CS 2302

Lab 3.B Report

Author: David Ayala

ID: 80617838

Instructor: Diego Aguirre

TA: Gerardo Barraza

**Introduction**

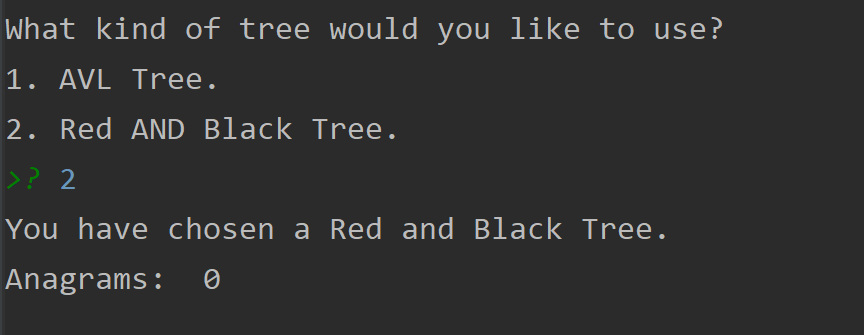
The purpose of this lab was to write a function that reads the file and populates the binary search tree with all the English words contained in the file. Ask the user what type of binary search tree he/she wants to use (AVL Tree or Red-Black Tree). Write another function called count\_anagrams that does not produce output, but returns the number of anagrams that a given word has. 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.

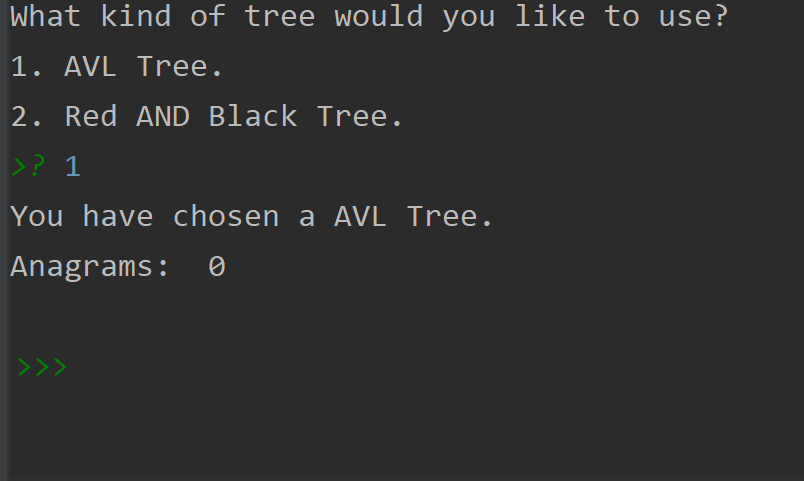
**Proposed Solution & Design Implementation**

So I understood the basic idea about how to do it, but I was unable accomplish this due to either me being unable to traverse the trees or maybe my search function just doesn’t work. The other problem could be that I really didn’t populated the tree or did so incorrectly. ultimately I keep trying how this lab until I had to finally just submit it

**Experimental Results**

So I don’t have any but here is what my program prints.





**Conclusion**

I tried to this lab for long periods of time but I also had to study for exams this week, so my time was very split between this lab and exams. I think that both things suffered and that I need to manage my time better so that I will be able to accomplish both next time. The bottom line is that I really tried to do this lab but I was unable to accomplish it so I will need to review zybooks and other resources that I may be able to this lab again later for my own improvement.

**Appendix**

# Course:CS 2302 MW 1:30-2:50, Author:David Ayala

# Assignment:Lab #3, Instructor: Diego Aguirre

# Teaching Assistant: ????, Date of last Modification: 10/14/2019

# Purpose of program: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).

# Write another function called count\_anagrams that does not produce output,

# but returns the number of anagrams that a given word has.

# 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.

from AVL import AVLNode

from AVL import AVLTree

from RedBlackT import RBT

from RedBlackT import RBTNode

def populateRBT(tempFile):

pop = RBT()

with open(tempFile) as file:

for currentLine in file:

temp = currentLine.split('\n')

pop.insert(temp[0])

countAnagrams(pop, 'spot')

def populateAVL(tempFile):

pop = AVLTree()

file = open(tempFile, 'r')

for currentLine in file:

temp = currentLine.split('\n')

node = AVLNode(temp[0])

pop.insert(node)

countAnagrams(pop, 'spot')

def countAnagrams(tree, word, populated=True):

listOfAnagram = []

if populated:

print('Anagrams: ', len(listOfAnagram))

return listOfAnagram

def main():

tempFile = 'words.txt'

print('What kind of tree would you like to use?')

print('1. AVL Tree.')

print('2. Red AND Black Tree.')

userInput = input()

if userInput is '1':

print('You have chosen a AVL Tree.')

populateAVL(tempFile)

elif userInput is '2':

print('You have chosen a Red and Black Tree.')

populateRBT(tempFile)

else:

print('enter either 1 or 2 for the respective tree.')

