Stephanie Galvan

Date: October 23, 2019

Lab #3 – Option B Anagrams

CS 2302 – Datat Structures MW 1:30PM – 2:50PM

Instructor: Diego Aguirre

TA: Gerardo Barraza

Fall 2019

Lab 3 – Option B Anagrams

Introduction

Lab 3 implements the use of two self-balancing trees, AVL trees and Red-Black trees, in order to utilize them as a dictionary to save the words from a provided file. The trees will contain 'valid English words' which will be used to find the valid anagrams of a given word. The program will ask for user input to choose between AVL and Red-Black trees and prompt the user to choose between looking for the valid anagrams of an input word or find the word that contains the largest amount of anagrams inside the selected tree.

Proposed Solution Design and Implementation

AVL Tree:

The AVL tree class has fields root and height; the root is derived from the node class which contains key, left, and right child. The height is updated every time a new node is inserted (removal was not necessary for this lab) and the height field allows the implementation of rebalancing the tree.

Red-Black Tree:

The Red-Black tree class only has the root as a field while the node class contains the parent node, left child, right child, key, and color (as a string). The unique properties of the red-black tree required for the root to always be defined as a black node, all red nodes contain two black nodes as children, and null nodes are considered black. Whenever a property is unmet, a left or right rotation would be implemented.

Counting Anagrams:

The operation to count anagrams was divided into two different functions (one which was defined as the helper function). First, the method would prompt the user to write the word they wanted to find the valid anagrams for. Once the input was entered, the helper function was called. The helper function was called _count_anagrams and it takes four parameters: English_words (represented as a previously chosen tree), the word the user input, an initial empty list, and a prefix with an empty default value. The function is recursive and it takes at least O(n^2) to compute all possible anagrams. At every call, the function will check if an anagram is part of English_words by calling the contains method; if it is true, it would append the anagram to the list. The function will return the length of the list which represents the number of anagrams of the given word. A possible downfall for the implementation is the use of the list to count anagram instead of using a int variable; a list was implement initially in order to print the list for debugging; however, a better design would use an int instead of a list to save space memory.

Greatest Anagrams:

The greatest anagrams function is not recursive; however, it does call another function that is recursive. The greatest anagrams method receives only English_words as a parameter and it iterates through the entire tree and calls the counting anagrams method for every word inside the tree. The method does not return anything but at the end it prints the word and the occurrences of the word

with the greatest number of anagrams. The runtime of this is at least O(n^3); this could be easily improved by not calling the count anagrams method if a word is one of the anagrams of the currently greatest word.

Run-time

Counting Anagrams:

T(n) = T(n^2) + n + 6 (in which n^2 refers to finding the anagrams of word n, and the extra work refers to n (length of word) and 6 (3 is related to the base case, and the other 3 refer to the else statement

Greatest Anagrams:

T(n) = n^2*n + 7 (in which n refers to the length of English_words and n^2 refers to the runtime of counting anagrams. The additional steps can be ignored and the expected runtime is O(n^3).

Experimental Results

Each tree was needed to perform three tasks: reading a file and converting the data into a tree, counting the anagrams of a given word, and finding the word with the greatest number of anagrams (searching). In order to compare the performance between AVL and Red-Black trees, each task was performed a total of 25 times and then the average time (in both seconds and minutes) was calculated to provide the tables and graph below in figures 1.A, 1.B, and 1.C.

The numbers below reflect the time when the program follows its normal behavior. Edge cases such as reading an empty file, finding the anagram of an "empty word" (such as a blank space), and using non-alphabet characters are supported but were not considered as part of the average time to perform the operations below because the edge cases are expected to raise an exception and not a concrete answer.

