



第六章 平摊分析

骆吉洲
计算机科学与技术学院



提纲

- 6.1 平摊分析原理
- 6.2 聚集方法
- 6.3 会计方法
- 6.4 势能方法
- 6.5 动态表操作的平摊分析
- 6.6 斐波那契堆性能平摊分析
- 6.7 并查集性能平摊分析



6.1 平摊分析原理

- 平摊分析的基本思想
- 平摊分析方法



生活给我们的启发



平常的日子
开销低



犒劳一下自己
开销适中



聚会
开销高



购物
开销高

生活是有预算的，人们总是这样安排自己的生活

- 平常日子占绝大多数
- 偶尔犒劳一下自己，然后回归平常的生活
- 大笔开销之后，会适当地节约

结果：生活中的日均花销总维持在适当水平
某段时间的生活费 \approx 日均花销 \times 天数



看看我们的算法吧

• Ch3的一个算法

Graham-Scan(Q)

/* 栈S从底到顶存储按逆时针方向排列的CH(Q)顶点 */

1. 求Q中y-坐标值最小的点 p_0 ;
2. 按照与 p_0 极角(逆时针方向)大小排序Q中其余点，结果为 $\langle p_1, p_2, \dots, p_m \rangle$;
3. Push(p_0, S); Push(p_1, S); Push(p_2, S);
4. FOR $i=3$ TO m DO
5. While NextToTop(S), Top(S)和 p_i 形成非左移动 Do
6. Pop(S);
7. Push(p_i, S);
8. Return S.

考察FOR循环的每次执行

- 如果5-6步被执行，代价较高
- 否则，代价很低

能否像日常生活的例子一样

- 分析操作的平摊代价

➢ 操作序列代价 = 操作个数 \times 平摊代价

- 算法操作数据，数据结构管理数据
- 根据算法需求设计或选用数据结构
- 能否根据预算来设计数据结构
 - 确保频发操作代价低
 - 确保偶发操作代价高
 - 每个操作的平摊代价低

操作序列代价 = 操作个数 \times 平摊代价



本章目标

目标1：掌握平摊分析技术

- 能阅读并理解文献中的平摊分析过程和结果
- 能用它分析连续作用到某数据结构上的一系列操作的总代价
- 初步学习根据算法需求利用平摊分析设计数据结构

目标2：积累三种有用的数据结构，理解现有算法分析结果

- 动态表
- 斐波那契堆
- 并查集

目标3：理解高级程序设计语言中的抽象实现

- Malloc申请的数组与定长数组操作代价相当
- Set的抽象实现的性能



平摊分析

算法经常在某个数据结构上执行一系列操作

- 每个操作的代价各不相同（有高、有低）

问题

- 从平均效果看，每个操作的代价如何分析？
- 操作序列的时间复杂度如何分析？

平摊分析

- 将操作序列的总代价分摊到每个操作上
- 不涉及每个操作被执行的概率
- 不同于平均复杂度分析



平摊分析的基本思想

- 聚集方法（每个操作的代价）
 - 为每个操作都赋予相同的平摊代价
 - 确定 n 个操作的上界 $T(n)$, 每个操作平摊 $T(n)/n$
- 会计方法（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
- 势能方法（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 聚集方法

- 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



一目了然知原理

日期	开销
5月7日	30元
5月8日	50元
5月9日	120元
5月10日	30元
5月11日	30元
.....

操作	开销
OP_1	C_1
OP_2	C_2
OP_3	C_3
OP_4	C_4
OP_5	C_5
.....

日均开销 = Σ 每日开销 ÷ 天数

平摊代价 = $\Sigma C_i \div$ 操作个数



聚集方法的原理

聚集方法的目的

- 分析平摊代价的上界

分析方法

- 分析操作序列中每个操作的代价上界 c_i
- 求得操作序列的总代价的上界 $T(n)=c_1+c_2+\dots+c_n$
- 将 $T(n)$ 平摊到每个操作上得到平摊代价 $T(n)/n$

特点

- 每个操作获得相同的平摊代价
- 较准确地计算 $T(n)$ 需要一定的技巧



聚集方法实例之一：栈操作系列

- 普通栈操作及其时间代价
 - $\text{Push}(S, x)$: 将对象 x 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 $\Theta(n)$



• 新的普通栈操作及其时间代价

– 操作 **Multipop(S, k)**:

去掉 S 的 k 个顶对象, 当 $|S| < k$ 时弹出整个栈

– 实现算法

Multipop(S, k)

```
1 While not STACK-EMPTY(S) and k ≠ 0 Do
2   Pop(S);
3   k ← 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**
- 每个 **Multipop** 的代价最坏是 $O(n)$
- 操作序列的最坏代价为 $T(n) = O(n^2)$
- 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 ← 0
2 while i < k and A[i] = 1 Do
3   A[i] ← 0;
4   i ← i + 1;
5 If i < k Then A[i] ← 1
```

⇒ 从第 1 位开始扫描, 把所有的 1 改为 0, 然后再把碰到的第 1 个 0 改为 1.

• 初始为零的计数器上 n 个 **INCREMENT** 操作分析


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



• 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: $k \cdot n = 8 \cdot 16 = 128$

• 精细分析



HIT CS&E


Counter Value	操作Cost=O(发生改变的位数)								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



HIT CS&E

• 精细分析


- $A[0]$ 每1次操作发生一次改变, 总次数为 n
- $A[1]$ 每2次操作发生一次改变, 总次数为 $n/2$
- $A[2]$ 每4次操作发生一次改变, 总次数为 $n/2^2$
- $A[3]$ 每8次操作发生一次改变, 总次数为 $n/2^3$
- 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)
- $T(n) = \sum_{0 \leq i \leq \lg n} n/2^i < n \sum_{0 \leq i < \infty} 1/2^i = 2n = O(n)$
- 每个 Increment 操作的平摊代价为 $O(1)$



HIT CS&E

6.3 会计方法

- 会计方法的原理
- 会计方法的实例之一
- 会计方法的实例之二



HIT CS&E

一目了然知原理

预算: 早餐5元
午餐10元
晚餐15元


日期	早餐	午餐	晚餐
5月7日	+1	+2	+3
5月8日	-2	-2	-2
5月9日	+0.5	+3	+2
5月10日	-0.5	-1	-1
5月11日	+1	+1	+1
...

预算: 第1类操作 α_1 元
.....
第 k 类操作 α_k 元

操作	种类	实际代价	结余代价
OP ₁	1	c_1	$c_1 - \alpha_1$
OP ₂	3	c_2	$c_2 - \alpha_3$
OP ₃	2	c_3	$c_3 - \alpha_2$
OP ₄	k	c_4	$c_4 - \alpha_k$
OP ₅	$k-2$	c_5	$c_5 - \alpha_{k-2}$
...

