Hash Tables – Literary Comparison

**Background**

Telling the difference between two different authors has many applications. This assignment will compare the relative frequency of words in works by William Shakespeare and Francis Bacon. Before the two works can be analyzed, they will need to be stored in memory. To do this, we will use two different hashing techniques, one that utilizes chaining and the other quadratic probing.

**Overview**

You are to implement two separate hash tables, one per author, and use them to store the number of times a word appears in a text file. You will also write code that iterates through the hash tables and computes the relative frequency of words stored in each of the hash tables.

**Getting Started**

Included are:

* Two text files: **hamlet.txt** and **bacon-essays.txt**. Code will be provided to read these works into arrays and add them to your hash tables. Note: they are very long. Do not print them.
* **FileInput.java**: This class provides two functions which read text files into String arrays. readShakespeare() and readBacon() will return their corresponding text files as an array of strings, where each element of the array represents a word (in order) from the original text.
* **Test.java:** This is where you will be expected to insert the data into your hash tables and then compare similarities between the two provided documents. There are four goals to accomplish in this file.

**–** Initialize the two hash tables and insert the elements from the readShakespeare() String array into one hash table and the readBacon() String array into the other. **Keep an associated count for the number of times a word is added in the hash table.**

* + Iterate through the elements of one hash table and calculate how frequently that word occurs in both texts by using findCount() of a certain word divided by the lengths of the arrays from the readShakespeare() and readBacon() functions in FileInput. Use a squared-error approach. If a word appears in one text and not in the other, then add the square of the frequency of that word to the error. If the word appears in both texts, then find the difference between the two frequencies and add the square to the total error. For example, if Shakespeare uses a word 0.001 of the time and Bacon uses the same word 0.0001 of the time, subtract these two frequencies from each other (which would be 0.00099), square that result and add it to a “total comparative error” variable which keeps track of the sum of all such events.
  + Repeat this process for the second hash table, making sure not to duplicate any of the squared-errors you have already added to the “total comparative error” variable from above. This means you only have to consider words that are not in the first hash table when calculating for the second. As a note: the sum from both hash tables should go into the same variable.
  + Print the results. In addition to printing the final “Total comparative error”, print the word with the highest difference in frequency. For example, if Shakespeare uses the word “dog” much more frequently than Bacon, your final printed result should look like:

Total Square Error: 1.126343843E-4 Most different word: dog

The number provided here is just an example, but you should expect numbers to be much smaller than one.

* The final files will be your two hash tables: **QPHash.java** and **ChainingHash.java**. Both will be required to support the same functionality, their only difference will be how they deal with collisions in the hash. Keep in mind that the differences in hash strategies will cause differences in all of the functions listed below. These are the functions we expect your hash tables to support:
  + **Two constructors:** the first instantiates the hash table to a default size (16) and the other instantiates the hash table to an input size.
  + **insert(Object keyToAdd):** This function will add the input string into your hash table. **If the string is already in the hash table, it should increase the count of that corresponding string. If not, initialize the count to one.** Resize the tables at a load factor of 50% and 75% for probing and buckets respectively. The new length should be the first prime number greater than twice the current length for both tables.
  + **findCount(Object keyToFind):** This function will return the count for a particular String key. To be clear, this count variable will be the same as returning the number of times a particular key has been added to the hash table.
  + **getNextKey():** Every call of getNextKey() will return the next key in the hash table. This function should utilize an interior cursor to iterate through the hash table. This function will be essential when computing the squared error. The logic here is identical to making this data structure iterable but with much less work.

**–** Computing the hash codes for a string can be done by using the [insert String variable name].hashCode() system call that is part of the Java library. Do not write your own hashcode formula.

**Advice**

**–** Try to finish early. Debugging can be very time consuming.

**–** Write test cases for every method (USE JUNIT).

**–** Add a toString() to each data structure to help debug.

**–** I would add Objects that you know cause collisions (Integers come to mind - 4, 14, 24, 34 etc.) and check to see if your code responds accordingly. This approach is much more telling and shows your mistakes. Once the test cases are written you can change your code and with the click of a button the testing framework will tell you if it passes the test cases.

**–** You can test the total comparative error using two QPHashes, two ChainingHashes, one QPHash and one ChaingingHash, and lastly test it against the java HashMap. Your errors should be the same (mine were to 16 decimal places) and you should get the same word with the highest frequency difference.

**–** Reduce redundancy! Several methods do the exact same thing so write a helper method (Hint: probing in QPHash and finding a node in ChainingHash).