Amortized Analysis

Outline for Today

- ***The Two-Stack Queue***
  - A simple, fast implementation of a queue.
- ***Amortized Analysis***
  - A little accounting trickery never hurt anyone, right?
- ***Red/Black Trees Revisited***
  - Some subtle and useful properties.

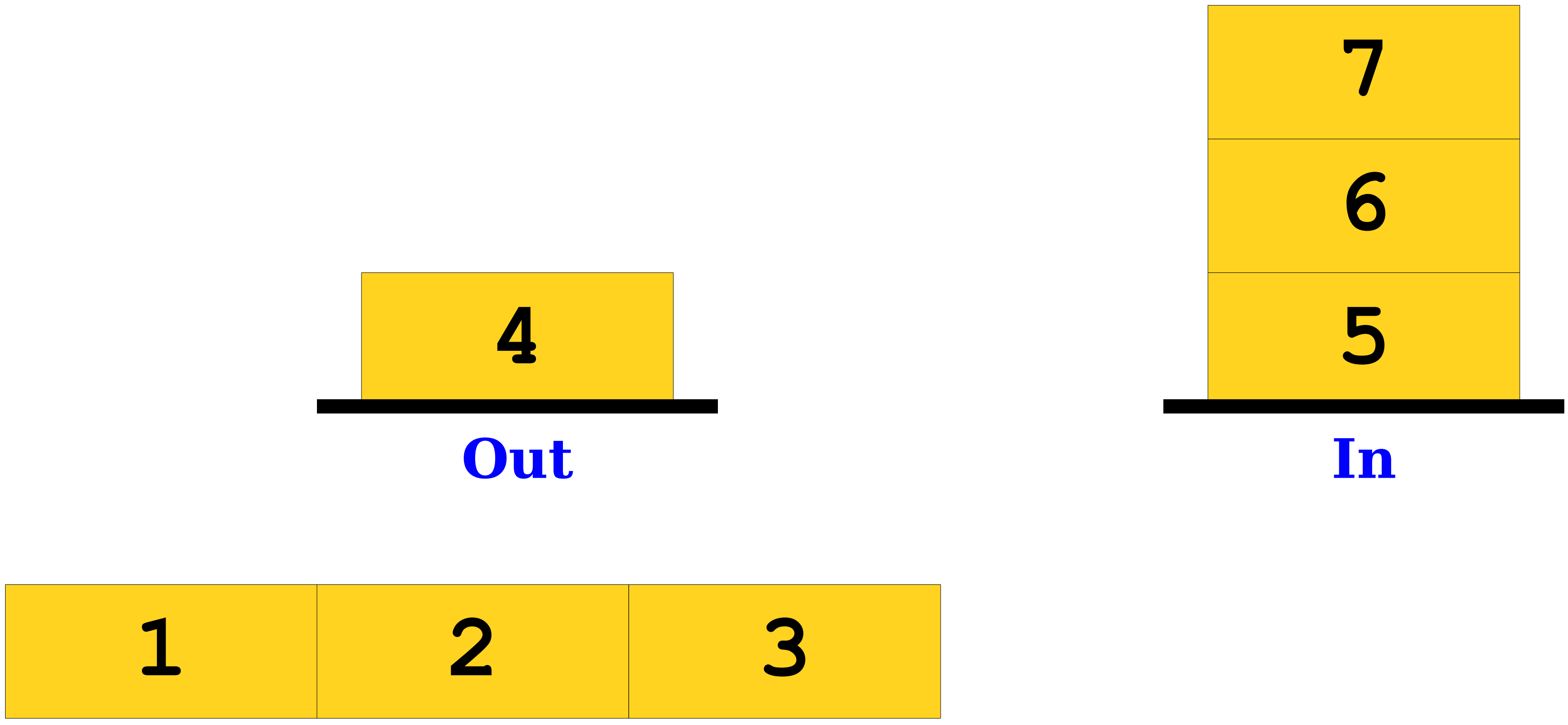
Two Worlds

- Data structures have different requirements in different contexts.
  - In real-time applications, each operation on a given data structure needs to be fast and snappy.
  - In long data processing pipelines, we care more about the total time used than we do the cost of any one operation.
- In many cases, we can get better performance in the long-run than we can on a per-operation basis.
  - Good intuition: “economy of scale.”

Key Idea: Design data structures that trade *per-operation efficiency* for *overall efficiency*.

Example: The Two-Stack Queue

The Two-Stack Queue

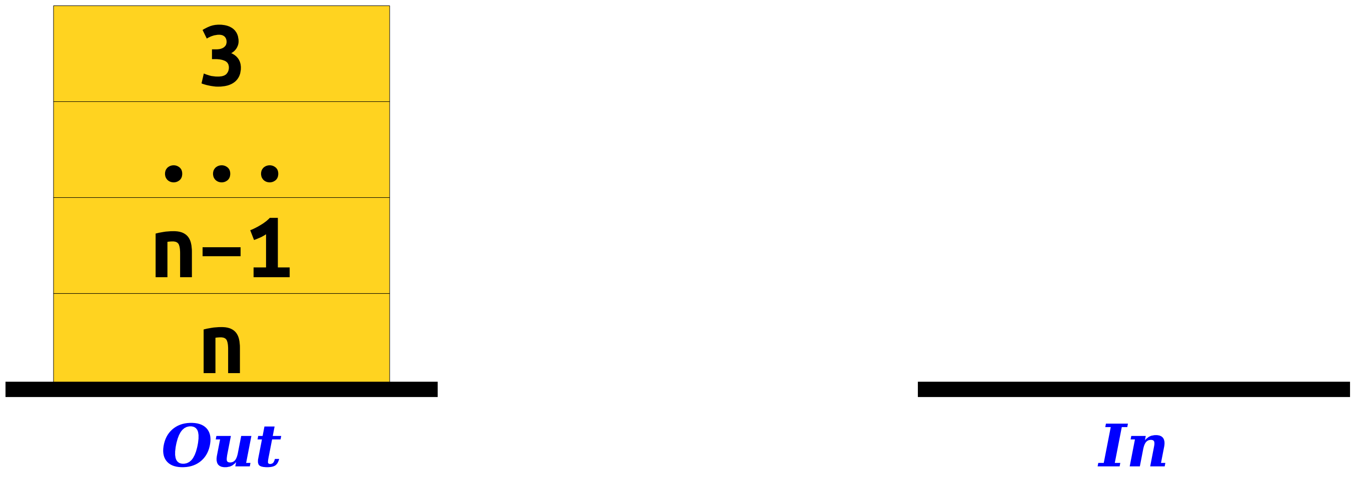


The Two-Stack Queue

- Maintain an ***In*** stack and an ***Out*** stack.
- To enqueue an element, push it onto the ***In*** stack.
- To dequeue an element:
  - If the ***Out*** stack is nonempty, pop it.
  - If the ***Out*** stack is empty, pop elements from the ***In*** stack, pushing them into the ***Out*** stack, until the bottom of the ***In*** stack is exposed.

The Two-Stack Queue

- Each enqueue takes time O(1).
  - Just push an item onto the ***In*** stack.
- Dequeues can vary in their runtime.
  - Could be O(1) if the ***Out*** stack isn't empty.
  - Could be Θ(n) if the ***Out*** stack is empty.

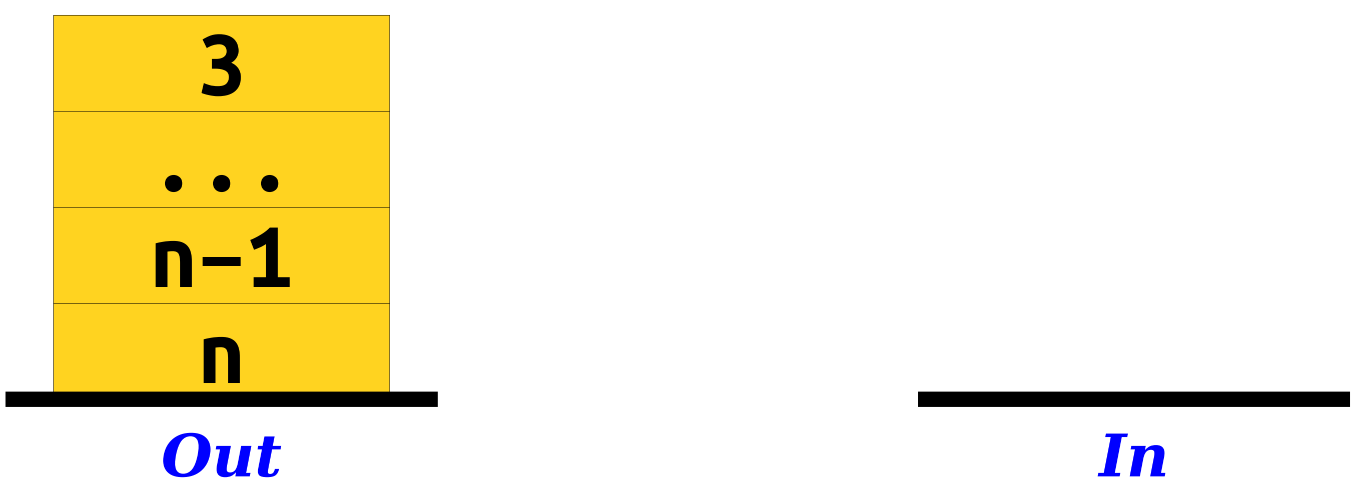


The Two-Stack Queue

- ***Intuition:*** We only do expensive dequeues after a long run of cheap dequeues.
- Think "carbon credits:" the fast enqueue operation introduces pollution that needs to be cleaned up every once and a while.
- Provided the cleanup is fast and pollution doesn't build up too quickly, this is a good idea!

The Two-Stack Queue

- Any series of m operations on a two-stack queue will take time $O(m)$.
- Every element is pushed at most twice and popped at most twice.
- ***Key Question:*** What's the best way to summarize the above idea in a useful way?
- This is a bit more subtle than it looks.

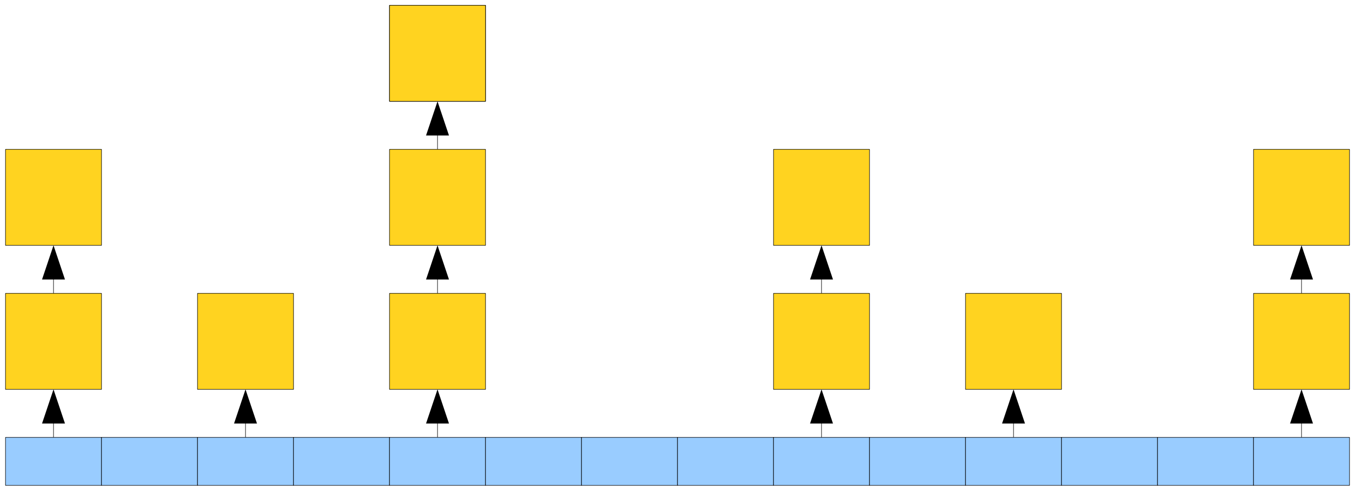


Analyzing the Queue

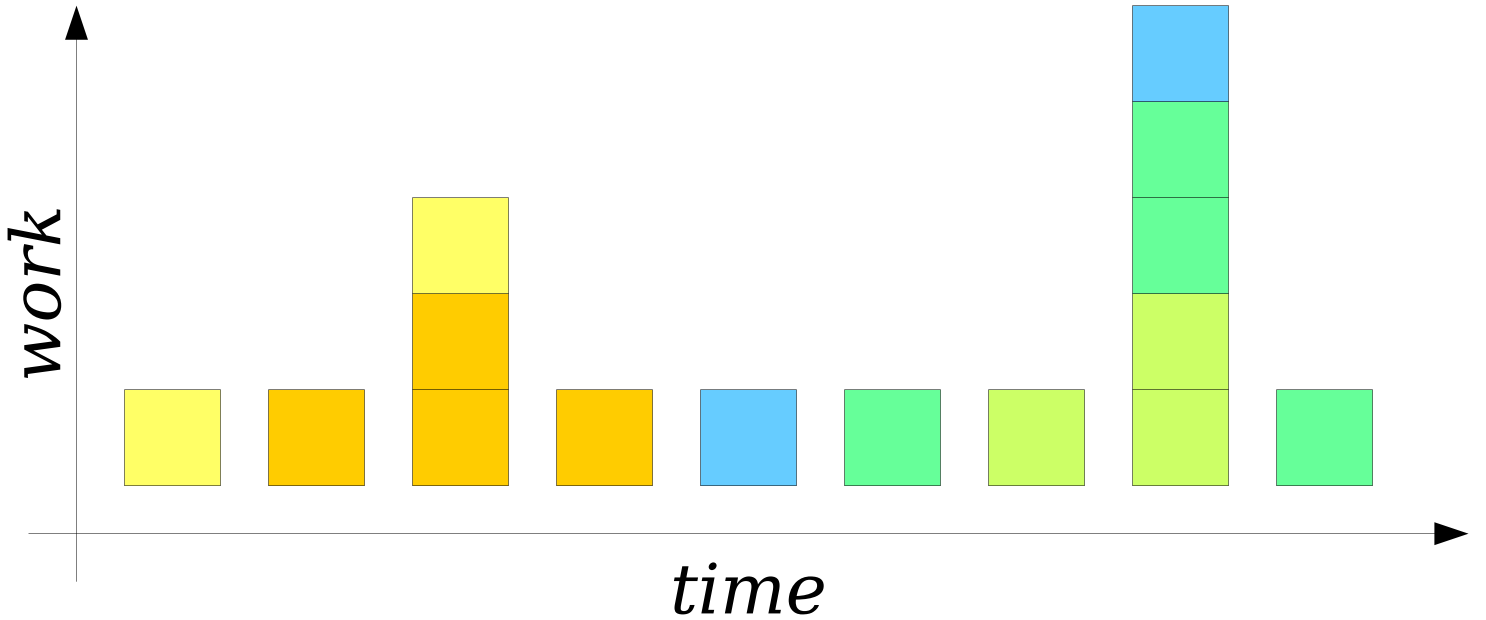
- ***Initial idea:*** Summarize our result using an average-case analysis.
  - If we do m total operations, the total work done is O(m).
  - Average amount of work per operation: O(1).
- Based on this argument, we can claim that the average cost of an enqueue or dequeue is O(1).
- ***Claim:*** While the above statement is true, it's not as precise as we might like.

The Problem with Averages

- Compare our two-stack queue to a chained hash table.



