Luis Renteria

ID 88740232

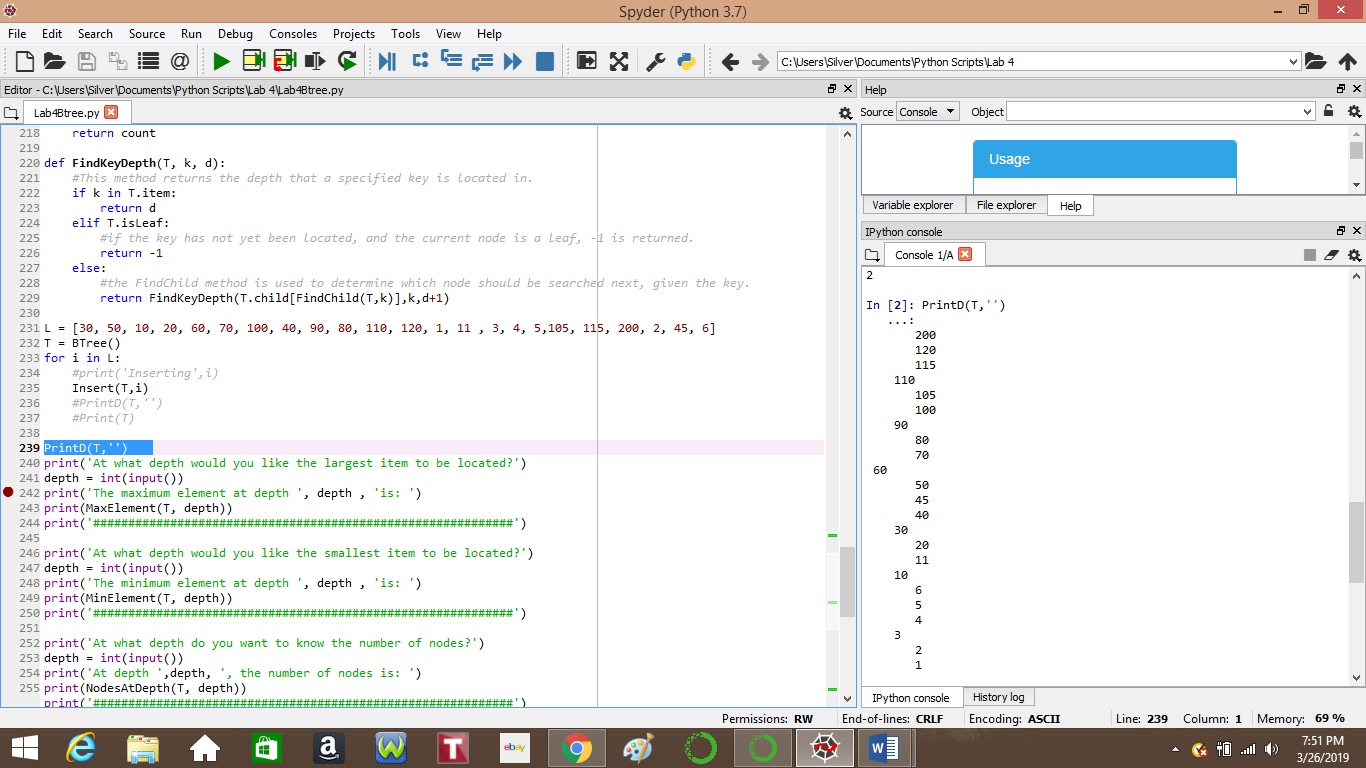
CS 2302

Dr. Fuentes

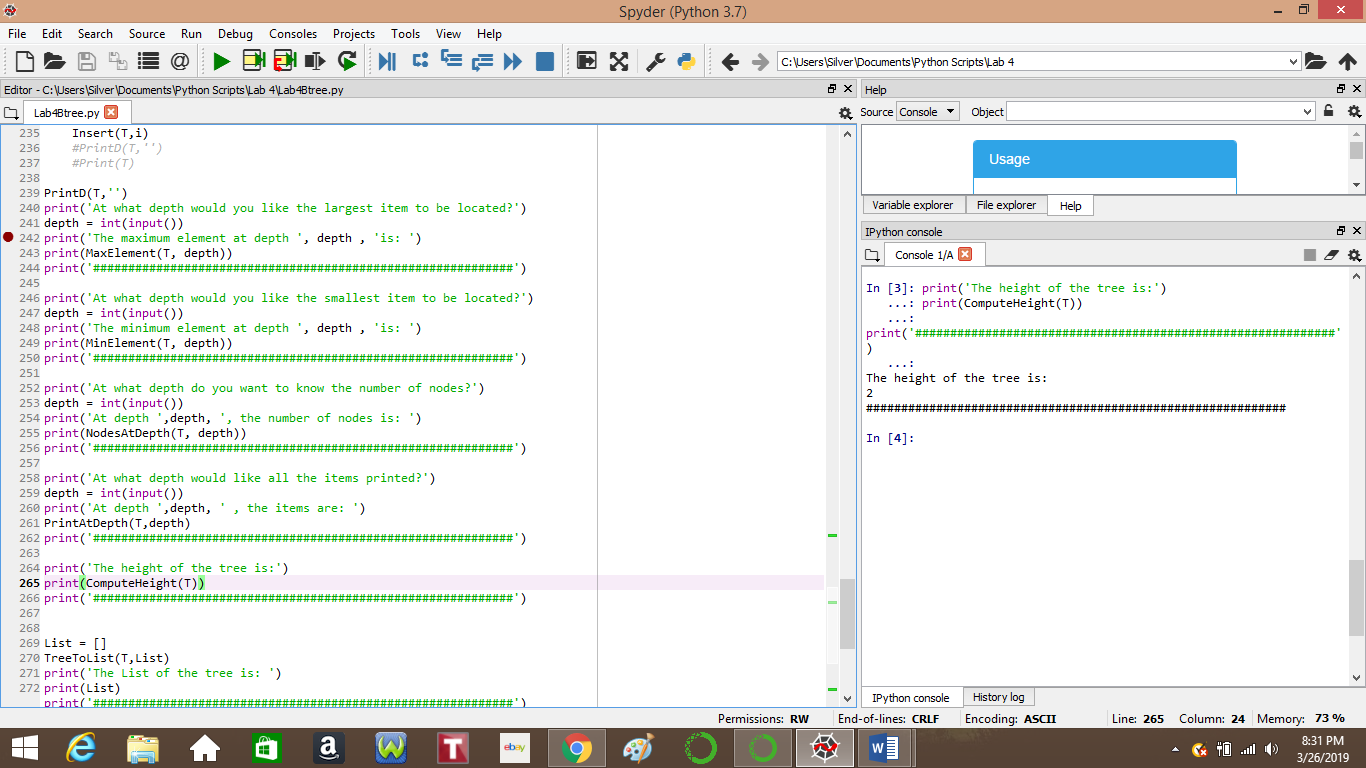
Lab Report 4

**Introduction**

The objective of our fourth lab project was to utilize a b-tree data structure to perform different tasks. Some of the tasks included finding the minimum and maximum, determining the number of nodes at certain depths and determining the number of nodes that are full. The following b-tree was used for testing, with some modifications made for a few methods.

