Amortized Analysis

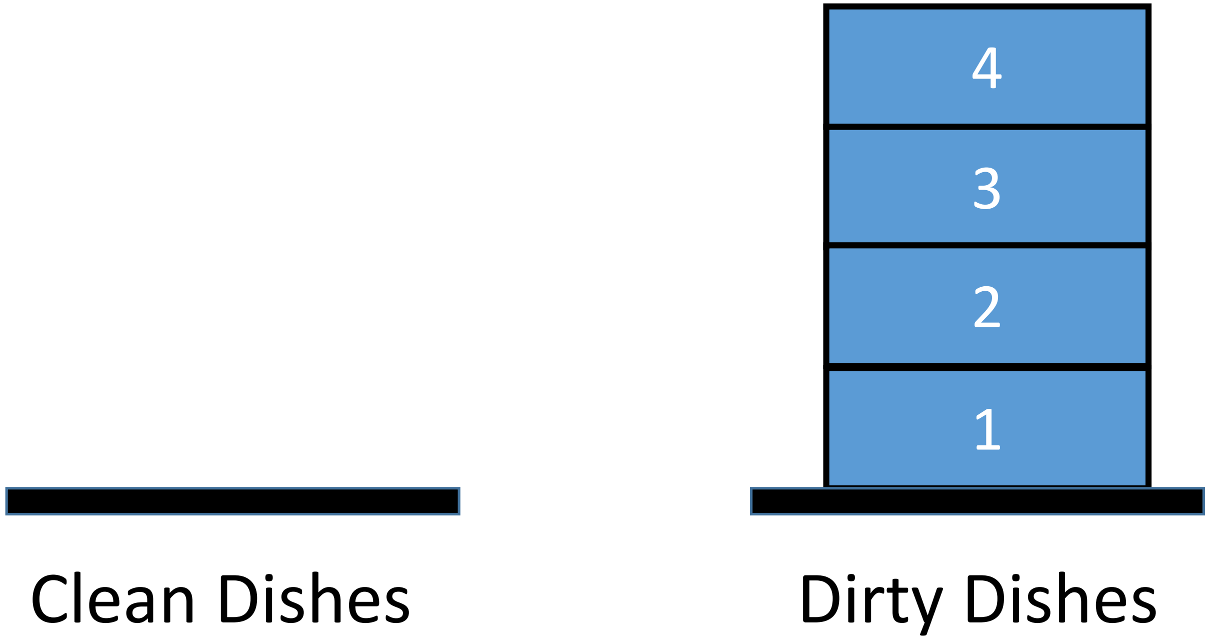
Doing the Dishes

- What do I do with a dirty dish or kitchen utensil?
- Option 1: Wash it by hand
  - 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.
- Option 2: Put it in the dishwasher rack, then run the dishwasher when it full.
  - Running dishwasher is faster in aggregate, but means I may have to wait a bit for dishes to be ready.

Key Idea

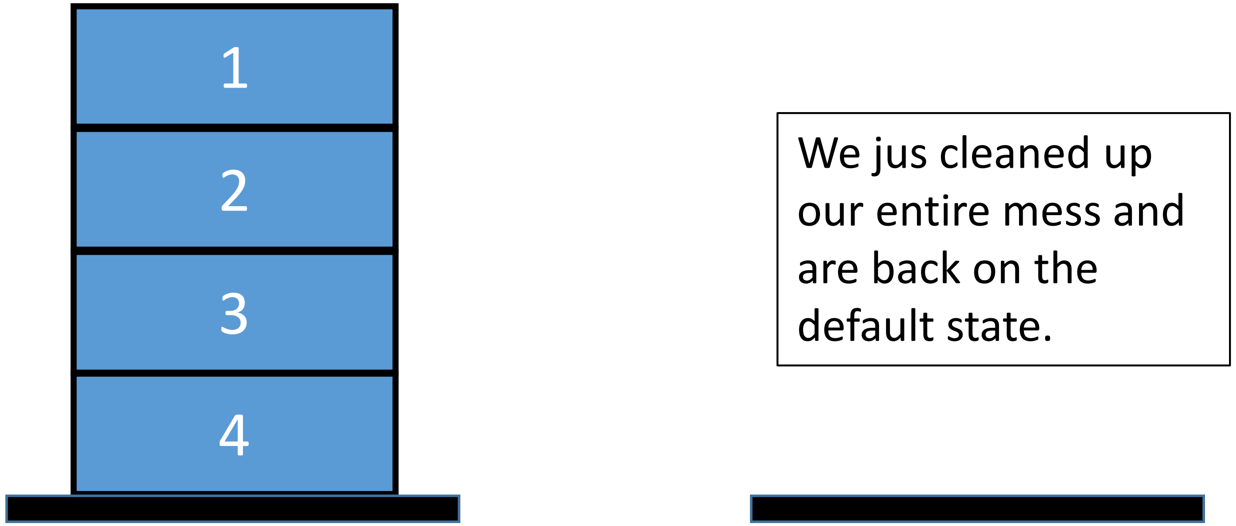
Designing the data structures that trades per-operation efficiency for overall efficiency.

The Two-Stack Queue



Dirty dishes are piling up because we did not do any work to clean them when we added them in.

The Two-Stack Queue

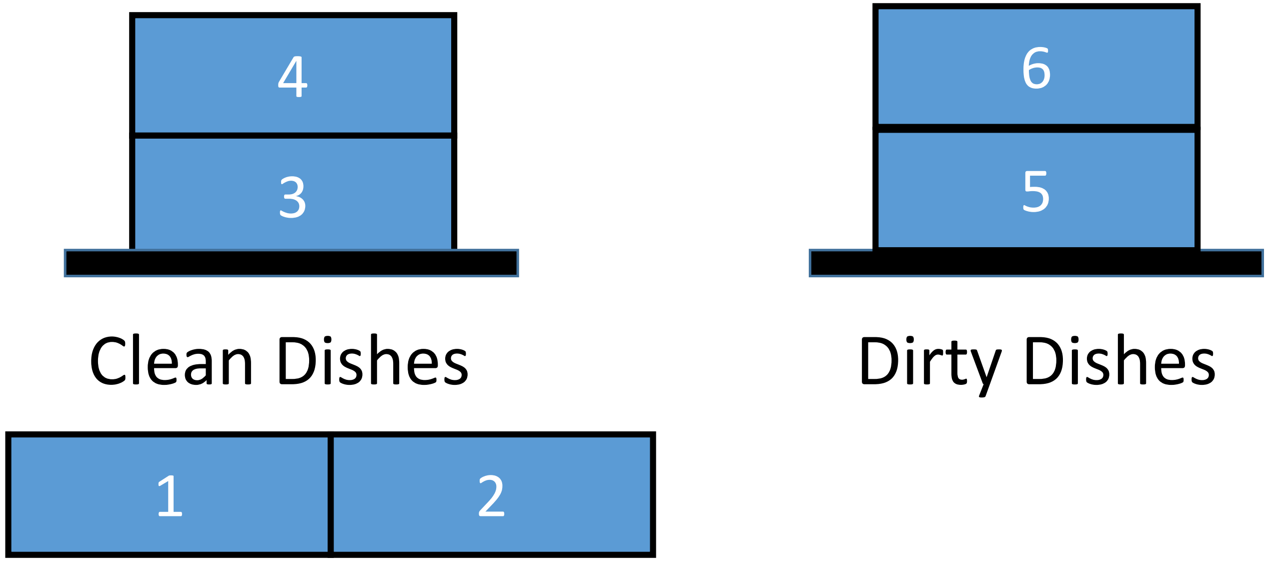


We jus cleaned up our entire mess and are back on the default state.

