

# **Chapter 17**

# **Amortized Analysis**

平摊（摊还、分摊、分期）分析

# How to analyze algorithms

MERGE-SORT( $A, p, r$ ) sorts the elements in the subarray  $A[p .. r]$ , initially,  $p = 1, r = n$ .

## MERGE( $A, p, q, r$ )

1	$n_1 \leftarrow q-p+1$	$T_{\text{sub}}(n)$
2	$n_2 \leftarrow r-q$	1
3	create arrays $L[1 .. n_1+1]$ and $R[1 .. n_2+1]$	1
4	<b>for</b> $i \leftarrow 1$ <b>to</b> $n_1$	$n_1+1$
5	$L[i] \leftarrow A[p+i-1]$	$n_1$
6	<b>for</b> $j \leftarrow 1$ <b>to</b> $n_2$	$n_2+1$
7	$R[j] \leftarrow A[q+j]$	$n_2$
8	$L[n_1+1] \leftarrow \infty$	1
9	$R[n_2+1] \leftarrow \infty$	1
10	$i \leftarrow 1$	1
11	$j \leftarrow 1$	1
12	<b>for</b> $k \leftarrow p$ <b>to</b> $r$	$r-p+2$
13	<b>if</b> $L[i] \leq R[j]$	$r-p+1$
14	$A[k] \leftarrow L[i]$	$x$
15	$i \leftarrow i+1$	$x$
16	<b>else</b> $A[k] \leftarrow R[j]$	$r-p+1-x$
17	$j \leftarrow j+1$	$r-p+1-x$

## MERGE-SORT( $A, p, r$ )

	$T(n)$
<b>1 if</b> $p < r$	1
2 $q \leftarrow \lfloor (p+r)/2 \rfloor$	1
3     MERGE-SORT( $A, p, q$ )	$T(n/2)$
4     MERGE-SORT( $A, q+1, r$ )	$T(n/2)$
5 <b>MERGE</b> ( $A, p, q, r$ )	$n$

(1) Sum

$$T_{\text{sub}}(n) = \sum \dots = n$$

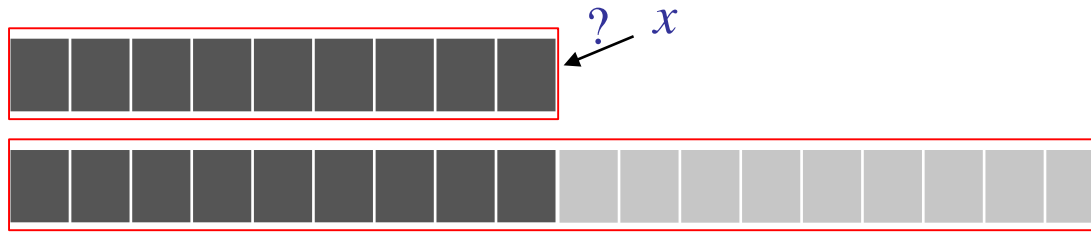
(2) Solving Recursive Equations

$$T(n) = 2T(n/2) + n \quad \text{//master theorem}$$

(3) Probability Analysis (quick sort)

(4) ...

# An example: Dynamic tables



```
void * malloc(size_t size);
```

```
c : malloc/free
```

```
c++: new/delete
```

- We might allocate space for a table, only to find out later that it is not enough. We must then reallocate the table with a larger size and copy all objects stored in the original table over into the new, larger table.



- Similarly, if many objects have been deleted from the table, it may be worthwhile to reallocate the table with a smaller size.



- We assume that the dynamic table supports the operations TABLE-INSERT and TABLE-DELETE.
  - ◆ TABLE-INSERT inserts into the table an item that occupies a single slot, that is, a space for one item.
  - ◆ Likewise, TABLE-DELETE removes an item from the table, thereby freeing a slot.

# Amortized Analysis

- ❑ In an amortized analysis, we average the time required to perform a sequence of data-structure operations over all the operations performed.  
在多个操作中，求一个操作的平均时间
- ❑ With amortized analysis, we can show that the average cost of an operation is small, if we average over a sequence of operations, even though a single operation within the sequence might be expensive.
- ❑ Amortized analysis differs from average-case analysis in that **probability is not involved**; an amortized analysis guarantees the average performance of each operation in the worst case.  
平摊分析不需要考虑输入的概率密度
- ❑ **Amortized cost**, 分摊消费：在一个操作上的平均消费 (**cost**)

# Amortized Analysis

---

17.1 Aggregate analysis (聚集分析)

17.2 The accounting method (记账法)

17.3 The potential method (势能法)

## 17.1 Aggregate analysis

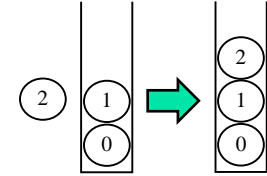
---

- In aggregate analysis, we show that for all  $n$ , a sequence of  $n$  operations takes worst-case time  $T(n)$  in total.
- In the worst case, the **average cost**, or **amortized cost**, per operation is therefore  $T(n)/n$ .

# (1) Stack operations

- $\text{PUSH}(S, x)$ : pushes object  $x$  onto stack  $S$ .
- $\text{POP}(S)$ : pops the top of stack  $S$  and returns the popped object. Calling POP on an empty stack generates an error.
- $\text{MULTIPOP}(S, k)$ : removes the  $k$  top objects of stack  $S$ , popping the entire stack if the stack contains fewer than  $k$  objects. (The stack has  $s$  objects.)

$O(1)$



$O(1)$

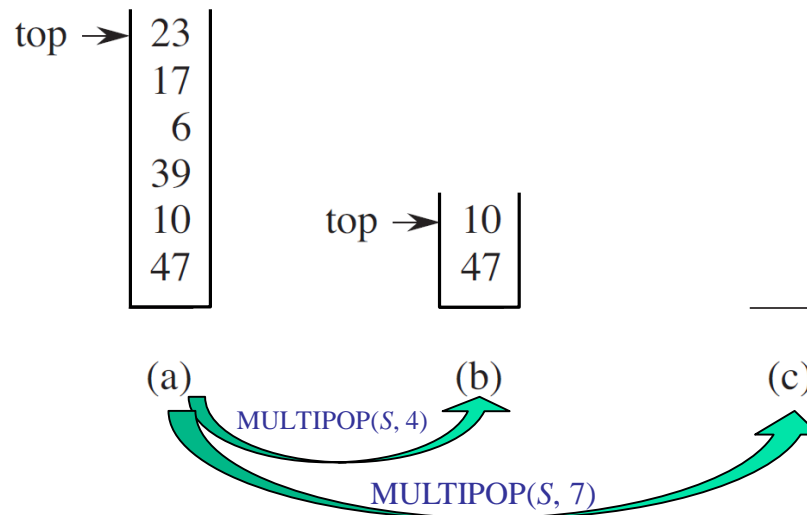
$O(\min(s, k))?$

## (1) Stack operations

**MULTIPOP( $S, k$ ):** removes the  $k$  top objects of stack  $S$ , popping the entire stack if the stack contains fewer than  $k$  objects. (The stack has  $s$  objects.)

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

$O(\min(s, k))$





# (1) Stack operations

Considering a sequence of  $n$  PUSH, POP, and MULTIPOP operations on an initially empty stack.

