CSE3144 · Advanced Data Structures · Jul–Nov Semester 2026 · Lectures 7–8 of 36 · CO CSE3144.2 · Unit II begins

# Binary Search Trees & AVL Trees, in Depth

Lecture 7: the dictionary problem, the BST that solves it, every operation animated — and the crack in the design. Lecture 8: the repair kit — rotations, AVL insertion and deletion, all animated.

---

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

**40 minutes**
Session outcome: explain BST operations and AVL balance properties

# Today, minute by minute

- **The dictionary problem — why arrays and lists both fail** 00–05
  Search fast OR update fast; we need both. The tree is binary search, frozen.
- **What a BST is, precisely** 05–10
  The BST property, node anatomy (key, left, right, parent), inorder = sorted.
- **Animation: building a BST, key by key** 10–17
  Seven insertions; every walk, every comparison, every parent link.
- **Animation: search — a hit and a miss** 17–21
  Find 40 in 3 comparisons; fall off the tree looking for 65.
- **Animation: deletion — all three cases** 21–29
  Leaf, one child, two children (the successor trick).
- **Animation: the betrayal — sorted input** 29–33
  Lecture 2's employee IDs return; the tree becomes a linked list.
- **AVL trees: balance, guaranteed** 33–38
  Balance factors, the height promise, and the cliffhanger: rotations.
- **Recap + homework** 38–40
  O(h) is the whole story — control h, control everything.

# Lecture 8 — the repair kit (continues below)

- **Rotations: the one legal move** 00–08
  Single rotations (LL, RR), animated — and why they preserve the BST property.
- **Double rotations (LR, RL)** 08–16
  Watch a single rotation FAIL on a zig-zag, then fix it in two moves.
- **AVL insertion, end to end** 16–27
  Six keys, balance factors live, two violations repaired on the way.
- **AVL deletion** 27–35
  Delete, walk up, rebalance — and why deletion can cascade.

## Recap + homework 35–40

The four-case cheat sheet; the median always wins.

# The dictionary problem: three operations, no compromises

A **dictionary** stores keyed records and must support three operations, repeatedly and in any order: **search(k)**, **insert(k)**, **delete(k)**. Phone contacts, student records by roll number, a compiler's symbol table, your browser history — all dictionaries. You already own two structures; watch them each fail one requirement:

| STRUCTURE | SEARCH | INSERT / DELETE | WHERE IT HURTS |
|---|---|---|---|
| **Sorted array** | O(log n) — binary search ✓ | O(n) ✗ | Inserting into the middle shifts everything after it. 1 lakh contacts → 1 lakh shifts. |
| **Linked list** | O(n) ✗ | O(1) at a known spot ✓ | But finding that spot costs O(n) — the pointer must crawl. |
| **What we want** | **O(log n)** | **O(log n)** | Fast everything, on data that keeps changing. |

THE UNIFYING INTUITION

## A BST is binary search, frozen into pointers

Binary search works because each comparison discards half the array. But the array's rigidity ruins updates. So take the *decision pattern* of binary search — "compare with the middle, go left or right" — and store it as a structure: the middle becomes the **root**, the two halves become **subtrees**, recursively. Searching walks the same path binary search would compute — but now inserting is just **hanging one new node**, no shifting.

WHY NOT THE FASTEST THING EVER?

**Direct addressing** — an array indexed by the key itself — gives O(1) everything. The price: an array as large as the key *universe*. Roll numbers like 229301234 would need an array of ~$10^9$ slots to store 60 students. The BST spends memory on **what you actually store**, not on what you might store; hashing (Unit II, later) is the other answer. Trees additionally keep keys **in order** — ranges, nearest-key, sorted listing — which hashing never will.

# The BST property — one rule, recursively everywhere

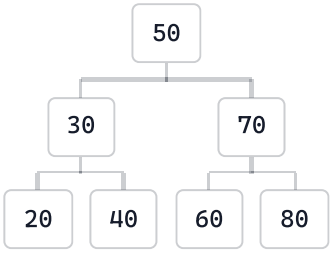
ANATOMY + THE INVARIANT

Each node carries: `key` · `left` (pointer to left child) · `right` · `p` (pointer to **parent** — the upward link; p[root] = NIL). Every child is linked to its parent from both directions.

> For **every** node x: all keys in x's **left** subtree ≤ key[x] ≤ all keys in x's **right** subtree

**Read the fine print:** the rule binds *entire subtrees*, not just children. In the tree alongside, 40 < 50 even though 40 is two levels down on 50's left. This global rule is exactly what makes every operation a single root-to-leaf walk — at each node, one comparison eliminates one whole subtree, just like binary search eliminates half the array.

OUR RUNNING EXAMPLE — 7 KEYS




EVERY NODE: LEFT SUBTREE ≤ NODE ≤ RIGHT SUBTREE · DOTTED = NIL (NONE HERE: TREE IS PERFECT)

**Free bonus — inorder traversal** (left subtree, node, right subtree) visits: `20, 30, 40, 50, 60, 70, 80` — **sorted, automatically**. A BST is a sorting of the keys, stored as geometry. This is also your instant correctness check for every exercise: if inorder isn't sorted, the tree is wrong.

**The one number that rules everything:** every operation we build today walks one path from the root downward, so every operation costs **O(h)**, where h is the tree's height. The entire drama of Unit II — AVL, Red-Black, B-Trees — is a fight to keep h small. Remember "O(h)" and today's lecture derives itself.

# Building the tree, one key at a time

TREE-INSERT, SIMPLIFIED

```
walk from the root, remembering the parent y:
    if new key < current → go left
    else                 → go right
when you step onto an EMPTY slot (NIL):
    place the new node there
    p[new] ← y            // child → parent link
    left[y] or right[y] ← new   // parent → child link
```

Two pointers set, in both directions — the new node is **stitched in**, never inserted "between" anything. No shifting, ever. Cost: the walk, O(h).

HOW TO READ THE ANIMATION

Keys arrive in this order: `50, 30, 70, 20, 40, 60, 80`. At each step the **amber** nodes are the comparison path walked, the **teal** node is the newly linked child — watch its connector tick attach to the parent — and **dotted boxes are empty slots (NIL)**: real places where a future key can land.

INTERACTIVE — SEVEN INSERTIONS, EVERY COMPARISON SHOWN

STEP 1 OF 8

THE TREE SO FAR (DOTTED = EMPTY NIL SLOTS)


**Start:** the dictionary is empty — the root slot is NIL. First key arriving: **50**.

---

STEP 2 OF 8

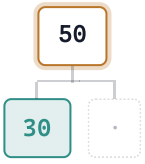
THE TREE SO FAR (DOTTED = EMPTY NIL SLOTS)


**Insert 50:** tree empty → 50 becomes the root. 0 comparisons. (Both its child slots are NIL, ready for future keys.)

---

STEP 3 OF 8

THE TREE SO FAR (DOTTED = EMPTY NIL SLOTS)

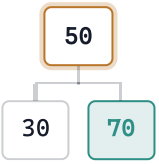



**Insert 30:** compare at 50 — 30 < 50 → go LEFT. Left slot is empty → link: left[50] ← 30, p[30] ← 50. Watch the tick connecting 30 up to 50: that is the parent link, drawn. 1 comparison.

