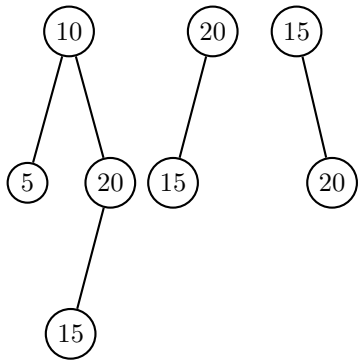
Illinois Institute of Technology
Department of Computer Science

Homework 3 Solutions

CS 430 Introduction to Algorithms
Fall Semester, 2010

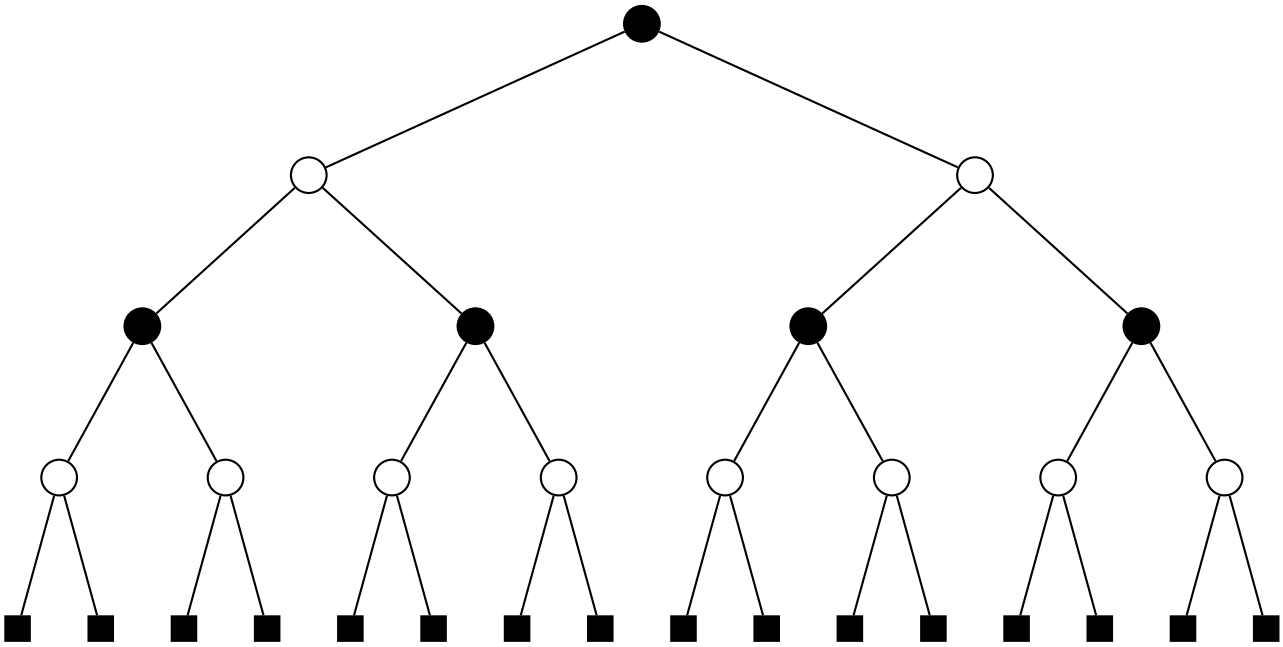
1. **Problem 12.3-4 on page 299**

   Solution: It is not commutative. Consider the leftmost tree below. If you delete 5 and then 10 from the leftmost tree, you end up with the middle tree. If you delete 10 and then 5 from the leftmost tree, you end up with the rightmost tree.



2. **Problem 13.1-7 on page 312**.

   Solution: The maximum ratio of red internal nodes to black internal nodes happens when the tree is *complete* and has even height (all leaves have even depth and root has depth zero). All nodes with odd depth are red, those with even depths are black. All non-leaf nodes in the lowest level are red, which results in the maximum possible red internal nodes. At each odd depth the number of red nodes is twice that of the black nodes in the previous level. This results in a ratio of two red internal nodes to every black internal node, which is the maximum. The following is an example of such a tree with height of four.



The smallest possible ratio of red internal nodes to black internal nodes is 0. This can be achieved by letting all nodes in this tree be black.

3. **Problem 13.3-5 on page 322**.

   Solution: When inserting a new node into a tree with at least a root, we color it red and give it two black leaves. If this node does not violate any of the red-black tree properties, then we have a red node in the tree. If there is a violation, it will be a violation to the red rule (rule 4, if parent is red then children are black) then we can fix it using one of three cases.

   (a) **Case 1** - When the new red node's uncle is red, number of red nodes decreases by one. There must be at least three red nodes for the operation to begin and therefore there will be at least one red node after the operation.

   (b) **Case 2** - When the new red node is a right child and its uncle is black, we only do a left rotation to make it a case 3 and this does not change the number of red nodes.

   (c) **Case 3** - When the new red node is a left child and its uncle is black, we do a right rotation and the symmetric color change and this do not change the number of red nodes.

   After each case we have at least one red node. Thus we can see that after inserting $n > 1$ nodes into an empty tree we always have at least one red node.

4. **Problem 13.4-1 on page 330**.

   Solution:

   When we examine RB-DELETE-FIXUP we see that the loop starting at line 1 terminates when x is the root of the tree or when x is red. Clearly if the loop terminates when x reaches the root, line 23 ensures that it will be black. Therefore, the only concern is if the loop terminates because x is red and is not the root. This situation is only possible when a descendant of the root is deleted, not the root itself. Since x is set to the root after cases 3 and 4 in RB-DELETE-FIXUP, these cases are not a concern. Case

1 is not a concern, since after executing it x is still black and it is not the root (so the loop does not terminate at this point). Furthermore, it does not change any ancestors of x beyond its grandparent and it ensure that x has a black grandparent. The loop may terminate after case 2 if x's parent was red. However, if x's parent was the root, then it is made black at line 23, otherwise the root is unaffected.

5. **Problem 14.2-1 on page 347**

   Solution: We can find the SUCCESSOR and PREDECESSOR of any node x in time $\Theta(1)$ by storing pointers to the successor and predecessor of x inside the node structure. Note that these values do not change during rotations, so we only need to show that we can maintain them during insertions and deletions. As soon as we insert a node x, we can call the regular TREE-SUCCESSOR(x) and TREE-PREDECESSOR(x) functions (that run in $O(\lg n)$) to set its fields. Once the successor and predecessor are determined, we can set the appropriate fields in x, and then update the SUCC field of the predecessor of x and the PRED field of the successor of x to point to x. When we delete a node, we update its successor and predecessor to point to each other rather than to x. To simplify the handling of border cases (e.g. calling SUCCESSOR on the largest element in the tree), it may be useful to keep a dummy node with key $-\infty$ and a dummy node with key $+\infty$.

   Finally, MINIMUM and MAXIMUM (which are global properties of the tree) may be maintained directly. When we insert a node, we check if its key is smaller than the current minimum or larger than the current maximum, and update the MINIMUM and/or MAXIMUM pointer if necessary. When we delete a node, we may need to set MINIMUM and/or MAXIMUM to its successor and/or predecessor, respectively, if we are deleting the minimum or maximum element in the tree. Thus, these operations run in $\Theta(1)$ and may be maintained without changing the asymptotic running time of other tree operations.