Nathan Smith

Programming Assignment Four

Dictionary Spellchecking

Due: 3/25/17

Abstract

For this assignment, we are required to store a dictionary from a file into an array of twenty-six MyLinkedLists. Then we must read in a book through a file parser to spellcheck all of the words. This is to be done by comparing the words to our dictionary. If they are found, they are spelled correctly. If they are not found, they are considered to be misspelled. Finally, we must find the average number of checks that was done to confirm a word is spelled correctly or confirm that it is not in our dictionary.

In order to load the dictionary, we use a file parser that reads in a word at a time. After changing that word to completely lowercase, we add it to the MyLinkedList at the appropriate spot in the array. This algorithm is dependent on the number of words in the input file. Further, because our MyLinkedList class holds a reference to the tail, adding at the end works like random access. This means our algorithm's time complexity is O(n), where n is the size of the dictionary. Spellchecking is more complicated. We read in the book word by word. We need to split any hyphenated words to check both sides of them. We also need to extract the left side of the words containing an apostrophe because our dictionary does not contain either of those. For example, although hat-trick and school's might not be in our dictionary, these words may be: hat, trick, school. Once we have isolated a valid word, we must check it with the appropriate MyLinkedList. Therefore, if n is the number of words in our book, and m is the length of the current MyLinkedList, the time complexity of our spell check is O(nm) because for n words, it may require m checks.

When loading the dictionary, we read words as stated previously. However, to get them to the correct alphabetical MyLinkedList, we subtract 97, the ASCII value of 'a', from the charAt(0) of our word. This will give us the array index of the appropriate list. For example, index 0 is the list of words beginning with letter a. When reading the book, we take in a word at a time, simultaneously changing it to lowercase and trimming any character that is not a-z, a hyphen, or an apostrophe. At this point, if the word is empty, we continue to the next iteration of the loop. Next, if the word contains a hyphen, we split it at the hyphen and use a for each loop. If a word in the returned array has an apostrophe, it will be trimmed to only the left side and again have all non a-z characters trimmed in case that left side has another apostrophe. Because this can return empty Strings, we must again use "continue;" if the word is empty. Finally, we check for the words in the dictionary using an overloaded contains method. This new method takes in an array of one integer that will store the number of comparisons that were done by the contains method. Then we increment the appropriate variable, either found or notFound, depending on the return of the contains method. Last, the appropriate comparison variable is supplemented by the current value of our one element array. Our second special case is when a word did not contain a hyphen, but did have an apostrophe. In that case, we use the same regular expression as before, then check the front of the word with the dictionary. Again, we update the appropriate count variables. If neither of these cases are evaluated, then we do a normal check accessing the appropriate list as described in loading the dictionary. The count variables will be updated in the same way.

Our program finds that there are about a million words being checked in the book. Over 900,000 of them are found in the dictionary. The average number of comparisons of a word not found is 7393.75 while the average number for words found is roughly half that. Words not found are basically the worst-case scenario, requiring the max number of checks. Therefore, the number of comparisons for words found makes sense because all it is saying is that on average, the average case scenario is what happens.

This program was made more difficult by the large size of the book. There were many instances of strange words that were difficult to deal with. For example, one was cal'la,te. There were a lot of ways to write our parser and we could interpret how to deal with these strange occurrences differently. The runtime is currently at about 36 seconds. It was shorter before it was written to include the special cases of hyphens and apostrophes. However, I think this is a reasonable amount of time. Finally, writing this program was really helpful in my opinion. It gave us a chance to solve a fairly large-scale problem with a good amount of freedom which forced us to use the API and discuss implementation when there isn't a completely right or wrong solution.

Outputs

```
Words found is 939275
Words not found is 51580
Number of comparisons of words found is 3342817150
Number of comparisons of words not found is 381369708
The average number of comparisons of words found is 3558.93
The average number of comparisons of words not found is 7393.75
```