Author: Carlos Fernando Castaneda

Class : CS 2302

Date Modified: October 22, 2019

Instructor: Diego Aguirre

Assignment: Lab 3 Binary Trees

TA: Gerardo Barraza

**Introduction:**

In python, binary trees can be used to search for items through a specific order, most notable is a binary search tree, which has an ordering property that any node's left subtree keys is less than or equal to the node's key, and the right subtree's keys is greater than or equal to the node's key. In this lab, we must implement two special trees called an AVL Tree, which is a BST(Binary Search Tree) that has a specific height balance and operations to construct itself, and a Red and Black Tree, which is a BST that with two node types, red and black, and its supporting operations ensure the tree is balanced when a node is inserted or removed. Once we implement the code for these trees, we must then insert a text file that will count the anagrams of a certain word, as well as create a method that will count the max number of anagrams of another implemented text file. In other words, this is going to be an interesting lab, nonetheless.

**Proposed solution design and implementation:**

Before I begin, there is a couple of things to note before I begin thinking about implementing anything in this lab to its potential. The first thing is that the professor and even the lab report mention that we can implement all the code from Zybooks for the AVL Tree and the Red and Black Tree including its pseudocode and concepts. The second thing to note so that I don’t occupy my program with unnecessary methods, there is no need to remove anything from the trees in this lab, so even though I could, I wont include any type of removal method in my code in this lab. All in all, with this in mind, the implementation of my code is two thirds done, now all we need is to create a main method.

One thing that I want have implement for sure is the fact that the program must ask the user what tree they want to use to implement the text files. This can be done easily by having a lot of if else statements looking for specific inputs. I would ideally like the inputs to be 1 and 2 for the options, as I personally am not a big fan of having a letter as an input.

The way I want to implement the method that will read the text file for the anagram is simple , I want to be able first insert the file through the means of either trees I need to implement, then I plan to use the print\_anagrams method to find the solution for the first problem. Of course, I would need to alter it so that it can read it with my trees and strings instead of numbers.

The second method, which is to find count anagrams, should do exactly as the lab indicates it to be, which is to return and print the number of anagrams that a certain word has, for that I plan to have deviation of the print anagram, but only counting the anagrams instead of the strings.

The last method, which I shall call it maximum\_anagrams, has to read another text file, and return the word with the greatest number of anagrams from the rest. To do this, I plan to read the text file, insert in into either an AVL or Red and Black tree, and then compare the count anagrams from each word, and return the one that has the highest count.

**Experimental results**:

Before I begin explaining my results, I want to note that I split my program into three separate files, one for the main method, one for the AVL Tree implementation, and One for the Red and Black Implementation. I mainly had to do this due to the fact that the two trees have different Node functions, and I found that my implementation was simpler if I just split the trees into their own class, and I just imported them into the main program like how the professor did in our midterm 1 programs. I also don’t see any reason to explain these Tree classes as they are brought straight from Zybooks but have been altered so that it can be used in Python. With that out of the way, let’s begin!

Like I mentioned before, when the program boots up, I have the user ask what text file they want to use during the program, as well as asking which Tree they wanted to use. Sadly, I wasn’t able use numbers as input as the program kept giving me an error, but I was able to us either the letter a or b as input, including caps so that the program won’t freak out.

The tree methods were very simple to implement, as I had to do is line by line read the strings, and insert into the respective trees, the hardest part of this lab for sure was to translate the pseudocode from zybooks into python, but once I was done with that, it was smooth sailing for this methods at least. I also want to note at this time that I implemented a running time function for the AVL and Red and Black Trees for a reason I will come back to later in the report.

The count anagrams method is a little cumbersome, as I had to alter the print anagrams method to meet the needs for this lab, and the solution was not as easy I thought it would be, but at the end all I needed to do for this method was instead of printing the anagram for that particular word, I just kept a counter that will return to the main method once it found the desired integer. I also want to mention that I used keys to find each count for each specific word.

The last method, maximum\_anagrams, also uses a user interface, as it can ask the user if it want to find or leave the program. I did this on purpose as while developing, I constantly needed to just find that Tree implementation worked well enough. I also want to mention now that the reason I had a running time for each Tree method was to check which one was the fastest to use for this method and I found that using the Red and Black Tree was the quickest implementation, so I used that Tree to use compare the new inserted text file. The method acts the same as count anagram, except for one minor difference, instead of just returning one count, it will return the biggest count from all the strings contained in that text file.

