17.4 Red-Black Trees

About 10 years after the introduction of AVL trees, Bayer introduced what are now known as red-black trees. (Bayer gave them the name "symmetric binary B-trees.") A red-black tree is balanced in the sense that no path from the root to a leaf has length more than twice the length of any other such path. However, this balance is

not part of the definition of a red-black tree, but rather is a consequence of the balance restrictions on the node coloring. The coloring balance restrictions for red-black trees are simpler to maintain than other balance restrictions such as AVL balance, with correspondingly simpler algorithms for the dictionary operations.

17.4.1 Definition and Depth of Red-Black Trees

The color balance properties of red-black trees are easier to state if we assume that augmented leaf nodes corresponding to unsuccessful searches are included as part of the tree. In particular, with augmented leaf nodes included, red-black trees are 2-trees. In what follows, for red-black trees we simply refer to the augmented leaf nodes as leaf nodes.

DEFINITION 17.4.1 A *red-black tree* is a binary search tree, augmented by (external) leaf nodes corresponding to unsuccessful search, where each node has been assigned either the color red or black subject to the following three properties.

i. Every leaf node is black.

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

iii. Any two paths from a given node v down to a leaf node contain the same number of black nodes.

Remark Including augmented leaf nodes in red-black trees makes the properties (i) and (iii) in Definition 17.4.1 have a powerful balancing effect. Indeed, if augmented leaf nodes were not included, then an all black, completely skewed binary search tree would be a red-black tree.

In Figure 17.6 (page 598) we show a red-black tree on 12 internal nodes. We have indicated the color outside each node.

The following proposition states that red-black trees have small depth, which makes them good search trees.

PROPOSITION 17.4.1 Let T be a red-black tree on n internal nodes. Then

$$\text{depth}(T) \leq 2\log_2(n+1). \tag{17.4.1}$$ □

We establish Proposition 17.4.1 with the aid of Lemma 17.4.2. To state Lemma 17.4.2, for any red-black tree T, we let $b(T)$ denote the number of black nodes, excluding the root, on any path from the root of T to a leaf. Property (iii) of Definition 17.4.1 shows that $b(T)$ is well defined.

LEMMA 17.4.2 Let T be a red-black tree. Then T has at least $2^{b(T)} - 1$ internal nodes.

Proof We prove Lemma 17.4.2 using induction on the number of internal nodes n in T.

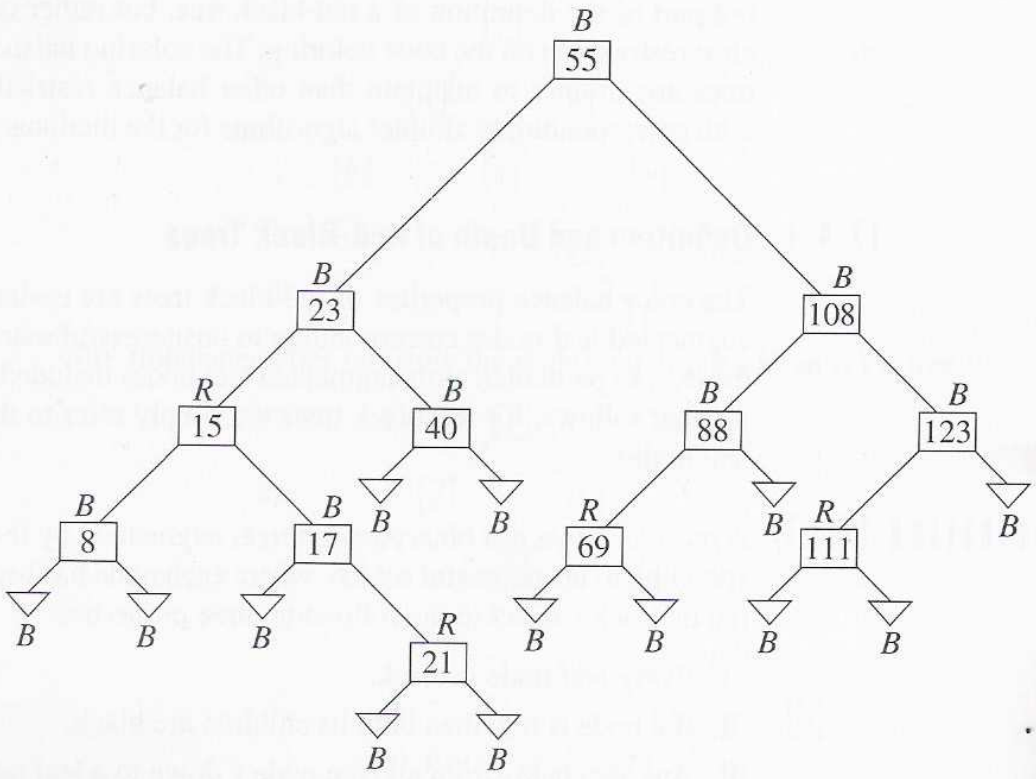


Figure 17.6 A red-black tree on 12 internal nodes

Basis step: $n = 1$. Then $b(T) = 1$. Since T has $1 = 2^1 - 1$ internal nodes, the basis step is established.

Induction step: Assume that Lemma 17.4.2 has been verified for all red-black trees having fewer than $n > 1$ internal nodes. Note that the subtree rooted at any node v of a red-black tree is also a red-black tree. Hence, the left and right subtrees L and R, respectively, of the root of T are both red-black trees having fewer than n internal nodes. Since T is a 2-tree, both L and R are nonempty. Thus, by induction hypothesis, L and R both have at least $2^{b(R)} - 1$ internal nodes. Now the values of $b(L)$ and $b(R)$ are either $b(T)$ or $b(T) - 1$, depending on whether their roots are colored red or black, respectively. It follows that T has at least $2(2^{b(T)-1} - 1) + 1 = 2^{b(T)} - 1$ nodes, which establishes the induction step and proves Lemma 17.4.2. ■

