Lab 3 had us create a Binary Search Tree, BST, and display an image of the newly formed BST, create an iterative version of search. On top of that we also had to build a balanced tree given a sorted list not using the insert operation in O(n) time and be able to create a sorted list from a BST in O(n) time. The final task was to print keys at every depth of the tree, so all elements at depth 0, 1, and so on until the entire tree is printed.

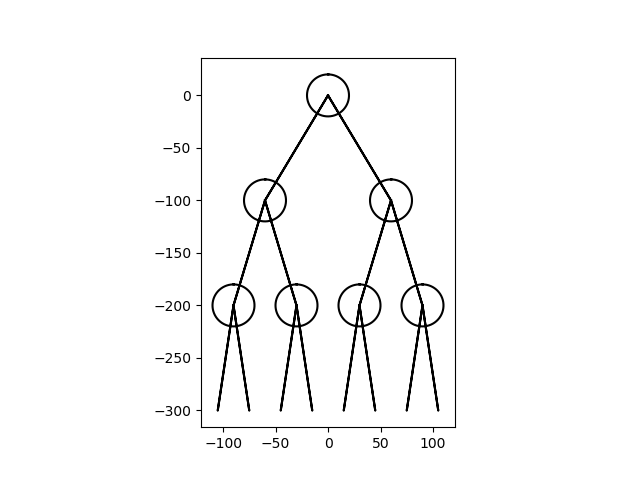
The first task was simple but tedious, I reused several functions from Lab1, namely circles draw circles and drawtree functions. These functions were modified from before such as removing n from the calls as I would only draw based on the nodes and not n number of times, and one addition would be adding T to the arguments which would serve as the T we would be drawing. Changing to base case from N>0 was another change to if T is None as we would not draw anything if there is no tree to draw. From there I would create np arrays that would save the coordinates of the branch vertices and would draw them using ax.plot. I would then draw a circle to represent the node and print out the current items value using the ax.text function. Then It was similar to before creating arrays based off of the current values and passing them to recursive calls if they had nodes.

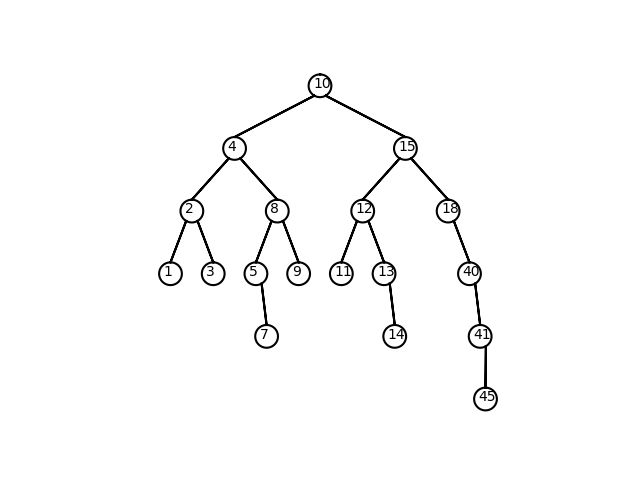
The second task to handle iterative search was extremely simple, I know I couldn’t use recursion so instead I used a while loop to go through the tree which would check for k in t.item and go to the left or right subtree depending on where k would be. If the end of the tree was reached then the function would simply return none and print that k, the item searched for, was not found in the tree.

The next was to build a balanced tree given a sorted array as input. This was a tad tricky but not too bad either. I began with 3 base cases. If the array was empty, length 1, or length 2. If it was empty then I would return None, If length of 1 then T.item was the only item in the array, If length was 2 then the first in the array became the parent node and the next node would be a left or right node depending on if it was larger or smaller than the first number. The res of the logic follows creating a tree by making the middle value the current node, and type casting the array into a BST object in order to take advantage of BST properties like .left and .right. Then I would recursively call the function to hand the left and right sub tree using all element to the left of middle item and right of middle item. The tricky parts of the code came down to typecasting the array into a BST object in order to have the .left and .right items in the Tree not be none and function properly.

