Splay Trees: the Tree That Learns Your Habits

No balance factors, no colors, no guarantees on any single operation — just one rule: whatever you touch, pull to the root. And the payoff Lecture 3 promised: the potential method finally earns its keep.

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

~50 minutes
Session outcome: apply splaying operations and analyze efficiency

~55 MIN TOTAL

Today, minute by minute

The need: real access patterns aren't random 00–05

Locality of reference; zero bookkeeping; where splay trees actually run.

Animation: the three splay steps 05–13

Zig, zig-zig, zig-zag — every move on one tree.

Animation: the showdown — naive rotate-to-root vs splaying 13–21

Watch the naive fix fail on a chain, then watch splaying fold the chain in half.

Animation: insertion via splaying 21–29

Insert 15, 10, 17, 7 — every single rotation of every splay shown on its own.

Animation: top-down deletion via splaying 29–37

Splay the victim out, then join — every rotation of both splays shown on its own.

The Lecture 3 callback — why "amortized" is the only honest word here 37–42

The battery, formalized: Φ for splay trees, now that you've watched it work.

Worked example: Φ on a real tree, cheap vs expensive splay 42–47

The chain from the showdown, numbers in hand: $\Delta\Phi = -1.585$ vs $\Delta\Phi = 0$.

Efficiency: the Access Lemma + recap 47–55

Amortized O(log n); the three-tree comparison; homework.

AVL and Red-Black optimize for an enemy that rarely shows up

THE OBSERVATION BOTH OF THEM IGNORE

Balanced trees treat every key as equally likely to be asked for next. Real workloads don't behave that way: you re-open the same few contacts, a compiler looks up the same identifiers hundreds of times, a router forwards to the same handful of routes — the classic **80/20 pattern**: 80% of accesses touch 20% of the keys, and **what you touched recently, you'll likely touch again** (locality of reference).

A balanced tree keeps a key you ask for a thousand times a day at depth 20, forever, because *fairness to all keys* is its only goal. What if the tree instead **moved the popular keys near the root** — automatically, with no counters, no statistics?

THE SPLAY TREE'S OFFER (SLEATOR & TARJAN, 1985)

- **One rule:** after every operation — search, insert, even a failed search — **splay the accessed node to the root** via special rotations. Hot keys drift up; cold keys sink. The tree *is* its own cache.
- **Zero bookkeeping:** AVL stores a balance factor per node, Red-Black stores a color bit. A splay tree stores **nothing** — plain BST nodes. Simplest code of the three.
- **Used in:** Windows NT (memory management), GCC internals, caches, network routers — anywhere access patterns are skewed.
- **The honest price:** no per-operation guarantee. A single access can cost **Θ(n)**. The tree can even *be* a chain at some moment. And yet — the total over any sequence is provably fast. Hold that thought for the next slide.

Analogy: your kitchen. AVL is the labelled spice rack — everything equally reachable, always tidy, effort spent tidying after every use. The splay tree is how kitchens actually work: **whatever you used stays in front**. The turmeric you use daily is at arm's reach; the star anise from last Diwali has drifted to the back. Nobody maintains this arrangement — using things *is* the arrangement.

Three moves, chosen by where x stands

THE DECISION RULE

To splay node x, repeat until x is the root — asking, each round, three questions: does x have a grandparent? which side of the parent is x? which side of the grandparent is the parent?

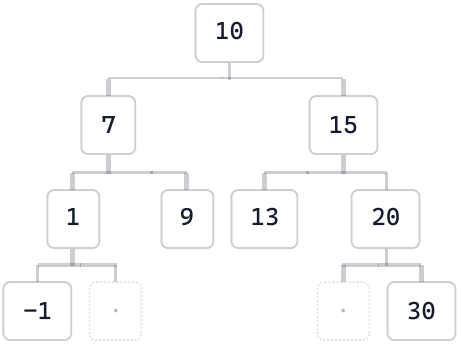
- **Zig / Zag** — x's parent IS the root: one single rotation (right if x is a left child, left otherwise). The terminal move.
- **Zig-zig / Zag-zag** — straight line (LL or RR): rotate the **grandparent edge first**, then the parent edge. Two rotations.
- **Zig-zag / Zag-zig** — bent path (LR or RL): rotate the **parent edge first**, then the grandparent edge — the same two-rotation repair you used for AVL's bent (LR/RL) cases: it lifts x over the bend while preserving sorted order.

The subtle heart of the algorithm: in the straight-line case, splaying rotates the *grandparent first*. Plain "rotate x up twice" looks almost identical — and destroys the amortized guarantee. The showdown on the next slide exists to prove that this tiny ordering choice is everything.

INTERACTIVE — ZIG, ZIG-ZIG, ZIG-ZAG — ONE TREE, THREE SPLAYS

STEP 1 OF 9

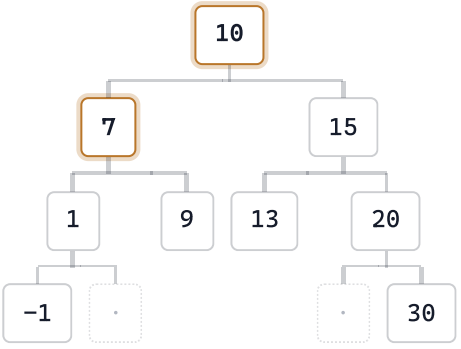
OUR PLAYGROUND TREE (AMBER = X AND ITS PATH; TEAL = X AFTER THE MOVE)



The playground. We'll splay three different nodes to see all three step types. First: **search 7**. Its parent (10) is the root — the terminal case.

STEP 2 OF 9

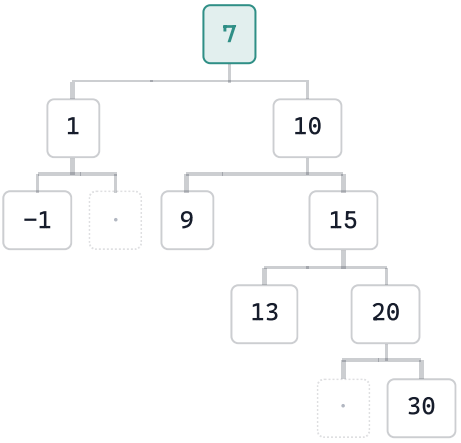
OUR PLAYGROUND TREE (AMBER = X AND ITS PATH; TEAL = X AFTER THE MOVE)



ZIG: x = 7 is the LEFT child of the root → one right rotation at the root. (Had x been a right child: one left rotation — "zag." That's the entire terminal move.)