---

STEP 4 OF 8

THE TREE SO FAR (DOTTED = EMPTY NIL SLOTS)

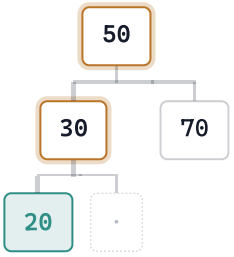



**Insert 70:** 70 > 50 → go RIGHT → empty → right[50] ← 70, p[70] ← 50. The tree is filling level by level — but only because this arrival order is friendly. 1 comparison.

---

STEP 5 OF 8

THE TREE SO FAR (DOTTED = EMPTY NIL SLOTS)

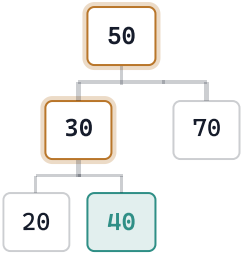



**Insert 20:** 20 < 50 → left; at 30: 20 < 30 → left → empty → left[30] ← 20, p[20] ← 30. 2 comparisons — exactly the depth where it landed.

---

STEP 6 OF 8

THE TREE SO FAR (DOTTED = EMPTY NIL SLOTS)

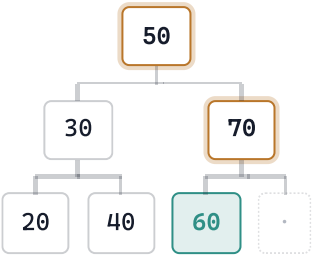



**Insert 40:** 40 < 50 → left; 40 > 30 → right → empty → right[30] ← 40. Note 40 obeys BOTH ancestors: < 50 and > 30. The invariant holds globally, by construction.

---

STEP 7 OF 8

THE TREE SO FAR (DOTTED = EMPTY NIL SLOTS)

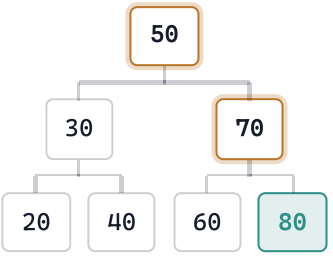



**Insert 60:** 60 > 50 → right; 60 < 70 → left → empty → left[70] ← 60.

---

STEP 8 OF 8

THE TREE SO FAR (DOTTED = EMPTY NIL SLOTS)




**Insert 80:** 80 > 50 → right; 80 > 70 → right → done. **Seven keys, height 2, no slot conflicts, nothing ever shifted.** Inorder check: 20 30 40 50 60 70 80 — sorted ✓. Keep this tree in mind; every remaining animation runs on it.

# Search: the same walk — a hit and a miss

TREE-SEARCH

```
Tree-Search(x, k):
    if x = NIL or k = key[x]: return x
    if k < key[x]: return Tree-Search(left[x], k)
    else:          return Tree-Search(right[x], k)
```

Each comparison discards an entire subtree — never to be looked at again. Running time: **O(h)**, one node per level.

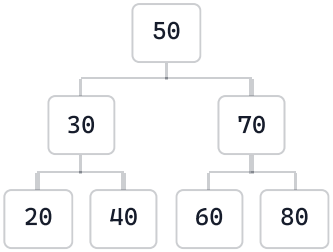
THE INSIGHT THE MISS TEACHES

Searching for an absent key doesn't fail randomly — it walks to **exactly the empty slot where that key would be inserted**. Search and insert are the *same walk* with different endings. (That's why insert was so easy.)

INTERACTIVE – SEARCH 40 (PRESENT), THEN 65 (ABSENT)

STEP 1 OF 8

SEARCHING IN THE 7-KEY TREE

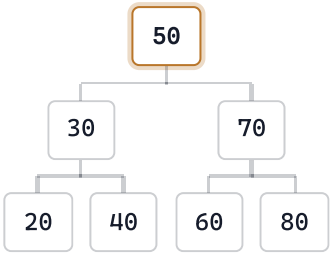



**Two searches:** first **40** (present), then **65** (absent). Every step is one comparison, one level down.

---

STEP 2 OF 8

SEARCHING IN THE 7-KEY TREE

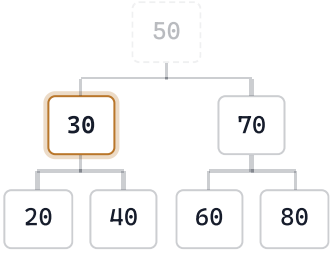



**Search 40 — step 1:** at root 50: 40 < 50 → go left. The entire right subtree {60, 70, 80} is **eliminated without being looked at**.

---

STEP 3 OF 8

SEARCHING IN THE 7-KEY TREE

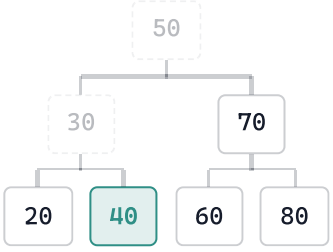



**Step 2:** at 30: 40 > 30 → go right. Subtree {20} eliminated.

---

STEP 4 OF 8

SEARCHING IN THE 7-KEY TREE

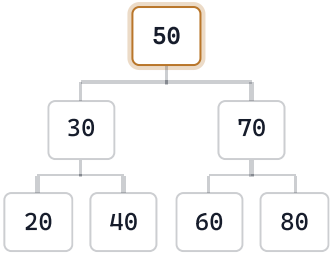



**Step 3:** at 40: match — **found in 3 comparisons** (= depth + 1). In a million-key balanced tree this walk is ~20 steps.

