HIT
MDC

第七章

平摊分析

张开旗
海量数据计算研究中心
计算学部

提要

7.1 平摊分析基本原理
7.2 聚集方法
7.3 会计方法
7.4 势能方法
7.5 动态表平摊分析
6.6 斐波那契堆性能平摊分析
6.7 并查集性能平摊分析

7.1 平推分析基本原理

- 平推分析的基本思想
- 平推分析方法

关注一系列**数据结构上**操作的时间复杂度：

考虑操作序列：$OP_1, OP_2, \ldots, OP_m$

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
  - Push(S, x): 将对象x压入栈S
  - 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)

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

        Pop(S);

        $k \leftarrow k-1$.
  - 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！

平摊分析的基本思想

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

平摊分析的基本思想

- 聚集方法
  - 确定n个操作的上界$T(n)$, 每个操作平摊$T(n)/n$
- 会计方法
  - 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
  - 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"

7.2 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

聚集方法实例之一：栈操作系列

- 普通栈操作及其时间代价
  - Push(S, x): 将对象x压入栈S
  - 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$\theta(n)$

HIT
MDC

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: Cost=t_1
操作2: Cost=t_2
.
.
.
操作n: Cost=t_n

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

每个操作平摊代价:
$T(n)/n$

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$T(n)/n = O(n)$
  - 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$T(n) \le 2n = O(n)$
    - 平摊代价为$T(n)/n=O(1)$

分析太粗糙！！！

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：$A[0..k\text{-}1]$，存储二进制数x

输出：$A[0..k\text{-}1]$，存储二进制数x+1 $mod\ 2^k$

INCREMENT(A)

$i \leftarrow 0$

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

  $A[i] \leftarrow 0$;

  $i \leftarrow i+1$;

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

- 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：$k*n=8*16=128$
- 精细分析

- 初始为零的计数器上n个INCREMENT操作分析

Counter Value N	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	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

- 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, …., 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$O(1)$

7.3 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

会计方法的原理

- 会计方法
  - 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余额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余额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
    - $\sum_{1\le i\le n}\alpha_i \ge \sum_{1\le i\le n}c_i$ 必须对于任意n个操作序列都成立

会计方法的原理

- 会计方法
  - 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余额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余额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余额在任何时候都必须非负，则$\Sigma_{1\le i\le n}\alpha_i - \Sigma_{1\le i\le n}c_i \ge 0$永远成立

栈操作序列的分析

- 各栈操作的实际代价
  - Cost(PUSH)=*1*
  - Cost(POP)=*1*
  - Cost(MULTIPOP)=$min\{k, s\}$
- 各栈操作的平摊代价
  - Cost(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
  - Cost(POP)=*0*
  - Cost(MULTIPOP)=*0*
- 平摊代价满足
  - $\Sigma_{1\le i\le n}\alpha_i - \Sigma_{1\le i\le n} c_i \ge 0$ 对于任意n个操作序列都成立
    - 因为栈中的元素个数≥0，即任何时刻，栈上的Credit非负
- 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 （每次有且只有1位被置1）
- 平摊代价满足
  - $\Sigma_{1\leq i\leq n}\alpha_i \geq \Sigma_{1\leq i\leq n} c_i$ 对于任意n个操作序列都成立
  - 即：任何时刻，计数器上的Credit余额非负
- 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 N	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	0	0	0	0	0	0	0	1 1	1
2	0	0	0	0	0	0	1 1	0 0	2
3	0	0	0	0	0	0	1 1	1 1	1
4	0	0	0	0	0	1 1	0 0	0 0	3
5	0	0	0	0	0	1 1	0 0	1 1	1
6	0	0	0	0	0	1 1	1 1	0 0	2
7	0	0	0	0	0	1 1	1 1	1 1	1
8	0	0	0	0	1 1	0 0	0 0	0 0	4
9	0	0	0	0	1 1	0 0	0 0	1 1	1
10	0	0	0	0	1 1	0 0	1 1	0 0	2
11	0	0	0	0	1 1	0 0	1 1	1 1	1
12	0	0	0	0	1 1	1 1	0 0	0 0	3
13	0	0	0	0	1 1	1 1	0 0	1 1	1
14	0	0	0	0	1 1	1 1	1 1	0 0	2
15	0	0	0	0	1 1	1 1	1 1	1 1	1
16	0	0	0	1 1	0 0	0 0	0 0	0 0	5

7.4 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

势能方法的原理

- 势能方法
  - 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
  - 势能方法把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

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

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

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
    - $\phi(D_0)=0$，D_0是00…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操作把t_i个位置成0，实际代价为$c_i=t_i+1$
    - 计算操作i的平摊代价$\alpha_i=c_i+\phi(D_i)-\phi(D_{i-1})$
      - 第i个操作后，计数器中1的个数为b_i
      - 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$O(n)$

7.5 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
- 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

动态表——基本概念

- 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。

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
- $num[T]$表T中的数据项个数
- $size[T]$是T的大小
- 开始时，$num[T]=size[T]=0$

扩张算法

```
算法：TABLE—INSERT(T, x)                    /*复杂的插入操作*/
    If  size[T]=0 Then                       /*开销为常数*/
        获取一个大小为1的表 table[T];
        size[T]←1;
    If  num[T]=size[T] Then                  /*开销取决于size[T]*/
        获取一个大小为 2×size[T]的新表new-table;
        将 table[T]中元素插入new-table;      /*简单插入操作*/
        释放 table[T];
        table[T]←new-table;
        size[T]←2×size[T];
    将 x插入table[T];
    num[T]←num[T]+1
```

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$num[T]<size[T]$, $C_i=1$;
  - 如果$num[T]=size[T]$, $C_i=i$;
- 共有n次操作
  - 最坏情况下,每次都进行扩张操作,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$\sum_{i=1}^{n} c_i \le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j < n + 2n = 3n$

★ 每一操作的平摊代价为$3n/n=3$

初始为空的表上n次插入操作的代价分析

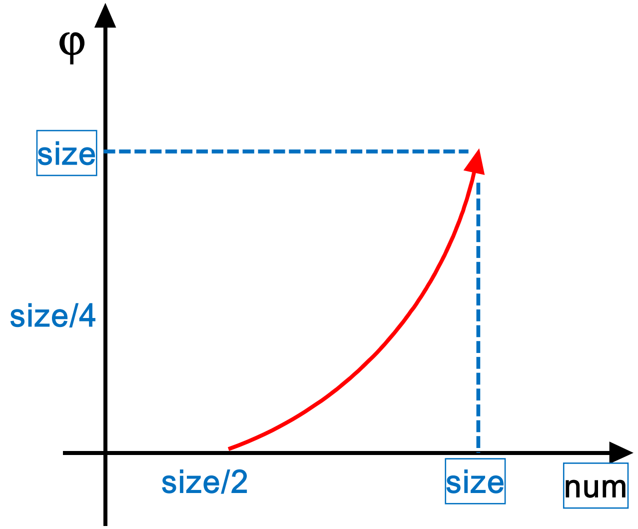
会计方法

- 每次执行TABLE—INSERT平摊代价为3
  - 1支付基本插入操作(伪代码中第10步)的实际代价
  - 1作为自身的存款（余额）
  - 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$3n$

初始为空的表上n次插入操作的代价分析

势能法分析

势能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$3n$

初始为空的表上n次插入操作的代价分析

第i次操作的平摊代价：

- 如果没有引发扩张
  - 则有$size_i(T)=size_{i-1}(T)$
  - $num_i(T)=num_{i-1}(T)+1$
  - 平摊代价$\alpha_i=c_i+\phi_i(T)-\phi_{i-1}(T)=1+((2\times num_i(T)-size_i(T))-(2\times num_{i-1}(T)-size_{i-1}(T))=1+2=3$。
- 如果引发扩张
  - 则有$size_i(T)=2\times size_{i-1}(T)$
  - $num_i(T)=num_{i-1}(T)+1=size_{i-1}(T)+1$
  - 平摊代价$\alpha_i=c_i+\phi_i(T)-\phi_{i-1}(T)=1+2=3$。

动态表的扩张与收缩

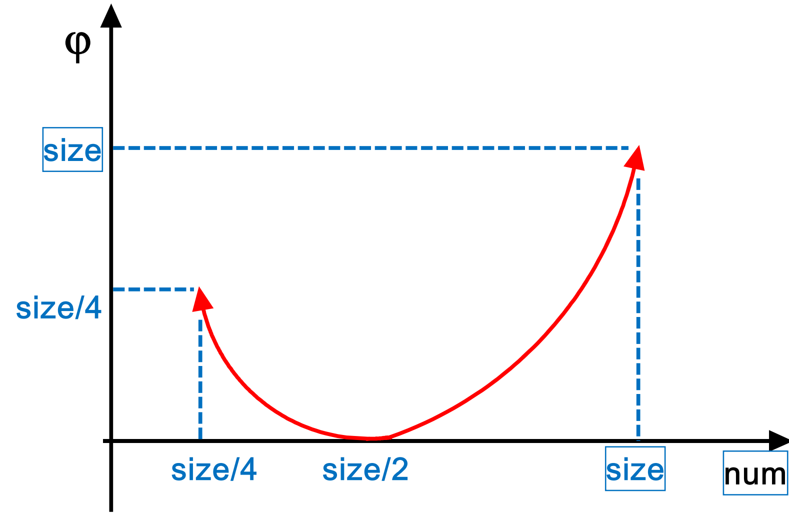
- Table-Delete：将指定的数据对象从表中删除
- 表的收缩：当动态表的装载因子很小时，对表进行收缩
  理想情况下，我们希望：
  - 表具有一定的丰满度
  - 表的操作序列的复杂度是线性的
- 表的收缩策略
  - 表的装载因子小于1/2时，收缩表为原表的一半
  - $n=2^k$,考察下面的一个长度为n的操作序列：
    - 前$n/2$个操作是插入，后跟I D D I I D D I I…
    - 每次扩张和收缩的代价为O(n)，共有O(n)扩张或收缩
    - 总代价为$O(n^2)$，而每一次操作的平摊代价为$O(n)$--每个操作的平摊代价太高

动态表的扩张与收缩

- 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

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

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

平摊代价的计算

- 第i次操作的平摊代价：$\alpha_i = c_i + \phi(T_i)\text{-}\phi(T_i\text{-}1)$
  - 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$O(n)$

6.6 斐波那契堆性能平摊分析

- 斐波那契堆及其基本操作
- 应用平摊分析得出斐波那契堆的操作代价
- 运用平摊分析进行算法设计的尝试
- 目标
  - ✓ 深入理解势能方法
  - ✓ 运用平摊分析分析算法复杂性
  - ✓ 积累一种有用的数据结构

堆的性能比较

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

堆的性能比较

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

n -堆中存储的元素个数　　　　平摊分析

插入、删除不可能均在*O*(1)时间内完成．为什么？

斐波那契堆的过去和现状

- 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

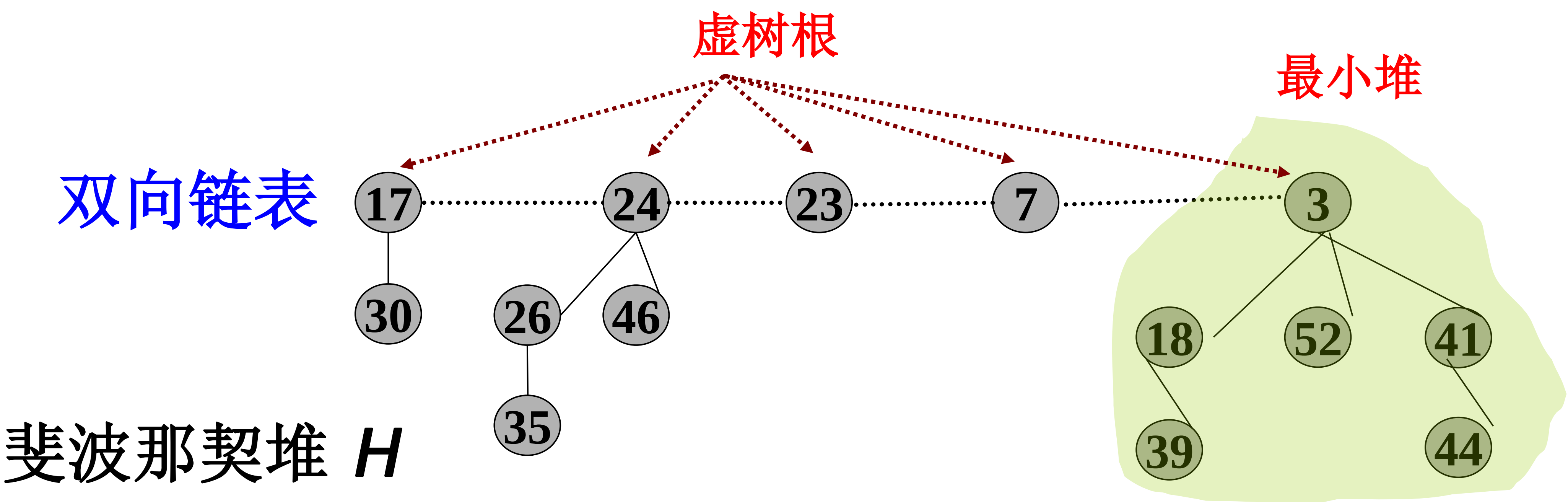
1.Strict fibonacci heaps， Symposium on the Theory of Computing , pp 1177-1184, 2012

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，每棵树均是一个最小堆
  - 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)

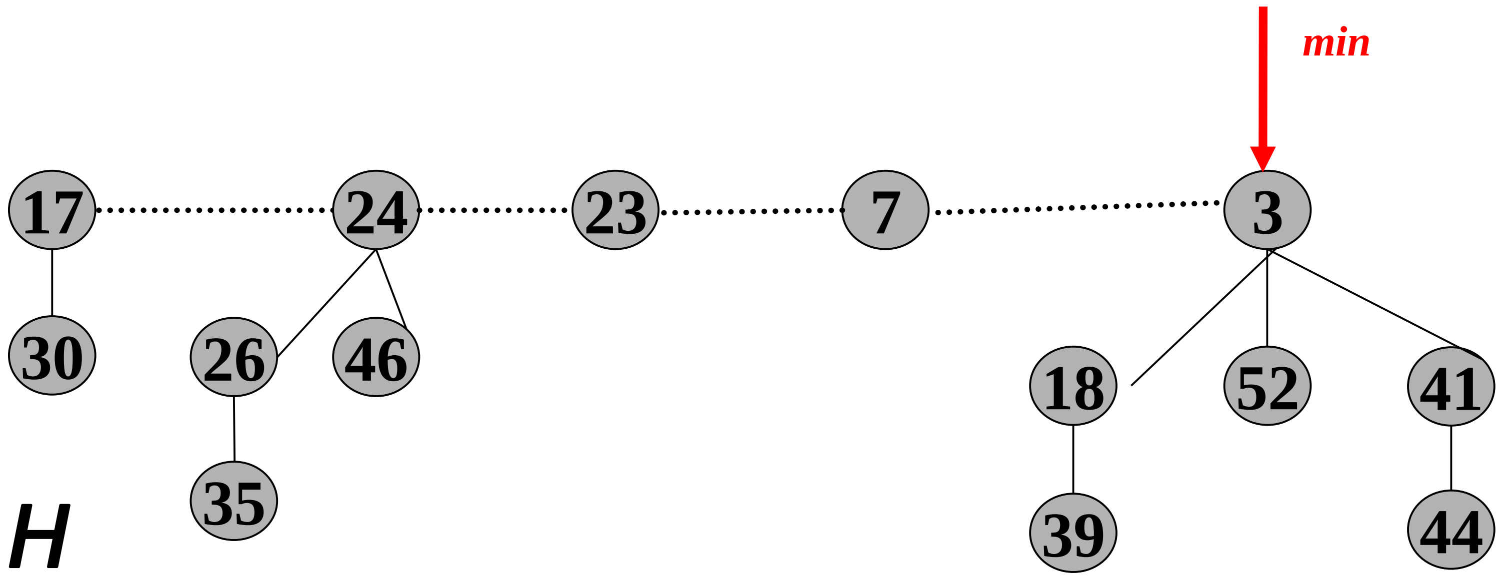
斐波那契堆：

– 一系列树，每棵树均是一个极小堆

– 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

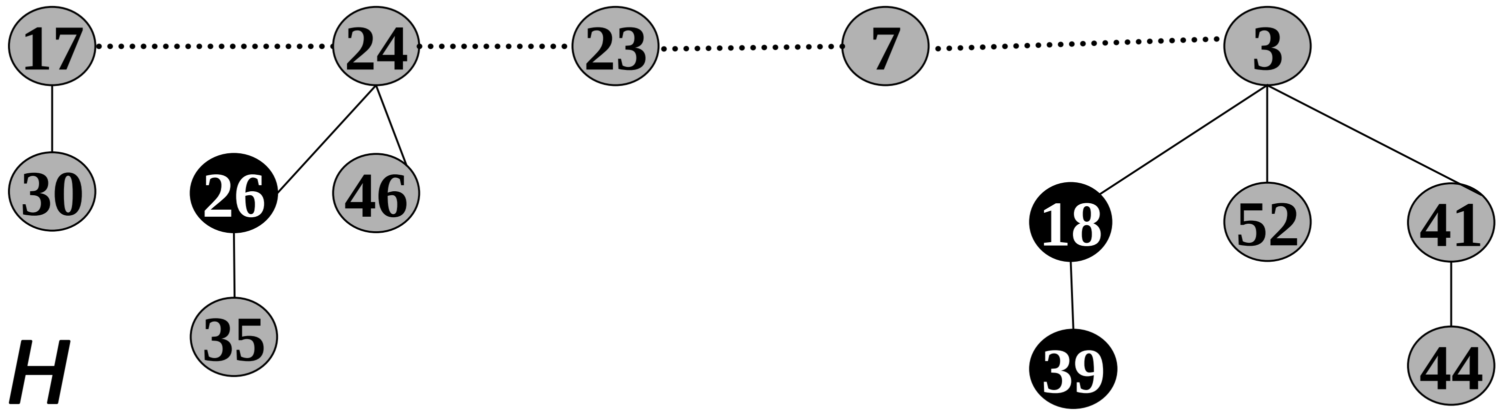


斐波那契堆 *H*

斐波那契堆的结构(3)

