算法设计与分析

第七章 平摊分析

丁小欧
dingxiaoou@hit.edu.cn

本章内容

7.1 平摊分析基本原理

7.2 聚集方法

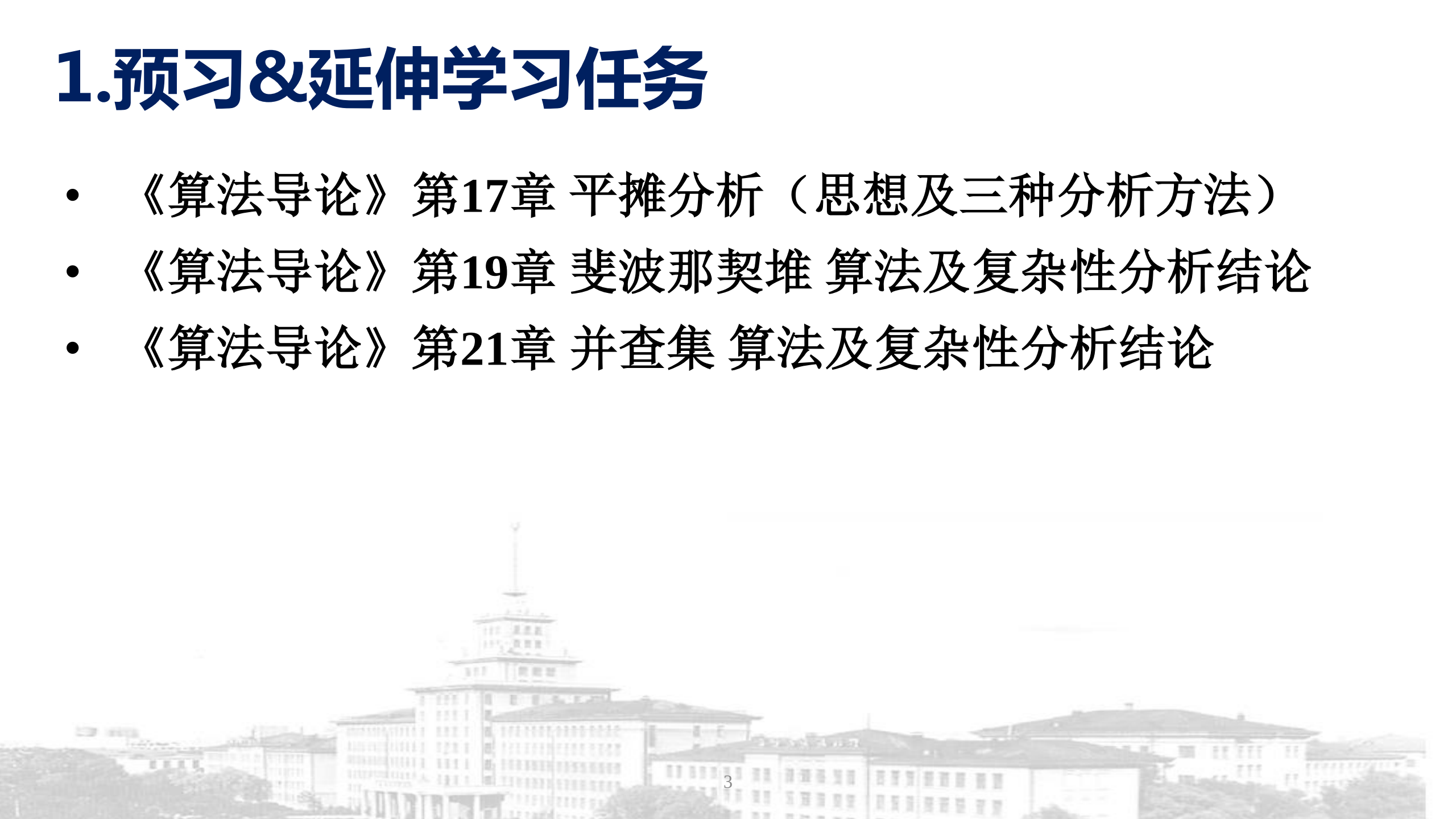
7.3 会计方法

7.4 势能方法

7.5 平摊分析案例

1.预习&延伸学习任务

- 《算法导论》第17章 平摊分析（思想及三种分析方法）
- 《算法导论》第19章 斐波那契堆 算法及复杂性分析结论
- 《算法导论》第21章 并查集 算法及复杂性分析结论



1.平摊分析基本原理

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！

1.平摊分析的基本原理：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

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

各个操作的平均代价？
整个序列的代价是多少？

将总的代价平摊到
每个操作上

平摊代价

不涉及概率
不同于平均分析

1.平摊分析的基本原理：方法分类

- 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”

1.平摊分析的基本原理：分析动机

- **举例: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*θ(n)***

1.平摊分析的基本原理：分析动机

●举例:

•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≠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)***

1.平摊分析的基本原理：分析动机

- **举例:**
- 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$T(n) = O(n^2)$

1.平摊分析的基本原理：分析动机

- **举例:**
- 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*

分析虽然没错，但过于粗糙，不是一个确界

原因：只关注于操作，忽略了数据结构！

本章内容

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 平摊分析案例

2.聚集方法：实例

- 栈操作
- 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$\theta(n)$

2.聚集方法：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

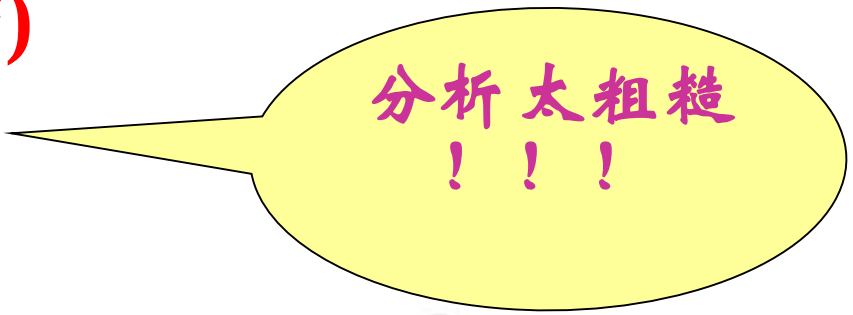
$$T(n)=\sum_{i=1}^{n} t_i$$

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

2.聚集方法：实例

- 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$T(n)/n = O(n)$



2.聚集方法：实例

- 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
  - 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*次
    - 最坏情况下操作序列的代价为$T(n) \le 2n = O(n)$
    - 平摊代价为$T(n)/n = O(1)$

*n-1*个*push*
*1*个*multipop*

2.聚集方法：实例二

- 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

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

2.聚集方法：实例二

- 二进制计数器

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<length[A]  Then  A[i]←1
```

2.聚集方法：二进制计数器

- 初始为零的计数器上 *16* 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

2.聚集方法：二进制计数器

- 初始为零的计数器上*n*个INCREMENT操作分析
  - 将二进制计数器递增16次：初始值为0，最终值为16
  - 将1加在第*i*位上，如果*A[i]=1*,则加1操作将第*i*位翻转为0，并产生一个进位——在下一步循环时将1加到第*i+1*位上

每次INCREMENT操作的代价与翻转的二进制位的数目呈线性关系

2.聚集方法：实例二

- 二进制计数器

计数器加1算法

- 粗略分析：正确但不紧界
  - 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$O(k)$
  - 例中：$k*n=8*16=128$

2.聚集方法：实例二

- 二进制计数器 计数器加1算法
- 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）
  - $T(n)=\sum_{0\le i\le lgn} n/2^i < n\sum_{0\le i\le\infty} 1/2^i = 2n = O(n)$
  - 每个Increment操作的平摊代价为*O*(*1*)

本章内容

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 平摊分析案例

3.会计方法(accounting)：原理

- 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

3.会计方法：原理

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

3.会计方法：原理

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

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

3.会计方法：栈操作序列分析

- 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***

3.会计方法：栈操作序列分析

- 各平摊代价满足
  - $\Sigma_{1\le i\le n}\alpha_i - \Sigma_{1\le i\le n} c_i \ge 0$ 对于任意n个操作序列都成立
    - 因为栈中的元素个数≥0，Pop(MultiPop中的Pop)的个数不大于Push的个数
    - 即任何时刻，栈上的Credit非负
- n个栈操作序列的总平摊代价
  - $O(n)$

3.会计方法：二进制计数器

- 初始为零的计数器上 *16* 个 **INCREMENT** 操作分析
  - 用1元支付置于1的开销
  - 用1元存储在该“1”位上，用于支付其被置为0的开销
  - 置0操作无需再付款

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	2
3	0	0	0	0	0	0	1	1	1	1
4	0	0	0	0	0	1	0	0	0	3
5	0	0	0	0	0	1	0	1	1	1
6	0	0	0	0	0	1	1	0	0	2
7	0	0	0	0	0	1	1	1	1	1
8	0	0	0	0	1	0	0	0	0	4
9	0	0	0	0	1	0	0	1	1	1
10	0	0	0	0	1	0	1	0	0	2
11	0	0	0	0	1	0	1	1	1	1
12	0	0	0	0	1	1	0	0	0	3
13	0	0	0	0	1	1	0	1	1	1
14	0	0	0	0	1	1	1	0	0	2
15	0	0	0	0	1	1	1	1	1	1
16	0	0	0	1	0	0	0	0	0	5

3.会计方法：二进制计数器

- Increment操作的平摊代价
  - 用1元支付置于1的开销
  - 用1元存储在该“1”位上，用于支付其被置为0的开销
  - 置0操作无需再付款
- 对于每个Increment操作
  - 找到左起的第一个0，将其翻转为1——支付平摊代价2
  - 将该0之前的所有1翻转成0——支付平摊代价0
  - 对这个Increment操作而言，支付了平摊代价2

3.会计方法：二进制计数器

- Increment操作的平摊代价
  - 用1元支付置于1的开销
  - 用1元存储在该“1”位上，用于支付其被置为0的开销
  - 置0操作无需再付款
- 平摊代价满足
  - $\Sigma_{1\le i\le n}\alpha_i \ge \Sigma_{1\le i\le n}c_i$ 对任意n个操作序列都成立，
  - 因为分析知 $\Sigma_{1\le i\le n}c_i < 2n$

3.会计方法：二进制计数器

- Increment操作的平摊代价
  - 用1元支付置于1的开销
  - 用1元存储在该“1”位上，用于支付其被置为0的开销
  - 置0操作无需再付款
- 对于长度为n的Increment操作序列
  - 支付平摊代价的总和为$2n$
- n个Increment操作序列的总平摊代价
  - $O(n)$

本章内容

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 平摊分析案例

4.势能法：原理

–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

4.势能法：原理

–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[illegible]据结构的势能来支付实

将“势能”释放出来即可
支付未来操作的代价！

4.势能法：原理

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

势差

4.势能法：原理

- 数据结构势能的定义
  - *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$\phi(D_0)=0$，$\phi(D_i)\geq0$(由具体问题确定)

4.势能法：原理

- 数据结构势能的定义
  - n个操作的总平摊代价

总平摊代价必须是
总实际代价的上界

平摊代价依赖所选择的势函数；
不同势函数可能会产生不同的平摊代价，
但它们都是实际代价的上界

  - 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$\phi(D_0)=0$，$\phi(D_i)\geq0$(由具体问题确定)

4.势能法：栈操作序列分析

- 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
    - 这样的ϕ定义的 n个操作的总平摊代价是实际代价的上界

4.势能法：栈操作序列分析

- 栈的势能定义
  - $\phi(D_m)$定义为栈D_m中对象的个数
  - 栈操作的平摊代价（设栈D_{i-1}中具有s个对象）
    - 如果第i个操作是**PUSH:**
      - 实际代价$c_i = 1$
      - 势差$\phi(D_i)-\phi(D_{i-1})=(s+1)-s=1$
      - 平摊代价：$\alpha_i=c_i+\phi(D_i)-\phi(D_{i-1})=1+(s+1)-s=2$

4.势能法：栈操作序列分析

- 栈的势能定义
  - $\phi(D_m)$定义为栈D_m中对象的个数
  - 栈操作的平摊代价（设栈D_{i-1}中具有s个对象）
    - 如果第i个操作是POP:
      - 实际代价$c_i = 1$
      - 势差$\phi(D_i)-\phi(D_{i-1})=(s-1)-s=-1$
      - 平摊代价：$\alpha_i=c_i+\phi(D_i)-\phi(D_{i-1})=1-1=0$

4.势能法：栈操作序列分析

- 栈的势能定义
  - $\phi(D_m)$定义为栈D_m中对象的个数
  - 栈操作的平摊代价（设栈D_{i-1}中具有s个对象）
    - 如果第i个操作是MULTIPOP(s, k):
    - 设$k'=min(k,s)$为实际代价（弹出了k'个对象）
      - 实际代价$c_i = k'$
      - 势差$\phi(D_i)-\phi(D_{i-1})=- k'$
      - 平摊代价：$\alpha_i=c_i+\phi(D_i)-\phi(D_{i-1})= k' - k' =0$

4.势能法：栈操作序列分析

- 栈的势能定义
  - $\phi(D_m)$定义为栈D_m中对象的个数
  - 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$k'=min(k,s)$为实际代价

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

4.势能法：栈操作序列分析

- 栈的势能定义
  - $\phi(D_m)$定义为栈D_m中对象的个数
  - 栈操作的平摊代价（设栈D_{i-1}中具有s个对象）
    - 每个栈操作的平摊代价都是$O(1)$
  - n个栈操作序列的平摊代价是$O(n)$
    - n个操作的总平摊代价是总的实际代价的一个上界

4.势能法：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，其中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，第$i$个操作之后的$\phi(D_i)$满足$\phi(D_i)\geq 0=\phi(D_0)$
    - 于是，n个操作的总平摊代价是实际代价的上界

4.势能法：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
  - INCREMENT操作的平摊代价
    - 第i个INC操作把 t_i个位置成0, 实际代价至多为$c_i=t_i+1$
    - 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位置为0(复位), 所以$b_{i-1}=k$, $t_i=k$
      - If $b_i>0$, 则$b_i=b_{i-1}-t_i+1$
      - 于是 $b_i \le b_{i-1}-t_i+1$

        势差：$\phi(D_i)-\phi(D_{i-1})=b_i-b_{i-1} \le b_{i-1}-t_i+1-b_{i-1}=1-t_i$
      - 平摊代价$\alpha_i=c_i+\phi(D_i)-\phi(D_{i-1}) \le (t_i+1)+(1-t_i)=2$

4.势能法：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
  - **INCREMENT操作的平摊代价**
    - 每个操作的平摊代价都是$O(1)$
  - n个操作序列的总平摊代价是$O(n)$
    - n个操作的总平摊代价是总的实际代价的一个上界

4.势能法相比于前两种方法的优势

- 势能法更常用
- 可以用势能法去分析不从*0*开始的计数器
  - 课后任务：试给出对初始状态从包含*m*个*1*开始的二进制计数器的平摊分析

本章内容

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 动态表性能的平摊分析

5.1动态表性能平摊分析：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$\alpha(T)$
  - $\alpha(T)$= 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

5.1动态表性能平摊分析：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

5.2动态表性能平摊分析：动态表的扩张

- 插入一个数组元素时,完成的操作(主要)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

5.2动态表性能平摊分析：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*

5.2动态表性能平摊分析：扩张算法

```
TABLE—INSERT(T, x)
1        If  size[T]=0 Then
2             获取一个大小为1的表 table[T];
3             size[T]←1;
4        If  num[T]=size[T] Then
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;
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
```

- *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

5.2动态表性能平摊分析：扩张算法

