Amortized Analysis

CS5800 — Algorithms

Contents

1 Introduction: aggregate and potential methods

1.1 The problem

- Standard worst-case analysis: cost of an operation × number of operations. This is often far too pessimistic when the expensive operations are rare and forced to be rare *by the cheap ones that must precede them.*
- Amortized analysis studies a *sequence* of n operations $\langle op_1, \ldots, op_n \rangle$ on a data structure, and bounds the **total** cost of the sequence — not the cost of any single operation.
- Goal: show $\sum_{i=1}^{n} c_i = O(f(n))$, where c_i is the true cost of op_i, even though $\max_i c_i$ may be much larger than $f(n)/n$.
- Three standard techniques: **aggregate method**, **accounting (banker's) method**, **potential (physicist's) method**. We focus on aggregate and potential; accounting is a bookkeeping variant of potential and we use it only in passing.

1.2 Running example: binary counter

- A k-bit binary counter starts at 0. INCREMENT adds 1 mod 2^k.
- Cost of one INCREMENT = number of bits flipped.
- Naive bound: a single increment can flip up to $k = O(\log n)$ bits (e.g. $0111 \to 1000$), so n increments cost $O(n \log n)$ in the worst case.
- This is not tight. A costly increment (many trailing 1's) can only happen after many cheap increments rebuilt those trailing 1's one bit at a time.

1.2.1 Aggregate method

- Idea: instead of bounding the cost of the worst iteration and multiplying by n, directly bound the *total* cost of all n operations by charging cost to a fixed resource (here: bit flips), then counting how often each unit of that resource is used across the whole sequence.
- Charge every bit *flip* to the bit that flips. Ask: over n increments, how many times does bit j flip?
  - bit 0 flips on every increment: n times.
  - bit 1 flips every other increment: $\leq n/2$ times.
  - bit 2 flips every 4th increment: $\leq n/4$ times.
  - in general bit j flips $\leq n/2^j$ times.
- Total cost $\leq \sum_{j \geq 0} n/2^j = 2n = O(n)$.
- Amortized cost per operation := (total cost)$/n = O(1)$.

- **Caveat:** this argument is tailored to the specific combinatorial structure of the counter (geometric decay in flip frequency). It does not generalize to more complex data structures — for those we need a tool that works operation-by-operation. That tool is the potential method.

1.2.2 Potential method

Definition. A potential function Φ maps each state D_i of the data structure (after operation i) to a real number, with $\Phi(D_0) = 0$ (or, more generally, $\Phi(D_0)$ known and $\Phi(D_i) \geq \Phi(D_0)$ for all i; the common convention is $\Phi \geq 0$ always and $\Phi(D_0) = 0$).

Define the **amortized cost** of operation i as

$$\hat{c}_i \;=\; c_i + \Phi(D_i) - \Phi(D_{i-1}),$$

where c_i is the true cost. Summing telescopes:

$$\sum_{i=1}^{n} \hat{c}_i \;=\; \sum_{i=1}^{n} c_i \;+\; \Phi(D_n) - \Phi(D_0) \;\geq\; \sum_{i=1}^{n} c_i,$$

using $\Phi(D_n) \geq \Phi(D_0) = 0$. So: *if we can find Φ such that $\hat{c}_i \leq c$ for every i (a fixed constant, or $O(f(n))$), then the true total cost is at most $\sum \hat{c}_i \leq nc$.* The potential Φ is bookkeeping: it represents "stored work" that expensive operations can draw down and cheap operations must build up.

Binary counter, potential method.

- Define $\Phi(D_i)$ = number of 1-bits in the counter after operation i. Note $\Phi \geq 0$ always, $\Phi(D_0) = 0$.
- Only operation is INCREMENT. Suppose before the i-th increment there are k trailing 1's (so the counter looks like $\ldots 0\underbrace{1\cdots 1}_{k}$).
- True cost: $c_i = k + 1$ (the k trailing 1's flip to 0, the first 0 flips to 1).
- Potential change: $\Phi(D_i) - \Phi(D_{i-1}) = (1\text{'s gained}) - (1\text{'s lost}) = 1 - k$.
- Amortized cost: $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = (k+1) + (1-k) = 2$.
- So $\hat{c}_i = 2$ for every increment, regardless of k. Total true cost over n increments is $\leq \sum \hat{c}_i = 2n = O(n)$. This recovers the aggregate bound, but the potential-method proof is *local* — a per-operation argument that never had to think about the whole sequence at once. That locality is exactly what generalizes to complex structures (heaps, trees, self-adjusting lists) where a global aggregate count is not available.

Full worked table, $i = 0$ **to** 16, tying both methods together: c_i is the true cost (bits flipped), $\Phi(D_i)$ is the potential (number of 1-bits), and $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$ is always exactly 2, so the two running totals (true vs. amortized) stay close together the whole way, with the amortized total always an upper bound on the true total.

i	binary	true cost c_i (bits flipped)	true cost running total	$\Phi(D_i)$ (# of 1's)	$\Phi(D_i) - \Phi(D_{i-1})$	amortized cost $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$	amortized running total
0	00000	—	0	0	—	—	0
1	00001	1	1	1	1	2	2
2	00010	2	3	1	0	2	4
3	00011	1	4	2	1	2	6
4	00100	3	7	1	−1	2	8
5	00101	1	8	2	1	2	10
6	00110	2	10	2	0	2	12
7	00111	1	11	3	1	2	14
8	01000	4	15	1	−2	2	16
9	01001	1	16	2	1	2	18
10	01010	2	18	2	0	2	20
11	01011	1	19	3	1	2	22
12	01100	3	22	2	−1	2	24
13	01101	1	23	3	1	2	26
14	01110	2	25	3	0	2	28
15	01111	1	26	4	1	2	30
16	10000	5	31	1	−3	2	32

2 Examples

2.1 Stack with multi-pop

- Operations: PUSH(x), cost 1; MULTI-POP(k), pops $\min(k, |S|)$ elements, cost = number of elements actually popped.
- Naive bound: a multi-pop can cost $O(n)$, and with n operations total this looks like $O(n^2)$.
- Key fact: an element can only be popped once. It must first have been pushed. So across the whole sequence, total pops $\leq$ total pushes $\leq n$.

