For lab 4 we had to use B-trees. B-trees is a self balancing tree data structure that maintains sorted data and allows searches and sequential access in logarithmic time. The B-tree are similar to a binary tree in that a its made up of nodes and has a root node. But unlike binary trees B-trees contain a list of elements in each node instead of one like the binary tree. In this lab I will have to create 9 function that deal with B-trees. They are the height, extract and sort into sorted list,return the minimum element a certain depth,return the maximum element a certain depth, return the number of nodes in the tree a certain depth, print all the items in a certain depth, return the number of nodes that re full, return the number of leaves that are full, and given a key k find the depth of it.

The first function I had to create was to find the height of a B-tree. This was an easy one as the height function given to us in the B-tree.py file the function height() was already created. This function works by passing T that is a reference to the root of the tree. We start by checking if T is a leaf and if it is we return 0, otherwise if T is not a leaf we return 1 plus the recursive call to the same method with T.child[0]. This will keep going to each level of the self balanced tree and increase the number we are returning by 1 until we hit the leaves which are the lowest level of the tree.

The second function is to extract the items in a B-tree and insert them into a sorted list. For this i created BTreeToSortedList() function that takes T and L as a parameter. T is the reference to the root of the tree and L is a blank python list. The way the function works is by traversing the left side of the tree and then right side of the tree. We then append each item in each node using a for loop into the empty list until we hit the leaf nodes. At that point we append the items in the leaf and we return the newly created sorted list. This was a revise way of writing the print function that traverses the whole tree but instead of printing we appended the item into an empty list.

The next function is to return the minimum and maximum element in the tree at a certain depth. Just like a binary tree the smallest element in the tree is found in the far left side of the tree for each given level of the tree and the maximum on the far right side of the ree at a given level. So for this I created the function MinimumAtDepthD() and MaximumAtDepthD(). They both take T and ‘d’ as parameter as T is the reference to the root node and ‘d’ is the depth we want. Both these function work the same but one traverse the left side of the tree while the other one traverses the right side of the tree. They way they wor is if we want the minimum then we check if d is equal to 0 if it is we return the T.item[0] the first item in the node. Otherwise we check if T is a leaf if it is it means that the depth we entered is greater than the height of the tree. If that is not the case if we are still not in our correct depth we recursively called the method inputting T.child[0] and subtracting 1 from ‘d’ until we reach or correct depth. Similarly working for the maximum we do the same but if d is equal to o we return T.child[-1] the last position of the list in the node and traverse the tree to the right using T.child[-1].