Clean Dishes

Dirty Dishes

The Two-Stack Queue

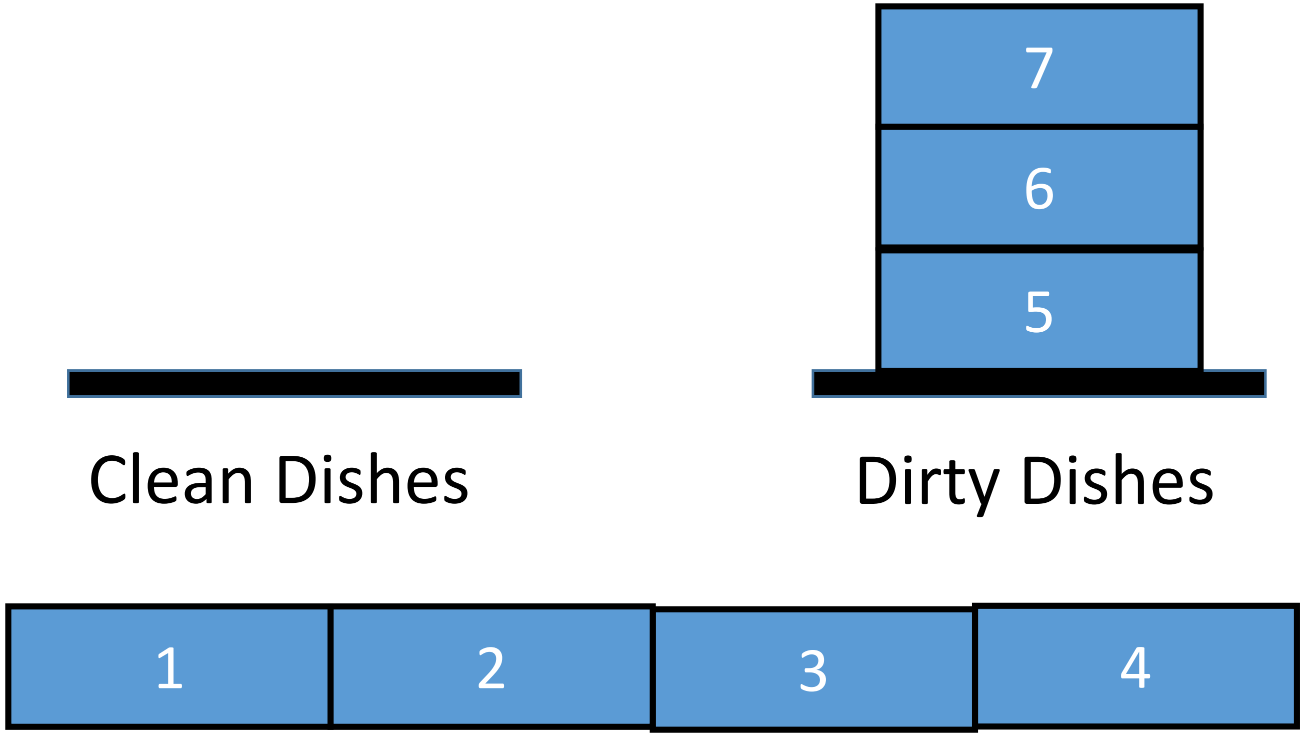


Clean Dishes

Dirty Dishes

We need to start the new batch of dirty dishes

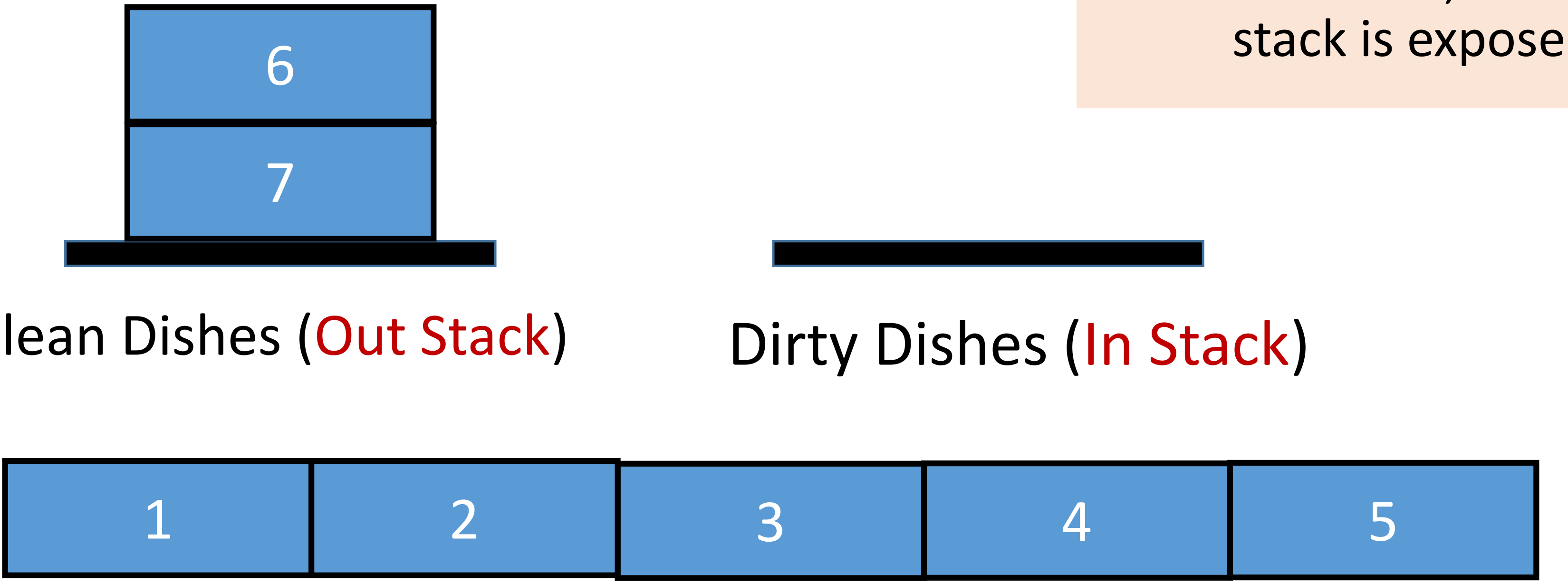
The Two-Stack Queue



We need to start the new batch of dirty dishes

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.



