Binary Heaps & Heap Operations

Lecture 15 asked "which tree for which job?" Here's a job none of those trees are built for: repeatedly hand over the largest (or smallest) item in a changing collection — fast, and without ever fully sorting anything.

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

~70 minutes
Session outcome: implement heap insertion, deletion, and heapify operations

~70 MIN

Today, part by part

- **Why binary heaps?** 00–05
  The priority-queue problem, and why sorted arrays, unsorted arrays, and balanced BSTs all fall short.
- **Shape property + heap property** 05–10
  A complete binary tree stored in a plain array — no pointers, ever.
- **Advantages and limitations** 10–15
  O(log n) insert/extract with nothing but an array — at the cost of no arbitrary search and no cheap merge.
- **The two algorithms: MAX-HEAPIFY and BUILD-MAX-HEAP** 15–21
  Both in full, plus why the loop starts at ⌊n/2⌋ and counts downwards.
- **Animation: BUILD-MAX-HEAP, bottom-up** 21–36
  10 values, 5 heapify calls, every comparison and every swap — including the ones that change nothing.
- **Why BUILD-MAX-HEAP is O(n), not O(n log n)** 30–34
  The tight analysis, in three lines.
- **Animation: HEAP-INSERT (sift-up)** 34–44
  One insertion that bubbles all the way to the root, one that stops immediately.
- **Animation: HEAP-EXTRACT-MAX (sift-down)** 44–53
  Pull the root, promote the last leaf, and watch it sink back into place.
- **Complexity, all operations** 53–57
  One table, every cost we just verified by hand.
- **Where heaps actually run** 57–62
  Huffman coding, external-sort run generation, and a look ahead to Dijkstra/Prim.
- **Recap & what's next** 62–70
  Lecture 17: Heap Sort and Priority Queue Applications.

The priority-queue problem: always hand over the extreme element, fast

THE PROBLEM

A collection keeps changing — items are added, and repeatedly you need to pull out the **current maximum** (or minimum), never a specific key, never the full sorted order. Job schedulers, event simulators, and Lecture 6's Huffman-code builder all have exactly this shape: "give me the most/least urgent item right now."

WHY THE OBVIOUS ANSWERS FALL SHORT

- **Unsorted array:** insert is O(1), but finding the max means scanning everything — O(n) every single extraction.
- **Sorted array:** extracting the max is O(1) (just take the end), but inserting means shifting elements to keep it sorted — O(n) every insertion.
- **Balanced BST** (Lecture 7–9): O(log n) for both — but it's a much bigger hammer than this job needs: pointers, rotations, and full ordering support you never asked for, just to repeatedly find an extreme value.

THE HEAP'S ANSWER

A binary heap gives O(log n) insert *and* O(log n) extract-max, matching the balanced BST — but needs nothing more than a plain array, no pointers, no rotation cases to get wrong, and can be **built from an unsorted array in O(n)**, faster than inserting n elements one at a time.

A complete binary tree, obeying one ordering rule, stored as a plain array

TWO PROPERTIES, TOGETHER

- **Shape property:** the tree is a *complete* binary tree — every level is completely filled except possibly the last, which fills left to right with no gaps.
- **Max-heap property:** for every node i (except the root), A[PARENT(i)] ≥ A[i]. The maximum is always at the root.
- **Min-heap property** (the mirror image): A[PARENT(i)] ≤ A[i] everywhere — the minimum is always at the root. Everything today works identically for min-heaps; we'll build max-heaps throughout.

THE ARRAY TRICK — NO POINTERS, EVER

Because the shape is always a *complete* tree, position alone determines structure. Store the tree in an array A, 1-indexed, level by level, left to right:

$$\text{PARENT}(i) = \lfloor i/2 \rfloor \quad \text{LEFT}(i) = 2i \quad \text{RIGHT}(i) = 2i+1$$

Every animation today shows the array and the tree **side by side** — they are the same object, just two views of it.

Exactly the right tool for "give me the extreme one" — and nothing more

ADVANTAGES

- **O(1) peek** at the max (or min) — it's always the root.
- **O(log n) insert** and **O(log n) extract-max** — matching a balanced BST with far simpler code.
- **O(n) build** from an arbitrary array — a genuinely surprising tight bound we'll prove today, not the loose O(n log n) you'd guess from "n calls, each O(log n)."
- **Pure array storage** — no pointers, no per-node overhead, excellent cache locality.

LIMITATIONS

- **No efficient arbitrary search.** The heap property only relates a node to its parent — it says nothing about left vs. right siblings or across subtrees. Finding "is key 7 in this heap?" is O(n), not O(log n).
- **No efficient decrease-key/increase-key** without an auxiliary index map to locate a given key inside the array first. This exact limitation is what motivates **Fibonacci heaps** (Lecture 19) for algorithms like Dijkstra's shortest path that decrease-key constantly.
- **No efficient merge** of two heaps — combining two binary heaps means an O(n) rebuild. This motivates **Binomial** and **Pairing heaps** (Lectures 18 and 20), built specifically to merge quickly.
- **Heap order ≠ sorted order.** Reading the array top to bottom does not give you sorted output — that takes repeated extraction, which is exactly what Lecture 17's Heap Sort does.

One helper does all the work; the builder just calls it n/2 times

Everything in the next animation is these two procedures. Arrays are **1-indexed**, so for node `i` the children are `2i` and `2i+1` and the parent is ⌊`i/2`⌋ — no pointers anywhere, just arithmetic on indices.

MAX-HEAPIFY(A, I) — PUSH ONE BAD VALUE DOWN

```
MAX-HEAPIFY(A, i, n):
    l = 2i          // left child
    r = 2i + 1      // right child
    largest = i

    if l ≤ n and A[l] > A[largest]:
        largest = l
    if r ≤ n and A[r] > A[largest]:
        largest = r

    if largest ≠ i:
        swap A[i] ⇔ A[largest]
        MAX-HEAPIFY(A, largest, n)
```

The precondition matters: the subtrees rooted at `2i` and `2i+1` must *already* be valid heaps. Only `A[i]` may be out of place. The job is to float that one value down to where it belongs — and the final recursive call is exactly that: having swapped it one level down, follow it and check again.

BUILD-MAX-HEAP(A) — FIX THE WHOLE ARRAY

```
BUILD-MAX-HEAP(A, n):
    for i = ⌊n/2⌋ downto 1:
        MAX-HEAPIFY(A, i, n)
```

That's the entire algorithm — three lines. Two questions it immediately raises, and both are answered by the precondition above:

- **Why start at ⌊n/2⌋?** Every index past ⌊n/2⌋ is a **leaf**, and a lone node is already a valid heap. For n = 10 that means indices 6…10 are skipped outright — half the array, for free.
- **Why count *downwards*?** MAX-HEAPIFY(A, i) is only allowed to run once i's children are already heaps. Going bottom-up guarantees exactly that: by the time you reach i, everything beneath it has been fixed.

TRACE ONE CALL BY HAND BEFORE WATCHING IT

Take the array from the next slide, `A = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]`, and run the very first call, **MAX-HEAPIFY(A, 5)**:

- `i = 5`, so `l = 10`, `r = 11`. A[5] = 16.
- l = 10 ≤ 10 and A[10] = 7 > 16? **No** — largest stays 5.
- r = 11 ≤ 10? **No**, past the end — skipped.

→ largest = i, so **no swap, no recursion.** 16 was already bigger than its only child.

