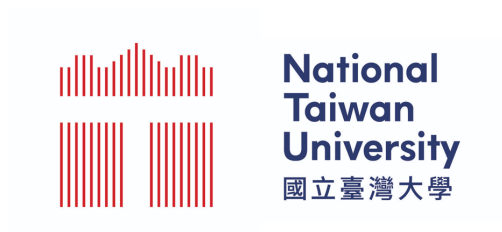


CSIE 2136 Algorithm Design and Analysis, Fall 2022



Amortized Analysis

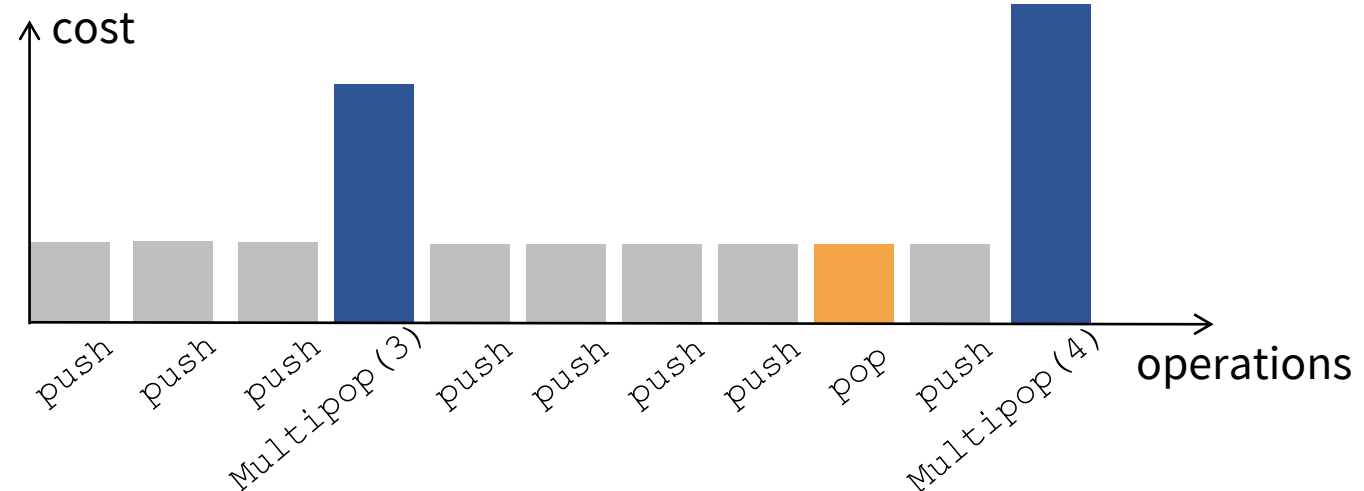
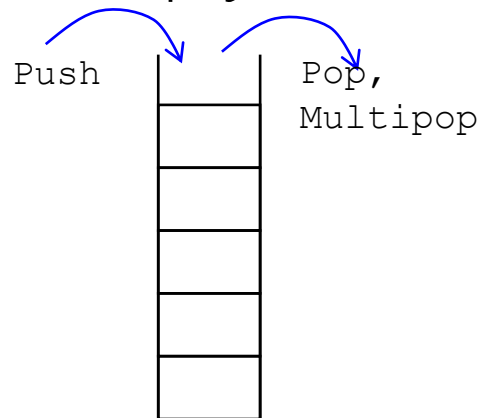
Hsu-Chun Hsiao

Agenda

- Why amortized analysis (均攤分析)
- Aggregate method (聚集方法)
- Accounting method (記帳方法) or banker's method
- Potential method (位能方法) or physicist's method
- Examples
 - #1: stack
 - #2: k-bit counter
 - #3: dynamic table (insertion only)
 - #4: dynamic table (insertion & deletion)

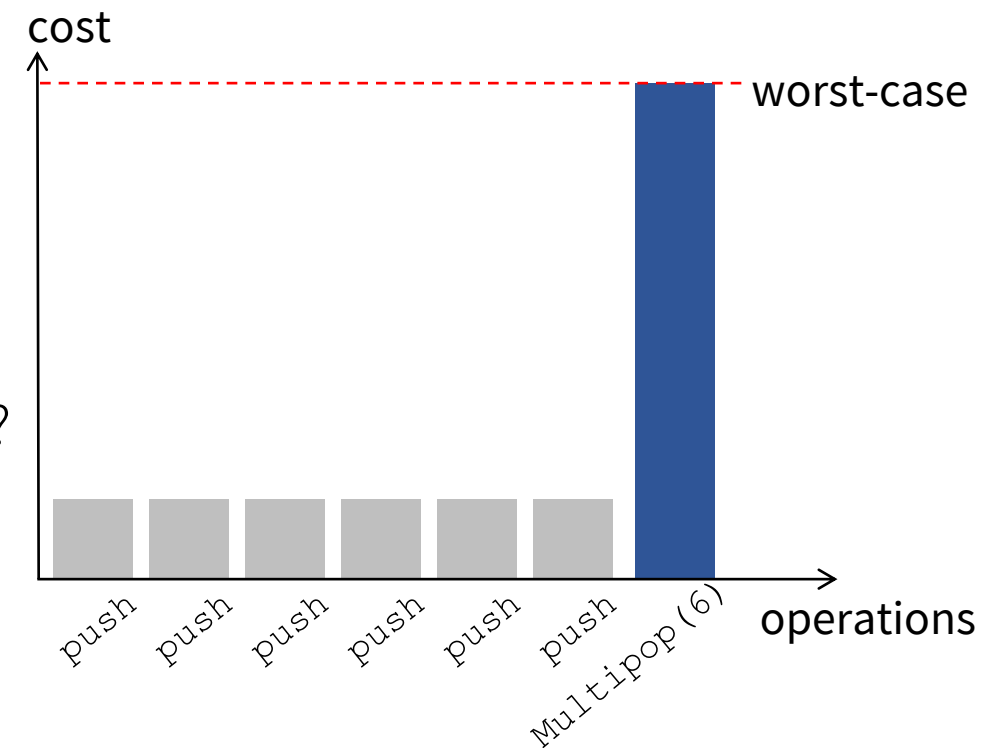
Why amortized analysis?

