Segment Trees & Interval Trees

From "find one key" to "which of these ranges does my query touch?" Two different structures, two different assumptions about the data — and today we build both, insert into both, query both, and trace every single step by hand.

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

~70 minutes
Session outcome: implement range and interval query operations

~70 MIN, LONGER THAN USUAL — THE TOPIC EARNS IT

Today, part by part

The question no BST answers well: "what touches this point?"

THE STABBING QUERY

Suppose n machines are each busy during some integer time interval — machine A from minute 15 to 20, machine B from 10 to 30, and so on. A scheduler constantly asks: **"which machines are busy during minute [2,3]?"** — a single point (or unit interval) "stabbing" through a set of ranges.

A plain BST indexes single values, not ranges — it has no natural notion of "does my point fall inside this stored interval?" Scanning all n intervals per query is **O(n)**. With thousands of intervals and thousands of queries, that's too slow.

WHERE THIS SHOWS UP

- **IP router filters:** a firewall rule like (10*, [20,60]) matches a rectangle of address space — deciding which rules match a packet is a stabbing query.
- **VLSI mask verification / motion tracking:** computational geometry problems reduce to "which segments does this sweep-line touch right now?"
- **Genome / calendar / network-event systems:** "which recorded intervals overlap this instant?" — the same question, different domain.

A **segment tree** answers this in **O(log n + k)**, where k is the number of intervals reported — by pre-organizing the integer universe itself, once, into a fixed hierarchy of ranges.

A binary tree over the number line itself, not over the data

THE RECURSIVE STRUCTURE

- Every node v represents a closed range **[s(v), e(v)]** with s(v) < e(v), both integers.
- The **root** represents the whole universe, [1, n].
- If e(v) = s(v) + 1, v is a **leaf** — a unit interval, [k, k+1]. It splits no further.
- Otherwise, split at the midpoint m = ⌊(s(v)+e(v))/2⌋: left child = [s(v), m], right child = [m, e(v)].

THE KEY MENTAL SHIFT

Notice what's fixed **before you insert a single interval**: the entire tree shape depends only on the universe size n — not on which intervals you'll eventually store. You build the skeleton once; then you decorate nodes of that skeleton with whichever intervals fall inside them. This is completely different from a BST, where the shape depends on the data itself.

Every unit interval [k, k+1] is called an **elementary interval** — these are the leaves. Height of this tree for [1, n] is at most $\lceil \log_2(n - 1) \rceil + 1$, so any root-to-leaf walk is O(log n) — exactly like a balanced BST, but the balance is *guaranteed by construction*, never by rotation.

Root range [1,13] — split until every leaf is a unit interval

WHY 13, AND WHY IT LOOKS SLIGHTLY UNEVEN

n = 13 is not a power of 2, so the tree isn't perfectly complete — some branches hit unit length one level sooner than others. That's normal and doesn't break anything: the height bound $\lceil \log_2 12 \rceil + 1 = 5$ still holds everywhere.

HOW TO READ THE ANIMATION

Teal boxes still have children (they split further). **Amber** boxes are leaves — unit intervals, done splitting. Watch the tree reveal one full level at a time.

INTERACTIVE — REVEAL THE TREE LEVEL BY LEVEL

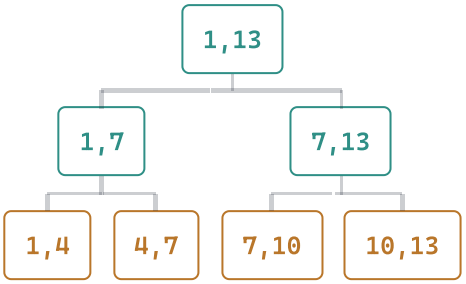
STEP 1 OF 5

Root: [1,13] — the whole universe. n = 13, so the height bound is $\lceil \log_2 12 \rceil + 1 = 5$.

STEP 2 OF 5

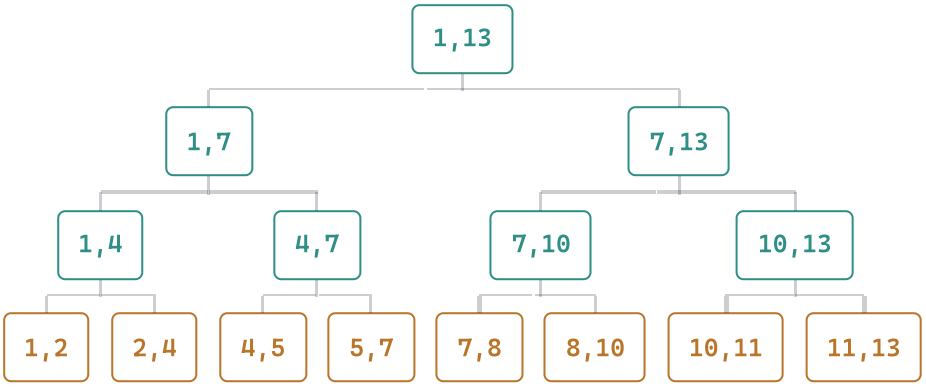
Depth 1: mid = ⌊(1+13)/2⌋ = 7. Split into [1,7] and [7,13].

STEP 3 OF 5



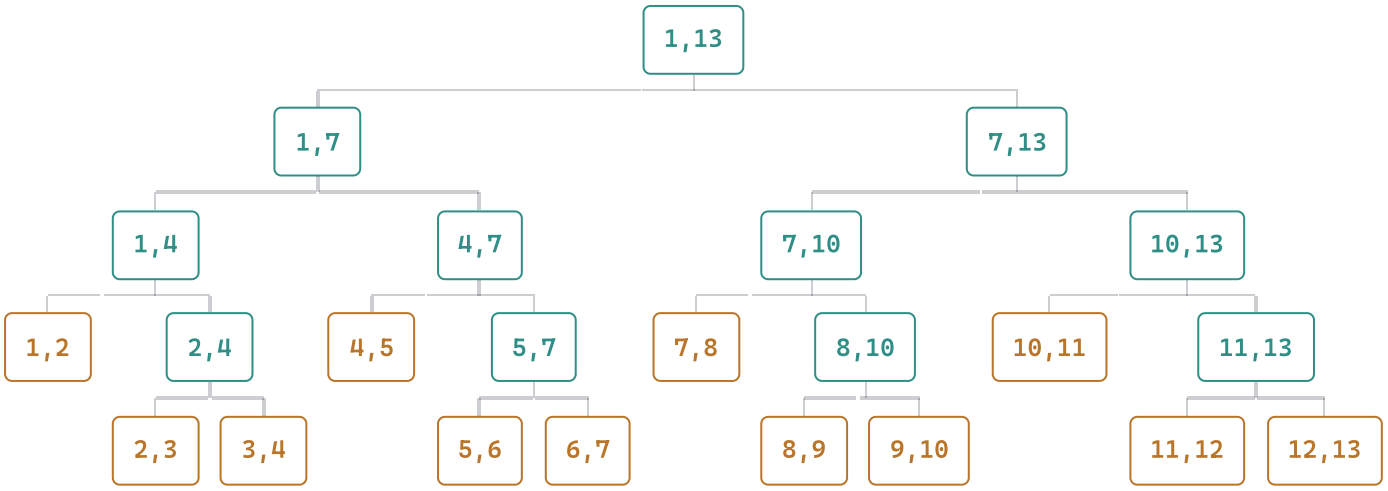
Depth 2: [1,7] splits at mid 4 into [1,4],[4,7]. [7,13] splits at mid 10 into [7,10],[10,13].

STEP 4 OF 5



Depth 3: some ranges are already length 1 and stop here (e.g. [1,2],[4,5],[7,8],[10,11] — amber, leaves). Others still have length > 1 and split again (e.g. [2,4],[5,7],[8,10],[11,13] — teal).

STEP 5 OF 5



Depth 4 (final): the last remaining ranges — [2,4],[5,7],[8,10],[11,13] — split into their unit-interval leaves. All 23 nodes now exist: 12 leaves, 11 internal nodes, height 4. This skeleton never changes again — only the *stored sets* inside its nodes will, as we insert and delete intervals.

Storing an interval: cover it with the fewest possible nodes

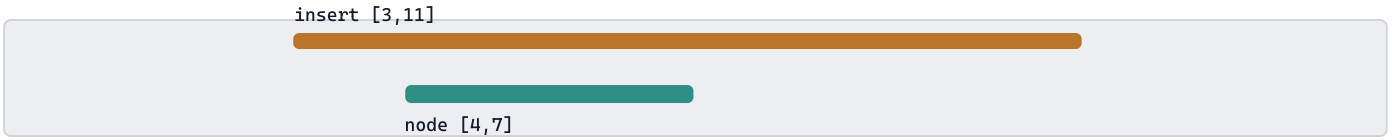
THE INSERT RULE

```
insert(s, e, v):
  // store [s,e] into the subtree rooted at v
  if s ≤ s(v) and e(v) ≤ e:
      add [s,e] to v            // v's whole range fits inside [s,e]
      // stop – do NOT recurse into v's children
  else:
      m = (s(v) + e(v)) / 2
      if s < m:  insert(s, e, v.left)
      if e > m:  insert(s, e, v.right)
```

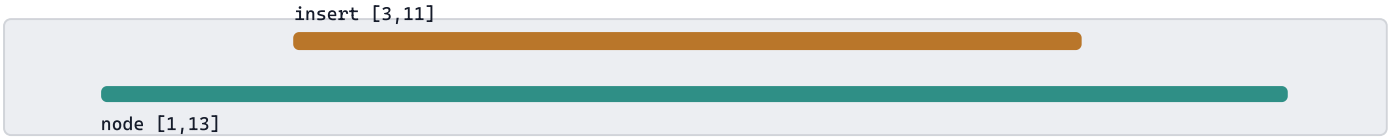
THREE PROPERTIES THIS GUARANTEES

- If [i,j] is stored at node v, it is **never** also stored at any ancestor or sibling of v — no duplication, no double-reporting.
- At any single level of the tree, [i,j] is stored in **at most 2** nodes.
- Since the tree has O(log n) levels, every interval is stored in **O(log n) nodes total** — this set of nodes is called its **canonical decomposition**.

THE CONDITION, ON A NUMBER LINE — SAME [3,11] YOU'LL WATCH GET INSERTED NEXT SLIDE

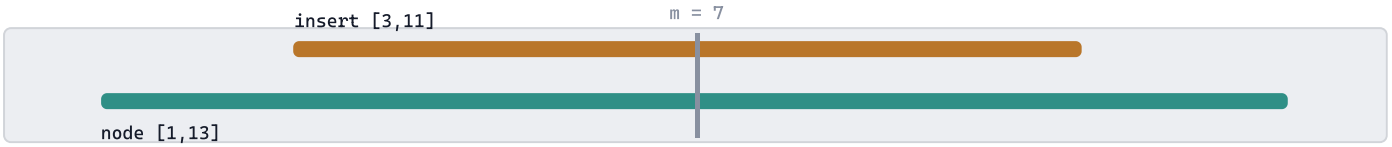


[4,7] sits entirely inside [3,11] — 3 ≤ 4 and 7 ≤ 11, both hold. Store **[3,11]** at this node, stop — no need to recurse, every point in [4,7] is genuinely covered.



[1,13] pokes out on both sides of [3,11] — the test needs 3 ≤ 1, which fails immediately. Do **NOT** store [3,11] here: storing it would wrongly claim positions 1–3 and 11–13 are covered too. Recurse into the children instead.

The else case — which child(ren)? Split node [1,13] at its own midpoint m = (1+13)/2 = 7, then ask the same question of each half separately:



- **Left half is [1,7].** s = 3 < m = 7 → the interval reaches left of the midpoint → **recurse into v.left**.
- **Right half is [7,13].** e = 11 > m = 7 → the interval reaches right of the midpoint too → **recurse into v.right**.
- Both tests pass here, so **both** children get visited — exactly the first step of the [3,11] trace on the next slide. (Had the interval stopped short of 7 on one side, only the child on that side would ever be called — that's how the recursion avoids wasted work.)

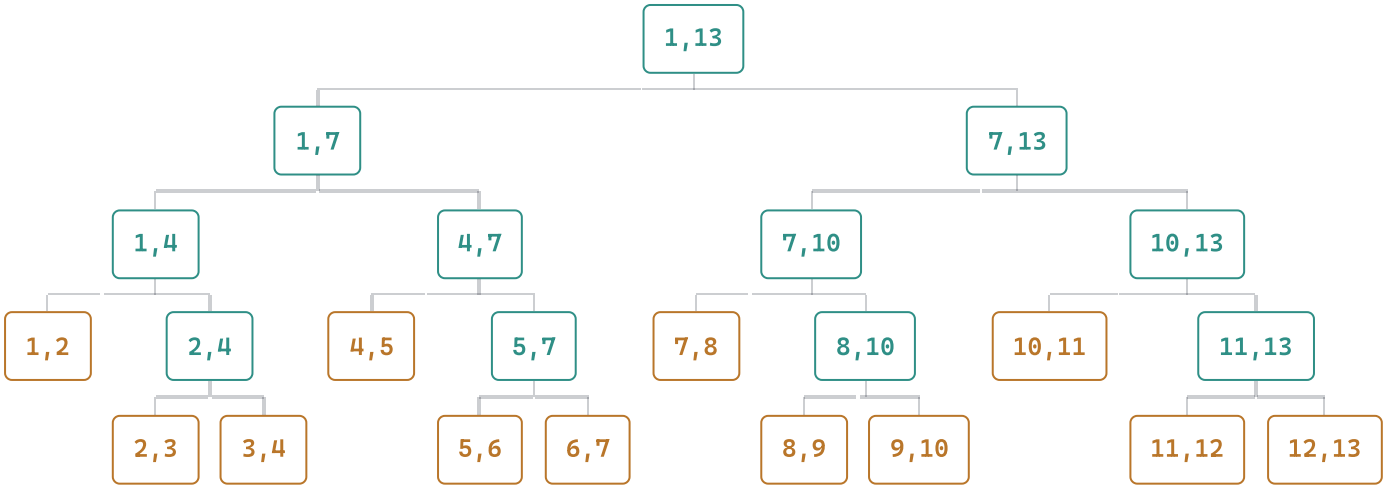
Each node holds **0 or more** stored intervals — most nodes will hold none; a handful will hold one or two. The next two animations trace this insert rule literally, call by call, for two different intervals over the tree we just built.

Every recursive call, including the ones that store nothing

We call `insert(3, 11, root)`. Ten nodes get visited before the recursion bottoms out everywhere. Watch the two-part test — "does s ≤ s(v) and e(v) ≤ e?" — get evaluated with real numbers at *every single one*, even the visits that lead nowhere. Back to L12·01's scheduling story: [3,11] is **Machine A**, busy from minute 3 to minute 11 — keep that mapping in mind for the query and delete slides ahead.

