Amortized Analysis

Data Structures and Algorithms

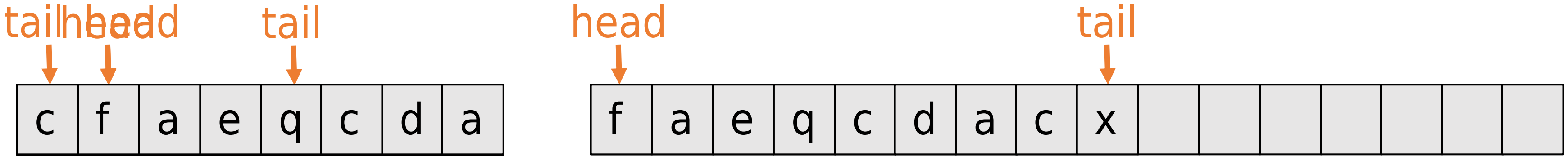
Nanjing University, Fall 2021
郑朝栋

Implement **Queue** with **CircularArray**

- **CircularArray** supports **Queue** operations in $O(1)$ time.
- But what to do when the array is full?! $\Theta(1)$
  - Allocate a new array of double size. $\Theta(n)$
  - Copy existing items to the new array, and insert new element. $\Theta(1)$
  - Delete old array.
- But now the **Insert** operation may take $\Theta(n)$ time!
- So a sequence of n operations can take $O(n^2)$ time?!

Correct but not tight!!

Insert(x)



Amortized Analysis

- Technique for analyzing “average cost”:
  - Often used in data structure analysis
  - (Expensive Op. and Cheap Op.) + (Expensive Op. *can’t be frequent*)
    => Average cost of Op. for *any* sequence of Op. must be low.

- In some sense, like “pay in installments”.
  - Is using iPhone expensive?
  - Sure, average monthly salary in Jiangsu≈ 8635 / 53
  - But you don’t but a new iPhone everyday!
    Pay < 550 per month if new iPhone every other yea

RMB 12999

1TB

Salary data from: http://www.jiangsu.gov.cn/art/2021/6/18/art_34151_9852928.html

Amortized Analysis

- Technique for analyzing “average cost”:
  - Often used in data structure analysis
  - (Expensive Op. and Cheap Op.) + (Expensive Op. *not frequent*)
    => Average cost of Op. for *any* sequence of Op. must be low.
- **Definition:** Operation has amortized cost $\hat{c}(n)$, if for *every* $k \in \mathbb{N}^+$, the total cost of *any* k operations is $\leq \sum_{i=1}^{k} \hat{c}(n_i)$.
  (n_i is the size of the data structure when applying the i^{th} op.)
- Different operations may have different amortized costs.

Amortized Analysis

- Consider a sequence operations: c_i = actual cost of the i^{th} op.; $\widehat{c_i}$ = amortized cost of the i^{th} op.
- For the amortized cost to be valid: $\sum_{i=1}^{k} c_i \leq \sum_{i=1}^{k} \widehat{c_i}$ for any $k \in \mathbb{N}^+$

- Total cost of k operations is $\leq \sum_{i=1}^{k} \widehat{c_i}$, not $\leq k \cdot \max\{c_i\}$.
- Average cost of k operations is $\leq \frac{\sum_{i=1}^{k} \widehat{c_i}}{k}$, not $\leq \max\{c_i\}$.

Amortized Analysis

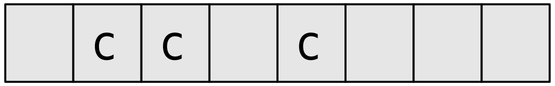
- **Definition:** Operation has amortized cost $\hat{c}(n)$, if for *every* $k \in \mathbb{N}^+$, the total cost of *any* k operations is $\leq \sum_{i=1}^{k} \hat{c}(n_i)$.

  (n_i is the size of the data structure when applying the i^{th} op.)

- Different operations may have different amortized costs.
  - Consider **CircularArray** implementation of **Queue**.
  - **Insert** have amortized cost 2? ($\hat{c}(n) = 2$ if op. is **Insert**.)
  - **Remove** has amortized cost 1? ($\hat{c}(n) = 1$ if op. is **Remove**.)

Ignore cost of array alloc and free for now.

