Andres Nava

3-10-19

CS 2302 Data Structures

1:30-2:50

80532614

**Introduction:**

This lab required us to get a sorted list and make a balanced b tree, find certain numbers, print out number s at a certain depth, and extract elements into a sorted list.

**Solutions:**

So for the iterative version of finding out if k was in the list I basically checked if the list was empty if it was then k was not there. I then checked if it was the root of the list after that I checked if it was greater or smaller than the root to see which way to send it.

For the balanced tree I essentially checked if the list was empty if it is then return nothing. I also checked if the length was one and if it was then just return the root. If it wasn’t I got half the length of the sorted list then got that specific item and made that the root then I set everything else to there respected areas.

For the elements into a sorted list I essentially just appended the numbers into there respected areas

For the print out the numbers in depth I checked if it was empty. I then checked if k was 0 if it is then just print out that layer. If Its not then call it again and make k -1 until it gets to 0.

**Experimental :**

For this one I had a specific sorted list made to see if all my methods would work. For each and every different method I had to see wat went wrong and had to fix the individual problems that had occurred. For the balanced tree method I used a way to get the middle number in the list and made that the root so that way it could be balanced.   
**Conclusion:**

I learned how to use B trees and how to make them, sort them, and then make them into a balanced tree

**Big O():**

Method1: O(n)

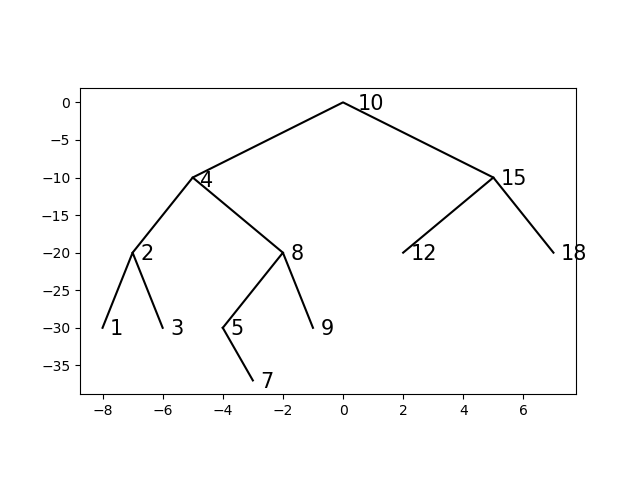
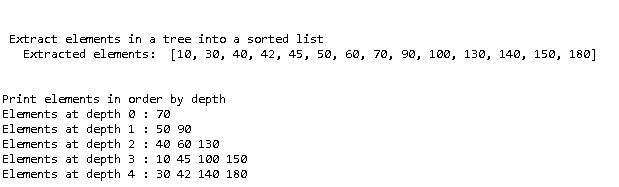
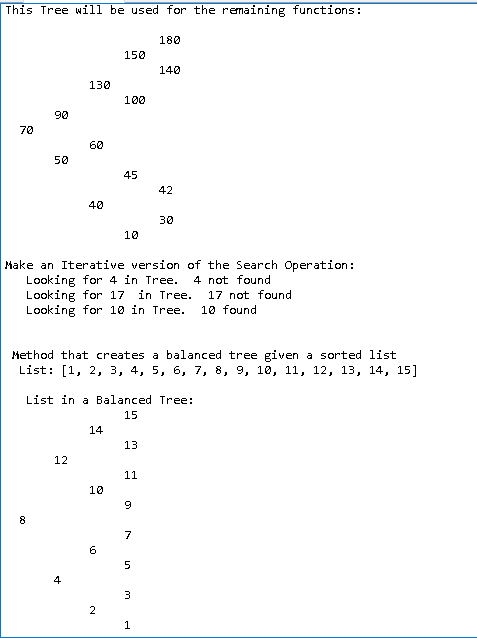
Method2:O(log(n))

Method3:O(n)

Method4:O(n)

Method5:O(n)

**Pictures:**



**Appendix:**

#Author: Andres Nava

#LDOM:3-10-19

#Course:CS 2302 Data Structures

#Lab 3

#TA:Anindita Nath

#Purpose: The purpose of this lab is too make a balanced btree and to find certain number, print out number

#at certain depth,and extract element into a sorted list.

