

---

# Data Structures and Algorithms in Java™

Sixth Edition

**Michael T. Goodrich**

Department of Computer Science  
University of California, Irvine

**Roberto Tamassia**

Department of Computer Science  
Brown University

**Michael H. Goldwasser**

Department of Mathematics and Computer Science  
Saint Louis University

---

## Instructor's Solutions Manual

WILEY

## Chapter

# 11

## Search Trees

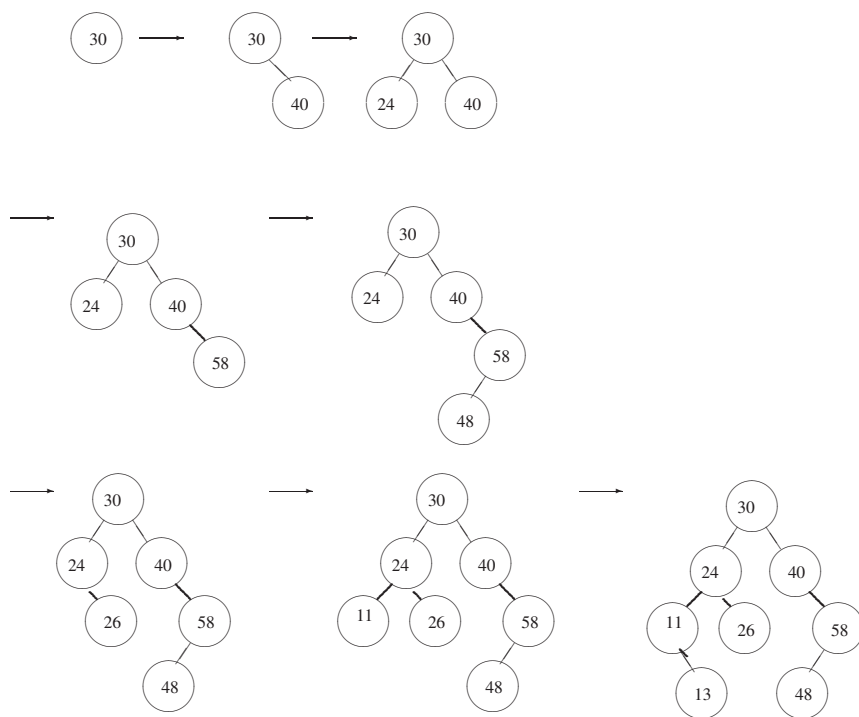
### Hints and Solutions

#### Reinforcement

**R-11.1) Hint** Recall the definition of where we perform an insertion in a binary search tree.

**R-11.2) Hint** You will need to draw 8 trees, but they are all small.

**R-11.2) Solution**



**R-11.3) Hint** You can enumerate them with pictures.

**R-11.3) Solution** 5 (2 w/ 1 as root, 1 w/ 2 as root, 2 w/ 3 as root).

**R-11.4) Hint** Try a few examples of five-entry binary search trees.

**R-11.4) Solution** There are several solutions. One is to draw the binary search tree created by the input sequence: 9, 5, 12, 7, 13. Now draw the tree created when you switch the 5 and the 7 in the input sequence: 9, 7, 12, 5, 13.

**R-11.5) Hint** Try a few examples of five-entry AVL trees.

**R-11.5) Solution** There are several solutions. One is to draw the AVL tree created by the input sequence: 9, 5, 12, 7, 13. Now draw the tree created when you switch the 5 and the 7 in the input sequence: 9, 7, 12, 5, 13.

**R-11.6) Hint** Use a loop to express the repetition

**R-11.6) Solution**

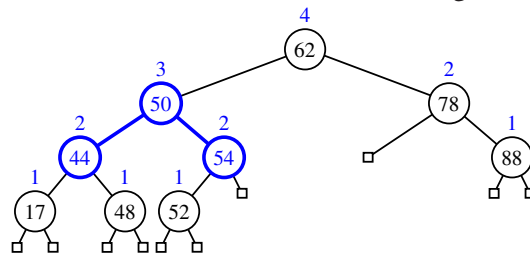
```
private Position<Entry<K,V>> treeSearch(Position<Entry<K,V>> p, K key) {
    Position<Entry<K,V>> walk = p;
    while (isInternal(walk)) {
        int comp = compare(key, walk.getElement());
        if (comp == 0)
            return walk;           // key found; return its position
        else if (comp < 0)
            walk = left(walk);
        else
            walk = right(walk);
    }
    return walk;           // unsuccessful search if this is reached
}
```

**R-11.7) Hint** There is one of each type. Which one is which?

**R-11.7) Solution** The rotation in Figure 11.11 is a double rotation. The rotation in Figure 11.13 is a single rotation.

