RED-BLACK TREES

Complete Reference | Self-Balancing BST | O(log n) Guaranteed

Properties | Rotations | Insertion (3 Cases) | Deletion (4 Cases) | CLRS Standard

What is a Red-Black Tree?

A self-balancing BST where each node stores an extra color bit (RED or BLACK).

Guarantees O(log n) search, insert, delete by enforcing 5 properties after every modification.

Used in: Linux CFS scheduler, Java TreeMap, C++ std::map, database indexing.

The 5 Sacred Properties (must hold at ALL times)

P1 **P1 Node Color:**
Every node is either RED or BLACK.

P2 **P2 Root Black:**
The root of the tree is always BLACK.

P3 **P3 NIL Leaves:**
Every leaf (NIL / null sentinel) is BLACK.

P4 **P4 Red Rule:**
If a node is RED, BOTH its children must be BLACK. (No two consecutive reds.)

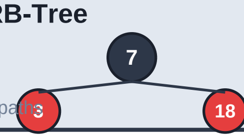
P5 **P5 Black Height:**
Every path from any node to its descendant NIL leaves has the SAME count of black nodes.

Mnemonic: "CoRoL ReB"

Co=Color Ro=Root L=Leaves Re=Red-rule B=Black-height

Remember: "Colors Root Leaves RedRed BlackPaths"

Example Valid RB-Tree



bh = 2 on all root-to-NIL paths

Why These Properties Guarantee O(log n)

P4 + P5 together ensure: longest root-to-leaf path <= 2 x shortest path.

Tree height h <= 2 * log2(n+1). A subtree at node x has >= 2^bh(x) - 1 internal nodes.

Time Complexity Comparison

Operation	BST (avg)	BST (worst)	RB-Tree	AVL Tree
Search	O(log n)	O(n)	**O(log n)**	O(log n)
Insert/Delete	O(log n)	O(n)	**O(log n)**	O(log n)

ROTATIONS: The Building Block

O(1) local restructuring | Preserves BST in-order property

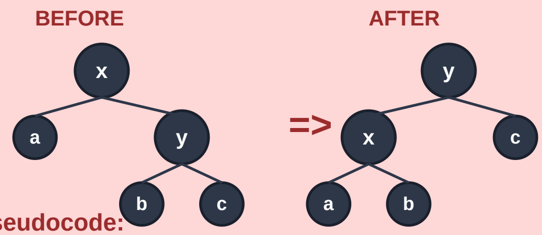
Key Facts About Rotations

O(1) time (max 5 pointer updates). In-order traversal is UNCHANGED. Used in insert/delete fixups.

LEFT-ROTATE: right child rises. RIGHT-ROTATE: left child rises. They are exact mirrors.

LEFT-ROTATE(T, x)

x goes DOWN-LEFT, y (x.right) goes UP

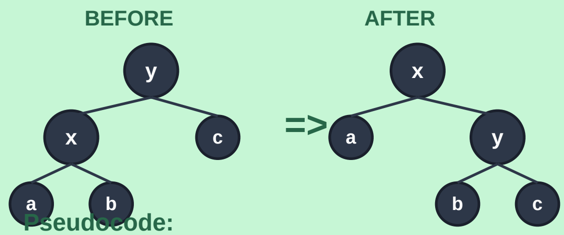


Pseudocode:

```
LEFT-ROTATE(T, x):
  y = x.right
  x.right = y.left        // adopt b
  if y.left != NIL:
    y.left.parent = x
  y.parent = x.parent     // y takes x's spot
  if x == root: root = y
  elif x == x.parent.left:
    x.parent.left = y
  else: x.parent.right = y
  y.left = x; x.parent = y
```

RIGHT-ROTATE(T, y)

y goes DOWN-RIGHT, x (y.left) goes UP



Pseudocode:

```
RIGHT-ROTATE(T, y):
  x = y.left
  y.left = x.right        // adopt b
  if x.right != NIL:
    x.right.parent = y
  x.parent = y.parent     // x takes y's spot
  if y == root: root = x
  elif y == y.parent.right:
    y.parent.right = x
  else: y.parent.left = x
  x.right = y; y.parent = x
```

Rotation Insights & Mnemonics

LEFT-ROTATE: "x sinks left, its right child y rises to take its place."

