CSE3144 · Advanced Data Structures · Jul–Nov Semester 2026 · Lecture 3 of 36 · CO CSE3144.1

Amortized Analysis Techniques

Aggregate, Accounting, and Potential methods — proving that "occasionally expensive" can still mean "cheap on average," with no probability involved.

Dr. Manu Shrivastava

Course Instructor · Consultation Fri 2–5 PM, LHC 308F

40 minutes

Session outcome: apply aggregate, accounting, and potential methods

Today, minute by minute

Meet the running examples 00–07

Multipop stack & binary counter from zero — full traces, and their naive $O(n^2)$ / $O(nk)$ bounds.

Why those naive bounds lie 07–12

Two structural reasons the "n × worst cost" habit overshoots.

What amortized analysis is — and is not 12–16

Worst-case over a *sequence*; no probability. How it relates to Big-O.

Method 1 · Aggregate 16–22

Total cost T(n), divide by n. Applied to the multipop stack and binary counter.

Method 2 · Accounting — plus the charging refinement 22–31

Overcharge cheap operations, bank the credit, spend it on expensive ones — then the 2-3 tree case where that alone isn't enough.

Method 3 · Potential 31–37

Credit becomes a function Φ of the structure's state. The telescoping trick.

The payoff: dynamic arrays 37–41

Why vector.push_back / ArrayList.add is O(1) — by all three methods.

Recap, where this returns, practice 41–43

Splay trees, Fibonacci heaps, and Union-Find all lean on today.

Example A: the Multipop Stack, from zero

REFRESHER — WHAT IS A STACK? (FROM CS 1301)

A pile of plates in the mess. You can only touch the **top**: place a plate on top (**PUSH**) or take the top plate off (**POP**). Last In, First Out — **LIFO**. Both touch one plate, so each costs **1 unit of work**.

The **new** operation today: **MULTIPOP(S, k)** — "pop up to k elements in one call."

```
MULTIPOP(S, k)
  while not EMPTY(S) and k ≠ 0
      POP(S)        // 1 unit per element removed
      k ← k − 1
```

Read the loop guard carefully — it stops when **either** k elements are popped **or** the stack runs empty, whichever comes first. So its cost is:

$$\text{cost of MULTIPOP(S, k)} = \min(|S|, k)$$

Example: MULTIPOP(S, 5) on a stack holding only 2 elements pops just 2 and stops. Cost = min(2, 5) = 2 — **not** 5.

FULL TRACE — 6 OPERATIONS, EVERY STEP SHOWN

#	OPERATION	STACK AFTER (TOP → BOTTOM)	COST	Σ
1	PUSH(10)	10	1	1
2	PUSH(20)	20, 10	1	2
3	PUSH(30)	30, 20, 10	1	3
4	POP() → 30	20, 10	1	4
5	PUSH(40)	40, 20, 10	1	5
6	MULTIPOP(3)	*empty*	3	8

Step 6 unrolled, iteration by iteration: k=3, pop 40 (k becomes 2) → pop 20 (k becomes 1) → pop 10 (k becomes 0, loop exits). Three pops, cost 3.

Total: 8 units for 6 operations — average 1.33 per operation, even though one operation alone cost 3. Hold that thought.

THE NAIVE WORST-CASE BOUND — DERIVED STEP BY STEP

- Question: what can a sequence of **n** mixed PUSH / POP / MULTIPOP operations cost in total?
- Step 1 — worst cost of *one* operation: a MULTIPOP. How big can min(|S|, k) get? The stack can hold at most n elements (only n operations happened, each PUSH adds one). So one MULTIPOP costs at most **n**.
- Step 2 — multiply: n operations × worst cost n each = **$O(n^2)$** total.
- Step 3 — smell test: for even *one* MULTIPOP to cost n, **all n earlier operations must have been PUSHes** — and then the sequence is over! The n operations cannot all be expensive MULTIPOPs. **You can't pop what was never pushed.** $O(n^2)$ is correct but wildly loose.

Example B: the k-bit Binary Counter, from zero

REFRESHER — COUNTING IN BINARY

A 4-bit counter is an array A[3] A[2] A[1] A[0] of 0s and 1s. A[0] is the **ones place**, A[1] the twos, A[2] the fours, A[3] the eights:

$$\text{value} = 8 \cdot A[3] + 4 \cdot A[2] + 2 \cdot A[1] + 1 \cdot A[0]$$

Counting works like a car odometer. In decimal, 199 + 1: the 9 rolls to 0 and **carries**, the next 9 rolls to 0 and carries, then 1 becomes 2 → 200. Binary is identical, except digits roll over at 1 instead of 9: `0111 + 1 = 1000`.

```
INCREMENT(A)   // cost = number of bits flipped
  i ← 0
  while i < k and A[i] = 1
      A[i] ← 0            // this 1 rolls over: carry
      i ← i + 1
  if i < k then A[i] ← 1  // carry lands here
```

In words: walk up from A[0], turning every 1 you meet into 0 (the carry rippling), and when you first meet a 0, turn it into 1 and stop. **Cost = number of bits you touched.**

FULL TRACE — INCREMENT ON 0111 (VALUE 7)

LOOP STEP	LOOKING AT	FOUND	ACTION	COUNTER NOW
i = 0	A[0]	1	set to 0, carry on	0110
i = 1	A[1]	1	set to 0, carry on	0100
i = 2	A[2]	1	set to 0, carry on	0000
i = 3	A[3]	0	loop exits; set A[3] ← 1	**1000**

Result: 0111 → 1000, which is 7 → 8. ✓ Four bits were flipped, so **this increment cost 4** — the worst possible on 4 bits.

