

# Lecture 6 摊销分析

绳伟光

上海交通大学微纳电子学系

2021-10-21



# 提纲

- 1 摊销分析简介
- 2 聚集方法
- 3 记帐方法
- 4 势能方法
- 5 动态表扩张



# 提纲

- 1 摊销分析简介
- 2 聚集方法
- 3 记帐方法
- 4 势能方法
- 5 动态表扩张



# 摊销 (Amortize)

与摊销相关的事例：

- 以功抵过：某大臣犯死罪，满朝文武求情，要求念及昔日功勋，从轻处理
- 成本摊销：固定资产折旧
- 债务摊销：无息信用卡分期
- 算法：对算法来说，可能存在某次运行不太顺畅，以至于时间复杂度很高，但也许在其他时候，运行非常顺畅，从而综合起来时间复杂度并不高



# 摊销分析概念

## 摊销分析

在摊销分析 (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。但摊销分析不同于平均情况分析，它不涉及概率。

常见摊销分析方法：

- 聚集方法
- 记帐方法
- 势能方法

摊销分析也被称为摊还分析、均摊分析，均指 amortized analysis!



# 摊销分析的意义

- 摊销分析针对的是最坏情况下的平均，而平均情况分析是平均情况下的平均
- 摊销分析针对的是一个操作序列在最坏情况下的运行状况，而平均情况分析针对的是输入在概率分布下算法运行一次的情况
- 平均情况分析需要使用概率，而摊销分析不需要使用概率
- 摊销分析不是将单个操作的代价简单叠加，即如果一次操作代价为  $m$ ，那么  $n$  次操作的代价并不能简单的认为是  $m \times n$
- 摊销分析是要从整体上对算法的效率进行把握，而不是以一次操作的情况来对算法效率做出结论
- 前面的算法分析处理的是渐进复杂度，经常会忽略具体操作的差别；摊销分析则重点关注不同操作间的差别



# 专业相关应用实例

- 在集成电路和体系结构设计中，经常涉及摊销的概念，比如对集成电路研发的一次性工程费用 (Non-recurring engineering, NRE) 的摊销
- 体系结构中的 Cache 会带来面积和功耗代价，需要数据的重用来摊销这些代价
- 在并行系统与并行程序的设计中，核间通信代价很高 (存储墙与通信墙问题)，所以需要提高核内计算的粒度，以更多的计算量摊销通信代价
- PPoPP'10 论文 "Data transformations enabling loop vectorization on multithreaded data parallel architectures" 中，依赖输入数据集大小的增加来摊销数据转换的代价
- HPCA'20 论文 "A Hybrid Systolic-Dataflow Architecture for Inductive Matrix Algorithms" 中，通过新颖的矢量流控制模型创建了可扩展的设计，可在整个架构通道的时间和空间上分摊控制开销

# 提纲

1 摊销分析简介

2 聚集方法

3 记帐方法

4 势能方法

5 动态表扩张





# 聚集方法原理

- 聚集方法分析步骤：
  - 首先证明  $n$  个操作构成的序列在最坏情况下总的时间  $T(n)$
  - 在最坏情况下，每个操作的平均代价就是  $T(n)/n$
- 聚集方法为每个操作赋予相同的摊销代价，即使序列中存在不同类型操作
- 聚集方法中每个操作的摊销代价即为平均代价
- 其它两种方法：记帐方法和势能方法对不同类型操作赋予不同的摊销代价



# 普通栈操作的聚集分析

## 普通栈操作分析：

- 普通栈操作

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

- 时间代价

- 1) 两个操作的运行时间都是  $O(1)$
- 2) 可把每个操作的代价视为 1, 则  $n$  个  $PUSH$  和  $POP$  操作序列的总代价是  $n$
- 3)  $n$  个操作的实际运行时间为  $\Theta(n)$



# 新栈操作的分析

- 新的栈操作:  $\text{MULTIPOP}(S, k)$ : 去掉  $S$  的  $k$  个栈顶元素, 或当  $S$  中包含少于  $k$  个对象时弹出整个栈
- 实现算法:

---

## $\text{MULTIPOP}(S, k)$

---

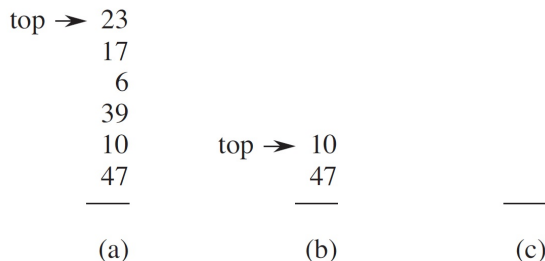
```
1: while not isEmpty( $S$ ) and  $k > 0$  do  
2:   POP( $S$ )  
3:    $k = k - 1$ 
```