Proof of Proposition 17.4.1 Conditions (i) and (ii) of Definition 17.4.1 imply that at least half of the nodes along any path from the root to a leaf, excluding the root, must be black. In other words, $b(T) \geq \text{depth}(T)/2$. Thus, using Lemma 17.4.2, we have

$$n \geq 2^{b(T)} - 1 \geq 2^{\text{depth}(T)/2} - 1. \tag{17.4.2}$$

Proposition 17.4.1 now follows directly from (17.4.2). ■

17.4.2 Implementation of Red-Black Trees

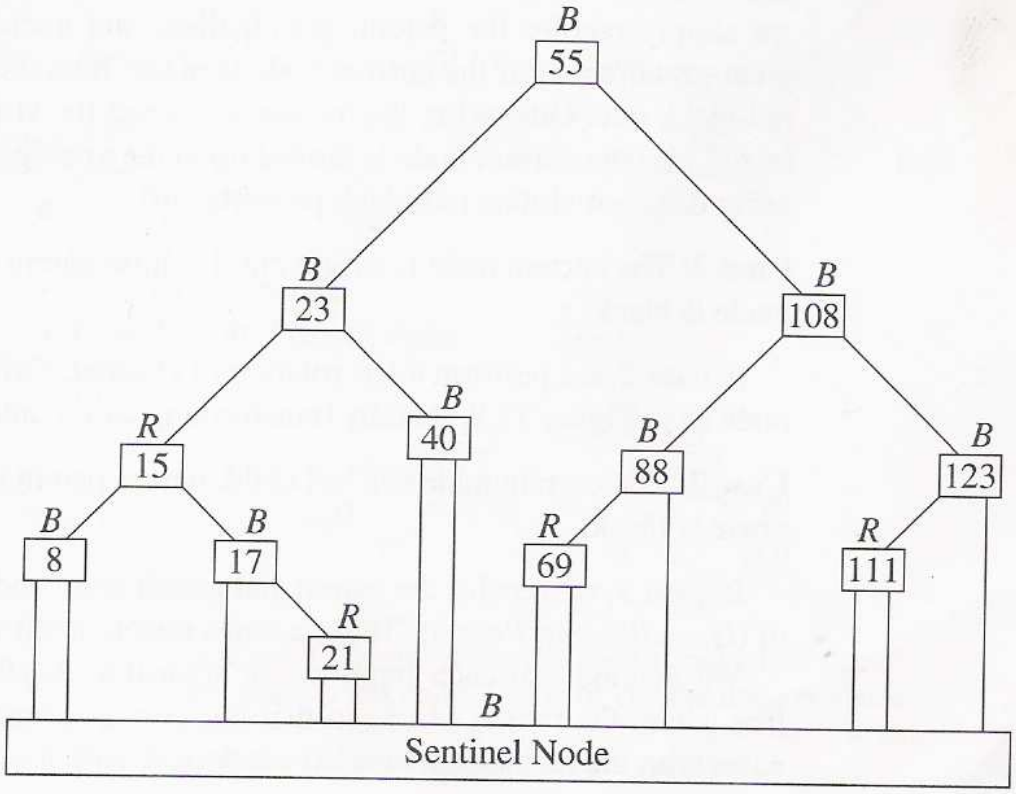
We assume that the nodes in a red-black tree have the following structure.

```
RedBlackTreeNode = record
    Info: InfoType
    Key: KeyType
    Color: Boolean {0 = red, 1 = black}
    LeftChild:    →RedBlackTreeNode
    RightChild:   →RedBlackTreeNode
    Parent:       →RedBlackTreeNode
end RedBlackTreeNode
```

The augmented leaf nodes of the red-black tree are assumed to have the same structure as the nodes in the original tree, except that their information and key fields do not contain meaningful values. However, the parent fields of the augmented leaf nodes would be appropriately defined. In actual practice, allocating memory locations for each augmented leaf node is simply too wasteful of space. Instead, a single augmented node is added, called a *sentinel node*. All child pointers in the original tree that pointed to **nil** augmented leaf nodes now hold the address of the sentinel node. Again, the sentinel node is assumed to have the same structure as the other nodes in the tree. In Figure 17.7 we show the same red-black tree as in Figure 17.6, but now a single sentinel node is used for the augmented leaf nodes.

Using a sentinel node to handle **nil** child pointers simplifies boundary conditions in algorithms that restore balance by suitably moving up the tree, sometimes starting from a **nil** leaf. For convenience, in some figures, we still represent **nil** augmented nodes as separate nodes as in Figure 17.6.

Figure 17.7 Red-black tree with sentinel node



A search algorithm for a red-black tree is the same as that for an ordinary binary search tree. We now consider the operations of inserting and deleting nodes.

17.4.3 Inserting a Node into a Red-Black Tree

When inserting a node into a red-black tree, we call the procedure *BinSrchTreeInsert* (modified to maintain parent pointers) and color the newly inserted node red. If the parent of the newly inserted node is black, we are done, since the tree is already a red-black tree. Otherwise, red-black property (ii) is violated. (It is easily seen that the other properties of a red-black tree remain satisfied.) At most two rotations and some recoloring restores property (ii), without violating the other red-black tree properties.

For any node v in a binary tree T, we define the *local neighborhood* of v to consist of v together with its parent, grandparent, and uncle. Each *correct-or-move-up* step in the eventual restoration of property (ii) involves recoloring and (possibly) rotating the nodes in the local neighborhood of the current node. As implied by the name, one of two things are accomplished by a correct-or-move-up step: Either the violation of property (ii) is corrected, or the violation of property (ii) is moved up two levels in the tree. During each correct-or-move-up step, no violations of the other two red-black properties are introduced. The current node is initially set to the newly inserted element.

For each correct-or-move-up step, there are six possible cases. Figure 17.8 illustrates three of these cases, which correspond to the parent of the current node being a red left child. The other three cases are mirror images of these cases.

Case 1: The current node has a red uncle, and its parent node is a red left child.

In case 1 it doesn't matter whether the current node is a left or a right child. For definiteness, in Figure 17.8 we have shown the current node as a left child. In case 1 we simply recolor the parent, grandparent, and uncle of the current node. If the great-grandparent of the current node is black, then the simple recoloring restores a red-black tree. Otherwise, the recoloring moves the violation of property (ii) up two levels, and the current node is moved up to the grandparent. In both cases, the recoloring does not violate red-black property (iii).