STACK( $S, n$ )

```
1  $S = \text{NULL}$ 
2 for  $i \leftarrow 1$  to  $n$ 
3   One of ( PUSH( $S, i$ ), POP( $S$ ), MULIPOP( $S, k$ ) )
```

MULTIPOP( $S, k$ )

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

- Running time?

$O(n^2)$

$O(\min(s, k))$

worst-case:  $O(n)$

- Correct. Not tight.
- Running time tightly?  $O(n)$
- We can pop each object from the stack at most once for each time we have pushed it onto the stack. Therefore, the number of times that POP can be called on a nonempty stack, including calls within MULTIPOP, is at most the number of PUSH operations, which is at most  $n$ .

push一个对象后，至多能被pop一次。但至多 $n$ 次push。

## (1) Stack operations

```
STACK( $S, n$ )  
1  $S = \text{NULL}$   
2 for  $i \leftarrow 1$  to  $n$   
3   One of ( PUSH( $S, i$ ), POP( $S$ ), MULIPOP( $S, k$ ) )
```

- Running time tightly  $O(n)$
- The average cost of an operation is  $O(n)/n = O(1)$
- In aggregate analysis, the **amortized cost** of each operation is the average cost.
- For STACK operations, the average cost (the running time) of a stack operation is  $O(1)$ , we did not use probabilistic reasoning.
- Dividing this total cost by  $n$  yielded the average cost per operation, or **the amortized cost**.  
平摊消费（每一个操作的消费）是总消费除以总操作数，即  $T(n)/n$

## (2) Incrementing a binary counter

Consider the problem of implementing a  $k$ -bit binary counter that counts upward from 0.



## (2) Incrementing a binary counter

- We use an array  $A[0 .. k-1]$  of bits as the counter. (使用位数组作为计数器)
- A number  $x$  that is stored in the counter has its lowest-order bit in  $A[0]$  and highest-order bit in  $A[k-1]$ , so that
$$x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$$
- Initially,  $x = 0$ , and thus  $A[i] = 0$  for all  $i$ .
- To **add 1** (modulo  $2^k$ ) to the value in the counter, we use the following procedures.

INCREMENT( $A$ ) // 加 1 算法

```
1   $i = 0$            //  $i$  is the index of  $A$ 
2  while  $i < A.length$  and  $A[i] == 1$ 
3       $A[i] = 0$ 
4       $i = i + 1$ 
5  if  $i < A.length$ 
6       $A[i] = 1$ 
```

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

## (2) Incrementing a binary counter

- Let us analyze a sequence of  $n$  INCREMENT operations on an initially zero counter.  
从 0 开始, 计数到  $n$

COUNTER( $A, n$ )

1  $A = 0$

2 **for**  $j \leftarrow 1$  **to**  $n$

3     INCREMENT( $A$ )

INCREMENT( $A$ )

1  $i = 0$

2 **while**  $i < A.length$  and  $A[i] == 1$

3      $A[i] = 0$

4      $i = i + 1$

5 **if**  $i < A.length$

6      $A[i] = 1$

- Running time of COUNTER?  $O(nk)$ ,  $k$  是  $A$  的位数
- Correct. Not tight.
- Tight running time?  $O(n)$

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

## (2) Incrementing a binary counter

COUNTER( $A, n$ )

```
1  $A = 0$ 
2 for  $j \leftarrow 1$  to  $n$ 
3   INCREMENT( $A$ )
```

INCREMENT( $A$ )

```
1  $i = 0$ 
2 while  $i < A.length$  and  $A[i] == 1$ 
3    $A[i] = 0$ 
4    $i = i + 1$ 
5 if  $i < A.length$ 
6    $A[i] = 1$ 
```

- Not all bits flip each time INCREMENT is called
- $A[0]$  does flip each time ( $n$  times) INCREMENT is called
- $A[1]$ , flips only every other time:  $n/2$  times
- $A[2]$ , flips only every fourth time:  $n/4$  times
- $A[i]$  flips  $n/2^i$  times
- The total number of flips in COUNTER

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

The amortized cost per operation:

$$O(n)/n = O(1)$$

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[1]位, 翻转 $n/2$ 次; A[0]位, 翻转 $n$ 次

# application of aggregate analysis -- convex hull

## GRAHAM-SCAN( $Q$ )

- 1 let  $p_0$  be the point in  $Q$  with the minimum y-coordinate, or the leftmost such point in case of a tie
- 2 let  $\langle p_1, p_2, \dots, p_m \rangle$  be the remaining points in  $Q$ , sorted by polar angle in counterclockwise order around  $p_0$  (if more than one point has the same angle, remove all but the one that is farthest from  $p_0$ )
- 3 PUSH( $p_0, S$ )
- 4 PUSH( $p_1, S$ )
- 5 PUSH( $p_2, S$ )
- 6 **for**  $i \leftarrow 3$  **to**  $n$
- 7     **while** ( the consecutive segments formed by points  
                NEXT-TO-TOP( $S$ ), TOP( $S$ ), and  $p_i$   
                make a nonleft turn)
- 8         POP( $S$ )
- 9         PUSH( $p_i, S$ )
- 10 **return**  $S$

$$T(n) = ?$$

$$\Theta(n)$$

$O(n \lg n)$ , using merge sort and the cross-product method to compare angles.

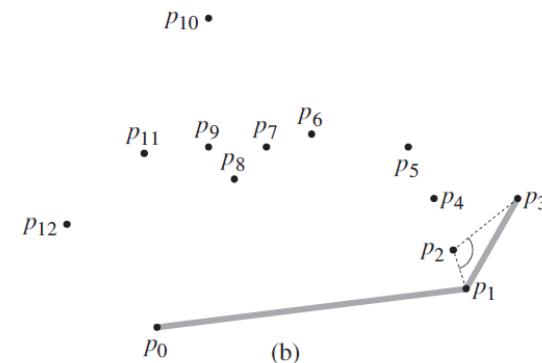
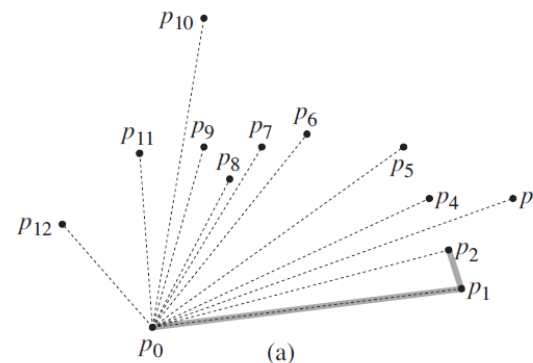
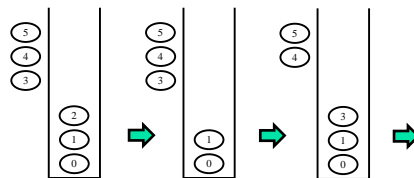
$$O(1)$$

$$O(1)$$

$$O(1)$$

$$O(n-3)$$

**Aggregate analysis:** while loop takes  $O(n)$  time overall. For  $i = 0, 1, \dots, n$ , each point  $p_i$  is pushed onto stack  $S$  exactly once, there is at most one POP operation for each PUSH operation. At least three points  $p_0, p_1$ , and  $p_n$  are never popped from the stack, so that in fact at most  $(n - 2)$  POP operations are performed in total?



