The task presented in this lab, was to create some functions related with binary search trees and Python native lists. Some of the functions were to find an element in the tree with out using recursion, to create a balanced binary search tree given a sorted list, create a sorted list given a binary search tree, print the elements of a given tree ordered by the depth where they are located.

To solve these problems, I tried tracing the process to understand the steps required to accomplish the objective, and to consider every possible situation. This is a better explanation of how I solved each problem:

**Search with iteration:** First of all, for this function was not allowed to use recursion, which would be the easiest solution. Considering that, I decided to use a why loop that will be executed only if the given tree is not empty. Inside the loop, the first thing I do is compare the current element with the one I am looking for, if this is true, the element has been found and it is returned. If the elements do not match, I compare the element that I am looking for with the root, if the element is smaller than the root, I just need to look for it in the left subtree, but if the element is bigger than the root, I will look for it in the right sub tree. And finally, if the element is not found in the tree, I will return none.

**Balanced tree:** Here we are assuming that the given list is already sorted. The first thing I do in this function is to check for an empty list, if this is the case, I return none. If this is not true, I look for the element in the middle of the list, and set it as the root of the tree. After that, to build the left subtree, I make a recursive call to this function, but only with the part of the list where I have elements that are smaller that the root. I do the same process for the right subtree, but in this case, I make a recursive call with the part of the list with elements bigger than the root. This process is repeated till the list is empty.