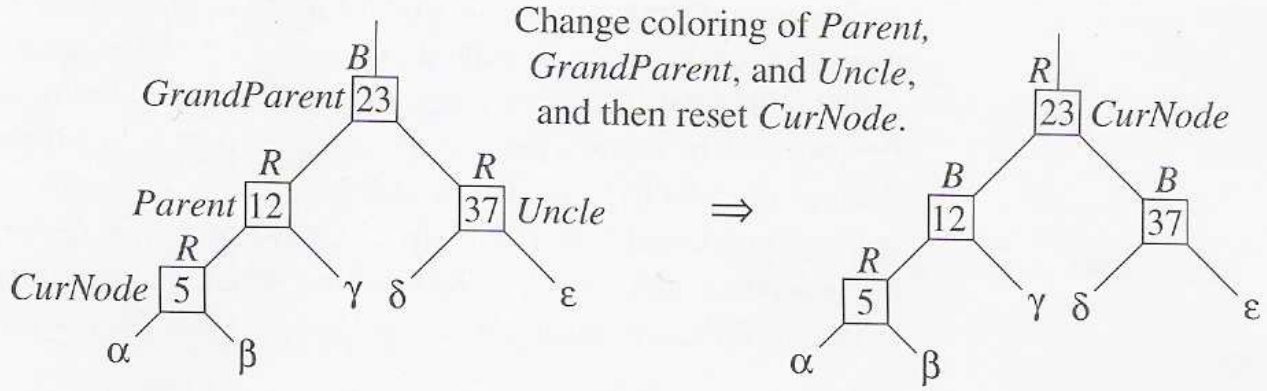
Case 2: The current node is a right child, whose parent is a red left child, and whose uncle is black.

In case 2, we perform a left rotation of (*Parent*, *CurNode*) and update the current node as in Figure 17.8, thereby transforming case 2 into case 3.

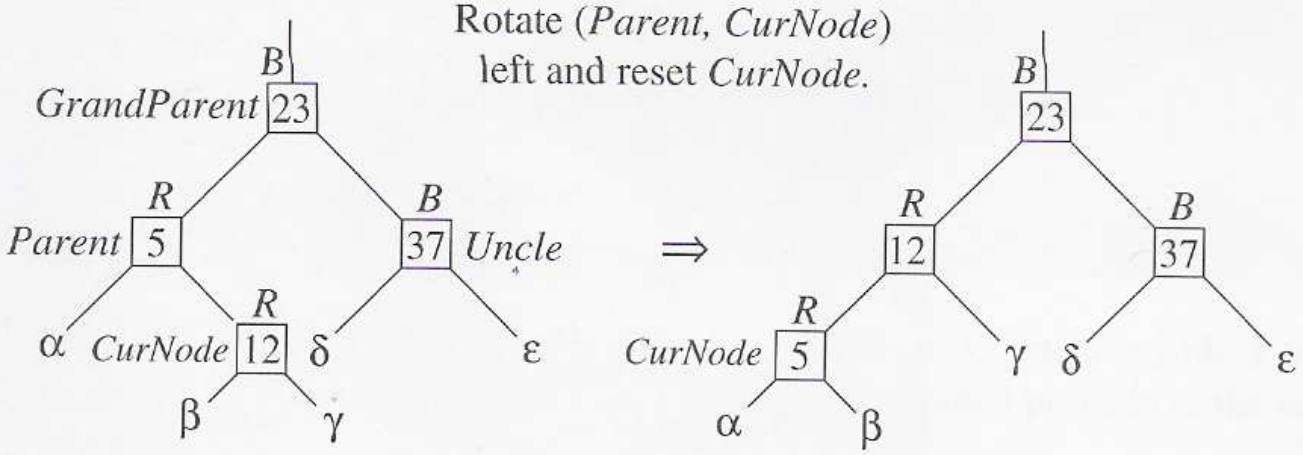
Case 3: The current node is a left child, whose parent is a red left child, and whose uncle is black.

In case 3, we recolor the parent and grandparent and then perform a right rotation of (*GrandParent*, *Parent*). These actions restore a red-black tree.

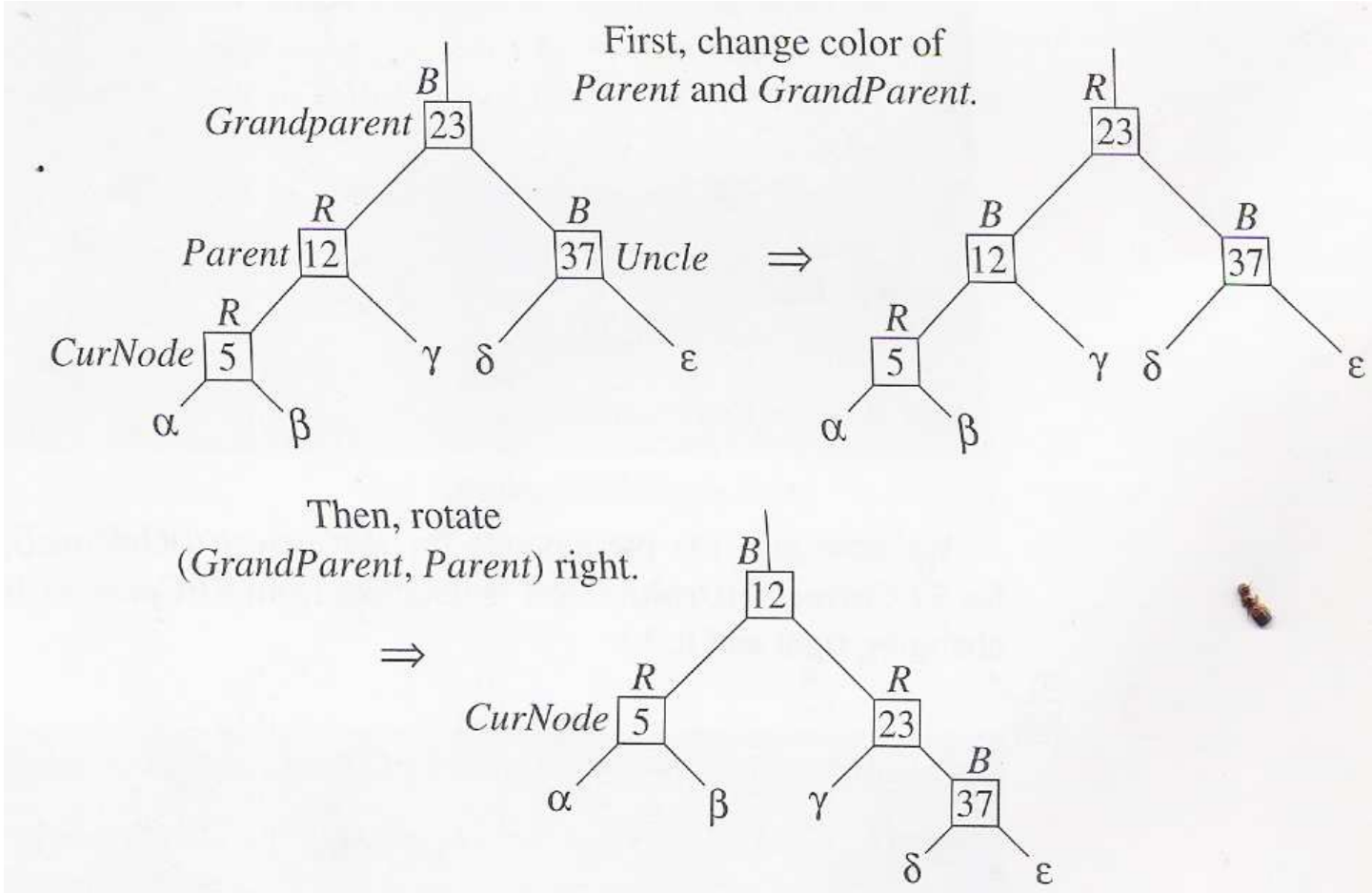
We assume that each input to the algorithm *RedBlackTreeInsert* is a red-black tree where the root is black, so that our correct-or-move-up process always terminates with the restoration of a red-black tree.