## 17.2 The accounting method

- **Amortized cost** : the amount we charge an operation.
  - ◆ **Credit** : when an operation's amortized cost exceeds its actual cost, we assign the difference to specific objects in the data structure as credit.  
分配的消费券大于实际价格, 多余券留下, 作为信用, 将来用 (某些分摊金额小于实际花费时使用)
  - ◆ Credit can help pay for later operations whose amortized cost is less than their actual cost.
- We denote the actual cost of the  $i$ th operation by  $c_i$  and the amortized cost of the  $i$ th operation by  $\hat{c}_i$ , we require

$$\sum_{i=1}^n \hat{c}_i \geq \sum_{i=1}^n c_i$$

For all sequences of  $n$  operations.

- The total credit

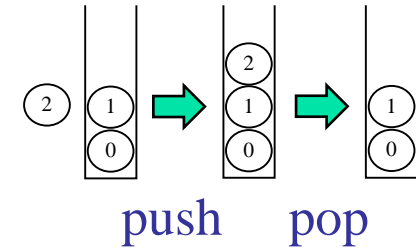
$$\sum_{i=1}^n \hat{c}_i - \sum_{i=1}^n c_i$$



## 17.2 The accounting method – (1) Stack operations

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

$$\sum_{i=1}^n \hat{c}_i \geq \sum_{i=1}^n c_i$$



- The actual costs of the operations

Push	1
Pop	1
Multipop	$\min(k, s)$

- We assign the amortized costs

Push	2
Pop	0
Multipop	0

- We can pay for any sequence of stack operations by charging the amortized costs?

## 17.2 The accounting method – (1) Stack operations

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

$$\sum_{i=1}^n \hat{c}_i \geq \sum_{i=1}^n c_i$$

- The actual costs of the operations

Push	1
Pop	1
Multipop	$\min(k, s)$

- We assign the amortized costs

Push	2
Pop	0
Multipop	0

- We can pay for any sequence of stack operations.
- For any sequence of  $n$  Push, Pop, and Multipop operations, the total amortized cost is  $O(n)$ , so is the total actual cost.

```
STACK( $S, n$ )  
1  $S = \text{NULL}$   
2 for  $i \leftarrow 1$  to  $n$   
3   One of ( PUSH( $S, i$ ), POP( $S$ ), MULIPOP( $S, k$ ) )
```

## 17.2 The accounting method – (2) Incrementing a binary counter

Charge an amortized cost of 2 dollars to set a bit from 0 to 1. When a bit is set, we use 1 dollar (out of the 2 dollars charged) to pay for the actual setting of the bit, and we place the other dollar on the bit as credit to be used later when we flip the bit back to 0. At any point in time, every 1 in the counter has a dollar of credit on it, and thus we can charge nothing to reset a bit to 0.

- ① 0变到1时, 支付分摊消费2元 (其中1元用于实际转化0到1, 结余1元作为信用);
- ② 1变到0时, 分摊消费为0元, 用信用来支付实际消费 (实际翻转)。

- The amortized cost of an INCREMENT is at most 2 dollars.
- For  $n$  INCREMENT operations, the total amortized cost is  $O(n)$ , which bounds the total actual cost.

COUNTER( $A, n$ )

```

1  $A = 0$ 
2 for  $j \leftarrow 1$  to  $n$ 
3   INCREMENT( $A$ )
    
```

INCREMENT( $A$ )

```

1  $i = 0$ 
2 while  $i < A.length$  and  $A[i] == 1$ 
3    $A[i] = 0$ 
4    $i = i + 1$ 
5 if  $i < A.length$ 
6    $A[i] = 1$ 
    
```

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

# application of accounting method -- convex hull

## GRAHAM-SCAN( $Q$ )

- 1 let  $p_0$  be the point in  $Q$  with the minimum y-coordinate, or the leftmost such point in case of a tie
- 2 let  $\langle p_1, p_2, \dots, p_m \rangle$  be the remaining points in  $Q$ , sorted by polar angle in counterclockwise order around  $p_0$  (if more than one point has the same angle, remove all but the one that is farthest from  $p_0$ )
- 3 PUSH( $p_0, S$ )
- 4 PUSH( $p_1, S$ )
- 5 PUSH( $p_2, S$ )
- 6 **for**  $i \leftarrow 3$  **to**  $n$
- 7     **while** ( the consecutive segments formed by points  
                NEXT-TO-TOP( $S$ ), TOP( $S$ ), and  $p_i$   
                make a nonleft turn)
- 8         POP( $S$ )
- 9         PUSH( $p_i, S$ )
- 10 **return**  $S$

$T(n) = ?$

$\Theta(n)$

$O(n \lg n)$ , using merge sort and the cross-product method to compare angles.

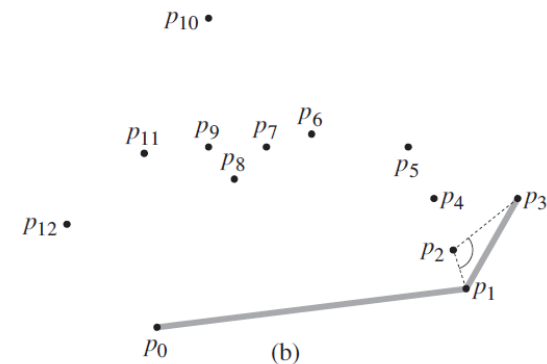
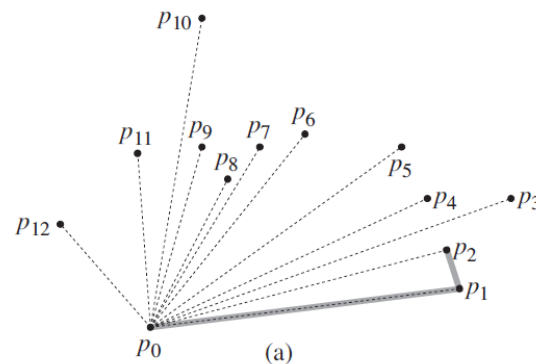
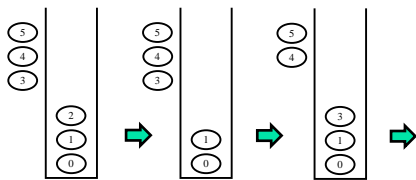
$O(1)$

$O(1)$

$O(1)$

$O(n-3)$

**Accounting method:** 每个顶点  $p_i$  都会且只会被PUSH一次, 给予每个PUSH操作分摊消费 2, 其中 1 个用于实际PUSH, 另 1 个作为“信用值”存储于该顶点上, POP操作的分摊消费为 0, 但POP能执行 (用“信用值”进行支付)。



# application of accounting method -- KMP

accounting:  $q$  (或  $k$ ) 加1时2个分摊消费 (1个用于实际消费, 1个是信用), 减少时0个分摊消费 (数 $q$ 上有 $q$ 个信用, 最多减少到0, 因此信用足够支付减少)

KMP-MATCHER( $T, P$ )

```
1   $n = T.length$ 
2   $m = P.length$ 
3   $\pi = \text{COMPUTE-PREFIX-FUNCTION}(P)$ 
4   $q = 0$ 
5  for  $i = 1$  to  $n$ 
6      while  $q > 0$  and  $P[q + 1] \neq T[i]$ 
7           $q = \pi[q]$  // cost 0
8      if  $P[q + 1] == T[i]$ 
9           $q = q + 1$  // cost 2
10     if  $q == m$ 
11         print "Pattern occurs with shift"  $i - m$ 
12      $q = \pi[q]$ 
```

