Red-Black Trees

AVL's pragmatic rival: slightly looser balance, much cheaper maintenance — five color rules that quietly run C++ std::map, Java TreeMap, and the Linux kernel.

Dr. Manu Shrivastava
Course Instructor · Consultation Fri 2–5 PM, LHC 308F

40 minutes
Session outcome: explain Red-Black properties and balancing rules

Today, minute by minute

The need: what's wrong with AVL? 00–05

Nothing — unless you insert a lot. The strictness–maintenance trade.

The five rules, and why they force balance 05–12

Colors as bookkeeping; black-height; h ≤ 2 log(n+1) from two rules.

Animation: insertion — recolor first, rotate if you must 12–24

A fully worked example (insert 10, 20, 30, 15, and more); uncle red vs uncle black.

Animation: deletion — the double-black debt 24–34

Deleting red is free; deleting black leaves a hole. The three sibling questions.

AVL vs Red-Black: choosing; recap + homework 34–40

Read-heavy vs write-heavy; where each lives in real systems.

AVL is a perfectionist. Perfection has a price.

THE COMPLAINT AGAINST AVL

AVL's contract — every bf ∈ {−1, 0, +1} — is **strict**. Strictness buys the shortest possible trees ($h \leq 1.44 \log n$, superb for *searching*), but the enforcement is jumpy: modest insertions trigger rotations, and a single deletion can cascade rotations all the way to the root. If your workload **modifies the tree constantly** — an index ingesting records, a scheduler adding and removing tasks — you pay the enforcement cost on every operation.

Question a lazy engineer would ask: **can we relax the balance rule just enough that repairs become rare and cheap, while h stays O(log n)?**

THE RED-BLACK ANSWER

Yes: tolerate height up to **2 log(n+1)** — about 40% taller than AVL in the worst case — and in exchange, most repairs become **recolorings**: O(1) paint jobs with no structural change at all. Rotations still exist but fire far less often (insertion: at most 2; deletion: at most 3).

The selection rule you'll be examined on: frequent insertion/deletion → Red-Black wins; search-dominated, rarely-modified data → AVL wins (it's more tightly balanced). That's why library implementers — who can't predict your workload but must survive the worst — almost universally chose Red-Black: `std::map`, `std::set`, Java `TreeMap`, the Linux kernel's scheduler and memory manager.

The intuition to carry: AVL stores balance as *numbers* (balance factors) and reacts to every ±2. Red-Black stores balance as *one bit per node* — a color — and reacts only when colors clash. Less information, looser guarantee, calmer maintenance. It's the same engineering trade you saw in Unit I: pay a small constant factor to avoid expensive reorganization.

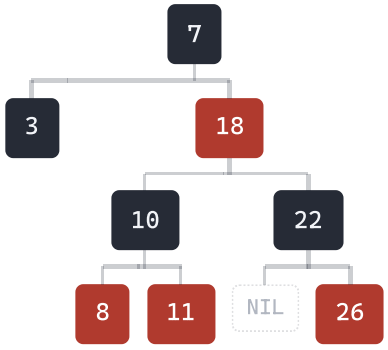
Five rules — two of which do all the work

THE FIVE RULES

- **1.** Every node is **red** or **black**.
- **2.** The **root is black**.
- **3.** Every leaf slot (NIL) counts as **black**.
- **4. A red node's children are black** — no two reds in a row on any path.
- **5.** Every root-to-NIL path contains the **same number of black nodes** (the tree's **black-height**, bh).

Why these force balance: rule 5 says all paths have equal black counts; rule 4 says reds can't stack, so a path can at most *alternate* black-red-black-red… Therefore the longest path (maximal reds) is at most **twice** the shortest (all black): $\mathbf{h \le 2 \cdot bh \le 2\ log_2(n+1)}$. Two local rules, one global guarantee.

A LEGAL RED-BLACK TREE — CHECK EVERY RULE



DARK FILL = BLACK · RED FILL = RED · DOTTED NIL SLOTS COUNT AS BLACK (RULE 3)

Verify rule 5 on any two paths: root→3→NIL crosses blacks {7, 3, NIL} = 3; root→18→10→8→NIL crosses {7, 10, NIL} = 3 (18 and 8 are red, they don't count). Every path: exactly bh = 3. **Red nodes are the slack in the system** — free vertical space that costs no black-height.

