13 Red-Black Trees

Chapter 12 showed that a binary search tree of height h can support any of the basic dynamic-set operations—such as SEARCH, PREDECESSOR, SUCCESSOR, MINIMUM, MAXIMUM, INSERT, and DELETE—in $O(h)$ time. Thus, the set operations are fast if the height of the search tree is small. If its height is large, however, the set operations may run no faster than with a linked list. Red-black trees are one of many search-tree schemes that are "balanced" in order to guarantee that basic dynamic-set operations take $O(\lg n)$ time in the worst case.

13.1 Properties of red-black trees

A **red-black tree** is a binary search tree with one extra bit of storage per node: its **color**, which can be either RED or BLACK. By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately **balanced**.

Each node of the tree now contains the attributes *color*, *key*, *left*, *right*, and *p*. If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value NIL. We shall regard these NILs as being pointers to leaves (external nodes) of the binary search tree and the normal, key-bearing nodes as being internal nodes of the tree.

A red-black tree is a binary tree that satisfies the following **red-black properties**:

1. Every node is either red or black.

2. The root is black.

3. Every leaf (NIL) is black.

4. If a node is red, then both its children are black.

5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

Figure 13.1(a) shows an example of a red-black tree.

As a matter of convenience in dealing with boundary conditions in red-black tree code, we use a single sentinel to represent NIL (see page 238). For a red-black tree T, the sentinel $T.nil$ is an object with the same attributes as an ordinary node in the tree. Its *color* attribute is BLACK, and its other attributes— p, *left*, *right*, and *key*—can take on arbitrary values. As Figure 13.1(b) shows, all pointers to NIL are replaced by pointers to the sentinel $T.nil$.

We use the sentinel so that we can treat a NIL child of a node x as an ordinary node whose parent is x. Although we instead could add a distinct sentinel node for each NIL in the tree, so that the parent of each NIL is well defined, that approach would waste space. Instead, we use the one sentinel $T.nil$ to represent all the NILs—all leaves and the root's parent. The values of the attributes p, *left*, *right*, and *key* of the sentinel are immaterial, although we may set them during the course of a procedure for our convenience.

We generally confine our interest to the internal nodes of a red-black tree, since they hold the key values. In the remainder of this chapter, we omit the leaves when we draw red-black trees, as shown in Figure 13.1(c).

We call the number of black nodes on any simple path from, but not including, a node x down to a leaf the ***black-height*** of the node, denoted $\mathrm{bh}(x)$. By property 5, the notion of black-height is well defined, since all descending simple paths from the node have the same number of black nodes. We define the black-height of a red-black tree to be the black-height of its root.

The following lemma shows why red-black trees make good search trees.

Lemma 13.1
A red-black tree with n internal nodes has height at most $2\lg(n + 1)$.

Proof   We start by showing that the subtree rooted at any node x contains at least $2^{\mathrm{bh}(x)} - 1$ internal nodes. We prove this claim by induction on the height of x. If the height of x is 0, then x must be a leaf ($T.nil$), and the subtree rooted at x indeed contains at least $2^{\mathrm{bh}(x)} - 1 = 2^0 - 1 = 0$ internal nodes. For the inductive step, consider a node x that has positive height and is an internal node with two children. Each child has a black-height of either $\mathrm{bh}(x)$ or $\mathrm{bh}(x) - 1$, depending on whether its color is red or black, respectively. Since the height of a child of x is less than the height of x itself, we can apply the inductive hypothesis to conclude that each child has at least $2^{\mathrm{bh}(x)-1} - 1$ internal nodes. Thus, the subtree rooted at x contains at least $(2^{\mathrm{bh}(x)-1} - 1) + (2^{\mathrm{bh}(x)-1} - 1) + 1 = 2^{\mathrm{bh}(x)} - 1$ internal nodes, which proves the claim.

To complete the proof of the lemma, let h be the height of the tree. According to property 4, at least half the nodes on any simple path from the root to a leaf, not

(a)

T.nil

(b)

(c)
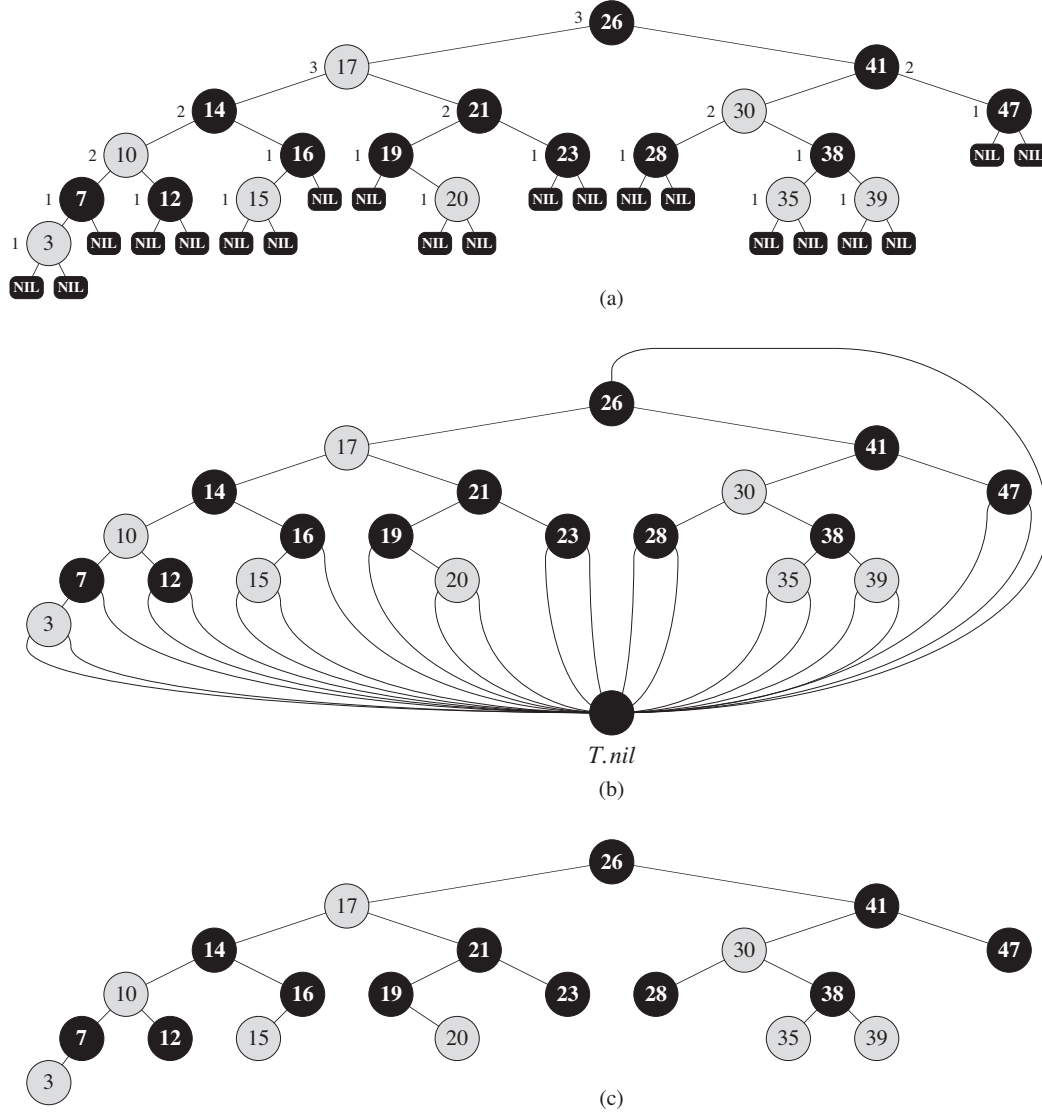
Figure 13.1   A red-black tree with black nodes darkened and red nodes shaded. Every node in a red-black tree is either red or black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. **(a)** Every leaf, shown as a NIL, is black. Each non-NIL node is marked with its black-height; NILs have black-height 0. **(b)** The same red-black tree but with each NIL replaced by the single sentinel *T.nil*, which is always black, and with black-heights omitted. The root's parent is also the sentinel. **(c)** The same red-black tree but with leaves and the root's parent omitted entirely. We shall use this drawing style in the remainder of this chapter.

including the root, must be black. Consequently, the black-height of the root must be at least $h/2$; thus,

$$n \geq 2^{h/2} - 1 \ .$$

Moving the 1 to the left-hand side and taking logarithms on both sides yields $\lg(n + 1) \geq h/2$, or $h \leq 2 \lg(n + 1)$.   ■

As an immediate consequence of this lemma, we can implement the dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR in $O(\lg n)$ time on red-black trees, since each can run in $O(h)$ time on a binary search tree of height h (as shown in Chapter 12) and any red-black tree on n nodes is a binary search tree with height $O(\lg n)$. (Of course, references to NIL in the algorithms of Chapter 12 would have to be replaced by $T.nil$.) Although the algorithms TREE-INSERT and TREE-DELETE from Chapter 12 run in $O(\lg n)$ time when given a red-black tree as input, they do not directly support the dynamic-set operations INSERT and DELETE, since they do not guarantee that the modified binary search tree will be a red-black tree. We shall see in Sections 13.3 and 13.4, however, how to support these two operations in $O(\lg n)$ time.

