Heap Sort & Priority Queue Applications

Lecture 16 built the machine. Today we point it at two different jobs: sort an entire array in place with a guarantee no comparison sort beats, and run the queue that decides who gets served next — in a hospital, an OS, or a router.

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

~75 minutes
Session outcome: apply heaps in sorting and scheduling problems

~75 MIN

Today, part by part

From operations to applications 00–04

Same heap, same BUILD-MAX-HEAP and MAX-HEAPIFY from Lecture 16 — two very different jobs.

Why Heap Sort? 04–09

In-place, guaranteed O(n log n) — the two things quicksort and merge sort can't both offer at once.

Heap Sort: advantages & limitations 09–14

What you get for giving up stability and cache locality.

Animation: Heap Sort, start to finish 14–32

Sorting the exact heap we built in Lecture 16 — 9 rounds, every exchange and every heapify shown.

Priority queues: why, and a real-life example 32–38

A hospital emergency room, triaged by urgency — not by who walked in first.

Priority queues: advantages & limitations 38–43

What a heap-backed queue buys you, and what it deliberately gives up.

Animation: the ER triage queue, live 43–55

Five patients, five priorities, arrivals and call-ins interleaved.

More scheduling applications 55–60

OS process scheduling, Dijkstra/Prim, network QoS, A* search.

Complexity, everything together 60–64

Heap sort and priority-queue operations, one table.

Recap & what's next 64–75

Lecture 18: Binomial Heaps.

Same heap, two different jobs

LAST TIME — LECTURE 16

We built the machinery: the shape + heap property, the array trick (PARENT/LEFT/RIGHT), BUILD-MAX-HEAP in a verified O(n), HEAP-INSERT (sift-up), and HEAP-EXTRACT-MAX (sift-down). All three run in O(log n) or better.

TODAY — PUT THE MACHINE TO WORK

- → **Heap Sort:** repeat EXTRACT-MAX exactly n−1 times on the *same array* — no extra memory, and the array sorts itself in place.
- → **Priority queues:** wrap HEAP-INSERT and EXTRACT-MIN/MAX behind a scheduling interface — "always serve the most urgent item next," whatever "urgent" means for your problem.

Guaranteed O(n log n), and not one extra byte of memory

THE GAP IT FILLS

Quicksort is fast in practice but has an $O(n^2)$ worst case on adversarial input. Merge sort guarantees O(n log n) but needs O(n) extra space for the merge step. **Heap Sort guarantees O(n log n) *and* sorts in place** — the one combination neither of the other two offers simultaneously.

THE IDEA, IN ONE LINE

BUILD-MAX-HEAP once (Lecture 16, O(n)). Then repeat: swap the root (the current maximum) with the last element of the still-unsorted region, shrink that region by one, and MAX-HEAPIFY the new root back into place. Do this $n-1$ times and the array is sorted, built entirely from the back forward.

REAL-WORLD RELEVANCE

Anywhere memory is tight and a worst-case guarantee matters more than average-case speed — embedded systems, real-time systems with hard memory budgets, or any library sort that must never degrade to $O(n^2)$ regardless of input — Heap Sort (or an introspective hybrid that falls back to it) is the standard answer.

What guaranteed in-place sorting costs you

ADVANTAGES

- **Guaranteed O(n log n)** in the worst case, every time — no adversarial input degrades it, unlike quicksort.
- **O(1) extra space** — sorts in place, unlike merge sort's O(n) auxiliary array.
- **Builds directly on operations you already trust** — no new machinery beyond BUILD-MAX-HEAP and MAX-HEAPIFY from Lecture 16.

LIMITATIONS

- **Not stable.** Equal-priority elements can be reordered relative to each other — the swap-with-last step has no memory of original relative order.
- **Poor cache locality in practice.** MAX-HEAPIFY jumps between indices i, 2i, 2i+1 — scattered memory access compared to quicksort's mostly-sequential partitioning, so heap sort is often *slower in practice* despite an equal or better asymptotic bound.
- **Not adaptive.** An already-sorted (or nearly-sorted) input gets no speed-up at all — heap sort does the same O(n log n) work regardless of how "easy" the input is, unlike insertion sort or adaptive variants of merge sort.

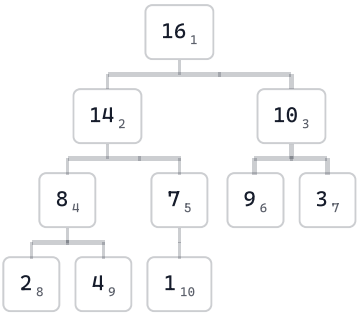
Sorting the exact heap we built last lecture — one extraction at a time

Starting point: [16, 14, 10, 8, 7, 9, 3, 2, 4, 1] — the max-heap Lecture 16 built from [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]. Each of the 9 rounds: exchange the root with the last element of the active heap (teal cells are already **locked into their final sorted position** and never touched again), shrink the heap, then MAX-HEAPIFY the new root. Nothing is skipped — every round is shown, including the cheapest ones.

