

Lab 4: B-Trees

Manuel Ruvalcaba

CS2302 1:30-2:50

1. Introduction

A B-Tree is a tree that is comprised of B-Nodes. B-Nodes allow multiple keys to be stored in the same node. Each node can also have more than two children, which is not allowed in a binary search tree. The keys in each node act as dividers for its subtrees. For example, in a tree with two keys, the leftmost subtree will be all items less than the first key, the middle subtree being all keys in the middle of both keys, and the right subtree being keys greater than the 2 keys. In this lab, we will be implementing different methods that do certain tasks relating to B-Trees. These include getting the height, extracting elements into a sorted list, returning the minimum and maximum at a given depth, returning the number of nodes in a tree at a given depth, print the items at a given depth, returning the nodes and leaves that are full, and returning the depth at which a key is found.

2. Proposed Solution Design and Implementation

To solve the problem with the height of the tree, we need to know the basics of the B-Tree. Since it is a self-balancing tree, we can go to the farthest left child recursively until we reach the leaf. I created a method called *Height* and it recursively goes into the leftmost child until it reaches a leaf and returns the height.

To solve the problem with extracting the elements from a tree into a sorted list, I utilized two methods. One of them named *Extract* and the other named *Extractor*. The *Extract* method creates an empty list and calls the *Extractor* method passing the tree and the empty list. The *Extractor* method then goes through the whole tree, appending the leaves, and then appending the keys in the internal nodes until the entire sorted list is built. The *Extract* method then returns the list.

To solve the problem with returning the smallest key in the tree at a specified depth, I used a single method called *SmallestAtDepthD* that accepts a tree and a depth as parameters. The method recursively goes into the farthest left subtree until it reaches the specified depth where it returns the smallest key. If the given depth exceeds the depth of the tree, it will return *None*.

To solve the problem with returning the largest key in the tree at a specified depth, I used a single method called *LargestAtDepthD* that accepts a tree and a depth as parameters. The method recursively goes into the farthest right subtree until it reaches the specified depth where it returns the largest key. If the given depth exceeds the depth of the tree, it will return *None*.

To solve the problem with returning the number of nodes at a specified depth, I used a single method called *NodesAtLevel* that accepts a tree and a depth as parameters. This method recursively goes into each child until it reaches the specified depth where it returns one. This is added to a *num* variable that is returned, being the number of nodes at a specified depth. If the given depth exceeds the depth of the tree, it will return zero.

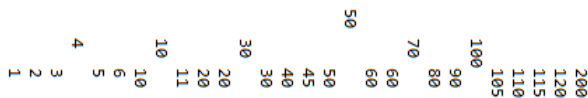
To solve the problem with printing the items at a specified depth, I used a single method called *PrintLevel* that accepts a tree and a depth as parameters. This method recursively goes into each child until it reaches the specified depth and prints the items in the node. If the given depth exceeds the depth of the tree, it will not print anything.

To solve the problem with returning the number of full nodes in a tree, I used a single method called *FullNodes* that accepts a tree as a parameter. This method goes into each node and if the number of items in the node is equal to the maximum number of items allowed in a node, it will return one. This is added to a *num* variable that is returned, being the number of full nodes.

To solve the problem with returning the number of full leaves in a tree, I used a single method called *FullLeaves* that accepts a tree as a parameter. This method goes into each node and if the node is a leaf and number of items in the node is equal to the maximum number of items allowed in a node, it will return one. This is added to a *num* variable that is returned, being the number of full leaves.

To solve the problem with returning the depth at which a key is found, I used a single method called *FindDepth* that accepts a tree and a key as parameters. If the key is in the node, it will return zero, if the node is a leaf, it will return -1. Otherwise, we will find the child that the key resides in, and then we will recursively go into the child that the key is to reside in until we find it or do not find it at all. Then the depth will be returned.

3. Experimental Results



This tree will be used to test the methods.

