# Tournament Trees, Buffering & Run Generation

Three engineering upgrades that turn last lecture's external merge sort into the industrial-strength algorithm inside every database.

---

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

**40 minutes**
Session outcome: explain tournament trees and optimal merge techniques

40 MIN TOTAL

# Today, minute by minute

### Three unpaid bills from Lecture 4 00–04

Fast min-of-k, buffers that never stall, runs longer than memory.

### Winner trees: knockout tournaments as data structures 04–09

n players, n−1 matches, replay one path in log k.

### Animation: a winner tree drives a 4-way merge 09–16

Watch every replay; count every comparison.

### Loser trees: the professional's version 16–21

Store the loser, halve the comparisons. Heap vs tournament.

### Buffering: why merges stall, and double buffering 21–27

Animation: fixed buffers hit a wall.

### Floating buffers, traced end to end 27–32

Animation: the textbook 2-run example, all 8 lines.

### Run generation: replacement selection 32–38

Animation: Lecture 4's 13 records give 3 runs instead of 5.

### Recap + homework 38–40

The complete industrial external sort, assembled.

# Lecture 4 left three bills on the table

We proved that raising k (merging k runs at once) cuts passes to $\log_k m$ and saves real disk I/O. But we ended with two catches, and quietly ignored a third problem. Today pays all three bills:

BILL 1 — CPU PER RECORD

### Find the min of k fronts, fast

Outputting one record means finding the smallest among k run-fronts. Scanning costs k−1 comparisons per record — at k = 1000, that's 999 comparisons for every single record. We need the next minimum in **log k**. Answer: **tournament trees**.

BILL 2 — IDLE HARDWARE

### Keep disk and CPU busy at once

Disk reads, disk writes, and in-memory merging are three *different* pieces of hardware. With the minimum k+1 buffers, whenever the output buffer is being written the merge must halt — the CPU twiddles its thumbs at disk speed. Answer: **double & floating buffers**.

BILL 3 — SHORT RUNS

### Make runs longer than memory

Phase 1 produced runs of exactly M records — memory-sized. Fewer, longer runs mean a smaller m and fewer passes. Can a memory of M records emit runs *longer* than M? Astonishingly yes, ~2M on average. Answer: **replacement selection**.

**Desk analogy, continued.** Bill 1: with 1000 sorted bundles open, you don't want to glance at all 1000 top sheets to pick the smallest — you want a knockout bracket. Bill 2: while the office boy carries a finished stack to the store room, you shouldn't stop working. Bill 3: as you remove sheets from the desk, new sheets keep arriving — many can join the *current* sorted bundle instead of waiting for the next one.

# A knockout tournament, frozen into a tree

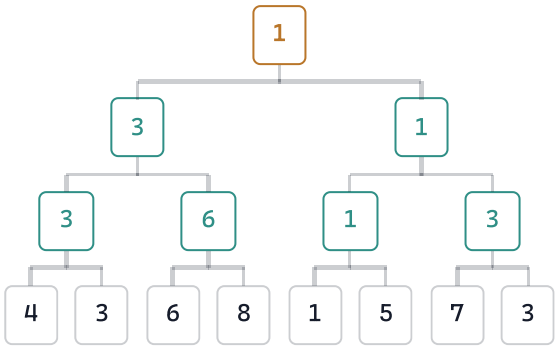
THE STRUCTURE

Think IPL playoffs or Wimbledon. A **winner tree** for n players is a **complete binary tree** with:

- **n external nodes** (leaves) — the players. For us: the current front record of each of the k runs.
- **n − 1 internal nodes** — the matches. Each stores the **winner** of its two children. We want the minimum, so *smaller wins*: a **min winner tree**.
- The **root holds the overall winner** — the smallest of all k fronts, readable in O(1).

Height (excluding the player level) = **$\log_2 n$**. Building it plays every match once: n − 1 matches, **O(n)**.

EXAMPLE — 8 PLAYERS, BUILT BOTTOM-UP




PLAYERS 4 3 6 8 1 5 7 3 (BOTTOM ROW) → QUARTER-FINALS → SEMI-FINALS → CHAMPION 1 — EVERY CONNECTING LINE IS A PLAYED MATCH

Read it like a sports bracket: 4 vs 3 → 3 wins; 6 vs 8 → 6; 1 vs 5 → 1; 7 vs 3 → 3. Semis: 3 vs 6 → 3; 1 vs 3 → 1. Final: 3 vs 1 → **champion 1**.

THE KEY MOVE — REPLACE THE WINNER AND REPLAY ONE PATH

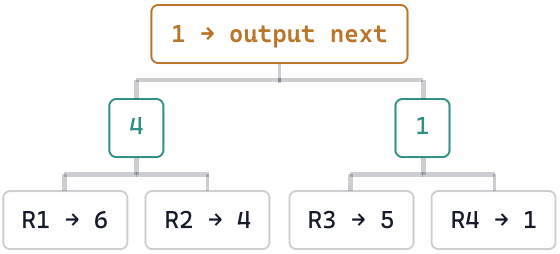
When champion 1 is removed (output to the merge) and its leaf gets a new value, who might the new champion be? **Only players on the path from that leaf to the root** — every other match's result is untouched. So we replay just $\log_2 n$ matches, not n − 1. That's the whole magic: **initialize O(n) once, then every extraction costs Θ(log n)**. Sorting with it: n extractions × log n = O(n log n) — this is "tournament sort."

# A winner tree drives a 4-way merge

Four sorted runs — R1: `6, 8` · R2: `4, 9` · R3: `5, 7` · R4: `1, 3` — merged into one output. The leaves hold the current front of each run; an exhausted run's leaf becomes ∞ (a sentinel that loses every match). Legend: **amber** = changed this step (new leaf + replayed path) · **teal** = root, the next output.

INTERACTIVE – EXTRACT, REPLACE, REPLAY

STEP 1 OF 9




ROOT (CHAMPION) · MATCH NODES · LEAVES = CURRENT RUN FRONTS – THE CONNECTING LINES ARE THE MATCHES

STILL INSIDE THE RUNS

R1: 8 | R2: 9 | R3: 7 | R4: 3

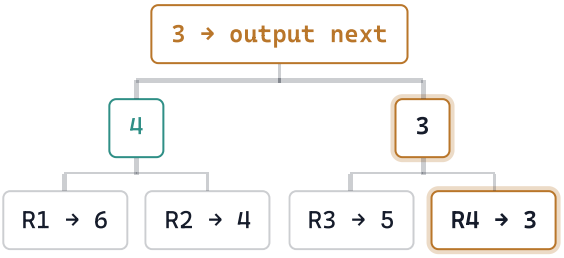
MERGED OUTPUT

– empty –

**Build once, O(k):** play all 3 matches. 6 vs 4 → 4 · 5 vs 1 → 1 · final 4 vs 1 → champion **1**. The root is always the smallest of all four fronts.

---

STEP 2 OF 9




ROOT (CHAMPION) · MATCH NODES · LEAVES = CURRENT RUN FRONTS – THE CONNECTING LINES ARE THE MATCHES

STILL INSIDE THE RUNS

R1: 8 | R2: 9 | R3: 7 | R4: –

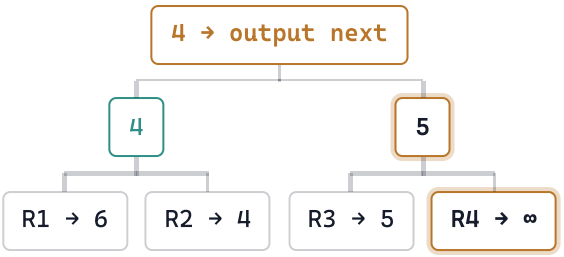
MERGED OUTPUT

1

Output **1**; R4 sends its next record **3** into that leaf. Replay only that leaf's path: 5 vs 3 → 3, then 4 vs 3 → 3. **2 comparisons** — the left half of the tree was never touched.

