第 17 章

Introduction to Algorithms, Third Edition

摊还分析

在**摊还分析**(amortized analysis)中，我们求数据结构的一个操作序列中所执行的所有操作的平均时间，来评价操作的代价。这样，我们就可以说明一个操作的平均代价是很低的，即使序列中某个单一操作的代价很高。摊还分析不同于平均情况分析，它并不涉及概率，它可以保证最坏情况下每个操作的平均性能。

本章前三节介绍了摊还分析中最常用的三种技术。17.1 节介绍聚合分析(aggregate analysis)，这种方法用来确定一个 n 个操作的序列的总代价的上界 $T(n)$。因而每个操作的平均代价为 $T(n)/n$。我们将平均代价作为每个操作的摊还代价，因此所有操作具有相同的摊还代价。

17.2 节介绍核算法(accounting method)，用来分析每个操作的摊还代价。当存在不止一种操作时，每种操作的摊还代价可能是不同的。核算法将序列中某些较早的操作的“余额”(overcharge)作为“预付信用”(prepaid credit)储存起来，与数据结构中的特定对象相关联。在操作序列中随后的部分，储存的信用即可用来为那些缴费少于实际代价的操作支付差额。

17.3 节讨论势能法(potential method)，与核算法类似，势能法也是分析每个操作的摊还代价，而且也是通过较早操作的余额来补偿稍后操作的差额。势能法将信用作为数据结构的“势能”储存起来，与核算法不同，它将势能作为一个整体储存，而不是将信用与数据结构中单个对象关联分开储存。

我们将使用两个例子来介绍这三种方法。一个例子是带有额外 MULTIPOP 操作的栈，该操作一次性从栈中弹出多个对象。另一个例子是二进制计数器，它从 0 开始计数，通过 INCREMENT 操作实现计数。

当学习本章时，要记住在摊还分析中赋予对象的费用仅仅是用来分析而已，不需要也不应该出现在程序中。例如，在利用核算法进行分析时，如果我们将一定的信用赋予对象 x，那么并不需要在程序中将相应的值赋予对象的某个属性，如 $x.credit$。

通过做摊还分析，通常可以获得对某种特定数据结构的认识，这种认识有助于优化设计。例如，在 17.4 节中，我们将用势能法分析一个动态扩充和收缩的表。

17.1 聚合分析

利用**聚合分析**，我们证明对所有 n，一个 n 个操作的序列最坏情况下花费的总时间为 $T(n)$。因此，在最坏情况下，每个操作的平均代价，或**摊还代价**为 $T(n)/n$。注意，此摊还代价是适用于每个操作的，即使序列中有多种类型的操作也是如此。本章中，我们将要学习的另外两种方法——核算法和势能法，对不同类型的操作可能赋予不同的摊还代价。

栈操作

第一个聚合分析的例子是分析扩充了新操作的栈。10.1 节提出了两种基本的栈操作，时间复杂性均为 $O(1)$：

PUSH(S，x)：将对象 x 压入栈 S 中。

POP(S)：将栈 S 的栈顶对象弹出，并返回该对象。对空栈调用 POP 会产生一个错误。

由于两个操作都是 $O(1)$时间的，我们假定其代价均为 1。因此一个 n 个 PUSH 和 POP 操作的序列的总代价为 n，而 n 个操作的实际运行时间为 $\Theta(n)$。

我们现在增加一个新的栈操作 MULTIPOP(S，k)，它删除栈 S 栈顶的 k 个对象，如果栈中对象数少于 k，则将整个栈的内容都弹出。当然，我们假定 k 是正整数，否则 MULTIPOP 会保持栈不变。在下面的伪代码中，STACK-EMPTY 在当前栈中没有任何对象时返回 TRUE，否则

返回 FALSE。

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

图 17-1 给出了 MULTIPOP 的一个例子。

在一个包含 s 个对象的栈上执行 MULTIPOP(S, k)操作的运行时间应该是怎样的呢？真正的运行时间是与实际执行的 POP 操作的次数呈线性关系的，因此，我们可以用 PUSH 和 POP 操作的抽象代价 1 来分析描述 MULTIPOP 的代价。**while** 循环执行的次数等于从栈中弹出的对象数，等于 $\min(s, k)$。每个循环步调用一次 POP(第 2 行)。因此，MULTIPOP 的总代价为 $\min(s, k)$，而真正的运行时间为此代价的线性函数。

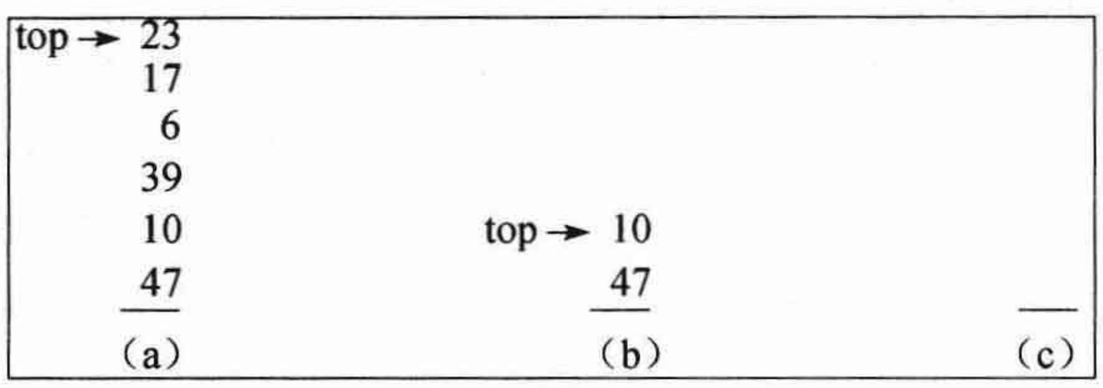

图 17-1　对栈 S 进行 MULTIPOP 操作，栈的初始格局如(a)所示。通过 MULTIPOP(S, 4)弹出栈顶 4 个对象，结果如(b)所示。因为栈中剩下的对象不足 7 个，下一个操作 MULTIPOP(S, 7)将栈清空，如(c)所示

