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***. Indeed, as we're about to see, the height of a red-black tree with n keys is at most $2\lg(n + 1)$, which is $O(\lg n)$.

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. Think of these NILs as pointers to leaves (external nodes) of the binary search tree and the normal, key-bearing nodes as internal nodes of the tree.

A red-black tree is a binary search 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 262). 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 $T.nil$.

Why use the sentinel? The sentinel makes it possible to treat a NIL child of a node x as an ordinary node whose parent is x. An alternative design would use a distinct sentinel node for each NIL in the tree, so that the parent of each NIL is well defined. That approach needlessly wastes space, however. Instead, just the one sentinel $T.nil$ represents 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. The red-black tree procedures can place whatever values in the sentinel that yield simpler code.

We generally confine our interest to the internal nodes of a red-black tree, since they hold the key values. The remainder of this chapter omits the leaves in drawings of 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. The black-height of a red-black tree is 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. Then node x has two children, either or both of which may be a leaf. If a child is black, then it contributes 1 to x's black-height but not to its own. If a child is red, then it contributes to neither x's black-height nor its own. Therefore, each child has a black-height of either $\mathrm{bh}(x) - 1$ (if it's black) or $\mathrm{bh}(x)$ (if it's red). Since the height of a child of x is less than the height of x itself, we can apply the inductive

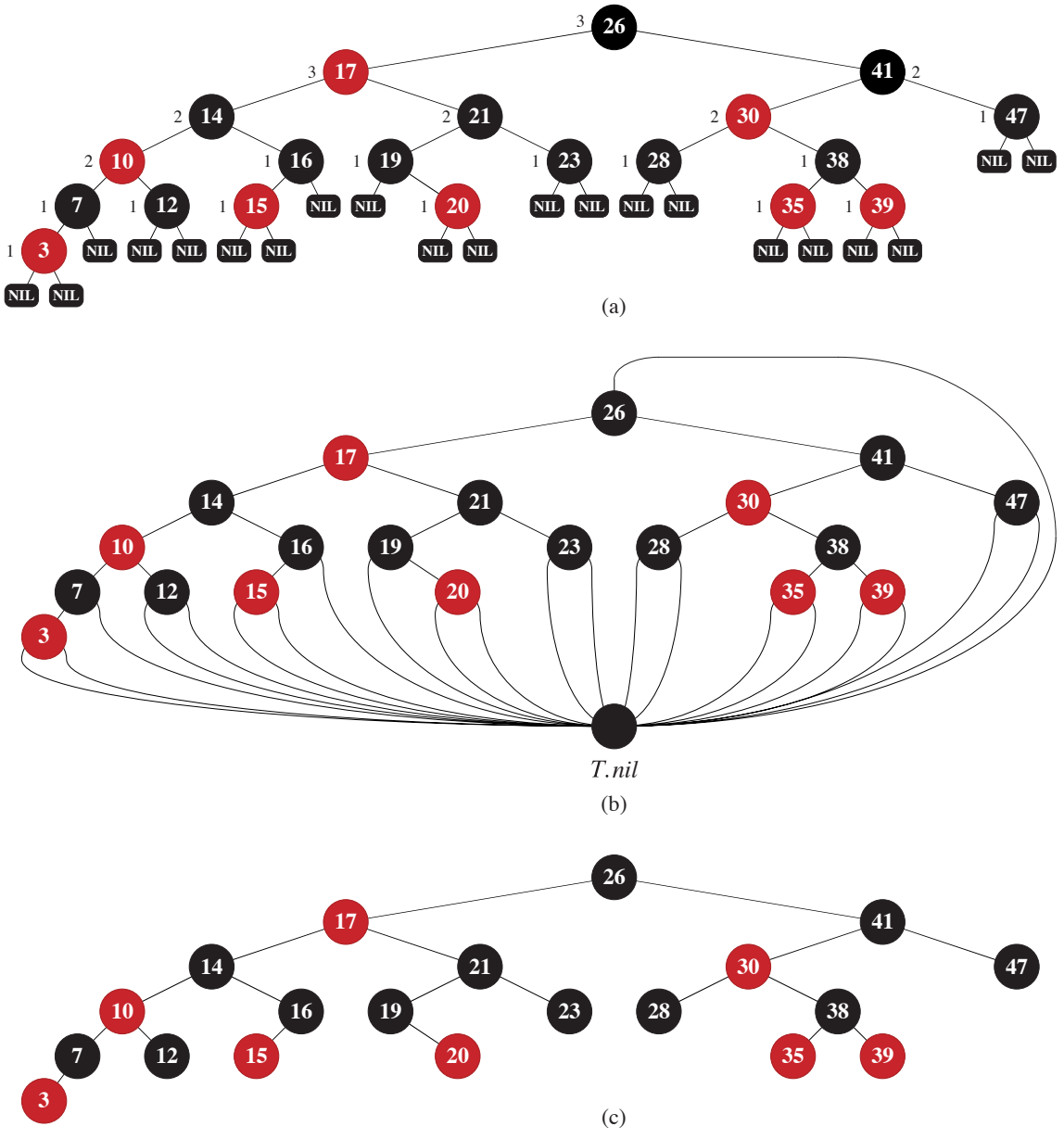

Figure 13.1 A red-black tree. Every node in a red-black tree is either red or black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. **(a)** Every leaf, shown as a NIL, is black. Each non-NIL node is marked with its black-height, where 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. The remainder of this chapter uses this drawing style.