Potential method. Let $\Phi(D_i) = |S_i|$ (stack size after operation i); $\Phi \geq 0$, $\Phi(D_0) = 0$.

- PUSH: $c_i = 1$, $\Delta\Phi = +1 \Rightarrow \hat{c}_i = 2$.
- MULTI-POP(k), popping $k' = \min(k, |S|)$ elements: $c_i = k'$, $\Delta\Phi = -k' \Rightarrow \hat{c}_i = k' - k' = 0$.

Every operation has $\hat{c}_i \leq 2$, so n operations cost $O(n)$ total — a single line of reasoning that immediately handles multi-pops of any size, unlike an aggregate argument which would have to separately track "how many things are currently on the stack to be popped later."

2.2 Dynamic table (array doubling)

- Table of capacity m, currently storing $n \leq m$ items. INSERT(x): if $n < m$, cost 1. If $n = m$ (full), allocate a new table of capacity $2m$, copy all n items over (cost $n + 1$), then insert.
- Naive bound: a resizing insert costs $O(n)$, so n inserts could look like $O(n^2)$.
- Aggregate method: resizes happen when $n = 1, 2, 4, 8, \ldots, 2^j, \ldots$. Total copying cost $\leq \sum_j 2^j \leq 2n$ (geometric series dominated by the last resize). Adding the n unit-cost inserts, total cost $\leq 3n = O(n)$, amortized $O(1)$ per insert.
- Potential method: define $\Phi(D_i) = 2n_i - m_i$ (twice the current occupancy minus current capacity), where n_i, m_i are size/capacity after operation i. Note right after a resize (or at the start) $n_i = m_i/2$, so $\Phi = 0$; just before a resize $n_i = m_i$, so $\Phi = n_i$ — exactly enough banked potential to pay for the n_i-item copy.
  - Non-resizing insert: $c_i = 1$; n increases by 1, m unchanged, so $\Delta\Phi = 2$. $\hat{c}_i = 1 + 2 = 3$.
  - Resizing insert ($n_{i-1} = m_{i-1} = m$): $c_i = m + 1$ (copy m old items, insert new one); after, $n_i = m + 1$, $m_i = 2m$, so $\Phi(D_i) = 2(m + 1) - 2m = 2$, and $\Phi(D_{i-1}) = 2m - m = m$.

$\Delta\Phi = 2 - m$. $\hat{c}_i = (m+1) + (2-m) = 3$.

- Either way $\hat{c}_i = 3 = O(1)$, confirming $O(n)$ total for n inserts. (Supporting deletion with table halving at $n = m/4$ — not $n = m/2$, to avoid thrashing — gives the same $O(1)$ amortized bound with a similarly-shaped Φ; left as an exercise.)

2.3 BST in-order traversal via repeated Successor ("Succ-sort")

- Given a BST on n keys, produce them sorted by: find the minimum, then call SUCCESSOR $n-1$ times.
- SUCCESSOR(x): if x has a right child, descend one right-edge then all-left edges to the leftmost node of the right subtree (cost = edges traversed downward). If x has no right child, walk up via parent pointers until taking a "left-child" step (or reaching the root) — cost = edges traversed upward.

Aggregate method (edge charging).

- Charge the cost of each SUCCESSOR call to the tree edges it traverses.
- Claim: over the entire sequence of $n-1$ calls, every tree edge is traversed *at most twice* in total — once downward, once upward.
- Why: the sequence of calls exactly reproduces the edge-touches of a standard recursive in-order traversal — each edge (parent→child) is entered once (descending into that subtree) and exited once (climbing back after finishing it). No edge is touched a third time.
- Total work $\leq 2(n-1) = O(n)$. Amortized cost per call $= O(n)/(n-1) = O(1)$.

Potential method (matching the aggregate bound). Define

$$\Phi(D_i) \;=\; \sum_{\text{edges } e} \big(2 - \#\text{times } e \text{ traversed by call } \leq i\big),$$

i.e. each edge starts with a budget of 2 and Φ tracks total remaining budget. $\Phi_0 = 2(n-1) \geq 0$ and Φ stays ≥ 0 (no edge traversed more than twice, by the claim above).

- Each traversed edge in call i decrements Φ by exactly 1, so $\Delta\Phi_i = -c_i$ (true cost = edges traversed in call i).
- $\hat{c}_i = c_i + \Delta\Phi_i = c_i - c_i = 0$ for every call.
- Telescoping: $\sum \hat{c}_i = \sum c_i + (\Phi_n - \Phi_0) = 0 \Rightarrow \sum c_i = \Phi_0 - \Phi_n \leq \Phi_0 = 2(n-1) = O(n)$.

Remark. This Φ is the aggregate argument re-encoded as a potential — it is literally "remaining edge budget," so it absorbs the true cost exactly and every amortized cost collapses to the trivial value 0. That is a valid proof but a degenerate one: it confirms the $O(n)$ total without producing a meaningful uniform per-call charge. A genuinely informative potential is $\Phi(D_i) = \text{depth(current node)}$: downward calls have amortized cost $2(1+k)$ (not bounded by a constant!) and upward calls have amortized cost 0. The total still telescopes to $O(n)$ (since depth is bounded and returns to comparable values), but individual calls are *not* uniformly $O(1)$ — one call can legitimately cost $\Theta(\text{depth})$. This is the signature case for preferring the aggregate method outright: some data structures have a clean global invariant (edges touched $\leq 2n$) with no clean uniform per-operation potential.

3 Heavy examples

3.1 Red-Black tree insertion

- Recall RB-INSERT: insert as in an ordinary BST, color the new node red, then fix up color violations by walking toward the root, at each step either **recoloring** (parent, uncle, grandparent) or performing $O(1)$ **rotations** and stopping.

- Worst case: the fixup loop can recolor $\Theta(\log n)$ nodes on a single insert (chain of alternating red/black up a path of height $\Theta(\log n)$). We want to show this is rare on average.

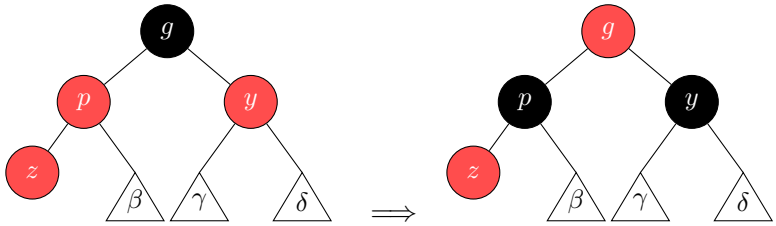
Potential function. Define, for a red-black tree T,

$$\Phi(T) \;=\; \#\{\text{red nodes in } T\}.$$

$\Phi \geq 0$ always; $\Phi = 0$ for the empty tree.

