平摊分析
Amortized analysis

钮鑫涛
Nanjing University
2023 Fall

The slides are mainly adapted from the original ones shared by Chaodong Zheng and Kevin Wayne. Thanks for their supports!

Implement Queue with CircularArray

- `CircularArray` supports Queue operations in $O(1)$ time.
- Recall that when the array is full:
  - Allocate a new array of **double size**.
  - Copy existing items to the new array, and insert new element.
  - 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?

Not tight!

Amortized Analysis

- Technique for analyzing “average cost”:
  - Often used in data structure analysis
  - Idea: even when expensive operations must be performed, it is often possible to get away with performing them rarely, so that the “average” cost per operation is not so high.
  - **Note**: **Amortized analysis** is different from what is commonly referred to as average case analysis, because it does **not** make any assumption about the distribution of the data values, whereas average case analysis assumes the data are not “bad".
    - That is, amortized analysis is a **worst case** analysis, but for **a sequence of operations**, rather than for **individual operations**.

Aggregate method for Amortized Analysis

- One assumes that there is no need to distinguish between the different operations on the data structure.
  - Then we just use *aggregate method*: add up the cost of all the operations and then divide by the number of operations.
  - aggregate cost $= \dfrac{\text{maximum amount of work done by any series of } m \text{ operations}}{m}$
  - Let $N = 2^k$ for some constant k. The maximum copying cost of N insertions in `CircularArray` is $1 + 2 + 4 + \ldots + 2^{k-1} = 2^k - 1 \approx N$

Ignore cost of array alloc and free for now

  - Therefore, the aggregate cost is $O(\dfrac{N+N}{N}) = O(1)$

Amortized Analysis

- Note: **Different** operations may have **different** amortized costs.
- **Definition**: Operations 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} operation.

Amortized Analysis

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

Total cost of k operations is $\le \sum_{i=1}^{k} \hat{c}_i$

Average cost of k operations is $\le \dfrac{\sum_{i=1}^{k} \hat{c}_i}{k}$

Amortized Analysis

- Definition: Operations 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} operation.
  - Different operations may have different amortized costs.
    - Consider CircularArray implementation of Queue.

Does *Insert* have amortized c ? ($\hat{c}(n) = 2$ if operation is **Insert**.)
Does *Remove* has amortized c ? ($\hat{c}(n) = 1$ if operation is **Remove**.)

Suppose we do not shrink array now

Ignore cost of array alloc and free for now

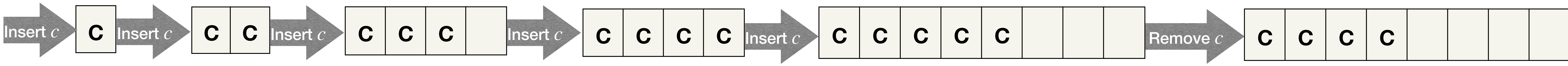
Amortized Analysis

- Definition: Operations 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} operation.
  - Different operations may have different amortized costs.
    - Consider CircularArray implementation of Queue.

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

Ignore cost of array alloc and free for now

So CircularArray operations has $O(1)$ amortized cost?
Even though some operation can cost $\Theta(n)$?



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

Amortized Analysis Techniques

The Accounting Method

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

Example: CircularArray based Queue

- $\hat{c}_i = 3$ if the i^{th} operation is **Insert,** $\hat{c}_i = 1$ if the i^{th} operation is **Remove.**
- **Goal:** Prove $\sum_{i=1}^{k} c_i \leq \sum_{i=1}^{k} \hat{c}_i$ for any $k \in \mathbb{N}^+$ operations.
- **Strategy**: account always non-negative via induction on k.
  - **[Basis]** Prior to 1^{st} op., account balance is 0.
  - **[Hypothesis]** Prior to i^{th} op., account balance is always non-negative.

Example: CircularArray based Queue

- **[Inductive Step]** Consider the i^{th} op.
  - If it's **Remove**, then we make no change to account value.

    $\hat{c}_i - c_i = 1 - 1 = 0$

  - If it's **Insert** without expansion, we add 2 to account value.

