1 Red Black Tree

1.1 definition

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 (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

注： The "usual" red-black trees can have two red children (but no red child to a red node) and correspond to 2-4 trees.

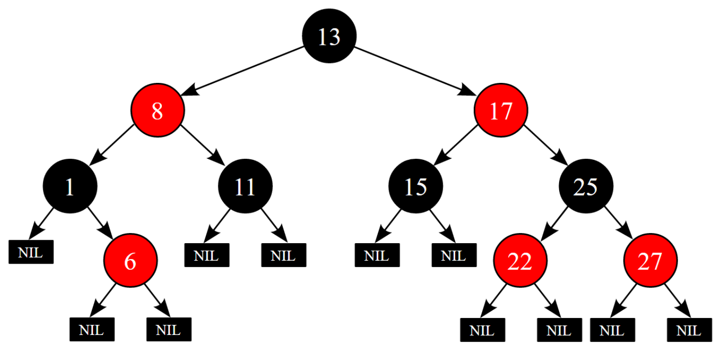

图 1.1: red black tree sample, node 25 has two red children.

Proposition 13.1
A red-black tree with n internal nodes has height at most 2lg(n+1).
Proof We start by showing 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 (T: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 $2^{bh(x)}-1$ 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.

1.2 rotations

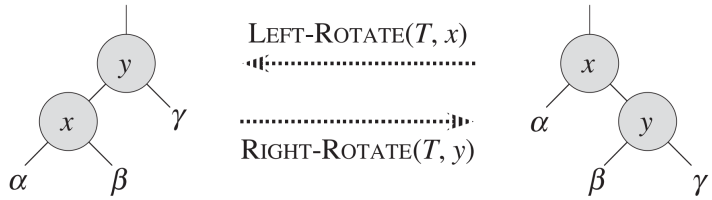

图 1.2: The rotation operations on a binary search tree. The operation LEFT-ROTATE(T,x) transforms the configuration of the two nodes on the right into the configuration on the left by changing a constant number of pointers.

```
LEFT-ROTATE(T, x)
y = x.right
x.right = y.left
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p
if x.p == T.nil
    T.root = y
elseif x == x.p.left
    x.p.left = y
else x.p.right = y
y.left = x
x.p = y
```

```
RIGHT-ROTATE(T, y)
x = y.left
y.left = x.right
if x.right ≠ T.nil
    x.right.p = y
x.p = y.p
if y.p == T.nil
    T.root = x
elseif y == y.p.left
    y.p.left = x
else y.p.right = x
x.right = y
y.p = x
```

1.3 insertion

RB-INSERT(T, z)

```
y = T.nil
x = T.root
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-INSERT-FIXUP(T, z)
```

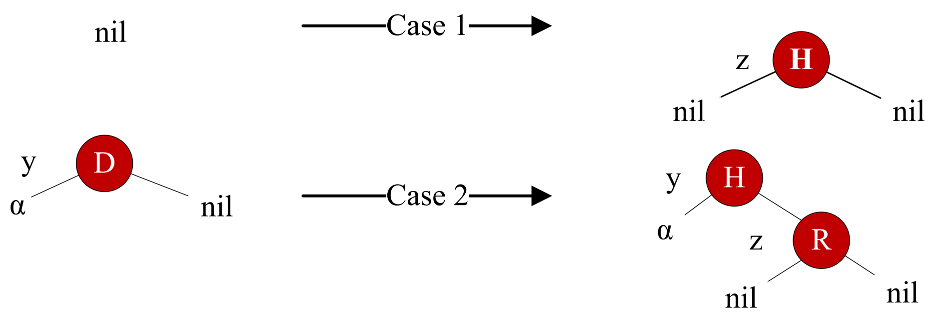

图 1.3: rb insert violated cases that need to be fixed up

RB-INSERT-FIXUP(T, z)

```
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK                   // case1
            y.color = BLACK
            z.p.p.color = RED
            z = z.p.p                           // case1
        else if z == z.p.right
                z = z.p                         // case2
                LEFT-ROTATE(T, z)               // case2
            z.p.color = BLACK                   // case3
            z.p.p.color = RED
            RIGHT-ROTATE(T, z.p.p)              // case3
    else (same as then clause with “right” and “left” exchanged)
T.root.color = BLACK
```

