Amortized Analysis

How large should a hash table be?

- As large as possible (time)
- As small as possible (space)
- [redacted] for n items $n = O(m)$ $\alpha = \frac{n}{m} \Rightarrow \alpha = O(1)$

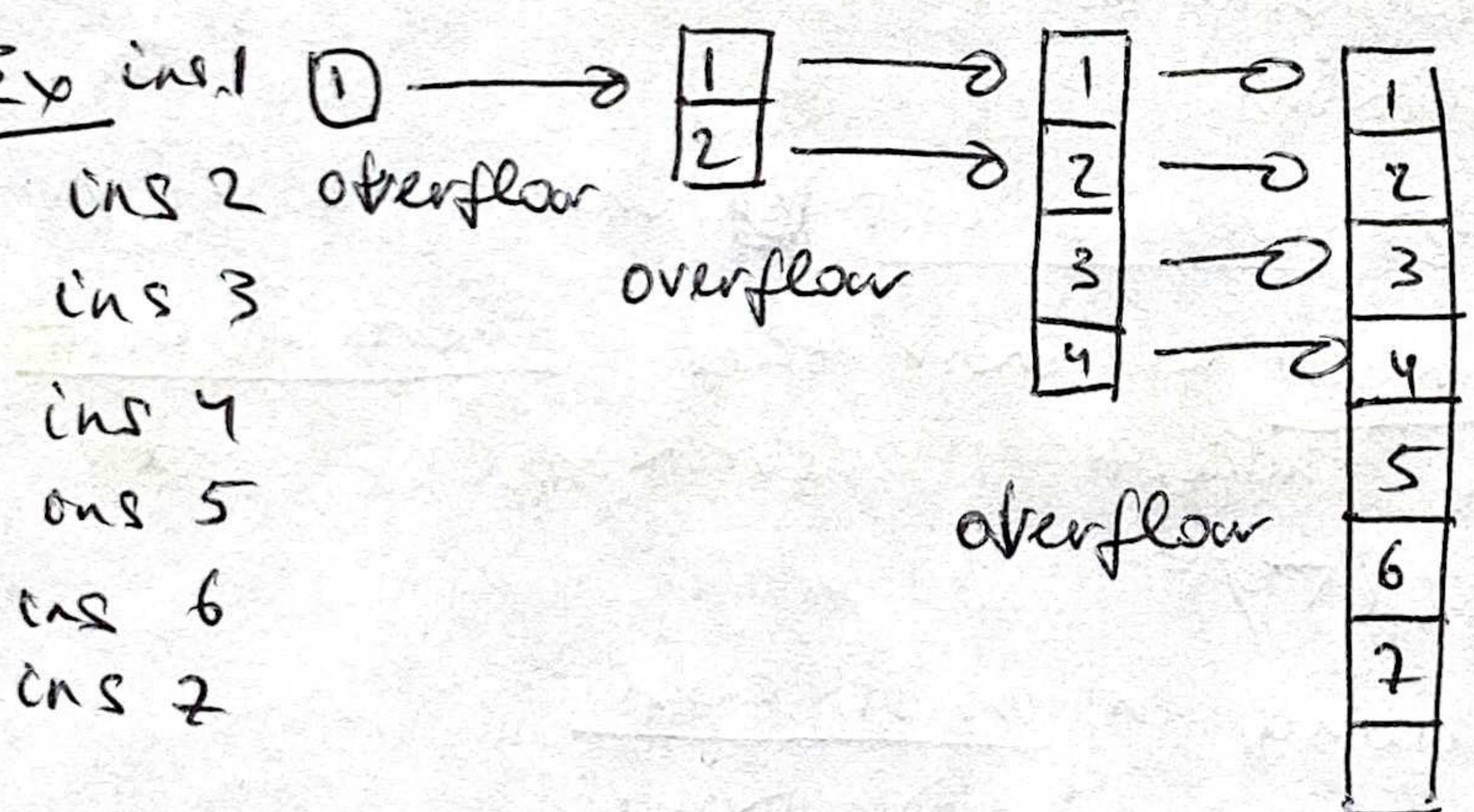
my comment: m must grow otherwise n depletes any O(1) difference.

Problem: what if we do not know in advance?