If 总结余 ≥ 0 then
总开销 $\leq \alpha_1 \times \text{第1类操作个数} + \dots + \alpha_k \times \text{第}k\text{类操作个数}$
总开销 $\leq (5+10+15) \times \text{天数}$

结余代价应该怎么管理呢?



HIT CS&E


Accounting方法的原理

- 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
 - 不同类型的操作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
 - 平摊代价可能比实际代价大, 也可能比实际代价小

数据结构中存储的 Credit 在任何时候都必须非负, 即 $\sum_{1 \leq i \leq n} \alpha_i - \sum_{1 \leq i \leq n} c_i \geq 0$ 恒成立

平摊代价的选择规则:

- 设 c_i 和 α_i 是操作 i 的实际代价和平摊代价
- $\sum_{1 \leq i \leq n} \alpha_i \geq \sum_{1 \leq i \leq n} c_i$ 必须对于任意 n 个操作序列都成立



HIT CS&E

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
 - 因为在 n 个操作序列中, POP 个数(包括 MULTIPOP 中的 POP)不大于 PUSH 个数
- 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美元存储在该“1”位上，用于支付其被置0时的开销
 - 置0操作无需再付款
 - Cost(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，因为从前面的分析可知 $\sum_{1 \leq i \leq n} c_i < 2n$
- n 个Increment操作序列的总平摊代价
 - $O(n)$



6.4 势能方法

- 势能方法的原理
- 势能方法的实例之一
- 势能方法的实例之二



一目了然知原理

大家有没有觉得会计方法很麻烦？

- 预算：早餐5元 午餐10元 晚餐15元
- 一个月30天，在饭卡中存 $30 \times (5+10+15) = 900$ 元
 - 月底的时候，如果饭卡中仍有余款，则总开销 ≤ 900
- 预算：第1类操作 α_1 元 第 k 类操作 α_k 元
- 给数据结构关联一个函数 ϕ 用于管理结余代价
 - 如果 Op_i 是第 j 类操作，且
 - 实际代价 c_i 低于 α_j ，则结余代价增长， ϕ 的函数值增长 $\alpha_j - c_i$
 - 实际代价 c_i 高于 α_j ，则结余代价减小， ϕ 的函数值减小 $c_i - \alpha_j$
 - 所有操作结束后，如果结余代价大于0
 - 总开销 $\leq \alpha_1 \times \text{第1类操作个数} + \dots + \alpha_k \times \text{第}k\text{类操作个数}$



Potential方法的原理

- Potential方法
 - 目的是分析 n 个操作序列的复杂性上界
 - 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方法把余额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



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（必须是实际代价的上界）

$$\sum_{i=1}^n \alpha_i = \sum_{i=1}^n (c_i + \phi(D_i) - \phi(D_{i-1}))$$

$$= \sum_{i=1}^n c_i + \phi(D_n) - \phi(D_0)$$
- 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 $\phi(D_0) = 0$ ， $\phi(D_i) \geq 0$



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
 - $\phi(D_0)=0$, D_0 中1的个数为0
 - $\phi(D_i) \geq 0 = \phi(D_0)$, 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 操作将 t_i 个1置0, 实际代价为 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 resets 所有 k 位, 所以 $b_{i-1}=t_i=k$
 - 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—基本概念

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



动态表的表示

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
- $num[T]$ 包含了表中的项数
- $size[T]$ 是 T 的大小
- 开始时, $num[T] = size[T] = 0$



动态表的扩张

- 插入一个数组元素时, 完成的操作包括
 - 分配一个包含比原表更多的槽的新表
 - 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



扩张算法

算法: TABLE-INSERT(T, x) /*复杂的插入操作*/

```

1  If size[T]=0 Then /*开销为常数*/
2    获取一个大小为1的表 table[T];
3    size[T] ← 1;
4  If num[T]=size[T] Then /*开销取决于size[T]*/
5    获取一个大小为 2×size[T] 的新表 new-table;
6    将 table[T] 中元素插入 new-table; /*简单插入操作*/
7    释放 table[T];
8    table[T] ← new-table;
9    size[T] ← 2×size[T];
10  将 x 插入 table[T];
11  num[T] ← num[T] + 1
```


HTT CS&E 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 $num[T] < size[T]$, $C_i=1$;
 - 如果 $num[T] = size[T]$, $C_i=i$;
- 共有 n 次操作
 - 最坏情况下, 每次进行 n 次操作, 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地整数幂时第 i 次操作才会引起一次表地扩张

聚集分析-精细分析

- 第 i 次操作的代价 C_i
 - 如果 $i=2^m$, $C_i=i$; 否则 $C_i=1$ $\sum_{i=1}^n c_i \leq n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j < n + 2n = 3n$
- n 次TABLE—INSERT操作的总代价为

★每一操作的平摊代价为 $3n/n=3$

HTT CS&E 初始为空的表上 n 次插入操作的代价分析

会计方法

- 每次执行TABLE—INSERT平摊代价为3
 - 1支付第11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, 数据的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 $3n$

HTT CS&E 初始为空的表上 n 次插入操作的代价分析

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 $\phi(T)=size(T)$
- $\phi(T)=2 \cdot num[T] - size[T]$
 - $num[T] \geq size[T]/2$, $\phi(T) \geq 0$
 - 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
 - 如果发生扩张, $c_i=3$
 - 如果未发生扩张, $c_i=1$

→ P104

初始为空的表上 n 次插入操作的代价的上界为 $3n$

HTT CS&E 动态表的扩张与收缩

- 表的扩张
 - 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1 D D I I D D I I ...
 - 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 $1/4$

HTT 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 $\phi(T)=num[T]$. 因而当删除某项而引起一次收缩时就可利用势来支付其代价

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

HTT CS&E 平摊代价的计算

- 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 $\alpha_i \leq 3$
- 作用于—动态表上的 n 个操作的实际时间为 $O(n)$

→ P105



6.6 斐波那契堆性能平摊分析



堆的性能比较

操作	链表	二叉堆	二项堆	斐波那契堆	松散堆
make-heap	1	1	1	1	1
is-empty	1	1	1	1	1
insert	1	$\log n$	$\log n$	1	1
delete-min	n	$\log n$	$\log n$	$\log n$	$\log n$
decrease-key	n	$\log n$	$\log n$	1	1
delete	n	$\log n$	$\log n$	$\log n$	$\log n$
union	1	n	$\log n$	1	1
find-min	n	1	$\log n$	1	1

n - 堆中存储的元素个数

平摊分析

定理. 从初始为空的斐波那契堆开始, 任意执行由 a_1 个插入, a_2 个删除, a_3 个键值减小操作构成的长度为 n 的操作序列, 其时间复杂度 $O(a_1 + a_2 \log n + a_3)$.