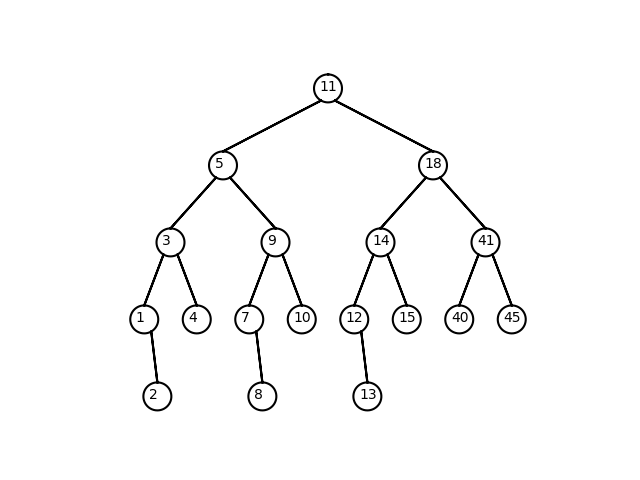
The fourth question was to get the items in a BST and put them into a sorted list which was quite simple as well. You would visit every node going in order of Left node, current node, right node, in order to put the items into an array saving each item in that order as well. If the Tree was None then it would return an empty array, if not then it would call a function to handle the left half of the tree save the current item then handle the right half of the tree. I would then concatenate the returned values of the recursive calls into the array and return the array.

The final part of this lab was to print all items at every depth of the tree. This was a simple task, to do it I used a while loop that ran until a Boolean variable was true; the variable would only become true if the function returned true. My function is similar to the printAtDepthD() function which prints all elements at the depth given. I repurposed that code and put it through a loop that started d at 0 so that it would print every item at every depth within the tree. The function is as said before runs until it has reached depth d and prints the item then returns, now I only added 2 variables to see if either recursive call returns false, if one returns false then the entire function returns false to the Boolean variable in the while loop cause it to loop one more time, if it returns true then the tree has no more items at a greater depth and causes the loop to stop.

While testing the first function I was having a lot of trouble getting the nodes to print correctly. I would often end up with this image .

This image didn’t work at all since the center of the circles was not white. I tried to use function like ax.fill but those wouldn’t work either, perhaps I wasn’t using them correctly or there was something I was missing from it, but after asking for a bit of help I was told to try moving the branches instead. After a lot of testing by honestly putting numbers in till it worked consistently, I found that the node branches had to be moved down and out at the start by an amount somewhat relative to the radius, and the end of them up relative to the radius, since the radius of the circles stays consistent. I then attempted to fix the issue of having the item be printed within the center of each circle correctly as they were off center by a bit initially, but after adding some spaces before each print it stayed fairly centered. In the end I ended up getting the following tree, which now is correctly displaying the tree by having the items displayed be in the center of each circle and being a relatively wide tree.

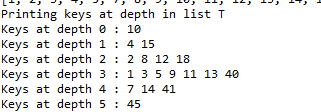
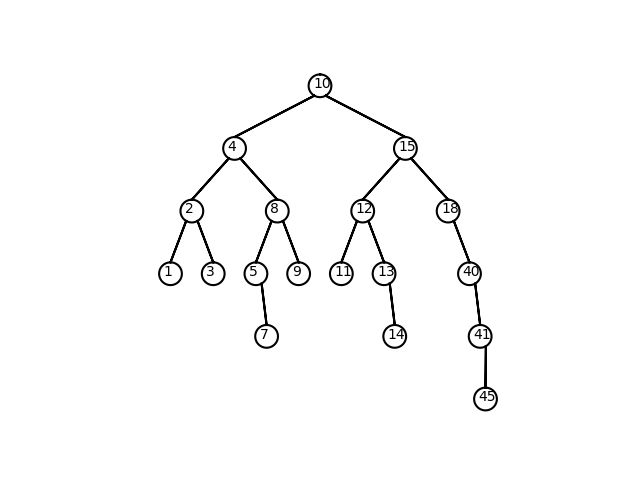
The second function I had less trouble testing, it was just comparing it to the normal search function and seeing if they would output to the same outputs consistently. I would consistently get the output that would say item k is found and item k not found when testing for an item in the tree and an item not in the tree.