Now increment again, 1000 → 1001 (8 → 9): the loop looks at A[0], finds 0, exits immediately, sets A[0] ← 1. **Cost 1.** The expensive increment zeroed the low bits — and made its successors cheap. Hold this thought: it becomes "Reason 1" on the next slide.

THE NAIVE WORST-CASE BOUND — DERIVED STEP BY STEP

- Question: what do **n** INCREMENTs cost in total, starting from all zeros?
- Step 1 — worst cost of *one* increment: all k bits flip (as in 0111 → 1000). So one INCREMENT costs at most **k**.
- Step 2 — multiply: n increments × k each = **O(nk)** total.
- Step 3 — smell test: flipping all k bits requires the counter to read 0111…1 — and immediately afterwards the low bits are all 0 again. A[0] flips on *every* increment, but A[1] only on every 2nd, A[2] on every 4th, A[3] on every 8th… **Most increments touch one or two bits.** O(nk) is correct but loose.

Both naive bounds are honest O-bounds — the sin is looseness, not error. The three methods of this lecture will each prove the tight answer for both structures: **O(1) amortized per operation, O(n) total.**

The worst-case habit that overcharges you

So far in your courses you've bounded **one execution of one operation**, then multiplied: *n* operations × worst cost each. That's exactly what we just did on the last two slides — and it gave $O(n^2)$ for the stack and $O(nk)$ for the counter, both of which felt too pessimistic against the traces we computed. The multiplication habit is always **correct**, but it can be wildly **loose**, for two distinct reasons:

REASON 1 — OPERATIONS TALK TO EACH OTHER

An expensive operation leaves the structure in a cheap state

The cost of operation i is not independent of operations 1…i−1: each operation reshapes the structure, and an *expensive* operation usually reshapes it in a way that makes the *following* operations cheap. Multiplying n × worst-cost silently assumes each operation faces a freshly worst-case structure — which the previous expensive operation just destroyed.

EXPENSIVE OPERATION	…AND THE CHEAP STATE IT LEAVES BEHIND
MULTIPOP of 50 elements (cost 50)	Stack is now nearly **empty** — the next MULTIPOP has almost nothing left to pop; it cannot also cost 50.
Counter increment 0111 → 1000 (4 flips)	The carry run **reset the low bits to 0** — each of the next several increments flips just one bit.

REASON 2 — THE WORST CASE NEEDS A RUN-UP

n operations can't all hit their worst case in one sequence

An operation's worst case requires the structure to be in a specific extreme state — and reaching that state **consumes operations from the same budget of n**. The expensive event and its own setup compete for the same sequence, so the expensive event is necessarily rare.

FOR THIS TO COST…	…THE SEQUENCE MUST FIRST SPEND
One MULTIPOP costing n	**n PUSHes** at cost 1 each. So a length-n sequence fits at most *one* such MULTIPOP — never n of them. Naive bound charges n × n; the truth is ≤ 2n.
One increment flipping all k bits	The counter must read 0111…1, which occurs only **once every 2^{k-1} increments**. The worst case is rare by arithmetic, not by luck.

Put together: cheap operations **fund** the expensive ones — pushes fund the multipop, 1-bits fund the carry cascade. Amortized analysis is the bookkeeping that makes this funding explicit and provable.

EVERYDAY ANALOGY

The hostel mess card

You pay ₹3,000 once a month, then eat "free" for 30 days. Is a meal expensive? The *single-payment* view says ₹3,000. The honest view spreads (amortizes) it: ₹100 per day, every day, **guaranteed** — no probability, no luck involved. Amortized analysis does exactly this to operation costs.

WHERE YOU'VE MET THIS ALREADY

Union-Find in Kruskal's MST

Kruskal's does $2|E|$ find operations and $|V|-1$ unions. One find with path compression can be expensive — but it flattens the tree, so the *next* finds on that path are nearly free. Charging every find its worst case ignores this "one pays, many benefit" structure entirely.

Amortized analysis, precisely

$$\text{amortized cost of an operation} = \textbf{worst-case total cost of any sequence of } n \textbf{ operations} \div n$$

It guarantees the average performance of each operation **in the worst case over the sequence** — even when some operations in the sequence are far costlier than others.

"BUT WE ALREADY HAVE BIG-O – ISN'T THIS THE SAME THING?"

