Algorithm Design and Analysis

Amortized Analysis

Dec 6^{th}, 2018

YUN-NUNG (VIVIAN) CHEN HTTP://ADA.MIULAB.TW

國立臺灣大學
National Taiwan University

Slides credited from Hsueh-I Lu & Hsu-Chun Hsiao

Outline

- Amortized analysis
- #1: Stack Operations
  - Aggregate method
  - Accounting method
  - Potential method
- #2: Binary Counter
  - Aggregate method
  - Accounting method
  - Potential method

Algorithm Design & Analysis

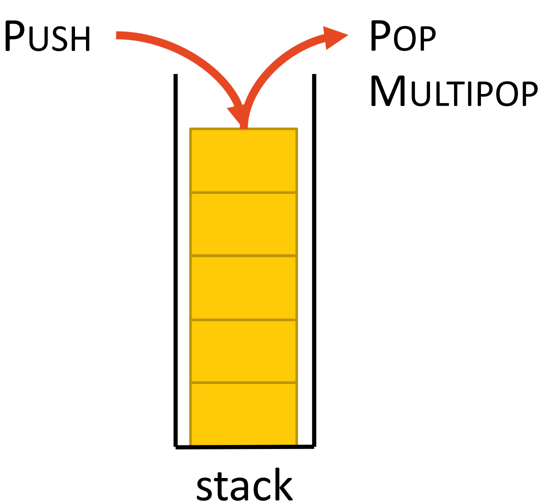
- Design Strategy
  - Divide-and-Conquer
  - Dynamic Programming
  - Greedy Algorithms
  - Graph Algorithms
- Analysis
  - Amortized Analysis

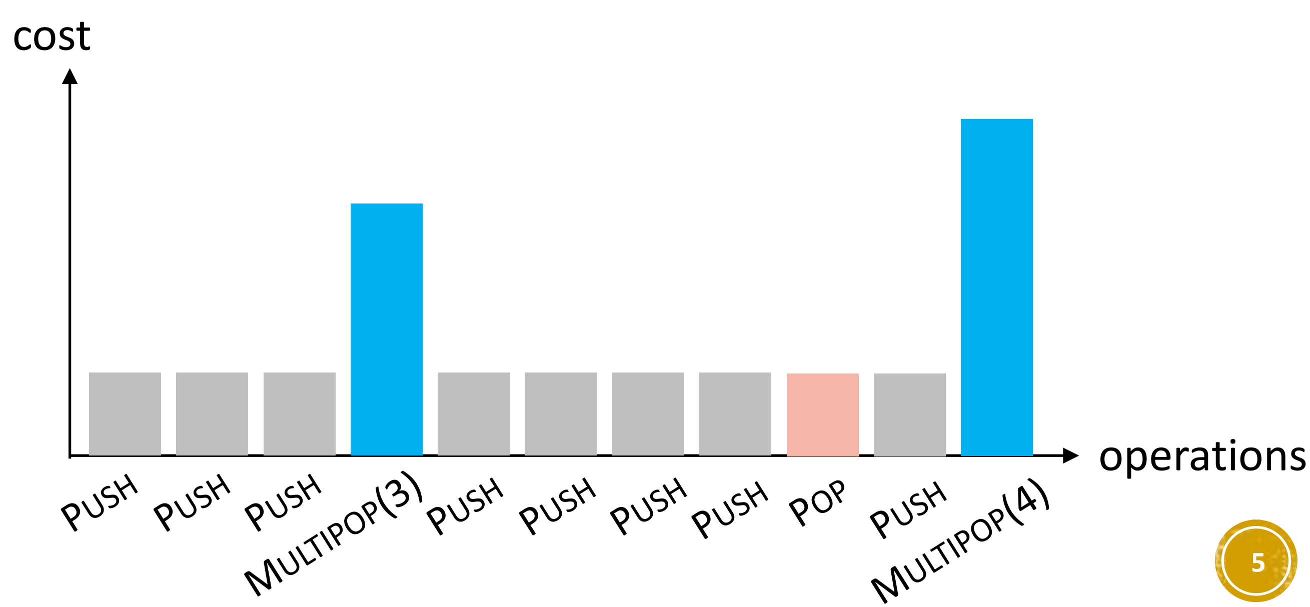
4 Amortized Analysis

Textbook Chapter 17 – Amortized Analysis

Data-Structure Operations

- A data structure comes with operations that organize the stored data
  - Different operations may have different costs
  - The same operation may have different costs