- The average cost of an insertion or lookup in a chained hash table with *n* elements is O(1).
- However, this use of “average” for a hash table means something different than the use of “average” for our two-stack queue.
- ***Why?***



Total work done: 15

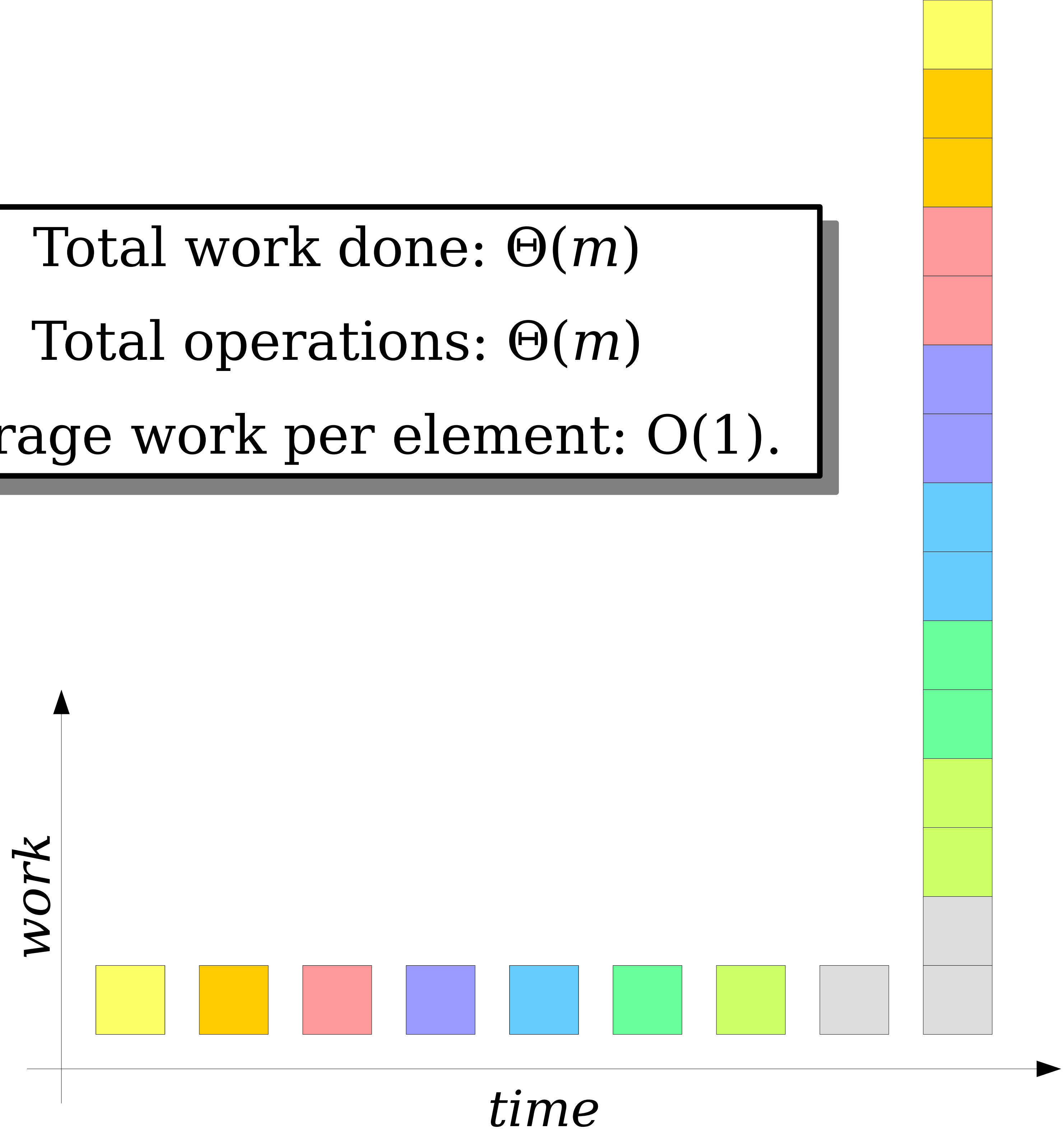
Total operations: 9

Average work per element: ≈1.66

Total work done: $\Theta(m)$

Total operations: $\Theta(m)$

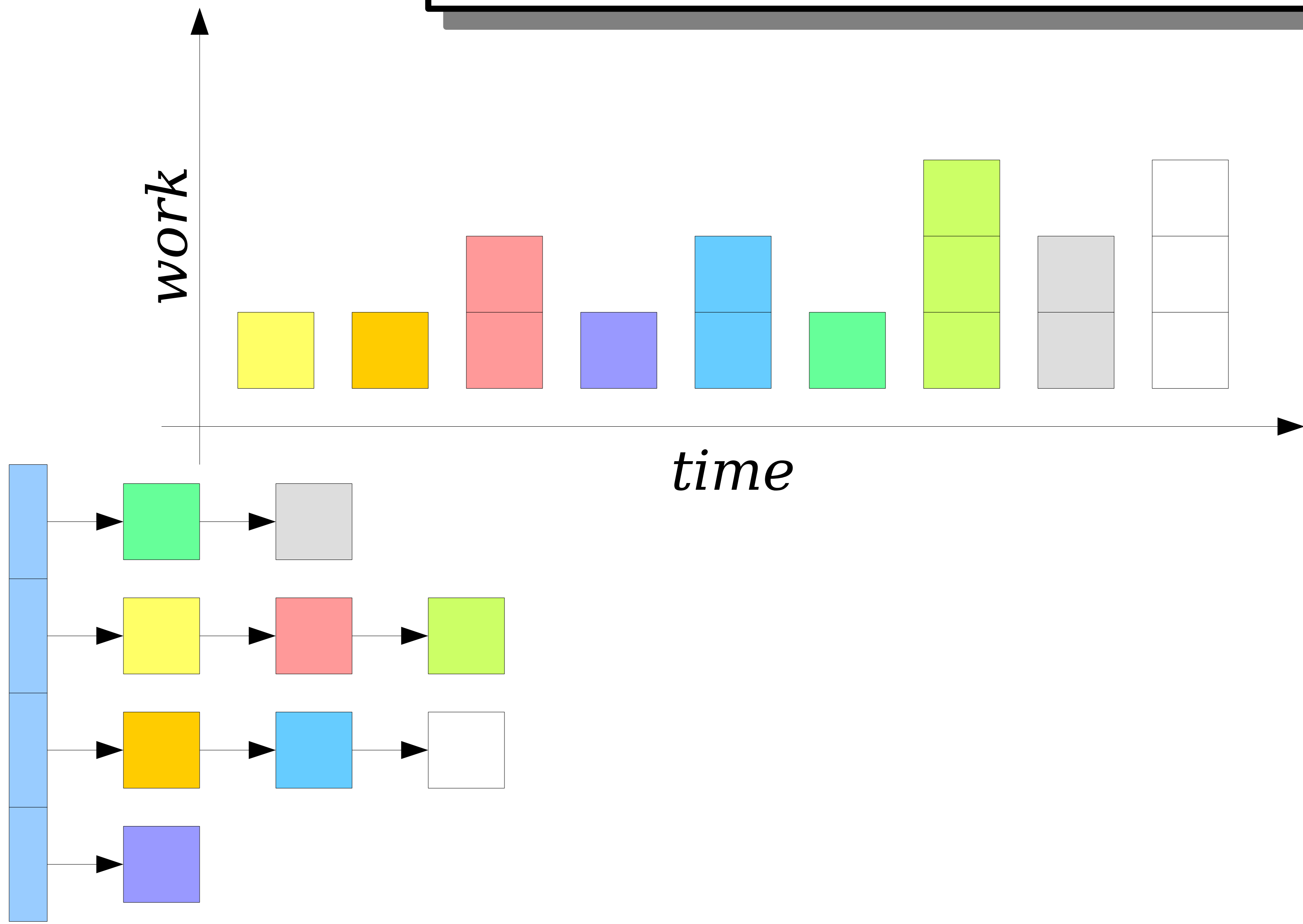
Average work per element: O(1).

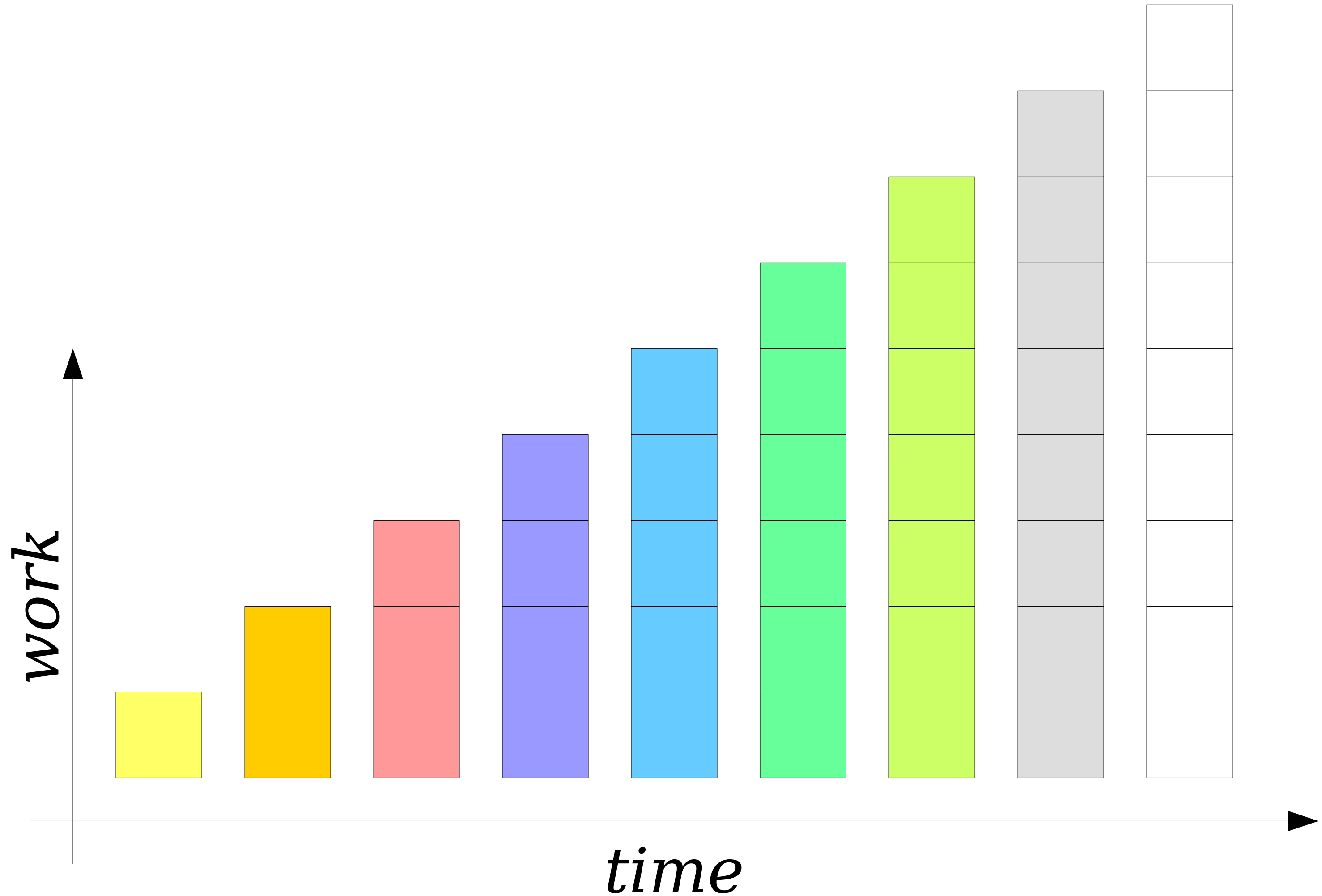


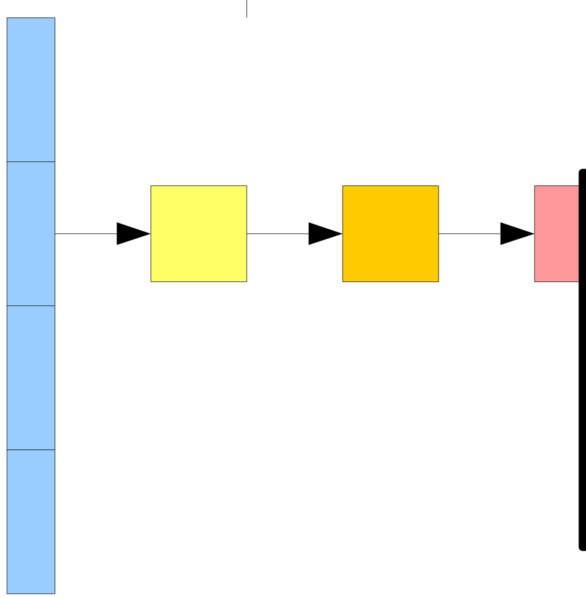
Total work done: 16

Total operations: 9

Average work per element: ≈1.8







Total work done: $\Theta(m^2)$

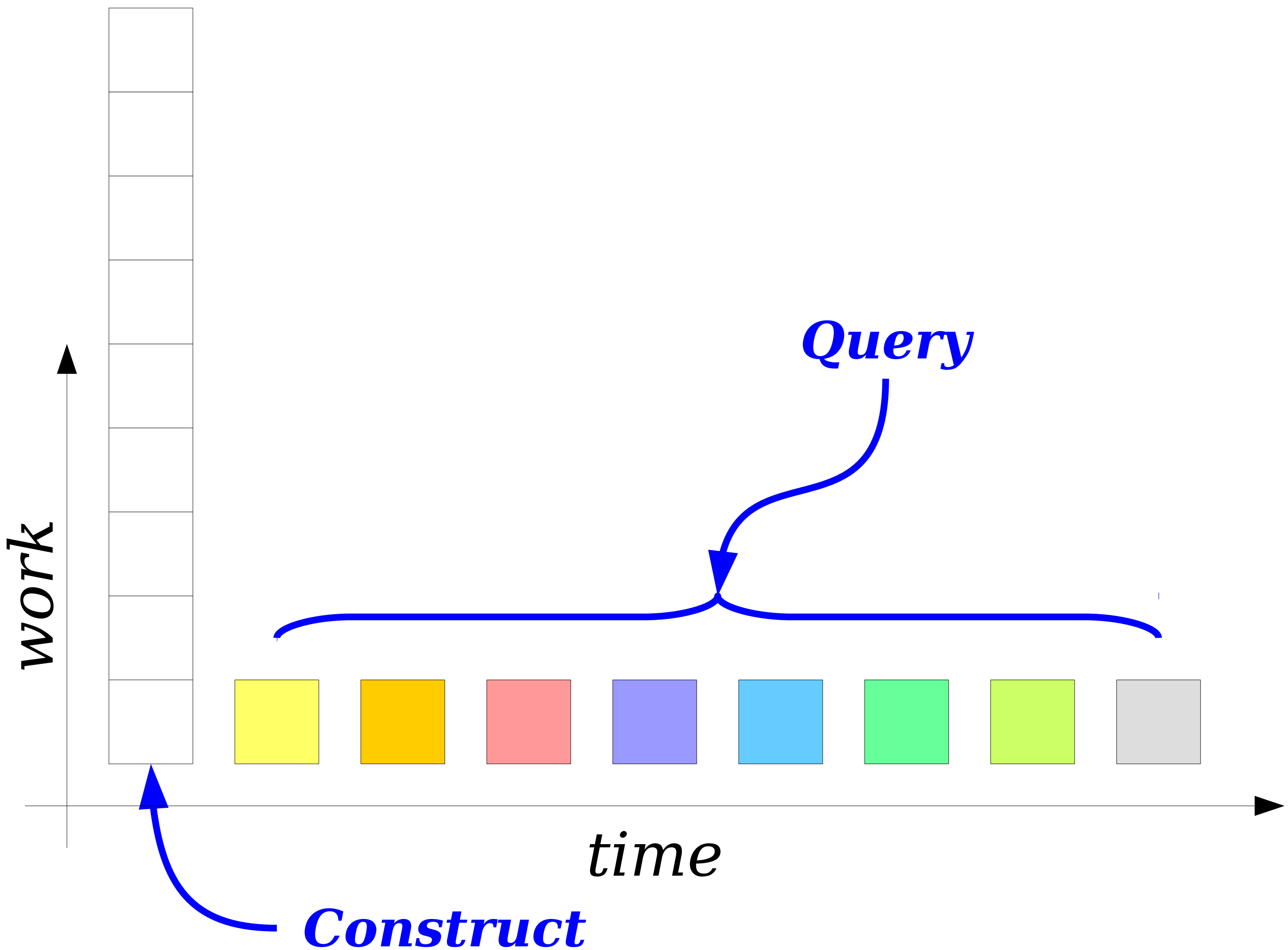
Total operations: $\Theta(m)$

Average work per element: $\Theta(m)$.

Issue 1: Terms like “average” or “expected” convey randomness. Our two-stack queue has zero probability of giving a long series of bad operations.