**R-11.8) Hint** Mimic the figure in the book.

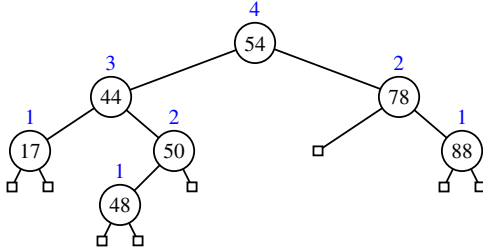
**R-11.8) Solution** The updated tree follows. The highlighted nodes (44, 50, 54) were involved in a trinode restructuring.



**R-11.9) Hint** Mimic the figure in the book.

**R-11.9) Solution** The updated tree follows. Not much has changed, as by our implementation, the root node takes on value 54 (the max of the

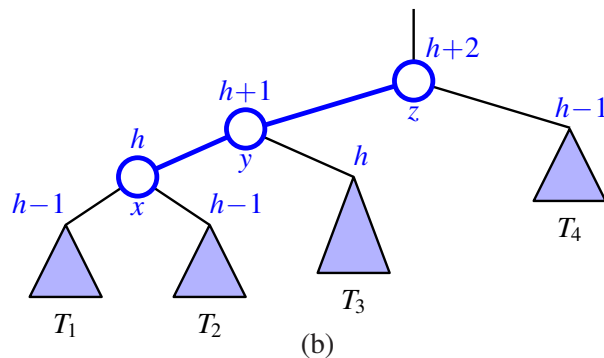
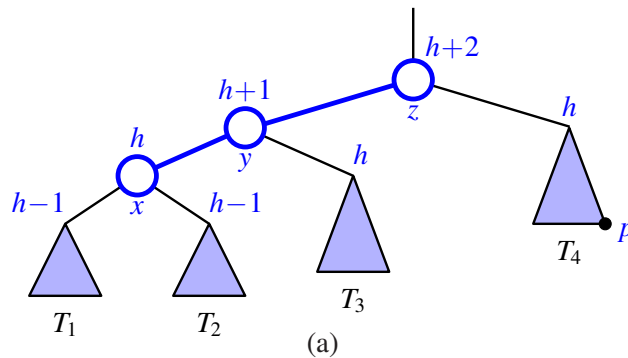
left subtree), and then the node that had held 54 is removed; however, no imbalance results from that deletion. (A more significant change would have occurred had the minimum value from the right subtree been used as a replacement for 62.)

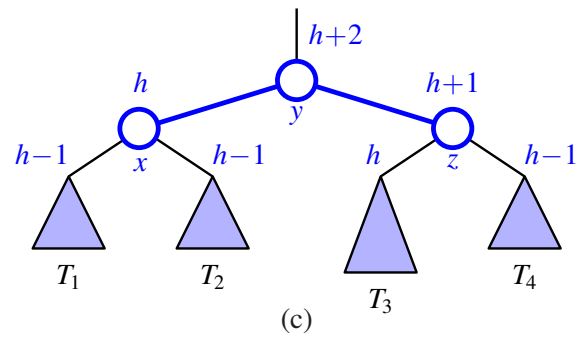


**R-11.10) Hint** Think about the data movements needed in an array list representation of a binary tree.

**R-11.11) Hint** Carefully note the heights of all subtrees before the deletion, and after the deletion but before the restructuring.

**R-11.11) Solution**

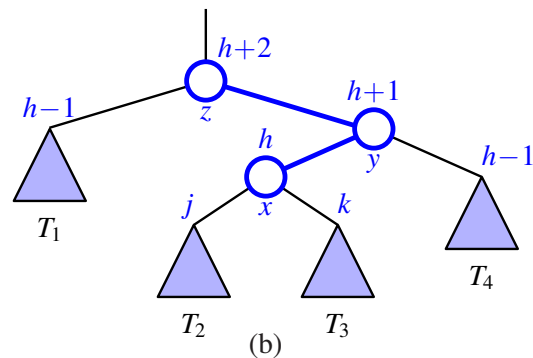
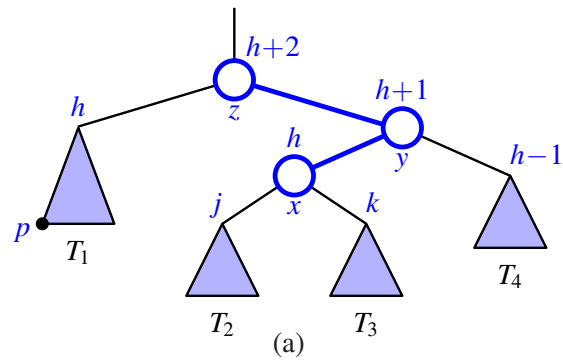