Clean Dishes (Out Stack)

Dirty Dishes (In Stack)

Stack operations

- Stack has two fundamental operations each of which takes O(1) time:

PUSH(S, x) pushes object x onto stack S

POP(S, x) pops the top of stack S and returns the popped object.

Since each of these operations runs in O(1) time, let us consider the cost of each to be 1. The total cost of a sequence of **n** PUSH and POP operations is therefore **n**, and the actual running time for n operations O(n).

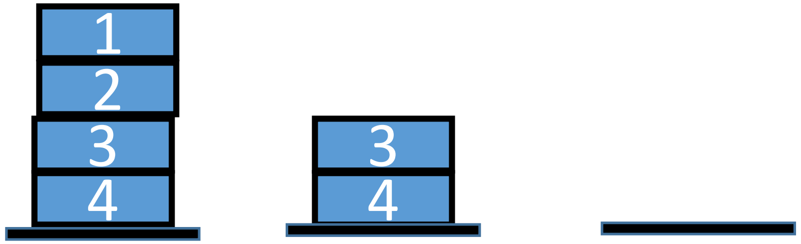
If we add he stack operation MULTIPOP(S, k), which removes the k top objects of stack S, or pops the entire stack if it contains less than **k** objects.

Stack Operations

- In the following pseudocode, the operation STACK-EMPTY returns TRUE if there are not objects currently on the stack, and FALSE otherwise.

```
Line 1)  MULTIPOP(S, k) // out Stack
Line 2)  While not STACK-EMPTY and k!=0
Line 3) do POP(S)
Line 4)       k <=k-1;
```

What is the running time of MULTIPOP(S, k)? For each iteration of the loop, one call is made to POP in line 2. The total cost of MULTIPOP is min(s, k) of object popped off the stack. The actual running time is a linear function of this cost.



(a) Clean Dishes

(b) Clean Dishes

(c) Clean Dishes

- The action of MULTIPOP on a stack S (clean dishes stack), shown:

a) Initially of stack S

b) The top 2 objects are popped by MULTIPOP(S, 2), whose result is shown in (b).

c) The next operation is MULTIPOP(S, 4), which empties the stack shown in c) since there were fewer than 4 objects remaining.

- In fact, although a single MULTIPOP operation can be **expensive**, any sequence of *n* PUSH, POP, and MULTIPOP operations on an initially empty stack can cost at most ***O*(*n*)**. Why?
- Each object can be popped at most once for each time it is pushed. Therefore, the number of times that POP can be called on a nonempty stack, including calls within MULTIPOP, is at most the number of PUSH operations, which is at most *n*. For any value of *n*, any sequence of *n* PUSH, POP, and MULTIPOP operations takes a total of *O*(*n*) time.
- **The amortized cost of an operation is the average: *O*(*n*)/*n* = *O*(1).**

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 is not empty -
  - Could be O(n) 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

- Intuition: We only do expensive dequeue after a long run a cheap enqueues.
- The “dishwasher”: we very slowly introduce a lot of dirty dishes to get cleanup 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!

Analyzing the Queue

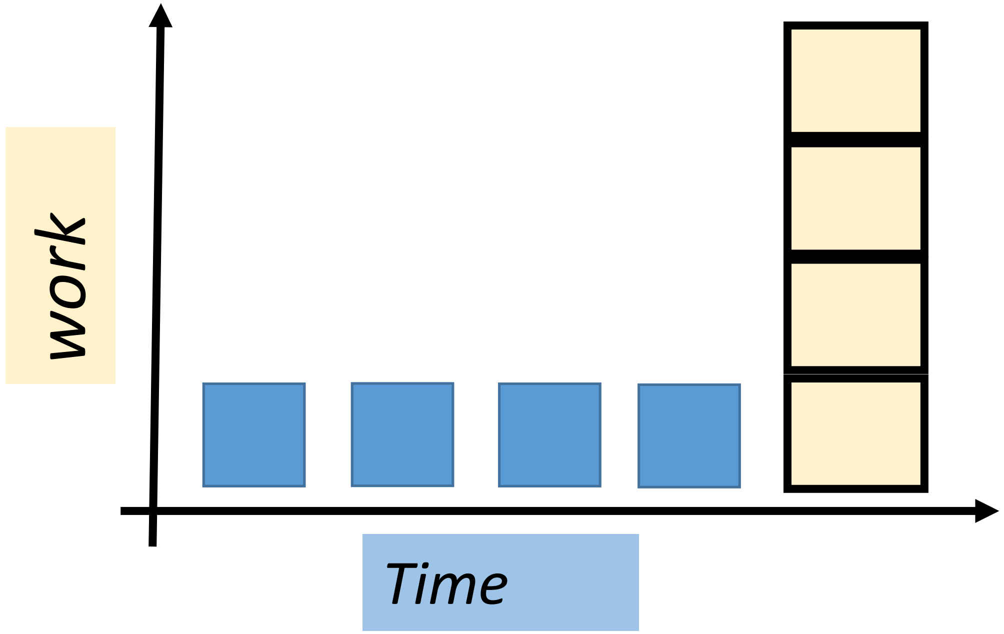
Summary our result using an average case analysis:

- 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).

Building a Dam

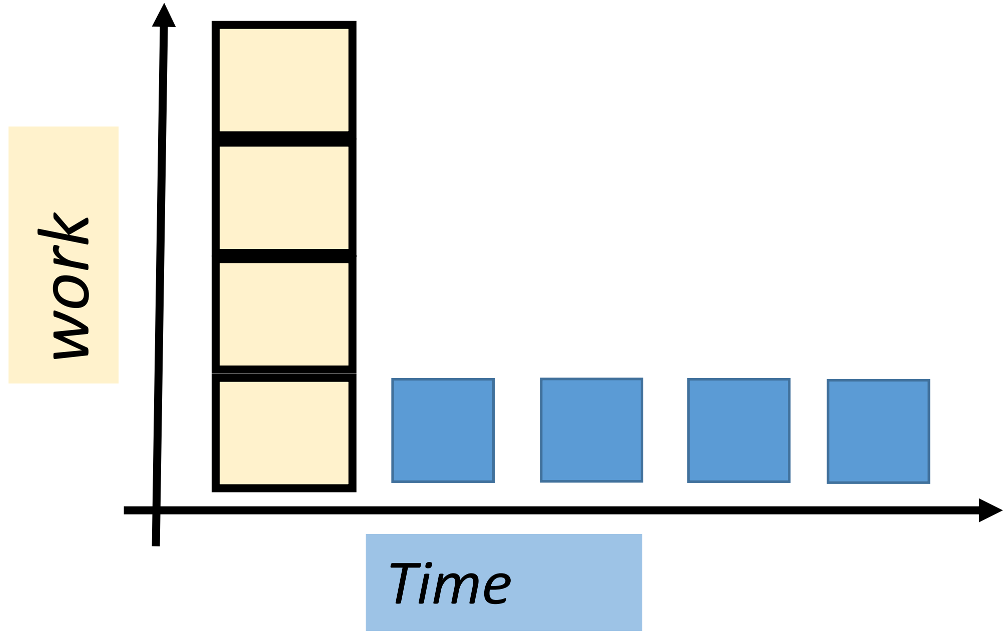
- Building this dam is an enormous up-front cost, but pays for itself in the long term … assuming it last long?