1.3.1 rb insert fixup cases explained

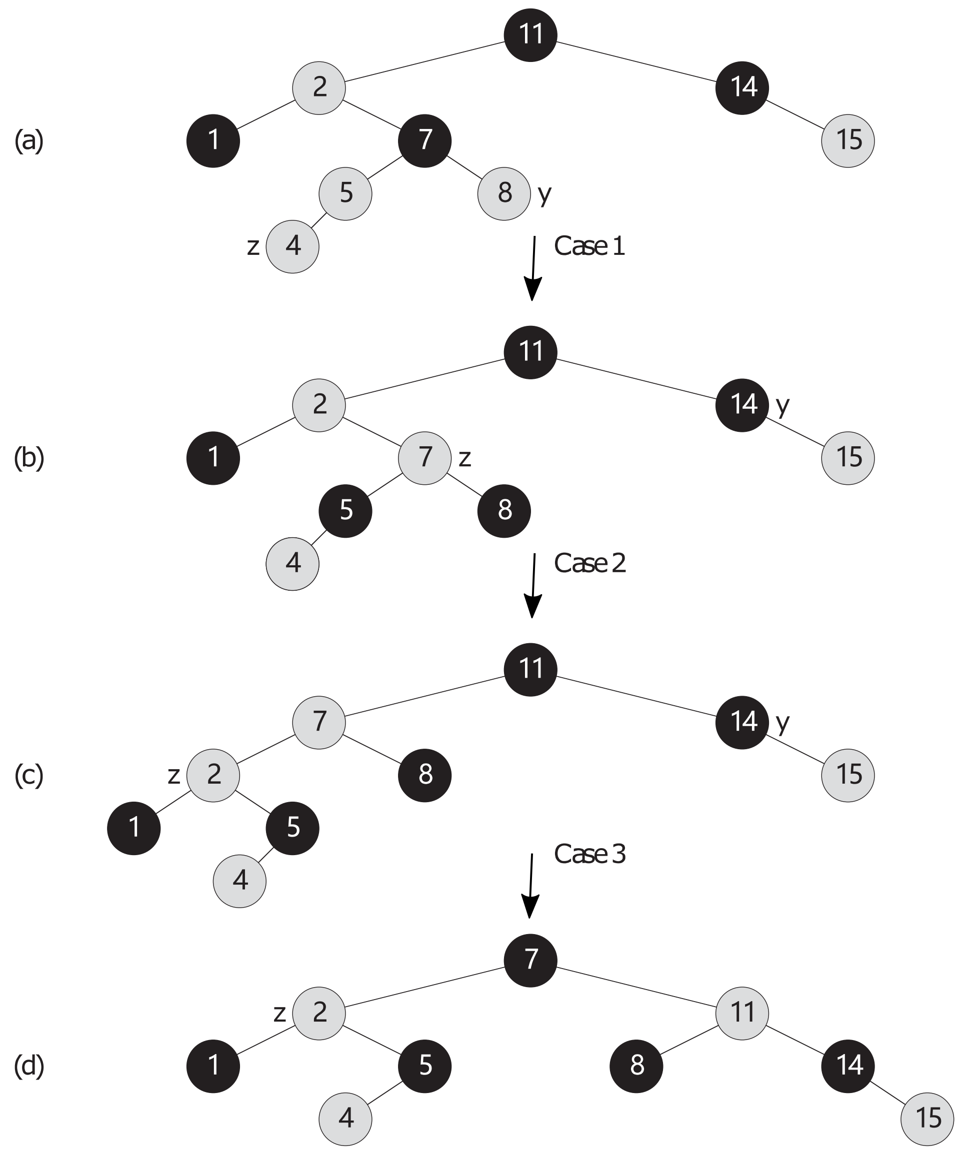

图 1.4: The cases in RB-INSERT-FIXUP.

(a) A node z after insertion. Because both z and its parent z:p are red, a violation of property 4 occurs. Since z' s uncle y is red, case 1 in the code applies. We recolor nodes and move the pointer z 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 z:p, case 2 applies. We perform a left rotation, and the tree that results is shown in (c).
Now, z is the left child of its parent, and case 3 applies. Recoloring and right rotation yield the tree in (d), which is a legal red-black tree.