That "nothing happened" case is a real step in the animation, not a skipped one — it is exactly what the `if largest ≠ i` test is there to detect.

WHAT EACH ONE COSTS

MAX-HEAPIFY follows one root-to-leaf path in the worst case, doing a constant amount of work per level → **O(log n)**.

BUILD-MAX-HEAP makes n/2 such calls, so the obvious bound is O(n log n) — but that is **loose**. Most of those calls are on short subtrees near the bottom, and the true total is **O(n)**. L16·05 does that summation properly; for now, just notice while watching that the early calls barely move anything, and only the last few travel far.

~15 MIN

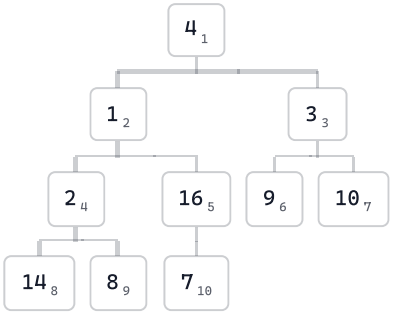
Turn an arbitrary array into a max-heap, bottom-up

Classic array A = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7] (10 elements). BUILD-MAX-HEAP calls MAX-HEAPIFY on every node from index ⌊n/2⌋=5 down to index 1 — skipping the leaves entirely, since a single node is trivially a valid heap. Every comparison, every swap, and every "no change needed" moment is shown, in both the array and the tree view.

INTERACTIVE — BUILD-MAX-HEAP(A), I = 5 DOWN TO 1

STEP 1 OF 17

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

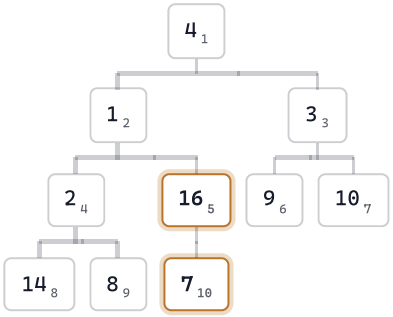


NOT YET A VALID HEAP — E.G. INDEX 4 (2) IS SMALLER THAN ITS CHILD INDEX 8 (14)

Start: A = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7], stored 1-indexed. BUILD-MAX-HEAP calls MAX-HEAPIFY on every index from ⌊10/2⌋=5 down to 1 — indices 6 through 10 are leaves and are already trivially valid heaps of size 1.

STEP 2 OF 17

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

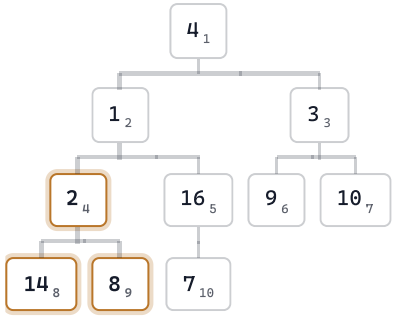


I = 5: COMPARE A[5]=16 WITH ITS ONLY CHILD A[10]=7

i = 5 (value 16). Its only child is index 10 (value 7). 16 > 7 — the max-heap property already holds here. **No swap needed** — MAX-HEAPIFY(5) does nothing and returns immediately.

STEP 3 OF 17

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

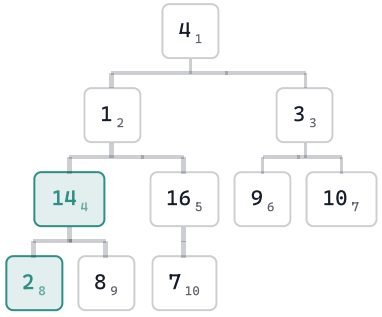


I = 4: COMPARE A[4]=2 WITH CHILDREN A[8]=14, A[9]=8

i = 4 (value 2). Children: index 8 (14) and index 9 (8). Largest of the three is index 8 (14) — 2 is not the largest, so a swap is required.

STEP 4 OF 17

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

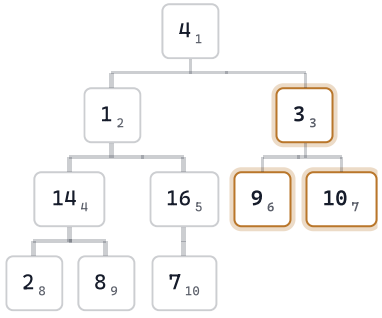


SWAP A[4]<->A[8]; RECURSE INTO INDEX 8 — A LEAF, ALREADY SETTLED

Swap A[4] and A[8]: 14 moves up, 2 moves down. Recurse into MAX-HEAPIFY(8) — index 8 is a leaf (no children within range), so it settles immediately. **1 swap this step.**

STEP 5 OF 17

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

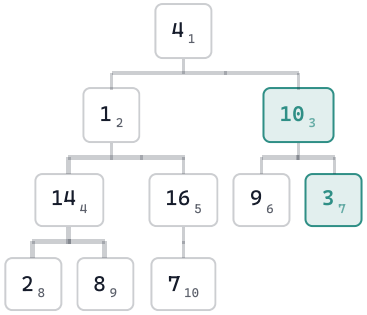


I = 3: COMPARE A[3]=3 WITH CHILDREN A[6]=9, A[7]=10

i = 3 (value 3). Children: index 6 (9) and index 7 (10). Largest is index 7 (10) — swap required.

STEP 6 OF 17

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

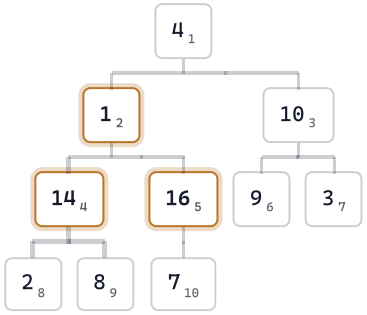


SWAP A[3]<->A[7]; RECURSE INTO INDEX 7 — A LEAF, ALREADY SETTLED

Swap A[3] and A[7]: 10 moves up, 3 moves down. Recurse into MAX-HEAPIFY(7) — a leaf, settles immediately. **1 swap this step.**

STEP 7 OF 17

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

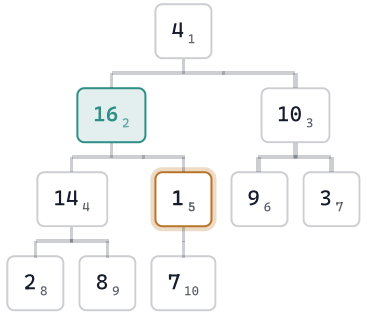


I = 2: COMPARE A[2]=1 WITH CHILDREN A[4]=14, A[5]=16

i = 2 (value 1). Children: index 4 (14) and index 5 (16). Largest is index 5 (16) — swap required.

