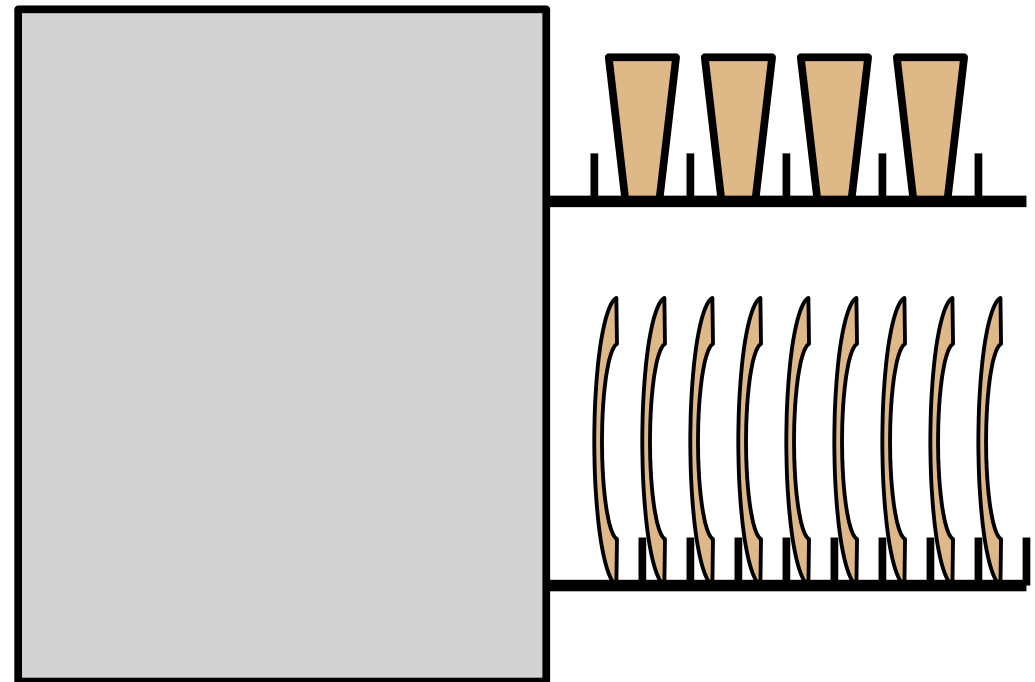
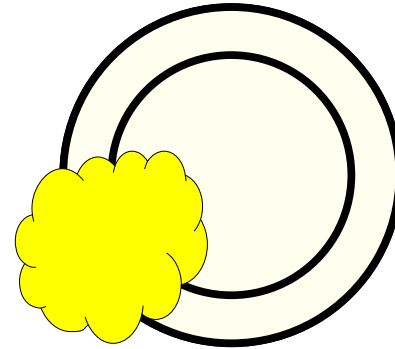


Amortized Analysis

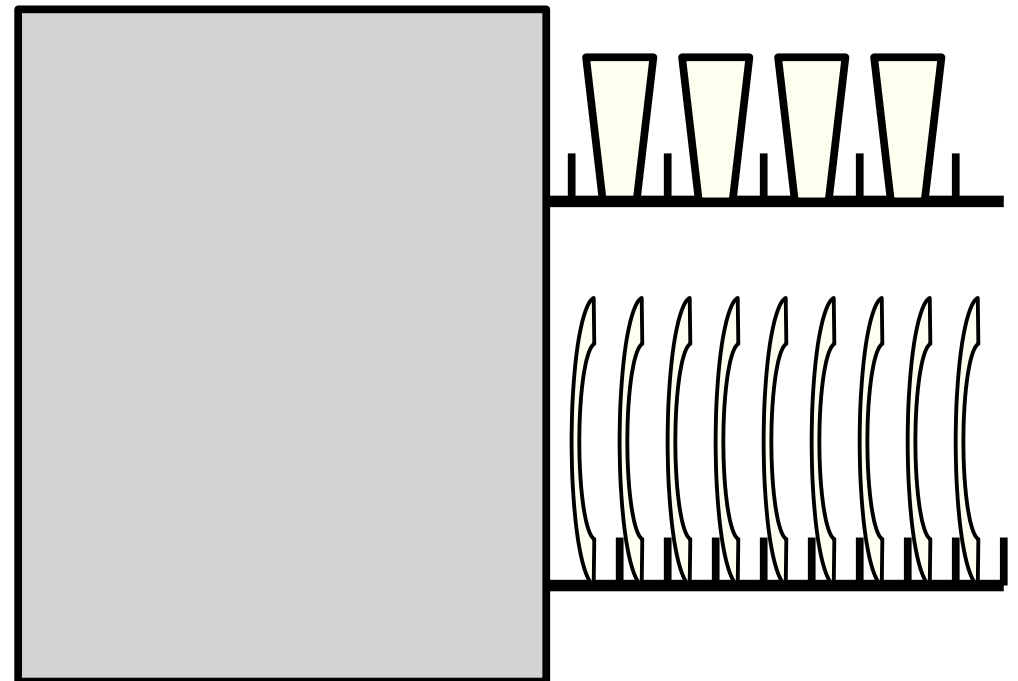
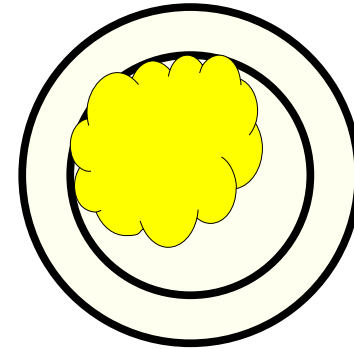
Doing the Dishes

- What do I do with a dirty dish or kitchen utensil?
- **Option 1:** Wash it by hand.
- **Option 2:** Put it in the dishwasher rack, then run the dishwasher if it's full.



Doing the Dishes

- Washing every individual dish and utensil by hand is *way* slower than using the dishwasher, but I always have access to my plates and kitchen utensils.
- Running the dishwasher is faster in aggregate, but means I may have to wait a bit for dishes to be ready.




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

Example: The Two-Stack Queue

The Two-Stack Queue


Out


In

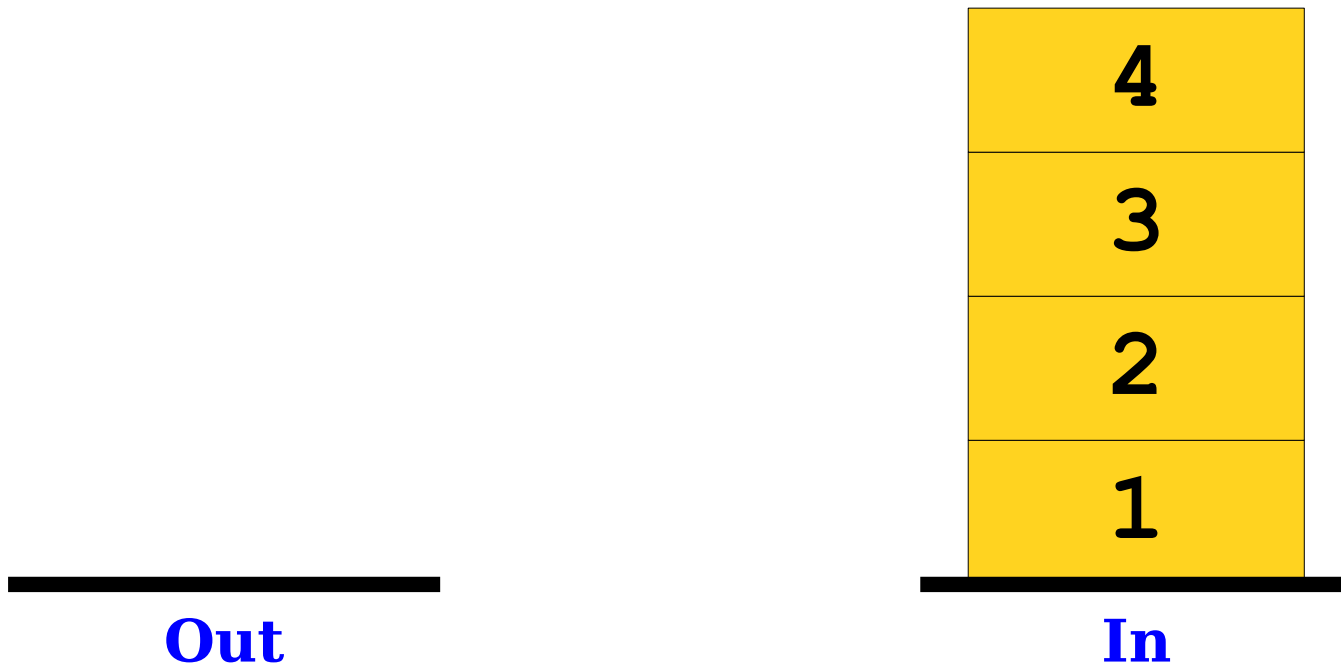
The Two-Stack Queue



The Two-Stack Queue



The Two-Stack Queue



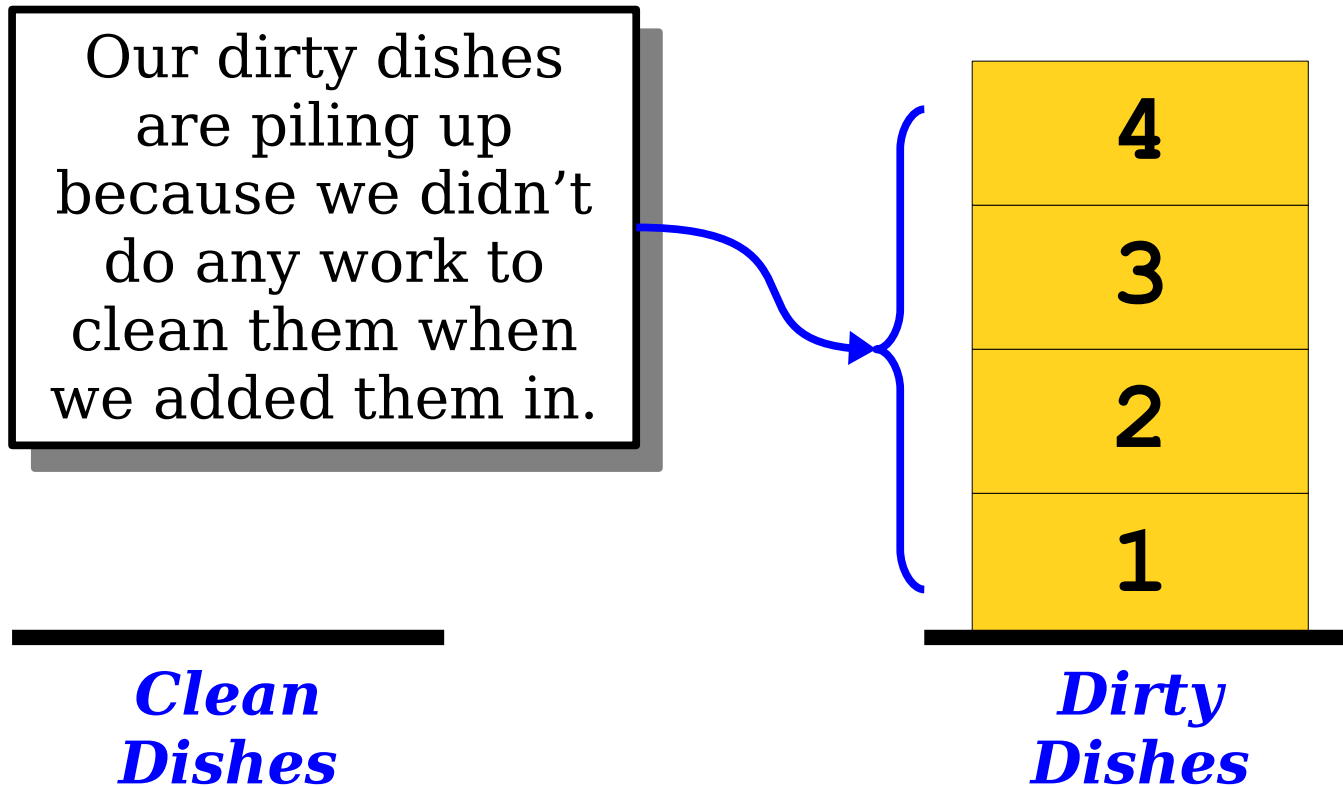
The Two-Stack Queue



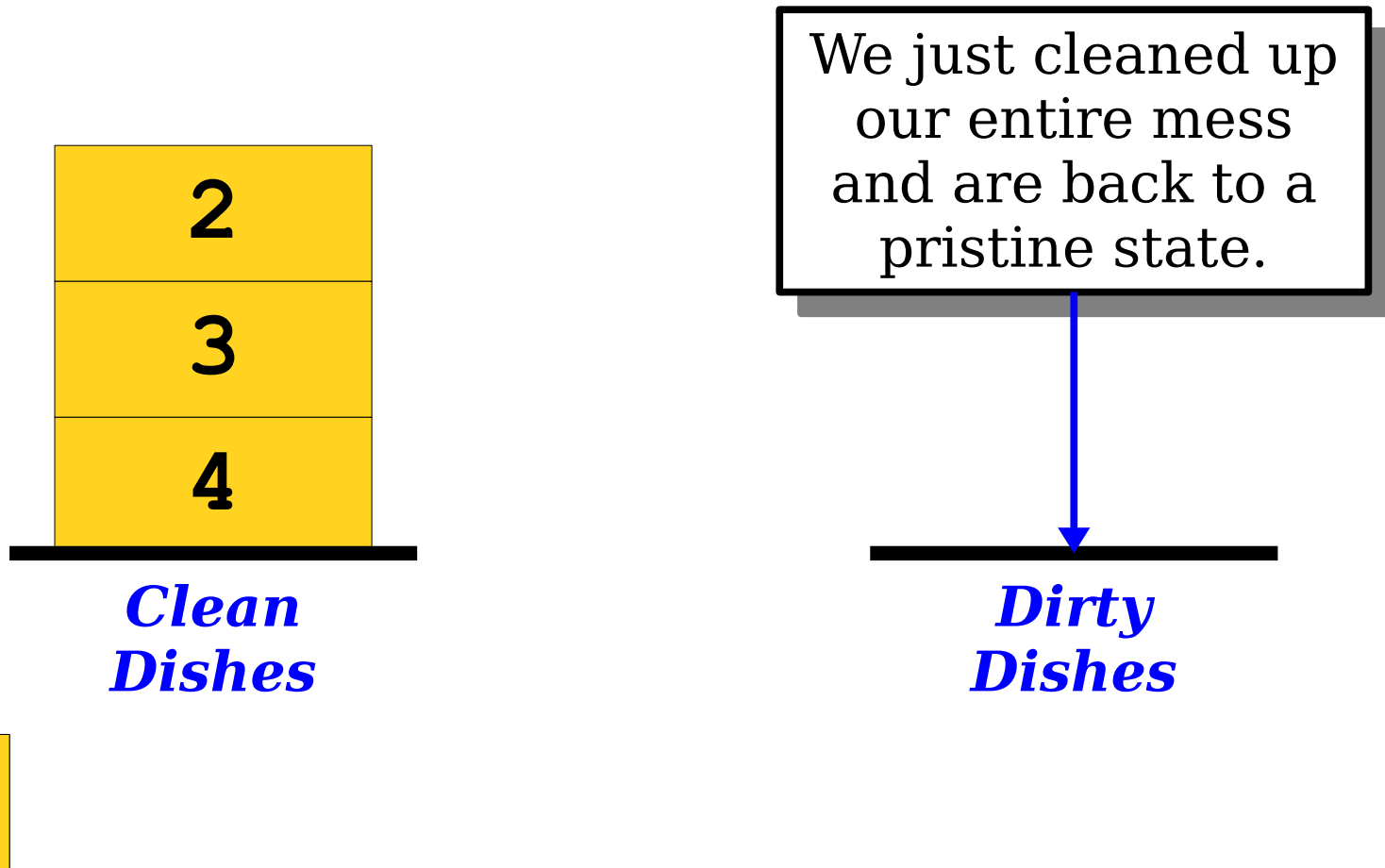
The Two-Stack Queue



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 $\Theta(n)$ if the **Out** stack is empty.