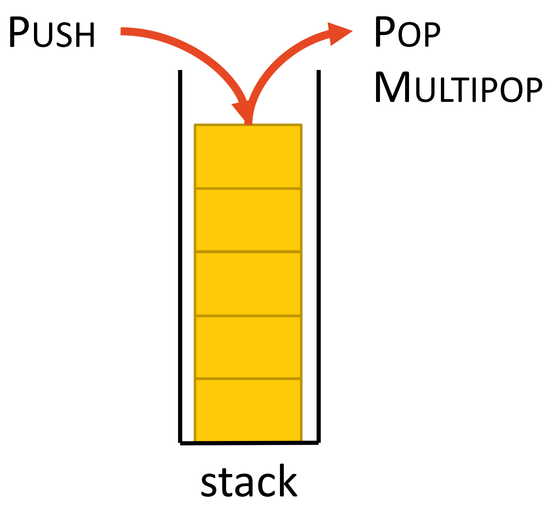
Worst Case Time Complexity

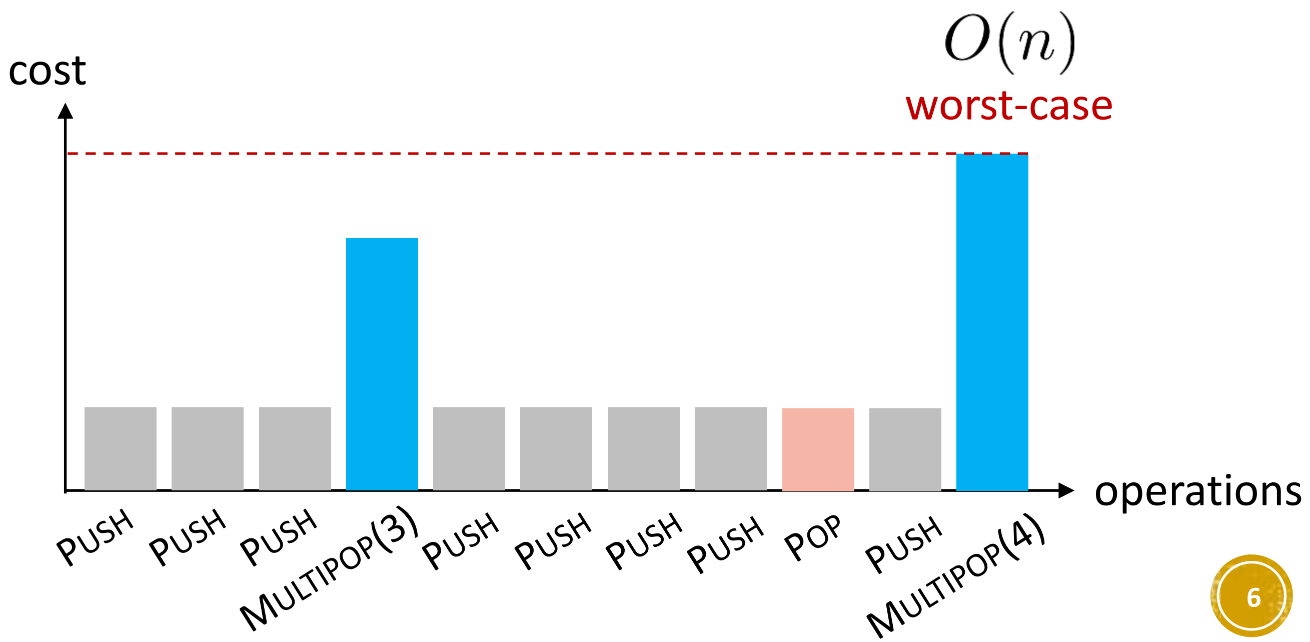
Cost of stack operations

PUSH(S, x) = O(1)

POP(S) = O(1)

MULTIPOP(S, k) = O(min(|S|, k))





Worst Case Time Complexity

Stack Operations
Suppose that we apply a sequence of n operations on a data structure. What is the time complexity of the procedure?