堆的性能比较

操作	链表	二叉堆	二项堆	斐波那契堆	松散堆
make-heap	1	1	1	1	1
is-empty	1	1	1	1	1
insert	1	$\log n$	$\log n$	1	1
delete-min	n	$\log n$	$\log n$	$\log n$	$\log n$
decrease-key	n	$\log n$	$\log n$	1	1
delete	n	$\log n$	$\log n$	$\log n$	$\log n$
union	1	n	$\log n$	1	1
find-min	n	1	$\log n$	1	1

n - 堆中存储的元素个数

平摊分析

插入, 删除和键值减小不可能均在 $O(1)$ 时间内完成. 为什么?



斐波那契堆的历史渊源

斐波那契堆的提出. [Fredman and Tarjan, 1986]

- 巧妙的数据结构和分析
- 提出动机: 改进 Dijkstra's 算法的性能
- Dijkstra 算法执行:
 - $|V|$ 次插入堆元素操作
 - $|V|$ 次抽取堆顶元素操作
 - $|E|$ 次减小键值操作
 - 总时间复杂度为 $O(|E| \log |V|)$
- 改进后时间复杂度为 $O(|E| + |V| \log |V|)$

斐波那契堆的最新研究成果

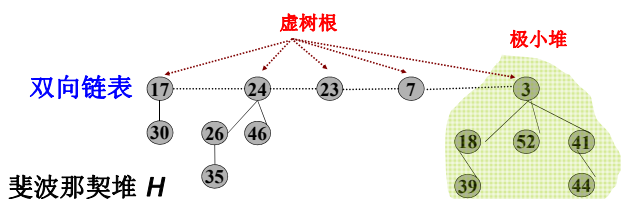
1. Strict fibonacci heaps, *Symposium on the Theory of Computing*, pp 1177-1184, 2012
2. Violation Heaps: A Better Substitute for Fibonacci Heaps, *Data Structures and Algorithms, 2008*



斐波那契堆的结构(1)

斐波那契堆:

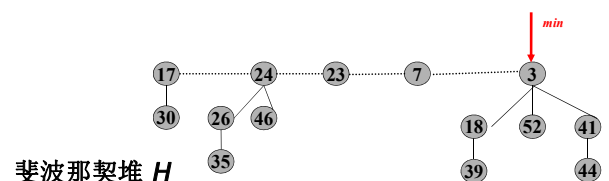
- 一系列树, 每棵树都是一个极小堆
 - 任意结点的键值不超过其孩子结点的键值
- 一个指针, 指向最小元素
- 一些标记顶点



斐波那契堆的结构(2)

斐波那契堆:

- 一系列树, 每棵树都是一个极小堆
- 一个指针, 指向最小元素
 - 从堆中查找最小元素 (Find_min 操作) 的开销为 $O(1)$
- 一些标记顶点



斐波那契堆的结构(3)

斐波那契堆:

- 一系列树，每棵树都是一个极小堆
- 一个指针，指向最小元素
- 一些标记顶点
 - 用于保持堆的“扁平结构”
 - 标记的具体含义是：x被标记，如果它以前是树根，但堆操作过程中变成其他结点的孩子且失去过孩子

⇒ 自成为其父节点的孩子结点后是否失去过孩子

斐波那契堆 H

斐波那契堆的结构(4)

斐波那契堆记号:

- n : 堆 H 中数据元素的个数
- $\text{rank}(x)$: 堆中结点 x 的孩子数
- $\text{rank}(H)$: 堆 H 的所有结点中的最大孩子数
- $t(H)$: 堆 H 中树的棵数
- $\text{mark}(H)$: 堆 H 中被标记顶点的个数
- $\Phi(H) = t(H) + 2 \cdot \text{mark}(H)$: 势能函数

$n=14$ $\text{rank}(H)=3$ $t(H)=5$ $\text{mark}(H)=3$ $\Phi(H)=11$

斐波那契堆 H

插入操作Insert

插入操作的过程:

- 以新结点为根，创建一棵新树
- 将新的树插入树根双向链表
- 如有必要，更新最小元素指针

H

插入操作Insert

插入操作的过程:

- 以新结点为根，创建一棵新树
- 将新的树插入树根双向链表
- 如有必要，更新最小元素指针

H

插入操作Insert分析

插入操作的代价分析: (势能函数 $\Phi(H) = t(H) + 2 \cdot \text{mark}(H)$)

- 实际开销 $O(1)$
- 平摊代价 $O(1)$
 - 操作完成后， $t(H)$ 增大1，但 $\text{mark}(H)$ 不变

H

删除堆顶元素Delete min(1)

链接调整操作

- 将大根树作为小根树的子树，链接在小根树的根下
- Delete_min操作中调用的子操作
- 实际开销为 $O(1)$

大根树 小根树 仍是极小堆

树 T_1 树 T_2 树 T'

HIT CS&E

删除堆顶元素Delete_min(2)

- Delete_min操作过程
 - 删除最小元素；将其所有孩子链入双向链表，更新最小元素指针
 - 合并双向链表中的树根，使得没有树根具有相同rank

HIT CS&E

删除堆顶元素Delete_min(3)

- Delete_min操作过程
 - 删除最小元素；将其所有孩子链入双向链表，更新最小元素指针
 - 实际开销 $O(n(H)+rank(x))$
 - 合并双向链表中的树根，使得没有树根具有相同rank

HIT CS&E

删除堆顶元素Delete_min(4)

- Delete_min操作过程
 - 删除最小元素；将其所有孩子链入双向链表，更新最小元素指针
 - 合并双向链表中的树根，使得没有树根具有相同rank

HIT CS&E

删除堆顶元素Delete_min(5)

