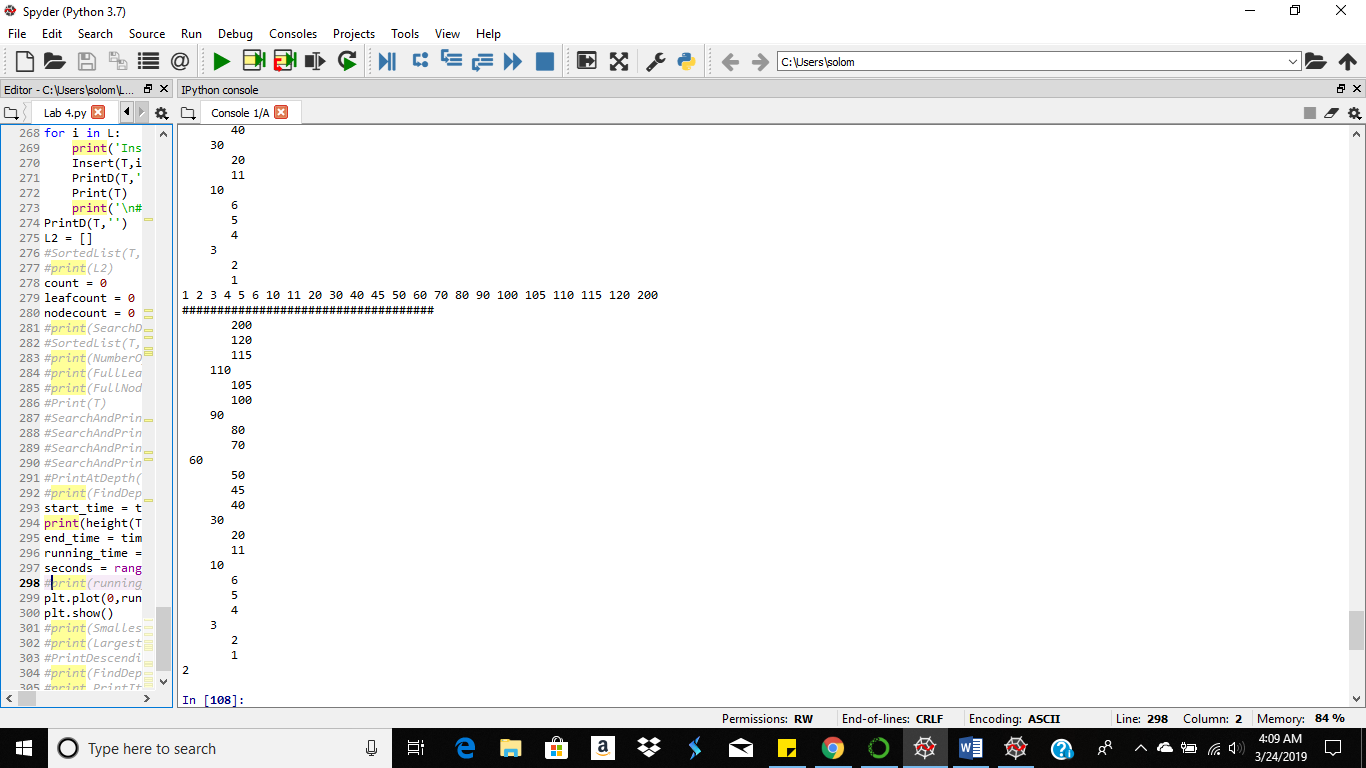
Solomon Davis Lab 4 Report

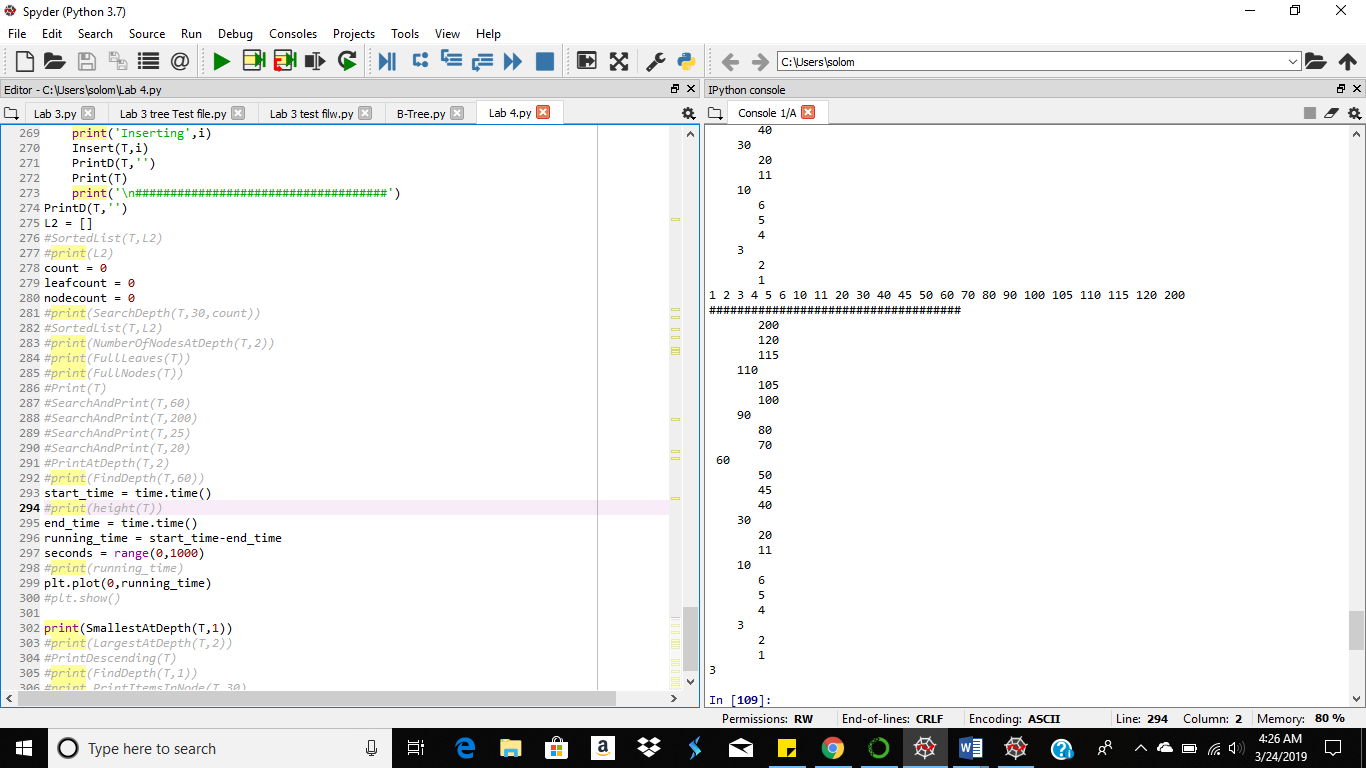
CS 2302 - MW 1:30 Spring 2019

In this lab I was able to carry out all the functions provided. For the height I had a count add 1 every time I went deeper into the tree. For the smallest value at a given depth I return the left most value at the giver depth. For the largest value at a given depth I return the right most value at the giver depth. For putting the list in sorted order, I appended the values of the list from the b-tree in order of smallest to largest. For printing the items at a given depth I simply printing all the items when d was 0. For returning the number of number of nodes at a given depth I had a count that added 1 when a node was found at the given depth. For counting the number of full nodes, I simply had a count that increased depending on the size of the node. For counting the number of full leaves, I simply had a count that increased depending on the size of the leaf. For finding the depth of a given key I returned the count which kept track of the depth at which the key was found.