---

STEP 3 OF 9




ROOT (CHAMPION) · MATCH NODES · LEAVES = CURRENT RUN FRONTS — THE CONNECTING LINES ARE THE MATCHES

STILL INSIDE THE RUNS

R1: 8 | R2: 9 | R3: 7 | R4: —

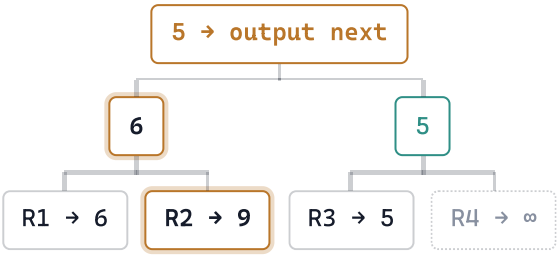
MERGED OUTPUT

1 3

Output **3**; R4 is exhausted → its leaf becomes the sentinel **∞**, which loses every match forever. Replay: 5 vs ∞ → 5, then 4 vs 5 → 4. Again 2 comparisons.

---

STEP 4 OF 9




ROOT (CHAMPION) · MATCH NODES · LEAVES = CURRENT RUN FRONTS — THE CONNECTING LINES ARE THE MATCHES

STILL INSIDE THE RUNS

R1: 8 | R2: — | R3: 7 | R4: —

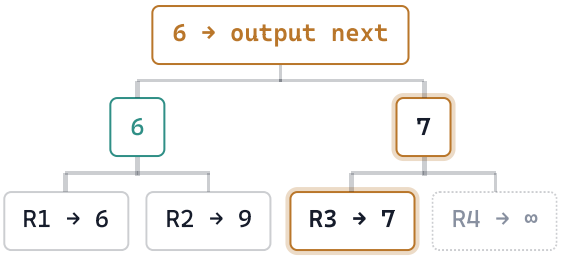
MERGED OUTPUT

1 3 4

Output **4** (it lived in R2's leaf); R2 sends **9**. Replay the LEFT path this time: 6 vs 9 → 6, then 6 vs 5 → 5.

---

STEP 5 OF 9




ROOT (CHAMPION) · MATCH NODES · LEAVES = CURRENT RUN FRONTS — THE CONNECTING LINES ARE THE MATCHES

STILL INSIDE THE RUNS

R1: 8 | R2: — | R3: — | R4: —

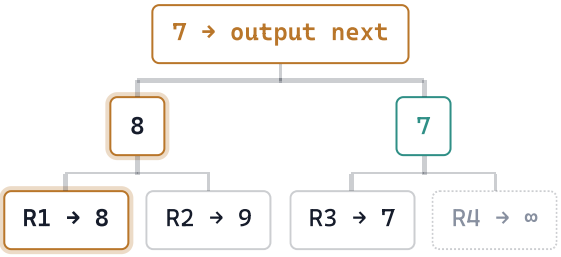
MERGED OUTPUT

1 3 4 5

Output **5**; R3 sends **7**. Replay: 7 vs ∞ → 7, then 6 vs 7 → 6.

---

STEP 6 OF 9




ROOT (CHAMPION) · MATCH NODES · LEAVES = CURRENT RUN FRONTS — THE CONNECTING LINES ARE THE MATCHES

STILL INSIDE THE RUNS

R1: — | R2: — | R3: — | R4: —

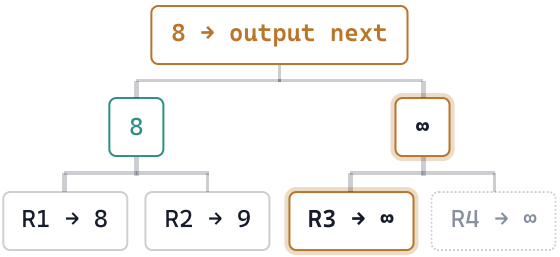
MERGED OUTPUT

1 3 4 5 6

Output **6**; R1 sends **8**. Replay: 8 vs 9 → 8, then 8 vs 7 → 7. Every run is now on its last record.

---

STEP 7 OF 9




ROOT (CHAMPION) · MATCH NODES · LEAVES = CURRENT RUN FRONTS — THE CONNECTING LINES ARE THE MATCHES

STILL INSIDE THE RUNS

R1: – | R2: – | R3: – | R4: –

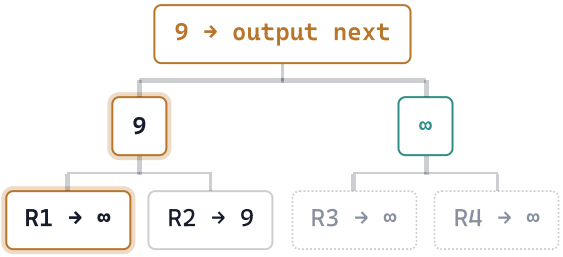
MERGED OUTPUT

1 3 4 5 6 7

Output **7**; R3 exhausted → ∞. Replay: ∞ vs ∞ → ∞, then 8 vs ∞ → 8.

---

STEP 8 OF 9




ROOT (CHAMPION) · MATCH NODES · LEAVES = CURRENT RUN FRONTS — THE CONNECTING LINES ARE THE MATCHES

STILL INSIDE THE RUNS

R1: – | R2: – | R3: – | R4: –

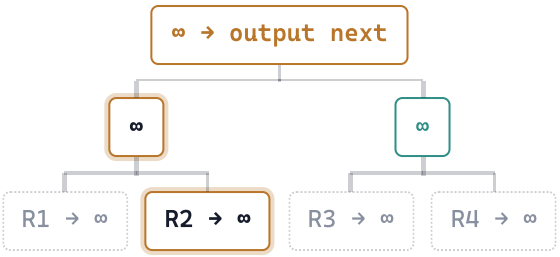
MERGED OUTPUT

1 3 4 5 6 7 8

Output **8**; R1 exhausted → ∞. Replay: ∞ vs 9 → 9, then 9 vs ∞ → 9.

---

STEP 9 OF 9




ROOT (CHAMPION) · MATCH NODES · LEAVES = CURRENT RUN FRONTS — THE CONNECTING LINES ARE THE MATCHES

STILL INSIDE THE RUNS

R1: – | R2: – | R3: – | R4: –

MERGED OUTPUT

1 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Output **9**; all leaves are ∞ — the root is ∞, which signals **merge complete**. Final tally: 8 records × 2 comparisons each = 16, versus 8 × 3 = 24 by scanning. At k = 1024: 10 vs 1023 per record.

**Count what you just saw:** 8 records output, and each extraction replayed exactly $\log_2 4$ = **2 matches**. A naive scan would compare k − 1 = 3 fronts per record. At k = 1024 the gap is 10 comparisons vs 1023 — **per record, across billions of records**. Internal merge time becomes $O(n \log_2 k)$ per pass × $\log_k m$ passes = $O(n \log_2 m)$, **independent of k** — exactly the claim from Lecture 4's "Catch 2."

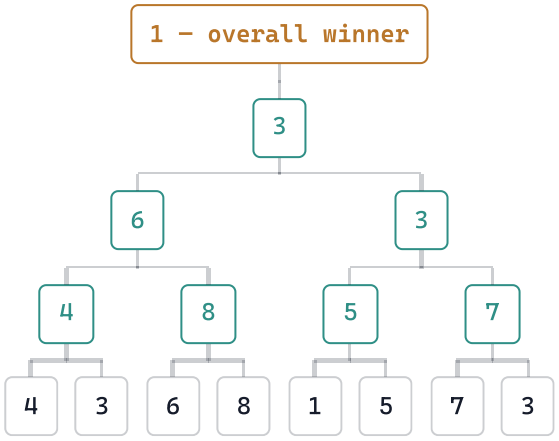
# Store the loser, halve the work

THE SUBTLE INEFFICIENCY IN REPLAY

Replaying a match in a winner tree needs the new value's **opponent**. But who is the opponent at each internal node? It is **the player who lost the last match played there** — the winner moved up and away! In a winner tree you must look at a node's *sibling subtree* to find that loser.