- Delete_min操作过程
 - 删除最小元素；将其所有孩子链入双向链表，更新最小元素指针
 - 合并双向链表中的树根，使得没有树根具有相同rank

HIT CS&E

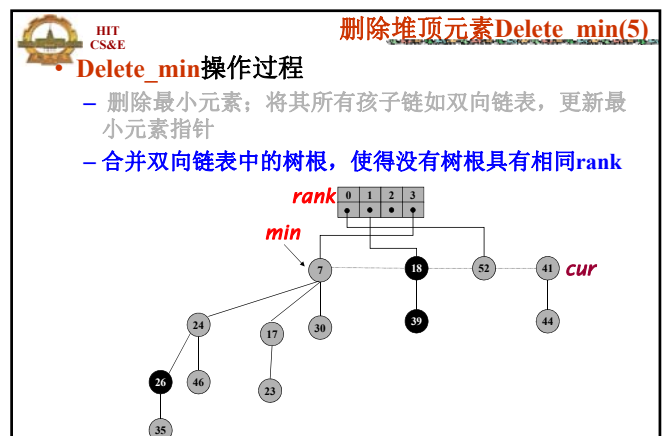
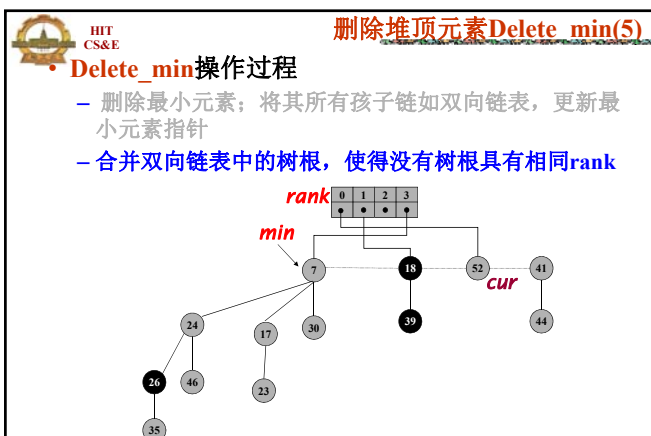
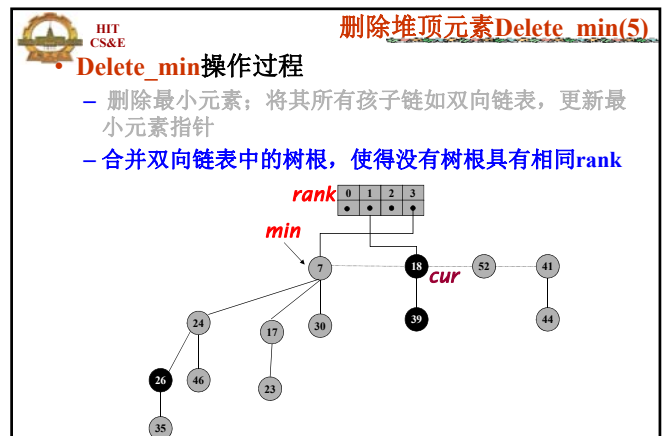
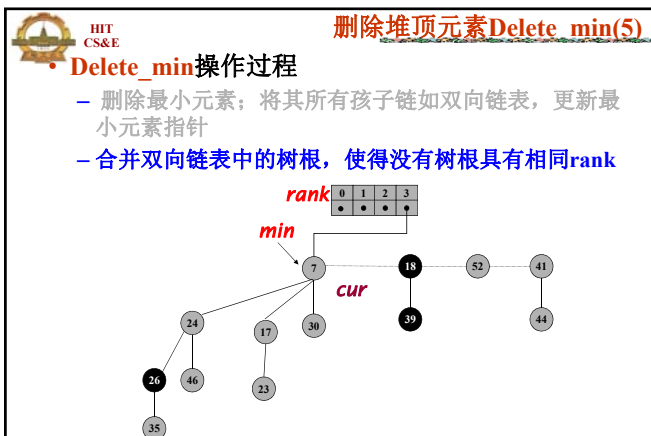
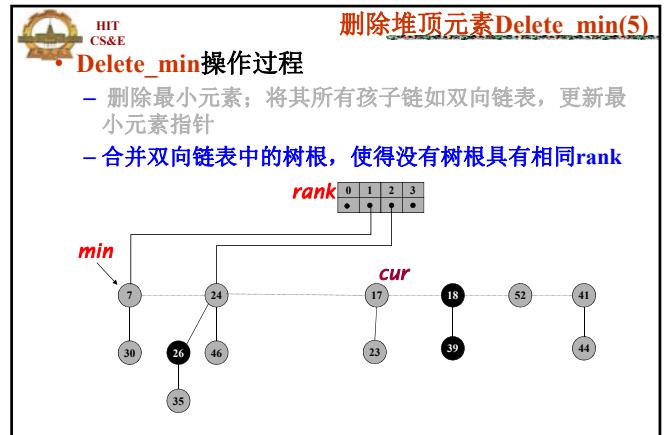
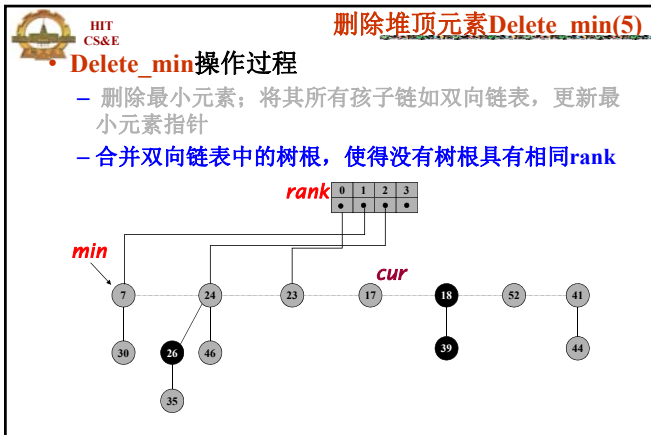
删除堆顶元素Delete_min(5)

- Delete_min操作过程
 - 删除最小元素；将其所有孩子链入双向链表，更新最小元素指针
 - 合并双向链表中的树根，使得没有树根具有相同rank

HIT CS&E

删除堆顶元素Delete_min(5)

- Delete_min操作过程
 - 删除最小元素；将其所有孩子链入双向链表，更新最小元素指针
 - 合并双向链表中的树根，使得没有树根具有相同rank



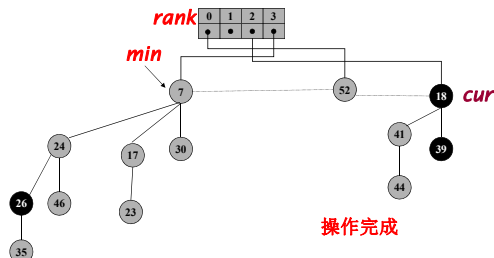
斐波那契堆记号:

- n : 堆H中数据元素的个数
- $rank(x)$: 堆中结点x的孩子数
- $rank(H)$: 堆H的所有结点中的最大孩子数
- $i(H)$: 堆H中树的棵数
- $mark(H)$: 堆H中被标记顶点的个数
- $\Phi(H) = i(H) + 2 \cdot mark(H)$: 势能函数

删除堆顶元素Delete min(5)

Delete_min操作过程

- 删除最小元素；将其所有孩子链如双向链表，更新最小元素指针
- 合并双向链表中的树根，使得没有树根具有相同rank



删除堆顶元素Delete min的分析

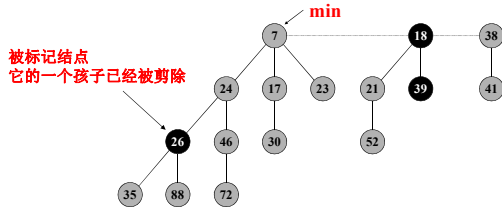
删除操作的代价分析: (势能函数 $\Phi(H) = t(H) + 2 \cdot \text{mark}(H)$)

