**Lab #4 report – B-Trees**

**Introduction:**

The main goal of this assignment was to learn how to use B-Trees as a data structure. For this lab assignment we were asked to do some operations and methods to learn how actually B-Trees work. I did things such as getting the height of the tree, convert the tree into a sorted list, finding the minimum and the maximum element of the tree, knowing the number of nodes at an specific depth and printing them, know how many nodes and leaves are full, and know the depth of a certain item.

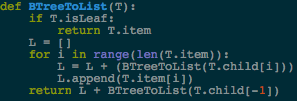
**Method #1: find the height of the tree**

This was the easiest one for me. In this one I created a method name *height()* that receives the tree as parameter. First, it checks if the tree is a leaf then it returns cero, if not it adds 1 and recursively call the next child at an index of 0.

 O(n)

**Method #2: convert a tree into a sorted list:**

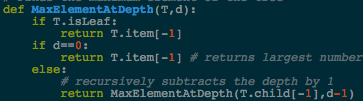
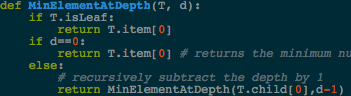
I created a method that checks first if the tree is a leaf and if it is then it returns the item. Otherwise, it initializes a list, then, in a for loop, it appends the items in ascending order by calling itself going to the child of the item. Finally, it just returns the list and the method with T.child[-1] to go to the right side of the tree.

 O(n^2)

**Method #3 and 4: Find the smallest and largest number of the tree at certain depth:**

To find the smallest number, if the tree is a leaf then it returns the item when the index is 0. If is not a leaf and the depth is 0 then returns the same thing, otherwise recursively returns the method by checking the child if the depth is more than 0 and also it keeps subtracting the depth by 1 that way at the end we have depth equals to 0 and print the item at an index of 0.

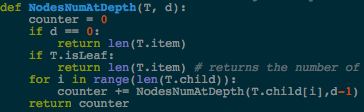
To find the largest number at a certain depth it does completely the same unless that instead of checking the index at 0, it prints the index -1 to get the largest element.



O(n) O(n)

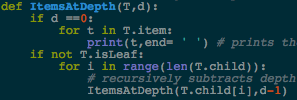
**Method #5: Number of nodes at a certain depth:**

For this method, if the tree is a leaf or if the depth is 0 it returns the length of the item. If not, in a for loop, we add the counter plus the method itself that way it can go to the next child until the depth is 0.

 O(n^2)

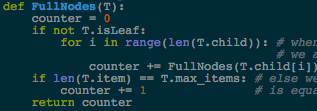
**Method #6: print the items at a certain depth:**

For this method, first, if the depth is equal to 0 it prints, by in a for loop the items in the tree. If the tree is not leaf then, in a for loop with the range of the children of the nodes, it calls the method itself to recursively repeat until the the depth is 0.

 O(n^2)

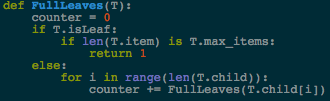
**Method #7: number of nodes that are full:**

In this method I initialize a counter that way it adds one whenever a set of nodes are full. When the tree is not a leaf then with a for loop it adds the counter plus the method itself to go ti the children of the tree. Finally if checks if the length of the item is equal or greater than the value of *max\_items* , if it’s true then we add the counter plus 1 and finally just return the counter.

 O(n^2)

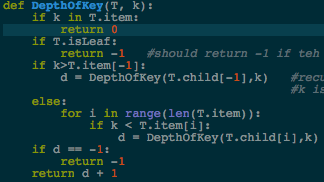
**Method #8: Number of leaves that are full:**

Same thing as the other, we check if the T is leaf, if it is we check if the length of the tree is equals a *max\_items,* returns 1 if it is full. When the tree is not leaf, with a for loop it adds a counter plus the method itself until depth is 0.

 O(n^2)

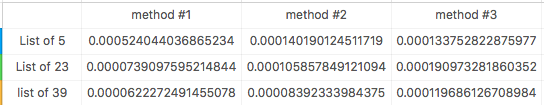
**Method #9: find the depth of the key:**

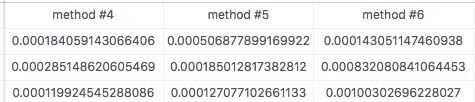
This method receives the tree and a key as parameters. It does check at what depth the key is located. First, if the key is in the item of the tree, then it returns 0, if the tree is a leaf then it returns -1, if the key is greater than the last number of the item then it recursively calls again to go the child at the last index of it. Otherwise, with a for loop checks if the key is smaller than the the number at index I, if it is then recursively calls to the child with an index of *i*.

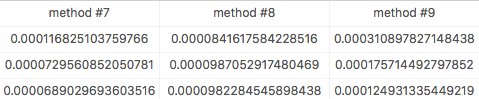
 O(n^2)

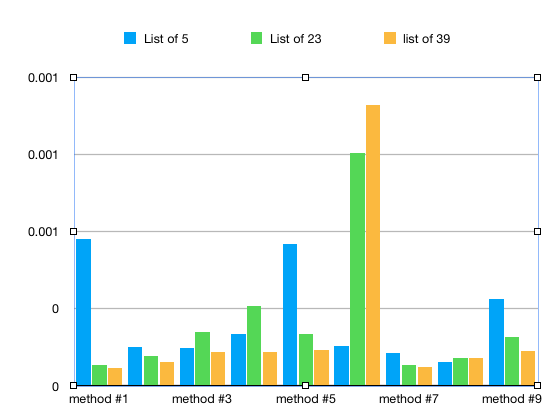
**Conclusion:**

I really like this lab because I think it was one of the easiest ones. Now I feel more confident when I have to use B-Trees as data structure, and not only that, actually with more types of trees because it is more understandable now.