$\Theta(n)$

COMPUTE-PREFIX-FUNCTION( $P$ )

```
1   $m = P.length$ 
2  let  $\pi[1..m]$  be a new array
3   $\pi[1] = 0$ 
4   $k = 0$ 
5  for  $q = 2$  to  $m$ 
6      while  $k > 0$  and  $P[k + 1] \neq P[q]$ 
7           $k = \pi[k]$  // cost 0
8      if  $P[k + 1] == P[q]$ 
9           $k = k + 1$  // cost 2
10      $\pi[q] = k$ 
11  return  $\pi$ 
```

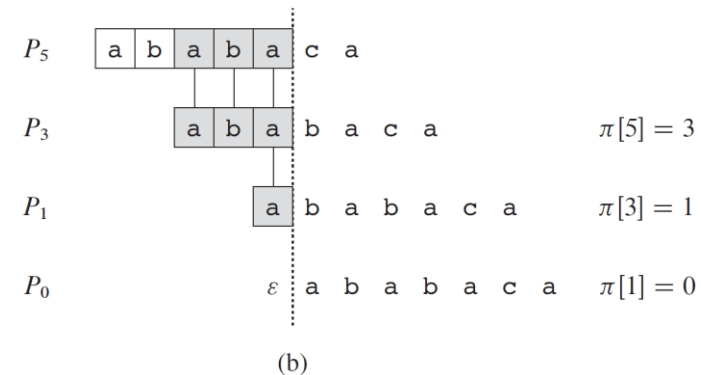
$\Theta(m)$

prefix function:

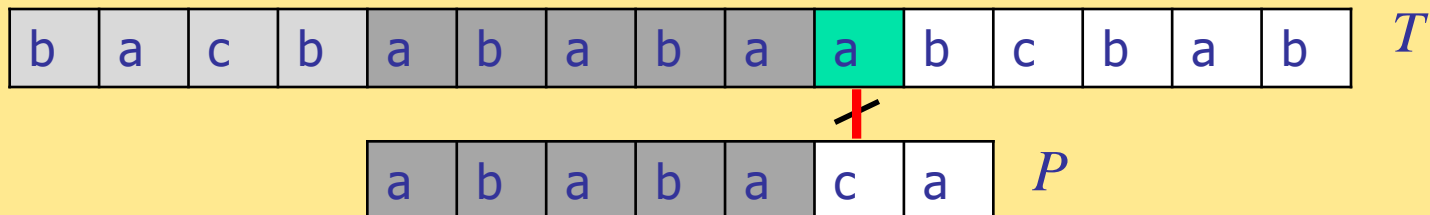
$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupseteq P_q\}$$

$i$	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

(a)



# Review of KMP



$P_5 \sqsupset T_{s+5}$ , but,  $T[s+5+1] \neq P[5+1]$ , ( $q = 5$ )

模版  $P$  的前5个字符与文本匹配, 第6个不匹配,  
模版  $P$  “漂移” 到什么位置 (然后继续匹配) ?

KMP-MATCHER( $T, P$ )

```
1   $n = T.length$ 
2   $m = P.length$ 
3   $\pi = \text{COMPUTE-PREFIX-FUNCTION}(P)$ 
4   $q = 0$ 
5  for  $i = 1$  to  $n$ 
6      while  $q > 0$  and  $P[q + 1] \neq T[i]$ 
7           $q = \pi[q]$ 
8      if  $P[q + 1] == T[i]$ 
9           $q = q + 1$ 
10     if  $q == m$ 
11         print “Pattern occurs with shift”  $i - m$ 
12          $q = \pi[q]$ 
```

# Review of KMP

b a c b a b a b a a b c b a b  $T$

a b a b a c a  $P$

$P_5 \sqsupset T_{s+5}$ , but,  $T[s+5+1] \neq P[5+1]$ , ( $q = 5$ )

模版  $P$  的前5个字符与文本匹配, 第6个不匹配,  
模版  $P$  “漂移” 到什么位置 (然后继续匹配) ?



b a c b a b a b a a b c b a b  $T$

a b a b a c a  $P$

$P$  的前缀  $P_3$  是已匹配的  $P_5$  的最大后缀,

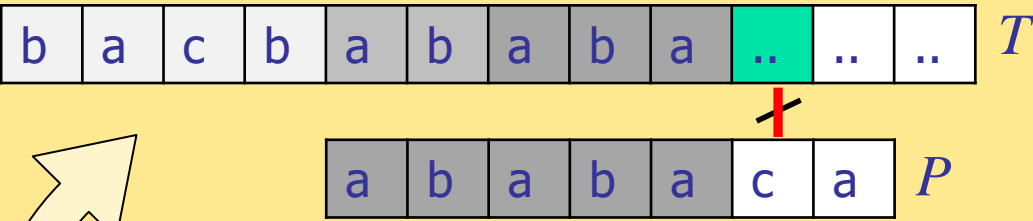
$q = \pi[5] = 3$ , 即  $P_5$  的后缀且是  $P$  的最大前缀  
为  $P_3$ 。即,  $P_3$  是已扫描的子文本  $T_{s+5}$  的 “最  
大” 后缀 (除已经处理过的  $P_5$  外), 从此处  
开始继续判断  $T[i]$  是否与  $P[q+1]$  匹配)。

KMP-MATCHER( $T, P$ )

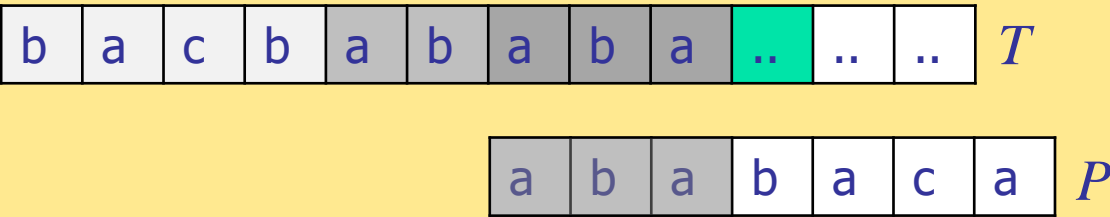
```
1   $n = T.length$ 
2   $m = P.length$ 
3   $\pi = \text{COMPUTE-PREFIX-FUNCTION}(P)$ 
4   $q = 0$ 
5  for  $i = 1$  to  $n$ 
6      while  $q > 0$  and  $P[q + 1] \neq T[i]$ 
7           $q = \pi[q]$ 
8      if  $P[q + 1] == T[i]$ 
9           $q = q + 1$ 
10     if  $q == m$ 
11         print “Pattern occurs with shift”  $i$ 
12          $q = \pi[q]$ 
```