So flip the storage: let each match node store the **loser**, and keep the overall winner in one extra node above the root. Now, walking up from the changed leaf, **your opponent is sitting right there in the node**: one comparison, write the new loser, carry the winner up. Same Θ(log n) asymptotics, but fewer memory accesses and exactly **one comparison per level**.

SAME 8 PLAYERS, AS A MIN LOSER TREE




MATCH NODES HOLD LOSERS; THE CHAMPION 1 SITS IN THE EXTRA BOX ABOVE THE ROOT

Check one node: the final was 3 vs 1 → the root stores the **loser 3**; winner 1 goes on top. When 1's leaf is replaced, its replay partners (5's node, 3's node…) are exactly the values stored on its path.

| FINDING MIN OF K FRONTS, PER RECORD | COMPARISONS | WHY |
|---|---|---|
| Linear scan | $k - 1$ | look at everything |
| Binary min-heap | $\approx 2 \log_2 k$ | sift-down compares **both children** at every level, root to bottom |
| **Loser tree** | $\log_2 k$ | bottom-up, opponent pre-stored: **one** comparison per level |

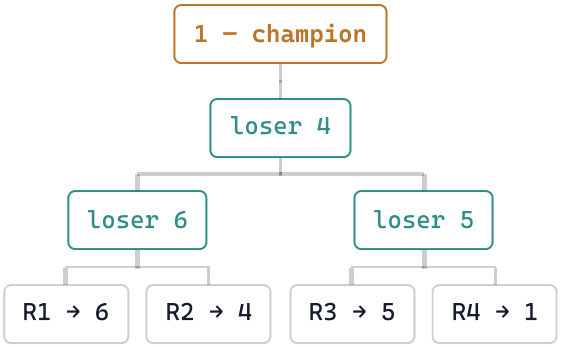
This is why the tournament (loser) tree is the preferred implementation over the heap for k-way merging — same O(n log k), half the constant. The run-generation algorithm of Walters, Painter & Zalk (this lecture's finale) is built on a loser tree too.

## The same merge, driven by a loser tree

Identical runs, identical extraction order as L5·03's winner-tree animation — R1: `6, 8` · R2: `4, 9` · R3: `5, 7` · R4: `1, 3` — so every step can be compared directly. Watch which node changes at each level, and notice how often the displayed loser value stays put even while the tree keeps working correctly underneath it.

INTERACTIVE — EXTRACT, REPLACE, REPLAY (LOSER TREE)

STEP 1 OF 9




CHAMPION (EXTRA BOX) · MATCH NODES STORE THE LOSER · LEAVES = CURRENT RUN FRONTS

STILL INSIDE THE RUNS

R1: 8 | R2: 9 | R3: 7 | R4: 3

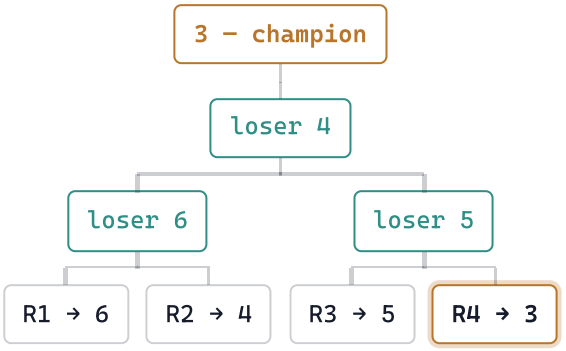
MERGED OUTPUT

— empty —

**Build once, O(k):** each match stores its LOSER. R1 vs R2: 6 vs 4 → loser **6**. R3 vs R4: 5 vs 1 → loser **5**. Root compares the two WINNERS (4 and 1): 4 vs 1 → loser **4**. The overall winner, 1, has nowhere to live inside the tree — it sits in the extra champion box above the root.

---

STEP 2 OF 9




CHAMPION (EXTRA BOX) · MATCH NODES STORE THE LOSER · LEAVES = CURRENT RUN FRONTS

STILL INSIDE THE RUNS

R1: 8 | R2: 9 | R3: 7 | R4: —

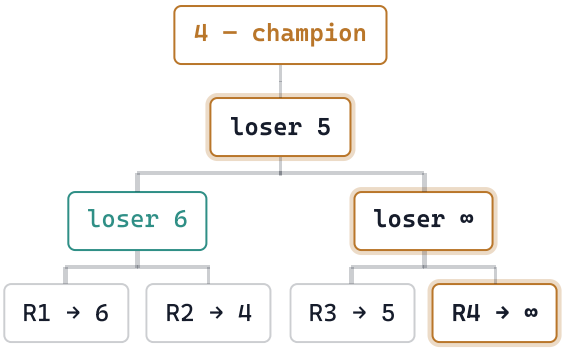
MERGED OUTPUT

1

Output **1**; R4 sends **3**. Replay R4's path, ONE node per level: at mid1, compare new value 3 against the loser already sitting there (5) — 3 wins, 5 remains the loser, nothing to overwrite. At root, compare 3 against the stored loser (4) — 3 wins again, 4 remains loser. New champion: **3**. Notice: zero detours to a sibling node — every comparison used a value already at the node we walked through.

---

STEP 3 OF 9




CHAMPION (EXTRA BOX) · MATCH NODES STORE THE LOSER · LEAVES = CURRENT RUN FRONTS

STILL INSIDE THE RUNS

R1: 8 | R2: 9 | R3: 7 | R4: —

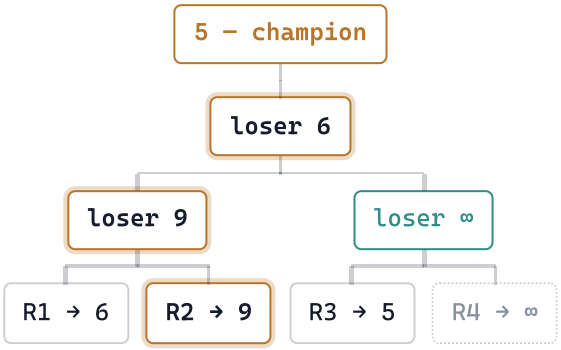
MERGED OUTPUT

1 | 3

Output **3**; R4 exhausted → ∞. At mid1: ∞ vs stored loser 5 → now 5 wins (5 < ∞), so mid1's stored value CHANGES: 5 → ∞. That change in mid1's winner (3 → 5) ripples up: at root, 5 vs stored loser 4 → 4 wins, root changes: 4 → 5. New champion: **4**.

---

STEP 4 OF 9




CHAMPION (EXTRA BOX) · MATCH NODES STORE THE LOSER · LEAVES = CURRENT RUN FRONTS

STILL INSIDE THE RUNS

R1: 8 | R2: — | R3: 7 | R4: —

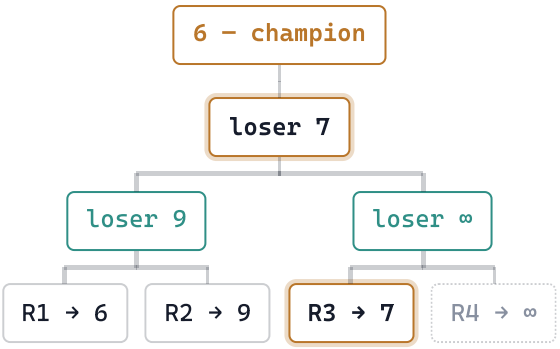
MERGED OUTPUT

1 | 3 | 4

Output **4** (it lived in R2's leaf); R2 sends **9**. Replay the LEFT path this time: at mid0, 9 vs stored loser 6 → 6 wins, mid0 CHANGES: 6 → 9. At root, 6 vs stored loser 5 → 5 wins, root changes: 5 → 6. New champion: **5**.

