Comparative Analysis of Advanced Tree Structures

Lectures 7 through 14 built nine different trees, one at a time. Today we stop building and start choosing: same input, different structures, and a real answer to "which one, and why."

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

~65 minutes
Session outcome: compare balanced and indexing tree structures

Today, part by part

- **Why compare at all?** 00–04
  Nine trees, one toolbox — the real skill is picking the right one under pressure.
- **The five axes of comparison** 04–08
  Height bound, branching factor, key type, rebalancing cost, memory vs. disk.
- **Recap: the balanced-BST family** 08–13
  BST, AVL, Red-Black, Splay — one gallery, one height-bound table.
- **Recap: the indexing family** 13–18
  B-trees, Tries/DST, Segment/Interval trees, Suffix trees — what each one indexes.
- **Animation: AVL vs. Red-Black, same 6 keys** 18–30
  Insert 10,20,30,40,50,25 into both, side by side — every rotation and recolor.
- **Animation: binary height vs. B-tree height** 30–38
  Same 15 keys, two structures — search for 13, count the node visits.
- **Decision guide: scenario → structure** 38–43
  A working flowchart you can actually use in a design review.
- **The grand complexity table** 43–48
  All nine structures, one table: search, insert, delete, space, height.
- **No free lunch: trade-offs, side by side** 48–53
  Every advantage in this table is paid for somewhere else in the same row.
- **Where these actually run in production** 53–58
  Standard libraries, kernels, databases, routers, genome tools.
- **Recap & what's next** 58–65
  Lecture 16: Binary Heaps and Heap Operations.

You now own nine trees. The job is picking the right one.

THE SITUATION

Across Lectures 7–14 you built, by hand, the BST, AVL tree, Red-Black tree, Splay tree, B-tree/B+-tree/B*-tree, Trie, Digital Search Tree, Segment tree, Interval tree, and Suffix tree. Each lecture asked "how does *this one* work?" None of them asked the question an engineer actually faces first: **given this workload, which one do I reach for?**

WHY THIS MATTERS IN PRACTICE

Picking wrong is expensive, not just inelegant. A database index built as a plain binary search tree instead of a B+-tree can turn a 3-disk-read lookup into a 20-disk-read lookup — on every single query. A router forwarding table built as a sorted array instead of a trie turns an O(1)-ish prefix match into a binary search that still doesn't naturally support "longest prefix." Today builds the decision framework so that choice is deliberate, not accidental.

Every tree in this course differs along the same five dimensions

1 · HEIGHT GUARANTEE

Is worst-case height bounded at all? By how tight a constant? Unbalanced BSTs have none; AVL and RB guarantee O(log n) with different constants; Splay guarantees it only *amortized*, not per operation.

2 · BRANCHING FACTOR

Binary (2 children) vs. multiway (B-trees: dozens to hundreds of children per node). Higher branching factor means shallower trees — the whole reason B-trees exist for disk-resident data.

3 · KEY TYPE SUPPORTED

Ordered scalar keys (BST family, B-tree family) vs. string/bit keys decomposed character-by-character (Trie, DST, Suffix tree) vs. ranges/intervals (Segment tree, Interval tree).

4 · REBALANCING COST

Rotations (AVL, RB, B-tree splits), pure recoloring (RB, cheaper than a rotation), or no explicit rebalancing at all but restructuring on *every access* (Splay).

5 · MEMORY VS. DISK ORIENTATION

Designed for in-RAM pointer-chasing (BST family) vs. designed to minimize block/page reads from secondary storage (B-tree family) — a difference that dominates real-world performance far more than the Big-O alone suggests.

THE FRAMEWORK

Every comparison in this lecture is really just: where does this structure sit on these five axes, and what does that cost or buy you? Keep this list open — we'll refer back to it constantly.

