A sequence of operations $OP_1, OP_2, OP_3, \ldots, OP_n$

Example: n = 10

OP_i	OP_1	OP_2	OP_3	OP_4	OP_5	OP_6	OP_7	OP_8	OP_9	OP_{10}
b_i	2	2	1	2	4	2	5	3	2	3
w_i	4	6	2	2	4	3	30	4	5	5
e_i	3	3	1.5	2	4	2.5	10	3.3	3	4

Total time complexity

best-case: $B(n) = \sum b_i$

worst-case: $T(n) = \sum w_i$

average-case: $A(n) = \sum e_i$

Single operation

best-case: 1

worst-case: 30

average-case: $A(n) / n$ (3.63)

amortized: $T(n) / n$ (6.5)

Note: amortized ≠ average-case

time t_i

push	1
push	1
push	1
push	1
push	1
⋮	⋮
push	1
push	1
multipop(S, n-1)	n-1

n (all operations above)

worst-case of t_i

* traditional:

⇨ $t_i \le n - 1$

⇨ $T(n) = \sum t_i \le n \times (n-1) = O(n^2)$

⇨ $T(n) / n = O(n^2/n) = O(n)$

correct, but not tight

	δ_i	$t_i = \lvert\delta_i\rvert$
push	+1	1
push	+1	1
push	+1	1
push	+1	1
pop	−1	1
push	+1	1
multipop 3	−3	3
push	+1	1
pop	−1	1
pop	−1	1

n

入 $\leq$ n

出 $\leq$ 入

$\Rightarrow T(n) = \sum t_i$ = 出 + 入

$\leq$ 2 入

$\leq$ 2n

k = 8 bits

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

Increment

Step 1.

From left to right, reset all 1's until a $b_i = 0$ is found

Step 2.

Set $b_i = 1$

Example

1 0 1 0 1 1 ~~0~~ ~~1 1 1~~

1 0 0 0

~~1 1 1 1 1 1 1 1 1 1~~

0 0 0 0 0 0 0 0 0 0

(overflow)

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

A[0] flips each time

A[1] flips every other time

A[2] flips every four times

⇨ A[i] flips every 2^i times

⇨ A[i] flips $\lfloor n/2^i \rfloor$ times

⇨ $T(n) = n + n/2 + n/4 + \ldots$
$\leq 2n$

Amortized cost : 一種表示法

用"每一個人的平均"去表示 "全部加總"

Example:

A copy machine	
OP_1	0.5~1 sec.
OP_2	0.5~1 sec.
⋮	⋮
OP_{499}	0.5~1 sec.
OP_{500}	0.5~1 + 120 sec.
OP_{501}	0.5~1 sec.
OP_{502}	0.5~1 sec.
⋮	⋮
OP_{1000}	0.5~1 + 120 sec.

Single operation

best-case: 0.5
(沒意義，廣告詞)

worst-case: 121 ⇨ $T(n) \le 121 * n$
(有品質保證，但太悲觀)

amortized: 1.24 ⇨ $T(n) \le 1.24 * n$
(這個表示法最好)

Note: amortized ≠ average-case

0.75+120*(1/500)

由學測表現評估兩所高中

1st:	75	74	best-case (沒意義，廣告詞)
last:	30	50	worst-case (太悲觀)
全校加總:	11000	6500	人海戰術 !? (不知人數，無法判斷)
全校平均:	55 (1100/200)	65 (6500/100)	amortized (這個表示法最好)

Example: A k-bit binary counter (single operation)

best-case : 1

worst-case : k ⇨ $T(n) \le k * n$

amortized : 2 (最 好 的 表 示 法) ⇨ $T(n) \le 2 * n$ (tighter!)

Why T(n)/n, not T(n)?

* Usually, we compare two DSs according to their single operation running time.

(How many times an OP will be performed is unknown.)

* Simple
① 1.24 sec. per page (✓)
② 620 sec. for 500 pages (×)
③ 1.24n sec. for n pages (×)

Aggregate Method

OP_1	t_1
OP_2	t_2
OP_3	t_3
⋮	⋮
OP_n	t_n

① Compute $T(n) = \sum t_i$

worst-case total time
(as tight as possible)

② Compute ①/n

Problem: It may be not easy to compute $\sum t_i$ tightly!

Accounting Method

Operation	Amortized cost	Actual cost	△credit
X	① Assign a_X	t_X	② How (specific object)
Y	a_Y	t_Y	
Z	a_Z	t_Z	

③ prove credit ≥ 0 (for any n)

$$\Rightarrow \underset{付}{\sum a_i} \geq \underset{花}{\sum t_i} \qquad (\underset{付}{\sum a_i} = \underset{花}{\sum t_i} + \underset{存}{C})$$

④ $\underset{花}{T(n)} = O(\underset{付}{\sum a_i})$

⑤ Compute ④/n

① Assign

	花 t_i	付 a_i
push A	1	2
push B	1	2
push C	1	2
push D	1	2
pop	1	0
push E	1	2
multipop 3	3	0
push F	1	2
pop	1	0

n

S
E \$1
C \$1
F \$1
A \$1

S

② How
(specific object)

* $C = |S| \geq 0$ ③

$\Rightarrow$ 付 $\geq$ 花

* $\underset{花}{T(n)} = O(\underset{付}{\sum a_i}) \leq 2n$ ④

* $T(n)/n = O(1)$ ⑤

(n operations)	time t_i	a_i
push	1	2
push	1	2
⋮	⋮	⋮
multipop(S, $n^{1/2}$)	$n^{1/2}$	0
⋮	⋮	⋮
multipop(S, n/10)	n/10	0
⋮	⋮	⋮
multipop(S, lg n)	lg n	0
⋮	⋮	⋮
multipop(S, $n^{1/3}$)	$n^{1/3}$	0
⋮	⋮	⋮
push	1	2

How to compute $T(n) = \sum t_i$?