- Example: **stack** supports Push, Pop, Multipop operations
 - $\text{push}(S, x) : O(1)$
 - $\text{pop}(S) : O(1)$
 - $\text{multipop}(S, k) : O(\min\{|S|, k\})$
- What is the worst-case time of a sequence of n operations on an initially empty stack?



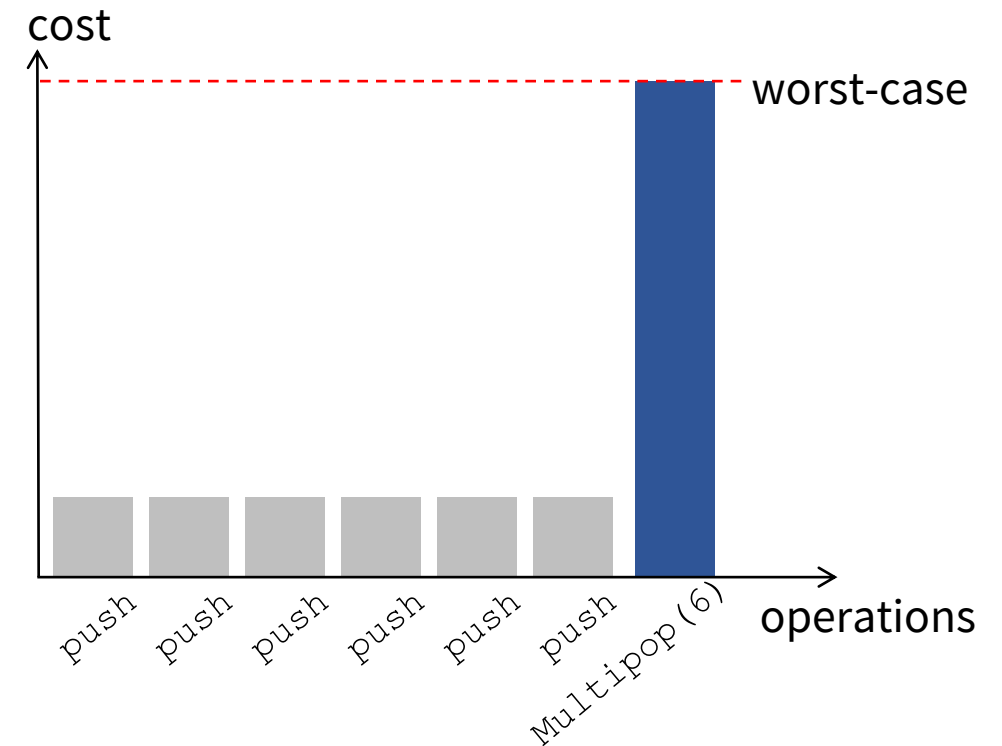
Worst-case analysis on individual operations may be loose

- What is the worst-case time of a sequence of n operations on an initially empty stack?
 - Worst-case time of the n^{th} operation = $\text{multipop}(S, n) = O(n)$
 - \Rightarrow Worst-case time of a sequence of n operations = $O(n^2)$
- However, this worst-case bound is **not tight**. Why?
 - The expensive `multipop` operation won't occur frequently!



Worst-case analysis on individual operations may be loose

- Given a sequence of n operations, their occurrences and costs may depend on others.
- Often happens when operating on the same data structure
- Thus, they should be analyzed together, not individually.



Goal of amortized analysis

- Obtain an asymptotic worst-case bound for a sequence of n operations

All of the valid operation sequences on stack when $n = 3$:

