算法设计与分析

平摊分析
Amortized Analysis

Ch.17 平摊分析

- 1985年，Robert提出一种算法效率分析方法。
- 我们已经学过了渐进效率分析，摊还分析与渐进效率有什么不同?
- 摊还分析比渐进效率分析更准确。例如，对于一个在需要增加大小的动态数组，正常的渐近分析只会认为添加一个元素的成本是O（n），因为它可能需要增长和复制所有元素到新的数组。摊还分析认为其中n / 2项不会引起数组扩张，其平均成本是O(n)/(n/2)=O（1）。

Ch.17 平摊分析

- 渐近分析是关于在大数据集上一个给定的操作的性能分析方法。摊销分析是在一个大的数据集上如何对所有的操作的效率进行平均。
- 摊还分析不比渐近效率差，有时会给出更好的效率估算。
- 如果你关心的是一个较长的工作总运行时间、摊还分析得到的效率你会更喜欢。这就是为什么在数组和哈希表中添加一个元素可能代价很大，但是均摊下来代价就很小。例如增长数组要O（n）的操作，但摊销是O（1）。
- 如果你是做实时编程（操作必须在可预见的时间内完成），平均运行速度快不重要，摊还分析就不重要。

摊还分析与渐进分析的区别

- 渐近和摊还分析之间的关键区别在于：前者是依赖于**输入**本身，后者则是依赖于将**执行的操作序列**。
- 渐近分析使我们能够断言，一个算法在给定一个最好的/最坏/平均情况下输入规模n的函数$f(n)$——其中n是输入规模；
- 摊销分析允许我们断言，操作个数已知的情况下，算法性能不会比函数$f(n)$——其中n操作序列的长度。

Ch.17 平摊分析

- **总体考虑效率分析问题**
- 代价分析技术：通过n次操作来计算平均代价
- 目标：某些单个操作代价太高，但整体平均代价低。
- 准确估计分析最差情形(*worst case*)的计算复杂度上界，平摊每个操作的平均代价

Ch.17 平摊分析

- 与平均情况分析的区别

  平摊分析不涉及到概率，保证其平摊性能是每个操作在最坏情况下具有的平均性能
- 三种平摊分析技术
  - 合计法、聚合方法（aggregate）
  - 核算法、会计法（accounting）
  - 势能法（potential）

17.1 聚合法/合计法

计算对所有的n个操作序列在最坏情况下的总时间为T(n)。因此，最坏情况下每个操作的平摊代价为T(n)/n

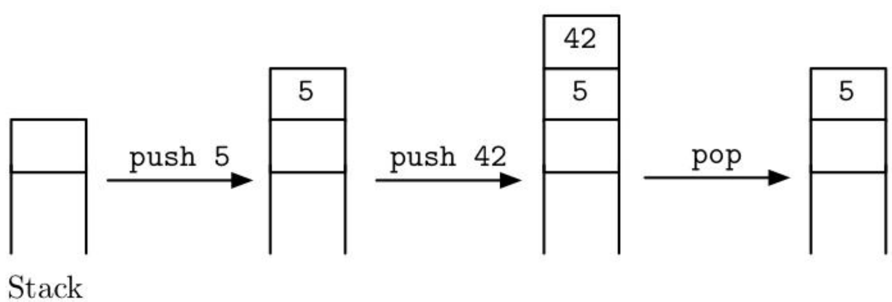
- 注意
  - 所有操作都赋予相同的平摊代价，即使这些操作不相同。
  - 各操作的平摊代价相同

17.1聚合法/合计法

1、栈操作（不同种类）

- 数据结构：栈S，初值为空
- 操作
  - Push(S, x):　　O(1)
  - Pop(S):　　　　O(1)
  - MultiPop(S, k): O(min(|S|, k))

  弹出min(|S|, k)个对象



17.1聚合法/合计法

栈上操作序列的时间分析

- 渐进分析方法得不到紧确界
- 设有n个Push，Pop和Multipop操作构成的序列作用在初值为空的栈S上。
- 一次Multipop的最坏时间为O(n)，因为$|S| \le n$，$K=O(n)$；
- 最坏情况下可能有O(n)个Multipop。
- 因此，该序列最坏时间为$O(n^2)$

