Fibonacci Heaps & Amortized Efficiency

Binomial heaps fixed merging but left DECREASE-KEY at O(log n). Today's structure gets it down to O(1) amortized — by doing almost nothing when you'd expect it to do work, and paying the bill later, all at once, whenever EXTRACT-MIN forces a cleanup.

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

~95 minutes
Session outcome: analyze lazy operations and amortized complexity

~128 MIN

Today, minute by minute

Why Fibonacci heaps — the decrease-key gap 00–06

Dijkstra and Prim call decrease-key far more often than extract-min. O(log n) per call adds up fast.

Where DECREASE-KEY earns its keep 06–11

One minute of Dijkstra: extract-min once per vertex, decrease-key once per edge.

Structure: the one rule — ordered by key 11–15

Circular doubly-linked lists, degree, mark — and the "do nothing now, clean up later" philosophy.

Structure: no shape rules at all 15–19

Same 8 nodes as a binomial B_3, none of the rules — and why giving up order is the whole point.

Structure: how it is stored 19–23

Four pointers per node, circular doubly-linked lists, and the single pointer that is the heap.

Structure: the mark bit 23–28

Lose one child quietly, lose two and you are cut loose — the cascading cut, animated.

Structure: what the mark bit prevents 28–32

Degree must be earned: a second loss disconnects the child and drops the parent's degree with it.

Animation: INSERT 32–37

Add a singleton tree to the root list. That's the whole operation.

Animation: EXTRACT-MIN, in full 37–61

Remove the root, promote its children, then consolidate — every array check, every link.

Animation: MERGE / UNION 61–65

Splice two circular root lists together. O(1), regardless of size.

Animation: DECREASE-KEY, all three cases 65–79

No cut, a single cut, and a cascading cut — traced on the same heap.

Animation: DELETE 79–91

Decrease to $-\infty$, then extract-min — no new machinery needed.

The potential function $\Phi = t(H) + 2m(H)$ 91–95

Trees measure looseness; marks measure deferred cleanup debt.

Why the coefficient on marks is 2 95–99

One cascading cut through the formula, both ways — and why INSERT is charged a surcharge rather than a refund.

Why decrease-key is O(1): the common case 99–102

One cut, one mark, no cascade — and the +3 it banks for later.

Why decrease-key is O(1): with a cascade 102–108

Actual cost O(c), potential falls by c — the two cancel whatever the length.

The operation that gets called the most is the one still costing O(log n)

REAL-LIFE EXAMPLE — ROAD-NETWORK ROUTING

Think of Dijkstra's shortest-path algorithm computing fastest routes across a road network with millions of intersections. Every time it finds a shorter path to an intersection it has already seen, it must **decrease that intersection's priority** in the queue. This happens once per road segment examined — potentially millions of times — while EXTRACT-MIN (pulling out "visit this intersection next") happens only once per intersection.

WHY BINOMIAL HEAPS STILL AREN'T ENOUGH

Lecture 18's binomial heap fixed merging, but DECREASE-KEY still costs O(log n) there — every decrease may need to bubble a node up through a binomial tree's structure. When decrease-key is called far more often than extract-min (exactly Dijkstra's and Prim's access pattern), that O(log n) constant is paid millions of times over.

THE FIBONACCI HEAP'S ANSWER

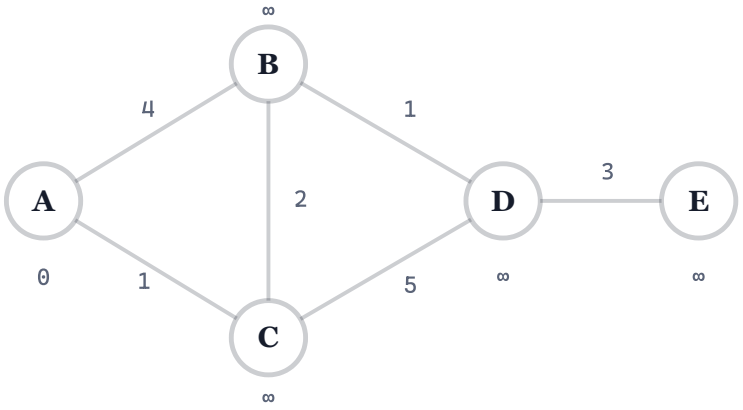
Make DECREASE-KEY amortized **O(1)** — by refusing to maintain a tidy tree shape at all times. A Fibonacci heap lets its structure get "messy" (many small trees, some nodes cut loose) during INSERT, UNION, and DECREASE-KEY, and only pays the cost of tidying up — consolidating everything into a clean shape — during EXTRACT-MIN. This is the "lazy" philosophy the whole structure is built around.

One minute of Dijkstra, to see why that one operation matters

The previous slide claimed DECREASE-KEY is called far more often than EXTRACT-MIN. Rather than assert it, here is a five-vertex shortest-path run reduced to **only its two heap operations** — no predecessor arrays, no path reconstruction. Watch which one fires more, and watch the moment a vertex *already sitting in the queue* gets a better distance.

INTERACTIVE — DIJKSTRA FROM A, SHOWING ONLY THE QUEUE OPERATIONS

STEP 1 OF 9

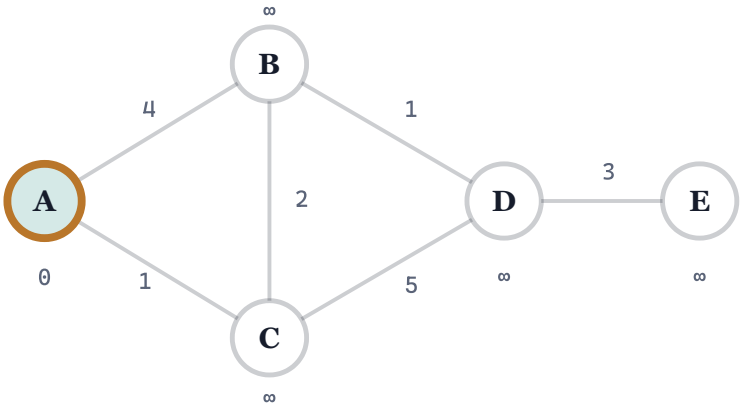


priority queue: A:0 B:∞ C:∞ D:∞ E:∞

EVERY VERTEX STARTS IN THE QUEUE — THE SOURCE AT 0, EVERYTHING ELSE AT ∞

Shortest paths from **A**. Every vertex goes into a priority queue keyed by its best known distance: **A at 0**, everything else at **∞**. From here the algorithm does only two things to that queue, over and over.

STEP 2 OF 9

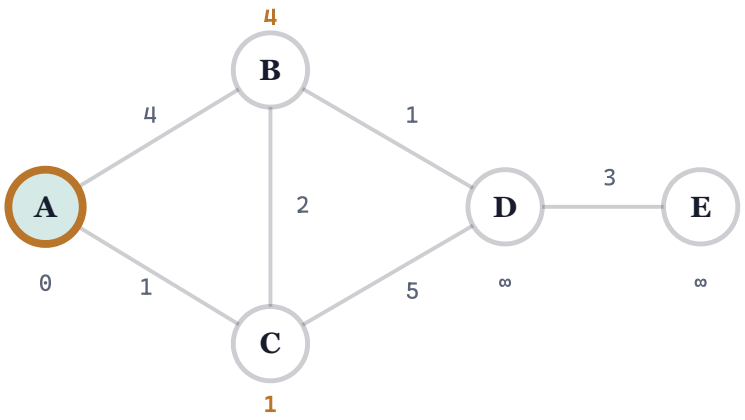


priority queue: B:∞ C:∞ D:∞ E:∞

EXTRACT-MIN #1 — A IS SETTLED AND NEVER RETURNS TO THE QUEUE

EXTRACT-MIN takes the smallest key: **A**, at distance 0. A is now settled — its shortest distance is final and it never re-enters the queue. That is the first of the two operations.

STEP 3 OF 9

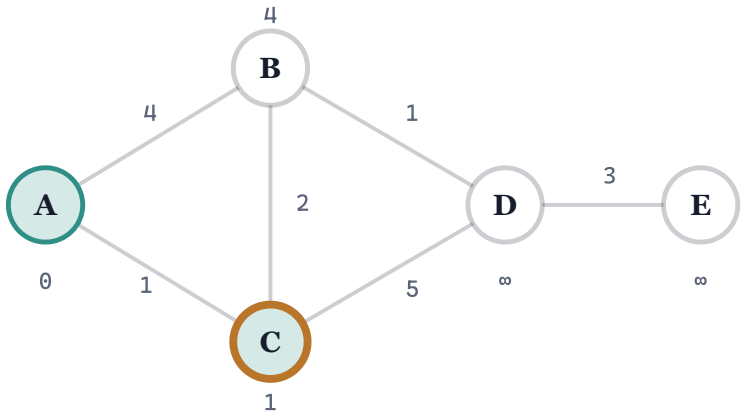


priority queue: B:4 C:1 D:∞ E:∞

TWO DECREASE-KEY CALLS — ONE PER EDGE LEAVING A

Now examine A's edges. A→B offers 4, and A→C offers 1 — both beat ∞, so both keys are lowered. **Two DECREASE-KEY calls, one per edge.** Notice B and C were already sitting in the queue; we did not insert them, we *improved* them.

STEP 4 OF 9

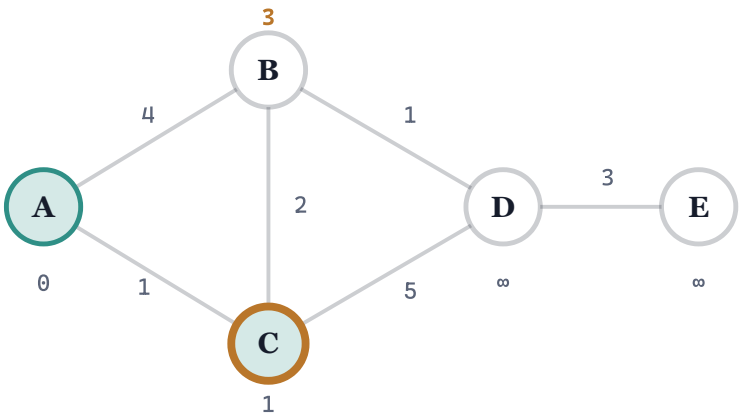


priority queue: B:4 D:∞ E:∞

EXTRACT-MIN #2 — C HAS THE SMALLEST KEY, 1

EXTRACT-MIN again: the smallest key in the queue is **C at 1**, so C is settled next.

STEP 5 OF 9

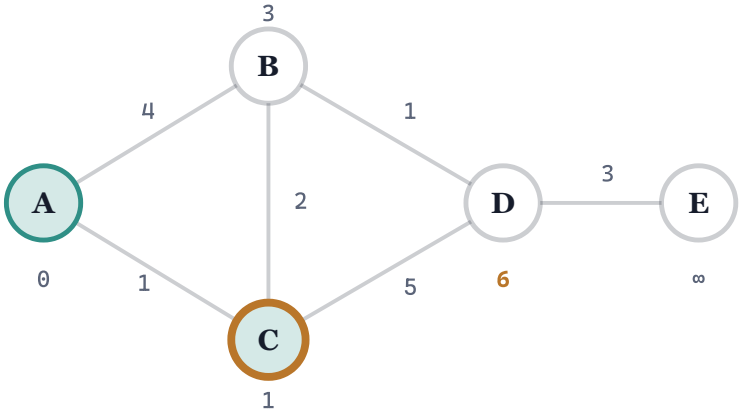


priority queue: B:3 D:∞ E:∞

B WAS ALREADY IN THE QUEUE AT 4 — THE ROUTE THROUGH C IS SHORTER, SO ITS KEY DROPS TO 3

This is the moment worth watching. Edge C→B costs 2, so reaching B via C costs 1 + 2 = **3** — better than the 4 it already had. B is *not* new: it is sitting in the queue with key 4, and that key must now be **decreased to 3**. The heap has to find that node and lower its key without disturbing anything else. **That is DECREASE-KEY, and it is the operation Dijkstra performs most.**

STEP 6 OF 9

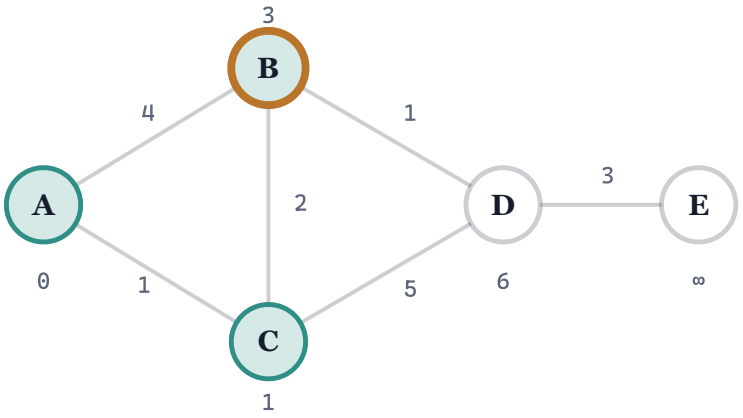


priority queue: B:3 D:6 E:∞

C'S OTHER EDGE — ANOTHER DECREASE-KEY

C's other edge, C→D with weight 5, gives D a distance of 6. Another **DECREASE-KEY**. Two extract-mins so far; four decrease-keys.

