算法设计与分析

第七章 平摊分析

哈尔滨工业大学
王宏志
wangzh@hit.edu.cn
http://homepage.hit.edu.cn/pages/wang/

本讲内容

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

?

各个操作的代价？

对一个数据结构要执行一系列操作：

有的代价很高

有的代价一般

有的代价很低

分析中，执行一

将总的代价平摊到每个操作上

平摊代价

不涉及概率
不同于平均情况分析

平摊分析的方法

- 聚集方法
- 会计方法
- 势能方法

本讲内容

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

聚集分析法-原理

对数据结构共有n个

最坏情况下:

操作1:　t_1

操作2:　t_2

。

。

。

操作n:　t_n

操作序列中的每个操作被赋予相同的代价，不管操作的类型

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

平摊代价:

T(n)/n

平摊分析实例1-栈操作

普通栈操作

PUSH(S,x)：将对象压入栈S

POP(S)：弹出并返回S的顶端元素

时间代价:

·两个操作的运行时间都是O(1)

·我们可把每个操作的代价视为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)

平摊分析实例1-栈操作

新的栈操作

操作MULTIPOP(S,k):

执行一次While循环要调用一次POP

实现算法

输入：栈S，k

输出：返回S顶端k个对象

```
MULTIPOP(S,k)
1  While  not STACK-EMPTY(S) and  k≠0  Do
2          POP(S);
3          k←k-1
```

MULTIPOP的总代价即为min(s,k)

平摊分析实例1-栈操作

初始为空的栈上的n个

由PUSH、POP和M

操作1: t_1
操作2: t_2
。
。
。
操作n: t_n

!

Note: 分析过程没有使用任何的概率!

T(n)<=2n

于是：最坏情况下这样的一个操作序列的时间复杂度最多为O(n)

平摊分析实例2-二进计数器

1. 问题定义

实现一个由 0 开始向上计数的k位二进计数器。

输入：k位二进制变量x，初始值为0。

输出：x+1 mod 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

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

平摊分析实例2-二进计数器

- **2. 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 mod 2^k

```
INCREMENT(A)
    i←0
    while  i<length[A] and A[i]=1  Do
          A[i]←0;
          i←i+1;
    If  i<length[A]  Then  A[i]←1
```

平摊分析实例2-二进计数器

- 3.初始为零的计数器上n个INCREMENT操作的分析

每隔8次发生一次改变

共$\lfloor n/8 \rfloor$

Counter

N	A[7]	A[6]	A[5]	A[4]	[illegible]	[illegible]	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	3
3	0								4
4	0								7
5	0	0	0				0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11

总共发生的改变为：

$\Sigma\lfloor n/2^i \rfloor$ (i=0,2,…,[$\log_2 n$]

<2n

本讲内容

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

会计方

一个操作序列中有不同类型的

平摊代价可能比实际代价大，也可能比实际代价小

于是：我们在各种操作上定义平摊代价使得任意操作序列上存款总量是非负的，将操作序列上平摊代价求和即可得到这个操作序列的复杂度上界

会计方法实例 1 — 栈操作

1. 各栈操作的实际代价：

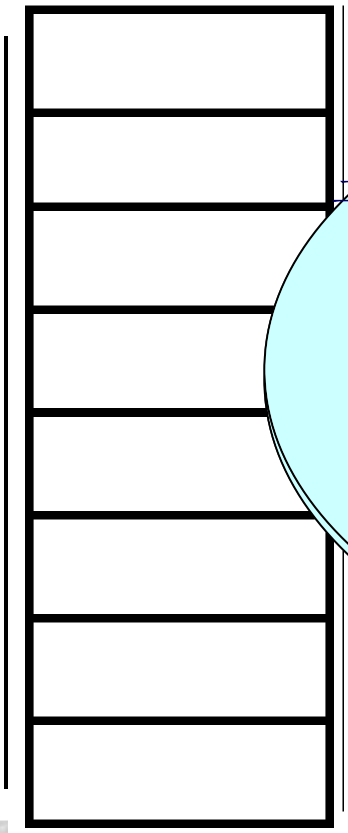
PUSH	1,
POP	1,
MULTIPOP	min(k,s)

2. 各栈操作的平摊代价：

PUSH	2,
POP	0,
MULTIPOP	0,

会计方法实例 1 — 栈操作

3. 栈操作序列代价分析



于是所有操作的平摊代价总和就是操作序列代价总和的上界

长度为n的操作序列中:

PUSH操作的个数<=n

于是:

= ?

平摊代价的总和<=2n

会计方法实例 2 –二进计数器

1. 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 mod 2^k

```
    INCREMENT(A)
    i←0
    while  i<length[A] and A[i]=1  Do
          A[i]←0;
          i←i+1;
    If  i<length[A]
    Then  A[i]←1
```

会计方法实例 2 –二进计数器

初始为零的计数器上n个INCREMENT操作的分析

任何操作序列，存款余额是计数器中1的个数，非负

因此，所有的翻转操作的平摊代价的和是这个操作序列代价的上界

本讲内容

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

势能分析—基本原理

在会计方法中，如果操作的平摊代价比实际代价大，我们将余额与具体的数据对象关联

如果我们将这些余额都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

势能分析—基本原理

势能的定义：

对一个初始数据结构D_0执行n个操作

对操作i:

实际代价c_i将数据结构D_{i-1}变为D_i

势函数ϕ将每个数据结构D_i映射为一个实数$\phi(D_i)$

平摊代价c'_i定义为：$c'_i = c_i + \phi(D_i) - \phi(D_{i-1})$

势能分析—基本原理

- n个操作的总的平摊代价为：

平摊代价依赖于所选择的势函数φ。不同的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

于是势函数φ满足φ(D_n)≥φ(D_0)，则总的平摊代价就是总的实际代价的一个上界

势能方法实例1 — 栈操作

$\phi(D)$=栈D中对象的个数

初始栈D_0为，$\phi(D_0)=0$

因为栈中的对象数始终非负，第i个超作之后的栈D_i满足$\phi(D_i) \geq 0 = \phi(D_0)$

于是：　以ϕ表示的n个操作的平摊代价的总和就表示了实际代价的一个上界

势能方法实例1 — 栈操作

作用于包含s个对象的栈上的栈操作的平摊代价

平摊分析：

每个栈操作的平摊代价都是O(1)

n个操作序列的总平摊代价就是O(n)

因为$\phi(D_i) \geq \phi(D_0)$， n个操作的总平摊代价即为总的实际代价的一个上界，即n个操作的最坏情况代价为O(n)

势能方法实例 2 –二进计数器

·$\phi(D)$=计数器D中1的个数

·计数器初始状态D_0中1的个数为0，$\phi(D_0)=0$

·因为栈中的对象数始终非负，第i个操作之后的栈D_i满 足$\phi(D_i)\geq 0=\phi(D_0)$

- ·于是：n个操作的平摊代价的总和就表示了实际代价的一个上界

势能方法实例 2 –二进计数器

第i次INCREMENT操作的平摊代价

计数器初始状态为0时的平摊分析 ：

每个栈操作的平摊代价都是O(1)

n个操作序列的总平摊代价就是O(n)

因为$\phi(D_i) \geq \phi(D_0)$， n个操作的总平摊代价即为总的实际代价的一个上界，即n个操作的最坏情况代价为O(n)

势能方法实例 2 -二进计数器

- 开始时不为零的计数器上n个INCREMENT操作的分析

设：开始时有b_0个1　　$0 \le b_0$

在n次INCREMENT操作之后有b_n个1

正是势能法，给我们这样的分析带来了方便！

因为$\phi(D_0)=b_0$，

作的总的实际代价

如果我们执行了至少$n=\Omega(k)$次INCREMENT操作，则无论计数器中包含什么样的初始值，总的实际代价都是O(n)

$i=1$　　$i=1$

本讲内容

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

动态表

- 动态表的概念
- 本节的目的：
  - 研究表的动态扩张和收缩的问题
  - 利用平摊分析证明插入和删除操作的平摊代价为O(1)，即使当它们引起了表的扩张和收缩时具有较大的实际代价
  - 研究如何保证一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-基本术语

- 动态表支持的操作
  ·TABLE-INSERT：将

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

设T表示一个表:

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

算法：TABLE—INSERT(T, x)

```
If  size[T]=0
Then  allocate table[T] with 1 slot;
    size[T]←1;
If  num[T]=size[T] Then
    allocate new table with 2×size[T] slots;
    insert all items in table[T] into new-table;
    free table[T];
    table[T]←new-table;
    size[T]←2×size[T];
Insert x into table[T];
num[T]←num[T]+1
```

开销由size[T]决定

动态表-表的扩张

初始为空的表上n次TABLE-INSERT操作的代价分析-粗略分析

算法：TABLE—INSERT(T, x)

```
        insert all items in table[T] into new-table;
        free table[T];
        table[T]←new-table;
        size[T]←2×size[T];
    Insert x into table[T];
    num[T]←num[T]+1
```

这个界不精确，因为执行n次TABLE—INSERT操作的过程中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

第i次操作的代价C_i:

如果i=1 $c_i=1$

如果表有空间 $c_i=1$

如果表是满的 $c_i=i$

如果以共有n次操作：

…情况下

每次进行…次操作

总的代价上界为n^2

动态表-表的扩张

初始为空的表上n次TABLE-INSERT操作的代价分析-**聚集分析**

```
算法：TABLE—INSERT(T, x)
    If  size[T]=0
    Then  allocate table[T] with 1 slot;
          size[T]←1;
    If  num[T]=size[T] Then
          allocate new table with 2×size[T] slots;
          insert all items in table[T] into new-table;
          free table[T];
          table[T]←new-table;
          size[T]←2×size[T];
  Insert x into table[T];
  num[T]←num[T]+1
```

第i次操作的代价C_i:

如果$i=2^m$　　$c_i=i$

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

动态表-表的扩张

初始为空的表上n次TABLE-INSERT操作的代价分析-会计法分析

算法：TABLE—INSERT(T, x)

1

h 1 slot;

4 If num[T]=size[T] Th

5 allocate new table with 2×size[T

6 insert all items in table[T] into

7 free table[T];

8 table[T]←new-table;

9 size[T]←2×size[T];

10 Insert x into table[T];

11 num[T]←num[T]+1

任何时候，存款总和非负

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

初始为空的表上n次TABLE-INSERT操作的平摊代价总和为3n

动态表–表的扩张

初始为空的表上n次TABLE-INSERT操作的代价分析-势能法分析

```
算法：TABLE—INSERT(T, x)
    If  size[T]=0
    Then  allocate table[T] with 1 slo
        size[T]←1;
    If  num[T]=size[T
        allocate new
        insert all item
        free table[T];
        table[T]←new-table;
        size[T]←2×size[T];
Insert x into table[T];
num[T]←num[T]+1
```

由

n次TA

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

第i次操作的平摊代价：

如果发生扩张： $c'_i = 3$

否则 $c'_i = 3$

初始为空的表上n次TABLE-INSERT操作的平摊代价总和为3n

动态表–表的扩张和收缩

表的扩张

表的收缩

理想情况下，我们希望表满足：

表具有一定的丰满度

表的操作序列的复杂度是线性的

动态表–表的扩张和收缩

表的收缩策略

上面的收缩策略可以改善，允许装载因子低于1/2

方法：当向满的表中插入一项时，还是将表扩大一倍

但当删除一项而引起表不足1/4满时，我们就将表缩小为原来的一半

这样，扩张和收缩过程都使得表的装载因子变为1/2
但是，表的装载因子的下界是1/4

动态表–表的扩张和收缩

- 由n个TABLE—INSERT和TABLE-DELETE操作构成的序列的代价的分析-势能法

第i次操作的平摊代价：$c'_i = c_i + \phi(T_i) - \phi(T_{i-1})$

第i次操作是TABLE—INSERT：未扩张 $c'_i \le 3$

第i次操作是TA $c'_i \le 3$

所以作用于一动态表上的n个操作的实际时间为O(n)

第i次操作是TABLE—DELETE：未收缩 $c'_i \le 3$

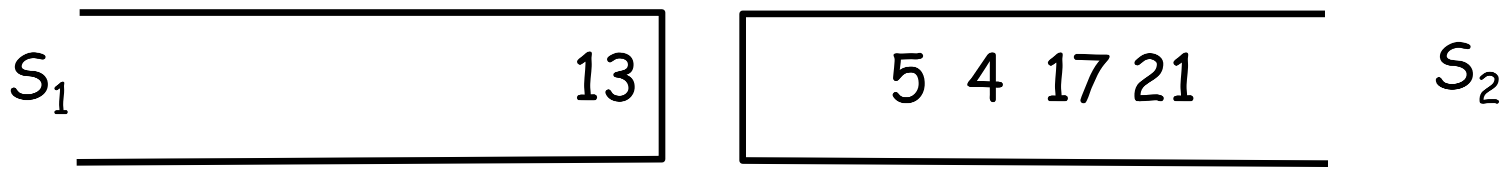
第i次操作是TABLE—DELETE：收缩 $c'_i \le 3$

补充内容一：队列的栈实现

队列

- Inject(x,Q) : 将最后一个元素X插入Q
- Pop(Q) : 删除Q中的第一个元素
- Empty?(Q): 如果Q空，返回yes
- Front(Q): 返回Q中的第一个元素
- Size(Q):
- Make-queue():

用栈实现



size=5

inject(x,Q): push(x,S_2); size ← size + 1

inject(2,Q)

用栈实现

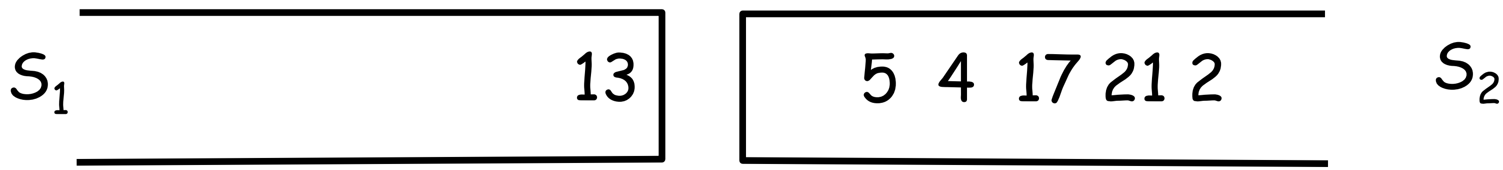
S_1 | 13 | | 5 4 17 21 2 | S_2

size=5

inject(x,Q): push(x,S_2); size ← size + 1

inject(2,Q)

用栈实现



size=6

inject(x,Q): push(x,S_2); size ← size + 1

inject(2,Q)

Pop

S_1 | 13 | | 5 4 17 21 2 | S_2

size=6

pop(Q): if empty?(Q) error
 if empty?(S_1) then move(S_2, S_1)
 pop(S_1); size ← size -1

pop(Q)