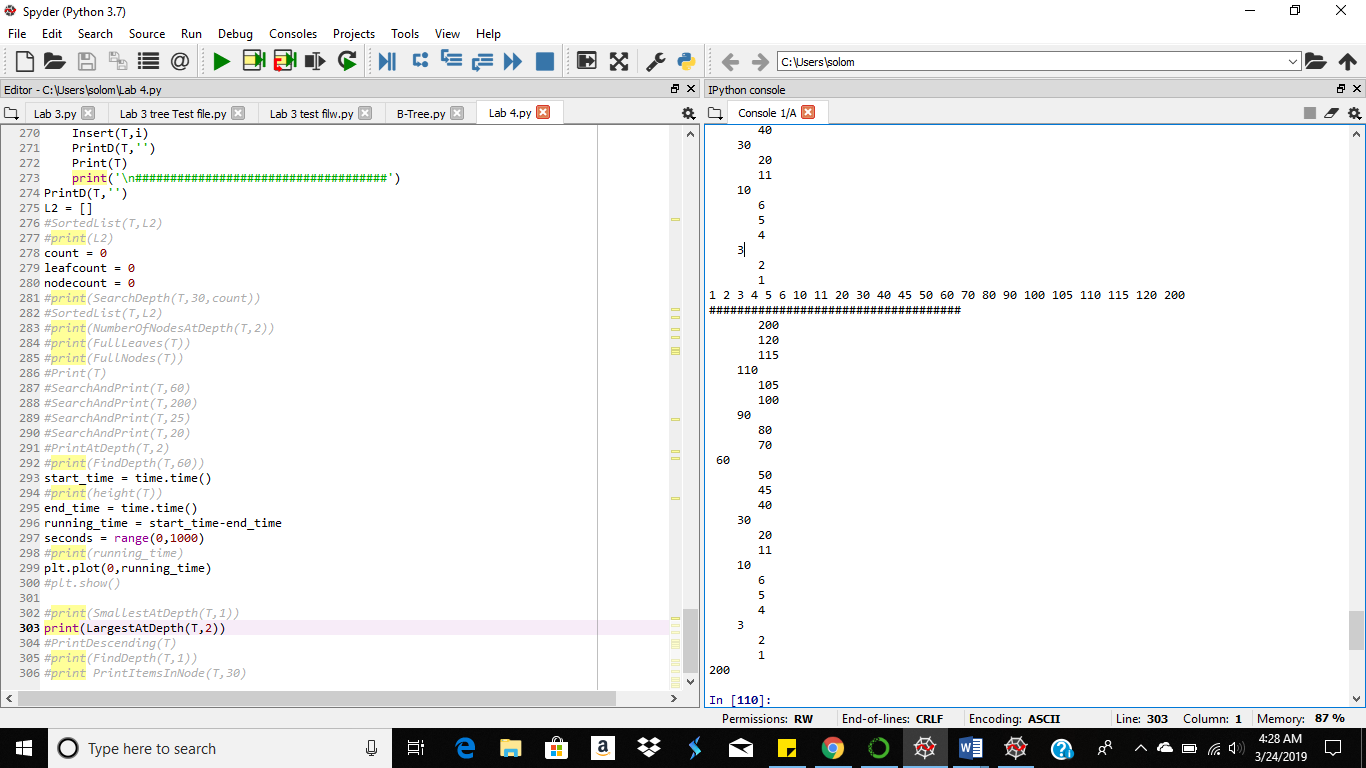
Height of Tree



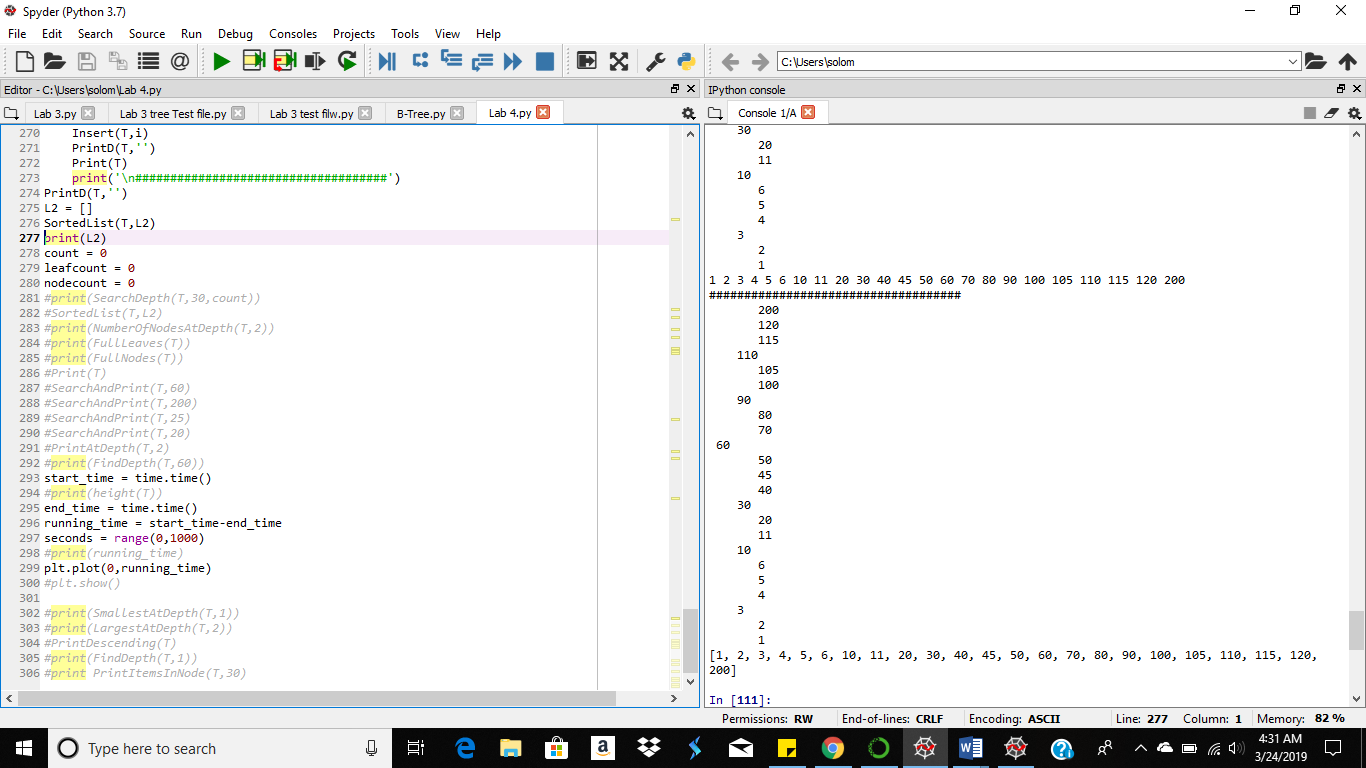
