CS 2302 Lab Report

Lab 3\_Option B

Jose Lujan

Lab 3 Option B

AVL Tree and Red-Black Tree

Objective of the Lab:

The main objective is to create an avl tree and a red-black tree and insert all the words, from the file given by the professor. Once all the methods has been created, when testing the lab the user will be ask to enter which tree do they want to work with. Once the avl tree or the red-black tree has been populated, it will ask the user which operation they want to perform.

Create a function that returns the number of anagrams that a given word has, and the last operation will be to determine which word has the greatest number of anagrams, comparing all the words from the file.

Solution:

# Lab 3 option B : AVL Trees and Red-Black Trees  
# Name: Jose Lujan  
# ID: 80572649  
# class: cs2302  
# class time 10:30-11:50  
  
  
  
# Function creates an AVL tree holding values inside of given file  
def create\_avl(file\_name):  
 english\_words = AVLTree()  
  
 # Open file and read first line  
 file = open(file\_name, "r")  
 line = file.readline()  
  
 # Loop will go trough every line in the file  
 while line:  
 # Add each word, lowercased, to avoid searching errors  
 new\_word = Node(line.rstrip().lower())  
 english\_words.insert(new\_word)  
 line = file.readline()  
  
 # Returns AVL Tree  
 return english\_words  
  
  
# Function creates a Red-Black tree holding values inside of given file  
def create\_red\_black(file\_name):  
 english\_words = RedBlackTree()  
  
 # Open file and read first line  
 file = open(file\_name, "r")  
 line = file.readline()  
  
 # Loop will go trough every line in the file  
 while line:  
 # Add each word, lowercased, to avoid searching errors  
 english\_words.insert(line.rstrip().lower())  
 line = file.readline()  
  
 # Returns Red-Black Tree  
 return english\_words  
  
  
# Function that prints every node in tree, from left to right, recursively  
def print\_tree(tree\_node):  
 if tree\_node is None:  
 print("Your tree is empty.")  
 return  
 # Print left sub-tree  
 if tree\_node.left is not None:  
 print\_tree(tree\_node.left)  
 print(tree\_node.key)  
 # Print right sub-tree  
 if tree\_node.right is not None:  
 print\_tree(tree\_node.right)  
  
  
# Function that generates all possible permutations from a given word  
def get\_perms(word):  
 if len(word) <= 1:  
 return word  
 else:  
 perm\_list = []  
 for perm in get\_perms(word[1:]):  
 for i in range(len(word)):  
 perm\_list.append(perm[:i] + word[0:1] + perm[i:])  
 return perm\_list  
  
  
# Function that prints all valid anagrams from a given word  
def print\_anagrams(word, tree):  
 permutations = get\_perms(word)  
  
 for i in range(len(permutations)):  
 if tree.search(permutations[i]):  
 print(permutations[i])  
 return  
  
  
# Function that returns the number of valid anagrams from a given word  
def count\_anagrams(word, tree):  
 permutations = get\_perms(word)  
 count = 0  
  
 for i in range(len(permutations)):  
 if tree.search(permutations[i]):  
 count += 1  
  
 return count  
  
  
# Function that returns the word with the most possible anagrams from a given file  
def most\_anagrams(file\_name, tree):  
 try:  
 if tree.root is None:  
 print("Your tree is empty.")  
 return None  
 # Open file and read first line  
 file = open(file\_name, "r")  
 line = file.readline()  
  
 highest\_anagrams = line.rstrip().lower() # Variable that holds word with most possible anagrams  
 highest\_count = count\_anagrams(highest\_anagrams, tree)  
  
 # Loop will go trough every line in the file  
 while line:  
 new\_word = line.rstrip().lower()  
 new\_count = count\_anagrams(new\_word, tree)  
 if new\_count > highest\_count:  
 highest\_anagrams = new\_word  
 highest\_count = new\_count  
 line = file.readline()  
  
 # Returns word with the most anagrams  
 return highest\_anagrams  
 except FileNotFoundError:  
 print("File not found. Please try again.")  
  