The Two-Stack Queue

- **Intuition:** We only do expensive dequeues after a long run of cheap enqueues.
- Think “dishwasher:” we very slowly introduce a lot of dirty dishes to get cleaned up all at once.
- Provided we clean up all the dirty dishes at once, and provided that dirty dishes accumulate slowly, this is a fast strategy!



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

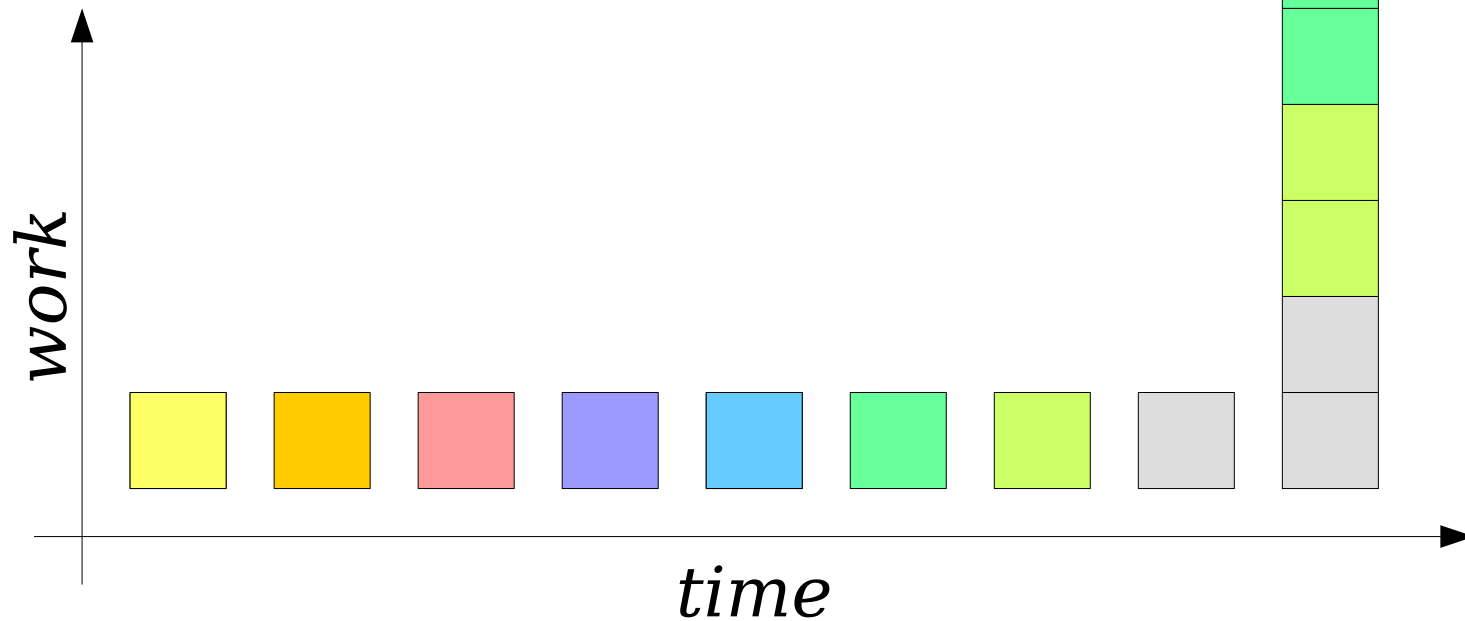
- **Initial idea:** Summarize our result using an average-case analysis.
 - If we do m total operations, the total work done is $O(m)$.
 - Average amount of work per operation: $O(1)$.
- Based on this argument, we can claim that the average cost of an enqueue or dequeue is $O(1)$.
- **Claim:** While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

Issue: When we say the average cost of an operation is $O(1)$, what are we averaging over?

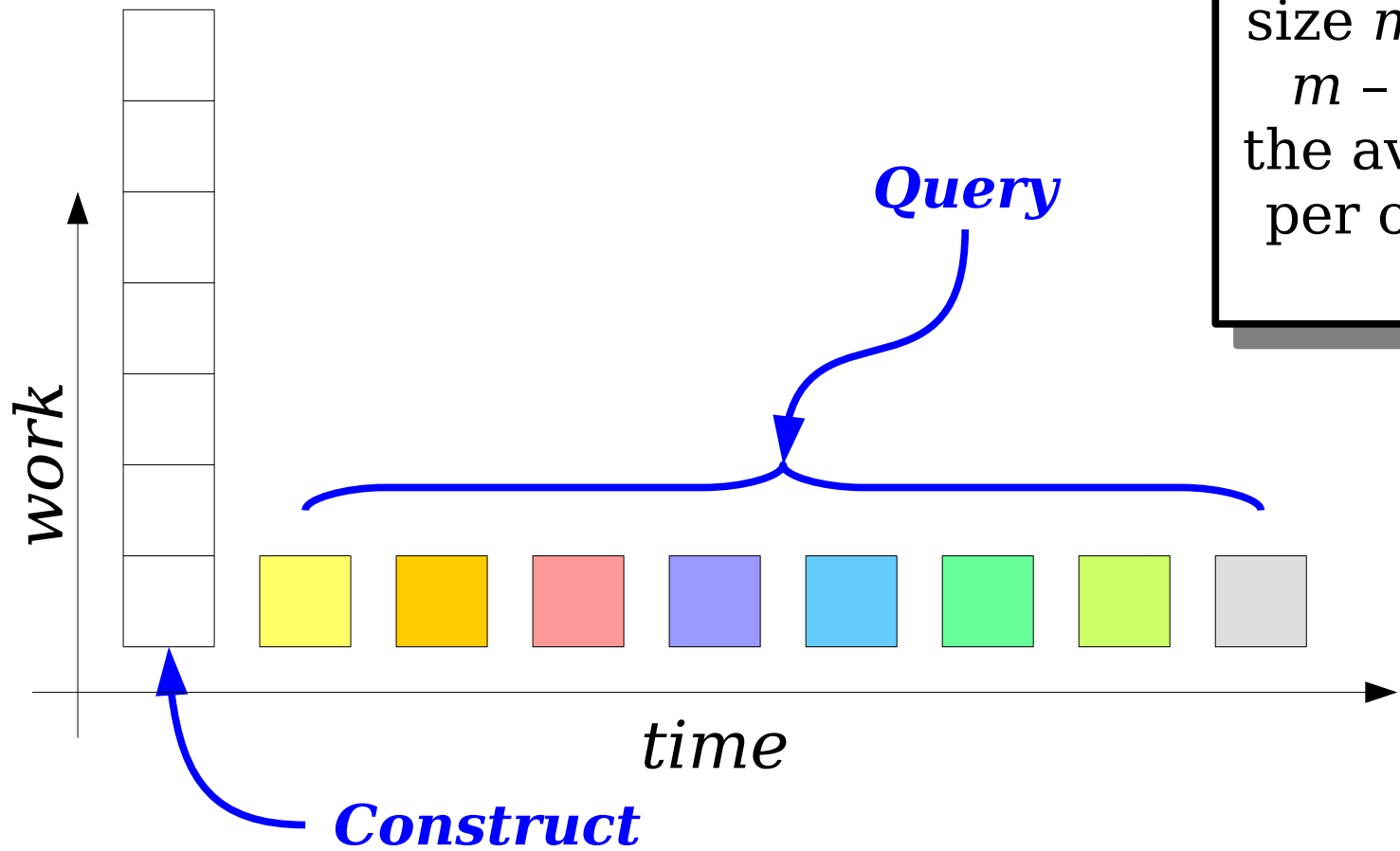
Total work done: $\Theta(m)$

Total operations: $\Theta(m)$

Average work per element: $O(1)$.

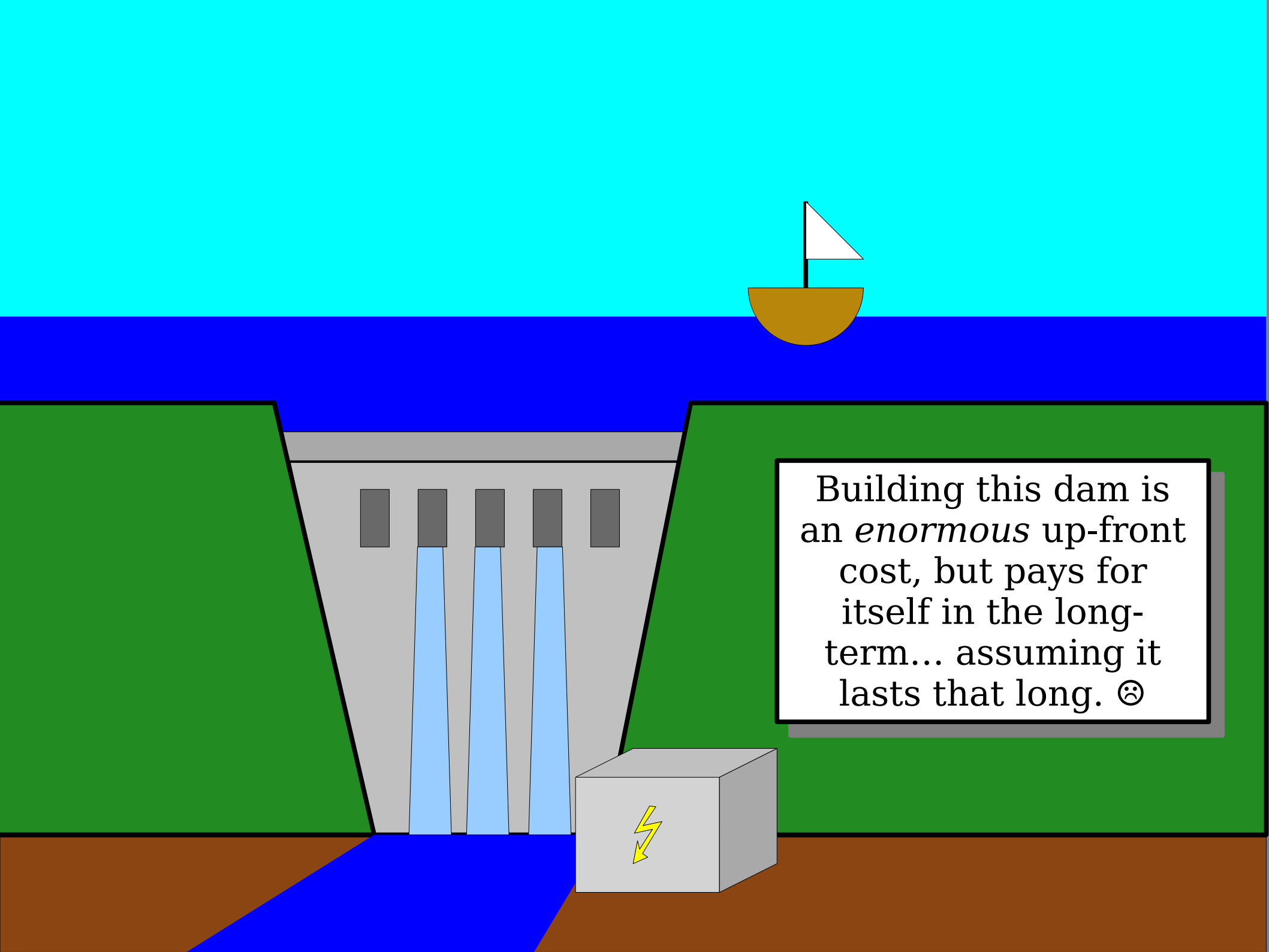


If the array has size m and we do $m - 1$ queries, the average work per operation is $O(1)$.



***Fischer-Heun
RMQ***

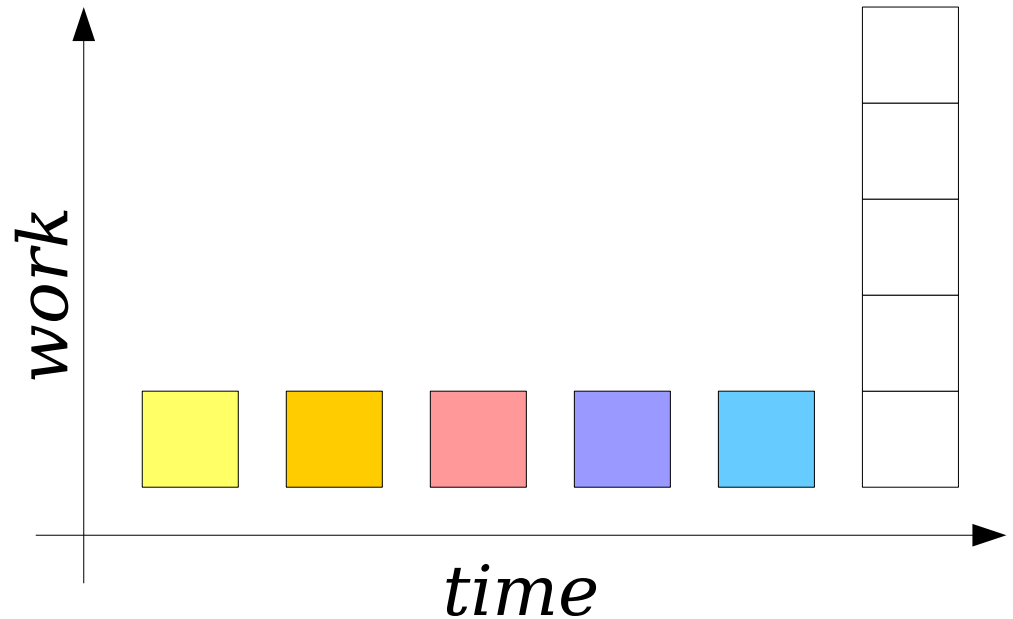
31	41	59	26	53	58	97	93
----	----	----	----	----	----	----	----



Building this dam is an *enormous* up-front cost, but pays for itself in the long-term... assuming it lasts that long. ☹️

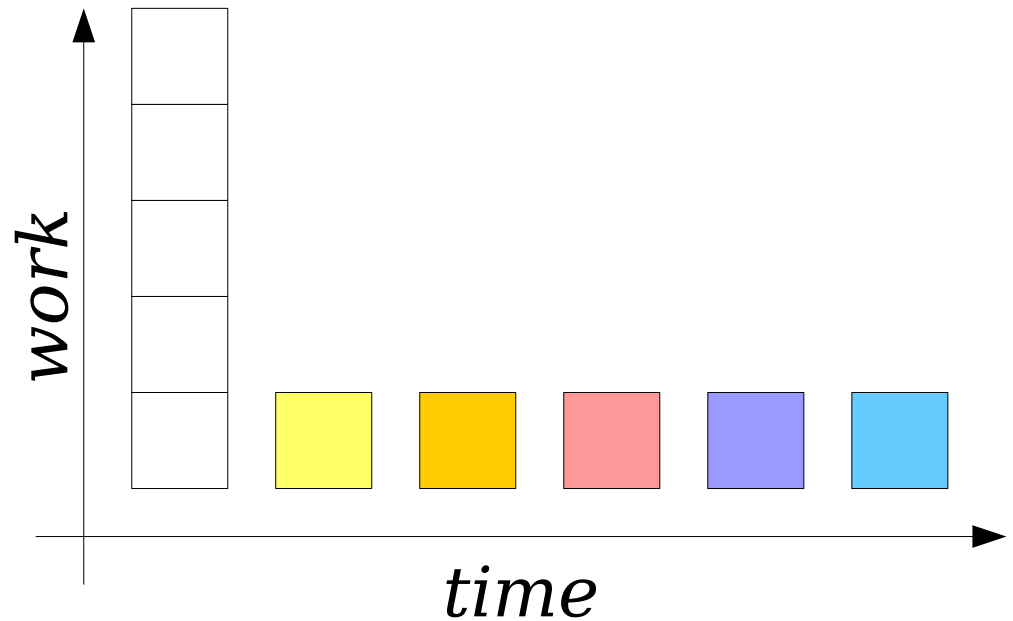
Dishwasher model: Lots of cheap operations that need to be made up for by an expensive one later.