BST → AVL → Red-Black → Splay: tightening (or trading away) the height guarantee

STRUCTURE	HEIGHT BOUND	REBALANCING MECHANIC	ONE-LINE CHARACTER
BST (L7)	None — O(n) worst case (a sorted-insert chain degenerates to a linked list)	None	The baseline every balanced tree is trying to fix.
AVL tree (L7–8)	$\leq 1.44 \log_2(n+2) - 0.33$ (Knuth) — the tightest of the three	Rotations on insert (≤2) and delete (up to O(log n) up the path)	Strict: every node's left/right subtree heights differ by at most 1.
Red-Black tree (L9)	$\leq 2 \log_2(n+1)$ — looser than AVL by roughly a factor of 1.4	≤2 rotations on insert, ≤3 on delete, plus cheap O(log n) recolors	Looser height rule, cheaper average fixup — the standard library default.
Splay tree (L10)	No worst-case bound per operation — a single access can visit O(n) nodes	Rotates the accessed node all the way to the root, every time (no separate "check and fix" step)	No guarantee per op, but O(log n) *amortized* over any sequence — and free adaptation to access locality (the working-set property).

All three balanced variants guarantee O(log n) search/insert/delete *in the worst case* except Splay, which trades that per-operation guarantee for automatic adaptation to real access patterns — cheap for the keys you use often, exactly the amortized-cost idea from Lecture 3's potential-method callback in Lecture 10.

B-trees, Tries/DST, Segment/Interval trees, Suffix trees: four different things to index

STRUCTURE	WHAT IT INDEXES	BRANCHING / SHAPE	SPACE
B-tree / B+-tree / B*-tree (L11)	Ordered scalar keys, optimized for block-device access	Multiway — order m means up to m children per node	O(n), but sized to fill disk blocks
Trie / Digital Search Tree (L13)	Strings or bit-patterns, decomposed character/bit by character/bit	Branching factor = alphabet size (Trie) or 2 (binary DST)	$\Theta(N \cdot \lvert\Sigma\rvert)$ worst case (Trie); more compact for DST
Segment tree / Interval tree (L12)	Ranges over a fixed universe (segment tree) or a dynamic set of intervals (interval tree)	Binary, built once over the universe or the key set	O(n)
Suffix tree (L14)	Every substring of one text, via its suffixes	Binary-to-multiway, compressed non-branching chains	O(m) guaranteed (m = text length)

Notice the pattern: every structure in this family exists because the balanced-BST family's assumption — "keys are single ordered scalars, and RAM pointer-chasing is cheap" — breaks down for *some* workload: disk-resident data, string keys, range queries, or substring queries. Indexing trees are balanced-BST ideas re-specialized for a specific kind of key or access pattern.

Insert 10, 20, 30, 40, 50, 25 into both — watch where they agree and where they diverge

This is the exact sequence from Lecture 7–8's AVL build. Every insertion is animated into **both** trees at once — including steps where one tree needs a fixup and the other doesn't, and steps that are pure recoloring with zero structural change. Nothing is skipped.

INTERACTIVE — INSERT 10, 20, 30, 40, 50, 25 INTO AN AVL TREE AND A RED-BLACK TREE, SIDE BY SIDE

STEP 1 OF 12

AVL TREE

empty

RED-BLACK TREE

empty

We insert the same 6 keys — **10, 20, 30, 40, 50, 25** — into an AVL tree and a Red-Black tree, one at a time. Same input, both trees, every rotation and every recolor shown.

STEP 2 OF 12

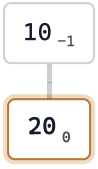
AVL TREE

RED-BLACK TREE

Insert 10. Both trees: a single root. AVL: balance factor 0. Red-Black: root is forced BLACK by the root-color rule — with one node, no violation is possible either way.

STEP 3 OF 12

AVL TREE



RED-BLACK TREE

Insert 20 as the right child of 10, in both. AVL: bf(10) becomes −1 — within tolerance, no rotation. Red-Black: new node 20 is RED; its parent 10 is BLACK, so the red-red rule isn't broken. Neither tree needs a fixup yet.

STEP 4 OF 12

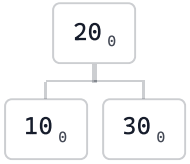
AVL TREE — BEFORE FIXUP

RED-BLACK TREE — BEFORE FIXUP

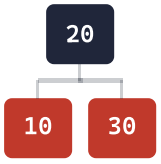
Insert 30 as the right child of 20 — a straight right-right chain. AVL: bf(10) drops to **−2**, a violation. Red-Black: 20 (RED) now has a RED child 30 — the red-red rule is broken. Both trees need a fixup.