THE MENTAL MODEL FOR EVERYTHING THAT FOLLOWS

Think of black nodes as **bricks** (structural, counted, rule 5 protects them) and red nodes as **rubber spacers** (uncounted, flexible, rule 4 just forbids stacking two spacers). **Insertion** always adds a rubber spacer (red) — it can never break the brick count, only bump into another spacer (red-red clash). **Deletion** is dangerous only when it removes a brick — one path suddenly has fewer bricks than the rest. Every algorithm today is just: resolve a spacer clash, or replace a missing brick.

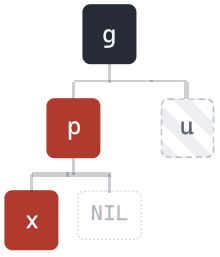
Insertion: paint it red, then ask the uncle

THE ALGORITHM

```
1. BST-insert the key; color it RED
   // red never breaks rule 5 —
   // only possibly rule 4 (red-red)
2. if x is root → paint BLACK, done
3. if parent is BLACK → done (no clash)
4. if parent is RED, look at the UNCLE:
   · uncle RED   → recolor: p → BLACK, u → BLACK,
                            g → RED;
                   x ← g, repeat from 2
   · uncle BLACK → rotate the g–p–x triple
                   (LL/LR/RR/RL, Lecture 8's moves);
                   recolor: new subtree root → BLACK,
                            old grandparent g → RED
                   // same two targets in all four cases —
                   // "new top" black, demoted g red
```

SETUP, PRECISELY

x is freshly inserted RED. If parent p is black, rules 4 and 5 both hold — done. If p is red, rule 4 is violated (g→p→x, two reds in a row). Since rule 4 held *before* this insertion, p's parent g **must be black** — otherwise g-p would already have violated rule 4. So the state is fixed: g black, p red, x red. The uncle u (g's other child) is the only unknown, and **its color alone** decides which fix is valid.



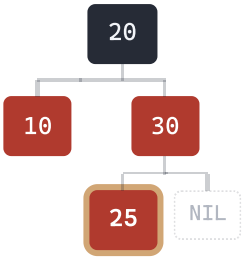
G = GRANDPARENT (FORCED BLACK) · P = PARENT (RED) · X = FRESHLY INSERTED (RED) · U = UNCLE, HATCHED = COLOR UNKNOWN — THE ONE OPEN QUESTION

CASE 1 — UNCLE U IS RED. FIX: RECOLOR P→BLACK, U→BLACK, G→RED

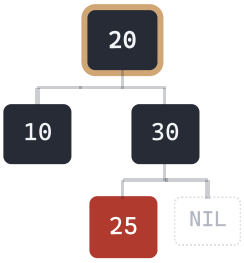
Rule 4 check: x stays red, but its parent p is now black — no violation at x. g is now red; if g's own parent is red, that is the same violation one level up — hence "x ← g, repeat."

Rule 5 check, by direct count: any path through g into p's subtree contributed g(1)+p(0)=1 black before, and g(0)+p(1)=1 after — identical. Any path through g into u's subtree contributed g(1)+u(0)=1 before, g(0)+u(1)=1 after — identical. Every path's black count through this region is unchanged, because u's count rose by exactly 1 while g's fell by exactly 1 — the cancellation only works because u was red beforehand. No rotation needed; the arithmetic already balances.

BEFORE — INSERT 25 CLASHES WITH PARENT 30 (BOTH RED); UNCLE 10 IS RED



AFTER — RECOLOR 30→BLACK, 10→BLACK, 20→RED, FORCED BACK TO BLACK (ROOT, RULE 2)

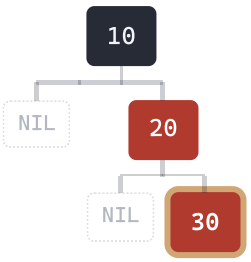


CASE 2 — UNCLE U IS BLACK. CASE 1'S RECOLOR IS PROVABLY INVALID HERE

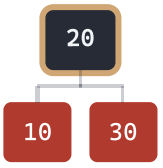
Try the Case 1 recolor anyway (u→black is now a no-op, g→red) and check the u-side path: before, g(1)+u(1)=2; after, g(0)+u(1)=1 — that path **lost** a black unit it had no surplus to give. Rule 5 breaks, because there was no red node on u's side to absorb the change. A pure recolor is asymmetric and invalid whenever u is black.

Fix: rotate, then recolor. Rotate the g–p–x triple (L8's LL/RR/LR/RL moves) so the median of the three keys becomes the new local root; g is demoted to a child. Recolor: new top→black, demoted g→red. Check from *above*: before, a path entering from g's parent crossed g(1)+p(0)/x(0)=1 black across this whole span; after, it crosses new-top(1)+demoted-g(0)=1 — unchanged. g's original parent sees no difference, so the fix is fully absorbed locally — the loop terminates, it never climbs further.