Exercises

13.1-1
In the style of Figure 13.1(a), draw the complete binary search tree of height 3 on the keys $\{1, 2, \ldots, 15\}$. Add the NIL leaves and color the nodes in three different ways such that the black-heights of the resulting red-black trees are 2, 3, and 4.

13.1-2
Draw the red-black tree that results after TREE-INSERT is called on the tree in Figure 13.1 with key 36. If the inserted node is colored red, is the resulting tree a red-black tree? What if it is colored black?

13.1-3
Let us define a ***relaxed red-black tree*** as a binary search tree that satisfies red-black properties 1, 3, 4, and 5. In other words, the root may be either red or black. Consider a relaxed red-black tree T whose root is red. If we color the root of T black but make no other changes to T, is the resulting tree a red-black tree?

13.1-4
Suppose that we "absorb" every red node in a red-black tree into its black parent, so that the children of the red node become children of the black parent. (Ignore what happens to the keys.) What are the possible degrees of a black node after all

its red children are absorbed? What can you say about the depths of the leaves of the resulting tree?

13.1-5

Show that the longest simple path from a node x in a red-black tree to a descendant leaf has length at most twice that of the shortest simple path from node x to a descendant leaf.

13.1-6

What is the largest possible number of internal nodes in a red-black tree with black-height k? What is the smallest possible number?

13.1-7

Describe a red-black tree on n keys that realizes the largest possible ratio of red internal nodes to black internal nodes. What is this ratio? What tree has the smallest possible ratio, and what is the ratio?
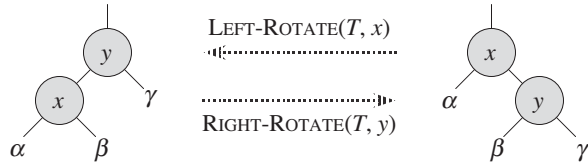
13.2   Rotations

The search-tree operations TREE-INSERT and TREE-DELETE, when run on a red-black tree with n keys, take $O(\lg n)$ time. Because they modify the tree, the result may violate the red-black properties enumerated in Section 13.1. To restore these properties, we must change the colors of some of the nodes in the tree and also change the pointer structure.

We change the pointer structure through ***rotation***, which is a local operation in a search tree that preserves the binary-search-tree property. Figure 13.2 shows the two kinds of rotations: left rotations and right rotations. When we do a left rotation on a node x, we assume that its right child y is not $T.nil$; x may be any node in the tree whose right child is not $T.nil$. The left rotation "pivots" around the link from x to y. It makes y the new root of the subtree, with x as y's left child and y's left child as x's right child.

The pseudocode for LEFT-ROTATE assumes that $x.right \neq T.nil$ and that the root's parent is $T.nil$.

Figure 13.2 The rotation operations on a binary search tree. The operation LEFT-ROTATE(T, x) transforms the configuration of the two nodes on the right into the configuration on the left by changing a constant number of pointers. The inverse operation RIGHT-ROTATE(T, y) transforms the configuration on the left into the configuration on the right. The letters α, β, and γ represent arbitrary subtrees. A rotation operation preserves the binary-search-tree property: the keys in α precede $x.key$, which precedes the keys in β, which precede $y.key$, which precedes the keys in γ.

LEFT-ROTATE(T, x)

```
 1  y = x.right              // set y
 2  x.right = y.left         // turn y's left subtree into x's right subtree
 3  if y.left ≠ T.nil
 4      y.left.p = x
 5  y.p = x.p                // link x's parent to y
 6  if x.p == T.nil
 7      T.root = y
 8  elseif x == x.p.left
 9      x.p.left = y
10  else x.p.right = y
11  y.left = x               // put x on y's left
12  x.p = y
```

Figure 13.3 shows an example of how LEFT-ROTATE modifies a binary search tree. The code for RIGHT-ROTATE is symmetric. Both LEFT-ROTATE and RIGHT-ROTATE run in $O(1)$ time. Only pointers are changed by a rotation; all other attributes in a node remain the same.
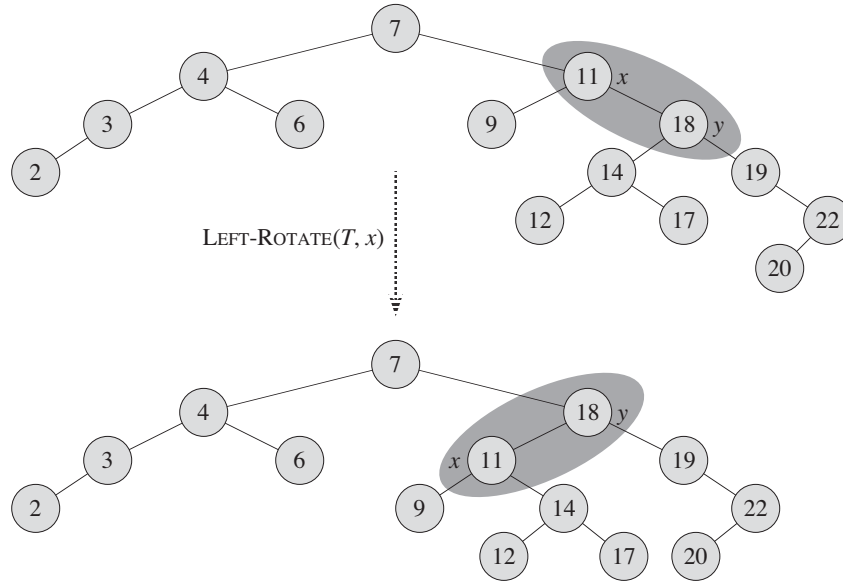
Exercises

13.2-1
Write pseudocode for RIGHT-ROTATE.

13.2-2
Argue that in every n-node binary search tree, there are exactly $n - 1$ possible rotations.

Figure 13.3   An example of how the procedure LEFT-ROTATE(T, x) modifies a binary search tree. Inorder tree walks of the input tree and the modified tree produce the same listing of key values.

13.2-3
Let a, b, and c be arbitrary nodes in subtrees α, β, and γ, respectively, in the left tree of Figure 13.2. How do the depths of a, b, and c change when a left rotation is performed on node x in the figure?

13.2-4
Show that any arbitrary n-node binary search tree can be transformed into any other arbitrary n-node binary search tree using $O(n)$ rotations. (*Hint:* First show that at most $n - 1$ right rotations suffice to transform the tree into a right-going chain.)

13.2-5   ★
We say that a binary search tree T_1 can be ***right-converted*** to binary search tree T_2 if it is possible to obtain T_2 from T_1 via a series of calls to RIGHT-ROTATE. Give an example of two trees T_1 and T_2 such that T_1 cannot be right-converted to T_2. Then, show that if a tree T_1 can be right-converted to T_2, it can be right-converted using $O(n^2)$ calls to RIGHT-ROTATE.

13.3 Insertion

We can insert a node into an n-node red-black tree in $O(\lg n)$ time. To do so, we use a slightly modified version of the TREE-INSERT procedure (Section 12.3) to insert node z into the tree T as if it were an ordinary binary search tree, and then we color z red. (Exercise 13.3-1 asks you to explain why we choose to make node z red rather than black.) To guarantee that the red-black properties are preserved, we then call an auxiliary procedure RB-INSERT-FIXUP to recolor nodes and perform rotations. The call RB-INSERT(T, z) inserts node z, whose *key* is assumed to have already been filled in, into the red-black tree T.

RB-INSERT(T, z)

```
 1   y = T.nil
 2   x = T.root
 3   while x ≠ T.nil
 4        y = x
 5        if z.key < x.key
 6             x = x.left
 7        else x = x.right
 8   z.p = y
 9   if y == T.nil
10        T.root = z
11   elseif z.key < y.key
12        y.left = z
13   else y.right = z
14   z.left = T.nil
15   z.right = T.nil
16   z.color = RED
17   RB-INSERT-FIXUP(T, z)
```

The procedures TREE-INSERT and RB-INSERT differ in four ways. First, all instances of NIL in TREE-INSERT are replaced by $T.nil$. Second, we set $z.left$ and $z.right$ to $T.nil$ in lines 14–15 of RB-INSERT, in order to maintain the proper tree structure. Third, we color z red in line 16. Fourth, because coloring z red may cause a violation of one of the red-black properties, we call RB-INSERT-FIXUP(T, z) in line 17 of RB-INSERT to restore the red-black properties.

