Red Black Trees

Manuel Serna-Aguilera

While bst's are nice and all, they may end up horribly balanced to the point where operations' run times are like that of a linked list. Red-black trees attempt to balance the tree so that operations take $O(\log_2 n)$ time.

A red-black tree is a binary search tree with one extra bit of storage–its color, which can be either be red or black. Now, each node contains the main attributes: *color*, *key*, *left*, *right*, and p, with satellite data.

Properties

A red-black tree is a binary search tree that satisfies the following red-black properties:

1. Every node is either red or black.
2. The root is black.
3. Every leaf is black (the NIL leaves).
4. If a node is red, then both of its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes (all paths from the root must contain the same number of black nodes).

Of course, we tack on the regular bst properties.

All leaf nodes have two children, they are NIL nodes, which are black. In terms of pointers, we can have all the children pointers of the leaf nodes point to a single special node called $T.nil$ and have its color be black.

Black height: we call the number of black nodes on any simple path from, but *not* including, a node x down to a leaf the black height of a node, denoted $bh(x)$. The ***black-height*** of a red-black tree is the black height of its root.

Lemma. *A red-black tree with n internal nodes has height at most $2 \cdot \log_2 (n+1)$.*

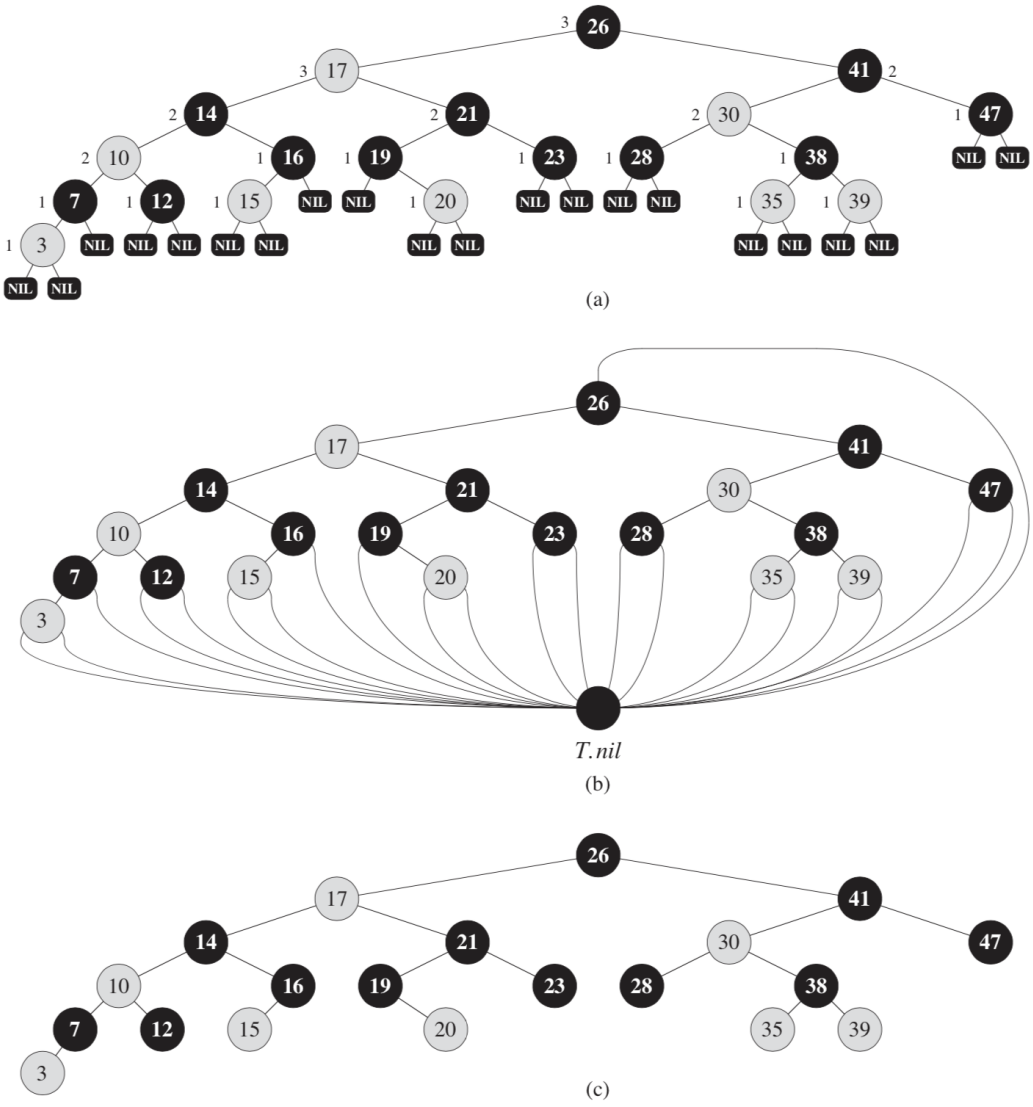

Figure 1: A red-black tree from the book, red nodes are shaded while black nodes are darkened. Notice all the red-black properties apply to this tree. **(a)** Each node is shown with its black height, while also showing all the NIL children, recall they must all be black. **(b)** Simplifies this view to have all pointers refer to a single node called $T.nil$. **(c)** What the book will draw to avoid having to draw all the lines and $T.nil$.