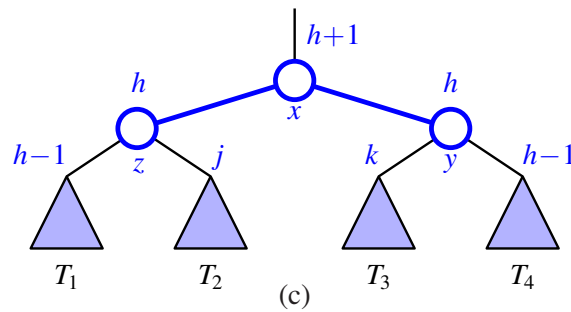


Rebalancing of a subtree during a deletion with both children of  $y$  having the same height: (a) before the deletion of  $p$ ; (b) after deletion of  $p$  but before restructuring; (c) after trinode restructuring. Note that in the end, the height of the entire subtree is unchanged, and thus no further restructuring is needed.

**R-11.12) Hint** Carefully note the heights of all subtrees before the deletion, and after the deletion but before the restructuring.

**R-11.12) Solution**





Rebalancing of a subtree during a deletion with one child of  $y$  having greater height: (a) before the deletion of  $p$ ; (b) after deletion of  $p$  but before restructuring; (c) after trinode restructuring. In this configuration, one of  $j$  and  $k$  must be  $h - 1$ , while the other is either  $h - 1$  or  $h - 2$ . In either case, notice that the height of the overall subtree has been reduced by one, and therefore the imbalance might propagate upward.

**R-11.13) Hint** Carefully trace the potential heights of various subtrees.

**R-11.13) Solution** Look back at the solution we gave for the previous problem, but assume that  $T_4$  has height  $h$ , such that the two children of  $y$  have the same height, yet the non-aligned one is chosen as  $x$ . The problem relates to the values of heights  $j$  and  $k$ . It may have been that  $j = h - 1$  and  $k = h - 2$ . But in that case, after the restructuring,  $y$  is unbalanced as it would have children with heights  $h - 2$  and  $h$ .

**R-11.14) Hint** Each entry is splayed to the root in increasing order.

**R-11.15) Hint** Use a pencil with a good eraser.

**R-11.16) Hint** Perform the splay operation on the lowest entry accessed for each operation.

**R-11.17) Hint** No. Why not?

**R-11.17) Solution** No. One property of a  $(2, 4)$  tree is that all external nodes are at the same depth. The multiway search tree of the example does not adhere to this property.

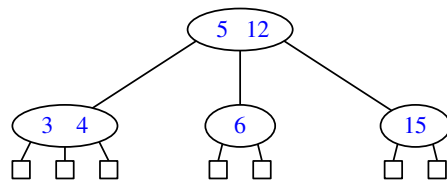
**R-11.18) Hint** It is not  $k_1$ . Why?

**R-11.18) Solution** The key  $k_2$  would be stored at  $w$ 's parent in this case. This is done in order to maintain the order of the keys within the tree. By storing  $k_2$  at  $w$ 's parent, all of the children would still be ordered such that the children to the left of the key are less than the key and those to the right of the key are greater than the key.

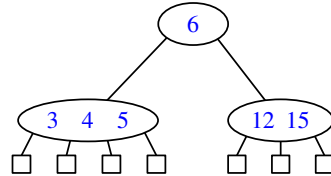
**R-11.19) Hint** You will need at list five entries to find a counterexample.

