CS310: Advanced Data Structures and Algorithms, PA0

Due: Wednesday, Sep. 14 2022, on Gradescope

Introduction

This assignment was written by Prof. Betty O'Neil (slightly adapted).

It aims to help you:

- Learn about Maps (Symbol Tables), and get experience with their JDK classes
- Think about performance and memory use
- A brief revision of Java
- Introduction to Gradescope (and hopefully to using Java on Unix/Linux)

Reading

Read S&W Section 3.1 on symbol tables (maps), Section 3.5 on applications including text indexes. Read the JDK API docs on at least Map, HashMap, and TreeMap.

Project Description

also see Suggested Steps for more information.

A simple application using JDK collection classes

The main server at your company maintains 500 terminal lines, numbered 1 through 500 inclusive. Every 10 minutes the system appends to a log file the terminal numbers and user name of the person currently logged on, one terminal number and username per line of text. (Not all terminals are in use at every moment.) A fragment of the log file looks like this:

```
  9  ALTEREGO
 12   ALIMONY
433   HOTTIPS
433   USERMGR
 12   BLONDIE
433   HOTTIPS
354   ALIMONY
```

This log file shows `HOTTIPS` was on terminal 433 twice but `USERMGR` was on that terminal at an intervening time. It also shows that `ALIMONY` was on terminal 12 at one time and terminal 354 later on. The log does not display the time at which each line was written. Your job is to write a program in Java that meets the following specifications.

The program first reads a log file into memory, storing the input data as it encounters it. **Read from a text file!** Make sure you know about absolute and relative paths and don't hard-code the file name. User names are ASCII (plain text) strings with no embedded whitespace and a maximum length of 40 characters.

After all the data has been read your program should print data about terminal usage: a header line and then one line of output for each terminal showing the terminal number, the most common user of that terminal (in the event of a tie, choose one user), and a count of how many times that user was on that terminal. Here is sample output:

```
Terminal   Most Common User   Count
1          OPERATOR           174983
2          HANNIBAL           432
3          <NONE>             0
4          SYSMGR             945
...        ...                ...
```

Implementation

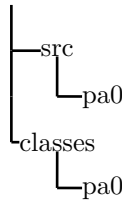
Use the provided template. You may add other functions if you want, but for the ones that I provided use the exact same function names as in the template. Since the terminals are numbered consecutively, it is easy to use an array with one spot for each terminal (500 entries in this case). Each line needs a container of user-count data, i.e. how many times each user has been seen on that line. The following class should be implemented:

- Create a class named `Usage` to hold a single username and its count together. It is quite simple. It should have getters – `getUser` and `getCount` (these names specifically). The constructor should get the name, then the number.
- Create a class named `LineUsage` to hold all the data on one particular terminal. In this `LineUsage` class, use a Map from the JDK (HashMap or TreeMap), mapping each username to its count, an Integer. The map's name should be `lines`. See the provided program `FrequencyCounter.java` for an example of using a Map to hold a count for each of many strings. `LineUsage` should have methods `addObservation(String username)` and `Usage findMaxUsage()`, which returns the `Usage` object for the user with the highest count. Note that `Usage` is not used in the map, only in the delivery of results once the Map is full of data.
- The top-level class `LineReport` has an array of `LineUsage` objects to hold all the data on all the terminal lines. See page 72 of S&W for discussion of an array of objects. `LineReport.java` should have several methods (at least two object methods or static methods), including a `main()` to run the program and a method `loadData(String fname)` which reads the data from the input file. Each method should be commented (a comment above the method to describe what it does).
- To test your code you can add a simple test called `TestLineUsage`, which tests `LineUsage`: just create one `LineUsage` object, add data to it, and print it out. **Do not submit it.**
- Since the array count starts from 0, remember to add 1 back to the index before you print.

Directory setup

In accordance with current Java programming practice, each project in this class will held in a directory system with a top level directory. We did not cover packages yet. For now, suffice to say that a java *package* is a module that groups together classes or interfaces that do similar things. Your sources should be a part of a package called `pa0`. This means they have to all reside in a directory called `pa0` (case sensitive!) and every java file has to start with the following line:
`package pa0;`

If the sources are not in a directory called pa0 or the java source files don't have a package statement is not at the top, the source files will not compile. If you do not put the java files in a package named pa0, the autograder will fail. I recommend that you start by creating a `src` directory for all Java sources, and the `pa0` subdirectory will be under `src`. If you are using an IDE, the binaries will be separately held in directory bin (or classes). The simple case (no IDE in use) will have the following directory structure:



The src directory contains the following:

- src/pa0/Usage.java
- src/pa0/LineUsage.java
- src/pa0/LineReport.java
- src/pa0/TestLineUsage.java (for testing, not submission)

Here forward slashes are used for directories. Under Windows, back slashes are used for directories, but we will consistently the Linux/Mac syntax of forward slashes: just reverse the slashes for Windows. You can create these on your home system, then transfer the whole pa0 directory system to a linux system for testing, e.g., users.cs.umb.edu. See AccessToCSHosts for file transfer instructions.

To build `LineReport`, cd to the `src` directory and use the following command, on Linux/Mac. The reason to be cd'd to this directory is that this is the easiest way to get the right java "classpath" in use, the search path for input files to the compiler (here `pa0/*.java`, remember the package discussion above) or the java command (here LineReport.class). The default classpath for java and javac is the current directory, and you've cd'd to the right place to set the desired classpath this way:

`javac -d ../classes pa0/*.java` (while cd'd to src)
`java pa0.LineReport inputfile` (while cd'd to classes. inputfile should be in your current working directory. If not, specify its path)

The -d command tells the compiler where to put the compiled .class files. For an IDE like eclipse, set up the directories as above (with src and bin or classes), put the sources in src, and then set up a project in the IDE. Eclipse can "open project from file system" once it is set up.

memo.txt

In the file `memo.txt`, answer the following questions, in one to two pages (60-120 lines) of text. I will not take off points for minor typos or spelling errors but please use standard English and complete sentences, like you would in an English class.

1. Discuss your experiences in writing these programs. What development tools (IDE, etc.) did you use? Did you develop on cs.umb.edu servers, or if on your own PC/Mac did you have any problem recompiling and running on Linux?
2. Analyze the worst case big-O CPU performance of `LineReport`, for N input lines and O(N) different users, and thus max O(N) entries on each list once partially done. To make it easier, try to first figure out what is the run time for each separate line.
3. Show the command line compile and run using commands as shown above, and display of directories by the Windows tree command or Linux/Mac `du` command.

Delivery

Before† the due time, assemble files in the pa0 (lowercase pa, number 0) subdirectory as mentioned above. Make sure they compile and run on Linux! They should, since Java is very portable. It's mainly a test that the file transfer worked OK. The following files should be uploaded to Gradescope:

- `memo.txt` (plain txt file, try "more memo.txt" on users to make sure it's readable on Linux)

- `Usage.java`
- `LineUsage.java`
- `LineReport.java`

See recorded demo for instructions how to upload files to Gradescope and what to expect from the autograder. The java file names should be exactly as instructed, or the automatic grader will not recognize them and fail. Remember that case counts on UNIX/Linux!

† When I say before I mean before. **Do not wait until the last minute!**. I will take points off for unauthorized late submissions.