RIGHT-ROTATE: Exact mirror -- "y sinks right, its left child x rises."

In-order stays same: a < x < b < y < c. Both operations are O(1).

Mnemonic: "Opposite child rises" -- LEFT rotate = RIGHT child up, RIGHT rotate = LEFT child up.

INSERTION: Algorithm & Fixup

Insert as RED, then fix violations via recolor/rotate

Insertion: 2-Phase Process

Phase 1: Standard BST Insert

Walk down tree. Insert new node z at correct position. Color z = RED.

Phase 2: RB-INSERT-FIXUP(T, z)

Fix violations. Only P2 (root black) or P4 (no red-red) can break. Loop up the tree.

Why Insert as RED?

BLACK insertion breaks P5 (black-height) on EVERY path -- very hard to fix! RED only maybe breaks P4.

RB-INSERT(T, z)

```
y = T.nil; x = T.root
while x != T.nil:        // find position
    y = x
    if z.key < x.key:
        x = x.left
    else: x = x.right
z.parent = y
if y == T.nil: T.root = z
elif z.key < y.key: y.left = z
else: y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED             // NEW = RED
RB-INSERT-FIXUP(T, z)     // fix it
```

RB-INSERT-FIXUP(T, z)

```
while z.parent.color == RED:
 if z.parent == z.parent.parent.left:
  y = z.parent.parent.right // uncle
  if y.color == RED:        // CASE 1
    z.parent.color = BLACK
    y.color = BLACK
    z.parent.parent.color = RED
    z = z.parent.parent
  else:
   if z == z.parent.right:  // CASE 2
    z=z.parent; LEFT-ROTATE(T,z)
   z.parent.color = BLACK   // CASE 3
   z.parent.parent.color = RED
   RIGHT-ROTATE(T, z.parent.parent)
 else: ... // symmetric (swap L/R)
T.root.color = BLACK
```

INSERT-FIXUP: 3 Cases (parent is LEFT child of grandparent)

Mirror cases exist when parent is RIGHT child (swap all left/right).

Case	Condition	Action	Result
Case 1	Uncle is RED	Recolor: parent, uncle -> BLACK; grandparent -> RED; z = grandparent	**Pushes up**
Case 2	Uncle BLACK, z = right child	z = z.parent; LEFT-ROTATE(T, z) -- converts to Case 3	**-> Case 3**
Case 3	Uncle BLACK, z = left child	Parent -> BLACK, grandparent -> RED, RIGHT-ROTATE(G)	**DONE!**

Mnemonic: "Uncle RED? Recolor up. Uncle BLACK? Straighten (C2) then Rotate (C3)."

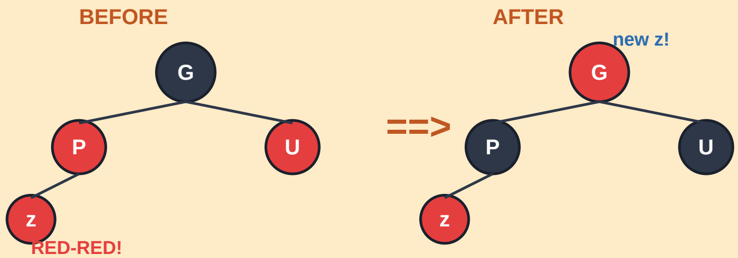
Insert Fixup Key Facts

At most 2 rotations total. O(log n) recolors (Case 1 can repeat up the tree).
Case 2 always leads to Case 3. Case 3 always terminates. Case 1 is the only one that loops.

