EECS 233 Programming Assignment #3: Hash Tables

Due: December 3, 2019 11:59pm EST

Description

Web search engines use a variety of information in determining the most relevant documents to a query. One important factor (especially in early search engines) is the frequency of occurrences of the query words in a document. In general, one can try to answer the question *"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 frequency for all words in a document, and one desirable solution to this is to use the hash table.

In this assignment, you will write a method called `wordCount` to read a file/document (the *input*) and print out into another file/document (the *output*) all the words encountered in input along with their number of occurrence and some other properties of the hash table. A set of function definition/input/output samples are provided at the end of the document.

For simplicity, we hold the following assumption:

- Any derivative words are distinct. "book" and "books", "eat" and "eating"... are all considered distinct;
- Words are defined to be simply strings of characters between two *delimiting characters**; **Father's** are consideredas two words **Farther** and **s**, because they are separated by a delimiting character **'**;
- The function is not case sensitive. **CASE** and **CaSe** and **case** are all considered as one word.

*delimiting characters**:White-space characters (' ', '\t' and '\n') and punctuation characters.

You may use java class `StringTokenizer` (which is sometimes viewed as deprecated but is actually a "legacy" class) or `String.split()` to extract words from an input string to save yourself some programming. You may also use appropriate methods (like `String.toLowerCase()`) to handle case issues easily.

To implement `wordCount()`, please use a hash table with separate chaining to keep the current counts for words you have already encountered while you are scanning the input file. Your general procedure would include the following steps:

1. Scan in the next word `next`;
2. Search if `next` is in the hash table;

3. If false, insert a new entry with `next` and its initial count of 1, increment the corresponding count otherwise;
4. If you inserted a new word, check if the hash table needs to be expanded.

After you scan the entire file, loop through the entire hash table and print out, sequentially in any order you like, the list of words and their counts. Also, report the average length of the collision lists (across all hash slots, so empty slots also contribute).

You also need the `main()` method which accepts the names of the two files above and passes them to `wordCount()`.

Additional instructions:

1. To implement your hash table, you can use Java's `hashCode` function on strings. For a word `w`, it's hash `h` will be `h = Math.abs(w.hashCode()) % tableSize`:
   - Using built-in hash tables like HashMap in Java is **prohibited**;
   - We take the absolute value of the hash because hashCode returns an int which can be negative;
   - You can also develop your own hash function if you like;
   - The output you get depends on the hash function you write;
2. Please use separate chaining to resolve collisions in your hash table:
   - With that, you do not need to make `tableSize` a prime number;
   - Any number will work as long as it is **not** a multiple of 31 (see lecture for the reason);
   - As an instance, initialize `tableSize` as a power of 2 and then double it if you need to expand the table. This will ensure that you do not have a multiple of 31;
3. `tableSize` should be adaptive depends on actual situation. **Setting `tableSize` as a constant will be penalized**;
4. In order to figure out when to expand, you can keep track of the *load factor*:
   - It is the ratio of the number of items (namely, the number of **link lists**, not the **words**, in the table) 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;
5. While expanding, hash value of some items may change. 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;
   - It should be moved from slot 5 to slot 21 while expanding;
   - **You are advised to create a helper method `rehash()` to perform this operation;**
6. You are also required to calculate the average collision length of non-empty slots:

    - This characterizes the expected running time of basic hash table operations (e.g., search) for an existing item;
    - Its value can be calculated from $\frac{Total\ length\ of\ all\ collision\ chains}{The\ number\ of\ non-empty\ slots}$
    - When a slot is empty, the length of its collision chain is 0;
    - When there is only one item in the slot, we regard the length of its collision chain as 1;
7. You are **not** supposed to use any third party libraries in this project;
8. Your project will be compiled and executed using commands:

```
javac WordCounter.java
java WordCounter inputFileName outputFileName
```

   So please make sure that you named your file(s)/class(es) properly, and write the `main()` function in the class correctly, so that it can read the two filenames from the command line arguments.