Aggregate method

* traditional:

  $n \times O(n) = O(n^2)$

* With some efforts

  $\sum t_i \le$ total (IN + OUT) $\le 2n$

Accounting method

* pay $a_i = 2$ for push
* pay $a_i = 0$ for pop/multipop
* always credit ≥ 0
* $\sum t_i \le \sum a_i \le 2n$

	花 t_i	付 a_i
INCR	1 (s)	2
INCR	2 (r,s)	2
INCR	1 (s)	2
INCR	3 (r,r,s)	2
INCR	1 (s)	2
INCR	2 (r,s)	2
INCR	1 (s)	2
INCR	4 (r,r,r,s)	2
INCR	1 (s)	2

⇨ a_i = 2 or 0 (overflow)

(Each INCR sets at most one bit)

					$1	$1	$1	$1
A	0	0	0	0	1	1	1	1

\# of 1s in A

* $C = |A| \ge 0$ ⇨ 付 $\ge$ 花 ③
* $\underset{花}{T(n)} = O(\underset{付}{\sum a_i}) \le 2n$ ④
* $T(n)/n = O(1)$ ⑤

Aggregate Method

OP_1	t_1
OP_2	t_2
OP_3	t_3
⋮	⋮
OP_n	t_n

① Compute $T(n) = \sum t_i$

worst-case total time (as tight as possible)

② Compute ①/n

Problem: It may be not easy to compute $\sum t_i$ tightly!

Accounting Method

Operation	Amortized cost	Actual cost	△credit
X	a_X	t_X	
Y	a_Y	t_Y	
Z	a_Z	t_Z	

① Assign

② How (specific object)

③ prove credit ≥ 0 (for any n)

$$\Rightarrow \underset{付}{\sum a_i} \geq \underset{花}{\sum t_i} \quad (\underset{付}{\sum a_i} = \underset{花}{\sum t_i} + \underset{存}{C})$$

④ $\underset{花}{T(n)} = O(\underset{付}{\sum a_i})$

⑤ Compute ④/n

Problem: ① and ② are not easy!

Potential Method

$$D_0 \xrightarrow{OP_1} D_1 \xrightarrow{OP_2} D_2 \xrightarrow{OP_3} \cdots\cdots \xrightarrow{OP_n} D_n$$

① Define $\Phi(D_i)$ ~ credit after OP_i

② prove $\Phi(D_i) - \Phi(D_0) \geq 0$ for any i → $\sum a_i \geq \sum t_i$ （付 ≥ 花）

OP	Amortized cost	Actual cost	Δcredit
X	a_X	t_X	
Y	③ compute a_Y	t_Y	$\Phi(D_i) - \Phi(D_{i-1})$
Z	a_Z	t_Z	

$a_i = t_i + [\Phi(D_i) - \Phi(D_{i-1})]$
付　花　存款變化

④ $T(n) = O(\sum a_i)$

⑤ Compute ④/n

a_i	t_i	Δcredit	D_i	$\Phi(D_i)$
			D_0	200
7	4	3	D_1	203
6	2	4	D_2	207
7	13	-6	D_3	201
6	1	5	D_4	206
6	8	-2	D_5	204
	$\sum t_i = ?$			

① ③

$\sum a_i \geq \sum t_i$ （付 ≥ 花）

② prove $\Phi(D_i) - \Phi(D_0) \geq 0$

$a_i = t_i + [\Phi(D_i) - \Phi(D_{i-1})]$
付　花　存款變化

④ $T(n) \leq \sum a_i \leq 7n$

⑤ $T(n)/n \leq 7n/n \leq 7$

Deletion: Implementation 1

a	b	c	d	e	f k	g	h	i	j						
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Delete c (put slot 3 into the free list)

Delete f (put slot 6 into the free list)

Insert k (extract a free slot 6)