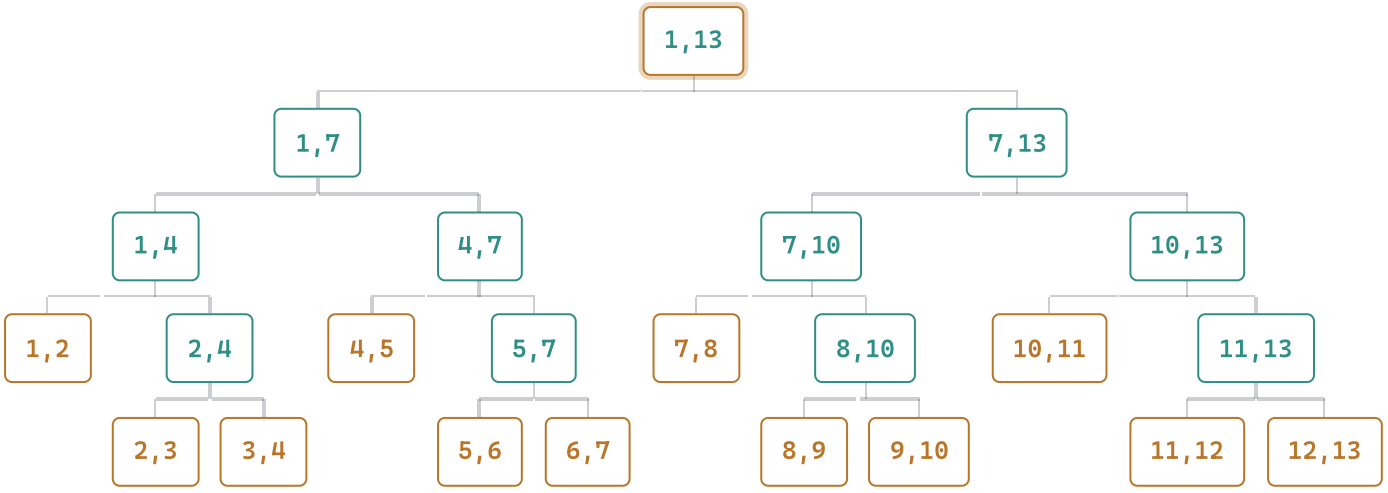
INTERACTIVE – TRACE INSERT(3, 11) NODE BY NODE

STEP 1 OF 12



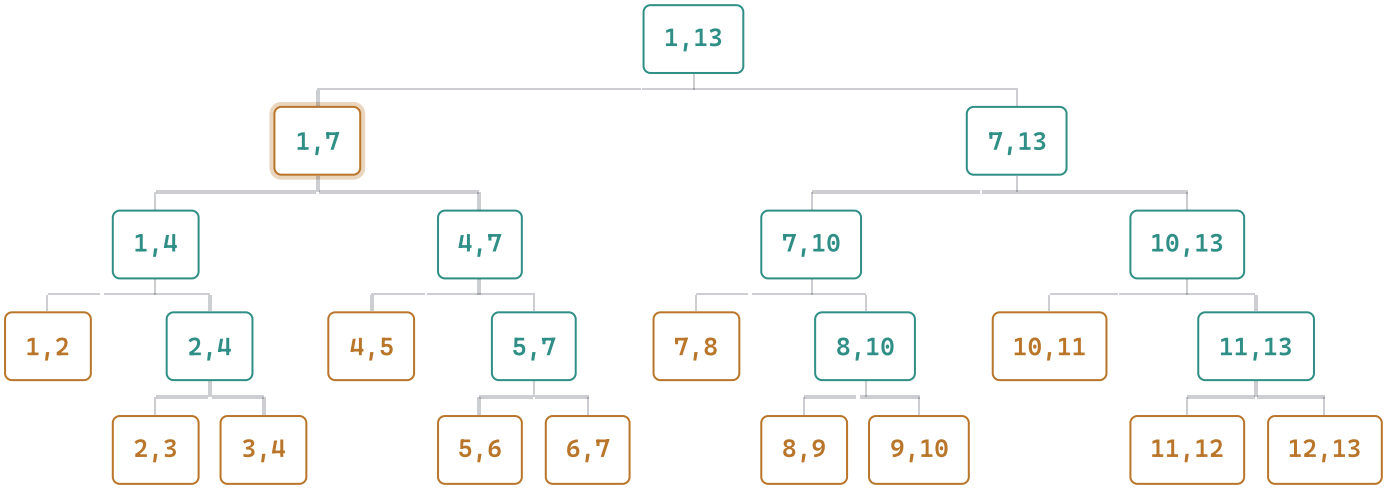
Call: insert(3, 11, root). We will test every node the recursion actually reaches — nothing skipped.

STEP 2 OF 12



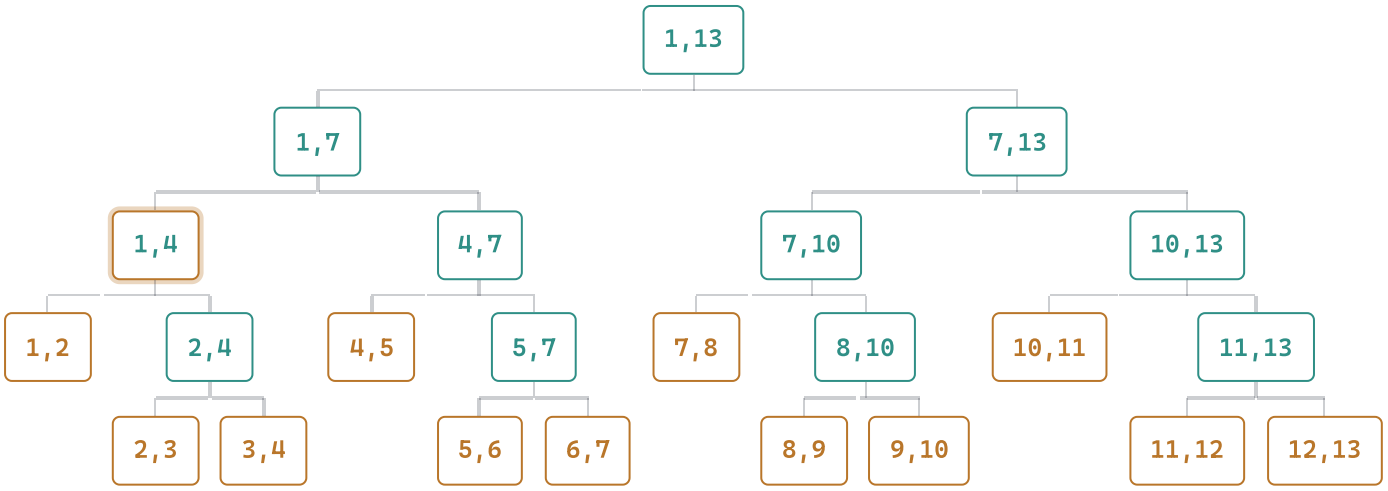
Visit [1,13]: covered? need 3 ≤ 1 — false. Not covered. mid = 7. Since 3 < 7, recurse left. Since 11 > 7, recurse right too.

STEP 3 OF 12



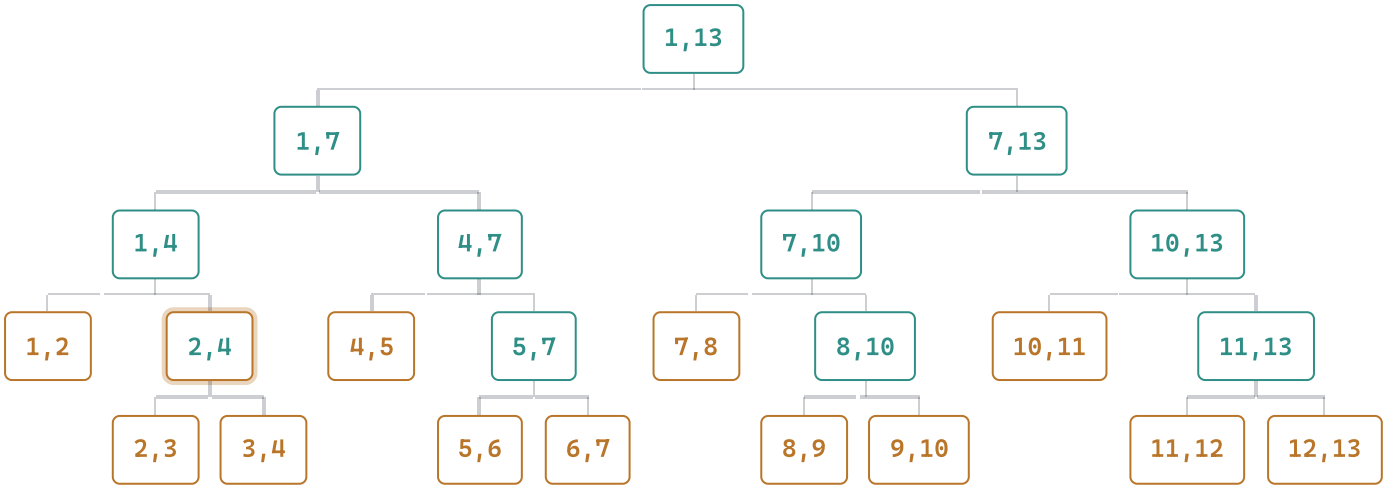
Visit [1,7] (from the left branch): covered? need 3 ≤ 1 — false. mid = 4. 3 < 4 → recurse left. 11 > 4 → recurse right.

STEP 4 OF 12



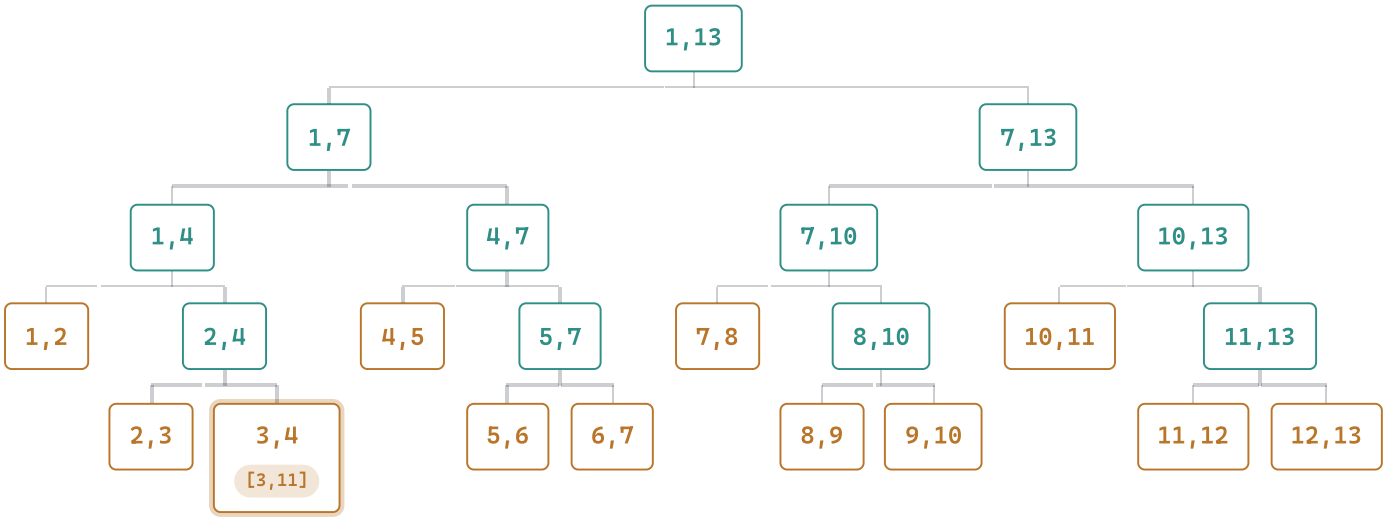
Visit [1,4]: covered? need 3 ≤ 1 — false. mid = 2. Is 3 < 2? No — do **not** recurse left into [1,2]. Is 11 > 2? Yes — recurse right only.

STEP 5 OF 12



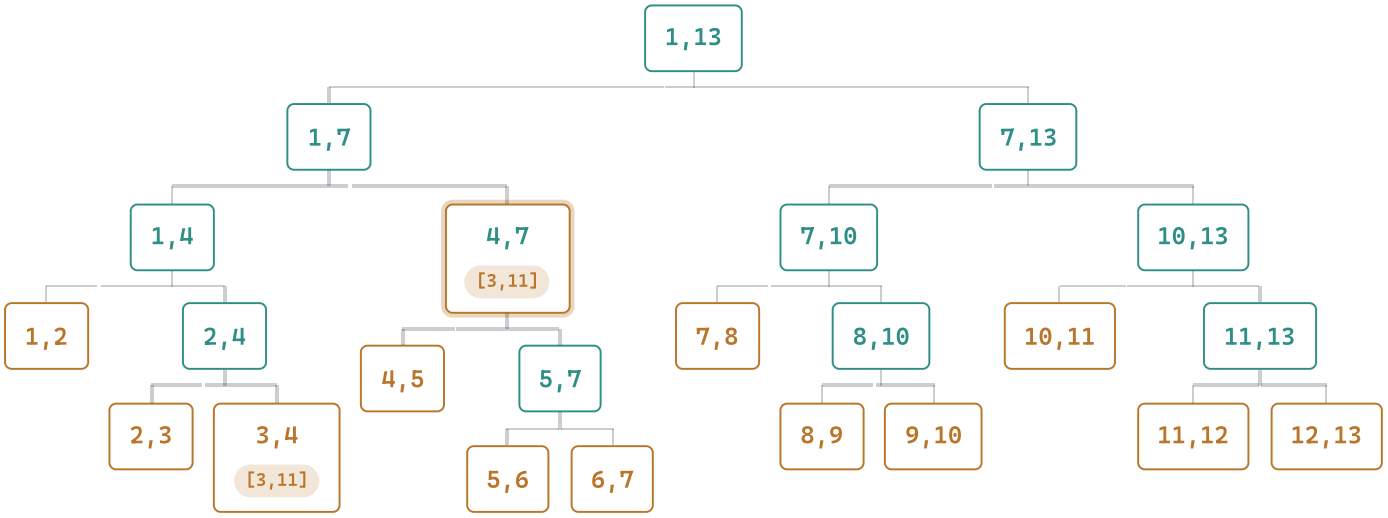
Visit [2,4]: covered? need 3 ≤ 2 — false. mid = 3. Is 3 < 3? No — skip left [2,3]. Is 11 > 3? Yes — recurse right only.

STEP 6 OF 12



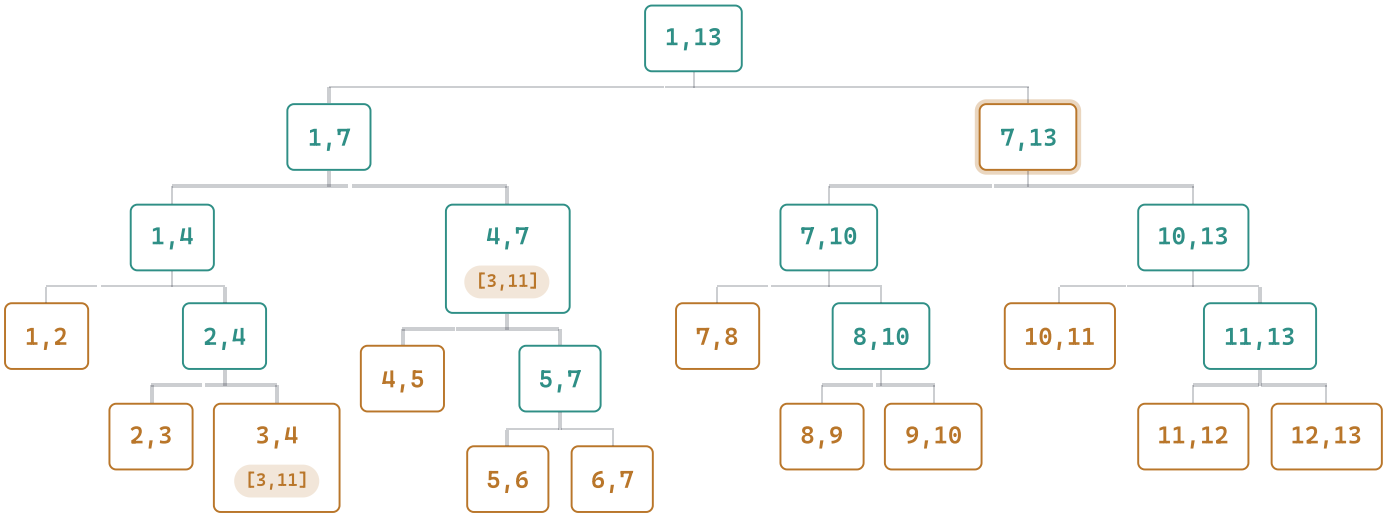
Visit [3,4]: covered? need 3 ≤ 3 **and** 4 ≤ 11 — both true! **Store [3,11] here.** This node is a leaf, so recursion stops on this branch.