---

STEP 5 OF 8

SEARCHING IN THE 7-KEY TREE

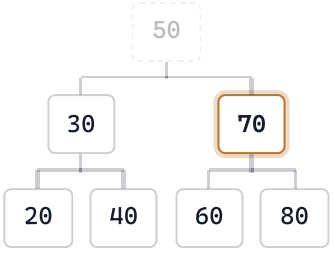



**Search 65 — step 1:** at 50: 65 > 50 → go right. (Left subtree {20, 30, 40} gone.)

---

STEP 6 OF 8

SEARCHING IN THE 7-KEY TREE

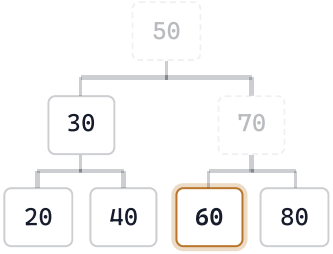



**Step 2:** at 70: 65 < 70 → go left.

---

STEP 7 OF 8

SEARCHING IN THE 7-KEY TREE

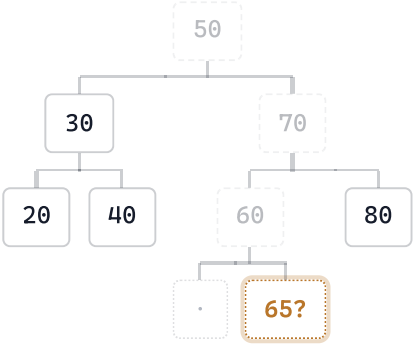



**Step 3:** at 60: 65 > 60 → go right…

---

STEP 8 OF 8

SEARCHING IN THE 7-KEY TREE




**Step 4:** 60's right slot is **NIL — not found**, after 3 comparisons. But look where we stopped: this dotted slot is *exactly* where insert(65) would link a new node. **Search and insert are the same walk with different endings.**

# Deletion: three cases, and a beautiful trick for the hard one

CASE 0 — LEAF

No children: just unlink it from its parent. Nothing else moves.

CASE 1 — ONE CHILD

Splice it out: the parent adopts the only child directly (p[child] ← p[x]). The chain closes over the gap.

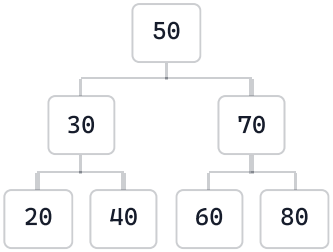
CASE 2 — TWO CHILDREN

Can't unlink — two orphaned subtrees! Trick: overwrite x's key with its **inorder successor** (the smallest key in x's right subtree), then delete the successor — which never has a left child, so it's always Case 0 or 1.

INTERACTIVE — DELETE 20 (LEAF), THEN 30 (ONE CHILD), THEN 50 (TWO CHILDREN)

STEP 1 OF 9

DELETING FROM THE 7-KEY TREE

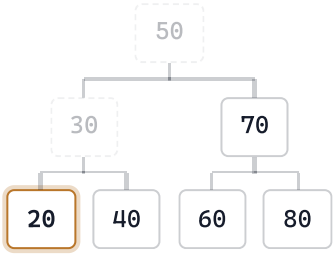



**Plan:** delete **20** (a leaf — Case 0), then **30** (one child — Case 1), then **50** (two children — Case 2). Watch the tree stay a valid BST throughout.

---

STEP 2 OF 9

DELETING FROM THE 7-KEY TREE

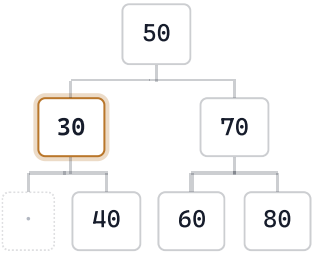



**Delete 20:** walk to it (20 < 50 left, 20 < 30 left — found). It has **no children** → Case 0.

---

STEP 3 OF 9

DELETING FROM THE 7-KEY TREE

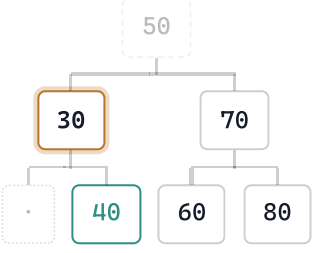



**Case 0 done:** one pointer write — left[30] ← NIL — and 20 is gone. The dotted slot under 30 shows the vacancy. Nothing else in the tree moved.

---

STEP 4 OF 9

DELETING FROM THE 7-KEY TREE

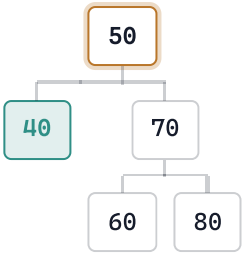



**Delete 30:** walk to it. It now has exactly **one child** (40) → Case 1: splice — its parent must adopt its child.

---

STEP 5 OF 9

DELETING FROM THE 7-KEY TREE

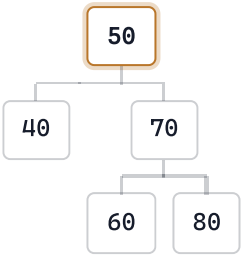



**Case 1 done:** left[50] ← 40 and p[40] ← 50 — grandparent and grandchild linked directly; 30 is unlinked from both directions. BST property intact: 40 is still < 50.

---

STEP 6 OF 9

DELETING FROM THE 7-KEY TREE

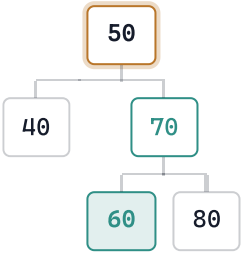



**Delete 50 — the root, with TWO children.** We cannot unlink it: 40 and the whole {60,70,80} subtree would be orphaned. Instead: find its **inorder successor** — the smallest key in the right subtree.

---

STEP 7 OF 9

DELETING FROM THE 7-KEY TREE

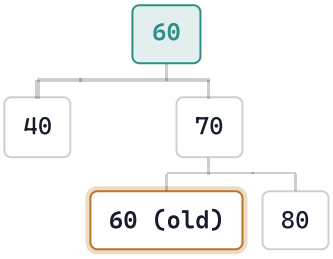



**Finding the successor:** step right once (to 70), then left as far as possible (to 60, whose left is NIL). **60 is the smallest key greater than 50** — the very next key in sorted order.

---

STEP 8 OF 9

DELETING FROM THE 7-KEY TREE

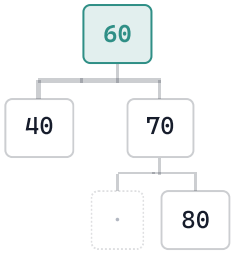



**The trick:** copy 60's key into the root — sorted order is preserved, because 60 is 50's immediate neighbour in the ordering. Now the problem shrinks: delete the *old* 60… which is a leaf. Case 2 always reduces to Case 0 or 1, because a successor can never have a left child.

---

STEP 9 OF 9

DELETING FROM THE 7-KEY TREE




**Done.** Final tree: root 60, children 40 and 70, leaf 80. Inorder: 40, 60, 70, 80 — sorted ✓. Every deletion was one walk plus O(1) pointer surgery: **O(h)**, like everything else today.

**Why the successor?** It is the key that sits immediately after x in sorted order — so promoting it preserves the BST property with zero other changes. (The inorder *predecessor* — largest in the left subtree — works symmetrically.) All three cases: one walk + O(1) pointer surgery = **O(h)**.