RB-INSERT-FIXUP(T, z)

```
 1   while z.p.color == RED
 2       if z.p == z.p.p.left
 3           y = z.p.p.right
 4           if y.color == RED
 5               z.p.color = BLACK                    // case 1
 6               y.color = BLACK                      // case 1
 7               z.p.p.color = RED                    // case 1
 8               z = z.p.p                            // case 1
 9           else if z == z.p.right
10                   z = z.p                          // case 2
11                   LEFT-ROTATE(T, z)                // case 2
12               z.p.color = BLACK                    // case 3
13               z.p.p.color = RED                    // case 3
14               RIGHT-ROTATE(T, z.p.p)               // case 3
15       else (same as then clause
                 with "right" and "left" exchanged)
16   T.root.color = BLACK
```

To understand how RB-INSERT-FIXUP works, we shall break our examination of the code into three major steps. First, we shall determine what violations of the red-black properties are introduced in RB-INSERT when node z is inserted and colored red. Second, we shall examine the overall goal of the **while** loop in lines 1–15. Finally, we shall explore each of the three cases[1] within the **while** loop's body and see how they accomplish the goal. Figure 13.4 shows how RB-INSERT-FIXUP operates on a sample red-black tree.

Which of the red-black properties might be violated upon the call to RB-INSERT-FIXUP? Property 1 certainly continues to hold, as does property 3, since both children of the newly inserted red node are the sentinel $T.nil$. Property 5, which says that the number of black nodes is the same on every simple path from a given node, is satisfied as well, because node z replaces the (black) sentinel, and node z is red with sentinel children. Thus, the only properties that might be violated are property 2, which requires the root to be black, and property 4, which says that a red node cannot have a red child. Both possible violations are due to z being colored red. Property 2 is violated if z is the root, and property 4 is violated if z's parent is red. Figure 13.4(a) shows a violation of property 4 after the node z has been inserted.

---

[1]Case 2 falls through into case 3, and so these two cases are not mutually exclusive.
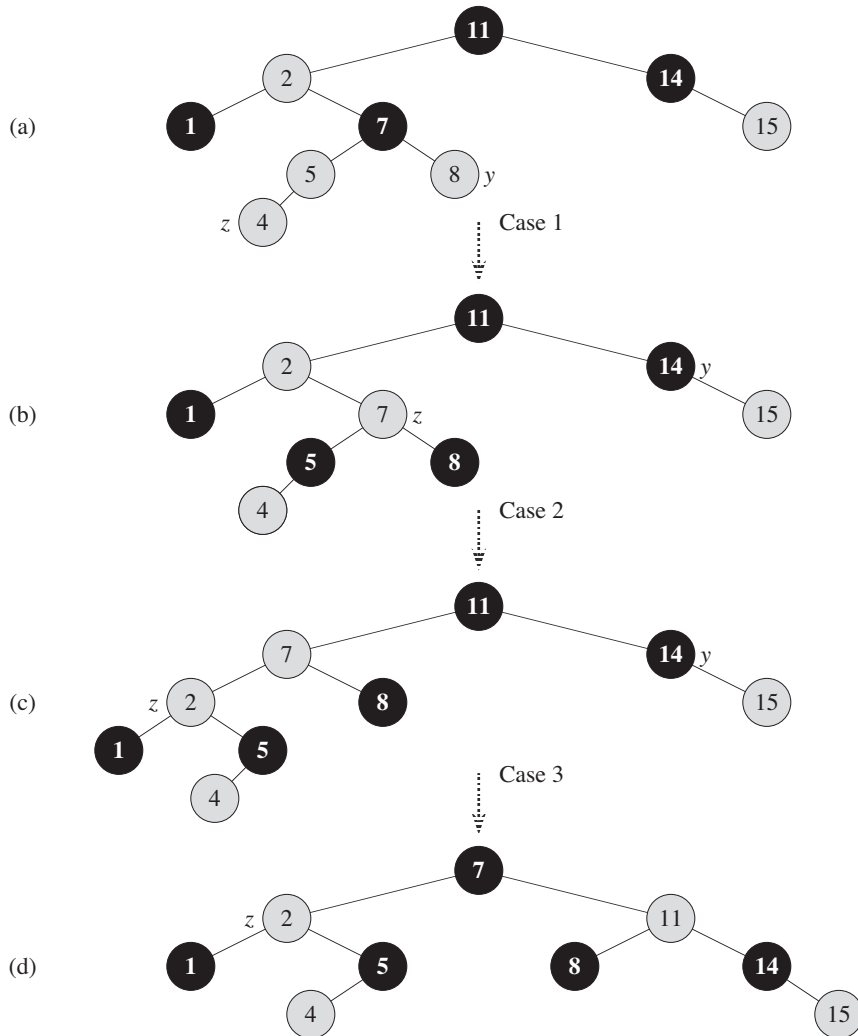
Figure 13.4 The operation of RB-INSERT-FIXUP. **(a)** A node z after insertion. Because both z and its parent $z.p$ are red, a violation of property 4 occurs. Since z's uncle y is red, case 1 in the code applies. We recolor nodes and move the pointer z up the tree, resulting in the tree shown in **(b)**. Once again, z and its parent are both red, but z's uncle y is black. Since z is the right child of $z.p$, case 2 applies. We perform a left rotation, and the tree that results is shown in **(c)**. Now, z is the left child of its parent, and case 3 applies. Recoloring and right rotation yield the tree in **(d)**, which is a legal red-black tree.

The **while** loop in lines 1–15 maintains the following three-part invariant at the start of each iteration of the loop:

a. Node z is red.

b. If $z.p$ is the root, then $z.p$ is black.

c. If the tree violates any of the red-black properties, then it violates at most one of them, and the violation is of either property 2 or property 4. If the tree violates property 2, it is because z is the root and is red. If the tree violates property 4, it is because both z and $z.p$ are red.

Part (c), which deals with violations of red-black properties, is more central to showing that RB-INSERT-FIXUP restores the red-black properties than parts (a) and (b), which we use along the way to understand situations in the code. Because we'll be focusing on node z and nodes near it in the tree, it helps to know from part (a) that z is red. We shall use part (b) to show that the node $z.p.p$ exists when we reference it in lines 2, 3, 7, 8, 13, and 14.

Recall that we need to show that a loop invariant is true prior to the first iteration of the loop, that each iteration maintains the loop invariant, and that the loop invariant gives us a useful property at loop termination.

We start with the initialization and termination arguments. Then, as we examine how the body of the loop works in more detail, we shall argue that the loop maintains the invariant upon each iteration. Along the way, we shall also demonstrate that each iteration of the loop has two possible outcomes: either the pointer z moves up the tree, or we perform some rotations and then the loop terminates.

Initialization: Prior to the first iteration of the loop, we started with a red-black tree with no violations, and we added a red node z. We show that each part of the invariant holds at the time RB-INSERT-FIXUP is called:

a. When RB-INSERT-FIXUP is called, z is the red node that was added.

b. If $z.p$ is the root, then $z.p$ started out black and did not change prior to the call of RB-INSERT-FIXUP.

c. We have already seen that properties 1, 3, and 5 hold when RB-INSERT-FIXUP is called.

If the tree violates property 2, then the red root must be the newly added node z, which is the only internal node in the tree. Because the parent and both children of z are the sentinel, which is black, the tree does not also violate property 4. Thus, this violation of property 2 is the only violation of red-black properties in the entire tree.

If the tree violates property 4, then, because the children of node z are black sentinels and the tree had no other violations prior to z being added, the

violation must be because both z and $z.p$ are red. Moreover, the tree violates no other red-black properties.

Termination: When the loop terminates, it does so because $z.p$ is black. (If z is the root, then $z.p$ is the sentinel $T.nil$, which is black.) Thus, the tree does not violate property 4 at loop termination. By the loop invariant, the only property that might fail to hold is property 2. Line 16 restores this property, too, so that when RB-INSERT-FIXUP terminates, all the red-black properties hold.

Maintenance: We actually need to consider six cases in the **while** loop, but three of them are symmetric to the other three, depending on whether line 2 determines z's parent $z.p$ to be a left child or a right child of z's grandparent $z.p.p$. We have given the code only for the situation in which $z.p$ is a left child. The node $z.p.p$ exists, since by part (b) of the loop invariant, if $z.p$ is the root, then $z.p$ is black. Since we enter a loop iteration only if $z.p$ is red, we know that $z.p$ cannot be the root. Hence, $z.p.p$ exists.

We distinguish case 1 from cases 2 and 3 by the color of z's parent's sibling, or "uncle." Line 3 makes y point to z's uncle $z.p.p.right$, and line 4 tests y's color. If y is red, then we execute case 1. Otherwise, control passes to cases 2 and 3. In all three cases, z's grandparent $z.p.p$ is black, since its parent $z.p$ is red, and property 4 is violated only between z and $z.p$.

Case 1: z's uncle y is red

Figure 13.5 shows the situation for case 1 (lines 5–8), which occurs when both $z.p$ and y are red. Because $z.p.p$ is black, we can color both $z.p$ and y black, thereby fixing the problem of z and $z.p$ both being red, and we can color $z.p.p$ red, thereby maintaining property 5. We then repeat the **while** loop with $z.p.p$ as the new node z. The pointer z moves up two levels in the tree.