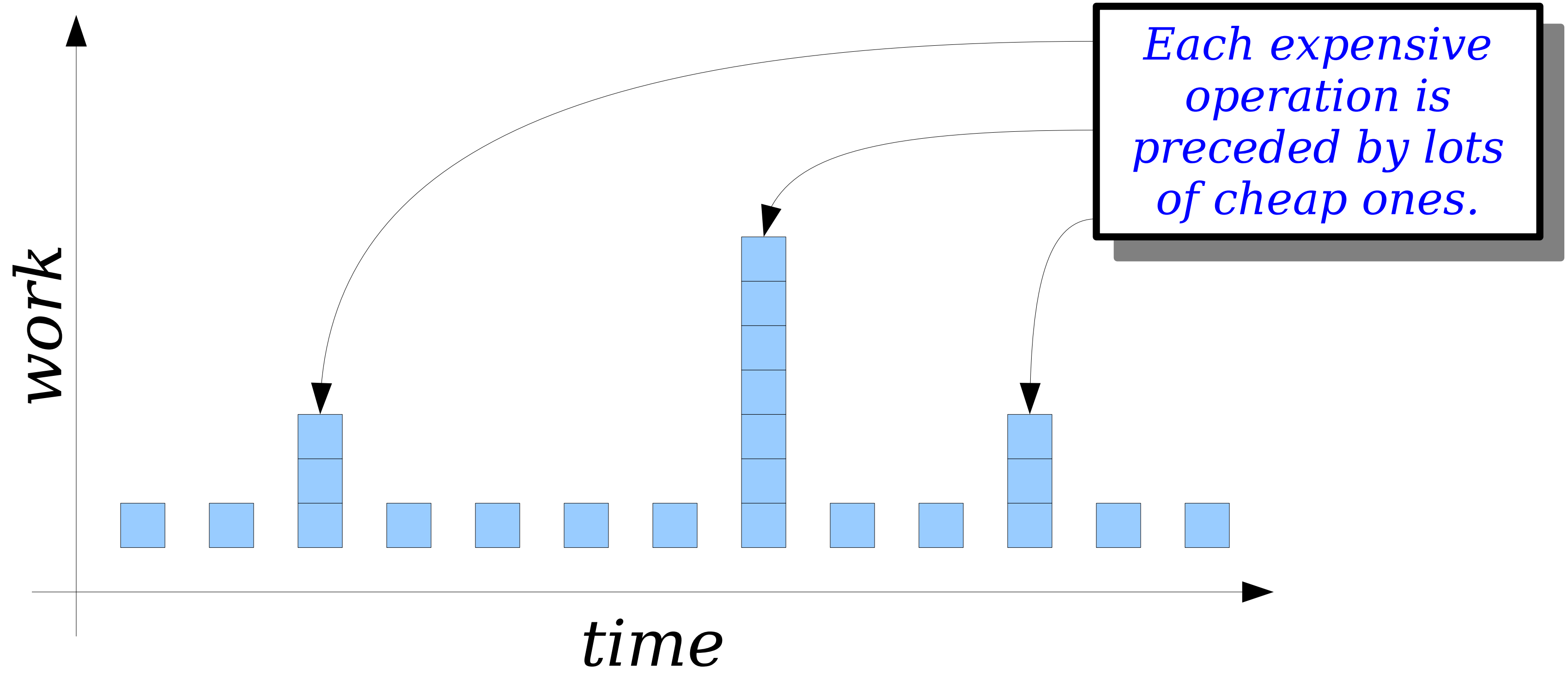
The Problem with Averages

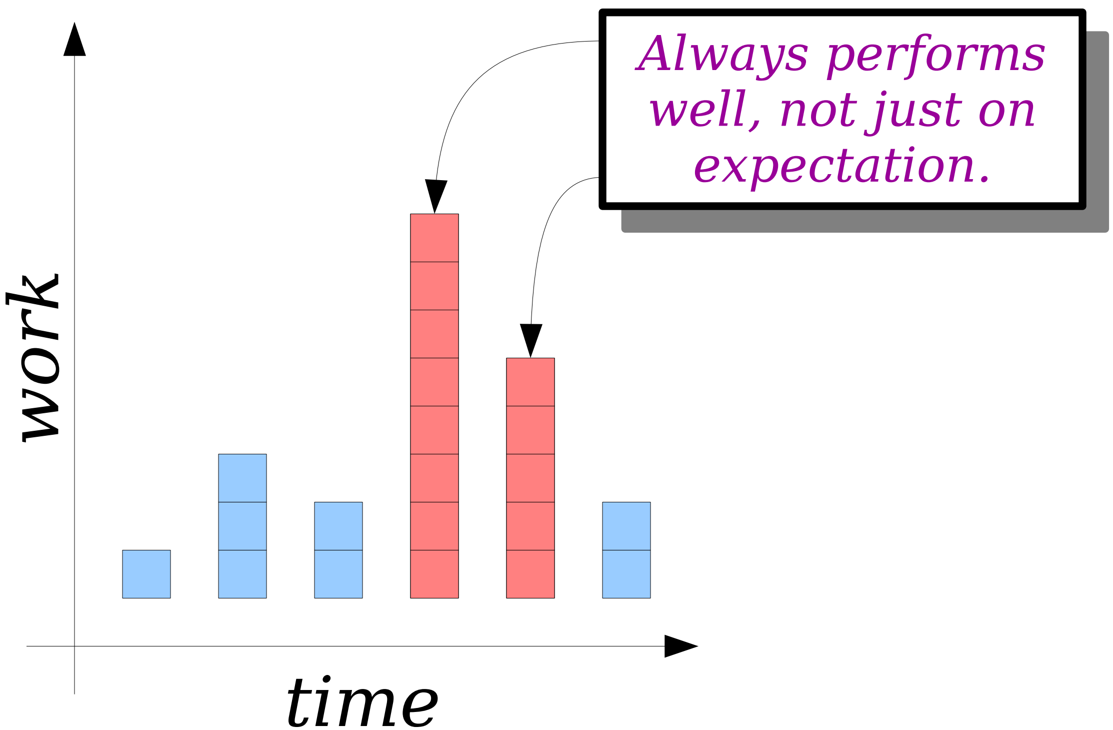
- I’m going to *(incorrectly!)* claim that the average cost of creating a Fischer-Heun structure or doing a query on a Fischer-Heun structure is O(1).
- ⚠ ***Argument:*** ⚠
  - Construct a Fischer-Heun structure on an array of length m in time $O(m)$.
  - Do $m - 1$ range minimum queries on it in total time $O(m)$.
  - Total work done is $O(m)$, and there were n operations performed.
  - Average cost of an operation (construct or query): O(1).
- Why doesn’t this argument work?
- How is this different from the two-stack queue?

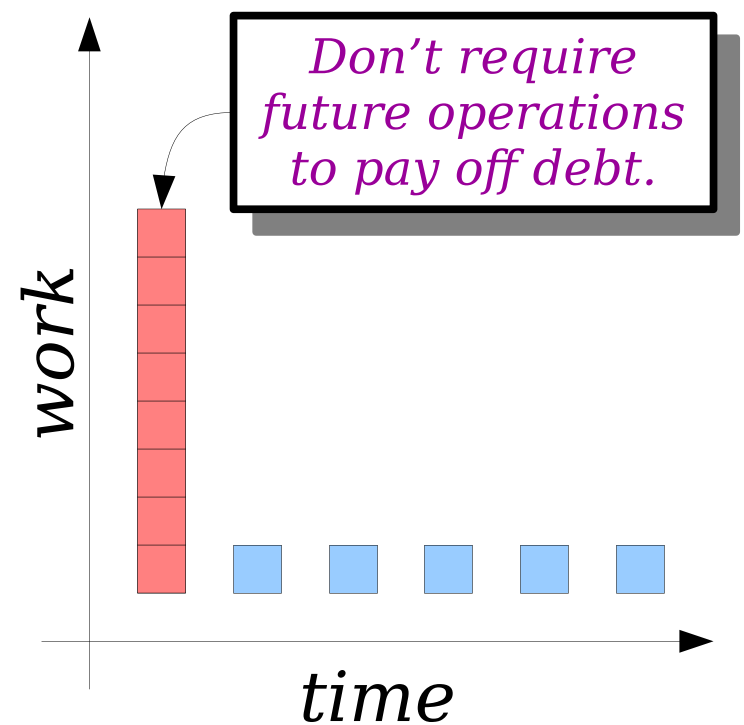

Query
work
time
Construct

Issue 2: It’s not just that the average operation time on a particular sequence is O(1). It’s true for *any* series of operations.

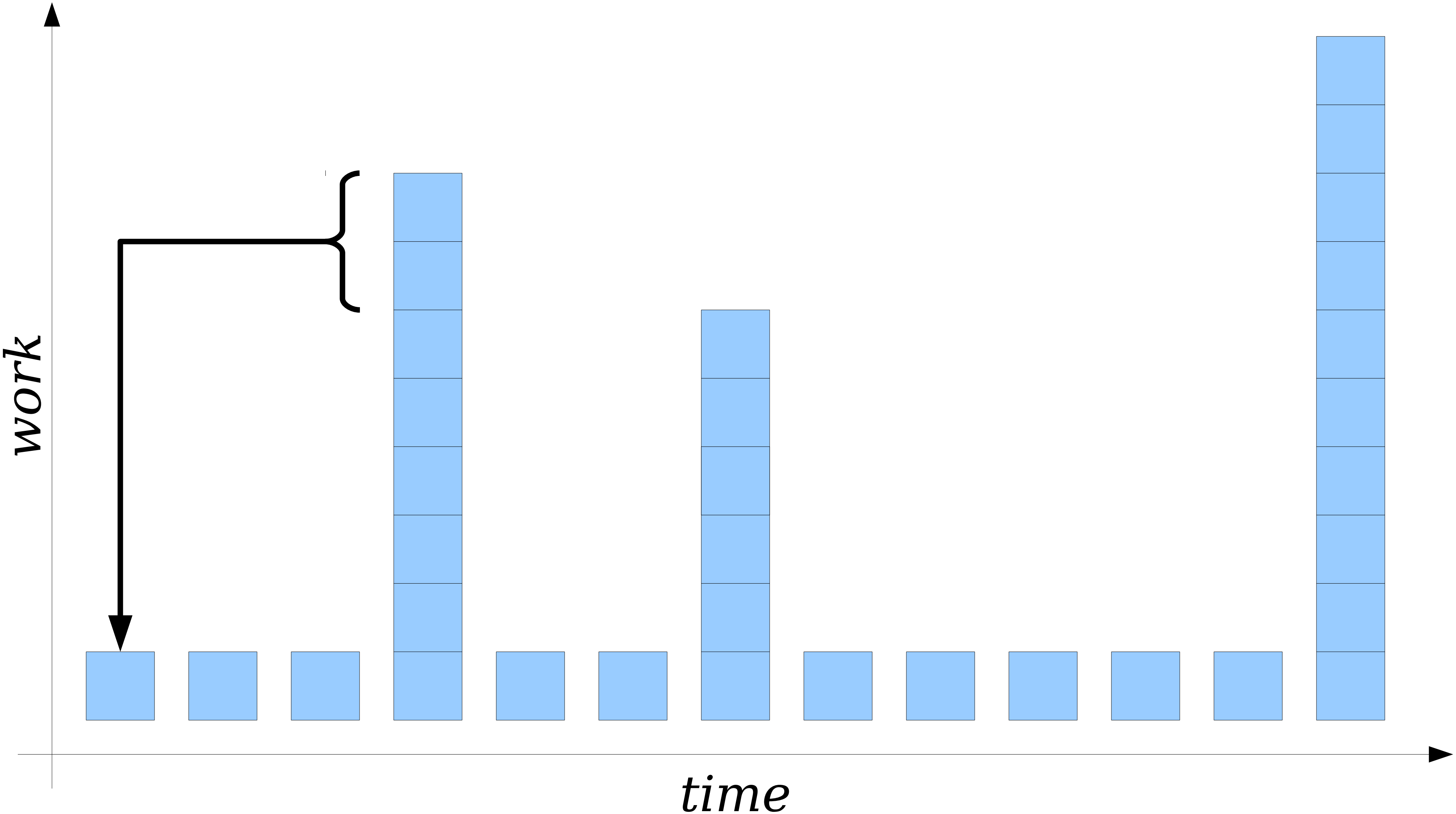
To Summarize


Each expensive operation is preceded by lots of cheap ones.
work
time

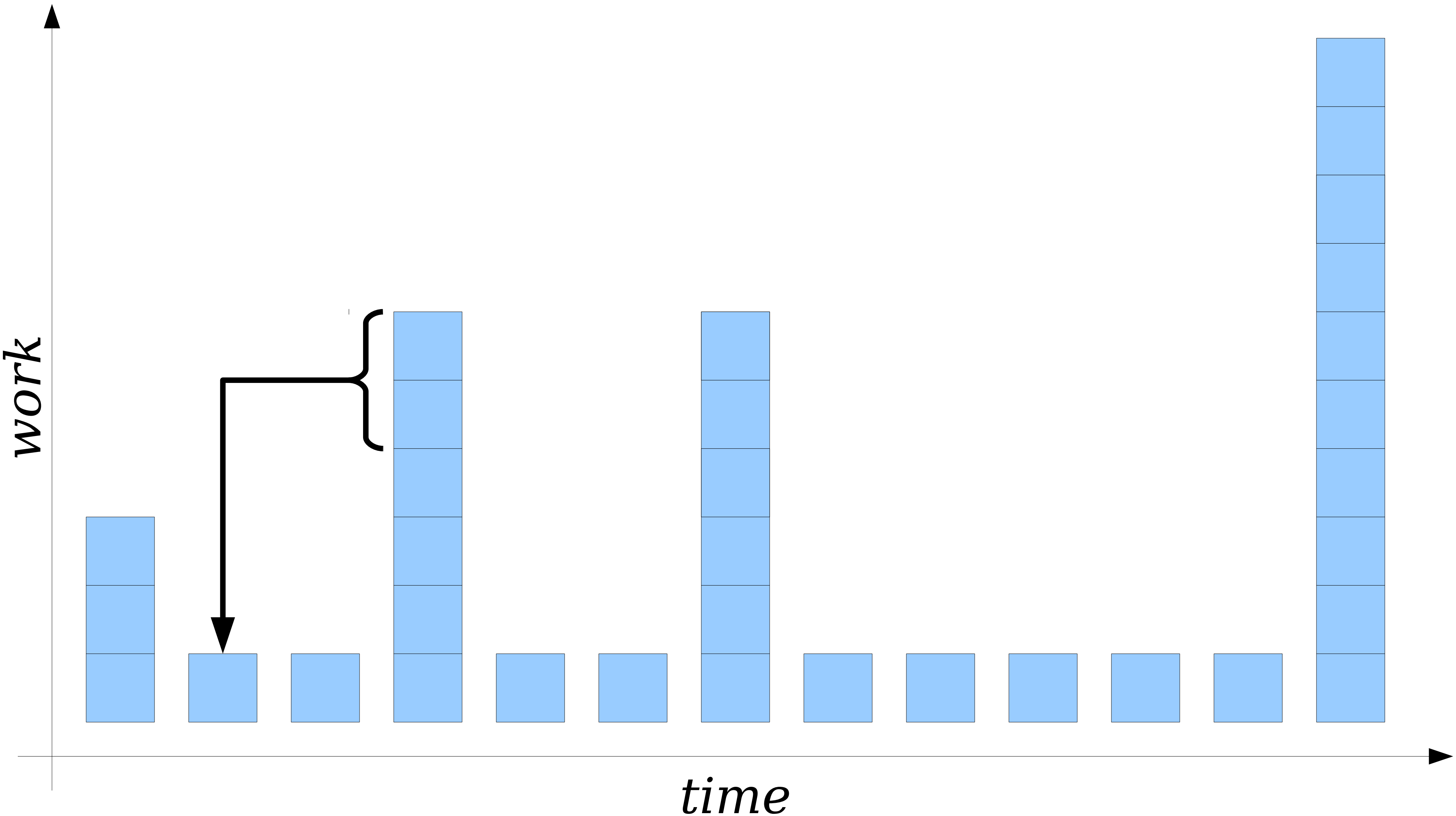

Always performs well, not just on expectation.
work
time


Don’t require future operations to pay off debt.
work
time

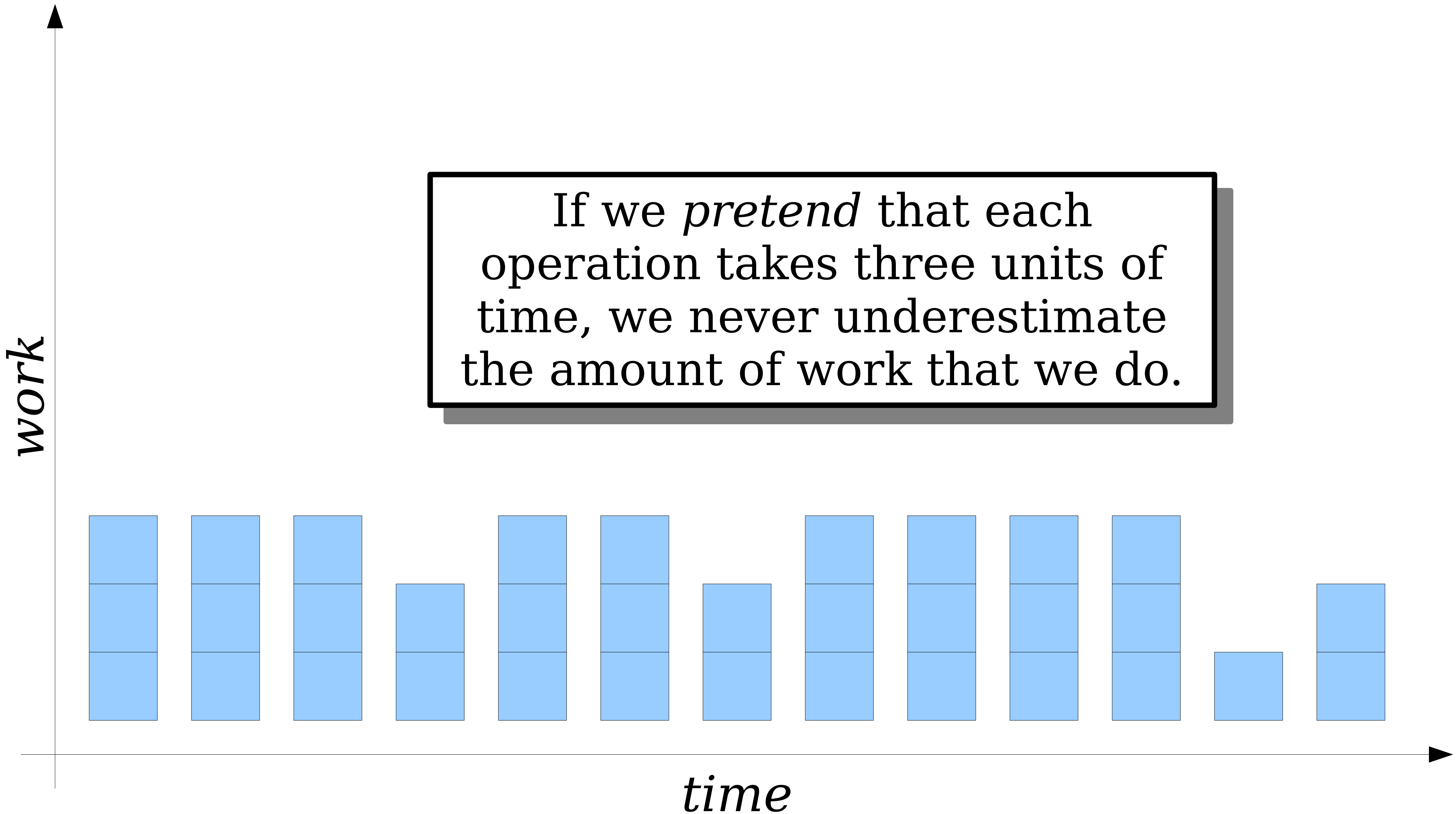
So What?



