Nicole Favela

Lab 4 report

CS 2302

March 26, 2019

**Introduction:**

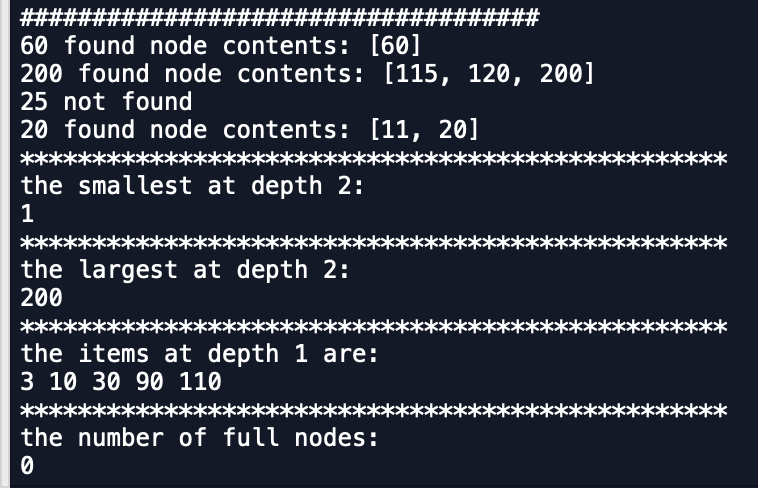
The problems I was trying to solve were described in parts 1-9 in the lab description. They involve performing basic operations on a B-tree. The first part was the basic height calculation of a B-tree. Part 2 was extracting all the items of a B-tree into a sorted list by utilizing the properties of a B-tree. Part 3 and 4 involved obtaining the smallest and largest items respectively at a certain depth. Part 5 counted the number of nodes at a certain depth provided. Part 6, PrintAtDepth, simply printed those items at depth d. Part 7 counted the number of full nodes in the tree and part 8 similarly counted the number of full leaves. Part 9 was searching for a key k and returning the depth d at which the item was found and -1 if not in the tree.

