



HITWH  
SE

# 第六章

## Amortized Analysis



HITWH  
SE

# 提要



- 6.1 平摊分析基本原理
- 6.2 聚集方法
- 6.3 会计方法
- 6.4 势能方法
- 6.5 动态表平摊分析



HITWH  
SE

参考资料



# Chapter 17 Amortized Analysis

**Pages 405 - 430**



## 6.1 平摊分析基本原理

- Amortized Analysis 的基本思想
- Amortized Analysis 方法



关注一系列**数据结构上**操作的时间复杂度：

考虑操作序列： $OP_1, OP_2, \dots, OP_m$

想确定这个操作序列可能花费的最长时间

一个可能的想法：考察每种操作 $OP_i$ 的最坏情况时间复杂度 $t_i$ ，

由此，上述操作序列可能的最长时间就是：

**每种操作 $OP_i$ 的最坏时间 $t_i$ 之和**

**不一定足够精确！**

**操作之间实际上是相互关联的，**

**不能假设它们是相互独立的！**



例如：

- 普通栈操作及其时间代价
  - $\text{Push}(S, x)$ : 将对象 $x$ 压入栈 $S$
  - $\text{Pop}(S)$ : 弹出并返回 $S$ 的顶端元素
  - 两个操作的运行时间都是 $O(1)$
  - 可把每个操作的实际代价视为 $1$
  - $n$ 个Push和Pop操作系列的总代价是 $n$
  - $n$ 个操作的实际运行时间为 $\theta(n)$



- 新的栈操作及其时间代价

- Multipop( $S, k$ ):

去掉 $S$ 的 $k$ 个栈顶对象, 当 $|S| < k$ 时弹出整个栈

- 实现算法

Multipop( $S, k$ )

1 While not STACK-EMPTY( $S$ ) and  $k \neq 0$  Do

2       Pop( $S$ );

3        $k \leftarrow k - 1$ .

- Multipop( $S, k$ )的实际代价 (设Pop的代价为1)

- Multipop的代价为 $\min(|S|, k)$



- 初始栈为空的 $n$ 个栈操作序列的分析
  - $n$ 个栈操作序列由Push、Pop和Multipop组成
  - 粗略分析
    - 最坏情况下，每个操作都是Multipop
    - 栈的大小至多为 $n$
    - 每个Multipop的代价最坏是 $O(n)$
    - 操作系列的最坏代价为 $T(n) = O(n^2)$

分析过于粗糙，不够精确！

原因：只关注于操作，忽略了数据结构！





# Amortized Analysis的基本思想

平摊分析目的是分析给定  
数据结构上的 $n$ 个操作代价上界

各个操作的平均代价？  
整个序列的代价是多少？

对一个数据结构  
执行一个操作序列：

有的代价很高  
有的代价一般  
有的代价很低

将总的代价平摊到  
每个操作上

平摊  
代价

不涉及概率  
异于平均分析



# Amortized Analysis的基本思想



- Aggregate Analysis方法（每个操作的代价）
  - 确定 $n$ 个操作的上界 $T(n)$ , 每个操作平摊 $T(n)/n$
- Accounting方法（整个操作序列的代价）
  - 不同类型操作赋予不同的平摊代价
  - 某些操作在数据结构的特殊对象上“预付”代价
- Potential方法（整个操作序列的代价）
  - 不同类型操作赋予不同的平摊代价
  - “预付”的代价作为整个数据结构的“能量”



## 6.2 Aggregate Analysis

- 聚集方法的原理
- 聚集方法的实例之一
- 聚集方法的实例之二



# 聚集方法的原理

目的是分析 $n$ 个操作系列中  
每个操作的复杂性上界

数据结构上共有 $n$ 个  
操作, 最坏情况下:

