Augmented Binary Search Trees

Jin Long Cao, Ethelia Choi

October 2022

Implementation of an Augmented BST

Recall the recursive procedures Tree-Insert and Tree-delete for BSTs, from lecture. Unlike our text, CLRS, these procedures do **not** use a parent pointers (references).

Assume that each BST node r has no parent pointer but **does** have a field **r.count** meant to record the number of nodes in the tree rooted at r.

Modify Tree-Insert and Tree-delete as recursive python procedures that correctly insert (respectively delete) nodes into (respectively from) BSTs and maintain **r.count**. Carefully explain why your procedures are correct and have the same asymptotic worst-case runtime as the versions given in lecture.

Solutions

TREE-INSERT(root, x):

```
1. if root is NIL:    # x.key not already in S
2.       # Found insertion point: create new node with empty children.
3.       root = TreeNode(x)
4.       root.count = 1
5. elif x.key < root.item.key:
6.       if TREE-SEARCH(root.left, x.key) is NIL:    # NIL if x does not exist in the subtree.
7.            root.left.count += 1
8.       root.left = TREE-INSERT(root.left, x)
9. elif x.key > root.item.key:
10.      if TREE-SEARCH(root.right, x.key) is NIL:    # NIL if x does not exist in the subtree.
11.           root.right.count += 1
12.      root.right = TREE-INSERT(root.right, x)
13. else:    # x.key == root.item.key
14.      root.item = x    # just replace root's item with x
15. return root
```

Before inserting, assume count for all nodes are already maintain.

Case 1: Some element y in the tree has y.key = x.key, then we replace y with x and r.count does not change for any node. TREE-SEARCH will return y (which means it will not return NIL), therefore none of the counts (for any node) will increase (line 7 and 11). The function will recursively run (line 8 or 12) until it reaches to the y element then line 14 will run, replacing y with x.

Case 2: The element we're inserting does not exist in the tree already and is added as a leaf (line 1 to 4). All the ancestor of that leaf has their count increased by 1. TREE-SEARCH will return NIL, therefore anytime it recursively TREE-INSERT (line 8 or 12), it will increase the count by one for the parent node (line 7 and 11). Once it found it's insertion point, it will create a node and set that node's count to 1.

In both cases, the TREE-SEARCH runtime will $O(\lg n)$ and the asymptotic worst-case runtime for this TREE-INSERT is $O(\lg n + \lg n) = O(2 \cdot \lg n) = O(\lg n)$, which is the same as the versions given in lecture.

TREE-DELETE(root, x):

```
1. if root is NIL: # x.key not in S −− should not happen!
2.     pass # nothing to remove
3. elif x.key < root.item.key:
4.     if TREE-SEARCH(root.left, x.key) is not NIL:    # NIL if x does not exist in the subtree.
5.         root.count -= 1
6.     root.left = TREE-DELETE(root.left, x)
7. elif x.key > root.item.key:
8.     if TREE-SEARCH(root.right, x.key) is not NIL:    # NIL if x does not exist in the subtree.
9.         root.count -= 1
10.    root.right = TREE-DELETE(root.right, x)
11. else: # x.key == root.item.key
12.    # Remove root.item.
13.    if root.left is NIL:
14.        # Missing left child (at least): replace with right child.
15.        # descendants nodes' count don't change
16.        root.count = root.right.count
17.        root = root.right # NIL if both children missing
18.    elif root.right is NIL:
19.        # Missing right child: replace with left child.
20.        # descendants nodes' count don't change
21.        root.count = root.left.count
22.        root = root.left
23.    else:
24.        # Root has two children: remove element with smallest key in
25.        # right subtree and move it to root (replace root with successor).
26.        root.count = root.left.count + root.right.count
27.        root.item, root.right = TREE-DELETE-MIN(root.right)
28. return root
```

TREE-DELETE-MIN(root):

```
29. # Remove element with smallest key in root's subtree; return item and
30. # root of resulting subtree.
31. if root.left is NIL:
32.    # Root stores item with smallest key; replace it with right child.
33.    return root.item, root.right
34. else:
35.    # Left subtree not empty: root not the smallest.
36.    root.count -= 1
37.    item, root.left = TREE-DELETE-MIN(root.left)
38.    return item, root
```

Case 1: TREE-DELETE is deleting an element that doesn't exist in the tree. This case should not happen (as stated in the pseudo-code) but should still be taken care of. Since the element does not exist in the tree, TREE-SEARCH will return NIL. If the element does not exist, it will recursively run (line 6 or 10) until "root is NIL" (line 1) and just pass. None of the counts (for any node) will change due to the fact that TREE-SEARCH is NIL (line 4 or 8).

Case 2: TREE-DELETE is deleting an element that does exist in the tree. TREE-SEARCH will not return NIL. Every ancestor of the element getting deleted will decrement their count by 1 and any descendants will keep the same count. The code will recursively run (line 6 or 10) and decrement the count for each ancestor (line 5 or 9) until it finds the element (line 11). Once the element is found, we replace it with one of their children (and their count) if the other one is NIL. If both children exist, we set the current node count to the sum of both children and find the successor to replace to it. Since the successor is replacing it, all ancestors of the successor node that is also a descendant of the deleted node will decrement their count (line 36).

In both cases, the TREE-SEARCH runtime will $O(\lg n)$ and the asymptotic worst-case runtime for this TREE-DELETE is $O(\lg n + \lg n) = O(2 \cdot \lg n) = O(\lg n)$, which is the same as the versions given in lecture.