算法设计与分析

第七章 摊还分析

户保田
e-mail:hubaotian@hit.edu.cn

哈尔滨工业大学（深圳）计算机学院

提 纲

6.1 摊还分析原理

6.2 聚集方法

6.3 会计方法

6.4 势能方法

6.5 动态表操作的摊还分析

基本思想

在摊还分析中，执行一系列数据结构操作所需要时间是通过对执行的所有操作时间求平均而得出的

基本思想

对一个数据结构
要执行一系列操作
- 有的代价很高
- 有的代价一般
- 有的代价很低

将总的代价摊还到每个操作上

摊
还
代
价

注意！这里是平摊，不涉及操作的执行概率，不同于平均情况分析

提 纲

6.1 摊还分析原理

6.2 聚集方法

6.3 会计方法

6.4 势能方法

6.5 动态表操作的摊还分析

聚集分析法—原理

对数据结构共有n个操作

操作1： t_1

操作2： t_2

⋮

操作n: t_n

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

摊还代价: $T(n)/n$

聚集分析法实例1—栈操作

普通栈操作：

- PUSH(S,x)：将对象压入栈S
- POP(S)：弹出并返回S的顶端元素

时间代价:

- 两个操作的运行时间都是O(1)
- 我们可把每个操作的代价视为1个时间单位
- 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$\theta(n)$

聚集分析法实例1—栈操作

新的栈操作：

- MULTIPOP(S,k)
- 去掉S的k个顶端对象
- 当S中包含少于k个对象时弹出整个栈

聚集分析法实例1—栈操作

输入： 栈S，k

输出：返回S顶端k个对象

```
MULTIPOP(S,k):
    While  not STACK-EMPTY(S) and  k≠0  Do
        POP(S);
        k←k-1
```

- 实际运行时间与实际执行的POP操作数成线性关系
- While循环一次，调用一次POP,循环执行的次数为：min(s,k)

聚集分析法实例1—栈操作

- 初始为空的栈上的n个栈操作序列的分析
  - 由PUSH、POP和MULTIPOP组成的长为n的栈操作序列
  - 最坏情况下的时间复杂度是多少?

最坏情况下，每个操作都是：MULTIPOP，每个MULTIPOP的代价最坏是n

$$T(n)=n^2$$

这样分析太粗糙了！完全没有考虑到数据结构的特点！

聚集分析法实例1—栈操作

- 一个对象在每次被压入栈后至多被弹出1次
- 一个非空栈上POP次数(包括在MULTIPOP内的调用)至多等于PUSH的次数
- 在n个栈操作序列上POP多执行n-1次

$$T(n)<2n$$

- 摊还代价为：$T(n)/n=O(1)$
- 最坏情况下这样的一个操作序列的时间复杂度最多为$O(n)$
- 注意：求解摊还代价过程中，我们没有使用任何的概率

聚集分析法实例2—二进制计数器

- 一个由 0 开始的k位二进制加法计数器A
- 存储k位二进制变量x，初始值为0
- 数据结构：A[0..k-1]存储x，最低位A[0]，最高位A[k-1]
- 操作：INCREMENT(A)每次加1

输入：A[0..k-1]存储的二进制数x

输出：A[0..k-1]存储的二进制数$x+1 \bmod 2^k$

INCREMENT(A):

1. $i \leftarrow 0$
2. while $i<length[A]$ and $A[i]=1$ Do
3. $\quad A[i] \leftarrow 0$;
4. $\quad i \leftarrow i+1$;
5. If $i<length[A]$ Then $A[i] \leftarrow 1$

聚集分析法实例2—二进制计数器

- 初始为零的计数器上n个INCREMENT操作的分析

每8次发生1次共改变n/8次

每4次发生1次共改变n/4次

每2次发生1次共改变n/2次

每1次发生1次共改变n次

Counter N	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	tal
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

- 每次INCREMENT操作的代价与被改变值的字位的个数成线性关系
- 粗糙分析：每次INCREMENT操作最多改变k位，n次操作，代价为nk
- 摊还分析：$T(n) = \sum_{i=0}^{\lfloor log_2 n \rfloor} n/2^i < 2n$次改变，故摊还代价为T(n)/n=O(1)

