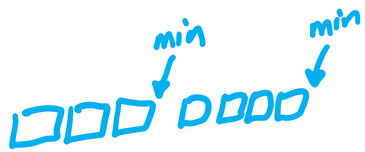
For advanced data structures like a Python list or a Priority Queue,

- We should care about the aggregate cost of a sequence of operations
- This might not be as bad as the per-operation worst cases suggest

TODAY

- Amortized costs and potential functions are a handy way to reason about aggregate costs let n = #items

```
class MinList<T>:
    def append(T value):
        # append a new value
    def flush():
        # empty the list
    def foreach(f):
        # do f(x) for each item
    def T min():
        # return the smallest
        # (without removing it)
```



Stage 0

- Use a linked list
- `min` iterates over the entire list

append is $\Theta(1)$
min is $\Theta(n)$

Stage 1

- Use a linked list
- `min` caches its result, so that next time it only needs to iterate over newer values

min is $\Theta(n)$
in the worst case

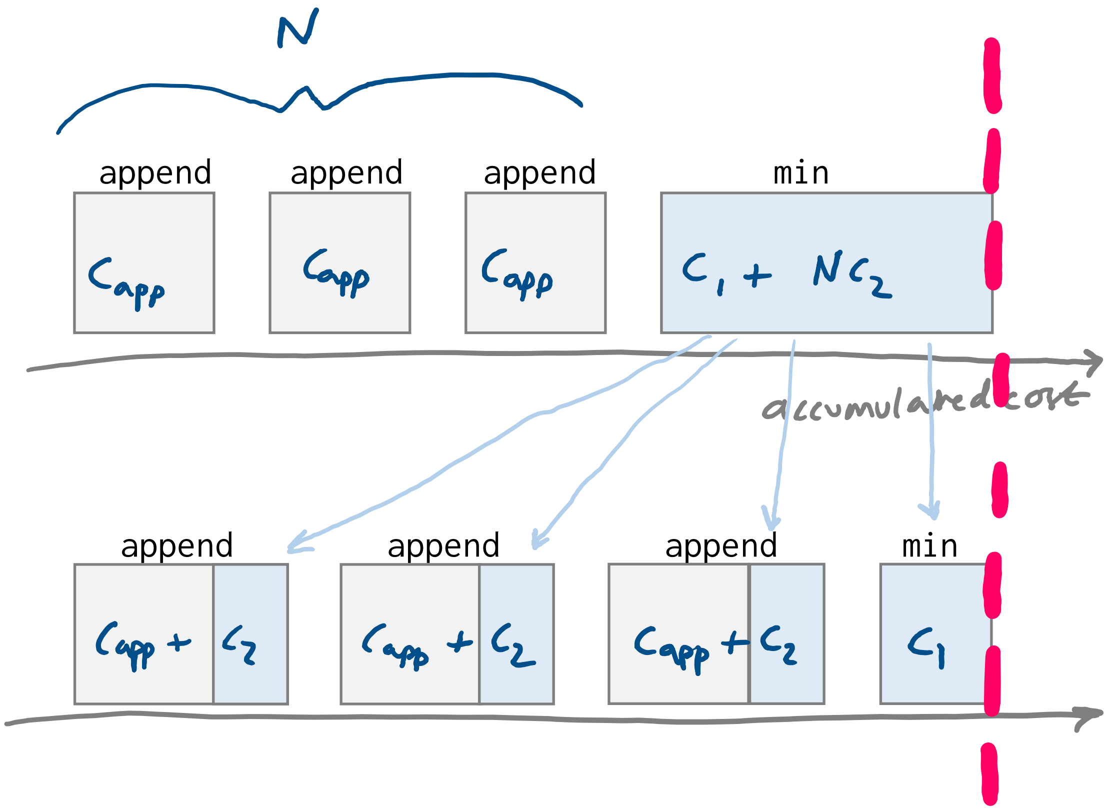
Stage 2

- Use a linked list
- Store the current minimum, and update it on every append

append is $\Theta(1)$
min is $\Theta(1)$

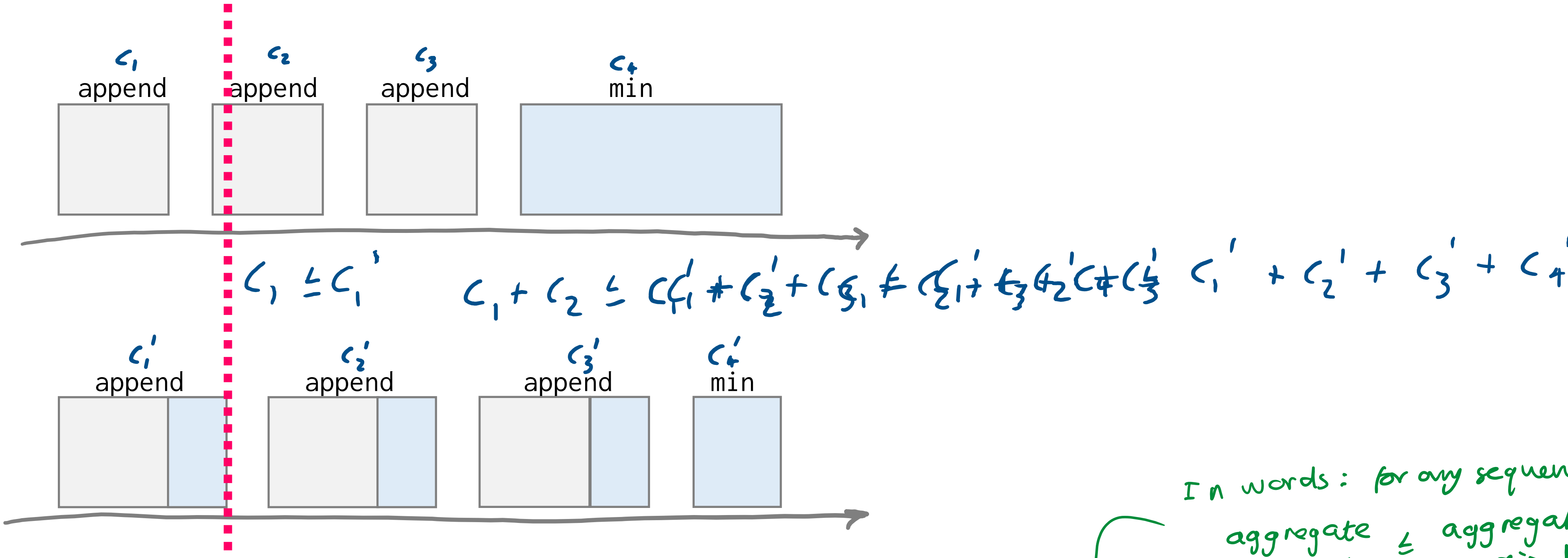
Stage 3

- `min` caches its result, the same as Stage 1
- ... but we argue it's just as good as Stage 2



Stage 3

- `min` caches its result, the same as Stage 1
- ... but we argue it's just as good as Stage 2