操作1:  $\text{Cost}=t_1$

操作2:  $\text{Cost}=t_2$

⋮

⋮

⋮

操作 $n$ :  $\text{Cost}=t_n$

$$T(n) = \sum_{i=1}^n t_i$$

平摊代价:  
 $T(n)/n$

每个操作被赋予相同  
代价, 不管操作类型



- 普通栈操作及其时间代价
  - $\text{Push}(S, x)$ : 将对象 $x$ 压入栈 $S$
  - $\text{Pop}(S)$ : 弹出并返回 $S$ 的顶端元素
  - 两个操作的运行时间都是 $O(1)$
  - 可把每个操作的实际代价视为 $1$
  - $n$ 个Push和Pop操作系列的总代价是 $n$
  - $n$ 个操作的实际运行时间为 $\theta(n)$



- 初始栈为空的 $n$ 个栈操作序列的分析
  - $n$ 个栈操作序列由Push、Pop和Multipop组成

- 粗略分析

- 最坏情况下，每个操作都是Multipop
- 每个Multipop的代价最坏是 $O(n)$
- 操作系列的最坏代价为 $T(n) = O(n^2)$
- 平摊代价为 $T(n)/n = O(n)$

分析太粗糙  
!!!

- 精细分析

- 一个对象在每次被压入栈后至多被弹出一次
- 在非空栈上调用Pop的次数(包括在Multipop内的调用)至多为Push执行的次数, 即至多为 $n$
- 最坏情况下操作序列的代价为 $T(n) \leq 2n = O(n)$
- 平摊代价= $T(n)/n = O(1)$

$n-1$ 个push  
1个multipop



## 聚集方法实例之二： 二进制计数器

- 问题定义： 由0开始计数的 $k$ 位二进制计数器

输入：  $k$ 位二进制变量 $x$ ，初始值为0

输出：  $x+1 \bmod 2^k$

数据结构：

$A[0..k-1]$ 作为计数器，存储 $x$

$x$ 的最低位在 $A[0]$ 中，最高位在 $A[k-1]$ 中

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



- 计数器加1算法

输入:  $A[0..k-1]$ , 存储二进制数  $x$

输出:  $A[0..k-1]$ , 存储二进制数  $x+1 \bmod 2^k$

INCREMENT( $A$ )

1      $i \leftarrow 0$

2     while  $i < k$  and  $A[i] = 1$  Do

3          $A[i] \leftarrow 0;$

4          $i \leftarrow i + 1;$

5     If  $i < \text{length}[A]$  Then  $A[i] \leftarrow 1$



操作Cost =  $O(\text{发生改变的位数})$

- 初始为零的计数器上 $n$ 个INCREMENT操作分析

Counter Value									每个操作 Cost
N	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[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	2
3	0	0	0	0	0	0	1	1	1
4	0	0	0	0	0	1	0	0	3
5	0	0	0	0	0	1	0	1	1
6	0	0	0	0	0	1	1	0	2
7	0	0	0	0	0	1	1	1	1
8	0	0	0	0	1	0	0	0	4
9	0	0	0	0	1	0	0	1	1
10	0	0	0	0	1	0	1	0	2
11	0	0	0	0	1	0	1	1	1
12	0	0	0	0	1	1	0	0	3
13	0	0	0	0	1	1	0	1	1
14	0	0	0	0	1	1	1	0	2
15	0	0	0	0	1	1	1	1	1
16	0	0	0	1	0	0	0	0	5



- 粗略分析

- 每个Increment的时间代价最多  $O(k)$
- $n$ 个Increment序列的时间代价最多  $O(kn)$
- $n$ 个Increment平摊代价为  $O(k)$
- 例如上例中:  $k*n=8*16=128$

- 精细分析



操作Cost =  $O(\text{发生改变的位数})$

Counter Value										Total Cost
N	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]		
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