STEP 5 OF 12

AVL TREE — AFTER FIXUP



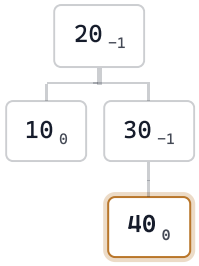
RED-BLACK TREE — AFTER FIXUP



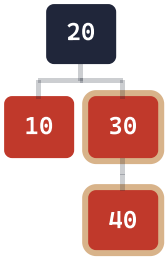
AVL fixes it with a **single left rotation at 10** (RR case) — 20 becomes the new subtree root, bf(20)=0. Red-Black fixes it with the *identical* rotation — single left rotation at 10 — then recolors: 20→BLACK, 10→RED. Same rotation, same resulting shape, different bookkeeping reason. Rotation tally so far — AVL: 1, RB: 1.

STEP 6 OF 12

AVL TREE — NO ROTATION NEEDED



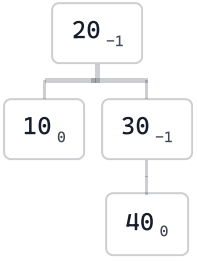
RED-BLACK TREE — BEFORE FIXUP



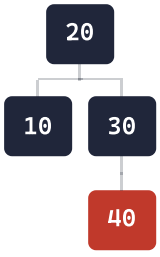
Insert 40 as the right child of 30, in both. AVL: bf(20) recomputes to −1 — balanced, **no rotation needed at all**, nothing more to do this step. Red-Black: 30 (RED) now has a RED child 40 — another red-red violation, and this time uncle 10 is also RED.

STEP 7 OF 12

AVL TREE — UNCHANGED



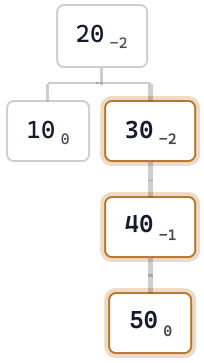
RED-BLACK TREE — AFTER RECOLOR



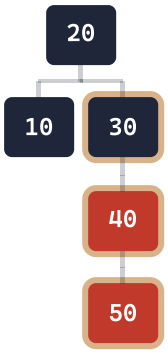
Red-Black fixes a **red uncle** violation by recoloring, not rotating: parent 30→BLACK, uncle 10→BLACK, grandparent 20 would flip to RED — but 20 is the root, so the root-color rule forces it to stay BLACK. Net effect: two nodes changed color, **zero rotations**. AVL needed zero changes this step; RB needed a recolor — cheaper than a rotation, but still not free. Rotation tally unchanged — AVL: 1, RB: 1.

STEP 8 OF 12

AVL TREE — BEFORE FIXUP



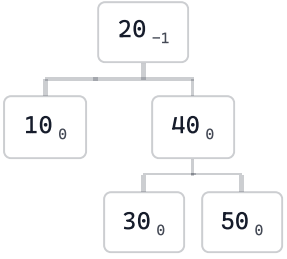
RED-BLACK TREE — BEFORE FIXUP



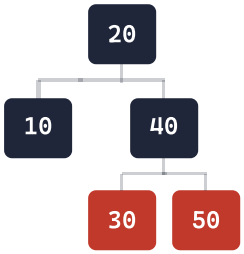
Insert 50 as the right child of 40 — another right-right chain, one level deeper. AVL: bf(30) drops to **−2**. Red-Black: 40 (RED) gets a RED child 50; uncle (30's empty left) is BLACK — this needs an actual rotation, not just a recolor.

