Baby Names

CS 151 - Introduction to Data Structures

Assignment 3 - due Friday 2/17

In this assignment, we'll explore linked lists and more complex custom-designed classes. The complexity of this assignment increases significantly from the previous two. You are given two weeks because I think you need it. Start EARLY!

1 Input File Format

We'll be taking input from files containing lines in the following format:

```
rank,male-name,male-number,female-name,female-number
```

where the comma-separated fields have the following meanings:

`rank`	the ranking of the names in this file
`male-name`	a male name of this rank
`male-number`	number of males with this name
`female-name`	a female name of this rank
`female-number`	number of females with this name

This is the format of database files obtained from the U.S. Social Secutiry Administration of the top 1000 registered baby names. Each line begins with a rank, followed by the male name at that rank, followed by the number of males with that name, etc. Here is an example showing data from the year 2002:

```
1,Jacob,30568,Emily,24463
2,Michael,28246,Madison,21773
3,Joshua,25986,Hannah,18819
4,Matthew,25151,Emma,16538
5,Ethan,22108,Alexis,15636
6,Andrew,22017,Ashley,15342
7,Joseph,21891,Abigail,15297
8,Christopher,21681,Sarah,14758
```

```
9,Nicholas,21389,Samantha,14662
10,Daniel,21315,Olivia,14630
...
996,Ean,157,Johana,221
997,Jovanni,157,Juana,221
998,Alton,156,Juanita,221
999,Gerard,156,Katerina,221
1000,Keandre,156,Amiya,220
```

As you can see from the above, in 2002, there were 30,568 male babies named Jacob and 24,463 female babies named Emily, making them the most popular names used in that year. Similarly, going down the list, we see that there were 220 newborn females named Amiya, making it the 1000th most popular female baby name.
The entire data set contains a file for each year from 1990 to 2017, named `names1990.csv`, ... , `names2017.csv` respectively.

2 Specific Tasks

You will be building two linked lists to store the baby names found in all files, one for the male names and one for the female names. You should program the linked list data structure from scratch and you are not allowed to use Java's built-in `LinkedList`. The two linked lists should be kept in alphabetically sorted order by name.
Specifically, the program needs to be able to look up a name and report the following statistics:

1. Linked list rank - just an integer indicating the position of the name in your linked list so that we can verify your list is sorted alphabetically.
2. For each year
   (a) rank - the rank of the name that year (for that gender)
   (b) number - the number of babies given that name that year (for that gender)
   (c) percentage - the percentage of babies given that name that year (for that gender)
3. Total
   (a) rank - the rank of the name among all years (for that gender)

(b) number - the number of babies given that name among all years (for that gender)

(c) percentage - the percentage of babies given that name among all years (for that gender)

For example, for the name “Mary” (female), the following statistics should be printed for all 28 files:

```
1424

1990
Mary: 35, 8666, 0.005432

1991
Mary: 38, 8760, 0.005596

...

2017
Mary: 126, 2381, 0.001877

Total
Mary: 51, 142630, 0.003630
```

Note that not all names appear in all years. The popularity of names vary greatly from generation to generation. If a name doesn’t appear in a particular data file, you should skip printing that particular year all together.

3 Look-up via Command-line Arguments

Your program should now take command-line arguments to input zero or more files names to process.
Add flags `-m name` and `-f name`, which indicate a male name or a female name to look up, respectively. For example:
```
java Main -f Dianna names1990.csv names2000.csv
```
will print out the rank, number and percentages (as explained in Section 2) of the female name Dianna used in 1990, 2000 as well as the combined statistics of these two years, as shown below.

```
398

1990
Dianna: 594, 384, 0.000241

2000
Dianna: 847, 262, 0.000182

Total
Dianna: 675, 646, 0.000213
```