The average work done at each point in time is low.



Dam model: Early, expensive operation that pays off in the long term.

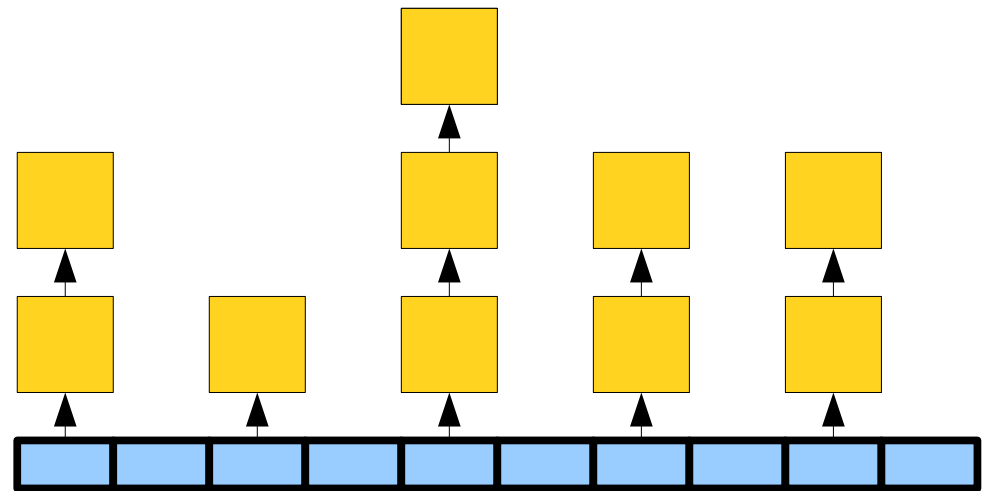
The average work done at each point in time is high until lots of operations are performed.



Nuance 1: The average cost of the operations done on a two-stack queue is *always* low, regardless of when we stop performing operations.

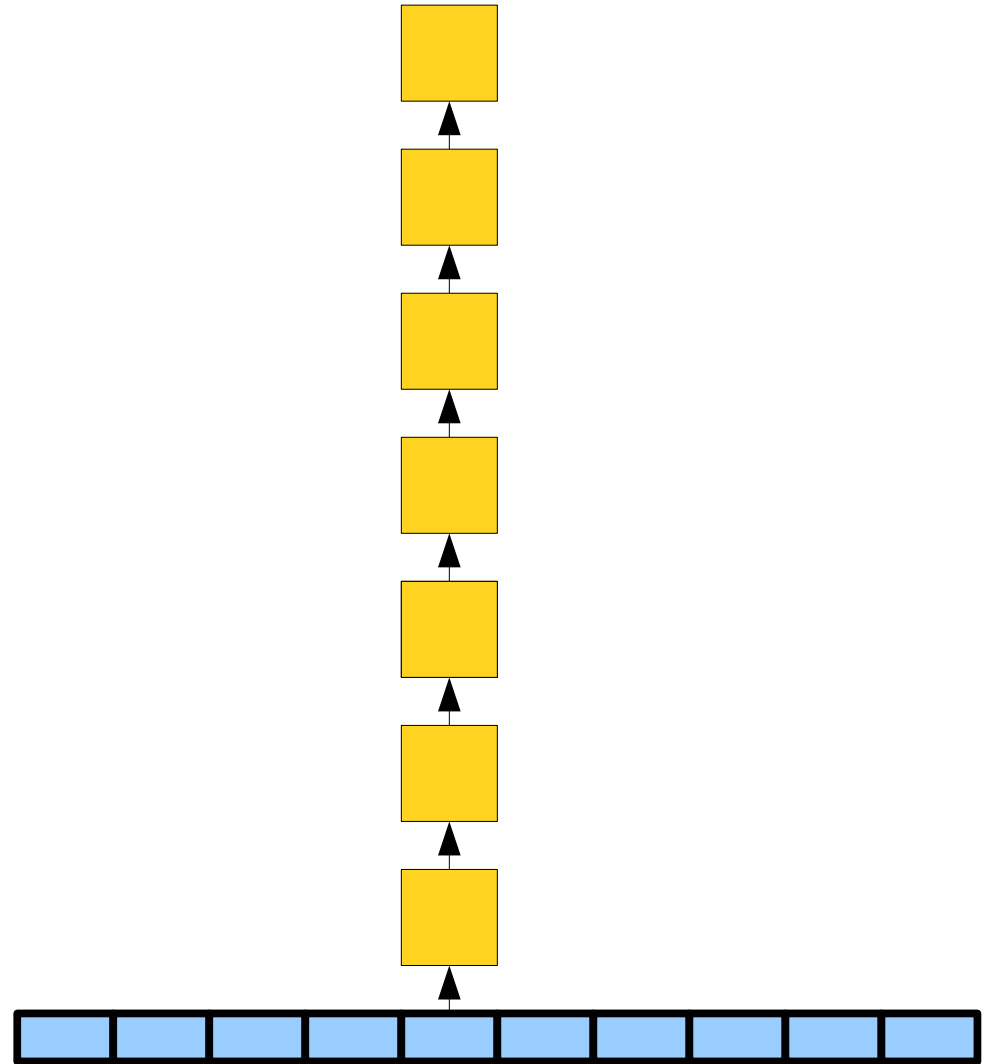
Averaging Over What?

- Compare our two-stack queue to a chained hash table.
- Assuming there are at least as many buckets as elements, the expected cost of an insertion or lookup is $O(1)$.
- However, it isn't *guaranteed* that the cost of a lookup or insertion is $O(1)$.

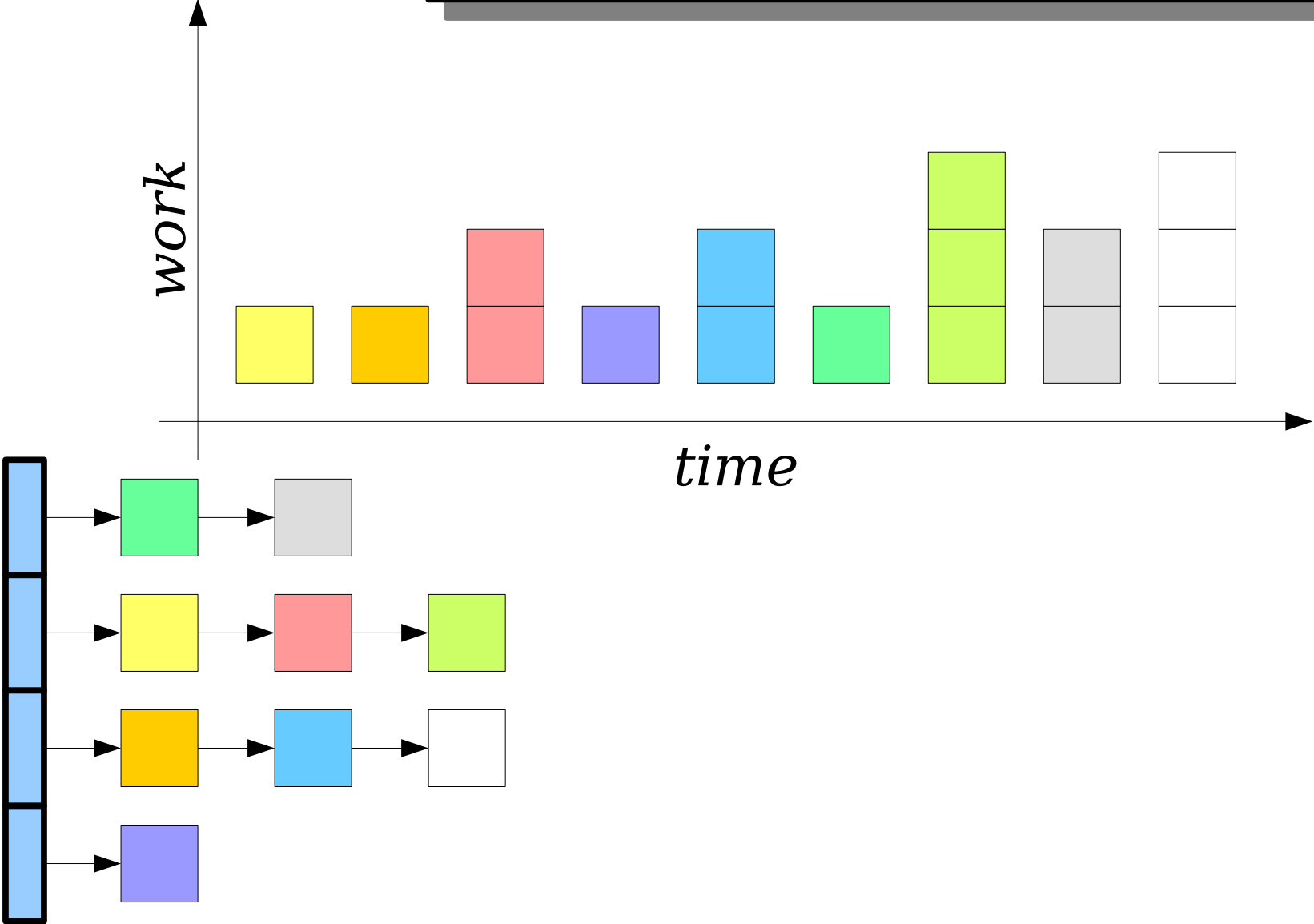


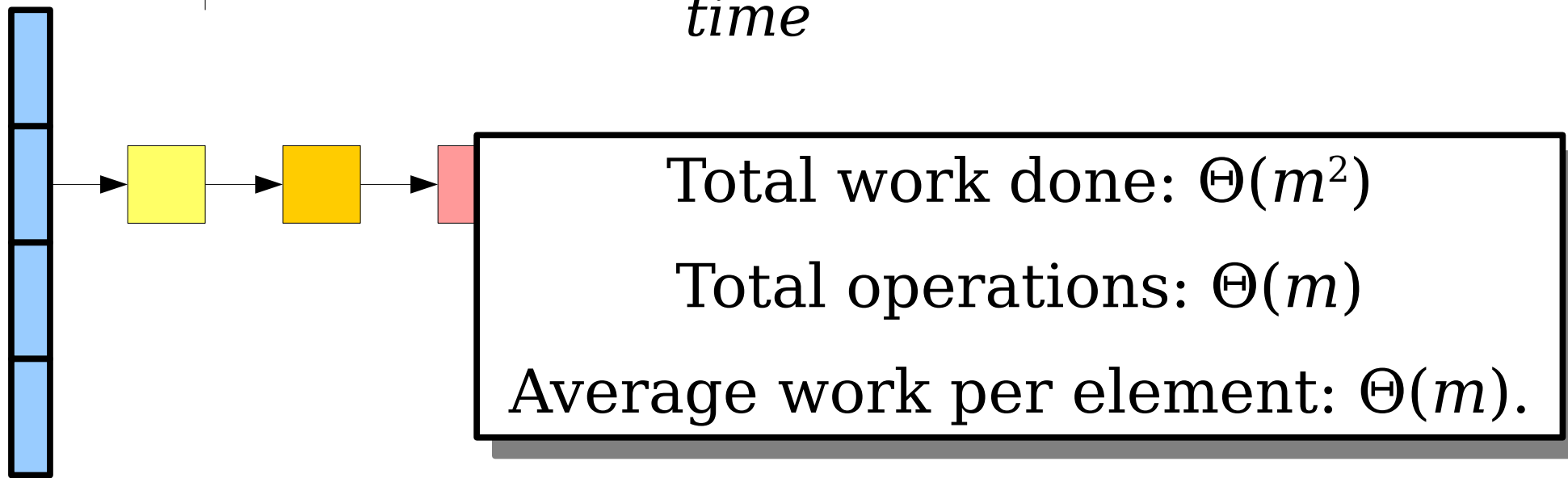
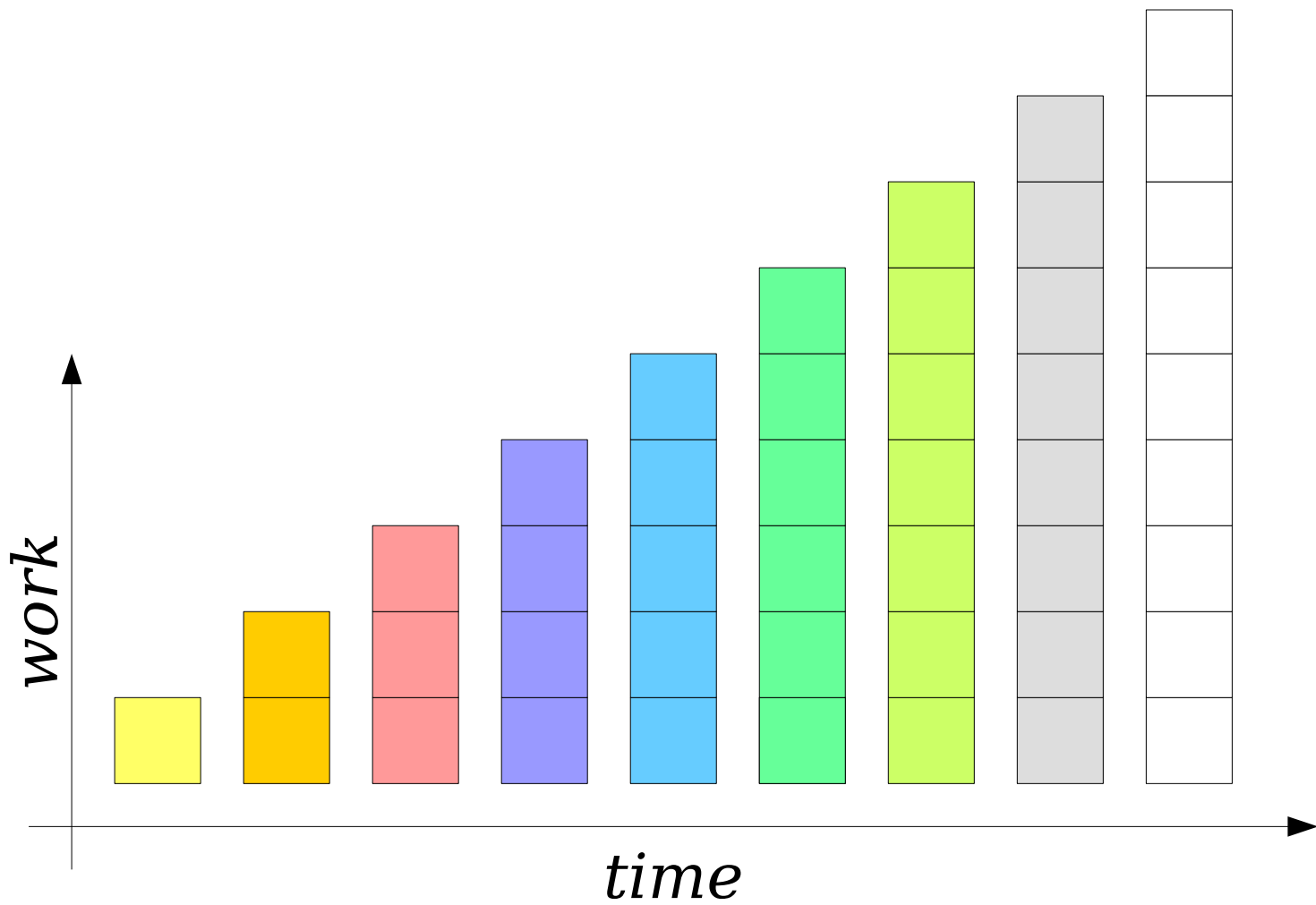
Averaging Over What?