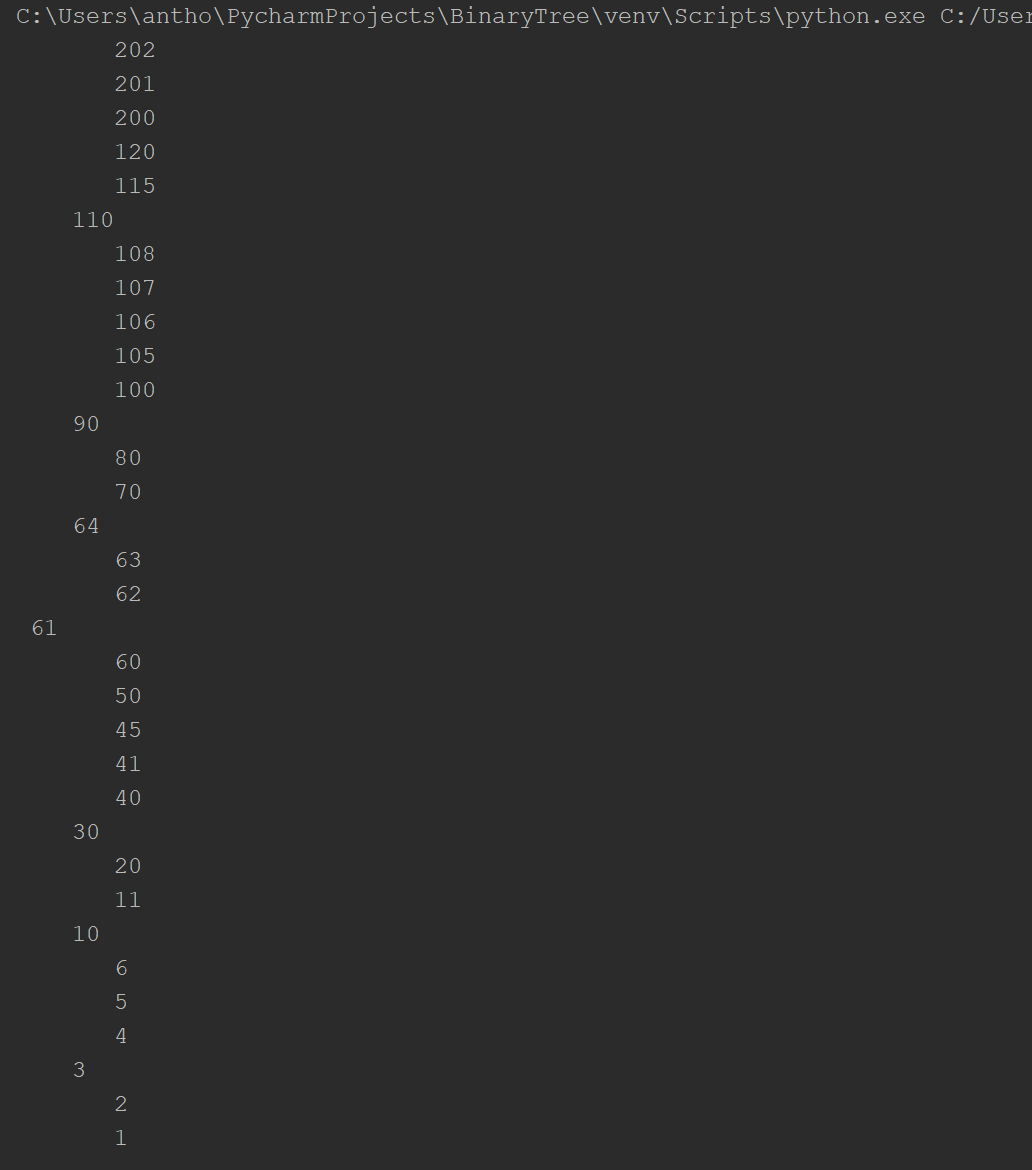
The next function is to return the number of nodes at a given depth. This was a bit of a challenge as we have to traverse the tree and just count the number of nodes when we reach the given depth that we want. For this I created a function called NumberOfNodesD() that takes T and ‘d’ as references. T is the reference to the root node and ‘d’ is the depth that we want. I start by creating a counter and initialize it to 0. I then created a variable ‘h’ that would contain the height of my tree to use as a stoping point when I reach the depth that I want. After that I check if d is greater than my hight if so i just return nothing, otherwise i check if ‘d’ is equal to 0 if it is I return 0. If its not equal to 0 i traverse by tree to the left and right side until i reach the depth that i want and increase my counter. In the end i return the counter and that is the number of nodes in that given depth.