No — and the distinction is a favourite exam question. **Asymptotic notation (O, Θ, Ω) is a language for describing how any quantity grows.** It doesn't say *which* quantity you measured. Worst-case, average-case, and amortized are three different **quantities** — three different questions about the same operation — and each answer is then *written* in asymptotic notation:

ANALYSIS	THE QUESTION IT ANSWERS	MEASURED OVER
Worst-case	"How bad can *one single call* be?"	One operation, adversarial state
Average-case	"What does a call cost on a *random* input?"	One operation, probability over inputs
Amortized	"What does each call cost when I run *many* — counting the total, worst sequence possible?"	A whole sequence, no probability

So **"worst-case Θ(n)" and "amortized O(1)" are not contradictory** — they can both be true of the same operation at the same time, because they answer different questions.

ONE OPERATION, THREE TRUE STATEMENTS — VECTOR.PUSH_BACK / ARRAYLIST.ADD

Appending to a dynamic array holding n elements (the array doubles when full — full analysis on slide 11):

STATEMENT	VALUE	WHY
Worst case of a single call	$\Theta(n)$	If this call triggers the resize, it copies all n existing elements.
Best case of a single call	$\Theta(1)$	There's a free slot — write one element, done.
Amortized, over any n calls	$O(1)$	Total work for n appends is provably $< 3n$, so each call's fair share is ≤ 3 — *no matter how adversarial the sequence*.

All three statements use asymptotic notation. All three are simultaneously true. If someone tells you only "push_back is O(n)," they haven't lied — they've answered the least useful of the three questions. When you write a loop with n appends, the amortized answer is the one that predicts your program's real running time: **O(n) total, not $O(n^2)$**.

AMORTIZED ANALYSIS IS

- A **worst-case** guarantee — over any adversarial sequence.
- Deterministic — holds for *every* run, not on average luck.
- An upper bound: total amortized ≥ total actual, always.

IT IS NOT

- **Not** average-case analysis — no probability distribution on inputs.
- **Not** a claim about one operation — a single call may still be slow.
- **Not** a replacement for Big-O — amortized costs are still *expressed* in Big-O; what changes is the quantity being bounded.

Exam trap: "amortized O(1)" and "average-case O(1)" are different claims. Hash-table lookup is average-case O(1) — probabilistic, can be beaten by bad luck or bad inputs. Dynamic-array append is amortized O(1) — deterministic, guaranteed for every sequence. Interviewers and exam setters love asking which is which.

The Aggregate Method

Compute the total cost T(n) of the whole sequence directly, then divide: amortized cost = T(n) / n. Every operation gets the same amortized cost.

> *"Stop asking what one operation costs. Ask what they can possibly cost together."*

Count pops against pushes

- A MULTIPOP is nothing but repeated POPs — so just count total PUSHes and total POPs (direct or inside a MULTIPOP).
- Each element can be popped **at most once**, and only after being pushed.
- With n operations there are at most n PUSHes ⟹ at most n POPs in total, *however they are grouped into MULTIPOPs*.

$$T(n) \le n_{push} + n_{pop} \le 2n \implies \text{amortized cost} = T(n)/n = \mathbf{O(1)}$$

Sanity check with a concrete run (n = 8): PUSH a, PUSH b, PUSH c, PUSH d, MULTIPOP 3, PUSH e, PUSH f, MULTIPOP 3. Actual costs: 1+1+1+1+3+1+1+3 = `12 ≤ 2×8 = 16`. ✓ The expensive MULTIPOP(3) was only possible because three earlier PUSHes each paid cost 1 to set it up.

Count flips per bit, not bits per flip

Flip the accounting: instead of "how many bits does increment i flip," ask "how many times does bit j flip across all n increments?"

BIT	FLIPS WHEN	TOTAL FLIPS
A[0]	every increment	n
A[1]	every 2nd	⌊n/2⌋
A[2]	every 4th	⌊n/4⌋
A[i]	every 2^i-th	$\lfloor n/2^i \rfloor$

$$T(n) = \Sigma_{i=0}^{k-1} \lfloor n/2^i \rfloor < n \cdot \Sigma_{i=0}^{\infty} 1/2^i = \mathbf{2n} \Longrightarrow O(1) \text{ amortized}$$

Trace, 4-bit counter — flipped bits highlighted (**0→1** / **1→0**):

VAL	A3	A2	A1	A0	COST	Σ
1	0	0	0	**1**	1	1
2	0	0	**1**	**0**	2	3
3	0	0	1	**1**	1	4
4	0	**1**	**0**	**0**	3	7
5	0	1	0	**1**	1	8
6	0	1	**1**	**0**	2	10
7	0	1	1	**1**	1	11
8	**1**	**0**	**0**	**0**	4	15

After 8 increments: total 15 < 2×8 = 16. ✓ The costly jump 7→8 (4 flips) was "paid for" by the five cheap 1-flip increments around it.