- Compare our two-stack queue to a chained hash table.
- Assuming there are at least as many buckets as elements, the expected cost of an insertion or lookup is $O(1)$.
- However, it isn't *guaranteed* that the cost of a lookup or insertion is $O(1)$.



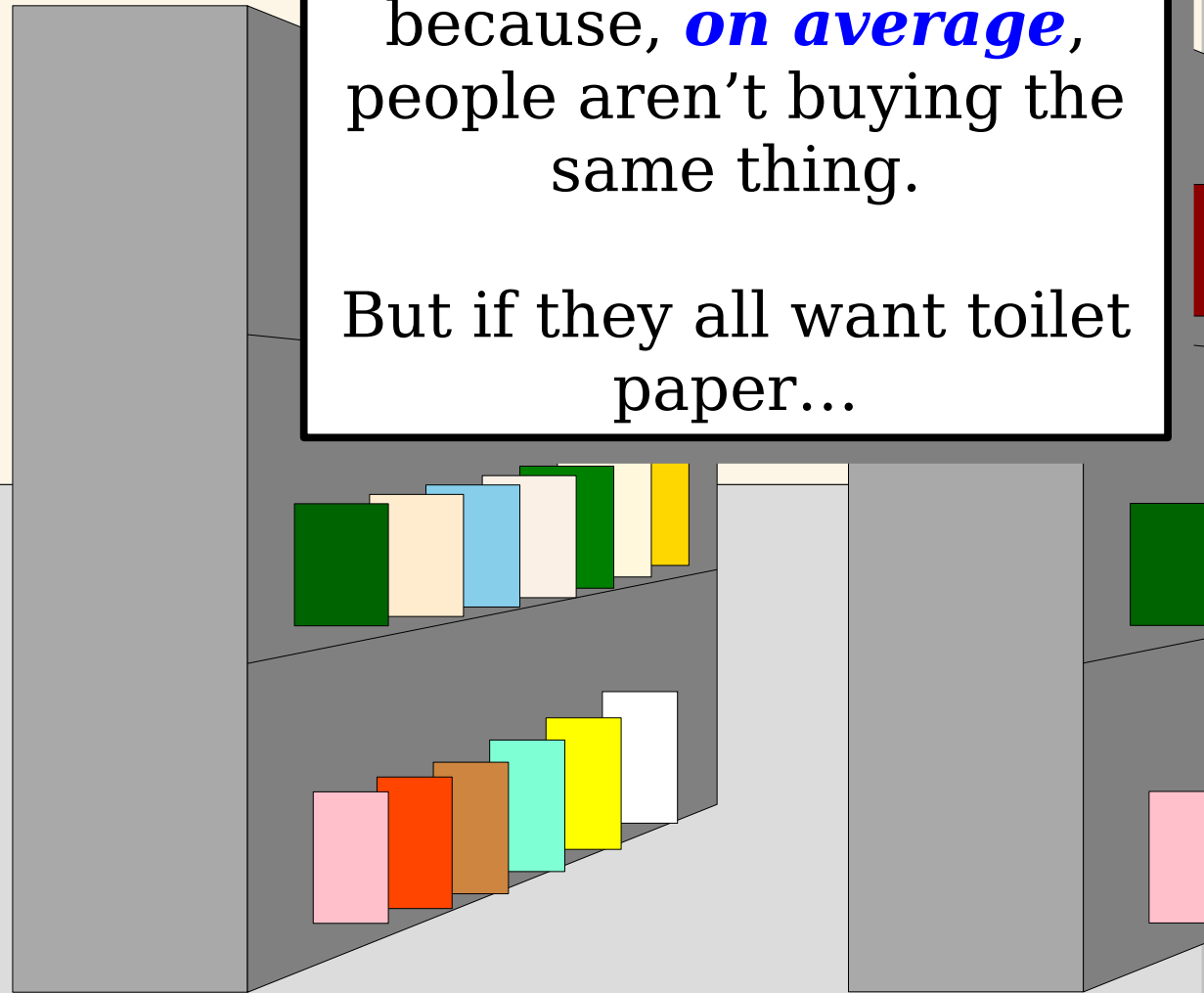
Total work done: 16
Total operations: 9
Average work per element: ≈ 1.8





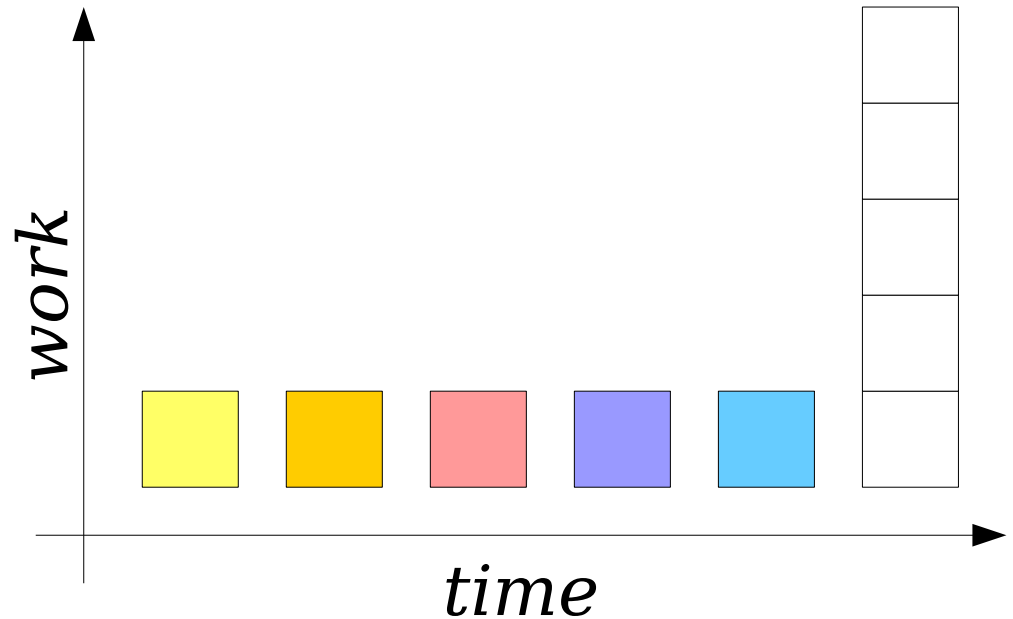
Grocery stores don't need to stock up huge quantities of every item because, *on average*, people aren't buying the same thing.

But if they all want toilet paper...



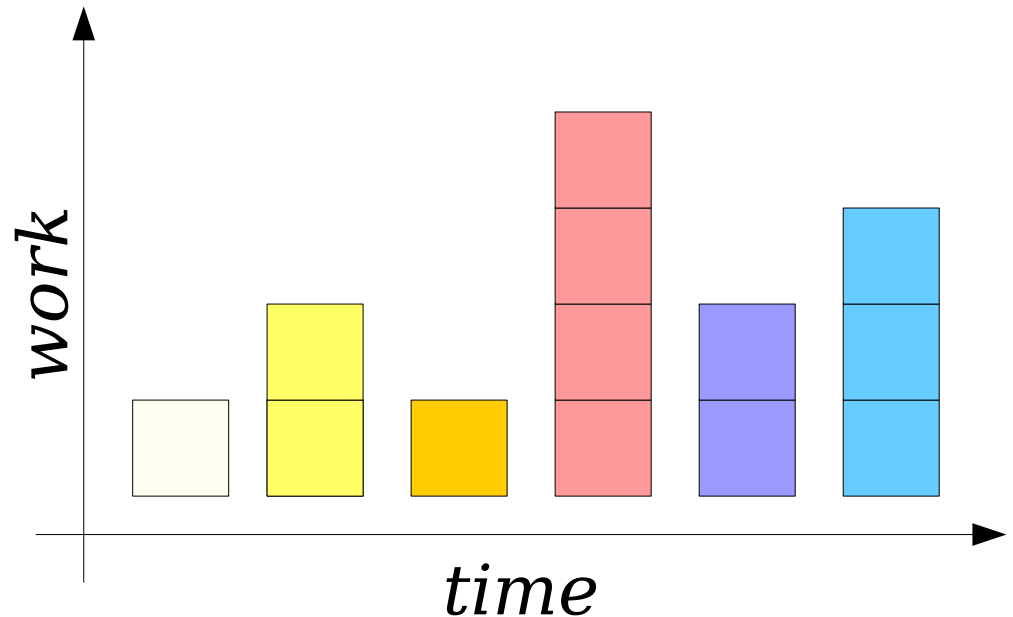
Dishwasher model: Lots of cheap operations that need to be made up for by an expensive one later.

The average work done at each point in time is low.



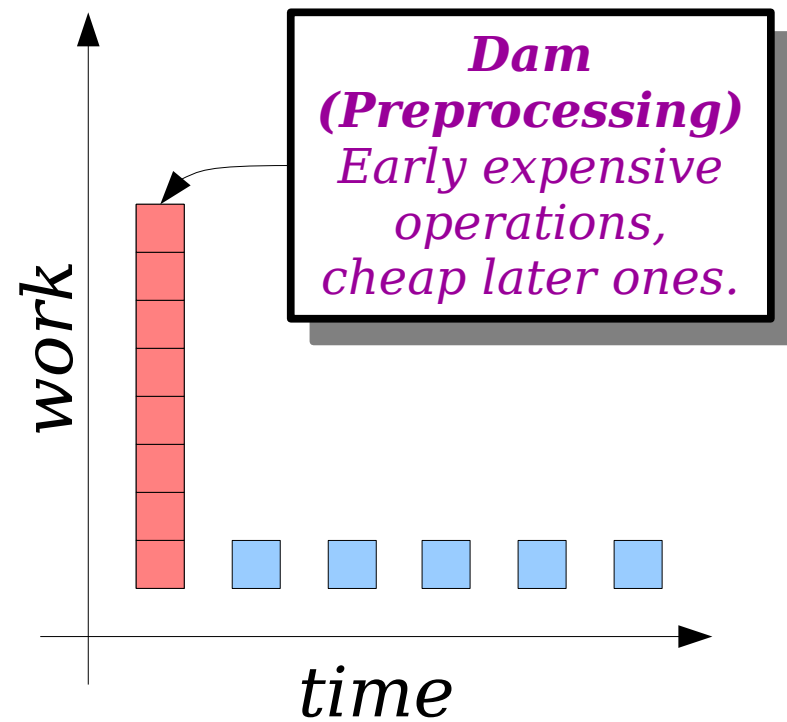
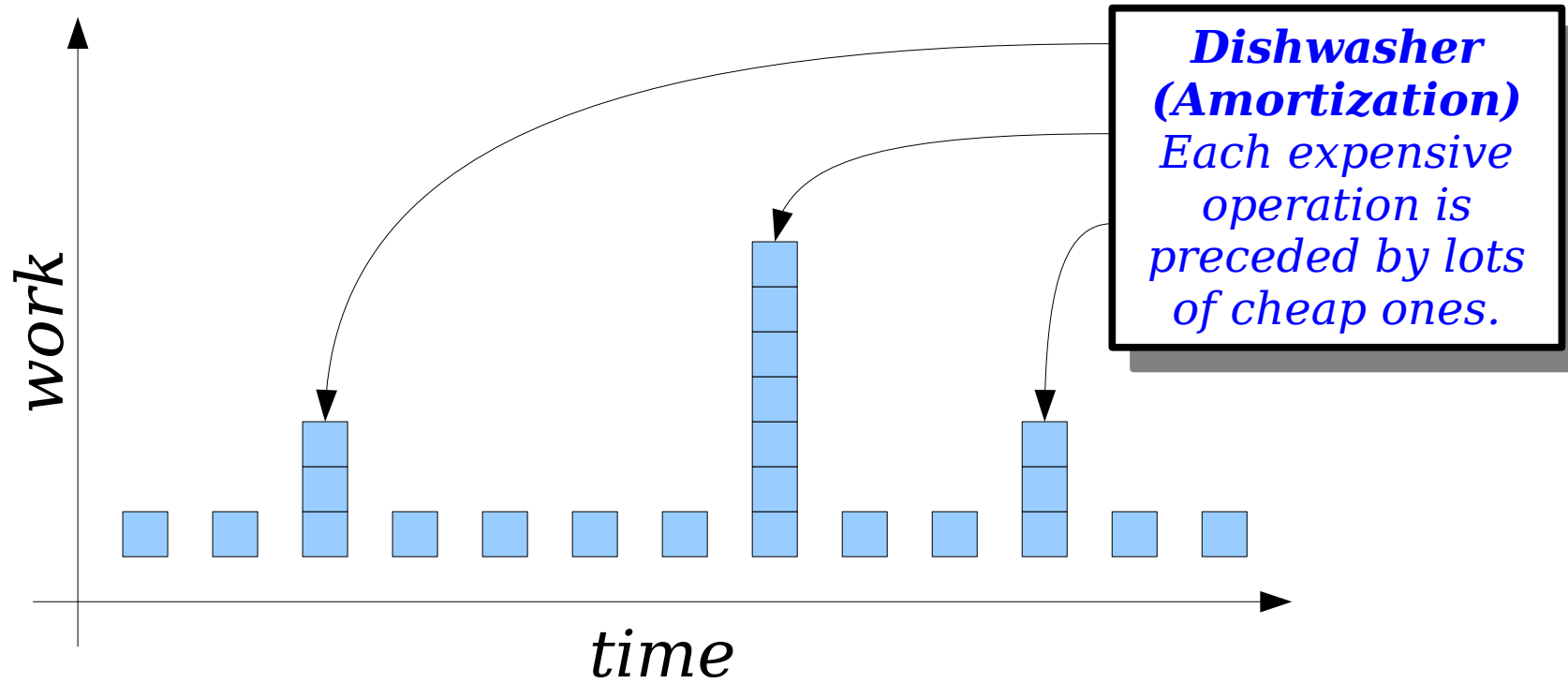
Grocery store model: It's unlikely that there will be any large operations because randomization saves the day.

Except that, every now and then, we run into trouble...

