



HITWH
SE

第六章

Amortized Analysis



- 6.1 平摊分析基本原理
- 6.2 聚集方法
- 6.3 会计方法
- 6.4 势能方法
- 6.5 动态表平摊分析
- 6.6 并差集平摊分析



HITWH
SE

参考资料



Chapter 17 Amortized Analysis

Pages 405 - 430



6.1 平摊分析基本原理

- Amortized Analysis 的基本思想
- Amortized Analysis 方法



关注一系列**数据结构上**操作的时间复杂度：

考虑操作序列： OP_1, OP_2, \dots, OP_m

想确定这个操作序列可能花费的最长时间

一个可能的想法：考察每种操作 OP_i 的最坏情况时间复杂度 t_i ，

由此，上述操作序列可能的最长时间就是：

每种操作 OP_i 的最坏时间 t_i 之和

不一定正确！

操作之间实际上是相互关联的，

不能假设它们是相互独立的！



例如：

- 普通栈操作及其时间代价
 - $\text{Push}(S, x)$: 将对象 x 压入栈 S
 - $\text{Pop}(S)$: 弹出并返回 S 的顶端元素
 - 两个操作的运行时间都是 $O(1)$
 - 可把每个操作的实际代价视为 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 $\theta(n)$



- 新的栈操作及其时间代价

- Multipop(S, k):

- 去掉 S 的 k 个栈顶对象, 当 $|S| < k$ 时弹出整个栈

- 实现算法

- Multipop(S, k)

- 1 While not STACK-EMPTY(S) and $k \neq 0$ Do

- 2 Pop(S);

- 3 $k \leftarrow k - 1$.

- Multipop(S, k)的实际代价 (设Pop的代价为1)

- Multipop的代价为 $\min(|S|, k)$



- 初始栈为空的 n 个栈操作序列的分析
 - n 个栈操作序列由Push、Pop和Multipop组成
 - 粗略分析
 - 最坏情况下，每个操作都是Multipop
 - 栈的大小至多为 n
 - 每个Multipop的代价最坏是 $O(n)$
 - 操作系列的最坏代价为 $T(n) = O(n^2)$

分析过于粗糙，不够紧确！

原因：只关注于操作，忽略了数据结构！



Amortized Analysis的基本思想

平摊分析目的是分析给定
数据结构上的 n 个操作代价上界

各个操作的平均代价？
整个序列的代价是多少？

对一个数据结构
执行一个操作序列：

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

平摊
代价

不涉及概率
异于平均分析



Amortized Analysis的基本思想



- Aggregate Analysis方法（每个操作的代价）
 - 确定 n 个操作的上界 $T(n)$, 每个操作平摊 $T(n)/n$
- Accounting方法（整个操作序列的代价）
 - 不同类型操作赋予不同的平摊代价
 - 某些操作在数据结构的特殊对象上“预付”代价
- Potential方法（整个操作序列的代价）
 - 不同类型操作赋予不同的平摊代价
 - “预付”的代价作为整个数据结构的“能量”



6.2 Aggregate Analysis

- 聚集方法的原理
- 聚集方法的实例之一
- 聚集方法的实例之二



- 普通栈操作及其时间代价
 - $\text{Push}(S, x)$: 将对象 x 压入栈 S
 - $\text{Pop}(S)$: 弹出并返回 S 的顶端元素
 - 两个操作的运行时间都是 $O(1)$
 - 可把每个操作的实际代价视为 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 $\theta(n)$



聚集方法的原理

目的是分析 n 个操作系列中
每个操作的复杂性上界

数据结构上共有 n 个
操作, 最坏情况下:

操作1: $\text{Cost}=t_1$

操作2: $\text{Cost}=t_2$

⋮

⋮

⋮

操作 n : $\text{Cost}=t_n$

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

平摊代价:
 $T(n)/n$

每个操作被赋予相同
代价, 不管操作类型



- 初始栈为空的 n 个栈操作序列的分析
 - n 个栈操作序列由Push、Pop和Multipop组成
 - 粗略分析

- 最坏情况下，每个操作都是Multipop
- 每个Multipop的代价最坏是 $O(n)$
- 操作系列的最坏代价为 $T(n) = O(n^2)$
- 平摊代价为 $T(n)/n = O(n)$

分析太粗糙
!!!

- 精细分析

- 一个对象在每次被压入栈后至多被弹出一次
- 在非空栈上调用Pop的次数(包括在Multipop内的调用)至多为Push执行的次数, 即至多为 n
- 最坏情况下操作序列的代价为 $T(n) \leq 2n = O(n)$
- 平摊代价= $T(n)/n = O(1)$

$n-1$ 个push
1个multipop



聚集方法实例之二： 二进制计数器

- 问题定义： 由0开始计数的 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： $x+1 \bmod 2^k$

数据结构：

$A[0..k-1]$ 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

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



- 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 $x+1 \bmod 2^k$

INCREMENT(A)

1 $i \leftarrow 0$

2 while $i < k$ and $A[i] = 1$ Do

3 $A[i] \leftarrow 0;$

4 $i \leftarrow i + 1;$

5 If $i < \text{length}[A]$ Then $A[i] \leftarrow 1$

- 初始为零的计数器上 n 个INCREMENT操作分析

Counter Value									每个操作 Cost
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	
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	2
3	0	0	0	0	0	0	1	1	1
4	0	0	0	0	0	1	0	0	3
5	0	0	0	0	0	1	0	1	1
6	0	0	0	0	0	1	1	0	2
7	0	0	0	0	0	1	1	1	1
8	0	0	0	0	1	0	0	0	4
9	0	0	0	0	1	0	0	1	1
10	0	0	0	0	1	0	1	0	2
11	0	0	0	0	1	0	1	1	1
12	0	0	0	0	1	1	0	0	3
13	0	0	0	0	1	1	0	1	1
14	0	0	0	0	1	1	1	0	2
15	0	0	0	0	1	1	1	1	1
16	0	0	0	1	0	0	0	0	5



- 粗略分析

- 每个Increment的时间代价最多 $O(k)$
- n 个Increment序列的时间代价最多 $O(kn)$
- n 个Increment平摊代价为 $O(k)$
- 例如上例中: $k*n=8*16=128$

- 精细分析



操作Cost = $O(\text{发生改变的位数})$

Counter Value										Total Cost
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
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



• 精细分析

- $A[0]$ 每次操作发生一次改变, 总次数为 n
- $A[1]$ 每两次操作发生一次改变, 总次数为 $n/2$
- $A[2]$ 每四次操作发生一次改变, 总次数为 $n/4$
- 一般地
 - 对于 $i=0, 1, \dots, \lg n$, $A[i]$ 改变次数为 $n/2^i$
 - 对于 $i > \lg n$, $A[i]$ 不发生改变
(因为 n 个操作结果为 n , 仅涉及 $A[0]$ 至 $A[\lg n]$ 位)
- $T(n) = \sum_{0 \leq i \leq \lg n} n/2^i < n \sum_{0 \leq i \leq \infty} 1/2^i = O(n)$
- 每个 Increment 操作的平摊代价为 $O(1)$



6.3 The Accounting Method

- Accounting方法的原理
- Accounting方法的实例之一
- Accoutning方法的实例之二



- Accounting方法

- 目的是分析 n 个操作序列的复杂性上界，一个操作序列中有不同类型的操作，不同类型操作的代价各不相同
- 于是我们为每种操作分配不同的平摊代价
 - 平摊代价可能比实际代价大，也可能比实际代价小
 - 如果平摊代价比实际代价高：一部分用于支付实际代价，多余部分作为Credit附加在数据结构的具体数据对象上
 - 当一个操作的平摊代价比实际代价低时：Credit用来补充支付实际代价
- 平摊代价的选择规则：
 - 设 α_i 和 c_i 是操作 i 的平摊代价和实际代价

数据结构中存储的Credit在任何时候都必须非负，

$$\text{即 } \sum_{1 \leq i \leq n} \alpha_i - \sum_{1 \leq i \leq n} c_i \geq 0 \text{ 永远成立}$$



栈操作序列的分析

- 各栈操作的实际代价
 - $\text{Cost}(\text{PUSH})=1$
 - $\text{Cost}(\text{POP})=1$
 - $\text{Cost}(\text{MULTIPOP})=\min\{k, s\}$
- 各栈操作的平摊代价
 - $\text{Cost}(\text{PUSH})=2$
 - 一个1用来支付PUSH的开销,
 - 另一个1存储在压入栈的元素上,预支POP的开销
 - $\text{Cost}(\text{POP})=0$
 - $\text{Cost}(\text{MULTIPOP})=0$
- 平摊代价满足
 - $\sum_{1 \leq i \leq n} \alpha_i - \sum_{1 \leq i \leq n} c_i \geq 0$ 对于任意 n 个操作序列都成立
 - 因为栈中的元素个数 ≥ 0
- n 个栈操作序列的总平摊代价
 - $O(n)$



