Lab 4 Adrian Monreal

In this lab Dr. Fuentes required us to implement what we know about b trees, their functions and how to navigate them, he assigned us to do 9 different operations 1. Compute the height of the tree. 2. Extract the items in the B-tree into a sorted list.3. Return the minimum element in the tree at a given depth d. 4. Return the maximum element in the tree at a given depth d.5. Return the number of nodes in the tree at a given depth d.6. Print all the items in the tree at a given depth d.7. Return the number of nodes in the tree that are full. 8. Return the number of leaves in the tree that are full.9. Given a key k, return the depth at which it is found in the tree, of -1 if k is not in the tree. All but 1 of these functions were easy only one of them I could not finish, the one where we were required to extract the items in the b-tree to a sorted list what was confusing me was how to create a list and store it with values from another function basically carrying the list thorough the recursive function.

For #1 I understood the logic as the tree is going to be of equal height through all of it so I created a recursive function to add one every time there is a child and return the sum at the end computing the height.

#2 I couldn’t understand how to carry the list when calling the recursive function if a lists is a local variable It can’t traverse.

#3 the minimum element will always be in the smallest child of the smallest node, it’s a recursive method that first checks if it is a leaf then if it’s not, it will call the function again with the first child as the node until it reaches a leaf and then prints the first node, which is the smallest.

#4 return the maximum element in a given tree at given depth d. the logic is that the biggest element in this tree will be the last child of the first key, then it’s the biggest child of the next key or node, it will stop when the depth reaches 0 which is the desired depth, each recursive call will decrease the depth by 1

#5 return the number of nodes at a given depth d. The logic is the same as the max and min except, it will travel to all the children at the given depth, it counts the nodes as it traverses the function, it will stop when the depth reaches 0 which is the desired depth, each recursive call will decrease the depth by 1.

#6 Print all the items in the tree at a given depth d. The logic is the same as the max and min except, it will travel to all the children at the given depth, instead of returning the number of nodes at that depth it will print them in order, it will stop when the depth reaches 0 which is the desired depth. Each recursive call will decrease the depth by 1.

#7Return the number of Nodes in the tree that are full the logic behind this one is that it will travel all the way to the leaf then work its way back up, once it reaches a leaf it returns 0 signifying to go back up one level. test each node to see which are full.

#8 Return the number of leaves in the tree that are full, just like number 7 it travels all the way to the leaves first. once it travels to each leaf it checks if its full if so, it returns a 1 if not return 0. This should count up all full leaves.

#9. Given a key k, return the depth at which it is found in the tree, of -1 if k is not in the tree. This is a method similar to the search method except it would want to return the depth, basically it searches each node for it recursively counting the depth as it makes its calls. If the k is not found it will return a -1

#Lab 4 BTrees  
#Adrian Monreal  
#80570881  
#Olac fuentes Cs2302  
  
  
# 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,' ')  
 else:  
 for i in range(len(T.item)):  
 Print(T.child[i])  
 print(T.item[i], ' ' )  
 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', ' ')  
 print('node contents:', node.item)  
  
  
#1 compute the height of the tree  
#I didnt realize that this code was given to us  
#However the logic is simple since the tree  
#is like a pyramid the base is the same so traverse all the way to a leaf  
#and count how many times you went down that is the height of the tree  
def HeightOfTree(T):  
 if T.isLeaf:  
 return 0  
 else:  
 return 1 + HeightOfTree(T.child[0])  
  
#2. Extract the items in the B-tree into a sorted list.  
  
def BTreeIntoSortedList(T):  
 if T.isLeaf:  
 for i in range(len(T.items)):  
 return T.items[i]+ ' '  
 else:  
 for i in range(len(T.child)):  
 return BTreeIntoSortedList(T.child[i]) + " " + T.items[i]  
  
  
  
#sub function of #2 splits what was returned into a list  
  
#3 return the minimum element in a given tree at given depth d  
# the logic is that the smallest element in this tree will be the first child of the key  
#then its the smallest child of the next key or node  
# it will stop when the depth reaches 0 which is the desired depth  
#each recursive call will decrease the depth by 1  
def MinElementAtDepth(T,d):  
 if T.isLeaf:  
 return None  
 elif d == 0:  
 return T.item[0]  
 else:  
 return MinElementAtDepth(T.child[0],d-1)  
  
