CSC263H Tutorial 3

Problem Set

Winter 2024

Work on these exercises *before* the tutorial. You don't have to come up with a complete solution, but you should be prepared to discuss them with your TA. We encourage you to work on these problems in groups of 2-3.

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 $BSTInsert$ does not handle this situation right now: it explicitly disallows duplicate keys. Modify the $BSTInsert$ 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 $BSTInsert$ operation; we want you to analyse the total time taken by all n calls to $BSTInsert$.

   Solutions:

```
def BSTInsert(key, D):
1   if D is empty:
2       D.root.key = key
3   elif D.root.key >= key:
4       BSTInsert(D.left, key)
5   else:
6       BSTInsert(D.right, key)
```

   The only changed is on the branching condition on line 3 so that duplicate keys are added to the left sub-tree. The worst case runtime complexity is $O(n^2)$ for inserting n identical keys since each new key is inserted into the left most leaf. The steps of steps required for inserting the first key is 1, second key is 2 ..., and nth key is n. In total $(1 + n) * n/2$ steps are required.

Improving $BSTInsert$

In the rest of this tutorial, you will explore different ways that you can improve on the running time of $BSTInsert$ 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 $BSTInsert$.

   Solutions:

```
def BSTInsert(key, D):
1   if D is empty:
2       D.root.key = key
3   if D.root.key == key:
4      if D.root.goLeft:
5            BSTInsert(D.left, key)
6      else:
7            BSTInsert(D.right, key)
8      D.root.goLeft = ! D.root.goLeft
9   elif D.root.key > key:
10       BSTInsert(D.left, key)
11   else:
12      BSTInsert(D.right, key)
```

   The changes are on line 3 - 8 that handles the cases when duplicate keys are inserted. If n identical keys are inserted, the tree will not grow to depth i unless all previously depth are completely filled. Therefore, inserting the ith key takes $log_2(i)$ steps, and inserting all n keys takes $\sum_{i=1}^{n} i * log_2 i$ steps, which is $O(nlog(n))$

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).

   Solutions:

```
def BSTInsert(key, D):
1   if D is empty:
2       D.root.key = key
3   if D.root.key == key:
4      if randombit():
5           BSTInsert(D.left, key)
6      else:
7          BSTInsert(D.right, key)
8   elif D.root.key > key:
9       BSTInsert(D.left, key)
10   else:
11      BSTInsert(D.right, key)
```

The changes are on line 3 - 7 that handles the cases when duplicate keys are inserted. If n identical keys are inserted, the worst case runtime is when all keys are inserted to the same direction. The result is the same as Q1, $O(n^2)$.

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: Idea: Suppose a node n is being inserted. If a node m with the same key is discovered in the BST, then n inherits m's children and then is added as a left child of m.

   Solutions:

```
def BSTInsert(key, D):
1   if D is empty:
2       D.root.key = key
3   if D.root.key == key:
4      old_left, old_right = D.root.key.left, D.root.key.right
5       BSTInsert(D.left, key)
6       D.left.left = old_left
7       D.left.right = old_right
8   elif D.root.key > key:
9       BSTInsert(D.left, key)
10   else:
11      BSTInsert(D.right, key)
```

   The changes are between line 3 - 7 where we insert a node with key as the left child of an existing node with the same key. The worst-case runtime complexity for inserting n identical keys is $O(n)$ since each inserted node will be added as a left child of the root.