STEP 7 OF 9

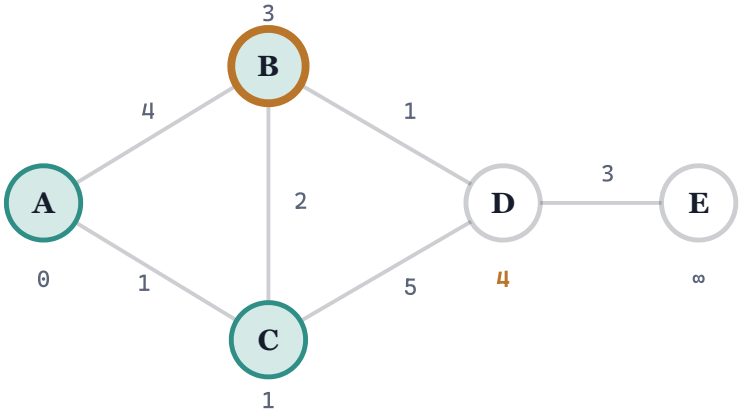


priority queue: D:6 E:∞

EXTRACT-MIN #3 — B, AT ITS IMPROVED DISTANCE OF 3

EXTRACT-MIN takes **B at 3** — the improved value, not the 4 it originally held. Had we not been able to decrease its key, B would still be carrying a stale 4 and the queue would be wrong.

STEP 8 OF 9

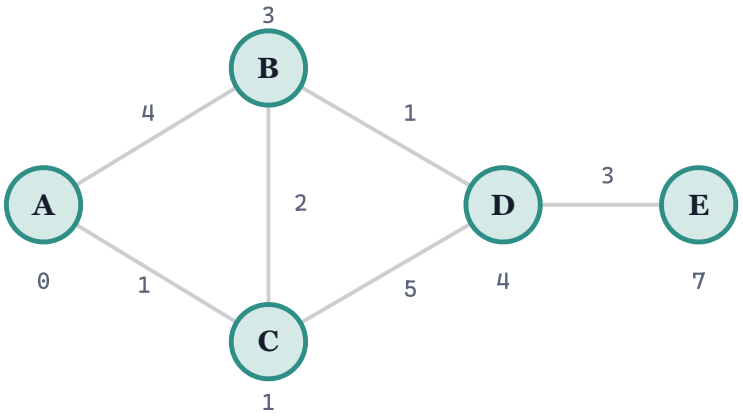


priority queue: D:4 E:∞

D IMPROVES AGAIN: 6 → 4 — A SECOND KEY ALREADY IN THE QUEUE BEING LOWERED

From B, edge B→D costs 1, so D can be reached in 3 + 1 = **4** instead of 6. D is in the queue, so once more its key is **decreased**. That is the second time a vertex already waiting in the queue has had its distance improved — and in a bigger graph this happens constantly.

STEP 9 OF 9



priority queue: empty

FINISHED — 5 VERTICES SETTLED, AND THE QUEUE IS EMPTY

EXTRACT-MIN: 5 (once per vertex) | DECREASE-KEY: 6 (once per edge) | |V| = 5, |E| = 6

Done — and look at the tally. EXTRACT-MIN ran **5** times, once per vertex. DECREASE-KEY ran **6** times, once per edge. This graph is tiny and already the edge count wins; in a road network or a router topology, where every node has many neighbours, **E dwarfs V** and DECREASE-KEY becomes overwhelmingly the most frequent operation. Making *it* free is worth far more than speeding up the rare extraction — which is exactly what a Fibonacci heap sets out to do.

COUNT THEM: THE TWO OPERATIONS ARE NOT CALLED EQUALLY OFTEN

- **EXTRACT-MIN runs once per vertex.** Each vertex is settled exactly once and never returns to the queue — **|V|** calls in total.
- **DECREASE-KEY runs once per edge.** Every edge is examined once, from the settled end, and improves the far end's key whenever it offers a shorter route — up to **|E|** calls.
- In this tiny run: **5** extract-mins against **6** decrease-keys. On a road network or a router topology, where each node has many neighbours, **E vastly outweighs V**.

SO WHICH OPERATION SHOULD YOU OPTIMISE?

Total cost is **|V| × (cost of EXTRACT-MIN) + |E| × (cost of DECREASE-KEY)**. With a binary heap both are logarithmic, so the E term dominates and drags the whole algorithm with it:

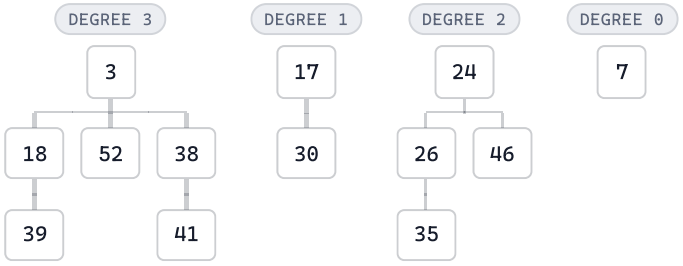
PRIORITY QUEUE	EXTRACT-MIN	DECREASE-KEY	DIJKSTRA TOTAL
binary heap	O(log V)	O(log V)	O(E log V)
Fibonacci heap	O(log V)	**O(1)** amortized	**O(E + V log V)**

Making the *rare* operation faster would barely help. Making the **common** one free is what changes the bound — and that is the single problem the rest of this lecture solves.

Ordered by key — and that is the only promise

A Fibonacci heap is a **forest** of min-heap-ordered trees. Every node's key is ≥ its parent's, exactly as in a binary heap. Hold on to that, because it is the *only* structural rule this heap enforces — the next slide is about everything it deliberately does **not** promise.

A PERFECTLY LEGAL FIBONACCI HEAP



13 NODES · MIN[H] → 3 · DEGREES 3, 1, 2, 0 — IN NO ORDER, AND NOTHING FORBIDS REPEATS

WHAT MIN-HEAP ORDER DOES BUY YOU

- Inside any one tree, the smallest key sits at **that tree's root** — keys only grow as you walk down.
- So the **global** minimum must be one of the roots. A single pointer, `min[H]`, is kept aimed at it, which makes MINIMUM **Θ(1)** — better than a binomial heap, which had to scan.

WHAT IT DOES *NOT* BUY YOU

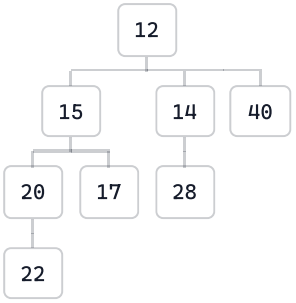
- **Nothing about siblings.** Look at 18, 52 and 38 above — three children of the same node, in no relation to each other whatsoever.
- **Nothing about shape, degree, or tree count.** The heap above has trees of degree 3, 1, 2 and 0 — in that order, with no pattern.

Where a binomial heap was rigid, this one promises nothing

Both trees below hold **8 nodes** and both are perfectly legal — for their own structure.
The difference is how much each one had to *pay* to stay in that shape.

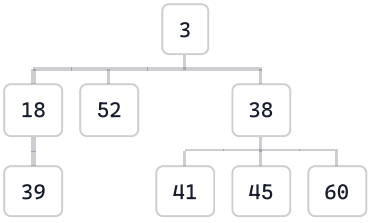
SAME 8 NODES, TWO DIFFERENT SETS OF RULES

LECTURE 18 — A BINOMIAL B_3



CHILDREN MUST READ B2, B1, B0 — EVERY INSERT AND MERGE MAINTAINS THIS

LECTURE 19 — A FIBONACCI TREE



ANY SHAPE AT ALL — NOTHING IS MAINTAINED, SO NOTHING CAN BE VIOLATED

THREE THINGS A FIBONACCI HEAP GIVES UP

- **Children are in no sequence.** "Unordered" here is the graph-theory sense — a node's children sit in their list in whatever order they happened to arrive.
- **Shape is unconstrained.** No fixed child count, no B_k template. Any tree shape at all can occur.
- **The root list is unordered too**, and **duplicate degrees are allowed** to pile up. A binomial heap could never hold two trees of the same degree; here they simply wait until EXTRACT-MIN consolidates them.

WHY GIVING UP ORDER IS THE WHOLE POINT

In Lecture 18 the child list was ordered by **decreasing degree**, and EXTRACT-MIN *depended* on it: it reversed that list to obtain a legal root list. Every operation had to preserve the invariant, and that upkeep is exactly what cost O(log n).

A Fibonacci heap has no invariant to preserve, so there is nothing to repair. EXTRACT-MIN just **splices the children straight into the root list** — no reversal, no ordering, no checks. DECREASE-KEY can rip a node out of the middle of a tree and drop it in the root list, and the structure is still legal by definition. **Work you never promised to do is work you never have to pay for.**

Four pointers per node — one more than a binomial heap, and it buys everything

Cutting a node out of the middle of a list in **O(1)** requires reaching its neighbours on *both* sides. That is the entire reason for the extra pointer.

WHAT ONE NODE HOLDS

`key`	the priority value
`parent`	→ up to its parent (NIL for a root)
`child`	→ **any one** child — just an entry point
`left`	→ previous sibling
`right`	→ next sibling
`degree`	how many children it has
`mark`	has it lost a child? (next slide)

Note `child` points at an *arbitrary* child, not a leftmost one — with no ordering, "leftmost" would mean nothing.

SIBLINGS FORM A CIRCULAR, DOUBLY-LINKED LIST

The three children of node 3, as they actually sit in memory:

Circular, so `38.right = 18` and `18.left = 38` — there is no first or last.

- **Remove any node in O(1)** — `x.left.right = x.right` and `x.right.left = x.left`. No traversal, no head-of-list special case. This is what DECREASE-KEY's cut needs.
- **Splice two lists in O(1)** — join them at any two points. This is what UNION and EXTRACT-MIN need.
- **Lecture 18 could do neither.** Its singly-linked `sibling` chain could only be walked forwards, which is why removing anything meant finding its predecessor first.

THE ROOT LIST — AND THE ONE POINTER THAT *IS* THE HEAP

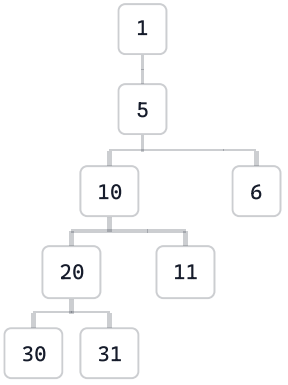
The roots use the very same `left/right` fields to form one circular list, and `min[H]` points at the root holding the smallest key (**3**, highlighted). A whole Fibonacci heap *is* that single pointer. Adding a tree is an O(1) splice anywhere in the ring — which is why INSERT and UNION are O(1), and why the root list is in no particular order.

One bit per node, to stop trees being shredded

DECREASE-KEY works by **cutting** a node out and dropping it in the root list. Losing nodes is harmless in itself — the heap just gets flatter, and EXTRACT-MIN rebuilds the depth later. The danger is subtler: unchecked cuts would let a node keep a high **degree** while the subtrees beneath its children are gutted, and the whole analysis rests on high degree *implying* many descendants. The `mark` bit is the entire defence: **a node may lose one child quietly, but not two.** Below we start from a **completely fresh tree with nothing marked**, run four DECREASE-KEY calls, and watch the marks be earned one at a time — until the fourth call sets off a cut that propagates through **three marked ancestors** before a root finally stops it.

INTERACTIVE — THE MARK BIT AND THE CASCADING CUT

STEP 1 OF 13

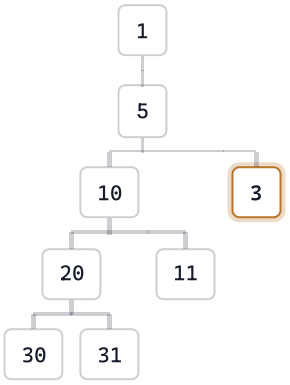


A FRESH TREE — NOTHING IS MARKED YET

nodes: 8 | trees: 1 | marked: 0 | Φ = t + 2m = 1

Start clean: one tree, eight nodes, four levels below the root, and **not a single mark anywhere**. Every node still has every child it was ever given. We will run four DECREASE-KEY calls and watch the marks appear one at a time — then watch what the fourth one costs.