Below you will find the results from various states in the program…

My results from running an AVL Tree, finding the word ‘weird’:

**A screenshot of a social media post

Description automatically generated**

My results from running a Red-Black Tree, finding the word ‘response’:

**A screenshot of a social media post

Description automatically generated**

**Run times:**

|  |  |  |
| --- | --- | --- |
| **Time** | **AVL Tree** | **Red Black Tree** |
| **1** | **23.94 seconds** | **24.82 seconds** |
| **2** | **24.28 seconds** | **26.02 seconds** |
| **3** | **24.47 seconds** | **23.54 seconds** |

**Conclusions**:

With this lab, I was able to learn to code better using the Python language, including using multiple python files to create different binary search trees. I was also able to implement a way to import a text file and insert the strings inside of either an AVL or Red Black tree.

**Appendix :**

**Main.py**

|  |
| --- |
| ''' |
|  | Author: Carlos Fernando Castaneda |
|  | Class : CS 2302 |
|  | Date Modified: October 21, 2019 |
|  | Instructor: Diego Aguirre |
|  | Assingment: Lab 3 Binary Trees (Part 1 of 3) |
|  | TA: Gerardo Barraza |
|  | Purpose: To practice using multiple methods |
|  | of creating Binary Trees, includong AVL and |
|  | Red-Black. |
|  | ''' |
|  | #Used to calculate the time for each tree |
|  | import time |
|  | #Imports the AVL\_Trees and RB\_Trees programs to build the binary trees |
|  | import AVL\_Trees |
|  | import RB\_Trees |
|  |  |
|  | #Method that creates and returns an AVL tree from the imported file. |
|  | def AVL(filename): |
|  | AVL = AVL\_Trees.AVLTree() |
|  | key = 0 |
|  | print("Preparing the AVL Tree now! Please wait...") |
|  | print() |
|  | with open(filename, encoding='windows-1252') as textFile: |
|  | for line in textFile: |
|  | string = line.split() |
|  | word = str(string) |
|  | stringedWord = word[2:len(word) - 2] |
|  | key = make\_key(stringedWord) |
|  | AVL.AVLTreeInsert(stringedWord, key) |
|  | return AVL |
|  |  |
|  | #Method that creates and returns an Red and Black tree from the imported file. |
|  | def RB(filename): |
|  | RB = RB\_Trees.RedBlack() |
|  | key = 0 |
|  | print("Preparing the Red and Black Tree now! Please wait...") |
|  | print() |
|  | with open(filename, encoding='windows-1252') as textFile: |
|  | for line in textFile: |
|  | string = line.split() |
|  | word = str(string) |
|  | stringedWord = word[2:len(word) - 2] |
|  | key = make\_key(stringedWord) |
|  | RB.RBTreeInsert(stringedWord, key) |
|  | return RB |
|  |  |
|  | #Method that finds the necessary anagrams for both the maximum and count methods for the anagrams by using a binary search tree |
|  | def search\_anagrams(tree, word, prefix=""): # |
|  | if len(word) <= 1: |
|  | str = prefix + word |
|  | key = make\_key(str) |
|  | searched\_word = BSTsearch(tree, key) |
|  | if (searched\_word != None): |
|  | return 1 |
|  | return 0 |
|  | else: |
|  | count\_anagrams = 0 |
|  | for i in range(len(word)): |
|  | cur = word[i: i + 1] |
|  | before = word[0: i] |
|  | after = word[i + 1:] |
|  | if cur not in before: |
|  | count\_anagrams += search\_anagrams(tree, before + after, prefix + cur) |
|  | return count\_anagrams |
|  |  |
|  | #Method that acts a binary search tree that will find the necessary anagrams for the find |
|  | def BSTsearch(tree, key): |
|  | cur = tree.root |
|  | while cur != None: |
|  | if key == cur.key: |
|  | return cur |
|  | elif key < cur.key: |
|  | cur = cur.left |
|  | else: |
|  | cur = cur.right |
|  | return None |
|  |  |
|  | #This method generates keys for each word by stringing together character numbers of each letter of a word. |
|  | def make\_key(word): |
|  | key = "" |
|  | res = list(word) |
|  | if (len(res[0]) > 1): |
|  | str\_word = str(res[0]) |
|  | return make\_key(str\_word) |