**R-11.19) Solution**

First, consider the insertion order 4, 6, 12, 15, 3, 5, which results in tree (as shown in Figure 11.26j):



Next, consider the insertion order 12, 3, 6, 4, 5, 15, which produces tree:



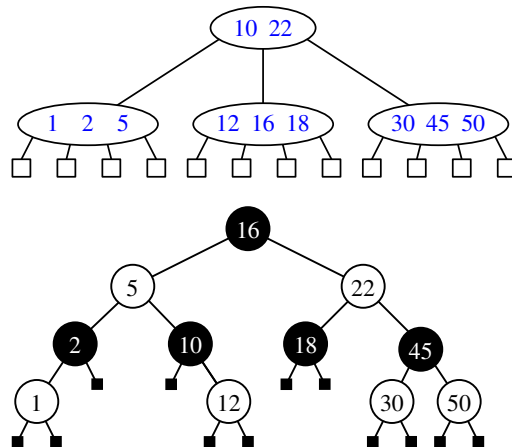
**R-11.20) Hint** Use the correspondence rules described in the chapter.

**R-11.21) Hint** Use a pencil with a good eraser.

**R-11.21) Solution** There is a  $(2,4)$  tree with 4 internal nodes having height 2, with keys  $\{4,8,12\}$  stored at the root, and four children, each holding 3 more keys. There is also a  $(2,4)$  tree with 15 internal nodes, effectively a perfectly balanced binary tree (8 is at the root).

**R-11.22) Hint** Use a pencil with a good eraser.

**R-11.22) Solution** Based on our algorithm descriptions, the results are:



**R-11.23) Hint** Consider looking at the  $(2,4)$  tree and red-black tree definitions again.

**R-11.24) Hint** Some have  $O(\log n)$  worst-case height and some have  $O(n)$  worst-case height. Make sure you know which. Also, try to get the constant factors right in this case.

**R-11.24) Solution**

- a. The worst-case height for a binary search tree is  $n$ .

- b. Based on the book's analysis, the AVL tree certainly has height no bigger than  $2\log_2 n$ . A tighter analysis shows that it is at most  $1.44\log_2 n$ .
- c. A splay tree could have height  $n$
- d. A  $(2,4)$  tree could have worst-case height of  $\log_2 n$ .
- e. A red-black tree could have worst-case height  $2\log_2 n$ .

**R-11.25) Hint** You need to create a node that does not satisfy the AVL balance condition, but would be acceptable in a red-black tree. A good example would be a tree with at least 6 nodes, but no more than 16.

**R-11.26) Hint** Case 2 is the only one that is repeated.

**R-11.26) Solution** Recall that an insertion begins with a downward search, the creation of a new leaf node, and then a potential upward effort to remedy a double-red violation. There may be logarithmically many recoloring operations due to an upward cascading of Case 2 applications, but a single application of the Case 1 action eliminates the double-red problem with a trinode restructuring. Therefore, at most one restructuring operation is needed for a red-black tree insertion.

**R-11.27) Hint** Case 2 is the only one that might be applied more than once.

**R-11.27) Solution** A deletion begins with the standard binary search-tree deletion algorithm, which requires time proportional to the height of the tree; for red-black trees, that height is  $O(\log n)$ . The subsequent rebalancing takes place along an upward path from the parent of a deleted node.

We considered three cases to remedy a resulting double black. Case 1 requires a trinode restructuring operation, yet completes the process, so this case is applied at most once. Case 2 may be applied logarithmically many times, but it only involves a recoloring of up to two nodes per application. Case 3 requires a rotation, but this case can only apply once, because if the rotation does not resolve the problem, the very next action will be a terminal application of either Case 1 or Case 2.

In the worst case, there will be  $O(\log n)$  recolorings from Case 2, a single rotation from Case 3, and a trinode restructuring from Case 1.

---

## Creativity

**C-11.28) Hint** Recall the definition of a binary search tree, in general.

**C-11.29) Hint** The method is similar to priority-queue sorting.

**C-11.30) Hint** Review what it means for splay trees to have  $O(\log n)$  amortized time performance.



**C-11.30) Solution** Yes, splay trees can sort in  $O(n \log n)$  time in the worst case: just insert all the elements and then do an inorder traversal of the splay tree. This runs in worst-case  $O(n \log n)$  time because each individual insert operation runs in  $O(\log n)$  amortized time and there are  $n$  insertions.

**C-11.31) Hint** The goal is to perform a single tree search.

**C-11.32) Hint** Show that  $O(n)$  rotations suffice to convert any binary tree into a *left chain*, where each internal node has an external right child.