  
# Main function  
# It is asked to the user what operation he/she wishes to perform.  
def main():  
 print("Please select the type of binary search tree you would like to create: ")  
 print()  
 tree = input("Press 1 for AVL Tree.\nPress 2 for Red-Black Tree.\n")  
  
 if tree == '1':  
 print("Generating your AVL Tree.")  
 avl\_tree = create\_avl("words.txt")  
 keep\_going = True  
 while keep\_going:  
 print("Please type the number of the operation you would like to perform:")  
 print(" 1. Get the number of possible anagrams from a given word.")  
 print(" 2. Compare a list of words from a given file and obtain the word with the most possible anagrams.")  
 print(" 3. Print all possible anagrams from a given word.")  
 print(" 4. ")  
 answer = input()  
  
 if answer == '1':  
 word = input("Please type the word to search for anagrams: ")  
 print(count\_anagrams(word, avl\_tree))  
 elif answer == '2':  
 file = input("Please type the name of the file: ")  
 highest\_anagrams = most\_anagrams(file, avl\_tree)  
 print("The word with the most anagrams is:", highest\_anagrams)  
 else:  
 print("Option you typed, is not listed")  
  
 loop = input("\nNew operation? y/n\n")  
  
 if loop == 'y':  
 keep\_going = True  
 elif loop == 'n':  
 keep\_going = False  
 else:  
 print("please try again.")  
  
  
 elif tree == '2':  
 print("Generating your Red-Black Tree.")  
 red\_black = create\_red\_black("words.txt")  
 keep\_going = True  
 while keep\_going:  
 print("Please type the number of the operation you would like to perform:")  
 print(" 1. Get the number of possible anagrams from a given word.")  
 print(" 2. Compare a list of words from a given file and obtain the word with the most possible anagrams.")  
  
 answer = input()  
  
 if answer == '1':  
 word = input("Please type the word to search for anagrams: ")  
 print(count\_anagrams(word, red\_black))  
 elif answer == '2':  
 file = input("Please type the name of the file: ")  
 highest\_anagrams = most\_anagrams(file, red\_black)  
 print("The word with the most anagrams is:", highest\_anagrams)  
 else:  
 print("Option you typed is not listed")  
  
 loop = input("\nNew operation? y/n\n")  
  
 if loop == 'y':  
 keep\_going = True  
 elif loop == 'n':  
 keep\_going = False  
 else:  
 print("Please try again.")  
 else:  
 print("You type an input which is not available, please run the program again.")  
  
  
main()

Red-Black Tree

from Red\_Black\_Node import RBTNode  
  
  
class RedBlackTree:  
 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\_node = RBTNode(key, None, True, None, None)  
 self.insert\_node(new\_node)  
  
 def insert\_node(self, node):  
 # Begin with normal BST insertion  
 if self.root is None:  
 # Special case for root  
 self.root = node  
 else:  
 current\_node = self.root  
 while current\_node is not None:  
 if node.key < current\_node.key:  
 if current\_node.left is None:  
 current\_node.set\_child("left", node)  
 break  
 else:  
 current\_node = current\_node.left  
 else:  
 if current\_node.right is None:  
 current\_node.set\_child("right", node)  
 break  
 else:  
 current\_node = current\_node.right  
  
 # Color the node red  
 node.color = "red"  
  
 # Balance  
 self.insertion\_balance(node)  
  
 def insertion\_balance(self, node):  
 # If node is the tree's root, then color node black and return  
 if node.parent is None:  
 node.color = "black"  
 return  
  
 # If parent is black, then return without any alterations  
 if node.parent.is\_black():  
 return  
  
 # References to parent, grandparent, and uncle are needed for remaining operations  
 parent = node.parent  
 grandparent = node.get\_grandparent()  
 uncle = node.get\_uncle()  
  
 # If parent and uncle are both red, then color parent and uncle black, color grandparent  
 # red, recursively balance grandparent, then return  
 if uncle is not None and uncle.is\_red():  
 parent.color = uncle.color = "black"  
 grandparent.color = "red"  
 self.insertion\_balance(grandparent)  
 return  
  
