Lab 4: B-trees

CS 2303 Data Structures

Laura Berrout

ID# 80607326

March 15, 2019

# Introduction

The program b-tree.py in the class webpage contains implementations of basic B-tree tree operations, including insertion, search and display. Write functions to perform the following operations:

1. Compute the height of the tree
2. Extract the items in the B-tree into a sorted list.
3. Return the minimum element in the tree at a given depth d.
4. Return the maximum element in the tree at a given depth d.
5. Return the number of nodes in the tree at a given depth d.
6. Print all the items in the tree at a given depth d.
7. Return the number of nodes in the tree that are full.
8. Return the number of leaves in the tree that are full.
9. Given a key k, return the depth at which it is found in the tree, of -1 if k is not in the tree.
10. Compute the height of the tree

## Proposed solution design and implementation

It was worked with the assumption that it’s like the Print function, but simpler. First, the base case was that if it reaches to a leaf, then return 0, since we are no longer counting the height. Else, it returned 1 and calls the function to the next child, but only the first child (T.child[0]) since the B-tree is balanced and we would not have a leaf on the last children.

In the next table are shown the experiments done for this exercise:

|  |  |  |  |
| --- | --- | --- | --- |
| Experiments | | Proposed Solution | Get Height of b-tree |
| **#** | **Changes** | **Description / Assumption** | **Results** |
| 1 |  | Recursive function to get the height |  |
| 2 | return 1 + call | Count by one until reach to the last leaf | If the reach to a leaf, then returns 0 and ends |
| 3 | L list | Test with a list of 10 items | Works fine, returns height correctly |
| 4 | L list | Test with a list of 23 items | Works fine, returns height correctly |

## Experimental results

Gets the height of the B-tree and was tested with other two list to make sure that the function was working correctly:

L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11 , 3, 4, 5,105, 115, 200, 2, 45, 6, 300, 301, 7, 8, 9, 10, 11, 12, 13, 106, 107, 108]

L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80]

## Code

def Heightbt(T):

if T.isLeaf:

return 0

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

PrintD(T,'')

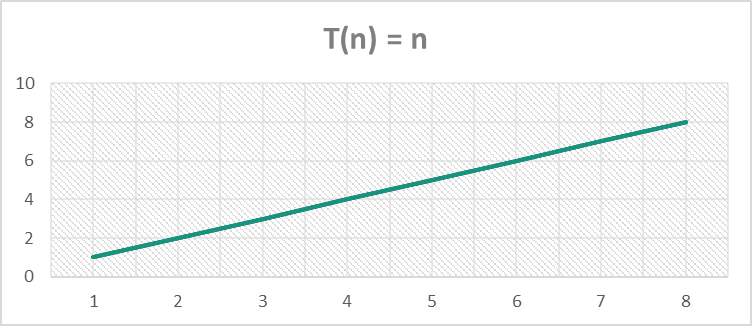
print("1) Height of b-tree: ",Heightbt(T))

## Time of execution

To get the running complexity and the Big-O notation, the steps were as follows:

|  |  |
| --- | --- |
| 1. 1 recall, so a = 1. 2. Size of the list: n, so T(n-1) 3. Recursive at the end, +1 | T(n) = aT(f(n)) + g(n)  T(n) = 1(T(n-1)) + 1 |
| * T(1) = 1 * T(2) = 2 * T(3) = 3 | Assumption: T(n) = n  **T(n) = O(n)** |

Running time: 1553658818.4603646 ns



# Extract the items in the B-tree into a sorted list

## Proposed solution design and implementation

It was also based on the Print function and follows the pre-order technique to get the left child, then the root and then the right child. Since the nodes have more than one child, insert all the elements in the child, then the root, and then all the elements on the right child. If it reaches to a leaf, insert all the items to the list. With the assumption to not create multiple list, the list was created before the call of the function and let it me a requirement to the call function (Createlist(T,list)), and then fills the list on the function.

