CS3 Lab 4 Report

Daniela Flores

Intro

Lab 4 consisted of writing various methods to the BTree program already provided to us by Dr.Fuentes. We had to Implement a method which computes the height of the BTree, another method to extract the items of a BTree into a sorted list, a method to return the minimum element in the tree at the given depth and another method to return the maximum element in the tree at a given depth. We also had to write a method that will return the number of nodes at the given depth, a method to print all the items at the given depth, a method to return the number of nodes in the tree that are full, another one that will return the number of full leaves that are in the BTree and a method that when given a key, we will return the depth that key is residing in, or –1 if key is not found.

Proposed Solution and Implementation:

I wrote the method to return the smallest element at the given depth by first setting the parameters to be the BTree 'T' and the given depth 'd.' I then implemented an if statement that checks if the given depth is equal to zero, and if it is to return the first item found in the current depth. Another if statement checks if T is leaf and if it is a nested if statement checks if d is not zero, if the nested if statement is entered that means that the given depth 'd' exceeds the height of tree, therefore I returned inf. Finally, I implemented an else were the recursive call is made with the first child of T, and variable 'd' being subtracted by 1. The method that'll return the maximum element at the given depth is very similar to the method for the minimum element described previously. The only difference is that instead of returning of the first element, it returns the last element and recursively calls the last child.

I wrote the method for getting the elements in a BTree (T) into a sorted list by first setting the parameters to be the BTree and an empty list. I first implemented an if statement that checks if T is leaf, and if it is a for loop appends all the items in the empty list. An else statement I then implemented; inside of the else statement a for loop iterates through the length of T.item. Inside the for loop a recursive call is made with the parameters being the children of T. after the recursive call the items of T are appended to the passed list. Outside the for loop is another recursive call is made. Finally, the populated list is returned.

The parameters for the method that returns the number of nodes at a given depth were the BTree(T) and the wanted depth(d). I first initialized a counter inside the method which will keep track of the number of nodes. An if statement then checks if the wanted depth is equal to zero, and if it is, to return 1. another if statement checks if T is leaf, and if it is, to return 0. A for loop then is used to traverse through the children of T. inside the for loop the counter keeps track of what is returned by the recursive calls made with the children of T. Outside the for loop the counter is returned.

The method that computes the height uses as parameter the BTree. An if statement checks if T is leaf, and if it is, to return 0. outside the if statement a recursive call is made with one of the children of T and to the recursive call a 1 is added to it.

I wrote the method to print all the items in the tree at the given depth first by setting the parameters the BTree and the wanted depth(d). An if statement first checks if d is zero, and if it is, a for loop within

the if statement prints all the items. Outside, another if statement checks if T is leaf, and if it is, return 0 as the depth were the leaves are is not the depth wanted. Finally, a for loop traverses through the children through a recursive call inside the for loop. The parameters of the recursive call are the children of T and d being decreased by 1.

The parameters of the method that returns the number of full nodes is the BTree and an empty list. This method first checks if T is leaf, if it is a nested if statement checks if the length of the node equals the same as max_items. If it is, the node appends to a list. Another elif statement checks if the length of the current node equals to max_items, and if it does the node is also appended to the list. A for loop then is used to traverse the whole tree with a recursive call inside the for loop. At the end the length of the list containing the full nodes is returned.

The method that returns the number of full leaves' only parameter is the BTree. A counter is first initialized, after it an if statement checks if it's a leaf, and if it is, a nested if statement checks if the length of it is the same as the number of max_items. If the length is the same as the number of max_items a 1 is returned, else it returns a 0. A for loop then is used to recursively traverse through the Btree. The counter keeps track of what is returned on every recursive call. At the end the counter is returned.