Key Idea: Backcharge expensive operations to cheaper ones.



Key Idea: Backcharge expensive operations to cheaper ones.



Key Idea: Backcharge expensive operations to cheaper ones.

Amortized Analysis

Amortized Analysis

- Suppose we perform a series of operations op_1, op_2, …, op_m.
- The amount of time taken to execute operation op_i is denoted by $\boldsymbol{t(op_i)}$.
- ***Goal:*** For each operation op_i, pick a value $\boldsymbol{a(op_i)}$, called the ***amortized cost*** of op_i, such that

$$\forall k \le m. \sum_{i=1}^{k} t(op_i) \le \sum_{i=1}^{k} a(op_i).$$

No matter when we stop performing operations…

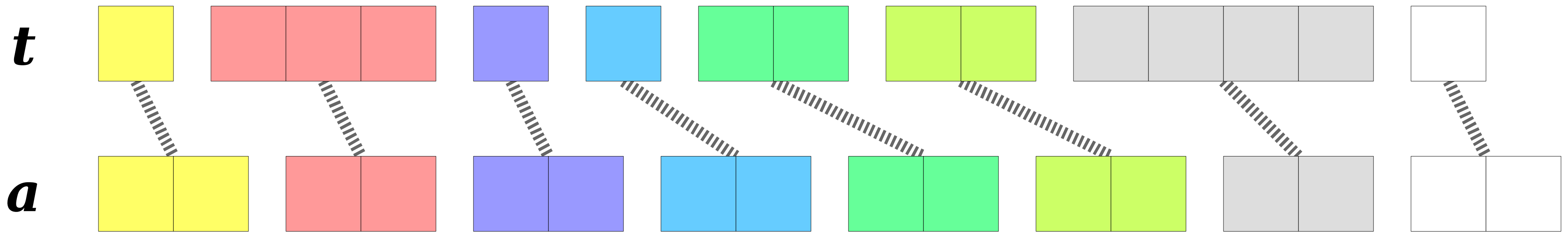
…the *actual* cost of performing those operations…

… is at most the *amortized* cost of performing those operations.

Amortized Analysis

- Suppose we perform a series of operations op_1, op_2, …, op_m.
- The amount of time taken to execute operation op_i is denoted by ***t(opᵢ)***.
- ***Goal:*** For each operation op_i, pick a value ***a(opᵢ)***, called the ***amortized cost*** of op_i, such that

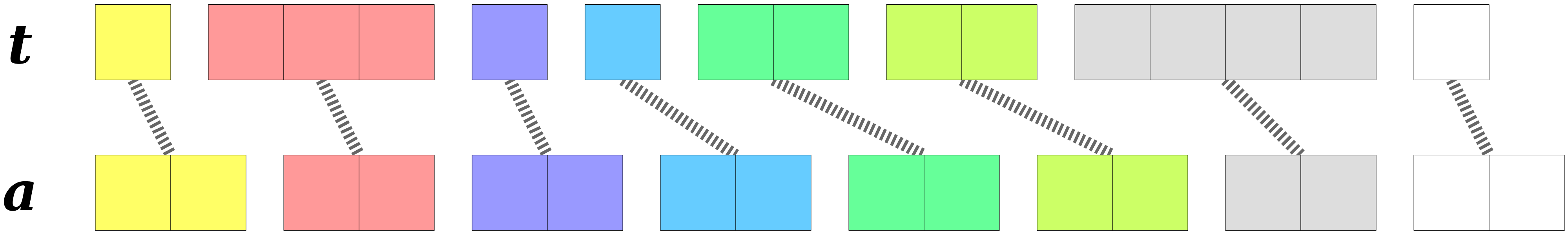
$$\forall k \leq m.\ \sum_{i=1}^{k} t(op_i) \leq \sum_{i=1}^{k} a(op_i).$$



Amortized Analysis

- The ***amortized*** cost of an enqueue or dequeue in a two-stack queue is O(1).
- ***Intuition:*** If you pretend that the *actual* cost of each enqueue or dequeue is O(1), you will never overestimate the total time spent performing queue operations.

$$\forall k \leq m. \sum_{i=1}^{k} t(op_i) \leq \sum_{i=1}^{k} a(op_i).$$



Amortized Analysis

- It’s helpful to contrast different ways of handling expensive operations:
- Preprocessing/runtime tradeoffs:

  “Yes, we have to do a lot of work, but it’s a one-time cost and everything is cheaper after that.”

- Randomization:

  “*We might have to do a lot of work, but it’s unlikely that we’ll do so.”*

- Amortization:

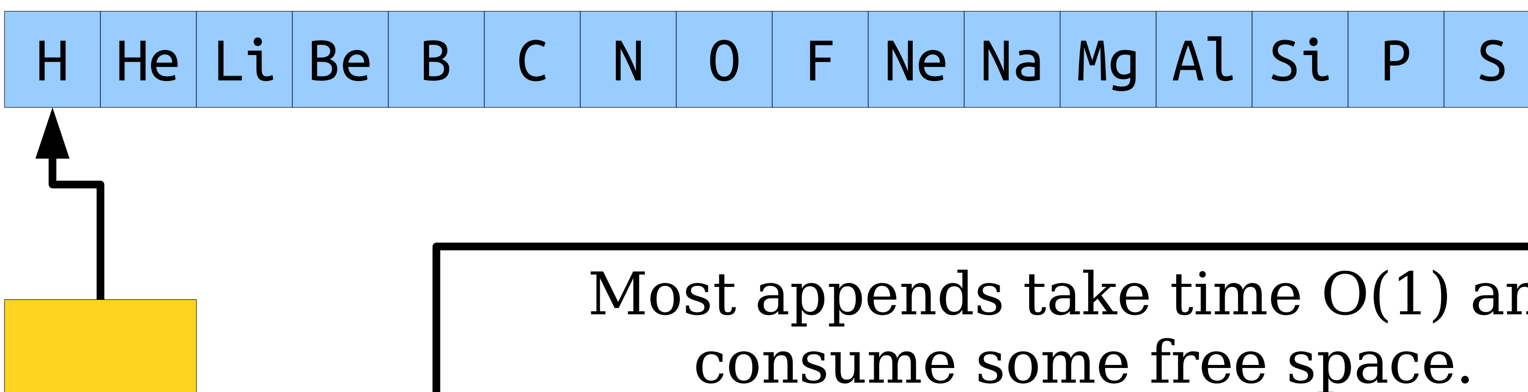
  “Yes, we have to do a lot of work every once and a while, but only after a period of doing very little.”

Major Questions

- In what situations can we nicely amortize the cost of expensive operations?
- How do we choose the amortized costs we want to use?
- How do we design data structures with amortization in mind?

When Amortization Works

When Amortization Works

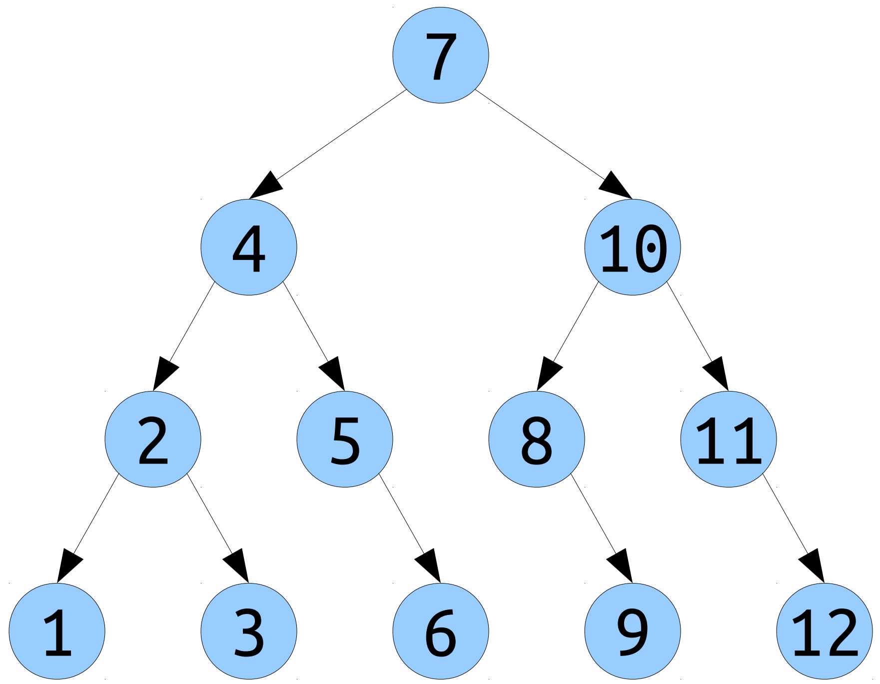


Most appends take time O(1) and consume some free space.

Every now and then, an append takes time O(*n*), but produce a lot of free space.

With a little math, you can show that the ***amortized*** cost of *any* append is O(1).

When Amortization Works



Most insertions take time O(log *n*) and unbalance the tree. Some insertions do more work, but balance large parts of the tree.

With the right strategy for rebuilding trees, *all* insertions can be shown to run in ***amortized*** time O(log *n*) each. (This is called a ***scapegoat tree***.)

Key Intuition: Amortization works best if

(1) imbalances accumulate slowly, and
(2) imbalances get cleaned up quickly.

Performing Amortized Analyses

Performing Amortized Analyses

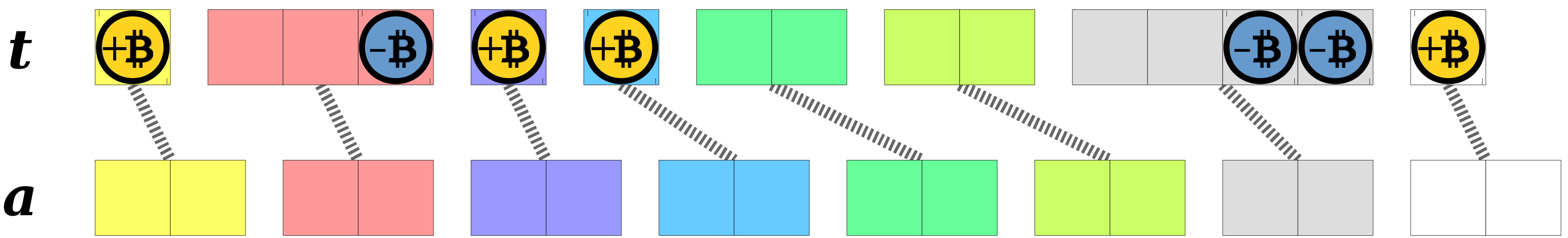
- You have a data structure where
  - imbalances accumulate slowly, and
  - imbalances get cleaned up quickly.
- You’re fairly sure the cleanup costs will amortize away nicely.
- How do you assign amortized costs?

The Banker's Method

- In the ***banker's method***, operations can place ***credits*** on the data structure or spend credits that have already been placed.
- Placing a credit on the data structure takes time O(1).
- Spending a credit previously placed on the data structure takes time -O(1). *(Yes, that’s negative time!)*
- The amortized cost of an operation is then

$$a(op_i) = t(op_i) + O(1) \cdot (added_i - removed_i)$$

- There aren’t any real credits anywhere. They’re just an accounting trick.



The Banker's Method

$$
\begin{aligned}
\sum_{i=1}^{k} a(op_i) &= \sum_{i=1}^{k} \left(t(op_i) + O(1)\cdot(added_i - removed_i)\right) \\
&= \sum_{i=1}^{k} t(op_i) + O(1)\sum_{i=1}^{k} (added_i - removed_i) \\
&= \sum_{i=1}^{k} t(op_i) + O(1)\left(\sum_{i=1}^{k} added_i - \sum_{i=1}^{k} removed_i\right) \\
&= \sum_{i=1}^{k} t(op_i) + O(1)\cdot(net\ credits\ added) \\
&\geq \sum_{i=1}^{k} t(op_i)
\end{aligned}
$$

(Assuming we never spend credits we don’t have.)

The Two-Stack Queue

Actual work: O(1)
Credits added: 1

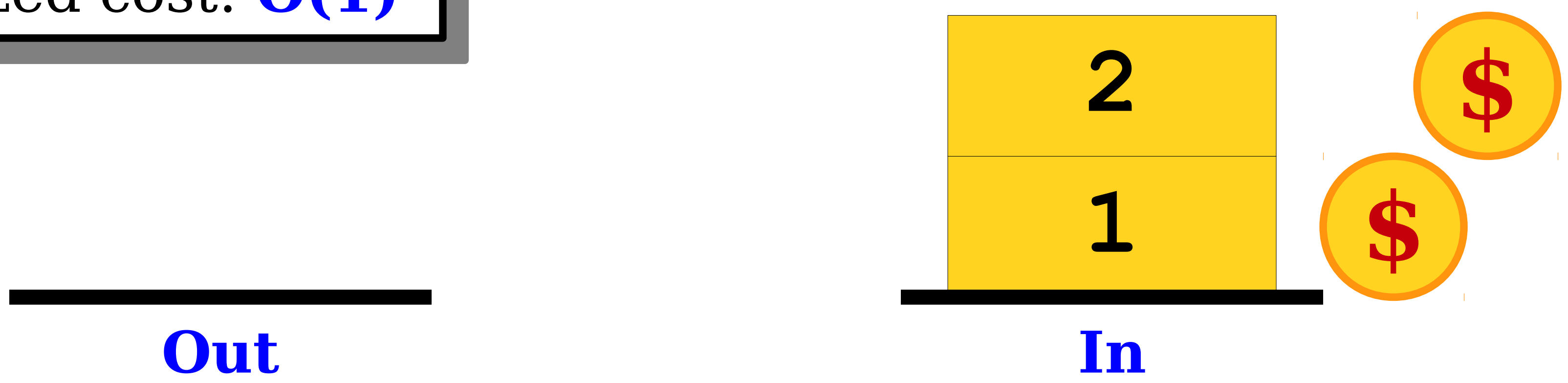
Amortized cost: **O(1)**

This credit will pay for the work to pop this element later on and push it onto the other stack.

