Armando Ortiz

Feb 27, 2019

CS 2302 Data Structures

Lab Report 4

UTEP

Instructor: Olac Fuentes

**Introduction**

The purpose of this lab was to write functions using the B-Tree data type. These functions are problems that are based on the three main types of problems of a B-Tree; following a single path, traversing the whole tree, and following a path to a single node. These functions are used to find the height, create a sorted list using the tree, find the min and max at a certain depth, count the number of full nodes, full leaves, number of nodes at a depth, and to find the depth of a key item.

**Proposed Solution Design and Implementation**

Finding the height of the B-tree was simple. B-trees are balanced by their nature, so the way of finding the height was by going through the children of the first item in the root, and stopping when the function reaches a leaf. There is an addition of one for each level of recursion and the base case of the leaf returns zero.

Creating a Sorted List was done by going through each branch to the leaves, and then adding the items after the recursive call to move up the tree, from left to right. This was implemented by having a for loop to iterate through each branch of the tree. Inside is the recursive call for each child, which breaks with a base case at the leaf. The leaf item has a loop to append items to list L, and then outside of the base case the items in the parent levels are added after. Outside of this loop is a recursive call for the final branch on the right.

There were similar approaches to finding the min and max of the B-tree at depth d. Finding the min was like finding the height, but instead subtracting d each call until it reached zero. If the leaf of the tree is reached before d is zero, infinity is returned. This is the same for finding the max, but instead negative infinity is returned.

Counting the number of nodes at a certain depth was done by moving down the tree until the variable d in the function call was zero. Once this was zero, the length of each item was returned and added in the recursive call using the variable n. A for loop was used to iterate through each branch of the tree. If the depth was past the height, zero is returned. Printing at depth was done in a similar manner, but the items at d were printed instead of adding the length.

Counting the number of full leaves and full nodes made use of the IsFull function to see if an item was full or not. Both functions used a loop to move through each child, and recursion to move down the branches. The main differences between these two functions are the base cases. The function for counting full nodes returns one if the item is full, and returns zero if it reaches the leaves without finding a full node. The function for counting full leaves stops at the leaf, and then checks if it’s full, returning one if it is and zero if it is not full.

Finding the depth of a key has the most comparisons to keep the function efficient. There is a check to is the item is larger than the last item to move straight to the right child of the tree. If the item is found, zero is returned and d is set to zero, but after one is added to d each level of recursion. There is a check within the for loop, that if d is zero the loop breaks, acting like a Boolean function which stops the loop from going past the branch with the key. If the key is not found negative one is returned.

**Experimental Results**

**Output**

**A screenshot of a social media post

Description automatically generated**

**Runtimes**

The runtimes for the functions are as follows: O(log n) for height, O(n) for creating the Sorted List, O(log n) for the min and max depths, O(log n) for counting and printing the nodes at a certain depth, O(n) for full leaves and full nodes, and O(log n) for counting the depth of a key.

The runtimes were found using Spyder on a 2017 Macbook Pro. The program was tested using trees of differing sizes, and the depths being at the height of the tree. The goal was to find the worst case scenario of each function, and find the runtime using that.

**Runtime table(using Time.Time())**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Tree Size (num of items)** | **20** | **40** | **60** | **80** | **100** |
| **Height** | 0.007 | 0.0001 | 0.0001 | 0.0002 | 0.0001 |
| **SortedList** | 2.6941 | 4.3869 | 2.9325 | 3.1948 | 3.7193 |
| **NodesAtDepth** | 7.8678 | 2.3127 | 1.5259 | 1.5974 | 5.5075 |
| **PrintAtDepth** | 0.0017 | 0.0014 | 0.0005 | 0.0004 | 0.0012 |
| **MinAtDepth** | 1.4066 | 3.7908 | 2.0027 | 2.0027 | 5.5075 |
| **MaxAtDepth** | 1.3113 | 3.6001 | 2.1935 | 2.0027 | 5.5790 |
| **FullNodes** | 1.0014 | 2.4080 | 1.4782 | 1.7166 | 5.7936 |
| **FullLeaves** | 6.9141 | 2.0027 | 1.1921 | 1.3828 | 4.8876 |
| **DepthOfKey** | 3.0994 | 5.7220 | 2.1458 | 2.1458 | 5.7220 |

**Conclusion**

The most noticeable aspect of a B-Tree is that while they seem complicated, moving through them can be quick due to the B-tree naturally being balanced and sorted. Most operations using a B-tree are O(log n) complexity so they are quick. The main difficulty was figuring out when to start and end the recursion in a B-tree and when to stop searching. Another benefit of a B-tree is that the height doesn’t grow exponentially, thanks to the changing amount of max items. These aspects made programming for a B-Tree simple once understood.

