

A_DSA Pseudo Codes - Red Black Tree

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 node colors on any simple 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 attributes *color*, *key*, *left*, *right*, and *p*. If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value NIL. We shall regard these NILs as being pointers to leaves (external nodes) of the binary search tree and the normal, key-bearing nodes as being internal nodes of the tree.

A red-black tree is a binary 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.

Rotations

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 not T:nil; *x* may be any node in the tree whose right child is not T: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.

Pseudo Code of Left Rotation

```
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 x.p.right = y
11 y.left = x                  // put x on y’s left
12 x.p = y
```

Insertion

We can insert a node into an n -node red-black tree in $O(\lg n)$ time. To do so, we use a slightly modified version of the TREE-INSERT procedure (Section 12.3) to insert node z into the tree T as if it were an ordinary binary search tree, and then we color z red. (Exercise 13.3-1 asks you to explain why we choose to make node z red rather than black.) To guarantee that the red-black properties are preserved, we then call an auxiliary procedure RB-INSERT-FIXUP to recolor nodes and perform rotations. The call RB-INSERT(T, z) inserts node z , whose *key* is assumed to have already been filled in, into the red-black tree T .

```

RB-INSERT( $T, z$ )
1   $y = T.nil$ 
2   $x = T.root$ 
3  while  $x \neq T.nil$ 
4       $y = x$ 
5      if  $z.key < x.key$ 
6           $x = x.left$ 
7      else  $x = x.right$ 
8   $z.p = y$ 
9  if  $y == T.nil$ 
10      $T.root = z$ 
11 elseif  $z.key < y.key$ 
12      $y.left = z$ 
13 else  $y.right = z$ 
14  $z.left = T.nil$ 
15  $z.right = T.nil$ 
16  $z.color = RED$ 
17 RB-INSERT-FIXUP( $T, z$ )

```

Pseudocode of RB-INSERT-FIXUP

```

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  $y.color == RED$ 
5               $z.p.color = BLACK$  // case 1
6               $y.color = BLACK$  // case 1
7               $z.p.p.color = RED$  // case 1
8               $z = z.p.p$  // case 1
9          else if  $z == z.p.p.right$ 
10              $z = z.p$  // case 2
11             LEFT-ROTATE( $T, z$ ) // case 2
12              $z.p.color = BLACK$  // case 3
13              $z.p.p.color = RED$  // case 3
14             RIGHT-ROTATE( $T, z.p.p$ ) // case 3
15      else (same as then clause
16             with “right” and “left” exchanged)
17   $T.root.color = BLACK$ 

```

Deletion

The procedure for deleting a node from a red-black tree is based on the TREE-DELETE procedure (Section 12.3). First, we need to customize the TRANSPLANT subroutine that TREE-DELETE calls so that it applies to a red-black tree:

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  $u.p.right = v$ 
6   $v.p = u.p$ 

```

Pseudo code of RB-DELETE

RB-DELETE(T, z)

```

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

```

Pseudo code of RB-DELETE-FIXUP

RB-DELETE-FIXUP(T, x)