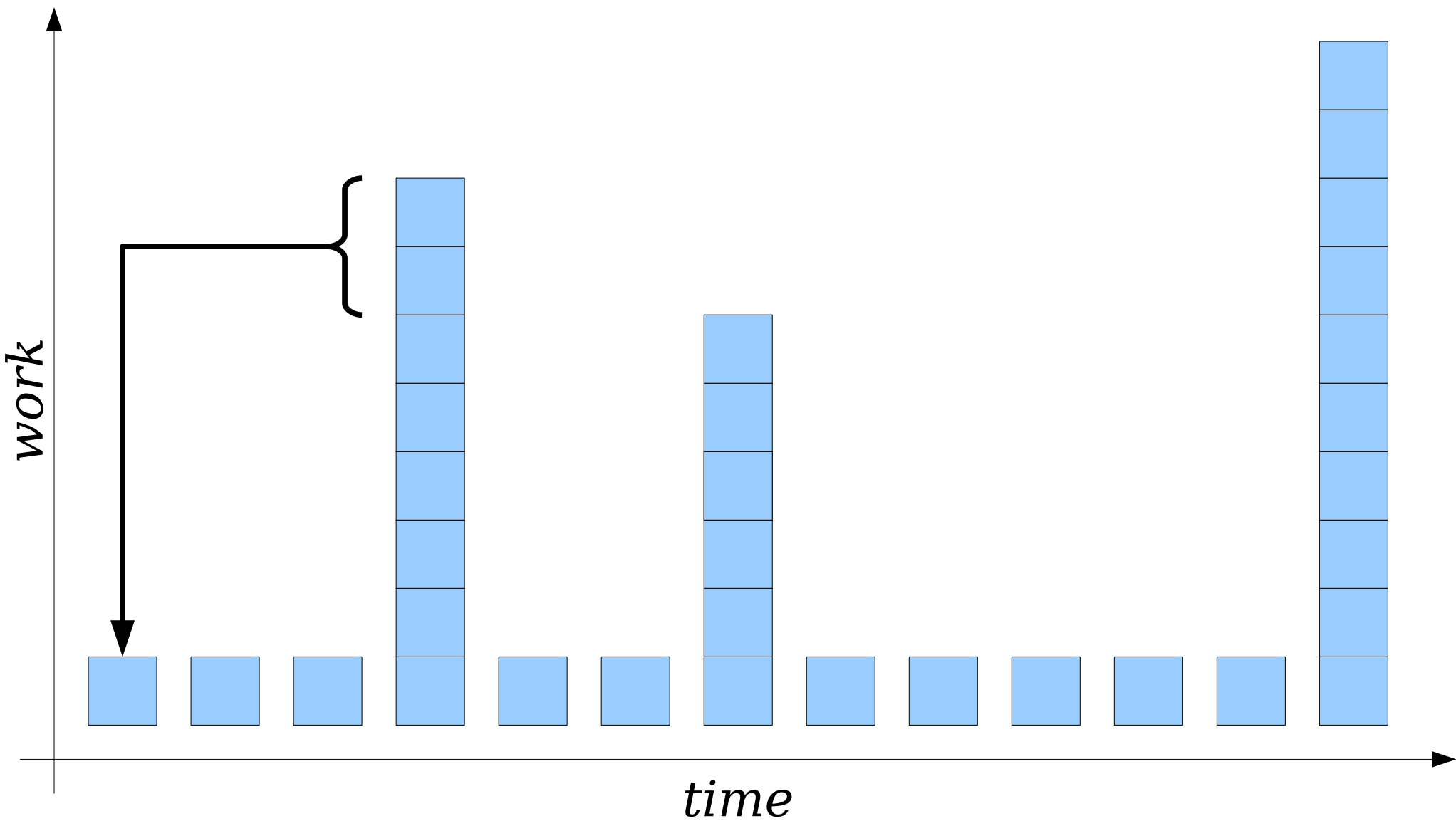


Nuance 2: The “average” mentioned in a two-stack queue is not based on any random variables. There is no chance that any sequence of operations on a two-stack queue takes “too long.”

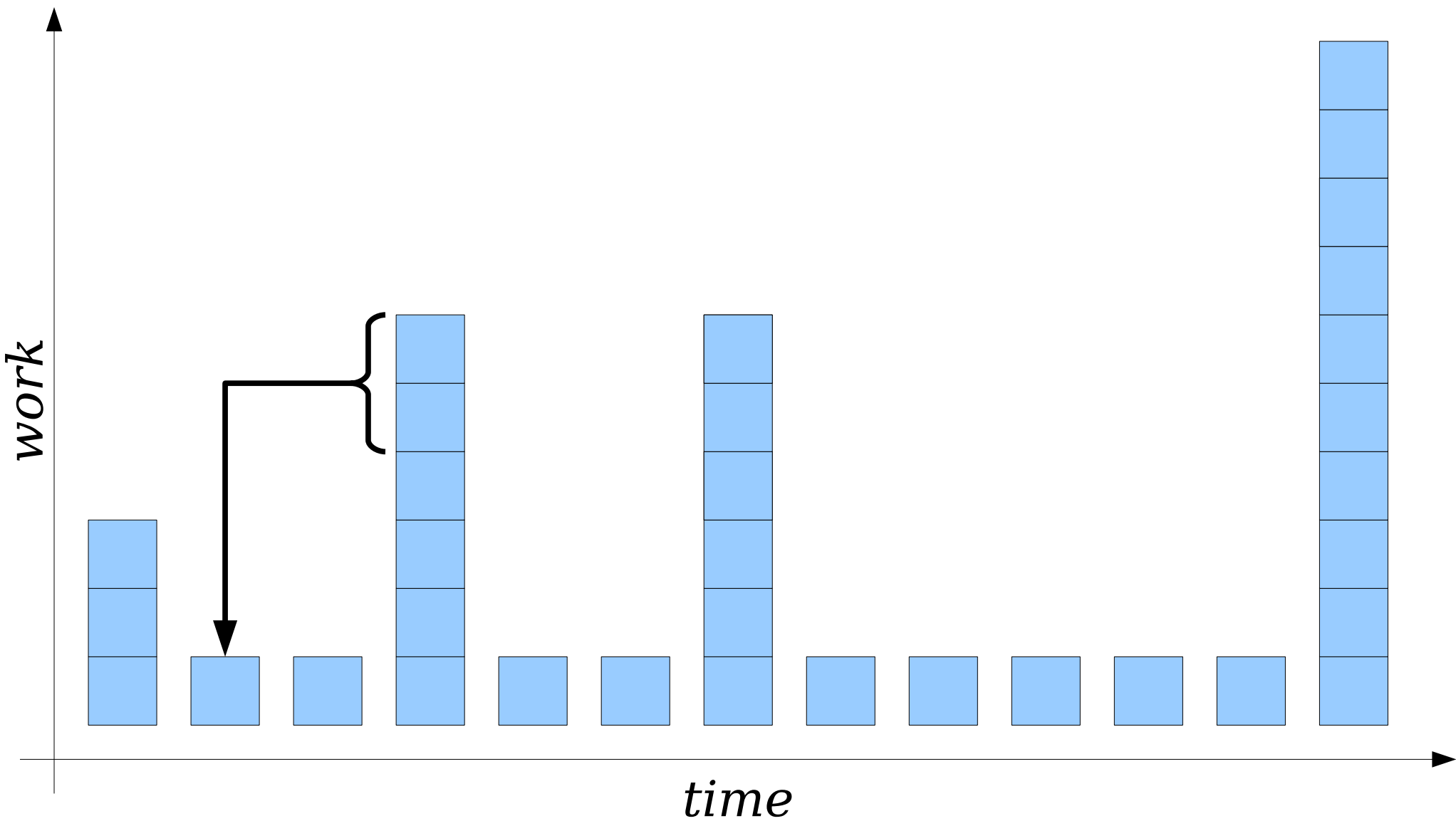
To Summarize



What Amortization Means

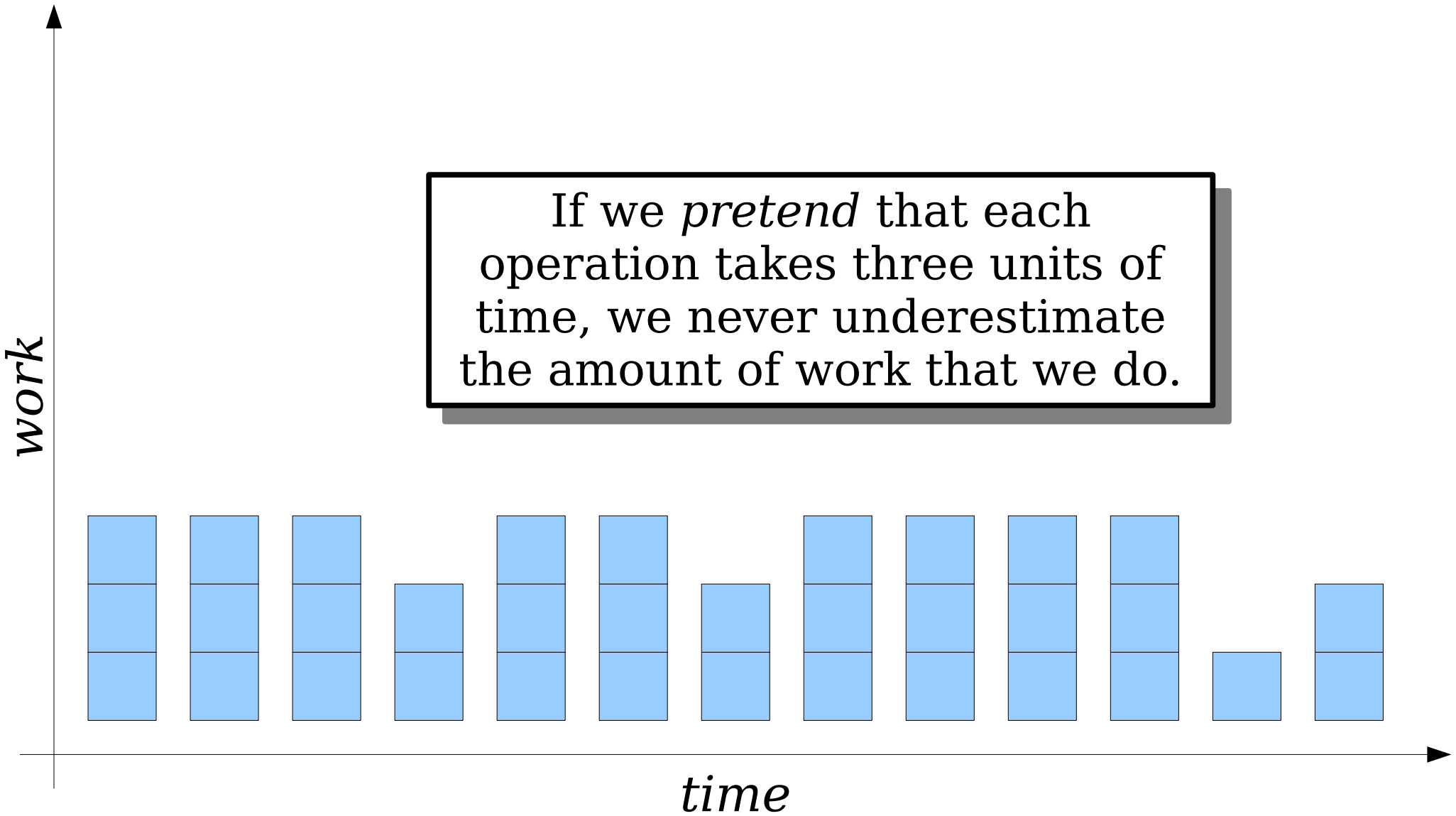


Key Idea: Backcharge expensive operations to cheaper ones.



Key Idea: Backcharge expensive operations to cheaper ones.

If we *pretend* that each operation takes three units of time, we never underestimate the amount of work that we do.



Key Idea: Backcharge expensive operations to cheaper ones.

Amortized Analysis

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

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

No matter when we stop performing operations...

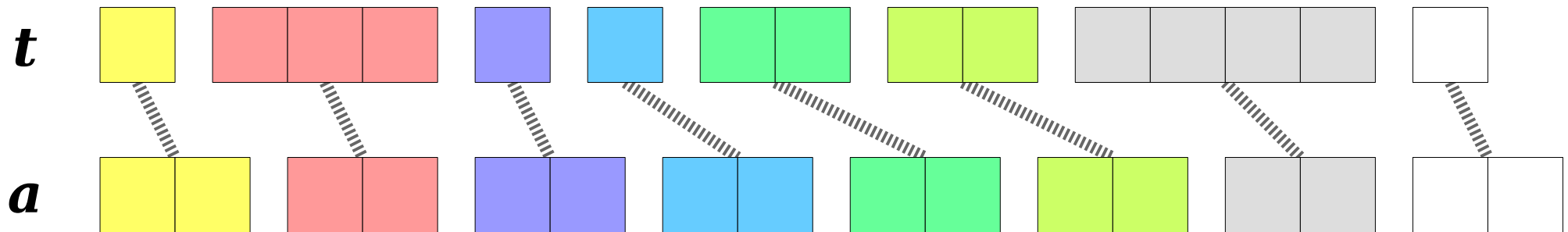
...the *actual* cost of performing those operations...

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

Amortized Analysis

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

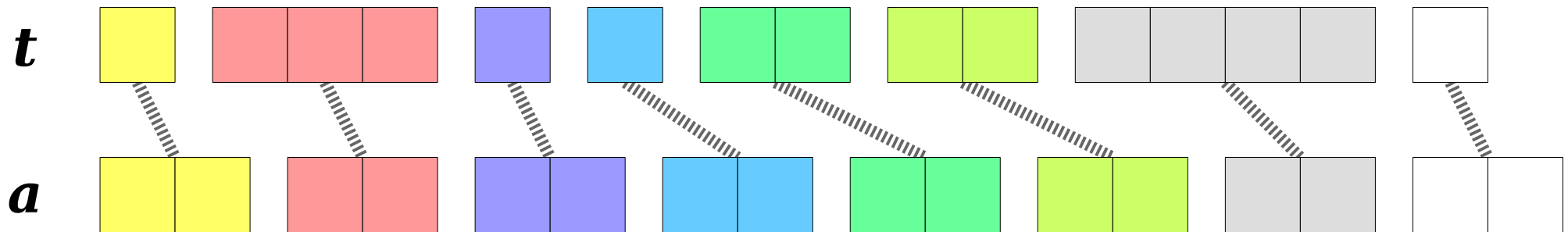
$$\forall k \leq m. \sum_{i=1}^k t(op_i) \leq \sum_{i=1}^k a(op_i).$$



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 underestimate the total time spent performing queue operations.

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

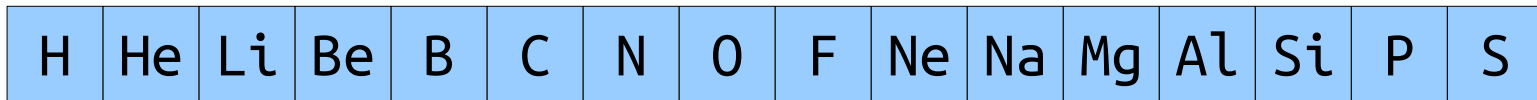


Major Questions

- 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

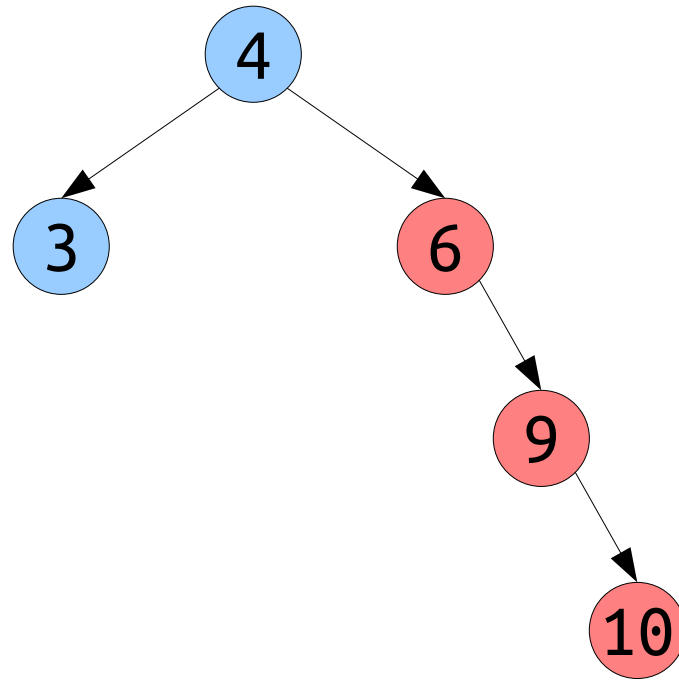


Most appends take time $O(1)$ and consume some free space.