The experiments and methods were as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Experiments | | Proposed Solution | | Extract the items in the B-tree into a sorted list. |
| **#** | **Changes** | **Description / Assumption** | | **Results** |
| 1 |  | | Taking the same approach as with bst, by preorder the b-tree, first insert the left(est) leaf and then the node and continue to the next one: def Createlist(T,list) | Place the elements of the b-tree T on the empty list |
| 2 | if T.isLeaf: | | Recursion base base is insert the item on T.item | Insert the first leaf items to the list |
| 3 | for i in range(len(T.item)) | | for every item in the T.child, insert to the list (follow the same principle as Print function) | inserts first leafs and then the node |
| 4 |  | | recursion call on the first child, insert item |  |
| 5 | for i in range(len(T.item)): Createlist(T.child[i],list) list.append(T.item[i]) Createlist(T.child[len(T.item)],list) | | recursion call to the next child, and returns the list | continues to the next child and leaf until it finishes |

## Code

def Createlist(T,list):

if T.isLeaf:

for t in T.item:

list.append(t)

else:

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

Createlist(T.child[i],list)

list.append(T.item[i])

Createlist(T.child[len(T.item)],list)

return list

print("2) List from the B-tree:")

print(Createlist(T,Sortedlist))

## Experimental results

Was also tested with the next list and confirm that returns it sorted.

L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11, 3, 4, 5,105, 115, 200, 2, 45, 6, 300, 301, 7, 8, 9, 10, 11, 12, 13, 106, 107, 108]

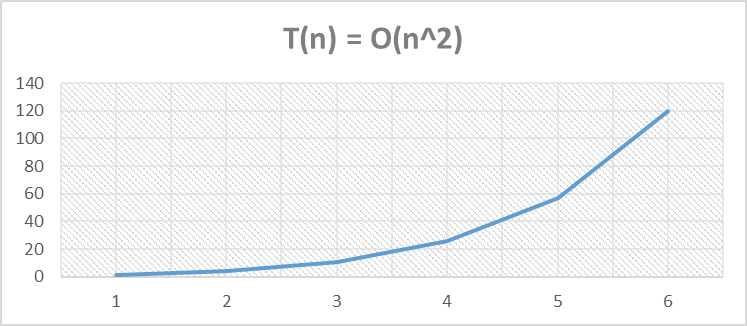
L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80]

## Time of execution

To get the running complexity and the Big-O notation, the steps were as follows:

|  |  |
| --- | --- |
| 1. 2 recalls, so a = 2. 2. Size of the list: n, so T(n-1) 3. Extra work: n (because of the for loop) | T(n) = aT(f(n)) + g(n)  T(n) = 1(T(n-1)) + n |
| T(n) = (2) T(n-1) + n   * T(1) = 1 * T(2) = * T(3) = T(2) + 3 = 3 + 3 = 6 * T(4) = T(3) + 4 = 6 + 4 = 10 * T(5) = T(4) + 5 = 10 + 5 = 15 | Assumption: T(n) = (n+1/2)\*n  1 \* 2/2 = 1  2 \* 3/2 = 3  3 \* 4/2 = 6  4 \* 5/2 = 10  5 \* 6/2 = 15 Guess is correct!  **T(n) = O(n2)** |

Running time: 1553658874.7526617 ns

****

## Return the minimum element in the tree at a given depth d

## Proposed solution design and implementation

For this function, first the assumption was to add a counter to add 1 every time than the recursive function was call with the next child but was remove after due was inefficient and not very easy to implement, and a lot of error. So for the given depth, the function calls for the next child[0] and depth was subtracted 1, and the base case was if depth is equal to zero (if depth==0), then return the element 0 of the item (T.item[0]), working with the property of the B-trees to have the minimum element to the farthest left. You can see the experiments in the next table:

|  |  |  |  |
| --- | --- | --- | --- |
| Experiments | | Proposed Solution | Return the minimum |
| **#** | **Changes** | **Description / Assumption** | **Results** |
| 1 | count =0 … count+=1 | adding a counter when reaches to the same depth, prints only the number in that depth |  |
| 2 | If depth is 0 | return the root | return the root, testing still to get the first assumption to work |
| 3 |  | based on the Print function, if count = depth then prints the number |  |
| 4 |  | Delete for loop to only print the first element in the depth | error |
| 5 |  | if the depth is greater than zero, so the next step is to call the child and print the first item. Increase count by 1 | Only work with depth = 1, not the next numbers |
| 6 |  | Remove count variable. Modify the function to print T.item[0] only when depth = 0. | Print the T.item[0] |
| 7 | MininDepth(T.child[0], depth-1) | if depth is not 0 when call, recursion call with T.child[0] and depth -1 | Print the first element in that depth |

## Experimental results

The function was tested using different depth (0,1,2) to get the minimum depth in each one and works perfectly.

