Lab 3: Binary Search Trees

CS 2303 Data Structures

Laura Berrout

ID# 80607326

March 8, 2019

# Introduction

The program *bst.py* in the class webpage contains implementations of basic binary search tree operations, including insertion, deletion, search and display.

1. Display the binary search tree as a figure, as shown below:



1. Iterative version of the search operation.
2. Building a balanced binary search tree given a sorted list as input. Note: this should not use the insert operation, the tree must be built directly from the list in O(n) time.
3. Extracting the elements in a binary search tree into a sorted list. As above, this should be done in O(n) time.
4. Printing the elements in a binary tree ordered by depth. The root has depth 0, the root’s children have depth one, and so on. For example, for the tree in the figure, your program should output:

1) Display the binary search tree as a figure

## Proposed solution design and implementation

## Experimental results

## Code

## Time of execution

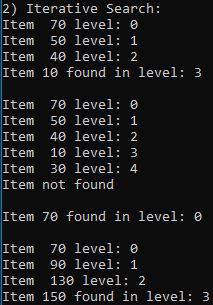
2) Iterative version of the search operation.

## Proposed solution design and implementation

Using the FindandPrint() function, the base case is call when the item is not found and the function reached at the end of the function and returns -1. Since *k* is the key, print the *T.item* if *k* is equal and returns the *level*. The *level* was incorporated to the function properties (Search(T,K,level) because of the later recursion call and level is added 1. If the key is not equal to T.item (which means is not on the root), then goes to the left if the key is less than the root, or right otherwise. Prints the current “root” and does a recursion call. Then again looks for the key in the root and makes the comparison (greater or lesser) to locate the key. When found, prints the current level, and the “root” (T.item).

## Experimental results

With Search(T,10,0), key = 10, starting level = 0, and bst T, looks for 10 in the bst T. Later the function was tested with 11, 70 and 150. The results are shown on the next figure:



## Code

def Search(T,k,level):

if T is None:

print("Item not found")

return -1

if T.item==k:

print("Item", T.item,"found in level:" ,level)

return level

if T.item<k:

print("Item ", T.item,"level:" ,level)

return Search(T.right,k,level+1)

else:

print("Item ", T.item,"level:" ,level)

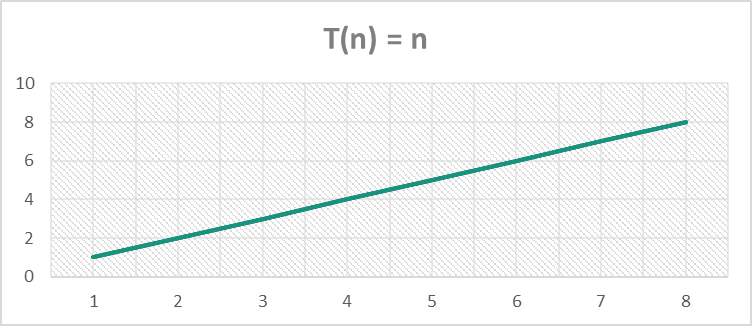
return Search(T.left,k,level+1)

## Time of execution

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

|  |  |
| --- | --- |
| 1. 1 recall, so a = 1 (is either one or the other) 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: 0.015620708465576172 ns



3) Building a balanced binary search tree given a sorted list as input. Note: this should not use the insert operation, the tree must be built directly from the list in O(n) time.

## Proposed solution design and implementation

Fist the base case looks for the length in the list, that if it is equal to 1, if yes then return. Else, calculates the middle of the list and creates two list, the left list which elements are from 0 to the mid element from the original list, and right that are from mid+1 to the end of the original list. Now, is created a tree, been the mid element the root, and the left and right children are created by a recursion call, been the list now left or right list, and T, T.left or T.right, and at the end returns the T.