Pop

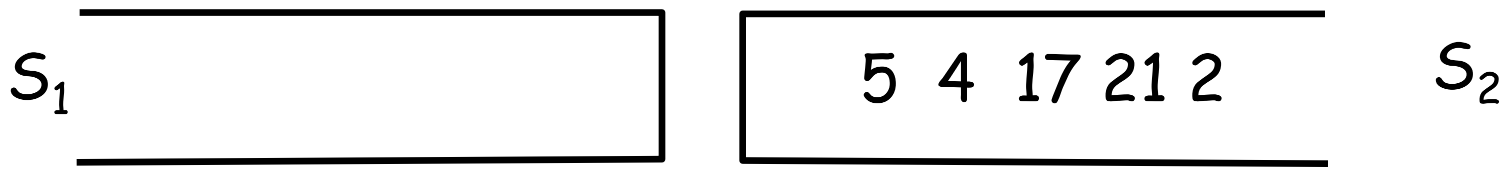
S_1 | | 5 4 17 21 2 | S_2

size=6

pop(Q): if empty?(Q) error
if empty?(S_1) then move(S_2, S_1)
pop(S_1); size $\leftarrow$ size -1

pop(Q)

Pop

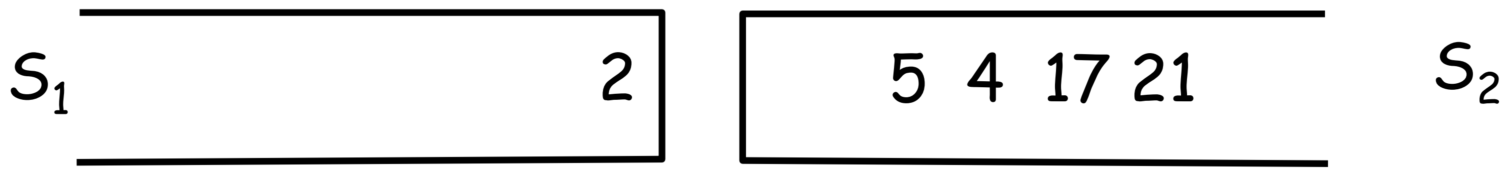


size=5

pop(Q): if empty?(Q) error
if empty?(S_1) then move(S_2, S_1)
pop(S_1); size ← size -1

pop(Q) pop(Q)

Pop

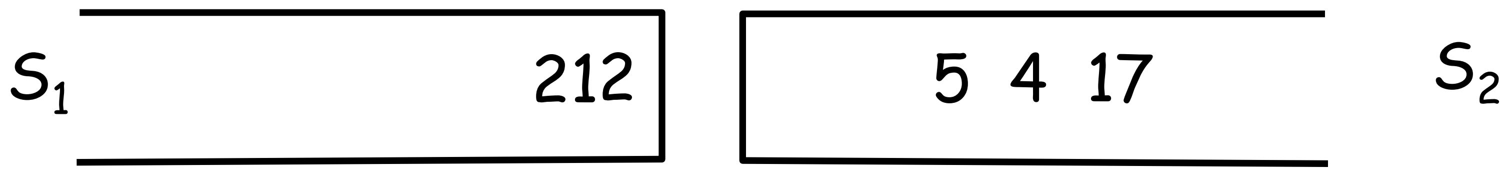


size=5

pop(Q): if empty?(Q) error
if empty?(S_1) then move(S_2, S_1)
pop(S_1); size ← size -1

pop(Q) pop(Q)

Pop

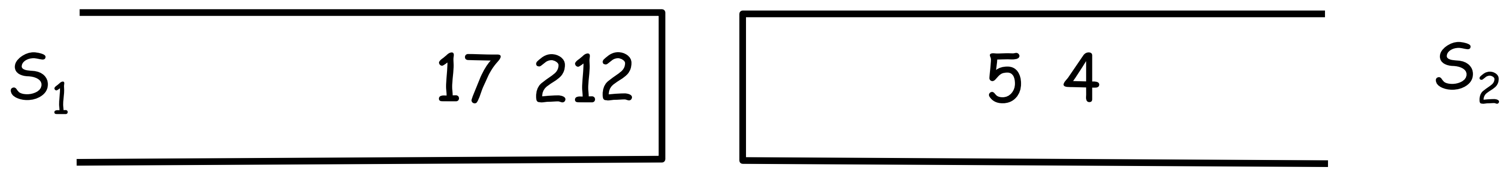


size=5

pop(Q): if empty?(Q) error
if empty?(S_1) then move(S_2, S_1)
pop(S_1); size $\leftarrow$ size -1

pop(Q) pop(Q)

Pop

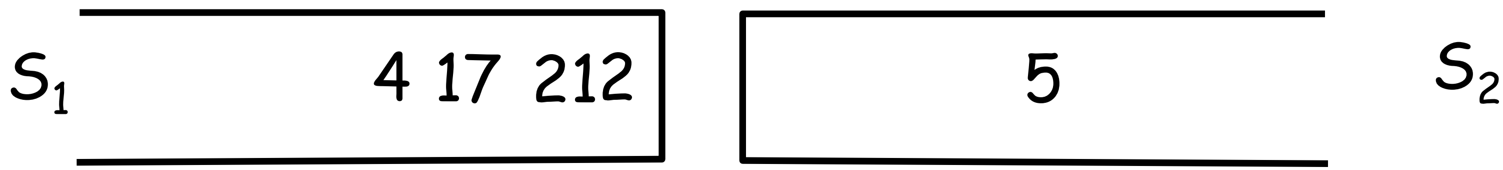


size=5

pop(Q): if empty?(Q) error
if empty?(S_1) then move(S_2, S_1)
pop(S_1); size ← size -1

pop(Q) pop(Q)

Pop

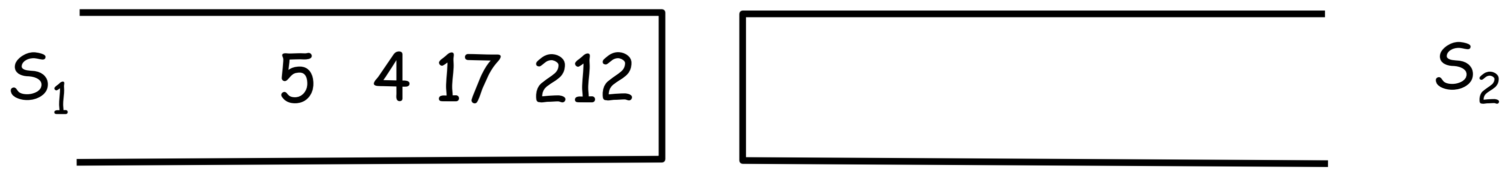


size=5

pop(Q): if empty?(Q) error
if empty?(S_1) then move(S_2, S_1)
pop(S_1); size ← size -1

pop(Q) pop(Q)

Pop

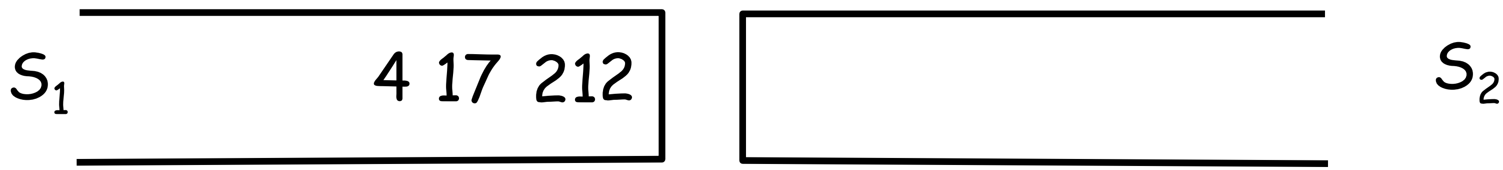


size=5

pop(Q): if empty?(Q) error
if empty?(S_1) then move(S_2, S_1)
pop(S_1); size ← size -1

pop(Q) pop(Q)

Pop



size=4

pop(Q): if empty?(Q) error
  if empty?(S_1) then move(S_2, S_1)
    pop(S_1); size ← size -1

pop(Q) pop(Q)

move(S_2, S_1)

while not empty?(S_2) do

x $\leftarrow$ pop(S_2) push(x,S_1)

分析

- 每个操作在最坏的情况下花费时间是O(n)

平摊分析

- 在最坏的情况下执行m个操作需要花费多少时间？
- O(nm)
- 是否正确？

观察

- 一个昂贵的操作不可能太频繁发生！

定理:如果我们从一个空队列开始，执行m个操作，花费时间为O(m)

证明

考虑

$$\Phi(D) = |S_2|$$

Amortized(op) = actual(op) + $\Delta\Phi$

如果移动没有发生，时间复杂度是$O(1)$

我们移动了S_2:

实际时间是$|S_2| + O(1)$

$\Delta\Phi = -|S_2|$

所以平摊代价是 $O(1)$

双端队列 (deque)

- Push(x,D)：在D中插入X作为D中的第一个元素
- Pop(D)：删除D中的第一个元素
- Inject(x,D): 在D中插入X作为D中的最后一个元素
- Eject(D): 删除D中最后一个元素
- Size(D)
- Empty?(D)
- Make-deque()

用栈实现

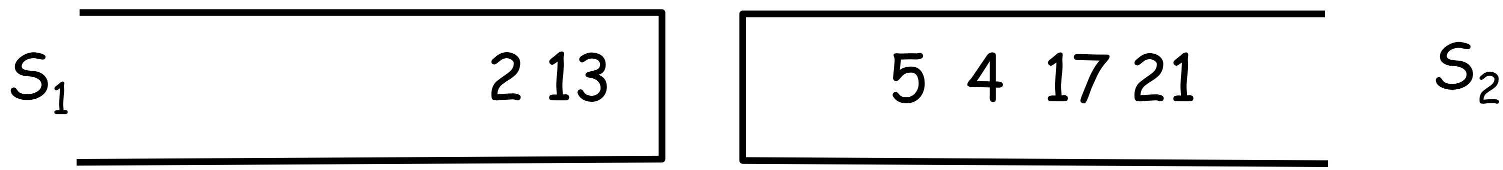
S_1 13 | 5 4 17 21 S_2

size=5

push(x,D): push(x,S_1)

push(2,D)

用栈实现



size=6

push(x,D): push(x,S_1)

push(2,D)

Pop

S_1 2 13

5 4 17 21 S_2

size=6

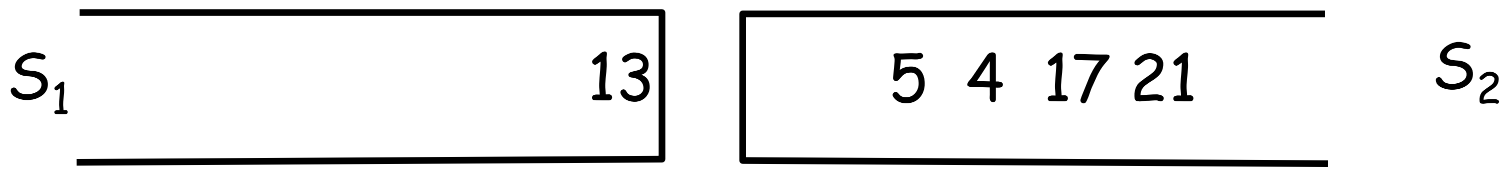
pop(D): if empty?(D) error

if empty?(S_1) then split(S_2, S_1)

pop(S_1)

pop(D)

Pop



size=5

pop(D): if empty?(D) error
if empty?(S_1) then split(S_2, S_1)
pop(S_1)

pop(D) pop(D)

Pop

S_1 | | $5\ 4\ 17\ 21$ | S_2

size=4

pop(D): if empty?(D) error
if empty?(S_1) then split(S_2, S_1)
pop(S_1)

pop(D) pop(D)

Pop

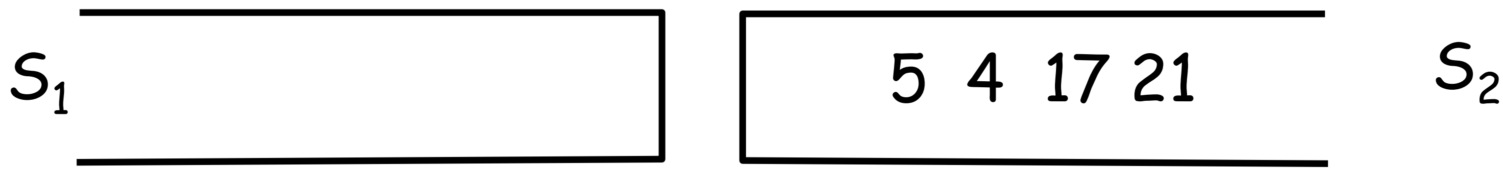
S_1 | | 5 4 17 21 | S_2

size=4

pop(D): if empty?(D) error
if empty?(S_1) then split(S_2, S_1)
pop(S_1)

pop(D)

Pop

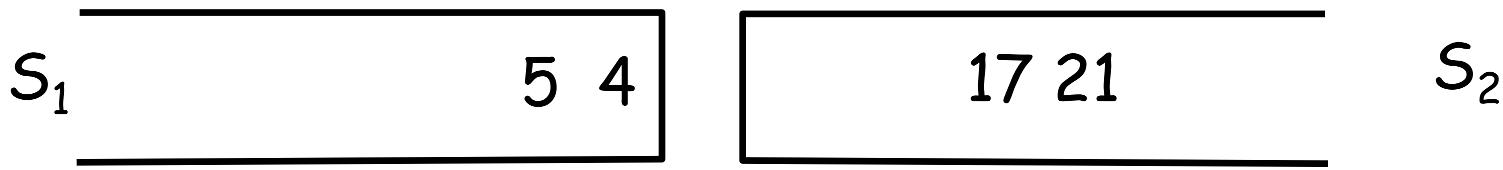


size=4

pop(D): if empty?(D) error
if empty?(S_1) then split(S_2, S_1)
pop(S_1)

pop(D)

Pop

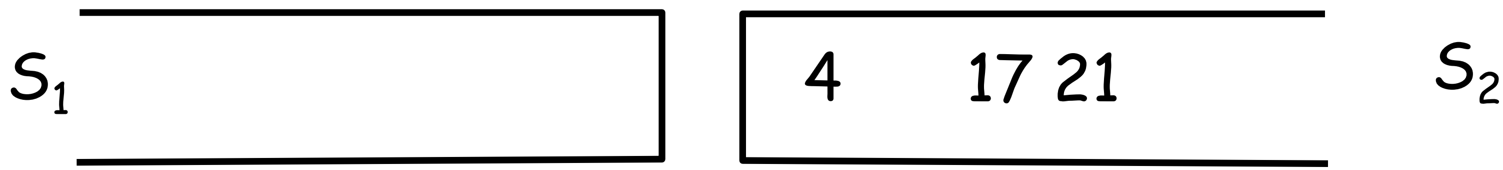


size=4

pop(D): if empty?(D) error
if empty?(S_1) then split(S_2, S_1)
pop(S_1)

pop(D)