INTERACTIVE — HEAPSORT(A), 9 ROUNDS

STEP 1 OF 20

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

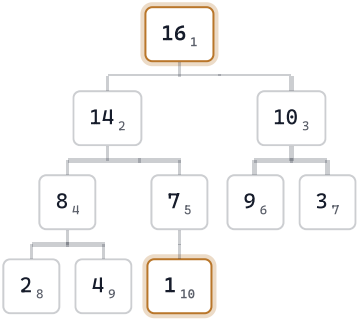


ACTIVE HEAP, SIZE 10 — NOTHING SORTED YET

Start: the max-heap from Lecture 16, A = [16,14,10,8,7,9,3,2,4,1]. HEAPSORT repeats, for i = n downto 2: exchange A[1] with A[i], shrink the heap, then MAX-HEAPIFY(1). Teal cells (none yet) mark positions already locked into their final sorted value.

STEP 2 OF 20

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

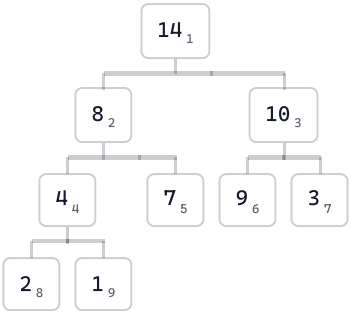


EXCHANGE A[1] AND A[10]

Round 1 of 9. Exchange A[1] (the max, 16) with A[10] (the last element of the active heap, 1).

STEP 3 OF 20

1	2	3	4	5	6	7	8	9	10
14	8	10	4	7	9	3	2	1	16

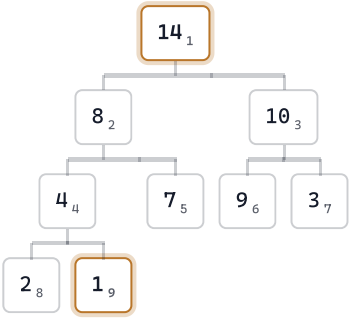


16 LOCKED AT POSITION 10; HEAP SHRINKS TO SIZE 9

16 locks into position 10 — its final sorted place, never touched again. Heap-size shrinks to 9. MAX-HEAPIFY(1) sinks the displaced 1 down through 3 swaps (1↔2, 2↔4, 4↔9) to restore the heap property.

STEP 4 OF 20

1	2	3	4	5	6	7	8	9	10
14	8	10	4	7	9	3	2	1	16

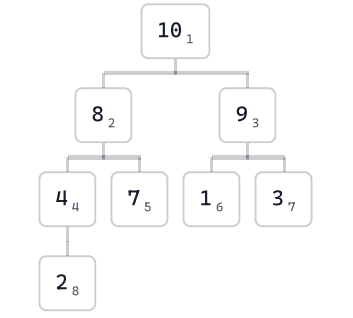


EXCHANGE A[1] AND A[9]

Round 2. Exchange A[1] (14) with A[9] (1), the last element of the 9-element active heap.

STEP 5 OF 20

1	2	3	4	5	6	7	8	9	10
10	8	9	4	7	1	3	2	14	16

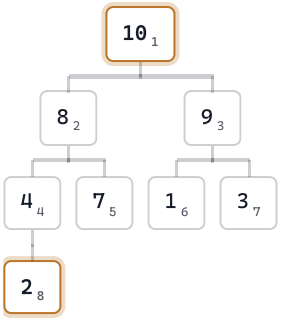


14 LOCKED AT POSITION 9; HEAP SHRINKS TO SIZE 8

14 locks into position 9. Heap-size shrinks to 8. MAX-HEAPIFY(1) sinks the displaced 1 down through 2 swaps (1↔3, 3↔6).

STEP 6 OF 20

1	2	3	4	5	6	7	8	9	10
10	8	9	4	7	1	3	2	14	16

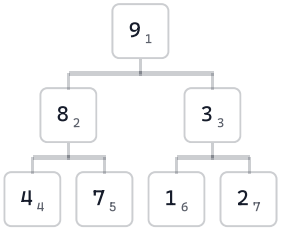


EXCHANGE A[1] AND A[8]

Round 3. Exchange A[1] (10) with A[8] (2), the last element of the 8-element active heap.

STEP 7 OF 20

1	2	3	4	5	6	7	8	9	10
9	8	3	4	7	1	2	10	14	16

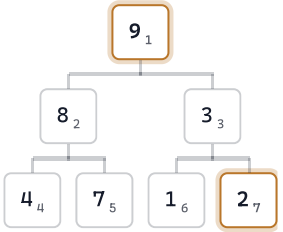


10 LOCKED AT POSITION 8; HEAP SHRINKS TO SIZE 7

10 locks into position 8. Heap-size shrinks to 7. MAX-HEAPIFY(1) sinks the displaced 2 down through 2 swaps (1↔3, 3↔7).