|  | for i in range(len(res)): |
|  | char = caps(res[i]) |
|  | char\_num = ord(char) |
|  | char\_str = str(char\_num) |
|  | key += char\_str |
|  | key\_int = int(key) |
|  | return key\_int |
|  |  |
|  | #Method activated by make key that will find the lower case letter by comparing it to its caps counterpart. |
|  | def caps(char): |
|  | cap\_list = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'N', 'M', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z'] |
|  | lower\_list = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'n', 'm', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'] |
|  | lower\_char = '' |
|  | for i in range(len(cap\_list)): |
|  | if (char == cap\_list[i]): |
|  | lower\_char = lower\_list[i] |
|  | return lower\_char |
|  | return char |
|  |  |
|  | #Method that counts the number of anagrams a certain word has. |
|  | def count\_anagrams(tree): |
|  | print("") |
|  | print("Please input a word for counting the number of anagrams:") |
|  | new\_word = input() |
|  | count = search\_anagrams(tree, new\_word) |
|  | print("") |
|  | print("Findings complete! Total was:", count) |
|  | print() |
|  |  |
|  | #Method that finds the maximum number of anagrams from another text file, the user can choose to ignore this part however. |
|  | def maximum\_anagrams(): |
|  | print("Want to load another file to find the word that has the most anagrams? ") |
|  | print("A. Yes") |
|  | print("B. No") |
|  | user\_selection = input() |
|  | count = 0 |
|  | if (user\_selection == "A" or user\_selection == "a"): |
|  | print("") |
|  | print("Input filename of the textfile:") |
|  | filename = input() |
|  | tree = RB(filename) |
|  | anagrams = "" |
|  | anagrams\_num = 0 |
|  | with open(filename, encoding='windows-1252') as textFile: |
|  | for line in textFile: |
|  | string = line.split() |
|  | word = str(string) |
|  | word\_str = word[2:len(word) - 2] |
|  | count = search\_anagrams(tree, word\_str) |
|  | if (anagrams\_num < count): |
|  | anagrams = word\_str |
|  | anagrams\_num = count |
|  | print("The word with the most anagrams is '", anagrams, "' with the total of", anagrams\_num, "anagrams.") |
|  | print("Terminating program, Farewell!") |
|  | elif (user\_selection == "B" or user\_selection == "b"): |
|  | print("User selected to exiting the program.") |
|  | print("Farewell.") |
|  | elif count == 3: |
|  | print("ERROR: Too many incorrect statements!") |
|  | print("Farewell.") |
|  | else: |
|  | print("ERROR: Input not valid!") |
|  | count+=1 |
|  | maximum\_anagrams() |
|  |  |
|  | #Main method of the program, this method is used for the user to select the options for the rest of the program. |
|  | def main(): |
|  | print("Welcome to the Anagram program! Please select the .txt file you want to implement:") |
|  | file = input() |
|  | print("") |
|  | print("Do you want to use an AVL tree or a Black-Red tree? Select from below:") |
|  | print("A. AVL Tree") |
|  | print("B. Red-Black Tree") |
|  | user\_selection = input() |
|  | count = 0 |
|  | if (user\_selection == 'A' or user\_selection == 'a'): |
|  | start1 = time.time() |
|  | tree = AVL(file) |
|  | end1 = time.time() |
|  | print("Tree insertion complete!") |
|  | print('Running time was: ', end1 - start1, 'seconds.') |
|  | count\_anagrams(tree) |
|  | elif (user\_selection == 'B' or user\_selection == 'b'): |
|  | start2 = time.time() |
|  | tree = RB(file) |
|  | end2 = time.time() |
|  | print("Tree insertion complete!") |
|  | print('Running time was: ', end2 - start2, 'seconds.') |
|  | count\_anagrams(tree) |
|  | elif count == 3: |
|  | print("ERROR: Too many incorrect statements!") |
|  | print("Farewell.") |
|  | else: |
|  | print("ERROR: Input not valid!") |
|  | count+=1 |
|  | main() |
|  |  |
|  | main() |
|  | maximum\_anagrams() |