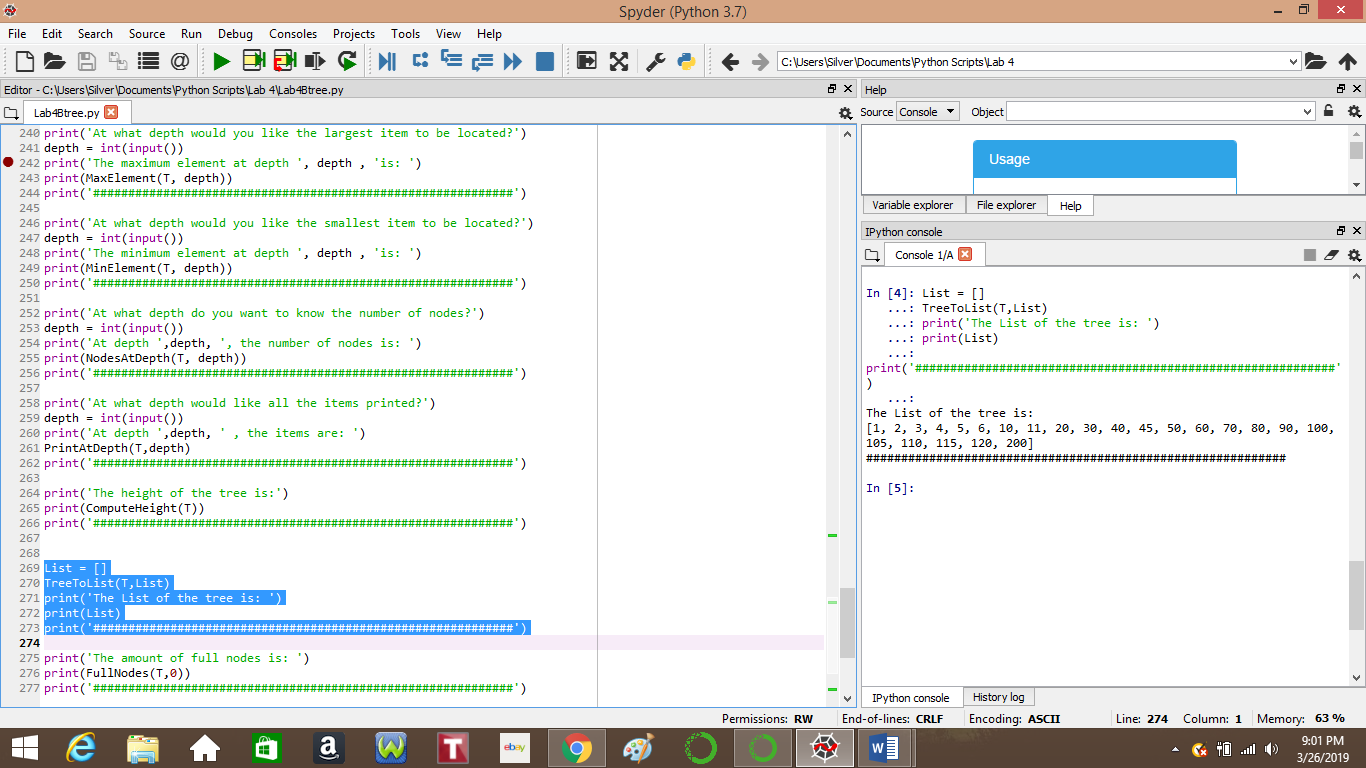


**B-tree Height**

Our first task was to determine the height of a b-tree. In order to determine this, I added one, and recursively called the method with the child of the previous node until a leaf is reached, which is when I return 0. The method overall returns the number of ones added, which is equivalent to the height of the tree.

