Lab 3 - Data Structures Report Brian Ricardo Perez

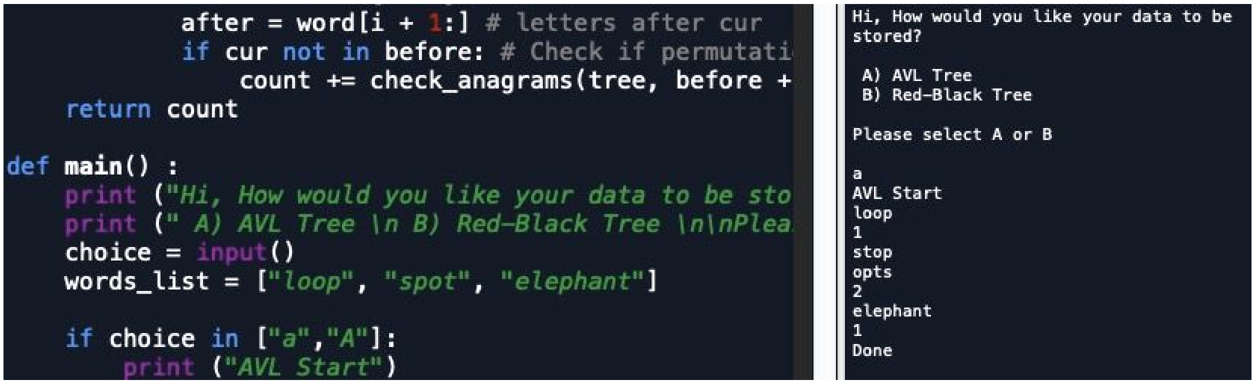
INTRODUCTION :

Balance Search Trees, more specifically AVL and Red Black Trees, the two are different in their own way, AVL using rotations to balance the binary search tree. And Red-Black Trees using kind of the same thing but using colors as labels to make different “balancing” acts. In this project, we are required to gather all the words in the english language and put them into two different types of trees to the discretion of the user. Then use that data structure to search through and discover real anagrams of a list of words.

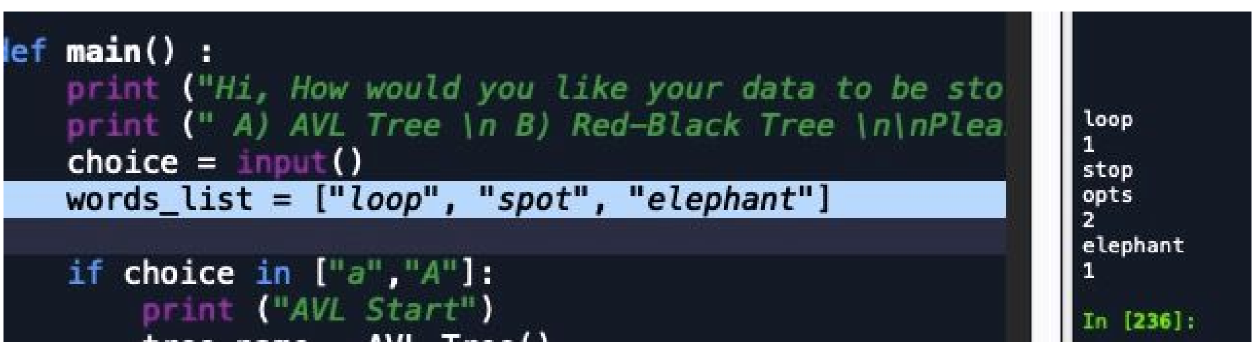
PROPOSED SOLUTION DESIGN AND IMPLEMENTATION:

My solution was very coordinated by the project constraints themselves, in addition to the fact that I also used code provided by zybooks and adapting it into the Python language for most of the construction of both AVL and RBTrees. When it came to implementing my own solutions it came in the form of creating a universal search method for both AVL and RB, (universal_Search(tree, node)) I only needed, value, left child, and right child traversal which of course both my RB and AVL tree nodes had. There was a lot of times I was breaking the code, there was this one instance in my insert for my RB tree where in line 60 and others like it I tried to insert a newly created node (= RBNode (cur.value)) with the current nodes value and see if it might work still. But as it should it created new node after new node, and did not “link” the nodes together in a tree.

