Yahir Rivas

Instructor: Dr. Olac Fuentes

CS 2302

**Lab 3 Report**

For this lab I was asked to submit 5 different methods related to binary search trees. I had to draw a binary search tree using the mathplot library, similar to the one done in the first lab. Then I had to implement 4 methods for binary search trees, including binary search, building a balanced tree with a sorted list as the input, extracting binary search tree elements into a sorted list, and printing elements in a binary tree ordered by their depth.

**Search**

For the search method I was asked to do it iteratively. So, I started with setting up the ‘base cases’. I made the case for the element being looked for was less than the current item, if so the list goes to the left, if not then it would go to the right until it either finds the element that we’re looking for or there’s no such element.

**Test:**

*T = None*

*A = [1,2,3,4,5,6,7,8,9,10,12,15,18]*

*for a in A:*

*T = Insert(T,a)*

*print("Searching for 8...Found:",Search(T,8))*

*print("Searching for 18...Found:",Search(T,18))*

*print("Searching for 25...Found:",Search(T,25))*

**Output:**

Searching for 8...Found: 8

Searching for 18...Found: 18

Searching for 25...Found: None

**Balanced Tree**

For the balanced tree method I knew the first thing I had to do was get the middle element of the sorted list and then create a left and a right list from that middle element. So I got the middle element and made that the root of the new tree. Then I created the left and right branches of the tree with the elements left and right of the root respectively.

**Test:**

*T = None*

*A = [1,2,3,4,5,6,7,8,9,10,12,15,18]*

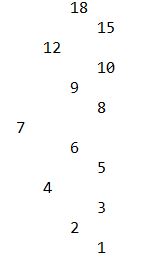
*for a in A:*

*T = Insert(T,a)*

*T = balancedTree(A)*

*space = ' '*

*InOrderD(T,space)*

**Output:**

**Sorted List**

In the sorted list method I established my base case to be if the tree is not empty. If it was not empty then I needed to go to the left branch for it to be sorted, so I made another case where it would check if the left branch was empty I would return the current element. Then I made another case for when the right branch was not empty and after the last case it would mean the left branch was not empty as well. If that was the case then I would add the current element while recursively calling the method with the left and right branch. My last case was checking if the right branch was empty while the left one wasn’t then I would add the current element while recursively calling the left branch, this ensured I had all cases for the tree covered and I could effectively sort the list.

**Test:**

*A = [1,2,3,4,5,6,7,8,9,10,12,15,18]*

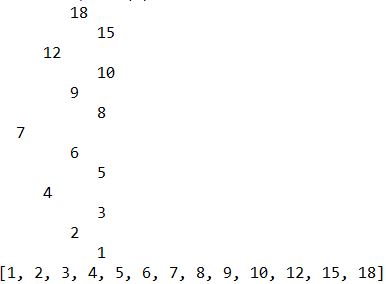
*T = balancedTree(A)*

*InOrderD(T, ' ')*

*S = SortedList(T)*

*print(S)*

**Output:**

****

**Print at Depth**

For the method Print at depth I made the same base case as most of the other methods, if the tree is empty return None. If not I checked if the depth was equal to zero, meaning I reached the desired depth, if I wasn’t on the desired depth then I called the recursive method with the left and right branch of the tree and subtracted 1 to the depth so I could reach my base case. Then I called this method with a for loop so I could print all the depth of the tree using i as the depth parameter.

**Test:**

*T = None*

*A = [1,2,3,4,5,6,7,8,9,10,12,15,18]*

*for a in A:*

*T = Insert(T,a)*

*for i in range(FindHeight(T)):*

*print("Elements at depth" , i , ": " , end = "", flush = True)*

*PrintAtDepth(T,i)*

**Output:**

Elements at depth 0 : 7

Elements at depth 1 : 4

12

Elements at depth 2 : 2

6

9

18

Elements at depth 3 : 1

3

5

8

10

15

**Draw Binary Tree**

I was unable to finish the draw binary tree method. Since I knew this was going to be the most difficult method out of the 5 that we had to do for this lab, I decided to do it until the end. First, I tried using the same method binary tree method used for the first lab, but I had some issues. I tried to call the method with the n parameter being the height of the tree but I had some errors that I was unable to figure out, so I decided to submit my lab incomplete before I had too many point taken off for late submission.