Deletion: Implementation 2

a	b	c	d	e	f	g	h	i k	j						

Delete c

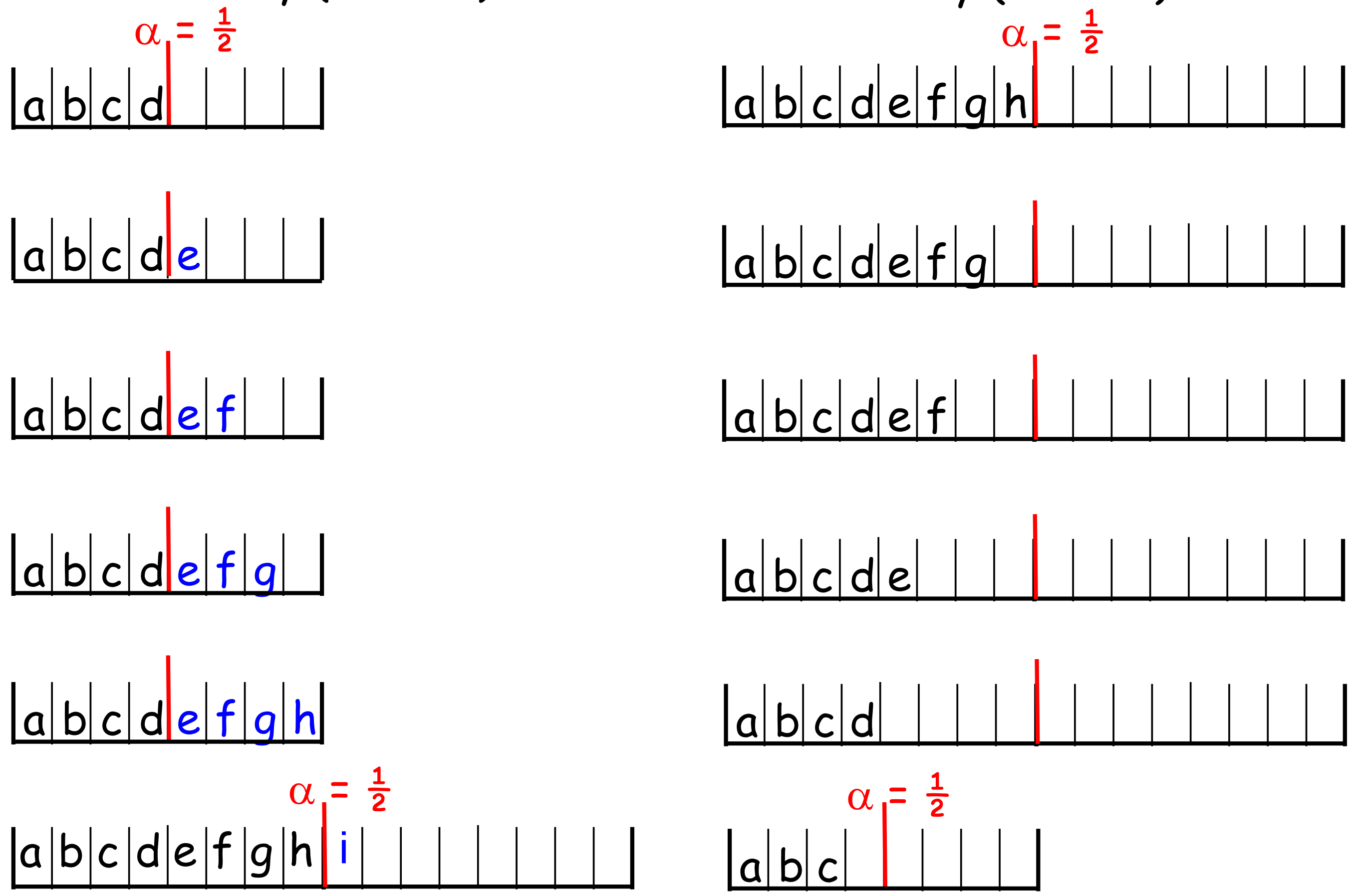
Delete f

Insert k

For convenience, assume that
