数据结构与算法

第三章 平摊（摊还）分析

裴文杰

计算机科学与技术学院 教授

本讲内容

7.1 平摊分析原理

7.2 聚集方法

7.3 会计方法

7.4 势能方法

7.5 动态表操作的平摊分析

《算法导论》第17章

基本思想

- 在平摊(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，不关注局部某一步的代价，而是关注一系列操作的整体代价
- 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
- 将总的代价平摊到每个操作上，就是平摊代价
  - 不涉及概率
- 平摊分析的方法
  - 1）聚集方法； 2）会计方法；3）势能方法

关键：通过分析具体任务中隐含的先验知识，从而得到更准确的分析

本讲内容

7.1 平摊分析原理

7.2 聚集方法（聚合分析）

7.3 会计方法

7.4 势能方法

7.5 动态表操作的平摊分析

聚集分析法-原理

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

操作序列中的每个操作被赋予相同的代价，即使操作的类型不同

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

平摊代价：T(n)/n

关注一个包含n个操作的序列的总代价的上界

聚集分析实例1-栈操作

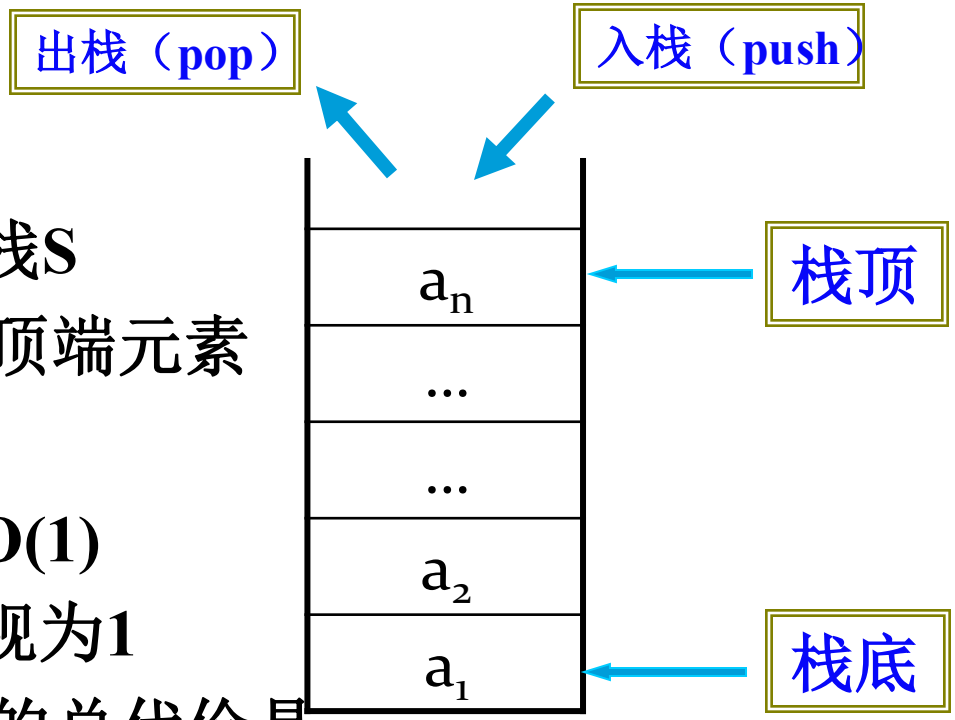
普通栈操作

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
- 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θ(n)



聚集分析实例1-栈操作

- 新的栈操作
  - 操作MULTIPOP(S,k):
    - 去掉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

实际运行时间与实际执行的POP操作数成线性关系

实现算法

输入：栈S，k

输出：返回S顶端k个对象

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

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)

MULTIPOP的总代价即为min(s,k)

聚集分析实例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，T(n)=?

