B-Trees and Their Variants

One family of trees, built around a single physical fact: touching a disk is a hundred-thousand times slower than touching RAM. B-trees, B+ trees, and B* trees are three answers to the same question — how do you stay balanced while paying for that as rarely as possible?

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

40 minutes
Session outcome: explain B-Trees, B+ Trees, and B* Trees

Today, minute by minute

Refresher: why disk breaks the O(log2n) promise 00–04

AVL/RB height is fine in RAM; on disk, every node is a seek. The printed-dictionary analogy.

B-tree definition: order, occupancy, uniform depth 04–07

The rules, stated once, that every animation below just applies.

Animation: building an order-3 B-tree, keys 1–10 07–14

Every insertion, every split, the cascading double-split at key 7.

Animation: deleting from that same tree 14–21

Simple delete, merge, borrow-through-parent, and the tree shrinking a level.

Why B+ trees? The range-query weakness 21–23

Data scattered across a B-tree makes "give me everything from 20 to 80" expensive.

B+ tree structure + animation: same 1–7, rebuilt 23–29

Index copies vs real data; copy-up vs move-up; the linked leaf chain.

Animation: B+ deletion and the index-update rule 29–32

Borrowing across leaves means fixing the signpost above them too.

B-tree vs B+ tree, side by side 32–34

Where data lives, and why it decides everything else.

Why B* trees? The wasted-space problem 34–36

Half-full nodes are the norm, not the exception, after enough deletes.

Animation: B* insertion — redistribute first, split only if forced 36–40

The easy case (sibling has room) and the hard case (two-to-three split), both on one tree.

Animation: B* deletion — borrow, then the three-into-two collapse 40–45

Why two min-occupied nodes can't just merge into one — and what happens instead.

Recap & what's next 45–47

One idea worn three ways. Lecture 12: Segment and Interval Trees.

Why balanced isn't enough once data lives on disk

REFRESHER — WHAT O(LOG N) ACTUALLY COST YOU

Lectures 7–9: AVL and Red-Black trees hold height to $O(\log_2 n)$ by capping every node at 2 children. For n = 1,000,000 records that's a height of about 20 — brilliant, as long as the whole tree sits in RAM.

But databases, filesystems, and search indexes don't fit in RAM. They live on disk. And "one node visited" on a root-to-leaf path now means "one possible trip to the disk" — which changes the arithmetic completely.

THE DISK FACT THAT CHANGES EVERYTHING

A RAM access takes on the order of 100 nanoseconds. A disk seek takes 5–10 *milliseconds* — roughly **100,000× slower**. A height-20 binary tree can mean 20 seeks per search: at 8ms each, that's 160ms for one lookup — brutal for a system answering thousands of queries a second.

And disks don't read "one key" at a time — they read in fixed-size **blocks** (typically 4–8 KB) no matter how little you asked for. So the right size for "one tree node" isn't one key; it's *as many keys as fit in one block*.

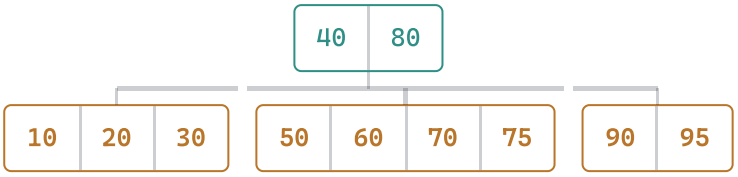
Analogy — looking up a word in a printed dictionary: imagine an absurd dictionary printed with **one word per page**. Finding "MERGE" means flipping to a page, reading its single word, deciding "earlier or later?", and flipping again — around 17 flips for 100,000 words, and every flip is physical work. That is a binary tree on disk: one key per node, one seek per node. A real dictionary instead packs **hundreds of sorted words on each page**, with guide words at the top corner. You flip to a page (that's the trip — the expensive part), then scan dozens of entries *on* the page for free — your eyes are already there — and the guide words tell you exactly which page to flip to next. Three or four flips finds any word among 100,000. That wide, shallow, many-keys-per-page shape is exactly a **B-tree**: page = disk block, page-flip = disk seek, within-page scanning = free RAM comparisons. Trade tree height for node width, because on disk it's the number of *trips* that costs you, not the number of comparisons inside a trip.

A search tree where every node is a full disk block

PROPERTIES OF AN ORDER-M B-TREE

- Every node holds up to **m − 1** sorted keys and up to **m** children.
- A node with k keys has exactly k + 1 children; child i holds every key strictly between key i−1 and key i — the same "between the keys" rule from ordinary BSTs, just generalized past 2 children.
- Every node except the root must have at least **⌈m/2⌉ children** (at least ⌈m/2⌉ − 1 keys) — no node may be less than half full.
- Every **leaf sits at exactly the same depth** — the tree never gets lopsided.

EVERY PROPERTY, VISIBLE ON ONE LEGAL ORDER-5 TREE



ORDER 5 · MIDDLE LEAF FULL (4 KEYS) · RIGHT LEAF AT THE MINIMUM (2 KEYS) · ALL LEAVES SAME DEPTH

- Order 5 ⟹ max **4 keys** per node — the middle leaf [50 60 70 75] is exactly full.
- Min ⌈5/2⌉ = 3 children ⟹ min **2 keys** — the right leaf [90 95] sits at the legal minimum; one key fewer and it would violate the half-full rule.
- Root [40 80] has 2 keys ⟹ exactly **3 children**, and each child's keys fall strictly in its gap: <40, between 40 and 80, >80.
- All three leaves at the **same depth** — count it.

Order 3 as the working example — and search, step by step

THE ORDER USED IN THIS LECTURE'S EXAMPLES

All worked examples in this lecture use **m = 3**, the smallest non-trivial order: max 2 keys / 3 children per node, min 1 key / 2 children (except the root). A small order keeps every split and merge fully visible within a few steps. Production database and filesystem B-trees use m in the hundreds or thousands, sized so one node = exactly one disk block.

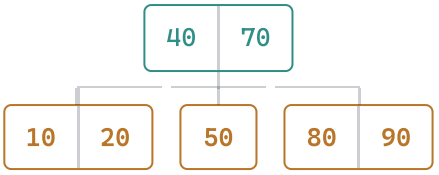
The insertion and deletion *rules* are identical for every order. Master the order-3 traces that follow, and the same procedure applies unchanged at order 500 in production — only the arithmetic scales, not the idea.

SEARCH — THE ONE OPERATION THAT NEEDS NO NEW MACHINERY

At each node, scan its sorted keys to find which of the k + 1 gaps your target falls into, then descend into that child. Repeat until found — or until you're standing in a leaf with nowhere left to descend (a definitive "not present"). The animation traces both outcomes on an order-3 tree. The only genuinely *new* machinery in this lecture is keeping every node between half-full and full as you insert and delete — that's what the next two animations handle.

INTERACTIVE — SEARCH 50 (A HIT), THEN SEARCH 85 (A MISS)

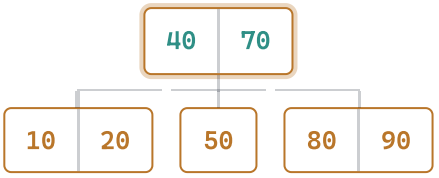
STEP 1 OF 6



ORDER 3 · ROOT [40 70] HAS 2 KEYS, SO EXACTLY 3 CHILDREN · EVERY LEAF AT DEPTH 1

The tree we'll search. Root [40 70] splits the key space into 3 gaps: <40 · between 40 and 70 · >70 — each gap owned by one child. First query: **SEARCH(50)**.

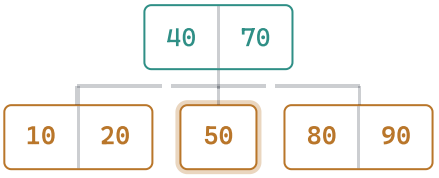
STEP 2 OF 6



VISIT 1 — THE ROOT BLOCK IS READ FROM DISK

SEARCH(50), at the root: scan the sorted keys left to right. 50 > 40 — keep scanning. 50 < 70 — stop. So 50 falls in the **middle gap** (between 40 and 70) → descend into child 2. That scan cost a few RAM comparisons; the node visit cost **one disk trip**.