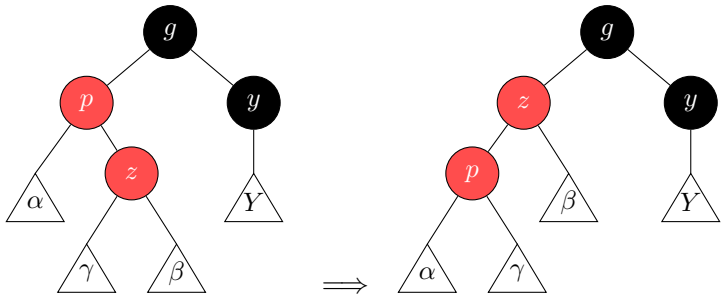
Case analysis of one fixup iteration. Following CLRS (and your slides) exactly, there are **three** cases for the current violating node z (red, with red parent p); g is z's grandparent (black) and y is z's uncle (g's other child).

Case 1 (uncle y red).



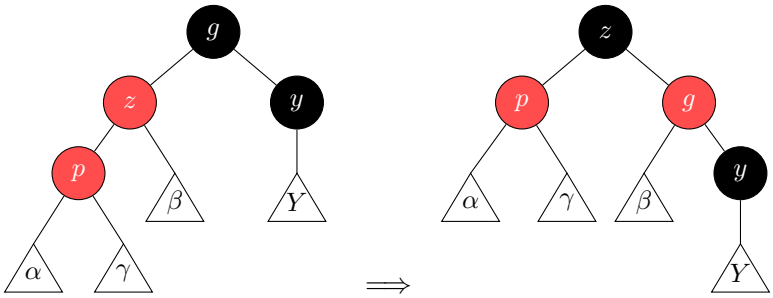
- Fix: recolor p, y black and g red; move z up to g and repeat (the loop continues one level higher).
- Cost $c = O(1)$. $\Delta\Phi$: p, y turn red→black (-2), g turns black→red ($+1$); net $\Delta\Phi = -1$. Amortized $\hat{c} = O(1) - 1 = O(1)$.

Case 2 (uncle y black, z a "triangle": z is the right child of p, p is the left child of g).



- Fix: left-rotate at p; no recoloring. This turns the "triangle" into a "line" — exactly the shape Case 3 expects, with the lower-left red node now playing the role of z (CLRS reassigns $z \leftarrow$ old p).
- Cost $c = O(1)$ (one rotation). $\Delta\Phi = 0$ (no colors changed). Amortized $\hat{c} = O(1)$. **Falls through immediately into Case 3** — CLRS's pseudocode has no separate loop iteration for Case 2 alone.

Case 3 (uncle y black, z a "line": z is the left child of p, p is the left child of g — either directly, or as produced by Case 2's rotation).



- Fix: right-rotate at g; recolor z black, g red (p stays red).

- Cost $c = O(1)$ (one rotation, two recolors). $\Delta\Phi$: one node (z) turns red→black (-1), one node (g) turns black→red ($+1$); net $\Delta\Phi = 0$. Amortized $\hat{c} = O(1)$. **Loop terminates** — no red-red violation remains, since z (the new subtree root) is now black.

Mapping back. Case 1 is exactly what the previous pass of these notes called *Case A*: the only case that iterates, each instance strictly decreasing Φ by 1. Cases 2+3 together are exactly what these notes called *Case B/C*: Case 2 is a pure, potential-neutral rotation that always falls straight through into Case 3, and Case 3 does the recoloring that ends the loop with $\Delta\Phi = O(1)$. Splitting 2 and 3 apart (as CLRS does, since they are literally different rotation directions — left vs. right, and mirror-symmetric left/right versions of each) doesn't change the amortized bookkeeping at all; it only matters for describing *which* rotation to physically perform.

Because Case 1 is the only case that iterates, and each Case-1 iteration strictly decreases Φ by 1 while $\Phi \geq 0$ always, the telescoping sum over one INSERT call's entire fixup loop is $O(1)$ amortized, even though the *true* cost of a single call can be $\Theta(\log n)$ when many Case-1 iterations chain together.

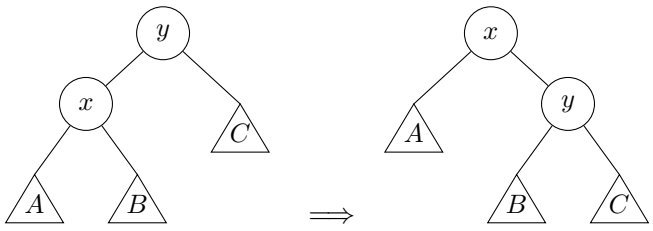
- Concretely: inserting the new red leaf itself costs $O(1)$ true cost and $\Delta\Phi = +1$ (one new red node), amortized $O(1)$. Then the fixup loop runs some number of Case-1 iterations (each amortized $O(1)$, paid for by the potential it destroys) followed by at most one Case-2→3 step.
- Total amortized cost of RB-INSERT is $O(1)$, hence n insertions cost $O(n)$ total — matching the familiar $O(\log n)$ *worst-case* per operation, but showing the *average* behavior is much better, and giving a clean reason why: every recoloring pays for itself out of red nodes it destroys, and rotations happen only $O(1)$ times per call.

3.2 Splay trees

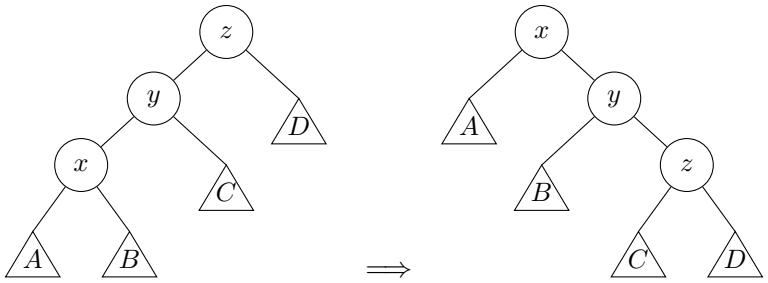
What is a splay tree?