估计的最坏情况的上界不准确

***Question*: 这样分析算法合理?**

***Answer*: 只有n个操作，不可能每个操作都需要n次操作。**

17.1聚合法/合计法

栈上操作序列的时间分析（续）

- 聚合法可以得到更准确的界（紧确界）
- 因为一个对象入栈后至多被弹出一次，所以在非空栈上调用Pop的次数（包括Multipop中调用的Pop）至多为Push的次数
- 因此，当S初值为空时，Push次数至多为O(n)，而Pop次数至多为O(n)（包括Multipop个数）
- 所以对任意整数n，操作序列长度为n时，总时间T(n)=O(n)，三个操作的平摊代价均为O(n)/n=O(1)
- 注意：没有使用概率分析

17.1聚合法/合计法

2、二进制计算器（同一类操作）

- 数据结构

设A[0..k-1]数组作为二进制计数器，初值为0，

A[i]=0, 0 ≤i ≤k-1。A中存储二进制数x：

$$x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$$ //低位在A[0]中，

A的分量 ⇔ 二进制位

- 操作：将二进制计数器A的值加1（模2^k，$0 \le x \le 2^k-1$）

```
Increment(A) {
  //最高位进位舍弃，即当X = 2^k时，A中全为0
  i ← 0;
  while(i < length[A]) and (A[i] = 1) do {
    //从低到高位扫描，若当前位为1，将其翻转
    //为0，进位向前，直到找到一个值为0的位或
    //所有位已扫描完为止
    A[i] ← 0;  //当前位为1，加1后为0
    i++;       //进位加到更高位上
  };
  if i<length[A] then  //i位上为0，且x<2^k-1
    A[i] ← 1;          //将进位加到第i位上
  //否则x=2^k-1，加1后A[0..k-1]=0
}
```

例子：二进制计数器Binary Counter

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
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

- **8-bit二进制计算器从0 变化到16对应的操作变化.**
- **每次操作平均代价31/16 < 2.**

17.1聚合法/合计法

- 时间分析
  - 渐进分析

    最坏时，Increment改变值的位数为O(k)　(k为二进制位数，例如k=8)，n个增量作用在初值为0的计数器上，总代价为O(nK)
  - 聚合法/合计法分析

    n次增量操作中，并非每次所有位都发生变化，上表告诉我们，无论n值为何，其总翻转位数<2n，即总代价为O(n)，更严格证明如下：

17.1聚合法/合计法

- 当位i ≤ └ lgn┘ 时（n为累加次数）

A[0], 每调用1次翻转1次，共n 次翻转

A[1], 每调用2次翻转1次，共└ n/2┘ 次翻转

A[2], 每调用4次翻转1次，共└ n/4┘ 次翻转

……

A[i], 每调用2^i次翻转1次，共└ n/2^i┘ 次翻转

- 当位i>└ lgn┘ 时

∵n的二进制表示最多有└ lgn┘ +1位

∴A[i]在n次增量操作中，始终不变(高位不翻转)。于是，n次操作总的翻转次数为:

17.1聚合法/合计法

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n, \text{对任意}n\text{成立}$$

即：

操作序列总代价(在最坏情况下)为O(n)

每次操作的平摊成本为O(n)/n=O(1)

17.2 核算法/记账法

- 费用分配（平摊代价）

为不同的操作分配不同的费用，每一操作分配到的费用称为该操作的平摊代价，它可视作为数据结构对该操作预收的费用（或理解为是该操作对数据结构预付的费用）

17.2核算法/记账法

- 超额收费(overcharge)(平摊代价>实际成本)

当一个操作的平摊代价大于其实际成本时，数据结构对该操作预收的费用过多，其超额部分作为存款存储在数据结构的某个特定对象上

- 收费不足(undercharge)(平摊代价<实际成本)

当一操作的平摊代价小于其实际成本时，数据结构对该操作预收的费用不足，其差额部分可由数据结构的特定对象(是该操作操作的对象)上的存款支付

- 以丰补歉

与合计法不同，这里不同的操作平摊代价可能不同，原则是以丰补歉

17.2核算法/记账法

- 正确选择各操作的平摊代价（平摊代价的正确性）

要说明每个操作的平摊代价是最坏情况下的平均代价，则必须保证对任意长度**n**的操作序列，总平摊代价是总的实际代价的一个上界：

$$\sum_{i=1}^{n} \hat{C}_i \geq \sum_{i=1}^{n} C_i; \ for \ any \ n$$

其中c_i **是实际的第*i*次操作代价，**$\hat{c}_i$ **是平摊的*i*次操作代价**

17.2核算法/记账法

- 正确选择各操作的平摊代价(续)