STEP 8 OF 20

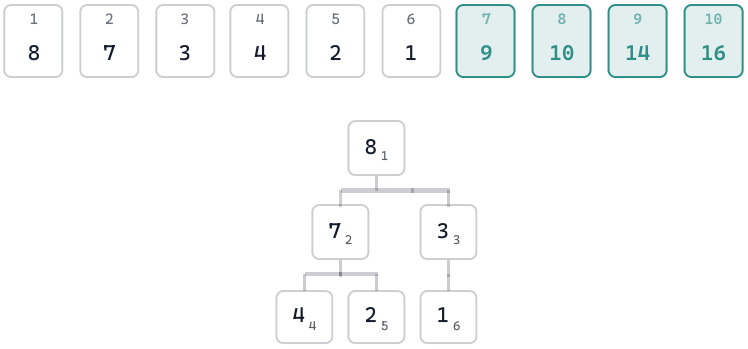
1	2	3	4	5	6	7	8	9	10
9	8	3	4	7	1	2	10	14	16



EXCHANGE A[1] AND A[7]

Round 4. Exchange A[1] (9) with A[7] (2), the last element of the 7-element active heap.

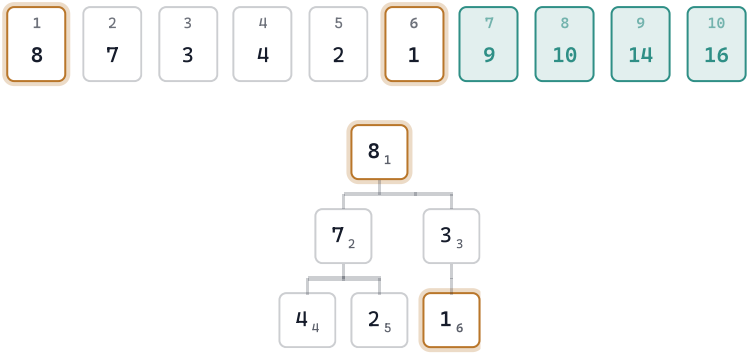
STEP 9 OF 20



9 LOCKED AT POSITION 7; HEAP SHRINKS TO SIZE 6

9 locks into position 7. Heap-size shrinks to 6. MAX-HEAPIFY(1) sinks the displaced 2 down through 2 swaps (1↔2, 2↔5).

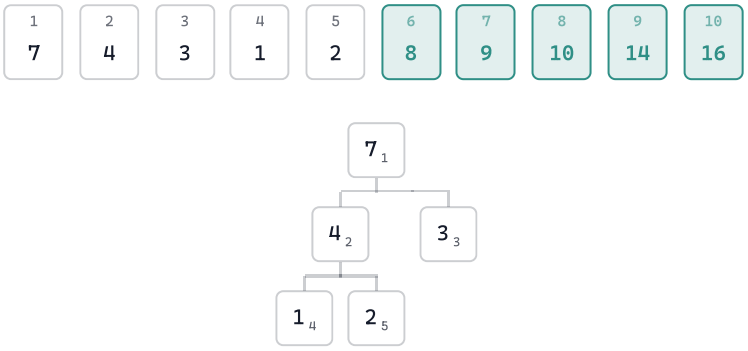
STEP 10 OF 20



EXCHANGE A[1] AND A[6]

Round 5. Exchange A[1] (8) with A[6] (1), the last element of the 6-element active heap.

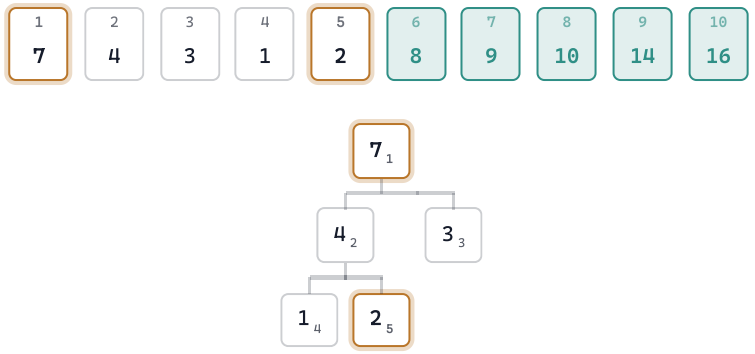
STEP 11 OF 20



8 LOCKED AT POSITION 6; HEAP SHRINKS TO SIZE 5

8 locks into position 6. Heap-size shrinks to 5. MAX-HEAPIFY(1) sinks the displaced 1 down through 2 swaps (1↔2, 2↔4).

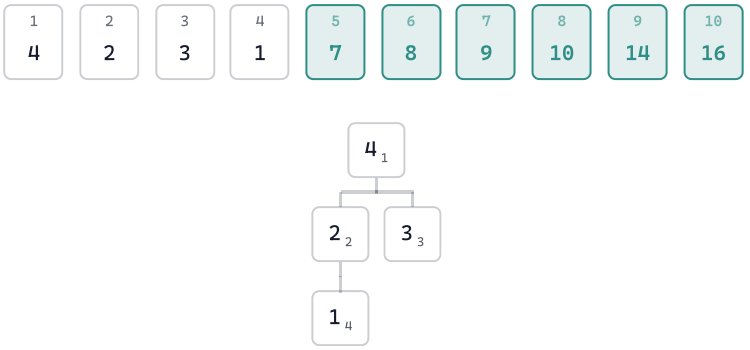
STEP 12 OF 20



EXCHANGE A[1] AND A[5]

Round 6. Exchange A[1] (7) with A[5] (2), the last element of the 5-element active heap.

