph class.

Appropriate constructors and destructors to match the basic structure of the constructors and destructors of the Graph class.

Overwrite any methods from Graph that need overloading to meet the specific semantic requirements of an UndirectedGraph.

Add the new method:

**int degree(int vertex)**

Returns the degree of the vertex passed as a parameter

DirectedGraph Class

You should develop the following methods for the DirectedGraph class. These may need to be overwritten in the subclasses, but should be methods callable on all Graph objects.

Appropriate constructors and destructors to match the basic structure of the constructors and destructors of the Graph class.

Overwrite any methods from Graph that need overloading to meet the specific semantics of a DirectedGraph.

Add the new methods:

**int inDegree(int vertex)**

Returns the in-degree of the vertex passed as a parameter

**int outDegree(int vertex)**

Returns the out-degree of the vertex passed as a parameter

Error Handling

Make sure that any method that takes in vertices or weights as parameters

1) Checks that the vertices are valid vertices (given the number of vertices assigned to the Graph when it is first constructed) and the weights are > 0

2) If a vertex or the weight is invalid, print an error message and handle the return value appropriately (don’t add an edge, indicate that the two things aren’t adjacent, return a 0 or -1 degree, etc).

Miscellaneous

For a graph with \( n \) vertices, assume they are labeled 0 to \( n-1 \).
If someone asks adjacent(2,3) in a directed graph, assume they are asking \( 2 \rightarrow 3 \)?