CS 381 HW 3

Ankur Dhoot

September 21, 2016

Q1

We'll give an algorithm that runs in O(lgn). The basic idea is as follows: To compute the sum of all key values greater than low, but less than high, we can compute the sum of all key values with keys less than high, compute the sum of all key values with keys less than or *equal* to low, and take the difference. It's clear that this difference is the sum of all key values with keys greater than low, but less than high.

Algorithm 1 Compute total of key values with key $\leq$ LO in the tree rooted at x

```
function SUM-LEQ(key LO, node x)
    if x == T.nil then return 0
    end if
    if x.key ≤ LO then
        return 1 + x.left.total + SUM-LEQ(x.right)
    else
        return SUM-LEQ(x.left)
    end if
end function
```

Algorithm 2 Compute total of key values with key $<$ HI in the tree rooted at x

```
function SUM-LT(key HI, node x)
    if x == T.nil then return 0
    end if
    if x.key < HI then
        return 1 + x.left.total + SUM-LT(x.right)
    else
        return SUM-LT(x.left)
    end if
end function
```

Algorithm 3 Compute total of all key values with key greater than LO, but less than HI, in tree T

```
function RANGE-TOTAL(key LO, key HI, Tree T)
    return SUM-LT(HI, T.root) - SUM-LEQ(LO, T.root)
end function
```

As in the book, we make the assumption that any leaf (T.nil) has total (T.nil.total) of 0. Correctness for finding the sum of key values $\leq$ LO is as follows:
Base Case: Leaf node has no internal nodes, so we return 0.

Otherwise, if the current node x has x.key $\leq$ LO, we know that x and all nodes in the left subtree of x have keys $\leq$ LO. Thus, we add 1 (for x) + x.left.total (key value sum of left subtree) + SUM-LEQ(x.right) (to compute sum of values of keys $\leq$ LO in the right subtree of x)

Else (current node has x.key $>$ LO), so we know that neither x nor any nodes in the right subtree of x have keys $\leq$ LO, so we recurse on the left subtree.

Analagous reasoning holds for SUM-LT, with $\leq$ replaced with $<$.

Since, we call the two recursive functions with T.root, we get the sum of key values in the whole tree with keys $<$ HI and keys $\leq$ LO, respectively. The difference returns the sum we desire.

As promised, the recursive algorithm runs in O(lgn). Each call to SUM-LEQ does constant work before calling itself recursively with x having moved down one level in the tree. Since a RB tree has O(lgn) levels, and we're doing a constant amount of work in each level, SUM-LEQ takes O(lgn) time. Similarly, for SUM-LT. Thus, our total is O(2lgn) = O(lgn).

Alternatively, we can give a single recursive method as follows:

Algorithm 4 Single method that can be used more generically

```
function TOTAL(key K, node x, Comparator comp)
    if x == T.nil then return 0
    end if
    if comp(x.key, K) then
        return 1 + x.left.total + Total(x.right)
    else
        return Total(x.left)
    end if
end function
```

In this new method, we pass in a comparator which would be the $\leq$ comparator or the $<$ comparator.

Thus, Total(HI, T.root, $<$) - Total(LO, T.root, $\leq$) will give us our desired sum and clearly with the same runtime.

Q2

(a) Assuming the size attribute has been set correctly (i.e if a node has count 2, it contributes 2 to the size of its parent)(which is part of a correct insertion algorithm for part (b)), the code changes are very few. The authors give the function OS-SELECT(x, i) for selecting the ith smallest key (based on inorder walk) in the subtree rooted at x. We'll modify OS-SELECT to take into account our new count attribute.

Algorithm 5 Return ith smallest element in subtree rooted at x (incorporate count attribute)

```
1:  function OS-SELECT(node x, int i)
2:      r = x.left.size + x.count
3:      if x.left.size + 1 ≤ i ≤ r then
4:          return x
5:      else if i < x.left.size + 1 then
6:          return OS-Select(x.left, i)
7:      else
8:          return OS-Select(x.right, i-r)
9:      end if
10: end function
```

Correctness is as follows:
Line 2 computes the number of elements(not necessarily the same as the number of nodes, since a node can have the same element multiple times now) up to and including x.key. If i lies between x.left.size + 1 and x.left.size + x.count, this means the ith smallest value in the subtree rooted at x is x.key.
Otherwise, if i < x.left.size + 1, the ith smallest element must be located in the left subtree of x, so we recurse on x.left.
Otherise (i > r), so the ith smallest element in the subtree rooted at x must be located in the right subtree of x. Thus, we recurse on x.right, except we look for the (i-r)th smallest element in x.right since we've already eliminated r elements less that i.

(b) When inserting a key, some minor modifications are necessary to the code given in the book. The book already describes how to maintain the size attribute (14.1), so we'll describe how to maintain the count attribute. In 13.3, the code given for RB-Insert can be modified inside the while loop:

In the modified while loop, we check if we're at a node with the same key. If so, we increment the count for that node and return. We don't continue to progress down the tree until we hit a leaf since we don't want to add a new node if the key already exists in the tree. As we progress along a path from the root, we increment the size attribute of every node as well to maintain

Algorithm 6 modified while loop for RB-Insert

```
while x ≠ T.nil do
    x.size += 1
    y = x
    if x.key == z.key then
        x.count += 1
        return
    else if z.key < x.key then
        x = x.left
    else
        x = x.right
    end if
end while
```

proper size counts. When inserting, we assume that the node being inserted, z, has z.count and z.size initialized to 1. This way, if z.key does not exist in the tree, we'll add z as a leaf with the proper count and size attributes. Everything else in RB-Insert remains the same. No changes are necessary to RB-Insert-Fixup, Left-Rotate, or Right-Rotate to maintain the count attribute since these operations only move around nodes, but the count of the node should remain the same. However, moving around nodes requires modifications to the size attribute of certain nodes, and this modification is given in the book. (Note that if the key already exists, RB-Insert-Fixup, Left-Rotate, Right-Rotate won't even be called since the tree will still be balanced after insertion of an existing key and we return inside the while loop) Thus, we've managed to maintain the count and size attributes during insertion with only minor code modifications.