#4 return the maximum element in a given tree at given depth d  
# the logic is that the biggest element in this tree will be the last child of the first key  
#then its the biggest child of the next key or node  
# it will stop when the depth reaches 0 which is the desired depth  
#each recursive call will decrease the depth by 1  
def MaxElementAtDepth(T,d):  
 if T.isLeaf and d == 0:  
 return None  
 elif d == 0:  
 return T.item[len(T.item)-1]  
 else:  
 return MaxElementAtDepth(T.child[len(T.item)],d-1)  
  
#5 return the number of nodes at a given depth d  
#the logic is the same as the max and min exccept  
#it will travel to all the children at the given depth  
#it counts the nodes as it traverses the function  
# it will stop when the depth reaches 0 which is the desired depth  
#each recursive call will decrease the depth by 1  
  
def NumNodesAtDepth(T,d):  
 if T.isLeaf:  
 return None  
 elif d == 0:  
 return len(T.item)  
 else:  
 TotalNodes = 0  
 for i in range(len(T.child)):  
 TotalNodes += NumNodesAtDepth(T.child[i],d-1)  
 return TotalNodes  
  
#6Print all the items in the tree at a given depth d.  
#the logic is the same as the max and min exccept  
#it will travel to all the children at the given depth  
# instead of returning the number of nodes at that depth  
#it will print them in order  
# it will stop when the depth reaches 0 which is the desired depth  
#each recursive call will decrease the depth by 1  
def printNodesAtDepth(T,d):  
 if T.isLeaf and d != 0:  
 print("Node is a leaf cannot procede")  
 elif d == 0:  
 for i in range(len(T.item)):  
 print T.item[i]  
 else:  
 for i in range(len(T.child)):  
 printNodesAtDepth(T.child[i],d-1)  
  
  
#7 Return the number of Nodes in the tree that are full  
#the logic behind this one is that it will travel all the way to the leaf  
#then work its way back up  
#once it reaches a leaf it returns 0 signifying to go back up one level  
#test each node to see which are full  
  
def FullNodes(T):  
 if T.isLeaf:  
 return 0  
 else:  
 amount = 0  
 for i in range(len(T.child)):  
 amount += FullNodes(T.child[i])  
 if IsFull(T) == True:  
 amount +=1  
 return amount  
  
#8 Return the number of leaves in the tree that are full.  
#just like number 7 it travels all the way to the leaves first  
#once it travels to each leaf it checks if its full if so  
#return a 1 if not return 0  
#this should count up all full leaves  
  
def fullLeaves(T):  
 if T.isLeaf:  
 if IsFull(T) == True:  
 return 1  
 else:  
 return 0  
 else:  
 amount = 0  
 for i in range(len(T.child)):  
 amount += fullLeaves(T.child[i])  
 return amount  
  
#9. Given a key k, return the depth at which it is found in the tree, of -1 if k is not in the tree.  
#this is a method similar to the search method except it would want to return the depth  
# basically it searches each node for it recursively counting the depth as it makes its calls  
# if the k is not found it will return a -1  
  
def searchForKdepth(T,d,k):  
 if k in T.item:  
 return d  
 if T.isLeaf:  
 return -1  
 return searchForKdepth(T.child[FindChild(T, k)],d+1, k)  
  
  
  
  
  
  
  
  
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))  
print("---------------------------")  
print("#1")  
print(HeightOfTree(T))  
print("---------------------------")  
#2  
print("#3")  
print(MinElementAtDepth(T,1))  
print("---------------------------")  
print("#4")  
print(MaxElementAtDepth(T,2))  
print("---------------------------")  
print("#5")  
print(NumNodesAtDepth(T,1))  
print("---------------------------")  
print("#6")  
printNodesAtDepth(T,3)  
print("---------------------------")  
print("#7")  
print(FullNodes(T))  
print("---------------------------")  
print("#8")  
print(fullLeaves(T))  
print("---------------------------")  
print("#9")  
print(searchForKdepth(T,0,90))

I Adrian Monreal 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.