- n-th operation takes MULTIPOP(S, n) = $O(n)$ time in the worst case
- n operations take $O(n^2)$ time

Can this be an over-estimate?

What if only a few operations take $O(n)$ time and the rest of them take $O(1)$ time?

The worst-case bound is not tight because this expensive Multipop operation cannot occur so frequently!

Amortized Analysis

- Goal: obtain an accurate worst-case bound in executing *a sequence of operations* on a given data structure
  - An upper bound for any sequence of *n* operations

- Comparison: types of running-time analysis

Type	Description
Worst case	Running time guarantee for **any input** of size n
Average case	**Expected** running time for a **random input** of size n
Probabilistic	**Expected** running time of a **randomized algorithm**
Amortized	Worst-case running time for **a sequence of *n* operations**

3 Methods for Amortized Analysis

Aggregate method (聚集法)

- Determine an upper bound $T(n)$ on the cost over any sequence of n operations
- The average cost per operation is then $T(n)/n$
- All operations have the same amortized cost

Accounting method (記帳法)

- Each operation is assigned an amortized cost (may differ from the actual cost)
- Each object of the data structure is associated with a credit
- Need to ensure that every object has sufficient credit at any time

Potential method (位能法)

- Similar to accounting method; each operation is assigned an amortized cost
- The data structure as a whole maintains a credit (i.e., potential)
- Need to ensure that the potential level is nonnegative at any time

10

Stack Operations

Textbook Chapter 17.1 – Aggregate analysis

Textbook Chapter 17.2 – The accounting method

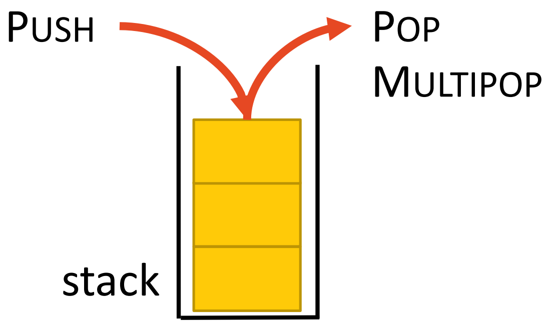
Textbook Chapter 17.3 – The potential method

Stack Operations

Stack Operations
Suppose that we apply a sequence of *n* operations on a data structure. What is the time complexity of the procedure?

- Implementation with an array or a linked list

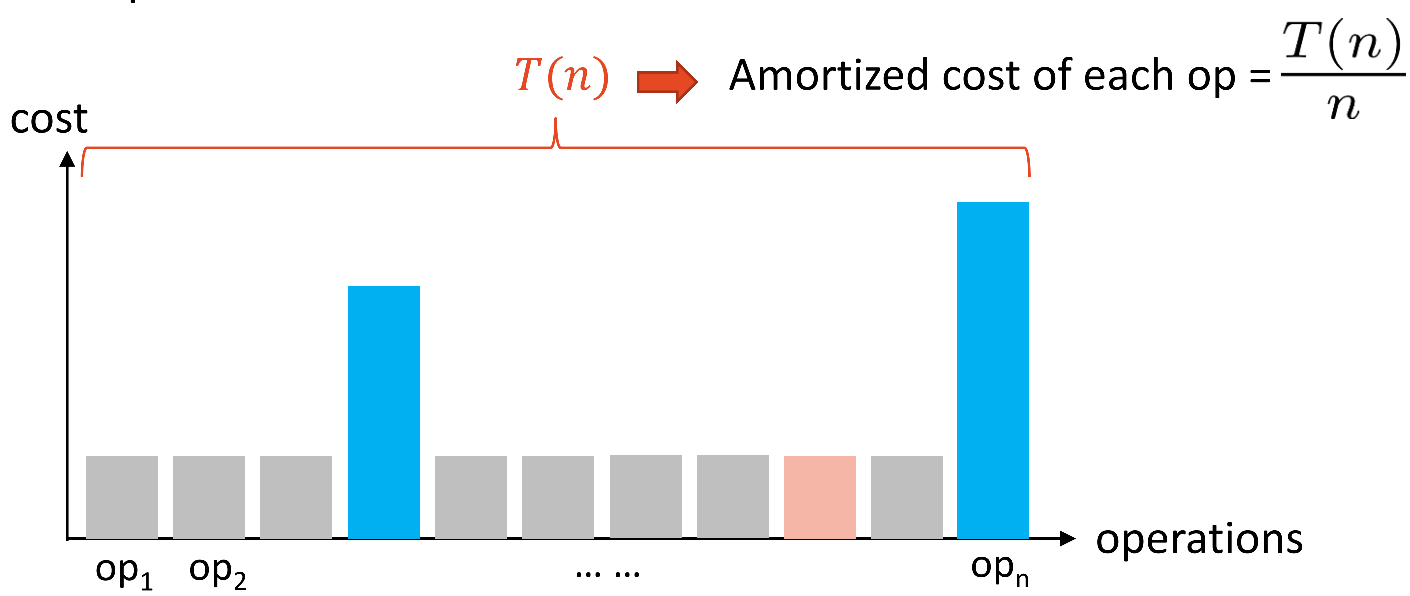
Operation Type	Cost
PUSH(S, x): inset an element x into S	$O(1)$
POP(S): pop the top element from S	$O(1)$
MULTIPOP(S, k): pop top k elements from S at once	$O(\min(\|S\|, k))$




```
MULTIPOP(S, k)
  while not STACK-EMPTY(S) and k > 0
    POP(S)
    k = k - 1
```