二进制计数器Increment操作序列分析

- Increment操作的平摊代价
 - 每次一位被置1时，付2美元
 - 1美元用于置1的开销
 - 1美元存储在該“1”位上，用于支付其被置0时的开销
 - 置0操作无需再付款
 - $\text{Cost}(\text{Increment})=2$
- 平摊代价满足
 - $\sum_{1 \leq i \leq n} \alpha_i \geq \sum_{1 \leq i \leq n} c_i$ 对于任意 n 个操作序列都成立
 - 即：任何时刻，计数器上的Credit非负
- n 个Increment操作序列的总平摊代价
 - $O(n)$

- 初始为零的计数器上 n 个INCREMENT操作分析

Counter Value										每个操作 Cost	
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			
1	0	0	0	0	0	0	0	1	1	1	
2	0	0	0	0	0	0	1	0	0	0	2
3	0	0	0	0	0	0	1	1	1	1	1
4	0	0	0	0	0	1	0	0	0	0	3
5	0	0	0	0	0	1	1	0	0	1	1
6	0	0	0	0	0	1	1	1	0	0	2
7	0	0	0	0	0	1	1	1	1	1	1
8	0	0	0	0	1	0	0	0	0	0	4
9	0	0	0	0	1	1	0	0	0	1	1
10	0	0	0	0	1	1	0	1	1	0	2
11	0	0	0	0	1	1	0	1	1	1	1
12	0	0	0	0	1	1	1	0	0	0	3
13	0	0	0	0	1	1	1	0	0	1	1
14	0	0	0	0	1	1	1	1	1	0	2
15	0	0	0	0	1	1	1	1	1	1	1
16	0	0	0	1	0	0	0	0	0	0	5



6.4 The Potential Method

- Potential方法的原理
- Potential方法的实例之一
- Potential方法的实例之二



- Potential方法

- 目的是分析 n 个操作系列的复杂性上界
- 在会计方法中，如果操作的平摊代价比实际代价大，我们将余额与数据结构的数据对象相关联
- Potential方法把Credit余额与整个数据结构关联，所有这样的余额之和，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



• 数据结构势能的定义

— 考虑在初始数据结构 D_0 上执行 n 个操作

— 对于操作 i

- 操作 i 的实际代价为 c_i
- 操作 i 将数据结构 D_{i-1} 变为 D_i
- 数据结构 D_i 的势能是一个实数 $\phi(D_i)$, ϕ 是一个正函数
- 操作 i 的平摊代价: $\alpha_i = c_i + \phi(D_i) - \phi(D_{i-1})$

— n 个操作的总平摊代价

$$\begin{aligned}\sum_{i=1}^n \alpha_i &= \sum_{i=1}^n (c_i + \phi(D_i) - \phi(D_{i-1})) \\ &= \sum_{i=1}^n c_i + \phi(D_n) - \phi(D_0)\end{aligned}$$

总平摊代价必须是
总实际代价的上界

$$\sum_{i=1}^n \alpha_i \geq \sum_{i=1}^n c_i$$

— 关键是 ϕ 的定义

- 保证 $\phi(D_n) \geq \phi(D_0)$, 使总平摊代价是总实际代价的上界
- 如果对于所有 i , $\phi(D_i) \geq \phi(D_0)$, 可以保证 $\phi(D_n) \geq \phi(D_0)$
- 实际可以定义 $\phi(D_0) = 0$, $\phi(D_i) \geq 0$ (由具体问题确定)



栈操作序列的分析

- 栈的势能定义

- $\phi(D_m)$ 定义为栈 D_m 中对象的个数，于是

- $\phi(D_0) = 0$, D_0 是空栈
- $\phi(D_i) \geq 0 = \phi(D_0)$, 因为栈中对象个数不会小于 0
- n 个操作的总平摊代价是实际代价的上界

- 栈操作的平摊代价 (设栈 D_{i-1} 中具有 s 个对象)

- PUSH: $\alpha_i = c_i + \phi(D_i) - \phi(D_{i-1}) = 1 + (s+1) - s = 2$
- POP: $\alpha_i = c_i + \phi(D_i) - \phi(D_{i-1}) = 1 + (s-1) - s = 0$
- MULTIPOP(S, k): 设 $k' = \min(k, s)$

$$\alpha_i = c_i + \phi(D_i) - \phi(D_{i-1}) = k' + (s - k') - s = k' - k' = 0$$

- n 个栈操作序列的平摊代价是 $O(n)$



二进制计数器操作序列分析

- 计数器的势能定义

- $\phi(D_m)$ 定义为第 m 个操作后计数器中 1 的个数 b_m

- $\phi(D_0) = 0$, 0 的计数中不包含 “1”
 - $\phi(D_i) \geq 0 = \phi(D_0)$, 因为计数器中 1 的个数不会小于 0
 - 于是, n 个操作的总平摊代价是实际代价的上界

- INCREMENT 操作的平摊代价

- 第 i 个 INCREMENT 操作把 t_i 个位置成 0, 实际代价为 $c_i \leq t_i + 1$

- 计算操作 i 的平摊代价 $\alpha_i = c_i + \phi(D_i) - \phi(D_{i-1})$

- If $b_i = 0$, 操作 i 把所有 k 位置 0, 所以 $b_{i-1} = k$, $t_i = k$

- If $b_i > 0$, 则 $b_i = b_{i-1} - t_i + 1$

- 于是 $b_i \leq b_{i-1} - t_i + 1$

$$\phi(D_i) - \phi(D_{i-1}) = b_i - b_{i-1} \leq b_{i-1} - t_i + 1 - b_{i-1} = 1 - t_i$$

- 平摊代价 $\alpha_i = c_i + \phi(D_i) - \phi(D_{i-1}) \leq (t_i + 1) + (1 - t_i) = 2$

- n 个操作序列的总平摊代价是 $O(n)$



6.5 动态表性能平摊分析

- 动态表的概念
- 动态表的扩张与收缩
- 仅含扩张操作的动态表平摊分析
- 一般的动态表平摊分析



- 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
- 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 $\alpha(T)$
 - $\alpha(T) = T$ 存储的对象数/表大小
 - 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
 - 空表的大小为0, 装载因子为1



- 虽然插入和删除操作可能会引起表的扩张和收缩，从而具有较高的实际代价
- 但是，利用平摊分析能够证明，插入和删除操作的平摊代价为 $O(1)$
- 同时保证动态表中未用的空间始终不超过整个空间的一部分。



设 T 表示一个动态表:

- $table[T]$ 是一个指向表示表的存储块的指针
- $num[T]$ 表 T 中的数据项个数
- $size[T]$ 是 T 的大小
- 开始时, $num[T]=size[T]=0$



算法: TABLE—INSERT(T, x)

1 If $\text{size}[T]=0$ Then

2 获取一个大小为1的表 $\text{table}[T]$;

3 $\text{size}[T] \leftarrow 1$;

4 If $\text{num}[T]=\text{size}[T]$ Then

5 获取一个大小为 $2 \times \text{size}[T]$ 的新表 new-table;

6 将 $\text{table}[T]$ 中元素插入 new-table; /*简单插入操作*/

7 释放 $\text{table}[T]$;

8 $\text{table}[T] \leftarrow \text{new-table}$;

9 $\text{size}[T] \leftarrow 2 \times \text{size}[T]$;

10 将 x 插入 $\text{table}[T]$;

11 $\text{num}[T] \leftarrow \text{num}[T] + 1$

/*复杂的插入操作*/

/*开销为常数*/

/*开销取决于 $\text{size}[T]$ */



- 插入一个数组元素时,完成的操作包括
 - 分配一个比原表包含更多的槽的新表
 - 再将原表中的各项复制到新表中去
- 常用的启发式技术是分配一个比原表大一倍的新表,
 - 只对表执行插入操作,则表的装载因子总是至少为 $1/2$
 - 浪费掉的空间就始终不会超过表总空间的一半



初始为空的表上 n 次插入操作的代价分析

聚集分析-粗略分析

— 考察第 i 次操作的代价 C_i

- 如果 $i=1$, $C_i=1$;
- 如果 $\text{num}[T] < \text{size}[T]$, $C_i=1$;
- 如果 $\text{num}[T] = \text{size}[T]$, $C_i=i$;