Rotations

When we insert or delete nodes to/from a red-black tree, the red-black properties may be violated, and we may have to recolor nodes and change the pointer structure. We change the *pointer structure* through rotation, which is a subroutine that move two nodes and their subtrees around. Note that colors are not changed in the procedure.

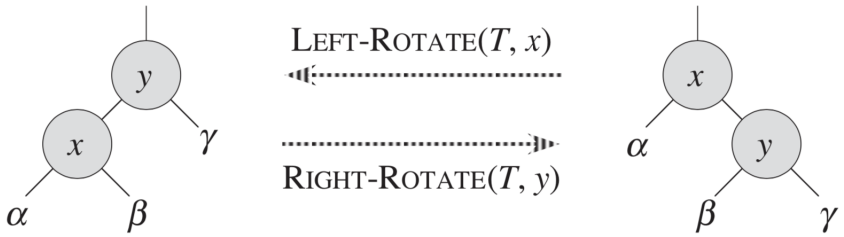

Figure 2: The rotation operations on a binary search tree.

For left rotate on x, we rotate around the link between x and its right non-nil child y. First we disconnect x and y so y does not reference x. Then we move the subtree β from being y's left subtree to being x's right subtree (this is the only subtree that "jumps" nodes). Next, have $y.p$ point to x's parent, while making x's parent y. We then make $y.left$ x. The process is symmetrical for a right rotate, I have both procedures below.

LEFT-ROTATE(T, x)

```
1  y = x.right // set y
2  x.right = y.left // turn y's left subtree into x's right subtree
3  if y.left ≠ T.nil
4        y.left.p = x
5  y.p = x.p // link x's parent to y
6  if x.p == T.nil
7        T.root = y
8  elseif x == x.p.left
9        x.p.left = y
10 else
11       x.p.right = y
12 y.left = x // put x on y's left
13 x.p = y
```

RIGHT-ROTATE(T, x)

```
1   x = y.left // set x
2   y.left = x.right // turn x's left subtree β into y's right subtree
3   if x.right ≠ T.nil // if β is a populated subtree, y points to x
4       x.right.p = y
5   x.p = y.p // link y's parent to x
6   if y.p == T.nil // make x new root if y was the root
7       T.root = x
8   elseif y == y.p.right
9       y.p.right = x
10  else
11      y.p.left = x
12  x.right = y // put y on x's right
13  y.p = x
```