- 实际开销 $O(\text{rank}(H)) + O(\alpha(H))$
 - $O(\text{rank}(H))$ 时间内将最小元素结点的孩子插入双向链表
 - $O(\text{rank}(H)) + O(\alpha(H))$ 时间内更新最小元素指针
 - $O(\text{rank}(H)) + O(\alpha(H))$ 维护双向链表
- 平摊代价 $O(\text{rank}(H))$
 - 操作完成后的堆记为 H' , 操作前的堆记为 H
 - $t(H') \leq \text{rank}(H) + 1$, 因为 H' 中没有两个树具有相同 rank
 - $\Delta \alpha(H) \leq \text{rank}(H) + 1 - t(H)$
- Delete_min 操作的平摊代价很好吗?
 - 对, 这个代价很好, 因为
 - 如果只有插入和删除操作, 则结果即为二项堆
 - 这意味着 $\text{rank}(H) \leq g_n$
 - 减小键值操作也将确保 $\text{rank}(H) \leq g_n$

键值减小操作Decrease key

直观上, 键值减小操作作用于结点 x , 进行

- 若减小键值后, 堆性质仍成立, 则仅需减小键值
- 否则, 将x子树剪切下来插入双向链表, 标记其父结点
- 为了保持堆的“扁平结构”, 一旦剪除某个孩子的第二个孩子结点, 则将该结点剪切下来插入双向链表 (如果必要, 需要去除该结点上的标记)

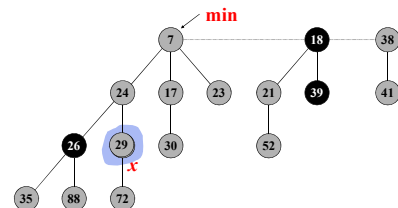


键值减小操作Decrease key(2)

直键值减小操作作用于结点 x (例如, 键值从46减小为29)

情形1：减小键值后堆性质仍成立

- 减小键值
- 如有必要，更新最小元素指针

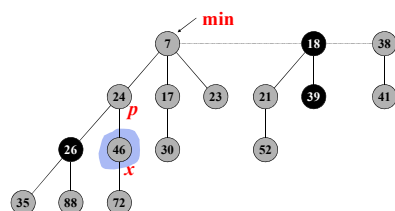


键值减小操作Decrease key(3)

直键值减小操作作用于结点x (例如, 键值从46减小为15)

情形2a: 减小键值后堆性质不成立

- 减小键值
- 剪切 x 子树, 插入双向链表, 将 x 置为未标记结点
- 若 x 的父结点 p 未标记, 则标价它
- 否则, 剪切 p , 插入双向链表, 将 p 置为未标记结点 (递归过程)

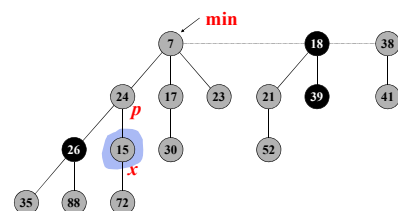


键值减小操作Decrease key(3)

直键值减小操作作用于结点 x (例如, 键值从46减小为15)

情形2a: 减小键值后堆性质不成立

- 减小键值
- 剪切 x 子树, 插入双向链表, 将 x 置为未标记结点
- 若 x 的父结点 p 未标记, 则标价它
- 否则, 剪切 p , 插入双向链表, 将 p 置为未标记结点 (递归过程)



键值减小操作Decrease key(5)

直键值减小操作作用于结点 x (例如, 键值从46减小为15)

情形2a: 减小键值后堆性质不成立

- 减小键值
- 剪切 x 子树, 插入双向链表, 将 x 置为未标记结点
- 若 x 的父结点 p 未标记, 则标价它
- 否则, 剪切 p , 插入双向链表, 将 p 置为未标记结点 (递归过程)

键值减小操作Decrease key(6)

直键值减小操作作用于结点 x (例如, 键值从46减小为15)

情形2a: 减小键值后堆性质不成立

- 减小键值
- 剪切 x 子树, 插入双向链表, 将 x 置为未标记结点
- 若 x 的父结点 p 未标记, 则标价它
- 否则, 剪切 p , 插入双向链表, 将 p 置为未标记结点 (递归过程)

键值减小操作Decrease key(7)

直键值减小操作作用于结点 x (例如, 键值从35减小为5)

情形2b: 减小键值后堆性质不成立

- 减小键值
- 剪切 x 子树, 插入双向链表, 将 x 置为未标记结点
- 若 x 的父结点 p 未标记, 则标价它
- 否则, 剪切 p , 插入双向链表, 将 p 置为未标记结点 (递归过程)

键值减小操作Decrease key(8)

直键值减小操作作用于结点 x (例如, 键值从35减小为5)

情形2b: 减小键值后堆性质不成立

- 减小键值
- 剪切 x 子树, 插入双向链表, 将 x 置为未标记结点
- 若 x 的父结点 p 未标记, 则标价它
- 否则, 剪切 p , 插入双向链表, 将 p 置为未标记结点 (递归过程)

键值减小操作Decrease key(9)

直键值减小操作作用于结点 x (例如, 键值从35减小为5)

情形2b: 减小键值后堆性质不成立

- 减小键值
- 剪切 x 子树, 插入双向链表, 将 x 置为未标记结点
- 若 x 的父结点 p 未标记, 则标价它
- 否则, 剪切 p , 插入双向链表, 将 p 置为未标记结点 (递归过程)

键值减小操作Decrease key(10)

直键值减小操作作用于结点 x (例如, 键值从35减小为5)

情形2b: 减小键值后堆性质不成立

- 减小键值
- 剪切 x 子树, 插入双向链表, 将 x 置为未标记结点
- 若 x 的父结点 p 未标记, 则标价它
- 否则, 剪切 p , 插入双向链表, 将 p 置为未标记结点 (递归过程)

HIT CS&E

键值减小操作Decrease_key(11)

直接键值减小操作作用于结点 x (例如, 键值从35减小为5)

情形2b: 减小键值后堆性质不成立

- 减小键值
- 剪切 x 子树, 插入双向链表, 将 x 置为未标记结点
- 若 x 的父结点 p 未标记, 则标价它
- 否则, 剪切 p , 插入双向链表, 将 p 置为未标记结点 (递归过程)

HIT CS&E

键值减小操作Decrease_key(12)

直接键值减小操作作用于结点 x (例如, 键值从35减小为5)

情形2b: 减小键值后堆性质不成立

- 减小键值
- 剪切 x 子树, 插入双向链表, 将 x 置为未标记结点
- 若 x 的父结点 p 未标记, 则标价它
- 否则, 剪切 p , 插入双向链表, 将 p 置为未标记结点 (递归过程)

HIT CS&E

键值减小操作Decrease_key的分析

键值减小操作的代价分析 (势能函数 $\Phi(H) = t(H) + 2 \cdot \text{mark}(H)$)

- 实际开销 $O(c)$
 - $O(1)$ 时间内减小键值
 - 执行 c 次剪切操作, 每次需要时间 $O(1)$
- 平摊代价 $O(1)$
 - 操作完成后的堆记为 H' , 操作前的堆记为 H
 - $t(H') \leq t(H) + c$
 - $\text{mark}(H') = \text{mark}(H) - c + 2$
 - $\Delta\Phi(H) \leq c + 2(-c + 2) = 4 - c$

$$\begin{aligned} \Delta i &= c + \Delta\Phi(H) \\ &\leq c + 4 - c = 4 \end{aligned}$$