— 共有 n 次操作

- 最坏情况下,每次都按照扩张操作计量,总的代价上界为 n^2

— 这个界不精确

- n 次TABLE—INSERT操作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

聚集分析-精细分析

— 第 i 次操作的代价 C_i

- 如果 $i-1=2^m$, $C_i=i$; 否则 $C_i=1$

— n 次TABLE—INSERT操作的总代价为 $\sum_{i=1}^n c_i \leq n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j < n + 2n = 3n$

★ 每一操作的平摊代价为 $3n/n=3$



会计方法

- 每次执行TABLE—INSERT平摊代价为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，数据复制的代价由数据上的存款来支付
- 任何时候，存款总和 非负
- 初始为空的表上 n 次TABLE-INSERT操作的平摊代价总和为 $3n$



势能法分析

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

- 刚扩充完, $\phi(T)=0$
- 表满时 $\phi(T)=size(T)$

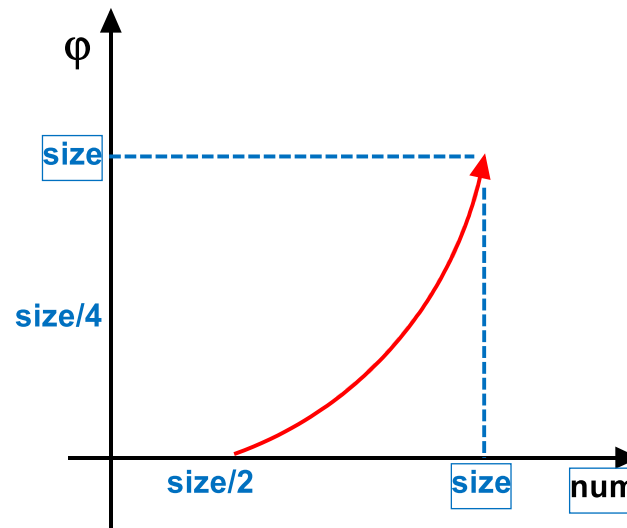
- 定义 $\phi(T)=2*num[T]-size[T]$

- 由于 $num[T] \geq size[T]/2$, 故 $\phi(T) \geq 0$
- 因此, n 次TABLE-INSERT操作的总的平摊代价是总的实际代价的一个上界

- 第 i 次操作的平摊代价

- 如果未发生扩张, $\alpha_i=3$
- 如果发生扩张, $\alpha_i=3$

Why???



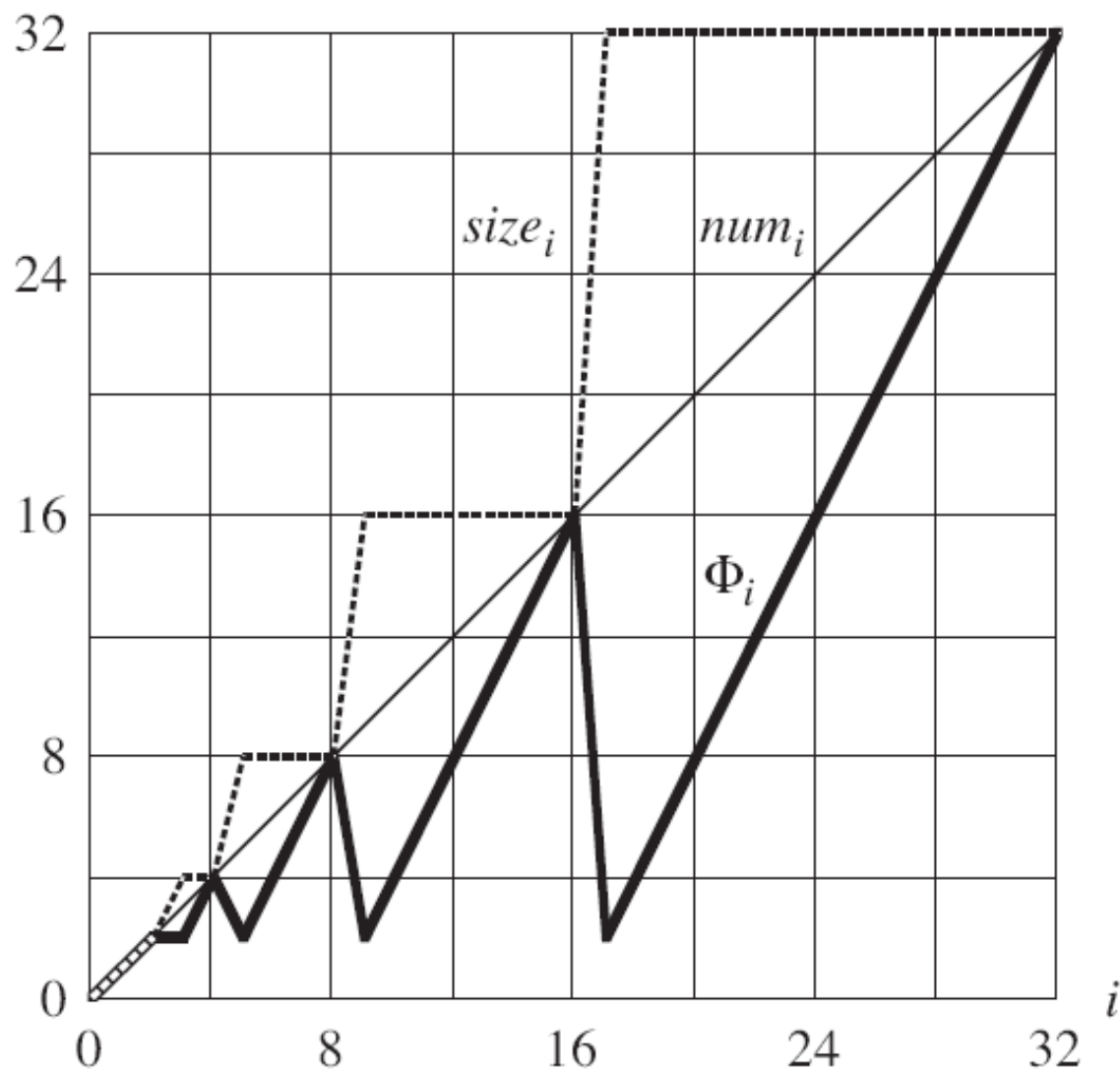
- 初始为空的表上 n 次插入操作的代价的上界为 $3n$



HITWH
SE

初始为空的表上 n 次插入操作的代价分析

32 Table-Insertions





- Table-Delete : 将指定的数据对象从表中删除
- 表的收缩: 当动态表的装载因子很小时, 对表进行收缩
理想情况下, 我们希望:
 - 表具有一定的丰满度
 - 表的操作序列的复杂度是线性的
- 表的收缩策略
 - 表的装载因子小于 $1/2$ 时, 收缩表为原表的一半
 - $n = 2^k$, 考察下面的一个长度为 n 的操作序列:
 - 前 $n/2$ 个操作是插入, 后跟IDDIIDDII...
 - 每次扩张和收缩的代价为 $O(n)$, 共有 $O(n)$ 扩张或收缩
 - 总代价为 $O(n^2)$, 而每一次操作的平摊代价为 $O(n)$ --每个操作的平摊代价太高



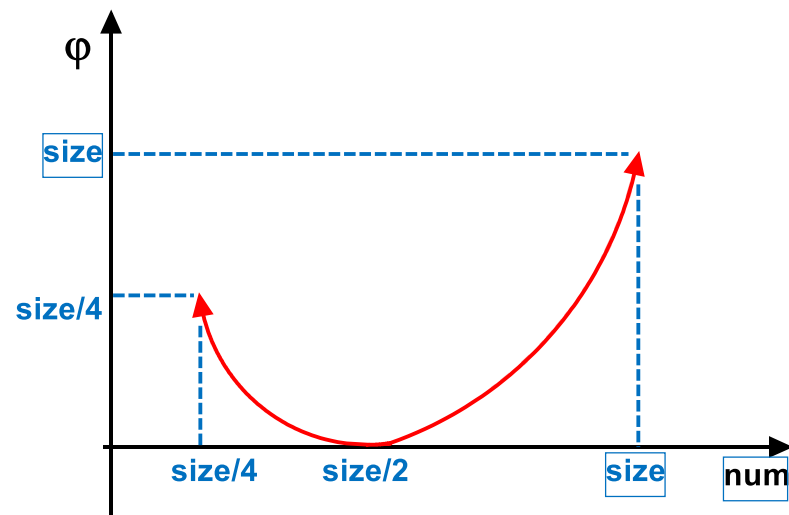
- 改进的收缩策略(允许装载因子低于 $1/2$)
 - 满表中插入数据项时, 将表扩大一倍
 - 删除数据项引起表不足 $1/4$ 满时, 将表缩小为原表的一半
 - 扩张和收缩过程都使得表的装载因子变为 $1/2$
 - 表的装载因子的下界是 $1/4$



