Rigoberto Quiroz

3/23/2019

Lab4 Report

CS2302 1:30PM – 2:50 PM

Description:

For this lab I had to manipulate a B-Tree (Balance Tree). I had to manipulate various things from the B-tree and turn them into functions, such as getting the height, getting its max and min elements a certain depth of the tree, printing the number of nodes and elements each node contains at a certain depth. Determining the number of full nodes, the tree has, meaning that if a node contains 5 elements then that node is full, and looking for specific elements in the B-Tree.

The way I was able to solve this lab was learning how to traverse a B-tree since the leaf nodes are so important because they determine your base cases. I was also able to solve the lab by 1) Since it’s a B-tree is a balanced then both the left and right branches have the same height. 2) Knowing that the left most element contains the lowest element in tree and rightmost element contains the highest element in tree. 3) Finally knowing that a full node contains exactly 5 elements.

**4.1 Compute Height**

Since we are working with a balance tree, we know that both left and right branches have the same height, so as long I can traverse a B-tree to compute one side’s height, I will always get the correct height. I also checked if the tree is empty by having the base case of, when we reach the leaf node, we will return 0. Since it is done recursively, each time I advance to the next level I add 1 until leaf node is reached.

Input:

T = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11 , 3, 4, 5,105, 115, 200, 2, 45, 6]

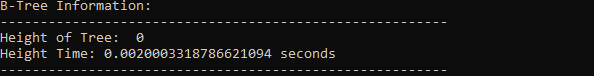
Output:



Input:

T = []

Output:



**4.2 Extract Items to list in sorted order**

Since we know that the leftmost node contains the smallest item and the rightmost element contains the greatest node, we first must visit left node and make our way to the right node. Since we are using recursion my base case is, when we reach our leaf node we will append all the elements in the node to our empty list, then make our way into its parent, append the first element then advance to the other leaf nodes. This process will be repeated until we reach our rightmost node.

Input:

T = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11 , 3, 4, 5,105, 115, 200, 2, 45, 6]

Output:



Input:

T = []

Output:



**4.3 Get Min Element**

Since I know that the smallest element is in the leftmost node, then I will traverse the B-tree until I reach the tree’s leaf and extract the first element of the node. Since I am using recursion my base case is, when leaf node is reach, extract first element from list, otherwise traverse the list.

Input:

T = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11, 3, 4, 5,105, 115, 200, 2, 45, 6]

Output:



Input:

T = []

Output:



**4.4 Get Max Element**

Since I know that the greatest element in the B-tree is always located in the rightmost node, then I will traverse the tree until I reach its leaf and extract the last element of the node. Since I am using recursion my base case is, when we reach the leaf node, take the last item of the list. Otherwise traverse the list to the right.

Input:

T = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11, 3, 4, 5,105, 115, 200, 2, 45, 6]

Output:



Input:

T = []

Output:



**4.5 Number of Nodes**

For this method we have to advance a certain level of the tree and get the number of nodes or list that level contains. If the depth is greater than the levels of the tree than we will return 0. We will traverse the tree until we reach our desired level. Since we are using recursion our base case must be when we reach our leaf node we will return 0, and if we reach our desired depth we will return 0. Otherwise we will traverse the tree for our left and right nodes.

Input:

T = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11, 3, 4, 5,105, 115, 200, 2, 45, 6]

Output:



Input:

T = []

Output:



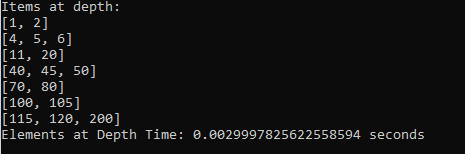
**4.6 Items at depth**

This method will function the same way as getting the number of nodes at a certain depth. Traverse the B-Tre until we reach our desired depth, if our depth is negative or exceeds our B-trees levels than we will return None or print nothing. Since we are using recursion our base cases must be, when we reach our leaf node return None, if we reach our desired depth print the list. Otherwise we will traverse the B-tree until we reach one of our base cases. We also check if the B-tree is empty.

