Robert Marc

80487972

CS 2302

MW 1:30-3:00

Lab 4 Report

Introduction

The purpose of this lab was to explore different methods of computing and gathering information from B-Trees.

Proposed Solutions

We were given nine problems for this lab. For the first problem, we were required to compute the height of a given B-Tree. To do this, the function simply recursively adds 1 for each time it moves on to a child node.

The second problem asks us to extract the elements from a B-Tree into a sorted list. It recursively progresses through the B-Tree in order and appends the elements in order to a temporary list. Each call returns the temporary list and concatenates it to the temporary list made in the call before it.

The third and fourth problems require us to find the minimum and maximum elements in a given B-Tree. The two functions simple move left or right through the B-Tree to the leaves of the tree and returns the left- or right-most elements in those leaves. The minimum element can be found on the left, and the maximum can be found on the right.

The fifth problem requires return the number of nodes at a given depth. The function recursively progresses through the B-Tree in order, returning 1 whenever it reaches a node at the requested depth and adding each of those up.

The sixth problem requires printing all the elements at a given depth. This function is similar to the fourth function. Instead of returning 1 whenever it reaches a node at the requested depth, a for loop prints all the elements in that node. Because it progresses through the B-Tree in order, it always goes smallest to largest.

The seventh and eighth problems are similar. The seventh asks for the number of nodes that are full. The eighth asks for the number of leaves that are full. For the seventh problem, the function moves through the B-Tree in order, checking if each node it reaches is full. If it is, it returns 1, recursively adding up all the full nodes. The eight function works the same way, but also checks if the node is a leaf too.

The ninth and final problem has us perform a search of the B-Tree for a key, returning the depth at which it was found, or -1 if it wasn’t found. The solution works the same way as the normal search function, but keeps track of the depth at which the node being accessed is at as well as if the temporary holder for the depth has changed away from -1. If the key is found, the temporary holder is changed, thus ending the comparisons and preventing the returned value from changing.

Experimental Results

Average Number of Microseconds for a given N across 100 tests

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| N = | 8 | 16 | 32 | 64 | 128 | 256 |
| Height | 50.55 | 10.29 | 43.22 | 69.54 | 145.13 | 142.74 |
| BTreeToList | 16.27 | 22.45 | 2.61 | 60.26 | 202.79 | 781.02 |
| MinAtDepth | 21.38 | 452.17 | 55.30 | 1,267.61 | 108.77 | 71.07 |
| MaxAtDepth | 32.00 | 798.62 | 24.76 | 7.43 | 509.80 | 46.99 |
| NodesAtDepth | 9.98 | 0.0 | 537.57 | 0.0 | 0.08 | 0.0 |
| ElementsAtDepth | 0.34 | 0.0 | 0.0 | 26.77 | 11.04 | 23.94 |
| FullNodes | 80.33 | 10.42 | 612.13 | 606.83 | 50.34 | 111.11 |
| FullLeaves | 52.16 | 99.75 | 119.02 | 40.80 | 542.90 | 79.92 |
| DepthOfKey | 30.66 | 19.97 | 90.28 | 217.89 | 119.18 | 90.07 |

Conclusions

This lab didn’t take long to do and was pretty straight forward in figuring out what the requested functions should do. The strangest part about this lab though was charting the average time for each function. None of the functions seemed to follow a specific pattern. None of them were linear, exponential, or logarithmic.

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.”



Appendix

# Code to implement a B-tree

# Programmed by Olac Fuentes

# Last modified February 28, 2019

"""

@Course: CS2302 MW 1:30-2:50 pm

@Author: Robert Marc, 80487972

@Assignment: Lab 4

@Instructor: Dr. Olac Fuentes

@TAs: Anindita Nath and Maliheh Zargaran

@Date of Last Modification: 3/24/19 @11:31 PM

@Purpose: To implement different operations on B-Trees

"""

import time

import random

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)

#Begin Student Code

def Height(T):

"""

Calculates the height of BTree T

If the method reaches a leaf it returns 1, adding 1 for everytime it moves

down

"""

if T.isLeaf:

return 1

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

def BTreeToList(T):

"""