**C-11.33) Hint** Where might the search path for  $k$  diverge from a path to one of the other keys?

**C-11.33) Solution** For the sake of contradiction, assume that an unsuccessful search for key  $k$  does not pass through the node with key  $k'$  which is the greatest key less than  $k$ . Note that if  $k'$  were modified to make it equal to  $k$ , the tree must still be a valid binary search tree, as this modification does not change the outcome of any comparison between two keys. Furthermore, a search for  $k$  in this modified tree must follow the same path, since we assume that path did not pass through the node with  $k'$  and thus all comparisons in the search are unchanged. Yet in this case, the search does not find  $k$ ; this violates the original correctness of the binary tree search algorithm. The rest of the argument is symmetric.

**C-11.34) Hint** Consider the maximum number of times the recursive method is called on a position that is not within the subrange.

**C-11.35) Hint** Consider a top-down recursive approach.

**C-11.36) Hint** Make sure that the result is a valid AVL tree.

**C-11.37) Hint** Note that this method returns a single integer, so it is not necessary to visit all  $s$  entries that lie in the range. You will need to extend the tree data structure, adding a new field to each node.

**C-11.37) Solution** For each node of the tree, maintain the size of the corresponding subtree, defined as the number of internal nodes in that subtree. While performing the search operation in both the insertion and deletion, the subtree sizes can be either incremented or decremented. During the rebalancing, care must be taken to update the subtree sizes of the three nodes involved (labeled  $a$ ,  $b$ , and  $c$  by the restructure algorithm).

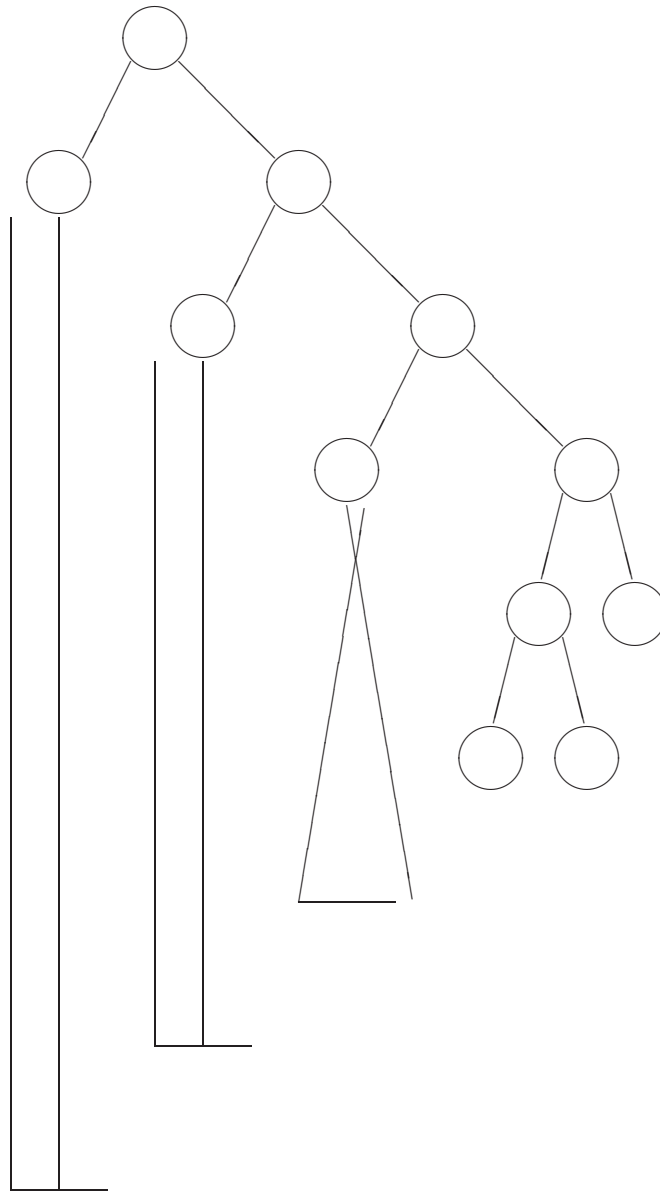
To calculate the number of nodes in a range  $(k_1, k_2)$ , search for both  $k_1$  and  $k_2$ , and let  $P_1$  and  $P_2$  be the associated search paths. Call  $v$  the last node common to the two paths. Traverse path  $P_1$  from  $v$  to  $k_1$ . For each internal node  $w \neq v$  encountered, if the right child of  $w$  is in not in  $P_1$ , add one plus the size of the subtree of the child to the current sum. Similarly, traverse path  $P_2$  from  $v$  to  $k_2$ . For each internal node  $w \neq v$  encountered, if the left child of  $w$  is in not in  $P_2$ , add one plus the size of the subtree of the left to the current sum. Finally, add one to the current sum (for the key stored at node  $v$ ).