print()

main()

main()mat(time.time() - startTime))

class AVLNode:

def \_\_init\_\_(self, key):

self.key = key

self.parent = None

self.left = None

self.right = None

self.height = 0

def set(self, currentSide, child):

if currentSide != 'left' and currentSide != 'right':

return False

if currentSide == 'left':

self.left = child

else:

self.right = child

if child is not None:

child.parent = self

self.updateHeight()

return True

def balance(self):

heightOfLeft = -1

if self.left is not None:

heightOfLeft = self.left.height

heightOfRight = -1

if self.right is not None:

heightOfRight = self.right.height

return heightOfLeft - heightOfRight

def updateHeight(self):

heightOfLeft = -1

if self.left is not None:

heightOfLeft = self.left.height

heightOfRight = -1

if self.right is not None:

heightOfRight = self.right.height

self.height = max(heightOfLeft, heightOfRight) + 1

def replace(self, current, new):

if self.left is current:

return self.set('left', new)

elif self.right is current:

return self.set('right', new)

return False

class AVLTree:

def \_\_init\_\_(self):

self.root = None

def insert(self, node):

if self.root is None:

self.root = node

node.parent = None

else:

current = self.root

while current is not None:

if node.key < current.key:

if current.left is None:

current.left = node

node.parent = current

current = None

else:

current = current.left

else:

if current.right is None:

current.right = node

node.parent = current

current = None

else:

current = current.right

node = node.parent

while node is not None:

self.calibrate(node)

node = node.parent

def rotateToTheleft(self, node):

rightOfLeft = node.right.left

if node.parent is not None:

node.parent.replace(node, node.right)

else:

self.root = node.right

self.root.parent = None

node.right.set('left', node)

node.set('right', rightOfLeft)

return node.parent

def rotateToTheRight(self, node):

leftOfRight= node.left.right

if node.parent is not None:

node.parent.replace(node, node.left)

else:

self.root = node.left

self.root.parent = None

node.left.set('right', node)

node.set('left', leftOfRight)

return node.parent

def calibrate(self, node):

node.updateHeight()

if node.balance() == -2:

if node.right.balance() == 1:

self.rotateToTheRight(node.right)

return self.rotateToTheleft(node)

elif node.balance() == 2:

if node.left.balance() == -1:

self.rotateToTheleft(node.left)

return self.rotateToTheRight(node)

return node

def search(self, key):

current = self.root

while current is not None:

if current.key.lower() == key.lower():

return current

elif current.key < key:

current = current.right

else:

current = current.left

return None

class RBTNode:

def \_\_init\_\_(self, key, parent, Red=False, left=None, right=None):

self.key = key

self.left = left

self.right = right

self.parent = parent

if Red:

self.color = 'red'

else:

self.color = 'black'

def ColorB(self):

return self.color == 'black'

def ColorR(self):

return self.color == 'red'

def set(self, currentSide, child):

if currentSide != 'left' and currentSide != 'right':

return False

if currentSide == 'left':

self.left = child

else:

self.right = child

def FindGrandparent(self):

if self.parent is None:

return None

return self.parent.parent

def FindSibling(self):

if self.parent is not None:

if self is self.parent.left:

return self.parent.right

return self.parent.left

return None

def FindUncle(self):

grandparent = self.FindGrandparent()

if grandparent is None:

return None

if grandparent.left is self.parent:

return grandparent.right

return grandparent.left

def count(self):

count = 1

if self.left is not None:

count = count + self.left.count()

if self.right is not None:

count = count + self.right.count()

return count

class RBT:

def \_\_init\_\_(self):

self.root = None

def \_\_len\_\_(self):

if self.root is None:

return 0

return self.root.count()

def insert(self, key):

new = RBTNode(key, None, True, None, None)

self.insertHelper(new)

def insertHelper(self, node):

if self.root is None:

self.root = node

else:

current = self.root

while current is not None:

if node.key < current.key:

if current.left is None:

current.set('left', node)

break

else:

current = current.left

else:

if current.right is None:

current.set('right', node)

break

else:

current = current.right

node.color = 'red'

self.balance(node)

def balance(self, node):

if node.parent is None:

node.color = 'black'

return

if node.parent.ColorB():

return

parent = node.parent

grandparent = node.FindGrandparent()

uncle = node.FindUncle()

if uncle is not None and uncle.ColorR():

parent.color = uncle.color = 'black'

grandparent.color = 'red'

self.balance(grandparent)

return

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

self.rotateToTheleft(parent)

node = parent

parent = node.parent

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

self.rotateToTheRight(parent)

node = parent

parent = node.parent

parent.color = 'black'

grandparent.color = 'red'

if node is parent.left:

self.rotateToTheRight(grandparent)

else:

self.rotateToTheleft(grandparent)

def rotateToTheleft(self, node):

rightOfLeft = node.right.left

if node.parent is not None:

node.parent.replace(node, node.right)

else:

self.root = node.right

self.root.parent = None

node.right.set('left', node)

node.set('right', rightOfLeft)

def rotateToTheRight(self, node):

leftOfRight = node.left.right

if node.parent is not None:

node.parent.replace(node, node.left)

else:

self.root = node.left

self.root.parent = None

node.left.set('right', node)

node.set('left', leftOfRight)

def search(self, key):

current = self.root

while current is not None:

if current.key == key:

return current

elif key < current.key:

current = current.left

else:

current = current.right

return None