## • 精细分析

- $A[0]$  每次操作发生一次改变，总次数为  $n$
- $A[1]$  每两次操作发生一次改变，总次数为  $n/2$
- $A[2]$  每四次操作发生一次改变，总次数为  $n/4$
- 一般地
  - 对于  $i=0, 1, \dots, \lg n$ ,  $A[i]$  改变次数为  $n/2^i$
  - 对于  $i > \lg n$ ,  $A[i]$  不发生改变  
(因为  $n$  个操作结果为  $n$ , 仅涉及  $A[0]$  至  $A[\lg n]$  位)
- $T(n) \leq \sum_{0 \leq i \leq \lg n} n/2^i < n \sum_{0 \leq i \leq \infty} 1/2^i = O(n)$
- 每个 Increment 操作的平摊代价为  $O(1)$



## 6.3 The Accounting Method

- Accounting方法的原理
- Accounting方法的实例之一
- Accoutning方法的实例之二



- Accounting方法

- 目的是分析 $n$ 个操作的序列的复杂性上界，操作序列中有不同类型的操作，不同类型操作的代价各不相同
- 于是我们为每种操作分配不同的平摊代价
  - 平摊代价可能比实际代价大，也可能比实际代价小
  - 如果平摊代价比实际代价高：一部分用于支付实际代价，多余部分作为Credit附加在数据结构的具体数据对象上
  - 当一个操作的平摊代价比实际代价低时：Credit用来补充支付实际代价
- 平摊代价的选择规则：
  - 设 $\alpha_i$ 和 $c_i$ 是操作 $i$ 的平摊代价和实际代价

数据结构中存储的Credit在任何时候都必须非负，

$$\text{即 } \sum_{1 \leq i \leq n} \alpha_i - \sum_{1 \leq i \leq n} c_i \geq 0 \text{ 永远成立}$$



# 栈操作序列的分析

- 各栈操作的实际代价
  - $\text{Cost}(\text{PUSH})=1$
  - $\text{Cost}(\text{POP})=1$
  - $\text{Cost}(\text{MULTIPOP})=\min\{k, s\}$
- 各栈操作的平摊代价
  - $\text{Cost}(\text{PUSH})=2$ 
    - 一个1用来支付PUSH的开销,
    - 另一个1存储在压入栈的元素上,预支POP的开销
  - $\text{Cost}(\text{POP})=0$
  - $\text{Cost}(\text{MULTIPOP})=0$
- 平摊代价满足
  - $\sum_{1 \leq i \leq n} \alpha_i - \sum_{1 \leq i \leq n} c_i \geq 0$  对于任意 $n$ 个操作序列都成立
    - 因为栈中的元素个数 $\geq 0$
- $n$ 个栈操作序列的总平摊代价
  - $O(n)$



# 二进制计数器Increment操作序列分析

- Increment操作的平摊代价
  - 每次一位被置1时，付代价2
    - 代价1用于置1的开销
    - 代价1存储在“1”位上，用于支付其被置0时的开销
    - 置0操作无需再付款
  - $\text{Cost}(\text{Increment})=2$
- 平摊代价满足
  - $\sum_{1 \leq i \leq n} \alpha_i \geq \sum_{1 \leq i \leq n} c_i$  对于任意 $n$ 个操作序列都成立
  - 即：任何时刻，计数器上的Credit非负
- $n$ 个Increment操作序列的总平摊代价
  - $O(n)$



- 初始为零的计数器上 $n$ 个INCREMENT操作分析

Counter Value										每个操作 Cost	
N	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]			
0	0	0	0	0	0	0	0	0			
1	0	0	0	0	0	0	0	1	1	1	
2	0	0	0	0	0	0	1	0	0	0	2
3	0	0	0	0	0	0	1	1	1	1	1
4	0	0	0	0	0	1	0	0	0	0	3
5	0	0	0	0	0	1	1	0	0	1	1
6	0	0	0	0	0	1	1	1	0	0	2
7	0	0	0	0	0	1	1	1	1	1	1
8	0	0	0	0	1	0	0	0	0	0	4
9	0	0	0	0	1	1	0	0	0	1	1
10	0	0	0	0	1	1	0	1	1	0	2
11	0	0	0	0	1	1	0	1	1	1	1
12	0	0	0	0	1	1	1	0	0	0	3
13	0	0	0	0	1	1	1	0	0	1	1
14	0	0	0	0	1	1	1	1	1	0	2
15	0	0	0	0	1	1	1	1	1	1	1
16	0	0	0	1	0	0	0	0	0	0	5