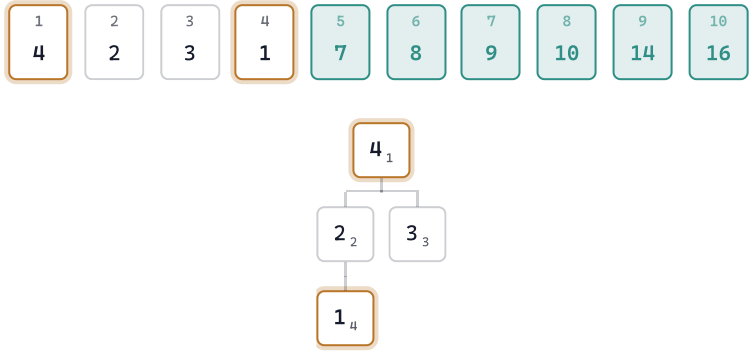
STEP 13 OF 20



7 LOCKED AT POSITION 5; HEAP SHRINKS TO SIZE 4

7 locks into position 5. Heap-size shrinks to 4. MAX-HEAPIFY(1) makes just 1 swap (1↔2) — the recursive check at index 2 finds its only child (1) already smaller, so no further swap is needed. The cheapest round so far.

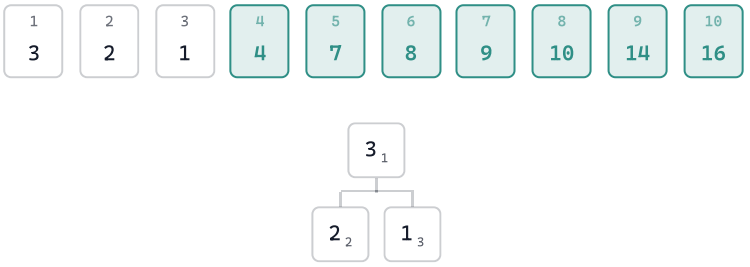
STEP 14 OF 20



EXCHANGE A[1] AND A[4]

Round 7. Exchange A[1] (4) with A[4] (1), the last element of the 4-element active heap.

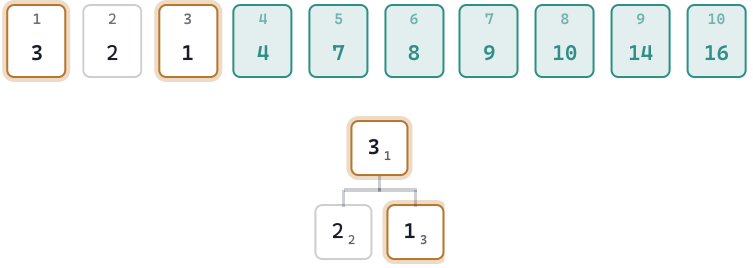
STEP 15 OF 20



4 LOCKED AT POSITION 4; HEAP SHRINKS TO SIZE 3

4 locks into position 4. Heap-size shrinks to 3. MAX-HEAPIFY(1) makes 1 swap (1↔3).

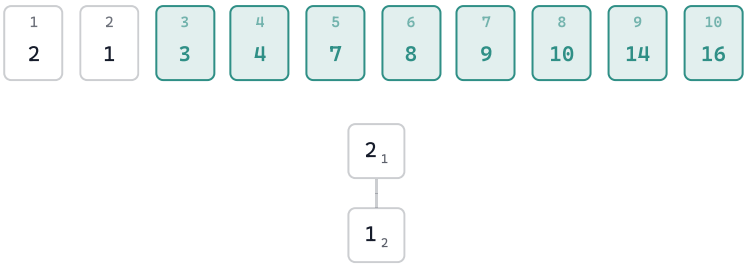
STEP 16 OF 20



EXCHANGE A[1] AND A[3]

Round 8. Exchange A[1] (3) with A[3] (1), the last element of the 3-element active heap.

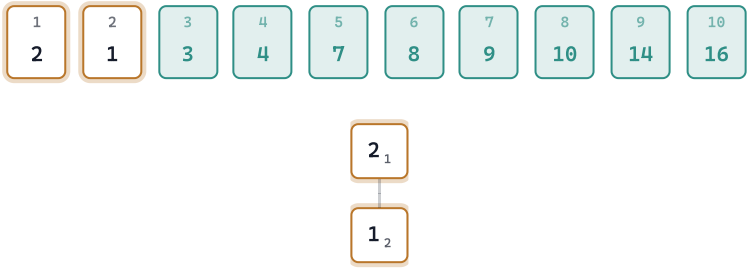
STEP 17 OF 20



3 LOCKED AT POSITION 3; HEAP SHRINKS TO SIZE 2

3 locks into position 3. Heap-size shrinks to 2. MAX-HEAPIFY(1) makes 1 swap (1↔2).

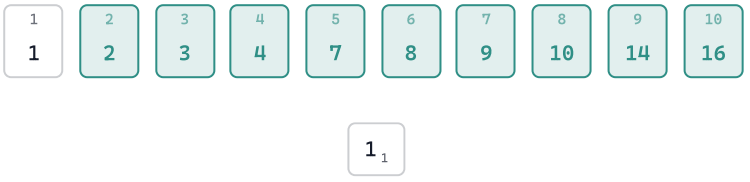
STEP 18 OF 20



EXCHANGE A[1] AND A[2] — FINAL ROUND

Round 9 (final). Exchange A[1] (2) with A[2] (1), the last element of the 2-element active heap.

