Pairing Heaps & Double-Ended Priority Queues

Two structures for the two things Lecture 19 left unresolved: a heap simple enough to actually deploy (not just cite in a proof), and a queue that answers "give me the smallest" and "give me the largest" from the same collection.

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

~110 minutes
Session outcome: implement advanced heap and DEPQ operations

~116 MIN

Today, in two parts

- **Part A: Why pairing heaps?** 00–06
  Fibonacci heaps are the theoretical champion nobody deploys. Pairing heaps are what people actually use.
- **Structure: one tree, a sibling list, no bookkeeping fields** 06–11
  No parent, degree, or mark field anywhere — a clever pointer trick replaces all three.
- **MELD: the one primitive everything is built from** 11–14
  Compare two roots, link the smaller under the larger. O(1). Always.
- **Advantages & limitations** 14–19
  Faster in practice than Fibonacci heaps — with a famous open problem sitting inside its analysis.
- **Animation: INSERT** 19–24
  Five inserts, watching the root change or stay put each time.
- **Animation: DECREASE-KEY** 24–29
  Detach using the sibling-list trick, no parent pointer needed, then meld back in.
- **Animation: REMOVE-MIN, two-pass scheme** 29–44
  Five orphaned subtrees, paired left to right, then combined right to left.
- **Complexity — and an honest open problem** 44–48
  Most bounds are proven. One famous one isn't, even today.
- **Part B: Why double-ended priority queues?** 48–54
  External quicksort needs both "smallest so far" and "largest so far" from one live structure.
- **Generic methods: dual, total, and leaf correspondence** 54–59
  Bolt a min-heap and a max-heap together — three ways to do it, with a real trade-off.
- **Interval heaps: the custom, elegant answer** 59–65
  One tree, every node holds an interval — a min-heap and a max-heap, embedded in the same shape.
- **Animation: building an interval heap** 65–71
  Eight inserts from empty — and why every one of them fits without bubbling.
- **Animation: INSERT, all four cases** 71–88
  Fits directly, bubbles via the min side, bubbles via the max side, fills an odd slot.
- **Animation: REMOVE-MIN** 88–98
  Promote from the last node, then reinsert it down through the embedded min-heap.
- **Advantages & limitations** 98–102
  Half the space of a dual structure, at the cost of trickier bookkeeping per node.
- **Complexity & where this runs** 102–109
  External sorting, and any system that needs both ends of a priority order live.

Recap & what's next 109–116

Lecture 21: Comparative Study of Heap Structures.

PART A

A single unordered tree, a sibling list, and one primitive — MELD — that every other operation reduces to.

The theoretical champion nobody deploys

REAL-LIFE EXAMPLE — THE SAME ROUTER, A DIFFERENT ENGINEER

Lecture 19 ended with a caveat: Fibonacci heaps win the asymptotic argument for Dijkstra's algorithm, but the parent/child/left/right/degree/mark bookkeeping per node, plus cascading-cut logic, carries real constant-factor cost. An engineer actually shipping a routing engine or a game-AI pathfinder needs something that wins *in practice*, at the graph sizes that really show up — not just in the limit as n → ∞.

WHAT EXPERIMENTS ACTUALLY SHOW

Pairing heaps were invented specifically to answer this. Experimentally, they are consistently **faster than Fibonacci heaps** in practice, simpler to implement correctly, carry smaller runtime overhead per operation, and use **less memory per node** — no degree field, no mark bit, no parent pointer at all.

THE TRADE-OFF, STATED HONESTLY UP FRONT

Pairing heaps give up something for this simplicity: several of their amortized bounds — most notably DECREASE-KEY — are **not fully proven** to match Fibonacci heaps' O(1). They are *conjectured* to be very fast and observed to be fast, but the tightest known proofs land somewhere between the conjecture and Fibonacci heaps' guarantee. We will be precise about exactly which bounds are solid and which are open when we reach the complexity slide.

One tree. A sibling list. No parent, degree, or mark field anywhere.

WHAT A PAIRING HEAP ACTUALLY IS

- A pairing heap is **a single min-heap-ordered tree** — not a forest like binomial or Fibonacci heaps. Every node's key is ≤ every key in its subtree.
- Children are **unordered** and held in a **doubly-linked sibling list** (not circular, unlike Fibonacci heaps' root/child lists).
- Each node stores exactly three pointers: `child` (its first child), `left` and `right` (siblings) — plus its key. That is the entire node.

THE TRICK THAT REMOVES THE PARENT POINTER

A node x is the **first** child in its sibling list exactly when `x.left.child = x`. So instead of wasting a slot on a separate parent pointer, the `left` pointer of a first child is simply repurposed to point at the **parent** — and every other sibling's `left` points at its left neighbour as usual.

This one trick is what makes DECREASE-KEY and REMOVE cheap without any cascading-cut bookkeeping: you can always find "where does this node detach from" in O(1), using only `left`/`right` — no `degree` or `mark` field is ever needed.

