CHAPTER 14: RED-BLACK TREES

Chapter 13 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 "balance" in order to guarantee that basic dynamic-set operations take *O*(lg *n*) time in the worst case.

14.1 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. Every leaf (NIL) is black.

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

4. Every simple path from a node to a descendant leaf contains the same number of black nodes.

An example of a red-black tree is shown in Figure 14.1.

We call the number of black nodes on any path from, but not including, a node *x* to a leaf the ***black-height*** of the node, denoted bh(*x*). By property 4, 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.

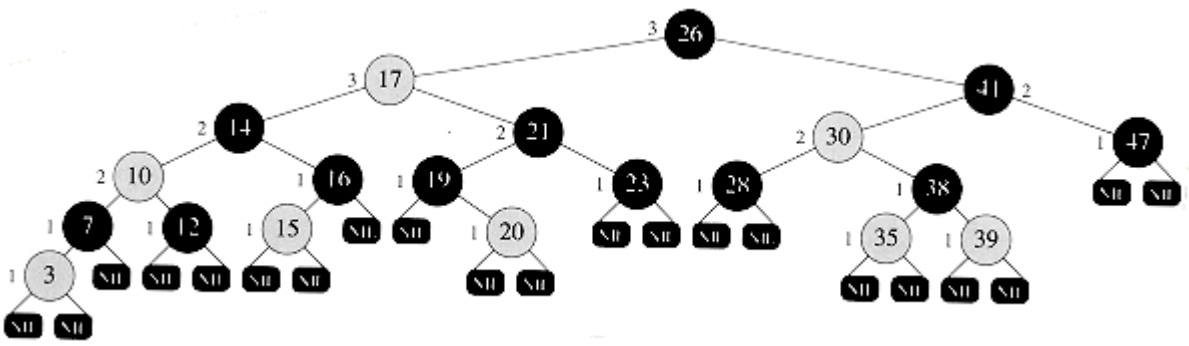

Figure 14.1 A red-black tree with black nodes darkened and red nodes shaded. Every node in a red-black tree is either red or black, every leaf (NIL) is black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. Each non-NIL node is marked with its black-height; NIL'S have black-height 0.

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

Lemma 14.1

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

Proof We first show that the subtree rooted at any node x contains at least $2^{bh(x)} - 1$ 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), and the subtree rooted at x indeed contains at least $2^{bh(x)} - 1 = 2^0 - 1 = 0$ 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 $bh(x)$ or $bh(x) - 1$, 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 $2^{bh(x)-1} - 1$ internal nodes. Thus, the subtree rooted at x contains at least $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ internal nodes, which proves the claim.

To complete the proof of the lemma, let h be the height of the tree. According to property 3, 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 $h/2$; 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 13) and any red-black tree on n nodes is a search tree with height $O(\lg n)$. Although the algorithms TREE-INSERT and TREE-DELETE from Chapter 13 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 14.3 and 14.4, however, that these two operations can indeed be supported in $O(\lg n)$ time.

Exercises

14.1-1

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.

14.1-2

Suppose that the root of a red-black tree is red. If we make it black, does the tree remain a red-black tree?

14.1-3

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.

14.1-4

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

14.1-5

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?

14.2 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 14.1.

To restore these properties, we must change the colors of some of the nodes in the tree and also change the pointer structure.

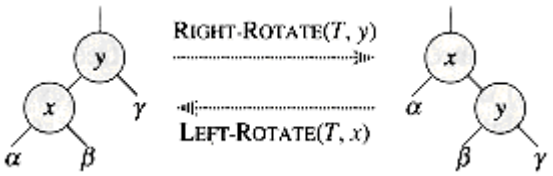

Figure 14.2 The rotation operations on a binary search tree. The operation RIGHT-ROTATE(T,x) transforms the configuration of the two nodes on the left into the configuration on the right by changing a constant number of pointers. The configuration on the right can be transformed into the configuration on the left by the inverse operation LEFT-ROTATE(T,y). The two nodes might occur anywhere in a binary search tree. The letters α, β, and γ represent arbitrary subtrees. A rotation operation preserves the inorder ordering of keys: the keys in α precede key[x], which precedes the keys in β, which precede key[y], which precedes the keys in γ.