# Feed it sorted keys, and the tree stops being a tree

LECTURE 2'S CASE, REPLAYED FOR REAL

Everything today cost O(h) — and we quietly hoped h ≈ log n. For *random* insertion order the height is provably O(log n) on average (the argument mirrors quicksort's average recursion depth). But real data is rarely random: **auto-increment IDs, roll numbers, timestamps arrive sorted**. Watch what our insert algorithm does to them →

THE BILL, AT SCALE

| N KEYS | H, BALANCED | H, SORTED INPUT |
|---|---|---|
| 7 | 2 | 6 |
| 1,000 | ~10 | 999 |
| 1,000,000 | ~20 | 999,999 |

Same structure, same code — a 50,000× slowdown, from input order alone.

INTERACTIVE — INSERT 10, 20, 30, 40, 50, 60 (ALREADY SORTED)

STEP 1 OF 6

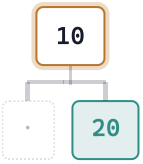
INSERTING 10, 20, 30, 40, 50, 60 — IN THAT ORDER


**Insert 10:** empty tree → root. So far so good. (These are Lecture 2's auto-increment IDs — 1001, 1002, … relabelled.)

---

STEP 2 OF 6

INSERTING 10, 20, 30, 40, 50, 60 — IN THAT ORDER

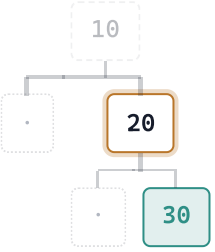



**Insert 20:** 20 > 10 → right → link. Notice the LEFT slot of 10 stays dotted — **no future key can ever fill it**, because every key still to come is larger than 10. Height 1.

---

STEP 3 OF 6

INSERTING 10, 20, 30, 40, 50, 60 — IN THAT ORDER

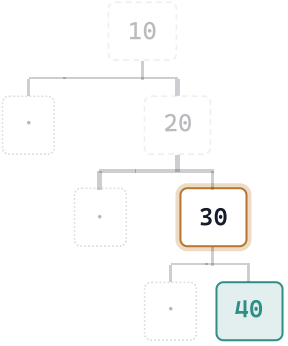



**Insert 30:** right of 10, right of 20 → link. 2 comparisons. Height 2. Every left slot: permanently dotted.

---

STEP 4 OF 6

INSERTING 10, 20, 30, 40, 50, 60 — IN THAT ORDER

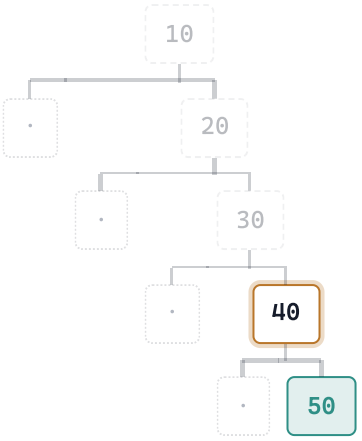



**Insert 40:** three rights → link. 3 comparisons. Height 3. The "tree" is growing in one direction only.

---

STEP 5 OF 6

INSERTING 10, 20, 30, 40, 50, 60 — IN THAT ORDER

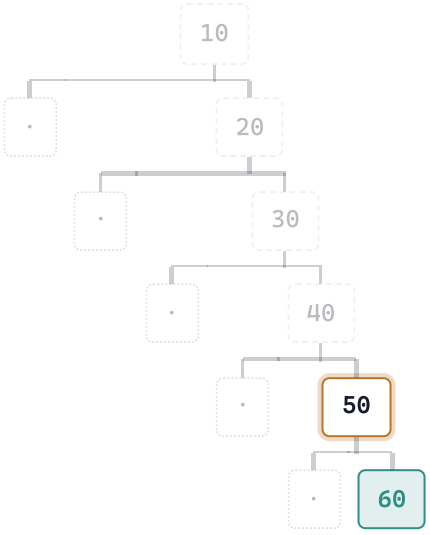



**Insert 50:** 4 comparisons. Height 4. Each insertion costs one MORE than the previous — total work so far is 0+1+2+3+4, a quadratic sum in the making.

---

STEP 6 OF 6

INSERTING 10, 20, 30, 40, 50, 60 — IN THAT ORDER




**Insert 60 — behold the "tree":** height 5 = n − 1. Every node has one dotted left slot: this is a **linked list wearing a tree costume**. Searching 60 costs 6 comparisons instead of 3; building it cost $\Sigma i = O(n^2)$. At a million sorted keys: $10^{12}$ operations. The structure didn't fail — the *input order* defeated it. Cue AVL →

# AVL: don't hope for balance — enforce it

THE DEFINITION (ADELSON-VELSKY & LANDIS, 1962)

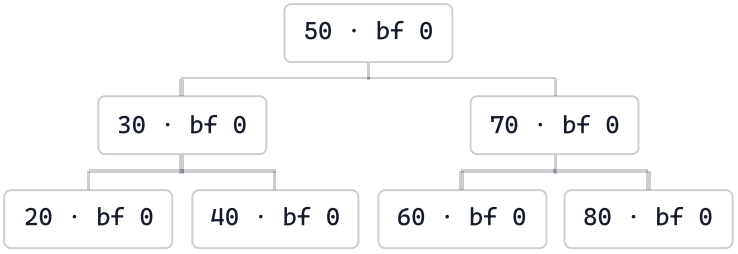
Give every node a **balance factor**:

$$bf(x) = \text{height(left subtree)} - \text{height(right subtree)}$$

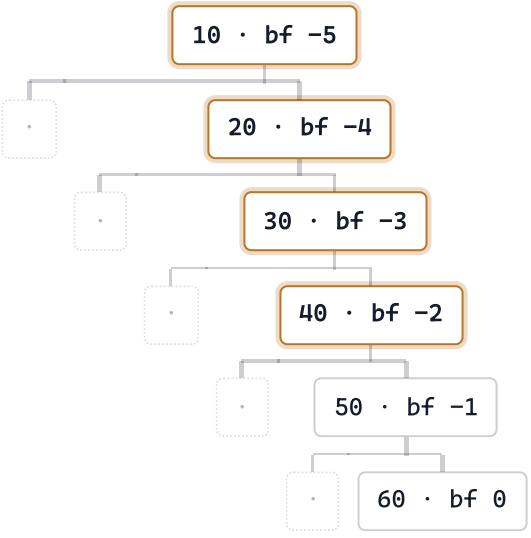
An **AVL tree** is a BST in which **every node has bf ∈ {−1, 0, +1}**. Perfectly equal heights aren't demanded — siblings may differ by one — just never more.

**The payoff, guaranteed:** this local rule forces $h \le 1.44 \log_2(n+2)$ — *whatever order the keys arrive in*. Sorted, reverse-sorted, adversarial: h stays O(log n), so search, insert, and delete stay O(log n). The hope becomes a contract.