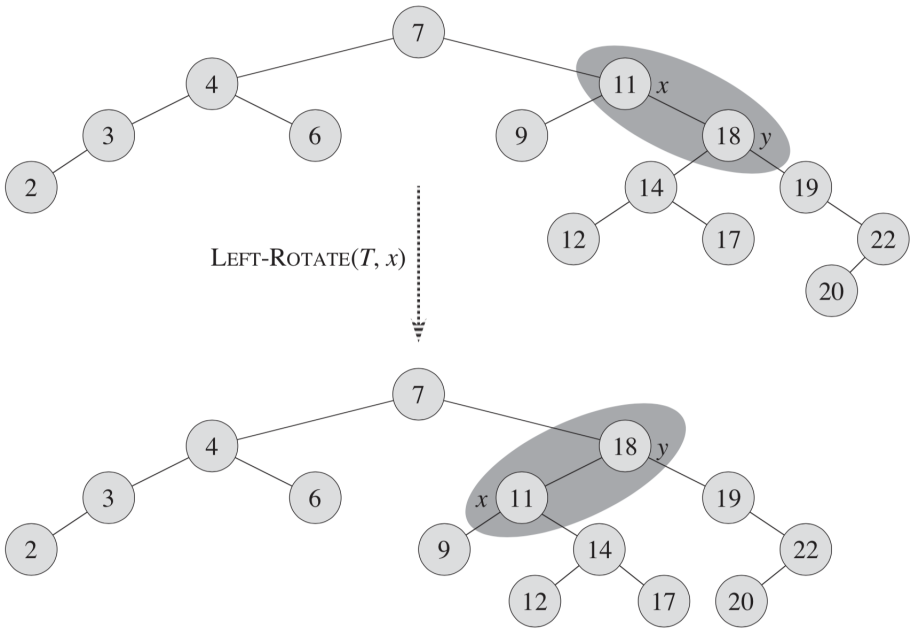

Figure 3: Calling LEFT-ROTATE on $x = 11$, as a note, when calling INORDER-TREE-WALK on both trees the traversals should output the same lists.

Insertion

Simple Overview

This is the basic process of inserting into a red-black tree.

1. Perform binary search to insert z and color it red.
2. Recolor and/or rotate nodes to fix violations.

Four cases to fix after inserting node z.

0. z is the root, thus, color z black (we can refer to this as case 0).
1. z's uncle is red, thus, recolor z's parent, uncle, and grandparent.

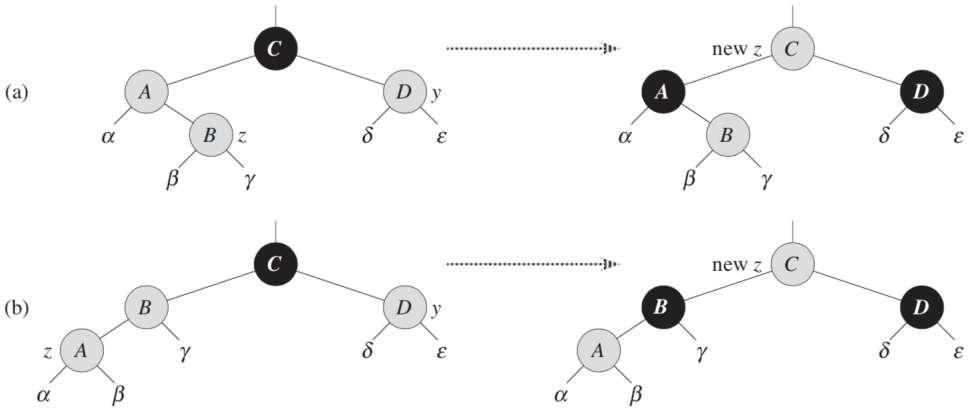

Figure 4: The book's depiction of case 1. It does not matter whether z is **(a)** a right child, or **(b)** a left child. Notice the black-height is preserved.

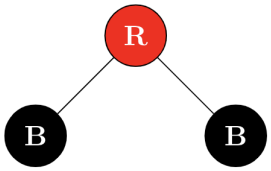

Figure 5: The generalized result of fixing case 1. The grandfather node will always be red, while its two chilren, z's parent and uncle, will be black.

> A side-effect to this is that the grandparent (new z) and its parent may both be red, thus we move on to case 2.

2. The "triangle" case: z's uncle y is black and z, its parent, and z's grandparent form a triangle-shape in the tree. We must rotate z's parent in the opposite direction of z, so if z is a right child, perform left-rotate on the parent, and vice versa. Notice no re-coloring is done, we only rotate, thus, we move on to case 3.

3. The "line" case: z's uncle y is black and z, its parent, and z's grandparent form a straight line in the tree. Node z and its parent are either both left or right children. To finally produce a valid red-black tree, rotate z's grandparent in the opposite direction of z.

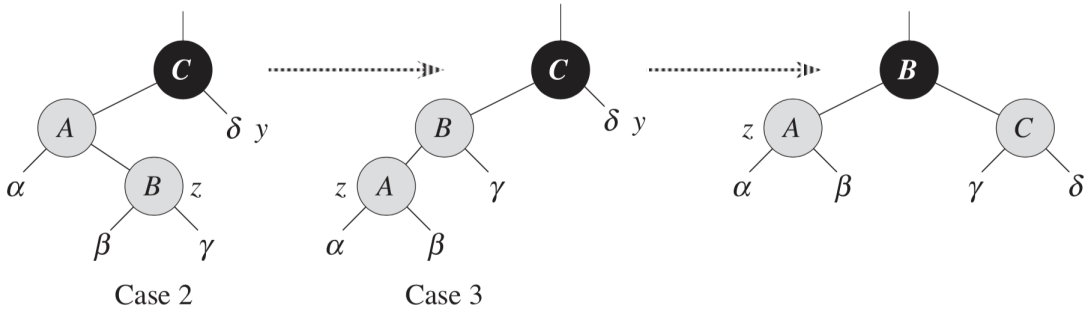

Figure 6: Case 2 leads into case 3. z's parent becomes the new root of the subtree, notice the black-height is preserved as the root is still black, we just moved red-nodes to the left and right subtrees of z's parent, we now have a valid red-black tree again!