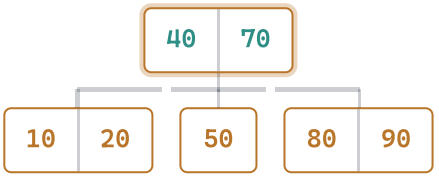
STEP 3 OF 6



VISIT 2 — ONE MORE DISK TRIP

At leaf [50]: scan — first key is 50. **FOUND.** Total cost: 2 node visits = **2 disk trips**, regardless of how many comparisons happened inside each block. Now the other outcome: **SEARCH(85)**.

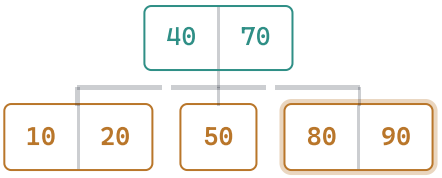
STEP 4 OF 6



VISIT 1 — BACK AT THE ROOT

SEARCH(85), at the root: 85 > 40 — keep scanning. 85 > 70 — past the last key. So 85 falls in the **rightmost gap** (>70) → descend into child 3.

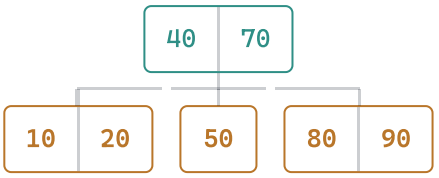
STEP 5 OF 6



VISIT 2 — THE RIGHTMOST LEAF

At leaf [80 90]: 85 > 80 — keep scanning. 85 < 90 — stop: 85 falls in the gap between 80 and 90. But this is a **leaf** — the gap has no child to descend into. **NOT FOUND, definitively:** if 85 existed anywhere, the gap rule would have led us exactly here.

STEP 6 OF 6



EVERY SEARCH — HIT OR MISS — COSTS HEIGHT + 1 = 2 TRIPS ON THIS TREE

The takeaway: both searches cost exactly the same — (height + 1) node visits, each one disk trip. Scale it up: with m in the hundreds, a **billion** keys fits in height 3–4, so any search is 4–5 disk trips. That number — trips, not comparisons — is the whole reason this structure exists.

Insert 1 through 10 into an order-3 B-tree — watch every split

THE ONLY RULE YOU NEED

```
insert k into its correct leaf (descend by
    the SEARCH of L11·02a: at each node, scan
    the sorted keys, take the gap k falls in)

if that leaf now holds m keys (one too many):
    split it at the median:
        left half keeps the smaller keys
        right half keeps the larger keys
        median key moves UP into the parent
    // if the parent now overflows too, split IT
    // the same way — this can cascade to the root
    // if the root overflows, a NEW root is created
    // and the tree grows one level taller — upward
```

HOW TO READ THE ANIMATION

Order 3 means **max 2 keys** per node. A node briefly holding 3 keys is shown with a **dashed red border** — overflow, must split now. **Amber** boxes are leaves; **teal** boxes are internal (routing) nodes. Watch the median: it always escapes *upward*, never sideways.

INTERACTIVE — INSERT 1, 2, 3, … , 10 ONE KEY AT A TIME

STEP 1 OF 16

∅ — EMPTY TREE

Start: empty order-3 B-tree (max 2 keys / 3 children per node). Insert 1.

STEP 2 OF 16

Insert 1: becomes the root, a single leaf [1].

STEP 3 OF 16

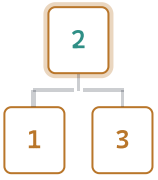
Insert 2: still one leaf. [1,2] — exactly at the 2-key limit, no overflow yet.

STEP 4 OF 16

LEAF HAS 3 KEYS — OVERFLOW (ORDER 3 MAX IS 2) — MUST SPLIT NOW

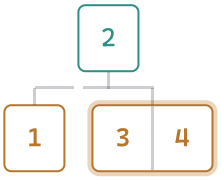
Insert 3: the leaf would hold [1,2,3] — 3 keys, over the order-3 limit of 2. Overflow (dashed red) — split now.

STEP 5 OF 16



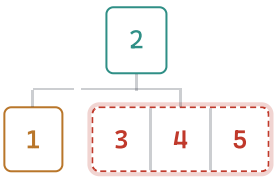
Split at the median: **2** is promoted to a brand-new parent; 1 and 3 become its two children.

STEP 6 OF 16



Insert 4: 4 > 2, so it lands in the right leaf → [3,4]. Still within the limit, no split.

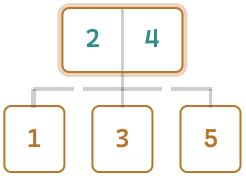
STEP 7 OF 16



LEAF HAS 3 KEYS — OVERFLOW — MUST SPLIT NOW

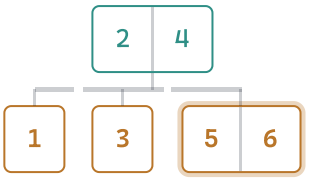
Insert 5: the right leaf would hold [3,4,5] — overflow (dashed red). Split now.

STEP 8 OF 16



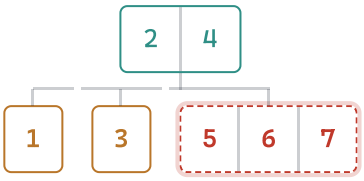
Split: median **4** rises into the root, which now holds [2,4] — two keys, still legal.

STEP 9 OF 16



Insert 6: goes to the rightmost leaf → [5,6]. No split.

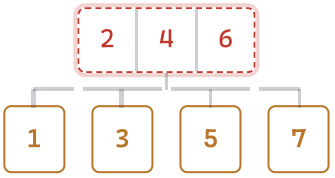
STEP 10 OF 16



LEAF HAS 3 KEYS — OVERFLOW — MUST SPLIT NOW

Insert 7: the rightmost leaf would hold [5,6,7] — overflow (dashed red). Split now.

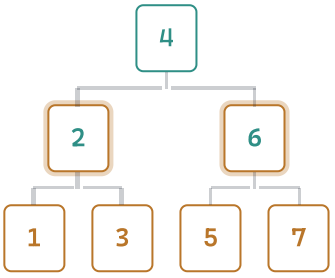
STEP 11 OF 16



ROOT NOW HAS 3 KEYS TOO — OVERFLOW, MUST SPLIT NEXT

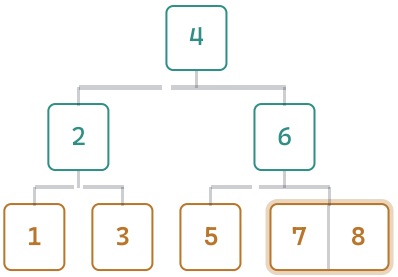
Split the leaf: median **6** rises. But the root already held [2,4] — now [2,4,6], three keys. The *root itself* overflows too (dashed red) — split it next.

STEP 12 OF 16



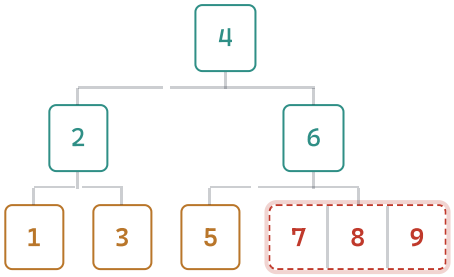
Splitting an overflowing root always grows the tree by one level: median **4** becomes the new root; [2] and [6] become its two children, each keeping the leaves it already owned. This is the *only* way a B-tree gets taller — always at the root, always keeping every leaf at the same depth.

STEP 13 OF 16



Insert 8: root sends it right (8 > 4), node 6 sends it right (8 > 6) → leaf [7] becomes [7,8]. No split.