---

STEP 5 OF 9




CHAMPION (EXTRA BOX) · MATCH NODES STORE THE LOSER · LEAVES = CURRENT RUN FRONTS

STILL INSIDE THE RUNS

R1: 8 | R2: – | R3: – | R4: –

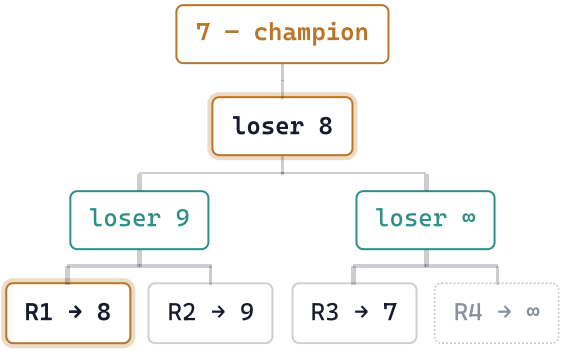
MERGED OUTPUT

1 3 4 5

Output **5**; R3 sends **7**. At mid1: 7 vs stored loser ∞ → 7 wins, mid1 stays ∞ (still the loser, unchanged) — but mid1's WINNER quietly changed underneath (5 → 7), even though the displayed loser didn't move. At root: 7 vs stored loser 6 → 6 wins, root changes: 6 → 7. New champion: **6**.

---

STEP 6 OF 9




CHAMPION (EXTRA BOX) · MATCH NODES STORE THE LOSER · LEAVES = CURRENT RUN FRONTS

STILL INSIDE THE RUNS

R1: – | R2: – | R3: – | R4: –

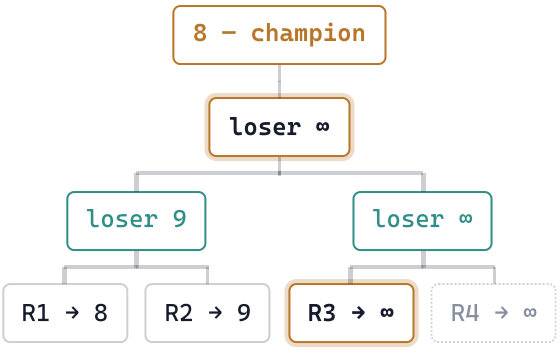
MERGED OUTPUT

1 3 4 5 6

Output **6**; R1 sends **8**. At mid0: 8 vs stored loser 9 → 8 wins, mid0 stays **9** (still the loser, unchanged — R2's 9 keeps losing to whatever R1 sends). At root: 8 vs stored loser 7 → 7 wins, root changes: 7 → 8. New champion: **7**.

---

STEP 7 OF 9




CHAMPION (EXTRA BOX) · MATCH NODES STORE THE LOSER · LEAVES = CURRENT RUN FRONTS

STILL INSIDE THE RUNS

R1: — | R2: — | R3: — | R4: —

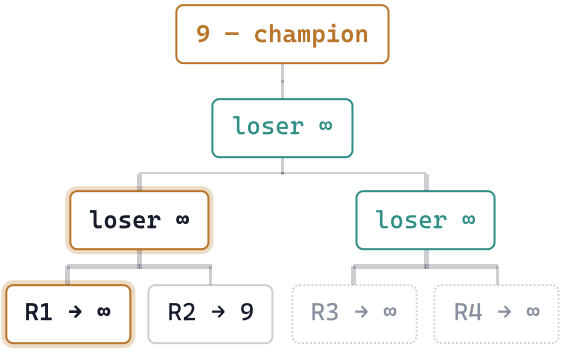
MERGED OUTPUT

1 | 3 | 4 | 5 | 6 | 7

Output **7**; R3 exhausted → ∞. At mid1: ∞ vs ∞ → tie, stays ∞. At root: 8 vs mid1's winner ∞ → 8 wins, root changes: 8 → ∞. New champion: **8**.

---

STEP 8 OF 9




CHAMPION (EXTRA BOX) · MATCH NODES STORE THE LOSER · LEAVES = CURRENT RUN FRONTS

STILL INSIDE THE RUNS

R1: — | R2: — | R3: — | R4: —

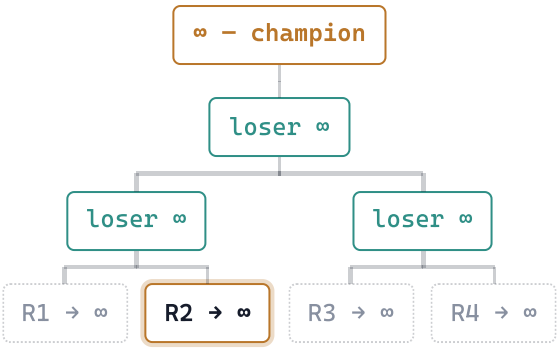
MERGED OUTPUT

1 | 3 | 4 | 5 | 6 | 7 | 8

Output **8**; R1 exhausted → ∞. At mid0: ∞ vs stored loser 9 → now 9 wins (9 < ∞), mid0 changes: 9 → ∞. At root: 9 vs stored loser ∞ → 9 wins, root stays ∞ (unchanged). New champion: **9**.

---

STEP 9 OF 9




CHAMPION (EXTRA BOX) · MATCH NODES STORE THE LOSER · LEAVES = CURRENT RUN FRONTS

STILL INSIDE THE RUNS

R1: – | R2: – | R3: – | R4: –

MERGED OUTPUT

1 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Output **9**; R2 exhausted → ∞. All leaves are ∞ — champion is ∞, signalling **merge complete**. Same count as the winner tree: 8 records × $\log_2 4$ = 2 comparisons = 16 total — the loser tree never does FEWER comparisons, it does the SAME comparisons with HALF the memory accesses (one node touched per level instead of two).

# Bill 2: the merge that keeps stopping

WHY I/O AND CPU CAN OVERLAP AT ALL

Every animation so far has treated a "read" or "write" as one instantaneous thing. In reality, disk transfers use **DMA (Direct Memory Access)**: the CPU tells the disk controller "copy this block to/from this memory address," then walks away — the controller moves the bytes in the background and interrupts the CPU only when done. So disk-in, disk-out, and CPU-merge really are **three independent workers** running at once, not a simplification for this course. That's what the animation on the next slide shows explicitly.

But this only pays off if each worker **always has somewhere to work**. With just the minimum k + 1 buffers:

- → While the output buffer is being **written to disk**, merged records have nowhere to go → **merge halts**. Fix: **two output buffers** — merge into one while the other writes.
- → While a run's input buffer is being **refilled**, if the merge needs that run's next record → **merge halts**. Fix: read ahead into a *second* buffer for that run.

So aim for **2k input + 2 output = 2k + 2 buffers**. But there's a trap: fixing two buffers *per run* still fails — watch it happen →

THE FIX: FLOATING BUFFERS

Don't marry buffers to runs. Keep all 2k input buffers in a common **pool (stack)**; each run owns a **queue** of the buffers currently holding its data — at least one each. After every block arrives, ask: **which run will run dry first?** Answer: the run whose *last loaded key* (lastKey) is smallest — its data is being consumed fastest. Pop a free buffer from the pool and start reading *that* run's next block. The buffer "floats" to wherever the need is. Sentinel +∞ blocks mark the end of each run.