Code

def MininDepth(T,depth):

if depth==0:

print(T.item[0])

else:

MininDepth(T.child[0],depth-1)

depth = 2

print("3) Minimum number in depth ", depth, ":", end=' ')

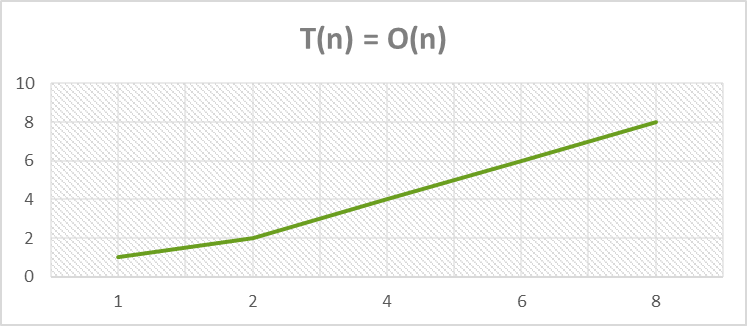
MininDepth(T,depth)

## Time of execution

To get the running complexity and the Big-O notation, the steps were as follows:

|  |  |
| --- | --- |
| 1. 1 recalls, so a = 1. 2. Size of the list: n, so T(n-1) 3. Recursive at the end, +1 (because of the for loop) | T(n) = aT(f(n)) + g(n)  T(n) = 1(T(n-1)) + 1 |
| T(n) =   * T(1) = 1 * T(2) = 2 * T(4) = 4 * T(6) = 6 * T(8) = 8 | Assumption: T(n) = n  **T(n) = O(n)** |

Running time: 1553716966.7570636 ns



# Return the maximum element in the tree at a given depth d

## Proposed solution design and implementation

Following the methodology done in the last function, it would also subtract 1 every time the recursive function is call, and the base case is if depth==0 then print the last time (print(T.item[len(T.item)-1])). The experiments are in the next table:

|  |  |  |  |
| --- | --- | --- | --- |
| Experiments | | Proposed Solution | Return the maximum |
| **#** | **Changes** | **Description / Assumption** | **Results** |
| 1 | print(T.item[len(T.item)]) | Trying to print the first number when depth is 0 (based on the MininDepth function | Error |
| 2 | print(T.item[len(T.item)-1]) | Prints the first element |  |
| 3 | MaxinDepth(T.child[ len(T.child)-1], depth-1) | Modify to call the last child | Works! |

## Experimental results

The function was tested using different depth (0,1,2) to get the minimum depth in each one and works perfectly, and tested with two different list (converted to two B-trees)

## Code

def MaxinDepth(T,depth):

if depth==0:

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

else:

MaxinDepth(T.child[len(T.child)-1],depth-1)

depth = 2

print("4) Maximum number in depth ", depth, ":",end=' ',)

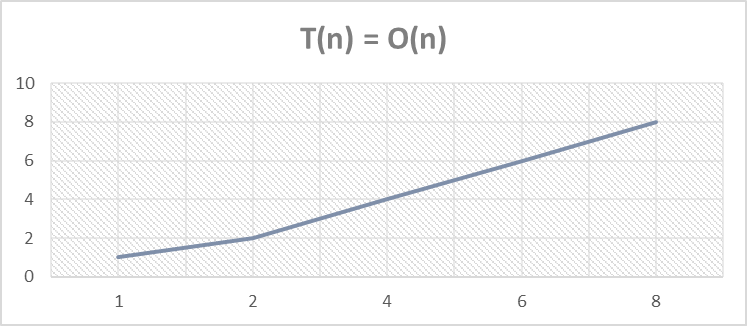
MaxinDepth(T,depth)