 # If node is parent's right child and parent is grandparent's left child, then rotate left  
 # at parent, update node and parent to point to parent and grandparent, respectively  
 if node is parent.right and parent is grandparent.left:  
 self.rotate\_left(parent)  
 node = parent  
 parent = node.parent  
 # Else if node is parent's left child and parent is grandparent's right child, then rotate  
 # right at parent, update node and parent to point to parent and grandparent, respectively  
 elif node is parent.left and parent is grandparent.right:  
 self.rotate\_right(parent)  
 node = parent  
 parent = node.parent  
  
 # Color parent black and grandparent red  
 parent.color = "black"  
 grandparent.color = "red"  
  
 # If node is parent's left child, then rotate right at grandparent, otherwise rotate left  
 # at grandparent  
 if node is parent.left:  
 self.rotate\_right(grandparent)  
 else:  
 self.rotate\_left(grandparent)  
  
 def rotate\_left(self, node):  
 right\_left\_child = node.right.left  
 if node.parent is not None:  
 node.parent.replace\_child(node, node.right)  
 else: # node is root  
 self.root = node.right  
 self.root.parent = None  
 node.right.set\_child("left", node)  
 node.set\_child("right", right\_left\_child)  
  
 def rotate\_right(self, node):  
 left\_right\_child = node.left.right  
 if node.parent is not None:  
 node.parent.replace\_child(node, node.left)  
 else: # node is root  
 self.root = node.left  
 self.root.parent = None  
 node.left.set\_child("right", node)  
 node.set\_child("left", left\_right\_child)  
  
 def \_bst\_remove(self, key):  
 node = self.search(key)  
 self.\_bst\_remove\_node(node)  
  
 def \_bst\_remove\_node(self, node):  
 if node is None:  
 return  
  
 # Case 1: Internal node with 2 children  
 if node.left is not None and node.right is not None:  
 # Find successor  
 successor\_node = node.right  
 while successor\_node.left is not None:  
 successor\_node = successor\_node.left  
  
 # Copy successor's key  
 successor\_key = successor\_node.key  
  
 # Recursively remove successor  
 self.\_bst\_remove\_node(successor\_node)  
  
 # Set node's key to copied successor key  
 node.key = successor\_key  
  
  
 # Case 2: Root node (with 1 or 0 children)  
 elif node is self.root:  
 if node.left is not None:  
 self.root = node.left  
 else:  
 self.root = node.right  
  
 # Make sure the new root, if not None, has parent set to None  
 if self.root is not None:  
 self.root.parent = None  
  
  
 # Case 3: Internal with left child only  
 elif node.left is not None:  
 node.parent.replace\_child(node, node.left)  
  
  
 # Case 4: Internal with right child OR leaf  
 else:  
 node.parent.replace\_child(node, node.right)  
  
 def is\_none\_or\_black(self, node):  
 if node is None:  
 return True  
 return node.is\_black()  
  
 def is\_not\_none\_and\_red(self, node):  
 if node is None:  
 return False  
 return node.is\_red()  
  
 def prepare\_for\_removal(self, node):  
 if self.try\_case1(node):  
 return  
  
 sibling = node.get\_sibling()  
 if self.try\_case2(node, sibling):  
 sibling = node.get\_sibling()  
 if self.try\_case3(node, sibling):  
 return  
 if self.try\_case4(node, sibling):  
 return  
 if self.try\_case5(node, sibling):  
 sibling = node.get\_sibling()  
 if self.try\_case6(node, sibling):  
 sibling = node.get\_sibling()  
  
 sibling.color = node.parent.color  
 node.parent.color = "black"  
 if node is node.parent.left:  
 sibling.right.color = "black"  
 self.rotate\_left(node.parent)  
 else:  
 sibling.left.color = "black"  
 self.rotate\_right(node.parent)  
  
 def remove(self, key):  
 node = self.search(key)  
 if node is not None:  
 self.remove\_node(node)  
 return True  
 return False  
  
 def remove\_node(self, node):  
 if node.left is not None and node.right is not None:  
 predecessor\_node = node.get\_predecessor()  
 predecessor\_key = predecessor\_node.key  
 self.remove\_node(predecessor\_node)  
 node.key = predecessor\_key  
 return  
  
 if node.is\_black():  
 self.prepare\_for\_removal(node)  
 self.\_bst\_remove(node.key)  
  
