上海科技大学
ShanghaiTech University

CS240 Algorithm Design and Analysis

Lecture 18

Amortized Analysis

Quan Li
Fall 2024
2024.11.28

Amortized Analysis

Amortized Analysis

- Suppose we want to bound the amount of time to perform n (possibly different) operations on a data structure
- If max amount of time for each operation is $f(n)$, $n \cdot f(n)$ is upper bound on the time for all the operations
- But some operations might take more time than others
  - Even the same operation can take different amounts of time each time it's executed
  - So $n \cdot f(n)$ may overestimate actual amount of time taken
  - The bound isn't tight.
- Amortized analysis looks at the average amount of time for each operation over all the operations
  - The average is taken over the worst-case execution, i.e., a sequence of operations with the highest average cost for the data structure

Example for Amortized Analysis

Stack operations:

- PUSH(S, x)
- POP(S)

- MULTIPOP(S, k)
  - **while** S not empty and k > 0
  - **do** POP(S)
  - k=k-1

Let us consider a sequence of n PUSH, POP, MULTIPOP.

- The worst-case cost for MULTIPOP in the sequence is O(n), since the stack size is at most n.
- Thus the cost of the sequence is $O(n^2)$. Correct, but not tight.

Aggregate Analysis

In fact, a sequence of n operations on an initially empty stack cost at most O(n).

- An object can be POPed only once (including in MULTIPOP) after it has been PUSHed.
- #POPs is at most #PUSHs, which is at most n.

Thus, the amortized cost of an operation is O(n)/n = O(1).

$$T(n) = \sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} \widehat{C}_i.$$

$\frac{1}{n}\sum_{i=1}^{n} \widehat{C}_i$

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} C_i \\ &= \#Push + \#Pop \\ &\leq 2 \times \#Push \\ &\leq 2n \end{aligned}$$

$\widehat{C}_i$: the amortized cost in step i

C_i: the actual cost in step i

Three Methods of Amortized Analysis

Aggregate analysis:

- Total cost of n operations / n

Accounting method:

- Assign each type of operation a (possibly different) amortized cost
- Overcharge some operations
- Store the overcharge as credit on specific objects
- Then use the credit for compensation for some later operations

Potential method:

- Same as accounting method
- But store the credit as "potential energy" as a whole

Another Example: Incrementing a Binary Counter

Binary counter A, initialized to all 0.

```
INCREMENT(A)
      i←0
      while i<length[A] and A[i]=1 do
         A[i]←0
           i←i+1

      if i<length[A]
         A[i]←1
```

- Consider a k-digit binary counter. When we increment the counter, we flip some bits
  - Suppose each bit flip costs 1 unit
- **What is the total cost for incrementing the counter n times, starting from 0?**
  - Since there are k digits, a trivial upper bound is O(nk)
  - However, the actual number of bit flips is much less, because most increments only flip a few bits

Amortized Analysis of INCREMENT(A)

Observation: The running time determined by #flips, but not all bits flip each time INCREMENT is called.

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Figure 17.2 An 8-bit binary counter as its value goes from 0 to 16 by a sequence of 16 INCREMENT operations. Bits that flip to achieve the next value are shaded. The running cost for flipping bits is shown at the right. Notice that the total cost is never more than twice the total number of INCREMENT operations.

A[0] flips every time, total n times.
A[1] flips every other time, $n/2$ times.
A[2] flips every forth time, $n/4$ times.
...
A[i] flips $n/2^i$ times.

Thus total #flips is $\sum_{i=0}^{\log n-1} \frac{n}{2^i} = O(n)$.

Amortized cost = O(1).

Amortized Analysis: Accounting Method

Idea:

- Assign an amortized cost to each type of operation.
- The amortized cost may be more or less than the actual cost.
- When amortized cost > actual cost, the difference is saved in specific objects in the data structure as credits.
- The credits can be used by later operations whose amortized cost < actual cost.

Accounting Method: Stack Operations

Actual costs:

- PUSH :1, POP :1, MULTIPOP: k.

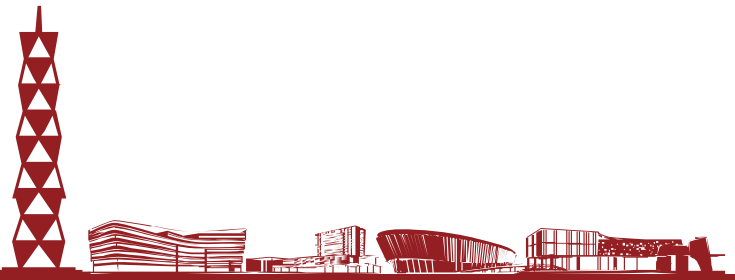
Let's assign the following amortized costs:

- PUSH:2, POP: 0, MULTIPOP: 0.

Similar to a stack of plates in a cafeteria.

- Suppose $1 represents a unit cost.
- When pushing a plate, use one dollar to pay the actual cost of the PUSH and leave one dollar on the plate as credit.
- When POPing a plate, the one dollar on the plate is used to pay the actual cost of the POP. (Same for MULTIPOP).
- By charging PUSH a little more, do not charge POP or MULTIPOP.