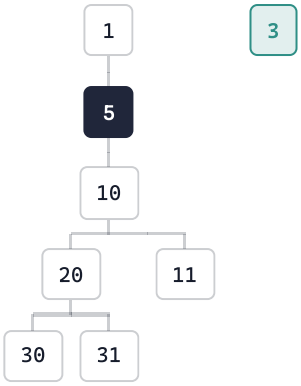
STEP 2 OF 13



KEY 6 IS LOWERED TO 3 IN PLACE — BUT 3 < 5, SO IT CANNOT STAY UNDER 5

Operation 1 — DECREASE-KEY(6, 3). The key is written straight into the node: the node that held **6** now holds **3**, still sitting where it was, as the right child of 5. Min-heap order is now broken, because **3 < 5**. Nothing has moved yet.

STEP 3 OF 13

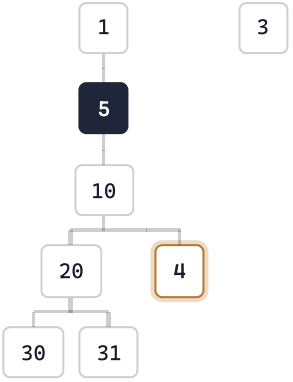


3 IS CUT TO THE ROOT LIST — 5 HAS LOST ITS FIRST CHILD, SO 5 IS NOW MARKED

cuts: 1 | trees: 2 | marked: 5 | Φ: 1 → 4

Cut. Rather than sift 3 upward, the heap simply cuts it out and drops it in the root list. Then the rule is applied to its old parent: **5 has lost its first child, so it is merely marked** and nothing else happens. A quiet, cheap loss — but Φ rose by 3 (one new tree, one new mark). Treat that rise as a deposit.

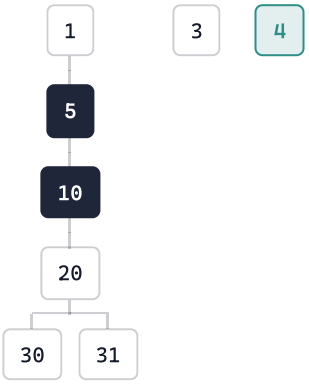
STEP 4 OF 13



KEY 11 IS LOWERED TO 4 IN PLACE — 4 < 10, ANOTHER VIOLATION

Operation 2 — DECREASE-KEY(11, 4). Same move one level down: the node holding 11 now holds **4**, still attached to its parent 10. Since **4 < 10**, it too must be cut.

STEP 5 OF 13

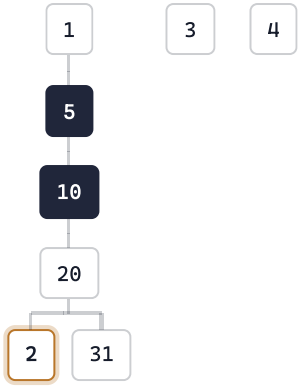


4 IS CUT OUT — 10 IS NOW MARKED TOO

cuts: 1 | trees: 3 | marked: 5, 10 | Φ: 4 → 7

Cut. 4 joins the root list, and **10 loses its first child and is marked**. Two marks now sit on the spine of the tree, one directly above the other.

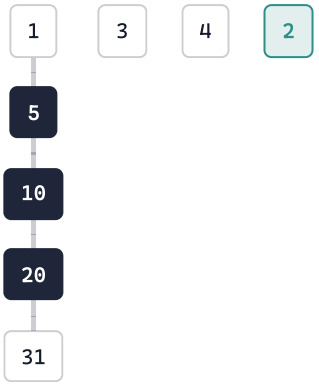
STEP 6 OF 13



KEY 30 IS LOWERED TO 2 IN PLACE — 2 < 20, SO IT MUST GO AS WELL

Operation 3 — DECREASE-KEY(30, 2). The node holding 30 now holds **2**, still a child of 20. **2 < 20**, so another cut is coming — and note that **20 is still unmarked**, so this will be its *first* loss.

STEP 7 OF 13

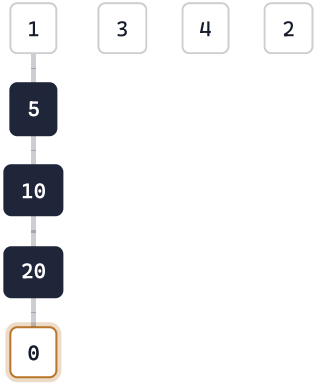


2 IS CUT OUT — **20 IS MARKED AS WELL**: THREE MARKS STACKED UP THE SPINE

`cuts: 1` `trees: 4` `marked: 5, 10, 20` `Φ: 7 → 10`

Cut. Again a first loss, again just a mark. Now look carefully at the tree: **20, 10 and 5 are all marked, one directly above the other**. Every one of them has spent its single free loss. The next node in that chain to lose a child cannot pay quietly.

STEP 8 OF 13



KEY 31 IS LOWERED TO 0 IN PLACE — 0 < 20, AND 20 IS **ALREADY MARKED**

Operation 4 — DECREASE-KEY(31, 0). The last child of 20 now holds **0**, and **0 < 20**, so it has to be cut. But this time the parent, **20, is already marked**: it is about to lose a *second* child. This is exactly the situation the mark bit exists to catch.

STEP 9 OF 13

CUT 1 OF 4 — 0 IS OUT; NOW TEST ITS PARENT 20, WHICH IS MARKED

cuts so far: 1 | trees: 5

Node 0 is cut into the root list as usual. Now the rule runs on **20** — and because 20 is marked, it does not merely get marked again: **20 must be cut loose as well**. The cascade starts here.

STEP 10 OF 13

CUT 2 OF 4 — 20 JOINS THE ROOT LIST, UNMARKED; TEST 10, WHICH IS MARKED

cuts so far: 2 | trees: 6 | marked: 5, 10

Cascade, step 1. 20 is cut from 10 and its mark is cleared on arrival in the root list. The same test now applies to **10** — marked as well, so it goes too. Each of these is O(1) pointer surgery; nothing is searched or rebuilt.

STEP 11 OF 13

CUT 3 OF 4 — 10 IS OUT, UNMARKED; TEST 5, WHICH IS MARKED

cuts so far: 3 | trees: 7 | marked: 5

Cascade, step 2. 10 is cut from 5, unmarked, and the test moves up again. **5 is marked** too — the deposit it made back in operation 1 is now being collected.

STEP 12 OF 13

CUT 4 OF 4 — 5 IS OUT; ITS PARENT IS 1, A ROOT, SO THE CASCADE STOPS

cuts: 4 | trees: 8 | marked: 0 | roots are never marked

Cascade, step 3 — and the stop. 5 is cut from 1 and unmarked. The rule is applied once more, to **1** — but 1 is a **root**. A root has no parent to answer to, so it is never marked and the cascade **halts**. Reaching the top of a tree is the only thing that terminates the chain.

STEP 13 OF 13

0 1 2 3 4 5 10 20

THE WHOLE TREE HAS DISSOLVED INTO **8 SEPARATE ROOTS** — AND NO NODE WAS LOST

nodes: 8 (unchanged) | trees: 1 → 8 | marked: 0 | Φ: 10 → 8 | min[H] = 0

Done. Operation 4 performed **4 cuts** — far more work than operations 1–3 — yet it is still O(1) amortized. Check the books: ΔΦ = +4 trees − 2×3 marks = **−2**, so the amortized cost is 4 + (−2) = **2**. The three quiet operations each deposited Φ = +3 in advance, and this one spent it. **That is what the mark bit buys: expensive cascades are pre-paid, so no single DECREASE-KEY is ever expensive on average.**

THE RULE, IN THREE LINES

- `mark[x]` = "has x lost a child since the last time x itself became someone's child?" A freshly linked node starts **unmarked**.
- Loses its **first** child → just set the mark. Nothing else happens.
- Loses a **second** child while marked → x is cut loose too, and the same test is applied to *its* parent — the **cascading cut**.

Roots are never marked. A node moving to the root list has its mark cleared, and the cascade stops there — a root has no parent to punish.

THE LAZY PHILOSOPHY, IN ONE LINE

INSERT, UNION and DECREASE-KEY all do the **minimum possible work** and dump the mess into the root list. Only EXTRACT-MIN ever tidies up — a step called **consolidation** — and it amortizes that cleanup across all the cheap operations that ran before it.

You will watch cascading cuts run inside a full DECREASE-KEY in **L19·06**, and the amortized argument is made properly in **L19·09**.

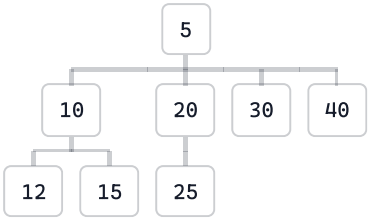
A node must never keep a degree it has not earned

You have just watched a cut cascade four levels and flatten a tree. The fair question is: **what was all that for?** Here is the single thing the rule protects — and it is not the shape of the tree.

THE THING THE RULE FORBIDS: A **HOLLOW** NODE

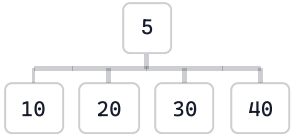
Both roots below report **degree 4**. Remember that degree counts *direct children only* — so cutting a node's **grandchildren** shrinks its subtree without touching its degree at all. That is the loophole.

LEGAL — THE DEGREE WAS EARNED



DEGREE 4, **8 NODES** — EVERY CHILD STILL CARRIES ITS OWN SUBTREE

WHAT THE MARK BIT MAKES IMPOSSIBLE

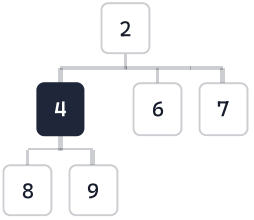


DEGREE 4, **5 NODES** — THE CHILDREN HAVE BEEN GUTTED

The left one earned its degree: it was built by four links, and every child still carries the subtree it brought. The right one is a **fraud** — same degree, three fewer nodes, and it would be free to keep shrinking. If nodes like that were allowed, "degree 4" would tell you nothing about how many nodes lie beneath.

INTERACTIVE — WHY A SECOND LOSS CANNOT BE HIDDEN

STEP 1 OF 3

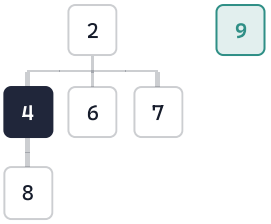


G = 2 HAS DEGREE 3; ITS CHILD P = 4 IS **MARKED** AND STILL HOLDS TWO CHILDREN

degree of G: 3 | P is marked | nodes below G: 5

Grandparent **2** has three children, so degree 3. One of them, **4**, is already **marked** — it lost a child earlier. Suppose an attacker now wants to gut 4 while leaving 2 sitting on its degree of 3. Watch why that is impossible.

STEP 2 OF 3



9 IS CUT TO THE ROOT LIST — THAT IS P'S SECOND LOSS, AND P IS MARKED

degree of G: still 3 | P has now lost two children | so P cannot stay

A DECREASE-KEY cuts **9** away. That is 4's **second** loss, and 4 is marked — so the rule does not let it simply absorb the loss. **4 must be cut loose from 2 as well.**

STEP 3 OF 3

P IS DISCONNECTED — AND G'S DEGREE HAS DROPPED FROM 3 TO 2

degree of G: 3 → 2 | G is now marked | nodes below G: 5 → 2

There it is. 4 leaves for the root list, and because it was one of 2's children, **2's degree falls from 3 to 2** at the same moment. The subtree shrank *and* the degree shrank, together. That is the whole mechanism: **you cannot hollow out a child without the parent paying for it in degree** — and 2 is now marked itself, so its own next loss will propagate one level higher again.

WHY THIS IS WORTH ONE BIT ON EVERY NODE

Look again at the two trees above. The heap has no way to inspect a subtree — when it needs to organise its trees, the only thing it reads off a root is its **degree**. Every EXTRACT-MIN sorts the trees into slots, **one slot per degree**, and how long that takes depends on **how many different degrees can turn up**.

That is what makes a hollow node dangerous. If a node were allowed to keep a large degree while holding almost nothing beneath it, large degrees would become *cheap* — even a small heap could throw up a great many different ones, the slots would multiply, and every EXTRACT-MIN would have more of them to sweep.