STEP 9 OF 12

AVL TREE — AFTER FIXUP



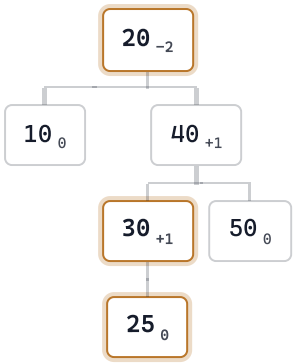
RED-BLACK TREE — AFTER FIXUP



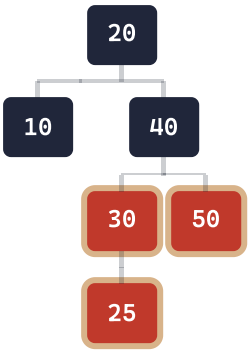
Both trees fix it the **same way**: a single left rotation at 30, promoting 40. AVL recolors nothing (it doesn't track color) — bf(40)=0. Red-Black recolors 40→BLACK, 30→RED. Rotation tally so far — AVL: 2, RB: 2. Identical so far.

STEP 10 OF 12

AVL TREE — BEFORE FIXUP



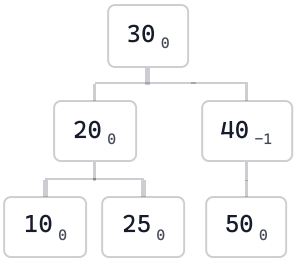
RED-BLACK TREE — BEFORE FIXUP



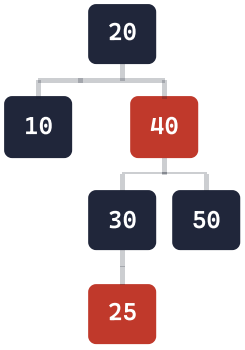
Insert 25 as the left child of 30, in both. AVL: bf(20) drops to −2 again — but this time the taller child (40, bf=+1) leans the *opposite* way: a zig-zag, not a straight line. Red-Black: 30 (RED) gets a RED child 25; uncle 50 is RED this time, not black.

STEP 11 OF 12

AVL TREE — FINAL (HEIGHT 2)



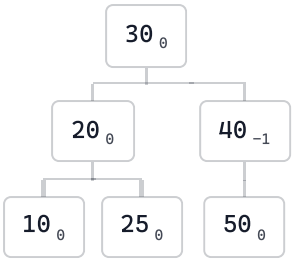
RED-BLACK TREE — FINAL (HEIGHT 3)



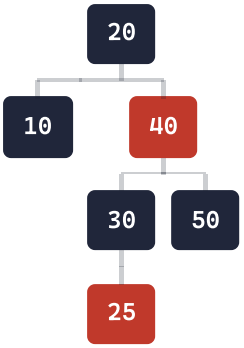
The trees genuinely diverge here. AVL needs a **double rotation** (right-rotate 40, then left-rotate 20) — its 3rd rebalancing event and its first *double* — ending with **30 as the new root**. Red-Black needs only another **recolor** (30→BLACK, 50→BLACK, 40→RED) — no rotation at all — keeping 20 as root. AVL used 3 rebalancing events (2 single + 1 double); Red-Black used only 2 rotations total, leaning on cheap recolors twice instead.

STEP 12 OF 12

AVL TREE — FINAL



RED-BLACK TREE — FINAL



AVL height: 2

AVL rebalances: 3 (2 single + 1 double)

RB height: 3

RB rotations: 2 + 2 recolor-only fixups

Same 6 keys, two different final shapes. AVL guarantees the tighter height bound ($\approx 1.44 \log_2 n$) but paid for it with 3 rebalancing events, including one costlier double rotation. Red-Black is one level taller but needed only 2 rotations total, leaning on cheap recolors twice. This is exactly why most language standard libraries (C++ `std::map`/`std::set`, Java `TreeMap`/`TreeSet`) default to red-black trees: general-purpose insert/delete-heavy workloads favor cheaper fixups over the absolute tightest height.

