Timothy P. McCrary
11/08/18

CS 2302 Lab 3 – Option A

INTRODUCTION

For this lab, we are trying to find the similarities between any two words using the file "glove.6B.50d.txt" that has embeddings for every word in order to compare the two words. In order to find and access the embeddings for each word, we are using two types of Binary Search Trees, AVL Trees and Red-Black Trees, which will allow us to store the data, manipulate it, and search for it with relatively low running times. Additionally, four methods needed to be created that computed the number of nodes in the tree, compute the height of the tree, create a file with all the words in the tree, and create a file with all the words in the tree at a certain depth.

PROPOSED SOLUTION DESIGN AND IMPLEMENTATION

The goals of this lab were to compare two words, and complete the four methods a, b, c, d with both the AVL tree and the Red-Black Tree. The first thing that needed to be done was to implement both the AVL Tree and Red-Black Tree using the Zybooks code. Once the base code for both tree types were running, the next step was to modify the Zybook code to have a variable for the embedding and then read the file "glove.6B.50d.txt" and store the files data in the trees, with the key being the word, and the embeddings being stored in the embedding variable as an array(the array would be size 50 as there are 50 numbers for the embedding). Once the trees were populated, all that needed to be done was read another file that had two words per line, find the words in either the AVL Tree or the Red-Black tree, and use the embeddings of each word to find the cosine difference and print their similarities.

After being able to find the similarities of two words with both the AVL Tree and Red-Black tree, the next step was to create the four methods for a, b, c, d. Method "a" consisted of creating a method that counted the number of nodes in the tree. This method was rather easy to implement recursively, with a counter counting each node on the left and right and returning the total. This method was the same in both trees.

Second, method "b" wanted the height of the trees. The code for the Red-Black Tree from Zybooks already had a get height function, so being that the two trees are Binary Search Trees, the get height method also worked on the AVL Tree, so all that needed to be done was to put that method in AVL class.

Third method "c" was to create a method that generated a new file and stored all the words from the tree in ascending order. This was a little more complicated, but the algorithm was to go to the left most node as it would be the smallest and then add the word, then go check if there was anything on the right, if there was, the word was added, if not we go back up and repeat the process.

Finally, method "d" requested the same as "c", however, this time it needed to stop at a certain depth. Once solving method "c", this method was straight forward to implement, as all that needed to be done was add a counter that counted down until the desired depth and counted from there.

For this lab, input from the user needed to be take in consideration as they needed to choose the tree they would be using (AVL Tree or Red-Black Tree). In order to accomplish this, texts ask the user for which tree they want to use and gives them the option “1” for AVL Tree and “2” for Red-Black tree. If the user types anything else besides 1 or 2, they will be prompted again for an input until it is correct.

EXPERIMENTAL RESULTS

For this lab, four tests were conducted in order to test the AVL Tree and the Red-Black Tree. Each test changed the input (a .txt file) for the two words that would be compared. These are the file names: “TEST 1 – LAB EXAMPLE.txt”, “TEST 2 – EMPTY.txt”, “TEST3 – GIBERISH.txt”, and “TEST4 – NOT FORMATTED”.

