Binomial Heaps

Lecture 16's binary heap could insert and extract in O(log n) — but try to merge two of them together, and it collapses to O(n). Today: a heap built entirely out of forests of trees whose shapes come straight from Pascal's triangle, designed to merge in O(log n) flat.

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

~75 minutes
Session outcome: explain heap merging and binomial tree structures

~105 MIN

Today, part by part

The one operation binary heaps were never built for: merge

REAL-LIFE EXAMPLE — CONTINUING LECTURE 17'S ER

A regional disaster strikes. Hospital A's emergency queue and Hospital B's emergency queue — each already a valid, fully-built priority queue with hundreds of waiting patients — must become **one single queue**, right now, so a combined response team can serve strictly by urgency. Neither hospital has time to rebuild a queue from scratch.

WHY A BINARY HEAP FAILS HERE

Lecture 16's binary heap has no efficient UNION. The only way to combine two binary heaps is to concatenate their arrays and re-run BUILD-MAX-HEAP (or BUILD-MIN-HEAP) from scratch — **Θ(n)**, where n is the combined size. For two mid-sized hospitals, that could mean re-heapifying thousands of records during the exact moment speed matters most.

WHAT WE ACTUALLY NEED

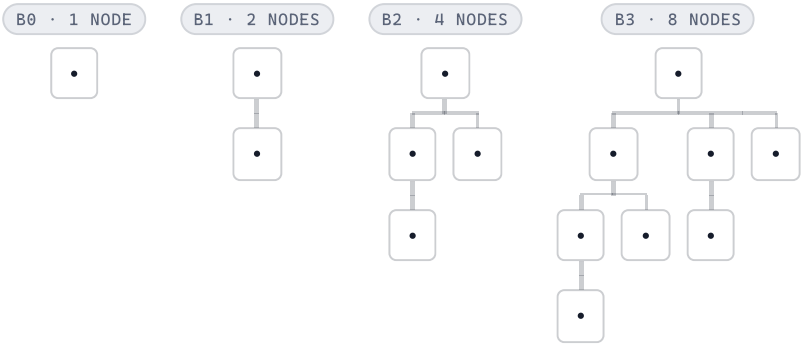
A **mergeable heap**: every operation a binary heap already supports (INSERT, MINIMUM, EXTRACT-MIN), plus a genuinely fast **UNION**. A binomial heap delivers UNION in **O(log n)** — by giving up the single-tree shape of a binary heap in favor of a small forest of trees with a very particular structure.

Before any definition — look at the first four, and how one is built

Everything in this lecture is written in terms of $\mathbf{B_k}$. Four things to fix in your head before any definition:

- **k is the tree's *order*** — a shape label, and it fixes the size completely.
- **It is an exponent, not a count.** B_k holds $\mathbf{2^k}$ nodes — so B_3 holds **8**, not 3.
- **It is stored, not just notation.** In code the order lives as the **degree of the root** — how many children it has.
- **That is what UNION reads.** Comparing root degrees is exactly how it later decides which two trees to link.

THE FIRST FOUR, DRAWN TO THE SAME SCALE

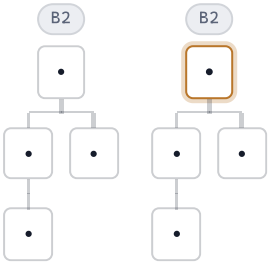


ORDER K = HEIGHT = NUMBER OF CHILDREN OF THE ROOT · NODE COUNT = 2^K

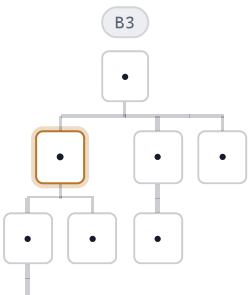
HOW A B_K IS MADE: LINK TWO B_{K-1}'S

Look again at the gallery: each tree is **two copies of the one before it**, one hooked under the other's root. That is the only rule there is — here it is, applied once to build that B_3:

BEFORE — TWO SEPARATE B2 TREES



AFTER — ONE B3, 8 NODES, HEIGHT 3



- **Nothing is rebuilt.** One root becomes a child of the other, bringing its subtree along — so a link is **O(1)**.
- **Every algorithm here is built from it.** UNION, INSERT and EXTRACT-MIN do nothing else.
- **Which root ends up on top? Right now, either.** No keys yet, so both choices give the identical shape. From L18·04 the heap property decides, and the **smaller key** stays on top.

READ THE ORDER K OFF THE PICTURE, THREE WAYS

ORDER	NODES	HEIGHT	CHILDREN OF THE ROOT
$\mathbf{B_0}$	$1 = 2^0$	0	0
$\mathbf{B_1}$	$2 = 2^1$	1	1
$\mathbf{B_2}$	$4 = 2^2$	2	2
$\mathbf{B_3}$	$8 = 2^3$	3	3

Every column reads **k** straight off the tree — nodes 2^k, height k, root children k. Spot any one and you know which tree you are looking at.

TWO THINGS B_K IS *NOT*

- **Not a binary tree.** The root of B_3 has three children, and nothing limits a node to two. Binomial trees are general trees — that surprise catches people who arrive straight from Lecture 16.
- **Not a heap by itself.** B_k is only a *shape*. The heap ordering (every parent ≤ its children) is an extra condition laid on top, which is what makes it a binomial *heap* later in L18·04.

You now know what B_k looks like and how one is built. The next slide writes that same rule down formally and works out what follows from it.

Defined by one rule: link two B_{k-1} trees together

Same rule you just watched on the previous slide — now written down properly, so we can work out what it forces to be true. Everything below is still **shape only**: no keys, no heap ordering.

THE RECURSIVE DEFINITION

B_0 is a single node. **B_k** is formed by taking **two copies of B_{k-1}** and *linking* them: the root of one becomes the new leftmost child of the root of the other. That's the entire definition — every other property falls out of it.

WHAT FALLS OUT OF IT (VERIFIED, NOT JUST ASSERTED)

- B_k has exactly $\mathbf{2^k}$ nodes (two copies of a 2^{k-1}-node tree).
- B_k has height exactly **k**.
- B_k has exactly **C(k,i)** nodes at depth i — the binomial coefficients, which is exactly where the name comes from.
- The root of B_k has degree **k**, and — left to right — its children are the roots of B_{k-1}, B_{k-2}, …, B_0.

WHY "BINOMIAL" — CHECK ROW 3 AGAINST PASCAL'S TRIANGLE

Count B_3's nodes by depth back in the L18·02 gallery: `1, 3, 3, 1`. Those are exactly C(3,0), C(3,1), C(3,2), C(3,3) — a row of Pascal's triangle. The same holds at every order:

TREE	NODES AT DEPTH 0, 1, 2, 3	TOTAL
B_1	`1, 1`	2
B_2	`1, 2, 1`	4
B_3	`1, 3, 3, 1`	8

The rows sum to 2^k, exactly as $\sum_i C(k,i) = 2^k$ says they must. The name is descriptive, not decorative.

Watch the recursive definition build itself

You have seen *one* link close up (L18·02) and the rule stated formally (L18·02a). Now watch the whole ladder built from nothing: $B_0 \to B_1 \to B_2 \to B_3$, where **every step is exactly one link** — take two identical trees, make one root the new leftmost child of the other.

SHAPE ONLY — NO HEAP PROPERTY YET

Everything from L18·02 to here is about **shape alone**. That is why every node is drawn as a bare •: **there are no keys in these trees**, so there is nothing to compare and no heap ordering to check or maintain. B_k at this point is a *pattern*, not a data structure holding anything. When two trees are linked below, "which root stays on top" is genuinely arbitrary — both choices give the identical shape. Keys, the heap property (*every parent ≤ its children*), and therefore a real rule for who wins a link, all arrive in **L18·04**, when these shapes start carrying data.

INTERACTIVE — $B_0 \to B_1 \to B_2 \to B_3$

STEP 1 OF 8

B0: A SINGLE NODE, HEIGHT 0, DEGREE 0

B0 — the base case. A single node. Height 0, root degree 0, $1 = 2^0$ node.

STEP 2 OF 8

TWO COPIES OF B0 — P WILL STAY ON TOP, Q WILL BE ABSORBED

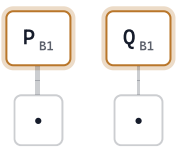
To build **B1**: take two copies of B0. Name their roots **P** and **Q** so you can follow them. **Which one stays on top?** For the shape alone it makes no difference — the two results are identical trees. (In a real binomial heap the *smaller key* wins; you will see that decide it in L18·06.) Here we keep **P**.