	Actual total cost		Amortized total cost
Insert(c)	1		2
Insert(c)	1+(1+1)=3		2+2=4
Insert(c)	3+(2+1)=6		4+2=6
Insert(c)	6+1=7		6+2=8
Insert(c)	7+(4+1)=12	>	8+2=10



Amortized Analysis

- **Definition:** Operation has amortized cost $\hat{c}(n)$, if for *every* $k \in \mathbb{N}^+$, the total cost of *any* k operations is $\leq \sum_{i=1}^{k} \hat{c}(n_i)$.

  (n_i is the size of the data struct

- Differen amortized c

So **CircularArray** operations has $O(1)$ amortized cost?

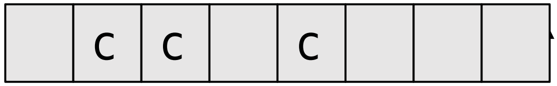
(Even though some op. can cost $\Theta(n)$.)

CircularArray implementation of **Queue**.

- **Insert** have amortized cost 3? ($\hat{c}(n) = 3$ if op. is **Insert**.)
- **Remove** has amortized cost 1? ($\hat{c}(n) = 1$ if op. is **Remove**.)

Ignore cost of array alloc and free for now.

	Actual total cost	**Amortized total cost**
Insert(c)	1	3
Insert(c)	1+(1+1)=3	3+3=6
Insert(c)	3+(2+1)=6	6+3=9
Insert(c)	6+1=7	9+3=12
Insert(c)	7+(4+1)=12	12+3=15
Remove()	12+1=13	15+1=16



Amortized Analysis

The Accounting Method

- Consider a sequence operations: c_i = actual cost of the the i^{th} op.; $\widehat{c_i}$ = amortized cost of the i^{th} op.
- For the amortized cost to be valid: $\sum_{i=1}^{k} c_i \leq \sum_{i=1}^{k} \widehat{c_i}$ for any $k \in \mathbb{N}^+$
- Imagine you have a bank account B.
- For the i^{th} op., you spend $\widehat{c_i}$ money:
  - Recall the actual cost of the i^{th} op. is c_i.
  - If $\widehat{c_i} \geq c_i$, pay c_i for the op., and deposit $\widehat{c_i} - c_i$ into B.
  - If $\widehat{c_i} < c_i$, pay c_i for the op., and withdraw $c_i - \widehat{c_i}$ from B.
- Amortized analysis valid if $B = \sum_{i=1}^{k} (\widehat{c_i} - c_i)$ always ≥ 0.

The Accounting Method

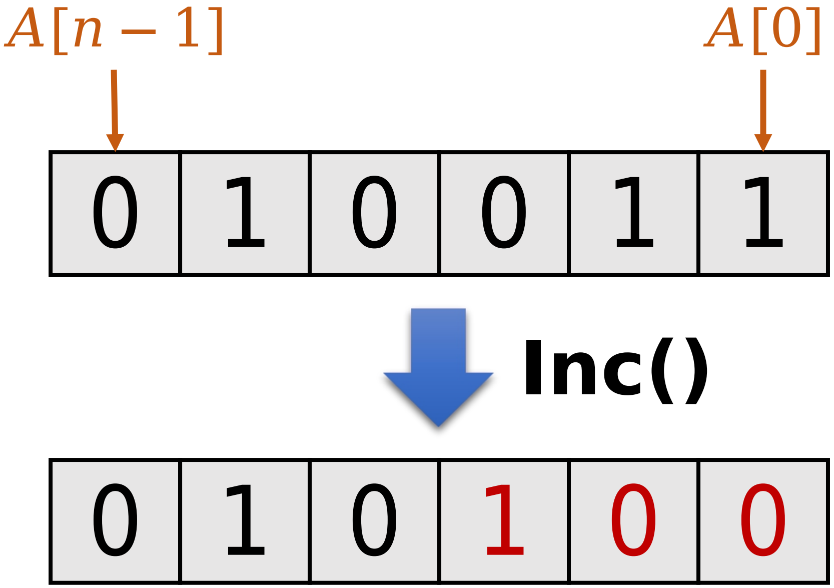
Example: **CircularArray** based **Queue**