## 6.4 The Potential Method

- Potential方法的原理
- Potential方法的实例之一
- Potential方法的实例之二



- Potential方法

- 目的是分析 $n$ 个操作的序列的复杂性上界
- 在会计方法中，如果操作的平摊代价比实际代价大，我们将余额与数据结构的数据对象相关联
- Potential方法把Credit余额与整个数据结构关联，所有这样的余额之和，构成数据结构的势能
  - 如果操作的平摊代价大于操作的实际代价，势能增加
  - 如果操作的平摊代价小于操作的实际代价，要用数据结构的势能来支付实际代价，势能减少



## • 数据结构势能的定义

— 考虑在初始数据结构 $D_0$ 上执行 $n$ 个操作

— 对于操作 $i$

- 操作 $i$ 的实际代价为 $c_i$
- 操作 $i$ 将数据结构 $D_{i-1}$ 变为 $D_i$
- 数据结构 $D_i$ 的势能是一个实数 $\phi(D_i)$ ,  $\phi$ 是一个正函数
- 操作 $i$ 的平摊代价:  $\alpha_i = c_i + \phi(D_i) - \phi(D_{i-1})$

—  $n$ 个操作的总平摊代价

$$\begin{aligned}\sum_{i=1}^n \alpha_i &= \sum_{i=1}^n (c_i + \phi(D_i) - \phi(D_{i-1})) \\ &= \sum_{i=1}^n c_i + \phi(D_n) - \phi(D_0)\end{aligned}$$

总平摊代价必须是  
总实际代价的上界

$$\sum_{i=1}^n \alpha_i \geq \sum_{i=1}^n c_i$$

— 关键是 $\phi$ 的定义

- 保证 $\phi(D_n) \geq \phi(D_0)$ , 使总平摊代价是总实际代价的上界
- 如果对于所有 $i$ ,  $\phi(D_i) \geq \phi(D_0)$ , 可以保证 $\phi(D_n) \geq \phi(D_0)$
- 实际可以定义 $\phi(D_0) = 0$ ,  $\phi(D_i) \geq 0$  (由具体问题确定)



- 栈的势能定义

- $\phi(D_m)$  定义为栈  $D_m$  中对象的个数，于是

- $\phi(D_0) = 0$ ,  $D_0$  是空栈
- $\phi(D_i) \geq 0 = \phi(D_0)$ , 因为栈中对象个数不会小于 0
- $n$  个操作的总平摊代价是实际代价的上界

- 栈操作的平摊代价 (设栈  $D_{i-1}$  中具有  $s$  个对象)

- PUSH:  $\alpha_i = c_i + \phi(D_i) - \phi(D_{i-1}) = 1 + (s+1) - s = 2$
- POP:  $\alpha_i = c_i + \phi(D_i) - \phi(D_{i-1}) = 1 + (s-1) - s = 0$
- MULTIPOP( $S, k$ ): 设  $k' = \min(k, s)$

$$\alpha_i = c_i + \phi(D_i) - \phi(D_{i-1}) = k' + (s - k') - s = k' - k' = 0$$

- $n$  个栈操作序列的平摊代价是  $O(n)$



# 二进制计数器操作序列分析

- 计数器的势能定义

- $\phi(D_m)$  定义为第  $m$  个操作后计数器中 1 的个数  $b_m$

- $\phi(D_0) = 0$ , 0 的计数中不包含 “1”
    - $\phi(D_i) \geq 0 = \phi(D_0)$ , 因为计数器中 1 的个数不会小于 0
    - 于是,  $n$  个操作的总平摊代价是实际代价的上界