(a) Case 1: Parent of current node is a red left child. Uncle of current node is also red.



(b) Case 2: Parent of current node is red left child. Uncle of current node is black. Current node is right child.



(c) Case 3: Parent of current node is red left child. Current node is left child. Uncle of current node is black.

Figure 17.8 Actions leading to restoration of a red-black tree after insertion

The pseudocode for the procedure *RedBlackTreeInsert* is a straightforward implementation of the cases illustrated in Figure 17.8. We assume that *RedBlackTreeInsert* is called with a red-black tree where the root is black. During the course of the algorithm, the root may be colored red, but the last statement before the end of *RedBlackTreeInsert* restores its color to black. For readability, we use the terms *red* and *black* to correspond to the values 0 and 1, respectively, of the Boolean field *Color*. *RedBlackTreeInsert* calls the procedure *RBCorrect(ii)OrMoveUpL* when we encounter cases 1–3, and calls the procedure *RBCorrect(ii)OrMoveUpR* for the other three symmetric cases.

```
procedure RedBlackTreeInsert(Root,X,Node)

Input:  Root (→RedBlackTreeNode) {points at black root of a red-black tree T}
        X (KeyType)              {key to be inserted in T}
Output: Root (→RedBlackTreeNode) {root of the red-black tree T'
                                  obtained by inserting X}
        Node (→RedBlackTreeNode) {points to node where X inserted}

  call BinSrchTreeInsert(Root,X,Node) {ordinary search tree insertion,
                                       where Node now points to inserted node containing X}
  Node→Color := red
  CurNode := Node
  while CurNode ≠ Root .and. (CurNode→Parent)→Color = red do
    Parent := CurNode→Parent
    GrandParent := (CurNode→Parent)→Parent
    if GrandParent→LeftChild = Parent then
      call RBCorrect(ii)OrMoveUpL(CurNode,Parent,GrandParent)
    else
      call RBCorrect(ii)OrMoveUpR(CurNode,Parent,GrandParent)
    endif
  endwhile
  Root→Color := black

end RedBlackTreeInsert
```

We now give the pseudocode for *RBCorrect(ii)OrMoveUpL*. (The pseudocode for *RBCorrect(ii)OrMoveUpR* is obtained from *RBCorrect(ii)OrMoveUpL* by interchanging right and left.)

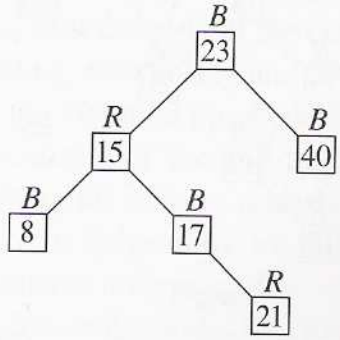
```
procedure RBCorrect(ii)OrMoveUpL(Root,CurNode,Parent,GrandParent)

Input:  CurNode (→RedBlackTreeNode) {points at node of a tree T
                                     whose parent violates property (ii)}
Output: CurNode (→RedBlackTreeNode) {root of the red-black tree T' arising
                                     from T by performing action according to Case 1, 2, or 3
                                     and then moving up CurNode}
```

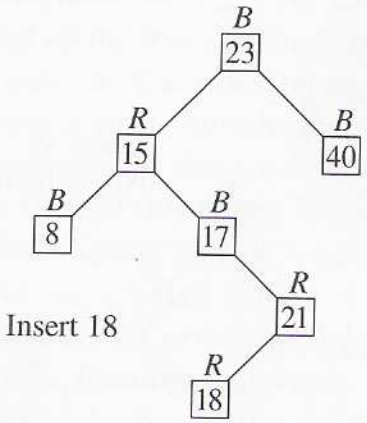
```
Uncle := GrandParent→RightChild
if Uncle→Color = red then                    {Case 1}
   Parent→Color := black
   GrandParent→Color := red
   Uncle→Color := black
   CurNode :→ GrandParent                    {move current node up two levels}
else
   if Parent→RightChild = CurNode            {Case 2}
      call RotateLeft(Root,Parent)
      CurNode := CurNode→LeftChild           {reset current node}
   endif
   Parent→Color := black                     {Case 3}
   GrandParent→Color := red
   call RotateRight(Root,GrandParent)
endif
end RBCorrect(ii)OrMoveUpL
```

We illustrate the insertion of the sequence of keys 18, 16, 55 into a red-black tree in Figure 17.9. For convenience, in Figure 17.9 we have omitted pointers to the sentinel leaf node.