    $\hat{c}_i - c_i = 3 - 1 = 2$

  - If it's **Insert** with expansion. Assume *expand* from n to $2n$, then we need to withdraw $n - 1$ value

    $c_i - \hat{c}_i = n + 1 - 2 = n - 1$

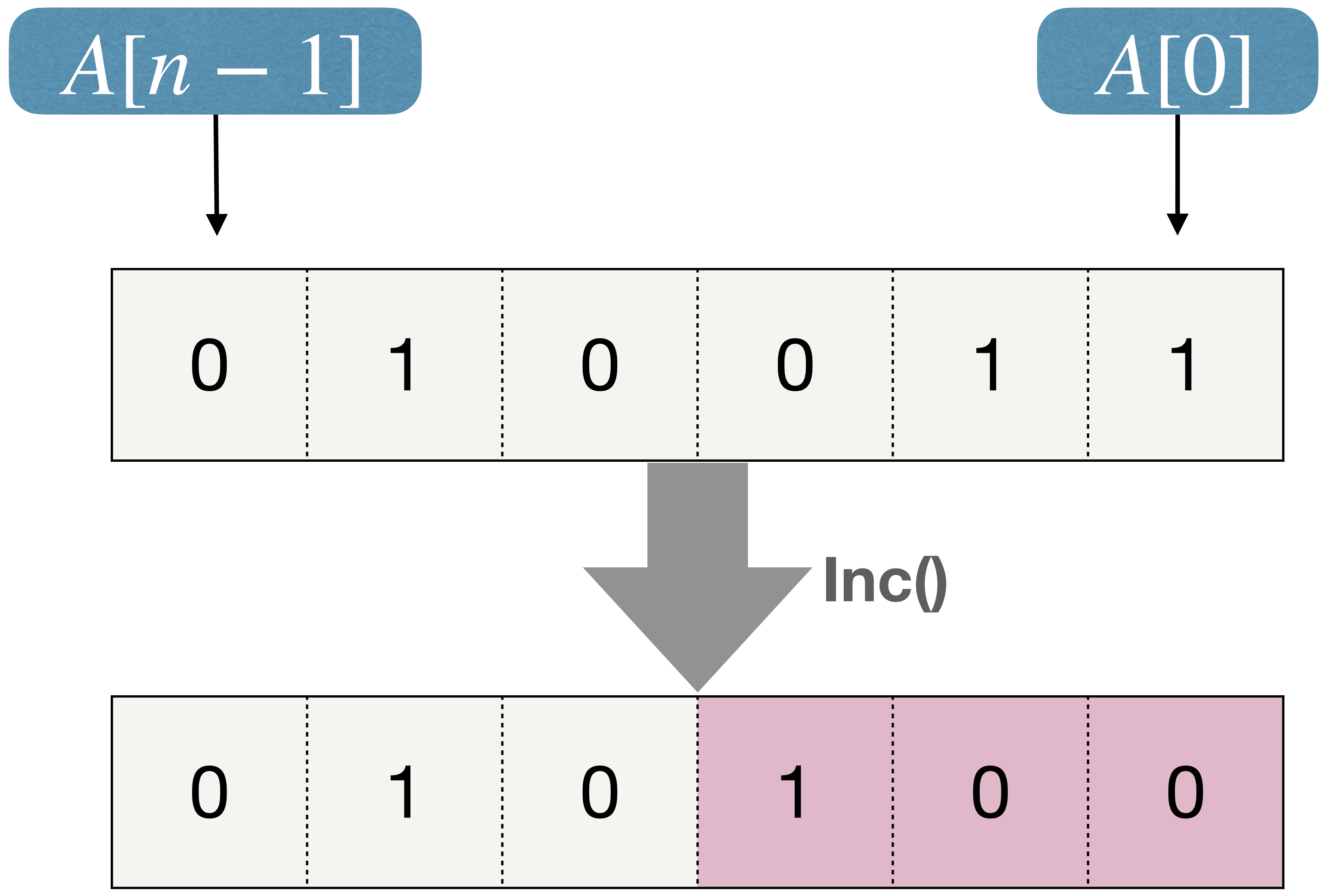
    - Last *expand* must be from $n/2$ to n.
    - Since last *expand*, each **Insert** adds 2, each **Remove** makes no change.
    - Since last *expand*, there are at least $n/2$ **Insert** op.
    - Immediately after last *expand*, account balance is non-negative (**Hypothesis**).
    - Thus prior to i^{th} op., account balance $\geq n$. This is enough!