我们来分析一下一个由 n 个 PUSH、POP 和 MULTIPOP 组成的操作序列在一个空栈上的执行情况。序列中一个 MULTIPOP 操作的最坏情况代价为 $O(n)$，因为栈的大小最大为 n。因此，任意一个栈操作的最坏情况时间为 $O(n)$，从而一个 n 个操作的序列的最坏情况代价为 $O(n^2)$，因为序列可能包含 $O(n)$个 MULTIPOP 操作，每个的执行代价为 $O(n)$。虽然这个分析是正确的，但我们通过单独分析每个操作的最坏情况代价得到的操作序列的最坏情况时间 $O(n^2)$，并不是一个确界。

通过使用聚合分析，我们考虑整个序列的 n 个操作，可以得到更好的上界。实际上，虽然一个单独的 MULTIPOP 操作可能代价很高，但在一个空栈上执行 n 个 PUSH、POP 和 MULTIPOP 的操作序列，代价至多是 $O(n)$。这是为什么呢？当将一个对象压入栈后，我们至多将其弹出一次。因此，对一个非空的栈，可以执行的 POP 操作的次数(包括了 MULTIPOP 中调用 POP 的次数)最多与 PUSH 操作的次数相当，即最多 n 次。因此，对任意的 n 值，任意一个由 n 个 PUSH、POP 和 MULTIPOP 组成的操作序列，最多花费 $O(n)$时间。一个操作的平均时间为 $O(n)/n=O(1)$。在聚合分析中，我们将每个操作的摊还代价设定为平均代价。因此，在此例中，所有三种栈操作的摊还代价都是 $O(1)$。

再次强调，虽然我们已经证明一个栈操作的平均代价，也就是平均运行时间为 $O(1)$，但并未使用概率分析。我们实际上得出的是一个 n 个操作的序列的最坏情况运行时间 $O(n)$，再除以 n 得到了每个操作的平均代价，或者说摊还代价。

二进制计数器递增

作为聚合分析的另一个例子，我们来看一个 k 位二进制计数器递增的问题，计数器的初值为 0。我们用一个位数组 $A[0..k-1]$作为计数器，其中 $A.length=k$。当计数器中保存的二进制值为 x 时，x 的最低位保存在 $A[0]$中，而最高位保存在 $A[k-1]$中，因此 $x=\sum_{i=0}^{k-1} A[i]\cdot 2^i$。初始时 $x=0$，因此对所有 $i=0, 1, \cdots, k-1$，$A[i]=0$。为了将 1(模 2^k)加到计数器的值上，我们使用如下过程：

```
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
```

图 17-2 显示了将一个二进制计数器递增 16 次的情况，初始值为 0，最终变为 16。当每次开始执行第2～4行的 **while** 循环时，我们希望将 1 加在第 i 位上。如果 $A[i]=1$，那么加 1 操作会将第 i 位翻转为 0，并产生一个进位——在下一步循环迭代时将 1 加到第 $i+1$ 位上。否则，循环结束，此时若有 $i<k$，我们知道 $A[i]=0$，因此第 6 行将 1 加到第 i 位上——将第 i 位翻转为 1。每次 INCREMENT 操作的代价与翻转的二进制位的数目呈线性关系。

与上一个关于栈的例子类似，对此算法的运行时间进行粗略的分析会得到一个正确但不紧的界。最坏情况下 INCREMENT 执行一次花费 $\Theta(k)$时间，最坏情况当数组 A 所有位都为 1 时发生。因此，对初值为 0 的计数器执行 n 个 INCREMENT 操作最坏情况下花费 $O(nk)$时间。

计数器值	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	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
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

图 17-2 一个 8 位的二进制计数器，经过 16 次的 INCREMENT 操作，其值从 0 增长到 16。发生翻转而取得下一个值的位加了阴影。右边给出了位翻转所需的运行代价。注意总代价始终不超过 INCREMENT 操作总次数的 2 倍

对于 n 个 INCREMENT 操作组成的序列，我们可以得到一个更紧的界——最坏情况下代价为 $O(n)$，因为不可能每次 INCREMENT 操作都翻转所有的二进制位。如图 17-2 所示，每次调用 INCREMENT 时 $A[0]$确实都会翻转。而下一位 $A[1]$，则只是每两次调用翻转一次，这样，对一个初值为 0 的计数器执行一个 n 个 INCREMENT 操作的序列，只会使 $A[1]$翻转$\lfloor n/2 \rfloor$次。类似地，$A[2]$每 4 次调用才翻转一次，即执行一个 n 个 INCREMENT 操作的序列的过程中翻转$\lfloor n/4 \rfloor$次。一般地，对一个初值为 0 的计数器，在执行一个由 n 个 INCREMENT 操作组成的序列的过程中，$A[i]$会翻转$\lfloor n/2^i \rfloor$次($i=0, 1, \cdots, k-1$)。对 $i \geqslant k$，$A[i]$不存在，因此也就不会翻转。因此，由公式(A.6)知，在执行 INCREMENT 序列的过程中进行的翻转操作的总数为

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

454
~
455

因此，对一个初值为 0 的计数器，执行一个 n 个 INCREMENT 操作的序列的最坏情况时间为 $O(n)$。每个操作的平均代价，即摊还代价为 $O(n)/n=O(1)$。

练习

17.1-1 如果栈操作包括 MULTIPUSH 操作，它将 k 个数据项压入栈中，那么栈操作的摊还代价的界还是 $O(1)$吗？

17.1-2　证明：如果 k 位计数器的例子中允许 DECREMENT 操作，那么 n 个操作的运行时间可能达到 $\Theta(nk)$。

17.1-3　假定我们对一个数据结构执行一个由 n 个操作组成的操作序列，当 i 严格为 2 的幂时，第 i 个操作的代价为 i，否则代价为 1。使用聚合分析确定每个操作的摊还代价。

17.2　核算法

用**核算法**(accounting method)进行摊还分析时，我们对不同操作赋予不同费用，赋予某些操作的费用可能多于或少于其实际代价。我们将赋予一个操作的费用称为它的**摊还代价**。当一个操作的摊还代价超出其实际代价时，我们将差额存入数据结构中的特定对象，存入的差额称为**信用**。对于后续操作中摊还代价小于实际代价的情况，信用可以用来支付差额。因此，我们可以将一个操作的摊还代价分解为其实际代价和信用(存入的或用掉的)。不同的操作可能有不同的摊还代价。这种方法不同于聚合分析中所有操作都赋予相同摊还代价的方式。