TEST1 – LAB EXAMPLE.TXT

The .txt file for test 1 used the words provided as an example from the lab instructions. The file was formatted correctly and every word in the file was in the “glove.6B.50d.txt” file. This test was primarily used to make sure the code worked in the first place. This code successfully runs through every problem.

```
Hello! Would you like to use an AVL Tree or a Red-Black Tree?
Type "1" for AVL Tree or "2" Red-Black Tree
1
_________AVL Tree Implementation_________
The file you are using: "TEST1 - LAB EXAMPLE.txt"
Word Similarities:
barley   shrimp   0.6364096966538705
barley   oat   0.8246518214533809
federer   baseball   0.4197679910825971
federer   tennis   0.8789281450204323
harvard   utep   0.10645062709578756
harvard   ant   -0.043897105494225555
raven   crow   0.7513273690939416
raven   whale   0.5326563247467417
spain   france   0.9161746262011153
spain   mexico   0.8756780040055848
mexico   france   0.6609092641859666

Running time = 23.64241909980774 seconds

Solutions for a, b, c, and d:
(a)Number of Nodes in the tree: 355704
(b)Height of the tree: 21
(c)Putting all tree keys into file: "all_keys.txt"...
(c)Done!
(d)Putting tree keys at certain depth into file: "depth_keys.txt"...
(d)Done!

Process finished with exit code 0
```

```
Hello! Would you like to use an AVL Tree or a Red-Black Tree?
Type "1" for AVL Tree or "2" Red-Black Tree
2
__________Red-Black Tree Implementation__________
The file you are using: "TEST1 - LAB EXAMPLE.txt"
Word Similarities:
barley   shrimp   0.6364096966538705
barley   oat   0.8246518214533809
federer   baseball   0.4197679910825971
federer   tennis   0.8789281450204323
harvard   utep   0.10645062709578756
harvard   ant   -0.043897105494225555
raven   crow   0.7513273690939416
raven   whale   0.5326563247467417
spain   france   0.9161746262011153
spain   mexico   0.8756780040055848
mexico   france   0.6609092641859666

Running time = 13.547909021377563 seconds

Solutions for a, b, c, and d:
(a)Number of Nodes in the tree: 355704
(b)Height of the tree: 22
(c)Putting all tree keys into file: "all_keys.txt"...
(c)Done!
(d)Putting tree keys at certain depth into file: "depth_keys.txt"...
(d)Done!

Process finished with exit code 0
```

TEST2 – EMPTY.TXT

Test 2 was used to test what would happen if the program was given an empty file. All that was created was a .txt file with nothing in it. The program does catch this error, tells the user and then exits the program for both tree types.

```
Hello! Would you like to use an AVL Tree or a Red-Black Tree?
Type "1" for AVL Tree or "2" Red-Black Tree
1
__________AVL Tree Implementation__________
The file you are using: "TEST2 - EMPTY.txt"
Word Similarities:
This file is empty.
STOPPING PROGRAM.

Process finished with exit code 0
```

```
Hello! Would you like to use an AVL Tree or a Red-Black Tree?
Type "1" for AVL Tree or "2" Red-Black Tree
2
__________Red-Black Tree Implementation__________
The file you are using: "TEST2 - EMPTY.txt"
Word Similarities:
This file is empty.
STOPPING PROGRAM.

Process finished with exit code 0
```

TEST3 – GIBERISH.TXT

Test 3 was used to check what would happen if a word could not be found in "glove.6B.5d.txt". The file has correct formatting (two words per line, separated by a space) but instead of words, it is just random character to insure that in can not be found. The program does detect if the word cannot be found, it tells the user which word it is, and then exits the program.

```
Hello! Would you like to use an AVL Tree or a Red-Black Tree?
Type "1" for AVL Tree or "2" Red-Black Tree
1
__________AVL Tree Implementation__________
The file you are using: "TEST3 - GIBERISH.txt"
Word Similarities:
The word "23as" is not in the list of words.
Make sure it is spelt correctly.
STOPPING PROGRAM.

Process finished with exit code 0
```

```
Hello! Would you like to use an AVL Tree or a Red-Black Tree?
Type "1" for AVL Tree or "2" Red-Black Tree
2
__________Red-Black Tree Implementation__________
The file you are using: "TEST3 - GIBERISH.txt"
Word Similarities:
The word "23as" is not in the list of words.
Make sure it is spelt correctly.
STOPPING PROGRAM.

Process finished with exit code 0
```