提纲

6.1 摊还分析原理

6.2 聚集方法

6.3 会计方法

6.4 势能方法

6.5 动态表操作的摊还分析

会计方法—基本原理

- **一个操作序列中有不同类型操作，不同类型操作代价不相同**
  - 为每种操作分配不同的摊还代价
  - 摊还代价可能比实际代价大，也可能比实际代价小。
- **操作被执行时，支付摊还代价**
  - 如果摊还代价比实际代价高：摊还代价的一部分用于支付实际代价，多余部分作为存款附加在数据对象上
  - 如果摊还代价比实际代价低：摊还代价及数据对象上的存款用来支付实际代价
- **摊还代价的总和与实际代价的总和的关系**
  - **只要我们能保证**：在任何操作序列上，存款的总额非负，则所有操作摊还代价的总和就是实际代价总和的上界

会计方法—基本原理

在各种操作上定义摊还代价，使得任意操作序列上存款总量是非负的，将操作序列上平摊代价求和即可得到这个操作序列的复杂度上界

会计方法实例1—栈操作

- **各栈操作的实际代价**：

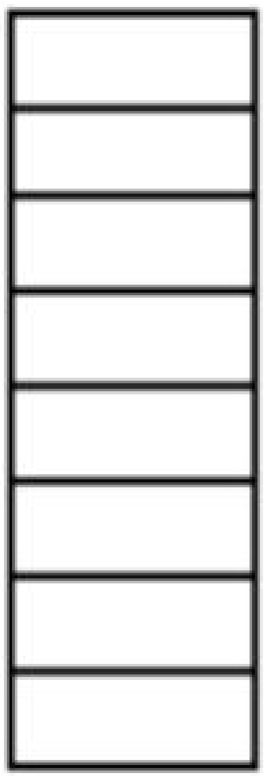
PUSH	1
POP	1
MULTIPOP	min(k,s)

