Red-Black tree

- Recall binary search tree
  - Key values in the left subtree <= the node value
  - Key values in the right subtree >= the node value
- Operations:
  - insertion, deletion
  - Search, maximum, minimum, successor, predecessor.
  - O(h), h is the height of the tree.

Red-black trees

- Definition: a binary tree, satisfying:
  1. Every node is red or black
  2. The root is black
  3. Every leaf is NIL and 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.
- Purpose: keep the tree balanced.
- Other balanced search tree:
  - AVL tree, 2-3-4 tree, Splay tree, Treap

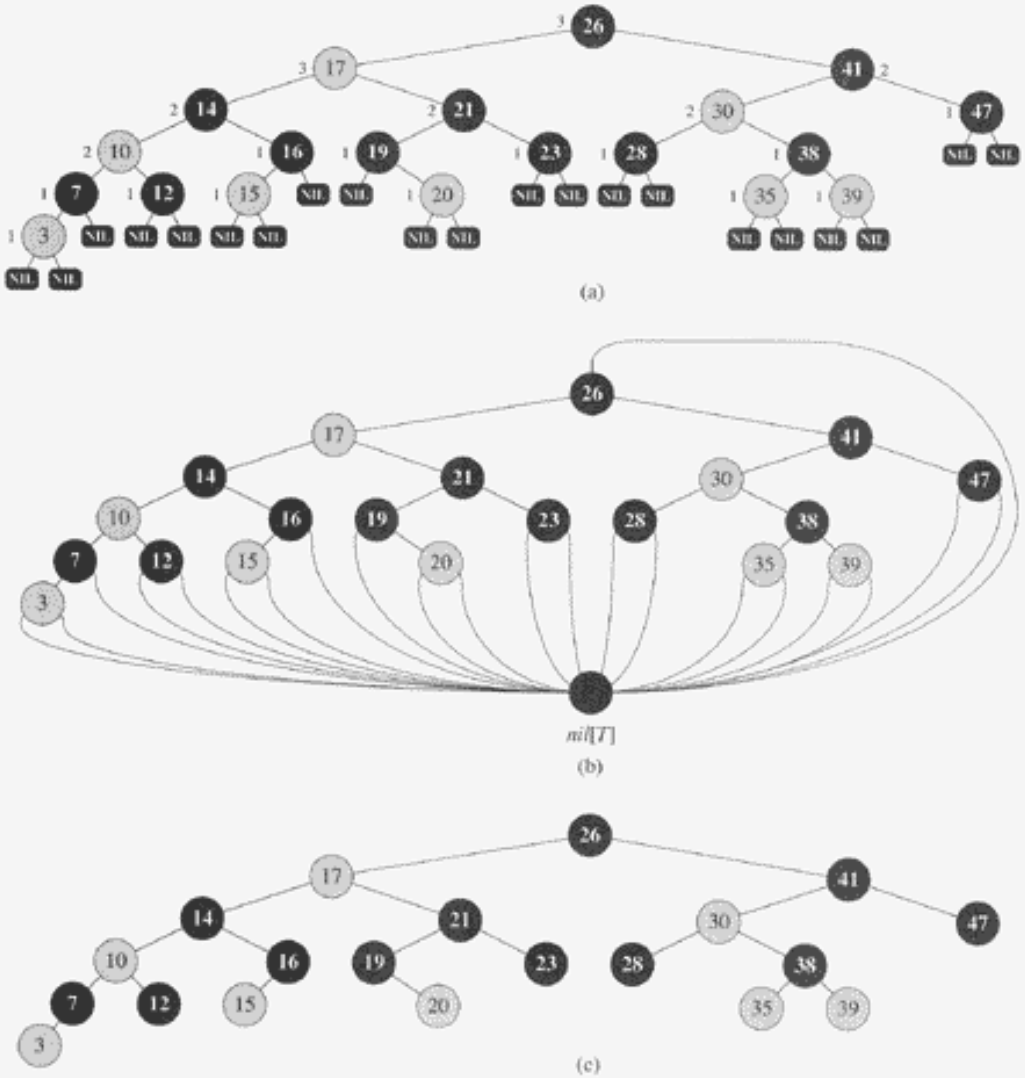


Figure 13.1 A red-black tree with black nodes darkened and red nodes shaded. Every node in a red-black tree is either red or black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. **(a)** Every leaf, shown as a 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.