我们必须小心地选择操作的摊还代价。如果我们希望通过分析摊还代价来证明每个操作的平均代价的最坏情况很小，就应确保操作序列的总摊还代价给出了序列总真实代价的上界。而且，与聚合分析一样，这种关系必须对所有操作序列都成立。如果用 c_i 表示第 i 个操作的真实代价，用 $\hat{c}_i$ 表示其摊还代价，则对任意 n 个操作的序列，要求

$$\sum_{i=1}^{n} \hat{c}_i \geqslant \sum_{i=1}^{n} c_i \tag{17.1}$$

数据结构中存储的信用恰好等于总摊还代价与总实际代价的差值，即 $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i$。由不等式(17.1)知，数据结构所关联的信用必须一直为非负值。如果在某个步骤，我们允许信用为负值(前面操作缴费不足，承诺在随后补齐账户欠费)，那么当时的总摊还代价就会低于总实际代价，对于到那个时刻为止的操作序列，总摊还代价就不再是总实际代价的上界了。因此，我们必须注意保持数据结构中的总信用永远为非负值。 456

栈操作

为了说明摊还分析的核算法，我们回到栈的例子。回顾前面的内容，操作的实际代价为

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

其中 k 是提供给 MULTIPOP 的参数，s 是调用时栈的规模。我们为这些操作赋予如下摊还代价：

PUSH	2
POP	0
MULTIPOP	0

注意，MULTIPOP 的摊还代价是一个常数(0)，而其实际代价是变量。在此例中，所有三个摊还代价都是常数。一般来说，所考虑的操作的摊还代价可能各不相同，渐近性也可能不同。

我们将证明，通过按摊还代价缴费，我们可以支付任意的栈操作序列(的实际代价)。假定使用 1 美元来表示一个单位的代价。我们从一个空栈开始。回忆 10.1 节中数据结构栈和自助餐店里一叠盘子间的类比。当将一个盘子放在一叠盘子的最上面，我们用 1 美元支付压栈操作的实际代价，将剩余的 1 美元存为信用(共缴费 2 美元)。在任何时间点，栈中的每个盘子都存储了与之对应的 1 美元的信用。

每个盘子存储的 1 美元，实际上是作为将来它被弹出栈时代价的预付费。当执行一个 POP 操作时，并不缴纳任何费用，而是使用存储在栈中的信用来支付其实际代价。为了弹出一个盘

子，我们取出此盘子的 1 美元的信用来支付 POP 操作的实际代价。因此，通过为 PUSH 操作多
457 缴一点费，我们可以在 POP 时不缴纳任何费用。

而且，对于 MULTIPOP 操作，我们也可以不缴纳任何费用。为了弹出第一个盘子，我们将其 1 美元信用取出来支付此 POP 操作的实际代价。为了弹出第二个盘子，我们再次取出盘子的 1 美元信用来支付此 POP 操作的实际代价，依此类推。因此，预付的费用总是足够支付 MULTIPOP 操作的代价。换句话说，由于栈中的每个盘子都存有 1 美元的信用，而栈中的盘子数始终是非负的，因此可以保证信用值也总是非负的。因此，对任意 n 个 PUSH、POP、MULTIPOP 操作组成的序列，总摊还代价为总实际代价的上界。由于总摊还代价为 $O(n)$，因此总实际代价也是。

二进制计数器递增

作为核算法的另一个例子，我们分析在一个从 0 开始的二进制计数器上执行 INCREMENT 操作。如我们之前所观察到的，此操作的运行时间与翻转的位数成正比，因此对此例，可以将翻转的位数作为操作的代价。我们再次使用 1 美元表示一个单位的代价(在此例中是翻转 1 位)。

在摊还分析中，对一次置位操作，我们设其摊还代价为 2 美元。当进行置位时，用 1 美元支付置位操作的实际代价，并将另外 1 美元存为信用，用来支付将来复位操作的代价。在任何时刻，计数器中任何为 1 的位都存有 1 美元的信用，这样对于复位操作，我们就无需缴纳任何费用，使用存储的 1 美元信用即可支付复位操作的代价。

现在可以确定 INCREMENT 的摊还代价。**while** 循环中复位操作的代价用该位储存的 1 美元来支付。INCREMENT 过程至多置位一次(第 6 行)，因此，其摊还代价最多为 2 美元。计数器中 1 的个数永远不会为负，因此，任何时刻信用值都是非负的。所以，对于 n 个 INCREMENT 操作，总摊还代价为 $O(n)$，为总实际代价的上界。

练习

17.2-1 假定对一个规模永远不会超过 k 的栈执行一个栈操作序列。执行 k 个操作后，我们复制整个栈来进行备份。通过为不同的栈操作赋予适合的摊还代价，证明：n 个栈操作(包
458 括复制栈)的代价为 $O(n)$。

17.2-2 用核算法重做练习 17.1-3。

17.2-3 假定我们不仅对计数器进行增 1 操作，还会进行置 0 操作(即将所有位复位)。设检测或修改一个位的时间为 $\Theta(1)$，说明如何用一个位数组来实现计数器，使得对一个初值为 0 的计数器执行一个由任意 n 个 INCREMENT 和 RESET 操作组成的序列花费时间 $O(n)$。(提示：维护一个指针一直指向最高位的 1。)

17.3 势能法

势能法摊还分析并不将预付代价表示为数据结构中特定对象的信用，而是表示为“势能”，或简称“势”，将势能释放即可用来支付未来操作的代价。我们将势能与整个数据结构而不是特定对象相关联。

势能法工作方式如下。我们将对一个初始数据结构 D_0 执行 n 个操作。对每个 $i=1, 2, \cdots, n$，令 c_i 为第 i 个操作的实际代价，令 D_i 为在数据结构 D_{i-1} 上执行第 i 个操作得到的结果数据结构。**势函数** Φ 将每个数据结构 D_i 映射到一个实数 $\Phi(D_i)$，此值即为关联到数据结构 D_i 的**势**。第 i 个操作的**摊还代价** $\hat{c}_i$ 用势函数 Φ 定义为：