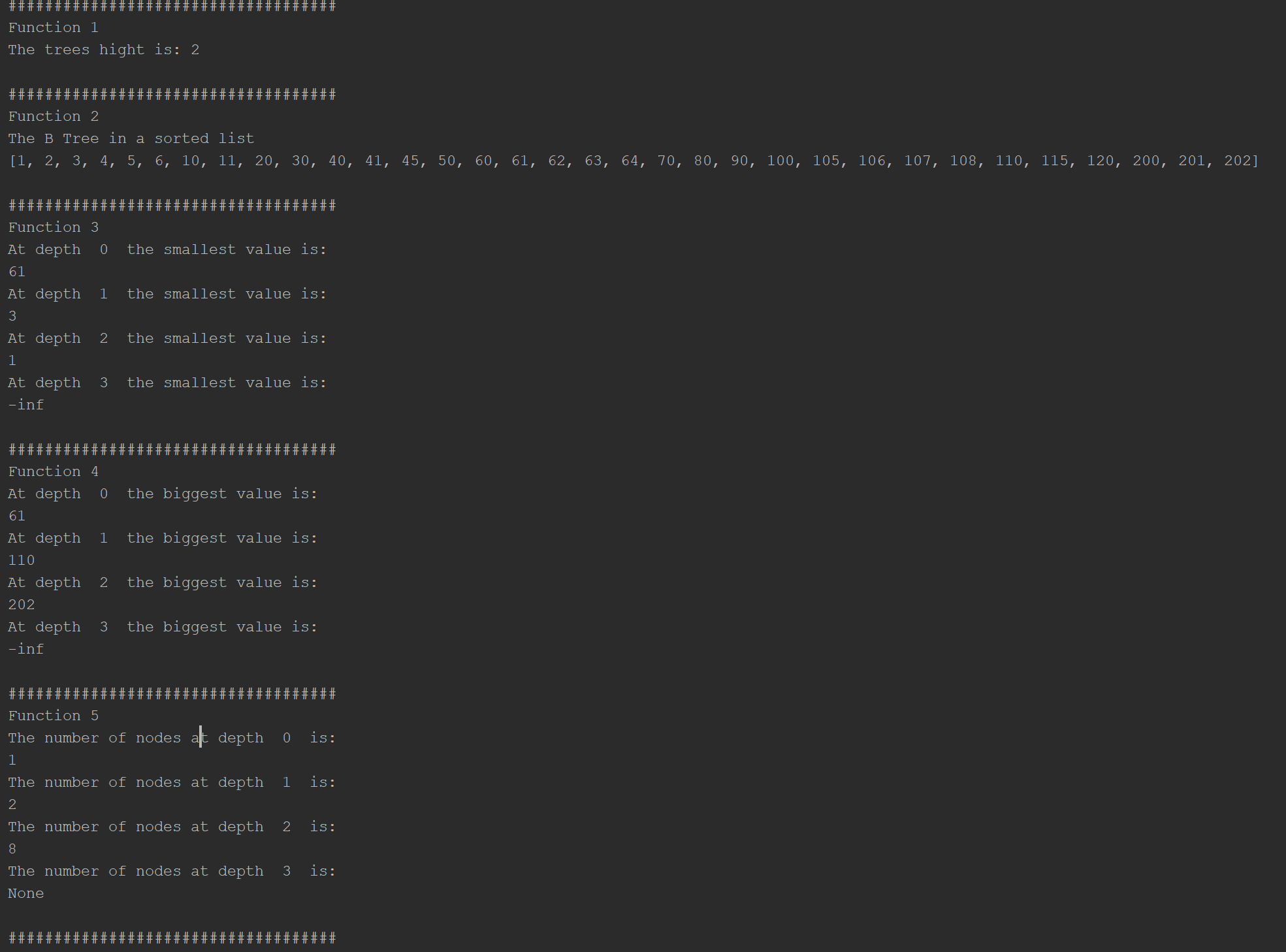
Next I had to print all the items in a given depth. For this one I use the same structure as the printing method again with some modifications. I created the function called PrintDepthD() that passes T and ‘d’ as refresens. T is the reference to the root node and ‘d’ is the given depth. Fist im checking if ‘d’ is equal to zero and if it is i print all the items in the node. If it's not I check if T that is not a leaf and traverse the tree to the left and right subtracting 1 from ‘d’. This will print all the items at any given depth.