- **各栈操作的摊还代价：**

PUSH	2
POP	0
MULTIPOP	0

会计方法实例1—栈操作

- 栈操作序列代价分析

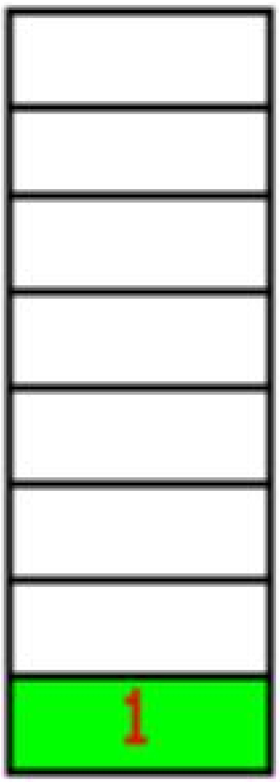


操作执行时

- 摊还代价比实际代价高：一部分用于支付实际代价，多余部分作为存款附加在具体数据对象上
- 摊还代价比实际代价低：摊还代价及数据对象上的存款用来支付实际代价

会计方法实例1—栈操作

- 栈操作序列代价分析



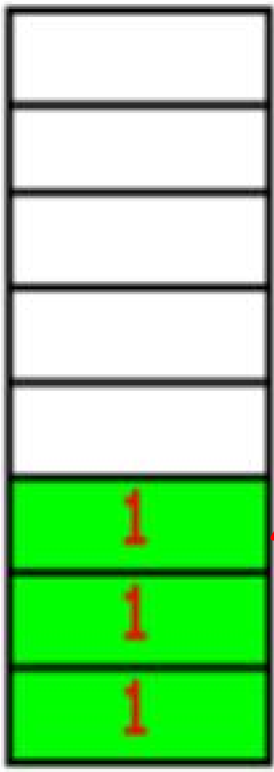
插入一个元素支付摊还代价2，1用于支付实际代价，1作为存款附着在数据对象上

操作执行时

- 摊还代价比实际代价高：一部分用于支付实际代价，多余部分作为存款附加在具体数据对象上
- 摊还代价比实际代价低：摊还代价及数据对象上的存款用来支付实际代价

会计方法实例1—栈操作

- 栈操作序列代价分析



插入一个元素支付摊还代价2，1用于支付实际代价，1作为存款附着在数据对象上

操作执行时

- 摊还代价比实际代价高：一部分用于支付实际代价，多余部分作为存款附加在具体数据对象上
- 摊还代价比实际代价低：摊还代价及数据对象上的存款用来支付实际代价

会计方法实例1—栈操作

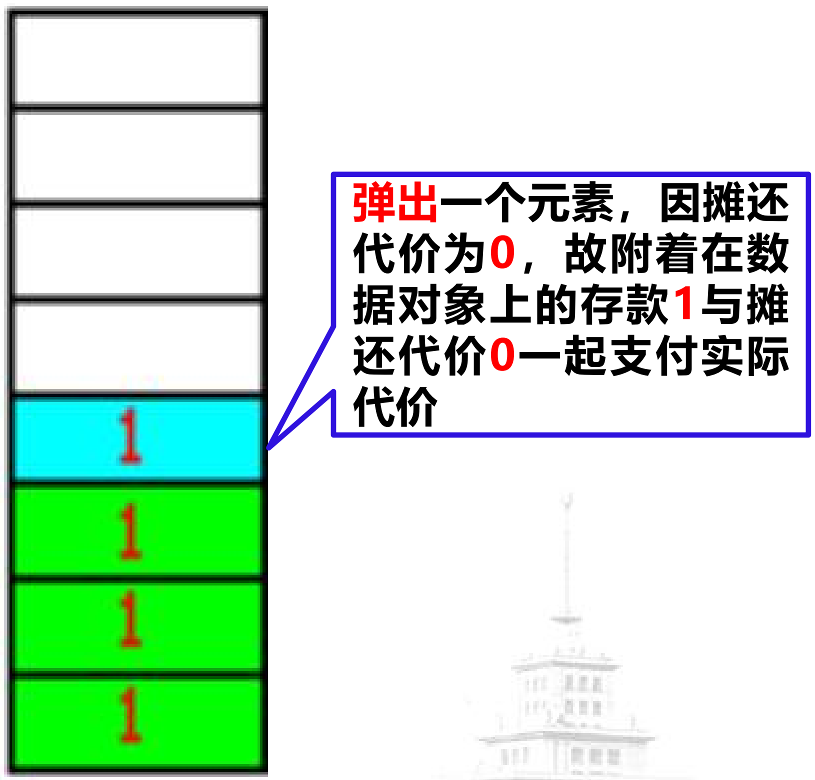
- 栈操作序列代价分析

操作执行时

- 摊还代价比实际代价高：一部分用于支付实际代价，多余部分作为存款附加在具体数据对象上
- 摊还代价比实际代价低：摊还代价及数据对象上的存款用来支付实际代价

会计方法实例1—栈操作

- 栈操作序列代价分析