INSERTION CASES: Visual Diagrams

Before/After tree diagrams for each case

Case 1: Uncle is RED -- Recolor Only, No Rotations

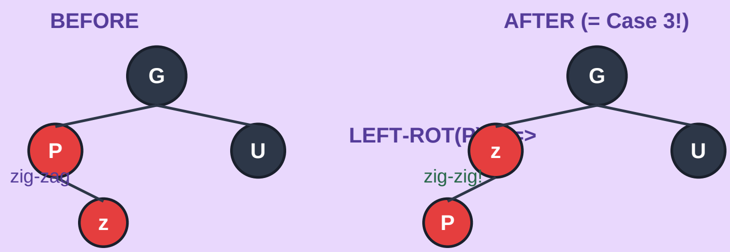


Steps:

1. Parent -> BLACK
2. Uncle -> BLACK
3. Grandparent -> RED
4. z = Grandparent
5. Continue loop

Black-height preserved (swapped colors evenly). Problem pushed up -- may repeat.

Case 2: Uncle BLACK, z is RIGHT child -- Zig-Zag: Rotate to Case 3

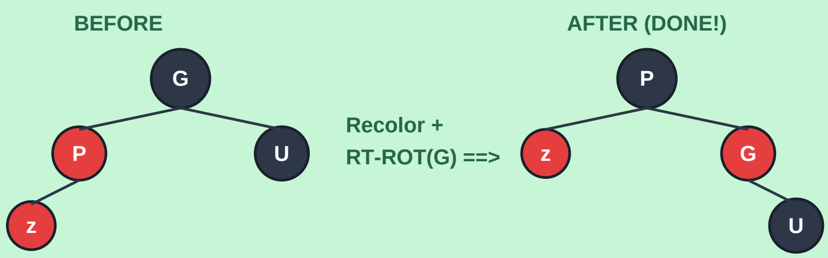


Steps:

1. z = z.parent
2. LEFT-ROTATE(T, z)
3. Now it's Case 3

Purpose: Convert zig-zag (right child) into zig-zig (left child) so Case 3 rotation works.

Case 3: Uncle BLACK, z is LEFT child -- Recolor + Right Rotate = DONE!



Steps:

1. Parent -> BLACK
2. Grandparent -> RED
3. RIGHT-ROTATE(T, G)

TERMINATES!

P becomes new subtree root (BLACK). All properties restored. Loop ends.

Insert Decision Flow

Uncle RED -> Case 1 (recolor, loop) | Uncle BLACK + zig-zag -> Case 2 -> Case 3 | Uncle BLACK + zig-zig -> Case 3 (DONE)

DELETION: Algorithm Overview

Most complex BST operation -- 4 fixup cases

Deletion: 3-Phase Process

Phase 1: Find node z

Standard BST locate. If 2 children, find in-order successor y.

Phase 2: Splice out

Remove the physically deleted node. Track replacement node x.

Phase 3: Fixup

RB-DELETE-FIXUP(T, x) called ONLY if removed node was BLACK.

Why Fix Only When BLACK Removed?

RED removed: no property breaks. BLACK removed: P5 (black-height) drops by 1 on affected paths.

RB-DELETE(T, z)

```
y = z; y_orig_color = y.color
if z.left == T.nil:        // 0-1 child
  x = z.right
  TRANSPLANT(T, z, z.right)
elif z.right == T.nil:
  x = z.left
  TRANSPLANT(T, z, z.left)
else:                      // 2 children
  y = MINIMUM(z.right)     // successor
  y_orig_color = y.color
  x = y.right
  if y.parent == z: x.parent = y
  else:
    TRANSPLANT(T, y, y.right)
    y.right = z.right
    y.right.parent = y
  TRANSPLANT(T, z, y)
  y.left = z.left
  y.left.parent = y
  y.color = z.color
if y_orig_color == BLACK:
  RB-DELETE-FIXUP(T, x)
```

RB-DELETE-FIXUP(T, x)

```
while x != T.root and x.color == BLACK:
 if x == x.parent.left:
  w = x.parent.right        // sibling
  if w.color == RED:         // CASE 1
    w.color=BLK; x.p.color=RED
    LEFT-ROTATE(T, x.parent)
    w = x.parent.right
  if w.left.color==B & w.right.color==B:
    w.color = RED            // CASE 2
    x = x.parent
  else:
    if w.right.color == BLACK:// CASE 3
      w.left.color=BLK; w.color=RED
      RIGHT-ROTATE(T, w)
      w = x.parent.right
    w.color = x.parent.color // CASE 4
    x.parent.color = BLACK
    w.right.color = BLACK
    LEFT-ROTATE(T, x.parent)
    x = T.root               // DONE
 else: ... // symmetric
x.color = BLACK
```

DELETE-FIXUP: 4 Cases (x is LEFT child of parent)

Mirror cases exist when x is RIGHT child.