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \tag{17.2}$$

因此，每个操作的摊还代价等于其实际代价加上此操作引起的势能变化。由公式(17.2)，n 个操

作的总摊还代价为

$$\sum_{i=1}^{n}\hat{c}_i=\sum_{i=1}^{n}(c_i+\Phi(D_i)-\Phi(D_{i-1}))=\sum_{i=1}^{n}c_i+\Phi(D_n)-\Phi(D_0) \tag{17.3}$$

第二个等式是根据公式(A.9)推导出来的，因为 $\Phi(D_i)$项是交叠的，可以消去。

如果能定义一个势函数 Φ，使得 $\Phi(D_n)\geqslant\Phi(D_0)$，则总摊还代价 $\sum_{i=1}^{n}\hat{c}_i$ 给出了总实际代价 $\sum_{i=1}^{n}c_i$ 的一个上界。实际中，我们不是总能知道将要执行多少个操作。因此，如果对所有 i，我们 459
要求 $\Phi(D_i)\geqslant\Phi(D_0)$，则可以像核算法一样保证总能提前支付。我们通常将 $\Phi(D_0)$简单定义为 0，然后说明对所有 i，有 $\Phi(D_i)\geqslant 0$。(处理的一种简单方法参见练习 17.3-1，其中 $\Phi(D_0)\neq 0$。)

直觉上，如果第 i 个操作的势差 $\Phi(D_i)-\Phi(D_{i-1})$是正的，则摊还代价$\hat{c}_i$ 表示第 i 个操作多付费了，数据结构的势增加。如果势差为负，则摊还代价表示第 i 个操作少付费了，势减少用于支付操作的实际代价。

公式(17.2)和公式(17.3)定义的摊还代价依赖于势函数 Φ 的选择。不同的势函数会产生不同的摊还代价，但摊还代价仍为实际代价的上界。在选择势函数时，我们常常发现可以做出一定的权衡，是否使用最佳势函数依赖于对时间界的要求。

栈操作

为了展示势能法，我们再次回到栈操作 PUSH、POP 和 MULTIPOP 的例子。我们将一个栈的势函数定义为其中的对象数量。对于初始的空栈 D_0，我们有 $\Phi(D_0)=0$。由于栈中对象数目永远不可能为负，因此，第 i 步操作得到的栈 D_i 具有非负的势，即

$$\Phi(D_i)\geqslant 0=\Phi(D_0)$$

因此，用 Φ 定义的 n 个操作的总摊还代价即为实际代价的一个上界。

下面计算不同栈操作的摊还代价。如果第 i 个操作是 PUSH 操作，此时栈中包含 s 个对象，则势差为

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

则由公式(17.2)，PUSH 操作的摊还代价为

$$\hat{c}_i=c_i+\Phi(D_i)-\Phi(D_{i-1})=1+1=2 \tag{460}$$

假设第 i 个操作是 MULTIPOP(S，k)，将 $k'=\min(k,s)$个对象弹出栈。对象的实际代价为 k'，势差为

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

因此，MULTIPOP 的摊还代价为

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

类似地，普通 POP 操作的摊还代价也为 0。

每个操作的摊还代价都是 $O(1)$，因此，n 个操作的总摊还代价为 $O(n)$。由于我们已经论证了 $\Phi(D_i)\geqslant\Phi(D_0)$，因此，$n$ 个操作的总摊还代价为总实际代价的上界。所以 n 个操作的最坏情况时间为 $O(n)$。

二进制计数器递增

作为势能法的另一个例子，我们再次分析二进制计数器递增问题。这一次，我们将计数器执行 i 次 INCREMENT 操作后的势定义为 b_i——i 次操作后计数器中 1 的个数。

我们来计算 INCREMENT 操作的摊还代价。假设第 i 个 INCREMENT 操作将 t_i 个位复位，则其实际代价至多为 t_i+1，因为除了复位 t_i 个位之外，还至多置位 1 位。如果 $b_i=0$，则第 i 个操作将所有 k 位都复位了，因此 $b_{i-1}=t_i=k$。如果 $b_i>0$，则 $b_i=b_{i-1}-t_i+1$。无论哪种情况，$b_i\leqslant b_{i-1}-t_i+1$，势差为

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

因此，摊还代价为

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

如果计数器从 0 开始，则 $\Phi(D_0) = 0$。由于对所有 i 均有 $\Phi(D_i) \geqslant 0$，因此，一个 n 个 INCREMENT 操作的序列的总摊还代价是总实际代价的上界，所以 n 个 INCREMENT 操作的最坏情况时间为 $O(n)$。

势能法给出了分析计数器问题的一个简单方法，即使计数器不是从 0 开始也可以分析。计数器初始时包含 b_0 个 1，经过 n 个 INCREMENT 操作后包含 b_n 个 1，其中 $0 \leqslant b_0, b_n \leqslant k$（回顾前文，$k$ 是计数器二进制位的数目）。于是可以将公式(17.3)改写为

461

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \hat{c}_i - \Phi(D_n) + \Phi(D_0) \tag{17.4}$$

对所有 $1 \leqslant i \leqslant n$，我们有 $\hat{c}_i \leqslant 2$。由于 $\Phi(D_0) = b_0$ 且 $\Phi(D_n) = b_n$，n 个 INCREMENT 操作的总实际代价为

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

特别要注意，由于 $b_0 \leqslant k$，因此只要 $k = O(n)$，总实际代价就是 $O(n)$。换句话说，如果至少执行 $n = \Omega(k)$ 个 INCREMENT 操作，不管计数器初值是什么，总实际代价都是 $O(n)$。

练习

17.3-1 假定有势函数 Φ，对所有 i 满足 $\Phi(D_i) \geqslant \Phi(D_0)$，但 $\Phi(D_0) \neq 0$。证明：存在势函数 Φ'，使得 $\Phi'(D_0) = 0$，对所有 $i \geqslant 1$ 满足 $\Phi'(D_i) \geqslant 0$，且使用 Φ' 的摊还代价与使用 Φ 的摊还代价相同。

