Part IV

Advanced Design and Analysis Techniques
高级、先进的设计与分析方法、技术

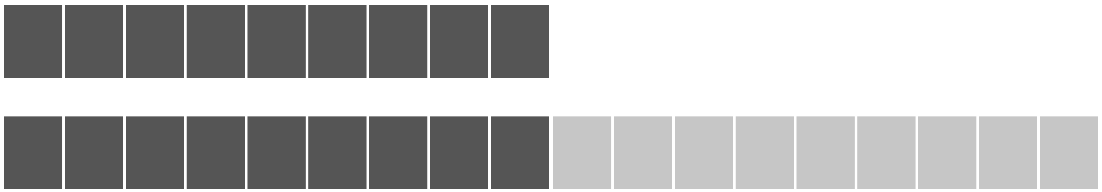
17 Amortized Analysis
平摊（摊还、分摊、分期）分析

一个例子:动态表

void * malloc(size_t size);

c : malloc/free

c++: new/delete



- 我们可能会为一个表分配空间，但后来发现它是不够的。 然后，我们必须重新分配更大的表，并将存储在原始表中的所有对象复制到新的更大的表中。
- 类似地，如果从表中删除了许多对象，那么重新分配一个较小的表可能是值得的。
- 我们假设动态表支持插入和删除操作。
  - 插入操作将一个项目插入到表中，该项目占用一个槽位，即一个项目的空间。
  - 同样的, 删除操作从表中删除一个项，从而释放一个槽。

平摊分析

- 在平摊分析中，我们将执行一系列数据结构操作所需的时间平均给所有执行的操作。
  （在多个操作中，求一个操作的平均时间）
- 通过平摊分析，我们可以表明如果对一个操作序列求平均值，即使序列中的单个操作可能开销较大，每个操作的平均成本并不高。
- 平摊分析与平均情况分析的不同之处在于不涉及概率; 平摊分析保证了在最坏情况下每个操作的平均性能。
- **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、聚合分析）

17.2 The accounting method（记账法）

17.3 The potential method （势能法）

17.1 聚合分析

- 在**聚合分析**中，我们证明了对于所有n，一个n个操作的序列最坏情况总共需要T(n)时间。
- 在最坏的情况下，每个操作的平均代价或平摊代价是T(n)/n。

(1) 栈操作

- **PUSH(*S*, *x*):将对象x存入栈S。** **O(1)**

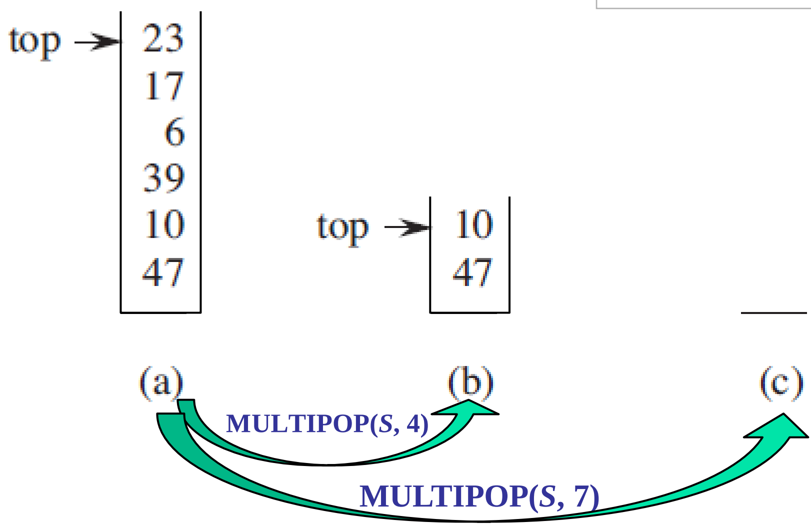
- **POP(*S*): 弹出栈S的顶部并返回弹出的对象。在空堆栈上调用POP会产生错误。** **O(1)**

- **MULTIPOP(*S*, *k*): 移除堆栈S的k个顶部对象，如果堆栈包含的对象少于k个，则弹出整个堆栈。(堆栈有s个对象。)** **O(min(*s*, *k*))?**