## Time of execution

To get the running complexity and the Big-O notation, the steps were as follows:

|  |  |
| --- | --- |
| 1. 1 recalls, so a = 1. 2. Size of the list: n, so T(n-1) 3. Recursive at the end, +1 (because of the for loop) | T(n) = aT(f(n)) + g(n)  T(n) = 1(T(n-1)) + 1 |
| T(n) =   * T(1) = 1 * T(2) = 2 * T(4) = 4 * T(6) = 6 * T(8) = 8 | Assumption: T(n) = n  **T(n) = O(n)** |

Running time: 1553719978.12486



# Return the number of nodes in the tree at a given depth d

## Proposed solution design and implementation

Based on the MininDepth function (print the smallest element in the given depth) and the Print function, this time was added a variable (sum) to count all the nodes. The base case was that if the depth is zero (if depth==0) then return sum and sum the len of the items (return sum + len(T.item). Else, for loop to sum and call the function to the next child, with depth-1, and at the end return the sum. Experiments on the next table:

|  |  |  |  |
| --- | --- | --- | --- |
| Experiments | | Proposed Solution | Return the number of nodes |
| **#** | **Changes** | **Description / Assumption** | **Results** |
| 1 | return sum + len(T.item) | Based on the MininDepth function, modify print statement to print the length of the child | Error |
| 2 | print(len(T.child[1].item)) | Testing with len(T.item) gives the length of the root, and len(T.child) the length of the children (of the current root) | Print the number of items in the child[i] |
| 3 | sum += NodesinDepth (T.child[i],depth-1) | Test with sum of the length with a for loop | Gives the sum in the current depth |

## Experimental results

To test the function, was used two different list, and change the depth value to confirm it works.

## Code

def NodesinDepth(T,depth):

sum = 0

if depth==0:

return sum + len(T.item)

else:

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

sum += NodesinDepth(T.child[i],depth-1)

return sum

depth = 2

print("5) Number of nodes in depth ", depth,":",end=' ')

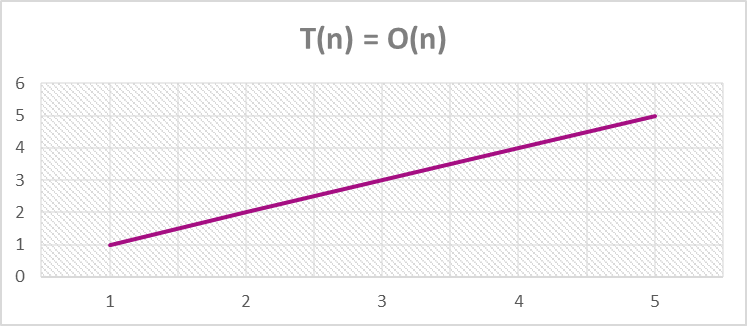
print(NodesinDepth(T,depth))

## Time of execution

To get the running complexity and the Big-O notation, the steps were as follows:

|  |  |
| --- | --- |
| 1. 1 recalls, so a = 1. 2. Size of the list: n, so T(n-1) 3. Recursive at the end, +1 | T(n) = aT(f(n)) + g(n)  T(n) = 1(T(n-1)) + n |
| T(n) = T(n-1) + 1   * T(1) = 1 * T(2) = 2 * T(3) = 3 * T(4) = 4 * T(5) = 5 | Assumption: T(n) = n  **T(n) = O(n)** |

Running time: 1553723000.011303 ns



# Print all the items in the tree at a given depth d

## Proposed solution design and implementation

Also based on the Print function and the previous one, the sum variable was no longer needed, it prints the items when depth==0.

|  |  |  |  |
| --- | --- | --- | --- |
| Experiments | | Proposed Solution | Print all the items in the tree at a given depth d |
| **#** | **Changes** | **Description / Assumption** | **Results** |
| 1 |  | Based on the lasrt function, delete the sum variable and place print in the base case | Works with depth = 0, but fails with a greater number |
| 2 |  | Change recursion call to the fucntion | Works! |

## Experimental results

To test the function, was used two different list, and change the depth value to confirm it works.