BEFORE — INSERT 30 CLASHES WITH PARENT 20 (BOTH RED); UNCLE = NIL, COUNTS BLACK (RULE 3)



AFTER — ROTATE LEFT AT 10: 20 BECOMES THE NEW TOP (BLACK), 10 AND 30 BOTH RED



WORKED INSTANTIATION — HAND-VERIFIABLE

Insert 10, 20, 30: 10 is root (black). 20 inserted red under 10 (black parent) — no clash. 30 inserted red under 20 (red parent) — clash. Grandparent 10, uncle = 10's other child = NIL, which counts as black (rule 3). **Uncle black → Case 2.** Rotate: 20 becomes new top (black), 10 and 30 become its children (both red). Check rule 5: root(20)→10→NIL = {20, NIL} = 2 blacks (10 red, skipped); root→30→NIL = {20, NIL} = 2 blacks. Equal.

Insert 25: lands as 30's left child, red, parent 30 red — clash. Grandparent 20, uncle = 20's other child = 10, which is red. **Uncle red → Case 1.** Recolor 30→black, 10→black, 20→red; since 20 is root, rule 2 forces it back to black. Check rule 5: every root-to-NIL path now crosses exactly 3 blacks (20, then 10 or 30, then NIL) — confirmed on all four paths.

INTERACTIVE — INSERT 10, 20, 30, 15, THEN 5 AND 12

STEP 1 OF 11

RED-BLACK TREE (AMBER RING = THE NODES IN PLAY THIS STEP)

Ground rule: every new key enters **RED**. Why? A red node adds zero to any path's black count — rule 5 (the brick count) is untouchable. The only thing that can go wrong is a red-red clash with the parent — a local, fixable problem. Insert **10**...

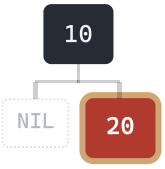
STEP 2 OF 11

RED-BLACK TREE (AMBER RING = THE NODES IN PLAY THIS STEP)

Insert 10: it's the root → rule 2 says paint it BLACK. (Whole tree's black-height becomes 1.)

STEP 3 OF 11

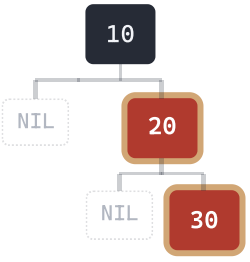
RED-BLACK TREE (AMBER RING = THE NODES IN PLAY THIS STEP)



Insert 20 (red): parent 10 is BLACK — no clash, **nothing to fix, done.** This is the everyday case, and it costs nothing. Note the dotted NIL slots: each counts as a black leaf (rule 3).

STEP 4 OF 11

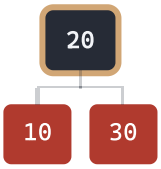
RED-BLACK TREE (AMBER RING = THE NODES IN PLAY THIS STEP)



Insert 30 (red): parent 20 is RED — **red-red clash** (rule 4). Ask the uncle: 30's uncle is 10's left child = NIL = **black**. Black uncle → rotation. Shape g→p→x = 10→20→30 = right-right: **RR case** — rotate so the median of the trio rises.

STEP 5 OF 11

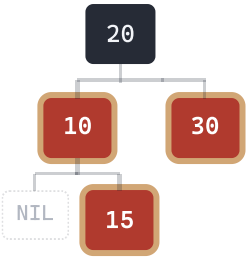
RED-BLACK TREE (AMBER RING = THE NODES IN PLAY THIS STEP)



Left-rotate g (10), then swap colors of g and p: 20 takes the top wearing black; 10 drops down wearing red. Check rule 5: every path root→NIL crosses exactly 2 blacks (20 + NIL) — balanced books. One rotation, clash gone, and this fix **never propagates**.

STEP 6 OF 11

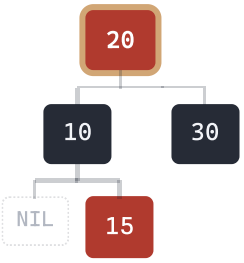
RED-BLACK TREE (AMBER RING = THE NODES IN PLAY THIS STEP)



Insert 15 (red): goes right of 10. Parent 10 is RED — clash. Ask the uncle: 15's uncle is **30, which is RED**. Red uncle → **pure recoloring**: paint parent (10) and uncle (30) BLACK, paint grandparent (20) RED, and move the problem pointer up to 20.