17.3-2 使用势能法重做练习 17.1-3。

17.3-3 考虑一个包含 n 个元素的普通二叉最小堆数据结构，它支持 INSERT 和 EXTRACT-MIN 操作，最坏情况时间均为 $O(\lg n)$。给出一个势函数 Φ，使得 INSERT 操作的摊还代价为 $O(\lg n)$，而 EXTRACT-MIN 操作的摊还代价为 $O(1)$，证明它是正确的。

17.3-4 执行 n 个 PUSH、POP 和 MULTIPOP 栈操作的总代价是多少？假定初始时栈中包含 s_0 个对象，结束后包含 s_n 个对象。

17.3-5 假定计数器初值不是 0，而是包含 b 个 1 的二进制数。证明：若 $n = \Omega(b)$，则执行 n 个 INCREMENT 操作的代价为 $O(n)$。（不要假定 b 是常量。）

462

17.3-6 证明：如何用两个普通的栈实现一个队列（练习 10.1-6），使得每个 ENQUEUE 和 DEQUEUE 操作的摊还代价为 $O(1)$。

17.3-7 为动态整数多重集 S（允许包含重复值）设计一种数据结构，支持如下两个操作：

INSERT(S，x)将 x 插入 S 中。

DELETE-LARGER-HALF(S)将最大的$\lceil |S|/2 \rceil$个元素从 S 中删除。

解释如何实现这种数据结构，使得任意 m 个 INSERT 和 DELETE-LARGER-HALF 操作的序列能在 $O(m)$时间内完成。还要实现一个能在 $O(|S|)$时间内输出所有元素的操作。

17.4 动态表

对某些应用程序，我们可能无法预先知道它会将多少个对象存储在表中。我们为一个表分配一定的内存空间，随后可能会发现不够用。于是必须为其重新分配更大的空间，并将所有对象从原表中复制到新的空间中。类似地，如果从表中删除了很多对象，可能为其重新分配一个更小的内存空间就是值得的。在本节中，我们研究这种动态扩张和收缩表的问题。我们将使用摊还分

析证明，虽然插入和删除操作可能会引起扩张或收缩，从而有较高的实际代价，但它们的摊还代价都是 $O(1)$。而且，我们将看到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。

我们假定动态表支持 TABLE-INSERT 和 TABLE-DELETE 操作。TABLE-INSERT 将一个数据项插入表中，它占用一个**槽**(slot)，即保存一个数据项的空间。同样，TABLE-DELETE 从表中删除一个数据项，从而释放一个槽。用什么样的数据结构来组织动态表并不重要：我们可以使用栈(10.1 节)、堆(第 6 章)或者散列表(第 11 章)。我们也可以使用数组或数组集来实现对象的存储，如 10.3 节中的方法。

我们会发现，分析散列方法时(第 11 章)引入的一个概念用于本节可以方便摊还分析。我们将一个非空表 T 的**装载因子** $\alpha(T)$ 定义为表中存储的数据项的数量除以表的规模(槽的数量)。我们赋予空表(没有数据项)的规模为 0，并将其装载因子定义为 1。如果一个动态表的装载因子被限定在一个常量之下，则其空闲空间相对于总空间的比例永远也不会超过一个常数。 463

我们首先分析只允许插入数据项的情况，然后考虑既允许插入也允许删除的一般情况。

17.4.1　表扩张

我们假定表的存储空间是一个槽的数组。当所有槽都已被使用时，表被填满，此时装载因子为 1[⊖]。在某些软件环境中，当试图向一个满的表插入一个数据项时，唯一的选择是报错退出。但我们假定，我们的软件环境与很多现代软件系统一样，提供了一个内存管理系统，可以根据要求分配和释放内存块。因此，当试图向一个满的表插入一个数据项时，我们可以**扩张**表——分配一个包含更多槽的新表。由于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数组，然后将数据项从旧表复制到新表中。

一个常用的分配新表的启发式策略是：为新表分配 2 倍于旧表的槽。如果只允许插入操作，那么装载因子总是保持在 1/2 以上，因此，浪费的空间永远不会超过总空间的一半。

在下面的伪代码中，我们假定 T 是一个对象，对应表。属性 $T.table$ 保存指向表的存储空间的指针，$T.num$ 保存表中的数据项数量，$T.size$ 保存表的规模(槽数)。初始时令表为空：$T.num = T.size = 0$。

```
TABLE-INSERT(T, x)
 1   if T.size == 0
 2       allocate T.table with 1 slot
 3       T.size = 1
 4   if T.num == T.size
 5       allocate new-table with 2 · T.size slots
 6       insert all items in T.table into new-table
 7       free T.table
 8       T.table = new-table
 9       T.size = 2 · T.size
10   insert x into T.table
11   T.num = T.num + 1
```

464

注意，此处有两个“插入”过程：TABLE-INSERT 自身及第 6 行和第 10 行的**基本插入**(elementary insertion)过程。我们可以将每次基本插入操作的代价设定为 1，然后用基本插入操作的次数来描述 TABLE-INSERT 的运行时间。假定 TABLE-INSERT 的实际运行时间与插入数

⊖ 在某些情况下，例如在一个开地址的散列表中，我们可能将表满定义为装载因子等于某个严格小于 1 的常数(参见练习 17.4-1)。

据项的时间呈线性关系，即第 2 行分配初始表的开销为常量，而第 5 行和第 7 行分配与释放内存空间的开销是由第 6 行的数据复制代价决定的。我们称第 5～9 行执行了一次**扩张**动作。

接下来，我们分析对一个空表执行 n 个 TABLE-INSERT 操作的代价。第 i 个操作的代价 c_i 是怎样的呢？如果当前的表有空间容纳新的数据项(或者这是第一个插入操作)，则 $c_i=1$，因为只需执行一次基本插入操作(第 10 行)。但如果当前表满，会发生一次扩张，则 $c_i=i$：第 10 行基本插入操作的代价为 1，再加上第 6 行将数据项从旧表复制到新表的代价 $i-1$。如果执行 n 个操作，一个操作的最坏情况时间为 $O(n)$，从而可得 n 个操作总运行时间的上界 $O(n^2)$。

