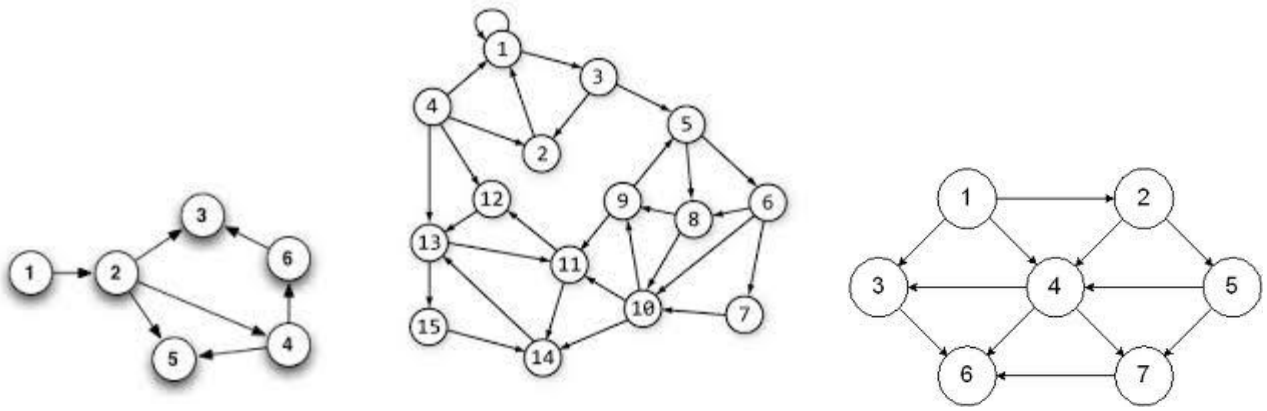
Programming Project 14

Due: Tuesday, 12/2/14 at 11:59 pm

Can I Get There from Here?

For this Java program, we can use the exact same data "format" as we used with the first hash table program: an array of linked lists of integer value. We will use this to represent a travel network.

Assume you have a small airline that flies planes between a small number of airports. Each airport will be given a number. If a plane flies from airport X to airport Y, the network will have an "edge" from X to Y. Below are a number of drawings that could represent this idea. The airports are represented by the circled numbers, the edges are represented by the arrows. Consider the first drawing. It has 6 airports. It has a plane that flies from airport 1 to airport 2. Three planes fly from airport 2, one to airport 3, one to airport 4 and one to airport 5. No planes leave from airport 3 or from airport 5 (yes, it would be stupid to strand planes at those airports, but ignore that fact for now). Planes fly from airport 4 to airports 5 and 6. Finally, planes fly from airport 6 to airport 3.



If the travel network has N airports, the array will have N linked lists, one for each airport. If airport X has planes flying to 3 different airports, the linked list for airport X would have 3 nodes.

The input for the operations will come from standard input and from files. The input will initially come from standard input. In order to make reading in the input a bit easier, the commands will be given as negative integer values. This allows the vast majority of the inputs to be entered as integer values. If the user specified the filename command (-7), your program will then read input from a file. See the description below for more details. The commands are to follow the descriptions given below. Note: that the form <int> could be any integer number and it will NOT be enclosed in angle brackets. <int> is just a notation to specify and integer value. If the first value on the line is not one of the following characters, print an error message and ignore the rest of the information on that line.

-1 - **Quit**: quit the program immediately.

-2 <int1> <int2> - **Travel**: display a message stating whether a person can travel from airport **<int1>** to airport **<int2>** in one or more flights.

-3 <int> - **Resize**: remove all values from the traffic network and resize the array to contain the number of airports as indicated by the given integer value. Be sure to properly deallocate all of the nodes in each linked list. The value of the integer must be greater than zero. The airports will be numbered from 1 to the given integer value.

-4 <int1> <int2> - **Insert**: insert the edge to indicate a plane flies from airport **<int1>** to airport **<int2>**.

-5 <int1> <int2> - **Delete**: delete the edge that indicates a plane flying from airport **<int1>** to airport **<int2>**.

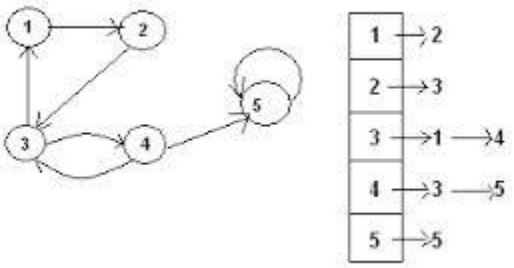
-6 - **List**: list all the items contained in the travel network. First display all of the airports (if any) that can be reached from the first airport in one flight (that have an edge in the network), followed by all the airports (if any) that can be reached from the second airport in one flight, etc.