M2

The Accounting Method

Charge each operation an invented "amortized price" — overcharge the cheap ones, park the surplus as credit on specific objects, and let the credit pay for expensive operations later.

> *"Every element pays for its own funeral at the moment it is born."*

You may invent prices — under two laws

Let c_i be the **actual** cost of operation i and $\hat{c}_i$ your invented **amortized** cost. You are free to choose the $\hat{c}_i$ — the analysis is valid only if:

LAW 1 — SOLVENCY AT THE END

$$\Sigma\, \hat{c}_i \geq \Sigma\, c_i \quad \text{(for the full sequence)}$$

Total amortized must upper-bound total actual — otherwise your "worst case" isn't one.

LAW 2 — SOLVENCY AT EVERY MOMENT

$$\Sigma_{i \leq t}\, \hat{c}_i - \Sigma_{i \leq t}\, c_i \geq 0 \quad \text{for every prefix } t$$

The credit balance can never go negative — you cannot pay today's cost with credit you hope to earn tomorrow.

The skill is in choosing **where to park the credit**: on the pushed element, on the bit that became 1 — on the exact object that will trigger the future expensive work.

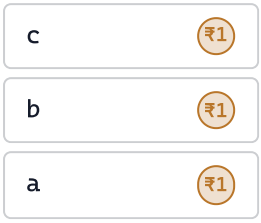
Why invent prices when aggregate already worked? Two reasons, previewed now and proven later in the course: accounting needs only a *local, per-step* check instead of a global sum over the whole sequence (the sum becomes impossible for splay trees), and it can give *different operations different amortized costs* (essential for Fibonacci heaps).

Charge PUSH ₹2: one to work, one to save

c_i is the real, measured cost of operation i; **$\hat{c}_i$** ("c-hat") is the amortized cost we invent for it. Below, ĉ is fixed per *operation type* — every PUSH is invented the same price, every POP the same, and so on, regardless of where it sits in the sequence.

OPERATION	ACTUAL COST C	AMORTIZED COST Ĉ
PUSH	1	2 — overcharged
POP	1	0 — undercharged
MULTIPOP(k)	min(|S|, k)	0 — undercharged

Each PUSH pays ₹1 for its own work and tapes ₹1 onto the element it pushed. When that element is later popped — directly or inside a MULTIPOP — the taped rupee pays for the pop. A MULTIPOP of 40 elements costs 40, and finds exactly 40 rupees waiting on those 40 elements.



after 3 PUSHes — credit ₹3

after MULTIPOP(2) — cost 2 paid by taped coins; credit ₹1

HOW LAW 1 HOLDS — $\Sigma\ \hat{c}_I \geq \Sigma\ c_I$

Sum the invented prices: $\Sigma\ \hat{c}_i = 2 \cdot$ (number of PUSHes), since only PUSH is ever charged anything. Sum the real work: $\Sigma\ c_i$ = (number of PUSHes) + (number of POPs, whether standalone or inside a MULTIPOP). Every popped element was pushed exactly once, so (number of POPs) ≤ (number of PUSHes). Therefore $\Sigma\ c_i \leq 2\cdot$(number of PUSHes) = $\Sigma\ \hat{c}_i$. **Law 1 holds** — total invented cost never falls short of total real cost, for *any* sequence.

HOW LAW 2 HOLDS — THE RUNNING BALANCE NEVER GOES NEGATIVE

At any prefix ending at operation t, $\Sigma_{i\leq t}\ \hat{c}_i - \Sigma_{i\leq t}\ c_i$ is exactly "total rupees taped on so far minus total rupees spent so far" — i.e. the rupees still taped onto elements *currently on the stack*. That equals the current stack size, a count of physical elements, which can never be negative. **Law 2 holds** at every single moment, not just at the very end.

Both laws hold ⇒ ĉ is a valid amortized cost ⇒ every PUSH / POP / MULTIPOP is amortized **O(1)**, no matter how they are mixed or ordered.

A bit pays ₹2 to become 1 — and never pays to fall back

Same notation as before: $\mathbf{c_i}$ is the real cost of flip i (always 1 — one bit changes); $\mathbf{\hat{c}_i}$ is the amortized cost we invent, fixed per flip type below.

FLIP TYPE	ACTUAL COST c	AMORTIZED COST ĉ
0 → 1 (set)	1	2 — ₹1 works, ₹1 sits on the bit
1 → 0 (reset, the carry cascade)	1	0 — paid by the coin sitting on the bit

The entire carry cascade — the expensive part of an increment — is a run of 1→0 resets, and every one of those 1-bits is already carrying a coin. What remains is **at most one** 0→1 set per increment. So each INCREMENT is charged at most ₹2.