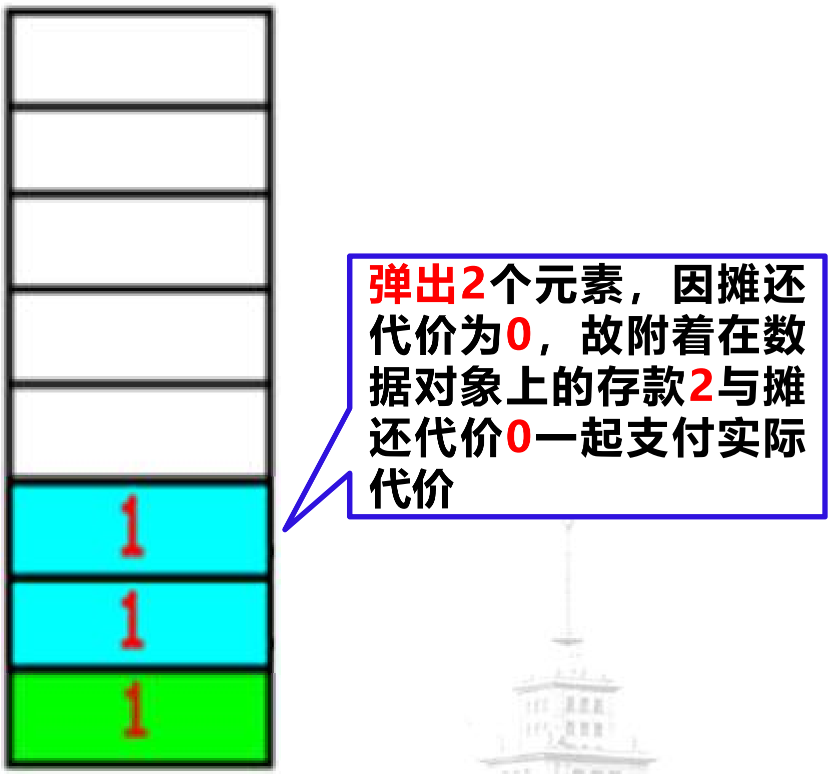


操作执行时

- 摊还代价比实际代价高：一部分用于支付实际代价，多余部分作为存款附加在具体数据对象上
- 摊还代价比实际代价低：摊还代价及数据对象上的存款用来支付实际代价

会计方法实例1—栈操作

- 栈操作序列代价分析

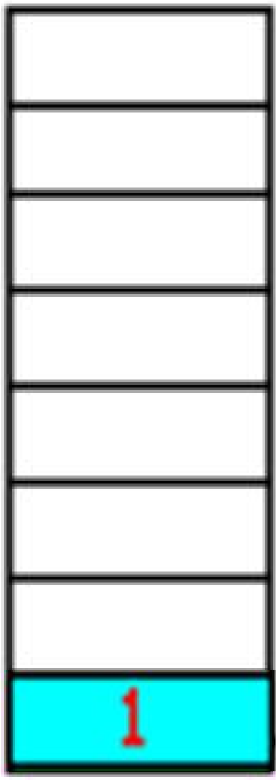


操作执行时

- 摊还代价比实际代价高：一部分用于支付实际代价，多余部分作为存款附加在具体数据对象上
- 摊还代价比实际代价低：摊还代价及数据对象上的存款用来支付实际代价

会计方法实例1—栈操作

- 栈操作序列代价分析



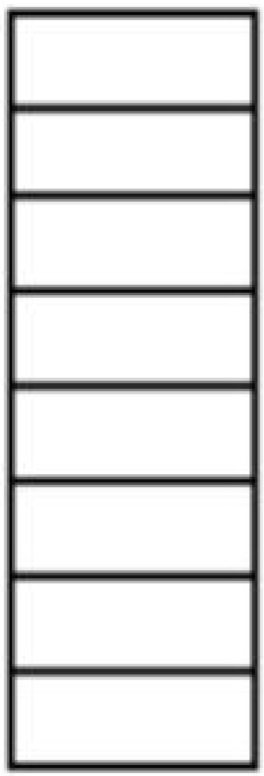
弹出一个元素，因摊还代价为**0**，故附着在数据对象上的存款**1**与摊还代价**0**一起支付实际代价

操作执行时

- 摊还代价比实际代价高：一部分用于支付实际代价，多余部分作为存款附加在具体数据对象上
- 摊还代价比实际代价低：摊还代价及数据对象上的存款用来支付实际代价

会计方法实例1—栈操作

- 栈操作序列代价分析



操作执行时

- 摊还代价比实际代价高：一部分用于支付实际代价，多余部分作为存款附加在具体数据对象上
- 摊还代价比实际代价低：摊还代价及数据对象上的存款用来支付实际代价

会计方法实例1—栈操作

- **栈操作序列代价分析**
  - 只要我们的操作序列是合理的，则可以保证存款总和非负
  - 所有操作的摊还代价总和就是操作序列实际代价总和的上界=?
  - 长度为n的操作序列中：PUSH操作的个数<=n，故摊还代价的总和<=2n，所以操作序列的实际代价为O(n)