STEP 7 OF 12



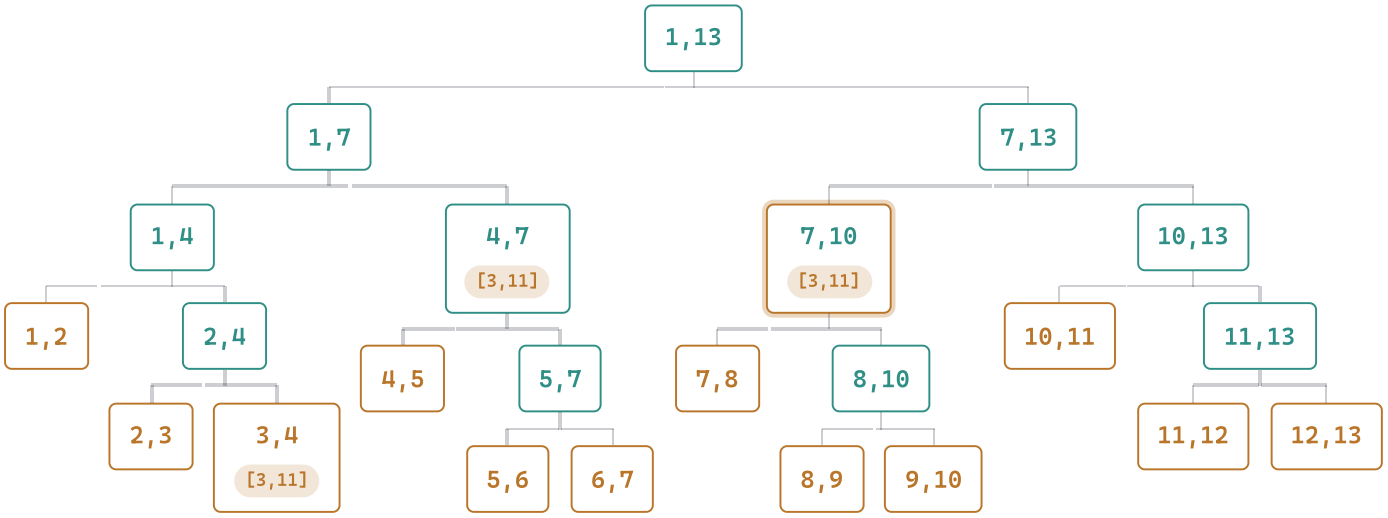
Visit [4,7] (the other branch from [1,7]'s step): covered? need 3 ≤ 4 **and** 7 ≤ 11 — both true! **Store [3,11] here.** [4,7] has children, but since it's fully covered we do NOT recurse into them — the whole subtree is represented by this one node.

STEP 8 OF 12



Visit [7,13] (back to the root's right branch from step 1): covered? need 3 ≤ 7 **yes**, but 13 ≤ 11 — **false**. Not covered. mid = 10. 3 < 10 → recurse left. 11 > 10 → recurse right.

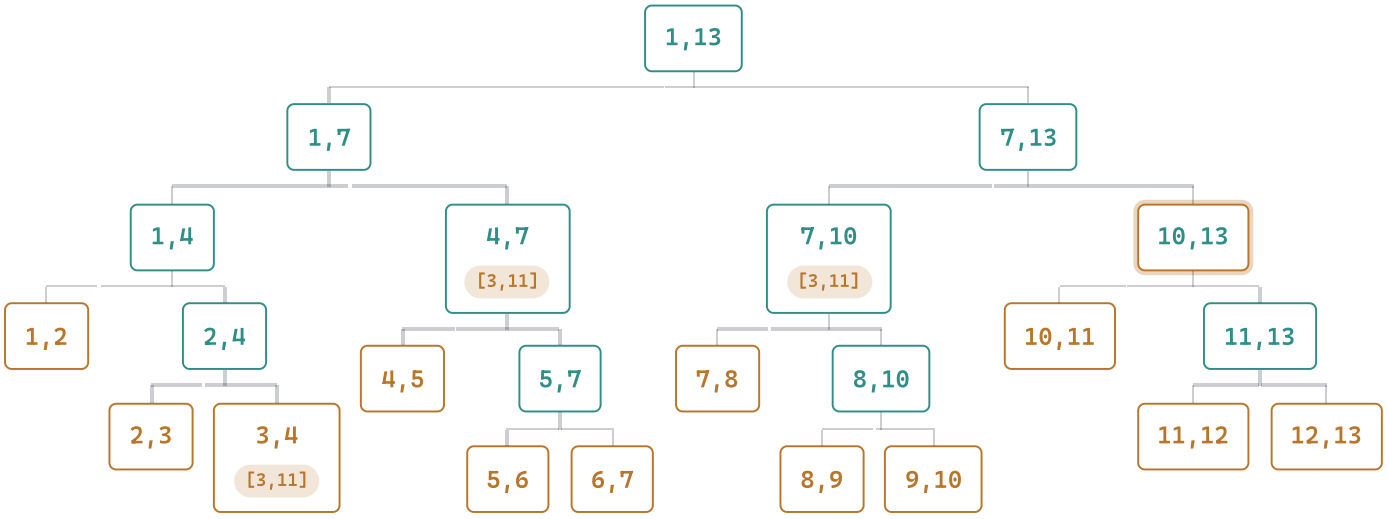
STEP 9 OF 12



3 STORED NODES SO FAR: [3,4], [4,7], [7,10]

Visit [7,10]: covered? need 3 ≤ 7 **and** 10 ≤ 11 — both true! **Store [3,11] here.**

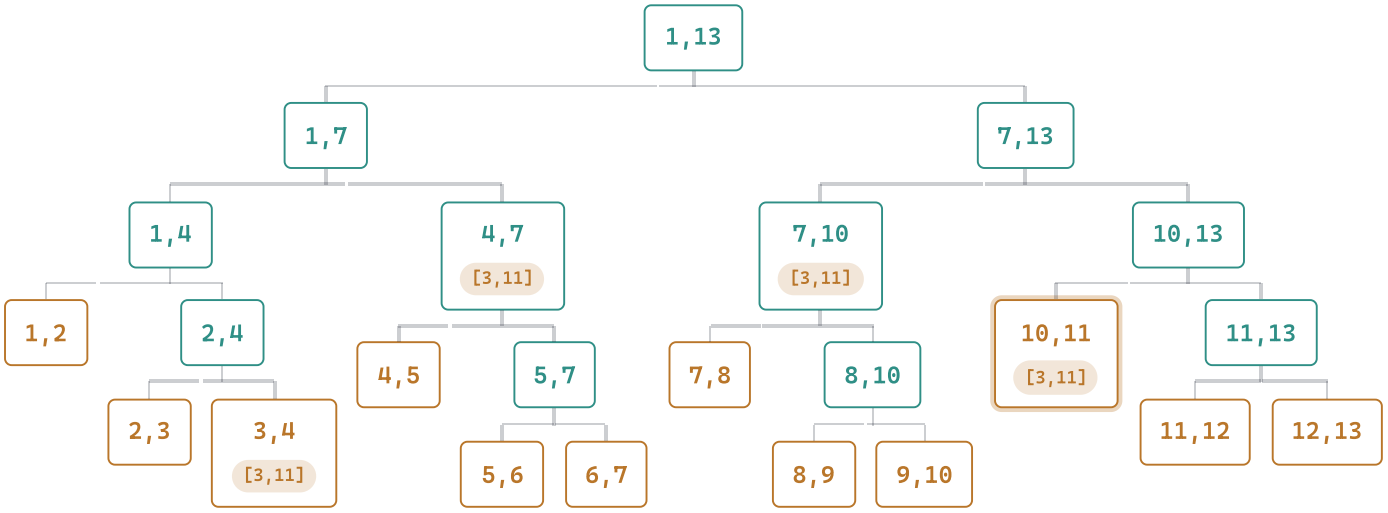
STEP 10 OF 12



CHECKING THE LAST BRANCH

Visit [10,13] (the other branch from [7,13]'s step): covered? need 3 ≤ 10 **yes**, but 13 ≤ 11 — **false**. Not covered. mid = 11. 3 < 11 → recurse left. Is 11 > 11? **No** — do not recurse right; [11,13] is never even visited.

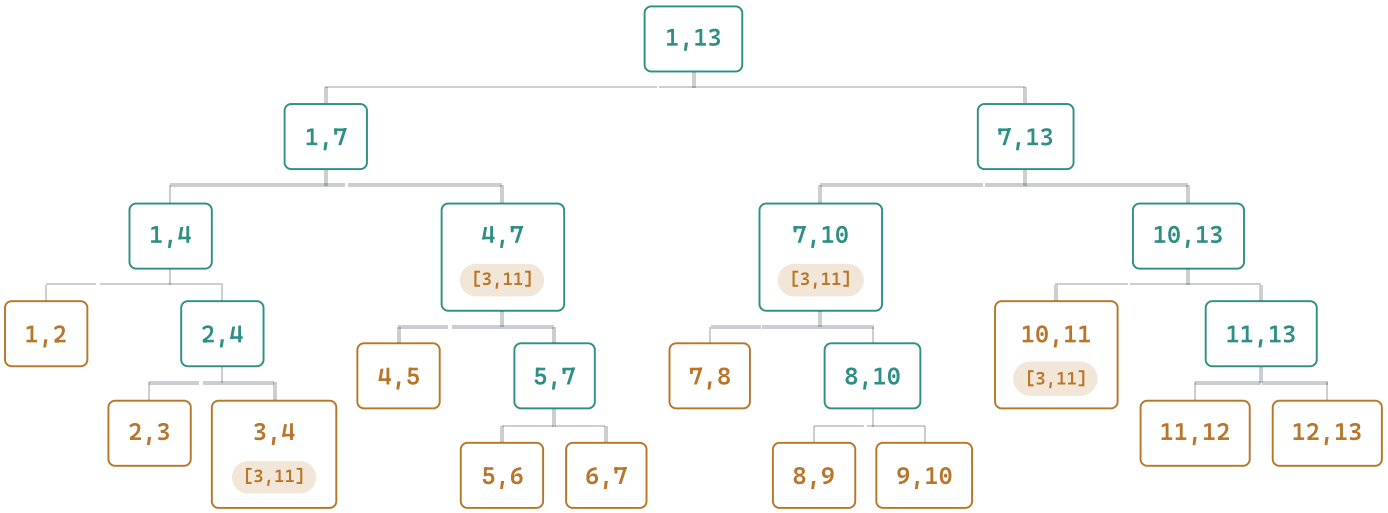
STEP 11 OF 12



FINAL CANONICAL DECOMPOSITION

Visit [10,11]: covered? need 3 ≤ 10 **and** 11 ≤ 11 — both true! **Store [3,11] here.** Recursion is now fully finished on every branch.

STEP 12 OF 12



CANONICAL DECOMPOSITION OF [3,11] = { [3,4], [4,7], [7,10], [10,11] } — 4 NODES

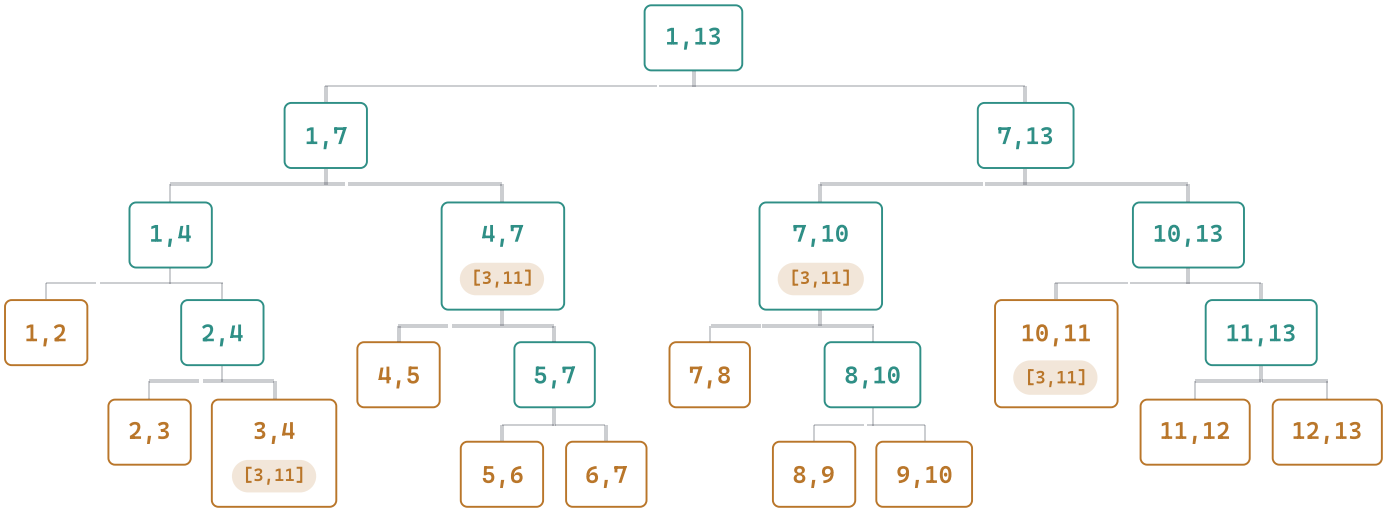
Done. Ten nodes were visited in total; exactly four of them stored [3,11]. Four nodes for one interval, out of 23 total — that's the O(log n) canonical decomposition in action.

Same rule, a boundary that lines up with the tree

[4,10]'s endpoints happen to be existing split points in the tree, so its canonical decomposition needs only **2** nodes instead of 4 — and both of them already hold [3,11] from the last insertion. Watch a node accumulate its second stored interval. In machine terms: [4,10] is **Machine B**, busy from minute 4 to minute 10 — its window sits entirely inside Machine A's, which is exactly why every node that stores B also already stores A.

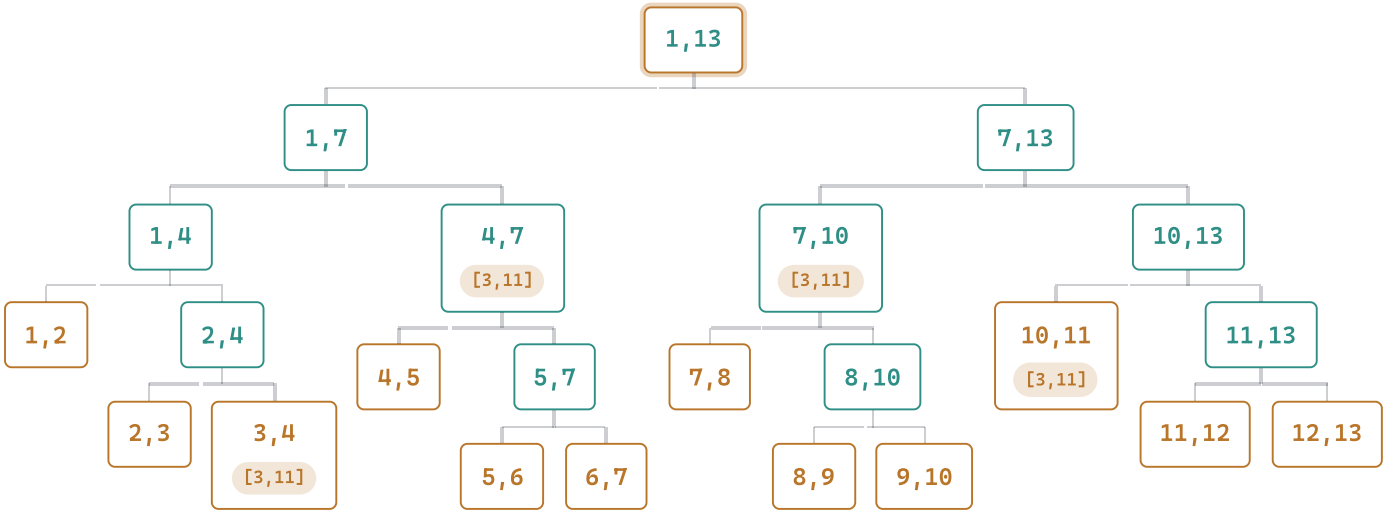
INTERACTIVE — TRACE INSERT(4, 10) NODE BY NODE

STEP 1 OF 6



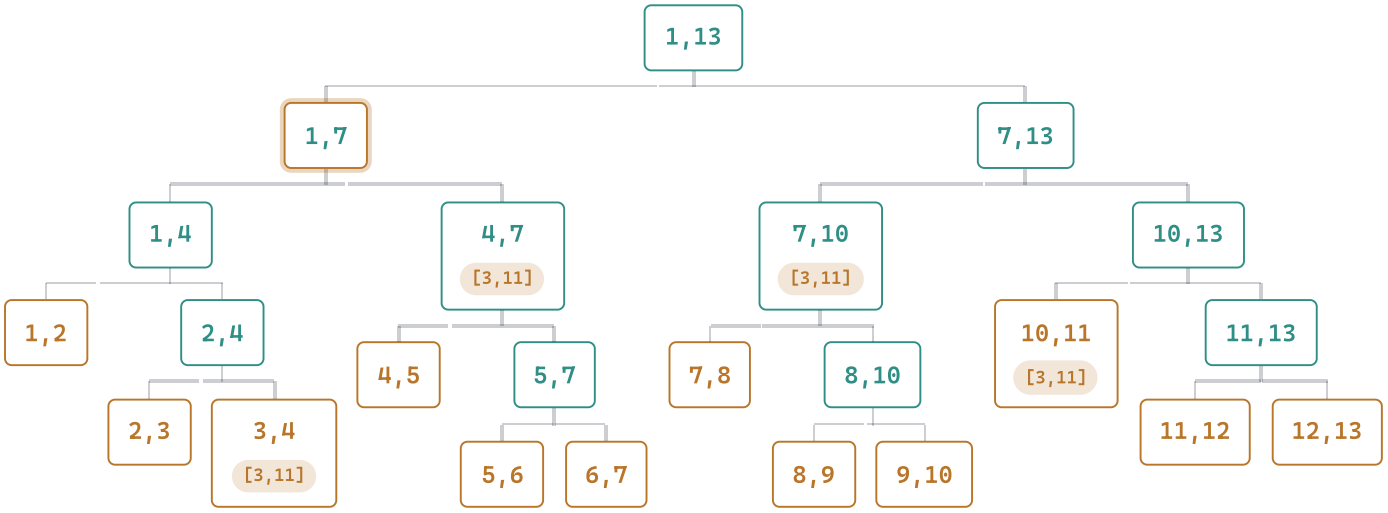
Starting point: the tree already stores [3,11] at [3,4], [4,7], [7,10], [10,11] from the last animation. Now call insert(4, 10, root).