STEP 3 OF 8

B1 — Q IS NOW P'S CHILD

Link. Q's root becomes a child of P. Count P's children: it had **0**, gained Q, so now **1** — degree 1. Total nodes $1 + 1 = 2 = 2^1$. That is **B1**.

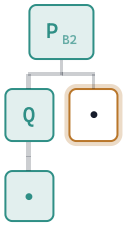
STEP 4 OF 8



TWO COPIES OF B1 — EACH ROOT ALREADY HAS 1 CHILD

To build **B2**: two copies of B1. Note where we are starting from — **P already has one child of its own**, and so does Q. Keep an eye on that child; it does not go anywhere.

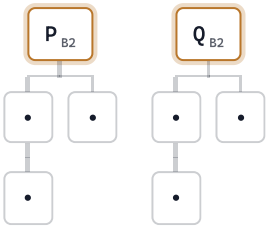
STEP 5 OF 8



B2 — LEFTMOST CHILD IS Q (A WHOLE B1); P'S ORIGINAL CHILD IS STILL THERE, ON THE RIGHT

Link. Q becomes P's **leftmost** child — and Q *brings its own subtree with it* (highlighted). Nothing is copied or rebuilt. Now count P's children: **1 original** (amber, on the right) **+ Q** = **2** → degree 2. Left to right they are the roots of a **B1** and a **B0**. Total nodes 4 = 2^2. That is **B2**.

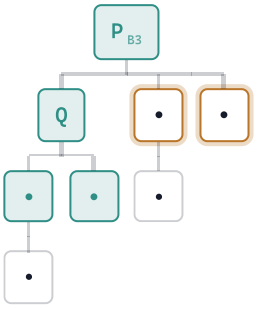
STEP 6 OF 8



TWO COPIES OF B2 — EACH ROOT ALREADY HAS 2 CHILDREN

To build **B3**: two copies of B2. Again, look at the starting point: **P already has two children** (the roots of a B1 and a B0). This is the step where the "where did the third child come from?" question gets answered.

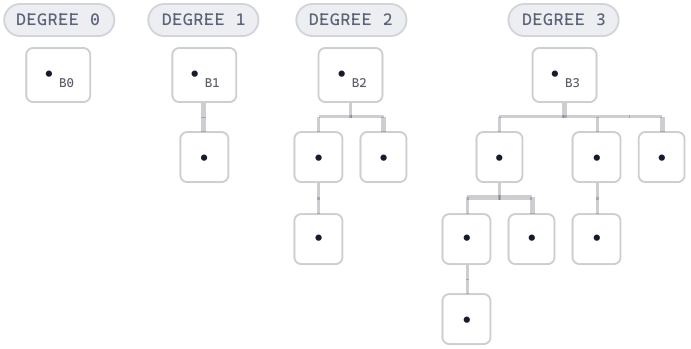
STEP 7 OF 8



B3 — Q (A WHOLE B2) JOINS P'S TWO EXISTING CHILDREN

Link. Q becomes P's new leftmost child, dragging its entire 4-node subtree along (highlighted). **P's two original children never moved** (amber). So the count is **2 original + Q = 3** children → degree 3 — that is where the third child comes from, not from anywhere new. Left to right the three are the roots of **B2, B1, B0**. Total nodes 4 + 4 = 8 = 2^3. That is **B3**.

STEP 8 OF 8



`nodes = $2^k$` `height = k` `depth-i count = C(k,i)` `root degree = k`

B0 through B3, side by side. Every link added exactly one child to the surviving root, which is why **degree k** and **order k** are the same number. Node count doubles each step (1, 2, 4, 8) because each Bk is literally two copies of Bk-1 — and that doubling is why the heap's at-most-one-tree-per-degree rule mirrors binary digits: degree k holds 2^k nodes, just as binary place value k is worth 2^k.

A forest of binomial trees — one tree per 1-bit of n

TWO PROPERTIES, TOGETHER

- **Min-heap-ordered:** every binomial tree in the heap obeys the min-heap property — a node's key is never less than its parent's.
- **At most one tree per degree:** for any k, the heap contains at most one B_k.

THE BINARY-NUMBER CONNECTION

Write n in binary. Bit i is 1 exactly when the heap contains a B_i. A heap of **13** patients (binary `1101`) is a B_3 (8 nodes) + a B_2 (4 nodes) + a B_0 (1 node) — never two trees of the same degree, exactly like a binary digit is never anything but 0 or 1.

ONE THING THIS FOREST CANNOT DO: LIVE IN AN ARRAY

Lecture 16's binary heap needed *no pointers at all* — children of index i sat at 2i and 2i+1. That trick only worked because the tree was **complete**, so the indices were dense with no gaps. A binomial heap is a forest of differently-shaped trees whose roots have wildly different degrees, so no index formula exists. It has to be built from real pointers — which is what the next slide shows, node by node.

Three pointers per node — whether it has 1 child or 20

The root of B_k has **k** children, and k changes from tree to tree. Giving every node a variable-length array of children would mean a separate allocation per node. The standard fix — **left-child / right-sibling** — avoids that entirely: each node stores only its *leftmost* child plus its *next sibling*, so a node's children become a linked list and every node is the same fixed size.

WHAT ONE NODE HOLDS

`parent`	→ up to its parent
`key`	the priority value
`degree`	how many children it has
`child`	→ its **leftmost** child only
`sibling`	→ the **next** sibling to its right

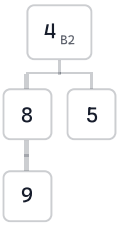
Five fields, fixed size, forever. A root with 20 children still stores exactly one `child` pointer — the other 19 are reached by hopping along `sibling`.

SO "VISIT ALL CHILDREN OF X" IS A WALK, NOT AN INDEX

```
y = x.child
while y ≠ NIL:
    visit y
    y = y.sibling
```

Compare with Lecture 16, where the same job was `A[2i]` and `A[2i+1]` — two array reads, no pointer chasing. This is the concrete cost of giving up the single complete tree: more memory per node, and worse cache locality.

A REAL B_2, DRAWN AS A TREE AND AS THE POINTERS THAT ACTUALLY EXIST



A B2 HOLDING KEYS 4, 8, 5, 9

A min-heap-ordered B_2: every parent's key is ≤ its children's (4 ≤ 8, 4 ≤ 5, 8 ≤ 9). The picture shows **4** with two children — but no node ever stores "two children".

What is really in memory:

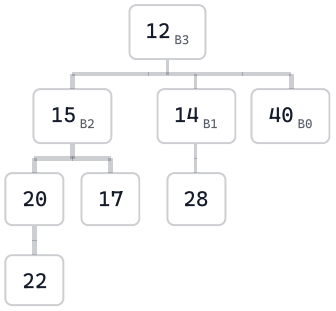
NODE	PARENT	DEGREE	CHILD	SIBLING
4 (root)	NIL	2	**8**	next root

8	4	1	**9**	**5**
5	4	0	NIL	NIL
9	8	0	NIL	NIL

Read row 1: node 4 points only at **8**. To find its second child you go `4.child = 8`, then `8.sibling = 5`. The children of 4 are the chain **8** → **5** → **NIL**.

INSIDE ONE TREE: THE CHILDREN COME OUT IN **DECREASING** DEGREE

The B_2 above has only two children, so the pattern is easy to miss. Here is a **B_3**, where it is unmistakable — read its root's children left to right:



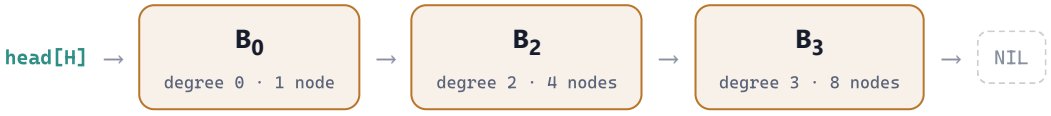
A B3 — ITS ROOT'S CHILDREN ARE B2, B1, B0

Following `12.child` and then the `sibling` pointers:

Degrees **2, 1, 0** — strictly decreasing, and every order below 3 present exactly once. That is the L18·02a property, now visible as an actual pointer chain.

THE ROOT LIST — THE HEAP ITSELF IS JUST A POINTER TO THIS CHAIN

The trees are not held in an array either. Their roots are strung together with the very same `sibling` field, forming one linked list. Here is the n = 13 heap from the last slide ($1101_2 \rightarrow B_3 + B_2 + B_0$):



Degrees **strictly increase** along the root list — 0, 2, 3 — with degree 1 simply absent, because bit 1 of 13 is 0. The whole heap is a single pointer, `head[H]`, to the front of this chain. Total nodes 1 + 4 + 8 = **13**.

