Nicole Favela

Lab 3 report

March 12, 2019

CS2302

**Introduction**

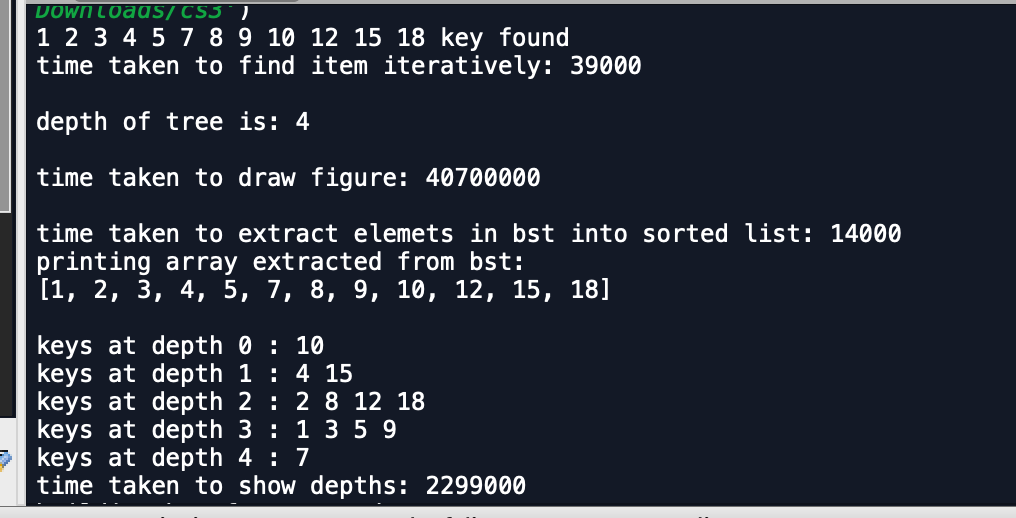
The problem I am trying to solve is to implement a BST (binary search tree) and perform various operations as stated in the lab instructions. Part 1 was to implement a BST from an array and display the figure as a tree with nodes as circles with data in the center. Part 2 was simply the iterative version of the search function we did in class. Part 3 was to build a BST from a sorted list. Part 4 was to extract the elements of a BST and put them into an array in sorted order. Part 5 was to print all the elements in every level or depth of the tree starting at depth 0 (the root).

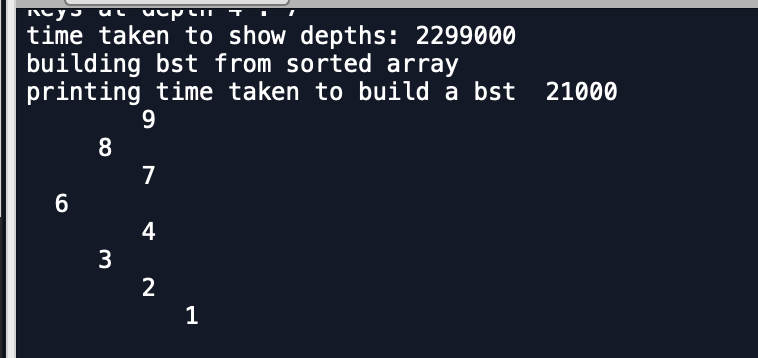
**Approach**

My approach to part 1 (displaying the figure) was to try to use my code from lab 1 and combine the recursive tree and the draw circles function. The most challenging part of that was to make the circles meet the lines perfectly. To get the numbers to appear in the nodes, I used ax.text to have the numbers appear in the middle as strings. From there I made a function called DrawTree which checks 3 conditions. The first is the base case and check if T is not None and draws the basic node with text using the draw\_circles function. The second condition checks if the left subtree is not None. If it is not, I call the draw\_line function I used in lab 1 to create the recursive tree using absolute value function. The other condition checks if T.right is not None. If it is not, it repeats the same functions but on the right side of the tree. My approach to part 2 was to just implement the search function that I already had which search for an element recursively. Instead I used a while loop that continued as long as T is not None. Part 3 involved construction a BST from sorted list. I knew that I could not simply use the insert function since inserting elements already in sorted order would lead to a completely unbalanced tree that resembled a linked list and no longer had the positive attributes of a BST where retrieval is done in O(logn) time. I decided to use the fact that python can split lists quite easily. From there I used recursion to collect the elements that were less than the middle index to build the left side of the tree and did the opposite on the right side. For part 4, extracting elements, I had to perform the opposite operation. I used the property of the BST that states that the farthest left elements contains the smallest item in the tree and the farthest right contains the largest. In extractInOrder, I started from the smallest and recursively stored elements from the left in an array followed by the right side or the tree. Part 5 was similar to another activity we had done in class which got the items at a certain depth. The only challenging part was to print keys at depth i and then call another method which returned the keys at a certain depth. I accomplished this with ShowDepths which printed keys at depth and iterated though i which was the depth of the tree and findAtLevel which went thought all the levels in the tree and returned the corresponding nodes. FindDepths and countDepth were also used to find the absolute depth of a tree by comparing the left and right side of the tree and returning the larger count.

