Nathan Smith

Programming Assignment Five

Dictionary Spellchecking with BST

Due: Tuesday, April 11

Abstract

In Assignment Five we were to implement the dictionary spellchecking from Assignment Four using BinarySearchTrees instead of MyLinkedLists. The problem remains similar. We need to load the words from the dictionary file into an array of BST's. Each BST contains words from a given letter of the alphabet. Then we must spellcheck oliver.txt with our loaded dictionary. We will consider words not found in our dictionary to be spelled incorrectly. Then we must display the total number of words found and not found. We also need the average number of comparisons when checking for words found and not found and their totals.

The algorithm used loops through the 26-length array and inserts words into BST's. This insertion is done by comparing words moving to the left if they are alphabetically before and to the right otherwise, until an empty spot is found. There they are inserted in to a new TreeNode that is attached as a child to the word previous. Then we spellcheck considering three cases. Hyphens are split to have the word before and after checked. Apostrophes have the word to the left of the apostrophe checked because there are no contractions in our dictionary. Finally, if neither of these cases are present, we simply check the word read in. Our search method loops through the BST traversing left or right until we hit null, or we find a match. Then it returns true or false. We reach the appropriate BST by subtracting 97 from the first character of our test word. This will be the index in our dictionary array of the correct lettered BST.

We use four long variables to keep track of the number of words and their respective comparisons. We need these to get the average at the end. Our constructor instantiates the array of BST's. A call to the loadDictionary parses in random_dictionary.txt and fills our BST's with all the words from their specified character of the alphabet. Finally, we call the spellCheck method. This uses an array to count the number of comparisons coming from our calls to an overloaded search method. When parsing the book, we read words one by one using the hasNext() method. These words are immediately trimmed to be lower case and have only hyphens and/or apostrophes. Each token read in by that is checked and evaluated according to the cases stated above. This is done using regular expressions that return both sides of a hyphen or the left side of an apostrophe. Finally, for each case, we update the count variables depending on whether the word was found. Then we reset the array pulling the number of comparisons from the method call.

As expected, I got the same number of word found and not found. however, they took much fewer comparisons. This is because searching a BST is O(logn) in the average case. The average number of comparisons for words found was only 16.35 and 9.94 for words not found. The runtime also fell from half a minute to a little over two seconds

Initially I thought there may be an issue with the averages because the words found is higher. However, after some thought, this makes sense. Consider searching for any plural word. It will be in our dictionary and we know we'll have to traverse through its singular root as well. Essentially, words spelled correctly will go down a trail of similarly spelled words which has a higher likelihood of being longer than a trail to a misspelled word. As another example, if we search "azq", a misspelled word, it will take very few comparisons because no word starts with

azq. To conclude, misspelled words can be confirmed as misspelled, on average, much sooner than words that are correct.

This method of spellchecking was much more efficient than its LinkedList counterpart. This is due to the binary nature of BST's. Searching them takes O(logn) time as opposed to O(n). The sequential access of a LinkedList never cuts out any of the input. A move to the left or right in a BST, however, cuts out every subtree on the side not chosen. We see that it is important to consider the algorithms necessary in a program when choosing the data structure to use. In this example, we were able to perform the same task in AssignmentFour fifteen times faster.

Outputs

```
Words found is 939275

Words not found is 51580

Number of comparisons of words found is 15353444

Number of comparisons of words not found is 512486

The average number of comparisons of words found is 16.35

The average number of comparisons of words not found is 9.94

Runtime: 2.0667 seconds
```