 # One special case if the root was changed to red  
 if self.root is not None and self.root.is\_red():  
 self.root.color = "black"  
  
 def search(self, key):  
 current\_node = self.root  
 while current\_node is not None:  
 # Return the node if the key matches.  
 if current\_node.key == key:  
 return current\_node  
  
  
 # Navigate to the left if the search key is  
 # less than the node's key.  
 elif key < current\_node.key:  
 current\_node = current\_node.left  
  
  
 # Navigate to the right if the search key is  
 # greater than the node's key.  
 else:  
 current\_node = current\_node.right  
  
 # The key was not found in the tree.  
 return None  
  
 def try\_case1(self, node):  
 if node.is\_red() or node.parent is None:  
 return True  
 return False # node case 1  
  
 def try\_case2(self, node, sibling):  
 if sibling.is\_red():  
 node.parent.color = "red"  
 sibling.color = "black"  
 if node is node.parent.left:  
 self.rotate\_left(node.parent)  
 else:  
 self.rotate\_right(node.parent)  
 return True  
 return False # not case 2  
  
 def try\_case3(self, node, sibling):  
 if node.parent.is\_black() and sibling.are\_both\_children\_black():  
 sibling.color = "red"  
 self.prepare\_for\_removal(node.parent)  
 return True  
 return False # not case 3  
  
 def try\_case4(self, node, sibling):  
 if node.parent.is\_red() and sibling.are\_both\_children\_black():  
 node.parent.color = "black"  
 sibling.color = "red"  
 return True  
 return False # not case 4  
  
 def try\_case5(self, node, sibling):  
 if self.is\_not\_none\_and\_red(sibling.left):  
 if self.is\_none\_or\_black(sibling.right):  
 if node is node.parent.left:  
 sibling.color = "red"  
 sibling.left.color = "black"  
 self.rotate\_right(sibling)  
 return True  
 return False # not case 5  
  
 def try\_case6(self, node, sibling):  
 if self.is\_none\_or\_black(sibling.left):  
 if self.is\_not\_none\_and\_red(sibling.right):  
 if node is node.parent.right:  
 sibling.color = "red"  
 sibling.right.color = "black"  
 self.rotate\_left(sibling)  
 return True  
 return False # not case 6

AVL Tree

class AVLTree:  
 # Constructor to create an empty AVLTree. There is only  
 # one data member, the tree's root Node, and it starts  
 # out as None.  
 def \_\_init\_\_(self):  
 self.root = None  
  
 # Performs a left rotation at the given node. Returns the  
 # new root of the subtree.  
 def rotate\_left(self, node):  
 # Define a convenience pointer to the right child of the  
 # left child.  
 right\_left\_child = node.right.left  
  
 # Step 1 - the right child moves up to the node's position.  
 # This detaches node from the tree, but it will be reattached  
 # later.  
 if node.parent is not None:  
 node.parent.replace\_child(node, node.right)  
 else: # node is root  
 self.root = node.right  
 self.root.parent = None  
  
 # Step 2 - the node becomes the left child of what used  
 # to be its right child, but is now its parent. This will  
 # detach right\_left\_child from the tree.  
 node.right.set\_child('left', node)  
  
 # Step 3 - reattach right\_left\_child as the right child of node.  
 node.set\_child('right', right\_left\_child)  
  
 return node.parent  
  
 # Performs a right rotation at the given node. Returns the  
 # subtree's new root.  
 def rotate\_right(self, node):  
 # Define a convenience pointer to the left child of the  
 # right child.  
 left\_right\_child = node.left.right  
  
 # Step 1 - the left child moves up to the node's position.  
 # This detaches node from the tree, but it will be reattached  
 # later.  
 if node.parent is not None:  
 node.parent.replace\_child(node, node.left)  
 else: # node is root  
 self.root = node.left  
 self.root.parent = None  
  