斐波那契堆：

– 一系列树，每棵树均是一个极小堆
– 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



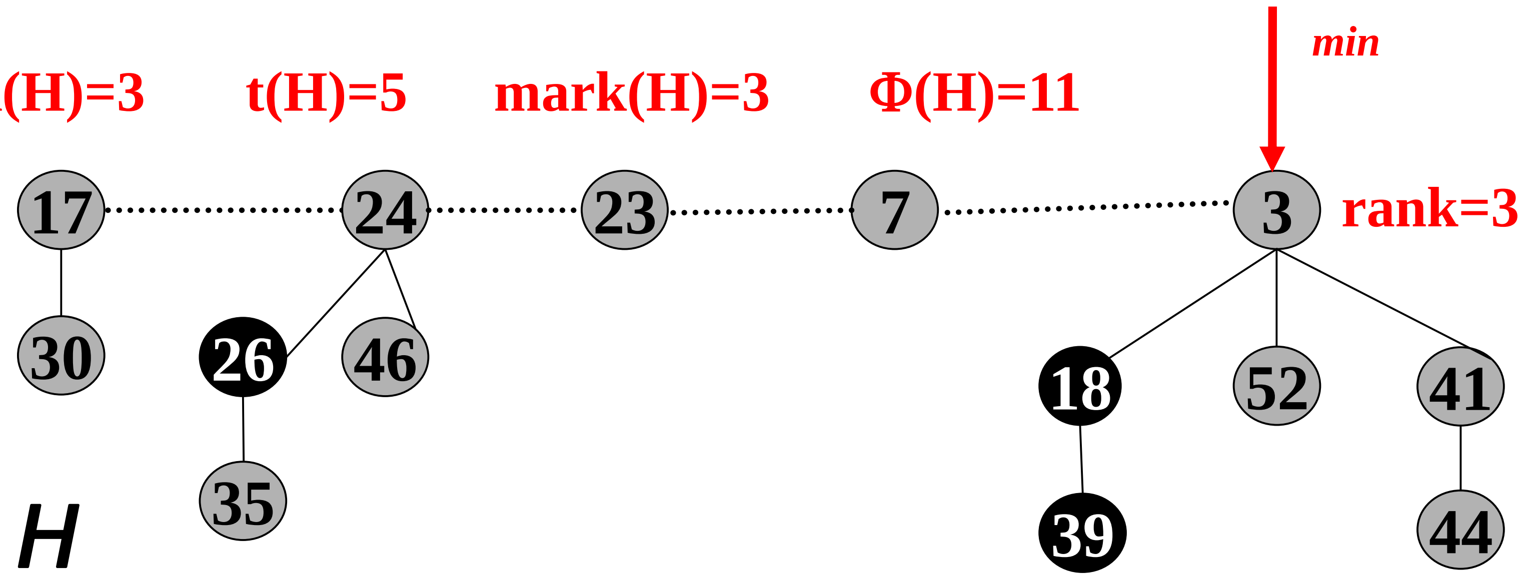
斐波那契堆 *H*

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
- $rank(x)$ ：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
- $t(H)$ ：堆H中树的棵数
- $mark(H)$ ：堆H中被标记顶点的个数
- $\Phi(H)=t(H)+2\cdot mark(H)$：势能函数

n=14　rank(H)=3　t(H)=5　mark(H)=3　Φ(H)=11

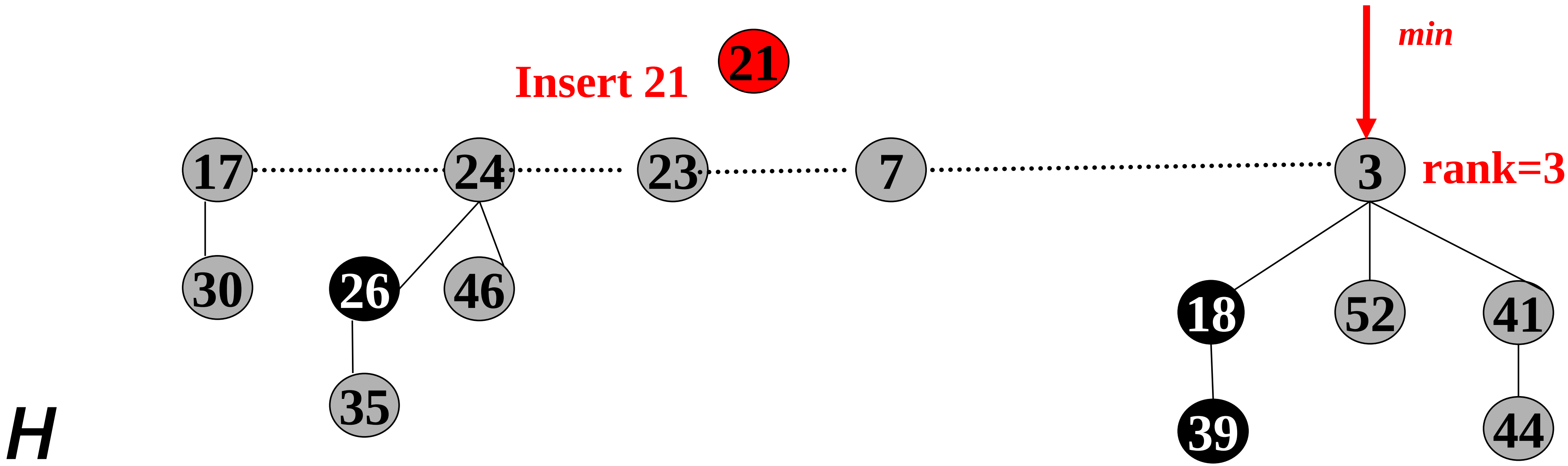


斐波那契堆 H

插入操作Insert

插入操作的过程：

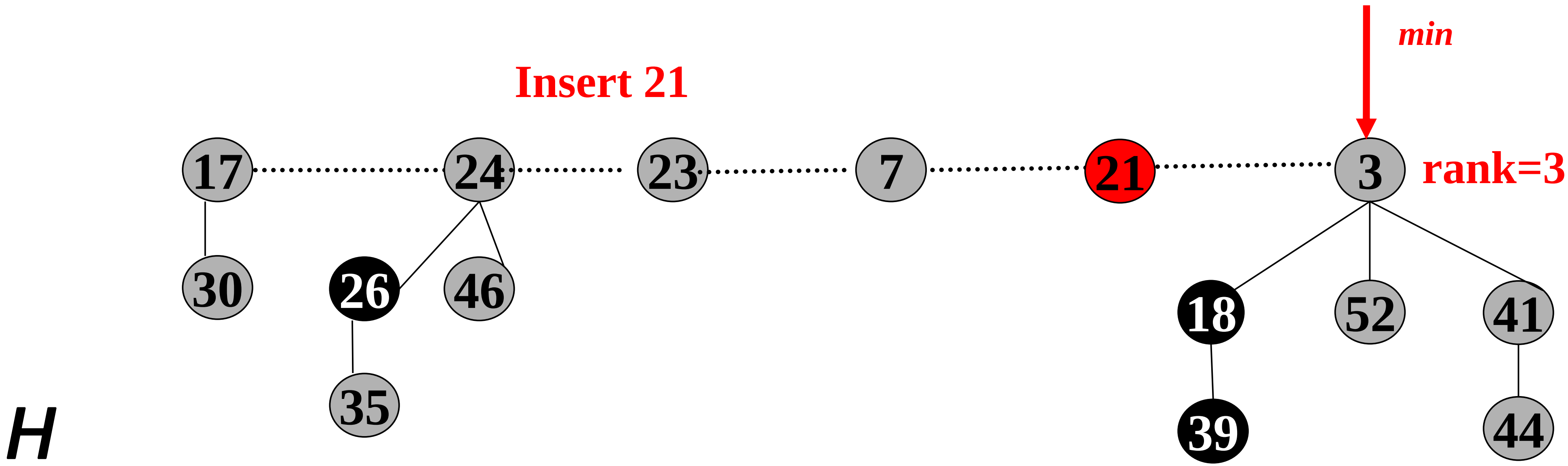
- 以新结点为根，创建一棵新树
- 将新的树插入树根双向链表
- 如有必要，更新最小元素指针



插入操作Insert

插入操作的过程：

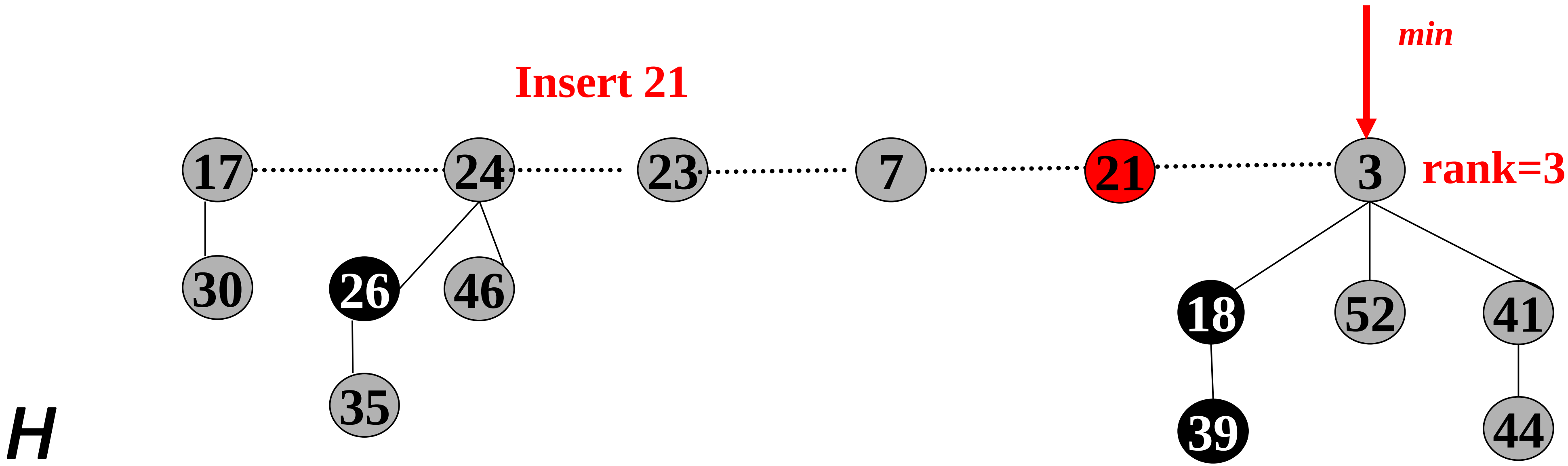
- 以新结点为根，创建一棵新树
- 将新的树插入树根双向链表
- 如有必要，更新最小元素指针



插入操作Insert分析

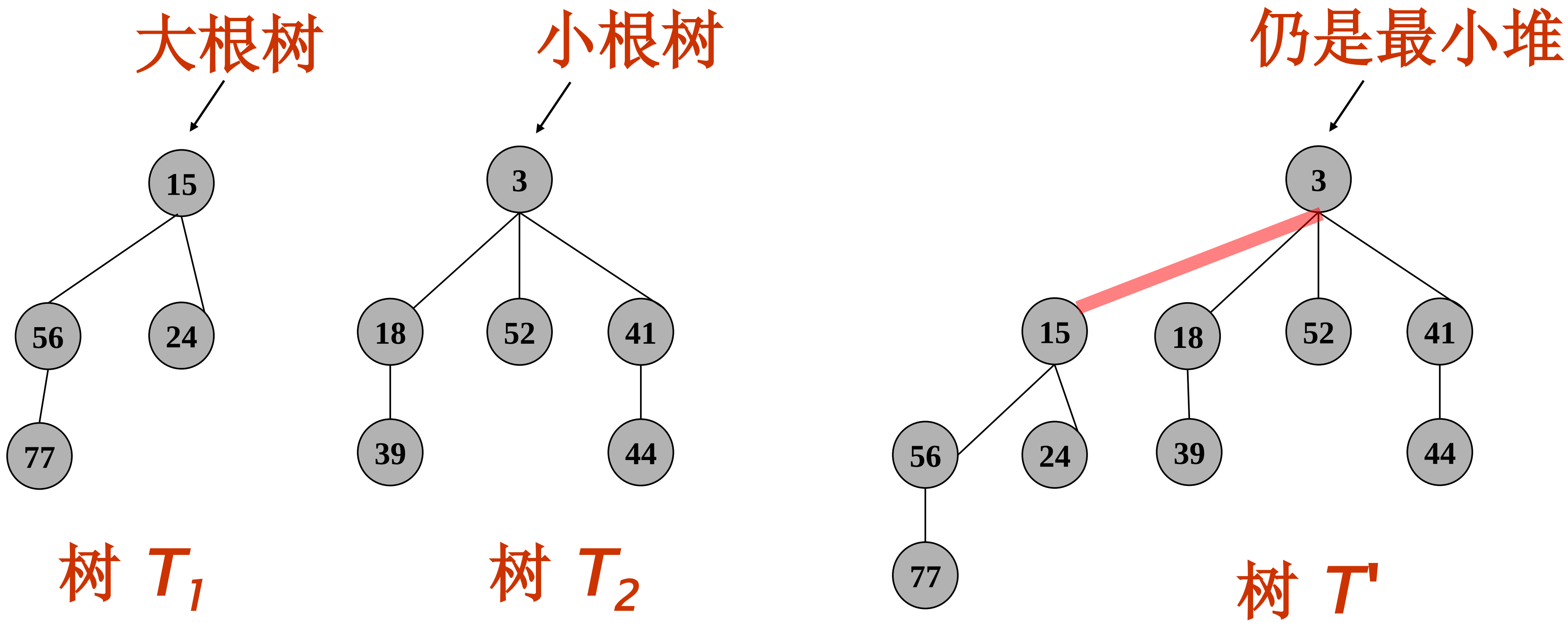
插入操作的代价分析： （势能函数$\Phi(H)=t(H)+2\cdot mark(H)$）

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



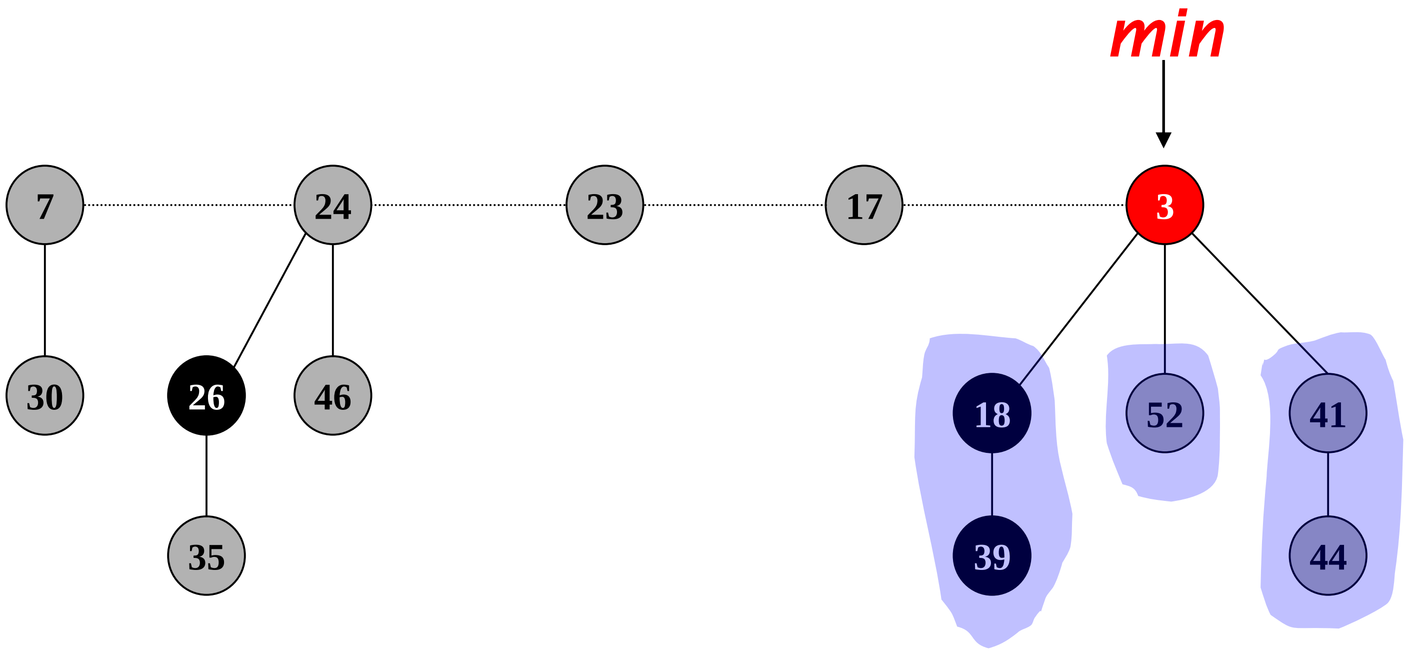
删除堆顶元素Delete_min(1)

- 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)



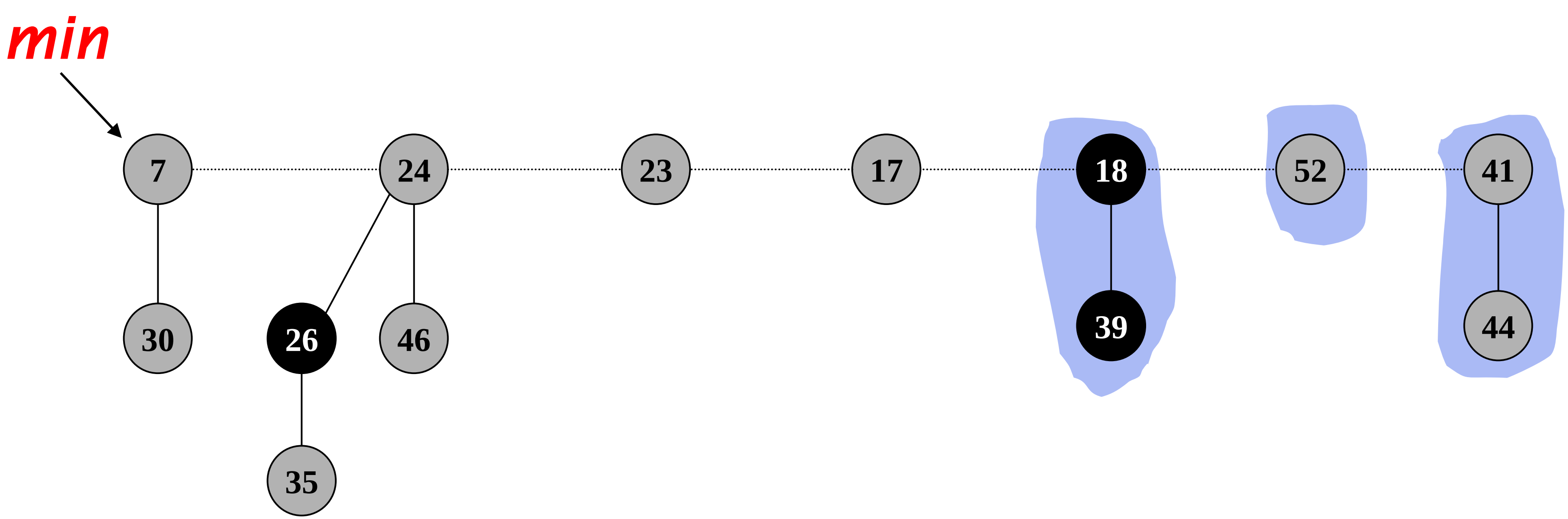
删除堆顶元素Delete_min(2)