STEP 19 OF 20



2 LOCKED AT POSITION 2; HEAP SHRINKS TO SIZE 1 — A SINGLE NODE NEEDS NO FURTHER HEAPIFY

2 locks into position 2. Heap-size shrinks to 1 — a single-node heap is trivially valid, so the loop stops here (i would next be 1, which HEAPSORT never processes, since a 1-element region is already sorted).

STEP 20 OF 20

1	2	3	4	5	6	7	8	9	10
1	2	3	4	7	8	9	10	14	16

heap empty — fully sorted

Rounds: 9 | Total swap operations: 23 | Total cost: O(n log n) | Extra space: O(1)

Done. A = [1,2,3,4,7,8,9,10,14,16] — fully sorted in ascending order, in place. 9 rounds, 23 total swap operations (the round's own exchange plus every internal MAX-HEAPIFY swap), for O(n log n) total work using nothing beyond the array itself.

A hospital emergency room doesn't serve first-come, first-served

THE REAL-LIFE EXAMPLE

Walk into an ER and it doesn't matter who arrived first. A triage nurse assigns each patient an urgency level; the **most urgent waiting patient is always seen next**, no matter how long anyone else has been sitting there. A patient arriving in cardiac arrest jumps ahead of five people who arrived hours earlier with a sprained ankle.

THE ABSTRACT VERSION

That's exactly a **priority queue**: an abstract data type supporting INSERT (a new item arrives, with a priority) and EXTRACT-MIN/MAX (serve the most urgent item waiting). It is a *generalization* of a plain queue (FIFO, priority = arrival time) and of a plain stack (LIFO) — a binary heap is simply the standard, efficient way to **implement** it.

WHY A HEAP, SPECIFICALLY

A priority queue needs exactly the two operations a binary heap is built for: insert a new arrival (sift-up, O(log n)) and extract the current extreme (sift-down, O(log n)). No other operation is required — which is precisely why a heap, and not a sorted list or a balanced BST, is the standard choice: it's the simplest structure that does exactly this and nothing more.

Exactly the right abstraction for "serve the most urgent next" — and only that

ADVANTAGES

- **O(log n) arrival and service** — scales to large, constantly-changing queues without ever re-sorting everything.
- **Priority, not arrival order, decides service** — modeling urgency directly, rather than bolting it onto a FIFO queue with manual re-ordering.
- **O(1) peek** at "who's next" without removing them — useful for dashboards/monitoring ("what's the most urgent item right now?").

LIMITATIONS

- **No fairness guarantee.** A steady stream of high-priority arrivals can starve low-priority items indefinitely — a real operational risk in scheduling systems, usually solved by aging (gradually increasing a waiting item's priority over time), which a plain heap does not do automatically.
- **Changing an item's priority is expensive** without extra bookkeeping (Lecture 16's decrease-key limitation) — if a waiting patient's condition worsens, updating their position needs an auxiliary index, not just a heap.
- **No efficient "list everyone waiting, in order"** — that requires repeated extraction (effectively a heap sort), not a cheap traversal.

Five patients, five priorities, arrivals and call-ins interleaved

Triage levels 1 (most critical) through 5 (least urgent) — a **min-priority queue**, since the lowest number is the most urgent. Watch what happens when a critical patient arrives *after* two others have already been waiting.

INTERACTIVE — ER TRIAGE QUEUE: ARRIVALS (INSERT) AND CALL-INS (EXTRACT-MIN)

STEP 1 OF 19

1. A arrives 3 · 2. B arrives 1 · 3. C arrives 4 · 4. call next · 5. D arrives 2 · 6. call next · 7. E arrives 1 · 8. call next · 9. call next · 10. call next

empty

EMPTY QUEUE

An ER triage queue is a **min-priority queue** on urgency level (1 = critical, 5 = minor). The row above is the full schedule of events, so you can always see what is coming next. Each INSERT is shown in two halves — **place at the end**, then **sift up** — and each EXTRACT-MIN likewise: **take the root and refill it**, then **sift down**.

STEP 2 OF 19

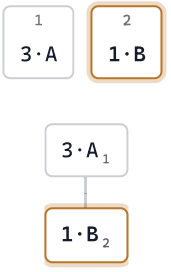
1. A arrives 3 · 2. B arrives 1 · 3. C arrives 4 · 4. call next · 5. D arrives 2 · 6. call next · 7. E arrives 1 · 8. call next · 9. call next · 10. call next

PLACED AT THE END — THE ONLY NODE, SO NOTHING TO COMPARE AGAINST

1 — A arrives, level 3. INSERT always puts the new patient at the **end of the array** (here, position 1). It is the root as well as the only node, so there is no parent to compare with and no sift-up to do.