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 including the root, must be black. Consequently, the black-height of the root must be at least $h/2$, and 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, each of the dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR runs in $O(\lg n)$ time on a red-black tree, 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 have to be replaced by $T.nil$.) Although the procedures TREE-INSERT and TREE-DELETE from Chapter 12 run in $O(\lg n)$ time when given a red-black tree as input, you cannot just use them to implement the dynamic-set operations INSERT and DELETE. They do not necessarily maintain the red-black properties, so you might not end up with a legal red-black tree. The remainder of this chapter shows how to insert into and delete from a red-black tree 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

Define a ***relaxed red-black tree*** as a binary search tree that satisfies red-black properties 1, 3, 4, and 5, but whose root may be either red or black. Consider a relaxed red-black tree T whose root is red. If the root of T is changed to black but no other changes occur, is the resulting tree a red-black tree?

13.1-4
Suppose that every black node in a red-black tree "absorbs" all of its red children, so that the children of any 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?

13.1-8
Argue that in a red-black tree, a red node cannot have exactly one non-NIL child.

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, colors and pointers within nodes need to change.

The pointer structure changes 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. Let's look at a left rotation on a node x, which transforms the structure on the right side of the figure to the structure on the left. Node x has a right child y, which must not be $T.nil$. The left rotation changes the subtree originally rooted at x by "twisting" the link between x and y to the left. The new root of the subtree is node y, with x as y's left child and y's original left child (the subtree represented by β in the figure) as x's right child.

The pseudocode for LEFT-ROTATE appearing on the following page assumes that $x.right \neq T.nil$ and that the root's parent is $T.nil$. Figure 13.3 shows an

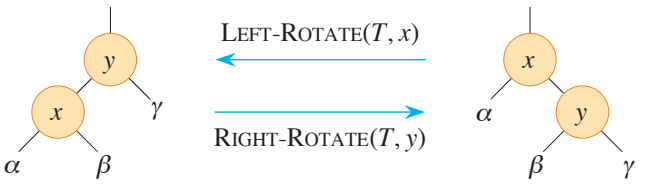

Figure 13.2 The rotation operations on a binary search tree. The operation LEFT-ROTATE(T, x) transforms the configuration of the two nodes on the right into the configuration on the left by changing a constant number of pointers. The inverse operation RIGHT-ROTATE(T, y) transforms the configuration on 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.

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, and all other attributes in a node remain the same.

```
LEFT-ROTATE(T, x)
 1  y = x.right
 2  x.right = y.left        // turn y's left subtree into x's right subtree
 3  if y.left ≠ T.nil       // if y's left subtree is not empty ...
 4      y.left.p = x        // ... then x becomes the parent of the subtree's root
 5  y.p = x.p               // x's parent becomes y's parent
 6  if x.p == T.nil         // if x was the root ...
 7      T.root = y          // ... then y becomes the root
 8  elseif x == x.p.left    // otherwise, if x was a left child ...
 9      x.p.left = y        // ... then y becomes a left child
10  else x.p.right = y      // otherwise, x was a right child, and now y is
11  y.left = x              // make x become y's left child
12  x.p = y
```

Exercises

13.2-1

Write pseudocode for RIGHT-ROTATE.

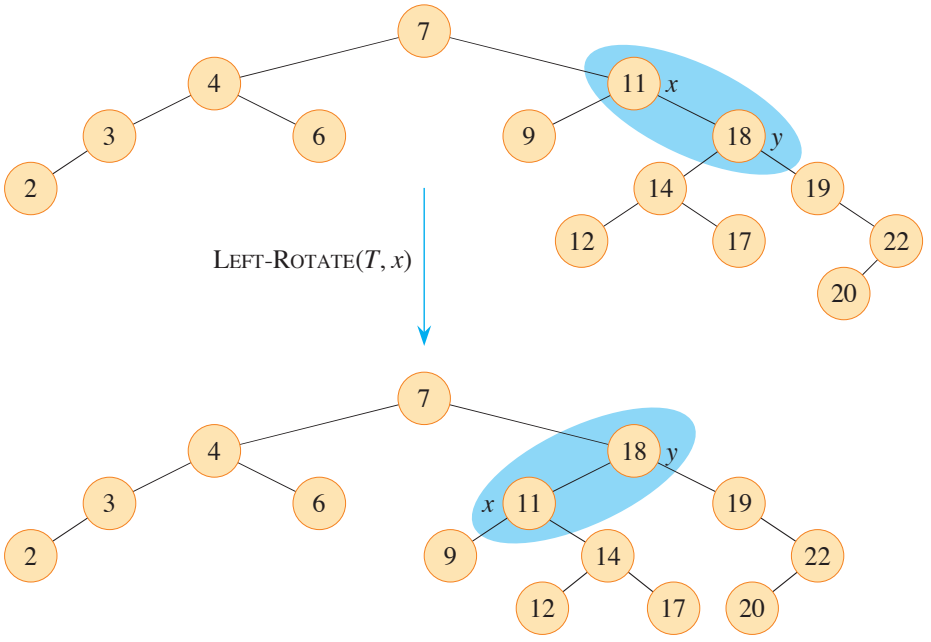

Figure 13.3 An example of how the procedure LEFT-ROTATE(T, x) modifies a binary search tree. Inorder tree walks of the input tree and the modified tree produce the same listing of key values.