 # Step 2 - the node becomes the right child of what used  
 # to be its left child, but is now its parent. This will  
 # detach left\_right\_child from the tree.  
 node.left.set\_child('right', node)  
  
 # Step 3 - reattach left\_right\_child as the left child of node.  
 node.set\_child('left', left\_right\_child)  
  
 return node.parent  
  
 # Updates the given node's height and rebalances the subtree if  
 # the balancing factor is now -2 or +2. Rebalancing is done by  
 # performing a rotation. Returns the subtree's new root if  
 # a rotation occurred, or the node if no rebalancing was required.  
 def rebalance(self, node):  
  
 # First update the height of this node.  
 node.update\_height()  
  
 # Check for an imbalance.  
 if node.get\_balance() == -2:  
  
 # The subtree is too big to the right.  
 if node.right.get\_balance() == 1:  
 # Double rotation case. First do a right rotation  
 # on the right child.  
 self.rotate\_right(node.right)  
  
 # A left rotation will now make the subtree balanced.  
 return self.rotate\_left(node)  
  
  
 elif node.get\_balance() == 2:  
  
 # The subtree is too big to the left  
 if node.left.get\_balance() == -1:  
 # Double rotation case. First do a left rotation  
 # on the left child.  
 self.rotate\_left(node.left)  
  
 # A right rotation will now make the subtree balanced.  
 return self.rotate\_right(node)  
  
 # No imbalance, so just return the original node.  
 return node  
  
 def insert(self, node):  
  
 # Special case: if the tree is empty, just set the root to  
 # the new node.  
 if self.root is None:  
 self.root = node  
 node.parent = None  
  
  
 else:  
 # Step 1 - do a regular binary search tree insert.  
 current\_node = self.root  
 while current\_node is not None:  
 # Choose to go left or right  
 if node.key < current\_node.key:  
 # Go left. If left child is None, insert the new  
 # node here.  
 if current\_node.left is None:  
 current\_node.left = node  
 node.parent = current\_node  
 current\_node = None  
 else:  
 # Go left and do the loop again.  
 current\_node = current\_node.left  
 else:  
 # Go right. If the right child is None, insert the  
 # new node here.  
 if current\_node.right is None:  
 current\_node.right = node  
 node.parent = current\_node  
 current\_node = None  
 else:  
 # Go right and do the loop again.  
 current\_node = current\_node.right  
  
 # Step 2 - Rebalance along a path from the new node's parent up  
 # to the root.  
 node = node.parent  
 while node is not None:  
 self.rebalance(node)  
 node = node.parent  
  
 def remove\_node(self, node):  
 # Base case:  
 if node is None:  
 return False  
  
 # Parent needed for rebalancing.  
 parent = node.parent  
  
 # Case 1: Internal node with 2 children  
 if node.left is not None and node.right is not None:  
 # Find successor  
 successor\_node = node.right  
 while successor\_node.left is not None:  
 successor\_node = successor\_node.left  
  
 # Copy the value from the node  
 node.key = successor\_node.key  
  
 # Recursively remove successor  
 self.remove\_node(successor\_node)  
  
 # Nothing left to do since the recursive call will have rebalanced  
 return True  
  
  
 # Case 2: Root node (with 1 or 0 children)  
 elif node is self.root:  
 if node.left is not None:  
 self.root = node.left  
 else:  
 self.root = node.right  
  
 if self.root is not None:  
 self.root.parent = None  
  
 return True  
  
  
 # Case 3: Internal with left child only  
 elif node.left is not None:  
 parent.replace\_child(node, node.left)  
  
  
 # Case 4: Internal with right child only OR leaf  
 else:  
 parent.replace\_child(node, node.right)  
  
 # node is gone. Anything that was below node that has persisted is already correctly  
 # balanced, but ancestors of node may need rebalancing.  
 node = parent  
 while node is not None:  
 self.rebalance(node)  
 node = node.parent  
  
 return True  
  
 # Searches for a node with a matching key. Does a regular  
 # binary search tree search operation. Returns the node with the  
 # matching key if it exists in the tree, or None if there is no  
 # matching key in the tree.  
 def search(self, key):  
 current\_node = self.root  
 while current\_node is not None:  
 # Compare the current node's key with the target key.  
 # If it is a match, return the current key; otherwise go  
 # either to the left or right, depending on whether the  
 # current node's key is smaller or larger than the target key.  
 if current\_node.key == key:  
 # print(current\_node.key)  
 return True  
 elif current\_node.key < key:  
 current\_node = current\_node.right  
 else:  
 current\_node = current\_node.left  
 return False  
  