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

$T(n)=n^2$

上面的分析太粗糙了！！！
我们从元素进出栈的情况来分析

聚集分析：综合考虑整个序列的n个操作，得到更好的上界

隐藏先验知识：一个对象在每次被压入栈后至多被弹出一次

关键：提取具体任务的隐藏信息。本例的隐藏信息：一个对象在每次被压入栈后至多被弹出一次

所以在一个非空栈上调用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

$T(n)<=2n$
$T(n)/n=O(1)$

聚集分析实例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

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

聚集分析实例2-二进制计数器

2. 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$x+1 \mod 2^k$

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

算法思路：
1)如果找到从低位起的第一个0，将他翻转成1;
2)将这个0之前与其相邻的所有1翻转成0。

聚集分析实例2-二进制计数器

3. 问题：初始为零的计数器上n个INCREMENT操作的代价分析，即时间复杂性

Counter									总代价
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

聚集分析实例2-二进制计数器

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

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

上面的分析太粗糙了！！！
能不能得到更紧的界呢？

A[0]列每次操作发生一次变化共n次

A[1]列每2次操作发生一次变化共$\lfloor n/2 \rfloor$次

A[2]列每4次操作发生一次变化共$\lfloor n/4 \rfloor$次

A[3]列每8次操作发生一次变化共$\lfloor n/8 \rfloor$次

总共发生的改变为：
$\Sigma \lfloor n/2^i \rfloor$ $(i=0,1,\ldots,k-1) < 2n$
即执行n个INCREMENT操作的总代价为O(n)，每个操作的平均代价为O(1)

关键：提取具体任务的隐藏信息。
本例的隐藏信息：每一位二进制的变化频率。

聚集分析

- 上述两个实例的聚集分析中，都没有用到概率分析，而是根据具体问题的特点进行相应的全局分析。

- 聚集分析只关注整个序列操作的总代价上界，而不关注每个操作的具体代价，将平均代价作为每个操作的平摊代价，因此所有的操作具有相同的平摊代价，即使操作类型不同。

- 通过分析具体任务的隐藏信息（先验知识），来确定更紧的代价上界。

本讲内容

7.1 平摊分析原理

7.2 聚集方法

7.3 会计方法（Accounting method）

7.4 势能方法

7.5 动态表操作的平摊分析

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
- 于是我们为每种操作分配不同的平摊代价
- 会计方法关键：平摊代价可能比实际代价大，也可能小
  - 执行时，如果平摊代价比实际代价高，平摊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
  - 如果平摊代价比实际代价低，平摊代价及数据对象上的存款用来支付实际代价
  - 只要我们能保证：在任何操作序列上，存款的总额非负，则所有操作平摊代价的总和即是实际代价总和的上界

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(k,s)

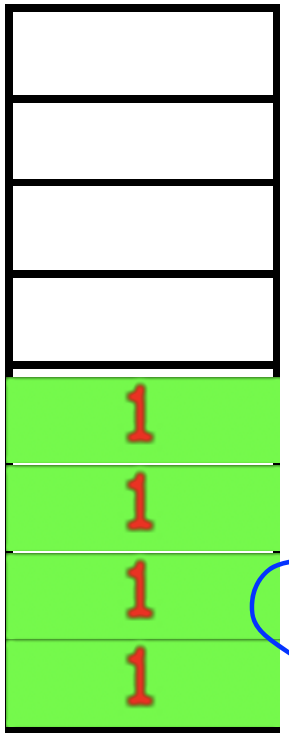
2. 为各栈操作赋予如下平摊代价：

PUSH	2,
POP	0,
MULTIPOP	0,

如何得到这个平摊代价分配方案？
关键：提取具体任务的隐藏信息。
本例的隐藏信息：一个对象在每次被压入栈后至多被弹出一次。
因此，给push操作两个代价，一个用于支付实际代价，一个作为存款用于支付可能的pop操作。