HOW LAW 1 HOLDS — $\Sigma\ \hat{c}_I \geq \Sigma\ c_I$

Sum the invented prices: $\Sigma\ \hat{c}_i = 2 \cdot$ (number of 0→1 sets), since only sets are ever charged anything. Sum the real work: $\Sigma\ c_i$ = (number of sets) + (number of resets) — every flip, set or reset, costs 1. A bit can only reset if it was set at some earlier point (it must become 1 before it can become 0 again), so (number of resets) ≤ (number of sets). Therefore $\Sigma\ c_i \leq 2\cdot$(number of sets) $= \Sigma\ \hat{c}_i$. **Law 1 holds** for any sequence of increments.

HOW LAW 2 HOLDS — THE RUNNING BALANCE NEVER GOES NEGATIVE

At any prefix ending at increment t, $\Sigma_{i\leq t}\ \hat{c}_i - \Sigma_{i\leq t}\ c_i$ is exactly "total rupees credited to bits so far minus total rupees spent so far" — i.e. the rupees still sitting on bits that are *currently* 1. That equals the number of 1-bits in the counter right now, a count that can never be negative. **Law 2 holds** at every single moment.

Both laws hold ⇒ ĉ is valid ⇒ INCREMENT is amortized **O(1)**, regardless of the starting value or how many increments run.

Ask yourself (30 seconds, discuss with your neighbour): in the trace on slide 05, the jump 7→8 flipped four bits. Point to the three coins that paid for the three resets. Where were they deposited?

Sometimes a coin taped today isn't enough by the time it's spent

WHERE PLAIN ACCOUNTING BREAKS

Try the accounting trick on a 2-3 tree: insert an element and tape an O(lg n) coin onto it, to be spent when *that same element* is later deleted — identical in spirit to the multipop stack's PUSH-prepays-a-POP move. Here it fails.

By the time the element is deleted, the tree may have grown to $n' > n$. Deleting now costs $O(\lg n')$ — but the coin taped at insertion only covers the smaller, stale $O(\lg n)$. The coin comes up short.

THE FIX — BILL THE DELETE TO A DIFFERENT OPERATION

Don't charge a delete to the insert of the *same* element. Charge it to whichever insert most recently grew the tree to its **current** size n' — a different operation, possibly far away in the sequence.

That insert can be charged at most once: for the tree to reach size n' again after shrinking, another insert has to regrow it. DELETE becomes amortized **O(1)**; INSERT absorbs its own $O(\lg n)$ work plus this one contingent future charge.

NAMING THE DISTINCTION

Accounting method: an operation prepays for its *own* future cost (PUSH prepays its own later POP). **Charging method:** an operation's cost is billed to *some other* operation — past or future — chosen so no single operation is ever billed twice. They're close cousins, and many treatments fold them together, but the 2-3 tree case above is exactly where the distinction earns its keep: "prepay for yourself" fails; "bill whoever caused the trouble" works.

Coming attraction: this same idea — define Φ as "the number of nodes one step from trouble" (a 3-node about to split) — is exactly how **Lecture 11's B-Trees** justify their split costs, and 2-3 trees are literally the smallest B-Tree. Same battery, taught here in miniature.

CSE3144 — Lecture 3 · Accounting method (advanced note)

The Potential Method

Stop taping coins to objects. Store all prepaid work as one number — a potential Φ of the data structure's current state, like potential energy in physics.

> *"The accounting method with the bookkeeping centralized: one bank account, whose balance is a function of the structure's shape."*

Think of it as a battery

THE ANALOGY

Meter at the wall socket, not at the device

An inverter at home: most of the day it **charges** quietly from the mains; during a power cut it **discharges** to run the fans. If you meter consumption *at the fans*, usage is spiky — nothing for hours, then a burst. If you meter *at the wall socket*, the draw is small and steady, because the burst was prepaid into the battery over many quiet hours.

The potential method meters your data structure at the wall socket. **Φ ("phi") = the battery's charge level**. Cheap operations do their small work *plus* a little charging (Φ rises). The expensive operation's burst is powered mostly by discharge (Φ falls). The steady wall-socket reading is the **amortized cost**:

$$\text{amortized} = \text{actual work} + \textbf{change in battery charge}$$

THE BRIDGE FROM THE ACCOUNTING METHOD

We were already doing this — just with coins

Look back at the accounting method's bank balance. At every moment, the total credit could be **read directly off the structure's current shape** — no memory of history needed:

STRUCTURE	COINS PARKED ON…	SO TOTAL CREDIT =
Multipop stack	each stacked element	`|S|` — the stack size
Binary counter	each 1-bit	`b` — the number of 1s

The potential method's only new idea: **skip the coins**. Don't track who holds what — just define the balance directly as a function of the current state, and call it Φ. One number replaces all the bookkeeping.

WATCH IT WORK — NUMBERS BEFORE FORMULAS (Φ = STACK SIZE)

#	OPERATION	ACTUAL COST C	Φ BEFORE	Φ AFTER	ΔΦ	AMORTIZED $\hat{C} = C + \Delta\Phi$
1	PUSH(a)	1	0	1	+1	2
2	PUSH(b)	1	1	2	+1	2
3	PUSH(c)	1	2	3	+1	2
4	MULTIPOP(2)	2	3	1	−2	**0**
Totals		**5**				**6**

- Every PUSH reads 2 at the wall socket: 1 unit of real work + 1 unit of charging.
- The MULTIPOP did 2 units of real work but reads **0** — entirely battery-powered (Φ dropped by exactly 2).
- Total amortized 6 ≥ total actual 5. The spare 1 is not lost — it's the charge still in the battery (final Φ = 1, element *a* still stacked). Overestimating is allowed; that's Law 1 again.