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

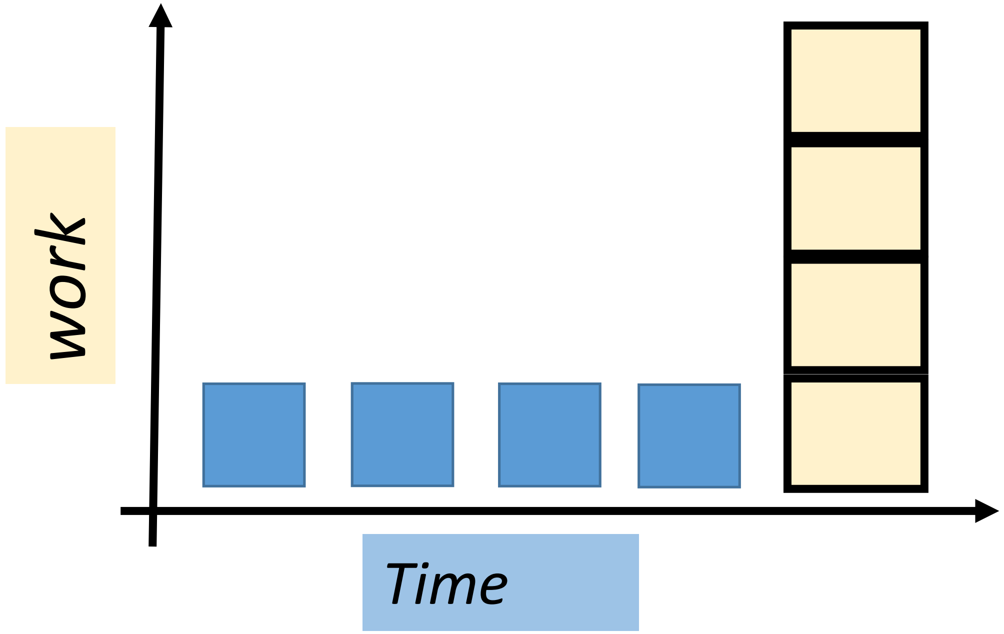
Early expensive operation that pays off in the long term.

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

Grocery Stores

- The grocery stores don’t need to stock up huge quantities of every items because, on **average**, people aren’t buying the same thing …

… But if they all want toilet paper …



Dishwasher Model

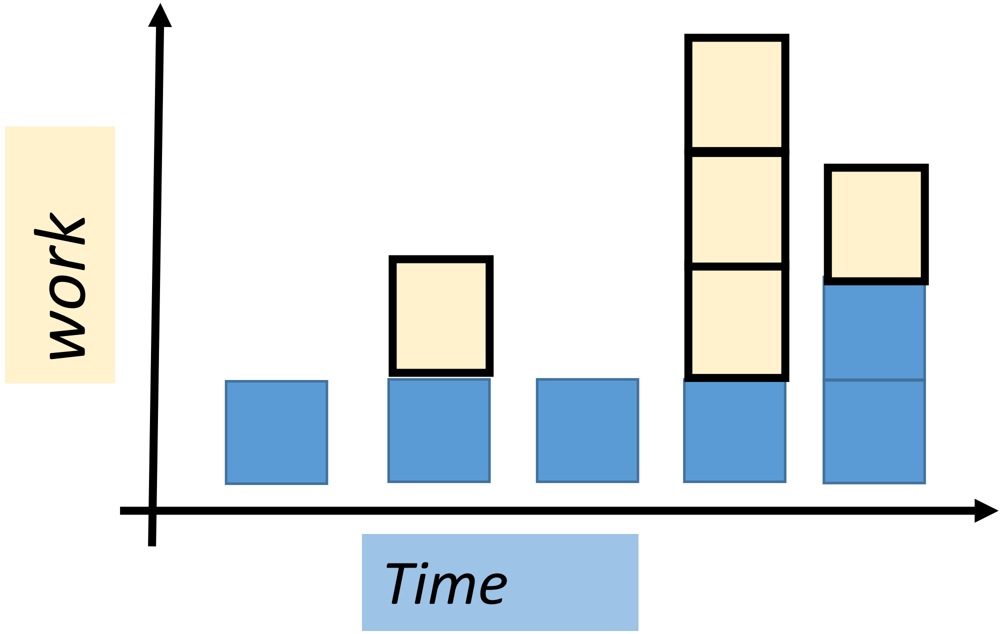
Lots of cheap operations that need to be made up for by an expensive one later.

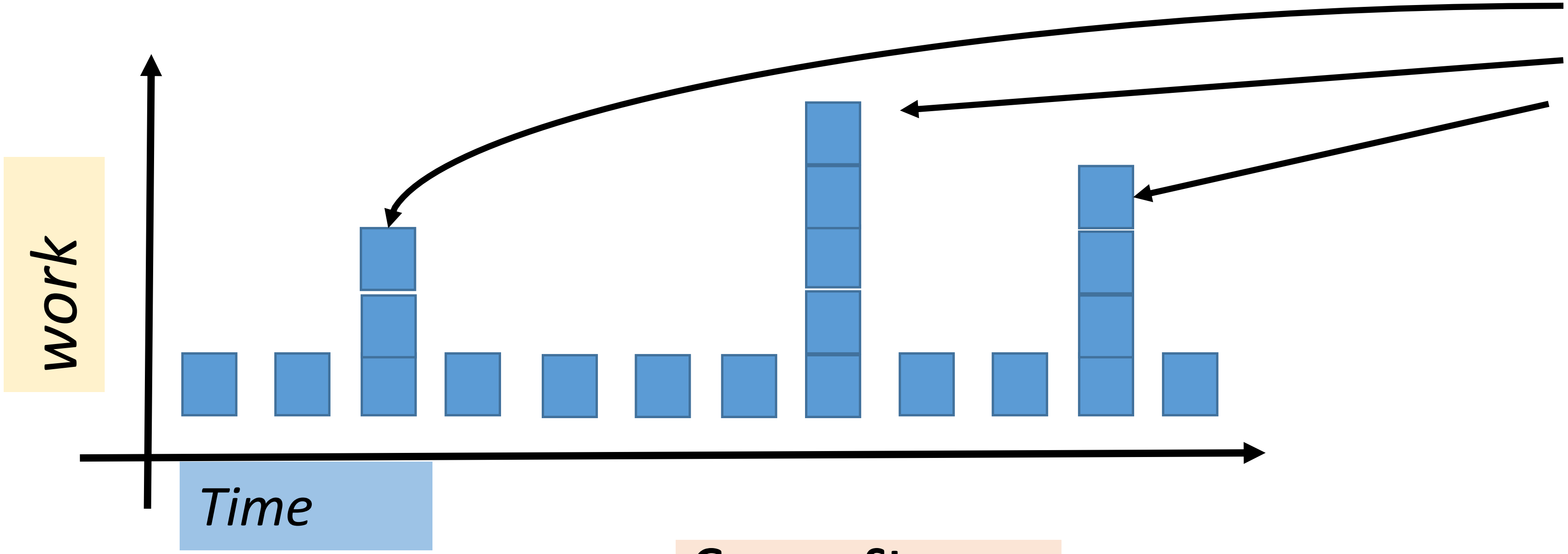
The **average** work done at each point in time is low.