1.3.2 Analysis

What is the running time of RB-INSERT? Since the height of a red-black tree on n nodes is O(lg n), lines 1–16 of RB-INSERT take O(lg n) time. In RB-INSERTFIXUP, the while loop repeats only if case 1 occurs, and then the pointer z moves two levels up the tree. The total number of times the while loop can be executed is therefore O(lg n). Thus, RB-INSERT takes a total of O(lg n) time. Moreover, it never performs more than two rotations, since the while loop terminates if case 2 or case 3 is executed.

1.4 delete

```
RB-TRANSPLANT(T,u,v)
if u.p == T.nil
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
v.p = u.p
```

```
RB-DELETE(T,z)
y = z
y-original-color = y.color
if z.left == T.nil
    x = z.right
    RB-TRANSPLANT(T,z,z.right)
elseif z.right == T.nil
    x = z.left
    RB-TRANSPLANT(T,z,z.left)
else y = TREE-MINIMUM(z.right)
    y-original-color = y.color
    x = y.right
    if y.p == z
        x.p = y
    else RB-TRANSPLANT(T,y,y.right)
        y.right = z.right
        y.right.p = y
    RB-TRANSPLANT(T,z,y)
    y.left = z.left
    y.left.p = y
    y.color = z.color
if y-original-color == BLACK
    RB-DELETE-FIXUP(T.x)
```

```
RB-DELETE-FIXUP(T,x)
while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        if w.color == RED
            w.color = BLACK                          // case1
            x.p.color = RED
            LEFT-ROTATE(T,x.p)
            w = x.p.right                            // case1
        if w.left.color == BLACK and w.right.color == BLACK
            w.color = RED                            // case2
            x = x.p                                  // case2
        else if w.right.color == BLACK
                w.left.color = BLACK                 // case3
                w.color = RED
                RIGHT-ROTATE(T.w)
                w = x.p.right                        // case3
            w.color = x.p.color                      // case4
            x.p.color = BLACK
            w.right.color = BLACK
            LEFT-ROTATE(T,x.p)
            x = T.root                               // case4
    else (same as then clause with "right" and "left" exchanged)
x.color = BLACK
```

```
RB-DELETE(T, z)
y = z
y-original-color = y.color
if z.left == T.nil
    x = z.right
    RB-TRANSPLANT(T, z, z.right)
elseif z.right == T.nil
    x = z.left
    RB-TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    y-original-color = y.color
    x = y.right
    if y.p == z
        x.p = y
    else RB-TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    RB-TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
    y.color = z.color
if y-original-color == BLACK
    RB-DELETE-FIXUP(T.x)
```