deletion always removes the last data item.

Insertion Only ($\alpha \ge 1/2$)

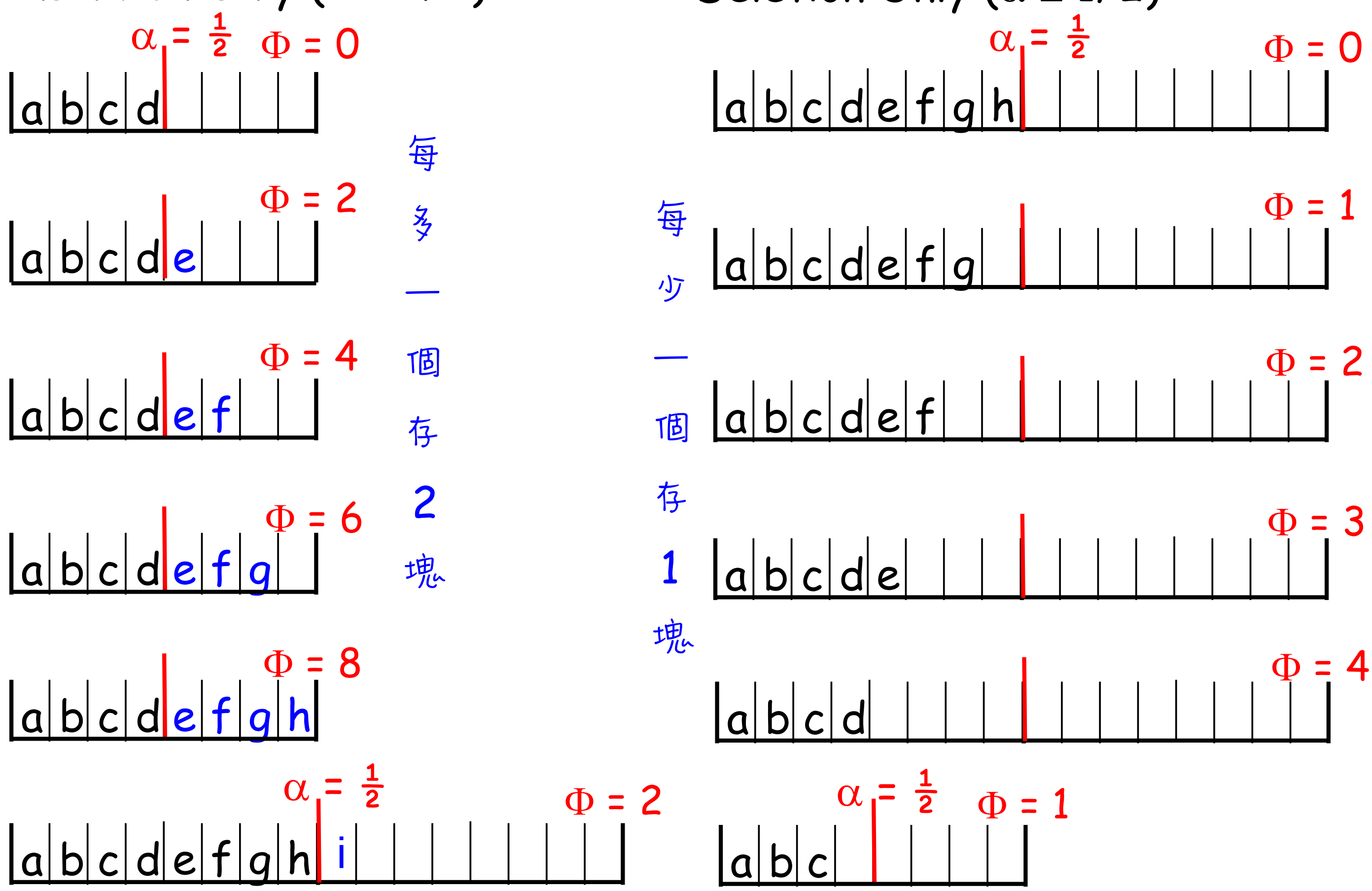
Deletion Only ($\alpha \le 1/2$)

