Tyler Salas

1:30-2:50 Class

Dr. Fuentes

80596221

Lab 4 Report

Lab 4, the lab assigned this week, was based around the sole idea of B-Trees

and how they can be utilized. We were required to write nine different methods for

this lab, using the B-Tree code that Dr. Fuentes shared with us on the class website.

The main idea was simply learning to traverse a B-Tree in different methods, making

different checks and things of the sort.

Below is the explanation of the different methods assigned (No.1-9) and a full

example of the codes output.

-------------------------------------------------------------------------------------

Displaying B-Tree:

200

120

115

110

109

108

107

106

105

104

103

102

100

94

93

92

91

90

80

70

60

50

45

40

30

20

11

10

8

7

6

5

4

3

2

1

Computing height of B-Tree: 2

Making Sorted List From Tree:

[1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 20, 30, 40, 45, 50, 60, 70, 80, 90, 91, 92, 93, 94, 100, 102, 103, 104, 105, 106, 107, 108, 109, 110, 115, 120, 200]

The Minimum At Depth 1 : 3

The Maximum At Depth 2 : 200

Number of nodes At Depth 2 : 10

Items At Depth 2 : [1, 2][4, 5, 6, 7, 8][11, 20][40, 45, 50][70, 80][91, 92][94, 100, 102][104, 105][107, 108, 109][115, 120, 200]None

Number of full nodes in T: 2

Number of full leaves in T: 1

Searching for item: 105

Item found at depth 2

-----------------------------------------------------------------------------------------------------

**No.1**

Problem number one of the lab required us to compute the height of the tree.

The idea for this one was simple, using the principle that all B-Trees are required

to have their leaves all at the same height. Once you have that idea down you simply

check if it is a leaf, if so, then return height of 0. Else you would return 1 + a call

to the method with child[0] of the tree. The method will stop it's recursion once there is

a leaf returning 1 for each level.

**Output**

Computing height of B-Tree: 2

This method has a O(n) runtime where n would be the height of the tree. If somebody

were to change the height of the tree, the method would automatically change its output

to the new height.

**Run-Times**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Test 1 | Test 2 | Test 3 |
| 35 Elements | .00099849700927734375 | .00098879600237335375 | .00099763211124764375 |
| 300 Elem. | .00117056438726080287 | .00124570782957023989 | .00101234878995824879 |
| 1000 ELem | .00099849700927734375 | .00195284730654872469 | .00191489652984569873 |

**No.2**

Problem number two required us to turn the elements of the B-Tree into a sorted

list. The method firstly declares an empty list and starts with a simple base case

checking if T is a leaf, if so the elements are added to the list and returned. If the

tree is not a leaf, the method goes into a for loop adding a recursive call to the

original list calling all the children, once that is done the items of each node are added,

traversing the list in order and appending the items as such.

**Output**

Making Sorted List From Tree:

[1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 20, 30, 40, 45, 50, 60, 70, 80, 90, 91, 92, 93, 94, 100, 102, 103, 104, 105, 106, 107, 108, 109, 110, 115, 120, 200]

This method has a O(n^2) runtime where n is the number of elements in the list.

This is due to the for loops that are required to iterate through the lists. If somebody

were to change the list the method would create the new list however it was modified.

**Run-Times**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Test 1 | Test 2 | Test 3 |
| 35 Elements | .00095367431640625 | .00096767538583222 | .00094333432215789 |
| 300 Elem. | .00100710845289688 | .0010152938750209944 | .00101705230857900 |
| 1000 Elem | .00099730491638183593 | .0011238975827629878 | .0017846398126386189 |

**No.3**

Problem number three in the lab required us to return the minimum element in the list

at a given depth. This method was simple as all it would do is move down the child[0] node,

reducing d by one each time. Once d were equal to zero, the item at T.item[0] would be returned

(T is the tree variable). If d became equal to zero and T ended up being a leaf this means

that the height d is not in the tree, which would make the method return -inf.

**Output**

The Minimum At Depth 1 : 3

This method has a O(n) runtime where n would be the height of the tree. This is because the worst case is depth d is not in the tree, making it iterate through all levels of the tree. If

somebody wanted to change the depth for the method to check they could change the variable

d at line 283.