## Experimental results

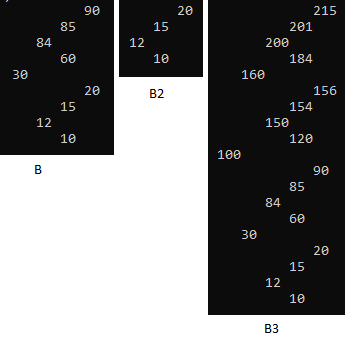
Was tested with the next lists:

B = [10, 12, 15, 20, 30, 60, 84, 85, 90,100]

B2 = [10, 12, 15, 20, 30]

B3 = [10, 12, 15, 20, 30, 60, 84, 85, 90,100, 120, 150, 154, 156, 160, 184, 200, 201, 215, 230]

And the results are shown on the next figure:



## Code

def BalanceTree(List,T):

if len(List)==1:

return

else:

num = len(List)

middle = List[num//2-1]

left = List[0:(num//2)]

right = List[(num//2):len(List)]

T= BST(0)

T.item = middle

T.left = BalanceTree(left,T.left)

T.right = BalanceTree(right,T.right)

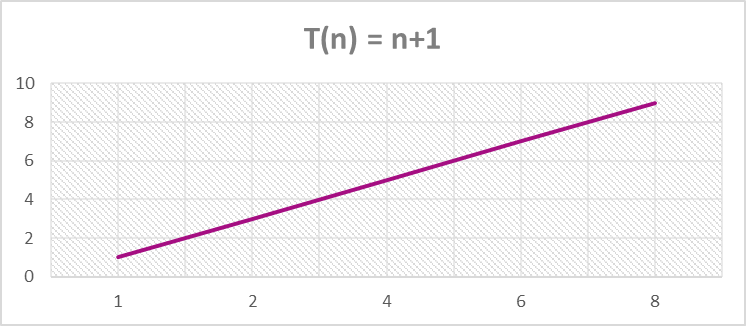
return T

## Time of execution

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

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

Running time: 0.015623807907104492 ns



4) Extracting the elements from a tree to a sorted list

## Proposed solution design and implementation

Created another bst (Tother) to have an empty bst. Then get the tree from the new tree created on the last function (from list to tree) so we have already a balanced tree. Then create a empty list (C=[]). For the function, the base case is if T is None then return list (“base case” since the function if is “if T is not None:). Else, recursion call to the left child, add the T.item to the list, then recursion call to the right child. Following a pre-order technique, first add the left children, then the root and at last the right children, and that way results in a sorted list.

## Experimental results

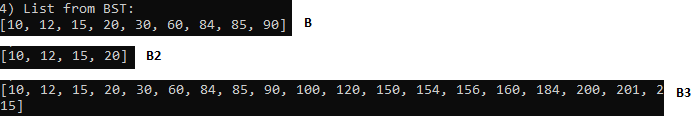
Was tested with the other list created to test the last function, to get the same sorted list:

B = [10, 12, 15, 20, 30, 60, 84, 85, 90,100]

B2 = [10, 12, 15, 20, 30]

B3 = [10, 12, 15, 20, 30, 60, 84, 85, 90,100, 120, 150, 154, 156, 160, 184, 200, 201, 215, 230]

And the results are shown on the next figure:



## Code

def Createlist(T,list):

if T is not None:

Createlist(T.left,list)

list.append(T.item) #insert root

Createlist(T.right,list)

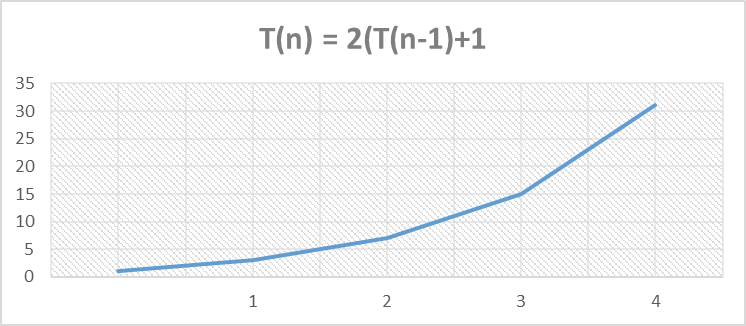
return list

## Time of execution

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

|  |  |
| --- | --- |
| 1. 2 recall, so a = 2. 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) = 2(T(n-1)) + 1 |
|  | **T(n) = O(2^n)** |