```
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];                       /*简单插入操作*/
11      num[T]←num[T]+1
```

5.2动态表性能平摊分析：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$num[T]=size[T]$, 发生一次扩张，$C_i=i$;
  - 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

5.2动态表性能平摊分析：初始为空的表上n次插入操作的代价分析

- 聚集分析-精细分析 （聚集法）
  - –第i次操作的代价Ci
    - •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

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

★n个代价为1的操作+由操作代价形成的等比数列

★每一操作的平摊代价为$3n/n=3$

5.2动态表性能平摊分析：

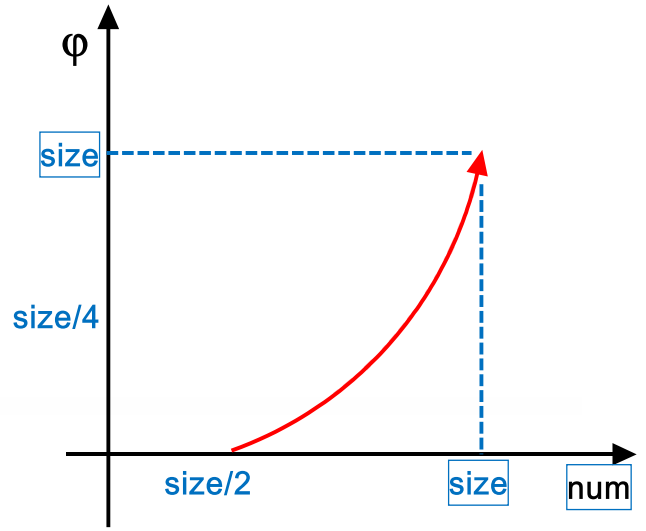
- 每次执行TABLE—INSERT平摊代价为3？
  - 思考：
  - 表插入分两种情况，一种是无扩张的表插入，一种是扩张的表插入，这两个操作的代价不同（可通过例子说明平摊代价必须大于2）
  - 平摊代价为n，无论怎么扩张，都足够支付扩张费用，但分析太粗糙
  - 尽可能最小化平摊代价，当其至少为3时，是足够支付扩张费用的

5.2动态表性能平摊分析：初始为空的表上*n*次插入操作的代价分析

- 会计法
- 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(表中保存的*m*个数据项已满时，每项都存储了1)
- 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3*n*

5.2动态表性能平摊分析：初始为空的表上*n*次插入操作的代价分析

- 势能法分析
- 怎么定义势能才能使得表满发生扩张时势能能支付扩张的代价?



- 预期势能函数应满足：
  - 刚扩充完,$\phi(T)=0$
  - 表满时$\phi(T)=size(T)$
- 定义$\phi(T)=2*num[T]-size[T]$
  - 由于表至少是半满：$num[T]\geq size[T]/2$，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

5.2动态表性能平摊分析：初始为空的表上n次插入操作的代价分析

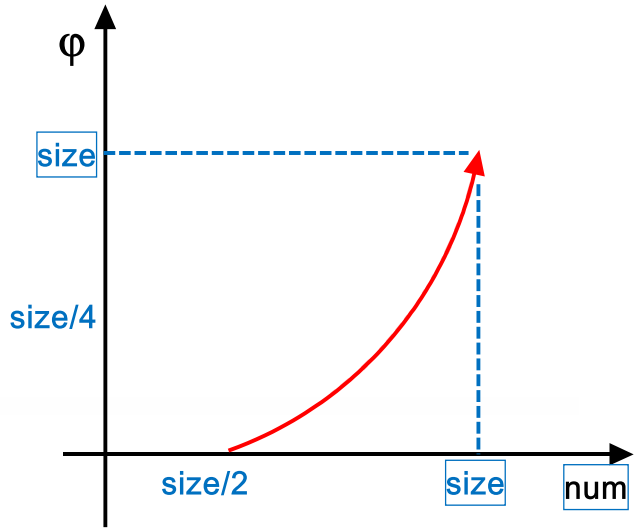
- 势能法分析
- 势能怎么定义才能使得表满发生扩张时势能能支付扩张的代价?

•第i次操作的平摊代价

–如果未发生扩张, α_i=3

–如果发生扩张, α_i =3

Why???



•初始为空的表上n次插入操作的代价的上界为$3n$

5.2动态表性能平摊分析：初始为空的表上n次插入操作的代价分析

- 势能法分析
- 第i次操作的平摊代价
  - 如果没有引发扩张
    - 有$size_i(T)=size_{i-1}(T)$
    - $num_i(T)=num_{i-1}(T)+1$
    - 平摊代价：$\alpha_i=c_i+\phi_i(T)-\phi_{i-1}(T)$

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

      $=1+2=3$
  - 如果引发了扩张

定义势函数

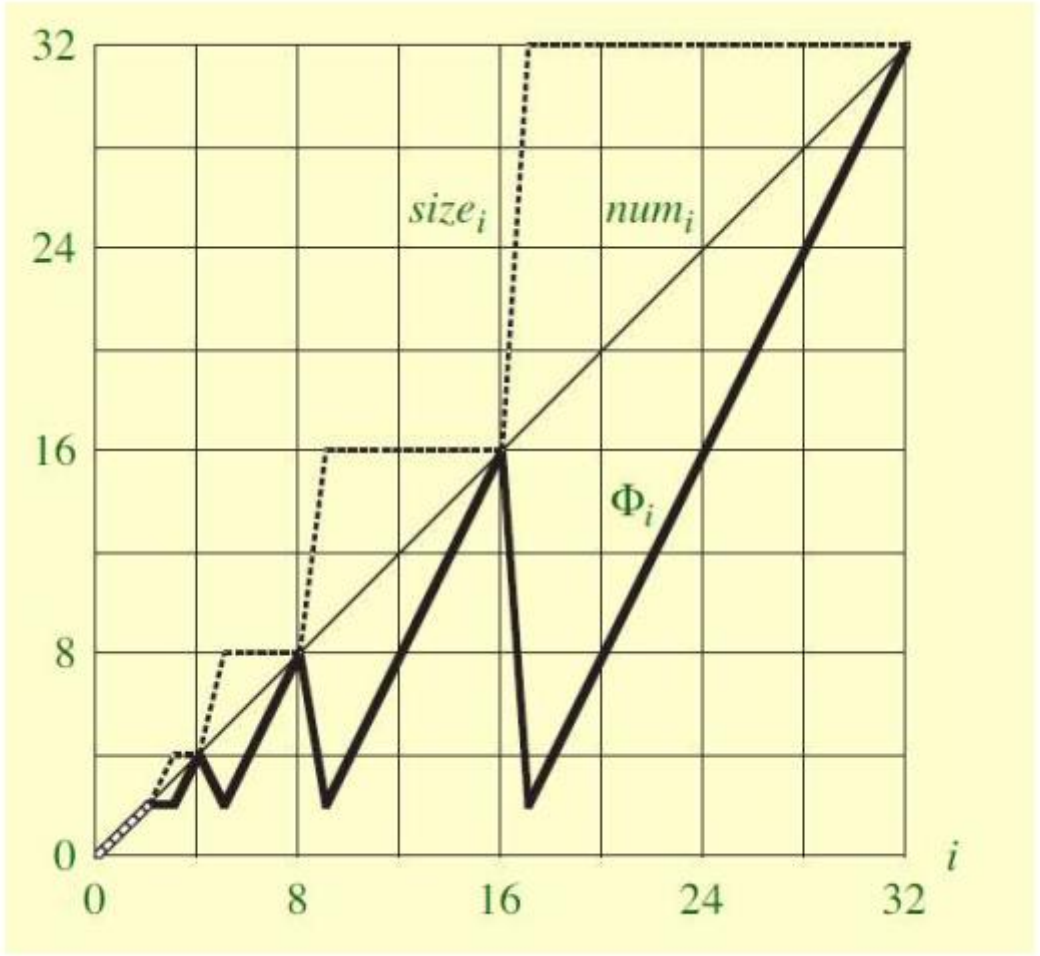
$\phi(T)=2*num[T]-size[T]$

5.2动态表性能平摊分析：初始为空的表上n次插入操作的代价分析

- 势能法分析
- 第i次操作的平摊代价
  - 如果没有引发扩张
    - 平摊代价： $\alpha_i=c_i+\phi_i(T)-\phi_{i-1}(T)=3$
  - 如果引发了扩张
    - 有$size_i(T)=2\times size_{i-1}(T)$
    - $num_{i-1}(T)=size_{i-1}(T)=num_i(T)-1$
    - 平摊代价$\alpha_i=c_i+\phi_i(T)-\phi_{i-1}(T)=1+2=3$

5.2动态表性能平摊分析：初始为空的表上*n*次插入操作的代价分析

- 势能法分析
- 注意：在第i个操作后测量每个参数
- 规律：在每次扩张前，势变为表中数据项的数量，进而有能力支付将所有数据项加入新表的代价
- 扩张后，势落为0，但会马上变为2——有引起扩展的数据项加入表中



5.3动态表性能平摊分析：动态表的扩展与收缩

- 现考虑引入表收缩操作，进一步提高其灵活性
- **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
  - 装载因子有正的常数下界
- 表的操作序列的（平摊）复杂度是线性的

5.3动态表性能平摊分析：动态表的扩展与收缩

- **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
- 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$O(n)$

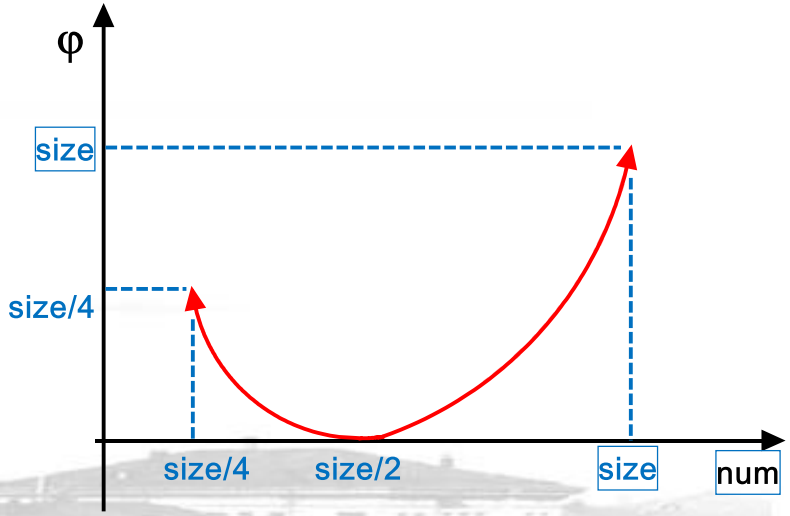
  --每个操作的平摊代价太高！

5.3动态表性能平摊分析：动态表的扩展与收缩

- 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

5.3动态表性能平摊分析：动态表的扩展与收缩

- 动态表上*n*次(插入、删除)操作的代价分析（用势能法）
  - 设计势能函数：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