---

- $\text{MULTIPOP}$  代价分析:
  - 设  $\text{MULTIPOP}(S, k)$  作用于一个包含  $s$  个对象的栈上
  - 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  $\min(s, k)$
  - $\text{MULTIPOP}$  的总代价即为  $\min(s, k)$



# MULTIPOP 操作示意



**Figure 17.1** The action of **MULTIPOP** on a stack  $S$ , shown initially in (a). The top 4 objects are popped by  $\text{MULTIPOP}(S, 4)$ , whose result is shown in (b). The next operation is  $\text{MULTIPOP}(S, 7)$ , which empties the stack—shown in (c)—since there were fewer than 7 objects remaining.



# 初始为空栈上 $n$ 个操作序列的分析

- 设 PUSH、POP 和 MULTIPOP 构成  $n$  个栈操作序列
- 粗略分析：
  - 1) 序列中一次 MULTIPOP 操作的最坏情况代价为  $O(n)$
  - 2) 任意栈操作的最坏情况时间为  $O(n)$ ， $n$  个操作的总代价就是  $O(n^2)$ ，因为 MULTITPOP 操作可能有  $O(n)$  个，每个代价为  $O(n)$
  - 3) 分析是正确的，但单独分析每个操作的最坏代价得到的操作序列的最坏情况代价  $O(n^2)$  却是不准确的，需通过摊销分析细化
- 细致分析：
  - 1) 对象在每次被压入栈后至多被弹出一次
  - 2) 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  $n$
  - 3) 包含任意  $n$  个 PUSH、POP 和 MULTIPOP 操作的序列的总时间为  $O(n)$
  - 4) 每个操作的摊销代价为： $O(n)/n = O(1)$
  - 5) 在此过程中，并未使用任何概率推理



# 二进制计数器递增

$k$  位二进制计数器递增: 用位数组  $A[0..k-1]$  作为计数器,  $A.length = k$ , 计数值  $x$  的最低位保存在  $A[0]$  中, 最高位保存在  $A[k-1]$  中,

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

---

## INCREMENT( $A$ )

---

```
1:  $i = 0$ 
2: while  $i < A.length$  and  $A[i] = 1$  do
3:    $A[i] = 0$ 
4:    $i = i + 1$ 
5: if  $i < A.length$  then
6:    $A[i] = 1$ 
```



# 8 位二进制计数器递增过程

| 计数值 | 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  |



# 二进制计数器的聚集分析

## 粗略分析

INCREMENT 最坏情况下的复杂度为  $\Theta(k)$ ，即数组所有位都为 1 的时候，因此对初值为 0 的  $n$  个 INCREMENT 操作的最坏情况代价为  $O(nk)$

## 摊销分析

对初值为 0 的  $n$  次 INCREMENT 操作：

- $A[0]$  每次都会发生翻转
- $A[1]$  每 2 次翻转一次
- $A[i]$  每  $2^i$  次翻转一次
- 总计翻转次数：

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

- 最坏情况复杂度  $O(n)$ ，每个操作的摊销代价为  $O(n)/n = O(1)$



# 提纲

1 摊销分析简介

2 聚集方法

3 记帐方法

4 势能方法

5 动态表扩张



# 记帐方法原理

- 记帐方法首先定义每个操作的摊销代价然后再计算总的摊销代价
- 执行不同的操作需要付出不同的费用，某些操作的费用可能比它们的实际代价多或少
- 执行一个操作需要付出的费用称为这个操作的摊销代价
- 当一个操作的摊销代价超过了其实际代价时，两者的差值被作为存款赋给数据结构中一些特定的对象，存款在以后用于补偿那些摊销代价低于其实际代价的操作
- 一个操作的摊销代价可看作为两部分：其实际代价与存款
- 记账方法不同于聚集方法，不同操作具有不同的摊销代价