BALANCE FACTORS, ANNOTATED ON OUR TWO TREES




THE BALANCED 7-KEY TREE: ALL BF = 0 — LEGAL AVL




THE SORTED-INPUT CHAIN: AMBER NODES VIOLATE |BF| ≤ 1 — AVL FORBIDS THIS SHAPE

The balanced tree: every bf is 0 — a valid AVL tree. The sorted-input chain: bf values −5, −4, −3, −2 **(violations highlighted)** — illegal at almost every node. An AVL tree would have refused to let this shape ever form.

THE CLIFFHANGER — HOW DOES IT REFUSE?

Insertion happens exactly as today — then the tree walks *back up the insertion path*, updating balance factors. The moment some node hits bf = ±2, the tree performs a **rotation**: an O(1) pointer surgery that lifts the deep side up and restores every bf to {−1, 0, +1}, while preserving the BST property. One rotation (sometimes a double) fixes everything. **Next lecture:** the four rotation cases — LL, RR, LR, RL — each animated the way today's operations were.

## Three takeaways

- **A BST is binary search stored as pointers:** the BST property (left ≤ node ≤ right, over whole subtrees) makes every operation a single root-to-leaf walk. Inorder traversal = sorted order, free.
- **Everything costs O(h):** search, insert (stitch a node onto an empty slot), delete (three cases; two-children reduces to the easy cases via the inorder successor).
- **h is a hostage of the input order:** sorted input drives h to n−1 — the everyday worst case. AVL's bf ∈ {−1,0,+1} rule caps h at 1.44 log n by construction; rotations (next lecture) are the enforcement mechanism.

LECTURE 8 — CONTINUES BELOW IN THIS SAME DECK

### AVL Rotations, Insertion & Deletion

LL, RR, LR, RL — the four repairs, animated; then full AVL insert/delete traces with balance factors updating live. Scroll on.

HOMEWORK — BRING TO LECTURE 8

- Build the BST for insertions `55, 25, 80, 15, 40, 70, 95, 30, 45`; draw it, then write its inorder traversal to verify.
- On your tree: delete 15, then 25, then 55 — identify which case each is, show the tree after each.
- For keys 1–7: give one insertion order producing height 2, and count how many orders produce height 6.
- Compute bf for every node of your homework tree. Is it AVL?
- Search 47 in your tree: list every comparison, and state where 47 would be inserted.
- Reading: CLRS ch. 12; Weiss §4.1–4.4.

CSE3144 — Lecture 7

## Lecture 8 — Rotations: the repair kit

Last time we ended on a promise: when a balance factor hits ±2, the tree performs an O(1) "rotation" that restores balance without breaking the BST property. Time to see the move.

> *"A rotation is the only structural edit a BST allows that changes heights but not the sorted order. AVL trees are simply BSTs that use it, immediately, every time balance slips."*

# The one legal move: rotate the deep side up

THE INTUITION

A violation means one side hangs too deep. The repair: **lift the middle key to the top** and let the other two spread beneath it. Three keys 10 < 20 < 30 can form five BST shapes — only one has height 1: **the median on top**. Every rotation is just a pointer-level way of saying "put the median on top."

Why is it legal? Because a rotation **never changes the inorder sequence** — it re-parents nodes but every key keeps the same left/right relationships to every other key. Sorted order in = sorted order out. Heights change; the dictionary doesn't.

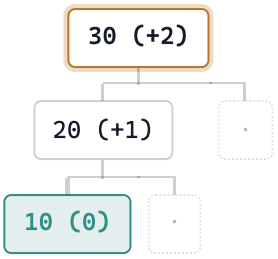
NAMING THE CASES

Let **z** = the lowest node with bf ±2, **y** = z's taller child, **x** = y's taller child. The shape of the path z→y→x names the case: **LL** (left-left: straight line leaning left) · **RR** (mirror) · **LR** and **RL** (zig-zags — next slide). Straight lines need ONE rotation; zig-zags need two.

INTERACTIVE — LL AND RR, NUMERIC THEN GENERAL

STEP 1 OF 6

THE SUBTREE BEING REPAIRED (LABELS SHOW KEY AND BALANCE FACTOR)

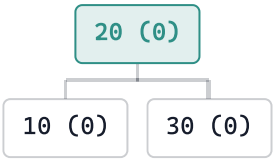



**Setup:** into the tree {30, 20} we insert **10**. Walking back up: bf(10)=0 ✓, bf(20)=+1 ✓, **bf(30)=+2 — violation.** The path 30 → 20 → 10 goes left, then left: a straight line. **Case LL.**

---

STEP 2 OF 6

THE SUBTREE BEING REPAIRED (LABELS SHOW KEY AND BALANCE FACTOR)

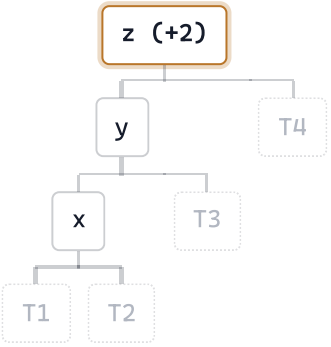



**Right rotation at 30:** the middle key 20 rises to the top; 30 swings down as its right child; 10 stays left. Three pointer writes, height 2 → 1, every bf = 0. And read the inorder: 10, 20, 30 — **unchanged**. The dictionary never noticed.

---

STEP 3 OF 6

THE GENERAL LL SHAPE — SUBTREES T1–T4 RIDE ALONG

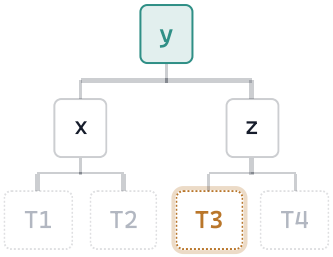



**The general case:** real trees have subtrees hanging everywhere. z is the violator, y its taller (left) child, x the deep grandchild; T1–T4 are arbitrary subtrees. Note where T3 hangs: on y's **right**.

---

STEP 4 OF 6

AFTER RIGHT-ROTATING Z

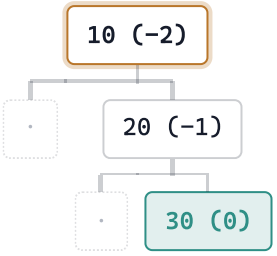



**After the rotation:** y on top, z demoted to y's right. Exactly ONE subtree changed parents: **T3, y's old right, becomes z's new left** — and it lands legally, because everything in T3 is between y and z. Check the frozen inorder: T1 · x · T2 · y · T3 · z · T4, before and after. That is the whole proof of correctness.

