算法设计与分析

第七章 摊还分析

Amortized Analysis

哈尔滨工业大学（深圳）
李穆

思考题：

小明去靶场打靶。

靶场的枪只有三种操作：a)装入一颗子弹；b)单发，射出一枚子弹；c)连发，射出三枚子弹。

小明每次只能选择其中的一种。

问：小明最多每次操作可以射出几枚子弹？

ADT GUN{

数据对象：子弹

数据关系：1by1

基本操作：load() , fire1(), fire3()

}

本讲内容

7.1 摊还分析原理

7.2 聚集方法

7.3 会计方法

7.4 势能方法

7.5 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

平摊代价

不涉及概率，
不同于平均情况分析

本讲内容

7.1 摊还分析原理

7.2 聚集方法

7.3 会计方法

7.4 势能方法

7.5 动态表操作的摊还分析

聚集分析法-原理

操作序列中的每个操作被赋予相同的代价，不管操作的类型

对数据结构共有n个操作

最坏情况下：

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x压入栈S；
- POP(S)：弹出并返回S的顶端元素，对空栈返回一个错误；

时间代价:

- 两个操作的时间复杂度都是$O(1)$
- n个PUSH和POP操作系列的总代价是n
- 我们可把每个操作的代价视为1
- n个操作的实际复杂度$\theta(n)$

聚集分析法实例1-栈操作

新的栈操作，MULTIPOP(S, k)**：**

- 去掉栈S的k个顶端对象
- 或当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

时间复杂度与实际执行的POP操作数成线性关系

输入： 栈S，k

输出： 返回S顶端k个对象

MULTIPOP(S, k):

执行一次While循环要调用一次POP，While循环执行的次数是从栈中弹出的对象数min(s, k)

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

MULTIPOP的总代价即为min (s, k)，其中s是栈S中对象个数

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长为n的栈操作序列

操作1:　t_1

操作2:　t_2

:

:

:

操作n:　t_n

最坏情况下，每个操作都是:MULTIPOP，每个MULTIPOP的代价最坏是n，因为栈的大小最大是n。

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

- n是所有操作的次数，不是输入规模！！
- 这样分析太粗糙了！！它这不是一个确界;
- 我们能不能从元素进出栈的情况来分析?

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长为n的栈操作序列
- 一个对象在每次被压入栈后至多被弹出一次
- 调用POP的次数（包括在MULTIPOP内的调用）至多等于PUSH的次数，即至多为$n-1$（比如$n-1$ 次PUSH，1次MULTIPOP ）

操作1:　t_1

操作2:　t_2

⋮

⋮

⋮

操作n:　t_n

$$T(n) \le 2n$$

注意：分析过程没有使用任何的概率！

- 摊还代价$= T(n)/n = O(1)$；
- 最坏情况下这样的一个操作序列的时间复杂度最多为$O(n)$；
- 利用聚集分析，得到了一个更好的上界。

聚集分析法实例2-二进制计数器

1. 问题定义

- 实现一个由 0 开始向上计数的 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：$(x+1) mod\ 2^k$
- 数据结构：

  k位数组$A[0 \cdots k-1]$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

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

聚集分析法实例2-二进制计数器

2. 计数器加1算法

- 输入：$A[0 \cdots k-1]$，存储二进制数x
- 输出：$A[0 \cdots k-1]$，存储二进制数$(x+1) mod\ 2^k$

```
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.
```

聚集分析法实例2-二进制计数器

3.初始为零的计数器上n个INCREMENT操作的分析

该列每8次发生一次改变共$n/8$次

该列每4次发生一次改变共$n/4$次

该列每2次发生一次改变共$n/2$次

该列每次操作发生一次变化共n次

Counter

N	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	total
0	0	0	0	0	0	0	0	0	0
1	[illegible]	[illegible]	[illegible]	[illegible]	0	0	0	1	1
2	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	1	0	3
3	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	1	1	4
4	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	1	0	0	7
5	[illegible]	[illegible]	[illegible]	[illegible]	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

总共发生的改变为：

$\sum \left\lfloor \frac{n}{2^i} \right\rfloor$ （$i = 0, 1, 2, \cdots, \log_2 n) < 2n$，