# 记账方法的注意事项

在选择操作的摊销代价时要非常小心，如果希望通过记账方法分析说明每次操作具有小的最坏情况平均代价，则操作序列的总的摊销代价就必须是该操作序列的总实际代价的一个上界。如果以  $c_i$  表示第  $i$  个操作的真实代价，用  $\hat{c}_i$  表示其摊销代价，则无论何时必须满足：

$$\sum_{i=1}^n \hat{c}_i \geq \sum_{i=1}^n c_i$$



# 栈操作的代价指定

- 设 PUSH、POP 和 MULTIPOP 构成  $n$  个栈操作序列
- 栈操作的实际代价：
  - 1) PUSH: 1
  - 2) POP: 1
  - 3) MULTIPOP:  $\min(s, k)$
- 摊销代价：
  - 1) PUSH: 2
  - 2) POP: 0
  - 3) MULTIPOP: 0

虽然 MULTIPOP 操作的摊销代价为 0，其实际代价确是个变量！

# 栈操作序列的记帐方法分析

- 栈可类比于餐馆中桌面上的一叠盘子
- 执行一次 PUSH 操作，类似于新放置一个盘子，其摊销代价为 2 元，1 元支付 PUSH 的实际代价，1 元作为存款放在盘子上
- 任一时间点，每个盘子上都有 1 元钱作支付将来 POP 操作的实际代价
- 对 POP 操作来说，每个盘子上面都有存款用于支付弹出所需代价，不用另付费
- 对 MULTIPOP 操作也无须收费，因为其由多次 POP 操作完成，而 POP 操作无需支付额外费用

对任意包含  $n$  次 PUSH、POP 和 MULTIPOP 操作的序列，总的摊销代价就是其总的实际代价的一个上界。又因为总的摊销代价为  $O(n)$ ，故总的实际代价也为  $O(n)$

# 二进制计数器的记帐方法分析

对初值为 0 的  $n$  次 INCREMENT 操作:

- 设置位操作 ( $0 \rightarrow 1$ ) 的摊销代价为 2 美元, 其中 1 美元用于实际置位操作, 1 美元作为信用存储
- 复位操作 ( $1 \rightarrow 0$ ) 的摊销代价为 0 美元
- 所有复位操作都可以用信用支付
- 每个 INCREMENT 过程至多置位 1 次 (第 6 行)
- $n$  次 INCREMENT 操作的摊销代价为  $O(n)$ , 是总实际代价的上界



# 提纲

1 摊销分析简介

2 聚集方法

3 记帐方法

4 势能方法

5 动态表扩张



# 势能方法原理

- 势能法不将预付代价表示为数据结构中特定对象的费用，而是表示为“势能”，通过释放积累的势能支付未来操作的代价
- 势能是与整个数据结构而不是特定对象相关联的
- 势能法的工作方式：

若对一个初始数据结构  $D_0$  执行  $n$  个操作，令  $c_i$  为操作  $i$  的实际代价，令  $D_i$  为在数据结构  $D_{i-1}$  上执行第  $i$  个操作后得到的数据结构。势函数  $\Phi$  将每个数据结构  $D_i$  映射为一个实数  $\Phi(D_i)$ ，此即关联到数据结构  $D_i$  的势能

- 摊销代价  $\hat{c}_i$  定义为其实际代价加上此操作引起的势能变化：

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

- $n$  个操作的序列的总摊销代价为：

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





# 势能在摊销分析中的作用

- 由上页公式，如果能定义一个势函数  $\Phi$  使得  $\Phi(D_n) \geq \Phi(D_0)$ ，则总摊销代价  $\sum_{i=1}^n \hat{c}_i$  给出了总实际代价  $\sum_{i=1}^n c_i$  的一个上界
- 如果第  $i$  个操作的势差  $\Phi(D_i) - \Phi(D_{i-1}) > 0$ ，则摊销代价  $\hat{c}_i$  就表示对第  $i$  个操作多收了费，同时数据结构的势能也随之增加了
- 如果第  $i$  个操作的势差  $\Phi(D_i) - \Phi(D_{i-1}) < 0$ ，则摊销代价  $\hat{c}_i$  就表示对第  $i$  个操作的收费不足，这时就通过减少势能来支付该操作的实际代价
- 摊销代价依赖于所选择的势函数  $\Phi$ ，不同的势函数可能会产生不同的摊销代价，但它们都是实际代价的上界



# 栈操作的势能法分析 — 势函数定义

势能函数定义:  $\Phi(D) =$  栈 $D$ 中对象的个数

- 初始栈  $D_0$ ,  $\Phi(D_0) = 0$
- 栈中对象数始终非负, 第  $i$  个操作后满足  $\Phi(D_i) \geq 0 = \Phi(D_0)$
- 以  $\Phi$  表示的  $n$  个摊销代价的总和就表示了实际代际的一个上界