  或者说：与数据结构相关的总存款在任何时刻都必须非负：

$$\sum_{i=1}^{n}\hat{C}_i-\sum_{i=1}^{n}C_i \geq 0$$

  否则，若某个n时总存款为负，则对于该时刻为止的操作序列（即对某个n），其总平摊代价不是总实际代价的上界，此时，各操作的平摊代价就不是最坏情况下的平均代价。

17.2 记账法

1、栈操作（不同种类操作）

	实际代价	平摊代价
Push	1	2
Pop	1	0
Multipop	min(k,s)	0

- 首先要保证这种平摊代价定义的正确性

  即对任何n，总平摊代价是总的实际成本的上界。

17.2核算法/记账法

求证：上述的平摊成本是实际代价的上界。

证明：

设每个代价单位：1元，　　　　栈——餐馆的盘子

Push： 将一盘子放入栈中，该操作付出的2元钱1元用来支付入栈操作的实际成本，剩余1元作为存款放到刚刚入栈的盘子上

Pop： 平摊成本为0，因为栈中每个盘子上均有1元存款，故可用该存款支付每个盘子出栈所需的实际成本

Multipop： 平摊成本为0，每弹出一个盘子，就利用这个盘子上的1元存款，每个盘子都有存款，不用再支付成本

综上所述：任意时刻，栈中盘子总存款≥0，上述的平摊成本是正确的。

17.2核算法/记账法

结论：

任意时刻，栈中盘子总存款≥0，故任意长度n的操作序列，总平摊成本是……的一个上界。

Push操作的平摊成本最大O(1)

∴每个操作平摊成本是O(1)，n个由Push，Pop和Multipop组成的操作序列总平摊成本是O(n)，因此总实际成本亦为O(n)。

17.2核算法/记账法

2、二进制计数器的增量（同种操作）

- Increment的实际成本——翻转位数，设每位翻转代价：1元
- 操作的平摊成本：

  置位(0→1)收费2元

  复位(1→0)收费0元

- 正确性：

  当某位被置1时，2元收费中1元用于支付实际成本，另1元存在该位上，则计数器中，值为1的位上均有1元存款。

  当某位复位时，用该位上1元存款来支付复位的实际成本。

  所以平摊成本的定义是正确的。

17.2核算法/记账法

∵任意时刻，计数器中值为1的位数非负，故总存款数≥0

∴n次增量操作的总平摊成本是总实际成本的一个上界。即一次增量操作的平摊成本是最坏情况的平均成本

∵一次操作最多只有一次置位($0 \rightarrow 1$)，则其平摊成本≤ 2

∴n次增量操作的总平摊成本$2n \in O(n)$

17.3 势能法

- 与记账法的区别

  存款——作为势能保存在整个数据结构上，而非其中的特定的对象上。需要时通过释放能量来支付操作的实际代价

- 势能、势差、势函数、各操作的平摊成本
  - 数据结构：D，初态记为D_0
  - 操作：n个，n可变
  - i^{th}操作OP_i的结果：$D_{i-1} \rightarrow D_i$，D_{i-1}和D_i表示第i个操作前后D的状态$(1 \le i \le n)$，数据结构从D_{i-1}的状态转换为D_i状态。

17.3 势能法

- 势函数$\Phi(D_i)$：对每个数据结构的状态D_i定义一个实数($1\le i\le n$)，这个值就是势能，描述当前数据结果的势。
- OP_i的实际成本为C_i
- OP_i的平摊成本为$\hat{C}_i = C_i+\Phi(D_i)-\Phi(D_{i-1})$

操作的平摊代价由两部分构成：实际成本C_i和势差（操作所引起的势能变化）

$$\begin{cases} \Phi(D_i)-\Phi(D_{i-1})>0, \hat{C}_i\text{超额收费，势能}\uparrow \\ \Phi(D_i)-\Phi(D_{i-1})<0, \hat{C}_i\text{收费不足，势能}\downarrow \\ \Phi(D_i)-\Phi(D_{i-1})=0, \quad \hat{C}_i=C_i \end{cases}$$