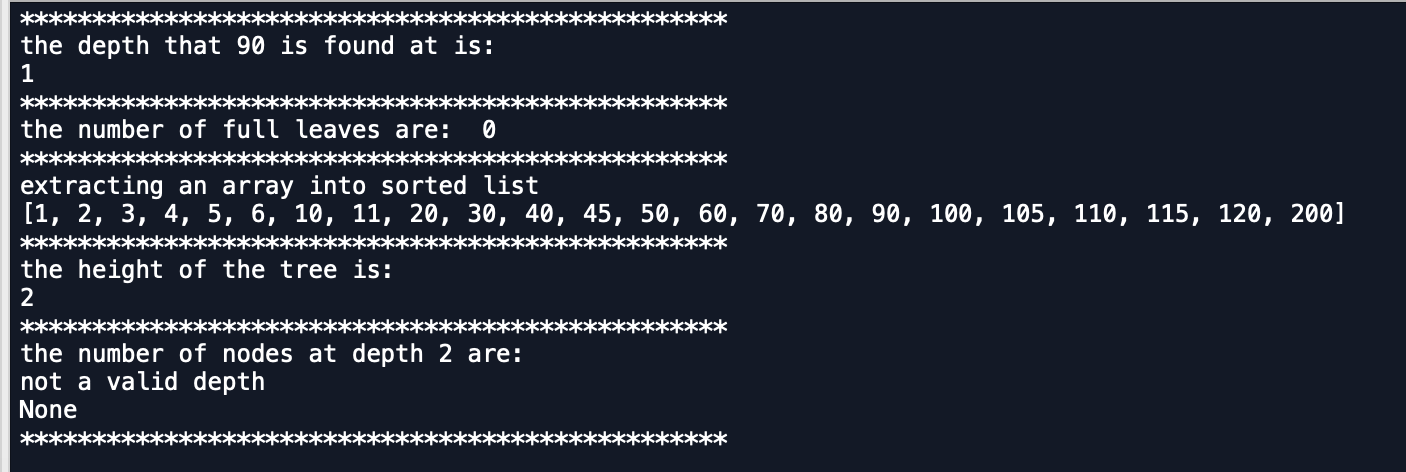
**Proposed Solution**

The height of the B-tree was a function already provided that recursively adds 1 to the tree with every call. In part 2, extractToSorted, was the function I created to extract all the items from the B-tree and create a sorted array of the integers. My approach to that was to use the properties of a B-tree where the farthest left item is the smallest and the largest is the item on the farthest right or the last index of T.child. From there, I started to accumulate items in the array using a for loop. Initially, the array was skipping some items in between which is why I figured out that I need to add another conditional within the while loop. My approach to part 3 and 4 was to use the last and first items in the B-tree respectively to obtain the smallest and largest items in the tree and add another parameter d for depth. These are both recursive functions that call the function again with T.child and decreased depth. For part 5, I created a function called numNodesAtDepth which counts the number of nodes at a specified depth. It is a recursive function that goes through the length of the B-tree and accumulates the sum of the items in the tree. I also added an error message if the depth entered is not a valid depth. For part 6, I created PrintAtDepth which prints all the items in the root node and all the other items recursively in the tree. For part 7, I created FullNodes which counts the number of full nodes, which for this particular tree, is defined to be length 5 (max\_items). I used another variable called counter to count the number of nodes in which the length is equal to T.max\_items. For part 8, I created NumFullLeaves which is similar to FullNodes. In that function, I also used a variable count to count the number of nodes where the length is equal to max\_items and is also a leaf. For part 9, I needed to first check all the items at the first level of the tree and if it is a leaf and not found returns -1. Then in other cases the recursive call checks the next T.child and repeats the process until the item is found and depth is returned or is not found.

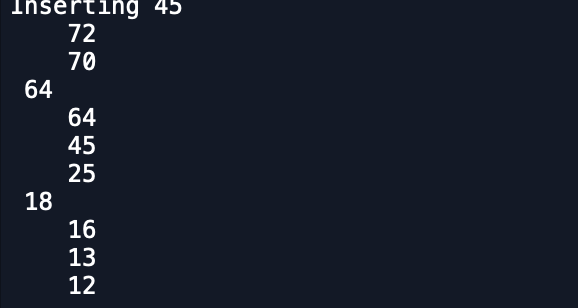
**Sample Outputs**

The sample console output is as shown in the following figure. These were the values as given in the code provided. The original B-tree image is omitted due to console space limitations.





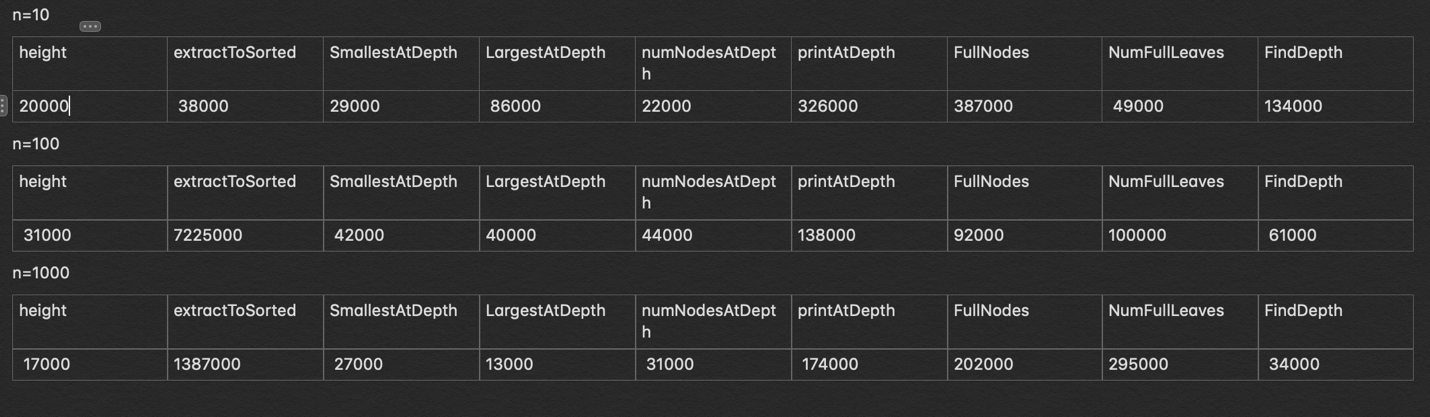
The following the B- tree was produced with array size 10.