# Review: KMP vs FA

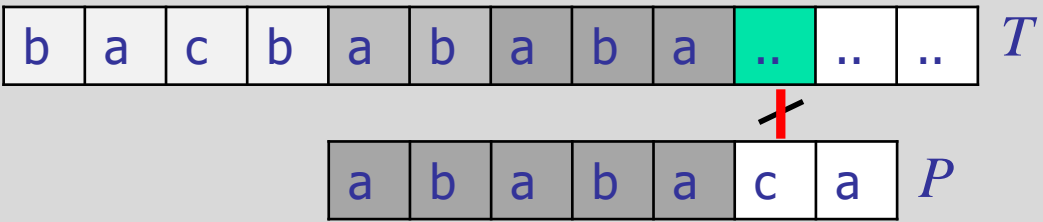
## KMP



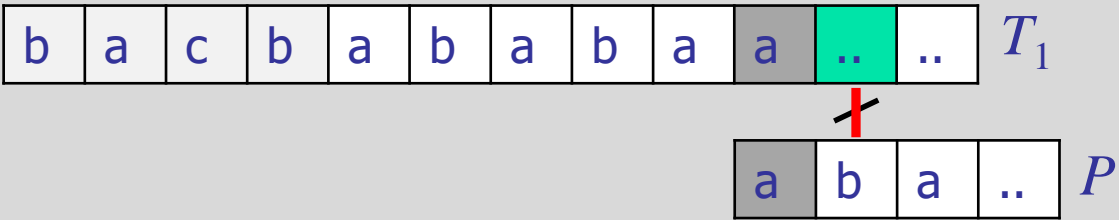
已知文本  $T$  中有  $q$  个字符与模板  $P$  匹配，下一个输入  $T[i]$  如果与  $P[q+1]$  不匹配，不管  $T[i]$  是什么，根据前缀函数  $\pi[q]$ ，能快速移动模板  $P$  到合适的位置。



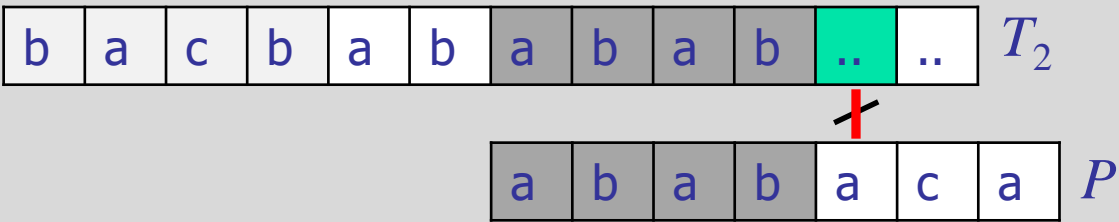
## FA



已知文本  $T$  中有  $q$  个字符与模板  $P$  匹配，下一个输入  $T[i]$  如果与  $P[q+1]$  不匹配，模板  $P$  移动到什么位置跟输入  $T[i]$  相关，根据转移函数  $\delta$  决定。



When  $T[i] = \text{a}$ ,  $\because \delta(5, \text{a}) = 1$ ,  $\therefore q = 1$



When  $T[i] = \text{b}$ ,  $\because \delta(5, \text{b}) = 4$ ,  $\therefore q = 4$



# Review of KMP

如何求  $\pi[q]$ ? KMP-MATCHER, COMPUTE-PREFIX-FUNCTION, 这两个函数的逻辑完全相同, 第一个是找  $T_x$  最大后缀, 第二个找  $P_q$  的最大后缀 (两者都是  $P$  的前缀)。

KMP-MATCHER( $T, P$ )

```

1   $n = T.length$ 
2   $m = P.length$ 
3   $\pi = \text{COMPUTE-PREFIX-FUNCTION}(P)$ 
4   $q = 0$ 
5  for  $i = 1$  to  $n$ 
6    while  $q > 0$  and  $P[q + 1] \neq T[i]$ 
7       $q = \pi[q]$  // cost 0
8    if  $P[q + 1] == T[i]$ 
9       $q = q + 1$  // cost 2
10   if  $q == m$ 
11     print "Pattern occurs with shift"  $i - m$ 
12    $q = \pi[q]$ 
    
```

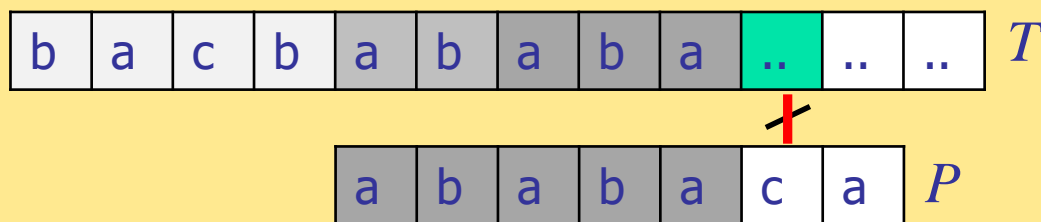
$\Theta(n)$

COMPUTE-PREFIX-FUNCTION( $P$ )

```

1   $m = P.length$ 
2  let  $\pi[1..m]$  be a new array
3   $\pi[1] = 0$ 
4   $k = 0$ 
5  for  $q = 2$  to  $m$ 
6    while  $k > 0$  and  $P[k + 1] \neq P[q]$ 
7       $k = \pi[k]$  // cost 0
8    if  $P[k + 1] == P[q]$ 
9       $k = k + 1$  // cost 2
10    $\pi[q] = k$ 
11  return  $\pi$ 
    
```

$\Theta(m)$

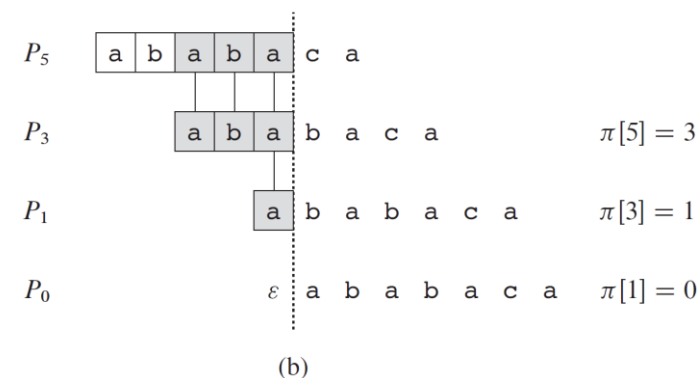


prefix function:

$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupseteq P_q\}$$

$i$	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

(a)



## 17.3 The potential method (势能法)

- The potential method of amortized analysis represents the prepaid work as “potential energy” (potential), which can be released to pay for future operations.
- We **associate the potential with the data structure (DS) as a whole** rather than with specific objects within the data structure.
- The potential method works as follows:
  - ◆ We will perform  $n$  operations, starting with an initial DS  $D_0$ .
  - ◆  $c_i$  : the actual cost of the  $i$ th operation ( $i = 1, 2, \dots, n$ ).
  - ◆  $D_i$  : the DS that results after applying the  $i$ th operation to DS  $D_{i-1}$ .
  - ◆  $\Phi$  : A potential function  $\Phi$  maps  $D_i$  to a real number  $\Phi(D_i)$ , which is the potential associated with  $D_i$ .
- **The amortized cost  $\hat{c}_i$**  of the  $i$ th operation with respect to potential function  $\Phi$  is defined by

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



## 17.3 The potential method (势能法)

- The potential method of amortized analysis represents the prepaid work as “potential energy” (potential), which can be released to pay for future operations.
- We associate the potential with the data structure (DS) as a whole rather than with specific objects within the data structure.
- The potential method works as follows:
  - ◆ We will perform  $n$  operations, starting with an initial DS  $D_0$ .
  - ◆  $c_i$  : the actual cost of the  $i$ th operation ( $i = 1, 2, \dots, n$ ).
  - ◆  $D_i$  : the DS that results after applying the  $i$ th operation to DS  $D_{i-1}$ .
  - ◆  $\Phi$  : A potential function  $\Phi$  maps  $D_i$  to a real number  $\Phi(D_i)$ , which is the potential associated with  $D_i$ .
- **The amortized cost  $\hat{c}_i$**  of the  $i$ th operation with respect to potential function  $\Phi$  is defined by

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

- The total amortized cost of the  $n$  operations is
$$\begin{aligned}\sum_{i=1}^n \hat{c}_i &= \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^n c_i + \Phi(D_n) - \Phi(D_0)\end{aligned}$$

## 17.3 The potential method (势能法)

- The amortized cost  $\hat{c}_i$  of the  $i$ th operation with respect to potential function  $\Phi$  is defined by

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

- The total amortized cost of the  $n$  operations is
$$\begin{aligned}\sum_{i=1}^n \hat{c}_i &= \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^n c_i + \Phi(D_n) - \Phi(D_0)\end{aligned}$$

- We usually just define  $\Phi(D_0)$  to be 0 and then show that  $\Phi(D_i) \geq 0$  for all  $i$ .
- Different potential functions may yield different amortized costs.