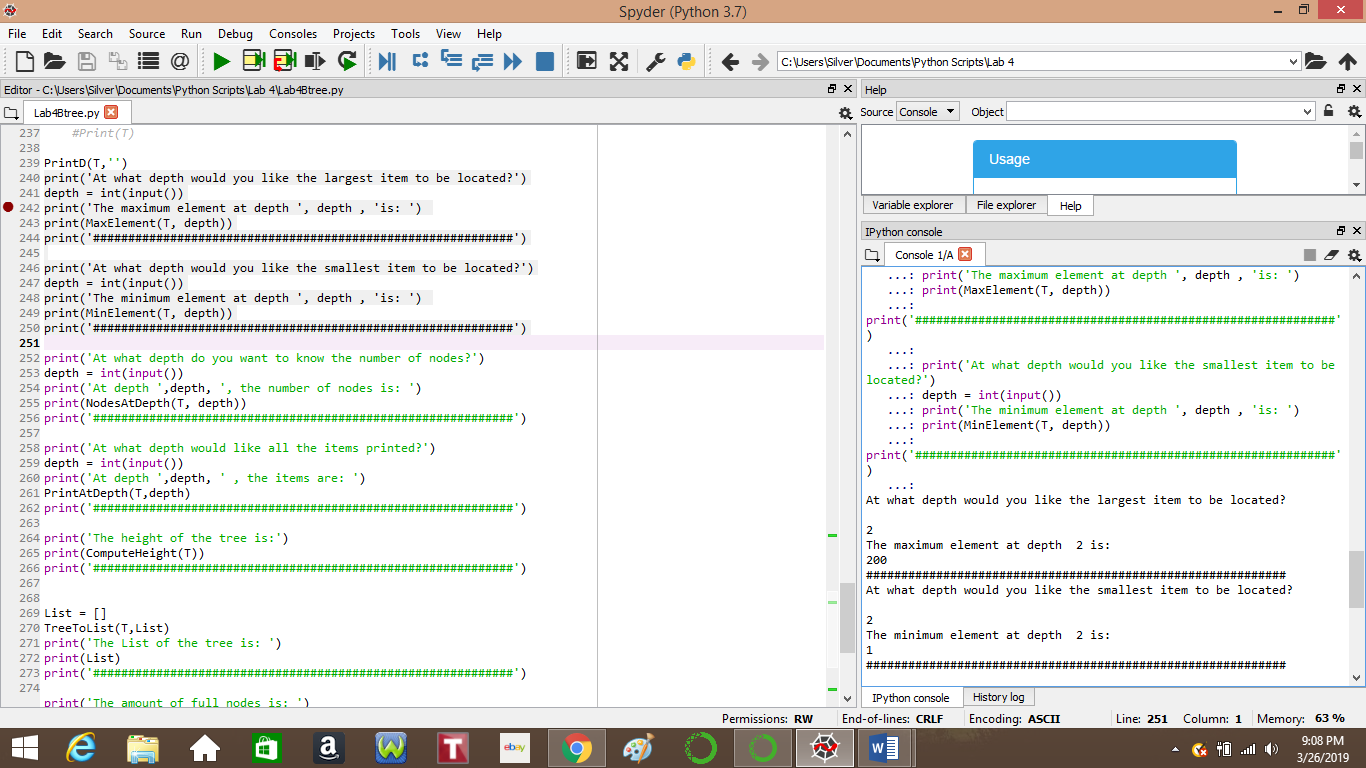
**B-tree to List**

Our second objective was to take a b-tree and input all the elements of the tree into a sorted list. In order to do this, I first recursively moved through the tree until a parent of a leaf was reached. Using a for loop, the children of the parent were added to the end of the list while a single element of the parent was added between each child to ensure an ascending order.