THE GOTCHA: THE TWO LISTS ARE SORTED *OPPOSITE* WAYS

LIST	ORDER	EXAMPLE
a node's **child** list	**decreasing** degree	B_3's root → B_2, B_1, B_0

the **root** list	**increasing** degree	B_0, B_2, B_3

This is not an inconsistency — each ordering is chosen for the operation that depends on it, as below.

NEITHER ORDER IS ARBITRARY

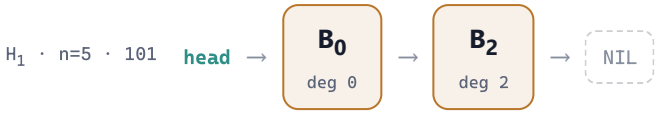
Each direction is chosen for the one operation that depends on it: the **increasing** root list is what lets UNION run in a single linear pass, and the **decreasing** child list is what lets EXTRACT-MIN hand back a legal heap without any repair work. Both are worth seeing drawn out — that is the next slide.

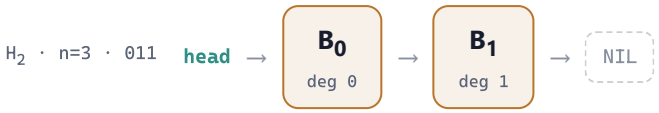
Two orderings, two operations, no coincidence

Sorting choices in a data structure are almost never aesthetic. Here each one exists to make a specific operation cheap — and if you flipped either, that operation would lose its O(log n).

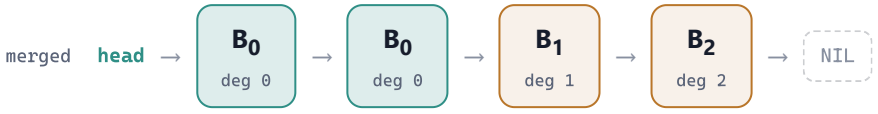
1 · ROOT LIST INCREASING — SO TWO HEAPS MERGE IN ONE PASS

Merge a heap of **5** patients with one of **3**. In binary that is `101 + 011`, and the root lists are already sorted by degree:





walk both lists once, always taking the smaller degree — the merge step of mergesort



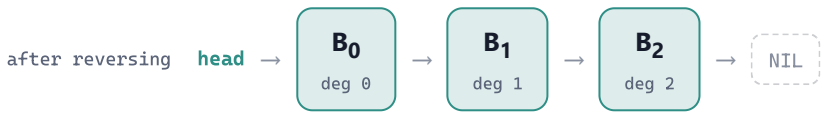
Here is the whole payoff: because both inputs were sorted, the two degree-0 trees came out **next to each other** (highlighted). Duplicates are always neighbours, so the linking sweep never has to *search* for a partner — it just checks the node in front of it. Two B_0's link into a B_1, which collides with the B_1 already there and links into a B_2, which collides with the B_2 and links into a $\mathbf{B_3}$. Final: 5 + 3 = 8 = `1000` — one tree, exactly as binary addition predicts.

Flip the ordering and this dies. The root list is a plain linked list — no random access, no binary search — so finding anything in it means *walking* it. What saves us is that the list is **short**: a heap on n nodes has at most ⌊lg n⌋+1 trees, so the list is only **O(log n)** long. Sorted, equal degrees are adjacent and each partner is found in **O(1)** — just check `x.sibling`. Unsorted, a partner could be anywhere, so each one costs a full **O(log n)** walk — once per root, and UNION degrades to **O(log² n)**.

2 · CHILD LIST DECREASING — SO A DELETED ROOT LEAVES A HEAP BEHIND

EXTRACT-MIN removes a whole root — specifically the root holding the **smallest key**, which has to be found by scanning the root list, since that list is ordered by degree and says nothing about keys. Whichever tree wins, the same thing happens to its children. Take a B_3 as the illustration: remove its root and you are left holding its children, which by the definition in L18·02a are stored in **decreasing** order:

decreasing — **not** a legal root list. Reverse the chain (one pass, O(log n) pointers)



That is now a **legal binomial heap** — increasing degrees, one tree per order — obtained by nothing but reversing a linked list. No rebuilding, no re-heapifying. It can be UNION-ed straight back into what remains.

CHECK THE ARITHMETIC — IT HAD TO COME OUT THIS WAY

Removing the root of a B_3 leaves `8 − 1 = 7` nodes, and the leftovers are $B_2 + B_1 + B_0$ = `4 + 2 + 1 = 7`. ✓

In binary, **7 = 111** — *every* bit set. That is not a coincidence: $2^k - 1$ is always k ones, which says the leftovers must be exactly **one tree of every order** from k−1 down to 0. This is the same "no gaps inside a tree" property from L18·02a, seen from the other side — and it is precisely why the leftovers are always a valid heap.

THE TWO ORDERINGS, SIDE BY SIDE

LIST	ORDER	WHAT IT BUYS — AND WHAT FLIPPING IT WOULD COST
root list	increasing	UNION merges in one linear pass, duplicates landing adjacent. Flipped: every link has to be searched for → $O(\log^2 n)$.
child list	decreasing	EXTRACT-MIN reverses once and holds a legal heap. Flipped: an extra sort or reversal on every extract.

You will watch both of these run for real: the merge-and-carry sweep in **L18·06**, and the detach-reverse-union in **L18·08**.

Where these queues come from: one patient at a time

The next slide merges two hospital queues. Neither one appeared fully formed — each was built by ordinary arrivals, one INSERT at a time. Watch Hospital A's queue grow from empty, and notice that **every arrival is the same two moves**: wrap the patient in a one-node heap, then fold it in, linking whenever two trees of equal degree meet.

INTERACTIVE — HOSPITAL A TAKES THREE PATIENTS: 2, THEN 6, THEN 9

STEP 1 OF 8

head[H] → NIL

n = 0 | binary 0 | 0 trees

Hospital A opens with an **empty** queue — the heap is just a head pointer at NIL. Three patients are about to arrive, in the order **2**, **6**, **9** (smaller number = more urgent).

STEP 2 OF 8

ARRIVING: PATIENT 2

DEGREE 0

2_{B0}

WRAPPED AS A ONE-NODE HEAP

H — STILL EMPTY

head[H] → NIL

Every INSERT starts identically: wrap the new patient in a **singleton heap**, a lone B_0. Now the job is just to fold that one-tree heap into H.

STEP 3 OF 8

DEGREE 0

2_{B0}

H AFTER THE FIRST ARRIVAL

n = 1 | binary 1 | no carry

H had no tree of degree 0 — nothing to collide with — so the singleton simply becomes the root list. **n = 1**, binary 1.

STEP 4 OF 8

ARRIVING: PATIENT 6

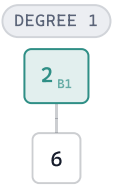
ANOTHER B_0

H − 1 PATIENT

H ALREADY HOLDS A B_0

Patient 6 arrives and is wrapped the same way. But this time H *already* has a degree-0 tree. **Two B_0's cannot coexist** — a binomial heap allows at most one tree per degree. In binary we are about to compute `1 + 1`.

STEP 5 OF 8



LINKED: 2 ≤ 6, SO 2 KEEPS THE ROOT

`n = 2` `binary 10` `carry`

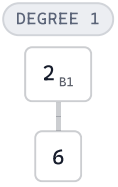
So they **link**. The smaller key wins the root: 2 ≤ 6, so 6 becomes 2's child and the pair is now a **B_1**. Watch the binary: `1 + 1 = 10` — the degree-0 slot empties and a degree-1 tree appears. **A carry in the counter is a link in the heap.**

STEP 6 OF 8

ARRIVING: PATIENT 9

A THIRD B_0

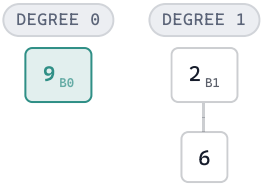
H − 2 PATIENTS



H HOLDS ONE B_1 − DEGREE-0 SLOT IS FREE

Patient 9 arrives. H now holds only a B_1, so the **degree-0 slot is empty** — no collision this time. Binary says the same thing: `10 + 1` needs no carry, because bit 0 is free.