The mark bit shuts that down. Because a second loss disconnects the child *and* costs the parent its degree, **a node can only hold a big degree if it genuinely holds a big subtree**. Big degrees stay expensive to obtain, so only a handful of different degrees can exist in a heap at any moment — and EXTRACT-MIN stays quick.

You will see those slots for yourself in L19·04, when EXTRACT-MIN consolidates: one slot per degree, filled and collided until at most one tree of each degree is left standing.

Add one singleton tree to the root list — nothing else happens

Building a Fibonacci heap from scratch with four inserts. No comparison against any tree's internal structure is ever needed — only against the current min.

INTERACTIVE — INSERT 23, 7, 17, 24 INTO AN EMPTY HEAP

STEP 1 OF 5

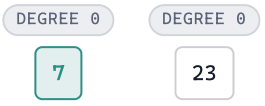
EMPTY HEAP

Start with an empty Fibonacci heap. INSERT just adds a new single-node tree to the root list — that is the whole operation.

STEP 2 OF 5

Insert 23. It becomes a lone tree; since the heap was empty, min[H] = 23.

STEP 3 OF 5

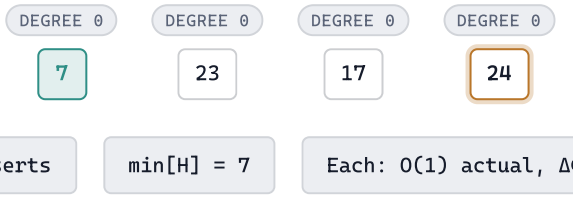


Insert 7. 7 < 23, so min[H] updates to 7.

STEP 4 OF 5

Insert 17. 17 > 7, so min[H] stays at 7.

STEP 5 OF 5



Insert 24. 24 > 7, min[H] stays at 7. Four single-node trees, zero structure beyond the root list — this is what "lazy" means: no consolidation ever happens on insert.

Remove the root, promote its children, then consolidate — every single check

A 9-node heap: root list = {3(min, children 18 & 9), 20, 15(child 25), 8(child 40), 12}.
Watch every array slot get checked, every collision, every link — nothing is skipped.

FIRST — WHERE DID THIS HEAP COME FROM? (IT COULD NOT HAVE COME FROM INSERTS)

The last slide left every inserted node sitting as its own degree-0 root, so the obvious objection to the heap above is: **how does anything have children yet?** The answer is the point of this whole slide. **Degree increases in exactly one place — consolidation, inside EXTRACT-MIN.** INSERT never links; UNION only concatenates root lists; DECREASE-KEY only ever *removes* children. So a heap holding a degree-2 root has already survived at least one extract-min.

Here is the smallest case that builds one, step by step. Insert **3, 18, 9, 20, 15** — five flat roots — then run a single EXTRACT-MIN and watch the degree array do the work:

INTERACTIVE — FIVE INSERTS, THEN ONE EXTRACT-MIN

STEP 1 OF 10

AN EMPTY HEAP — NOTHING AT ALL

Start empty. The question this answers: **how does any node ever acquire a child?** Watch closely — there is only one moment in the whole lecture where it happens.

STEP 2 OF 10

DEGREE 0 DEGREE 0 DEGREE 0 DEGREE 0 DEGREE 0

3 18 9 20 15

FIVE INSERTS → FIVE FLAT ROOTS, EVERY ONE OF DEGREE 0

inserts: 5 trees: 5 links so far: 0

INSERT **3, 18, 9, 20, 15**. Exactly as in the previous slide, each one becomes its own single-node tree in the root list. **Nothing is linked, nothing has a child.** This is as much structure as inserting can ever produce.

STEP 3 OF 10

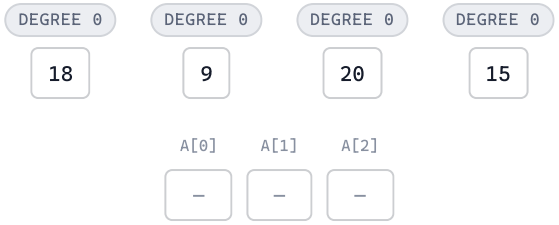
DEGREE 0 DEGREE 0 DEGREE 0 DEGREE 0 DEGREE 0

3 18 9 20 15

MIN[H] = 3 — EXTRACT-MIN BEGINS

Now call **EXTRACT-MIN**. The minimum is **3**; it has no children, so there is nothing to promote. Remove it and return it.

STEP 4 OF 10



FOUR DEGREE-0 ROOTS · THE DEGREE ARRAY STARTS EMPTY

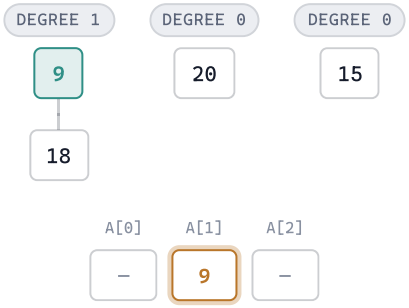
Consolidate. Walk the root list once, keeping an array `A[d]` = "the root of degree d seen so far". Whenever two roots share a degree they must be linked, because a Fibonacci heap may finish an extract-min with at most one tree per degree.

STEP 5 OF 10

A[0] WAS EMPTY — JUST PARK 18 THERE

First root: **18**, degree 0. `A[0]` is empty, so there is nothing to collide with — park it and move on.

STEP 6 OF 10

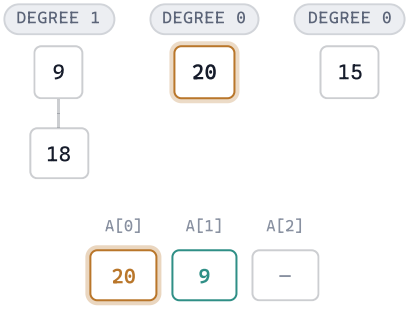


COLLISION AT `A[0]` → 9 < 18, SO 18 BECOMES 9'S CHILD — 9 IS NOW DEGREE 1

`links: 1` `first child ever created`

Next root: **9**, also degree 0 — and `A[0]` already holds 18. **Collision.** Link them, smaller key on top: 9 < 18, so **18 becomes 9's child**. That is the first child created in this entire lecture. 9 is now degree 1, so A[0] empties and 9 moves to `A[1]`.

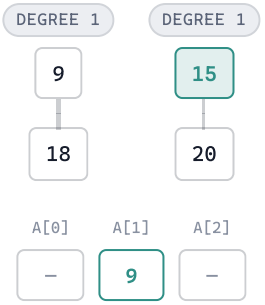
STEP 7 OF 10



A[0] IS FREE AGAIN — PARK 20

Next root: **20**, degree 0. `A[0]` was emptied by the last link, so 20 simply parks there. No collision.

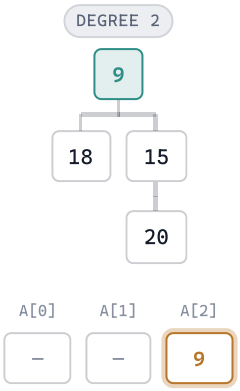
STEP 8 OF 10



COLLISION AT A[0] → 15 < 20, SO 20 BECOMES 15'S CHILD — BUT NOW TWO ROOTS HAVE DEGREE 1

Last root: **15**, degree 0, and `A[0]` holds 20. Collide and link: 15 < 20, so **20 becomes 15's child**. 15 is now degree 1 — and `A[1]` is already occupied by 9. **The carry propagates.**

STEP 9 OF 10

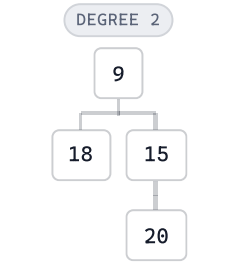


COLLISION AT A[1] → 9 < 15, SO 15 BECOMES 9'S CHILD — 9 IS NOW DEGREE 2

`links: 3` `trees: 4 → 1` `root degree: 2`

Second collision in a row: two degree-1 roots, 9 and 15. Link them — 9 < 15, so **15 (carrying 20) becomes 9's child**. 9 now has two children, 18 and 15: **degree 2**. A[1] empties, and 9 lands in `A[2]`.

STEP 10 OF 10



THE RESULT: ONE TREE, ROOT 9, DEGREE 2

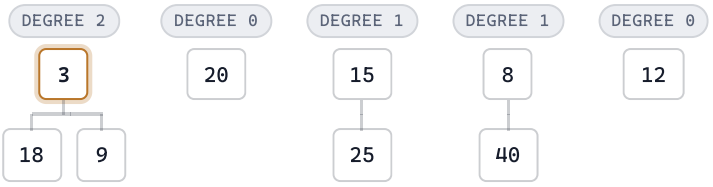
`5 inserts + 1 extract-min` `every degree now distinct` `depth reached: 2`

Done — and there is your degree-2 root. Structure was built by a **cascade of collisions**: each link bumps a degree, and the bumped tree can immediately collide one slot up. It is exactly the carrying you saw in Lecture 18, except a binomial heap paid it on *every insert*, while a Fibonacci heap defers the whole bill to this one moment.

One more clue in the heap below. Its roots are degree 2, 0, 1, 1, 0 — **two** of degree 1 and **two** of degree 0. Consolidation always leaves every degree *distinct*, so this heap is not sitting straight after one: an earlier extract-min built the degree-1 and degree-2 trees, and then further inserts piled fresh singletons on top. That is the normal life of a Fibonacci heap — **inserts flatten it, extract-min tidies it, inserts flatten it again.**

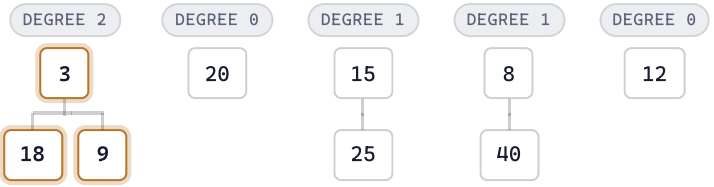
INTERACTIVE — EXTRACT-MIN(H)

STEP 1 OF 22



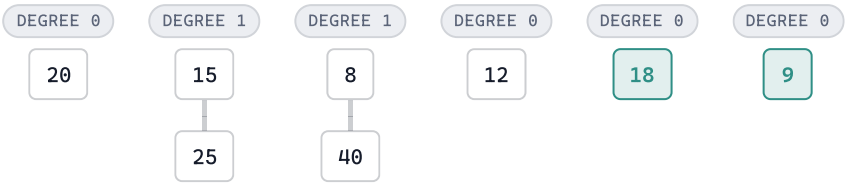
Starting heap: 5 trees, min[H] = 3 (children 18 and 9). EXTRACT-MIN has two steps: (1) remove the min and promote its children to the root list, (2) consolidate so no two roots share a degree.

STEP 2 OF 22



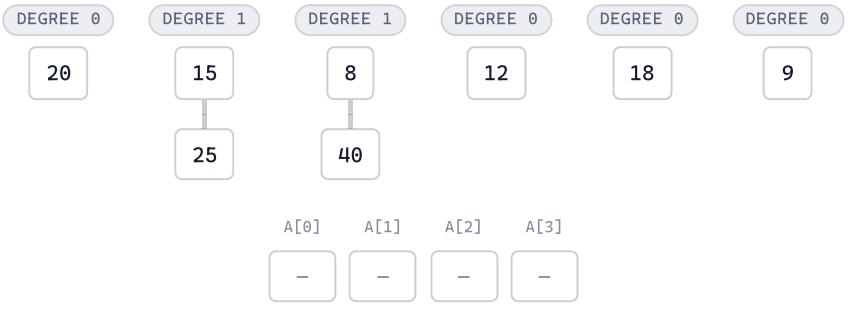
Step 1: remove node 3 (the minimum) — save its value to return. Its two children, 18 and 9, are cut loose.

STEP 3 OF 22



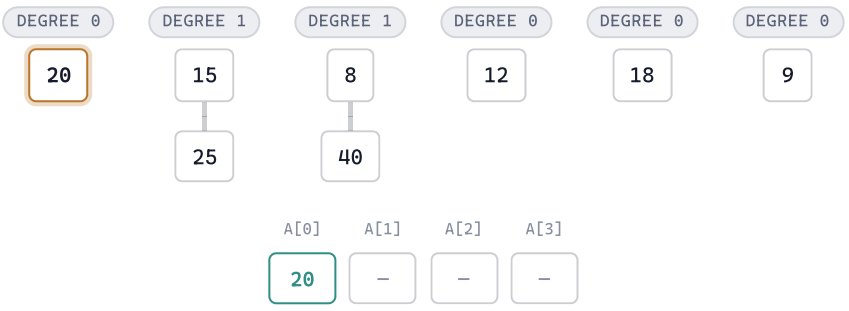
18 and 9 join the root list as new singleton trees. Root list now has 6 trees: 20, 15, 8, 12, 18, 9 — degrees 0,1,1,0,0,0. Time so far: O(D(n)) just to relink these children — no comparisons yet.