STEP 3 OF 19

1. A arrives 3 · 2. B arrives 1 · 3. C arrives 4 · 4. call next · 5. D arrives 2 · 6. call next · 7. E arrives 1 · 8. call next · 9. call next · 10. call next

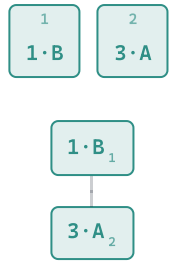


PLACED AT POSITION 2 — HEAP PROPERTY NOW VIOLATED

2 — B arrives, level 1. First half of INSERT: B goes at the end, position 2. Compare with its parent at position ⌊2/2⌋ = 1: parent A is level 3, B is level 1. In a *min*-heap the smaller number must sit above — so this is a violation and B must sift up.

STEP 4 OF 19

1. A arrives 3 | **2. B arrives 1** | 3. C arrives 4 | 4. call next | 5. D arrives 2 | 6. call next | 7. E arrives 1 | 8. call next | 9. call next | 10. call next

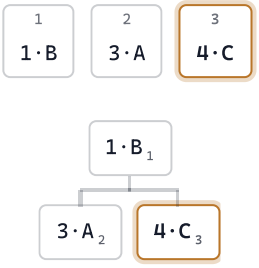


SIFT-UP: B SWAPPED WITH A, NOW AT THE ROOT

2 — sift up. Swap B with its parent. B is now at position 1 and has no parent left, so the sift stops. B is the front of the queue.

STEP 5 OF 19

1. A arrives 3 | 2. B arrives 1 | **3. C arrives 4** | 4. call next | 5. D arrives 2 | 6. call next | 7. E arrives 1 | 8. call next | 9. call next | 10. call next

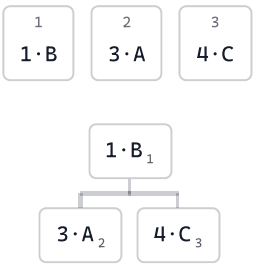


PLACED AT POSITION 3 — PARENT IS POSITION 1

3 — C arrives, level 4. C goes at the end, position 3. Its parent is position ⌊3/2⌋ = 1, holding B at level 1. C (4) is *larger* than its parent, so the min-heap property already holds.

STEP 6 OF 19

1. A arrives 3 | 2. B arrives 1 | **3. C arrives 4** | 4. call next | 5. D arrives 2 | 6. call next | 7. E arrives 1 | 8. call next | 9. call next | 10. call next

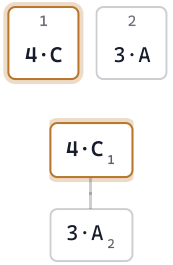


NO SWAP NEEDED — C SETTLES WHERE IT LANDED

3 — sift up: nothing to do. Zero swaps. This is the cheap case, and it is why INSERT is O(log n) in the *worst* case but often far less. B is still the most urgent.

STEP 7 OF 19

1. A arrives 3 | 2. B arrives 1 | 3. C arrives 4 | **4. call next** | 5. D arrives 2 | 6. call next | 7. E arrives 1 | 8. call next | 9. call next | 10. call next



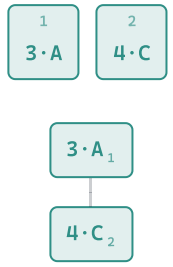
B REMOVED; THE LAST ELEMENT (C) MOVED INTO THE ROOT

Served so far: B (1)

4 — ER calls the next patient. EXTRACT-MIN returns the root, **B** (level 1). First half: the root is now empty, so the **last** element in the array — C — is moved up to fill it. That keeps the tree complete, but almost always breaks the heap property.

STEP 8 OF 19

1. A arrives 3 | 2. B arrives 1 | 3. C arrives 4 | **4. call next** | 5. D arrives 2 | 6. call next | 7. E arrives 1 | 8. call next | 9. call next | 10. call next



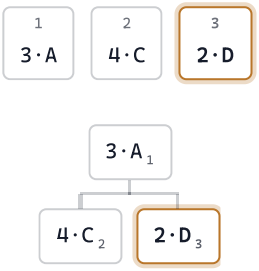
SIFT-DOWN: C SWAPPED WITH ITS SMALLER CHILD A

Served so far: B (1)

4 — sift down. Compare the new root C (4) with its children: only position 2 exists, holding A (3). A is smaller, so swap. C reaches the bottom and stops. A is the new front.

STEP 9 OF 19

1. A arrives 3 | 2. B arrives 1 | 3. C arrives 4 | 4. call next | **5. D arrives 2** | 6. call next | 7. E arrives 1 | 8. call next | 9. call next | 10. call next

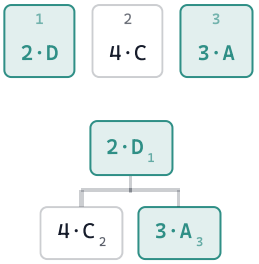


PLACED AT POSITION 3 — PARENT IS POSITION 1 (A, LEVEL 3)

5 — D arrives, level 2. D goes at the end, position 3. Parent is position 1, holding A at level 3. D (2) is more urgent than A (3) — violation, so D sifts up.

