CSE 274 – Fall 2018

PROJECT #8: 90 points – Due Sunday, December 2, by 11:59 p.m.

Outcomes:

- Implement a data structure to solve a problem efficiently

Naming requirements (not following any of these results in a score of 0):

- The Eclipse project name must be **Project8**.
- You are given 4 source code files. You should submit these same 4 source code files, along with any other source code files you create. Those are:
    - **LookupInterface.java** (an interface)
    - **StudentLookup.java** (this will be your implementation of the interface)
    - **LookupJUnitTester.java** (a JUnit test to make sure your class provides the expected functionality, but does not test for efficiency)
    - **Controller.java** (what your instructor will use to perform some tests on the efficiency of your implementation)
- You are also given some data files to help you test your code with large data sets. These files are the text from several public domain books. You should test your code with small data sets by simply manually creating your own arrays. But, these large data files will be useful as you start testing the performance of your code. You do not need to add these files to your project, but you will want to copy them to the Project8 folder so that you can work with them as needed.
    - **data.zip** contains sample input files
    - **data_out.zip** contains the frequency of each word in each of the input files
- Use the **default package** (this means there should be no package statements in any of your files).

Preliminaries:

Review the data structures you have learned this semester. Review the efficiency of various implementations. Review hash table implementations, Horner's method for hashing strings, and various ways to handle collision detection (linear probing, quadratic probing, linked nodes)

Introduction:

Analyzing data is something that has widespread usage, and when you're analyzing words, you have to look at and store a lot of them. Storing data, accessing data, and storing data about data are all things you should be proficient in. However, the problem gets a lot more complicated if you want to look at large amounts of data in a timely manner. One popular tool in which this need is obvious and important is Google Trends, a site for tracking popular search terms. In Google Trends, Google takes search data and looks for patterns in that data. If you view "Top Charts" on Google Trends, you quickly learn the most popular search terms in various categories.

In this assignment, you are given a lookup interface with four methods -- methods that would be needed to implement basic functionality required for a Google Trends monitor -- that you have to implement in

a StudentLookup class. In the StudentLookup class you may implement whatever private methods or private variables you like, but the only public methods you may provide in the class are those that are required by the interface.

RESTRICTION:

For this assignment you should be implementing your own data structures. There will be a 20% penalty for using any Java data structure library *except ArrayList*. And, of course, you may use arrays. So, do not use Java's HashMap, Hashtable, Queue, and so on. In other words: you can do it if you get desperate, but the expectation is that you are writing all supporting data structures yourself. The same applies to any of the textbook implementations of data structures. The goal here is, as much as possible, to implement your own data structure.

Methods:

In testing your class we will be reading in a stream of terms (coming from a text file), feeding them to your StudentLookup object, and querying your object at various times to get certain information about the frequency of terms. For example, we might feed it a collection of words and ask what the 3rd most popular word is in the collection.

You will need to implement these methods:

- **void addString(int amount, String s)** This method is called when you see a string s, causing the count of the number of occurrences to increase by amount. So, for example, calling addString(15, "the") would increase the number of occurrences of "the" by 15.
- **int lookupCount(String s)** This method will return the number of times string s has been seen.
- **String lookupPopularity(int n)** This method returns the string that that is the nth most popular string seen. That is, if n=0 it should return the most popular string (the string with the highest count), if n=1 it should return the 2nd most popular string, etc... (Clearly, what string is the most popular might change over time as addString is called more times.) Ties should be broken alphabetically. (So if "Marvin" and "Eddie" have each been seen the most, lookupPopularity(0) should return "Eddie".)
- **int numEntries()** The number of unique entries in the table. That is, the number of different strings we have seen. (So if "Perfect" has been entered twice and "Dent" has been entered four times, and nothing else has been entered, numEntries() should return 2.)

Important things to consider:

- Which data structure will you use to store the strings that you are keeping track of? Java provides many built-in options, however, you will not be using them.
- When will you sort them? Will you keep them sorted on the fly or will you only sort them when you need them to be sorted? Or will you use some combination of both? Consider the benefits of doing all of the sorting at once versus keeping everything sorted. On one hand, values might be moving up and down a lot, which is more work, but on the other hand, if everything is already sorted, it's easier to insert something into a fully sorted list.

- How will you keep track of how many times the string has been found already? You could keep the values in a separate data structure and try to keep track of both of them, you could keep them in a single inner class, a separate class, or you could come up with some other solution? You could even use regular arrays, if you can manage the bookkeeping properly.
- Keep in mind, speed does count in this project. If your method runs properly, but moves at a slow pace, it will get a few points off. That being said, if you are exceptionally clever, you could get it to run very fast.

Extra Credit:

The first goal is to get your code working. The second goal is to get this working fast. There will be extra credit for speed. Specifically, I will run a variant of the Controller on your code. (It won't be exactly the same, but it will test for the same things.) The four fastest implementations will earn 10 extra points. The next two will earn 7 extra points, the next two will earn 5 extra points, the next two will earn 3 extra point and the next two will earn 1 extra point. **You are only eligible for the extra credit if you pass all unit tests, do not use Java data structures, and submit on time.**

Submission:

Please add all source code files in or under your src folder to the repository. This includes the code you were given, your StudentLookup.java file, and files for any other data structures you might have used. You do not need to submit the sample text files.

Scoring:

Outcome	**Max score**
Methods pass the JUnit tests	45
Methods handle data efficiently. You should be able to process all sample data files in a reasonable amount of time. If it takes longer than 2 minutes total to process all 7 data files, your code is taking too long.	45
Extra credit for being fastest	Up to 10 points
Solved the problem using your own data structures. You did not use any of Java's built-in data structures (except perhaps arrays and ArrayList). 14-point deduction for using other Java data structures or any of the textbook implementations.	0 (14-point deduction only)
Code formatted according to generally accepted standards. Note that Javadoc comments are not required because you are implementing an already documented interface.	0 (deductions only)
Code follows good programming guidelines (avoid excessively long methods, use helper methods, name variables appropriately, and so on)	0 (deductions only)