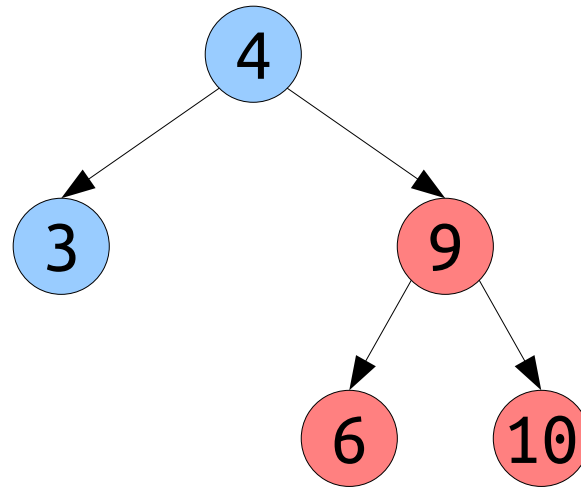
Every now and then, an append takes time $O(n)$, but produce a lot of free space.

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

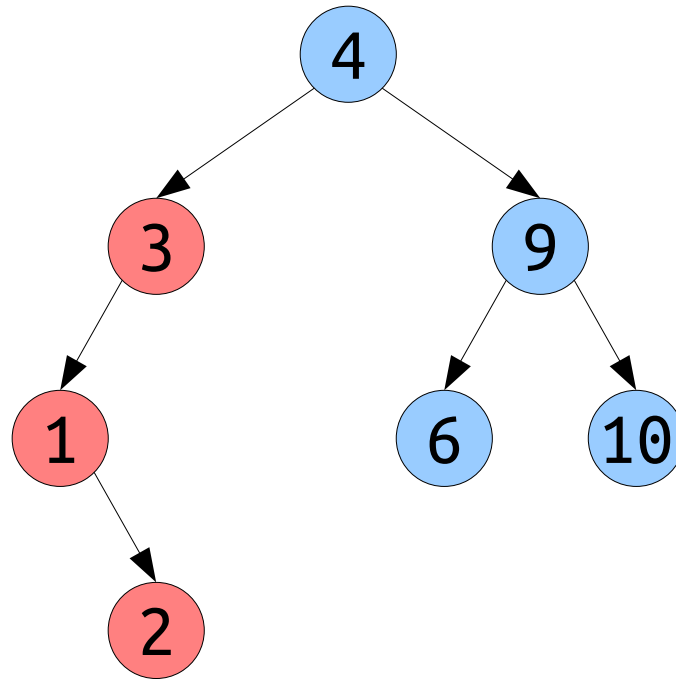
When Amortization Works



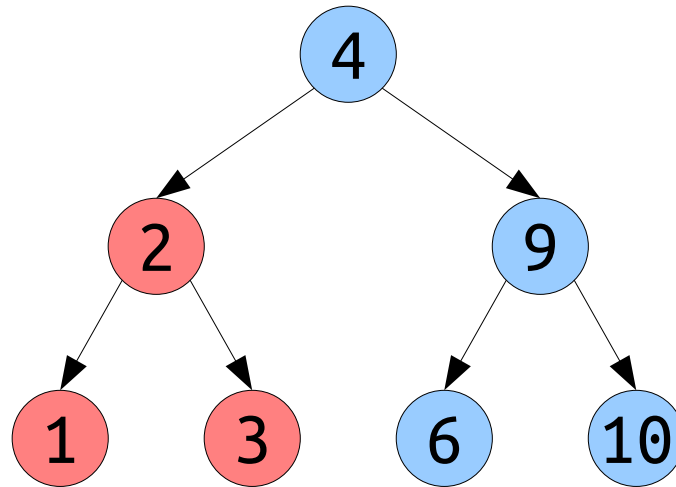
When Amortization Works



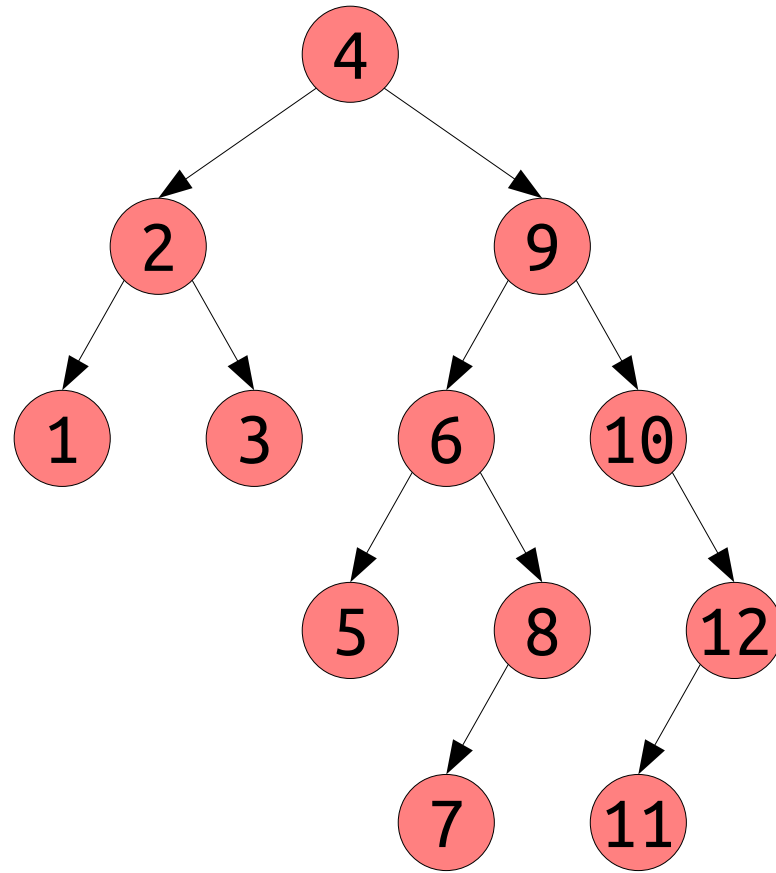
When Amortization Works



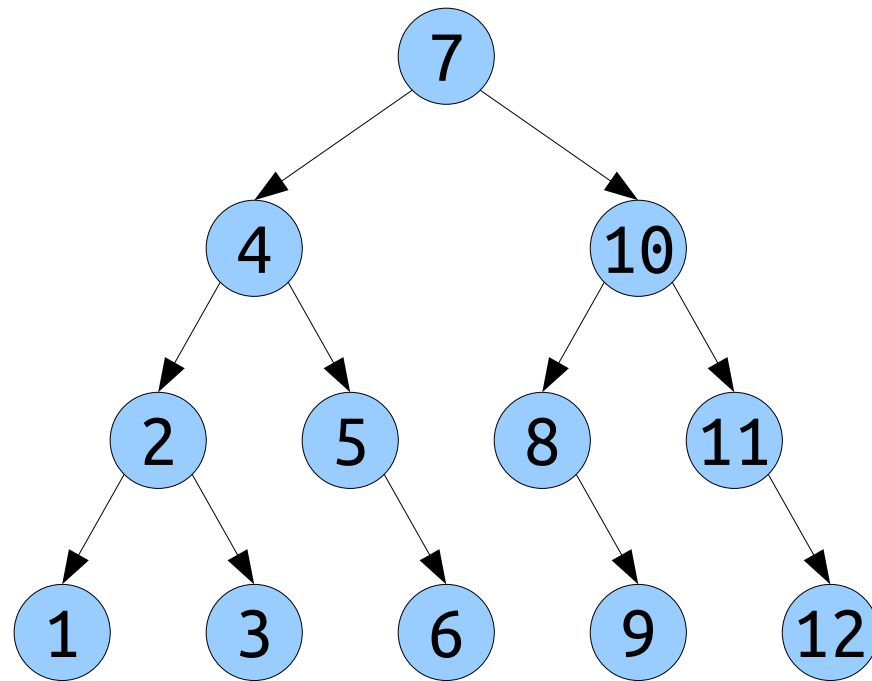
When Amortization Works



When Amortization Works



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

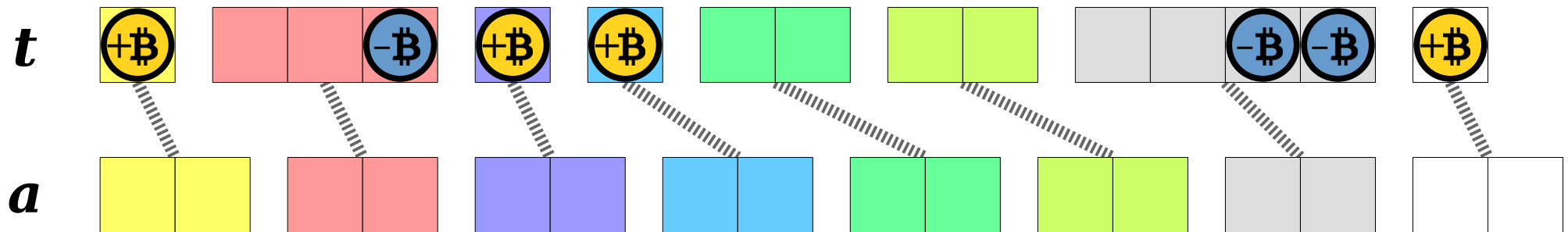
- You have a data structure where
 - imbalances accumulate slowly, and
 - imbalances get cleaned up quickly.
- You're fairly sure the cleanup costs will amortize away nicely.
- How do you assign amortized costs?

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 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.

Out

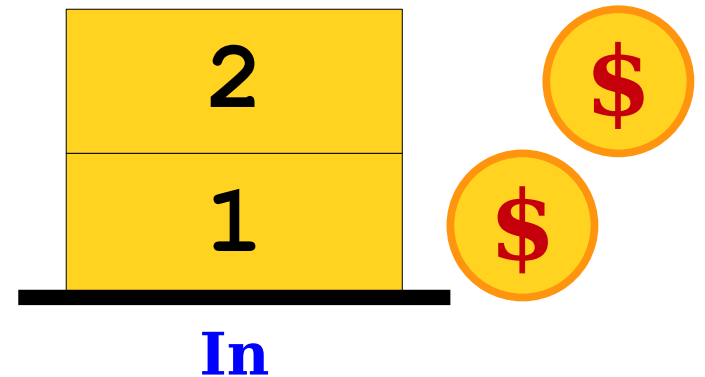
1 
In

The Two-Stack Queue

Actual work: $O(1)$
Credits added: 1

Amortized cost: **$O(1)$**

Out

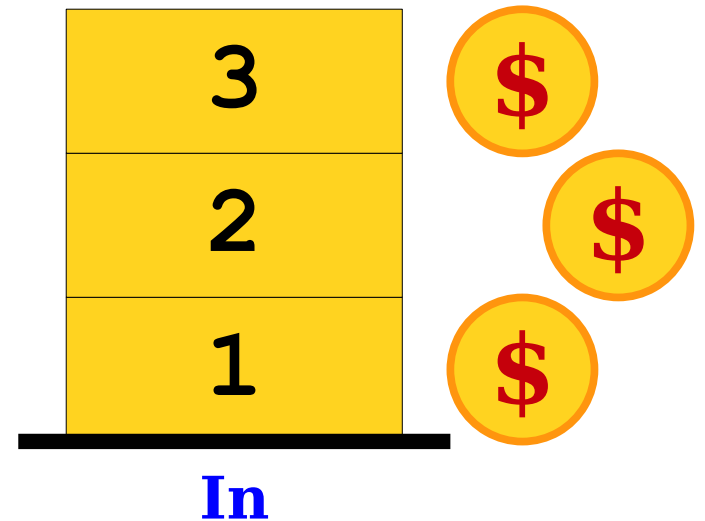


The Two-Stack Queue

Actual work: $O(1)$
Credits added: 1

Amortized cost: **$O(1)$**

Out

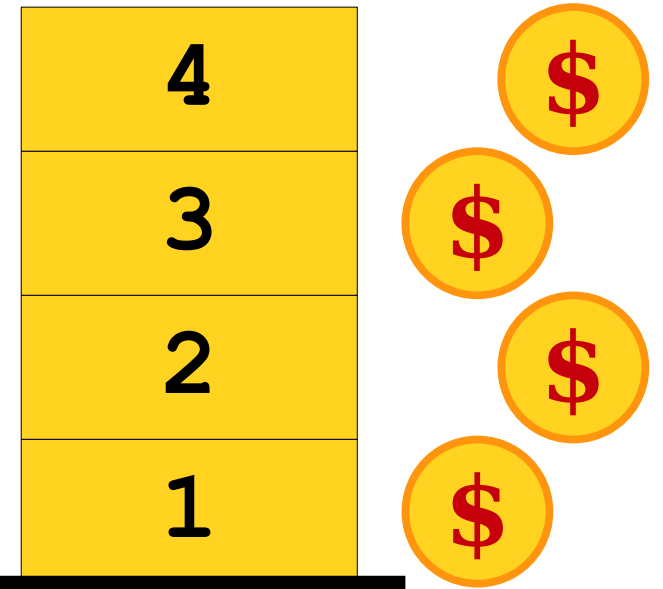


The Two-Stack Queue

Actual work: $O(1)$
Credits added: 1

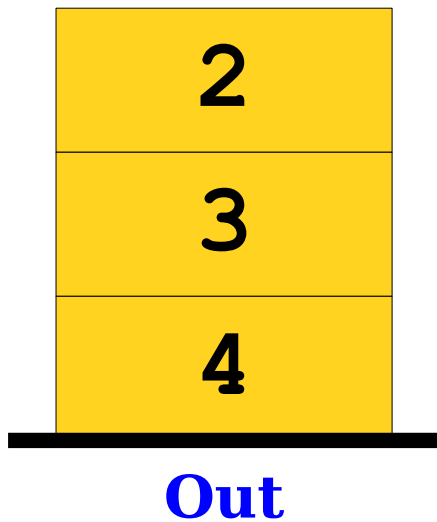
Amortized cost: **$O(1)$**

Out



In

The Two-Stack Queue



Actual work: $\Theta(k)$
Credits spent: k
Amortized cost: **$O(1)$**



Why This Works

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

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

Using the Banker's Method

- To perform an amortized analysis using the banker's method, do the following:
 - Figure out the actual runtimes of each operation.
 - Indicate where you'll place down credits, and compute the amortized cost of operations that place credits this way.
 - Indicate where you'll spend credits, and ***justify why the credits you intend to spend are guaranteed to be there***. Then, compute the amortized cost of each operation that spends credits this way.

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.