 # Attempts to remove a node with a matching key. If no node has a matching key  
 # then nothing is done and False is returned; otherwise the node is removed and  
 # True is returned.  
 def remove\_key(self, key):  
 node = self.search(key)  
 if node is None:  
 return False  
 else:  
 return self.remove\_node(node)

Red-Black Tree Node file

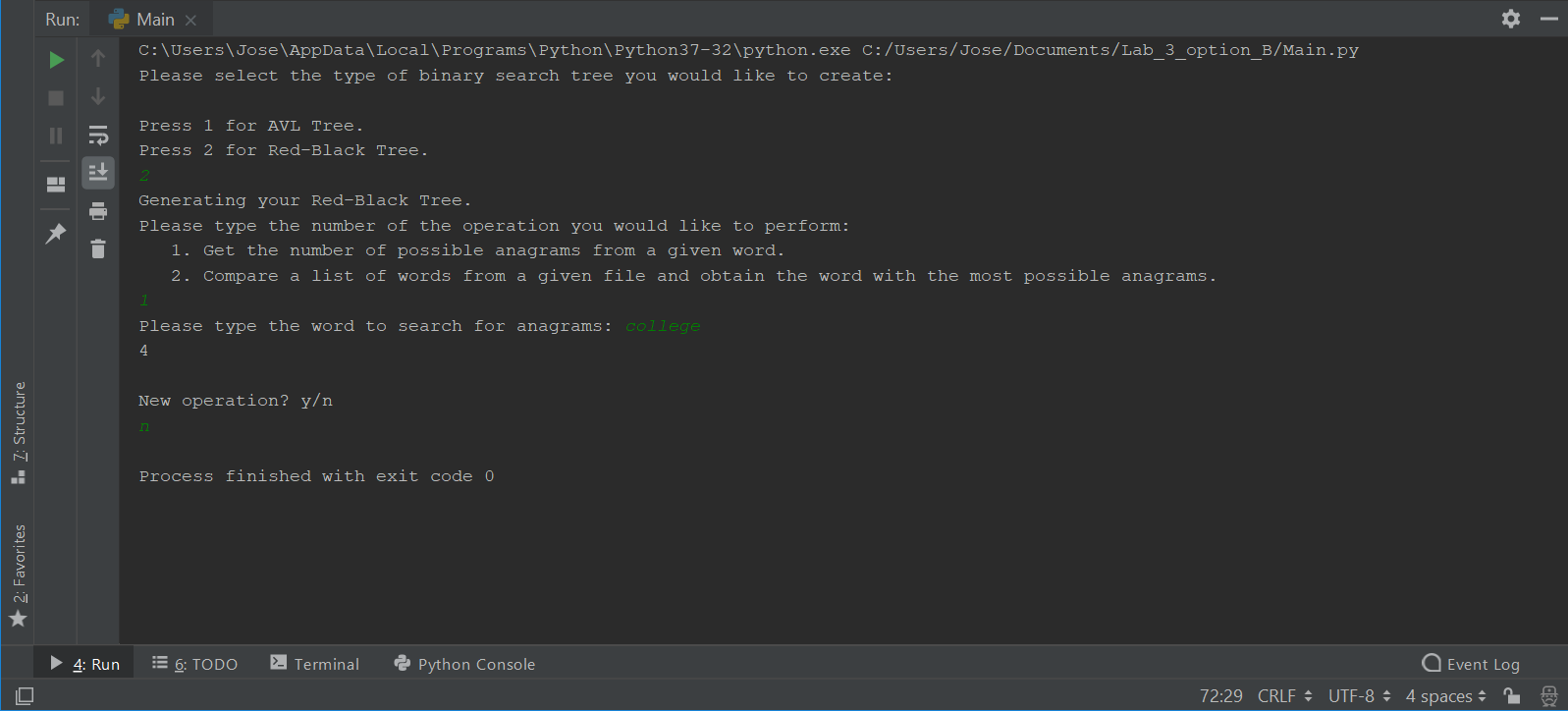
# RBTNode class - represents a node in a red-black tree  
class RBTNode:  
 def \_\_init\_\_(self, key, parent, is\_red=False, left=None, right=None):  
 self.key = key  
 self.left = left  
 self.right = right  
 self.parent = parent  
  