- **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



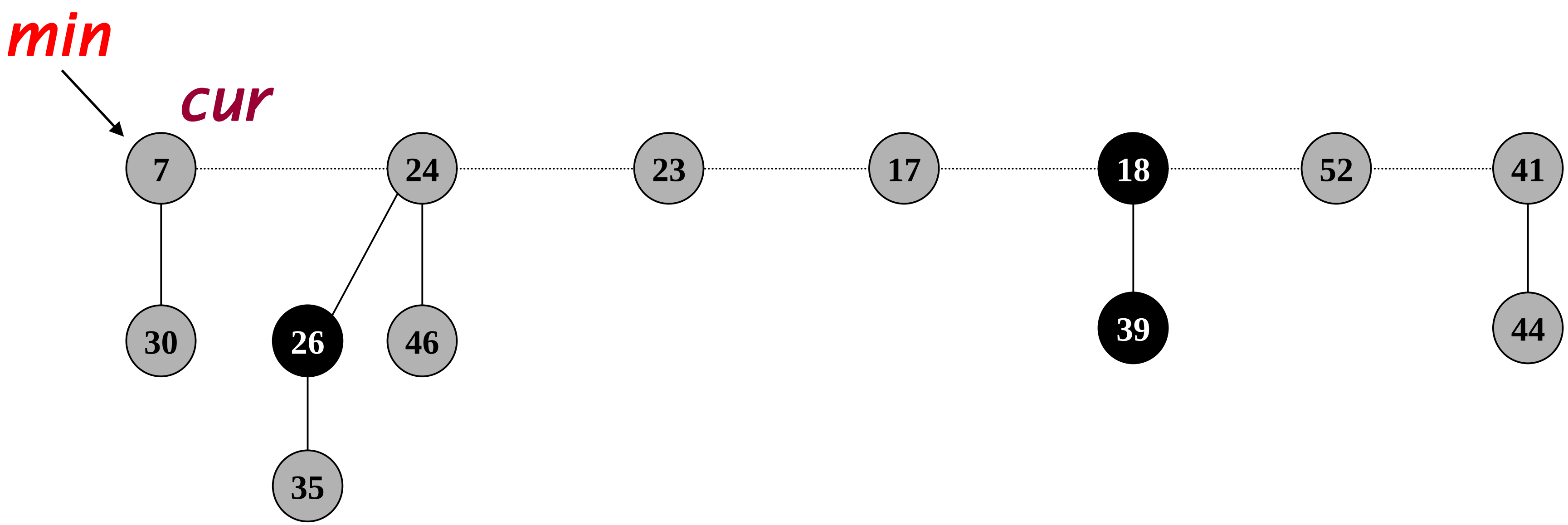
删除堆顶元素Delete_min(3)

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



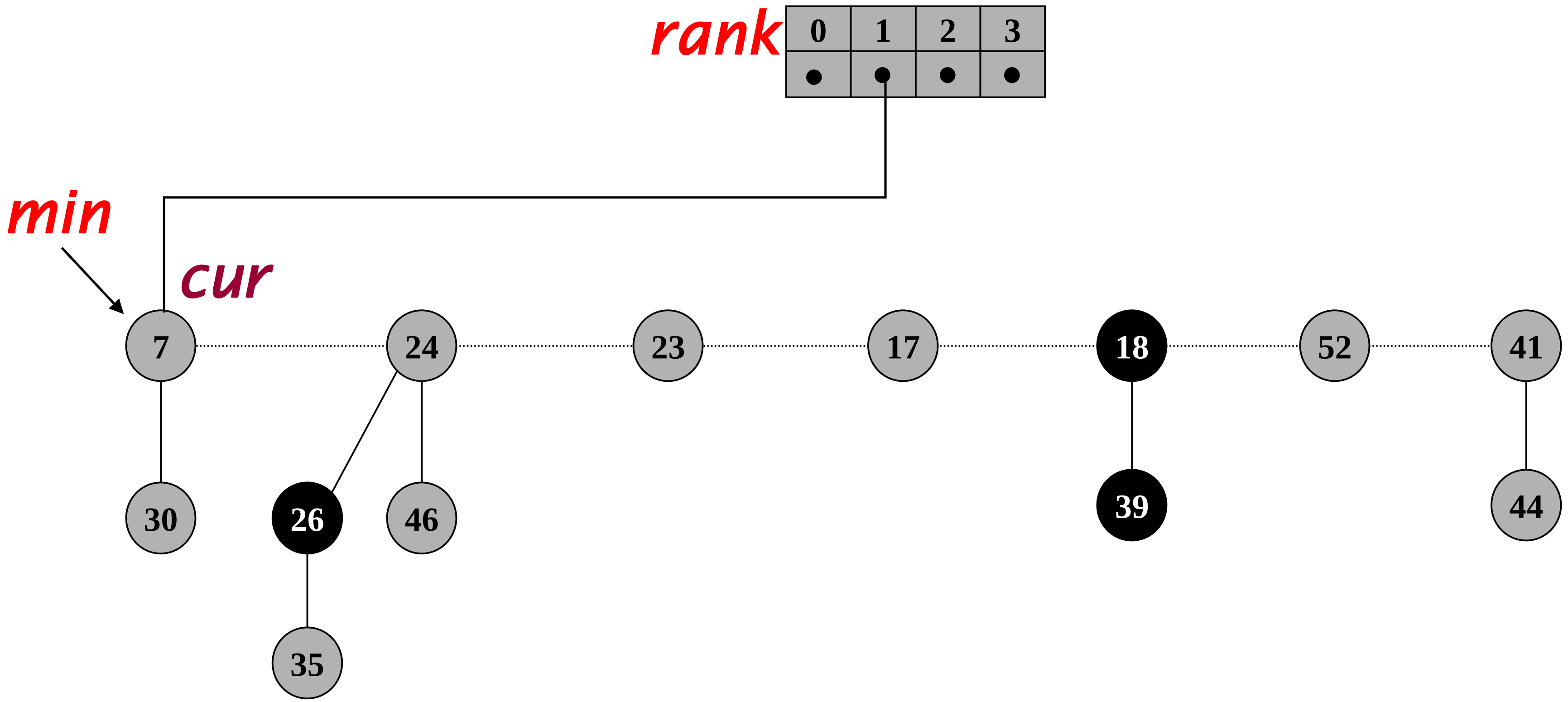
删除堆顶元素Delete_min(4)

- **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



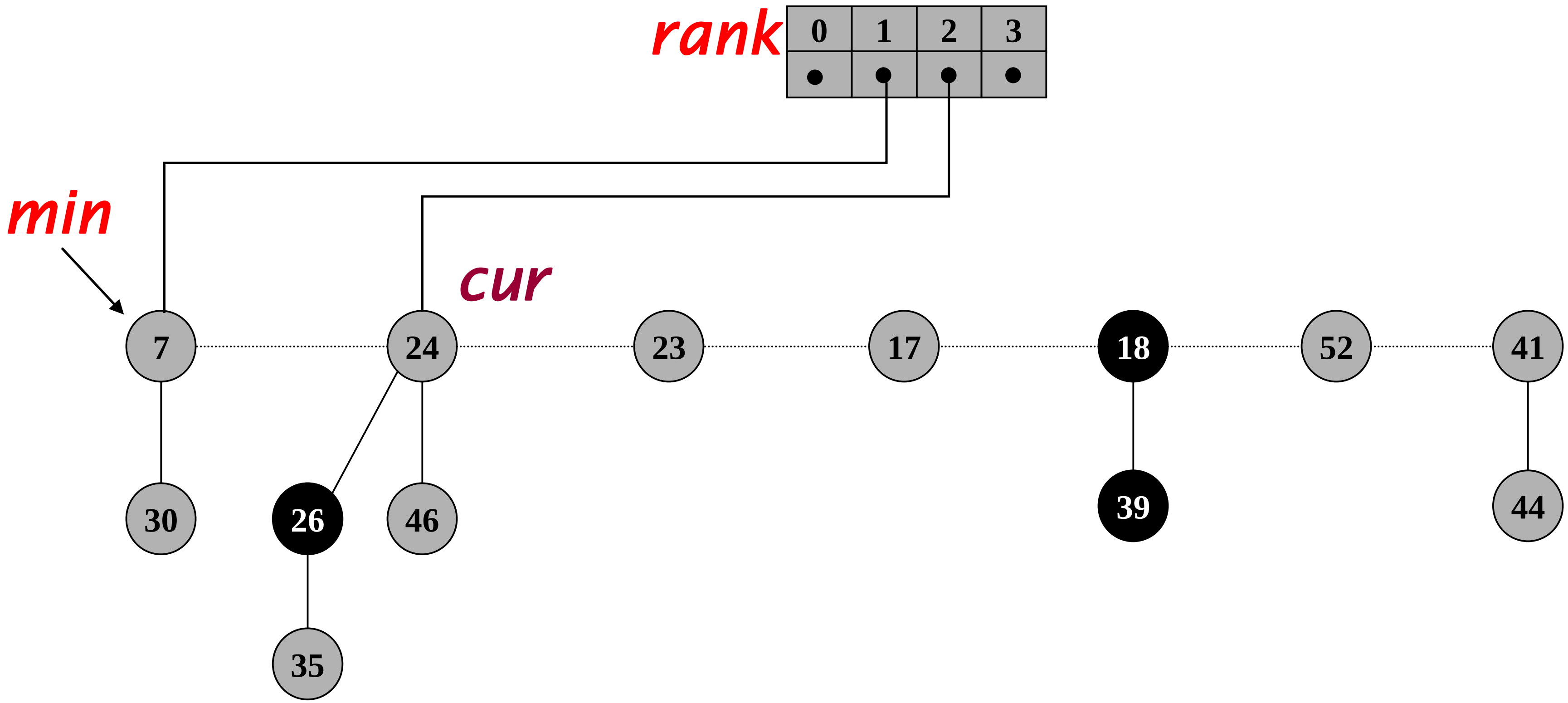
删除堆顶元素Delete_min(5)

- **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



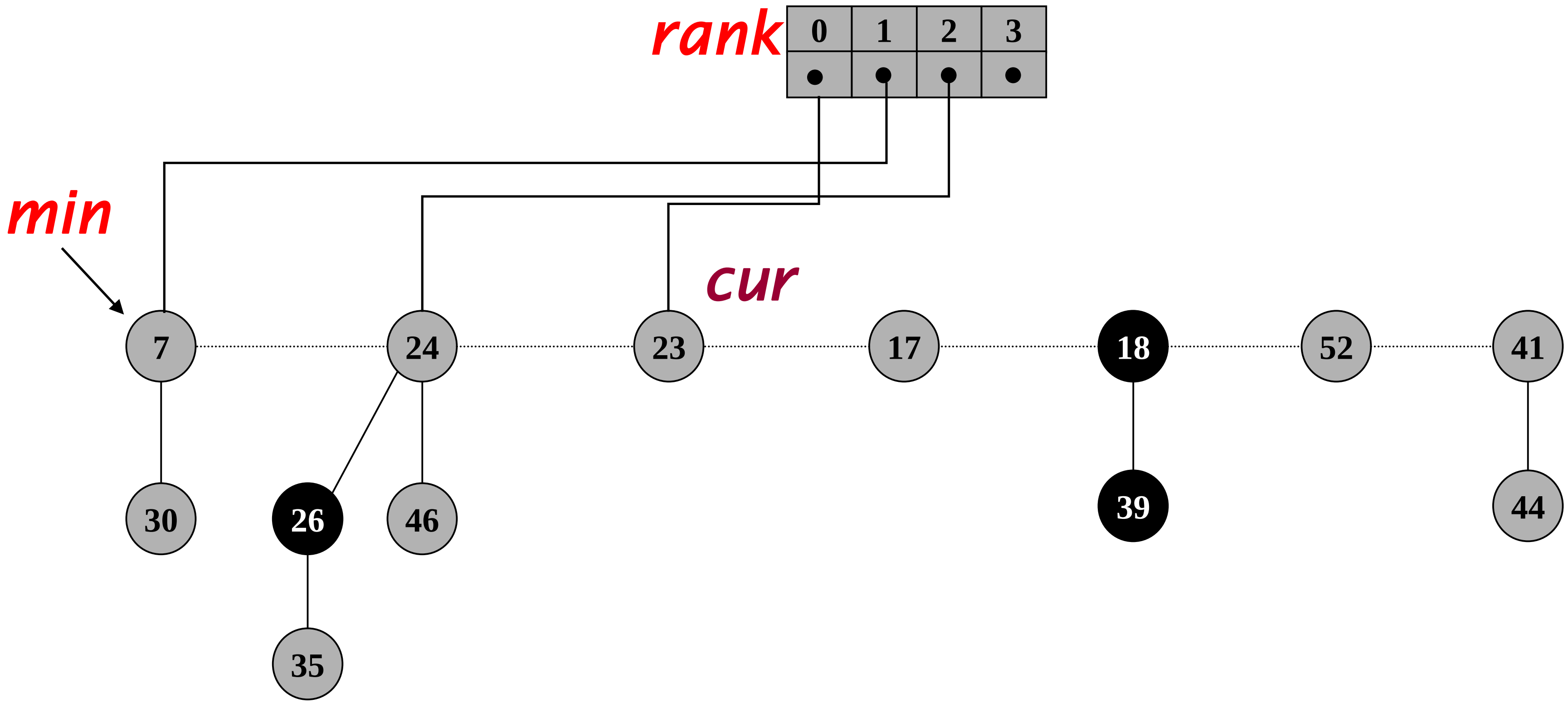
删除堆顶元素Delete_min(5)

- **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



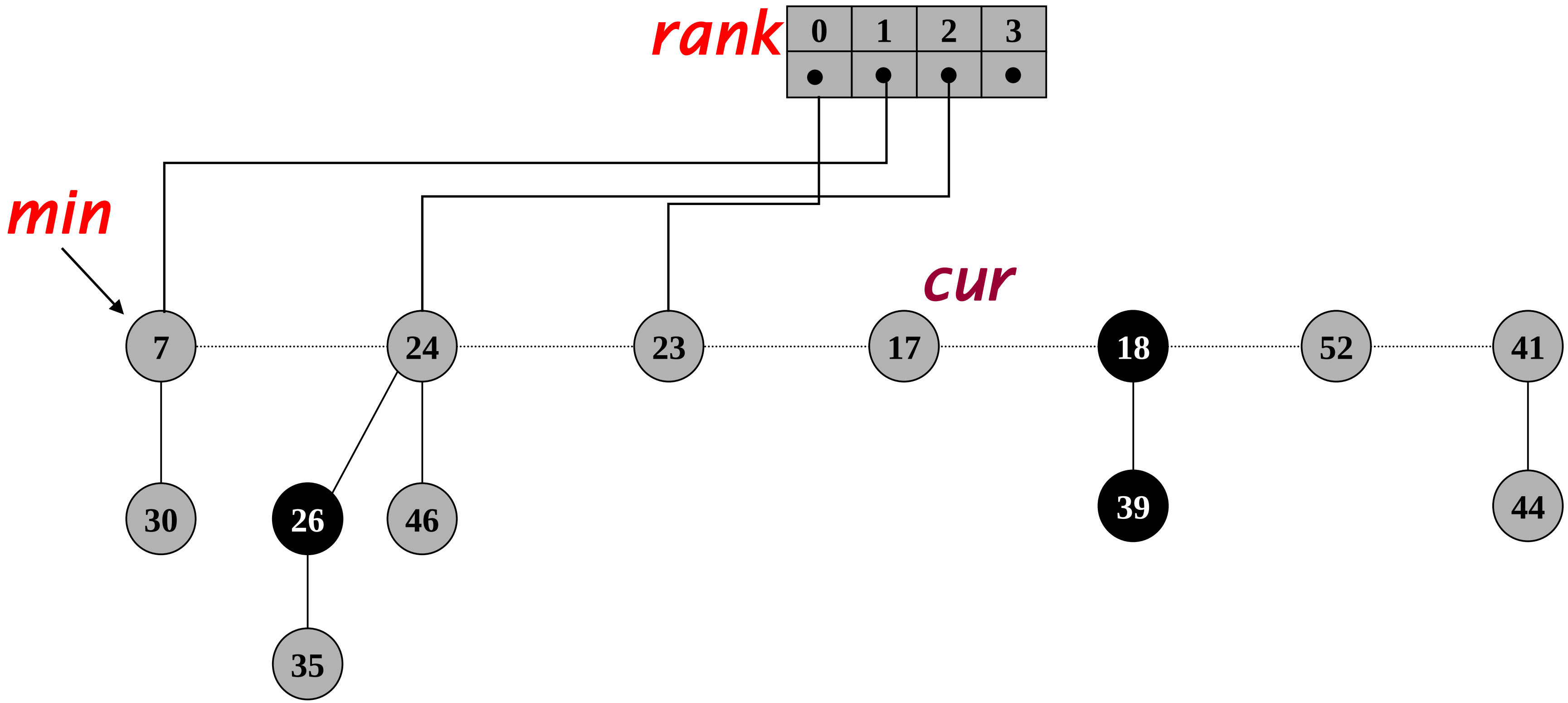
删除堆顶元素Delete_min(5)