STEP 7 OF 8

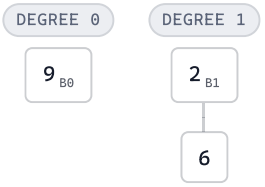


ROOT LIST IN INCREASING DEGREE: B_0, THEN B_1

n = 3 | binary 11 | 2 trees | no carry

It slots straight into the root list, ahead of the B_1 — **increasing degree**, as the root list always is. **n = 3**, binary **11**: both bits set, so both trees present.

STEP 8 OF 8

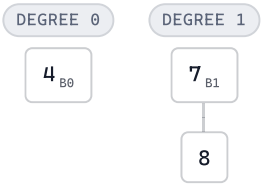


HOSPITAL A'S QUEUE: $B_0 = \{9\}$, $B_1 = \{2, 6\}$

3 patients | binary 11 | arrivals: 2, 6, 9 | 1 link in total

Done — and this is exactly the left-hand heap of the next slide. Three arrivals cost one link between them. Hospital B's queue is built the same way from arrivals 7, 8, 4. Once both exist, the interesting question is what happens when the two hospitals have to combine their queues.

HOSPITAL B, BUILT EXACTLY THE SAME WAY



B0 = {4}, B1 = {7, 8}

Arrivals **7**, then **8**, then **4**. Same story: 7 and 8 collide at degree 0 and link (7 ≤ 8, so 7 takes the root), then 4 finds the degree-0 slot empty and simply joins the root list. Three patients, binary **11**.

These two queues — Hospital A and Hospital B — are precisely the two inputs merged on the next slide.

EVERY INSERT IS +1 IN BINARY

AFTER ARRIVAL	N	BINARY	TREES PRESENT	CARRY?
—	0	0	none	—
2	1	1	B_0	no
6	2	10	B_1	**yes**

9	3	`11`	B_0, B_1	no

Read the binary column downward: `0, 1, 10, 11` — a counter incrementing. A **carry** in the counter is a **link** in the heap, and the two happen at exactly the same moments. L18·07 pushes this further with a triple cascade.

~15 MIN

Two hospitals, one disaster, one merged queue

Hospital A's queue: B_0={9}, B_1={2,6} (3 patients). Hospital B's queue: B_0={4}, B_1={7,8} (3 patients). UNION(H1,H2) has two phases: MERGE the two root lists by degree, then LINK equal-degree roots until at most one tree of each degree remains — exactly like adding 011 + 011 in binary.

READ THIS FIRST — THE FOUR CASES THE LINKING SWEEP CHOOSES BETWEEN

Phase 2 walks the merged root list with two pointers, `x` and `next-x`, and at each position takes exactly one of four branches. The step commentary below names them, so it is worth having them in front of you:

CASE	WHEN	WHAT IT DOES
1	degree(x) ≠ degree(next-x)	nothing to link here — just **advance** the pointers
2	**three** roots in a row share a degree	**advance** without linking, deferring the collision to the later pair
3	exactly two share a degree, and **key(x) ≤ key(next-x)**	**x stays the root**; next-x is linked underneath it
4	exactly two share a degree, and **key(x) > key(next-x)**	**next-x becomes the root**; x is linked underneath it

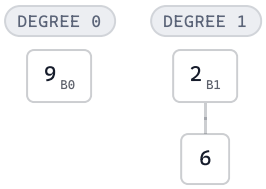
- **Cases 3 and 4 are the same link with the winner swapped** — the smaller key always ends up on top. They are written separately only because of pointer bookkeeping: in Case 3 `x` keeps its place in the root list, while in Case 4 `x` is spliced out and the traversal continues from `next-x`.
- **Cases 1 and 2 do not link at all** — they only move the pointers along.
- Because both input heaps held at most one tree per degree, the merged list can never contain *four* roots of the same degree, which is why three cases of collision (2, 3, 4) are enough.

In this particular run you will see Cases 2, 3 and 4. Case 1 never fires here, because after the merge every adjacent pair the sweep examines happens to have equal degree — a reminder that the cases are branches of one loop, not four stages that all must occur.

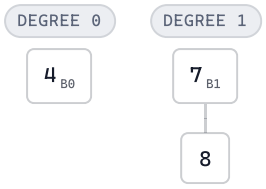
INTERACTIVE — BINOMIAL-HEAP-UNION(H1, H2)

STEP 1 OF 8

HOSPITAL A — 3 PATIENTS

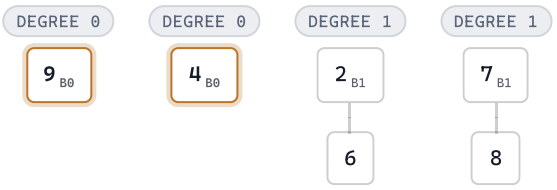


HOSPITAL B — 3 PATIENTS



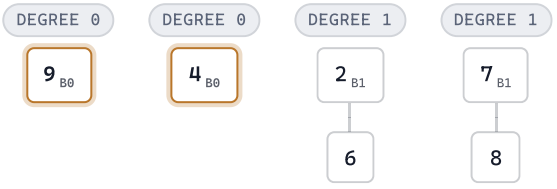
Hospital A: B0={9}, B1={root 2, child 6}. Hospital B: B0={4}, B1={root 7, child 8}. A mass-casualty event means both queues must become one, fast. UNION(H1,H2) begins.

STEP 2 OF 8



Phase 1 — MERGE: splice both root lists together, sorted by increasing degree: 9(deg0), 4(deg0), 2(deg1), 7(deg1). Each input heap had at most one tree per degree, so the merged list has at most TWO roots of any given degree.

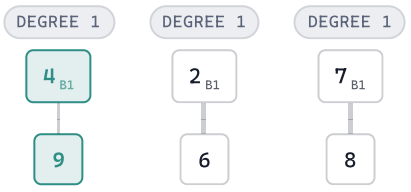
STEP 3 OF 8



X=9, NEXT-X=4 — THE FOLLOWING ROOT (2) HAS A DIFFERENT DEGREE

Phase 2 begins. x=9 (degree 0), next-x=4 (degree 0). The root after next-x is 2 (degree 1) — different degree, so this is NOT a three-way collision, just a simple pair (ruling out Cases 1 and 2). Compare keys: 9 vs. 4.

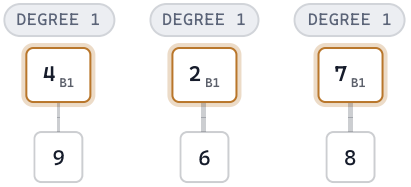
STEP 4 OF 8



THREE ROOTS, ALL DEGREE 1!

key[next-x]=4 < key[x]=9, so **Case 4** applies (two roots of equal degree, and key(x) > key(next-x)): the smaller key (next-x=4) becomes the parent — 9 is linked as its leftmost child, forming a new B1. Root list now: 4(B1,child9), 2(B1,child6), 7(B1,child8) — three roots, all degree 1.

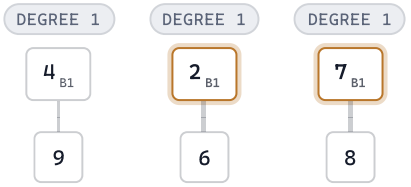
STEP 5 OF 8



X=4, NEXT-X=2, AND SIBLING[NEXT-X]=7 IS ALSO DEGREE 1

Case 2 — a three-way collision. x=4 (degree 1), next-x=2 (degree 1) — but the root FOLLOWING next-x, which is 7, is *also* degree 1. We do not link yet; we just advance the pointers one step, deferring the collision.

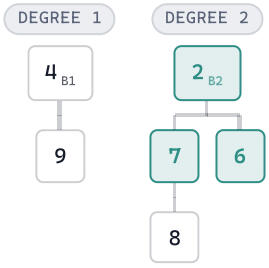
STEP 6 OF 8



X=2, NEXT-X=7 — NOTHING FOLLOWS 7

Pointers advance: now x=2 (degree 1), next-x=7 (degree 1), and nothing follows 7 — exactly two roots of degree 1 left. Compare keys: 2 vs. 7.

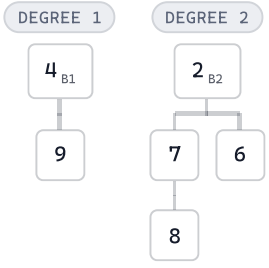
STEP 7 OF 8



UNION COMPLETE

key[x]=2 < key[next-x]=7, so **Case 3** applies (two roots of equal degree, and key(x) ≤ key(next-x)): x (2) stays the parent — the {7,8} tree is linked as x's new leftmost child, forming a B2. Root list now: 4(B1,child9), 2(B2, children 7&6, where 7 keeps its own child 8). No degree collides with any other — UNION is done.