13.2-2

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

13.2-3

Let a, b, and c be arbitrary nodes in subtrees α, β, and γ, respectively, in the right 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

In order to insert a node into a red-black tree with n internal nodes in $O(\lg n)$ time and maintain the red-black properties, we'll need to slightly modify the TREE-INSERT procedure on page 321. The procedure RB-INSERT starts by inserting node z into the tree T as if it were an ordinary binary search tree, and then it colors z red. (Exercise 13.3-1 asks you to explain why to make node z red rather than black.) To guarantee that the red-black properties are preserved, an auxiliary procedure RB-INSERT-FIXUP on the facing page recolors nodes and performs 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  x = T.root                  // node being compared with z
 2  y = T.nil                   // y will be parent of z
 3  while x ≠ T.nil             // descend until reaching the sentinel
 4      y = x
 5      if z.key < x.key
 6          x = x.left
 7      else x = x.right
 8  z.p = y                     // found the location—insert z with parent y
 9  if y == T.nil
10      T.root = z              // tree T was empty
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
14  z.left = T.nil              // both of z's children are the sentinel
15  z.right = T.nil
16  z.color = RED               // the new node starts out red
17  RB-INSERT-FIXUP(T, z)       // correct any violations of red-black properties
```

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, lines 14–15 of RB-INSERT set $z.left$ and $z.right$ to $T.nil$, in order to maintain the proper tree structure. (TREE-INSERT assumed that z's children were already NIL.) Third, line 16 colors z red. Fourth, because coloring z red may cause a violation of one of the red-black properties, line 17 of RB-INSERT calls RB-INSERT-FIXUP(T, z) in order 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                  // is z's parent a left child?
 3          y = z.p.p.right                   // y is z's uncle
 4          if y.color == RED                 // are z's parent and uncle both red?
 5              z.p.color = BLACK             }
 6              y.color = BLACK               } case 1
 7              z.p.p.color = RED             }
 8              z = z.p.p                     }
 9          else
10              if z == z.p.right
11                  z = z.p                   }
12                  LEFT-ROTATE(T, z)         } case 2
13              z.p.color = BLACK             }
14              z.p.p.color = RED             } case 3
15              RIGHT-ROTATE(T, z.p.p)        }
16      else // same as lines 3–15, but with "right" and "left" exchanged
17          y = z.p.p.left
18          if y.color == RED
19              z.p.color = BLACK
20              y.color = BLACK
21              z.p.p.color = RED
22              z = z.p.p
23          else
24              if z == z.p.left
25                  z = z.p
26                  RIGHT-ROTATE(T, z)
27              z.p.color = BLACK
28              z.p.p.color = RED
29              LEFT-ROTATE(T, z.p.p)
30  T.root.color = BLACK
```

To understand how RB-INSERT-FIXUP works, let's examine the code in three major steps. First, we'll determine which violations of the red-black properties might arise in RB-INSERT upon inserting node z and coloring it red. Second, we'll consider the overall goal of the **while** loop in lines 1–29. Finally, we'll explore each of the three cases within the **while** loop's body (case 2 falls through into case 3, so these two cases are not mutually exclusive) and see how they accomplish the goal.

In describing the structure of a red-black tree, we'll often need to refer to the sibling of a node's parent. We use the term ***uncle*** for such a node.[1] Figure 13.4 shows how RB-INSERT-FIXUP operates on a sample red-black tree, with cases depending in part on the colors of a node, its parent, and its uncle.

What violations of the red-black properties might occur upon the call to RB-INSERT-FIXUP? Property 1 certainly continues to hold (every node is either red or black), as does property 3 (every leaf is black), 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 may arise because z is 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.

The **while** loop of lines 1–29 has two symmetric possibilities: lines 3–15 deal with the situation in which node z's parent $z.p$ is a left child of z's grandparent $z.p.p$, and lines 17–29 apply when z's parent is a right child. Our proof will focus only on lines 3–15, relying on the symmetry in lines 17–29.

We'll show that the **while** loop 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, but not both. 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'll 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. Part (b) will help show that z's grandparent $z.p.p$ exists when it's referenced in lines 2, 3, 7, 8, 14, and 15 (recall that we're focusing only on lines 3–15).

---

[1] Although we try to avoid gendered language in this book, the English language lacks a gender-neutral word for a parent's sibling.