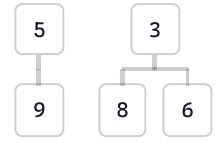
Compare two roots. The larger tree becomes a child of the smaller. O(1).

THE ONE PRIMITIVE

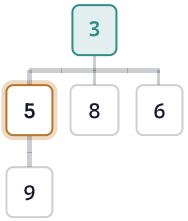
Given two pairing-heap trees, MELD compares their roots. The tree rooted at the **larger** key becomes the new **leftmost child** of the tree rooted at the smaller key — one pointer rewire, no traversal of either tree's interior. Every other operation in this lecture — INSERT, DECREASE-KEY, REMOVE-MIN, REMOVE — is built entirely out of calls to MELD.

WORKED EXAMPLE

Tree 1: root 5, child [9]. Tree 2: root 3, children [8, 6].



MELD THESE TWO TREES



5 > 3, SO TREE(5) BECOMES 3'S NEW LEFTMOST CHILD

Fast in practice, simple to build — with one famous asterisk

ADVANTAGES

- **Every core operation reduces to MELD**, an O(1) actual-cost primitive — the whole implementation is short and hard to get subtly wrong.
- **Smaller node footprint** than Fibonacci heaps: 3 pointers + a key (4 fields total), versus 4 pointers + a key + degree + mark (7 fields total).
- **Experimentally faster** than Fibonacci heaps across realistic workloads — better cache behaviour, fewer pointer chases per operation.
- **INSERT and MELD are O(1) actual cost** — both are single MELD calls.

LIMITATIONS

- **REMOVE-MIN needs a good multi-way-meld strategy.** Melding the orphaned children left-to-right in a naive chain gives Θ(n) amortized cost — genuinely bad. The two-pass and multipass schemes fix this to O(log n) amortized, but you must implement one of them correctly.
- **DECREASE-KEY's tight amortized bound is a famous open problem.** It is conjectured to be O(1) amortized (matching Fibonacci heaps) and performs like it in practice, but the best *proven* upper bound is more subtle, and a matching lower bound rules out a naive O(1) proof via the obvious potential function.
- **Worst-case degree and height are both Θ(n)** for a single operation sequence (e.g. inserting keys in sorted order) — exactly like a Fibonacci heap tree can look, before any consolidation.

Create a 1-node tree, MELD it in — that's the whole operation

Building a pairing heap with five inserts: 15, 9, 20, 6, 12. Watch the root change twice (whenever the new key is smaller) and stay put three times (whenever it isn't).

INTERACTIVE — INSERT 15, 9, 20, 6, 12

STEP 1 OF 10

EMPTY HEAP

Start with an empty pairing heap. INSERT creates a 1-node tree and MELDs it with the existing heap.

STEP 2 OF 10

Insert 15. It becomes the whole tree — trivially the root.

STEP 3 OF 10

NEW 1-NODE TREE {9}

EXISTING HEAP

Insert 9: first create a brand-new 1-node tree holding just {9}. Now MELD it with the existing tree(15) — compare the two roots, 9 vs. 15.

STEP 4 OF 10

MELD result: 9 < 15, so 9 wins — 15 becomes 9's new child.

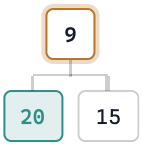
STEP 5 OF 10

NEW 1-NODE TREE {20}

EXISTING HEAP

Insert 20: create the 1-node tree {20}, then MELD it with the existing tree(9→[15]) — compare 20 vs. 9.

STEP 6 OF 10

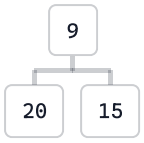


MELD result: 20 > 9, so 9 wins — {20} becomes 9's new LEFTMOST child, pushing 15 to second position.

STEP 7 OF 10

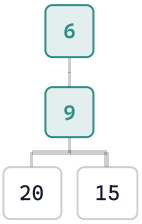
NEW 1-NODE TREE {6}

EXISTING HEAP



Insert 6: create the 1-node tree {6}, then MELD it with the existing tree(9→[20,15]) — compare 6 vs. 9.

STEP 8 OF 10

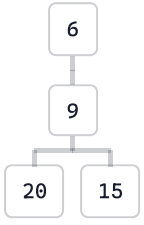


MELD result: 6 < 9, so 6 wins — the ENTIRE tree(9→[20,15]) becomes 6's new child.

STEP 9 OF 10

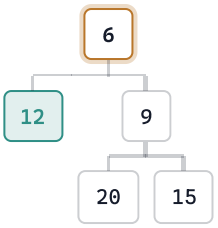
NEW 1-NODE TREE {12}

EXISTING HEAP



Insert 12: create the 1-node tree {12}, then MELD it with the existing tree(6→[9(→[20,15])]) — compare 12 vs. 6.

