Lab 4

Javier Soto

CS2302

Lab four on B-trees was an exercise to demonstrate my knowledge of the use and traversal of B-trees as implemented by Professor Olac Fuentes. At the most basic level I have learn the three different ways of traversing a B-Tree. I can traverse a path a tree from root to leaf independent of the content in those I traverse. This type of traversal is use full for finding height of tree as well as returning the maximum and minimum items of trees or sub trees. Another traversal method is to traverse the entirety of the tree making sure to visit every node contained in the tree. This method of traversal is useful when printing all content of the tree or getting the sum of all items in the tree. The final and more challenging traversal method is to follow a specific path from root to leaf based on the content of each node. This traversal method is used in functions such as the search function. Much like a binary search tree depending on the item you are searching for there are three options either the searchee will be equal to the current item or it will be larger or smaller than the current item. Given this information we will know which path we need to search cutting the run time down form O(n), which it would take to check every item, to O(logn) which is at most the height of the tree.

Question one was to compute the height of the B-Tree. To do so I utilized the provided height function in the B-Tree class. What the height function does is first check if the current node is a leaf node, if so then we return zero. If this base case is not met then we return 1, identifying the current node, plus what ever is returned form the subtree at child zero passed to the same height method. As mentioned above this is the first case traversal where we traverse from root to leaf independent of content. Since this is the case, if I were to switch the code to traverse to the child at index -1, the right most child, then we would achieve the same results. The reason for this is because the definition of a B-Tree make it so that all leaf nodes are at the same distance from the root and there are no empty leaf nodes.

Question two asked to extract the items of a B-Tree into a sorted list. For this method we can take advantage of the structure of B-Tree since the smallest item in the tree is at index 0 of the left most child we first navigate to this node append items of this node to list next add the parent of this node followed by the right child of the parent node and so on. From the larger perspective the functioned is viewed as, you are passed a root of a B-Tree. We append the current node to the list created by a reclusive call of the subtree at child index 0. To this list we append the next child index followed by the next item in our current node and so forth till we are at the last item the node. Since we know there is a subtree at the right child to this item, we append that subtree to the list. Following this method, the newly created list will have all items in order. The recurrence relation for this method is T(n)=2(n/2)+1.

Question 3 and 4 were asking to return the largest and smallest item at depth respectively.

Again, this is the first type of traversal where we follow a path independent of the content. It does not matter where we pass we already know the depth that we need to be at will have the smallest item at index 0 of the left most node and the largest item will be at the -1 index of the right most node. With this information, the question becomes, how do we navigate to the correct depth of the tree?

Let’s start with if d, the desired depth, equaling zero. If d is zero, then we do not need to navigate any where we simply return the largest or smallest item at this depth. Now let’s take for example d is not 0, if we are searching for the smallest node we navigate to the child at index zero and subtract 1 from depth since we have moved down one. Had we been searching for the largest item then we would have navigated to the right most child and as before subtracted zero. We can now repeat this process till d does equal zero in which case we use instructions for when d is zero. Now what if we are at a leaf node and d is still greater than zero? We have two options what I did is before calling my method I would check that the depth does not exceed the height of the tree, using the height function from question 1. Option two would be to add an addition base case if the current node is a leaf and d is grater than zero then return so that we do not try to access a nonexistent child node. Both functions for question 3 and 4 have a recurrence relation of T(N)=T(n/2)+1

Question 5 asked to navigate to a specific depth and return the number of nodes at that depth. This is going to be a modified version of a type one traversal we need to traverse an unspecific path multiple times to cover every item at that given depth. Using the pseudo code form question 3 and 4 about traversing to a given depth we know once we are at the correct depth we need to do something. In this case if d is 0 then we return the length of the item since the length of that item is equal to the number of nodes in that item. Now when d is not zero we need to descend to every child this current item has, this differs from earlier functions where we choose to descend to a single child. To achieve these results, we simply implement a for loop. Every call to each child will return the number of nodes at the given depth, remember we are subtracting one form d at every level descended. When these numbers are returned from all reclusive calls we must add them together once every child node is accounted for we then return the total number of nodes at that given depth. The recurrences relation for this method changes to T(n)=2T(n/2)+n since we have to implement a for loop to descend every child and one additional recursive call to account for the right most subtree.