Pop

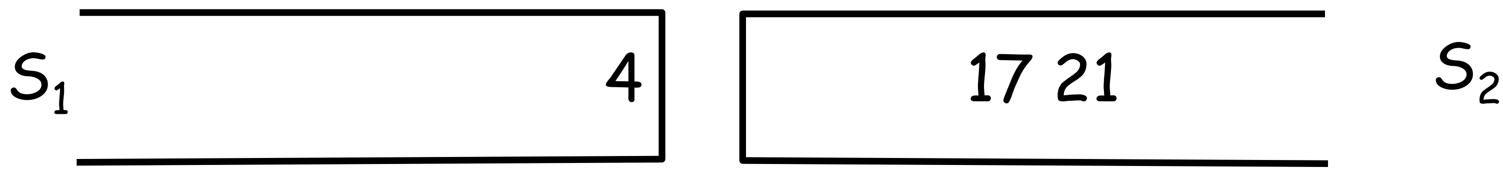


size=4

pop(D): if empty?(D) error
if empty?(S_1) then split(S_2, S_1)
pop(S_1)

pop(D)

Pop



size=3

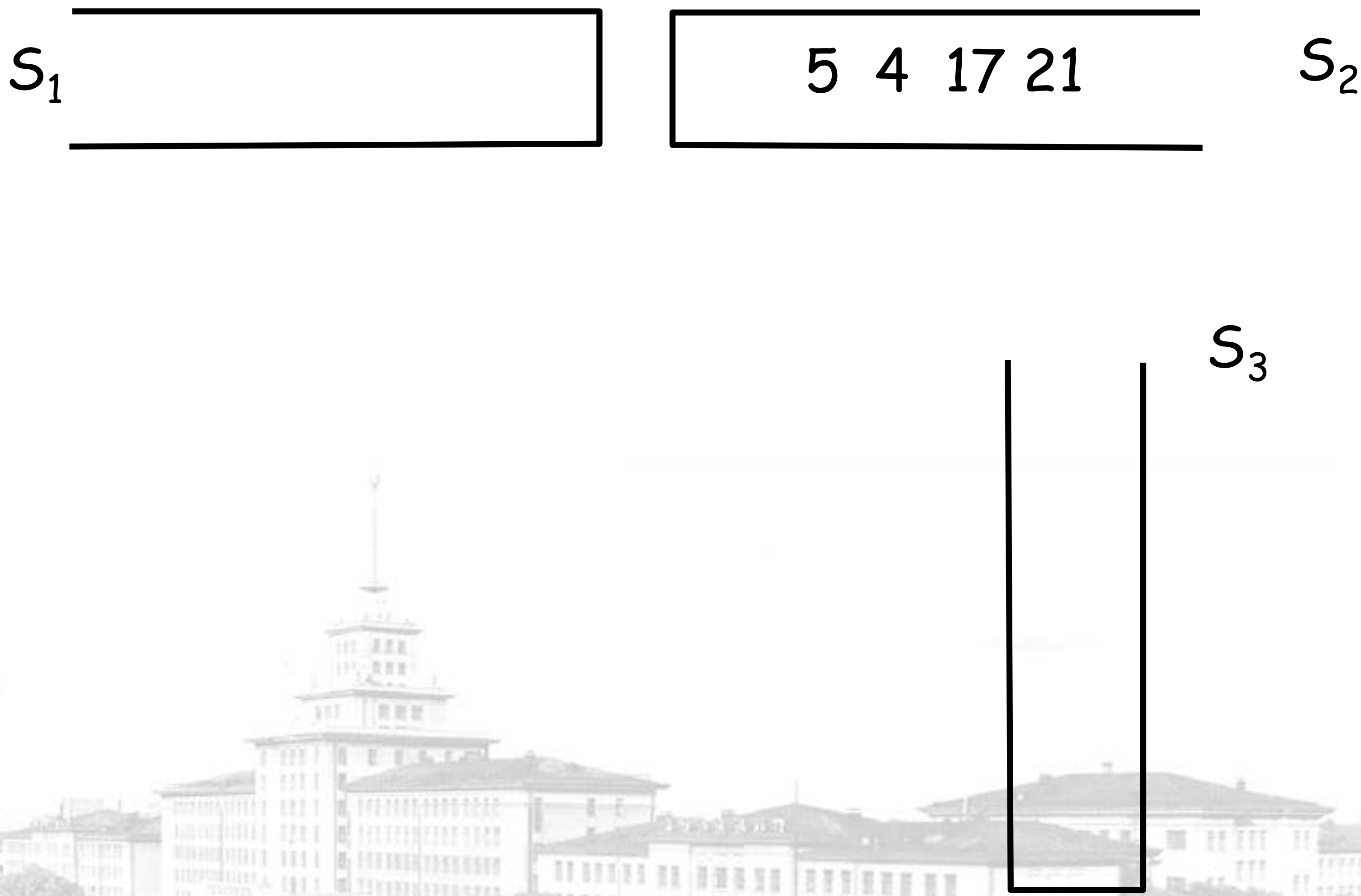
pop(D): if empty?(D) error

if empty?(S_1) then split(S_2, S_1)

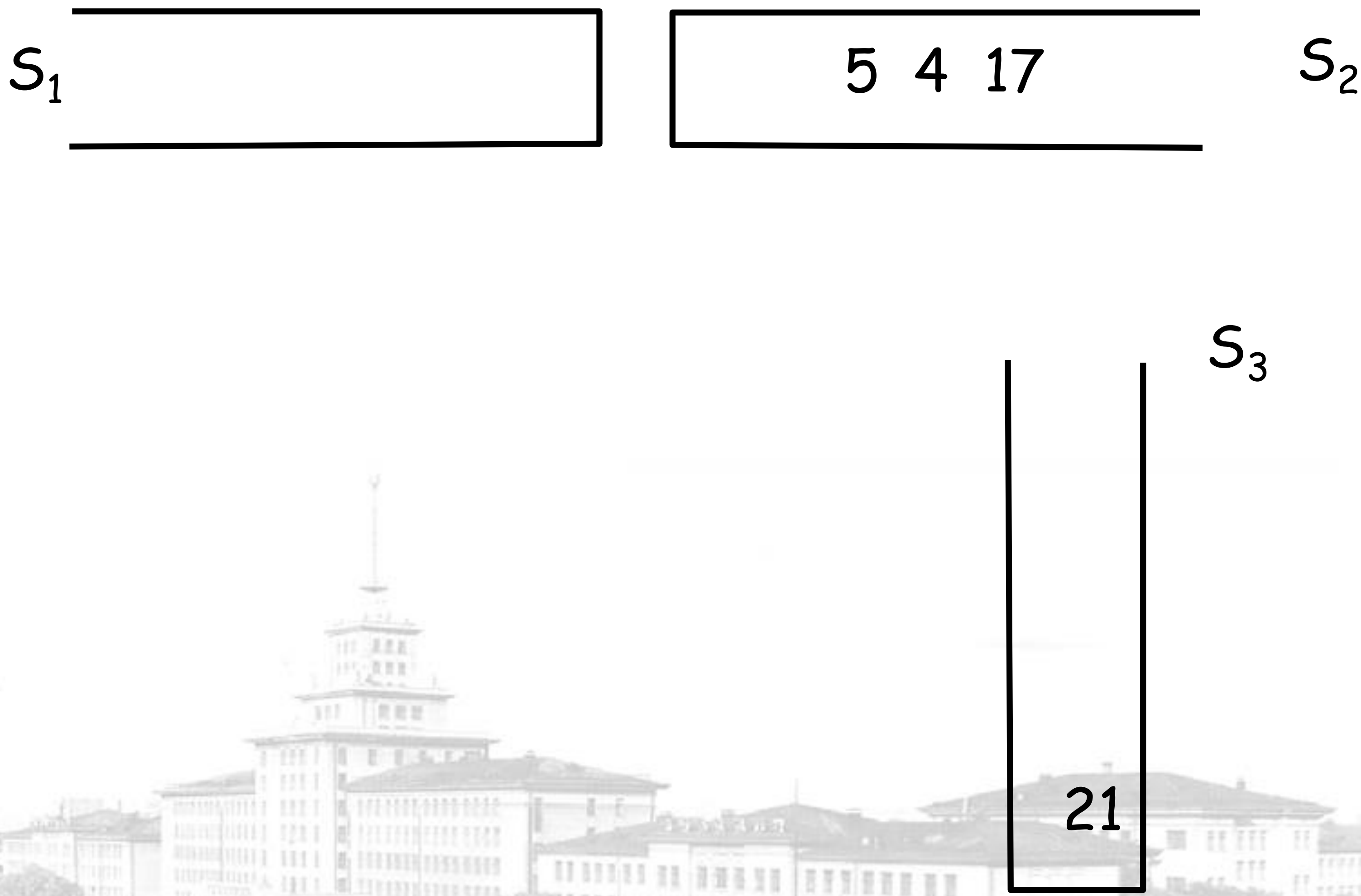
pop(S_1)

pop(D)