## 17.3 The potential method – (1) Stack operations

- The amortized cost  $\hat{c}_i$  of the  $i$ th operation with respect to potential function  $\Phi$  is defined by

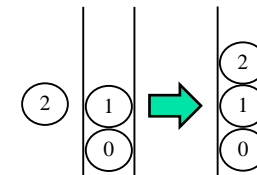
$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

- The total amortized cost of the  $n$  operations is 
$$\begin{aligned} \sum_{i=1}^n \hat{c}_i &= \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^n c_i + \Phi(D_n) - \Phi(D_0) \end{aligned}$$

```
STACK(S, n)
1 S = NULL
2 for i ← 1 to n
3   One of (PUSH(S, i), POP(S), MULIPOP(S, k))
```

- We **define the potential function  $\Phi$  on a stack to be the number of objects** in the stack.
  - for the empty stack  $D_0$  with which we start, we have  $\Phi(D_0) = 0$
  - after the  $i$ th operation, for the stack  $D_i$ ,  **$\Phi(D_i) \geq 0$  ?**
- If the  $i$ th operation on a stack containing  $s$  objects is a PUSH operation, the potential difference is  $\Phi(D_i) - \Phi(D_{i-1}) = (s+1) - s = 1$ .
- The amortized cost of this **PUSH** operation is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$$



## 17.3 The potential method – (1) Stack operations



- If the  $i$ th operation is **MULTIPOP**( $S, k$ ), which causes  $k' = \min(k, s)$  objects to be popped off the stack. The actual cost of the operation is  $k'$ , and the potential difference is  $\Phi(D_i) - \Phi(D_{i-1}) = -k'$
- The amortized cost of this **MULTIPOP** operation is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k' - k' = 0$$

- Similarly, the amortized cost of an ordinary **POP** operation is 0.

- The worst-case cost of **STACK** is therefore  $O(n)$ ?

```
STACK(S, n)
1  S = NULL
2  for i ← 1 to n
3    One of ( PUSH(S, i), POP(S), MULIPOP(S, k) ) // 2 at most
```

## 17.3 The potential method – (2) Incrementing a binary counter

- We define the potential  $\Phi$  of the counter after the  $i$ th INCREMENT operation to be  $b_i$ , the number of 1s in the counter. 第  $i$  次 “INCREMENT 加1” 操作后, 势函数  $b_i = \Phi(D_i)$  为计数器中 1 的个数
- Suppose that the  $i$ th INCREMENT operation resets  $t_i$  bits (from 1 to 0). The actual cost of the operation is therefore at most, that is  $c_i \leq t_i + 1$ , since in addition to resetting  $t_i$  bits, it sets at most one bit to 1.  
实际的消费  $c_i \leq t_i + 1$ , 即, 把  $t_i$  个 1 变为 0, 至多把 1 个 0 变为 1
  - If  $b_i = 0$ , then the  $i$ th operation resets all  $k$  bits. 计数器中 1 的个数为 0, 即第  $i$  次操作重置所有  $k$  位。原  $k$  位都是 1, 已经是最大数, 加 1 操作后越界限, 变成 1000..00, 最高位 1 舍去, 因此  $b_{i-1} = t_i = k$ . (第  $i$  次操作前有  $k$  个 1)
  - If  $b_i > 0$ , then  $b_i = b_{i-1} - t_i + 1$ . 有效计数范围内,  $t_i$  个 1 变为 0 (while 循环, 即第 2~4 行), 1 个 0 变为 1 (加 1 算法的最后一行, 即第 6 行)
  - In either case,  $b_i \leq b_{i-1} - t_i + 1$ , and the potential difference is

$$\Phi(D_i) - \Phi(D_{i-1}) = b_i - b_{i-1} \leq (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i$$

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq (t_i + 1) + (1 - t_i) = 2$$

- The amortized cost of INCREMENT is therefore 2.

```
COUNTER(A, n)
1 A = 0
2 for j ← 1 to n
3   INCREMENT(A)
```

```
INCREMENT(A)
1 i = 0
2 while i < A.length and A[i] == 1
3   A[i] = 0
4   i = i + 1
5 if i < A.length
6   A[i] = 1
```

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

## 17.3 The potential method – (2) Incrementing a binary counter

- We define the potential  $\Phi$  of the counter after the  $i$ th INCREMENT operation to be  $b_i$ , the number of 1s in the counter. 第  $i$  次操作后, 势函数  $b_i$  为计数器中 1 的个数
- The amortized cost is therefore

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq (t_i + 1) + (1 - t_i) = 2$$

COUNTER( $A, n$ )

1  $A = 0$

2 **for**  $j \leftarrow 1$  **to**  $n$

3     INCREMENT( $A$ )

INCREMENT( $A$ )

1  $i = 0$

2 **while**  $i < A.length$  and  $A[i] == 1$

3      $A[i] = 0$

4      $i = i + 1$

5 **if**  $i < A.length$

6      $A[i] = 1$

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

- The worst-case cost of COUNTER is therefore  $O(n)$ ?



## 17.4 Dynamic tables



```
void * malloc(size_t size);
```

```
c : malloc/free
```

```
c++: new/delete
```

- We might allocate space for a table, only to find out later that it is not enough. We must then reallocate the table with a larger size and copy all objects stored in the original table over into the new, larger table.



- Similarly, if many objects have been deleted from the table, it may be worthwhile to reallocate the table with a smaller size.



- We assume that the dynamic table supports the operations TABLE-INSERT and TABLE-DELETE.
  - ◆ TABLE-INSERT inserts into the table an item that occupies a single slot, that is, a space for one item.
  - ◆ Likewise, TABLE-DELETE removes an item from the table, thereby freeing a slot.

## 17.4 Dynamic tables

- The details of the data-structuring method used to organize the table are unimportant. It might be:

a stack, or a heap, or a hash table, or an array

数据结构用什么没关系，可以是栈、堆、hash表、数组、...