 Third function was to build a balanced tree from a sorted list which proved tasking. The first two base cases I came up with in case of A being to small, the third one I came up with due to while testing without it I would get errors that would tell me the item didn’t exist or I would get none pointer errors. To combat the none pointer errors I found I had to type cast the array into a BST object, so that it had the .item, .left, .right properties so that I did not have to use the insert function within the class. The first item was always the middle of the array, so I made a value named mid which was half the length of the array to find the correct index. The item would be stored, and I would recursively call the function to handle each half of the array that’s left. The next issue I would run into was that the .left and .right of the newly made node would not be BST items or have anything inside of them, to avoid this I type casted them to be a BST object. This had to be done in order to avoid having None pointer errors and because we were not allowed to use the insert function. From there T.left and T.right was then equal to the recursive calls for their respective half of the array. Then it was as simple as returning the tree which when put through the function to print the image of the tree to properly see if it was a balanced BST I would get the following image.

Which proves the tree is balanced as the difference in depth is never more than 1. The function described does in in O(n) time because after breaking the code down into a equation I would get 2T(n/2) +1 which when reduced you get O(n) runtime.

The next function turning a tree into a sorted list was also simple in practice. I would just go from the left side of the tree, concatenate it to the array, then add the middle value, then concatenate the right side of the tree. The ability python has to just combine lists by using the addition symbol made this a very trivial exercise only using 2 recursive calls and some extra line I was able to create the list in O(n) time as the time complexity of this function was the same as the previous, 2T(n/2) + 1. To debug this function I would simply print the array returned and check If It was in order myself.

I used the tree from problem one to test this and I would get this array which is in order proving the function I created works properly.

The final function was tested using the same tree, the tree from problem 1, so to ensure that the code was correctly printing items at the correct depth I just compared the figure made there to the output of the code.

When comparing the output of problem 5 to the tree you can see that it does indeed print out the keys at the depth they are at. To achieve this all I did was loop the function using a Boolean variable that kept it going until there was no more nodes left to print in the array.

This lab taught me better navigation skill and a slightly better understanding of how BST’s work and function. This also taught me that trial and error is sometimes a solution approach that can work when trying to properly format visuals like figures and items using ax.plot and printing functions.

Appendix ---

import numpy as np

import matplotlib.pyplot as plt

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 IterativeSearch(T, k):

while T != None:#goes through tree using a loop rather recursion searching for item k

if T.item == k:

print('Item', k, 'found')

return T

if k >T.item:

T = T.right

else:

T = T.left

#only reached if k is not in tree

print('Item', k ,'not found')

return None

def BuildBalancedT(T, A):

if len(A) == 0:

return None

if len(A) == 1:

T.item = A[0]

return T

if len(A) == 2:#This case was made to solve previous issues during testing where tree would not form correctly

T.item = A[0]

if A[1] > T.item:

T.right = BST(A[1])

else:

T.left = BST(A)[1]

return T

mid = len(A)//2

T.item = A[mid]

T.left = BST(A)#makes T.left a BST object

T.right = BST(A)#makes T.right a BST object

T.left = BuildBalancedT(T.left, A[:mid])

T.right = BuildBalancedT(T.right, A[mid+1:])#builds upon tree and returns values to T.left and T.right

return T

def TreeToList(T):

if T == None:

return []

A = []#Creates empty native list

L = [T.item]#turns T.item into a native list object

A = A + TreeToList(T.left)

A = A + L

A = A + TreeToList(T.right)#previous 3 lines concatenates lists

return A

def PrintAtDepth(T, d):

if T is None:

return True

if d == 0:

print(T.item, end = ' ')

return False

Check1= True

Check2= True#Check 1 and 2 are made to make sure that the loop this function is in runs 1 more than the depth of the whole tree to print it all

