**Lab 3 – Anagram Counter**

For our third lab, we were to adapt the AVL Tree and Red-Black Tree classes from Zybooks in order to create an anagram counting program given a user-provided file of English words, a user-provided word, and the anagram printing program provided by Professor Aguirre. The program itself creates the different permutations of the user-provided word, compares them against the desired data structure, and prints the different anagrams if they are in fact in the list. For this lab, however, we were to count the number of anagrams that the user-provided word had within the data structure and return the final count. In addition, we were to find the word with the most anagrams in any given user-provided file.

Implementing this program was relatively easy, given that the Zybooks classes of AVL Trees and Red-Black Trees had all of the necessary components to implement the two data structures. The only modification in the class was in the search function, which now returned a Boolean decision rather than the memory location of the node where the “key” or desired element was located. The search algorithms were largely the same for both classes, so the same modification was made to both. With the Boolean condition replacing the return of the memory address, determining whether the user-provided word was in the list was quicker, and it made updating the anagram counter much simpler.

Since the user was to specify which data structure they preferred to use, two methods were created in order to “load” their given file into either the Red-Black Tree or the AVL Tree. Once the user chooses and the data structure is created, the “**print\_anagrams**” program was used in conjunction with the modified search methods in the AVL and RB classes, and counting the number of anagrams was as simple as printing them. A few updates in both the AVL class and RB Tree class was sufficient to get this portion of the lab up and running.

The portion of the lab that required the determining of the word with the greatest number of anagrams in a given word file was slightly more difficult. At first, figuring out how to use the first element of a given file as the first word to be checked in the anagram counting function seemed simple enough, however, it was difficult to find a way to keep the program iterating through all of the other words in the file. At first, I was trying to simply use all of the methods and counters that I already had, however, I realized that it would be necessary to open up the file reader once again to use it as an iteration tool to get through all the elements in the word file, run them through the anagram counter, and finally find the one with the largest count. All in all, with anagram printing program, AVL, and RB-Tree classes provided, only two programs were created to do the heavy lifting: **count\_anagrams** and **max\_anagram**. There were some complications with my use of a global variable (**anagramCount**) to avoid issues in counting in recursive methods, however, by resetting it to 0 in the correct places, I saw no issues with the output of my code.

The way I tested my program was by creating smaller word files that had a pre-determined word with the largest number of anagrams. For instance, after a little online research, it was determined that the words **rates** and **adrien** have 28 and 52 anagrams, respectively. I created a file with a few of the other words in the words.txt file and the words with the large number of anagrams and, as expected, those words came up as the ones with the greatest number of anagrams and it provided the correct count. Other word files were created as well with different words having the largest number of anagrams to confirm the performance of the program. The following are screen shots of certain outputs:



-Word file with the word “**adorn**” as having the greatest number of anagrams. **AVL** Tree route.



-Word file with the word “**adorn**” as having the greatest number of anagrams. **Red-Black** Tree.



-Word file with the word “**adrien**” having the most anagrams. **AVL** Tree implementation.



-Word file with the word “**adrien**” having the most anagrams. **Red-Black** Tree implementation.

As demonstrated in the screen shots, the correct output was obtained for every test case. In addition, more can be covered during demo of the lab.

The time complexity of this lab was somewhere in the **O(n\*logn)** ball park, as the creation of either the AVL tree or the RB Tree was done in that time (since the file had n lines, or n words, and insertion took about logn time, hence n\*logn). In addition, the search method for either the RB Tree or the AVL tree operates at a **O(logn)** time complexity, since they are, at the end of the day, both binary search trees. Since time complexity is determined by the fastest growing term, the over-all running time of this program is **O(n\*logn).**

I really appreciated this lab as it allowed us to get a feel of what working with existing code and programs might feel like. Given that most of us will get jobs within the industry for companies that already have a long track record of success, we will have to find ways of using existing code, modifying it, and finding ways to make it better than it was before while still adhering to certain properties. In addition, it was a pleasure not having to create our own implementations of AVL and RB Trees, since that could be rather time consuming.