`push, push, pop`

`push, push, multipop(1)`

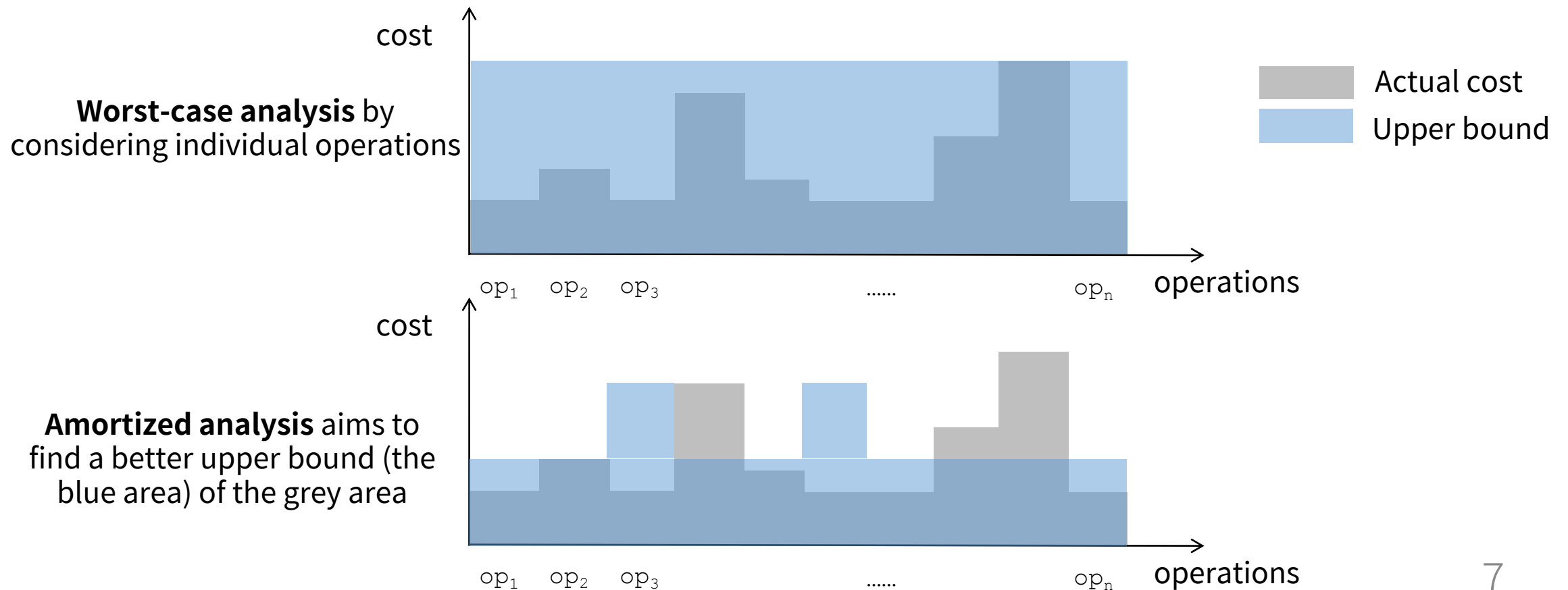
`push, push, multipop(2)`

`push, pop, push`

`push, multipop(1), push`

Goal of amortized analysis

- Obtain an **asymptotic worst-case bound** for a **sequence of n operations**



Amortized analysis: 3 common techniques

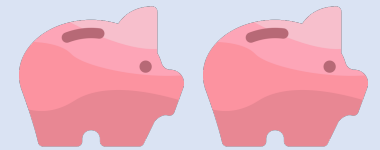
Aggregate method (聚集方法)

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



Accounting method (記帳方法)

- 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 (位能方法)

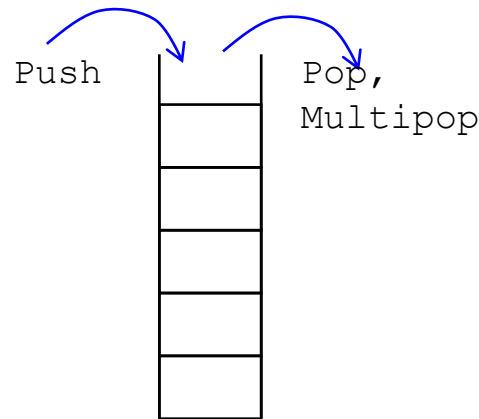
- Similar to accounting method; each operation is assigned an amortized cost
- The data structure as a whole maintains a credit (i.e., potential)
- Need to ensure that the potential level is nonnegative at any time



Note: these are **for analysis purpose only**, not for implementation!

Example #1: stack

- A stack is empty initially and supports three types of operations
- Implemented using an array or linked list



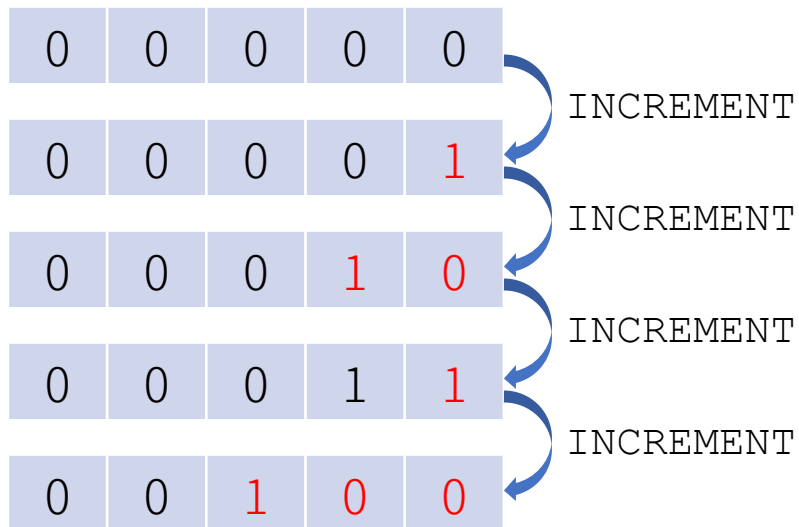
```
MULTIPOP (S, k) :
```

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

Operation type	Cost
Push (S, x)	$O(1)$
Pop (S)	$O(1)$
Multipop (S, k) : pop top k objects at once	$O(\min\{ S , k\})$

Example #2: k-bit counter

- Counts up from 0 by an operation, INCREMENT
- Implemented using a k -bit array
- If flipping one bit costs $O(1)$, INCREMENT costs $O(k)$



```
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
```

Example #3: dynamic table (insertion only)

- A table stores objects and supports `insertion`
 - Initially the table T is empty
 - Each operation inserts an object to T
 - **Double** its size when out of slots (Suppose resizing takes time linear to the number of objects currently in T); create a table of size 1 first if it's empty

i^{th} operation	Table	Actual cost (add + resize)
	[]	-
1: insertion	[1]	1
2: insertion	[1, 2]	1+1=2
3: insertion	[1, 2, 3, _]	1+2=3
4: insertion	[1, 2, 3, 4]	1
5: insertion	[1, 2, 3, 4, 5, _, _, _]	1+4=5

Example #4: dynamic table (insertion & deletion)