Aggregate Method (聚集法)

- Approach:
  1. Determine an upper bound $T(n)$ on the cost of any sequence of n operations
  2. Calculate the amortized cost per operation as $T(n)/n$
  3. All operations have the **same amortized cost**



Aggregate Method for Stack

- The number of each operation type

Operation Type	#Operations
PUSH(S, x): inset an element x into S	n_{push}
POP(S): pop the top element from S	n_{pop}
MULTIPOP(S, k): pop top k elements from S at once	$n_{multipop}$

} n

- These n_{pop} + $n_{multipop}$ operations together take at most $O(n_{push})$

Key idea: #pop elements ≤ #push operations/elements

- Total cost for n operations: $n_{push} \cdot O(1) + O(n_{push}) = O(n)$
- Amortized cost per operation: $\frac{O(n)}{n} = O(1)$

Another Thinking

- Once the push operation is taken, we prepare the additional cost for the future usage of multipop

Key idea: #pop elements ≤ #push operations/elements

$$n_{push} \cdot 2 \cdot O(1) = O(n)$$

Accounting Method (記帳法)

- Idea: save credits from the operations that take less cost for future use of operations that take more cost (針對使用花費較低的operations時先存錢未雨綢繆, 供未來花費較高的operations使用)
- Approach:
  1. Each operation is assigned a *valid* amortized cost
     - If amortized cost > actual cost, the difference becomes credit (存)
     - Credit is deposited in an object of the data structure
     - If amortized cost < actual cost, then withdraw (提) stored credits
  2. **Validity check**: ensure that every object has sufficient credit for any sequence of *n* operations
  3. Calculate total amortized cost based on individual ones

Accounting Method (記帳法)

- **Validity check**: ensure that every object has sufficient credit for any times of *n* operations (不能有赤字)
  - c_i: the actual cost of the i-th operation
  - $\hat{c}_i$: the amortized cost of the i-th operation

→ For all sequences of *n* operations, we require $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$

- **Accounting Method**
  - Each type of operations can have a different amortized cost
  - Assign valid amortized costs first and then compute T(n)

- **Aggregate Method**
  - Each type of operations have its actual cost
  - Compute amortized cost using T(n)

Accounting Method for Stack

1. Assign the amortized cost

Operation Type	Actual Cost	Amortized Cost
PUSH(S, x)	1	2
POP(S)	1	0
MULTIPOP(S, k)	min(|S|, k)	0

2. Show that for each object s.t. $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$
   - PUSH: the pushed element is deposited $1 credit
   - POP and MULTIPOP: use the credit stored with the popped element
   - There is always enough credit to pay for each operation
3. Each amortized cost is $O(1)$ → total amortized cost is $O(n)$

Potential Method (位能法)

- Idea: represent the prepaid work as "potential," which can be released to pay for future operations (the potential is associated with the whole data structure rather than specific objects)
- Approach:
  1. Select a **potential function** that takes the **current data structure state** as input and outputs a "potential level"
  2. **Validity check**: ensure that the potential level is nonnegative
  3. Calculate the amortized cost of each operation based on the potential function
  4. Calculate total amortized cost based on individual ones