To test the *Height* method, I called the method with a tree and the height returned was as expected, being a height of two. Runtime of the method was 0.25394300018888316 ms.

```
IPython console
Console 1/A
#####
2
Time: 0.25394300018888316
```

To test the *Extract* method, I called the method with the tree as a parameter and it returned the sorted list. The runtime of the method was 0.5700880001313635 milliseconds.

```
IPython console
Console 1/A
[1, 2, 3, 4, 5, 6, 10, 10, 11, 20, 20, 30, 30, 40, 45, 50, 50, 60, 60, 70, 80, 90, 100, 105, 110, 115, 120, 200]
Time: 0.5700880001313635
```

To test the *SmallestAtDepthD* method, I called the method twice with the tree finding the smallest at depths one and two. The results were correct with the smallest at depth one being 4 and the smallest at depth two being 1. The runtimes of these two calls were 0.22001600063958904 ms and 0.22155799979373114 ms respectively.

```
IPython console
Console 1/A
#####
4
Time: 0.22001600063958904
1
Time: 0.22155799979373114
```

To test the *LargestAtDepthD* method, I called the method twice with the tree finding the largest at depths one and two. The results were correct with the largest at depth one being 100 and the largest at depth two being 200. The runtimes of these two calls were 0.175807000232453 ms and 0.16758199944888474 ms respectively.

```
IPython console
Console 1/A
#####
100
Time: 0.175807000232453
200
Time: 0.16758199944888474
```

To test the *NodesAtLevel* method, I called the method twice with the tree printing the number of nodes at depths one and two. The results were correct with the number of nodes at depth one being 2 and the number of nodes at depth two being 7. The runtimes of these two calls were 0.23081099971022923 ms and 0.20356599998194724 ms respectively.

```

IPython console
Console 1/A
#####
2
Time: 0.23081099971022923
7
Time: 0.20356599998194724

```

To test the *PrintLevel* method, I called the method with the tree and depth 2 as parameters and it printed the items in the level. The runtime of the method was 2.1215090000623604 ms.

```

IPython console
Console 1/A
#####
1 2 3 5 6 10 11 20 20 30 40 45 50 60 60 80 90 105 110 115 120 200
Time: 2.1215090000623604

```

To test the *FullNodes* method, I called the method with the tree as a parameter and it returned the correct answer, 1. The runtime of the method was 0.4801280001629493 ms.

```

IPython console
Console 1/A
#####
1
Time: 0.4801280001629493

```

To test the *FullLeaves* method, I called the method with the tree as a parameter and it returned the correct answer, 1. The runtime of the method was 0.26782299937622156 ms.

```

IPython console
Console 1/A
#####
1
Time: 0.26782299937622156

```

To test the *FindDepth* method, I called the method twice with the tree and searching for keys 100 and 101. The key 101 is not in the tree. The results were correct because since the key 100 was at depth 1, the method returned 1. Since the key 101 is not in the tree, the method returned -1. The runtimes of these two calls were

0.20510800004558405 milliseconds and 0.14907700006006053 ms respectively.

```

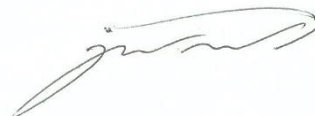
IPython console
Console 1/A
#####
1
Time: 0.20510800004558405
-1
Time: 0.14907700006006053

```

4. Conclusion

With this project, I have learned to use a B-Tree effectively and do various operations like extracting the elements into a sorted list and printing the elements from specified depths.

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.



5. Appendix

...