STEP 7 OF 11

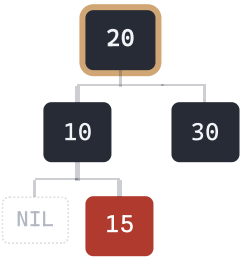
RED-BLACK TREE (AMBER RING = THE NODES IN PLAY THIS STEP)



After the repaint: the clash at 10–15 is gone (10 is black now), but 20 turned red — is *it* in trouble? Check: 20 is the **root** → just paint it black. No rotation anywhere in this whole fix — three paint strokes total.

STEP 8 OF 11

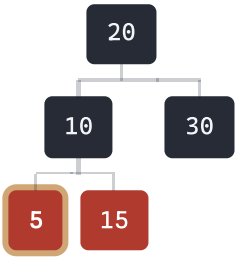
RED-BLACK TREE (AMBER RING = THE NODES IN PLAY THIS STEP)



Example complete: 10, 20, 30, 15 inserted — one rotation, one recoloring wave. Every path: 2 blacks + NIL = bh 3... verify it on any path. Two more insertions to see the patterns repeat →

STEP 9 OF 11

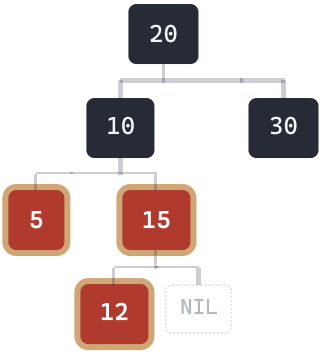
RED-BLACK TREE (AMBER RING = THE NODES IN PLAY THIS STEP)



Insert 5 (red): parent 10 is BLACK → **done instantly, zero repairs.** In a busy Red-Black tree roughly half of all insertions end exactly like this — the statistical reason RB beats AVL under heavy writes.

STEP 10 OF 11

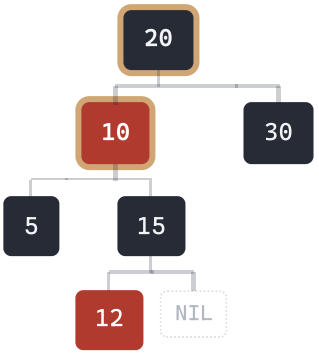
RED-BLACK TREE (AMBER RING = THE NODES IN PLAY THIS STEP)



Insert 12 (red): under 15 (left). Parent 15 is RED — clash. Uncle of 12 is **5, RED** → recolor: 15 and 5 go BLACK, grandparent 10 goes RED, problem climbs to 10.

STEP 11 OF 11

RED-BLACK TREE (AMBER RING = THE NODES IN PLAY THIS STEP)



Check 10 (now red): its parent 20 is **BLACK** — no clash, the climb **stops here**. Final tree: legal on all five rules. Scoreboard for 6 insertions: **1 rotation, 2 recoloring waves, 2 free passes** — mostly paint, hardly any surgery. That is Red-Black's whole personality.

Deletion: the double-black debt, and how to pay it

TWO DEFINITIONS, STATED PRECISELY — EVERYTHING BELOW IS JUST THESE TWO CASES IN DETAIL

After BST-delete physically unlinks node y, look at y's color and the color of whatever replaces it in that spot (its one child, or NIL if y was a leaf — NIL counts as black, rule 3):

Case 1 — free, no repair. If y was RED, or its replacement is RED: (re)paint the replacement BLACK if it wasn't already, and stop. Rule 5 is untouched — a red node never counted toward any path's black total in the first place, so removing it or absorbing it changes nothing.

Case 2 — double-black (DB), repair required. If y was BLACK and its replacement is also BLACK (including a NIL replacement): that spot is marked **double black** — a bookkeeping label, not a real color, meaning "this position must count as 2 blacks until the tree is fixed, because the path through here is one black short of every other path." The three sibling questions below are exactly the procedure for discharging that debt.

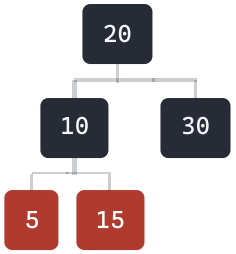
FROM LECTURE 7'S BST-DELETE TO HERE

L7 gave three cases: delete a leaf (unlink it); delete a one-child node (parent adopts the child); delete a two-children node by **copying the in-order successor's key** up, then deleting the successor from its original spot — which is itself always a leaf or one-child node. So **every** RB deletion, no matter which case it starts as, bottoms out at physically unlinking one leaf-or-one-child node, y.