会计方法实例1-栈操作

3. 栈操作序列代价分析



每次插入（PUSH）一个元素，支付平摊代价2：1用于支付实际代价，1用于存款附加在数据对象上

每次弹出(POP)一个元素，附加在数据对象上的存款1和平摊代价0一起用于支付实际代价

只要我们的操作序列是合理的，则可以保证存款总和非负，因为存款总额总是等于栈中的对象个数。于是所有操作的平摊代价总和就是操作序列代价总和的上界

长度为n的操作序列中，PUSH操作的个数<=n，所以，平摊代价的总和<=2n，即平摊代价总和=O(n)

会计方法实例1-二进制计数器

1. 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 mod 2^k

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

会计方法实例1-二进制计数器

- 初始为零的计数器上n个INCREMENT操作的代价分析
  - 显然，这个操作序列的代价与0-1或者1-0翻转发生的次数成正比
  - 定义：0-1翻转的平摊代价为2；1-0翻转的平摊代价为0

如何得到这个平摊代价分配方案？
关键：提取具体任务的隐藏信息。
本例的隐藏信息：1翻转为0之前，先要从0翻转为1，类似于栈操作的push和pop操作。

*当进行0-1翻转时，用平摊代价1支付实际代价，将剩余代价1作为存款附加在值为1的位上。
*当进行1-0翻转时，用存款1和平摊代价0支付实际代价，

任何操作序列，存款余额是计数器中1的个数，一定为非负。
因此，所有的翻转操作的平摊代价的和是这个操作序列代价的上界。

对每个INCREMENT操作：
找到从低位起的第一个0，将他翻转成1—支付平摊代价2；
将这个0之前的所有1翻转成0—支付平摊代价0，实际代价通过附加存款支付；
对这个INCREMENT操作而言，支付了平摊代价2。

对于长度为n的INCREMENT操作序列：
支付的平摊代价的总和为2n
因此，这样一个操作序列的复杂度上界为2n

会计方法

- 会计方法的要点是定义每个操作的平摊代价， 需要满足以下条件：
  - 总的平摊代价总和大于等于实际代价总和，因此要求存款总额都是非负。

同样需要挖掘任务背后的隐藏信息（先验知识），从而合理分配每个操作的平摊代价。

本讲内容

7.1 平摊分析原理

7.2聚集方法

7.3 会计方法

7.4 势能方法

7.5 动态表操作的平摊分析

势能方法-基本原理

- 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

势能方法-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是非递减的

势能方法-基本原理

- n个操作的总的平摊代价为

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

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

在实践中，我们定义$\phi(D_0)$为0，然后要求对所有i有$\phi(D_i) \geq 0$

于是如果定义一个势函数ϕ，满足$\phi(D_n) \geq \phi(D_0)$，则总的平摊代价就是总的实际代价的一个上界

平摊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平摊代价，但它们都是实际代价的上界

势能方法实例1-栈操作

- 定义势函数：**$\phi(D)$=栈D中对象的个数**
- 初始栈D_0为，$\phi(D_0)=0$
- 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：基于势函数ϕ的n个操作的平摊代价的总和就表示了实际代价的一个上界

如何得到合理的势函数定义？
关键：提取具体任务的隐藏信息。
本例的隐藏信息：一个对象在每次被压入栈后至多被弹出一次。
因此总的存款总额，即总的势能，等于栈中对象的个数，与会计法保持一致。

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平摊代价

如果第i个操作是个PUSH操作
·实际代价：$c_i=1$
·势差：$\phi(D_i)-\phi(D_{i-1})=(s+1)-s=1$，
·平摊代价：$c'_i=c_i+\phi(D_i)-\phi(D_{i-1})=1+1=2$

如果第i个操作是POP
·实际代价：$c_i=1$
·势差：$\phi(D_i)-\phi(D_{i-1})=-1$
·平摊代价：$c'_i=c_i+\phi(D_i)-\phi(D_{i-1})=1-1=0$