EXPERIMENTAL RESULTS:



In the picture above you can see I created a list of words to test anagrams in the entire data structure of words in the english language. In this instance the output came out as it should counting all the anagrams and returning the amount found in the Tree. The running time of the AVL tree was at least nlogn, being that it has to go through the entire tree, but also traverse the list size n, so n * log n. The AVL tree is very fast, the english language is obviously very large and it only took less than 10 seconds to fully finish the code doing multiple traversals through the entire tree. Because of the AVL tree's strong constraints the balance in the end is very well done.



This is the Red-Black Tree, it was much slower to my actual surprise, until I realized that even with all the labeling with red and black, its main use is speed in insertion, and we really only have to insert all the words once, and that's where this tree lacks comparatively to the AVL. It has strict balancing rules, but nowhere near to the AVL's. The running time of this version of the code is n^2 log n. Also Side note, tiny glitch in my code, when it finishes it has this massive space that is created but I'm not sure where it comes from. (as shown on the right)

CONCLUSIONS:

All in all, this was a very challenging lab but a kind of fun one towards the end. What I learned [illegible] the case that matters, the AVL tree is very much quicker due to this lab lacking its strong suit.

```
"""
Created on Wednesday October  9 12:40:04 2019
Course: CS 2302 - Data Structures
Author: Brian Perez
Assignment: Lab 3
Instructor: Diego Aguirre
D.O.L.M.: 11/1/19
"""
class node:
    def __init__(self,value=None):
        self.value=value
        self.left_child=None
        self.right_child=None
        self.parent=None # pointer to parent node in tree
        self.height=1 # height of node in tree (max dist. to leaf) NEW FOR AVL

class RBNode:
    def __init__(self, value = None):
        self.value = value
        self.color = None
```

```
        self.left_child = None
        self.right_child = None
        self.parent = None

class RB_Tree:
    def __init__(self):
        self.root=None

    def RBprint_tree(self):
        if self.root!=None:
            self._RBprint_tree(self.root)

    def _RBprint_tree(self, RBNode):
        if RBNode != None:
            self._RBprint_tree(RBNode.left_child)
            print(str(RBNode.value), str(RBNode.color))
            #if RBNode.parent:
            #   print("Parent node of ","'", (str(RBNode.value)),"':", (str(RBNode.parent.value)))
            self._RBprint_tree(RBNode.right_child)

    def RBTreeInsert(self, value):
        RBNodes = RBNode(value)
        self.RBinsert(RBNodes)
        RBNodes.color = "red"
        #print("Vanilla values:", RBNodes.value, RBNodes.color)
        #print("hi yall wyd lol")
        self.RBTreeBalance(RBNodes)


    def RBinsert(self, new_node):
        if (self.root is None):
            self.root = new_node
            new_node.left_child = None
            new_node.right_child = None
        else:
            cur = self.root
            while (cur is not None):
                if (new_node.value < cur.value):
                    if (cur.left_child is None):
                        cur.left_child = new_node
                        cur.left_child.parent = cur
                        cur = None
                    else:
                        cur = cur.left_child
                else:
                    if (cur.right_child is None):
                        cur.right_child = new_node
                        cur.right_child.parent = cur
                        cur = None
                    else:
```

```
                cur = cur.right_child
        new_node.left_child = None
        new_node.right_child = None

    def RBTreeGetGrandparent(self, RBNode):
        if (RBNode.parent == None):
            return None
        return RBNode.parent.parent

    def RBTreeGetUncle(self, RBNode):
        grandparent = None
        if (RBNode.parent != None):
            grandparent = RBNode.parent.parent
        if (grandparent == None):
            return None
        if (grandparent.left_child == RBNode.parent):
            return grandparent.right_child
        else:
            return grandparent.left_child


    def RBTreeBalance(self, RBNode):
        if (RBNode.parent == None):
            RBNode.color = "black"
            return
        if (RBNode.parent.color == "black"):
            return
        parent = RBNode.parent
        grandparent = self.RBTreeGetGrandparent(RBNode)
        uncle = self.RBTreeGetUncle(RBNode)
        if (uncle != None and uncle.color == "red"):
            parent.color = uncle.color = "black"
            grandparent.color = "red"
            self.RBTreeBalance(grandparent)
            return
