**Introduction**

The lab that I was supposed to do is to implement 5 things that are specifies below. The major concept of this lab is to grasp data structure of binary search tree as well as balanced binary search tree. By understanding this lab, it will help to implement b-tree and get deep understand of recursive method.

**Part-1: draw a figure**

The figure is similar to lab1 part 3 since it is drawing a tree. The only things that is left is to have if statements which will stop when the node is the leaf and mark a number inside of a circle. The first step is to draw the tree with needed edges. After I accomplished that, the only thing that I am left with is to draw a circle which does not include lines inside of it. At first, I used plt.circle method to draw a circle, however, I could not figure out how to make is solid and remove the lines. Thus, I decided to use plt.plot with white circle and changed the size and set edgecolor to black.

**Part2: Search an item (iterative)**

This method is very similar to search an item recursively, instead of recursive call, what should be done is to assign new node to temporary variable based on “greater, less than, or equal” in order to traverse the tree in an appropriate way. Whenever the number is found, it will return the node, or None if not found (while loop breaks).

**Part3: Build a balanced binary tree with given sorted array**

Since this method has to be balanced binary tree, we cannot use the insert method, unless the tree will be linear. In order to avoid this from happening, the root has to be the middle point of an array. There will be two recursive calls, one each for left and right child. For left child, the array is from index 0 to middle-1, for right child is from middle-1 to the end to have balanced tree. After the length of the array reaches 0, we know that the array is converted into tree, thus, we can return the T (tree).

**Part4: Extract a number in sorted list**

This method is to get the number in infix order which is left, node, right: (2+3). The idea of extract a tree with infix order is to start traverse the left child and get the root, and then traverse the right child. They are several conditions that we need to consider when extracting an array in infix order. The first is whenever the left child is None, which is when the item should be returned because it is the smallest number. The second consideration is when right child is not None which means there is a need to traverse both left and right child since “if T.left is not None” comes first than the second consideration. When calling both methods, it needs to add the item at T node if not, it will do (left+right) child and that is not infix order. The third case is when left child is not None, but right child is None. Thus, we only traverse for left child and add T.item at the end. When the T is None it will not return anything since it is an array, it will ignore any [].

**Part5: Display elements in a binary tree ordered by depth**

When calling this method, I am passing three variable which is the reference to the root of a tree, current depth (k) that will be display, and the height of a tree (h). I separated this part into two different methods to make it simpler. One method is to add 1 to k in order to display each number. Base case is when k reaches h which means current depth has passed the maximum depth. Before doing the recursive call to this method, I am calling to other method which receives T and k. Inside of this method, I am doing two recursive calls which subtracts the 1 from k. Whenever k is 0, the item will be displayed on the screen. The recursive call is done first for left child, therefore, it will be display in ascending order.

**Output:**

10

30

40

50

70

90

100

130

150

There is 130 in the tree.

Extract tree into an array: [1, 2, 3, 4, 5, 7, 8, 9, 10, 12, 15, 18]

Keys at depth 0 : 8

Keys at depth 1 : 4 12

Keys at depth 2 : 2 7 10 18

Keys at depth 3 : 1 3 5 9 15

A close up of a necklace

Description automatically generated

**Big – O notation for methods**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Part1 | Part2 | Part3 | Part4 | Part5 |
| O() | O(n)` | O(logn) | O(n) | O(N) | O(n) |

"""""""""""""""""""""

Lab 3 - Binary Search Trees

03/08/2019 - Ken M. Amamori

CS2302 MW 10:30 - 11:50

TA: Anindita Nath, Maliheh Zargaran

"""""""""""""""""""""

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

""""""""""""""""""