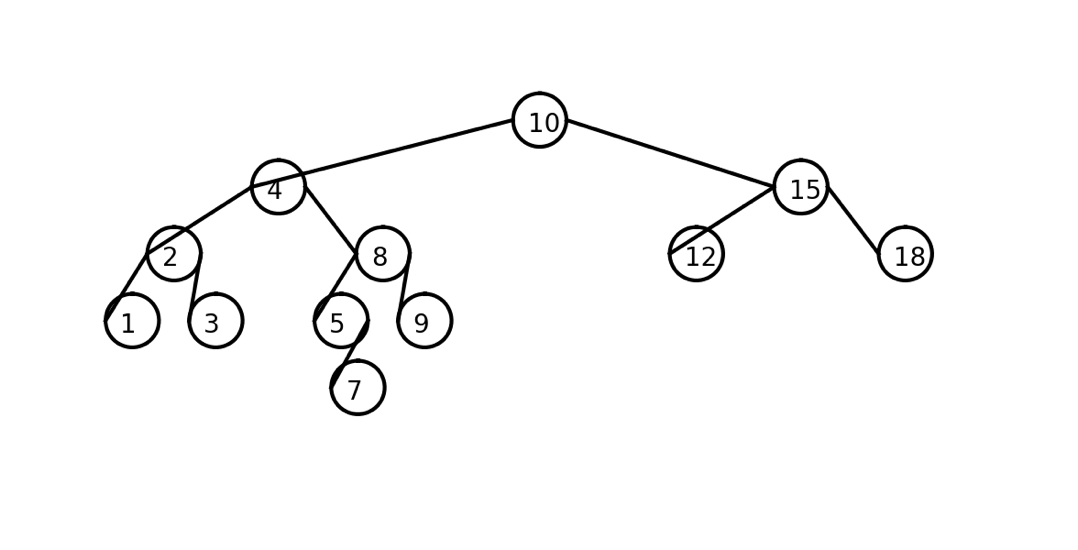
# Sample runs and output:

For my sample runs I created the same figure shown in the lab instructions. To accomplish this, I used an array of the numbers shown in the following order: [10,4,15,2,8,12,18,1,3,5,9,7]. For the sorted array that I used in the buildBST function I used the numbers 1-9 in order. The console output is seen in the following figures.





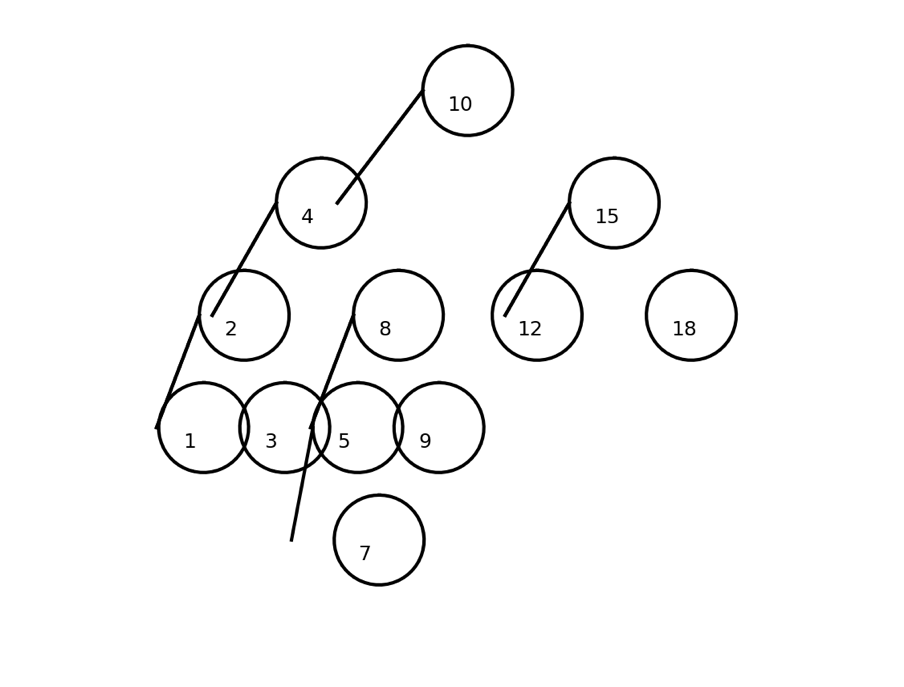
The tree is produced as follows using my DrawTree function.