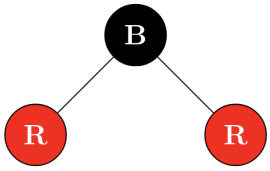

Figure 7: The generalized result of fixing cases 2 and 3. Node z and its grandparent are now red and children to z's parent, which is now black.

Procedures

The procedure RB-INSERT adds several new lines to TREE-SEARCH in order to properly insert a node into a red-black tree in $O(\log_2 n)$ time. It inserts the node z, assigns $T.nil$ to its children pointers, then assigns its color to be red, and finally calls the subroutine RB-INSERT-FIXUP to check for violations.

```
RB-INSERT(T, z)
 1  y = T.nil
 2  x = T.root
 3  while x ≠ T.nil // perform regular binary search
 4        y = x
 5        if z.key < x.key
 6              x = x.left
 7        else
 8              x = x.right
 9  z.p = y // make y the parent, insert z in the appropriate place
10  if y == T.nil
11        T.root = z
12  elseif z.key < y.key
13        y.left = z
14  else
15        y.right = z
16  z.left = T.nil
17  z.right = T.nil
18  z.color = RED
19  RB-INSERT-FIXUP(T, z)
```

The procedure RB-INSERT-FIXUP will check if z's parent is red, if so, we must then see which subtree the parent falls under, at this point solving for left or right subtrees is symmetrical. The procedure then checks which cases to fix discussed above, afterwards, the procedure calls itself on the grandparent to check if the violations have moved up the tree. Finally, the root is colored black to take care of case 0.

```
RB-INSERT-FIXUP(T, z)
 1  while z.p.color == RED
 2      if z.p == z.p.p.left
 3          y = z.p.p.right
 4          if y.color = RED // case 1
 5              z.p.color = BLACK
 6              y.color = BLACK
 7              z.p.p.color = RED
 8              z = z.p.p
 9          else
10              if z == z.p.right // case 2
11                  z = z.p
12                  LEFT-ROTATE(T, z)
13              z.p.color = BLACK // case 2 leads into case 3
14              z.p.p.color = RED
15              RIGHT-ROTATE(T, z.p.p)
16      else
17          y = z.p.p.left
18          if y.color = RED // case 1
19              z.p.color = BLACK
20              y.color = BLACK
21              z.p.p.color = RED
22              z = z.p.p
23          else
24              if z == z.p.left // case 2
25                  z = z.p
26                  RIGHT-ROTATE(T, z)
27              z.p.color = BLACK // case 2 leads into case 3
28              z.p.p.color = RED
29              LEFT-ROTATE(T, z.p.p)
30  T.root.color == BLACK
```

Deletion

Like the other red-black procedures, deletion takes $O(\log_2 n)$ time. Deletion is more complicated than regular deletion from a binary search tree, the procedure the book uses is optimized, there are other (many more) implementations that deal with **six** cases instead of four.
To start, the TRANSPLANT procedure needs to be modified to be used with a red-black tree. The procedure uses $T.nil$ instead of NIL, and u's parent becomes v's parent unconditionally.

```
RB-TRANSPLANT(T, u, v)
1  if u.p == T.nil
2      T.root = v
3  elseif u == u.p.left
4      u.p.left = v
5  else
6      u.p.right = v
7  v.p = u.p
```

The delete procedure for a red-black tree keeps track of two nodes, y and x, that might cause violations.