每个操作的平均代价，

即摊还代价为$O(n)/n = O(1)$

- 每次INCREMENT操作的代价与被改变值的字位的个数成线性关系
- 粗略地讲：每次INCREMENT操作最多改变计数器中k位，n次INCREMENT操作，代价为nk

本讲内容

7.1 摊还分析原理

7.2 聚集方法

7.3 会计方法

7.4 势能方法

7.5 动态表操作的摊还分析

- 小明开了一家公司，决定投资一个项目，每个月固定给这个项目投资一笔钱，这笔钱的数目是前一年公司开会的时候决定的。
- 每个月投资2w，项目的实际开销如下

1月	2月	3月	4月	5月	6月	7月
1w	1.5w	2w	1w	3w	3w	1w

- 这个项目可以顺利开展吗？

会计方法-基本原理

- **一个操作序列中有不同类型的操作，不同类型的操作代价各不相同**
  - 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结构的**具体数据对象**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（摊还代价大于等于实际代价）

会计方法-基本原理

于是：我们在各种操作上定义平摊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

1. 各栈操作的实际代价：

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

2. 各栈操作的摊还代价：

PUSH	2,
POP	0,
MULTIPOP	0,

能不能保证在任何操作序列上，存款的总额非负？

会计方法实例 1—栈操作

- 栈操作序列代价分析

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结构的具体数据对象上
- 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

弹出一个元素支付摊还代价0，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1

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

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
- 于是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$\leq n$

于是：摊还代价的总和 $\leq 2n$，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

- **计数器加1算法**
  - 输入：$A[0 \cdots k-1]$，存储二进制数x
  - 输出：$A[0 \cdots k-1]$，存储二进制数$(x+1) \bmod 2^k$

```
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

显然：这个操作序列的代价与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

1-0翻转的摊还代价为0

0-1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

1-0支付摊还代价0，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1

0-1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

Counter N	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	total
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 +1	0 0	3
3	0	0	0	0	0	0	1 +1	1 +1	4
4	0	0	0	0	0	1 +1	0 0	0 0	7
5	0	0	0	0	0	1 +1	0 0	1 +1	8
6	0	0	0	0	0	1 +1	1 +1	0 0	10
7	0	0	0	0	0	1 +1	1 +1	1 +1	11
8	0	0	0	0	1 +1	0 0	0 0	0 0	15

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

  定义：

  0-1翻转的摊还代价为2

  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，非负

因此，所有的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

定义：

0-1翻转的摊还代价为2

1-0翻转的摊还代价为0