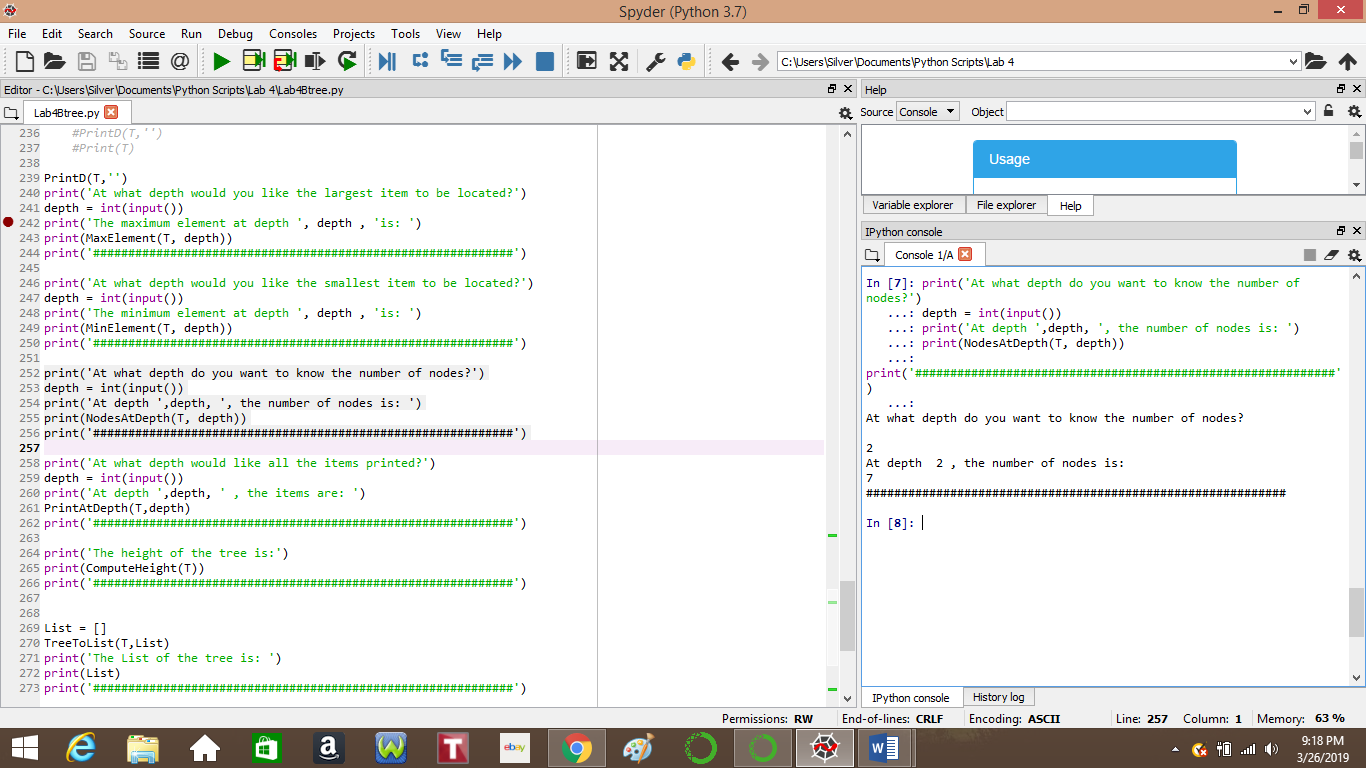
**Minimum and Maximum**

The third and fourth tasks were to find the minimum element and maximum element of the b-tree. The methods for these two tasks are nearly identical, with the only difference being the indices of the child node. Both methods recursively go through the tree until a leaf is reached. Minimum uses 0, to reach the very first child, while maximum uses -1, to reach the last child. Once a leaf is reached, the