 if is\_red:  
 self.color = "red"  
 else:  
 self.color = "black"  
  
 # Returns true if both child nodes are black. A child set to None is considered  
 # to be black.  
 def are\_both\_children\_black(self):  
 if self.left is not None and self.left.is\_red():  
 return False  
 if self.right is not None and self.right.is\_red():  
 return False  
 return True  
  
 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  
  
 # Returns the grandparent of this node  
 def get\_grandparent(self):  
 if self.parent is None:  
 return None  
 return self.parent.parent  
  
 # Gets this node's predecessor from the left child subtree  
 # Precondition: This node's left child is not None  
 def get\_predecessor(self):  
 node = self.left  
 while node.right is not None:  
 node = node.right  
 return node  
  
 # Returns this node's sibling, or None if this node does not have a sibling  
 def get\_sibling(self):  
 if self.parent is not None:  
 if self is self.parent.left:  
 return self.parent.right  
 return self.parent.left  
 return None  
  
 # Returns the uncle of this node  
 def get\_uncle(self):  
 grandparent = self.get\_grandparent()  
 if grandparent is None:  
 return None  
 if grandparent.left is self.parent:  
 return grandparent.right  
 return grandparent.left  
  
 # Returns True if this node is black, False otherwise  
 def is\_black(self):  
 return self.color == "black"  
  
 # Returns True if this node is red, False otherwise  
 def is\_red(self):  
 return self.color == "red"  
  
 # Replaces one of this node's children with a new child  
 def replace\_child(self, current\_child, new\_child):  
 if self.left is current\_child:  
 return self.set\_child("left", new\_child)  
 elif self.right is current\_child:  
 return self.set\_child("right", new\_child)  
 return False  
  
 # Sets either the left or right child of this node  
 def set\_child(self, which\_child, child):  
 if which\_child != "left" and which\_child != "right":  
 return False  
  
 if which\_child == "left":  
 self.left = child  
 else:  
 self.right = child  
  
 if child is not None:  
 child.parent = self  
  
 return True

AVL-Tree Node file

# AVLNode class - represents a node in an AVL tree  
class Node:  
 # Constructor with a key parameter creates the Node object.  
 def \_\_init\_\_(self, key):  
 self.key = key  
 self.parent = None  
 self.left = None  
 self.right = None  
 self.height = 0  
  
 # Calculate the current nodes' balance factor,  
 # defined as height(left subtree) - height(right subtree)  
 def get\_balance(self):  
 # Get current height of left subtree, or -1 if None  
 left\_height = -1  
 if self.left is not None:  
 left\_height = self.left.height  
  
 # Get current height of right subtree, or -1 if None  
 right\_height = -1  
 if self.right is not None:  
 right\_height = self.right.height  
  
 # Calculate the balance factor.  
 return left\_height - right\_height  
  
 # Recalculate the current height of the subtree rooted at  
 # the node, usually called after a subtree has been  
 # modified.  
 def update\_height(self):  
 # Get current height of left subtree, or -1 if None  
 left\_height = -1  
 if self.left is not None:  
 left\_height = self.left.height  
  
 # Get current height of right subtree, or -1 if None  
 right\_height = -1  
 if self.right is not None:  
 right\_height = self.right.height  
  
 # Assign self.height with calculated node height.  
 self.height = max(left\_height, right\_height) + 1  
  
 # Assign either the left or right data member with a new  
 # child. The parameter which\_child is expected to be the  
 # string "left" or the string "right". Returns True if  
 # the new child is successfully assigned to this node, False  
 # otherwise.  
 def set\_child(self, which\_child, child):  
 # Ensure which\_child is properly assigned.  
 if which\_child != "left" and which\_child != "right":  
 return False  
  