We change the pointer structure through ***rotation***, which is a local operation in a search tree that preserves the inorder key ordering. Figure 14.2 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 non-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 *right*[*x*] ≠ NIL.

```
LEFT-ROTATE(T, x)
 1  y ← right[x]            ▷ Set y.
 2  right[x] ← left[y]      ▷ Turn y's left subtree into x's right subtree.
 3  if left[y] ≠ NIL
 4     then p[left[y]] ← x
 5  p[y] ← p[x]             ▷ Link x's parent to y.
 6  if p[x] = NIL
 7     then root[T] ← y
 8     else if x = left[p[x]]
 9             then left[p[x]] ← y
10             else right[p[x]] ← y
11  left[y] ← x             ▷ Put x on y's left.
12  p[x] ← y
```

Figure 14.3 shows how LEFT-ROTATE operates. The code for RIGHT-ROTATE is similar. Both LEFT-ROTATE and RIGHT-ROTATE run in *O*(l) time. Only pointers are changed by a rotation; all other fields in a node remain the same.

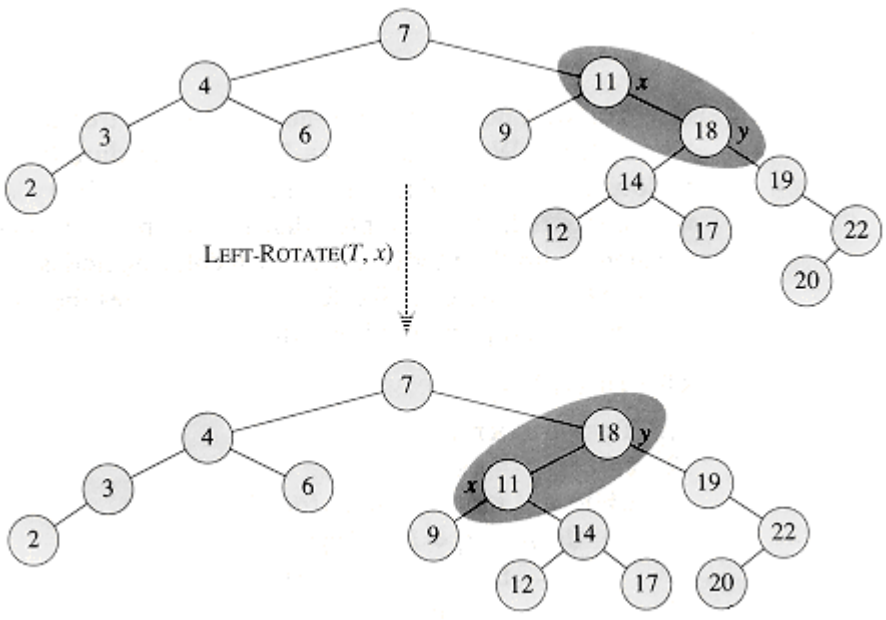

Figure 14.3 An example of how the procedure LEFT-ROTATE(T,x) modifies a binary search tree. The NIL leaves are omitted. Inorder tree walks of the input tree and the modified tree produce the same listing of key values.

Exercises

14.2-1

Draw the red-black tree that results after TREE-INSERT is called on the tree in Figure 14.1 with key 36. If the inserted node is colored red, is the resulting tree a red-black tree? What if it is colored black?

14.2-2

Write pseudocode for RIGHT-ROTATE.

14.2-3

Argue that rotation preserves the inorder key ordering of a binary tree.

14.2-4

Let *a, b*, and *c* be arbitrary nodes in subtrees α, β, and γ, respectively, in the left tree of Figure 14.2. How do the depths of *a, b*, and *c* change when a left rotation is performed on node *x* in the figure?