Now, we show that case 1 maintains the loop invariant at the start of the next iteration. We use z to denote node z in the current iteration, and $z' = z.p.p$ to denote the node that will be called node z at the test in line 1 upon the next iteration.

a. Because this iteration colors $z.p.p$ red, node z' is red at the start of the next iteration.

b. The node $z'.p$ is $z.p.p.p$ in this iteration, and the color of this node does not change. If this node is the root, it was black prior to this iteration, and it remains black at the start of the next iteration.

c. We have already argued that case 1 maintains property 5, and it does not introduce a violation of properties 1 or 3.
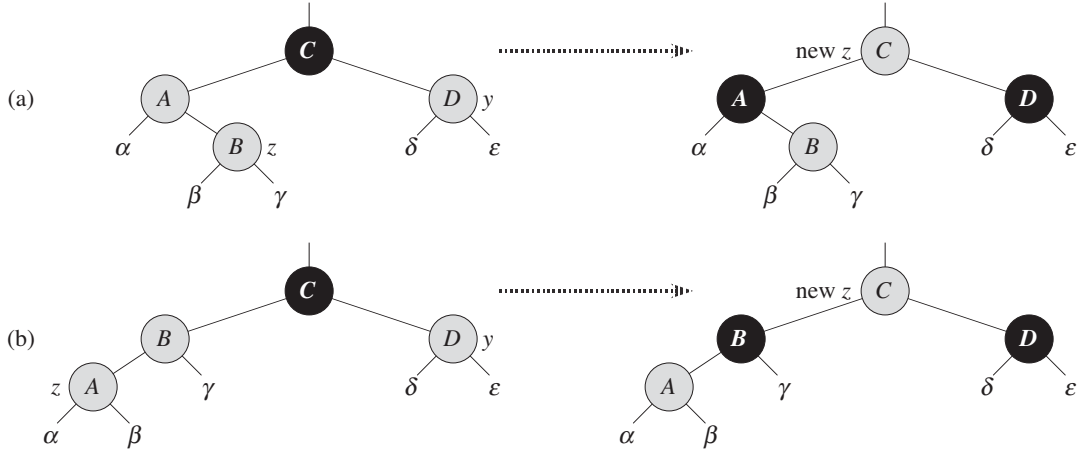
Figure 13.5 Case 1 of the procedure RB-INSERT-FIXUP. Property 4 is violated, since z and its parent $z.p$ are both red. We take the same action whether **(a)** z is a right child or **(b)** z is a left child. Each of the subtrees α, β, γ, δ, and ε has a black root, and each has the same black-height. The code for case 1 changes the colors of some nodes, preserving property 5: all downward simple paths from a node to a leaf have the same number of blacks. The **while** loop continues with node z's grandparent $z.p.p$ as the new z. Any violation of property 4 can now occur only between the new z, which is red, and its parent, if it is red as well.

If node z' is the root at the start of the next iteration, then case 1 corrected the lone violation of property 4 in this iteration. Since z' is red and it is the root, property 2 becomes the only one that is violated, and this violation is due to z'.

If node z' is not the root at the start of the next iteration, then case 1 has not created a violation of property 2. Case 1 corrected the lone violation of property 4 that existed at the start of this iteration. It then made z' red and left $z'.p$ alone. If $z'.p$ was black, there is no violation of property 4. If $z'.p$ was red, coloring z' red created one violation of property 4 between z' and $z'.p$.

Case 2: z's uncle y is black and z is a right child
Case 3: z's uncle y is black and z is a left child

In cases 2 and 3, the color of z's uncle y is black. We distinguish the two cases according to whether z is a right or left child of $z.p$. Lines 10–11 constitute case 2, which is shown in Figure 13.6 together with case 3. In case 2, node z is a right child of its parent. We immediately use a left rotation to transform the situation into case 3 (lines 12–14), in which node z is a left child. Because
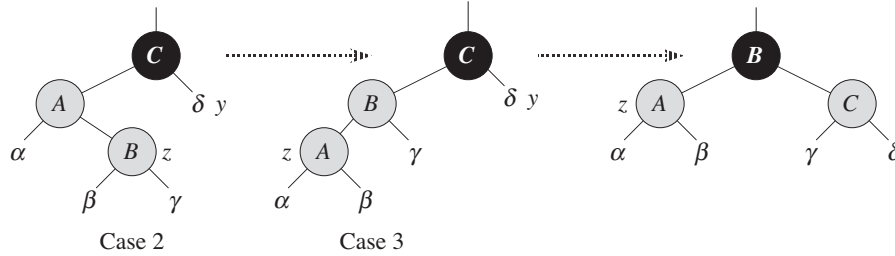
Figure 13.6   Cases 2 and 3 of the procedure RB-INSERT-FIXUP. As in case 1, property 4 is violated in either case 2 or case 3 because z and its parent $z.p$ are both red. Each of the subtrees α, β, γ, and δ has a black root (α, β, and γ from property 4, and δ because otherwise we would be in case 1), and each has the same black-height. We transform case 2 into case 3 by a left rotation, which preserves property 5: all downward simple paths from a node to a leaf have the same number of blacks. Case 3 causes some color changes and a right rotation, which also preserve property 5. The **while** loop then terminates, because property 4 is satisfied: there are no longer two red nodes in a row.

both z and $z.p$ are red, the rotation affects neither the black-height of nodes nor property 5. Whether we enter case 3 directly or through case 2, z's uncle y is black, since otherwise we would have executed case 1. Additionally, the node $z.p.p$ exists, since we have argued that this node existed at the time that lines 2 and 3 were executed, and after moving z up one level in line 10 and then down one level in line 11, the identity of $z.p.p$ remains unchanged. In case 3, we execute some color changes and a right rotation, which preserve property 5, and then, since we no longer have two red nodes in a row, we are done. The **while** loop does not iterate another time, since $z.p$ is now black.

We now show that cases 2 and 3 maintain the loop invariant. (As we have just argued, $z.p$ will be black upon the next test in line 1, and the loop body will not execute again.)

a. Case 2 makes z point to $z.p$, which is red. No further change to z or its color occurs in cases 2 and 3.

b. Case 3 makes $z.p$ black, so that if $z.p$ is the root at the start of the next iteration, it is black.

c. As in case 1, properties 1, 3, and 5 are maintained in cases 2 and 3.

   Since node z is not the root in cases 2 and 3, we know that there is no violation of property 2. Cases 2 and 3 do not introduce a violation of property 2, since the only node that is made red becomes a child of a black node by the rotation in case 3.

   Cases 2 and 3 correct the lone violation of property 4, and they do not introduce another violation.

Having shown that each iteration of the loop maintains the invariant, we have shown that RB-INSERT-FIXUP correctly restores the red-black properties.

Analysis

What is the running time of RB-INSERT? Since the height of a red-black tree on n nodes is $O(\lg n)$, lines 1–16 of RB-INSERT take $O(\lg n)$ time. In RB-INSERT-FIXUP, the **while** loop repeats only if case 1 occurs, and then the pointer z moves two levels up the tree. The total number of times the **while** loop can be executed is therefore $O(\lg n)$. Thus, RB-INSERT takes a total of $O(\lg n)$ time. Moreover, it never performs more than two rotations, since the **while** loop terminates if case 2 or case 3 is executed.

Exercises

13.3-1
In line 16 of RB-INSERT, we set the color of the newly inserted node z to red. Observe that if we had chosen to set z's color to black, then property 4 of a red-black tree would not be violated. Why didn't we choose to set z's color to black?

13.3-2
Show the red-black trees that result after successively inserting the keys $41, 38, 31, 12, 19, 8$ into an initially empty red-black tree.

13.3-3
Suppose that the black-height of each of the subtrees $\alpha, \beta, \gamma, \delta, \varepsilon$ in Figures 13.5 and 13.6 is k. Label each node in each figure with its black-height to verify that the indicated transformation preserves property 5.

13.3-4
Professor Teach is concerned that RB-INSERT-FIXUP might set $T.nil.color$ to RED, in which case the test in line 1 would not cause the loop to terminate when z is the root. Show that the professor's concern is unfounded by arguing that RB-INSERT-FIXUP never sets $T.nil.color$ to RED.

13.3-5
Consider a red-black tree formed by inserting n nodes with RB-INSERT. Argue that if $n > 1$, the tree has at least one red node.

13.3-6
Suggest how to implement RB-INSERT efficiently if the representation for red-black trees includes no storage for parent pointers.

13.4 Deletion

Like the other basic operations on an n-node red-black tree, deletion of a node takes time $O(\lg n)$. Deleting a node from a red-black tree is a bit more complicated than inserting a node.

The procedure for deleting a node from a red-black tree is based on the TREE-DELETE procedure (Section 12.3). First, we need to customize the TRANSPLANT subroutine that TREE-DELETE calls so that it applies to a red-black tree:

RB-TRANSPLANT(T, u, v)

```
1   if u.p == T.nil
2       T.root = v
3   elseif u == u.p.left
4       u.p.left = v
5   else u.p.right = v
6   v.p = u.p
```

The procedure RB-TRANSPLANT differs from TRANSPLANT in two ways. First, line 1 references the sentinel $T.nil$ instead of NIL. Second, the assignment to $v.p$ in line 6 occurs unconditionally: we can assign to $v.p$ even if v points to the sentinel. In fact, we shall exploit the ability to assign to $v.p$ when $v = T.nil$.