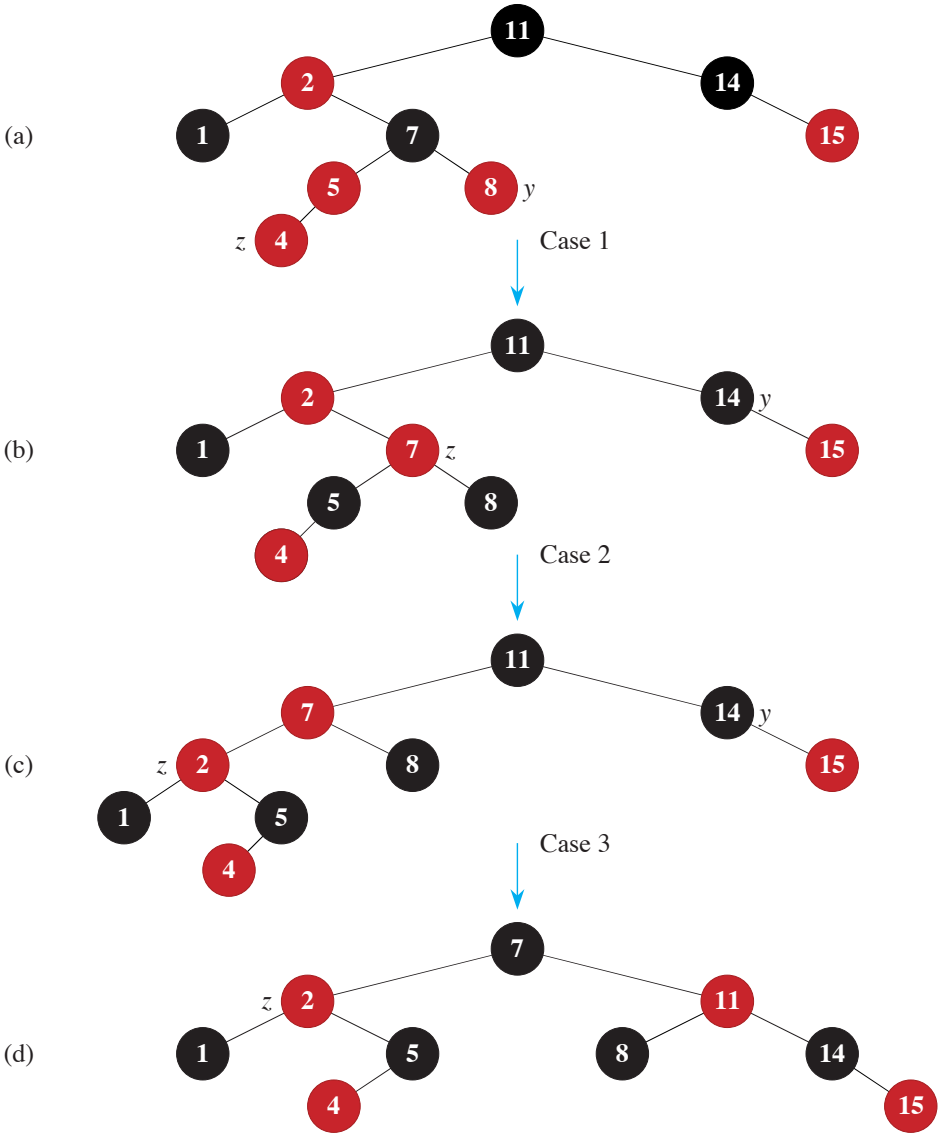

Figure 13.4 The operation of RB-INSERT-FIXUP. **(a)** A node z after insertion. Because both z and its parent $z.p$ are red, a violation of property 4 occurs. Since z's uncle y is red, case 1 in the code applies. Node z's grandparent $z.p.p$ must be black, and its blackness transfers down one level to z's parent and uncle. Once the pointer z moves up two levels in the tree, the tree shown in **(b)** results. Once again, z and its parent are both red, but this time z's uncle y is black. Since z is the right child of $z.p$, case 2 applies. Performing a left rotation results in the tree 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.

Recall that to use a loop invariant, we need to show that the invariant is true upon entering the first iteration of the loop, that each iteration maintains it, that the loop terminates, and that the loop invariant gives us a useful property at loop termination. We'll see that each iteration of the loop has two possible outcomes: either the pointer z moves up the tree, or some rotations occur and then the loop terminates.

Initialization: Before RB-INSERT is called, the red-black tree has no violations. RB-INSERT adds a red node z and calls RB-INSERT-FIXUP. We'll 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 before 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 (the root must be black), 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 (both children of a red node are black). 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.

Maintenance: There are six cases within the **while** loop, but we'll examine only the three cases in lines 3–15, when node z's parent $z.p$ is a left child of z's grandparent $z.p.p$. The proof for lines 17–29 is symmetric. 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 RB-INSERT-FIXUP enters a loop iteration only if $z.p$ is red, we know that $z.p$ cannot be the root. Hence, $z.p.p$ exists.

Case 1 differs from cases 2 and 3 by the color of z's uncle y. 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 case 1 executes. 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 $z.p$.

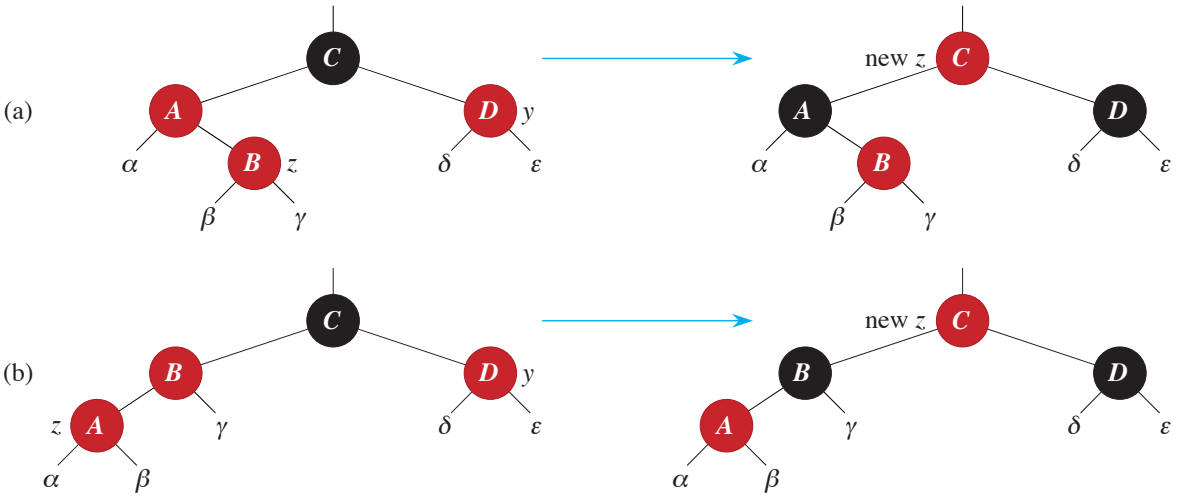

Figure 13.5 Case 1 of the procedure RB-INSERT-FIXUP. Both z and its parent $z.p$ are red, violating property 4. In case 1, z's uncle y is red. The same action occurs regardless of whether **(a)** z is a right child or **(b)** z is a left child. Each of the subtrees α, β, γ, δ, and ε has a black root—possibly the sentinel—and each has the same black-height. The code for case 1 moves the blackness of z's grandparent down to z's parent and uncle, 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. If the action of case 1 causes a new violation of property 4 to occur, it must be only between the new z, which is red, and its parent, if it is red as well.

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's grandparent $z.p.p$ is black, its blackness can transfer down one level to both $z.p$ and y, thereby fixing the problem of z and $z.p$ both being red. Having had its blackness transferred down one level, z's grandparent becomes red, thereby maintaining property 5. The **while** loop repeats with $z.p.p$ as the new node z, so that 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.