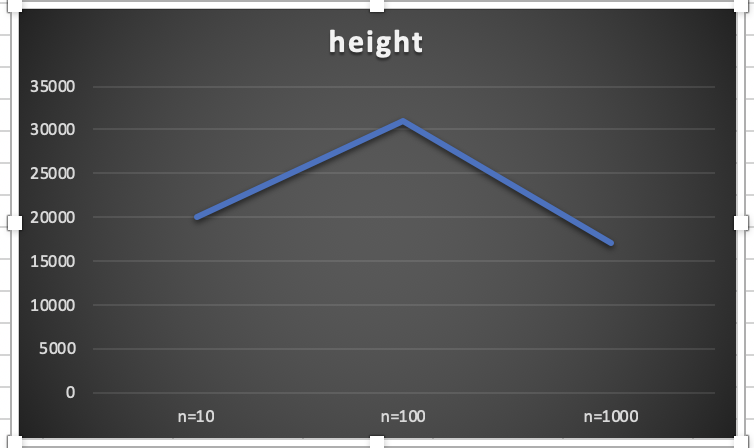
1

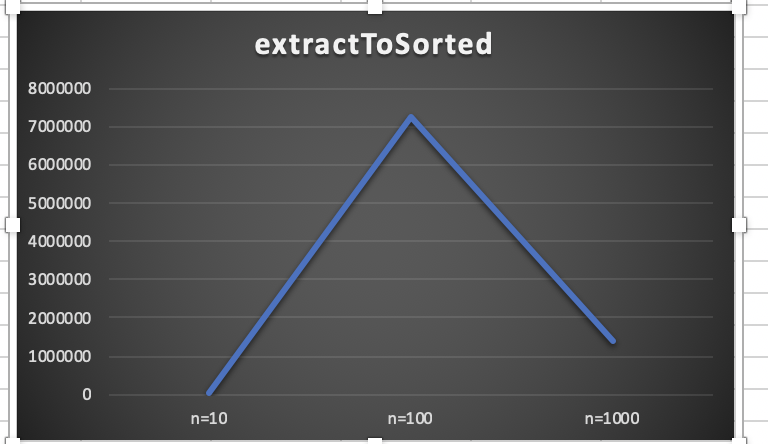
**Runtimes:**

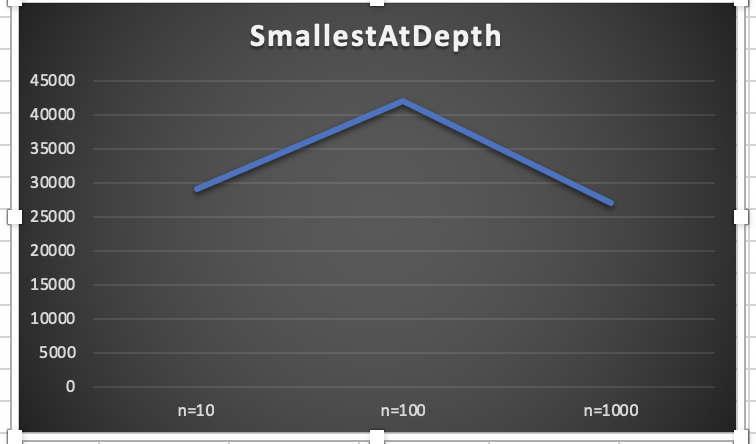
For the sample runtimes, I used ﻿numpy as np to create arrays of sizes 10, 100, and 1000. These arrays of random integers were used to construct the B-trees. As usual, I recorded the start and end times using ﻿time.time\_ns(). In every case where depth was a parameter, I used d = 1 in order to make sure the depth was always a valid depth within the tree. For FindDepth(T,k) where k is a key to be found and depth is returned, I used 90. In every case, 90 was found when performing sample runs. A table of the outputs in nanoseconds is as shown below.