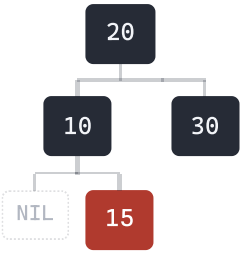
The subtlety that matters here: in the two-children case, the key that disappears is the one you asked to delete — but the **node** that gets physically unlinked, and therefore the **color** that decides how hard the fix is, belongs to the *successor*, which can be a completely different color from the key you targeted. Always check the color of y, the node actually removed — never the color of the key you started with.

WHY Y'S COLOR IS EVERYTHING — ON ONE SMALL TREE

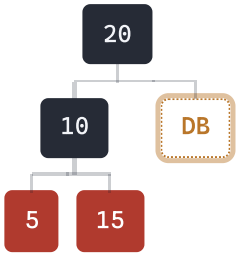
BEFORE ANY DELETION — EVERY PATH CROSSES 3 BLACKS



AFTER DELETING 5 (RED LEAF) — STILL 3, NO CHANGE NEEDED



AFTER DELETING 30 (BLACK LEAF) — THE EMPTY SLOT OWES ONE BLACK



Delete 5 (RED leaf): unlink it, 10's left becomes NIL directly. Path 20→10→NIL: blacks = {20, 10, NIL} = 3 — unchanged (5 was red, it was never counted). **Free**, exactly as claimed.

Delete 30 (BLACK leaf): unlink it, 20's right becomes NIL directly. Path 20→NIL: blacks = {20, NIL} = **2** — down from 3, while every other path is still 3. Rule 5 is broken by exactly one black, on exactly this one path. That empty NIL slot is marked **double black (DB)**: a bookkeeping label meaning "this position must count as 2 blacks until the tree is repaired" — not a real color, just a flag tracking the one-black debt while the sibling-based repair (next card) pays it off.

THE TWO-CHILDREN CASE, TRACED — WHERE THE SUBTLETY BITES

Ask to delete **10** (black, two children) from the same starting tree. Successor = leftmost key in 10's right subtree = **15** itself. Copy 15's key up into node 10's position (that node keeps its own identity and color — still black — only its key changes), then delete the *original* 15-node, which is a red leaf. That's a Case-1 free deletion: node 20→[key 15, still black]→5→NIL gives blacks {20, node, NIL} = 3; the now-empty right side gives {20, node, NIL} = 3. Nothing broke — even though the key you asked to delete, 10, was black. **The color that decided the outcome was the successor's (red), not the target's.**

THE THREE SIBLING QUESTIONS — DISCHARGING A DOUBLE-BLACK DEBT

Insertion asked the **uncle**; deletion asks the **sibling** of the DB node — a good exam trap.

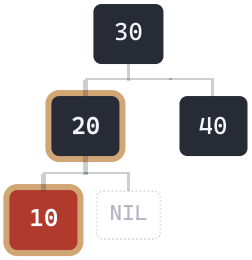
- **Sibling black, with a red child?** Two shapes, decided by *which* child is red:
  - **Far child red** (the one on the side away from the hole) — regardless of the near child, even if it's red too: rotate the far child straight up through the parent. Recolor **three** nodes: new subtree top (the sibling) ← old parent's color; old parent ← BLACK; the rotated red child ← BLACK.
  - **Only the near child red** (far child black): first rotate that near child up through the sibling and swap those two nodes' colors — this converts the shape into "far child red" above. Then apply that fix.

  Either way: debt paid structurally, O(1), at most 2 rotations. (Four sub-shapes: LL/LR/RR/RL, as ever.)
- **Sibling black, both children black?** No red to borrow — repaint the sibling RED (subtracting one black from *both* sides equalizes them) and push the DB up to the parent — three possible outcomes: **parent is red** → paint it black, done; **parent is black and is the root** → absorb, done, bh drops by 1; **parent is black and is not the root** → the parent itself is now the double-black node — repeat these same three questions one level higher, using the parent's own sibling. This is the only case that can climb, and it can climb repeatedly, bounded only by the tree's height.
- **Sibling red?** Rotate the sibling up and recolor — this converts to one of the black-sibling cases above.

INTERACTIVE — FIVE DELETION SCENARIOS, WORKED END TO END

STEP 1 OF 14

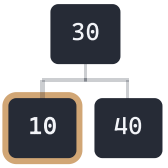
RED-BLACK TREE



Scenario 1 — Case 1 (free). Delete **20** (one child: red 10). BST-delete splices 10 up into 20's place. y = 20 was black, but its replacement 10 is red — Case 1 applies: just **paint the replacement black** — the path's black count is unchanged.