STEP 8 OF 8



`6 patients total` `Root comparisons: 4` `Links performed: 2` `Minimum: 2 (Hospital A)`

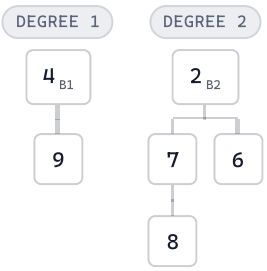
Done. UNION completed in O(log n) — just 4 root-list comparisons and 2 links, regardless of how large either hospital's queue actually was. The combined minimum is 2 (Hospital A's patient), found by scanning only the 2 remaining roots — 4 and 2.

One insert with zero carries, one with a full carry chain

INSERT(H,x) is just UNION(H, {x}) — a singleton heap merged in. Starting from the merged 6-patient queue (binary 110), watch two arrivals: patient priority 1 (no cascade: 110+1=111) and then patient priority 0 (a full triple cascade: 111+1=1000).

INTERACTIVE — INSERT(1), THEN INSERT(0)

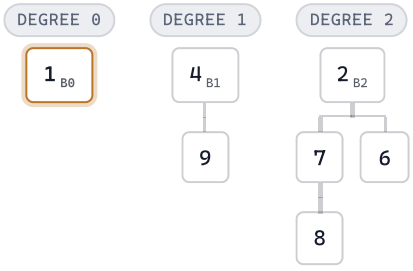
STEP 1 OF 8



1 1 0

Starting from the merged 6-patient queue: B1{4,9} + B2{2,{7,8},6}. Binary view: **110**. A new patient arrives, priority **1**. INSERT is just UNION with a fresh singleton heap {1}.

STEP 2 OF 8

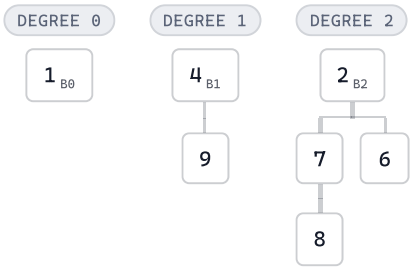


THREE DIFFERENT DEGREES — NO COLLISIONS AT ALL

1 1 0

Merge {1} into the root list by degree: 1(deg0), 4(deg1), 2(deg2) — three different degrees. Every comparison in the linking phase is a trivial pass-through (Case 1). **No linking needed.**

STEP 3 OF 8

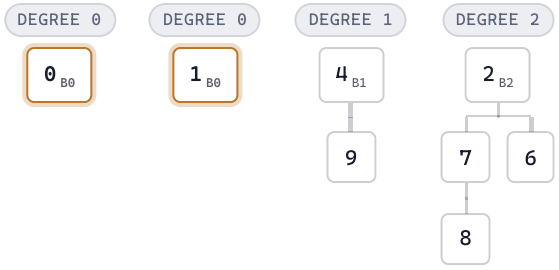


1 1 1

Links this insert: 0 | Cheapest possible case — O(1) actual work

Result: 3 trees, 7 patients total. Binary view: **110 + 1 = 111** — the last bit was 0, so no carry propagated. This is INSERT's best case.

STEP 4 OF 8

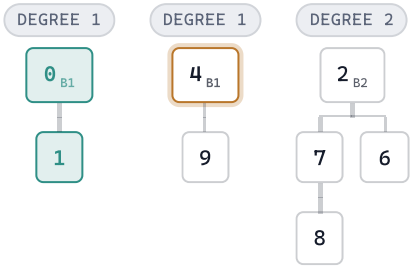


A SECOND NEW PATIENT, PRIORITY 0 — THE NEW DEGREE-0 NODE COLLIDES WITH THE EXISTING B0{1}

1 1 1

A second patient arrives, priority **0** — most urgent yet. Binary view: **111 + 1** — watch every bit carry. Merge phase: the new node (0, degree0) collides with the EXISTING degree-0 tree {1}.

STEP 5 OF 8

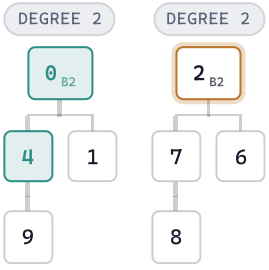


FIRST CARRY: A NEW B1{0,CHILD1} NOW COLLIDES WITH THE EXISTING B1{4,9}

1 1 0

0 < 1, so 0 stays parent — 1 links beneath it, forming a fresh B1{0,child1}. **First carry.** This new B1 immediately collides with the ALREADY-EXISTING B1{4,child9} — same degree again.

STEP 6 OF 8

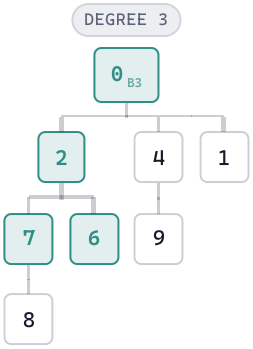


SECOND CARRY: A NEW B2 NOW COLLIDES WITH THE EXISTING B2{2,...}

1 0 0

0 < 4, so 0 stays parent again — the {4,9} tree links beneath it, forming a B2. **Second carry.** This new B2 collides with the ALREADY-EXISTING B2{2,{7,8},6} — same degree once more.

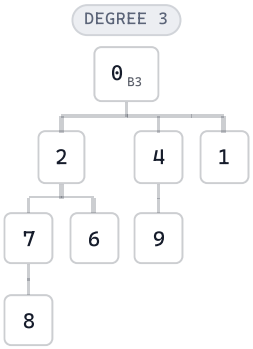
STEP 7 OF 8



A SINGLE B3 — NOTHING LEFT TO COLLIDE WITH

0 < 2, so 0 stays parent a third time — the {2,7,8,6} tree links beneath it, forming a B3. **Third carry** — and now there is nothing left to collide with.

STEP 8 OF 8



`Links this insert: 3 (full cascade)` `8 patients = $2^3$` `Amortized cost over both inserts: O(1)`

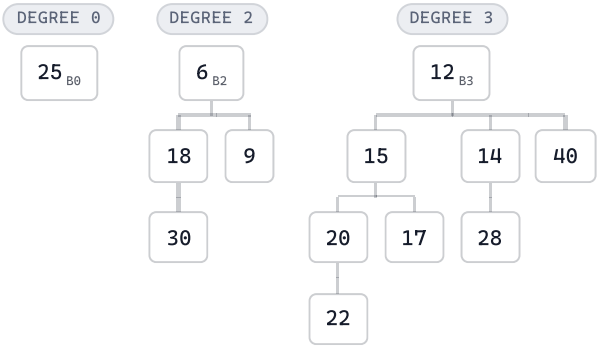
Result: ONE tree, B3, 8 patients — exactly 2^3. Binary view confirmed: **111 + 1 = 1000**, three carries matching the three links just traced. This is exactly the amortized argument behind incrementing a binary counter (Lecture 3's callback): a full cascade costs O(log n), but most inserts — like the very first one — cost O(1), so the amortized cost per insert is O(1).

Scan the roots, remove the winner, reverse, union

A 13-patient queue — B_0, B_2, B_3. Two questions the shape alone cannot answer: **which** root gets removed, and what happens to the rest afterwards. The root list is sorted by **degree**, not by key, so the minimum has to be hunted for — and here it is deliberately in the *middle* tree, so neither "the first root" nor "the biggest tree" is the answer.

INTERACTIVE — BINOMIAL-HEAP-EXTRACT-MIN(H)

STEP 1 OF 11

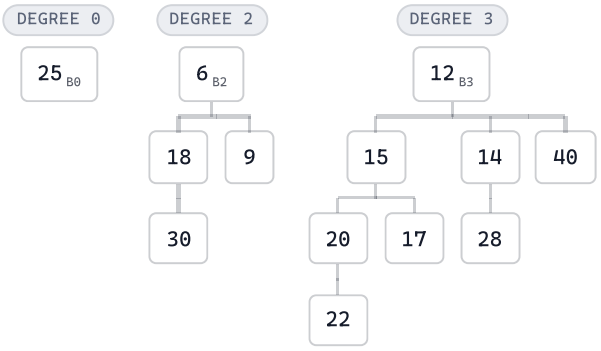


ROOT LIST ORDERED BY **DEGREE** 0, 2, 3 — THE KEYS 25, 6, 12 ARE IN NO ORDER AT ALL

n = 13 | binary 1101 | 3 trees

A 13-patient queue: **B0** (root 25), **B2** (root 6), **B3** (root 12). Remember what is sorted here and what is not — the root list is in increasing **degree**, but the root *keys* 25, 6, 12 follow no pattern. So we cannot simply read off the minimum.