Running time: 0 ns



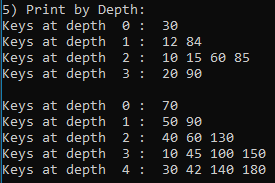
5) Print the elements ordered by depth

## Proposed solution design and implementation

First was created another function to print only the items on the given depth, and the main function to print the items by depth, and a third function to get the length of the bst. The base case for the main function (PrintbyDepth) is when it reaches the last depth and returns. The base case for the function PrintinDepth returns when T is None, as the LenTree function, except that returns the lent. PrintinDepth prints only when the depth has reach to 0, since the recursion call decreases by 1 the given depth, and it’s call for both left and right children. The PrintbyDepth just call the PrintinDepth function and goes to the next depth (i+1).

## Experimental results

Was tested with two trees and the results are shown on the next figure:



## Code

def PrintinDepth(T,depth):

if T is None:

return

else:

if depth==0:

print(T.item,end=' ')

else:

PrintinDepth(T.left,depth-1)

PrintinDepth(T.right,depth-1)

def PrintbyDepth(T,i):

lent = LenTree(T)

if i>lent:

return

else:

print("Keys at depth ", i,": ",end=' ')

PrintinDepth(T,i)

print()

PrintbyDepth(T,i+1)

def LenTree(T):

lent = 0

if T is None:

return lent

else:

lent+=1

lent +=LenTree(T.left)

#LenTree(T.right)

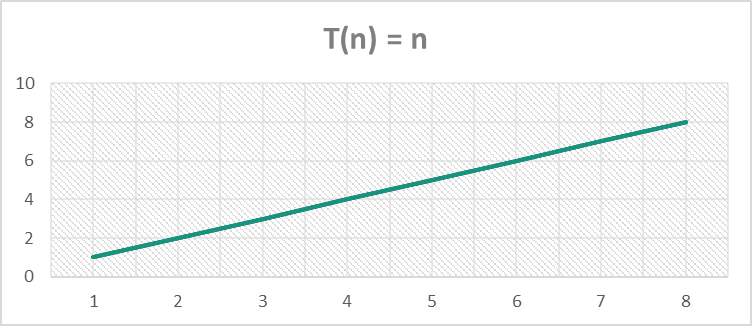
return lent

## 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(n) = O(n)** |

Running time: 0 ns



# Appendix

## Source code:

"""

Course: Data Structures CS2302

Author: Laura Berrout

Assignment: Lab #3

Instructor: Dr. Olac Fuentes

T.A.:

Date of last modification: 03/11/2019

Purpose: Binary Search Tree

"""

import time

class BST(object):

# Constructor

def \_\_init\_\_(self, item, left=None, right=None):

self.item = item

self.left = left

self.right = right

def Insert(T,newItem):

if T == None:

T = BST(newItem)

elif T.item > newItem:

T.left = Insert(T.left,newItem)

else:

T.right = Insert(T.right,newItem)

return T

def Delete(T,del\_item):

if T is not None:

if del\_item < T.item:

T.left = Delete(T.left,del\_item)

elif del\_item > T.item:

T.right = Delete(T.right,del\_item)

else: # del\_item == T.item

if T.left is None and T.right is None: # T is a leaf, just remove it

T = None

elif T.left is None: # T has one child, replace it by existing child

T = T.right

elif T.right is None:

T = T.left

else: # T has two chldren. Replace T by its successor, delete successor

m = Smallest(T.right)

T.item = m.item

T.right = Delete(T.right,m.item)

return T

def InOrder(T):

# Prints items in BST in ascending order

if T is not None:

InOrder(T.left)

print(T.item,end = ' ')

InOrder(T.right)

def InOrderD(T,space):

# Prints items and structure of BST

if T is not None:

InOrderD(T.right,space+' ')

print(space,T.item)

InOrderD(T.left,space+' ')

def SmallestL(T):