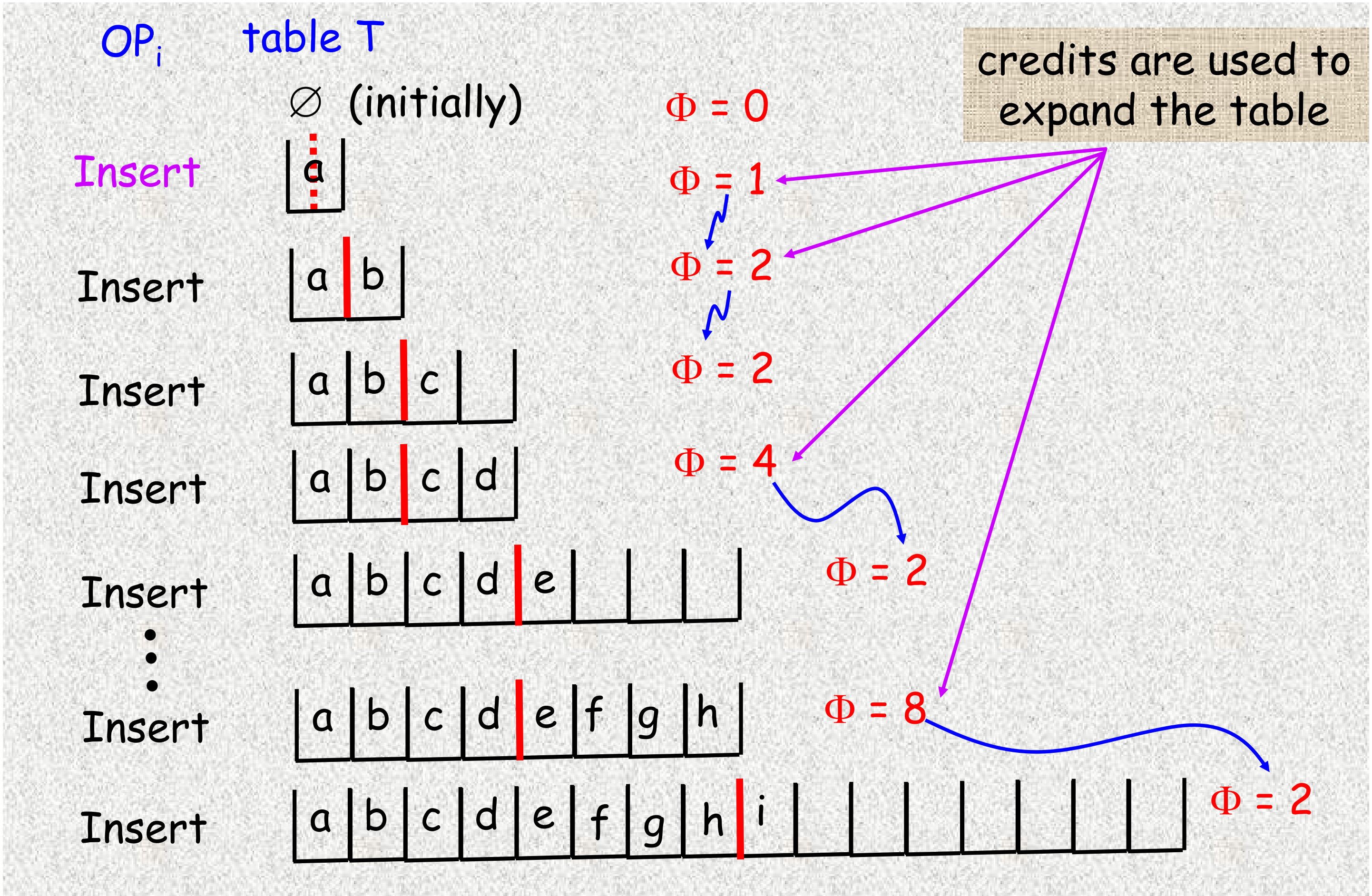


17-10a

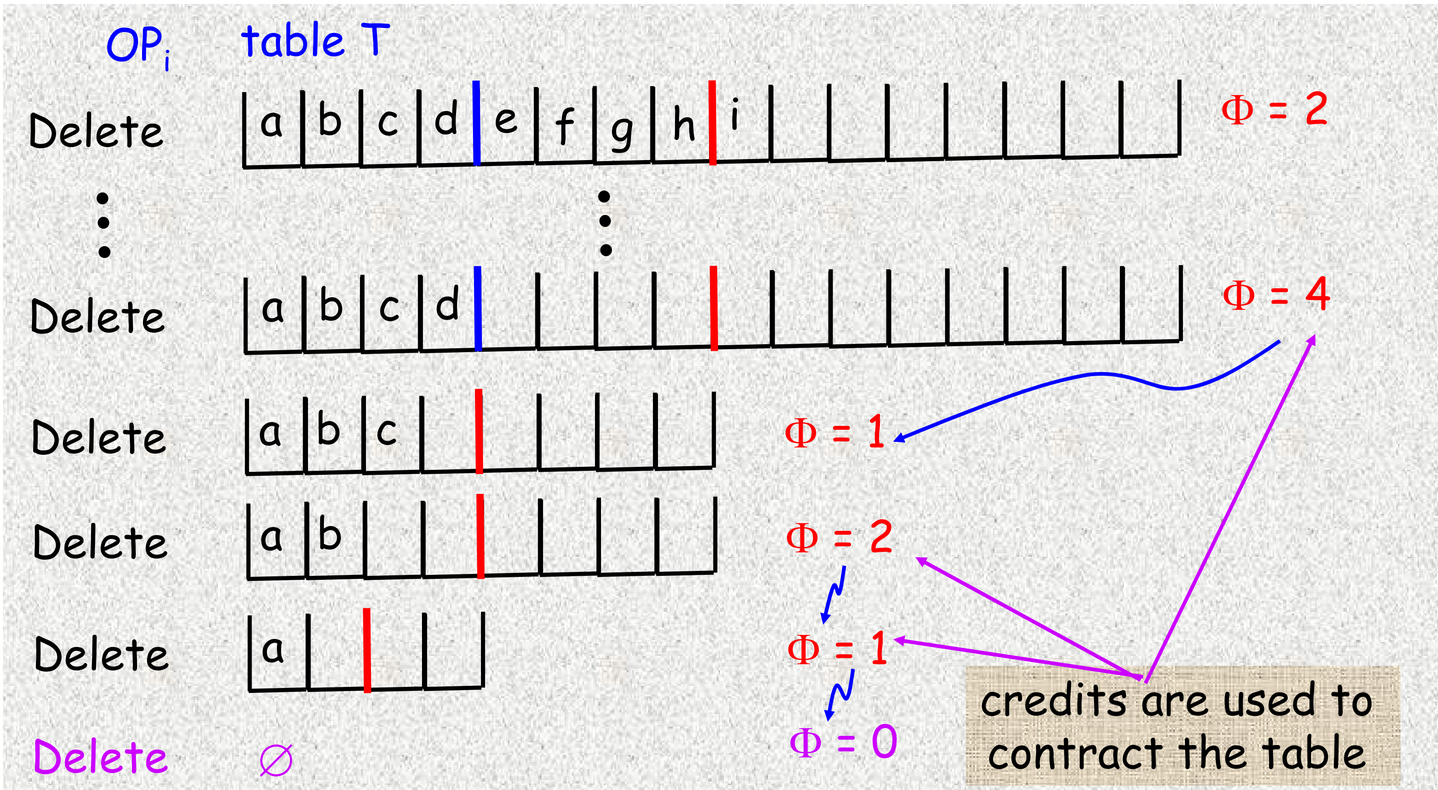
Insertion Only ($\alpha \ge 1/2$)

Deletion Only ($\alpha \le 1/2$)