- Have node y point to z, and keep track of $y = z$'s original color.
- To handle the cases where z has no or one child, the first two if statements on lines 3 and 6 replace z with its non-nil subtree, called x, recall the transplant procedures take care of subtree assignment and parent pointers to and from the new node.
- If z has two non-nil children, then reassign y to be z's successor by taking the minimum of the right subtree (and store the color of this successor). Node y will take z's place if we go into the else statement starting in line 9. Node x in this case moves into y's original position, and points to y's right subtree.

```
RB-DELETE(T, u, v)
 1  y = z
 2  y-original-color = y.color
 3  if z.left == T.nil
 4       x = z.right
 5       RB-TRANSPLANT(T, z, z.right) // replace z with right non-nil subtree
 6  elseif z.right == T.nil
 7       x = z.left
 8       RB-TRANSPLANT(T, z, z.left) // replace z with left non-nil subtree
 9  else
10       y = TREE-MINIMUM(z.right)
11       y-original-color = y.color
12       x = y.right
13       if y.p == z
14            x.p = y
15       else RB-TRANSPLANT(T, y, y.right)
16            y.right = z.right
17            y.right.p = y
18       RB-TRANSPLANT(T, z, y)
19       y.left = z.left
20       y.left.p = y
21       y.color = z.color
22  if y-original-color == BLACK
23       RB-DELETE-FIXUP(T, x)
```

If node y was red (the node we moved to replace z), we are done. The black-height did not change. No red nodes are adjacent because y takes on z's color; if y was not z's right child, y's right child x that replaces it has to be black, we essentially removed a red node from one path in the tree, which does not damage the red-black height property. Since y could not have been the root, the root remains black.
If in these steps, node y was black, we may have violated one or more red-black properties, thus the call to RB-DELETE-FIXUP. According to the book, these are the potential causes:

- If y had been the root and a red child of y becomes the new root, we have violated property 2.
- If both x and $x.p$ are red, then we have violated property 4.
- Moving y within the tree causes any simple path that previously contained y to have one fewer black node. Thus, property 5 is now violated by any ancestor of y in the tree.

To fix the violation of property 5, the idea of a "**double black**" node is introduced. We must essentially move the double black to another node and perform rotations to move this to a satisfactory spot. The procedure RB-DELETE-FIXUP fixes this issue, it focuses on x, recall it replaced y, which replaced z.

```
RB-DELETE-FIXUP(T, u, v)
 1  while x ≠ T.root and x.color == BLACK
 2      if x == x.p.left
 3          w = x.p.right
 4          if w.color == RED // case 1 in then clause
 5              w.color = BLACK
 6              x.p.color = red
 7              LEFT-ROTATE(T, x.p)
 8              w = x.p.right
 9          if w.left.color == BLACK and w.right.color == BLACK // case 2
10              w.color = red
11              x = x.p
12          else
13              if w.right.color == BLACK // case 3
14                  w.left.color = BLACK
15                  w.color = red
16                  RIGHT-ROTATE(T, w)
17                  w = x.p.right
18              w.color = x.p.color // case 4
19              x.p.color = BLACK
20              w.right.color = BLACK
21              LEFT-ROTATE(T, x.p)
22              x = T.root
23      else
24          w = x.p.left
25          if w.color == RED // case 1 in then clause
26              w.color = BLACK
27              x.p.color = red
28              RIGHT-ROTATE(T, x.p)
29              w = x.p.left
30          if w.right.color == BLACK and w.left.color == BLACK // case 2
31              w.color = red
32              x = x.p
33          else
34              if w.left.color == BLACK // case 3
35                  w.right.color = BLACK
36                  w.color = red
37                  LEFT-ROTATE(T, w)
38                  w = x.p.left
39              w.color = x.p.color // case 4
40              x.p.color = BLACK
41              w.left.color = BLACK
42              RIGHT-ROTATE(T, x.p)
43              x = T.root
44  x.color = BLACK
```

Looking at the procedure RB-INSERT-FIXUP, x always points to a nonroot doubly black node. Next, determine if x is a left or right child (either case is symmetrical, just switch "left" and "right"). The pointer w refers to x's sibling, note that w cannot be $T.nil$ as the black-height from $x.p$ to x and to w must be the same. In each of the four cases RB-INSERT-FIXUP fixes, property 5 must be preserved after each transformation executed on the *subtrees* rooted at x and w.

Case 1: x's sibling w is red
Since w is red, its children must be black, thus we switch the colors of $x.p$ and w, and then perform a left rotation on $x.p$. Node x's new sibling, previously one of w's children, is now black. ***Move on to 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.

Case 2: x's sibling w is black, and both of w's children are black
We take one black from both x and w, and put it on $x.p$ (which could be red or black). Next, treat $x.p$ as the new x, and keep going up the red-black tree. This is how we recolor nodes red to adjust the black-height, thus satisfying property 5. ***Stop, we are done***!