如果第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$\phi(D_i)-\phi(D_{i-1})=-k'$
·平摊代价：$c'_i=c_i+\phi(D_i)-\phi(D_{i-1})=k'-k'=0$

平摊分析：
- 每个栈操作的平摊代价都是O(1)
- n个操作序列的总平摊代价是O(n)（<=2n）
- 因为$\phi(D_i)\geq\phi(D_0)$，n个操作的总平摊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)

因为用了和会计法相同的隐藏信息：一个对象在每次被压入栈后至多被弹出一次。且会计法中的存款总额也等于栈中的对象个数，与势能定义一致，因此得到的每个操作的平摊代价是一致的。

势能方法实例2-二进制计数器

- 定义势函数：**ϕ(D)=计数器中1的个数**
- 计数器初始状态D_0中1的个数为0，$\phi(D_0)=0$
- 因为计数器中1的个数始终非负，第i个操作之后的栈D_i满足$\phi(D_i)\geq 0=\phi(D_0)$
- 于是：n个操作的平摊代价的总和就表示了实际代价的一个上界

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

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

势能方法实例2-二进制计数器

- 第i次INCREMENT操作的平摊代价（计数器初始为0）

设第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：$c_i \le t_i+1$

·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：$\phi(D_i)-\phi(D_{i-1}) \le (b_{i-1}- t_i+1)-b_{i-1}=1- t_i$,

·平摊代价：$c'_i = c_i+\phi(D_i)-\phi(D_{i-1}) \le (t_i+1)+(1- t_i)=2$

如果b_{i-1}中全部k位为1，那么b_i将会将全部位都置0，此时没有置1操作

计数器初始状态为0时的平摊分析 ：

- 每个操作的平摊代价都是O(1)
- n个操作序列的总平摊代价就是O(n)
- 因为$\phi(D_i) \ge \phi(D_0)$， n个操作的总平摊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)

因为用了和会计法相同的隐藏信息：1翻转为0之前，需要先从0翻转为1. 且会计法中的存款总额也等于计数器中1的个数，与势能定义一致，因此得到的increment操作的平摊代价是一致的。

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的操作的分析
  - 设：开始时有b_0个1，其中$0 \le b_0 \le k$（k为计数器二进制位数）
  - 在n次INCREMENT操作之后有b_n个1，$0 \le b_n \le k$

将公式重写为：$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} c'_i - \Phi(D_n) + \Phi(D_0)$

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：

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

正是势能法，给我们这样的分析带来了方便！

如果我们执行了至少$n=\Omega(k)$(即$k=O(n)$)次INCREMENT操作，因为$b_n <= k$, $b_0 <= k$，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)

势能方法

- 势能方法的要点是定义势能函数，需要满足以下条件：
  - 对于每个操作过后，势函数值总为非负，即对所有i有 $\phi(D_i) \geq 0$

- 根据定义的势函数，计算每个操作的平摊代价，然后相加求和，得到总的平摊代价，作为实际代价的上界

势能方法

- 势能方法与会计方法的区别
  - 会计方法将多余的平摊代价以存款的形式与具体的数据对象相关联，而势能方法将多余的平摊代价以势能的形式与整个数据结构相关联
  - 会计方法预先定义每种操作的平摊代价，然后推断存款，并要保证存款总额总是非负的，而势能方法预先定义势函数，然后反推每种操作的平摊代价，并要保证势能总是非负的
  - 两种方法采用的隐藏信息（先验知识）可能是一样的，如果一样，那么势能方法的势能值和会计方法的存款总额是一致的

本讲内容

7.1 平摊分析原理

7.2聚集方法

7.3 会计方法

7.4 势能方法

7.5 动态表操作的平摊分析

动态表

- 动态表的概念
  - 对于某些应用程序，我们可能事先无法预料它所需的存储空间。我们为一个表分配一定的存储空间，随后如果空间不够用，需要对表进行扩张——分配一个包含更多空间的新表；如果从表中删除了很多对象，可能需要对表进行收缩——重新分配一个更小的内存空间（表）。