# Fixed buffers hit the wall (2-way merge)

ANIMATION — FIXED BUFFERS HIT THE WALL (2-WAY MERGE)

Run 0 begins `1, 3, 5, 7, 8, 9…` · Run 1 begins `2, 4, 6, 15, 20, 25…` Buffers hold 2 records. in[0], in[2] are fixed to Run 0; in[1], in[3] to Run 1. Three independent workers run at once — **CPU merge (amber)**, **disk READ (teal)**, **disk WRITE (violet)** — and the strip at the top of every step shows all three simultaneously: ○ **idle** (faded), ● **busy** (colored), or ✓ **done** (colored + ring), so you can see exactly what every worker is doing at every single moment. No two events are ever compressed into one step — even when two workers finish at nearly the same instant, each gets its own step.

STEP 1 OF 14

CPU · MERGE
● BUSY
Merging in[0](1,3) + in[1](2,4) → building ou[0]

DMA · READ
● BUSY
Reading Run 0's block 2 [5,7] → in[2]

DMA · WRITE
○ IDLE
Nothing to write yet

INPUT BUFFERS (FIXED: IN[0], IN[2] → RUN 0 · IN[1], IN[3] → RUN 1)

`in[0]·R0: 1 3` `in[1]·R1: 2 4` `in[2]·R0: —` `in[3]·R1: —`

OUTPUT BUFFERS

`ou[0]: —` `ou[1]: —`

WRITTEN TO DISK

— nothing yet —

**Start:** one block from each run. CPU begins merging into ou[0]; independently (DMA — see the card above), the disk begins reading Run 0's next block [5 7] into in[2] (Run 0's lastKey 3 < Run 1's 4, so Run 0 is predicted to drain first).

STEP 2 OF 14

CPU · MERGE
✓ DONE
ou[0] = 1, 2

DMA · READ
● BUSY
Still reading in[2] (started last step)

DMA · WRITE
● BUSY
Just started: writing ou[0] (1,2) to disk

INPUT BUFFERS

`in[0]·R0: 3` `in[1]·R1: 4` `in[2]·R0: … reading` `in[3]·R1: —`

OUTPUT BUFFERS

`ou[0]: 1 2` `ou[1]: —`

WRITTEN TO DISK

— nothing yet —

**CPU worker:** finishes merging — ou[0] = **1 2**. This triggers a WRITE of ou[0] to begin in the background. Meanwhile the READ of Run 0's [5 7] into in[2], started at step 1, is still in flight — two independent clocks are now ticking.

STEP 3 OF 14

CPU · MERGE

● BUSY

Now merging in[0]'s 3 vs in[1]'s 4 → building ou[1]

DMA · READ

✓ DONE

in[2] = 5, 7 → immediately starts reading Run 1's next block [6,15] into in[3]

DMA · WRITE

● BUSY

Still writing ou[0] (started last step)

INPUT BUFFERS

`in[0]·R0: 3` `in[1]·R1: 4` `in[2]·R0: 5 7` `in[3]·R1: … reading`

OUTPUT BUFFERS

`ou[0]: 1 2 … writing` `ou[1]: –`

WRITTEN TO DISK

`– nothing yet –`

**Disk-read worker:** the READ lands — in[2] = **5 7**. The read channel is free again, so it immediately starts fetching Run 1's next block [6 15] into in[3]. Notice ou[0]'s WRITE (triggered last step) is still going — the disk is running two jobs, one finishing as another begins. And now that in[0]/in[1] both have data again, the CPU doesn't wait — it starts merging toward ou[1] right away.

---

STEP 4 OF 14

CPU · MERGE

✓ DONE

ou[1] = 3, 4 — in[0] and in[1] both empty now

DMA · READ

● BUSY

Still reading in[3] (started last step)

DMA · WRITE

● BUSY

Still writing ou[0]

INPUT BUFFERS

`in[0]·R0: –` `in[1]·R1: –` `in[2]·R0: 5 7` `in[3]·R1: … reading`

OUTPUT BUFFERS

`ou[0]: 1 2 … writing` `ou[1]: 3 4`

WRITTEN TO DISK

`– nothing yet –`

**CPU worker:** drains the last of in[0] (3) and in[1] (4) — ou[1] = **3 4**. Both runs' FIRST buffers are now empty; the merge will read straight from in[2]/in[3] from here on. ou[0]'s write and in[3]'s read are both still in flight underneath this.

---

STEP 5 OF 14

CPU · MERGE

○ IDLE

Waiting — needs Run 1's next block (in[3]) before it can pair against in[2]'s 5

DMA · READ

✓ DONE

in[3] = 6, 15 → immediately starts reading Run 0's next block [8,9] into in[0]

DMA · WRITE

● BUSY

Still writing ou[0] (started 3 steps ago)

INPUT BUFFERS

`in[0]·R0: … reading` `in[1]·R1: –` `in[2]·R0: 5 7` `in[3]·R1: 6 15`

OUTPUT BUFFERS

`ou[0]: 1 2 … writing` `ou[1]: 3 4`

WRITTEN TO DISK

– nothing yet –

**Disk-read worker:** the READ lands — in[3] = **6 15**. The read channel is free again, so it immediately starts fetching Run 0's next block [8 9] into in[0]. ou[0]'s WRITE (started 3 steps ago) is still going underneath, completely unaffected. But notice the CPU: it had nothing to do this whole time — in[2] had a 5 ready, but with in[3] empty there was no partner to compare it against. That's a real, visible CPU stall, just a brief one.

---

STEP 6 OF 14

CPU · MERGE

● BUSY

in[3] just arrived — now merging in[2]'s 5 vs in[3]'s 6 → building ou[0]

DMA · READ

● BUSY

Still reading in[0] (started last step)

DMA · WRITE

✓ DONE

1, 2 written to disk → ou[0] free → immediately starts writing ou[1] (3,4) next

INPUT BUFFERS

`in[0]·R0: … reading` `in[1]·R1: –` `in[2]·R0: 5 7` `in[3]·R1: 6 15`

OUTPUT BUFFERS

`ou[0]: –` `ou[1]: 3 4 … writing`

WRITTEN TO DISK

`1 2`

**Disk-write worker:** the WRITE lands — **1 2** joins the sorted output file, and ou[0] is free again. Since ou[1] already has 3, 4 waiting (produced 3 steps ago), the write channel immediately picks it up next. And with in[3] now holding data, the CPU — idle last step — resumes at once, comparing in[2]'s 5 against in[3]'s 6.

---

STEP 7 OF 14

CPU · MERGE

✓ DONE

ou[0] = 5, 6

DMA · READ

● BUSY

Still reading in[0] (started 2 steps ago)

DMA · WRITE

● BUSY

Still writing ou[1] (3,4)

INPUT BUFFERS

`in[0]·R0: … reading` `in[1]·R1: –` `in[2]·R0: 7` `in[3]·R1: 15`

OUTPUT BUFFERS

`ou[0]: 5 6` `ou[1]: 3 4 … writing`

WRITTEN TO DISK

`1 2`

**CPU worker:** merges in[2]'s 5 against in[3]'s 6 (in[3]'s value wins the second comparison too) — ou[0] = **5 6**. in[0]'s read (Run 0's [8 9], started last step) and ou[1]'s write (of 3 4) are both still in progress.

---

STEP 8 OF 14

CPU · MERGE

● BUSY

Building ou[1]: takes in[2]'s last value 7 (Run 0's 7 beats Run 1's 15) — now waiting on in[0] for Run 0's next value

DMA · READ

✓ DONE

in[0] = 8, 9 → Run 0 now holds 3 buffered records (7,8,9) vs Run 1's 1 (15) → the read channel switches to Run 1, starts fetching its next block [20,25] into in[1]

DMA · WRITE

● BUSY

Still writing ou[1] (started last step)

INPUT BUFFERS

`in[0]·R0: 8 9` `in[1]·R1: … reading` `in[2]·R0: 7` `in[3]·R1: 15`

OUTPUT BUFFERS

`ou[0]: 5 6` `ou[1]: 3 4 … writing`

WRITTEN TO DISK

`1 2`

**Disk-read worker:** the READ lands — in[0] = **8 9**. Now check: Run 0 already holds 7, 8, 9 — **three** buffered records — while Run 1 holds only 15 — **one**. Run 1 is the one running low now, so the read channel starts fetching Run 1's next block [20 25] into in[1] instead of refilling Run 0 again. Meanwhile the CPU has already taken in[2]'s 7 into ou[1], and is about to take in[0]'s freshly-arrived 8 next.

