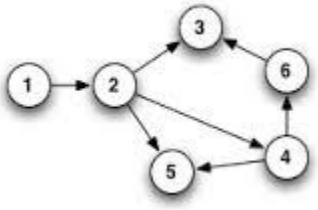
Programming Project 7

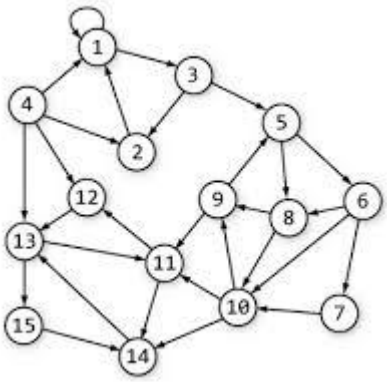
Due: Friday, 5/1/15 at 11:59 pm

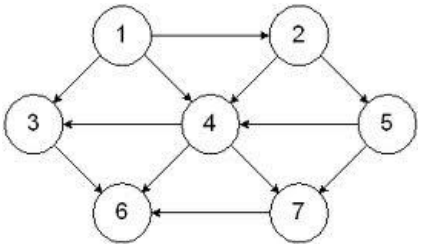
Can I Get There from Here?

For this program, you will write a Java Program to represent a travel network. This travel network will use an array of linked lists as its primary storage structure. This type of storage structure is typically called an adjacency list. Please note: this write-up was originally taken from a C assignment. Any reference to C items were not properly "translated" from C to Java. Such items should use their Java counterparts.

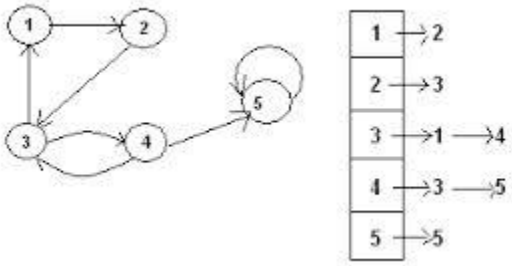
Assume you have a small airline that flies planes between a small number of airports. Each airport will be given a number. If a plane flies from airport X to airport Y, the network will have an "edge" from X to Y. Below are a number of drawings that could represent this idea. The airports are represented 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.







In an adjacency list, each location/airport needs a list of those locations/airports that one can get to in one move/flight. In this program, we need a list for each airport. 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. Consider the following image showing a travel network and an adjacency list:



There are 5 airports, so we have an array of 5 linked lists. Since Airport 3 can fly planes to two Airports, namely Airport 1 and Airport 4, the linked list for Airport 3 has two nodes. One node containing the value 1. Another node containing the value 4.

Program Input and Commands

The input for the operations will come from standard input and from files. The input will initially come from standard input. If the user specified the f command, 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. The integer value is to be input on the same line as the command character. If the first character on the line is not one of the following characters, print an error message and ignore the rest of the information on that line.

q - quit the program immediately.

? - display a list of the commands the user can enter for the program.

- ignore this line of input. Treat the line of input as a comment

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

r <int> - remove all values from the traffic network and resize the array to contain the number of airports as indicated by the given integer value. The value of the integer must be greater than zero. The airports will be numbered from 1 to the given integer value. If this were a C program, you would need to properly deallocate all of the nodes in each linked list when resizes. This is a reason to be thankful for Java's garbage collection policy.

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

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

l - 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.

f <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 previous input source. This must be handled using recursion. Beware of a possible case of an infinite recursive loop, the **f** command is may not call a file that is currently in use.

Initially your program should have the array to hold 10 airports. If a command specifies an airport outside of the current valid range, print an error message and ignore the command.