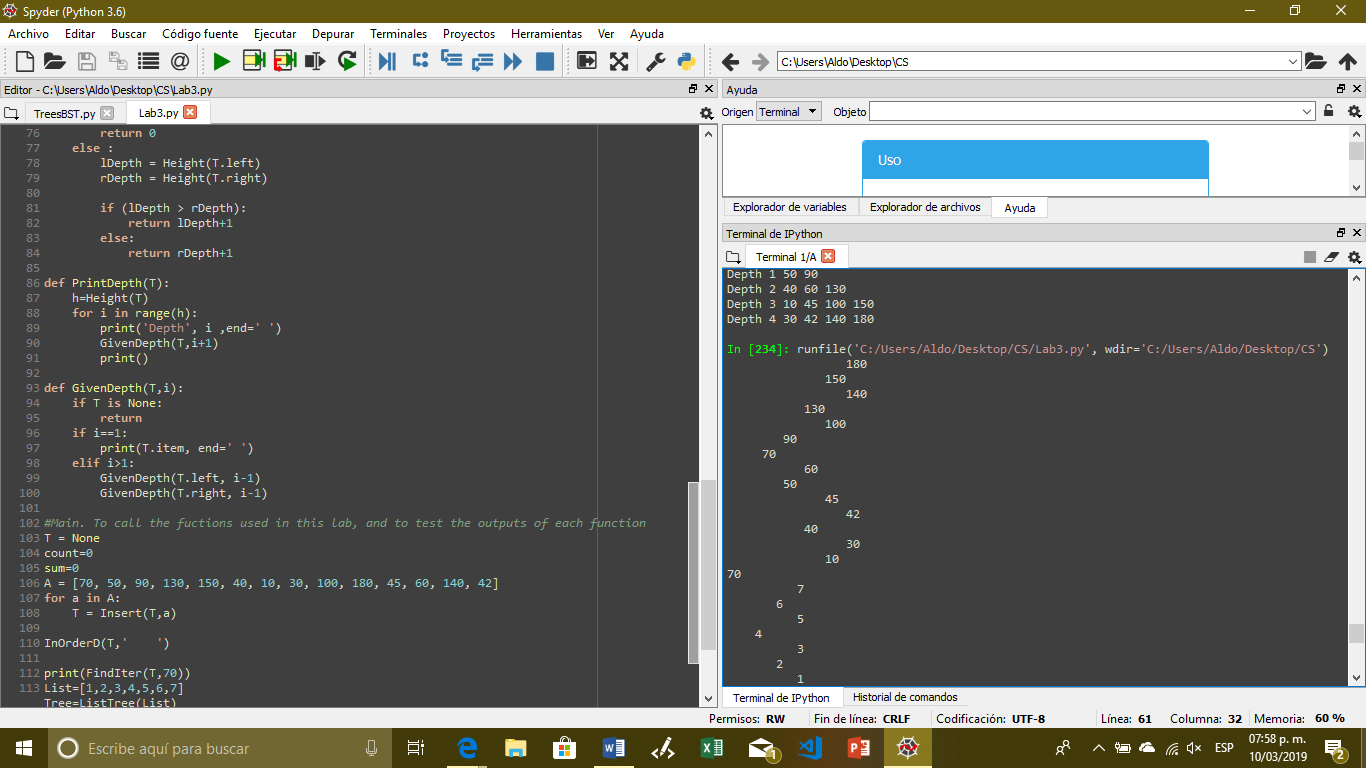
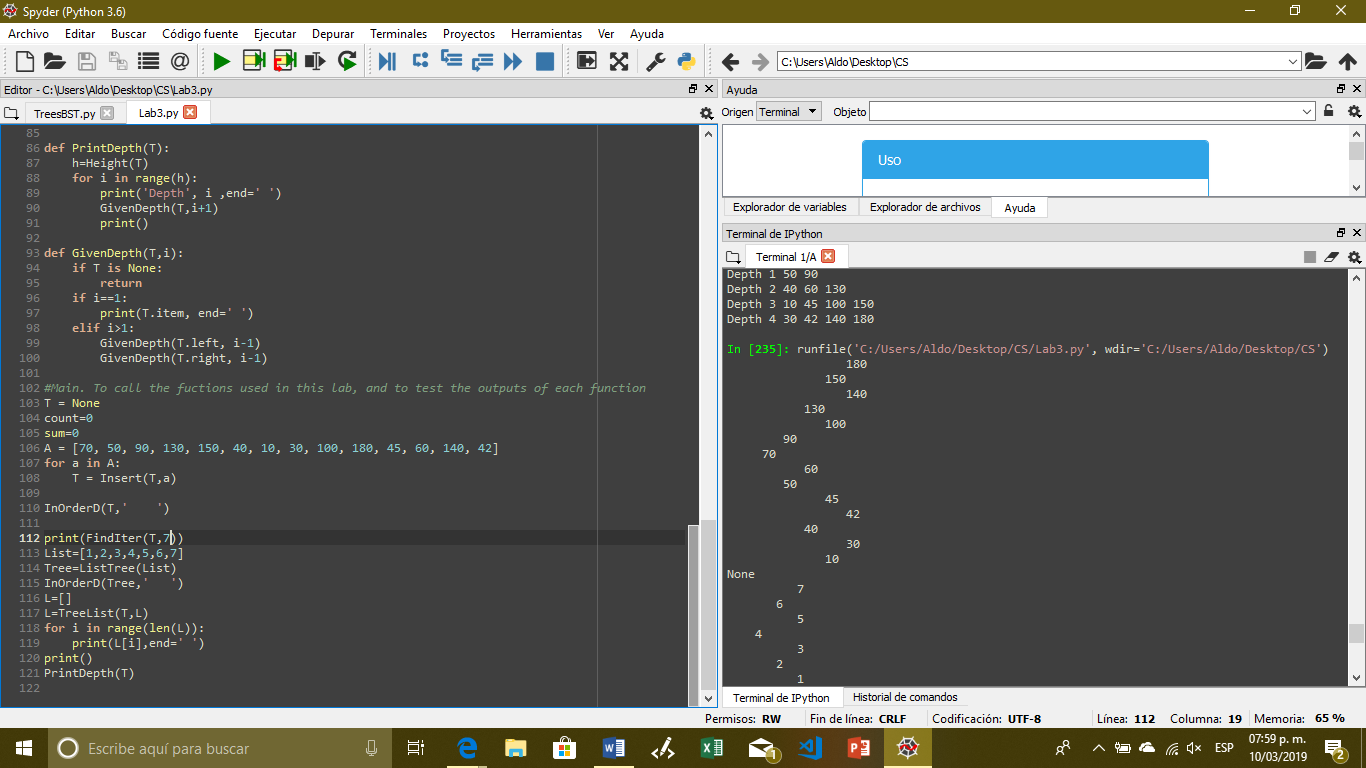
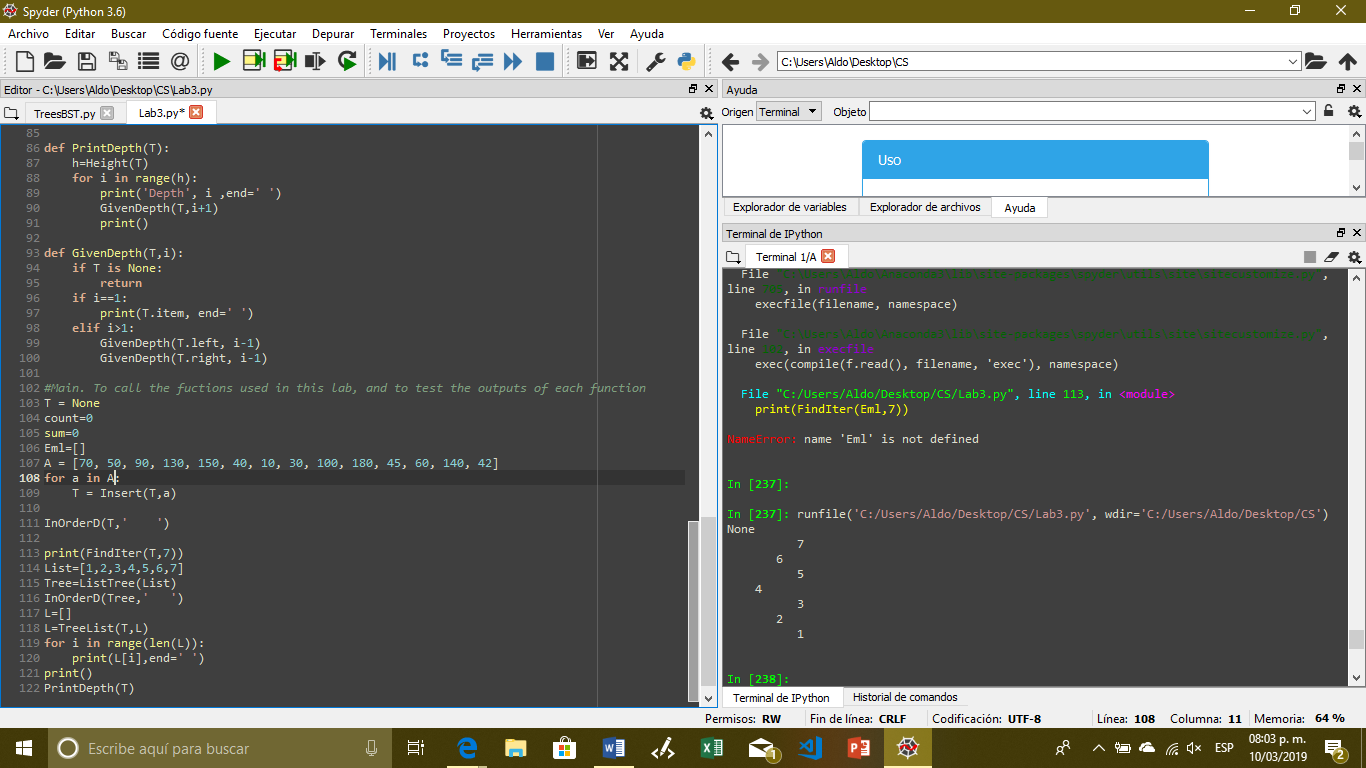
**Sorted list:** The first thing I do in this function, is to check for an empty tree, if this is the case, I return an empty list. If the tree is not empty, I start a recursive call with the left sub tree to add all the elements there at the beginning of the list, this because the elements in the left subtree are smaller than the root. After the left subtree is empty, I start a second recursive call with the right sub tree repeating the same process.