- A table stores objects and supports `insertion` and `deletion` of objects
 - Initially the table T is empty
 - Each operation inserts or deletes an object
 - **Double** its size when out of slots (i.e., load factor = 1)
 - **Halve** its size when the **load factor is $\leq 1/4$**
 - Suppose resizing takes time linear to the number of objects currently in T

More examples

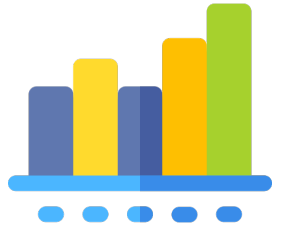
- Dynamic binary search [Problem 17.2]
- Queue with two stacks
- Disjoint-set implementation (linked list with weighted union, forest with union-by-rank and path compression) [Ch. 21.4]
- Splay tree
- Cuckoo hashing ^[1]

[1]: <https://web.stanford.edu/class/archive/cs/cs166/cs166.1146/lectures/13/Small13.pdf>

Aggregate Method (聚集方法)

Chapter 17.1

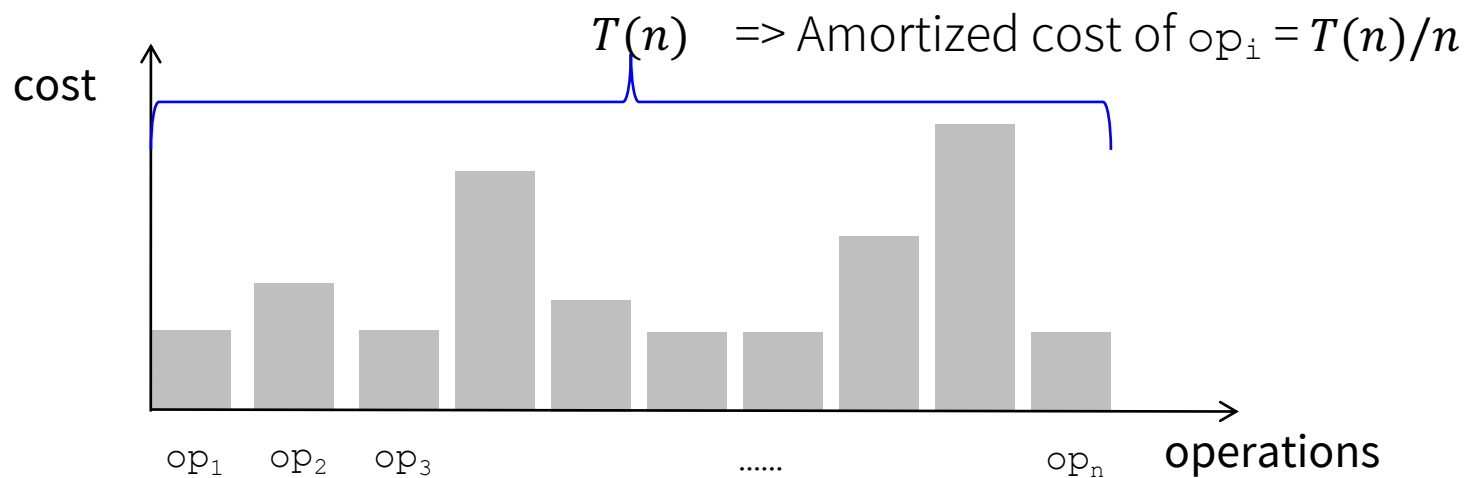
Aggregate method



Idea: 直接觀察 n 次操作的總花費的上限

Approach:

1. Determine an **upper bound** $T(n)$ on the cost of any sequence of n operations
 - **Validity check**: show that $T(n) \geq$ total actual cost
2. Calculate the amortized cost per operation as $T(n)/n$
 - All operations have the **same amortized cost**



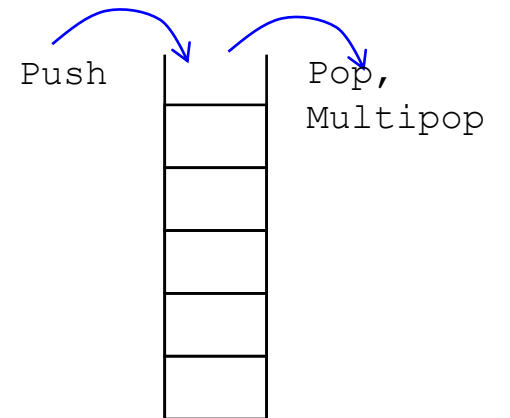
Aggregate method for stack

Observations:

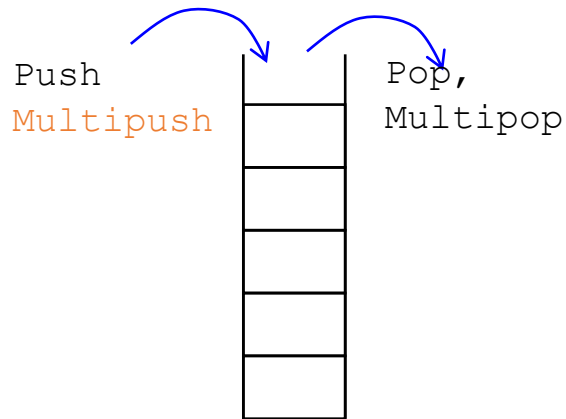
1. Total cost = # of popped objects + # of pushed objects
2. For a sequence of n operations, maximum # of pushed objects is n
3. # of popped objects \leq # of pushed objects
 - 出來的不可能比進去的多

=> Total cost for an entire sequence is $O(n)$

=> Amortized cost per operation is $O(n)/n = O(1)$



Consider a variant of the stack data structure supporting `mutlipush (S, X)`, where X is a set of objects. Does this stack variant also have $O(1)$ amortized cost per operation?