        if (RBNode == parent.right_child and parent == grandparent.left_child):
            self.RBTreeRotateLeft(parent)
            RBNode = parent
            parent = RBNode.parent
        elif (RBNode == parent.left_child and parent == grandparent.right_child):
            self.RBTreeRotateRight(parent)
            RBNode = parent
            parent = RBNode.parent
        parent.color = "black"
        grandparent.color = "red"

        if (RBNode == parent.left_child):
            self.RBTreeRotateRight(grandparent)
        else:
            self.RBTreeRotateLeft(grandparent)
```

```
    def RBTreeSetChild(self, parent, whichChild, child):
        if ((whichChild != "left") and (whichChild != "right")):
            return False
        if (whichChild == "left"):
            parent.left_child = child
        else:
            parent.right_child = child
        if (child != None):
            child.parent = parent
        return True

    def RBTreeReplaceChild(self, parent, currentChild, newChild):
        if (parent.left_child == currentChild):
            return self.RBTreeSetChild(parent, "left", newChild)
        elif (parent.right_child == currentChild):
            return self.RBTreeSetChild(parent, "right", newChild)
        return False

    def RBTreeRotateLeft(self, RBNode):
        rightLeftChild = RBNode.right_child.left_child
        if (RBNode.parent != None):
            self.RBTreeReplaceChild(RBNode.parent, RBNode, RBNode.right_child)
        else:
            self.root = RBNode.right_child
            self.root.parent = None
        self.RBTreeSetChild(RBNode.right_child, "left", RBNode)
        self.RBTreeSetChild(RBNode, "right", rightLeftChild)

    def RBTreeRotateRight(self, RBNode):
        leftRightChild = RBNode.left_child.right_child
        if (RBNode.parent != None):
            self.RBTreeReplaceChild(RBNode.parent, RBNode, RBNode.left_child)
        else:
            self.root = RBNode.left_child
            self.root.parent = None
        self.RBTreeSetChild(RBNode.left_child, "right", RBNode)
        self.RBTreeSetChild(RBNode, "left", leftRightChild)

class AVL_Tree:
    def __init__(self):
        self.root=None

    def __repr__(self):
        if self.root==None: return ''
        content='\n' # to hold final string
        cur_nodes=[self.root] # all nodes at current level
        cur_height=self.root.height # height of nodes at current level
        sep=' '*(2**(cur_height-1)) # variable sized separator between elements
        while True:
```

```
            cur_height+=-1 # decrement current height
            if len(cur_nodes)==0: break
            cur_row=' '
            next_row=''
            next_nodes=[]

            if all(n is None for n in cur_nodes):
                break

            for n in cur_nodes:

                if n==None:
                    cur_row+='   '+sep
                    next_row+='   '+sep
                    next_nodes.extend([None,None])
                    continue
                if n.value!=None:
                    buf=' '*int((5-len(str(n.value)))/2)
                    cur_row+='%s%s%s'%(buf,str(n.value),buf)+sep
                else:
                    cur_row+=' '*5+sep

                if n.left_child!=None:
                    next_nodes.append(n.left_child)
                    next_row+=' /'+sep
                else:
                    next_row+='  '+sep
                    next_nodes.append(None)

                if n.right_child!=None:
                    next_nodes.append(n.right_child)
                    next_row+='\ '+sep
                else:
                    next_row+='  '+sep
                    next_nodes.append(None)

            content+=(cur_height*'   '+cur_row+'\n'+cur_height*'   '+next_row+'\n')
            cur_nodes=next_nodes
            sep=' '*int(len(sep)/2) # cut separator size in half

        return content

    def insert(self,value):
        if self.root==None:
            self.root=node(value)
        else:
            self._insert(value,self.root)

    def _insert(self,value,cur_node):
        if value<cur_node.value:
```

```
            if cur_node.left_child==None:
                cur_node.left_child=node(value)
                cur_node.left_child.parent=cur_node # set parent
                self._inspect_insertion(cur_node.left_child)
            else:
                self._insert(value,cur_node.left_child)
        elif value>cur_node.value:
            if cur_node.right_child==None:
                cur_node.right_child=node(value)
                cur_node.right_child.parent=cur_node # set parent
                self._inspect_insertion(cur_node.right_child)
            else:
                self._insert(value,cur_node.right_child)
        else:
            print("Value already in tree!")

    def print_tree(self):
        if self.root!=None:
            self._print_tree(self.root)