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 In ***Inc***: number of bits it flipped.
- Average cost of k ***Inc*** operations?
  - Easy answer: $O(n)$
  - More careful analysis… (Amortized analysis…)

Inc(A):

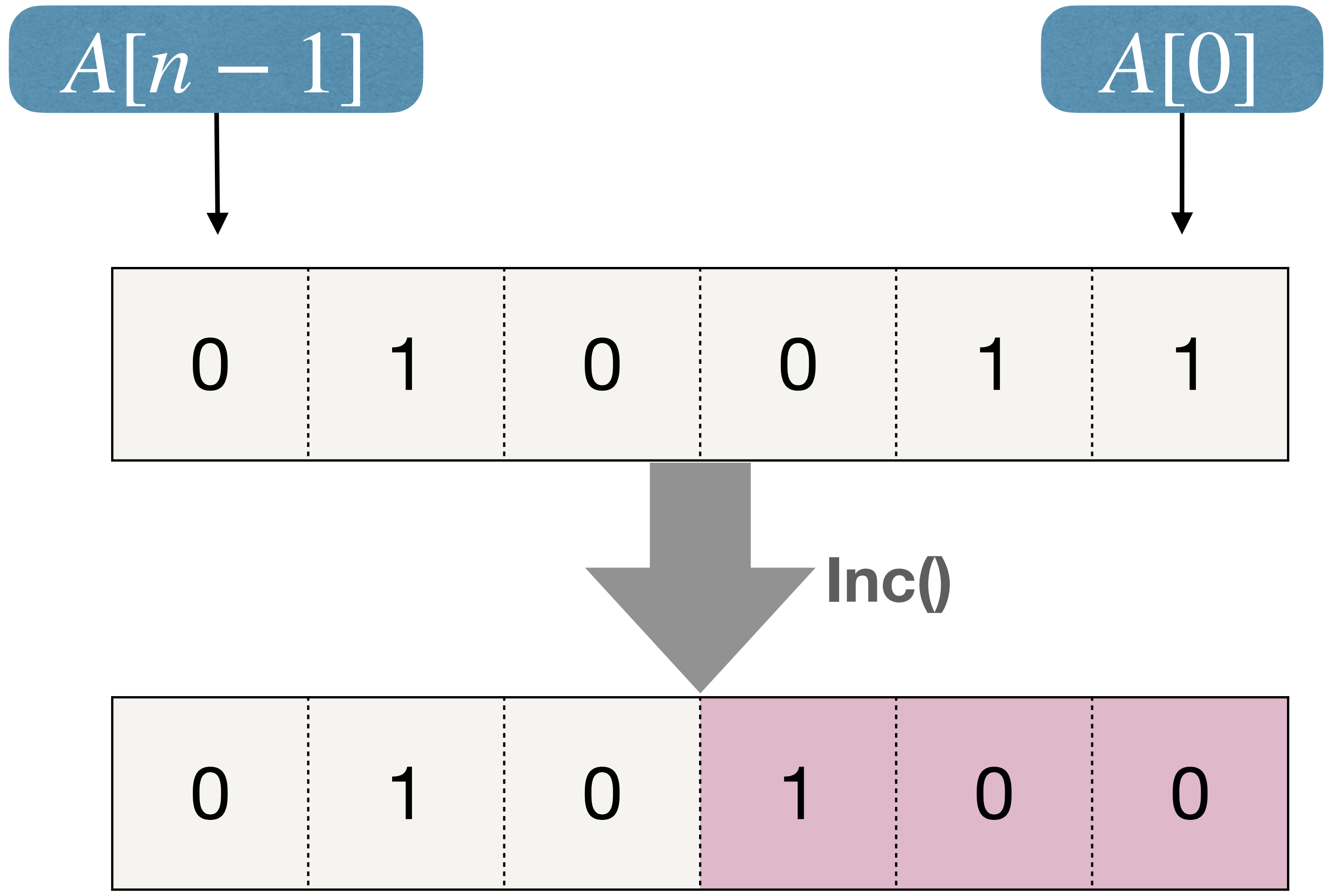
$i := 0$
while $i < n$ **and** $A[i] = 1$
    $A[i] := 0$
    $i := i + 1$
if $i < n$
    $A[i] := 1$