- load factor  $\alpha(T)$  of a nonempty table  $T$ :

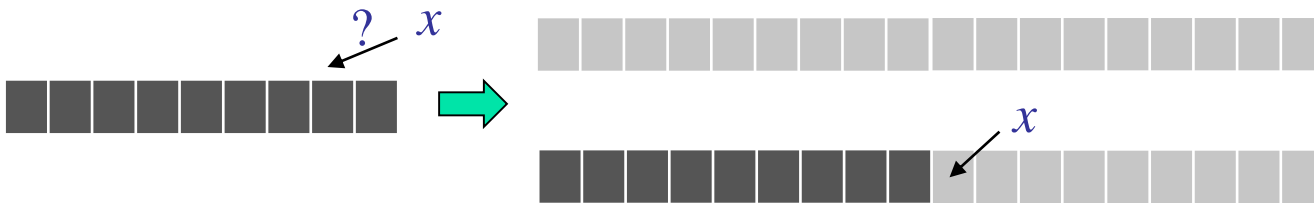
$$\alpha(T) = T.num/T.size$$



- We assign an empty table (one with no items) size 0, and we define its load factor to be 1.

## 17.4.1 Table expansion

- We assume that storage for a table is allocated as an **array** of slots.
- Upon inserting an item into a full table, we can **expand** the table by allocating a new table with more slots than the old table had. 往一个满表插入新元素时，需要先进行表扩展
- A common heuristic allocates a new table with **twice** as many slots as the old one.
- If the only table operations are **insertions**, then the load factor of the table is always at least  $1/2$ , and thus the amount of wasted space never exceeds half the total space in the table.



往一个满表插入新元素时，需要先进行表扩展（一般是容量倍增），然后把原来的元素拷贝到扩展后的表，然后再插入新元素。

TABLE-INSERT( $T, x$ )

```
1  if  $T.size == 0$ 
2      allocate  $T.table$  with 1 slot
3       $T.size = 1$ 
4  if  $T.num == T.size$ 
5      allocate  $new-table$  with  $2 \cdot T.size$  slots
6      insert all items in  $T.table$  into  $new-table$ 
7      free  $T.table$ 
8       $T.table = new-table$ 
9       $T.size = 2 \cdot T.size$ 
10 insert  $x$  into  $T.table$ 
11  $T.num = T.num + 1$ 
```

<  $T.table$  : a pointer to  $T$  >

## 17.4.1 Table expansion

从 0 开始, 插入  $n$  个元素, The running time of TABLE-EXPANSION?

```
TABLE-EXPANSION( $T, n$ )
```

```
1  $T = \text{NULL}$   
2 for  $i \leftarrow 1$  to  $n$   
3   TABLE-INSERT( $T, x$ )
```

```
TABLE-INSERT( $T, x$ )
```

```
1 if  $T.\text{size} == 0$   
2   allocate  $T.\text{table}$  with 1 slot  
3    $T.\text{size} = 1$   
4 if  $T.\text{num} == T.\text{size}$   
5   allocate  $\text{new-table}$  with  $2 \cdot T.\text{size}$  slots  
6   insert all items in  $T.\text{table}$  into  $\text{new-table}$   
7   free  $T.\text{table}$   
8    $T.\text{table} = \text{new-table}$   
9    $T.\text{size} = 2 \cdot T.\text{size}$   
10  insert  $x$  into  $T.\text{table}$   
11   $T.\text{num} = T.\text{num} + 1$ 
```



- What is the cost  $c_i$  of the  $i$ th operation?
  - ◆ If the current table has room for the new item (or the first operation), then  $c_i = 1$  ?
  - ◆ If the current table is full, an expansion occurs, then  $c_i = i$  ?
- The worst-case cost of an operation is  $O(n)$  ?
- The upper bound of TABLE-EXPANSION is  $O(n^2)$  ?
- Correct. Not tight ?

## 17.4.1 Table expansion – (1) aggregate analysis

从 0 开始，插入  $n$  个元素，The running time of TABLE-EXPANSION?

TABLE-EXPANSION( $T, n$ )

```
1  $T = \text{NULL}$ 
2 for  $i \leftarrow 1$  to  $n$ 
3   TABLE-INSERT( $T, x$ )
```

TABLE-INSERT( $T, x$ )

```
1 if  $T.\text{size} == 0$ 
2   allocate  $T.\text{table}$  with 1 slot
3    $T.\text{size} = 1$ 
4 if  $T.\text{num} == T.\text{size}$ 
5   allocate  $\text{new-table}$  with  $2 \cdot T.\text{size}$  slots
6   insert all items in  $T.\text{table}$  into  $\text{new-table}$ 
7   free  $T.\text{table}$ 
8    $T.\text{table} = \text{new-table}$ 
9    $T.\text{size} = 2 \cdot T.\text{size}$ 
10 insert  $x$  into  $T.\text{table}$ 
11  $T.\text{num} = T.\text{num} + 1$ 
```

What is the cost  $c_i$  of the  $i$ th operation?

$$c_i = \begin{cases} i & , \text{ if } i-1=2^k, \\ 1 & , \text{ otherwise.} \end{cases} \quad \Rightarrow \quad \sum_{i=1}^n c_i \leq n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j < n + 2n = 3n$$

No.	cost	$T$							
1	1	Red							
2	2	Red	Red						
3	3	Red	Red	Red					
4	1	Black	Black	Black	Red				
5	5	Red	Red	Red	Red	Red	Gray	Gray	Gray
6	1	Black	Black	Black	Black	Black	Red	Gray	Gray

Red: 执行插入元素操作 (次数)

Gray: 空槽 null slot

Black: 有元素的槽

# 17.4.1 Table expansion – (1) aggregate analysis

The running time of TABLE-EXPANSION?

TABLE-EXPANSION( $T, n$ )







```

1  $T = \text{NULL}$ 
2 for  $i \leftarrow 1$  to  $n$ 
3   TABLE-INSERT( $T, x$ )
    
```

TABLE-INSERT( $T, x$ )

```

1 if  $T.\text{size} == 0$ 
2   allocate  $T.\text{table}$  with 1 slot
3    $T.\text{size} = 1$ 
4 if  $T.\text{num} == T.\text{size}$ 
5   allocate  $\text{new-table}$  with  $2 \cdot T.\text{size}$  slots
6   insert all items in  $T.\text{table}$  into  $\text{new-table}$ 
7   free  $T.\text{table}$ 
8    $T.\text{table} = \text{new-table}$ 
9    $T.\text{size} = 2 \cdot T.\text{size}$ 
10 insert  $x$  into  $T.\text{table}$ 
11  $T.\text{num} = T.\text{num} + 1$ 
    
```

No.	cost	$T$
1	1	
2	2	
3	3	
4	1	
5	5	
6	1	

What is the cost  $c_i$  of the  $i$ th operation?



插入每个元素所需的开销?

1	2	3	1	5	1	1	1	9	...
---	---	---	---	---	---	---	---	---	-----

1	1	1	1	1	1	1	1	1	...
---	---	---	---	---	---	---	---	---	-----

插入元素的开销

	1	2		4				8	...
--	---	---	--	---	--	--	--	---	-----

扩展表且拷贝元素的开销

$$c_i = \begin{cases} i, & \text{if } i-1=2^k, \\ 1, & \text{otherwise.} \end{cases} \quad \Rightarrow \quad \sum_{i=1}^n c_i \leq n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j < n + 2n = 3n$$

# 17.4.1 Table expansion – (2) accounting method

从 0 开始，插入  $n$  个元素，The running time of TABLE-EXPANSION?

```
TABLE-EXPANSION( $T, n$ )
1  $T = \text{NULL}$ 
2 for  $i \leftarrow 1$  to  $n$ 
3   TABLE-INSERT( $T, x$ )
```

```
TABLE-INSERT( $T, x$ )
1 if  $T.\text{size} == 0$ 
2   allocate  $T.\text{table}$  with 1 slot
3    $T.\text{size} = 1$ 
4 if  $T.\text{num} == T.\text{size}$ 
5   allocate  $\text{new-table}$  with  $2 \cdot T.\text{size}$  slots
6   insert all items in  $T.\text{table}$  into  $\text{new-table}$ 
7   free  $T.\text{table}$ 
8    $T.\text{table} = \text{new-table}$ 
9    $T.\text{size} = 2 \cdot T.\text{size}$ 
10 insert  $x$  into  $T.\text{table}$ 
11  $T.\text{num} = T.\text{num} + 1$ 
```

What is the cost  $c_i$  of the  $i$ th operation?

$$c_i = \begin{cases} i & , \text{ if } i-1=2^k, \\ 1 & , \text{ otherwise.} \end{cases} \quad \Rightarrow \quad \sum_{i=1}^n c_i \leq n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j < n + 2n = 3n$$

We charge an amortized cost of 3 dollars for each insertion, 1 for actual insertion, 2 for credits.