14.2-5

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

14.3 Insertion

Insertion of a node into an *n*-node red-black tree can be accomplished in $O(\lg n)$ time. We use the TREE-INSERT procedure (Section 13.3) to insert node *x* into the tree *T* as if it were an ordinary binary search tree, and then we color *x* red. To guarantee that the red-black properties are preserved, we then fix up the modified tree by recoloring nodes and performing rotations. Most of the code for RB-INSERT handles the various cases that can arise as we fix up the modified tree.

```
RB-INSERT(T, x)
 1  TREE-INSERT(T, x)
 2  color[x] ← RED
 3  while x ≠ root[T] and color[p[x]] = RED
 4        do if p[x] = left[p[p[x]]]
 5              then y ← right[p[p[x]]]
 6                   if color[y] = RED
 7                     then color[p[x]] ← BLACK                ▷ Case 1
 8                          color[y] ← BLACK                   ▷ Case 1
 9                          color[p[p[x]]] ← RED               ▷ Case 1
10                          x ← p[p[x]]                        ▷ Case 1
11                     else if x = right[p[x]]
12                            then x ← p[x]                   ▷ Case 2
13                                 LEFT-ROTATE(T, x)          ▷ Case 2
14                          color[p[x]] ← BLACK               ▷ Case 3
15                          color[p[p[x]] ← RED               ▷ Case 3
16                          RIGHT-ROTATE(T, p[p[x]])          ▷ Case 3
17              else (same as then clause
                      with "right" and "left" exchanged)
18  color[root[T]] ← BLACK
```

The code for RB-INSERT is less imposing than it looks. 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 lines 1-2 when the node *x* is inserted and colored red. Second, we shall examine the overall goal of the **while** loop in lines 3-17. Finally, we shall explore each of the three cases into which the **while** loop is broken and see how they accomplish the goal. Figure 14.4 shows how RB-INSERT operates on a sample red-black tree.

Which of the red-black properties can be violated after lines 1-2? Property 1 certainly continues to hold, as does property 2, since the newly inserted red node has NIL'S for children. Property 4, which says that the number of blacks is the same on every path from a given node, is satisfied as well, because node *x* replaces a (black) NIL, and node *x* is red with NIL children. Thus, the only property that might be violated is property 3 which says that a red node cannot have a red child. Specifically, property 3 is violated if *x*'s parent is red, since *x* is itself colored red in line 2. Figure 14.4(a) shows such a violation after the node *x* has been inserted.