**AVL\_Trees.py**

|  |
| --- |
| ''' |
|  | Author: Carlos Fernando Castaneda |
|  | Class : CS 2302 |
|  | Date Modified: October 21, 2019 |
|  | Instructor: Diego Aguirre |
|  | Assingment: Lab 3 Binary Trees (Part 2 of 3) |
|  | TA: Gerardo Barraza |
|  | Purpose: To practice using multiple methods |
|  | of creating Binary Trees, includong AVL and |
|  | Red-Black. |
|  | ''' |
|  | class AVLNode(object): |
|  | def \_\_init\_\_(self, word, key, left = None, parent = None, right = None): |
|  | self.word = word #Added attribute |
|  | self.key = 0 |
|  | self.left = None |
|  | self.parent = None |
|  | self.right = None |
|  | self.height = -1 |
|  |  |
|  | class AVLTree(object): |
|  |  |
|  | def \_\_init\_\_(self, root = None): |
|  | self.root = None |
|  |  |
|  | def AVLTreeUpdateHeight(self, node): |
|  | left\_height = -1 |
|  | if (node.left != None): |
|  | left\_height = node.left.height |
|  | right\_height = -1 |
|  | if (node.right != None): |
|  | right\_height = node.right.height |
|  | node.height = max(left\_height, right\_height) + 1 |
|  |  |
|  | def AVLTreeSetChild(self, parent, which\_child, child): |
|  | if (which\_child != "left" and which\_child != "right"): |
|  | return False |
|  | if (which\_child == "left"): |
|  | parent.left = child |
|  | else: |
|  | parent.right = child |
|  | if (child != None): |
|  | child.parent = parent |
|  | self.AVLTreeUpdateHeight(parent) |
|  | return True |
|  |  |
|  | def AVLTreeReplaceChild(self, parent, curr\_child, new\_child): |
|  | if (parent.left == curr\_child): |
|  | return self.AVLTreeSetChild(parent, "left", new\_child) |
|  | elif (parent.right == curr\_child): |
|  | return self.AVLTreeSetChild(parent, "right", new\_child) |
|  | return False |
|  |  |
|  | def AVLTreeGetBalance(self, node): |
|  | left\_height = -1 |
|  | if (node.left != None): |
|  | left\_height = node.left.height |
|  | right\_height = -1 |
|  | if (node.right != None): |
|  | right\_height = node.right.height |
|  | return left\_height - right\_height |
|  |  |
|  | def AVLTreeRotateRight(self, node): |
|  | left\_right\_child = node.left.right |
|  | if (node.parent != None): |
|  | self.AVLTreeReplaceChild(node.parent, node, node.left) |
|  | else: # node is root |
|  | self.root = node.left |
|  | self.root.parent = None |
|  |  |
|  | self.AVLTreeSetChild(node.left, "right", node) |
|  | self.AVLTreeSetChild(node, "left", left\_right\_child) |
|  |  |
|  | def AVLTreeRotateLeft(self, node): |
|  | right\_left\_child = node.right.left |
|  | if (node.parent != None): |
|  | self.AVLTreeReplaceChild(node.parent, node, node.right) |
|  | else: |
|  | self.root = node.right |
|  | self.root.parent = None |
|  | self.AVLTreeSetChild(node.right, "left", node) |
|  | self.AVLTreeSetChild(node, "right", right\_left\_child) |
|  |  |
|  | def AVLTreeRebalance(self, node): |
|  | self.AVLTreeUpdateHeight(node) |
|  | if (self.AVLTreeGetBalance(node) == -2): |
|  | if (self.AVLTreeGetBalance(node.right) == 1): |
|  | self.AVLTreeRotateRight(node.right) |
|  | return self.AVLTreeRotateLeft(node) |
|  | elif (self.AVLTreeGetBalance(node) == 2): |
|  | if (self.AVLTreeGetBalance(node.left) == -1): |
|  | self.AVLTreeRotateLeft(node.left) |
|  | return self.AVLTreeRotateRight(node) |
|  | return node |
|  |  |
|  | def AVLTreeInsert(self, words, key): |
|  | node = AVLNode(words, key) |
|  | node.key = key |
|  | if (self.root == None): |
|  | self.root = node |
|  | node.parent = None |
|  | return |
|  | cur = self.root |
|  | while (cur != None): |
|  | if (node.key < cur.key): |
|  | if (cur.left == None): |
|  | cur.left = node |
|  | node.parent = cur |
|  | cur = None |
|  | else: |
|  | cur = cur.left |
|  | else: |
|  | if (cur.right == None): |
|  | cur.right = node |
|  | node.parent = cur |
|  | cur = None |
|  | else: |
|  | cur = cur.right |
|  | node = node.parent |
|  | while (node != None): |
|  | self.AVLTreeRebalance(node) |
|  | node = node.parent |