The Two-Stack Queue

Actual work: O(1)
Credits added: 1

Amortized cost: **O(1)**



The Two-Stack Queue

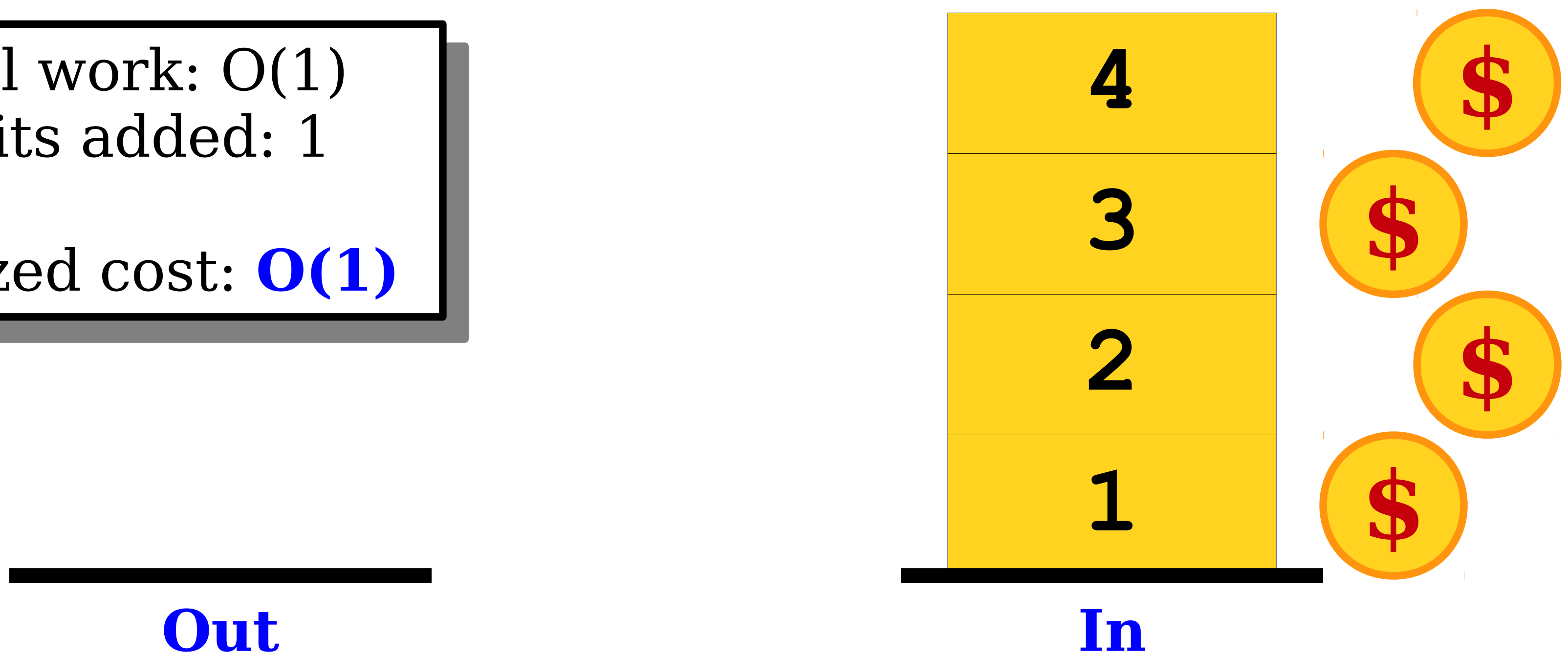
Actual work: O(1)
Credits added: 1

Amortized cost: **O(1)**

The Two-Stack Queue

Actual work: O(1)
Credits added: 1

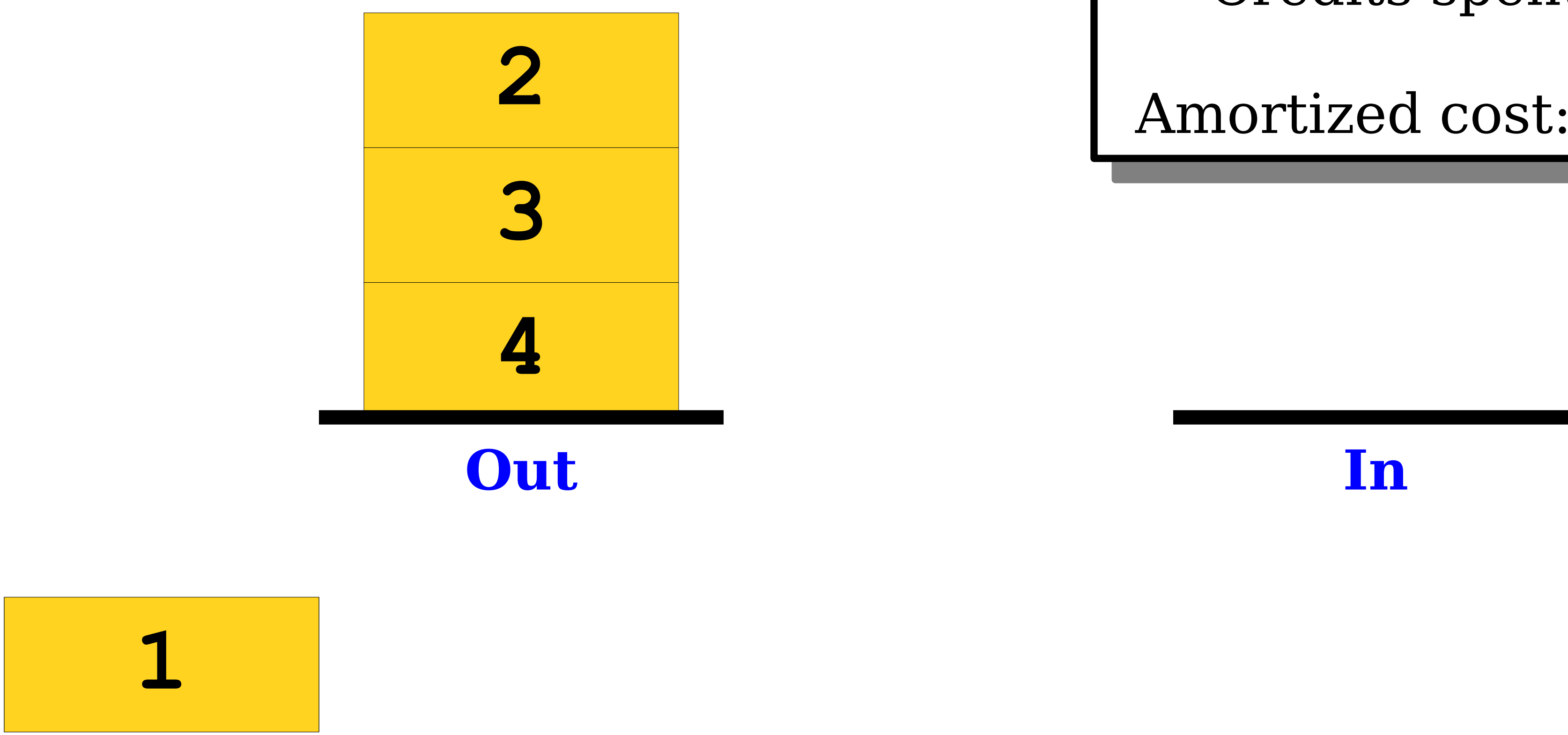
Amortized cost: **O(1)**



The Two-Stack Queue

Actual work: $\Theta(k)$
Credits spent: k

Amortized cost: **O(1)**



An Observation

- The amortized cost of an operation is

$$a(op_i) = t(op_i) + O(1) \cdot (added_i - removed_i)$$

- Equivalently, this is

$$a(op_i) = t(op_i) + O(1) \cdot \Delta credits_i.$$

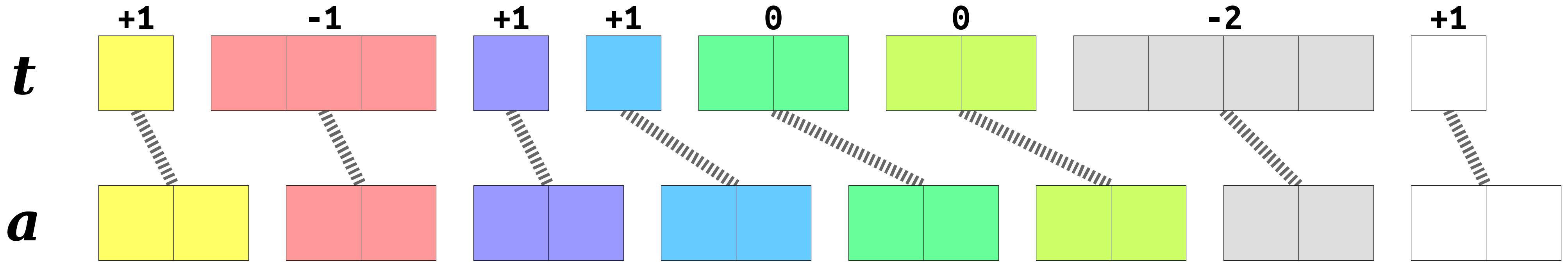
- Some observations:
  - It doesn't matter where these credits are placed or removed from.
  - The total number of credits added and removed doesn't matter; all that matters is the *difference* between these two.

The Potential Method

- In the ***potential method***, we define a ***potential function*** Φ that maps a data structure to a non-negative real value.
- Define $a(op_i)$ as

$$\mathbf{a(op_i) = t(op_i) + O(1) \cdot \Delta\Phi_i}$$

- Here, $\Delta\Phi_i$ is the change in the value of Φ during the execution of operation op_i.



The Potential Method

$$\sum_{i=1}^{k} a(op_i) = \sum_{i=1}^{k} \left(t(op_i) + O(1) \cdot \Delta\Phi_i\right)$$

$$= \sum_{i=1}^{k} t(op_i) + O(1) \cdot \sum_{i=1}^{k} \Delta\Phi_i$$

Think "fundamental theorem of calculus,"
but for discrete derivatives!

$$\int_a^b f'(x)\,dx = f(b) - f(a)$$

$$\sum_{x=a}^{b} \Delta f(x) = f(b+1) - f(a)$$

Look up ***finite calculus*** if you're curious to learn more!

The Potential Method

$$\sum_{i=1}^{k} a(op_i) = \sum_{i=1}^{k} \left(t(op_i) + O(1) \cdot \Delta\Phi_i\right)$$

$$= \sum_{i=1}^{k} t(op_i) + O(1) \cdot \sum_{i=1}^{k} \Delta\Phi_i$$

$$= \sum_{i=1}^{k} t(op_i) + O(1) \cdot (\textit{net change in potential})$$

$$\geq \sum_{i=1}^{k} t(op_i)$$

(Assuming our potential doesn't end up below where it started)