- **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



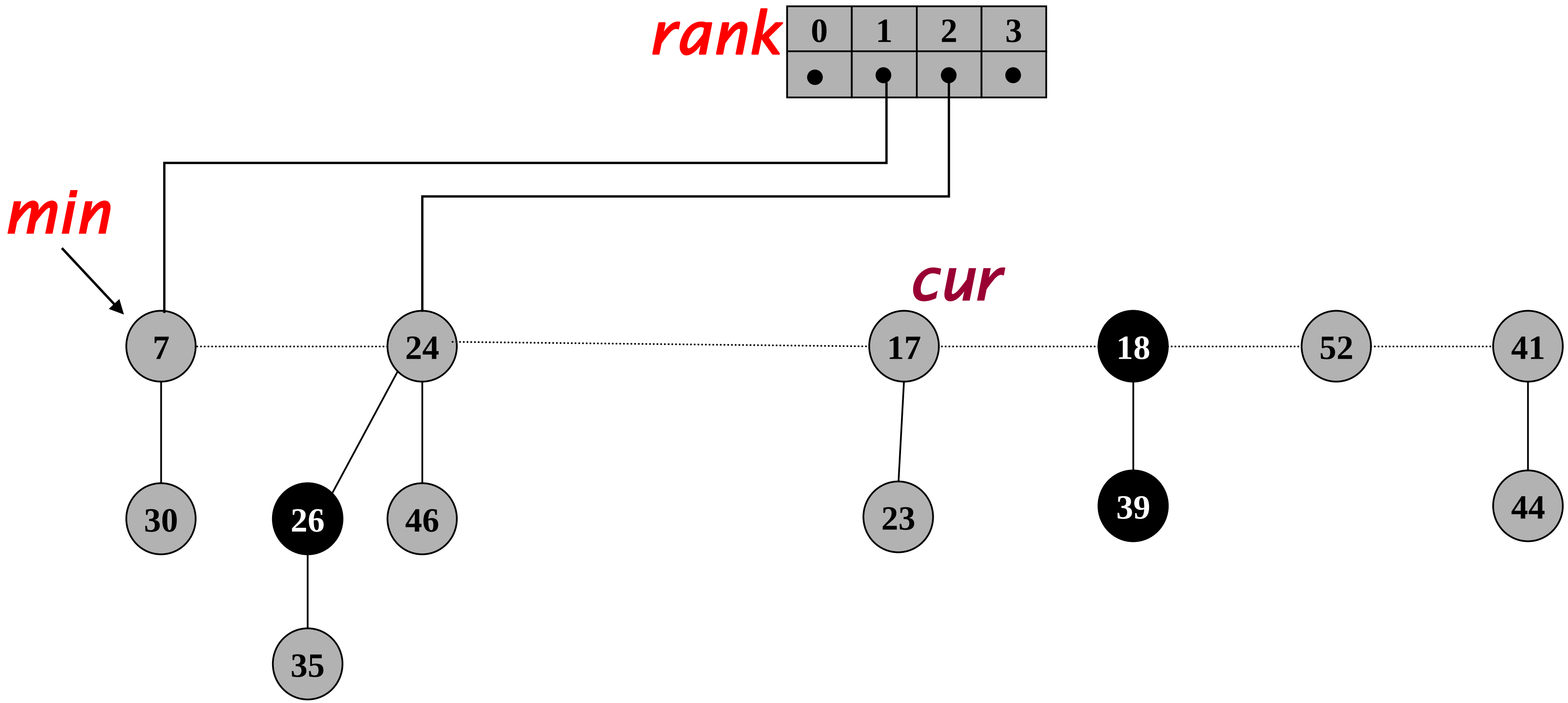
删除堆顶元素Delete_min(5)

- **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



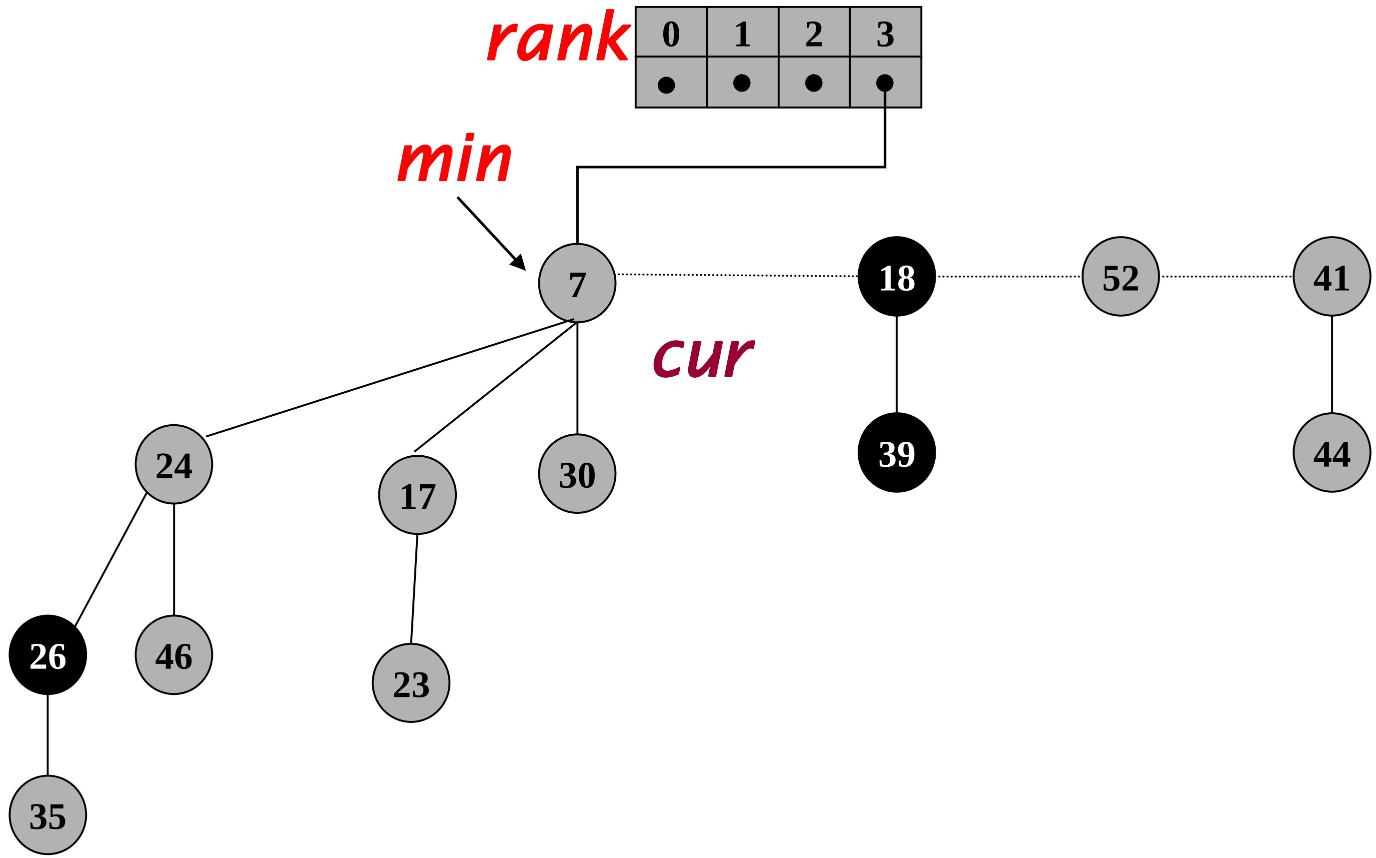
删除堆顶元素Delete_min(5)

- **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



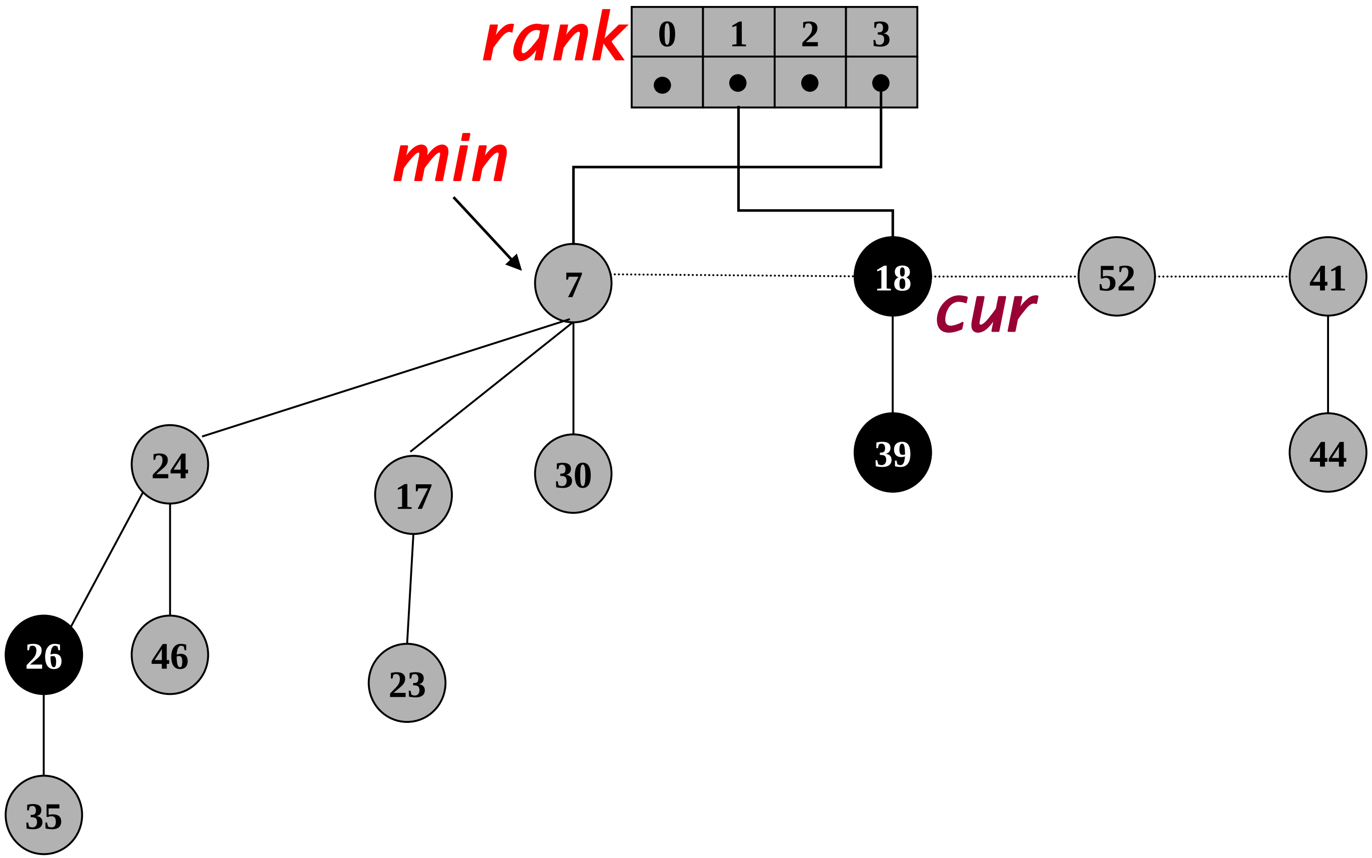
删除堆顶元素Delete_min(5)

- **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



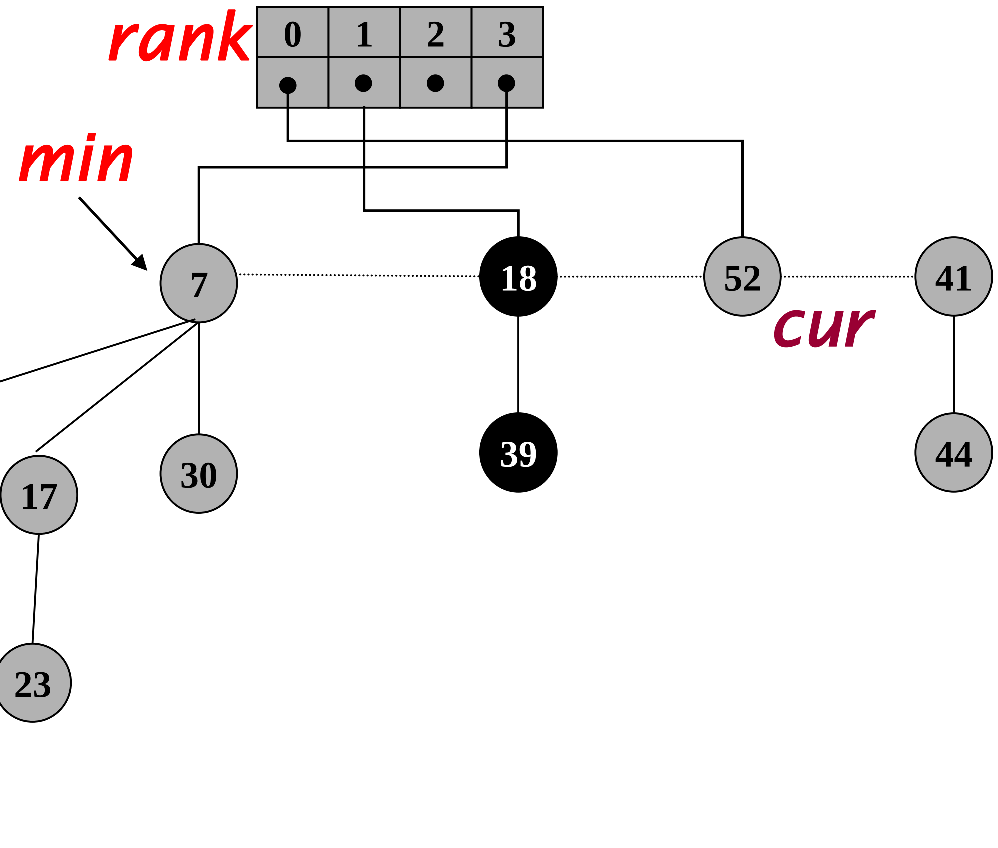
删除堆顶元素Delete_min(5)

- **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

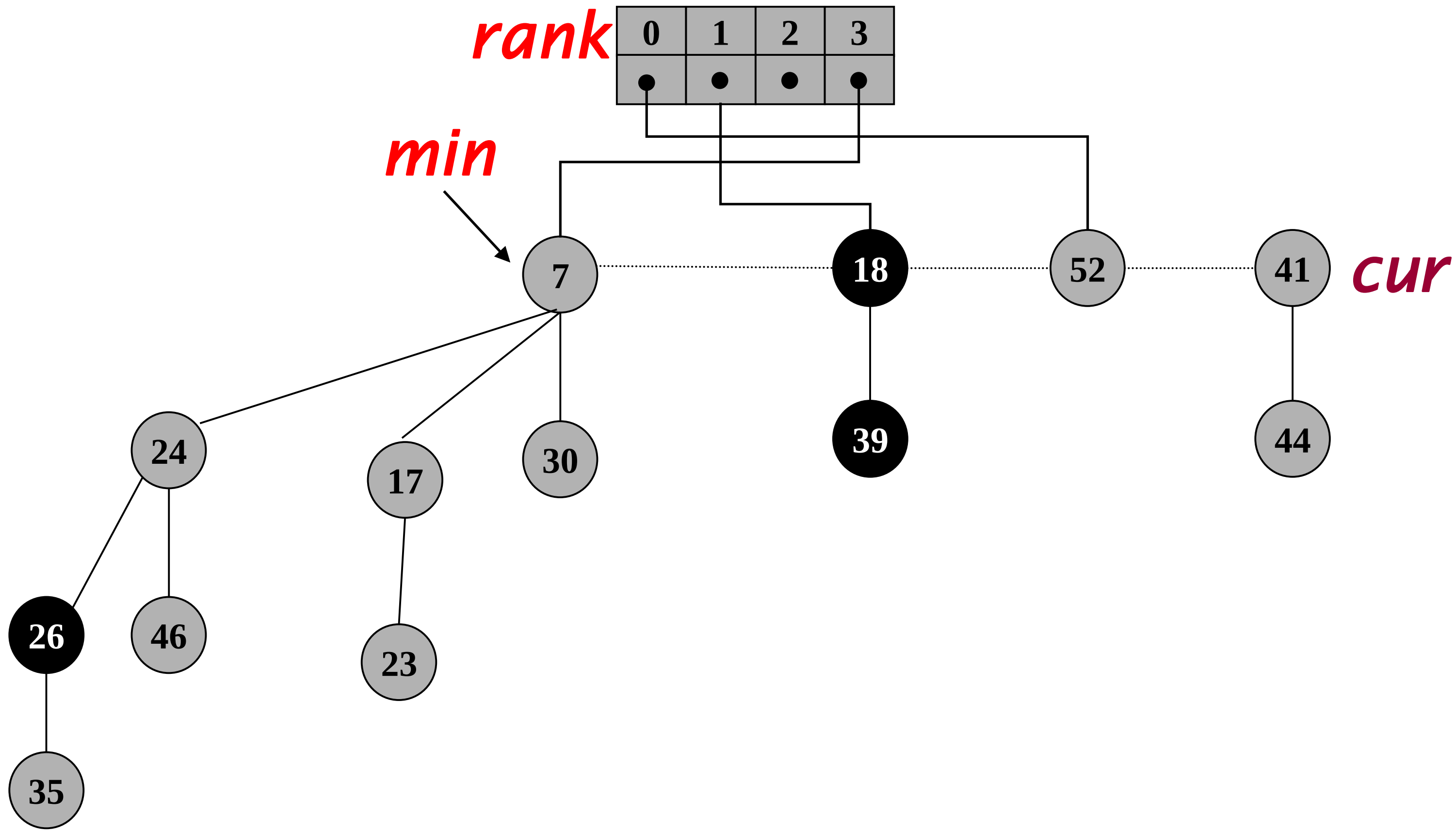


删除堆顶元素Delete_min(5)