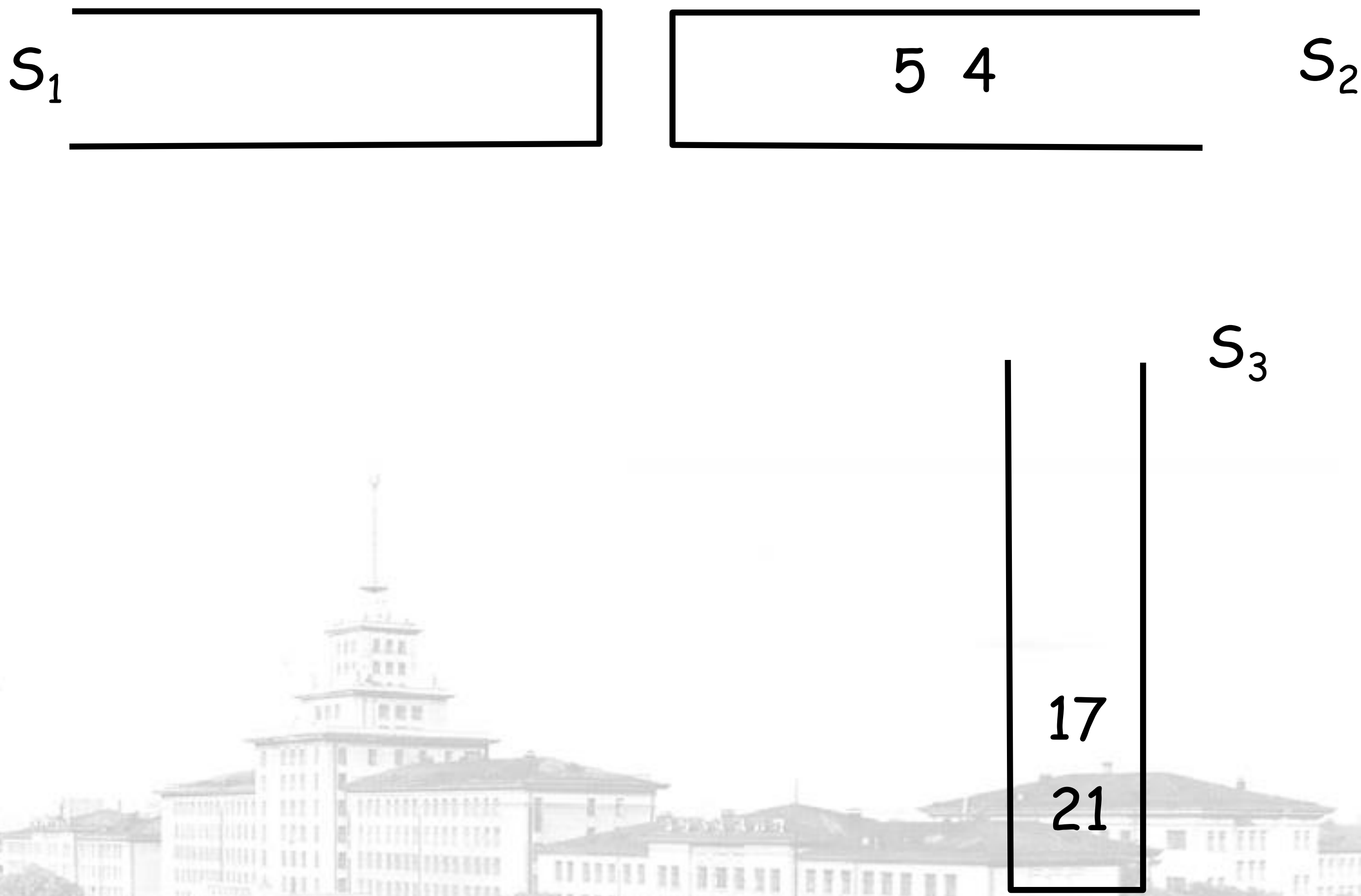
Split



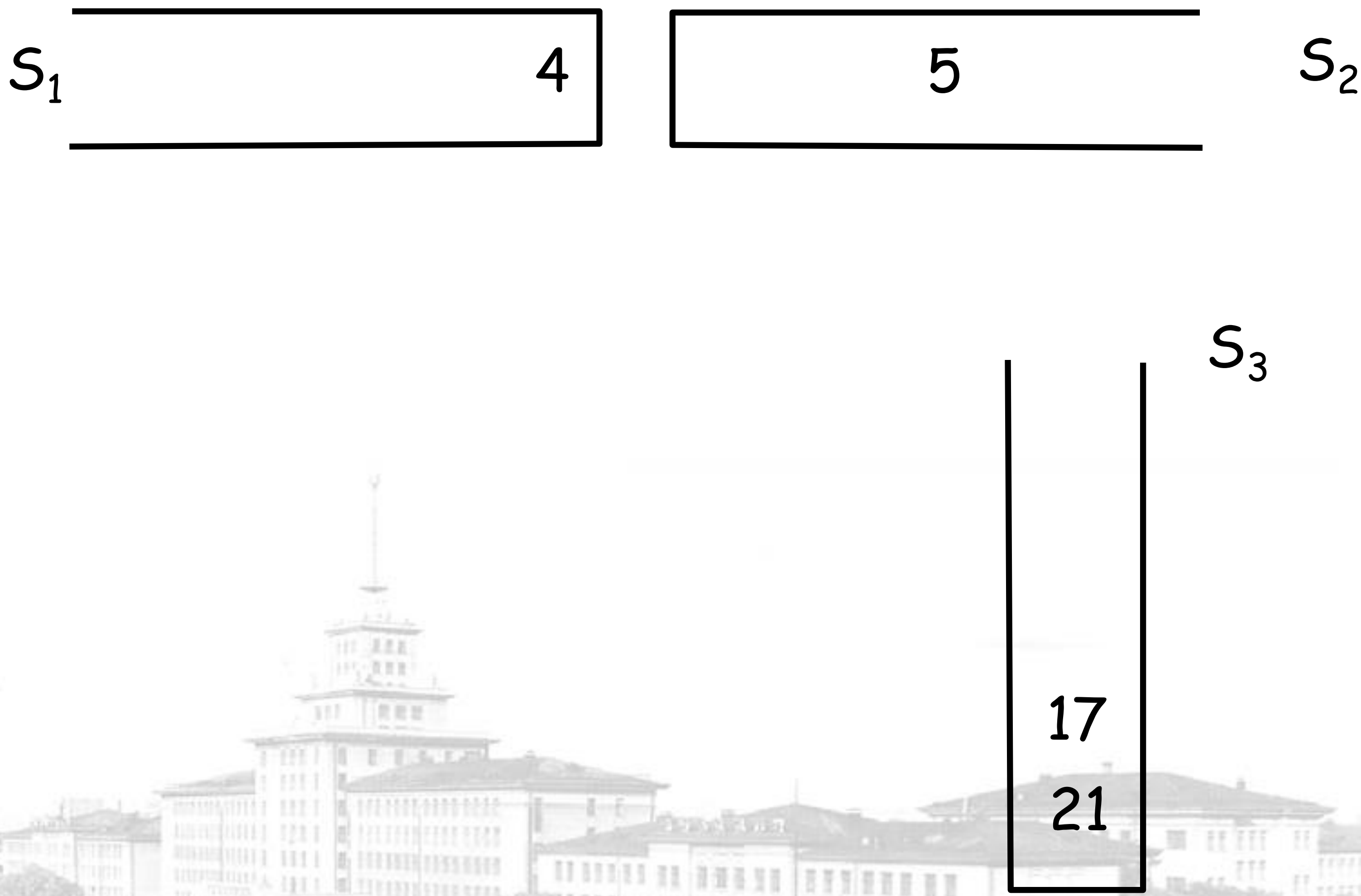
Split



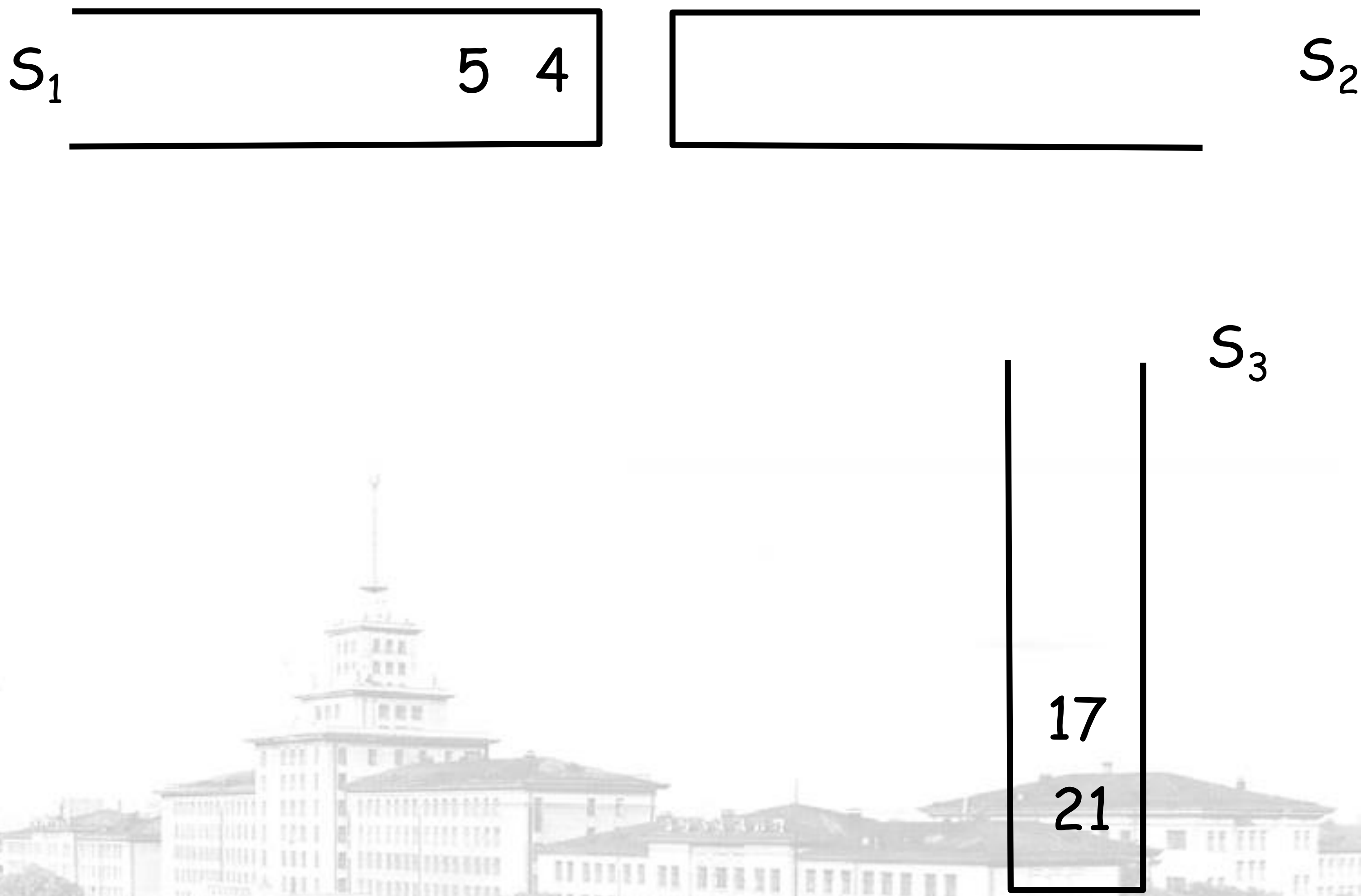
Split



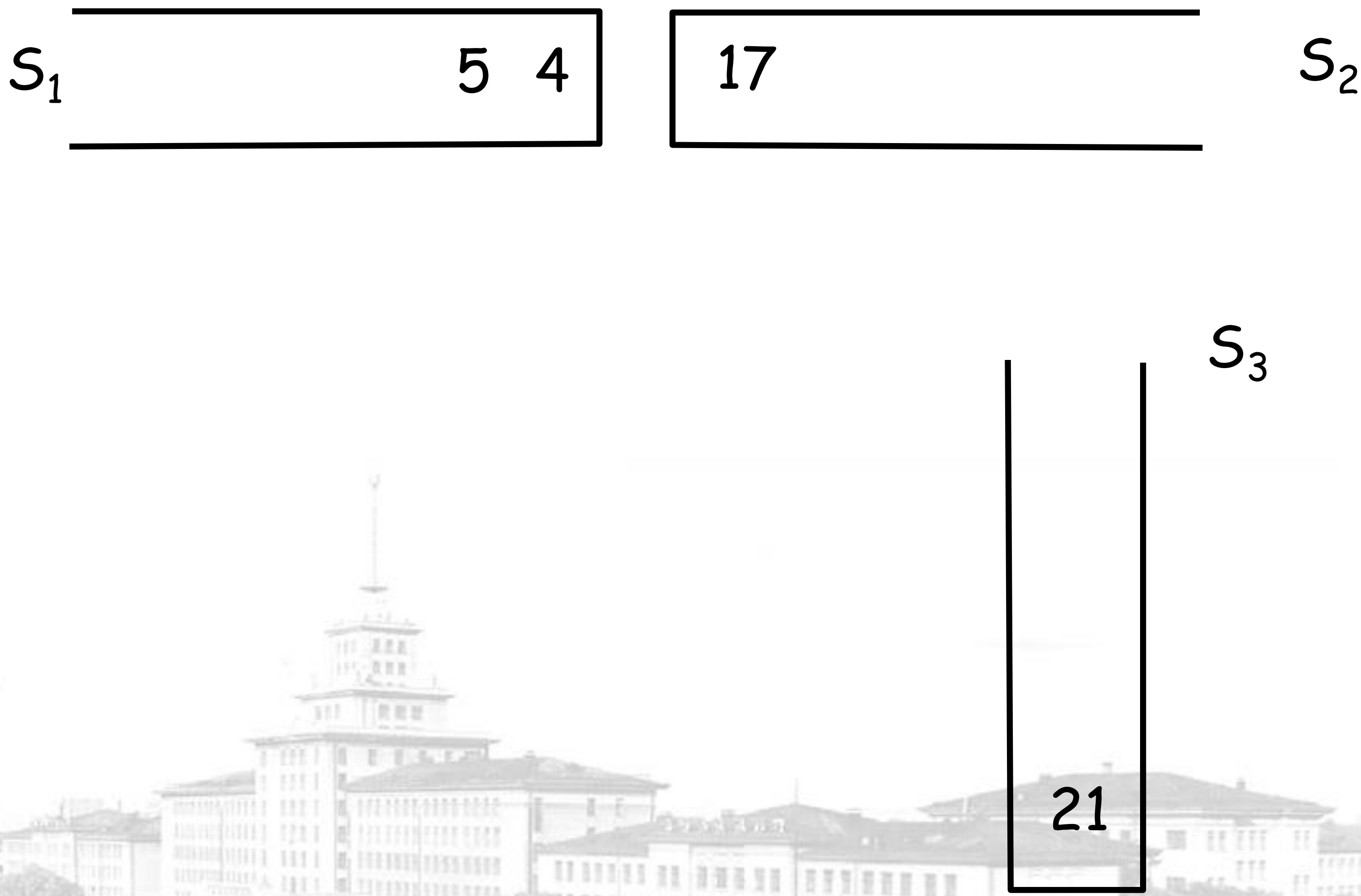
Split



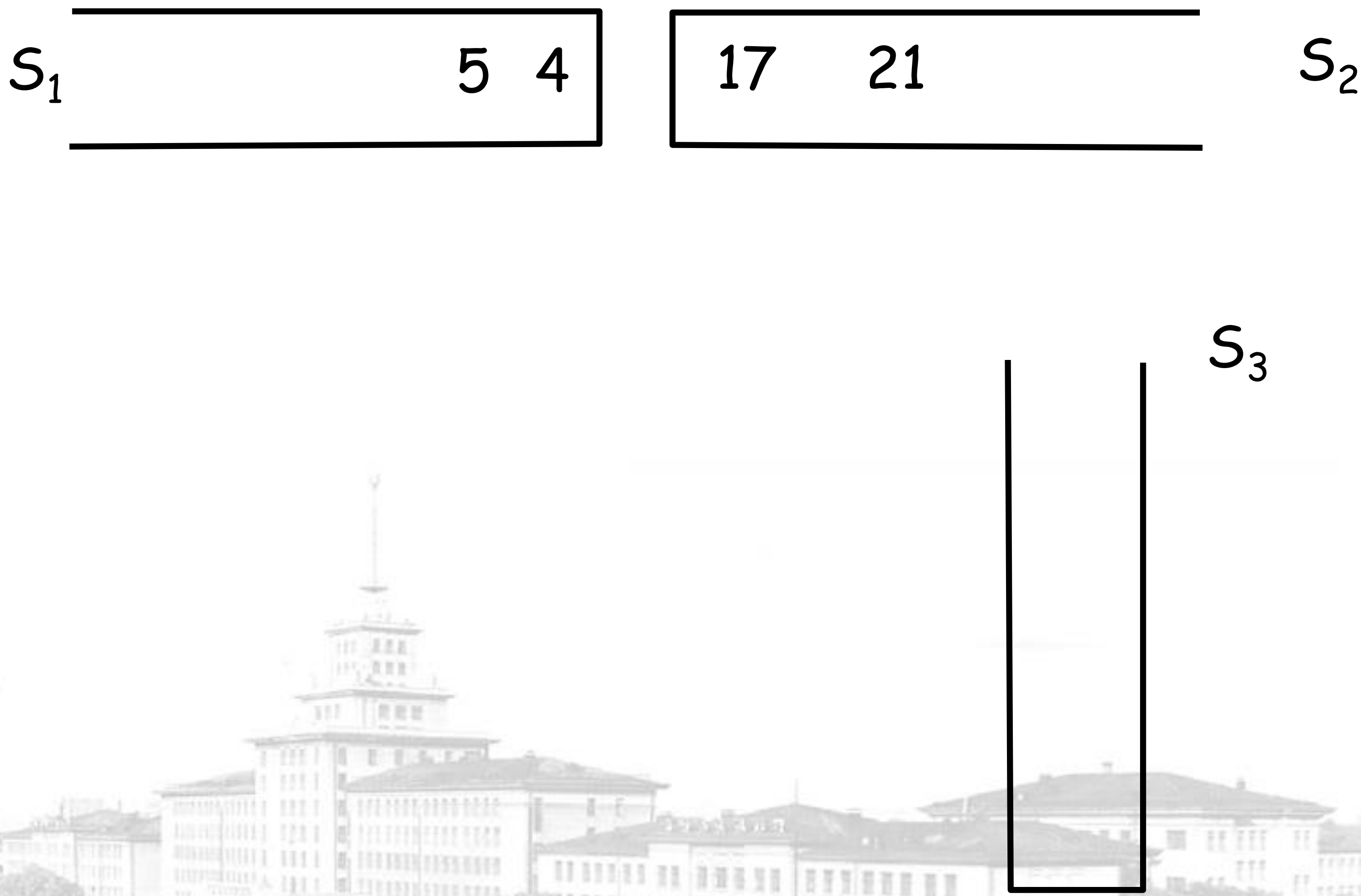
Split



Split



Split



split(S_2, S_1)

$S_3 \leftarrow$ make-stack()

$d \leftarrow$ size(S_2)

while ($i \leq \lfloor d/2 \rfloor$) do

$x \leftarrow$ pop(S_2) push(x, S_3) $i \leftarrow i+1$

while ($i \leq \lceil d/2 \rceil$) do

$x \leftarrow$ pop(S_2) push(x, S_1) $i \leftarrow i+1$

while ($i \leq \lfloor d/2 \rfloor$) do

$x \leftarrow$ pop(S_3) push(x, S_2) $i \leftarrow i+1$

分析

- 每个操作在最坏的情况下花费的时间是O(n)

定理:如果我们用一个空的双端队列进行m个操作，花费的时间为O(m)