STEP 2 OF 14

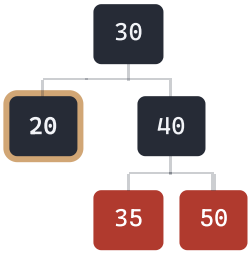
RED-BLACK TREE



Done, O(1): 10 sits where 20 was, painted black. Every path still crosses 2 blacks. Deleting red — or absorbing into red — never disturbs rule 5. Now Case 2, the debt case…

STEP 3 OF 14

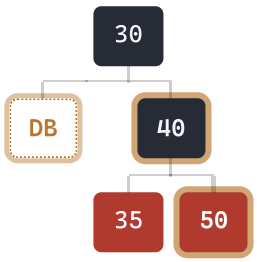
RED-BLACK TREE



Scenario 2 — Case 2 (double-black). Delete **20**: it's black with no children; its replacement is NIL, also black. Both black — Case 2 applies. The left path just **lost a black node** — rule 5 is broken. Mark the hole **double-black (DB)**: a debt of one black, owed by that path.

STEP 4 OF 14

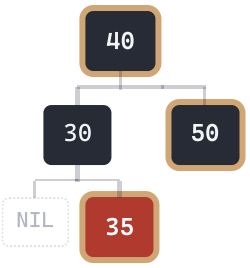
RED-BLACK TREE



Ask the sibling: s = 40 is **black and has a red child** (50, on the far side: RR shape). A red child is **spare paint that can become a brick**: left-rotate the parent 30, pull 40 up, and paint 50 black.

STEP 5 OF 14

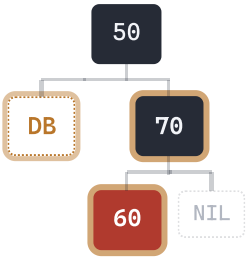
RED-BLACK TREE



Debt paid structurally. The general rule recolors *three* targets: new top ← old parent's color, old parent ← BLACK, the rotated red child ← BLACK. Here two are invisible — 40 (new top) and 30 (old parent) were **already** black, so the assignment changes nothing you can see. Only **50** actually flips color. (You'd see all three light up in a case where the parent started red.) Check rule 5: every path crosses 2 blacks + NIL ✓. One rotation, O(1), done — this is the deletion analogue of the black-uncle insertion case.

STEP 6 OF 14

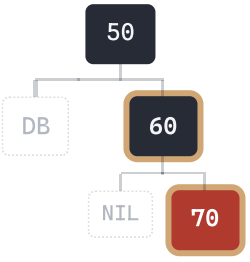
RED-BLACK TREE



Scenario 2b — the same case, but only the NEAR child is red. DB on 50's left; sibling 70 is black, but this time only its **near** (left) child 60 is red — the far (right) side is NIL, black. Rotating straight through the parent now would leave the wrong shape (try it mentally: 50 needs to inherit a *far* red child, and it doesn't have one yet).

STEP 7 OF 14

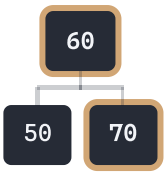
RED-BLACK TREE



Preliminary step — rotate at the sibling itself, not the parent. Right-rotate 70, and swap colors: 60 (the near child) rises to become the new sibling, painted **BLACK**; old sibling 70 drops down, painted **RED**, now sitting on the far side. Nothing is fixed yet — the debt is still owed. This purely repositions the red into the far-child shape you already know how to close.

STEP 8 OF 14

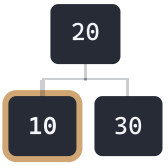
RED-BLACK TREE



Now apply the far-red fix, exactly as in Scenario 2. Left-rotate parent 50; recolor: new top 60 ← old parent's color (black — no-op again), old parent 50 ← BLACK (no-op), far red child 70 ← BLACK. Debt paid — but this sub-case cost **2 rotations** instead of 1, exactly the extra preliminary step the near-child shape demands.

STEP 9 OF 14

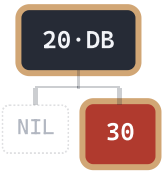
RED-BLACK TREE



Scenario 3 — nobody has spare red. Delete **10** (black leaf) from this all-black tree: DB appears on the left; sibling 30 is black with two black (NIL) children. Nothing to rotate in.

STEP 10 OF 14

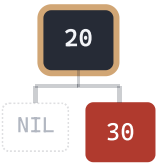
RED-BLACK TREE



Repaint sibling 30→RED; push the DB up to the parent 20. Take one black away from BOTH sides: 20's own two children are locally balanced again (left: just NIL = 1; right: red 30 + NIL = 1). But 20 itself, seen from its own parent, now contributes one fewer black than before — the debt hasn't vanished, it has moved up one level and now sits *on node 20*.