Course: CS2302 MW 1:30-2:50

Author: Manuel A. Ruvalcaba

Assignment: Lab #4 B-Trees

Instructor: Dr. Olac Fuentes

TA: Anindita Nath, Maliheh Zargaran

Date of Last Modification: March 15, 2019

Purpose of the Program: The purpose of this program is to calculate the height, extract the items into a list, return the minimum and maximum, return the number of nodes, print all the items, return the number of nodes that are full, return the number of leaves that are full, and return the depth of a key in a B-Tree

...

```
import timeit
```

```
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):
    #Inserts and item as a leaf
    T.item.append(i)

```

```

T.item.sort()

def IsFull(T):
    #Returns a true if the node is full, false otherwise
    return len(T.item) >= T.max_items

def Insert(T,i):
    #Inserts an item into a tree
    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):
    #Returns the height of a tree
    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):
    #Calls the Search method and prints the whether the key is found or not
    node = Search(T,k)
    if node is None:
        print(k,'not found')
    else:
        print(k,'found',end=' ')
        print('node contents:',node.item)

def Height(T):

```

```

#Calculates and returns the height of the tree
if T.isLeaf:
    return 0
return 1 + height(T.child[0])

def Extract(T):
    #Creates a list and returns what the Extractor method returns
    a = []
    return Extractor(T,a)

def Extractor(T,a):
    #Appends the items at every level in ascending order into a list
    if T.isLeaf:
        for t in T.item:
            a.append(t)
    else:
        for i in range(len(T.item)):
            Extractor(T.child[i],a)
            a.append(T.item[i])
        Extractor(T.child[len(T.item)],a)
    return a

def SmallestAtDepthD(T,d):
    #Returns the smallest item at a specified depth 'd'
    if d == 0:
        return T.item[0]
    if T.isLeaf:
        return None
    else:
        return SmallestAtDepthD(T.child[0],d-1)

def LargestAtDepthD(T,d):
    #Returns the largest item at a specified depth 'd'

```



```

    if d == 0:
        return T.item[-1]
    if T.isLeaf:
        return None
    else:
        return LargestAtDepthD(T.child[-1],d-1)

def NodesAtLevel(T,d):
    #Returns the number of Nodes at the a specified level 'd'
    num = 0
    if d == 0:
        return 1
    if T.isLeaf:
        return 0
    elif d > 0 :
        for i in range(len(T.child)):
            num += NodesAtLevel(T.child[i], d-1)
        return num

def PrintLevel(T,d):
    #Prints the items at a specified level 'd'
    if d == 0:
        for i in range(len(T.item)):
            print(T.item[i], end = ' ')
    if T.isLeaf:
        return
    elif d > 0 :
        for i in range(len(T.child)):
            PrintLevel(T.child[i], d-1)

def FullNodes(T):
    #Returns the number of nodes that are full
    if T.isLeaf and len(T.item) == T.max_items:

```

```

        return 1
    if len(T.item) == T.max_items:
        return 1
    if T.isLeaf:
        return 0
    num = 0
    for i in range(len(T.child)):
        num += FullNodes(T.child[i])
    return num

```

```

def FullLeaves(T):
    #Returns the number of leaves that are full
    if T.isLeaf and len(T.item) == T.max_items:
        return 1
    if T.isLeaf:
        return 0
    num = 0
    for i in range(len(T.child)):
        num += FullNodes(T.child[i])
    return num

```

```

def FindDepth(T,k):
    #Finds and returns the depth of a given key, returns -1 if not found
    if k in T.item:
        return 0
    if T.isLeaf:
        return -1
    else:
        l = 0
        for i in range(len(T.item)):
            if k < T.item[i]:
                l = i
                break

```

```

        else:
            l = len(T.item)
            depth = 0
            depth = FindDepth(T.child[l],k)
            if depth < 0:
                return -1
            else:
                return depth + 1

```

#This is where you test the code

```

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:

```

```

    print('Inserting',i)
    Insert(T,i)
    PrintD(T,'')
    #Print(T)
    print('\n#####')

```

```

print(height(T))
print(SmallestAtDepthD(T,2))
print(LargestAtDepthD(T,2))
print(Extract(T))
PrintLevel(T,2)
print()
print(FullNodes(T))
print(FullLeaves(T))
print(NodesAtLevel(T,2))

```

```

start = timeit.default_timer() #TimeIt library will be used to get the running times
print(FindDepth(T,200))

```

```
stop = timeit.default_timer()  
print('Time: ', (stop - start)*1000)
```