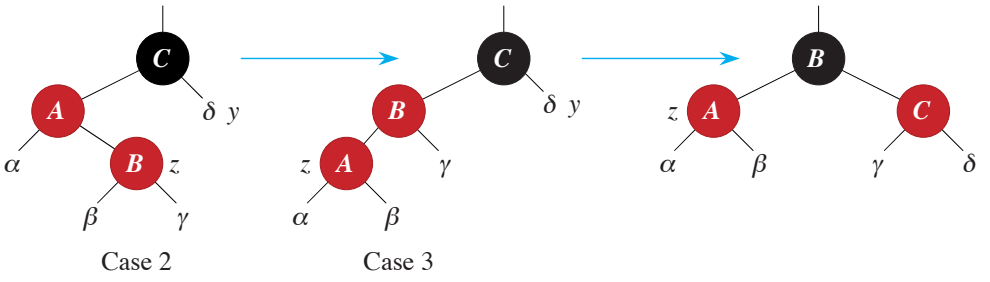

Figure 13.6 Cases 2 and 3 of the procedure RB-INSERT-FIXUP. As in case 1, property 4 is violated in either case 2 or case 3 because z and its parent $z.p$ are both red. Each of the subtrees α, β, γ, and δ has a black root (α, β, and γ from property 4, and δ because otherwise case 1 would apply), and each has the same black-height. Case 2 transforms 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.

c. We have already argued that case 1 maintains property 5, and it does not introduce a violation of properties 1 or 3.

   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, which assume that z's parent $z.p$ is red and a left child, according to whether z is a right or left child of $z.p$. Lines 11–12 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. A left rotation immediately transforms the situation into case 3 (lines 13–15), in which node z is a left child. Because both z and $z.p$ are red, the rotation affects neither the black-heights of nodes nor property 5. Whether case 3 executes directly or through case 2, z's uncle y is black, since otherwise case 1 would have run. 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 11 and then down one level in line 12, the identity of $z.p.p$ remains unchanged. Case 3 performs some color changes and a right rotation, which preserve property 5. At this point, there are no longer two red nodes in a row. The **while** loop terminates upon the next test in line 1, 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.

Termination: To see that the loop terminates, observe that if only case 1 occurs, then the node pointer z moves toward the root in each iteration, so that eventually $z.p$ is black. (If z is the root, then $z.p$ is the sentinel $T.nil$, which is black.) If either case 2 or case 3 occurs, then we've seen that the loop terminates. Since the loop terminates because $z.p$ is black, 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 30 restores this property by coloring the root black, so that when RB-INSERT-FIXUP terminates, all the red-black properties hold.

Thus, 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
Line 16 of RB-INSERT sets the color of the newly inserted node z to red. If instead z's color were set to black, then property 4 of a red-black tree would not be violated. Why not 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 $O(\lg n)$ time. Deleting a node from a red-black tree is more complicated than inserting a node.

The procedure for deleting a node from a red-black tree is based on the TREE-DELETE procedure on page 325. First, we need to customize the TRANSPLANT

subroutine on page 324 that TREE-DELETE calls so that it applies to a red-black tree. Like TRANSPLANT, the new procedure RB-TRANSPLANT replaces the subtree rooted at node u by the subtree rooted at node v. The RB-TRANSPLANT procedure 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: the procedure can assign to $v.p$ even if v points to the sentinel. We'll take advantage of the ability to assign to $v.p$ when $v = T.nil$.

```
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-DELETE on the next page is like the TREE-DELETE procedure, but with additional lines of pseudocode. The additional lines deal with nodes x and y that may be involved in violations of the red-black properties. When the node z being deleted has at most one child, then y will be z. When z has two children, then, as in TREE-DELETE, y will be z's successor, which has no left child and moves into z's position in the tree. Additionally, y takes on z's color. In either case, node y has at most one child: node x, which takes y's place in the tree. (Node x will be the sentinel $T.nil$ if y has no children.) Since node y will be either removed from the tree or moved within the tree, the procedure needs to keep track of y's original color. If the red-black properties might be violated after deleting node z, RB-DELETE calls the auxiliary procedure RB-DELETE-FIXUP, which changes colors and performs rotations to restore the red-black properties.

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.

In detail, here are the other differences between the two procedures:

- Lines 1 and 9 set node y as described above: line 1 when node z has at most one child and line 9 when z has two children.
- 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 node z has two children, then nodes y and z are