Operation type	Cost
<code>Push (S, x)</code>	$O(1)$
<code>Pop (S)</code>	$O(1)$
<code>Multipop (S, k)</code> : pop top k objects at once	$O(\min\{ S , k\})$
<code>Multipush (S, X)</code> : X is a set of objects	$O(X)$

Aggregate method for k-bit 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

Total cost = # of bit flips = # of RED in the table

Aggregate method for k-bit 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 k-bit counter

Observation: Total # of bit flips in n increment operations
$$= n + \lfloor \frac{n}{2} \rfloor + \lfloor \frac{n}{4} \rfloor + \cdots + \lfloor \frac{n}{2^{k-1}} \rfloor < 2n$$

=> Total cost of the sequence is $O(n)$

=> Amortized cost per operation is $O(n)/n = O(1)$

Consider a variant of the k -bit counter data structure where flipping the i th bit costs 2^i instead of $O(1)$.

Does this variant also have $O(1)$ amortized cost per operation?

If not, what is its amortized cost?

No.

Total # of bit flips in n increment operations

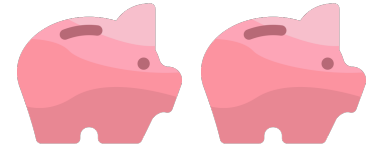
$$= n * 2^0 + \left\lfloor \frac{n}{2} \right\rfloor * 2^1 + \left\lfloor \frac{n}{4} \right\rfloor * 2^2 + \dots + \left\lfloor \frac{n}{2^{k-1}} \right\rfloor * 2^{k-1} \\ \leq kn$$

Amortized cost per operation is $O(k)$

Accounting Method (記帳方法)

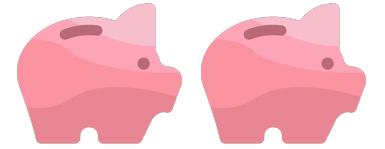
Chapter 17.2

Accounting method: idea



- 猜測每種操作的均攤費用
- 想像資料結構中每個物件有一個帳戶，初始金額為零
- 對某個物件進行操作時，
 - 若實際費用比均攤費用低，就存錢到其帳戶
 - 若實際費用比均攤費用高，從其帳戶裡拿錢補貼
- 若進行任意 n 個操作的過程不會造成任何帳戶透支，則為合理的均攤費用
- 跟 aggregate method 的主要差別
 - 不同操作可以有不同的均攤費用
 - 先猜測每種操作的均攤費用，再推算 $T(n)$

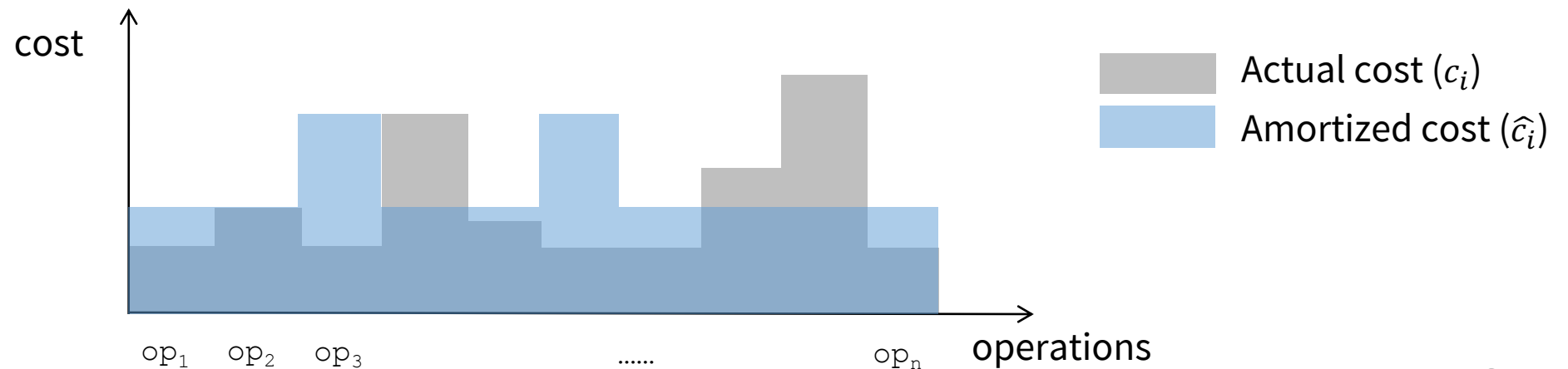
Accounting method: approach



1. **Guess** each operation type's amortized cost
2. **Validity check:** Check if the per-op amortized costs are valid
 - Assume every object initially has **credit = 0**
 - Let c_i and \hat{c}_i be the actual and amortized costs of the i^{th} op, respectively
 - Check if it has sufficient credit (≥ 0) for any sequence of n ops
 - If actual cost < amortized cost ($c_i < \hat{c}_i$), the difference becomes **credit**
 - If actual cost > amortized cost ($c_i > \hat{c}_i$), then **withdraw** stored credit
 - If the check fails, go back to Step 1
3. Calculate the total cost $T(n) = \sum_{i=1}^n \hat{c}_i$

Following the accounting method, show that $T(n) = \sum_{i=1}^n \hat{c}_i$ is an upper bound for the actual cost $\sum_{i=1}^n c_i$.

Hint: Show that for any sequence of n operations, if every object has sufficient credit (≥ 0) throughout the execution, then $\sum_{i=1}^n (\hat{c}_i - c_i) \geq 0$.