# 包含 $s$ 个对象的栈上的栈操作的摊销代价

- 第  $i$  个操作是个 PUSH 操作：
  - 实际代价:  $c_i = 1$
  - 势差:  $\Phi(D_i) - \Phi(D_{i-1}) = (s + 1) - s = 1$
  - 摊销代价:  $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$
- 第  $i$  个操作是个 MULTIPOP( $s, k$ ) 操作：
  - 实际代价:  $c_i = \min(s, k)$
  - 势差:  $\Phi(D_i) - \Phi(D_{i-1}) = -\min(s, k)$
  - 摊销代价:  $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = \min(s, k) - \min(s, k) = 0$
- 第  $i$  个操作是个 POP 操作：
  - 实际代价:  $c_i = 1$
  - 势差:  $\Phi(D_i) - \Phi(D_{i-1}) = -1$
  - 摊销代价:  $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 - 1 = 0$
- 摊销分析：
  - 每个栈操作摊销代价为  $O(1)$ ,  $n$  个操作的总摊销代价就是  $O(n)$
  - 因为  $\Phi(D_i) \geq \Phi(D_0)$ ,  $n$  个操作的总摊销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  $O(n)$



# 二进制计数器的势能法摊销分析

- 将势能定义为  $b_i$ ，即  $i$  次操作后计数器中 1 的个数
- 设第  $i$  个 INCREMENT 操作将  $t_i$  个位复位，则实际代价  $c_i \leq t_i + 1$ ，即除了复位  $t_i$  个位之外，至多还置位 1 次
- 势能计算：
  - 若  $b_i = 0$ ，表示第  $i$  个操作将所有  $k$  位都复位了，则  $b_{i-1} = t_i = k$
  - 若  $b_i > 0$ ，则  $b_i = b_{i-1} - t_i + 1$
  - 无论哪种情况，都有  $b_i \leq b_{i-1} - t_i + 1$
- 势差： $\Phi(D_i) - \Phi(D_{i-1}) \leq (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i$
- 摊销代价： $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq (t_i + 1) + (1 - t_i) = 2$
- 计数器从 0 开始， $\Phi(D_0) = 0$ ，任何时刻  $\Phi(D_i) \geq 0$ ，则  $n$  个 INCREMENT 操作的序列其总摊销代价是总实际代价的上界，最坏情况  $O(n)$

注意上述  $\hat{c}_i$  的分析并未限制计数器的初值为何，总有  $\hat{c}_i \leq 2$



# 从非 0 位置开始计数的二进制计数器

- 计数器初始时包含  $b_0$  个 1, 经  $n$  次操作后计数器中 1 的个数为  $b_n$ , 有  $0 \leq b_0, b_n \leq k$

- 总摊销代价:

$$\sum_{i=1}^n \hat{c}_i = \sum_{i=1}^n c_i + \Phi(D_n) - \Phi(D_0) = \sum_{i=1}^n c_i + b_n - b_0$$

- 总实际代价:

$$\sum_{i=1}^n c_i = \sum_{i=1}^n \hat{c}_i - b_n + b_0$$

- 从总实际代价式子可知: 由于  $\hat{c}_i \leq 2$ ,  $b_0 \leq k$ , 只要  $k = O(n)$ , 总实际代价就是  $O(n)$ , 不管计数器初值为何 (总实际代价越小越好, 所以无需关心  $\hat{c}_i$  是否过小导致总实际代价出现负数)



# 对摊销分析的进一步理解 \*

- 摊销分析中赋予数据结构对象的某些属性或费用仅仅是用来辅助分析，不需要也不应该出现在程序中
- 通过摊销分析，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
- 摊销分析不是平均情况分析，不能简单理解和分析  $n$  次操作序列的总计算量。它不是从宏观的整体操作上进行分析，虽然摊销分析确实也对“ $n$  次操作的总计算量”进行分析，但是它的方法却不是直接从  $n$  次操作的总体进行，而是对每一步单独操作进行细致的分析 (分析该操作引入的“摊销时间复杂度”)，然后再加总得出一个“ $n$  次操作的和的概念”
- 摊销分析，特别是“势函数”分析的作用，在于可以让真实复杂度高低不定、难以分析的操作 (比如教材中的动态表扩张的例子)，转变成稳定、易于分析的“摊销时间复杂度”