**Appendix**

﻿"""

Author: Armando Ortiz

Course: CS 2302 Data Structures

Assignment: Lab 4

Instructor: Olac Fuentes

T.A.: Anindita Nath, Maliheh Zargaran

Last Modification: March 24, 2019

Purpose: The purpose of this program is to use a B-tree to create basic functions that:

- Compute the height of the tree

- Extract the items into a sorted list

- Return the min and max elements in the tree

- Return number of nodes at a certain depth, nodes that are full, and leaves that are full

- Print the nodes at a certain depth

- Find the depth of an item in the tree, or -1 if the item is not in the tree

"""

import math

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+' ')

# Returns height of tree

def height(T):

if T.isLeaf:

return 0

return 1 + height(T.child[0])

# Creates sorted list using B-Tree

def SortedList(T,L):

if T.isLeaf:

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

L.append(T.item[t])

else:

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

SortedList(T.child[i],L)

L.append(T.item[i])

SortedList(T.child[len(T.item)],L)

# Finds smallest item at depth d

def MinAtDepth(T,d):

if d == 0:

print(T.item[0])

return T.item[0]

if T.isLeaf:

return math.inf

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

# Finds largest item at depth d

def MaxAtDepth(T,d):

if d == 0:

print(T.item[-1])

return T.item[-1]

if T.isLeaf:

return -math.inf

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

# Returns number of nodes at depth d

def NodesAtDepth(T,d):

if d ==0:

return len(T.item)

if T.isLeaf:

return 0

else:

n = 0

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

n = n + NodesAtDepth(T.child[i],d-1)

return n

# Prints items at depth d

def PrintAtDepth(T,d):

if d ==0:

for t in T.item:

print(t,end=' ')

if not T.isLeaf:

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

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

# Returns number of full nodes

def FullNodes(T):

if IsFull(T):

return 1

if T.isLeaf:

return 0

else:

n = 0

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

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

return n

# Returns number of full leaves

def FullLeaves(T):

if T.isLeaf:

if IsFull(T):

return 1

else:

return 0

else:

n = 0

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

n = n + FullLeaves(T.child[i])

return n

# Finds depth of k, if k not found returns -1

def DepthOfKey(T,k):

if k in T.item:

return 0

if T.isLeaf: #If item not found

return -1

if k > T.item[-1]:#Search last branch

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

else:#Search rest of tree

d = -1

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

if k < T.item[i]:

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

if d == 0:# if item was found

break# Stops loop from searching rest of tree

if d == -1:

return -1

return d + 1

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("")

start = time.time()

print("Height:",height(T))

end = time.time()

heightRunTime = end - start

L2 = []

start = time.time()

SortedList(T,L2)

end = time.time()

SortedRunTime = end - start

print("List:",L2)

print("Items at depth",2,":",end=' ')

start = time.time()

PrintAtDepth(T,2)

end = time.time()

printDepthRunTime = end - start

start = time.time()

smallest = MinAtDepth(T,3)

end = time.time()

minRunTime = end-start

start = time.time()

largest = MaxAtDepth(T,3)

end = time.time()

maxRunTime = end - start

start = time.time()

count = NodesAtDepth(T,1)

end = time.time()

depthRunTime = end - start

start = time.time()

full = FullNodes(T)

end = time.time()

FullNodesRunTime = end - start

start = time.time()

fLeaf = FullLeaves(T)

end = time.time()

FullLeavesRunTime = end - start

start = time.time()

kDepth = DepthOfKey(T,105)

end = time.time()

KeyDepthRunTime = end - start

print("")

print("Min:", smallest)

print("Max: ", largest)

print("Nodes at Depth: ", count)

print("Full Nodes: ",full)

print("Full leaves", fLeaf)

print("Key found at Depth: ",kDepth)

print("")

print("Runtimes")

print("Height: ",heightRunTime)

print("SortedList: ",SortedRunTime)

print("Print at Depth: ",printDepthRunTime)

print("Number of nodes at Depth: ",depthRunTime)

print("Min Node at Depth: ",minRunTime)

print("Max Node at Depth: ",maxRunTime)

print("Full Nodes: ",FullNodesRunTime)

print("Full Leaves: ",FullLeavesRunTime)

print("Key at Depth: ",KeyDepthRunTime)

A screenshot of a cell phone

Description automatically generated