Comparative Study of Heap Structures

Binary, Binomial, Fibonacci, and Pairing heaps all answer the same question — "give me the smallest, fast" — with four completely different bets about when to do the work. Today we run the identical operation sequence through all four, side by side, and watch the bets pay off differently.

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

~90 minutes
Session outcome: compare Binary, Binomial, Fibonacci, and Pairing heaps

~90 MIN

One race, four contestants

Why compare now? 00–08

Four lectures (16, 18, 19, 20) built four structures for one job. Today: put them next to each other.

Structural recap, side by side 08–18

Array vs. forest-of-trees vs. lazy-forest vs. single-multiway-tree — four different answers to "how do I hold a set of keys?"

The four comparison axes 18–28

Eagerness, degree bound, merge cost, bookkeeping overhead — the real design variables underneath the Big-O.

The grand complexity table 28–38

Every operation, all four structures, one table — including the one bound that's still an open problem.

Animation: the same-sequence race 38–68

Insert 2, 8, 4, 15, 10 → merge → decrease-key → extract-min. Applied to all four heaps at once. Every operation, every structure, nothing skipped.

Final tally — and an honest caveat 68–74

Who did the least work on this one small example — and why that is not a proof of anything asymptotic.

Decision guide 74–82

Bounded array PQ, merge-heavy workload, decrease-key-heavy graph algorithm, need-it-simple-and-fast — which heap, and why.

Where each one actually runs 82–88

Standard libraries, graph engines, and why the "theoretical champion" is rarely the one deployed.

Recap & what's next 88–90

Lecture 22: Introduction to Spatial Data Structures.

Four lectures, one job: hold a set, answer "what's smallest?" fast

THE SHARED CONTRACT

Every structure in Lectures 16–20 implements the same abstract priority queue: INSERT, FIND-MIN, EXTRACT-MIN, and (for three of the four) UNION/MERGE and DECREASE-KEY. If the interface is identical, the only thing left to compare is **how each structure pays for that interface** — and that's a genuinely different story for each one.

THE RECURRING LESSON

Each new heap in this course existed to fix exactly one weakness of the previous one: Binary heap can't merge in less than O(n) → **Binomial heap** fixes merge to O(log n) by using a forest, not one array. Binomial heap's decrease-key is still O(log n) → **Fibonacci heap** fixes it to O(1) amortized by going lazy. Fibonacci heap's constant factors are painful in practice → **Pairing heap** fixes that by throwing away every bookkeeping field except three pointers.

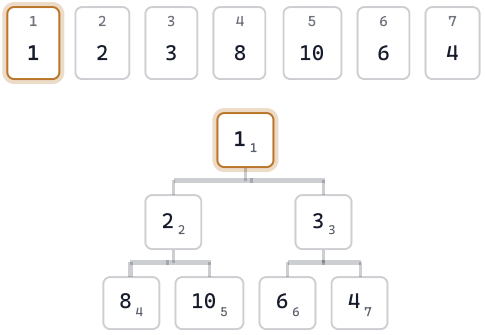
TODAY'S METHOD

Rather than compare asymptotics in the abstract, we build all four structures from empty, feed the **exact same sequence** of operations into each one, and watch what actually happens — structurally and in terms of real pointer work — at every single step. This mirrors Lecture 15's dual-panel AVL-vs-Red-Black comparison, extended to four panels.

Four different answers to "how do I hold a set of keys?"

ARRAY, IMPLICIT TREE

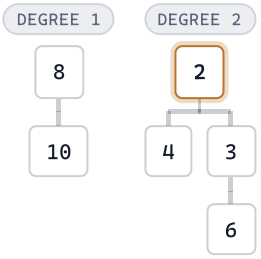
Binary heap (L16)



ONE ARRAY. PARENT(I)=⌊I/2⌋. COMPLETE TREE, SO HEIGHT IS ALWAYS Θ(LOG N).

FOREST, ONE TREE PER SET BIT

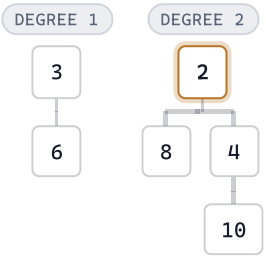
Binomial heap (L18)



N = 6 → BINARY 110 → EXACTLY A B1 AND A B2 TREE, NEVER TWO OF THE SAME DEGREE.

