Ricardo Godoy

Lab Report

The objective of this lab was to utilize a binary search tree, either AVL or Red Black Tree, to store the find the anagrams of a given English word. To populate the tree, I utilized a text file with more than 300,00 English words. In order to solve the problem, I used the Zybooks' implementation of AVL and Red Black Trees. This was used to create the tree, insert the nodes, and search for the anagrams of a given English word. The program lets the user decide what type of tree they want to use, and the program will make the required computation depending on the user input. After the user selects the type of binary search tree they want to use, the program will perform the creation of the desired tree and populate it with the text file provided.

The program starts by prompting the user to select the type of tree they want to use. After that, the program will create the selected tree and populate it using the provided functions. The AVL function reads the file line by line and inserts the words into the tree, with the corresponding insert algorithm of the AVL tree. This function runs if the user inputs 1, which is the AVL tree selection. The function RBT is in charge of populating the Red Black Tree with the words, reading them line by line, and inserting them into the tree using the insert algorithm in the RedBlackTree class. After the tree desired by the user is created and assigned the name of english_words, the program will prompt the user what word they want to use to find the anagrams of. The function in charge of this task is print_anagrams, which takes care of finding the permutations of the words in the tree, and if it does, it will print them. After this task, I needed to create an object Counter to keep track of the number of anagrams of each word, along with the maximum number of anagrams and the corresponding word. For computing the number of anagrams of a given word, the function count_anagrams is a variation of the function print_anagrams, which instead of printing a valid permutation of the word, it increases the count of the Counter object by one each time the word is found in the tree. To compute the maximum number of permutations of a word, there is the function max_anagrams, which creates an object of the Counter class for each line in the file, and counts the number of valid permutations of each word, and computes the maximum number of anagrams, and the word with more valid permutations. The tasks to populate the tree had a time complexity of O(n), due to the fact that the function had to go through the entire word file to populate the tree and insert the nodes. The print_anagrams and count_anagrams had a time complexity of O(log n), in respect to the size of the tree. Finally, the max_anagrams function had a time complexity of O(n log n), because of the function having to go through the entire file, and traverse the tree.

My test cases consisted in reducing significantly the number of words in the text file, due to the large amount of running time. The words included in the text file were all anagrams of the words "spot" from the original words text file. Here is the output:

```
What type of Binary Search Tree do you want? Type 1 for AVL
Tree or 2 for Red-Black Tree: 1

Which word do you want to permutate? spot
spot
stop
post
pots
opts
tops
Number of permutations of the word:  6
spot 6
6
stop 6
6
post 6
6
pots 6
6
opts 6
6
tops 6
6
Aar 0
0
Word with more anagrams: spot
Max number of anagrams: 6

In [2]:
```

```
What type of Binary Search Tree do you want? Type 1 for AVL
Tree or 2 for Red-Black Tree: 2

Which word do you want to permutate? spot
spot
stop
post
pots
opts
tops
Number of permutations of the word:  6
spot 6
6
stop 6
6
post 6
6
pots 6
6
opts 6
6
tops 6
6
Aar 0
0
Word with more anagrams: spot
Max number of anagrams: 6

In [3]:
```

For the second text, I increased the size of the list. This time, I entered the word “aerst” to test the program’s efficiency in founding the different valid permutations of different words. This was the result:

```
What type of Binary Search Tree do you want? Type 1 for AVL
Tree or 2 for Red-Black Tree: 1

Which word do you want to permutate? aerst
arest
aster
astre
Number of permutations of the word:  3
```

Both trees were used in the test cases to demonstrate the functionality of the program.

For this lab, I learned how to implement AVL and Red Black Trees. Even though in this case we used words to populate the trees, I understand that trees can also contain objects. I encountered myself having very long runtimes and difficulties reading the file and populating the tree correctly, mostly due to the new line symbol on the file.

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 the class.