Solution: Dynamic tables

Idea: Whenever the table gets too full (overflows), "grow" it.

1) Allocate (malloc or new) a larger table
2) Move items from old table to new
3) Free old table

Ex


each time a table overflows create a new table of 2x size.

Analysis (aggregate)

Sequence of Insert operations.

worst case cost of 1 Insert $= \Theta(n)$

worst case cost of n Inserts $= n\,\Theta(n) = \Theta(n^2)$ WRONG!

n Inserts take $\Theta(n)$ time in worst case

Let c_i = cost of ith Insert

$$= \begin{cases} i & \text{if } i-1 \text{ is power of } 2 \\ 1 & \text{otherwise} \end{cases}$$

i	1	2	3	4	5	6	7	8	9	10	
$size_i$	1	2	4	4	8	8	8	8	16	16	
c_i	1	2	3	1	5	1	1	1	9	1	
c_i {	1	1	1	1	1	1	1	1	1	1	insert
		1	2		4				8		copy
$\hat{c}_i$	2	3	3	3	3	3	3	3	3	3	
$bank_i$	1	2	2	4	2	4	6	8	2	4	

↑ undercharge the first

my comment

[redacted]

Cost of n Inserts $= \sum_{i=1}^{n} c_i = \underbrace{n}_{\text{insert}} + \underbrace{\sum_{j=0}^{\lfloor lg(n-1) \rfloor} 2^j}_{\text{copy}}$

$$\frac{(1 - 2^{lg(n-1)+1})}{1-2} = 2\,2^{lg(n-1)} - 1 = 2n - 3$$

$= \cancel{4}\, n + 2n - 3 \le 3n = \Theta(n)$

Thus, average cost of Insert

$= \frac{\Theta(n)}{n} = \Theta(1)$ amortized the occasional cost of copying over previous insertions,

↳ average cost [redacted] per Insert is $\Theta(1)$

Amortized analysis:

Analyze a sequence of operations to show that average cost per operation is small, even though 1 operation may be expensive. NO PROBABILITY.

- average performance in worst case
- cost amortized over n operations
- worst case bound but over a sequence of operations

Types of amortized arguments.

- aggregate (above)
- accounting } more precise → allocate specific amort cost to each operation
- potential

Accounting method:

- Charge ith operation a fictitious amortized cost $\hat{c}_i$ ($1 pays for 1 unit of work)
- Fee is consumed to perform operation
- unused amount stored in "bank" for use by later operations.

- Bank balance must not go negative.

Must have $\underbrace{\sum_{i=1}^{n} c_i}_{\substack{\text{whatever} \\ \text{is paid to} \\ \text{operate}}} \le \underbrace{\sum_{i=1}^{n} \hat{c}_i}_{\substack{\text{whatever} \\ \text{was charged} \\ \text{before}}} \quad \forall n$

covered by

total true cost

total amortized cost $\Rightarrow$ upper bound on the total true cost

Dynamic table (accounting method)

- Charge $\hat{c}_i = \$3$ for ith Insert
  - \$1 pays for immediate insert
  - \$2 stored for table doubling
- When table doubles: \$1 moves recent item
  - \$1 moves old item

ex: just doubled from $4 \to 8 \to 16$

\$ amount

0	0	0	0	2	2	2	2

0	0	0	0	0	0	0	0	0	2	2	2				

ins
\$3 charged
\$1 used
\$2 stored

\$8, 8 items to copy

$\Rightarrow$ if we charge \$3 for each insertion, can handle table doubling

Invariant:

Bank balance ≥ 0

$\underbrace{\sum_{i=1}^{n} \hat{c}_i}_{3n} \ge \sum_{i=1}^{n} c_i$

\$3 per insert

different charging schemes can work:

e.g. charge first 2 and 3 all others

charge 3 all

charge 4 all ... $\leftarrow$ looser bound

but charging 2 all will not work.
$\hookrightarrow$ balance will go negative

$\Rightarrow$ have to try/design [redacted] charging to get a tight bound

Potential method

"Bank account" viewed as potential energy of dynamic set.

