Cody Main

Programming Assignment 5

Spell-Checker Problem (Using Binary Search Trees)

November 11, 2016

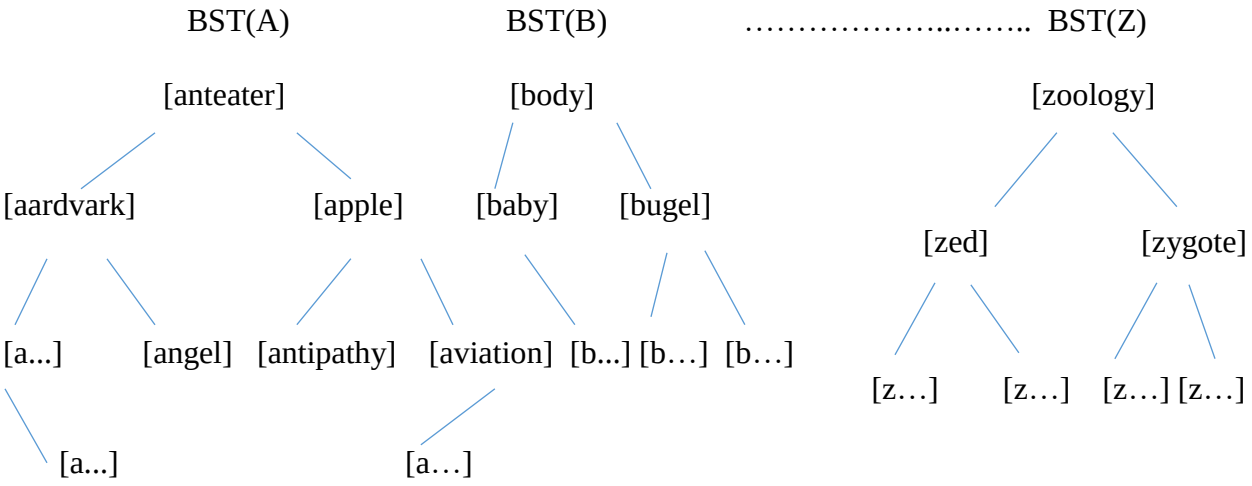
Analysis:

In this particular programming assignment, the objective is to create a Spell-Checker Program out of Binary Search Trees that will read in a dictionary file, read in a large text file, compare whether each word in the large text file matches a word in the dictionary and display counters concerning the results. These counters count the total words found, the total words not found, the total comparisons in each node taken to reach the total words found, and the total comparisons in each node taken in to reach the total words not found.

In order to successfully complete this assignment, an array of 26 Binary Search Trees representing each of the letters of the alphabet must be created. These nodes will store all the words of the dictionary according to the first letter of each word in alphabetical order. Then comparisons of each of the words of the large file will be read in. If a word in the file matches a word in the dictionary, the counters for the total words found and the total comparisons taken to reach total words found will be incremented. Otherwise, the other two counters that count the opposite situation will increment. However the averages for these two comparison counters are actually displayed. In order to achieve the averages, we take the comparisons taken to reach the total words found and divide it by the total words found. We do the same pattern for the opposite counters.

Design:

The design of the Binary Search Tree is displayed below…



Every element in these Binary Search Trees contains a set of nodes that contain all the letters of a dictionary in alphabetical order according to the very first letter of each word. This list of nodes is not sorted and is grouped only according to the first letter of each word.

Once the large file is read in, it is parsed line by line and split by the delimiters, essentially making each line an array of strings. Then it is checked to see if each letter in each string is a letter. If it is a letter, it is appended using the StringBuilder method. Otherwise, it is not appended.

Then the string is checked to see if it can be found in the dictionary according to its very first character. If it is not found in the section pertaining to its first character, then the string is not a word. The counters increment depending on the situation.

Observations:

After implementing all of the coding for this assignment, there are two observances that can be noted in the output and an important contrast to using Linked Lists for solving this problem…

1) The words found do not include the words with spelling errors. This means the program has exhaustively checked and compared all the words in the dictionary with the word being checked according to its first letter, and the all the ones that have matches are counted as words found. This means that the program has performed accurately.

2) The both the average comparisons taken to reach the words found is lower than the words found. This is because we are only displaying the average comparisons. In reality the total comparisons taken to reach the total words found is of much greater value because it happened more often. The same goes for the total comparisons taken to reach the total words not found. However, because we are taking the averages, these are far lower. This is slightly different than Programming Assignment 4 because a better method of parsing words into the reader was discovered when the program was designed. Otherwise the words found and the words not found would have had similar results to Programming Assignment 4.

3) However the comparisons taken to find the words found are dramatically less. This is because Binary Search Trees repeatedly cut the search space in half until the correct word is found. When using Linked Lists to solve this problem, the comparisons have to go through all the words leading up to the correct word until it is found. Using Binary Search Trees drastically reduces the comparisons and makes this program much more efficient.

4) Using Binary Search Trees to solve this problem remarkably increases the speed of the program. The average time to run the program using Linked Lists was around one minute and twelve seconds. However, with Binary Search Trees, the time it takes to run the program is a mere six seconds. This is because of the fact that Binary Search Trees are not linked together like Linked Lists. They do not have to travel through every node linearly until they find the correct word. This makes the time complexity of using Binary Search Trees equal to O(log n) compared to O(n) when using Linked Lists to solve this problem.

Output:

The output has been displayed below according to the assignment specifications…

```
run:

================================================

Counter Display:

================================================


Number of Words Found: 940258.0


Number of Words Not Found: 59283.0


Average Comparisons Found: 16.345462628342435


Average Comparisons Not Found: 9.666565457213704


================================================

BUILD SUCCESSFUL (total time: 6 seconds)
```