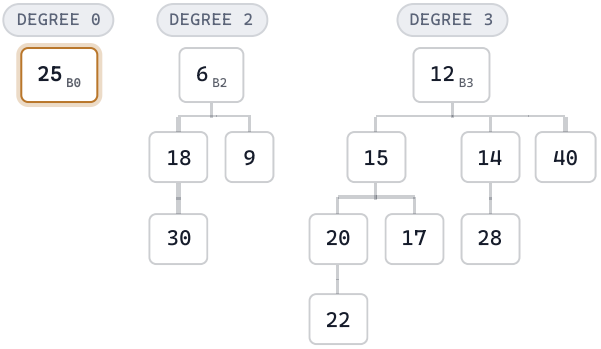
STEP 2 OF 11



THE HEAP PROPERTY RUNS DOWN EACH TREE — NEVER SIDEWAYS BETWEEN TREES

Where can the minimum hide? Inside any one tree every node is ≥ its parent, so each tree's own smallest key sits at **its root**. The global minimum must therefore be one of the three roots — but nothing tells us *which*. **MINIMUM must scan the whole root list.**

STEP 3 OF 11

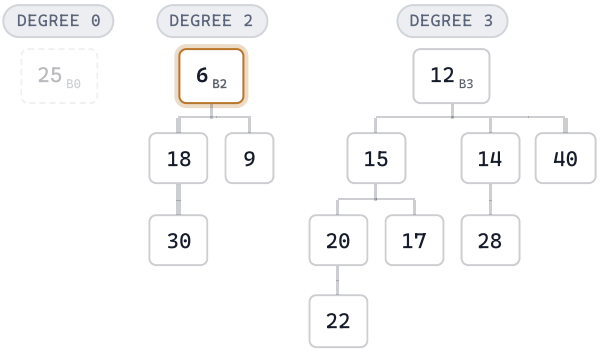


BEST SO FAR: 25

comparisons: 1

Scan the root list. First root: **25**. Nothing to compare against yet, so it is the best so far.

STEP 4 OF 11

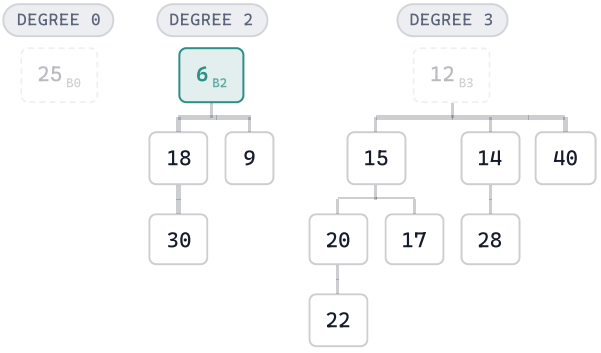


BEST SO FAR: 6 (6 < 25)

comparisons: 2

Second root: **6**. Since 6 < 25, the best so far becomes **6** — sitting at the root of the **B2**.

STEP 5 OF 11



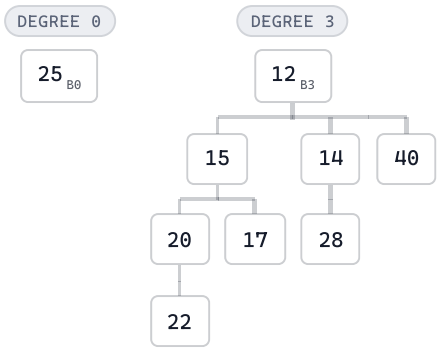
WINNER: 6 — THE MIDDLE TREE, NEITHER FIRST NOR LARGEST

comparisons: 3 | roots scanned: all 3 | cost O(log n)

Third root: **12**. 12 > 6, so the winner stands: **min = 6**, the root of the B2. Note what did *not* decide this — its **degree**. The minimum happened to be in the middle tree; it could equally have been the lone B0 or the big B3. **Return 6** and remove that root.

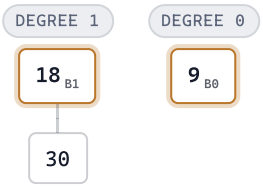
STEP 6 OF 11

H′ — WHAT IS LEFT OF THE HEAP (9 NODES)



STILL LEGAL: DEGREES 0, 3 — DISTINCT AND INCREASING

THE ORPHANED CHILDREN OF 6 (3 NODES)

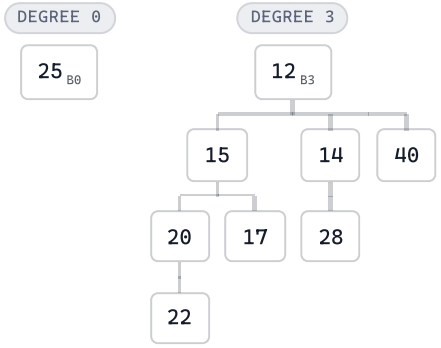


AS STORED: **DECREASING** — NOT A LEGAL ROOT LIST

Splice root 6 out of the root list. Two separate things are now in our hands: **H′**, the untouched B0 and B3 — still a perfectly legal heap — and the **orphans**, 6's children, sitting in stored order B1, B0. Node count checks out: 9 + 3 = 12 = 13 − 1.

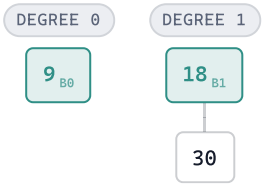
STEP 7 OF 11

H′ — 9 NODES



STILL LEGAL: DEGREES 0, 3 — DISTINCT AND INCREASING

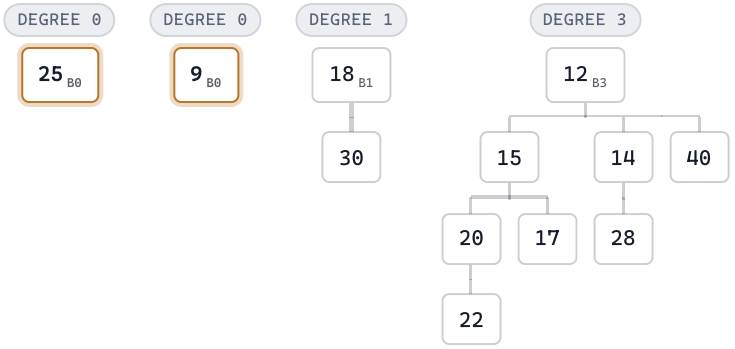
H″ — REVERSED, NOW A LEGAL HEAP, 3 NODES



REVERSED: **INCREASING** — A VALID BINOMIAL HEAP IN ITS OWN RIGHT

Reverse the orphan chain: B1, B0 becomes **B0, B1**. That single pointer reversal is all it takes — **H″** is now a legitimate binomial heap. This is exactly why child lists are stored in decreasing order (L18·04b).

STEP 8 OF 11

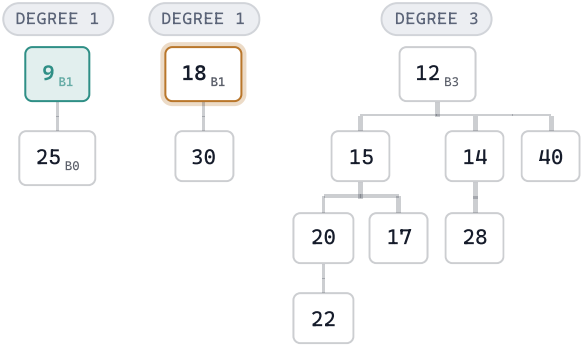


MERGED BY DEGREE — THE TWO B0'S LANDED SIDE BY SIDE

H': 0, 3 | H": 0, 1 | merged: 0, 0, 1, 3

UNION(H', H") — and this is the step the single-tree example never showed. Merge the two root lists by degree in one pass: 0, 0, 1, 3. Because both were sorted, the duplicate degrees are **adjacent**, so the sweep only ever looks at its neighbour.

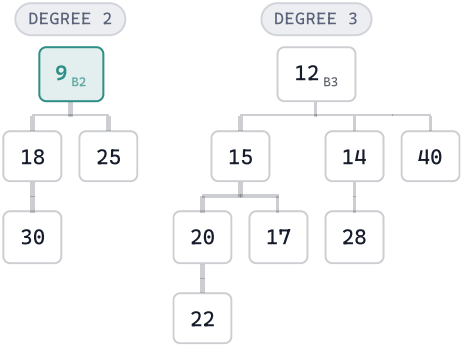
STEP 9 OF 11



CARRY: TWO B0'S LINKED → B1 · BUT NOW TWO B1'S COLLIDE

First collision: two **B0**'s. The **smaller key wins the root** — 9 < 25, so 25 becomes 9's child, giving a **B1** rooted at 9. But that new B1 now sits next to the B1 rooted at 18 — the carry propagates, exactly like adding 1 + 1 in binary.