Fields and property

- Left, right, ,parent, color, key
- bh(x), black-height of x, the number of black nodes on any path from x (excluding x) to a leaf.
- A red-black tree with n internal nodes has height at most 2log(n+1).
  - Note: internal nodes: all normal key-bearing nodes. External nodes: Nil nodes or the Nil Sentinel.
  - A subtree rooted at x contains at least $2^{bh(x)}$-1 internal nodes.
  - By property 4, bh(root)≥h/2.
  - $n \geq 2^{h/2}-1$

Some operations in log(n)

- Search, minimum, maximum, successor, predecessor.
- Let us discuss insert or delete.

Right rotation:
x=left[y]; left[y]=right[x];
If(right[x]!=nil) p[right[x]]=y;
p[x]=p[y]; if(p[y]==nil) root=x;
If(left[p[y]]=y) left[p[y]]=x;
else right[p[y]]=x;
right[x]=y; p[y]=x;

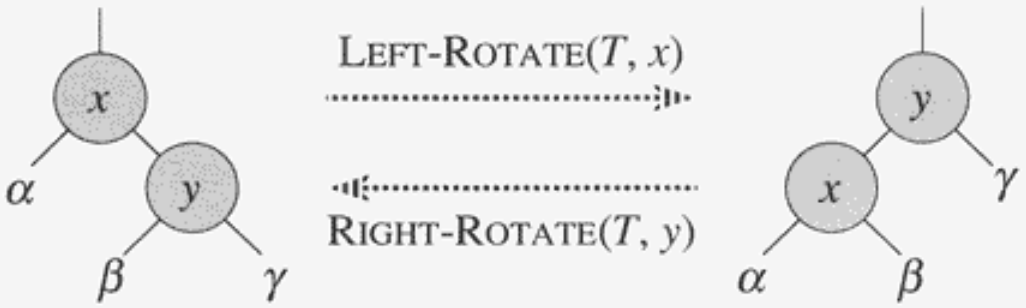


Figure 13.2 The rotation operations on a binary search tree. The operation LEFT-ROTATE(T, x) transforms the configuration of the two nodes on the left into the configuration on the right by changing a constant number of pointers. The configuration on the right can be transformed into the configuration 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 rotation:
y=right[x]; right[x] ←left[y]; If(left[y]!=nil) p[left[y]]=x; p[y]=p[x];
if(p[x]==nil) {root=y;} else if (left[p[x]]==x) left[p[x]]=y; else right[p[x]]=y;
left[y]=x; p[x]=y;