会计方法实例2—二进制计数器

- 一个由 0 开始的k位二进制加法计数器A
- 存储k位二进制变量x，初始值为0
- 数据结构：A[0..k-1]存储x，最低位A[0]，最高位A[k-1]
- 操作：INCREMENT(A)每次加1

输入：A[0..k-1]存储的二进制数x

输出：A[0..k-1]存储的二进制数x+1 mod 2^k

```
INCREMENT(A):
    i←0
    while  i<length[A] and A[i]=1  Do
          A[i]←0;
          i←i+1;
    If  i<length[A]  Then  A[i]←1
```

会计方法实例2—二进制计数器

- 初始为零的计数器上n个INCREMENT操作的分析
  - 操作序列上的总代价与0-1或者1-0翻发生的次数成正比
  - 定义0-1翻转的摊还代价为2
  - 定义1-0翻转的摊还代价为0

任何操作序列，存款余额是计数器中1的个数，因此非负。所以，所有翻转操作的摊还代价的和是这个操作序列代价的上界

会计方法实例2—二进制计数器

- 初始为零的计数器上n个INCREMENT操作的分析
  - 操作序列上的总代价与0-1或者1-0翻发生的次数成正比
  - 定义0-1翻转的摊还代价为2
  - 定义1-0翻转的摊还代价为0

对每个INCREMENT操作

- 只做一次将0翻转成1：摊还代价2，实际代价1，存入存款1
- 之后所有将1翻转成0：摊还代价0，实际代价1，消耗存款1
- 对每个INCREMENT操作而言，支付了摊还代价2

会计方法实例2—二进制计数器

- 初始为零的计数器上n个INCREMENT操作的分析
  - 操作序列上的总代价与0-1或者1-0翻发生的次数成正比
  - 定义0-1翻转的摊还代价为2
  - 定义1-0翻转的摊还代价为0

对于长度为n的INCREMENT操作序列

- 支付的摊还代价的总和：2n
- 这样一个操作序列的复杂度上界：2n
- 因此，摊还代价为：$2n/n=2=O(1)$

提纲

6.1 摊还分析原理

6.2 聚集方法

6.3 会计方法

6.4 势能方法

6.5 动态表操作的摊还分析

势能分析—基本原理

- 会计方法中，如果操作的摊还代价比实际代价大，我们将余额与具体的数据对象关联
- 如果我们将这些余额与整个数据结构关联，所有的这样的余额之和，构成——数据结构的**势能**
- 操作的摊还代价大于操作的实际代价，则**势能增加**
- 操作的摊还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，则**势能减少**

势能分析—基本原理

势能：对一个初始状态为D_0的数据结构执行n个操作，对于操作i：

- 实际代价c_i 将数据结构从状态D_{i-1} 变为D_i
- 势能函数Φ将数据结构的状态D_i映射为实数$\Phi(D_i)$
- 摊还代价c_i'定义为：$c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1})$

势能分析—基本原理

- n个操作的总的摊还代价为：

$$\sum_{i=1}^{n} c_i' = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}))$$

$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

- **于是势函数Φ需满足$\Phi(D_n) \geq \Phi(D_0)$，才能保证总的摊还代价是总的实际代价的上界**
- **在实践中，我们定义$\Phi(D_0)$为0，再证明对所有i有$\Phi(D_i) \geq 0$**
- **摊还代价依赖于所选择的势函数Φ。不同的势函数可能会产生不同的摊还代价，它们都是实际代价的上界**

势能方法实例1—栈操作

- $\Phi(D)$：栈D中对象的个数
- 初始栈为空，$\Phi(D_0)=0$
- 栈中的对象个数始终非负，第i个操作之后的栈D_i满足$\Phi(D_i)\geq 0=\Phi(D_0)$
- 因此， 以Φ表示的n个操作的摊还代价的总和就表示了实际代价的一个上界

势能方法实例1—栈操作

- 作用于包含s个对象的栈上的操作的摊还代价

第i个操作是个PUSH操作

- 实际代价：$c_i=1$
- 势差：$\Phi(D_i)-\Phi(D_{i-1})=(s+1)-s=1$
- 摊还代价：$c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1+1=2$