The procedure RB-DELETE is like the TREE-DELETE procedure, but with additional lines of pseudocode. Some of the additional lines keep track of a node y that might cause violations of the red-black properties. When we want to delete node z and z has fewer than two children, then z is removed from the tree, and we want y to be z. When z has two children, then y should be z's successor, and y moves into z's position in the tree. We also remember y's color before it is removed from or moved within the tree, and we keep track of the node x that moves into y's original position in the tree, because node x might also cause violations of the red-black properties. After deleting node z, RB-DELETE calls an auxiliary procedure RB-DELETE-FIXUP, which changes colors and performs rotations to restore the red-black properties.

RB-DELETE(T, z)

```
 1  y = z
 2  y-original-color = y.color
 3  if z.left == T.nil
 4      x = z.right
 5      RB-TRANSPLANT(T, z, z.right)
 6  elseif z.right == T.nil
 7      x = z.left
 8      RB-TRANSPLANT(T, z, z.left)
 9  else y = TREE-MINIMUM(z.right)
10      y-original-color = y.color
11      x = y.right
12      if y.p == z
13          x.p = y
14      else RB-TRANSPLANT(T, y, y.right)
15          y.right = z.right
16          y.right.p = y
17      RB-TRANSPLANT(T, z, y)
18      y.left = z.left
19      y.left.p = y
20      y.color = z.color
21  if y-original-color == BLACK
22      RB-DELETE-FIXUP(T, x)
```

Although RB-DELETE contains almost twice as many lines of pseudocode as TREE-DELETE, the two procedures have the same basic structure. You can find each line of TREE-DELETE within RB-DELETE (with the changes of replacing NIL by $T.nil$ and replacing calls to TRANSPLANT by calls to RB-TRANSPLANT), executed under the same conditions.

Here are the other differences between the two procedures:

- We maintain node y as the node either removed from the tree or moved within the tree. Line 1 sets y to point to node z when z has fewer than two children and is therefore removed. When z has two children, line 9 sets y to point to z's successor, just as in TREE-DELETE, and y will move into z's position in the tree.

- Because node y's color might change, the variable *y-original-color* stores y's color before any changes occur. Lines 2 and 10 set this variable immediately after assignments to y. When z has two children, then $y \neq z$ and node y moves into node z's original position in the red-black tree; line 20 gives y the same color as z. We need to save y's original color in order to test it at the

end of RB-DELETE; if it was black, then removing or moving y could cause violations of the red-black properties.

- As discussed, we keep track of the node x that moves into node y's original position. The assignments in lines 4, 7, and 11 set x to point to either y's only child or, if y has no children, the sentinel $T.nil$. (Recall from Section 12.3 that y has no left child.)

- Since node x moves into node y's original position, the attribute $x.p$ is always set to point to the original position in the tree of y's parent, even if x is, in fact, the sentinel $T.nil$. Unless z is y's original parent (which occurs only when z has two children and its successor y is z's right child), the assignment to $x.p$ takes place in line 6 of RB-TRANSPLANT. (Observe that when RB-TRANSPLANT is called in lines 5, 8, or 14, the second parameter passed is the same as x.)

  When y's original parent is z, however, we do not want $x.p$ to point to y's original parent, since we are removing that node from the tree. Because node y will move up to take z's position in the tree, setting $x.p$ to y in line 13 causes $x.p$ to point to the original position of y's parent, even if $x = T.nil$.

- Finally, if node y was black, we might have introduced one or more violations of the red-black properties, and so we call RB-DELETE-FIXUP in line 22 to restore the red-black properties. If y was red, the red-black properties still hold when y is removed or moved, for the following reasons:

  1. No black-heights in the tree have changed.
  2. No red nodes have been made adjacent. Because y takes z's place in the tree, along with z's color, we cannot have two adjacent red nodes at y's new position in the tree. In addition, if y was not z's right child, then y's original right child x replaces y in the tree. If y is red, then x must be black, and so replacing y by x cannot cause two red nodes to become adjacent.
  3. Since y could not have been the root if it was red, the root remains black.

If node y was black, three problems may arise, which the call of RB-DELETE-FIXUP will remedy. First, if y had been the root and a red child of y becomes the new root, we have violated property 2. Second, if both x and $x.p$ are red, then we have violated property 4. Third, moving y within the tree causes any simple path that previously contained y to have one fewer black node. Thus, property 5 is now violated by any ancestor of y in the tree. We can correct the violation of property 5 by saying that node x, now occupying y's original position, has an "extra" black. That is, if we add 1 to the count of black nodes on any simple path that contains x, then under this interpretation, property 5 holds. When we remove or move the black node y, we "push" its blackness onto node x. The problem is that now node x is neither red nor black, thereby violating property 1. Instead,

node x is either "doubly black" or "red-and-black," and it contributes either 2 or 1, respectively, to the count of black nodes on simple paths containing x. The *color* attribute of x will still be either RED (if x is red-and-black) or BLACK (if x is doubly black). In other words, the extra black on a node is reflected in x's pointing to the node rather than in the *color* attribute.

   We can now see the procedure RB-DELETE-FIXUP and examine how it restores the red-black properties to the search tree.

RB-DELETE-FIXUP(T, x)

```
 1  while x ≠ T.root and x.color == BLACK
 2      if x == x.p.left
 3          w = x.p.right
 4          if w.color == RED
 5              w.color = BLACK                              // case 1
 6              x.p.color = RED                             // case 1
 7              LEFT-ROTATE(T, x.p)                         // case 1
 8              w = x.p.right                               // case 1
 9          if w.left.color == BLACK and w.right.color == BLACK
10              w.color = RED                               // case 2
11              x = x.p                                     // case 2
12          else if w.right.color == BLACK
13                  w.left.color = BLACK                    // case 3
14                  w.color = RED                           // case 3
15                  RIGHT-ROTATE(T, w)                      // case 3
16                  w = x.p.right                           // case 3
17              w.color = x.p.color                         // case 4
18              x.p.color = BLACK                           // case 4
19              w.right.color = BLACK                       // case 4
20              LEFT-ROTATE(T, x.p)                         // case 4
21              x = T.root                                  // case 4
22      else (same as then clause with "right" and "left" exchanged)
23  x.color = BLACK
```

The procedure RB-DELETE-FIXUP restores properties 1, 2, and 4. Exercises 13.4-1 and 13.4-2 ask you to show that the procedure restores properties 2 and 4, and so in the remainder of this section, we shall focus on property 1. The goal of the **while** loop in lines 1–22 is to move the extra black up the tree until

1. x points to a red-and-black node, in which case we color x (singly) black in line 23;

2. x points to the root, in which case we simply "remove" the extra black; or

3. having performed suitable rotations and recolorings, we exit the loop.

Within the **while** loop, x always points to a nonroot doubly black node. We determine in line 2 whether x is a left child or a right child of its parent $x.p$. (We have given the code for the situation in which x is a left child; the situation in which x is a right child—line 22—is symmetric.) We maintain a pointer w to the sibling of x. Since node x is doubly black, node w cannot be $T.nil$, because otherwise, the number of blacks on the simple path from $x.p$ to the (singly black) leaf w would be smaller than the number on the simple path from $x.p$ to x.

The four cases[2] in the code appear in Figure 13.7. Before examining each case in detail, let's look more generally at how we can verify that the transformation in each of the cases preserves property 5. The key idea is that in each case, the transformation applied preserves the number of black nodes (including x's extra black) from (and including) the root of the subtree shown to each of the subtrees $\alpha, \beta, \ldots, \zeta$. Thus, if property 5 holds prior to the transformation, it continues to hold afterward. For example, in Figure 13.7(a), which illustrates case 1, the number of black nodes from the root to either subtree α or β is 3, both before and after the transformation. (Again, remember that node x adds an extra black.) Similarly, the number of black nodes from the root to any of γ, δ, ε, and ζ is 2, both before and after the transformation. In Figure 13.7(b), the counting must involve the value c of the *color* attribute of the root of the subtree shown, which can be either RED or BLACK. If we define count(RED) = 0 and count(BLACK) = 1, then the number of black nodes from the root to α is $2 + \text{count}(c)$, both before and after the transformation. In this case, after the transformation, the new node x has *color* attribute c, but this node is really either red-and-black (if $c =$ RED) or doubly black (if $c =$ BLACK). You can verify the other cases similarly (see Exercise 13.4-5).

Case 1: x's sibling w is red

Case 1 (lines 5–8 of RB-DELETE-FIXUP and Figure 13.7(a)) occurs when node w, the sibling of node x, is red. Since w must have black children, we can switch the colors of w and $x.p$ and then perform a left-rotation on $x.p$ without violating any of the red-black properties. The new sibling of x, which is one of w's children prior to the rotation, is now black, and thus we have converted case 1 into case 2, 3, or 4.