Case	Condition	Action	Next
Case 1	Sibling w is RED	w->BLK, parent->RED, LEFT-ROTATE(parent), update w	**-> C2/3/4**
Case 2	w BLK, both w-children BLK	w->RED, x = x.parent (push double-black up)	**Loop/done**
Case 3	w BLK, w.left RED, w.right BLK	w.left->BLK, w->RED, RIGHT-ROTATE(w), update w	**-> Case 4**
Case 4	w BLK, w.right RED	w = parent color, parent->BLK, w.right->BLK, LEFT-ROT(P)	**DONE!**

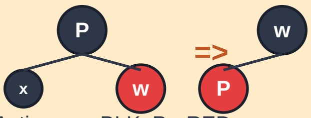
DELETION CASES: Visual Diagrams

x has "extra black" -- resolve or push up

"Double Black" Concept

x carries an "extra black" to maintain P5. Goal: absorb it (red+extra=black) or push up to root.

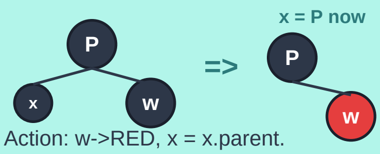
Case 1: Sibling w is RED



Action: w->BLK, P->RED, x under P, new BLACK w
LEFT-ROTATE(P), update w.

Converts to Case 2, 3, or 4.

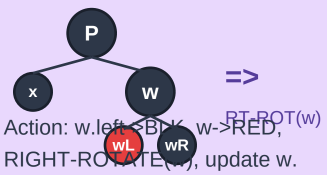
Case 2: w BLK, both kids BLK



Action: w->RED, x = x.parent.
Push double-black up.

Only case that loops.

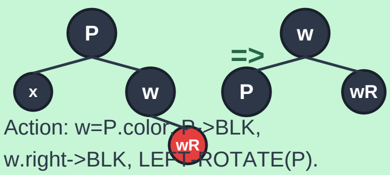
Case 3: w BLK, w.left RED, w.right BLK



Action: w.left->BLK, w->RED,
RIGHT-ROTATE(w), update w.

Transforms into Case 4.

Case 4: w BLK, w.right RED (TERMINAL)



Action: w=P.color, P->BLK,
w.right->BLK, LEFT-ROTATE(P).

DONE! x = root, loop ends.

DELETE-FIXUP Decision Flowchart

START: x has extra-black, x != root
Is sibling w RED? --> YES: Case 1 (rotate, recolor) -> now w is BLACK, re-check
NO: Are BOTH of w's children BLACK?
YES -> Case 2: w->RED, x=parent (push up, loop)
NO -> Is w's FAR child (w.right) BLACK?
YES -> Case 3: rotate w, swap colors -> now it's Case 4
NO -> Case 4: rotate parent, recolor -> DONE!

Delete Fixup Mnemonic

"Red Sibling? -> Both Black? -> Far child? -> DONE!"

C1 -> C2/3/4. C2 loops. C3 -> C4. C4 terminates. At most 3 rotations total.

WORKED EXAMPLE: Step-by-Step Insertion

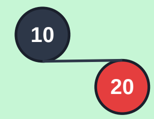
Insert sequence: 10, 20, 30, 15, 25

Step 1: Insert 10

First node -> RED, then recolor BLACK (P2: root must be black).

Step 2: Insert 20

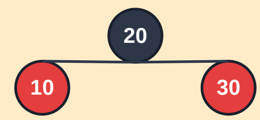
20 > 10, right child. Parent(10) BLACK -> no violation.