So the amortized cost is 2 for PUSH, and 0 for POP and MULTIPOP,



Accounting Method: Binary Counter

In this analysis, the amortized cost will be decided later.

- Let \$1 represent each unit of cost (i.e., the flip of one bit).
- Whenever a bit is set to 1, we will spend \$2: \$1 to pay the actual cost and store another \$1 on the 1-bit as credit.
- When a bit is set to 0, the stored \$1 pays the cost.
- At most one bit is set in each operation, so the amortized cost of 2 is enough to cover the cost.

OP	Real Cost C_{OP}	Amortized Cost $\widehat{C_{OP}}$
`flip(0→1)`	1	2
`flip(1→0)`	1	0

$$
\begin{aligned}
T(n) &= \sum_{i=1}^{n} C_i \\
&= \#flip(0 \to 1) + \#flip(1 \to 0) \\
&\leq 2\#flip(0 \to 1) \\
&\leq 2n
\end{aligned}
$$

The Potential Method

To keep track of the true total cost of a sequence of operations, we use a potential function $\Phi: D \to \mathbb{R}$, where D is the set of states of the data structure.

Let D_i be the state of the data structure after applying the i'th operation, and c_i be the cost of the i'th operation.

Def The amortized cost for the i'th operation is $\hat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1})$.

Using the amortized cost, we sometimes overcharge and sometimes undercharge for operations.

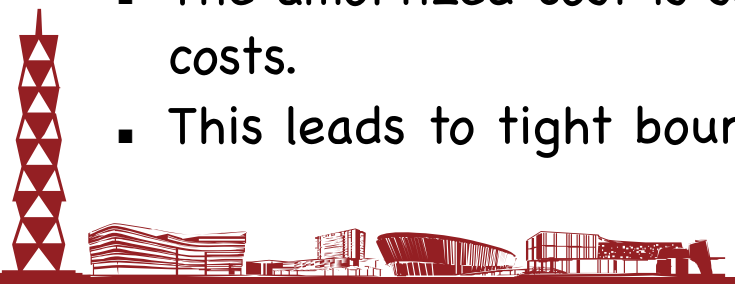
- i.e., when $\hat{c_i} > c_i$, we overcharge, and when $\hat{c_i} < c_i$ we undercharge.

However, the total amortized cost is at least the total actual cost, i.e. $\sum_i \hat{c_i} \geq \sum_i c_i$.

- So total amortized cost is an upper bound on total actual cost.

If we design the right potential function, we can keep track of the total cost by tracking the amortized costs.

- The amortized cost is sometimes easier to analyze than directly keeping track of actual costs.
- This leads to tight bounds for many data structures.



The Potential Method

- When we overcharge, i.e. $\hat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) > c_i$, Φ increases.
  - We "store" the extra amortized cost $\hat{c_i} - c_i$ we charged the i'th operation in Φ
  - Φ is also called "credit" or "potential" (energy).
- When we undercharge, i.e. $\hat{c_i} < c_i$, Φ decreases.
  - We use some of the stored credit to pay for the $c_i - \hat{c_i}$ amount of actual cost that the amortized cost doesn't account for.

- Lemma Suppose $\Phi(D_n) \geq \Phi(D_0)$. Then $\sum_{i=1}^{n} \hat{c_i} \geq \sum_{i=1}^{n} c_i$.
- Proof

  $\sum_{i=1}^{n} \hat{c_i} = \sum_{i=1}^{n}(c_i + \Phi(D_i) - \Phi(D_{i-1})) = (\sum_{i=1}^{n} c_i) + \Phi(D_n) - \Phi(D_0) \geq \sum_{i=1}^{n} c_i$.

  The second equality follows because all the terms except $\Phi(D_n), \Phi(D_0)$ telescope away.
- A simple way to ensure $\Phi(D_n) \geq \Phi(D_0)$ is to design Φ so that $\Phi(D_0) = 0$, and $\Phi(D_i) \geq 0$ for all i.

Potential Method: Stack Operation

Potential for a stack := the number of objects in the stack.

- So $\Phi(D_0)=0$, and $\Phi(D_i) \geq 0$

Amortized cost of stack operations:

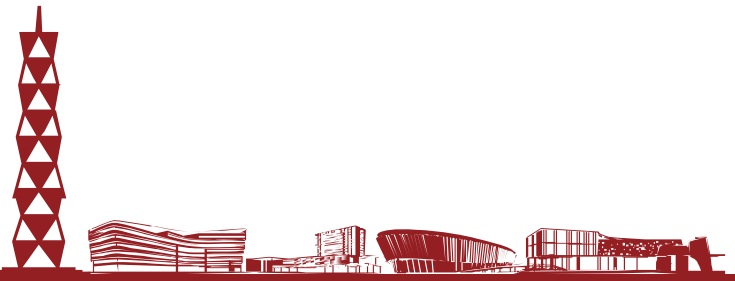
- PUSH:
  - Potential change: $\Phi(D_i) - \Phi(D_{i-1}) = 1$.
  - Amortized cost: $c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$.
- POP:
  - Potential change: $\Phi(D_i) - \Phi(D_{i-1}) = -1$.
  - Amortized cost: $c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (-1) = 0$.
- MULTIPOP(S, k):
  - Potential change: $\Phi(D_i) - \Phi(D_{i-1}) = -k$.
  - Amortized cost: $c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k + (-k) = 0$.