Accounting method for stack

1. Guess per-op amortized costs:

Operation	Actual cost	Amortized cost	Credit change
<code>push(S, x)</code>	1	2	存 1 元在 x 的帳戶裡
<code>pop(S)</code>	1	0	從 popped object 的帳戶領 1 元
<code>multipop(S, k)</code>	$\min\{ S , k\}$	0	從每個 popped object 的帳戶領 1 元

Accounting method for stack

1. Guess per-op amortized costs:

Operation	Actual cost	Amortized cost	Credit change
<code>push(S, x)</code>	1	2	存 1 元在 x 的帳戶裡
<code>pop(S)</code>	1	0	從 popped object 的帳戶領 1 元
<code>multipop(S, k)</code>	$\min\{ S , k\}$	0	從每個 popped object 的帳戶領 1 元

2. Validity check: Show that every object has credit ≥ 0

- **push**: the pushed object is deposited \$1 credit
- **pop and multipop**: use the credit stored with the popped object
- There is **always enough credit** to pay for each operation

Accounting method for stack

1. Guess per-op amortized costs:

Operation	Actual cost	Amortized cost	Credit change
<code>push(S, x)</code>	1	2	存 1 元在 x 的帳戶裡
<code>pop(S)</code>	1	0	從 popped object 的帳戶領 1 元
<code>multipop(S, k)</code>	$\min\{ S , k\}$	0	從每個 popped object 的帳戶領 1 元

2. Validity check: Show that every object has credit ≥ 0

- **push**: the pushed object is deposited \$1 credit
- **pop and multipop**: use the credit stored with the popped object
- There is **always enough credit** to pay for each operation

3. Per-op amortized costs are all $O(1)$, so total cost is $T(n) = O(n)$

Accounting method for k-bit counter

1. Guess per-op amortized costs:

Operation	Actual cost	Amortized cost	Credit change
INCREMENT	# of bits flipped	?	
i^{th} bit 0 \rightarrow 1	1	\$2	存 1 元在 i^{th} bit 裡
i^{th} bit 1 \rightarrow 0	1	\$0	用掉存在 i^{th} bit 的 1 元

Accounting method for k-bit counter

1. Guess per-op amortized costs:

Operation	Actual cost	Amortized cost	Credit change
INCREMENT	# of bits flipped	\$2	
i^{th} bit 0 \rightarrow 1	1	\$2	存 1 元在 i^{th} bit 裡
i^{th} bit 1 \rightarrow 0	1	\$0	用掉存在 i^{th} bit 的 1 元

2. Validity check:

- Counter 起始值為 $00\cdots 0$
- 每次 INCREMENT 都會把一個 0 設成 1，可能把很多 1 設成 0
- 可在被設為 1 的 bit 存一元
- 把 1 設成 0 時，花掉存在這個 bit 的一元即可

Accounting method for k-bit counter

1. Guess per-op amortized costs:

Operation	Actual cost	Amortized cost	Credit change
INCREMENT	# of bits flipped	\$2	
i^{th} bit 0 \rightarrow 1	1	\$2	存 1 元在 i^{th} bit 裡
i^{th} bit 1 \rightarrow 0	1	\$0	用掉存在 i^{th} bit 的 1 元

2. Validity check:

- Counter 起始值為 $00\cdots 0$
- 每次 INCREMENT 都會把一個 0 設成 1，可能把很多 1 設成 0
- 可在被設為 1 的 bit 存一元
- 把 1 設成 0 時，花掉存在這個 bit 的一元即可

3. Per-op amortized cost is $O(1) \Rightarrow$ total cost $T(n) = O(n)$

Potential Method (位能方法)

Chapter 17.3

Potential method: idea



- 想像將資料結構的狀態對應到位能，初始值為 0
- 對資料結構的狀態做操作時，
 - 若實際所需能量比均攤值低，將多餘能量的儲存成位能
 - 若實際所需能量比均攤值高，用儲存的位能補貼
- 若任意 n 個操作都有足夠位能，則從位能的變化算出個別操作的均攤費用
- 跟 accounting method 的主要差異：資料結構本身有 potential/credit，而不是每個物件都有 credit

Potential method: approach



1. Guess a **potential function** Φ that takes the current data structure state as input and outputs a potential level
2. **Validity check**: check if the potential level is never lower than the initial value after any sequence of n operations
 - Let D_i be the state of data structure after i^{th} operation
 - WLOG, **check if $\Phi(D_0) = 0$ and $\forall i = 1 \dots n, \Phi(D_i) \geq 0$**
 - If the check fails, go back to Step 1
3. Calculate the per-op amortized costs based on the potential function (see next slide)
4. Calculate the total cost based on per-op amortized costs $T(n) = \sum_{i=1}^n \hat{c}_i$

Potential function



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

Show that $T(n) = \sum_{i=1}^n \hat{c}_i$ is an upper bound for the actual cost $\sum_{i=1}^n c_i$.