# 提纲

1 摊销分析简介

2 聚集方法

3 记帐方法

4 势能方法

5 动态表扩张



# 表扩张问题

- 给定一张存于内存空间的表格，如何根据其内存存储元素的数量调整表格的大小？
- 摊销分析用于证明：虽然插入和删除操作会引起表扩张和收缩，从而具有较高的实际代价，但这些操作的摊销代价都是  $O(1)$
- 动态表支持的操作包括 TABLE-INSERT 和 TABLE-DELETE
- 表中每个数据占用一个槽 (Slot)
- 表的实现方式无限制，堆、栈、数组、散列表都可以
- 装载因子  $\alpha(T)$ ：表中存储的数据项的数量与表规模（槽数）的比值
- 认为空表的规模为 0，因而其装载因子定为 1





# 表扩张算法

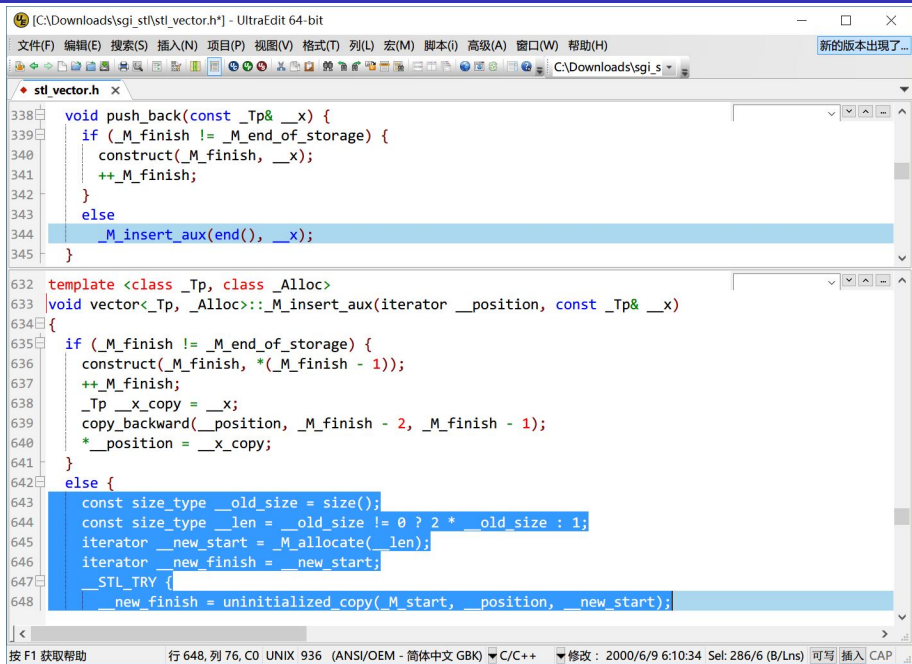
- 表扩张策略：每次扩张将表容量加倍，装载因子始终大于  $1/2$
- $T.table$ ：指向表实际存储空间的指针
- $T.num$ ：表中数据项的数量； $T.size$ ：表的规模；初始时  $T.num = T.size = 0$

TABLE-INSERT( $T, x$ )

```
1: if  $T.size == 0$  then  
2:   allocate  $T.table$  with 1 slot  
3:    $T.size = 1$   
4: if  $T.num == T.size$  then  
5:   allocate  $newtable$  with  $2 \cdot T.size$  slots  
6:   insert all items in  $T.table$  into  $newtable$   
7:   free  $T.table$   
8:    $T.table = newtable$   
9:    $T.size = 2 \cdot T.size$   
10: insert  $x$  into  $T.table$   
11:  $T.num = T.num + 1$ 
```



# 动态表扩张实例 — push\_back in SGI STL vector



```
[C:\Downloads\sgi_stl\stl_vector.h] - UltraEdit 64-bit
文件(F) 编辑(E) 搜索(S) 插入(N) 项目(P) 视图(V) 格式(T) 列(L) 宏(M) 脚本(i) 高级(A) 窗口(W) 帮助(H) 新的版本出现了...

stl_vector.h x
338 void push_back(const _Tp& __x) {
339     if (_M_finish != _M_end_of_storage) {
340         construct(_M_finish, __x);
341         ++_M_finish;
342     }
343     else
344         _M_insert_aux(end(), __x);
345 }

632 template <class _Tp, class _Alloc>
633 void vector<_Tp, _Alloc>::_M_insert_aux(iterator __position, const _Tp& __x)
634 {
635     if (_M_finish != _M_end_of_storage) {
636         construct(_M_finish, *(_M_finish - 1));
637         ++_M_finish;
638         _Tp __x_copy = __x;
639         copy_backward(__position, _M_finish - 2, _M_finish - 1);
640         *__position = __x_copy;
641     }
642     else {
643         const size_type __old_size = size();
644         const size_type __len = __old_size != 0 ? 2 * __old_size : 1;
645         iterator __new_start = _M_allocate(__len);
646         iterator __new_finish = __new_start;
647         _STL_TRY {
648             __new_finish = uninitialized_copy(_M_start, __position, __new_start);
649         }
```