## Code

def PrintinDepth(T,depth):

if depth==0:

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

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

else:

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

PrintinDepth(T.child[i],depth-1)

depth = 1

print("6) Numbers in depth ", depth,":",end=' ')

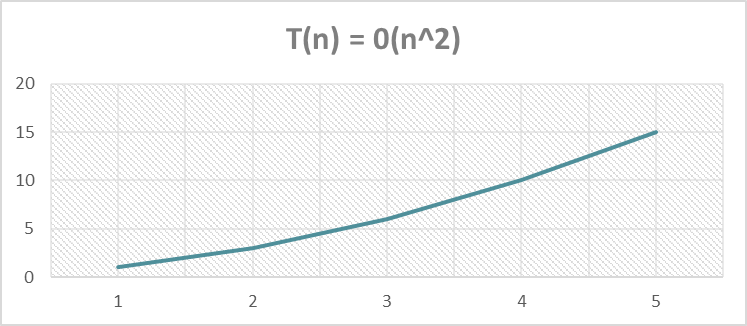
PrintinDepth(T,depth)

## Time of execution

To get the running complexity and the Big-O notation, the steps were as follows:

|  |  |
| --- | --- |
| 1. 1 recalls, so a = 1. 2. Size of the list: n, so T(n-1) 3. Recursive at the end, +n (because of the for loop) | T(n) = aT(f(n)) + g(n)  T(n) = 1(T(n-1)) + n |
| T(n) = T(n-1) + n   * T(1) = 1 * T(2) = 3 * T(3) = 6 * T(4) = 10 * T(5) = 15 | Assumption: T(n) = (n+1/2)\*n  1 \* 2/2 = 1  2 \* 3/2 = 3  3 \* 4/2 = 6  4 \* 5/2 = 10  5 \* 6/2 = 15 Guess is correct!  **T(n) = O(n2)** |

Running time: 1553724551.7838714 ns



# Return the number of nodes in the tree that are full

## Proposed solution design and implementation

Again, added a sum variable to sum the nodes in the tree that are full, used IsFull(T) function. Based on the CreateList(T,list) function, been similar and having the same base case (if T.isLeaf:) it returns the sum since have reach to the end and it’s looking only for the nodes. Else, starts by verifying that the current T (current, because later is used T.child[0] as T); if it is full, adds 1 to the sum and then checks to the next child by a for loop and the results are added to the sum variable, at the end returns the sum. Experiments on the next table:

|  |  |  |  |
| --- | --- | --- | --- |
| Experiments | | Proposed Solution | Return the number of nodes in the tree that are full |
| **#** | **Changes** | **Description / Assumption** | **Results** |
| 1 | print(IsFull(T.child[1].child[2])) | Test IsFull function to see how it works | Returns True or False if the T (T.child[i]) is full |
| 2 |  | Work with the assumption that the function is similar to CreateList. | Same base case |
| 3 | if IsFull(T): sum +=1 for i in range(len(T.child)): | sum is T is full and then move to the next child | sums and return the sum |

## Experimental results

To test the code, used two list (of different lengths) and added more items to get multiple full nodes.

## Code

def Nodesfull(T):

sum = 0

if T.isLeaf:

return sum

else:

if IsFull(T):

sum +=1

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

sum += Nodesfull(T.child[i])

return sum

print()

print("7) Number of nodes full: ", end=' ')

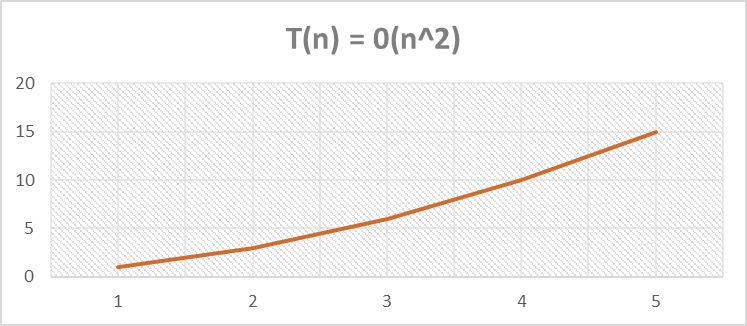
print(Nodesfull(T))