Grocery Store Model

It is unlikely that there will be any large operations because of randomization.

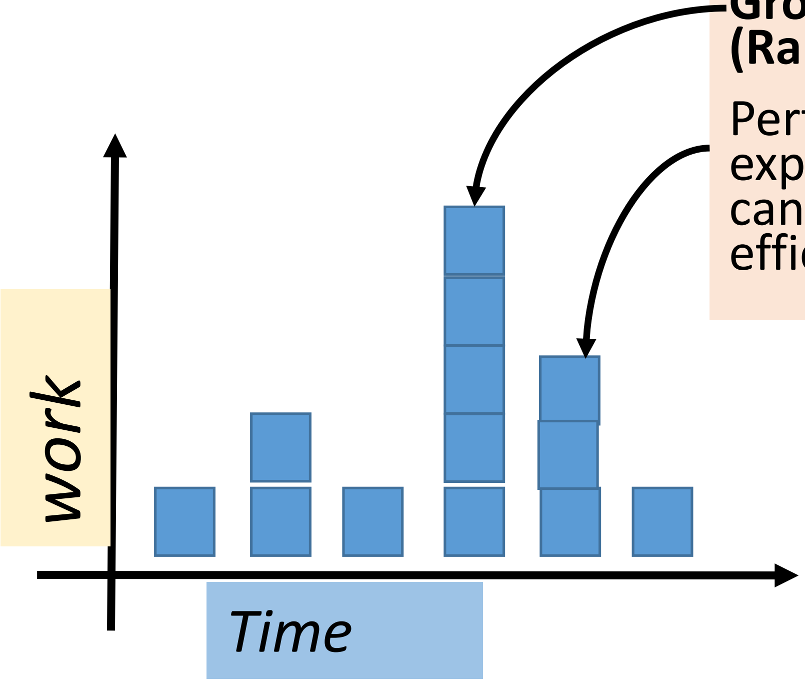
Except that, every now and then, we run into trouble …





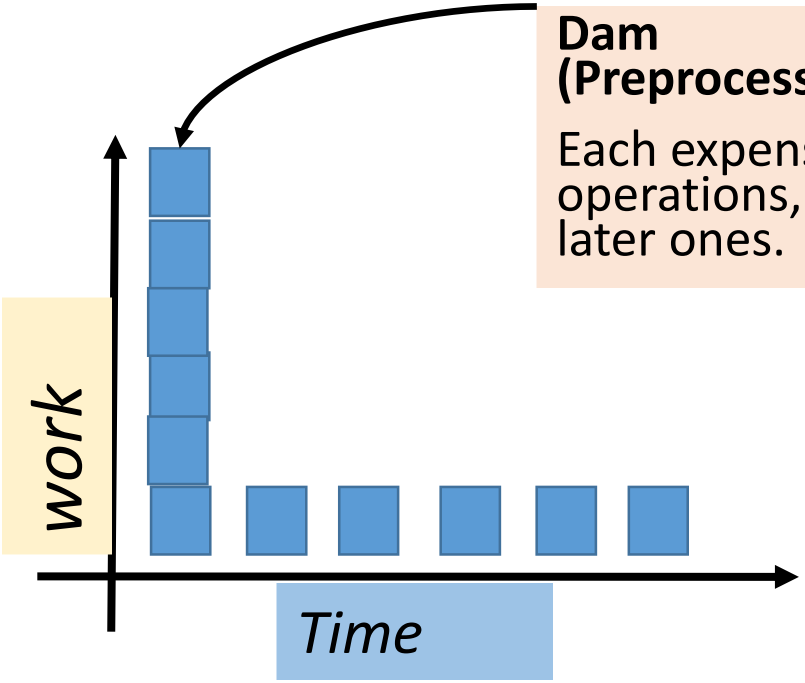
Dishwasher (Amortization)

Each expensive operation is preceded by lots of cheap ones.



Grocery Store (Randomization)

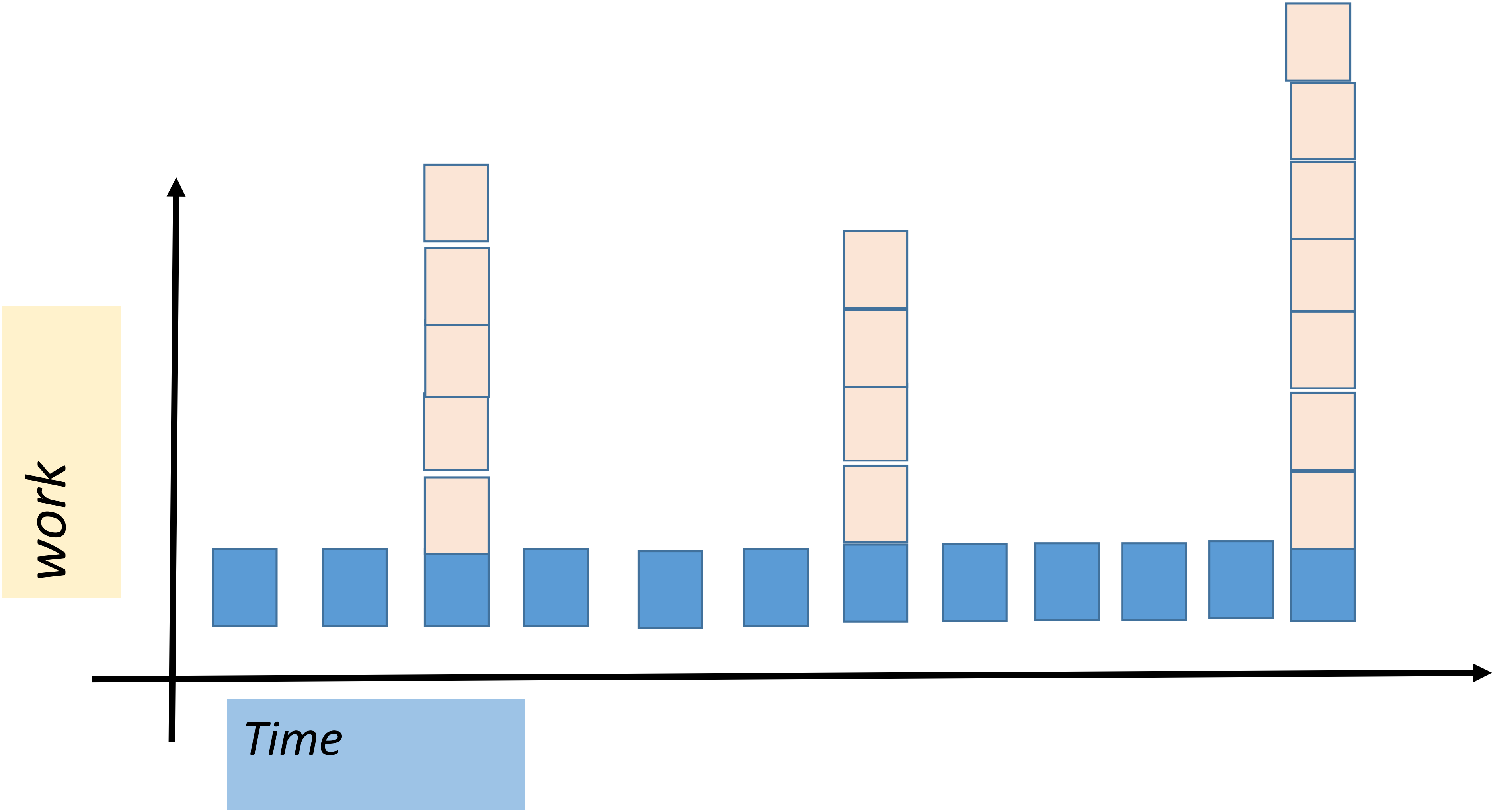
Performs well on expectation, can’t guarantee efficiency.