按 F1 获取帮助 行 648, 列 76, C0 UNIX 936 (ANSI/OEM - 简体中文 GBK) C/C++ 修改: 2000/6/9 6:10:34 Sel: 286/6 (B/Lns) 可写 插入 CAP



# 表扩张操作的粗略分析

- 区分两个插入过程：TABLE-INSERT 自身和第 6/10 行的基本插入过程，认为基本插入的操作代价为 1
- 第 2 行分配初始表的代价认为是常量时间
- 第 5 和第 7 行分配、释放内存空间的开销由第 6 行数据复制代价决定
- 第 5–9 行称为执行了一次扩张操作
- 对空表执行  $n$  个 TABLE-INSERT 操作，第  $i$  个操作代价：
  - 当前表有空白空间： $c_i = 1$
  - 当前表满，需扩张：第 6 行数据项从旧表复制到新表代价  $i-1$ ，第 10 行插入代价 1，则  $c_i = i$
- 一个操作的最坏情况时间为  $O(n)$ ， $n$  个操作的总运行时间上界  $O(n^2)$



# 表扩张操作的聚集分析

- 扩张操作是很少的，只有在 2 的幂时才会引起扩张操作
- 第  $i$  个操作的代价：

$$c_i = \begin{cases} i, & i-1 = 2^k \\ 1, & i-1 \neq 2^k \end{cases}$$

- $n$  个 TABLE-INSERT 操作总代价：

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

- 单一操作的摊销代价至多为 3



# 表扩张操作的记帐方法分析

- 处理每个数据项要付出 3 次基本插入操作的代价：
  - 将其插入当前表中
  - 当表扩张时移动它
  - 当表扩张时移动另一个已经移动过的数据
- 假设一次扩张后表的规模为  $m$ ，则其中包含  $m/2$  个数据项，且当前信用为 0，则一次操作的 3 美元花费如下：
  - 1 美元支付当前插入操作
  - 1 美元存起来作为以后该项的移动费用
  - 同时为另外已在表中的  $m/2$  个数据中的一项存储 1 美元的移动费用
- 当表中已满，包含  $m$  个数据项时，每个数据项都存储了足够扩张的费用



# 表扩张操作的势能分析 1

- 定义势能函数：扩张后势能为 0，表满时正好是表的规模
- 势函数： $\Phi(T) = 2 \cdot T.num - T.size$
- 一次扩张后： $T.num = T.size/2$ ,  $\Phi(T) = 0$
- 扩张之前： $T.num = T.size$ ,  $\Phi(T) = T.num$
- 势  $\Phi(T)$  初值为 0，且始终非负， $n$  个操作的摊销代价之和是实际代价的上界

- 如第  $i$  个操作未触发表扩张， $size_i = size_{i-1}$ ，摊销代价为：

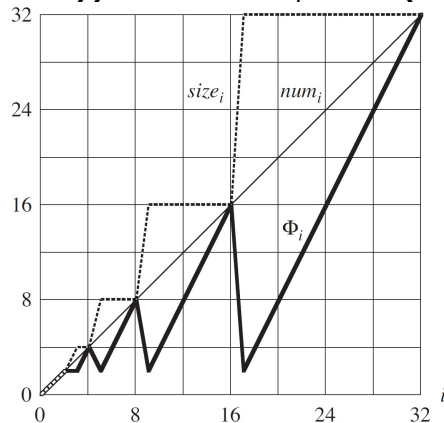
$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\ &= 1 + (2 \cdot num_i - size_i) - (2(num_i - 1) - size_i) \\ &= 3\end{aligned}$$

- 如果第  $i$  个操作触发了表扩张， $size_i = 2 \cdot size_{i-1}$ ,  $size_{i-1} = num_{i-1} = num_i - 1$