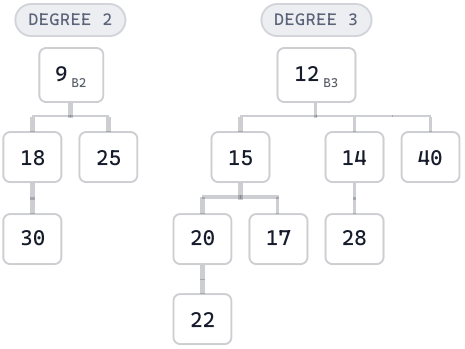
STEP 10 OF 11



CARRY RESOLVED: B2 ROOTED AT 9 · DEGREES 2, 3 ARE DISTINCT — SWEEP STOPS

Second collision: two **B1**'s. Again the smaller key takes the root — 9 < 18 — so 18's subtree becomes 9's new **leftmost** child, ahead of 25. That gives a **B2** rooted at 9 (children in degree order B1, B0 — decreasing, as always). Next along is the B3; degrees 2 and 3 differ, so the sweep is done.

STEP 11 OF 11



returned min: 6 | n: 13 → 12 | binary 1101 → 1100 | new min: 9

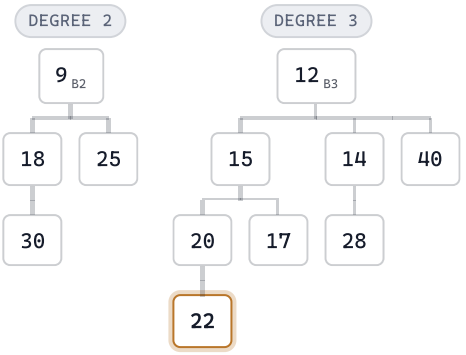
Done. EXTRACT-MIN returned **6** and left a valid heap: B2 rooted 9, B3 rooted 12 — 4 + 8 = **12** nodes. In binary the heap went **1101** → **1100**: removing one patient is a binary decrement, just as inserting one is an increment. And finding the new minimum, 9, will again take a scan of the roots.

A patient deteriorates: raise their priority, walk it up

A patient already in the queue gets worse, so their key must **drop**. Continuing with the 12-patient heap left by the last slide, we decrease the deepest node of the B_3 — key **22** — to **5**. Watch what moves and, more importantly, **what does not**.

INTERACTIVE — BINOMIAL-HEAP-DECREASE-KEY(H, X, 5)

STEP 1 OF 7

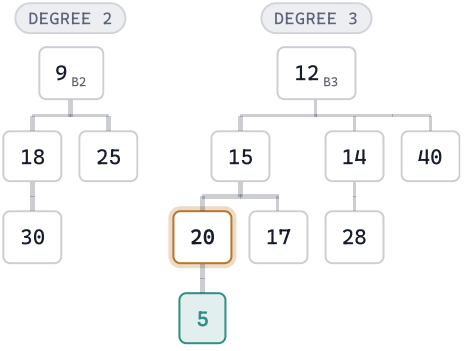


X = THE NODE HOLDING 22 — DEPTH 3, THE DEEPEST IN THE HEAP

`12 patients` `current min: 9` `path to root: 22 → 20 → 15 → 12`

The heap left by EXTRACT-MIN: a B_2 rooted at 9 and a B_3 rooted at 12. Patient **22** has deteriorated. We hold a **pointer** to their node — we do not go looking for it — and we want the key lowered to **5**.

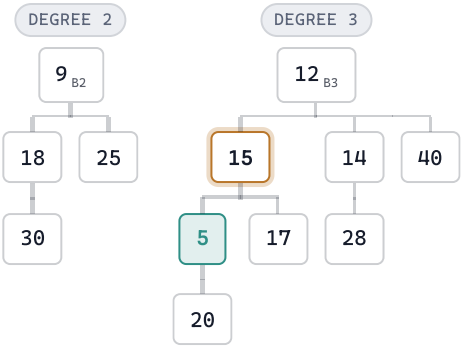
STEP 2 OF 7



KEY WRITTEN: 5 — BUT 5 < 20, SO THE HEAP PROPERTY IS BROKEN

Write the new key in place: `x.key = 5`. The structure is still a perfect B_3, but the **heap property is now violated**: 5 sits below its parent 20. Compare the two — 5 < 20 — so they must swap.

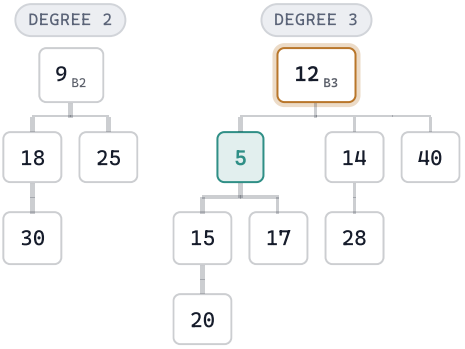
STEP 3 OF 7



SWAP 1 OF 3 — NOW COMPARE AGAINST 15

Swap the keys of 5 and 20 — not the nodes. 5 rises one level, 20 sinks one. Move up and compare again: the parent is **15**, and 5 < 15, so we are not finished.

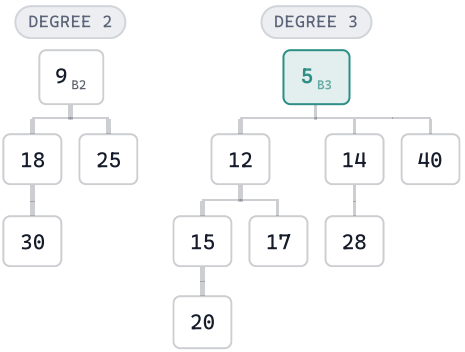
STEP 4 OF 7



SWAP 2 OF 3 — NOW COMPARE AGAINST THE ROOT, 12

Swap again. 5 is now the root of the B_2 subtree, with 15 beneath it and 20 beneath that — that whole branch stays correctly ordered. One comparison left: the tree root **12**, and 5 < 12.

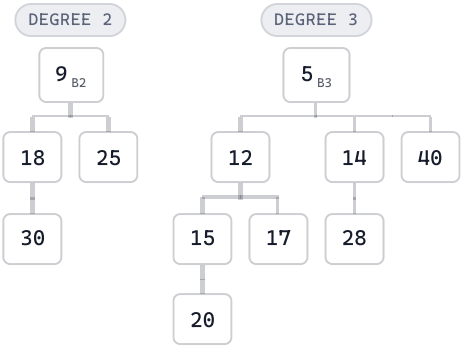
STEP 5 OF 7



SWAP 3 OF 3 — 5 IS NOW A ROOT, SO Z = NIL AND THE LOOP ENDS

Final swap: 5 takes the root of the B_3 and 12 moves down into the B_2 branch. Now `y.parent = NIL` — there is nothing above to compare against — so the loop **stops**. Three swaps for a tree of height 3.

STEP 6 OF 7

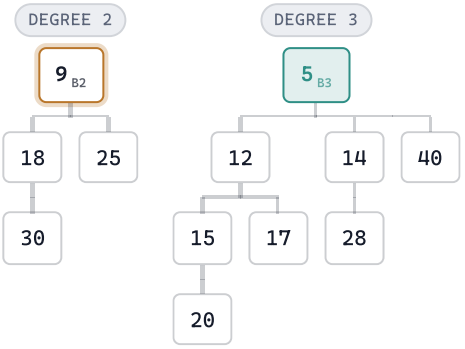


STILL A B_2 AND A B_3 — DEGREES 2 AND 3, EXACTLY AS BEFORE

swaps: 3 | height of B_3: 3 | links: 0 | root list: unchanged

Check what did not happen. No tree was linked, no tree was split, no root joined or left the root list, and every degree is what it was. Only four key fields changed value. The heap property holds everywhere: 5 ≤ 12 ≤ 15 ≤ 20, and 12 ≤ 17, 5 ≤ 14 ≤ 28, 5 ≤ 40.

STEP 7 OF 7



THE MINIMUM MOVED: 9 → 5

new min: 5 | MINIMUM still scans all roots | DECREASE-KEY: O(log n)

One consequence worth noticing: the global minimum has changed from **9** to **5**. DECREASE-KEY did nothing to record that — it never looked at the other tree. MINIMUM still finds it the only way it can, by **scanning the root list** (L18·08). And because a decreased key can only ever rise, it can only ever *become* the minimum — never stop being it.