STEP 4 OF 22



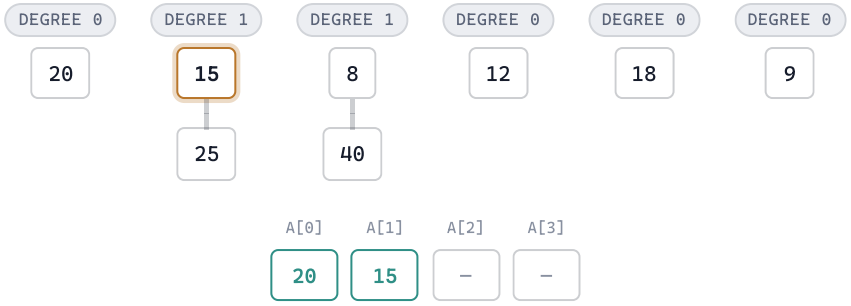
Step 2: consolidate. Walk the root list left to right, using an array A[0..3] indexed by degree — A[k] will hold "the root of degree k we've seen so far," if any.

STEP 5 OF 22



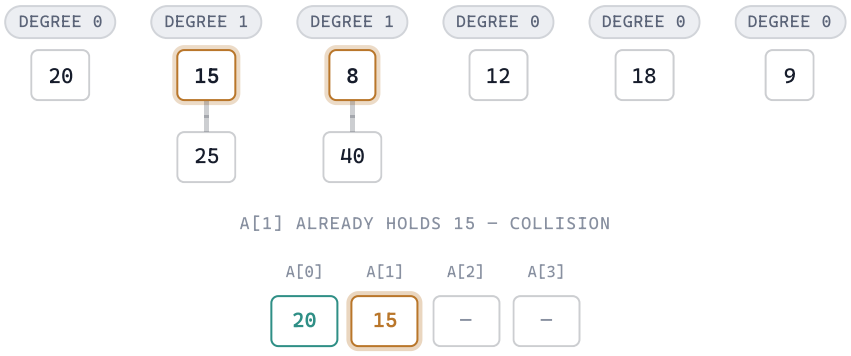
current = 20, degree 0. A[0] is empty — just record it there. No merge.

STEP 6 OF 22



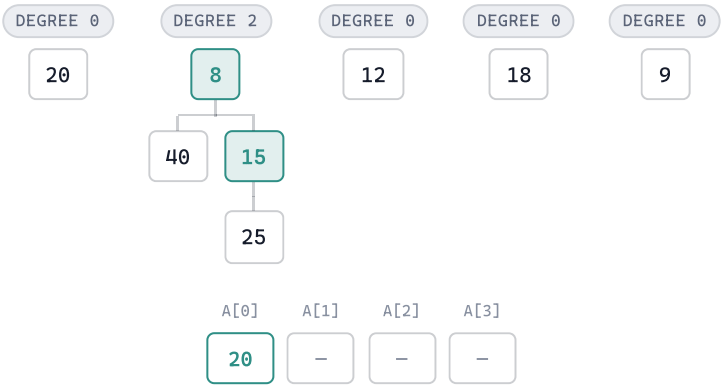
current = 15, degree 1. A[1] is empty — record it. No merge.

STEP 7 OF 22



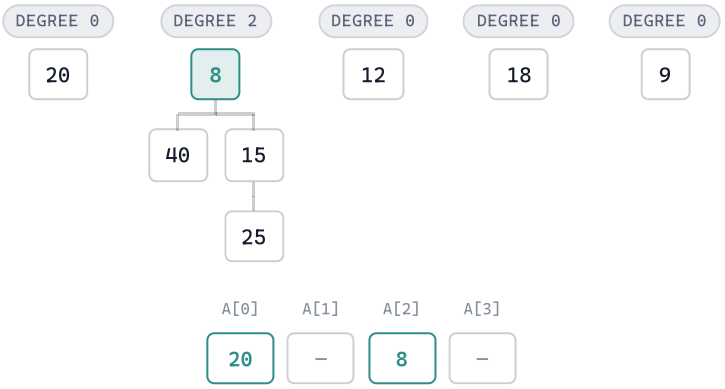
current = 8, degree 1 — but A[1] already holds 15! Same degree: compare keys. 8 < 15, so 8 wins.

STEP 8 OF 22



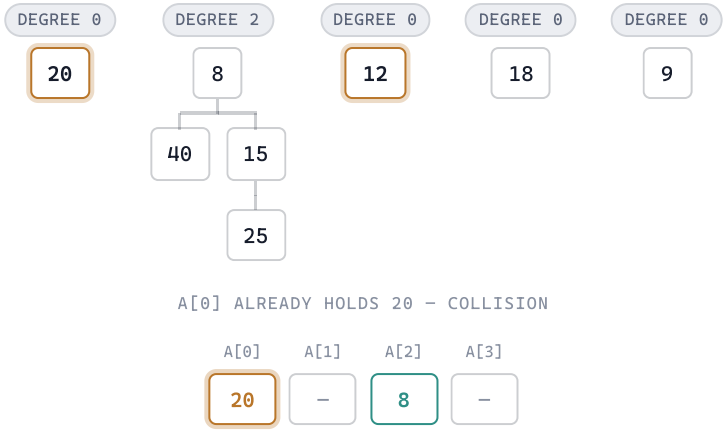
Link: 15 (still carrying its own child 25) becomes the new child of 8. 8 is now degree 2. Clear A[1] — that slot is free again.

STEP 9 OF 22



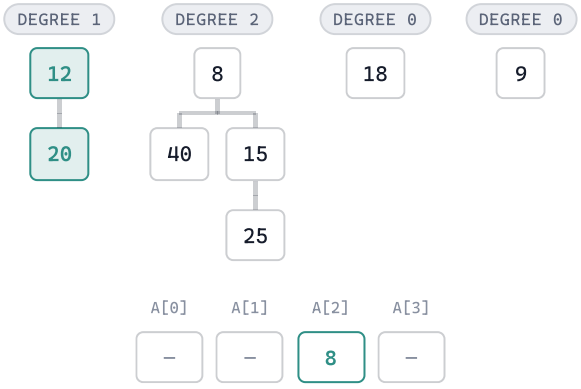
Re-check A[2] for this newly degree-2 node: empty. Place 8 there. Move on to the next original root.

STEP 10 OF 22



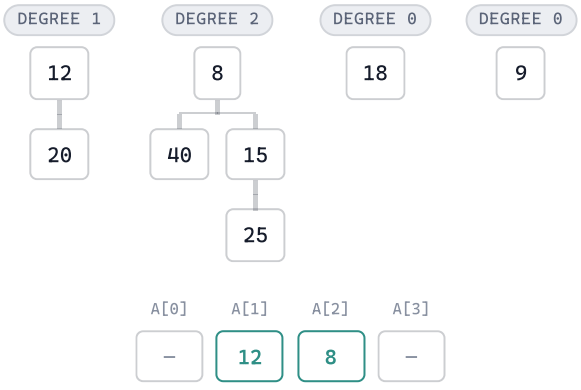
current = 12, degree 0 — A[0] already holds 20! Compare keys: 12 < 20, so 12 wins.

STEP 11 OF 22



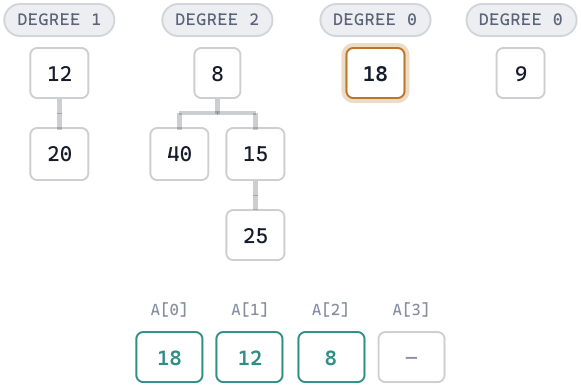
Link: 20 becomes the new child of 12. 12 is now degree 1. Clear A[0].

STEP 12 OF 22



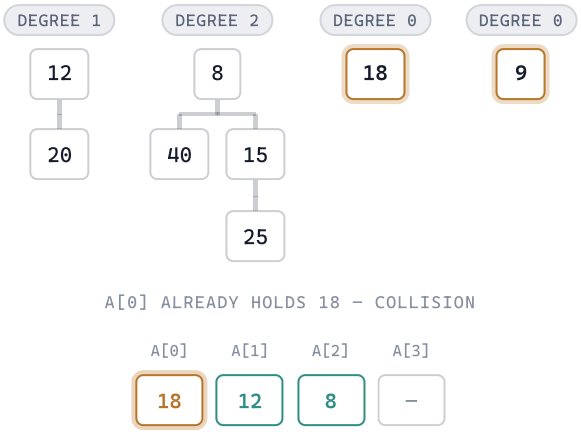
Re-check A[1]: empty (we cleared it earlier). Place 12 there.

STEP 13 OF 22



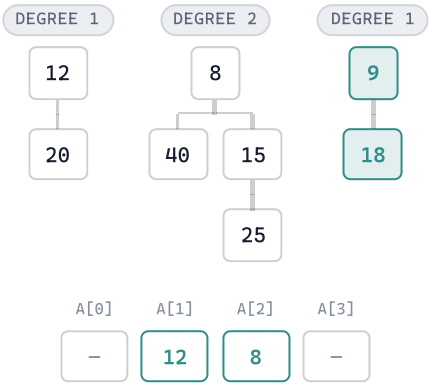
current = 18, degree 0. A[0] is empty — record it.

STEP 14 OF 22



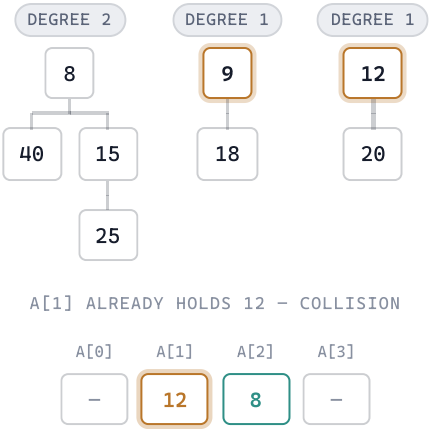
current = 9, degree 0 — A[0] already holds 18! Compare keys: 9 < 18, so 9 wins.

STEP 15 OF 22



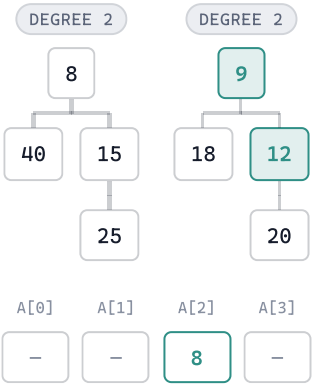
Link: 18 becomes the new child of 9. 9 is now degree 1. Clear A[0].

STEP 16 OF 22



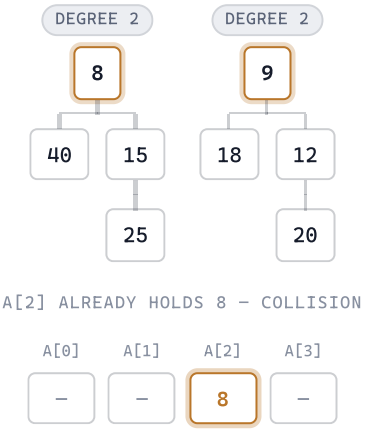
Re-check A[1] for this newly degree-1 node (9): it already holds 12! Compare keys: 9 < 12, so 9 wins again.

STEP 17 OF 22



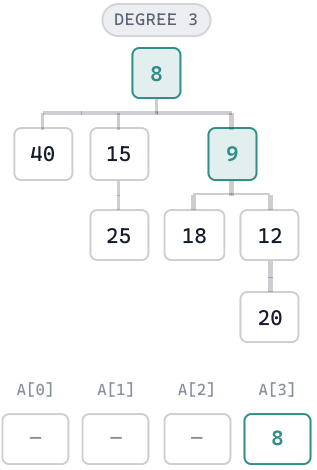
Link: 12 (still carrying its own child 20) becomes the new child of 9. 9 is now degree 2. Clear A[1].