```
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.
```

对每个**INCREMENT**操作：

- 找到右起的第一个**0**，将他翻转成**1—**支付摊还代价**2**
- 将这个**0**之后的所有**1**翻转成**0—**支付摊还代价**0**
- 对这个**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

  定义：

  0-1翻转的摊还代价为2

  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为$2n$
- 因此，这样一个操作序列的复杂度上界为$2n$

本讲内容

7.1 摊还分析原理

7.2 聚集方法

7.3 会计方法

7.4 势能方法

7.5 动态表操作的摊还分析

势能分析—基本原理

- 在会计方法中，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都与**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**
- 如果操作的摊还代价大于操作的实际代价-**势能增加**
- 如果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-**势能减少**

势能分析—基本原理

势能的定义：对一个初始数据结构D_0执行n个操作，对操作i：

- 实际代价c_i将数据结构D_{i-1}变为D_i
- 势能函数ϕ将每个数据结构D_i映射为一个实数$\phi(D_i)$
- $\phi(D_i)$就是关联到数据结构D_i的势能
- 摊还代价c'_i定义为：$c'_i = c_i + \phi(D_i) - \phi(D_{i-1})$
- 势能函数ϕ是非负的

势能分析—基本原理

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，但它们都是实际代价的上界

- n个操作的总的摊还代价为：

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

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

在实践中，我们定义$\phi(D_0)$为0，然后再证明对所有i有$\phi(D_i) \geq 0$

于是势函数ϕ满足$\phi(D_n) \geq \phi(D_0)$，则总的摊还代价就是总的实际代价的一个上界

势能方法实例1 — 栈操作

- $\phi(D)$ =栈D中对象的个数
- 初始空栈D_0，$\phi(D_0) = 0$
- 因为栈中的对象数始终非负，第i个操作之后的栈D_i满足$\phi(D_i) \geq 0 = \phi(D_0)$
- 于是：以势能函数$\phi(D)$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个对象的栈上的栈操作的摊还代价

如果第i个操作是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个对象的栈上的栈操作的摊还代价

如果第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个对象的栈上的栈操作的摊还代价

如果第i个操作是MULTIPOP(S,k)且弹出了$k'=\min(s,k)$个对象

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

这也间接地给出了会计法为什么设置

PUSH	2,
POP	0,
MULTIPOP	0,

势能方法实例1 — 栈操作

- 作用于包含s个对象的栈上的栈操作的摊还代价

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$\phi(D_i) \geq \phi(D_0)$，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 2 — 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$\phi(D_i) \geq 0 = \phi(D_0)$
- 于是：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 2 — 二进制计数器

- 第i次INCREMENT操作的摊还代价

设第i次INCREMENT操作对前t_i个位进行了置0，将置t_i+1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$b_i = b_{i-1} - t_i + 1$
- 势差：$\phi(D_i) - \phi(D_{i-1}) = (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i$
- 摊还代价：$c_i' = c_i + \phi(D_i) - \phi(D_{i-1}) \leq (t_i + 1) + (1 - t_i) = 2$

势能方法实例 2 — 二进制计数器

- 第i次INCREMENT操作的摊还代价

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$\phi(D_i) \geq \phi(D_0)$，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 2 — 二进制计数器

- 开始时不为零的计数器上n个INCREMENT操作的分析
  - 设开始时有b_0个1且$b_0 \geq 0$
  - 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操作的总实际代价为：

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} 2 - b_n + b_0 = 2n - b_n + b_0$$

  - 如果我们执行了至少$n = \omega(k)$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$O(n)$

本讲内容

7.1 摊还分析原理

7.2 聚集方法

7.3 会计方法

7.4 势能方法

7.5 动态表操作的摊还分析

动态表

- 本节的目的：
  - 研究表的动态扩张和收缩的问题
  - 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（表的扩张和收缩不是一直发生的）
  - 研究如何保证一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
- 存储结构:用数组来实现动态表
- 非空表T的装载因子$\alpha(T) = \dfrac{T\text{存储的对象数}}{\text{表大小}}$
  - 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

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
- 装载因子$\alpha(T) = \dfrac{T\text{存储的对象数}}{\text{表大小}} = \dfrac{\text{num}[T]}{\text{size}[T]}$
- 开始时，num[T] = size[T] = 0，$\alpha(T) = 1$

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$\frac{1}{2}$，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

算法：TABLE-INSERT(T, x)

If $size[T] == 0$ Then /*空表*/
    allocate $table[T]$ with 1 slot（槽）；
    $size[T] \leftarrow 1$;
If $num[T] == size[T]$ Then /*满表*/
    allocate new table with $2 \times size[T]$ slots;
    insert all items in table$[T]$ into new-table;
    free table$[T]$;
    $table[T] \leftarrow$new-table;
    $size[T] \leftarrow 2 \times size[T]$;
Insert x into $table[T]$;
$num[T] \leftarrow num[T] + 1$.

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

算法：TABLE-INSERT(T, x)

1. If $size[T] == 0$ Then
2. allocate $table[T]$ with 1 slot;
3. $size[T] \leftarrow 1$;
4. If $num[T] == size[T]$
5. allocate new table with $2 \times size[T]$ slots
6. insert all items in table[T] into new-t
7. free table[T];
8. $table[T] \leftarrow$new-table
9. $size[T] \leftarrow 2 \times size[T]$;
10. Insert x into $table[T]$;
11. $num[T] \leftarrow num[T] + 1$.

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$c_i = i$

如果以共有n次操作，最坏情况下：

每次进行n次操作，总的代价上界为n^2

这个界不精确，因为执行n次TABLE—INSERT操作的过程中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。

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**