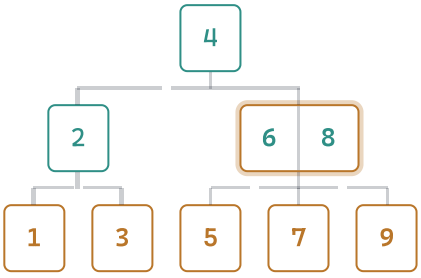
STEP 14 OF 16



LEAF HAS 3 KEYS — OVERFLOW — MUST SPLIT NOW

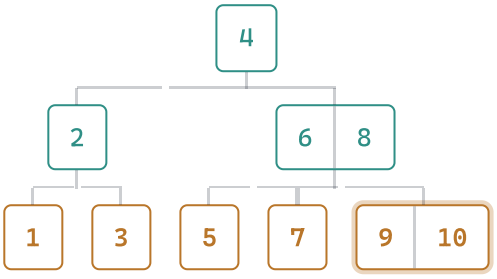
Insert 9: the leaf would hold [7,8,9] — overflow (dashed red). Split now.

STEP 15 OF 16



Split: median **8** rises into its parent, which becomes [6,8] — exactly 2 keys, no further overflow this time.

STEP 16 OF 16



Insert 10: lands in the rightmost leaf [9] → [9,10]. All ten keys are placed, and the tree height never grew past 3 levels. That uniform shortness is the entire promise of a B-tree.

Delete 10, 9, 5, 4 — every deletion case, one tree

THE FOUR CASES, IN THE ORDER YOU'LL MEET THEM

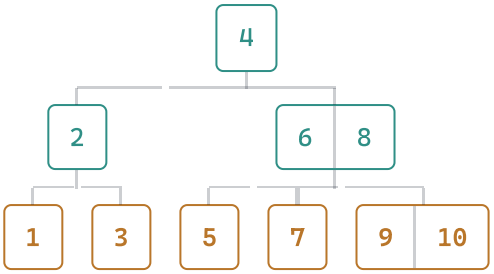
- **No underflow:** the leaf still has ≥ 1 key after removal — done.
- **Merge:** leaf empties, sibling is also at the minimum — merge them, pulling the parent's separator key down between them.
- **Borrow through the parent:** leaf empties, but a sibling has a spare key — rotate one key down from the parent and one up from the sibling.
- **Delete an internal key → successor swap, then repair:** a key can only be physically removed at a leaf, so replace it with its inorder successor (always in a leaf), then fix that leaf's underflow — the repair can cascade upward and shrink the tree by a whole level.

WHY THIS ORDER WAS CHOSEN — AND HOW TO READ THE ANIMATION

This is the exact same tree you just built. Deleting 10 → 9 → 5 → 4 in sequence walks you through all four cases without ever introducing a new example — by the last step you'll have personally traced why deleting one internal key can ripple all the way to the root. Every removal is shown as its own step *before* the repair: a **dashed red** node marks **underflow** — a node below its key minimum (· = empty) that the next step must fix.

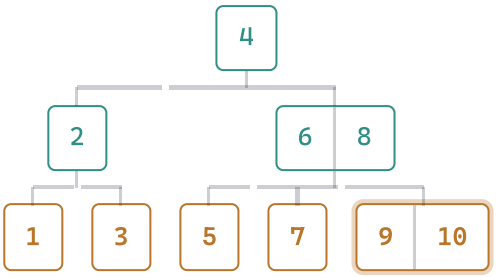
INTERACTIVE — DELETE 10, THEN 9, THEN 5, THEN 4, THEN 7, THEN 3

STEP 1 OF 16



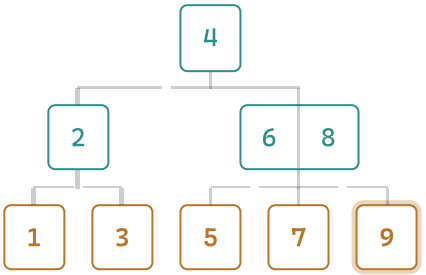
Starting tree after all ten insertions. Now delete 10, then 9, then 5, then 4 — one at a time — to meet every deletion case a B-tree can throw at you. Every removal is its own step; every repair is the step after.

STEP 2 OF 16



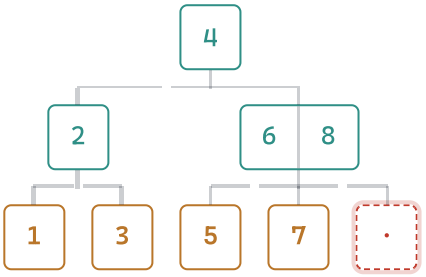
Delete 10 — locate it: search descends root [4] (10 > 4 → right), node [6,8] (10 > 8 → right) → leaf [9,10]. The leaf holds 2 keys and the minimum is 1, so removing one key keeps it legal.

STEP 3 OF 16



Delete 10 — done: [9,10] → [9]. Still ≥ 1 key: no rebalancing needed at all. Simplest possible case.

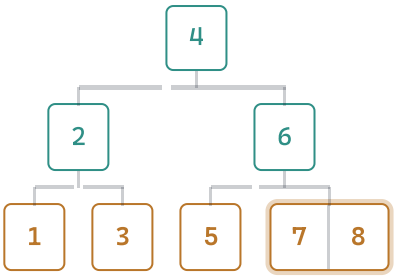
STEP 4 OF 16



DASHED RED = UNDERFLOW: 0 KEYS, MINIMUM IS 1 — MUST BE REPAIRED BEFORE ANYTHING ELSE

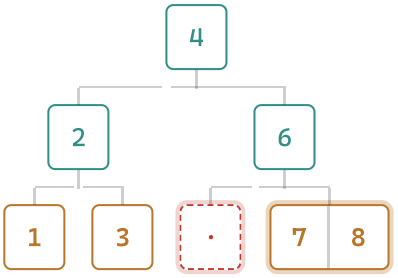
Delete 9 — remove it first: the leaf empties to **zero keys**, below the 1-key minimum — **underflow** (dashed red). The tree may not stay in this state. Diagnose: can the sibling [7] lend a key?

STEP 5 OF 16



Repair — merge: sibling [7] has only 1 key, the bare minimum — it cannot lend without underflowing itself. So **merge**: the empty leaf, the parent's separator key 8 (pulled down), and [7] fuse into [7,8]. The parent shrinks from [6,8] to [6].

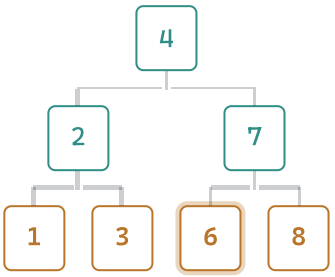
STEP 6 OF 16



UNDERFLOW AGAIN — BUT LOOK AT THE SIBLING BEFORE CHOOSING THE REPAIR

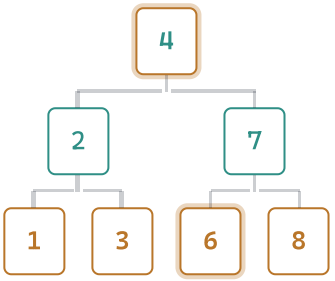
Delete 5 — remove it first: its leaf empties → **underflow** again. Same diagnosis question, different answer this time: the sibling [7,8] holds 2 keys — **one to spare**.

STEP 7 OF 16



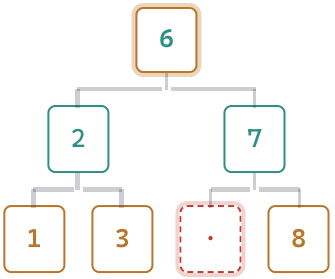
Repair — borrow through the parent: the parent's key 6 drops into the empty leaf, and the sibling's smallest key 7 rises into the parent's place. No merge, no shrinking — the tree keeps its shape. Never merge when a borrow is available.

STEP 8 OF 16



Delete 4 — the hard one: it lives in the **root**, an internal key. A key can only be physically removed at a **leaf** — the same principle as Lecture 7's BST delete. (Note that merging 4 with its children is NOT an option: [2]+4+[7] would make a 3-key node, over the order-3 maximum of 2.) So find 4's **inorder successor**: descend right to [7], then keep left → leaf [6]. Successor = **6**.