17.3 势能法

- 总的平摊代价及势函数的选择(正确性)

$$\sum_{i=1}^{n}\hat{C}_i = \sum_{i=1}^{n}(C_i + \Phi(D_i) - \Phi(D_{i-1}))$$

$$= \sum_{i=1}^{n} C_i + \Phi(D_n) - \Phi(D_0) \qquad (17.3)$$

即n个操作后，势差为：序列最终的势能和最初的势能之差。此势差≥0或$\Phi(D_n) \geq \Phi(D_0)$，则总的平摊代价是总的实际成本的一个上界，但须对任意的n成立，故有：

$\forall i \in I$，$\Phi(D_i) \geq \Phi(D_0)$，通常定义$\Phi(D_0) = 0$

则有：$\forall i \in I$，$\Phi(D_i) \geq 0$

17.3 势能法

$\Phi(D_0) = 0$，$\forall i \in I$，$\Phi(D_i) \geq 0$

- 满足上式就可以保证平摊代价的正确性。
- 势函数选择常常需要某种折衷，根据相应的时间界来选定，最佳选择往往使界尽量紧致。

17.3 势能法

1、栈操作

- 势函数Φ：栈中对象数目
  - 数据结构：栈
  - 初始：设D_0表示空栈，$\Phi(D_0)=0$
  - 对任意i，$\Phi(D_i) \geq 0 = \Phi(D_0)$，即任意时刻，栈中对象数非负
  - 结论：Φ定义保证了n个操作的总平摊代价是总的实际代价的一个上界

17.3 势能法

- 各栈操作的平摊成本

设当前栈中对象数$|S|=s$，OP_i是：

①Push

Push操作的实际代价，1

势差： $\Phi(D_i)-\Phi(D_{i-}$ //多收费

平摊成本： $\hat{C}_i = C_i+\Phi(D_i)-\Phi(D_{i-1})=1+1=2$ $//C_i=1$

②Pop

势差： $\Phi(D_i)-\Phi(D_{i-1}) = (s-1)-s = -1$ //少收费

平摊成本： $\hat{C}_i = C_i+\Phi(D_i)-\Phi(D_{i-1})=1-1=0$ $//C_i=1$

③Multipop(S,k): 弹出对象数$k'=\min(s,k)$

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

平摊成本： $\hat{C}_i = C_i-k' = 0$ $//\ C_i = k'$

17.3 势能法

- 总的平摊成本

T(n)=O(n)   //各操作的平摊成本为O(1)

$$\sum \hat{C}_i \geq \sum C_i$$

17.3 势能法

2、二进制计数器的增量操作

- 势函数Φ：执行i次增量操作后，计数器中1的数目b_i

1)初始势能为0

$\Phi(D_0)=0$，计数器初值为0

$\forall i$， $\Phi(D_i)\geq 0=\Phi(D_0)$，即任一时刻，计数器中1的数目非负

17.3 势能法

①实际成本

假设OP_i中将t_i位复位（置0），则实际代价最多为t_i+1（因为最多再多1个置位），即 $C_i \le t_i+1$

②势差

设OP_i操作之后，计数器中1的数目为b_i，即:

$\Phi(D_i)= b_i$。

势差

- 势差： $\Phi(D_i) - \Phi(D_{i-1}) = b_i - b_{i-1}$
- 如果$b_i = 0$, OP_i操作将所有k位置0，但未置位为1,
  因此 $b_{i-1} = t_i = k$ ➔ $b_i = b_{i-1} - t_i = 0$.（该情况发生于当所有k位都为1时）

  t_i位置0，1的个数少了t_i个
- 如果$b_i > 0$, OP_i操 置一位为1,
  因此$b_i = b_{i-1} - t_i + 1$.

  1的个数多1个

综合可得 $b_i \le b_{i-1} - t_i + 1$ //若有置位，左右应相等

17.3 势能法—计算器

- 因为 $b_i \le b_{i-1} - t_i + 1$,
  $\Delta(D_i) = \Phi(D_i) - \Phi(D_{i-1}) = b_i - b_{i-1} \le (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i$
- 因此, $\hat{c}_i = c_i + \Delta(D_i) \le (t_i + 1) + (1 - t_i) = 2$
- 如果计数器初始为0, $\Phi(D_0) = 0$.