```
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
        insert all items in table[
        free table[T];
        table[T]←new-table;
        size[T]←2 × size[T];
Insert x into table[T];
num[T]←num[T] + 1.
```

第i次操作的代价c_i:

如果$i = 2^j + 1$　　$c_i = i$

否则　　　　$c_i = 1$

n次TABLE-INSERT操作的总代价为：

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

每一操作的摊还代价为$3n/n = 3$

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**

算法：TABLE-INSERT(T, x)

1. If $size[T] == 0$ Then
2. allocate $table[T]$ with 1 slot;
3. $size[T] \leftarrow 1$;
4. If $num[T] == size[T]$ Then
5. allocate new table with $2 \times size[T]$ slots;
6. insert all items in $table[T]$ into new-table;
7. free $table[T]$;
8. $table[T] \leftarrow$ new-table;
9. $size[T] \leftarrow 2 \times size[T]$;
10. Insert x into $table[T]$;
11. $num[T] \leftarrow num[T] + 1$.

每次执行TABLE—INSERT摊还代价为2

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

当发生表的扩张时，数据的复制的代价由数据上的存款来支付

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

插入2，表满，扩张

	1	
存款	0	

插入2

	1	2
存款	0	1

插入3，表满，扩张

1上的存款不够，怎么办

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，数据的复制的代价由数据上的存款来支付

算法：TABLE-INSERT(T, x)

1. If $size[T] == 0$ Then
2. allocate $table[T]$ with
3. $size[T] \leftarrow 1$；
4. If $num[T] == size[T]$ Then
5. allocate new table with $2 \times size[T]$ slots；
6. insert all items in table$[T]$ into new-table；
7. free table$[T]$；
8. $table[T] \leftarrow$new-table；
9. $size[T] \leftarrow 2 \times size[T]$；
10. Insert x into $table[T]$；
11. $num[T] \leftarrow num[T] + 1$.

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-**会计法分析**

插入1（注:第一次摊还代价为2，其余为3）

	1
存款	1

插入2，表满扩张

	1	
存款	0	

插入2

	1	2
存款	1	1

插入3，表满扩张

	1	2		
存款	0	0		

插入3

	1	2	3	
存款	1	0	1	

每次执行TABLE—INSERT摊还代价为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，数据的复制的代价由数据上的存款来支付**

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

插入4

i	1	2	3	4				
存款	0	0	0	0				

插入5，表满，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

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-**会计法分析**

算法：TABLE-INSERT(

1. If $size[T] == 0$ Then
2. allocate $table[T]$ with
3. $size[T] \leftarrow 1$；
4. If $num[T] == size[T]$ Then
5. allocate new table with $2 \times size[T]$ slots;
6. insert all items in $table[T]$ into new-table；
7. free $table[T]$;
8. $table[T] \leftarrow$new-table；
9. $size[T] \leftarrow 2 \times size[T]$；
10. Insert x into $table[T]$；
11. $num[T] \leftarrow num[T] + 1$.

任何时候，存款总和非负

每次执行TABLE—INSERT摊还代价为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，数据的复制的代价由数据上的存款来支付

初始为空的表上n次TABLE-INSERT操作的摊还代价总和为$3n$

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-**势能法分析**

- 势能函数怎么定义，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（还未插入），$\phi(T) = 0$；
  - 表满时$\phi(T) = size(T)$，是否可行？

势能函数可行并且：
- $num[T] \geq size[T]/2$，$\phi(T) \geq 0$
- n 次 TABLE-INSERT 操作的总的摊还代价就是总的实际代价的一个上界

算法：TABLE-INSERT(T, x)

1. If $size[T] == 0$ Then
2. allocate $table[T]$ with 1 slot；
3. $size[T] \leftarrow 1$；
4. If $num[T] == size[T]$ Then
5. allocate new table with $2 \times size[T]$ slots；
6. insert all items in table$[T]$ into new-table；
7. free table$[T]$；
8. $table[T]$ ←new-table；
9. $size[T] \leftarrow 2 \times size[T]$；
10. Insert x into $table[T]$；
11. $num[T] \leftarrow num[T] + 1$.

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-势能法分析

```
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.
```

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

第i次操作的摊还代价：

$c_i' = c_i + \phi(T_i) - \phi(T_{i-1})$

如果发生扩张： $c'_i = 3$

发生扩张： $c_i = num_i$

$size_i = 2 * size_{i-1}; num_{i-1} = size_{i-1} = num_i - 1 \Rightarrow size_i = 2 * (num_i - 1)$

势能

$$
\begin{aligned}
c_i' &= c_i + \phi(T_i) - \phi(T_{i-1}) = num_i + (2 * num_i - size_i) - (2 * num_{i-1} - size_{i-1}) \\
&= num_i + (2 * num_i - 2 * (num_i - 1)) - (2(num_i - 1) - (num_i - 1)) \\
&= num_i + 2 - num_i + 1 \\
&= 3
\end{aligned}
$$

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**

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.
```