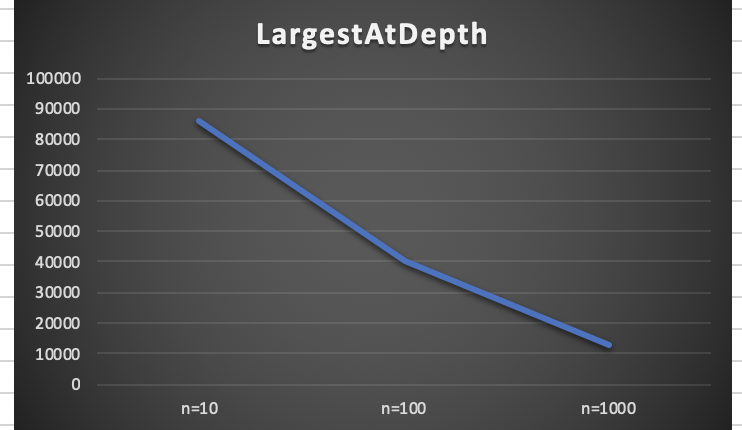


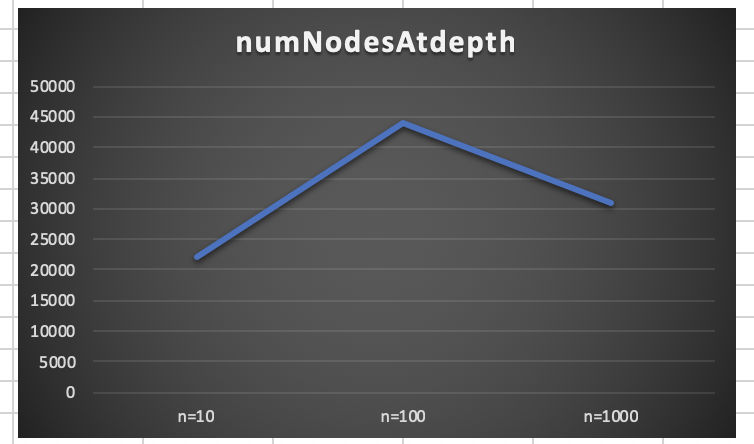
As expected, the last case where array size = 1000 took the longest to execute in terms of the insert method already provided since it prints the updated B-tree in every instance. However, most of my functions operated in O(logn) time. For reference my computer is running mac OS Mojave and has a 2.3 GHz Intel core i5 processor. The following are graphs of the runtimes of all functions in nanoseconds.