5.3动态表性能平摊分析

- 动态表上n次(插入、删除)操作的代价分析
  - 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

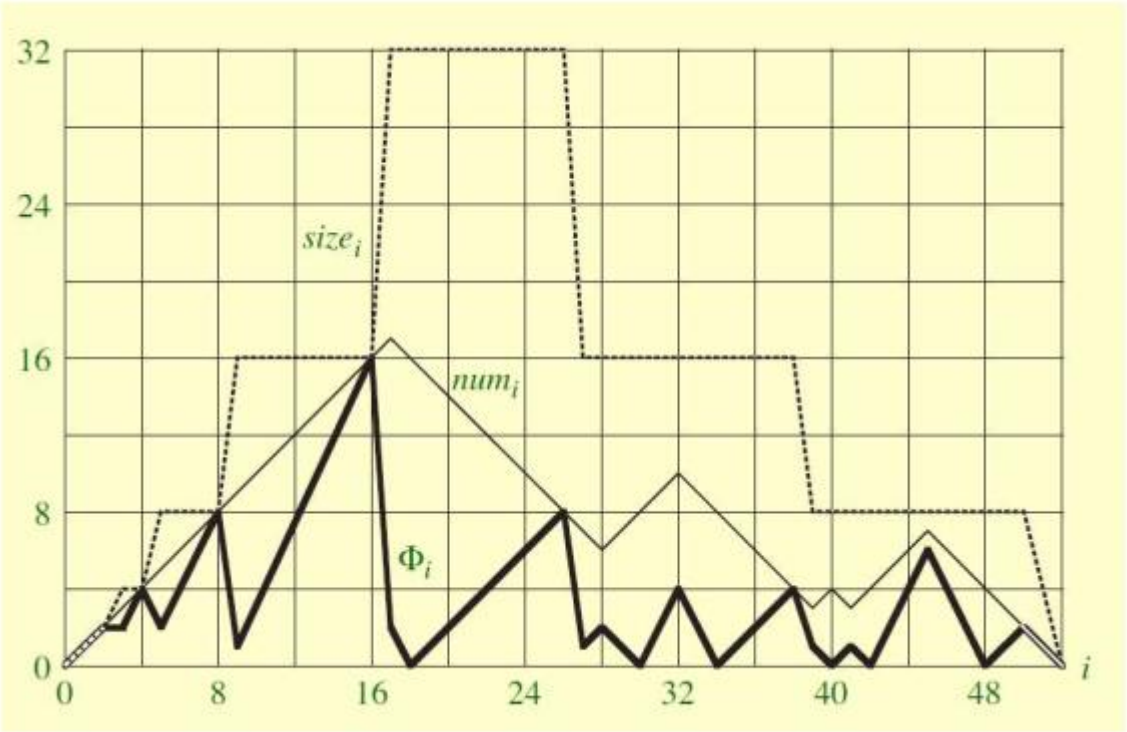
$$\Phi(T)=\begin{cases}2\cdot num[T]-size[T] & \alpha(T)\geq 1/2\\ size[T]/2-num[T] & \alpha(T)<1/2\end{cases}$$

5.3动态表性能平摊分析

- 动态表上n次(插入、删除)操作的代价分析
- 平摊代价计算
- 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 $\alpha_i \leq 3$
- 动态表上的n个操作的实际时间为$O(n)$

5.2动态表性能平摊分析：

- 势能法分析
- 注意：在第*i*个操作后测量每个参数
- 规律：在每次扩张前，势变为表中数据项的数量，进而有能力支付扩张过程中数据项的移动代价
- 在每次收缩前，势的积累也达到了数据项的数量



本章内容

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

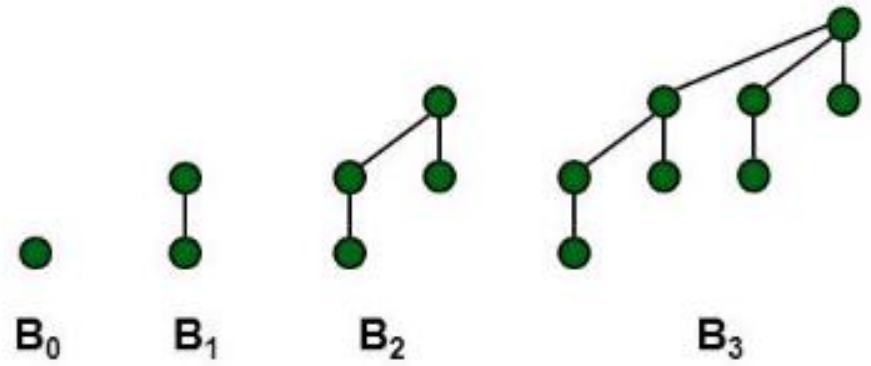
7.5 动态表性能的平摊分析

7.6 *斐波那契堆性能平摊分析

7.7 并查集性能平摊分析

斐波那契堆平摊分析

- 二叉堆：就是普通的最小堆，用二叉树的形式表示
- 二项堆：是指满足以下性质的二项树的集合
  - 每棵二项树都满足最小堆性质，即结点关键字大于等于其父结点的关键字
  - 不能有两棵或以上的二项树有相同度数（包括度数为0）
    - 度为k的二项树的唯一根结点下有k个孩子，其孩子分别是度数为*k-1,k-2,,,,2,1,0*的二项树的根



斐波那契堆平摊分析

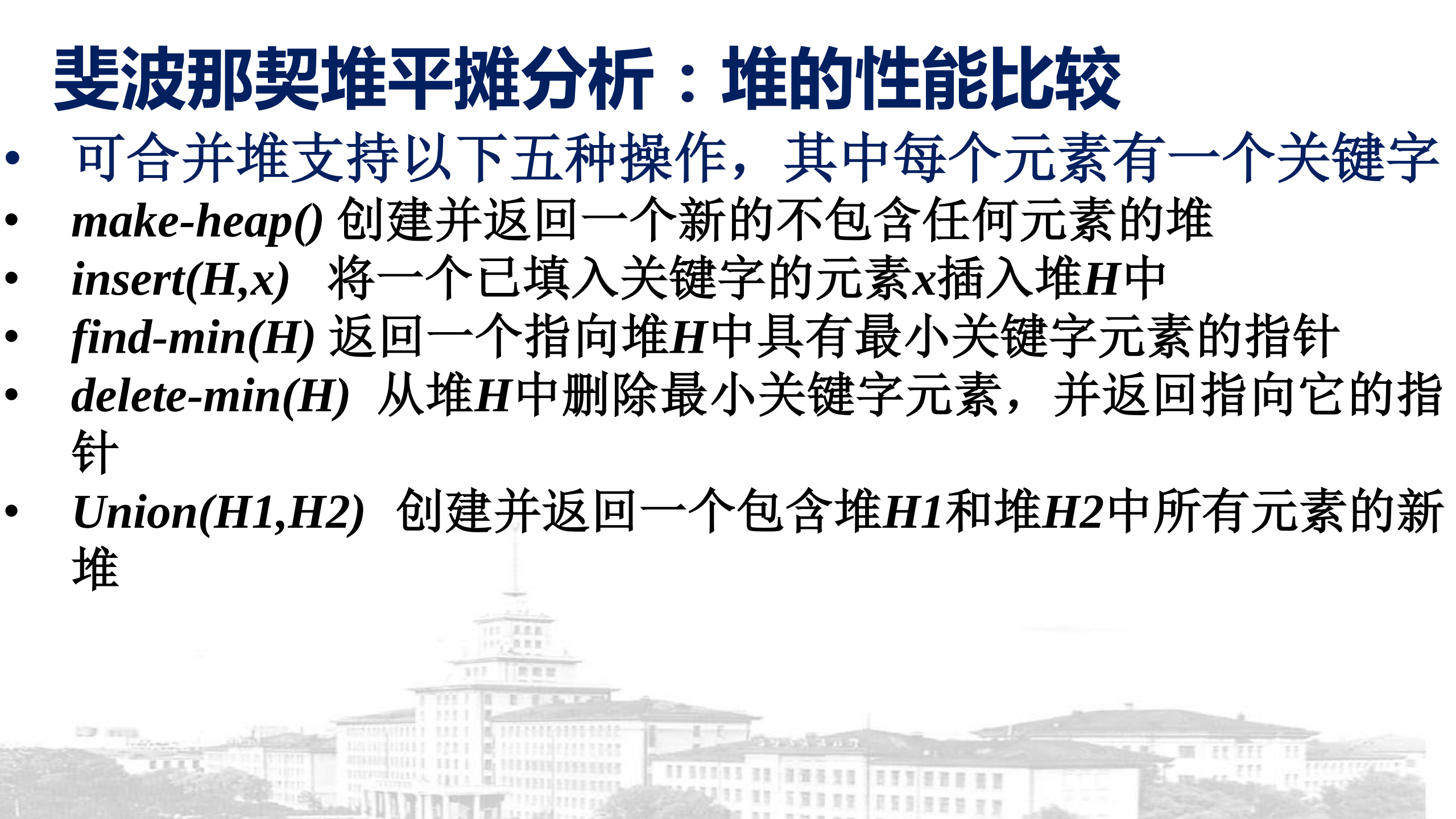
- 斐波那契堆
  - 由一组最小堆有根树构成，每个节点的度数为其子节点的数目。树的度数为其根节点的度数。
  - 斐波那契堆中的树都是有根的但是无序。每个节点x包含指向父节点的指针p[x]和指向任意一个子结点的child[x]
  - 斐波那契堆中所有树的根结点用一个双向链表链接
  - 一个指针指向斐波那契堆中的最小元素

斐波那契堆平摊分析

- 斐波那契堆
  - 由一组最小堆有根树构成，每个节点的度数为其子节点的数目。树的度数为其根节点的度数。
  - 斐波那契堆中的树都是有根的但是无序。每个节点x包含指向父节点的指针p[x]和指向任意一个子结点的child[x]
  - 斐波那契堆中所有树的根结点用一个双向链表链接
  - 一个指针指向斐波那契堆中的最小元素
  - 比二项堆有更好的平摊分析性能！

斐波那契堆平摊分析：堆的性能比较

- 可合并堆支持以下五种操作，其中每个元素有一个关键字
- ***make-heap()*** 创建并返回一个新的不包含任何元素的堆
- ***insert(H,x)*** 将一个已填入关键字的元素***x***插入堆***H***中
- ***find-min(H)*** 返回一个指向堆***H***中具有最小关键字元素的指针
- ***delete-min(H)*** 从堆***H***中删除最小关键字元素，并返回指向它的指针
- ***Union(H1,H2)*** 创建并返回一个包含堆***H1***和堆***H2***中所有元素的新堆



斐波那契堆平摊分析：堆的性能比较

- 斐波那契堆还支持两种操作
- ***make-heap()*** 创建并返回一个新的不包含任何元素的堆
- ***insert(H,x)*** 将一个已填入关键字的元素*x*插入堆*H*中
- ***find-min(H)*** 返回一个指向堆*H*中具有最小关键字元素的指针
- ***delete-min(H)*** 从堆H中删除最小关键字元素，并返回指向它的指针
- ***Union(H1,H2)*** 创建并返回一个包含堆*H1*和堆*H2*中所有元素的新堆
- ***decrease-key(H,x,k)*** 将堆*H*中元素*x*的关键字赋予新值*k*
- （默认合并最小堆，则k不大于当前的关键字）
- ***Delete(H,x)*** 从堆*H*中删除元素*x*

斐波那契堆平摊分析：堆的性能比较

- 斐波那契堆对操作insert,union, decrease-key，比二项堆有更好的渐近运行时间

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	1	1

n -堆中存储的元素个数

平摊分析

斐波那契堆平摊分析：堆的性能比较

定理. 从初始为空的斐波那契堆开始，任意执行由$a1$个插入，$a2$个删除，$a3$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$O(a1 + a2 \log n + a3)$.

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	1	1

n -堆中存储的元素个数　　平摊分析

斐波那契堆平摊分析：堆的性能比较

- 斐波那契堆对操作insert,union, decrease-key，比二项堆有更好的渐近运行时间
- 启示：在最小生成树和单源最短路径问题的快速计算中可以发挥巨大作用！

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	1	1

n -堆中存储的元素个数　　平摊分析

斐波那契堆平摊分析：堆的性能比较

- 斐波那契堆对操作insert,union, decrease-key，比二项堆有更好的渐近运行时间
- 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 $O(|E| + |V| \log |V|)$

斐波那契堆平摊分析：堆的性能比较