Smallest value at depth 1



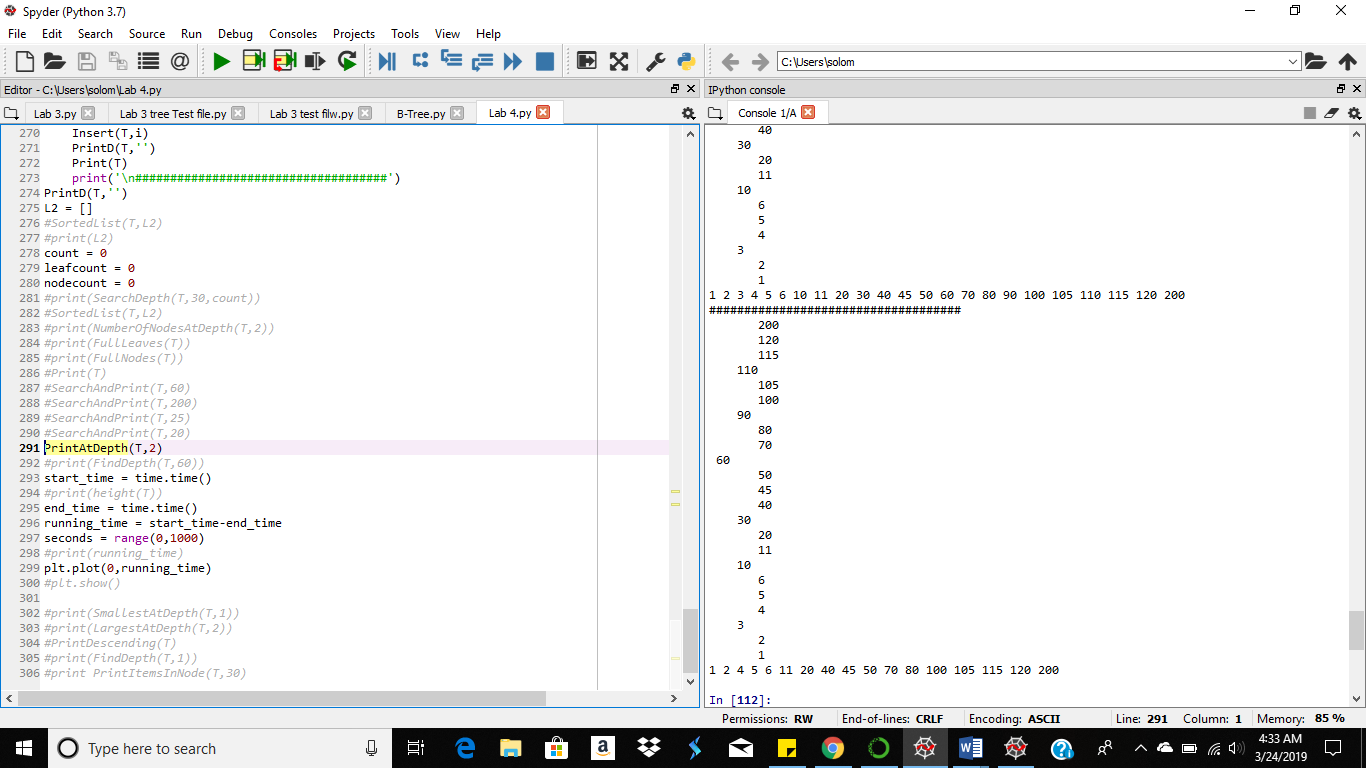