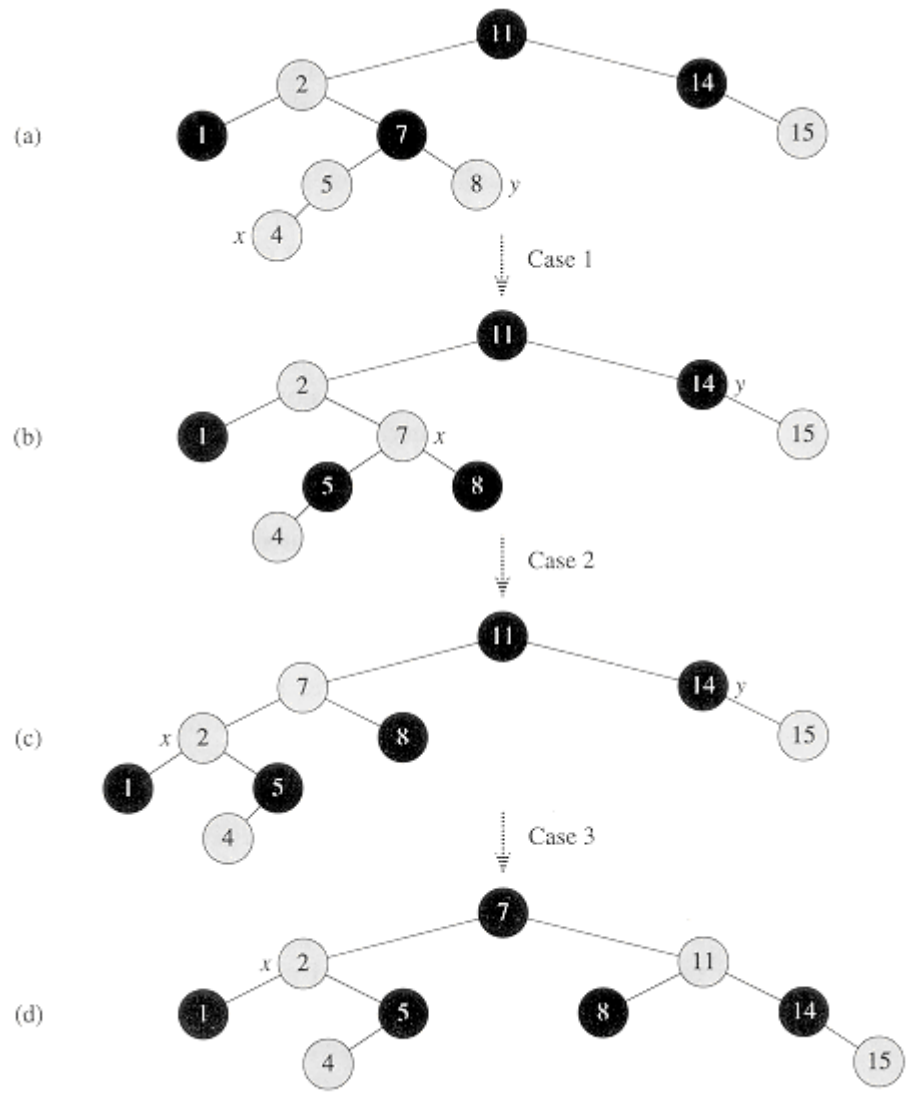

Figure 14.4 The operation of RB-INSERT. (a) A node x after insertion. Since x and its parent p[x] are both red, a violation of property 3 occurs. Since x's uncle y is red, case 1 in the code can be applied. Nodes are recolored and the pointer x is moved up the tree, resulting in the tree shown in (b). Once again, x and its parent are both red, but x's uncle y is black. Since x is the right child of p[x], case 2 can be applied. A left rotation is performed, and the tree that results is shown in (c). Now x 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.

The goal of the **while** loop in lines 3-17 is to move the one violation of property 3 up the tree while maintaining property 4 as an invariant. At the beginning of each iteration of the loop, *x* points to a red node with a red parent--the only violation in the tree. There are two possible outcomes of each iteration of the loop: the pointer *x* moves up the tree, or some rotations are performed and the loop terminates.

There are actually six cases to consider in the **while** loop, but three of them are symmetric to the other three, depending on whether *x*'s parent *p*[*x*] is a left child or a right child of *x*'s grandparent *p*[*p*[*x*]], which is determined

in line 4. We have given the code only for the situation in which $p[x]$ is a left child. We have made the important assumption that the root of the tree is black--a property we guarantee in line 18 each time we terminate--so that $p[x]$ is not the root and $p[p[x]]$ exists.

Case 1 is distinguished from cases 2 and 3 by the color of x's parent's sibling, or "uncle." Line 5 makes y point to x's uncle *right* $[p[p[x]]]$, and a test is made in line 6. If y is red, then case 1 is executed. Otherwise, control passes to cases 2 and 3. In all three cases, x's grandparent $p[p[x]]$ is black, since its parent $p[x]$ is red, and property 3 is violated only between x and $p[x]$.

The situation for case 1 (lines 7-10) is shown in Figure 14.5. Case 1 is executed when both $p[x]$ and y are red. Since $p[p[x]]$ is black, we can color both $p[x]$ and y black, thereby fixing the problem of x and $p[x]$ both being red, and color $p[p[x]]$ red, thereby maintaining property 4. The only problem that might arise is that $p[p[x]]$ might have a red parent; hence, we must repeat the **while** loop with $p[p[x]]$ as the new node x.

In cases 2 and 3, the color of x's uncle y is black. The two cases are distinguished by whether x is a right or left child of $p[x]$. Lines 12-13 constitute case 2, which is shown in Figure 14.6 together with case 3. In case 2, node x is a right child of its parent. We immediately use a left rotation to transform the situation into case 3 (lines 14-16), in which node x is a left child. Because both x and $p[x]$ are red, the rotation affects neither the black-height of nodes nor property 4. Whether we enter case 3 directly or through case 2, x's uncle y is black, since otherwise we would have executed case 1. We execute some color changes and a right rotation, which preserve property 4, 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 $p[x]$ is now black.

What is the running time of RB-INSERT? Since the height of a red-black tree on n nodes is $O(\lg n)$, the call to TREE-INSERT takes $O(\lg n)$ time. The **while** loop only repeats if case 1 is executed, and then the pointer x moves 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.