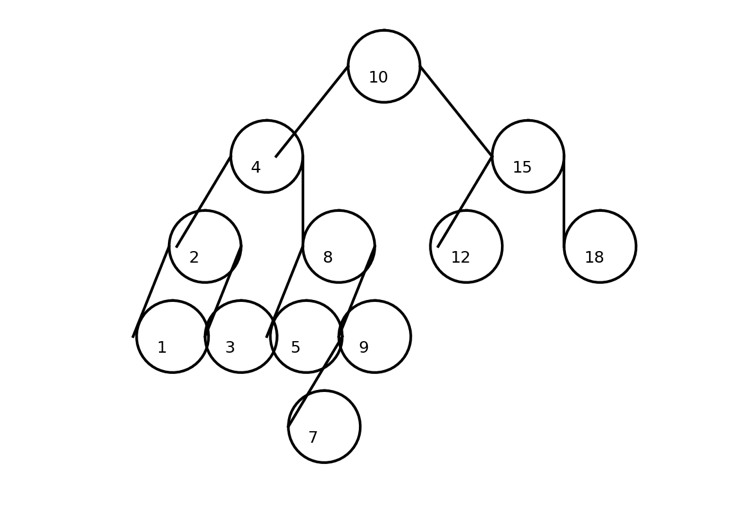


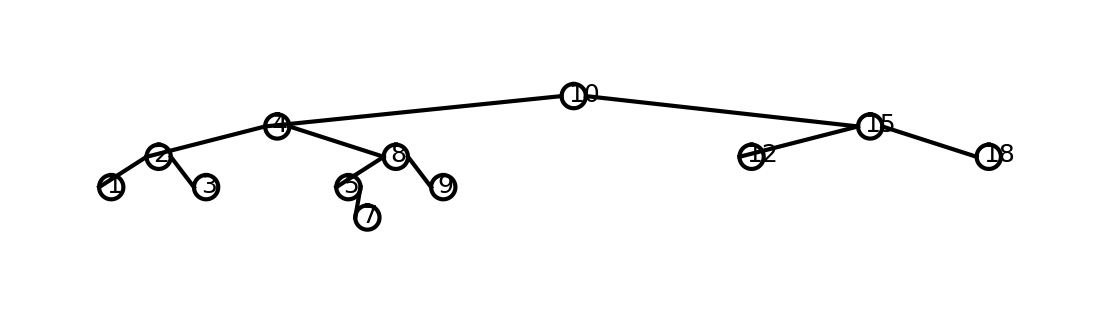
## Runtimes and experimental results:

For my runtime analysis, I used an array of random integers from 1 to 100 using ﻿np.random.randint and varied the sizes. I used an input of n = 10, n=100, and n=1000. I tested all the functions with a loop running 10 times except for the the DrawTree function which I ran only once due to processor limitations. As specified in the lab, parts 3 and 4 were to be executed in O(n) time, this would have been more obvious if I had plotted more n values in between the 3 inputs selected. I verified using the master method to check that these functions were in fact O(n) time. They both fall into the same case where O(n^(log\_2(2)) which equals O(n).

When drawing the figures initially, I got several different results. The first ones started with the lines not connecting and later iterations produces lines that were off center. I ended up using the radius in the second recursive call to increment x and multiplied the depth by powers of 2.5 which yielded the closest result.