(1) 栈操作

- **MULTIPOP(*S*, *k*): 移除堆栈S的k个顶部对象，如果堆栈包含的对象少于k个，则弹出整个堆栈。(堆栈有s个对象。)**

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



O(min(*s*, *k*))

(1) 栈操作

- **让我们分析一个初始空栈上的n个PUSH、POP和MULTIPOP操作序列。**

```
STACK(S, n)
    S = NULL
    for i←1  to  n
        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)

- 正确。不紧。
- 紧的运行时间? **O(n)**
- 每次将每个对象推入堆栈时，我们最多只能从堆栈中弹出一次。 因此，可以在非空堆栈上调用**POP**的次数，包括在**MULTIPOP**内的调用，最多是**PUSH**操作的次数，最多是**n**。

（**push**一个对象后，至多能被**pop**一次。但至多n次**push**。）

O(min(s, k))

worst-case: O(n)

(1) 栈操作

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

- 紧的运行时间。 **$O(n)$**
- 每个操作的平均开销 **$O(n)/n = O(1)$**
- 在聚合分析中，我们将每个操作的平摊成本指定为平均成本。
- 对于栈操作，堆栈操作的平均成本(运行时间)是O(1)，我们没有使用概率推理。
- 用这个总成本除以n就得到了每次操作的平均成本，或平摊成本。

(2) 对二进制计数器递增

考虑实现一个从零开始的k位二进制计数器

(2) 对二进制计数器递增

- 我们使用位数组A[0...k-1]来作为计数器。
- 储存在计数器中的数字 x 最低位在 A[0]中，最高位在 A[k-1]中，所以

$$x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$$

- 最初，$x = 0$，因此对于 $i = 0,\ldots,k-1, A[i] = 0$。
- 要对计数器中的值加1(取2^k的模)，我们使用以下过程

```
INCREMENT(A)            // 加 1 算法
  i = 0                 // index of A
  while i < A.length and A[i] == 1
      A[i] = 0
      i = i + 1
  if i < A.length
      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) 对二进制计数器递增

- 让我们分析初始为0的计数器上的n个INCREMENT操作序列。（从0开始，计数到n）

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

```
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
```

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

- **COUNTER的运行时间?**

  O(*nk*), *k* 是 *A* 的位数

- 正确，但不紧。
- 紧的运行时间? **O(*n*)**

(2) 对二进制计数器递增

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

```
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
```

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

- 并不是每次调用**INCREMENT**时所有的位都翻转。
- ***A*[0]** 每次调用**INCREMENT**时都翻转。
- ***A*[1],每隔一次翻转:n/2次**
- ***A*[2], 每四次翻转一次:n/4次**
- ***A*[*i*] 翻转 $n/2^i$ 次**
- **COUNTER中的翻转总数**

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

The amortized cost per operation:

O(*n*)/*n* = O(1)

聚合分析的应用——凸包

GRAHAM-SCAN(Q)

let p_0 be the point in Q with the minimum y-coordinate, or the leftmost such point in case of a tie

let $\langle p_1, p_2, \ldots, p_m\rangle$ 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)

PUSH(p_0, S)

PUSH(p_1, S)

PUSH(p_2, S)

for $i \leftarrow 3$ **to** m

    while (the consecutive segments formed by points NEXT-TO-TOP(S), TOP(S), and p_i make a nonleft turn**)**

        POP(S)

    PUSH(p_i, S)

return S

T(n) = ?