动态表上 n 次(插入、删除)操作的代价分析

势能函数的定义

- 操作序列过程, 势能总是非负的
 - 保证一系列操作的总平摊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
- 势能函数应满足
 - $num(T)=size(T)/2$ 时, 势最小
 - 当 $num(T)$ 减小时, 势增加直到收缩
 - 当 $num(T)$ 增加时, 势增加直到扩充
- 势能函数特征的细化
 - 当装载因子为 $1/2$ 时, 势为0
 - 装载因子为1时, 有 $num[T]=size[T]$, 即 $\phi(T)=num[T]$ 。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, $size[T]=4 \cdot num[T]$ 。即 $\phi(T)=num[T]$ 。因而当删除某项引起一次收缩时就可利用势来支付其代价



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

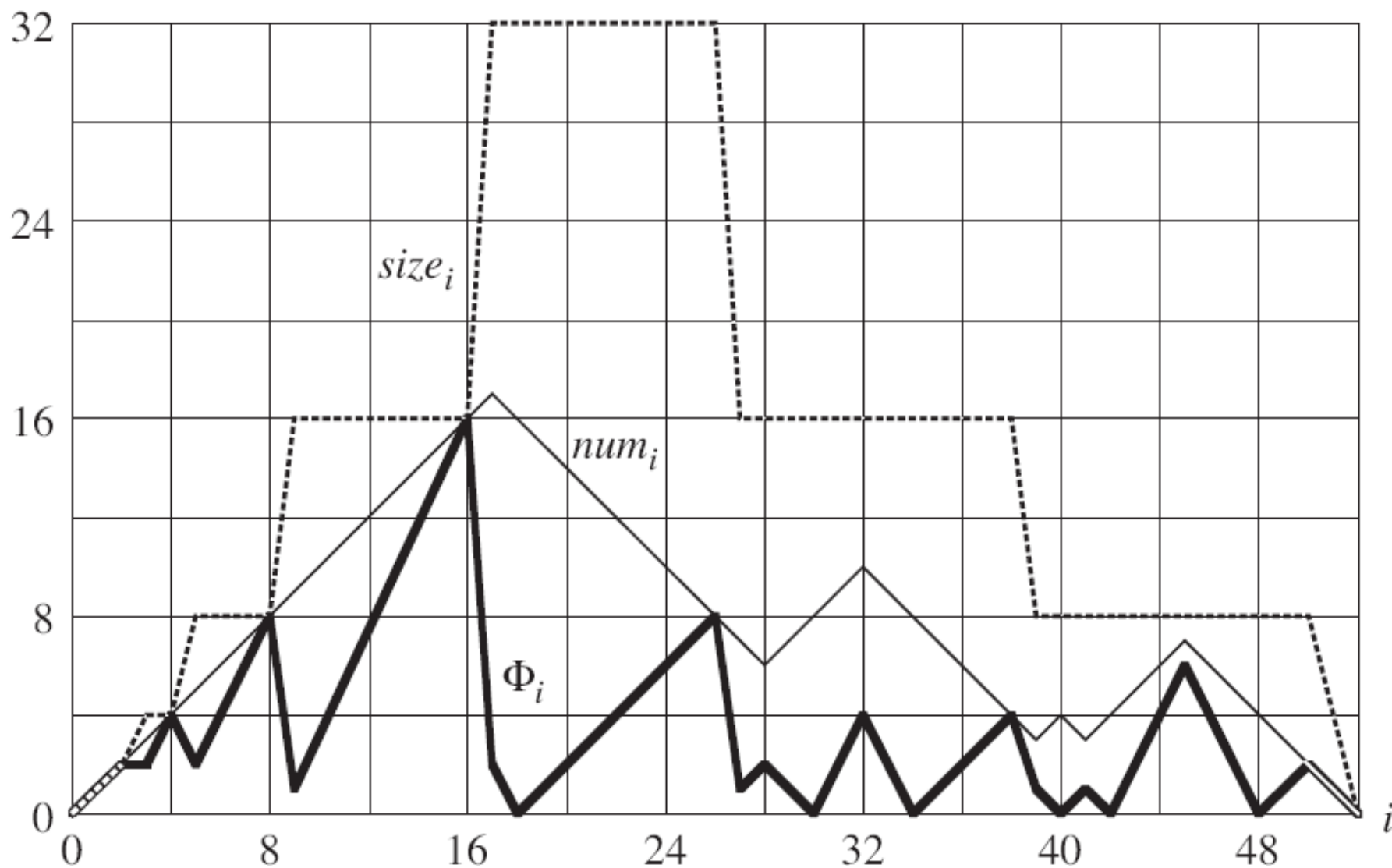


- 第 i 次操作的平摊代价: $\alpha_i = c_i + \phi(T_i) - \phi(T_{i-1})$
 - 第 i 次操作是TABLE—INSERT: 未扩张 $\alpha_i \leq 3$
 - 第 i 次操作是TABLE—INSERT: 扩张 $\alpha_i \leq 3$
 - 第 i 次操作是TABLE—DELETE: 未收缩 $\alpha_i \leq 3$
 - 第 i 次操作是TABLE—DELETE: 收缩 $\alpha_i \leq 3$
- 动态表上的 n 个操作的实际时间为 $O(n)$



初始为空的表上 n 次插入操作的代价分析

52次操作：Table-Insertion与Table-Deletion混合





6.6 并查集性能平摊分析

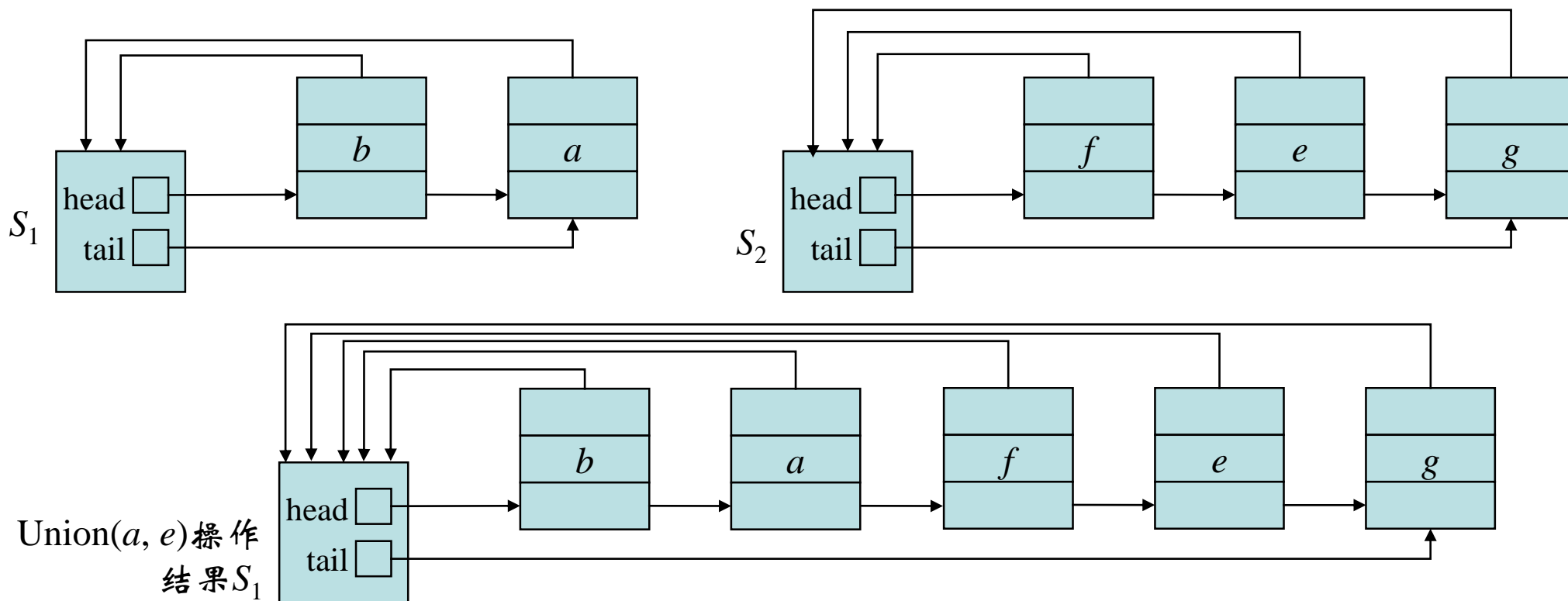
- 并查集的概念和基本操作
- 并查集的线性链表实现
- 并查集的森林实现
- 并查集性能的性能平摊分析