    def _print_tree(self,cur_node):
        if cur_node!=None:
            self._print_tree(cur_node.left_child)
            print ((str(cur_node.value)))
            self._print_tree(cur_node.right_child)

    def height(self):
        if self.root!=None:
            return self._height(self.root,0)
        else:
            return 0

    def _height(self,cur_node,cur_height):
        if cur_node==None: return cur_height
        left_height=self._height(cur_node.left_child,cur_height+1)
        right_height=self._height(cur_node.right_child,cur_height+1)
        return max(left_height,right_height)

    def find(self,value):
        if self.root!=None:
            return self._find(value,self.root)
        else:
            return None

    def _find(self,value,cur_node):
        if value==cur_node.value:
            return cur_node
        elif value<cur_node.value and cur_node.left_child!=None:
            return self._find(value,cur_node.left_child)
        elif value>cur_node.value and cur_node.right_child!=None:
```

```
            return self._find(value,cur_node.right_child)

    def delete_value(self,value):
        return self.delete_node(self.find(value))

    def delete_node(self,node):

        ## -----
        # Improvements since prior lesson

        # Protect against deleting a node not found in the tree
        if node==None or self.find(node.value)==None:
            print("Node to be deleted not found in the tree!")
            return None
        ## -----

        # returns the node with min value in tree rooted at input node
        def min_value_node(n):
            current=n
            while current.left_child!=None:
                current=current.left_child
            return current

        # returns the number of children for the specified node
        def num_children(n):
            num_children=0
            if n.left_child!=None: num_children+=1
            if n.right_child!=None: num_children+=1
            return num_children

        # get the parent of the node to be deleted
        node_parent=node.parent

        # get the number of children of the node to be deleted
        node_children=num_children(node)

        # break operation into different cases based on the
        # structure of the tree & node to be deleted

        # CASE 1 (node has no children)
        if node_children==0:

            if node_parent!=None:
                # remove reference to the node from the parent
                if node_parent.left_child==node:
                    node_parent.left_child=None
                else:
                    node_parent.right_child=None
            else:
                self.root=None
```

```
        # CASE 2 (node has a single child)
        if node_children==1:

            # get the single child node
            if node.left_child!=None:
                child=node.left_child
            else:
                child=node.right_child

            if node_parent!=None:
                # replace the node to be deleted with its child
                if node_parent.left_child==node:
                    node_parent.left_child=child
                else:
                    node_parent.right_child=child
            else:
                self.root=child

            # correct the parent pointer in node
            child.parent=node_parent

        # CASE 3 (node has two children)
        if node_children==2:

            # get the inorder successor of the deleted node
            successor=min_value_node(node.right_child)

            # copy the inorder successor's value to the node formerly
            # holding the value we wished to delete
            node.value=successor.value

            # delete the inorder successor now that it's value was
            # copied into the other node
            self.delete_node(successor)

            # exit function so we don't call the _inspect_deletion twice
            return

        if node_parent!=None:
            # fix the height of the parent of current node
            node_parent.height=1+max(self.get_height(node_parent.left_child),self.get_height(node_parent.right_child))

            # begin to traverse back up the tree checking if there are
            # any sections which now invalidate the AVL balance rules
            self._inspect_deletion(node_parent)

    def search(self,value):
        if self.root!=None:
            return self._search(value,self.root)
```

```
    else:
        return False

def _search(self,value,cur_node):
    if value==cur_node.value:
        return True
    elif value<cur_node.value and cur_node.left_child!=None:
        return self._search(value,cur_node.left_child)
    elif value>cur_node.value and cur_node.right_child!=None:
        return self._search(value,cur_node.right_child)
    return False


# Functions added for AVL...

def _inspect_insertion(self,cur_node,path=[]):