```
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)          // replace z by its right child
 6  elseif z.right == T.nil
 7       x = z.left
 8       RB-TRANSPLANT(T, z, z.left)           // replace z by its left child
 9  else y = TREE-MINIMUM(z.right)             // y is z's successor
10       y-original-color = y.color
11       x = y.right
12       if y ≠ z.right                        // is y farther down the tree?
13            RB-TRANSPLANT(T, y, y.right)     // replace y by its right child
14            y.right = z.right                // z's right child becomes
15            y.right.p = y                    //       y's right child
16       else x.p = y                          // in case x is T.nil
17       RB-TRANSPLANT(T, z, y)                // replace z by its successor y
18       y.left = z.left                       // and give z's left child to y,
19       y.left.p = y                          //       which had no left child
20       y.color = z.color
21  if y-original-color == BLACK               // if any red-black violations occurred,
22       RB-DELETE-FIXUP(T, x)                 //       correct them
```

distinct. In this case, line 17 moves y into z's original position in the tree (that is, z's location in the tree at the time RB-DELETE was called), and line 20 gives y the same color as z. When node y was originally black, removing or moving it could cause violations of the red-black properties, which are corrected by the call of RB-DELETE-FIXUP in line 22.

- As discussed, the procedure keeps track of the node x that moves into node y's original position at the time of call. 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 $T.nil$.
- Since node x moves into node y's original position, the attribute $x.p$ must be set correctly. If node z has two children and y is z's right child, then y just moves into z's position, with x remaining a child of y. Line 12 checks for this case. Although you might think that setting $x.p$ to y in line 16 is unnecessary since x is a child of y, the call of RB-DELETE-FIXUP relies on $x.p$ being y even if x is $T.nil$. Thus, when z has two children and y is z's right child, executing

line 16 is necessary if y's right child is $T.nil$, and otherwise it does not change anything.

Otherwise, node z is either the same as node y or it is a proper ancestor of y's original parent. In these cases, the calls of RB-TRANSPLANT in lines 5, 8, and 13 set $x.p$ correctly in line 6 of RB-TRANSPLANT. (In these calls of RB-TRANSPLANT, the third parameter passed is the same as x.)

- Finally, if node y was black, one or more violations of the red-black properties might arise. The call of RB-DELETE-FIXUP in line 22 restores 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. (See Exercise 13.4-1.)
  2. No red nodes have been made adjacent. If z has at most one child, then y and z are the same node. That node is removed, with a child taking its place. If the removed node was red, then neither its parent nor its children can also be red, so moving a child to take its place cannot cause two red nodes to become adjacent. If, on the other hand, z has two children, then y takes z's place in the tree, along with z's color, so there cannot be 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. Since y is red, x must be black, and so replacing y by x cannot cause two red nodes to become adjacent.
  3. Because 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 was the root and a red child of y became the new root, property 2 is violated. Second, if both x and its new parent are red, then a violation of property 4 occurs. Third, moving y within the tree causes any simple path that previously contained y to have one less 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 when the black node y is removed or moved, its blackness transfers to the node x that moves into y's original position, giving x 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. But now another problem emerges: 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.

The procedure RB-DELETE-FIXUP on the next page restores properties 1, 2, and 4. Exercises 13.4-2 and 13.4-3 ask you to show that the procedure restores properties 2 and 4, and so in the remainder of this section, we focus on property 1. The goal of the **while** loop in lines 1–43 is to move the extra black up the tree until

1. x points to a red-and-black node, in which case line 44 colors x (singly) black;
2. x points to the root, in which case the extra black simply vanishes; or
3. having performed suitable rotations and recolorings, the loop exits.

Like RB-INSERT-FIXUP, the RB-DELETE-FIXUP procedure handles two symmetric situations: lines 3–22 for when node x is a left child, and lines 24–43 for when x is a right child. Our proof focuses on the four cases shown in lines 3–22.

Within the **while** loop, x always points to a nonroot doubly black node. Line 2 determines whether x is a left child or a right child of its parent $x.p$ so that either lines 3–22 or 24–43 will execute in a given iteration. The sibling of x is always denoted by a pointer w. 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.

Recall that the RB-DELETE procedure always assigns to $x.p$ before calling RB-DELETE-FIXUP (either within the call of RB-TRANSPLANT in line 13 or the assignment in line 16), even when node x is the sentinel $T.nil$. That is because RB-DELETE-FIXUP references x's parent $x.p$ in several places, and this attribute must point to the node that became x's parent in RB-DELETE—even if x is $T.nil$.

Figure 13.7 demonstrates the four cases in the code when node x is a left child. (As in RB-INSERT-FIXUP, the cases in RB-DELETE-FIXUP are not mutually exclusive.) 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 the roots of 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 the root of 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 the root of any of γ, δ, ε, and ζ is 2, both before and after the transformation.[2] 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.

---

[2] If property 5 holds, we can assume that paths from the roots of $\gamma, \delta, \varepsilon$, and ζ down to leaves contain one more black than do paths from the roots of α and β down to leaves.