STEP 3 OF 9

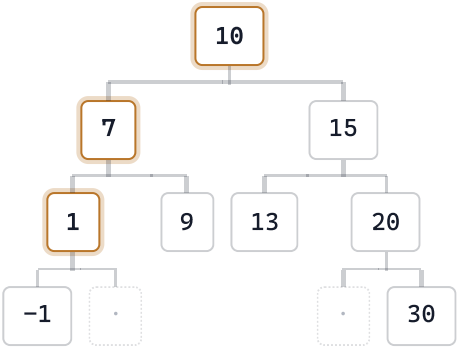
OUR PLAYGROUND TREE (AMBER = X AND ITS PATH; TEAL = X AFTER THE MOVE)



7 is the root: 10 swung down-right keeping 9 (which crossed over as 10's left) and the whole right side. One rotation, done. Now reset the tree and try something deeper: **search 1**.

STEP 4 OF 9

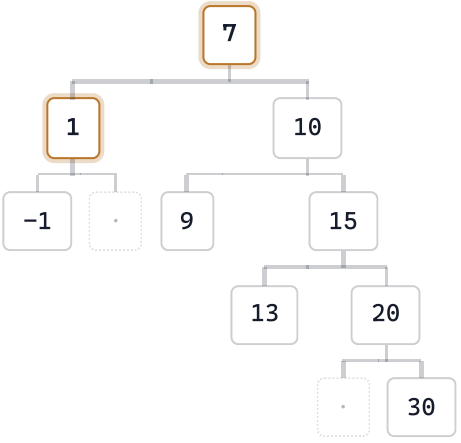
OUR PLAYGROUND TREE (AMBER = X AND ITS PATH; TEAL = X AFTER THE MOVE)



ZIG-ZIG: x = 1, parent 7, grandparent 10 — a straight left-left line. Rule: rotate the **grandparent edge FIRST** (right-rotate 10), then the parent edge. NOT "rotate x up twice."

STEP 5 OF 9

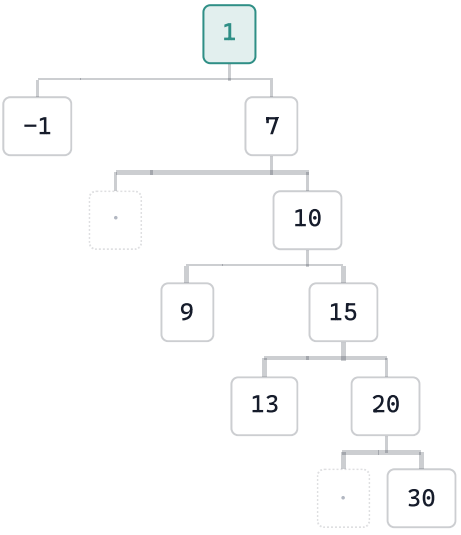
OUR PLAYGROUND TREE (AMBER = X AND ITS PATH; TEAL = X AFTER THE MOVE)



After rotation 1 (at the grandparent 10): 7 rose to the top; x = 1 is now the root's child — one more rotation to go.

STEP 6 OF 9

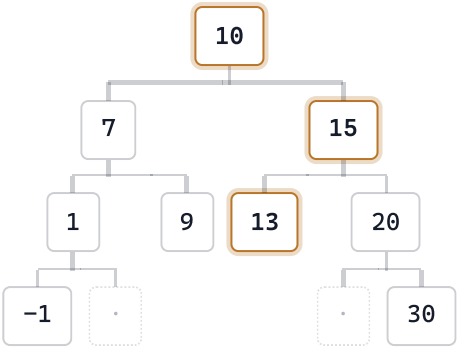
OUR PLAYGROUND TREE (AMBER = X AND ITS PATH; TEAL = X AFTER THE MOVE)



1 is the root: the chain 10–7–1 has become 1–7–10 hanging *rightward*, each with its subtrees relocated legally. Now the bent case: reset, and **search 13**.

STEP 7 OF 9

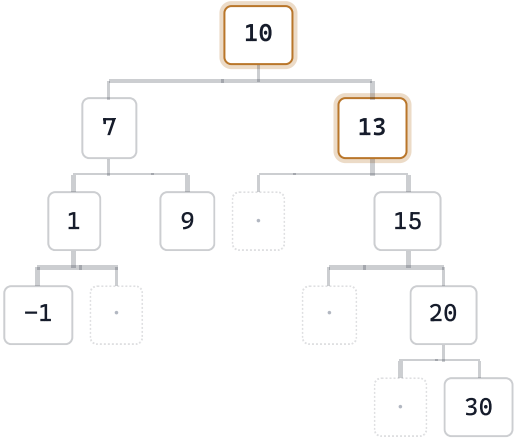
OUR PLAYGROUND TREE (AMBER = X AND ITS PATH; TEAL = X AFTER THE MOVE)



ZIG-ZAG: x = 13 is the LEFT child of 15, which is the RIGHT child of 10 — a bend. Rule for bends: rotate the **parent edge first** (right-rotate at 15) — two rotations carrying x over the bend, sorted order intact.

STEP 8 OF 9

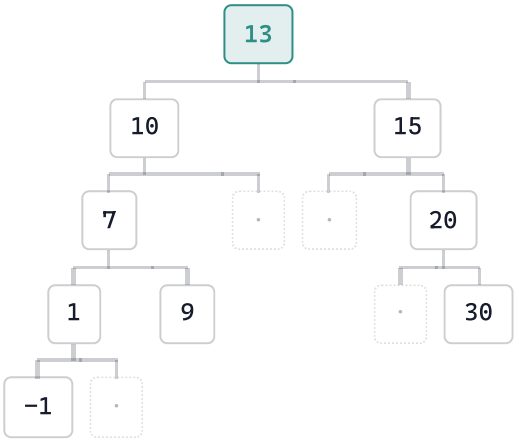
OUR PLAYGROUND TREE (AMBER = X AND ITS PATH; TEAL = X AFTER THE MOVE)



After rotation 1 (at 15): 13 rose over 15; the bend is straightened into a line under 10. One left rotation at the root finishes it.

STEP 9 OF 9

OUR PLAYGROUND TREE (AMBER = X AND ITS PATH; TEAL = X AFTER THE MOVE)



13 is the root. Score card: **zig** = one rotation at the root; **zig-zig** (straight) = grandparent edge first, then parent; **zig-zag** (bent) = parent edge first, then grandparent. The mirrored versions (zag, zag-zag, zag-zig) are the same moves flipped.