Step 3: Insert 30

30 > 20, right of 20. Parent(20) RED, Uncle=NIL(BLK). Case 3: LEFT-ROTATE(10).

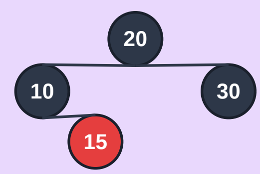
After: 20(B) root, 10(R) left, 30(R) right



Step 4: Insert 15

15: right of 10. Parent(10) RED, Uncle(30) RED -> Case 1: recolor.

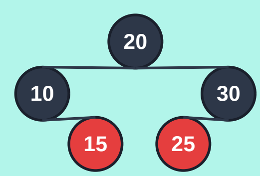
Case 1: P,U->BLK, G->RED->BLK(root)



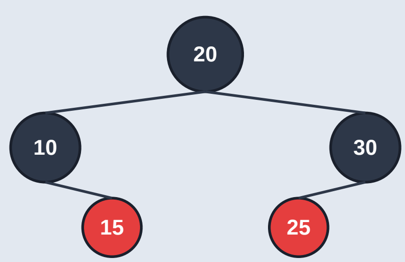
Step 5: Insert 25

25: left of 30. Parent(30) BLACK -> no violation. Just insert RED.

No fixup needed. Parent is BLACK.



Final Red-Black Tree



Verify all 5 properties:

P1: All nodes R or B [OK]
P2: Root(20) is BLACK [OK]
P3: NIL leaves BLACK [OK]
P4: No RED-RED pairs [OK]
P5: bh=2 on all paths [OK]

MASTER CHEATSHEET & EXAM REFERENCE

5 Properties "CoRoL ReB"

P1: Every node is RED or BLACK
P2: Root is BLACK
P3: NIL leaves are BLACK
P4: RED -> both children BLACK
P5: Equal black-height all paths

Rotations O(1)

LEFT-ROTATE: x sinks left, x.right rises
RIGHT-ROTATE: y sinks right, y.left rises
Preserves BST in-order property
Max 5 pointer updates each
"Opposite child rises" mnemonic

INSERT FIXUP (3 Cases)

Insert RED. Only P2/P4 can break.

C1: Uncle RED
Recolor P,U->B, G->R, z=G, loop
C2: Uncle B, zig-zag
Rotate z.parent -> Case 3
C3: Uncle B, zig-zig
Recolor + rotate G. DONE!

Max 2 rotations, O(log n) recolors

DELETE FIXUP (4 Cases)

Only if BLACK removed. x has "extra black".

C1: Sib RED: Rotate+recolor -> C2/3/4

C2: Sib B, both kids B: Sib->R, x=parent, loop

C3: Sib B, near RED, far B: Rotate sib -> Case 4

C4: Sib B, far RED: Rotate parent, recolor. DONE!

Max 3 rotations, O(log n) recolors

Complexity Summary

Operation	Time	Rotations	Recolors	Space
Search	O(log n)	0	0	O(1)
Insert	O(log n)	<= 2	O(log n)	O(1)
Delete	O(log n)	<= 3	O(log n)	O(1)

RB-Tree vs AVL Tree

AVL: stricter balance, faster search (shorter tree)
RB: fewer rotations, faster insert/delete
AVL h <= 1.44 log(n), RB h <= 2 log(n+1)
RB for insert-heavy, AVL for lookup-heavy

All Mnemonics

"CoRoL ReB" = 5 Properties
"Opposite child rises" = Rotations
"Uncle RED? Recolor. BLACK? Rotate."
"Red Sib? Both Black? Far child? DONE"

Exam Tips & Key Takeaways

1. Always verify all 5 properties after every insert/delete step in your answer.
2. Draw NIL nodes explicitly -- marks often depend on showing them correctly.
3. For deletion with 2 children: find in-order successor, copy key, delete successor.
4. Insert breaks P2 or P4 only. Deletion of BLACK node breaks P5 primarily.
5. Black-height = count of black nodes from node to NIL leaf (not including node, per CLRS).
6. Insert fixup: max 2 rotations. Delete fixup: max 3 rotations.