STEP 2 OF 6



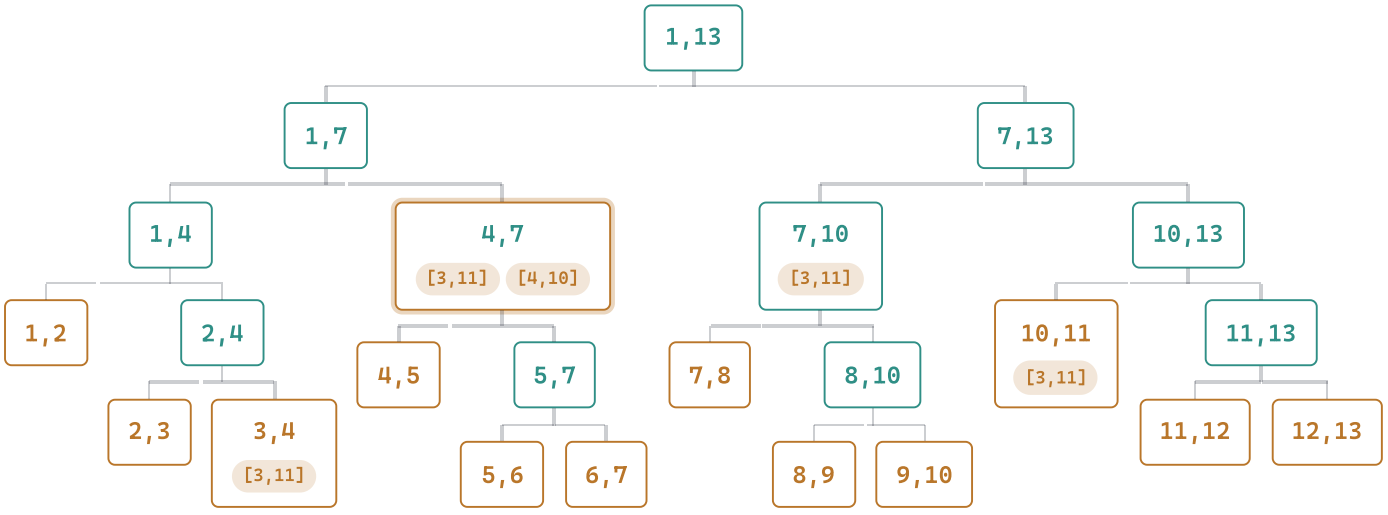
Visit [1,13]: covered? need 4 ≤ 1 — false. mid = 7. 4 < 7 → recurse left. 10 > 7 → recurse right.

STEP 3 OF 6



Visit [1,7]: covered? need 4 ≤ 1 — false. mid = 4. Is 4 < 4? **No** — skip left [1,4] entirely. Is 10 > 4? Yes — recurse right only.

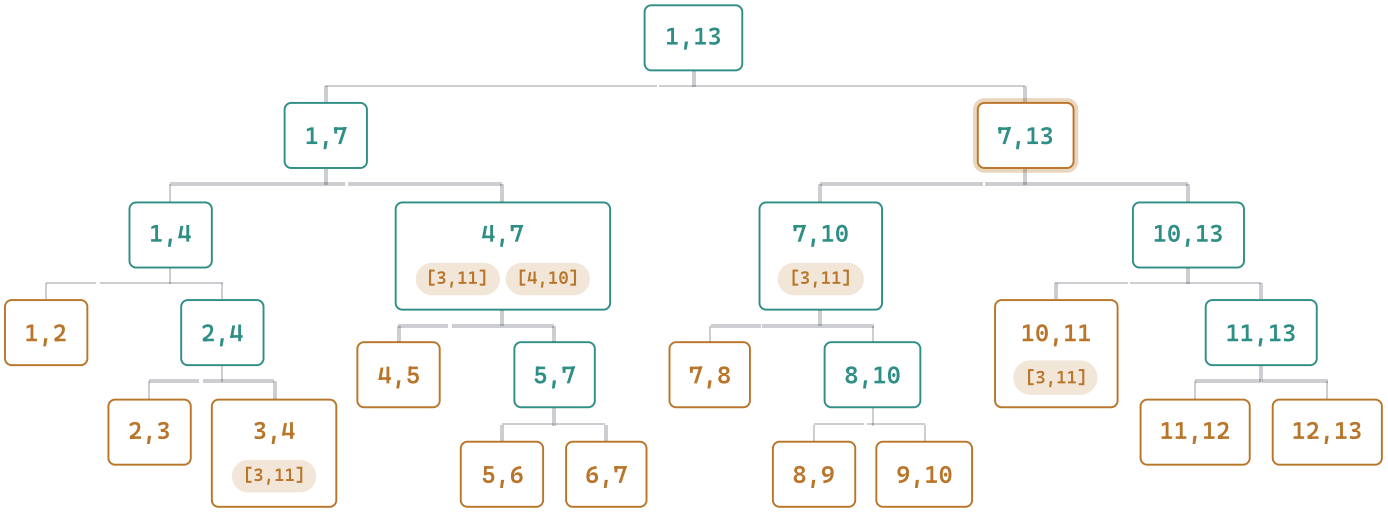
STEP 4 OF 6



NOW HOLDS TWO INTERVALS

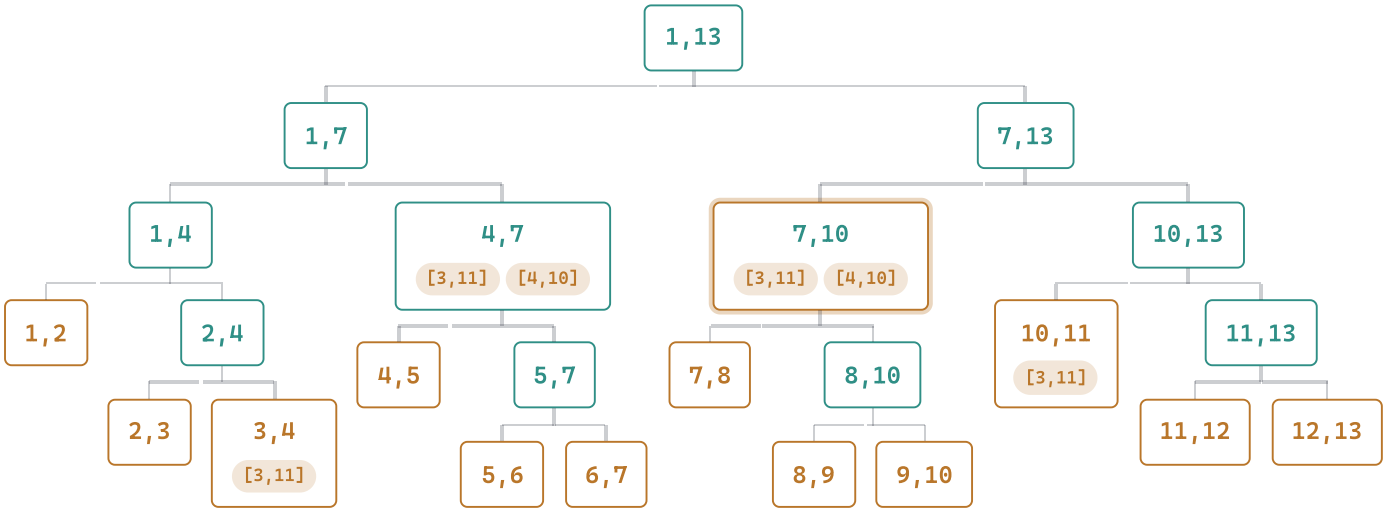
Visit [4,7]: covered? need 4 ≤ 4 **and** 7 ≤ 10 — both true! **Store [4,10] here** — this node already held [3,11], so it now holds **both**. Stop; do not recurse into its children.

STEP 5 OF 6



Visit [7,13] (back to the root's right branch): covered? need 4 ≤ 7 **yes**, but 13 ≤ 10 — **false**. mid = 10. 4 < 10 → recurse left. Is 10 > 10? **No** — skip right [10,13] entirely.

STEP 6 OF 6



CANONICAL DECOMPOSITION OF [4,10] = { [4,7], [7,10] } — 2 NODES

Visit [7,10]: covered? need 4 ≤ 7 **and** 10 ≤ 10 — both true! **Store [4,10] here** too — again on top of the [3,11] already stored. Just 5 nodes visited, only 2 stores — a tidier cover than [3,11]'s 4, purely because 4 and 10 line up exactly with existing split points.

One unique path from root to leaf — collect everything stored on it

WHY THIS WORKS

A unit interval like [5,6] corresponds to exactly **one leaf**, and a tree has exactly one path from root to any leaf. Any interval whose canonical decomposition includes a node on that path necessarily overlaps [5,6] — and by the "no duplication" property, nothing is ever reported twice.

WHAT TO EXPECT

Both [3,11] (**Machine A**) and [4,10] (**Machine B**) happen to be canonically stored at node [4,7] — which sits directly on the path to leaf [5,6]. So this single query should report **both** machines as busy at minute 5, from that one stop, and nothing else.

WHAT "QUERY [5,6]" MEANS, AND HOW EACH STEP DECIDES LEFT VS. RIGHT

This is another **stabbing query** — the same question as L12·01's "which machines are busy at minute [2,3]?" [5,6] is just how this tree names the single point **5**: every unit interval [k,k+1] is a leaf (L12·02), so asking "what covers position 5?" means walking down to leaf [5,6] and collecting everything stored along the way.

The decision rule is the same midpoint test as insert's else-branch: at each node, compute mid = (s(v)+e(v))/2; if 5 < mid, go left; if 5 ≥ mid, go right. Nothing new — it's the identical comparison that just decided v.left vs v.right for [3,11] on the canonical-decomposition slide.

What's genuinely different from insert: insert checks `s < m` *and* `e > m` separately, because a wide interval can straddle the midpoint and need *both* children. A query like [5,6] is only one unit wide — it can never straddle a midpoint, so exactly one of those two conditions ever holds. That's why this walk never branches: one single root-to-leaf path, never two — the whole reason a query is cheap.

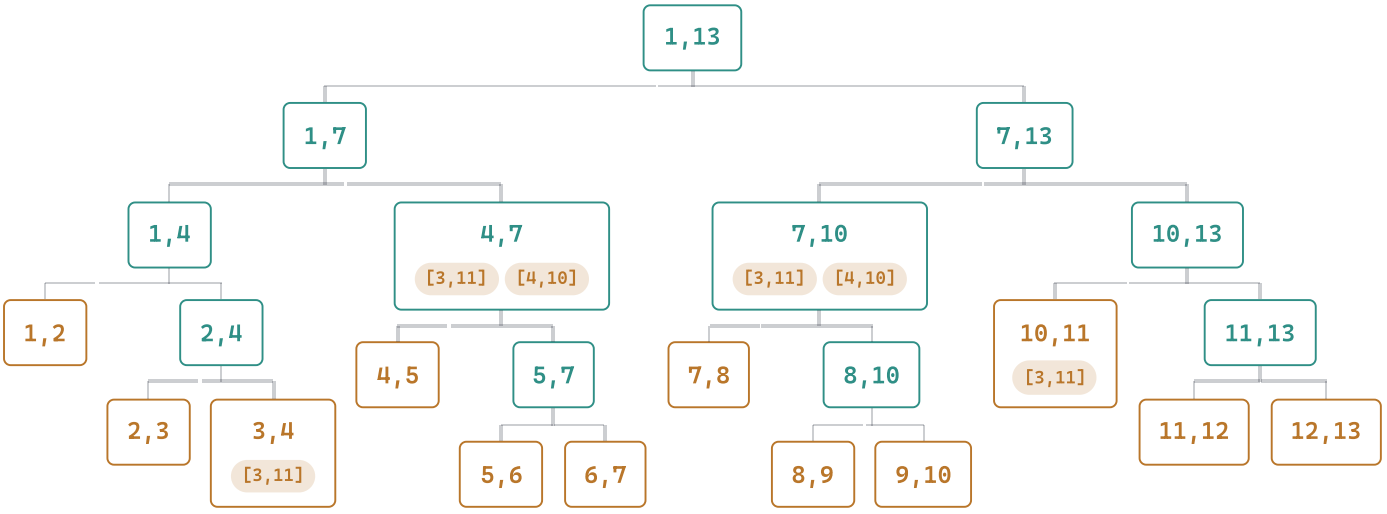
The whole rule, done at every single node, in this order:

- **1. Report:** whatever this node stores — add it to the answer. This has nothing to do with left/right; it happens whether you go on to branch or not.
- **2. Branch** — only if this node isn't the target leaf yet: mid = ⌊(s+e)/2⌋; if **5 < mid**, go left; otherwise, go right.
- **3. Stop** once e = s+1 — that node *is* leaf [5,6], and the walk is over.

That's it — the same three-line checklist, five times in a row below. Nothing changes step to step except the numbers.