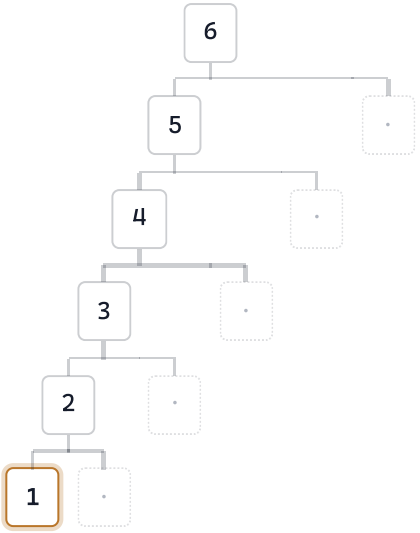
Naive rotate-to-root vs splaying — same chain, opposite fates

The worst tree we know: the left chain from inserting 6, 5, 4, 3, 2, 1. Both strategies bring the accessed node 1 to the root using rotations. Only one of them deserves to be an algorithm.

INTERACTIVE — WATCH THE NAIVE FIX FAIL, THEN WATCH SPLAYING FOLD THE CHAIN

STEP 1 OF 13

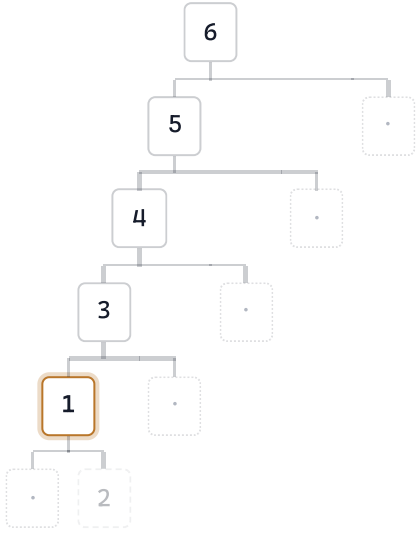
THE WORST-CASE TREE: KEYS 6, 5, 4, 3, 2, 1 INSERTED IN THAT ORDER



The patient: a left chain of height 5 — Lecture 7's nightmare, alive again. Accessing **1** costs 6 comparisons. Both strategies now bring 1 to the root by rotations. Strategy A: the "obvious" one — **rotate 1 up, one level at a time**, a single rotation per step, no look-ahead at grandparents.

STEP 2 OF 13

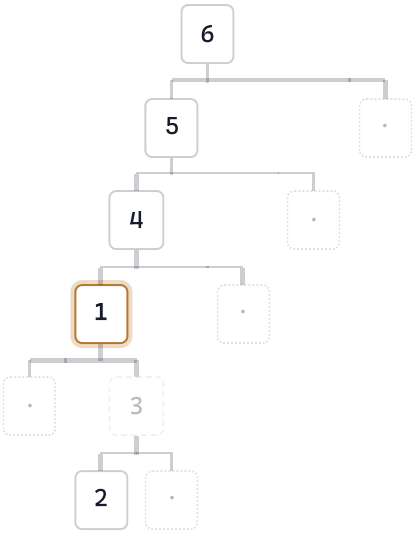
THE WORST-CASE TREE: KEYS 6, 5, 4, 3, 2, 1 INSERTED IN THAT ORDER



Rotation 1 of 5 — right-rotate the edge (2, 1): 1 rises one level (depth 5 → 4); its old parent 2 drops to become 1's right child. That's the whole move — one edge, no glance further up the tree.

STEP 3 OF 13

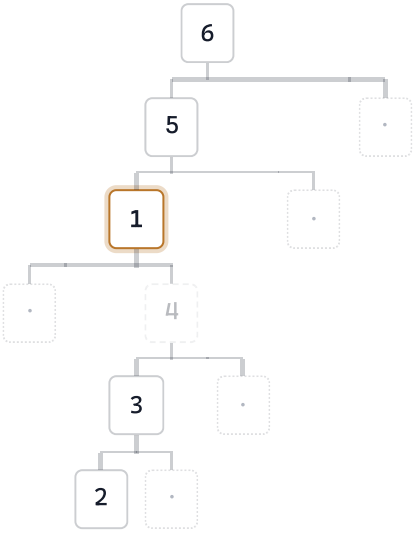
THE WORST-CASE TREE: KEYS 6, 5, 4, 3, 2, 1 INSERTED IN THAT ORDER



Rotation 2 of 5 — right-rotate (3, 1): 1 rises again (depth 4 → 3); 3 becomes 1's new right child, and 2 rides along as 3's left child. Still just single edge-rotations, blind to the grandparent.

STEP 4 OF 13

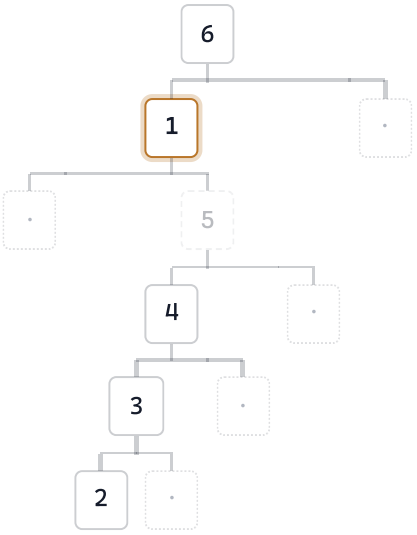
THE WORST-CASE TREE: KEYS 6, 5, 4, 3, 2, 1 INSERTED IN THAT ORDER



Rotation 3 of 5 — right-rotate (4, 1): depth 3 → 2. The little {2,3} clump from before rides along as 4's new left child.

STEP 5 OF 13

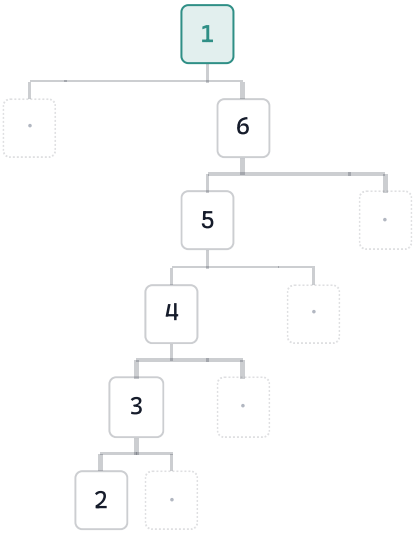
THE WORST-CASE TREE: KEYS 6, 5, 4, 3, 2, 1 INSERTED IN THAT ORDER



Rotation 4 of 5 — right-rotate (5, 1): depth 2 → 1. One rotation from the root.