#find the height of a tree

def find\_height(T):

if T is not None: #base case

return (1+max([(find\_height(T.left)), find\_height(T.right)])) #1 + (the higher number)

else:

return -1

#print elements at depth k

def printAtdepth(T, k):

if T is None: # base case

return

if k==0: #reached the depth

print(T.item, " ", end='')

else:

printAtdepth(T.left, k-1) #recursive call for left child

printAtdepth(T.right, k-1) #recursive call for right child

#print each depth and stops when counter reaches the height of the tree

def print\_depthorder(T, k, h):

if h != k: #base case

print("Keys at depth",k,": ", end='')

printAtdepth(T, k) #call a method which prints items

print()

print\_depthorder(T, k+1, h) #recursive call, add one to k

#Extract elements in a tree into sorted array

def extract\_tree(T):

if T is not None: #base case

if T.left is None: #reached the last element

return [T.item] #return the current number

if T.right is not None: #both left and right child exist

return extract\_tree(T.left) + [T.item] + extract\_tree(T.right) #recursive call: left,node,right (infix)

else: #no right child

return extract\_tree(T.left) + [T.item]

#build a balanced binary search tree with given sorted list A

def balanced\_tree(T, A):

l = len(A) #get the length of the tree

if l>0: #base case

l = l//2 #get the middle number

T = BST(A[l]) #insert number

T.left = balanced\_tree(T.left, A[:l]) #recursive call for left child with new array

T.right = balanced\_tree(T.right, A[l+1:]) #recursive call for right child with new array

return T

#print item if T is not None, "no such item" when None node

def print\_node(T):

if T is not None:

return T.item

else:

return "no such item"

#search a number k in the tree T, return node with the same number if it was found, none if not found

def search\_num(T, k):

temp = T #temporary variable for T

while temp is not None: #iterate through necessary nodes

if temp.item > k: #the number is smaller

temp = temp.left

elif temp.item < k: #the number is larger

temp = temp.right

else: #current item == k

return temp #return node with the same number as k

return None #not found

#draw a tree with numbers

def draw\_tree(ax, c, T, dx, dy):

if T.left is not None: #checks if left child exists

ax.plot([c[0], c[0]-dx], [c[1], c[1]-dy], color='k') #draw a line towards left child

draw\_tree(ax, [c[0]-dx, c[1]-dy], T.left, dx\*0.5, dy\*0.9) #recursive call with new root

if T.right is not None: #checks if right child exists

ax.plot([c[0], c[0]+dx], [c[1], c[1]-dy], color='k') #draw a line towards right child

draw\_tree(ax, [c[0]+dx, c[1]-dy], T.right, dx\*0.5, dy\*0.9) #recursive call with new root

plt.plot(c[0], c[1], 'wo', markersize=23, markeredgecolor='black') #draw a white circle at 'c' with size of 23 and black edge color

ax.annotate(T.item, [c[0]-2.5, c[1]-2]) #plot a number inside of a circle

#insert a number into binary search tree

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

#print a tree in inorder

def InOrder(T, space):

if T is not None:

InOrder(T.left, space+' ')

print(space, T.item)

InOrder(T.right, space+' ')

T = None

A = [70, 50, 90, 130, 150, 40, 10, 30, 100]

for a in A:

T = Insert(T,a)

InOrder(T, ' ')

"""Part-2"""

print("There is", print\_node(search\_num(T, 130)), "in the tree.")

"""Part-3"""

T1 = None

A1 = [1, 2, 3, 4, 5, 7, 8, 9, 10, 12, 15, 18]

T1 = balanced\_tree(T1, A1)

"""Part-4"""

print("Extract tree into an array:", extract\_tree(T1))

"""Part-5"""

print\_depthorder(T1, 0, find\_height(T1)+1)

"""Part-1"""

plt.close("All")

fig, ax = plt.subplots()

draw\_tree(ax, [0,0], T1, 50, 50) #ax, center, tree, dx, dy

ax.set\_aspect(1.0)

ax.axis('off')

plt.show()

Conclusion:

Throughout this lab, I was able to grasp and get the deep understanding of a binary tree. In addition, I learned the effectiveness of using recursive method. Since trees has a pattern, and it requires to manage for left and right child, implementing recursively is a lot simpler. Most of the methods I wrote is recursively, and I don’t see how it will look if I am supposed to use iteration way. Also, I understand how to apply lab1 part3 into this lab to have a better user interface than just printing numbers in a not user-friendly way.

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.