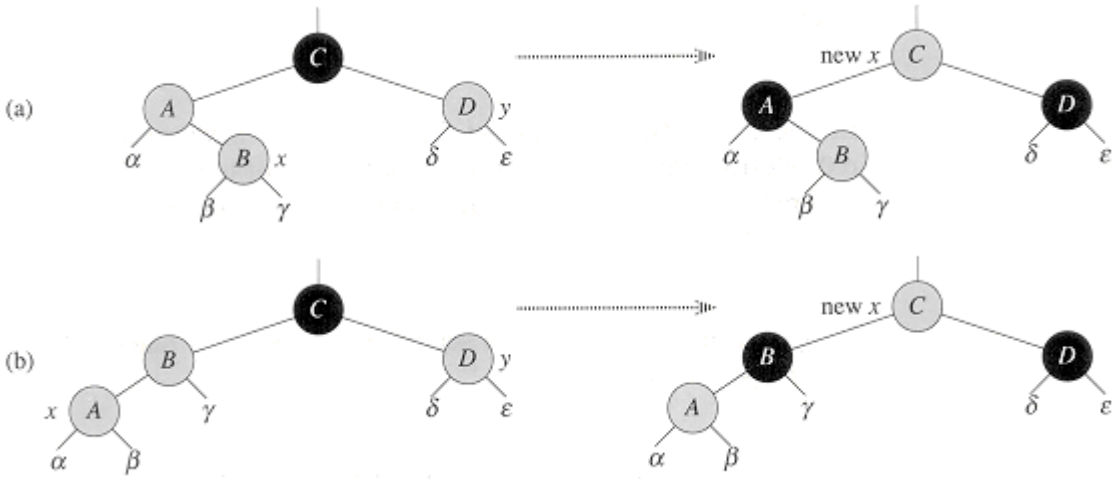

Figure 14.5 Case 1 of the procedure RB-INSERT. Property 3 is violated, since x and its parent p[x] are both red. The same action is taken whether (a) x is a right child or (b) x is a left child. Each of the subtrees α, β, γ, δ and ε has a black root, and each has the same black-height. The code for case 1 changes the colors of some nodes, preserving property 4: all downward paths from a node to a leaf have the same number of blacks. The while loop continues with node x's grandparent p[p[x]] as the new x. Any violation of property 3 can now occur only between the new x, which is red, and its parent, if it is red as well.

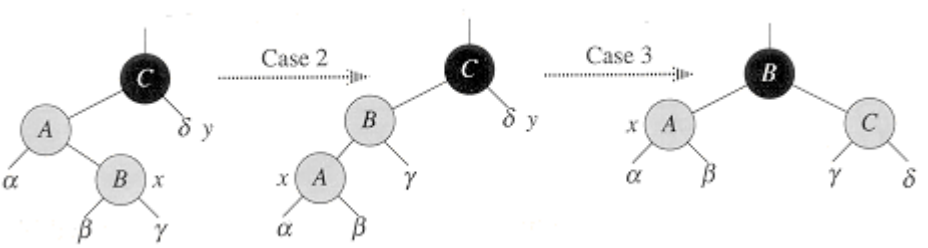

Figure 14.6 Cases 2 and 3 of the procedure RB-INSERT. As in case 1, property 3 is violated in either case 2 or case 3 because x and its parent p[x] are both red. Each of the subtrees α, β, γ, and δ has a black root, and each has the same black-height. Case 2 is transformed into case 3 by a left rotation, which preserves property 4: 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 4. The while loop then terminates, because property 3 is satisfied: there are no longer two red nodes in a row.

Exercises

14.3-1

In line 2 of RB-INSERT, we set the color of the newly inserted node *x* to red. Notice that if we had chosen to set *x*'s color to black, then property 3 of a red-black tree would not be violated. Why didn't we choose to set *x*'s color to black?

14.3-2

In line 18 of RB-INSERT, we set the root's color to black. What is the advantage of doing so?

14.3-3

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.

14.3-4

Suppose that the black-height of each of the subtrees α, β, γ, δ, ε in Figures 14.5 and 14.6 is *k*. Label each node in each figure with its black-height to verify that property 4 is preserved by the indicated transformation.

14.3-5

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.

14.3-6

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

14.4 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 only slightly more complicated than inserting a node.

In order to simplify boundary conditions in the code, we use a 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. In the red-black tree, all pointers to NIL are replaced by pointers to the sentinel *nil*[*T*].

We use sentinels so that we can treat a NIL child of a node *x* as an ordinary node whose parent is *x.* We could add a distinct sentinel node for each NIL in the tree, so that the parent of each NIL is well defined, but that would waste space. Instead, we use the one sentinel *nil*[*T*] to represent all the NIL'S. When we wish to manipulate a child of a node *x,* however, we must be careful to set *p*[*nil*[*T*]] to *x* first.