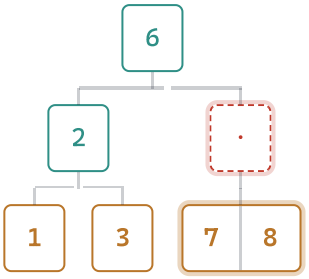
STEP 9 OF 16



ROOT IS LEGAL — BUT THE SUCCESSOR'S LEAF IS NOW EMPTY: UNDERFLOW

Successor swap: copy 6 up to replace 4, and remove 6 from its leaf. The root [6] is perfectly legal — but the leaf just underflowed to zero keys (dashed red). From here it is the **same repair machinery as before**, applied bottom-up.

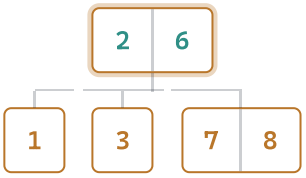
STEP 10 OF 16



THE UNDERFLOW HAS CLIMBED: NOW AN INTERNAL NODE IS BELOW MINIMUM

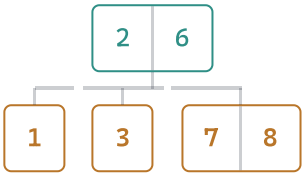
Repair the leaf — merge: its sibling [8] is at the bare minimum, so no borrow. Separator 7 drops down and fuses with [8] → [7,8]. But that pulled the only key out of the internal node above it — **the underflow climbed one level** (compare Lecture 9's double-black debt climbing the tree).

STEP 11 OF 16



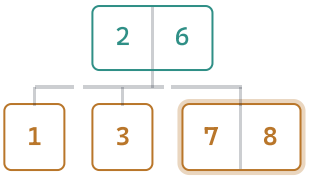
Repair the internal node — merge again: its sibling [2] is also at the minimum → merge [2], the root's separator 6, and the empty node into **[2,6]**, adopting the children [1], [3], [7,8] — 2 keys, 3 children: legal. The old root emptied out, so the merged node *is* the new root — the tree shrinks one level.

STEP 12 OF 16



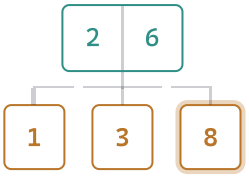
Delete 4 — done. Final tree: root [2,6] over leaves [1], [3], [7,8] — inorder 1 2 3 6 7 8 ✓, every node within its limits, all leaves still at the same depth. Notice deletion never created an over-full node: internal-key deletion = **successor swap at a leaf + bottom-up underflow repairs**, which can cascade to the root and shrink the tree — the exact mirror of insertion growing it at the root.

STEP 13 OF 16



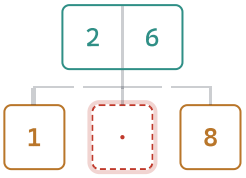
One more round — delete 7: search descends root [2,6] (7 > 6 → right) → leaf [7,8]. Found. The leaf holds 2 keys, minimum is 1 — removing one keeps it legal.

STEP 14 OF 16



Delete 7 — done: [7,8] → [8]. Still ≥ 1 key: no rebalancing needed. The everyday case, one more time.

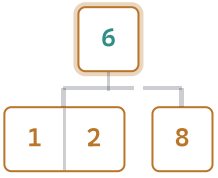
STEP 15 OF 16



UNDERFLOW — BUT THIS TIME BOTH SIBLINGS ARE ALREADY AT THE MINIMUM

Delete 3: root [2,6] → 3 sits between 2 and 6 → middle leaf [3]. Removing it empties the leaf — **underflow**. Diagnose both neighbors this time: left sibling [1] holds 1 key, right sibling [8] holds 1 key — **both already at the minimum**. Neither can lend.

STEP 16 OF 16



No borrow available on either side → merge. Fuse the empty leaf with the LEFT sibling [1], pulling down the root's separator 2 → **[1,2]**. The root loses that key and that child, shrinking from [2,6] to just [6] — still legal, since even the root only needs ≥1 key. **Final tree: root [6], children [1,2], [8]** — inorder 1, 2, 6, 8 ✓. (Merging with the RIGHT sibling [8] instead — pulling down 6 — would be equally legal, giving root [2] over [1] and [6,8]. Which side to prefer when both are tied at the minimum is an implementation choice, not something the algorithm is forced into.)

One weakness of a plain B-tree: range queries

THE PROBLEM

In a B-tree, real data can sit at **any** node — root, internal, or leaf. So answering "give me every record between 20 and 80" means jumping in and out of internal nodes with no shortcut between neighboring leaves. Yet this exact kind of query is everywhere: `SELECT … WHERE age BETWEEN 20 AND 80`, listing a directory alphabetically, streaming a large file's blocks in order.

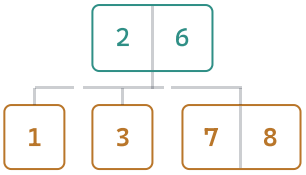
THE B+ TREE FIX

Force **all real data into the leaves only**. Internal nodes hold nothing but routing *copies* of keys — pure signposts, never touched by a range scan. Then thread every leaf to its right neighbor with a pointer. Once you've found where a range starts, you just walk the linked leaves left to right. This is exactly how MySQL InnoDB, PostgreSQL, and the NTFS/ext4 directory indexes work — B+ trees, not plain B-trees.

See it, not just read it: run the identical range query — every key from 2 to 7 — on two order-3 trees holding the same six keys (1, 2, 3, 6, 7, 8). The left tree is exactly where Section 04's deletion animation left off; the right tree is a preview of the B+ shape Section 07 builds properly from scratch. Step through both and watch which one ever has to climb back up.

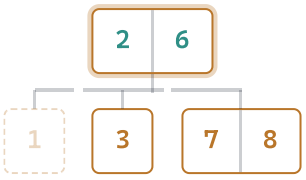
B-TREE — RANGE QUERY [2,7]

STEP 1 OF 5



Range query [2,7] on the B-tree: report every key k with 2 ≤ k ≤ 7. This is exactly the tree Section 04's deletion animation ended with. Data can live at ANY node — root, internal, or leaf — so watch which levels we have to revisit.

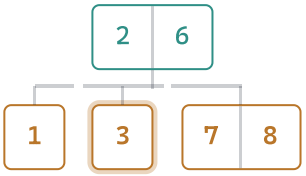
STEP 2 OF 5



DISK TRIP 1 — LEFT CHILD [1] PRUNED, DASHED: NOTHING <2 CAN QUALIFY

Visit root [2,6]: both 2 and 6 fall in [2,7] → report both. Since nothing less than 2 can qualify, the left child [1] (which only holds keys < 2) is skipped entirely — no trip there at all.

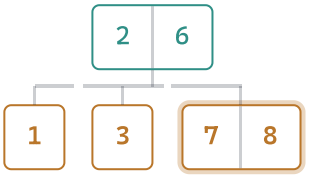
STEP 3 OF 5



DISK TRIP 2

Descend to the middle child, leaf [3]: the gap between 2 and 6 could still hold qualifying keys, so we go down. 3 falls in [2,7] → report it. It's a leaf — nothing further to descend into.

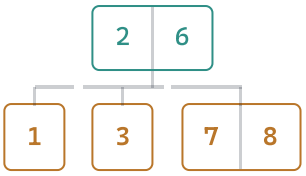
STEP 4 OF 5



DISK TRIP 3 — BACK THROUGH THE ROOT TO GET HERE

Back up to the root, then down the right child, leaf [7,8]: since 7 ≤ 7, this gap could still qualify → descend. 7 falls in [2,7] → report it. 8 does not (8 > 7) → skip.

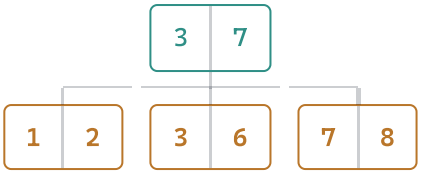
STEP 5 OF 5



Done — reported {2, 3, 6, 7} in 3 disk trips. But look at the PATH: root → leaf[3] → back up to the root → leaf[7,8]. That "back up" is unavoidable — leaf[3] has no pointer straight to leaf[7,8]; the only route between two leaves runs back through their shared ancestor.