**C-11.38) Hint** How do the rebalancing actions affect the information stored at each node?

**C-11.38) Solution** Whenever a rotation is performed, the sizes of the two affected subtrees must be recalculated. This can be implemented within `TreeMap.rotate`, so that the support is inherited by any of our balanced tree subclasses.

**C-11.39) Hint** Use triangles to represent subtrees that are not affected by this operation, and think of how to cascade the imbalances up the tree.

**C-11.39) Solution**



**C-11.40) Hint** Study closely the balance property of an AVL tree and the rebalance operation. Also, make a node high up in tree have its AVL balance depend on the node that just got inserted.

**C-11.41) Hint** Study closely the balance property of an AVL tree and the rebalance operation.

**C-11.42) Hint** Think carefully about how the balance of a node and its ancestors changes immediately after an insertion or deletion.

**C-11.43) Hint** Just consider the operations that could change the leftmost position or who points to it.

**C-11.44) Hint** How do the rebalancing actions affect the minimum?

**C-11.44) Solution** No changes are necessary. Rebalancing does not affect which node contains the smallest key.

**C-11.45) Hint** These operations will be easier if you know the size of each subtree.

**C-11.46) Hint** Is it possible for an splay tree to also be a red-black tree?

**C-11.47) Hint** Find the right place to “splice” one tree into the other to maintain the (2,4) tree property. Also, it is okay to destroy the old versions of  $T$  and  $U$ .

**C-11.47) Solution** Assume that  $T$  and  $U$  have heights  $h_t$  and  $h_u$  and  $h_t > h_u$ . Remove the smallest entry from  $U$ . Insert this entry into the rightmost node of tree  $T$  at height  $h_t - h_u - 1$ . Link this node to the root of tree  $U$ . The remove operation on  $U$  takes  $O(\log m)$  time and the insert operation on  $T$  takes  $O(\log n)$  time.

**C-11.48) Hint** Search down for  $k$  and cut along this path. Now consider how to “glue” the pieces back together in the right order.

**C-11.49) Hint** You don’t need to use induction here.

**C-11.50) Hint** Think about a way of using the structure of the binary search tree itself to indicate color.

**C-11.50) Solution** Since the nodes store keys in order, when keys are distinct, we can detect when the children’s subtrees are swapped. This indication can be used to mark nodes as red. Detecting this indication increases the search and update times by a constant factor, but the asymptotic running times remain the same.

**C-11.51) Hint** Consider the red and black meaning of the three possible balance factors in an AVL tree.

**C-11.51) Solution** All nodes should initially be black. If a node is black and its left and right sub trees have different heights, the node should be colored red (excluding the root).

**C-11.52) Hint** Since you know the node  $x$  will eventually become the root, maintain a tree of nodes to the left of  $x$  and a tree of nodes to the right of  $x$ , which will eventually become the two children of  $x$ .

**C-11.53) Hint** The analysis in the book works also for half-splay trees, with minor modifications.

**C-11.54) Hint** If you are having trouble with this problem, you may wish to gain some intuition about splay trees by “playing” with an interactive splay tree program on the Internet.

---

## Projects

**P-11.55) Hint** One of the biggest challenges is what to do for an unsuccessful tree search.

**P-11.56) Hint** Consider having an entry instance keep a reference to the node at which it is stored.

**P-11.57) Hint** We've provided the implementations; you need to develop the experiment.

**P-11.58) Hint** In this case, you will need a skip list implementation to test.

**P-11.59) Hint** The order is implicit in the tree, so adding these methods should not be hard.

**P-11.60) Hint** Use a recursive method to do the conversion.

**P-11.61) Hint** The most significant challenge is how to handle the insertion of duplicate, given that the original tree search will stop when it finds the existing key.

**P-11.62) Hint** Review the cases for zig-zag, zig-zig, and zig. Make sure you do splaying right before doing anything else.

**P-11.63) Hint** First figure out a way that works assuming that all keys in existing mergeable heaps are distinct, and then work out how this is not strictly necessary.