LAZY FOREST, MARKS TRACK CUTS

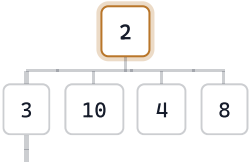
Fibonacci heap (L19)



CONSOLIDATION ONLY HAPPENS AT EXTRACT-MIN — EVERYTHING ELSE JUST TOUCHES THE ROOT LIST.

SINGLE MULTIWAY TREE

Pairing heap (L20)



6

NO FOREST AT ALL — ONE TREE, SIBLING LIST, NO PARENT/DEGREE/MARK FIELD.

The real design variables underneath the Big-O

AXIS 1 — EAGERNESS

Binary heap restores the heap property immediately after every operation (eager). **Binomial heap** is also eager but only within a UNION/consolidation call. **Fibonacci heap** is deliberately **lazy** — it defers all structural cleanup to the next EXTRACT-MIN. **Pairing heap** is lazy in the same spirit but pays as it goes on every MELD, with cleanup concentrated at REMOVE-MIN.

AXIS 2 — SHAPE: ONE TREE, OR MANY?

Binary heap: one complete binary tree, stored flat in an array. **Binomial heap**: a forest of binomial trees, at most one per degree — a direct mirror of binary counter bits. **Fibonacci heap**: an unordered forest of arbitrary-shaped trees, root list circular and doubly linked. **Pairing heap**: a single unordered multiway tree — no forest at all.

AXIS 3 — WHAT MERGE COSTS

This is the single biggest structural fork in the family. Binary heap's array layout makes merging two heaps fundamentally rebuild-from-scratch work: **O(n)**. The other three all store multiple independent trees (or a splice-friendly single tree), so merging is just relinking a constant or logarithmic number of root pointers: **O(log n)** for Binomial, **O(1)** for Fibonacci and Pairing.

AXIS 4 — PER-NODE BOOKKEEPING

Binary heap: zero extra fields (position is implicit in the array index). Binomial heap: parent, child, sibling, degree. Fibonacci heap: parent, child, left, right, degree, **and** a mark bit — the richest node of the four, and the reason its constant factors are the worst. Pairing heap: child, left, right — **no** parent, degree, or mark field, thanks to the "first child's left pointer is the parent" trick from Lecture 20.

Every operation, all four structures, one table

OPERATION	BINARY HEAP (WORST-CASE)	BINOMIAL HEAP (WORST-CASE)	FIBONACCI HEAP (AMORTIZED)	PAIRING HEAP (AMORTIZED)
MAKE-HEAP	Θ(1)	Θ(1)	Θ(1)	Θ(1)
INSERT	Θ(log n)	O(log n)	Θ(1)	O(1)
FIND-MIN	Θ(1)	O(log n)	Θ(1)	O(1)
EXTRACT-MIN	Θ(log n)	Θ(log n)	O(log n)	O(log n) — proven
UNION / MERGE	**Θ(n)**	**O(log n)**	**Θ(1)**	**O(1)**
DECREASE-KEY	Θ(log n)	Θ(log n)	Θ(1)	conjectured O(1); best proven bound is looser — a genuine open problem
DELETE	Θ(log n)	Θ(log n)	O(log n)	O(log n) — proven, same machinery as EXTRACT-MIN

Read this table as a story, not just a lookup: Binomial heap exists purely to fix Binary heap's bolded UNION row. Fibonacci heap exists purely to fix DECREASE-KEY from O(log n) to O(1). Pairing heap keeps Fibonacci's practical speed but honestly cannot yet *prove* the same DECREASE-KEY bound — the one asterisk in an otherwise-solid table, exactly as Lecture 20 flagged it.

~30 MIN

Insert 2, 8, 4, 15, 10 → merge in {6, 3} → decrease-key 15→1 → extract-min

Watch all four structures respond to the **identical** sequence of eight operations, one operation per step — nothing merged into a single step. Each structure was built completely independently and hand-verified; every one arrives at the same underlying key set after each shared operation, but by very different internal work. A running "structural operations so far" counter tracks pointer-rewires (swaps or links) for each heap.

INTERACTIVE — SAME SEQUENCE, FOUR HEAPS

STEP 1 OF 9

BINARY HEAP

EMPTY HEAP

BINOMIAL HEAP

EMPTY — NO ROOTS

FIBONACCI HEAP

EMPTY — NO ROOTS

PAIRING HEAP

EMPTY — NO TREE YET

Binary ops so far: 0

Binomial ops so far: 0

Fibonacci ops so far: 0

Pairing ops so far: 0

All four heaps start empty. We will run the identical 8-operation sequence through each one, one operation per step.