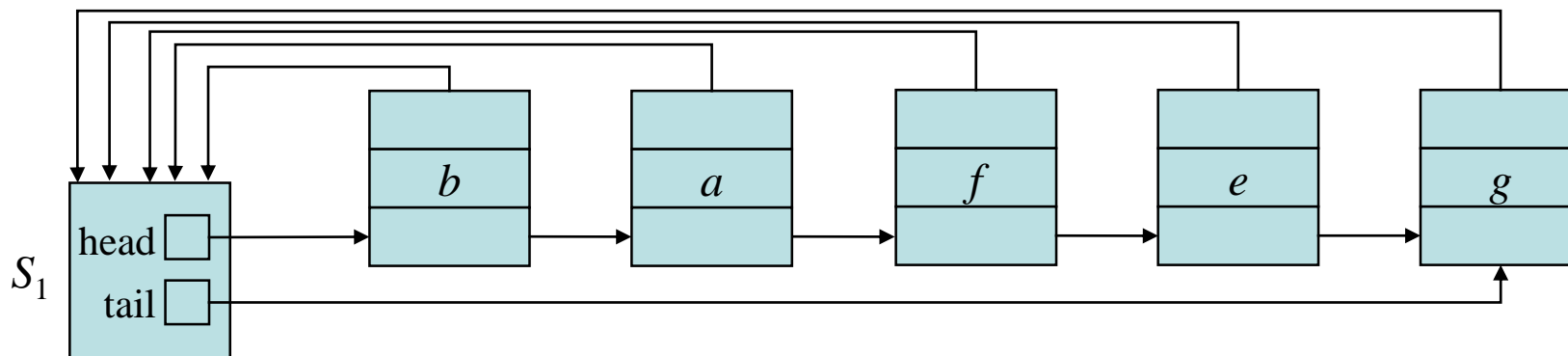


- 很多应用需要对集合进行存储、并对之查询等
 - 元素插入集合
 - 查找给定元素所在的集合
 - 集合合并等
- 若所处理集合为不相交集合，则可利用并查集，使得在并查集上执行 n 次集合操作的代价几乎总为 $O(n)$
- 目的：管理 n 个不相交的集合 $C=\{S_1, \dots, S_n\}$
 - 每个集合 S_i 有一个代表元素 x_i
- - MAKE-SET(x): 创建仅含元素 x 的集合.
 - UNION(x, y) : 合并包含元素 x 和 y 的集合
 - FIND-SET(x) : 返回 x 所在集合的代表元素



- 实现方法1：可以将集合存储为一个链表
 - 集合插入或合并操作的代价为 $O(1)$
 - 每次查找某一元素所在集合的操作代价为 $O(n)$ ，则执行 n 次集合操作的代价为 $O(n^2)$
- 实现方法2：添加指向Head的指针

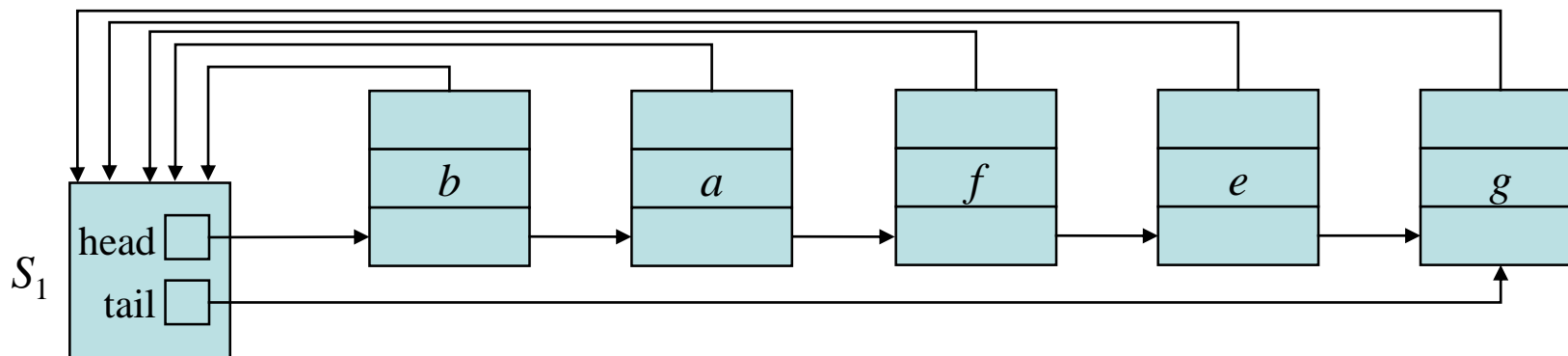




• 性能：

- MAKE-SET(x): $O(1)$
- FIND-SET(x): $O(1)$
- UNION(x, y): $O(n)$ 更新指向Head的指针的代价为 $O(n)$

- m 个MAKE-SET, UNION和FIND-SET操作，其中 n 个操作是MAKE-SET
- 总体代价为 $\theta(m+n^2)$



- 改进前的性能：

- $\text{UNION}(x, y): O(n)$ 更新指向Head的指针的代价为 $O(n)$

- 改进策略：

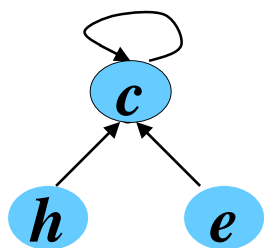
- 更新短链表中指向Head的指针

- m 个MAKE-SET, UNION和FIND-SET操作，其中 n 个操作是MAKE-SET

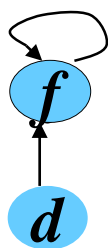
- 改进后总体代价为 $O(m+n\lg n)$



- 目的：管理 n 个不相交的集合 $C=\{S_1, \dots, S_n\}$
 - 每个集合 S_i 有一个代表元素 x_i
 - 将每个集合组织为一个有根树，树的每个节点对应集合中的一个元素
 - 每个节点 y 维护一个指针，指向其父节点 $p(y)$
 - 根节点中存储的元素为集合的代表元素，其指针指向自身



Set $\{c, h, e\}$



Set $\{f, d\}$



- 支持的操作

- MAKE-SET(x): 创建仅含元素 x 的集合.

$O(1)$

即: 创建仅含元素 x 的一棵树

- FIND-SET(x): 返回 x 所在集合的代表元素

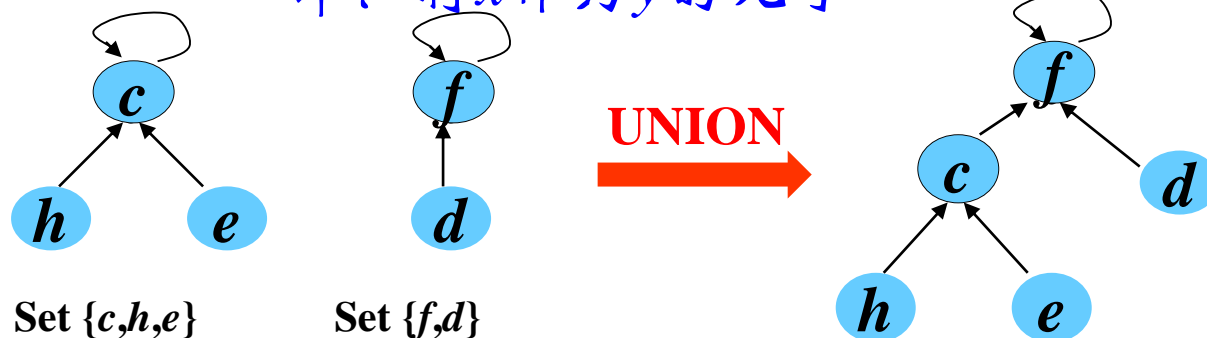
$O(h)$

即: 从结点 x 沿父指针访问直到树根

- UNION(x, y): 合并包含元素 x 和 y 的集合

$O(h)$

即: 将 x 作为 y 的儿子



- n 次合并操作可能得到深度为 n 的树 (简单路径)
- FIND-SET(x)的最坏时间复杂性为 $O(n)$
- n 次FIND-SET(x)操作的时间复杂度可能达到 $O(n^2)$



- 改进目标：使得如下操作序列的代价尽可能低
 - n 个MAKE-SET 操作 (开始阶段执行).
 - m 个MAKE-SET, UNION, FIND-SET操作(后续)
 - $m \geq n$, UNION操作至多执行 $n-1$ 次
- 首要改进对象：
 - FIND-SET(x) : 返回 x 所在集合的代表元素
即：从结点 x 沿父指针访问直到树根 $O(h)$
 - UNION(x, y) : 合并包含元素 x 和 y 的集合
即：将 x 作为 y 的儿子 $O(h)$