STEP 6 OF 13

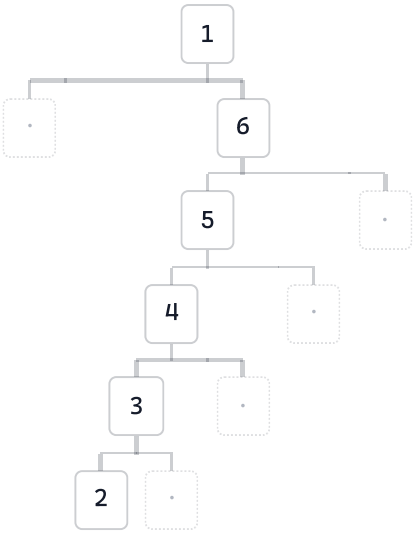
THE WORST-CASE TREE: KEYS 6, 5, 4, 3, 2, 1 INSERTED IN THAT ORDER



Rotation 5 of 5 — right-rotate (6, 1): 1 reaches the root. **Look closely at what's left — it's still a chain!** Height 5: 6 hangs directly under 1, and 5, 4, 3, 2 continue the exact same left-leaning chain shape underneath, just shifted down one level. Five single rotations, and the tree is exactly as lopsided as it started.

STEP 7 OF 13

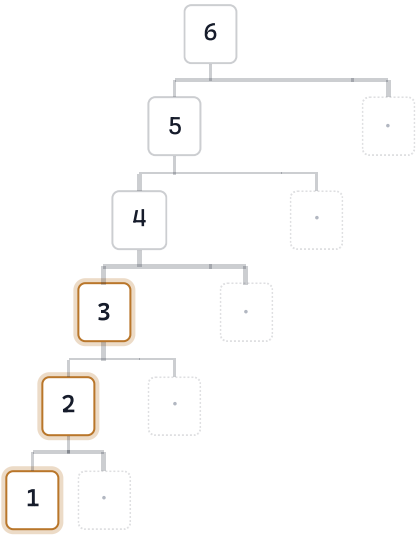
THE WORST-CASE TREE: KEYS 6, 5, 4, 3, 2, 1 INSERTED IN THAT ORDER



The verdict: access 2 next — it's still 4 levels down, ~n steps again, and repeated naive rotate-to-root just keeps re-forming a height-(n−1) chain. A sequence of n such accesses costs **$\Theta(n^2)$**. The battery never discharged — the naive fix paid the same actual cost as splaying, but moved the trouble around without spending it.

STEP 8 OF 13

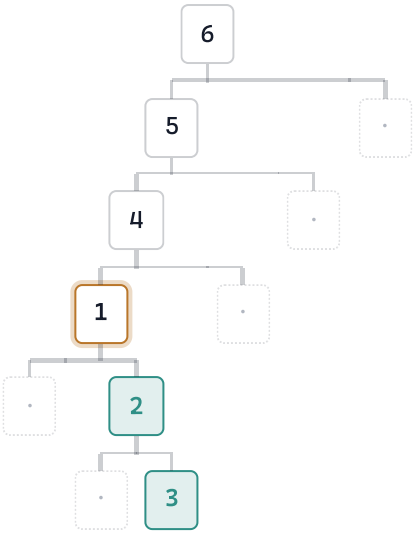
THE WORST-CASE TREE: KEYS 6, 5, 4, 3, 2, 1 INSERTED IN THAT ORDER



Rewind. Strategy B — true splaying. x = 1, parent 2, grandparent 3: a straight line → **zig-zig**: rotate the grandparent edge (right-rotate 3), then the parent edge (right-rotate 2).

STEP 9 OF 13

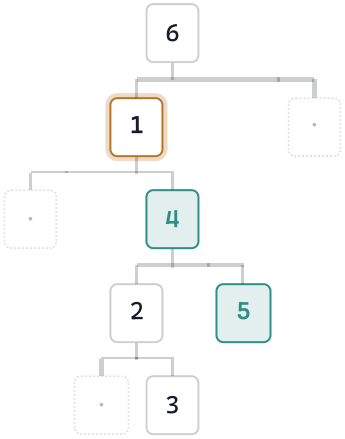
THE WORST-CASE TREE: KEYS 6, 5, 4, 3, 2, 1 INSERTED IN THAT ORDER



After zig-zig #1: the bottom of the chain folded — 2 and 3 now hang *rightward under 1* instead of stacking above it. 1 jumped two levels. Next: x = 1, parent 4, grandparent 5 — another straight line.

STEP 10 OF 13

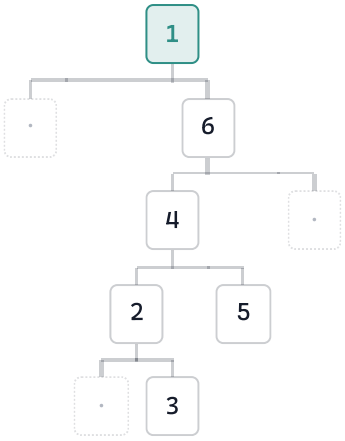
THE WORST-CASE TREE: KEYS 6, 5, 4, 3, 2, 1 INSERTED IN THAT ORDER



After zig-zig #2: 4 and 5 folded into a little balanced clump under 1's right. See the shape forming? The chain is being **zipped up into bushes**. One step left: 1's parent 6 is the root.

STEP 11 OF 13

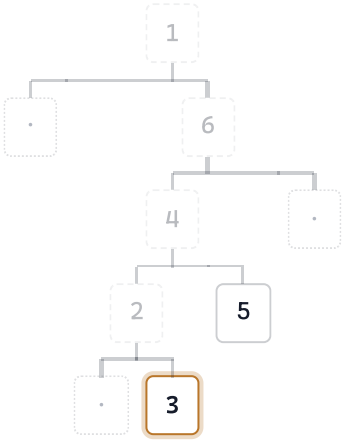
THE WORST-CASE TREE: KEYS 6, 5, 4, 3, 2, 1 INSERTED IN THAT ORDER



Zig, done. Compare with Strategy A: same node lifted, same actual cost paid — but the tree is now height 4 with real branching, and the deep nodes 2, 3, 4, 5 all rose. **The access path folded roughly in half.** Let's cash the benefit: access **3** (it's at depth 4).

STEP 12 OF 13

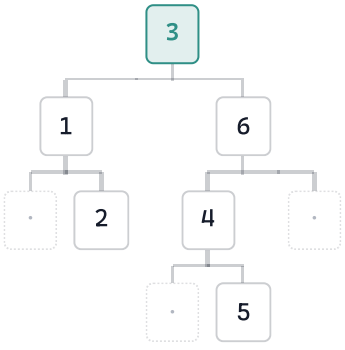
THE WORST-CASE TREE: KEYS 6, 5, 4, 3, 2, 1 INSERTED IN THAT ORDER



Splay(3): path 1 → 6 → 4 → 2 → 3. First round: x = 3, parent 2, grandparent 4 — 3 is RIGHT of 2, 2 is LEFT of 4: a bend → **zig-zag** (parent edge first).