STEP 18 OF 22



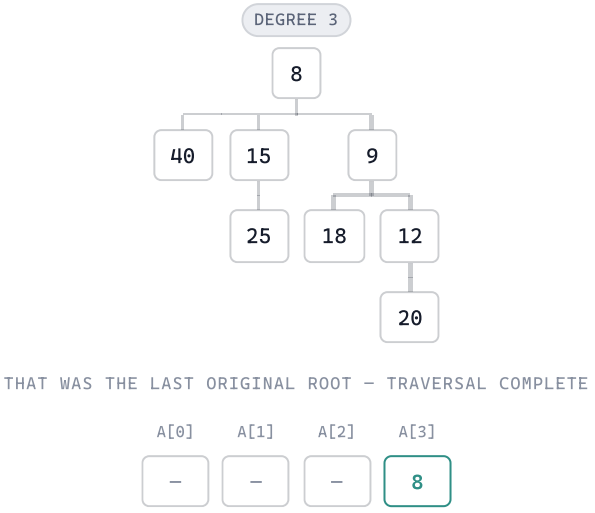
Re-check A[2] for this newly degree-2 node (9): it already holds 8! Compare keys: 8 < 9, so 8 wins.

STEP 19 OF 22



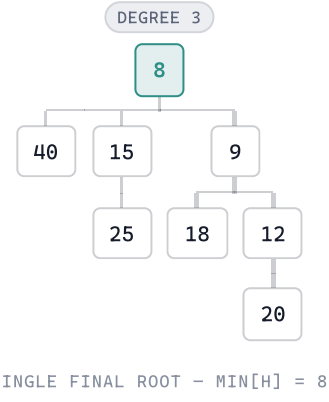
Link: 9 (carrying its own children 18 and 12→20) becomes the new child of 8. 8 is now degree 3. Clear A[2].

STEP 20 OF 22



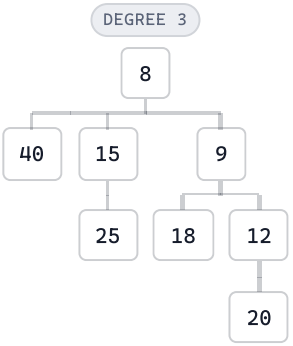
Re-check A[3]: empty. Place 8 there. That was the last of the 6 original roots (9) — the traversal is complete.

STEP 21 OF 22



Scan the array: only slot 3 is occupied, by 8. All 6 roots from Step 1 have cascaded into ONE tree, rooted at 8. New min[H] = 8 (the only root left).

STEP 22 OF 22



Returned: 3 | Links performed: 5 | Final trees: 1 (from 6) | min[H] = 8

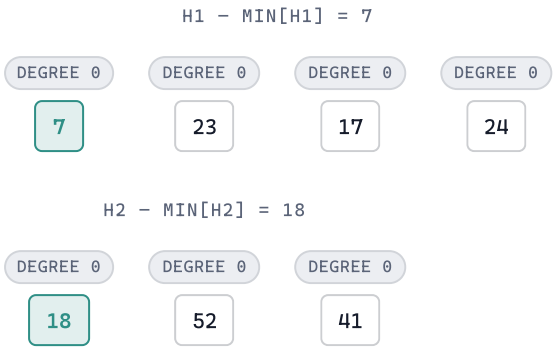
Done. EXTRACT-MIN returned 3. Five link events cascaded a 6-tree root list down to a single tree — exactly why the potential method charges O(D(n)+t(H)) actual work but the amortized cost comes out to O(D(n)) = O(log n): the shrinking tree count refunds almost all of that work.

Splice two circular lists together — O(1), regardless of size

H1 = {7(min), 23, 17, 24} — the heap built by the four inserts in L19·03. H2 = a fresh heap built from inserting 18, 52, 41 (min = 18).

INTERACTIVE — UNION(H1, H2)

STEP 1 OF 3

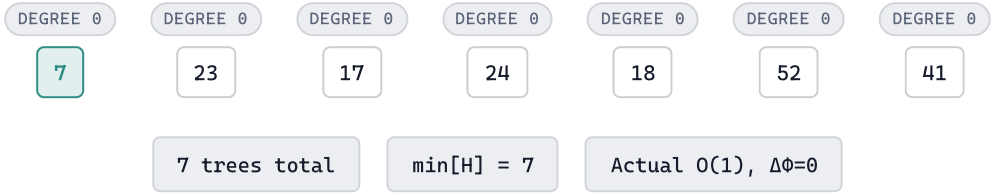


Two Fibonacci heaps: H1 = {7(min),23,17,24}, H2 = {18(min),52,41}. UNION must combine them.

STEP 2 OF 3

Splice H2's root list into H1's — both are circular doubly-linked lists, so this is a pointer-only operation: O(1), no matter how large either heap is.

STEP 3 OF 3



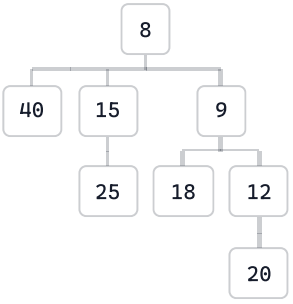
Compare the two old minimums (7 vs 18) and keep the smaller: min[H] = 7. Done — potential change is exactly 0 (same total trees and marks, just one heap now). This is the operation binary heaps (Θ(n)) and binomial heaps (O(log n)) cannot match.

No cut, a single cut, and a cascading cut — on the same tree

Continuing from EXTRACT-MIN's result: one tree, root 8, children 40, 15(child 25), 9(children 18, 12(child 20)). Every node starts unmarked — consolidation always unmarks a node when linking it.

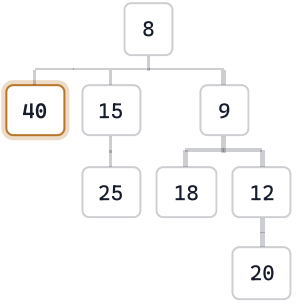
INTERACTIVE — DECREASE 40→35, THEN 18→6, THEN 12→2

STEP 1 OF 9



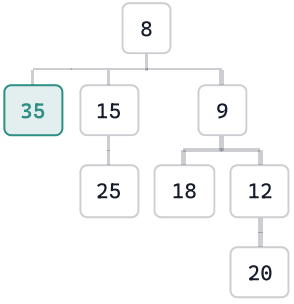
Starting from EXTRACT-MIN's result: one tree, root 8, children 40, 15(→25), 9(→18, 12(→20)). Every node is currently unmarked — linking during consolidation always unmarks. We'll decrease three different keys, one per case.

STEP 2 OF 9



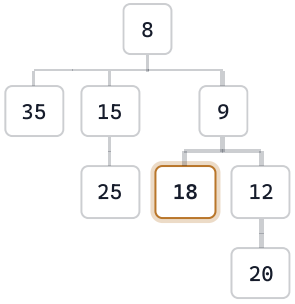
Case 0 — decrease 40 to 35. Compare the new value against its parent, 8.

STEP 3 OF 9



35 ≥ 8, so the min-heap property still holds. Just update the key in place — no cut, no mark change, nothing else happens. The cheapest possible case.

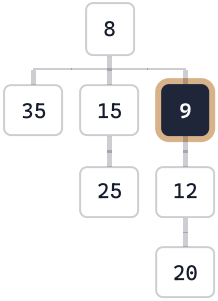
STEP 4 OF 9



Case 1 — decrease 18 to 6. Compare the new value against its parent, 9.

STEP 5 OF 9

TREE (9 NOW MARKED)



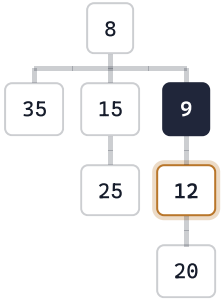
NEW ROOT

DEGREE 0

6 < 9 — the min-heap property is violated. CUT: remove 6 from 9's children, add it as a new root, and MARK 9 (its parent) — 9 has lost a child for the first time since it was last linked. min[H] updates to 6 (6 < the previous min, 8).

STEP 6 OF 9

TREE — 9 IS ALREADY MARKED



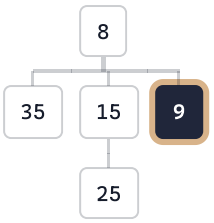
ROOT LIST

DEGREE 0

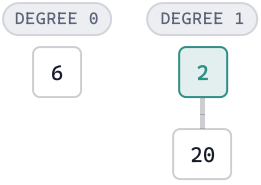
Case 2 — decrease 12 to 2. Compare the new value against its parent, 9. Notice: 9 is already marked from the last step.

STEP 7 OF 9

TREE — 9 JUST LOST ITS 2ND CHILD



ROOT LIST



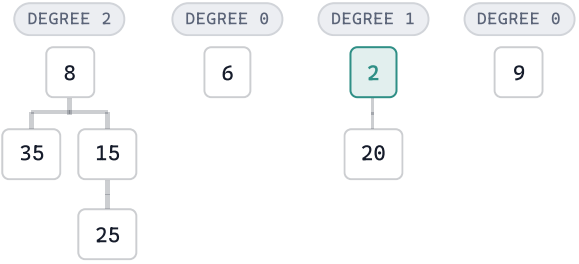
2 < 9 — cut 12 (still carrying its own child, 20) away from 9 and make it a new root. But 9 was already marked — this is 9's SECOND lost child. A marked node doesn't get marked again; it gets cut itself. Cascading cut begins.

STEP 8 OF 9

CASCADE REACHES 8 — 8 IS A ROOT, NO PARENT TO CUT, CASCADE STOPS

9 is cut from 8 as well, and unmarked on the way out — the cascade climbs one level. Now the rule applies to **8**, the node that has just lost a child: ordinarily it would be marked. But **8 is a root**, and **roots are never marked** — the algorithm discovers this by looking for a parent and finding none — so nothing happens and the **cascade stops here**. Final root list: 4 trees.

STEP 9 OF 9



3 decrease-keys	Cuts: 0, 1, 2 (cascading)	min[H] = 2	All: O(1) amortized

min[H] = 2 (2 < 6). All three decrease-keys were O(1) amortized — even the 2-level cascade in Case 2 — because Φ=t(H)+2m(H) drops sharply every time a cascade fires. We'll verify that arithmetic next.

Decrease to −∞, then extract-min — no new machinery required

Continuing from DECREASE-KEY's result: root list = {8(children 35, 15→25), 6, 2(child 20), 9}, min = 2. We'll delete node 15.

INTERACTIVE — DELETE(15)

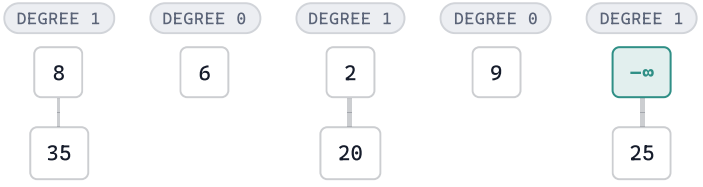
STEP 1 OF 10

DELETE(x) is just DECREASE-KEY(x, −∞) followed by EXTRACT-MIN. Let's delete node 15.

STEP 2 OF 10

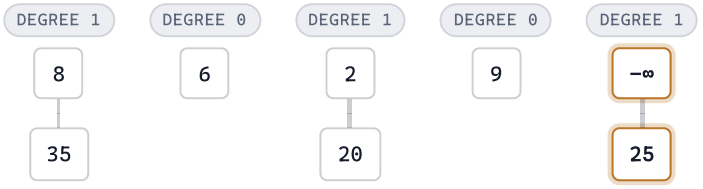
Step 1: decrease-key(15, −∞). Compare −∞ against 15's parent, 8 — it is smaller than everything, so this always triggers a cut.

STEP 3 OF 10



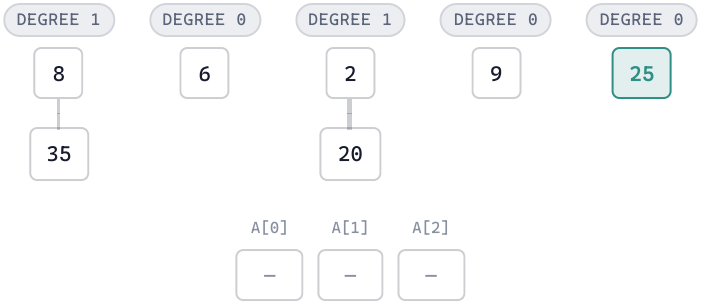
Cut 15 (still carrying its child 25) from 8, add it as a new root now holding the key −∞. min[H] becomes −∞ — guaranteed, since nothing can be smaller. 8 is now degree 1.

