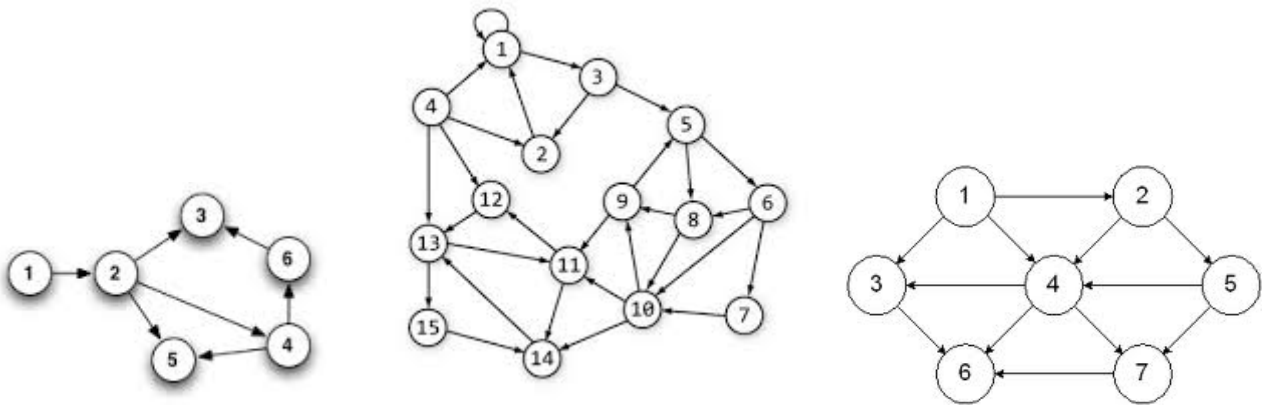
Programming Project 14

Due: Monday, 12/2/13 at 11:59 pm

Can I Get There from Here?

For this program, we will 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. 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.

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. 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.

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 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 use of the file input commands like fscanf or fgets with the value of **stdin** for the file will read from standard input. Storing this in a variable of type FILE* can be useful. See the web pages of:

- http://www.cplusplus.com/reference/cstdio/fscanf/
- http://www.cplusplus.com/reference/cstdio/fgets/
- http://www.cplusplus.com/reference/cstdio/stdin/

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)
     printf ("You can get from airport %d to airport %d in one or more flights\n", x, y);
  else
     printf ("You can not get from airport %d to airport %d in one or more flights\n", x, y);
}

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. It must also have a makefile to help with the compilation of the program. All of the storage structure code (the linked list code and the array of linked list headers) is to be in one source code file. The other code is to be in a different source file.

You must also create a header file. The job of the header file is to contain the information so the source code files can talk to each other. The header file (.h file) should contain the function prototypes and any struct and/or typedef statements. Please review the .h file in the example below.

The makefile MUST seperately compile each source code file into a ".o" file and separately link the ".o" files together into an executable file. Review the makefile in the example below to see how this is done. The command to create the .o file is:

gcc –c program1.c

The command to link the files program1.o, program2.o and program3.o into an executable file is:

gcc program1.o program2.o program3.o

The above command will just name the executable file using the default name, most often the –o option is given to provide a specific name for the executable file.

gcc program1.o program2.o program3.o –o program.exe

Example of Multiple Source Code Files

Consider the program contained in the following files:

- max3a.c
- max3b.c
- max3.h
- makefile

This example shows how to set up this simplest of multiple source code file program. Note that max3a.c and max3b.c just contain functions and a #include of max3.h. The file max3.h contains the prototypes (or forward declarations) for all of the functions that are called from outside its source code file and any "globally" needed information.

The makefile is a special file that helps in the compilation of the source code into the object files into the executable file. A makefile is executed by the use of the **make** command. The syntax of a makefile can be strange. I have always found that it is easiest to modify an existing makefile rather than trying to create one from scratch. The makefile will contain multiple rules. Each rule has the following syntax:

```
target:  dependencyList
         commandLine
```

The multiple rules in the make file are separated by a blank line. Also note (this is VERY IMPORTANT) the commandLine must use a TAB for its indentation! An example of a rule is:

```
max3a.o: max3a.c max3.h
	gcc -c max3a.c
```

The **commandLine** is **gcc -c max3a.c**, which will compile the source code file of max3a.c into the object code file of max3a.o.

The **target** in the above example is the file **max3a.o**, which is also the name of the file created when the commandLine is executed. This relationship is what causes makefiles to work.

The **dependencyList** in the above example is the two files: **max3a.c** and **max3.h**, which are the files needed for the commandLine to properly run. Again, this relationship is what causes makefiles to work.

The make command uses the timestamps of the files in the target and the dependencyList. If any file in the dependencyList has a more recent timestamp than the target file, the commandLine is executed. The idea is that if the user has recently changed either **max3a.c** or **max3.h**, then the object file **max3a.o** needs to be re-compiled. Make is designed to help the programmer keep track of what needs to be compiled next.

Make and makefile tutorials can be found at:

- http://mrbook.org/tutorials/make/
- http://www.gnu.org/software/make/manual/make.html
- http://www.opussoftware.com/tutorial/TutMakefile.htm

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.c