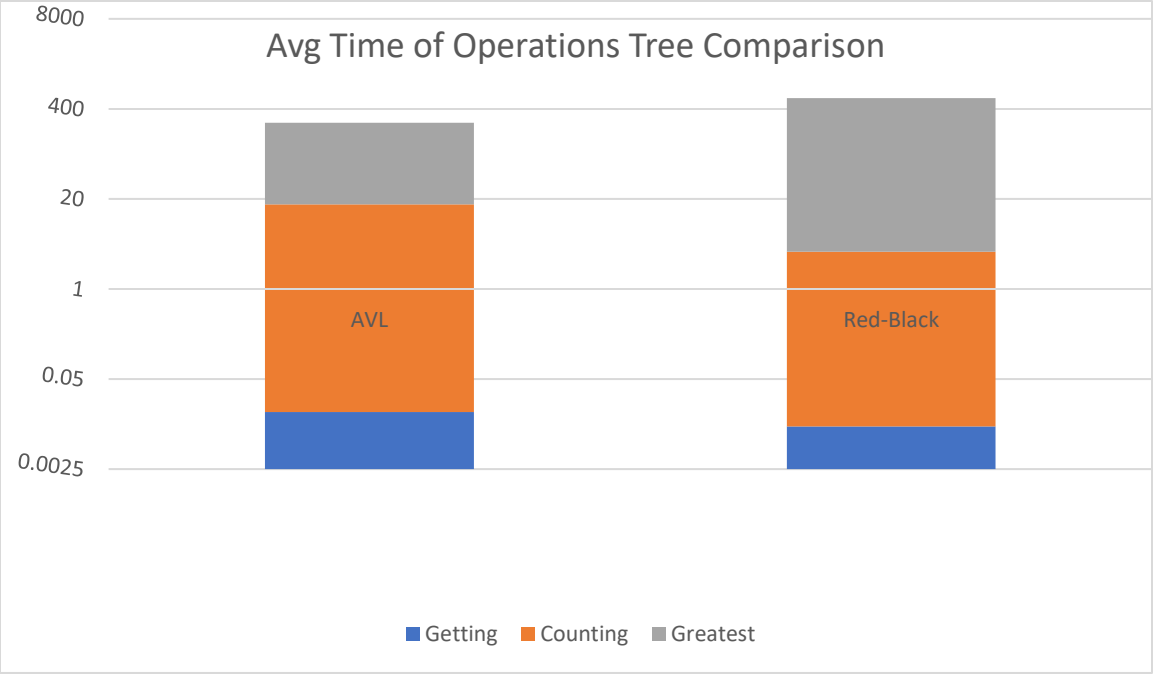
Figure 1.A: Average Time in Seconds to perform each operation per Tree

	AVL	RED-BLACK
Getting Tree	0.01667s	0.010356s
Counting	14.6624s	3.4338s
Greatest	235.609s	567.9002s

Figure 1.B: Average Time in Minutes to perform each operation per Tree

	AVL	RED-BLACK
Getting Tree	0.00277 min	0.0001726 min
Counting	0.0244 min	0.05723 min
Greatest	3.92 min	9.46 min

Figure 1.C: Average Time (Seconds) of Operations Tree Comparison Graph



Conclusion

As demonstrated on the tables and graph, both AVL and Red-Black tree have a similar run-time for implementing the tree. On average, it appears the red-black tree is faster to populate (since it takes less time), but the difference is small enough that it can be considered insignificant. However, the biggest differences lay in their performances for counting anagrams and finding the greatest anagrams of the entire tree. As it is clearly evident, the AVL tree is better for searching for the greatest anagrams of a word with an average runtime of almost 4 minutes while Red-Black is better for counting the anagrams of a given word with the average time of around 3.5 seconds. It is important to note that searching the greatest anagrams of a word takes longer time to compute than counting anagrams because searching for the greatest depends on the use of counting anagrams. The implementation for the AVL tree was more understandable and easier to follow than the red-black tree since the red-black tree requires being able to point to the parent, grandparent, uncle, and children. Ultimately, the performance of the methods implemented could be greatly improved by implementing hash sets and hash maps instead of trees; however, the lab explores the benefits and potential great performances of self-balancing binary search trees.

Appendix

Source code found at https://github.com/stephypy/CS2302_Anagrams

Code originally written in py charm.

Print Anagrams (provided):

```
def print_anagrams(english_words, word, prefix=""):
    if len(word) <= 1:
        str = prefix + word

        if str in english_words:
            print(prefix + word)
    else:
        for i in range(len(word)):
            cur = word[i: i + 1]
            before = word[0: i]  # letters before cur
            after = word[i + 1:]  # letters after cur

            if cur not in before:  # Check if permutations of cur have not been generated.
                print_anagrams(english_words, before + after, prefix + cur)
```

Honesty Certification:

I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class.

INITIALS: **SG** FULL NAME: **Stephanie Galvan** DATE: **October 23, 2019**