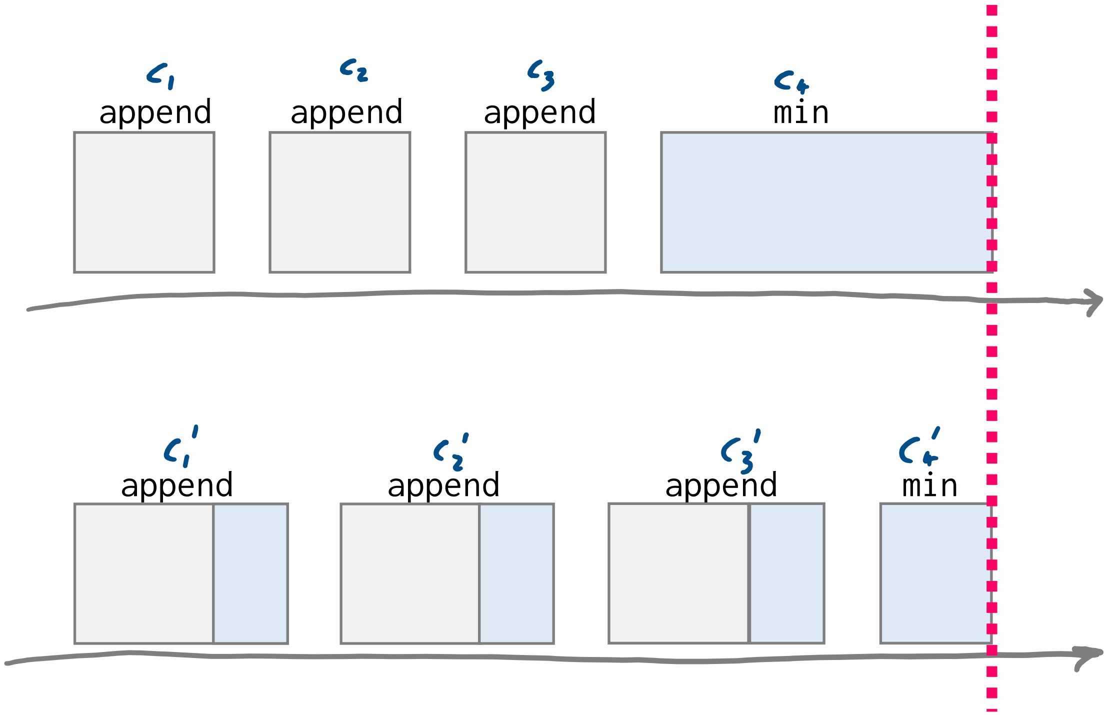
In words: for any sequence of operations, aggregate true cost ≤ aggregate amortized cost.

We'd like to be able to reason about aggregate true costs. If someone clever can think up amortized costs for us, it makes it easy to do so.

FUNDAMENTAL INEQUALITY OF AMORTIZATION

Let there be a sequence of m operations, applied to an initially-empty data structure, whose true costs are $c_1, c_2, \ldots, c_m$. Suppose someone invents $c_1', c_2', \ldots, c_m'$. These are called **amortized costs** if

$$c_1 + \cdots + c_j \le c_1' + \cdots + c_j' \quad \text{for all} \quad j \le m$$