```
RB-DELETE-FIXUP(T, x)
 1  while x ≠ T.root and x.color == BLACK
 2      if x == x.p.left                    // is x a left child?
 3          w = x.p.right                   // w is x's sibling
 4          if w.color == RED
 5              w.color = BLACK
 6              x.p.color = RED
 7              LEFT-ROTATE(T, x.p)                          } case 1
 8              w = x.p.right
 9          if w.left.color == BLACK and w.right.color == BLACK
10              w.color = RED                                } case 2
11              x = x.p
12          else
13              if w.right.color == BLACK
14                  w.left.color = BLACK
15                  w.color = RED
16                  RIGHT-ROTATE(T, w)                       } case 3
17                  w = x.p.right
18              w.color = x.p.color
19              x.p.color = BLACK
20              w.right.color = BLACK                        } case 4
21              LEFT-ROTATE(T, x.p)
22              x = T.root
23      else // same as lines 3–22, but with "right" and "left" exchanged
24          w = x.p.left
25          if w.color == RED
26              w.color = BLACK
27              x.p.color = RED
28              RIGHT-ROTATE(T, x.p)
29              w = x.p.left
30          if w.right.color == BLACK and w.left.color == BLACK
31              w.color = RED
32              x = x.p
33          else
34              if w.left.color == BLACK
35                  w.right.color = BLACK
36                  w.color = RED
37                  LEFT-ROTATE(T, w)
38                  w = x.p.left
39              w.color = x.p.color
40              x.p.color = BLACK
41              w.left.color = BLACK
42              RIGHT-ROTATE(T, x.p)
43              x = T.root
44  x.color = BLACK
```

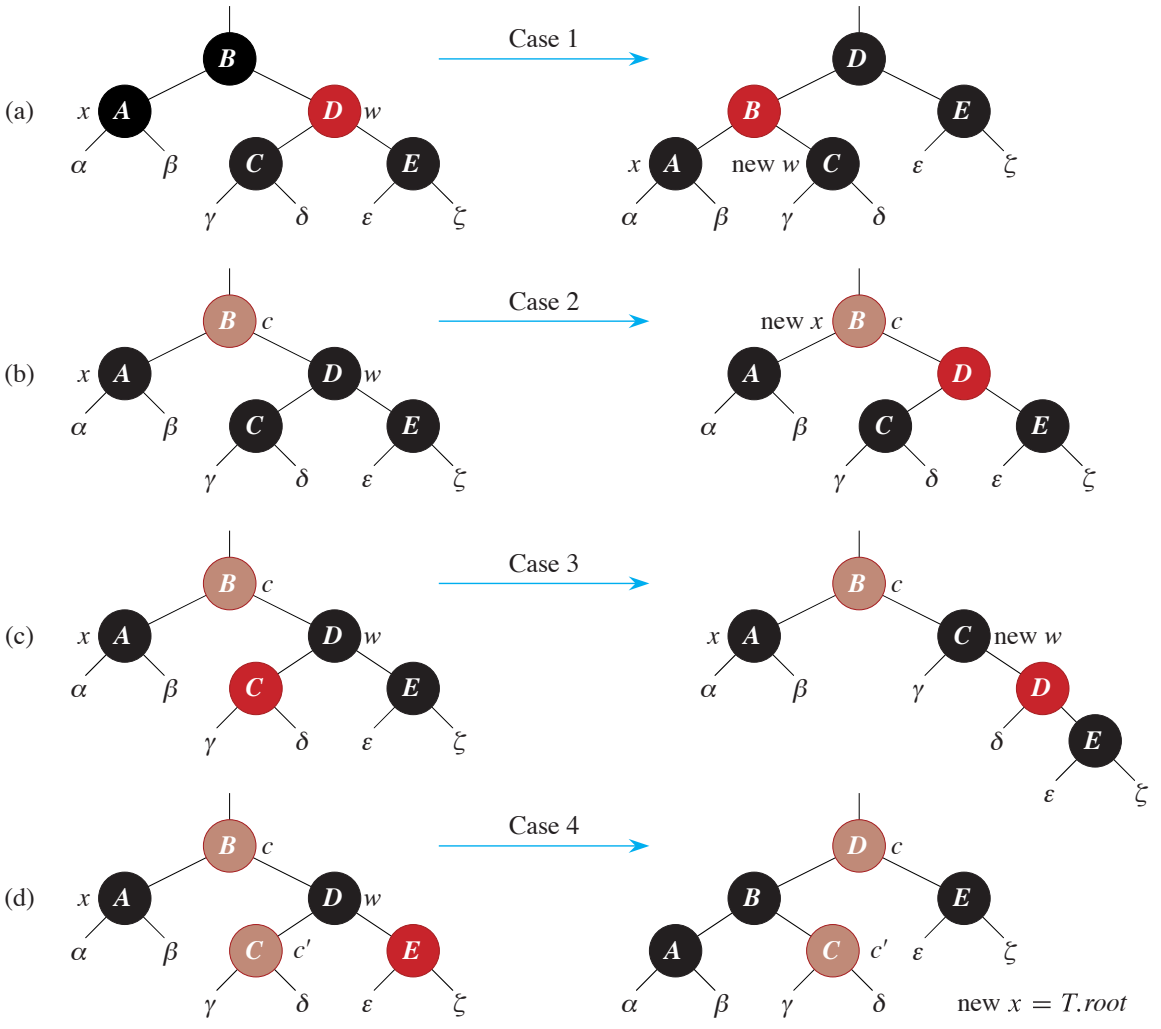

Figure 13.7 The cases in lines 3–22 of the procedure RB-DELETE-FIXUP. Brown nodes have *color* attributes represented by c and c', which may be either RED or BLACK. The letters $\alpha, \beta, \ldots, \zeta$ represent arbitrary subtrees. Each case transforms the configuration on the left into the configuration 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 into 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 case 2 is entered 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.

If we define count(RED) $= 0$ and count(BLACK) $= 1$, then the number of black nodes from the root to α is $2 +$ 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 13.4-6).

Case 1: x's sibling w is red

Case 1 (lines 5–8 and Figure 13.7(a)) occurs when node w, the sibling of node x, is red. Because w is red, it must have black children. This case switches the colors of w and $x.p$ and then performs 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 case 1 converts into one of cases 2, 3, or 4.

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

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

In case 2 (lines 10–11 and Figure 13.7(b)), both of w's children are black. Since w is also black, this case removes one black from both x and w, leaving x with only one black and leaving w red. To compensate for x and w each losing one black, x's parent $x.p$ can take on an extra black. Line 11 does so by moving x up one level, so that the **while** loop repeats with $x.p$ as the new node x. If case 2 enters 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. Line 44 then colors the new node x (singly) black.

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

Case 3 (lines 14–17 and Figure 13.7(c)) occurs when w is black, its left child is red, and its right child is black. This case switches the colors of w and its left child $w.left$ and then performs 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 case 3 falls through into case 4.

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

Case 4 (lines 18–22 and Figure 13.7(d)) occurs when node x's sibling w is black and w's right child is red. Some color changes and a left rotation on $x.p$ allow the extra black on x to vanish, making it singly black, without violating any of the red-black properties. Line 22 sets x to be the root, and the **while** loop terminates when it next 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 $O(\lg n)$.

Exercises

13.4-1
Show that if node y in RB-DELETE is red, then no black-heights change.

13.4-2
Argue that after RB-DELETE-FIXUP executes, the root of the tree must be black.

13.4-3
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).

13.4-4
In Exercise 13.3-2 on page 346, 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 $8, 12, 19, 31, 38, 41$.

13.4-5
Which lines of the code for RB-DELETE-FIXUP might examine or modify the sentinel $T.nil$?

13.4-6
In each of the cases of Figure 13.7, give the count of black nodes from the root of the subtree shown to the roots of each of the subtrees $\alpha, \beta, \ldots, \zeta$, 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.

13.4-7
Professors Skelton and Baron worry that at the start of case 1 of RB-DELETE-FIXUP, the node $x.p$ might not be black. If $x.p$ is not black, then lines 5–6 are

wrong. Show that $x.p$ must be black at the start of case 1, so that the professors need not be concerned.

13.4-8

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 always the same as the initial red-black tree? Justify your answer.

★ ***13.4-9***

Consider the operation RB-ENUMERATE(T, r, a, b), which outputs all the keys k such that $a \le k \le b$ in a subtree rooted at node r in an n-node red-black tree T. Describe how to implement RB-ENUMERATE in $\Theta(m + \lg n)$ time, where m is the number of keys that are output. Assume that the keys in T are unique and that the values a and b appear as keys in T. How does your solution change if a and b might not appear in T?

