13 Red-Black Trees

Chapter 12 showed that a binary search tree of height h can implement 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; but if its height is large, their performance may be no better 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 way nodes can be colored on any 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 fields *color*, *key*, *left*, *right*, and *p*. If a child or the parent of a node does not exist, the corresponding pointer field of the node contains the value NIL. We shall regard these NIL's as being pointers to external nodes (leaves) of the binary search tree and the normal, key-bearing nodes as being internal nodes of the tree.

A binary search tree is a red-black tree if it 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 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 206). For a red-black tree T, the sentinel $nil[T]$ is an object with the same fields as an ordinary node in the tree. Its *color* field is BLACK, and its other fields—p, *left*, *right*, and *key*—can be set to arbitrary values. As Figure 13.1(b) shows, all pointers to NIL are replaced by pointers to the sentinel $nil[T]$.

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 $nil[T]$ to represent all the NIL's—all leaves and the root's parent. The values of the fields 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 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 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 ($nil[T]$), 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

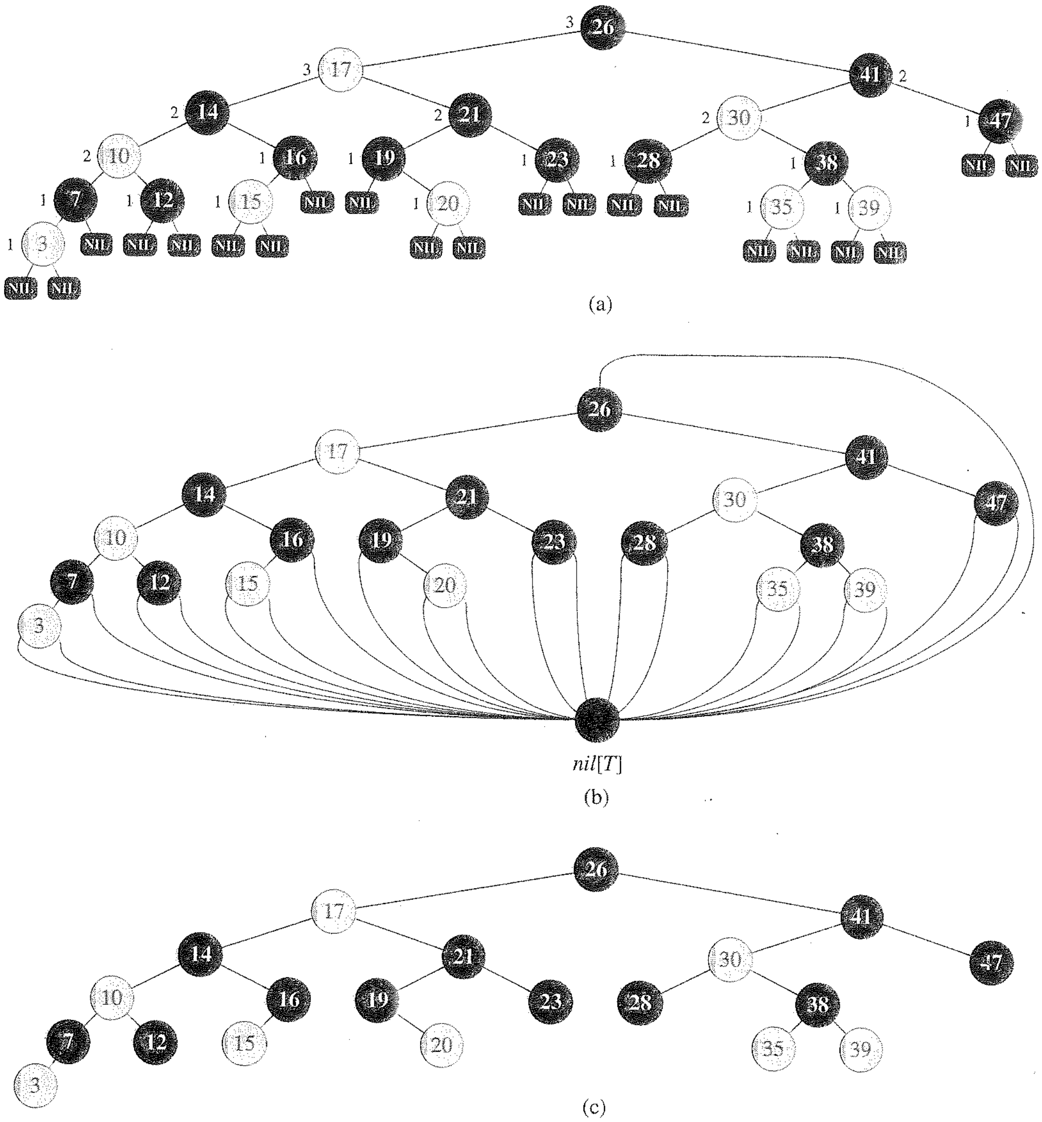

Figure 13.1 A red-black tree with black nodes darkened and red nodes shaded. Every node in a red-black tree is either red or black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. **(a)** Every leaf, shown as a NIL, is black. Each non-NIL node is marked with its black-height; NIL's have black-height 0. **(b)** The same red-black tree but with each NIL replaced by the single sentinel *nil*[*T*], 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 $h \leq 2\lg(n+1)$. ■

