CSC 325 Adv Data Structures

<u>Program #5</u>

Graphs

Description:

For this assignment you are to read in data from two files, given to you as "input.txt" and "input_weights.txt". These files contain information on the layout of two graphs (I highly recommend opening it and examining the contents). They are both in the adjacency list format.

You are to read in the file "input.txt" (make sure it is in the same directory as your .java file) and convert it into an adjacency matrix. You then use this adjacency matrix to perform both a breadth-first search and a depth first search (algorithms for both are given in the notes) outputting the vertices' names as they are visited. <u>Use vertex **a** as your starting vertex</u>.

You are also to read in the file "input_weights.txt". This file is a directed graph with weights on the arcs. Convert this into an adjacency matrix (with the weights, using 0 to indicate there is no edge there). Perform Dijkstra's algorithm on the adjacency matrix, <u>with vertex **a** as your source</u>.

Take the total output generated and print it to a file called "output.txt". You will need to create and write to this file in your code.

<u>To be clear, you are using File I/O for this assignment. You should be reading directly from both files and writing directly to the output.txt file (that your code should generate). So, no redirection is happening here.</u>

Entry Point:

You will only create a single public class called Main. This class will contain all the code you need to create the file output below (including your entry point). Feel free to have as many methods and fields as you like.

About data structures:

You will need several data structures to accomplish this. You are to use the ones that come with Java. You will need a Stack and Queue but you may also find using additional data structures (such as a HashMap and/or ArrayList) to be helpful.

To be clear, your adjacency matrix should be a simple 2d array, defined with `int[][]` as the datatype. Example:

```
    int[][] adjMat;    // or whatever name you give it
```

You could also use an ArrayList of ArrayList of Integer, storing the 1's and 0's in that, however I think using a normal 2d array would be a far simpler approach. Your call.

Grading:

For grading purposes, I might change up the graphs in the input files to give your program different graphs. It should work for any graph I give it. I will keep the weights as single digits (for consistency in scanning). I highly recommend trying different graphs out and ensuring that BFS and DFS are correct for any undirected, unweighted graph you give it and that Dijkstra's is correct for any directed, weighted graph you give it.

To be clear, and so you don't have to write extra code, I will keep the vertex names consecutive. This means that if the graph has 4 vertices, they will be 'a', 'b', 'c', and 'd'. If the graph has 5 vertices, they will be 'a', 'b', 'c', 'd', and 'e'. You will always have a vertex 'a' and the next vertex will always be 'b', then 'c', etc. The graph will also never contain less than 3 vertices, nor more than 9 vertices. And the weights (as stated above) will be single digits in the range of 1 to 9 (i.e. not 0 and not negative).

Sample Output:

Your output must look exactly like the output below (if given the original input files):

```
$ javac Main.java && java Main
$ cat output.txt
Adjacency Matrix (undirected):
  abcdefghi
a 001000000
b 000100000
c 100110000
d 011011000
e 001100000
f 000100100
g 000001011
h 000000100
i 000000100

BFS: acdebfghi
DFS: acedfgihb

Adjacency Matrix (directed w/weights):
  abcdefghi
a 004000000
b 000000000
c 000170000
d 030023009
e 000000000
f 000320100
g 000000043
h 000000000
i 000000000

Dijkstra's: a:0,b:8,c:4,d:5,e:7,f:8,g:9,h:13,i:12
```

Rubric:

#	ITEM	POINTS
1	read from input file	5
2	convert into adjacency matrix representation	5
3	breadth-first search	10
4	depth first search	10
5	Dijkstra’s	10
6	write to output file	5
7	output matches	5
	TOTAL	50

#	PENALTIES	POINTS
1	Doesn’t compile	-50%
2	Doesn’t execute once compiled (i.e. it crashes)	-25%
3	Late up to 1 day	-25%
4	Late up to 2 days	-50%
5	Late after 2 days	-100%