STEP 2 OF 9

Binary ops so far: 0 | Binomial ops so far: 0 | Fibonacci ops so far: 0 | Pairing ops so far: 0

INSERT 2. All four: trivial — a single node/root/1-element array. Zero structural work anywhere yet.

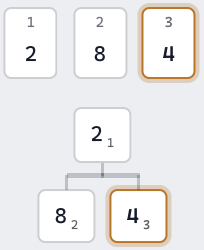
STEP 3 OF 9

Binary ops so far: 0 | Binomial ops so far: 1 | Fibonacci ops so far: 0 | Pairing ops so far: 1

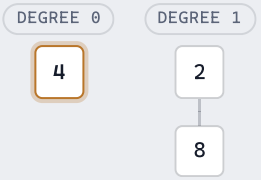
INSERT 8. Binary: 8 > parent 2, satisfies heap order already, 0 swaps. Binomial: two B0 trees collide → LINK (8 becomes 2's child), 1 link. Fibonacci: just appended as a new root, 0 links (lazy). Pairing: MELD{2},{8} → 2 < 8, so 8 becomes 2's child, 1 link.

STEP 4 OF 9

BINARY HEAP

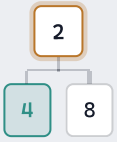


BINOMIAL HEAP



FIBONACCI HEAP

PAIRING HEAP

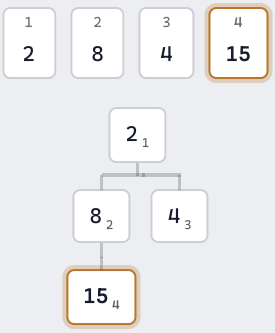


`Binary ops so far: 0` `Binomial ops so far: 1` `Fibonacci ops so far: 0` `Pairing ops so far: 2`

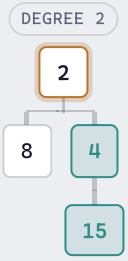
INSERT 4. Binary: 4 > parent 2, 0 swaps. Binomial: no B0 to collide with (existing heap has only a B1) → just added, 0 links. Fibonacci: appended, 0 links. Pairing: MELD{2→8},{4} → 2 < 4, so 4 becomes 2's new LEFTMOST child, pushing 8 right — 1 link.

STEP 5 OF 9

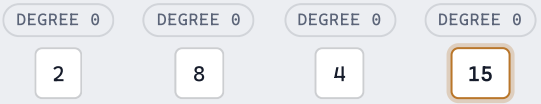
BINARY HEAP



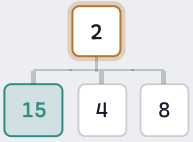
BINOMIAL HEAP



FIBONACCI HEAP



PAIRING HEAP

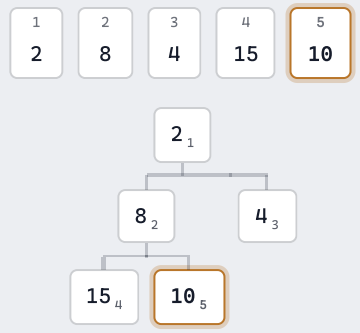


Binary ops so far: 0 | Binomial ops so far: 3 | Fibonacci ops so far: 0 | Pairing ops so far: 3

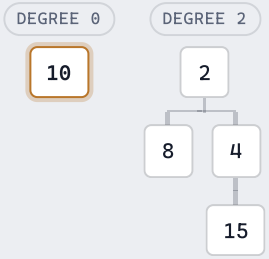
INSERT 15. Binary: 15 > parent 8, 0 swaps. Binomial: this is the expensive case — new B0{15} collides with existing B0{4} (link: 4<15, 15 becomes 4's child, forming a new B1) — THAT B1 immediately collides with the existing B1{2,[8]} (link: 2<4, the new B1 becomes 2's child) — 2 links, a genuine "carry", forming a single B2. Fibonacci: appended, 0 links. Pairing: MELD → 2 < 15, so {15} becomes 2's new leftmost child — 1 link.

