Lab 3 Report – CS 2302

This 3rd lab consisted of 5 different problems regarding Binary Search Trees (BST). The first one was a combination of a problem form lab 1 (draw trees recursively) and BSTs. The second problem is just an iterative version of a recursive method already provided (search for an element in a BST and return a reference to the node storing the key). The third problem was to build a BST given a sorted list, and the fourth the opposite (extract a sorted list form a BST). Lastly, problem five asked to print the keys of a BST by depth (print the levels of the tree).

The first problem was solved by modifying the code for the draw-trees problem from lab 1. The figure requested had the extra elements of circles and printed text on the python plot plane. The process of drawing a circle was already given previously, and the text was printed using ‘ax.text()’ with the same center of the circle, so they represent a node with an element stored. The steps of the method consisted of having a method to draw a line between 2 given points (used to draw a line between current node and child node) and a circle with text inside as long as the node is not none, all while traversing the tree in pre-order traversal. A line to the child node is only drawn when a child node exists, and then 2 recursive calls (one for each child) are made. The text has a fixed font size, which will not be ideal for trees of a different size. The method can draw trees of any size, but the scale will not display the tree properly if the size is too big. The calculations of the coordinates of the next node are basically the same as that of the centers of the original method.

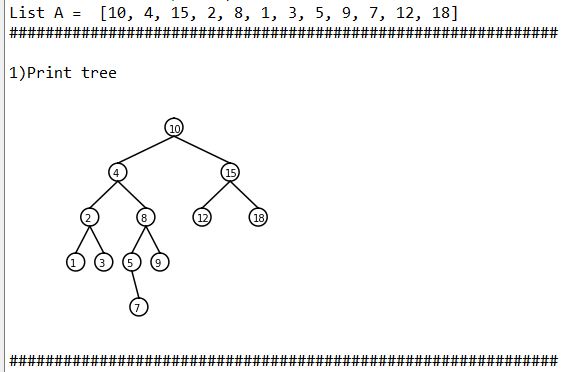
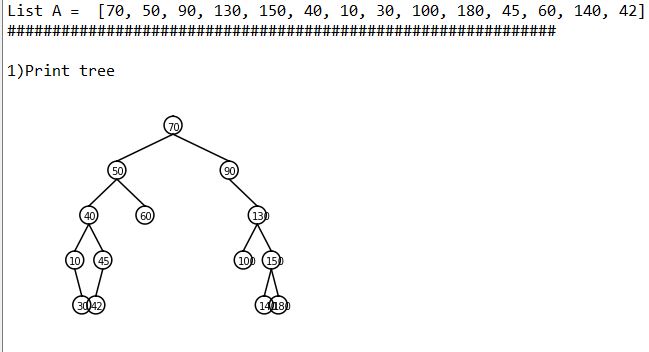
Iterative search was done by using a temp variable (in order to not delete the root of the tree while traversing it) and moving through the tree with a while loop. If the current item was the number k, the method returns a reference to the current node, but if the current item was greater than k, the current node is moved to the left child (otherwise the right child), and if the current node was none (which means that the item is not in the tree) None was returned.

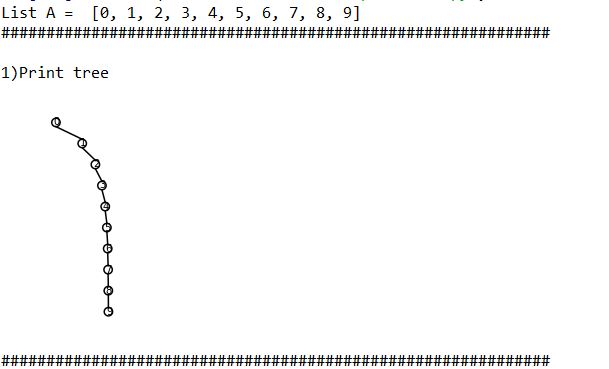
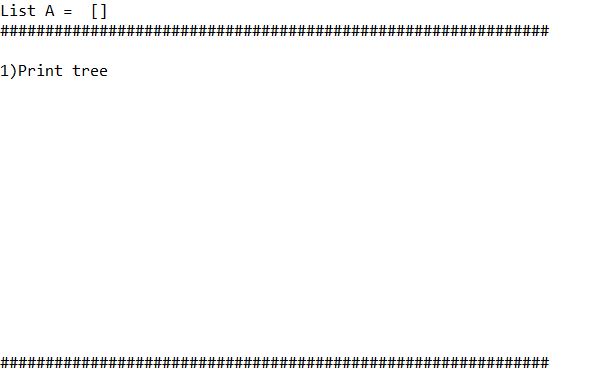
Building a balanced BST was relatively easy because a sorted list was provided as input. Knowing this, the root of the whole tree will always be the middle item of the list, and the children the left and right halves of the list. The recursive solution has a base case that returns none if the list is empty; if the length of the list is exactly one, then make a node with the item; else make the middle of the current list (which contains at least 2 elements) a root, and their children a recursive call with the first half of the list and the second half (excluding the middle element). Lastly, a reference to the current node was returned.