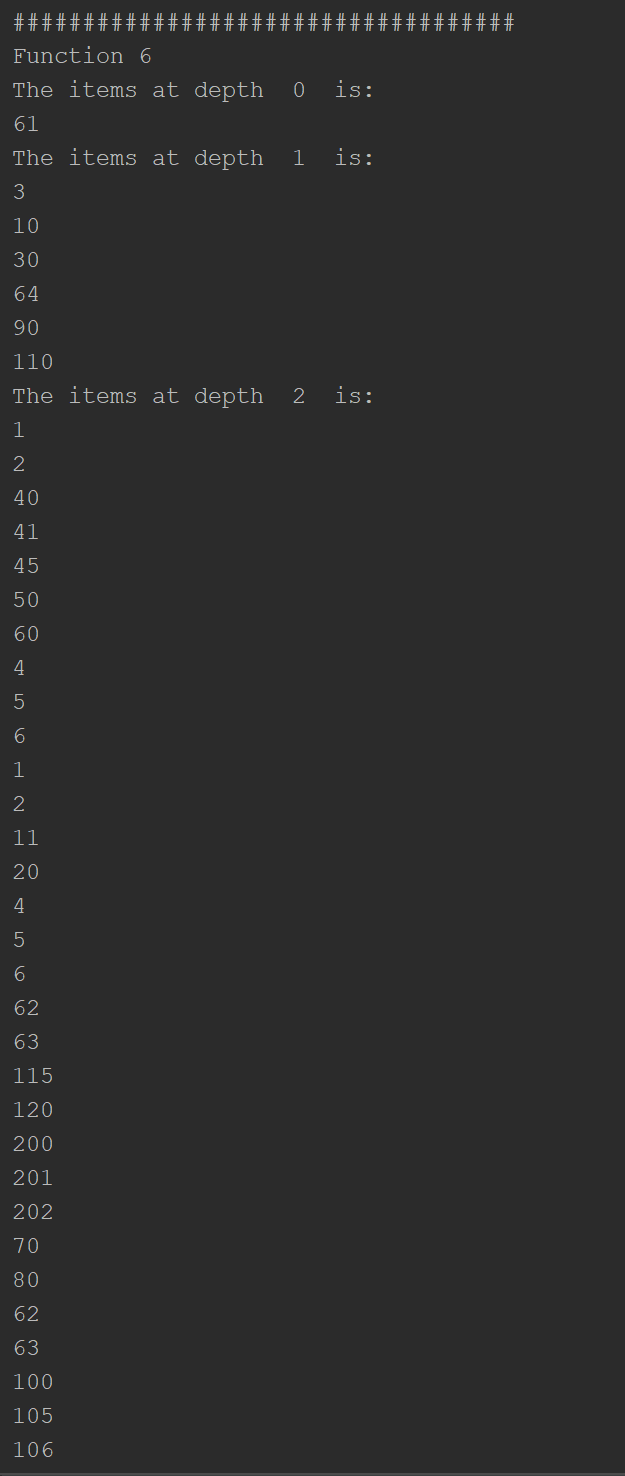
Next I had to return the number of nodes in the tree that are full and the number of leaves that are full. For the first one I created FullNode() that takes T. T is the reference to the root node. I started by creating a counter and initializing it to 0. Then I check if T is not leaf if it’s not then using a for loop I traverse all the childes for T and recursively call them into FullNode(). After that i check if length of T.item for each child is equal to the number of max items if it is i add 1 to counter. In the end i return my counter. So basically i'm traversin the tree and checking if the list in each node is full or not. Similarly to check if the leaves are full i do the same but i add another check to see if T is also a leaf before increasing the counter by 1.

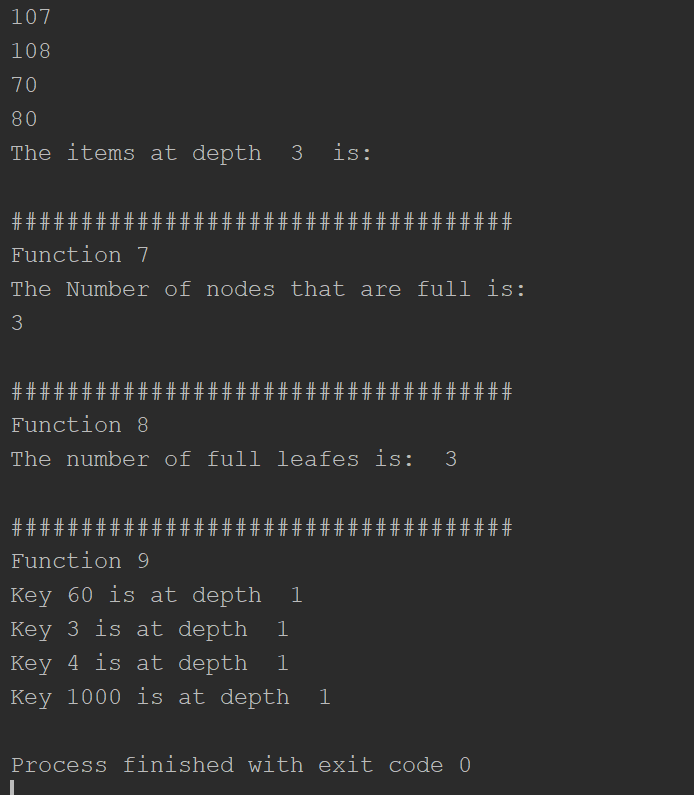
Lastly given a key k I had to return the depth at which it is found, if it wasn’t found then return -1. This was the hardest method for me as I had a hard time figuring it out. In the end i wasn’t able to make it work. The way I try to solve it was by creating a function called FindDepth() that takes T a reference to the root node and ‘k’ the item im looking for. For this i created a counter to increase to find the depth. I did this by first checking if T is not a leaf if so then using a for loop in range of T.child I increase before recursively calling all the children into the function. After that i checked if k was in T.items if it was I returned what count was other while i would subtract 1 from my count and return. IF after all this i did not find ‘k’ i would return -1. After trying to test my function all i would get was -1 return to me if i found k or not.