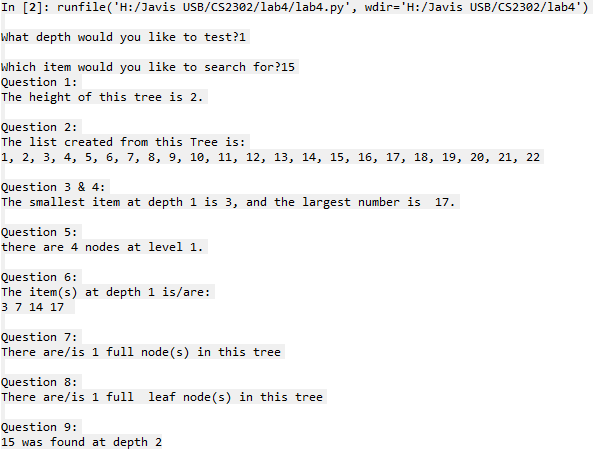
Question 6 now want to instead of counting all the nodes at depth now we must print all items at a desired depth. Since we already know how to get to a decided depth and we now know how to travers over every item at that depth now we must print the node. We can simply recycle our function for question 5 and make small changes. Starting with d is zero, we create a for loop and print every visited item. This is the only major change we need to make other than the redaction of the addition of all items. Both methods can be combined into one but for ease of tracing I have made two separate methods. Another addition I made to both methods was to instead of comparing the desired depth to the height of the tree I instead implemented a safety base case if current node is a leaf do not try and descend further.

For the moment I will skip question 7 and return to it later, for question 8 I am asked to count the number of leaf nodes that are full. This is the last of the type one traversals we see in this lab. What we do is if we are at a leaf node then we check whether this node is full or not. In this implementation of the B-Tree data type there exists a class attribute, max item, that identifies the max items the node can hold before it needs to be split. If the length of the item is equal to the max allowed, then we know the node is full thus we return 1 if not full we return 0. Using the previous pseudocode for traversing to all children we add the all the leaf nodes that are full and return that number.

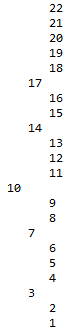
For question 7 we will use the same algorithm of question 8 however before decent to the child node we check if the current node is full if so we add one to the total count else we descend without the addition. The recurrence relation for this function is T(n)= 2t(n/2)+n^2.

Question 9 is the only type 3 traversal we will see in this code. We are asked to search for a key in the tree and return the depth of where the item is or -1 if item is not found. To find the depth our initial call can set d = 0 every descent will add one to d If the key is found we will return d if the key is not found we return -1. Now to find the key we take advantage of the B-tree structure as mentioned earlier, there are three options the key is in the current item the key is larger than item or the key is smaller than item. If we have exhausted the search, meaning we are at a leaf node and none of these three cases are true, we know key is not in tree, so we return -1. I utilized the FindChild function, so I shall describe the way that function works. A for loop is implemented to travers a B-tree node of items from 0 through length of the node. For each iteration we check is the key smaller than the item if key is smaller then we return that index, so we can access the child at that index if not we continue the loop. If the loop is exhausted, then we return the last possible option the right most child. This function assumes that the key is in fact in a child node and not the current node. If the key is at say index 1 then FindChild function will return index 2 which is larger than the key since the key is the item at index 1 so we know that the key will not be in the returned child index. This is fine so long that we know how FindChild works we can use it correctly in our code. Once FindChild return the index we descend to that child and continue our search.

In conclusion this lab was a great exercise to practice every traversal technique of a B-Tree. I feel confident after this lab that I can use the B-Tree data type in a way that will maximize the efficiency of the data type. I have printed all questions to console for easy readability by user.



A printed version of the tree to confirm my results



# -\*- coding: utf-8 -\*-

"""

Created on Tue Mar 12 12:54:07 2019

@author: Javier Soto

Professor Olac fuentes

TAs:

-Anindita Nath

-Maliheh Zaragaran

IA

-Eduardo Lara

Peer Leader

-Erick Macik

the puspose of this lab is to demonstrate my knowldge of implementing Binary

search trees as provided by Professor Olac Fuentes

"""

import btree as bt

def Question1(T):

#Compute the height of the tree

return bt.height(T)

def Question2(T,L):

#Extract the items in the B-tree into a sorted list.