The extract method was simpler, in the sense that recursion is allowed. The base case should be when the current node is none and there is nothing to extract, so an empty list is returned. When there is something to extract, we construct a list consisting of the left child, the current node, and the right child, and then we return that list.

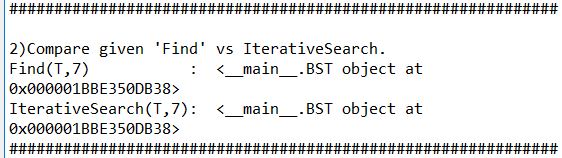
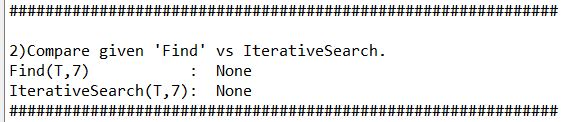
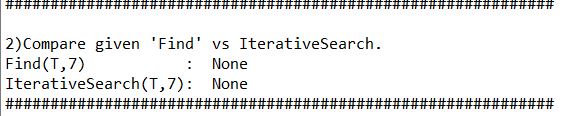
Lastly, all the levels of a tree must be printed. This was achieved by having a method that prints all the nodes in a desired level, and then looping that method as many times as the height of the tree, which was calculated by another method. This said method traverses the whole tree and returns a +1 to each level of the longest branch. The method that prints by level takes a reference to the root of the tree and *d,* the desired level, then it will advance through the tree until *d* is 0 and then print the current items of the node, this is repeated for the left and the right as well.

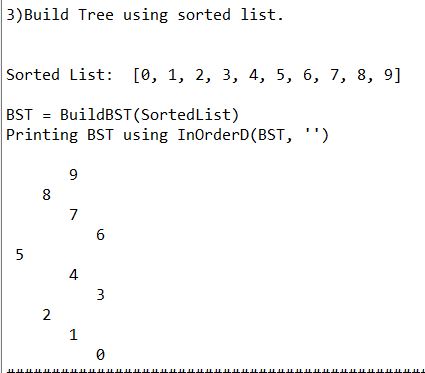
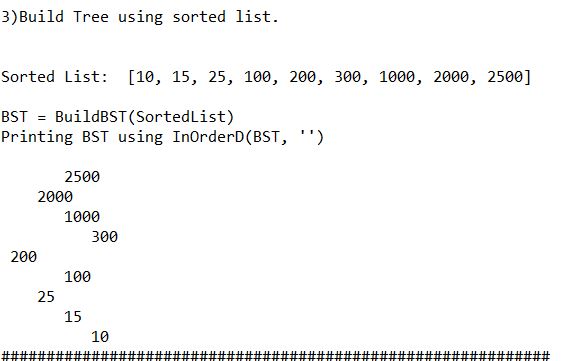
**Experimental Results**

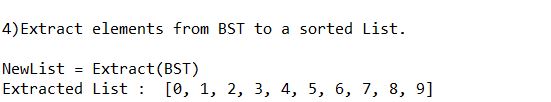
#1 This method was tested with 4 different trees: The prompted tree, a larger tree, a smaller tree, and an empty tree. The results were the following:

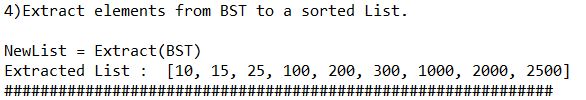


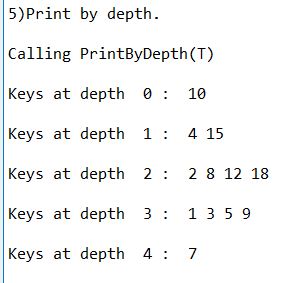
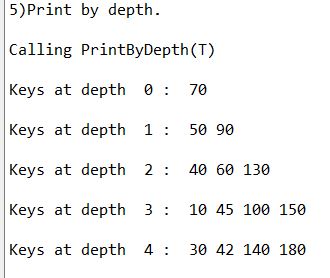
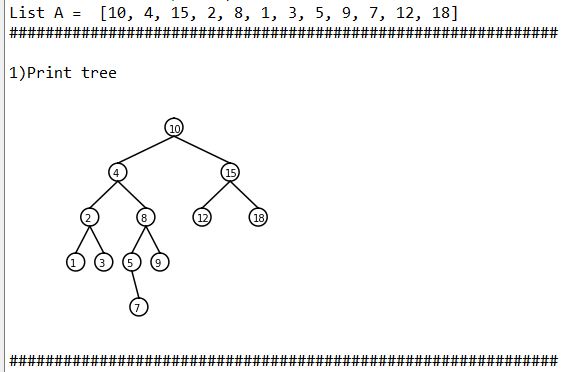
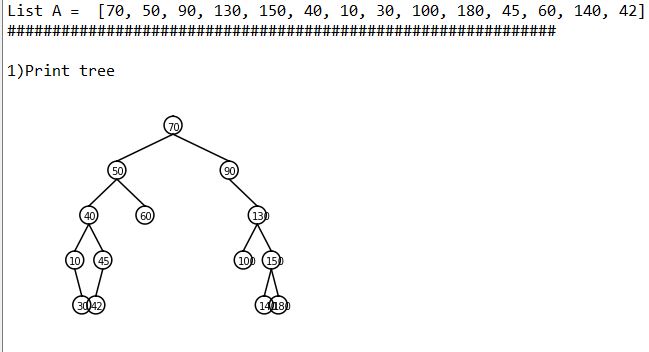
Note how a larger input is not an ideal fit for the scale, but the numbers and locations are correct.

#2 The iterative version of the search method was tested against the output of the provided search method. They were tested by looking for a number in the tree, a list without the number that is being searched for, and an empty tree. They both provided the same outputs.

#3 The method that build a tree given a sorted list was tested with 2 different list, and then was printed using the provided InOrderD method, which prints the structure of the tree. This was done to verify that the tree is being constructed correctly, and that all numbers are in their right place.

#4 The extraction method was tested with the previously constructed trees, so the expected output is the initial lists used in #3.



#5 The BSTs used to test the methods were the first 2 trees printed on #1, so their structure can be easy to compare for correctness.

The running time of the different problems is shown below. Because of the multiple recursion calls that problem 1 does, the running time increases exponentially O(n^2), as the size of the tree increases. Problems 2, 3, and 4 run in constant time, and Problem 5 takes O(n) to compute.

**Appendix**

# -\*- coding: utf-8 -\*-

"""

Created on Tue Mar 5 20:55:43 2019

CS 2302 - Andres Silva

> Teacher: Olac Fuentes

> TAs: Anindita Nath & Maliheh Zargaran

> Lab #3

> The purpose of this lab is to work with binary search trees and their structure. Mainly, the navigation

of them using recursion and iteration.

> LAST MODIFIED: MARCH 8th, 2019

"""

# Code to implement a binary search tree

# Programmed by Olac Fuentes

# Last modified February 27, 2019

import matplotlib.pyplot as plt

import numpy as np

import math

import time

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 PrintStructure(T,space):

if T is not None:

print(space,T.item)

PrintStructure(T.right,space+' ')

PrintStructure(T.left,space[:-2])

def IterativeSearch(T,k):

t = T

while(t is not None): #Move through tree until finding k or reaching the end.

if k == t.item:# If k is found, return the reference to current node.

return t

else:

if (k > t.item): #If k is greater than current node, move right.

t = t.right

else:

t = t.left #Else, move left.

return None #Exiting loop means k was not found, return None as reference.

def lines(ax,center,p): #Plots 1 line between 2 given points (modifes the center for neatness)

if p != None:

ax.plot([center[0] ,p[0]],[center[1] - 100,p[1] + 300] ,color='k')#Center and P1

def circle(center,rad): #Model code stays unmodified, it just traces a circle with a center and a radius.