# 表扩张操作的势能分析 2

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



# 混合表扩张和收缩

- 装载因子太小时启动表收缩操作
- 装载因子有一个正的常数下界, 比如  $1/2$
- 装载因子低于  $1/2$  时启动表收缩操作的方案不可取:
  - 设  $n$  恰好是 2 的幂
  - 考虑前  $n/2$  个操作后,  $T.num = T.size = n/2$
  - 考虑如下接下来的  $n/2$  个操作: 插入、删除、删除、插入、插入、删除、删除、...
  - 对上述操作序列, 表的规模在  $n$  和  $n/2$  之间来回扩张和收缩, 每次代价  $\Theta(n)$ , 扩张和收缩次数  $\Theta(n)$ , 总代价  $\Theta(n^2)$ , 每个操作摊销代价  $\Theta(n)$
- 改进措施: 将装载因子下界设为  $1/4$





# 混合表扩张和收缩的势函数设计

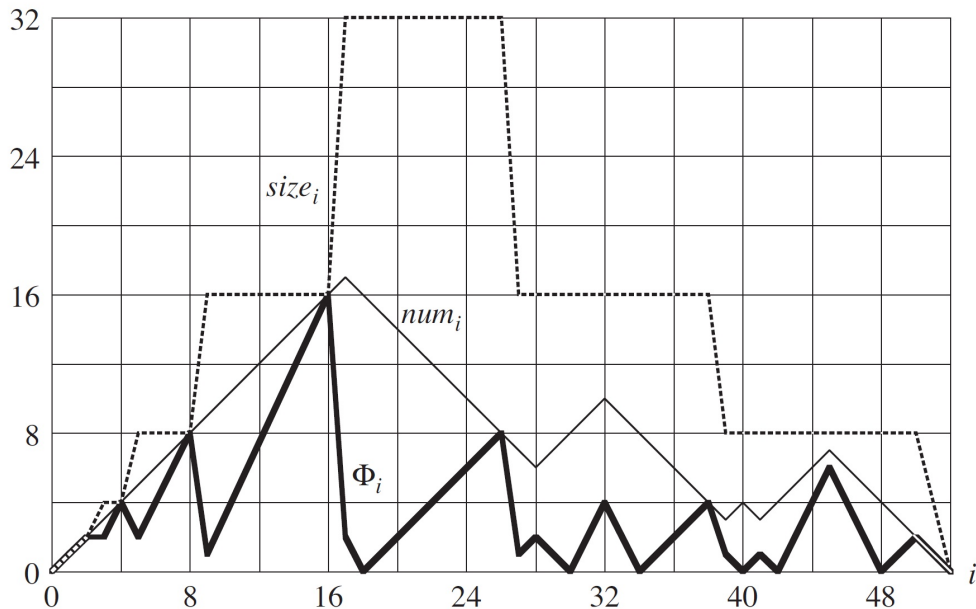
- 令装载因子  $1/2$  为中心点，其势能为  $0$
- 装载因子偏移  $1/2$  时势能增加，使得到真正扩张和收缩时已积累了足够的势能支付扩张和收缩的费用
- 装载因子为  $1$  或  $1/4$  时，势函数值增至  $T.num$
- 扩张和收缩后，装载因子变回  $1/2$ ，势能为  $0$
- 设空表  $T.num = T.size = 0$ ， $\alpha(T) = 1$
- 势函数：

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

- 可知空表势为  $0$ ，且势永远非负，总摊销代价可作为总实际代价上界



# 混合表扩张和收缩的势变化



# 混合表扩张和收缩的势能分析

- 令  $c_i$  和  $\hat{c}_i$  分别表示第  $i$  个操作的实际代价和摊销代价
- $num_i$  表示第  $i$  个操作后的数据项数量； $size_i$  表示第  $i$  个操作后表的规模
- $\alpha_i$  为第  $i$  个操作后的装载因子， $\Phi_i$  表示第  $i$  个操作后的势能
- 初始情况： $num_0 = 0$ ,  $size_0 = 0$ ,  $\alpha_0 = 1$ ,  $\Phi_0 = 0$
- 第  $i$  个操作为 TABLE-INSERT:
  - 1)  $\alpha_{i-1} \geq 1/2$
  - 2)  $\alpha_{i-1} < 1/2$  且  $\alpha_i < 1/2$
  - 3)  $\alpha_{i-1} < 1/2$  且  $\alpha_i \geq 1/2$
- 第  $i$  个操作为 TABLE-DELETE:
  - 1)  $\alpha_{i-1} \geq 1/2$  且  $\alpha_i \geq 1/2$
  - 2)  $\alpha_{i-1} \geq 1/2$  且  $\alpha_i < 1/2$
  - 3)  $\alpha_{i-1} < 1/2$  且未引起表收缩
  - 4)  $\alpha_{i-1} < 1/2$  且触发表收缩