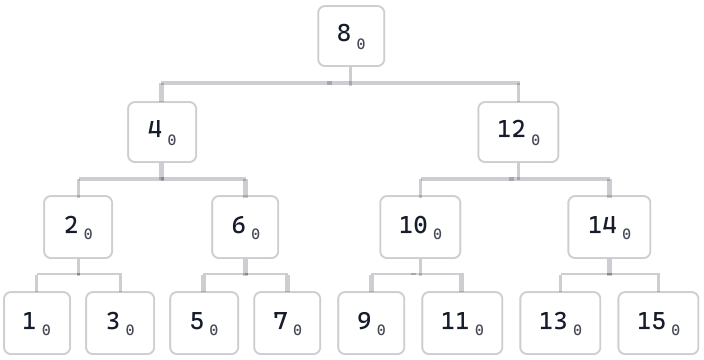
Same 15 keys, two structures — search for 13, count the node visits

Keys 1–15 stored two ways: a perfectly balanced AVL tree (binary, height 3) and an order-5 B-tree (up to 4 keys / 5 children per node, height 1). Watch both search for the same key, one node-visit at a time — in a disk-resident structure, each visit is a potential disk read.

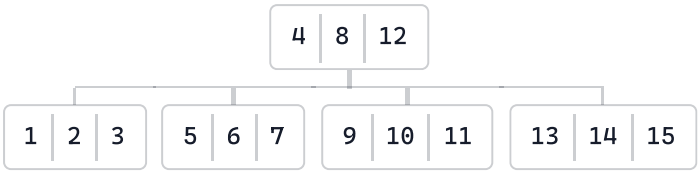
INTERACTIVE — SEARCH BOTH STRUCTURES FOR KEY 13

STEP 1 OF 6

AVL TREE — 15 KEYS, HEIGHT 3



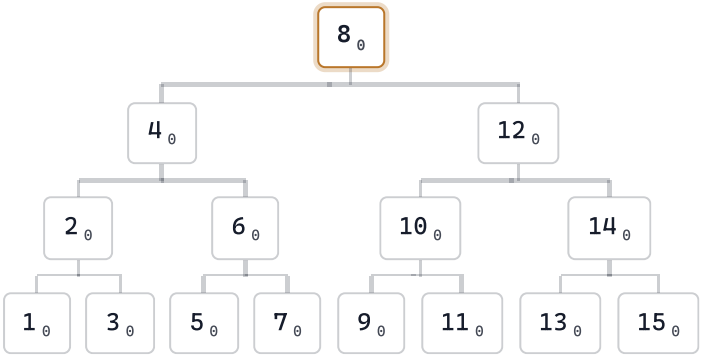
ORDER-5 B-TREE — 15 KEYS, HEIGHT 1



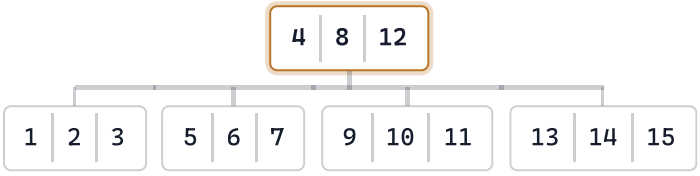
Same 15 keys (1–15), two structures: a perfectly balanced AVL tree (binary, height 3) and an order-5 B-tree (up to 4 keys / 5 children per node, height 1). We'll search both for **key 13** and count node visits — in a disk-resident structure, each visit is a potential disk read.