STEP 10 OF 10



5 inserts | root changed twice (9, then 6) | Each: O(1) actual

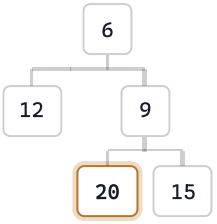
MELD result: 12 > 6, so 6 wins — {12} becomes 6's new leftmost child. Final tree: root 6, children [12, 9(→[20,15])].

Detach using the sibling list, update the key, MELD back in

Starting from INSERT's result: root 6, children [12, 9(→[20, 15])]. We'll decrease node 20 to 2. Since there is no parent pointer, the ONLY way to check for a violation is to just always detach and re-meld.

INTERACTIVE — DECREASE-KEY(20, TO 2)

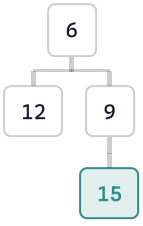
STEP 1 OF 4



Starting tree: root 6, children [12, 9(→[20,15])]. We decrease 20 to 2. Since nodes carry no parent pointer, DECREASE-KEY always detaches the node and re-melds — there is no cheaper "check the parent first" shortcut.

STEP 2 OF 4

REMAINING TREE

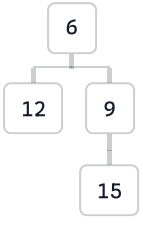


DETACHED

Detach: 20 is the first child of 9, so 20.left currently points to 9 (the parent-pointer trick). Rewire 9's child pointer to 15 (the next sibling), and 15's left pointer to 9. 20 is now a lone detached tree — O(1), no traversal needed.

STEP 3 OF 4

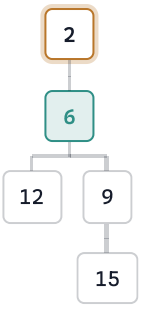
REMAINING TREE



KEY UPDATED

Update the detached node's key from 20 to 2.

STEP 4 OF 4



1 detach + 1 meld | Actual cost: O(1) | New root: 2

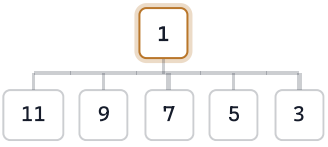
MELD the 1-node tree {2} back with the remaining tree (root 6). 2 < 6, so 2 wins — the whole remaining tree becomes 2's new child. Final: root 2, single child 6(→[12, 9(→15)]).

Five orphaned subtrees, paired left to right, combined right to left

A fresh tree built by inserting 1, 3, 5, 7, 9, 11 in increasing order — the worst case for degree, exactly as noted in the trade-offs slide. Result: root 1, children [11, 9, 7, 5, 3], all leaves. Watch every pairing and every combine — nothing is skipped.

INTERACTIVE — REMOVE-MIN(H)

STEP 1 OF 11



Starting tree (built by inserting 1,3,5,7,9,11 in increasing order — the worst case for degree): root 1, children [11,9,7,5,3], all leaves.

STEP 2 OF 11

ROOT 1 REMOVED — 5 ORPHANED SUBTREES

Remove the root (1) — save it to return. Its 5 children are orphaned: 11, 9, 7, 5, 3 (left to right). Now they must be melded back into one tree.

STEP 3 OF 11

Pass 1 of the two-pass scheme: examine subtrees left to right, melding pairs. First pair: (11, 9).

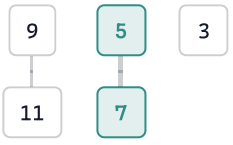
STEP 4 OF 11

9 < 11, so 9 wins — 11 becomes 9's child. First melded pair done: 9(→11).

STEP 5 OF 11

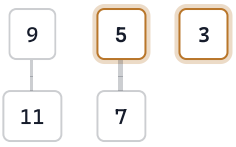
Second pair: (7, 5).

STEP 6 OF 11



5 < 7, so 5 wins — 7 becomes 5's child. Second melded pair done: 5(→7).

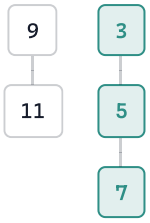
STEP 7 OF 11



5 SUBTREES IS ODD — THE LEFTOVER (3) MELDS WITH THE LAST NEWLY-MADE PAIR

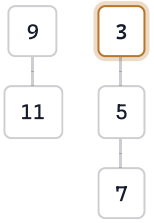
Odd count (5 subtrees, 2 pairs made so far) — the leftover subtree, 3, melds with the *last newly generated* pair, 5(→7), not with the first one.

STEP 8 OF 11



3 < 5, so 3 wins — the pair 5(→7) becomes 3's child. Pass 1 complete: 5 subtrees have become 2 trees: [9(→11), 3(→[5(→7)])].

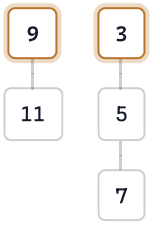
STEP 9 OF 11



RIGHTMOST TREE OF PASS 1 BECOMES THE WORKING TREE

Pass 2: start with the RIGHTMOST tree from Pass 1 as the "working tree" — that's 3(→[5(→7)]).