- **Potential Method**
  - The data structure has credits

- **Accounting Method**
  - Each object within the data structure has its credit

Potential Method (位能法)

- Potential function Φ maps any state of the data structure to a real number
  - D_0: the initial state of data structure
  - D_i: the state of data structure after *i*-th operation
  - c_i: the actual cost of *i*-th operation
  - $\hat{c}_i$: the amortized cost of *i*-th operation, **defined** as $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

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

$$= \sum_{i=1}^{n} c_i + (\Phi(D_n) - \Phi(D_{n-1}) + \cdots + \Phi(D_1) - \Phi(D_0))$$

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

Potential Method (位能法)

- Total amortized cost

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

- To obtain an upper bound on the actual cost $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$
  - *Define* a potential function such that $\Phi(D_n) - \Phi(D_0) \geq 0$
  - Usually we set $\Phi(D_0) = 0, \Phi(D_i) \geq 0$

c_i: the actual cost of *i*-th operation
$\hat{c}_i$: the amortized cost of *i*-th operation

Potential Method for Stack

1. Define $\Phi(D_i)$ to be **the number of elements in the stack** after the *i*-th operation
2. Validity check:
   - The stack is initially empty → $\Phi(D_0) = 0$
   - The number of elements in the stack is always ≥ 0 → $\Phi(D_i) \geq 0$
3. Compute amortized cost of each operation:
   - PUSH(S, x): $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (|S| + 1) - |S| = 2$
   - POP(S): $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (|S| - 1) - |S| = 0$
   - MULTIPOP(S, k): $\hat{c}_i = 0$

Practice: justify why it is zero

4. All operations have *O*(*1*) amortized cost → total amortized cost is *O*(*n*)

22 Fibonacci Heap

Prim's Time Complexity

```
MST-PRIM(G, w, r) // w = weights, r = root
  for u in G.V
    u.key = ∞
    u.π = NIL
  r.key = 0
  Q = G.V
  while Q ≠ empty
    u = EXTRACT-MIN(Q)
    for v in G.adj[u]
      if v ∈ Q and w(u, v) < v.key
        v.π = u
        v.key = w(u, v) // DECREASE-KEY
```

Annotations: lines "for u in G.V" through "Q = G.V": $O(n)$; while loop: n times; EXTRACT-MIN: $O(\log n)$; for v in G.adj[u]: m times; DECREASE-KEY: $O(1)$

- Fibonacci heap (Textbook Ch. 19)
  - BUILD-MIN-HEAP: $O(n)$
  - EXTRACT-MIN: $O(\log n)$ (amortized)
  - DECREASE-KEY: $O(1)$ (amortized)
- Total complexity: $O(m + n \log n)$

Dijkstra's Time Complexity

```
DIJKSTRA(G, w, s)
  INITIALIZATION(G, s)
  S = empty
  Q = G.v // INSERT                $O(n)$
  while Q ≠ empty                  $n$ times
    u = EXTRACT-MIN(Q)             $O(\log n)$
    S = S∪{u}
    for v in G.adj[u]              $m$ times
      RELAX(u, v, w)
```

```
INITIALIZATION(G, s)
  for v in G.V        ┐
    v.d = ∞           │
    v.π = NIL         ├ $O(n)$
  s.d = 0             ┘
```

```
RELAX(u, v, w)                     $O(1)$
  if v.d > u.d + w(u, v)
    // DECREASE-KEY
    v.d = u.d + w(u, v)
    v.π = u
```

- Fibonacci heap (Textbook Ch. 19)
  - BUILD-MIN-HEAP: $O(n)$
  - EXTRACT-MIN: $O(\log n)$ (amortized)
  - DECREASE-KEY: $O(1)$ (amortized)
- Total complexity: $O(m + n \log n)$

25

Binary Counter

01100
10110
11110

Textbook Chapter 17.1 – Aggregate analysis

Textbook Chapter 17.2 – The accounting method

Textbook Chapter 17.3 – The potential method

Binary Counter

01100
10110
11110

Binary Counter
Suppose that a counter is initially zero. We increment the counter *n* times. How many bits are altered throughout the process?