STEP 2 OF 6

AVL TREE — VISIT 1



B-TREE — VISIT 1

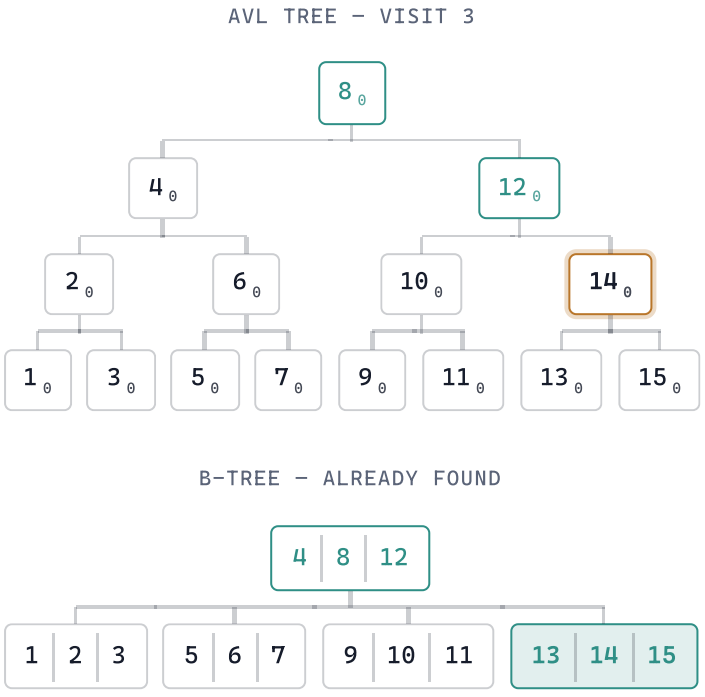


Step 1 — both start at the root. AVL: compare 13 vs. 8 → 13>8, go right. B-tree: scan the root's 3 keys [4,8,12] → 13 is greater than all of them, descend into the 4th (rightmost) child.

STEP 3 OF 6

Step 2. AVL: compare 13 vs. 12 → 13>12, go right (one more level to go). B-tree: we've already reached a **leaf** — scan its 3 keys [13,14,15] → 13 is found immediately. **B-tree search done in 2 node visits.**

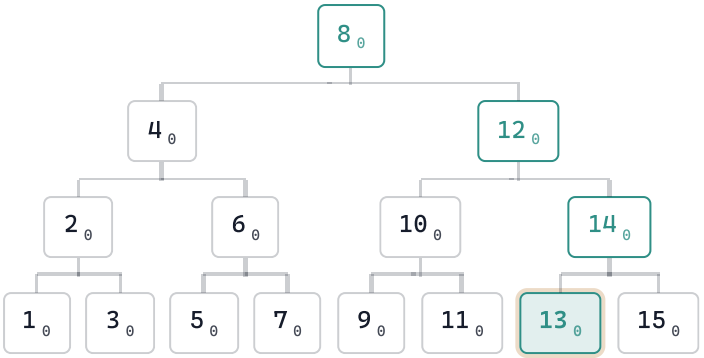
STEP 4 OF 6



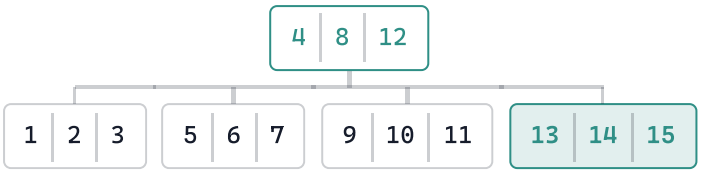
Step 3. AVL: compare 13 vs. 14 → 13<14, go left. Still not found. The B-tree already finished — holding at its answer while AVL keeps working.

STEP 5 OF 6

AVL TREE — VISIT 4 (FOUND)



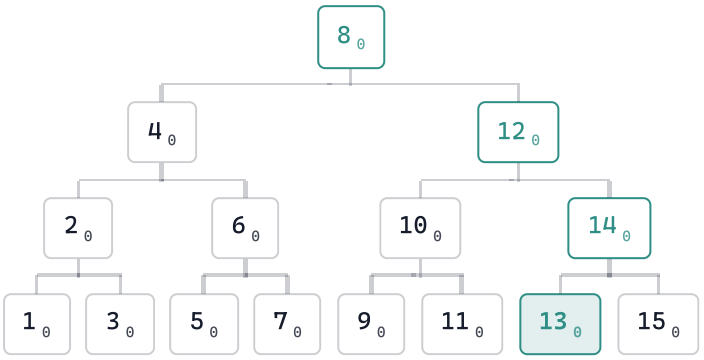
B-TREE — ALREADY FOUND



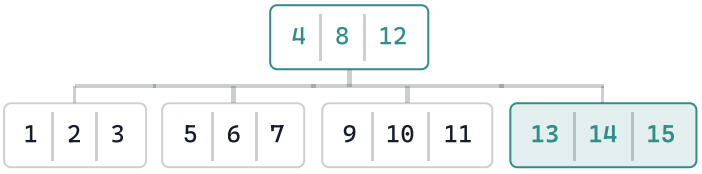
Step 4. AVL: compare 13 vs. 13 → found. AVL needed **4 node visits** across 3 levels of comparisons; the B-tree needed only **2**, because each B-tree node packs up to 4 keys and 5 children into a single visit.