Input:

T = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11, 3, 4, 5,105, 115, 200, 2, 45, 6]

Output:



Input:

T = []

Output:



**4.7 Full Nodes**

This method will check which node are full, meaning that a node contains 5 elements through out the entire tree. Since we are using recursion we will have some base cases such as, if B-tree is empty, if a node contains 5 elements, and if we reach our leaf nodes. Otherwise we will traverse the B-tree, left and right nodes. When we find a node that is full we will add 1 to our counter, once we are finish we will return that total number of full nodes we have.

Input:

T = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11, 3, 4, 5,105, 115, 200, 2, 45, 6]

Output:



Input:

T = []

Output:



**4.8 Leaf Nodes that are Full**

For this method we will check which leaf node is full, meaning if a leaf node has 5 elements. We will not check the other nodes. Since we are using recursion we will have some base cases such as, if B-tree is empty, if leaf is full, otherwise we will traverse the B-tree. Each time we find a leaf node we will add one to our count variable and return that variable once we have traverse the entire tree.

Input:

T = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11, 3, 4, 5,105, 115, 200, 2, 45, 6]

Output:



Input:

T = []

Output:



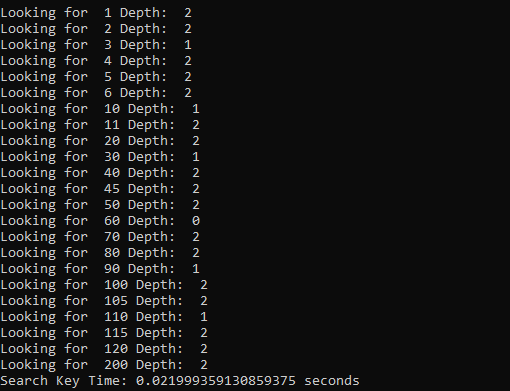
**4.9 Finding elements at B-Tree**

Since we are looking for specific keys in our tree we will have to traverse the tree either left or right depending on the key or element we are looking for. For example, we are looking for the second greatest key we will check the node and depending on the amount of the items in the node we will traverse the tree. Since we are using recursion our base cases will consist of, if our B-tree is empty, and if our traversal process reaches the leaf. As we are traversing the tree we will check all the items in the current node, if we find our key we will return 0, otherwise we will traverse the B-tree.

Input:

T = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11, 3, 4, 5,105, 115, 200, 2, 45, 6]

Output:



Input:

T = []

Output:



**Summary:**

For this lab I was able to learn more about B-trees or balance trees, the most important I learned of Trees was how to Traverse a B-tree. Traversing a B-tree is important because a B-tree is very different from other data structures, since most of the functions are very similar with the methods of Binary trees, I was able to translate most methods into this lab.

**Appendix:**

# Author: Rigoberto Quiroz

# Section: 1:30PM - 2:50 PM

# This program will create a B-Tree(Balance Tree), and will display the height

# of the tree. It will extract the elements in ascsending order into a native

# list. It will find the max in the min node at a given depth. It will also

# output the total number of nodes in the B-Tree, the Nodes that are full

# (5 elements). Print all the items at a given depth and finally and find a given

# and return the output

import time

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 extract(T,sL):

if T.isLeaf:

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

# appends single elements into list

sL.append(T.item[i])

return

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

extract(T.child[i],sL)

# appends the leftmost element then continues to the middle and so on

sL.append(T.item[i])

# goes to the right section of the tree

extract(T.child[len(T.item)],sL)

return sL

def minAtDepth(T,depth):

# BST is empty

if len(T.item) <= 0:

return None

# min element from the leftmost list

if depth == 0:

return T.item[0]

# if depth exceeds B-Tree depth

if T.isLeaf:

return None

# moves left to find the the min value

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

def maxAtDepth(T,depth):

#BST is empty

if len(T.item) <= 0:

return None

# max element found

if depth == 0:

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

# depth exceeds B-Tree depth

if T.isLeaf:

return None

# moves right

return maxAtDepth(T.child[len(T.item)],depth-1)

def totalAtDepth(T, depth):