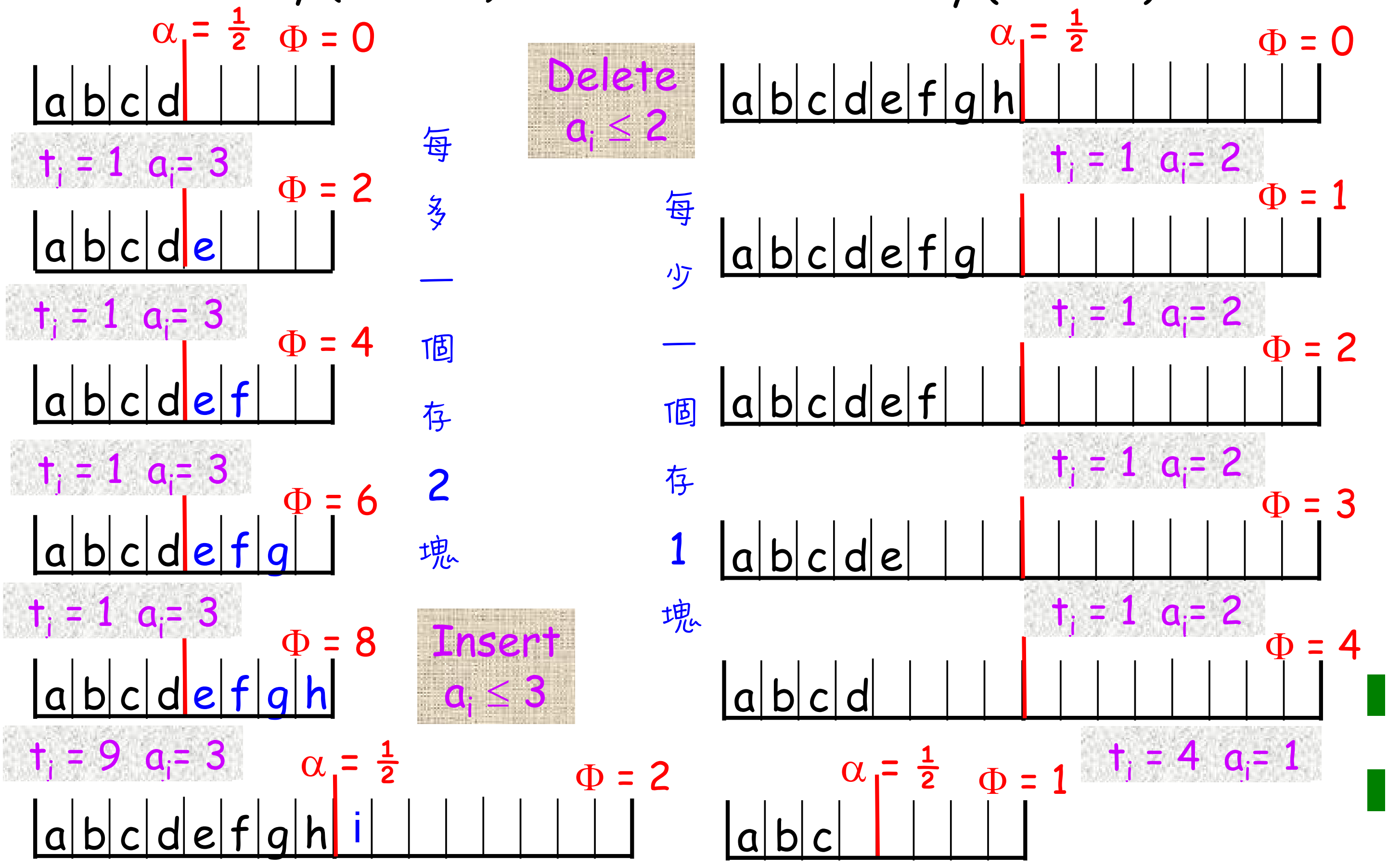

OP$_i$
table T
∅ (initially)
Insert
Insert
Insert
Insert
Insert
Insert
Insert
a
a b
a b c
a b c d
a b c d e
a b c d e f g h
a b c d e f g h i
Φ = 0
Φ = 1
Φ = 2
Φ = 2
Φ = 4
Φ = 2
Φ = 8
Φ = 2
credits are used to expand the table

17-11x


OP$_i$
table T
Delete
Delete
Delete
Delete
Delete
Delete
a b c d e f g h i
a b c d
a b c
a b
a
∅
Φ = 2
Φ = 4
Φ = 1
Φ = 2
Φ = 1
Φ = 0
credits are used to contract the table