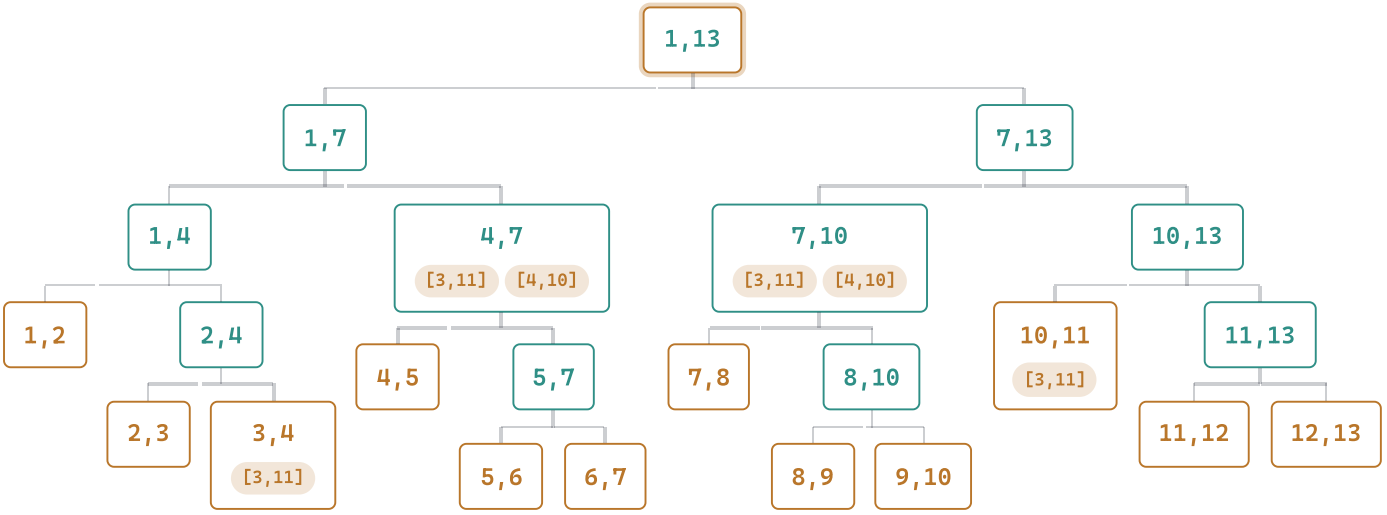
INTERACTIVE — WALK THE PATH TO LEAF [5,6]

STEP 1 OF 7



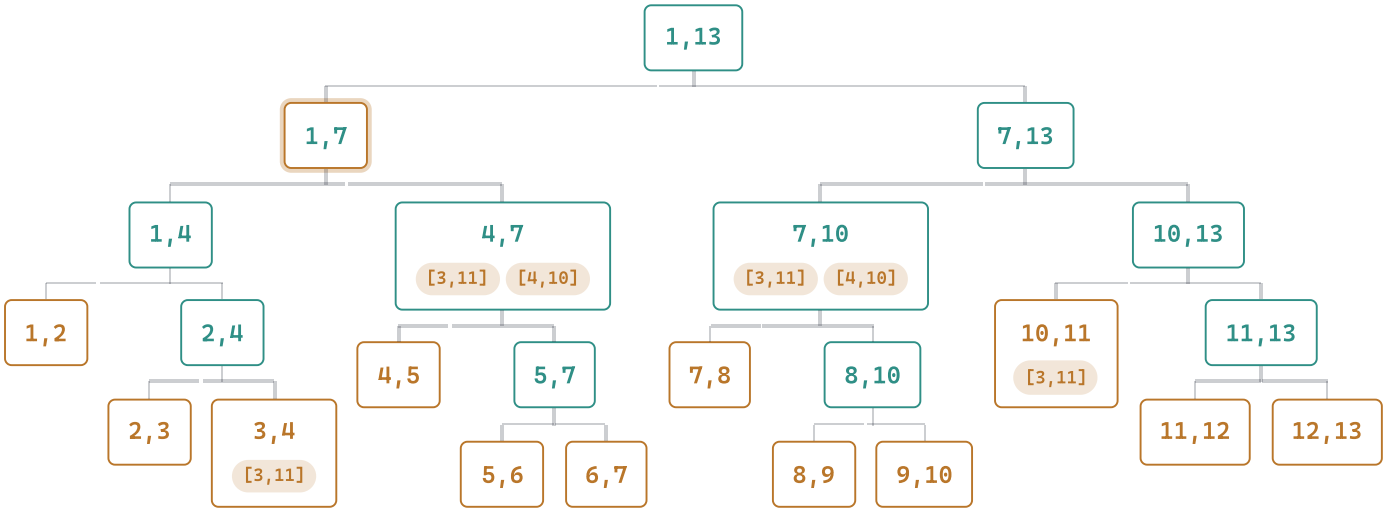
Query: which stored intervals touch unit interval [5,6]? There is exactly one path from the root to leaf [5,6] — walk it and, at every node, apply the same 3-step checklist: report, branch, stop-at-leaf.

STEP 2 OF 7



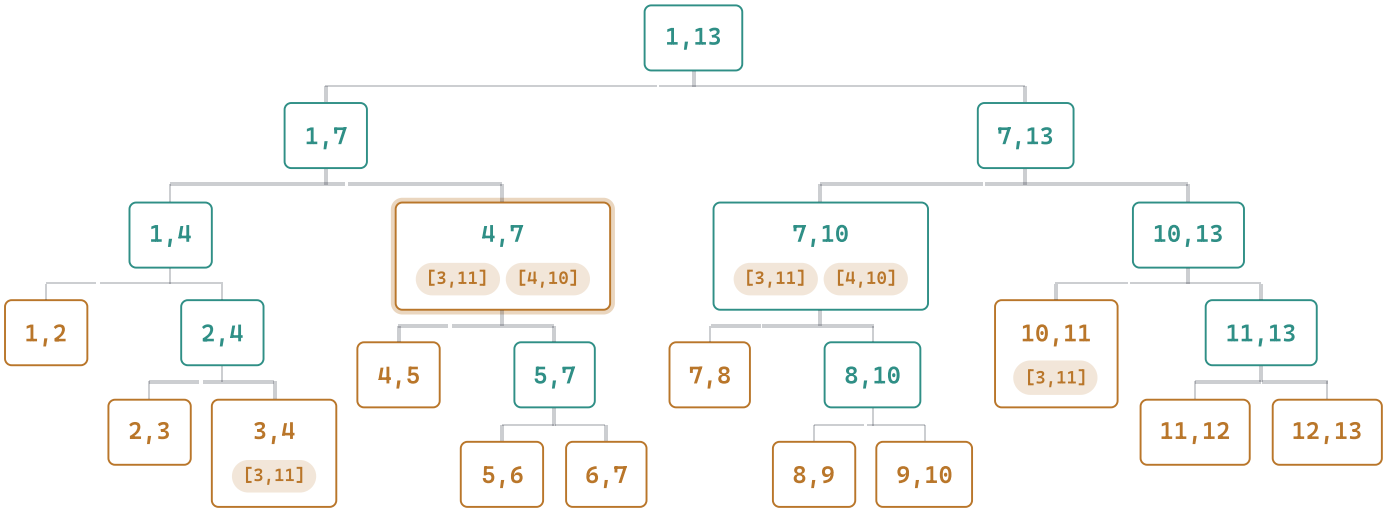
At [1,13]: 1. Report — stores nothing. 2. Branch — mid = ⌊(1+13)/2⌋ = 7; is 5 < 7? **Yes** → go left toward [1,7].

STEP 3 OF 7



At [1,7]: 1. Report — stores nothing. 2. Branch — mid = ⌊(1+7)/2⌋ = 4; is 5 < 4? **No** → go right toward [4,7].

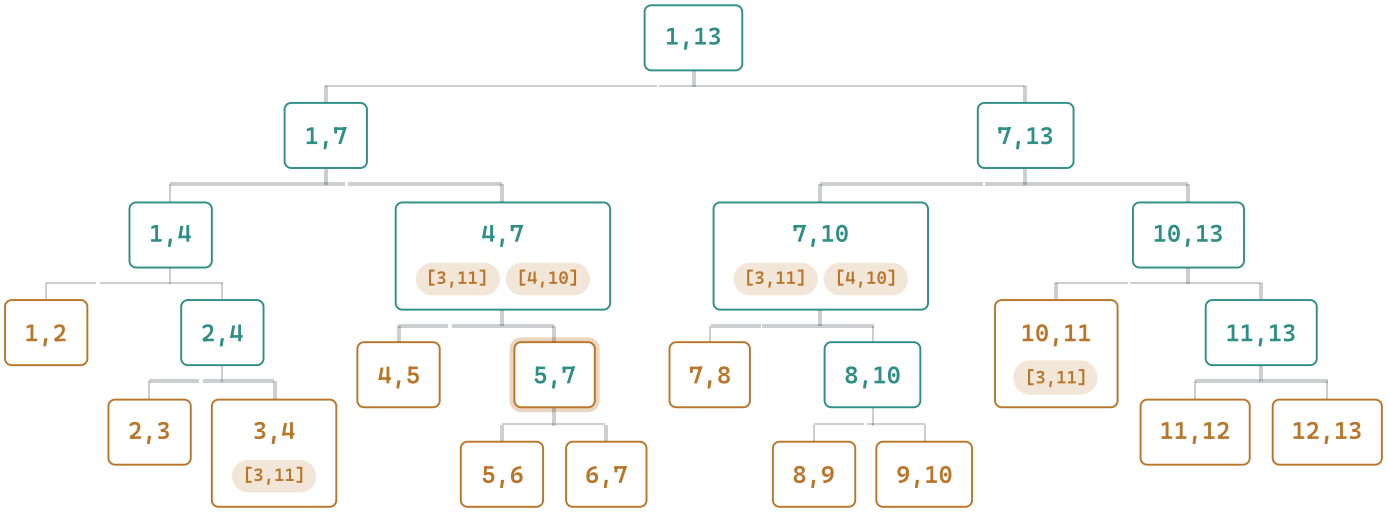
STEP 4 OF 7



REPORT: [3,11] AND [4,10]

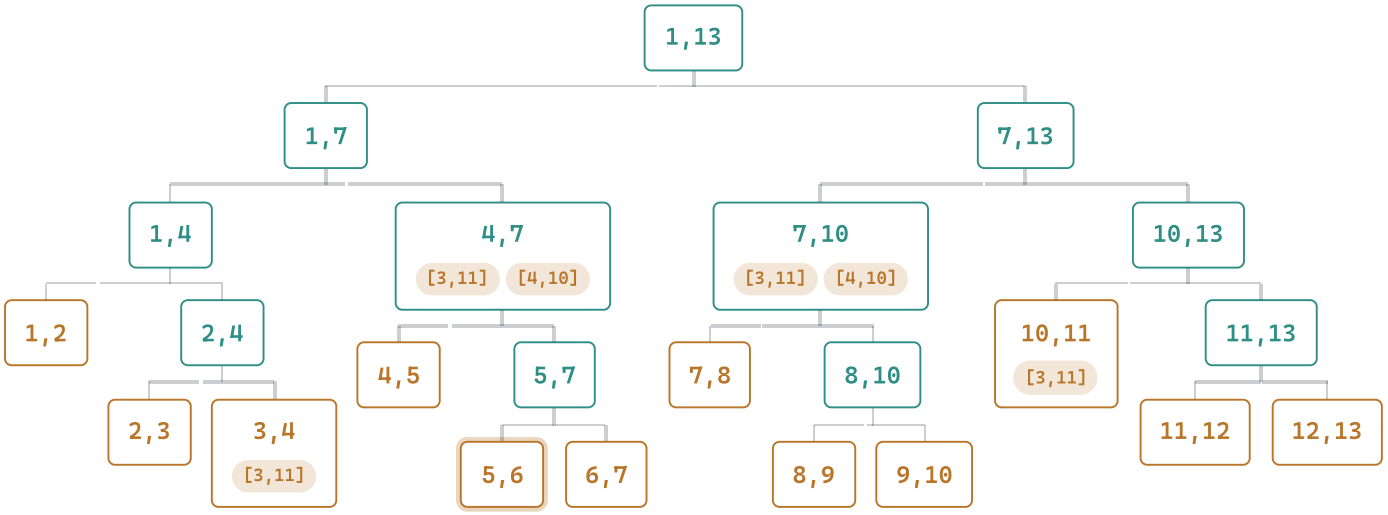
At [4,7]: 1. Report — stores **[3,11] and [4,10]**, add both. 2. Branch — mid = ⌊(4+7)/2⌋ = 5; is 5 < 5? **No** → go right toward [5,7].

STEP 5 OF 7



At [5,7]: 1. Report — stores nothing. 2. Branch — mid = ⌊(5+7)/2⌋ = 6; is 5 < 6? **Yes** → go left toward [5,6].

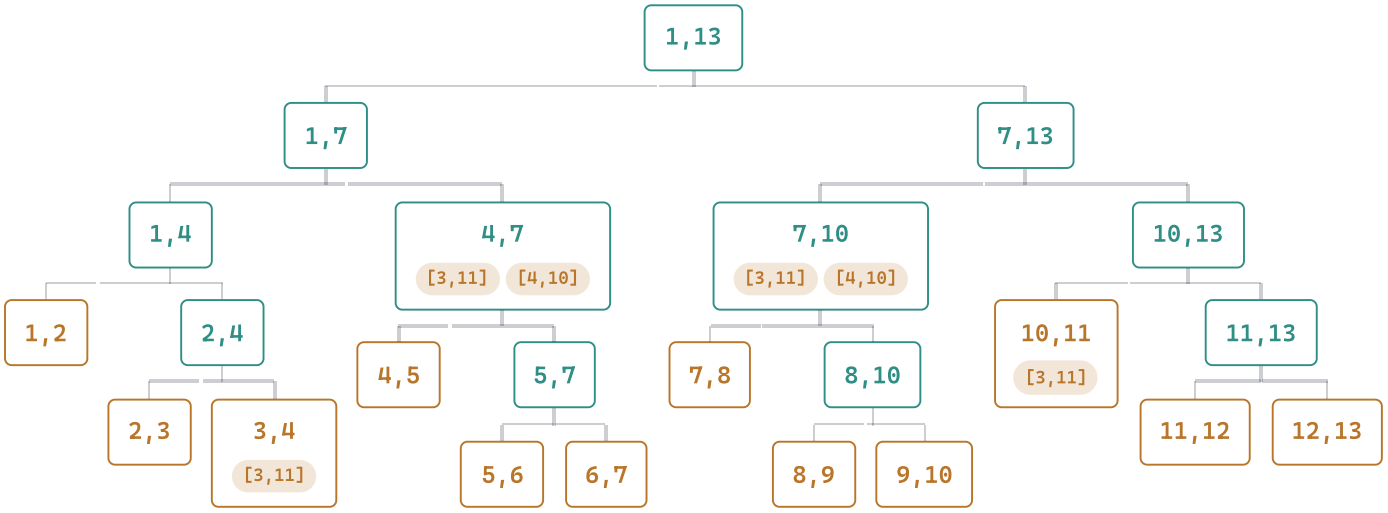
STEP 6 OF 7



TARGET LEAF REACHED — PATH COMPLETE

At [5,6]: 1. Report — stores nothing. 3. Stop — e = s+1, this *is* the target leaf. Path complete.

STEP 7 OF 7



REPORTED = { [3,11], [4,10] } — FOUND IN ONE STOP, NO DUPLICATES POSSIBLE

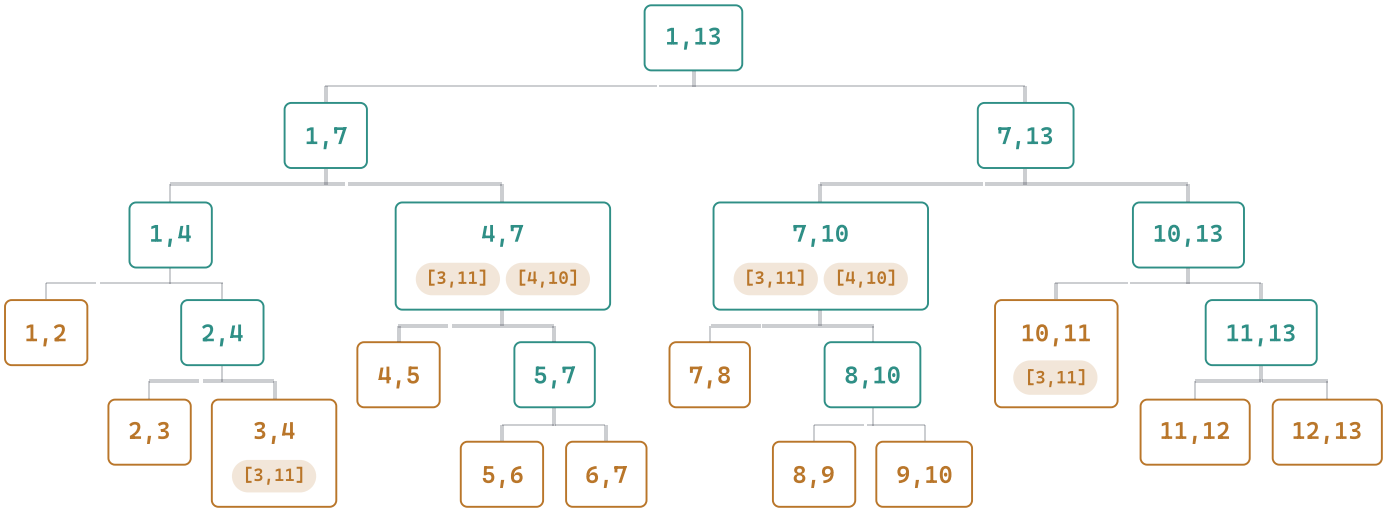
Result: reported set = {[3,11], [4,10]}, both found at the single node [4,7]. Five nodes visited total — O(log n) for n = 13 — and the "no interval in both a node and its ancestor" property guarantees nothing here could ever be double-counted.