**Print by depth:** To solve this problem I used 3 functions, one to get the height of the tree, one to print the elements with a loop, and one to make sure that I am printing the elements in the right depth. In the method to print the elements, I use a for loop that will be executed times the height of the tree, then I print an indicator of the level I am printing, and finally I make a call to the method that will return the elements in that level.

After completing a function, I tested it with some inputs representing the possible situations. These are some of the test cases that I used:

**Search with iteration:**

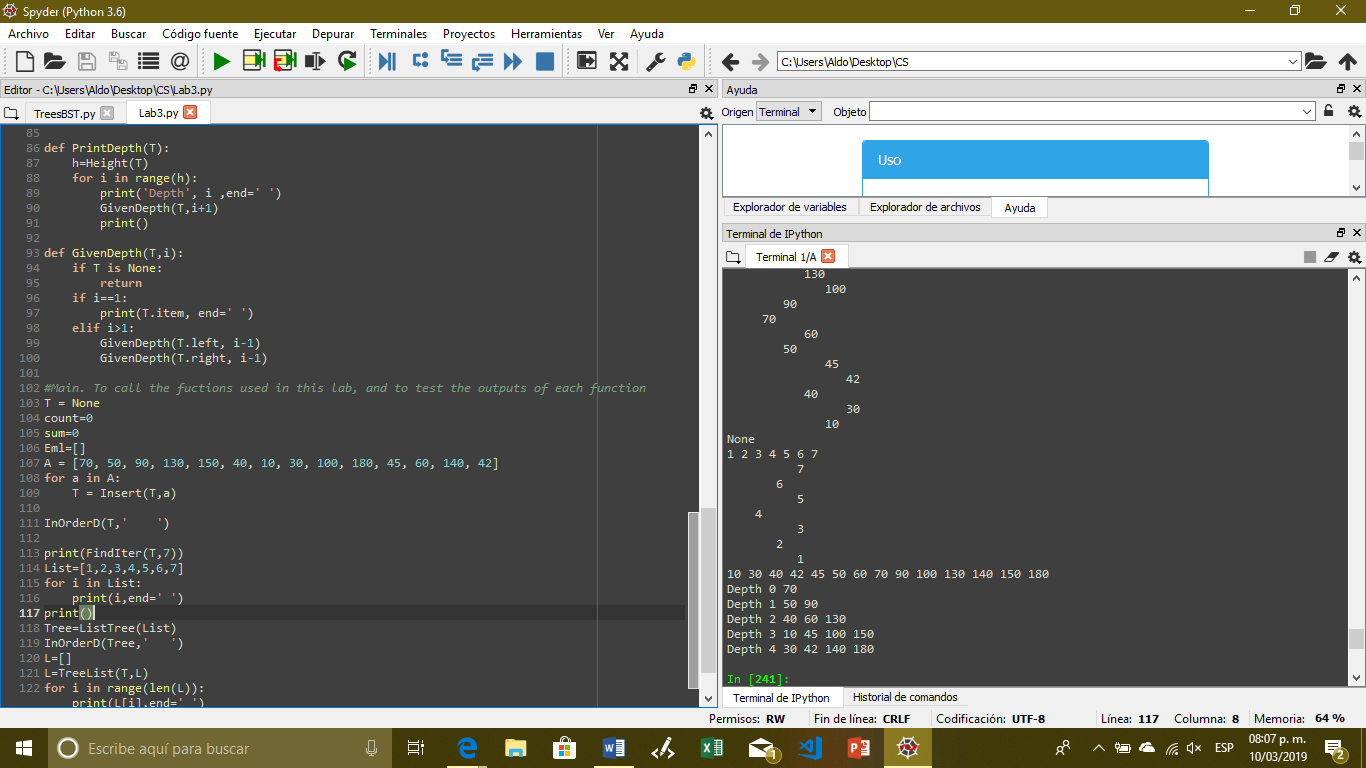
Input: A normal tree, elements present Input: Normal tree, element not present Input: An empty tree

