Amortized Analysis	Average-Case Analysis
No probability	May involve prob.
Obtain average performance in the worst-case	Average performance
No bad sequences	Possible bad sequences

Techniques for Amortized Analysis

- Aggregate analysis
- Accounting method
- Potential method.

What we did in disjoint sets was aggregate analysis.

Example: Stack with an added operation.

Push(S, x) : Push x onto stack

Pop (S) : Pop top of stack and return popped object

Multipop (S, k) : Remove $\min(|S|, k)$ objects from top.

Multipop (S, k)
  while not stack-empty (S) and $k \neq 0$
    do Pop (S)
      $k \leftarrow k-1$

Running time of Multipop is $O(\min(|S|, k))$
which means in the worst-case it's $O(n)$

Therefore, a sequence of n operations takes $O(n^2)$ time.

Aggregate Analysis

Get a better bound by considering the entire sequence of n operations.

Claim: Any sequence of n stack operations take $O(n)$ time.

- Any object can be popped at most once after it's pushed. The # times a pop is called on a non-empty stack (including those in a multipop) is at most equal to # pushes.
- Given n operations that result in m pops from within multipop, the running time is $O(n+m)$

But $m \leq n$, so $O(2n) = O(n)$.

Each operation runs in $O(1)$ amortized time

Aggregate: "Treat every operation the same way in terms of time"

Another example of aggregate analysis:

In the book: Incrementing a binary counter n times.