 # Assign the left or right data member.  
 if which\_child == "left":  
 self.left = child  
 else:  
 self.right = child  
  
 # Assign the parent data member of the new child,  
 # if the child is not None.  
 if child is not None:  
 child.parent = self  
  
 # Update the node's height, since the subtree's structure  
 # may have changed.  
 self.update\_height()  
 return True  
  
 # Replace a current child with a new child. Determines if  
 # the current child is on the left or right, and calls  
 # set\_child() with the new node appropriately.  
 # Returns True if the new child is assigned, False otherwise.  
 def replace\_child(self, current\_child, new\_child):  
 if self.left is current\_child:  
 return self.set\_child("left", new\_child)  
 elif self.right is current\_child:  
 return self.set\_child("right", new\_child)  
  
 # If neither of the above cases applied, then the new child  
 # could not be attached to this node.  
 return False

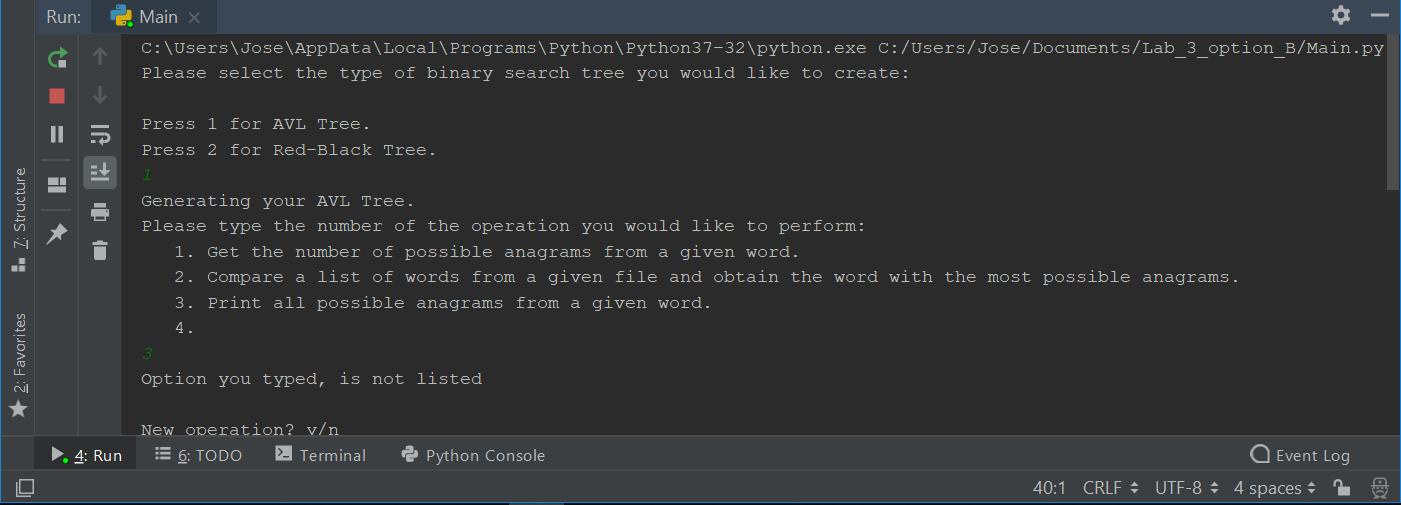
Test #1

In test #1 I created a red-black Tree, once the tree got generated I decided to ask the program which was the number of anagrams that the word “college” has, which is 4.



Test #2

In this test, I decided to create an AVL -tree and once the tree got generated I entered, something that wasn’t in the options given by the program. So I go a test , saying : “try again, option you typed is not listed.



Once we start testing the lab, the first it will be to ask the user which tree the wish to work with. In this case the options will be: AVL tree and the Red-Black tree. After the program gets an answer it will take some seconds to generate the tree, it all depends on the option the user chose. One thing we should take in consideration is that the Red-Black takes less time to generate.