An immediate consequence of this lemma is that the dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR can be implemented in $O(\lg n)$ time on red-black trees, since they can be made to run in $O(h)$ time on a search tree of height h (as shown in Chapter 12) and any red-black tree on n nodes is a search tree with height $O(\lg n)$. (Of course, references to NIL in the algorithms of Chapter 12 would have to be replaced by $nil[T]$.) 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, that these two operations can indeed be supported 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.

Sol *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.

Sol good

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

easy

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 $nil[T]$; x may be any node in the tree whose right child is not $nil[T]$. 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 $right[x] \neq nil[T]$ and that the root's parent is $nil[T]$.

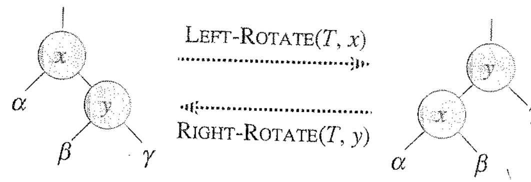

Figure 13.2 The rotation operations on a binary search tree. The operation LEFT-ROTATE(T, x) transforms the configuration of the two nodes on the left into the configuration on the right by changing a constant number of pointers. The configuration on the right can be transformed into the configuration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

```
LEFT-ROTATE(T, x)
  y ← right[x]                  ▷ Set y.
  right[x] ← left[y]            ▷ Turn y's left subtree into x's right subtree.
  p[left[y]] ← x
  p[y] ← p[x]                   ▷ Link x's parent to y.
  if p[x] = nil[T]
     then root[T] ← y
     else if x = left[p[x]]
             then left[p[x]] ← y
             else right[p[x]] ← y
  left[y] ← x                   ▷ Put x on y's left.
  p[x] ← y
```

Figure 13.3 shows how LEFT-ROTATE operates. 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 fields 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.

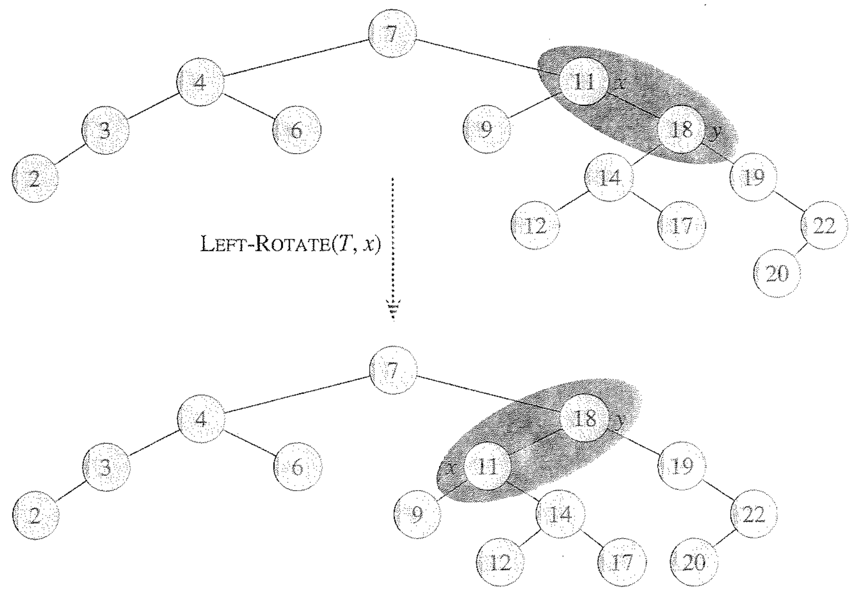

Figure 13.3 An example of how the procedure LEFT-ROTATE(T, x) modifies a binary search tree. Inorder tree walks of the input tree and the modified tree produce the same listing of key values.