B+ TREE — THE SAME QUERY [2,7]

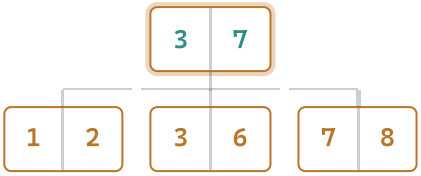
STEP 1 OF 6



LEAF CHAIN — 1,2 → 3,6 → 7,8

The same query, [2,7], on a B+ tree holding the same six keys. This exact tree is built step by step, from scratch, in Section 07 — for now, just watch the shape of the search. Data lives ONLY in the leaves; the root [3,7] holds nothing but routing copies.

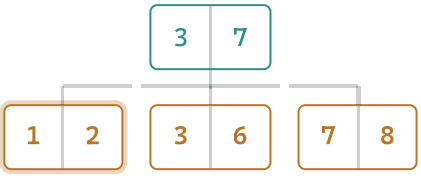
STEP 2 OF 6



LEAF CHAIN — 1,2 → 3,6 → 7,8

Step 1 — find where the range STARTS: at root [3,7], 2 < 3 → the range must begin in the left leaf. One descent, no data here to report — the root is a pure signpost.

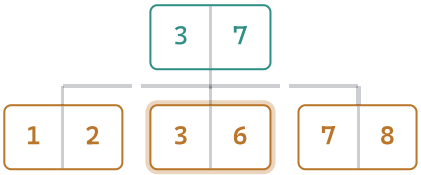
STEP 3 OF 6



LEAF CHAIN — 1,2 → 3,6 → 7,8

Arrive at leaf [1,2] — the only descent this whole query ever makes. Scan left to right: 1 < 2, skip; 2 falls in [2,7] → report it.

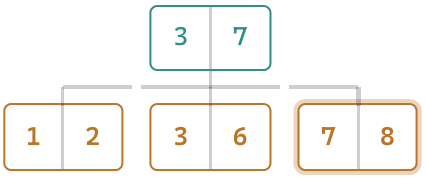
STEP 4 OF 6



LEAF CHAIN — 1,2 → 3,6 → 7,8 ← FOLLOWING NEXT POINTERS, NEVER BACK THROUGH THE ROOT

Follow the NEXT pointer — no trip back through the root — to leaf [3,6]: 3 and 6 both fall in [2,7] → report both.

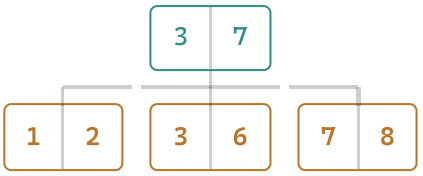
STEP 5 OF 6



LEAF CHAIN — 1,2 → 3,6 → 7,8 ← FOLLOWING NEXT POINTERS, NEVER BACK THROUGH THE ROOT

Follow NEXT again to leaf [7,8]: 7 falls in [2,7] → report it. 8 does not (8 > 7) → stop scanning — and since leaves are sorted, stop the whole query right here.

STEP 6 OF 6



LEAF CHAIN — 1,2 → 3,6 → 7,8

Done — reported {2, 3, 6, 7}, the identical set the B-tree found. But the PATH: root → leaf[1,2] → leaf[3,6] → leaf[7,8]. One descent, then a straight forward walk. No trip ever goes back up — that's the entire fix.

Both queries return the identical set — {2, 3, 6, 7} — and cost about the same number of disk trips at this tiny scale. What differs is the *shape* of the trip: the B-tree's path is root → leaf → **back to root** → leaf, revisiting the top of the tree mid-query. The B+ tree's path is root → leaf → leaf → leaf — one descent, then a straight forward walk that never looks back up. At real scale (millions of keys, ranges spanning thousands of them) that difference is the entire reason range-heavy systems choose B+ trees.

Two kinds of node now, not one

SAME SLOT-AND-POINTER LAYOUT, ONE CRUCIAL DIFFERENCE: WHERE DOES DATA SIT?

A B-TREE NODE — EVERY KEY CARRIES ITS OWN DATA, WHETHER THE NODE IS THE ROOT, INTERNAL, OR A LEAF

A B+ TREE INTERNAL NODE — SAME P-K-P-K-P SLOT PATTERN, BUT EVERY KEY IS A BARE SIGNPOST COPY

A B+ TREE LEAF NODE — NO CHILD POINTERS AT ALL, JUST REAL DATA AND ONE POINTER TO THE NEXT LEAF

INTERNAL (INDEX) NODES

Store only **copies** of keys, purely to steer a search left or right. Hold no data of their own. Obey the same max-m-children rule as a B-tree.

LEAF NODES

Store the **real** keys (and, in a database, the record or a pointer to it) — plus one extra pointer to the *next* leaf, threading every leaf into one sorted linked list.

This double bookkeeping is the trade B+ trees make: a key can exist **twice** — once as real data in a leaf, once as a signpost copy in an internal node — and if the data ever moves to a different leaf, its signpost copy must be updated to match. You'll watch that exact update happen in the deletion animation below.

Same keys, 1 through 7 — a different shape

WHAT'S DIFFERENT FROM THE B-TREE BUILD

A **leaf** split never removes a key — it **copies** the smallest key of the new right leaf up as the separator (the data must stay in a leaf). How many keys land on each side follows one rule at every order: when a leaf overflows to **m** keys, the left leaf keeps ⌈m/2⌉ of them and the right leaf keeps ⌊m/2⌋. Order-3's overflow of 3 keys splits 2-left/1-right, so the copied-up key — the smallest of the right leaf — happens to *also* be the largest of the three, as the animation below shows. That's a coincidence of the 1-key right half, not the rule: at order 4 (4 keys → 2-and-2) or order 5 (5 keys → 3-and-2), the copied-up key is just the **first** key of whichever keys landed in the right half, no longer the largest overall. An **internal** split **moves** its true median up, exactly like a B-tree (there's no data attached to a signpost, so nothing needs to stay behind).

WATCH THE LEAF CHAIN

Below every tree snapshot you'll see the leaf-to-leaf linked list spelled out. However tangled the tree above gets, that chain always stays a single, unbroken, sorted sequence — the whole point of a B+ tree.

INTERACTIVE — INSERT 1, 2, 3, … , 7 INTO AN ORDER-3 B+ TREE

STEP 1 OF 12

LEAF CHAIN (LINKED LIST) — ∅

Start: empty order-3 B+ tree. Watch where the DATA lives at every step — always in the leaves.

STEP 2 OF 12

LEAF CHAIN (LINKED LIST) — 1

Insert 1: a single leaf, which is also the root.

STEP 3 OF 12

LEAF CHAIN (LINKED LIST) — 1,2

Insert 2: leaf [1,2]. Still within the limit.

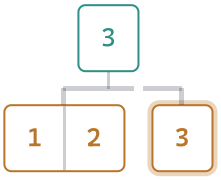
STEP 4 OF 12

LEAF HAS 3 KEYS — OVERFLOW (ORDER 3 MAX IS 2) — MUST SPLIT NOW

LEAF CHAIN (LINKED LIST) — 1,2,3

Insert 3: the leaf would hold [1,2,3] — 3 keys, over the order-3 limit of 2. Overflow (dashed red) — split now.

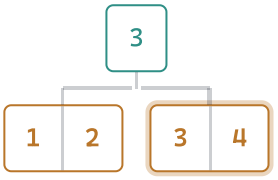
STEP 5 OF 12



LEAF CHAIN (LINKED LIST) — 1,2 → 3

Split: the separator key is **copied** up, not moved — 3 still lives in the leaf as real data; the copy in the root [3] is only a signpost.

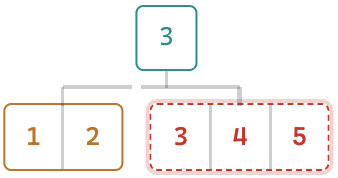
STEP 6 OF 12



LEAF CHAIN (LINKED LIST) — 1,2 → 3,4

Insert 4: goes to the rightmost leaf → [3,4]. No split.