Largest value at depth 2



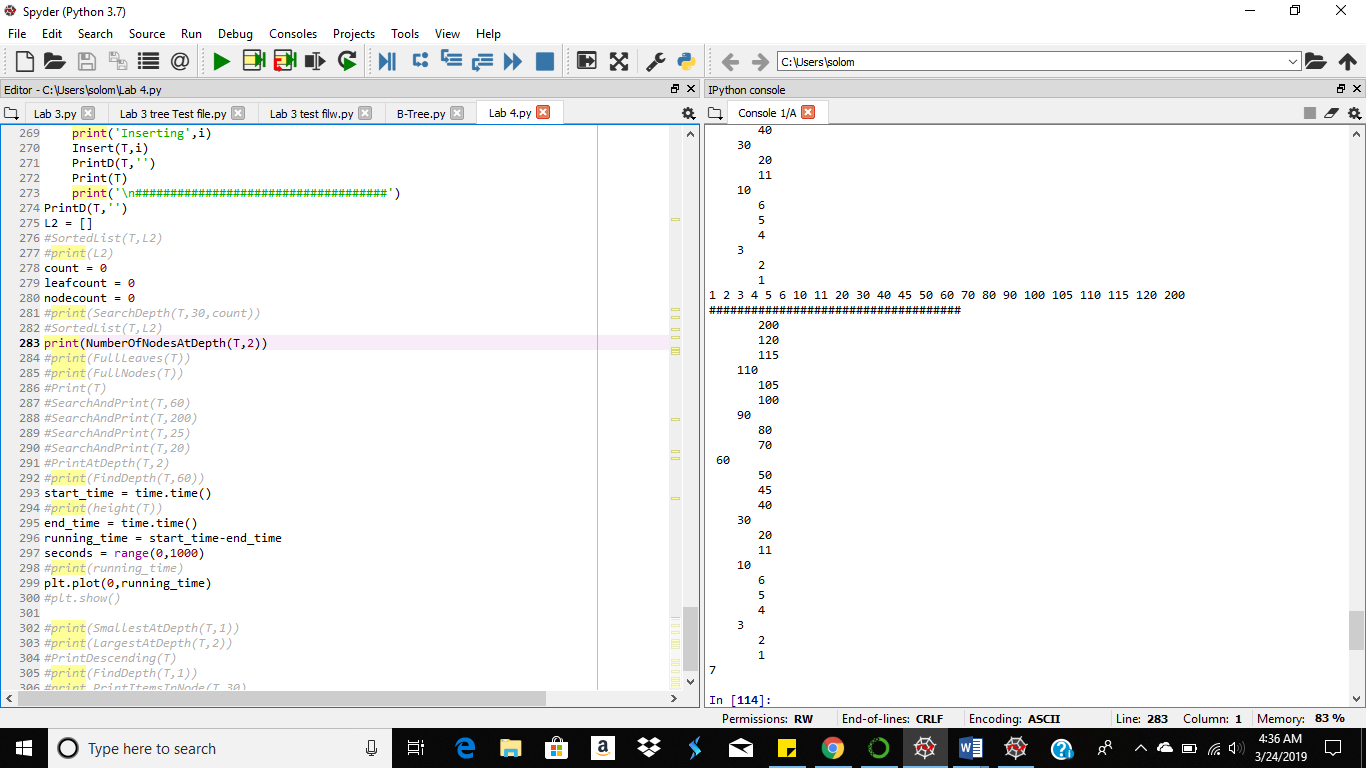
