Introduction: The task of this lab is to add more functionality to the program other than the basic B-tree operations such as insertion, search and display.

Problem 1:

Proposed solution design and implementation:

Most of the design was already provided, therefore, only some testing with different lengths of lists was performed.

Experimental Results:

Test: L = []

Result: Returned zero

Test: L = [30, 50, 10, 20, 60, 70]

Result: Returned 1

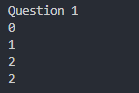
Test: L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11, 3, 4, 5, 105, 115, 200]

Result: Returned 2

Test: L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11, 3, 4, 5, 105, 115, 2, 45, 46, 47]

Result: Returned 2

*Console output*:



Problem 2:

Proposed solution design and implementation:

Came up with a simple algorithm that traverses through the Tree through the access of each child branch with a for loop to iterate through each index. Then it recursively calls the right branches.

Additionally, if the Tree is empty then it should do nothing and if the traversal reaches the leaves of the Tree then it just adds the leaf’s values to the array being built.

Experimental Results:

Test: L = []

Result: Returned [10, 20, 30, 50, 60]

Test: L = [30, 50, 10, 20, 60, 70]

Result: Returned [10, 20, 30, 50, 60, 70]

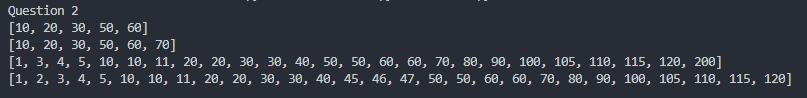
Test: L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11, 3, 4, 5, 105, 115, 200]

Result: Returned [1, 3, 4, 5, 10, 10, 11, 20, 20, 30, 30, 40, 50, 50, 60, 60, 70, 80, 90, 100, 105, 110, 115, 120, 200]

Test: L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11, 3, 4, 5, 105, 115, 2, 45, 46, 47]

Result: Returned [1, 2, 3, 4, 5, 10, 10, 11, 20, 20, 30, 30, 40, 45, 46, 47, 50, 50, 60, 60, 70, 80, 90, 100, 105, 110, 115, 120]

*Console output*:



Problem 3:

Proposed solution design and implementation:

Thought about checking the height of the Tree versus the depth levels to search the minimum. If the level of depth to search is greater than the height of the Tree, then the function returns -1 since the depth level doesn’t exist. But if the depth level is zero, it returns the current depth leftmost node value. Else it recurses through the tree calling the leftmost child, which it eventually reaches the minimum value.

Experimental Results:

Test: L = [], depth = 1

Result: Returned -1 since List is empty

Test: L = [30, 50, 10, 20, 60, 70], depth = 1

Result: Returned 10

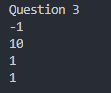
Test: L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11, 3, 4, 5, 105, 115, 200], depth = 2

Result: Returned 1

Test: L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11, 3, 4, 5, 105, 115, 2, 45, 46, 47], depth = 2

Result: Returned 1

*Console output:*



Problem 4:

Proposed solution design and implementation:

This function was very similar to the MinAtDepth function since it obtains the same functionality but only finding the opposite, the maximum node’s value. Similarly checks the height of the Tree and determines if the depth level to search is valid, else it returns -1. When the depth level reaches 0 in recursion, it returns the rightmost item from the node, else it recursively traverses through the Tree and subtracts a depth level.

Experimental Results:

Test: L = [], depth = 1

Result: Returns -1 since List is empty

Test: L = [30, 50, 10, 20, 60, 70]

Result: Returned 70

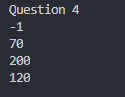
Test: L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11, 3, 4, 5, 105, 115, 200]

Result: Returned 200

Test: L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11, 3, 4, 5, 105, 115, 2, 45, 46, 47]

Result: Returned 120

*Console output*:



Problem 5:

Proposed solution design and implementation:

Came up with a solution to the function by also obtaining the height of the Tree beforehand since any depth level greater than the height of the Tree is impossible to access, so it returns -1. Else, it recursively traverses through the Tree and subtracts one depth level. Once the depth level reaches zero it returns the length of the nodes at the provided depth level.