The Two-Stack Queue

Φ = Height of ***In*** Stack

Actual work: O(1)
$\Delta\Phi$: +1

Amortized cost: **O(1)**

The Two-Stack Queue

Φ = Height of ***In*** Stack

Actual work: O(1)
ΔΦ: +1

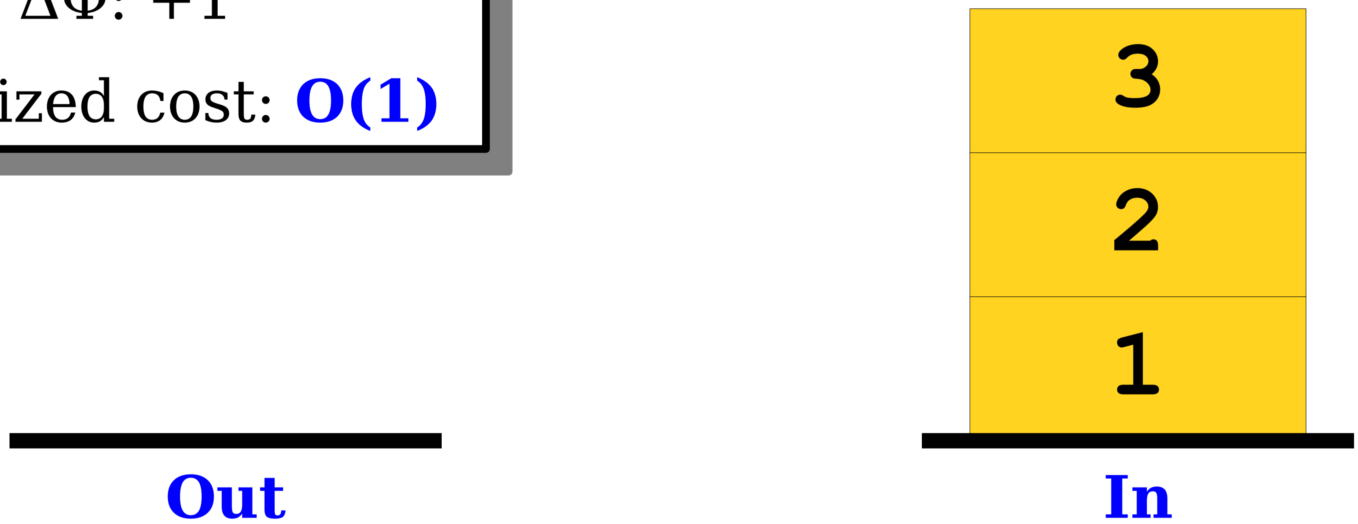
Amortized cost: **O(1)**

The Two-Stack Queue

Φ = Height of ***In*** Stack

Actual work: O(1)
ΔΦ: +1

Amortized cost: **O(1)**

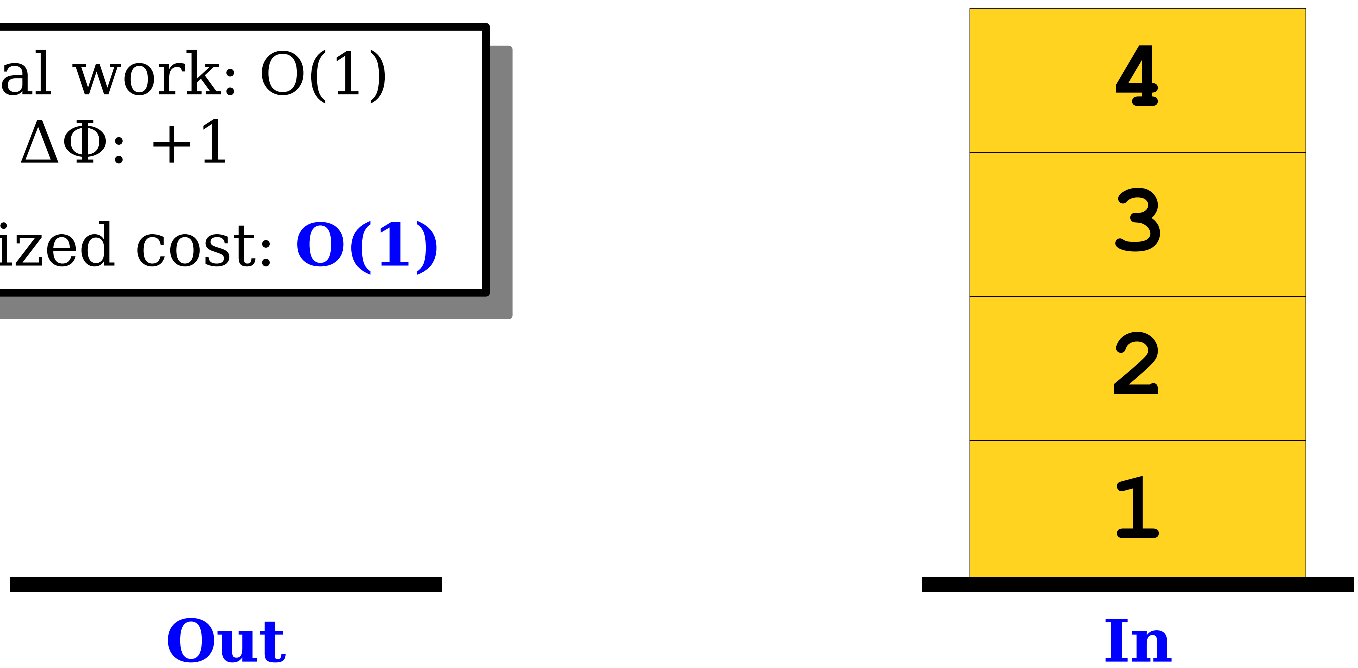


The Two-Stack Queue

Φ = Height of ***In*** Stack

Actual work: O(1)
ΔΦ: +1

Amortized cost: **O(1)**

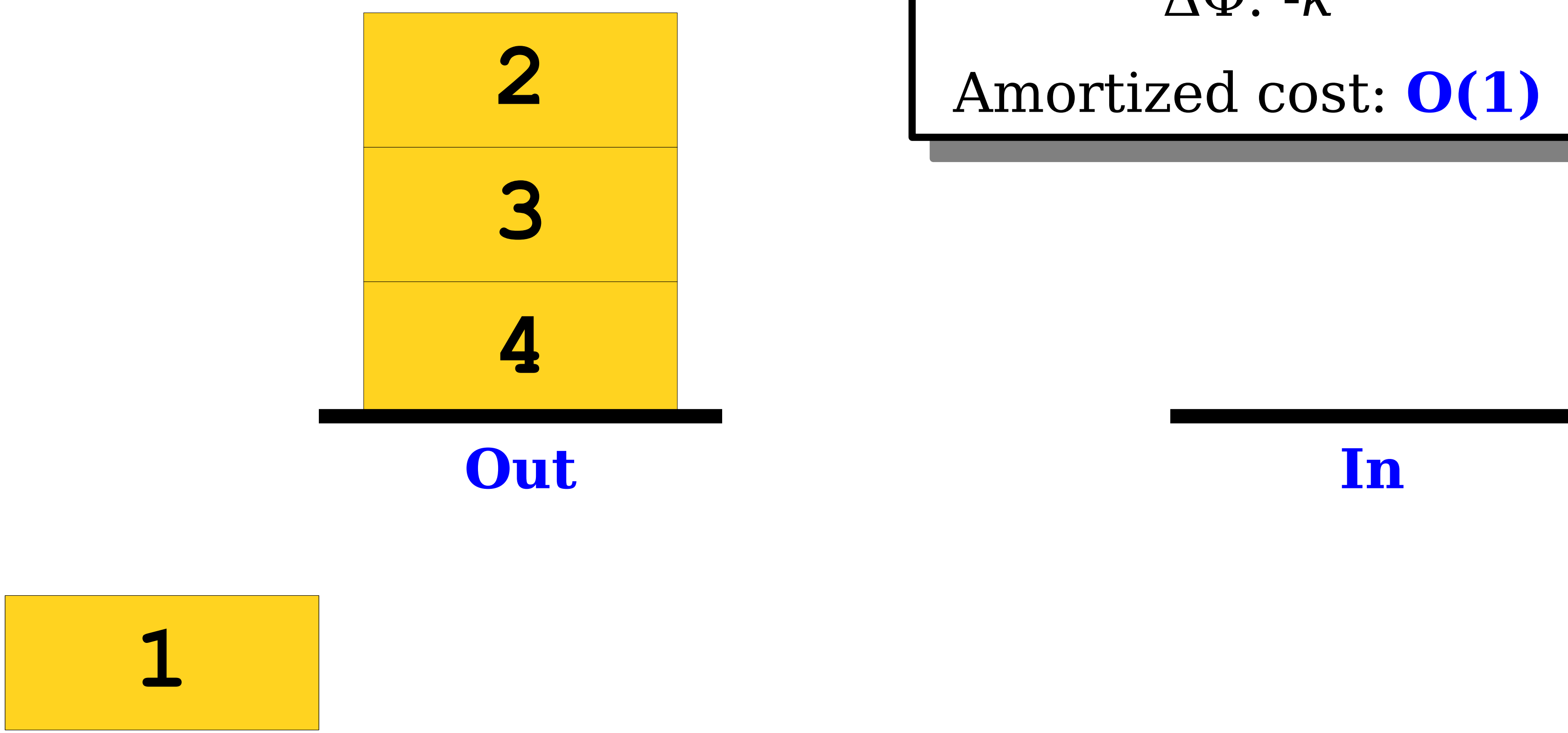


The Two-Stack Queue

Φ = Height of ***In*** Stack

Actual work: $\Theta(k)$
$\Delta\Phi$: $-k$

Amortized cost: **O(1)**



The Story So Far

- We assign ***amortized costs*** to operations, which are different than their real costs.
- The requirement is that the sum of the amortized costs never underestimates the sum of the real costs.
- The ***banker's method*** works by placing credits on the data structure and adjusting costs based on those credits.
- The ***potential method*** works by assigning a potential function to the data structure and adjusting costs based on the change in potential.

Time-Out for Announcements!

Problem Sets

- Problem Set Two was due at 2:30PM.
  - Need more time? Use a late period to extend the deadline to Saturday at 2:30PM.
- Problem Set Three goes out today. It's due on Tuesday, May 7th.
  - Play around with balanced and augmented trees!
  - Explore data structure isometries and multiway trees!
  - See how everything fits together!

Project Proposals

- Proposals for the final project are due next Thursday, May 2nd, at 2:30PM.
  - ***No late periods may be used here***. We'll be doing a global matchmaking and will want to give everyone as much lead time as possible.
- What you need to do:
  - Find a team of three people. (If you want to work in a pair, you'll need to email us to get permission.)
  - Rank your top four project topics and find sources for each.
- Looking for topics to pick from? Check out Handout 10, "Suggested Final Project Topics."
- Looking for teammates? Use Piazza's "Search for Teammates" feature!

Back to CS166!

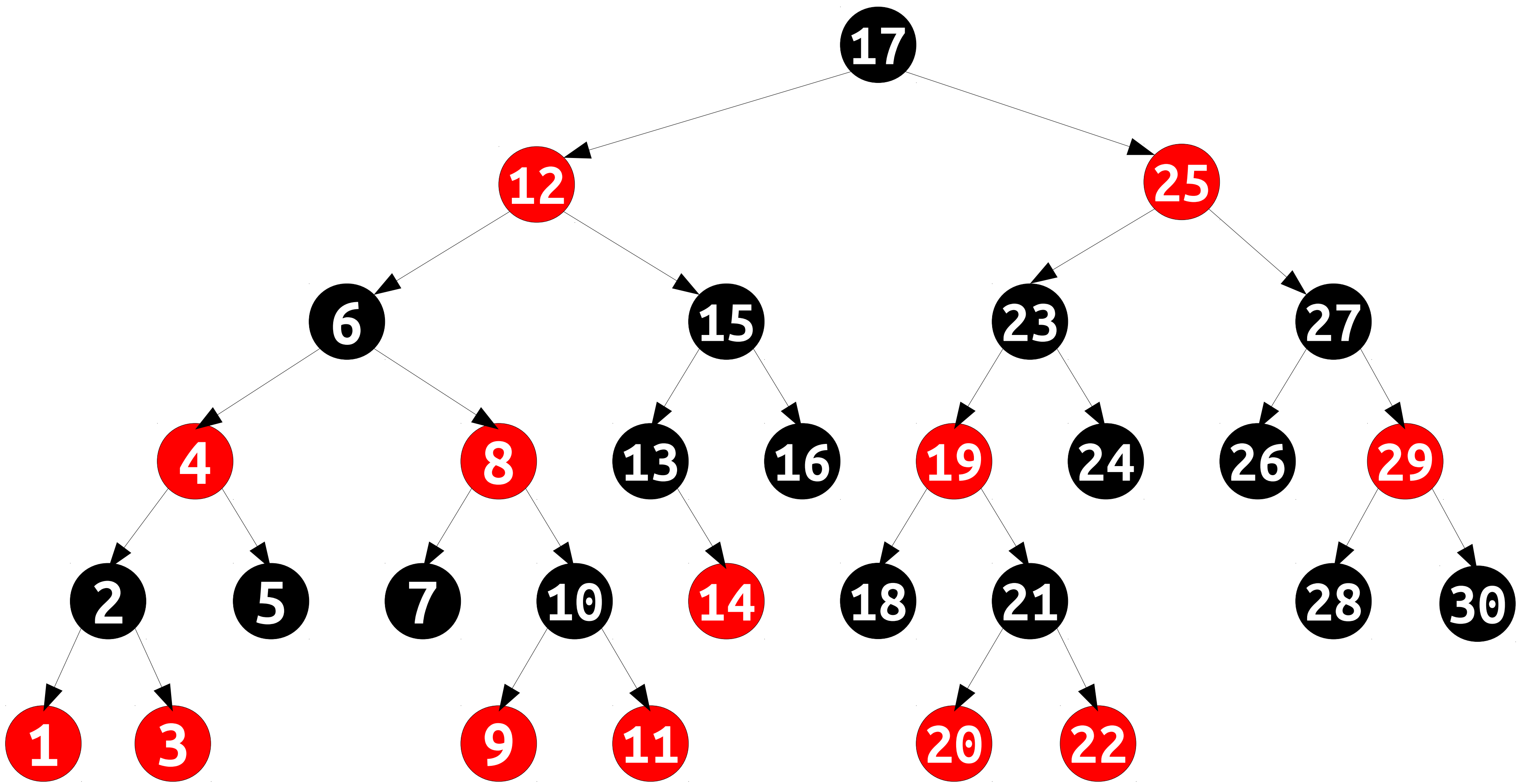
Deleting from a BST

BST Deletions

- We've seen how to do insertions into a 2-3-4 tree.
  - Put the key into the appropriate leaf.
  - Keep splitting big nodes and propagating keys upward as necessary.
- Using our isometry, we can use this to derive insertion rules for red/black trees.
- ***Question:*** How do you delete from a 2-3-4 tree or red/black tree?

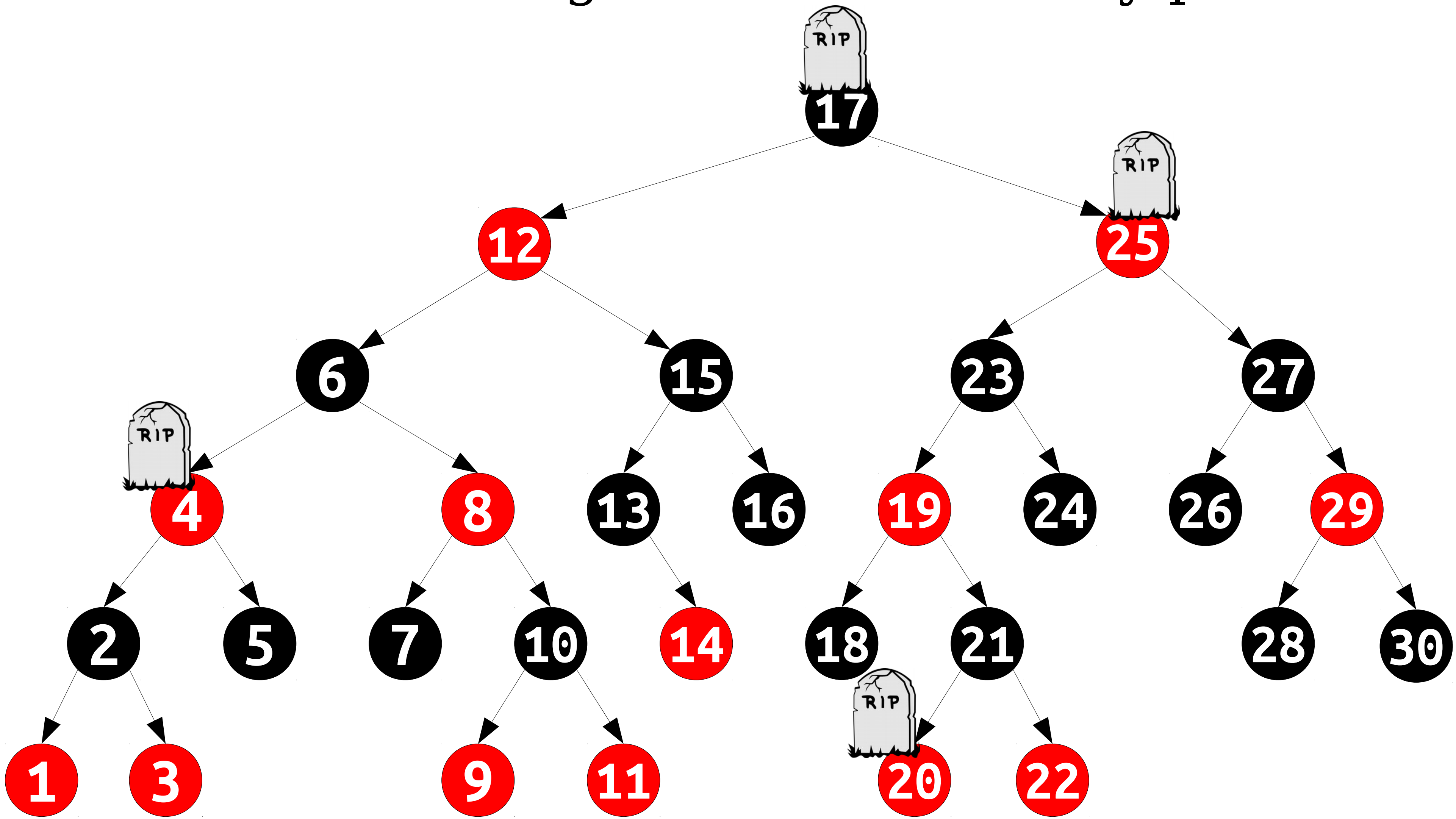
BST Deletions

- ***Question:*** How do we delete 20 from this tree? How about 4? How about 25? How about 17?



Dead Simple Deletions

- ***Idea:*** Delete things in the laziest way possible.

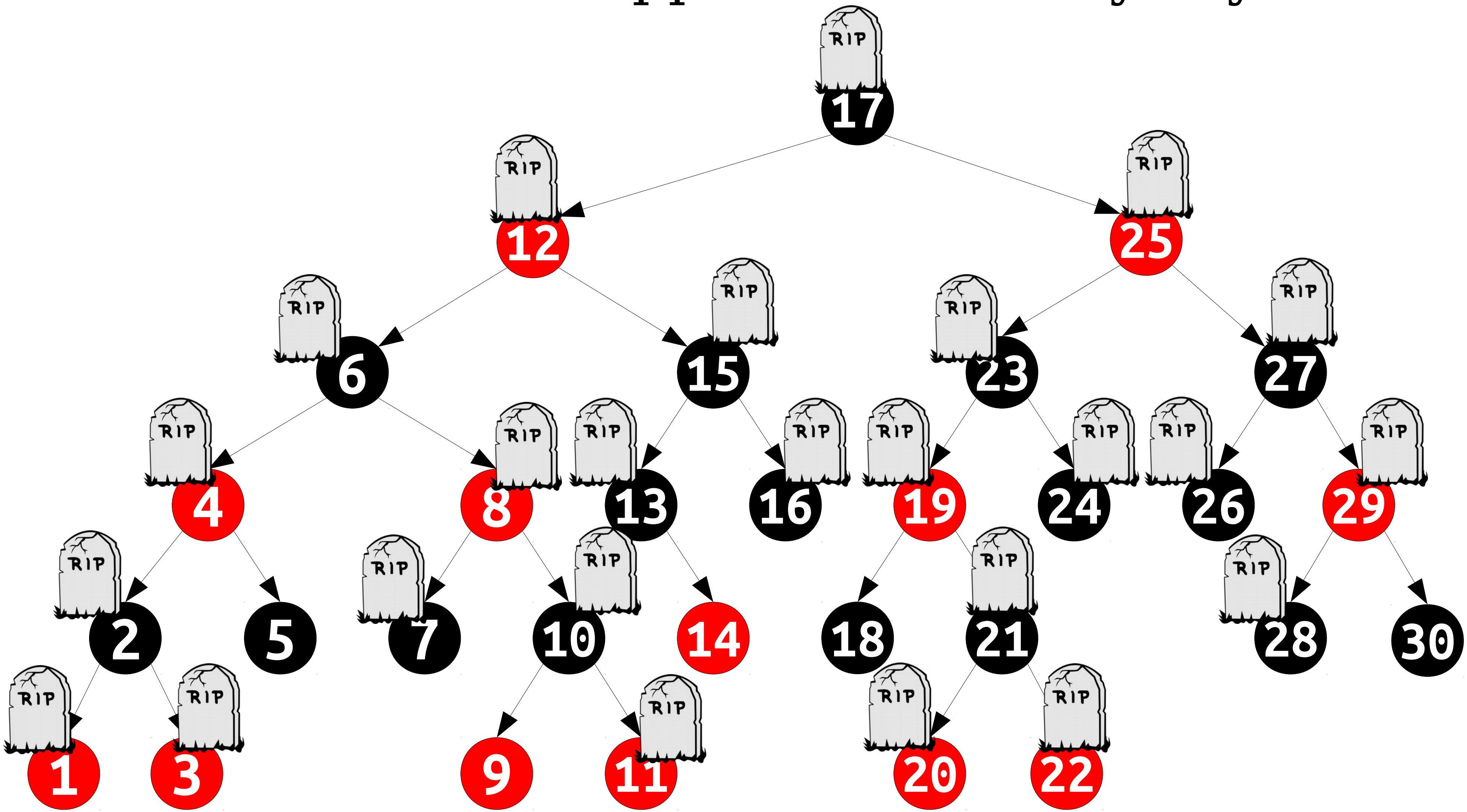


Dead Simple Deletions

- Each key is either ***dead*** (removed) or ***alive*** (still there).
- To remove a key, just mark it dead.
- Do lookups as usual, but pretend missing keys aren’t there.
- When inserting, if a dead version of the key is found, resurrect it.