More than one name may be searched, each with the appropriate preceeding `-f` or `-m`. For example, `java Main -f Dianna -m Adam names1990.csv names2000.csv`, or `java Main -f Dianna -f Aline names1990.csv names2000.csv` are both valid inputs.
You may assume that the list of filenames is always last, i.e. the first non-flag argument you encounter is assumed to be the beginning of the list of file names. Make sure you error-check your arguments thoroughly, i.e. illegal/badly-formated options, non-existent options, etc. Remember the order of flags should not matter. Your program should behave rationally no matter how unreasonable the input or the value of flags. Simply reporting error and quitting are completely acceptable for bad input. Throwing uncaught exceptions or giving errorneous output are not. Try at the Linux commandline to see examples of flag and error handling - just experiment with say `ls` and flags.
Remember that the wildcard `*` expansion will let you specify multiple file names that fit a certain pattern easily, for example:

```
java Main -f Dianna names200*
```

will be automatically expanded to:

```
java Main -f Dianna names2000.csv names2001.csv names2002.csv
names2003.csv names2004.csv names2005.csv names2006.csv names2007.csv
names2008.csv names2009.csv
```

by the shell for you. Using this feature to test will reduce tedious typing.

4 Design Notes

You should design a `Name` class that stores all the relevant stats for a particular name. Although generic linked lists have many advantages, your code will be

simplified with non-generic linked lists that are locked to the `Name` class. This is acceptable. If however, you choose to implement generic linked lists from scratch instead, you will be awarded extra credit. Make sure you state this clearly in your README so that the TAs are aware.
Computing the yearly percentages, as well as total number and total percentage require additional auxiliary data structures besides the linked lists. Consider what you need and decide where and how to store the information carefully.
Calculating total rank is non-trivial. Think through your data structure and algorithm needs before you start. You should write the program so that it makes the best use of available storage. Resist the urge to store redundant information in many different places.
Suggested steps below. Note that for each step, you should test with one input file, then multiple input files (Just print partial contents of your linked lists), before moving onto the next one. You can hardcode the filenames in the early steps. This will get fixed once you have step 3 and beyond working.

1. Read the files into two lists of unique names in sorted order. (Your `Name` class needs to have only a `String`, for now) If you are having trouble debugging the sorted order, I suggest creating a smaller input-file (by keeping only the first 10 or 20 names) and using that instead.
2. Expand your `Name` class to provide storage for yearly number and rank. Modify your file-reading code to create and insert (in sorted order) a new `Name` object if it's not already in the list, or update it with the given yearly stats if it is.
3. Enable single name lookup on a single file (via commandline arguments)
4. Enable single name lookup on multiple (or all) files
5. Compute all the necessary totals to enable yearly percentage reporting and storing them in reasonable data structures.
6. Compute additional totals to enable total number and total percentage reporting
7. Design an algorithm to compute total rank
8. Enable multiple name lookup on multiple files

5 Testing

1. Start by testing manually. Select a few names and input files combinations to spot check.

2. Then compare your output with the two samples you see in this handout. The output for Mary isn't shown for every year, but if the three numbers for the grand totals match, then you are probably in safe hands.

3. For a bigger sanity check, look in the usual `tests/a3` in my handouts. Multiple expected output files are provided for you there, as well as a bash shell script to automate testing. Start by reading the README in the subdirectory and follow instructions there.

6 Write-up

Please include a write-up in your README that explains your class design and algorithms. In particular, address the following questions:

1. Which instance variables do you have in your `Name` class?
2. How do you keep the linked lists in alphabetically sorted order?
3. How do you organize the storage of the yearly statistics per name versus the totals?
4. Where are the overall totals stored and where are the yearly totals stored?
5. How is total rank computed?
6. If you implemented with generic linked lists, please mention it!

7 Electronic Submissions

1. README: The usual plain text file `README`

Your name:

How to compile: Leave empty if it's just `javac Main.java`

How to run it: Leave empty if it's just `java Main`

Known Bugs and Limitations: List any known bugs, deficiencies, or limitations with respect to the project specifications. Documented bugs will receive less deduction versus uncaught ones.

Write-up: Contents as discussed above

2. Source files: all `.java` files

3. Data files used: the entire `names` data file folder

DO NOT INCLUDE: Please delete all executable bytecode (`.class`) files prior to submission.

To submit, store everything (README, source files and data files) in a directory called `A3`. Then follow the directions here:
`https://cs.brynmawr.edu/systems/submit_assignments.html`