Introduction

In this assignment, you will do two tasks.
In part 1 of this assignment, you will design a graph class. In part 2, you will put the graph you designed in the graph homework to use by modeling the Marvel Comics universe. This application builds a graph containing thousands of nodes and edges. The size of the graph might expose performance issues. With a well-designed implementation, your program will run in a matter of seconds. Bugs or suboptimal data structures can increase the runtime to anywhere from several minutes to 30 minutes or more.

GRAPH ADT

Your graph ADT

```
public class Graph {

    private final int V; // number of vertices
    private int E; // number of edges
    private final Bag<Integer>[] adj; // adjacency lists
    private final String[] VertexNames;
    private final String[] EdgeNames;
```

The MarvelPaths Application

In this application, your graph models a social network among characters in Marvel comic books. Each node in the graph represents a character, and an edge ⟨`Char1`, `Char2`⟩ indicates that `Char1` appeared in a comic book that `Char2` also appeared in. There should be a separate edge for every comic book any two characters appear in, labeled with the name of the book. For example, if Zeus and Hercules appeared in five issues of a given series, then Zeus would have five edges to Hercules, and Hercules would have five edges to Zeus, each edge associated with the name of the book as a value.

You will write a class `marvel.MarvelPaths` (create the file yourself: `MarvelPaths.java`) that reads the Marvel data from the given files, builds a graph, and finds paths between characters in the graph.

Part 1: Building the Graph

The first step in your program is to parse the provided csv files.

You have two CSV files.
LISTOFheroesANDcomics.csv: each line has a name and whether they are a hero or a comic
heroesINcomics.csv: each line has a hero and a comic in which they appear.

CSV means comma separated values. So, the two values in each line of the above two files are separated by a comma.

The first step in your program is to construct your graph of the Marvel universe from these data files.

It might be useful to think of it in two parts:

Part 1: a parsing stage, where you read the files into suitable data structures.
Part 2: a graph construction stage.

Remember that, for each comic book relationship between Character A and Character B, there should be two edges: one from Character A to Character B, and one the other way around, from Character B to Character A.

At this point, it's a good idea to test the parsing and graph-building operation in isolation. Verify that your program builds the graph correctly before continuing.

Part 2: Finding Paths

The real meat of `MarvelPaths` is the ability to find paths between two characters in the graph. Given the name of two characters, `MarvelPaths` should search for and return a path through the graph connecting them. How the path is subsequently used, or the format in which it is printed out, depends on the requirements of the particular application using `MarvelPaths`, such as your test driver (see below).

Your program should return the shortest path found via breadth-first search (BFS). A BFS from node u to node v visits all of u's neighbors first, then all of u's neighbors' neighbors, then all of u's neighbors' neighbors' neighbors, and so on until v is found or all nodes with a path from u have been visited. Below is a general BFS pseudocode algorithm to find the shortest path

between two nodes in a graph G. For readability, you should use more descriptive variable names in your actual code:

```
start = starting node

dest = destination node

Q = queue, or "worklist", of nodes to visit: initially empty

M = map from nodes to paths: initially empty.

    // Each key in M is a visited node.

    // Each value is a path from start to that node.

    // A path is a list; you decide whether it is a list of node
s, or edges,

    // or node data, or edge data, or nodes and edges, or someth
ing else.


Add start to Q

Add start->[] to M (start mapped to an empty list)

while Q is not empty:

    dequeue next node n

    if n is dest

        return the path associated with n in M

    for each edge e=⟨n,m⟩:

        if m is not in M, i.e. m has not been visited:

            let p be the path n maps to in M

            let p' be the path formed by appending e to p

            add m->p' to M

            add m to Q


// If the loop terminates, then no path exists from start to des
t.
```

```
// The implementation should indicate this to the client. Note t
hat

// BFS returns the path with the fewest number of edges.
```

Many character pairs will be connected by multiple shortest paths with the same length

Using the full Marvel dataset, your program must be able to construct the graph and find a path in less than 30 seconds.