➔ **n个操作的平摊代价=**

$$\sum_{i=1}^{n} \hat{c}_i \le \sum_{i=1}^{n} 2 = 2n = O(n)$$

17.3 势能法

注意：在上式中$\hat{C}_i \leqslant 2$，与b_0是否为0无关，故可利用它直接分析总的实际成本。

这里要求k=O(n)，或执行Increment操作至少k次(n≥Ω(k))，即可保证：

无论计数器的初值为何，总的实际代价都是O(n)

Ch.17 平摊分析

- 合计法/聚合法
  - 简单，常用：先求出合计，然后摊薄
  - 先求出操作序列里所有n个操作的总代价上界T(n)，每个操作的平摊代价T(n)/n
  - 每个操作的平摊代价相同

Ch.17 平摊分析

- 核算法/记账法
  - 对操作序列中的各操作收费，以支付操作的实际代价
  - 每个不同操作的平摊代价可以不同
  - 先确定操作序列中各种操作的平摊代价（费用），对不同的操作收费可以不同
  - 对有的操作超额收费：即该操作实际成本低于该收费，余款作为“预付存款”存储在数据结构某些特殊对象上
  - 对有的操作收费不足：即该操作实际成本大于此收费，不足部分由特殊对象上的存款支付

Ch.17 平摊分析

- 势能法
  - 与记账法类似之处是也须先确定每个操作的平摊成本（收费），对某些操作预先超额收费以补偿后续收费不足的操作
  - 不同之处是：存款是作为整个数据结构的“势能”加以维护，而不是将存款与数据结构中某些个体对象联系起来

Ch.17 平摊分析

- 平摊分析特点

  1）它是一种分析的方法，适用于分析一个彼此相关的操作序列

  其分析方法不是孤立地分析每个操作的时间界限，而是将整个操作序列作为一个整体考虑，利用各操作彼此的关系求整个操作序列的时间界限，然后摊薄得到各操作的平摊代价

  2）操作序列的总代价是操作序列长度的函数，而不是输入量规模的函数

Ch.17 平摊分析

- 平摊分析特点

  3）不仅是分析方法，也是设计算法和数据结构的一种思维方法

  因为设计算法与分析时间性能紧密相关，

  所以通过平摊分析可优化算法设计，加深对算法所操作的数据结构特性的认识，从而设计出时空性能更优的数据结构

17.4. 动态表

一个存储管理系统，须能根据实际需要来动态地分配与释放存储块

- 表扩张：表满时插入x引起扩张

①申请更大的表—需要连续表空间

②原表copy到新表

③释放原表

④插入x

17.4. 动态表

- 表收缩：表中对象数小于某数目时，删去x引起收缩

①申请更小的表

②原表copy到新表

③释放原表

④删去x

17.4. 动态表

- 动态表上插，删操作序列的总成本O(n)

每个插、删的平摊成本为O(1)，尽管某次插、删的实际成本较大

- 装填因子

①非空表：

②空表：$T=\Phi$是指size[T]=0，定义$\alpha(\Phi)=1$

$T\neq\Phi$，num[T]=0时，定义$\alpha(T)=0$

num[T]：表空间被使用情况

17.4. 动态表

- 保证浪费的空间不至于太大

未用空间 / 整个空间≤某常数，或要求表的装填因子有一常数下界：

$$\alpha(T) \geq \beta > 0$$

即：浪费的空间有一常数上界，等价于α有一常数下界。因此有：

未用空间 / 整个空间= (size-num) / size

$$= 1-\alpha \leq 1-\beta$$

//例如：β=0.25，则浪费≤75%

动态表Dynamic Table: Table Expansion

- **When a new insert causes a table overflow, create a new table with double the size of the old table.**

TABLE-INSERT(T, x)

```
if T.size == 0
    allocate T.table with 1 slot
    T.size = 1
if T.num == T.size                                  // expand?
    allocate new-table with 2 · T.size slots
    insert all items in T.table into new-table      // T.num elem insertions
    free T.table
    T.table = new-table
    T.size = 2 · T.size
insert x into T.table                               // 1 elem insertion
T.num = T.num + 1
```

动态表Dynamic Table: Cost Analysis (1)

- ***Question***: 怎么估计插入的最大代价?
- 有两种插入情况:
  - ***Type 1***: 只需简单将数据插入到已有的表中
  - ***Type 2***: 需要创建新表：将旧表数据拷贝到新表中，删除旧表
    - 假设创建新表和释放旧表的时间是常数
- *明显Type 2*是最差的情形，操作代价会非常高，需要拷贝旧表数据
  - 假设插入代价与插入目标的数目相关
- 如何估计n个操作的总共代价?