Same recursion, remove instead of store

`delete(4,10,v)` follows **exactly the same branching** we just traced for `insert(4,10,v)` — same covered-check, same mid comparisons, node for node. The only difference is the action taken when a node's range is fully covered: remove the interval from its stored set instead of adding it. In the scheduling story: **Machine B** ([4,10]) has finished its job and leaves the tree; **Machine A** ([3,11]) stays behind, untouched. The animation below walks all 5 visited nodes one at a time, exactly like the insert trace did — nothing skipped.

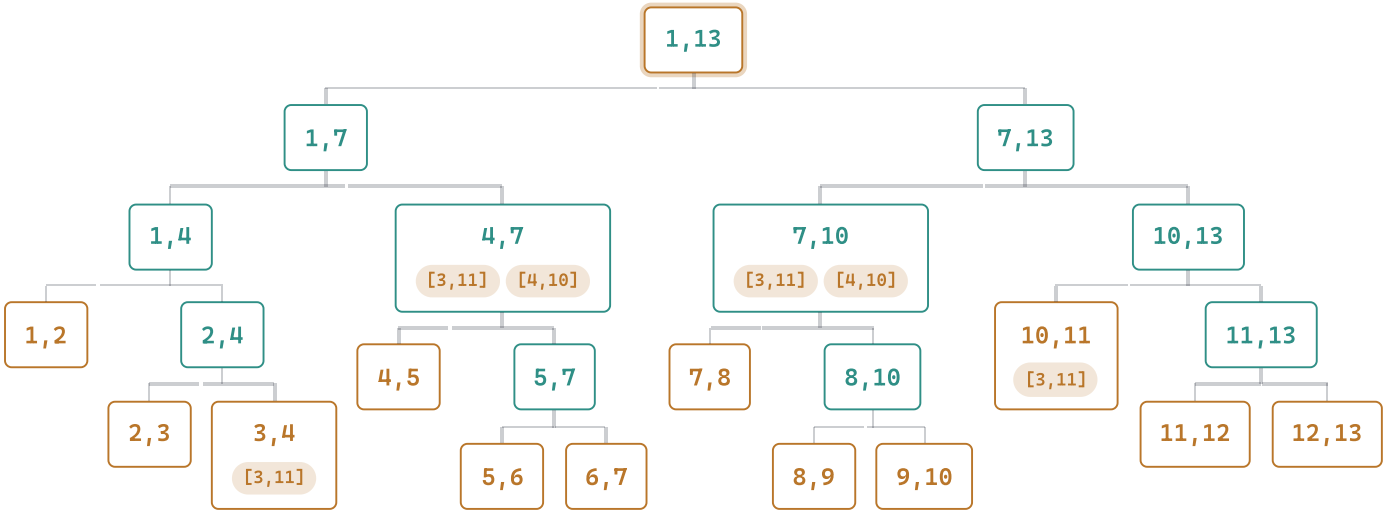
INTERACTIVE — TRACE DELETE(4, 10) NODE BY NODE

STEP 1 OF 7



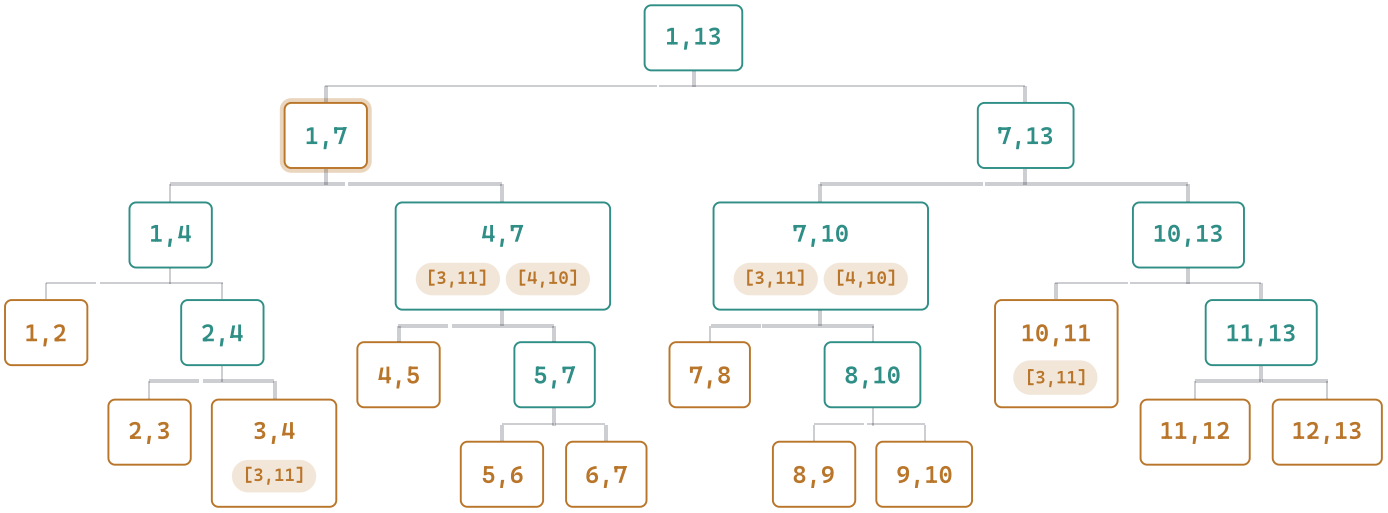
Call: delete(4, 10, root). Same recursion shape as insert(4,10) — same covered-check, same mid comparisons at every node — but the action at a covered node is **remove** instead of store. Watch it node by node.

STEP 2 OF 7



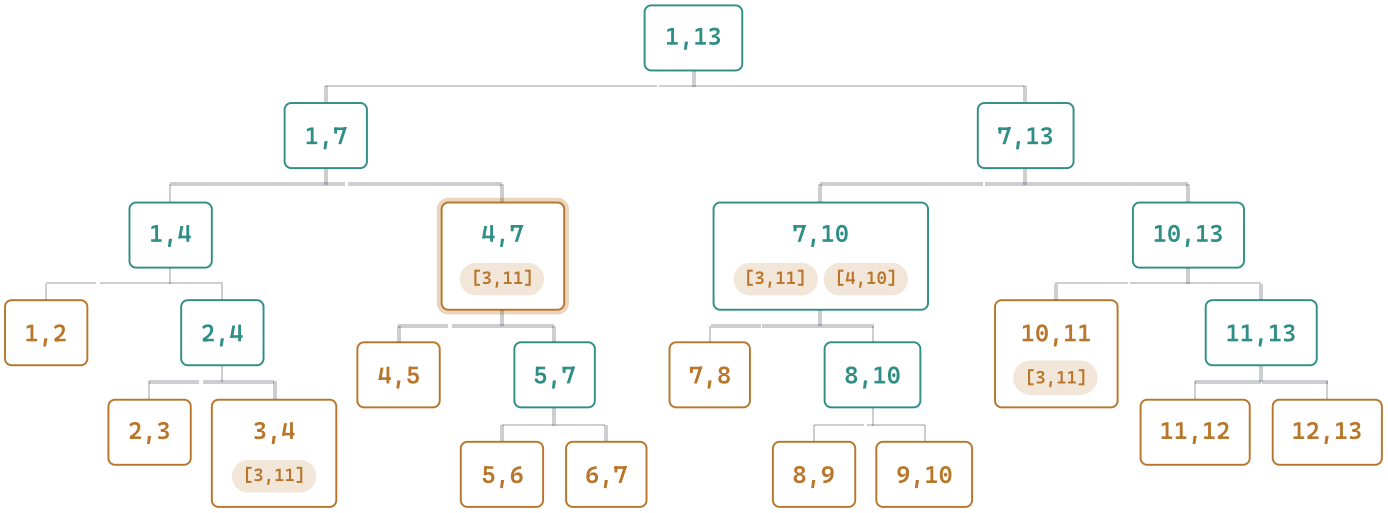
Visit [1,13]: covered? need 4 ≤ 1 — false. mid = 7. 4 < 7 → recurse left. 10 > 7 → recurse right.

STEP 3 OF 7



Visit [1,7]: covered? need 4 ≤ 1 — false. mid = 4. Is 4 < 4? **No** — skip left [1,4] entirely. Is 10 > 4? Yes — recurse right only.

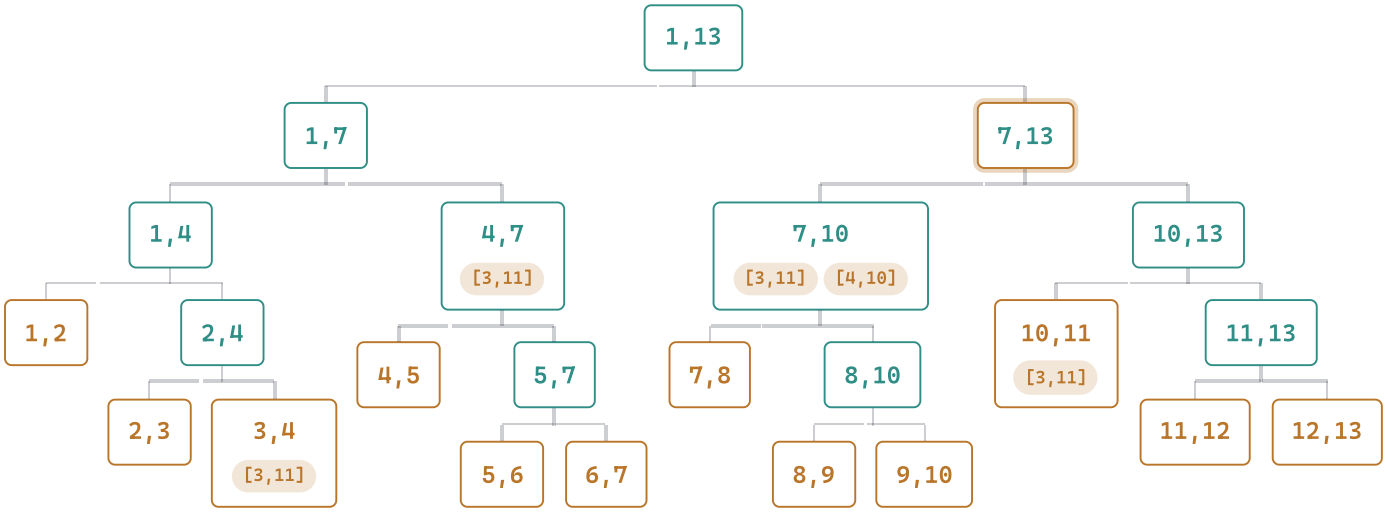
STEP 4 OF 7



NOW HOLDS ONLY [3,11]

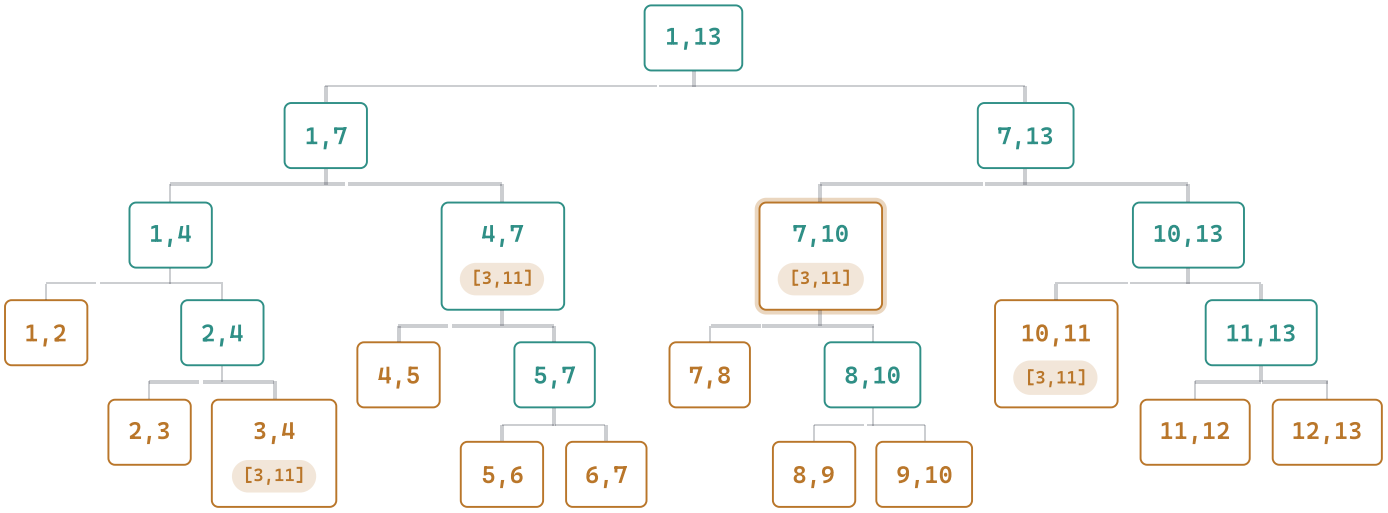
Visit [4,7]: covered? need 4 ≤ 4 **and** 7 ≤ 10 — both true! **Remove [4,10] here** — this node held [3,11] and [4,10]; it now holds only [3,11]. Stop; do not recurse into its children.

STEP 5 OF 7



Visit [7,13] (back to the root's right branch): covered? need 4 ≤ 7 **yes**, but 13 ≤ 10 — **false**. mid = 10. 4 < 10 → recurse left. Is 10 > 10? **No** — skip right [10,13] entirely.

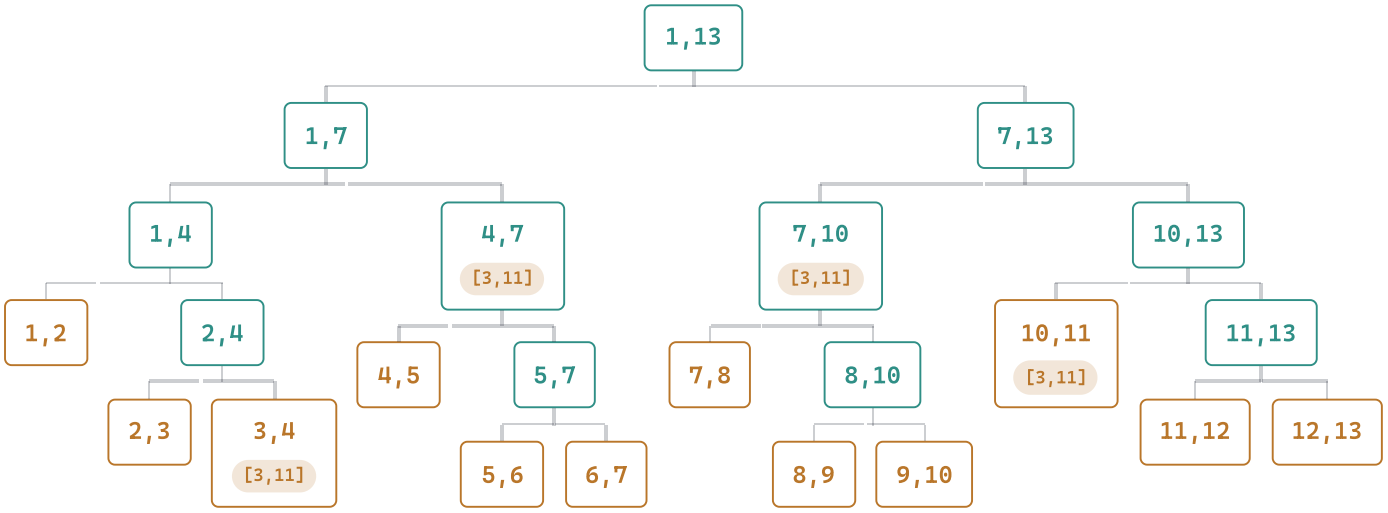
STEP 6 OF 7



NOW HOLDS ONLY [3,11]

Visit [7,10]: covered? need 4 ≤ 7 **and** 10 ≤ 10 — both true! **Remove [4,10] here** too. Both canonical nodes are now back to storing only [3,11] — exactly the state the tree was in right before [4,10] was ever inserted.