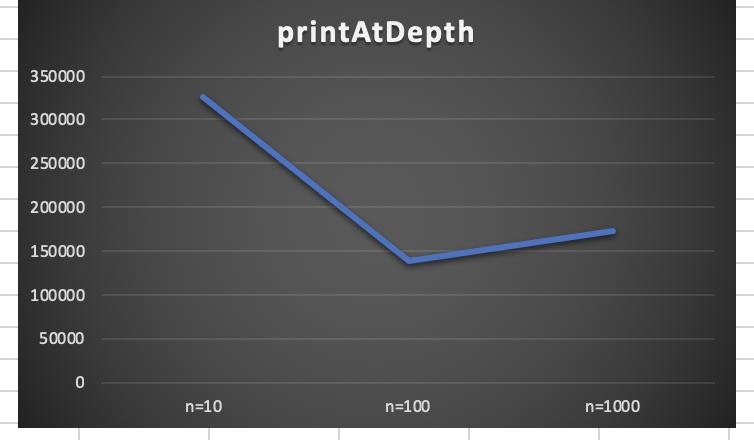


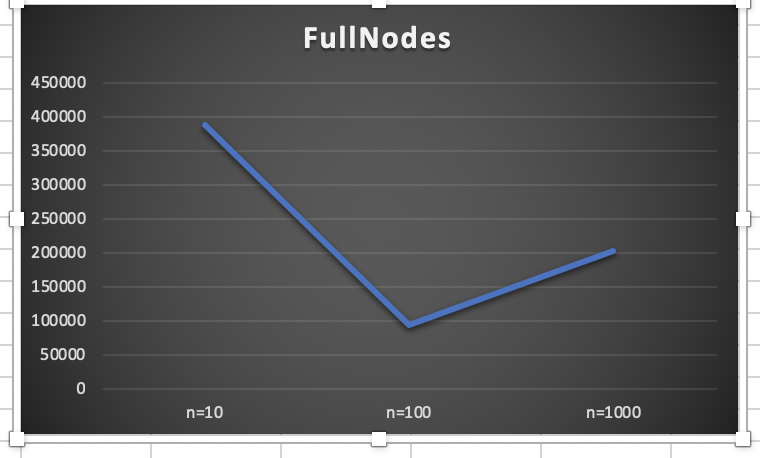


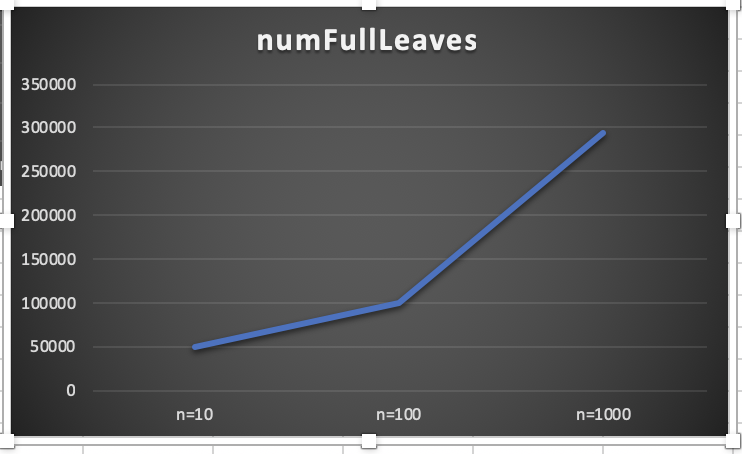


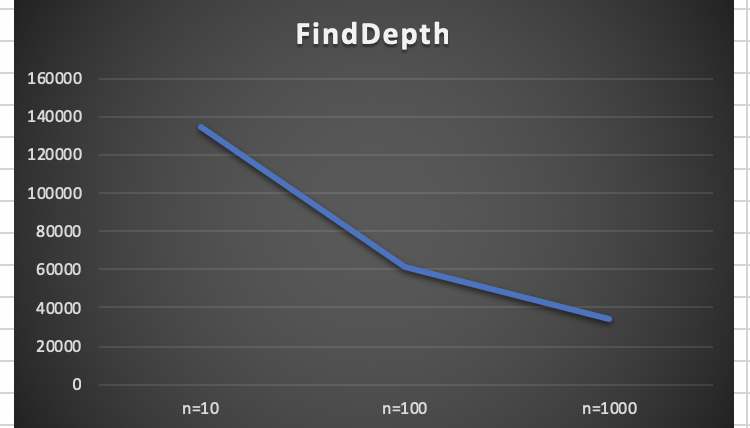












**Conclusion:**

I learned how to do several useful operations on a B-tree. Since most of the functions were recursive ones, it was also helped me practice recursion. The B-tree operations were similar in logic to the binary search tree ones in the previous lab, however, their implementations are somewhat different since B-trees involve referencing items using indices of an array. I can see how B-trees would be useful in applications where quick access is required. B-trees are generally shorter than a BST since more data can be stored at every node and data in nodes are sorted in increasing order.