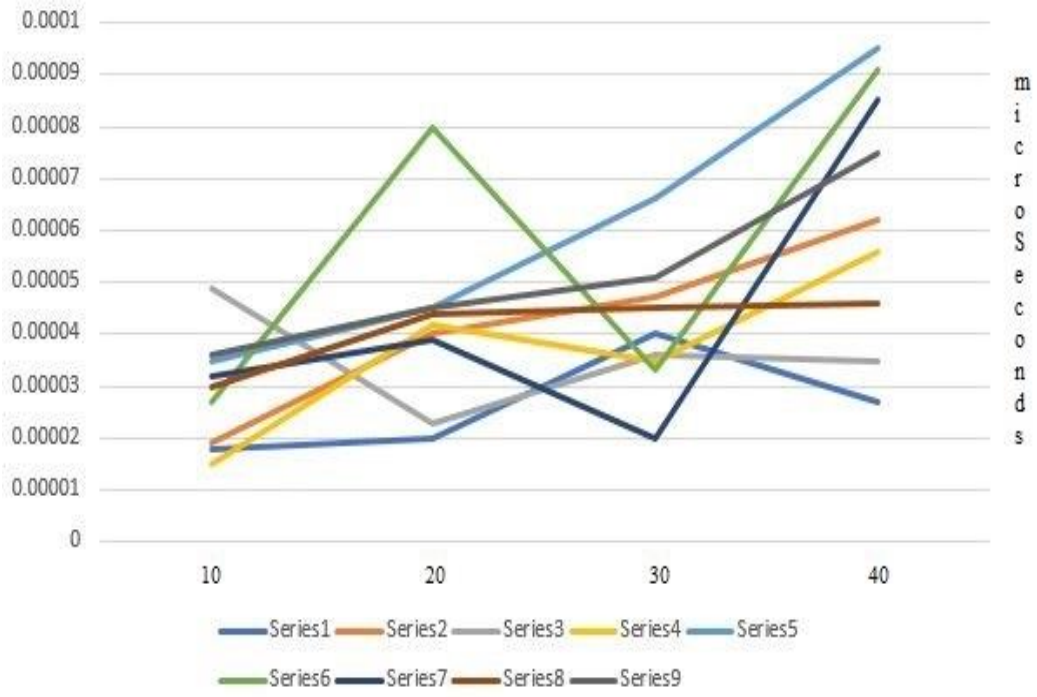
The last method, which is the one that returns the depth a given key is residing in the BTree, was first started by setting its parameters to be the BTree, the key and the height of the BTree. First an if statement checks if the key is in the items of current node, and if it is, to return 0. another if statement checks if T is leaf, inside of this statement a nested if statement checks if the key is not in the leaves. If it enters the previously mentioned nested if statement, it means that the key is not in the tree, therefore, to return –1. I first returned minus height to cancel out the depth that the method is currently storing, and then a negative 1(*return -height -1*). Finally, an else statement is used, what is returned inside of it is a 1 being added to the recursive call.

Experimental Results:

I experimented first by commenting out certain if statements in the code provided by Dr.Fuentes to see what role they played and to better understand how one is able to traverse through the children. I then experimented again with the methods I wrote by changing the size of the BTree to see if my program would still give the desired outputs. I also experimented with the depths sent as parameter to see what the output would be to detect any bugs within my program. The following image is what my program's output is when run:

```
In [1]: runfile('C:/Users/flore/.spyder-py3/
cs3Lab4.py', wdir='C:/Users/flore/.spyder-py3')
        200
        120
        115
    110
        105
        100
    90
        80
        70
 60
        50
        45
        40
    30
        20
        11
    10
        6
        5
        4
    3
        2
        1
height: 2

depth of key 30: 1
depth of key 1: 2
depth of key 60: 0
depth of key 300: -1

smallest at depth 2: 1
largest at depth 2: 200

Btree to sorted list:  [1, 2, 3, 4, 5, 6, 10, 11, 20,
30, 40, 45, 50, 60, 70, 80, 90, 100, 105, 110, 115,
120, 200]

num of full leaves: 0
nodes at depth 1: 2
items at depth 1: 3 10 30 90 110
Number of full nodes in Tree: 0
```

Running Times:




```
Series1 = height
Series2 = depth of given key
Series3 = smallest element at depth
Series4 = largest element at depth
Series5 = BTree to list
Series6 = number of full leaves
Series7 = nodes at depth
Series8 = items at depth
Series9 = number of full nodes
```

Conclusion:

What I learned from this lab is a new data structure and how to work with it. I also learned how to traverse though this data structure, how to extract the data inside this and how to get the data in a specific location within this BTree. I was also able to see how a BTree differs from a BST.

Appendix:

#Course:CS2302

#aAuthor:Daniela Flores

#Lab4

#Instructor:Olac Fuentes

#T.A: Dita Nath

```
#date of last Mod: 3/15/19
#purpose of program: in this program I had to write various functions to better understand BTrees.
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)

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)
##########################################################################
#########################################################
#returns height of BTree by adding 1 till leaves are reached
def height(T):
    if T.isLeaf:
        return 0
    return 1 + height(T.child[0])
#########################################################
#returns smallest element in BTree
def smallAtDepth(T,d):
#when reaching desired depth, itll return its first element
  if d == 0:
    return T.item[0]
```

```
#checks if given depth 'd' is the one were leaves are, if it is continue
# if not, that means given depth 'd' is greater than height of tree, therefore return inf
 if T.isLeaf:
   if d != 0:
      return math.inf


 else:
   return smallAtDepth(T.child[0],d-1)
############################################################
#returns largest element in BTree
def largestAtDepth(T,d):
#checks if given depth 'd' is the one were leaves are, if it is continue
# if not, that means given depth 'd' is greater than height of tree, therefore return inf
 if T.isLeaf:
   if d != 0:
      return math.inf
#when reaching desired depth, the last element will be returned
 if d == 0:
   return T.item[-1]
 else:
   return largestAtDepth(T.child[-1],d-1)
############################################################
#when given a key, this method will return the depth the key was found or -1 if it wasnt found
def depthOfKey(T,k,h):
   #checks if k is in elements of current depth
   if k in T.item:
      return 0
   if T.isLeaf:
      #if statement checks if key wasnt found in leaves
```

```
        if k not in T.item:
        # since it has already traversed through whole tree without finding the key,depth would be
        #equal to the height of tree, therefore to make it return -1, I subtracted the height of tree
        #to make it equal 0, then added -1
          return -h+-1
    else:
     #one is added every time it goes down Btree to keep track of which depth is operating at.
     return 1+ depthOfKey(T.child[FindChild(T,k)],k,h)
############################################################3
#turns BTree to sorted list
def BTreeToList(T,L):
    #it appends leaves elements into list
    if T.isLeaf:
        for t in T.item:
            L.append(t)
    else:
        #traverses through the children
        for i in range(len(T.item)):
            BTreeToList(T.child[i],L)
            #appends items into list
            L.append(T.item[i])
        BTreeToList(T.child[len(T.item)],L)
   #sorted list is returned
    return L
#########################################################
#returns the number of leaves that are full
def fullLeaves(T):
    # counter to keep track of full leaves
    counter = 0
```

```
    if T.isLeaf:
        #checks if leaf node is full or not
        if len(T.item) ==T.max_items:
            return 1
        else:
            return 0
    #tarverses through children
    for i in range(len(T.child)):
        counter+= fullLeaves(T.child[i])
    return counter
############################################################
#returns how many nodes are at given depth
def nodesAtDepth(T,d):
    counter = 0
    #if node is found at depth d, return 1
    if d == 0:
        return 1
    #returns 0 if the depth leaves are isnt the depth wanted
    if T.isLeaf:
        return 0
    #traverses through children
    for i in range(len(T.child)):
        counter += nodesAtDepth(T.child[i],d-1)
    return counter
############################################################
#prints items at given depth
def itemsAtDepth(T,d):
    #prints out items when desired depth is reached
    if d == 0:
```

```
        for j in range(len(T.item)):
            print(T.item[j],end=' ')

    #returns 0 if depth of leaves isnt the one wanted
    if T.isLeaf:
        return 0
    #traverses through children
    for i in range(len(T.child)):
        itemsAtDepth(T.child[i],d-1)
##########################################################
#returns number of nodes in BTree that are full
#length 2 = 5
#length 3 = 4
#length 1 = 1
def numFullNodes(T,L):

    if T.isLeaf:
        #checks if leaf nodes are full or not
        if len(T.item) ==T.max_items:
            L.append(T.item)

    #checks if current node is full or not
    elif len(T.item) == T.max_items:
        L.append(T.item)

    #traverses children
    for i in range(len(T.child)):
        numFullNodes(T.child[i],L)
    #returns length of L, which has all full nodes stored in it
```

```
    return len(L)


##############################################################################
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('height:',height(T))
print('')
hT = height(T)
print('')
print('depth of key %d:'%L[0],depthOfKey(T,L[0],hT))
print('depth of key 1:',depthOfKey(T,1,hT))
print('depth of key 60:',depthOfKey(T,60,hT))
print('depth of key 300:',depthOfKey(T,300,hT))
print()


print('smallest at depth %d: '%2,end = '')
print(smallAtDepth(T,2))
print('largest at depth %d: '%2,end = '')
print(largestAtDepth(T,2))
print('')

L = []
print('Btree to sorted list: ',BTreeToList(T,L))
```

```
print('')


print('num of full leaves:',fullLeaves(T))


print('nodes at depth %d:'%1,nodesAtDepth(T,1))

print('items at depth 1:',end = ' ')

itemsAtDepth(T,1)

print()

M = []

print('Number of full nodes in Tree:', numFullNodes(T,M))
```

Contract:

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 inappropiate assistance to any student in the class.

X