Check1= PrintAtDepth(T.left, d-1)

Check2= PrintAtDepth(T.right, d-1)

if Check1 is False or Check2 is False:#if either is false then it runs one more time

return False

return Check1

def circle(center,rad):#makes a circle

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\_circles(ax,center,radius):#draws circles

x,y = circle(center,radius)

ax.plot(x,y,color='k')

#cuts each circle in half of current radius

def draw\_tree(ax, v, Dx, Dy, radius, nextCenter, T):

if T is not None:

Lbranch = np.array([[v[0]- radius /2 , v[1] -radius \*.87 ] , [v[0] - Dx ,v[1] - Dy], [v[0]- radius/2, v[1] - radius \* .87 ]])#modifications to v[0] and v[1] were found by testing, the point v[0] needs to be moved out half the radius from the center

Rbranch = np.array([[v[0] + radius / 2, v[1] - radius \*.87], [v[0] + Dx , v[1] - Dy],[v[0] + radius/2, v[1]- radius \* .87]])#this was found by examining the image with the cursor, and v[1] needs to move down radius \*.87 which was found by examining the image again and various adjustments

#^Code above saves current vertex and the end points of the other two branches by taking off how man Dx units moved to the left or right and how many Dy units down

if T.left != None:

ax.plot(Lbranch[:,0],Lbranch[:,1] , color = 'k')#draws left branches if a node exists

if T.right != None:

ax.plot(Rbranch[:,0],Rbranch[:,1] , color = 'k')#draws right branches if a node exists next

draw\_circles(ax,[v[0] , nextCenter[1]],radius)#draws node

nextCenter = [v[0], v[1]- radius - Dy]

s = str(T.item)

ax.text(v[0]- radius \*.6,v[1] - radius \*.2, s , fontdict = None, withdash = False)#prints T.item inside circles, the movement of x y values were found through experimentation of what visually looked most centered

vL = np.array([Lbranch[1, 0], Lbranch[1, 1] - radius])#left vertex's corodinates

vR = np.array([Rbranch[1, 0], Rbranch[1, 1] - radius])#right vertex's corodinates

if T.left != None:

draw\_tree(ax, vL, Dx/2,Dy, radius, nextCenter, T.left)#draws left node with half current Dx value to make smaller branches

if T.right != None:

draw\_tree(ax, vR, Dx/2, Dy, radius, nextCenter, T.right)#draws right node with half current Dx value to make smaller branches

# Code to test the functions above

T = None

A = [10,4,15,2,8,12,18,1,3,5,9,7, 13, 40, 41, 45, 11, 14]

for a in A:

T = Insert(T,a)

#^^ forms tree T

plt.close('all')

v = np.array([0,0])

r = 20

nC = [0,0]

fig, ax = plt.subplots()

draw\_tree(ax, v, 150, 90, r, nC, T)

ax.set\_aspect(1.0)

ax.axis('off')

plt.show()

fig.savefig('BST.png')

# ^^ question 1

IterativeSearch(T, A[4])

IterativeSearch(T, 42)

#^^ question 2

list.sort(A)

test = BuildBalancedT(BST(A), A)

#^^ question 3

v = np.array([0,0])

r = 20

nC = [0,0]

fig, ax2 = plt.subplots()

draw\_tree(ax2, v, 150, 90, r, nC, test)

ax2.set\_aspect(1.0)

ax2.axis('off')

plt.show()

fig.savefig('BST2.png')

#^^ Prints question 3's Tree

B = TreeToList(T)

print('Printing list B made from tree T')

print(B)

#^^ question 4

done = False

i = 0

print('Printing keys at depth in list T')

while done != True:#loops function to print keys at depth i

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

done = PrintAtDepth(T, i)

print()

i +=1

# ^^ question 5

print("Question 1's tree is figure 1, Question 2's tree is figure 2")

“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.”

- Seth Abel Flores