
Introduction



This lab will cover the use of B-Trees. In this structure, each node has one key and up to two children. A B-tree with order K is a tree where nodes can have up to K-1 keys and up to K children. The order is the maximum number of children a node can have.

In this project we performed different methods using B-Trees to better understand the data structure. Such as searching for an item, obtaining the maximum and minimum item at a certain depth and converting the B-Tree into a sorted list.

Design & Implementation

Problem #1: Tree Height

This problem is simple enough. A B-Tree is a self-balancing tree data structure, knowing this we can proceed to just look for the depth of one child. The code proceeds to count the number of levels it encounters until reaching a leaf node.

```
def height(T):  
    if T.isLeaf:   
        return 0  
    return 1 + height(T.child[0]) 
```

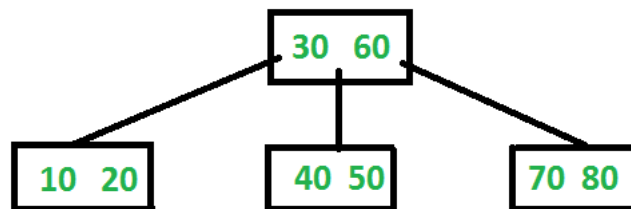
Bottom of Tree has been reached, return 0 since the root starts at height 0

Add 1 with every recursion and continue iterating using the left most child as the new root.

Problem #2: B-Tree to Sorted List

Here it can be recalled a previously seen method in BST. Where the list was created using the root node as the median and then the left side of this list would be generated in the same way using the left node as root. The same would be repeated with the right side of the list.

This method can be represented in the following way.



The sorted list would be: 10, 20, 30, 40, 50, 60, 70, 80.

It can be seen how the order of the list is: $T.child[0]$, $T.item[0]$, $T.child[1]$, $T.item[1]$ and $T.child[-1]$.

Following this, the method needs to return the item list when it reaches a leaf, it needs to check for a child and then the item at the same index the child is, and finally add the last child.

When attaching only one item to the list we need to append it since it throws an error if we try just adding it.

```
def Tree2List(T):  
    if T.isLeaf:  
        return T.item  
    a = []  
    for i in range(len(T.item)):  
        a = a + (Tree2List(T.child[i]))  
        a.append(T.item[i])  
    return a + Tree2List(T.child[-1])
```

A leaf node is reached, return the list of items in the node since we are already creating a list

Look for more children and the items in the child node, add them to the list

Repeat the process for the last child since we are not checking it in the loop. Return the sorted list.

Problem #3: Minimum Element at D

The method will first check if 'D' is equal to 0 meaning the wanted depth has been reached and then return *T.item[0]*, because that's where the current smallest item is located.

After the previous check has been made, another check will be done to see if the current node is a leaf node. This check will return -1 if it is True, meaning that the B-Tree has no nodes at 'D'.

Since we are looking for the minimum element the method will iterate over the left most child since that's where the smallest items are located and continuously reduce 'D' by 1.

```
def MinAtDepth(T,d):  
    if d == 0:  
        return T.item[0]  
    if T.isLeaf:  
        print("Chosen depth surpasses the tree's depth.")  
        return -1  
    return MinAtDepth(T.child[0],d - 1)
```

Return the smallest item of current node.

If check is True, then the B-Tree has no depth 'D'.

Iterate using the child with the smallest items, reduce 'D' by 1

Problem #4: Maximum Element at D

Same as with the method looking for a minimum element. This time the method will return *T.item[-1]*, since this is where the largest item is located and iterate over the right most child because that is where the list with the largest items is found.

Problem #5: Number of Nodes at D

Here we need to find the number of nodes that are located at a given depth. To accomplish this the method will need to use a counter variable to store the number of nodes found.

Using recursion, the base case will be if 'D' is equal to 0 return 1. This means we have reached a node at the wanted depth and returned a 1 to the counter variable.

Then another check will be made to see if we currently are at a leaf node, if we are then return infinity. This means the B-Tree has no nodes at depth 'D' since we have already reached the end of it.

The method will use a for loop to check each of the current node children and add a 1 or infinity to the counter variable after checking the previously mentioned constraints and also reducing 'D' by 1 in every recursive call.

Finally, the method will return the counter variable

Problem #6: Print Items at D

For this problem we will be following the same logic as previous method.

Using recursion, the base case will be when 'D' is equal to 0 but this time instead of returning one, the method will print the item list of the current node.

If we are at a leaf and the previous check failed then the B-Tree has no items at 'D', return.

A for loop checking the current node children will be made to print all the items in the B-Tree by reducing 'D' by one in each recursive call.

Problem #7: Find Full Nodes

Here we need to find the number of nodes where the length of *T.items* is equal to *T.max_items* which is the maximum amount of items a node's list can hold.

Same as with finding the number of nodes at a certain depth we will be creating a counter variable to store the amount of full nodes found.

First performing a check to see if the current node is full, if it is then we will add 1 to the counter variable but we will not return it since we still need to check if the current node's children are full.

The second check will see if the current node is a leaf if it is we return counter since we need to return either 0 if the node was not full or 1 if it was.