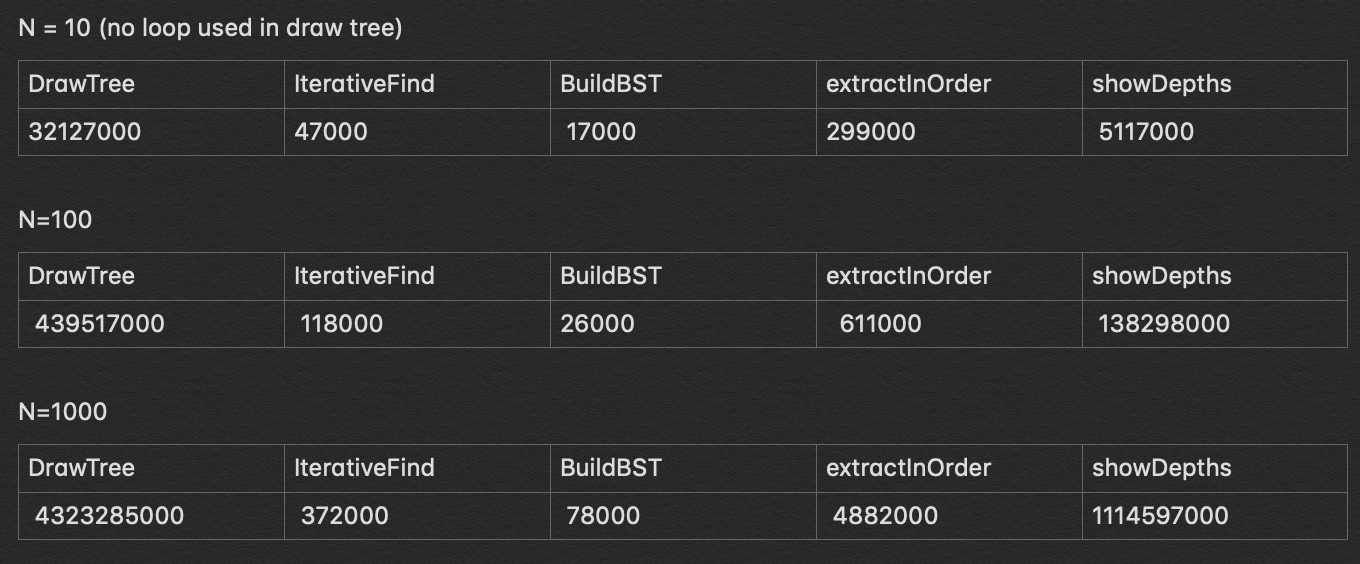


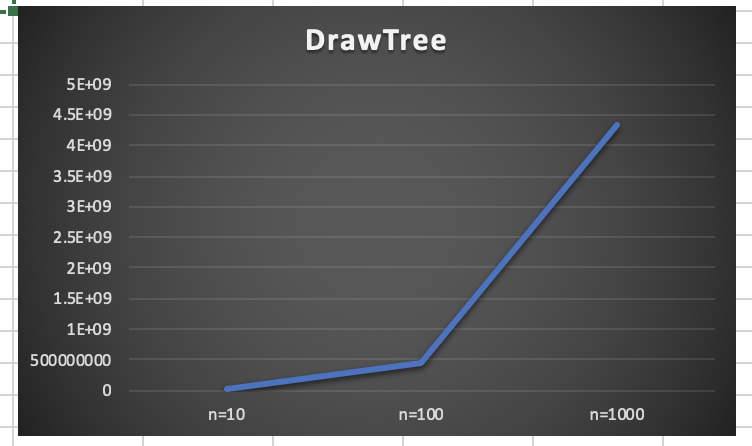


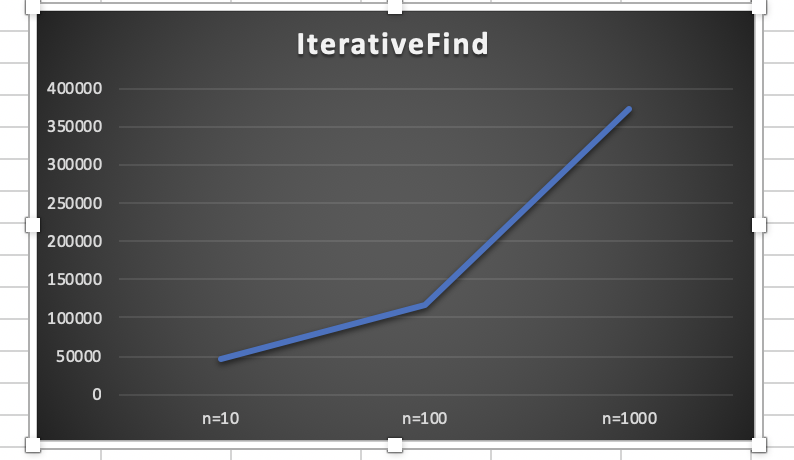


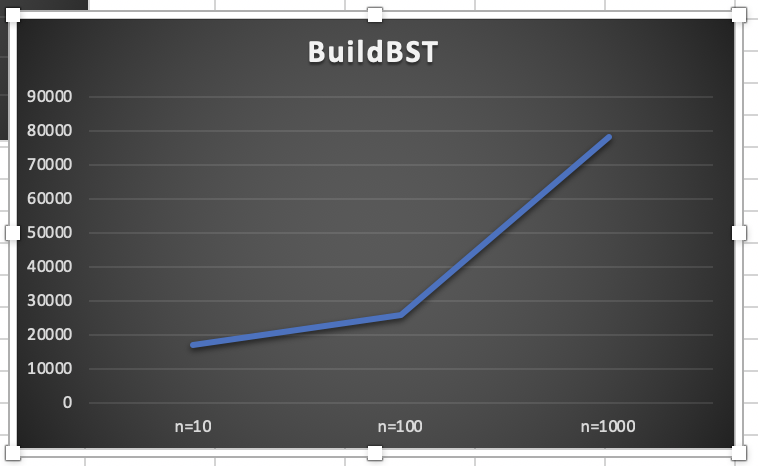
For iterative search, when I ran it when n=10, the key was not found (which was the worst case). For n=100 size BST with random integers, the key was in the tree as well as for n=1000.

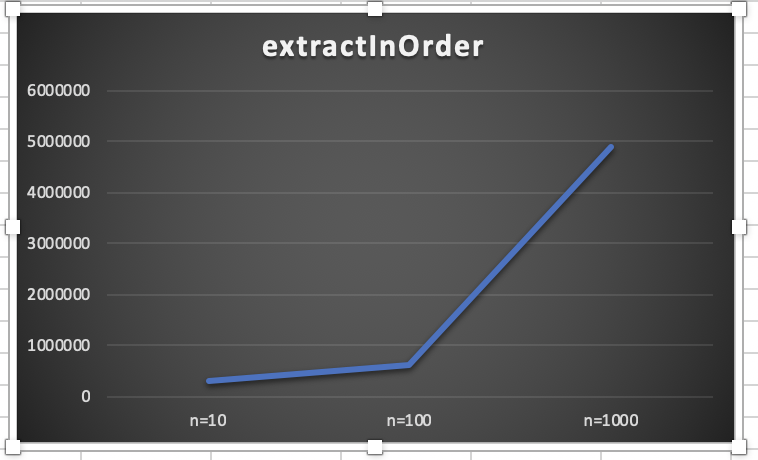
Runtimes in nanoseconds:

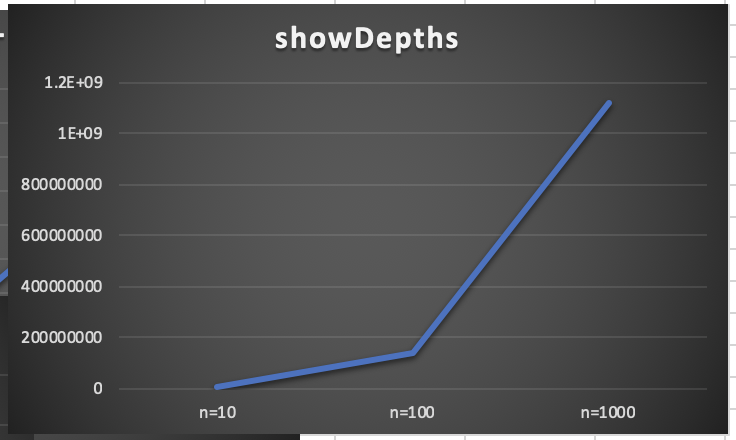












**Conclusion:**

From this lab, I learned how to do basic operations on a BST. I also got to incorporate what I did in lab 1 using python’s ﻿matplotlib.pyplot library in order to display a figure that gave a better visualization the actual BST. This also helped me practice recursion more and learn how to put elements onto a BST as well as take them out in order which I had never coded before. This made me see the usefulness of a BST. The fact that you can always split the work in half if you know where it is in relation to the root makes retrieval of items faster. It also taught me a new way of creating a BST with an already sorted list.