The same idea, in symbols

Everything on the previous slide, formalized in three steps. Operations take the structure through states $D_0 \to D_1 \to \dots \to D_n$ (D_0 is the initial state; D_i is the state after operation i, which had actual cost c_i).

Step 1 — choose a battery gauge. Pick any function Φ that reads a single number off a state (Φ(D) = stack size, Φ(D) = number of 1-bits, …). This is the *only* creative act; the rest is mechanical.

Step 2 — define amortized cost as actual work plus the change in the gauge:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \quad \text{(work + charging, or work – discharge)}$$

Step 3 — sum over the sequence and watch it collapse. Write out the sum for, say, three operations — every intermediate Φ appears once with + and once with −:

$$\hat{c}_1 + \hat{c}_2 + \hat{c}_3 = (c_1 + \mathbf{\Phi(D_1)} - \Phi(D_0)) + (c_2 + \mathbf{\Phi(D_2)} - \mathbf{\Phi(D_1)}) + (c_3 + \Phi(D_3) - \mathbf{\Phi(D_2)})$$

+Φ(D1) kills −Φ(D1), +Φ(D2) kills −Φ(D2) — a **telescoping** sum. Only the two ends survive:

$$\Sigma\, \hat{c}_i = \Sigma\, c_i + \mathbf{\Phi(D_n) - \Phi(D_0)}$$

THE ONE CONDITION — AND WHY

We need $\Sigma\, \hat{c}_i \geq \Sigma\, c_i$ (Law 1). By the boxed equation, that holds exactly when $\mathbf{\Phi(D_n) \geq \Phi(D_0)}$ — and since the sequence can stop at any point, we demand it at every step: $\mathbf{\Phi(D_i) \geq \Phi(D_0) = 0}$ **for all i**. In battery language: *the battery starts empty and may never be borrowed below empty.* A battery that could go negative would let the wall-socket meter understate the true consumption.

READING ΔΦ

ΔΦ > 0 → operation **overcharged**: it did its work and also charged the battery (every PUSH: ĉ = 1 + 1 = 2).

ΔΦ < 0 → operation **undercharged**: the battery discharged to pay most of its real cost (MULTIPOP: ĉ = k′ − k′ = 0).

How do you choose Φ? Rule of thumb: Φ = "how much trouble is stored in the structure right now" — the amount of future expensive work the current state can trigger. Big stack → many pending pops. Many 1-bits → a long carry is brewing. Choose Φ so that *cheap operations raise it a little and expensive operations must drain*

it a lot; then the drain cancels their cost. The Φ you choose **is** the analysis — everything after that is the telescoping formula.

One line of Φ, and both proofs fall out

MULTIPOP STACK — $\Phi = |S|$ (STACK SIZE)

$\Phi(D_0) = 0$ (empty), $\Phi \geq 0$ always. ✓

OP	C	ΔΦ	$\hat{C} = C + \Delta\Phi$
PUSH	1	+1	2
POP	1	−1	0
MULTIPOP	k′	−k′	0

($k' = \min(|S|, k)$.) Every operation is amortized ≤ 2 ⟹ **O(1)**. Note it reproduces the accounting method's prices — with zero creativity about where coins live.

BINARY COUNTER — $\Phi = b$ (NUMBER OF 1-BITS)

$\Phi(D_0) = 0$ (all zeros), $\Phi \geq 0$ always. ✓

Say increment i resets t_i bits (the carry run) and sets at most one bit:

$$c_i \leq t_i + 1$$
$$\Delta\Phi \leq 1 - t_i$$
$$\hat{c}_i = c_i + \Delta\Phi \leq (t_i + 1) + (1 - t_i) = 2$$

The t_i — the entire expensive cascade — **cancels algebraically**. That cancellation is the whole method.

INCREMENT is amortized **O(1)**.

Dynamic arrays: the amortized O(1) you use every day

REFRESHER — WHAT A DYNAMIC ARRAY ACTUALLY DOES

A dynamic array (C++ `vector`, Java `ArrayList`, Python `list`) keeps two numbers: **n** = elements stored, **cap** = slots allocated. Append works like this:

```
APPEND(x)
  if n = cap                      // table full?
      allocate new array of size 2·cap
      copy all n elements across  // cost n
      cap ← 2·cap
  write x into slot n; n ← n+1    // cost 1
```

So an append costs **1** normally, but **n + 1** when it triggers a doubling. Worst case of one call: Θ(n). Yet every language calls append "O(1)" — amortized, and now we can prove it three ways.