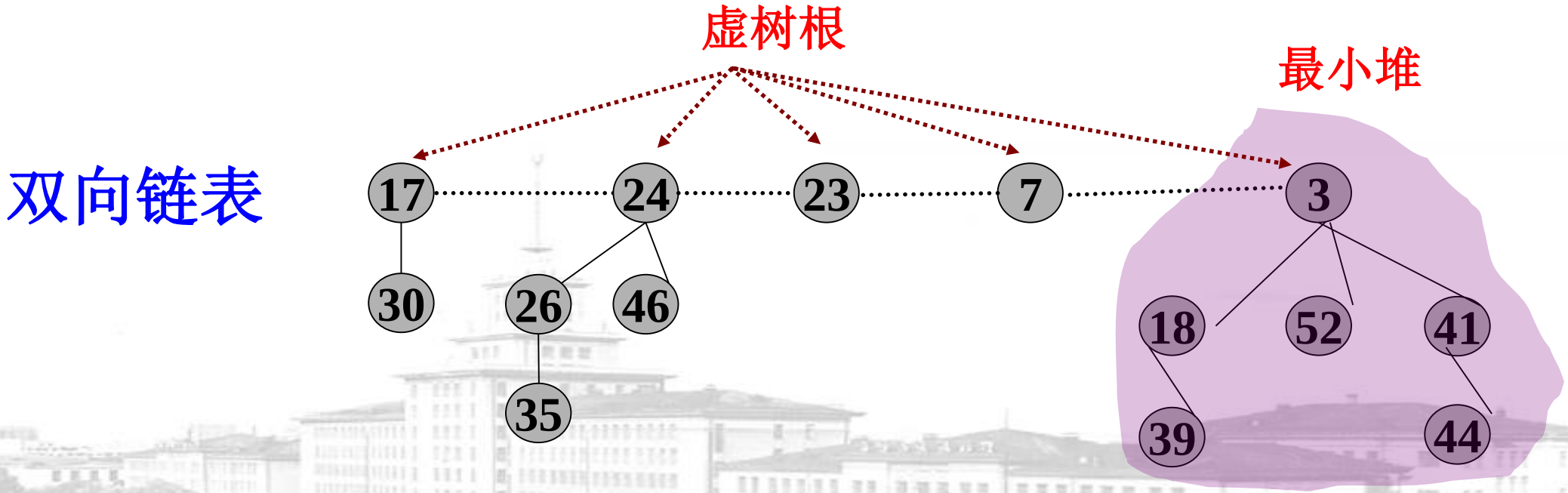
- 斐波那契堆的一个侧面：
  - 编程实现复杂
  - 与二项堆对**search**操作均低效：找到给定关键字的元素
- 斐波那契堆的最新研究成果

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*

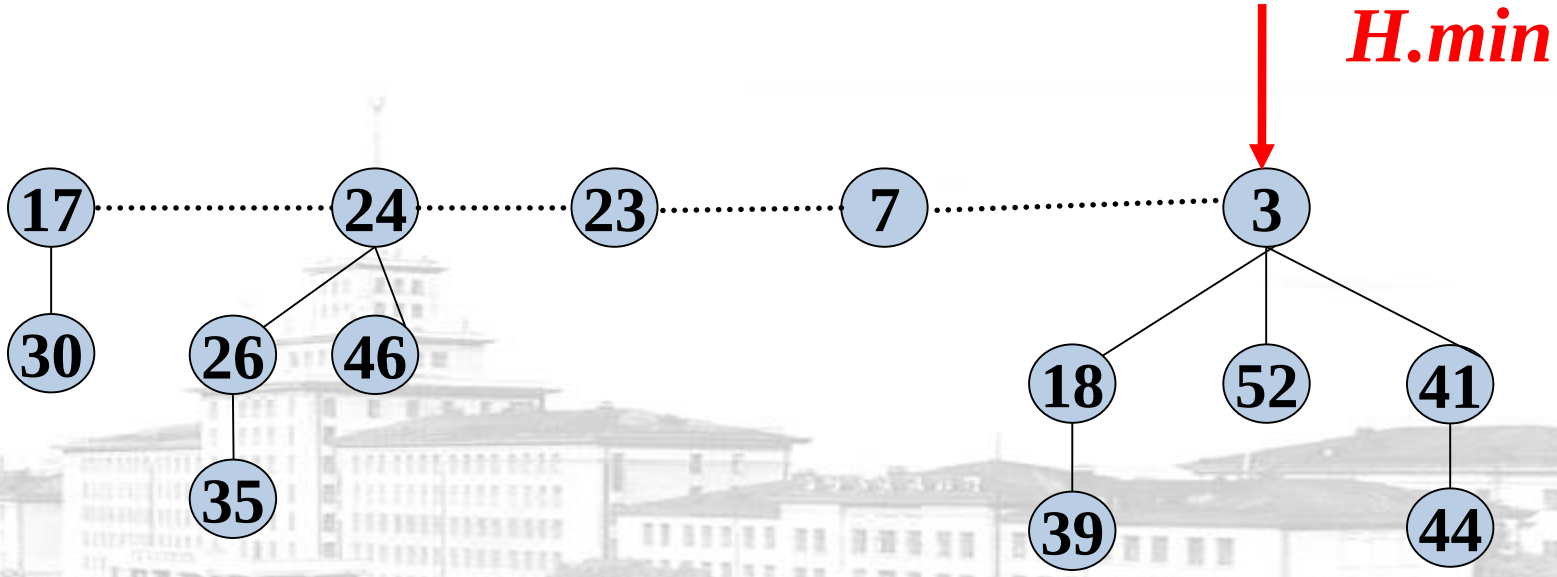
斐波那契堆平摊分析：结构

- 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
  - 5棵最小堆树和14个节点的斐波那契堆：



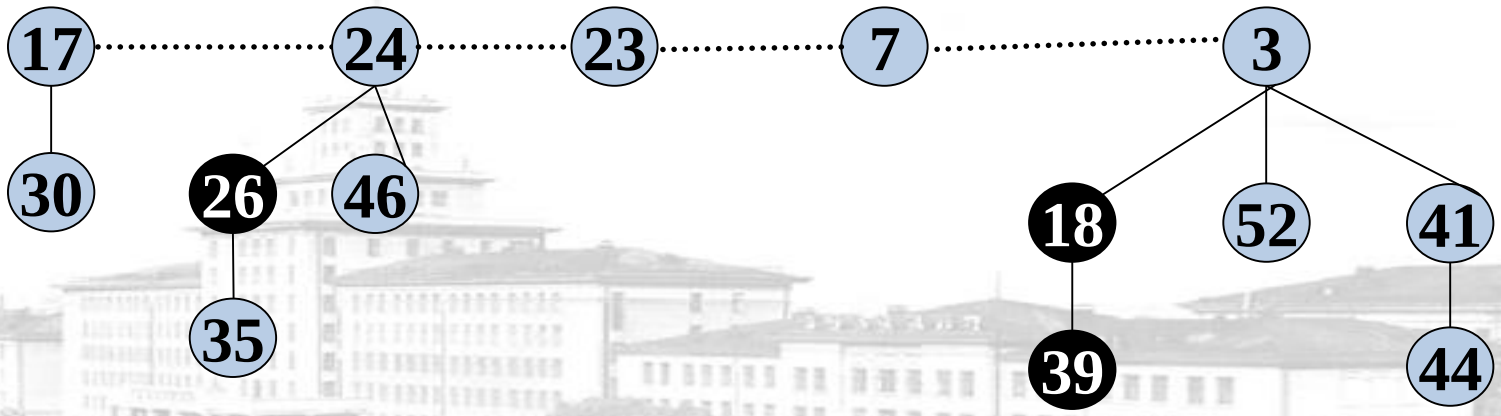
斐波那契堆平摊分析：结构

- 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
  - 从堆中查找最小元素(Find_min操作)的开销为O(1)



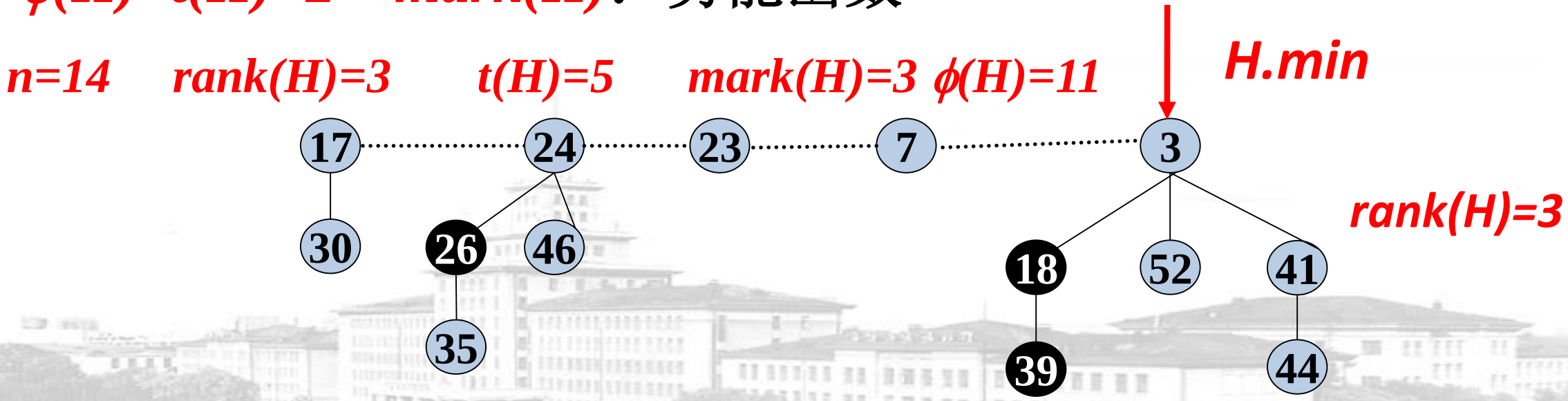
斐波那契堆平摊分析：结构

- 斐波那契堆一系列树，每棵树均是一个最小堆
- 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



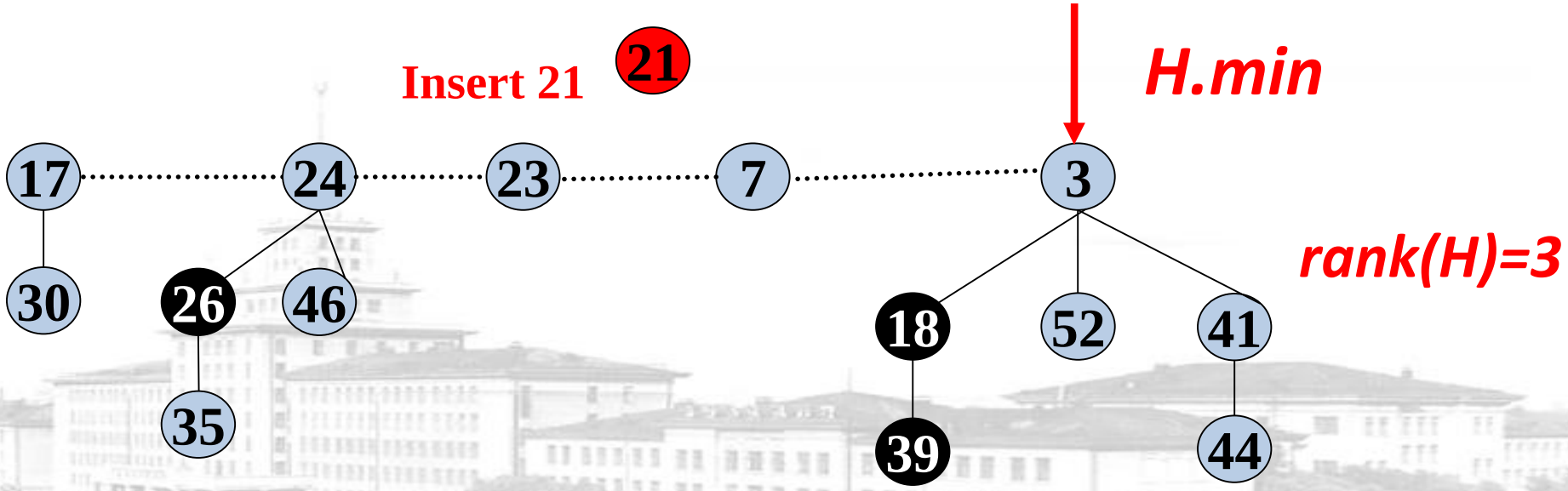
斐波那契堆平摊分析：一些记号

- 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
- $\phi(H)=t(H)+2$ $mark(H)$：势能函数



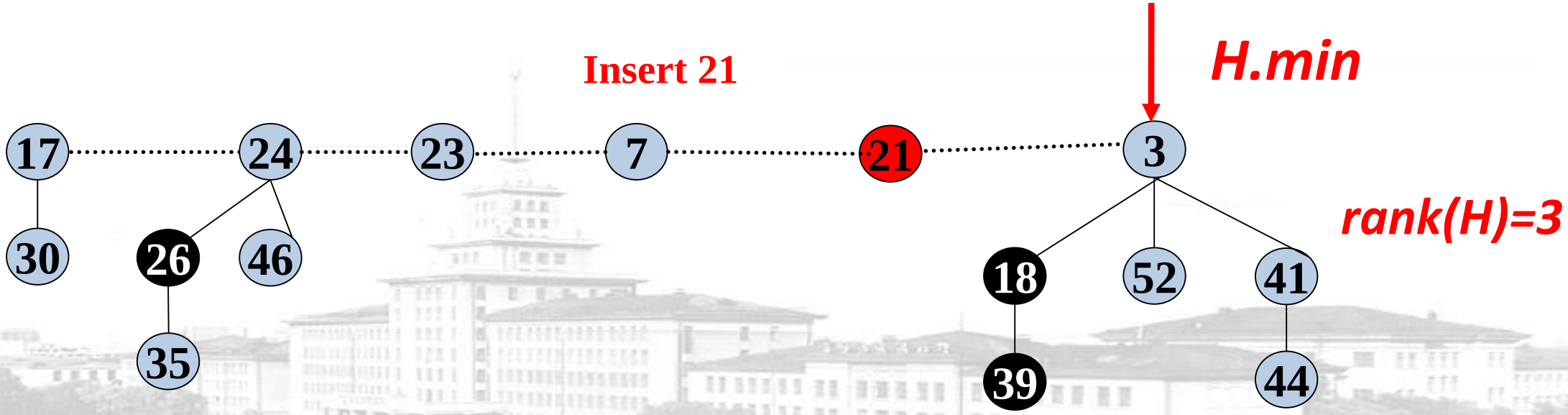
斐波那契堆平摊分析：insert操作

- 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



斐波那契堆平摊分析：insert操作

- 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



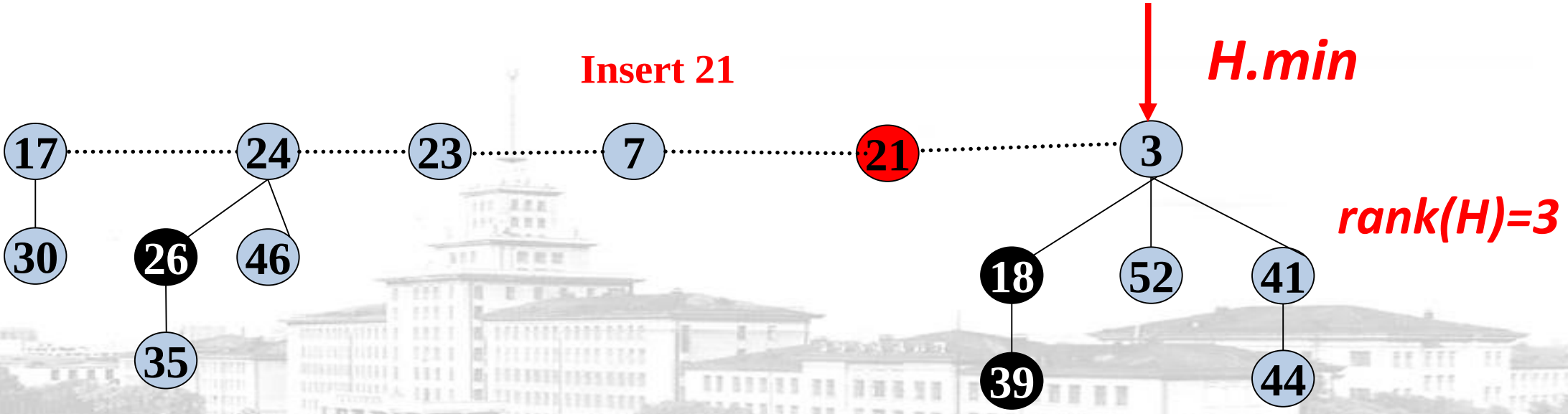
斐波那契堆平摊分析：insert操作

- 插入操作的代价分析：（势能函$\Phi(H)=t(H)+2\cdot mark(H)$）
  - 实际开销O(1)
  - 平摊代价O(1)

$t(H)$:堆H中树的棵数

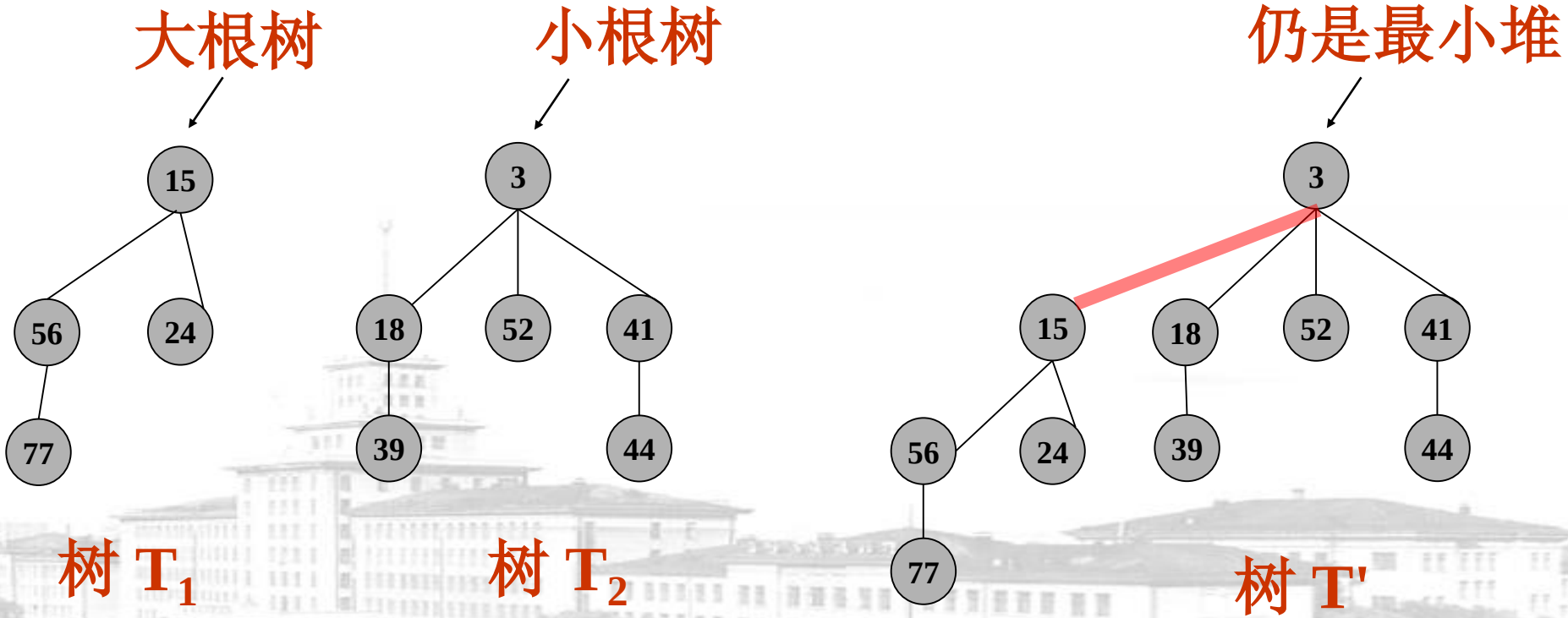
$mark(H)$:堆H中被标记顶点的个数

  - 操作完成后，$t(H)$增大1，但mark(H)不变



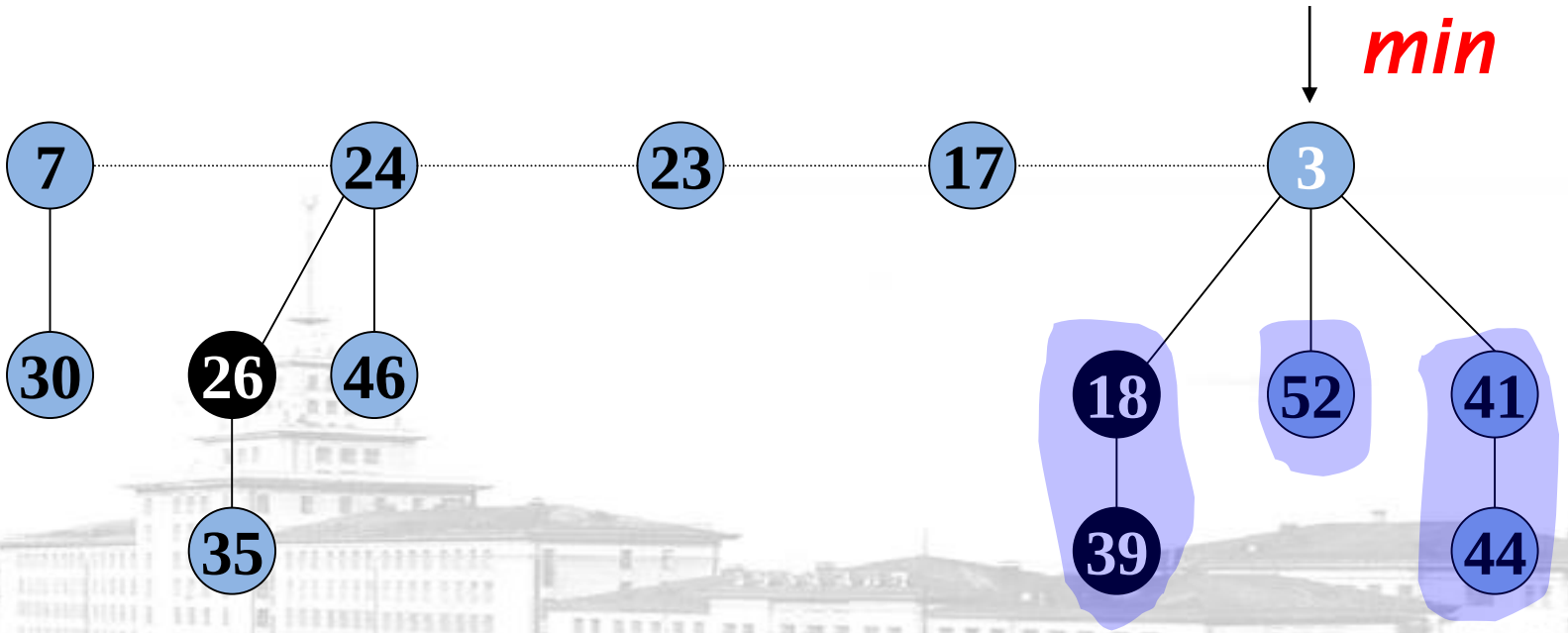
斐波那契堆平摊分析：Delete_min操作

- 删除堆顶元素： 连接调整操作
  - 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$O(1)$



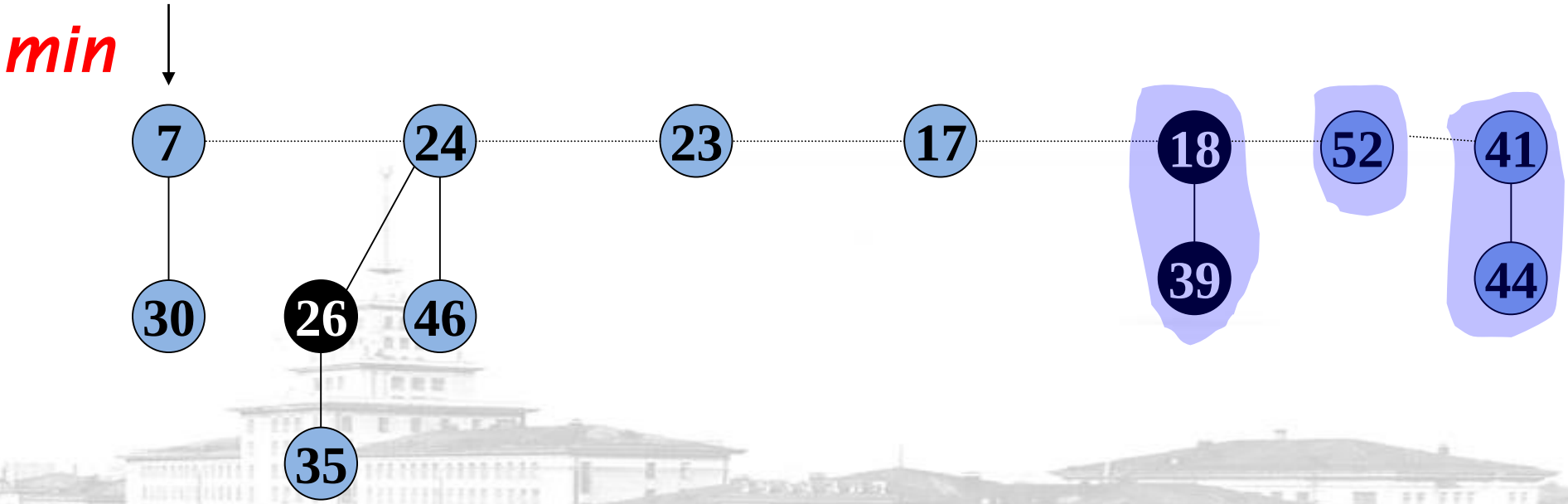
斐波那契堆平摊分析：Delete_min操作

- **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



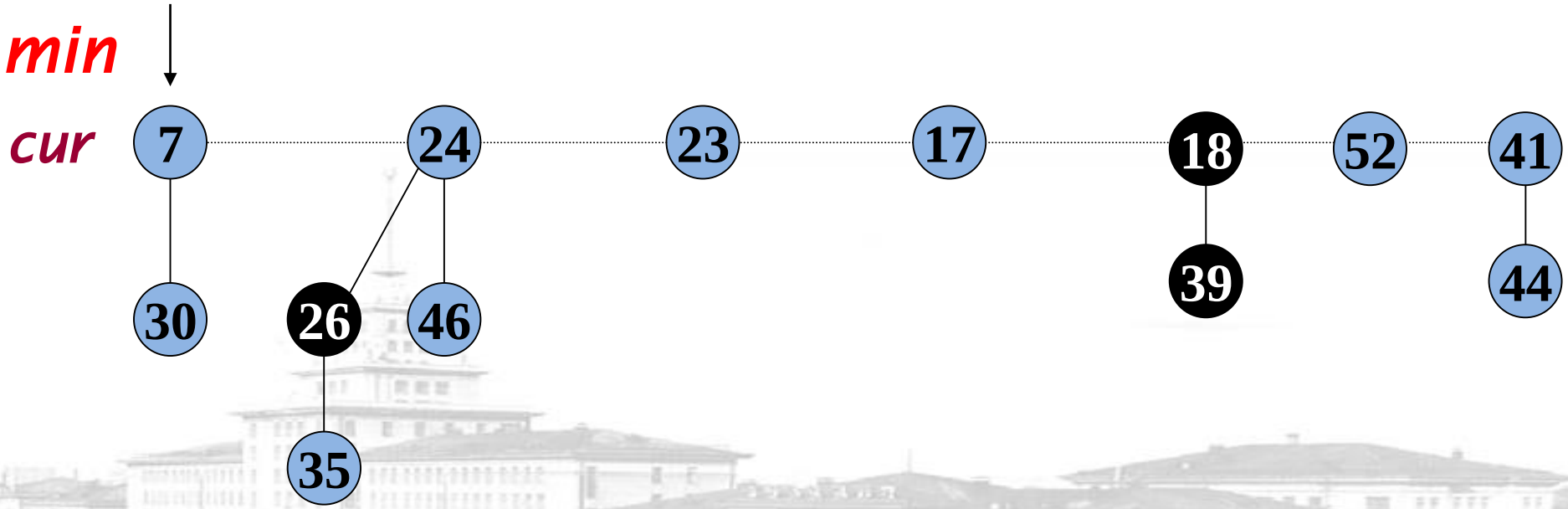
斐波那契堆平摊分析：Delete_min操作

- **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$t(H)+rank(x)$
  - 合并双向链表中的树根，使得没有树根具有相同rank



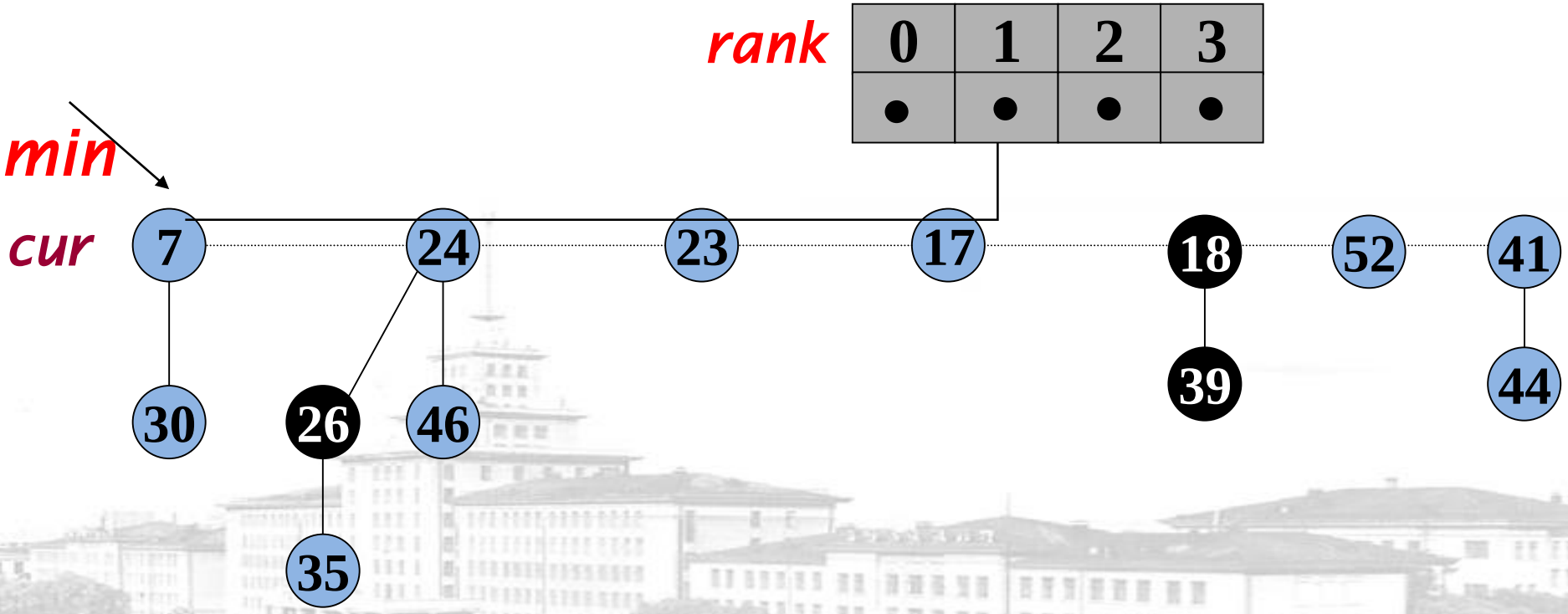
斐波那契堆平摊分析：Delete_min操作

- **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***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**



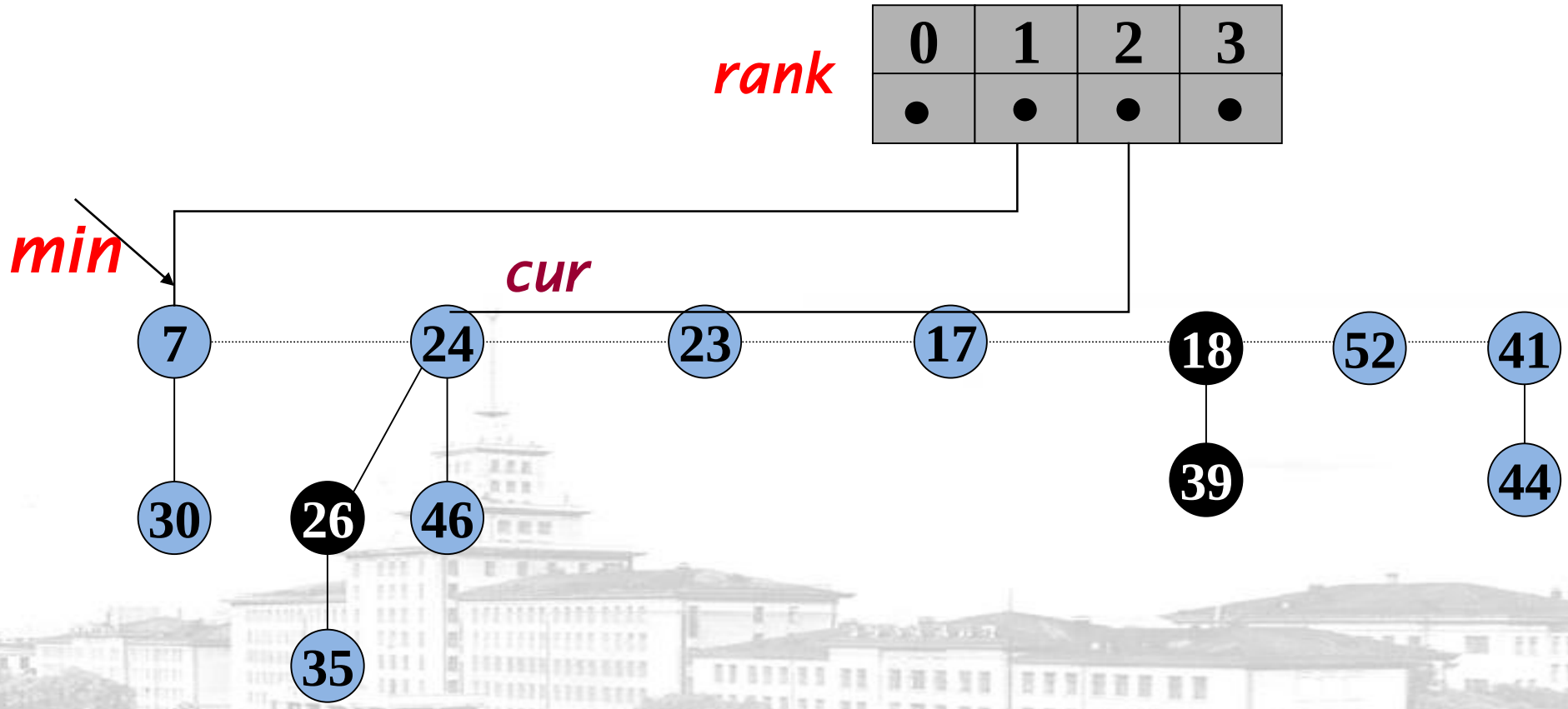
斐波那契堆平摊分析：Delete_min操作

- **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***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**



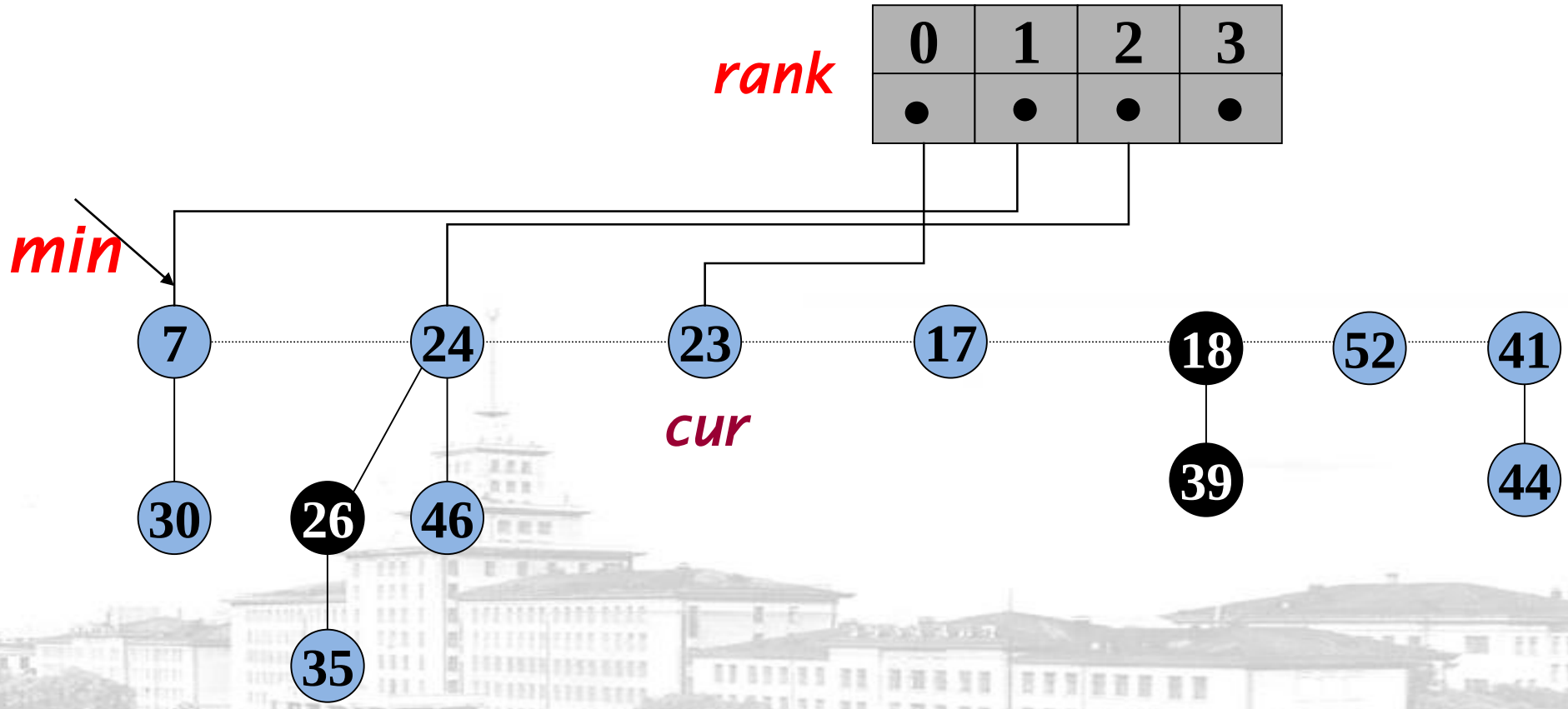
斐波那契堆平摊分析：Delete_min操作

- **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**



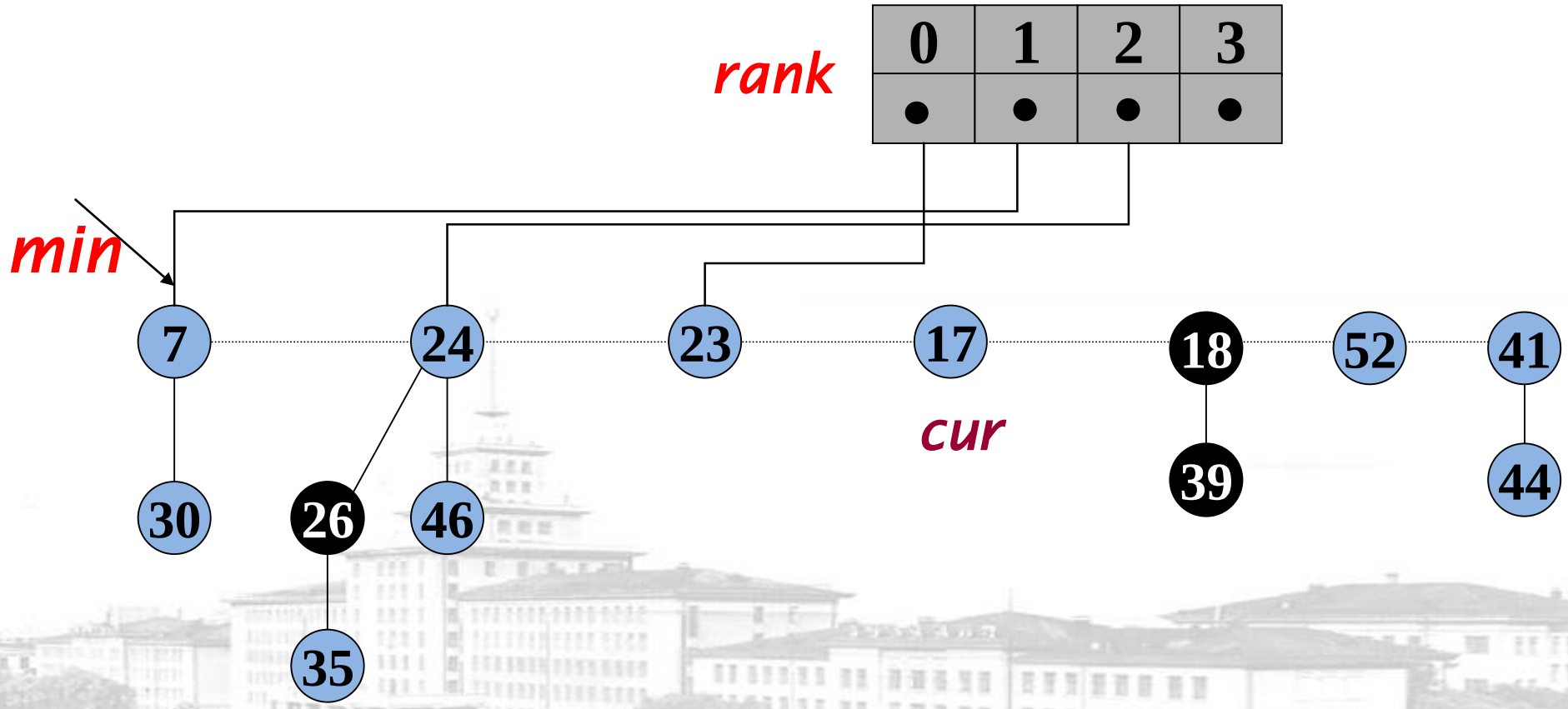
斐波那契堆平摊分析：Delete_min操作

- **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**



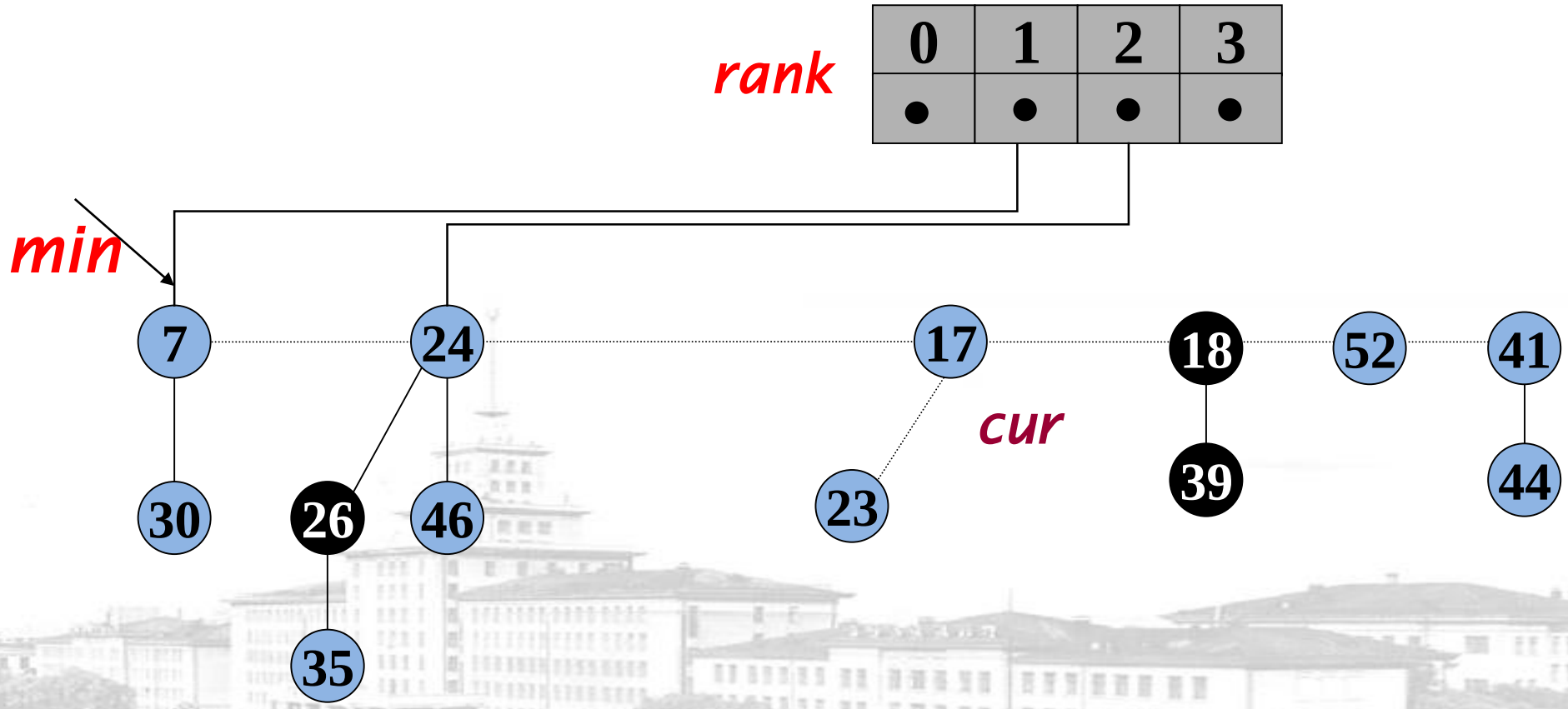
斐波那契堆平摊分析：Delete_min操作

- **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**



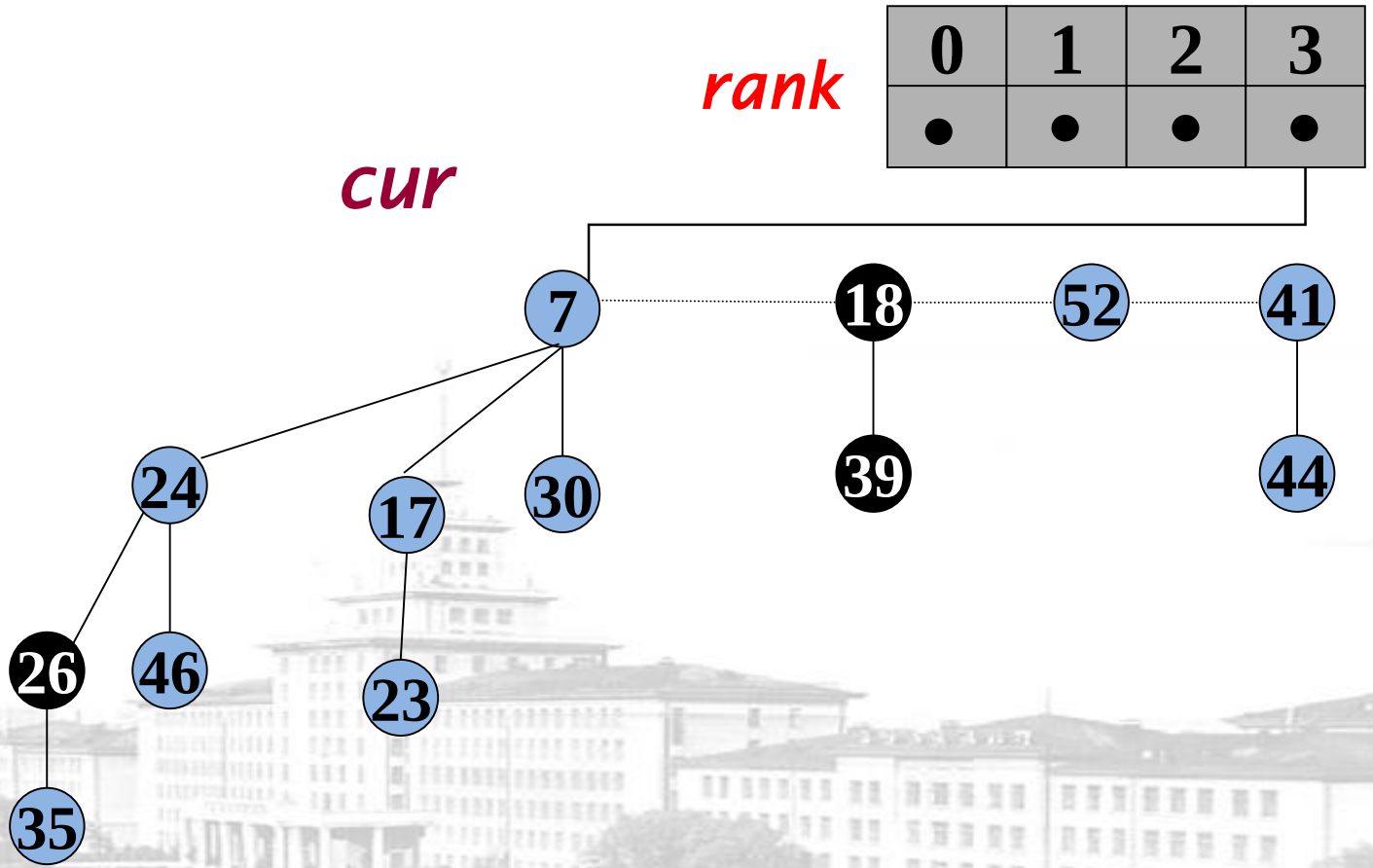
斐波那契堆平摊分析：Delete_min操作

- **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**



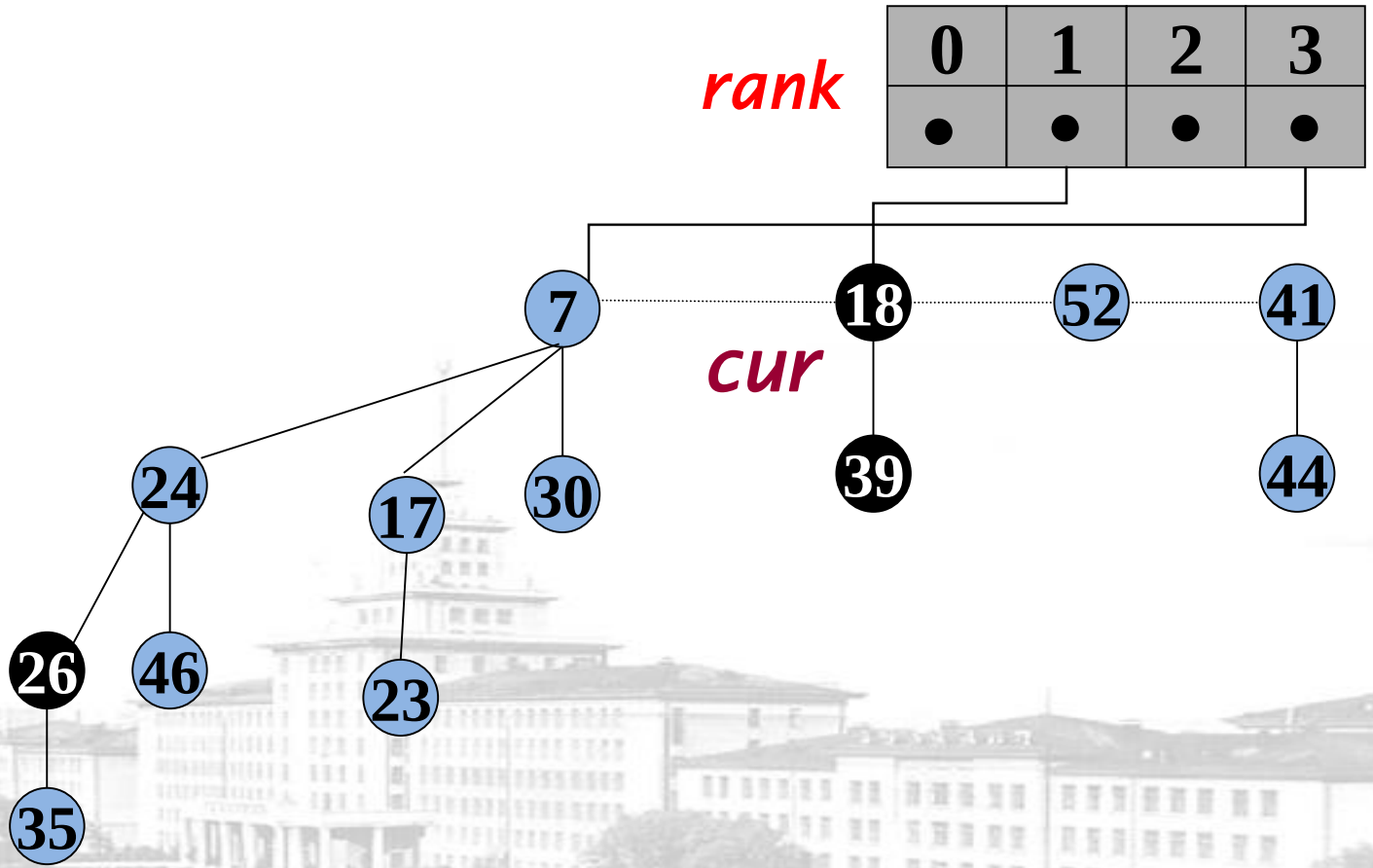
斐波那契堆平摊分析：Delete_min操作

- **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**



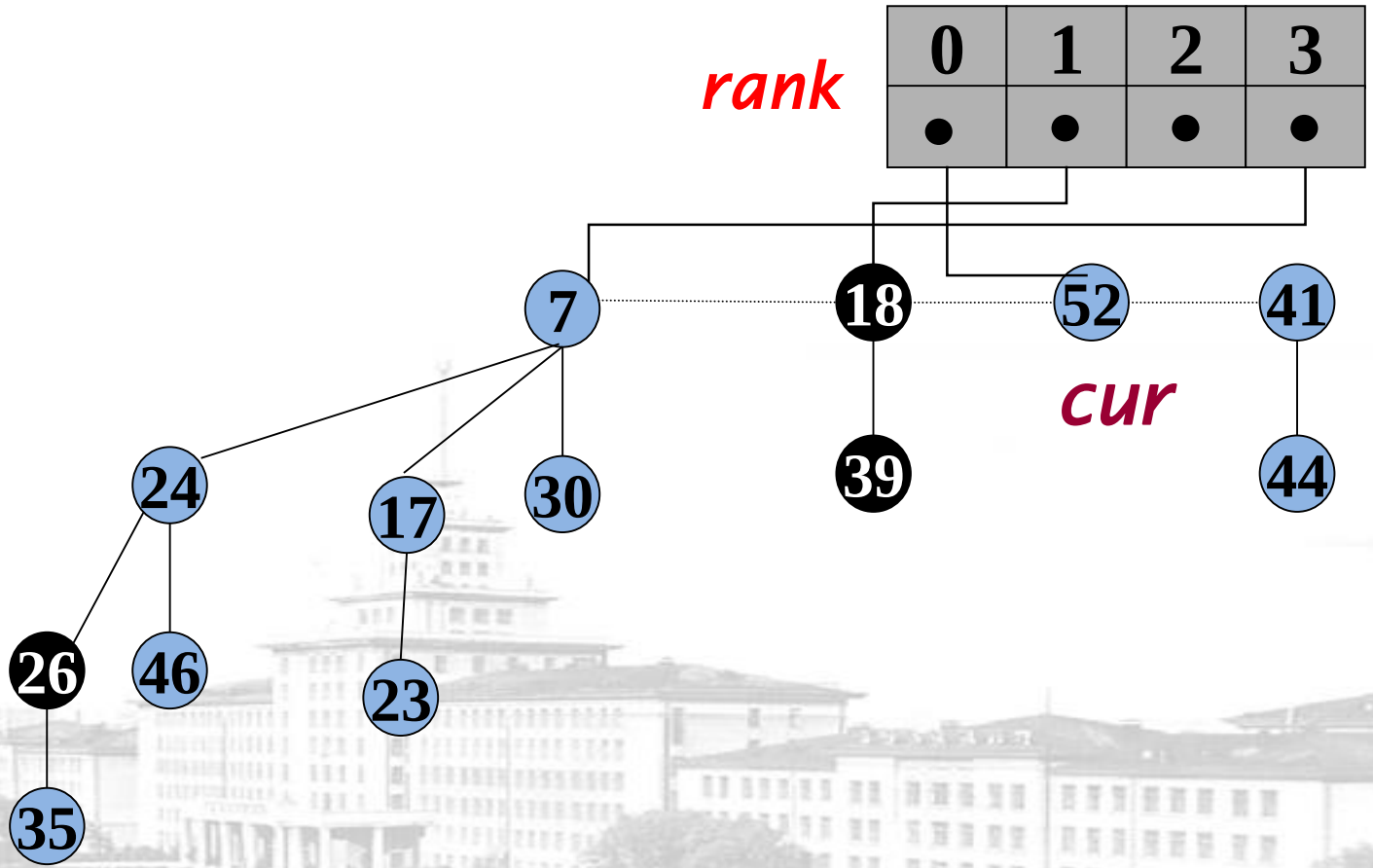