- **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

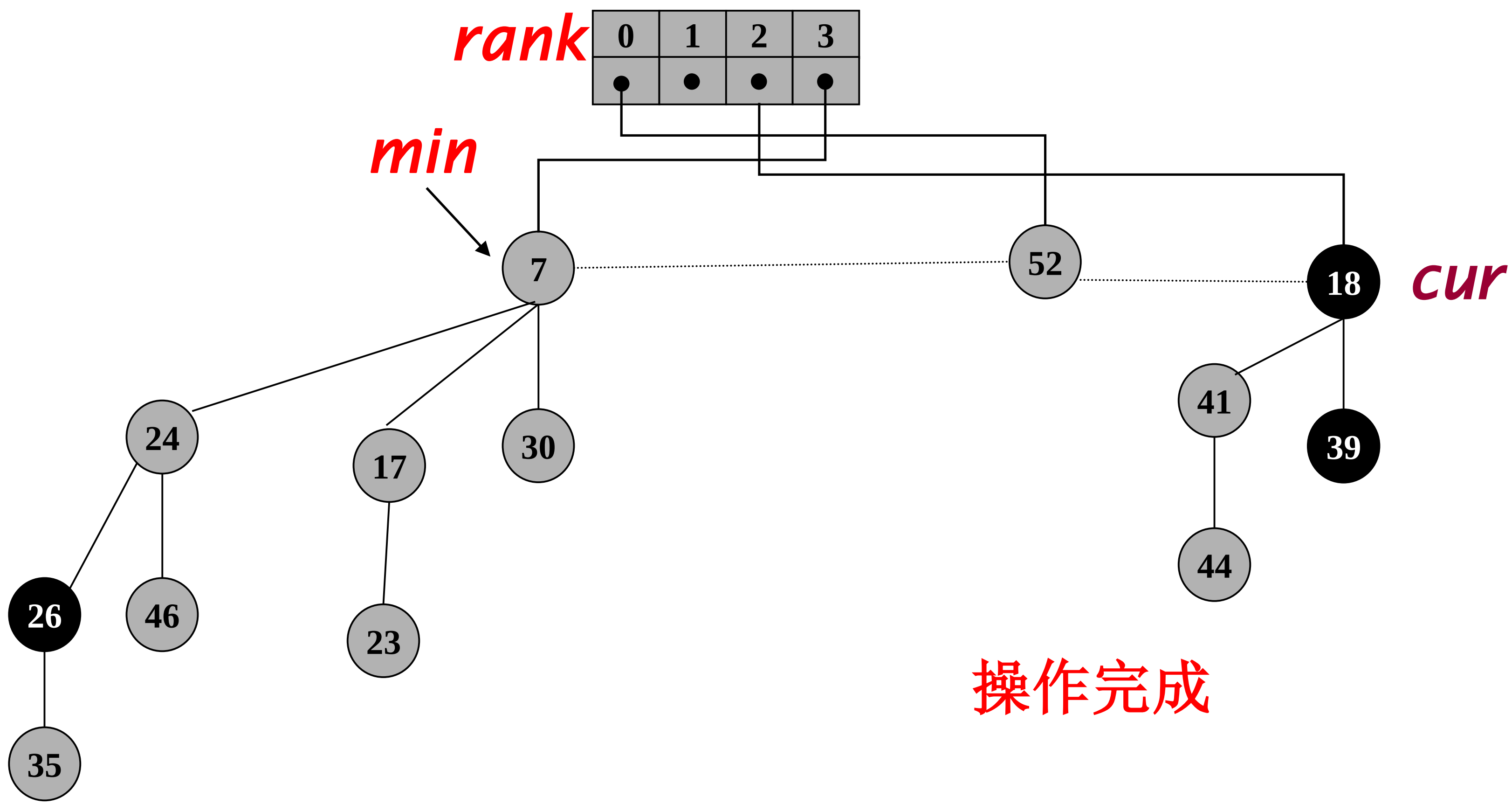


- **Delete_min操作过程**
  - 删除最小元素；将其所有孩子链如双向链表，更新最小元素指针
  - 合并双向链表中的树根，使得没有树根具有相同rank



删除堆顶元素Delete_min(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

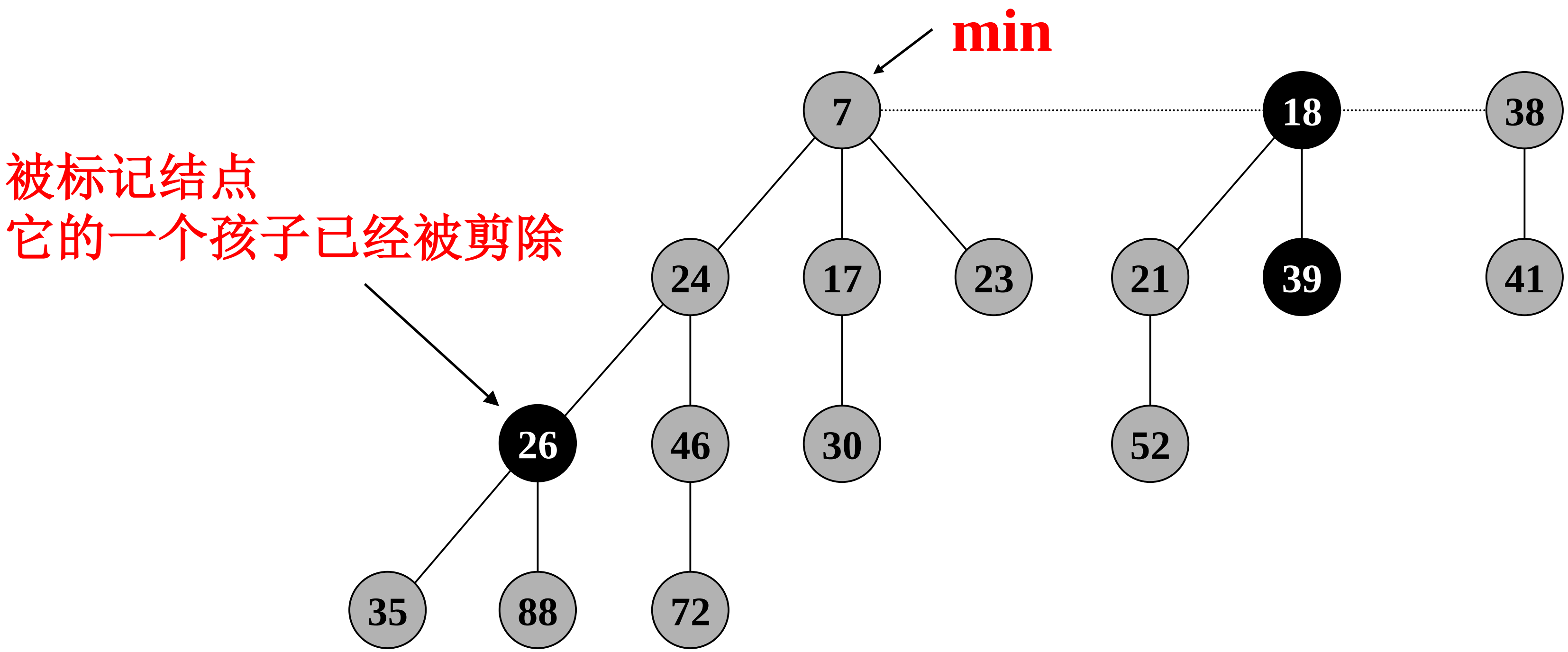
删除操作的代价分析：（势能函数 $\Phi(H)=t(H)+2\cdot mark(H)$）

- 实际开销$O(rank(H)) + O(t(H))$
  - $rank(H)$ 时间内将最小元素结点的孩子插入双向链表
  - $rank(H) + t(H)$ 时间内更新最小元素指针+维护双向链表
- 平摊代价$O(rank(H))$
  - 操作完成后的堆记为H',操作前的堆记为H
  - $t(H') \leq rank(H) + 1$,因为H'中没有两个树具有相同$rank$
  - $\Delta\Phi(H) \leq rank(H) + 1 - t(H)$
- *Delete_min*操作的平摊代价？
  - 这个代价很好
  - $rank(H)\leq\log n$
  - 减小键值操作也将确保$rank(H)\leq\log n$

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”，一旦剪除某个结点的第二个孩子结点，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）

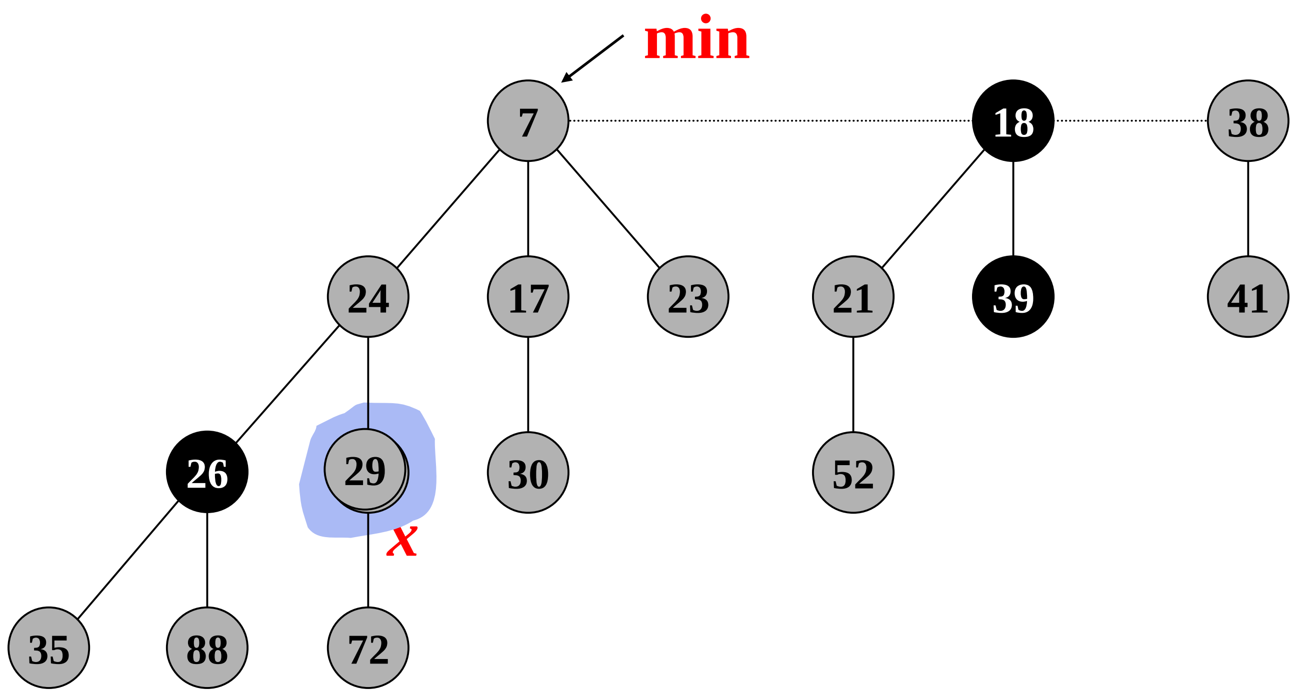


键值减小操作Decrease_key(2)

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

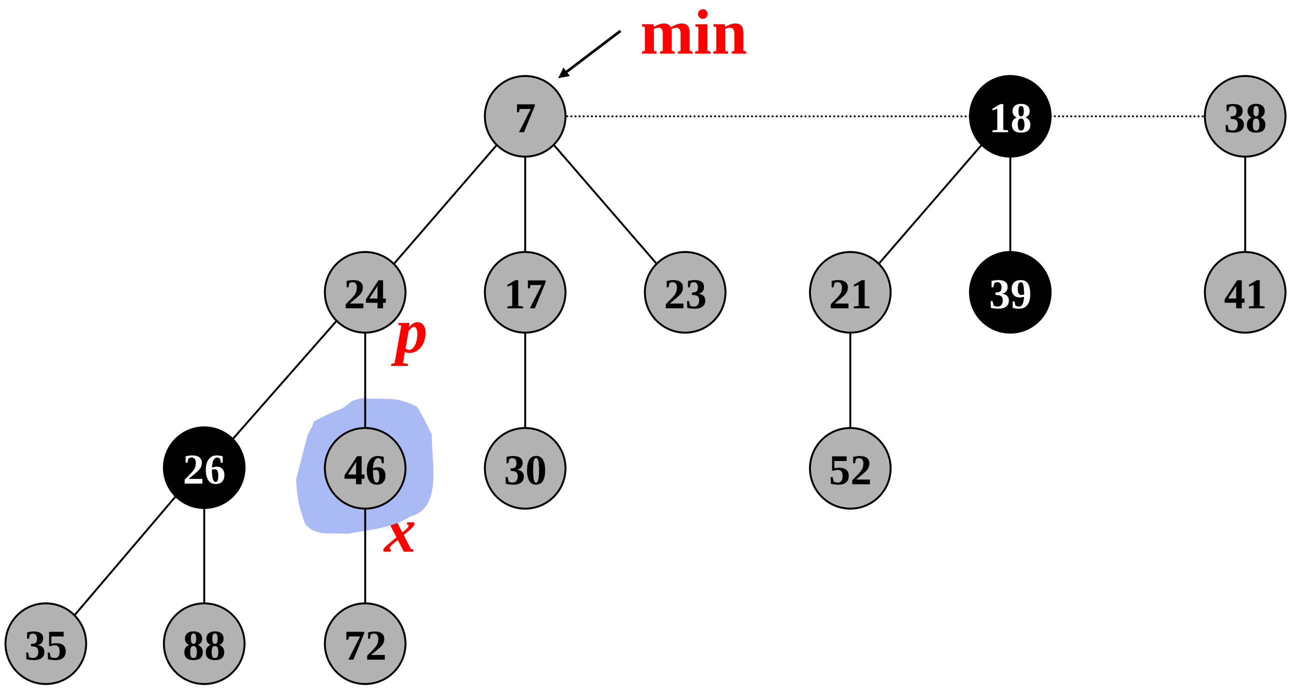


键值减小操作Decrease_key(3)

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）

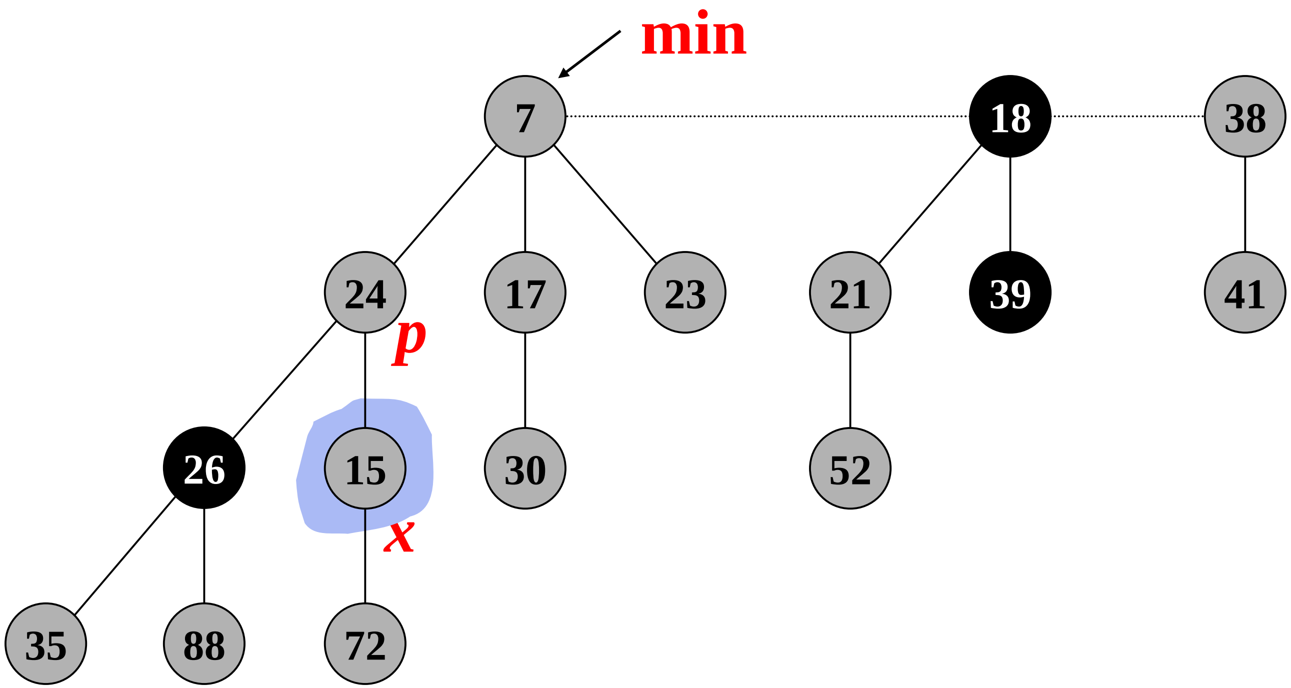


键值减小操作Decrease_key(3)

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）

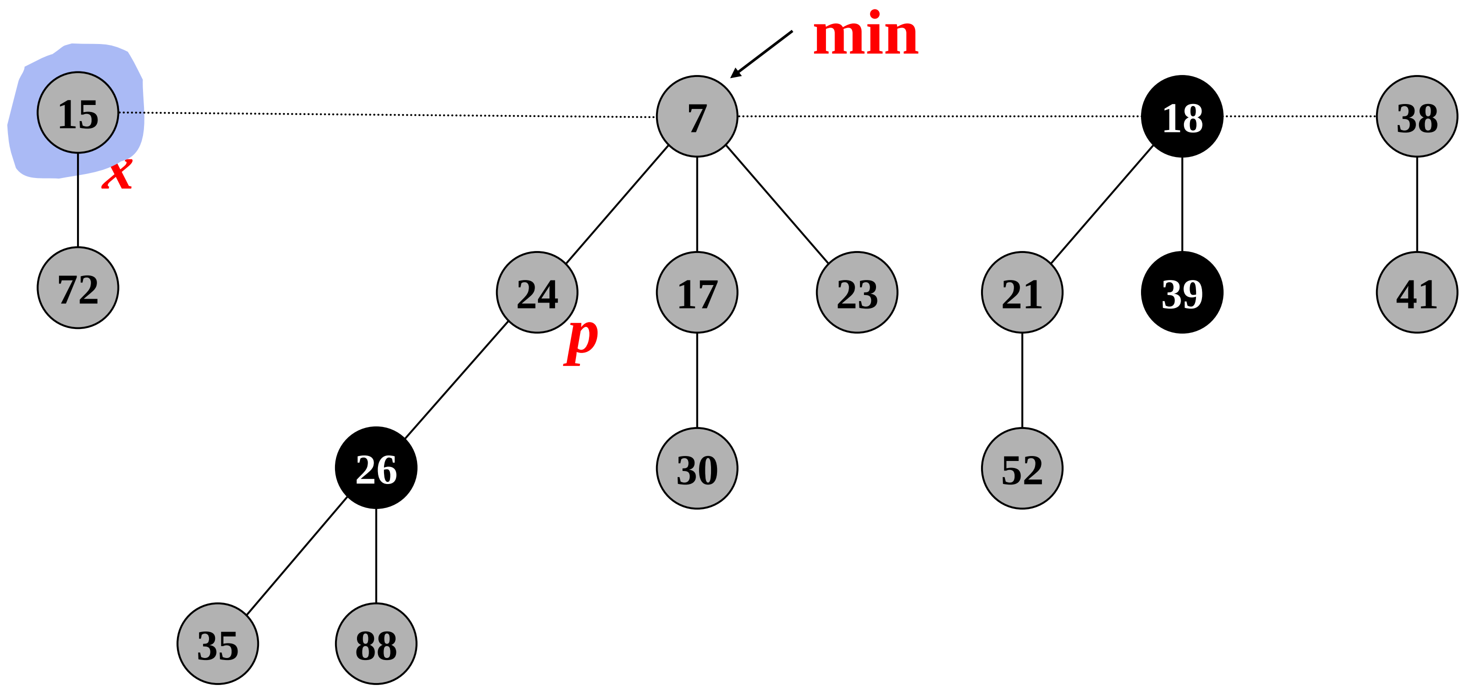


键值减小操作Decrease_key(5)