```
1  while  $x \neq T.root$  and  $x.color == BLACK$ 
2      if  $x == x.p.left$ 
3           $w = x.p.right$ 
4          if  $w.color == RED$ 
5               $w.color = BLACK$  // case 1
6               $x.p.color = RED$  // case 1
7              LEFT-ROTATE( $T, x.p$ ) // case 1
8               $w = x.p.right$  // case 1
9          if  $w.left.color == BLACK$  and  $w.right.color == BLACK$ 
10              $w.color = RED$  // case 2
11              $x = x.p$  // case 2
12         else if  $w.right.color == BLACK$ 
13              $w.left.color = BLACK$  // case 3
14              $w.color = RED$  // case 3
15             RIGHT-ROTATE( $T, w$ ) // case 3
16              $w = x.p.right$  // case 3
17              $w.color = x.p.color$  // case 4
18              $x.p.color = BLACK$  // case 4
19              $w.right.color = BLACK$  // case 4
20             LEFT-ROTATE( $T, x.p$ ) // case 4
21              $x = T.root$  // case 4
22         else (same as then clause with “right” and “left” exchanged)
23      $x.color = BLACK$ 
```

CASES

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.

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-DELETE-FIXUP 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 $O(\lg n)$.

Example

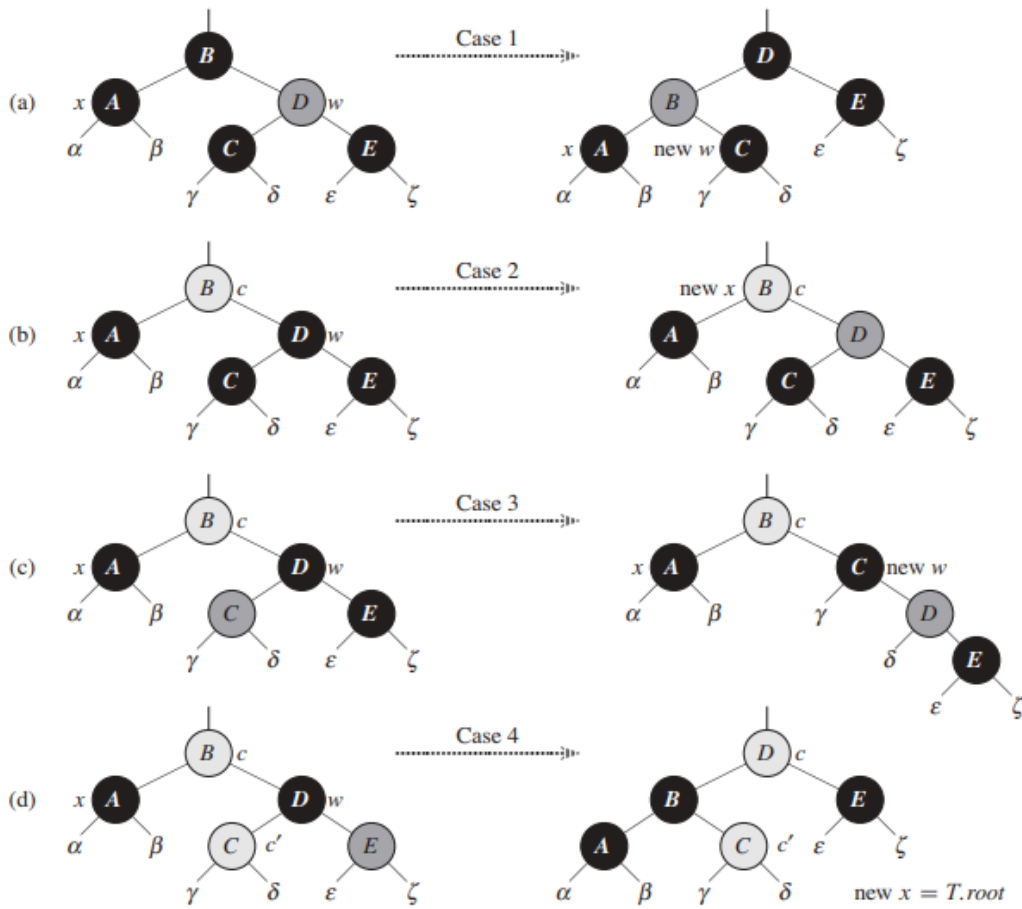


Figure 13.7 The cases in the **while** loop of the procedure **RB-DELETE-FIXUP**. Darkened nodes have *color* attributes **BLACK**, heavily shaded nodes have *color* attributes **RED**, and lightly shaded nodes have *color* attributes represented by c and c' , which may be either **RED** or **BLACK**. The letters $\alpha, \beta, \dots, \zeta$ 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.