STEP 13 OF 13

THE WORST-CASE TREE: KEYS 6, 5, 4, 3, 2, 1 INSERTED IN THAT ORDER



Two rounds later, 3 is the root — and look at the tree: height 3, from a chain of height 5, after just two accesses. No balance rule was ever consulted; the accesses themselves did the balancing. That is self-adjustment — and the potential method is how we'll prove it never stops working.

The takeaway, in Lecture-3 language: both strategies paid the same actual cost to lift node 1. But the naive one left the battery fully charged — the same Θ(n) disaster is waiting for the very next access. Splaying **spent the potential**: the walked path folded, so future accesses into that region are cheap. Expensive operations that repair the structure — that is the entire theory of this course paying off in one animation.

Insert: BST-insert, then splay — every rotation on its own

SEARCH(K)

BST search, then **splay the found node** — or, if absent, splay the **last node visited** before NIL. Even failures reorganize the tree (a failed lookup near k makes the whole neighbourhood of k cheaper next time).

INSERT(K)

BST insert (Lecture 7), then **splay the new node to the root**. Fresh data starts hot — usually the right guess. Below: every single rotation from L10·03's decision rule (zig / zig-zig), shown one at a time — no two rotations ever collapsed into one step.

INTERACTIVE — INSERT 15, 10, 17, 7 — EVERY ROTATION, ONE CLICK AT A TIME

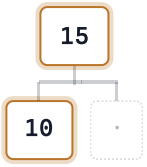
STEP 1 OF 10

SPLAY TREE

Insert 15 into an empty tree: it becomes the root. Splaying a root is a no-op. (The full sequence: 15, 10, 17, 7.)

STEP 2 OF 10

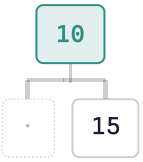
SPLAY TREE



Insert 10: BST-insert puts it left of 15. Now splay: parent is the root → **zig** (single right rotation) is the only move needed.

STEP 3 OF 10

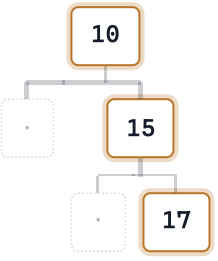
SPLAY TREE



Zig applied — 10 is the root. Every insert finishes with the new key on top — new data starts "hot," which matches most real workloads (you usually touch what you just added).

STEP 4 OF 10

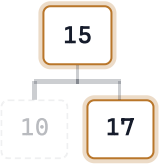
SPLAY TREE



Insert 17: BST-insert: right of 15, which is right of root 10 — a straight right-right line. x = 17, parent 15, grandparent 10 → **zig-zig**, two rotations. First: rotate the **grandparent edge** (left-rotate 10).

STEP 5 OF 10

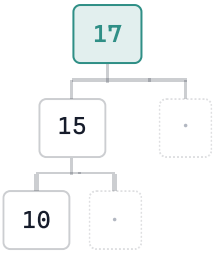
SPLAY TREE



Rotation 1 of 2 done. 15 has risen to the top of this span, with 10 and 17 as its two children — but x = 17 isn't on top yet. Second rotation: rotate the **parent edge** (left-rotate 15).

STEP 6 OF 10

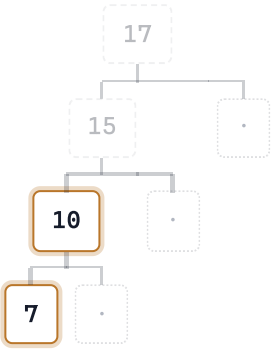
SPLAY TREE



Zig-zig complete — 17 is the root (17 → 15 → 10 leaning left). Yes, it's a chain right now! Splay trees tolerate ugly moments — the potential is charged, and the next deep access will spend it.

STEP 7 OF 10

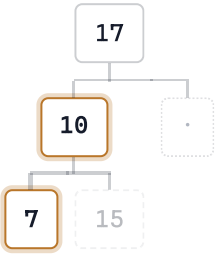
SPLAY TREE



Insert 7: BST-insert: leftmost, at depth 3. x = 7, parent 10, grandparent 15 — straight left-left line → zig-zig again. First: rotate the **grandparent edge** (right-rotate 15).

STEP 8 OF 10

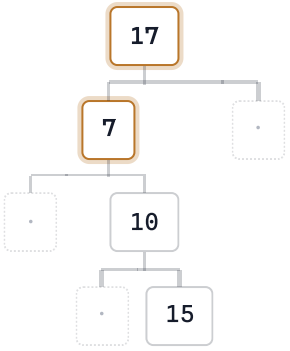
SPLAY TREE



Rotation 1 of 2 done. 10 has risen, with 7 and 15 as its two children. x = 7 still needs one more rotation to reach the top of this span. Second: rotate the **parent edge** (right-rotate 10).

STEP 9 OF 10

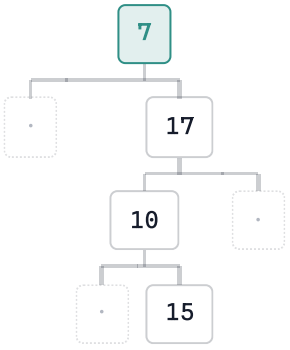
SPLAY TREE



Zig-zig complete. 7 rose two levels and folded 10, 15 rightward beneath it. One more move: x = 7's parent is now 17, the root → a final **zig** (single rotation) finishes the splay.

STEP 10 OF 10

SPLAY TREE



Zig applied — 7 is the root. Insertion sequence complete: three inserts needed 1 + 2 + 3 = 6 rotations total, each shown on its own. Deletion is next, on its own slide.

Delete: top-down — splay the victim out, then join — every rotation on its own

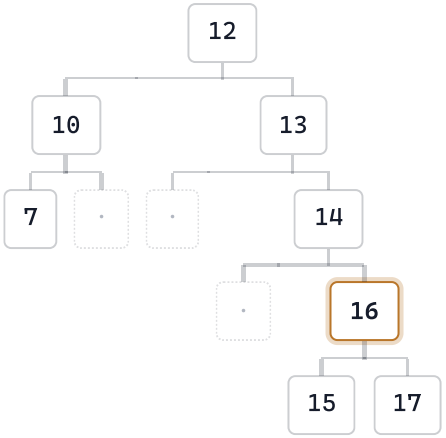
DELETE(K) — TOP-DOWN

Splay k to the root, remove it — leaving left subtree L and right subtree R. **Join:** splay the maximum of L to L's root (it provably has no right child once it's there!), and hang R directly onto that empty slot. Two splays, no successor-copying, no case analysis. A fresh, larger tree below — every rotation of both splays gets its own step.