- $\widehat{c_i} = 3$ if the i^{th} op is **Insert**, $\widehat{c_i} = 1$ if the i^{th} op is **Remove**.
- **Goal:** Prove $\sum_{i=1}^{k} c_i \leq \sum_{i=1}^{k} \widehat{c_i}$ for any $k \in \mathbb{N}^+$ operations.
- **Strategy:** account always non-negative via induction on k.
- [Basis] Prior to 1st op., account value is 0.
- [Hypothesis] Prior to i^{th} op., account value is always non-negative.
- [Inductive Step] Consider the i^{th} op.
  - If it's **Remove**, then we make no change to account value.
  - If it's **Insert** without expansion, we add 2 to account value.
  - If it's **Insert** with expansion. Assume expand from n to $2n$.
    Last expand must be from $n/2$ to n.
    Since last expand, each **Insert** adds 2, each **Remove** makes no change.

The Accounting Method

Example: Binary Counter

- Use length n binary array A to represent a number.
- The number is 0 initially, and **Inc** op. adds 1 to this number.
- Cost of **Inc**: number of bits it flipped.
- Average cost of k **Inc** operations?
  - Easy answer: $O(n)$
  - More careful analysis… (Amortized analysis…)



$A[n-1]$ $A[0]$

0	1	0	0	1	1

Inc()

0	1	0	1	0	0

```
Inc(A):
i=0
while (i<n and A[i]==1)
  A[i]=0
  i=i+1
if (i<n)
  A[i]=1
```

The Accounting Method

Example: Binary Counter

- The number is 0 initially, and **Inc** op. adds 1 to this number.
- Cost of **Inc**: number of bits it flipped.
- In each **Inc**: 0 → 1: at most 1 bit; 1 → 0: many bits.
- But a bit has to be set to 1 before it resets to 0!
- If we deposit 1 whenever we 0 → 1, later 1 → 0 are "**free**"!
- ost c is

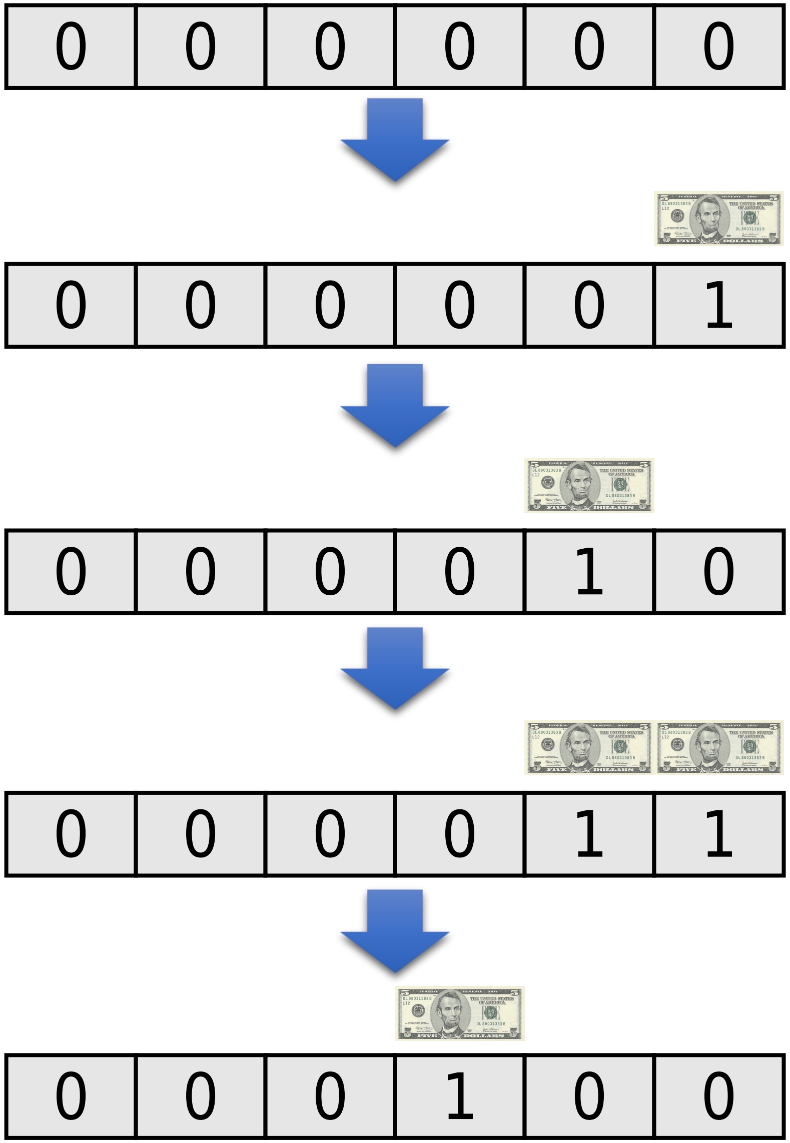
0	1	0	0	1	1

Inc()