THE ALGORITHM — AND WHY IT IS O(LOG N)

```
DECREASE-KEY(H, x, k)
  if k > x.key: error
  x.key = k
  y = x;  z = y.parent
  while z ≠ NIL and y.key < z.key:
      swap y.key ⇔ z.key
      y = z;  z = y.parent
```

- **Keys move, nodes do not.** Only key fields are swapped — every `child`, `sibling` and `degree` is untouched, so the tree stays a valid B_3 and the root list never changes.
- **The loop is bounded by height.** It walks x up to its root, and a B_k has height $k \le \lfloor \lg n \rfloor$ — so at most **O(log n)** swaps. That is the "sift-up path" the complexity table refers to.

→ **This is what `parent` is for.** It is the only operation in this lecture that travels *upward*; L18·04a's fifth pointer field finally earns its place.

TWO THINGS STUDENTS ALWAYS ASK

1. How do you find x in the first place?

You do not search for it — that would be O(n) and would wreck the bound. DECREASE-KEY takes a **pointer** to the node. Real users keep a handle: Dijkstra and Prim hold a `vertex → node` array, so relaxing an edge jumps straight to the right node.

2. What about DELETE?

It reuses what you already have — no new machinery:

```
DELETE(H, x)
  DECREASE-KEY(H, x, −∞)
  EXTRACT-MIN(H)
```

Drive the key below every other key and it sifts all the way to the root of its tree; being the minimum, EXTRACT-MIN then removes it. Two O(log n) steps, so DELETE is **O(log n)** as well.

WHY LECTURE 19 EXISTS

Nothing above is wasteful — yet DECREASE-KEY is the operation that keeps binomial heaps out of the top spot for Dijkstra and Prim, where edge relaxations vastly outnumber extractions. Every relaxation pays an O(log n) walk up the tree. **Fibonacci heaps** attack exactly this: instead of sifting up, they *cut* the node out and drop it into the root list, deferring all the tidying — O(1) amortized. That single change is the subject of the next lecture.

One column changes dramatically; the rest barely move

OPERATION	BINARY HEAP (WORST-CASE)	BINOMIAL HEAP (WORST-CASE)	FIBONACCI HEAP (AMORTIZED) — LECTURE 19
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)
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)

WHERE THOSE COSTS COME FROM — EVERY SHAPE PROPERTY FROM L18·02A, CASHED IN

Each property of B_k is load-bearing for exactly one column above. Now that you have watched all three operations run, they can be lined up:

PROPERTY OF B_k	WHAT IT BUYS	SEEN IN
root has **degree k**	lets UNION spot two trees of the *same* order and link them — the carry step	L18·06, L18·07
children are roots of $\mathbf{B_{k-1}, \ldots, B_0}$	deleting a root leaves trees of *distinct* orders — already a legal heap, so no repair is needed	L18·08
height = k	bounds every sift-up path, which is what makes DECREASE-KEY and DELETE O(log n) — watched in L18·08a	the table above
$\mathbf{2^k}$ nodes	one tree per order mirrors one binary digit per place — so at most ⌊lg n⌋+1 trees ever exist	L18·04, L18·07

Notice the one row that is a genuine *loss*: MINIMUM. A binary heap reads it from A[1] in Θ(1); a binomial heap must scan every root, and there are O(log n) of them. That is the price of being a forest instead of one tree — and against Θ(n) UNION, it is a bargain.

n = total nodes across the heap(s) involved. Every binomial-heap number above was exercised today: UNION took 4 root comparisons and 2 links for two 3-node heaps; INSERT ranged from 0 links (best case) to 3 cascading links (worst case, still O(log n)).

O(log n) merging — and what it costs to get there

ADVANTAGES

- **UNION in O(log n)** — the headline improvement over a binary heap's Θ(n), verified today on two real 3-node heaps.
- Every other core operation (INSERT, MINIMUM, EXTRACT-MIN, DECREASE-KEY, DELETE) still runs in **O(log n)** or better — no regression versus a binary heap.
- A genuinely clean mental model: a binomial heap behaves exactly like a **binary counter** — INSERT is "add 1," UNION is "binary addition," carries and all.
- INSERT is O(log n) worst case but **O(1) amortized** — most insertions cause zero cascading links, exactly like most binary-counter increments flip only the lowest bit.

LIMITATIONS

- **More overhead per node.** Parent/child/sibling/degree pointers cost real memory, versus a binary heap's implicit array indices — and pointer-chasing is less cache-friendly than sequential array access.
- **More complex to implement correctly.** The four linking cases in UNION (Lecture 18's own flagship animation) are meaningfully trickier than a binary heap's single sift-up/sift-down rule.
- **No benefit if you never need UNION.** A plain binary heap remains simpler, more cache-friendly, and equally fast for insert/extract-min alone — don't reach for a binomial heap unless merging is actually part of your workload.
- **Still not the best at DECREASE-KEY-heavy workloads** (e.g. Dijkstra's algorithm with many edge relaxations) — Lecture 19's Fibonacci heap improves exactly that operation, from O(log n) to O(1) amortized.

Not a hypothetical exercise — merging comes up constantly

EMERGENCY RESPONSE (TODAY'S EXAMPLE)

Combining two independently-run triage queues — hospitals, disaster-response teams, or field units — into one, fast, without ever rebuilding from scratch.

DISTRIBUTED JOB SCHEDULERS

When two compute clusters or worker pools are consolidated (auto-scaling merges, data-center failover), their pending-job priority queues need to become one queue without an expensive full re-sort.

MINIMUM SPANNING TREES

A classic MST algorithm variant maintains one edge-priority-queue per connected component and repeatedly extracts the cheapest edge; whenever two components merge, their two queues must merge too — exactly a binomial-heap UNION, once per merge.

EVENT-DRIVEN SIMULATION AT SCALE

Large distributed simulations run independent event queues per node; periodically synchronizing (merging) them into a global queue benefits directly from fast UNION.

VERSION-CONTROL / CRDT MERGES

Systems that maintain ordered pending-operation queues on independent replicas, then merge after a network partition heals, face the same "combine two already-valid structures cheaply" problem.

LEADERBOARD / MATCHMAKING CONSOLIDATION

Regional game servers merging after a scheduled maintenance window need to combine independent priority-ranked matchmaking queues into one.

A forest that merges like a binary counter adds

- **The binomial tree B_k** is defined by one rule (link two B_{k-1} trees) and that rule alone forces 2^k nodes, height k, and C(k,i) nodes at depth i — verified today by building B_0 through B_3 from scratch.
- **A binomial heap is a forest with at most one tree per degree** — literally the binary representation of n, with INSERT behaving exactly like incrementing a binary counter (verified: 110+1=111 with zero carries, then 111+1=1000 with three).
- **UNION runs in O(log n)**, not Θ(n) — verified on a real 6-node merge covering all three non-trivial linking cases (a lone pair, a three-way collision deferred by one step, and a resolved pair).
- **EXTRACT-MIN is "scan the roots, remove the winner, reverse its children into a new heap, union"** — the scan is needed because the root list is ordered by degree and not by key; the rest reuses UNION rather than inventing new machinery.
- **The trade-off:** more pointer overhead and implementation complexity than a binary heap, paid only to buy fast merging — the right tool exactly when UNION is actually part of the workload.

LECTURE 19 — NEXT

Fibonacci Heaps and Amortized Efficiency

Binomial heaps still cost O(log n) for DECREASE-KEY. Fibonacci heaps get it down to O(1) amortized — by being lazier about when they actually fix the structure.

HOMEWORK — BRING TO LECTURE 19

- Draw B_4 by hand from the recursive definition (two B_3s linked). Confirm it has 16 nodes, height 4, and that depth-2 has exactly C(4,2)=6 nodes.
- Build two small binomial heaps of your own (e.g. 4 nodes and 5 nodes) and trace BINOMIAL-HEAP-UNION by hand. Which of the four cases does each step trigger?
- Starting from a 7-node binomial heap (binary 111), trace 1 more INSERT by hand. Confirm you get a single B_3 (binary 1000) with exactly 3 cascading links.
- In 3–4 sentences: why is INSERT O(log n) worst-case but O(1) amortized? Connect this explicitly to the binary-counter argument from Lecture 3's amortized analysis.

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

> *Next: Lecture 19 — Fibonacci Heaps and Amortized Efficiency.*