TEST4 – NOT FORMATTED.TXT

This test was preformed to see what would happen if the input file is not formatted correctly. Once again a .txt was created with words, however is break the format of two words per line separated by a space. The program catches this, tells the user, and once again exits the program.

```
Hello! Would you like to use an AVL Tree or a Red-Black Tree?
Type "1" for AVL Tree or "2" Red-Black Tree
1
__________AVL Tree Implementation__________
The file you are using: "TEST4 - NOT FORMATTED.txt"
Word Similarities:
Input file not formatted correctly. Two words per line, separated by a space.
STOPPING PROGRAM.

Process finished with exit code 0
```

```
Hello! Would you like to use an AVL Tree or a Red-Black Tree?
Type "1" for AVL Tree or "2" Red-Black Tree
2
__________Red-Black Tree Implementation__________
The file you are using: "TEST4 - NOT FORMATTED.txt"
Word Similarities:
Input file not formatted correctly. Two words per line, separated by a space.
STOPPING PROGRAM.

Process finished with exit code 0
```

CONCLUSION

This lab allowed me to use and implement the AVL Tree and Red-Black Tree which help me understand in code how they work, and more importantly what makes them different but also the same. Furthermore, this lab showed me what makes the AVL Tree desirable and what makes the Red-Black Tree desirable in your code, and why you would use one over the other. In all, this lab helped me visualize and understand how these two trees work in code.