**Balanced tree:**

Input: Normal list Input: Empty list

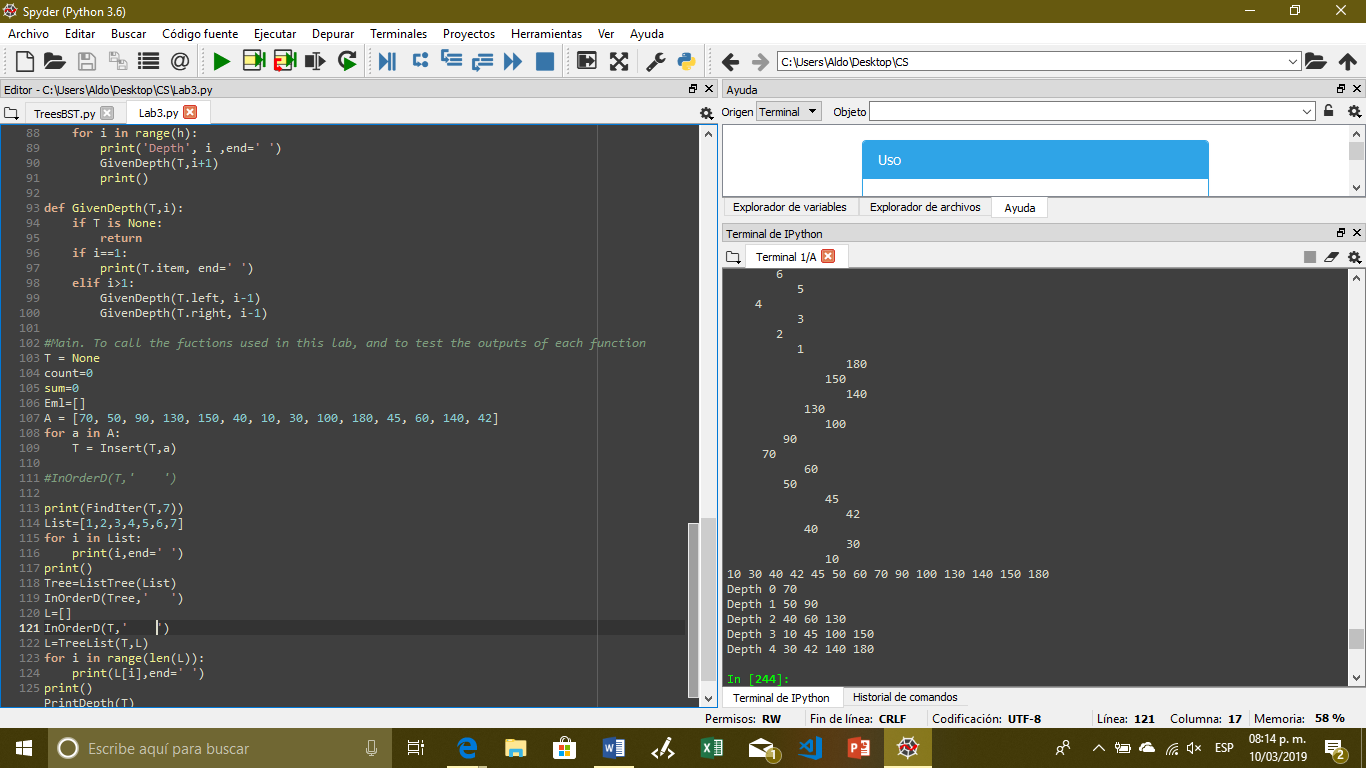
In this case if the list is empty nothing is printed