HIT CS&E

目前得到的分析结果

在势能函数 $\Phi(H) = t(H) + 2 \cdot \text{mark}(H)$ 下

- 插入操作的平摊代价 $O(1)$
- 键值减小操作的平摊代价 $O(1)$
- delete_min的代价是 $O(\text{rank}(H))$
- 删除操作的平摊代价 $O(\text{rank}(H))$
 - 先将节点 x 的键值减小为 $-\infty$, 则 x 将出现在堆顶, 平摊代价 $O(1)$
 - 调用delete_min删除堆顶元素, 平摊代价 $O(\text{rank}(H))$

欲得定理结论, 需证明 $\text{rank}(H) = O(\log n)$
这意味着, 斐波那契堆管理的元素个数是 $\text{rank}(H)$ 的指数

HIT CS&E

$\text{rank}(H)$ 的上界分析

引理1. 设 x 是斐波那契堆中的任意结点, 记 $k = \text{rank}(x)$ 。设 y_1, y_2, \dots, y_k 是结点 x 的所有孩子结点且恰好按其成为 x 的孩子的先后次序列出, 则 $\text{rank}(y_1) \geq 0$, 且 $\text{rank}(y_i) \geq i-2$ 对 $i=2, 3, \dots, k$ 成立

证明. 当 y_i 成为 x 的孩子时,

x 已有孩子 y_1, y_2, \dots, y_{i-1} , 故 $\text{rank}(x) = i-1$
 y_i 要成为 x 的孩子, 需满足 $\text{rank}(y_i) = \text{rank}(x) - 1 = i-2$

y_i 成为 x 的孩子之后, 至多失去一个孩子

HIT CS&E

$\text{rank}(H)$ 的上界分析

引理2. 设 x 是斐波那契堆中的任意结点, 记 $k = \text{rank}(x)$, 则 x 所在的极小堆中至少有 $\left\lceil \frac{1+\sqrt{5}}{2} \right\rceil$ 个结点。

证明. 令 s_k 表示阶为 k 的斐波那契堆中顶点的最小个数。归纳证明 $s_k \geq 1 + F(k)$

根据引理1, 用归纳法统计 x 及其子孙结点的个数有 $8 + 13 = 21$

$s_k = 1 + s_0 + s_1 + \dots + s_{k-2}$

根 1 2 3 k

$\geq 1 + 1 + F(0) + F(1) + \dots + F(k-2)$ (归纳假设)
 $= 1 + F(k)$ (斐波那契数列性质, 习题3.2)
 $\geq \left\lceil \frac{1+\sqrt{5}}{2} \right\rceil^k$ (斐波那契数列性质, 习题3.2)

HIT
CS&Erank(H)的上界分析引理3. 设 H 是含有 n 个结点的斐波那契堆, 则 $\text{rank}(H)=O(\log n)$

证明.

- 记 $\text{rank}(H)=k$
- 考虑 H 中结点的个数 n
- 由引理2可知, 结点个数至少为 $\left(\frac{1+\sqrt{5}}{2}\right)^{k-1}$
- 于是, $n \geq \left(\frac{1+\sqrt{5}}{2}\right)^{k-1}$

HIT
CS&E斐波那契堆上的合并操作Union合并操作作用于堆 H_1 和 H_2 - 合并 H_1 和 H_2 的根结点双向链表- 实际开销 $O(1)$ - 平摊开销 $O(1)$

- $t(H_1 \cup H_2) = t(H_1) + t(H_2)$
- $\text{mark}(H_1 \cup H_2) = \text{mark}(H_1) + \text{mark}(H_2)$
- $\Phi(H) = t(H) + 2 \cdot \text{mark}(H)$
- 势能改变量为0

HIT
CS&E斐波那契堆上的删除操作Delete删除操作作用于堆的任意结点 x - 将 x 的键值减小为 $-\infty$

- 删除堆顶元素

- 平摊开销 $O(\log n)$

- 减小键值操作的平摊代价为 $O(\text{rank}(H))$
- $\text{mark}(H) = O(\log n)$
- 删除堆顶元素的平摊代价为 $O(1)$

HIT
CS&E结论

操作	链表	二叉堆	二项堆	斐波那契堆	松散堆
make-heap	1	1	1	1	1
is-empty	1	1	1	1	1
insert	1	$\log n$	$\log n$	1	1
delete-min	n	$\log n$	$\log n$	$\log n$	$\log n$
decrease-key	n	$\log n$	$\log n$	1	1
delete	n	$\log n$	$\log n$	$\log n$	$\log n$
union	1	n	$\log n$	1	1
find-min	n	1	$\log n$	1	1

 n - 堆中存储的元素个数

平摊分析

定理. 从初始为空的斐波那契堆开始, 任意执行由 a_1 个插入, a_2 个删除, a_3 个键值减小操作构成的长度为 n 的操作序列, 其时间复杂度 $O(a_1 + a_2 \log n + a_3)$.