The procedure RB-DELETE is a minor modification of the TREE-DELETE procedure (Section 13.3). After splicing out a node, it calls an auxiliary procedure RB-DELETE-FIXUP that changes colors and performs rotations to restore the red-black properties.

```
RB-DELETE (T, z)
```

```
1 if left[z] = nil[T] or right[z] = nil[T]
2      then y ← z
3      else y ← TREE-SUCCESSOR(z)
4 if left[y] ≠ nil[T]
5     then x ← left[y]
6     else x ← right[y]
7 p[x] ← p[y]
8 if p[y] = nil[T]
9    then root[T] ← x
10    else if y = left[p[y]]
11            then left[p[y]] ← x
12            else right[p[y]] ← x
13 if y ≠ z
14     then key[z] ← key[y]
15          ▷ If y has other fields, copy them, too.
16 if color[y] = BLACK
17     then RB-DELETE-FIXUP (T,x)
18 return y
```

There are three differences between the procedures TREE-DELETE and RB-DELETE. First, all references to NIL in TREE-DELETE have been replaced by references to the sentinel *nil*[*T*] in RB-DELETE. Second, the test for whether *x* is NIL in line 7 of TREE-DELETE has been removed, and the assignment *p*[*x*] ← *p*[*y*] is performed unconditionally in line 7 of RB-DELETE. Thus, if *x* is the sentinel *nil*[*T*], its parent pointer points to the parent of the spliced-out node *y*. Third, a call to RB-DELETE-FIXUP is made in lines 16-17 if *y* is black. If *y* is red, the red-black properties still hold when *y* is spliced out, since no black-heights in the tree have changed and no red nodes have been made adjacent. The node *x* passed to RB-DELETE-FIXUP is the node that was *y*'s sole child before *y* was spliced out if *y* had a non-NIL child, or the sentinel *nil*[*T*] if *y* had no children. In the latter case, the unconditional assignment in line 7 guarantees that *x*'s parent is now the node that was previously *y*'s parent, whether *x* is a key-bearing internal node or the sentinel *nil*[*T*].

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

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

If the spliced-out node *y* in RB-DELETE is black, its removal causes any path that previously contained node *y* to have one fewer black node. Thus, property 4 is now violated by any ancestor of *y* in the tree. We can correct this problem by thinking of node *x* as having an "extra" black. That is, if we add 1 to the count of black nodes on any path that contains *x*, then under this interpretation, property 4 holds. When we splice out the black node *y*, we "push" its blackness onto its child. The only problem is that now node *x* may be "doubly black," thereby violating property 1.

The procedure RB-DELETE-FIXUP attempts to restore 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 node, in which case we color the node black in line 23, (2) *x* points to the root, in which case the extra black can be simply "removed," or (3) suitable rotations and recolorings can be performed.

Within the **while** loop, *x* always points to a nonroot black node that has the extra black. We determine in line 2 whether *x* is a left child or a right child of its parent *p*[*x*]. (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 *nil*[*T*]; otherwise, the number of blacks on the path from *p*[*x*] to the NIL leaf *w* would be smaller than the number on the path from *p*[*x*] to *x*.

The four cases in the code are illustrated in Figure 14.7. 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 4. The key idea is that in each case the number of black nodes from (and including) the root of the subtree shown to each of the subtrees $\alpha, \beta, \ldots, \zeta$ is preserved by the transformation. For example, in Figure 14.7(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. (Remember, the pointer *x* adds an extra black.) Similarly, the number of black nodes from the root to any of $\gamma, \delta, \varepsilon$, *and* ζ is 2, both before and after the transformation. In Figure 14.7(b), the counting must involve the color *c*, 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 + count(*c*), both before and after the transformation. The other cases can be verified similarly (Exercise 14.4-5).

Case 1 (lines 5-8 of RB-DELETE-FIXUP and Figure 14.7(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 *p*[*x*] and then perform a left-rotation on *p*[*x*] without violating any of the red-black properties. The new sibling of *x*, one of *w*'s children, 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. In case 2 (lines 10-11 of RB-DELETE-FIXUP and Figure 14.7(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, and add an extra black to *p*[*x*]. We then repeat the **while** loop with *p*[*x*] as the new node *x*. Observe that if we enter case 2 through case 1, the color *c* of the new node *x* is red, since the original *p*[*x*] was red, and thus the loop terminates when it tests the loop condition.