n = int(4\*rad\*math.pi)

t = np.linspace(0,6.3,n)

x = center[0]+rad\*np.sin(t)

y = center[1]+rad\*np.cos(t)

return x,y

def draw\_trees(ax,T,center,segment\_height,segment\_width):

p1 =[center[0] - segment\_width/4 ,center[1] - segment\_height] #The two children points are calculated and stored in point 1 (p1) and point 2 (p2).

p2 =[center[0] + segment\_width/4 ,center[1] - segment\_height]

if T != None:#Only draw when node exist.

x,y = circle([center[0], center[1] + 100],200) #Root Circle

ax.plot(x,y,color='k') #Plot the circle.

ax.text(center[0] - 120, center[1] , T.item, fontsize=10) #Print item of node

if T.left != None: #Draw line to left child only if left child exists.

lines(ax,center,p1)

if T.right != None: #Draw line to right child only if left child exists.

lines(ax,center,p2)

draw\_trees(ax,T.left,p1,segment\_height,segment\_width/2) #Repeat using left node as root.

draw\_trees(ax,T.right,p2,segment\_height,segment\_width/2)

def Extract(T):

if T is None: #return empty list when T is none.

return []

# elif T.left == None and T.right == None: This is reduntant, but helps to understand.

# L = [T.item]

# return L

else:

List = Extract(T.left) + [T.item] + Extract(T.right) #Concatenate the left child, the current node, and the right child.

return List #return glued list.

def BuildBST(L):

if len(L) < 1: #If L is empty

return None

if len(L) == 1: #If L has 1 element, make a node with that element and return it.

T = BST(L[0]) #Make a node with it, and return it.

return T

else:

T = BST(L[len(L)//2]) #Make the root the middle element

T.left = BuildBST(L[ :(len(L)//2)]) #Make the left child with the list from 0 to middle (exclusive)

T.right = BuildBST(L[(len(L)//2) + 1 : ]) #Make the right from middle + 1 (exlcuding root) to end of list.

return T #Return 'glued tree'.

def PrintLevel(T,d):

if T == None:

return

if d == 0: #If at root level, print root.

print(T.item, end=' ')

elif d > 0: #Keep going and decrease d down to desired level to print

PrintLevel(T.left, d-1)

PrintLevel(T.right, d-1)

def GetHeight(T):

if T == None: #At a none node, add 0 to our sum.

return 0

else:

left = GetHeight(T.left) #Get height of left and right subtree.

right = GetHeight(T.right)

if right > left: #add 1 to the largest subtree and return int.

return 1 + right

else:

return 1 + left

def PrintByDepth(T):

HeightOfTree = GetHeight(T)#Get Height of tree

for i in range(HeightOfTree): #Print all levels of the tree.

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

PrintLevel(T,i)

print("\n")

# Code to test the functions above

T = None

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

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

#A = [0,1,2,3,4,5,6,7,8,9]

#A = []

#Build Tree with list A.

for a in A:

T = Insert(T,a)

#################################################################

#################################################################

### METHOD TESTING ##############################################

print("List A = ", A)

print("#############################################################")

print("\n1)Print tree")

plt.close("all")

fig, ax = plt.subplots()#Call method below this line.

start = time.time()

draw\_trees(ax,T,[0,0],1000,5000)

end = time.time()

print(end - start)

ax.set\_aspect(1.0) #And before this one .

ax.axis('off')

plt.show()

print("#############################################################")

print("\n2)Compare given 'Find' vs IterativeSearch.")

print("Find(T,7) : ",Find(T,7))

start = time.time()

print("IterativeSearch(T,7): ", IterativeSearch(T,7))

end = time.time()

print(end - start)

print("#############################################################")

print("\n3)Build Tree using sorted list.")

print('\n')

SortedList = A

print("Sorted List: ",SortedList)

print("\nBST = BuildBST(SortedList)")

start = time.time()

BST1 = BuildBST(SortedList)

end = time.time()

print(end - start)

print("Printing BST using InOrderD(BST, '')\n")

InOrderD(BST1, '')

print("#############################################################")

print("\n4)Extract elements from BST to a sorted List.")

print("\nNewList = Extract(BST)")

start = time.time()

NewList = Extract(BST1)

end = time.time()

print(end - start)

print("Extracted List : ", NewList)

print("#############################################################")

print("\n5)Print by depth.")

print("\nCalling PrintByDepth(T)\n") #First list created

start = time.time()

PrintByDepth(T)

end = time.time()

print(end - start)

**Academic Honesty Statement**

“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.”

- Andres Silva.