INTERACTIVE — TOP-DOWN DELETE 16 — EVERY ROTATION, ONE CLICK AT A TIME

STEP 1 OF 10

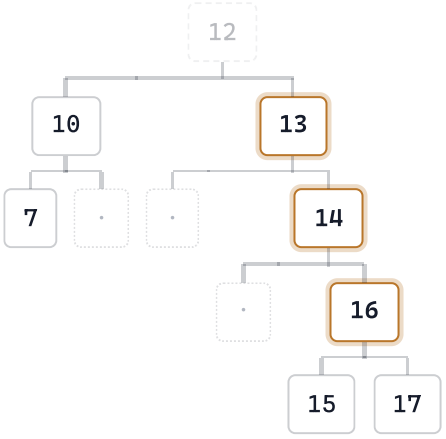
SPLAY TREE



A fresh, larger tree. Delete **16**. Top-down deletion's first move: splay 16 to the root *before* touching anything — bring the victim up first, then remove it from the top.

STEP 2 OF 10

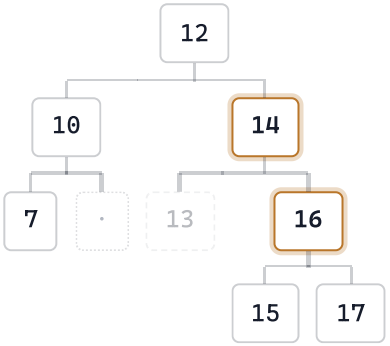
SPLAY TREE



Path to 16: 12 → 13 → 14 → 16. x = 16, parent 14, grandparent 13 — straight right-right line (13 → 14 → 16 all lean right) → zig-zig first, then one more move once we reach the root. First: rotate the **grandparent edge** (left-rotate 13).

STEP 3 OF 10

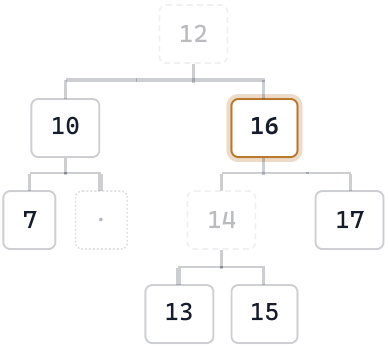
SPLAY TREE



Rotation 1 of 2 done. 14 has risen to 12's right child, with 13 and 16 as its two children. x = 16 still needs one more rotation. Second: rotate the **parent edge** (left-rotate 14).

STEP 4 OF 10

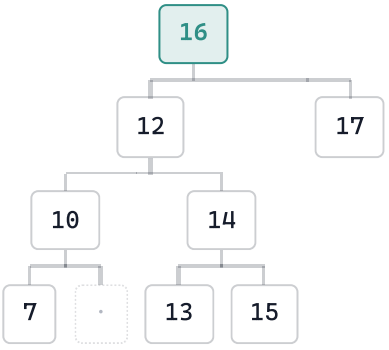
SPLAY TREE



Zig-zig complete. 16 is now 12's right child directly. x = 16's parent is 12, the root → one final **zag** (left-rotate 12) finishes the splay.

STEP 5 OF 10

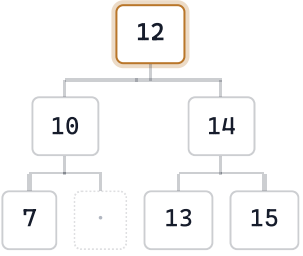
SPLAY TREE



Zag applied — 16 is the root. Three rotations total to splay it up (matching the path length). Now the actual deletion: remove 16, keeping its two subtrees.

STEP 6 OF 10

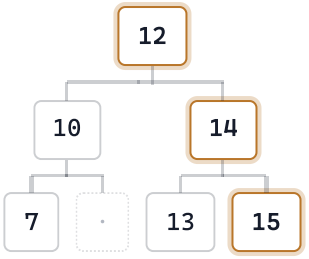
L (EVERYTHING < 16) — R = THE SINGLE NODE 17, HELD ASIDE, NOT YET ATTACHED



16 removed. Its left subtree becomes L (rooted at 12); its right subtree, R, was just the single node 17 — set aside for now, not attached to anything. **Join, step 1:** splay the **maximum** of L to L's own root. Find it: 12 → 14 → 15 (rightmost).

STEP 7 OF 10

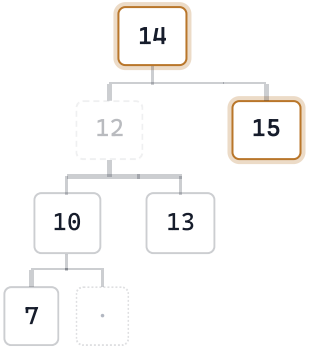
L ONLY



x = 15, parent 14, grandparent 12 (L's own root) — straight right-right line → zig-zig, and since the grandparent here is already L's root, these two rotations will finish this splay completely. First: rotate the **grandparent edge** (left-rotate 12).

STEP 8 OF 10

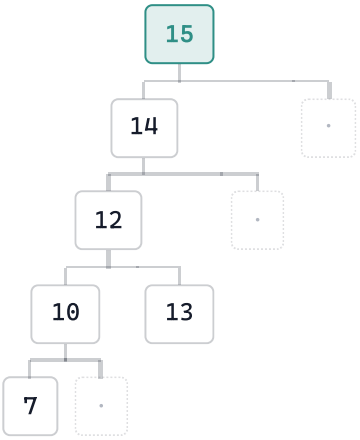
L ONLY



Rotation 1 of 2 done. 14 is now L's root, with 12 and 15 as its two children. x = 15 needs one more rotation. Second: rotate the **parent edge** (left-rotate 14).

STEP 9 OF 10

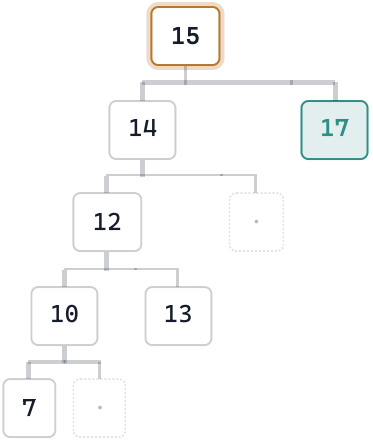
L ONLY — RIGHT SLOT CONFIRMED EMPTY



Zig-zig complete — 15 is L's root, with NO right child. This is exactly why we splayed the *maximum*: the largest key in L can never have anything larger to its right, so once it's on top, that slot is guaranteed empty — precisely where R belongs.