更好的界

考虑

$$\Phi(D) = ||S_1| - |S_2||$$

Amortized(op) = actual(op) + $\Delta\Phi$

如果没有split，时间代价为$O(1)$

如果对S_1实施split操作

那么实际时间是 $|S_1| + O(1)$

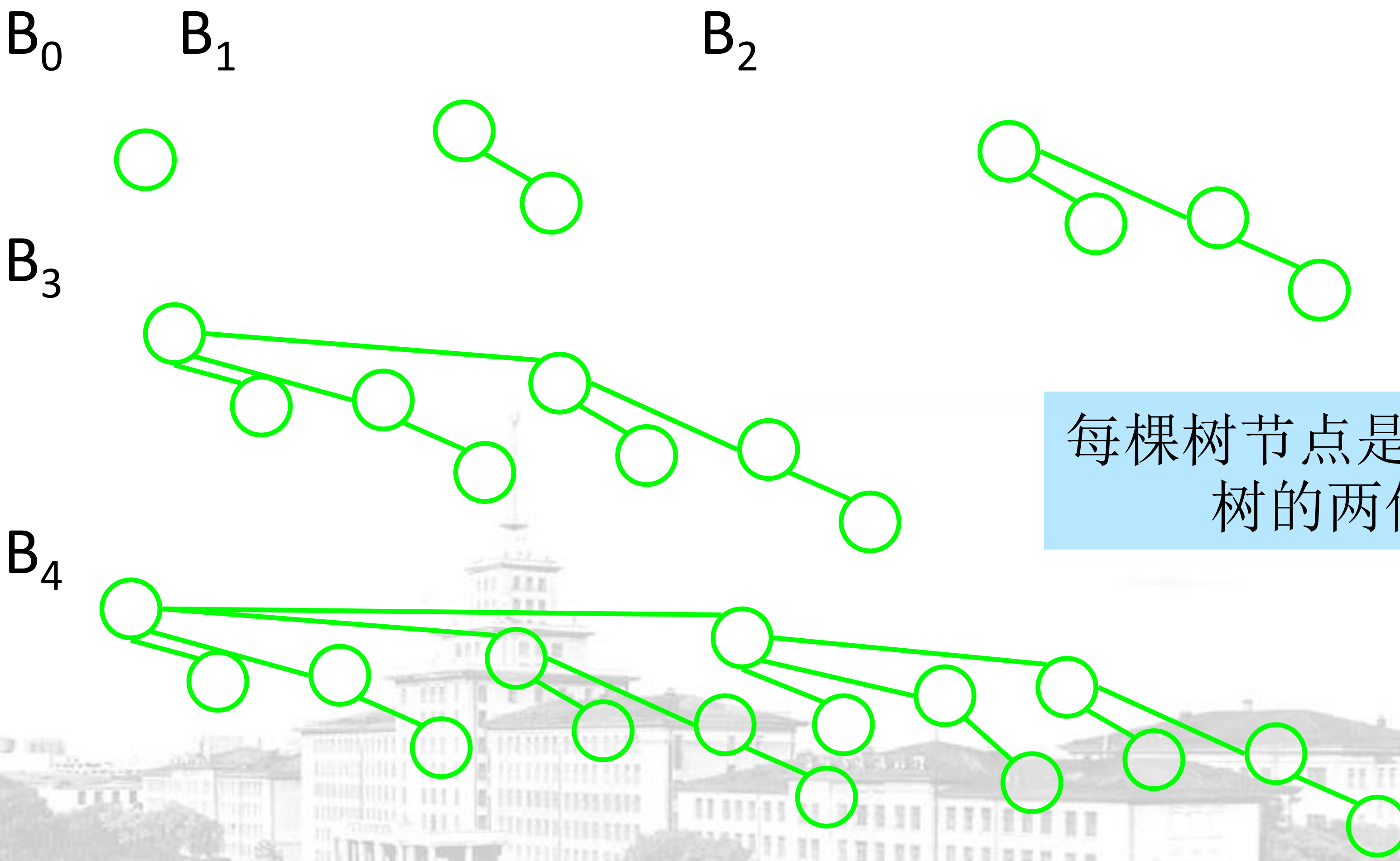
$\Delta\Phi = -|S_1|$

因此平摊代价为 $O(1)$

补充内容二：二项队列

二项队列

- 二叉树:



每棵树节点是之前的树的两倍

二项队列

- 二项式队列
- 一群二叉树，“森林”.
- 每一棵树本质上都是一个二叉树格式的堆.
- 例子: B_0, B_2, B_3
- 插入：创建B_0并且合并
- 删除：从树B_k中删除最小值（根）。

这会剩下一群树: B_0, B_1, ..., B_{k-1}.

将此队列与原始队列的其余部分合并。

二项队列举例

- 最重要的步骤是合并
- 合并规则: (对于两个二项队列)
  - 0 或者1个 B_k 树 ➔ 直接合并
  - 2 个B_k 树 ➔ 合并到 1个 B_{k+1} 树.
  - 3个 B_k trees ➔ 将两个合并为 1个 B_{k+1}, 然后剩下第三个.
- 插入过程:
  - M + 1个步骤，其中M不在森林中的最小树。 当不存在最小的树B_{k+1} 时，最坏情况是k + 2。 k如何与树中的节点总数相关联？

  k = lg n, 那么 (非平摊) 最坏时间复杂度 O(lg n).

二项队列举例

- MakeBinQ问题：建立N个元素的二项式队列。（像makeHeap）。
  这需要多长时间？
- 插入时最坏情况下的运行时间:
  - 一个插入操作最坏情况下的时间O(lg n) ➔ n个插入操作最坏情况下的时间O(n lg n)、
    但是我们想让它为 O(n) – 像makeHeap一样
- 直接尝试平摊分析：
  - 考虑合并的每个连接步骤。 第一，第三，第五，ettc ...奇数步骤不需要连接步骤，因为不会有B_0。 所以所有插入的1/2都不需要连接，类似地，只需要1个连接步骤。
  - 我们可以按这种方式继续思考，但是删除时会遇到困难（我们需要势能分析）。

二项队列举例

- 间接分析（时间= M + 1）
  - 没有B_0 ➔ 代价为1 (M = 0)
    - 结果是 1 个B_0 树 加入森林
  - 有 B_0 没有B_1 ➔ 代价为 2 (M =1)
    - 结果一样(新的B_1 但是 B_0 消失)
  - 当大家为 3 (M = 2)
    - 减少1个树（新的B2但删除B0和B1）...
  - 当代价为 c (M = c – 1)
    - 结果增加2 - c棵树

二项队列举例

- 增加2 - c棵树
- 怎样去使用它？

T_i =第i次迭代后的树数

$T_0 = 0$ =最初的＃棵树

C_i =第i次迭代的成本

那么，$T_i = T_{i-1} + (2-C_i) \Leftrightarrow$

$$C_i + (T_i - T_{i-1}) = 2$$

这只是第i次迭代

二项队列举例

$$C_i + (T_i - T_{i-1}) = 2$$

- 获得所有迭代：

$$C_1 + (T_1 - T_0) = 2$$

$$C_2 + (T_2 - T_1) = 2$$

…

$$C_{n-1} + (T_{n-1} - T_{n-2}) = 2$$

$$C_n + (T_n - T_{n-1}) = 2$$

$$\sum_{i=1}^{n} C_i + (T_n - T_0) = 2n$$

二项队列举例

$$\sum_{i=1}^{n} C_i + (T_n - T_0) = 2n$$

- $T_0 = 0$且T_n绝对不是负数，所以T_n-T_0不是负数。

$$\Rightarrow \sum_{i=1}^{n} C_i <= 2n$$

因此，总代价<2n $\Rightarrow$

makeBinQ = O(n)

因为，makeBinQ由O（n）次插入组成，所以每个插入的最坏情况是O（1）。

补充内容三：斐波那契堆

优先队列的性能汇总

操作	链表	二进制堆	二项堆	斐波那契堆†	Relaxed Heap
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 是优先队列中的元素数量　　　　† 平摊后

优点队列的性能汇总

操作	链表	二进制堆	二项堆	斐波那契堆 †	Relaxed Heap
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 是优先队列中的元素数量

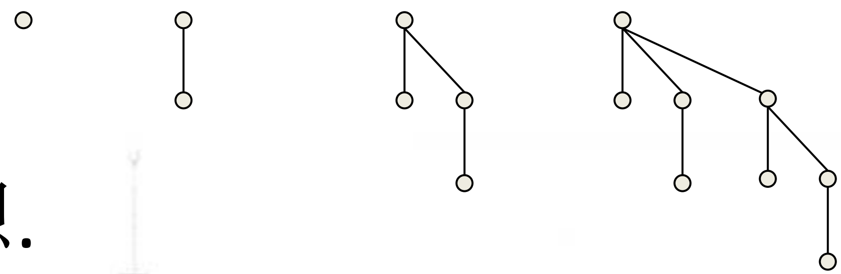
† 平摊后

斐波那契堆

- 历史. [Fredman and Tarjan, 1986]
  - 巧妙的数据结构和分析.
  - 最初的动机：提高 Dijkstra 最短路径算法的性能

    V次insert, V次delete-min, E次decrease-key

    从 $O(E \log V)$ 到 $O(E + V \log V)$.



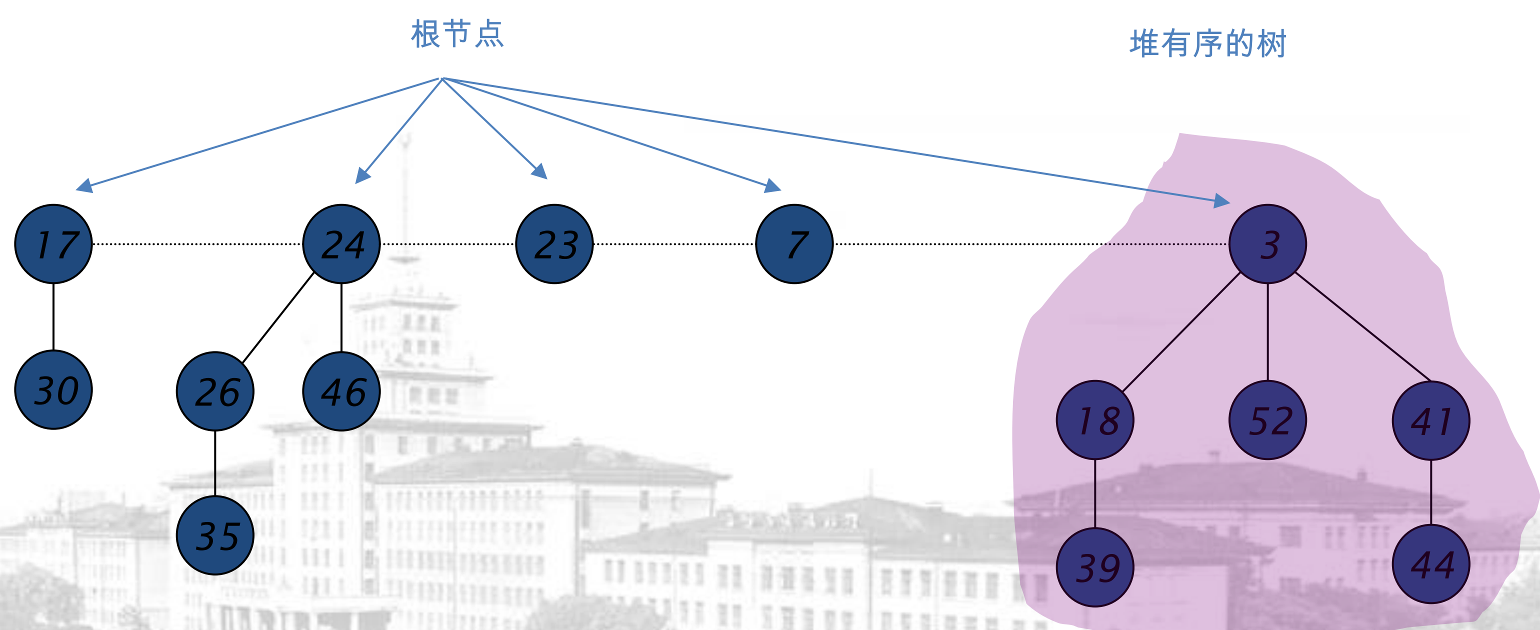
- 基本思想.
  - 类似于二项堆，但结构更加灵活。
  - 二项堆: 在每次*insert*之后都会急切地合并树。

斐波那契堆：结构

- 斐波那契堆.

  每个父节点比它的子节点小.

  - 一个堆有序（heap-ordered）树集合.
  - 保持指向最小元素的指针.
  - 一组标记的节点.

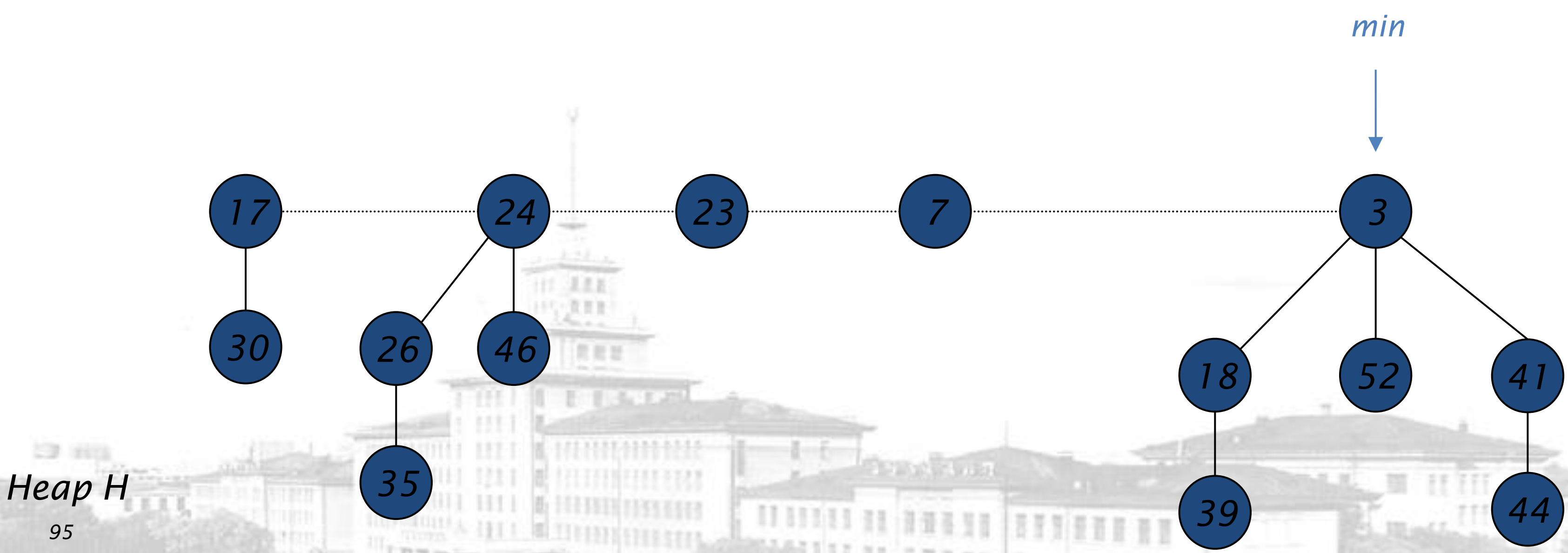


Heap H

斐波那契堆：结构

- 斐波那契堆.
  - 一个堆有序（ heap-ordered ）树集合.
  - 保持指向最小元素的指针.
  - 一组标记的节点.