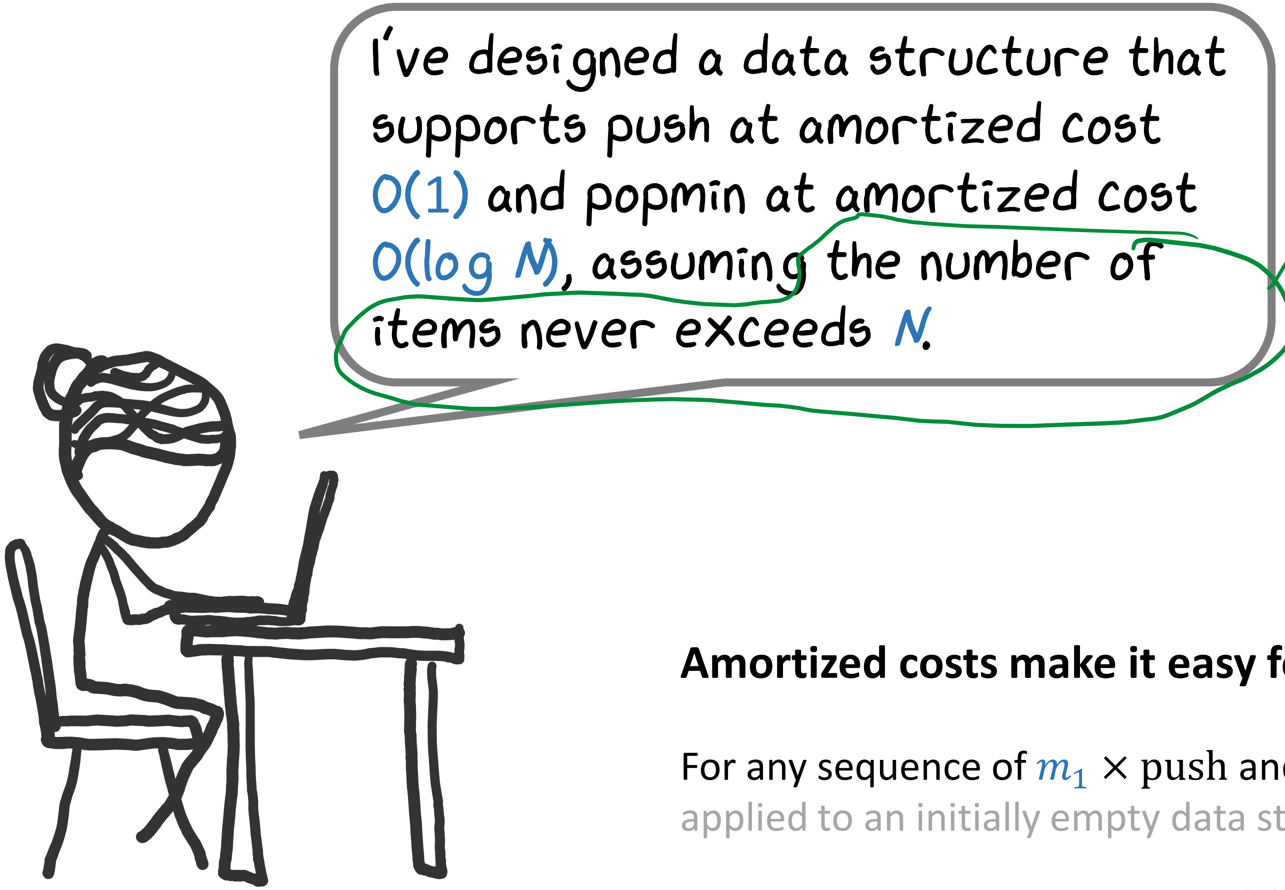


Ex. sheet 6 q.6 asks you to think through why this is a sensible restriction

FUNDAMENTAL INEQUALITY OF AMORTIZATION

Let there be a sequence of m operations, applied to an initially-empty data structure, whose true costs are $c_1, c_2, \ldots, c_m$. Suppose someone invents $c_1', c_2', \ldots, c_m'$. These are called **amortized costs** if

$$c_1 + \cdots + c_j \leq c_1' + \cdots + c_j' \quad \text{for all} \quad j \leq m$$

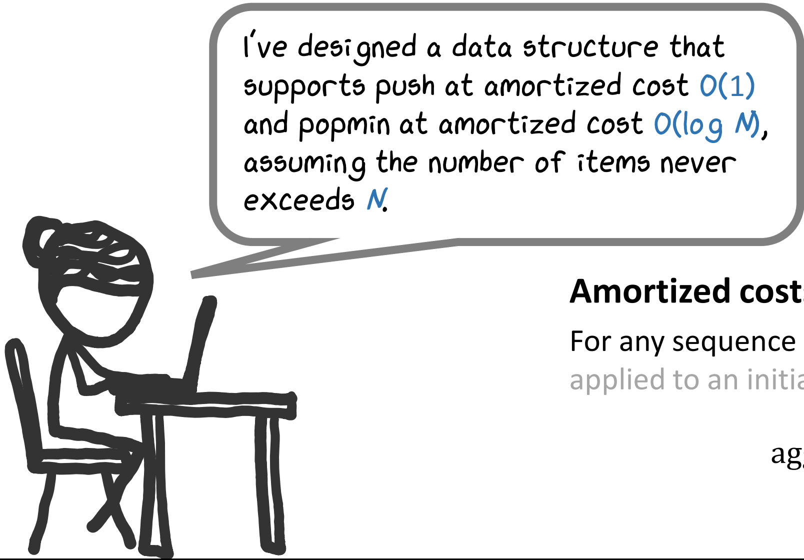


we want to talk about aggregate costs of sequences of operations, involving a mix of push and popmin. Thus # items fluctuates over the course of these operations. That's why the bound involves this circuitous language about "never exceeds N".

Amortized costs make it easy for the user to reason about aggregate costs.

For any sequence of $m_1 \times$ push and $m_2 \times$ popmin,
applied to an initially empty data structure,

$$\text{aggregate cost} \leq m_1\, O(1) + m_2\, O(\log N) = O(m_1 + m_2 \log N)$$



Amortized costs make it easy for the user to reason about aggregate costs.