STEP 8 OF 17

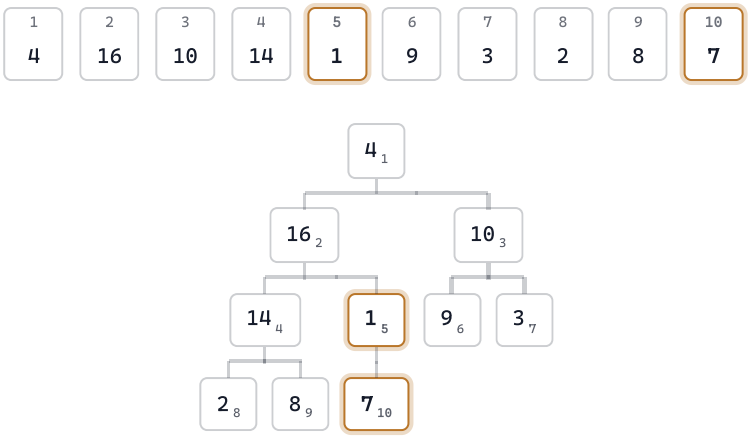
1	2	3	4	5	6	7	8	9	10
4	16	10	14	1	9	3	2	8	7



SWAP A[2]<->A[5]; RECURSE INTO INDEX 5, WHICH NOW HOLDS THE DISPLACED 1

Swap A[2] and A[5]: 16 moves up, 1 moves down to index 5. But index 5 might now violate the property against *its* children — recurse into MAX-HEAPIFY(5).

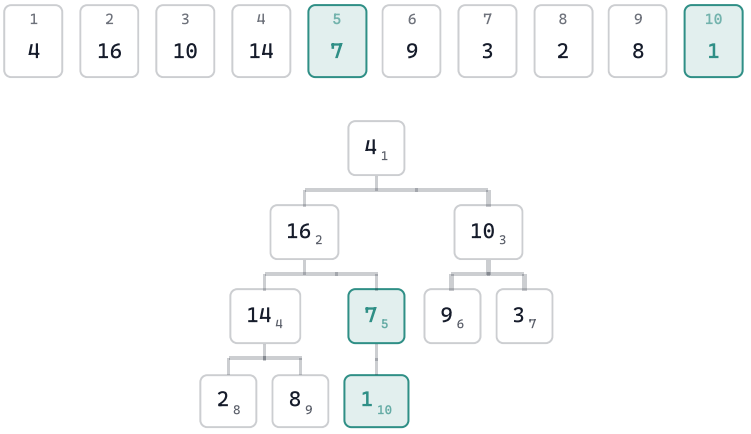
STEP 9 OF 17



COMPARE A[5]=1 WITH ITS ONLY CHILD A[10]=7

Within the recursion: index 5 (value 1) has one child, index 10 (value 7). 7 > 1 — swap required.

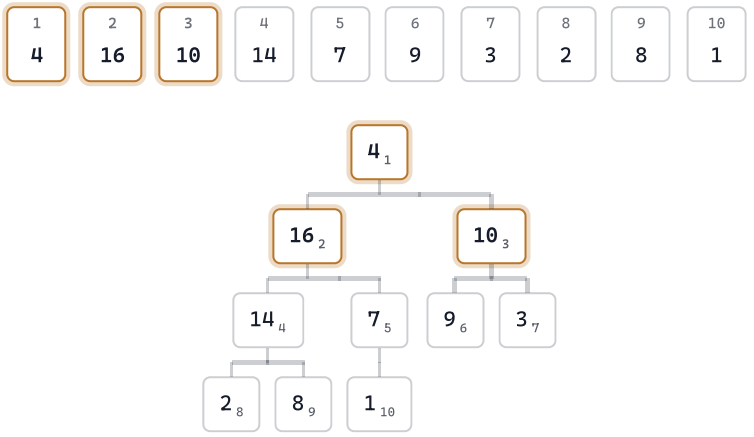
STEP 10 OF 17



SWAP A[5]<->A[10]; INDEX 10 IS A LEAF — SETTLED

Swap A[5] and A[10]: 7 moves up, 1 moves down to a leaf position — settled immediately. **i = 2 total: 2 swaps**, cascading two levels down.

STEP 11 OF 17

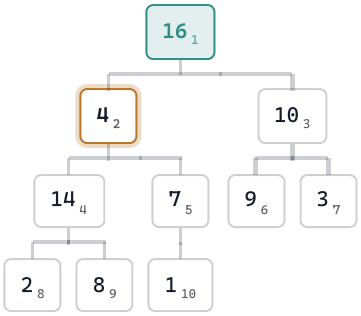


I = 1: COMPARE A[1]=4 WITH CHILDREN A[2]=16, A[3]=10

i = 1 — the root (value 4). Children: index 2 (16) and index 3 (10). Largest is index 2 (16) — swap required.

STEP 12 OF 17

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

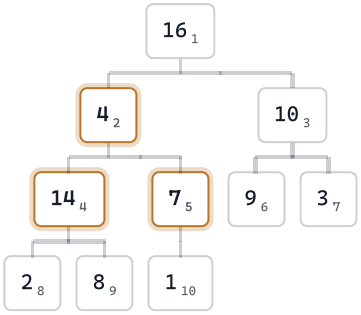


SWAP A[1]<->A[2]; RECURSE INTO INDEX 2, WHICH NOW HOLDS THE DISPLACED 4

Swap A[1] and A[2]: 16 moves to the root, 4 moves down to index 2. Recurse into MAX-HEAPIFY(2) to check whether 4 now violates the property against its own children.

STEP 13 OF 17

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

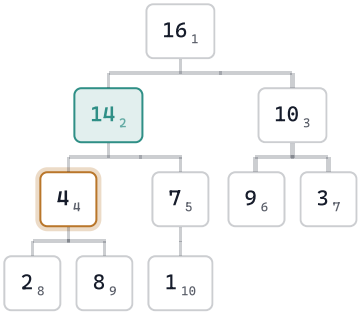


COMPARE A[2]=4 WITH CHILDREN A[4]=14, A[5]=7

Within the recursion: index 2 (value 4). Children: index 4 (14) and index 5 (7). Largest is index 4 (14) — swap required.

STEP 14 OF 17

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

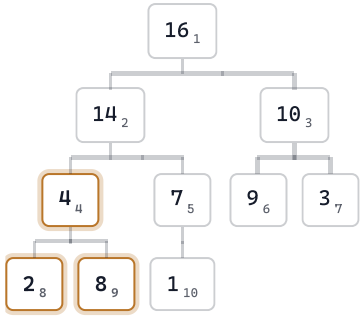


SWAP A[2]<->A[4]; RECURSE INTO INDEX 4, WHICH NOW HOLDS THE DISPLACED 4

Swap A[2] and A[4]: 14 moves up, 4 moves down to index 4. Recurse again into MAX-HEAPIFY(4).

STEP 15 OF 17

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

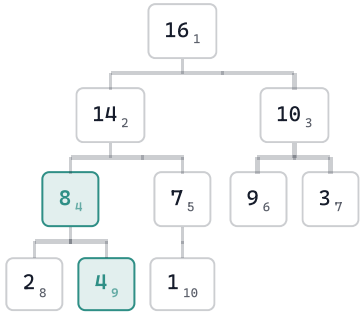


COMPARE A[4]=4 WITH CHILDREN A[8]=2, A[9]=8

Within the recursion: index 4 (value 4). Children: index 8 (2) and index 9 (8). Largest is index 9 (8) — swap required.

STEP 16 OF 17

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

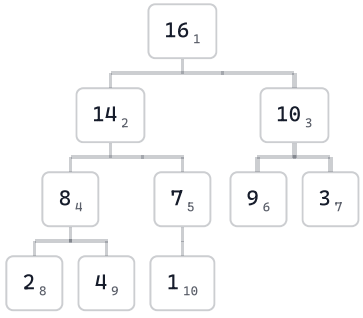


SWAP A[4]<->A[9]; INDEX 9 IS A LEAF — SETTLED. I = 1 TOTAL: 3 SWAPS, CASCADING ALL THREE LEVELS

Swap A[4] and A[9]: 8 moves up, 4 moves down to a leaf — settled. **i = 1 total: 3 swaps**, cascading through all three levels of the tree — the most expensive single call in this entire build.