    if cur_node.parent==None: return
    path=[cur_node]+path

    left_height =self.get_height(cur_node.parent.left_child)
    right_height=self.get_height(cur_node.parent.right_child)

    if abs(left_height-right_height)>1:
        path=[cur_node.parent]+path
        self._rebalance_node(path[0],path[1],path[2])
        return

    new_height=1+cur_node.height
    if new_height>cur_node.parent.height:
        cur_node.parent.height=new_height

    self._inspect_insertion(cur_node.parent,path)

def _inspect_deletion(self,cur_node):
    if cur_node==None: return

    left_height =self.get_height(cur_node.left_child)
    right_height=self.get_height(cur_node.right_child)

    if abs(left_height-right_height)>1:
        y=self.taller_child(cur_node)
        x=self.taller_child(y)
        self._rebalance_node(cur_node,y,x)

    self._inspect_deletion(cur_node.parent)

def _rebalance_node(self,z,y,x):
    if y==z.left_child and x==y.left_child:
        self._right_rotate(z)
    elif y==z.left_child and x==y.right_child:
```

```
            self._left_rotate(y)
            self._right_rotate(z)
        elif y==z.right_child and x==y.right_child:
            self._left_rotate(z)
        elif y==z.right_child and x==y.left_child:
            self._right_rotate(y)
            self._left_rotate(z)
        else:
            raise Exception('_rebalance_node: z,y,x node configuration not recognized!')

    def _right_rotate(self,z):
        sub_root=z.parent
        y=z.left_child
        t3=y.right_child
        y.right_child=z
        z.parent=y
        z.left_child=t3
        if t3!=None: t3.parent=z
        y.parent=sub_root
        if y.parent==None:
                self.root=y
        else:
            if y.parent.left_child==z:
                y.parent.left_child=y
            else:
                y.parent.right_child=y
        z.height=1+max(self.get_height(z.left_child),
            self.get_height(z.right_child))
        y.height=1+max(self.get_height(y.left_child),
            self.get_height(y.right_child))

    def _left_rotate(self,z):
        sub_root=z.parent
        y=z.right_child
        t2=y.left_child
        y.left_child=z
        z.parent=y
        z.right_child=t2
        if t2!=None: t2.parent=z
        y.parent=sub_root
        if y.parent==None:
            self.root=y
        else:
            if y.parent.left_child==z:
                y.parent.left_child=y
            else:
                y.parent.right_child=y
        z.height=1+max(self.get_height(z.left_child),
            self.get_height(z.right_child))
        y.height=1+max(self.get_height(y.left_child),
```

```
            self.get_height(y.right_child))

    def get_height(self,cur_node):
        if cur_node==None: return 0
        return cur_node.height

    def taller_child(self,cur_node):
        left=self.get_height(cur_node.left_child)
        right=self.get_height(cur_node.right_child)
        return cur_node.left_child if left>=right else cur_node.right_child

def create_list(LL):
    file = open("vivendi.txt", 'r')
    file2 = open("activision.txt", 'r')

    for x in file.read().split('\n'):
        LL.add_last(x)
    for x in file2.read().split ('\n'):
        LL.add_last(x)

def AVL_tree_fill(tree_name):
    file = open ("words.txt", 'r')

    for x in file.read().split('\n'):
        tree_name.insert(x)
    #print(tree_name)
    #tree_name.print_tree()

def RB_tree_fill(tree_name):
    file = open ("words.txt", 'r')

    for x in file.read().split('\n'):
        tree_name.RBTreeInsert(x)
        print("\n")
        #tree_name.RBprint_tree()

def universal_Search(tree, key):
    cur = tree.root
    while (cur is not None):
        if (key == cur.value):
            return True
        elif (key < cur.value):
            cur = cur.left_child
        else:
            cur = cur.right_child
    return False

def check_anagrams (tree, word, prefix=""):
    count = 0
    if len(word) <= 1:
```

```
        string = prefix + word
        if universal_Search(tree, string):
            print(prefix + word)
            return 1
    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.
                count += check_anagrams(tree, before + after, prefix + cur)
    return count

def main() :
    print ("Hi, How would you like your data to be stored?\n")
    print (" A) AVL Tree \n B) Red-Black Tree \n\nPlease select A or B")
    choice = input()
    words_list = ["loop", "spot", "elephant"]

    if choice in ["a","A"]:
        print ("AVL Start")
        tree_name = AVL_Tree()
        AVL_tree_fill(tree_name)
        for x in words_list:
            print (check_anagrams(tree_name, x))

        print("Done")

    if choice in ["b","B"]:
        print ("RBT Start")
        tree_name = RB_Tree()
        RB_tree_fill(tree_name)
        for x in words_list:
            print (check_anagrams(tree_name, x))

   #a print_anagrams("word")


main ()
```