Case 3 (lines 13-16 and Figure 14.7(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 *left*[*w*] 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 (lines 17-21 and Figure 14.7(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 *p*[*x*], we can remove the extra black on *x* 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.

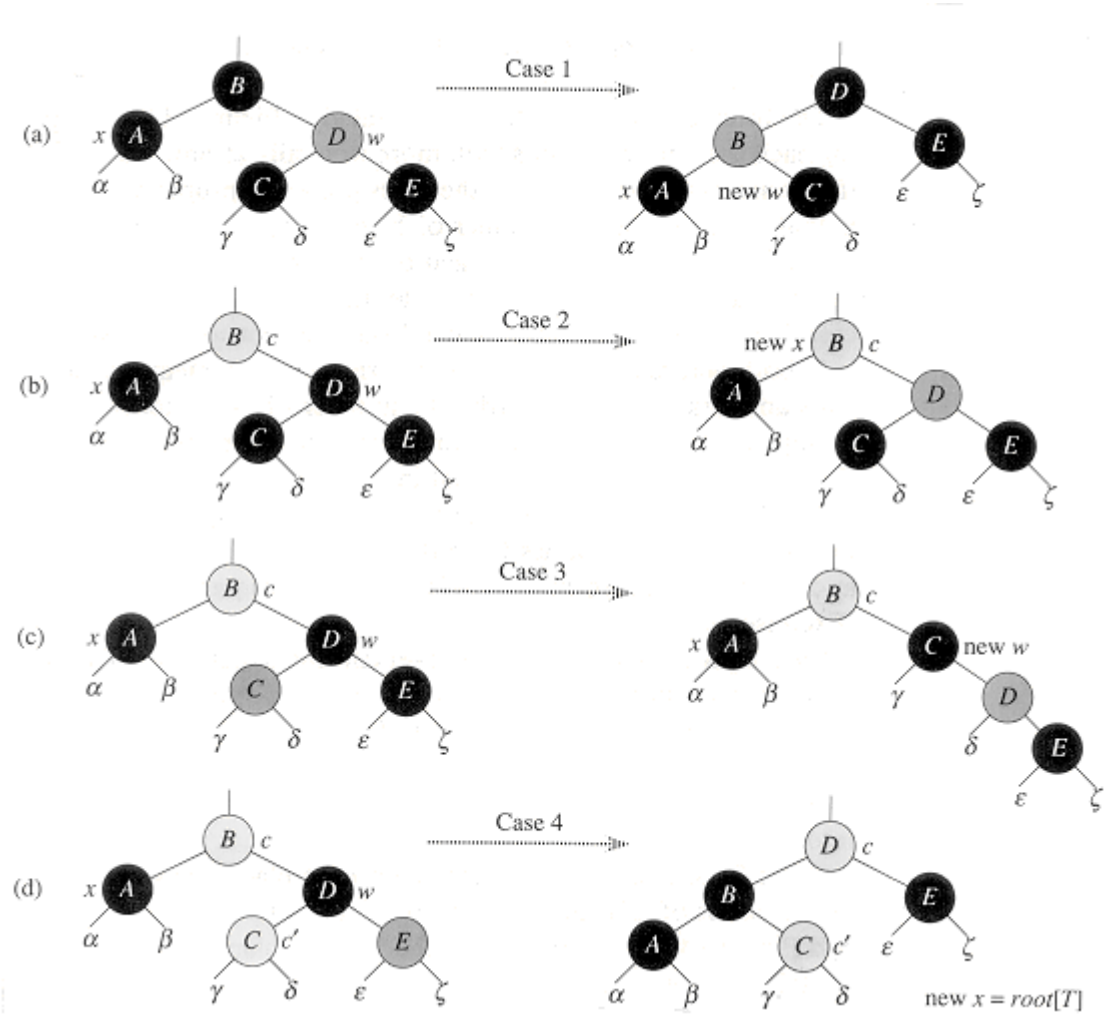

Figure 14.7 The cases in the while loop of the procedure RB-DELETE. Darkened nodes are black, heavily shaded nodes are red, and lightly shaded nodes, which may be either red or black, are represented by c and c'. The letters α, β, . . ., ζ represent arbitrary subtrees. In each case, the configuration on the left is transformed into the configuration on the right by changing some colors and/or performing a rotation. A node pointed to by x has an extra black. The only case that causes the loop to repeat is case 2. (a) Case I 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 is moved 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, since the color c 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) In case 4, the extra black represented by x can be removed by changing some colors and performing a left rotation (without violating the red-black properties), and the loop terminates.