$\Theta(n)$

$O(n\lg n)$, 用归并排序和叉积法比较角度。

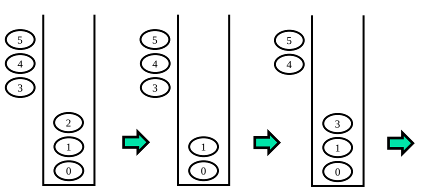
$O(1)$

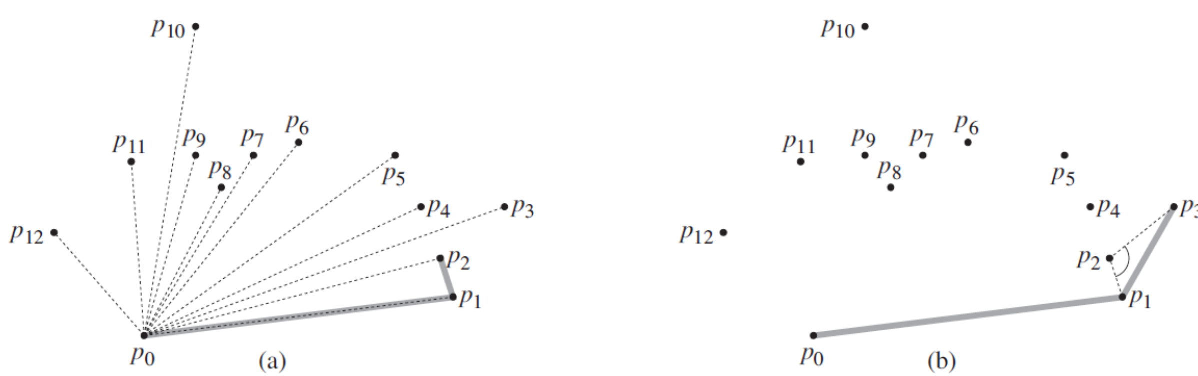
$O(1)$

$O(1)$

$O(n\text{-}3)$

Aggregate analysis: **while**循环总共花费 $O(n)$ 时间。 对于 $i = 0, 1, \ldots, m$，每个点 p_i 只被推入栈S一次，每个PUSH操作最多有一个POP操作。 至少有三个点 p_0, p_1, and p_m从未从堆栈中弹出，因此实际上总共执行最多$(m - 2)$个POP操作。





17.2 记账法

- **平摊成本**:我们在一次操作中收取的费用。
  - **信用**:当一个操作的平摊成本超过其实际成本时，我们将差额分配给数据结构中的特定对象作为信用。
  - 信用可以帮助支付后期的操作，使其平摊成本小于实际成本。
- 我们用 c_i 表示第 i 个操作的实际开销，用 $\hat{c}_i$ 表示第i个操作的平摊开销。我们要求

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

对于n个操作的所有序列。

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

17.2 记账法 - 栈操作

- 第 i 个操作的实际开销： c_i
- 第 i 个操作的平摊开销： $\hat{c}_i$

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

- 操作的实际开销

PUSH	1 ,
POP	1 ,
MULTIPOP	$\min(k, s)$,

- 我们分配以下的平摊开销

PUSH	2 ,
POP	0 ,
MULTIPOP	0 .

- 我们可以通过收取平摊代价来支付栈操作的任何序列?

17.2 记账法 - 栈操作

- 第 i 个操作的实际开销：c_i
- 第 i 个操作的平摊开销：$\widehat{c}_i$

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

- 操作的实际开销

PUSH	1 ,
POP	1 ,
MULTIPOP	$\min(k, s)$,

- 我们分配以下的平摊开销

PUSH	2 ,
POP	0 ,
MULTIPOP	0 .

- 我们可以通过收取平摊代价来支付栈操作的任何序列?
- 对于任何n个PUSH,POP和MULTIPOP的操作序列，总的平摊开销是O(n)，总实际开销也是。

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

17.2 记账法 - 对二进制计数器递增

将位从0设置为1，平摊代价为2美元。当一个比特被设置好后，我们用1美元(从2美元中扣除)来支付比特的实际设置，我们把另外1美元作为信用放在比特上，以便稍后将比特翻转回0时使用。 在任何时间点，计数器上的每一个1都有1美元的信用，因此我们可以不收取任何费用将位重置为0。

(1) 0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到0时，分摊消费为0元，用信用来支付实际消费。

- 一个INCREMENT的平摊代价最多是2美元。
- 对于n个INCREMENT操作，
  总平摊代价是O(n)，
  这限制了总实际成本。

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

COUNTER(A, n)
  $A = 0$
  for $j \leftarrow 1$ **to** n
    INCREMENT(A)

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
```

application of accounting method -- convex hull

