LAB #3 Report

***TASK***:

Write another function called count\_anagrams that does not produce output, but returns the number of anagrams that a given word has. For example, count\_anagrams(”spot”) should return 6. Finally, write another function that reads another file that contains words (feel free to create it yourself) and finds the word in the file that has the greatest number of anagrams.

def print\_anagrams(word, prefix=""):

if len(word) <= 1:

str = prefix + word

if str in engish\_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(before + after, prefix + cur)

The method uses a data structure called engish\_words to determine if a given anagram is a valid word in the English language. You can think of engish\_words as a container of all the words in the English language. We will implement this data structure using a binary search tree. To populate engish\_words, we will use a text file called words.txt that contains 354,984 English words. You can download words.txt from the following URL: <https://github.com/dwyl/english-words/>. Once you have downloaded words.txt, write a function that reads the file and populates the binary search tree with all the English words contained in the file. Ask the user what type of binary search tree he/she wants to use (AVL Tree or Red-Black Tree). You are free to use the implementation provided in your zyBook for these two types of trees. Adapt zyBook’s code to include the word and make it the key.

Write another function called count\_anagrams that does not produce output, but returns the number of anagrams that a given word has. For example, count\_anagrams(”spot”) should return 6. Finally, write another function that reads another file that contains words (feel free to create it yourself) and finds the word in the file that has the greatest number of anagrams.

***APPROACH:***

To accomplish this lab, I first had to investigate and understand the correct usage of AVL and Black-Red trees. Once I correctly understood how to use the trees, the lab was quite straight-forward. I then implemented the methods to how the lab is required to have; since the anagram algorithm was provided, I just needed to implement it by reading the file.

class AVL:

count = 0

root = None

def \_\_init\_\_(self,root = None):

self.root = root

# Gets tree hieght

@staticmethod

def get\_TreeHeight(root):

if root is None:

return 0

Lh = root.get\_TreeHeight(root.left)

Rh = root.get\_TreeHeight(root.right)

if Lh > Rh :

return Lh + 1

else:

return Rh + 1

# Updates items to AVL

@staticmethod

def AVL\_Update(nodeData):

leftH = -1

if nodeData is not None:

leftH = nodeData.height

rightH = -1

if nodeData.right is not None:

rightH = nodeData.right.height

nodeData.height = max(leftH,rightH) + 1

# Gets hieght of tree

@staticmethod

def getHeight(root):

H = root.get\_TreeHeight(root)

return H

def AVL\_Rebalance(self,nodeData):

self.AVL\_Update(nodeData)

if self.AVL\_Update(nodeData) == -2:

self.AVLtreeRotateR(nodeData.right)

return self.AVLtreeRotateL(nodeData)

def AVL\_Set(self,parent,WhichChild,child):

if WhichChild != "left" and WhichChild != "right":

return False

if WhichChild == "left":

parent.left = child

else:

parent.right = child

if child != None:

child.parent = parent

def AVL\_Replace(parent,currChild,newChild):

c = AVL()

if parent.left is currChild:

return c.AVL\_Set(parent,"left", newChild)

elif(parent.right is currChild):

return c.AVL\_Set(parent,"right", newChild)

return False

# Rotate Right

def AVL\_RRotate(self,nodeD):

newRoot = nodeD.left

temp = newRoot.right

newRoot.right = nodeD

nodeD.left = temp

nodeD.height = 1 + max(self.getHeight(nodeD.left),self.getHeight(nodeD.right))

newRoot.height = 1 + max(self.getHeight(newRoot.left),self.getHeight(newRoot.right))

return newRoot

# Left Rotate

def AVL\_LRotate(self,node):

newRoot = node.right

temp = newRoot.left

newRoot.left = node

node.right = temp

node.height = 1 + max(self.getHeight(node.left),self.getHeight(node.right))

newRoot.height = 1 + max(self.getHeight(newRoot.left),self.getHeight(newRoot.right))

return newRoot

def AVLinsert(self,data):

new = Node(data)

if self.root is None:

self.root = new

self.root.left = None

self.root.right = None

new.height = new.height + 1

return

#curr = root

new.height = new.height + 1

currNode = self.root

while currNode is not None:

if new.item < currNode.item:

if currNode.left is None:

currNode.left = new

new.parent = currNode

currNode = None

else:

currNode = currNode.left

else:

if currNode.right is None:

currNode.right = new

new.parent = currNode

currNode = None

else:

currNode = currNode.right

new = new.parent

while new is not None:

self.AVL\_Rebalance(new)

new = new.parent

# return self.root

class RedBlacktree:

root = None

#count = 0

height = 0

def \_\_init\_\_(self,root = None):

self.root = root

self.color = "black"