STEP 4 OF 10



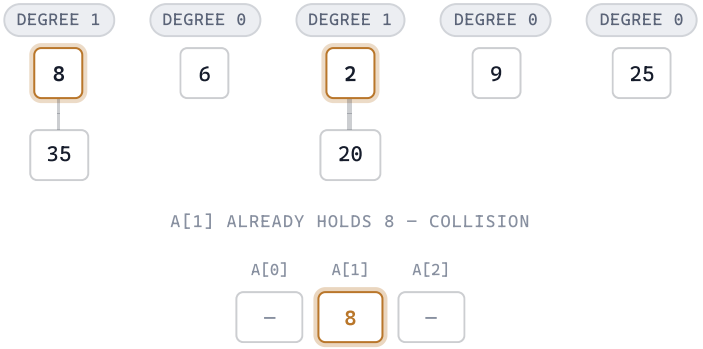
Step 2: EXTRACT-MIN. The current minimum is our −∞ node — remove it, and promote its one child, 25, to the root list.

STEP 5 OF 10



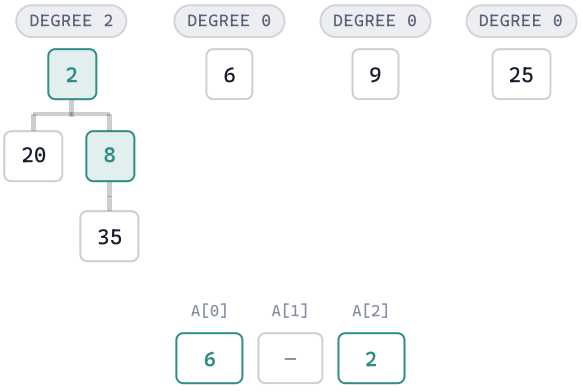
Root list now: 8(deg1), 6(deg0), 2(deg1), 9(deg0), 25(deg0). Two roots share degree 1 (8 and 2) — consolidation will merge them. Two roots also share degree 0 (6 and 9). Consolidate, left to right: 8, 6, 2, 9, 25.

STEP 6 OF 10



current = 8 (deg 1) placed in A[1] first. Then current = 6 (deg 0) placed in A[0]. Then current = 2 (deg 1): A[1] already holds 8! Compare keys: 2 < 8, so 2 wins.

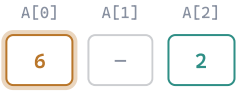
STEP 7 OF 10



Link: 8 (with its child 35) becomes a child of 2. 2 is now degree 2 — placed in A[2]. A[1] is cleared and empty again.

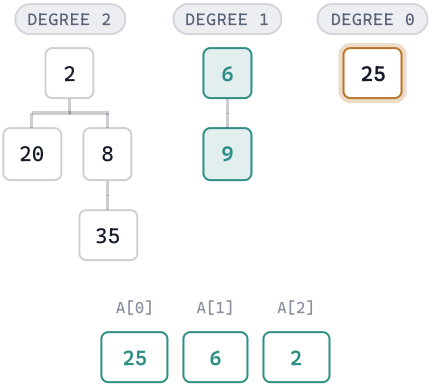
STEP 8 OF 10

A[0] ALREADY HOLDS 6 — COLLISION



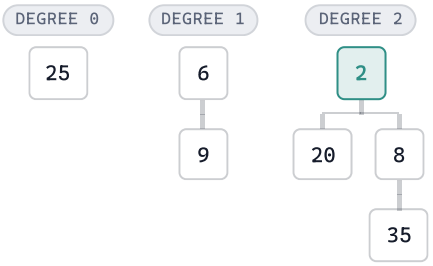
current = 9 (deg 0): A[0] already holds 6! Compare keys: 6 < 9, so 6 wins.

STEP 9 OF 10



Link: 9 becomes a child of 6. 6 is now degree 1 — placed in A[1]. Last root, 25 (deg 0): A[0] is empty (6 moved out) — place it there. Traversal complete.

STEP 10 OF 10



`3 final roots` `25, 6(→9), 2(→20,8(→35))` `min[H] = 2`

Final scan: three roots remain — 25, 6(→9), and 2(→20, 8(→35)). min[H] = 2, unchanged from before the delete (2 was already the smallest surviving key). DELETE cost: O(1) for the decrease-key step, plus O(D(n)) for extract-min — so DELETE is amortized O(log n) overall, same as EXTRACT-MIN.

Φ(H) = t(H) + 2·m(H) — trees measure looseness, marks measure debt

THE TWO INGREDIENTS

- **t(H)** = the number of trees currently in the root list. More trees means more looseness — more consolidation work waiting to happen.
- **m(H)** = the number of currently marked nodes. Each mark is a standing debt: "one cascading cut is already pre-authorised here."

Both are plain counts of things you can see in the heap — nothing hidden, nothing derived. The next slide explains the one part of this formula that always draws a question: **why marks are multiplied by 2.**

$\Phi(H_0) = 0$ for an empty heap, and $\Phi(H) \geq 0$ always — both $t(H)$ and $m(H)$ are counts, never negative.

A mark is a promise of exactly two units of future work

The 2 is not a safety margin, and it is not chosen to make the algebra tidy. It is the **size of the debt a mark represents** — and you can itemise that debt the moment the mark is set, long before anything is cut.

START FROM THE MARK, NOT FROM THE CUT

Setting a mark on a node **y** is a declaration: "*y has already lost one child. If it loses another, y itself will be cut out of its parent.*" That future cut is now guaranteed to cost two separate things, and both can be priced immediately:

THE PROMISED CUT WILL NEED	BECAUSE
1 unit — to perform the cut	splicing y out and moving it to the root list is O(1) work, and somebody has to pay for it
1 unit — to endow the new tree	y becomes a root, so t rises by 1 — and **every tree must carry one unit** for the consolidation EXTRACT-MIN will eventually do on it

Total liability: 2 units, incurred the instant the mark is set. So a mark must be worth 2 in Φ. Not 2 because the arithmetic works out — 2 because that is what it owes. The operation that sets the mark pays the deposit up front (that is the **+2** inside the +3 you saw on L19·09), and it sits at exactly the node where the work will later be needed.

CHECK IT: RUN ONE REAL CUT THROUGH THE FORMULA

When the promise is called in, the cut node **joins the root list** (one more tree) and its **mark is cleared** (one fewer mark). Apply Φ to the state before and to the state after:

$$\Phi_{\text{before}} = t + 2m$$
$$\Phi_{\text{after}} = (t + 1) + 2(m - 1)$$

and subtract:

$$\Delta\Phi = (t + 1) + 2(m - 1) - (t + 2m)$$
$$= t + 1 + 2m - 2 - t - 2m = \mathbf{-1}$$

Read that as the debt being settled: the mark releases **2**, the new tree keeps **1** of them as its own endowment, and the remaining **1** pays for the cutting work. Net −1 against an actual cost of 1, so the cut is **free** — exactly as the promise said it would be.

AND IF A MARK WERE WORTH ONLY 1?

Then the deposit covers only *one* of the two obligations. With Φ = t + m, the same cut gives:

$$\Phi_{\text{before}} = t + m$$
$$\Phi_{\text{after}} = (t + 1) + (m - 1) = t + m$$

$$\Delta\Phi = \mathbf{0}$$

The single unit released is entirely consumed by the new tree's endowment, leaving **nothing for the cutting work**. Every cut in a chain must then be paid out of pocket, so a DECREASE-KEY triggering **c** cascading cuts costs **c** — and c is unbounded. DECREASE-KEY would stop being O(1), and the whole reason for using a Fibonacci heap would be gone.

So 2 is forced, and 2 is enough. One unit per obligation, no more and no less — which is why the coefficient is exactly 2 and not 1, and equally not 3.

First, the case that happens almost every time: one cut, one mark, no cascade

Most DECREASE-KEY calls set off no cascade at all. Getting this case right first makes the general one easy, because the cascade turns out to be the same accounting repeated.

CASE A – THE KEY STILL FITS

The new key is still ≥ its parent's, or the node is already a root. Write the key in place, update `min[H]` if needed, and stop.

$$\text{actual} = 1 \quad \Delta\Phi = 0 \quad \text{amortized} = \mathbf{1}$$

No tree is created and no mark changes, so the potential does not move at all.

CASE B – ONE CUT, AND THE PARENT WAS UNMARKED

The key drops below the parent y, so the node is cut into the root list. Then CASCADING-CUT runs on y — and because **y is unmarked**, it simply marks y and returns. Nothing cascades.

$$\Phi_{\text{before}} = t + 2m$$
$$\Phi_{\text{after}} = (t + 1) + 2(m + 1)$$
$$\Delta\Phi = 1 + 2 = \mathbf{+3}$$

$$\text{actual} = 1 \quad \text{amortized} = 1 + 3 = \mathbf{4 = O(1)}$$

One more tree (**+1**) and one more mark (**+2**). The operation does one unit of work and **banks three** — a surcharge, not a refund.

WHAT THAT DEPOSIT IS FOR

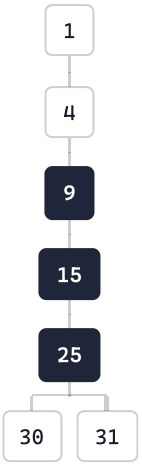
Those 3 units are not wasted. The **+1** covers the extra tree, which EXTRACT-MIN will later have to consolidate. The **+2** is attached to the mark just created — and as L19·08a showed, two units is exactly what one future cascading cut costs. **This operation is pre-paying for a cascade that has not happened yet**, at the very node that will trigger it. The next slide spends that money.

A longer cascade does more work — and releases proportionally more potential

Now the hard case. Let **c** be the number of CASCADING-CUT calls the operation makes. Case B above was simply c = 1. Here c can be any length, and the actual work grows with it — so the only question is whether the potential falls fast enough to keep up.

INTERACTIVE — COUNT C, THE CUTS, AND THE MARKS, ON ONE REAL CASCADE

STEP 1 OF 8

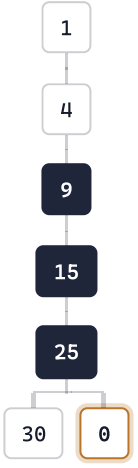


ONE TREE · 9, 15 AND 25 ARE MARKED · 4 IS NOT, AND 1 IS THE ROOT

t = 1 | m = 3 | Φ = t + 2m = 7

A chain with three marked nodes stacked above node 25, and one **unmarked** node (4) above them. Starting books: **t = 1** tree, **m = 3** marks, so **Φ = 7**. We will now run one DECREASE-KEY and count every quantity the formula mentions.

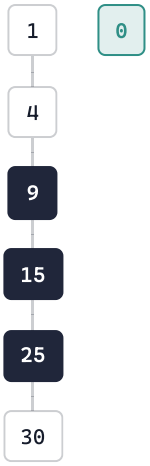
STEP 2 OF 8



KEY 31 LOWERED TO 0 IN PLACE — 0 < 25, SO IT MUST BE CUT

DECREASE-KEY(31, 0). The key is written in place and now violates min-heap order against its parent 25. Nothing has moved yet.

STEP 3 OF 8

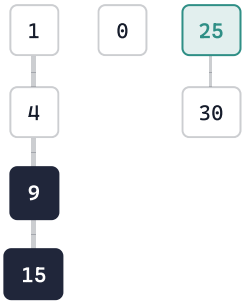


THE INITIAL CUT — THIS ONE IS *NOT* A CASCADING-CUT CALL

c (CASCADING-CUT calls): 0	cuts so far: 1	marks cleared: 0	marks set: 0

The initial cut. Node 0 is cut from 25 and joins the root list. Note carefully: **this cut is not counted in c**. c counts CASCADING-CUT calls, and the first of those has not happened yet — it is about to be made on 25.

STEP 4 OF 8

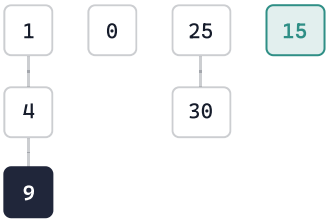


CALL 1: 25 WAS MARKED → CUT IT, AND CLEAR ITS MARK

c (CASCADING-CUT calls): 1	cuts so far: 2	marks cleared: 1	marks set: 0

CASCADING-CUT(25) — call 1. 25 has a parent and is **marked**, so it is cut loose and its mark is cleared on arrival in the root list. Then the procedure recurses on 25's old parent, 15.