```
GRAHAM-SCAN(Q)
let p0 be the point in Q with the minimum y-coordinate,
    or the leftmost such point in case of a tie
let <p1, p2, ..., pm> be the remaining points in Q, sorted by
    polar angle in counterclockwise order around p0
    (if more than one point has the same angle, remove all
    but the one that is farthest from p0 )
PUSH(p0, S)
PUSH(p1, S)
PUSH(p2, S)
for i ← 3 to m
    while ( the consecutive segments formed by points
            NEXT-TO-TOP(S), TOP(S), and pi
            make a nonleft turn)
        POP(S)
    PUSH(pi , S)
return S
```

T(n) = ?

$\Theta(n)$

$O(n\lg n)$，用归并排序和叉积法比较角度。

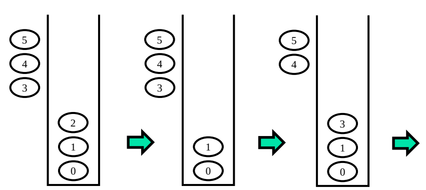
$O(1)$

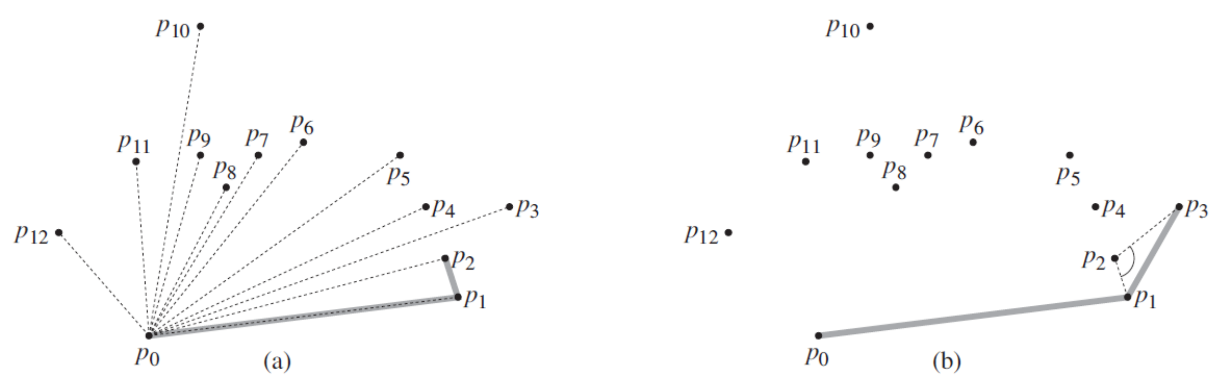
$O(1)$

$O(1)$

$O(n\text{-}3)$

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）。





(a) (b)

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)
n = T.length
m = P.length
π = COMPUTE-PREFIX-FUNCTION(P)
q = 0
for i = 1 to n
    while q > 0 and P[q + 1] ≠ T[i]
        q = π[q]
    if P[q + 1] == T[i]
        q = q + 1
    if q == m
        print "Pattern occurs with shift"
        q = π[q]
```

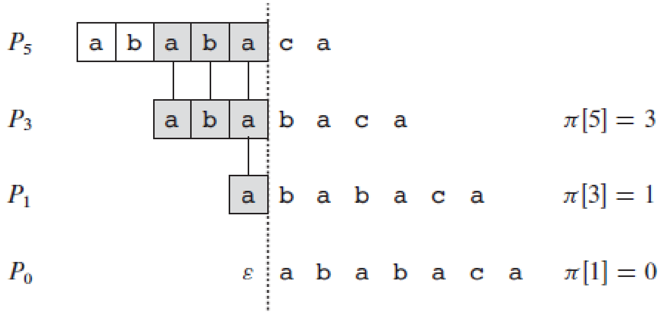
```
COMPUTE-PREFIX-FUNCTION(P)
m = P.length
let π[1..m] be a new array
π[1] = 0
k = 0
for q = 2 to m
    while k > 0 and P[k + 1] ≠ P[q]
        k = π[k]
    if P[k + 1] == P[q]
        k = k + 1
    π[q] = k
return π
```

prefix function:

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

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)



(b)