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）

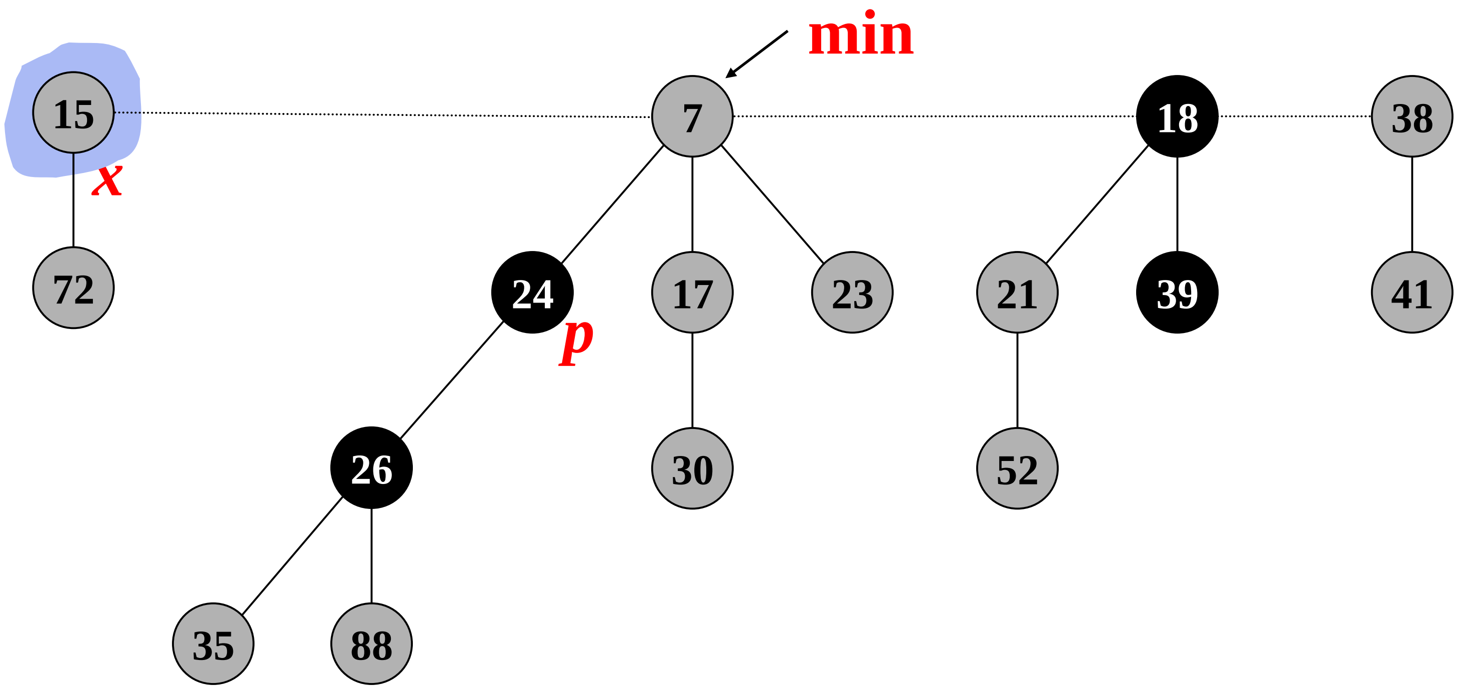


键值减小操作Decrease_key(6)

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）

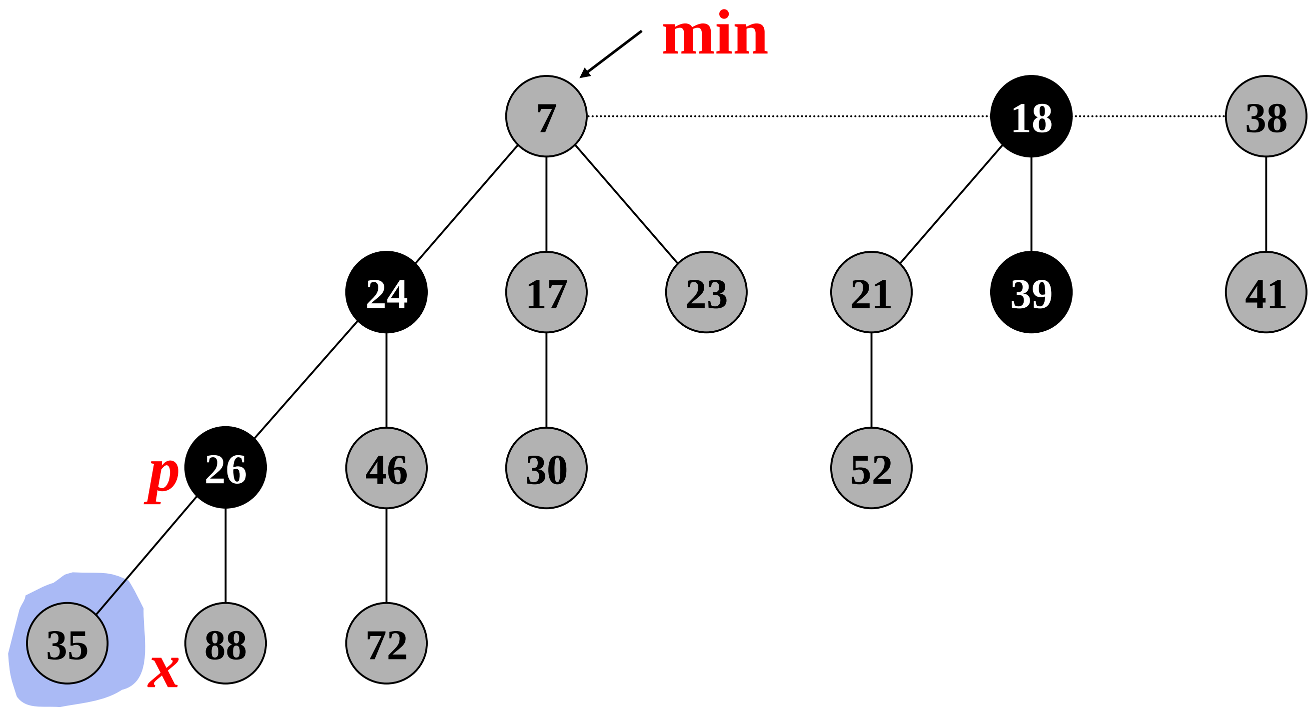


键值减小操作Decrease_key(7)

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）

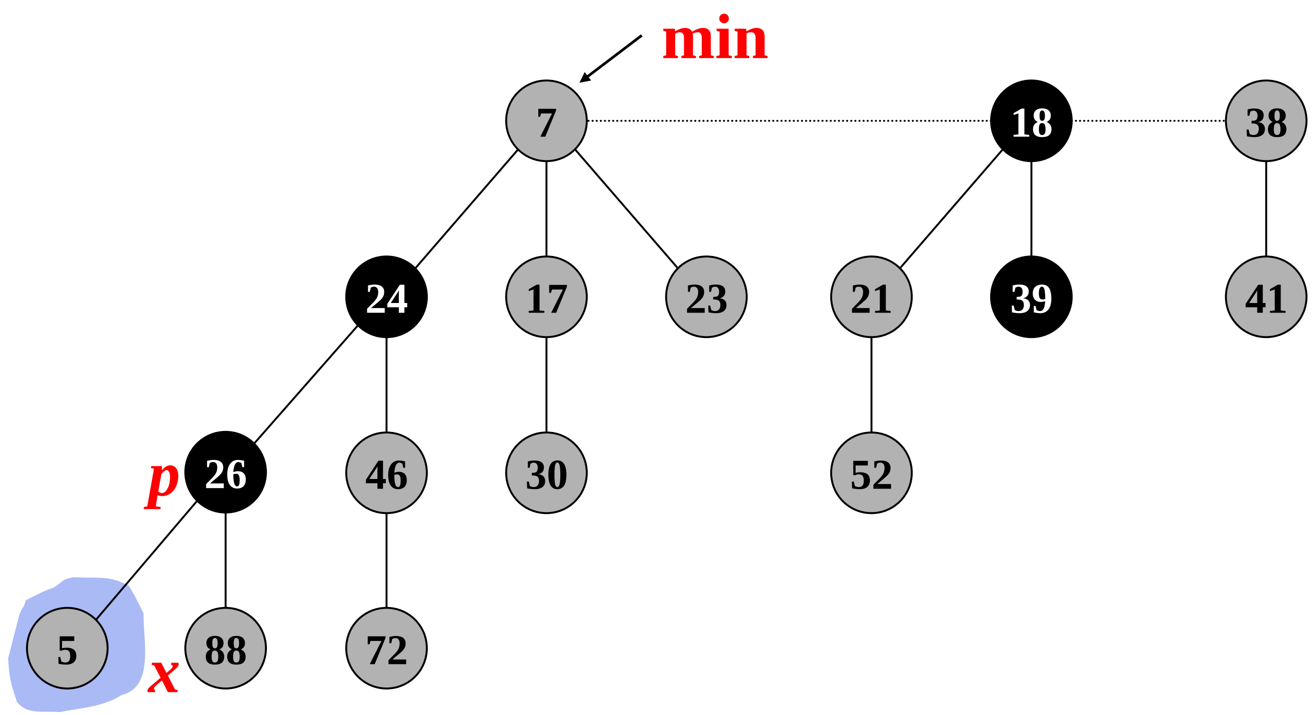


键值减小操作Decrease_key(8)

直键值减小操作作用于结点*x*（例如，键值从35减小为5）

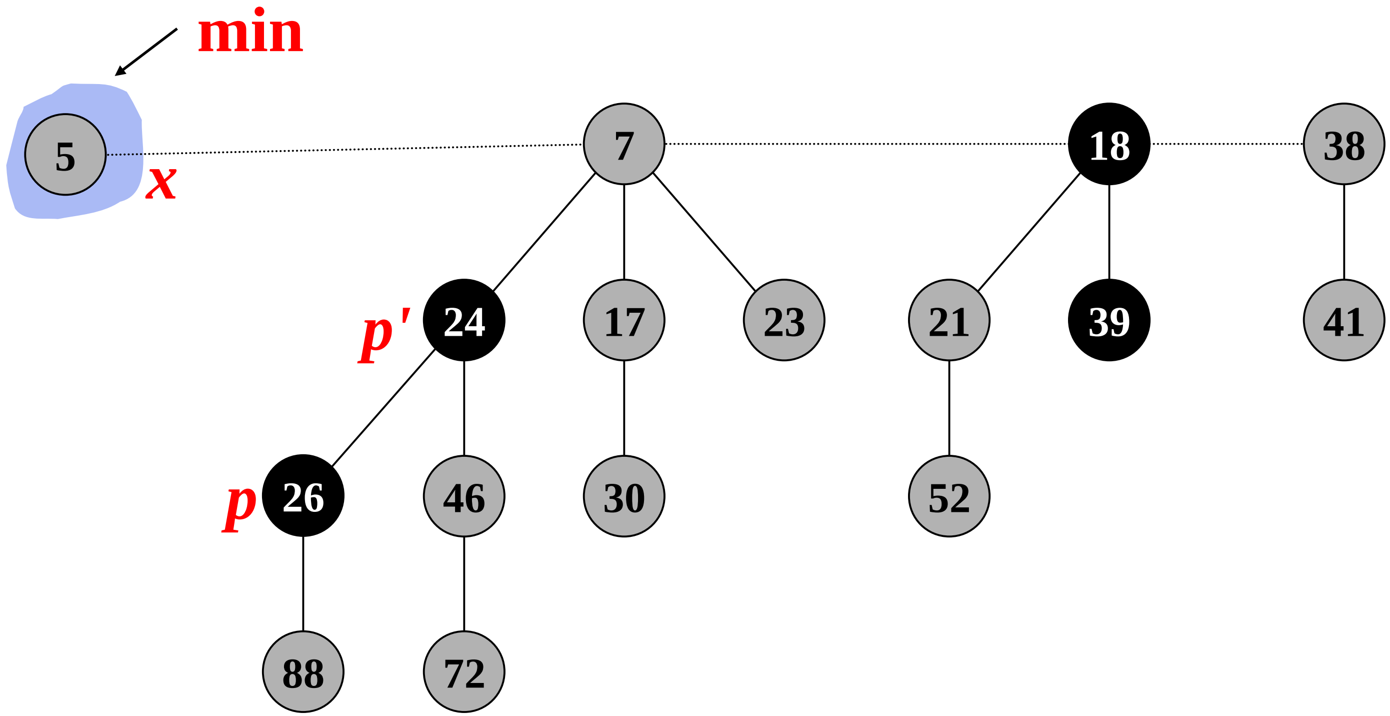
情形2b：减小键值后堆性质不成立

- 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）

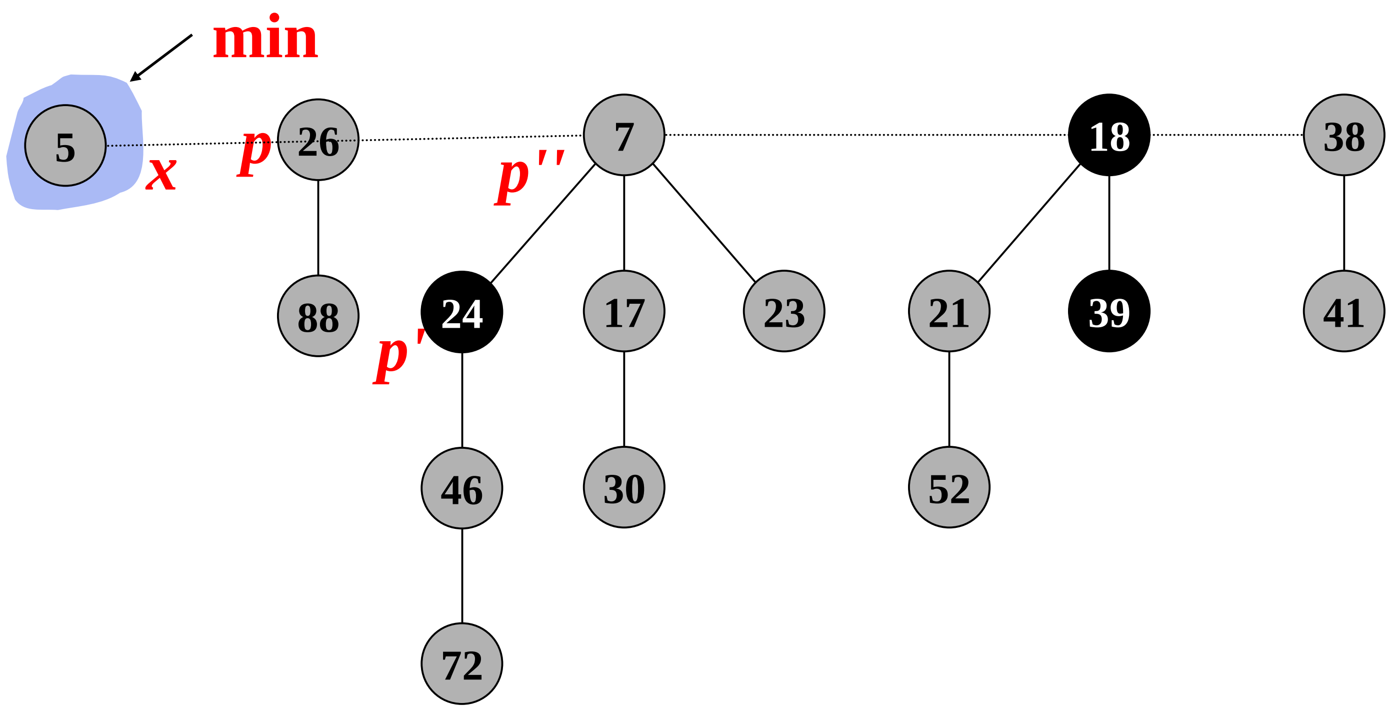


键值减小操作Decrease_key(10)

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）

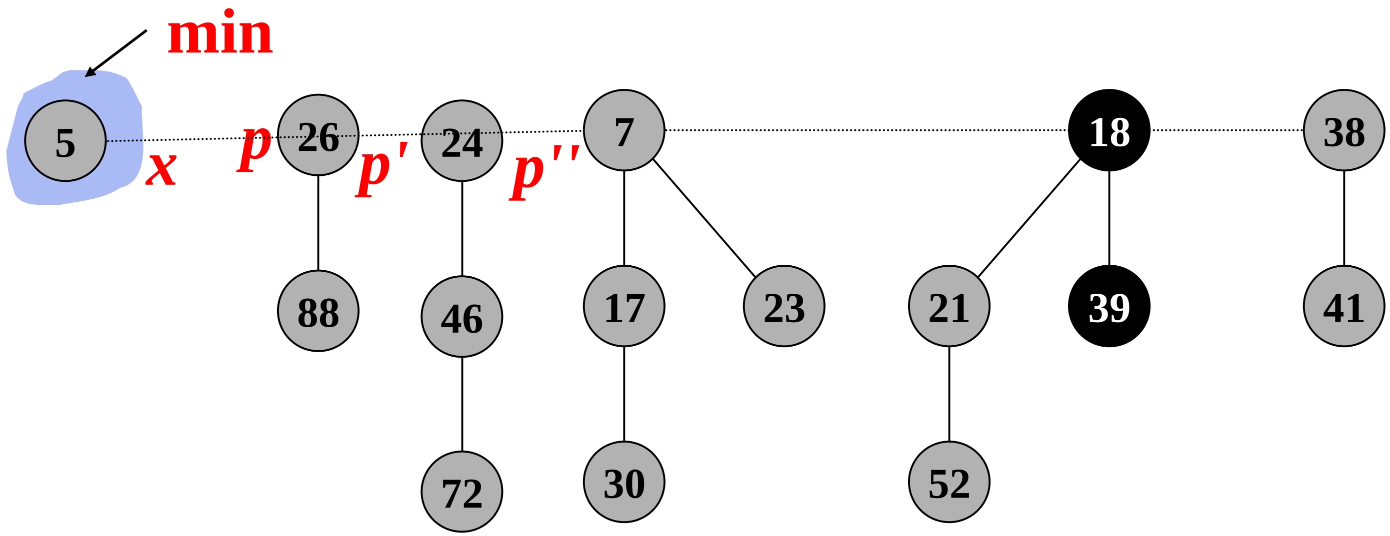


键值减小操作Decrease_key(11)