- INCREMENT 操作的平摊代价

- 第  $i$  个 INCREMENT 操作把  $t_i$  个位置成 0, 实际代价为  $c_i \leq t_i + 1$

- 计算操作  $i$  的平摊代价  $\alpha_i = c_i + \phi(D_i) - \phi(D_{i-1})$

- If  $b_i = 0$ , 操作  $i$  把所有  $k$  位置 0, 所以  $b_{i-1} = k$ ,  $t_i = k$

- If  $b_i > 0$ , 则  $b_i = b_{i-1} - t_i + 1$

- 于是  $b_i \leq b_{i-1} - t_i + 1$

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

- 平摊代价  $\alpha_i = c_i + \phi(D_i) - \phi(D_{i-1}) \leq (t_i + 1) + (1 - t_i) = 2$

- $n$  个操作序列的总平摊代价是  $O(n)$



## 6.5 动态表性能平摊分析

- 动态表的概念
- 动态表的扩张与收缩
- 仅含扩张操作的动态表平摊分析
- 一般的动态表平摊分析



- 动态表
- 动态表支持的操作
  - TABLE-INSERT: 将某一元素插入表中
  - TABLE-DELETE: 将一个元素从表中删除
- 数据结构: 用一个(一组)数组来实现动态表
- 非空表 $T$ 的装载因子 $\alpha(T)$ 
  - $\alpha(T) = T$ 存储的对象数/表大小
  - 如果动态表的装载因子以一个常数为下界, 则表中未使用的空间就始终不会超过整个空间的一个常数部分
  - 空表的大小为0, 装载因子为1





- 虽然插入和删除操作可能会引起表的扩张和收缩，从而具有较高的实际代价
- 但是，利用平摊分析能够证明，插入和删除操作的平摊代价为  $O(1)$
- 同时保证动态表中未用的空间始终不超过整个空间的一部分。



设 $T$ 表示一个动态表:

- $table[T]$ 是一个指向表示表的存储块的指针
- $num[T]$ 表 $T$ 中的数据项个数
- $size[T]$ 是 $T$ 的大小
- 开始时,  $num[T]=size[T]=0$



算法: TABLE—INSERT( $T, x$ )

1 If  $\text{size}[T]=0$  Then

2     获取一个大小为1的表  $\text{table}[T]$ ;

3      $\text{size}[T] \leftarrow 1$ ;  $\text{num}[T] \leftarrow 0$ ;

4 If  $\text{num}[T]=\text{size}[T]$  Then

5     获取一个大小为  $2 \times \text{size}[T]$  的新表 new-table;

6     将  $\text{table}[T]$  中元素插入 new-table;   /\*简单插入操作\*/

7     释放  $\text{table}[T]$ ;

8      $\text{table}[T] \leftarrow \text{new-table}$ ;

9      $\text{size}[T] \leftarrow 2 \times \text{size}[T]$ ;

10    将  $x$  插入  $\text{table}[T]$ ;

11     $\text{num}[T] \leftarrow \text{num}[T] + 1$ ;

/\*复杂的插入操作\*/

/\*开销为常数\*/

/\*开销取决于  $\text{size}[T]$ \*/



- 插入一个数组元素时,完成的操作包括
  - 分配一个比原表包含更多的槽的新表
  - 再将原表中的各项复制到新表中去
- 常用的启发式技术是分配一个比原表大一倍的新表,
  - 只对表执行插入操作,则表的装载因子总是至少为 $1/2$
  - 浪费掉的空间就始终不会超过表总空间的一半



# 初始为空的表上 $n$ 次插入操作的代价分析

## 聚集分析-粗略分析

### — 考察第 $i$ 次操作的代价 $C_i$

- 如果 $i=1$ ,  $C_i=1$ ;
- 如果 $\text{num}[T] < \text{size}[T]$ ,  $C_i=1$ ;
- 如果 $\text{num}[T] = \text{size}[T]$ ,  $C_i=i$ ;