Cases 2, 3, and 4 occur when node w is black; they are distinguished by the colors of w's children.

---

[2]As in RB-INSERT-FIXUP, the cases in RB-DELETE-FIXUP are not mutually exclusive.

Case 2: x's sibling w is black, and both of w's children are black

In case 2 (lines 10–11 of RB-DELETE-FIXUP and Figure 13.7(b)), both of w's children are black. Since w is also black, we take one black off both x and w, leaving x with only one black and leaving w red. To compensate for removing one black from x and w, we would like to add an extra black to $x.p$, which was originally either red or black. We do so by repeating the **while** loop with $x.p$ as the new node x. Observe that if we enter case 2 through case 1, the new node x is red-and-black, since the original $x.p$ was red. Hence, the value c of the *color* attribute of the new node x is RED, and the loop terminates when it tests the loop condition. We then color the new node x (singly) black in line 23.

Case 3: x's sibling w is black, w's left child is red, and w's right child is black

Case 3 (lines 13–16 and Figure 13.7(c)) occurs when w is black, its left child is red, and its right child is black. We can switch the colors of w and its left child $w.left$ and then perform a right rotation on w without violating any of the red-black properties. The new sibling w of x is now a black node with a red right child, and thus we have transformed case 3 into case 4.

Case 4: x's sibling w is black, and w's right child is red

Case 4 (lines 17–21 and Figure 13.7(d)) occurs when node x's sibling w is black and w's right child is red. By making some color changes and performing a left rotation on $x.p$, we can remove the extra black on x, making it singly black, without violating any of the red-black properties. Setting x to be the root causes the **while** loop to terminate when it tests the loop condition.

Analysis

What is the running time of RB-DELETE? Since the height of a red-black tree of n nodes is $O(\lg n)$, the total cost of the procedure without the call to RB-DELETE-FIXUP takes $O(\lg n)$ time. Within RB-DELETE-FIXUP, each of cases 1, 3, and 4 lead to termination after performing a constant number of color changes and at most three rotations. Case 2 is the only case in which the **while** loop can be repeated, and then the pointer x moves up the tree at most $O(\lg n)$ times, performing no rotations. Thus, the procedure RB-DELETE-FIXUP takes $O(\lg n)$ time and performs at most three rotations, and the overall time for RB-DELETE is therefore also $O(\lg n)$.
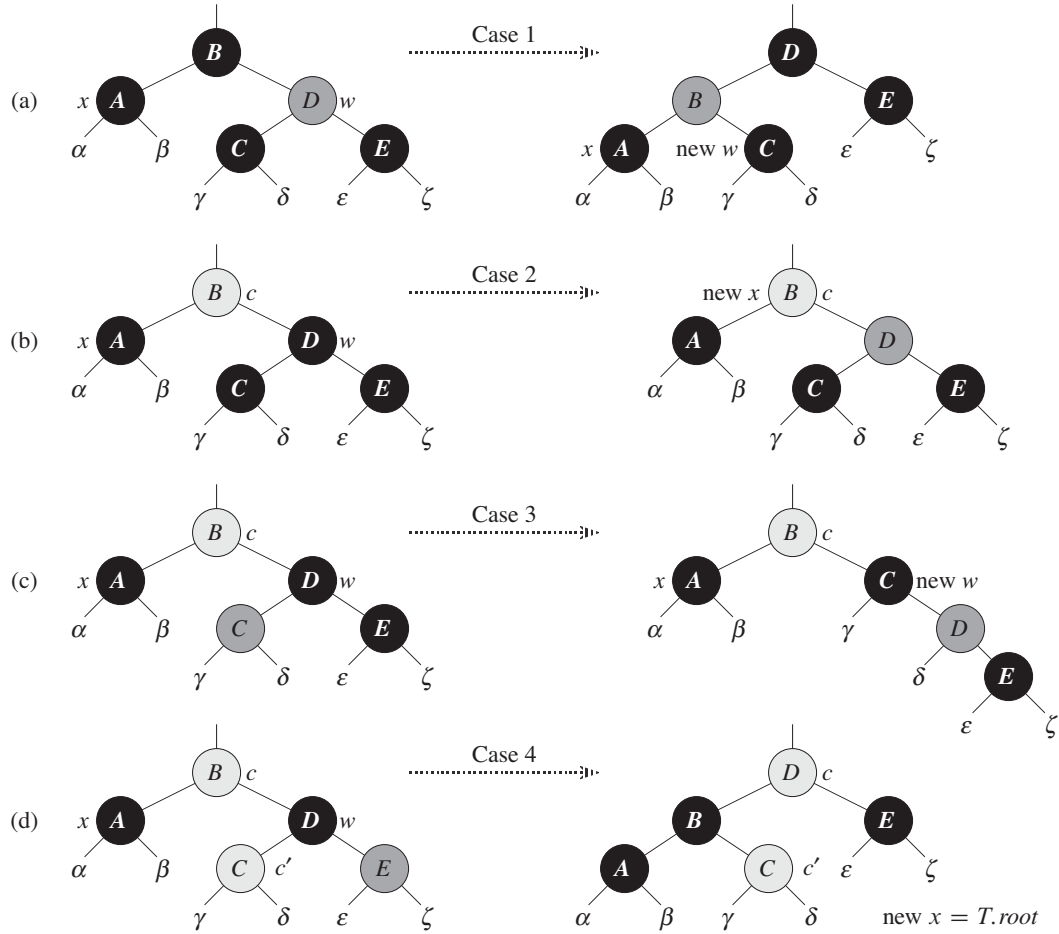
Figure 13.7 The cases in the **while** loop of the procedure RB-DELETE-FIXUP. Darkened nodes have *color* attributes BLACK, heavily shaded nodes have *color* attributes RED, and lightly shaded nodes have *color* attributes represented by c and c', which may be either RED or BLACK. The letters $\alpha, \beta, \ldots, \zeta$ represent arbitrary subtrees. Each case transforms the configuration on the left into the configuration on the right by changing some colors and/or performing a rotation. Any node pointed to by x has an extra black and is either doubly black or red-and-black. Only case 2 causes the loop to repeat. **(a)** Case 1 is transformed to case 2, 3, or 4 by exchanging the colors of nodes B and D and performing a left rotation. **(b)** In case 2, the extra black represented by the pointer x moves up the tree by coloring node D red and setting x to point to node B. If we enter case 2 through case 1, the **while** loop terminates because the new node x is red-and-black, and therefore the value c of its *color* attribute is RED. **(c)** Case 3 is transformed to case 4 by exchanging the colors of nodes C and D and performing a right rotation. **(d)** Case 4 removes the extra black represented by x by changing some colors and performing a left rotation (without violating the red-black properties), and then the loop terminates.

Exercises

13.4-1
Argue that after executing RB-DELETE-FIXUP, the root of the tree must be black.

13.4-2
Argue that if in RB-DELETE both x and $x.p$ are red, then property 4 is restored by the call to RB-DELETE-FIXUP(T, x).

13.4-3
In Exercise 13.3-2, you found the red-black tree that results from successively inserting the keys $41, 38, 31, 12, 19, 8$ into an initially empty tree. Now show the red-black trees that result from the successive deletion of the keys in the order $8, 12, 19, 31, 38, 41$.

13.4-4
In which lines of the code for RB-DELETE-FIXUP might we examine or modify the sentinel $T.nil$?

13.4-5
In each of the cases of Figure 13.7, give the count of black nodes from the root of the subtree shown to each of the subtrees $\alpha, \beta, \dots, \zeta$, and verify that each count remains the same after the transformation. When a node has a *color* attribute c or c', use the notation count(c) or count(c') symbolically in your count.

13.4-6
Professors Skelton and Baron are concerned that at the start of case 1 of RB-DELETE-FIXUP, the node $x.p$ might not be black. If the professors are correct, then lines 5–6 are wrong. Show that $x.p$ must be black at the start of case 1, so that the professors have nothing to worry about.

13.4-7
Suppose that a node x is inserted into a red-black tree with RB-INSERT and then is immediately deleted with RB-DELETE. Is the resulting red-black tree the same as the initial red-black tree? Justify your answer.

Problems

13-1 *Persistent dynamic sets*

During the course of an algorithm, we sometimes find that we need to maintain past versions of a dynamic set as it is updated. We call such a set **persistent**. One way to implement a persistent set is to copy the entire set whenever it is modified, but this approach can slow down a program and also consume much space. Sometimes, we can do much better.

Consider a persistent set S with the operations INSERT, DELETE, and SEARCH, which we implement using binary search trees as shown in Figure 13.8(a). We maintain a separate root for every version of the set. In order to insert the key 5 into the set, we create a new node with key 5. This node becomes the left child of a new node with key 7, since we cannot modify the existing node with key 7. Similarly, the new node with key 7 becomes the left child of a new node with key 8 whose right child is the existing node with key 10. The new node with key 8 becomes, in turn, the right child of a new root r' with key 4 whose left child is the existing node with key 3. We thus copy only part of the tree and share some of the nodes with the original tree, as shown in Figure 13.8(b).