Figure 17.9 Inserting the sequence of keys 18, 16, 55 into a red-black tree



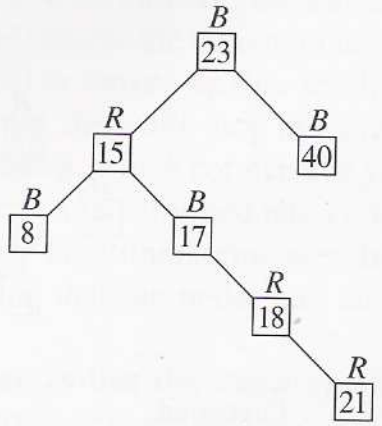
(a) Original red-black tree



Insert 18

Case 2′

⇒

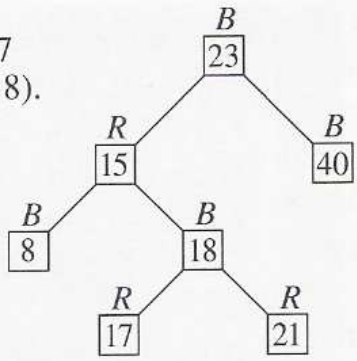


Case 3′

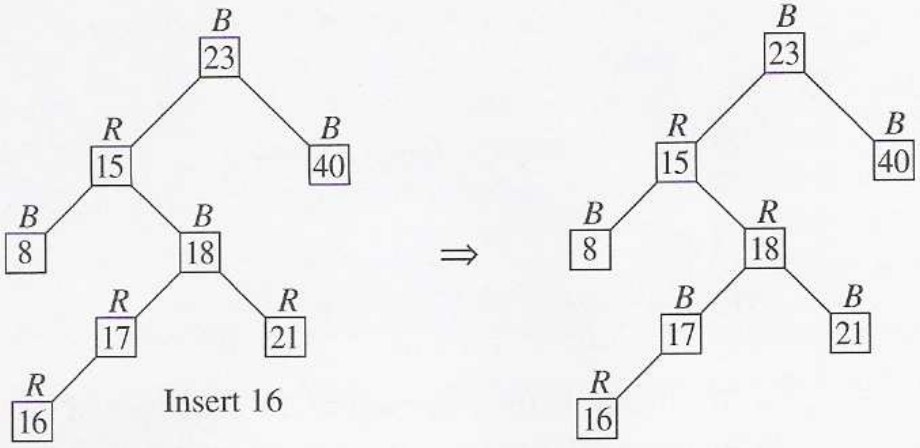
Continued

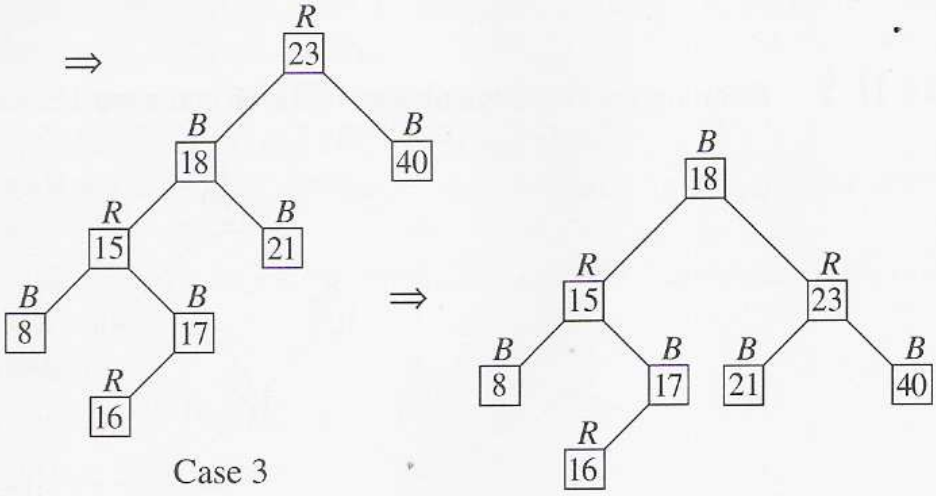
Recolor 18 and 17
and left rotate (17,18).

⇒

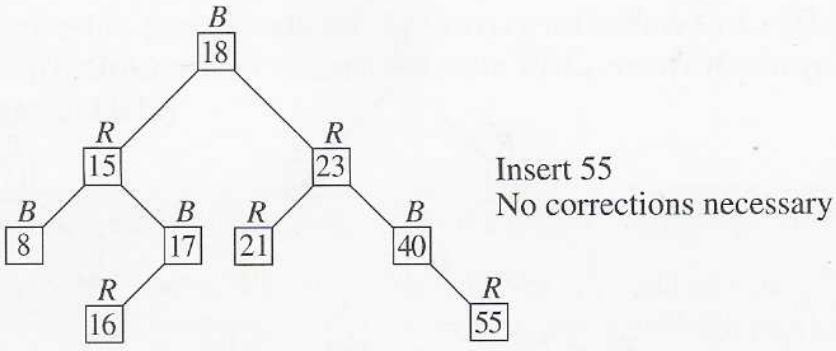


(b) Final red-black tree after inserting 18





(c) Final red-black tree after inserting 16



(d) Insert 55

Figure 17.9 Continued

17.4.4 Deleting a Node from a Red-Black Tree

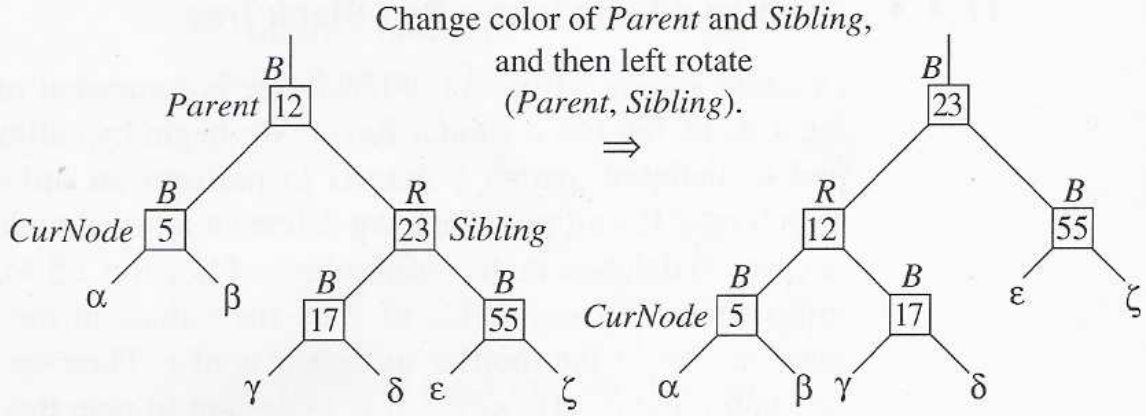
Deleting a node v from a red-black tree is somewhat more complicated than inserting a node, but has a similar flavor. We begin by calling *BinSrchTreeDelete* (modified to maintain parent pointers) to perform an ordinary deletion from a binary search tree. Recall that when we delete an internal node v with two non-nil children (a case (3) deletion in the terminology of Section 2.5.4), we replace the values in the information and key fields of v by the values in the information and key fields, respectively, of the inorder successor w of v. Then we delete w, which has at most one non-nil child. However, it is important to note that we do *not* replace the value of the color field of v by the value of the color field of w. Thus, for a case (3) deletion, when we talk about the color of the deleted node, we mean the color of w, not v.