Moves through the BTree in order

Whenever it reaches a node that is a leaf, it appends the items to the list

in order. It concatenates the list together through recursive calls

"""

temp = []

if T.isLeaf:

for i in T.item:

temp.append(i)

return temp

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

temp = temp + BTreeToList(T.child[j])

temp.append(T.item[j])

return temp + BTreeToList(T.child[len(T.item)])

def MinAtDepth(T,d):

"""

Moves to the furthest left item and returns the minimum element

"""

if d == Height(T):

print("Requested depth greater than tree depth.")

elif d == 0:

return T.item[0]

else:

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

def MaxAtDepth(T,d):

"""

Moves to the furthest right item and returns the minimum element

"""

if d == Height(T):

print("Requested depth greater than tree depth.")

elif d == 0:

return T.item[len(T.item)-1]

else:

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

def NodesAtDepth(T,d):

"""

Recursively progresses to nodes until the requested depth

Returns 1 when it reaches a node at the requested depth

Uses a for loop to add up all nodes at that depth

"""

if d == Height(T):

print("Requested depth greater than tree depth.")

elif d == 0:

return 1

else:

temp = 0

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

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

return temp

def ElementsAtDepth(T,d):

"""

Functions in the same way as NodesAtDepth without returning 1

Prints the elements in order at each node at the requested depth

"""

if d == Height(T):

print("Requested depth greater than tree depth.")

elif d == 0:

for i in T.item:

print(i, end=' ')

else:

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

ElementsAtDepth(T.child[j],d-1)

def FullNodes(T):

"""

Progresses through the tree in order checking for full nodes

Returns 1 when a full node is found

Uses a for loop to count up all of the full nodes in the BTree

"""

if IsFull(T):

return 1

else:

temp = 0

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

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

return temp

def FullLeaves(T):

"""

Functions the same way as FullNodes, but also checks if the node is a leaf

"""

if T.isLeaf and IsFull(T):

return 1

else:

temp = 0

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

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

return temp

def DepthOfKey(T,k):

"""

Main method for DepthOfKey, only returns results of DepthOfKeyHelper

"""

return DepthOfKeyHelper(T,k,0)

def DepthOfKeyHelper(T,k,d):

"""

Progresses through the tree in order, keeping track of the depth

If key is found, changes the holder for the depth. If the holder

changes away from -1, it stops recursively calling, and returns the

found depth

"""

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

if T.item[i] == k:

return d

temp = -1

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

if temp == -1:

temp = DepthOfKeyHelper(T.child[j],k,d+1)

return temp

"""

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,' ')

"""

N = 256

HeightAvgTime = 0

BTTLAvgTime = 0

MinDAvgTime = 0

MaxDAvgTime = 0

NodesDAvgTime = 0

ElemDAvgTime = 0

FullNAvgTime = 0

FullLAvgTime = 0

DepthKAvgTime = 0

for i in range(100):

T = BTree()

for j in range(N):

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

start = time.time()

print(Height(T))

HeightAvgTime += time.time() - start

start = time.time()

BuiltList = BTreeToList(T)

BTTLAvgTime += time.time() - start

print(BuiltList)

start = time.time()

print(MinAtDepth(T,(Height(T)-1)))

MinDAvgTime += time.time() - start

start = time.time()

print(MaxAtDepth(T,(Height(T)-1)))

MaxDAvgTime += time.time() - start

start = time.time()

print(NodesAtDepth(T,(Height(T)-1)))

NodesDAvgTime = time.time() - start

start = time.time()

ElementsAtDepth(T,(Height(T)-1))

ElemDAvgTime = time.time() - start

print()

start = time.time()

print(FullNodes(T))

FullNAvgTime += time.time() - start

start = time.time()

print(FullLeaves(T))

FullLAvgTime += time.time() - start

start = time.time()

print(DepthOfKey(T,random.randint(0,N)))

DepthKAvgTime += time.time() - start

print("HeightAvgTime: ",HeightAvgTime/100)

print("BTTLAvgTime: ",BTTLAvgTime/100)

print("MinDAvgTime: ",MinDAvgTime/100)

print("MaxDAvgTime: ",MaxDAvgTime/100)

print("NodesDAvgTime: ",NodesDAvgTime/100)

print("ElemDAvgTime: ",ElemDAvgTime/100)

print("FullNAvgTime: ",FullNAvgTime/100)

print("FullLAvgTime: ",FullLAvgTime/100)

print("DepthKAvgTime: ",DepthKAvgTime/100)