### — 共有 $n$ 次操作

- 最坏情况下,每次都按照扩张操作计量,总的代价上界为 $n^2$

### — 这个界不精确

- $n$ 次TABLE—INSERT操作并不常常包括扩张表的代价
- 仅当 $i-1$ 为2的整数幂时第 $i$ 次操作才会引起一次表的扩张

## 聚集分析-精细分析

### — 第 $i$ 次操作的代价 $C_i$

- 如果 $i-1=2^m$ ,  $C_i=i$ ; 否则 $C_i=1$

### — $n$ 次TABLE—INSERT操作的总代价为 $\sum_{i=1}^n c_i \leq n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j < n + 2n = 3n$

★ 每一操作的平摊代价为 $3n/n=3$



## 会计方法

- 每次执行TABLE—INSERT平摊代价为3
  - 1支付第10步中的基本插入操作的实际代价
  - 1作为自身的存款
  - 1存入表中第一个没有存款的数据上
- 当发生表的扩张时，数据复制的代价由数据上的存款来支付
- 任何时候，存款总和 非负
- 初始为空的表上 $n$ 次TABLE-INSERT操作的平摊代价总和为 $3n$



## 势能法分析

势能怎么定义才能使得表满发生扩张时势能能支付扩张的代价?

- 如果势能函数满足

- 刚扩充完,  $\phi(T)=0$
- 表满时  $\phi(T)=size(T)$

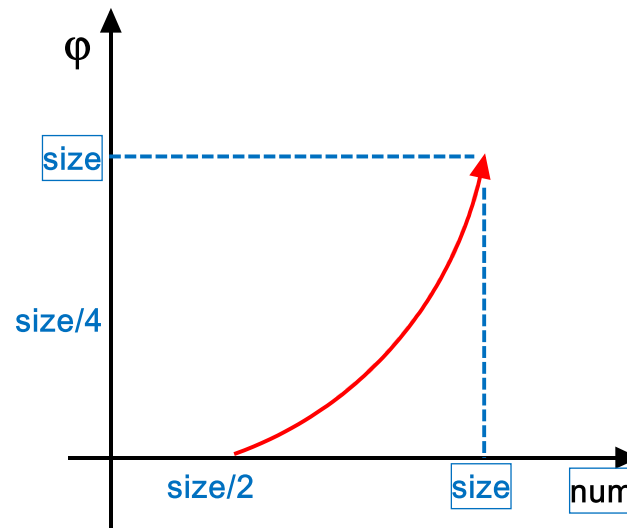
- 定义  $\phi(T)=2*num[T]-size[T]$

- 由于  $num[T] \geq size[T]/2$ , 故  $\phi(T) \geq 0$
- 因此,  $n$ 次TABLE-INSERT操作的总的平摊代价是总的实际代价的一个上界