**Appendix (source code):**

﻿

#CS2302

#Nicole Favela

#last modified: March 15, 2019

#Lab4

#purpose: to practice basic B tree operations

#instructor: Olac Fuentes

#TAs: Anindita Nath and Maliheh Zargaran

import math

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)

#part 1

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)

#part 4

def LargestAtDepth(T,d):

if d==0:

return T.item[len(T.item)-1] #always gets far right element

if T.isLeaf: # if depth is out of range

return -math.inf

return LargestAtDepth(T.child[len(T.item)],d-1)

#part 3

def SmallestAtDepth(T,d):

if d==0:

return T.item[0] #always gets far left item

if T.isLeaf:

return math.inf #if depth is out of range

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

#part 6 prints number of nodes at given depth

def PrintAtDepthD(T,d):

if d==0: #if at root

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

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

i+=1

else:

if T.isLeaf is False:

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

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

i+=1

#part 7 number of full nodes

def FullNodes(T):

count = 0

#for if root is full

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

count+=1

return count

#goes to rest of children and increments count

elif not T.isLeaf:

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

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

return count

#part 9

#takes key k and return the depth at which it was found

def FindDepth(T,k):

i = 0

#uses i to count the items in b tree without passing k

while i <len(T.item) and k > T.item[i]:

i+=1

#checks if k is less than value of node at i

if (i == len(T.item)) or (k < T.item[i]):

if T.isLeaf:

return -1

else:

#recursively traversing b tree

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

#if key not found recursively return -1 or count depth

if depth == -1:

return -1

else:

return depth + 1

#if k found in root

else:

return 0

#returns the sum of all the number leaves in the b tree

def NumLeafs(T):

if T.isLeaf:

return 1

sum=0

for i in T.child:

sum += NumLeafs(i)

return sum

#part 8 number of full leaves in tree

def NumFullLeaves(T):

count = 0

#base case

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

return 1

#goes to rest of list and recursively adds to count

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

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

return count

#part 2 extract to sorted list

def extractToSorted(T,arr):

if T.isLeaf:

return T.item

#empty array to store items

arr=[]

#goes through rest of list

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

#accululates items recursively

arr+=extractToSorted(T.child[i],arr)

#appends larger items to the end

if i < len(T.item):

arr.append(T.item[i])

return arr #returns array

#part 5

#number of nodes at specified depth

def numNodesAtDepth(T,d):

#stores number of nodes in root

if d==0:

return len(T.item)

if T.isLeaf and d>0:

return -1

#error message

if d >height(T):

print('not a valid depth')

return

sum = 0

#recursively goes through b tree until depth is 0

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

sum+= numNodesAtDepth(T.child[i],d-1)

return sum

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

SearchAndPrint(T,60)

SearchAndPrint(T,200)

SearchAndPrint(T,25)

SearchAndPrint(T,20)

#part 3

print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

print('the smallest at depth 2:')

print(SmallestAtDepth(T,2))

print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

#part 4

print('the largest at depth 2:')

print(LargestAtDepth(T,2))

print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

#part 6

print('the items at depth 1 are:')

PrintAtDepthD(T,1)

print()

print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

#part 7

print('the number of full nodes:')

print(FullNodes(T))

print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

#part 9

print('the depth that 90 is found at is:')

print(FindDepth(T,90))

print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

#part 8

print('the number of full leaves are: ',NumFullLeaves(T))

print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

print('extracting an array into sorted list')

sortedArray = []

#part 2

extractedArrayFromBTree = extractToSorted(T,sortedArray)

print(extractedArrayFromBTree)

print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

#part 1

print('the height of the tree is:')

print(height(T))

print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

#part 5

print('the number of nodes at depth 2 are:')

print(numNodesAtDepth(T,2))

print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

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.

-Nicole Favela