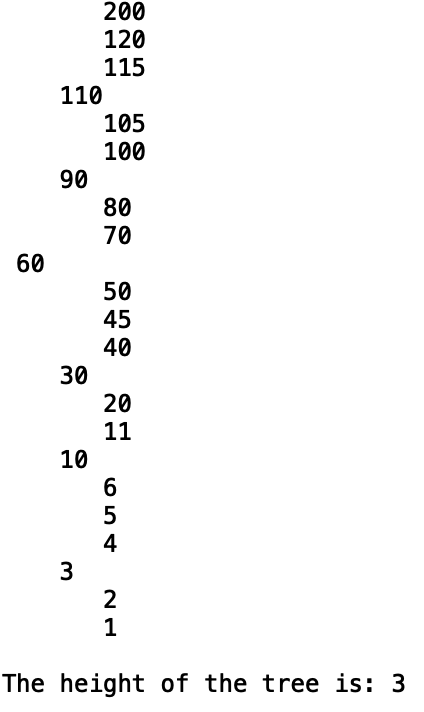
Ericka Najera

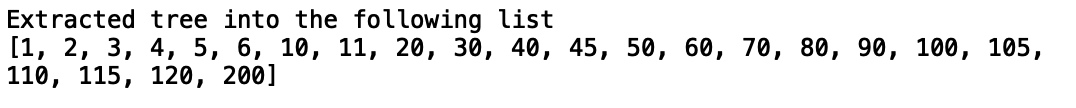
21 March 2019

Lab 4

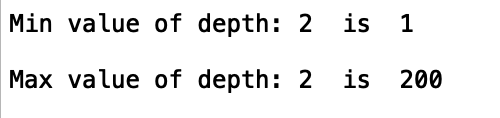
The purpose of lab 4 was to practice the functions of a b tree. We had to practice identifying the depths of the tree as well as traversing over the children arrays. The methods we had to accomplish were get the height of the tree, extract the items of the tree, return the min and max element of the tree, print and count numbers in a certain depth, check if the nodes are full and search the tree for the key.

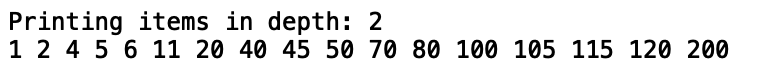
 The height method I thought about counting one each time there was a child. So I did two bases cases for my recursive method. The first base case would be if it was none since we wouldn’t count one if it was empty of it was the end of the tree. If the method was a leaf them it would return one and not send anything else since there was nothing else below that child. Since the b tree moves up instead of down when you insert new keys then we can simply send the first child of each tree. The height is different from the depth, so instead of the depth starting at 0 it would start at one.

The method of extracting the keys from the tree into an array in ascending order was kind of printing in ascending order. I had to do it like last weeks lab method of extracting and send a new list as a parameter. For the base case I thought about checking for if the tree was a leaf. If it was then I would append each item using a for loop. The next case would be do a for loop for all the children and call them while appending it to the list.

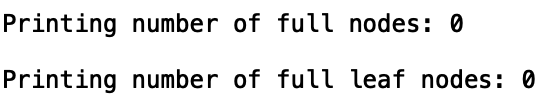


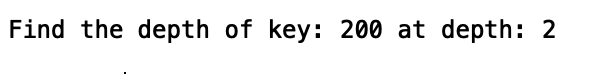
The next two methods would be finding the min and max key of that depth . I thought about for the min just calling the very first child of each parent and for the max calling the last child until there are no more left all while decreasing the depth each time. For both methods I did the same structure. I initiated with a base case of checking if that tree was null, if it was then I would simply return avoiding an error this would also avoid the input being a depth not found. It the depth was a zero then it would mean it reached that depth and we would simply return that first or last item. Also if the depth was not zero but that tree was already at the leaf then the call would go down to depth = -1 and that is invalid so we would simply return. If the method didn’t reach any of those conditions then we would call while decreasing the depth it is in and sending the first or last child depending if you are looking for the max or min item.



 For the methods of counting and printing the nodes at certain of that depth I thought about the structure of the min and max methods. We check the depth for when its zero and if not keep decreasing. Then we also check if the tree is a leaf or null and return if it is. For the module of counting the nodes in the tree we would return 1 if the depth is 0 since we would have reach the desired depth. And if the depth is not zero and the tree is at the leaf then we return zero. If the depth is not zero we decrease it in a for loop with a counter to keep track of those child nodes we are calling. As for the printing we would do a for loop and print the items of the that node when depth is at zero. If the depth is not zero them we would decrease the depth by one and send each child into the recursive call.

The counting full nodes and full leaf nodes go hand and hand. For the full nodes we would check if the length that node is equal to the number of max items which is a method provided in the class. We check if the tree itself is none and return if true. We initiate a counter and check only if that tree is not a leaf. The we proceed with a for loop to traverse the tree. Then we add the recursive call to the counter, if not then the count of the nodes gets lost. After that we compare if the length of the node is equal to the max items number and if it is true then we add one to the counter. As for the full leaf nodes we do the same thing but backwards. We check if the tree itself is none and return if true. We initiate a counter and check only if that tree is a leaf then, compare if the length of the node is equal to the max items number and if it is true then we add one to the counter. The we proceed with a for loop to traverse the tree and do the same thing of adding the recursive call to the counter.