17.3 势能法

- 平摊分析的势能法将预付的功表示为“势能”(势)，可以释放出来支付未来的操作。
- 我们将势与数据结构(DS)作为一个整体相关联，而不是与数据结构中的特定对象相关联。
- 势能法的工作原理如下:
  - 我们将执行 n 个操作, 从初始 DS D_0 开始。
  - c_i : 第 i 个操作的实际开销 ($i = 1, 2, \ldots, n$)。
  - D_i :对DS D_{i-1}执行第 i 个操作后得到的DS 。
  - Φ : 势函数 Φ 将每个 DS D_i 映射到实数 $\Phi(D_i)$, 这是与 DS D_i 相关的势。
- 关于势函数 Φ 的第 i 个操作的平摊开销 $\hat{c}_i$ 是

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

- n个操作的总平摊开销是

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

17.3 势能法

- 关于势函数 Φ 的第 i 个操作的平摊开销 $\hat{c}_i$ 是

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

- n个操作的总平摊开销是
$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)\ .\end{aligned}$$

- 我们通常只定义 $\Phi(D_0)$ 为 0 然后证明对于所有 i，$\Phi(D_i) \geq 0$。

- 不同的势函数可能产生不同的平摊开销。

17.3 势能法 - 栈操作

- 关于势函数 Φ 的第 i 个操作的平摊开销 $\hat{c}_i$ 是

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

- n个操作的总平摊开销是

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

- 我们将堆栈上的势函数Φ定义为堆栈中对象的数量。
  - 对于初始的空栈 D_0，我们有 $\Phi(D_0) = 0$
  - 第 i 次操作后,对于栈 D_i，$\Phi(D_i) \geq 0$?
- 如果包含s个对象的堆栈上的第i个操作是PUSH操作，则势能差为

$$\Phi(D_i) - \Phi(D_{i-1}) = (s+1) - s = 1.$$

- 这个**PUSH**操作的平摊代价是

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

17.3 势能法 - 栈操作

- 关于势函数 Φ 的第 i 个操作的平摊开销 $\hat{c}_i$ 是

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

- n个操作的总平摊开销是

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)\,.\end{aligned}$$

- 若第i个操作是 **MULTIPOP(*S*, *k*)**,这让 $k' = \min(k, s)$ 个对象从栈中弹出。操作的实际开销是 k', 势能差 $\Phi(D_i) - \Phi(D_{i-1}) = -k'$
- **MULTIPOP**操作的平摊代价是

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

- 类似地，普通**POP**操作的平摊代价为0。

- 因此STACK的最坏情况开销是O(n)？

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

17.3 势能法 - 对二进制计数器递增

- 我们定义第 i 个INCREMENT操作后计数器的势 Φ 为 b_i，即计数器中1的个数。(第 i 次操作后，势函数 b_i = phi(D_i)为计数器中 1 的个数)
- 假设第 i 个INCREMENT操作重置 t_i 位(**从 1到 0**)。因此该操作的实际开销最多为 $t_i + 1$，因为除了**重置** t_i 位之外，它最多将1位重置为1。
    - 若 $b_i = 0$,第 i 个操作重置所有 k 位 (k位都是1，到最大数了，加1操作后越界限，变成 1000..00，最高位 1 舍去), 因此 $b_{i\text{-}1} = t_i = k$. (k位1全变为0)
    - 若 $b_i > 0$,则 $b_i = b_{i\text{-}1} - t_i + 1$. (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))
    - 无论如何，$b_i \le b_{i\text{-}1} - t_i + 1$, 势能差是

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

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

- 因此平摊开销是

```
COUNTER(A, n)
    A = 0
    for j←1 to n
        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 势能法 - 对二进制计数器递增

- 我们定义第 i 个INCREMENT操作后计数器的势 Φ 为 b_i，即计数器中1的个数。(第i次操作后，势函数b_i为计数器中1的个数)
- 因此平摊开销是

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

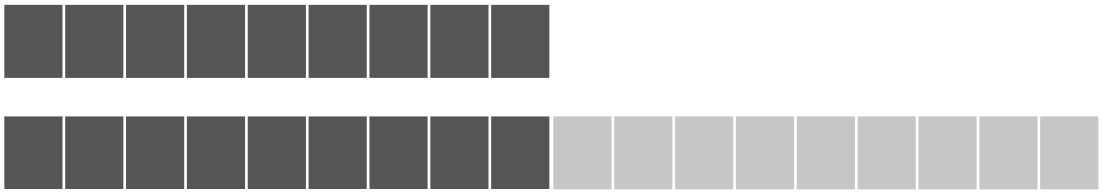
- 因此最坏情况COUNTER的代价为 O(n)?

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