**RB\_Trees.py**

|  |
| --- |
| ''' |
|  | Author: Carlos Fernando Castaneda |
|  | Class : CS 2302 |
|  | Date Modified: October 21, 2019 |
|  | Instructor: Diego Aguirre |
|  | Assingment: Lab 3 Binary Trees (Part 3 of 3) |
|  | TA: Gerardo Barraza |
|  | Purpose: To practice using multiple methods |
|  | of creating Binary Trees, includong AVL and |
|  | Red-Black. |
|  | ''' |
|  | class RBNode(object): |
|  | def \_\_init\_\_(self, word = "", key = 0, left = None, parent = None, right = None, color = -1): |
|  | self.word = word |
|  | self.key = 0 |
|  | self.left = None |
|  | self.parent = None |
|  | self.right = None |
|  | self.color = -1 |
|  |  |
|  | class RedBlack(object): |
|  | def \_\_init\_\_(self, root = None): |
|  | self.root = None |
|  |  |
|  | def RBTreeSetChild(self, parent, which\_child, child): |
|  | if (which\_child != "left" and which\_child != "right"): |
|  | return False |
|  | if (which\_child == "left"): |
|  | parent.left = child |
|  | else: |
|  | parent.right = child |
|  | if (child != None): |
|  | child.parent = parent |
|  | return True |
|  |  |
|  | def RBTreeReplaceChild(self, parent, current\_child, new\_child): |
|  | if (parent.left == current\_child): |
|  | return self.RBTreeSetChild(parent, "left", new\_child) |
|  | elif (parent.right == current\_child): |
|  | return self.RBTreeSetChild(parent, "right", new\_child) |
|  | return False |
|  |  |
|  | def RBTreeRotateLeft(self, node): |
|  | right\_left\_child = node.right.left |
|  | if (node.parent != None): |
|  | self.RBTreeReplaceChild(node.parent, node, node.right) |
|  | else: |
|  | self.root = node.right |
|  | self.root.parent = None |
|  | self.RBTreeSetChild(node.right, "left", node) |
|  | self.RBTreeSetChild(node, "right", right\_left\_child) |
|  |  |
|  | def RBTreeRotateRight(self, node): |
|  | left\_right\_child = node.left.right |
|  | if (node.parent != None): |
|  | self.RBTreeReplaceChild(node.parent, node, node.left) |
|  | else: |
|  | self.root = node.left |
|  | self.root.parent = None |
|  | self.RBTreeSetChild(node.left, "right", node) |
|  | self.RBTreeSetChild(node, "left", left\_right\_child) |
|  |  |
|  | def RBTreeInsert(self, words, key): |
|  | node = RBNode(words, key) |
|  | node.key = key |
|  | self.BSTinsert(node) |
|  | node.color = 1 |
|  | self.RBTreeBalance(node) |
|  |  |
|  | def RBTreeGetGrandparent(self, node): |
|  | if (node.parent == None): |
|  | return None |
|  | return node.parent.parent |
|  |  |
|  | def RBTreeGetUncle(self, node): |
|  | grandparent = None |
|  | if (node.parent != None): |
|  | grandparent = node.parent.parent |
|  | if (grandparent == None): |
|  | return None |
|  | if (grandparent.left == node.parent): |
|  | return grandparent.right |
|  | else: |
|  | return grandparent.left |
|  | def RBTreeBalance(self, node): |
|  | if (node.parent == None): |
|  | node.color = 0 |
|  | return |
|  | if (node.parent.color == 0): |
|  | return |
|  | parent = node.parent |
|  | grandparent = self.RBTreeGetGrandparent(node) |
|  | uncle = self.RBTreeGetUncle(node) |
|  | if (uncle != None and uncle.color == 1): |
|  | parent.color = uncle.color = 0 |
|  | grandparent.color = 1 |
|  | self.RBTreeBalance(grandparent) |
|  | return |
|  | if (node == parent.right and parent == grandparent.left): |
|  | self.RBTreeRotateLeft(parent) |
|  | node = parent |
|  | parent = node.parent |
|  | elif (node == parent.left and parent == grandparent.right): |
|  | self.RBTreeRotateRight(parent) |
|  | node = parent |
|  | parent = node.parent |
|  | parent.color = 0 |
|  | grandparent.color = 1 |
|  | if (node == parent.left): |
|  | self.RBTreeRotateRight(grandparent) |
|  | else: |
|  | self.RBTreeRotateLeft(grandparent) |
|  |  |
|  | def BSTinsert(self, node): |
|  | if (self.root == None): |
|  | self.root = node |
|  | node.left = None |
|  | node.right = None |
|  | else: |
|  | cur = self.root |
|  | while (cur != None): |
|  | if (node.key < cur.key): |
|  | if (cur.left == None): |
|  | cur.left = node |
|  | cur = None |
|  | else: |
|  | cur = cur.left |
|  | else: |
|  | if (cur.right == None): |
|  | cur.right = node |
|  | cur = None |
|  | else: |
|  | cur = cur.right |
|  | node.left = None |
|  | node.right = None |

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 provide inappropriate assistance to any student in the class.