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

Insertion of a node into an n-node red-black tree can be accomplished in $O(\lg n)$ time. 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. 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* field is assumed to have already been filled in, into the red-black tree T.

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

There are four differences between the procedures TREE-INSERT and RB-INSERT. First, all instances of NIL in TREE-INSERT are replaced by *nil*$[T]$. Second, we set *left*$[z]$ and *right*$[z]$ to *nil*$[T]$ 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 color[p[z]] = RED
 2      do if p[z] = left[p[p[z]]]
 3            then y ← right[p[p[z]]]
 4                 if color[y] = RED
 5                   then color[p[z]] ← BLACK                    ▷ Case 1
 6                        color[y] ← BLACK                       ▷ Case 1
 7                        color[p[p[z]]] ← RED                   ▷ Case 1
 8                        z ← p[p[z]]                            ▷ Case 1
 9                   else if z = right[p[z]]
10                          then z ← p[z]                        ▷ Case 2
11                               LEFT-ROTATE(T, z)               ▷ Case 2
12                        color[p[z]] ← BLACK                    ▷ Case 3
13                        color[p[p[z]]] ← RED                   ▷ Case 3
14                        RIGHT-ROTATE(T, p[p[z]])               ▷ Case 3
15            else (same as then clause
                        with "right" and "left" exchanged)
16  color[root[T]] ← 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 the 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] into which the **while** loop is broken 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 can 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 $nil[T]$. Property 5, which says that the number of black nodes is the same on every 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.

The **while** loop in lines 1–15 maintains the following three-part invariant:

---

[1]Case 2 falls through into case 3, and so these two cases are not mutually exclusive.