WATCH NINE APPENDS — SPIKES GET RARER AS THEY GET BIGGER

APPEND #	1	2	3	4	5	6	7	8	9
Capacity after	1	2	4	4	8	8	8	8	16
Actual cost	1	**2**	**3**	1	**5**	1	1	1	**9**
Running total / 3n line	1/3	3/6	6/9	7/12	12/15	13/18	14/21	15/24	24/27

A cost-2^j spike happens only once every 2^j appends — rarity and size grow at exactly the same rate, so the running total never crosses the 3n ceiling. That "3" is about to appear in all three proofs.

PROOF 1 · AGGREGATE

Total work of n appends = n unit writes + all the copying. Copies happen at sizes 1, 2, 4, 8, … so total copy work = $1 + 2 + 4 + \ldots + 2^{\lfloor \log(n-1) \rfloor} < 2n$ (a geometric sum — each term is more than everything before it combined, and the last term is < n… so the sum is < 2n).

T(n) < n + 2n = 3n ⟹ 3 per append, O(1).

PROOF 2 · ACCOUNTING — WHERE "₹3" COMES FROM

Stand at the moment just after a doubling: capacity 2m, m elements, **all savings spent**. Before the next doubling, exactly m more appends arrive. Charge each **₹3**: ₹1 writes it, ₹2 banked. At the next doubling, 2m elements must be copied, and the bank holds m × ₹2 = ₹2m — **exactly enough**. Each new element's ₹2 copies *itself* plus *one old element* whose savings died in the previous doubling.

PROOF 3 · POTENTIAL — BOTH CASES VERIFIED

Φ = 2n − cap (≥ 0: right after a doubling the table is exactly half full, and it only fills from there).

Normal append: c = 1; n rises by 1, cap unchanged → ΔΦ = +2; **ĉ = 3**.

Doubling append: just before, table is full: n = cap, so Φ = 2n − n = n. The operation copies n and writes 1: c = n + 1. After: n+1 elements, cap = 2n → Φ = 2(n+1) − 2n = 2. So ΔΦ = 2 − n, and **ĉ = (n+1) + (2−n) = 3**. The n's cancel — the same signature cancellation as the counter's carry run.

VERIFYING PROOF 3 ON THE CONCRETE 9-APPEND TRACE ABOVE

Applying Φ = 2n − cap to every single append from the table above — before *and* after each one — so there's no ambiguity about which n and cap go where. n before append i is always one less than n after (each append adds exactly one element); cap before append i is whatever cap after was left at by the previous append.

APPEND I	N BEFORE→AFTER	CAP BEFORE→AFTER	Φ BEFORE→AFTER	ΔΦ	C	Ĉ = C+ΔΦ
1	0→1	0→1	0→1	+1	1	2 (bootstrap)
2	1→2	1→2	1→2	+1	2	3
3	2→3	2→4	2→2	0	3	3
4	3→4	4→4	2→4	+2	1	3
5	4→5	4→8	4→2	−2	5	3
6	5→6	8→8	2→4	+2	1	3
7	6→7	8→8	4→6	+2	1	3
8	7→8	8→8	6→8	+2	1	3
9	8→9	8→16	8→2	−6	9	3

Append 1 is a harmless one-time exception — bootstrapping cap from 0 (nothing allocated yet) doesn't cleanly fit the doubling formula (doubling 0 gives 0, not 1), so it comes out at 2 instead of 3. From append 2 onward, every single one — cheap writes and the big resize spikes alike — reads exactly 3, whether the real cost is 1 or 9.

Three languages, one answer: **every append is amortized cost 3 = O(1)**. And notice the battery reading: Φ = 2n − cap is literally "how close to full are we" — trouble brewing toward the next resize.

How to invent Φ yourself: a four-step recipe

Every Φ in this lecture looks like magic until you see the pattern. Here is the pattern — use it on any new structure, in homework or exams:

Step 1 — Find the spike and its trigger

Which operation is occasionally expensive, and **what state of the structure makes it fire?** A full array fires a resize. A run of trailing 1s fires a carry cascade. A tall stack enables a big MULTIPOP.

Step 2 — Name the trouble

Define a measurable quantity that **cheap operations build up** and **the spike consumes** — chosen so the spike's actual cost ≈ the amount of trouble it destroys. This is the creative step; everything else is arithmetic.

Step 3 — Set Φ = k × trouble, then tune k

Compute $\hat{c}$ for the spike; pick the constant k so the discharge cancels the spike's cost (the n's or t_i's must cancel). Then check the ground rules: Φ = 0 at the start, Φ ≥ 0 always.