势能方法实例1—栈操作

- 作用于包含s个对象的栈上的操作的摊还代价

第i个操作是MULTIPOP(S, k)且弹出了k'=min(k,s)个对象

- 实际代价：c_i=k'
- 势差为：$\Phi(D_i)-\Phi(D_{i-1})$=-k'
- 摊还代价：$c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1})$ =k' – k'=0

势能方法实例1—栈操作

- 作用于包含s个对象的栈上的操作的摊还代价

第i个操作是POP

- 实际代价：$c_i=1$
- 势差：$\Phi(D_i)-\Phi(D_{i-1})=-1$
- 摊还代价：$c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1})$ =1–1=0

势能方法实例1—栈操作

- 作用于包含s个对象的栈上的操作的摊还代价

摊还分析

- 每个栈操作的摊还代价都是O(1)
- n个操作序列的总摊还代价就是O(n)
- 因为$\Phi(D_i)\geq 0=\Phi(D_0)$， n个操作的总摊还代价为总的实际代价的上界，即n个操作的最坏情况代价为O(n)

势能方法实例1—二进制计数器

- $\Phi(D)$：计数器D中1的个数
- 计数器初始状态D_0中1的个数为0，$\Phi(D_0)=0$
- 因为数组中的1的个数始终为非负，第i个操作之后的D_i满足$\Phi(D_i)\geq 0=\Phi(D_0)$
- n个操作的摊还代价总和是实际代价的一个上界

势能方法实例1—二进制计数器

- 第i次INCREMENT操作的摊还代价

第i次INCREMENT操作对t_i个位进行了置0, 至多将一位置1

- 该操作的实际代价：$c_i=t_i+1$
- 在第i次操作后计数器中1的个数为：$b_i \le b_{i-1}-t_i+1$
- 势差：$\Phi(D_i)-\Phi(D_{i-1}) \le (b_{i-1}-t_i+1)-b_{i-1}=1-t_i$
- 摊还代价：$c_i' = c_i+\Phi(D_i)-\Phi(D_{i-1}) \le (t_i+1)+(1-t_i)=2$

势能方法实例1—二进制计数器

计数器初始状态为0时的摊还分析

- 每个操作的摊还代价都是O(1)
- n个操作序列的总摊还代价就是O(n)
- 因为$\Phi(D_i) \geq 0 = \Phi(D_0)$，n个操作的总摊还代价即为总的实际代价的一个上界，即n个操作的最坏情况代价为O(n)

势能方法实例1——二进制计数器

初始不为零的摊还分析

- 设开始时有 $b_0 \geq 0$ 个1
- 在n次INCREMENT操作之后有 b_n 个1
- 一系列操作的实际代价为：$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} c_i' - \Phi(D_n) + \Phi(D_0)$
- 因为$\Phi(D_0) = b_0$，$\Phi(D_n) = b_n$，n次INCREMENT操作的总的实际代价为：$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} 2 - b_n + b_0$
- 如果我们执行了至少n=Ω(k)次INCREMENT操作，则无论计数器中包含什么样的初始值，总的实际代价都是O(n)

提纲

6.1 摊还分析原理

6.2 聚集方法

6.3 会计方法

6.4 势能方法

6.5 动态表操作的摊还分析

动态表

- 研究表的动态扩张和收缩问题
- 利用摊还分析证明插入和删除操作的摊还代价为O(1)，即使当它们引起了表的扩张和收缩时具有较大的实际代价
- 研究如何保证动态表中未用的空间始终不超过整个空间的一部分

动态表—基本术语

- 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:用数组来实现动态表
- 非空表T的装载因子$\alpha(T)=$ T存储的对象数/表大小
  - 空表的大小为0，装载因子为1
  - 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比例

动态表—基本术语

设T表示一个表:

- table[T]是一个指向表示表的存储块的指针
- num[T]包含了表中的项数
- size[T]是T的大小
- 初始时，num[T]=size[T]=0

动态表—表的扩张

- 向表中插入一个数组元素时，分配一个包含比原表更多的槽的新表，再将原表中的各项复制到新表中去
- 一种常用的启发式技术是分配一个比原表大1倍的新表，如果只对表执行插入操作，则表的装载因子总是至少为1/2，这样浪费掉的空间就始终不会超过表总空间的一半