斐波那契堆平摊分析：Delete_min操作

- **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**



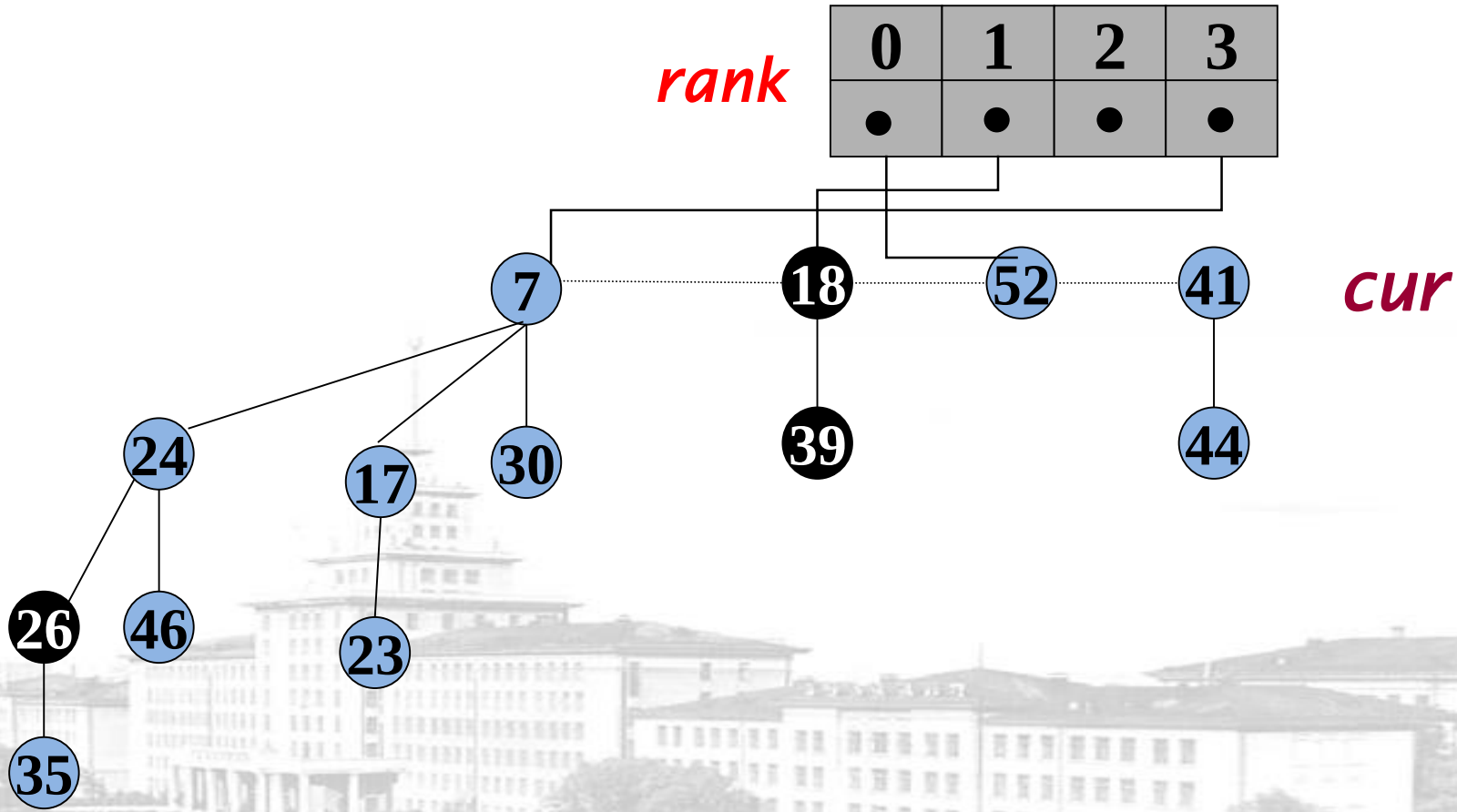
斐波那契堆平摊分析：Delete_min操作

- **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**



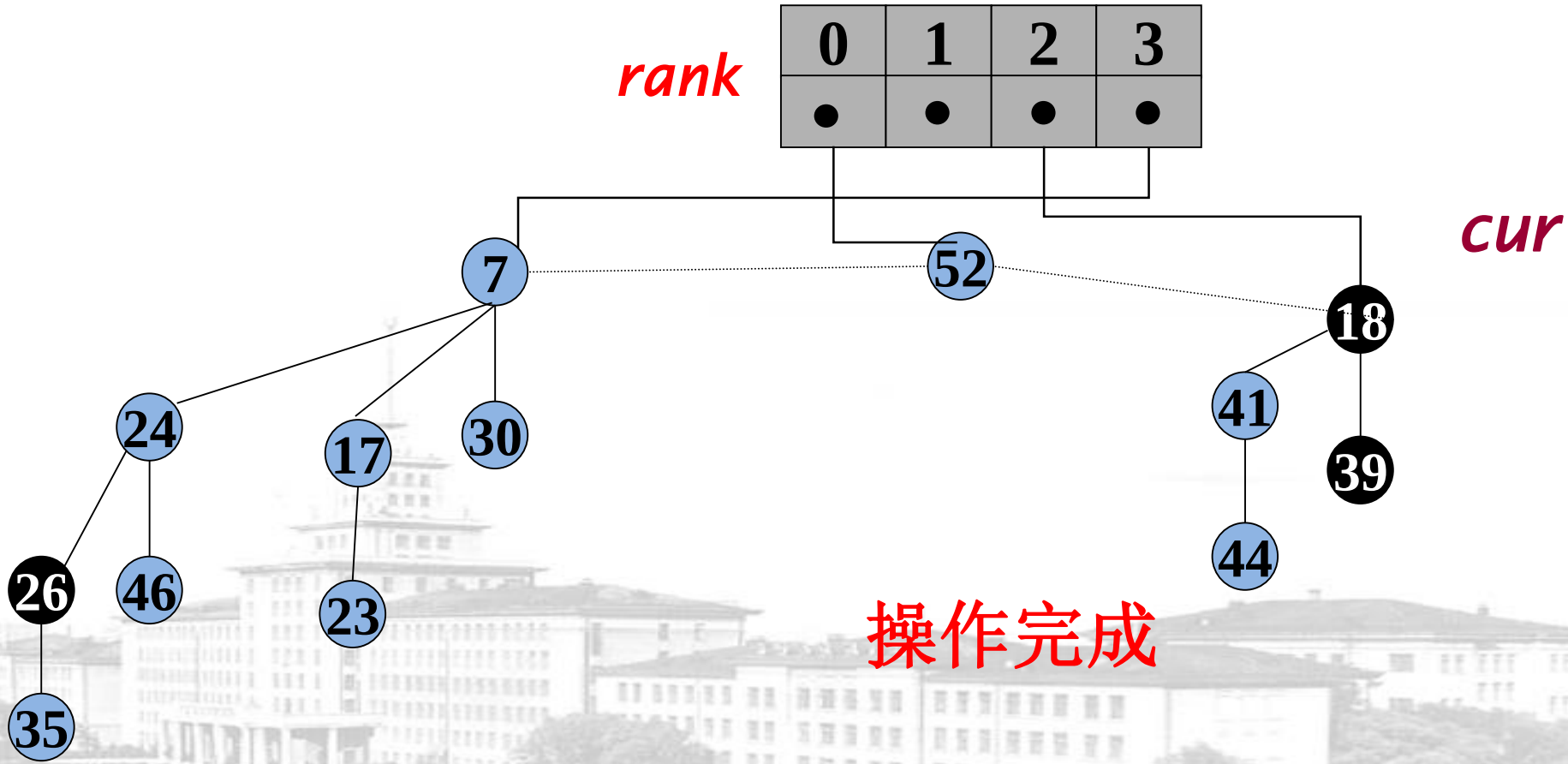
斐波那契堆平摊分析：Delete_min操作

- **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**



斐波那契堆平摊分析：Delete_min操作

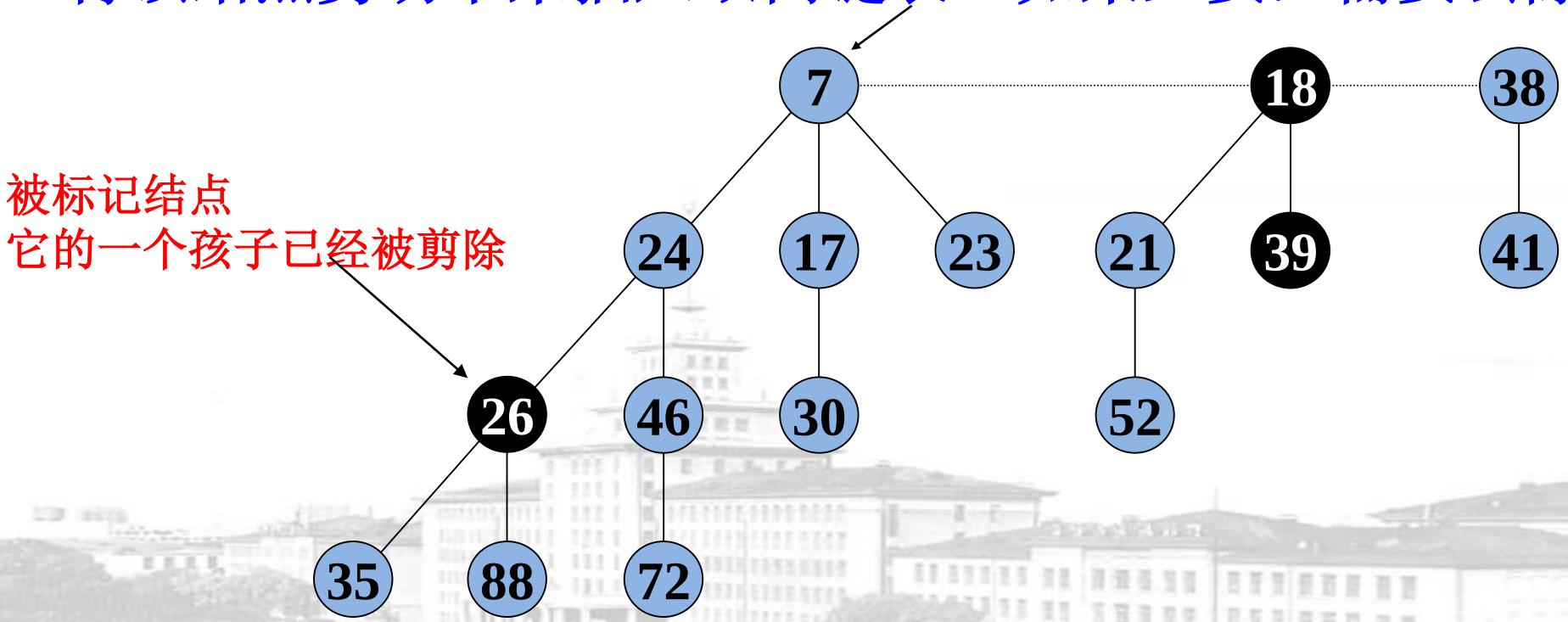
- **Delete_min**操作过程
  - 删除最小元素；将其所有孩子链入双向链表，更新最小元素指针
  - 合并双向链表中的树根，使得没有树根具有相同**rank**



斐波那契堆平摊分析：Decrease_key操作

键值减小操作： 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）

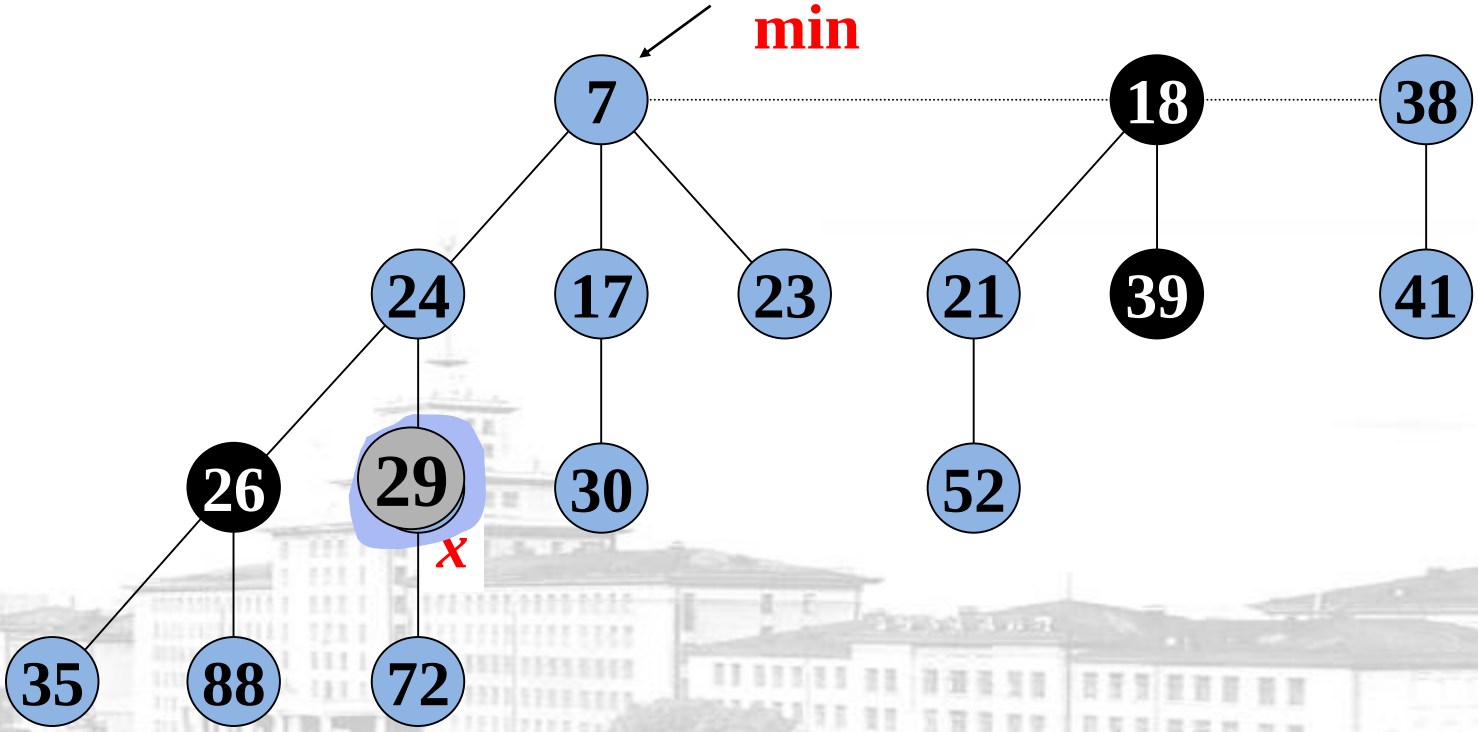


斐波那契堆平摊分析：Decrease_key操作

- 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

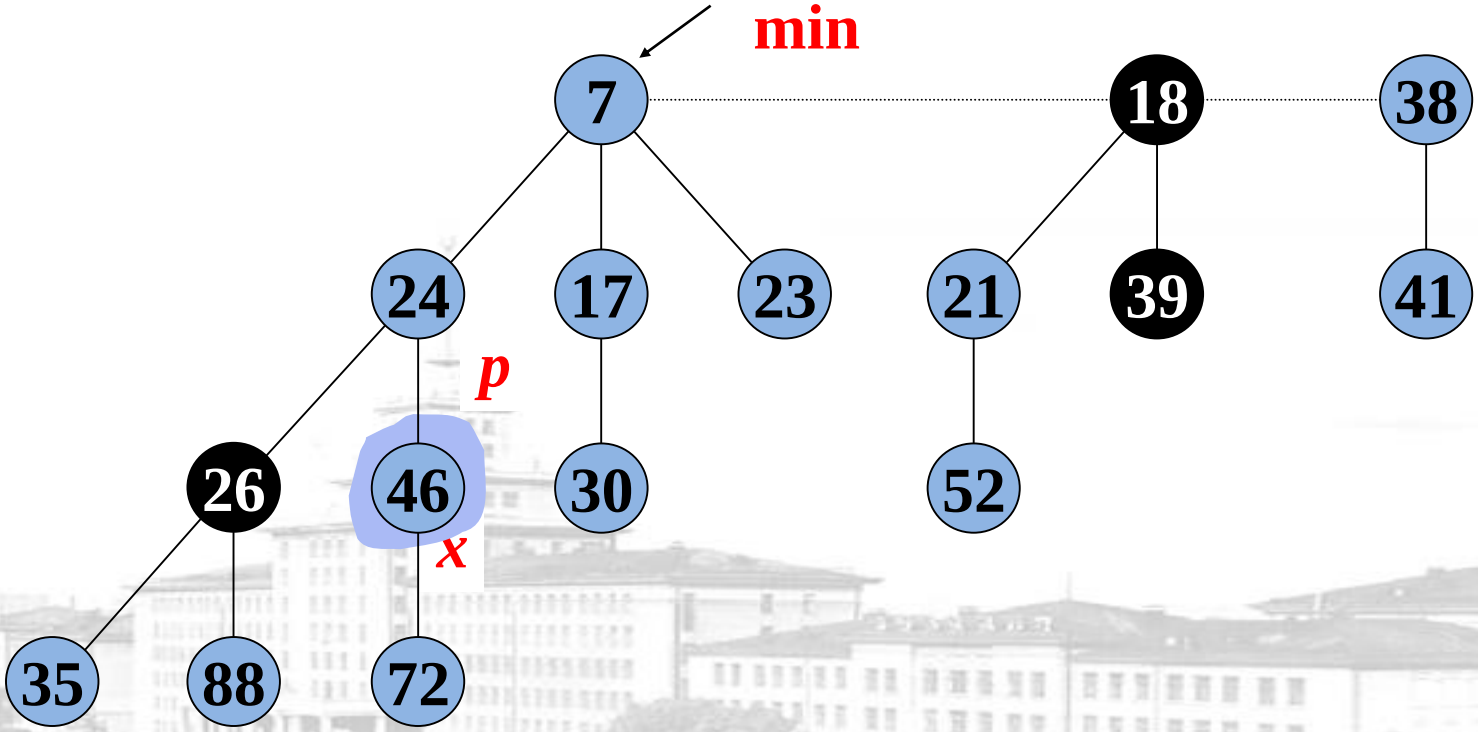


斐波那契堆平摊分析：Decrease_key操作

- 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

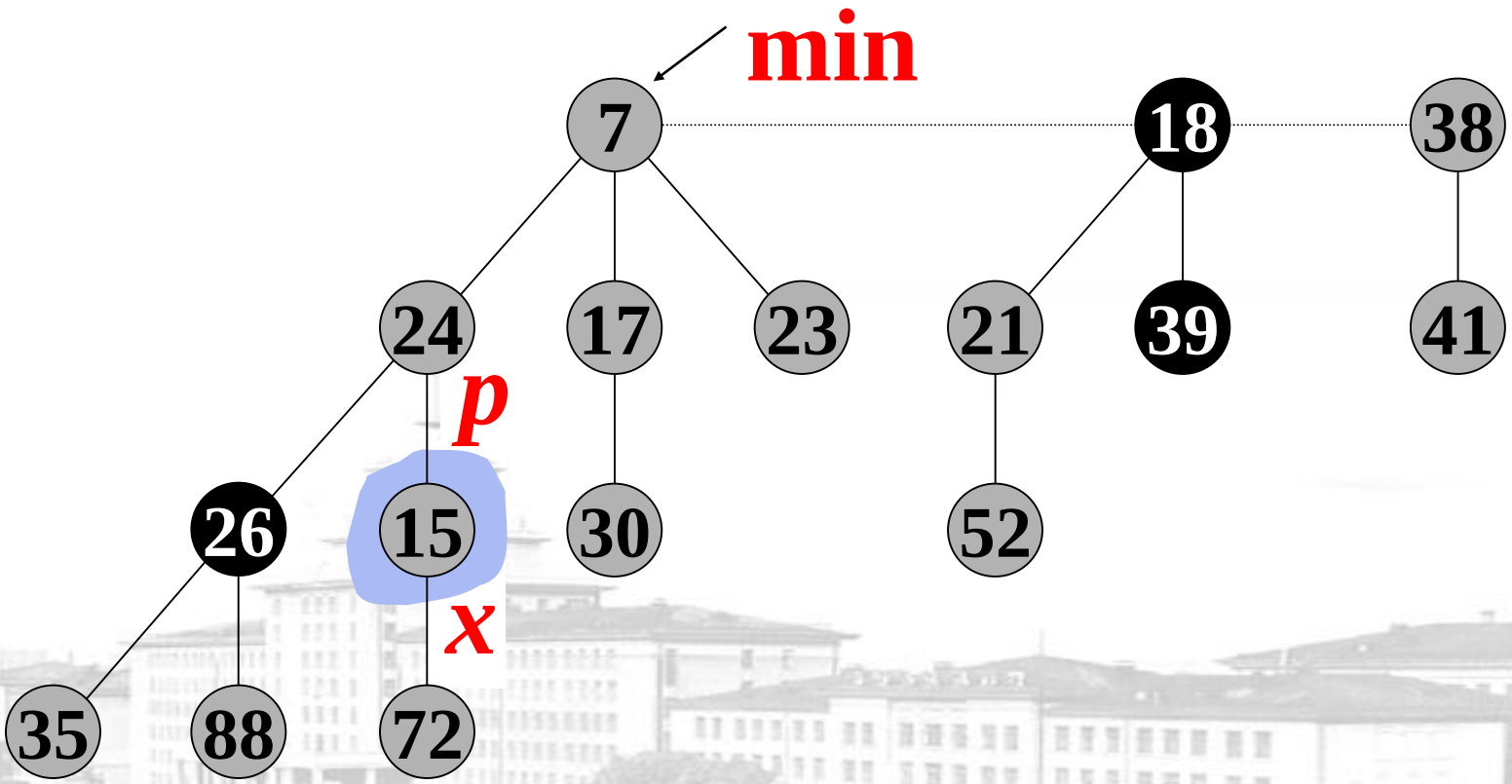


斐波那契堆平摊分析：Decrease_key操作

- 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



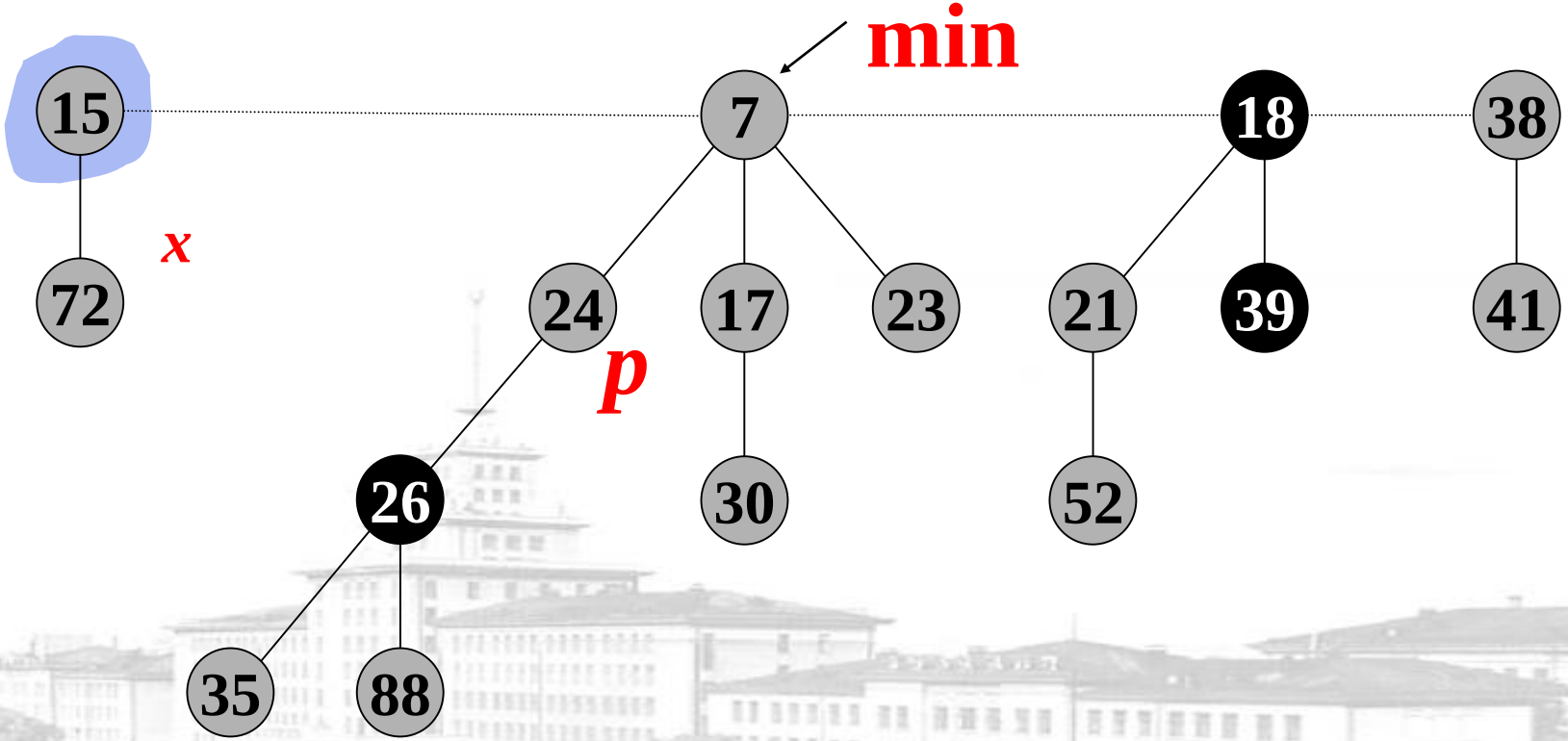
斐波那契堆平摊分析：Decrease_key操作

- 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



斐波那契堆平摊分析：Decrease_key操作

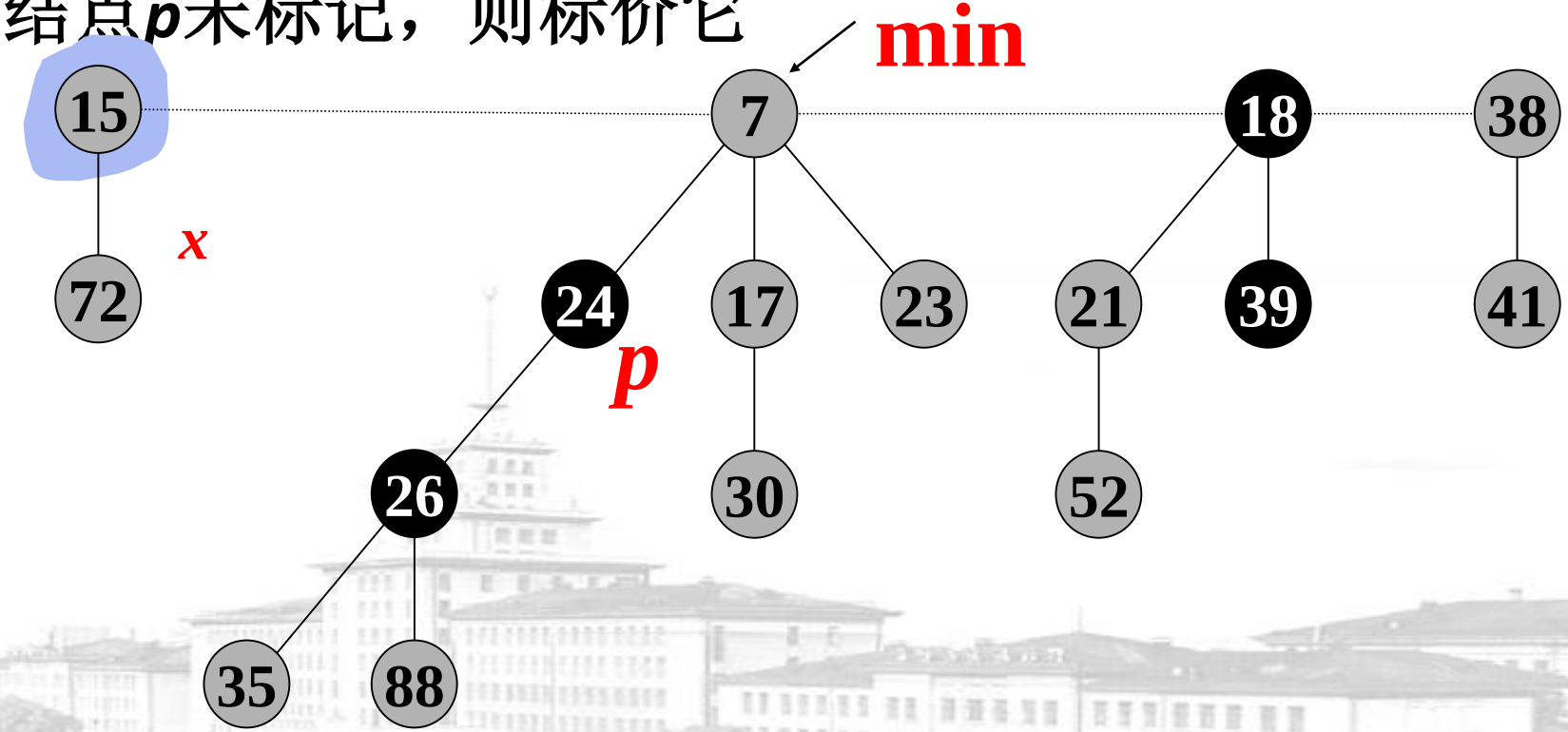
- 键值减小操作作用于结点*x*（例如，键值从46减小为15）

  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



斐波那契堆平摊分析：Decrease_key操作

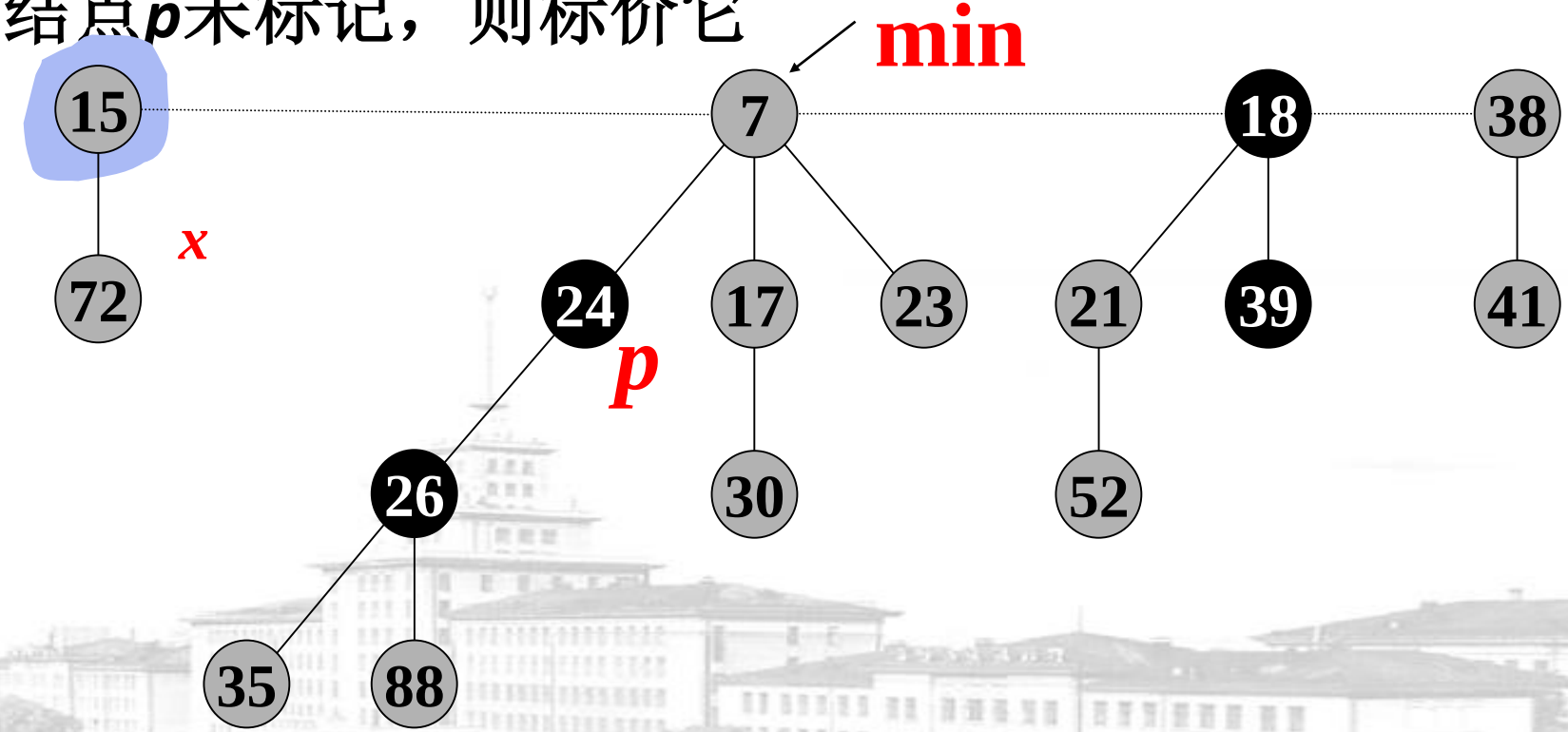
- 键值减小操作作用于结点*x*（例如，键值从46减小为15）

  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



斐波那契堆平摊分析：Decrease_key操作

- 键值减小操作作用于结点*x*（例如，键值从46减小为15）

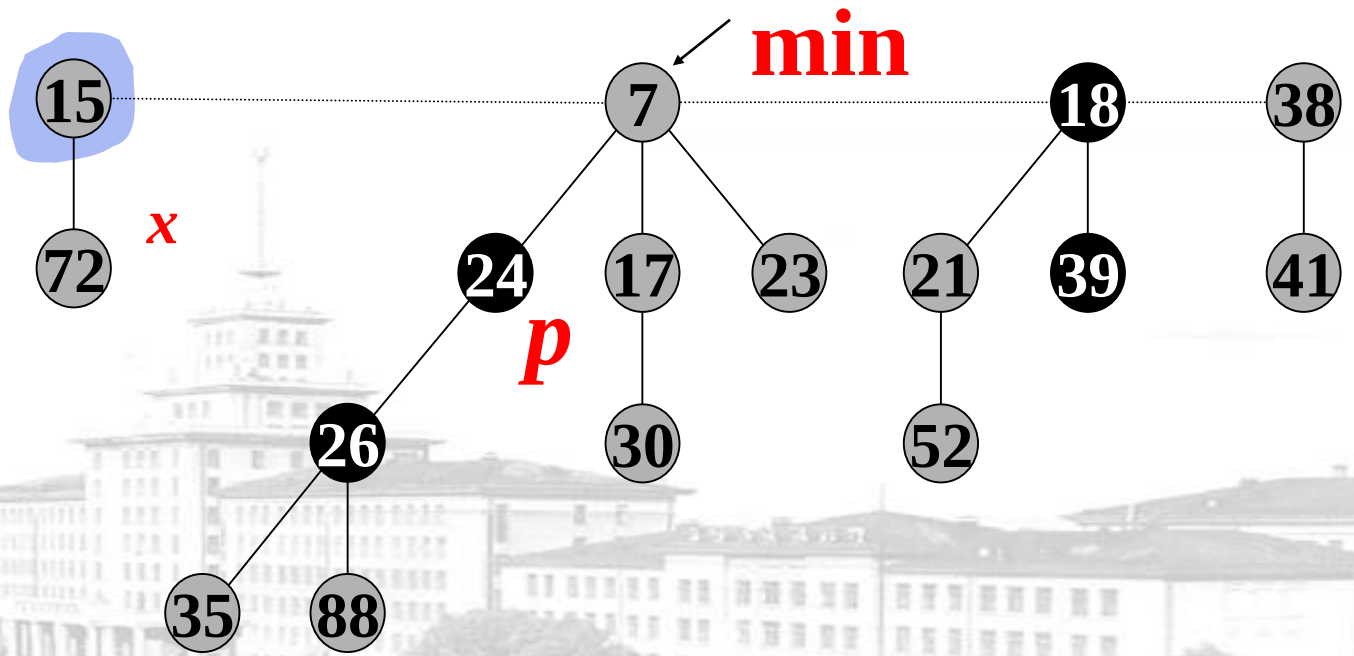
情形2a：减小键值后堆性质不成立

•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）



斐波那契堆平摊分析：Decrease_key操作

- 键值减小操作作用于结点*x*（例如，键值从35减小为5）

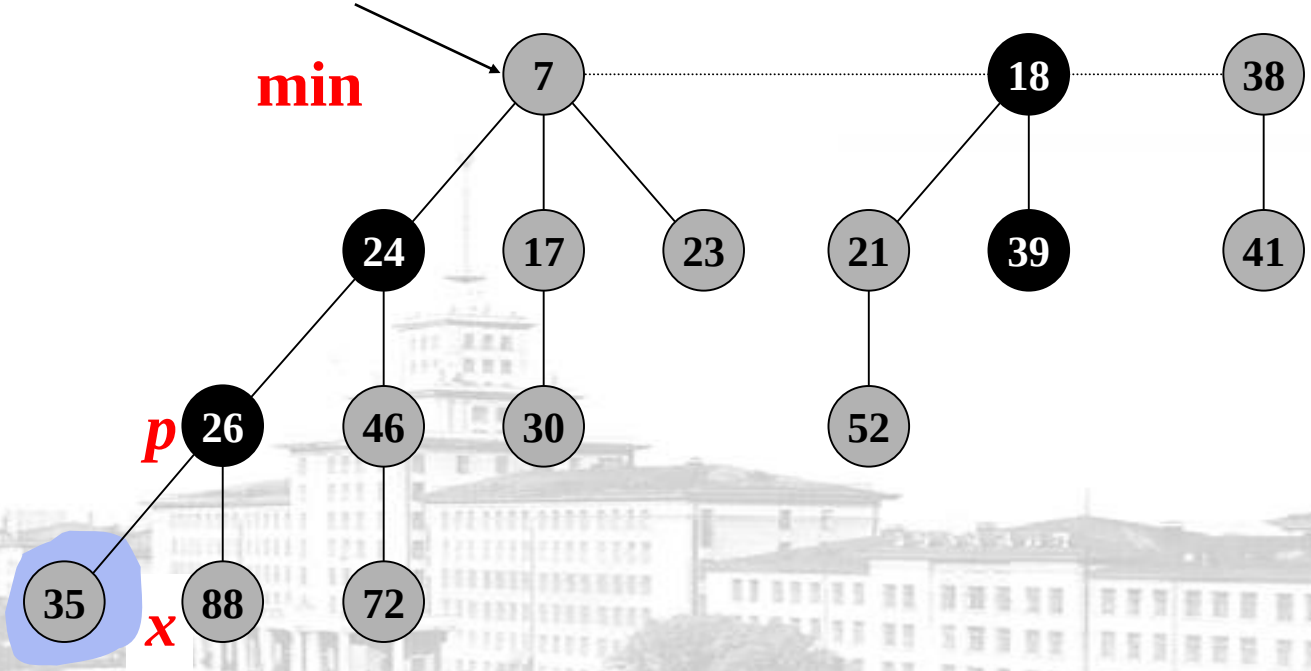
  情形2b：减小键值后堆性质不成立

  •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）



斐波那契堆平摊分析：Decrease_key操作

- 键值减小操作作用于结点*x*（例如，键值从35减小为5）

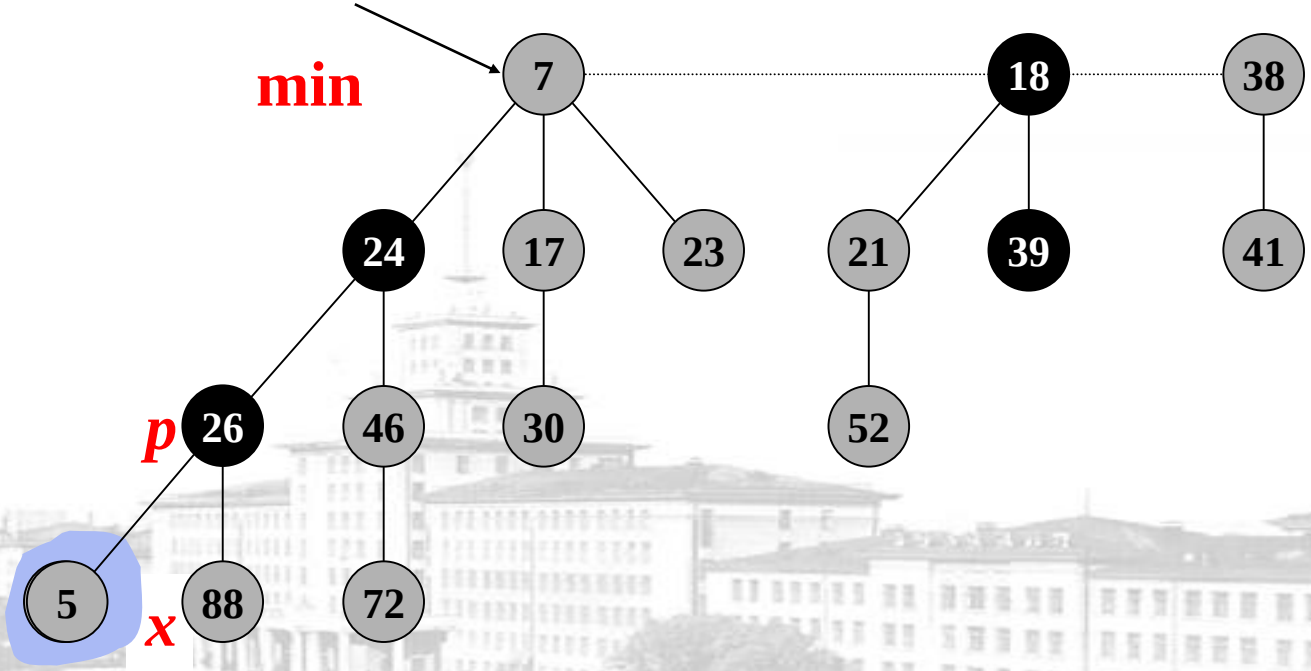
  情形2b：减小键值后堆性质不成立

  •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）



斐波那契堆平摊分析：Decrease_key操作

- 键值减小操作作用于结点*x*（例如，键值从35减小为5）

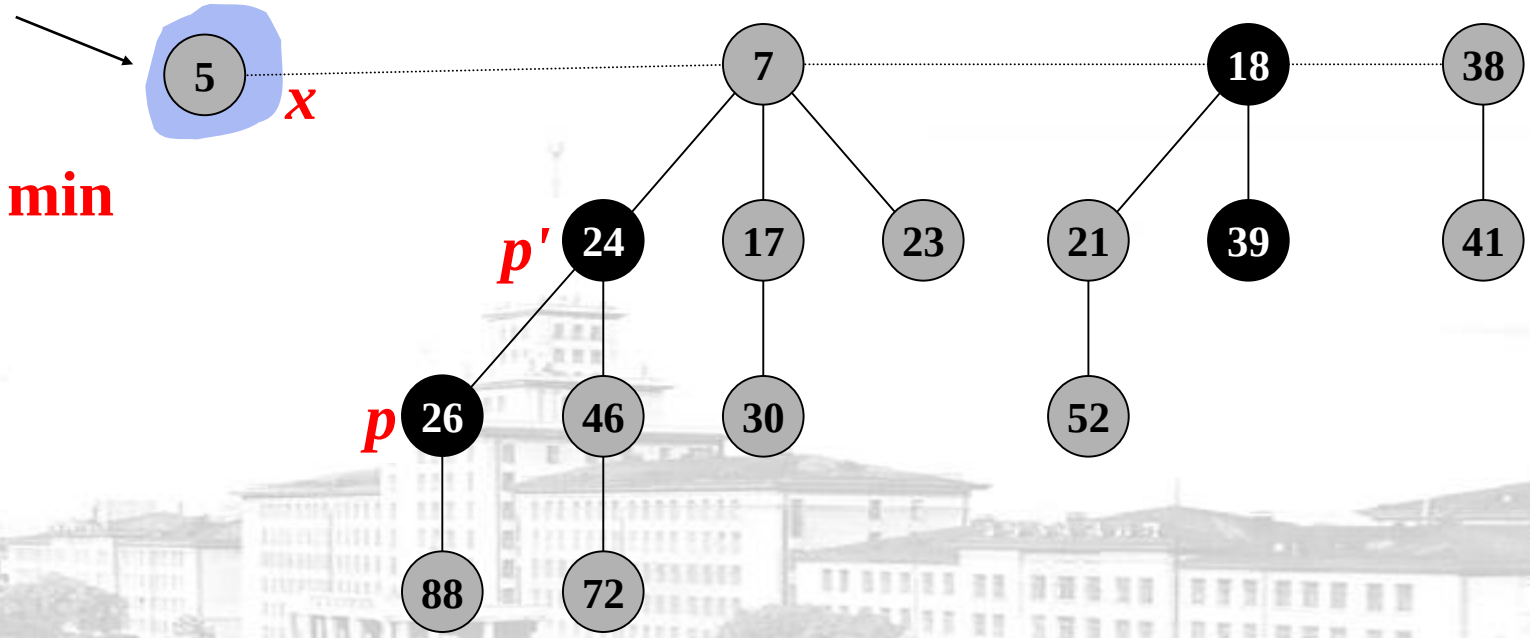
  情形2b：减小键值后堆性质不成立

  •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）



斐波那契堆平摊分析：Decrease_key操作

- 键值减小操作作用于结点*x*（例如，键值从35减小为5）

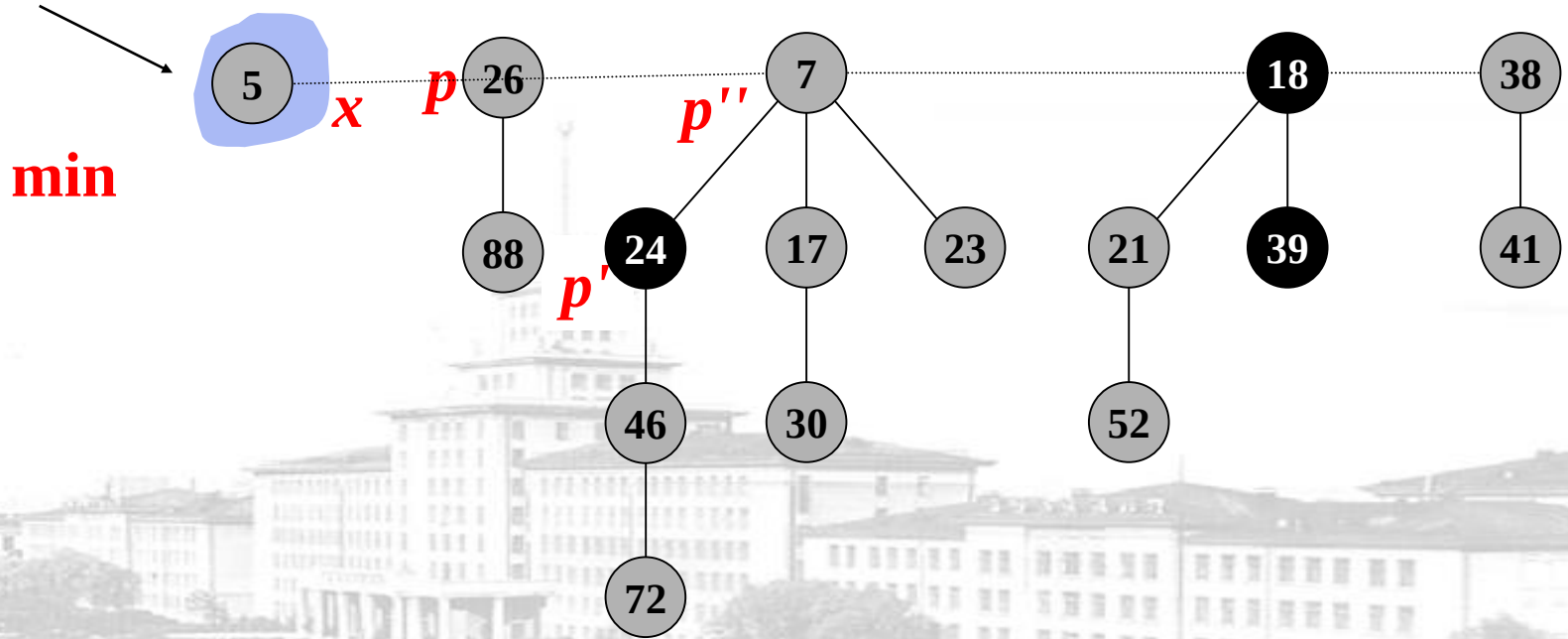
  情形2b：减小键值后堆性质不成立

  •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）



斐波那契堆平摊分析：Decrease_key操作

- 键值减小操作作用于结点*x*（例如，键值从35减小为5）

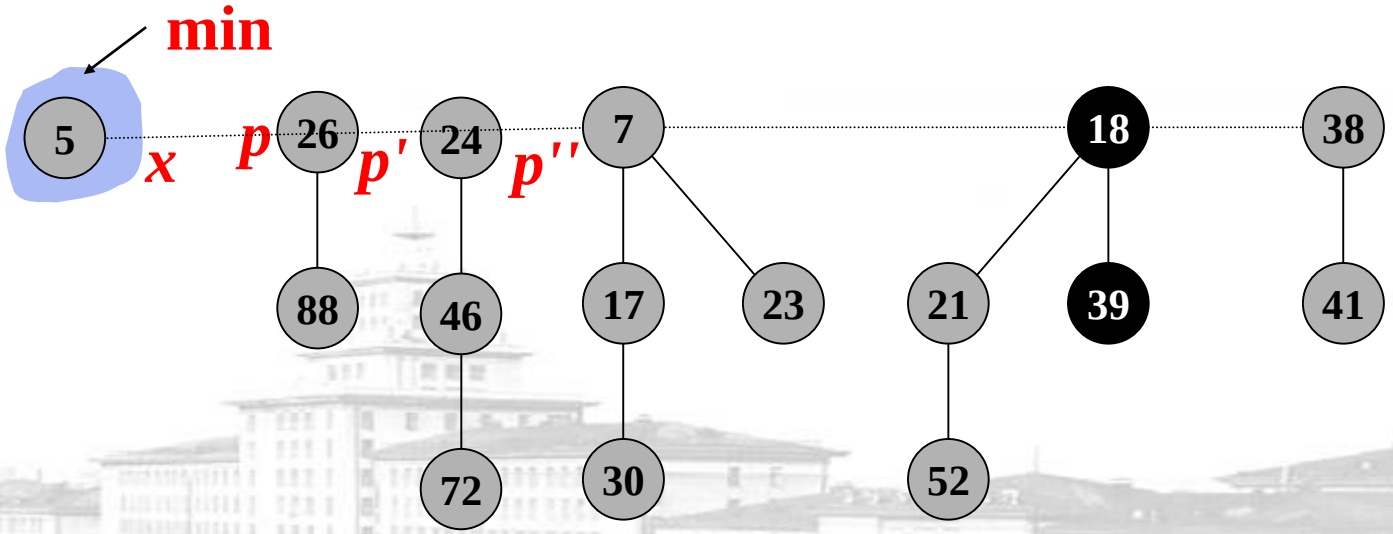
  情形2b：减小键值后堆性质不成立

  •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）



斐波那契堆平摊分析：Decrease_key操作

- 键值减小操作作用于结点*x*（例如，键值从35减小为5）

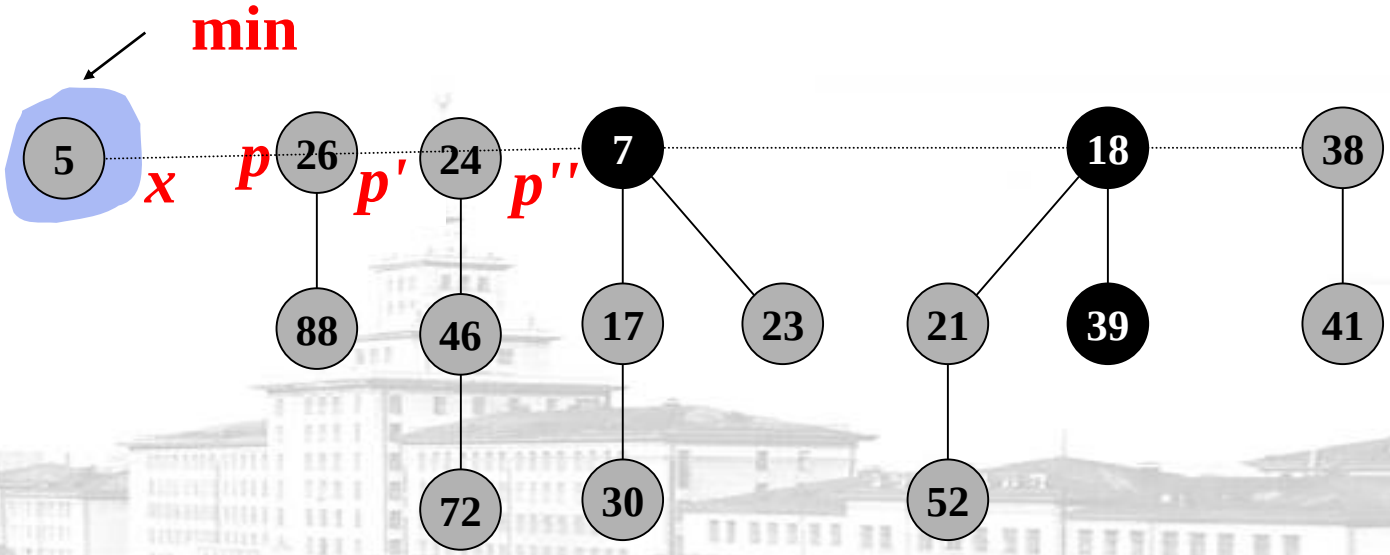
  情形2b：减小键值后堆性质不成立

  •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）