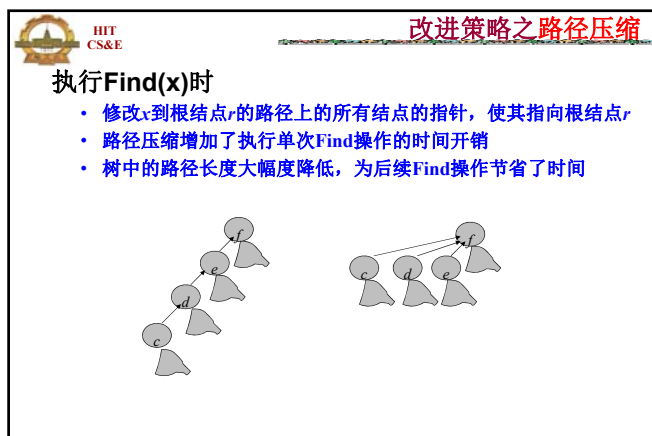
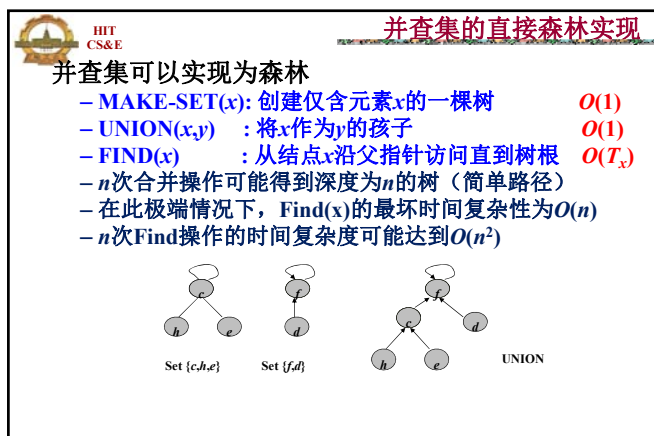
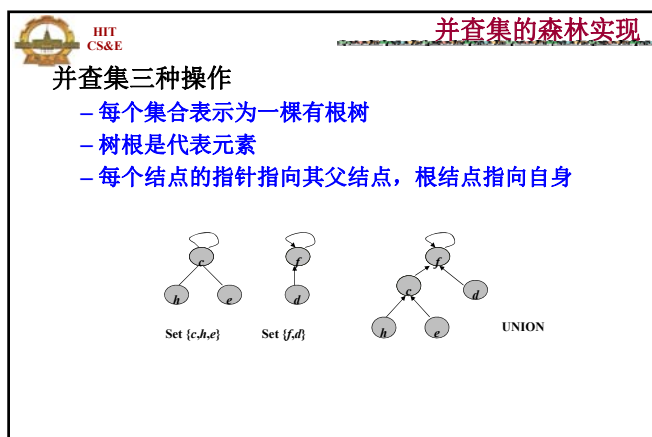
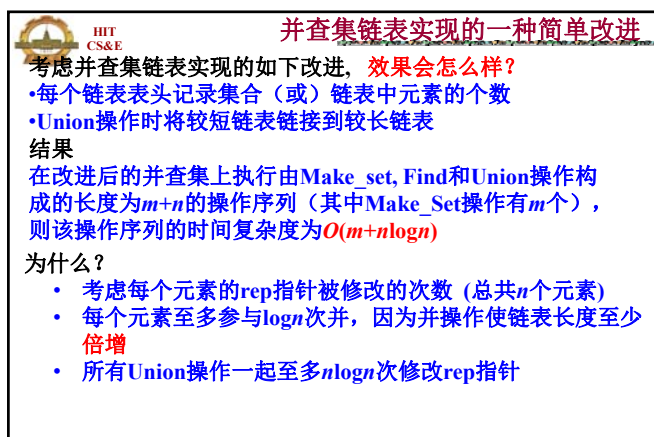
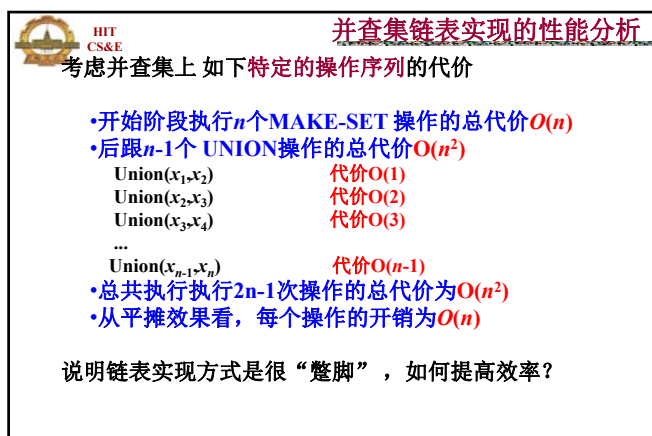
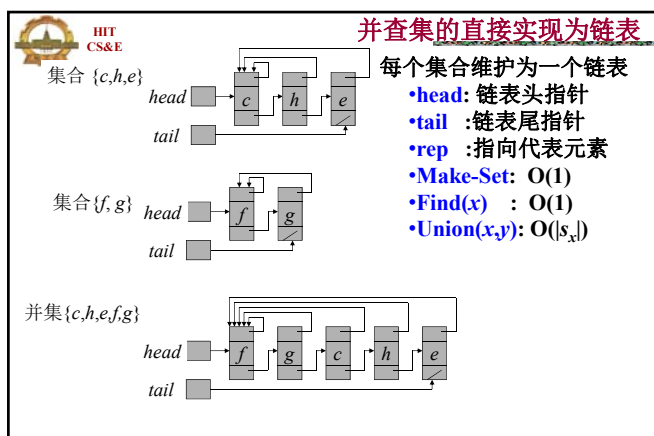
HIT
CS&E6.7 并查集性能平摊分析

- 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
- 并查集性能平摊分析

HIT
CS&E并查集

- 目的: 管理 n 个不相交的集合 $C = \{S_1, \dots, S_n\}$
 - 每个集合 S_i 维护一个代表元素 x_i
- 支持的操作
 - MAKE-SET(x): 创建仅含元素 x 的集合.
 - UNION(x, y): 合并代表元素分别为 x 和 y 的集合
 - FIND-SET(x): 返回 x 所在集合的代表元素
- 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
- 典型应用 (管理图的连通分支)
 - 找出图的连通分支
 - Kruskal算法中维护生成树产生过程中的连通分支
- 最新研究

1. Finding dominators via disjoint set union, *Journal of Discrete Algorithms*, vol23, pp 2-20, 2013
 2. Disjoint set union with randomized linking, *Symp. on Discrete Algorithms*, pp1005-1017, 2014





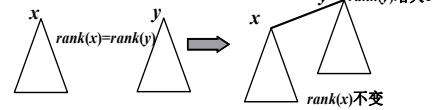
改进策略之按秩合并

根据以下规则，维护每个结点的秩

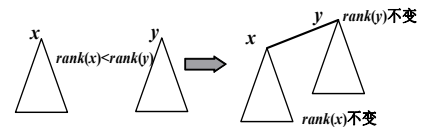
- **MakeSet(x)**操作执行时定义结点x的秩为0
- **Find**操作不改变任意顶点的秩
- **Union(x,y)** 分两种情况修改结点的秩：
 - **情形a: $\text{rank}(x) = \text{rank}(y)$** 。此时，令x指向y且y是并集的代表元素， $\text{rank}(y)$ 增加1， $\text{rank}(x)$ 不变（其他结点的秩也保持不变）
 - **情形b: $\text{rank}(x) < \text{rank}(y)$** 。此时，令x指向y且y是并集的代表元素， $\text{rank}(y)$ 和 $\text{rank}(x)$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



情形a



情形b



并查集操作算法

UNION(x,y)

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

MAKE-SET(x)

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

FIND(x)

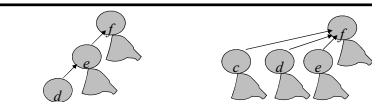
1. $Q \leftarrow \emptyset$
2. While $x \neq p[x]$ Do
3. 将x插入Q;
4. $x \leftarrow p[x]$;
5. For $\forall y \in Q$ do
6. $p[y] \leftarrow x$;
7. 输出x

LINK(x,y)

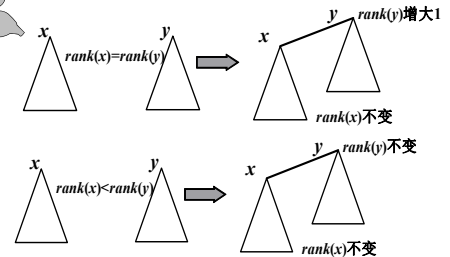
1. if $\text{rank}[x] > \text{rank}[y]$ then
2. $p[y] \leftarrow x$
3. else $p[x] \leftarrow y$
4. if $\text{rank}[x] = \text{rank}[y]$ then
5. $\text{rank}[y] \leftarrow \text{rank}[y] + 1$