STEP 17 OF 17

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



VALID MAX-HEAP — EVERY PARENT ≥ BOTH CHILDREN, CHECKED AT EVERY NODE

`Heapify calls: 5 (i = 5,4,3,2,1)` `Total swaps: 7` `Total cost: O(n), not O(n log n)`

Done. A = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1] is now a valid max-heap. Swap tally by call: i=5→0, i=4→1, i=3→1, i=2→2, i=1→3 — **7 swaps total**. Most of the work (i=5,4,3) was cheap; only the top couple of calls cascaded deep — exactly the pattern the tight O(n) analysis predicts.

The loose bound says O(n log n). The tight bound says O(n).

THE LOOSE (WRONG) ARGUMENT

"n calls to MAX-HEAPIFY, each costing O(log n)" gives O(n log n). This is a valid upper bound — but it's not tight, because it assumes *every* node sits near the top of the tree, costing a full O(log n). Most nodes don't.

THE TIGHT ARGUMENT

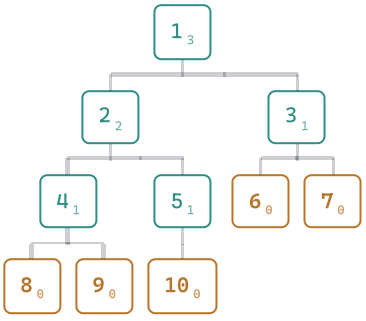
MAX-HEAPIFY on a node of height h costs O(h) — and a heap of n nodes has at most $\lceil n/2^{h+1} \rceil$ nodes at height h. So total cost is bounded by:

$$\sum_{h=0}^{\lg n} \lceil n/2^{h+1} \rceil \cdot O(h) = O(n \cdot \sum_{h=0}^{\lg n} h/2^h) = \mathbf{O(n)}$$

The sum $\sum h/2^h$ converges to a constant (2) regardless of n — most nodes are near the bottom (cheap, O(1)-ish), and only a few are near the top (expensive, O(log n)), and that trade-off exactly cancels out to linear total work.

THE SAME FORMULA, WITH ACTUAL NUMBERS — THE N = 10 HEAP FROM L16·04

First, what "height" means here. It is counted **upward from the bottom**, not downward from the root: a **leaf has h = 0**, its parent h = 1, and the root has the largest height of all. (Depth is the other one — that counts down from the root.) That is why the exponent is h+1: at h = 0 the formula gives ⌈n/2⌉, i.e. *half the nodes are leaves* — exactly right for a complete tree.



NODE INDEX, WITH ITS HEIGHT AS THE SUBSCRIPT · AMBER = HEIGHT 0 (LEAF)

HEIGHT H	WHICH NODES	ACTUAL	BOUND $\lceil N/2^{H+1} \rceil$
0 (leaves)	6, 7, 8, 9, 10	**5**	⌈10/2⌉ = 5
1	3, 4, 5	**3**	⌈10/4⌉ = 3
2	2	**1**	⌈10/8⌉ = 2
3	1 (root)	**1**	⌈10/16⌉ = 1

The counts total 10, and the heap's own height is 3 = ⌊lg 10⌋. Notice h = 2: the bound says "at most 2" while the truth is 1 — it *is* only an upper bound, which is all the proof needs.

And this is the whole reason the answer is O(n). The *most numerous* class is h = 0 — about half of every heap — and it costs **O(0)**. BUILD-MAX-HEAP does not even call those: the loop stops at ⌊n/2⌋ = 5, so nodes 6…10 are skipped outright. Meanwhile the *expensive* class, h = lg n, costs O(log n) but contains exactly **one** node. Every time the cost goes up by one, the population halves — and it is that skew, not the per-call cost, that collapses the total to linear.

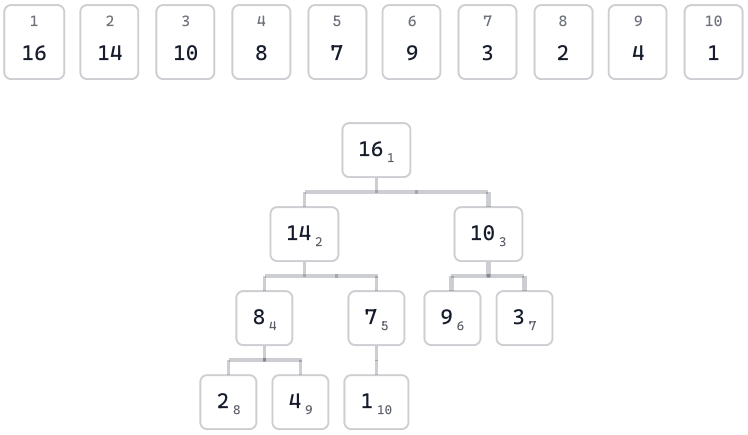
~10 MIN

Append at the end, then bubble up until the heap property is restored

Two insertions into the heap we just built: **17**, which bubbles all the way from a leaf to the new root (the worst case), and **5**, which settles in a single comparison (the best case). Both are shown in full — including the comparison that causes *zero* swaps.

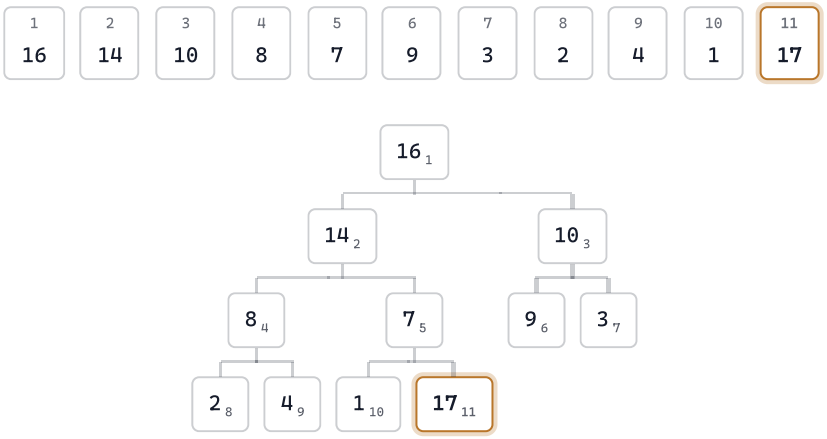
INTERACTIVE — INSERT 17, THEN INSERT 5

STEP 1 OF 12



Starting heap (10 elements) from BUILD-MAX-HEAP. **Insert 17:** append it at the next free position — index 11 — then let it sift up.

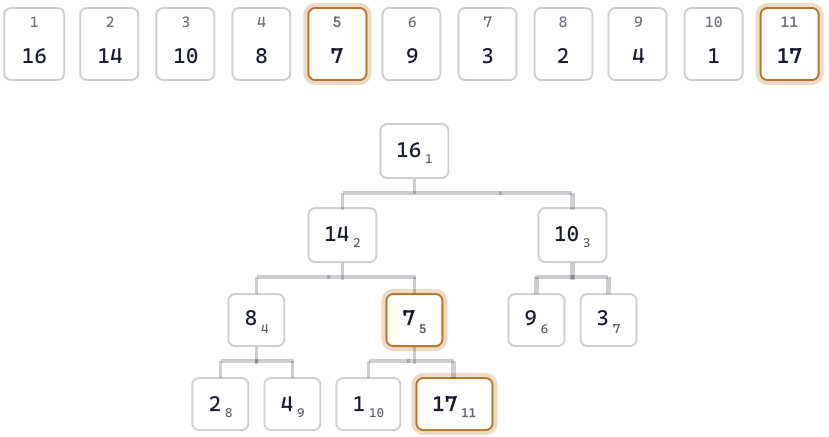
STEP 2 OF 12



APPENDED AT INDEX 11, THE NEXT OPEN SLOT IN THE COMPLETE TREE

Append 17 at index 11. The shape property is preserved (still a complete tree) — but the heap property might now be broken between index 11 and its parent.