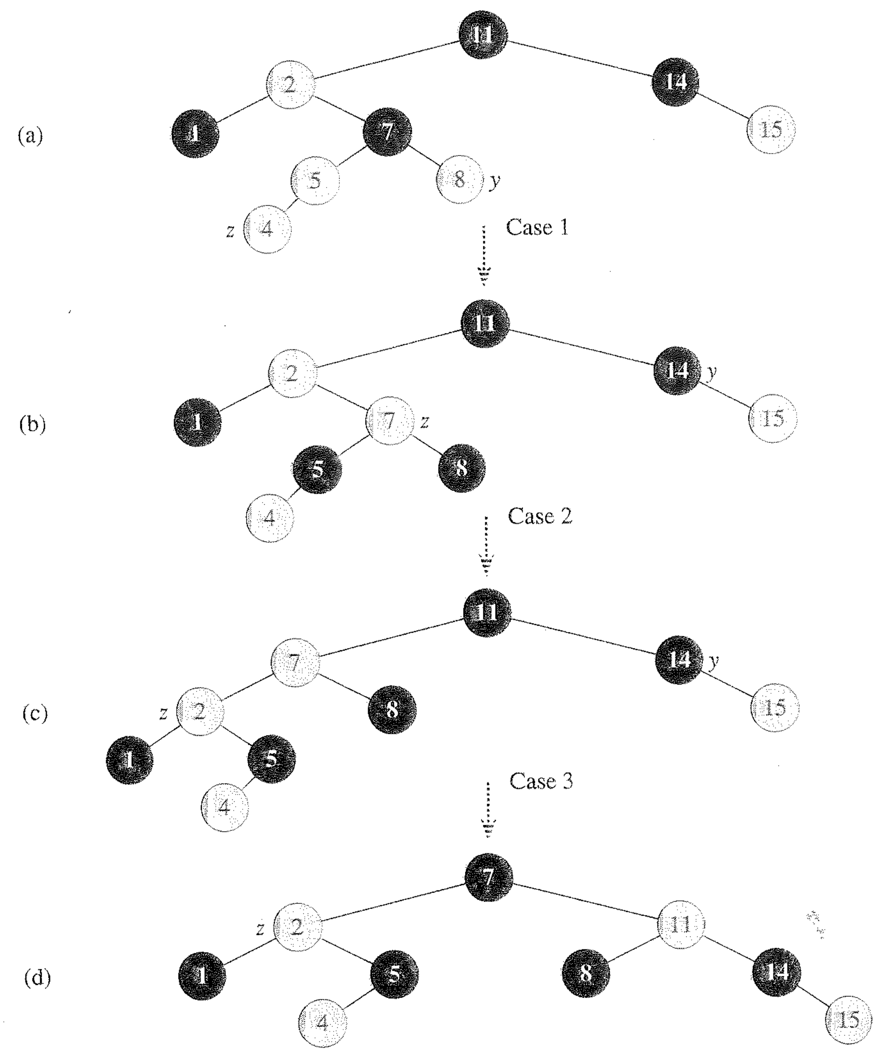

Figure 13.4 The operation of RB-INSERT-FIXUP. **(a)** A node z after insertion. Since z and its parent $p[z]$ are both red, a violation of property 4 occurs. Since z's uncle y is red, case 1 in the code can be applied. Nodes are recolored and the pointer z is moved up the tree, resulting in the tree 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 $p[z]$, case 2 can be applied. A left rotation is performed, and the tree that results is shown in **(c)**. Now z is the left child of its parent, and case 3 can be applied. A right rotation yields the tree in **(d)**, which is a legal red-black tree.

At the start of each iteration of the loop,

a. Node z is red.

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

c. If there is a violation of the red-black properties, there is at most one violation, and it is of either property 2 or property 4. If there is a violation of property 2, it occurs because z is the root and is red. If there is a violation of property 4, it occurs because both z and $p[z]$ 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 will be focusing on node z and nodes near it in the tree, it is helpful to know from part (a) that z is red. We shall use part (b) to show that the node $p[p[z]]$ 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 will also demonstrate that there are two possible outcomes of each iteration of the loop: the pointer z moves up the tree, or some rotations are performed and 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 $p[z]$ is the root, then $p[z]$ 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 there is a violation of 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, there is not also a violation of property 4. Thus, this violation of property 2 is the only violation of red-black properties in the entire tree.

If there is a violation of 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 $p[z]$ are red. Moreover, there are

Termination: When the loop terminates, it does so because $p[z]$ is black. (If z is the root, then $p[z]$ is the sentinel $nil[T]$, which is black.) Thus, there is no violation of 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: There are actually six cases to consider in the **while** loop, but three of them are symmetric to the other three, depending on whether z's parent $p[z]$ is a left child or a right child of z's grandparent $p[p[z]]$, which is determined in line 2. We have given the code only for the situation in which $p[z]$ is a left child. The node $p[p[z]]$ exists, since by part (b) of the loop invariant, if $p[z]$ is the root, then $p[z]$ is black. Since we enter a loop iteration only if $p[z]$ is red, we know that $p[z]$ cannot be the root. Hence, $p[p[z]]$ exists.

Case 1 is distinguished 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 $right[p[p[z]]]$, and a test is made in line 4. If y is red, then case 1 is executed. Otherwise, control passes to cases 2 and 3. In all three cases, z's grandparent $p[p[z]]$ is black, since its parent $p[z]$ is red, and property 4 is violated only between z and $p[z]$.

Case 1: z's uncle y is red

Figure 13.5 shows the situation for case 1 (lines 5–8). Case 1 is executed when both $p[z]$ and y are red. Since $p[p[z]]$ is black, we can color both $p[z]$ and y black, thereby fixing the problem of z and $p[z]$ both being red, and color $p[p[z]]$ red, thereby maintaining property 5. We then repeat the **while** loop with $p[p[z]]$ 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 $z' = p[p[z]]$ to denote the node z at the test in line 1 upon the next iteration.

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

b. The node $p[z']$ is $p[p[p[z]]]$ 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 clearly 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'.