动态表—表的扩张

输入：表T，待插入的元素x

输出：插入元素后的表T

算法：TABLE—INSERT(T, x)

```
   If  size[T]=0   Then
       allocate table[T] with 1 slot;
       size[T]←1;
   If  num[T]=size[T] Then
       allocate new table with 2×size[T] slots;
       insert all items in table[T] into new-table;
       free table[T];
       table[T]←new-table;
       size[T]←2×size[T];
Insert x into table[T];
num[T]←num[T]+1
```

动态表—表的扩张

初始为空的表上n次TABLE-INSERT操作的代价分析-**粗略分析**

第i次操作的代价C_i

- 如果i=1： $c_i=1$
- 如果表有空间：$c_i=1$
- 如果表是满的：$c_i=i$
- 如有n次操作，最坏情况下：每次$c_i=i$，总的代价上界为$O(n^2)$

这个界不精确，n次TABLE—INSERT操作并不常有扩张表的代价。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

动态表—表的扩张

初始为空的表上n次TABLE-INSERT操作的代价分析-**聚集分析**

第i次操作的代价C_i

- 如果$i=2^m+1$：$c_i=i$；否则$c_i=1$
- n次TABLE—INSERT操作的总代价为：

$$\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{[log_2(n-1)]} 2^j$$

$$\leq n + 2n = 3n$$

- TABLE-INSERT的摊还代价为:3n/n=3 =O(1)

动态表—表的扩张

初始为空的表上n次TABLE-INSERT操作的代价分析-**会计法分析**

每次执行TABLE—INSERT摊还代价为3

- 1支付插入x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
- 当表扩张时，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的摊还代价和为3n

动态表—表的扩张

初始为空的表上n次TABLE-INSERT操作的代价分析-**会计法分析**

	1
存款	1

第1次插入(注:第一次摊还代价为2,其余为3)

	1	
存款	0	

扩张

	1	2
存款	1	1

第2次插入

	1	2		
存款	0	0		

扩张

	1	2	3	
存款	1	0	1	

第3次插入

动态表—表的扩张

初始为空的表上n次TABLE-INSERT操作的代价分析-**会计法分析**

i	1	2	3	4
存款	1	1	1	1

第4次插入

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			

第5次插入

i	1	2	3	4	5	6		
存款	1	1	0	0	1	1		

第6次插入

依此类推

动态表—表的扩张

初始为空的表上n次TABLE-INSERT操作的代价分析-势能法分析

- 势能函数需要表满发生扩张时势能足以支付扩张的代价

设计思路

- $\Phi(T_0)$=0 初始为空表时;
- 刚扩张的新表T末插入新元素时，势能也为0，即$\Phi(T)$=0;
- 表满时，即将需要扩张，需要消耗的势能较大，令其势能为$\Phi(T) = size(T)$;

$$\Phi(T) = 2 * num[T] - size(T)$$

动态表—表的扩张

初始为空的表上n次TABLE-INSERT操作的代价分析-势能法分析

第i次操作的摊还代价

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

- 如果发生扩张：

$$c_i' = (num[T_{i-1}] + 1)$$
$$+(2 * (num[T_{i-1}] + 1) - 2 * num[T_{i-1}])$$
$$-(2 * num[T_{i-1}] - num[T_{i-1}]) = 3$$

- 否则：

$$c_i' = 1 + 2 * (num[T_{i-1}] + 1) - size(T)$$
$$-(2 * num[T_{i-1}] - size(T)) = 3$$

- 初始为空的表上n次TABLE-INSERT操作的摊还代价总和为3n

$$\Phi(T) = 2 * num[T] - size(T)$$

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
- 非空表T的装载因子$\alpha(T)$= T存储的对象数/表大小
  - 空表的大小为0，装载因子为1
  - 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
- **理想情况下，我们希望动态表满足：**
  - **表具有一定的丰满度**
  - **表的操作序列的复杂度是线性的**

动态表—表的扩张

- 向表中插入一个数组元素时，分配一个包含比原表更多的槽的新表，再将原表中的各项复制到新表中去
- 分配一个比原表大一倍的新表，如果只对表执行插入操作，则表的装载因子总是至少为1/2，这样浪费掉的空间就始终不会超过表总空间的一半

动态表—表的收缩

根据表的扩张策略，很自然地想到表的收缩策略：当表的装载因子小于1/2时，收缩表为原表的一半

是否合适?