Assume that each tree node has the attributes *key*, *left*, and *right* but no parent. (See also Exercise 13.3-6.)

(a)   (b)
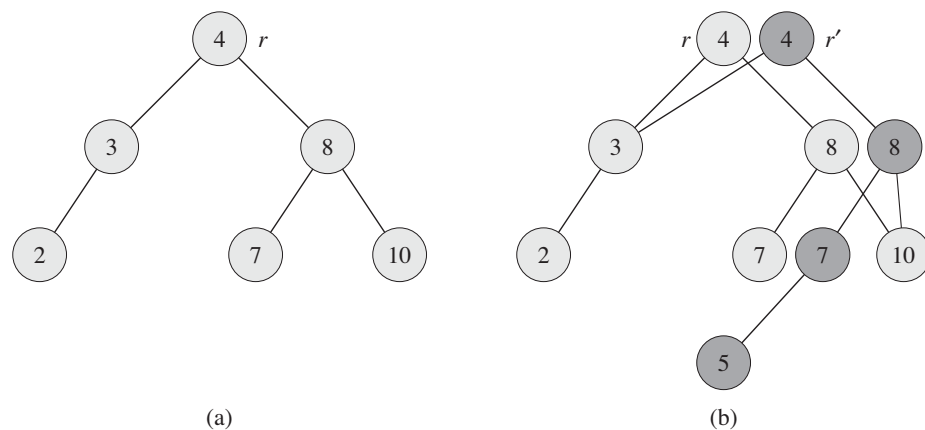
Figure 13.8 (a) A binary search tree with keys 2, 3, 4, 7, 8, 10. (b) The persistent binary search tree that results from the insertion of key 5. The most recent version of the set consists of the nodes reachable from the root r', and the previous version consists of the nodes reachable from r. Heavily shaded nodes are added when key 5 is inserted.

a. For a general persistent binary search tree, identify the nodes that we need to change to insert a key k or delete a node y.

b. Write a procedure PERSISTENT-TREE-INSERT that, given a persistent tree T and a key k to insert, returns a new persistent tree T' that is the result of inserting k into T.

c. If the height of the persistent binary search tree T is h, what are the time and space requirements of your implementation of PERSISTENT-TREE-INSERT? (The space requirement is proportional to the number of new nodes allocated.)

d. Suppose that we had included the parent attribute in each node. In this case, PERSISTENT-TREE-INSERT would need to perform additional copying. Prove that PERSISTENT-TREE-INSERT would then require $\Omega(n)$ time and space, where n is the number of nodes in the tree.

e. Show how to use red-black trees to guarantee that the worst-case running time and space are $O(\lg n)$ per insertion or deletion.

13-2   *Join operation on red-black trees*

The ***join*** operation takes two dynamic sets S_1 and S_2 and an element x such that for any $x_1 \in S_1$ and $x_2 \in S_2$, we have $x_1.key \leq x.key \leq x_2.key$. It returns a set $S = S_1 \cup \{x\} \cup S_2$. In this problem, we investigate how to implement the join operation on red-black trees.

a. Given a red-black tree T, let us store its black-height as the new attribute $T.bh$. Argue that RB-INSERT and RB-DELETE can maintain the bh attribute without requiring extra storage in the nodes of the tree and without increasing the asymptotic running times. Show that while descending through T, we can determine the black-height of each node we visit in $O(1)$ time per node visited.

We wish to implement the operation RB-JOIN(T_1, x, T_2), which destroys T_1 and T_2 and returns a red-black tree $T = T_1 \cup \{x\} \cup T_2$. Let n be the total number of nodes in T_1 and T_2.

b. Assume that $T_1.bh \geq T_2.bh$. Describe an $O(\lg n)$-time algorithm that finds a black node y in T_1 with the largest key from among those nodes whose black-height is $T_2.bh$.

c. Let T_y be the subtree rooted at y. Describe how $T_y \cup \{x\} \cup T_2$ can replace T_y in $O(1)$ time without destroying the binary-search-tree property.

d. What color should we make x so that red-black properties 1, 3, and 5 are maintained? Describe how to enforce properties 2 and 4 in $O(\lg n)$ time.

e. Argue that no generality is lost by making the assumption in part (b). Describe the symmetric situation that arises when $T_1.bh \leq T_2.bh$.

*f.* Argue that the running time of RB-JOIN is $O(\lg n)$.

13-3 AVL trees

An *AVL tree* is a binary search tree that is *height balanced*: for each node x, the heights of the left and right subtrees of x differ by at most 1. To implement an AVL tree, we maintain an extra attribute in each node: $x.h$ is the height of node x. As for any other binary search tree T, we assume that $T.root$ points to the root node.

a. Prove that an AVL tree with n nodes has height $O(\lg n)$. (*Hint:* Prove that an AVL tree of height h has at least F_h nodes, where F_h is the hth Fibonacci number.)

b. To insert into an AVL tree, we first place a node into the appropriate place in binary search tree order. Afterward, the tree might no longer be height balanced. Specifically, the heights of the left and right children of some node might differ by 2. Describe a procedure BALANCE(x), which takes a subtree rooted at x whose left and right children are height balanced and have heights that differ by at most 2, i.e., $|x.right.h - x.left.h| \leq 2$, and alters the subtree rooted at x to be height balanced. (*Hint:* Use rotations.)

c. Using part (b), describe a recursive procedure AVL-INSERT(x, z) that takes a node x within an AVL tree and a newly created node z (whose key has already been filled in), and adds z to the subtree rooted at x, maintaining the property that x is the root of an AVL tree. As in TREE-INSERT from Section 12.3, assume that $z.key$ has already been filled in and that $z.left = $ NIL and $z.right = $ NIL; also assume that $z.h = 0$. Thus, to insert the node z into the AVL tree T, we call AVL-INSERT($T.root, z$).

d. Show that AVL-INSERT, run on an n-node AVL tree, takes $O(\lg n)$ time and performs $O(1)$ rotations.

13-4 Treaps

If we insert a set of n items into a binary search tree, the resulting tree may be horribly unbalanced, leading to long search times. As we saw in Section 12.4, however, randomly built binary search trees tend to be balanced. Therefore, one strategy that, on average, builds a balanced tree for a fixed set of items would be to randomly permute the items and then insert them in that order into the tree.

What if we do not have all the items at once? If we receive the items one at a time, can we still randomly build a binary search tree out of them?
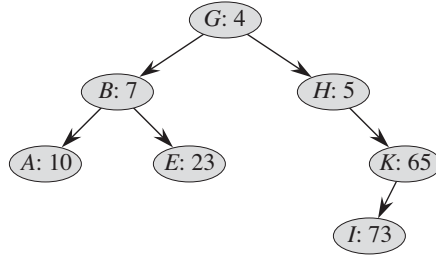
Figure 13.9   A treap. Each node x is labeled with $x.key$: $x.priority$. For example, the root has key G and priority 4.

We will examine a data structure that answers this question in the affirmative. A *treap* is a binary search tree with a modified way of ordering the nodes. Figure 13.9 shows an example. As usual, each node x in the tree has a key value $x.key$. In addition, we assign $x.priority$, which is a random number chosen independently for each node. We assume that all priorities are distinct and also that all keys are distinct. The nodes of the treap are ordered so that the keys obey the binary-search-tree property and the priorities obey the min-heap order property:

- If v is a left child of u, then $v.key < u.key$.

- If v is a right child of u, then $v.key > u.key$.

- If v is a child of u, then $v.priority > u.priority$.

(This combination of properties is why the tree is called a "treap": it has features of both a binary search tree and a heap.)

It helps to think of treaps in the following way. Suppose that we insert nodes $x_1, x_2, \ldots, x_n$, with associated keys, into a treap. Then the resulting treap is the tree that would have been formed if the nodes had been inserted into a normal binary search tree in the order given by their (randomly chosen) priorities, i.e., $x_i.priority < x_j.priority$ means that we had inserted x_i before x_j.

a. Show that given a set of nodes $x_1, x_2, \ldots, x_n$, with associated keys and priorities, all distinct, the treap associated with these nodes is unique.

b. Show that the expected height of a treap is $\Theta(\lg n)$, and hence the expected time to search for a value in the treap is $\Theta(\lg n)$.