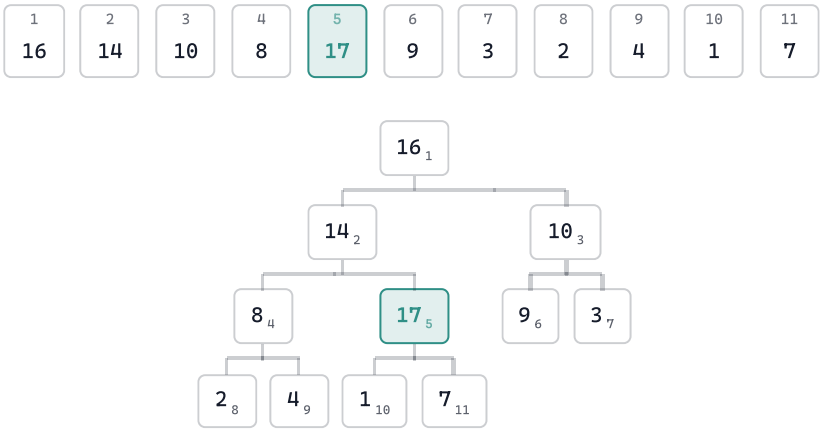
STEP 3 OF 12



COMPARE A[11]=17 WITH PARENT A[5]=7

Compare with parent, index 5 (value 7). 17 > 7 — violates the max-heap property. Swap required.

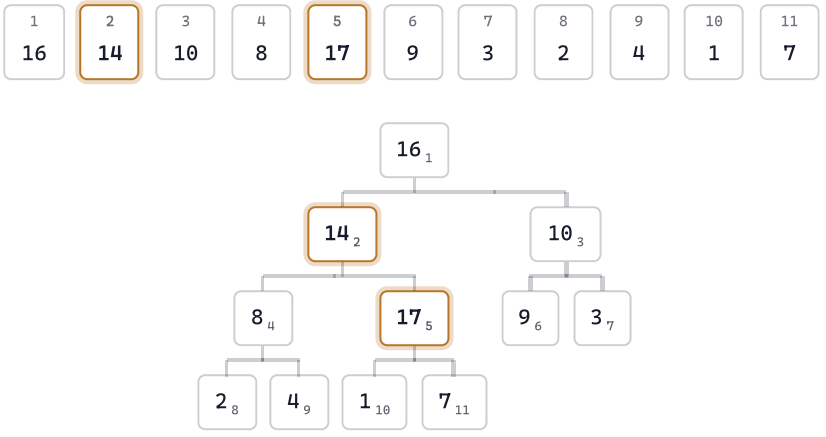
STEP 4 OF 12



17 NOW AT INDEX 5; CONTINUE COMPARING UPWARD

Swap: 17 moves to index 5, 7 moves down to index 11. 17 is not at the root yet — keep comparing upward.

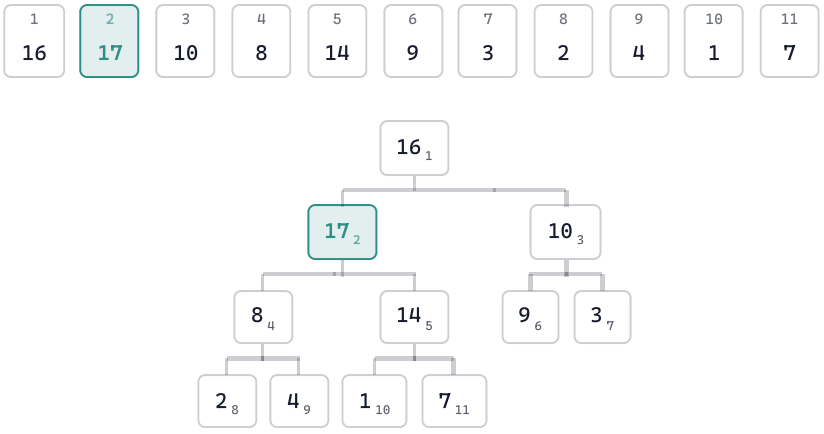
STEP 5 OF 12



COMPARE A[5]=17 WITH PARENT A[2]=14

Compare with the new parent, index 2 (value 14). 17 > 14 — swap required again.

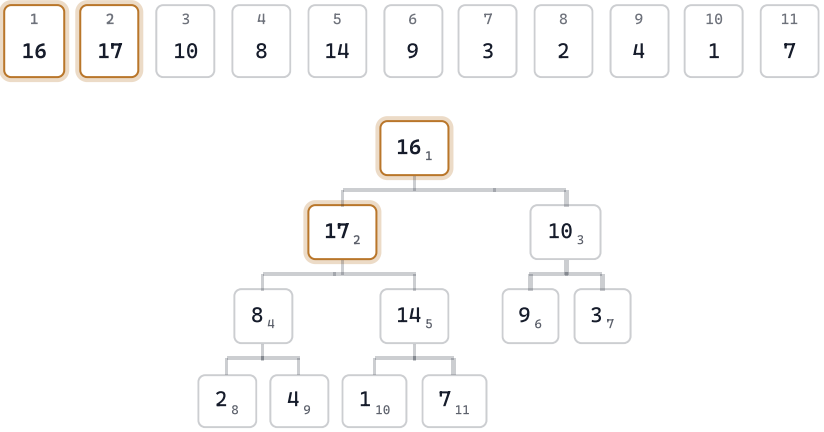
STEP 6 OF 12



17 NOW AT INDEX 2; CONTINUE COMPARING UPWARD

Swap: 17 moves to index 2, 14 moves down to index 5. Still not at the root — keep comparing.

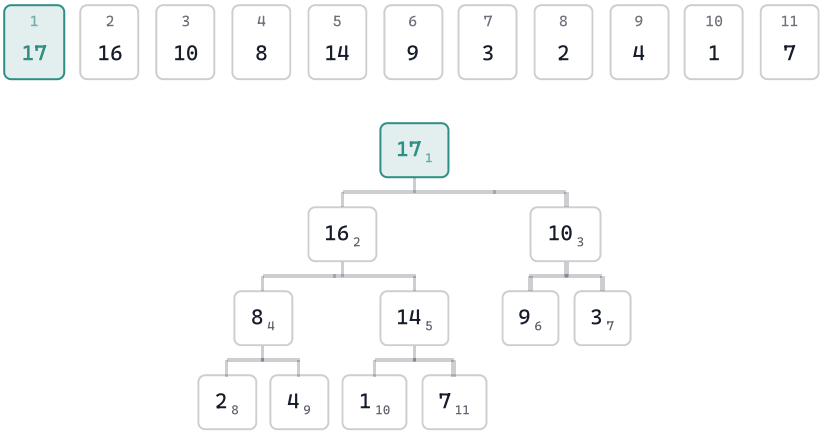
STEP 7 OF 12



COMPARE A[2]=17 WITH PARENT A[1]=16

Compare with the root, index 1 (value 16). 17 > 16 — one more swap required.