The method will make use of a for loop to check the current node children using recursion to check for further children. Adding the returned values to the counter variable.

Finally, the method will return the counter variable.

Problem #8: Find Full Leaves

This method is almost the same as the one to find full nodes. Here the base case will be when we reach a leaf node. Then a check will be performed to see if the leaf is full or not and return 1 or 0 respectively.

Returning the counter variable at the end.

Problem #9: Depth of Key

Here we need to find the depth at which a given key is located or return negative 1 if the item was not found.

The method will use a counter variable to store our current depth.

Making use of a while loop as long as the current node is not a leaf. The loop will make an amount of checks to find the key.

First checking if the key is in the current node's list, if it is then return the current value of the depth variable.

If the first check fails then check if the key is bigger than the last item of the current node, if it is then check the right most child.

If the previous checks have failed, then the item can be in one of the current node's children. Check each of them by comparing the key to each item in the current node starting by the smallest one. If the key is smaller than the current item, then enter that item's child node.

After all previous checks are made add 1 to the depth counter meaning we have entered a node's child.

Lastly after the while loop has ended we need to perform one last check to see if key is in the current leaf node. If it is then return the depth variable, else return -1.

Design

The code uses a static list to create a B-Tree, this list can be modified to add or remove items from the tree. Additionally, there is a user interface that asks the user for multiple inputs when there is a depth or key implied. Outputs are provided to show answers for each problem.

Conclusion

This lab furthered my knowledge when using a B-Tree data structure, how the child and item lists work, forms to traverse through them, how they self-balance, inserting and deleting items in them. The data structure also seems well suited for databases since each node can store a large amount of data but still the information can be accessed in logarithmic time.

Outputs

----Generated List:

[30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 11, 1, 3, 4, 5, 105, 115, 200, 2, 45, 6, 7, 8, 12, 13, 14, 15, 21, 22, 23]

----Generated Tree:

```
      200
     120
    115
  110
   105
   100
  90
   80
   70
60
   50
   45
   40
30
  23
  22
  21
20
  15
  14
13
  12
  11
10
  8
  7
  6
  5
  4
3
  2
  1
```

----Tree to Sorted List:

[1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 15, 20, 21, 22, 23, 30, 40, 45, 50, 60, 70, 80, 90, 100, 105, 110, 115, 120, 200]

----Choose depth to find the smallest item: 1

Smallest item at depth 1 : 3

----Choose depth to find the largest item: 2

Largest item at depth 2 : 200

----Choose depth to find the number of nodes: 2

Nodes at depth 2 : 9

```

----Generated List:
[30, 50, 10, 20, 60, 70, 100, 115, 200, 2, 45, 6, 7, 8, 12, 13, 14, 15, 21, 22, 23]

----Generated Tree:
      200
     115
    100
   70
  60
 50
 45
30
 23
 22
 21
 20
15
 14
 13
12
 10
 8
7
 6
 2

----Tree to Sorted List:
[2, 6, 7, 8, 10, 12, 13, 14, 15, 20, 21, 22, 23, 30, 45, 50, 60, 70, 100, 115, 200]

----Choose depth to find the smallest item: 0
Smallest item at depth 0 : 15

----Choose depth to find the largest item: 2
Largest item at depth 2 : 200

----Choose depth to find the number of nodes: 1
Nodes at depth 1 : 2

----Choose depth to print the items found: 1
Items at depth 1 : 7 12 30 70

----Full nodes found: 0

----Full leafs found: 0

----Choose key to find at a certain depth: 115
Key 115 found at depth: 2

```

Appendix

"""

Height -----

Keep adding 1 after each recursive call until reaching a leaf node. All leaf nodes of a B-Tree have the same height allowing us to just iterate over one child.

"""

```
def height(T):
```

```
    if T.isLeaf:
```

```
        return 0
```

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

"""

B-Tree to List -----

The same train of thought for a BST, the head is the center of the sorted list and we just need to obtain the left side list and right side list. The childs at 0 are all smaller than the item at 0, then childs at 1 are smaller than the item at 1. Following this logic we just need to apply this rule when appending items to the sorted list and finally return this previous list and append the child that is bigger than the last item.

"""

```
def Tree2List(T):
```

```
    if T.isLeaf:
```

```
        return T.item
```

```
    a = []
```

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

```
        a = a + (Tree2List(T.child[i]))
```

```
    a.append(T.item[i]) #since the item is an integer we need to append it
```

```
    return a + Tree2List(T.child[-1])
```

"""

Return Minimum at Depth -----

Keep iterating in the child with the smallest items until d equals 0 and return the item at 0, if this check fails and the current node is a leaf then depth parameter was bigger than the tree's depth, return -1.

"""

```
def MinAtDepth(T,d):
```

```
    if d == 0:
```

```
        return T.item[0]
```

```
    if T.isLeaf:
```

```
        print("Chosen depth surpasses the tree's depth.")
```

```
        return -1
```

```
    return MinAtDepth(T.child[0], d-1) #T.child[0] carries the smallest items
```

"""

Return Maximum at Depth -----

Keep iterating in the child with the largest items until d equals 0 and return the item at -1, if this check fails and the current node is a leaf then depth parameter was bigger than the tree's depth, return -1.