- Implementation with a *k*-bit array

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

- Each operation takes $O(\log n)$ time in the worst case
- n operations take $O(n \log n)$ time

increment ↓

0	
1	1001
10	1010
11	1011
100	1100
101	1101
110	1110
111	1111
1000	10000

Aggregate Method for Binary Counter

Counter Value	A[3]	A[2]	A[1]	A[0]	Total Cost of First *n* Operations
0	0	0	0	0	0
1	0	0	0	**1**	1
2	0	0	**1**	**0**	3
3	0	0	1	**1**	4
4	0	**1**	**0**	**0**	7
5	0	1	0	**1**	8
6	0	1	**1**	**0**	10
7	0	1	1	**1**	11
8	**1**	**0**	**0**	**0**	15

flip every increment

flip every 2 increments

flip every 4 increments

flip every 8 increments

Aggregate Method for Binary Counter

- Total #bits flipping in *n* increment operations:

$$n + \frac{n}{2} + \frac{n}{4} + \cdots + \frac{n}{2k} < 2n$$

- Total cost of the sequence: $O(n)$
- Amortized cost per operation: $\frac{O(n)}{n} = O(1)$

Accounting Method for Binary Counter

1. Assign the amortized cost

Operation	Actual Cost	Amortized Cost
bit 0 → bit 1	1	2 (存$1到bit 1)
bit 1 → bit 0	1	0 (用掉存在bit 1裡面的$1)
increment	#flipped bits	2 for setting a bit to 1

2. Validity check:
   - Each bit 0 to bit 1, we save additional $1 in the bit 1
   - When bit 1 becomes to bit 0, we spend the saved cost
3. Each increment
   - Change many 1s to 0s → free
   - Change exactly a 0 to 1 → $O(1)$

- Each amortized cost is $O(1)$ → total amortized cost is $O(n)$

Accounting Method for Binary Counter

Counter Value	A[3]	A[2]	A[1]	A[0]	Total Cost of First *n* Operations
0	0	0	0	0	0
1	0	0	0	**1**	1
2	0	0	**1**	**0**	3
3	0	0	1	**1**	4
4	0	**1**	**0**	**0**	7
5	0	1	0	**1**	8
6	0	1	**1**	**0**	10
7	0	1	1	**1**	11
8	**1**	**0**	**0**	**0**	15

Amortized cost per operation is *O*(*1*)
Total amortized cost of *n* operations is *O*(*n*)

c_i: the actual cost of *i*-th operation
$\hat{c}_i$: the amortized cost of *i*-th operation

Potential Method for Binary Counter

1. Define $\Phi(D_i)$ to be **the number of 1s in the counter** after the *i*-th operation
2. Validity check:
   - The counter is initially zero → $\Phi(D_0) = 0$
   - The number of 1's cannot be negative → $\Phi(D_i) \geq 0$
3. Compute amortized cost of each INCREMENT:
   - Let $LSB_0(i)$ be the number of continuous 1s in the suffix
   - For example, LSB_0(01011011) = 2, and LSB_0(01011111) = 5

$$
\begin{aligned}
\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&= (LSB_0(i-1)+1) + (\Phi(D_{i-1}) - LSB_0(i-1) + 1) - \Phi(D_{i-1}) \\
&= 2
\end{aligned}
$$

4. All operations have *O*(*1*) amortized cost → total amortized cost is *O*(*n*)

Concluding Remarks

Aggregate method (聚集法)

- Determine an upper bound $T(n)$ on the cost over any sequence of n operations
- The average cost per operation is then $T(n)/n$
- All operations have the same amortized cost

Accounting method (記帳法)

- Each operation is assigned an amortized cost (may differ from the actual cost)
- Each object of the data structure is associated with a credit
- Need to ensure that every object has sufficient credit at any time

Potential method (位能法)

- Similar to accounting method; each operation is assigned an amortized cost
- The data structure as a whole maintains a credit (i.e., potential)
- Need to ensure that the potential level is nonnegative at any time

Three analyzing methods reach the same answer, and choose your preference

Question?

Important announcement will be sent to @ntu.edu.tw mailbox
& post to the course website

Course Website: http://ada.miulab.tw

Email: ada-ta@csie.ntu.edu.tw