LAB #4 Report

***TASK***:

Write another function called count\_anagrams that does not produce output, but returns the number of anagrams that a given word has. For example, count\_anagrams(”spot”) should return 6. Finally, write another function that reads another file that contains words (feel free to create it yourself) and finds the word in the file that has the greatest number of anagrams.

def print\_anagrams(word, prefix=""):

if len(word) <= 1:

str = prefix + word

if str in engish\_words:

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)

The method uses a data structure called engish\_words to determine if a given anagram is a valid word in the English language. You can think of engish\_words as a container of all the words in the English language. We will implement this data structure using a binary search tree. To populate engish\_words, we will use a text file called words.txt that contains 354,984 English words. You can download words.txt from the following URL: <https://github.com/dwyl/english-words/>. Once you have downloaded words.txt, write a function that reads the file and populates the binary search tree with all the English words contained in the file. Ask the user what type of binary search tree he/she wants to use (AVL Tree or Red-Black Tree). You are free to use the implementation provided in your zyBook for these two types of trees. Adapt zyBook’s code to include the word and make it the key.

Write another function called count\_anagrams that does not produce output, but returns the number of anagrams that a given word has. For example, count\_anagrams(”spot”) should return 6. Finally, write another function that reads another file that contains words (feel free to create it yourself) and finds the word in the file that has the greatest number of anagrams.

***APPROACH:***

To accomplish this lab, I first had to investigate and understand the correct usage of AVL and Black-Red trees. Once I correctly understood how to use the trees, the lab was quite straight-forward. I then implemented the methods to how the lab is required to have; since the anagram algorithm was provided, I just needed to implement it by reading the file.

Once I understood how to do it in an AVL tree, the logic works well. Now all I have to do is implement it with a B-Tree data Structure. For this, I used MR. Aguirre’s Blackboard resource code. After that it was quite simple.

import matplotlib.pyplot as plt

# from AVLtree import AVLTree

# from RedBlackTree import RedBlackTree

# # from B-tree import BTree

class Node:

def \_\_init\_\_(self, key):

self.key = key

self.left = None

self.right = None

class BTreeNode:

# Constructor

def \_\_init\_\_(self, keys=[], children=[], is\_leaf=True, max\_num\_keys=5):

self.keys = keys

self.children = children

self.is\_leaf = is\_leaf

if max\_num\_keys < 3: # max\_num\_keys must be odd and greater or equal to 3

max\_num\_keys = 3

if max\_num\_keys % 2 == 0: # max\_num\_keys must be odd and greater or equal to 3

max\_num\_keys += 1

self.max\_num\_keys = max\_num\_keys

def is\_full(self):

return len(self.keys) >= self.max\_num\_keys

class BTree:

# Constructor

def \_\_init\_\_(self, max\_num\_keys=5):

self.max\_num\_keys = max\_num\_keys

self.root = BTreeNode(max\_num\_keys=max\_num\_keys)

def find\_child(self, k, node=None):

# Determines value of c, such that k must be in subtree node.children[c], if k is in the BTree

if node is None:

node = self.root

for i in range(len(node.keys)):

if k < node.keys[i]:

return i

return len(node.keys)

def insert\_internal(self, i, node=None):

if node is None:

node = self.root

# node cannot be Full

if node.is\_leaf:

self.insert\_leaf(i, node)

else:

k = self.find\_child(i, node)

if node.children[k].is\_full():

m, l, r = self.split(node.children[k])

node.keys.insert(k, m)

node.children[k] = l

node.children.insert(k + 1, r)

k = self.find\_child(i, node)

self.insert\_internal(i, node.children[k])

def split(self, node=None):

if node is None:

node = self.root

# print('Splitting')

# PrintNode(T)

mid = node.max\_num\_keys // 2

if node.is\_leaf:

left\_child = BTreeNode(node.keys[:mid], max\_num\_keys=node.max\_num\_keys)

right\_child = BTreeNode(node.keys[mid + 1:], max\_num\_keys=node.max\_num\_keys)

else:

left\_child = BTreeNode(node.keys[:mid], node.children[:mid + 1], node.is\_leaf, max\_num\_keys=node.max\_num\_keys)

right\_child = BTreeNode(node.keys[mid + 1:], node.children[mid + 1:], node.is\_leaf, max\_num\_keys=node.max\_num\_keys)

return node.keys[mid], left\_child, right\_child

def insert\_leaf(self, i, node=None):

if node is None:

node = self.root

node.keys.append(i)

node.keys.sort()

def leaves(self, node=None):

if node is None:

node = self.root

# Returns the leaves in a b-tree

if node.is\_leaf:

return [node.keys]

s = []

for c in node.children:

s = s + self.leaves(c)

return s

def insert(self, i, node=None):

if node is None:

node = self.root

if not node.is\_full():

self.insert\_internal(i, node)

else:

m, l, r = self.split(node)

node.keys = [m]

node.children = [l, r]

node.is\_leaf = False

k = self.find\_child(i, node)

self.insert\_internal(i, node.children[k])

def height(self, node=None):

if node is None:

node = self.root

if node.is\_leaf:

return 0

return 1 + self.height(node.children[0])

def print(self, node=None):

