Lab 4 Report – CS 2302

This 4th lab consisted of similar problems of those of lab 3, but instead of using Binary Search Trees, B-Trees were used. The task given were to compute the height of the tree, return the smallest and largest elements in the tree at a given depth, return the number of nodes at a given depth, print the number of nodes and leaves that are full, and lastly to return the depth at which a given element is (or -1 if k is not in the tree).

The first problem simply computes the height of the tree, and because the trees must be balanced, taking the height of any leaf is taking the height of the tree. The code returns 0 at the leaf level and a 1 through each iteration until reaching a leaf.

The next problems, returning the min and max at depth d, had almost identical solutions. B-trees always store the smallest element at the leftmost node and rightmost for the largest. The method simply traverses down to the corresponding path until the current node is a leaf, and the returns the first or last element of the current item array.

The MaxedNodes method works by counting at each level the length of the items the current node holds and if the length equals to the maximum number of items allowed, then it increases the counter by 1. The traversal of the tree is achieved with a for loop, which calls the method for each child in T. Lastly, it returns the counter, which cumulative. The MaxedLeaves works identically, with the only difference being that it will only increase the counter when the node is full and it is a leaf.

The extract method returns a list with all the elements of the B-tree. This is done recursively, the base case consisting of T being a leaf, where there is nothing next to return. If there is, then it will extract a list for all elements of the current node, concatenating all children and current item and finally returning the list.

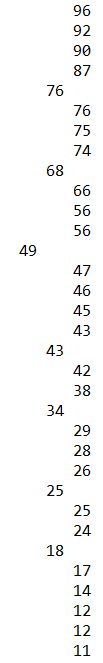
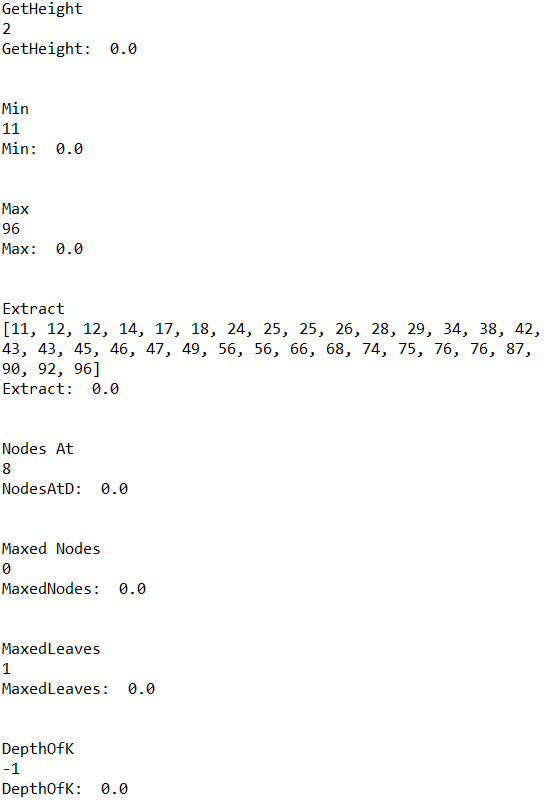
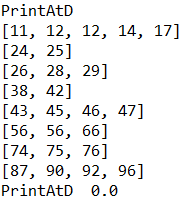
Returning the number of nodes at depth d was very similar to printing the nodes at d. For both methods the base case consists of reaching the desired level d, in which case a 1 is returned (or the current item printed), they also check if the current node is a leaf (where there is nothing next to count or print) and return. The counting and printing for the level is done with a for loop, which computes all of the children.

For the findK problem, the method does not completely work because it fails to look into the left sides of the array sometimes (this is known because the method might return a -1 when the item is in the tree.

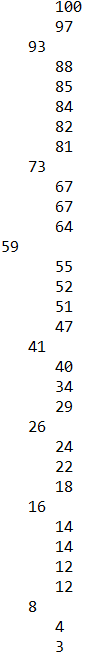
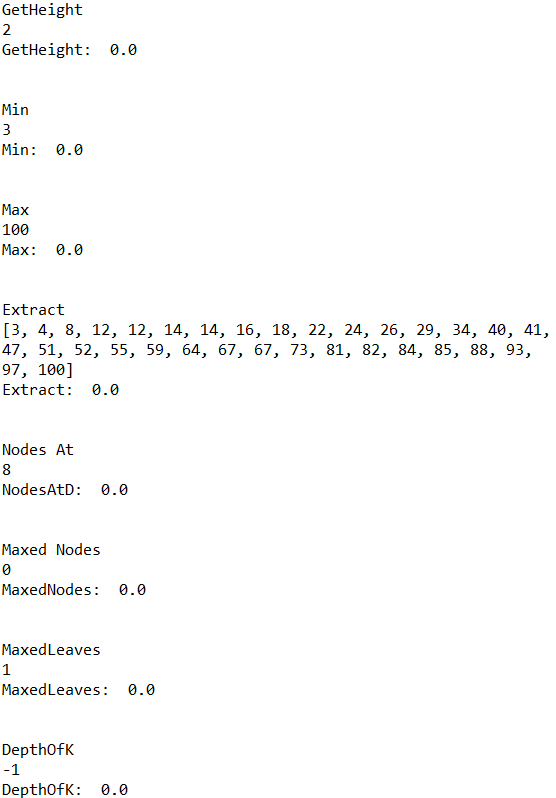
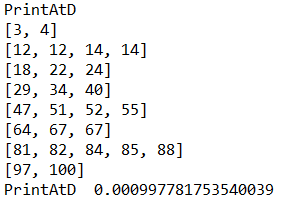
**Experimental Results**

All methods were tested with trees of different sizes and randomly generated numbers between 0 and 100.