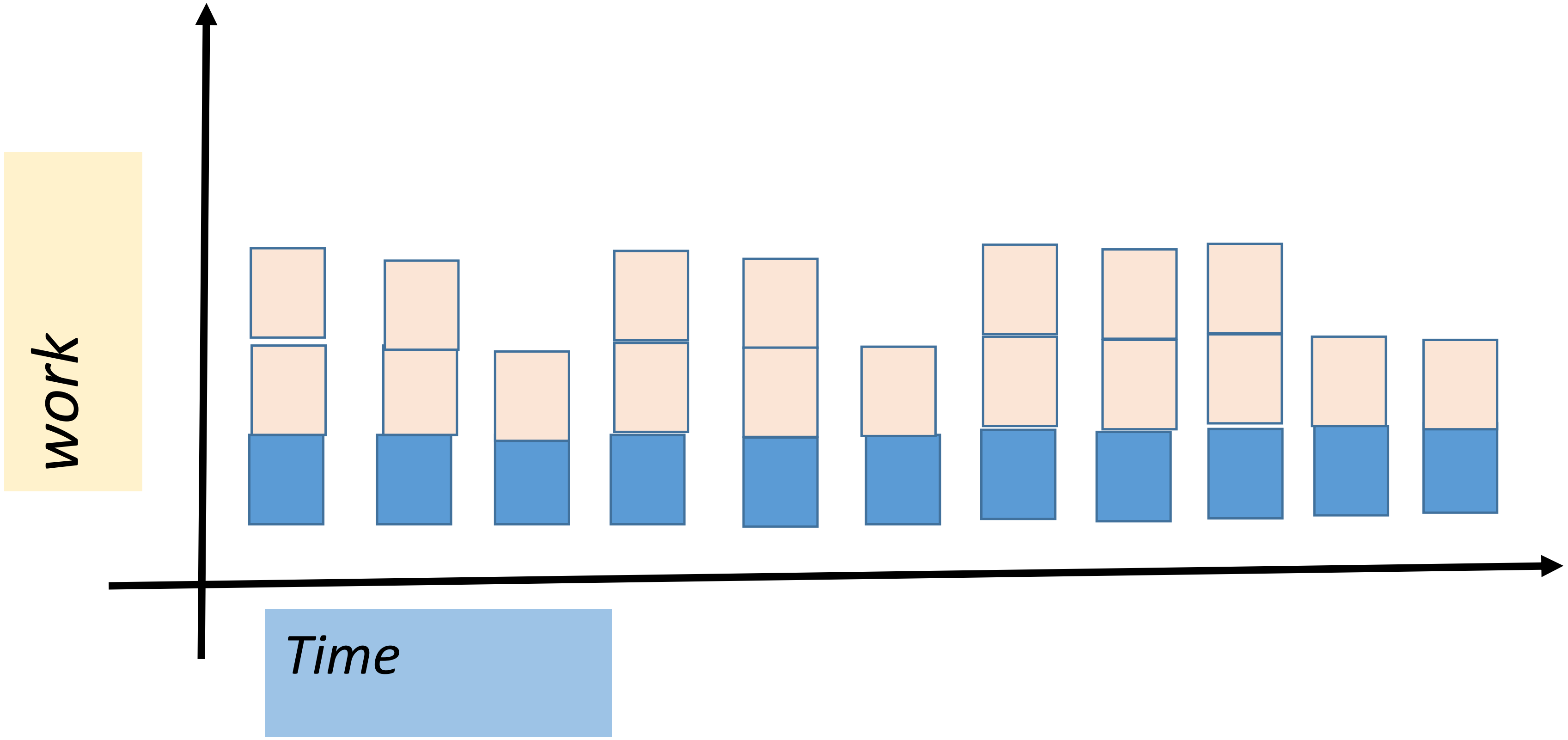
Dam (Preprocessing)

Each expensive operations, cheap later ones.

What Amortization Means?



Key Takeaway: Back-charge expensive operations to cheaper ones.



Key Takeaway: Back-charge expensive operations to cheaper ones.

Amortization works best if

1. imbalances accumulates slowly, and
2. Imbalances get cleaned up quickly

Amortized Analysis

- Suppose we perform a series of operation op1, op2, …, opm.
- The amount of time taken to execute operation opi is denoted by t(opi)
- For each operation opi, pick a value a(opi), called the amortized cost of opi, such that.

$$\underbrace{\forall k \leq m.}_{\boxed{\text{When we stop performing operations (k <= m)}}} \; \underbrace{\sum_{i=1}^{k} t(op_i)}_{\boxed{\text{The actual cost of performing those operations …}}} \leq \underbrace{\sum_{i=1}^{k} a(op_i).}_{\boxed{\text{… is at most the amortized cost of performing those operations}}}$$

Performing Amortized Analyses

- You have a data structure where
    - Imbalances accumulate slowly, and
    - Imbalance get cleaned up quickly.
- You are 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 negative time –O(1).
- The amortized cost of an operation is then