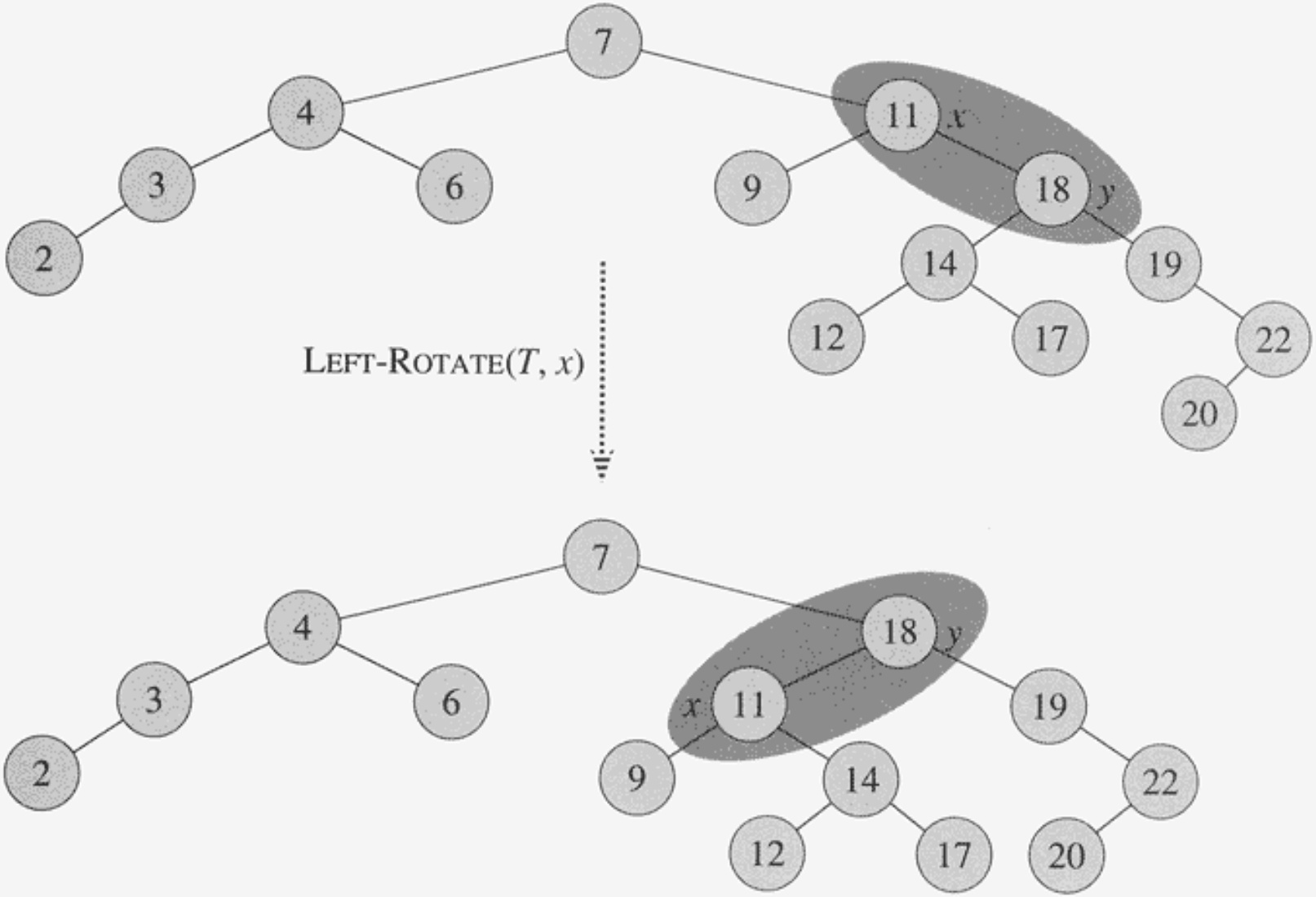


Figure 13.3 An example of how the procedure LEFT-ROTATE(T, x) modifies a binary search tree. Inorder tree walks of the input tree and the modified tree produce the same listing of key values.

```
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)
```

Properties violations

- Property 1 (each node black or red): hold
- Proper 3: (each leaf is black sentinel): hold.
- Property 5: same number of blacks: hold
- Property 2: (root is black), not, if z is root (and colored red).
- Property 4: (the child of a red node must be black), not, if z’s parent is red.

Case 1,2,3: p[z] is the left child of p[p[z]].
Correspondingly, there are 3 other cases,
In which p[z] is the right child of p[p[z]]

```
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
```