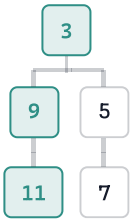
STEP 10 OF 11



COMBINE REMAINING TREES RIGHT TO LEFT, ONE AT A TIME

Combine the remaining trees into the working tree, one at a time, from right to left. Only one remains: 9(→11). Compare 3 vs. 9.

STEP 11 OF 11



`Returned: 1` `Melds: 4 (3 in Pass 1, 1 in Pass 2)` `New root: 3`

3 < 9, so 3 wins — 9(→11) becomes 3's new child. Pass 2 complete: ONE final tree remains. REMOVE-MIN returned 1, and cost only 4 melds total — not the Θ(n) the "bad way" could cost.

WHY NOT JUST MELD LEFT TO RIGHT?

The "bad way": `currentTree = compareLink(currentTree, nextTree)` repeated left to right. If you alternate n/2 inserts then n/2 remove-mins in the worst order, the total remove-min cost becomes $(n/2-1)+\ldots+2+1+0 = \Theta(n^2)$ — if insert is O(1) amortized, that forces remove-min to be Θ(n) amortized. Genuinely bad.

THE MULTIPASS ALTERNATIVE

Put all orphaned subtrees in a FIFO queue; repeatedly dequeue 2, meld them, enqueue the result; stop at 1 tree. Same O(log n) amortized bound as two-pass, but two-pass shows better observed performance in practice — which is why it's the one we animated in full.

Most bounds are proven. One famous one isn't.

OPERATION	ACTUAL COST	AMORTIZED COST
MELD	O(1)	O(1)
INSERT	O(1)	O(1)
FIND-MIN	O(1)	O(1)
DECREASE-KEY	O(1)	conjectured O(1); best proven bound is looser — a genuine open problem
REMOVE-MIN (two-pass or multipass)	O(degree of root)	O(log n) — proven
REMOVE (arbitrary node)	O(n) worst case	O(log n) — proven, same machinery as REMOVE-MIN

n = number of nodes at the time of the operation. Verified today: INSERT and DECREASE-KEY both cost a small constant number of pointer rewires regardless of heap size; REMOVE-MIN's two-pass scheme on a 5-child root took exactly 4 melds to finish (3 in pass 1, 1 in pass 2) — matching $\lceil \log_2 5 \rceil$-ish work, not the n^2 blow-up of the naive approach.

PART B

One collection, two questions always answerable: what's the smallest,
and what's the largest — right now.

Sometimes you need BOTH ends of the priority order, live

DEFINITION

A **double-ended priority queue (DEPQ)** supports: `isEmpty()`, `size()`, `getMin()`, `getMax()`, `put(x)`, `removeMin()`, `removeMax()`. A regular priority queue only ever exposes one end; a DEPQ exposes both, simultaneously, at all times.

REAL-LIFE EXAMPLE — EXTERNAL QUICKSORT

Quicksort partitions into Left (≤ pivot), Middle (the pivot), Right (≥ pivot). When the data is too big for memory, an **external** quicksort makes the Middle group as large as possible using a DEPQ: fill the DEPQ from disk; for every further element, if it's ≤ the DEPQ's current min, ship it straight to Left; if it's ≥ the current max, ship it to Right; otherwise it belongs in the middle — remove either the current min or max to make room, and let the DEPQ hold onto the new element instead. This needs **both ends live, continuously** — exactly the DEPQ contract.

Bolt a min-heap and a max-heap together — three ways to do it

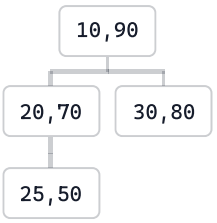
METHOD	IDEA	TRADE-OFF
Dual structure	Maintain a full min-PQ and a full max-PQ of every element, with a correspondence pointer linking each element's copy in one to its copy in the other.	Simplest to implement; each element effectively stored twice — most space.
Total correspondence	Split elements roughly in half between a min-PQ and a max-PQ; pair every min-PQ element with a distinct max-PQ element (a,b) where priority(a) ≤ priority(b). One buffered element if the count is odd.	Half the space of dual structure; noticeably more complex algorithms.
Leaf correspondence	Same halved split, but only *leaf* elements of the min-PQ and max-PQ are required to be paired — internal nodes need no pairing.	Same space savings as total correspondence, but generally the **fastest** of the three in practice.

Any of these, layered on a PQ structure that also supports an efficient `remove(theNode)` — a binary heap, a pairing heap, or a height-biased leftist tree — gives put/removeMin/removeMax in O(log n) (amortized, for pairing heaps) and everything else in O(1). But there's a more elegant, purpose-built option next.

One tree. Every node holds an interval, not a single key.