STEP 6 OF 9

BINARY HEAP

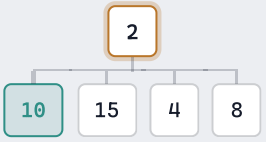


BINOMIAL HEAP



FIBONACCI HEAP

PAIRING HEAP

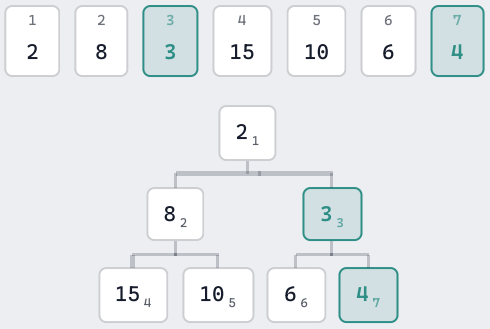


Binary ops so far: 0 | Binomial ops so far: 3 | Fibonacci ops so far: 0 | Pairing ops so far: 4

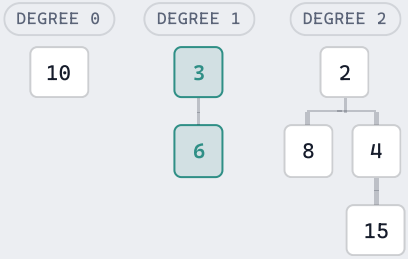
INSERT 10 (last of the 5 inserts). Binary: 10 > parent 8, 0 swaps. Binomial: no B0 to collide with (heap now has only a B2) → just added, 0 links. Fibonacci: appended, 0 links. Pairing: MELD → 2 < 10, so {10} becomes 2's new leftmost child — 1 link. After 5 inserts: Binary 0 swaps, Binomial 3 links, Fibonacci 0 links, Pairing 4 links.

STEP 7 OF 9

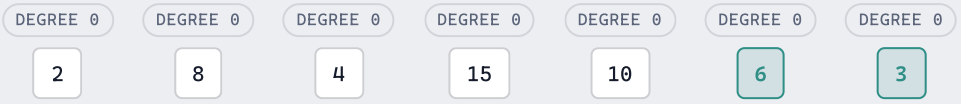
BINARY HEAP



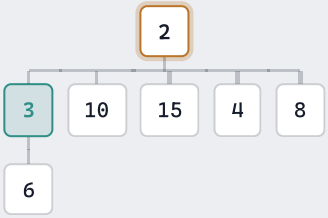
BINOMIAL HEAP



FIBONACCI HEAP



PAIRING HEAP

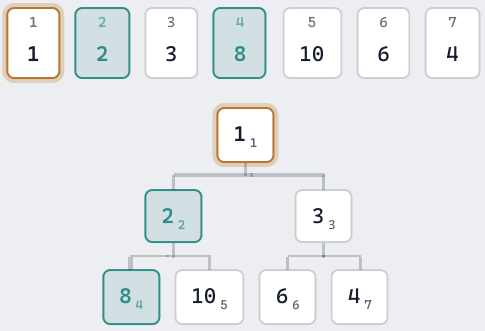


Binary ops so far: 1 | Binomial ops so far: 3 | Fibonacci ops so far: 0 | Pairing ops so far: 5

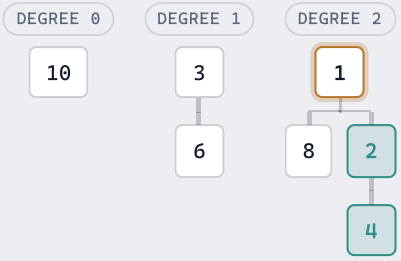
MERGE with a second heap {6, 3} (built the same way: insert 6, then insert 3 → 3 wins, 6 becomes its child). Binary heap has no efficient merge — it must rebuild: concatenate the arrays and run BUILD-MIN-HEAP over all 7 elements. Every non-leaf position (indices 3, 2, 1) is inspected; only one violation is found and fixed (swap 4 ↔ 3 at positions 3 and 7) — 1 swap, but O(n) work regardless. Binomial: n1=5 (101), n2=2 (010), no shared degree → simple splice, 0 links. Fibonacci: splice the root lists, update the min pointer — 0 links, always O(1). Pairing: one MELD, tree(2…) vs tree(3→6) → 2 < 3, so tree(3→6) becomes 2's new leftmost child — 1 link.