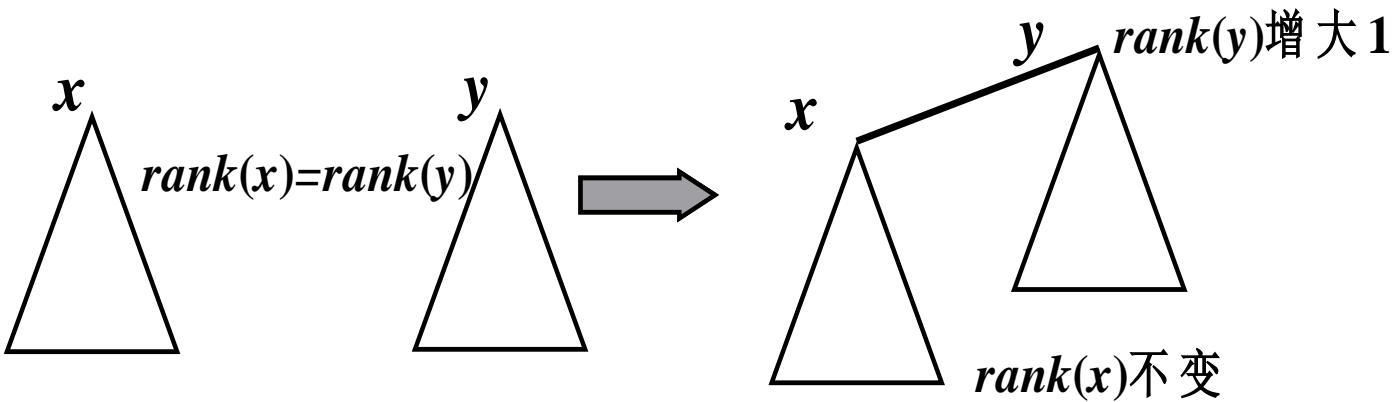
根据以下规则，维护每个结点的秩

- Make-Set(x)操作执行时定义结点 x 的秩为0
- Find-Set(x)操作不改变任意顶点的秩
- Union(x, y) 分两种情况修改结点的秩：
 - 情形a: $\text{rank}(x) = \text{rank}(y)$ 。此时，令 x 指向 y 且 y 是并集的代表元素， $\text{rank}(y)$ 增加1， $\text{rank}(x)$ 不变（其他结点的秩也保持不变）
 - 情形b: $\text{rank}(x) < \text{rank}(y)$ 。此时，令 x 指向 y 且 y 是并集的代表元素， $\text{rank}(y)$ 和 $\text{rank}(x)$ 保持不变（其他结点的秩也保持不变）

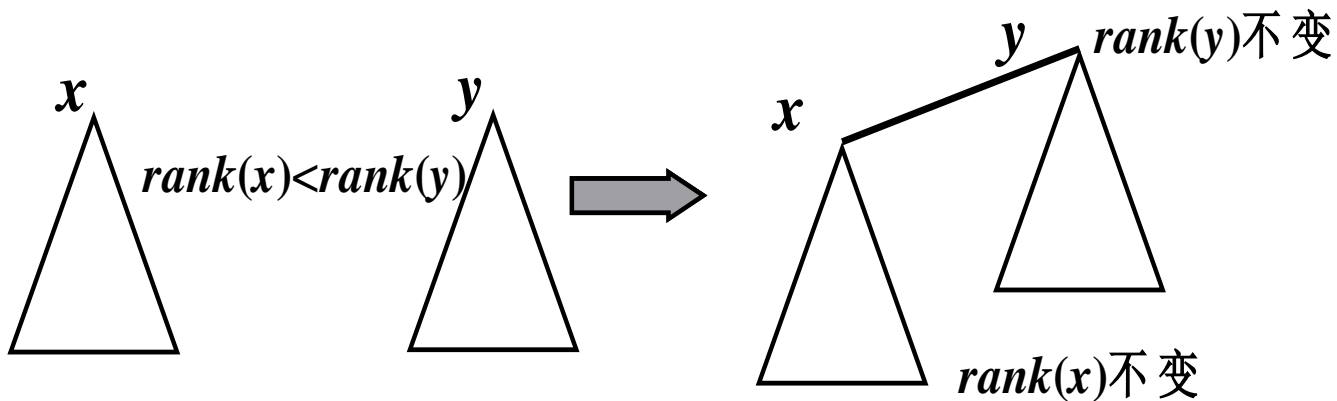


HITWH
SE

情形a



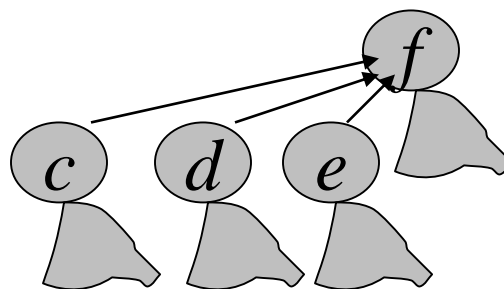
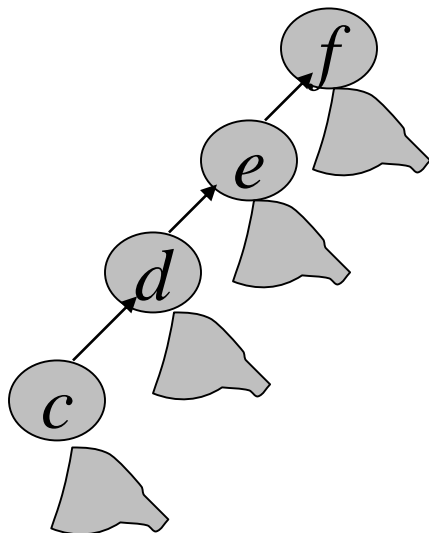
情形b





执行FIND-SET(x)时

- 修改 x 到根结点 r 的路径上的所有结点的指针，使其指向根结点 r
- 路径压缩增加了执行单次FIND-SET(x)操作的时间开销
- 树中的路径长度大幅度降低，为后续FIND-SET(x)操作节省了时间





MAKE-SET(x)

1. $\text{rank}[x] \leftarrow 0$
2. $p[x] \leftarrow x$

FIND-SET(x)

1. $Q \leftarrow \emptyset$
2. While $x \neq p[x]$ Do
3. 将 x 插 $\wedge Q$;
4. $x \leftarrow p[x]$;
5. For $\forall y \in Q$ do
6. $p[y] \leftarrow x$;
7. 输出 x

LINK(x, y)

1. if $\text{rank}[x] > \text{rank}[y]$ then
2. $p[y] \leftarrow x$
3. else $p[x] \leftarrow y$
4. if $\text{rank}[x] = \text{rank}[y]$ then
5. $\text{rank}[y] \leftarrow \text{rank}[y] + 1$

UNION(x, y)

1. LINK(FIND-SET(x), FIND-SET(y))



在并查集上执行 m 个操作的时间复杂度为 $O(m\alpha(n))$

- n 是Make-Set操作的个数
- $\alpha(n) \leq 4$, 对于绝大多数应用成立
- 近似地看, 并查集上的操作序列的时间复杂度几乎是线性的

欲得上述结果, 需要

- 讨论一个增长缓慢的函数-阿克曼函数的逆函数
- 讨论秩的性质
- 证明上述时间复杂度



阿克曼函数是定义在 $k \geq 0, j \geq 1$ 上的递归函数

$$A_k(j) = \begin{cases} j+1 & \text{如果 } k=0 \\ A_{k-1}^{(j+1)}(j) & \text{如果 } k \geq 1 \end{cases}$$

利用数学归纳法，不难验证欲得上述结果，需要

- $A_1(j) = 2j+1$
- $A_2(j) = 2^{j+1}(j+1)-1$
- $A_k(j)$ 是一个“急速”增长的函数

$$A_0(1) = 1+1=2$$

$$A_1(1) = 2*1+1=3$$

$$A_2(1) = 2^{1+1}(1+1)-1=7$$

$$A_3(1) = A_2^{(1+1)}(1) = A_2^{(2)}(1) = A_2(A_2(1)) = A_2(7) = 2^{7+1}(7+1)-1=2047$$

$$A_4(1) = A_3^{(2)}(1) = A_3(A_3(1)) = A_3(2047) = A_2^{(2048)}(2047)$$

$$>> A_2(2047) = 2^{2048} \cdot 2048 - 1 > 2^{2048} = (2^4)^{512} = (16)^{512} > 10^{80}$$



阿克曼函数的逆函数定义为

$$\alpha(n) = \min \{k \mid A_k(1) \geq n\}$$

由于阿克曼函数急速增长，需要 $\alpha(n)$ 缓慢增长

$\alpha(n) \leq 4$ 在人类实践认知范围总成立

n	$0 \leq n \leq 2$	$n=3$	$4 \leq n \leq 7$	$8 \leq n \leq 2047$	$2048 \leq n \leq A_4(1)$...
$\alpha(n)$	0	1	2	3	4	...