element at index 0 is reached for the minimum, and index -1 is used for maximum.



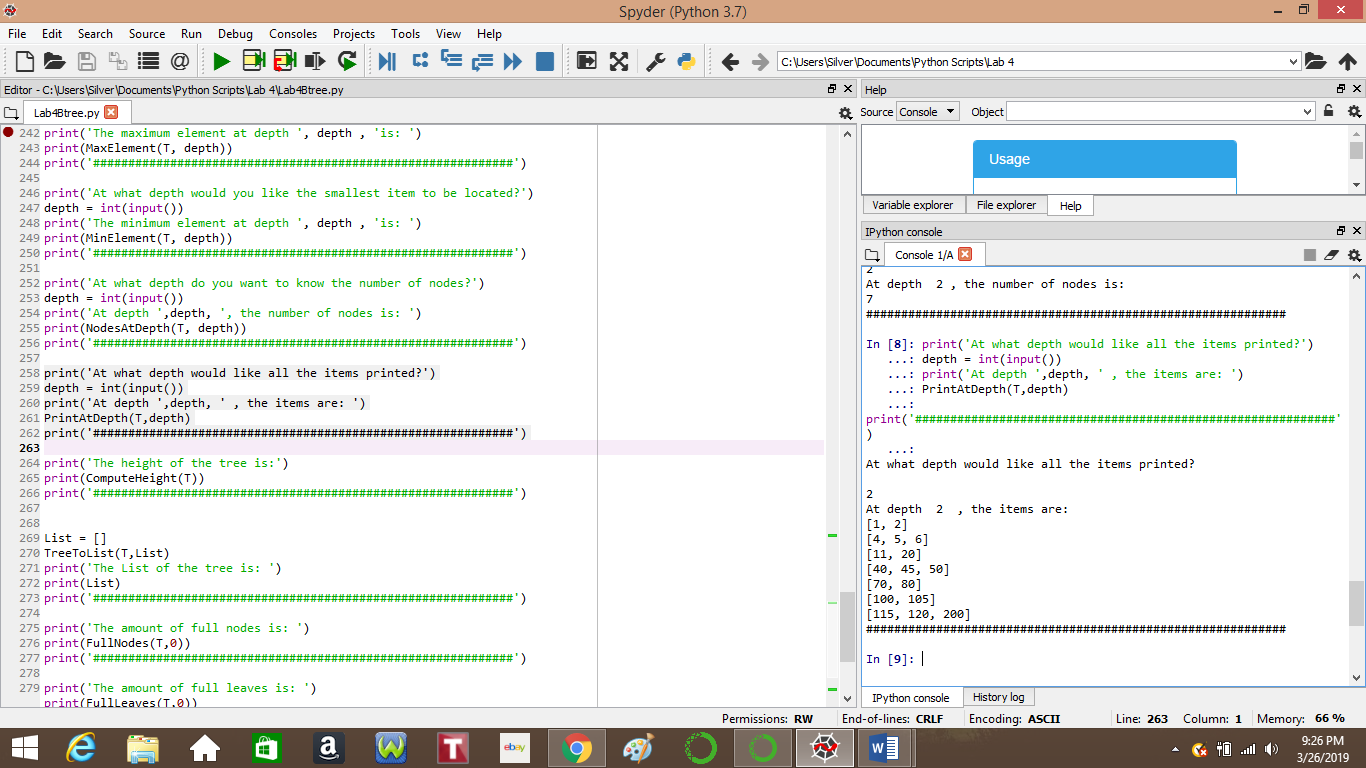
**Nodes at a Given Depth**

Our fifth task was to determine the number of nodes at a given depth. The user inputs a desired depth and the method recursively goes through the tree until the level above the desired depth is reached. The length of the child array is returned and added with the length of other children at that level.