DEFINITION

An **interval heap** is a complete binary tree in which every node (except possibly the last) holds **two elements**, a ≤ b — representing the closed interval [a,b]. The rule: every child's interval is **contained** in its parent's interval. If the last node holds a single element c, then a ≤ c ≤ b for its parent's interval [a,b].



A VALID 4-NODE INTERVAL HEAP — EVERY CHILD'S INTERVAL FITS INSIDE ITS PARENT'S

TWO HEAPS, HIDING IN PLAIN SIGHT

- All the **left** endpoints, read together, form a valid **min-heap**.
- All the **right** endpoints, read together, form a valid **max-heap**.
- The root's left endpoint is always the overall minimum; the root's right endpoint is always the overall maximum. getMin/getMax are O(1) — just read the root.
- Stored compactly in an array exactly like an ordinary heap, just with two slots per position. Height is Θ(log n) for n elements.

Eight inserts, from empty, to the heap every later slide reuses

Every example so far started from a heap already fully formed: root [10,90], left [20,70], right [30,80], left-left [25,50]. Here it is built from nothing, one INSERT at a time — and every single value happens to fit its parent's containment on arrival, so watch for what a bubble-free build actually looks like before L20·12 shows you what happens when one doesn't.

INTERACTIVE — INSERT 10, 90, 20, 70, 30, 80, 25, 50

STEP 1 OF 9

EMPTY INTERVAL HEAP

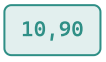
Start empty. We will insert 10, 90, 20, 70, 30, 80, 25, 50 in that order — and, as a deliberate choice, none of them will need to bubble.

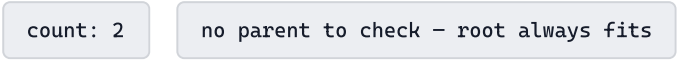
STEP 2 OF 9

ROOT CREATED, HOLDING JUST 10

Insert 10. Heap was empty, so a brand-new node is created — this becomes the root. It holds a single element: root = [10].

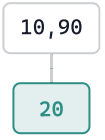
STEP 3 OF 9





Insert 90. Count was 1 (odd), so 90 fills the root's free second slot. 10 < 90, so root becomes [10, 90]. The root has no parent, so there is nothing to bubble against.

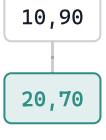
STEP 4 OF 9

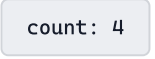


NEW NODE "LEFT" CREATED, HOLDING 20

Insert 20. Count was 2 (even), so a new node is created at the next complete-tree position — root's first child, "left". It holds 20 alone. Check containment against root [10,90]: is 10 ≤ 20 ≤ 90? Yes — fits directly, no bubbling.

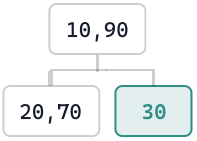
STEP 5 OF 9





Insert 70. Count was 3 (odd), so 70 fills "left"'s free second slot. 20 < 70, so "left" becomes [20, 70]. Check against root: 70 ≤ 90? Yes — fits, no bubbling.

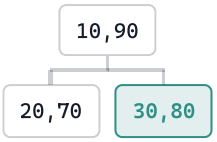
STEP 6 OF 9



NEW NODE "RIGHT" CREATED, HOLDING 30

Insert 30. Count was 4 (even), so a new node is created at the next position — root's second child, "right". Check against root [10,90]: 10 ≤ 30 ≤ 90? Yes — fits directly.

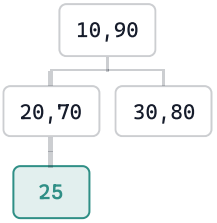
STEP 7 OF 9



count: 6

Insert 80. Count was 5 (odd), fills "right"'s free slot. 30 < 80, so "right" becomes [30, 80]. Check against root: 80 ≤ 90? Yes — fits.

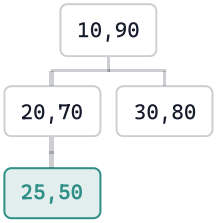
STEP 8 OF 9



NEW NODE "LEFT-LEFT" CREATED, HOLDING 25

Insert 25. Count was 6 (even), new node created at the next position — "left"'s first child, "left-left". Check against its PARENT, "left" = [20,70]: is 20 ≤ 25 ≤ 70? Yes — fits directly.

STEP 9 OF 9



8 elements | 4 nodes | zero bubbles the entire build

Insert 50. Count was 7 (odd), fills "left-left"'s free slot. 25 < 50, so "left-left" becomes [25, 50]. Check against "left" [20,70]: 50 ≤ 70? Yes — fits. Final heap: root [10,90], left [20,70], right [30,80], left-left [25,50] — exactly the heap every later slide starts from, built with zero bubbling.

WHERE EACH NEW NODE GOES

Same shape rule as a binary heap: if the element count is currently **odd**, a half-full node already exists — the new value fills its free second slot. If the count is **even**, every node is full, so a brand-new node is created at the next position in the complete tree (left to right, level by level).