STEP 8 OF 9

BINARY HEAP

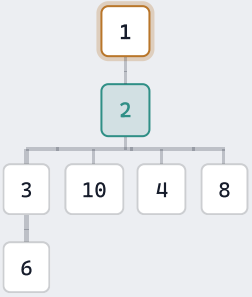


BINOMIAL HEAP



FIBONACCI HEAP

PAIRING HEAP

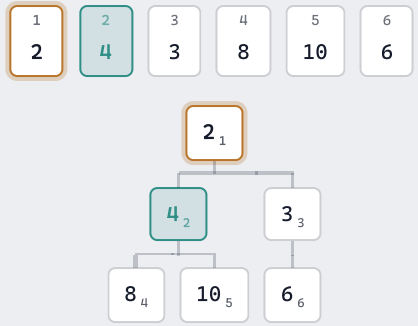


Binary ops so far: 3 | Binomial ops so far: 5 | Fibonacci ops so far: 0 | Pairing ops so far: 7

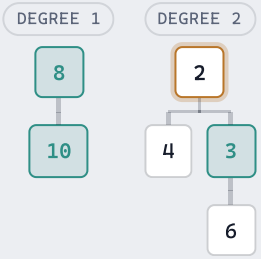
DECREASE-KEY: the node holding 15 becomes 1. Binary: 1 is stored at index 4, whose value was 15; sift-up swaps it past index 2 (value 8) then index 1 (value 2) — 2 swaps, ends at the root. Binomial: same idea inside the B2 tree — the node holding 15 (child of 4) swaps up past 4, then past 2 — 2 key-swaps, node holding 1 ends at that tree's own root. Fibonacci: the node holding 15 was still an UNCONSOLIDATED ROOT (nothing has forced a link on it yet) — decreasing a root's key never violates heap order, so this costs 0 cuts, purely because of where this node happened to sit. Pairing: detach the node from the sibling list (O(1), no parent pointer needed — Lecture 20's trick), update its key to 1, MELD the 1-node tree back in: 1 < 2, so the ENTIRE remaining tree becomes 1's new child — 1 detach + 1 meld = 2 ops.

STEP 9 OF 9

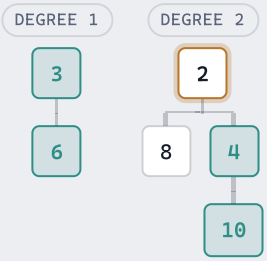
BINARY HEAP



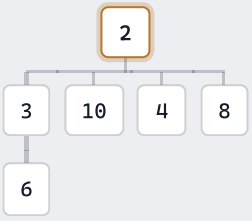
BINOMIAL HEAP



FIBONACCI HEAP



PAIRING HEAP



`Binary ops so far: 4` `Binomial ops so far: 7` `Fibonacci ops so far: 4` `Pairing ops so far: 7`

EXTRACT-MIN — all four correctly return **1**, leaving the same set {2,3,4,6,8,10}. Binary: move last element (4) to the root, shrink, sift down — swaps past 2, then stops (4 ≤ both children) — 1 swap. Binomial: removing root 1 orphans its 2 children [8, {2,[4]}]; these plus the 2 remaining roots [10, {3,[6]}] consolidate: degree-0 pair (8,10) links, degree-1 pair ({2,[4]},{3,[6]}) links — 2 links. Fibonacci: removing root 1 (a singleton, no children) leaves 6 singleton roots to consolidate via the degree array — going from 6 roots down to the 2 final trees takes exactly 4 link events (each link removes one root), the SAME final shape as Lecture 19's method. Pairing: root 1 had only ONE child (the tree rooted at 2) — with just one orphan there is nothing to pair, it is promoted directly — 0 melds, the best case of the two-pass scheme.

Final tally after all 8 operations: Binary 4 swaps (+ an O(n)-touch rebuild), Binomial 7 links, Fibonacci 4 links, Pairing 7 links.

Fewest total pointer-rewires: Fibonacci heap, by a wide margin. That is not the whole story.

STRUCTURE	TOTAL STRUCTURAL OPS ACROSS ALL 8 OPERATIONS	WHERE THE COST CONCENTRATED
Binary heap	4 swaps (+ an O(n)-touch rebuild on merge)	The merge step, structurally — every non-leaf position had to be inspected even though only 1 swap resulted.
Binomial heap	7 links	Spread evenly: 3 during inserts (the "carry" at insert 15), 2 at decrease-key, 2 at the extract-min consolidation.
Fibonacci heap	4 links	All of it deferred to the one EXTRACT-MIN — every insert, the merge, and this particular decrease-key cost exactly zero, because the node happened to already be a root.
Pairing heap	7 links	Spread across every operation — pairing heap never defers, it pays a small O(1) cost immediately, every time.

WHY THIS IS NOT A PROOF OF ANYTHING

This is **one small example** with n = 7 at its largest. Fibonacci heap's near-zero count here is partly luck: the decrease-key target happened to already be a root, so no cut was needed at all — with a different heap history (one where that node were buried three levels deep and marked), Fibonacci heap would show real cut-and-cascade activity too. Binary heap's low swap count similarly hides that its *merge* step touches Θ(n) positions regardless of how many actually move — a cost that gets dramatically worse as n grows, unlike the other three. The complexity table in Section 04 is the trustworthy source for what happens at scale; this animation's job was only to make the mechanics tangible enough to trust that table.