STEP 8 OF 12

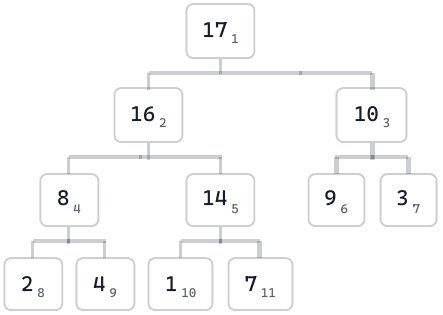


17 HAS REACHED INDEX 1 — THE ROOT; NO PARENT LEFT TO COMPARE AGAINST

Swap: 17 reaches the root. There is no parent left to compare against, so HEAP-INSERT stops. **3 swaps total** — matching the tree's height ($\lfloor \log_2 11 \rfloor = 3$), the worst case for insertion.

STEP 9 OF 12

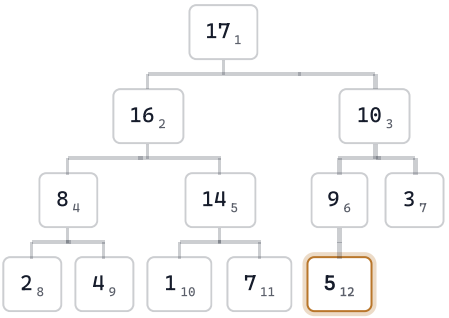
1	2	3	4	5	6	7	8	9	10	11
17	16	10	8	14	9	3	2	4	1	7



Insert 5 next, into this 11-element heap: append at index 12, the next free slot.

STEP 10 OF 12

1	2	3	4	5	6	7	8	9	10	11	12
17	16	10	8	14	9	3	2	4	1	7	5

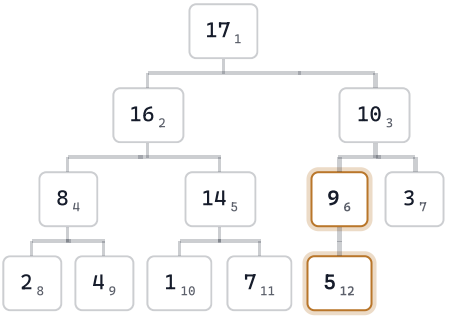


APPENDED AT INDEX 12

Append 5 at index 12.

STEP 11 OF 12

1	2	3	4	5	6	7	8	9	10	11	12
17	16	10	8	14	9	3	2	4	1	7	5

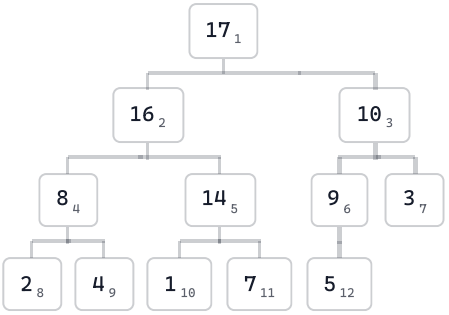


COMPARE A[12]=5 WITH PARENT A[6]=9 — NO SWAP NEEDED

Compare with parent, index 6 (value 9). 5 < 9 — the max-heap property already holds. **No swap needed** — HEAP-INSERT stops after a single comparison. Best case: O(1) for this particular insertion, still within the O(log n) worst-case bound.

STEP 12 OF 12

1	2	3	4	5	6	7	8	9	10	11	12
17	16	10	8	14	9	3	2	4	1	7	5



VALID 12-ELEMENT MAX-HEAP

Insert 17: 3 swaps (full bubble to root)

Insert 5: 0 swaps (settles immediately)

Both within O(log n) worst case

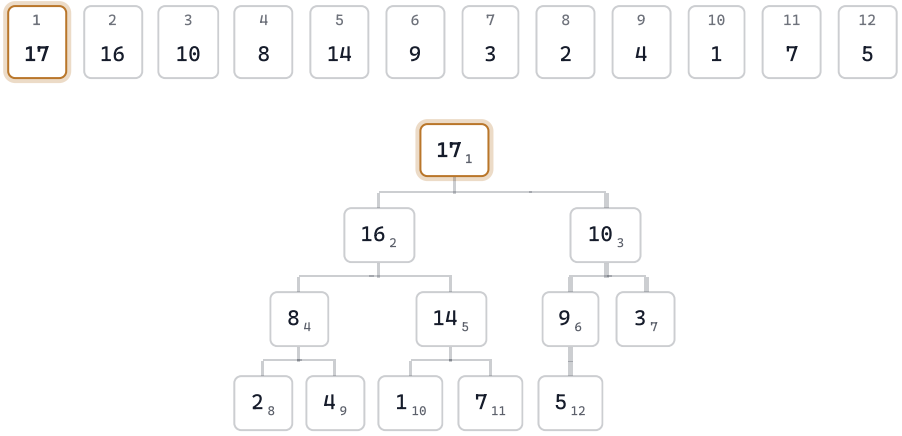
Both insertions land within the same O(log n) worst-case bound, but the *actual* work varies enormously with the value being inserted — 3 swaps versus 0. This 12-element heap is exactly what HEAP-EXTRACT-MAX will operate on next.

Take the root, promote the last leaf, then sink it back into place

Extracting the maximum from the 12-element heap built by the last two insertions. The root (17) is removed and returned; the *last* element takes its place (keeping the shape property intact); then MAX-HEAPIFY sinks it down until the heap property holds again — three full levels, every comparison shown.

INTERACTIVE — HEAP-EXTRACT-MAX(A)

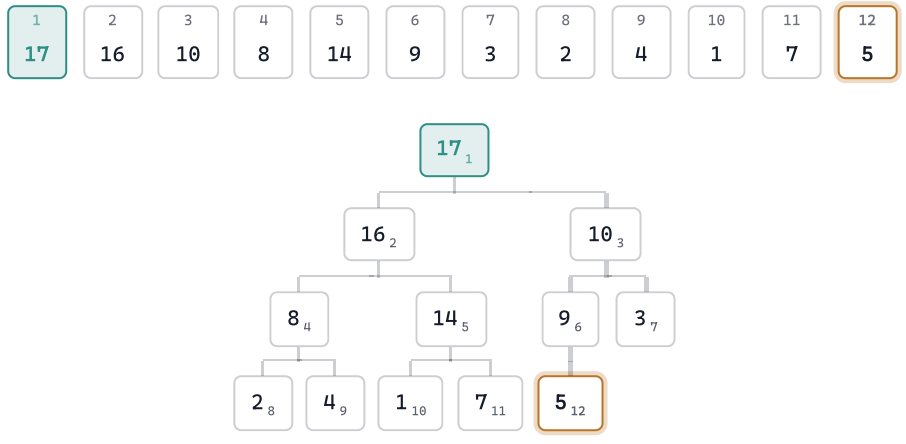
STEP 1 OF 10



THE ROOT ALWAYS HOLDS THE MAXIMUM

HEAP-EXTRACT-MAX: the root, index 1, always holds the maximum — here, **17**. This is the value the operation returns.