这不是一个紧确界，因为在执行 n 个 TABLE-INSERT 操作的过程中，扩张操作是很少的。具体地说，仅当 $i-1$ 恰为 2 的幂时，第 i 个操作才会引起一次扩张。一次插入操作的摊还代价实际上是 $O(1)$，我们可以用聚合分析来证明这一点。第 i 个操作的代价为

$$c_i=\begin{cases} i & \text{若 } i-1 \text{ 恰为 2 的幂} \\ 1 & \text{其他} \end{cases}$$

因此，n 个 TABLE-INSERT 操作的总代价为

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

由于包含至多 n 个代价为 1 的操作，而其他操作的代价形成一个等比数列，所以我们得到了上述结果。由于 n 个 TABLE-INSERT 操作的总代价以 $3n$ 为上界，因此，单一操作的摊还代价至多为 3。

通过使用核算法，我们会对为什么一次 TABLE-INSERT 操作的摊还代价应该为 3 有一点感觉。直观上，处理每个数据项要付出 3 次基本插入操作的代价：将它插入当前表中，当表扩张时移动它，当表扩张时移动另一个已经移动过一次的数据项。例如，假定表的规模在一次扩张后变为 m，则表中保存了 $m/2$ 个数据项，且它当前没有储存任何信用。我们为每次插入操作付 3 美元。立刻发生的基本插入操作花去 1 美元。我们将另外 1 美元储存起来作为插入数据项的信用。我们将最后 1 美元储存起来作为已在表中的 $m/2$ 个数据项中某一个的信用，这样，当表中保存了 m 个数据项已满时，每个数据项都储存了 1 美元，用于支付扩张时基本插入操作的代价。

465

我们也可以用势能法来分析 n 个 TABLE-INSERT 操作的序列，我们还将在 17.4.2 节中用势能法设计一个摊还代价为 $O(1)$ 的 TABLE-DELETE 操作。我们定义一个势函数 Φ，在扩张操作之后其值为 0，而表满时其值为表的规模，这样就可以用势能来支付下次扩张的代价。势函数定义为

$$\Phi(T)=2 \cdot T.num - T.size \tag{17.5}$$

可以满足上述要求。当一次扩张后，我们有 $T.num=T.size/2$，因此 $\Phi(T)=0$。而扩张之前，我们有 $T.num=T.size$，因此 $\Phi(T)=T.num$。势的初值为 0，且表总是至少半满的，即 $T.num \geqslant T.size/2$，于是 $\Phi(T)$ 总是非负的。因此，n 个 TABLE-INSERT 操作的摊还代价之和给出了实际代价之和的上界。

为了分析第 i 个 TABLE-INSERT 操作的摊还代价，我们令 num_i 表示第 i 个操作后表中数据项的数量，$size_i$ 表示第 i 个操作后表的总规模，Φ_i 表示第 i 个操作后的势。初始时，我们有 $num_0=0$，$size_0=0$ 及 $\Phi_0=0$。

如果第 i 个 TABLE-INSERT 操作没有触发扩张，那么有 $size_i=size_{i-1}$，此操作的摊还代价为

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

466

如果第 i 个 TABLE-INSERT 操作触发了一次扩张，则有 $size_i=2 \cdot size_{i-1}$ 及 $size_{i-1}=num_{i-1}=num_i-1$，这意味着 $size_i=2 \cdot (num_i-1)$。因此，此操作的摊还代价为