- A splay tree is a binary search tree with **no** explicit balance information (no color bits, no rank/height fields) — nothing stored beyond the usual key and child pointers.
- Instead, it stays efficient via a single rule applied after *every* access, insert, or delete of a node x: **splay x to the root**, i.e., repeatedly rotate x upward until it becomes the root of the whole tree.
- Consequence: whatever was just touched is now sitting at the root — cheap to reach again. Recently-accessed items stay near the top; this is the same intuition as move-to-front (§3.4), applied to a tree instead of a list.
- Splaying is done via three kinds of local rotation steps, applied to x together with its parent y (and grandparent z, when one exists), repeated until x reaches the root:

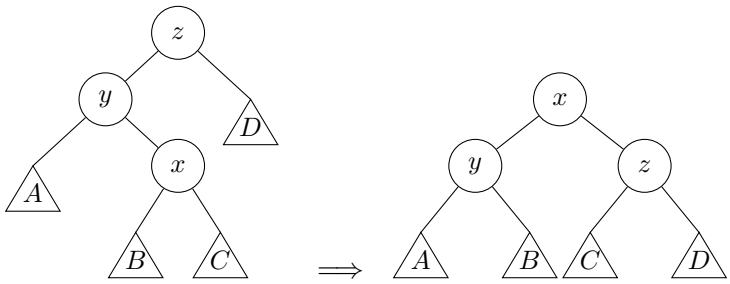
Zig — y is the root (no grandparent). Single rotation of x over y. Happens at most once per splay, always as the very last step.



Zig-zig — x, y are both left children (or both right children) of their respective parents. Rotate y over z *first*, then x over y.



Zig-zag — one of x, y is a left child, the other a right child. Rotate x over y, then x over z (equivalently: x jumps straight up two levels, y and z become its two children).



- All three preserve the BST ordering (check the in-order sequence of subtrees A, B, C, D — it's identical before and after each transformation).
- Find(x, S), Insert(x, S), Delete(x, S) are all implemented by first doing the ordinary BST operation and then splaying the relevant node — see your slides / CLRS-adjacent references for the small bookkeeping this takes for insert and delete.
- Worst case for a single splay: $\Theta(n)$ (e.g. splaying the deepest node of a long path). We now show the *amortized* cost of a splay is only $O(\log n)$ — this is the payoff for the "no balance info at all" simplicity.

Amortized analysis

Potential function. For each node x, let $s(x)$ = number of nodes in the subtree rooted at x (including x), and define the **rank** $\mathrm{rk}(x) = \lfloor \log_2 s(x) \rfloor$. Define

$$\Phi(T) = \sum_{x \in T} \mathrm{rk}(x).$$

Lemma 1 (Access Lemma)**.** *The amortized cost of splaying node x to the root of a tree T (counting one unit of true cost per rotation) is at most*

$$3\big(\mathrm{rk}(T) - \mathrm{rk}(x)\big) + 1,$$

where $\mathrm{rk}(T)$ *denotes the rank of the root of* T*.*

Proof sketch, per splay step (with $\mathrm{rk}, \mathrm{rk}'$ *denoting rank before/after the step):*

- **Zig** (single rotation, happens ≤ 1 time, at the top): true cost 1. One shows the amortized cost is $\le 3(\mathrm{rk}'(x) - \mathrm{rk}(x)) + 1$: the $+1$ absorbs this one low-level rotation, using $\mathrm{rk}'(x) = \mathrm{rk}(T)$ (after a zig at the root, x has the tree's old root rank).
- **Zig-zig**: true cost 2 (two rotations). The key inequality is $\mathrm{rk}'(x)+\mathrm{rk}'(y)+\mathrm{rk}'(z) \le \mathrm{rk}(x)+\mathrm{rk}(y)+\mathrm{rk}(z)$ is *not* always true, but a careful case split (using $\mathrm{rk}(x) = \mathrm{rk}'(z)$ and concavity of log, i.e., if $a+b \le c$ with $\lfloor \log a \rfloor = \lfloor \log b \rfloor$ then $\lfloor \log(a+b+1) \rfloor > \lfloor \log a \rfloor$ unless $a = b$) shows the amortized cost is $\le 3(\mathrm{rk}'(x) - \mathrm{rk}(x))$, with equality possible only in a degenerate case that itself provides one unit of slack to pay for the two rotations.

- **Zig-zag**: true cost 2; a symmetric argument gives amortized cost $\leq 2(\mathrm{rk}'(x) - \mathrm{rk}(x)) \leq 3(\mathrm{rk}'(x) - \mathrm{rk}(x))$.
- Summing a full splay's steps telescopes (each step's bound is $3(\mathrm{rk}'(x) - \mathrm{rk}(x))$ except the last which adds $+1$): total amortized cost $\leq 3(\mathrm{rk}(T) - \mathrm{rk}(x)) + 1$, where $\mathrm{rk}(x)$ here means x's rank *before* the splay begins.

Consequence. Since $\mathrm{rk}(T) \leq \lfloor \log_2 n \rfloor$ always, every splay costs $O(\log n)$ amortized, so any sequence of m operations (find/insert/delete, each $O(1)$ splays) on a tree that never exceeds n nodes costs $O(m \log n)$ total — matching a balanced BST, without ever maintaining balance information. (The deeper theorem, the *static optimality / dynamic optimality conjecture* territory, is that splay trees are additionally competitive with the *optimal static* BST for the actual access sequence — stated here as folklore, proof omitted.)

3.3 Scapegoat trees

What is a scapegoat tree?

- Like a splay tree, a scapegoat tree stores **no** per-node balance information at all — not even a rank/height field. The only extra bookkeeping is two global integers: n (current number of nodes) and q (the largest n has been since the tree was last completely rebuilt from scratch).
- Fix a constant $\alpha \in (\frac{1}{2}, 1)$ (a common choice is $\alpha = 2/3$). Call a node v **α-weight-balanced** if

$$\mathrm{size}(\mathrm{left}(v)) \leq \alpha \cdot \mathrm{size}(v) \quad \text{and} \quad \mathrm{size}(\mathrm{right}(v)) \leq \alpha \cdot \mathrm{size}(v),$$

  where $\mathrm{size}(v)$ counts v and all its descendants. Intuitively: neither child holds more than an α fraction of v's subtree.
- Invariant maintained after every operation: the **height** of the whole tree never exceeds $\log_{1/\alpha}(n)$. (Not every node needs to be α-weight-balanced forever — only the global height bound is the actual guarantee.)