0	1	0	1	0	0

Inc(A):

```
i=0
while (i<n and A[i]==1)
  A[i]=0
  i=i+1
if (i<n)
  A[i]=1
```



0	0	0	0	0	0
0	0	0	0	0	1
0	0	0	0	1	0
0	0	0	0	1	1
0	0	0	1	0	0

Actual Total Cost	Amortized Total Cost
1	× 2
1 + 2 = 3	× 4
3 + 1 = 4	× 6
4 + 3 = 7	× 8

Amortized Analysis

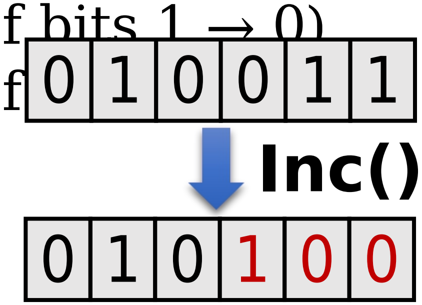
The Potential Method

- Consider a sequence operations:
  c_i = actual cost of i^{th} op.; $\widehat{c_i}$ = amortized cost of i^{th} op.
- For the amortized cost to be valid:
  $\sum_{i=1}^{k} c_i \leq \sum_{i=1}^{k} \widehat{c_i}$ for any $k \in \mathbb{N}^+$
- Design a **potential function** Φ that maps D.S. status to real values.
  - $\Phi(D_0)$: initial potential of D.S., usually set to 0.
  - $\Phi(D_i)$: potential of D.S. after i^{th} operation.
- Define $\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1})$
- For amortized cost to be valid, need $\Phi(D_k) \geq \Phi(D_0)$ for all k.
- “Potential” is like the **balance in account** in “Counting Method”.
  - Potential slowly accumulates during “cheap” operations (deposit).
  - Potential drops a lot after an “expensive” operation (withdraw).
- But the Potential Method could be more powerful in general…

The Potential Method

Example: Binary Counter

- Design a **potential function** Φ that maps D.S. status to real values.
  - $\Phi(D_0)$: initial potential of D.S., usually set to 0.
  - $\Phi(D_i)$: potential of D.S. after ith operation.
- Define $\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1})$, need $\Phi(D_k) \geq \Phi(D_0)$ for all k.
- How to define $\Phi(D_i)$ for Binary Counter? (Recall potential is like "balance".)
- $\Phi(D_i)$ = number of 1s in the array after the ith **Inc** operation.
- Clearly "$\Phi(D_k) \geq \Phi(D_0)$ for all k" is satisfied, how large is $\widehat{c_i}$?
- $$\begin{aligned} c_i &= (\text{\# of bits } 0 \rightarrow 1) + (\text{\# of bits } 1 \rightarrow 0) \\ \Phi(D_i) - \Phi(D_{i-1}) &= (\text{\# of bits } 0 \rightarrow 1) - (\text{\# of} \end{aligned}$$
- $\widehat{c_i} = 2 \cdot (\text{\# of bits } 0 \rightarrow 1) \leq 2$



0	1	0	0	1	1

0	1	0	1	0	0

Back to **CircularArray** based **Queue**

- **Problem:** Array has limited size, what to do when it's full?
- **Solution:** Double the size when array is full and **Insert** comes.
  (Copy items to new array, insert new item, and delete old array.)
- **Solution is Good:** amortized cost of **Insert** and **Remove** both $O(1)$.

- **New Problem:** Lots of **Insert**, then lots of **Remove**. A lot of space wasted!
- **Solution:** Reduce array size to half when array only half loaded after **Remove**.
  (Allocate new array of half size, copy items to new array, and delete old array.)
- Does the above solution achieves $O(1)$ amortized cost?
- **No!** Consider a full array and following ops: **Insert**, **Remove**, **Insert**, **Remove**, ...

- Better solutions? How to prove new solutions indeed "better"?

Reading

- [CLRS] Ch.17 (including 17.4)