STEP 11 OF 14

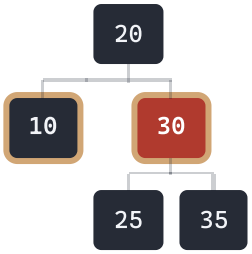
RED-BLACK TREE



Check where the debt landed: 20 is the ROOT. There's no parent above to push it to, so the tree simply **absorbs** it here — the whole tree's black-height permanently drops by 1, and we stop. Had 20 been red instead, painting it black on the spot would have settled the debt without needing the root at all. Had 20 been black and *not* the root, we'd repeat these same three sibling questions one level higher — exactly what the cascading example below shows happening twice.

STEP 12 OF 14

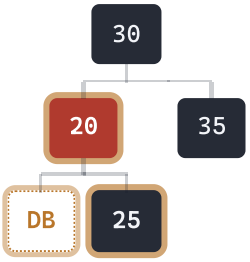
RED-BLACK TREE



Scenario 4 — the sibling itself is red. Delete **10**: DB on the left, and sibling 30 is **RED**. A red sibling tells us nothing about spare bricks — so first **rotate the sibling up** (left-rotate parent 20; swap colors of 20 and 30)…

STEP 13 OF 14

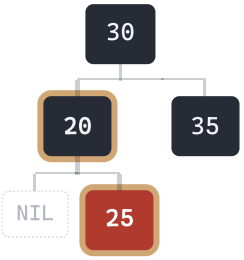
RED-BLACK TREE



…which converts it to a black-sibling case: DB's sibling is now 25 (black, black children) → Scenario-3 fix: paint 25 red, push DB to parent 20 — and **20 is red**: paint it black, debt settled.

STEP 14 OF 14

RED-BLACK TREE



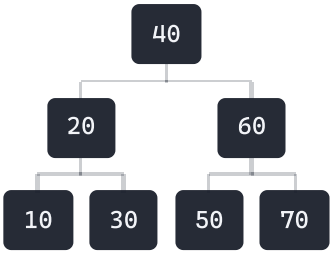
Final tree — all rules hold (check rule 5: two blacks + NIL on every path). The full deletion algorithm is just these moves composed: **red? free · black sibling with red child? rotate & repaint · black sibling, all black? push the debt up · red sibling? rotate first.** At most 3 rotations, O(log n) total — the calm, bounded repair bill that made Red-Black the industry default.

INTERACTIVE — WHEN THE PARENT ISN'T THE ROOT: THE CASCADE ACTUALLY CLIMBS

Scenario 3 above resolved in a single push only because that parent happened to already be the root. Add one more level — root 40(B), children 20(B) and 60(B), each with two black leaf children — and watch the identical debt genuinely climb **twice** before it's settled.

STEP 1 OF 6

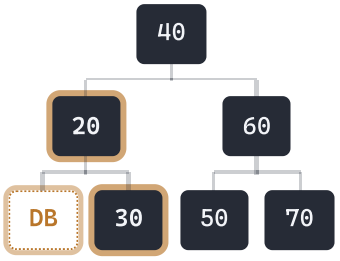
RED-BLACK TREE



Setup: root 40(B), children 20(B) and 60(B), each with two black leaf children. Every root-to-NIL path crosses {40, 20-or-60, child, NIL} = **4** blacks — check all four.

STEP 2 OF 6

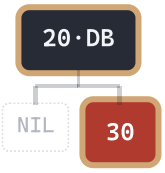
RED-BLACK TREE



Delete 10 (black leaf). Replacement is NIL, also black — Case 2. DB appears at 20's left. Diagnose the sibling: 30 is black, and both its children are black (NIL) — matches "sibling black, both children black."

STEP 3 OF 6

RED-BLACK TREE

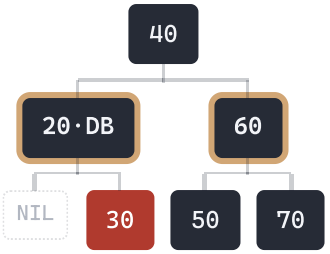


ZOOMED IN ON THE LEFT SUBTREE

Repaint 30→red; push the DB up to 20. 20's own two children are now locally balanced (left: just NIL = 1; right: red 30 + NIL = 1). But 20 itself, seen from ITS parent 40, now contributes one fewer black than before. **20 is black and is NOT the root** — so we do not stop here.