**Run-Times**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Test 1 | Test 2 | Test 3 |
| 35 Elements | .0009918212890625 | . 0009642742111238 | . 00097322122404221 |
| 300 Elem | .0011828329749375 | .00129438750198749 | .00112098375428799890 |
| 1000 Elem | .00134157781946793 | .0009961128234863281 | .000995647832695631278 |

**No.4**

Problem four of the lab required us to return the maximum at a specified depth of the B-Tree. The method does exactly the same thing as the No.3 method, the only difference being that it returns T.child[-1] through its iterations and T.item[-1] for its final. In the same fashion this method returns -inf if the height d is not in the tree.

**Output**

The Maximum At Depth 2 : 200

This method has a O(n) runtime where n would be the height of the tree. This is because the worst case is depth d is not in the tree, making it iterate through all levels of the tree. If

somebody wanted to change the depth for the method to check they could change the variable

de at line 289.

**Run-Times**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Test 1 | Test 2 | Test 3 |
| 35 Elements | .00099658965325678433 | . 00094234512221781905 | .00097583243297654321 |
| 300 Elem. | .00100445747375488284 | .001029485719384759011 | .00112394843985010948 |
| 1000 Elem | .00099546381854879326 | .001329814398709708970 | .00155849328579827059 |

No.5

Method number five required us to return the number of nodes in a B-Tree at a given depth. The method I created starts with a case if d is equal to 0, then one would be returned. The next if statement would check if T was a leaf, meaning that height d was not in the tree, if so

returning -inf. Then an elif statement checking if d were equal to one, if so returning the length

of the child list. Then an else statement that would have a for loop iterating through all the

children of the tree that was called and reducing d by one, this iterates til one of the base cases

is reached, adding it to a count variable, returning the count variable once all is done.

**Output**

Number of nodes At Depth 2 : 10

This method has a O(n^2) due to having to iterate through every node that is in the B-Tree in the worst case. If somebody wanted to change the height at which to get the nodes they'd simply change the de variable at line 295.

**Run-Times**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Test 1 | Test 2 | Test 3 |
| 35 Elements | .00098463726354617384 | . 00097837485935869485 | .00096948573610495846 |
| 300 Elem | .00104212760925292968 | .001074618956924676574 | .00115842795672469853 |
| 1000 Elem | .00112855986767294888 | .001003248766874689109 | .00127682365896906078 |

**No.6**

Number six in the lab required us to print all the elements in a B-Tree at a given depth in

the tree. This method has the exact same structure as the No.5 in the lab in the way where if d

is not in the initial base cases it will iterate through each child, reducing d by one each time

until a base case is met. And of course, this method will print the items in the list instead of

returning the number of nodes like No.5 does. My code contained a bug where it would always print none at the end of every statement and I'm not sure where it is from. Though it works if depth d is not in the tree it will simply print None.

**Output**

Items At Depth 2 : [1, 2][4, 5, 6, 7, 8][11, 20][40, 45, 50][70, 80][91, 92][94, 100, 102][104, 105][107, 108, 109][115, 120, 200]None

This method has a O(n^2) due to having to iterate though each node in the Tree, along with all the children nodes. If somebody wanted to change the height at which to print, they'd change the variable de at line 301.

**Run-Times**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Test 1 | Test 2 | Test 3 |
| 35 Elements | .00099611282348632812 | . 00097734217584621532 | .0009553264277734131 |
| 300 Elem | .00099992752075195312 | .00098991876482954629 | .0010491386489128734 |
| 1000 Elem | .00121732868547386999 | .001434345187645187690 | .00097587264598269758 |

**No7**

Method number seven of the lab was required to return the number of full nodes in a given B-Tree. This method has a base case checking if T is a leaf and returning accordingly if it is so. Next an if statement checks if the current node is full and if so adding one to a count

variable declared at the beginning to equal to 0. Next the method runs a for loop that recursively

adds to count, checking all the children of the given tree. Once the for loop is over, count is returned.

**Output**

Number of full nodes in T: 2

This method has a O(n^2) runtime due to having to run through a for loop along with a recursive call nested in the for loop. If somebody were to change the number of full nodes in the B-Tree the method would automatically return the correct number of full nodes.

**Run-Times**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Test 1 | Test 2 | Test 3 |
| 35 Elements | .0009833021786005 | . 0009743584322688 | . 00097547999432368 |
| 300 Elem | .00100564956665033242 | .0009932849173416087 | .009995148576269327 |
| 1000 Elem | .0011184897130409634 | .0014198375718346579 | .0099913458992458399 |

**No.8**

The number eight method in the lab required for us to return the number of full leaves in the tree. This one was simple as all it did was iterate through the children until the leaves were found, once there the method would check if they were full, if so adding one to a count variable and later returning it.