No.	cost	$T$							
1	1	2	total is 3: 1 for cost, 2 left for credits.						
2	2	1	2						
3	3	0	1	2					
4	1	0	1	2	2				
5	5	-1	0	1	1	2			
6	1	-1	0	1	1	2	2		

### 17.4.1 Table expansion – (3) potential method

- potential function :  $\Phi(T) = 2T.num - T.size$ 
  - Immediately after an expansion, we have  $T.num = T.size/2$ , thus  $\Phi(T) = 0$ .
  - Immediately before an expansion, we have  $T.num = T.size$ , thus  $\Phi(T) = T.num$ .
- If the  $i$ th TABLE-INSERT operation **does not trigger** an expansion, then we have  $size_i = size_{i-1}$  and the amortized cost of the operation is



$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\ &= 1 + (2 \cdot num_i - size_i) - (2(num_i - 1) - size_i) \\ &= 3\end{aligned}$$

```

TABLE-EXPANSION( $T, n$ )
1  $T = \text{NULL}$ 
2 for  $i \leftarrow 1$  to  $n$ 
3   TABLE-INSERT( $T, x$ )

```

No.	cost	$T$							
1	1	red							
2	2	red	red						
3	3	red	red	red	gray				
4	1	black	black	black	red				
5	5	red	red	red	red	red	gray	gray	
6	1	black	black	black	black	black	red	gray	gray



## 17.4.1 Table expansion – (3) potential method

- potential function :  $\Phi(T) = 2T.num - T.size$ 
  - Immediately after an expansion, we have  $T.num = T.size/2$ , thus  $\Phi(T) = 0$ .
  - Immediately before an expansion, we have  $T.num = T.size$ , thus  $\Phi(T) = T.num$ .
- If the  $i$ th TABLE-INSERT operation **trigger** an expansion,  $size_i = 2size_{i-1}$ , and  $size_{i-1} = num_{i-1} = num_i - 1$ , thus



$$\begin{aligned}
 \hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
 &= num_i + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\
 &= num_i + (2 \cdot num_i - 2 \cdot (num_i - 1)) - (2(num_i - 1) - (num_i - 1)) \\
 &= num_i + 2 - (num_i - 1) \\
 &= 3
 \end{aligned}$$

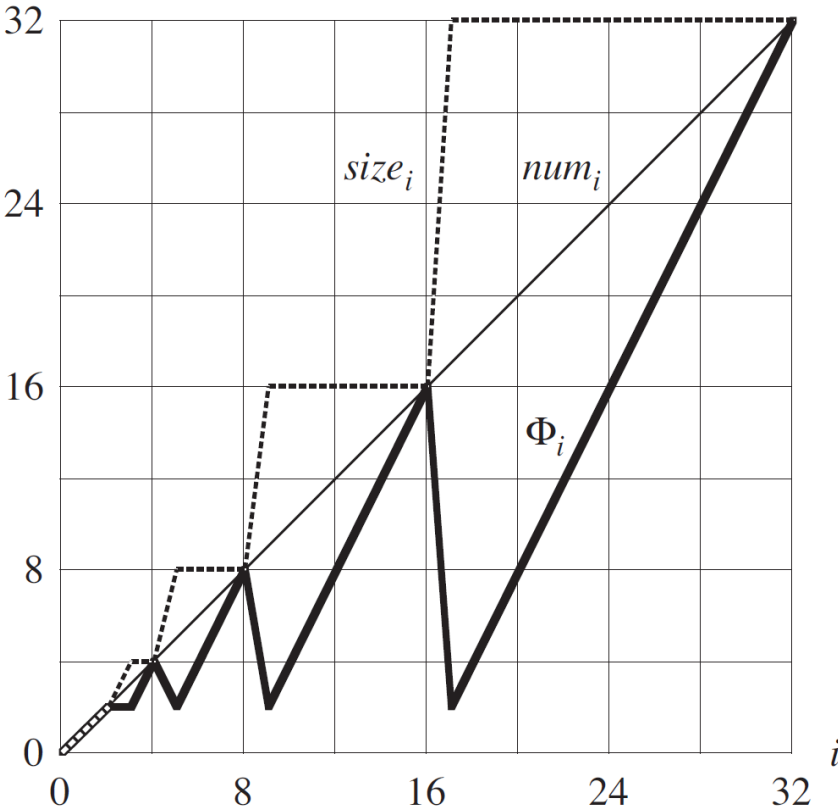
```

TABLE-EXPANSION( $T, n$ )
1  $T = \text{NULL}$ 
2 for  $i \leftarrow 1$  to  $n$ 
3   TABLE-INSERT( $T, x$ )
    
```

No.	cost	$T$
1	1	
2	2	
3	3	
4	1	
5	5	
6	1	

# 17.4.1 Table expansion – (3) potential method

- potential function :  $\Phi(T) = 2T.num - T.size$ 
  - ◆ Immediately after an expansion, we have  $T.num = T.size/2$ , thus  $\Phi(T) = 0$ .
  - ◆ Immediately before an expansion, we have  $T.num = T.size$ , thus  $\Phi(T) = T.num$ .
- For the  $i$ th TABLE-INSERT operation, the amortized cost is 3



```
TABLE-EXPANSION( $T, n$ )
1  $T = \text{NULL}$ 
2 for  $i \leftarrow 1$  to  $n$ 
3   TABLE-INSERT( $T, x$ )
```

No.	cost	$T$
1	1	<div><div></div><div></div><div></div><div></div><div></div><div></div></div>
2	2	<div><div></div><div></div><div></div><div></div><div></div><div></div></div>
3	3	<div><div></div><div></div><div></div><div></div><div></div><div></div></div>
4	1	<div><div></div><div></div><div></div><div></div><div></div><div></div></div>
5	5	<div><div></div><div></div><div></div><div></div><div></div><div></div></div>
6	1	<div><div></div><div></div><div></div><div></div><div></div><div></div></div>

## 17.4.2 Table contraction (表缩小)

---

自学

- 所有的课后习题。
- 再举几个例子（找凸包、KMP，我已经用过了，不能再用），说明如何用Amortized Analysis进行算法的复杂度分析。
- 把本书中所有用Amortized Analysis进行复杂度分析的算法找出来，并指出其分析方法与结果。

# exercises

把本书（也可以不限于本书）中所有用Amortized Analysis进行复杂度分析是算法找出来，并指出其分析方法与结果（可作为期末讲解的大作业）。