STEP 7 OF 12

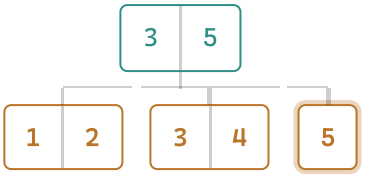


LEAF HAS 3 KEYS — OVERFLOW — MUST SPLIT NOW

LEAF CHAIN (LINKED LIST) — 1,2 → 3,4,5

Insert 5: the rightmost leaf would hold [3,4,5] — overflow (dashed red). Split now.

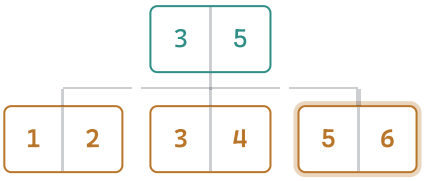
STEP 8 OF 12



LEAF CHAIN (LINKED LIST) — 1,2 → 3,4 → 5

Split: copy **5** up. Root becomes [3,5] — still within the order-3 limit.

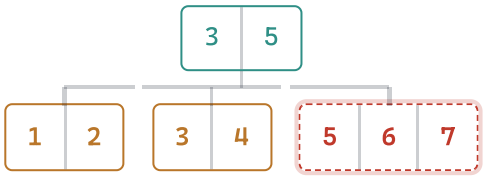
STEP 9 OF 12



LEAF CHAIN (LINKED LIST) — 1,2 → 3,4 → 5,6

Insert 6: rightmost leaf → [5,6]. No split.

STEP 10 OF 12

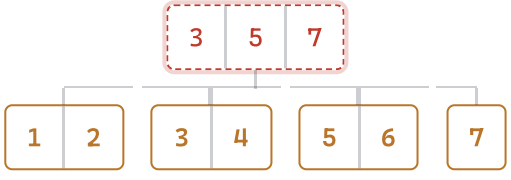


LEAF HAS 3 KEYS — OVERFLOW — MUST SPLIT NOW

LEAF CHAIN (LINKED LIST) — 1,2 → 3,4 → 5,6,7

Insert 7: the rightmost leaf would hold [5,6,7] — overflow (dashed red). Split now.

STEP 11 OF 12

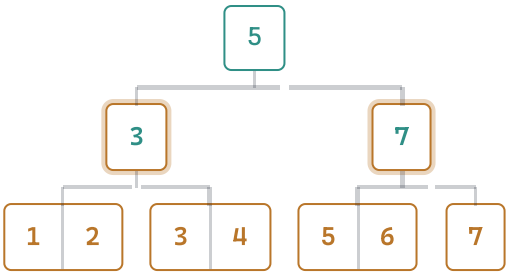


ROOT NOW HAS 3 KEYS TOO — OVERFLOW, MUST SPLIT NEXT

LEAF CHAIN (LINKED LIST) — 1,2 → 3,4 → 5,6 → 7

Split the leaf: copy **7** up. But the root already held [3,5] — now [3,5,7], three keys: the root itself overflows too (dashed red) — split it next.

STEP 12 OF 12



LEAF CHAIN (LINKED LIST) — 1,2 → 3,4 → 5,6 → 7

Splitting the overflowing root: median **5 MOVES** up — internal splits move, they don't copy, since there's no data attached to a pure index key. New root [5], with two index children [3] and [7]. Every leaf is still at the same depth, and the leaf chain below is unbroken.

Delete 4, then 7, then 6 — signpost repair, then a merge

THE B+-ONLY RULE TO WATCH FOR

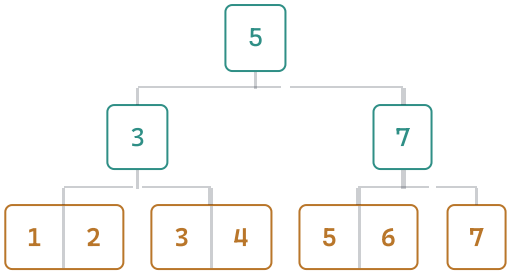
Deleting a key that was **never** copied into an internal node is exactly as simple as a leaf-only delete gets. Deleting one that borrows across a boundary is different: after the borrow, the internal signpost above it is now **stale** and must be rewritten to the new smallest key of the leaf it points to. When **neither** sibling has a key to spare, borrowing isn't an option — the leaves merge instead, and the parent's separator is simply **discarded** (it was only ever a copy, so nothing needs to move down). If that empties the parent's own key count, the underflow cascades one level up — and there the separator **does** get pulled down, exactly like a plain B-tree merge, because an internal node carries no data of its own to protect.

WHY THIS MATTERS IN PRACTICE

Every insert or delete near a leaf boundary in a real database index pays this small bookkeeping cost. It's the price of keeping the leaf chain always walkable in order — cheap per operation, and the reason range queries stay fast forever.

INTERACTIVE — DELETE 4, THEN 7, THEN 6

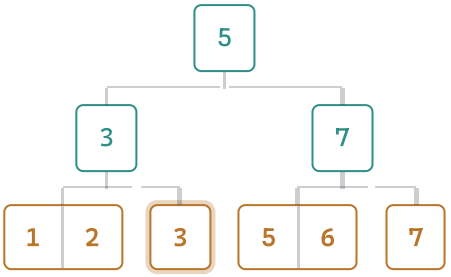
STEP 1 OF 8



LEAF CHAIN (LINKED LIST) — 1,2 → 3,4 → 5,6 → 7

Starting tree. Now delete 4, then 7 — watch what happens to the index copies above them.

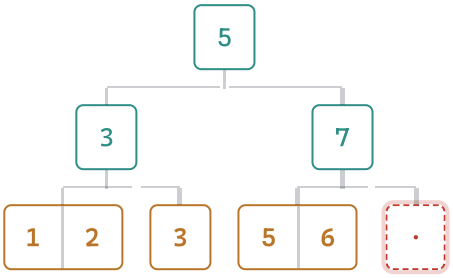
STEP 2 OF 8



LEAF CHAIN (LINKED LIST) — 1,2 → 3 → 5,6 → 7

Delete 4: leaf [3,4] → [3]. Still meets the minimum. And since 4 was *never* copied up as an index key (only 3, 5, 7 were), nothing else in the tree needs to change — the simplest possible B+ deletion.

STEP 3 OF 8

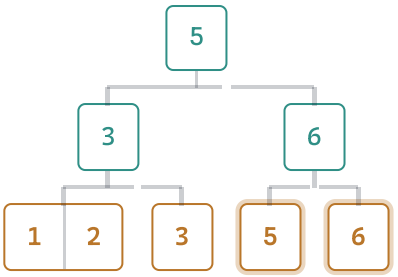


DASHED RED = UNDERFLOW: 0 KEYS — MUST BE REPAIRED

LEAF CHAIN (LINKED LIST) — 1,2 → 3 → 5,6 → ·

Delete 7 — remove it first: its leaf empties to zero keys — **underflow**. Note the index copy of 7 above is untouched for the moment; diagnose the sibling before repairing: [5,6] holds 2 keys — **one to lend**.

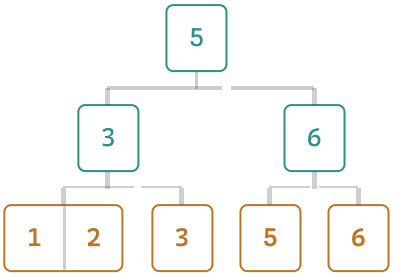
STEP 4 OF 8



LEAF CHAIN (LINKED LIST) — 1,2 → 3 → 5 → 6

Repair — borrow: the sibling lends its largest, 6: [5,6] shrinks to [5], the empty leaf becomes [6]. The B+-only step: the index key routing to this leaf must be **updated** to match the new boundary — 7 becomes 6.

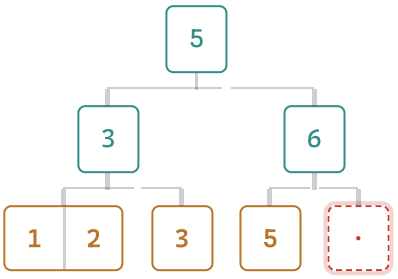
STEP 5 OF 8



LEAF CHAIN (LINKED LIST) — 1,2 → 3 → 5 → 6

Tree after delete 4, then 7: root [5], left index [3] over leaves [1,2] | [3], right index [6] over leaves [5] | [6]. Now delete 6 — this time neither sibling has anything to spare, so watch what a **merge** does instead of a borrow.