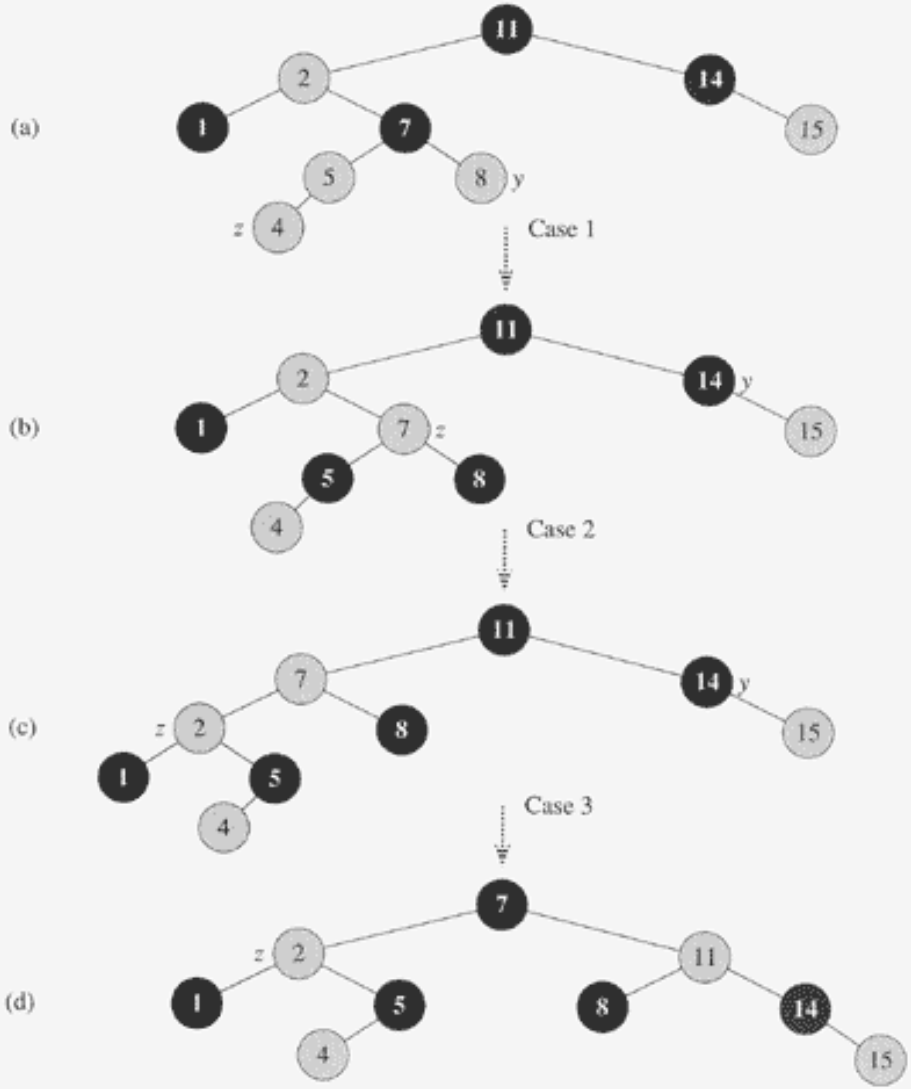
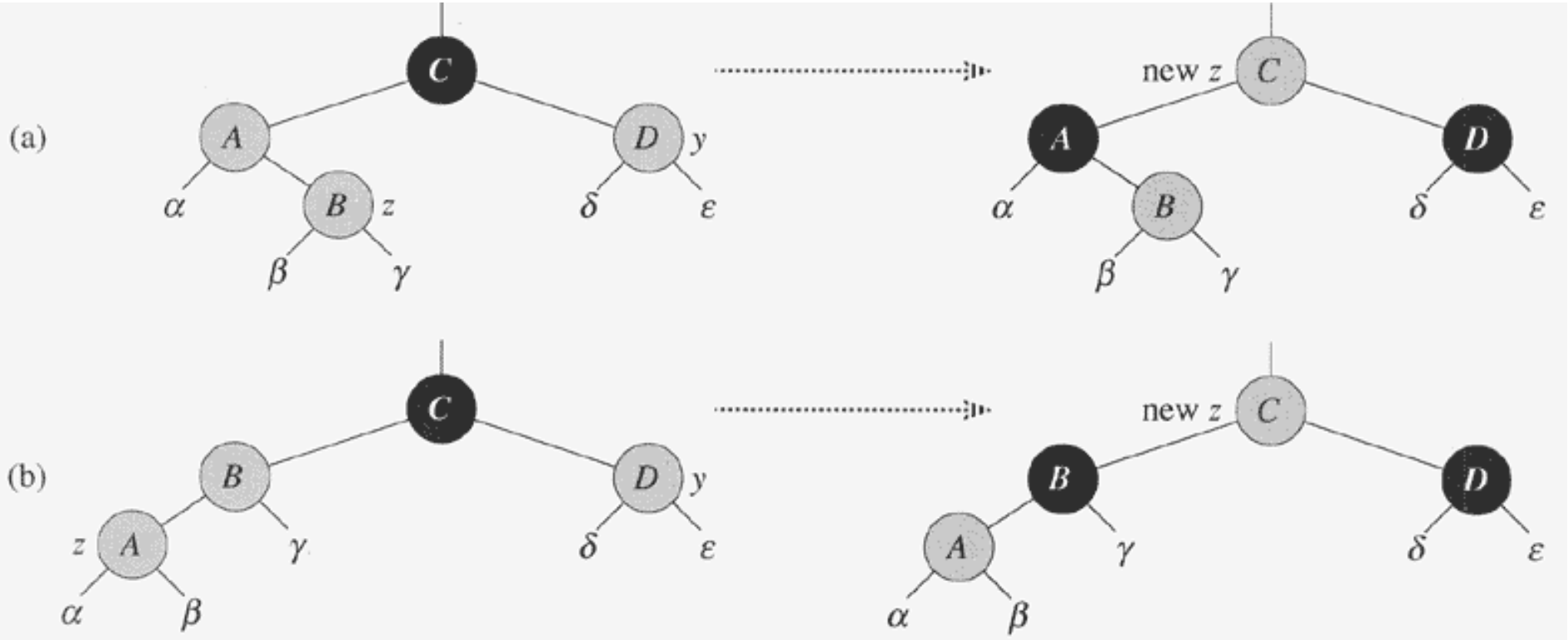