#BST is empty

if len(T.item) <= 0:

return None

# number of elements at depth

if depth == 0:

return 1

# if depth exceeds B-Tree depth

if T.isLeaf:

return 0

count = 0

# records left and right branches

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

count += totalAtDepth(T.child[i],depth-1)

count += totalAtDepth(T.child[len(T.item)],depth-1)

return count

def itemsAtDepth(T,depth):

if len(T.item) <= 0:

return None

# prints elements at depth

if depth == 0:

print(T.item)

return

# if depth exceeds B-Tree depth

if T.isLeaf:

return None

# moves left and right

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

itemsAtDepth(T.child[i],depth-1)

itemsAtDepth(T.child[len(T.item)],depth-1)

def fullNodes(T):

if len(T.item) <= 0:

return None

# if list is full we add 1

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

return 1

# reaching leaves

if T.isLeaf:

return 0

count = 0

# B-Tree traversal

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

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

count += fullNodes(T.child[len(T.item)])

return count

def fullLeafNodes(T):

if len(T.item) <= 0:

return None

# when we reach leafs we check which leaf list are full

if T.isLeaf:

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

return 1

else:

return 0

count = 0

# B-Tree traversal

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

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

count += fullLeafNodes(T.child[len(T.item)])

return count

def keyAtDepth(T, key,h):

if len(T.item) <= 0:

return None

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

# Key Found

if T.item[i] == key:

return 0

if T.item[i] > key:

return 1 + keyAtDepth(T.child[i],key,h)

# key not found in B-Tree

if T.isLeaf:

return -(h+1)

return 1 + keyAtDepth(T.child[len(T.item)],key,h)

L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11 , 3, 4, 5,105, 115, 200, 2, 45, 6]

#L = []

T = BTree()

for i in L:

print('Inserting',i)

Insert(T,i)

PrintD(T,'')

#Print(T)

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

TotalTimeStart = time.time()

print('B-Tree Information:')

#--------------------------------------------------------------

print('--------------------------------------------------------')

startTime = time.time()

print('Height of Tree: ',height(T))

endTime = time.time()

print('Height Time:', endTime - startTime, 'seconds')

print('--------------------------------------------------------')

#--------------------------------------------------------------

h = height(T)

a = extract(T,[])

#-------------------------------------------------------------

startTime = time.time()

print('Extracted elements from list:', a)

endTime = time.time()

print('Extraction Time:', endTime - startTime, 'seconds')

print('--------------------------------------------------------')

#-------------------------------------------------------------

startTime = time.time()

print('Min at depth: ',minAtDepth(T,2))

endTime = time.time()

print('Min Element Time:', endTime - startTime, 'seconds')

print('--------------------------------------------------------')

#-------------------------------------------------------------

startTime = time.time()

print('Max at depth:', maxAtDepth(T,2))

endTime = time.time()

print('Max Element Time:', endTime - startTime, 'seconds')

print('--------------------------------------------------------')

#-------------------------------------------------------------

startTime = time.time()

print('Number of nodes at depth:', totalAtDepth(T,2))

endTime = time.time()

print('Number of Nodes Time:', endTime - startTime, 'seconds')

print('--------------------------------------------------------')

#-------------------------------------------------------------

startTime = time.time()

print('Items at depth: ')

itemsAtDepth(T,2)

endTime = time.time()

print('Elements at Depth Time:', endTime - startTime, 'seconds')

print()

print('--------------------------------------------------------')

#-------------------------------------------------------------

startTime = time.time()

print('Nodes that are full:', fullNodes(T))

endTime = time.time()

print('Full Nodes Time:', endTime - startTime, 'seconds')

print('--------------------------------------------------------')

#-------------------------------------------------------------

startTime = time.time()

print('Leave nodes that are full:', fullLeafNodes(T))

endTime = time.time()

print('Full leave Nodes Time:', endTime - startTime, 'seconds')

print('--------------------------------------------------------')

#-------------------------------------------------------------

if a is None:

print('Tree is empty')

else:

startTime = time.time()

for i in range(len(a)):

print('Looking for ', a[i], end=' ')

print('Depth: ',keyAtDepth(T,a[i],h))

endTime = time.time()

print('Search Key Time:', endTime - startTime, 'seconds')

print('--------------------------------------------------------')

#-------------------------------------------------------------

TotalTimeEnd = time.time()

print('Total Program Time:', TotalTimeEnd - TotalTimeStart, 'seconds')