 For the module of searching for the key in the tree and returning the depth. We would also check if the tree is null and if it is then we return. Then we check if the key is in that node and if it is then we return zero. If both of those cases fail then we return -1 because it wasn’t found. Then I did a if statement to check if the key if bigger than the last key in that node the we would do a recursive call and save it in a variable. If that case is false then we would do a for loop and traverse the tree. At the end of the method the variable would either be a zero or -1 one. If it’s a -1 then we return -1 because it not found but if its not zero then we return d +1 which would increment the depth until its found.

|  |  |  |  |
| --- | --- | --- | --- |
| Method | Time in Nanoseconds for each case | | |
|  | depth is 0 | depth in middle | depth is nonexistent |
| Height | ﻿55000 | ﻿37000 | ﻿35000 |
| Extract | ﻿15000 | ﻿9000 | ﻿10000 |
| Min Value | ﻿85000 | ﻿73000 | ﻿73000 |
| Max Value | ﻿81000 | ﻿72000 | ﻿73000 |
| Nodes in depth | ﻿240000 | ﻿117000 | ﻿115000 |
| Print Depth | ﻿25000 | ﻿288000 | ﻿7000 |
| Full Nodes | ﻿﻿27000 | ﻿32000 | ﻿46000 |
| Full Leaf Nodes | ﻿﻿26000 | ﻿37000 | ﻿23000 |
| depth of key | ﻿71000 | ﻿142000 | ﻿75000 |

﻿

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 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 height(T):

#computes the height of the tree which is different from the depth

#base case if the tree is null then it will return zero

if T is None:

return 0

#if T is a leaf then it returns 1 becaause there height of one

if T.isLeaf:

return 1

#else it recursively calls the tree and adds one

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

def extractItems(T,extracted):

#if the root itself is a leaf append all items to the list

if T.isLeaf:

for t in T.item:

extracted.append(t)

else:

#if not call all the child in the t.item and append that t.item to the list

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

extractItems(T.child[i],extracted)

extracted.append(T.item[i])

extractItems(T.child[len(T.item)],extracted)

return extracted

def minAtDepth(T,d):

#base case if the tree is null and there is more depth than tree

if T is None:

return

#if it has reached that depth returns the first element in the list which is the smallest

if d == 0:

return T.item[0]

#if both the depth is not reached but T is a leaf it has reached the end of the tree and depth is nonexistent

if T.isLeaf:

return

#if none ifs those of true them call the first child while decreasing the depth

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

def maxAtDepth(T,d):

#base case if the tree is null and there is more depth than tree

if T is None:

return

#if it has reached that depth returns the last element in the list which is the biggest

if d == 0:

return T.item[-1]

#if both the depth is not reached but T is a leaf it has reached the end of the tree and depth is nonexistent

if T.isLeaf:

return

#if none ifs those of true them call the first child while decreasing the depth

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

def nodesInDepth(T,d):

#base case if the tree is null and there is more depth than tree

if T is None:

return

#if the depth is at 0 then we will return one for that node

if d == 0:

return 1

#if the t is a leaf and d hasnt reached 0 then return

if T.isLeaf:

return 0

else:

#save the count while traversing the child of T and return

count=0

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

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

return count

def PrintAtDepth(T,d):

#base case if the tree is null and there is more depth than tree

if T is None:

return

#if the depth is zero then print all items in T

if d == 0:

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

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

return

#if both cases where not reached then either the tree or depth is null

if T.isLeaf:

return

#else call the child of T with the depth decreasing by 1

else:

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

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

def fullNodes(T):

#base case if the tree is null and there is more depth than tree

if T is None:

return

#initiate a variable to count

count = 0

#if the T is not a leaf traverse the tree

if not T.isLeaf:

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

count += fullNodes(T.child[i])

#if that T is full then count increments

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

count += 1

return count

def fullLeafNodes(T):

#base case if the tree is null and there is more depth than tree

if T is None:

return

#initiate a variable to count

count = 0

#if the T is a leaf and it has reached a max items count is incremented

if T.isLeaf:

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

return 1

else:

#for loop traverses the child items sends the rest of the tree

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

count =+ fullLeafNodes(T.child[i])

return count

def FindDepth(T, k):

#base case if the tree is null and there is more depth than tree

if T is None:

return

#if the key is inside that node return 0

if k in T.item:

return 0

#if both cases were not reached then return -1 since it wasnt found

if T.isLeaf:

return -1

#if the item is bigger than the biggest value in T then call that last child in the list

if k>T.item[-1]:

d = FindDepth(T.child[-1],k)

else:

#if not then check send each node and if it is less then that child send that child list

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

if k < T.item[i]:

d = FindDepth(T.child[i],k)

#if the tree is traversed and key was no found return -1

if d == -1:

return -1

#else return the number of depth +1 because of the then it would increment depth at all

return d +1

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:

Insert(T,i)

PrintD(T,'')

print()

print('The height of the tree is:',height(T))

print()

print('Extracted tree into the following list')

extracted = []

extracted =extractItems(T,extracted)

print(extracted)

print()

depth= 2

print('Min value of depth:',depth,' is ',minAtDepth(T,depth) )

print()

print('Max value of depth:',depth,' is ',maxAtDepth(T,depth) )

print()

print('Number of Nodes in depth', depth, ':',nodesInDepth(T,depth))

print()

print('Printing items in depth:', depth)

PrintAtDepth(T,depth)

print()

print()

print('Printing number of full nodes:',fullNodes(T))

print()

print('Printing number of full leaf nodes:',fullLeafNodes(T))

print()

key = 200

print('Find the depth of key:',key,'at depth:', FindDepth(T,key))

I Ericka Najera certify that this project is entirely my own work. I wrote, debugged, and teste 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.