Insertion

- Insert the new key as an ordinary BST leaf. Update $n \mathrel{+}= 1$; update $q \leftarrow \max(q, n)$.
- If the new leaf's depth exceeds $\log_{1/\alpha}(n)$: walk back up from the leaf toward the root, and find the **first** (deepest) ancestor z that is *not* α-weight-balanced — call z the **scapegoat**.
- Rebuild the entire subtree rooted at z into a perfectly balanced BST (read its keys via an in-order traversal — they come out sorted — then recursively build a balanced tree from the sorted array, root = median). This costs $O(\mathrm{size}(z))$.

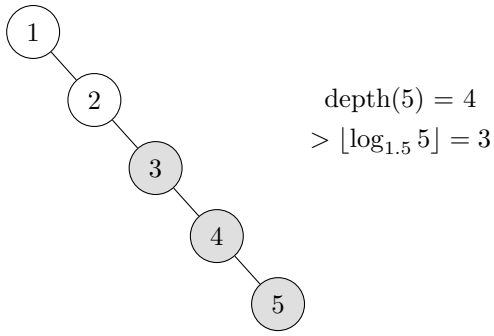
Lemma 2 (Scapegoat existence)**.** *If a leaf's depth after insertion exceeds $\log_{1/\alpha}(n)$, some proper ancestor of that leaf is not α-weight-balanced.*

Proof. Suppose, toward a contradiction, that every ancestor on the path from the leaf (depth d, size 1) up to the root *is* α-weight-balanced. Weight-balance at a node on the path says the child containing the path has size $\leq \alpha \cdot$ (parent's size), i.e. the parent's size is at least $1/\alpha$ times the child's size. Applying this once per level from the leaf to the root,

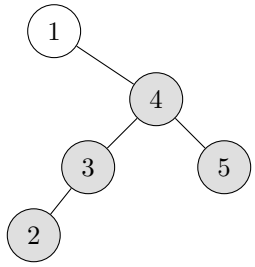
$$n = \mathrm{size}(\mathrm{root}) \;\geq\; (1/\alpha)^d \cdot \mathrm{size}(\mathrm{leaf}) \;=\; (1/\alpha)^d.$$

Taking logs, $d \leq \log_{1/\alpha} n$ — contradicting $d > \log_{1/\alpha} n$. So some ancestor must fail to be α-weight-balanced. □

Example ($\alpha = 2/3$). Insert $1, 2, 3, 4$ in increasing order — with no rebalancing this is the classic BST worst case, a right-leaning chain — then insert 5.



Walking up from the new leaf 5: node 4's subtree $\{4, 5\}$ is balanced; node 3's subtree $\{3, 4, 5\}$ is still balanced ($2 \leq \frac{2}{3} \cdot 3$); node 2's subtree $\{2, 3, 4, 5\}$ has right-side size $3 > \frac{2}{3} \cdot 4$ — **node 2 is the scapegoat**. Rebuild its subtree $\{2, 3, 4, 5\}$ into a perfectly balanced BST:



Height of the whole tree drops from 4 to 3, restoring the invariant; rebuild cost was $O(4)$, proportional to the scapegoat's subtree size — not the whole tree.

Amortized analysis

Lemma 3 (Resistance to re-imbalance)**.** *Once a subtree of size m is rebuilt perfectly balanced, at least $c \cdot m$ further insertions into that subtree must occur (for a constant $c = c(\alpha) > 0$) before its root can again become a scapegoat.*

Proof. Right after a balanced rebuild, the two child subtrees have sizes $\lfloor (m-1)/2 \rfloor, \lceil (m-1)/2 \rceil$ — call them both $\approx m/2$. Suppose k further insertions land entirely in the larger side (the worst case for triggering imbalance again). Sizes become $\approx m/2$ and $\approx m/2 + k$, total size $\approx m + k$. For the node to become unbalanced we'd need

$$\tfrac{m}{2} + k \;>\; \alpha(m + k) \iff k(1 - \alpha) > m(\alpha - \tfrac{1}{2}) \iff k > \frac{\alpha - \frac{1}{2}}{1 - \alpha} \cdot m.$$

Since $\alpha \in (\frac{1}{2}, 1)$, the constant $c = \frac{\alpha - 1/2}{1-\alpha}$ is a fixed positive number, so $k = \Omega(m)$ is required. □

Putting it together. Charge every insertion 1 credit toward a shared "rebuild fund" for whichever subtree it lands in. By the lemma, a subtree of size m cannot be rebuilt again until it has absorbed $\geq c \cdot m$ insertions since its last rebuild — i.e., $\geq c \cdot m$ credits are on deposit by the time it becomes a scapegoat, enough to pay its $O(m)$ rebuild cost at $O(1/c) = O(1)$ per credit. So:

- Every insertion costs $O(\log n)$ just to walk down to its leaf (the tree's height is always $\leq \log_{1/\alpha} n$ by the maintained invariant), **plus** $O(1)$ amortized for its share of the periodic rebuilds.
- Total amortized cost per INSERT: $O(\log n)$.

Deletion. Delete the key as an ordinary BST delete (no local rebuild needed); update $n \mathrel{-}= 1$. If $n < \alpha \cdot q$ (the tree has shrunk too far below its high-water mark q), rebuild the *entire* tree into a perfectly balanced

BST — cost $O(n)$ — and reset $q \leftarrow n$. This is exactly the same "shrink only when far enough below the last resize point" trick as the dynamic table in §2.2 (there: halve only at $n = m/4$, not $m/2$, to avoid thrashing): between two full rebuilds, q stays fixed and n must fall by $(1-\alpha)q = \Omega(q)$, i.e. $\Omega(q)$ deletions must occur to trigger the next $O(q)$-cost rebuild, giving $O(1)$ amortized per deletion for this part — plus the usual $O(\log q)$ to walk down and physically delete. Total amortized cost per DELETE: $O(\log n)$.