APPENDIX

```
#
********************************************************************************
******************************
# NAME: Timothy P. McCrary
# CLASS: CS 2302
# LAB 3 OPTION A
# INSTRUCTOR: Diego Aguirre
# TA: Manoj Pravaka Saha
# DATE: 11/1/2018
# PURPOSE: The purpose of this lab was to be able to create and use AVL Trees and Red-
Black Trees in order to search
# keys in each.
#
********************************************************************************
******************************
import math
import os
import sys
import time
```

```
two_words_file_name = "TEST1 - LAB EXAMPLE.txt"  # File that has the two words that
will be compared.
keys_to_file_name = "all_keys.txt"  # File that is created to write all the keys into
from one of trees.
keys_depth_to_file_name = "depth_keys.txt"  # File that is created to write keys at
certain depth from one of the trees.


def main():
    # Checks if the file given exists.
    if os.path.isfile(two_words_file_name) is False:
        print("File given to compare two words does't exist.\nSTOPPING PROGRAM")
        sys.exit()

    print("Hello! Would you like to use an AVL Tree or a Red-Black Tree?\nType \"1\"
for AVL Tree or \"2\" Red-Black Tree")
    user_input = input()

    # Loop that make sure the user inputs either 1 or 2 to select a tree type.
    correct_input = False
    while correct_input is False:
        if user_input == "1":
            correct_input = True

            print("__________AVL Tree Implementation__________")
            start_time = time.time()
            avltree = file_to_avltree("glove.6B.50d.txt")  # Reads file and puts it
into AVL Tree
            check_similarity(avltree, two_words_file_name)  # Checks similarity of
words from file given.
            print("\nRunning time = %s seconds" % (time.time() - start_time))
            print_solutions(avltree)  # Function that prints the methods a, b, c, d.

        elif user_input == "2":
            correct_input = True

            print("__________Red-Black Tree Implementation__________")
            start_time = time.time()
            rbtree = file_to_rbtree("glove.6B.50d.txt")  # Reads file and puts it into
Red-Black Tree.
            check_similarity(rbtree, two_words_file_name)  # Checks similarity of
words from file given.
            print("\nRunning time = %s seconds" % (time.time() - start_time))
            print_solutions(rbtree)  # Function that prints the methods a, b, c, d.

        else:
            print("Please type \"1\" for AVL Tree or \"2\" for Red-Black Tree.")
            user_input = input()


# Grabs the two words from file given, separates them into an array and then
# checks if the two words given are in the list of words with embeddings. Once the
words are found in its tree
# the embeddings are used to calculate their similarities and the results are printed.
def check_similarity(tree, file_name):
    print('The file you are using: "' + file_name + '"')
    print("Word Similarities:")
    file = open(file_name, "r")
    if file.readline() == "":
        print("This file is empty.\nSTOPPING PROGRAM.")
        sys.exit()
    file.close()
```

```
    file = open(file_name, "r")

    for line in file:
        file_word = line.split(" ", 1)
        if len(file_word) != 2:
            print("Input file not formatted correctly. Two words per line, separated
by a space.\nSTOPPING PROGRAM.")
            sys.exit()

        file_word[1] = file_word[1].replace("\n", "")

        word1_node = tree.search(file_word[0])
        word2_node = tree.search(file_word[1])

        if word1_node is None:
            print('The word \"' + file_word[0] + '\" is not in the list of
words.\nMake sure it is spelt correctly.\nSTOPPING PROGRAM.')
            sys.exit()
        if word2_node is None:
            print('The word \"' + file_word[1] + '\" is not in the list of
words.\nMake sure it is spelt correctly.\nSTOPPING PROGRAM.')
            sys.exit()

        dot_product = 0.0
        magnitude = 0.0
        for i in range(len(word1_node.embedding)):
            dot_product = word1_node.embedding[i] * word2_node.embedding[i] +
dot_product
            magnitude = math.sqrt(math.pow(word1_node.embedding[i], 2.0)) * math.sqrt(
                math.pow(word2_node.embedding[i], 2.0)) + magnitude
        similarity = dot_product / magnitude
        print(word1_node.key, " ", word2_node.key, " ", similarity)


# Takes a file with words and their embeddings and stores them into an AVL Tree and
then returns the tree.
def file_to_avltree(file_name):
    avltree = AVLTree()
    file = open(file_name, encoding="utf8")

    for line in file:
        if line.startswith(('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z')):
            s = line.split(" ", 1)
            string_embeddings = s[1].split(" ")
            float_embeddings = [float(i) for i in string_embeddings]
            node = Node(s[0], float_embeddings)
            avltree.insert(node)

    return avltree


# Takes a file with words and their embeddings and stores them into an Red-Black Tree
and then returns the tree.
def file_to_rbtree(file_name):
    rbtree = RedBlackTree()
    file = open(file_name, encoding="utf8")

    for line in file:
        if line.startswith(('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z')):
            s = line.split(" ", 1)
            string_embeddings = s[1].split(" ")
```

```
            float_embeddings = [float(i) for i in string_embeddings]
            rbtree.insert(s[0], float_embeddings)

    return rbtree


# Prints 4 solutions a, b, c, d
def print_solutions(tree):