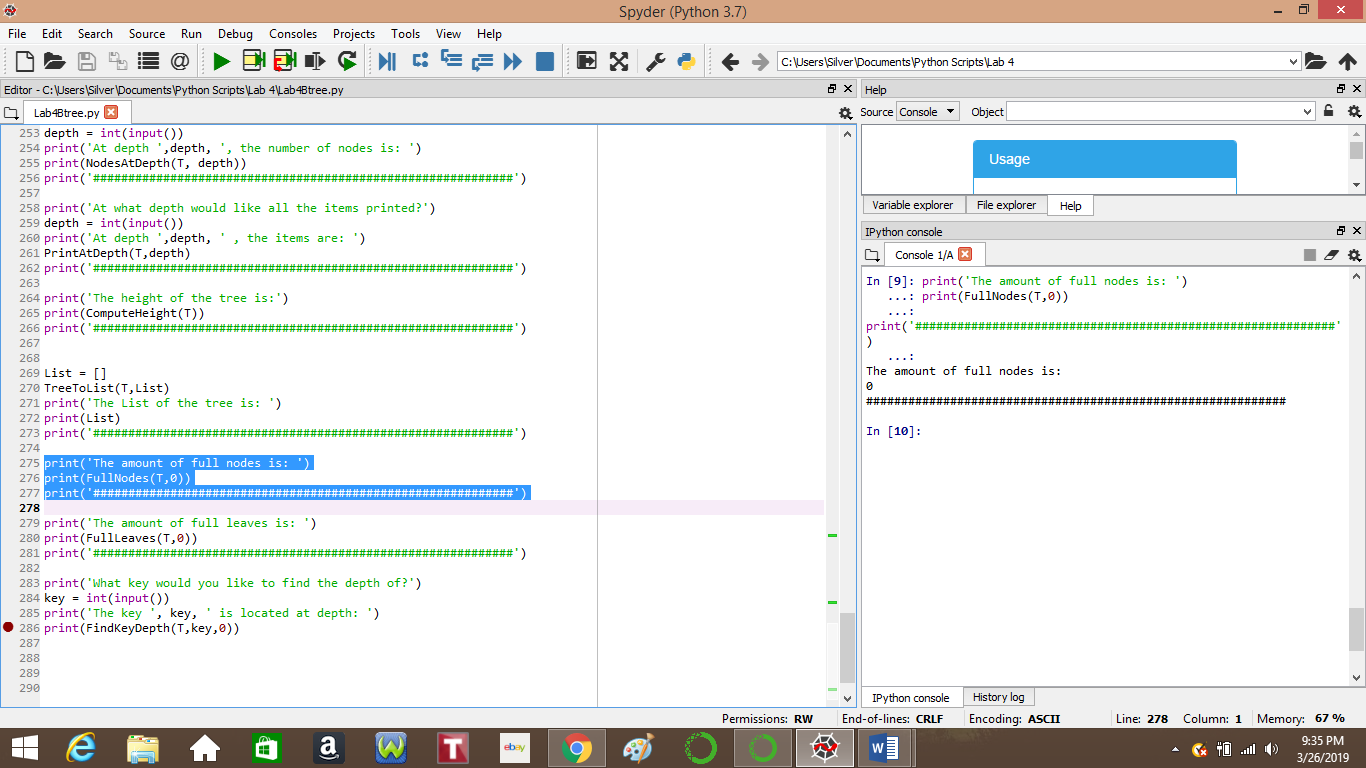
**Print at Depth**

For the sixth part of our assignment, we were required to print out the elements at a given depth. The method uses recursion to reach the level above the desired depth, and prints out the desired elements by the use of a for loop that uses the length of the parent to iterate through each child.