---

STEP 5 OF 6

THE SUBTREE BEING REPAIRED (LABELS SHOW KEY AND BALANCE FACTOR)

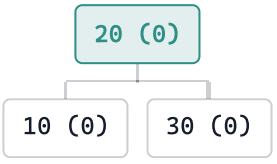



**The mirror — RR:** insert 30 into {10, 20}: path 10 → 20 → 30 goes right, then right. bf(10) = −2. Same disease, opposite side.

---

STEP 6 OF 6

THE SUBTREE BEING REPAIRED (LABELS SHOW KEY AND BALANCE FACTOR)



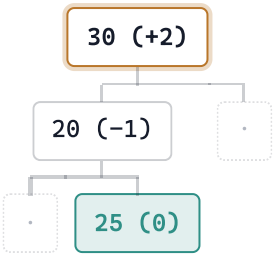

**Left rotation at 10:** 20 rises, 10 swings down-left. Identical logic, mirrored — and notice both LL and RR produced the *same* final tree: **the median on top**. That phrase is the master key to every rotation.

# Zig-zags: where one rotation fails — watch it fail

INTERACTIVE — LR: THE FAILED SHORTCUT, THEN THE REAL FIX; RL MIRRORED

STEP 1 OF 7

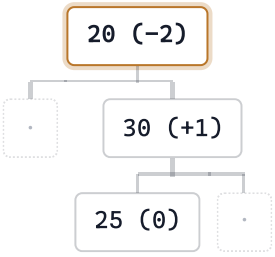
THE SUBTREE BEING REPAIRED




**Setup:** into {30, 20} we insert **25**. bf(30) = +2 → left-heavy. But look at the path: 30 → 20 goes LEFT, 20 → 25 goes RIGHT — a **zig-zag**. bf(y) = bf(20) = −1 confirms it: **Case LR.**

STEP 2 OF 7

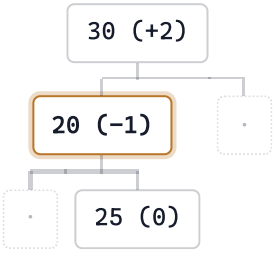
THE SUBTREE BEING REPAIRED




**The tempting shortcut — just right-rotate 30 like before… FAILS.** Result: bf(20) = −2. Still broken — the zig-zag didn't straighten, it flipped to the other side (25 is still in the middle of the path, still deep). **Lesson: single rotations only cure straight lines.**

STEP 3 OF 7

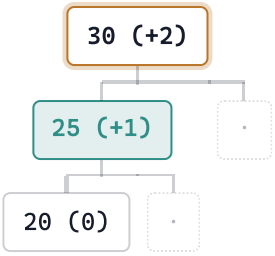
THE SUBTREE BEING REPAIRED




**Rewind.** The correct plan has two moves: first **straighten the zig-zag** — rotate at the CHILD y = 20, not at z.

STEP 4 OF 7

THE SUBTREE BEING REPAIRED

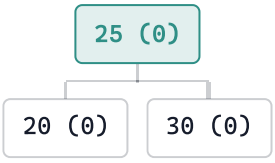



**Move 1 — left-rotate y (20):** 25 rises over 20. The path 30 → 25 → 20 is now left-left — **a straight LL line**, which we know how to fix.

---

STEP 5 OF 7

THE SUBTREE BEING REPAIRED

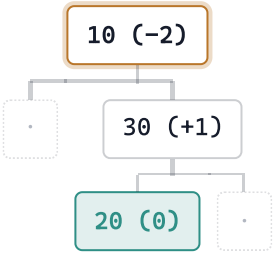



**Move 2 — right-rotate z (30):** done. **LR = left-rotate the child, right-rotate the grandparent.** And the winner is 25 — the median of {20, 25, 30} — on top. Always the median.

---

STEP 6 OF 7

THE SUBTREE BEING REPAIRED

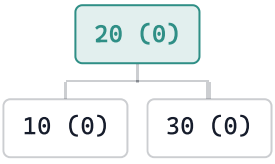



**The mirror — RL:** insert 20 into {10, 30}: right then left, bf(z) = −2 with bf(y) = +1. Repair: **right-rotate y (30), then left-rotate z (10)**.

---

STEP 7 OF 7

THE SUBTREE BEING REPAIRED




**RL complete** — median 20 on top, of course. All four cases seen. Two facts to carry out of this slide: zig-zags need two rotations, and the median always wins.

| CASE | SHAPE OF Z → Y → X | DETECT BY | REPAIR |
|---|---|---|---|
| **LL** | left, then left (straight) | `bf(z) = +2, bf(y) ≥ 0` | right-rotate z |
| **RR** | right, then right (straight) | `bf(z) = −2, bf(y) ≤ 0` | left-rotate z |
| **LR** | left, then right (zig-zag) | `bf(z) = +2, bf(y) = −1` | left-rotate y, then right-rotate z |
| **RL** | right, then left (zig-zag) | `bf(z) = −2, bf(y) = +1` | right-rotate y, then left-rotate z |

**The one-line memory trick:** in every case, of the three nodes x, y, z, **the median key ends up on top** with the other two as its children. If you forget the case table in an exam, just ask: "which of these three keys is the middle one?" — that's your new subtree root.

# AVL insertion: BST insert + walk back up + repair once

THE ALGORITHM

```
1. insert exactly as in Lecture 7 (walk, link)
2. walk BACK UP the insertion path:
      recompute bf at each ancestor
3. at the FIRST node with bf = ±2:
      classify (LL/RR/LR/RL), rotate
      stop – the subtree height is restored,
      ancestors above are automatically fine
```

**At most one repair per insertion** (single or double) — because the rotation returns the subtree to its pre-insertion height, so no ancestor ever notices anything happened. Total: O(log n) walk + O(1) surgery.

THE DEMO SEQUENCE

We insert `10, 20, 30, 40, 50, 25` — mostly ascending, exactly the poison that killed the plain BST in Lecture 7. Watch the AVL tree take the same input and stay height 2. Node labels show **key (bf)**; a bf of ±2 glows amber; teal = restructured.

INTERACTIVE — SIX INSERTIONS, TWO VIOLATIONS, TWO REPAIRS

STEP 1 OF 12

AVL TREE (LABELS: KEY (BF); AMBER = VIOLATION / PATH, TEAL = RESTRUCTURED)


**Same poison as Lecture 7:** keys arriving mostly sorted — 10, 20, 30, 40, 50, then 25. The plain BST collapsed into a chain. Watch the AVL tree refuse.