STEP 10 OF 10

SPLAY TREE



Join, step 2: hang R (17) directly onto 15's empty right slot. Deletion complete. Inorder: 7 10 12 13 14 15 17 ✓. Five rotations total (three to splay 16 out, two to splay 15 up, zero for the join itself) — no successor-copying, no case analysis, splaying alone did all the work.

Remember the battery? This is what it was for.

RECAP — LECTURE 3 IN THREE LINES

- **Amortized** = worst case over a *sequence*, no probability. A single operation may be slow if the sequence stays fast.
- **Potential method:** pick Φ = "trouble stored in the structure." Amortized cost ĉ = actual cost + ΔΦ. Cheap ops charge the battery; expensive ops run on discharge.
- Back then we *practised* on toy examples and told you: "park the doubt — a structure is coming that cannot be analyzed any other way." **It has arrived — you just watched it.**

WHY SPLAY TREES NEED IT

A splay tree makes **no promise about any single operation** — an access can cost Θ(n), and no worst-case-per-op analysis can say anything better. AVL had bf-arithmetic; Red-Black had color rules; the splay tree's only defense is: **"the expensive access repaired the tree — the sequence as a whole stays cheap."** That sentence IS amortized analysis. There is no other honest vocabulary for this structure.

The battery, concretely: Φ(tree) = Σ over all nodes of log(size of that node's subtree) — the "rank potential." A lopsided chain has HUGE potential (a fully charged battery of trouble); a balanced tree has small potential. An expensive splay of a deep node **discharges** the battery — the tree it leaves behind is flatter, and the discharge pays for the work. Exactly the inverter analogy: metered at the wall socket, every access reads O(log n).

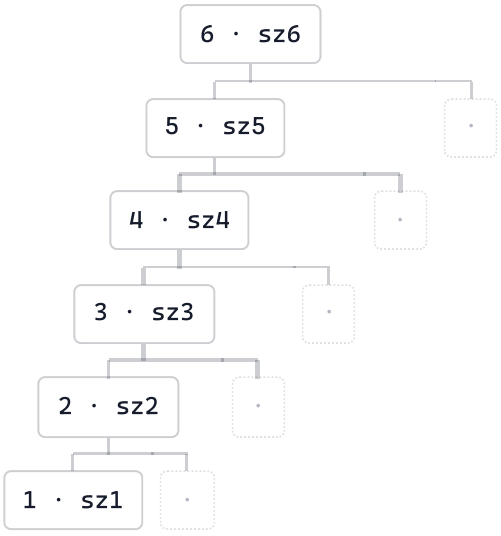
LOOK BACK THROUGH THIS LENS

Every animation you just watched now reads differently. Zig-zig and zig-zag guarantee that a costly splay leaves the walked path **roughly half as deep** — that's a big drop in Φ, discharging exactly what the deep access spent. The showdown proved naive rotate-to-root doesn't do this — no drop in Φ, so it stays stuck at Θ(n) per access forever. Next: the Access Lemma states this discharge precisely.

A worked example: Φ across three splays

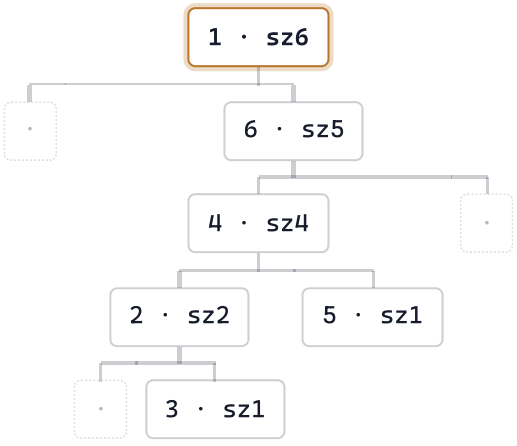
Φ(tree) = Σ over every node x of $\log_2$(size(x)), where size(x) = nodes in x's subtree. Actual cost c(splay(x)) ≈ depth(x). Amortized $\hat{c}$ = c + ΔΦ — read it exactly like Lecture 3's PUSH / MULTIPOP table. First two trees: same 6-node chain as the L10·03 showdown.

D₀ — START: PURE CHAIN



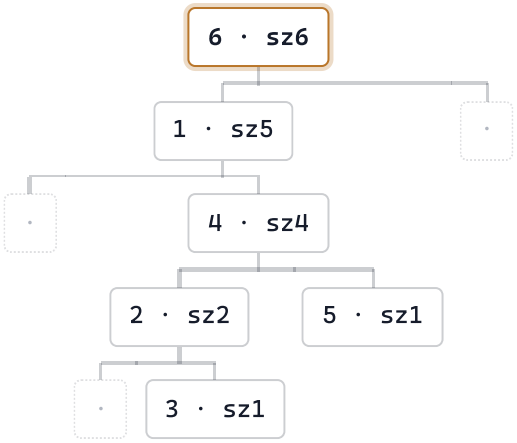
Φ = LOG₂6+LOG₂5+LOG₂4+LOG₂3+LOG₂2+LOG₂1 = 9.492

D₁ — AFTER SPLAY(1), DEPTH 5→0



Φ = LOG₂6+LOG₂5+LOG₂4+LOG₂2+LOG₂1+LOG₂1 = 7.907

D₂ — AFTER SPLAY(6), DEPTH 1→0

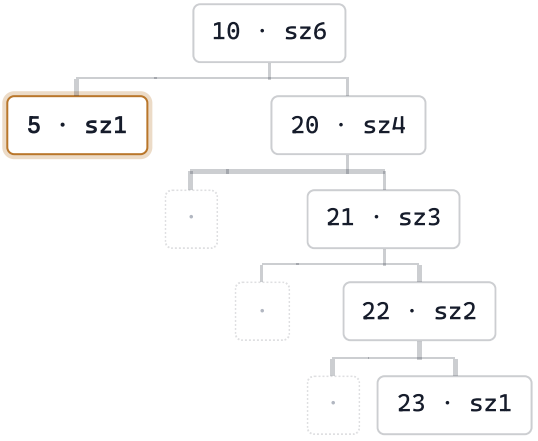


Φ = LOG₂6+LOG₂5+LOG₂4+LOG₂2+LOG₂1+LOG₂1 = 7.907 (UNCHANGED)