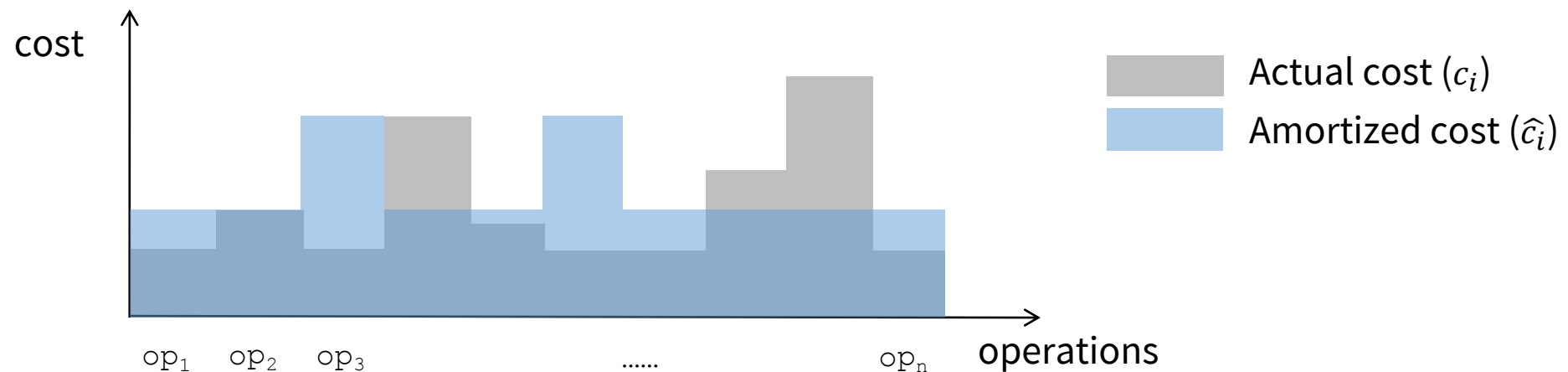
◦ Hint: Show that for any sequence of n ops, if $\Phi(D_i) \geq \Phi(D_0)$ throughout the execution, then $\sum_{i=1}^n (\hat{c}_i - c_i) \geq 0$.

◦ Based on the definition, we have

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

◦ Passing the validity check ensures $\Phi(D_n) \geq \Phi(D_0)$

◦ Thus, $\sum_{i=1}^n (\hat{c}_i - c_i) \geq 0$



Potential method for stack

1. Guess $\Phi(D_i)$ to be the # of objects in the stack after the i -th op
2. Validity check:
 - $\Phi(D_0) = 0$, because stack is initially empty
 - $\Phi(D_i) \geq 0$, because # of objects in stack is always ≥ 0

$\Phi(D_i)$: the # of objects in the stack after the i -th op

c_i : the actual cost of the i -th op

\hat{c}_i : the amortized cost of the i -th op, defined as $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

3. Compute per-op amortized cost:

- For `push(S, x)`: $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (|S| + 1) - |S| = 2$
- For `pop(S)`: $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (|S| - 1) - |S| = 0$
- For `multipop(S, k)`: $\hat{c}_i = 0$

4. All operations have $O(1)$ amortized cost, so total cost of n operations is $O(n)$

Q: justify why $\hat{c}_i = 0$ for `multipop(S, k)`

Potential method for k-bit counter

1. Guess $\Phi(D_i)$ to be the # of 1's in the counter after the i -th op
2. Validity check:
 - $\Phi(D_0) = 0$, because counter is initially all 0's
 - $\Phi(D_i) \geq 0$, because # of 1's cannot be negative

$\Phi(D_i)$: the # of 1's in the counter after the i -th op

c_i : the actual cost of the i -th op

\hat{c}_i : the amortized cost of the i -th op, defined as $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

3. Compute the amortized cost of INCREMENT:

- Let $LSB_0(x)$ be the index of the least significant 0 bit of x
- For example, $LSB_0(01011011) = 2$, and $LSB_0(01011111) = 5$
- $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
 $= (LSB_0(i-1) + 1) + (-LSB_0(i-1) + 1) = 2$

4. All operations have $O(1)$ amortized cost, so the total cost of n operations is $O(n)$

Example #3: dynamic table (insertion only)

- A table stores objects and supports `insertion`
 - Initially the table T is empty
 - Each operation inserts an object to T
 - Double its size when out of slots (Suppose resizing takes time linear to the number of objects currently in T); create a table of size 1 first if it's empty

i^{th} operation	Table	Actual cost (add + resize)
	[]	-
1: insertion	[a]	1
2: insertion	[a, b]	1+1=2
3: insertion	[a, b, c, _]	1+2=3
4: insertion	[a, b, c, d]	1
5: insertion	[a, b, c, d, e, _, _, _]	1+4=5

Example #3: dynamic table (insertion only)

- Aggregate method
 - The actual cost $c_i = \begin{cases} i, & \text{if } i - 1 \text{ is } 2\text{'s power} \\ 1, & \text{otherwise} \end{cases}$
 - $\Rightarrow T(n) = \sum_{i=1}^n c_i \leq n + \sum_{k=1}^{\lfloor \log_2(n-1) \rfloor} 2^k \leq 3n$
 - \Rightarrow The amortized cost of insertion is $O(1)$

i th operation	Table	Actual cost (add + resize)
	[]	-
1: insertion	[a]	1
2: insertion	[a, b]	1+1=2
3: insertion	[a, b, c, _]	1+2=3
4: insertion	[a, b, c, d]	1
5: insertion	[a, b, c, d, e, _, _, _]	1+4=5

Example #3: dynamic table (insertion only)