The deletion of a node with at most one non-nil child is accomplished by splicing it out (see Figure 2.19). Now if the deleted node was red, clearly we still have a red-black tree and nothing else need be done. If the deleted node was black, and it has a red non-nil child, then clearly we restore a red-black tree by recoloring the non-nil child black. Hence, we are left with the problem of restoring a red-black tree when the deleted node and its children are all black. Property (iii) is then violated, since any path that starts at a given node y, goes through the parent of the deleted node, and then goes on to the "spliced-in child" *CurNode* and down to a descendant leaf, will have one too few black nodes than a path from y to a leaf that doesn't go through *CurNode*. Also, any two paths from a node to a leaf that do not go through *CurNode* have the same number of black nodes. Thus, we can consider the violation of property (iii) to be "rooted" at *CurNode*, the spliced-in child.

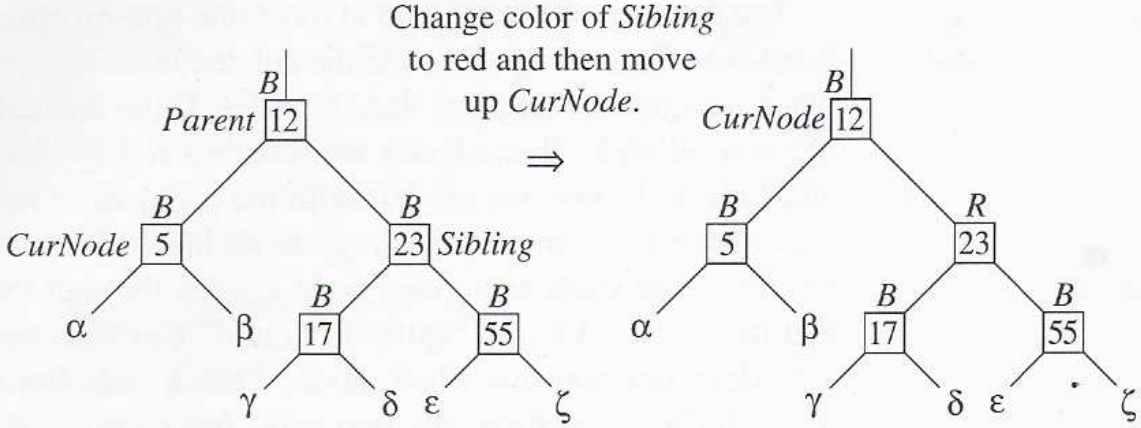
As before, we perform recolorations and rotations to restore a red-black tree, but our local neighborhood now consists of the current node, its parent, its sibling, and the two children of the sibling. Recoloring and rotation will either correct the violation of property (iii) or move the violation up one level in the tree (at the same time not introducing any other violations of the red-black properties). There are ten cases to consider, five cases where the current node is a left child, and five cases where it is a right child. In Figure 17.10 (page 606), we illustrate the five cases handled by *RBCorrect(iii)OrMoveUpL* (current node is a left child). We use the variable names *CurNode*, *Parent*, and *Sibling* to denote the nodes in a local neighborhood of the current node.

In case 1, *Sibling* is red, whereas in the other four cases, *Sibling* is black. Case 1 transforms into case 2b, case 3, or case 4. In case 2a, the violation is moved one level up the tree and the loop is repeated (unless the current node becomes the root). In case 2b, the violation of property (iii) is moved up one level, but a violation of property (ii) is introduced there. However, the while loop in *RedBlackTreeDelete* is terminated, so that *CurNode* is colored black, which corrects the violation of properties (ii) and (iii) in one fell swoop. Case 3 is transformed into case 4, where the violation of property (iii) is corrected and the algorithm terminates. In Figure 17.10, we show sample key values in the nodes so that the nodes can be traced during the action of *RBCorrect(iii)OrMoveUpL*.

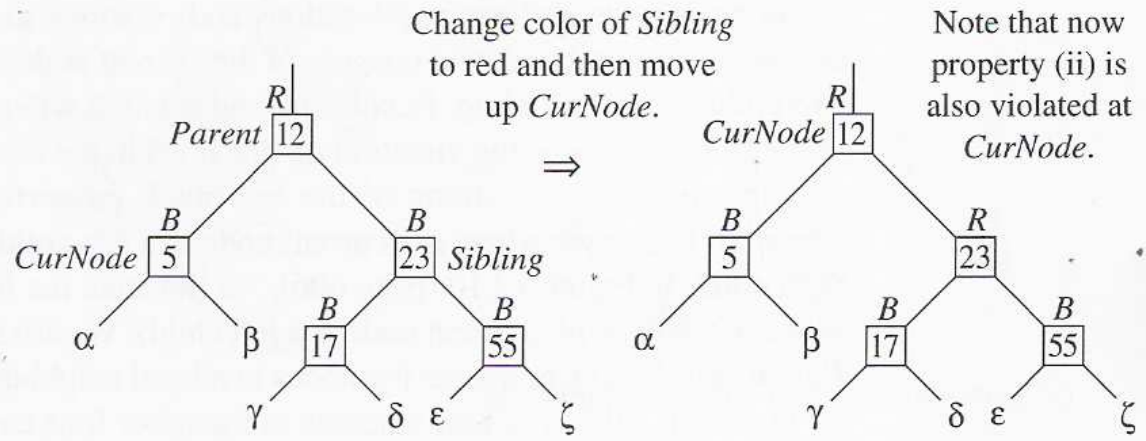
It is important to verify in all ten cases that the transformations introduce no other violations of the red-black properties (except as noted for case 2b, which is



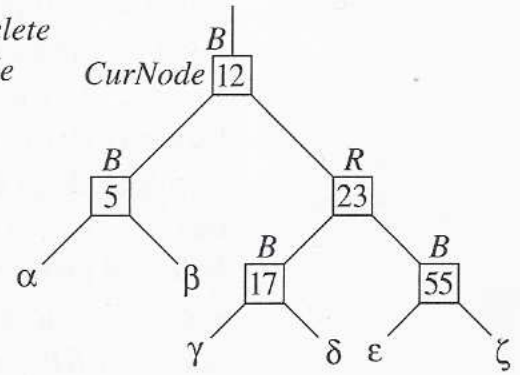
(a) Case 1: Transform to case 2b, 3, or 4