STEP 4 OF 6

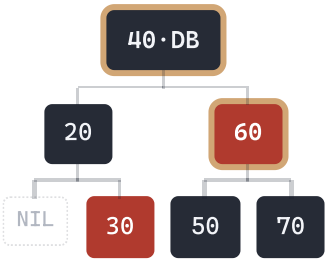
RED-BLACK TREE



Repeat the same three questions, one level up. The DB is now on node 20; from 40's perspective, 20's sibling is 60 — black, with both children (50, 70) black. The identical case applies again.

STEP 5 OF 6

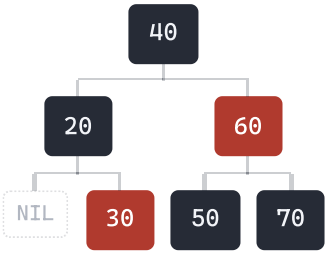
RED-BLACK TREE



Repaint 60→red; push the DB up to 40. 40 is now one black short, seen from further above — but **40 is the root.** Absorb the debt right here: stop. The tree's overall black-height permanently drops by 1.

STEP 6 OF 6

RED-BLACK TREE



Final tree, verified: 40→20→NIL = {40,20,NIL} = 3. 40→20→30→NIL = {40,20,NIL} = 3 (30 red, skipped). 40→60→50→NIL = {40,50,NIL} = 3 (60 red, skipped). 40→60→70→NIL = {40,70,NIL} = 3. All four paths agree — down from 4, but consistent, which is all rule 5 ever requires. Two levels climbed, zero rotations: pure recoloring, bounded only by the tree's height.

Complexity: each case does O(1) recoloring/rotation; only the "push DB up" case climbs, so deletion is **O(log n)** with at most 3 rotations — versus AVL's possible O(log n) rotation cascade. This is precisely where Red-Black earns its keep in write-heavy systems.

AVL vs Red-Black: an engineering choice, not a ranking

	AVL (LECTURE 8)	RED-BLACK (TODAY)
Balance rule	every bf ∈ {−1, 0, +1}	five color rules
Height bound	`1.44 log n — shorter`	`2 log(n+1)`
Search speed	**faster** (shorter tree)	slightly slower
Insert repair	≤ 1 rotation	often **just recoloring**; ≤ 2 rotations
Delete repair	up to O(log n) rotations	**≤ 3 rotations**
Best for	read-heavy, rarely modified	write-heavy, constantly modified
Lives inside	lookup tables, some databases' in-memory indexes	std::map / std::set, Java TreeMap/TreeSet, Linux CFS scheduler, epoll, nginx timers

And the third contender: when the tree lives on *disk*, neither wins — binary nodes are too small for 4 KB blocks. That problem demands nodes with hundreds of keys: the **B-Tree**, Lecture 11 — Unit I's disk lessons and Unit II's tree lessons finally meet.

Three takeaways

- **Two rules carry everything:** no red-red (rule 4) + equal black counts (rule 5) ⟹ longest path ≤ 2× shortest ⟹ h ≤ 2 log(n+1). Bricks and rubber spacers.
- **Insertion:** paint red, then ask the **uncle** — red uncle: recolor and climb; black uncle: one Lecture-8 rotation + color swap. Most fixes are paint, not surgery.
- **Deletion:** red is free; black leaves a **double-black** debt — ask the **sibling**: borrow a red child, or repaint and push up, or rotate a red sibling into position. ≤ 3 rotations, O(log n).

LECTURE 10

Splay Trees

No colors, no factors, no guarantees per operation — just "move what you touch to the root," and the Lecture 3 potential method finally earns its keep: O(log n) amortized.

HOMEWORK — BRING TO LECTURE 10

- Insert `7, 3, 18, 10, 22, 8, 11, 26` into an empty RB tree; show every recolor/rotation. (It should reproduce the rules slide's example tree.)
- Continue today's animation: insert **13**, then **6** into the final tree — name each case you hit.
- From the final insertion tree, delete **30**, then **20**: identify each deletion case.
- Give a coloring of a 7-node chain proving no valid RB coloring exists (argue via rules 4 and 5).
- Compute bh(x) for every node of the rules slide's tree; verify h ≤ 2·bh.
- One paragraph: your app does 95% lookups on a phonebook loaded once at startup — AVL or Red-Black, and why?
- Reading: CLRS ch. 13 (Red-Black Trees).

Questions?

Dr. Manu Shrivastava — LHC 308F — Friday 2:00–5:00 PM

> *Next: Lecture 10 — Splay Trees and Self-Adjusting Trees, where Lecture 3's potential method pays off.*