STEP 6 OF 6

AVL TREE — FINAL



B-TREE — FINAL



`AVL visits: 4 (height 3)` `B-tree visits: 2 (height 1)`

`Same 15 keys — 2× fewer visits for the B-tree`

For 15 keys the gap is already visible; it only grows. For n≈1,000,000 keys, a balanced binary tree needs roughly $\log_2 n \approx 20$ levels (up to ≈29 in AVL's guaranteed worst case), while an order-100 B-tree needs only about 4. When each node visit is a disk seek costing milliseconds, that difference is the entire reason database and filesystem indexes are built as B-trees/B+-trees, never as binary search trees.

Scenario in, structure out

IF YOUR WORKLOAD LOOKS LIKE…	…REACH FOR	BECAUSE
General-purpose in-memory ordered map/set, frequent insert and delete	Red-Black tree	Cheapest average rebalancing cost among the guaranteed-O(log n) options
Lookup-heavy workload, insertions/deletions rare after initial build	AVL tree	Tightest height bound → fewer comparisons per search, the cost you actually pay repeatedly
Access pattern is skewed / has strong temporal locality (caches, LRU-like use)	Splay tree	Working-set property — recently touched keys become cheap automatically
Data lives on disk/SSD — database index, filesystem directory	B-tree / B+-tree	High branching factor minimizes the number of block reads, which dominates real latency
Keys are strings and you need prefix search, autocomplete, or routing	Trie (or compressed trie / DST for tighter space)	Cost depends on key length, not on how many keys are stored
Repeated "which intervals overlap X" or "sum/max over range [lo,hi]" queries	Interval tree (dynamic interval set) or Segment tree (fixed range universe)	Purpose-built augmentation (max_high, canonical decomposition) makes these O(log n) instead of an O(n) scan
Repeated substring / pattern-matching queries against one large fixed text	Suffix tree (or suffix array in production)	O(p) search per pattern regardless of text length, after one O(m) build

All nine structures, one table

STRUCTURE	SEARCH	INSERT	DELETE	SPACE	HEIGHT / DEPTH
BST (unbalanced)	O(h)	O(h)	O(h)	O(n)	O(n) worst case
AVL tree	O(log n)	O(log n)	O(log n)	O(n)	$\leq 1.44 \log_2 n$
Red-Black tree	O(log n)	O(log n)	O(log n)	O(n)	$\leq 2 \log_2 n$
Splay tree	O(log n) amortized	O(log n) amortized	O(log n) amortized	O(n)	no per-op bound
B-tree / B+-tree (order m)	O(log_t n), t=⌈m/2⌉	O(log_t n)	O(log_t n)	O(n)	very small — grows with branching factor
Trie	O(L)	O(L)	O(L)	$\Theta(N \cdot \lvert\Sigma\rvert)$ worst case	O(L), independent of n
Digital Search Tree	O(b)	O(b)	O(b)	O(n·b) worst case	O(b), b = key bit-length
Segment tree	O(log n) per query	O(log n) per point update	O(log n)	O(n)	O(log n), fixed universe
Interval tree	O(log n + k) to report k overlaps	O(log n)	O(log n)	O(n)	O(log n)
Suffix tree	O(p), pattern length p	O(m) full build	rarely used dynamically	O(m)	O(m) worst case, typically far less

L = key length (Trie), b = key bit-length (DST), m = text length (Suffix tree), n = number of keys, h = tree height, t = B-tree minimum degree. Every one of these numbers was independently derived and hand-verified in its own lecture — this table just puts them side by side.