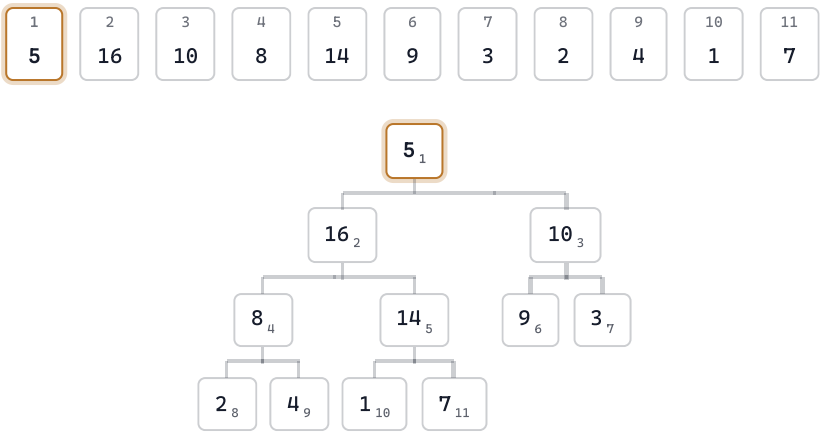
STEP 2 OF 10



INDEX 12 (VALUE 5) IS THE LAST ELEMENT IN THE ARRAY

Save 17 as the return value. Now take the very **last** element — index 12 (value 5) — because moving it (not any arbitrary node) is what keeps the shape property a valid complete tree after we shrink by one.

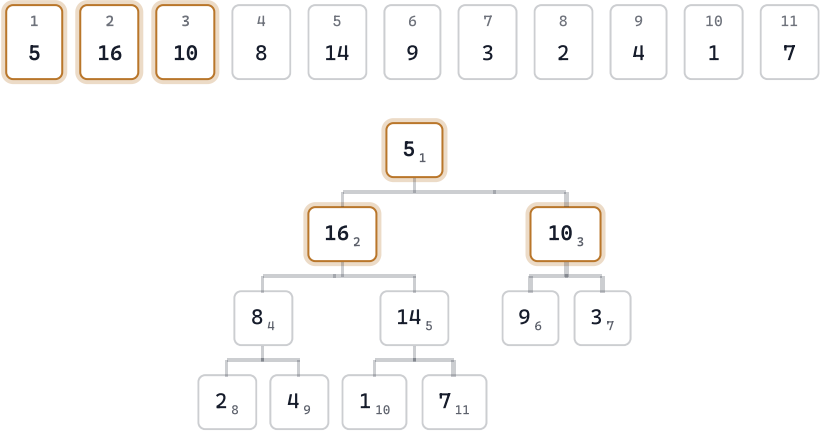
STEP 3 OF 10



INDEX 12 REMOVED; HEAP SIZE SHRUNK FROM 12 TO 11; INDEX 1 NOW HOLDS 5

Move 5 into index 1 (the root) and shrink the heap size to 11 — index 12 no longer exists. The shape property holds again, but index 1 almost certainly breaks the heap property now. Call MAX-HEAPIFY(1).

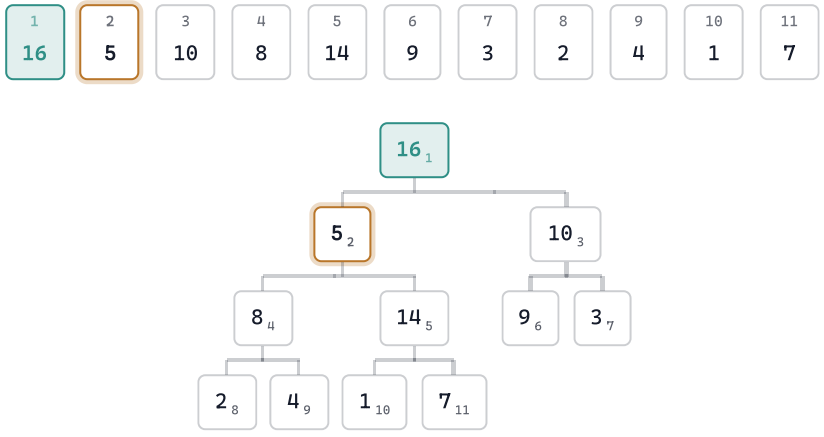
STEP 4 OF 10



COMPARE A[1]=5 WITH CHILDREN A[2]=16, A[3]=10

Compare index 1 (value 5) with its children: index 2 (16) and index 3 (10). Largest is index 2 (16) — swap required.

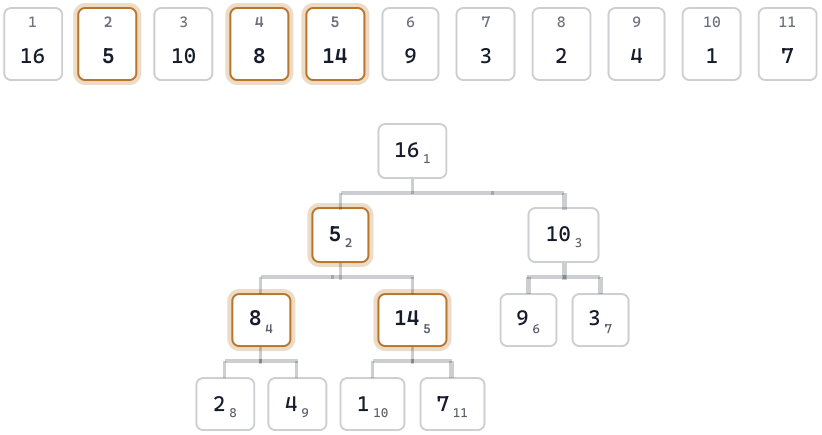
STEP 5 OF 10



SWAP A[1]<->A[2]; RECURSE INTO INDEX 2

Swap: 16 moves to the root, 5 moves down to index 2. Recurse into MAX-HEAPIFY(2) to check 5 against its own children.

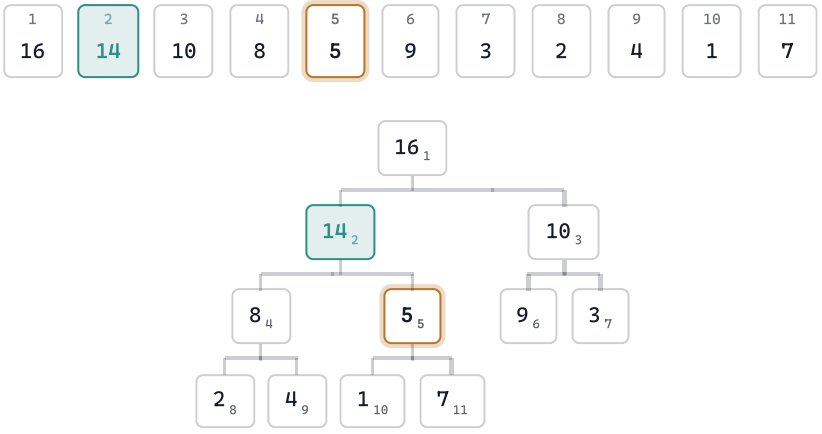
STEP 6 OF 10



COMPARE A[2]=5 WITH CHILDREN A[4]=8, A[5]=14

Compare index 2 (value 5) with children: index 4 (8) and index 5 (14). Largest is index 5 (14) — swap required.