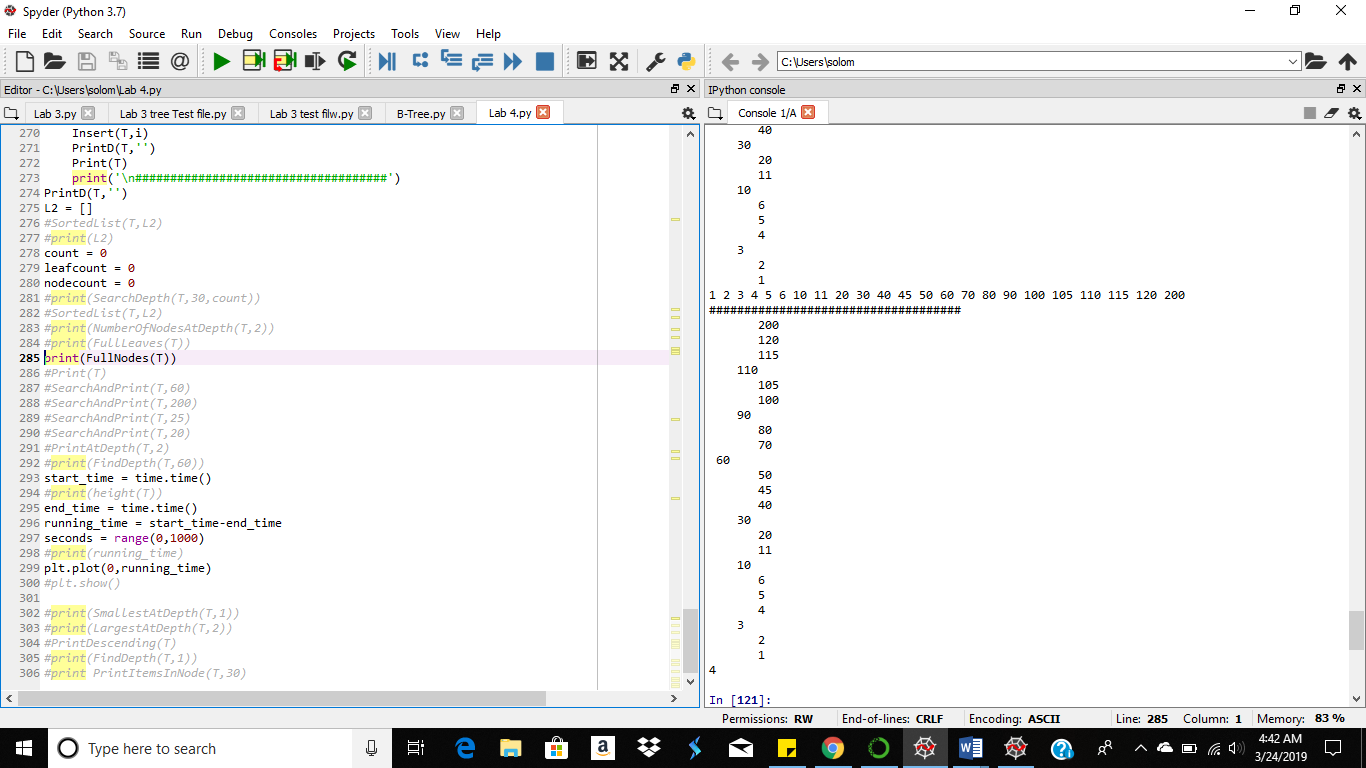
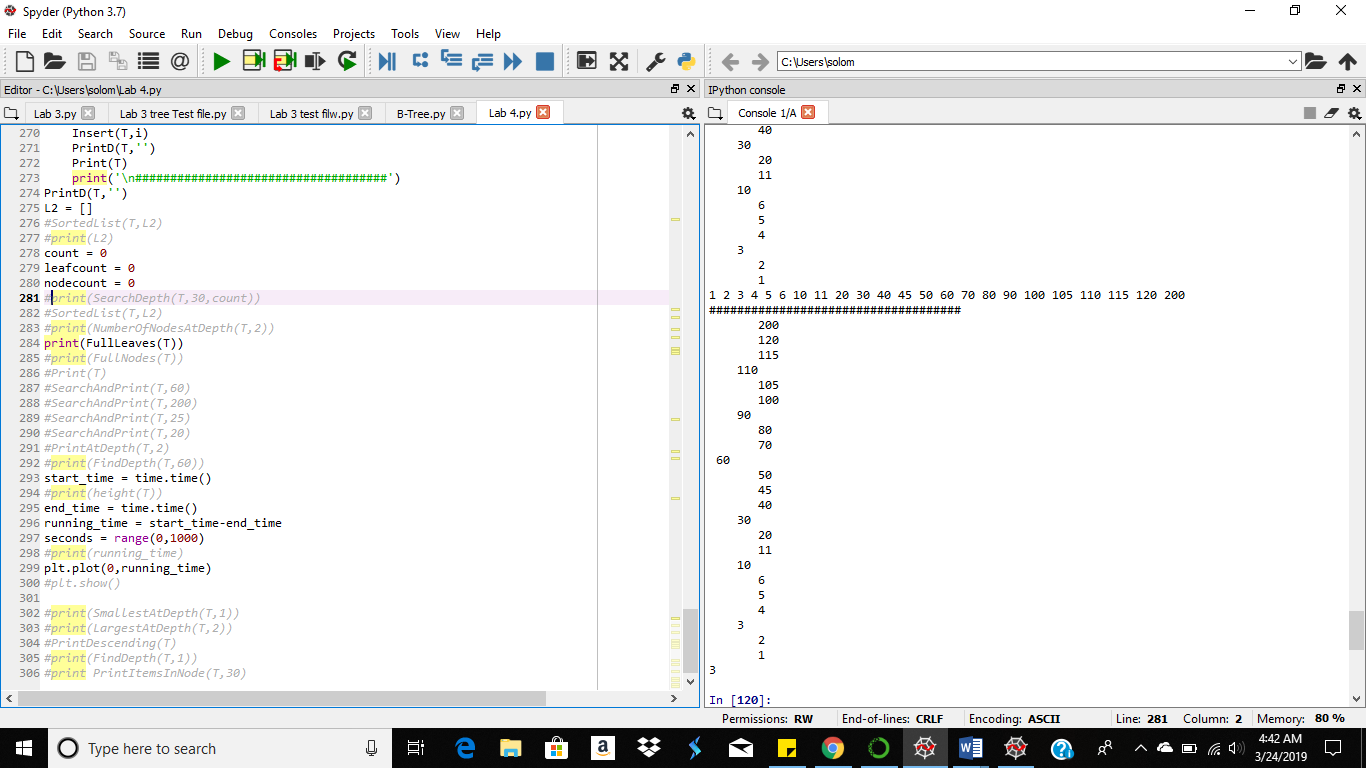
Sorted List