Every advantage in the last table is paid for somewhere else in the same row

ADVANTAGES, RESTATED AS TRADE-OFFS

- **AVL's tight height** costs more rotations per insert/delete than Red-Black — verified numerically in this lecture's own head-to-head (3 rebalancing events vs. 2, for the same 6 keys).
- **B-trees' shallow height** costs wasted space in partially-full nodes (a node can legally be as little as half full) and more complex split/merge logic than a binary rotation.
- **Tries' O(L) search** costs $\Theta(N \cdot |\Sigma|)$ space in the worst case — fixed by compression (Lecture 13), which then costs implementation complexity.
- **Suffix trees' O(p) pattern search** costs a genuinely intricate O(m) build (Ukkonen's algorithm) — which is exactly why suffix *arrays* often win in production despite being conceptually simpler, not more powerful.
- **Splay trees' automatic adaptation** costs any worst-case per-operation guarantee at all — one unlucky access can legitimately cost O(n).

THE ONE-SENTENCE VERSION

Every structure in this course is a different answer to the same question — "where should the cost go?" — and the right answer always depends on which operation your workload actually calls the most, not on which structure looks the most sophisticated on a slide.

Not a hypothetical exercise — these choices are load-bearing in real systems

RED-BLACK TREE

C++'s `std::map`/`std::set` and Java's `TreeMap`/`TreeSet` are implemented as red-black trees in their standard library implementations. The Linux kernel uses red-black trees for the CFS process scheduler's run-queue and for tracking virtual memory areas.

B+-TREE

The default on-disk index structure for most relational databases (e.g. InnoDB's clustered indexes in MySQL) and many filesystems — high branching factor is chosen specifically to match the disk block size, minimizing seeks.

TRIE / COMPRESSED TRIE

IP routers use trie-like structures (compressed/PATRICIA tries) for longest-prefix-match forwarding lookups; search engines use tries for autocomplete and prefix-based term indices.

SPLAY TREE

Well-suited to caches and any workload with strong temporal locality — recently or frequently accessed keys migrate toward the root and stay cheap, with no separate cache-eviction bookkeeping needed.

SEGMENT / INTERVAL TREE

Computational-geometry engines, scheduling systems ("which meetings overlap this time slot?"), and graphics/collision-detection pipelines that need fast range or overlap queries.

SUFFIX TREE / ARRAY

Bioinformatics tools for genome alignment and repeat-finding; full-text search engines and version-control diff tools that need fast substring queries over large fixed texts.

Same problem, nine tools — the choice is now yours to defend

- **Five axes decide everything:** height guarantee, branching factor, key type, rebalancing cost, and memory-vs-disk orientation. Every structure in this course is a specific point in that five-dimensional space.
- **Verified today, not just asserted:** inserting 10,20,30,40,50,25 into AVL vs. Red-Black gives final heights 2 vs. 3, using 3 rebalancing events vs. 2 — a real, traceable instance of "tighter balance costs more restructuring."
- **Verified today, not just asserted:** the same 15 keys need 4 node visits to find 13 in a balanced binary tree, but only 2 in an order-5 B-tree — and that gap only widens as n grows, which is the entire justification for disk-oriented indexing structures.
- **No structure dominates on every axis.** Picking the right one means naming the workload's dominant operation first, then matching it against the decision guide and the grand complexity table.

LECTURE 16 — NEXT

Binary Heaps and Heap Operations

From "find/compare a specific key" to "always give me the extreme one" — a different family problem, a different structure.

HOMEWORK — BRING TO LECTURE 16

- Insert the keys 5, 3, 8, 1, 4, 7, 9, 2, 6 into both an AVL tree and a Red-Black tree by hand. Count rotations and recolors for each. Which used fewer structural changes?
- For the same 9 keys, compare the resulting AVL/RB height against an order-4 B-tree built on the same keys. How many node visits does each need to find key 6?
- Pick any one real system you use daily (a phone's contacts app, a text editor's autocomplete, a map app's routing). Which tree family from this course would you guess sits underneath it, and why?
- In 3–4 sentences: why does "amortized O(log n)" (Splay tree) not mean the same guarantee as "worst-case O(log n)" (AVL/RB)? Give a concrete access sequence where the difference would show up.

Questions?

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

> *Next: Lecture 16 — Binary Heaps and Heap Operations.*