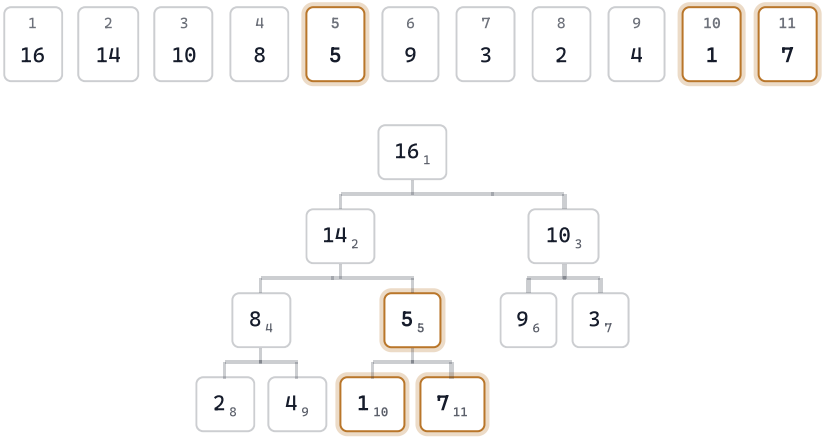
STEP 7 OF 10



SWAP A[2]<->A[5]; RECURSE INTO INDEX 5

Swap: 14 moves up, 5 moves down to index 5. Recurse into MAX-HEAPIFY(5).

STEP 8 OF 10

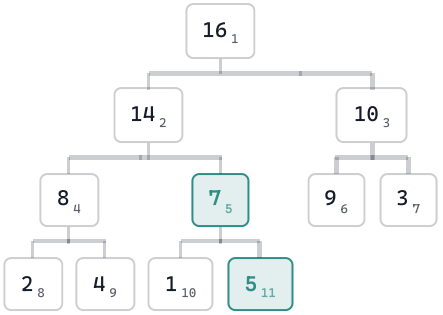


COMPARE A[5]=5 WITH CHILDREN A[10]=1, A[11]=7

Compare index 5 (value 5) with children: index 10 (1) and index 11 (7). Largest is index 11 (7) — swap required.

STEP 9 OF 10

1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	7	9	3	2	4	1	5

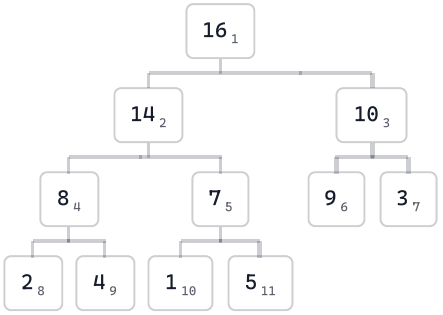


SWAP A[5]<->A[11]; INDEX 11 IS A LEAF WITHIN THE SHRUNK HEAP — SETTLED

Swap: 7 moves up, 5 moves down to index 11 — a leaf in this now-11-element heap, so MAX-HEAPIFY stops here. **3 swaps total** to restore the property, cascading root → index 2 → index 5 → index 11.

STEP 10 OF 10

1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	7	9	3	2	4	1	5



VALID 11-ELEMENT MAX-HEAP

Returned max: 17 | Swaps to restore heap: 3 | Height of an 11-node heap: 3 — matches exactly

Done. HEAP-EXTRACT-MAX returned 17 and left a valid 11-element max-heap behind, using exactly 3 swaps — matching the O(log n) bound (height 3 for 11 nodes) precisely, just like HEAP-INSERT's worst case earlier.

Everything we just verified by hand, in one table

OPERATION	COST	VERIFIED TODAY AS
PEEK-MAX / PEEK-MIN	O(1)	always the root, A[1]
MAX-HEAPIFY (sift-down from height h)	O(h) = O(log n) worst case	up to 3 comparisons in a 12-node heap
HEAP-INSERT (sift-up)	O(log n) worst case	3 swaps for insert(17); 0 swaps for insert(5)
HEAP-EXTRACT-MAX (sift-down)	O(log n) worst case	3 swaps to restore a 12-node heap
BUILD-MAX-HEAP	**O(n)** tight bound (not O(n log n))	7 total swaps across 5 heapify calls on 10 elements

n = number of elements currently in the heap; h = height of the node being sifted, which is at most $\lfloor \log_2 n \rfloor$.

Not a hypothetical exercise — you've already used one

HUFFMAN CODING — LECTURE 6

The classic Huffman algorithm repeatedly extracts the *two lowest-frequency* nodes and reinserts their merged parent — exactly the extract-min / insert cycle we animated today, run n−1 times, backed by a min-priority queue.

EXTERNAL SORTING — LECTURE 5

Lecture 5's tournament trees solved the same underlying problem — repeatedly identify the current minimum across many runs — for merge and replacement selection. A binary heap is the in-memory, array-based cousin of that same repeated-extract-min idea.

HEAP SORT — NEXT LECTURE

BUILD-MAX-HEAP once, then repeatedly swap the root with the last element and MAX-HEAPIFY the shrinking heap — an in-place O(n log n) sort using nothing we haven't already built today. Full treatment in Lecture 17.

JOB / EVENT SCHEDULING

Operating-system schedulers and discrete-event simulators use a priority queue keyed on deadline or timestamp — "run the most urgent job next" is the priority-queue problem, verbatim.

GRAPH ALGORITHMS (GENERAL CS KNOWLEDGE)

Dijkstra's shortest-path and Prim's minimum-spanning-tree algorithms both repeatedly pick the cheapest available edge/vertex — the standard implementation uses a binary heap as the priority queue, upgraded to a Fibonacci heap (Lecture 19) when decrease-key dominates.

MEDIAN / ORDER-STATISTICS MAINTENANCE

A pair of heaps (one max-heap for the lower half, one min-heap for the upper half) maintains a running median in O(log n) per insertion — a common interview and streaming-data pattern.

One array, two properties, three operations — all O(log n) or better

- **Shape property + heap property** together let a complete binary tree live entirely inside a plain array — PARENT(i)=⌊i/2⌋, LEFT(i)=2i, RIGHT(i)=2i+1, no pointers anywhere.
- **BUILD-MAX-HEAP is O(n), not O(n log n)** — verified today on a 10-element array (7 total swaps across 5 heapify calls) and proven in general via the height-weighted sum bound.
- **HEAP-INSERT sifts up** in at most O(log n) swaps — verified with a full root-to-leaf bubble (insert 17, 3 swaps) and an immediate settle (insert 5, 0 swaps).
- **HEAP-EXTRACT-MAX sifts down** in at most O(log n) swaps — verified with a full 3-level cascade after removing the root and promoting the last leaf.
- **The trade-off:** heaps are the right structure exactly when the only operation you need repeatedly is "give me the extreme element" — not arbitrary search, not fast merging, not sorted iteration. Lectures 18–21 build heap variants that relax exactly those limitations.

LECTURE 17 — NEXT

Heap Sort and Priority Queue Applications

Turn today's BUILD-MAX-HEAP and MAX-HEAPIFY into an in-place O(n log n) sort, then apply priority queues to scheduling problems.

HOMEWORK — BRING TO LECTURE 17

- Run BUILD-MAX-HEAP by hand on A = [1, 7, 3, 12, 5, 9, 2, 15, 8]. Show every MAX-HEAPIFY call and count total swaps.
- Starting from your heap above, insert 20 and then insert 4. Which one bubbles further, and why?
- From the heap after both insertions, run HEAP-EXTRACT-MAX twice in a row. Show the array after each extraction.
- In 3–4 sentences: why can't you binary-search a max-heap's array to check whether a given key is present? What property would you need that a heap doesn't guarantee?

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

> *Next: Lecture 17 — Heap Sort and Priority Queue Applications.*