In conclusion i feel more acostume to using B-trees. It was my first time using B-trees in python and i felt like it was a lot easier to implement than in java. The traversal of the B-tree was the hardest part I believe for this lab as it was important to know how to traverse it, but I feel like I understand it more now after this lab. Hopefully after more practise I will become more acostume to it and be able to use it more efficiently.









# Code to implement a B-tree

# Programmed by Anthony Herrera

# Last modified March 2412%7, 2019

import math

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)

def BTreeToSortedList(T,L):

if T.isLeaf:

for t in T.item:

L.append(t)

else:

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

BTreeToSortedList(T.child[i],L)

L.append(T.item[i])

BTreeToSortedList(T.child[len(T.item)],L)

return L

def MinimumAtDepthD(T,d):

if d==0:

return T.item[0]

if T.isLeaf:

return -math.inf

else:

return MinimumAtDepthD(T.child[0],d-1)

def MaximumAtDepthD(T,d):

if d == 0:

return T.item[-1]

if T.isLeaf:

return -math.inf

else:

return MaximumAtDepthD(T.child[-1], d - 1)

def NumberOFNodesD(T,d):

count = 0

h = height(T)

if d >h:

return

if d == 0:

return 1

if T.isLeaf:

return 1

else:

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

count += NumberOFNodesD(T.child[i],d-1)

return count + NumberOFNodesD(T.child[-1],d-1)

def PrintAtDepthD(T,d):

if d == 0:

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

print(T.item[x])

else:

if T.isLeaf != True:

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

PrintAtDepthD(T.child[i],d-1)

PrintAtDepthD(T.child[i-1],d-1)

def FullNodes(T):

count = 0

if not T.isLeaf:

for x in T.child:

count += FullNodes(x)

if len(T.item) == T.max\_items:

count += 1

return count

def FindDepth(T,k):

count = 0

if not T.isLeaf:

for x in T.child:

count += 1

return FullNodes(x)

if k in T.items:

return count

else:

count -= 1

return -1

def FullLeaves(T):

count = 0

if not T.isLeaf:

for x in T.child:

count += FullNodes(x)

if len(T.item) == T.max\_items and T.isLeaf:

count += 1

return count

L = [30, 50, 10, 20, 60,61,62,63,64, 70, 100, 40, 90, 80, 110, 120, 1, 11, 3, 4, 5, 105, 115, 200, 2, 45, 6,201,202,106,107,108,41]

T = BTree()

for i in L:

#print('Inserting', i)

Insert(T, i)

# Print(T)

PrintD(T, '')

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

print('Function 1')

print('The trees hight is:',height(T))

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

print('Function 2')

L = []

L = BTreeToSortedList(T,L)

print('The B Tree in a sorted list')

print(L)

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

print('Function 3')

for x in range(4):

print('At depth ',x,' the smallest value is:')

print(MinimumAtDepthD(T,x))

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

print('Function 4')

for x in range(4):

print('At depth ',x,' the biggest value is:')

print(MaximumAtDepthD(T,x))

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

print('Function 5')

for x in range(4):

print('The number of nodes at depth ',x,' is:')

print(NumberOFNodesD(T,x))

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

print('Function 6')

for x in range(4):

print('The items at depth ',x,' is:')

PrintAtDepthD(T,x)

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

print('Function 7')

print('The Number of nodes that are full is:')

print(FullNodes(T))

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

print('Function 8')

print('The number of full leafes is: ',FullLeaves(T))

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

print('Function 9')

print('Key 60 is at depth ',FindDepth(T,60))

print('Key 3 is at depth ',FindDepth(T,3))

print('Key 4 is at depth ',FindDepth(T,4))

print('Key 1000 is at depth ',FindDepth(T,1000))

Academic dishonesty includes but is not limited to cheating, plagiarism and collusion. Cheating may involve copying from or providing information to another student, possessing unauthorized materials during a test, or falsifying data (for example program outputs) in laboratory reports. Plagiarism occurs when someone represents the work or ideas of another person as his/her own. Collusion involves collaborating with another person to commit an academically dishonest act. Professors are required to - and will - report academic dishonesty and any other violation of the Standards of Conduct to the Dean of Students.

* Anthony Herrera