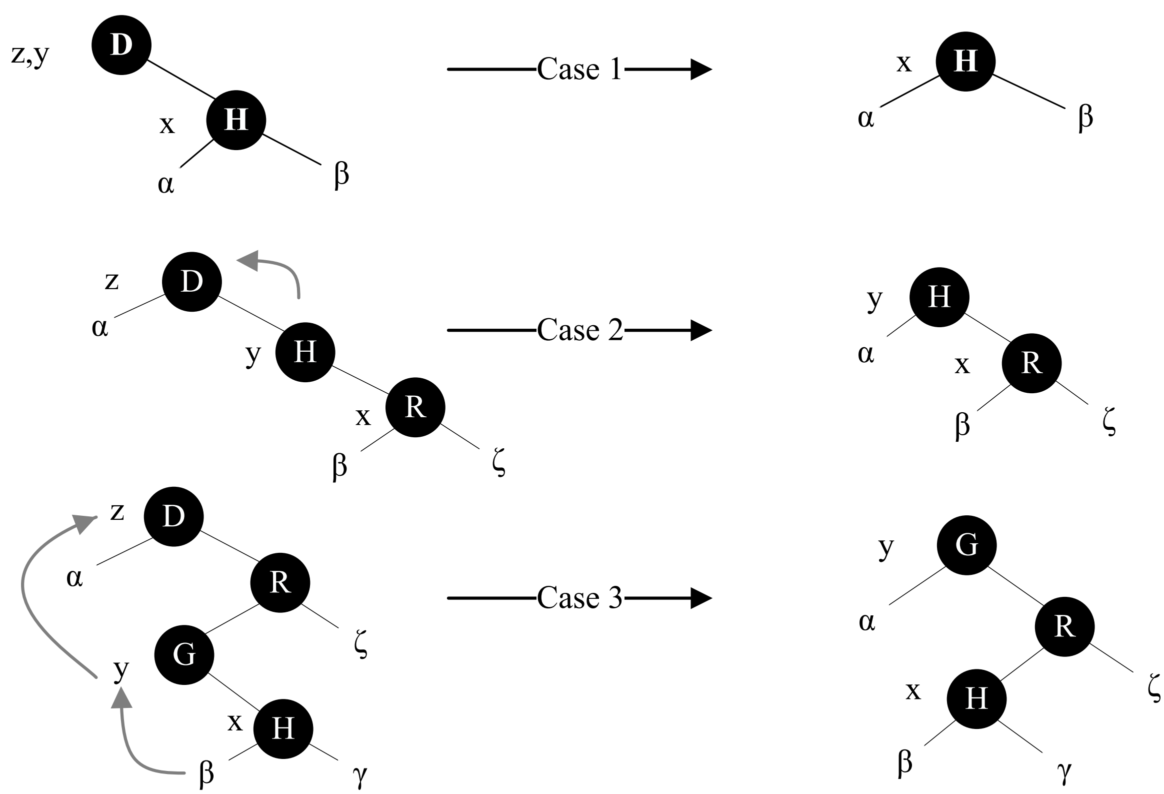

图 1.5: The cases in RB-DELETE.