引理1. 对于含有 n 个结点的并查集，秩具有如下性质：

- (1) 如果 $x \neq p(x)$ ，则 $rank(x) < rank(p(x))$
- (2) $rank(x)$ 的初始值为0，逐步递增直到 x 不再是集合的代表元素，此后保持不变
- (3) 对于任意 x ， $rank(p(x))$ 是在操作过程中单调递增
- (4) $rank(x) \leq n-1$ 对任意结点成立

证明. 根据秩的定义和并查集上的操作算法可得



对并查集中的每个结点 x ,定义

$$Level(x) = \max \{ k \mid rank(p(x)) \geq A_k(rank(x)) \}$$

$$Iter(x) = \max \{ i \mid A_{Level(x)}^{(i)}(rank(x)) \leq rank(p(x)) \}$$

- 直观上

- $Level(x)$ 是阿克曼函数的最大级,使得该函数在自变量 $rank(x)$ 上的函数值不超过 $rank(p(x))$;
- $Iter(x)$ 是 $Level(x)$ 级阿克曼函数在 $rank(x)$ 上迭代的最大次数,使得迭代结果不超过 $rank(p(x))$



对并查集中的每个非根结点 x , 定义

$$Level(x) = \max \{k \mid rank(p(x)) \geq A_k(rank(x))\}$$

$$Iter(x) = \max \{i \mid A_{Level(x)}^{(i)}(rank(x)) \leq rank(p(x))\}$$

- $0 \leq Level(x) < \alpha(n)$, 且 $Level(x)$ 随时间递增
 - $0 \leq Level(x)$, 因为 $rank(p(x)) \geq rank(x) + 1 = A_0(rank(x))$
 - $Level(x) < \alpha(n)$, 因为 $A_{\alpha(n)}(rank(x)) \geq A_{\alpha(n)}(1) \geq n > rank(p(x))$
- $1 \leq Iter(x) \leq rank(x)$, 且只要 $Level(x)$ 不变则 $Iter(x)$ 不变或增大
 - $1 \leq Iter(x)$, 因为 $rank(p(x)) \geq A_{Level(x)}(rank(x)) = A_{Level(x)}^{(1)}(rank(x))$
 - $Iter(x) \leq rank(x)$, 因为 $A_{Level(x)}^{(rank(x)+1)}(rank(x)) = A_{Level(x)+1}(rank(x)) > rank(p(x))$
 - 由于 $rank(p(x))$ 随时间单调递增, 仅当 $Level(x)$ 增大时 $Iter(x)$ 减小
 - 换言之, 只要 $Level(x)$ 不变则 $Iter(x)$ 不变或增大



定义并查集上 q 个操作之后结点 x 的势能 $\phi_q(x)$ 为

$$\phi_q(x) = \begin{cases} \alpha(n) \cdot \text{rank}(x) & \text{若 } x \text{ 是树根或 } \text{rank}(x) = 0 \\ [\alpha(n) - \text{Level}(x)] \cdot \text{rank}(x) - \text{Iter}(x) & \text{若 } x \text{ 不是树根且 } \text{rank}(x) \geq 1 \end{cases}$$

- $0 \leq \phi_q(x) \leq \alpha(n) \text{rank}(x)$
 - 若 x 是树根, 显然
 - 若 x 不是树根, 则
 - $\phi_q(x) = [\alpha(n) - \text{Level}(x)] \text{rank}(x) - \text{Iter}(x)$
 $\geq [\alpha(n) - (\alpha(n) - 1)] \text{rank}(x) - \text{rank}(x)$
 $= 0$
 - $\phi_q(x) = [\alpha(n) - \text{Level}(x)] \text{rank}(x) - \text{Iter}(x)$
 $\leq [\alpha(n) - (0)] \text{rank}(x) - 0$
 $= \alpha(n) \text{rank}(x)$



定义并查集上 q 个操作之后结点 x 的势能 $\phi_q(x)$ 为

$$\phi_q(x) = \begin{cases} \alpha(n) \cdot \text{rank}(x) & \text{若 } x \text{ 是树根或 } \text{rank}(x) = 0 \\ [\alpha(n) - \text{Level}(x)] \cdot \text{rank}(x) - \text{Iter}(x) & \text{若 } x \text{ 不是树根且 } \text{rank}(x) \geq 1 \end{cases}$$

- 若 x 不是树根，第 $q+1$ 个操作是Link或Find-Set，则 $\phi_{q+1}(x) \leq \phi_q(x)$
 - $\text{rank}(x)$ 和 $\alpha(n)$ 不变
 - 若 $\text{rank}(x)=0$, $\phi_{q+1} = \phi_q = 0$, 论断成立
 - 若 $\text{rank}(x) \geq 1$, ($\text{Level}(x)$ 单调递增)
 - $\text{Level}(x)$ 保持不变, $\text{Iter}(x)$ 增大, $\phi_{q+1}(x) \leq \phi_q(x) - 1$
 - $\text{Level}(x)$ 增大, $\text{Iter}(x)$ 不变或减小,
 $[\alpha(n) - \text{Level}(x)] \text{rank}(x)$ 至少减小 $\text{rank}(x)$
 $\text{Iter}(x)$ 至多减小 $\text{rank}(x) - 1$, 因为 $\text{Iter}(x) < \text{rank}(x)$
 $\phi_{q+1}(x) \leq \phi_q(x) - 1$



定义并查集在 q 个操作之后的势能 ϕ_q 为

$$\phi_q = \sum_x \phi_q(x)$$

- $\phi_q \geq 0$ 恒成立, 因为 $0 \leq \phi_q(x) \leq \alpha(n) \text{rank}(x)$ 对任意 x 成立
- 并查集上任意操作序列的总平摊代价 \geq 总实际代价



势能 $\phi_q = \sum_x \phi_q(x)$

$$\phi_q(x) = \begin{cases} \alpha(n) \cdot \text{rank}(x) & \text{若 } x \text{ 是树根或 } \text{rank}(x) = 0 \\ [\alpha(n) - \text{Level}(x)] \cdot \text{rank}(x) - \text{Iter}(x) & \text{若 } x \text{ 不是树根且 } \text{rank}(x) \geq 1 \end{cases}$$

Make-Set操作的平摊代价为 $O(1)$

Make-Set(y):

- 实际代价为 $O(1)$
- 势能的增量为0
 - 新增一棵以 y 为树根的树, y 的势能为0
 - 不改变其他树的结构和rank, 其他结点的势能不变

势能 $\phi_q = \sum_x \phi_q(x)$

并查集森林实现性能的平摊分析(6)

$$\phi_q(x) = \begin{cases} \alpha(n) \cdot \text{rank}(x) & \text{若 } x \text{ 是树根或 } \text{rank}(x) = 0 \\ [\alpha(n) - \text{Level}(x)] \cdot \text{rank}(x) - \text{Iter}(x) & \text{若 } x \text{ 不是树根且 } \text{rank}(x) \geq 1 \end{cases}$$

Link(y, z)操作的平摊代价为 $O(\alpha(n))$

- 实际代价为 $O(1)$
- 势能增量为 $O(\alpha(n))$
 - 不妨设合并后, y是z的父结点
 - 操作仅可能改变rank(y)
 - 势能发生变化的结点只能是y, z和操作之前y的子结点w
 - w不是树根, 必有 $\phi_{q+1}(w) \leq \phi_q(w)$ (参照前面的性质)
 - z的势能不会增加
 - 操作前, z是树根, 故 $\phi_q(z) = \alpha(n)\text{rank}(z)$
 - 操作后, rank(z)不变, 且 $0 \leq \phi_{q+1}(z) \leq \alpha(n)\text{rank}(z)$
 - y的势能至多增大 $\alpha(n)$
 - 操作前, y是树根, 故 $\phi_q(y) = \alpha(n)\text{rank}(y)$
 - 操作时, rank(y)增大1或保持不变
 - 操作后, y仍是树根, $\phi_{q+1}(y) \leq \phi_q(y) + \alpha(n)$

