EECS 233 Programming Assignment #3: Hash Tables

Due November 16, 2018 before 11:59 pm (EST)

Web search engines use a variety of information to determine the most relevant documents to a query. One important factor (especially in early search engines) is the frequency of occurrences of the query values in a document. In general, one can try to answer a question of how similar or dissimilar two documents are based on the similarity of their word frequency counts (relative to the document size). A necessary step in answering these types of questions is to compute the word frequencies of all words in a document. This step requires many search operations to be done within a word database. In order to perform these search operations in an efficient manner, hash tables can be used as the data structure of this database.

In this assignment, you will write a method `wordCount` that reads a file (document) and outputs (into another file) all the words encountered in the document along with their number of occurrences. The method should have the following signature:

```
public static void wordCount(String inputFileName, String outputFileName);
```

While implementing method `wordCount`, please use a hash table with separate chaining to keep the current counts of words already encountered in the input file. Here, words are defined to be simply strings of characters between two delimiting characters, which include a space and punctuation characters.

- Assuming that something like “Father’s” is two words (“Father” and “s”, because they are separated by delimiters) is OK for our purposes.
- For simplicity, assume any derivative words to be distinct e.g. “book” and “books”, “eat” and “eating” are all considered distinct.
- Do not distinguish words that only differ in upper or lower case of their characters, e.g. “Father” and “father” correspond to the same word.
  - You can use appropriate methods of the `String` class to handle this easily (e.g. `String.toLowerCase` method).
  - To extract words from an input string, you can use `String.split()` or java class `StringTokenizer` (which is sometimes viewed as deprecated but it’s not, it’s considered a “legacy” class) to save yourself some programming.

The general procedure for obtaining word counts should include the following steps:

1. Scan in the next word
2. Search for this word in the hash table
3. If not found, insert a new entry for this word with an initial count of 1, otherwise increment the count.
4. If you’re inserting a new word, check if the hash table needs to be expanded. Hash tables should not require a (constant) table size to be provided. Therefore, **implementations that use a constant hash table size will be penalized.**

After scanning the entire file, loop through the entire hash table and create an output file that contains the list of all words and their counts (in any order you like). Please use the following format for the output file:

```
(word1,count1) (word2,count2) (word3,count3) ...
```

For example: `(father,30) (fishing,12) (aspirin,45) ...`

The program should also print out the following properties of the hash table (each separated by a comma):

- The size (capacity) of the hash table i.e. the total number of slots in the hash table
- The number of non-empty slots
- The number of items in the hash table
- The load factor (see *Additional Instructions* for more details)
- Average length of the collision lists across all non-empty hash slots. So, empty slots **do not** contribute.

For example: `64, 37, 42, 0.65625, 1.13514`

You also need a main method that accepts the names of the two files above and passes them to the `wordCount` method. Please first run your program on the provided "`toyFile.txt`" and make sure it works. Note that, **your program should work on any input plain text file, not just on the "toy" file**. In order to test your code for different input files, you can use the same input files as Programming Assignment 2. If you skipped that assignment, please refer to it for instructions on how to obtain a realistic input file.

Additional Instructions:

1. While implementing your hash table, you can use Java's `hashCode` function on strings.
   a. Your hash function will be `h = Math.abs(word.hashCode()) % tableSize`.
   b. You obviously **cannot** use built-in hash tables like `HashMap` in Java. Note that, we take the absolute value of `hashCode` because `hashCode` returns an integer which can be negative.

2. Please use separate chaining to resolve collisions in your hash table.
   a. When using separate chaining, you do not need to have `tableSize` to be a prime number.
   b. Any number will work as long as it is not a multiple of 31 (since `hashCode` of the `String` class uses 31 as a multiplier).
   c. E.g., starting with `tableSize` as a power of 2 and then doubling if you need to expand (this ensures that you do not end up with a multiple of 31).

3. In order to figure out when to expand, you can keep track of the *load factor* i.e. ratio of the number of items in the table to the current `tableSize`. When the *load factor* becomes larger than some predefined threshold (a good value could be 0.75), you can simply double the `tableSize`.
   a. This threshold value controls the tradeoff between time and space. Higher values decrease space overhead but increase the number of collisions (Therefore increasing the time to look up an item).

4. While expanding, hash value of some items may change. Hence, you would need to move those items to their updated hash slots. For example, an item which has a hash value of 5 for a table size of 16 could have a hash value of 21 (5 + 16) for a table size of 32. Thus, it would need to be moved from slot 5 to slot 21 while expanding. **You are advised to create a helper `rehash` method to perform this operation.**

5. Calculate average collision length of non-empty slots:
   a. This characterizes the expected running time of basic hash table operations (e.g., search) for an existing item.
   b. It is given by: $\frac{Total\ length\ of\ all\ collision\ chains}{Total\ number\ of\ non-empty\ slots}$
   c. This is the same thing as counting all items in the hash table and dividing by the number of slots that are not `null`.
   d. This ratio is always greater than or equal to 1. This is because number of items in the table is always greater than or equal to the number of non-empty slots.
   e. *Note:* We assume that when a hash slot is `null`, the length of the corresponding chain is 0. Whereas, when there is a single item in a slot (i.e., there are no collisions), the length of the corresponding chain is 1 (not 0). This is a good way to count because it's faster to run a search when the relevant slot is empty than when there is an item. In the former case, you simply declare failure. Whereas, in the latter case, you must compare the search key with the key of the item in the slot.

Submission:

1. Name your file ***P3_YourCaseID_YourLastName.zip*** and submit your **zip** file electronically to Canvas. This should contain:
    a. All source codes, including sufficient comments for the TA to understand what you have done (code without comments will be penalized). This must include the main method. I will run your program using command:

    ```
    java WordCounter inputFile outputFile
    ```

    Hence, please name your classes accordingly.

    b. Please provide the following outputs for the "`toyFile.txt`":
        i. An output text file named "`outputFile.txt`" that you generated which contains the word counts of all words in the provided "`toyFile.txt`".
        ii. A text file named "`hashTableProperties.txt`" that contains the properties of the corresponding hash table that are printed out by the program (such as size, number of items, load factor etc.).

Grading:

While grading your program, I will test your code both on the provided "toy" file and on a *random* text file (i.e., a "test" file). Therefore, please make sure your programs work for any plain text input files before your submission.

Task	**Points**
Implementation and design of the hash table	**30**
Output of word counts on the "toy" file	**20**
Calculate the properties of hash table on the "toy" file	**10**
Output of word counts on a "test" file	**20**
Calculate the properties of hash table on a "test" file	**10**
Style and completeness of classes & submission	**10**
	100