"""

```
def MaxAtDepth(T,d):
```

```
    if d == 0:
```

```
        return T.item[-1]
```

```
    if T.isLeaf:
```

```
        print("Chosen depth surpasses the tree's depth.")
```

```
        return -1
```

```
    return MaxAtDepth(T.child[-1],d-1) #T.child[-1] carries the largest items
```

"""

Return Nodes at Depth -----

Use a variable to store everytime d is 0 meaning we reached a node at a wanted depth. If this check fails and the current node is a leaf then this

means the depth parameter was bigger than the tree's depth, return infinite.

```
"""
```

```
def NodesAtDepth(T,d):
```

```
    counter = 0
```

```
    if d == 0:
```

```
        return 1 #Node at a wanted depth found
```

```
    if T.isLeaf:
```

```
        return math.inf
```

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

```
        counter += NodesAtDepth(T.child[i],d-1) #iterate over each child
```

```
    return counter
```

```
"""
```

Print Items at Depth -----

If d is 0 then print the items in the current node, if this check fails and the current node is a leaf then the depth parameter was bigger than the tree's depth, return.

```
"""
```

```
def ItemsAtDepth(T,d):
```

```
    if d == 0: #Node found print items in it
```

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

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

```
    if T.isLeaf:
```

```
        return
```

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

```
        ItemsAtDepth(T.child[i],d-1) #iterate over each child
```

```
"""
```

Full Nodes -----

If the length of the current node item list's equal to the max number of items possible in a node then add 1 the counter variable. If this check fails and it is a leaf node means the leaf node wasn't full, return 0. When the for

loop ends return the counter variable.

```
"""
```

```
def FullNodes(T):
```

```
    counter = 0
```

```
    if len(T.item) == T.max_items:
```

```
        counter += 1 #don't return to check current node childs.
```

```
    if T.isLeaf:
```

```
        return counter
```

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

```
        counter += FullNodes(T.child[i])
```

```
    return counter
```

```
"""
```

Full Leafs -----

Same as the method before but this time check if it is a leaf and then check if it is full or not.

```
"""
```

```
def FullLeafs(T):
```

```
    counter = 0
```

```
    if T.isLeaf:
```

```
        if len(T.item) == T.max_items:
```

```
            return 1
```

```
        else:
```

```
            return 0
```

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

```
        counter += FullLeafs(T.child[i])
```

```
    return counter
```

```
"""
```

Depth of Key -----

Check if the key item is bigger or smaller than the items in the current node and enter the node in which the key could be located. Each time we enter a node

add 1 to the depth variable. If the key is in the current checked node then return the depth variable. Lastly, check if the item is in the current leaf node, if it isn't then return -1

```
"""
```

```
def KeyDepth(T,k):
```

```
    depth = 0
```

```
    while T.isLeaf != True:
```

```
        if k in T.item:
```

```
            return depth
```

```
        if k > T.item[-1]:
```

```
            T = T.child[-1]
```

```
        elif k < T.item[0]: #kept getting an error when doing this inside loop
```

```
            T = T.child[0]
```

```
    else:
```

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

```
            if k < T.item[i]:
```

```
                T = T.child[i]
```

```
    depth += 1
```

```
    if k in T.item:
```

```
        return depth
```

```
    print("Item not found.")
```

```
    return -1
```

```
"""
```

```
Main -----
```

```
"""
```

```
L = [30, 50, 10, 20, 60, 70, 100,
```

```
     115, 200, 2, 45, 6, 7, 8, 12, 13, 14, 15, 21, 22, 23]
```

```
T = BTree()
```

```
for i in L:
```

```
    Insert(T,i)
```

```
print("----Generated List:\n",L)
```

```
print("\n----Generated Tree:")
```

```
PrintD(T,"")
```

```
print("\n----Tree to Sorted List:\n",Tree2List(T))
```

```
depthMin = int(input("----Choose depth to find the smallest item: "))
```

```
print("Smallest item at depth",depthMin,":",MinAtDepth(T,depthMin))
```

```
depthMax = int(input("----Choose depth to find the largest item: "))
```

```
print("Largest item at depth",depthMax,":",MaxAtDepth(T,depthMax))
```

```
depthNodes = int(input("----Choose depth to find the number of nodes: "))
```

```
if NodesAtDepth(T,depthNodes) == math.inf:
```

```
    print("Chosen depth surpasses the tree's depth")
```

```
print("Nodes at depth",depthNodes,":",NodesAtDepth(T,depthNodes))
```

```
depthItems = int(input("----Choose depth to print the items found: "))
```

```
print("Items at depth",depthItems,": ",end="")
```

```
ItemsAtDepth(T,depthItems)
```

```
print("\n\n----Full nodes found:",FullNodes(T))
```

```
print("\n\n----Full leafs found:",FullLeafs(T))
```

```
key = int(input("----Choose key to find at a certain depth: "))
```

```
print("Key",key,"found at depth:",KeyDepth(T,key))
```

I certify that this essay is original work prepared by me, Jesus A Hernandez.