Problems

13-1 Persistent dynamic sets

During the course of an algorithm, you sometimes find that you 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 a lot of space. Sometimes, you can do much better.

Consider a persistent set S with the operations INSERT, DELETE, and SEARCH, which you implement using binary search trees as shown in Figure 13.8(a). Maintain a separate root for every version of the set. In order to insert the key 5 into the set, create a new node with key 5. This node becomes the left child of a new node with key 7, since you 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. Thus, you 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 on page 346.)

a. For a persistent binary search tree (not a red-black tree, just a binary search tree), identify the nodes that need to change to insert or delete a node.

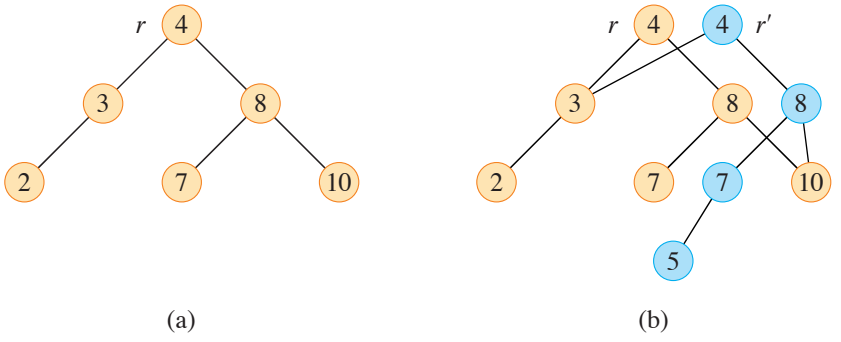

Figure 13.8 **(a)** A binary search tree with keys $2, 3, 4, 7, 8, 10$. **(b)** The persistent binary search tree that results from the insertion of key 5. The most recent version of the set consists of the nodes reachable from the root r', and the previous version consists of the nodes reachable from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

b. Write a procedure PERSISTENT-TREE-INSERT(T, z) that, given a persistent binary search tree T and a node z to insert, returns a new persistent tree T' that is the result of inserting z into T. Assume that you have a procedure COPY-NODE(x) that makes a copy of node x, including all of its attributes.

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 nodes that are copied.)

d. Suppose that you include the parent attribute in each node. In this case, the PERSISTENT-TREE-INSERT procedure needs to perform additional copying. Prove that PERSISTENT-TREE-INSERT then requires $\Omega(n)$ 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. You may assume that all keys are distinct.

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 $x_1.key \le x.key \le x_2.key$. 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. Suppose that you store the black-height of a red-black tree T 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 how to determine the black-height of each node visited while descending through T, using $O(1)$ time per node visited.

Let T_1 and T_2 be red-black trees and x be a key value such that for any nodes x_1 in T_1 and x_2 in T_2, we have $x_1.key \leq x.key \leq x_2.key$. You will show how 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 you 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, maintain an extra attribute h in each node such that $x.h$ is the height of node x. As for any other binary search tree T, 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, 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, so that $|x.right.h - x.left.h| \leq 2$, and alters the subtree rooted at x to be height balanced. The procedure should return a pointer to the node that is the root of the subtree after alterations occur. (*Hint:* Use rotations.)

c. Using part (b), describe a recursive procedure AVL-INSERT(T, z) that takes an AVL tree T and a newly created node z (whose key has already been filled in), and adds z into T, maintaining the property that T is an AVL tree. As in TREE-INSERT from Section 12.3, assume that $z.key$ has already been filled in and that $z.left = \text{NIL}$ and $z.right = \text{NIL}$. Assume as well that $z.h = 0$.

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

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, where each node has either two or three children. Chapter 18 covers a generalization of 2-3 trees introduced by Bayer and McCreight [39], called "B-trees."

Red-black trees were invented by Bayer [38] under the name "symmetric binary B-trees." Guibas and Sedgewick [202] studied their properties at length and introduced the red/black color convention. Andersson [16] gives a simpler-to-code variant of red-black trees. Weiss [451] calls this variant AA-trees. An AA-tree is similar to a red-black tree except that left children can never be red.

Sedgewick and Wayne [402] present red-black trees as a modified version of 2-3 trees in which a node with three children is split into two nodes with two children each. One of these nodes becomes the left child of the other, and only left children can be red. They call this structure a "left-leaning red-black binary search tree." Although the code for left-leaning red-black binary search trees is more concise than the red-black tree pseudocode in this chapter, operations on left-leaning red-black binary search trees do not limit the number of rotations per operation to a constant. This distinction will matter in Chapter 17.

Treaps, a hybrid of binary search trees and heaps, were proposed by Seidel and Aragon [404]. They are the default implementation of a dictionary in LEDA [324], which is a well-implemented collection of data structures and algorithms.

There are many other variations on balanced binary trees, including weight-balanced trees [344], k-neighbor trees [318], and scapegoat trees [174]. Perhaps

the most intriguing are the "splay trees" introduced by Sleator and Tarjan [418], which are "self-adjusting." (See Tarjan [429] 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 16) of each operation on an n-node tree is $O(\lg n)$. Splay trees have been conjectured to perform within a constant factor of the best offline rotation-based tree. The best known competitive ratio (see Chapter 27) for a rotation-based tree is the Tango Tree of Demaine et al. [109].

Skip lists [369] 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 $O(\lg n)$ expected time on a skip list of n items.