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）

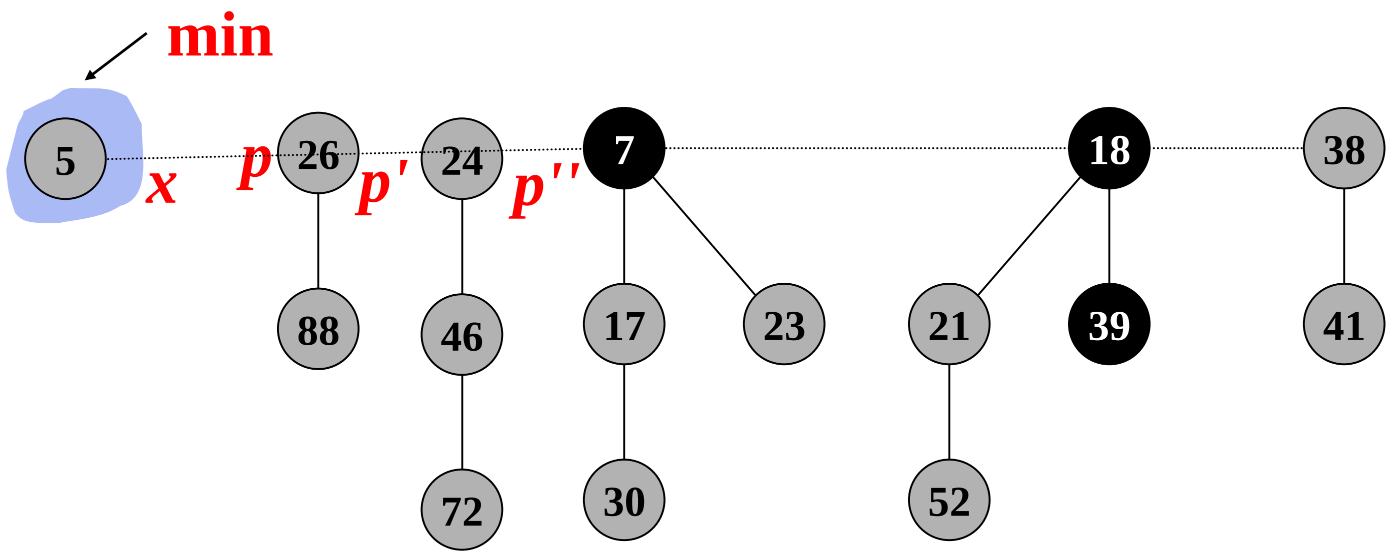


键值减小操作Decrease_key(12)

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的分析

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

- 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
  - $t(H') \le t(H) + c$
  - $mark(H')=mark(H)-(c-1)+1= mark(H) - c +2$
  - $\Delta\Phi(H) =c+2(-c+2)=4-c$

HIT
MDC

目前得到的分析结果

在势能函数$\Phi(H)=t(H)+2\cdot mark(H)$ 下

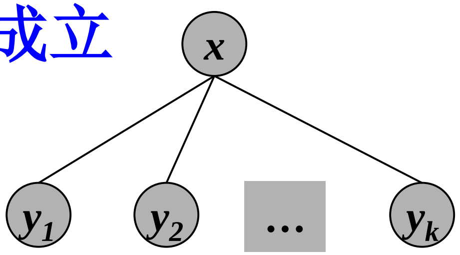
- 插入操作的平摊代价O(1)
- 键值减小操作的平摊代价O(1)
- delete_min的代价是O($rank(H)$)
- 删除操作的平摊代价$O(rank(H))$
    - 先将节点x的键值减小为-∞,则x将出现在堆顶，平摊代价O(1)
    - 调用delete_min删除堆顶元素,平摊代价O($rank(H)$)

欲得定理结论，需证明$rank(H)=O(\log n)$

这意味着，斐波那契堆管理的元素个数是$rank(H)$的指数

$rank(H)$的上界分析

引理1. 设x是斐波那契堆中的任意结点，记$k=rank(x)$。设 $y_1,y_2,\ldots,y_k$ 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$rank(y_1)\geq 0$,且$rank(y_i)\geq i-2$对$i=2,3,\ldots,k$成立



证明. 当y_i成为x的孩子时，

x已有孩子$y_1,y_2,\ldots,y_{i-1}$,故$rank(x)=i-1$

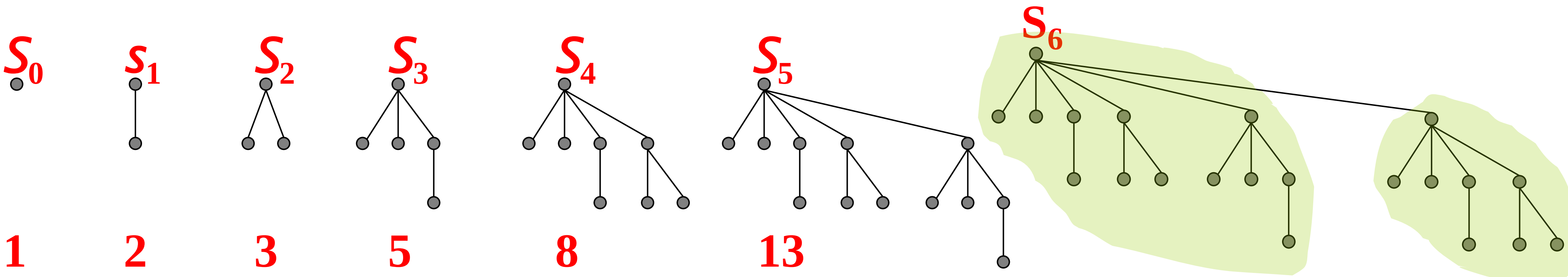
y_i要成为x的孩子，需满足$rank(y_i)=rank(x)=i-1$

y_i成为x的孩子之后，至多失去一个孩子。 *Why？*

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*=*rank*(*x*)，则*x*所在的最小堆中至少有 $\left(\frac{1+\sqrt{5}}{2}\right)^{k-2}$ 个结点。

证明.令s_k表示rank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

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

根 1 2 3 *k*

$\geq 1+1+F(0)+F(1)+\ldots+F(k-2)$ （归纳假设）

$=1+F(k)$ （斐波那契数列性质）

$\geq \left(\frac{1+\sqrt{5}}{2}\right)^{k-2}$ （斐波那契数列性质）

$rank(H)$的上界分析

引理3. 设H是含有n个结点的斐波那契堆，则 $rank(H)=O(\log n)$

证明.

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

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$O(1)$
  - $t(H_1 \cup H_2)=t(H_1)+t(H_2)$
  - $mark(H_1 \cup H_2)=mark(H_1)+mark(H_2)$
  - $\Phi(H)=t(H)+2 \cdot mark(H)$
  - 势能改变量为0

删除操作作用于堆的任意结点***x***

- 将***x***的键值减小为-∞
- 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$O(\mathrm{rank}(H))$
  - $rark(H)=O(\log n)$
  - 删除堆顶元素的平摊代价为$O(1)$

6.7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
- 并查集性能的平摊分析

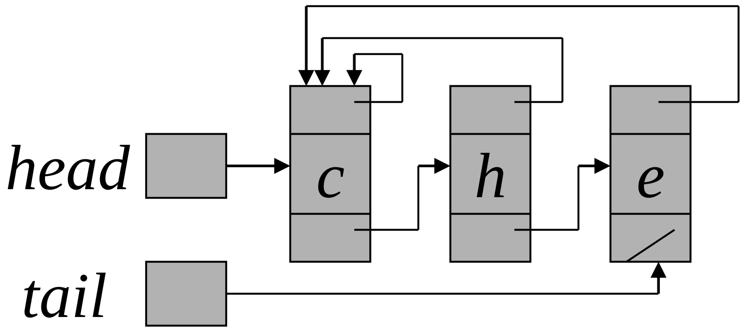
- 目的：管理*n*个不相交的集合$C=\{S_1,\ldots,S_n\}$
  - 每个集合S_i维护一个代表元素x_i
- 支持的操作
  - **MAKE-SET(*x*): 创建仅含元素*x*的集合.**
  - **UNION(*x*,*y*)　：合并代表元素分别*x*和*y*的集合**
  - **FIND-SET(*x*)：返回*x*所在集合的代表元素**
- 目标：使得如下操作序列的代价尽可能低
  - ***n*个MAKE-SET 操作 (开始阶段执行).**
  - ***m*个UNION, FIND-SET操作(后续)**
  - ***m*≥*n*,UNION操作至多执行 *n*-1次**
- 典型应用 （管理图的连通分支）
  - 找出图的连通分支
  - **Krusal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,<u>*Journal of Discrete Algorithms,vol23, pp 2-20,*</u> *2013*

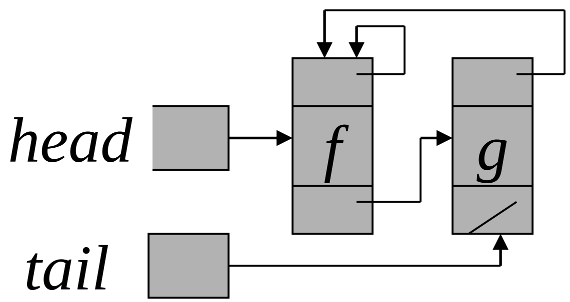
2. Disjoint set union with randomized linking, <u>*Symp. on Discrete Algorithms,*</u> *pp1005-1017, 2014*

并查集的直接实现为链表

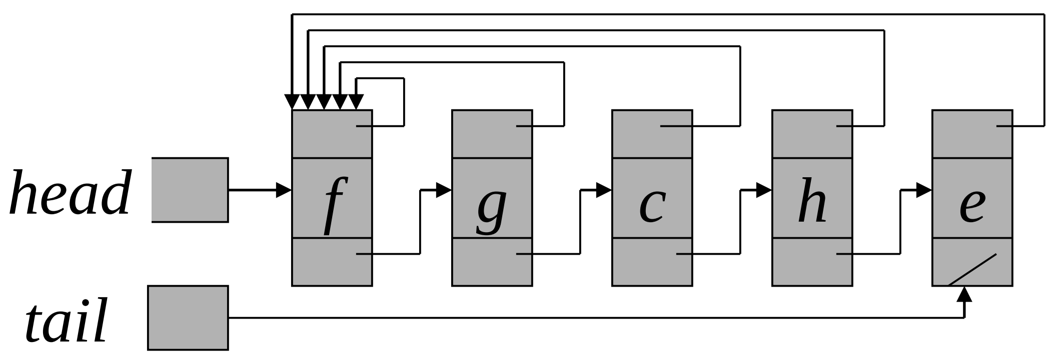
集合 $\{c,h,e\}$



集合$\{f, g\}$



并集$\{c,h,e,f,g\}$



每个集合维护为一个链表

- **head**: 链表头指针
- **tail** :链表尾指针
- **rep** :指向代表元素
- **Make-Set**: **O(1)**
- **Find(x)** : **O(1)**
- **Union(x,y)**: **O($|s_x|$)**

并查集链表实现的性能分析

考虑并查集上 如下特定的操作序列的代价

- 开始阶段执行n个MAKE-SET 操作的总代价$O(n)$
- 后跟n-1个 UNION操作的总代价O(n^2)

Union(x_1,x_2)	代价O(1)
Union(x_2,x_3)	代价O(2)
Union(x_3,x_4)	代价O(3)
…	
Union(x_{n-1},x_n)	代价O(n-1)

- 总共执行执行2n-1次操作的总代价为O(n^2)
- 从平摊效果看，每个操作的开销为$O(n)$

说明链表实现方式是很“蹩脚”，如何提高效率？

并查集链表实现的一种简单改进

考虑并查集链表实现的如下改进，效果会怎么样？

•每个链表表头记录集合（或）链表中元素的个数

•Union操作时将较短链表链接到较长链表

结果

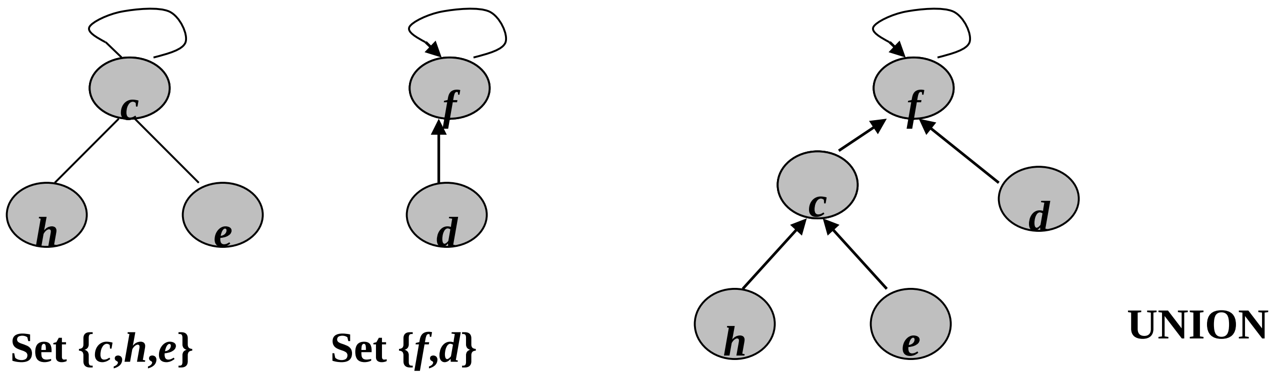
在改进后的并查集上执行由Make_set, Find和Union操作构成的长度为$m+n$的操作序列（其中Make_Set操作有n个），则该操作序列的时间复杂度为$O(m+n\log n)$

为什么？

- 考虑每个元素的rep指针被修改的次数 (总共n个元素)
- 每个元素至多参与$\log n$次并，因为并操作使链表长度至少倍增
- 所有Union操作一起至多$n\log n$次修改rep指针

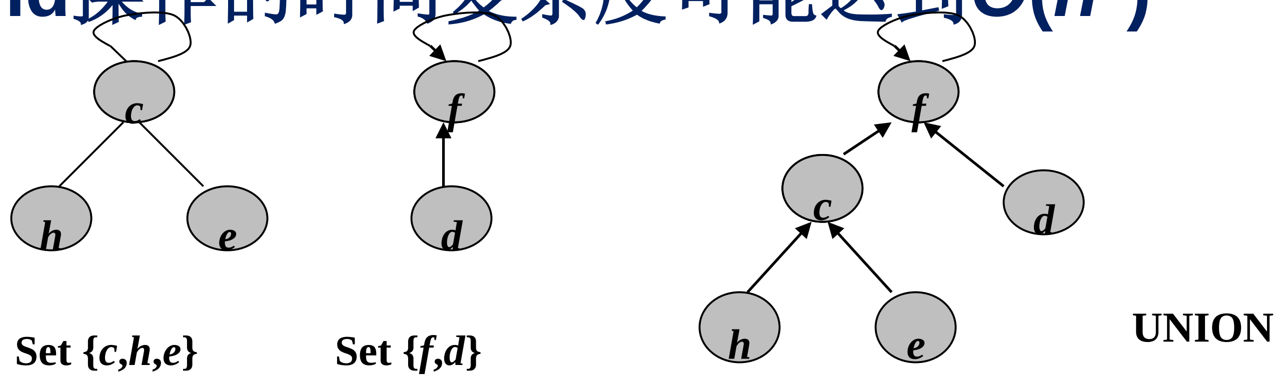
并查集三种操作

- 每个集合表示为一棵有根树
- 树根是代表元素
- 每个结点的指针指向其父结点，根结点指向自身



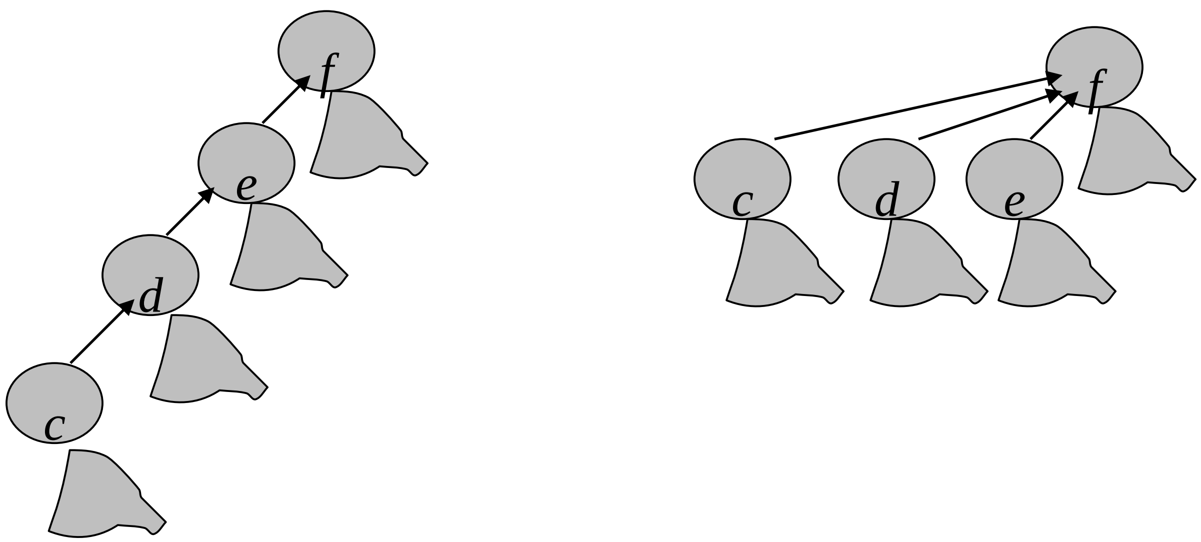
并查集可以实现为森林

- MAKE-SET(x): 创建仅含元素x的一棵树 $O(1)$
- UNION(x,y) : 将x作为y的孩子 $O(1)$
- FIND(x) : 从结点x沿父指针访问直到树根 $O(T_x)$
- n次合并操作可能得到深度为n的树（简单路径）
- 在此极端情况下，Find(x)的最坏时间复杂性为$O(n)$
- n次Find操作的时间复杂度可能达到$O(n^2)$



执行Find(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操作的时间开销
- 树中的路径长度大幅度降低，为后续Find操作节省了时间