```
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
```

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

17.4 动态表



void * malloc(size_t size);

c : malloc/free

c++: new/delete

- 我们可能会为一个表分配空间，但后来发现它是不够的。然后，我们必须重新分配更大的表，并将存储在原始表中的所有对象复制到新的更大的表中。
- 类似地，如果从表中删除了许多对象，那么重新分配一个较小的表可能是值得的。
- 我们假设动态表支持插入和删除操作。
  - 插入操作将一个项目插入到表中，该项目占用一个槽位，即一个项目的空间。
  - 同样的, 删除操作从表中删除一个项，从而释放一个槽。

17.4 动态表

- 用于组织表的数据结构的细节并不重要。 它可能是:
  堆栈， 堆， 哈希表， 或者数组

- 非空表 T 的负载因子 $\alpha(T)$:

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

- 我们分配一个空表(一个没有项的表)，大小为0，并将其负载因子定义为1。

17.4.1 扩展表

- 我们假设表的存储空间被分配为插槽数组。
- 在向表中插入项直至全满后，我们可以通过分配比旧表拥有更多槽的新表来扩展表。
- 一种常见的启发式分配新表的槽数是旧表的两倍。
  - 如果唯一的表操作是插入，那么表的负载因子总是至少是1/2，因此浪费的空间量永远不会超过表中总空间的一半。

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

```
TABLE-INSERT(T, x)
if T.size == 0
    allocate T.table with 1 slot
    T.size = 1
if T.num == T.size
    allocate new-table with 2 · T.size slots
    insert all items in T.table into new-table
    free T.table
    T.table = new-table
    T.size = 2 · T.size
insert x into T.table
T.num = T.num + 1
```

17.4.1 扩展表

- 扩展表的运行时间?

```
TABLE-EXPANSION(T, n)
    T = NULL
    for j←1  to  n
        TABLE-INSERT(T, x)
```

```
TABLE-INSERT(T, x)
if T.size == 0
    allocate T.table with 1 slot
    T.size = 1
if T.num == T.size
    allocate new-table with 2 · T.size slots
    insert all items in T.table into new-table
    free T.table
    T.table = new-table
    T.size = 2 · T.size
insert x into T.table
T.num = T.num + 1
```

- 第 i 个操作的开销 c_i 是多少?
  - 如果当前表有空间容纳新项(或者是第一个操作)，那么 $c_i = 1$?
  - 如果当前表已满，则发生扩展，那么 $c_i = i$?
- 最坏情况操作的代价是 O(n) ?
- 扩展表的上界是O(n^2) ?
- 正确。不紧?

(1) 聚合分析

- 扩展表的运行时间？

```
TABLE-EXPANSION(T, n)
    T = NULL
    for j←1 to n
        TABLE-INSERT(T, x)
```

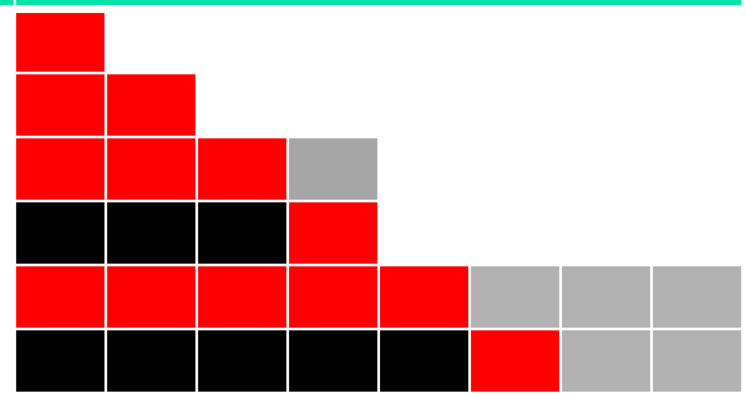
- 第 i 个操作的开销 c_i 是多少?

$$c_i = \begin{cases} i \ , & \text{if } \ i\text{-}1=2^k, \\ 1 \ , & \text{otherwise.} \end{cases}$$

$$\begin{aligned} \sum_{i=1}^{n} c_i &\le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< n + 2n \\ &= 3n \ , \end{aligned}$$