Takeaway. Scapegoat trees are the "aggregate/accounting method" sibling of splay trees' "potential method" story: same selling point (zero balance metadata, $O(\log n)$ amortized operations), but the proof mechanism is a direct credit-charging argument tied to a combinatorial resistance-to-reimbalance fact, rather than a numerically-tracked potential function. They're also the clearest illustration in this course of "rare, expensive, wholesale rebuild" amortization — closer in spirit to the dynamic table (§2.2) than to any of the incremental-repair structures (Fibonacci heaps, red-black trees) seen so far.

3.4 Move-to-front (MTF) list heuristic

- Self-organizing linked list L of n items. ACCESS(x) costs $\mathrm{rk}_L(x)$ (position of x from the front, 1-indexed) to find it, then MTF moves x to the front using adjacent-item swaps, at cost $2 \cdot \mathrm{rk}_L(x)$ total (find + move) in the standard accounting.
- Goal: compare MTF (an *online* algorithm — doesn't know future accesses) against OPT, the best *offline* algorithm for the same access sequence S (which may know the whole sequence in advance and can also reorder via adjacent swaps at cost 1 each).

Theorem 1. *$C_{\mathrm{MTF}}(S) \le 4\, C_{\mathrm{OPT}}(S)$ for every access sequence S (MTF is 4-competitive).*

Potential function. Let L_i, L_i^* be the MTF and OPT lists after the i-th access. Define

$$\Phi(L_i) = 2 \cdot |\{(x,y) : x \text{ precedes } y \text{ in } L_i \text{ but } y \text{ precedes } x \text{ in } L_i^*\}| = 2 \cdot (\#\text{ inversions between } L_i \text{ and } L_i^*).$$

$\Phi \ge 0$ always, and $\Phi(L_0) = 0$ if both lists start identical.

Proof sketch. Suppose access i requests item x. Partition the other items (as they stood just before this access) by their relative order to x in each list:

$$A = \{y : y \prec x \text{ in both}\}, \quad B = \{y : y \prec x \text{ in } L,\ y \succ x \text{ in } L^*\}, \quad C = \{y : y \succ x \text{ in } L,\ y \prec x \text{ in } L^*\}.$$

Let $r = \mathrm{rk}_L(x) = |A| + |B| + 1$ and $r^* = \mathrm{rk}_{L^*}(x) = |A| + |C| + 1$. Moving x to the front in L destroys the $|B|$ inversions between x and B and creates $|A|$ new inversions between x and A; each adjacent swap OPT performs (there are t_i of them) changes Φ by at most $+2$. So

$$\Phi(L_i) - \Phi(L_{i-1}) \ \le\ 2(|A| - |B| + t_i).$$

The true MTF cost is $c_i = 2r$. Using $r = |A| + |B| + 1$ to eliminate $|B|$, then $r^* = |A| + |C| + 1 \ge |A| + 1$ to bound $|A|$:

$$\hat{c}_i = c_i + \Delta\Phi \ \le\ 2r + 2(|A| - |B| + t_i) \ =\ 4|A| + 2 + 2t_i \ \le\ 4(r^* + t_i) \ =\ 4\, c_i^*,$$

where $c_i^* = r^* + t_i$ is OPT's true cost for the same access (find cost r^* plus t_i reordering swaps). Summing over the sequence and telescoping ($\Phi(L_0) = 0,\ \Phi(L_{|S|}) \ge 0$):

$$C_{\mathrm{MTF}}(S) = \sum_i c_i = \sum_i \left(\hat{c}_i - \Delta\Phi_i\right) \le 4 \sum_i c_i^* + \Phi(L_0) - \Phi(L_{|S|}) \le 4\, C_{\mathrm{OPT}}(S). \qquad \blacksquare$$

Takeaway. The inversion-count potential is exactly the "disagreement" between the online algorithm's list and whatever the offline optimum happens to be doing, and MTF's rule (always move accessed item to front) is precisely aggressive enough to keep that disagreement from growing unboundedly relative to OPT's cost — a very different flavor of potential argument from the structural (size/rank/red-count) potentials of the earlier examples, since it compares *two* evolving data structures rather than bounding one against a fixed baseline.

4 Fibonacci heaps

- History: Fredman–Tarjan 1986, motivated by speeding up Dijkstra's algorithm and Prim's MST via a heap where DECREASE-KEY is cheap.
- Structure: a collection of heap-ordered trees (min-heap property: parent key $\leq$ children keys), a pointer to the overall-minimum root, and a set of **marked** nodes used to keep trees from becoming too unbalanced.
- Idea vs. binomial heaps: binomial heaps *eagerly* consolidate (merge trees of equal degree) after every insert; Fibonacci heaps are *lazy* — they let the root list grow unchecked and only consolidate during EXTRACT-MIN.
- Notation: n = number of nodes, $t(H)$ = number of trees (roots) in H, $m(H)$ = number of marked nodes, $D(n)$ = upper bound on the max degree of any node (one can show $D(n) = O(\log n)$, via a Fibonacci-number argument — not needed for the amortized bookkeeping below).

Potential function.

$$\Phi(H) = t(H) + 2\,m(H).$$

We now compute, for each operation, the true cost c, the change $\Delta\Phi$, and the resulting amortized cost $\hat{c} = c + \Delta\Phi$.

Insert

- Create a new singleton one-node tree, splice it into the root list, update `min` if needed.
- True cost: $c = O(1)$.
- Change in potential: one new tree, no new marks $\Rightarrow \Delta\Phi = +1$.
- Amortized cost: $\hat{c} = O(1) + 1 = O(1)$.

Extract-Min

- Remove the min root; splice its (up to $D(n)$) children into the root list; then **consolidate**: repeatedly link together any two roots of equal degree (make the larger key a child of the smaller) until all roots have distinct degree; finally scan the (now $\leq D(n) + 1$ length) root list to find the new min.
- True cost: $O(D(n))$ to meld in the children, plus $O(t(H))$ to do the consolidating links and rescan the root list (each link reduces the root-list length by 1, so # links $\leq$ old root-list length). Total: $c = O(D(n) + t(H))$.
- Change in potential: after consolidation, at most $D(n) + 1$ roots remain (all distinct degrees, each degree $\leq D(n)$), and no node becomes newly marked during Extract-Min. So $t(H') \leq D(n) + 1$, giving

$$\Delta\Phi \leq \big(D(n) + 1\big) - t(H).$$

- Amortized cost: $\hat{c} = c + \Delta\Phi = O(D(n) + t(H)) + (D(n) + 1 - t(H)) = O(D(n))$.
- Since $D(n) = O(\log n)$: $\hat{c} = O(\log n)$. Crucially, the $-t(H)$ term in $\Delta\Phi$ exactly cancels the $O(t(H))$ true cost of doing all the consolidating links — *that* is why a heap that has been allowed to accumulate many cheap, lazy roots (from prior inserts / decrease-keys) doesn't blow up the cost here: the linking work is paid for by the potential those roots were carrying.