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

第i次操作的摊还代价：

$c_i' = c_i + \phi(T_i) - \phi(T_{i-1})$

如果发生扩张： $c'_i = 3$

否则 $c'_i = 3$

初始为空的表上n次TABLE-INSERT操作的摊还代价总和为$3n$

不扩张： $c_i = 1$

$size_i = size_{i-1}; num_{i-1} = num_i - 1$

势能

$$
\begin{aligned}
c_i' = \mathbf{c_i} + \phi(T_i) - \phi(T_{i-1}) &= \mathbf{1} + (2 * num_i - size_i) - (2 * num_{i-1} - size_{i-1}) \\
&= 1 + (2 * num_i - size_i) - (2(num_i - 1) - size_i) \\
&= 1 + 2 \\
&= 3
\end{aligned}
$$

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$\alpha(T) = \dfrac{T\text{存储的对象数}}{\text{表大小}}$
  - 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

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$O(n^2)$，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
- 但当删除一项而引起表的装载因子不足1/4满时，我们就将表缩小为原来的一半
- 当向满的表中插入一项时，还是将表扩大一倍
- 这样扩张和收缩过程都使得表的装载因子变为1/2，但是表的装载因子的下界是1/4

动态表—表的收缩

算法：TABLE-DELETE(T, x)

1. If $num[T] == size[T] / 4$ Then
2. allocate new table with $size[T]/2$ slots;
3. copy all items except for x from table[T] into new-table;
4. free table[T];
5. $table[T] \leftarrow$ new-table;
6. $size[T] \leftarrow size[T]/2$;
7. Else Delete x from $table[T]$;
8. $num[T] \leftarrow num[T] - 1$.

没有收缩，代价1，收缩代价， $num[T]$

动态表-表的扩张和收缩

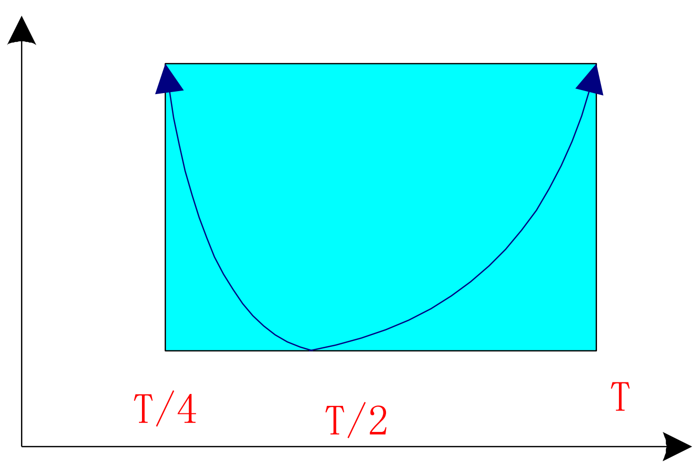
- 由n个TABLE-INSERT和TABLE-DELETE操作构成序列的代价分析-势能法
- 势能函数
  - 操作序列过程中表T的势总是非负的；这样才能保证一列操作的总摊还代价即为其实际代价的一个上界
  - 表的扩张和收缩过程要消耗大量的势

势能的要求：

1、$num(T) = size(T)/2$时，势最小

2、当$num(T)$减小时，势增加直到收缩

3、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TABLE-INSERT和TABLE-DELETE操作构成序列的代价分析-势能法
- 势函数定义

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

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TABLE-INSERT和TABLE-DELETE操作构成序列的代价分析-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$num[T]=size[T]$，这就意味着$\boldsymbol{\phi}(T)=num[T]$。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，$size[T]=4*num[T]$，它意味着$\boldsymbol{\phi}(T)=num[T]$。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TABLE-INSERT和TABLE-DELETE操作构成序列的代价分析-势能法
- 势函数定义

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

第i次操作的摊还代价：$c_i' = c_i + \phi(T_i) - \phi(T_{i-1})$

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 $c'_i \leq 3$

所以作用于一动态表上的n个操作的实际时间为$O(n)$

《算法导论》P269