**Index**

import matplotlib.pyplot as plt

import numpy as np

import math

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 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 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 FindAndPrint(T,k):

f = Find(T,k)

if f is not None:

print(f.item,'found')

else:

print(k,'not found')

def SumTree(T):

if T is not None:

return SumTree(T.left.item) + SumTree(T.right.item)

return None

def FindDepth(T,k):

if T is None:

return -1

if T.item > k:

return 1 + FindDepth(T.left)

if T.item < k:

return 1 + FindDepth(T.right)

def FindHeight(T):

if T is None:

return 0

else:

leftDepth = FindHeight(T.left)

rightDepth = FindHeight(T.right)

if(leftDepth > rightDepth):

return 1 + FindHeight(T.left)

else:

return 1 + FindHeight(T.right)

'''

def DrawTree(ax,p,n,x,y):

if n>0:

#changes points of x and y

ax.plot([p[0],p[0]-x],[p[1],p[1]-y], color='k') #Left branches are graphed

ax.plot([p[0],p[0]+x],[p[1],p[1]-y], color='k') #Right branches are graphed

DrawTree(ax,[p[0]-x,p[1]-y],n-1,x\*.5,y\*.9) #Left side recursive call

DrawTree(ax,[p[0]+x,p[1]-y],n-1,x\*.5,y\*.9) #Right side recursive call

fig, ax = plt.subplots()

T = None

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

A = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140]

for a in A:

T = Insert(T,a)

n = FindHeight(T) - 1

DrawTree(ax, [0,0], FindHeight(T), 75, 75)

ax.set\_aspect(1.0)

ax.axis('off')

plt.show()

fig.savefig('binary.png')

'''

def Search(T,k):

temp = T

while temp is not None:

if temp.item > k: #if k is smaller than current element

temp = temp.left

elif temp.item < k: #if k is larger than current element

temp = temp.right

else: #if current element is equal to k

return temp.item

return None

def balancedTree(A):

if not A:

return None

mid = len(A)//2 #gets middle index of sorted list

bT = BST(A[mid]) #creates a new tree with the middle element as the root

bT.left = balancedTree(A[:mid]) #creates left branch with numbers less than root

bT.right = balancedTree(A[mid+1:]) #creates right branch with numbers greater than root

return bT

def PrintAtDepth(T,d):

if T is None: #if tree is empty

return None

if d == 0: #if depth was reached

print(T.item)

else:

PrintAtDepth(T.left, d-1) #recursive call for left tree

PrintAtDepth(T.right,d-1) #recursive call for right tree

def SortedList(T):

if T is not None:

if T.left is None:

return [T.item] #returns current element

if T.right is not None:

return SortedList(T.left) + [T.item] + SortedList(T.right) #calls recursion with left node, adds current element and calls right node

else:

return SortedList(T.left) + [T.item] #calls recursion with left node and adds current element

T = None

A = [1,2,3,4,5,6,7,8,9,10,12,15,18]

for a in A:

T = Insert(T,a)

'''Problem 2'''

print("Searching for 8...Found:",Search(T,8))

print("Searching for 18...Found:",Search(T,18))

print("Searching for 25...Found:",Search(T,25))

'''Problem 3'''

T = balancedTree(A)

space = ' '

InOrderD(T,space)

'''Problem 4'''

T = balancedTree(A)

InOrderD(T, ' ')

S = SortedList(T)

print(S)

'''#Problem 5'''

for i in range(FindHeight(T)):

print("Elements at depth" , i , ": " , end = "", flush = True)

PrintAtDepth(T,i)

**Conclusion**

Even though this lab was not as difficult as I expected it to be, I was unable to finish it. I felt like this lab was a very good practice on recursion. This lab helped me understand that I need to make more time to do my assignments so I’m not as pressured when the assignment is due. I also learned that recursion is as easy as you make it, and if you have a good understanding on it you can program very efficiently.