斐波那契堆平摊分析：Decrease_key操作分析

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

斐波那契堆平摊分析：操作的分析结果

- 在势能函数$\Phi(H)=t(H)+2\cdot mark(H)$下
- 创建一个新的斐波那契堆
  - H中不存在树，空H的$\Phi(H)=0$
  - Make-Fib-Heap的代价为O(1)

斐波那契堆平摊分析：操作的分析结果

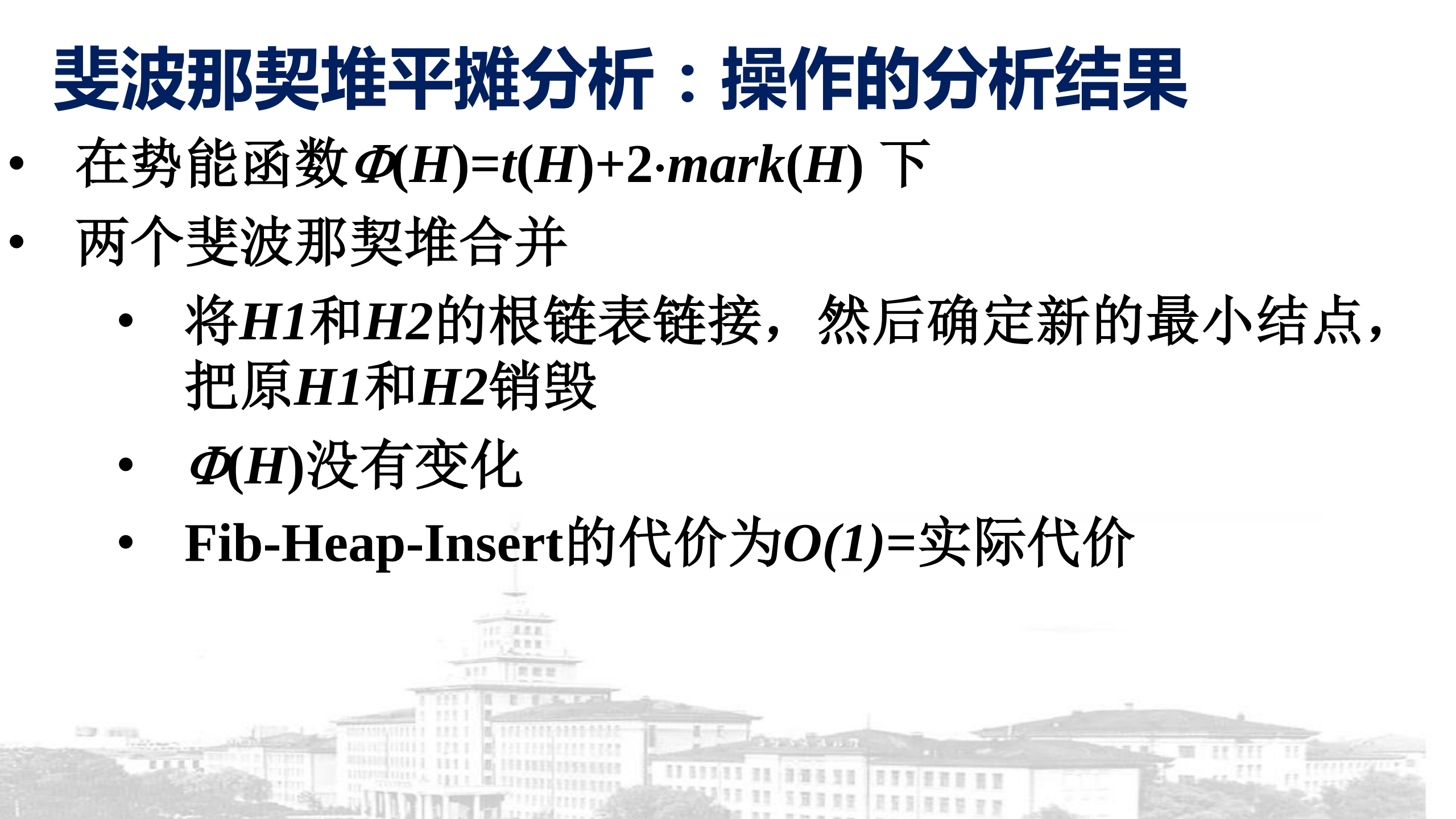
- 在势能函数$\Phi(H)=t(H)+2\cdot mark(H)$ 下
- 插入一个结点
  - 设H'是结果堆，$t(H')=t(H)+1$, $mark(H')=mark(H)$
  - $\Phi(H)$的增量为1
  - **Fib-Heap-Insert**的代价为$O(1)$=实际代价$O(1)+1=O(1)$

斐波那契堆平摊分析：操作的分析结果

- 在势能函数$\Phi(H)=t(H)+2\cdot mark(H)$ 下
- 寻找最小结点
  - 实际代价：通过指针$H.min$在$O(1)$时间内找到
  - $\Phi(H)$没有变化
  - **Find-min**的代价为$O(1)$=实际代价$O(1)$

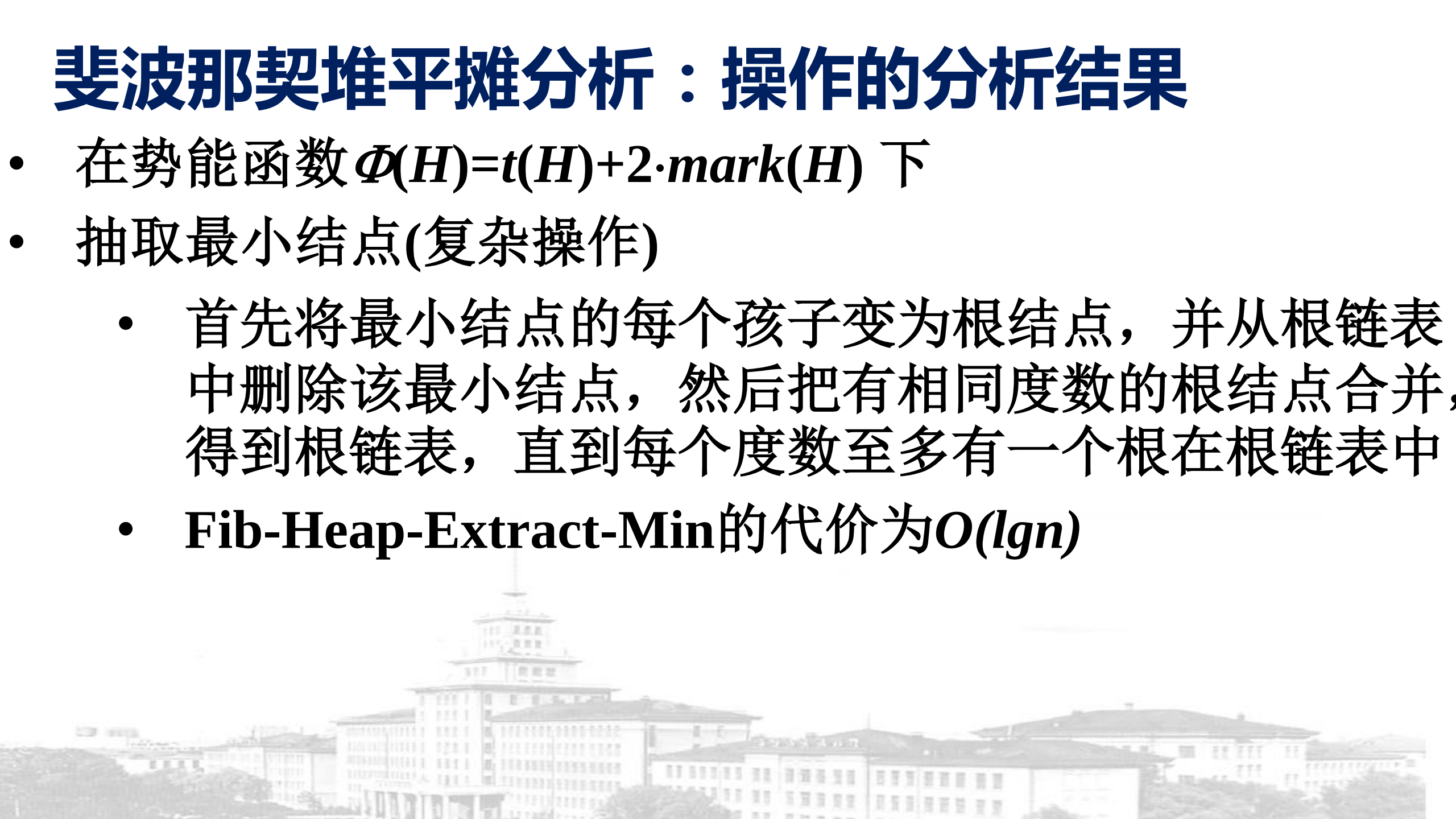
斐波那契堆平摊分析：操作的分析结果

- 在势能函数$\Phi(H)=t(H)+2\cdot mark(H)$下
- 两个斐波那契堆合并
  - 将*H1*和*H2*的根链表链接，然后确定新的最小结点，把原*H1*和*H2*销毁
  - $\Phi(H)$没有变化
  - **Fib-Heap-Insert**的代价为$O(1)$=实际代价



斐波那契堆平摊分析：操作的分析结果

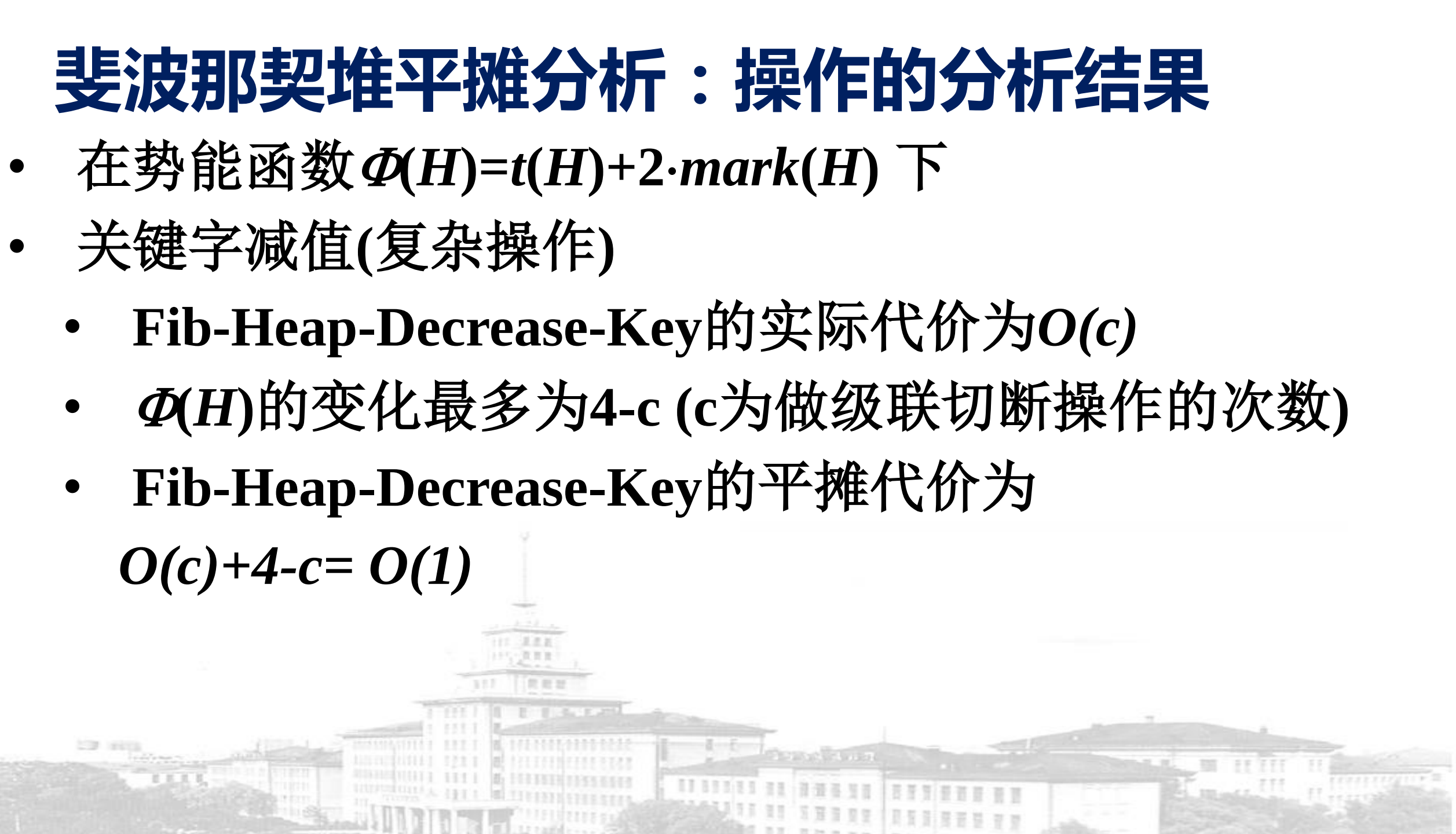
- 在势能函数$\Phi(H)=t(H)+2\cdot mark(H)$ 下
- 抽取最小结点(复杂操作)
  - 首先将最小结点的每个孩子变为根结点，并从根链表中删除该最小结点，然后把有相同度数的根结点合并，得到根链表，直到每个度数至多有一个根在根链表中
  - **Fib-Heap-Extract-Min**的代价为$O(lgn)$



斐波那契堆平摊分析：操作的分析结果

- 在势能函数$\Phi(H)=t(H)+2\cdot mark(H)$ 下
- 关键字减值(复杂操作)
  - **Fib-Heap-Decrease-Key**的实际代价为$O(c)$
  - $\Phi(H)$的变化最多为4-c (c为做级联切断操作的次数)
  - **Fib-Heap-Decrease-Key**的平摊代价为

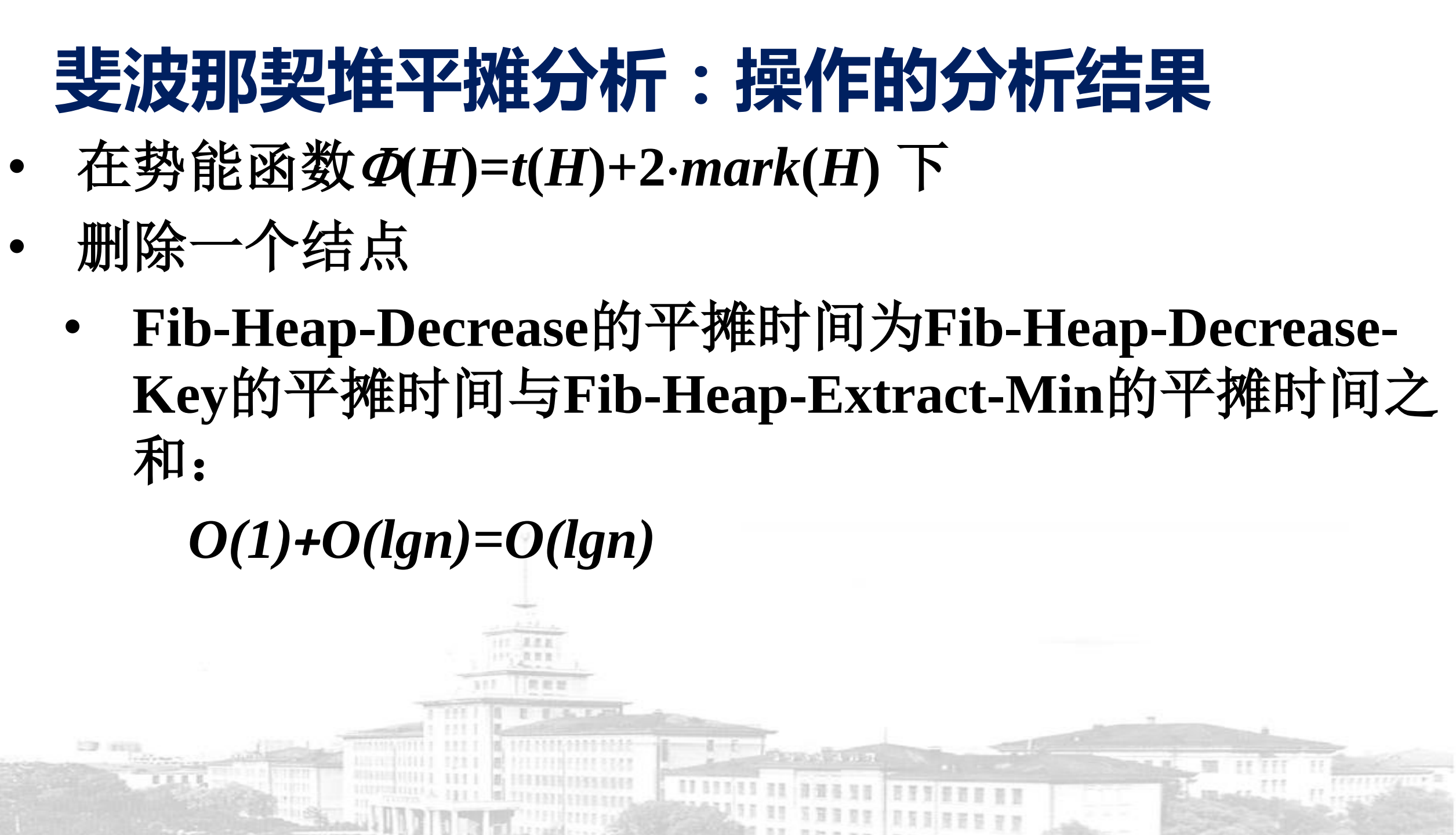
    $O(c)+4-c= O(1)$



斐波那契堆平摊分析：操作的分析结果

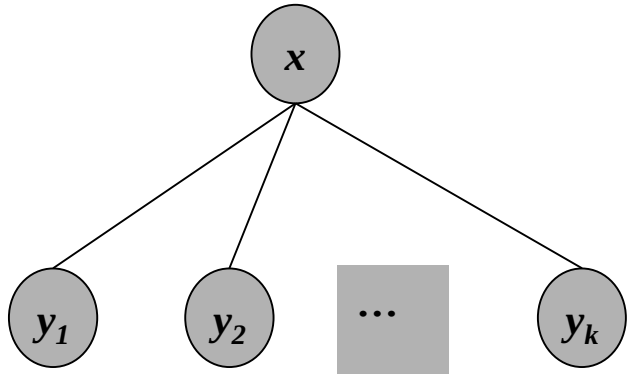
- 在势能函数$\Phi(H)=t(H)+2\cdot mark(H)$ 下
- 删除一个结点
  - **Fib-Heap-Decrease的平摊时间为Fib-Heap-Decrease-Key的平摊时间与Fib-Heap-Extract-Min的平摊时间之和：**

$$O(1)+O(lgn)=O(lgn)$$



斐波那契堆平摊分析：$rank(H)$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$rank(x)=i-1$

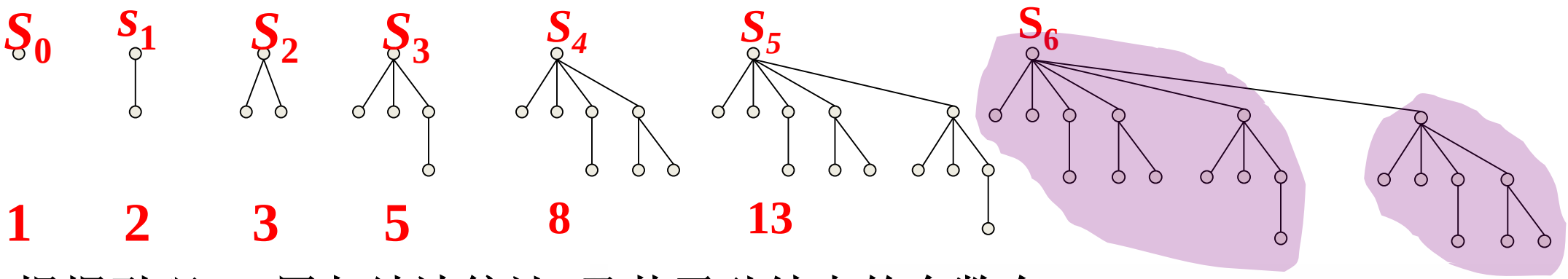
y_i要成为x的孩子，需满足$rank(y_i)=rank(x)=i-1$

y_i成为x的孩子之后，至多失去一个孩子！

斐波那契堆平摊分析：*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

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

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）

斐波那契堆平摊分析：*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，$n\geq\left(\frac{1+\sqrt{5}}{2}\right)^{k-2}$

本章内容

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 动态表性能的平摊分析

7.6 *斐波那契堆性能平摊分析

7.7 并查集性能平摊分析

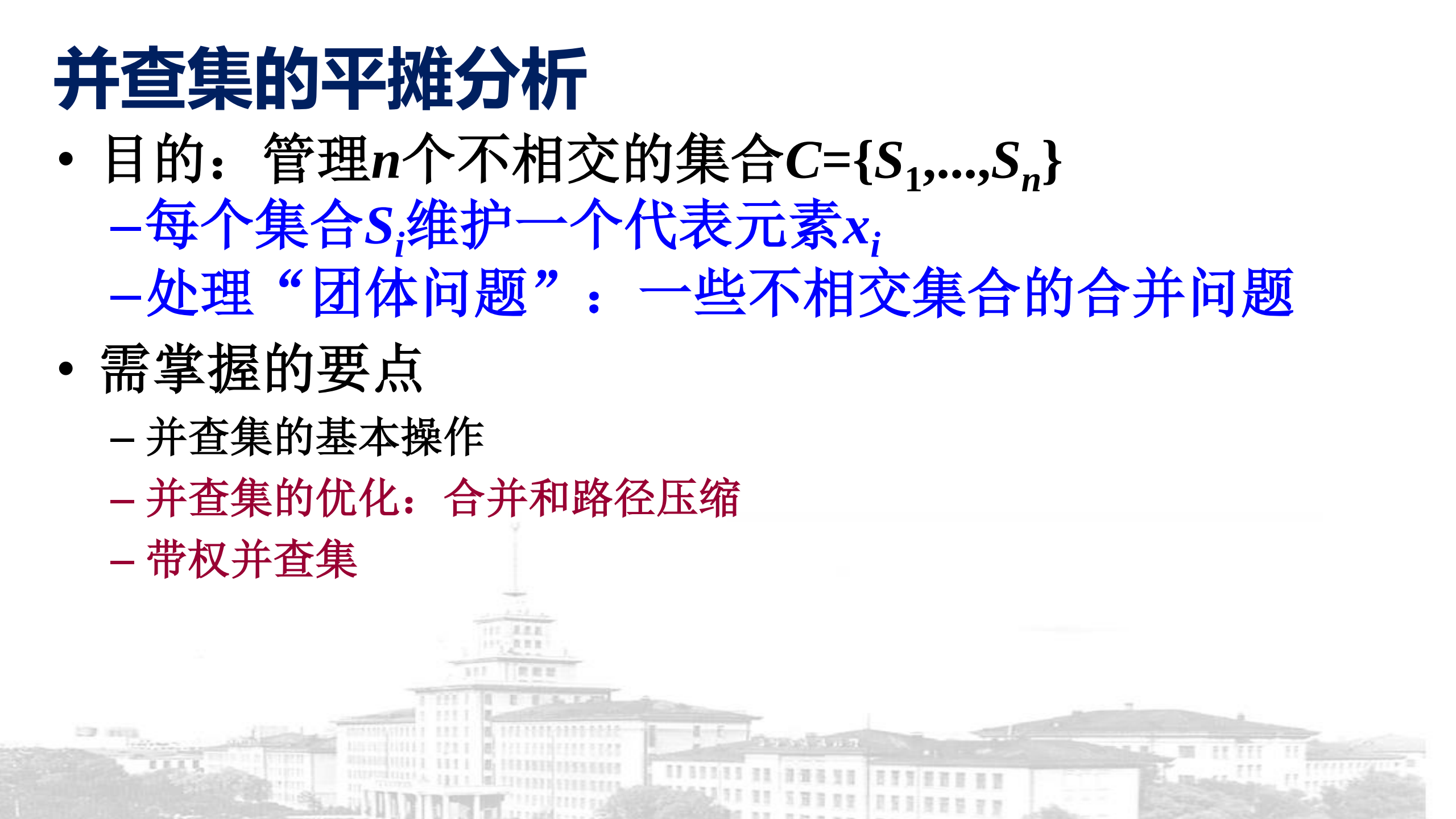
并查集的平摊分析(disjoint-set)

- 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
  - 处理“团体问题”：一些不相交集合的合并问题
- 应用场景
  - 校友关系查找
  - 亲缘的判断

被认为是一种简洁而优雅的高级数据结构！

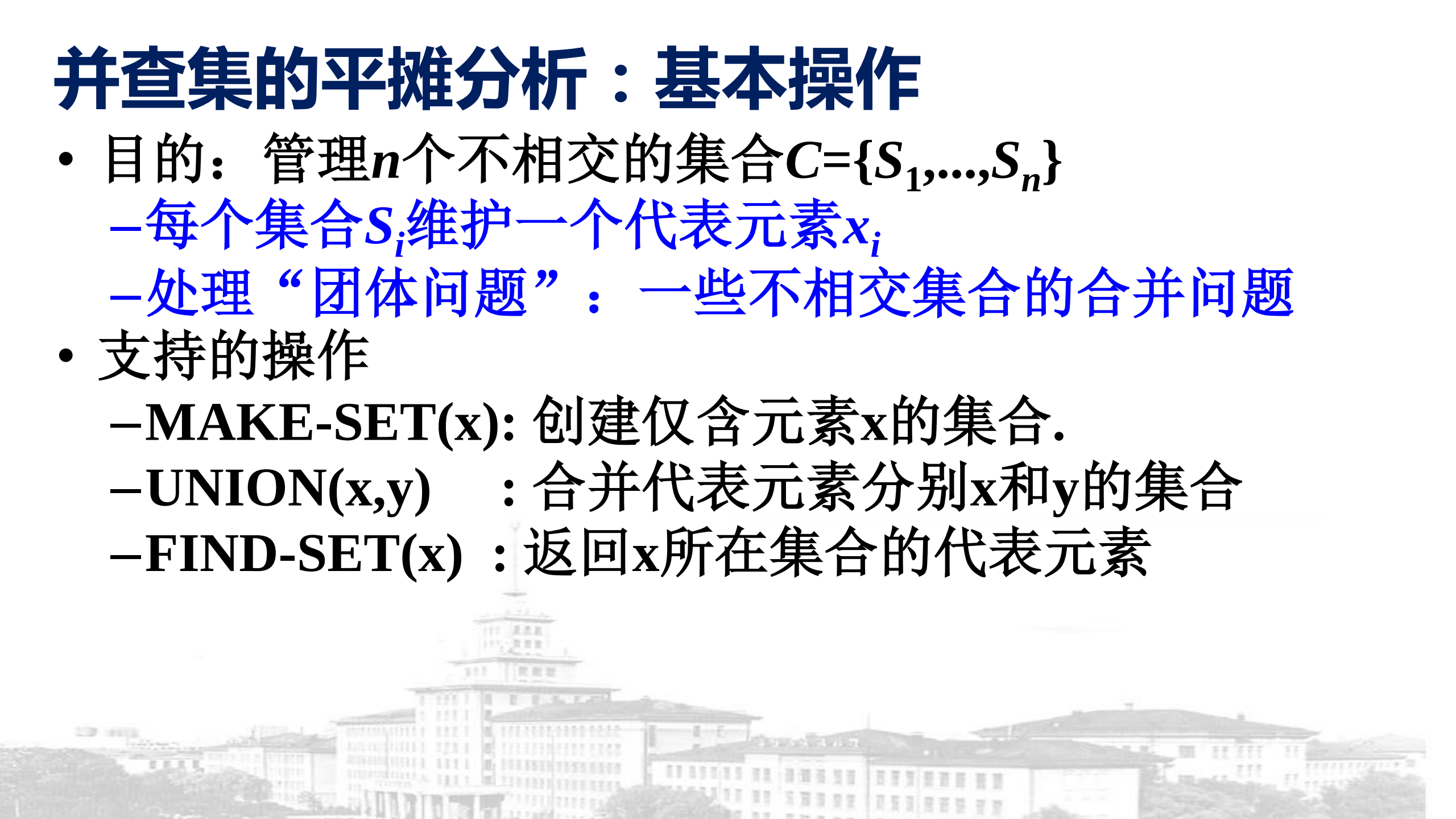
并查集的平摊分析

- 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
  - 处理“团体问题”：一些不相交集合的合并问题
- 需掌握的要点
  - 并查集的基本操作
  - 并查集的优化：合并和路径压缩
  - 带权并查集



并查集的平摊分析：基本操作

- 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
  - 处理“团体问题”：一些不相交集合的合并问题
- 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



并查集的平摊分析