n bits : $\xleftrightarrow{\quad n \quad}$

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0
0 0 0 0 0 0 1 1
0 0 0 0 0 1 0 0
        ⋮
```

An increment takes $O(n)$ time

Incrementing n times $\Rightarrow O(n^2)$ time

A sequence of n increment operations take $O(n)$ time.

So each operation takes $O(1)$ amortized time.

$$\#\text{flips} : \underset{1^{st}\text{ bit}}{n} + \underset{2^{nd}\text{ bit}}{\frac{n}{2}} + \frac{n}{4} + \frac{n}{8} + \cdots\cdots \leq 2n$$

Accounting method (each op. has a different amortized time)

- Assign an amortized cost $\hat{c}_i$ for operation i
- Let c_i be actual cost of operation i.
- If $\hat{c}_i > c_i$, add $\hat{c}_i - c_i$ to credit (store that much on some object)
- If $\hat{c}_i < c_i$, subtract $c_i - \hat{c}_i$ from credit. (use that much from stored)

As long as credit is always ≥ 0, we have $\sum_i (\hat{c}_i - c_i) \geq 0$

So $\sum_i c_i \leq \sum_i \hat{c}_i$

credit must always be non-negative

Stack:

	$\hat{c}$	c	
Push(S, x)	2	1	put a credit of 1 on pushed obj.
Pop(S)	0	1	use credit placed on object
Multipop(S, k)	0	$\min(\lvert S\rvert, k)$	same as above

- Given n stack operations, $\sum_{i=1}^{n} \hat{c}_i \leq 2n = O(n)$
- If credit is always ≥ 0, then $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i = O(n)$.
- Is it?

  Total credit is always equal to # objects in stack, and that's ≥ 0.

Potential method

- Same as "credit" but associated with data structure as a whole
- Let D_i be the data structure after the i^{th} operation. (Initially D_0)
- Define a "Potential function" $\phi(D)$ such that

$$\phi(D_i) \geqslant 0$$

$$\phi(D_0) = 0$$

- $\phi(D)$ measures how "difficult" the data structure D is
- Let amortized cost of operation i be

$$\hat{c}_i = c_i + \underbrace{\phi(D_i) - \phi(D_{i-1})}_{\Delta\phi_i}$$

- $$\sum \hat{c}_i = \sum c_i + \sum \phi(D_i) - \phi(D_{i-1}) = \sum c_i + \big(\cancel{\phi(D_1)} - \phi(D_0) + \cancel{\phi(D_2)} - \cancel{\phi(D_1)} + \phi(D_3) - \cancel{\phi(D_2)} \cdots \big)$$

$$= \sum c_i + \phi(D_n) - \phi(D_0) \geqslant \sum c_i .$$

Idea: An operation i with high c_i might have

$$\Phi(D_i) - \Phi(D_{i-1}) < 0$$

which makes the structure easier for later operations.

Stack: Define $\Phi(S_i)$ = size of stack after ith operation.

Observe $\Phi(S_i) \geq 0$ and $\Phi(S_0) = 0$.

Push(S, x): $\hat{c}_i = c_i + \Phi(S_i) - \Phi(S_{i-1}) = 1 + 1 = 2$

Pop(S): $\hat{c}_i = c_i + \Phi(S_i) - \Phi(S_{i-1}) = 1 - 1 = 0$

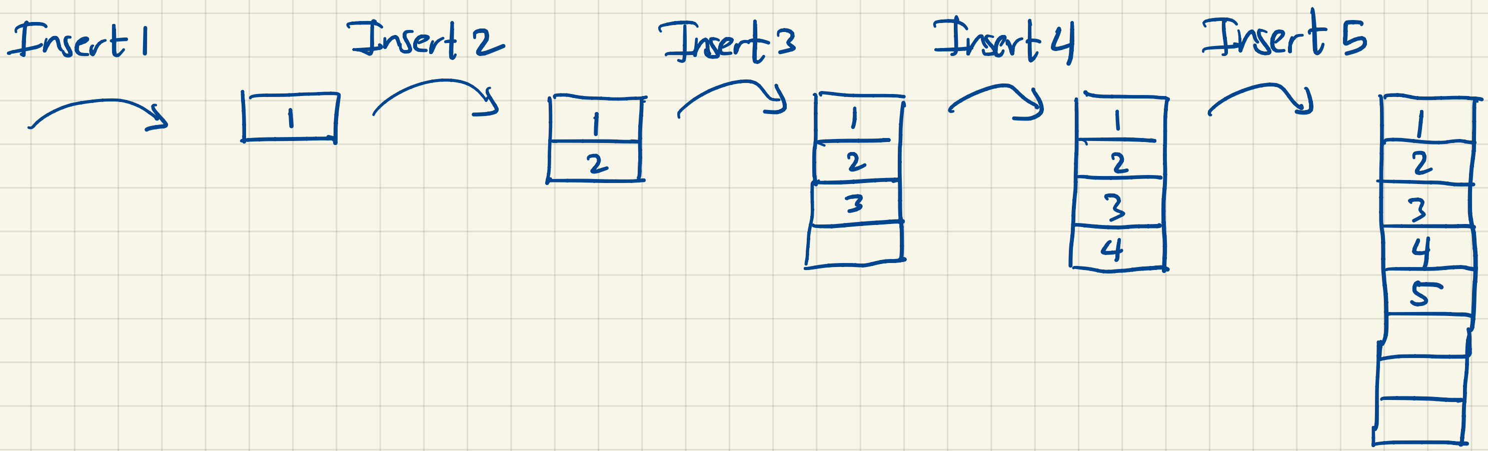