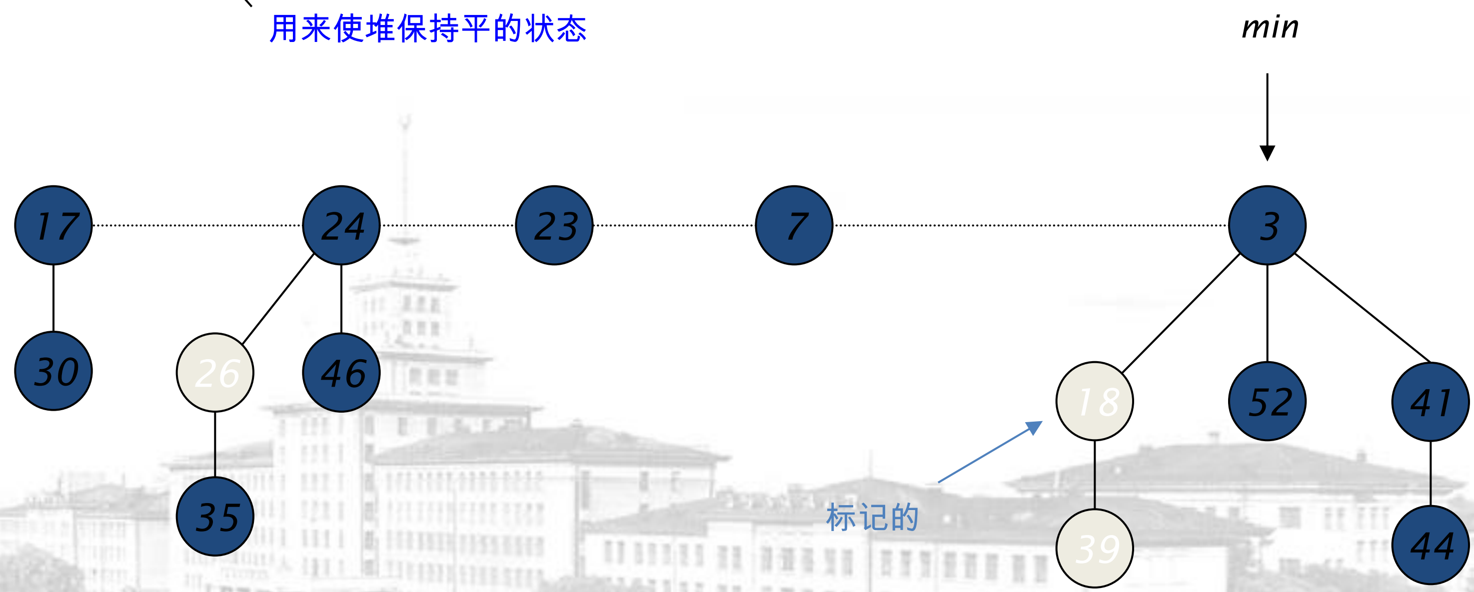
find-min 花费 *O(1)* 的时间



斐波那契堆：结构

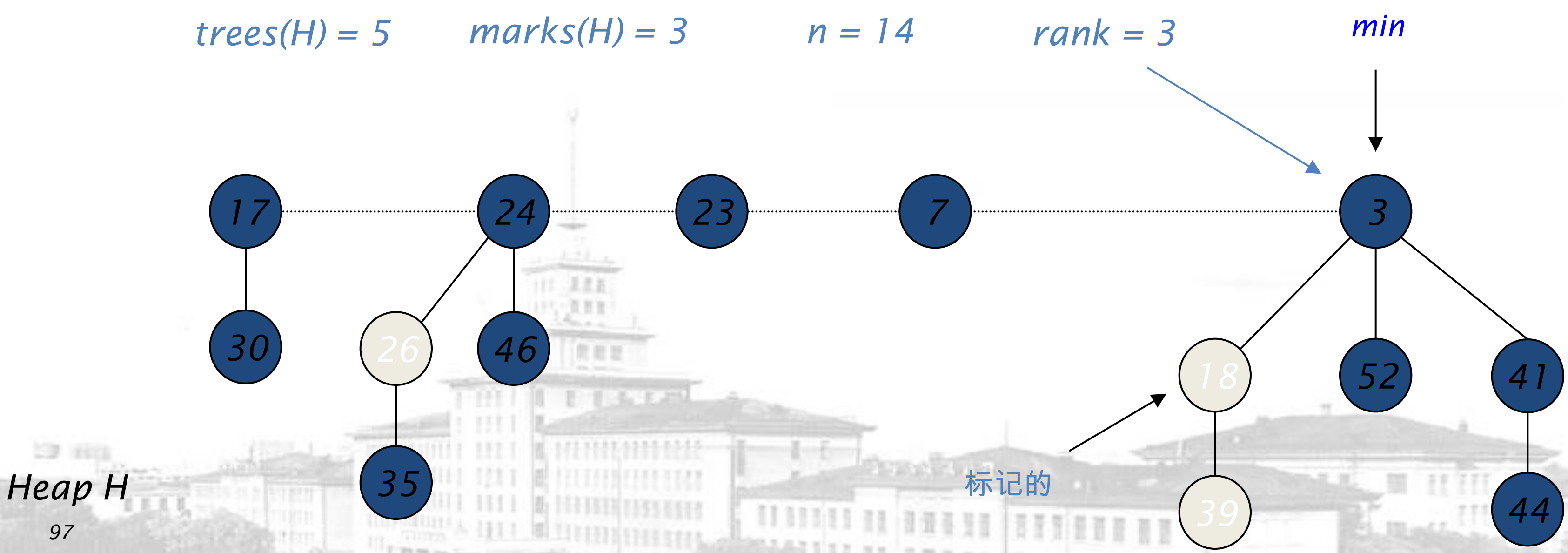
- 斐波那契堆.
  - 一个堆有序（ heap-ordered ）树集合.
  - 保持指向最小元素的指针.
  - 一组标记的节点.

用来使堆保持平的状态



斐波那契堆：符号说明

- 符号说明.
  - *n* = 堆中节点的数目
  - *rank(x)* = 节点x的子节点数目.
  - *rank(H)* = 堆 *H* 中所有节点中最大的*rank*.
  - *trees(H)* = 堆 *H* 中所有树的数目.
  - *marks(H)* = 堆 *H* 中标记节点的数目.

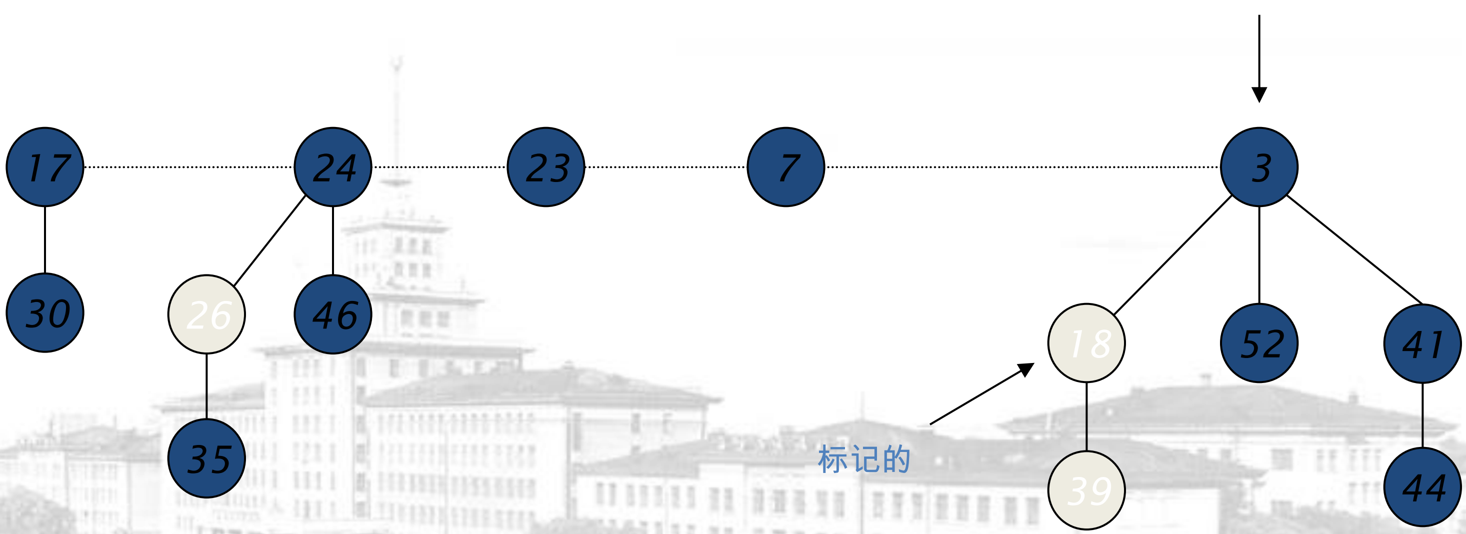


斐波那契堆： 势函数

$$\Phi(H) = trees(H) + 2 \cdot marks(H)$$

堆 H的势函数

$trees(H) = 5$ $marks(H) = 3$ $\Phi(H) = 5 + 2 \cdot 3 = 11$ min

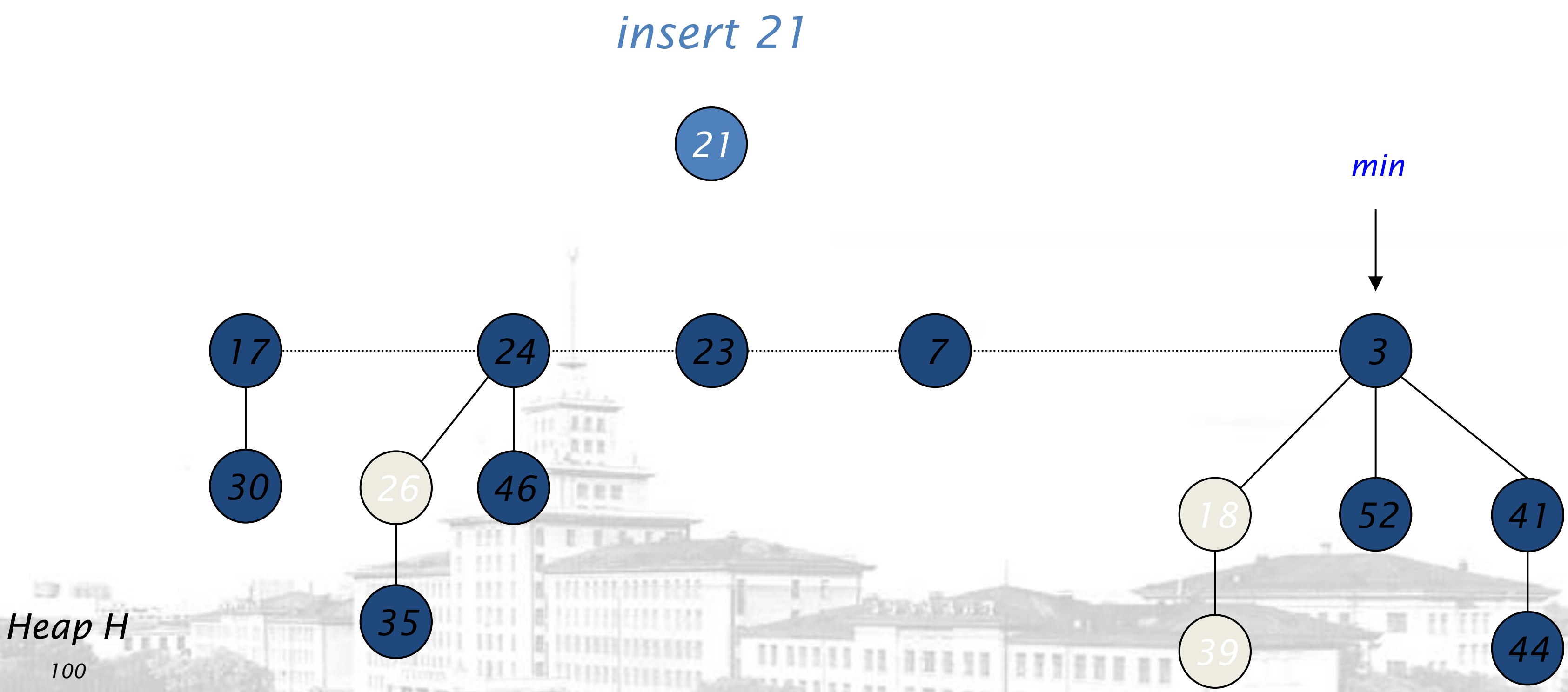


Heap H

Insert

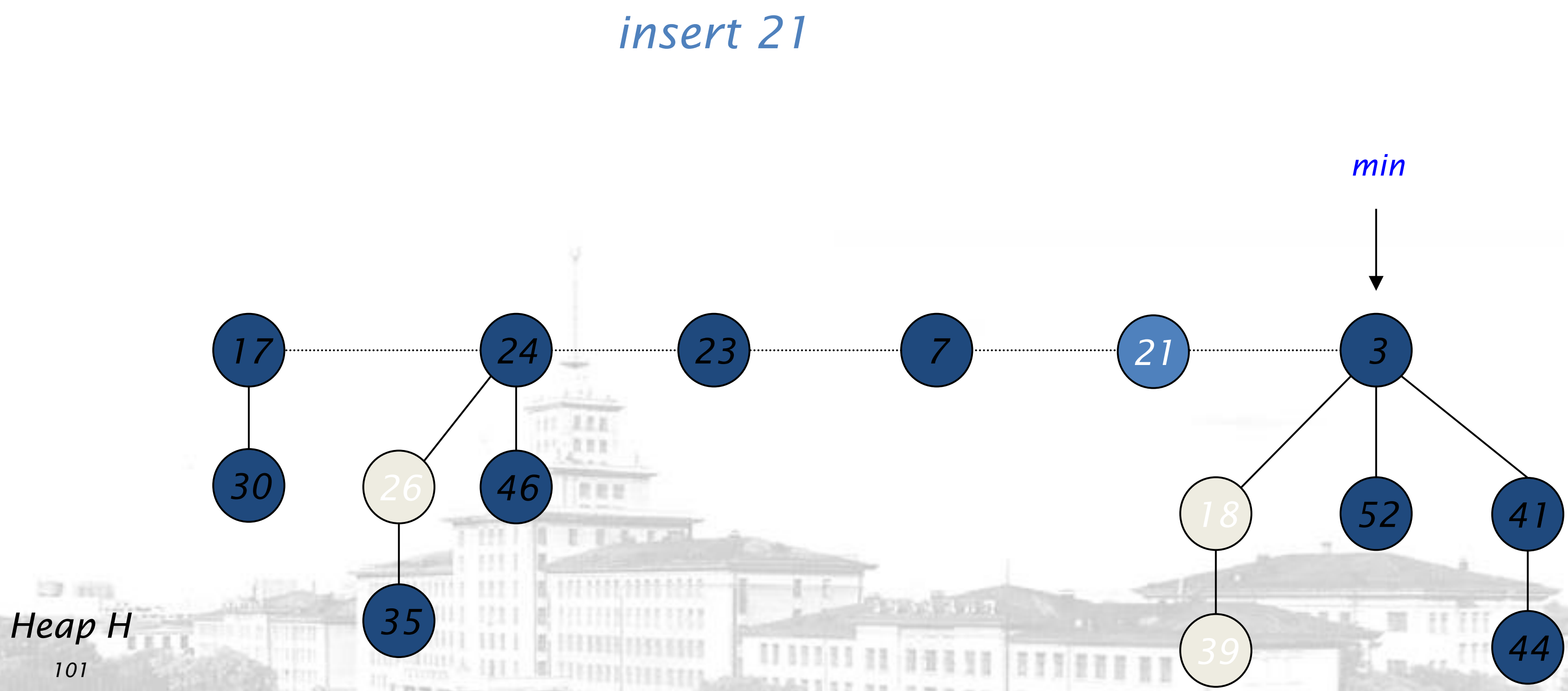
斐波那契堆：Insert

- Insert.
  – 创建一棵仅包含一个节点的树.
  – 将其添加到根节点的表中；更新指向最小值的指针（如果有必要的话）.



斐波那契堆：Insert

- Insert.
  - 创建一棵仅包含一个节点的树.
  - 将其添加到根节点列表中；更新指向min的指针（如果有必要的话）.