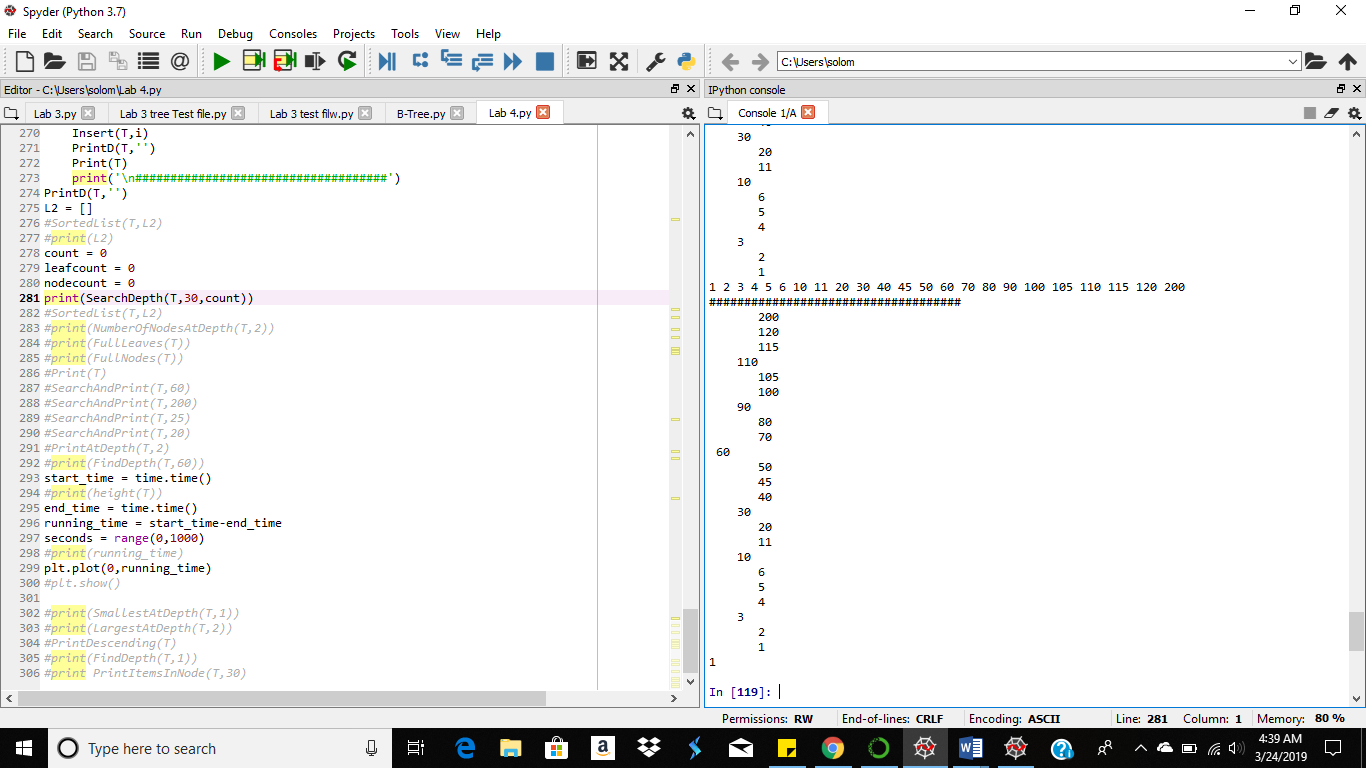
Returns the number of nodes at depth 2



Returns the number of nodes at depth 2 Number of full nodes Number of full leaves

Searching for the depth where value 30 is found



#Course: CS2302 - Spring 2019

#Author: Solomon Davis

#Lab Number: 4

#Instructor: Olac Fuentes

#Last Modified: March 24, 2019

#Due Date: March 15, 2019

#Description: This code will use b-trees to carry out specific tasks. These

#tasks include commputing the height of the tree,extracting items from a b-tree

#into a sorted list, returning the minimum and maximum value, returning the

#number of nodes at a specific depth, printing the values at a specific depth,

#returning the of full nodes and full trees, and lastly given the depth of a

#specific key in the b-tree.

import time

import matplotlib.pyplot as plt

class BTree(object):

# Constructor

def \_\_init\_\_(self,item=[],child=[],isLeaf=True,max\_items=5):

self.item = item

self.child = child

self.isLeaf = isLeaf

if max\_items <3: #max\_items must be odd and greater or equal to 3

max\_items = 3

if max\_items%2 == 0: #max\_items must be odd and greater or equal to 3

max\_items +=1

self.max\_items = max\_items

def FindChild(T,k):

# Determines value of c, such that k must be in subtree T.child[c], if k is in the BTree

for i in range(len(T.item)):

if k < T.item[i]:

return i

return len(T.item)

def InsertInternal(T,i):

# T cannot be Full

if T.isLeaf:

InsertLeaf(T,i)

else:

k = FindChild(T,i)

if IsFull(T.child[k]):

m, l, r = Split(T.child[k])

T.item.insert(k,m)

T.child[k] = l

T.child.insert(k+1,r)

k = FindChild(T,i)

InsertInternal(T.child[k],i)

def Split(T):

#print('Splitting')

#PrintNode(T)

mid = T.max\_items//2

if T.isLeaf:

leftChild = BTree(T.item[:mid])

rightChild = BTree(T.item[mid+1:])

else:

leftChild = BTree(T.item[:mid],T.child[:mid+1],T.isLeaf)

rightChild = BTree(T.item[mid+1:],T.child[mid+1:],T.isLeaf)

return T.item[mid], leftChild, rightChild

def InsertLeaf(T,i):

T.item.append(i)

T.item.sort()

def IsFull(T):

return len(T.item) >= T.max\_items

def Insert(T,i):

if not IsFull(T):

InsertInternal(T,i)

else:

m, l, r = Split(T)

T.item =[m]

T.child = [l,r]

T.isLeaf = False

k = FindChild(T,i)

InsertInternal(T.child[k],i)

def height(T):

#Retutns the height of the B-tree

if T.isLeaf:

return 0

return 1 + height(T.child[0])

def Search(T,k):

# Returns node where k is, or None if k is not in the tree

if k in T.item:

return T

if T.isLeaf:

return None

return Search(T.child[FindChild(T,k)],k)

def Print(T):

# Prints items in tree in ascending order

if T.isLeaf:

for t in T.item:

print(t,end=' ')

else:

for i in range(len(T.item)):