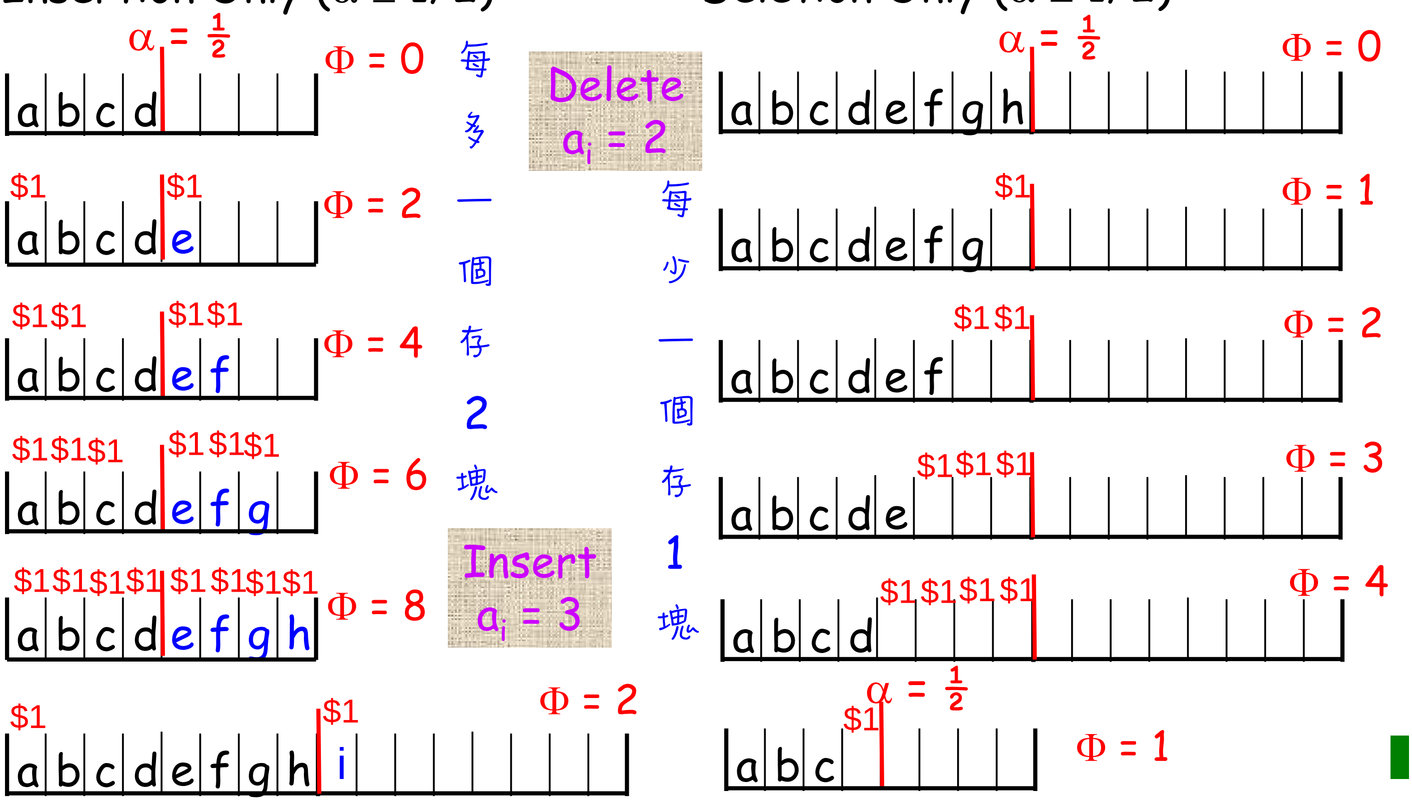
17-11y

Insertion Only ($\alpha \geq 1/2$) / Deletion Only ($\alpha \leq 1/2$)



Another viewpoint -- Accounting Method

Insertion Only ($\alpha \geq 1/2$) / Deletion Only ($\alpha \leq 1/2$)



Amortized cost：一種表示法

用"每一個人的平均"去表示"全部加總"

Example:

A copy machine

OP_1	0.5~1 sec.
OP_2	0.5~1 sec.
⋮	⋮
OP_{499}	0.5~1 sec.
OP_{500}	0.5~1 + 120 sec.
OP_{501}	0.5~1 sec.
OP_{502}	0.5~1 sec.
⋮	⋮
OP_{1000}	0.5~1 + 120 sec.

Single operation

best-case: 0.5
(沒意義，廣告詞)

worst-case: 121
(有品質保證，但太悲觀)

amortized: 1.24
(這個表示法最好)

Another Example: 期中考後你一直擔心沒考好
(你都會寫但擔心可能不到六十分)

發考卷前老師想安慰你說：別擔心，

(1) 最高分有 100 分 (best-case)

(2) 最低分有 10 分 (worst-case)

(3) 全班得 5000 分 (沒感覺)

(4) 平均每一個人得 50 分 (有感覺了，我應該比平均好)

(但是別高興，如果有人得100分，你可能只有0分)

Amortized: 全部加起來一樣，有人多就有人少

- 人快就有人慢，有人運氣好就有人倒霉
- 如果老師一個一個報成績：

聽到低分就高興；聽到高分就生氣

(表示你直的懂 amortized)

17-13y

17-13a

Amortized cost：一種表示法

用"每一個人的平均"去表示 "全部加總"

Selection of a DS (for a library)

	worst-case	amortized	
DS_1	O(n)	O(1)	→ good for lib (or a group of users)
DS_2	O(lgn)	O(lgn)	

O(lgn) → good for a single user

* 如果需要多次呼叫，amortized 比 worst-case 有意義
* single-operation worst-case 好 ⇨ 每次都很好 (快)
* single-operation amortized cost 好

⇨ 整體表現好 (偶爾很差 (慢))

⇨ 快 快 … 快，很 慢，快，快 … 快，很 慢，…

(存 存 … 存，花，存，存，… 存，花，…)

Why amortized analysis?

for i = 1 to n **do**
  OP_i

Time: $t_1, t_2, t_3, \ldots, t_n$

(1) Analysis

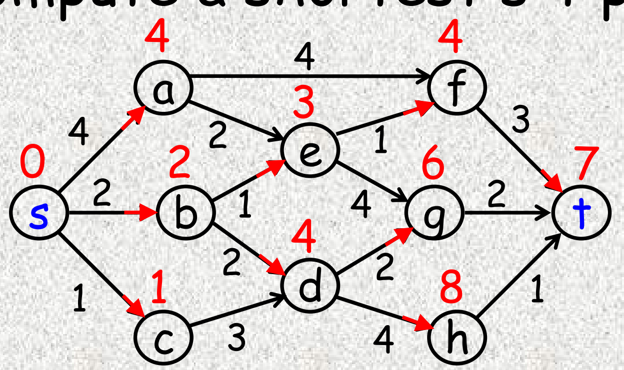
(a) Traditional

√ $t_i = O(f(n))$

√ $T(n) = n \times O(f(n)) = O(n \times f(n))$

(b) Amortized: compute T(n) directly

Compute a shortest s-t path on a DAG



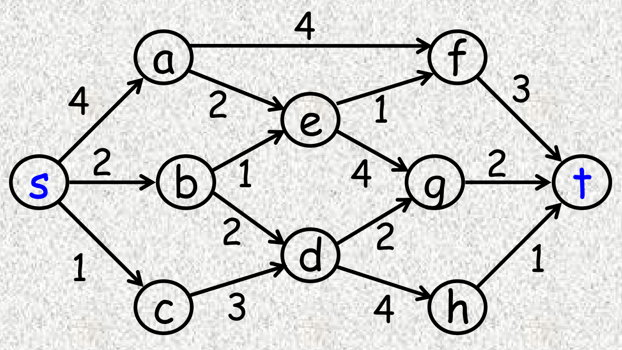
$n = |V|, m = |E|$

for i = 1 **to** n **do** (DP, from left to right)

  compute $d(s, v_i)$ — OP_i, t_i = in_degree(v_i)