Submission:

The file you submit should:

- Be in the form of a zipped file (to be more specific, a file with **.zip** as suffix);
- Be named as ***P3_YourCaseID_YourLastName.zip*** (e.g. P3_wxs123_Skywalker.zip);
- Include all code file(s) necessary to compile your project (**10% grade for compilation!**), code should be properly commented;
- Include a file *toyOutput.txt* as corresponding output for the provided input *toyInput.txt* to show your code does work.

Grading Criteria

Task	Category	Description	Grade
1	Logic	Retrieve data from *testing files** properly	10%
2		Correctly implement hashing, rehashing, table expansion and other desired features	50%
3	Output	Pass compilation	10%
4		Being able to output as demanded as the sample provided below	10%

Task	Category	Description	Grade
5	Style	Coding style related (**proper** comment, file/class names, indentations, etc)	20%

*testing files**: A testing file might include continuous delimiting characters.

Samples

Function Definition:

```
public static void wordcount(String inputFileName, String outputFileName) {
  // Write your code here
}
```

Input(Save it as *toyInput.txt*)**:**

```
When you are old and grey and full of sleep
And nodding by the fire, take down this book
And slowly read, and dream of the soft look
Your eyes had once, and of their shadows deep
Your teaching assistant Derek is a nice person

How many loved your moments of glad grace
And loved your beauty with love false or true
But one man loved the pilgrim soul in you
And loved the sorrows of your changing face
wise, kind, warm-hearted and good at cooking

And bending down beside the glowing bars
Murmur, a little sadly, how Love fled
And paced upon the mountains overhead
And hid his face amid a crowd of stars
his roommate wrote this example to praise him
And I guess he will not notice this trick
```

Output** (For output format reference only)**:**

```
128
63
0.734375
1.4461538461538461
(when, 1) (you, 2) (are, 1) (old, 1) (and, 13) (grey, 1) (full, 1) (of, 6) (sleep, 1) (nodding, 1)
(by, 1) (the, 6) (fire, 1) (take, 1) (down, 2) (this, 3) (book, 1) (slowly, 1) (read, 1) (dream, 1)
(soft, 1) (look, 1) (your, 5) (eyes, 1) (had, 1) (once, 1) (their, 1) (shadows, 1) (deep, 1)
(teaching, 1) (assistant, 1) (derek, 1) (is, 1) (a, 3) (nice, 1) (person, 1) (how, 2) (many, 1)
(loved, 4) (moments, 1) (glad, 1) (grace, 1) (beauty, 1) (with, 1) (love, 2) (false, 1) (or, 1) (true,
1) (but, 1) (one, 1) (man, 1) (pilgrim, 1) (soul, 1) (in, 1) (sorrows, 1) (changing, 1) (face, 2)
(wise, 1) (kind, 1) (warm, 1) (hearted, 1) (good, 1) (at, 1) (cooking, 1) (bending, 1) (beside, 1)
(glowing, 1) (bars, 1) (murmur, 1) (little, 1) (sadly, 1) (fled, 1) (paced, 1) (upon, 1) (mountains,
1) (overhead, 1) (hid, 1) (his, 2) (amid, 1) (crowd, 1) (stars, 1) (roommate, 1) (wrote, 1) (example,
1) (to, 1) (praise, 1) (him, 1) (i, 1) (guess, 1) (he, 1) (will, 1) (not, 1) (notice, 1) (trick, 1)
```

*Output*** : Again, the output you get depends on the hash function you write.

Requirements:

1. Assume there are `n` words in *input*, your *output* should contain exactly **5** lines;
2. The first line is the size of your hash table (`tableSize`);
3. The second line is the number of **empty** slot(s) in your hash table;
4. The third line is the *load factor**** ;
5. The fourth line is the average length of the the collision lists across all **non-empty** hash slots;
6. The following fifth line are the words and their corresponding frequency in pairs, separated by space characters. Each pair should be in the form of `(word, frequecy)`

*load factor**** : See Additional instructions #4.