#

import matplotlib.pyplot as plt

class BST(object):

# Constructor

def \_\_init\_\_(self, item, left=None, right=None):

self.item = item

self.left = left

self.right = right

def Insert(T,newItem):

if T == None:

T = BST(newItem)

elif T.item > newItem:

T.left = Insert(T.left,newItem)

else:

T.right = Insert(T.right,newItem)

return T

def Delete(T,del\_item):

if T is not None:

if del\_item < T.item:

T.left = Delete(T.left,del\_item)

elif del\_item > T.item:

T.right = Delete(T.right,del\_item)

else: # del\_item == T.item

if T.left is None and T.right is None: # T is a leaf, just remove it

T = None

elif T.left is None: # T has one child, replace it by existing child

T = T.right

elif T.right is None:

T = T.left

else: # T has two chldren. Replace T by its successor, delete successor

m = Smallest(T.right)

T.item = m.item

T.right = Delete(T.right,m.item)

return T

def InOrder(T):

# Prints items in BST in ascending order

if T is not None:

InOrder(T.left)

print(T.item,end = ' ')

InOrder(T.right)

def InOrderD(T,space):

# Prints items and structure of BST

if T is not None:

InOrderD(T.right,space+' ')

print(space,T.item)

InOrderD(T.left,space+' ')

def SumTree(T):

if T is None:

return 0

if T is not None:

Sum = T.item

if T.right is not None:

Sum += SumTree(T.left)

Sum += SumTree(T.right)

return Sum

def SmallestL(T):

# Returns smallest item in BST. Returns None if T is None

if T is None:

return None

while T.left is not None:

T = T.left

return T

def Smallest(T):

# Returns the smallest item in BST. Error if T is None

if T.left is None:

return T

else:

return Smallest(T.left)

def Largest(T):

if T.right is None:

return T

else:

return Largest(T.right)

def Find(T,k):

# Returns the address of k in BST, or None if k is not in the tree

if T is None or T.item == k:

return T

if T.item<k:

return Find(T.right,k)

return Find(T.left,k)

def FindDepth(T,k):

if T is None:

return -1

if T.item == k:

return 0

while T is not None:

depth = 0

if T.item < k:

depth += 1

return depth + FindDepth(T.right,k)

if T.item > k:

depth +=1

return depth + FindDepth(T.left,k)

return depth

def SumAtDepth(T,d):

if T is None:

return 0

if d is 0:

return T.item

while T is not None:

depth = 0

Sum = 0

if depth < d:

depth += 1

return depth + FindDepth(T.right,d)

if depth > d:

depth -=1

return

return Sum

def FindAndPrint(T,k):

f = Find(T,k)

if f is not None:

print(f.item,'found')

else:

print(k,'not found')

#1

def DisplayTree():

fig, ax = plt.subplots()

ax.plot((-5,0,5),(-10,0,-10), color='k')

ax.plot((-7,-5,-2),(-20,-10,-20), color='k')

ax.plot((2,5,7),(-20,-10,-20), color='k')

ax.plot((-8,-7,-6),(-30,-20,-30), color='k')

ax.plot((-4,-2,-1),(-30,-20,-30), color='k')

ax.plot((-4,-3),(-30,-37), color='k')

ax.set\_aspect(.25)

ax.axis('on')

ax.text(0.5, -1, '10', fontsize=15)

ax.text(-4.75, -11.25, '4', fontsize=15)

ax.text(5.25, -11, '15', fontsize=15)

ax.text(-6.75, -21, '2', fontsize=15)

ax.text(-1.75, -21, '8', fontsize=15)

ax.text(2.25, -21, '12', fontsize=15)

ax.text(7.25, -21, '18', fontsize=15)

ax.text(-7.75, -31, '1', fontsize=15)

ax.text(-5.75, -31, '3', fontsize=15)

ax.text(-3.75, -31, '5', fontsize=15)

ax.text(-.75, -31, '9', fontsize=15)

ax.text(-2.75, -38, '7', fontsize=15)

##2 A find implementaion with recursive method.

def FindIter(T,k):

global count

count =0

if T is None:

return None

while T is not None:

if k == T.item: #checks if the first item is the item

count += 1

return T

if k > T.item: # if not checks if item is bigger or smaller to send to the right direction

count += 1

T = T.right

if k < T.item:

count += 1

T = T.left

return None

#Finds and prints if found or not.

def FindAndPrintIter(T,k):

f = FindIter(T,k)

if f is not None:

print(f.item,'found') #calls the above method if true returns found if not returns not found

else:

print(k,'not found')