STEP 7 OF 7



DELETE COMPLETE

Done. [4,10] has been fully removed from its canonical decomposition, {[4,7], [7,10]}. Same 5 nodes visited, same O(log n) cost, same path — node for node — as inserting it.

One structure, four costs

OPERATION	COST	WHY
Build the skeleton (universe [1,n])	O(n)	One-time; total nodes = 2·(number of leaves) − 1, all created up front.
Insert an interval	O(log n)	Canonical decomposition touches O(log n) nodes — verified: 10 visits, 4 stores, for [3,11].
Delete an interval	O(log n)	Identical recursion to insert.
Stabbing query [a,a+1]	O(log n + k)	One root-to-leaf path (O(log n)), reporting k stored intervals found on it.

What if there's no fixed universe [1,n] to pre-split?

WHERE SEGMENT TREES START TO STRAIN

A segment tree needs the **whole integer range known in advance** — build once over [1,n], then insert intervals into it. But real scheduling data rarely looks like that: meeting times are real-valued and span years; genome coordinates run into the billions; network events arrive with no known upper bound. Rebuilding a segment tree's skeleton every time the universe changes is wasteful, or simply impossible if it's unbounded.

THE INTERVAL TREE IDEA

Stop indexing the number line. Index the **intervals themselves**, directly, the way a BST indexes keys — ordered by their low endpoint — and **augment** every node with one extra number: the largest high endpoint anywhere in its subtree. That single number is enough to search, insert, and delete in O(log n), fully dynamically, with no universe size fixed in advance.

Real uses: calendar/meeting-room scheduling, overlapping-gene detection in genome analysis, network event simulation.

An augmented BST, ordered by low endpoint

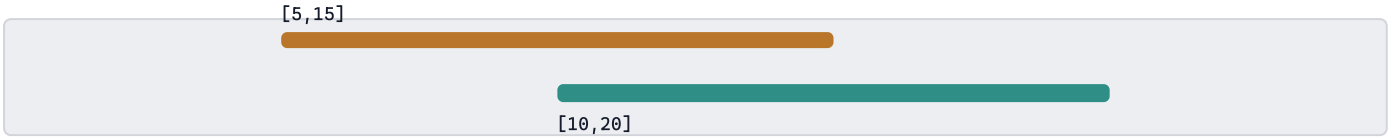
WHAT EVERY NODE STORES

- Its own interval, **[low, high]**.
- **max** — the largest high endpoint anywhere in the subtree rooted here (itself included).

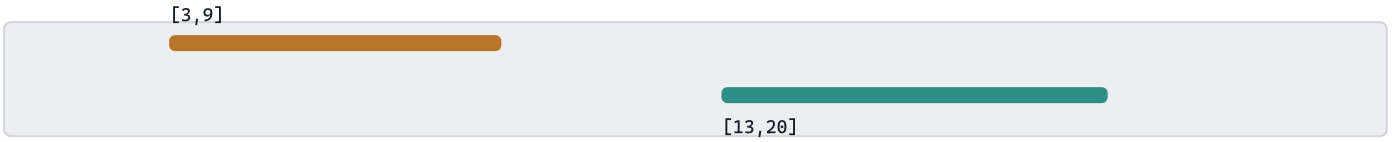
BST ordering is by **low** value only. The underlying tree can be a plain BST, or a self-balancing one — we'll use an **AVL tree** (Lectures 7–9) as the skeleton, since you've already mastered its rotations. The genuinely new machinery today is the **max** augmentation, not rebalancing.

THE OVERLAP CONDITION

Intervals [a,b] and [c,d] overlap exactly when **a ≤ d and c ≤ b** — equivalently, "neither one ends before the other begins."



Overlapping: 5 ≤ 20 and 10 ≤ 15 — both hold.



Not overlapping: 3 ≤ 20 holds, but 13 ≤ 9 fails — [3,9] ends before [13,20] begins.

Insert (15,20), (10,30), (17,19), (5,20), (12,15), (30,40) — one at a time

WHAT EVERY INSERTION ACTUALLY DOES

1. Descend by comparing **low** values, exactly like an ordinary BST insert.
2. Place the new node as a leaf; its own max starts equal to its own high.
3. Walk back **up** the same path to the root, and at every ancestor recompute max = the largest of: its own high, its left child's max (if any), its right child's max (if any).

WATCH FOR THE "NOTHING CHANGED" STEPS

Most of the max-update steps below will report **no change** — the new value simply wasn't the largest one already known at that ancestor. That's just as important to see as the steps where max *does* change: it's the reason this update only ever costs O(depth), never more.

INTERACTIVE — EVERY COMPARISON, PLACEMENT, AND MAX UPDATE

STEP 1 OF 16

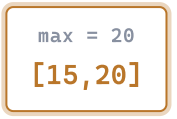
1. (15,20) 2. (10,30) 3. (17,19) 4. (5,20) 5. (12,15) 6. (30,40)

EMPTY TREE

Start: empty interval tree. Insert (15,20), (10,30), (17,19), (5,20), (12,15), (30,40) — one at a time, ordered by **low** endpoint. The row above tracks progress: done, current, and still to come.

STEP 2 OF 16

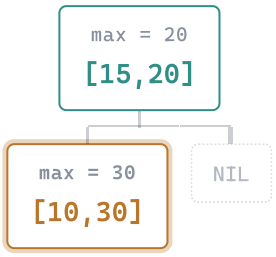
1. (15,20) 2. (10,30) 3. (17,19) 4. (5,20) 5. (12,15) 6. (30,40)



Insert (15,20): tree is empty → becomes the root. Its own max = its own high = 20 (no children yet).

STEP 3 OF 16

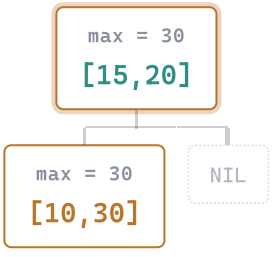
1. (15,20) **2. (10,30)** 3. (17,19) 4. (5,20) 5. (12,15) 6. (30,40)



Insert (10,30) — descend: compare low 10 vs root's low 15 → 10 < 15, go left. Root has no left child → place [10,30] here. Its own max = its own high = 30 (leaf).

STEP 4 OF 16

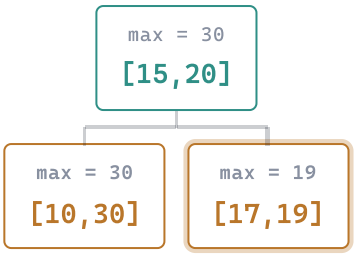
1. (15,20) **2. (10,30)** 3. (17,19) 4. (5,20) 5. (12,15) 6. (30,40)



Insert (10,30) — propagate up to the root: root.max = max(own high 20, left.max 30) = **30**. Changed from 20 to 30.

STEP 5 OF 16

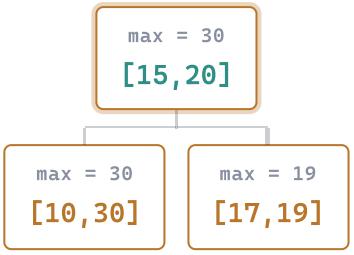
1. (15,20) 2. (10,30) **3. (17,19)** 4. (5,20) 5. (12,15) 6. (30,40)



Insert (17,19) — descend: compare low 17 vs root's low 15 → 17 ≥ 15, go right. Root has no right child → place [17,19] here. Its own max = 19 (leaf).

STEP 6 OF 16

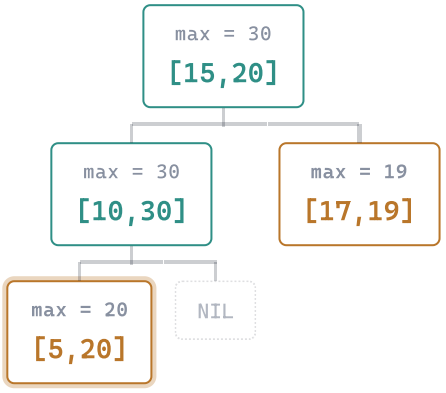
1. (15,20) 2. (10,30) **3. (17,19)** 4. (5,20) 5. (12,15) 6. (30,40)



Insert (17,19) — propagate up to the root: root.max = max(own high 20, left.max 30, right.max 19) = **30 — unchanged.** 19 wasn't big enough to matter. Still worth checking, and still O(1) to check.

STEP 7 OF 16

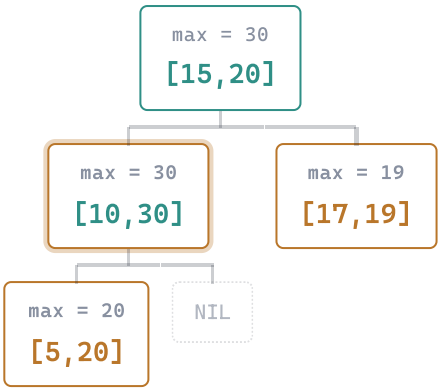
1. (15,20) 2. (10,30) 3. (17,19) **4. (5,20)** 5. (12,15) 6. (30,40)



Insert (5,20) — descend: 5 < 15 → left to [10,30]; 5 < 10 → left again; [10,30] has no left child → place [5,20] here. Own max = 20 (leaf).

STEP 8 OF 16

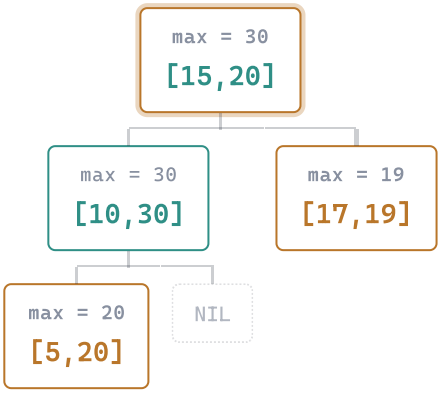
1. (15,20) 2. (10,30) 3. (17,19) **4. (5,20)** 5. (12,15) 6. (30,40)



Insert (5,20) — propagate, step 1 of 2 (at [10,30]): max = max(own high 30, left.max 20) = **30 — unchanged.** [10,30]'s own high was already the biggest thing in its subtree.

STEP 9 OF 16

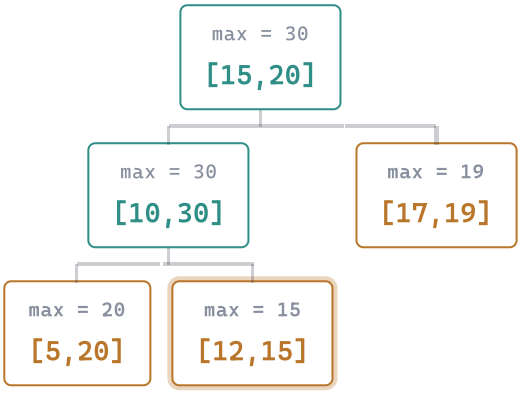
1. (15,20) 2. (10,30) 3. (17,19) **4. (5,20)** 5. (12,15) 6. (30,40)



Insert (5,20) — propagate, step 2 of 2 (at the root): max = max(20, left.max 30, right.max 19) = **30 — unchanged again.** Two ancestors checked, zero values changed — and that's fine; the check is what guarantees correctness, not the change.

STEP 10 OF 16

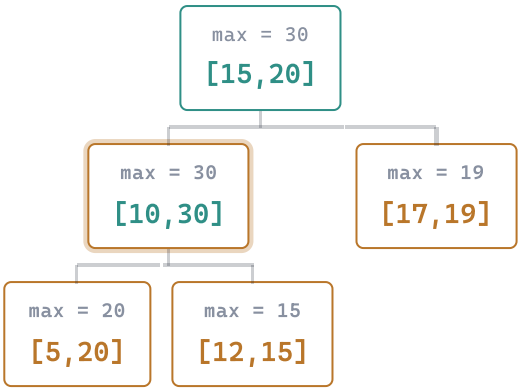
1. (15,20) 2. (10,30) 3. (17,19) 4. (5,20) **5. (12,15)** 6. (30,40)



Insert (12,15) — descend: 12 < 15 → left to [10,30]; 12 ≥ 10 → right; [10,30] has no right child → place [12,15] here. Own max = 15 (leaf).

STEP 11 OF 16

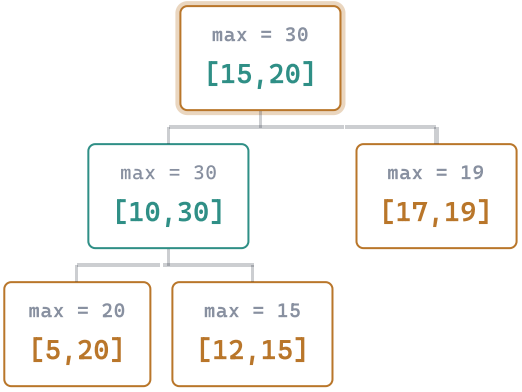
1. (15,20) 2. (10,30) 3. (17,19) 4. (5,20) **5. (12,15)** 6. (30,40)



Insert (12,15) — propagate, step 1 of 2 (at [10,30]): max = max(30, left.max 20, right.max 15) = **30 — unchanged.**

STEP 12 OF 16

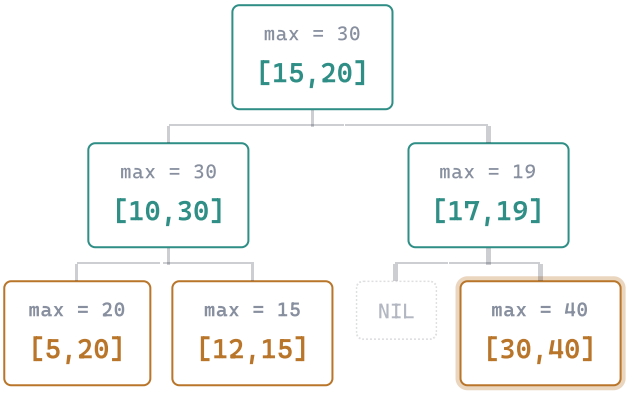
1. (15,20) 2. (10,30) 3. (17,19) 4. (5,20) **5. (12,15)** 6. (30,40)



Insert (12,15) — propagate, step 2 of 2 (at the root): max = max(20, 30, 19) = **30 — unchanged.**

STEP 13 OF 16

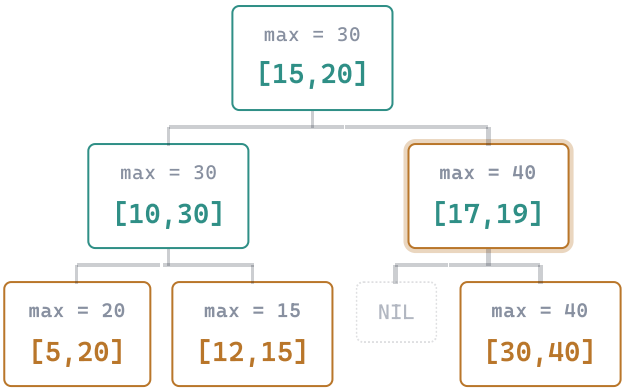
1. (15,20) 2. (10,30) 3. (17,19) 4. (5,20) 5. (12,15) **6. (30,40)**



Insert (30,40) — descend: 30 ≥ 15 → right to [17,19]; 30 ≥ 17 → right again; [17,19] has no right child → place [30,40] here. Own max = 40 (leaf).