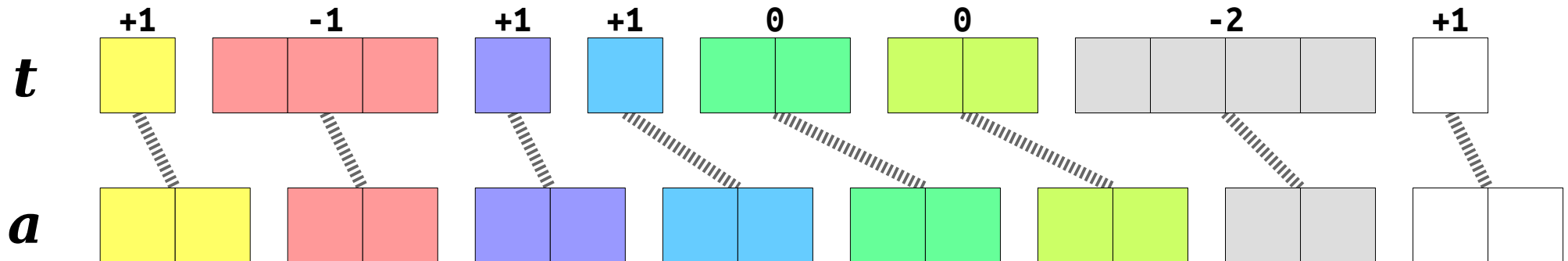
The Potential Method

- 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

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

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



The Two-Stack Queue

Φ = Height
of **In** Stack

Actual work: $O(1)$

$\Delta\Phi$: +1

Amortized cost: **$O(1)$**

Out

1
In

The Two-Stack Queue

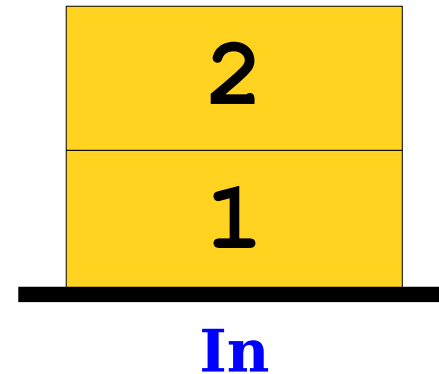
Φ = Height
of *In* Stack

Actual work: $O(1)$

$\Delta\Phi$: +1

Amortized cost: **$O(1)$**

Out



The Two-Stack Queue

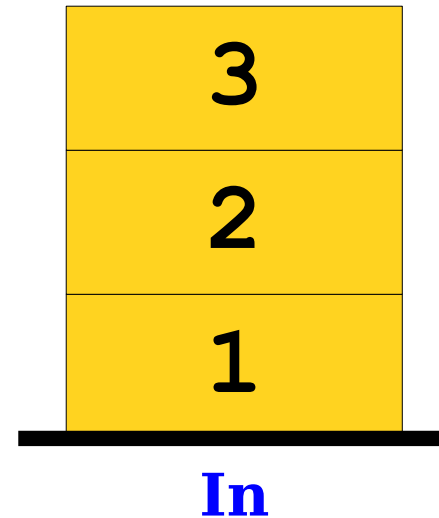
Φ = Height
of *In* Stack

Actual work: $O(1)$

$\Delta\Phi$: +1

Amortized cost: **$O(1)$**

Out

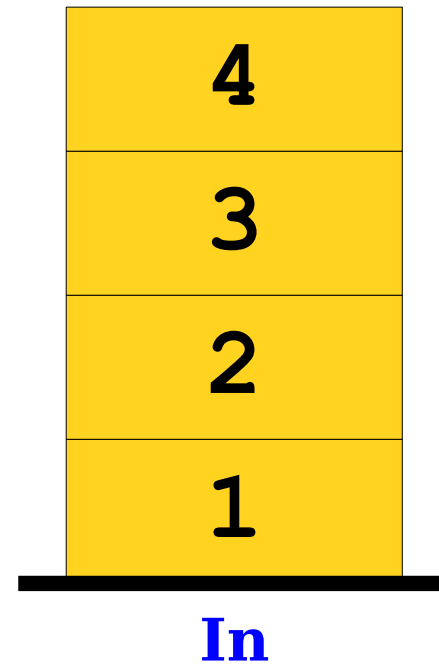


The Two-Stack Queue

Φ = Height
of *In* Stack

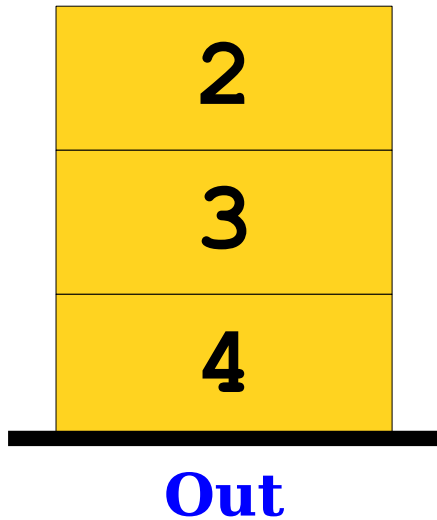
Actual work: $O(1)$
 $\Delta\Phi$: +1
Amortized cost: **$O(1)$**

Out



The Two-Stack Queue

Φ = Height
of **In** Stack



Actual work: $\Theta(k)$
 $\Delta\Phi$: $-k$
Amortized cost: **$O(1)$**



Why This Works

$$\begin{aligned}\sum_{i=1}^k a(op_i) &= \sum_{i=1}^k (t(op_i) + O(1) \cdot \Delta\Phi_i) \\ &= \sum_{i=1}^k t(op_i) + O(1) \cdot \sum_{i=1}^k \Delta\Phi_i\end{aligned}$$

Think “fundamental theorem of calculus,”
but for discrete derivatives!

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

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

Why This Works

$$\begin{aligned}\sum_{i=1}^k a(op_i) &= \sum_{i=1}^k (t(op_i) + O(1) \cdot \Delta\Phi_i) \\ &= \sum_{i=1}^k t(op_i) + O(1) \cdot \sum_{i=1}^k \Delta\Phi_i \\ &= \sum_{i=1}^k t(op_i) + O(1) \cdot (\textit{net change in potential}) \\ &\geq \sum_{i=1}^k t(op_i)\end{aligned}$$

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

Using the Potential Method

- To perform an amortized analysis using the potential method, do the following:
 - Figure out the actual runtimes of each operation.
 - Define your potential function Φ , and ***explain why it's initially zero or otherwise account for a nonzero start potential.***
 - For each operation, determine its $\Delta\Phi$.
 - Compute the amortized costs of each operation.

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.

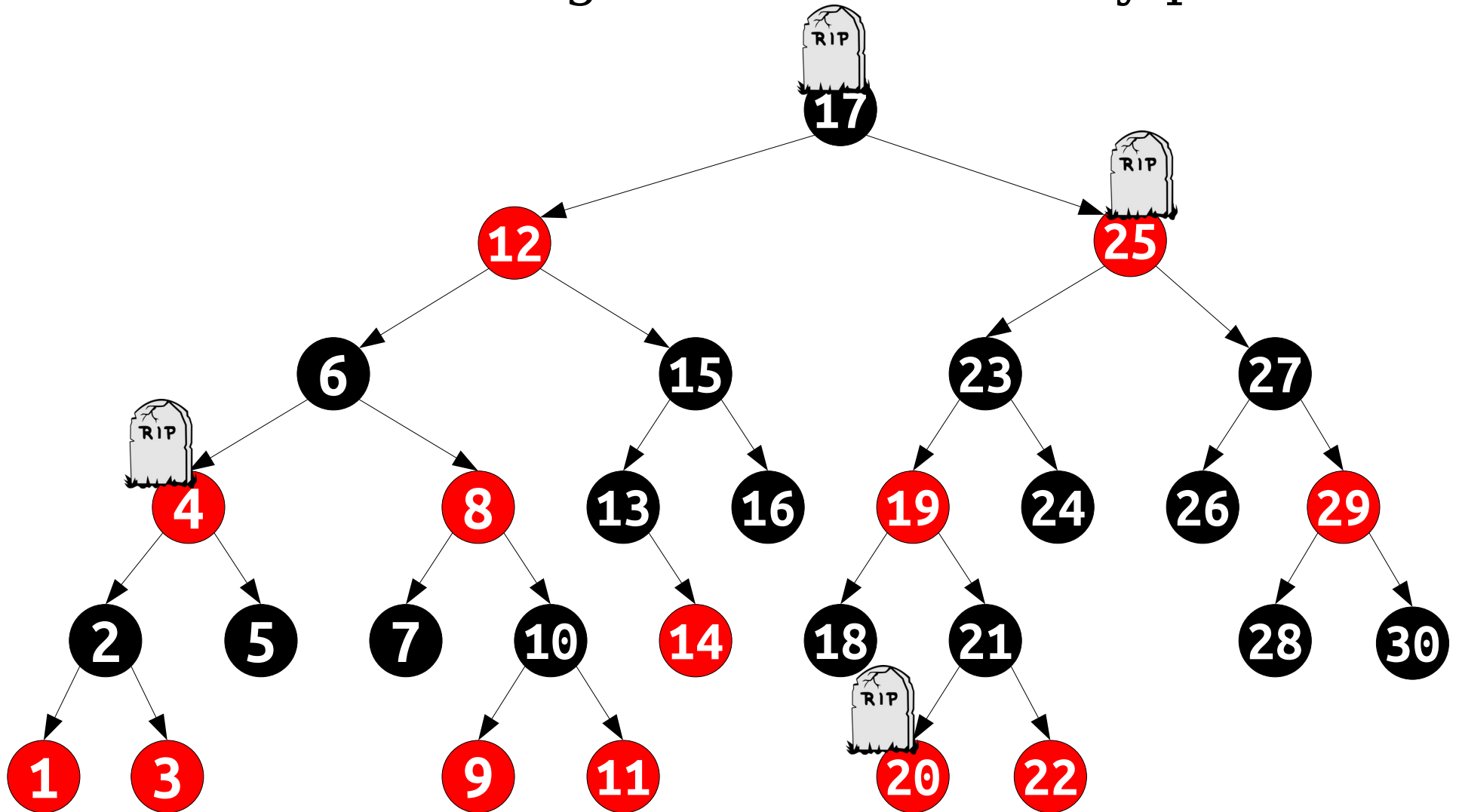
Deleting from a BST

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?

Dead Simple Deletions

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

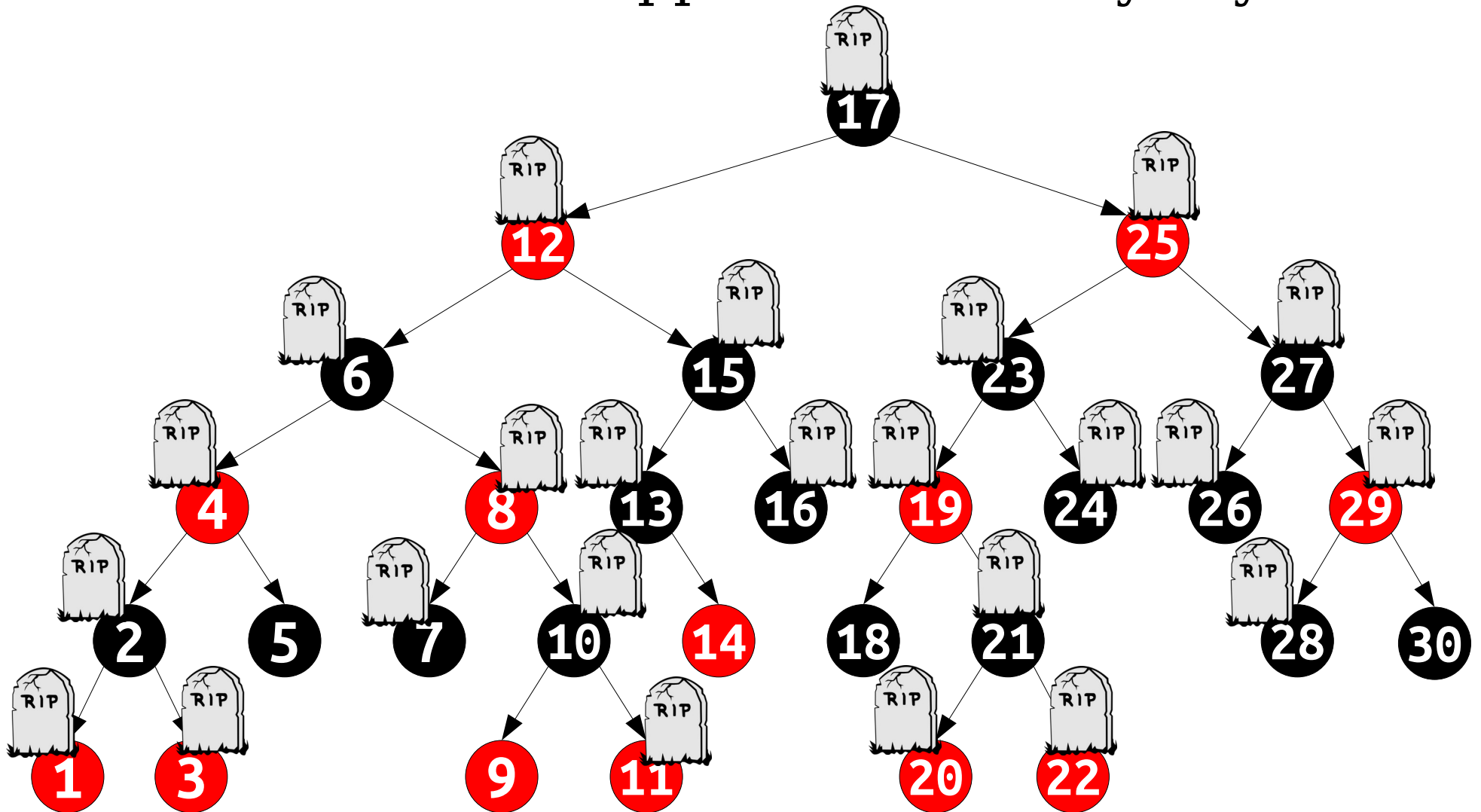


Dead Simple Deletions

- 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.