- 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
  - 处理“团体问题”：一些不相交集合的合并问题
- 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)
  - $m \geq n$,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
  - 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, *Journal of Discrete Algorithms,vol23, pp 2-20, 2013*

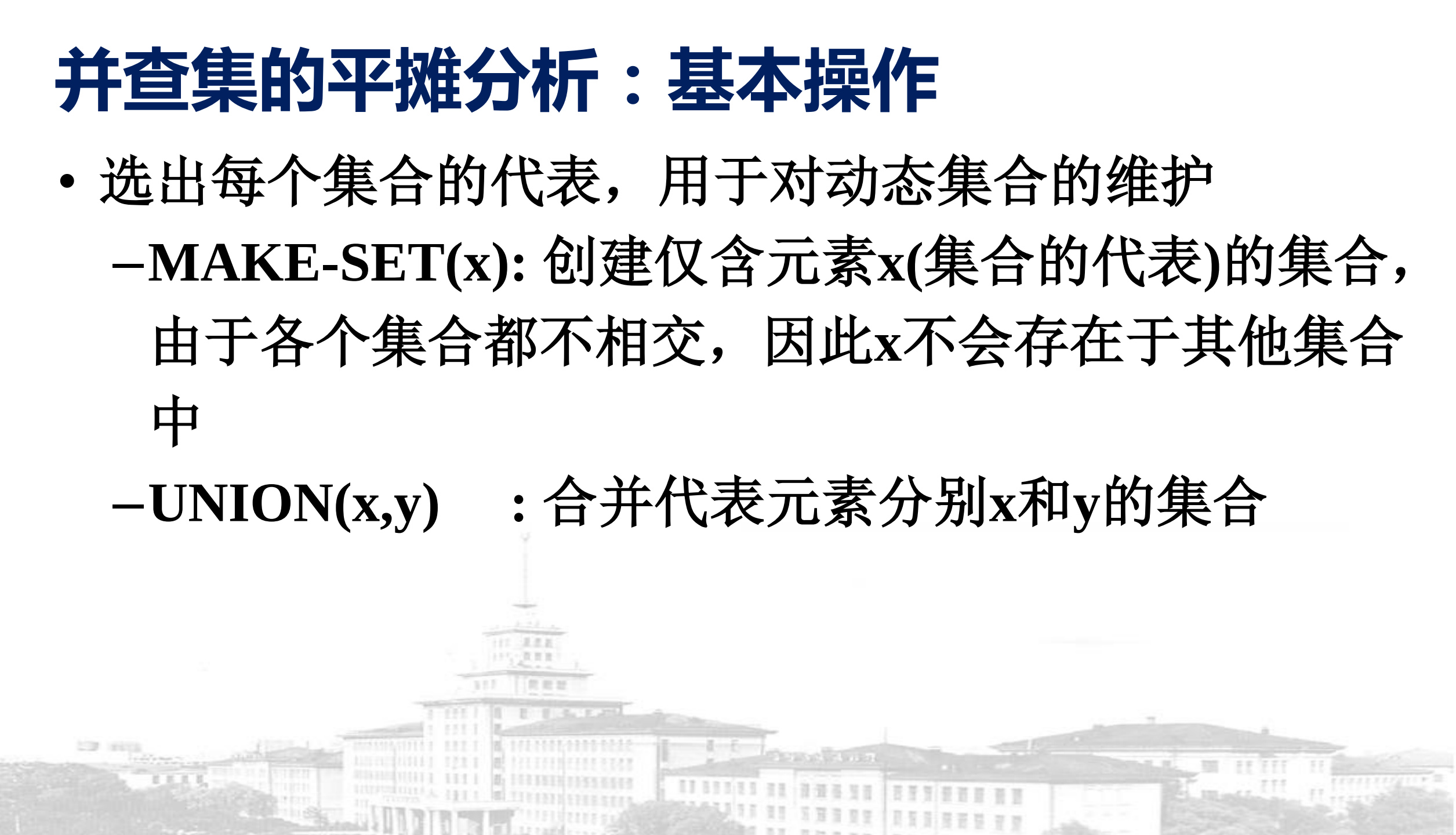
2. Disjoint set union with randomized linking, *Symp. on Discrete Algorithms, pp1005-1017, 2014*

并查集的平摊分析：基本操作

- 选出每个集合的代表，用于对动态集合的维护
  - MAKE-SET(x): 创建仅含元素x(集合的代表)的集合，由于各个集合都不相交，因此x不会存在于其他集合中

并查集的平摊分析：基本操作

- 选出每个集合的代表，用于对动态集合的维护
  - MAKE-SET(x): 创建仅含元素x(集合的代表)的集合，由于各个集合都不相交，因此x不会存在于其他集合中
  - UNION(x,y)　：合并代表元素分别x和y的集合



并查集的平摊分析：基本操作

- 选出每个集合的代表，用于对动态集合的维护
  - MAKE-SET(x)
  - UNION(x,y)
  - FIND-SET(x) ：返回一个指针，其指向x所在(唯一)集合

并查集的平摊分析：基本操作次数

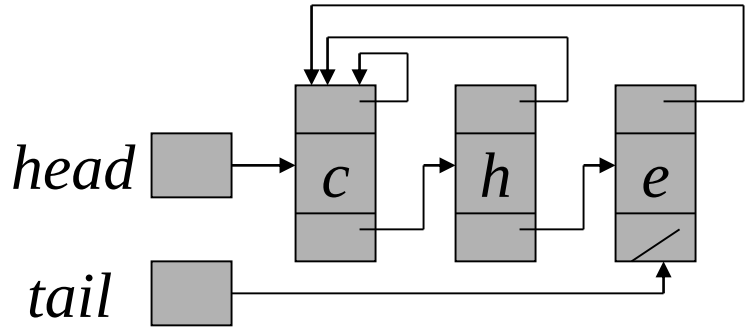
- 选出每个集合的代表，用于对动态集合的维护
  - *n*个MAKE-SET 操作 (总是最先执行的操作).
  - *m*个UNION, MAKE-SET ,FIND-SET操作(后续)
  - UNION操作至多执行 *n*-1次
  - $m \geq n$

并查集的平摊分析：并查集的直接实现为链表

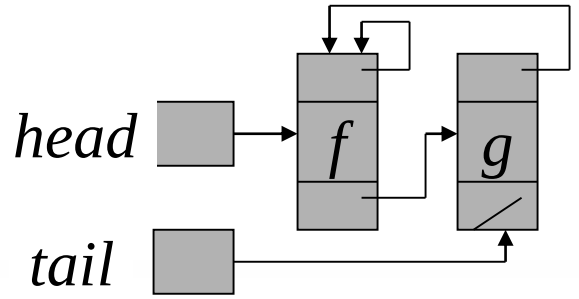
每个集合维护为一个链表

- **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($|s_x|$)

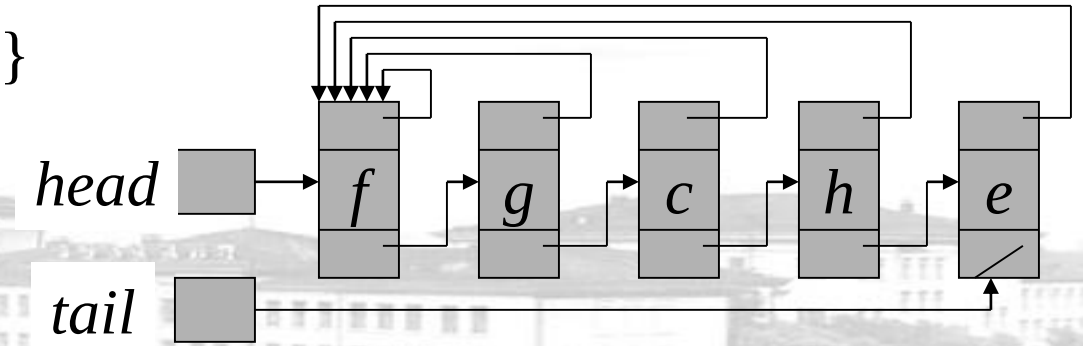
集合 {*c*,*h*,*e*}



集合{*f*, *g*}



并集{*c*,*h*,*e*,*f*,*g*}



并查集的平摊分析：并查集的直接实现为链表

- 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)

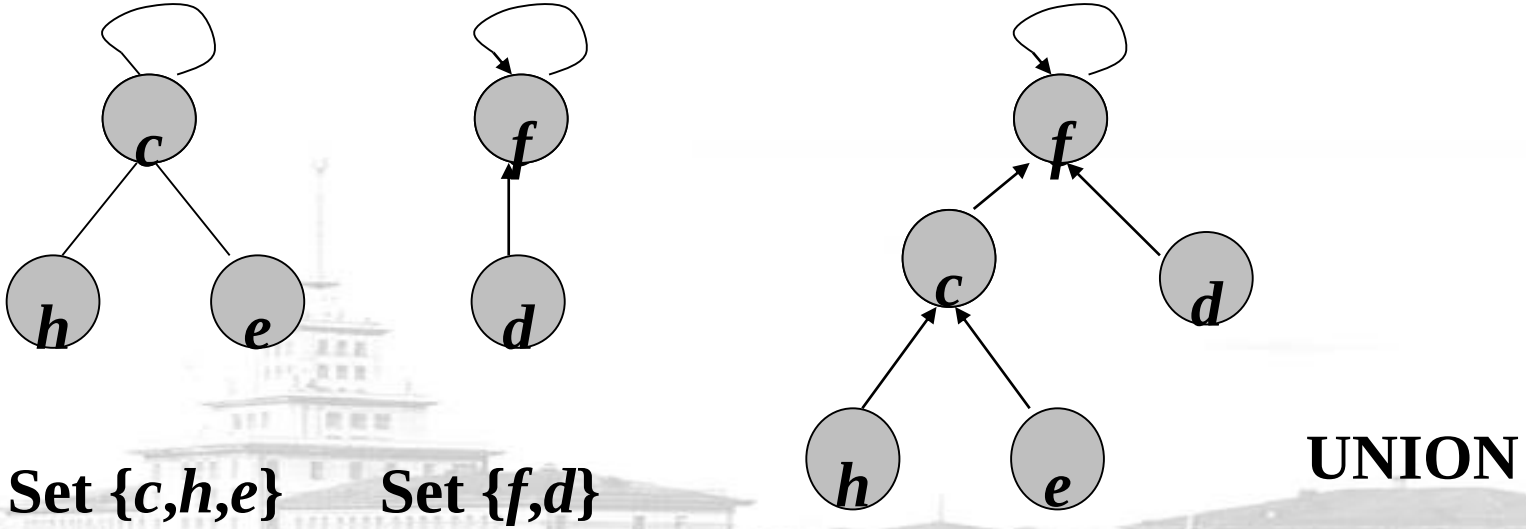
- 总共执行执行2n-1次操作的总代价为$O(n^2)$
- 从平摊效果看，每个操作的开销为$O(n)$

说明链表实现方式是很“蹩脚”，如何提高效率？

并查集的平摊分析：并查集的森林实现

并查集三种操作

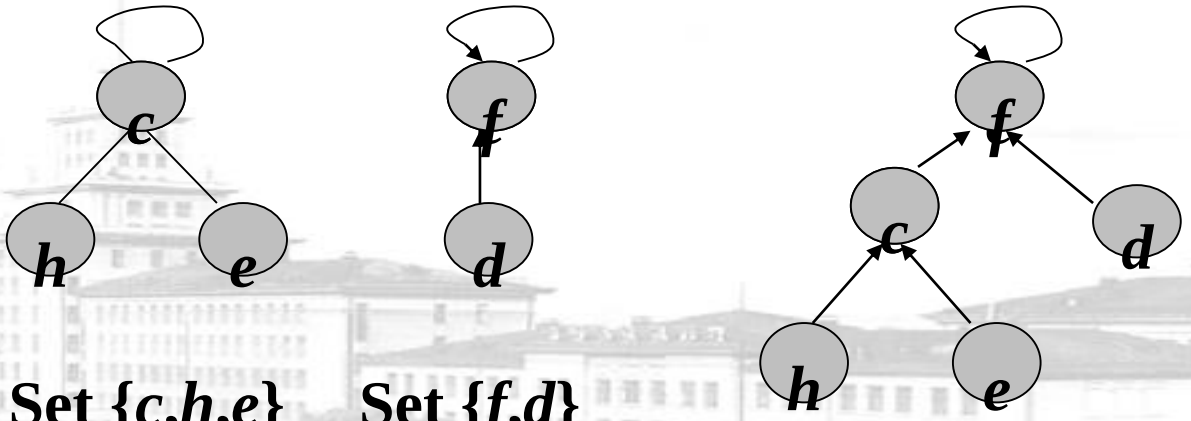
- 每个集合表示为一棵有根树
- 树根是代表元素
- 每个结点的指针指向其父结点，根结点指向自身



并查集的平摊分析：并查集的直接森林实现

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(令一棵树的根指向另一颗树的根)　$O(1)$
- 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$O(n^2)$



并查集的平摊分析：改进策略之一路径压缩

合并的优化

- 合并元素x和y时，先搜到它们的根结点，然后再合并这两个根结点，即把一个根结点的集合改成另一个根结点。这两个根结点的高度不同，如果把高度较小的集合并到较大的集合上，能减少树的高度

并查集的平摊分析：改进策略之一路径压缩

合并的优化

- 合并元素x和y时，先搜到它们的根结点，然后再合并这两个根结点

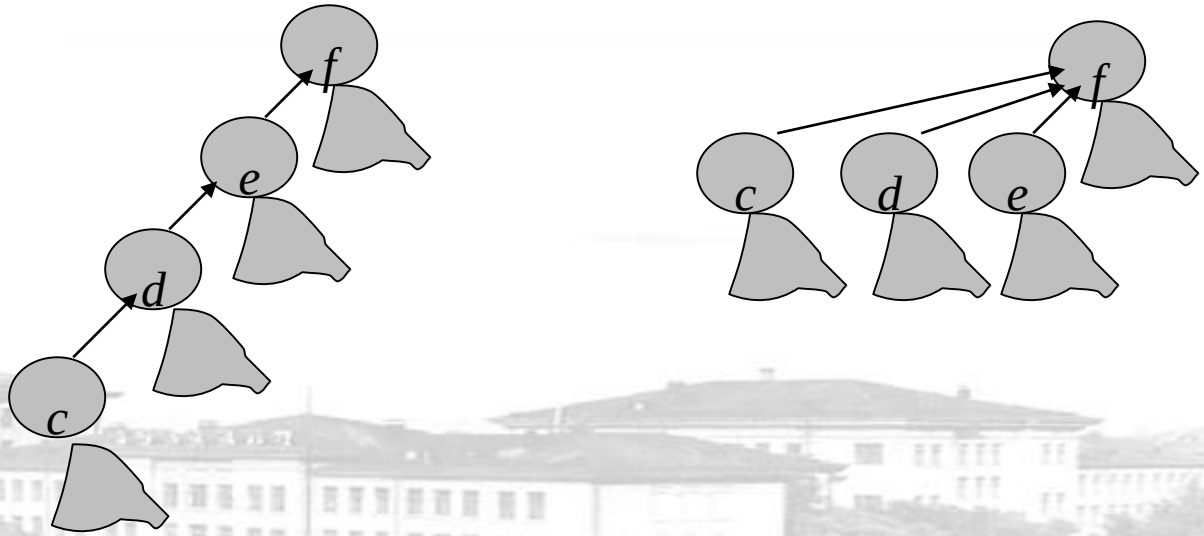
```
int height[maxn];
void init_set(){
    for(int i = 1; i <= maxn; i++){
        s[i] = i;
        height[i]=0; //树的高度
    }
}
void merge_set(int x, int y){ //优化合并操作
    x = find_set(x);
    y = find_set(y);
    if (height[x] == height[y]) {
        height[x] = height[x] + 1; //合并，树的高度加一
        s[y] = x;
    }
    else{ //把矮树并到高树上，高树的高度保持不变
        if (height[x] < height[y]) s[x] = y;
        else    s[y] = x;
    }
}
```

并查集的平摊分析：改进策略之一路径压缩

查询的优化——路径压缩

- 在find_set()中，查询元素*i*所属的集，需要搜索路径找到根结点，返回的结果是根结点。这条搜索路径可能很长
- 如果在返回的时候，顺便把*i*所属的集改成根结点，那么下次再搜的时候，就能在O(1)的时间内得到结果

树中的路径长度大幅度降低，为后续查找操作节省了时间！



并查集的平摊分析：改进策略之一路径压缩

查询的优化——路径压缩

- 整个搜索路径上的元素，在递归过程中，从元素i到根结点的所有元素，其所属的集合都被改为根结点
- 路径不仅优化了下一次查询，也优化了合并（合并中也用到了查询）

