**Lab 3 Report**

The main problem that I am trying to solve is through a given AVL and Red-black classes (Zybook’s implementation) along with a print\_anagrams function (given by professor Aguirre) I have to create a function that will count the anagrams and returns the number of anagrams of a given word. Also, I have to implement another function that will read another file that contains different words and finds the word that has the greatest number of anagrams. My proposed solution is to break down the problem with simpler text files and create a method that will also read the file effectively and ask the user for which tree they will like to use either AVL or Red-black tree. In addition, in order to accomplish it I want to also make sure that no infinite loops are thrown which at the moment I have none. After inserting the methods, I have some issues because the file will not read appropriately and will thrown more errors which at the beginning, I did not understand but after going through the code I realized I was not reading line by line and that I still needed to call the methods starting by reading the file first. However, one of the most difficult parts in my opinion was when I was keeping track of the counter my math did not seem to work and had to break down the problem even less with a smaller and simpler file and create an additional method. When the program is tested with a smaller file and when it populates different English words t creates an error or simply it doesn’t not return the anagrams of that n word or neither the count of anagrams because since it was not in the main file it will only compare between whichever words are found in the file. However, when compared to a greater file or possibly a file hat has all possible English words this program can be more accurate since it will count all the anagrams on the file and create a better output without missing a word. What I learned of this project besides implementing these different methods within the lab by counting the anagrams, reading the file, and finding the greatest numbers of anagrams I learned that AVL trees take more time to run than Red Black Trees on this specific anagram example. However, AVL trees provide faster lookups than RB trees because they are more strictly balanced while RB trees provide faster insertion and removal as fewer rotations are done due to relatively relaxed balancing.

# CS2302 Data Structures  
# Programmed by Luis Garcia.  
# Last modified October 22, 2018.  
# Instructor Diego Aguirre.  
# Implementation of AVL Tree and Red black tree with the purpose of reading  
# a file called words.txt and return the anagrams of n word.  
# In addition there will be a method that will count the number of anagrams  
# of each anagram and the possible outcomes based on the file read.  
# lab3  
  
# Zybooks implementation  
class avlNode: # Class for the avl-tree node  
  
 def \_\_init\_\_(self, value):  
 self.value = value  
 self.left\_child = None  
 self.right\_child = None  
 self.parent = None  
 self.height = 0  
  
 def get\_balance(self):  
 left\_height = -1  
 if self.left\_child is not None:  
 left\_height = self.left\_child.height  
 right\_height = -1  
 if self.right\_child is not None:  
 right\_height = self.right\_child.height  
 return left\_height - right\_height  
  
 def update\_height(self):  
 left\_height = -1  
 if self.left\_child is not None:  
 left\_height = self.left\_child.height  
 right\_height = -1  
 if self.right\_child is not None:  
 right\_height = self.right\_child.height  
 self.height = max(left\_height, right\_height) + 1  
  
 def set\_child(self, which\_child, child):  
 if which\_child != "left" and which\_child != "right":  
 return False  
 if which\_child == "left":  
 self.left\_child = child  
 else:  
 self.right\_child = child  
 if child is not None:  
 child.parent = self  
 self.update\_height()  
 return True  
  
 def replace\_child(self, current\_child, new\_child):  
 if self.left\_child is current\_child:  
 return self.set\_child("left", new\_child)  
 elif self.right\_child is current\_child:  
 return self.set\_child("right", new\_child)  
 return False  
# Zybook's  
  
class AVLTree:  
  
 def \_\_init\_\_(self):  
 # Constructor to create an empty AVLTree. There is only  
 # one data member, the tree's root Node, and it starts  
 # out as None.  
 self.root = None  
  
 def get\_height(self, current):  
 if current is None: return 0  
 return current.height  
  
 def insert(self, value): # Insertion method for the avl tree  
 node = avlNode(value)  
 if self.root is None: # If the tree is empty, new node is the root  
 self.root = node  
 self.root.parent = None  
 return  
 cur = self.root  
 while cur is not None: # Insertion as a standard binary search tree  
 if node.value < cur.value:  
 if cur.left\_child is None:  
 cur.left\_child = node  
 node.parent = cur  
 cur = None  
 else:  
 cur = cur.left\_child  
 else:  
 if cur.right\_child is None:  
 cur.right\_child = node  
 node.parent = cur  
 cur = None  
 else:  
 cur = cur.right\_child  
 node = node.parent  
 while node is not None: # Rebalance from the new node's parent up  
 self.rebalance(node)  
 node = node.parent  
  
 def rebalance(self, node): # Method to rebalance an avl tree  
 node.update\_height()  
 if node.get\_balance() == -2:  
 if node.right\_child.get\_balance() == 1:  
 self.right\_rotate(node.right\_child)  
 return self.left\_rotate(node)  
 elif node.get\_balance() == 2:  
 if node.left\_child.get\_balance() == -1:  
 # Double rotation case  
 self.left\_rotate(node.left\_child)  
 return self.right\_rotate(node)  
 return node  
  