﻿

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 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)

# ---------------------- METHODS FOR THE LAB -------------------

# Finds the height of the tree

def height(T):

if T.isLeaf:

return 0

return 1 + height(T.child[0]) # recursively adds 1 if the tree is not leaf

# Finds the minimum element of the tree

def MinElementAtDepth(T, d):

if T.isLeaf:

return T.item[0]

if d==0:

return T.item[0] # returns the minimum number of the tree until depth = 0

else:

# recursively subtract the depth by 1

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

# Finds the maximum element of the tree

def MaxElementAtDepth(T,d):

if T.isLeaf:

return T.item[-1]

if d==0:

return T.item[-1] # returns largest number until depth is 0

else:

# recursively subtracts the depth by 1

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

# Prints the items at a given depth

def ItemsAtDepth(T,d):

if d ==0:

for t in T.item:

print(t,end= ' ') # prints the items until depth = 0

if not T.isLeaf:

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

# recursively subtracts depth by 1

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

# Returns the number of items at a given depth

def NodesNumAtDepth(T, d):

counter = 0

if d == 0:

return len(T.item)

if T.isLeaf:

return len(T.item) # returns the number of

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

counter += NodesNumAtDepth(T.child[i],d-1)

return counter

# Checks how many nodes are full

def FullNodes(T):

counter = 0

if not T.isLeaf:

for i in range(len(T.child)): # when T is.Leaf then recursively

# we add 1 to each element

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

if len(T.item) == T.max\_items: # else we add 1 if the length of the item

counter += 1 # is equals to max\_items

return counter

# Checks how many leaves are full

def FullLeaves(T):

counter = 0

if T.isLeaf:

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

return 1

else:

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

counter += FullLeaves(T.child[i]) #this one do the same as the other

#one, but here recursivley checks if

#the item is leaf

return counter

# returns the depth of a certain number

def DepthOfKey(T, k):

if k in T.item:

return 0

if T.isLeaf:

return -1 #should return -1 if teh key is not in the tree

if k>T.item[-1]:

d = DepthOfKey(T.child[-1],k) #recursively looks for for the key if

#k is in the right side

else:

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

if k < T.item[i]:

d = DepthOfKey(T.child[i],k) # check the left side

if d == -1:

return -1

return d + 1 #returns the depth

# Converts the tree into a sorted list

def BTreeToList(T):

if T.isLeaf:

return T.item

L = []

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

L = L + (BTreeToList(T.child[i])) # it appends the items of the tree

L.append(T.item[i]) # in ascending order

return L + BTreeToList(T.child[-1]) # returns the list and recursively

# and goes to the last item

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

#L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11 , 134, 135, 136,137,138,139,140,141,142,143,144,145,146,147, 149, 150,3, 4, 5,105, 115, 200, 2, 45, 6]

T = BTree()

for i in L:

Insert(T,i)

depth = 2

key = 105

print('This is the actual tree:')

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

PrintD(T, ' ')

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

#time1 = time.time()

print('1. The height of the tree is:',height(T))

#print("--- %s seconds ---" % (time.time() - time1))

#time2 = time.time()

print('2. B-Tree converted into a sorted list:')

print(BTreeToList(T))

#print("--- %s seconds ---" % (time.time() - time2))

#time3 = time.time()

print('3. The minimum element at a depth of',depth,'is:',MinElementAtDepth(T,depth))

#print("--- %s seconds ---" % (time.time() - time3))

#time4 = time.time()

print('4. The maximum element at a depth of',depth,'is:',MaxElementAtDepth(T,depth))

#print("--- %s seconds ---" % (time.time() - time4))

#time5 = time.time()

print('5. Number of nodes at a depth of', depth,':',NodesNumAtDepth(T,depth))

#print("--- %s seconds ---" % (time.time() - time5))

#time6 = time.time()

print('6. All items at a depth of', depth, ':')

ItemsAtDepth(T,depth)

print()

#print("--- %s seconds ---" % (time.time() - time6))

#time7 = time.time()

print('7. Number of nodes that are full:',FullNodes(T))

#print("--- %s seconds ---" % (time.time() - time7))

#time8 = time.time()

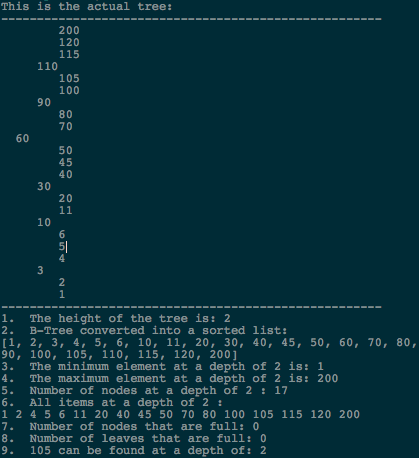
print('8. Number of leaves that are full:',FullLeaves(T))

#print("--- %s seconds ---" % (time.time() - time8))

#time9 = time.time()

print('9. ',key,'can be found at a depth of:',DepthOfKey(T,key))

#print("--- %s seconds ---" % (time.time() - time9))



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.

* David A. Davis