**Source code (Appendix):**

﻿

#CS2302

#Nicole Favela

#last modified: March 8, 2019

#Lab3

#instructor: Olac Fuentes

#TAs: Anindita Nath and Maliheh Zargaran

import math

import numpy as np

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 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')

#iterative search for part 2

def IterativeFind(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

while T is not None:

if k > T.item:

T = T.right

elif k < T.item:

T = T.left

elif k == T.item:

print('key found')

return T

else:

print('key not found')

return None

#part 3

#builds a bst from sorted array

def buildBST(A):

if not A:

return None

middle = (len(A))//2

root = BST(A[middle]) #makes root the middle element

root.left = buildBST(A[:middle]) #makes items less than left the, L subtree

root.right = buildBST(A[middle+1:]) #makes items greater, the R subtree

return root #returns reference to root

##part 4

#extracts the elements of bst in order

def extractInOrder(T,arr):

if T is not None:

extractInOrder(T.left,arr)

arr+=[T.item] #accumulates data in array

extractInOrder(T.right,arr)

return arr #returns array

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

#prints all nodes at a every depth for part 5

def ShowDepths(T):

counter=DepthCounter(T) #gets the depth of the tree

for i in range(counter): #prints cycled index and calls method to print keys at certain level

print('keys at depth',i,':',end=' '),findAtLevel(T,i)

print()

def DepthCounter(T):

if T is None:

return 0

CountL=DepthCounter(T.left) #gets count of depth in L tree

CountR=DepthCounter(T.right) #gets count of depth in R tree

if CountL>CountR:

return 1+CountL #returns larger count

return 1+CountR

#gets items at particular depths

def findAtLevel(T,Height):

if T is None:

return

if Height==0:

print(T.item,end=' ')

else:

findAtLevel(T.left,Height-1)

findAtLevel(T.right,Height-1)

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

#finds the max depth of the tree

def countDepth(T):

if T is None:

return 0

else:

depthOfLeft = 1+countDepth(T.left)

depthOfRight = 1+countDepth(T.right)

if depthOfLeft < depthOfRight:

return depthOfRight-1

else:

return depthOfLeft

#part 1 dispay figure

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

def DrawTree(ax,x,y,size,depth,T):

rad = 4

if T is not None:

draw\_circle(ax,[x,y],rad)

ax.text(x+2.2,y-1.8,T.item,size=10)

if T.left is not None:

draw\_line(ax,x,y,x-(2.5\*\*depth),y-10)

DrawTree(ax,x-(2.5\*\*depth),y-10,size,depth-1,T.left)

if T.right is not None:

draw\_line(ax,x+rad\*2,y,x+(2.5\*\*depth),y-10)

DrawTree(ax,x+(2.5\*\*depth),y-10,size,depth-1,T.right)

#draws lines equidistant aprt

def draw\_line(ax, x1,y1,x2,y2):

n = int(max( abs(x1-x2), abs(y1-y2)) )#the range of the lines

x = np.linspace(x1,x2,n)

y = np.linspace(y1,y2,n)

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

def draw\_circle(ax,center,radius):

x,y = circle(center,radius)

ax.plot((x+radius),y,color=(0,0,0))

def circle(center,rad):

n = int(4\*rad\*math.pi)#radius of each circle

t = np.linspace(0,6.3,n)#creating the circles

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

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

return x,y

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

T = None

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

sortedArray = [1,2,3,4,6,7,8,9]

#creates tree form array A

for a in A:

T = Insert(T,a)

InOrder(T)

#part 2 iterative search

result = IterativeFind(T,9)

print()

#returns the total depth of the tree

depthMax = countDepth(T)

print('depth of tree is:', depthMax)

print()

#draws tree image in graph

fig, ax = plt.subplots()

ax.set\_aspect(1.0)

#draws tree for part 1

DrawTree(ax,0,0,30,depthMax,T)

plt.show()

plt.axis('off')

print()

#part 4 extracting elements form BST to create array of sorted items

extractionArray = []

array = extractInOrder(T,extractionArray)

print('printing array extracted from bst:')

print(array)

print()

#part 5

ShowDepths(T)

#part 3 builds bst form sorted array

print('building bst from sorted array')

T = buildBST(sortedArray)

InOrderD(T,' ')