- 本节的目的
  - 研究表的动态扩张和收缩的问题
  - 利用平摊分析证明插入和删除操作的平摊代价为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
  - 如何保证动态表中的空闲空间相对于总空间的比例永远不超过一个常量分数。

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:用一个（一组）数组来实现动态表
- 非空表T的装载因子α(T)= T存储的对象数/表大小
  - 空表的大小为0，装载因子定义为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

设T表示一个表:
table[T]是一个指向表的存储块的指针
num[T]表中存放的数据项数（对象数）
size[T]是T的大小
开始时，num[T]=size[T]=0 (初始表为空)

动态表-表的扩张

- 向表中插入一个数组元素时，如果此时表满，需要分配一个包含比原表更多的槽的新表，因为我们总是需要表位于连续的内存空间中，因此必须为更大的新表分配一个新的内存空间，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

```
算法：TABLE—INSERT(T, x)
1       If  size[T]=0 Then
2               allocate table[T] with 1 slot;
3               size[T]←1;
4       If  num[T]=size[T] Then
5               allocate new table with 2×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]←2×size[T];
10      Insert x into table[T];
11      num[T]←num[T]+1
```

不考虑其他操作的开销

开销为常数

算法从空表开始执行，第i次操作如果当前表满，需要扩张，那么复制代价是i-1

基本插入操作，代价为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]←1;
4       If  num[T]=size[T] Then
5               allocate new table with 2×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]←2×size[T];
10      Insert x into table[T];
11      num[T]←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　　$c_i=i$

如果一共有n次操作：

最坏情况下

每次操作代价为O(n)

总的代价上界为$O(n^2)$

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

表的大小一定是2的整数倍

```
算法：TABLE—INSERT(T, x)
1       If  size[T]=0 Then
2               allocate table[T] with 1 slot;
3               size[T]←1;
4       If  num[T]=size[T] Then
5               allocate new table with 2×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]←2×size[T];
10      Insert x into table[T];
11      num[T]←num[T]+1
```

第i次操作的代价C_i:

如果$i-1=2^m$（2的整数幂）$c_i=i$

否则 $c_i=1$

n次TABLE—INSERT操作的总代价为：

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

每一操作的平摊代价为 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]←1;
4       If  num[T]=size[T] Then
5               allocate new table with 2×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]←2×size[T];
10      Insert x into table[T];
11      num[T]←num[T]+1
```

任何时候，存款总和非负

每次执行TABLE—INSERT平摊代价为3（第一步是2）：

1支付这一步中的基本插入操作的实际代价

1作为自身的存款　自身复制需要

1存入表中第一个没有存款的数据上　扩张后空数据需要

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操作的平摊代价总和为3n

动态表-表的扩张

	1
存款	1

第一步操作平摊代价是2,1个用于插入操作，一个用以存款

	1	
存款	0	

扩张

消耗1个存款用于支付扩张时的数据复制

	1	2
存款	1	1

一个代价用于实际代价（插入操作）；
一个代价用于自身存款；
一个代价作为存款存入没有存款的数据项（地址1）

	1	2		
存款	0	0		

扩张

消耗2个存款用于支付扩张时的数据复制

	1	2	3	
存款	1	0	1	

插入3

一个代价用于实际代价（插入操作）；
一个代价用于自身存款；
一个代价作为存款存入没有存款的数据项（地址1）

动态表-表的

一个代价用于实际代价（插入操作）；
一个代价用于自身存款；
一个代价作为存款存入没有存款的数据项（地址2）

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

消耗4个存款用于支付扩张时的数据复制

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

。。。以此类推

为什么存款总是足够（且刚好）支付复制的开销？
每次扩张，表变为原来的两倍，插入新的数据带来的额外存款，正好可以给旧的数据存款。

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)
1       If  size[T]=0 Then
2               allocate table[T] with 1 slot;
3               size[T]←1;
4       If  num[T]=size[T] Then
5               allocate new table with 2×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]←2×size[T];
10      Insert x into table[T];
11      num[T]←num[T]+1
```