Framework:

- Start with data structure D_0
- Op i transforms $D_{i-1} \to D_i$
- Cost of op i is c_i
- Define potential function

$$\Phi : \{D_i\} \to \mathbb{R}$$

real val. potential is associated with each data structure

$$\Phi(D_0) = 0 \text{ and } \Phi(D_i) \geq 0 \ \forall i$$

- Amortized cost $\hat{c}_i$ with respect to Φ is:

$$\hat{c}_i = c_i + \underbrace{\Phi(D_i) - \Phi(D_{i-1})}_{\substack{\text{potential difference} \\ \Delta\Phi_i}}$$

If $\Delta\Phi_i > 0$, then $\hat{c}_i > c_i$

Op i stores work in data structure for later — bank account went up.

If $\Delta\Phi_i < 0$, then $\hat{c}_i < c_i$

Data structure delivers up stored work to help pay for op i — bank account went down.

Accounting	vs.	Potential
specify amortized cost, make sure the bank account does not go negative → analyze bank account		specify bank account all the time → analyze amortized costs

Total amort cost of n ops is:

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}))$$

$$= \sum_{i=1}^{n} c_i + \underbrace{\Phi(D_n)}_{\geq 0} - \underbrace{\Phi(D_0)}_{= 0}$$

$$\geq \sum_{i=1}^{n} c_i$$

tot. amortized cost is upper bound on tot. true cost

Table doubling:

Define $\Phi(D_i) = \underbrace{2i}_{\text{added to account so far}} - \underbrace{2^{\lceil \lg i \rceil}}_{\text{subtracted so far due to doublings}}$ at point i

Assume $2^{\lceil \lg 0 \rceil} = 0$.

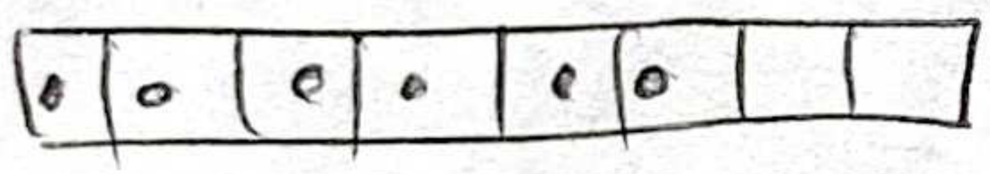
Note $\Phi(D_0) = 0$ ✓

$\Phi(D_i) \geq 0 \quad \forall i$ $\qquad \lceil \lg i \rceil \leq \lg(i) + 1$

$2^{\lg(i)+1} = 2i$

acc: 0 0 0 0 2 2

Ex



$\Phi = 2 \cdot 6 - 2^3 = 4$

Amortized cost of ith insert:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \qquad c_i = \begin{cases} i & \text{if } i-1 \text{ is power } 2 \\ 1 & \text{otherwise} \end{cases}$$

$$= c_i + (2i - 2^{\lceil \lg i \rceil}) - (2(i-1) - 2^{\lceil \lg(i-1) \rceil})$$

$$= \begin{cases} i & \text{if} \ldots \\ 1 & \text{otherwise} \ldots \end{cases} + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg(i-1) \rceil}$$

my comment
if $i-1$ is power o 2, $\lceil \lg i \rceil = \lceil \lg(i-1) + 1 \rceil$

Case 1: $i-1$ is power of 2

$$c_i = i, \quad \hat{c}_i = i + 2 - \underbrace{2^{\lceil \lg i \rceil}}_{-2(i-1)} + 2^{\lceil \lg(i-1) \rceil} = i + 2 + (i-1) =$$

$$= i + 2 - 2(i-1) + \cancel{(i-1)} = i + 2 - i + 1 = 3$$

Case 2 : $i-1$ is not exact power of 2

$$\hat{c}_i = 1 + 2 - \cancel{2^{\lceil \lg i \rceil}} + \cancel{2^{\lceil \lg(i-1) \rceil}} = 3$$

$\hat{c}_i = 3$ for every Insert

n Inserts cost $\Theta(n)$ in worst case, $\Theta(1)$ per Insert, on average

Conclusions :

Amortized costs provide a clean abstraction for data structure performance.

- Any method can be used, but each has situations where it is arguably simplest or most precise.
- Different potential functions or accounting costs may yield different bounds.