SPLAY	C (ACTUAL)	Φ BEFORE	Φ AFTER	ΔΦ	Ĉ = C+ΔΦ
splay(1) — **expensive**	5	9.492	7.907	**−1.585**	**3.415**
splay(6) — **cheap, neutral**	1	7.907	7.907	**0**	**1**

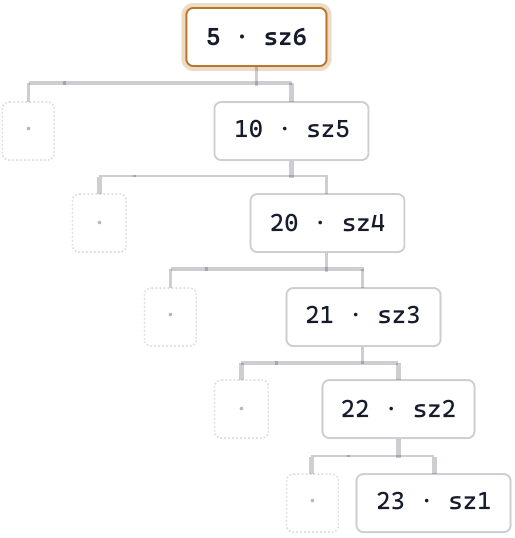
A third case — a cheap splay that actually CHARGES the battery: a different 6-node fragment. Root r has a leaf child x (a featherweight, size 1) and a 4-node subtree s that this splay never touches. Splay(x) — a single zig.

BEFORE — X IS A LIGHTWEIGHT LEAF



Φ = LOG₂6+LOG₂1+LOG₂4+LOG₂3+LOG₂2+LOG₂1 = 7.170

AFTER SPLAY(X) — X NOW OWNS THE WHOLE TREE



$\Phi = \log_2 6 + \log_2 5 + \log_2 4 + \log_2 3 + \log_2 2 + \log_2 1 = 9.492$

SPLAY	C (ACTUAL)	Φ BEFORE	Φ AFTER	ΔΦ	Ĉ = C+ΔΦ
splay(x) — **cheap, charges**	1	7.170	9.492	**+2.322**	**3.322**

READING IT — THE FULL SPECTRUM

ΔΦ < 0 → discharge: splay(1) did 5 real units of rotation work, but folded the chain enough that Φ dropped by 1.585 — the meter reads 3.415, close to $\log_2 6 \approx 2.58$, the O(log n) the Access Lemma promises. **ΔΦ = 0 → neutral:** splay(6) barely touches the tree's shape and lands exactly on the fence. **ΔΦ > 0 → charge:** a splay always drags x to the root, and the root's subtree is the whole tree — so rank(x) jumps to $\log_2 n$ no matter how small x's subtree was. If x was already a heavyweight (splay(6)'s subtree held 5 of 6 nodes), that jump is small and whoever gets demoted can lose a lot — net discharge or neutral. But if x is a featherweight leaf next to the root, splaying it is a huge rank *gain* for x while the demoted ancestor barely shrinks — net charge, exactly like Lecture 3's PUSH banking a coin for later.

The one guarantee that never varies: a genuinely *deep* splay always produces a large negative ΔΦ, because folding a long path in half is precisely what collapses many nodes' subtree sizes at once. Cheap splays near the top are what quietly top the battery back up in between — banking exactly the capacity a future deep splay will need to discharge.

The Access Lemma — the payoff, stated

THE RESULT (SLEATOR–TARJAN, 1985)

With rank r(x) = $\log_2$(size of x's subtree) and Φ = Σ r(x):

$$\text{amortized cost of splay}(x) \le 3\cdot(r(\text{root}) - r(x)) + 1 = \mathbf{O(\log n)}$$

- Hence **any sequence of m operations** on an n-node splay tree costs **O((m + n) log n)** — deterministic, no probability, any adversarial order. (Compare: naive rotate-to-root admits sequences costing Θ(m·n).)
- Where the +1 and the 3 come from: only the zig-zig / zig-zag steps make the telescoping work — the proof is exactly a Lecture-3 potential argument, three pages long, and every tool it uses — ranks, Φ, telescoping — is already yours from Lecture 3. Weiss §11.5 has the full write-up; bring questions to consultation hours.
- Bonus (beyond the syllabus): splay trees are *statically optimal* — over a long skewed sequence they perform within a constant of the best fixed tree built with full knowledge of the frequencies.

THE UNIT II SCOREBOARD SO FAR

	AVL	RED–BLACK	SPLAY
Guarantee	worst-case / op	worst-case / op	**amortized**
Height	`1.44 log n`	`2 log n`	`up to n − 1 (!)`
Extra storage	bf per node	1 color bit	**none**
Adapts to access pattern	no	no	**yes — its whole point**
Single op can be slow	no	no	yes, Θ(n)
Avoid when	write-heavy	—	hard real-time (a single slow op is unacceptable)

Three trees, three philosophies: enforce balance strictly (AVL), enforce it loosely (RB), or don't enforce it at all and let the workload shape the tree (Splay).

Three takeaways

- **One rule, no metadata:** splay whatever you touch to the root — zig at the top, grandparent-first zig-zig on straight lines, parent-first zig-zig on bends. Hot keys float; the tree becomes its own cache.
- **The ordering detail is the algorithm:** naive rotate-to-root leaves the chain a chain; true splaying folds the access path in half. Same cost paid, opposite futures.
- **Amortized O(log n) via the potential method** — the Access Lemma, with Φ = Σ log(subtree sizes). A single op may cost Θ(n); the sequence never suffers. Lecture 3's IOU: paid in full.

LECTURE 11

B-Trees, B+ Trees, B* Trees

Unit I meets Unit II: when the tree lives on disk, binary nodes waste the 4 KB block you paid a seek for. Nodes with hundreds of keys — the structure inside every database index.

HOMEWORK — BRING TO LECTURE 11

- Insert `50, 40, 60, 30, 20` into an empty splay tree, drawing the tree after each splay. Name every step (zig / zig-zig / zig-zag).
- Take the 7-chain from inserting 7, 6, 5, 4, 3, 2, 1 and splay(1). Then splay(2) on the result. Report the height after each — what do you observe?
- On the final tree of the operations demo: search 13 (present) and search 11 (absent). Show the splays both trigger.
- Delete 10 from the operations demo's final tree using top-down deletion (splay, remove, join). Show both splays.
- In 3–4 sentences, Lecture-3 vocabulary mandatory: why is "splay trees are O(log n)" a true statement and "each splay-tree operation is O(log n)" a false one?
- Reading: Weiss §4.5 (splay trees) and §11.5 (the amortized proof — skim once, then read twice).

Questions?

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

> *Next: Lecture 11 — B-Trees and Variants: the tree that runs your database.*