势怎么定义？
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：
1） 刚扩充完，$\phi(T)=0$
2） 表满时 $\phi(T)=size(T)$

定义如下势能函数：$\phi(T)=2*num[T]-size[T]$

势能函数满足要求并且：
由于$num[T]\geq size[T]/2$，$\phi(T)\geq 0$
n次TABLE—INSERT操作的总的平摊代价就是总的实际代价的一个上界

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)
1       If  size[T]=0 Then
2               allocate table[T] with 1 slot;
3               size[T]←1;
4       If  num[T]=size[T] Then
5               allocate new table with 2×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]←2×size[T];
10      Insert x into table[T];
11      num[T]←num[T]+1
```

Next：分析每一步的平摊代价：

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

任意一步操作，都是从状态D_{i-1}变为D_i,有两种情况：
1）如果发生扩张（扩张前表是满的），
$\phi(D_{i-1})=size[T]$,
$\phi(D_i)=2*(size[T]+1)-2size[T]=2$
$c_i=size(T)+1$
故 $c'_i=c_i+\phi(D_i)-\phi(D_{i-1}) = 3$

2) 如果未发生扩张（扩张前表未满），
$\phi(D_{i-1})=2*num[T]-size[T]$
$\phi(D_i)=2*(num[T]+1)-size[T]$
$c_i= 1$（实际代价c_i只是在表中插入一个元素）
故 $c'_i= c_i+\phi(D_i)-\phi(D_{i-1}) = 3$

综上，每次操作后平摊代价都是3，故n次操作的平摊总代价为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