For any sequence of $m_1 \times$ push and $m_2 \times$ popmin,
applied to an initially empty data structure,

$$\text{aggregate cost} \leq m_1\, O(1) + m_2\, O(\log N) = O(m_1 + m_2 \log N)$$

When we've told "push has amortized cost O(1), popmin has amortized cost O(log N)", this means:

- each push/popmin call might have its own slightly different amortized cost, but nonetheless
  $\exists\, K > 0, N_0$ s.t. $\forall\, N \geq N_0$ every single popmin's amortized cost is $\leq K \log N$.
  every single push's amortized cost is $\leq K$
  in any sequence of ops (starting from empty) where # items is always $\leq N$.

Then, by the Fundamental Inequality, for any sequence of ops (starting from empty)

$$\begin{matrix}\text{aggregate} \\ \text{true cost} \\ \text{of these} \\ \text{operations}\end{matrix} \leq \begin{matrix}\text{aggregate} \\ \text{amortized cost} \\ \text{of these} \\ \text{operations}\end{matrix} \leq m_1 K + m_2 K \log N$$

where $m_1 = \#$ push, $m_2 = \#$ popmin.

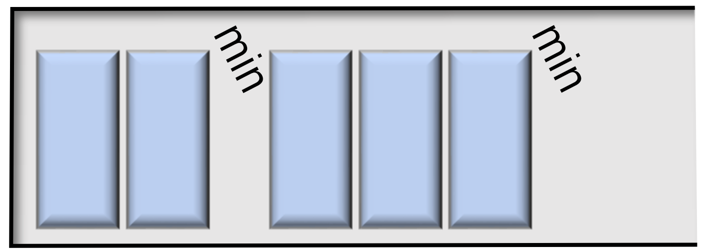
SECTION 7.4

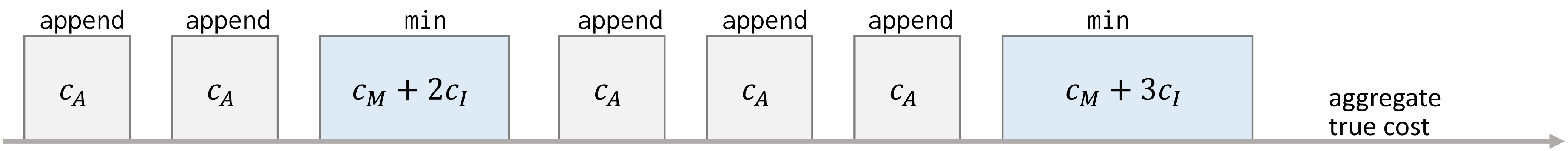
Potential functions

or, how on earth do we come up with useful amortized costs?

- Suppose we can store 'credit' in the data structure, and operations can either store or release credit
- Let the 'accounting' cost of an operation be: $\begin{pmatrix}\text{accounting}\\ \text{cost}\end{pmatrix} = \begin{pmatrix}\text{true}\\ \text{cost}\end{pmatrix} + \begin{pmatrix}\text{credit}\\ \text{it stores}\end{pmatrix} - \begin{pmatrix}\text{credit}\\ \text{it releases}\end{pmatrix}$
- Let's 'pay ahead' for the potentially-costly operations


```
class MinList<T>:
    def append(T value):
        # append a new value
    def T min():
        # caches the result, so we
        # only need to iterate over
        # newly-appended items
```