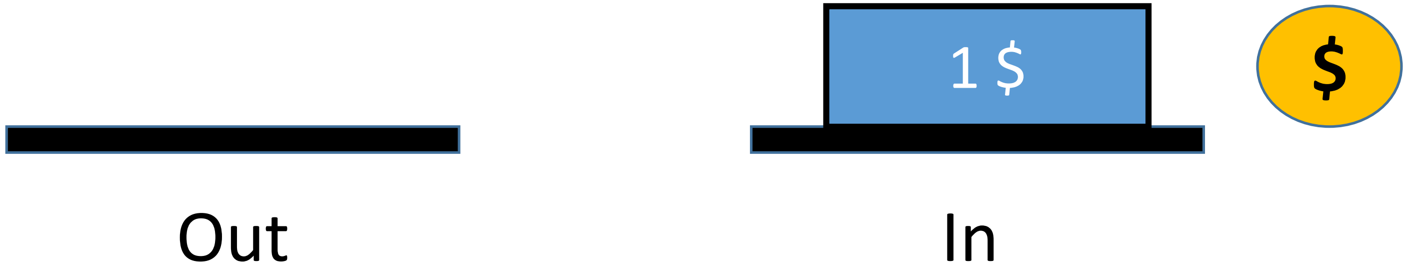
a(opi) = t(opi) + O(1) x (added i – removed i)

There are not any real credits anywhere. They are just an accounting trick.

The Banker method – Two-stack queue

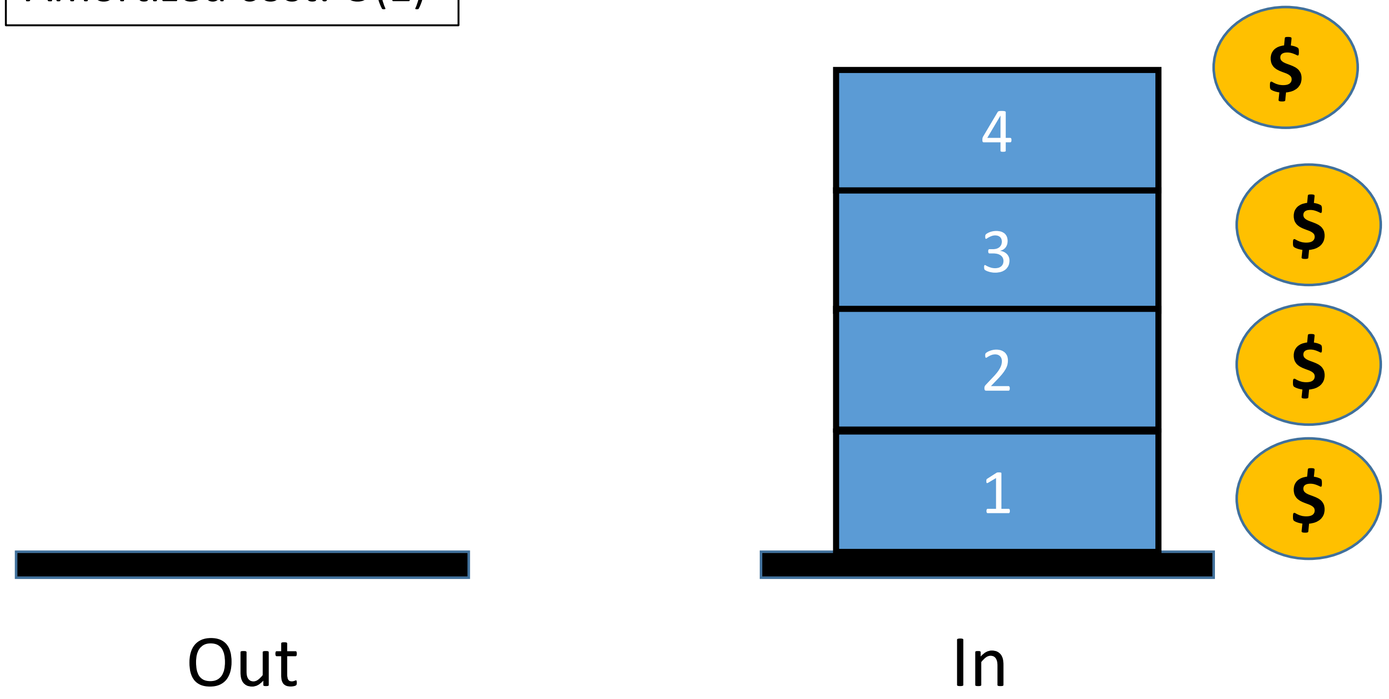
Actual Work: O(1)
Credit added: 1
Amortized cost: O(1)

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



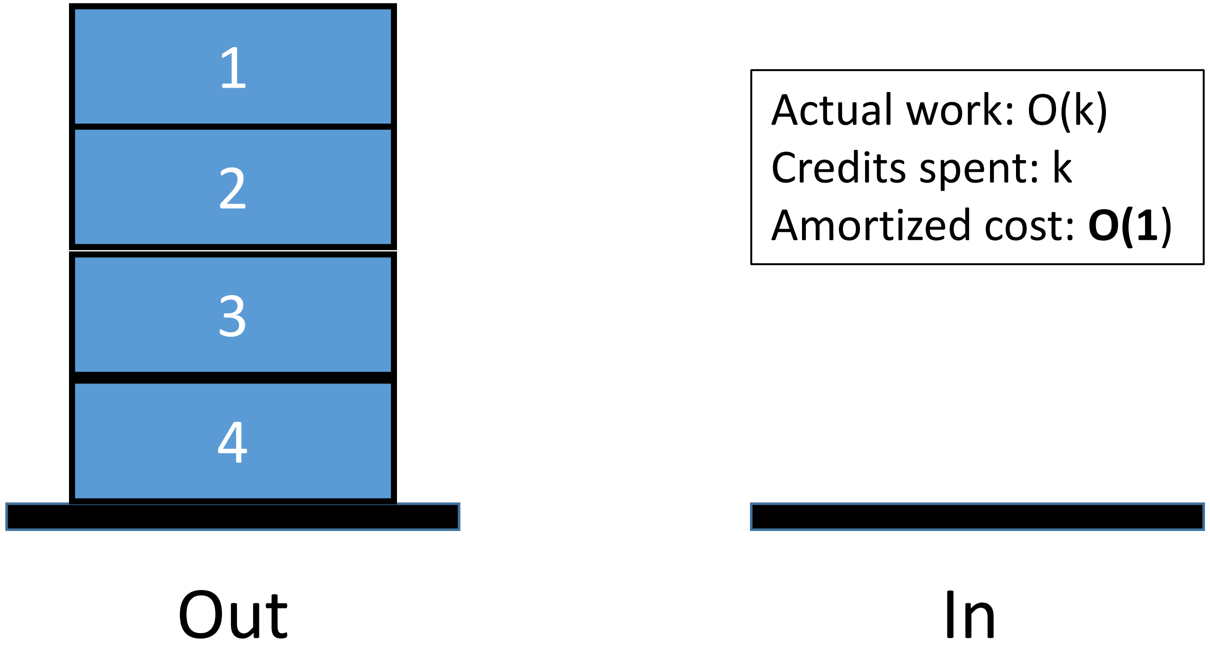
The Banker method – Two-stack queue

Actual Work: O(1)
Credit 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 Banker method – Two-stack queue



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

Use 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 will place down credits, and compute the amortized cost of operations that place credits this way.
- Indicate where you will spend credits, and **justify why the credits you intend to spend are guaranteed to be there.** Then, computer the amortized cost of each operation that spends credits this way.