Decrease-Key

- Decrease x's key. If heap-order still holds (new key $\geq$ parent's key), done.
- Otherwise: cut the subtree rooted at x, splice it into the root list, unmark x. If x's parent p was already marked (i.e., p had previously lost one child), **cascade**: cut p too, splice it into the root

list, unmark it, and repeat on p's parent — and so on up the tree, stopping at the first unmarked ancestor, which gets marked (unless it's a root, which is never marked).

- Let c' = number of cuts performed (the initial cut of x, plus the cascading cuts). True cost: $O(1)$ per cut (constant-time splice) $\Rightarrow c = O(c')$.
- Change in potential: each of the c' cut subtrees becomes a new root $\Rightarrow t(H') = t(H) + c'$. Of the $c' - 1$ cascading cuts, each one *unmarks* a node that was marked (net -1 mark each), and the final ancestor in the chain gets newly *marked* (net $+1$), so the mark count changes by at most $-(c' - 1) + 1 = -(c' - 2)$, i.e., $m(H') \le m(H) - (c' - 1) + 1$. So

  $$\Delta\Phi \le c' + 2\big(-(c'-1)+1\big) = c' - 2c' + 4 = -c' + 4 \ \text{(roughly} -2(c'-1) \text{ from unmarking, dominating).}$$

  (The clean textbook statement: $\Delta\Phi \le 4 - c'$, since we gain c' trees at weight 1 each but lose at least $2(c' - 1)$ from unmarking $c' - 1$ nodes, offset by at most $+2$ for marking one new node.)
- Amortized cost: $\hat{c} = c + \Delta\Phi = O(c') + (4 - c') = O(1)$.
- So each Decrease-Key is $O(1)$ amortized *regardless of how many cascading cuts it triggers* — the true cost of the cascade is entirely paid for by the potential drop from unmarking nodes on the way up. This is the whole point of the marking scheme: it converts "a node has been cut once already" into stored potential that finances the eventual second cut.

Union

- Concatenate the two (circular, doubly-linked) root lists; keep the smaller of the two mins.
- True cost: $c = O(1)$. Change in potential: t, m are each just the sums of the two heaps' values, so $\Delta\Phi = 0$.
- Amortized cost: $\hat{c} = O(1)$.

Delete

- Delete(x) = Decrease-Key($x, -\infty$) followed by Extract-Min.
- Amortized cost = (amortized Decrease-Key) + (amortized Extract-Min) = $O(1) + O(\log n) = O(\log n)$.

Summary table (amortized).

	Insert	Find-Min	Extract-Min	Union	Decrease-Key	Delete
Binary heap	$\log n$	1	$\log n$	n	$\log n$	$\log n$
Binomial heap	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$
Fibonacci heap	**1**	**1**	$\log n$	**1**	**1**	$\log n$

The headline win is $O(1)$ amortized Decrease-Key, which is exactly what makes Dijkstra / Prim run in $O(E + V \log V)$ instead of $O(E \log V)$: those algorithms call Decrease-Key up to E times but Extract-Min only V times.

5 Disjoint sets (Union–Find)

- ADT: Make-Set(x), Find-Set(x) (returns a representative of x's set), Union(x, y) (merges the sets containing x, y).
- Tree-based ("disjoint-set forest") representation: each set is a rooted tree; each node points to its parent; the root is the representative. Find-Set(x) walks up parent pointers to the root.
- Two independent heuristics, each individually good, and spectacular together.

5.1 Union by rank

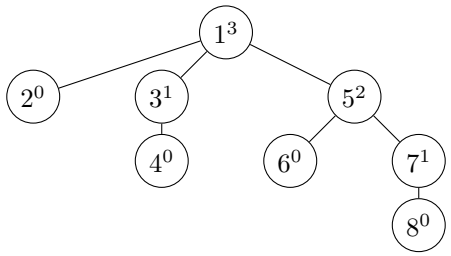
- Maintain a rk[x] upper-bounding the height of the subtree rooted at x. UNION attaches the root of *smaller* rank under the root of *larger* rank (ties broken arbitrarily, incrementing the resulting root's rank by 1).
- Aggregate argument for the height bound: a node's rank only increases when it stops being a root, and only by attaching to a strictly larger (or equal, then +1) tree — so a tree of rank r has size $\geq 2^r$ (easy induction). Hence rank $\leq \log_2 n$ always, so every FIND-SET costs $O(\log n)$ *worst-case* — no amortization even needed yet for this heuristic alone.

5.2 Path compression

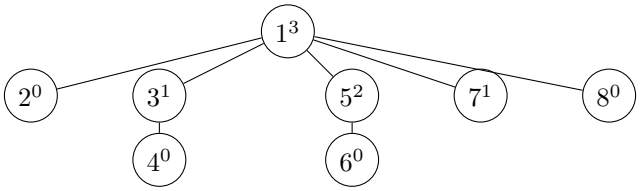
- During FIND-SET(x), after locating the root r, repoint *every* node visited on the path directly to r.
- Alone (without union-by-rank), path compression already gives an amortized $O(\log n)$ per operation (aggregate argument: a long find path can only occur a limited number of times before compression flattens it — omitted here since we get a much stronger bound combining both heuristics).

5.3 Example: building and compressing a forest

Sequence: UNION$(1,2)$, UNION$(3,4)$, UNION$(5,6)$, UNION$(7,8)$ — each pair has equal rank 0; break ties by making the first element's root the new root, rank $\to 1$. Then UNION$(1,3)$, UNION$(5,7)$ — equal rank 1, rank $\to 2$. Then UNION$(1,5)$ — equal rank 2, rank $\to 3$. Ranks shown as superscripts.