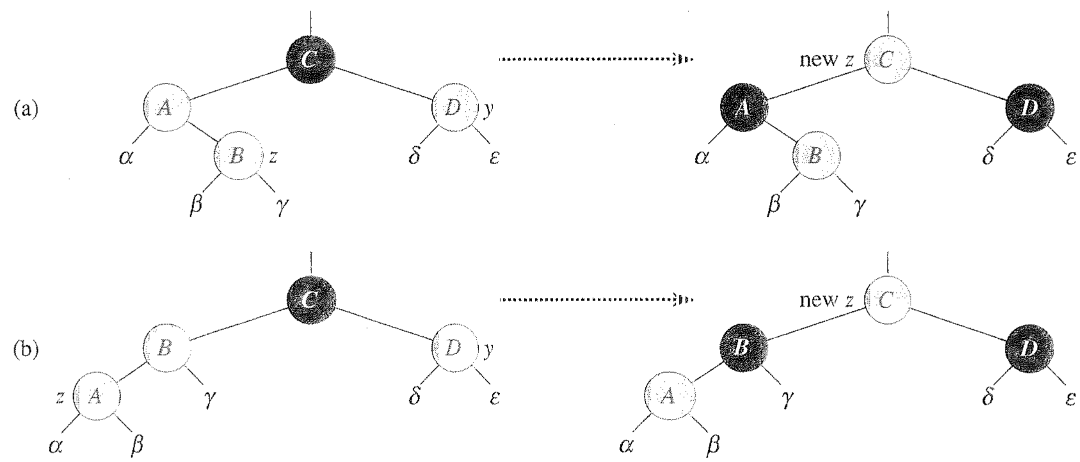

Figure 13.5 Case 1 of the procedure RB-INSERT. Property 4 is violated, since z and its parent $p[z]$ are both red. The same action is taken whether **(a)** z is a right child or **(b)** z is a left child. Each of the subtrees α, β, γ, δ, and ε has a black root, and each has the same black-height. The code for case 1 changes the colors of some nodes, preserving property 5: all downward paths from a node to a leaf have the same number of blacks. The **while** loop continues with node z's grandparent $p[p[z]]$ 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 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 $p[z']$ alone. If $p[z']$ was black, there is no violation of property 4. If $p[z']$ was red, coloring z' red created one violation of property 4 between z' and $p[z']$.

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. The two cases are distinguished by whether z is a right or left child of $p[z]$. 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 both z and $p[z]$ 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 $p[p[z]]$ exists, since we have argued that this node existed at the time that

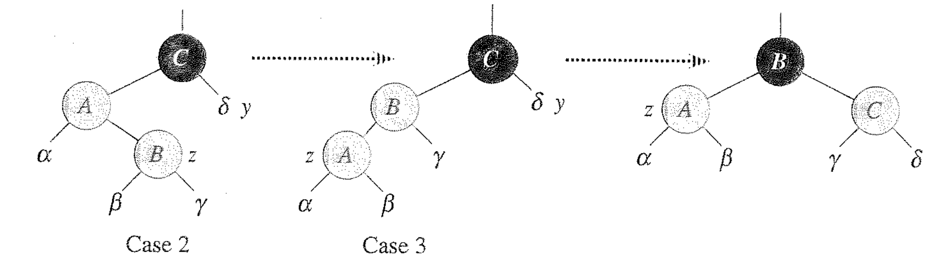

Figure 13.6 Cases 2 and 3 of the procedure RB-INSERT. As in case 1, property 4 is violated in either case 2 or case 3 because z and its parent $p[z]$ are both red. Each of the subtrees α, β, γ, and δ has a black root (α, β, and γ from property 4, and δ because otherwise we would be in case 1), and each has the same black-height. Case 2 is transformed into case 3 by a left rotation, which preserves property 5: all downward 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.

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 $p[p[z]]$ 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 body of the **while** loop is not executed another time, since $p[z]$ is now black.

Now we show that cases 2 and 3 maintain the loop invariant. (As we have just argued, $p[z]$ 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 $p[z]$, which is red. No further change to z or its color occurs in cases 2 and 3.

b. Case 3 makes $p[z]$ black, so that if $p[z]$ 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 is executed, 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. Interestingly, 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. Notice 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?

sol

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 property 5 is preserved by the indicated transformation.

13.3-4
Professor Teach is concerned that RB-INSERT-FIXUP might set $color[nil[T]]$ 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 $color[nil[T]]$ to RED.

sol

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.