STEP 10 OF 19

1. A arrives 3 | 2. B arrives 1 | 3. C arrives 4 | 4. call next | **5. D arrives 2** | 6. call next | 7. E arrives 1 | 8. call next | 9. call next | 10. call next



SIFT-UP: D SWAPPED PAST A, STRAIGHT TO THE ROOT

5 — sift up. Swap D with A. D lands at the root and stops. Note it overtook *both* A and C, even though both had been waiting longer.

STEP 11 OF 19

1. A arrives 3 | 2. B arrives 1 | 3. C arrives 4 | 4. call next | 5. D arrives 2 | **6. call next** | 7. E arrives 1 | 8. call next | 9. call next | 10. call next

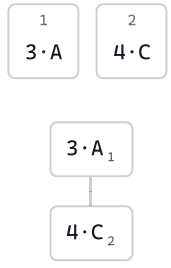
D REMOVED; LAST ELEMENT (A) MOVED INTO THE ROOT

Served so far: **B (1) → D (2)**

6 — ER calls the next patient. EXTRACT-MIN returns **D** (level 2) — *not* A, who has been waiting since the very first arrival. The last element, A, moves up into the empty root.

STEP 12 OF 19

1. A arrives 3 | 2. B arrives 1 | 3. C arrives 4 | 4. call next | 5. D arrives 2 | **6. call next** | 7. E arrives 1 | 8. call next | 9. call next | 10. call next



SIFT-DOWN: A (3) IS ALREADY SMALLER THAN ITS CHILD C (4) — NO SWAP

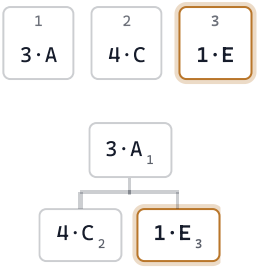
Served so far: **B (1) → D (2)**

6 — sift down: nothing to do. The new root A (3) is compared with its only child C (4). A is already the smaller, so the heap property holds and the sift stops immediately.

STEP 13 OF 19

1. A arrives 3 | 2. B arrives 1 | 3. C arrives 4 | 4. call next | 5. D arrives 2 | 6. call next | **7. E arrives 1**

8. call next | 9. call next | 10. call next



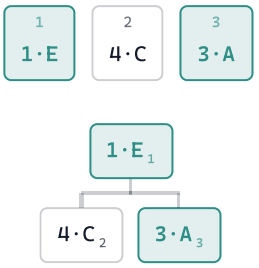
PLACED AT POSITION 3 — PARENT IS A (LEVEL 3)

7 — E arrives, level 1 (critical). E goes at the end, position 3. Parent at position 1 is A (3). E (1) is far more urgent — violation, sift up.

STEP 14 OF 19

1. A arrives 3 | 2. B arrives 1 | 3. C arrives 4 | 4. call next | 5. D arrives 2 | 6. call next | **7. E arrives 1**

8. call next | 9. call next | 10. call next



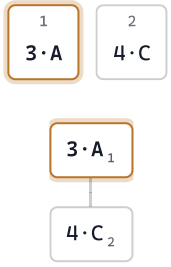
SIFT-UP: E SWAPPED TO THE ROOT ON ARRIVAL

7 — sift up. Swap E with A; E reaches the root and stops. A critical patient jumps the whole queue the moment they walk in — ahead of A (waiting since event 1) and C (since event 3).

STEP 15 OF 19

1. A arrives 3 | 2. B arrives 1 | 3. C arrives 4 | 4. call next | 5. D arrives 2 | 6. call next | 7. E arrives 1

8. call next | 9. call next | 10. call next



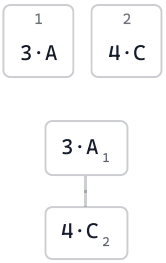
E REMOVED; LAST ELEMENT (A) MOVED INTO THE ROOT

Served so far: B (1) → D (2) → E (1)

8 — ER calls the next patient. EXTRACT-MIN returns **E** (level 1) — served almost immediately after arriving. A moves up into the root.

STEP 16 OF 19

1. A arrives 3 | 2. B arrives 1 | 3. C arrives 4 | 4. call next | 5. D arrives 2 | 6. call next | 7. E arrives 1 | **8. call next** | 9. call next | 10. call next



SIFT-DOWN: A (3) < C (4) — NO SWAP

Served so far: B (1) → D (2) → E (1)

8 — sift down: nothing to do. This is the defining behaviour of a priority queue: it is **not** first-come-first-served. E arrived 4th and was served 3rd.

STEP 17 OF 19

1. A arrives 3 | 2. B arrives 1 | 3. C arrives 4 | 4. call next | 5. D arrives 2 | 6. call next | 7. E arrives 1 | 8. call next | **9. call next** | 10. call next

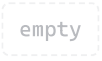
A REMOVED; LAST ELEMENT (C) MOVED INTO THE ROOT — NO CHILDREN LEFT

Served so far: B (1) → D (2) → E (1) → A (3)

9 — ER calls the next patient. EXTRACT-MIN returns **A** (level 3), finally served after waiting through four other events. C moves into the root and has no children, so there is no sift-down to do.