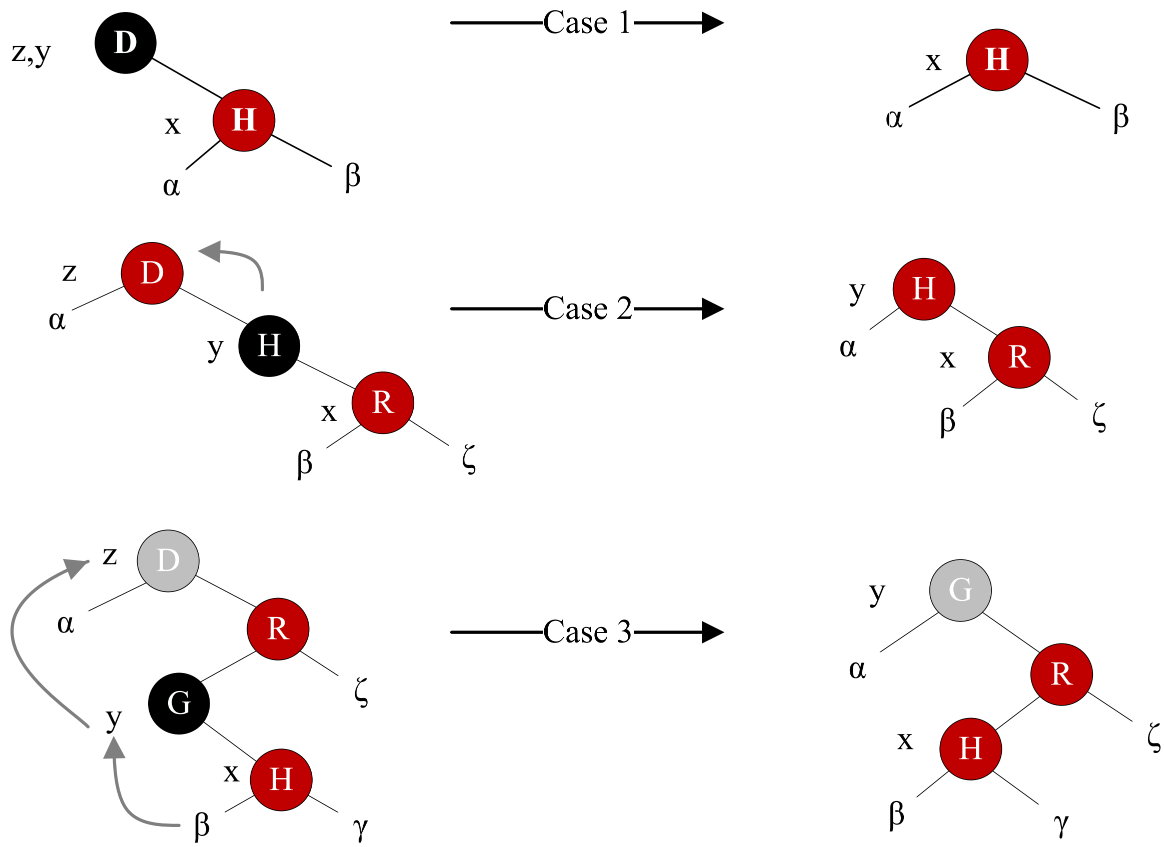

图 1.6: if y-original-color is black, the cases we need to fixup

1.4.1 If y-original-color is black, we need to fix up.

case1

1. If Z is root, x is red, property 1 violated
2. 2 if Z is not root, property 5 violated

case2

1. If x is red and z is red, property 4 violated
2. black height of the subtree rooted at z minus 1, property 5 violated.

case3

1. If x is red and node R is red, property 4 violated
2. black height of the subtree rooted at z minus 1, property 5 violated.

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

1.4.2 fix up cases when delete

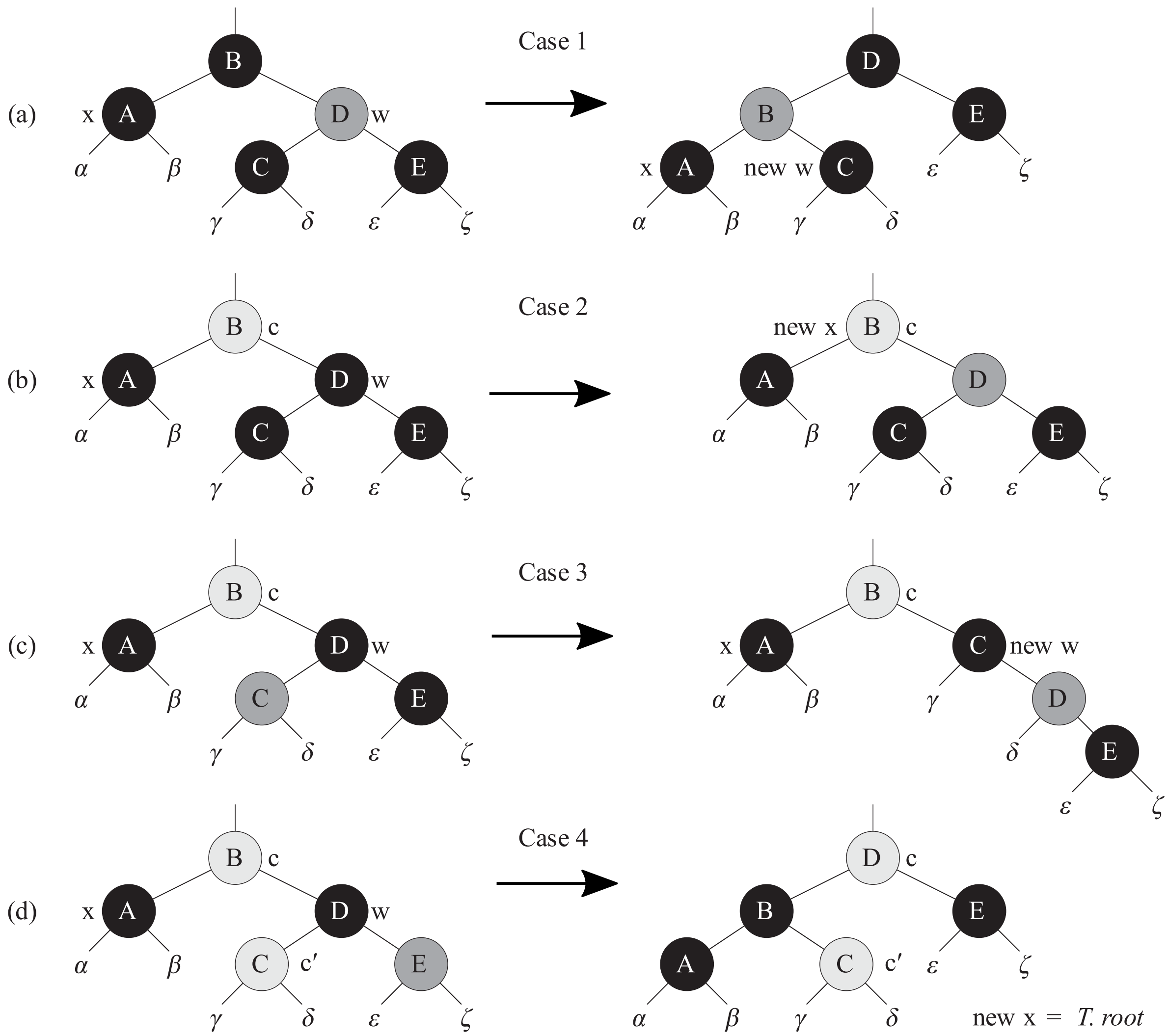

图 1.7: The cases in the while loop of the procedure RB-DELETE-FIXUP.