Step 4 — Verify every operation type

Make the little table: $\hat{c} = c + \Delta\Phi$ for each operation. All entries should hit your target bound. **If a cheap operation comes out expensive, your trouble measure is wrong — go back to Step 2.**

STRUCTURE	SPIKE & TRIGGER	TROUBLE MEASURE	Φ	WHY THAT CONSTANT
Multipop stack	MULTIPOP — fires when the stack is tall	pending pops = stacked elements	`|S|`	k = 1: each stored element will cost exactly 1 pop
Binary counter	carry cascade — fires on trailing 1s	1-bits waiting to be reset	`b = #1s`	k = 1: each 1-bit will cost exactly 1 reset
Dynamic array	resize — fires when the table is full	elements added since the last resize = n − cap/2	`2n − cap`	k = 2: each new element must fund copying *itself* + *one old element*

DYNAMIC ARRAY'S TROUBLE MEASURE, VERIFIED ON THE CONCRETE TRACE

Every resize doubles capacity from C to 2C, and it only fires when the table is completely full (n = C). So immediately after any resize, the new cap is exactly twice whatever n was at that instant — meaning **cap/2 always equals the element count the table had right around its most recent doubling**, with no separate variable needed to remember when that was. "n − cap/2" is just n measured against that self-updating baseline.

MOMENT (FROM THE 9-APPEND TRACE ABOVE)	N	CAP	N − CAP/2
Just resized (after append 3)	3	4	1
One cheap append later (append 4)	4	4	2
Just resized again (after append 5)	5	8	1
Table completely full (after append 8)	8	8	4
Just resized again (after append 9)	9	16	1

The value climbs by exactly 1 with every cheap append, then snaps back down the moment the next resize fires — because cap itself jumps up, dragging cap/2 up to meet n. Right when the table is completely full and about to trigger the next resize, n − cap/2 peaks at exactly **cap/2** — but that resize is about to copy *all* n = cap elements, twice as many as the trouble measure counted. That gap is exactly why k = 2: the trouble measure only tracks the newly-added half of the table, so each tracked element must fund copying itself *and* one untracked "old" element sitting alongside it.

Exam strategy: when a question *gives* you Φ, your job is only Steps 3–4 — verify $\Phi \geq 0$, then compute $\hat{c} = c + \Delta\Phi$ per operation type, showing the cancellation. When a question asks you to *design* the analysis, write the recipe explicitly: name the spike, name the trouble, state Φ, verify. That structure earns method marks even if the constant needs adjusting.

Same truth, three vocabularies

	AGGREGATE	ACCOUNTING	POTENTIAL
Core move	Bound total T(n), divide by n	Invent per-operation prices; bank surplus as credit on objects	Define Φ(state); $\hat{c} = c + \Delta\Phi$
Amortized costs	Same for every operation	Can differ per operation type	Can differ per operation type
Correctness obligation	The counting argument itself	Credit never negative (Law 2)	$\Phi(D_i) \geq \Phi(D_0) = 0$
Feels like	Arithmetic	Bookkeeping	Physics
Reach	Simple structures	Medium — needs a good "where to park credit" story	Most powerful — Splay trees, Fibonacci heaps need this one

They are increasing generality, not competing truths: every accounting scheme corresponds to some Φ, and any Φ can be read as a crediting scheme.

"AGGREGATE WORKED FINE FOR EVERYTHING TODAY — WHY LEARN THE OTHER TWO?"

Fair objection — on today's toy structures the three methods are interchangeable, which is exactly why we practise all three here. On the advanced structures ahead, the aggregate method doesn't become wrong; it becomes **unusable**, in two ways:

FAILURE MODE	WHERE YOU'LL MEET IT
The global sum becomes intractable. Aggregate needs one counting argument bounding the total cost of any sequence ("pops ≤ pushes"). For a splay tree, the shape after operation i depends on the *entire history* — no such one-liner exists. The known proof is a potential-function argument.	Lecture 10 Splay Trees
One averaged number becomes uninformative. Aggregate outputs a single blended cost that depends on the operation mix. But a Fibonacci heap's whole point is that *different operations have different amortized costs*: insert and decrease-key O(1), extract-min O(log n). Only accounting/potential can price operations individually — valid for every mix.	Lecture 19 Fibonacci Heaps Lecture 31 Union-Find

So if accounting and potential feel like extra ceremony today, park the doubt — you are learning the mechanics on easy examples **because** the structures that need them are too hard to learn the mechanics on. By Lecture 10 the ceremony will be the only thing standing.

Three takeaways, three future appointments

- Amortized ≠ average-case: it is a **worst-case guarantee over a sequence**, with no probability.
- Expensive operations must be **set up** by cheap ones — pops by pushes, carry cascades by set bits, big copies by many small appends. All three methods just formalize who pays for the setup.
- Aggregate = divide the total; Accounting = tape coins to objects; Potential = one energy function Φ. Master Φ — it's the one the hard structures need.

LECTURE 10

Splay Trees

O(log n) amortized via a beautiful (and tricky) potential function.

LECTURE 19

Fibonacci Heaps

decrease-key in O(1) amortized — the potential method's signature result.

LECTURE 31

Union-Find

Path compression: near-O(1) amortized — resolving today's opening puzzle.

Questions?

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

> *Next: Lecture 4 — External Sorting and the Memory Hierarchy.*