STEP 6 OF 8

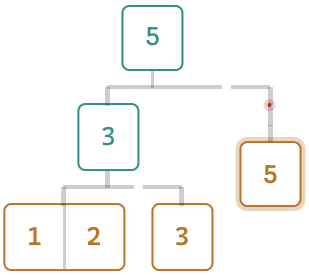


DASHED RED = UNDERFLOW: 0 KEYS — MUST BE REPAIRED

LEAF CHAIN (LINKED LIST) — 1,2 → 3 → 5 → ·

Delete 6: its leaf empties to zero keys — underflow. Diagnose the sibling: [5] holds only 1 key — already at the order-3 minimum. Nothing to lend. Borrowing is off the table; we must **merge**.

STEP 7 OF 8

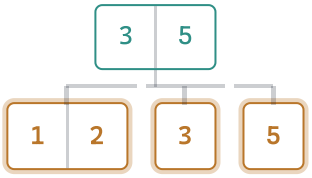


DASHED RED = THE INTERNAL NODE ITSELF NOW UNDERFLOWS: 0 KEYS, 1 CHILD

LEAF CHAIN (LINKED LIST) – 1,2 → 3 → 5

Leaf merge: concatenate the two leaves' real keys — [] ∪ [5] = [5]. The parent's separator key 6 is simply **discarded**, not pulled down — it was only ever a routing copy, so there's nothing that needs to "come home" to a leaf. But the right internal node is left with 0 keys and just 1 child: the underflow has moved up a level.

STEP 8 OF 8



LEAF CHAIN (LINKED LIST) – 1,2 → 3 → 5

Internal merge: the sibling index [3] is also at its minimum — 1 key, nothing to lend. So the two internal nodes merge too, and this time the parent's separator **does** get pulled down — exactly like a plain B-tree merge, because an internal node holds no data to protect. Merged keys = [3] + root's 5 + [] = [3,5], with all three leaves now hanging off it. The root is left with 0 keys and 1 child, so it's discarded — the merged node becomes the new root. **Final tree:** root [3,5] over leaves [1,2] | [3] | [5].

Where data lives decides everything else

ASPECT	B-TREE	B+ TREE
Where real data lives	Any node — root, internal, or leaf	Leaves only
Internal node content	Real keys, with data attached	Routing copies only, no data
Range query / sequential scan	Must revisit internal nodes repeatedly	One hop to the next leaf via the linked list
Search cost (same order m)	$O(\log_m n)$	$O(\log_m n)$, then a walk of the leaf level
Space overhead	No key duplication	Some duplication — routing copies
Typical real-world use	Some in-memory / general-purpose indexes	Almost every production DB index (MySQL InnoDB, PostgreSQL) and filesystem directory index (NTFS, ext4)

The wasted-space problem

WHAT REPEATED INSERTS AND DELETES LEAVE BEHIND

A split only guarantees a node stays **at least half full** ([m/2] − 1 keys minimum). After years of real-world churn, most nodes in a live B-tree or B+ tree settle near that 50% floor — meaning roughly half of every disk block read is empty space. At the scale of a production index, that's an enormous amount of wasted I/O and cache.

THE B* IDEA

Delay splitting for as long as possible. Before splitting a full node, first check whether a **sibling** has spare room and shift keys sideways through the parent — an insertion-time version of the borrow-from-sibling trick you already used in deletion. Only split when **both** a node and its sibling are completely full — and even then, split two full nodes into **three** roughly-two-thirds-full ones, instead of one full node into two half-full ones.

Same skeleton, a stricter occupancy floor

B*-TREE OF ORDER M — THREE RULES

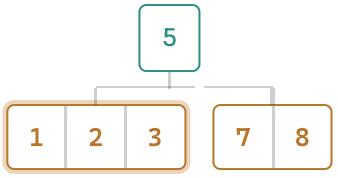
- The **root** has between 2 and 2⌊(2m−2)/3⌋+1 children.
- Every other **internal node** has between ⌈(2m−1)/3⌉ and m children — roughly **two-thirds full to completely full**, not one-half to full.
- Every **leaf** sits at the same depth — identical to a B-tree.

THE COST OF THE BETTER PACKING

Insertion and — especially — deletion logic must now respect a two-thirds floor instead of a one-half floor, which makes both noticeably more intricate to implement correctly. That extra complexity is exactly why B* trees, despite better space usage, are far rarer in production than B and B+ trees. The most famous real user: the classic Apple **HFS** filesystem's catalog file.

INTERACTIVE — REDISTRIBUTE (NO SPLIT), THEN THE TWO-TO-THREE SPLIT, ORDER M = 4

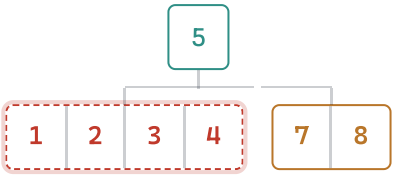
STEP 1 OF 8



L IS FULL (3 KEYS, THE ORDER-4 MAX) — BUT R HAS ROOM (2 KEYS, COULD TAKE ONE MORE)

Setup — the easy case first: an order-4 B*-tree fragment. Leaf L = [1,2,3] is full; its sibling R = [7,8] has **spare room** (only 2 of 3 slots used). We must insert **4** into L. A plain B-tree would split L immediately — a B* tree checks the sibling before ever splitting.

STEP 2 OF 8



DASHED RED = OVERFLOW: 4 KEYS IN A MAX-3 LEAF — THIS IS A TEMPORARY STATE, NEVER ACTUALLY STORED

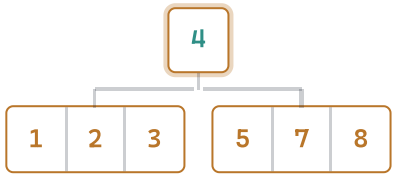
Insert 4 into L first: L temporarily holds [1,2,3,4] — one key over the order-4 limit of 3. This overflow is exactly what triggers a repair. Before splitting, a B* tree always checks a sibling first: R = [7,8] still has room (2 of 3 slots).

STEP 3 OF 8

POOL EVERYTHING: L'S KEYS + NEW KEY (4) + OLD SEPARATOR (5) + R'S KEYS — NO NEW NODE YET

Sibling has room → redistribute, don't split: pool L's overflowed [1,2,3,4], the old separator 5, and R's 2 keys — 6 values, and still only **2** leaves to fill (nobody is being created). Rebalance the boundary between L and R.

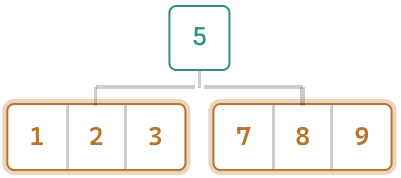
STEP 4 OF 8



NO NEW NODE — L BACK TO 3 KEYS, R ABSORBED THE OLD SEPARATOR, ONE KEY PROMOTED

Done — no split at all: L's overflow resolves back down to [1,2,3] (4 never actually stays there), the old separator 5 slides down into R → [5,7,8], and 4 rises to become the **new** separator. Total node count: still 2 leaves. This is the move a plain B-tree never even attempts.

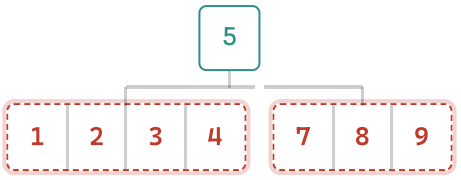
STEP 5 OF 8



NOW BOTH LEAVES ARE LEGALLY FULL (3 KEYS) — NO SPARE ROOM ANYWHERE

Now the hard case: a fresh fragment, but this time R is *also* full — [7,8,9], no spare room. Insert **4** into L again. The redistribute trick above needs a sibling with room; there isn't one. Only now does a B* tree consider splitting.