Time: $\sum_{v_i} O(\text{in\_deg}(v_i))$

Compute a shortest s-t path on a DAG



$n = |V|, m = |E|$

worst-case of t_i

Time: $\sum_{v_i} O(in\_deg(v_i)) = \sum_{v_i} O(n) = O(n) \times O(n) = O(n^2)$

Time: $\sum_{v_i} O(in\_deg(v_i)) = O(n + \text{total-in-degree})$

$= O(n + m)$ total-in-degree = E

17-13z

17-13b

Why amortized analysis?

for i = 1 to n **do**
OP_i

Time: $t_1, t_2, t_3, \ldots, t_n$

(1) Analysis

(a) Traditional

√ $t_i = O(f(n))$

√ $T(n) = n \times O(f(n)) = O(n \times f(n))$

(b) Amortized: compute T(n) directly

√ aggregate method

√ accounting method

√ potential method

用規律方式付錢，再用全部付款的去估實際花費

(e.g., 一年生活費?)

(use when most t_i are small and f(n) occurs only a few times)

for i = 1 to n **do**
OP_i

Time: $t_1, t_2, t_3, \ldots, t_n$

(2) Design of algorithms or data structures

(a) Traditional:
- √ Try to reduce f(n) (worst case of each t_i)
- √ Every t_i should be small

(b) Amortized:
- √ Try to reduce T(n) (overall running time)
- √ Most t_i are small
- √ But, allow a few t_i to be large
- √ Have more flexibility in designing
- √ Have more chance to get a better T(n)
  (See CH21: disjoint sets)

Data Structures for Disjoint Sets (Ch21)

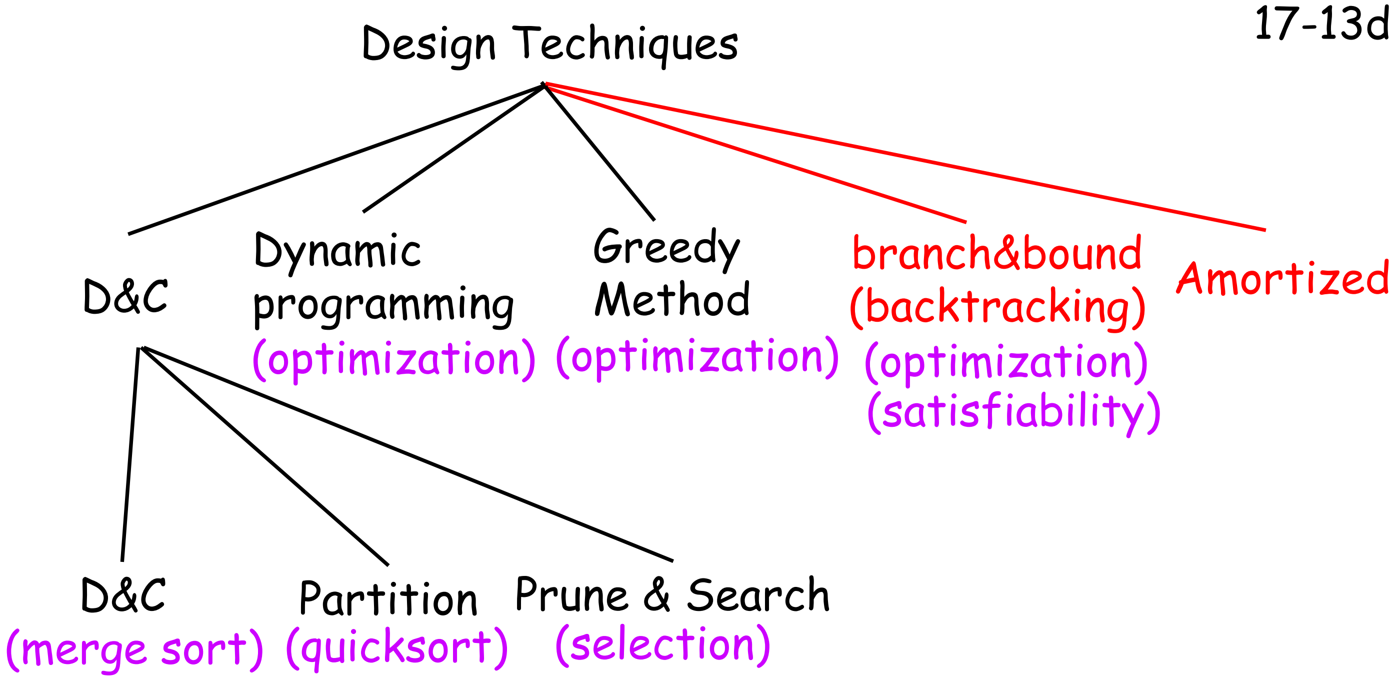
try to reduce worst-case

try to reduce overall time

Procedure	2-3 tree
single operation worst-case	O(lg n)
single operation amortized	
☆overall time	O(n lg n)

much simpler

better & simpler


Design Techniques
D&C
Dynamic programming
(optimization)
Greedy Method
(optimization)
branch&bound
(backtracking)
(optimization)
(satisfiability)
Amortized
D&C
(merge sort)
Partition
(quicksort)
Prune & Search
(selection)