- 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.

1.4.3 Detailed explanation for the cases in the while loop of the procedure RB-DELETE-FIXUP .

Case 1: x's sibling w is red
Case 1 (lines 5–8 of RB-DELETE-FIXUP and Figure 13.7(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 $x.p$ and then perform a left-rotation on $x.p$ without violating any of the red-black properties. The new sibling of x, which is one of w's children prior to the rotation, 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.

Case 2: x's sibling w is black, and both of w's children are black
In case 2 (lines 10–11 of RB-DELETE-FIXUP and Figure 13.7(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. To compensate for removing one black from x and w, we would like to add an extra black to $x.p$, which was originally either red or black. We do so by repeating the **while** loop with $x.p$ as the new node x. Observe that if we enter case 2 through case 1, the new node x is red-and-black, since the original $x.p$ was red. Hence, the value c of the color attribute of the new node x is RED, and the loop terminates when it tests the loop condition. We then color the new node x (singly) black in line 23.

Case 3: x's sibling w is black, w's left child is red, and w's right child is black
Case 3 (lines 13–16 and Figure 13.7(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 $w.left$ 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: x's sibling w is black, and w's right child is red
Case 4 (lines 17–21 and Figure 13.7(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 $x.p$, we can remove the extra black on x, making it singly black, 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.

1.4.4 Analysis

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-DELETEFIXUP takes O(lg n) time. Within RB-DELETE-FIXUP, each of cases 1, 3, and 4 lead to termination 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, performing no rotations. 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)

2 reb black tree understanding

2.1 questions about deletion

the choose of y and x

in rb-delete, choose which node as y and x in different cases?

```
RB-DELETE(T, z)
y = z
y-original-color = y.color
if z.left == T.nil
    x = z.right
    RB-TRANSPLANT(T, z, z.right)
elseif z.right == T.nil
    x = z.left
    RB-TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    y-original-color = y.color
    x = y.right
    if y.p == z
        x.p = y
    else RB-TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    RB-TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
    y.color = z.color
if y-original-color == BLACK
    RB-DELETE-FIXUP(T.x)
```

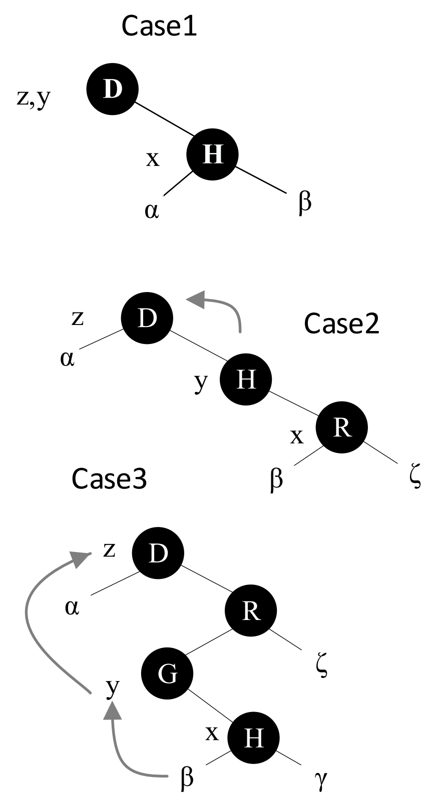

图 2.1: 3 cases in rb-delete

- z: the deleting node.
- 删除的3种case, 在case 1, 2, z 的node被抹去，z原来所在的node color也被抹去。
- case3 中,y节点替换z节点原来的位置，但y.color = z.color, 即z原来占位的节点，color没有丢失。但y原来所占的节点，color丢失了。

综上，y用来表示丢失color的节点，x表示y的右孩子