(b) Case 2a: Violation of property (iii) moves up one level



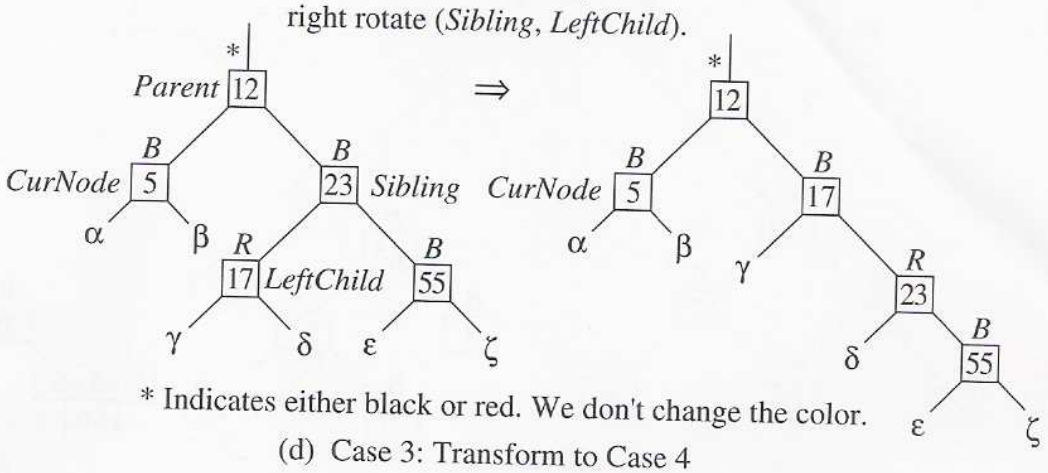
While loop in *RedBlackTreeDelete* then terminates, and *CurNode* is then colored black, thus restoring red-black tree.



(c) Case 2b

Figure 17.10 Restoring a red-black tree after deleting a node

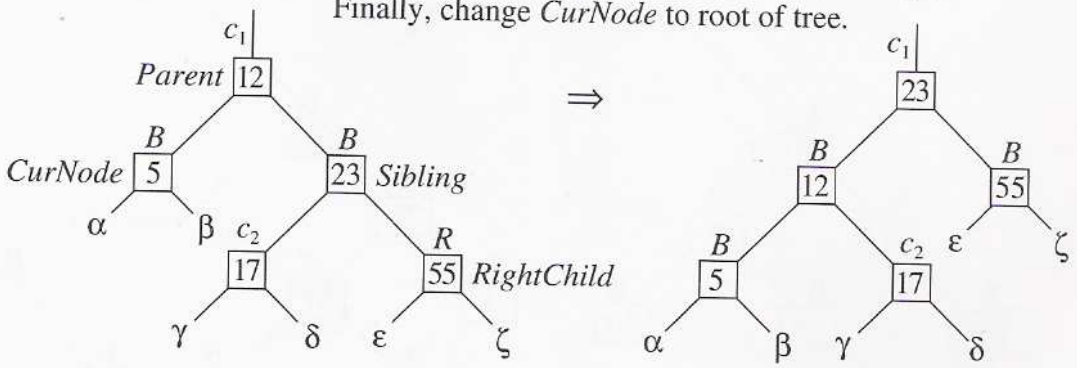
Change color of *Sibling* to red and its left child to black, and then right rotate (*Sibling*, *LeftChild*).



* Indicates either black or red. We don't change the color.

(d) Case 3: Transform to Case 4

Nodes colored c_1 and c_2 are either black or red. These colors remain unchanged. Change color of *Sibling*'s right child to black, color *Parent* black, color *Sibling* c_1, and then left rotate (*Parent*, *Sibling*). Finally, change *CurNode* to root of tree.

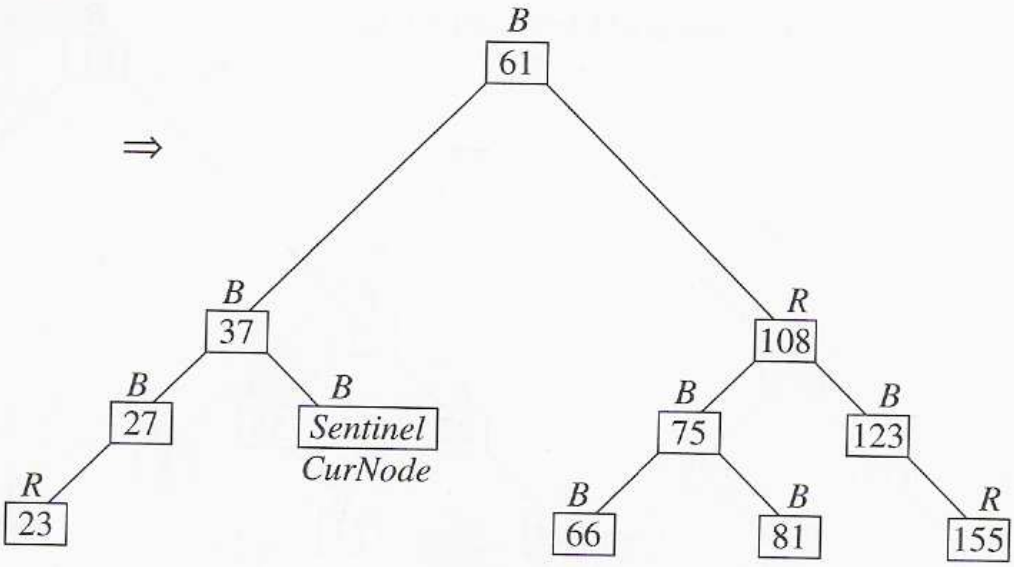


(e) Case 4

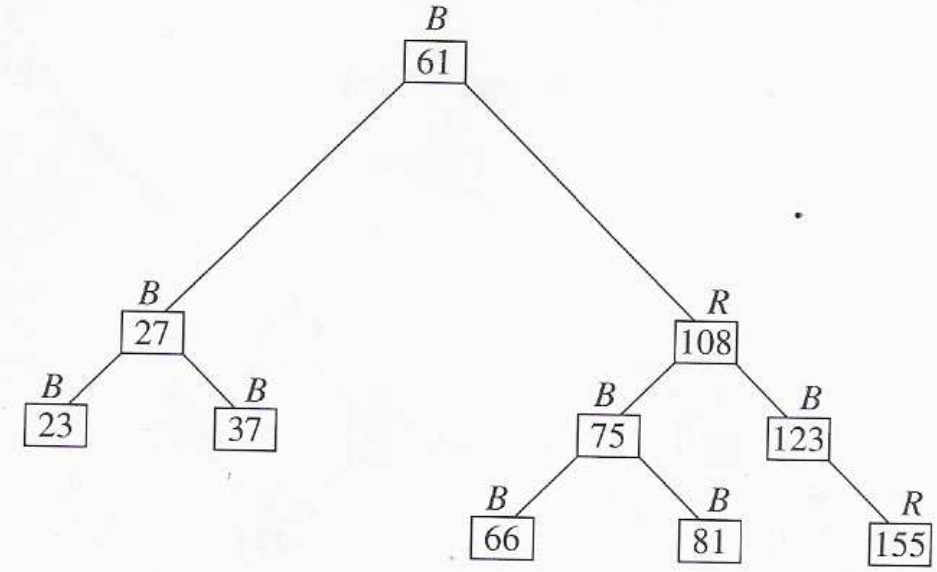
then corrected), that each case except case 2a results in restoring a red-black tree, and that case 2a moves the root of the violation of property (iii) one level up the tree. For example, in case 1, note from Figure 17.10a that any path starting a vertex v and going through either node 5, 17, or 55 on its way to a leaf node will encounter the same number of black nodes before and after the transformation. Moreover, no other paths to leaf nodes are affected by the transformation, so the violation of property (iii) remains rooted at node 5. The complete verifications of all ten cases are left as exercises.