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

图 17-3 画出了 num_i、$size_i$ 和 Φ_i 随 i 变化的情况。注意势是如何累积来支付表扩张代价的。

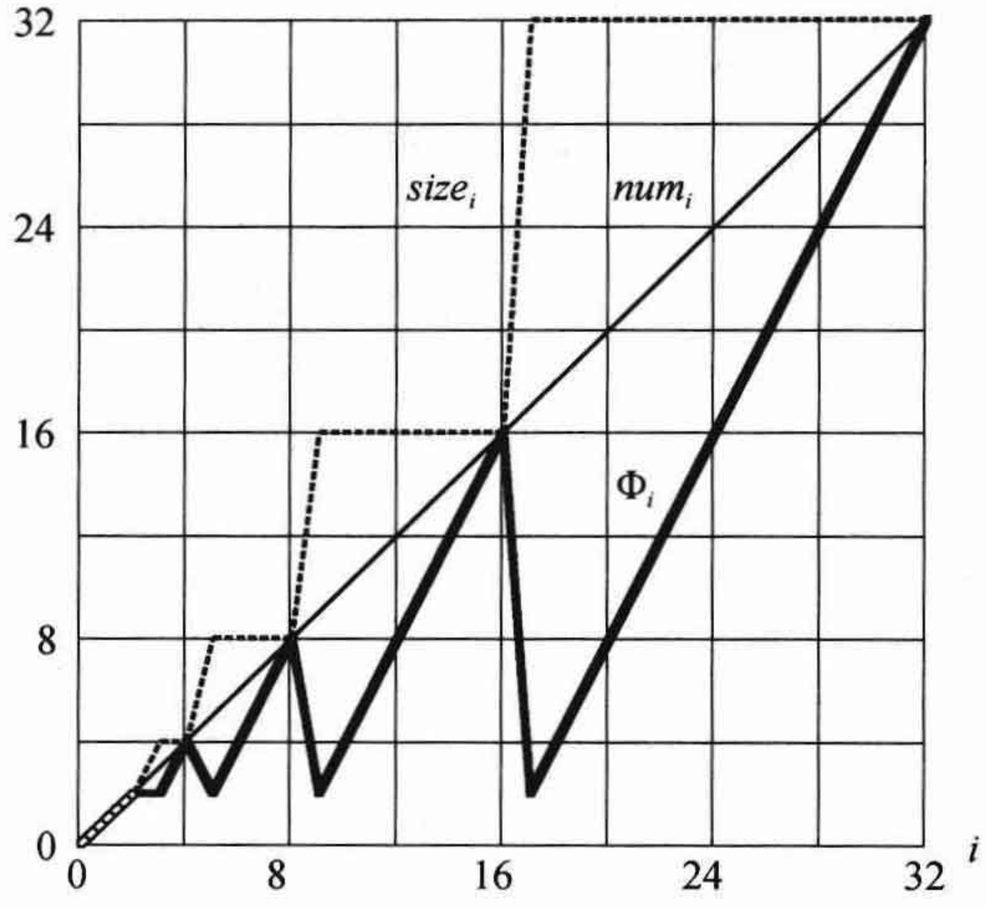

图 17-3　执行 n 个 TABLE-INSERT 操作过程中，表中数据项数量 num_i、表规模 $size_i$ 及势 $\Phi_i = 2 \cdot num_i - size_i$ 的变化，每个值都是在第 i 个操作后测量。细线显示了 num_i 的变化，虚线显示了 $size_i$ 的变化，粗线显示了 Φ_i 的变化。注意，在一次扩张前，势变为表中数据项的数量，因此可以用来支付将所有数据项移动到新表所需的代价。而扩张之后，势变为 0，但会立即变为 2——引起扩张的那个数据项被插入表中

17.4.2　表扩张和收缩

为了实现 TABLE-DELETE 操作，将指定数据项从表中删除是很简单的。但为了限制浪费的空间，我们可以在装载因子变得太小时对表进行**收缩**操作。表收缩与表扩张是类似的操作：当表中的数据项数量下降得太少时，我们分配一个新的更小的表，然后将数据项从旧表复制到新表中。之后可以释放旧表占用的内存空间，将其归还内存管理系统。理想情况下，我们希望保持两个性质：

- 动态表的装载因子有一个正的常数的下界。
- 一个表操作的摊还代价有一个常数上界。

我们假定用基本插入、删除操作的次数来衡量动态表操作的代价。

你可能认为当插入一个数据项到满表时应该将表规模加倍，那么当删除一个数据项导致表空间利用率不到一半时就应该将表规模减半。此策略可以保证表的装载因子永远不会低于 1/2，但遗憾的是，这会导致操作的摊还代价过大。考虑如下场景。我们对一个表 T 执行 n 个操作，其中 n 恰好是 2 的幂。前 $n/2$ 个操作是插入，由之前的分析可知，其总代价为 $\Theta(n)$。在插入序列结束时，$T.num = T.size = n/2$。接下来的 $n/2$ 个操作是这样的：

插入、删除、删除、插入、插入、删除、删除、插入、插入、…

第一个插入操作导致表规模扩张至 n。接下来两个删除操作导致表规模收缩至 $n/2$。接下来两个插入操作引起另一次扩张，依此类推。每次扩张和收缩的代价为 $\Theta(n)$，而收缩和扩张的次数为 $\Theta(n)$。因此，n 个操作的总代价为 $\Theta(n^2)$，使得每个操作的摊还代价为 $\Theta(n)$。

此策略的缺点是很明显的：在表扩张之后，我们无法删除足够多的数据项来为收缩操作支付费用；类似地，在表收缩之后，我们无法插入足够多的表项来支付扩张操作。

我们可以改进此策略，允许表的装载因子低于 1/2。具体地，当向一个满表插入一个新数据项时，我们仍然将表规模加倍，但只有当装载因子小于 1/4 而不是 1/2 时，我们才将表规模减半。因此装载因子的下界为 1/4。

可能我们直觉上认为装载因子为 1/2 比较理想，而表的势此时应该为 0。随着装载因子偏离 1/2，势应该增长，使得当扩张或收缩表时，表已经储存了足够的势来支付复制所有数据项至新表的代价。因此，我们需要这样一个势函数，当装载因子增长为 1 或下降为 1/4 时，势函数值增长为 $T.num$。而表扩张或收缩之后，装载因子重新变为 1/2，而表的势降回 0。

我们略过 TABLE-DELETE 的代码，因为它与 TABLE-INSERT 完全类似。对于分析，我们需要假定无论何时表中数据项数量下降为 0，都会将表占用的内存空间释放掉。也就是说，若 $T.num=0$，则 $T.size=0$。

现在我们用势能法分析 n 个 TABLE-INSERT 和 TABLE-DELETE 操作组成的序列的代价。
468 首先定义一个势函数 Φ，在扩张或收缩操作之后其值为 0，而当装载因子增长到 1 或降低到 1/4 时，累积到足够支付扩张或收缩操作代价的值。我们将非空表 T 的装载因子定义为 $\alpha(T)=T.num/T.size$。由于对空表 $T.num=T.size=0$ 且 $\alpha(T)=1$，因此，无论表是否为空，我们总是有 $T.num=\alpha(T)\cdot T.size$。定义势函数如下：

$$\Phi(T)=\begin{cases}2\cdot T.num-T.size & \text{若 } \alpha(T)\geqslant 1/2\\ T.size/2-T.num & \text{若 } \alpha(T)<1/2\end{cases} \tag{17.6}$$

观察到空表的势为 0，且势永远不可能为负。因此，用势函数 Φ 定义的操作序列的总摊还代价是总实际代价的上界。

在进行精确分析之前，我们先观察图 17-4 所示的势函数的一些特性。注意到，当装载因子为 1/2 时，势为 0。当装载因子为 1 时，$T.size=T.num$，意味着 $\Phi(T)=T.num$，因此，势足够支付插入操作引起的表扩张的代价。当装载因子为 1/4 时，我们有 $T.size=4\cdot T.num$，这意味着 $\Phi(T)=T.num$，因此，势也足够支付删除操作引起的表收缩的代价。