**Sorted list:**

Input: Normal tree Input: Empty tree

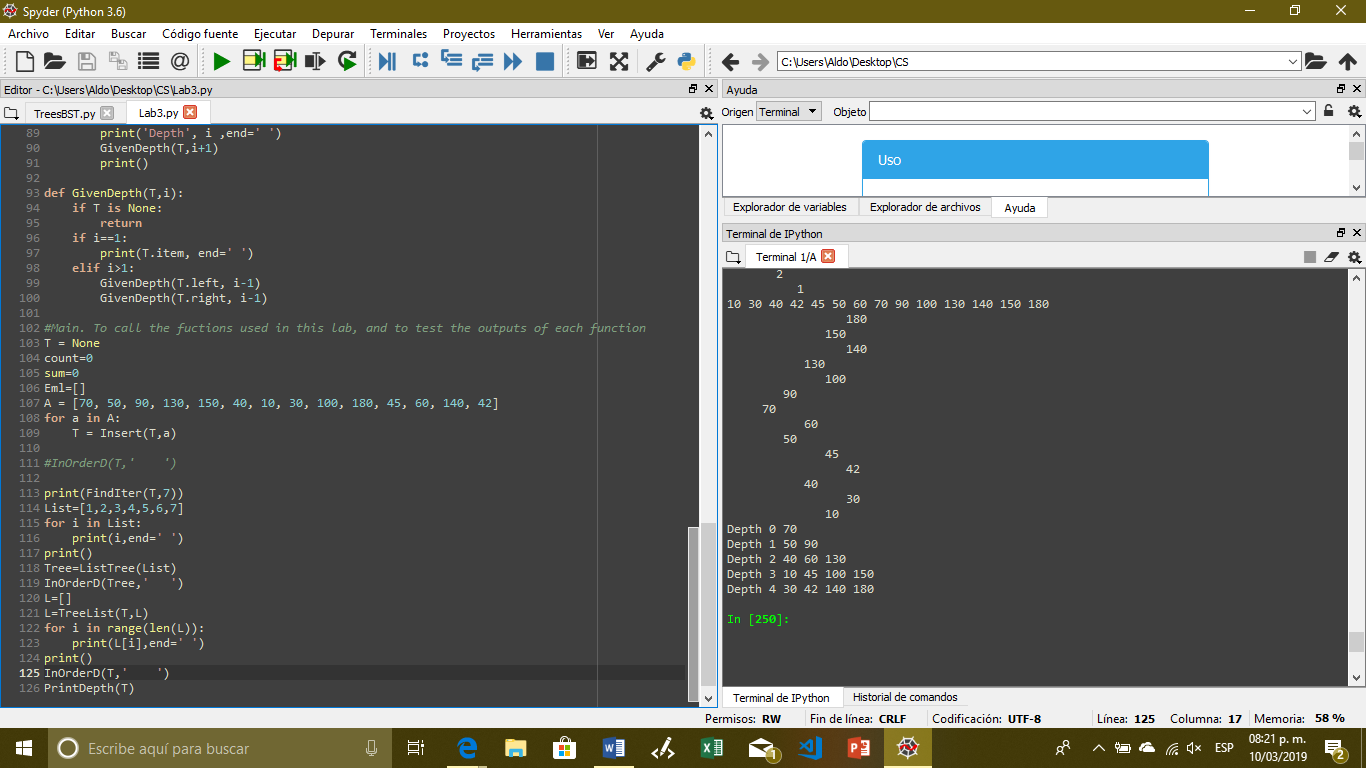
As in the previous function, if the tree is empty nothing is printed



**Print by depth:**

Input: Normal tree Input: Empty tree

Again, if the tree is empty, nothing is printed



After doing this lab I have a better notion of how binary search trees work, and how to handle errors related with them.

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 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+' ')

#2 This function recieves aa tree and a number, the purpose is to return the value if

#it is present in the tree, return none otherwise. This process shoul be done with out

#using recursion

def FindIter(Tree,k):

T=Tree

while T is not None:

if T.item==k:

return T.item

elif T.item>k:

T=T.left

else:

T=T.right

return None

#3 This function builds a balanced binary search tree using a sorted list

def ListTree(A):

if not A:

return None

mid=len(A)//2

T=BST(A[mid])

T.left=ListTree(A[:mid])

T.right=ListTree(A[mid+1:])

return T

#4 This function builds a sorted list using a binary search tree

def TreeList(T, A=[]):

if T is not None:

TreeList(T.left, A)

A+=[T.item]

TreeList(T.right,A)

return A

#5 This three functions are used to print the elements of a binary search tree

#ordered by depth

def Height(T):

if T is None:

return 0

else :

lDepth = Height(T.left)

rDepth = Height(T.right)

if (lDepth > rDepth):

return lDepth+1

else:

return rDepth+1

def PrintDepth(T):

h=Height(T)

for i in range(h):

print('Depth', i ,end=' ')

GivenDepth(T,i+1)

print()

def GivenDepth(T,i):

if T is None:

return

if i==1:

print(T.item, end=' ')

elif i>1:

GivenDepth(T.left, i-1)

GivenDepth(T.right, i-1)

#Main. To call the fuctions used in this lab, and to test the outputs of each function

T = None

count=0

sum=0

Eml=[]

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

for a in A:

T = Insert(T,a)

InOrderD(T,' ')

print(FindIter(T,7))

List=[1,2,3,4,5,6,7]

for i in List:

print(i,end=' ')

print()

Tree=ListTree(List)

InOrderD(Tree,' ')

L=[]

L=TreeList(T,L)

for i in range(len(L)):

print(L[i],end=' ')

print()

PrintDepth(T)

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.

Aldo A. Venzor