FIND-SET(8) walks $8 \to 7 \to 5 \to 1$ (depth 3) to locate the root, then path compression re-points $8, 7, 5$ directly to 1:



Node 6 is untouched (it hangs off 5, but 6 itself was never on the find path), while $8, 7, 5$ all now point directly at the root — a later FIND-SET(6) would cost only 2 edge-traversals instead of the original 3, and would itself compress 6 to point directly at 1 too.

5.4 Combined: union by rank + path compression

- With both heuristics, a sequence of m MAKE-SET/UNION/FIND-SET operations (with n elements, $m \geq n$) costs

$$O\big(m\,\alpha(n)\big),$$

where α is the (extremely slowly growing) inverse Ackermann function — for any n that could ever be written down, $\alpha(n) \leq 4$. In practice this is "amortized constant."

Detailed sketch of the amortized argument (still not a fully formal proof). We now unpack the "block" idea in more depth. This is the real mechanism behind the $O(m\,\alpha(n))$ bound, but making every step airtight (Tarjan 1975; CLRS §21.4) requires more careful bookkeeping than we do here — we flag exactly what's being skipped at the end.

Step 1: a concrete block partition. Define the **tower function** $T(1) = 1$, $T(k) = 2^{T(k-1)}$ for $k \geq 2$:

$$T(1) = 1, \quad T(2) = 2, \quad T(3) = 4, \quad T(4) = 16, \quad T(5) = 65536, \quad T(6) = 2^{65536} \text{ (astronomical)}.$$

For a node v with frozen rank $r = \mathrm{rk}(v)$ (rank is assigned once, when v stops being a root, and never changes again), define $\mathrm{block}(v) = \min\{k : T(k) \geq r\}$. Concretely: block 1 holds rank 1; block 2 holds rank 2; block 3 holds ranks 3–4; block 4 holds ranks 5–16; block 5 holds ranks 17–65536; block 6 already covers every rank up to 2^{65536} — far more than the number of atoms in the observable universe, let alone any realistic n. This is the concrete reason $\alpha(n) \leq 4$ or 5 for every n anyone will ever actually run this on, even though $\alpha(n) \to \infty$ in the limit as $n \to \infty$.

Step 2: charge each edge on a find path to one of two buckets. During FIND-SET(x), look at the path $v_1 = x, v_2, \ldots, v_L = \text{root}$ *before* compression happens, and classify each edge (v_i, v_{i+1}):

- **Bucket (a):** $\mathrm{block}(v_{i+1}) > \mathrm{block}(v_i)$ — parent lies in a strictly higher block.
- **Bucket (b):** $\mathrm{block}(v_{i+1}) = \mathrm{block}(v_i)$ — parent and child share a block. (The parent's block is never *lower*, since rank strictly increases going up any tree.)

Bucket (a) is the easy part. Along one find path, block numbers only increase going up, and there are at most $\mathrm{block}(n) = O(\alpha(n))$ distinct blocks in existence at all. So at most $O(\alpha(n))$ of the edges on *any single* find can be bucket-(a) edges — immediately giving $O(m\,\alpha(n))$ total over m finds.

Bucket (b) needs the compression mechanism itself. Key extra fact (stated here, not fully proved): once path compression repoints a node v to some ancestor p, then at every *future* compression touching v, v's new parent has rank $\geq \mathrm{rank}(p)$ — parent-rank is non-decreasing over v's lifetime, since you're always repointed to an ancestor at least as high on the original find path. So for a node v sitting in block k (its own rank $r \in (T(k-1), T(k)]$), its parent's rank can pass through at most $T(k)$ distinct values while staying inside block k before permanently exceeding $T(k)$ — after which v's edge is a bucket-(a) edge forever. So: *a single node can be the source of a bucket-(b) charge at most $O(T(k))$ times across the algorithm's entire run.*

Combine this with the earlier fact that at most $n/2^r$ nodes ever have rank exactly r (§5.1): total bucket-(b) charges from all rank-r nodes in block k is at most $(n/2^r) \cdot T(k)$. Summing over the ranks in block k (i.e., $r = T(k-1)+1, \ldots, T(k)$):

$$\sum_{r=T(k-1)+1}^{T(k)} \frac{n}{2^r} \cdot T(k) \;=\; n\,T(k) \sum_{r=T(k-1)+1}^{T(k)} 2^{-r} \;\leq\; n\,T(k) \cdot 2^{-T(k-1)} \;=\; n \cdot 2^{T(k-1)} \cdot 2^{-T(k-1)} \;=\; O(n),$$

using $T(k) = 2^{T(k-1)}$ in the last step — the tower function's doubling-in-the-exponent structure is precisely what makes the geometric decay (2^{-r}, summed) exactly cancel the $T(k)$ prefactor, leaving each block contributing only $O(n)$ bucket-(b) charges, *independent of how large that block's ranks are.* With $O(\alpha(n))$ blocks total, bucket (b) contributes $O(n\,\alpha(n))$ overall.

Putting the buckets together.

$$O(m\,\alpha(n)) \;+\; O(n\,\alpha(n)) \;=\; O\big((m+n)\,\alpha(n)\big) \;=\; O(m\,\alpha(n)) \quad (\text{since } m \geq n).$$

What this sketch skips. This is deliberately not a complete proof. In particular we have not: verified the "$\leq T(k)$ distinct parent-rank values" claim carefully against compressions triggered by *other* finds passing through v's ancestors (not just finds through v itself); tracked the $\pm O(1)$ slack in the geometric-series bound precisely; or handled boundary cases (block 1, roots, nodes that are never compressed at

all). Filling in all of this rigorously is exactly what Tarjan's original 1975 analysis does (and CLRS §21.4 packages it slightly differently, using a two-parameter $\alpha(m, n)$ rather than a single-parameter $\alpha(n)$). The headline result and the mechanism above are both correct; we simply stop short of the fully formal version here.

Takeaway. Neither heuristic alone gets below $\Theta(\log n)$ amortized; it's specifically the *interaction* — union-by-rank guarantees a controlled, exponentially-growing-with-rank tree size, and path compression exploits that guarantee by collapsing long paths whenever they're walked — that pushes the bound down to (essentially) constant. This is the deepest amortized-analysis result in the course: the case where the "potential" being paid down isn't a simple counted quantity (red nodes, list size, marks) but a multi-level accounting scheme whose recursive structure literally *is* the inverse Ackermann function.