#Travrse the tree in the same way as the print method and append to list

# in order

if T.isLeaf:

for t in T.item:

L.append(t)

else:

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

Question2(T.child[i],L)

L.append(T.item[i])

Question2(T.child[-1],L)

def Question3and4(T,d):

#used to check if d exceeds the depth of tree

# in seperate method to not affect the time complexity of the recursive call

if d> (bt.height(T)):

print('This depth is larger than the height of the tree')

return

MinItem= Q3Helper(T,d)

MaxItem = Question4(T,d)

print('The smallest item at depth', end =' ')

print(str(d), end = " ")

print('is ' + str(MinItem) + ', and the largest number is ', end =' ')

print(str(MaxItem) + ('.'))

def Q3Helper(T,d):

#3. Return the minimum element in the tree at a given depth d.

#will take a btree make d recursice calls and return the smallest element

# at that level

if d == 0:

return T.item[0]

return Q3Helper(T.child[0],d-1)

def Question4(T,d):

#Return the maximum element in the tree at a given depth d.

#will take a b tree make d recursive calls and returnt the largest item at

#that level

if d == 0:

return T.item[-1]

return Question4(T.child[-1],d-1)

def Question5(T,d):

#Return the number of nodes in the tree at a given depth d.

#will take a tree make d recursive calls and return all the noedes at that

#level

tot=0

if d == 0:

return len(T.item)

if T.isLeaf:

return 0

else:

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

tot += Question5(T.child[i],d-1)

return tot + Question5(T.child[-1],d-1)

def Question6(T,d):

#Print all the items in the tree at a given depth d.

if d ==0:

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

print(str(T.item[i]), end =" ")

if T.isLeaf:

return

else:

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

Question6(T.child[i], d-1)

Question6(T.child[-1], d-1)

def Question7(T):

#Return the number of nodes in the tree that are full.

fullNodes =0

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

return 1

if T.isLeaf:

return 0

else:

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

fullNodes =+ Question7(T.child[i])

return fullNodes + Question7(T.child[-1])

def Question8(T):

#Return the number of leaves in the tree that are full.

full =0

if T.isLeaf:

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

return 1

else:

return 0

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

full += Question8(T.child[i])

return full + Question8(T.child[-1])

def Question9(T,k):

#Given a key k, return the depth at which it is found in the tree, of -1 if k is not in the tree.

if(k in T.item):

return 0

if T.isLeaf:

return -1

else:

#used to find the child since we only need to follow one specific path

#either k is smaller or larger than itmes in currect list

depth = 1+ Question9(T.child[bt.FindChild(T,k)],k)

if depth >0:

return depth

return -1

def FillBTree():

L = [6, 3, 16, 11, 7, 17, 14, 8, 5, 19, 15, 1, 2, 4, 18, 13, 9, 20, 10, 12, 21, 22]

T = bt.BTree()

for i in L:

bt.Insert(T,i)

return T

inSize =int(input('What depth would you like to test?'))

searchee = int(input('Which item would you like to search for?'))

T = FillBTree()

T2 = FillBTree()

SorList =[]

print('Question 1:')

print('The height of this tree is ' + str(Question1(T)) + '.')

print()

print('Question 2:')

Question2(T,SorList)

print('The list created from this Tree is:')

print(\*SorList , sep = ", ")

print()

print('Question 3 & 4:')

Question3and4(T,inSize)

print()

print('Question 5:')

print('there are ' + str(Question5(T,inSize)) + ' nodes at level ' + str(inSize) + '.')

print()

print('Question 6:')

print("The item(s) at depth " + str(inSize) + " is/are:" )

Question6(T,inSize)

print()

print()

print('Question 7:')

print("There are/is " + str(Question7(T)) + " full node(s) in this tree")

print()

print('Question 8:')

print('There are/is ' + str(Question8(T)) + " full leaf node(s) in this tree" )

print()

print('Question 9:')

foundAt = Question9(T,searchee)

if foundAt<0:

print(str(searchee) + " is not an item in this tree. ")

else:

print(str(searchee) + " was found at depth " + str(foundAt))

bt.PrintD(T,'')

# Code to implement a B-tree

# Programmed by Olac Fuentes

# Last modified February 28, 2019

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)

'''

SearchAndPrint(T,60)

SearchAndPrint(T,200)

SearchAndPrint(T,25)

SearchAndPrint(T,20)

print(height(T))

'''

Academic dishonesty

I, Javier Soto, certify that this script and lab report are of my own unless otherwise documented above.