Appendix

```
# -*- coding: utf-8 -*-
"""

Created on Mon Oct 29 08:19:37 2018

@author: rgodoy
"""

# Course: CS2302
# Author: Ricardo Godoy
# Assignment: Lab 3 B
# T.A: Saha, Manoj
# Instructor: Diego Aguirre
# Date of last modification: 11/08/18

# This program reads a file that contains all english words and inserts them into a tree, selected
# by the user. After the tree selected is populated with the words of the wile, there is a method
# that prints the anagrams of a specific word, only if it is in the tree. Along with that,
# the program also computes the word with the most number of anagrams and the number of anagrams
# of each word

class Node:
    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

class BinarySearchTree:
    def __init__(self):

        self.root = None

    def search(self, desired_key):
```

```
    current_node = self.root
    while current_node is not None:
        # Return the node if the key matches.
        if current_node.key == desired_key:
            return current_node

        # Navigate to the left if the search key is
        # less than the node's key.
        elif desired_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 insert(self, node):

    # Check if the tree is empty
    if self.root is None:
        self.root = node
    else:
        current_node = self.root
        while current_node is not None:
            if node.key < current_node.key:
```

```
                # If there is no left child, add the new
                # node here; otherwise repeat from the
                # left child.
                if current_node.left is None:
                    current_node.left = node
                    current_node = None
                else:
                    current_node = current_node.left
            else:
                # If there is no right child, add the new
                # node here; otherwise repeat from the
                # right child.
                if current_node.right is None:
                    current_node.right = node
                    current_node = None
                else:
                    current_node = current_node.right



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

# 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: return True
        elif current_node.key < key: current_node = current_node.right
```

```
            else: current_node = current_node.left
        return False

    def preOrder(self, root):
        if not root:
            return
        print (root.key)
        self.preOrder(root.left)
        self.preOrder(root.right)

# 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 != None and self.left.is_red():
            return False
```

```
    if self.right != None and self.right.is_red():
        return False
    return True

def count(self):
    count = 1
    if self.left != None:
        count = count + self.left.count()
    if self.right != 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 != None:
            child.parent = self

        return True

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 != 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 != 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 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 AVL(filename):
    #Constructs the tree and inserts all words from the file into the tree


    english_words= AVLTree()

    with open(filename) as file:
        for line in file:

            node = Node(line.lower().replace("\n", ""))
            english_words.insert(node)

    return english_words

def RBT(filename):
    #Constructs the tree and inserts all words from the file into the tree

    english_words = RedBlackTree()

    with open(filename) as file:
```

```
        for line in file:
#           node = RBTNode(line.lower().replace("\n", ""), )

            english_words.insert(line.lower().replace("\n", ""))
    return english_words

def print_anagrams(word, prefix=""):
    #Prints the anagrams of a specific word
    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)



def count_anagrams(obj, word, prefix=""):
    #Counts the anagrams found in the tree
```

```
    if len(word) <= 1:
        str = prefix + word
        if english_words.search(str):
            obj.count = obj.count + 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_anagrams(obj, before + after, prefix + cur)
    return obj.count



def max_anagrams(obj, filename):
    #Computes the word with the maximum number of anagrams
    file = open(filename)
    max_count = 0
    max_word = ""

#    num_words = 4
    for line in file:
        obj = Counter()
        obj.count = count_anagrams(obj, line.replace("\n", ""))  # -1 index means last one
#         print(line[0:-1], obj.count)
#         print(obj.count)
        if obj.count > max_count:
            max_count = obj.count
```

```
            max_word = line.replace("\n", "")
        #
        #       num_words -= 1
        #
        #       if num_words == 0:
        #           break


    print("Word with more anagrams: " + max_word)
    print("Max number of anagrams: " + str(max_count))



    #def preOrder(root):
    #    if not root:
    #        return
    #    print (root.key)
    #    preOrder(root.left)
    #    preOrder(root.right)

     #Creates the class of the counter to count the word with more anagrams and the number of anagrams
    class Counter:


        def __init__(self):
            self.count = 0


    def main():
```

```
    global english_words

    filename = "words_short.txt"

    tree_type = input("What type of Binary Search Tree do you want? Type 1 for AVL
Tree or 2 for Red-Black Tree: ")

    if tree_type == "1":
        english_words = AVLTree()
        english_words = AVL(filename)

    elif tree_type == "2":
        english_words = RedBlackTree()
        english_words = RBT(filename)

    anagram_word = input("Which word do you want to permutate? ")
    print_anagrams(anagram_word)

    obj = Counter()

    count_anagrams(obj, anagram_word)
    print("Number of permutations of the word: ", obj.count)

    max_anagrams(obj, filename)
```

main()