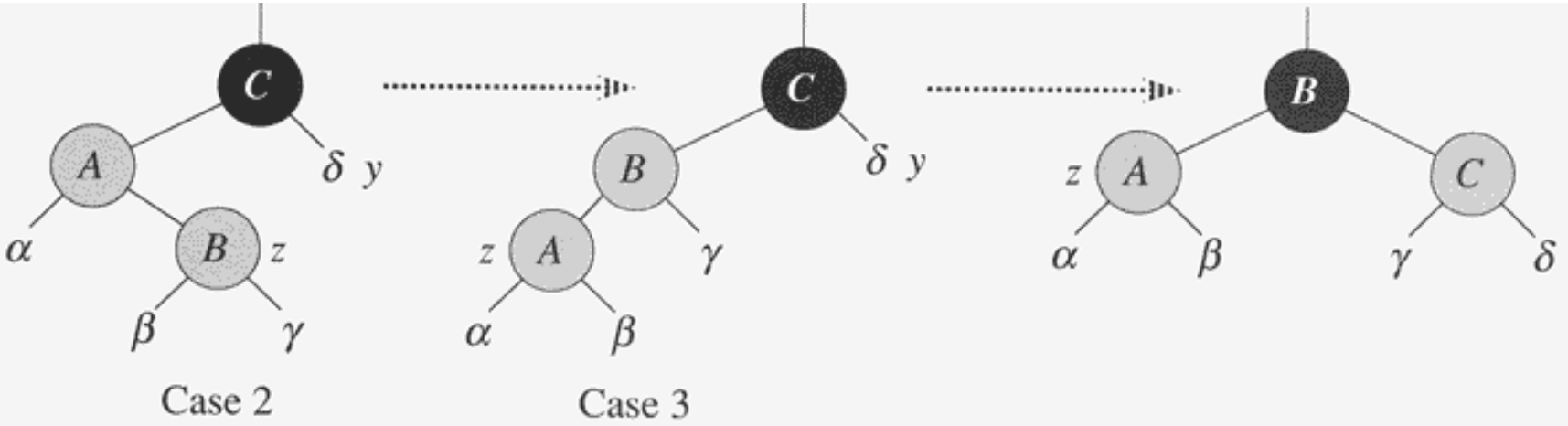