# Prints keys in tree in ascending order

if node is None:

node = self.root

if node.is\_leaf:

for t in node.keys:

print(t, end=' ')

print('\n')

else:

for i in range(len(node.keys)):

self.print(node.children[i])

print(node.keys[i], end=' ')

print('\n')

self.print(node.children[len(node.keys)])

def print\_d(self, space, node=None):

if node is None:

node = self.root

# Prints keys and structure of B-tree

if node.is\_leaf:

for i in range(len(node.keys) - 1, -1, -1):

print(space, node.keys[i])

else:

self.print\_d(space + ' ', node.children[len(node.keys)])

for i in range(len(node.keys) - 1, -1, -1):

print(space, node.keys[i])

self.print\_d(space + ' ', node.children[i])

def search(self, k, node=None):

if node is None:

node = self.root

# Returns node where k is, or None if k is not in the tree

if k in node.keys:

return node

if node.is\_leaf:

return None

return self.search(k, node.children[self.find\_child(k, node)])

def \_set\_x(self, dx, node=None):

if node is None:

node = self.root

# Finds x-coordinate to display each node in the tree

if node.is\_leaf:

return

else:

for c in node.children:

self.\_set\_x(dx, c)

d = (dx[node.children[0].keys[0]] + dx[node.children[-1].keys[0]] + 10 \* len(node.children[-1].keys)) / 2

dx[node.keys[0]] = d - 10 \* len(node.keys) / 2

def \_draw\_btree(self, dx, y, y\_inc, fs, ax, node=None):

if node is None:

node = self.root

# Function to display b-tree to the screen

# It works fine for trees with up to about 70 keys

xs = dx[node.keys[0]]

if node.is\_leaf:

for itm in node.keys:

ax.plot([xs, xs + 10, xs + 10, xs, xs], [y, y, y - 10, y - 10, y], linewidth=1, color='k')

ax.text(xs + 5, y - 5, str(itm), ha="center", va="center", fontsize=fs)

xs += 10

else:

for i in range(len(node.keys)):

xc = dx[node.children[i].keys[0]] + 5 \* len(node.children[i].keys)

ax.plot([xs, xs + 10, xs + 10, xs, xs], [y, y, y - 10, y - 10, y], linewidth=1, color='k')

ax.text(xs + 5, y - 5, str(node.keys[i]), ha="center", va="center", fontsize=fs)

ax.plot([xs, xc], [y - 10, y - y\_inc], linewidth=1, color='k')

self.\_draw\_btree(dx, y - y\_inc, y\_inc, fs, ax, node.children[i])

xs += 10

xc = dx[node.children[-1].keys[0]] + 5 \* len(node.children[-1].keys)

ax.plot([xs, xc], [y - 10, y - y\_inc], linewidth=1, color='k')

self.\_draw\_btree(dx, y - y\_inc, y\_inc, fs, ax, node.children[-1])

def draw(self):

# Find x-coordinates of leaves

ll = self.leaves()

dx = {}

d = 0

for l in ll:

dx[l[0]] = d

d += 10 \* (len(l) + 1)

# Find x-coordinates of internal nodes

self.\_set\_x(dx)

# plt.close('all')

fig, ax = plt.subplots()

self.\_draw\_btree(dx, 0, 30, 10, ax)

ax.set\_aspect(1.0)

ax.axis('off')

plt.show()

# code given from lab assignment

def print\_anagrams(word, english\_words, prefix=""):

if len(word) <= 1:

str = prefix + word

if str in english\_words:

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(word, english\_words, prefix=""): #this method does not work and was supoost to permute word and increment count of words

global count

count = 0

if len(word) <= 1:

elem = prefix + word

count += 1

if english\_words.search(prefix + word, english\_words.root):

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(before + after, english\_words, prefix + cur)

return count

# Reads file and puts them in the avl tree

def AVL\_read\_file(): #method to read file

word\_file = open("words.txt", "r")

avl = AVLTree()

for line in word\_file:

word = line.replace("\n", "")

avl.AVL\_insert(word)

return avl

# Reads file and puts them in the rb tree

def RB\_read\_file(): #method to read my file

word\_file = open("words.txt", "r")

line = word\_file.readline()

rb = RedBlackTree()

for line in word\_file:

word = line.replace("\n", "")

rb.RB\_insert(word)

return rb

def Btree\_read\_file():

word\_file = open("short.txt", "r")

btree = BTree()

for line in word\_file:

word = line.replace("\n", "")

btree.insert(word)

return btree

def main():

user\_option = input("What type of tree would you like to use? Type 'a' for AVL or 'b' for Red-Black tree or 'c' for Btree ")

if user\_option is "a":

avl = AVL\_read\_file() # Tree

avl.print\_tree(avl.root) # printing, not supposed to return anything

print(count\_anagrams("spot", avl, prefix=""))

elif user\_option is "b":

r = RB\_read\_file()

r.print\_tree(r.root)

elif user\_option is "c":

b = Btree\_read\_file()

b.print(b.root)

print("choose a word to find anagrams for: ")

user\_input = input()

print\_anagrams(user\_input, b)

print("total number of anagrams: ")

count\_words = count\_anagrams(user\_input, b)

if count\_words == 0:

print("No Anagrams found")

else:

print(count\_words)

else:

print("invalid option, try again.")

# print(print\_anagrams())

main()