- 第 $i$ 次操作的平摊代价

- 如果未发生扩张,  $\alpha_i=3$
- 如果发生扩张,  $\alpha_i=3$

- 初始为空的表上 $n$ 次插入操作的代价的上界为 $3n$

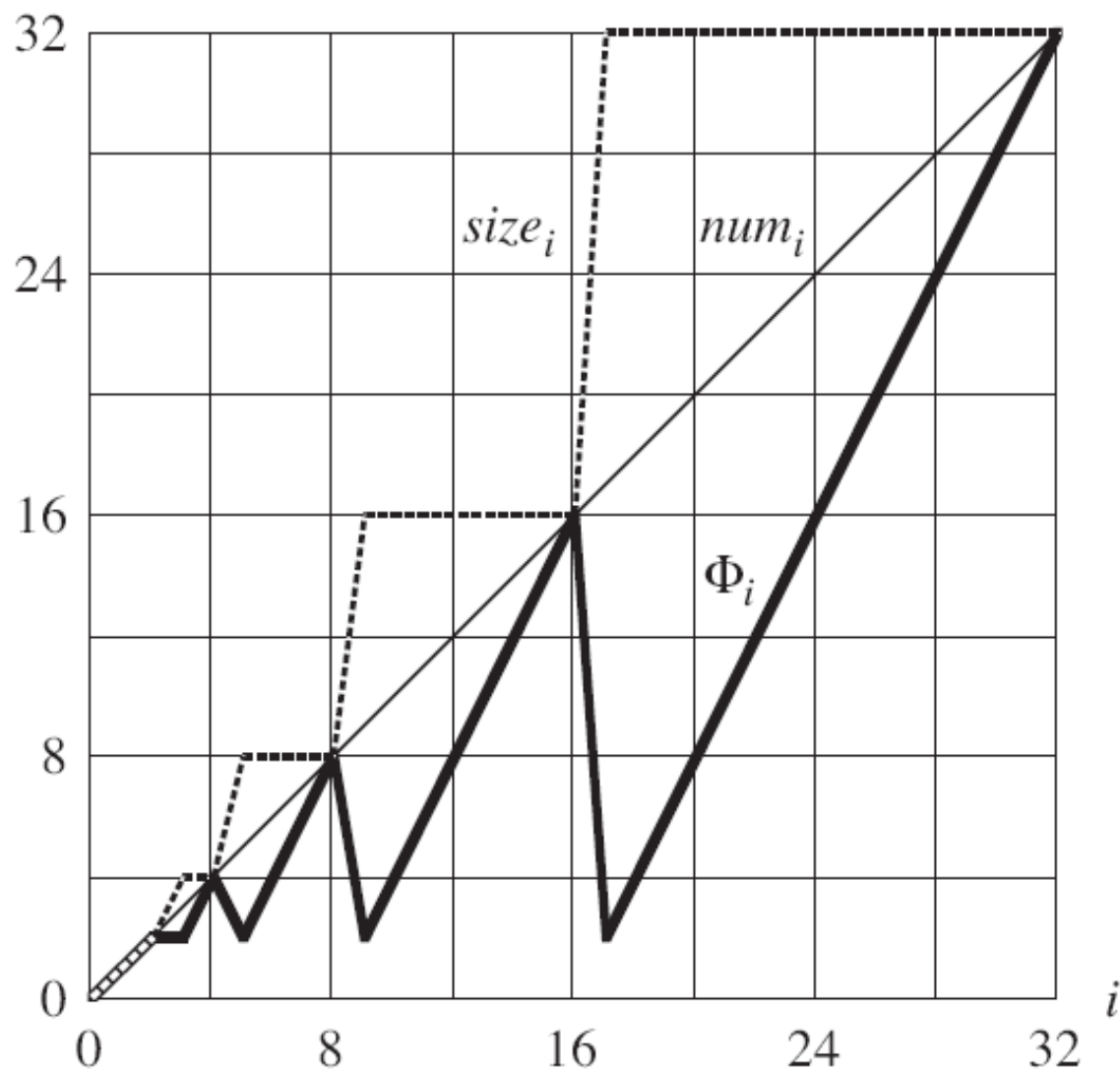




HITWH  
SE

# 初始为空的表上 $n$ 次插入操作的代价分析

## 32 Table-Insertions







- Table-Delete : 将指定的数据对象从表中删除
- 表的收缩: 当动态表的装载因子很小时, 对表进行收缩  
理想情况下, 我们希望:
  - 表具有一定的丰满度
  - 表的操作序列的复杂度是线性的
- 表的收缩策略
  - 表的装载因子小于 $1/2$ 时, 收缩表为原表的一半
  - $n = 2^k$ , 考察下面的一个长度为 $n$ 的操作序列:
    - 前 $n/2$ 个操作是插入, 后跟IDDIIDDII...
    - 每次扩张和收缩的代价为 $O(n)$ , 共有 $O(n)$ 扩张或收缩
    - 总代价为 $O(n^2)$ , 而每一次操作的平摊代价为 $O(n)$ --每个操作的平摊代价太高