    print("\nSolutions for a, b, c, and d:")
    print("(a)Number of Nodes in the tree:", tree.get_num_nodes())
    print("(b)Height of the tree:", tree.get_height())
    print('(c)Putting all tree keys into file: \"' + keys_to_file_name + '\"...')
    tree.tree_to_file(keys_to_file_name)
    print("(c)Done!")
    print('(d)Putting tree keys at certain depth into file: \"' +
keys_depth_to_file_name + '\"...')
    tree.tree_to_file_at_depth(10, keys_depth_to_file_name)
    print("(d)Done!")



# Rest of functions and methods are AVL Tree and Red-Black Tree given by Zybooks. Only
modifications are adding an
# embedding variable to both the AVL tree and the Red-Black Tree.
class Node:
    # Constructor with a key parameter creates the Node object.
    def __init__(self, key, embedding):
        self.key = key
        self.embedding = embedding
        self.parent = None
        self.left = None
        self.right = None
        self.height = 0

    # Method to calculate the current nodes's balance factor node,
    # 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 right subtree's current height, 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

    # Recalculates 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 new child's parent data member,
        # if the child is not None.
        if child is not None:
            child.parent = self

        # Update the node's height, since the structure
        # of the subtree 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 AVLTree:
    def __init__(self):
        # Constructor to create an empty AVLTree. There is only
        # one data member, the tree's root Node, and it starts
        # out as None.
        self.root = None

    # Performs a left rotation at the given node. Returns the
    # subtree's new root.
    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

    # Insert a new node into the AVLTree. When insert() is complete,
    # the AVL tree will be balanced.
    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 current_node
            elif current_node.key < key:
                current_node = current_node.right
            else:
                current_node = current_node.left

    # 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)

    # Removes the given node from the tree. The left and right subtrees,
    # if they exist, will be reattached to the tree such that no imbalances
    # exist, and the binary search tree property is maintained. Returns True
    # if the node is found and removed, or False if the node is not found in
    # the tree.es return after directly removing the node.
    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 != 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

    # Returns the height of this tree
    def get_height(self):
        return self._get_height_recursive(self.root)

    def _get_height_recursive(self, node):
        if node is None:
            return -1
        left_height = self._get_height_recursive(node.left)
        right_height = self._get_height_recursive(node.right)
        return 1 + max(left_height, right_height)

    # Returns the number of nodes in this tree
    def get_num_nodes(self):
        return self._get_num_nodes_recursive(self.root)

    def _get_num_nodes_recursive(self, node):
        if node is None:
            return 0
        left_height = self._get_num_nodes_recursive(node.left)
        right_height = self._get_num_nodes_recursive(node.right)
        return 1 + left_height + right_height

    # Puts all keys from tree into text file.
    def tree_to_file(self, file_name):
        file = open(file_name, "w", encoding="utf8")
        return self._tree_to_file_recursive(self.root, file)

    def _tree_to_file_recursive(self, node, file):

        if node is not None:
            self._tree_to_file_recursive(node.left, file)
            file.write(node.key)
            file.write("\n")
            self._tree_to_file_recursive(node.right, file)

    # Puts all keys from tree into text file at curtain depth.
    def tree_to_file_at_depth(self, depth, file_name):
        file = open(file_name, "w", encoding="utf8")
        return self._tree_to_file_at_depth_recursive(self.root, file, depth)

    def _tree_to_file_at_depth_recursive(self, node, file, depth):
        if depth == 0:
            return
        if node is not None:
            self._tree_to_file_at_depth_recursive(node.left, file, depth - 1)
            file.write(node.key)
```

```
            file.write("\n")
            self._tree_to_file_at_depth_recursive(node.right, file, depth - 1)


    # Overloading the __str__() operator to create a nicely-formatted text
representation of
    # the tree. Derived from Joohwan Oh at:
    #    https://github.com/joowani/binarytree/blob/master/binarytree/__init__.py
    def __str__(self):
        return pretty_tree(self)


# Derived from:
https://github.com/joowani/binarytree/blob/master/binarytree/__init__.py
# The following license applies to the TreePrint.py file only.
# MIT License

# Copyright (c) 2016 Joohwan Oh

# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:

# The above copyright notice and this permission notice shall be included in all
# copies or substantial portions of the Software.

# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
# SOFTWARE.
def _pretty_tree_helper(root, curr_index=0):
    if root is None:
        return [], 0, 0, 0

    line1 = []
    line2 = []
    node_repr = str(root.key)

    new_root_width = gap_size = len(node_repr)