# 混合表扩张和收缩的势能分析 TABLE-INSERT 1

- $\alpha_{i-1} \geq 1/2$ : 与单纯表扩张的分析相同, 摊销代价至多为 3
- $\alpha_{i-1} < 1/2$  且  $\alpha_i < 1/2$ :

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (\text{size}_i/2 - \text{num}_i) - (\text{size}_{i-1}/2 - \text{num}_{i-1}) \\ &= 1 + (\text{size}_i/2 - \text{num}_i) - (\text{size}_i/2 - (\text{num}_i - 1)) \\ &= 0\end{aligned}$$



## 混合表扩张和收缩的势能分析 TABLE-INSERT 2

- $\alpha_{i-1} < 1/2$  且  $\alpha_i \geq 1/2$ :

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot \text{num}_i - \text{size}_i) - (\text{size}_{i-1}/2 - \text{num}_{i-1}) \\ &= 1 + (2 \cdot (\text{num}_{i-1} + 1) - \text{size}_{i-1}) - \left(\frac{1}{2}\text{size}_{i-1} - \text{num}_{i-1}\right) \\ &= 3 \cdot \text{num}_{i-1} - \frac{3}{2}\text{size}_{i-1} + 3 \\ &= 3\alpha_{i-1}\text{size}_{i-1} - \frac{3}{2}\text{size}_{i-1} + 3 \\ &< \frac{3}{2}\text{size}_{i-1} - \frac{3}{2}\text{size}_{i-1} + 3 = 3\end{aligned}$$



# 混合表扩张和收缩的势能分析 TABLE-DELETE 1

- $\alpha_{i-1} \geq 1/2$  且  $\alpha_i \geq 1/2$ :

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot \text{num}_i - \text{size}_i) - (2 \cdot \text{num}_{i-1} - \text{size}_{i-1}) \\ &= 1 + (2 \cdot (\text{num}_{i-1} - 1) - \text{size}_{i-1}) - (2 \cdot \text{num}_{i-1} - \text{size}_{i-1}) \\ &= -1\end{aligned}$$

- $\alpha_{i-1} \geq 1/2$  且  $\alpha_i < 1/2$ :

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (\text{size}_i/2 - \text{num}_i) - (2 \cdot \text{num}_{i-1} - \text{size}_{i-1}) \\ &= 1 + (\text{size}_{i-1}/2 - (\text{num}_{i-1} - 1)) - (2 \cdot \text{num}_{i-1} - \text{size}_{i-1}) \\ &= 2 + \frac{3}{2}\text{size}_{i-1} - 3 \cdot \text{num}_{i-1} = 2 + \frac{3}{2}\text{size}_{i-1} - 3\alpha_{i-1}\text{size}_{i-1} \\ &\leq 2 + \frac{3}{2}\text{size}_{i-1} - \frac{3}{2}\text{size}_{i-1} = 2\end{aligned}$$



## 混合表扩张和收缩的势能分析 TABLE-DELETE 2

- $\alpha_{i-1} < 1/2$  且未引起表收缩:

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (\text{size}_i/2 - \text{num}_i) - (\text{size}_{i-1}/2 - \text{num}_{i-1}) \\ &= 1 + (\text{size}_{i-1}/2 - (\text{num}_{i-1} - 1)) - (\text{size}_{i-1}/2 - \text{num}_{i-1}) \\ &= 2\end{aligned}$$

- $\alpha_{i-1} < 1/2$  且触发表收缩: 操作实际代价  $c_i = \text{num}_i + 1$ ,  $\text{size}_i/2 = \text{size}_{i-1}/4 = \text{num}_{i-1} = \text{num}_i + 1$ , 则

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= (\text{num}_i + 1) + (\text{size}_i/2 - \text{num}_i) - (\text{size}_{i-1}/2 - \text{num}_{i-1}) \\ &= (\text{num}_i + 1) + (\text{num}_i + 1 - \text{num}_i) - (2 \cdot (\text{num}_i + 1) \\ &\quad - (\text{num}_i + 1)) = 1\end{aligned}$$

无论哪种情况, 摊销代价的上界都是一个常数, 因而执行任意  $n$  个操作的实际运行时间  $O(n)$ !

# 小结

本节课重点：

- 摊销分析背后的思想
- 掌握三种摊销分析的方法
- 通过表扩张的分析掌握势能方法（重点）