WHY NOTHING BUBBLES HERE

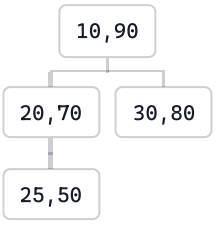
Each pair of values (10&90, then 20&70, then 30&80, then 25&50) was inserted back to back, right after its node was created — and each pair already sits inside its parent's interval the moment it arrives. That is a deliberately convenient order. L20·12 reuses this exact heap and inserts 15, 100, and 65 into it — values that do *not* fit so conveniently, forcing the bubbling you have not seen yet.

Fits directly, bubbles via the min side, bubbles via the max side, or fills an odd slot

Starting heap: root [10,90], left [20,70], right [30,80], left-left [25,50] — 4 nodes, 8 elements (verified valid: every child's interval sits inside its parent's). Each case below starts fresh from this same heap, with a new node A about to be added as left's second child.

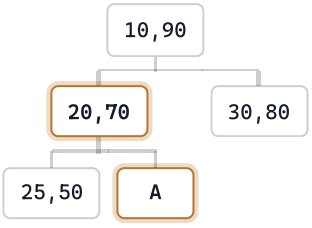
INTERACTIVE — FOUR INDEPENDENT SINGLE-ELEMENT INSERTS, ONE FLAGSHIP CHAIN

STEP 1 OF 11



Starting interval heap: root [10,90], left [20,70], right [30,80], left-left [25,50] — 8 elements, containment verified at every node. A new node A (left's second child) is about to be added.

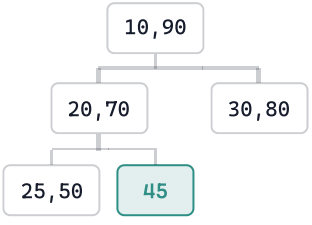
STEP 2 OF 11



CASE 1 — INSERT 45

Case 1 — insert 45. A's parent is "left" = [20,70]. Since 20 ≤ 45 ≤ 70, the new element fits directly into A — no bubbling needed.

STEP 3 OF 11

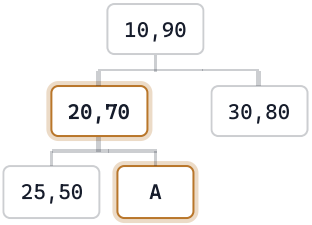


Comparisons: 1

No bubbling — cheapest case

A becomes a single-element node holding 45. Done — the cheapest of the four cases.

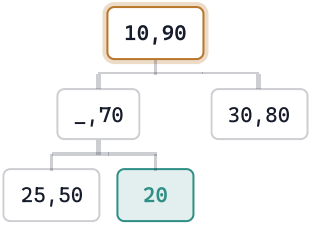
STEP 4 OF 11



CASE 2 — INSERT 15

Case 2 — insert 15. 15 < 20, left's own left endpoint — too small to fit at A directly. We must bubble UP through the embedded min-heap, starting at A's parent.

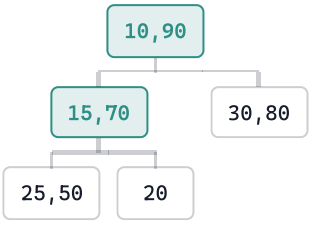
STEP 5 OF 11



LEFT'S LEFT ENDPOINT (20) DEMOTED DOWN TO A; CHECK ROOT NEXT

At "left": is 20 ≤ 15? No. So 20 moves DOWN into A (A now holds 20), and we continue checking upward at the parent — the root, [10,90].

STEP 6 OF 11

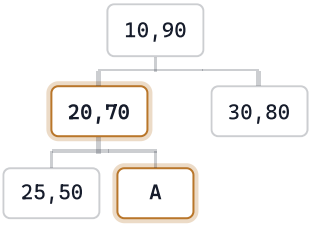


Comparisons: 2

Bubbled up 1 level via the min side

At root: is 10 ≤ 15? Yes — stop here. Root keeps 10 (still the true minimum); 15 settles into "left" as its new left endpoint (replacing the 20 that was demoted). Final: root [10,90], left [15,70], A holds 20.

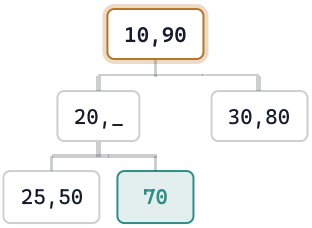
STEP 7 OF 11



CASE 3 — INSERT 100

Case 3 — insert 100. 100 > 70, left's own right endpoint — too large to fit at A directly. We bubble UP through the embedded MAX-heap this time, moving right endpoints down instead of left endpoints.