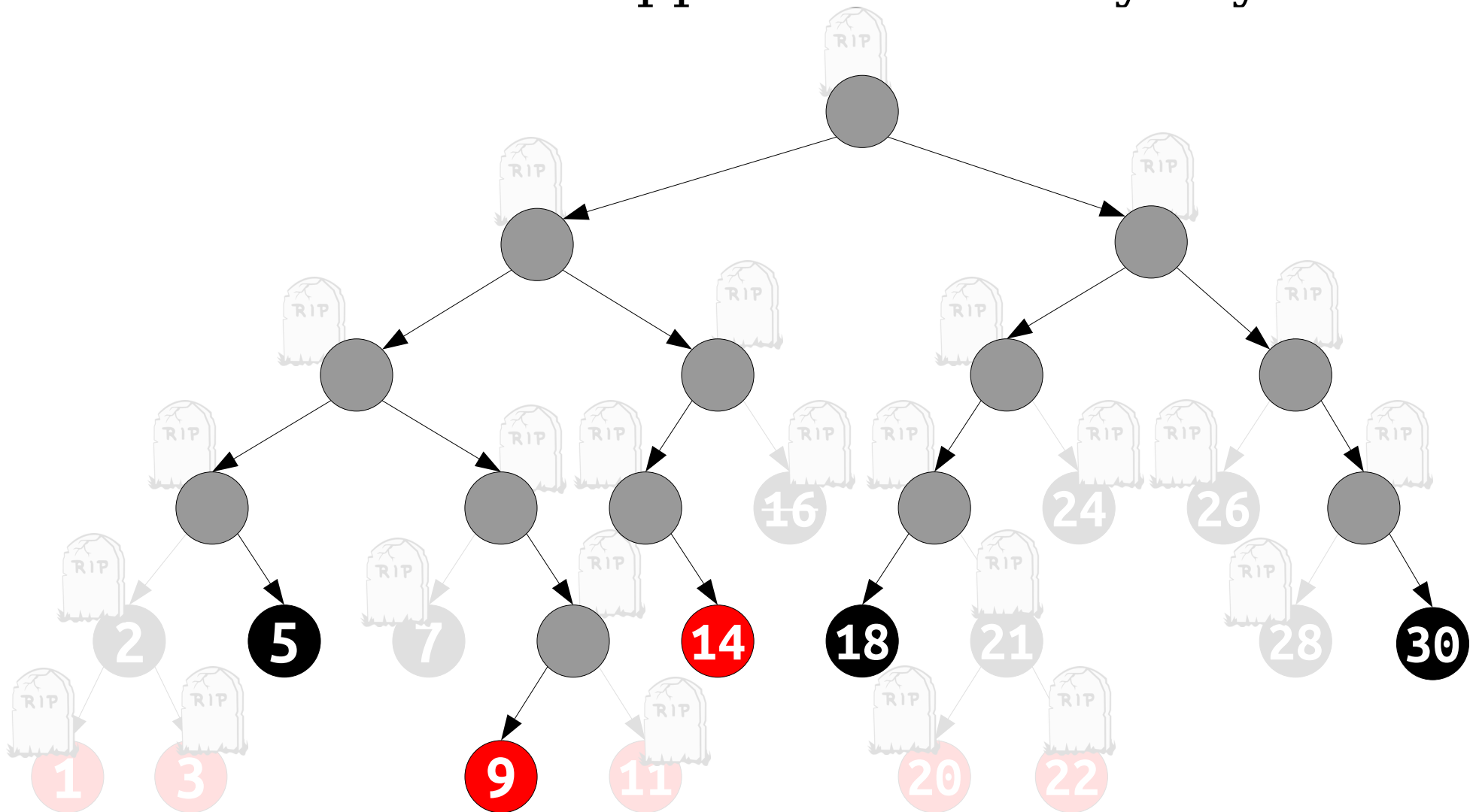
Dead Simple Deletions

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



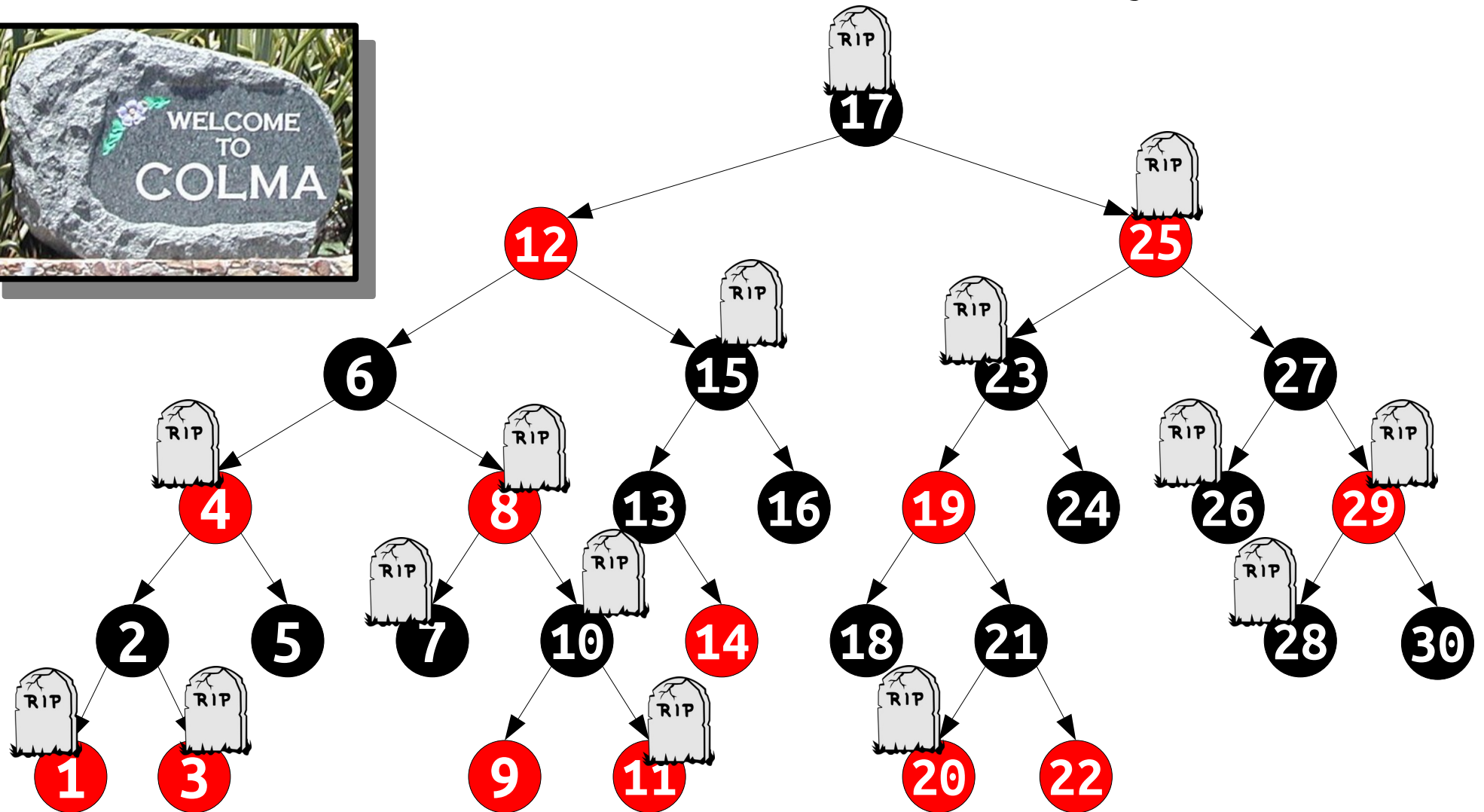
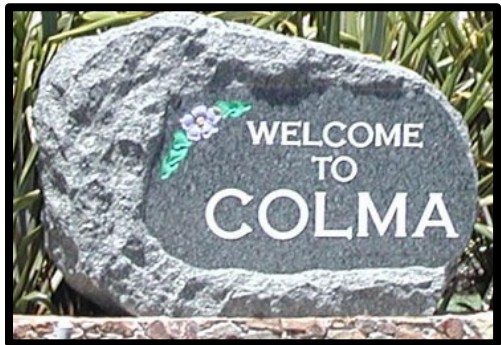
Dead Simple Deletions

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



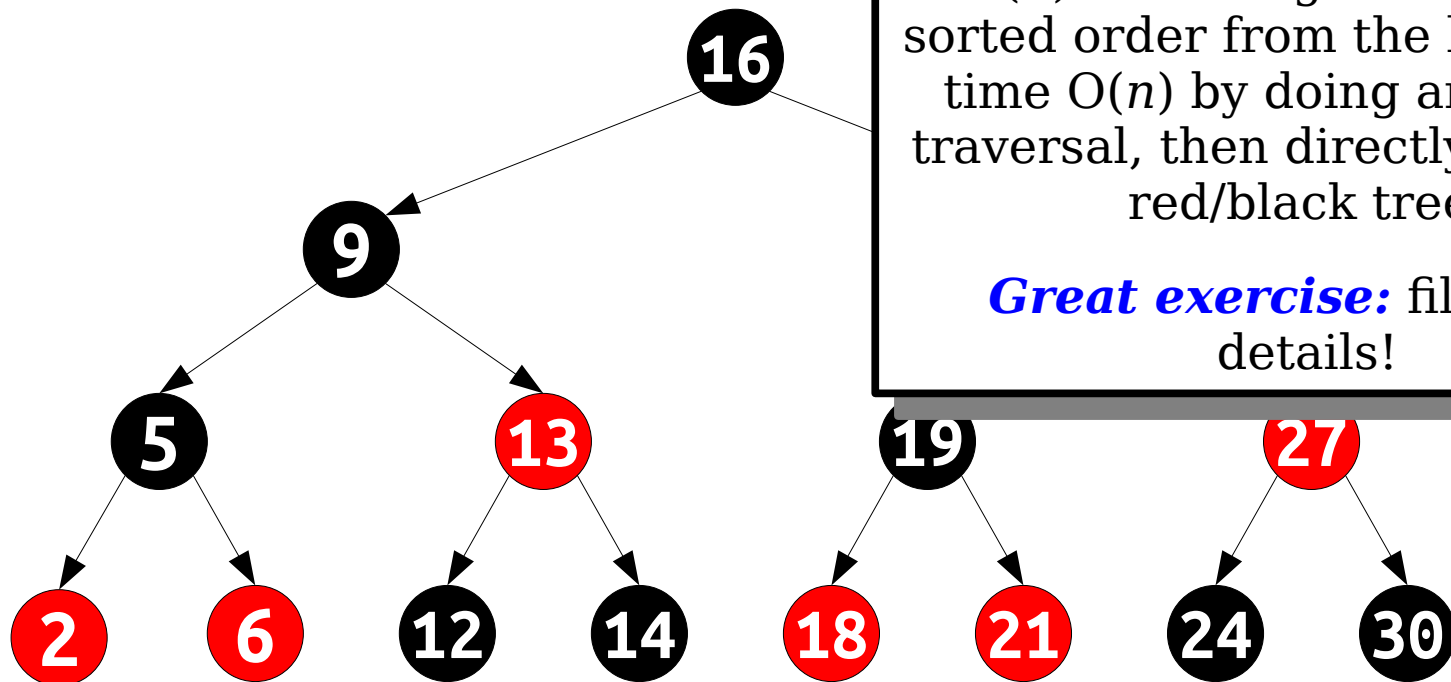
Dead Simple Deletions

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



Dead Simple Deletions

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

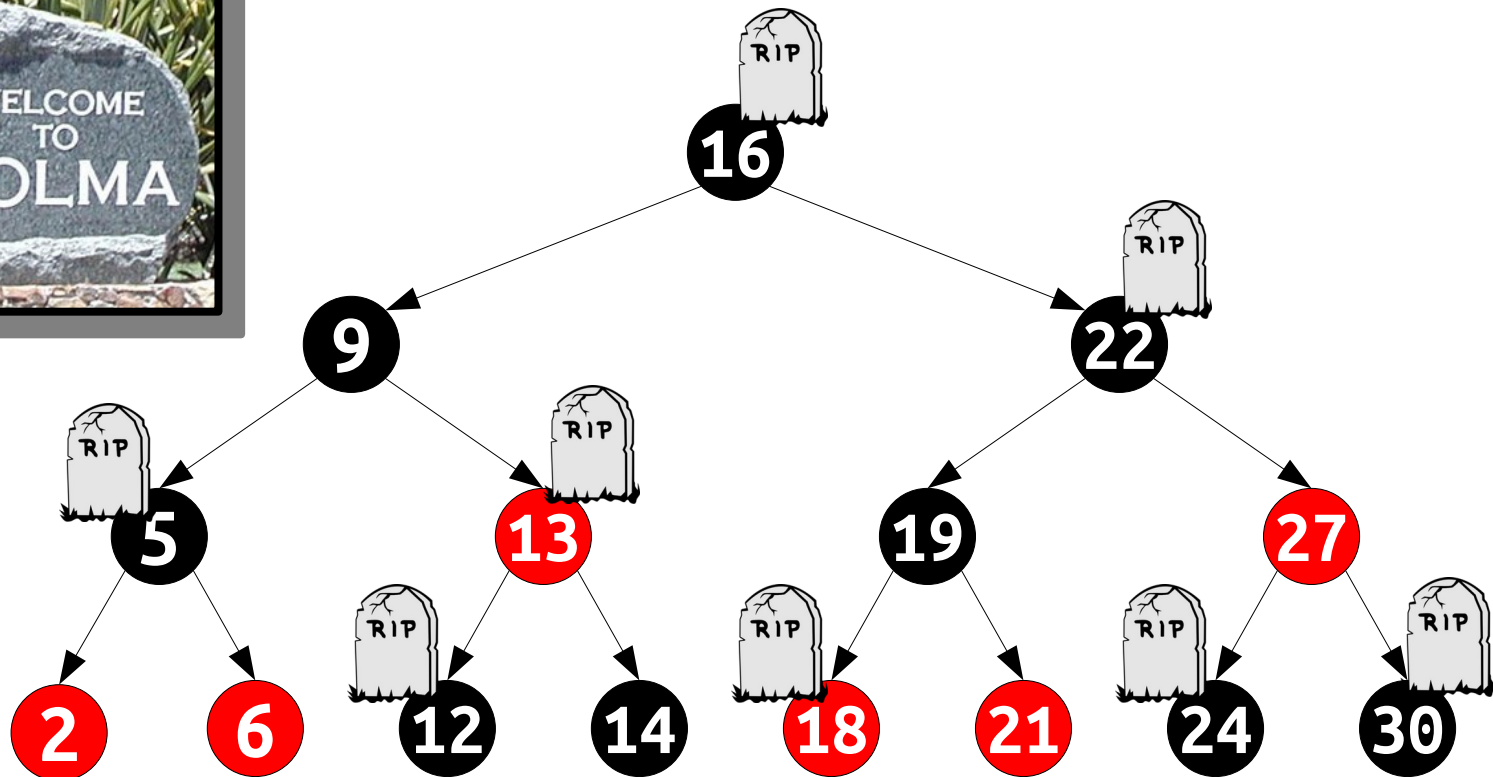
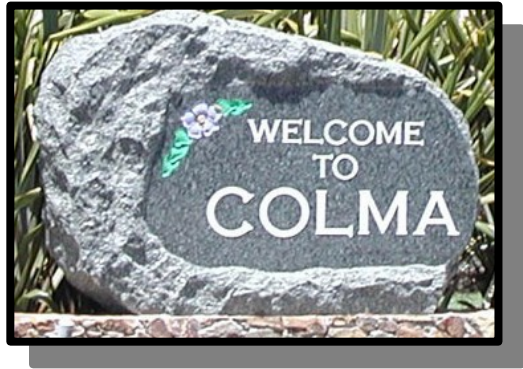


We can rebuild this tree in time $O(n)$. We can get the keys in sorted order from the last BST in time $O(n)$ by doing an inorder traversal, then directly build the red/black tree.

Great exercise: fill in the details!

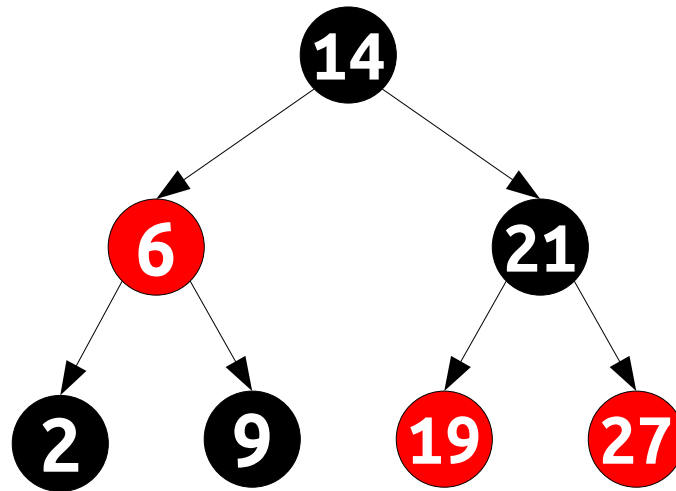
Dead Simple Deletions

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



Dead Simple Deletions

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

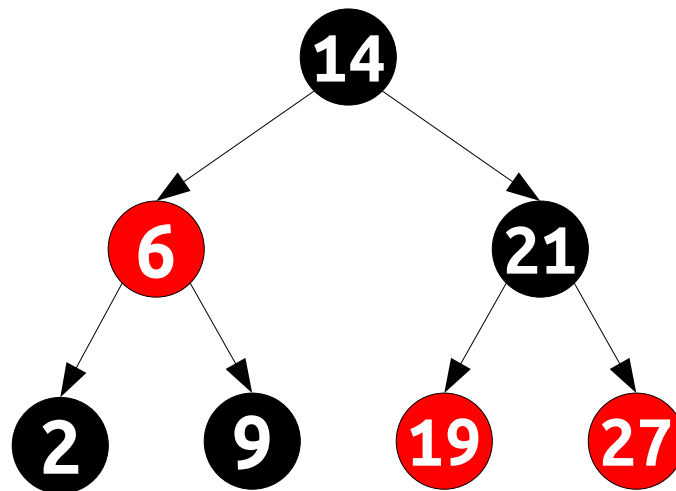


Analyzing Lazy Rebuilding

- 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?

Amortized 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: $\Theta(n)$.
 - 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.