---

STEP 9 OF 14

CPU · MERGE

● BUSY

Still building ou[1] — about to take in[0]'s 8 next

DMA · READ

● BUSY

Still reading in[1] (started last step)

DMA · WRITE

✓ DONE

3, 4 written to disk → ou[1] free → immediately starts writing ou[0] (5,6) next, waiting since the CPU finished it 2 steps ago

INPUT BUFFERS

`in[0]·R0: 8 9` `in[1]·R1: … reading` `in[2]·R0: 7` `in[3]·R1: 15`

OUTPUT BUFFERS

`ou[0]: 5 6 … writing` `ou[1]: –`

WRITTEN TO DISK

`1 2` `3 4`

**Disk-write worker:** the WRITE lands — **3 4** joins the sorted file, ou[1] is free. ou[0] (5,6) has been sitting ready since the CPU finished it 2 steps ago, so the write channel picks it up immediately.

---

STEP 10 OF 14

CPU · MERGE

✓ DONE

ou[1] = 7, 8 — in[2] now fully drained (no read queued for it yet — the read channel has been busy with in[1] since 2 steps ago)

DMA · READ

● BUSY

Still reading in[1] (Run 1's block [20,25])

DMA · WRITE

● BUSY

Still writing ou[0] (5,6)

INPUT BUFFERS

`in[0]·R0: 9` `in[1]·R1: … reading` `in[2]·R0: –` `in[3]·R1: 15`

OUTPUT BUFFERS

`ou[0]: … writing` `ou[1]: 7 8`

WRITTEN TO DISK

`1 2` `3 4`

**CPU worker:** merges in[2]'s last value 7, then in[0]'s 8 (in[3]'s 15 loses both comparisons) — ou[1] = **7 8**. in[2] is now fully drained — and no read has been queued for it yet, because the read channel has been busy with in[1] since last step. ou[0]'s write (of 5 6) is also still going.

---

STEP 11 OF 14

CPU · MERGE

○ IDLE

Both output buffers are occupied — ou[1] just completed, ou[0] still being written — nothing free to merge into yet

DMA · READ

✓ DONE

in[1] = 20, 25 → only NOW does the read channel turn back to Run 0 — in[2]'s refill starts, LATE

DMA · WRITE

● BUSY

Still writing ou[0] (started last step)

INPUT BUFFERS

`in[0]·R0: 9` `in[1]·R1: 20 25` `in[2]·R0: … reading (just started)` `in[3]·R1: 15`

OUTPUT BUFFERS

`ou[0]: … writing` `ou[1]: 7 8`

WRITTEN TO DISK

`1 2` `3 4`

**Disk-read worker:** the READ lands — in[1] = **20 25**. Only now does the read channel turn back to Run 0 — in[2]'s refill starts, *late*. Meanwhile Run 0 has just ONE buffered record left anywhere (9 in in[0]), and the CPU — with both output buffers full — has nothing to do but wait.

---

STEP 12 OF 14

CPU · MERGE

● BUSY

ou[0] just freed — begins merging in[0]'s last value (9) into it

DMA · READ

● BUSY

Still reading in[2] (just started last step) — Run 0 has only 1 buffered record left anywhere (9, in in[0])

DMA · WRITE

✓ DONE

5, 6 written to disk → ou[0] free → immediately starts writing ou[1] (7,8) next

INPUT BUFFERS

`in[0]·R0: 9` `in[1]·R1: 20 25` `in[2]·R0: … still reading` `in[3]·R1: 15`

OUTPUT BUFFERS

`ou[0]: –` `ou[1]: 7 8 … writing`

WRITTEN TO DISK

`1 2` `3 4` `5 6`

**Disk-write worker:** the WRITE lands — **5 6** joins the sorted file, ou[0] is free. ou[1] (7,8) has been waiting since the CPU finished it 2 steps ago, so the write channel picks it up immediately. And the CPU, finally freed, grabs Run 0's last buffered value (9) and starts filling ou[0] with it.

---

STEP 13 OF 14

CPU · MERGE

✓ DONE

ou[0] = 9 — one slot short of full. Run 0's NEXT record must come from in[2], whose refill (started 2 steps ago) is nowhere close to done

DMA · READ

● BUSY

Still reading in[2] — Run 0's block 4

DMA · WRITE

● BUSY

Still writing ou[1] (7,8)

INPUT BUFFERS

in[0]·R0: –

in[1]·R1: 20 25

in[2]·R0: … still reading

in[3]·R1: 15

OUTPUT BUFFERS

ou[0]: 9 (only 1 of 2 – waiting)

ou[1]: … writing

WRITTEN TO DISK

1 2

3 4

5 6

**CPU worker:** outputs 9 — the last record Run 0 had buffered anywhere. ou[0] holds just 9, one slot short of full. The merge needs Run 0's *next* record to continue — but in[2]'s read (Run 0's block 4), which only just started two steps ago, is nowhere close to done.

---

STEP 14 OF 14

CPU · MERGE

○ IDLE

BLOCKED — nothing to compare against for Run 0; cannot output anything else until in[2]'s read completes

DMA · READ

● BUSY

Still reading in[2] — far from done

DMA · WRITE

✓ DONE

7, 8 written to disk — the write channel is now idle too, with nothing new to write

INPUT BUFFERS

in[0]·R0: –

in[1]·R1: 20 25

in[2]·R0: … still reading

in[3]·R1: 15

OUTPUT BUFFERS

ou[0]: 9 …stuck

ou[1]: –

WRITTEN TO DISK

1 2

3 4

5 6

7 8

**STALL.** ou[1]'s write finishes — 7 8 joins the sorted file — but in[2]'s read is still far from done. The merge has nothing to compare against for Run 0 and cannot output anything else until that read completes. **Two fixed buffers per run were not enough:** the read channel spent its attention on Run 1 right when Run 0 needed it most — a fixed per-run assignment can never redirect a spare buffer to wherever the real need is. Hence: floating buffers.

## The textbook trace, line by line

Two runs on disk, blocks of 2 records, sentinel +∞ ending each run — Run 1: `[2 4] [6 8] [9 10] [+∞]`, Run 2: `[3 5] [7 16] [21 26] [+∞]`. Four floating buffers in the pool, two output buffers. Each step: merge two records to the output buffer, write it out, and — in parallel — load the next block from the run whose lastKey is minimum.

INTERACTIVE – THE FULL 8-LINE TRACE

STEP 1 OF 8

RUN 1 QUEUE (LASTKEY DRIVES PREDICTION)

2 4

RUN 2 QUEUE

3 5

OUTPUT BUFFER (JUST MERGED)

–

WRITTEN TO DISK SO FAR

– nothing yet –

NEXT BLOCK WILL BE READ FROM

Run 1 (lastKey 4 < 5)

**Line 1:** pop two buffers from the pool, load the first block of each run. lastKey(R1)=4, lastKey(R2)=5 → R1 will starve first → prefetch R1's next block.

---

STEP 2 OF 8

RUN 1 QUEUE (LASTKEY DRIVES PREDICTION)

4 | 6 8

RUN 2 QUEUE

5

OUTPUT BUFFER (JUST MERGED)

2 3

WRITTEN TO DISK SO FAR

– nothing yet –

NEXT BLOCK WILL BE READ FROM

Run 2 (lastKey 5 < 8)

**Line 2:** merge the two smallest (2, 3) into the output buffer and write it; meanwhile [6 8] arrived on R1's queue. Now lastKey(R2)=5 is the minimum → prefetch R2.

