Nick Tinsley
Programming Assignment 5
Spellchecker
April 11, 2017

Abstract

In assignment five, we face the problem of reading in a file, oliver.txt, and comparing words in this file with a dictionary sorted into 26 binary search trees. First, we must create 26 binary search trees, one for each letter of the alphabet, that will hold all the words starting with the same letter in one binary search tree. Then, we must read in the random_dictionary.txt file to give us something to compare words from the oliver.txt file to. We need everything to be lowercase in the dictionary, so we lowercase everything. To access a specific tree to store a word, we look at the first character of the word and using ASCII values we can get the offset for each word from 'a'. Since 'a' has an ASCII value of 97 we use this as our base and when we call the tree we will take the first character of each word's ASCII value and subtract 97 from it (tree[word.charAt(0)-97]). This gives us the binary search tree the word will be stored in and we insert it into the tree. After we have the binary search trees loaded with all the words we can start comparing words from the oliver.txt file to see if they are spelled correctly or contained in the dictionary. We must read the oliver.txt file word by word by lowercasing every word and by removing every special character. Removing the special characters uses a regex of replaceAll("[^a-z]"). This will remove all special characters. If we hit a null reference throughout the file, we skip over to the next word so an exception is not thrown. We will be using the search method from our binary search tree to see if each word is contained in the dictionary, but we will need to create an overloaded search method to check the comparisons found and not found. This method is the same as the previous search method, but we add an integer array variable(count) that allows us to increment each comparison in the client. We call the tree[word.charAt(0)-97] now in the readOliver() method, but instead of insert we call the overloaded search method. If this method returns true, it means the word was found so we increment the wordsFound counter, and we also increment comparisonsFound counter by one and add the value of 'i'. If the word is not found, returns false, we increment the wordsNotFound counter and we increment the comparisonsNotFound counter by one and add 'i'. After finding all these counts we can get the averages for comparisons for words found and comparisons for words not found. These are the average depth of the trees for words found and not found. The time complexity of Binary Search Tree is O(log n) and for the Linked List used in Assignment Four it was O(n). This gives a much faster output and is why the average comparisons for words found and not found was much lower.

```
run:
Words found 914054
Words not found 64537
Comparisons found 14950610
Comparisons not found 741543
Average number of comparisons per word found 16
Average number of comparisons per word not found 11
BUILD SUCCESSFUL (total time: 10 seconds)
```