@staticmethod

def RedBlack\_HieghtUpdate(nodeData):

leftH = -1

if nodeData is not None:

leftH = nodeData.height

rightH = -1

if nodeData.right is not None:

rightH = nodeData.right.height

nodeData.height = max(leftH,rightH) + 1

# insert the item in the corresponding place

def RBInsert(self, data):

newItem = Node(data)

if self.root is None:

self.root = newItem

self.root.color = "black"

newItem.height = newItem.height + 1

return

newItem.height = newItem.height + 1

currNode = self.root

while currNode is not None:

if newItem.item < currNode.item:

if currNode.left is None:

currNode.left = newItem

newItem.parent = currNode

currNode = None

else:

currNode = currNode.left

else:

if currNode.right is None:

currNode.right = newItem

newItem.parent = currNode

currNode = None

else:

currNode = currNode.right

newItem.color = "red"

self.RedBlackB

while newItem is not None:

self.AVLTreeRebalance(newItem)

newItem = newItem.parent

def RedBlack\_Set(self,parent,WhichChild,child):

if WhichChild != "left" and WhichChild != "right":

return False

if WhichChild == "left":

parent.left = child

else:

parent.right = child

if child != None:

child.parent = parent

def RedBlack\_Replace(parent,currChild,newChild):

c = AVL()

if parent.left is currChild:

return c.AVL\_Set(parent,"left", newChild)

elif(parent.right is currChild):

return c.AVL\_Set(parent,"right", newChild)

return False

#Rotate left

def RedBlack\_LRotate(self,nodeData):

rightLChild = nodeData.right.left

if nodeData.parent is None:

self.RedBlack\_Replace(nodeData.parent,nodeData,nodeData.right)

else:

self.root = nodeData

self.root.parent = None

self.RedBlack\_Set(nodeData.right,"left",nodeData)

self.RedBlack\_Set(nodeData,"right",rightLChild)

#Rotate Right

def RedBlack\_RRotate(self,nodeData):

leftRChild = nodeData.left.right

if nodeData.parent is not None:

self.RedBlack\_Replace(nodeData.parent,nodeData,nodeData.right)

else:

self.root = nodeData.right

self.root.parent = None

self.RedBlack\_Set(nodeData.left,"right",nodeData)

self.RedBlack\_Set(nodeData,"left",leftRChild)

@staticmethod

def RedBlack\_getGParent(node):

if node.parent is None:

return None

return node.parent.parent

@staticmethod

def RedBlack\_GetUncle(node):

grandparent = None

if node.parent is not None:

grandparent = node.parent.parent

if grandparent is None:

return None

if grandparent.left is node.parent:

return grandparent.right

else:

return grandparent.left

# Creates a balanced

def BalanceRedBlackTree(self,nodeData):

if nodeData.parent is None:

nodeData.color = "black"

return

if nodeData.parent.color == "black":

return

parent = nodeData.parent

grandparent = self.RedBlack\_getGParent(nodeData)

uncle = self.RedBlack\_GetUncle(nodeData)

if uncle is not None and uncle.colot == "red":

parent.color = uncle.color = "black"

self.BalanceRedBlackTree(grandparent)

return

if nodeData is parent.right and parent is grandparent.left:

self.RedBlack\_RRotate(parent)

nodeData = parent

parent = nodeData.parent

elif nodeData is parent.left and parent is grandparent.right:

self.RedBlack\_RRotate(parent)

nodeData = parent

parent = nodeData

parent.color = "black"

grandparent.color = "red"

if nodeData is parent.left:

self.RedBlack\_RRotate(grandparent)

else:

self.RedBlack\_LRotate(grandparent)

# Search

def BSTSearch(word):

c = AVL()

if c.root is None or c.root == word: # if the word its on root return true

return True

if c.root.item < word:

return c.BSTSearch(c.root.right,word)

if c.root.item > word:

return c.BSTSearch(c.root.left,word)

return False# if not false

2 #print all possible combinations

def print\_anagrams(word, prefix=""):

counter = 0

if len(word) <= 1:

str = prefix + word

if BSTSearch(str):

counter += 1

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 cur has not been generated.

print\_anagrams(before + after, prefix + cur)

# AVLTreeMethod read the text file line by line and insert it to the corresponding

# tree

def AVLTreeMethod(UserInput,file,word):

call = AVL()

line = file.readline()

while line:

line = file.readline()

#root =

call.AVLinsert(line)

# call.height = call.getHeight(root)

file.close()

print\_anagrams(word)

# RedBlackTree reads the the text file and insert it each value in the tree

def RedBlackTreeMethod(UserInput,file,word):

call = RedBlacktree()

line = file.readline()

while line:

line = file.readline()

call.RBInsert(line)

file.close()

print\_anagrams(word)

def main():

file = open("words.txt")

print("please choose an option:")

print("A.AVLTree")

print("B.RBlackTree")

# if not tree.remove\_key(30):

# print("\*\*\* Key not found. Tree is not changed. \*\*\*")

# print(tree)

UserInput= input()

word = input("input a word you want to find :")

if UserInput == "A" or UserInput == "a":

AVLTreeMethod(UserInput,file,word)

if UserInput == "B" or UserInput == "b":

RedBlackTreeMethod(UserInput,file,word)

main()