STEP 18 OF 19

1. A arrives 3 | 2. B arrives 1 | 3. C arrives 4 | 4. call next | 5. D arrives 2 | 6. call next | 7. E arrives 1 | 8. call next | 9. call next | **10. call next**



QUEUE EMPTY

Served so far: B (1) → D (2) → E (1) → A (3) → C (4)

10 — ER calls the last patient. EXTRACT-MIN returns **C** (level 4). Nothing remains to move up, and the queue is empty.

STEP 19 OF 19

1. A arrives 3 | 2. B arrives 1 | 3. C arrives 4 | 4. call next | 5. D arrives 2 | 6. call next | 7. E arrives 1 | 8. call next | 9. call next | 10. call next

Final service order: **B** (1) → **D** (2) → **E** (1) → **A** (3) → **C** (4)

Arrival order: A, B, C, D, E

Service order: B, D, E, A, C

Every INSERT/EXTRACT-MIN: O(log n)

Done. E was served 3rd despite arriving 4th (after A and C) — its priority (1) beat theirs (3 and 4) outright. Every arrival was a *place-then-sift-up* and every call-in a *take-root-then-sift-down* — exactly the O(log n) operations from Lecture 16. A priority queue is a binary heap wearing a scheduling hat.

The same pattern, well beyond hospitals

OS PROCESS SCHEDULING

Operating systems keep a ready-queue of processes keyed by priority (and often dynamically adjusted to prevent starvation) — "run the highest-priority ready process next" is the priority-queue problem verbatim.

DIJKSTRA & PRIM (LECTURE 16 CALLBACK)

Both algorithms repeatedly pick the cheapest available vertex/edge — implemented with exactly the min-priority queue we just animated, typically a binary heap, upgraded to a Fibonacci heap (Lecture 19) when decrease-key dominates the running time.

NETWORK PACKET SCHEDULING (QOS)

Routers prioritize latency-sensitive traffic (voice/video) over bulk transfers using priority queues on packets — the same "urgent jumps the line" behavior as the ER example.

A* SEARCH

Pathfinding's "open list" is a min-priority queue ordered by estimated total cost — always expand the most promising node next, exactly like always calling the most urgent patient next.

PRINTER / JOB QUEUES

Print spoolers and batch job systems often support priority levels (e.g. a rush job) — small jobs or urgent flags jump ahead of a long queue of routine ones.

EVENT-DRIVEN SIMULATION

Discrete-event simulators keep a priority queue of future events ordered by timestamp — "process the next event in time order" is a priority queue keyed on time instead of urgency.

Heap Sort and priority-queue operations, one table

OPERATION	COST	VERIFIED TODAY AS
BUILD-MAX-HEAP (once, at the start)	O(n)	from Lecture 16 — reused as-is
HEAPSORT total (n−1 rounds of exchange + MAX-HEAPIFY)	O(n log n)	9 rounds, 23 total swap operations on 10 elements
HEAPSORT extra space	O(1) — sorts in place	only the array itself, plus O(1) temporaries for each swap
Priority queue INSERT	O(log n)	each ER arrival — at most 1–2 sift-up comparisons in a 3-patient queue
Priority queue EXTRACT-MIN/MAX	O(log n)	each ER call-in — at most 1 sift-down comparison in a 3-patient queue
Priority queue PEEK	O(1)	"who's next" without removing them — always the root

~10 MIN

One heap, two jobs — sort everything, or serve the most urgent

- **Heap Sort turns the same heap outward-in:** repeated EXTRACT-MAX, each result parked at the shrinking heap's boundary — verified today on Lecture 16's own array, ending at the fully sorted [1,2,3,4,7,8,9,10,14,16] in 9 rounds and 23 total swaps.
- **Heap Sort's trade-off:** the only comparison sort that is simultaneously guaranteed O(n log n) *and* in-place — at the cost of stability, cache locality, and adaptiveness.
- **A priority queue is an ADT, a heap is its implementation:** INSERT and EXTRACT-MIN/MAX are all it needs, and a binary heap provides both in O(log n).
- **Priority, not arrival order, decides service** — verified in the ER simulation, where a patient arriving fifth (E, priority 1) was served *before* patients who had been waiting since the first and third arrivals (A, priority 3, and C, priority 4).
- **The trade-off:** no fairness/starvation guarantee and no cheap decrease-key without extra bookkeeping — problems Lectures 18–19's Binomial and Fibonacci heaps address directly.

LECTURE 18 — NEXT

Binomial Heaps

What a binary heap can't do at all: merge two heaps together efficiently. Binomial heaps are built specifically to fix that.

HOMEWORK — BRING TO LECTURE 18

- Run HEAPSORT by hand on the max-heap you built for last lecture's homework array. Show the array after every round and count total swaps.
- Design a min-priority-queue trace for a print-spooler scenario: 4 jobs arrive with page counts as priority (fewer pages = more urgent), with at least one job arriving mid-queue that should jump ahead of an earlier one. Show the queue after each event.
- In 3–4 sentences: explain why HEAPSORT is not a *stable* sort, using a concrete example with two equal-priority elements that could end up reordered.
- In 3–4 sentences: what does "starvation" mean for a priority queue, and sketch (in words, no code needed) how "aging" — slowly increasing a waiting item's priority over time — could prevent it.

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

> *Next: Lecture 18 — Binomial Heaps.*