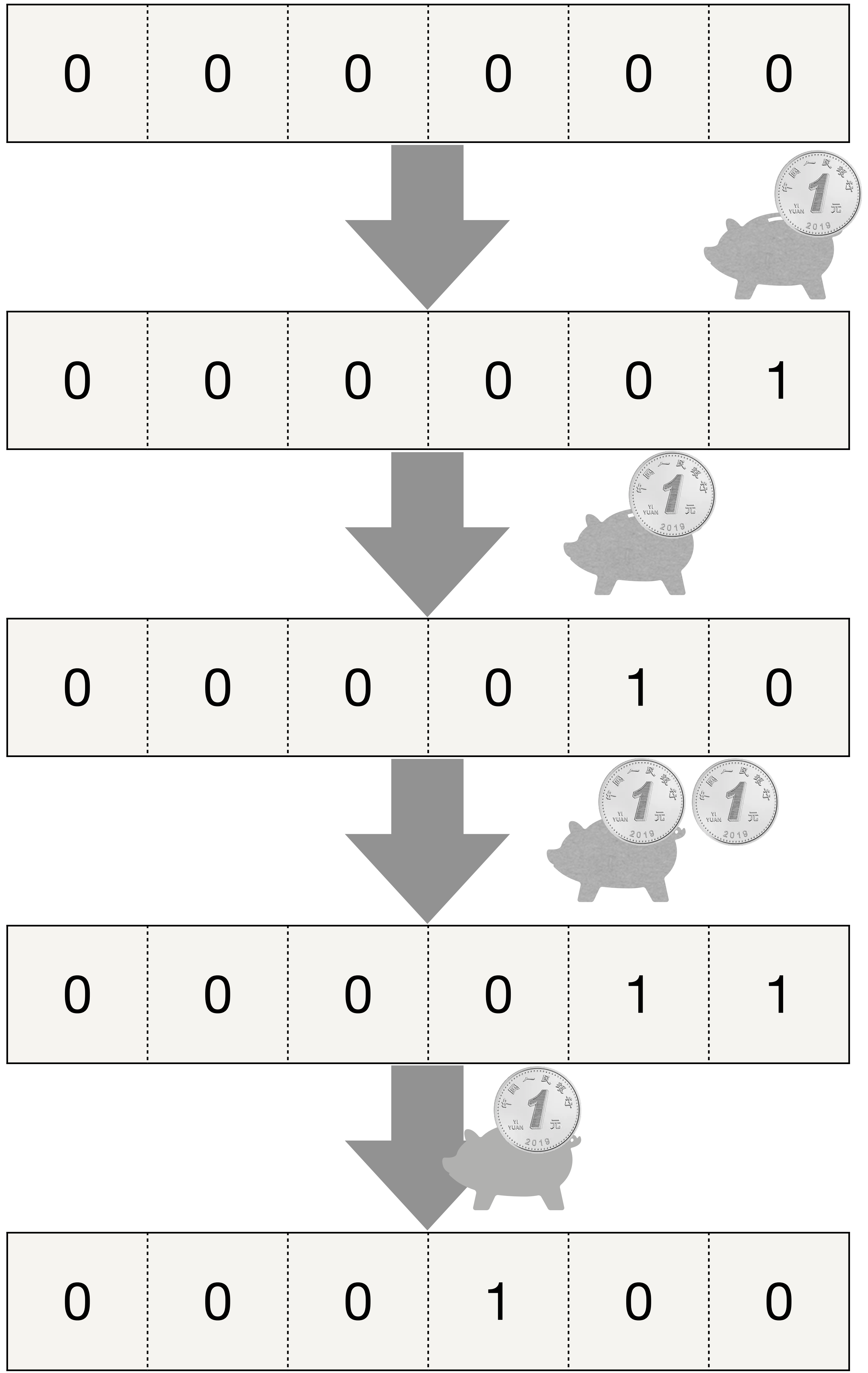
Example: Binary Counter

- The number is 0 initially, and ***Inc*** op. adds 1 to this number.
- Cost of In ***Inc***: number of bits it flipped.
- In each ***Inc***: $0 \to 1$: at most 1 bit, while $1 \to 0$: many bits!
- **But a bit has to be set to 1 before it resets to 0!**
- If we **deposit** 1 whenever we $0 \to 1$, later $1 \to 0$ are **"free"!**
- Each ***Inc*** does $0 \to 1$ at most once, so amortized cost is: $2 = O(1)$

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



Example: Binary Counter



Actual total cost	Amortized total cost
1	$\times 2$
$1+2=3$	$\times 4$
$3+1=4$	$\times 6$
$4+3=7$	$\times 8$

The Potential Method

- Consider a sequence operations: c_i = actual cost of the i^{th} operation; $\hat{c}_i$ = amortized cost of the i^{th} operation. Then, the amortized cost to be valid: $\sum_{i=1}^{k} c_i \leq \sum_{i=1}^{k} \hat{c}_i$ for any $k \in \mathbb{N}^+$.