```
算法：TABLE—INSERT(T, x)
1       If  size[T]=0 Then
2               allocate table[T] with 1 slot;
3               size[T]←1;
4       If  num[T]=size[T] Then
5               allocate new table with 2×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]←2×size[T];
10      Insert x into table[T];
11      num[T]←num[T]+1
```

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

第i次操作的平摊代价：

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操作的平摊代价总和为3n

以下内容为自学内容，不考

动态表-表的扩张和收缩

- 同时包含表的扩张和收缩操作
  - 收缩操作：当表中数据太少时，分配一个新的更小的表，然后把数据项从旧表复制到新表。

理想情况下，我们希望表满足：

1）表具有一定的丰满度：装载因子大于一个常数下界。
比如num[T]≥size[T]/2 （现有元素个数要大于等于表长的一半）

2）表的操作序列的复杂度（总平摊代价）是线性的。
比如n次包含扩张操作的平摊总代价为O(n)， 而平摊代价是时间复杂度的上界。

动态表-表的扩张和收缩

- 表的收缩策略

根据表的扩张策略，很自然地想到下列的收缩策略：

当表的装载因子小于1/2时，收缩表为原表的一半。

假设n是2的幂，考虑下面的一个长度为n的操作序列（I: Insert, D:Delete)：
前n/2个操作是插入，此时表满，后跟I D D I I D D I I D D...
执行第一个插入操作I导致表规模由n/2扩张至n，接下来两个删除操作D导致表的装载因子小于1/2，表会收缩至n/2,然后执行两个插入操作(I I)引起另一次扩张，再执行两次删除(D D)表会收缩…

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$O(n^2)$

每个操作的平摊代价太高

动态表-表的扩张和收缩

- 表的收缩策略

上面的收缩策略可以改善，允许装载因子低于1/2。

方法：当向满的表中插入一项时，还是将表扩大一倍

但当删除一项而引起表不足1/4满时，我们就将表缩小为原来的一半

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-表的扩张和收缩

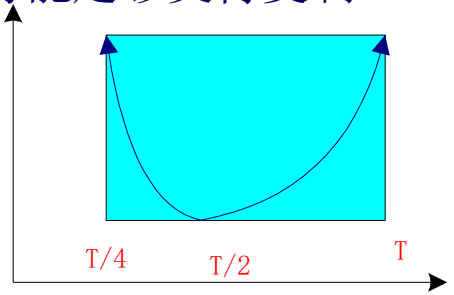
- 由n个TABLE-INSERT和TABLE-DELETE操作构成的序列的代价的分析-势能法
  - 势函数的定义
    - 势总是非负，从而保证平摊代价总和是实际代价的上界
    - 表的收缩和扩张需要消耗大量的势，需要满足：

1） num(T)=size(T)/2时，势最小

2） 当num(T)减小时，势增加直到收缩，使得收缩时，势能足够支付复制

3） 当num(T)增加时，势增加直到扩充，使得扩充时，势能足够支付复制

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

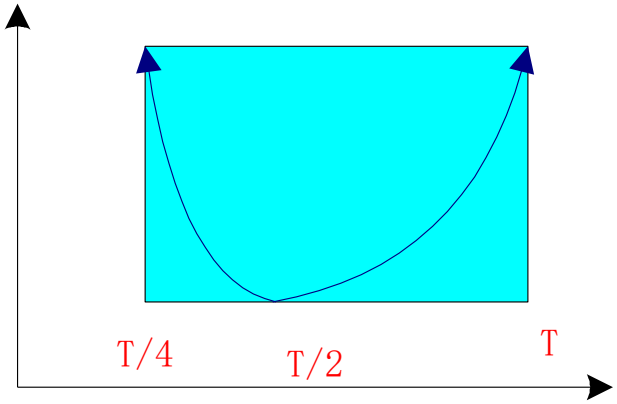


空表的势为0，且势总是非负的。这样，以ϕ表示的一列操作的总平摊代价即为其实际代价的一个上界

动态表-表的扩张和收缩

- 由n个TABLE-INSERT和TABLE-DELETE操作构成的序列的代价的分析-势能法
  - 势函数的定义

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



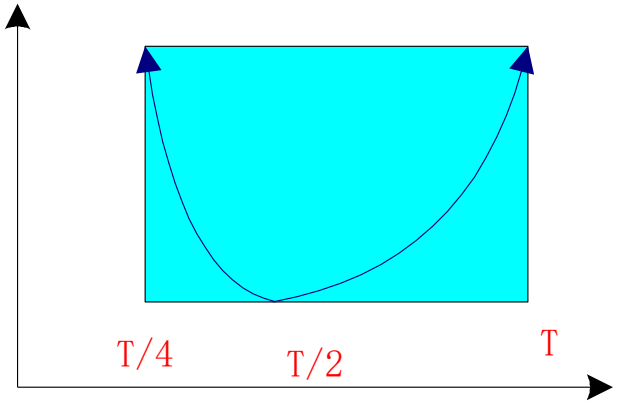
势函数的某些性质：
- 当装载因子为1/2时，势为0。
- 当装载因子为1时，有num[T]=size[T]，这就意味着ϕ(T)=num[T]。这样当因插入一项而引起一次扩张时，就可用势来支付其代价。
- 当装载因子为1/4时，size[T]=4·num[T]。它意味着ϕ(T)=num[T]。因而当删除某项引起一次收缩时就可用势来支付其代价。

扩张和收缩，复制的代价都是num[T]，势能都刚好足够支付复制代价

动态表-表的扩张和收缩

- 由n个TABLE-INSERT和TABLE-DELETE操作构成的序列的代价的分析-势能法
  - 势函数的定义

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



第i次操作的平摊代价：$c'_i=c_i+\phi(T_i)-\phi(T_{i-1})$	
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)

具体计算过程仿照ppt 42页，也可参照书17.4.2节（page 269）