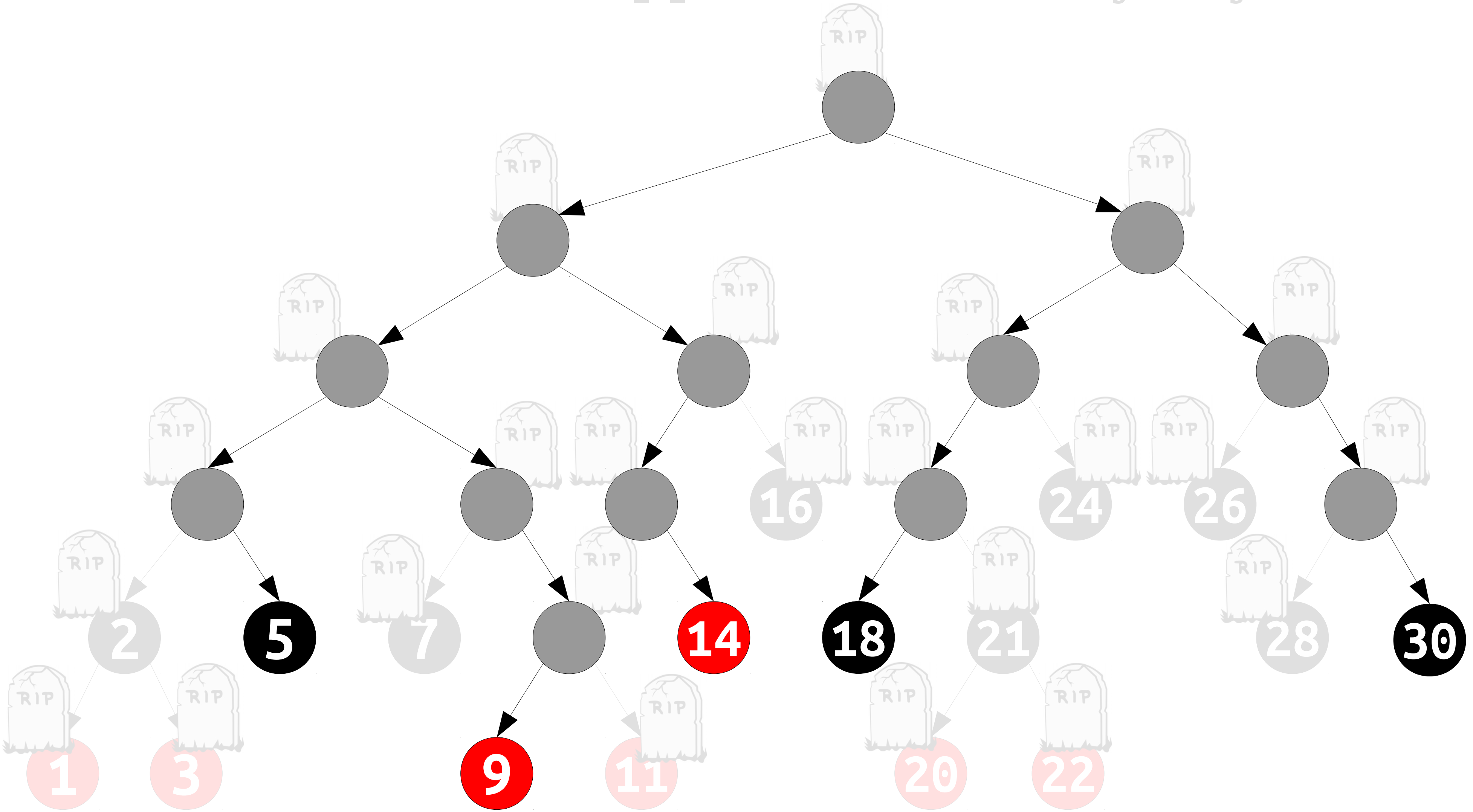
Dead Simple Deletions

- ***Problem:*** What happens if too many keys die?



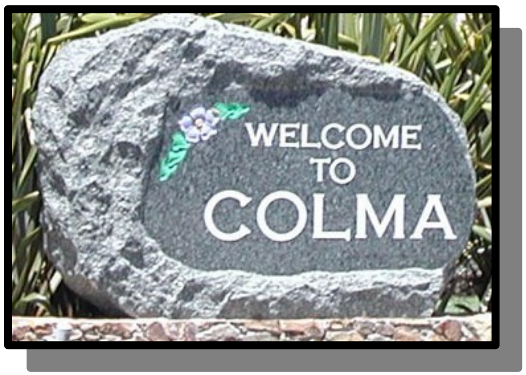
Dead Simple Deletions

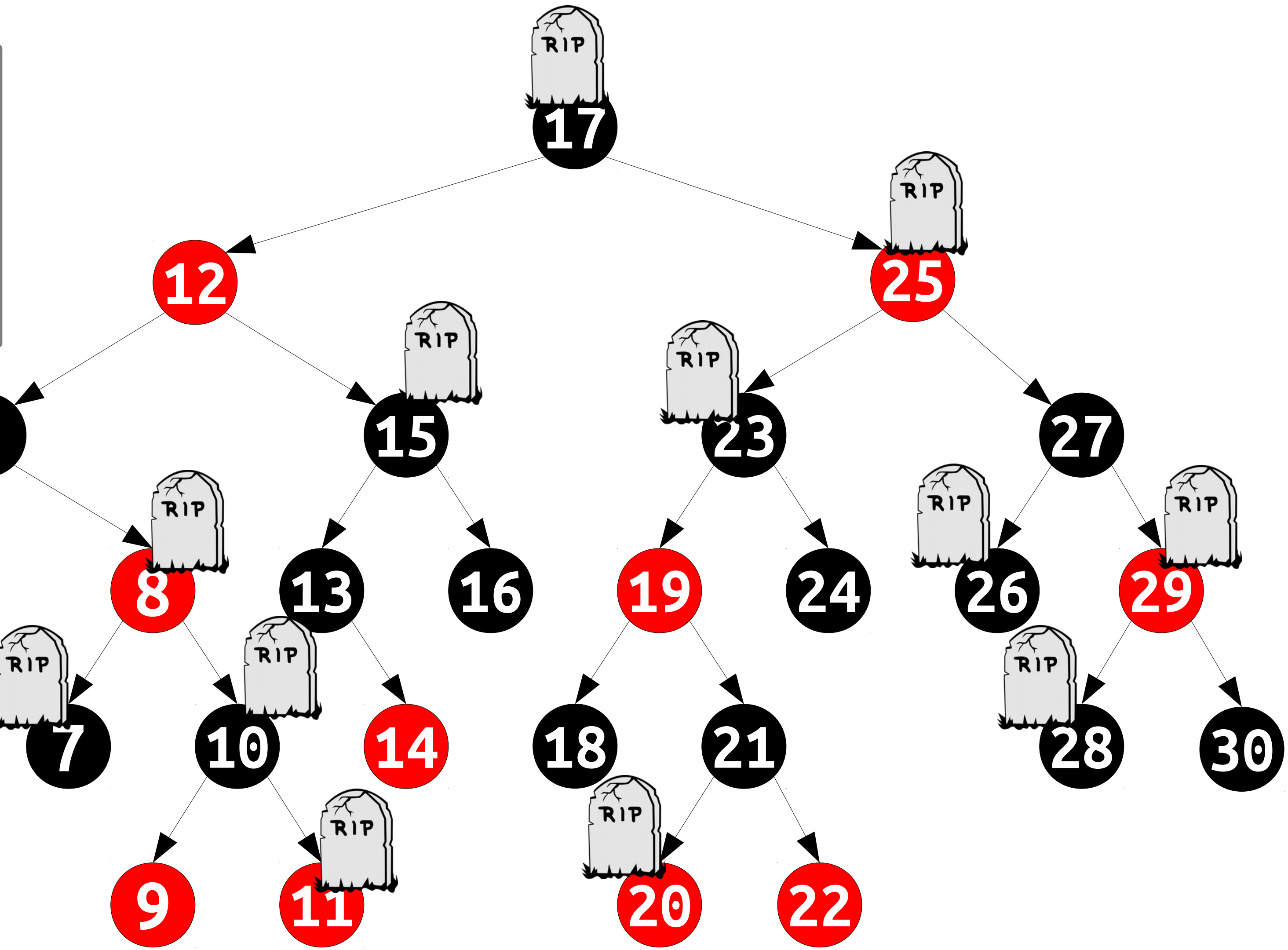
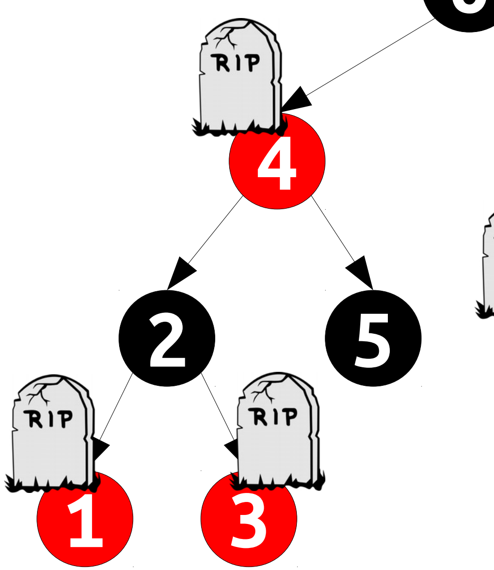
- ***Problem:*** What happens if too many keys die?



Dead Simple Deletions

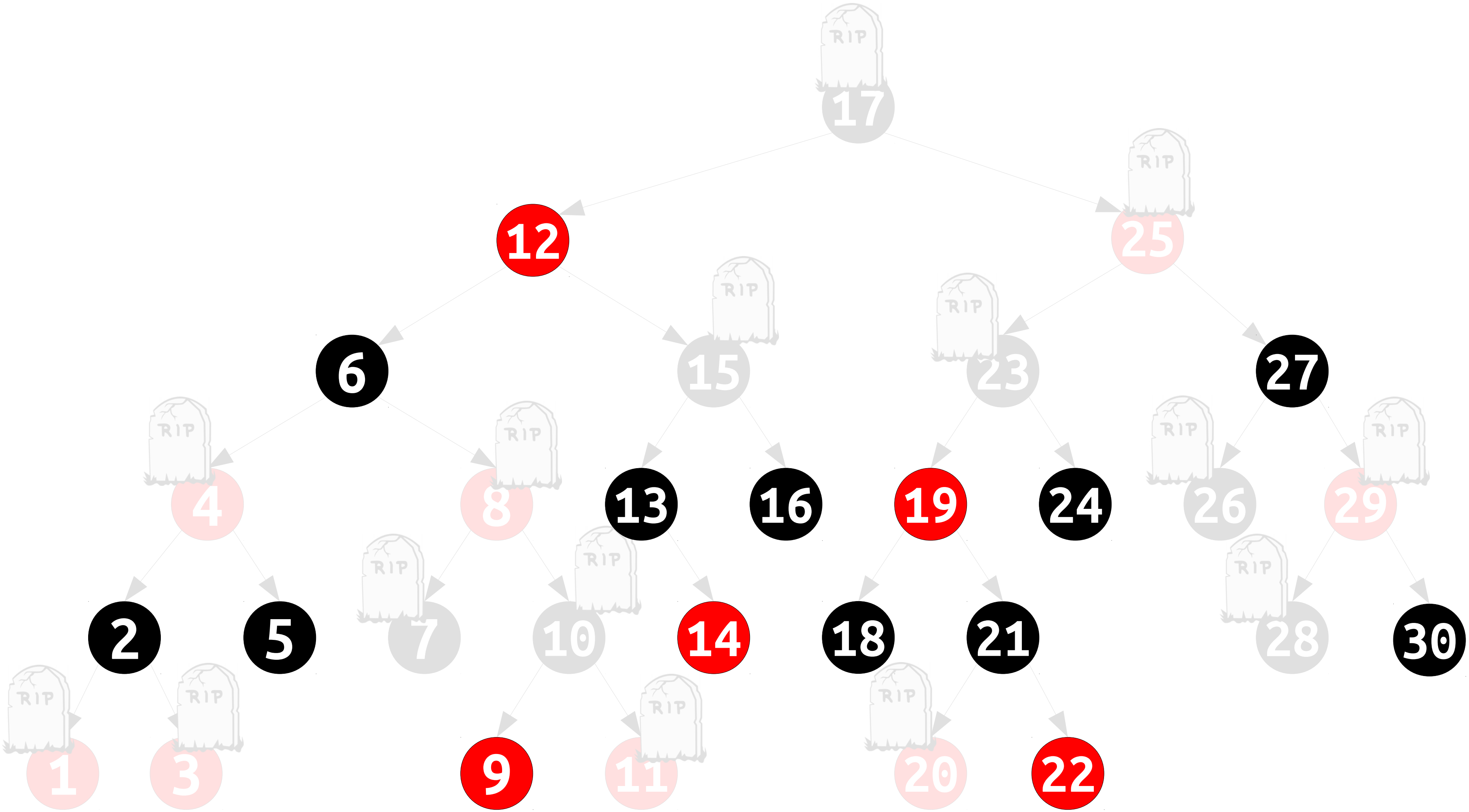
- ***Idea:*** Rebuild the tree when half the keys are dead.



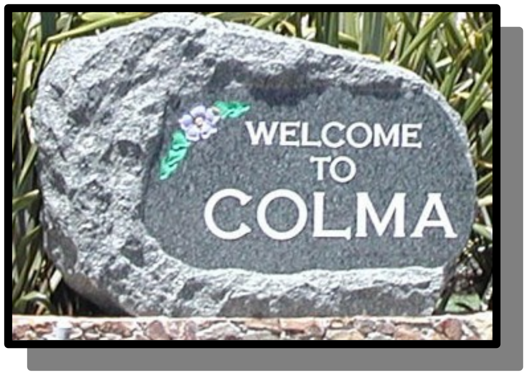
Dead Simple Deletions

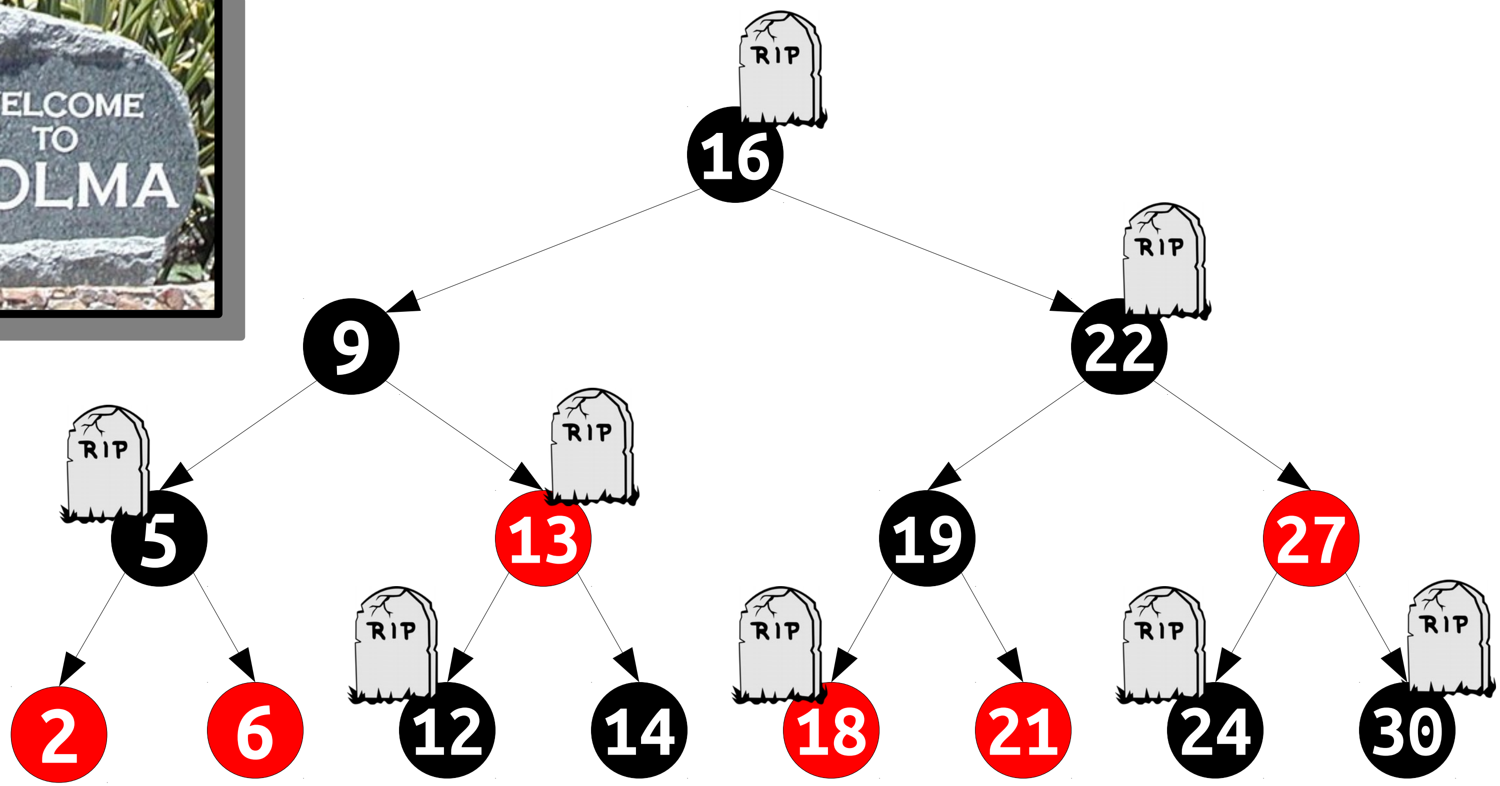
- ***Idea:*** Rebuild the tree when half the keys are dead.