---

STEP 3 OF 8

RUN 1 QUEUE (LASTKEY DRIVES PREDICTION)

6 8

RUN 2 QUEUE

7 16

OUTPUT BUFFER (JUST MERGED)

4 5

WRITTEN TO DISK SO FAR

2 3

NEXT BLOCK WILL BE READ FROM

Run 1 (lastKey 8 < 16)

**Line 3:** merge 4, 5 out; [7 16] arrived for R2; R1's emptied buffer returns to the pool. lastKey(R1)=8 < 16 → prefetch R1.

---

STEP 4 OF 8

RUN 1 QUEUE (LASTKEY DRIVES PREDICTION)

8 9 10

RUN 2 QUEUE

16

OUTPUT BUFFER (JUST MERGED)

6 7

WRITTEN TO DISK SO FAR

2 3 4 5

NEXT BLOCK WILL BE READ FROM

Run 1 (lastKey 10 < 16)

**Line 4:** merge 6, 7; [9 10] arrived for R1 — note R1 now holds TWO buffers while R2 holds one: the pool floats to where consumption is fastest. Still R1: 10 < 16.

---

STEP 5 OF 8

RUN 1 QUEUE (LASTKEY DRIVES PREDICTION)

10 | +∞

RUN 2 QUEUE

16

OUTPUT BUFFER (JUST MERGED)

8 9

WRITTEN TO DISK SO FAR

2 3 | 4 5 | 6 7

NEXT BLOCK WILL BE READ FROM

Run 2 (16 < ∞)

**Line 5:** merge 8, 9; R1's **sentinel block +∞** arrived — R1 has no real data left coming. lastKey(R1)=∞, so prediction switches to R2.

---

STEP 6 OF 8

RUN 1 QUEUE (LASTKEY DRIVES PREDICTION)

+∞

RUN 2 QUEUE

21 26

OUTPUT BUFFER (JUST MERGED)

10 16

WRITTEN TO DISK SO FAR

2 3 | 4 5 | 6 7 | 8 9

NEXT BLOCK WILL BE READ FROM

Run 2 (26 < ∞)

**Line 6:** merge 10, 16; [21 26] arrived for R2. R1's queue is just the sentinel now.

---

STEP 7 OF 8

RUN 1 QUEUE (LASTKEY DRIVES PREDICTION)

+∞

RUN 2 QUEUE

+∞

OUTPUT BUFFER (JUST MERGED)

21 26

WRITTEN TO DISK SO FAR

2 3 | 4 5 | 6 7 | 8 9 | 10 16

NEXT BLOCK WILL BE READ FROM

no next — both runs ended

**Line 7:** merge 21, 26; R2's sentinel arrives too. Every future comparison is ∞ vs ∞ — nothing left to load.

---

STEP 8 OF 8

RUN 1 QUEUE (LASTKEY DRIVES PREDICTION)

+∞

RUN 2 QUEUE

+∞

OUTPUT BUFFER (JUST MERGED)

+∞ (sentinel reached)

WRITTEN TO DISK SO FAR

2 3 | 4 5 | 6 7 | 8 9 | 10 16 | 21 26

NEXT BLOCK WILL BE READ FROM

no next — both runs ended

**Line 8:** the sentinel is merged into the output — the signal to **terminate**. Final file: 2 3 4 5 6 7 8 9 10 16 21 26. Throughout, reading, writing, and merging overlapped — no stalls, because buffers floated to the hungriest run.

**What to notice:** the prediction rule (load for the run with the *smallest* lastKey) means a buffer is always arriving just before it is needed — input, output, and merging overlap almost perfectly. When merge time ≈ block read time ≈ block write time, the disk never waits for the CPU and the CPU never waits for the disk.

# Bill 3: runs longer than memory — replacement selection

THE INTUITION (DESK VERSION)

Lecture 4's Phase 1 was: fill the desk, sort, ship the whole bundle, repeat — runs of exactly M. But notice: the moment you ship the *smallest* sheet, its spot is free and a **new sheet arrives immediately**. If the new sheet's number is **larger than what you just shipped**, it can still join the *current* sorted bundle! Only when it's *smaller* must it wait (**frozen ✱**) for the next bundle.

So: keep a pool of M records (in practice, a **loser tree** — Walters, Painter & Zalk). Repeatedly output the smallest **unfrozen** record, refill from input, freeze arrivals smaller than the last output. When everything is frozen, close the run, thaw, continue. Runs come out ≈ **2M long on average** (Knuth's "snowplow" argument), so m halves, and passes = $\log_k$ m shrink.

ANIMATION — LECTURE 4'S 13 RECORDS, M = 3, SAME DATA NEW TRICK

STEP 1 OF 15

INPUT STREAM STILL ON DISK (→ ARRIVES LEFT TO RIGHT)

96 12 35 17 99 28 58 41 75 15

MEMORY, M = 3 (* = FROZEN FOR THE NEXT RUN)

81 94 11

RUN 1 (BEING WRITTEN)

— empty —

COMPLETED RUNS

— none yet —

**Setup:** fill memory with the first M = 3 records (81, 94, 11). The rest of the file will arrive one record at a time, as space frees up.

---

STEP 2 OF 15

INPUT STREAM STILL ON DISK (→ ARRIVES LEFT TO RIGHT)

96 12 35 17 99 28 58 41 75 15

MEMORY, M = 3 (* = FROZEN FOR THE NEXT RUN)

81 94 96

RUN 1 (BEING WRITTEN)

11

COMPLETED RUNS

— none yet —

Smallest in memory is **11** → output it to Run 1. Its slot is refilled by the next arrival, **96**. Is 96 ≥ 11? Yes → 96 can still join Run 1. Nothing frozen.

---

STEP 3 OF 15

INPUT STREAM STILL ON DISK (→ ARRIVES LEFT TO RIGHT)

96 12 35 17 99 28 58 41 75 15

MEMORY, M = 3 (* = FROZEN FOR THE NEXT RUN)

12 * 94 96

RUN 1 (BEING WRITTEN)

11 81

COMPLETED RUNS

— none yet —

Output **81** (smallest unfrozen). In comes **12** — but 12 < 81, the last record shipped! 12 can never appear after 81 in a sorted run → **frozen** * for the next run.

---

STEP 4 OF 15

INPUT STREAM STILL ON DISK (→ ARRIVES LEFT TO RIGHT)

96 12 35 17 99 28 58 41 75 15

MEMORY, M = 3 (* = FROZEN FOR THE NEXT RUN)

12 * 35 * 96

RUN 1 (BEING WRITTEN)

11 81 94

COMPLETED RUNS

– none yet –

Output **94**. In comes **35** < 94 → frozen. Memory is filling with frozen records — the current run is running out of eligible players.

---

STEP 5 OF 15

INPUT STREAM STILL ON DISK (→ ARRIVES LEFT TO RIGHT)

96 12 35 17 99 28 58 41 75 15

MEMORY, M = 3 (* = FROZEN FOR THE NEXT RUN)

12 * 35 * 17 *

RUN 1 (BEING WRITTEN)

11 81 94 96

COMPLETED RUNS

– none yet –

Output **96**. In comes **17** < 96 → frozen. Now **every record is frozen** → Run 1 must close. Look at it: **length 4, from a memory of 3** — already longer than Lecture 4 could ever produce.

---

STEP 6 OF 15

INPUT STREAM STILL ON DISK (→ ARRIVES LEFT TO RIGHT)

96 12 35 17 99 28 58 41 75 15

MEMORY, M = 3 (* = FROZEN FOR THE NEXT RUN)

12 35 17

RUN 2 (BEING WRITTEN)

– empty –

COMPLETED RUNS

Run 1: 11 81 94 96

**Thaw.** Run 1 is sealed on disk. The frozen records (12, 35, 17) wake up as the starting memory for Run 2. The cycle restarts.