---

STEP 2 OF 12

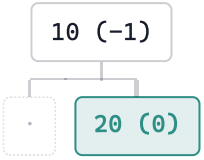
AVL TREE (LABELS: KEY (BF); AMBER = VIOLATION / PATH, TEAL = RESTRUCTURED)


**Insert 10:** root. bf = 0. Nothing to check.

---

STEP 3 OF 12

AVL TREE (LABELS: KEY (BF); AMBER = VIOLATION / PATH, TEAL = RESTRUCTURED)

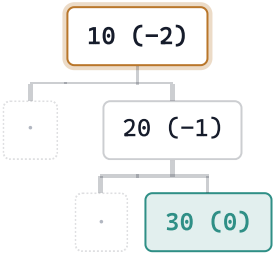



**Insert 20:** right of 10. Walk up: bf(10) = −1 — legal. No repair.

---

STEP 4 OF 12

AVL TREE (LABELS: KEY (BF); AMBER = VIOLATION / PATH, TEAL = RESTRUCTURED)

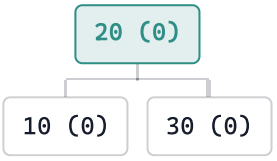



**Insert 30:** right, right — the chain is forming, exactly like Lecture 7... walk up: bf(20) = −1 ✓, **bf(10) = −2** ✗. Path right-right = **RR** → left-rotate at 10.

---

STEP 5 OF 12

AVL TREE (LABELS: KEY (BF); AMBER = VIOLATION / PATH, TEAL = RESTRUCTURED)

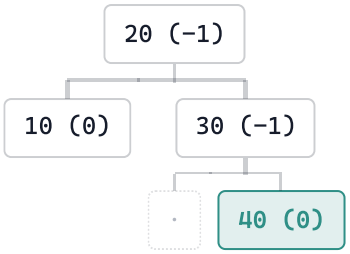



**Repaired instantly:** 20 (the median) on top. The chain never got to live. And we **stop** — no ancestors to check, the subtree is back to its old height.

---

STEP 6 OF 12

AVL TREE (LABELS: KEY (BF); AMBER = VIOLATION / PATH, TEAL = RESTRUCTURED)

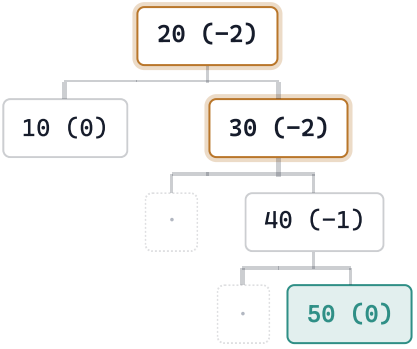



**Insert 40:** walk up: bf(30) = −1 ✓, bf(20) = −1 ✓. Legal lean — AVL tolerates ±1. No repair.

---

STEP 7 OF 12

AVL TREE (LABELS: KEY (BF); AMBER = VIOLATION / PATH, TEAL = RESTRUCTURED)

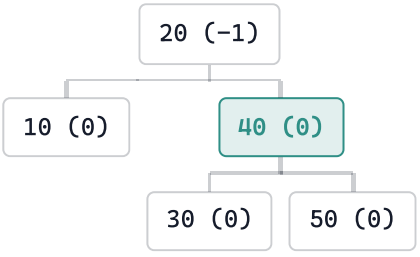



**Insert 50:** walking up: bf(40) = −1 ✓, **bf(30) = −2 ✗** — and bf(20) = −2 too! Rule: repair at the **lowest** violator first — here 30. Path 30 → 40 → 50 = RR → left-rotate at 30.

---

STEP 8 OF 12

AVL TREE (LABELS: KEY (BF); AMBER = VIOLATION / PATH, TEAL = RESTRUCTURED)

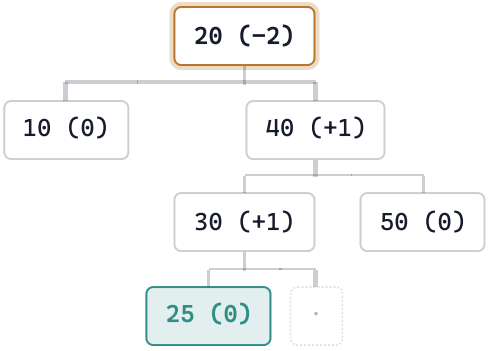



**One rotation, two violations gone:** 40 tops its subtree, and bf(20) fell back to −1 automatically — the repair restored the subtree's old height, so the ancestor healed for free. This is why insertion never needs a second repair.

---

STEP 9 OF 12

AVL TREE (LABELS: KEY (BF); AMBER = VIOLATION / PATH, TEAL = RESTRUCTURED)

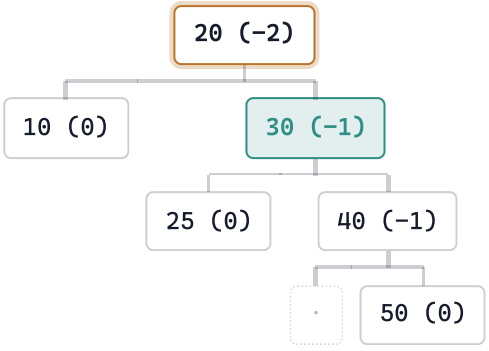



**Insert 25:** walk 20 → 40 → 30 → left slot. Going up: bf(30) = +1 ✓, bf(40) = +1 ✓, **bf(20) = −2** ✗. Now look at the shape: z = 20 is RIGHT-heavy, but its child y = 40 is LEFT-heavy (bf +1) — **zig-zag, case RL**. Double rotation required.

---

STEP 10 OF 12

AVL TREE (LABELS: KEY (BF); AMBER = VIOLATION / PATH, TEAL = RESTRUCTURED)

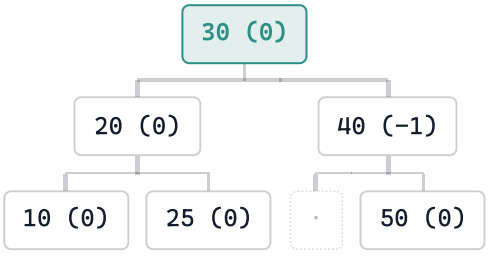



**Move 1 — right-rotate the child (40):** 30 rises over 40; 30's old right (nothing) would go to 40's left. The kink is straightened: 20 → 30 → … is now right-right.

---

STEP 11 OF 12

AVL TREE (LABELS: KEY (BF); AMBER = VIOLATION / PATH, TEAL = RESTRUCTURED)

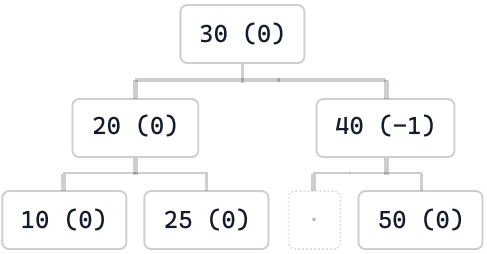



**Move 2 — left-rotate z (20):** 30 takes the root; 30's old left (25) crosses over to become 20's right — the one subtree transfer, landing legally since 20 < 25 < 30. **Final: 6 near-sorted keys, height 2, all bf legal.** The plain BST reached height 4 on this input.