STEP 14 OF 16

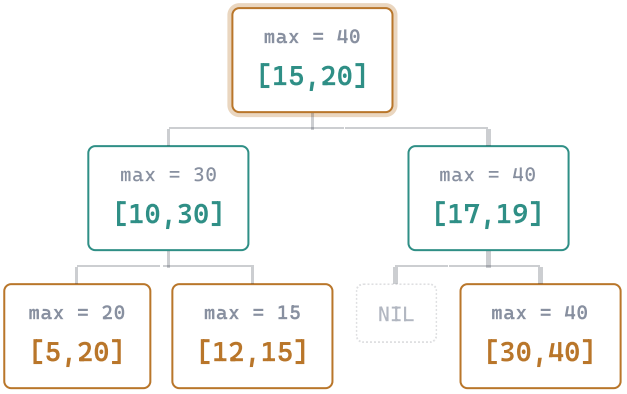
1. (15,20) 2. (10,30) 3. (17,19) 4. (5,20) 5. (12,15) **6. (30,40)**



Insert (30,40) — propagate, step 1 of 2 (at [17,19]): max = max(own high 19, right.max 40) = **40 — changed from 19.** [17,19] had no left child, so this is the first real change on this path.

STEP 15 OF 16

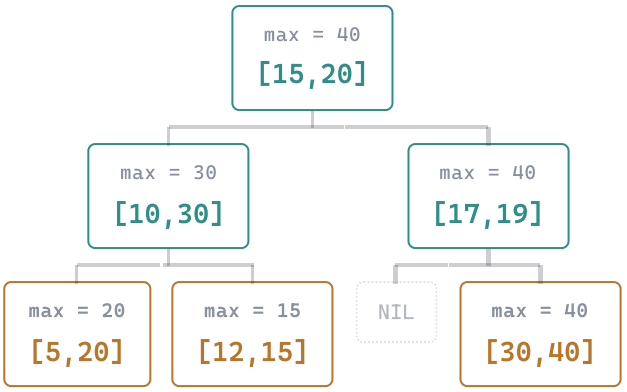
1. (15,20) 2. (10,30) 3. (17,19) 4. (5,20) 5. (12,15) **6. (30,40)**



Insert (30,40) — propagate, step 2 of 2 (at the root): max = max(20, left.max 30, right.max 40) = **40 — changed from 30.** The new deepest value has now surfaced all the way to the root.

STEP 16 OF 16

1. (15,20) 2. (10,30) 3. (17,19) 4. (5,20) 5. (12,15) 6. (30,40)



FINAL TREE — MATCHES THE SOURCE DIAGRAM EXACTLY

Final tree: root [15,20] (max 40), left subtree rooted at [10,30] (max 30) with leaves [5,20] and [12,15], right subtree rooted at [17,19] (max 40) with its one child [30,40]. This insertion order happened to need **zero** AVL rotations — the balance factors already stay within ±1 at every node. Six insertions, fourteen max-checks along the way, only two of them actually changed a value.

Report every overlap — and prune whatever can't possibly match

THE SEARCH RULE AT EVERY NODE V

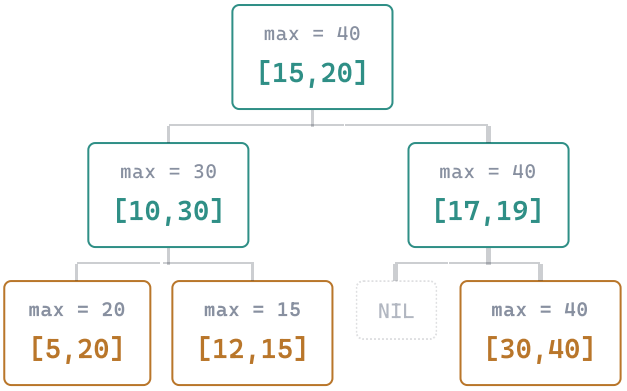
```
findOverlaps(v, i):
  if v is null: return
  if overlap(v.interval, i): report v.interval
  if v.left exists and v.left.max ≥ low(i):
      findOverlaps(v.left, i)
  if v.right exists and low(v) ≤ high(i):
      findOverlaps(v.right, i)
```

WHY THE PRUNING IS SAFE

v.left.max is the **largest high endpoint anywhere in the left subtree**. If even that largest value is smaller than low(i), *nothing* in the left subtree can overlap i — skip it entirely, without visiting a single node inside it. That's the entire payoff of the max augmentation.

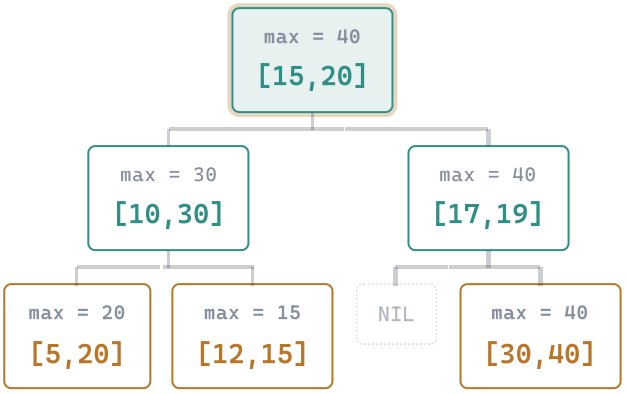
INTERACTIVE – TRACE FINDOVERLAPS(ROOT, [14,16])

STEP 1 OF 8



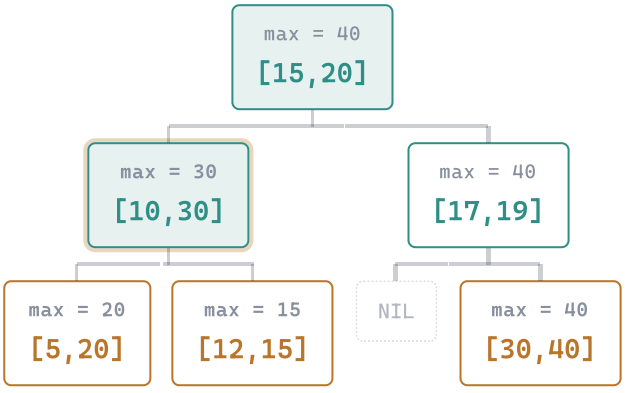
Query: findOverlaps(root, [14,16]) — report every interval that overlaps [14,16], using left.max to prune whole subtrees.

STEP 2 OF 8



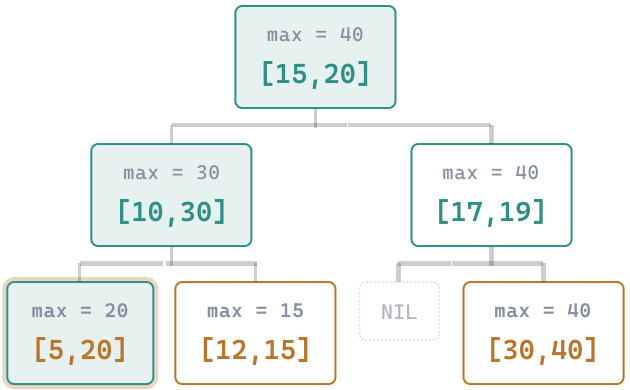
At [15,20]: overlap? 14 ≤ 20 and 15 ≤ 16 — both true. **Report [15,20].** Left child [10,30] exists and its max (30) ≥ low(i)=14 → will recurse left. Right child [17,19] exists and low(v)=15 ≤ high(i)=16 → will recurse right.

STEP 3 OF 8



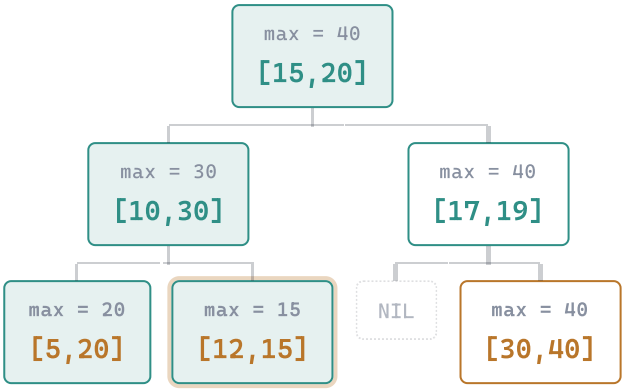
At [10,30] (left child): overlap? 14 ≤ 30 and 10 ≤ 16 — both true. **Report [10,30].** Left child [5,20] exists, its max (20) ≥ 14 → recurse left. Right child [12,15] exists, low(v)=10 ≤ 16 → recurse right.

STEP 4 OF 8



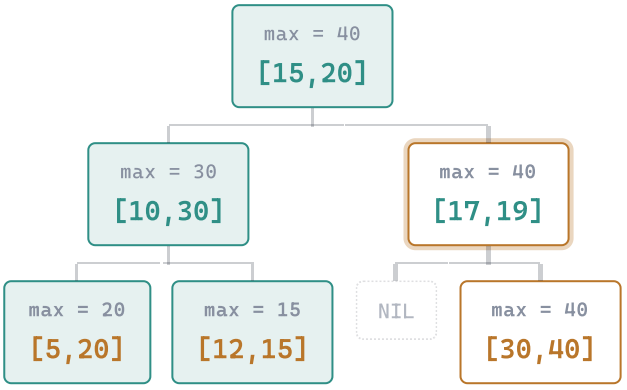
At [5,20] (leaf): overlap? 14 ≤ 20 and 5 ≤ 16 — both true. **Report [5,20].** No children — this branch ends here.

STEP 5 OF 8



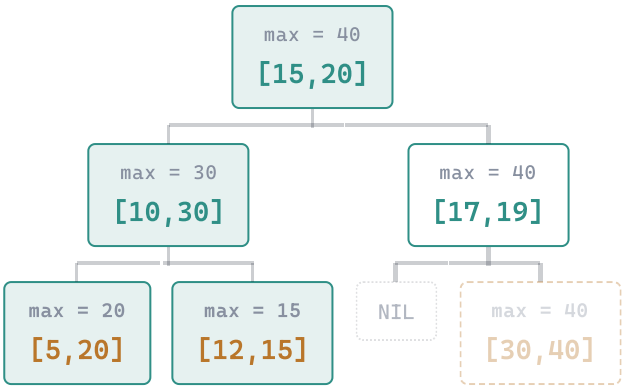
At [12,15] (leaf): overlap? 14 ≤ 15 and 12 ≤ 16 — both true. **Report [12,15].** No children — this branch ends too.

STEP 6 OF 8



Back to the root's right child, [17,19]: overlap? 14 ≤ 19 is true, but 17 ≤ 16 is **false** — no overlap, do not report. No left child to check.

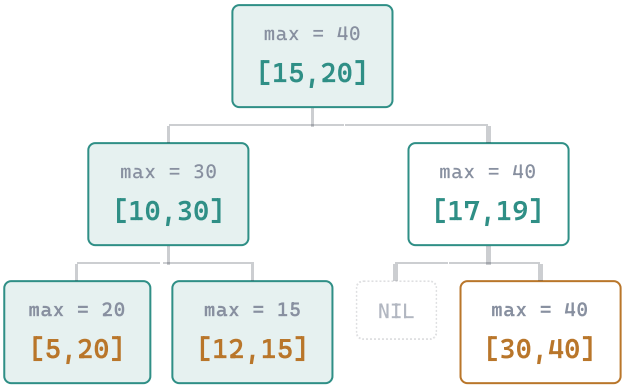
STEP 7 OF 8



PRUNED — NEVER VISITED

Pruning check at [17,19]'s right child, [30,40]: the rule is low(v) ≤ high(i), i.e. 17 ≤ 16 — **false**. Skip entirely. [30,40] is never even opened; we already know from [17,19]'s own low value that nothing on this side can reach back to 16.

STEP 8 OF 8



Result: reported = {[15,20], [10,30], [5,20], [12,15]} — exactly 4 intervals. One full subtree ([30,40]) was pruned without a single comparison inside it — the entire reason the max augmentation exists.

Why [14,16] is a genuinely different question from [5,6]

THE SEGMENT TREE'S QUESTION — A SNAPSHOT

query [5,6] asked: **"who's busy at this one instant — minute 5?"** A single point in time. The answer is whichever machines happen to be running *right then*, and only them.

THE INTERVAL TREE'S QUESTION — A WINDOW

findOverlaps(root, [14,16]) asks something broader: **"who's busy at any point during this whole 2-minute window?"** Not an instant — a span. A machine doesn't need to be running for the *whole* window, or even most of it — sharing a single minute with the query is enough to count.

CHECKED AGAINST ALL SIX MACHINES FROM THE BUILD ANIMATION

MACHINE'S BUSY WINDOW	OVERLAPS [14,16]?	REPORTED?
[15,20]	Yes — running through 15, 16	✓
[10,30]	Yes — the whole window sits inside its schedule	✓
[5,20]	Yes — running through 14, 15, 16	✓
[12,15]	Yes — shares just minute 14 with the window	✓
[17,19]	No — doesn't start until minute 17	✗
[30,40]	No — doesn't start until minute 30 (pruned, never even checked)	✗

Notice [12,15] makes the list despite overlapping the window by a single minute — that's the whole point of a range query. Tying back to L12·10's motivation: **"who's busy right now"** is a point question, but **"does this new 2pm–3pm meeting conflict with anything already on the calendar?"** is a range question — you're not asking about one instant, you're asking whether a whole proposed window collides with any existing booking. That is the real question interval trees are built to answer.

Same problem, two different assumptions about the data

OPERATION	COST
Insert	O(log n)
Delete	O(log n)
Overlap search (one match)	O(log n)
Overlap search (report all k matches)	O(log n + k)

ASPECT	SEGMENT TREE	INTERVAL TREE
What's indexed	The number line [1,n] itself	The set of intervals directly
Needs a known universe size?	Yes — built once over [1,n]	No — works over any real-valued domain
Underlying shape	Fixed by n, never rebalanced	A balanced BST (AVL/Red-Black) by low endpoint
Best query type	Stabbing query — a single point/unit range	Overlap query — against any interval, any size
Query cost	O(log n + k)	O(log n) for one match, O(log n + k) for all
Typical use	Fixed discretized domains: pixel rows, router filters, time slots	Dynamic real-valued domains: calendars, genomes, network events

Two structures, one underlying question

- **A segment tree pre-splits a fixed number line into O(n) nodes once**, then stores each interval in the O(log n) nodes of its canonical decomposition — verified today with [3,11] (4 nodes) and [4,10] (2 nodes) on the same [1,13] skeleton.
- **An interval tree augments a balanced BST with a max_high value at every node**, letting it prune entire subtrees during an overlap search — verified today with a 6-interval build and a search for [14,16] that correctly found all 4 overlaps while never even visiting [30,40].
- Both cut a naive O(n)-per-query scan down to **O(log n)**-class costs — the same recurring lesson from Lectures 7–11: pre-organize the data once, pay a little bookkeeping on every update, and every query gets dramatically cheaper.

LECTURE 13 — NEXT

Tries and Digital Search Trees

Apply prefix searching and dictionary operations — indexing by the shape of the key itself, not by comparison.

HOMEWORK — BRING TO LECTURE 13

- Build the segment tree for universe [1,9]. How many leaves, how many total nodes, and what height does the formula predict?
- On that tree, trace insert(2,7) node by node, listing every stored node exactly as we did for [3,11].
- Insert (2,6), (1,4), (9,15), (5,8) into an interval tree in that order. Draw the final tree with every node's max value.
- Using the interval tree built in this lecture, trace findOverlaps for the query [25,35]. Which nodes get pruned, and why?
- In 3–4 sentences: why can't a segment tree be used directly for real-valued (non-integer) intervals without modification?
- Reading: de Berg et al., *Computational Geometry*, Ch. 10 (segment trees) & Ch. 14 (interval trees); CLRS §14.3 (interval trees).

Questions?

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

> *Next: Lecture 13 — Tries and Digital Search Trees.*