So amortized cost of each operation is O(1).

$$T(n) = \sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} \hat{C}_i = 2n_1$$

Potential Method: Binary Counter

- Let $\Phi(D_i) = b_i$, where D_i is the state of the counter after i increments, and b_i is the number of 1's in D_i.
- Suppose the i'th operation sets t_i bits from 1 to 0.
  - Then the actual cost is $c_i = t_i + 1$.
  - This sets t_i bits from 1 to 0, and one bit from 0 to 1 for the carry.
- If $b_i = 0$, then the i'th operation reset all the bits.
  - Also, all the bits were set in D_{i-1}.
  - So $t_i = b_{i-1} = k$.
- If $b_i > 0$, then $b_i = b_{i-1} - t_i + 1$.
  - t_i bits went from 1 to 0, and one carry bit went from 0 to 1.
- In both cases, $b_i \leq b_{i-1} - t_i + 1$.



Potential Method: Binary Counter

- Since $b_i \leq b_{i-1} - t_i + 1$, then $\Phi(D_i) - \Phi(D_{i-1}) = b_i - b_{i-1} \leq 1 - t_i$.
- So the amortized cost $\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq (t_i + 1) + (1 - t_i) = 2$.
- Finally, $\Phi(D_0) = 0$, since the counter is initially 0, and $\Phi(D_n) \geq 0$.
- Thus, by the lemma the total cost for all n increments is $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \widehat{c_i} \leq 2n$.

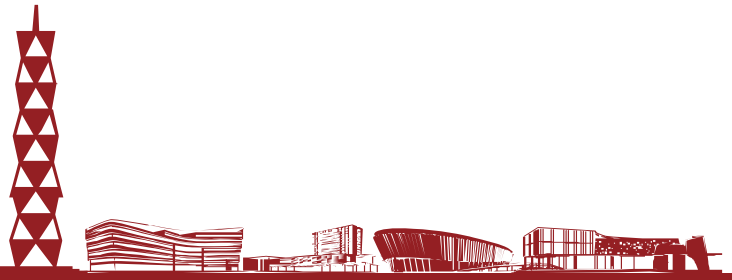
Amortized Analysis: Dynamic Table

Scenario:

- A table (e.g., an array)
- Do not know how large in advance
- Insertion at the end: O(1) time if there is space
- Deletion: Mark the element as "deleted".
- Rebuilding the table takes time linear to the table size

Goal:

- O(1) amortized cost.
- Load factor $\alpha \geq$ constant fraction.



Dynamic Table: Expansion with Insertion Only

Doubling strategy: When the table becomes full, double its size and rebuild.

- Guarantees $\alpha \geq 1/2$.

Aggregate analysis:

- Total cost of n insertions themselves: O(n)
- Total cost of all rebuilds:
  - 1 + 2 + 4 + 8 + … + n = O(n)
- Amortized cost = O(1)

Particularly,

Total cost after n push calls:

1 + 1 + … + 1, fill, n times

1 + 2 + 4 + … + n, copy $\log_2 n$ times

Amortized cost = (n + 2n − 1)/n = 3O(1)

$$\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\lfloor lgn \rfloor} 2^j < n + 2n = 3n$$

Accounting Analysis

Charge $3 per insertion of x.

- $1 pays for x's insertion
- $1 is stored as credit with x
- $1 is stored as credit with some other element in the table currently having no credit

Analysis.

- Suppose we've just expanded, with size = m.
- Will expand again after another m insertions.
- Each insertion will put $1 on one of the m items that were in the table just after expansion and will put $1 on the item inserted.
- Have $2m credits by next expansion: Just enough to pay for the expansion, with no credit left over!

	1
Credit	1

	1	
Credit	0	

	1	2
Credit	1	1

	1	2		
Credit	0	0		

	1	2	3	
Credit	1	0	1	

	1	2	3	4
Credit	1	1	1	1

Potential Analysis of Dynamic Table

- Define the potential of the table after the ith insertion by $\Phi(D_i)=2*num_i - size_i$
- So $\Phi(D_0) = 0$, $\Phi(D_i)>=0$, for all i

Example

$\Phi = 2 \cdot 6 - 8 = 4$

- Case 1: i − 1 is not an exact power of 2

$\hat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1})$ = 1 + ($2*num_i - size_i$) − ($2*num_{i-1} - size_{i-1}$)

= 1 + ($2*num_i - size_i$) − ($2*(num_i - 1) - size_i$) = 3

- Case 2: i − 1 is an exact power of 2

$\hat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1})$ = num_i + ($2*num_i - size_i$) − ($2*num_{i-1} - size_{i-1}$)

= num_i + ($2*num_i - 2(num_i-1)$) − ($2(num_i-1)-(num_i-1)$) = num_i + 2 − (num_i − 1) = 3

Therefore, n insertions cost O(n) in the worst case