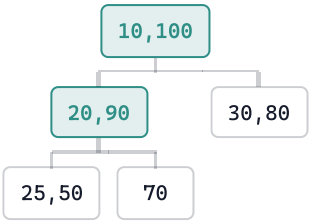
STEP 8 OF 11



LEFT'S RIGHT ENDPOINT (70) DEMOTED DOWN TO A; CHECK ROOT NEXT

At "left": is 70 ≥ 100? No. So 70 moves DOWN into A (A now holds 70), and we continue checking upward at the root, [10,90].

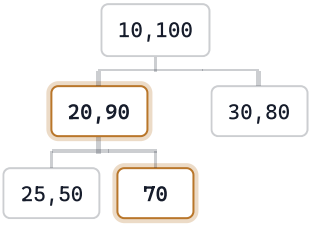
STEP 9 OF 11



Comparisons: 2 | Bubbled up 1 level via the max side

At root: is 90 ≥ 100? No — and root has no parent, so we stop here regardless. 100 becomes the root's new right endpoint; the old 90 is demoted down to "left" as its new right endpoint. Final: root [10,100], left [20,90], A holds 70.

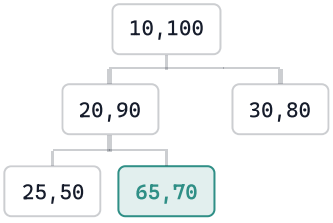
STEP 10 OF 11



CASE 4 – CONTINUING FROM CASE 3: INSERT 65 (HEAP HAS 9 ELEMENTS, ODD)

Case 4 — insert 65, continuing from Case 3's result. The heap now has 9 elements (odd) — inserting does NOT add a new node; A already exists with one free slot. A's parent is "left" = [20,90]. 20 ≤ 65 ≤ 90, so 65 fits directly into A.

STEP 11 OF 11



Comparisons: 1 | A now full: [65,70] | Heap back to 10 elements, even

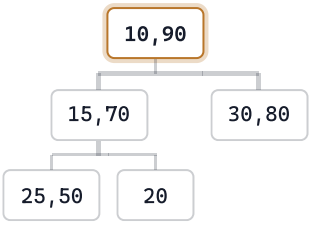
A held a single element, 70. Since 65 < 70, 65 becomes A's new left endpoint and 70 shifts to be its right endpoint: A = [65,70]. No new node was created — the odd slot just filled up.

Take the root's left endpoint, promote from the last node, reinsert down

Continuing from the "bubbles via the min side" case: root [10,90], left [15,70], node A [20] (last node, single element). removeMin() must remove 10 and restore a valid interval heap using only these 9 elements.

INTERACTIVE – REMOVEMIN()

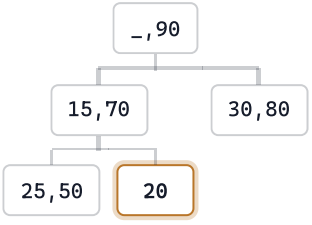
STEP 1 OF 8



STARTING HEAP – 9 ELEMENTS (CASE 2'S RESULT)

removeMin(): the interval heap has more than one element, so we return the root's LEFT endpoint, 10, and remove it from the root.

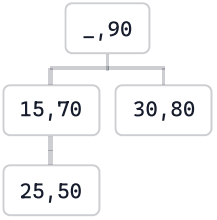
STEP 2 OF 8



ROOT NOW MISSING ITS LEFT ENDPOINT; LAST NODE IS A = [20]

Root now has only its right endpoint (90) — it needs a new left endpoint. The last node in the heap is A, holding single element 20.

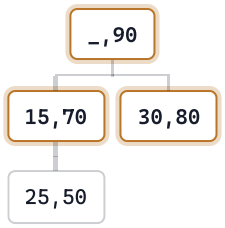
STEP 3 OF 8



A IS NOW EMPTY, SO IT IS NO LONGER PART OF THE TREE – HEAP SHRINKS TO 4 NODES

Remove the point p = 20 from A. A becomes empty, so it is dropped entirely: the tree shrinks back to 4 nodes. p = 20 must now be reinserted into the embedded min-heap, starting at the root.

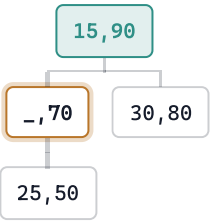
STEP 4 OF 8



COMPARE CHILDREN'S LEFT ENDPOINTS: 15 VS. 30

At the root: compare the left endpoints of its two children — "left"=15 and "right"=30. The smaller is 15. Is 15 < p (20)?

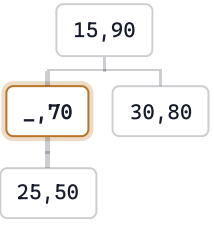
STEP 5 OF 8



P STILL NEEDS A HOME — DESCEND INTO "LEFT"

Yes, 15 < 20 — so 15 is promoted UP into the root (root's left endpoint becomes 15), and we continue the walk DOWN into "left" (where 15 came from), still carrying p = 20 to place.