```
TABLE-INSERT(T, x)
if T.size == 0
    allocate T.table with 1 slot
    T.size = 1
if T.num == T.size
    allocate new-table with 2 · T.size slots
    insert all items in T.table into new-table
    free T.table
    T.table = new-table
    T.size = 2 · T.size
insert x into T.table
T.num = T.num + 1
```

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



Red: 执行的增加元素操作（次数）

Gray: 空槽 null slot

Black: 有元素的槽

(1) 聚合分析

- 扩展表的运行时间？

```
TABLE-EXPANSION(T, n)
    T = NULL
    for j←1  to  n
        TABLE-INSERT(T, x)
```

```
TABLE-INSERT(T, x)
    if T.size == 0
        allocate T.table with 1 slot
        T.size = 1
    if T.num == T.size
        allocate new-table with 2 · T.size slots
        insert all items in T.table into new-table
        free T.table
        T.table = new-table
        T.size = 2 · T.size
    insert x into T.table
    T.num = T.num + 1
```

第 i 个操作的开销 c_i 是多少？

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	...
	1	2		4				8	...

$$c_i = \begin{cases} i \ , & \text{if } \ i\text{-}1 = 2^k , \\ 1 \ , & \text{otherwise.} \end{cases}$$

$$\begin{aligned} \sum_{i=1}^{n} c_i &\le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< n + 2n \\ &= 3n , \end{aligned}$$

(2) 记账方法

- 扩展表的运行时间？

```
TABLE-EXPANSION(T, n)
    T = NULL
    for j←1  to  n
        TABLE-INSERT(T, x)
```

```
TABLE-INSERT(T, x)
if T.size == 0
    allocate T.table with 1 slot
    T.size = 1
if T.num == T.size
    allocate new-table with 2 · T.size slots
    insert all items in T.table into new-table
    free T.table
    T.table = new-table
    T.size = 2 · T.size
insert x into T.table
T.num = T.num + 1
```

- 第 i 个操作的开销 c_i 是多少?

$$c_i = \begin{cases} i\ , & \text{if } i\text{-}1=2^k, \\ 1\ , & \text{otherwise.} \end{cases}$$

$$\begin{aligned} \sum_{i=1}^{n} c_i &\le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< n + 2n \\ &= 3n\ , \end{aligned}$$

- 每次插入的平摊费用是3美元，实际插入1美元，信用2美元。

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		

(2) 记账方法

	1
存款	1

插入1(注:第一次平摊代价为2,其余为3)

	1	
存款	0	

扩张

	1	2
存款	1	1

插入2

	1	2		
存款	0	0		

扩张

	1	2	3	
存款	1	0	1	

插入3

https://blog.csdn.net/rebort

i	1	2	3	4
存款	1	1	1	1

插入4

i	1	2	3	4				
存款	0	0	0	0				

扩张

i	1	2	3	4	5			
存款	1	0	0	0	1			

插入5

i	1	2	3	4	5	6		
存款	1	1	0	0	1	1		

插入6

https://blog.csdn.net/rebort

(3) 势能方法

- 势能函数：$\Phi(T) = 2T.num - T.size$
  - 在扩展之后，我们有 $T.num = T.size/2$, 因此 $\Phi(T) = 0$.
  - 在扩展之前，我们有 $T.num = T.size$, 因此 $\Phi(T) = T.num$.
- 如果第i个TABLE-INSERT操作没有触发扩展表操作，则$size_i = size_{i-1}$，该操作的平摊代价为

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