---

STEP 7 OF 15

INPUT STREAM STILL ON DISK (→ ARRIVES LEFT TO RIGHT)

96 12 35 17 99 28 58 41 75 15

MEMORY, M = 3 (* = FROZEN FOR THE NEXT RUN)

99 35 17

RUN 2 (BEING WRITTEN)

12

COMPLETED RUNS

Run 1: 11 81 94 96

Output **12**; in comes **99** ≥ 12 → stays active.

---

STEP 8 OF 15

INPUT STREAM STILL ON DISK (→ ARRIVES LEFT TO RIGHT)

96 12 35 17 99 28 58 41 75 15

MEMORY, M = 3 (* = FROZEN FOR THE NEXT RUN)

99 35 28

RUN 2 (BEING WRITTEN)

12 17

COMPLETED RUNS

Run 1: 11 81 94 96

Output **17**; in comes **28** ≥ 17 → active.

---

STEP 9 OF 15

INPUT STREAM STILL ON DISK (→ ARRIVES LEFT TO RIGHT)

96 12 35 17 99 28 58 41 75 15

MEMORY, M = 3 (* = FROZEN FOR THE NEXT RUN)

99 35 58

RUN 2 (BEING WRITTEN)

12 17 28

COMPLETED RUNS

Run 1: 11 81 94 96

Output **28**; in comes **58** → active. Notice the input is being kind: most arrivals are large enough to keep joining.

---

STEP 10 OF 15

INPUT STREAM STILL ON DISK (→ ARRIVES LEFT TO RIGHT)

96 12 35 17 99 28 58 41 **75** **15**

MEMORY, M = 3 (* = FROZEN FOR THE NEXT RUN)

**99** **41** **58**

RUN 2 (BEING WRITTEN)

**12** **17** **28** **35**

COMPLETED RUNS

**Run 1: 11 81 94 96**

Output **35**; in comes **41** → active.

---

STEP 11 OF 15

INPUT STREAM STILL ON DISK (→ ARRIVES LEFT TO RIGHT)

96 12 35 17 99 28 58 41 75 **15**

MEMORY, M = 3 (* = FROZEN FOR THE NEXT RUN)

**99** **75** **58**

RUN 2 (BEING WRITTEN)

**12** **17** **28** **35** **41**

COMPLETED RUNS

**Run 1: 11 81 94 96**

Output **41**; in comes **75** → active.

---

STEP 12 OF 15

INPUT STREAM STILL ON DISK (→ ARRIVES LEFT TO RIGHT)

96 12 35 17 99 28 58 41 75 15

MEMORY, M = 3 (* = FROZEN FOR THE NEXT RUN)

**99** **75** **15 ***

RUN 2 (BEING WRITTEN)

**12** **17** **28** **35** **41** **58**

COMPLETED RUNS

**Run 1: 11 81 94 96**

Output **58**; in comes **15** < 58 → frozen. That was the **last record of the input file**.

---

STEP 13 OF 15

INPUT STREAM STILL ON DISK (→ ARRIVES LEFT TO RIGHT)

96 12 35 17 99 28 58 41 75 15

MEMORY, M = 3 (* = FROZEN FOR THE NEXT RUN)

99 · 15 *

RUN 2 (BEING WRITTEN)

12 17 28 35 41 58 75

COMPLETED RUNS

Run 1: 11 81 94 96

Output **75**. No input left to refill — the slot stays empty.

---

STEP 14 OF 15

INPUT STREAM STILL ON DISK (→ ARRIVES LEFT TO RIGHT)

96 12 35 17 99 28 58 41 75 15

MEMORY, M = 3 (* = FROZEN FOR THE NEXT RUN)

· · 15 *

RUN 2 (BEING WRITTEN)

12 17 28 35 41 58 75 99

COMPLETED RUNS

Run 1: 11 81 94 96

Output **99**. Only the frozen 15 remains → Run 2 closes at **length 8 — from a memory of 3!**

---

STEP 15 OF 15

INPUT STREAM STILL ON DISK (→ ARRIVES LEFT TO RIGHT)

96 12 35 17 99 28 58 41 75 15

MEMORY, M = 3 (* = FROZEN FOR THE NEXT RUN)

· · ·

RUN 3

15

COMPLETED RUNS

Run 1: 11 81 94 96 Run 2: 12 17 28 35 41 58 75 99

**Final tally:** 3 runs (lengths 4 + 8 + 1) instead of Lecture 4's 5 runs — so 2-way merging needs $\lceil \log_2 3 \rceil = 2$ passes instead of 3. One full read+write of the file, saved before merging even began. In practice the "smallest unfrozen" is found with a loser tree over M records: O(log M) per record.

| PHASE 1 METHOD (SAME 13 RECORDS, M = 3) | RUNS PRODUCED | 2-WAY MERGE PASSES = ⌈LOG₂ M⌉ |
|---|---|---|
| Fill–sort–ship (Lecture 4) | 5 runs (3+3+3+3+1) | 3 passes |
| **Replacement selection** | **3 runs (4+8+1)** | **2 passes** — a full read+write of the file saved |

A run of length **8 from a memory of 3** — because the input happened to arrive in a friendly order. Best case: input already sorted → **one single run**, regardless of M. Worst case: input reverse-sorted → runs of exactly M, no worse than Lecture 4. Average: 2M. Bonus: the whole thing runs with parallel input/output, like everything else today.

# The industrial external sort, fully assembled

- **Winner/loser trees:** min of k fronts in $\log_2 k$ — one comparison per level (loser tree beats the heap's 2 log k). Initialize O(k), replay Θ(log k).
- **Double + floating buffers:** 2k + 2 buffers, pooled, dispatched to the run with the smallest lastKey — disk-in, disk-out, and CPU merge overlap fully.
- **Replacement selection:** a loser tree over M records emits runs averaging 2M — fewer runs, fewer passes, less I/O before merging even starts.

Pipeline: `replacement selection → long runs` → `k-way merge via loser tree` → `floating buffers keep everything busy`. That, plus Lecture 4's pass-counting, is the algorithm inside Unix sort and every database spill.

BONUS APPLICATION

Tournament trees aren't only for merging: **bin packing / truck loading** by First Fit runs in O(n log n) using a *max* winner tree where players are bins and values are remaining capacities — find the leftmost bin that fits in log n. Same structure, different game.

LECTURE 6

## Huffman Trees & Optimal Merging

Runs now come out *unequal* (4, 8, 1…). In what order should we merge unequal runs? The lone "15" question from Lecture 4, answered optimally — weighted external path length.

HOMEWORK — BRING TO LECTURE 6

- Build the min **winner** tree for 4 3 6 8 1 5 7 3 2 6 9 4 5 2 5 8; replace the winner with 6 and replay (show each match).
- Build the min **loser** tree for the same 16 keys; replace the winner with 9 and replay. Count comparisons in both exercises.
- Run **replacement selection** with M = 4 on: 50 3 27 9 41 12 88 2 60 35 7 91. How many runs? What are their lengths?
- Trace a **3-run floating-buffer** merge exactly like today's animation (blocks of 2 records, six floating buffers, sentinel +∞ ending each run): Run1 = [22 27][28 30][31 32], Run2 = [25 31][36 38][40 62], Run3 = [26 30][33 35][42 45]. Produce the full line-by-line table: each run's queue, the output block, and which run the lastKey rule prefetches.
- For k = 8 runs and n total records, count comparisons per record for linear scan, heap, and loser tree.
- Reading: Weiss §7.11 (external sorting and multiway merging).

### Questions?

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

> *Next: Lecture 6 — Huffman Trees and Optimal Merging of Runs.*