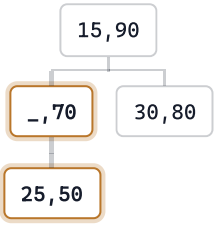
STEP 6 OF 8



CHECK: DOES P (20) NEED TO SWAP WITH THIS NODE'S RIGHT ENDPOINT (70)?

At "left": first check the LOCAL invariant. Is p (20) > this node's right endpoint (70)? No (20 < 70) — no swap needed here.

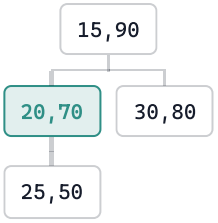
STEP 7 OF 8



"LEFT" HAS ONLY ONE REMAINING CHILD NOW (LEFT-LEFT) — COMPARE ITS LEFT ENDPOINT TO P

Now check "left"'s children. Since A no longer exists, "left" has only one child left: left-left = [25,50]. Compare its left endpoint (25) to p (20): is 25 < 20?

STEP 8 OF 8



Returned: 10 | Reinsertion depth: 1 level | New root: [15,90]

No (25 > 20) — p does not need to descend further. p = 20 settles here, as "left"'s new left endpoint: left = [20,70]. Final heap: root [15,90], left [20,70], right [30,80], left-left [25,50] — 8 elements, containment still holds throughout.

Half the space of a dual structure — for some extra bookkeeping per node

ADVANTAGES

- **Every element stored exactly once** — no correspondence pointers, no duplicated storage, unlike the dual-structure method.
- **O(1) getMin and getMax** — both just read the root.
- **O(log n) put/removeMin/removeMax** — a single tree, a single array, ordinary heap-style bookkeeping.
- Widely regarded as the simplest and most efficient of the heap-based DEPQ adaptations — simpler than min-max heaps, twin heaps, deaps, or diamond deques.

LIMITATIONS

- **Every node carries two elements**, and every insert/remove must maintain the a ≤ b invariant *and* the containment property simultaneously — more bookkeeping per step than a plain single-value heap node.
- **Odd-count edge cases** (a single-element last node) need special-casing throughout insert and remove.
- **Removing an arbitrary interior element** (not just min or max) is possible in O(log n) but is a genuinely fiddlier procedure than an ordinary heap's remove(theNode).

O(1) for reading either end, O(log n) for changing the collection

OPERATION	COST (INTERVAL HEAP)
isEmpty(), size(), getMin(), getMax()	O(1)
put(x), removeMin(), removeMax()	O(log n)
Initializing an n-element interval heap from scratch	O(n)

EXTERNAL SORTING (TODAY'S EXAMPLE)

External quicksort's middle group, kept maximal via a live DEPQ — exactly the interval heap's textbook application.

SLA / QOS MONITORING

A system tracking both "worst current latency" and "best current latency" across live requests needs both ends of a priority order simultaneously, updated in real time.

ORDER STATISTICS / RANGE FILTERING

Interval heaps also answer "which points lie outside [a,b]?" (the complementary range search problem) in O(k) time for k reported points — a nice bonus of the same structure.

A practical heap, and a heap that answers from both ends

- **Pairing heaps reduce everything to MELD** — verified today: INSERT (5 keys), DECREASE-KEY (a real detach-and-remeld), and REMOVE-MIN via the two-pass scheme (5 orphaned subtrees, paired then combined, 3 total melds).
- **Pairing heaps trade a proof for practice** — every bound except DECREASE-KEY is solid; DECREASE-KEY is conjectured O(1) amortized and behaves like it, but the tightest proven bound remains a genuine open research question.
- **A DEPQ needs both ends live** — verified with external quicksort's actual algorithm, which cannot work without simultaneous getMin/getMax.
- **Interval heaps store every element once**, embedding a min-heap (left endpoints) and a max-heap (right endpoints) in one tree — verified today across all 4 insert cases and a full removeMin reinsertion trace.

LECTURE 21 — NEXT

Comparative Study of Heap Structures

Binary, Binomial, Fibonacci, and Pairing heaps, side by side — when to reach for which one, mirroring Lecture 15's tree comparison.

HOMEWORK — BRING TO LECTURE 21

- Build a pairing heap by inserting 20, 4, 15, 8, 30 in that order. Trace REMOVE-MIN using the two-pass scheme, showing every meld.
- On paper, show why melding the orphaned children in a single left-to-right chain (the "bad way") can cost Θ(n) for one remove-min, using a small concrete example.
- Insert the 8 elements 5, 40, 12, 35, 18, 28, 22, 30 into an empty interval heap, two at a time (filling each node's two slots as you go). Show the final tree and verify the containment property at every node.
- In 3–4 sentences: why does a DEPQ built from a plain binary heap need a "correspondence pointer" scheme at all — what breaks if you just keep a separate min-heap and max-heap with no links between them?

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

> *Next: Lecture 21 — Comparative Study of Heap Structures.*