-7 <filename> - **Filename**: open the file indicated by the **<filename>** (assume it is in the current directory) and read commands from this file. When the end of the file is reached, continue reading commands from standard input. To stop a possible case of an infinite loop, the **f** command is only valid when reading from standard input.

Note that the array will only be resized while it is being emptied of all values. Initially your program should have the array to hold 10 airports. If a command specifies an airport outside of the valid range, print an error message and ignore the command.

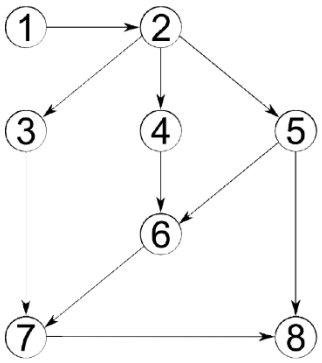
The Storage Structure for the Travel Network

There are many ways in which the travel network could be stored. The most common way is called an "adjacency list". In an adjacency list, each location/airport needs a list of those locations/airports that one can get to in one move/flight. In the Maze program and the Game of Life program, the adjacent locations were determined by location in the 2-D array. Here the adjacent locations have to be given by the input. In this program, we need a list for each airport. If we store all the airport information in an array, we would have an array of lists, which is the same storage structure as the Hash Table. This can be a very useful storage structure as it can be adapted to many different kind of programming applications. Consider the following image showing a travel network and an adjacency list:



Not the above list of adjacent locations could actually be stored as an array since we know that the number of items in the list can only be the number of airports in the travel matrix.

Another approach is an "adjacency matrix". An adjacency matrix is a 2-D array of Boolean values. The value at location adjMatrix[i][j] is true if there is a flight from location/airport i to location/airport j. Consider the following as an example of an adjacency matrix (note line from 3 to 7 is hard to see):



	1	2	3	4	5	6	7	8
1		1						
2			1	1	1			
3							1	
4						1		
5						1		1
6							1	
7								1
8								

Java also has the built-in library classes of ArrayList and Vector as part of the Java Foundation Class libraries. These classes implement a dynamic array automatically for the programmer. Both ArrayLists and Vectors use Generics which is extremely useful once you understand the needed syntax. The Vector class is a special version of ArrayList designed to handle concurrency issues. So for this program where concurrency is not being used, ArrayList would be the proper class to use. A good page on the simple uses of ArrayList can be found at:

http://javarevisited.blogspot.com/2011/05/example-of-arraylist-in-java-tutorial.html

Items 1, 2, 5, 7, 8, 11 and perhaps 15 would be useful.

For this program, you may use whichever implementation you wish: adjacency lists (storing the lists using linked lists, arrays or ArrayLists) or an adjacency matrix.

Algorithm to See if Someone Can Travel from X to Y

To determine if a person can travel from airport X to airport Y in one or more flights, a recursive depth-first-search algorithm should be used. For this algorithm to work, we will need to be able to mark each airport as visited. Setting up an array of TRUE/FALSE values will work. The pseudo code for this algorithm is as shown below. Note that asking to go from airport X to airport X in one or more flights, is a valid question. It really asks, "If I leave airport X, can I return to it?"

```
void depthFirstSearchHelper (int x, int y)
{
  mark all airports as unvisited;
  if ( dfs (x, y) = = TRUE)
     output ("You can get from airport " + x + "  to airport " + y + " in one or more flights");
  else
     output ("You can not get from airport " + x + " to airport " + y + " in one or more flights");
}

Boolean dfs (int a, int b)
{
  for (each airport c that can be reached from a in one flight)
      {
        if (c = = b)
            return TRUE;
        if ( airport c is unvisited )
           {
             mark airport c as visited;
             if ( dfs (c, b) = = TRUE )
                  return TRUE;
           }
       }
  return FALSE;
}
```

MULTIPLE SOURCE CODE FILES

Your program is to be written using at least two source code files. One way to divide the code is to have:

- one file for the class that stores the travel network and
- another file to contain main and the code needed to process the input

Program Submission

You are to submit the programs for this lab via the Assignments Page in Blackboard.

To help the TA, name your file with your net-id and the assignment name, like:

- ptroy1LabX.java