Figure 13.4 The operation of RB-INSERT-FIXUP. **(a)** A node z after insertion. Since z and its parent $p[z]$ are both red, a violation of property 4 occurs. Since z's uncle y is red, case 1 in the code can be applied. Nodes are recolored and the pointer z is moved up the tree, resulting in the tree 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.



case 1: z's uncle is red.

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.



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

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.

What is the running time of RB_INSERT_FIX? And RB_INSERT?

```
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          copy y's satellite data into z
16  if color[y] = BLACK
17     then RB-DELETE-FIXUP(T, x)
18  return y
```

```
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
```

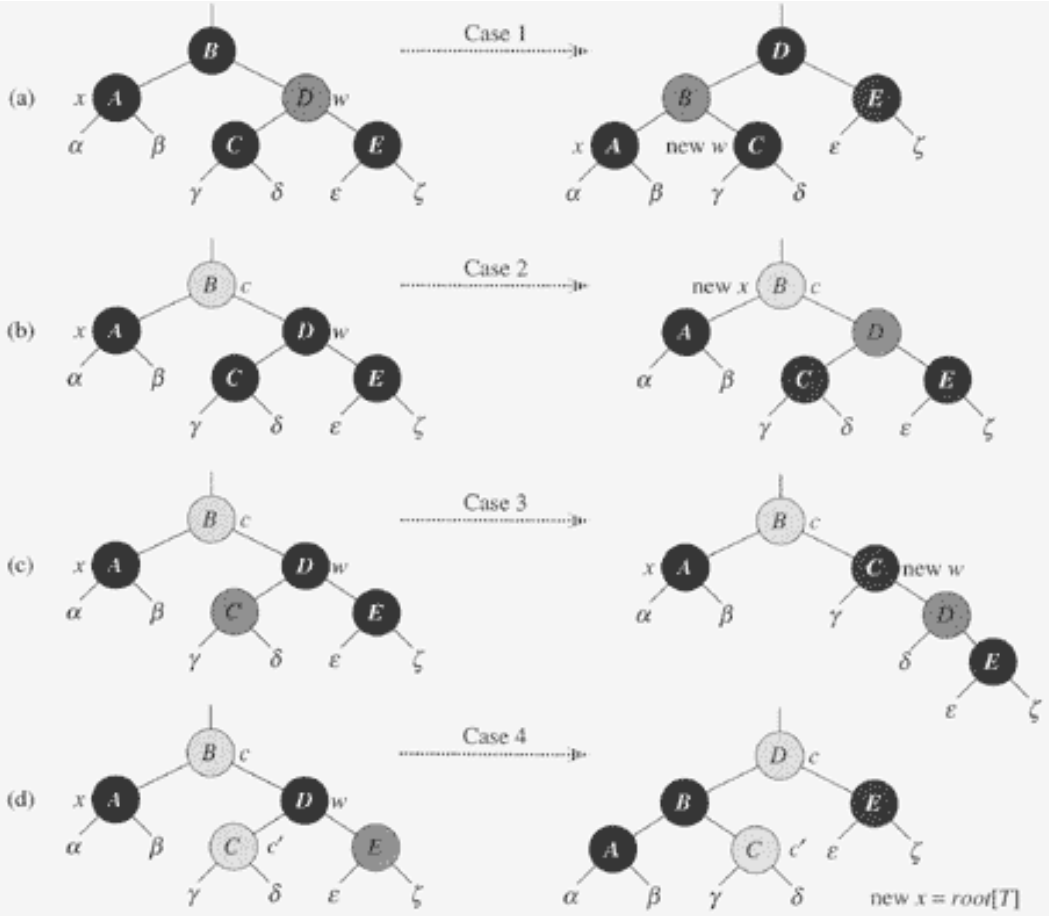


Figure 13.7 The cases in the **while** loop of the procedure RB-DELETE-FIXUP. Darkened nodes have *color* attributes BLACK, heavily shaded nodes have *color* attributes RED, and lightly shaded nodes have *color* attributes represented by c and c', which may be either RED or BLACK. The letters $\alpha, \beta, \ldots, \zeta$ represent arbitrary subtrees. 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. Any node pointed to by x has an extra black and is either doubly black or red-and-black. The only case that causes the loop to repeat is case 2. (a) Case 1 is transformed to case 2, 3, or 4 by exchanging the colors of nodes B and D and performing a left rotation. (b) In case 2, the extra black represented by the pointer x 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 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) 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.