An Observation

- The amortized cost of an operation is:

a(opi) = t(opi) + O(1) x (added I – removed i)

- Equivalently, this is

a(opi) = t(opi) + O(1) x Δ Credits

- Some observations:

- It does not 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.

- Actual work: O(k)
- Credits spent: k
- Amortized cost: **O(1**)

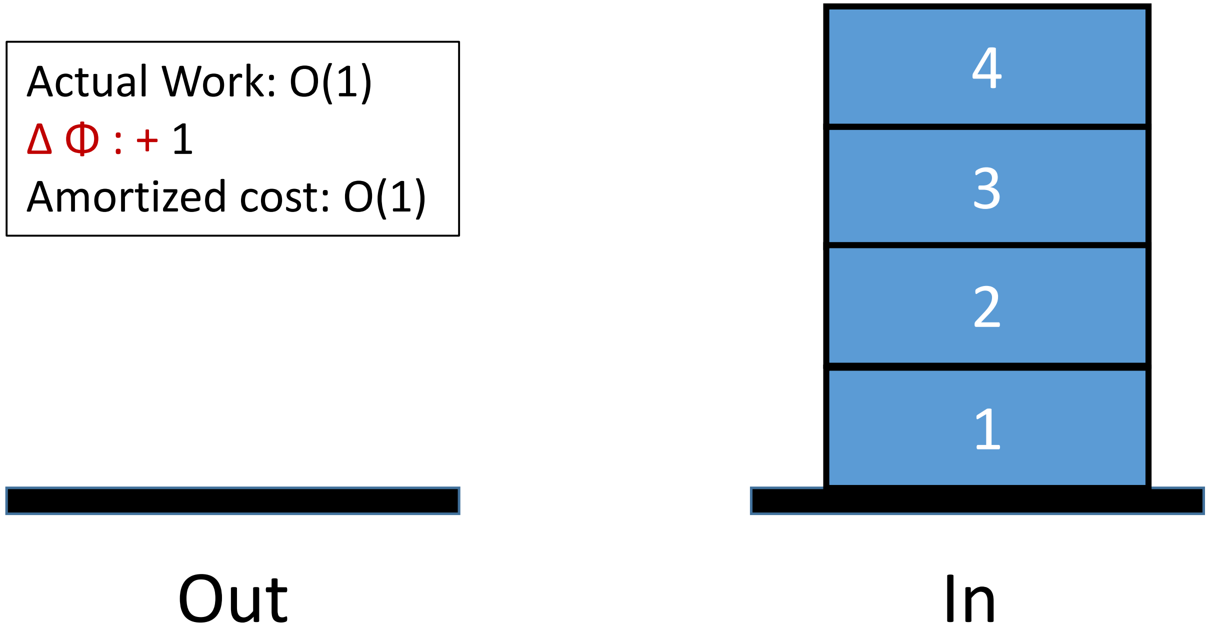
The Potential Method

- In the potential method, we define a potential function Φ that maps a data structure to a non-negative real value.
- Define $a(opi)$ as $a(opi) = t(opi) + O(1)\ x\ \Delta \Phi i$
- The $\Delta \Phi i$ is the change in the value of Φ during execution of operation opi.

The Potential Method – Two-Stack Queue

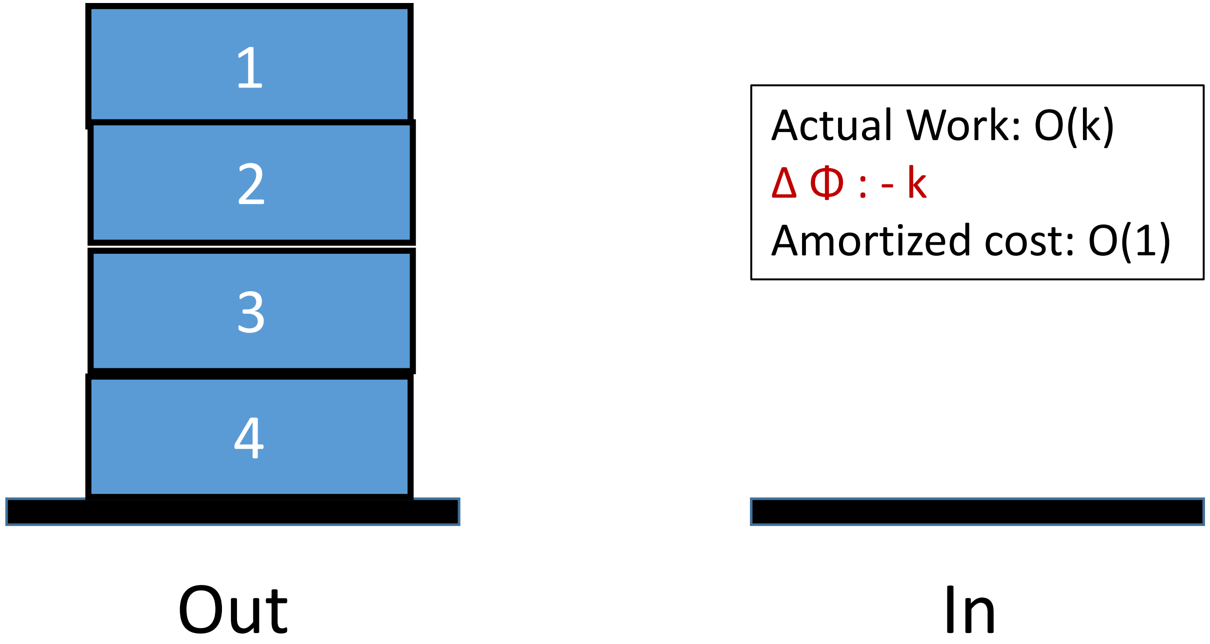
Φ = height of In Stack

Actual Work: O(1)
Δ Φ : + 1
Amortized cost: O(1)



The Potential Method – Two-Stack Queue

Φ = height of In Stack



Actual Work: O(k)

Δ Φ : - k

Amortized cost: O(1)

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 is initially zero or otherwise account for a nonzero start potential.

For each operation, determine its $\Delta\ \Phi$

Compute the amortized costs of each operation.

What We learned 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.

Amortized Complexity – Main idea

- Worst case analysis of run time complexity is often too pessimistic. Average case analysis may be difficult because (1) it is not clear what is “average data”, (2) uniformly random data is usually not average, (3) probabilistic arguments can be difficult.

- Amortized complexity analysis is a different way to estimate run times. The main ideas is to spread out the cost of operations, charging more than necessary when they are cheap — thereby “saving up for later use” when they occasionally become expensive.

The End!