For the tree below, these are the following outputs with their running times. Finding min max, and printAtDepth were set at depth 2. FindK was set to search for 5.



A different tree, same parameters.



For time complexity, all methods appear to run in constant time from T size 10 to 10000 as shown in the graph below. Legend stands for the problems in the order printed above.

**Academic Honesty Statement**

“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.”

-Andres Silva.

**Appendix**

'''

Created on Wed Mar 15 20:55:43 2019

CS 2302 - Andres Silva

> Teacher: Olac Fuentes

> TAs: Anindita Nath & Maliheh Zargaran

> Lab #4

> The purpose of this lab is to work with B-trees and understand their structure.

> LAST MODIFIED: MARCH 27th, 2019

'''

# Code to implement a B-tree

# Programmed by Olac Fuentes

# Last modified February 28, 2019

import math

import random

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 GetHeight(T): #It is a property of B-trees that all children are at the same leve.

if T.isLeaf:#If at bottom, return 0

return 0

else:

return 1 + GetHeight(T.child[0]) #Return height + each level.

def Min(T,d): #Look for the leftmost element.

if d == 0:

return T.item[0]

if T == None:

return

else:

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

def Max(T,d): #Look for the rightmost element.

if d == 0:

return T.item[-1]

if T == None:

return

else:

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

def MaxedNodes(T):

c = 0 # Counter of Nodes whose length is equal to the max ammount of elements allowed.

if T is None: #If T is None, return.

return

if not T.isLeaf and len(T.item) == T.max\_items: #Increase counter by one if condition is true.

c += 1

for i in range(0,len(T.child)): #Traverse entire tree

c += MaxedNodes(T.child[i])

return c #Return count.

def MaxedLeaves(T): #Do the same as above, but only increase counter when the node is a leaf

c = 0 # Counter of leaves whose length is equal to the max ammount of elements allowed.

if T is None: #If T is None, return.

return

if T.isLeaf and len(T.item) == T.max\_items: #Increase counter by one if condition is true.

c += 1

for i in range(0,len(T.child)): #Traverse entire tree

c += MaxedLeaves(T.child[i])

return c #Return count.

def Extract(T):

List = [] #Create a list to be returned.

if (T.isLeaf): #If there is no next child to exctract, return itself.

return T.item

else:

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

List += Extract(T.child[i]) + [T.item[i]] #Extract the of current node + all elements of item array

List += Extract(T.child[-1]) #Return the very last element (which was not extracted previously)

return List #return list.

def NodesAtD(T,d):

if d == 0: #If at desired level return 1

return 1

if T.isLeaf: #If there is no more nodes to look into, return 0.

return 0

else:

num = 0

for i in range(len(T.child)):#Count all Nodes of the child array.

num += NodesAtD(T.child[i],d-1)

return num #Return the count.

def PrintAtD(T,d):

if d == 0: #If at desired level, print current node.

print(T.item, ' ')

return #Return statement to stop.

if T.isLeaf:#If there is nothing else to print, return

return

else:

for i in range(len(T.child)): #Repeat for all existing children.

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

def DepthOfK(T,k): #Needs fix.

if k in T.item:

return 0

if T.isLeaf:

return -1

if k > T.item[-1]:

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

else:

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

if k < T.item[i]:

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

if d == -1:

return -1

return d + 1

#L = [17,8,12,28,34,4,7,9,11,13,15,16,24,27,30,33,37,40,42,50]

T = BTree()

for i in range(0,33):

Insert(T,random.randint(0,100))

PrintD(T,' ')

start = time.time()

print("\n\nGetHeight")

print(GetHeight(T))

end = time.time()

print("GetHeight: ", end - start)

print("\n\nMin")

start = time.time()

print(Min(T,2))

end = time.time()

print("Min: ", end - start)

print("\n\nMax")

start = time.time()

print(Max(T,2))

end = time.time()

print("Max: ", end - start)

print("\n\nExtract")

start = time.time()

print(Extract(T))

end = time.time()

print("Extract: ", end - start)

print("\n\nNodes At")

start = time.time()

print(NodesAtD(T,2))

end = time.time()

print("NodesAtD: ", end - start)

print("\n\nMaxed Nodes")

start = time.time()

print(MaxedNodes(T))

end = time.time()

print("MaxedNodes: ", end - start)

print("\n\nMaxedLeaves")

start = time.time()

print(MaxedLeaves(T))

end = time.time()

print("MaxedLeaves: ", end - start)

print("\n\nDepthOfK")

start = time.time()

print(DepthOfK(T,5))

end = time.time()

print("DepthOfK: ", end - start)

print("\n\nPrintAtD")

start = time.time()

PrintAtD(T,2)

end = time.time()

print("PrintAtD ", end - start)