## Time of execution

To get the running complexity and the Big-O notation, the steps were as follows:

|  |  |
| --- | --- |
| 1. 1 recalls, so a = 1. 2. Size of the list: n, so T(n-1) 3. Recursive at the end, + n (because of the for loop) | T(n) = aT(f(n)) + g(n)  T(n) = 1(T(n-1)) + 1 |
| T(n) = T(n-1) + n   * T(1) = 1 * T(2) = 3 * T(3) = 6 * T(4) = 10 * T(5) = 15 | Assumption: T(n) = (n+1/2)\*n  1 \* 2/2 = 1  2 \* 3/2 = 3  3 \* 4/2 = 6  4 \* 5/2 = 10  5 \* 6/2 = 15 Guess is correct!  **T(n) = O(n2)** |

Running time: 1553724589.6788173 ns



# Return the number of leaves in the tree that are full

## Proposed solution design and implementation

Since the IsFull(T) function doesn’t work with leaves, the assumed implementation was that if the next item is a leaf them check that the current T.child[i] is full, if not then go to the next child until the next “child” is a leaf. So, based on the last function. The difference was that the if statement is inside the for loop and checks first that the next item is a leaf and it is, then checks if is full and adds 1 to the sum variable.

|  |  |  |  |
| --- | --- | --- | --- |
| Experiments | | Proposed Solution | Return the number of leaves in the tree that are full. |
| **#** | **Changes** | **Description / Assumption** | **Results** |
| 1 | if T.isLeaf: return sum | Based on NodesFull, has to check before reaches to the leaf | just returns the sum, to end |
| 2 | if T.child[i].isLeaf: | checks if the next child are leaves | since IsFull doesn't work with items (T.item) then checks better the next leaf |
| 3 | for i in range(len(T.child)):   if T.child[i].isLeaf:  if IsFull(T.child[i]):  sum +=1 | for loop where checks first if the next element is a leaf. I9f true then is it's full. If true, add 1 to the sum variable. | checks next is a leaf and it's full |
| 4 | sum +=Leafsfull(T.child[i]) | At the end, sums with the return variable from all the recursion calls | sum 1 if it's a leaf |

## Experimental results

The same test was the last one, using two list (converted to B-trees) and adding item to the list to have multiple full leafs.

## Code

def Leafsfull(T):

sum = 0

if T.isLeaf:

return sum

else:

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

if T.child[i].isLeaf:

if IsFull(T.child[i]):

sum +=1

sum +=Leafsfull(T.child[i])

return sum

print("8) Number of leafs full: ", end=' ')

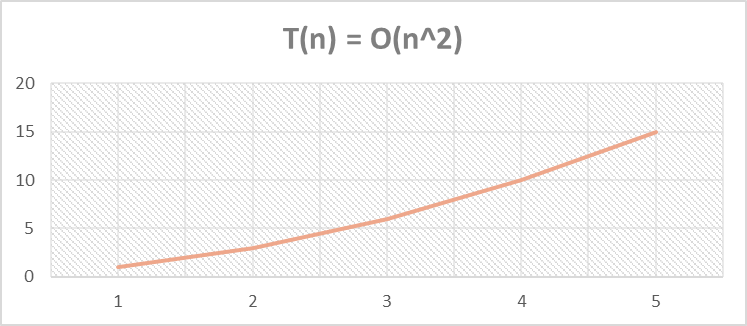
print(Leafsfull(T))