    # Get the left and right sub-boxes, their widths, and root repr positions
    l_box, l_box_width, l_root_start, l_root_end = _pretty_tree_helper(root.left, 2 *
curr_index + 1)
    r_box, r_box_width, r_root_start, r_root_end = _pretty_tree_helper(root.right, 2 *
curr_index + 2)

    # Draw the branch connecting the current root to the left sub-box
    # Pad with whitespaces where necessary
    if l_box_width > 0:
        l_root = (l_root_start + l_root_end) // 2 + 1
        line1.append(' ' * (l_root + 1))
        line1.append('_' * (l_box_width - l_root))
        line2.append(' ' * l_root + '/')
        line2.append(' ' * (l_box_width - l_root))
        new_root_start = l_box_width + 1
        gap_size += 1
    else:
```

```
        new_root_start = 0

    # Draw the representation of the current root
    line1.append(node_repr)
    line2.append(' ' * new_root_width)

    # Draw the branch connecting the current root to the right sub-box
    # Pad with whitespaces where necessary
    if r_box_width > 0:
        r_root = (r_root_start + r_root_end) // 2
        line1.append('_' * r_root)
        line1.append(' ' * (r_box_width - r_root + 1))
        line2.append(' ' * r_root + '\\')
        line2.append(' ' * (r_box_width - r_root))
        gap_size += 1
    new_root_end = new_root_start + new_root_width - 1

    # Combine the left and right sub-boxes with the branches drawn above
    gap = ' ' * gap_size
    new_box = [''.join(line1), ''.join(line2)]
    for i in range(max(len(l_box), len(r_box))):
        l_line = l_box[i] if i < len(l_box) else ' ' * l_box_width
        r_line = r_box[i] if i < len(r_box) else ' ' * r_box_width
        new_box.append(l_line + gap + r_line)

    # Return the new box, its width and its root positions
    return new_box, len(new_box[0]), new_root_start, new_root_end


def pretty_tree(tree):
    lines = _pretty_tree_helper(tree.root, 0)[0]
    return '\n' + '\n'.join((line.rstrip() for line in lines))


#*****************************************************************************
*******************************


# RBTNode class - represents a node in a red-black tree
class RBTNode:
    def __init__(self, key, embedding, parent, is_red=False, left=None, right=None):
        self.key = key
        self.embedding = embedding
        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 _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)

    # Returns the height of this tree
    def get_height(self):
        return self._get_height_recursive(self.root)

    def _get_height_recursive(self, node):
```

```
        if node is None:
            return -1
        left_height = self._get_height_recursive(node.left)
        right_height = self._get_height_recursive(node.right)
        return 1 + max(left_height, right_height)

    def insert(self, key, embedding):
        new_node = RBTNode(key, embedding, 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)

    # Performs an in-order traversal, calling the visitor function for each node in
the tree
    def in_order(self, visitor_function):
        self.in_order_recursive(visitor_function, self.root)

    # Performs an in-order traversal
    def in_order_recursive(self, visitor_function, node):
        if node is None:
            return
        # Left subtree, then node, then right subtree
        self.in_order_recursive(visitor_function, node.left)
        visitor_function(node)
        self.in_order_recursive(visitor_function, 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 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 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) and
self.is_none_or_black(sibling.right) and 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) and self.is_not_none_and_red(
                sibling.right) and node is node.parent.right:
            sibling.color = "red"
            sibling.right.color = "black"
            self.rotate_left(sibling)
            return True
        return False  # not case 6

    # Returns the number of nodes in this tree
```

```
    def get_num_nodes(self):
        return self._get_num_nodes_recursive(self.root)

    def _get_num_nodes_recursive(self, node):
        if node is None:
            return 0
        left_height = self._get_num_nodes_recursive(node.left)
        right_height = self._get_num_nodes_recursive(node.right)
        return 1 + left_height + right_height

    # Puts all keys from tree into text file.
    def tree_to_file(self, file_name):
        file = open(file_name, "w", encoding="utf8")
        return self._tree_to_file_recursive(self.root, file)

    def _tree_to_file_recursive(self, node, file):

        if node is not None:
            self._tree_to_file_recursive(node.left, file)
            file.write(node.key)
            file.write("\n")
            self._tree_to_file_recursive(node.right, file)

    # Puts all keys from tree into text file at curtain depth.
    def tree_to_file_at_depth(self, depth, file_name):
        file = open(file_name, "w", encoding="utf8")
        return self._tree_to_file_at_depth_recursive(self.root, file, depth)

    def _tree_to_file_at_depth_recursive(self, node, file, depth):
        if depth == 0:
            return
        if node is not None:
            self._tree_to_file_at_depth_recursive(node.left, file, depth - 1)
            file.write(node.key)
            file.write("\n")
            self._tree_to_file_at_depth_recursive(node.right, file, depth - 1)


if __name__ == '__main__':
    main()
```

“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.”

X____Timothy P. McCrary____