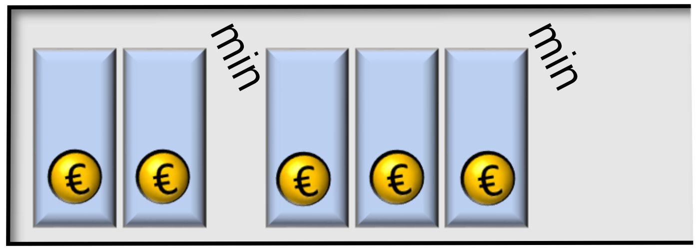


- Suppose we can store ‘credit’ in the data structure, and operations can either store or release credit
- Let the ‘accounting’ cost of an operation be: $\begin{pmatrix}\text{accounting}\\ \text{cost}\end{pmatrix} = \begin{pmatrix}\text{true}\\ \text{cost}\end{pmatrix} + \begin{pmatrix}\text{credit}\\ \text{it stores}\end{pmatrix} - \begin{pmatrix}\text{credit}\\ \text{it releases}\end{pmatrix}$
- Let’s ‘pay ahead’ for the potentially-costly operations


```
class MinList<T>:
    def append(T value):
        # append a new value
    def T min():
        # caches the result, so we
        # only need to iterate over
        # newly-appended items
```

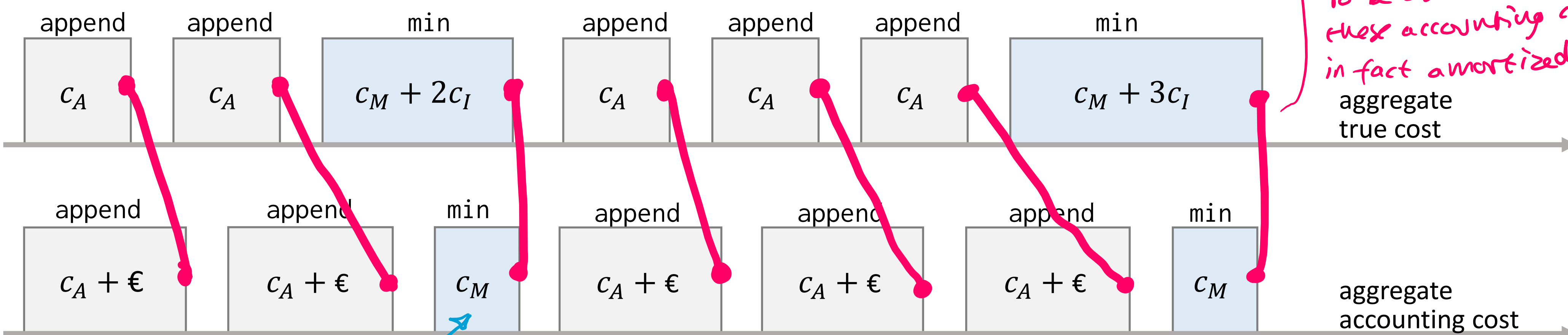
and store 1€ of credit

and release all credit



For all subsequences (starting from empty), agg. true cost ≤ agg. accounting cost.

We need this, in order to be able to claim that these accounting costs are in fact amortized costs.



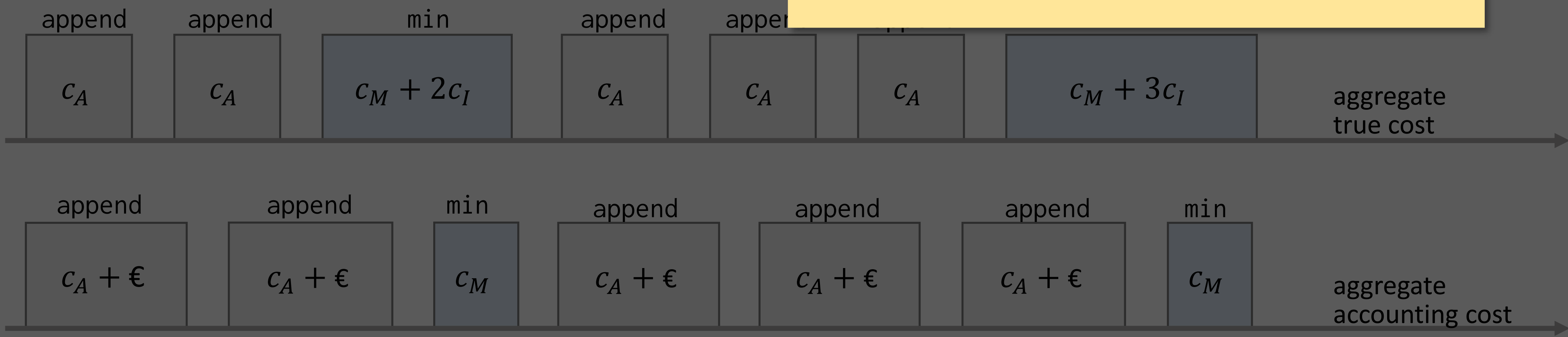
accounting cost = $c_M + 2c_I + 0 - 2€$. Let's set $1€ = c_I$, so acc. cost $= c_M$.