Dead Simple Deletions

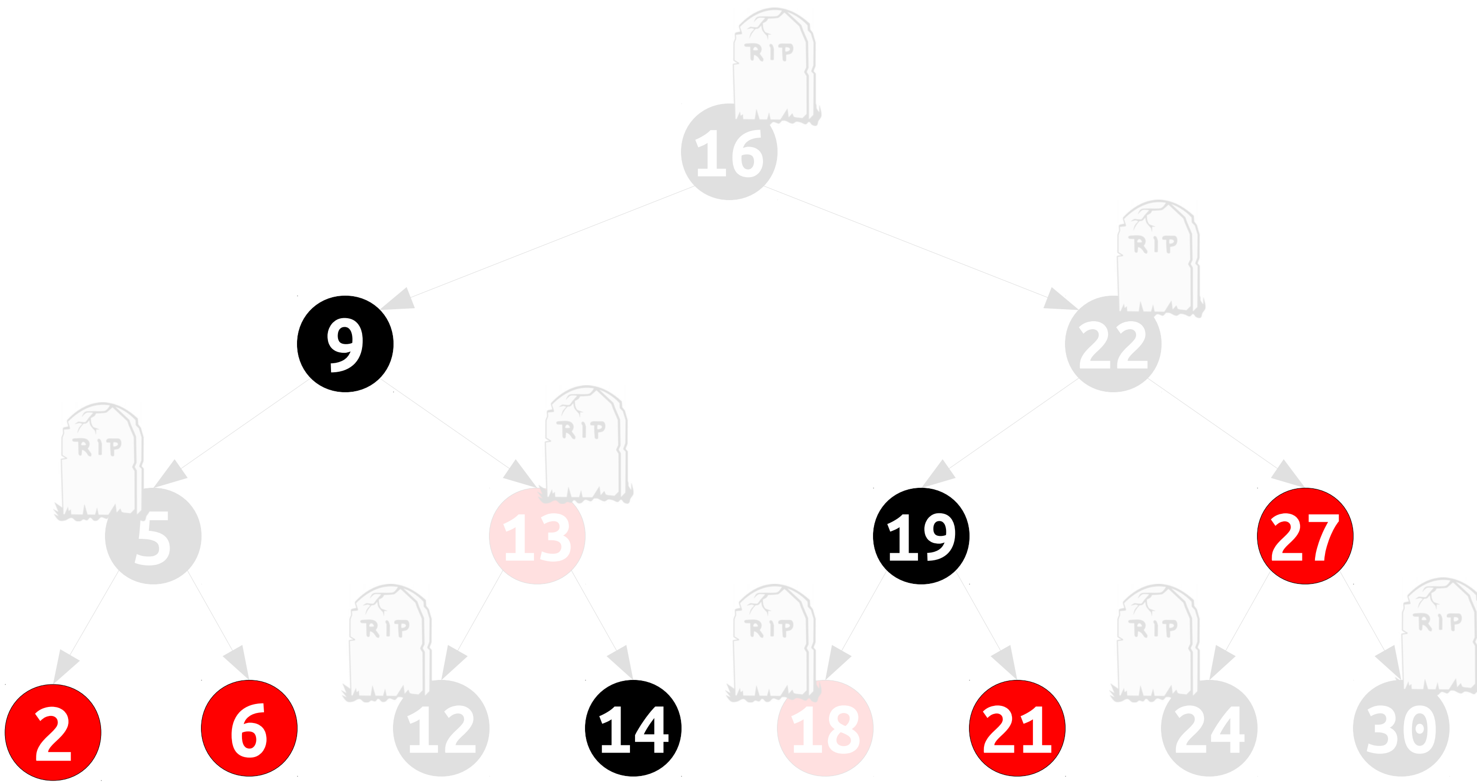
- ***Idea:*** Rebuild the tree when half the keys are dead.





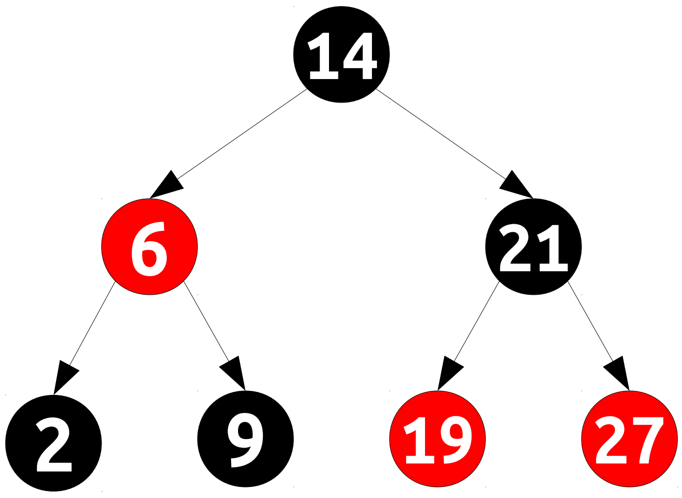
Dead Simple Deletions

- ***Idea:*** Rebuild the tree when half the keys are dead.



Dead Simple Deletions

- ***Idea:*** Rebuild the tree when half the keys are dead.



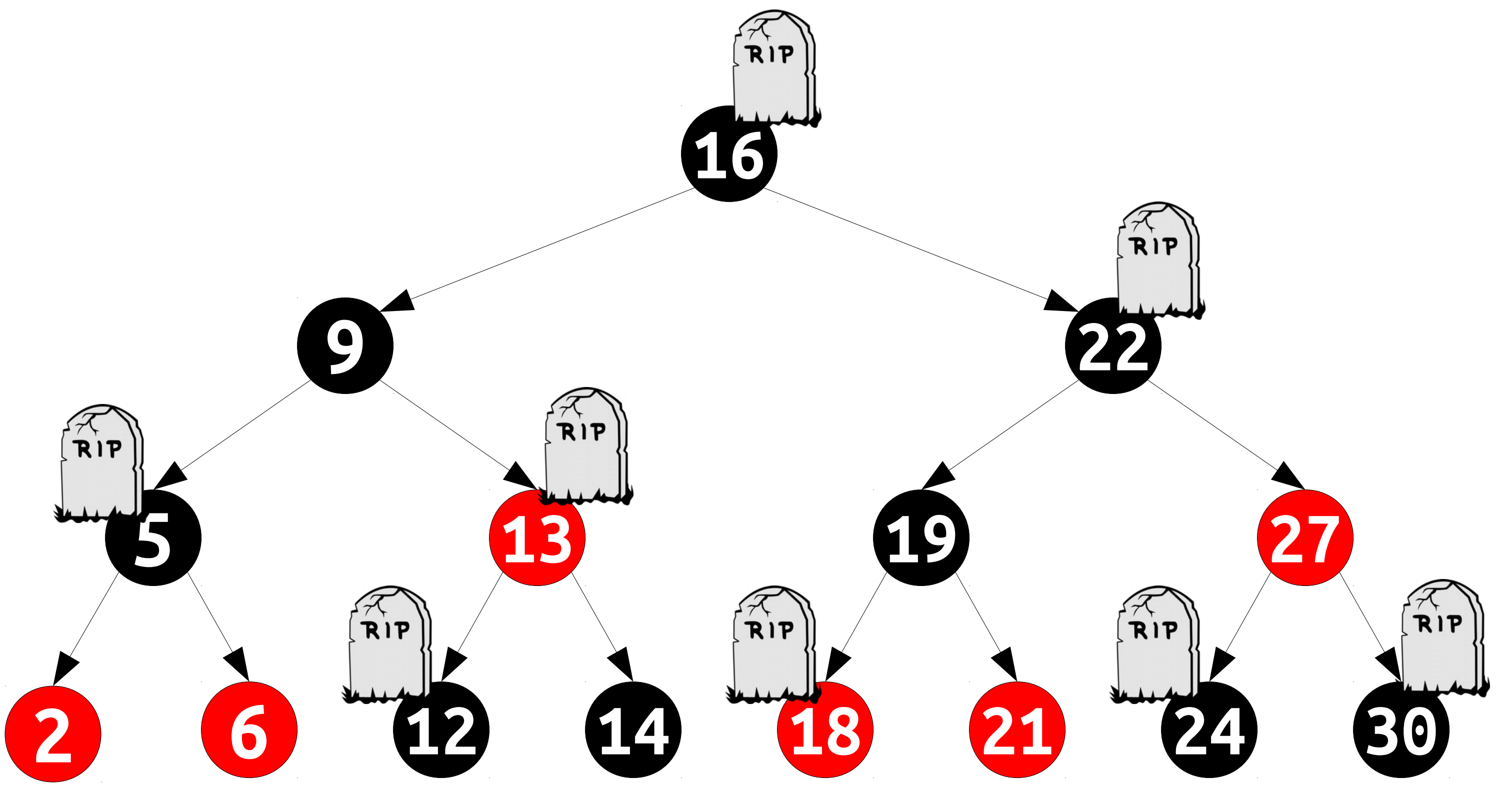
Analyzing Lazy Rebuilding

- What is the cost of an insertion or lookup in a tree with n (living) keys?
  - Total number of nodes: at most $2n$.
  - Cost of the operation: $O(\log 2n) = O(\log n)$.
- What is the cost of a deletion?
  - Most of the time, it's $O(\log n)$.
  - Every now and then, it's $O(n)$.
  - Can we amortize these costs away?

You *can* rebuild the red/black tree in time $O(n)$. How?

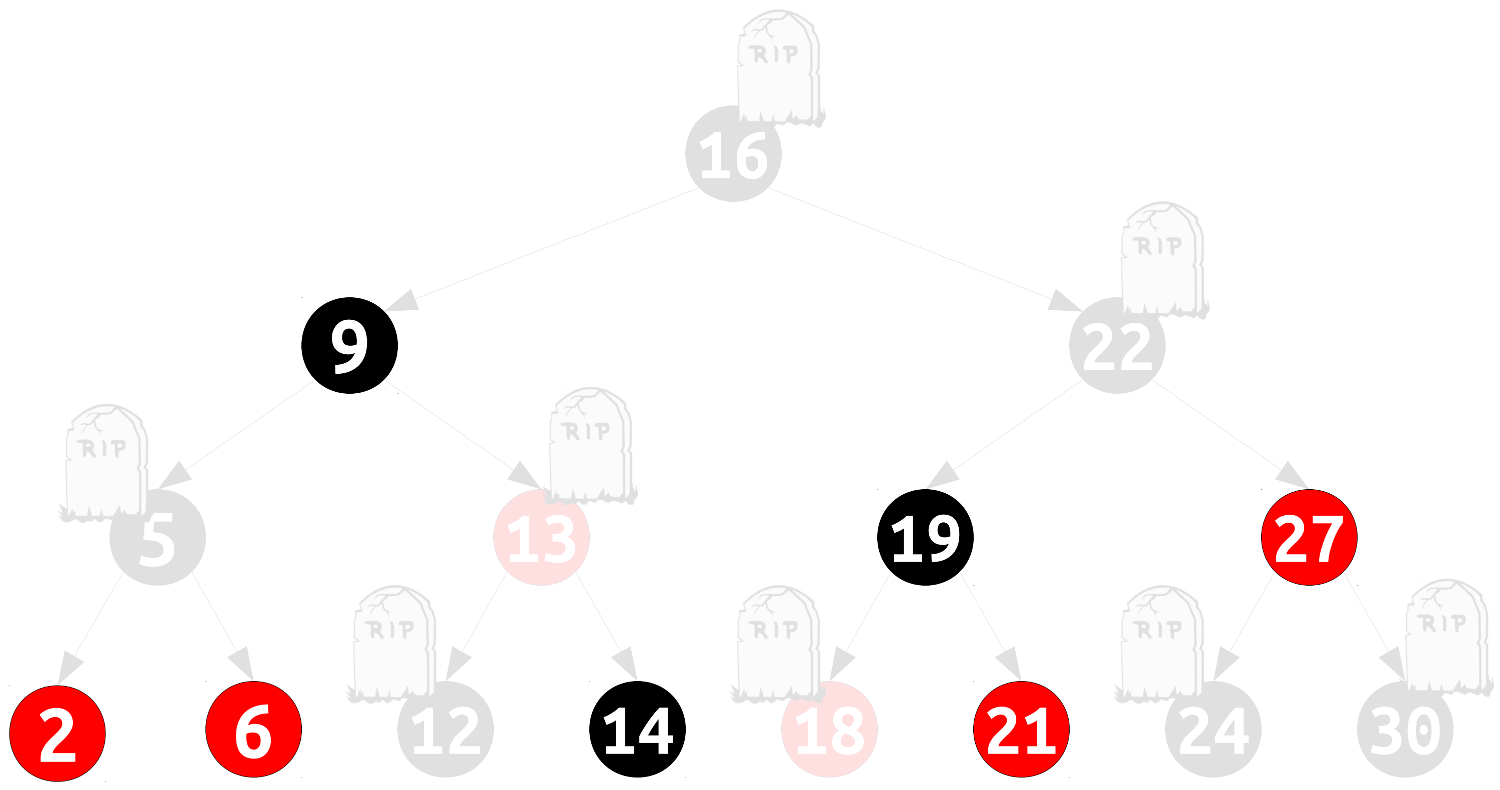
A***mort***ized Analysis

- ***Idea:*** Place a credit on each dead key.
- When we do a rebuild, there are Θ(*n*) credits on the tree, which we can use to pay for the Θ(*n*) rebuild cost.



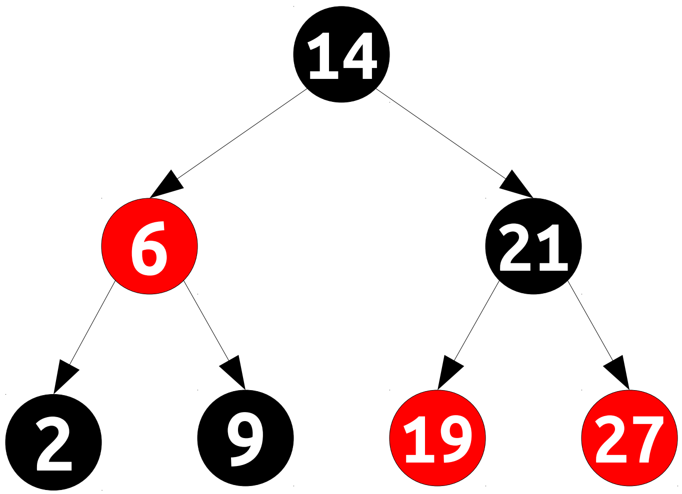
A***mort***ized Analysis

- ***Idea:*** Place a credit on each dead key.
- When we do a rebuild, there are $\Theta(n)$ credits on the tree, which we can use to pay for the $\Theta(n)$ rebuild cost.



A***mort***ized Analysis

- ***Idea:*** Place a credit on each dead key.
- When we do a rebuild, there are $\Theta(n)$ credits on the tree, which we can use to pay for the $\Theta(n)$ rebuild cost.



Lazy Rebuilding

- The amortized cost of a lookup or insertion is O(log n). *(Do you see why?)*
- If a deletion doesn't rebuild, its amortized cost is

$$O(\log n) + O(1) = \mathbf{O(\log n)}.$$

- If a deletion triggers a rebuild:
  - When we start, we have n / 2 credits.
  - When we end, we have 0 credits.
  - Cost of the tree search: O(log n).
  - Cost of the tree rebuild: Θ(n).
  - Amortized cost: $O(\log n) + \Theta(n) - O(1) \cdot \Theta(n) = \mathbf{O(\log n)}$.
- ***Intuition:*** Imbalances build up over time, then get fixed all at once, so we'd expect costs to spread out nicely.

Lazy Deletions

- This approach isn't perfect.
  - Queries for the min or max are slower.
  - Augmentation is a bit harder.
  - Successor / predecessor / range searches slower.
- There are a number of papers about being lazy during BST deletions, many of which have led to new, fast tree data structures.
- Check out WAVL and RAVL trees – these might make for great final project topics!

Next Time

- ***Binomial Heaps***
  - A simple and versatile heap data structure based on binary arithmetic.
- ***Lazy Binomial Heaps***
  - Rejiggering binomial heaps for fun and profit.