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, cases 1, 3, and 4 each terminate 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 and no rotations are performed. 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

14.4-1

Argue that the root of the red-black tree is always black after RB-DELETE executes.

14.4-2

In Exercise 14.3-3, 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.

14.4-3

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

14.4-4

Simplify the code for LEFT-ROTATE by using a sentinel for NIL and another sentinel to hold the pointer to the root.

14.4-5

In each of the cases of Figure 14.7, give the count of black nodes from the root of the subtree shown to 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 c or c', use the notation count(c) or count(c') symbolically in your count.

14.4-6

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

Problems

14-1 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. Such a set is called ***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 14.8(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 14.8(b).

a. For a general persistent binary search tree, identify the nodes that need to be changed 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. Assume that each tree node has the fields *key*, *left*, and *right* but no parent field. (See also Exercise 14.3-6.)

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 field 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 is $O(\lg n)$ per insertion or deletion.

14-2 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 $key[x_1] \le key[x] \le key[x_2]$. 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.

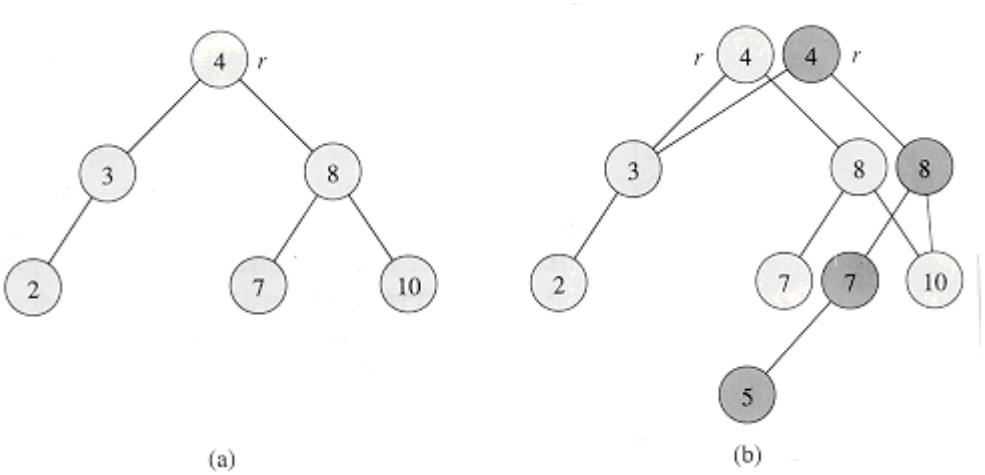

Figure 14.8 (a) A binary search tree with keys 2, 3, 4, 7, 8, 10. (b) The persistent binary search tree that results from the insertion of key 5. The most recent version of the set consists of the nodes reachable from the root r', and the previous version consists of the nodes reachable from r. Heavily shaded nodes are added when key 5 is inserted.

a. Given a red-black tree T, we store its black-height as the field $bh[T]$. Argue that this field can be maintained by RB-INSERT and RB-DELETE without requiring extra storage in 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 without loss of generality that $bh[T_1] \ge bh[T_2]$. 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 $bh[T_2]$.

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

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

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

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. Balance is maintained in AVL trees by rotations, but as many as $\Theta(\lg n)$ rotations may be required after an insertion to maintain balance in an n-node tree. Another class of search trees, called "2-3 trees," was introduced by J. E. Hopcroft (unpublished) in 1970. Balance is maintained in a 2-3 tree by manipulating the degrees of nodes in the tree. A generalization of 2-3 trees introduced by Bayer and McCreight [18], called B-trees, is the topic of Chapter 19.

Red-black trees were invented by Bayer [17] under the name "symmetric binary B-trees." Guibas and Sedgewick [93] studied their properties at length and introduced the red/black color convention.

Of the many other variations on balanced binary trees, perhaps the most intriguing are the "splay trees" introduced by Sleator and Tarjan [177], which are "self-adjusting." (A good description of splay trees is given by Tarjan [188].) 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 18) of each operation on an n-node tree is $O(\lg n)$.

Go to Chapter 15 Back to Table of Contents