In Figure 17.11 (page 608), we illustrate the deletion of the sequence of nodes 88, 8, 55, and 61 from a red-black tree. Again, for convenience we omit pointers to the sentinel leaf node *Sentinel*, except when the initial value of *CurNode* points at *Sentinel* as in Figures 17.11c and 17.11d.

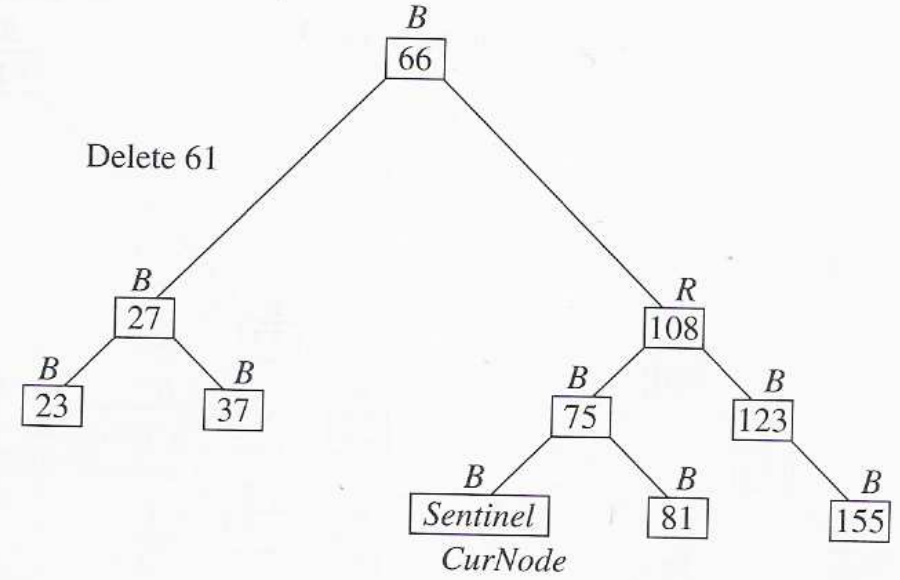
Note that we perform at most two rotations to correct the violation of property (iii) and restore a red-black tree. When restoring the red-black properties after



Case 4′

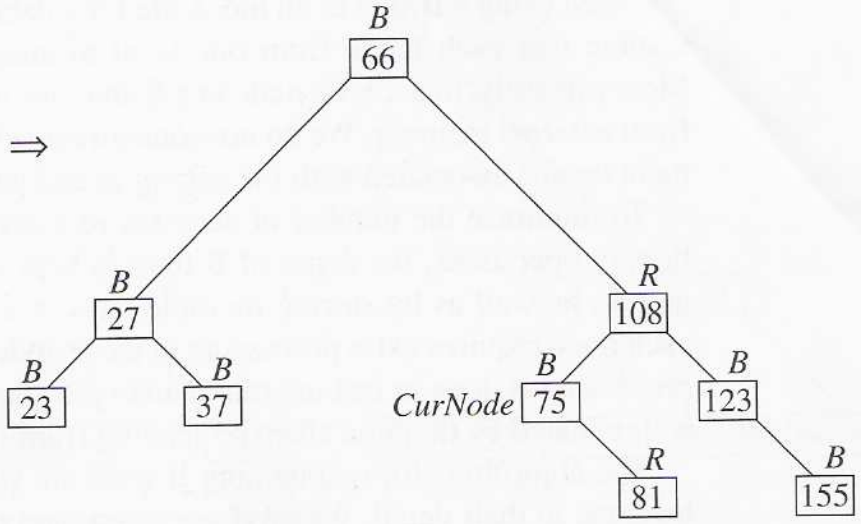


(e) Deleting 61



Case 2a: Replace 61 by its inorder successor 66, then delete original occurrence of 66

Figure 17.11 Continued



Case 2b

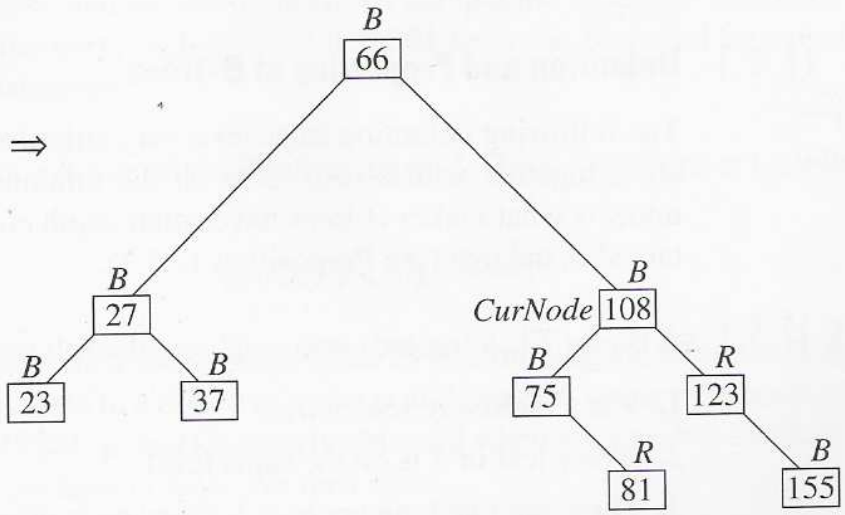


Figure 17.11 Continued

inserting or deleting node, each time we call the correct-or move-up algorithms we perform a constant number of pointer and other field reassignments. Thus, the complexity of both inserting and deleting an element in a red-black tree T having n internal nodes belongs to $O(\text{depth}(T)) = O(\log n)$. The pseudocode for deleting a node from a red-black tree is left as an exercise.