斐波那契堆：Insert的分析

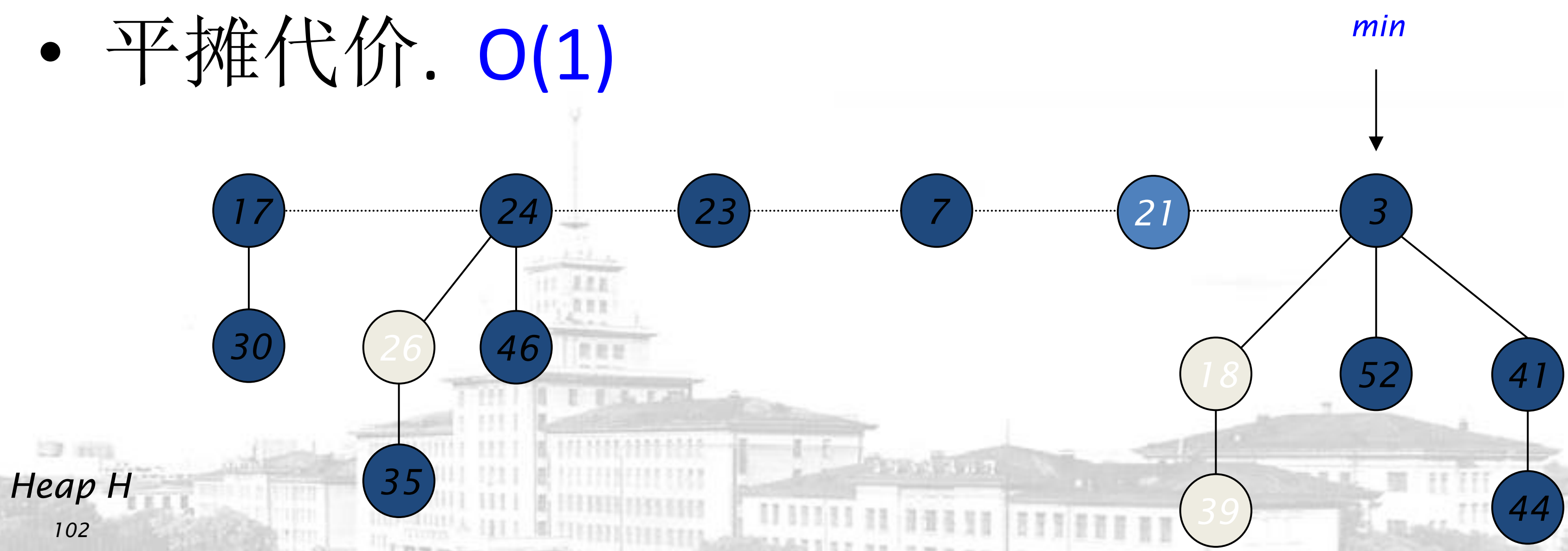
$$\Phi(H) = trees(H) + 2 \cdot marks(H)$$

堆H的势函数

- 实际代价. O(1)

- 势函数的改变. +1

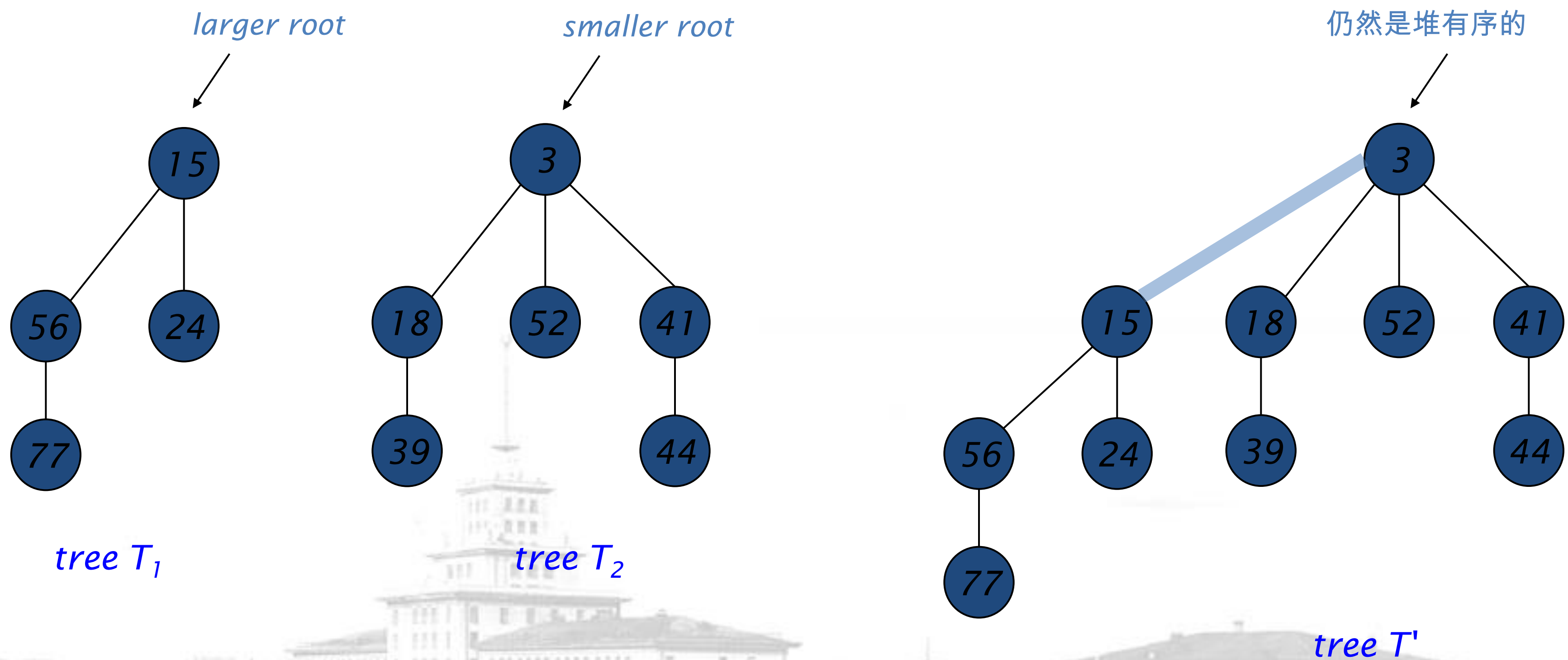
- 平摊代价. O(1)



Delete Min

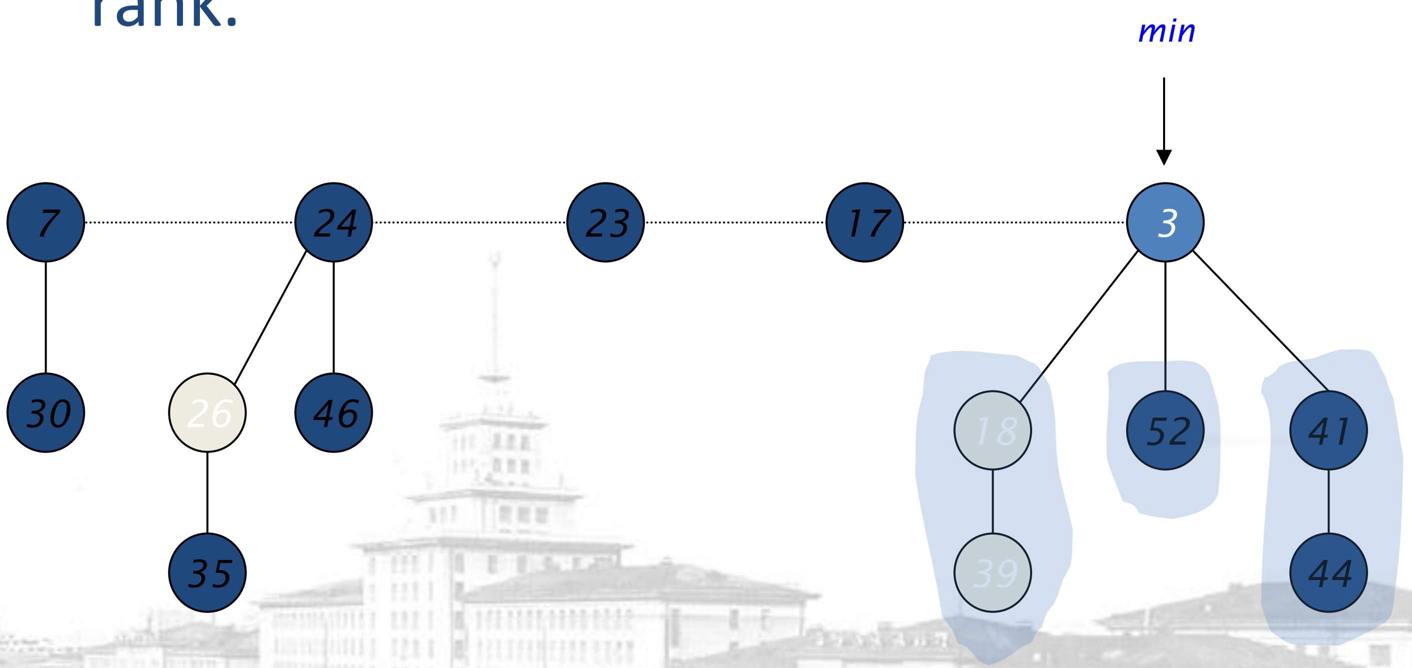
链接操作

- 链接操作. 让larger root成为smaller root的子节点。



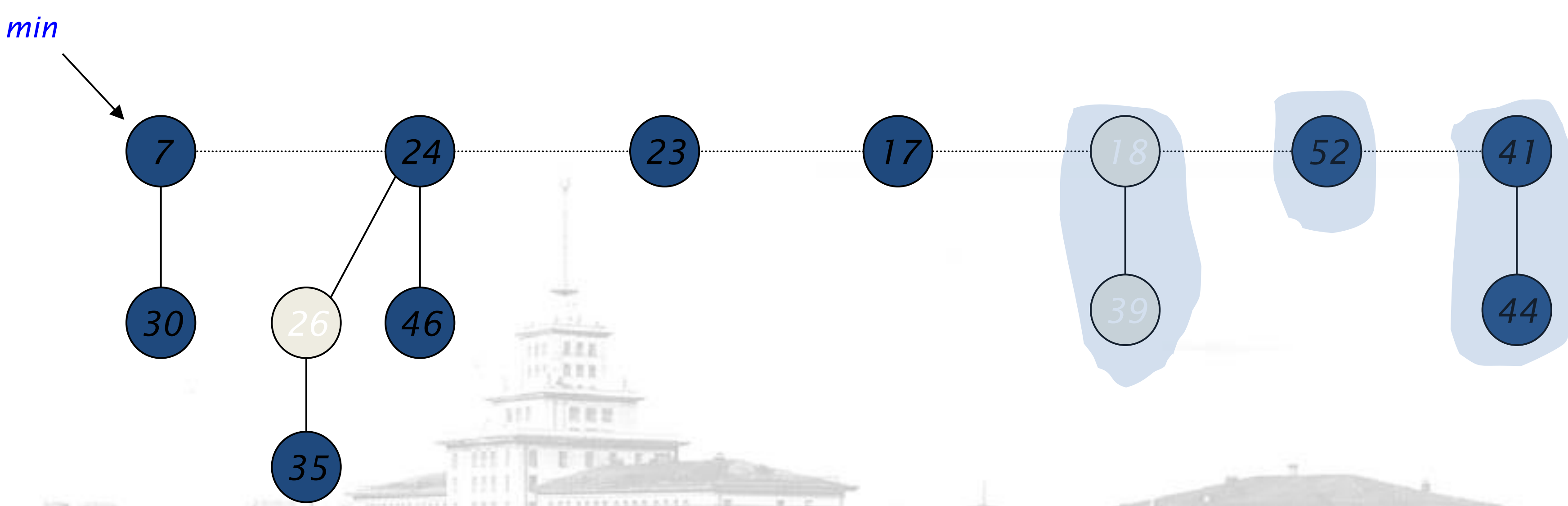
斐波那契堆：Delete Min

- Delete min.
  - 删除 min; 将其子节点放到根节点列表中; 更新 min.
  - 合并tree，使得没有两个根节点具有相同的 rank.



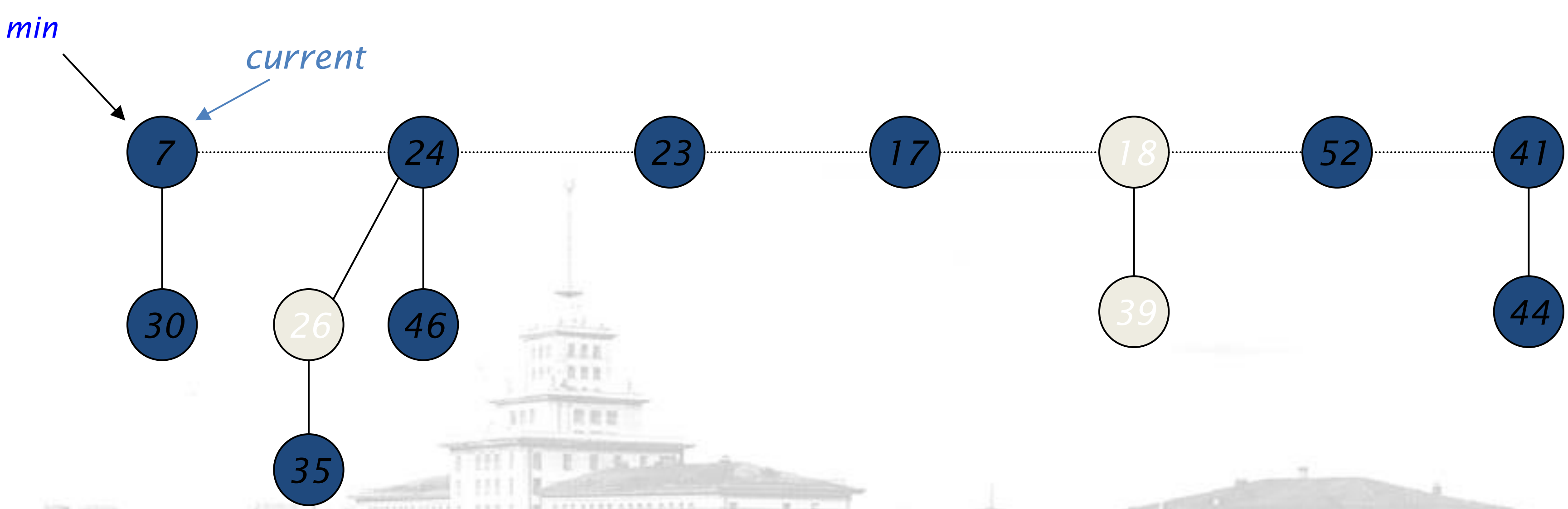
斐波那契堆：Delete Min

- Delete min.
  - 删除 min; 将其子节点放到根节点列表中; 更新 min.
  - 合并tree，使得没有两个根节点具有相同的 rank.