动态表Dynamic Table: Cost Analysis (2)

- 另c_i 是第i个插入操作代价

$$c_i = \begin{cases} i & \text{if } i-1 \text{ is an exact power of 2} \\ 1 & \text{otherwise} \end{cases}$$

- **Example:**

Operation	**Table Size**	**Cost**
`Insert(1)`	1	1
`Insert(2)`	2	1 + 1
`Insert(3)`	4	1 + 2
`Insert(4)`	4	1
`Insert(5)`	8	1 + 4
`Insert(6)`	8	1
`Insert(7)`	8	1
`Insert(8)`	8	1
`Insert(9)`	16	1 + 8

动态表Dynamic Table: 聚合分析

- N个操作的共代价

$$T(n)=\sum_{i=1}^{n}c_i \le n+\sum_{j=0}^{\lfloor \lg n\rfloor}2^j = n+\frac{2^{\lfloor \lg n\rfloor+1}-1}{2-1}$$

$$\le n+(2n-1)<3n$$

- 每个操作的平均平摊代价$\boldsymbol{T(n)/n=O(1)}$.

➔ 动态表和固定长度表的代价都是
  - 每个操作$\boldsymbol{O(1)}$

17.4.1. 表扩张

③核算法/记账法分析

TableInsert操作的平摊成本为3

当一个元素x新插入表中时(基本插入)，收费3元

i) 其中1元支付x本身的插入

ii) 1元存放到x上作为存款，支付下次扩张时该元素的copy成本

iii)1元作为存款放在上次扩张到本表中(已移动过一次)的某个元素上，作为下次扩张时的copy成本

17.4.1. 表扩张

例如：设当前表大小为m，则m/2个元素是本表扩张前由原表copy过来的，它们没有存款，故当往表中插入后一半元素时，应为前一半元素各存1元，当m个表目填满时，各元素均有1元支付新扩张的copy费用

m/2

x

当前表

上一次扩张

下一次扩张

∵任意时刻总存款≥0

∴总平摊成本O(n)≥总实际成本

17.4.1. 表扩张

④势能分析法

i) 势函数Φ：刚完成扩张时势最小(0)，表满时势能最大(表的项数)，以支付下次扩张copy的代价

$\Phi(T)=2*num[T]-size[T]$ (17.5)

显然：

刚扩张时，$\because num[T]=size[T]/2 \therefore \Phi(T)=0$

表满时，$\because num[T]=size[T] \therefore \Phi(T)=size[T]$

ii)正确性

$\because \alpha\geq 1/2$, $num[T]\geq size[T]/2$ //表至少半满

$\therefore \Phi(T)\geq 0=\Phi_0$

17.4.1. 表扩张

iii)OP_i的平摊成本

设OP_i之后表项数、size及势分别为num_i， $size_i$和Φ_i，显然$num_0=size_0=\Phi_0=0$

(a)若未扩张，则$size_i=size_{i-1}$, $num_i=num_{i-1}+1$

$$\hat{C}_i = C_i + \Phi_i - \Phi_{i-1}$$

$$= 1 + (2num_i - size_i) - (2num_{i-1} - size_{i-1})$$

$$= 1 + (2num_i - size_i) - (2(num_i - 1) - size_i)$$

$$= 3 \text{ //增加2个单位的势}$$

17.4.1. 表扩张

(b)若扩张，则

$size_i=2size_{i-1}= 2(num_i-1)$//$num_{i-1}=num_i-1$

$$\begin{aligned}\hat{C}_i &= num_i + (2num_i - size_i) \\ &\qquad - (2num_{i-1} - size_{i-1}) // \text{替换为} \mathrm{num}_i \\ &= num_i + (2num_i - 2num_i + 2) \\ &\qquad - (2num_i - 2 - num_i + 1) \\ &= 3 // \text{势差为} 3 - num_i \text{或者} 2 - num_{i-1}\end{aligned}$$

17.4.1. 表扩张

Fig17.3： num_i，$size_i$和Φ_i相对于i的关系

扩张前势最大（等于$size_i$=num_i）；

扩张后势为0，但是引起扩张的元素立即插入后，使势马上增加到2.