STEP 6 OF 8



DASHED RED = OVERFLOW ON L (4 KEYS); R IS ALSO FULL WITH NOWHERE TO SEND THEM — REDISTRIBUTION IS IMPOSSIBLE

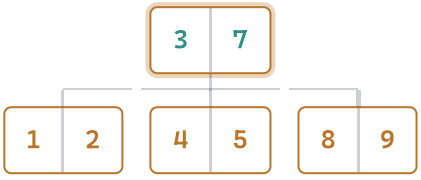
Insert 4 into L: same overflow as before — L temporarily holds [1,2,3,4]. But this time checking the sibling fails: R = [7,8,9] is completely full too, so there's no room anywhere to redistribute into. A split is now unavoidable.

STEP 7 OF 8

POOL EVERYTHING: L'S KEYS + OLD SEPARATOR (5) + NEW KEY (4) + R'S KEYS

Both full → the two-to-three split: pool *everything* — L's overflowed [1,2,3,4], the old separator 5, and R's 3 keys — into one sorted array of 8 values. Redistribute across **three** leaves instead of two — one brand-new node, not a plain 50/50 split of one.

STEP 8 OF 8



Redistribute evenly: [1,2] | 3 | [4,5] | 7 | [8,9] — three leaves, each now two-thirds full instead of half, and the overflow is fully resolved. That's the whole B* idea, both cases together: borrow sideways whenever you can; split 2-into-3 only when you truly can't. The cost is a noticeably more intricate insertion (and especially deletion — next slide) — exactly why B* trees are rarer in practice than B and B+ trees.

Why two minimum-occupied nodes can't just merge into one

THE MIRROR OF INSERTION — WITH ONE EXTRA WRINKLE

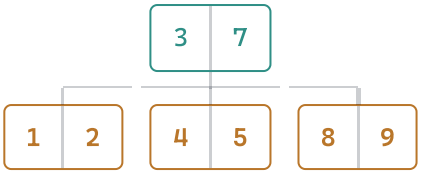
- **Underflow, sibling has spare keys:** borrow through the parent, exactly like a B-tree — no node count change.
- **Underflow, sibling is also at the minimum:** a plain B-tree would just merge the two into one. A B* tree **can't** — two-thirds-full plus two-thirds-full is *more than* one node's capacity (for order 4: 2+2 keys don't fit in a max-3 node).
- So B* deletion pulls in a **second sibling** too — three nodes at/near the minimum — and redistributes all of them across only **two** nodes: the exact mirror of the two-to-three split, run backwards.

HOW TO READ THE ANIMATION

Same order-4 tree the insertion animation just built, continued: [1,2] · 3 · [4,5] · 7 · [8,9]. First a quick insert to set up an honest "sibling has spare keys" case, then two deletions — one that borrows cleanly, and one that forces the three-into-two collapse.

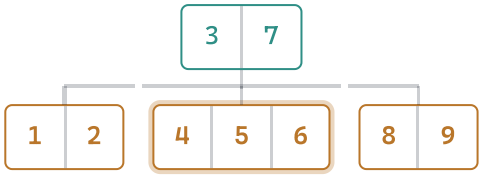
INTERACTIVE — INSERT 6, DELETE 9 (BORROWS), DELETE 4 (THREE-INTO-TWO)

STEP 1 OF 8



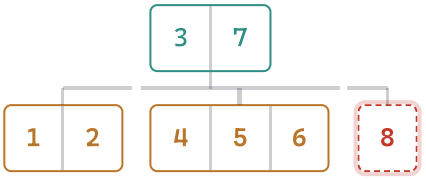
Starting tree — exactly where the insertion animation left off: 8 keys, every leaf sitting at the minimum (2 keys). First, one quick insert to set up an honest example.

STEP 2 OF 8



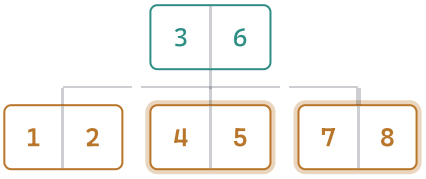
Insert 6 (setup, not the point of this slide): 3 < 6 < 7, lands in the middle leaf → [4,5,6]. Full (3 keys), but legal — no split needed. Now the middle leaf has a spare key others don't.

STEP 3 OF 8



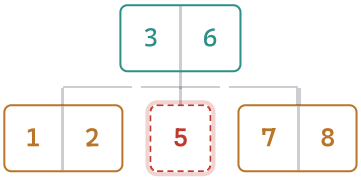
Delete 9: rightmost leaf [8,9] → [8] — **underflow** (1 key, below the minimum of 2). Diagnose the only neighbor: the middle leaf [4,5,6] has 3 keys — one more than its minimum. It can lend.

STEP 4 OF 8



Borrow through the parent: separator 7 drops into the empty slot next to 8 → [7,8]; the middle leaf's largest key, 6, rises to replace it as the new separator; middle leaf shrinks to [4,5]. No merge, no node destroyed — every leaf legal again at exactly 2 keys.

STEP 5 OF 8



Delete 4 — the hard case: middle leaf [4,5] → [5] — **underflow** again. Check its LEFT neighbor [1,2]: exactly at the minimum, can't lend. Check its RIGHT neighbor [7,8]: also exactly at the minimum, can't lend either. Both blocked.

STEP 6 OF 8

WOULD A PLAIN 2-INTO-1 MERGE WITH EITHER NEIGHBOR FIT? [1,2] + 3 + [5] = 4 KEYS — ORDER-4 MAX IS 3

1 2 3 5 → 4 keys

The B*-specific problem: an ordinary B-tree would just merge [5] with a neighbor now. But two nodes at the B* minimum (2 + 2 keys) plus a separator is **4 keys — too many for one order-4 node (max 3)**. A simple 2-into-1 merge literally does not fit. B* deletion needs a second neighbor.

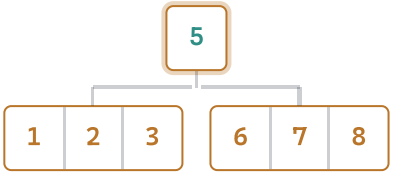
STEP 7 OF 8

POOL THREE NODES: [1,2] + SEPARATOR 3 + [5] + SEPARATOR 6 + [7,8]

1 2 3 5 6 7 8

Pull in the second sibling too: [1,2], the underflowing [5], and [7,8], plus both separators 3 and 6 — seven values in total. Redistribute across only **two** leaves, deleting the third — the exact mirror of the two-to-three split, run backwards.

STEP 8 OF 8



Three-into-two, done: [1,2,3] | 5 | [6,7,8] — one new separator (5) promoted, two full, legal leaves, and the root shrinks from 3 children to 2. Compare this to the B-tree delete demo's merge: same motivation, different arithmetic, because the B* floor is 2/3 full, not 1/2.

One idea, worn three ways

ASPECT	B-TREE	B+ TREE	B* TREE
Minimum occupancy	~50%	~50% (both levels)	~67%
Where data lives	Any node	Leaves only	Any node
Checks sibling before splitting	No	No	Yes — the two-to-three split
Real-world example	General-purpose on-disk indexes	MySQL InnoDB, PostgreSQL, NTFS, ext4	Classic Mac OS HFS catalog file

Every idea in this lecture is really one idea worn three ways: keep the tree **short** (wide branching, not two children) and keep every node **densely packed**, because on disk the number of *nodes touched* matters far more than the number of comparisons inside each one. B+ trees additionally chain their leaves so a whole range of answers costs one linked-list walk. B* trees push occupancy further still, at the price of trickier bookkeeping.

LECTURE 12 — NEXT

Segment Trees and Interval Trees

From "find one key" to "answer a query about a whole range" — a different kind of tree, still built to answer fast.

HOMEWORK — BRING TO LECTURE 12

- Build an order-3 B-tree by inserting 20, 40, 10, 30, 50, 5, 60. Verify every leaf ends at the same depth.
- Starting from this lecture's final B-tree (root [2,6], leaves [1], [3], [7,8]) delete 8. Which of the four deletion cases applies?
- Build an order-3 B+ tree for the same keys as Q1, and write out the leaf chain.
- In two sentences: why does a B+ tree answer a range query faster than a B-tree of the same order, even though both have the same search height?
- Reading: CLRS §18 (B-Trees); Weiss §4.7.

Questions?

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

> *Next: Lecture 12 — Segment Trees and Interval Trees.*