**Output**

Number of full leaves in T: 1

This method has a O(n^2) run-time due to running a for loop with a recursive call inside of it. If the number of full leaves were changed in the tree; the method would return accordingly.

**Run-Times**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Test 1 | Test 2 | Test 3 |
| 35 Elements | .00099730491638183593 | . 00097834276584627821 | .00096543624677952112 |
| 300 Elem | .00099754333496093752 | .001029759174580270034 | .00097164391634896493 |
| 1000 Elem | .00101932957209759058 | .000958392487410378709 | .00132875798047590208 |

**No.9**

The last method was required to search for an element k and return the depth at which the element would be found. The initial call contains d equal to 0. The base case checks if T is in the initial node and will return 0. The method then checks if T is a leaf and d is not 0 meaning depth d is not in the tree and will return -1 if so. The method then calls another premade method shared by Dr. Fuentes called FindChild that finds the element in the list, though keeping track of the depth variable and ready to return such.

**Output**

Searching for item: 105

Item found at depth 2

This method has O(n^2) runtime because it must call its own method and the find child method each time. If somebody wanted to search a new number, they would change the variable k at line 319.

**Run-Times**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Test 1 | Test 2 | Test 3 |
| 35 Elements | .00099658966064453125 | . 00094724784635721625 | .00092583284373797421 |
| 300 Elem | .00105023384094238281 | .001057074372687908738 | .00112376859872645892 |
| 1000 Elem | .00195489727297625043 | .000945329666137695312 | .00133485892473509801 |

The data gathered from the runtimes that you can see in the tables seems to stay the same with minor increases as the number of elements increases. My guess would be that the methods written and the way that a B-Tree works is that it runs on the height of the tree for the most part as seen with this when the height of the tree would increase so would the run times.

I tested all these methods by modifying the B-Tree to receive expected results. To get a higher number of elements in the list, I would create a random list of a set number and create a B-Tree out of it. This method helped to obtain the runtimes of these higher element lists.

Overall this lab helped me in understanding the data structure concept of B-Trees. It helped me figure out how to iterate through one and obtain different attributes about it along he way. It was an interesting topic to learn to use.

#CS2302

#Tyler Salas

#Lab4

#Dr.Fuentes

#Anindita Nath

#Create and implement different methods of using a B-Tree

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 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)

# Gets the number of items in the list

def itemsAtDepth(T,d):

count = 0

#Gettnig the number of items in the first

if d == 0:

#Case if d is 0

return len(T.item)

if T.isLeaf:

#Case if T is a leaf

count += len(T.item)

elif d == 1:

#Case if d is 1

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

count += len(T.child[i].item)

else:

#Case if d is greater than 0 or 1

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

count += itemsAtDepth(T.child[i],d-1)

return count

# No.1 Computes height of the tree by moving down the left side of the tree since all leaf nodes must be at same height

def computeHeight(T):

if T.isLeaf:

return 0

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

# No.2 Creates sorted list by traversing through the list in order and appending elements along the way

def tree2List(T):

lst = []

if T.isLeaf:

for t in T.item:

lst.append(t)

else:

# For loop to add child elements

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

lst += tree2List(T.child[i])

# If statement to check if i is in range of item

if i < len(T.item):

lst.append(T.item[i])

return lst

# No.3 Returns the minimum element at depth d of the tree, returns -inf if the d is not a present depth in the tree

def minAtDepth(T,d):

#Base case once depth d is found

if d==0:

return T.item[0]

#Return statement if d is not in tree

if T.isLeaf:

return float('-inf')

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

# No.4 Returns the maximum element at depth d of the tree, returns -inf if the d is not a present depth in the tree

def maxAtDepth(T,d):

#Base case once depth d is found

if d==0:

return T.item[-1]

#Return statement if d is not in tree

if T.isLeaf:

return float('-inf')

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

# No.5 Returns the number of nodes at given depth d, returning -inf if d is not in the tree

def nodesAtDepth(T,d):

count = 0

if d == 0:

#Case if d is 0

return 1

if T.isLeaf:

#Case if T is a leaf

return float('-inf')

elif d == 1:

#Case if d is 1

count += len(T.child)

else:

#Case if d is greater than 0 or 1

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

count += nodesAtDepth(T.child[i],d-1)

return count

# Np.6 Prints all the items in the given depth d, returns none if d is not present

def printAtDepth(T,d):

#Getting the number of items in the first

if d == 0:

#Case if d is 0

if T.item is not None:

print(T.item)

return

if T.isLeaf:

#Case if T is a leaf and d is out of bounds

return float('-inf')

elif d == 1:

#Case if d is 1 will print the children of 0

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

if T.child[i].item is not None:

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

return

else:

#Case if d is greater than 0 or 1

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

if T.child[i] is not None:

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

return

# No.7 Will Return the Number of Full Nodes in the B-Tree

def fullNodes(T):

count = 0

# Case if T is only a leaf, will return 1 if full

if T.isLeaf:

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

return 1

# Case if T is not just a leaf will initially check root node

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

count += 1

# After basic Node is checked will iterate through all of T's children

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

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

return count

# No.8 will return the Number of full leaves in a B-Tree

def fullLeaves(T):

count = 0

# Case if node is a leaf and will only add to count if so

if T.isLeaf:

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

return 1

# Tracks all iterations of leaves to add them together

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

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

return count

# No.9 will return d, the depth at which an item being searched for can be found, returning -1 if not in the tree

def searchDepth(T,k,d):

# Base case that will stop the search at and return depth d if item is in this depth

if k in T.item:

return d

# Case if the key is not in the tree

if T.isLeaf:

return -1

return searchDepth(T.child[FindChild(T,k)],k,d+1)

import random

def createList(n):

L = []

for i in range(n):

L.append(random.randint(0,101))

return L

x = createList(1000)

L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11 , 3, 4, 5,105, 115, 200, 2, 45, 6, 102, 103, 104, 106, 107,91 ,92, 93, 94, 108, 109,7 ,8]

T = BTree()

for i in x:

Insert(T,i)

#Print(T)

print("Displaying B-Tree:")

#PrintD(T,' ')

print()

start = time.time()

# No.1 Computes Height of Tree

print("Computing height of B-Tree: ",end='')

print(computeHeight(T))

end = time.time()

res = (start-end)

print("{0:.98f}".format(res))

print()

# No.2 Creates A Sorted List From Tree

start = time.time()

print("Making Sorted List From Tree:")

print(tree2List(T))

end = time.time()

res = start-end

print("{0:.98f}".format(res))

print()

# No.3 Finds minimum at depth d, returning -inf if depth d is not in the tree

start = time.time()

d = 1

print("The Minimum At Depth",d,": ",end='')

print(minAtDepth(T,d))

end = time.time()

res = (start-end)

print("{0:.98f}".format(res))

print()

# No.4 Finds maximum at depth d, returning -inf if depth d is not in the tree

start = time.time()

de = 2

print("The Maximum At Depth",de,": ",end='')

print(maxAtDepth(T,de))

end = time.time()

res = (start-end)

print("{0:.98f}".format(res))

print()

# No.5 Finds the number of nodes at depth d, returning -inf if depth d is not in the tree

start = time.time()

de = 2

print("Number of nodes At Depth",de,": ",end='')

print(nodesAtDepth(T,de))

end = time.time()

res = (start-end)

print("{0:.98f}".format(res))

print()

# No.6 prints the items at a given depth and will print None and nothing else if element is out of bounds

start = time.time()

de = 2

print("Items At Depth",de,": ",end='')

print(printAtDepth(T,de))

end = time.time()

res = (start-end)

print("{0:.98f}".format(res))

print()

# No.7 Will return the number of full nodes in a B-Tree

start = time.time()

print("Number of full nodes in T: ",end='')

print(fullNodes(T))

end = time.time()

res = (start-end)

print("{0:.98f}".format(res))

print()

# No.8 Will return the number of leaf nodes that are full in a B-Tree

start = time.time()

print("Number of full leaves in T: ",end='')

print(fullLeaves(T))

end = time.time()

res = (start-end)

print("{0:.98f}".format(res))

print()

# No.9 Will return the depth at which a key k can be found, will return -1 if k is not in the B-Tree

start = time.time()

k = 48

print("Searching for item: ",k)

print("Item found at depth ",end='')

print(searchDepth(T,k,0))

end = time.time()

res = (start-end)

print("{0:.98f}".format(res))

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.



-Tyler Salas