Use the accounting method

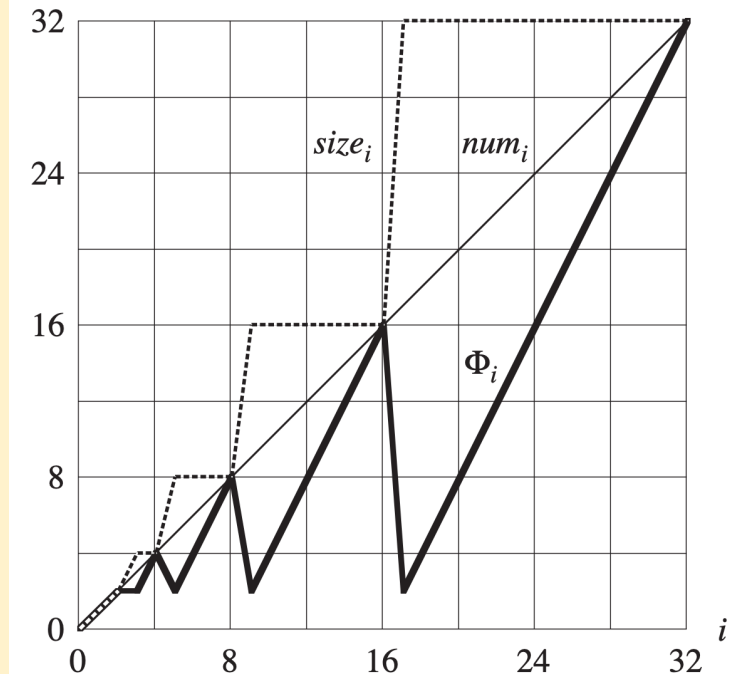
Hint: choose an amortized cost of 3

i^{th} operation	Table	Actual cost (add + resize)
	[]	-
1: insertion	[a]	1
2: insertion	[a, b]	$1+1=2$
3: insertion	[a, b, c, _]	$1+2=3$
4: insertion	[a, b, c, d]	1
5: insertion	[a, b, c, d, e, _, _, _]	$1+4=5$

Example #3: dynamic table (insertion only)

Use the potential method

Hint: Consider a potential function $\Phi(T) = 2 * T.num - T.size$, where $T.num$ is the number of inserted objects and $T.size$ is the table size



Example #4: dynamic table (insertion & deletion)

- A table stores objects and supports `insertion` and `deletion` of objects
 - Initially the table T is empty
 - Each operation inserts or deletes an object
 - Double its size when out of slots (i.e., load factor = 1)
 - Halve its size when the load factor is $\leq 1/4$
 - Suppose resizing takes time linear to the number of objects currently in T

Example #4: dynamic table (insertion & deletion)

i th operation	Table	Actual cost (add/delete + resize)
	[]	-
1: insertion	[a]	1
2: insertion	[a, b]	1+1=2
3: insertion	[a, b, c, _]	1+2=3
4: insertion	[a, b, c, d]	1
5: insertion	[a, b, c, d, e, _, _, _]	1+4=5
6: deletion	[a, b, c, d, _, _, _, _]	1
7: deletion	[a, b, c, _, _, _, _, _]	1
8: deletion	[a, b, _, _, _, _, _, _]	1
9: deletion	[a, _, _, _]	1+1=2
10: insertion	[a, b, _, _]	1

Example #4: dynamic table (insertion & deletion)

Use the aggregate method

Example #4: dynamic table (insertion & deletion)

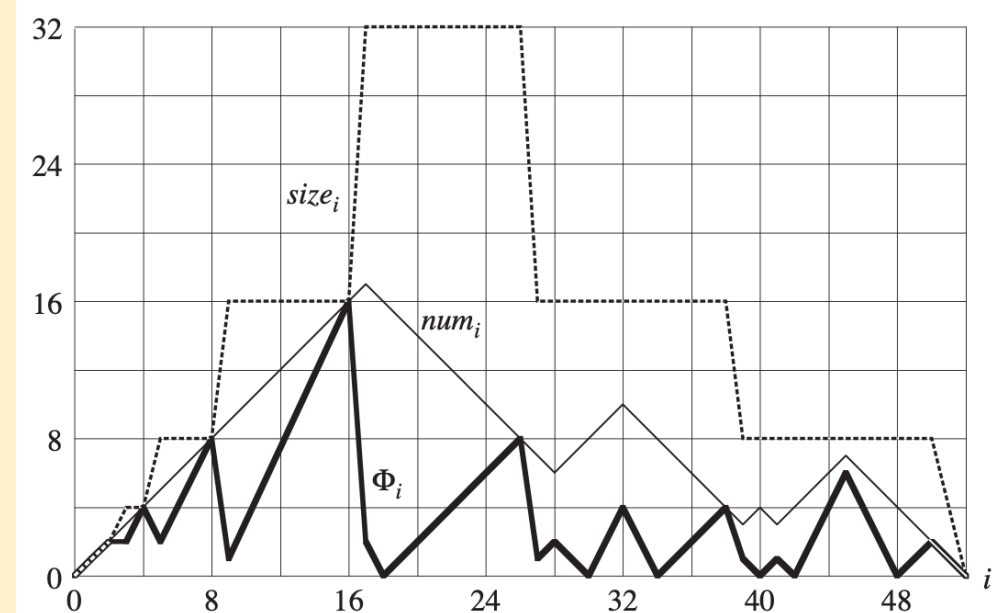
Use the accounting method

Hint: insertion's amortized cost = 3, deletion's amortized cost = 2

Example #4: dynamic table (insertion & deletion)

Use the potential method

Hint: Consider a potential function $\Phi(T) = 2 * num - size$ when $\alpha \geq \frac{1}{2}$ and $\Phi(T) = size/2 - num$ when $\alpha < \frac{1}{2}$, where num is the number of inserted objects, $size$ is the table size, $\alpha = \frac{num}{size}$ is the load factor.



Amortized analysis: 3 common techniques

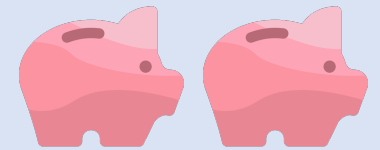
Aggregate method (聚集方法)

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



Accounting method (記帳方法)

- 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 (位能方法)

- 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



* 三種方法一般都能獲得相同的分析結果，可依個人偏好採用