# Returns smallest item in BST. Returns None if T is None

if T is None:

return None

while T.left is not None:

T = T.left

return T

def Smallest(T):

# Returns smallest item in BST. Error if T is None

if T.left is None:

return T

else:

return Smallest(T.left)

def Largest(T):

if T.right is None:

return T

else:

return Largest(T.right)

def Find(T,k):

# Returns the address of k in BST, or None if k is not in the tree

if T is None or T.item == k:

return T

if T.item<k:

return Find(T.right,k)

return Find(T.left,k)

def FindAndPrint(T,k):

f = Find(T,k)

if f is not None:

print(f.item,'found')

else:

print(k,'not found')

# Code to test the functions above

T = None

A = [70, 50, 90, 130, 150, 40, 10, 30, 100, 180, 45, 60, 140, 42]

for a in A:

T = Insert(T,a)

#InOrderD(T,'')

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* My Code \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#1) Display the bst as the figure

#2) iterative search

def Search(T,k,level):

if T is None:

print("Item not found")

return -1

if T.item==k:

print("Item", T.item,"found in level:" ,level)

return level

if T.item<k:

print("Item ", T.item,"level:" ,level)

return Search(T.right,k,level+1)

else:

print("Item ", T.item,"level:" ,level)

return Search(T.left,k,level+1)

print("2) Iterative Search: ")

Search(T,10,0)

print()

Search(T,11,0)

print()

Search(T,70,0)

print()

start = time.time()

Search(T,150,0)

print()

#3) Build a balanced tree

B = [10, 12, 15, 20, 30, 60, 84, 85, 90,100]

B2 = [10, 12, 15, 20, 30]

B3 = [10, 12, 15, 20, 30, 60, 84, 85, 90,100, 120, 150, 154, 156, 160, 184, 200, 201, 215, 230]

def BalanceTree(List,T):

if len(List)==1:

return

else:

num = len(List)

middle = List[num//2-1]

left = List[0:(num//2)]

right = List[(num//2):len(List)]

T= BST(0)

T.item = middle

T.left = BalanceTree(left,T.left)

T.right = BalanceTree(right,T.right)

return T

Tnew = BST(0)

print("3) Balanced BST from list: ")

InOrderD(BalanceTree(B,Tnew),' ')

print()

InOrderD(BalanceTree(B2,Tnew),' ')

print()

InOrderD(BalanceTree(B3,Tnew),' ')

print()

#4) Extracting the elements from a tree to a sorted list

Tother = BST(0)

Tother = BalanceTree(B,Tnew)

C = []

def Createlist(T,list):

if T is not None:

Createlist(T.left,list)

list.append(T.item) #insert root

Createlist(T.right,list)

return list

print("4) List from BST: ")

print(Createlist(Tother,C))

print()

#5) Print the elements ordered by depth

def PrintinDepth(T,depth):

if T is None:

return

else:

if depth==0:

print(T.item,end=' ')

else:

PrintinDepth(T.left,depth-1)

PrintinDepth(T.right,depth-1)

def PrintbyDepth(T,i):

lent = LenTree(T)

if i>lent:

return

else:

print("Keys at depth ", i,": ",end=' ')

PrintinDepth(T,i)

print()

PrintbyDepth(T,i+1)

def LenTree(T):

lent = 0

if T is None:

return lent

else:

lent+=1

lent +=LenTree(T.left)

#LenTree(T.right)

return lent

print("5) Print by Depth: ")

#PrintinDepth(T,3)

PrintbyDepth(Tother,0)

print()

start = time.time()

PrintbyDepth(T,0)

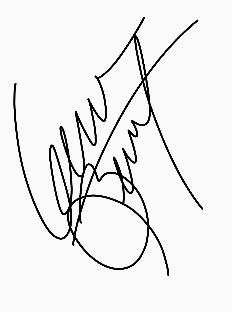
end = time.time()

print("Time: ",end-start)

print()

#print("Length: ",LenTree(T))

#InOrderD(T,' ')



Laura Berrout, March 8, 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.”