Multipop(S, k): $\hat{c}_i = c_i + \Phi(S_i) - \Phi(S_{i-1}) = \min(|S_{i-1}|, k) - \min(|S_{i-1}|, k) = 0$

A sequence of n stack operations have amortized cost $\leq 2n$.

Another example: Dynamic tables.

- Insert objects in table.
- Start with table size 0
- To insert, if size $=0$, make it 1.
- When no more space, double size of table $\underbrace{\text{(and copy entire table)}}_{O(\text{size table})}$

Example:



Naive analysis: In the worst case, each insert must copy, so quadratic time?

Aggregate analysis: In a sequence of n inserts, the ith insert causes a copy of $(i-1)$ elements if $(i-1)$ is a power of 2.

So n operations take

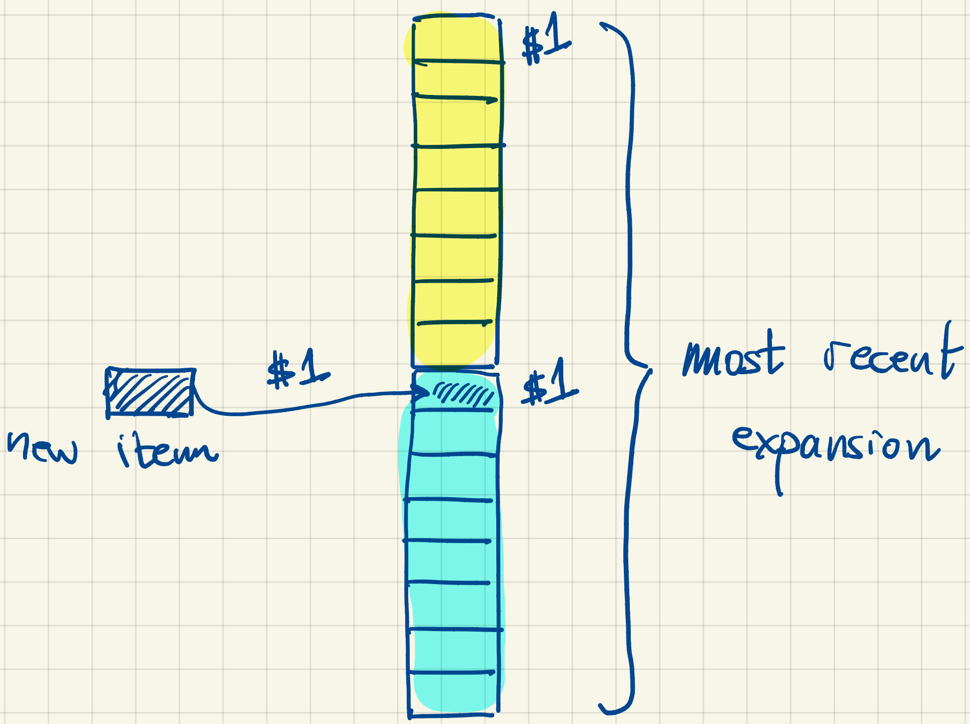
$$\sum_{i=1}^{n} c_i = \underbrace{n}_{\text{insert}} + \underbrace{\sum_{j=0}^{\lfloor \lg n \rfloor} 2^j}_{\text{copying}} = n + \underbrace{1+2+4+\cdots+2^{\lfloor \lg n \rfloor}}_{< n + \frac{n}{2} + \frac{n}{4} + \cdots = n\left(1+\frac{1}{2}+\frac{1}{4}+\cdots\right)} < n + 2n = 3n = O(n)$$

Accounting method: charge each operation i, $\hat{c}_i = 3$

- use 1 for actual operation
- store 2 as credit on two objects

copy yourself in future   copy another object

Illustration of accounting method



By the time table is full again, we would have already payed to move all elements.

Potential method:

Define $\phi(T_i) = 2\,num_i - size_i$

num_i = # elements after i^{th} insert

$size_i$ = size of table after i^{th} insert

Observe: $\phi(T_0) = 2 \times 0 - 0 = 0$

$\phi(T_i) \geq 0$ always since $num_i \geq \frac{size_i}{2}$

(table always at least $\frac{1}{2}$ full)

Insert (No expansion):

$$\hat{c}_i = c_i + \phi(T_i) - \phi(T_{i-1}) = 1 + (2num_i - size_i) - (2num_{i-1} - size_{i-1})$$

$$= 1 + 2(num_i - num_{i-1}) - (size_i - size_{i-1})$$

$$= 1 + 2 \times 1 - 0 = 3$$

Insert (with expansion):

$$\hat{c}_i = c_i + \phi(T_i) - \phi(T_{i-1}) = \underbrace{num_i}_{1+\#\text{copies}} + (2num_i - size_i) - (2num_{i-1} - size_{i-1})$$

$$= num_i + 2(num_i - num_{i-1}) - (size_i - size_{i-1})$$

Expansion means: $size_i = 2(num_{i-1})$ (Doubled)

$size_{i-1} = num_{i-1}$ (Full)

$$\hat{c}_i = num_i + 2 - num_{i-1} = 2 + num_i - num_{i-1} = 2 + 1 = 3.$$

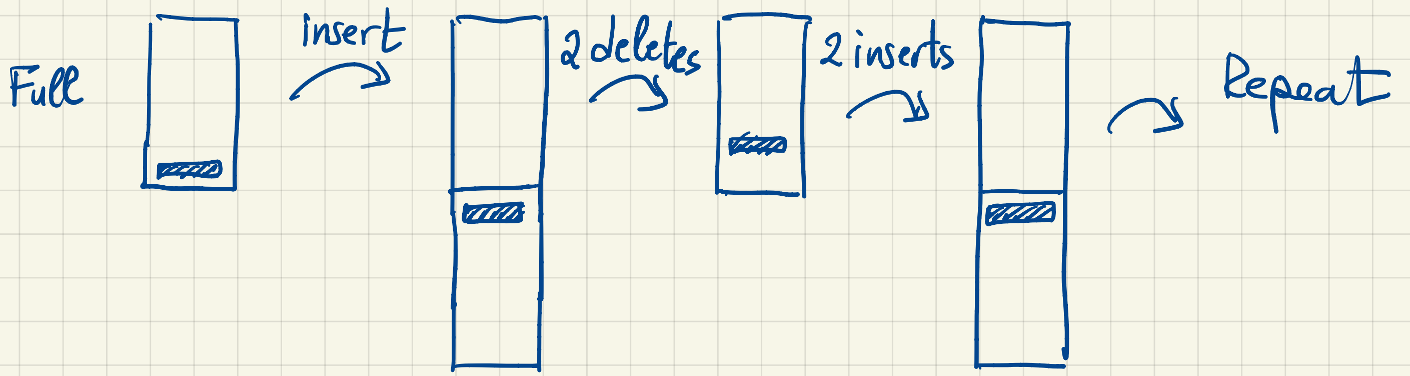
What about deletions?

Simple strategy:

table full $\Rightarrow$ double it upon insert as before

table $< \frac{1}{2}$ full $\Rightarrow$ halve it after delete

Does not work:



$O(n)$ time every 2 operations on average $\Rightarrow O(n)$ time / op.

Idea: Allow table size to drop below half full, e.g. use $\frac{1}{4}$