STEP 5 OF 8

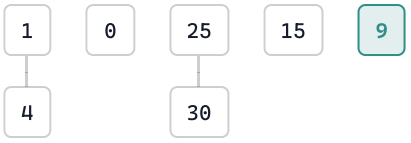


CALL 2: 15 WAS MARKED → CUT IT, CLEAR ITS MARK

c (CASCADING-CUT calls): 2	cuts so far: 3	marks cleared: 2	marks set: 0

CASCADING-CUT(15) — call 2. Same again: marked, so cut and unmarked. Recurse on 9. Notice the pattern — **every call that finds a marked node performs a cut and clears exactly one mark.**

STEP 6 OF 8

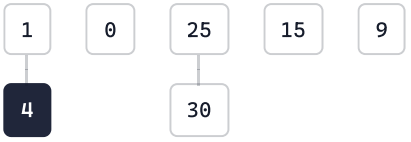


CALL 3: 9 WAS MARKED → CUT IT, CLEAR ITS MARK

c (CASCADING-CUT calls): 3	cuts so far: 4	marks cleared: 3	marks set: 0

CASCADING-CUT(9) — call 3. The third and last marked node in the chain. Cut, unmarked, and the procedure recurses on **4** — which is *not* marked.

STEP 7 OF 8

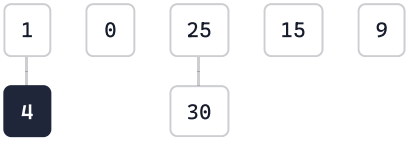


CALL 4: 4 IS A NON-ROOT AND **UNMARKED** → MARK IT AND **STOP**

c (CASCADING-CUT calls): 4	cuts so far: 4	marks cleared: 3	marks set: 1

CASCADING-CUT(4) — call 4. 4 has a parent (so it is not a root) but it is **unmarked**, so the rule takes the other branch: **set its mark and stop**. No cut this time. This is the call that ends the cascade, and it is the reason the formula carries a **+1 for a mark that gets created**.

STEP 8 OF 8



FINISHED · 7 NODES THROUGHOUT · T = 5, M = 1

c = 4	cuts = 4	cleared = 3	set = 1	t: 1 → 5	m: 3 → 1	Φ: 7 → 7

Now read the general formula off the tally. Of the **c = 4** calls, the first **c − 1 = 3** each cut a node and cleared a mark; the last one set a mark instead. So: **cuts = 1 initial + (c−1) = c = 4**, giving **t → t + c** (1 → 5 ✓). And **m → m − (c−1) + 1 = m − c + 2** (3 → 1 ✓). Hence **ΔΦ = c + 2(−c + 2) = 4 − c = 0**, and Φ indeed stayed at 7. Actual cost 4, amortized 4 + 0 = **4 = O(1)**. This run meets every bound with equality, so nothing here is slack.

ACTUAL COST GROWS WITH C

One O(1) cut for the decreased node, plus one O(1) cut for each cascading step:

$$\text{actual} = O(1 + c) = \mathbf{O(c)}$$

Taken alone this is bad news — c is not bounded by anything. A single DECREASE-KEY really can do a lot of work, as you saw when one call dissolved a whole tree.

POTENTIAL FALLS WITH C TOO

After the operation the heap holds **t + c** trees (the decreased node, plus one per cascading cut). Marks: every cascading cut but the last *clears* a mark, and the last call may *set* one — so at most **m − c + 2** remain.

$$\Delta\Phi \le (t + c) + 2(m - c + 2) - (t + 2m)$$
$$= c - 2c + 4 = \mathbf{4 - c}$$

$$\text{amortized} \le O(c) + (4 - c) = \mathbf{O(1)}$$

The c's cancel. Every extra cut adds one unit of work and releases one unit of potential, so the length of the cascade drops out of the answer entirely.

TWO SANITY CHECKS

- **The bound is tight at c = 1.** Substituting c = 1 gives $\Delta\Phi \le 3$ — exactly the +3 computed on the previous slide. The no-cascade case is not a separate rule; it is this formula at its smallest c.
- **Against the run in L19·02c:** that cascade made c = 4 calls. Trees went 4 → 8 = t + c ✓. Marks went 3 → 0, within the bound m − c + 2 = 1 ✓. And Φ fell 10 → 8, so $\Delta\Phi = -2$, comfortably inside 4 − c = 0 ✓.

Notice the sign flip: at c = 1 the operation **banks** potential (+3), and once c > 4 it **spends** it. That is the whole mechanism in one line — the cheap calls fund the expensive ones, and no single call is ever expensive on average.

The potential drop refunds almost all of the consolidation work

ACTUAL COST

Let **D(n)** = the largest degree any node can have in an n-node heap. **Step 1** promotes the minimum's children into the root list — there are at most D(n) of them, so O(D(n)).

Step 2 consolidates, which walks the root list once. Count how long that list is when the sweep begins:

$$\begin{aligned} &\mathbf{t(H)} \text{ roots at the start} \\ -\ &\mathbf{1} \text{ the minimum leaves} \\ +\ &\mathbf{D(n)} \text{ its children arrive} \\ &\text{————————————} \\ \leq\ &\mathbf{D(n) + t(H) - 1} \text{ roots} \end{aligned}$$

That −1 is simply the extracted node leaving. It is the value being returned, so it is no longer in the list to be walked over — its children take its place. Each remaining root is then checked in O(1), giving:

$$\text{actual cost} = O(D(n) + t(H))$$

The −1 disappears inside the O( ) and never affects the bound; it is worth seeing only so the count is honest. Verified in L19·04: 5 links across 6 initial roots, each O(1).

POTENTIAL CHANGE

Apply Φ to the heap before, and to the heap afterwards. Two facts pin down the "after" state: consolidation leaves **at most one tree per degree**, and degrees run 0, 1, …, D(n) — that is **D(n) + 1** slots, so t can be no larger than that. (Verified in L19·04: 6 roots collapsed to just 1.) And **marks can only fall** during extract-min, never rise: a marked node loses its mark if it is linked as somebody's child, and nothing here sets one.

$$\begin{aligned} \Phi_{\text{before}} &= t(H) + 2m(H) \\ \Phi_{\text{after}} &= t(H') + 2m(H') \\ &\leq (D(n) + 1) + 2m(H) \end{aligned}$$

Subtracting, the mark terms disappear:

$$\begin{aligned} \Delta\Phi &= \Phi_{\text{after}} - \Phi_{\text{before}} \\ &\leq [(D(n) + 1) + 2m(H)] \\ &\quad - [t(H) + 2m(H)] \\ &= \mathbf{(D(n) + 1) - t(H)} \end{aligned}$$

The 2m(H) terms cancel outright — which is why marks play no part in this analysis at all. They matter only for DECREASE-KEY, where cascading cuts change them. Here the whole argument is about trees: t collapses from however large it had grown down

to at most D(n)+1, and **that collapse is the refund**.

$$\text{amortized} = O(D(n)+t(H)) + (D(n)+1-t(H)) = \mathbf{O(D(n))}$$

The t(H) terms cancel — however long the root list had grown from lazy inserts and cuts, the potential those operations banked pays for sweeping it. What survives is D(n) alone, and since D(n) = O(log n), the amortized cost is **O(log n)**.

TWO CONSTANTS THAT LOOK ALIKE BUT MEAN DIFFERENT THINGS

Each card above carries a small constant beside D(n), and they are easy to confuse. They come from different places:

WHERE	EXPRESSION	WHAT THE CONSTANT MEANS
actual work	D(n) + t(H) **− 1**	the minimum **left** the root list, so there is one fewer root to sweep
potential afterwards	D(n) **+ 1**	how many roots can **survive**: one per degree, and degrees 0 … D(n) inclusive is D(n)+1 slots

The first says *one root is gone*; the second says *how many may remain*. Neither changes the asymptotics — but mixing them up makes the derivation look arbitrary, which it is not.

The best known amortized bounds — and a real cost to get them

ADVANTAGES

- **DECREASE-KEY in Θ(1) amortized** — the headline improvement over both binary and binomial heaps, verified today with a real cascading-cut example.
- **INSERT and UNION in Θ(1) amortized** — both are pure root-list operations, no consolidation.
- **EXTRACT-MIN and DELETE stay O(log n) amortized** — no regression versus binomial heaps, while everything else got strictly cheaper.
- These are the asymptotically best known bounds for a mergeable priority queue with decrease-key — exactly why graph algorithms like Dijkstra's cite Fibonacci heaps as the theoretical optimum.

LIMITATIONS

- **Large constant factors in practice.** The bookkeeping (parent/child/left/right/degree/mark per node, cascading-cut logic) often makes Fibonacci heaps *slower in practice* than a plain binary heap for the input sizes most real programs see — despite better asymptotics.
- **Genuinely intricate implementation.** Circular doubly-linked lists at every level, plus the cascading-cut recursion, are meaningfully harder to implement correctly than a binary heap's single sift rule or even a binomial heap's four link cases.
- **EXTRACT-MIN's O(D(n)) bound relies on D(n) = O(log n)** — a real theorem (proof omitted here, as in the source material), not an obvious fact; it depends on every tree of degree k having at least F_{k+2} nodes (the Fibonacci numbers — hence the name).
- **No benefit if decrease-key is rare.** If your workload is insert/extract-min only, a binary or binomial heap remains simpler with no real performance downside.

Binary vs. binomial vs. Fibonacci

OPERATION	BINARY HEAP (WORST-CASE)	BINOMIAL HEAP (WORST-CASE)	FIBONACCI HEAP (AMORTIZED)
MAKE-HEAP	Θ(1)	Θ(1)	Θ(1)
INSERT	Θ(log n)	O(log n)	Θ(1)
MINIMUM	Θ(1)	O(log n)	Θ(1)
EXTRACT-MIN	Θ(log n)	Θ(log n)	O(log n)
MERGE/UNION	Θ(n)	O(log n)	**Θ(1)**
DECREASE-KEY	**Θ(log n)**	**Θ(log n)**	**Θ(1)**
DELETE	Θ(log n)	Θ(log n)	O(log n)

n = number of nodes at the time of the operation. Every Fibonacci-heap number above was exercised today with real, hand-traced examples — not just asserted.

The theoretical benchmark for graph algorithms — with a practical asterisk

DIJKSTRA'S SHORTEST PATH

With a Fibonacci heap, Dijkstra's algorithm runs in O(E + V log V) — the E decrease-key calls (one per edge relaxed) cost O(1) amortized each, versus O(E log V) with a binary heap. This is the textbook headline result that motivates the whole structure.

PRIM'S MINIMUM SPANNING TREE

Same pattern: Prim's algorithm repeatedly decreases a vertex's "cheapest known connection" as better edges are found — exactly the decrease-key-heavy access pattern Fibonacci heaps are built for.

THE PRACTICAL ASTERISK

In practice, most production graph libraries use a plain binary heap or a simpler pairing heap (Lecture 20) instead — the constant-factor overhead of Fibonacci heaps' bookkeeping often outweighs the asymptotic win at realistic graph sizes. Fibonacci heaps remain essential for the theoretical result, even where they're rarely the implementation of choice.

Lazy now, tidy later — and the bill always comes out to O(1) or O(log n)

- **Φ(H) = t(H) + 2m(H)** turns "how messy is this heap right now" into a single number — trees measure looseness, marks measure pre-authorized cascading cuts.
- **INSERT and UNION are Θ(1)** — pure root-list operations, verified today with zero consolidation.
- **EXTRACT-MIN's consolidation** — verified today: 6 roots, 5 link events, collapsing to a single tree, with the potential drop refunding all but O(log n) of that work.
- **DECREASE-KEY is Θ(1) amortized even with a cascading cut** — verified today with a genuine 2-level cascade, where the marks-weighted-by-2 potential exactly cancels the extra cutting cost.
- **DELETE is just decrease-to-−∞ plus extract-min** — no new machinery, verified today end to end.

LECTURE 20 — NEXT

Pairing Heaps and Double-Ended Priority Queues

A structure almost as fast as Fibonacci heaps in theory, dramatically simpler in practice — and DEPQs, which support both extract-min AND extract-max.

HOMEWORK — BRING TO LECTURE 20

- Build a Fibonacci heap by inserting 5, 11, 2, 8. Union it with a second heap built from inserting 6, 14. Trace EXTRACT-MIN by hand, showing the degree array at every step.
- On the heap resulting from the previous question, find a node with a grandparent and force a 2-level cascading cut via two decrease-key calls. Show the mark bits changing at each step.
- In 3–4 sentences: why does the potential function weight marks by 2 and not by 1? What would break if it were 1?
- In 3–4 sentences: explain, without any formulas, why Dijkstra's algorithm benefits more from a fast decrease-key than from a fast extract-min.

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

> *Next: Lecture 20 — Pairing Heaps and Double-Ended Priority Queues.*