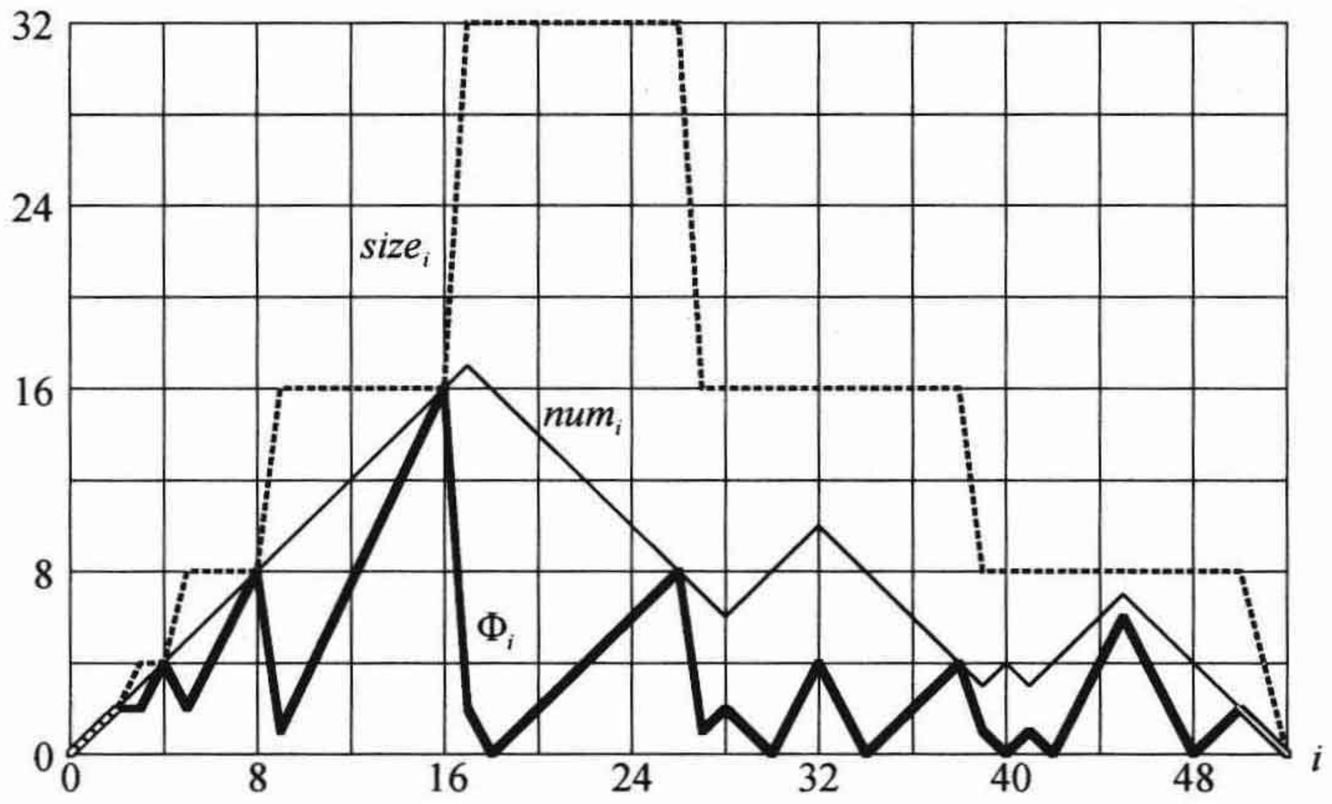

图 17-4 执行 n 个 TABLE-INSERT 和 TABLE-DELETE 操作的过程中，表中数据项数量 num_i、表规模 $size_i$ 及势的变化情况，其中势的定义为

$$\Phi_i=\begin{cases}2\cdot num_i-size_i & \text{若 } \alpha_i\geqslant 1/2\\ size_i/2-num_i & \text{若 } \alpha_i<1/2\end{cases}$$

每个值都是在第 i 个操作后测量得到的。细线显示了 num_i 的变化，虚线显示了 $size_i$ 的变化，粗线显示了 Φ_i 的变化。注意，在一次扩张前，势累积到表中数据项的数量，因此可以用来支付扩张过程中数据项移动的代价。同样，在一次收缩前，势也累积到表中数据项的数量

为了分析 n 个 TABLE-INSERT 和 TABLE-DELETE 操作的序列，令 c_i 表示第 i 个操作的实际代价，$\hat{c}_i$ 为用势函数 Φ 定义的摊还代价，num_i 表示表中第 i 个操作后存储的数据项的数量，$size_i$ 表示第 i 个操作后表的规模，α_i 表示第 i 个操作后的装载因子，Φ_i 表示第 i 个操作后的势。初始时，$num_0=0$，$size_0=0$，$\alpha_0=1$，$\Phi_0=0$。

首先分析第 i 个操作为 TABLE-INSERT 的情况。若 $\alpha_{i-1}\geqslant 1/2$，分析与 17.4.1 节表扩张的分析相同。无论表是否扩张，操作的摊还代价 $\hat{c}_i$ 至多为 3。若 $\alpha_{i-1}<1/2$，则第 i 个操作并不能令表扩张，因为只有当 $\alpha_{i-1}=1$ 时表才会扩张。若 α_i 也小于 1/2，则第 i 个操作的摊还代价为

$$\begin{aligned}\hat{c}_i &= c_i+\Phi_i-\Phi_{i-1}\\ &= 1+(size_i/2-num_i)-(size_{i-1}/2-num_{i-1})\\ &= 1+(size_i/2-num_i)-(size_i/2-(num_i-1))\\ &= 0\end{aligned}$$

若 $\alpha_{i-1}<1/2$ 但 $\alpha_i\geqslant 1/2$，则

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

因此，一个 TABLE-INSERT 操作的摊还代价至多为 3。

我们现在来分析第 i 个操作是 TABLE-DELETE 的情况。在此情况下，$num_i=num_{i-1}-1$。若$\alpha_{i-1}<1/2$，则必须考虑删除操作是否引起表收缩。如果未引起表收缩操作，则 $size_i=size_{i-1}$且操作的摊还代价为

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

469
⌇
470

若 $\alpha_{i-1}<1/2$ 且第 i 个操作触发了收缩操作，则操作的实际代价为 $c_i=num_i+1$，因为我们删除了一个数据项，又移动了 num_i 个数据项。我们知道 $size_i/2=size_{i-1}/4=num_{i-1}=num_i+1$，因此操作的摊还代价为

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

当第 i 个操作是 TABLE-DELETE 且 $\alpha_{i-1}\geqslant 1/2$ 时，摊还代价上界是一个常数，我们将这个分析过程留作练习 17.4-2。

总之，由于每个操作的摊还代价的上界是一个常数，在一个动态表上执行任意 n 个操作的实际运行时间是$O(n)$。

练习

17.4-1　假定我们希望实现一个动态的开地址散列表。为什么我们需要当装载因子达到一个严格