=============================================================================

CSC 263 Tutorial 3 Winter 2019

=============================================================================

Suppose we want to use a Binary Search Tree to store only keys (without any

additional information), and we want to allow duplicate keys. For example, if

we insert the key 12 twice, the BST should behave exactly as if there were

two separate copies of the key 12 stored in the tree.

1. The algorithm that we covered in lecture for TreeInsert does not handle

this situation right now: it explicitly disallows duplicate keys. Modify

the TreeInsert algorithm from lecture so that duplicate keys can be

inserted: treat equal keys the same way as larger keys (or smaller keys

-- just pick one and use it consistently, to keep the algorithm simple).

Write the complete algorithm and point out the changes you made. (Yes,

this involves mostly copy-and-pasting the algorithm from class and making

small changes -- it is meant to make you write down the complete

algorithm and understand it, as preparation for the later questions.)

Then, give a careful worst-case analysis of the running time of your

algorithm when it is used to insert n identical keys into a BST that is

initially empty.

This is a little different from what we've done up until now: we don't

want you to analyse the complexity of just one TreeInsert operation;

we want you to analyse the \*total\* time taken by \*all\* n calls to

TreeInsert.

1. Algorithm:  
  
        TreeInsert(root, x):  
            if root is NIL:  
                root <- TreeNode(x)  
            elif x.key < root.key:  
                root.left <- TreeInsert(root.left, x)  
            else:  
                root.right <- TreeInsert(root.right, x)  
            return root  
  
    Modifications:  
        Removed the second "elif" so that the algorithm recurses on the right  
        subtree when x.key >= root.key.  
  
    Runtime:  
        The worst-case running time of one call to TreeInsert on a BST with m  
        keys is Theta(height). When all keys are equal, this is Theta(m): the  
        BST ends up with a linear chain of nodes (going down the right side  
        only).  
  
        To make the calculation simpler, say the running time for each call  
        is exactly Cm, for some constant C. When inserting n identical keys,  
        the total running time of all n calls to TreeInsert is therefore  
        equal to:  
            C\*0 + C\*1 + ... + C\*(n-1) = C\*n(n-1)/2 = Theta(n^2).

In the rest of this tutorial, you will explore different ways that you can

improve on the running time of TreeInsert when duplicate keys are allowed.

2. First strategy: **ensure duplicate keys are not always inserted on the same**

**side.** Store a boolean flag goLeft in each node, initially set to True.

During insertion, when the key to be inserted is equal to the current

node's key, use the value of the current node's goLeft to determine

whether to insert the key in the left subtree or in the right subtree,

and flip the value of the current node's goLeft.

Write the complete algorithm and point out changes from your answer to

question 1.

Then, give a careful worst-case analysis of the running time of your

algorithm when it is used to insert n identical keys into a BST that is

initially empty. As before, analyse the \*total\* time taken by \*all\* n

calls to TreeInsert.

answer:

2. Algorithm:  
  
        TreeInsert(root,x):  
            if root is NIL:  
                root <- TreeNode(x)  
            elif x.key < root.key:  
                root.left <- TreeInsert(root.left, x)  
            elif x.key > root.key:  
                root.right <- TreeInsert(root.right, x)  
            else:  
                if root.goLeft:  
                    root.left <- TreeInsert(root.left, x)  
                else:  
                    root.right <- TreeInsert(root.right, x)  
                root.goLeft <- not root.goLeft  
            return root  
  
    Modifications:  
        Added back the second "elif" to handle x.key > root.key, and code to  
        use goLeft to determine which subtree to recurse into when x.key =  
        root.key.  
  
    Runtime:  
        The worst-case running time of one call to TreeInsert on a BST with m  
        keys is Theta(height). When all keys are equal, this is Theta(log m):  
        each node has subtrees of roughly equal size (+/-1) because insertion  
        alternates strictly between left and right children.  
  
        To make the calculation simpler, say the running time for each call  
        is exactly C log(m+1), for some constant C (the "+1" is to simplify  
        the case m = 0). When inserting n identical keys, the total running  
        time of all n calls to TreeInsert is therefore equal to:  
            C log1 + ... + C logn = C SUM\_{i=1}^n log i.  
        We can bound this expression as follows:  
            C SUM\_{i=1}^n log\_2 i <= C SUM\_{i=1}^n log n  
                                   = C n log n  
            C SUM\_{i=1}^n log\_2 i >= C SUM\_{i=n/2}^n log (n/2)  
                                   = C (n/2) log(n/2)  
                                   = (C/2) n (logn - 1)  
        Hence, the total is Theta(n log n).

3. Second strategy: during insertion, when the key to be inserted is equal

to the current node's key, determine whether to insert in the left

subtree or the right subtree **at random -- with equal probabilities** for

left and right.

Write the complete algorithm and point out changes from your answer to

question 1. Then, analyse the worst-case performance of your algorithm

when it is used to insert n identical keys into a BST that is initially

empty, as before (this should be quick).

Finally, give an informal analysis of the expected running time of your

algorithm.

answer:

3. Algorithm:  
  
        TreeInsert(root,x):  
            if root is NIL:  
                root <- TreeNode(x)  
            elif x.key < root.key:  
                root.left <- TreeInsert(root.left, x)  
            elif x.key > root.key:  
                root.right <- TreeInsert(root.right, x)  
            else:  
                if random(0,1] <= 0.5:  
                    root.left <- TreeInsert(root.left, x)  
                else:  
                    root.right <- TreeInsert(root.right, x)  
            return root  
  
    Modifications:  
        Same as question 2 except using random choices for equal keys.  
  
    Runtime:  
        The worst case degenerates to the same as question 1 (Theta(n^2)),  
        when all random choices are the same.  
        The expected case becomes the same as question 2, because on average  
        we expect the choices to be roughly equally balanced between left and  
        right at each node.

4. Can you come up with a better strategy? There are at least two simple

ideas that will work better for the particular case we are considering

(inserting n identical keys in an initially empty BST).

Describe your strategy in one concise paragraph, then write the complete

algorithm (point out changes from previous parts) and give a careful

worst-case analysis of your algorithm when it is used to insert n

identical keys into a BST that is initially empty (as before).

answer:

4. Idea:  
        Each node stores a "count" equal to the number of times that an  
        identical key has been inserted.  
  
    Algorithm:  
  
        TreeInsert(root,x):  
            if root is NIL:  
                root <- TreeNode(x)  # with root.count <- 1  
            elif x.key < root.key:  
                root.left <- TreeInsert(root.left, x)  
            elif x.key > root.key:  
                root.right <- TreeInsert(root.right, x)  
            else:  
                root.count <- root.count + 1  
            return root  
  
    Modifications:  
        When x.key = root.key, simply increment root's counter.  
  
    Runtime:  
        The worst-case running time of one call to TreeInsert on a BST with m  
        keys is Theta(height). When all keys are equal, this is Theta(1):  
        only one node is ever used.  
  
        To make the calculation simpler, say the running time for each call  
        is exactly C, for some constant C. When inserting n identical keys,  
        the total running time of all n calls to TreeInsert is therefore  
        equal to Cn = Theta(n).

    Alternative idea:  
        Use a linked chain of separate nodes for equal keys -- so each node  
        has a left child (for smaller keys), a right child (for larger keys),  
        and a middle child (for equal keys). Insertion in the middle chain is  
        done by adding at the front of the chain so it takes constant time.