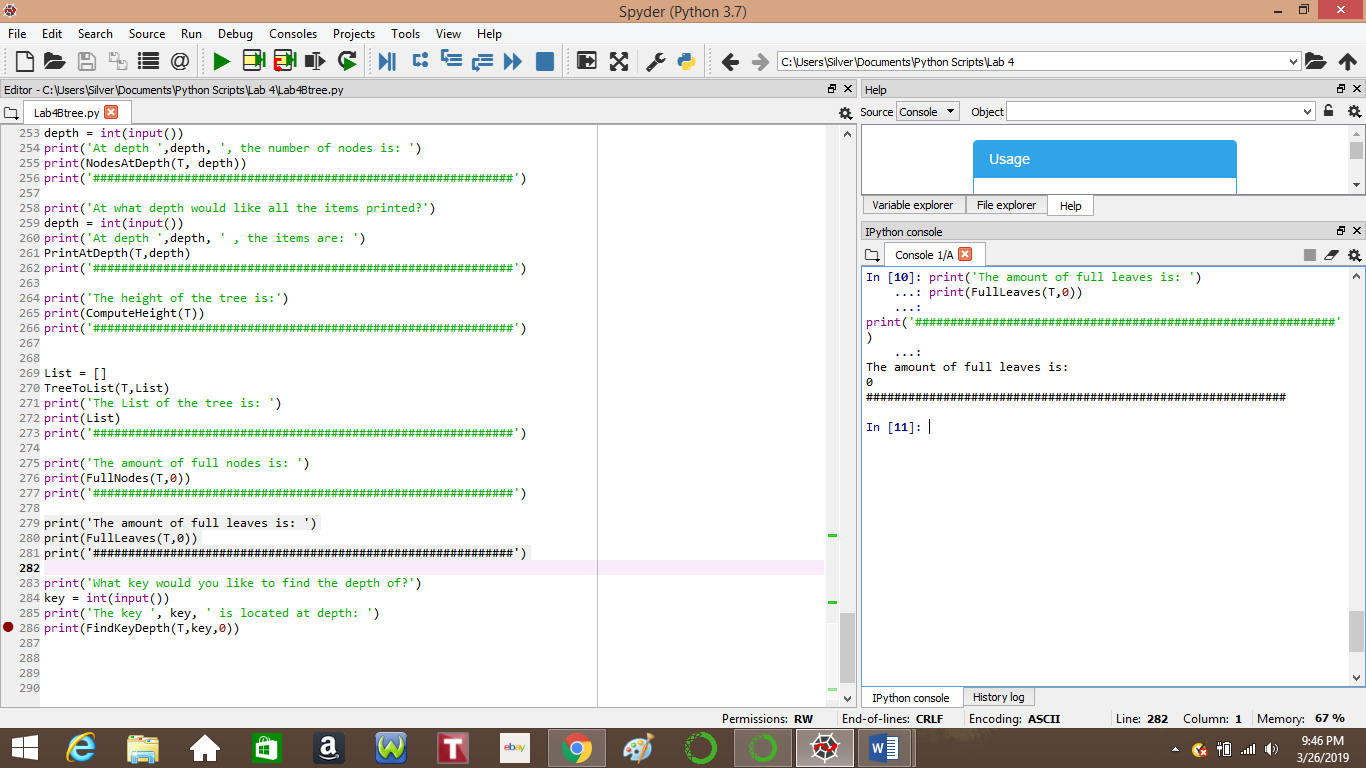
**Full Nodes**

Our seventh task was to return the number of nodes in a b-tree that are full. In order to accomplish this, I used both a for loop and recursion. The for loop iterated through the child of every node while keeping track of the number of full nodes, while the recursion compared the length of the current node to determine whether or not it was full. If the leaf was full, one was added to the count. Once a leaf was reached, the method returned the count and the loop continued to iterate through the b-tree until all nodes were compared. Although the tree that I used to test had no full nodes, I modified the method to check for nodes of 3 elements, 2 elements and 1 element in order to ensure that the method worked, which it did.