Let us see how to insert a new node into an existing treap. The first thing we do is assign to the new node a random priority. Then we call the insertion algorithm, which we call TREAP-INSERT, whose operation is illustrated in Figure 13.10.
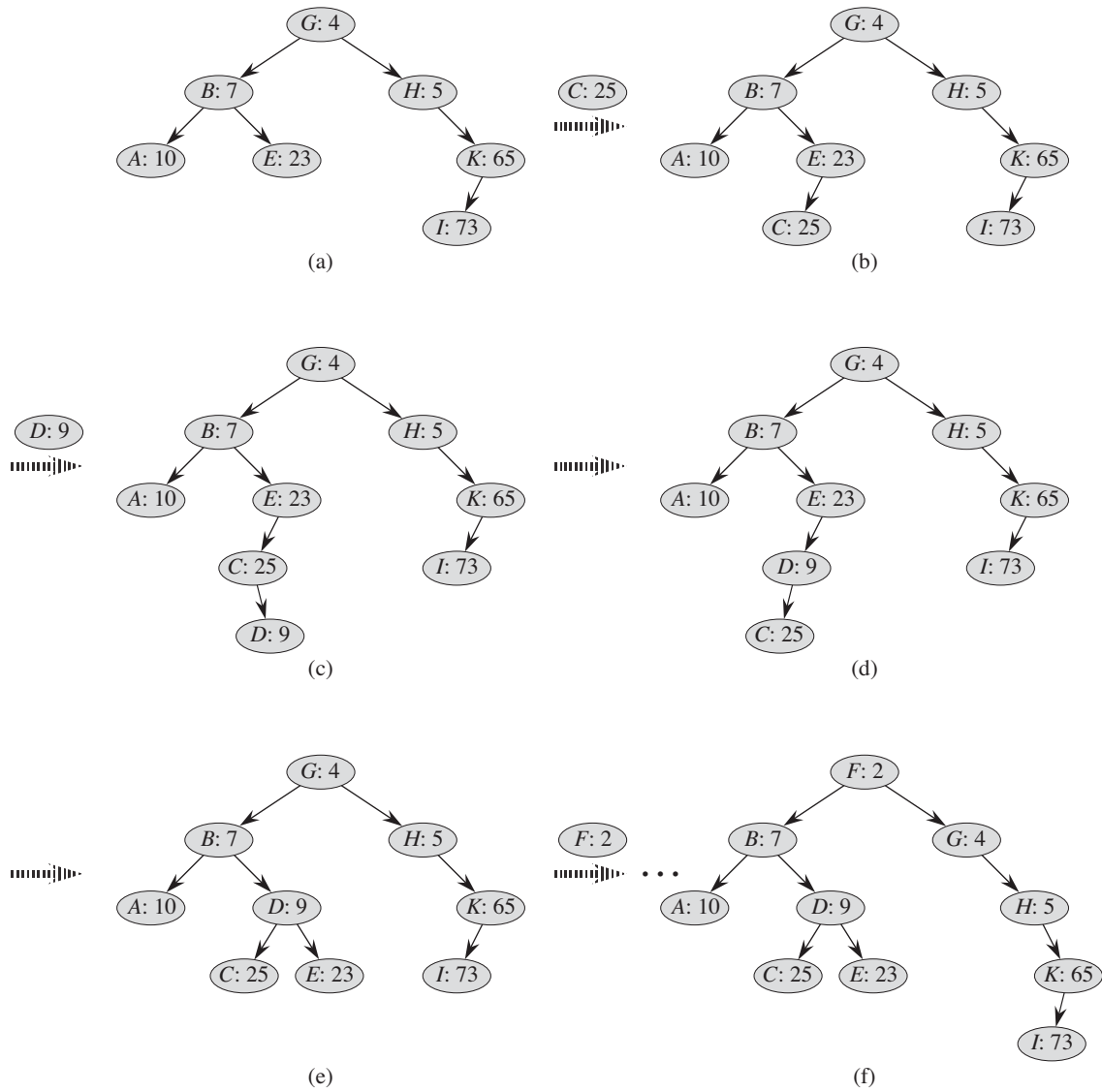
Figure 13.10 The operation of TREAP-INSERT. **(a)** The original treap, prior to insertion. **(b)** The treap after inserting a node with key *C* and priority 25. **(c)–(d)** Intermediate stages when inserting a node with key *D* and priority 9. **(e)** The treap after the insertion of parts (c) and (d) is done. **(f)** The treap after inserting a node with key *F* and priority 2.
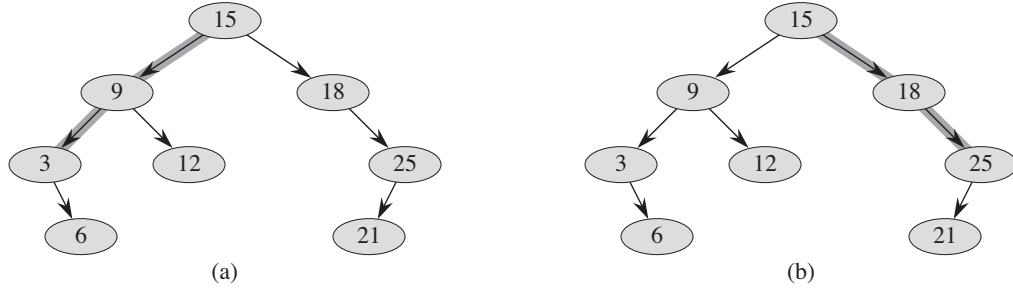
Figure 13.11   Spines of a binary search tree. The left spine is shaded in **(a)**, and the right spine is shaded in **(b)**.

c. Explain how TREAP-INSERT works. Explain the idea in English and give pseudocode. (*Hint:* Execute the usual binary-search-tree insertion procedure and then perform rotations to restore the min-heap order property.)

d. Show that the expected running time of TREAP-INSERT is $\Theta(\lg n)$.

TREAP-INSERT performs a search and then a sequence of rotations. Although these two operations have the same expected running time, they have different costs in practice. A search reads information from the treap without modifying it. In contrast, a rotation changes parent and child pointers within the treap. On most computers, read operations are much faster than write operations. Thus we would like TREAP-INSERT to perform few rotations. We will show that the expected number of rotations performed is bounded by a constant.

In order to do so, we will need some definitions, which Figure 13.11 depicts. The ***left spine*** of a binary search tree T is the simple path from the root to the node with the smallest key. In other words, the left spine is the simple path from the root that consists of only left edges. Symmetrically, the ***right spine*** of T is the simple path from the root consisting of only right edges. The ***length*** of a spine is the number of nodes it contains.

e. Consider the treap T immediately after TREAP-INSERT has inserted node x. Let C be the length of the right spine of the left subtree of x. Let D be the length of the left spine of the right subtree of x. Prove that the total number of rotations that were performed during the insertion of x is equal to $C + D$.

We will now calculate the expected values of C and D. Without loss of generality, we assume that the keys are $1, 2, \ldots, n$, since we are comparing them only to one another.

For nodes x and y in treap T, where $y \neq x$, let $k = x.key$ and $i = y.key$. We define indicator random variables

$X_{ik} = \mathrm{I}\{y$ is in the right spine of the left subtree of $x\}$.

f. Show that $X_{ik} = 1$ if and only if $y.priority > x.priority$, $y.key < x.key$, and, for every z such that $y.key < z.key < x.key$, we have $y.priority < z.priority$.

g. Show that

$$
\begin{aligned}
\Pr\{X_{ik} = 1\} &= \frac{(k - i - 1)!}{(k - i + 1)!} \\
&= \frac{1}{(k - i + 1)(k - i)} .
\end{aligned}
$$

h. Show that

$$
\begin{aligned}
\mathrm{E}[C] &= \sum_{j=1}^{k-1} \frac{1}{j(j + 1)} \\
&= 1 - \frac{1}{k} .
\end{aligned}
$$

i. Use a symmetry argument to show that

$$
\mathrm{E}[D] = 1 - \frac{1}{n - k + 1} .
$$

j. Conclude that the expected number of rotations performed when inserting a node into a treap is less than 2.

Chapter notes

The idea of balancing a search tree is due to Adel'son-Vel'skiĭ and Landis [2], who introduced a class of balanced search trees called "AVL trees" in 1962, described in Problem 13-3. Another class of search trees, called "2-3 trees," was introduced by J. E. Hopcroft (unpublished) in 1970. A 2-3 tree maintains balance by manipulating the degrees of nodes in the tree. Chapter 18 covers a generalization of 2-3 trees introduced by Bayer and McCreight [35], called "B-trees."

Red-black trees were invented by Bayer [34] under the name "symmetric binary B-trees." Guibas and Sedgewick [155] studied their properties at length and introduced the red/black color convention. Andersson [15] gives a simpler-to-code

variant of red-black trees. Weiss [351] calls this variant AA-trees. An AA-tree is similar to a red-black tree except that left children may never be red.

Treaps, the subject of Problem 13-4, were proposed by Seidel and Aragon [309]. They are the default implementation of a dictionary in LEDA [253], which is a well-implemented collection of data structures and algorithms.

There are many other variations on balanced binary trees, including weight-balanced trees [264], k-neighbor trees [245], and scapegoat trees [127]. Perhaps the most intriguing are the "splay trees" introduced by Sleator and Tarjan [320], which are "self-adjusting." (See Tarjan [330] for a good description of splay trees.) Splay trees maintain balance without any explicit balance condition such as color. Instead, "splay operations" (which involve rotations) are performed within the tree every time an access is made. The amortized cost (see Chapter 17) of each operation on an n-node tree is $O(\lg n)$.

Skip lists [286] provide an alternative to balanced binary trees. A skip list is a linked list that is augmented with a number of additional pointers. Each dictionary operation runs in expected time $O(\lg n)$ on a skip list of n items.