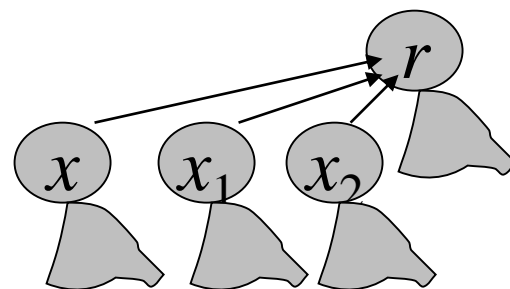
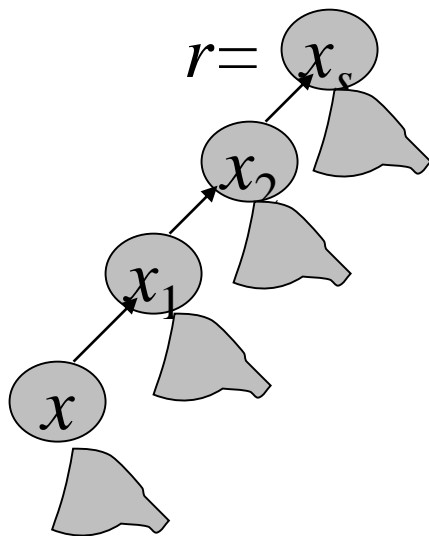
势能 $\phi_q = \sum_x \phi_q(x)$

并查集森林实现性能的平均分析(7)

$$\phi_q(x) = \begin{cases} \alpha(n) \cdot \text{rank}(x) & \text{若 } x \text{ 是树根或 } \text{rank}(x) = 0 \\ [\alpha(n) - \text{Level}(x)] \cdot \text{rank}(x) - \text{Iter}(x) & \text{若 } x \text{ 不是树根且 } \text{rank}(x) \geq 1 \end{cases}$$

Find-Set(x)操作的平摊代价为 $O(\alpha(n))$

- 实际代价为 $O(s)$



- $x = x_1, x_2, \dots, x_{s-1}$ 的势能不会增加

因为它们不是树根，故 $\phi_{q+1}(x_i) \leq \phi_q(x_i)$ (前面的结论)

- 树根 r 的势能不会发生变化

$\text{rank}(r)$ 未发生变化

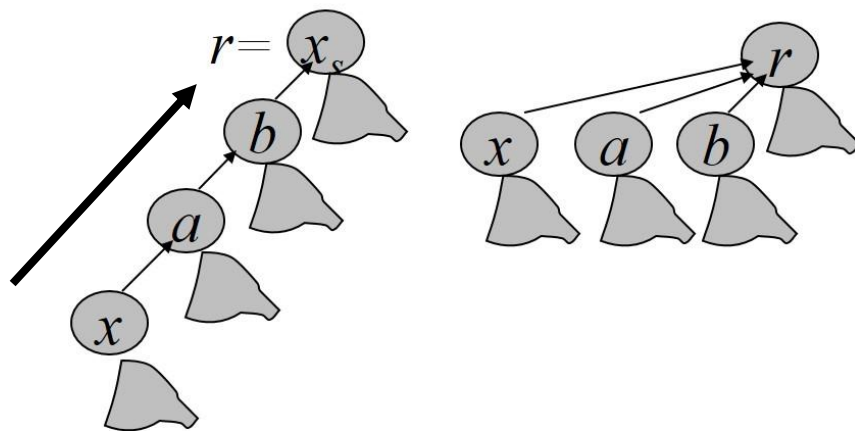


势能 $\phi_q = \sum_x \phi_q(x)$

$$\phi_q(x) = \begin{cases} \alpha(n) \cdot \text{rank}(x) & \text{若 } x \text{ 是树根或 } \text{rank}(x) = 0 \\ [\alpha(n) - \text{Level}(x)] \cdot \text{rank}(x) - \text{Iter}(x) & \text{若 } x \text{ 不是树根且 } \text{rank}(x) \geq 1 \end{cases}$$

Find-Set(x)操作的平摊代价为 $O(\alpha(n))$

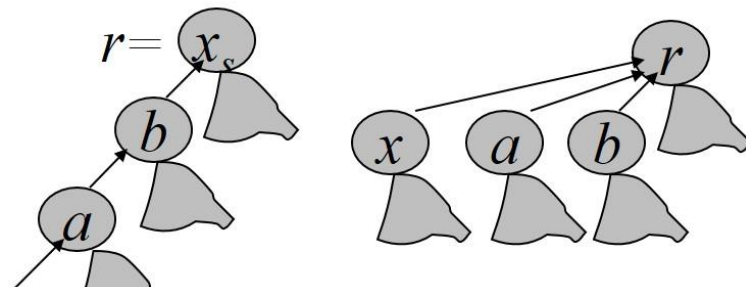
- 路径 x, x_1, \dots, x_s 上至少有 $s - [\alpha(n) + 2]$ 个结点的势能至少减小1
 - 哪些节点的势能至少减少1呢?
 - 结点 b 在 a 之后出现在从 x 到根的路径上, 若 $\text{rank}(a) > 0$, $\text{Level}(a) = \text{Level}(b)$, b 不是根, 则结点 a 的势能至少减少1





- 结点 b 在 a 之后出现在从 x 到根的路径上, 若 $rank(a) > 0$, $Level(a) = Level(b)$, b 不是根, 则结点 a 的势能至少减少1
 - 设 $k = Level(a) = Level(b)$
 - $rank(p(a)) \geq A_k^{(iter(x))}(rank(x))$ (根据 $Iter(a)$ 的定义)
 - $rank(p(b)) \geq A_k(rank(b))$ (根据 $Level(b)$ 的定义)
 - $rank(b) \geq rank(p(a))$ (根据 b 和 a 的出现顺序)
 - 于是有如下关系:
 - $rank(p(b)) \geq A_k(rank(b))$
 $\geq A_k(rank(p(a)))$
 $\geq A_k(A_k^i(rank(a)))$
 $= A_k^{(i+1)}(rank(a))$

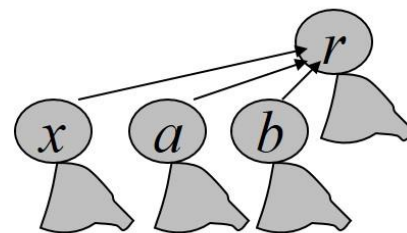
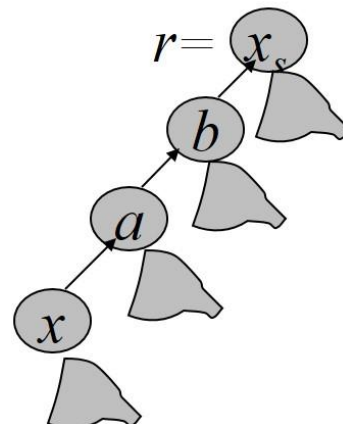
i 为FIND-SET之前 $Iter(x)$ 的取值



FIND-SET(x)之后 a 和 b 的父结点都是 $p(b)$, 即节点 a 的父结点的秩增长之后不小于 $A_{Level(a)}^{Iter(a)+1}(rank(a))$.
 $Level(a)$ 或 $Iter(a)$ 将发生变化, 导致 a 的势能至少减少1.



- 结点 b 在 a 之后出现在从 x 到根的路径上, 若 $\text{rank}(a) > 0$, $\text{Level}(a) = \text{Level}(b)$, b 不是根, 则结点 a 的势能至少减少1
- 从 x 到根的路径上, 不满足上述条件的结点最多有多少个? $\alpha(n) + 2$:
 - 根节点、叶子节点不满足上述条件
 - 对于 $i = 0, 1, \dots, \alpha(n) - 1$, 最后一个出现并满足 $\text{Level}(c) = i$ 的节点 c 不满足上述条件;
- 路径 x_0, x_1, \dots, x_s 上至少有 $s - [\alpha(n) + 2]$ 个结点的势能至少减小1
- $\Phi_i - \Phi_{i+1} \geq s - (\alpha(n) + 2)$
- $\widehat{c_{i+1}} = c_{i+1} + \Phi_{i+1} - \Phi_i$
$$\leq s - (s - (\alpha(n) + 2))$$
$$= \alpha(n) + 2 = O(\alpha(n))$$





在并查集上执行 m 个操作的时间复杂度为 $O(m\alpha(n))$

- Make-Set操作的平摊代价为 $O(1)$
- Union操作的平摊代价为 $O(\alpha(n))$
- Union操作的平摊代价为 $O(\alpha(n))$
- n 是Make-Set操作的个数，亦即并查集管理的数据对象的个数
- $\alpha(n) \leq 4$ ，对于绝大多数应用成立
- 近似地看，并查集上的操作序列的时间复杂度几乎是线性的