**Appendix**

#!/usr/bin/env python3  
# -\*- coding: utf-8 -\*-  
"""  
File: Main.py  
Name: Angel Villalpando  
Date: 11/03/2018  
Course: CS 2302 - Data Structures  
Description: Program checks the number of anagrams for a user-specified word in a user provided file and also checks  
the file for the word with the the greatest number of anagrams.  
"""  
  
*from* RedBlackTree *import* RedBlackTree, RBTNode  
*from* AVLTree *import* AVLTree, Node  
  
anagramCount = 0 # global variable for anagram count to make counting easier in recursive methods  
  
*def* rb\_loader(*file*): # this method loads the user provided file into a RB Tree  
 rb\_Tree = RedBlackTree()  
  
 *for* line *in file*:  
 word = line.strip('\n').lower()  
 rb\_Tree.insert(word)  
 *file*.close()  
  
 *return* rb\_Tree  
  
  
*def* avl\_loader(*file*): # this method simply loads the user provided file into an AVL Tree  
 avl\_Tree = AVLTree()  
  
 *for* line *in file*:  
 word = line.strip('\n').lower()  
 avl\_Tree.insert(Node(word))  
 *file*.close()  
  
 *return* avl\_Tree  
  
  
*def* print\_Tree(*root*): # while not used in this particular lab, this method prints tree in post order  
 *if root* == *None*:  
 *return None* print(*root*.key)  
 print\_Tree(*root*.left)  
 print\_Tree(*root*.right)  
  
  
*def* count\_anagrams(*word*, *word\_list*, *prefix*=""): # this is the modified method that now counts number of anagrams  
 *global* anagramCount  
 *if* len(*word*) <= 1:  
 str = *prefix* + *word  
  
 if word\_list*.search(str):  
 anagramCount += 1  
 *else*:  
 *for* i *in* range(len(*word*)):  
 curr = *word*[i: i + 1]  
 before = *word*[0: i]  
 after = *word*[i + 1:]  
  
 *if* curr *not in* before:  
 count\_anagrams(before + after, *word\_list*, *prefix* + curr)  
  
 *return* anagramCount # while a global variable need not be returned it is done for assignment purposes  
  
  
*def* max\_anagram(*file*, *word\_list*): # method determines the words with the most anagrams given a user file  
 *global* anagramCount  
  
 usrFile = open(*file*, "r") #re-opens file to scan every element in the file and compare against data structure  
 maxCount = 0  
 count = 0  
 maxWord = ""  
  
 *for* line *in* usrFile:  
 word = line.strip('\n').lower()  
 count = count\_anagrams(word, *word\_list*) # each iteration of the file read gives count a new value  
 *if* count > maxCount:  
 maxCount = count  
 maxWord = word  
 anagramCount = 0 # global variable updated, in order to provide new count value in count\_anagrams method return  
 count = 0  
 usrFile.close()  
  
 print("Word with the most anagrams is: ",maxWord, "\nThe number of anagrams it has is: ", maxCount, "\n")  
  
  
*def* main():  
  
 *global* anagramCount  
  
 userFile = input("\n\nPlease provide the file name for anagram analysis: ") ## user file request prompt  
 file = open(userFile, "r")  
  
 usrChoice = input("\nPlease choose a Data Structure:\n1. AVL Tree\n2. Red-Black Tree\n") ## data structure prompt  
  
 *if* int(usrChoice) == 1:  
 english\_words = avl\_loader(file)  
 usrAnagram = input("\nWhat word would you like to permute and check against the AVL Tree? ")  
 print("The word '",usrAnagram,"' has a total of ", count\_anagrams(usrAnagram, english\_words), " anagrams.")  
 anagramCount = 0  
 *elif* int(usrChoice) == 2:  
 english\_words = rb\_loader(file)  
 usrAnagram = input("\nWhat word would you like to permute and check against the Red-Black Tree? ")  
 print("The word '",usrAnagram,"' has a total of ", count\_anagrams(usrAnagram, english\_words), " anagrams.")  
 anagramCount = 0  
 *else*:  
 print("Invalid Selection. Good Bye.")  
  
 usrChoice2 = input("\nWould you like to analyze the given file for the word with most anagrams? (Enter Y or N): ")  
  
 *if* usrChoice2 == "Y" *or* usrChoice2 == "y":  
 max\_anagram(userFile, english\_words)  
 *else*:  
 print("Goodbye! \n")  
  
  
main()

* Only Main.py is provided in Report. AVLTree.py and RBTree.py are in Github repository.

**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 students in the class.”

Angel Villalpando

11/04/2018