Isaac Acosta |CS2302 |

Lab 3 Report

Option B

# Introduction

In this lab, we will use an algorithm that finds an anagram of a given string. And the data structure implementation will be either AVL and Red-Black trees. Using a text file containing a dictionary of as many English words as possible, will be scanned and used to populate either an AVL or Red-Black tree. And with either tree, the algorithm to find a string’s anagram by traversing and searching every single anagram of a given string.

# Design and implementation

The design and implementations used for this project follow as is, for the instructed data structures an AVL and Red-Black tree structures are used. Both AVL and Red-Black tree classes get the string and convert it all to a lower case string. This is done to prevent any wrong comparisons of strings because in Python, higher case characters are lower value than lower case characters. The search functions for both classes also use lower case strings to find the desired key string. The reference passed for each search key is converted to a lower case to prevent a return of None.

The insertions process for both tree structures is very similar. Both follow the same balancing rules. The Red-Black tree follows a specific set of rules in which is balances the red and black color scheme of the entire tree depending on a node’s siblings, parent, grandparent, and uncle. Both insert and search functions have the same search() and insert() name to each so that the main program can use the same names and still function for both tree classes.

On an additional node, both classes use their own respective Node classes to construct their tree structures.

**Main Program**

In the main program, aside from the main, it uses three additional functions, populate\_tree(), greatest\_anagrams(), num\_of\_anagrams().

* populate\_tree(): In this function, an object file and a tree object is passed as reference. The referenced file is read line by line. And as the function loops through the text file, each line is inserted into the referenced tree. After the loop finalized, the function returns the tree. This method should word for either a Red-Black or AVL tree because the insert() for both tree classes are named the same
* greatest\_anagrams(): This function finds the word with the greatest number of anagrams out of a list of words in a text file. It gets a tree object and a file object as references. Within the text file, it loops line by line, and for every line the function num\_of\_anagrams() is called and if the number returned is greater than the max value, then the newly returned number from the function becomes the max and the newly read line becomes the word with maximum anagrams up to that point. And before finalizing, the function checks if the max is still zero, and if this condition returns true, “NO ANAGRAMS FOUND” is returned, or else the line referenced as being the word with the max number of anagrams and the max value is returned in conjunction. In an additional note, the referenced tree object is used to meet the requirement of having a tree object passed as reference for function num\_of\_anagrams().
* num\_of\_anagrams(): In this function, a string, tree object, integer n, and prefix is used as reference. And for a string, an arbitrary anagram is created using the prefix, and if said anagram is found in the tree, then this means that said arbitrary anagram is an actual anagram, so an increment of 1 is added to counter n. This function does this procedure for every possible anagram of a given string recursively, passing every combination of a string’s letter combination, avoiding permutations, until all proper anagrams are counted.

The main program starts by reading a text file. It then proceeds to ask the user if they wish to use an AVL or Red-Black tree. After collecting the response, engish\_words is created and depending on the user’s decision of which tree structure, engish\_words either becomes an AVL or Red-Black tree. And this is done by making engish\_words equal to greatest\_anagrams(), with the tree object passed to this function being either a black AVL tree object or Red-black tree object.

After engish\_words is a fully populated tree, the programs proceeds to find the word in a text file with the greatest number of anagrams. In this process, the program starts by reading another text file expected to contain a list of words with the purpose to find their anagrams, so the list of words should be significantly shorter than the first text file. A string value and an integer value(greatest\_word and count) are initialized. And these values are defined when function, greatest\_anagrams(), is called. Values, greatest\_word and count, are then printed out in the console.

Lastly, the program proceeds to simply find the number of anagrams of a given string just by calling num\_of\_anagrams() independently and passing a single defined string as a reference. This is done by initializing a string value, anagram\_key, and having it being an arbitrary given string. Another value, anagram\_count, is initialized and is defined when calling num\_of\_anagrams(anagram\_key, engish\_words, 0, prefix=””). Finally, this procedure ends by printing the number of anagrams for the arbitrary string, anagram\_key.

# Experimental results

These are the running times for both the AVL and Red-Black tree implementations. Aide from the text file containing the dictionary of words(words.txt), a text file containing 14 different words to run the word with greatest number of anagrams algorithm (myWords.txt).

These are the times in Seconds:

AVL implementation:

* 18.4 seconds to create tree.
* 0.04 seconds to find the greatest anagram in text file.

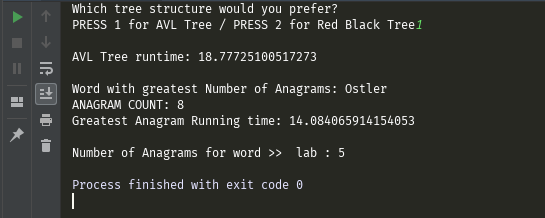
Red-Black implementation:

* 8.07 seconds to create tree.
* 0.03 seconds to find the greatest anagram in text file.

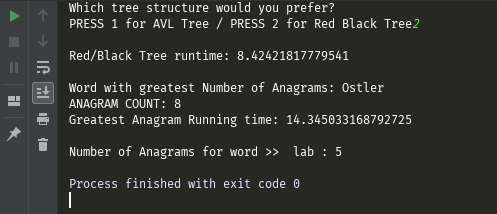
21.7/12.1 52 29.05/19.17

As it can be illustrated, the Red-Black tree was the quickest to execute in comparison to the AVL tree implementation. This could be mainly due to the structuring differences of both tree formats and the way both trees handle insertions and tree balancing.

**Screenshots**

AVL tree implementation (104 words)

Red-Black tree implementation (104 words)



# Conclusion

What I learned from this project is the differences between AVL and Red-Black trees and their different way of implementation and how with each their own they handle insertions and balancing. The amount of time it takes for each tree structure is very different from one another. It can also be learned that depending how long the dictionary text file affects how long each tree structure will take to populate. And as far as finding the anagrams, the longer the words, the more time it will take for both tree implementations to execute and finish fully.

**Academic Honesty**

“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.”