The use of the Scanner class for reading in input is expected to be used. It allows for a nice, simple elegant solution for the input needs of this program. While using the Scanner class is not going to be made a requirement, don't expect help from the instructor or TA's if you decide not to use the Scanner class. You can use the Scanner class to read from standard input or from a file with the use of the proper constructor. You are required to use method(s) to read in your input that take an instance of the Scanner class as a parameter (see F Command below). For ideas on the Scanner class, see the web pages of:

- http://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html
- http://www.cs.utexas.edu/users/ndale/Scanner.html

Travel Algorithm and the Airport Object

To determine if a person can travel from airport X to airport Y in one or more flights, a recursive depth-first-search algorithm must be used. For this algorithm to work, we will need to be able to mark each airport as visited. Setting up an Airport class/object is the best way to do this in Java. This object will contain the head of the linked list for the airport's adjacency list and the object will also contain a Boolean value to determine if an airport has been visited or not. The travel network **MUST** be a dynamic array of these Airport objects. The adjacency list will also need a Node class/object to store the linked list information.

The pseudo code for this algorithm is as shown below. Note it is valid to ask, can I to go from airport X to airport X in one or more flights. It really asks, "If I leave airport X, can I return to it?" This algorithm is recursive and you MUST use this recursive algorithm in your program.

```
void depthFirstSearchHelper (int x, int y)
{
  mark all airports as unvisited;
  if ( dfs (x, y) = = TRUE)
     print ("You can get from airport " + x + " to airport " + y " in one or more flights");
  else
     print ("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;
}
```

The FILE Command: f

The f command may seem difficult to implement at first, but it has a creative solution that you are to use. The code in the file Proj7Troya.java is intended to give you an idea on how this solution is to be implemented.

First note that main( ), is extremely short. It creates an instance of the Proj7Troya class and calls the processCommandLoop( ) method with an instance of the Scanner class that reads from standard input.

The method processCommandLoop( ) reads from the input source specified by the parameter and determines the which command is being invoked.

When the f command is invoked, it is to open the file specified by the command, create a new instance of the Scanner class that reads from this file. Then make a recursive call to processCommandLoop( ) with this new instance of the Scanner class so the next line of input comes from the specified file instead of where the previous command came from. When the end of a file is reached, the program is to revert back to the previous input source that contained the f command. This previous input source could be standard input or a file. By making these calls recursively, reverting back to the previous input source is a complete no-brainer.

However, this can cause an infinite loop if you try to access a file that your program is already reading from. Consider this scenario. Assume the user enters a command from standard input to start reading from file A. However; file A tells you to read from file B, file B tells you to read from file C, and file C tells you to read from file A. Since you always start reading from the top of the file, when file C eventually tells the program to read from file A, the program will reprocess the command to read from file B, which will reprocess the command to read from file C, which will reprocess the command to read from file A, which will reprocess the command to read from file B, which will reprocess the command to read from file C, which will reprocess the command to read from file A, which will reprocess…

In order to stop this, you are required to maintain a linked list of file names. Before the f command attempts to create a new instance of the Scanner class that read from file X, the f command is to check if the linked list of file names already contains the name of X.

- If the name X already exists in the linked list, the f command will NOT create a new instance of the Scanner class and it will NOT make the recursive call to processCommandLoop( ).
- If the name X does not exist in the linked list, the f command will add the name X to the linked list before making the recursive call to processCommandLoop( ) and it must remove the name X from the linked list after the call t0 processCommandLoop( ) returns.

You are responsible to write the code for this linked list yourself. Note that this will most likely be a linked list of Strings, while each airport's adjacency list will most likely be a linked list of integers.

You are not allowed to use any of the classes from the Java Collections Framework in this program. These classes include ArrayList, Vector, List, Set, HashMap, etc. **If you need such a class, you are to write it yourself.** A full listing of the Java Collections Framework can be found at:
http://docs.oracle.com/javase/7/docs/technotes/guides/collections/reference.html

Multiple Source Code Files

Your program is to be written using at least two source code files. One of the source file files is to contain the main method of the program named in a file using your NetId and Program name, like:

Ptroy1Proj7.java

The other source code file must contain your Airport class in a file named:

Airport.java

You may use additional source code files if you wish, but these two are required.

Program Submission

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

To help the TA, zip your files together and name your zip file with your net-id and the assignment name, like: Ptroy1Proj7.zip