```
TABLE-EXPANSION(T, n)
    T = NULL
    for j←1  to  n
        TABLE-INSERT(T, x)
```

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

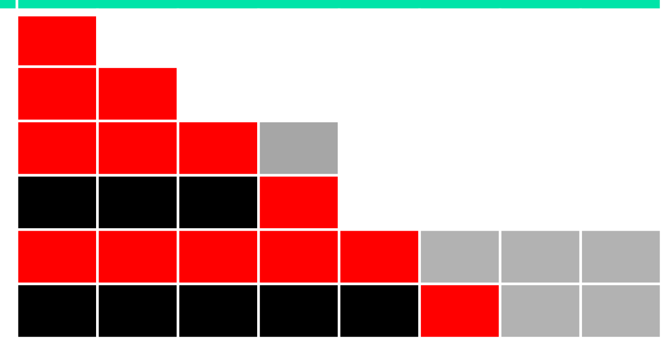
(3) 势能方法

- 势能函数：$\Phi(T) = 2T.num - T.size$
  - 在扩展之后，我们有 $T.num = T.size/2$, 因此 $\Phi(T) = 0$.
  - 在扩展之前，我们有 $T.num = T.size$, 因此 $\Phi(T) = T.num$.
- 如果第i个TABLE-INSERT操作触发扩展表操作，则$size_i = 2size_{i\text{-}1}$，且 $size_{i\text{-}1} = num_{i\text{-}1} = num_i - 1$，因此

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

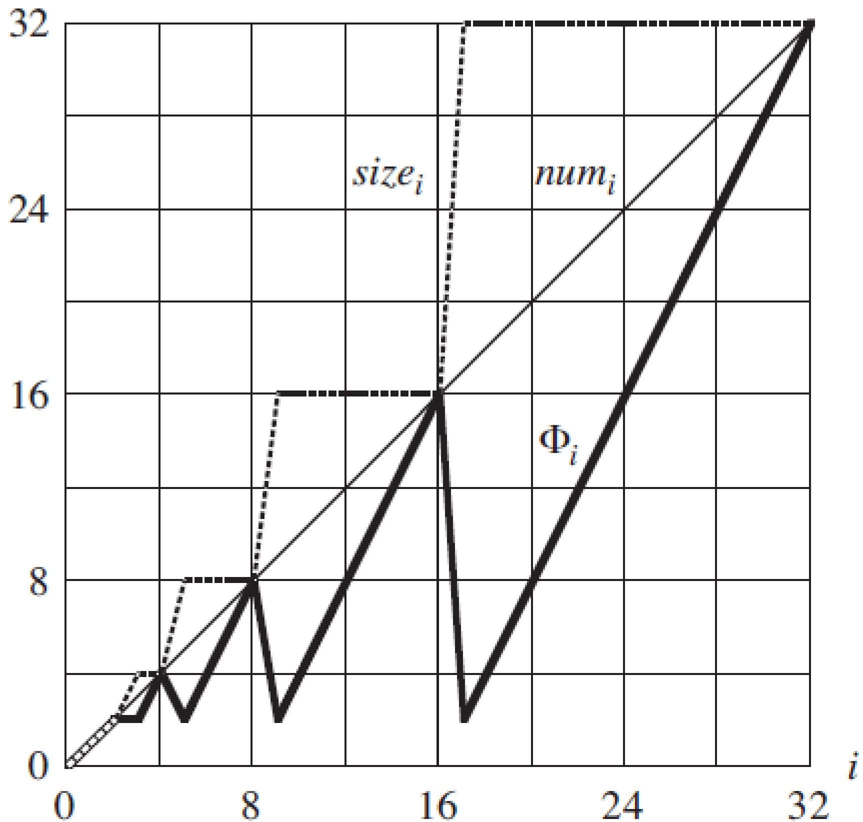
```
TABLE-EXPANSION(T, n)
    T = NULL
    for j←1  to  n
        TABLE-INSERT(T, x)
```

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



(3) 势能方法

- 势能函数 : $\Phi(T) = 2T.num - T.size$
  - 在扩展之后，我们有 $T.num = T.size/2$, 因此 $\Phi(T) = 0$.
  - 在扩展之前，我们有 $T.num = T.size$, 因此 $\Phi(T) = T.num$.
- 对于第 i 个 TABLE-INSERT 操作, 平摊开销是 3




```
TABLE-EXPANSION(T, n)
    T = NULL
    for j←1 to n
        TABLE-INSERT(T, x)
```

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

17.4.2 Table expansion and contraction(收缩)

自学

Conclusion and exercise-in-class

	Aggregate analysis	The accounting method	The potential method
Stack operations			
Incrementing a binary counter			
Dynamic Table expansion			
Dynamic Table contraction			
convex hull			
KMP			
……			

exercises

- 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。