Programming Assignment # 4

`FakeMake`: A `make`-like program for managing precedence constraints.

Due: Tuesday, April 17 by 9:29am

In this project you will create an interactive program which mimics many of the features of the `make` program which goes back to the early days of UNIX and remains widely used. In order to do this you will manage and manipulate a graph data structure which encodes precedence relations between nodes in the graph. For information on the `make` program, see the wikipedia entry for it or just type `man make` into a UNIX/Linux shell.

Description

Your main class will be called `FakeMake`. It will take a single command line argument which will be a file containing the precedence constraints. There are two parts to the input file. First there is a list of filenames; these are "basic" files which do not depend on any other files. After this, there is a list of target files (with names) and their dependencies.

All files (basic and target) must have distinct names. Your program will have to make sure of this. The file format is as follows:

```
<file1>
<file2>
 ...
<fileN>
<target1> : <dep1> <dep2> ...
<target2> : <dep1> <dep2> ...
...
<targetK> : <dep1> <dep2> ...
```

In the above specification, there is a total of N "basic" files and K composite targets for a total of $N + K$ vertices. An example file (with 10 vertices) is as follows.

```
a
b
c
d
e
f
x : a b
y : b c d
z : e f
something : x y z
```

I have intentionally put white space between the target-name and the subsequent colon; and also between the colon and the first dependency. This is to make parsing easier.

When you read in the file, you must make sure of the following:

- No two files (targets) can have the same name.
- There can be no cycles in the specified graph.

After reading the file, you will enter an interactive mode in which you will support the commands listed below.

`time`: This command reports the current value of the clock. **Runtime:** $O(1)$.

`touch <filename>`: This command changes the time stamp on the specified filename to the current clock value is and increments the clock. This applies only to "basic" files; not targets with dependencies – an attempt to touch such a file should result in an error message (and the program should continue). **Runtime:** $O(1)$.

`timestamp <filename>`: This prints the time stamp on the specified file. This is the last time the file – a basic file **or** target – was modified. When your program begins, all files (basic and target) receive time stamps of 0. **Runtime:** $O(1)$.

`make <target>`: This command takes the target and brings it up to date. Basic files are always up to date. This is basically like the `make` command. See the text above for description. **Runtime:** $O(V + E)$.

An important issues is the notion of a "clock". The clock will be used in setting timestamps of the various files. The clock begins at time 1 and is incremented each time the clock value is assigned as the timestamp of a file.

Most of the commands should be clear but the `make` command requires some explanation. When is a file already up to date?

- A "basic" file is always up to date since it doesn't depend on any other files.
- A target is up to date if all of the files it depends on are up to date (i.e., this is a recursive definition) **and** the time stamp of the target is greater than or equal to the time stamps of the files it depends on.

When the program starts, all files and targets are assumed to have a time stamp of 0. (This implies that everything is up-to-date at the beginning – only by performing `touch` operations can things become out of date).

So, when the `make` command is invoked and the target is already up to date, the program will just report that the target is already up to date.

If it is not up to date, it brings each of the files on which it depends up to date (if necessary) and then updates the target. As files get updated, their timestamps get updated, the clock incremented and the actions are reported. As an example based on the above file, you might have something like this (assuming the program has already performed some operations):

```
> touch a
  File 'a' has been modified
> timestamp a
  10
> make something
  making x...done
```

```
  y is up to date
  z is up to date
  making something...done
> timestamp x
  11
> timestamp something
  12
> time
  13
```

Important: you will need to be careful to ensure that you don't "update" targets more than necessary. A good example to consider is:

```
a
b : a
c : b
d : b c
```

In this example, if the user invokes `make d`, you need to be careful to *not* make target `b` more than once (if that is necessary).

Rules

For this project, you *will* be allowed to use Java Platform Standard Edition `Collection` classes (e.g., `ArrayList`, `LinkedList`) and `Map` classes (probably `HashMap`). This means that you will have to create your own classes to represent the precedence graph (e.g., a `DAG`, a `Vertex` class, etc.), but that these classes may use the above Java library classes. I don't think you will need or want anything else, but if you are thinking of using an existing class but aren't sure if it is allowed, please just ask!

An exception: with respect to parsing, (input files and/or user input), virtually any existing class in the Java library is ok with me! If and when you find something really handy for parsing input, you can also freely share this information with your classmates (e.g., through the discussion board).

Appendix: A Real Makefile

```
# Makefile for a C++ project
#
# A '#' starts a comment line
# format:
# <target-name>: <list-of-dependencies>
#       <action-to-take-when-to-bring-target-up-to-date>
# Note:  targets may not be actual files (e.g., 'all' below).

all: hello

hello: main.o factorial.o hello.o
        g++ main.o factorial.o hello.o -o hello

main.o: main.cpp
        g++ -c main.cpp

factorial.o: factorial.cpp
        g++ -c factorial.cpp

hello.o: hello.cpp
        g++ -c hello.cpp

clean:
        rm -rf *o hello
```