```
int find_set(int x){
    if(x != s[x])
        s[x] = find_set(s[x]); //路径压缩
    return s[x];
}
```

并查集的平摊分析：并查集操作算法

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

MAKE-SET(x)
1. rank[x]←0
2. p[x]←x

FIND(x)
1. Q←∅
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) 是UNION调用的子过程
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] ← rank[y] +1

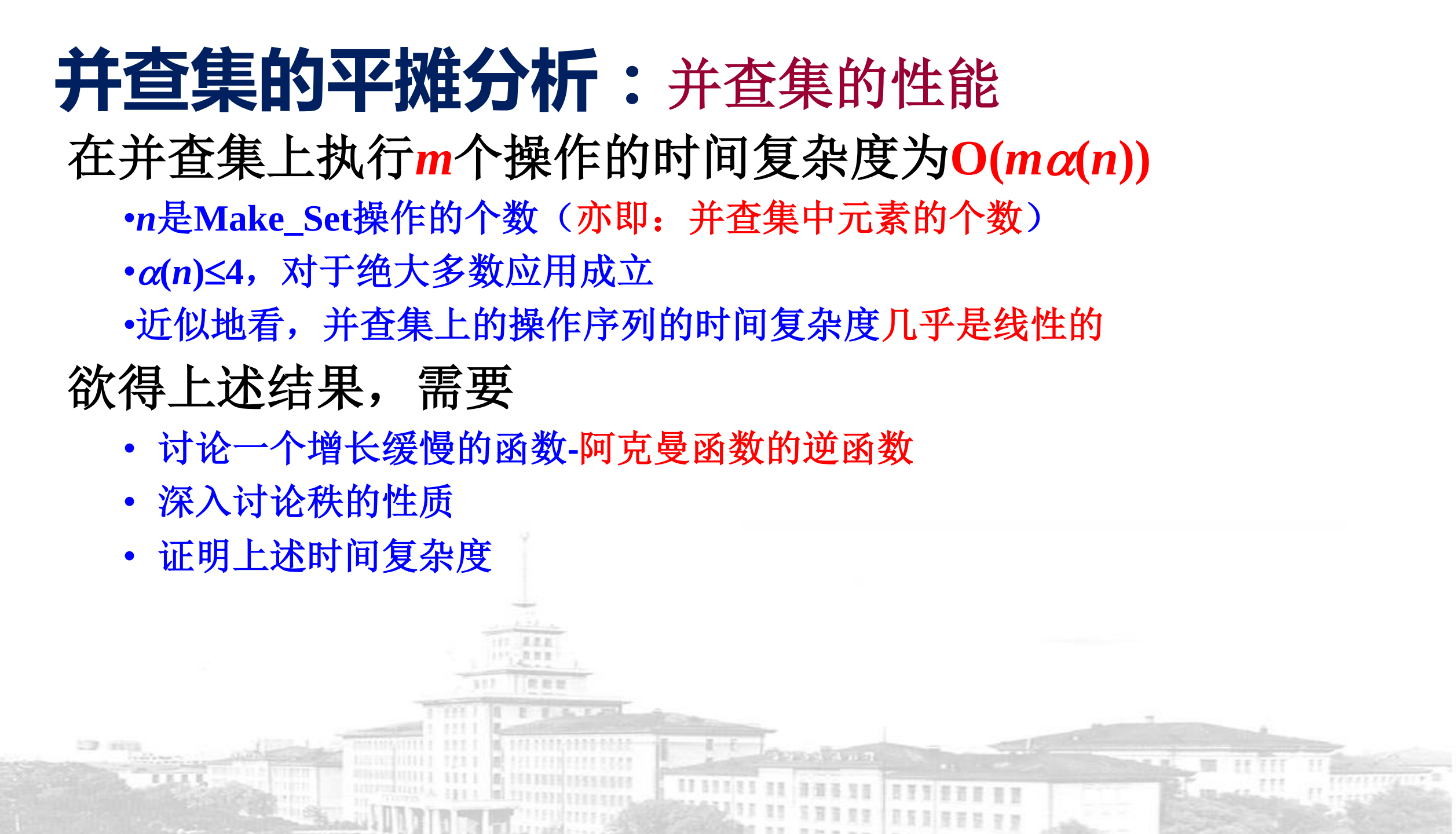
并查集的平摊分析：并查集的性能

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



并查集的平摊分析：阿克曼函数的逆函数

阿克曼函数的逆函数定义为

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

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

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？

并查集的平摊分析：并查集性能的平摊分析

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

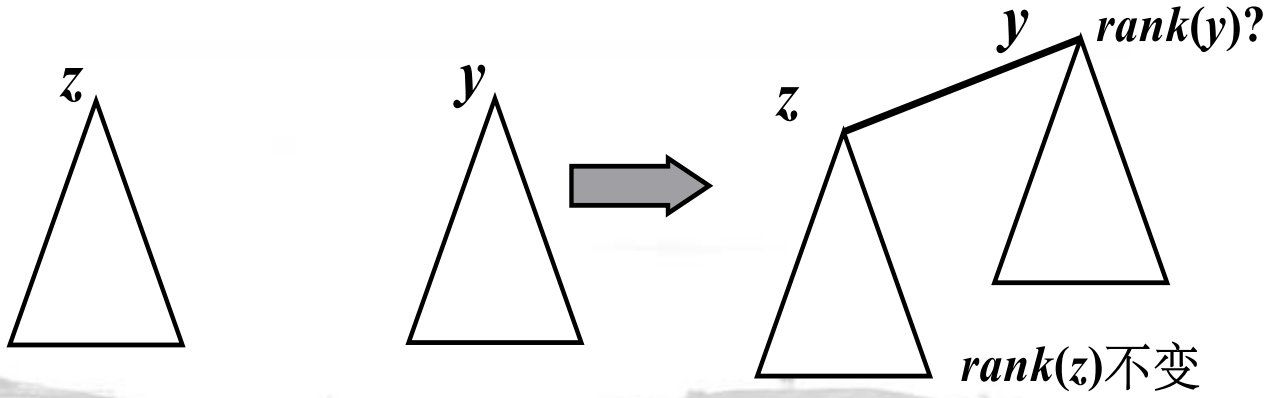
并查集的平摊分析：并查集性能的平摊分析

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？

并查集的平摊分析：并查集性能的平摊分析

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$\Theta(\alpha(n))$

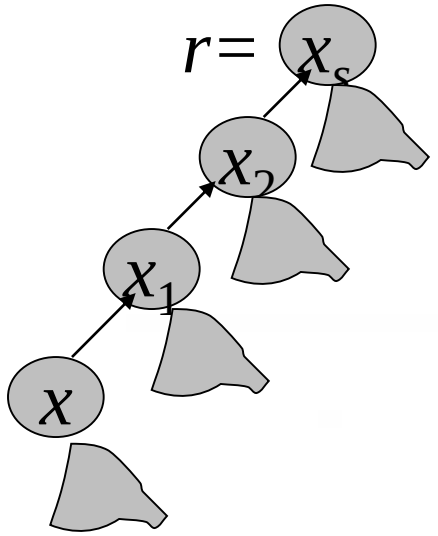
- 实际代价为$\Theta(1)$
- 势能增量为$\Theta(\alpha(n))$
  - 不妨设合并后，y是z的父结点
  - 操作仅可能改变rank(y)
  - 势能发生变化的结点只能是y，z和操作之前y的子结点w

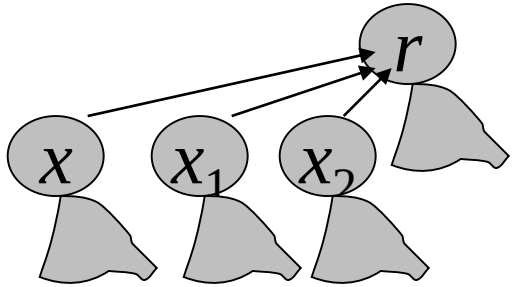
并查集的平摊分析：并查集性能的平摊分析

势能$\phi_q = \Sigma_x \phi_q(x)$

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

Find(x)操作的平摊代价为$\Theta(\alpha(n))$





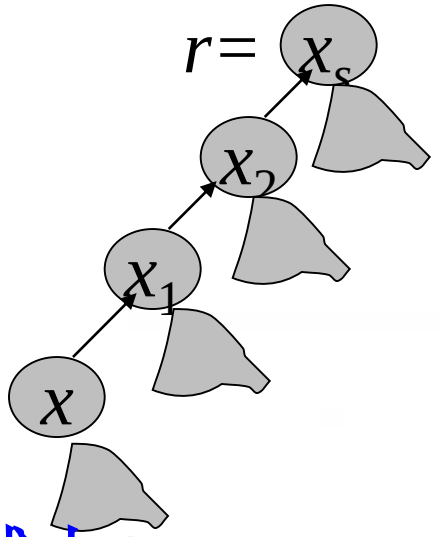
- 实际代价为$\Theta(s)$

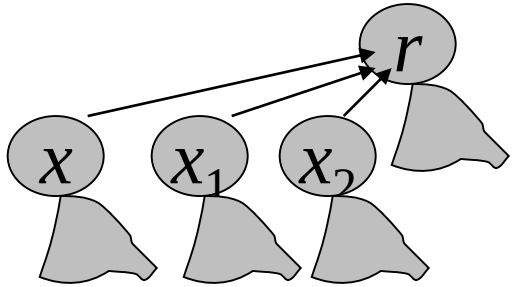
- $x=x_0,x_1,\ldots,x_{s-1}$的势能不会增加
  因为它们不是树根，故$\phi_{q+1}(x_i) \leq \phi_q(x_i)$（前面的结论）
- 树根r的势能不会发生变化
  rank(r)未发生变化

并查集的平摊分析：并查集性能的平摊分析

势能$\phi_q = \Sigma_x \phi_q(x)$

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





Find(x)操作的平摊代价为$\Theta(\alpha(n))$

- 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

并查集的平摊分析：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$O(\alpha(n))$
- Find-set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！

并查集的平摊分析：习题

- 有n个人一起吃饭，有些人互相认识，认识的人坐在一起，陌生人不坐在一起。例如A认识B，B认识C，则ABC坐在一张桌子上.

给出认识的人，设计算法求需要多少张桌子。

（提示：一张桌子是一个集合，合并朋友关系，统计集合的数量）