Case 3: x's sibling w is black, w's left child is red, and and w's right child is black
Switch the colors of w and $w.left$, and then right-rotate on w. Node x's new sibling w is black with a red right child. ***Move on to case 4***.

Case 4: x's sibling w is black, and w's right child is red
This case makes several color changes (look at the procedure), and then left-rotate on $x.p$. Doing so allows w to remove the extra black on x without any violations. Setting x as the root satisfies the first condition for the while loop and the red-black tree is finally fixed. ***Stop, we are done***!

Analysis

With a red-black tree with n nodes having a height of $O(\log_2 n)$, only deleting takes $O(\log_2 n)$ time. In RB-DELETE-FIXUP, cases 1, 3, and 4 only take constant time, case 2 is the only case where the while loop repeats going up the tree at most $O(\log_2 n)$ times. Thus RB-DELETE-FIXUP takes $O(\log_2 n)$.

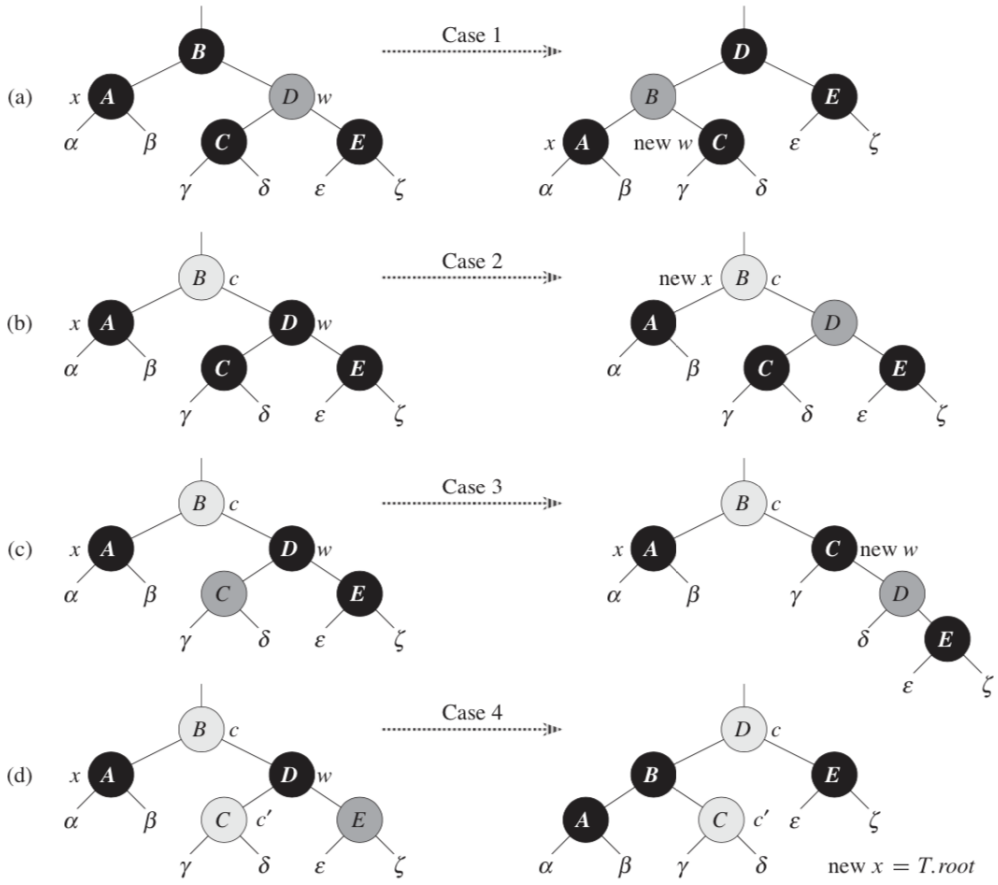

Figure 13.7 The cases in the **while** loop of the procedure RB-DELETE-FIXUP. Darkened nodes have *color* attributes BLACK, heavily shaded nodes have *color* attributes RED, and lightly shaded nodes have *color* attributes represented by c and c', which may be either RED or BLACK. The letters $\alpha, \beta, \ldots, \zeta$ represent arbitrary subtrees. Each case transforms the configuration on the left into the configuration on the right by changing some colors and/or performing a rotation. Any node pointed to by x has an extra black and is either doubly black or red-and-black. Only case 2 causes the loop to repeat. **(a)** Case 1 is transformed 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 moves 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)** Case 4 removes the extra black represented by x by changing some colors and performing a left rotation (without violating the red-black properties), and then the loop terminates.