Experimental Results:

Test: L = [], depth = 1

Result: Returned -1 since list is empty

Test: L = [30, 50, 10, 20, 60, 70], depth = 1

Result: Returned 5

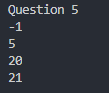
Test: L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11, 3, 4, 5, 105, 115, 200], depth = 2

Result: Returned 20

Test: L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11, 3, 4, 5, 105, 115, 2, 45, 46, 47], depth = 2

Result: Returned 21

*Console output*:



Problem 6:

Proposed solution design and implementation:

Came up with the exact same functionality for the PrintAtDepth from the NodesAtDepth function. However instead of finding out the length of the depth level provided, it prints all the nodes’ values. It checks if the depth level given is valid by comparing it to the height of the Tree, if valid then it recursively traverses through the Tree and until it reaches 0 for the depth level in recursion it prints the current nodes values.

Experimental Results:

Test: L = [], depth = 1

Result: Returned nothing, displayed nothing

Test: L = [30, 50, 10, 20, 60, 70], depth = 1

Result: Returned “10 20 50 60 70”

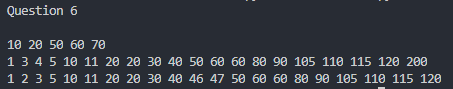
Test: L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11, 3, 4, 5, 105, 115, 200], depth = 2

Result: Returned “1 3 4 5 10 11 20 20 30 40 50 60 60 80 90 105 110 115 120 200”

Test: L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11, 3, 4, 5, 105, 115, 2, 45, 46, 47], depth = 2

Result: Returned “1 2 3 5 10 11 20 20 30 40 46 47 50 60 60 80 90 105 110 115 120”

*Console output*:



Problem 7:

Proposed solution design and implementation:

The method needed to refer back to the constructor’s property of “max\_items” since it needs to compare the length of each node other than the leaf nodes, to see if the node is a full node. However, if the Tree traversal reaches the leaf nodes then it returns 0.

Experimental Results:

Test: L = []

Result: Returned 0

Test: L = [30, 50, 10, 20, 60, 70]

Result: Returned 0

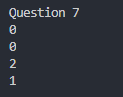
Test: L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11, 3, 4, 5, 105, 115, 200]

Result: Returned 2

Test: L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11, 3, 4, 5, 105, **109**, 115, 2, 45, 46, 47]

Result: Returned 1

*Console output*:



Problem 8:

Proposed solution design and implementation:

Came up with a similar approach to the FullNodes function, except it only compares the length of each leaf node once it reaches that depth level. Else it recursively traverses through the Tree up the leaf nodes.

Experimental Results:

Test: L = []

Result: Returned 0

Test: L = [30, 50, 10, 20, 60, 70]

Result: Returned 0

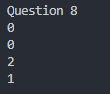
Test: L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11, 3, 4, 5, 105, 115, 200]

Result: Returned 2

Test: L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11, 3, 4, 5, 105, **109**, 115, 2, 45, 46, 47]

Result: Returned 1

*Console output*:



Appendix:

*# Course: 2302-001*

*# Author: Esteban Retana*

*# Assignment:*

*# Instructor: Olac Fuentes*

*# TA: Anindita Nath*

*# Date of last modification:3/27/19*

*# Purpose:*

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)

*# Converts BTree to sorted array list*

def BTreeToSorted(T):

if T == None:

return

t = []

if T.isLeaf:

return T.item

else:

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

t += BTreeToSorted(T.child[i]) + [T.item[i]]

t += BTreeToSorted(T.child[len(T.item)])

return t

*# Finds out the minimum node at a depth level*

def MinAtDepth(T,k):

*# Find out if given depth is possible to search*

h = height(T)

if k > h:

return -1

*# Returns smallest item in depth*

if k == 0:

return T.item[0]

else:

return MinAtDepth(T.child[0],k-1)

*# Finds out the max value from a certain depth level*

def MaxAtDepth(T,k):