## Time of execution

To get the running complexity and the Big-O notation, the steps were as follows:

|  |  |
| --- | --- |
| 1. 1 recalls, so a = 1. 2. Size of the list: n, so T(n-1) 3. Recursive at the end, +n (because of the for loop) | T(n) = aT(f(n)) + g(n)  T(n) = 1(T(n-1)) + 1 |
| T(n) = T(n-1) + n   * T(1) = 1 * T(2) = 3 * T(3) = 6 * T(4) = 10 * T(5) = 15 | Assumption: T(n) = (n+1/2)\*n  1 \* 2/2 = 1  2 \* 3/2 = 3  3 \* 4/2 = 6  4 \* 5/2 = 10  5 \* 6/2 = 15 Guess is correct!  **T(n) = O(n2)** |

Running time: 1553725476.5649529 ns



# Given a key k, return the depth at which it is found in the tree, of -1 if k is not in the tree

## Proposed solution design and implementation

Based on the LeafsFull function, first checks that the key is on the b-tree using the Search function. If not returns -1, is true then continues, and this is the base case. Then if the k is in T.item, return the depth, if it’s in the root, depth is already 0 and returns 0, if not, then increases the depth variable (int variable, depth =0) on the for loop and calls the the function plus adding the results to the depth variable. At the endm returns the depth.

|  |  |  |  |
| --- | --- | --- | --- |
| Experiments | | Proposed Solution | Given a key k, return the depth at which it is found in the tree, of -1 if k is not in the tree |
| **#** | **Changes** | **Description / Assumption** | **Results** |
| 1 | elif k in T.item: return depth | Based on the LeafsFull function, depth increases if the key is not on the first node (base case) | If the key is in the root, return depth = 0 |
| 2 | node = Search(T,k) | Added function Search(T,k) to look first if the key is in the key, if not return -1 | Checks first and returns -1 if the key is not on the tree |
| 3 | depth += DepthK(T.child[i],k) | for loop with the statement to add the next depth | continues to the next child and return the depth where the key is |

## Experimental results

The key variable was modified to test the results.

## Code

def DepthK(T,k):

node = Search(T,k)

depth = 0

if node is None:

return -1

elif k in T.item:

return depth

else:

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

depth +=1

depth += DepthK(T.child[i],k)

return depth

print("9) Key found in depth: ", end=' ')

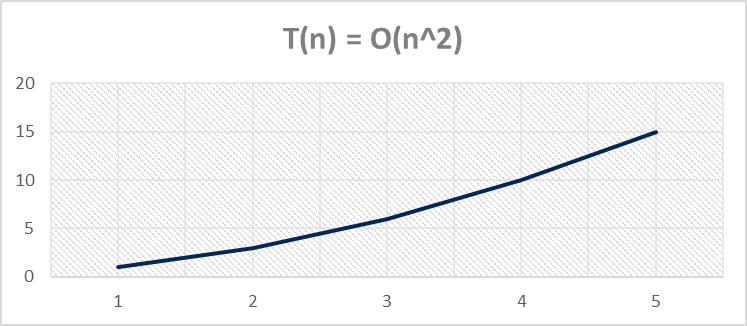
print(DepthK(T,60))

## Time of execution

To get the running complexity and the Big-O notation, the steps were as follows:

|  |  |
| --- | --- |
| 1. 1 recalls, so a = 1. 2. Size of the list: n, so T(n-1) 3. Recursive at the end, +n (because of the for loop) | T(n) = aT(f(n)) + g(n)  T(n) = 1(T(n-1)) + 1 |
| T(n) = T(n-1) + n   * T(1) = 1 * T(2) = 3 * T(3) = 6 * T(4) = 10 * T(5) = 15T(4) = * T(6) = * T(8) = | Assumption: T(n) = (n+1/2)\*n  1 \* 2/2 = 1  2 \* 3/2 = 3  3 \* 4/2 = 6  4 \* 5/2 = 10  5 \* 6/2 = 15 Guess is correct!  **T(n) = O(n2)** |

Running time: 1553716966.7570636 ns



# Conclusions

The implementation of the variables (like the sum variable) made the implementation of the function more difficult, but making those mistakes was easier to understand how could the function can be designed in a different way, since I was looking to simplify the code.

The B-tree were a little difficult to understand because I was used to the bst structure but has a lot of advantages.