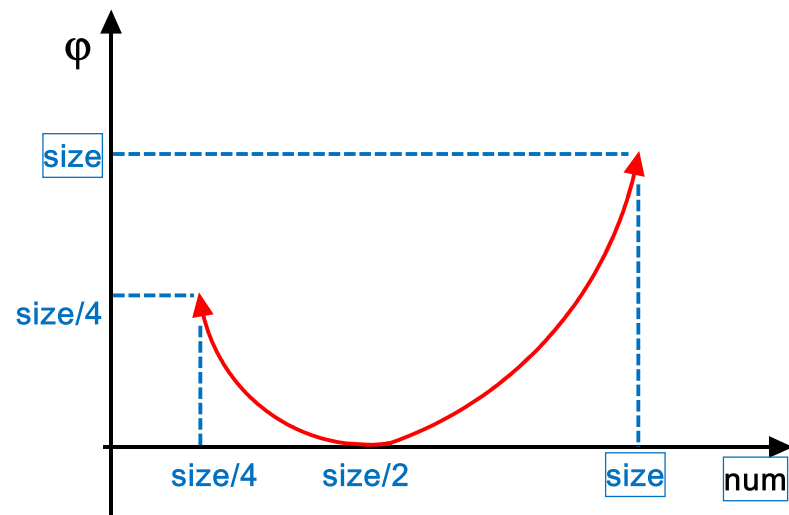
- 改进的收缩策略(允许装载因子低于 $1/2$ )
  - 满表中插入数据项时, 将表扩大一倍
  - 删除数据项引起表不足 $1/4$ 满时, 将表缩小为原表的一半
  - 扩张和收缩过程都使得表的装载因子变为 $1/2$
  - 表的装载因子的下界是 $1/4$



# 动态表上 $n$ 次(插入、删除)操作的代价分析

## 势能函数的定义

- 操作序列过程, 势能总是非负的
  - 保证一系列操作的总平摊代价即为其实际代价的一个上界
- 表的扩张和收缩过程要消耗大量的势
- 势能函数应满足
  - $num(T)=size(T)/2$ 时, 势最小
  - 当 $num(T)$ 减小时, 势增加直到收缩
  - 当 $num(T)$ 增加时, 势增加直到扩充
- 势能函数特征的细化
  - 当装载因子为 $1/2$ 时, 势为0
  - 装载因子为1时, 有 $num[T]=size[T]$ , 即 $\phi(T)=num[T]$ 。这样当因插入一项而引起一次扩张时, 就可用势来支付其代价
  - 当装载因子为 $1/4$ 时,  $size[T]=4 \cdot num[T]$ 。即 $\phi(T)=num[T]$ 。因而当删除某项引起一次收缩时就可利用势来支付其代价



$$\Phi(T) = \begin{cases} 2 \cdot num[T] - size[T] & \alpha(T) \geq 1/2 \\ size[T]/2 - num[T] & \alpha(T) < 1/2 \end{cases}$$



- 第 $i$ 次操作的平摊代价:  $\alpha_i = c_i + \phi(T_i) - \phi(T_{i-1})$ 
  - 第 $i$ 次操作是TABLE—INSERT: 未扩张  $\alpha_i \leq 3$
  - 第 $i$ 次操作是TABLE—INSERT: 扩张  $\alpha_i \leq 3$
  - 第 $i$ 次操作是TABLE—DELETE: 未收缩  $\alpha_i \leq 3$
  - 第 $i$ 次操作是TABLE—DELETE: 收缩  $\alpha_i \leq 3$
- 动态表上的 $n$ 个操作的实际时间为 $O(n)$



# 初始为空的表上 $n$ 次插入操作的代价分析

52次操作：Table-Insertion与Table-Deletion混合