- Suppose we can store ‘credit’ in the data structure, and operations can either store or release credit
- Let the ‘accounting’ cost of an operation be: $\binom{\text{accounting}}{\text{cost}} = \binom{\text{true}}{\text{cost}} + \binom{\text{credit}}{\text{it stores}} - \binom{\text{credit}}{\text{it releases}}$
- Let’s ‘pay ahead’ for the potentially-costly operations


```
class MinList<T>:
    def append(T value):
        # append a new value
    def T min():
        # caches the result, so we
        # only need to iterate over
        # newly-appended items
```

THEOREM. These are valid amortized costs i.e. for any sequence of operations on an initially-empty data structure

$$\begin{array}{c}\text{aggregate}\\ \text{true}\\ \text{cost}\end{array} \le \begin{array}{c}\text{aggregate}\\ \text{amortized}\\ \text{cost}\end{array}$$



Let Ω be the set of all states our data structure might be in.
A function $\Phi : \Omega \to \mathbb{R}$ is called a **potential function** if

$$\Phi(\mathcal{S}) \geq 0 \quad \text{for all} \quad \mathcal{S} \in \Omega$$
$$\Phi(\text{empty}) = 0$$

Φ = bank balance
= total credit stored in data structure

For an operation $\underbrace{\mathcal{S}_{\text{ante}}}_{\text{state before}} \to \underbrace{\mathcal{S}_{\text{post}}}_{\text{state after}}$ with true cost c, define the **accounting cost** to be

$$c' = c + \Phi(\mathcal{S}_{\text{post}}) - \Phi(\mathcal{S}_{\text{ante}}) \quad = c + \Delta\Phi$$

THE 'POTENTIAL' THEOREM: These are valid amortized costs.

PROOF: Consider an arbitrary sequence of operations, starting from empty: $\underbrace{\mathcal{S}_0}_{=\text{empty}} \xrightarrow{c_1} \mathcal{S}_1 \xrightarrow{c_2} \mathcal{S}_2 \to \cdots \xrightarrow{c_m} \mathcal{S}_m$

$$\text{aggregate accounting cost} = c_1' + c_2' + \cdots + c_m'$$

$$= -\Phi(\mathcal{S}_0) + c_1 + \cancel{\Phi(\mathcal{S}_1)} - \cancel{\Phi(\mathcal{S}_1)} + c_2 + \cancel{\Phi(\mathcal{S}_2)} \cdots - \cancel{\Phi(\mathcal{S}_{m-1})} + c_m + \Phi(\mathcal{S}_m)$$

telescopic sum

$$= -\underbrace{\cancel{\Phi(\mathcal{S}_0)}}_{=0 \text{ by defn. of } \Phi} + c_1 + \cdots + c_m + \underbrace{\Phi(\mathcal{S}_m)}_{\geq 0 \text{ by defn. of } \Phi}$$

This is why we need the fussy language about "sequence of operations applied to an initially empty data structure".

$$\geq c_1 + \cdots + c_m = \text{aggregate true cost}$$

EXAMPLE: DYNAMIC ARRAY

A Python list is implemented as a dynamically-sized arrays. It starts with capacity 1, and doubles its capacity whenever it becomes full.

Suppose the cost of writing an item is 1, and the cost of doubling capacity from m to $2m$ (and copying across the existing items) is κm.

Show that the amortized cost of append is $O(1)$.

Let's set Φ = # newly added items since last doubling × €

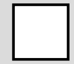
$c' = c + \Delta\Phi$

Let's set 1 € = 2κ. (This way, $\Delta\Phi$ "pays off" for the variable amount of work involved in doubling.)