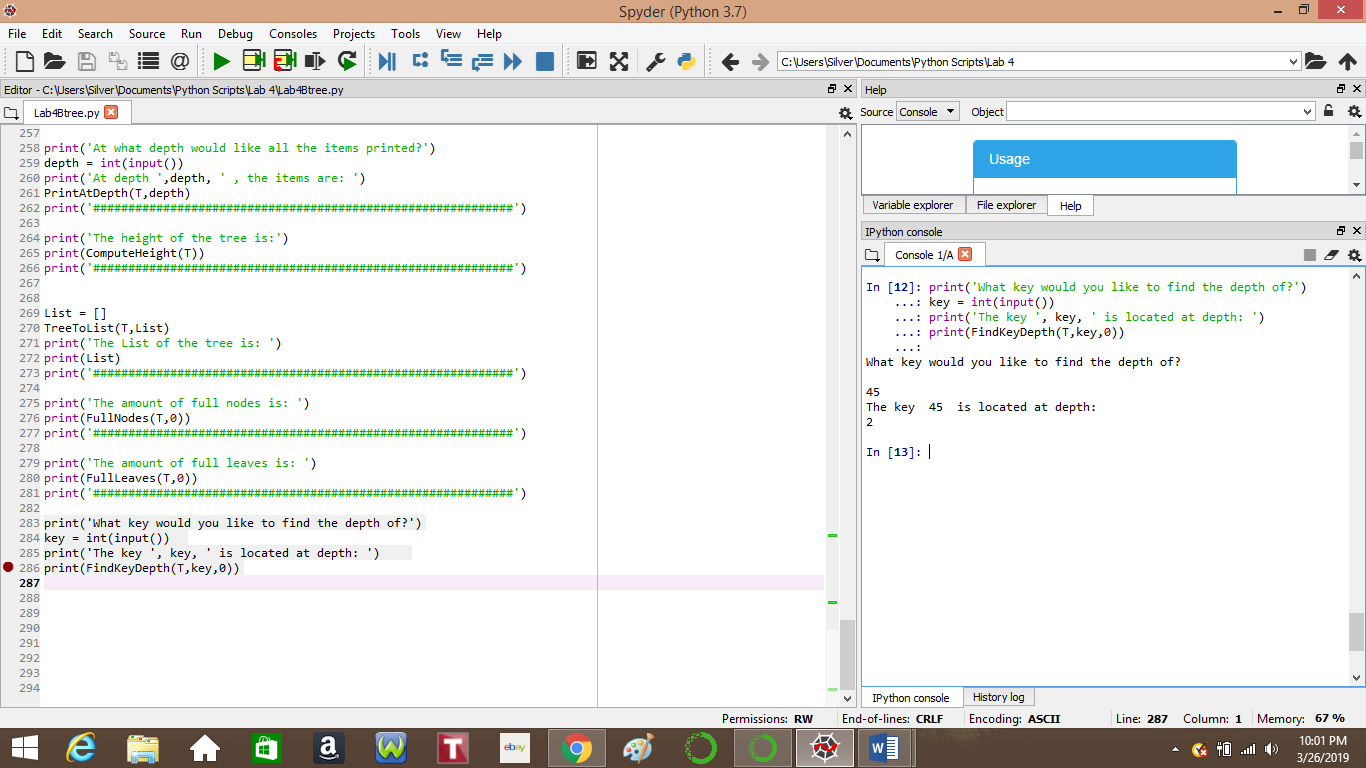
**Full Leaves**

Our eighth task was to determine the number of leaves in ab-tree that are full. Much like the Full nodes method, I employed the use of both a for loop and recursion, but instead of checking each node, leaves were the only nodes that were checked. Again, if the node was full, the 1 was added to the count and then returned. Once al leaves had been reached and compared, the overall count was returned. Again, the b-tree I used had no full leaves, but I tested for nodes with 3, 2, and 1 element, which returned the correct number of nodes.



**Depth of a Key**

For our final task, we had to determine the depth of a given number or “key” in a b-tree, and if the key was not within the tree, the method was to return -1. In order to keep track of the depth, I used a variable in the parameters of the method and set it to 0 in the original call. From there, the method recursively goes through the tree using the given FindChild method to determine which child to move to, given the key. At each node visited, the list is checked for the key. If a leaf is reached before the key is found, then the key is not present in the tree and -1 is returned. If the key is located, the method returns the depth, variable d.



**Conclusion**

Overall, I found this lab assignment to be challenging but I was able to complete it. It did take a lot of work and testing, but I was able to learn how to locate different elements within a b-tree as well as different nodes. I find the b-tree to be more efficient than a binary tree, but in turn, it is more complicated and takes more understanding of the structure to properly utilize it.

**Academic 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.

Signed, Luis Renteria.

**Appendix**

Below is the code Lab 4 was based on, as given to us by Dr. Fuentes on class website.

# Code to implement a B-tree

# Programmed by Olac Fuentes

# Last modified February 28, 2019

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):

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)

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####################################')

SearchAndPrint(T,60)

SearchAndPrint(T,200)

SearchAndPrint(T,25)

SearchAndPrint(T,20)

print(height(T))