Now let us consider the amortized cost in a higher level than the specific value in one operation (accounting)!

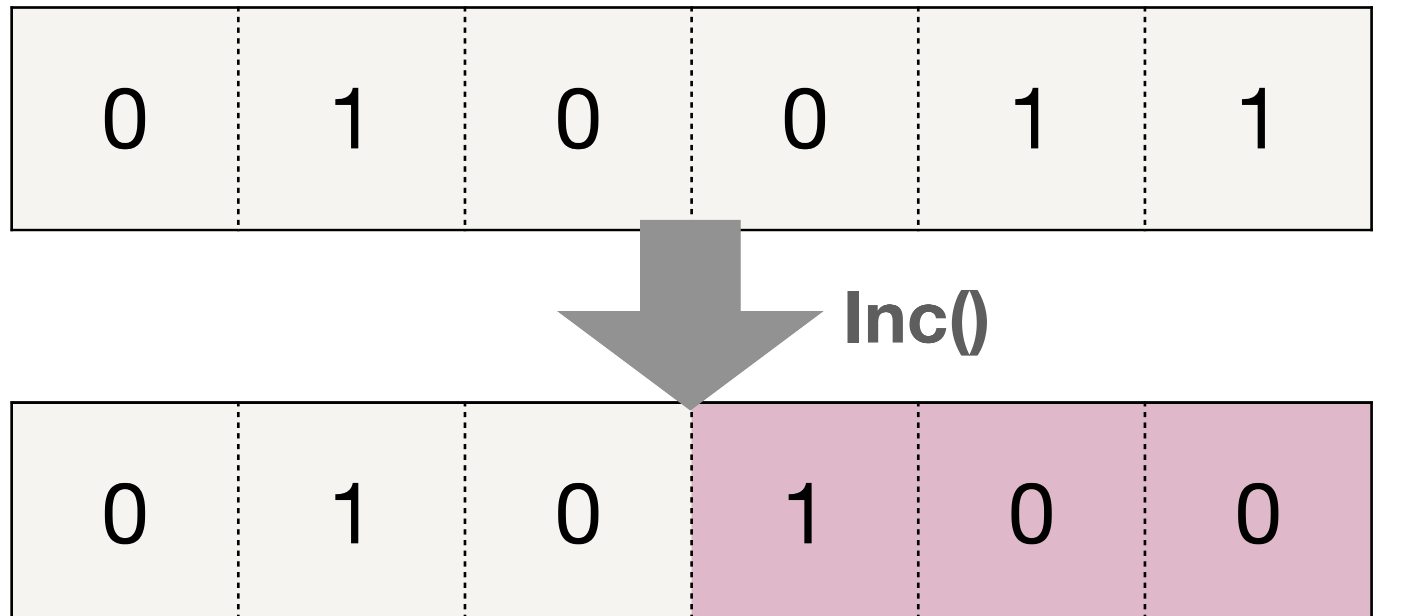
- Design a **potential function** Φ that maps data structure status to real values
  - $\Phi(D_0)$: initial potential of the data structure, usually set to 0.
  - $\Phi(D_i)$: potential of the data structure after i^{th} operation.
- Define $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
- For amortized cost to be valid, need $\Phi(D_k) \geq \Phi(D_0)$ for all k.

The Potential Method

- “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…

Example: Binary Counter

- 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 i^{th} ***Inc*** operation.
- Clearly “$\Phi(D_k) \geq \Phi(D_0)$ for all k.” is satisfied, how large is $\hat{c}_i$?
  - c_i = (number of bits $0 \rightarrow 1$) + (number of bits $1 \rightarrow 0$)
  - $\Phi(D_i) - \Phi(D_{i-1})$ = (number of bits $0 \rightarrow 1$) - (number of bits $1 \rightarrow 0$)
- $\hat{c}_i = 2 \cdot$ (number of bits $0 \rightarrow 1$) ≤ 2



Back to CircularArray based Queue

- Now suppose we need to shrink array for space consideration
  - **Solution(1):** 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.)
  - **Solution(2):** Reduce array size to half when array only $1/4$ loaded after Remove. (Allocate new array of half size, copy items to new array, and delete old array.)
- Quiz: which one is better with respect to amortized cost?

Further reading

- [CLRS] Ch.17