 def right\_rotate(self, node): # Right rotation for the avl tree  
 left\_right\_child = node.left\_child.right\_child  
 if node.parent is not None:  
 node.parent.replace\_child(node, node.left\_child)  
 else:  
 self.root = node.left\_child  
 self.root.parent = None  
 node.left\_child.set\_child('right', node)  
 node.set\_child('left', left\_right\_child)  
 return node.parent  
  
 def left\_rotate(self, node): # Left rotation for the avl tree  
 right\_left\_child = node.right\_child.left\_child  
 if node.parent is not None:  
 node.parent.replace\_child(node, node.right\_child)  
 else:  
 self.root = node.right\_child  
 self.root.parent = None  
 node.right\_child.set\_child("left", node)  
 node.set\_child("right", right\_left\_child)  
 return node.parent  
  
 def search(self, value): # Search method for the avl tree  
 temp = self.root  
 while temp is not None:  
 if temp.value == value:  
 return True  
 elif temp.value < value:  
 temp = temp.right\_child  
 else:  
 temp = temp.left\_child  
 return False  
  
# Zybooks implementation of Red Black Trees  
# Class node for Red Black trees  
class RBTNode:  
  
 def \_\_init\_\_(self, key, parent, is\_red=False, left=None, right=None):  
 self.key = key  
 self.left\_child = left  
 self.right\_child = right  
 self.parent = parent  
  
 if is\_red:  
 self.color = "red"  
 else:  
 self.color = "black"  
  
 def both\_children\_black(self): # Method that returns true if both children are black  
 if self.left\_child is not None and self.left\_child.is\_red():  
 return False  
 if self.right\_child is not None and self.right\_child.is\_red():  
 return False  
 return True  
  
 def count(self):  
 count = 1  
 if self.left\_child is not None:  
 count += self.left\_child.count()  
 if self.right\_child is not None:  
 count += self.right\_child.count()  
 return count  
  
 def get\_grandparent(self): # Method that returns the grandparent of given node  
 if self.parent is None:  
 return None  
 return self.parent.parent  
  
 def get\_siblings(self): # Method that returns the node's sibling  
 if self.parent is not None:  
 if self is self.parent.left\_child:  
 return self.parent.right\_child  
 return self.parent.left\_child  
 return None  
  
 def get\_uncle(self): # Method that returns the node's uncle  
 grandparent = self.get\_grandparent()  
 if grandparent is None:  
 return None  
 if grandparent.left\_child is self.parent:  
 return grandparent.right\_child  
 return grandparent.left\_child  
  
 def is\_black(self): # Checks if node is black  
 return self.color == "black"  
  
 def is\_red(self): # Checks if node is red  
 return self.color == "red"  
  
 def replace\_child(self, current\_child, new\_child):  
 if self.left\_child is current\_child:  
 return self.set\_child("left", new\_child)  
 elif self.right\_child is current\_child:  
 return self.set\_child("right", new\_child)  
 return False  
  
 def set\_child(self, which\_child, child):  
 if which\_child is not "left" and which\_child is not "right":  
 return False  
 if which\_child == "left":  
 self.left\_child = child  
 else:  
 self.right\_child = child  
 if child is not None:  
 child.parent = self  
 return True  
  
# Zybooks implementation of Red Black Trees  
# Class of Red Black Trees  
class rb\_tree(object):  
  
 def \_\_init\_\_(self):  
 self.root = None  
  
 def insert(self, key):  
 new\_node = RBTNode(key, None, True, None, None)  
 self.insert\_node(new\_node)  
  
 def insert\_node(self, new\_node):  
 if self.root is None: # If the root is empty  
 self.root = new\_node  
 else:  
 temp = self.root  
 while temp is not None:  
 if new\_node.key < temp.key:  
 if temp.left\_child is None:  
 temp.set\_child("left", new\_node)  
 break  
 else:  
 temp = temp.left\_child  
 else:  
 if temp.right\_child is None:  
 temp.set\_child("right", new\_node)  
 break  
 else:  
 temp = temp.right\_child  
  
 new\_node.color = "red" # Set to red  
 self.insertion\_balance(new\_node) # Balance with the new node  
  
 def insertion\_balance(self, new\_node):  
 if new\_node.parent is None: # If the bode is tree's root, color black  
 new\_node.color = "black"  
 return  
 if new\_node.parent.is\_black(): # If parent is black then we return  
 return  
 parent = new\_node.parent # Saving parent, grandparent, and uncle node  
 grandparent = new\_node.get\_grandparent()  
 uncle = new\_node.get\_uncle()  
 if uncle is not None and uncle.is\_red(): # If parent and uncle red, change to black, color grandparent red  
 parent.color = uncle.color = "black"  
 grandparent.color = "red"  
 self.insertion\_balance(grandparent)  
 return  
  
 if new\_node is parent.right\_child and parent is grandparent.left\_child:  
 self.left\_rotate(parent)  
 new\_node = parent  
 parent = new\_node.parent  
  