动态表—表的扩张和收缩

- n是2的方幂，下面的一个长度为n的操作序列：
  - 前n/2个操作是插入，
  - 之后跟I D D I I D D I I…，I表示插入操作，D表示删除操作

每次扩张和收缩的代价为O(n)，共有O(n)次扩张或收缩总代价为O(n²)，而每一次操作的摊还代价为O(n)，每个操作的摊还代价太高！

动态表—表的扩张和收缩

改进表的收缩策略：

- 当删除元素时，允许装载因子低于1/2
- 当删除一项而引起表不足1/4满时，将表缩小为原来的1/2
- 当向满的表中插入一项时，还是将表扩大一倍
- 扩张和收缩过程都使得表的装载因子变为1/2，但表的装载因子的下界是1/4

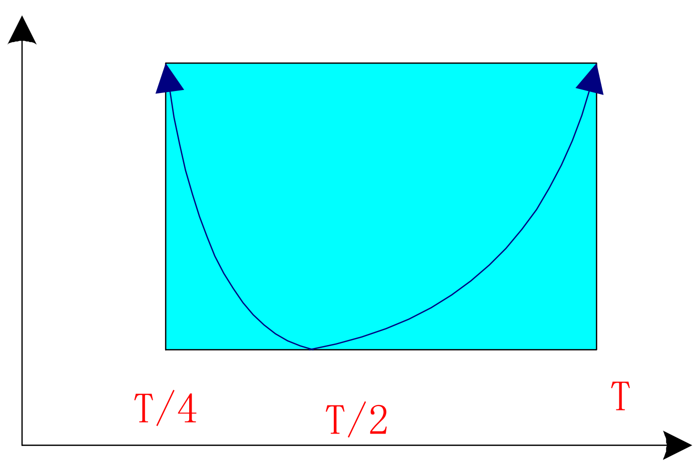
收缩完的时候，得到的状态 类似于 新扩张完的状态

动态表—表的扩张和收缩

- 由n个由TABLE—INSERT和TABLE-DELETE构成的操作序列的分析-**势能法**
- 势函数
  - 操作序列过程中表T的势总是非负的；这样才能保证一列操作的总摊还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

势能的要求：

- 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，势增加直到扩张



动态表—表的扩张和收缩

- 由n个由TABLE—INSERT和TABLE-DELETE构成的操作序列的分析-**势能法**
- 势函数定义

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

空表的势为0，且势总是非负的。这样Φ表示的一列操作的总摊还代价即为其实际代价的一个上界

动态表—表的扩张和收缩

- 由n个由TABLE—INSERT和TABLE-DELETE构成的操作序列的分析-**势能法**
- 势函数的某些性质：

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

  - 当装载因子为1/2时，势为0
  - 当装载因子为1时，有num[T]=size[T]，意味着Φ(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*num[T]，意味着Φ(T)=num[T]。因而当删除某项引起一次收缩时就可用势来支付其代价。

动态表—表的扩张和收缩

- 由n个由TABLE—INSERT和TABLE-DELETE构成的操作序列的分析-**势能法**

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

第i次操作的摊还代价

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

- 第i次操作是TABLE—INSERT：未扩张，$c'_i \leq 3$
- 第i次操作是TABLE—INSERT：扩张，$c'_i \leq 3$
- 第i次操作是TABLE—DELETE：未收缩，$c'_i \leq 3$
- 第i次操作是TABLE—DELETE：收缩，$c'_i \leq 3$
- n个TABLE—INSERT和TABLE-DELETE构成的序列的摊还代价总和上界为3n