##3 Takes a sorted list and then meks it into a balanced tree

def BalanceTree(List):

global count

count = 0

if List is None or len(List) == 0: # checks if list is empty

return None

if len(List) == 1:

count +=1

T = BST(List[0]) #checks if list is only 1 number

return T

else:

mid = len(List)//2 #gets the middle number to make it the root

T = BST(List[mid])

T.left = BalanceTree(List[0:mid]) # sets all that are smaller to left

T.right = BalanceTree(List[mid+1:]) # sets all to right if bigger than root

count +=1

return T

##4

def ExtractToList(T, L):

global count

count = 0

if T is not None:

ExtractToList(T.left,L) #grabs a list and makes it into another copy

List.append(T.item)

ExtractToList(T.right,L)

count +=1

##5

def ElementsAtDepth(T,k):

global count

count = 0

if T is None:

return None # checks if t is none

if k == 0:

count +=1

print(T.item, end = ' ') #once k is 0 return the item

else:

ElementsAtDepth(T.left, k-1) #checks all of left

ElementsAtDepth(T.right, k-1) #checks all of right

count +=1

#To test the other codes

#T = None

#

#A = [10, 4, 15, 2, 8, 12, 18, 1, 3, 5, 9, 7]

#for a in A:

# T = Insert(T,a)

#

#S = None

#

#InOrderD(T,'')

#

#Find(T, 10)

#

#print()

#

#print(SmallestL(T).item)

#print(Smallest(T).item)

#

#FindAndPrint(T,10)

#FindAndPrint(T,110)

#

#n=1

#print('Delete',n,'Case 1, deleted node is a leaf')

#T = Delete(T,n) #Case 1, deleted node is a leaf

#InOrderD(T,'')

#print('####################################')

#

#n=5

#print('Delete',n,'Case 2, deleted node has one child')

#T = Delete(T,n) #Case 2, deleted node has one child

#InOrderD(T,'')

#print('####################################')

#

#n=15

#print('Delete',n,'Case 3, deleted node has two children')

#T = Delete(T,n) #Case 3, deleted node has two children

#InOrderD(T,'')

#

#n=2

#print('Delete',n,'Case 3, deleted node has two children')

#T = Delete(T,n) #Case 3, deleted node has two children

#InOrderD(T,'')

#DisplayTree()

#

print()

print("Display the binary search tree as a figure.")

DisplayTree()

print()

print()

print("This Tree will be used for the remaining functions:")

print()

T = None

A = [70, 50, 90, 130, 150, 40, 10, 30, 100, 180, 45, 60, 140, 42]

for a in A:

T = Insert(T,a)

InOrderD(T, ' ')

print()

print("Make an Iterative version of the Search Operation: ")

k = 70

print(" Looking for", k, "in Tree. ", end= " ")

FindAndPrintIter(T,k)

print('Count: ', count)

k = 17

print(" Looking for", k, " in Tree. ", end= " ")

FindAndPrintIter(T,k)

print('Count: ', count)

k = 10

print(" Looking for", k, "in Tree. ", end= " ")

FindAndPrintIter(T,k)

print('Count: ', count)

print()

print()

print(" Method that creates a balanced tree given a sorted list")

List = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]

print(" List:", List)

print()

print(" List in a Balanced Tree:")

SA = BalanceTree(List)

InOrderD(SA, " ")

print('Count: ', count)

print()

print()

print(" Extract elements in a tree into a sorted list")

List = []

ExtractToList(T,List)

print(" Extracted elements: ", List)

print('Count: ', count)

print()

print()

print("Print elements in order by depth")

k=0

print("Elements at depth",k,":",end= " ")

ElementsAtDepth(T,k)

print()

print('Count: ', count)

print()

k=1

print("Elements at depth",k,":",end= " ")

ElementsAtDepth(T,k)

print()

print('Count: ', count)

print()

k=2

print("Elements at depth",k,":",end= " ")

ElementsAtDepth(T,k)

print()

print('Count: ', count)

print()

k=3

print("Elements at depth",k,":",end= " ")

ElementsAtDepth(T,k)

print()

print('Count: ', count)

print()

k=4

print("Elements at depth",k,":",end= " ")

ElementsAtDepth(T,k)

print()

print('Count: ', count)

print()

ElementsAtDepth(T,k)

print()

“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.”