Which heap, and why

YOU NEED A SIMPLE, BOUNDED, IN-MEMORY PRIORITY QUEUE

Use a **binary heap**. No merge, no decrease-key-heavy workload, cache-friendly array layout, smallest constant factors of the four. This is what almost every standard library's default priority queue actually is.

YOU NEED TO REPEATEDLY MERGE WHOLE QUEUES

Use a **binomial heap** if you specifically want a clean, well-understood, worst-case O(log n) merge with a textbook binary-counter analogy (Lecture 18's hospital-merger scenario) — or a **pairing heap** if you want O(1) actual-cost merges and don't need the binomial heap's more rigid degree structure.

YOU'RE RUNNING DIJKSTRA'S OR PRIM'S ALGORITHM AT SCALE, AND IT'S DECREASE-KEY-BOUND

Fibonacci heap is the textbook answer — it is the reason Dijkstra's algorithm has an O(E + V log V) bound at all. But per Lecture 20's honest caveat: most production graph libraries use a **pairing heap** or even a plain binary heap here, because Fibonacci heap's constant-factor overhead rarely pays off before graphs get very large.

YOU WANT THE CLOSEST THING TO "JUST WORKS, FAST, IN PRACTICE"

Use a **pairing heap**. It is the structure real systems reach for when they need merge and decrease-key to be fast without paying Fibonacci heap's per-node bookkeeping cost — accepting, in exchange, that DECREASE-KEY's tightest amortized bound remains an open research question rather than a closed proof.

The theoretical champion is rarely the one deployed

BINARY HEAP

C++'s `std::priority_queue` and Python's `heapq` module are both plain array-based binary heaps — by far the most-used priority queue implementation in existence, precisely because most real workloads never need an efficient merge.

BINOMIAL HEAP

Common in purely functional programming languages (Haskell, Standard ML), where the forest-of-trees structure supports efficient *persistent* (immutable, structure-sharing) merges — a natural fit for a language where nothing is mutated in place.

FIBONACCI HEAP

Mostly appears in algorithms textbooks and research code as the structure that gives Dijkstra's, Prim's, and network-flow algorithms their best theoretical bounds. Rarely the production implementation, exactly as flagged in Lecture 19 — the per-node overhead outweighs the asymptotic win at everyday graph sizes.

PAIRING HEAP

The practical choice in graph libraries and competitive-programming implementations that specifically need decrease-key — chosen over Fibonacci heap precisely because it is simpler to implement correctly and faster in measured practice, the trade Lecture 20 built the whole lecture around.

Same contract, four different bets about when to pay

- **Binary heap** pays eagerly, every time, from a flat array — cheap and cache-friendly, except MERGE.
- **Binomial heap** fixes MERGE by going to a forest shaped like a binary counter — O(log n) merge, still eager decrease-key.
- **Fibonacci heap** fixes DECREASE-KEY by going lazy — defers almost everything to the next EXTRACT-MIN, at the cost of the richest node structure of the four.
- **Pairing heap** keeps Fibonacci's practical speed with the leanest node of all — three pointers, no parent/degree/mark field — at the cost of one still-open proof.
- Verified today on one shared 8-operation sequence: Binary heap 4 swaps (+ an O(n)-touch merge), Binomial heap 7 links, Fibonacci heap 4 links, Pairing heap 7 links — a concrete, small-scale illustration of the complexity table, not a substitute for it.

LECTURE 22 — NEXT

Introduction to Spatial Data Structures

CO CSE3144.4 — multidimensional data representation, setting up k-d Trees, Quad Trees, and R-Trees.

HOMEWORK — BRING TO LECTURE 22

- Build all four heaps by inserting 5, 40, 12, 35, 18 in that order (matching today's structure, different keys). Draw each final structure and count the structural operations for each.
- In 3–4 sentences: why is Binary heap's UNION fundamentally O(n) while the other three can all do better — tie your answer to the array-vs-forest distinction from Axis 2.
- Explain, in your own words, why Fibonacci heap's low operation count in today's animation was partly a matter of luck (which node happened to be decreased) rather than a guarantee — then describe a decrease-key call on today's final Fibonacci heap that WOULD trigger a cut.

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

> *Next: Lecture 22 — Introduction to Spatial Data Structures.*