Find



Union



并查集中每个元素的秩有什么基本性质呢？



秩的性质

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 $\text{rank}(x) < \text{rank}(p(x))$
- (2) $\text{rank}(x)$ 的初始值为0，缓慢递增直到x不再是集合的代表元素，此后保持不变
- (3) 对于任意x, $\text{rank}(p(x))$ 是在操作过程中单调缓慢递增
- (4) $\text{rank}(x) \leq n-1$ 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

$\text{rank}(x)$ 缓慢增长

如果将 $\text{rank}(x)$ 与一个缓慢增长的函数相关联
能否得出秩的其他性质呢？



并查集的性能

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（亦即：并查集中元素的个数）
- $\alpha(n) \leq 4$ ，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
- 深入讨论秩的性质
- 证明上述时间复杂度

阿克曼函数

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

- $A_1(j) = A_0(A_0(\dots(A_0(j))\dots))$ (j+1层递归)
- $= A_0(A_0(\dots(A_0(j))\dots)) + 1$ (j层递归)
- $= A_0(A_0(\dots(A_0(j))\dots)) + 1 + 1$ (j-1层递归)
- $= \dots$
- $= A_0(j) + j$
- $= 2j + 1$

阿克曼函数

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

- $A_2(j) = A_1(A_1(\dots(A_1(j))\dots))$ (j+1层递归)
- $= 2A_1(A_1(\dots(A_1(j))\dots)) + 1$ (j层递归)
- $= 2[2A_1(A_1(\dots(A_1(j))\dots)) + 1] + 1$ (j-1层递归)
- $= 2^2 A_1(A_1(\dots(A_1(j))\dots)) + 2 + 1$ (j-1层递归, 整理上式)
- $= \dots$
- $= 2^j A_1(j) + 2^{j-1} + \dots + 2 + 1$ (1层递归)
- $= 2^j(2j+1) + 2^{j-1} + 2^{j-2} + \dots + 2 + 1$
- $= 2^{j+1}j + 2^{j+1} - 1$ (整理上式)
- $= 2^{j+1}(j+1) - 1$

阿克曼函数

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

$A_0(1) = 1 + 1 = 2$
 $A_1(1) = 2 * 1 + 1 = 3$
 $A_2(1) = 2^{1+1}(1+1) - 1 = 7$
 $A_3(1) = A_2^{(1+1)}(1) = A_2(A_2(1)) = A_2(7) = 2^{7+1}(7+1) - 1 = 2047$
 $A_4(1) = A_3^{(2)}(1) = A_3(A_3(1)) = A_3(2047) = A_2^{(2048)}(2047)$
 $\gg A_2(2047) = 2^{2048} \cdot 2048 - 1 > 2^{2048} = (2^4)^{512} = (16)^{512} > 10^{80}$

阿克曼函数的逆函数

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, 故 $\alpha(n)$ 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	...

将 $\alpha(\cdot)$ 作用到 $\text{rank}(x)$ 上, 能否得出 $\text{rank}(x)$ 的其他性质?

并查集性能的平摊分析

对并查集中的每个结点 x , 定义

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

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

$p(x)$ \bullet $\text{rank}(p(x))$
 \bullet $\text{rank}(x)$ 与 $\text{rank}(p(x))$ 的距离用 $\alpha(\cdot)$ 来刻画
 x \bullet $\text{rank}(x)$

- 直观上
 - 一条路径上有 Level 值或 Iter 值较大的节点, 路径就很短, 继而 $\text{Find}(x)$ 的代价就低
 - Level 值、 Iter 可能有助于建立势能函数, 它们有什么性质呢?

并查集性能的平摊分析(1)

对并查集中的每个结点 x , 定义

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

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

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



并查集性能的平摊分析(2)

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
 - $\phi_q(x) = [\alpha(n) - \text{Level}(x)] \text{rank}(x) - \text{Iter}(x)$
 $\geq [\alpha(n) - (\alpha(n)-1)] \text{rank}(x) - \text{rank}(x) = 0$
 - $\phi_q(x) = [\alpha(n) - \text{Level}(x)] \text{rank}(x) - \text{Iter}(x)$
 $\leq [\alpha(n) - (0)] \text{rank}(x) - 0 = \alpha(n) \text{rank}(x)$



并查集性能的平摊分析(3)

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Union或Find, 则 $\phi_{q+1}(x) \leq \phi_q(x)$
 - $\text{rank}(x)$ 和 $\alpha(n)$ 不变
 - 若 $\text{rank}(x)=0$, 由 $\text{Iter}(x) \leq \text{rank}(x)$ 可知, 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 $\text{Iter}(x) < \text{rank}(x)$, $\phi_{q+1}(x) \leq \phi_q(x) - 1$



并查集性能的平摊分析(4)

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



并查集性能的平摊分析(5)

$$\text{势能 } \phi_q = \sum_x \phi_q(x)$$

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

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

$$\text{势能 } \phi_q = \sum_x \phi_q(x)$$

并查集性能的平摊分析(6)

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

Union(y, z)操作的平摊代价为 $\Theta(\alpha(n))$

- 实际代价为 $\Theta(1)$
- 势能增量为 $\Theta(\alpha(n))$
 - 不妨设合并后, y 是 z 的父结点
 - 操作仅可能改变 $\text{rank}(y)$
 - 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, $\text{rank}(z)$ 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, $\text{rank}(y)$ 增大1或保持不变
 - 操作后, y 仍是树根, $\phi_{q+1}(y) \leq \phi_q(y) + \alpha(n)$

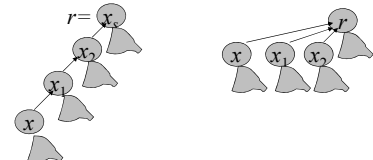
$$\text{势能 } \phi_q = \sum_x \phi_q(x)$$

并查集性能的平摊分析(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(x)操作的平摊代价为 $\Theta(\alpha(n))$

- 实际代价为 $\Theta(s)$



- $x = x_0 \rightarrow x_1 \rightarrow \dots \rightarrow x_{s-1}$ 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

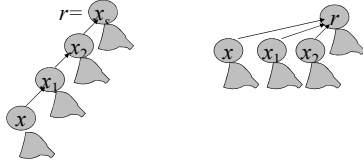
势能 $\phi_q = \sum_x \phi_q(x)$

并查集性能的平摊分析(8)

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

Find(x)操作的平摊代价为 $\Theta(\alpha(n))$

- 实际代价为 $\Theta(s)$



- 平摊代价为 $\Theta(\alpha(n))$

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 (参见讲义)



结论

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 $\Theta(\alpha(n))$
- Union操作的平摊代价为 $\Theta(\alpha(n))$
- 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