---

STEP 12 OF 12

AVL TREE (LABELS: KEY (BF); AMBER = VIOLATION / PATH, TEAL = RESTRUCTURED)




**Scoreboard:** six insertions, two repairs (one single, one double), each O(1), each triggered only when a bf hit ±2 on the walk up. Inorder: 10 20 25 30 40 50 ✓. This tree will hold h ≤ 1.44 log n *forever*, whatever arrives next.

# AVL deletion: the same idea, with one twist

THE ALGORITHM

```
1. delete exactly as in Lecture 7
   (leaf / one-child / successor trick)
2. walk back UP from the deleted spot,
   recomputing bf at each ancestor
3. rotate wherever bf hits ±2
   // but do NOT stop after the first fix…
```

**The twist:** a deletion-repair can *shrink* the subtree it fixes — which may unbalance the ancestor above, which needs its own rotation, and so on. Deletion can cascade up to **O(log n) rotations** (insertion never needs more than one). Still O(log n) total.

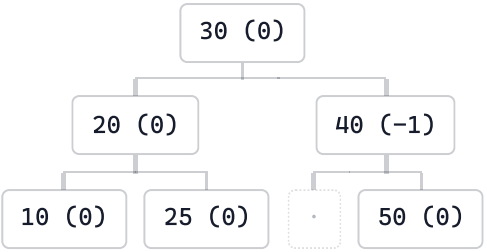
THE DEMO

We start from the tree the insertion animation just built and delete `50`, then `40` — the second deletion strips the right side bare and forces a rotation with a subtle feature: the violating node's child has **bf = 0**, a shape that only deletion can produce (insertion never creates it — good quiz question).

INTERACTIVE — TWO DELETIONS, ONE ROTATION

STEP 1 OF 5

AVL TREE (LABELS: KEY (BF))

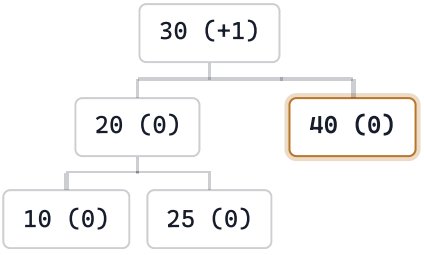



**Start:** the tree our insertion demo just built. AVL deletion = Lecture 7 deletion (leaf / one-child / successor trick) + a bf sweep **all the way up**, rotating wherever ±2 appears.

---

STEP 2 OF 5

AVL TREE (LABELS: KEY (BF))

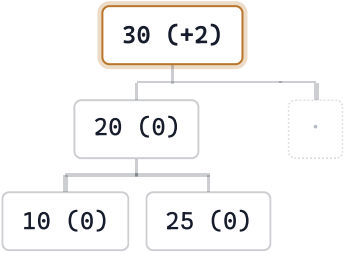



**Delete 50** (a leaf — Case 0, unlink). Sweep up: bf(40) = 0 ✓, bf(30) = +1 ✓. Legal all the way; no rotation. Deletion is often this quiet.

---

STEP 3 OF 5

AVL TREE (LABELS: KEY (BF))

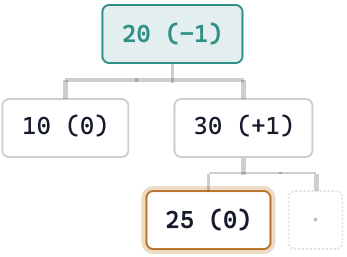



**Delete 40** (leaf again). Sweep up: **bf(30) = +2** ✗ — the right side is bare. z = 30; its taller child y = 20 has **bf = 0**. The rule (see the case table): bf(y) ≥ 0 → treat as **LL** → single right rotation. Fun fact: bf(y) = 0 at repair time can ONLY happen after a deletion — insertions never produce it.

---

STEP 4 OF 5

AVL TREE (LABELS: KEY (BF))

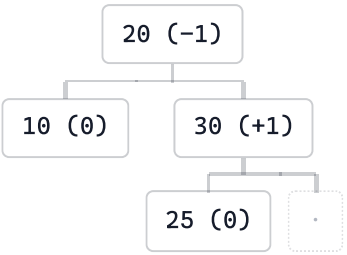



**Right rotation at 30:** 20 up, 30 down-right — and the crossing subtree: **25, which was 20's right, becomes 30's left** (legal: 20 < 25 < 30). New bfs: 30:+1, 20:−1 — all legal. Inorder: 10 20 25 30 ✓.

---

STEP 5 OF 5

AVL TREE (LABELS: KEY (BF))




**The twist, stated precisely:** here the repair kept the subtree's height, so the sweep ended. But when a deletion-repair *lowers* the subtree height, the parent's bf shifts and may itself hit ±2 — so you must **keep sweeping to the root**: up to O(log n) rotations per deletion, versus at most one per insertion. Total cost, either way: O(log n).

# Lecture 8 takeaways

- **A rotation is O(1) pointer surgery that preserves inorder** — heights change, sorted order never does. Straight-line violations (LL/RR) take one rotation; zig-zags (LR/RL) take two. Either way: **the median of x, y, z ends up on top.**
- **Insertion:** BST insert, walk up, repair at the *first* ±2 — and stop. At most one repair, ever.
- **Deletion:** BST delete, walk up, repair *every* ±2 you meet — repairs can cascade, up to O(log n) of them. All operations: guaranteed O(log n), any input order.

LECTURE 9

### Red-Black Trees

AVL's rival: slightly looser balance (h ≤ 2 log n), but cheaper maintenance — the tree behind C++ std::map and Java TreeMap. Same rotations, new bookkeeping: colors.

HOMEWORK — BRING TO LECTURE 9

- Insert `1, 2, 3, 4, 5, 6, 7` into an AVL tree, in order. Show every balance factor and every rotation. (Sorted input — the plain BST's nightmare; count how many rotations AVL pays to stay height ≤ 2.9 log n.)
- Insert `50, 25, 75, 10, 30, 5` (an LL case), then separately `50, 25, 75, 10, 30, 28` (an LR case). Draw before/after for each rotation.
- From your Lecture 7 homework tree: compute all bf; if it isn't AVL, find the minimum set of rotations to fix it.
- Delete the root from the final tree of today's insertion demo; show the successor trick + any rebalancing.
- Argue in 3–4 sentences: why does an insertion repair never propagate above the first fixed node, while a deletion repair can?
- Reading: Weiss §4.4 (AVL trees); CLRS ch. 13 intro (for next time).



## Questions?

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

> *Next: Lecture 9 — Red-Black Trees and Operations.*