# Appendix

"""

Course: Data Structures CS2302

Author: Laura Berrout

Assignment: Lab #2

Instructor: Dr. Olac Fuentes

T.A.:

Date of last modification: 03/15/2019

Purpose: Write functions to perform several operations with b-trees

"""

#Taken from code by Dr. Olac Fuentes

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)

L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11 , 3, 4, 5,105, 115, 200, 2, 45, 6,300,301,7,8,9,10,11,12,13,106,107,108]

#L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80]

T = BTree()

for i in L:

#print('Inserting',i)

Insert(T,i)

#PrintD(T,'')

#Print(T)

#print('\n####################################')

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* My code \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# 1. Compute the height of the tree

def Heightbt(T):

if T.isLeaf:

return 0

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

PrintD(T,'')

print("1) Height of b-tree: ",Heightbt(T))

print()

# 2. Extract the items in the B-tree into a sorted list.

Sortedlist = [] #create list to add the elements of the b-tree

def Createlist(T,list):

if T.isLeaf:

for t in T.item:

list.append(t)

else:

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

Createlist(T.child[i],list)

list.append(T.item[i])

Createlist(T.child[len(T.item)],list)

return list

print("2) List from the B-tree:")

print(Createlist(T,Sortedlist))

print()

#3. Return the minimum element in the tree at a given depth d

#if depth is greater than height, return than depth is greater than the height of the b-tree

def MininDepth(T,depth):

if depth==0:

print(T.item[0])

else:

MininDepth(T.child[0],depth-1)

depth = 2

print("3) Minimum number in depth ", depth, ":", end=' ')

MininDepth(T,depth)

print()

# 4. Return the maximum element in the tree at a given depth d.

def MaxinDepth(T,depth):

if depth==0:

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

else:

MaxinDepth(T.child[len(T.child)-1],depth-1)

depth = 2

print("4) Maximum number in depth ", depth, ":",end=' ',)

MaxinDepth(T,depth)

print()

# 5. Return the number of nodes in the tree at a given depth d.

def NodesinDepth(T,depth):

sum = 0

if depth==0:

return sum + len(T.item)

else:

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

sum += NodesinDepth(T.child[i],depth-1)

return sum

depth = 2

print("5) Number of nodes in depth ", depth,":",end=' ')

print(NodesinDepth(T,depth))

print()

# 6. Print all the items in the tree at a given depth d.

def PrintinDepth(T,depth):

if depth==0:

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

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

else:

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

PrintinDepth(T.child[i],depth-1)

depth = 1

print("6) Numbers in depth ", depth,":",end=' ')

PrintinDepth(T,depth)

print()

# 7. Return the number of nodes in the tree that are full.

def Nodesfull(T):

sum = 0

if T.isLeaf:

return sum

else:

if IsFull(T):

sum +=1

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

sum += Nodesfull(T.child[i])

return sum

print()

print("7) Number of nodes full: ", end=' ')

print(Nodesfull(T))

print()

# 8. Return the number of leaves in the tree that are full.

#if the next is a leaf then check if is full

#if not, then go to the next child

elapsed\_time = time.time()

def Leafsfull(T):

sum = 0

if T.isLeaf:

return sum

else:

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

if T.child[i].isLeaf:

if IsFull(T.child[i]):

sum +=1

sum +=Leafsfull(T.child[i])

return sum

print("8) Number of leafs full: ", end=' ')

print(Leafsfull(T))

print("Elapsed time: ", elapsed\_time)

print()

# 9. Given a key k, return the depth at which it is found in the tree, of -1 if k is not in the tree

def DepthK(T,k):

node = Search(T,k)

depth = 0

if node is None:

return -1

elif k in T.item:

return depth

else:

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

depth +=1

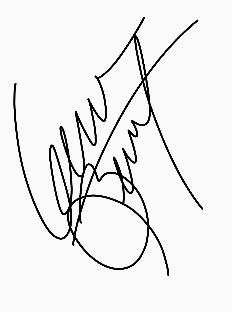
depth += DepthK(T.child[i],k)

return depth

print("9) Key found in depth: ", end=' ')

print(DepthK(T,60))

print()



Laura Berrout, March 15, 2019

“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.”