Observe that the amortized costs are slightly different (some 1+κ, some 1+2κ), but in all cases

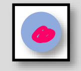
amortized cost of append $\leq 1 + 2\kappa$

initially empty



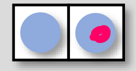
$\Phi = 0$

append() $c = 1$ $c' = 1 + € = 1 + 2\kappa$



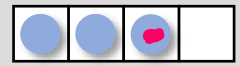
$\Phi = €$

append(), requires doubling $c = \kappa + 1$ $c' = \kappa + 1 = 1 + \kappa$



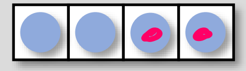
$\Phi = €$

append(), requires doubling $c = 2\kappa + 1$ $c' = 2\kappa + 1 - € = 1 + 2\kappa$



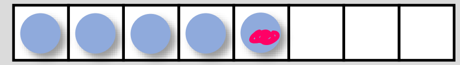
$\Phi = €$

append() $c = 1$ $c' = 1 + € = 1 + 2\kappa$



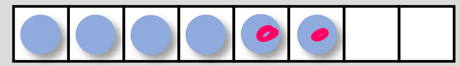
$\Phi = 2€$

append(), requires doubling $c = 4\kappa + 1$ $c' = 4\kappa + 1 - € = 1 + 2\kappa$



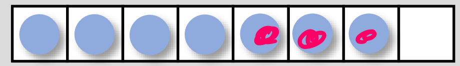
$\Phi = €$

append() $c = 1$ $c' = 1 + € = 1 + 2\kappa$



$\Phi = 2€$

append() $c = 1$ $c' = 1 + € = 1 + 2\kappa$



$\Phi = 3€$

In other words, am. cost of append is $O(1)$, asymptotic in N = # items in array,

$\exists\, \kappa' > 0,\ N_0 .\ \forall N \geq N_0$ am. cost of append on array with N items $\leq \kappa'$.

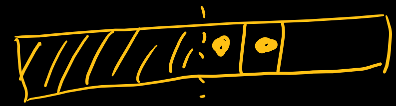
EXAMPLE: DYNAMIC ARRAY (sloppy style)

A Python list is implemented as a dynamically-sized arrays. It starts with capacity 1, and doubles its capacity whenever it becomes full.

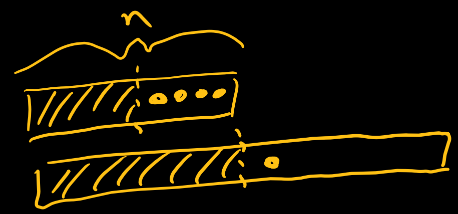
Suppose the cost of writing an item is $O(1)$, and the cost of doubling capacity from m to $2m$ (and copying across the existing items) is $O(m)$.

Show that the amortized cost of append is $O(1)$.

let $\Phi = 2 \times \#$ items in array $-$ size of array.



There are two ways that append() might play out.



- If the array size has to double from n to $2n$,

  $c = O(n) + O(1)$ $\qquad \Phi_{ante} = 2n - n$ $\qquad \Phi_{post} = 2(n+1) - 2n$ $\qquad \Delta\Phi = 2 - n$

  $\Rightarrow c' = c + \Delta\Phi = \cancel{O(n)} + O(1) + 2 - \cancel{n} = O(1)$ [what we really mean here is: define $\Phi = (2n - \text{size}) \times \epsilon$, and set the exchange rate equal to the constant hidden inside $O(n)$. Then, $-n\epsilon$ really does cancel out $O(n)$.]

- If the array doesn't need doubling,

  $c = O(1)$ $\qquad \Delta\Phi = 2$ $\qquad \Rightarrow c' = c + \Delta\Phi = O(1) + 2 = O(1)$

In both cases, am. cost is $O(1)$.

[Technically, this Φ isn't a potential function. A potential function must be ≥ 0, and $= 0$ when empty, but $\Phi(\text{empty}) = -1$. We can create a proper potential function Φ' that's like Φ except at empty. This changes some of the amortized costs, but only finitely many, so big-O results remain true, just with a possibly-larger K.]