Print(T.child[i])

print(T.item[i],end=' ')

Print(T.child[len(T.item)])

def PrintD(T,space):

# Prints items and structure of B-tree

if T.isLeaf:

for i in range(len(T.item)-1,-1,-1):

print(space,T.item[i])

else:

PrintD(T.child[len(T.item)],space+' ')

for i in range(len(T.item)-1,-1,-1):

print(space,T.item[i])

PrintD(T.child[i],space+' ')

def SearchAndPrint(T,k):

node = Search(T,k)

if node is None:

print(k,'not found')

else:

print(k,'found',end=' ')

print('node contents:',node.item)

def Smallest(T):

if T.isLeaf:

return T.item[0]

return Smallest(T.child[0])

def Largest(T):

if T.isLeaf:

return T.item[-1]

return Largest(T.child[len(T.item)])

def SmallestAtDepth(T,d):

#Returns the smallest value at the given depth d

if T.isLeaf: #If at depth d is the Btree is a leaf it returns the item with the smallest value

return T.item[0]

if d==0: #If the depth is 0 the smallest value is then returned from the given values

return T.item[0]

return SmallestAtDepth(T.child[0],d-1) #return the left most child in the Btree until the function is in the correct depth

def LargestAtDepth(T,d):

#Returns the largest value at the given depth d

if T.isLeaf: #If at depth d is the Btree is a leaf it returns the item with the largest value

return T.item[-1]

if d==0: #If the depth is 0 the largest value is then returned from the given values

return T.item[-1]

return LargestAtDepth(T.child[len(T.item)],d-1) #return the right most child in the Btree until the function is in the correct depth

def NumItems(T):

#count = 0

if T.isLeaf:

return len(T.item)

for i in range(len(T.child)):

NumItems(T.child[i])

#count +=1

#print(count)

return NumItems(T.child[i]) + len(T.item)

def FindDepth(T,k):

if k != T.item:

return FindDepth(T.item,k) + 1

if k == T.item:

return 0

#for i in range(len(T.item)-1,-1,-1)

if k>T.item:

return FindDepth(T.item,k) + 1

if k<T.item:

return FindDepth(T.item,k) + 1

def PrintDescending(T):

if T.isLeaf:

for i in range(len(T.item)-1,-1,-1):

print(T.item[i],end=' ')

else:

PrintDescending(T.child[len(T.item)])

for i in range(len(T.item)-1,-1,-1):

print(T.item[i],end=' ')

PrintDescending(T.child[i])

def PrintItemsInNode(T,k):

if T.child.item[i] == k:

return -1

if T.child.item[i] < k:

return PrintItemsInNode(T,k)

return PrintItemsInNode(T,k)

#def Height(T):

# if T.isLeaf:

# return 1

# else:

# for i in range(len(T.item)):

# return 1 + Height(T.child[i])

def PrintAtDepth(T,d):

#Prints all the values at the given depth in the b-tree

if d == 0:

for i in range(len(T.item)):

print(T.item[i],end=' ') #Print the item in the tree

for i in range(len(T.child)): #Loop that traverses through entire the Btree

PrintAtDepth(T.child[i],d-1) #Calls back to the function until the funtion is in the correct depth

def NumberOfNodesAtDepth(T,d):

#Count the number of nodes at a given a depth d

global count

if d == 0:

count +=1 #Count that adds one when there is a value at a depth

for i in range(len(T.child)): #Loop that traverses through entire the Btree

NumberOfNodesAtDepth(T.child[i],d-1) #Calls back to the function until the funtion is in the correct depth

return count

def SortedList(T,L2):

#Takes the values in the b-tree and puts it into a sorted list

if T.isLeaf:

for t in T.item:

L2.append(t) #Appends value from tree to list

else:

for i in range(len(T.item)): #Loop that traverses through entire the Btree

SortedList(T.child[i],L2)

L2.append(T.item[i]) #Appends value from tree to list

SortedList(T.child[len(T.item)],L2)

def SearchDepth(T,k,count):

# Returns Depth where k is, or -1 if k is not in the tree

if k in T.item: #If the value k is equal to the item in the Btree the count is returned

return count

if T.isLeaf: #If the value k is not found in the Btree then -1 is returned

return -1

#count +=1

return SearchDepth(T.child[FindChild(T,k)],k,count+1)

def FullNodes(T):

#Returns the number of full nodes in the b-tree

global nodecount

if T.isLeaf:

if len(T.item) == 3:

nodecount +=1

if len(T.child) == 3:

nodecount +=1

for i in range(len(T.child)): #Loop that traverses through entire the Btree

FullNodes(T.child[i])

return nodecount #Returns the count of full nodes

def FullLeaves(T):

#Returns the number of full leaves in the b-tree

global leafcount

if T.isLeaf:

if len(T.item) == 3: #If the length of the leaf is 3 the count adds 1 to the counter

leafcount +=1

else:

for i in range(len(T.child)): #Loop that traverses through entire the Btree

FullLeaves(T.child[i])

return leafcount #Returns the count of full leaves

L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11 , 3, 4, 5,105, 115, 200, 2, 45, 6]

T = BTree()

for i in L:

print('Inserting',i)

Insert(T,i)

PrintD(T,'')

Print(T)

print('\n####################################')

PrintD(T,'')

L2 = []

#start\_time = time.time()

SortedList(T,L2)

#end\_time = time.time()

#running\_time = start\_time-end\_time

#plt.plot(0,running\_time,'o',color='k')

print(L2)

count = 0

leafcount = 0

nodecount = 0

#start\_time1 = time.time()

print(SearchDepth(T,30,count))

#end\_time1 = time.time()

#running\_time1 = start\_time1-end\_time1

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

#plt.plot(1,running\_time1,'o',color='k')