 elif new\_node is parent.left\_child and parent is grandparent.right\_child:  
 self.right\_rotate(parent)  
 new\_node = parent  
 parent = new\_node.parent  
  
 parent.color = "black"  
 grandparent.color = "red"  
 if new\_node is parent.left\_child:  
 self.right\_rotate(grandparent)  
 else:  
 self.left\_rotate(grandparent)  
  
 def right\_rotate(self, selected\_node): # Method that performs a right rotation  
 left\_right\_child = selected\_node.left\_child.right\_child  
 if selected\_node.parent is not None:  
 selected\_node.parent.replace\_child(selected\_node, selected\_node.left\_child)  
  
 else:  
 self.root = selected\_node.left\_child  
 self.root.parent = None  
 selected\_node.left\_child.set\_child("right", selected\_node)  
 selected\_node.set\_child("left", left\_right\_child)  
  
 def left\_rotate(self, selected\_node): # Method that performs a left rotation  
 right\_left\_child = selected\_node.right\_child.left\_child  
 if selected\_node.parent is not None:  
 selected\_node.parent.replace\_child(selected\_node, selected\_node.right\_child)  
  
 else:  
 self.root = selected\_node.right\_child  
 self.root.parent = None  
 selected\_node.right\_child.set\_child("left", selected\_node)  
 selected\_node.set\_child("right", right\_left\_child)  
  
 def search(self, key):  
 temp = self.root  
 while temp is not None:  
 if temp.key == key:  
 return True  
 elif key < temp.key:  
 temp = temp.left\_child  
 else:  
 temp = temp.right\_child  
 return False  
  
  
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''  
  
# Method given by profeesor  
def print\_anagrams(word, prefix=""): # Method given to us by lab  
 if len(word) <= 1:  
 str = prefix + word  
  
 if english\_words.search(str):  
 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(before + after, prefix + cur)  
  
  
  
# Returns the number of anagrams that a given word has.  
# For example, count\_anagrams(”spot”) should return 7  
# This method counts the number of anagrams that a word contains.  
def count\_anagrams( word, prefix=""):  
 if len(word) <= 1: # base case if the word has only 1 letter  
 # then it can not have more than 1 anagram ; itself.  
 str = prefix + word  
 if english\_words.search(str):  
 return 1  
 else:  
 return 0  
 else:  
 counter = 0  
 for i in range(len(word)):  
 current = word[i: i + 1]  
 before = word[0: i]  
 after = word[i + 1:]  
 if current not in before:  
 counter += count\_anagrams(before + after, prefix + current)  
 return counter  
  
'''def count\_anagrams( word, prefix=""):  
 if len(word) <= 1: # base case if the word has only 1 letter  
 # then it can not have more than 1 anagram ; itself.  
 str = prefix + word  
 if english\_words.search(str):  
 return 1  
 else:  
 return 0  
 else:  
 counter = 0  
'''  
  
# The following method searches for the word in the file that has the greatest number of anagrams  
def greatestAnagrams(file\_name):  
 file = open(file\_name, "r")  
 max\_count = 0  
 max\_word = ""  
  
 for line in file: # For every line in the file  
 count = count\_anagrams(line[0:-1])  
 print("The word ", line[0:-1], " has:", count, " anagrams")  
 if count > max\_count:  
 max\_count = count  
 max\_word = line[0:-1]  
 print("The word that has the greatest amount of anagrams is ", max\_word, " ", " ", max\_count)  
  
  
# askUser asks the user to choose between an AVL Tree or Red Black-Tree and does so by  
# asking user to type in each specific tree, once a tree is chosen then the function begins to populate the chosen tree from the txt file provided  
def main():  
 def askUser(type, filename):  
 global english\_words  
 file = open(filename)  
 # If user's input matches AVL then an AVl tree is created for chosen tree  
 if type == 'AVL':  
 english\_words = AVLTree()  
 for line in file:  
 english\_words.insert(line[:-1].lower())  
 print("AVL Tree has been created.")  
 # If user's input matches Red Black then a Red Black tree is created for user  
 elif type == 'Red Black':  
 english\_words = rb\_tree()  
 for line in file:  
 english\_words.insert(line[:-1].lower())  
 print("Red-Black Tree has been created.")  
 # If user does not input one of the options then it throws and error and makes sure that they choose between "AVL" or "Red Black"  
 else:  
 print("Please choose between AVL or Red Black")  
 return False  
 return True  
  
  
 userInput = input("Please choose between AVL or RB Tree (Enter AVL for AVL or Enter Red Black for RB Tree)\n")  
 nameofFile = "words.txt"  
 askUser(userInput, nameofFile)  
 print()  
 word = input("Type a word that you will like to analyze with an anagram\n")  
 print\_anagrams(word.lower())  
 print("Greatest possible anagram for ", word, " are/is:", count\_anagrams(word.lower()))  
 print()  
 greatestAnagrams(nameofFile)  
  
main()

“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 class.”