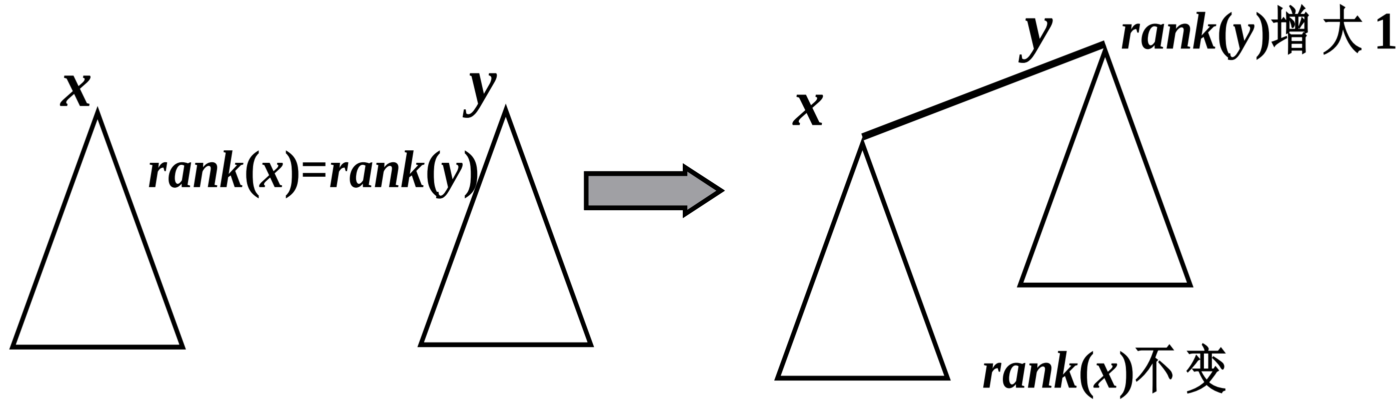
改进策略之按秩合并

根据以下规则，维护每个结点的秩

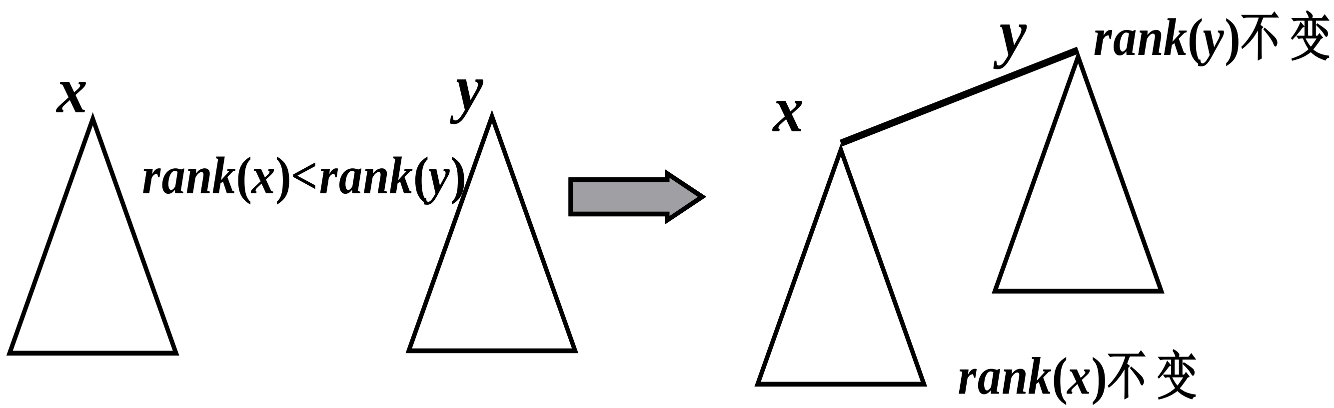
- MakeSet(x)操作执行时定义结点x的秩为0
- Find操作不改变任意顶点的秩
- Union(x,y) 分两种情况修改结点的秩：
  - 情形a：rank(x)=rank(y)。此时，令x指向y且y是并集的代表元素，rank(y)增加1，rank(x)不变（其他结点的秩也保持不变）
  - 情形b：rank(x)<rank(y)。此时，令x指向y且y是并集的代表元素，rank(y)和rank(x)保持不变（其他结点的秩也保持不变）

HIT
MDC

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. rank[x]$\leftarrow$0
2. p[x]$\leftarrow x$

FIND(x)

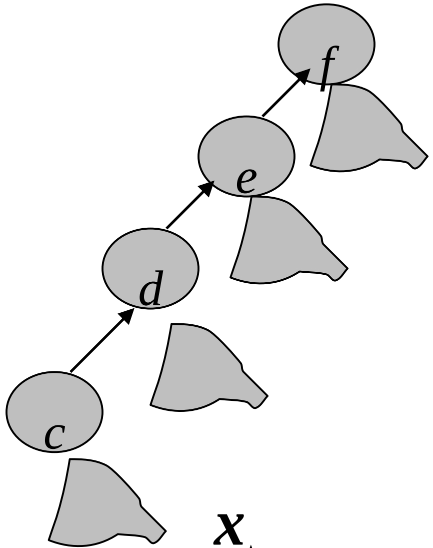
1. Q$\leftarrow\varnothing$
2. While $x\neq$ p[x] Do
3. 将x插入Q;
4. $x\leftarrow p[x]$;
5. For $\forall y\in Q$ do
6. $p[y]\leftarrow x$;
7. 输出x

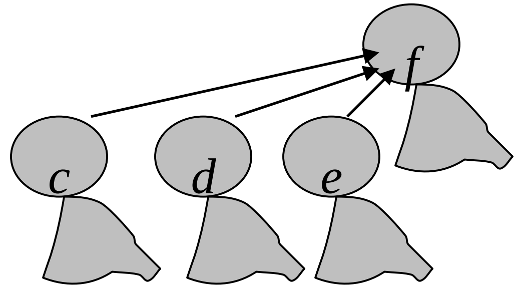
LINK(x,y)

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

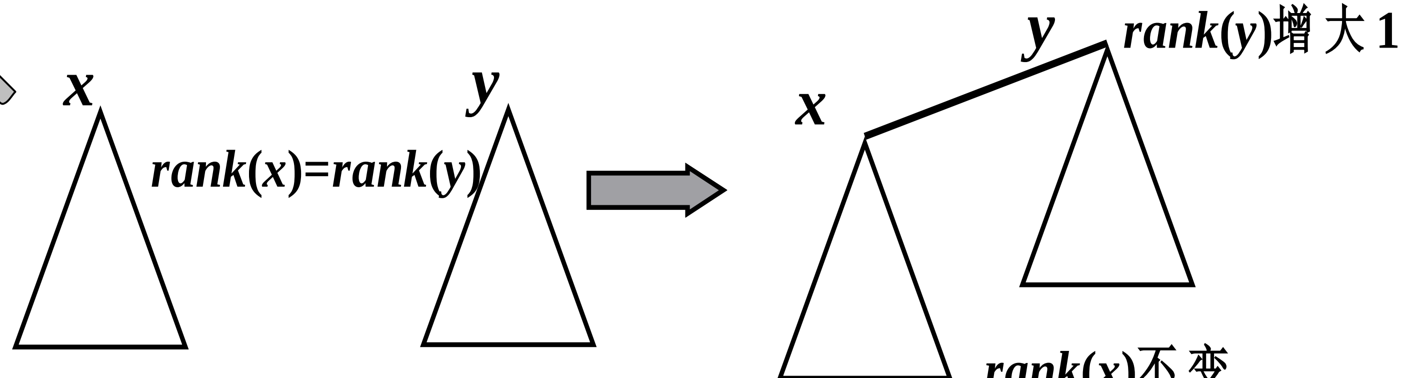
HIT
MDC

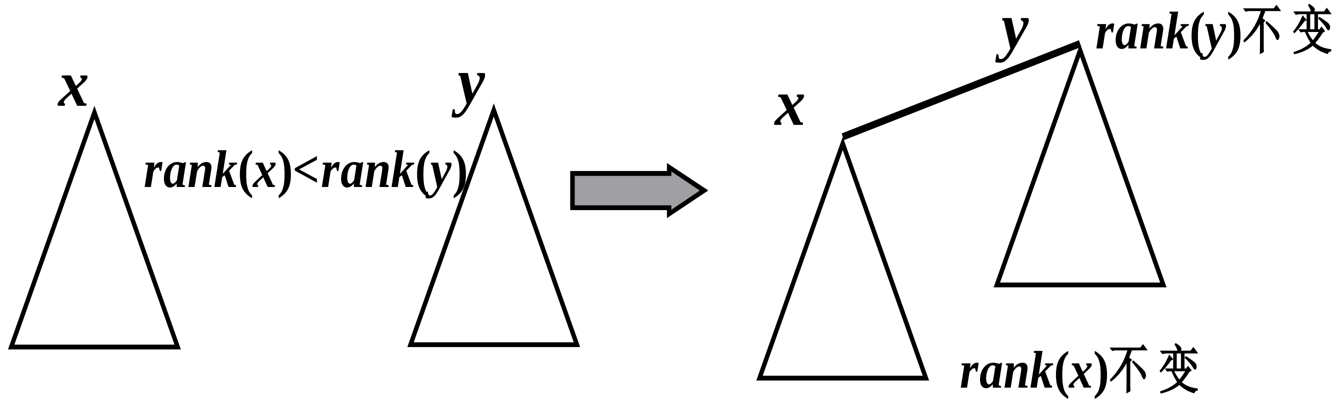
Find





Union





并查集中每个元素的秩有什么基本性质呢？

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，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, $rank(p(x))$是在操作过程中单调缓慢递增

(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

rank(x)缓慢增长

如果将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）
- $\alpha(n)\le 4$，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

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

  $= A_0(A_0(\ldots(A_0(j))\ldots)) +1$ (j层递归)

  $= A_0(A_0(\ldots(A_0(j))\ldots)) +1+1$ (j-1层递归)

  $= \ldots$

  $= A_0(j)+j$

  $=2j+1$

阿克曼函数是定义在$k \geq 0, j \geq 1$上的递归函数

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

利用数学归纳法，不难验证欲得上述结果，需要

- $A_2(j)=A_1(A_1(\ldots(A_1(j))\ldots))$ （j+1层递归）

$= 2A_1(A_1(\ldots(A_1(j))\ldots)) +1$ (j层递归)

$= 2[2A_1(A_1(\ldots(A_1(j))\ldots)) +1]+1$ (j-1层递归)

$= 2^2A_1(A_1(\ldots(A_1(j))\ldots))+2+1$ (j-1层递归,整理上式)

$= \ldots$

$= 2^jA_1(j)+2^{j-1}+ 2^{j-2}+\ldots+2+1$ (1层递归)

$=2^j(2j+1) +2^{j-1}+ 2^{j-2}+\ldots+2+1$

$= 2^{j+1}j+2^{j+1}-1$ （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

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

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

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

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

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

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

$A_k^{(j)}(x)$的性质

k-层， 层越大，增长越快

j-迭代次数，迭代次数越大，增长越快

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	...

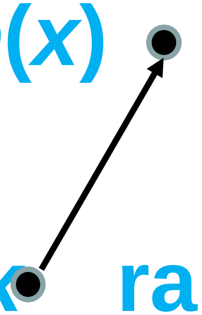
将$\alpha(.)$作用到rank(x)上，能否得出rank(x)的其他性质？

对并查集中的每个结点x,定义

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

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

$p(x)$ rank($p(x)$)



rank(x) 与rank($p(x)$)的距离用$\alpha(.)$来刻画

x rank(x)

- 直观上

路径上rank值单调递增有界

– 一条路径上有***Level***值或**Iter**值较大的节点，路径就很短,继而**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

对并查集中的每个结点x,定义

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

$$Iter(x) = \max\{i \mid A^{(i)}_{Level(x)}(rank(x)) \le rank(p(x))\}$$

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

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

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

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

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

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

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

$$\phi_q = \Sigma_x \phi_q(x)$$

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

并查集性能的平摊分析(5)

势能 $\phi_q = \Sigma_x \phi_q(x)$

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

Make_Set操作的平摊代价为*O*(1)

Make_Set(*y*):

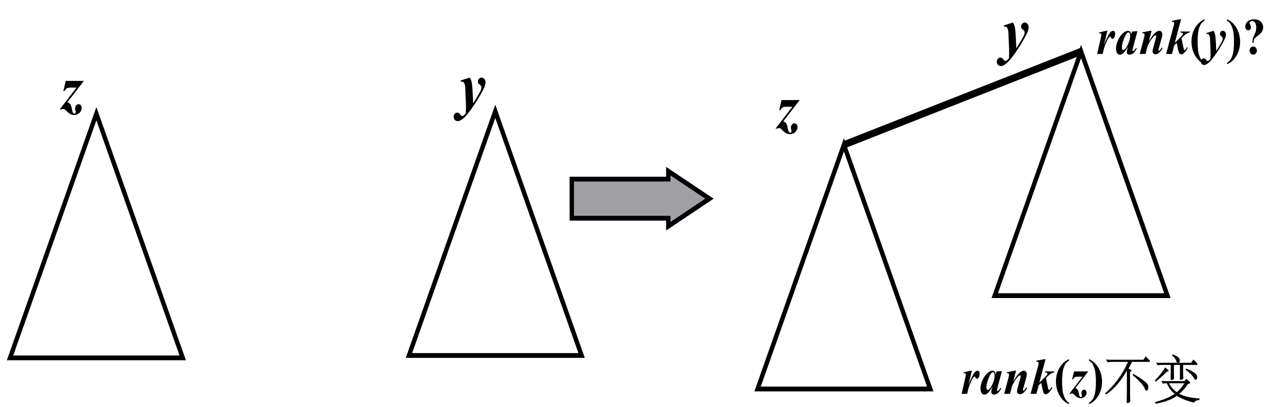
- 实际代价为***O*(1)**
- 势能的增量为**0**
  - 新增一棵以*y*为树根的树，*y*的势能为**0**
  - 不改变其他树的结构和**rank**，其他结点的势能不变

势能 $\phi_q = \Sigma_x\, \phi_q(x)$

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

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rank(*y*)



哪些节点的势能可能会发生改变？可能会怎么变？

势能 $\phi_q = \Sigma_x\, \phi_q(x)$

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

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$\phi_{q+1}(w)\le\phi_q(w)$（参照前面的性质）
    - z的势能不会增加

      操作前，z是树根，故$\phi_q(z)= \alpha(n)\mathrm{rank}(z)$

      操作后，rank(z)不变，且$0\le\phi_{q+1}(z)\le\alpha(n)\mathrm{rank}(z)$
    - y的势能至多增大$\alpha(n)$

      操作前，y是树根，故$\phi_q(y)= \alpha(n)\mathrm{rank}(y)$

      操作时，rank(y)增大1或保持不变

      操作后，y仍是树根，$\phi_{q+1}(y)\le\phi_q(y)+\alpha(n)$

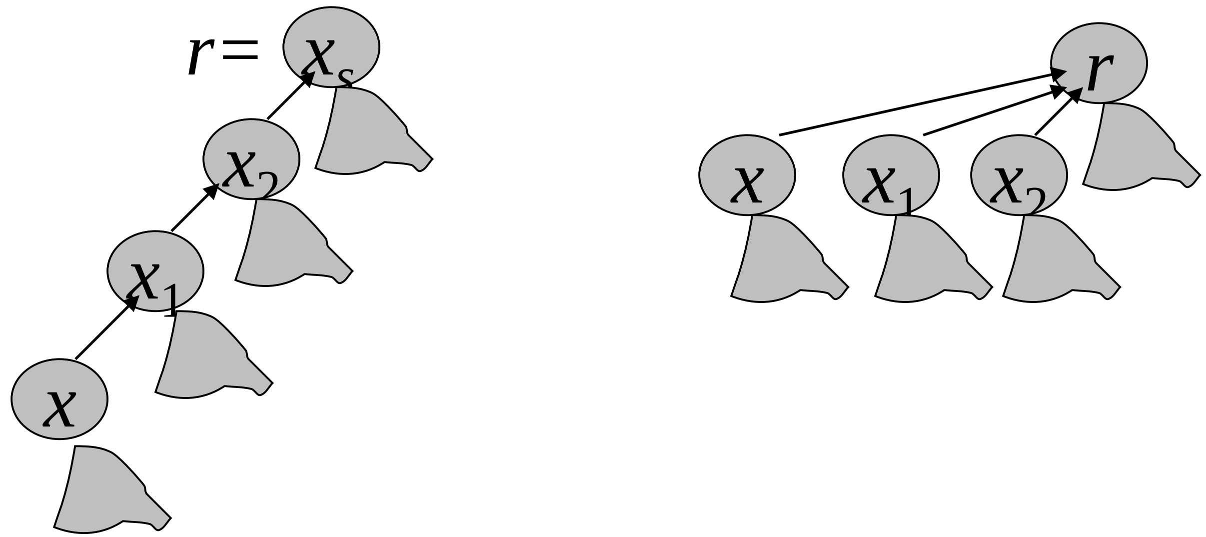
势能$\phi_q = \Sigma_x \phi_q(x)$

并查集性能的平摊分析(7)

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

Find(x)操作的平摊代价为$\Theta(\alpha(n))$

- 实际代价为$\Theta(s)$



- $x=x_0,x_1,\ldots,x_{s-1}$的势能不会增加

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

- 树根r的势能不会发生变化

  rank(r)未发生变化

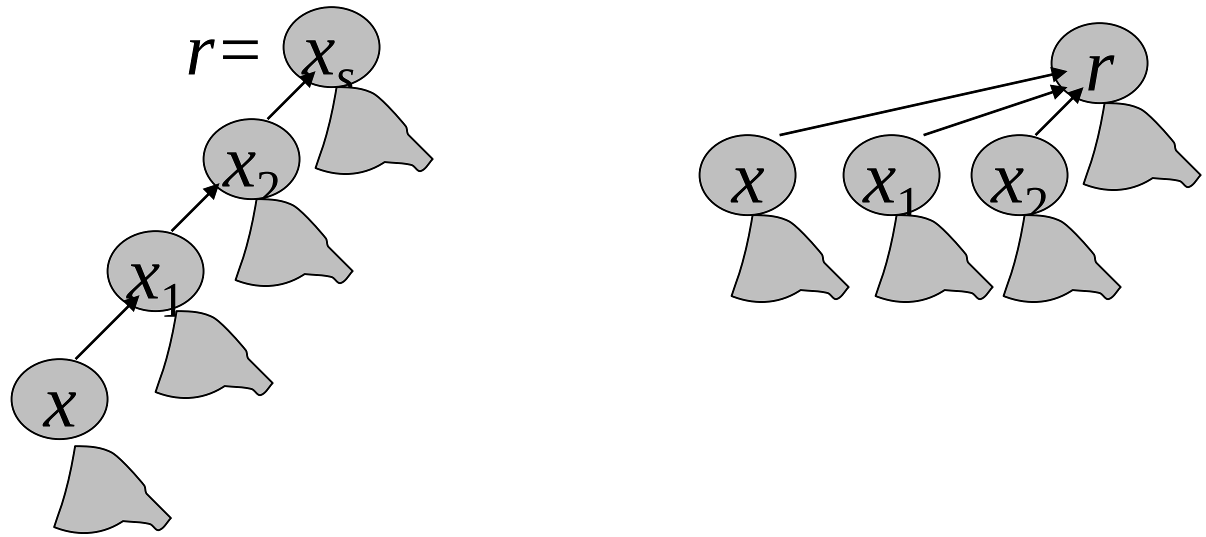
势能$\phi_q = \Sigma_x \phi_q(x)$

并查集性能的平摊分析(8)

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

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$x,x_1,\ldots,x_s$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