#start\_time2 = time.time()

print(NumberOfNodesAtDepth(T,2))

#end\_time2 = time.time()

#running\_time2 = start\_time2-end\_time2

#plt.plot(2,running\_time2,'o',color='k')

#start\_time3 = time.time()

print(FullLeaves(T))

#end\_time3 = time.time()

#running\_time3 = start\_time3-end\_time3

#plt.plot(3,running\_time3,'o',color='k')

#start\_time4 = time.time()

print(FullNodes(T))

#end\_time4 = time.time()

#running\_time4 = start\_time4-end\_time4

#plt.plot(4,running\_time4,'o',color='k')

#Print(T)

#SearchAndPrint(T,60)

#SearchAndPrint(T,200)

#SearchAndPrint(T,25)

#SearchAndPrint(T,20)

#start\_time5 = time.time()

PrintAtDepth(T,2)

#end\_time5 = time.time()

#running\_time5 = start\_time5-end\_time5

#plt.plot(5,running\_time5,'o',color='k')

#print(FindDepth(T,60))

#start\_time6 = time.time()

print(height(T))

#end\_time6 = time.time()

#running\_time6 = start\_time6-end\_time6

#plt.plot(6,running\_time6,'o',color='k')

#seconds = range(0,1000)

#print(running\_time)

#plt.plot(6,running\_time,'o',color='k')

#plt.show()

#start\_time7 = time.time()

print(SmallestAtDepth(T,1))

#end\_time7 = time.time()

#running\_time7 = start\_time7-end\_time7

#plt.plot(7,running\_time7,'o',color='k')

#start\_time8 = time.time()

print(LargestAtDepth(T,2))

#end\_time8 = time.time()

#running\_time8 = start\_time8-end\_time8

#plt.plot(8,running\_time8,'o',color='k')

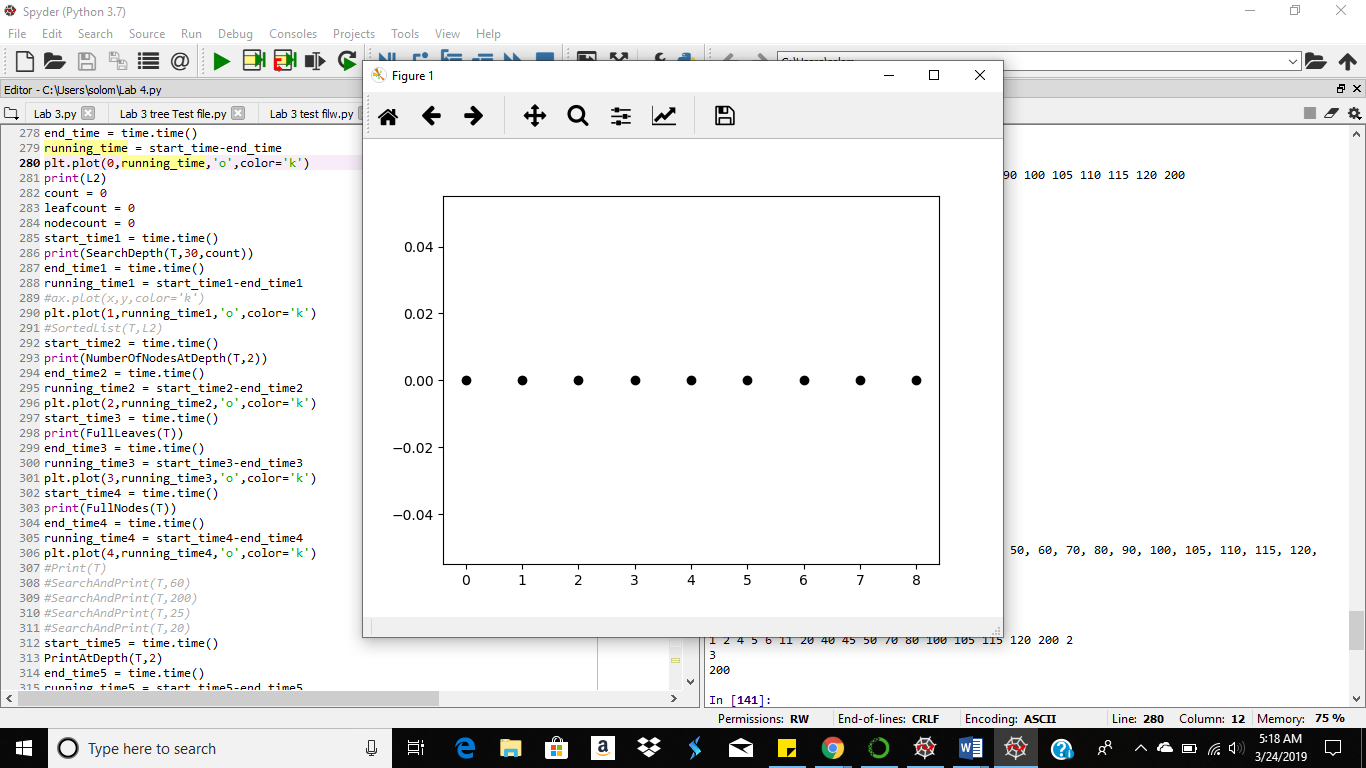
#PrintDescending(T)

#print(FindDepth(T,1))

#print PrintItemsInNode(T,30)

#plt.show()

Running Times (Seconds) – O(logn)

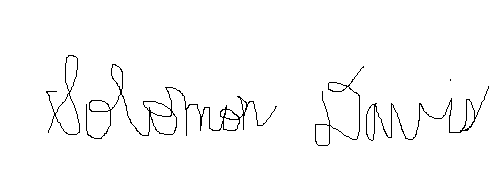


Function

Running times are linear according to the graph

Academic Service Certificate:

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.



Solomon Davis