*# Find out if given depth is possible to search*

h = height(T)

if k > h:

return -1

*# Returns largest item in depth*

if k == 0:

return T.item[len(T.item)-1]

else:

return MaxAtDepth(T.child[len(T.item)],k-1)

*# Determiness the amount of nodes at a depth level*

def NodesAtDepth(T,k):

t = 0

*# Find out if given depth is possible to search*

h = height(T)

if k > h:

return -1

*# Stop at one depth before the given value to sum the lengths of all childs*

if k >= 1:

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

t += NodesAtDepth(T.child[i], k-1)

t += NodesAtDepth(T.child[len(T.item)],k-1)

*# Returns the length of node*

if k == 0:

return len(T.item)

return t

*# Prints all nodes values at a certain depth level*

def PrintAtDepth(T,k):

h = height(T)

if k > h:

return

if k >= 1:

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

PrintAtDepth(T.child[i],k-1)

PrintAtDepth(T.child[len(T.item)],k-1)

if k == 0:

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

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

*# Finds out the total number of full nodes*

def FullNodes(T):

t = 0

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

return 1

if T.isLeaf:

return 0

else:

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

t += FullNodes(T.child[i])

t += FullNodes(T.child[len(T.item)])

return t

*# Finds out the toal number of full leaf nodes*

def FullLeafs(T):

t = 0

if T.isLeaf:

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

return 1

else:

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

t += FullLeafs(T.child[i])

t += FullLeafs(T.child[len(T.item)])

return t

*# def FindDepth(T,k):*

*# if*

*# L = []*

*# for i in range(100):*

*# L[i] = i + 1*

*# print(L)*

*# No full nodes*

M = [30, 50, 10, 20, 60, 70]

*# Top full node*

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

*# full leaft*

O = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11 , 3, 4, 5, 105, 109, 115, 2, 45, 46, 47]

T = BTree([])

U = BTree()

V = BTree()

W = BTree()

*# for i in L:*

*# Insert(T,i)*

for i in M:

*# print('Inserting',i)*

Insert(U,i)

*# PrintD(W,'')*

*# Print(T)*

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

for i in N:

*# print('Inserting',i)*

Insert(V,i)

*# PrintD(W,'')*

*# Print(T)*

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

for i in O:

*# print('Inserting',i)*

Insert(W,i)

*# PrintD(W,'')*

*# Print(T)*

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

*# SearchAndPrint(T,60)*

*# SearchAndPrint(T,200)*

*# SearchAndPrint(T,25)*

*# SearchAndPrint(T,20)*

*# Question 1*

print("Question 1")

print(height(T))

print(height(U))

print(height(V))

print(height(W))

*# Question 2*

print("Question 2")

print(BTreeToSorted(T))

print(BTreeToSorted(U))

print(BTreeToSorted(V))

print(BTreeToSorted(W))

*# Question 3*

print("Question 3")

print(MinAtDepth(T,1))

print(MinAtDepth(U,1))

print(MinAtDepth(V,2))

print(MinAtDepth(W,2))

*# Question 4*

print("Question 4")

print(MaxAtDepth(T,1))

print(MaxAtDepth(U,1))

print(MaxAtDepth(V,2))

print(MaxAtDepth(W,2))

*# Question 5*

print("Question 5")

print(NodesAtDepth(T,1))

print(NodesAtDepth(U,1))

print(NodesAtDepth(V,2))

print(NodesAtDepth(W,2))

*# Question 6*

print("Question 6")

PrintAtDepth(T,1)

print()

PrintAtDepth(U,1)

print()

PrintAtDepth(V,2)

print()

PrintAtDepth(W,2)

*# Question 7*

print("Question 7")

print(FullNodes(T))

print(FullNodes(U))

print(FullNodes(V))

print(FullNodes(W))

*# Question 8*

print("Question 8")

print(FullLeafs(T))

print(FullLeafs(U))

print(FullLeafs(V))

print(FullLeafs(W))

*# Question 9*

*# print("Question 9")*

Academic Honesty:

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.