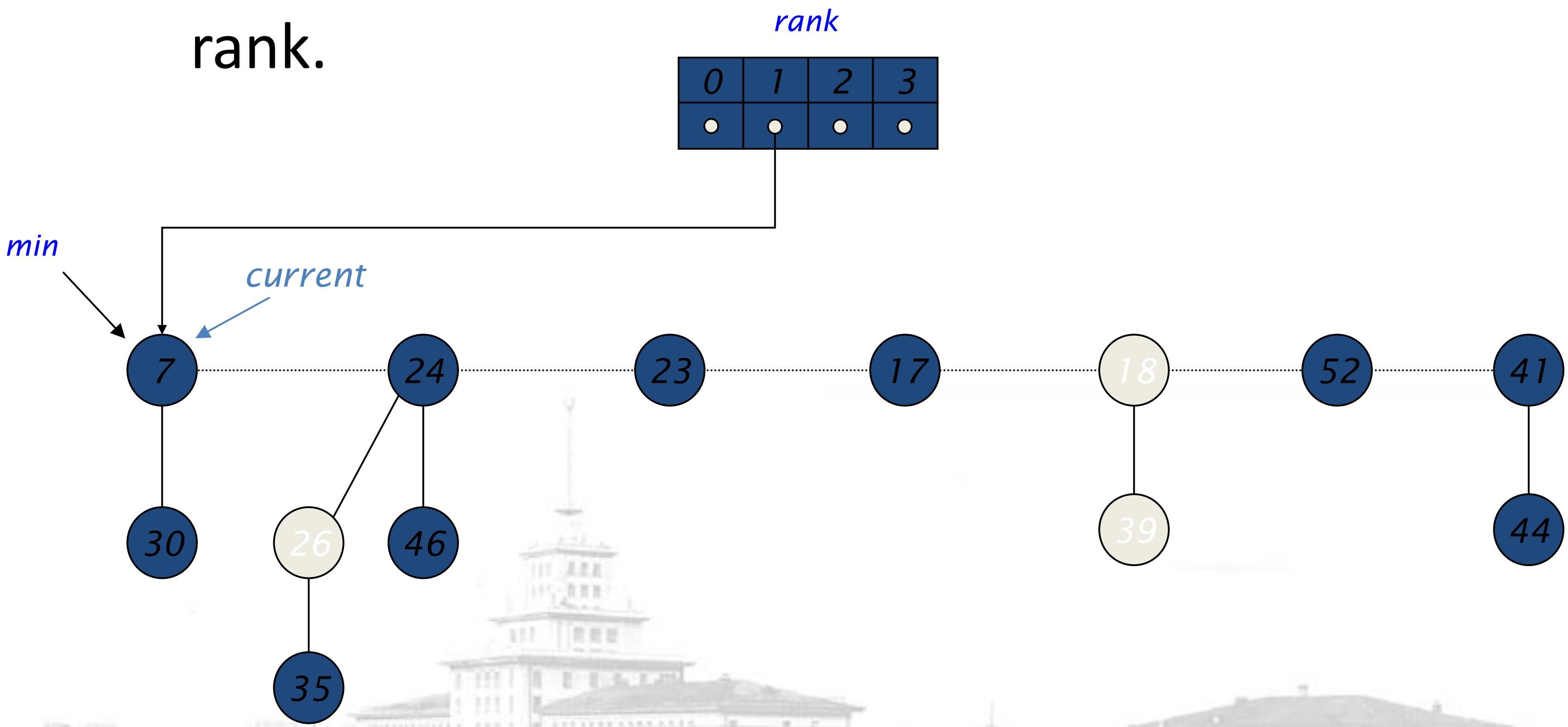
斐波那契堆：Delete Min

- Delete min.
  - 删除 min; 将其子节点放到根节点列表中; 更新 min.
  - 合并tree，使得没有两个根节点具有相同的 rank.



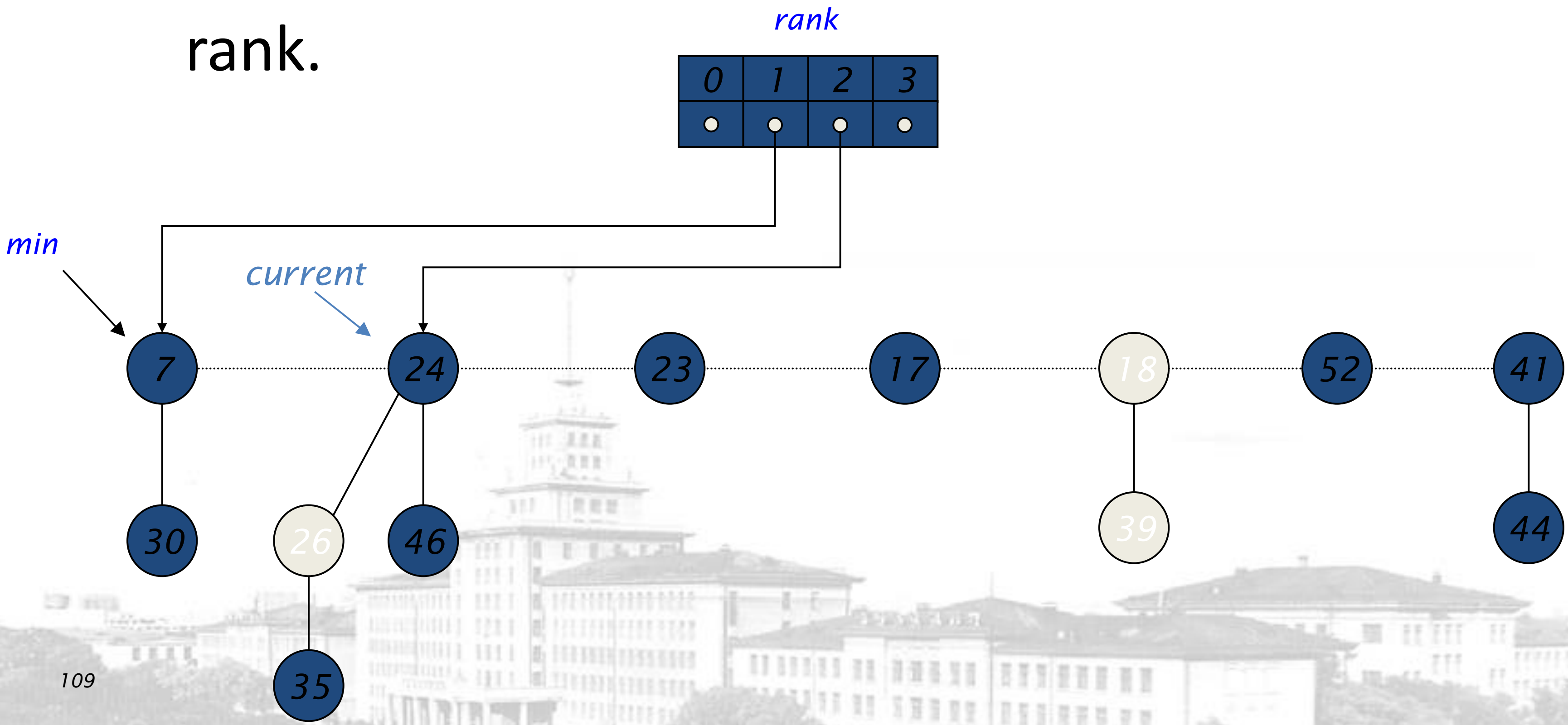
斐波那契堆：Delete Min

- Delete min.
  - 删除 min; 将其子节点放到根节点列表中; 更新 min.
  - 合并tree，使得没有两个根节点具有相同的 rank.



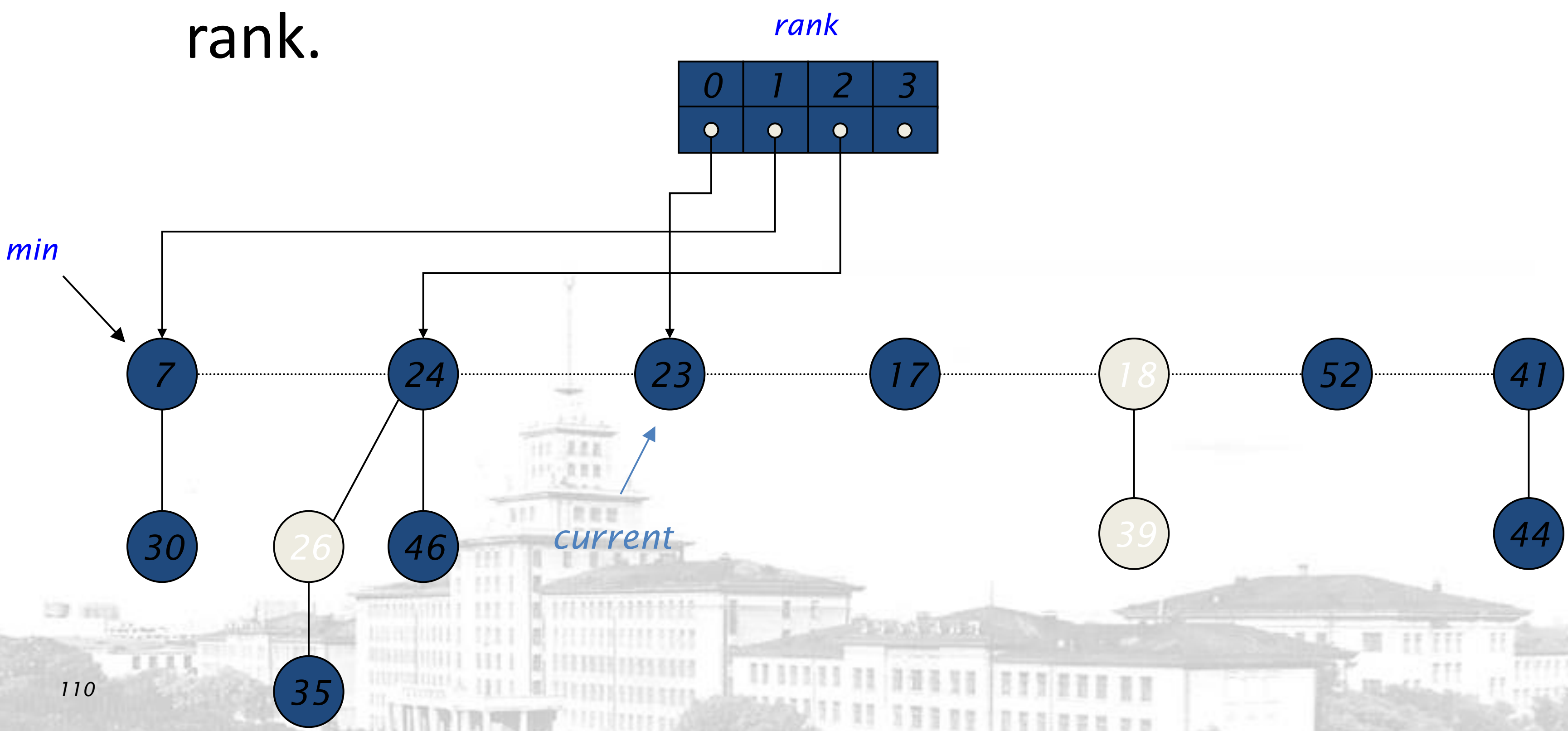
斐波那契堆：Delete Min

- Delete min.
  - 删除 min; 将其子节点放到根节点列表中; 更新 min.
  - 合并tree，使得没有两个根节点具有相同的 rank.



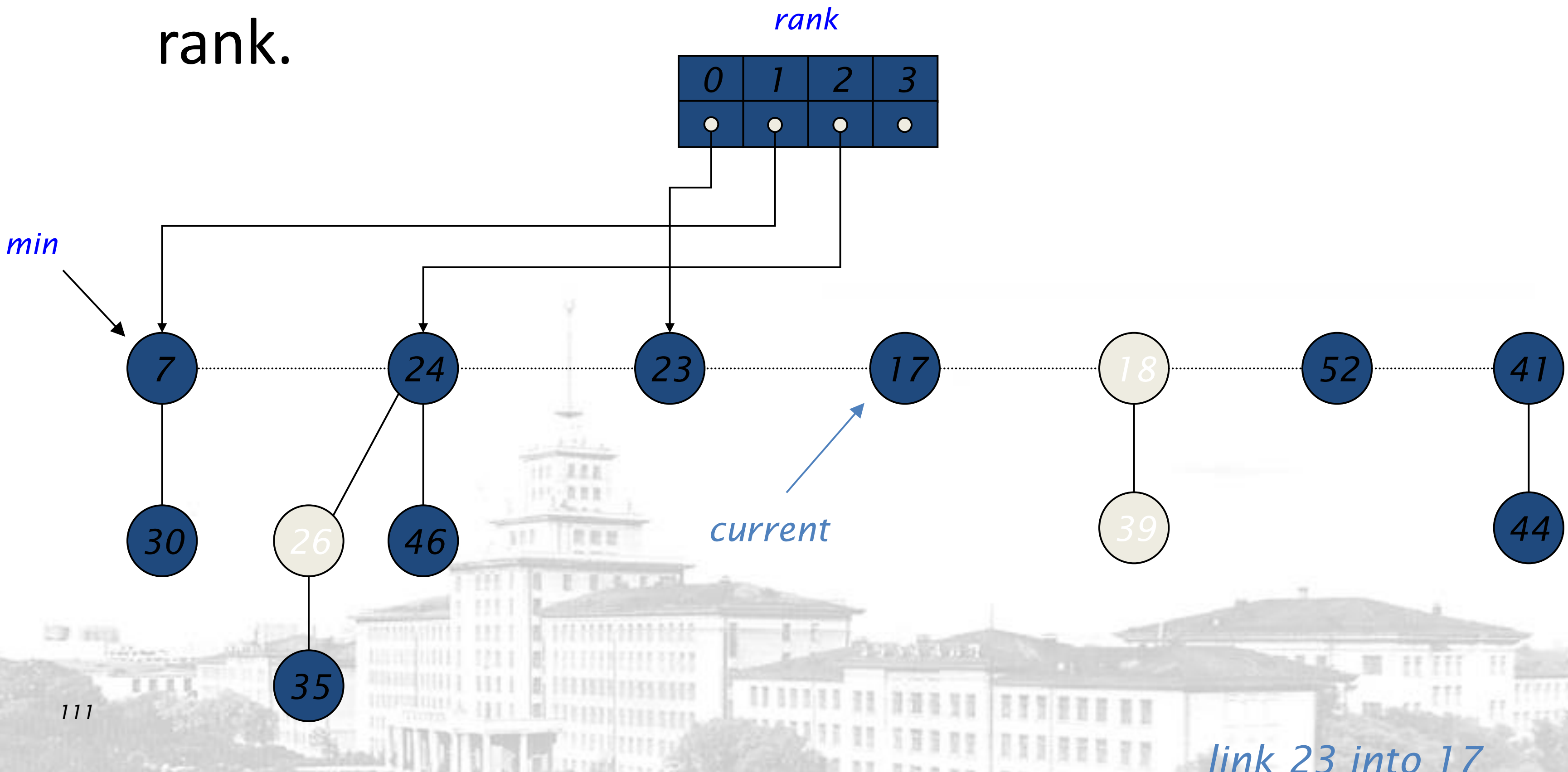
斐波那契堆：Delete Min

- Delete min.
  - 删除 min; 将其子节点放到根节点列表中; 更新 min.
  - 合并tree，使得没有两个根节点具有相同的 rank.



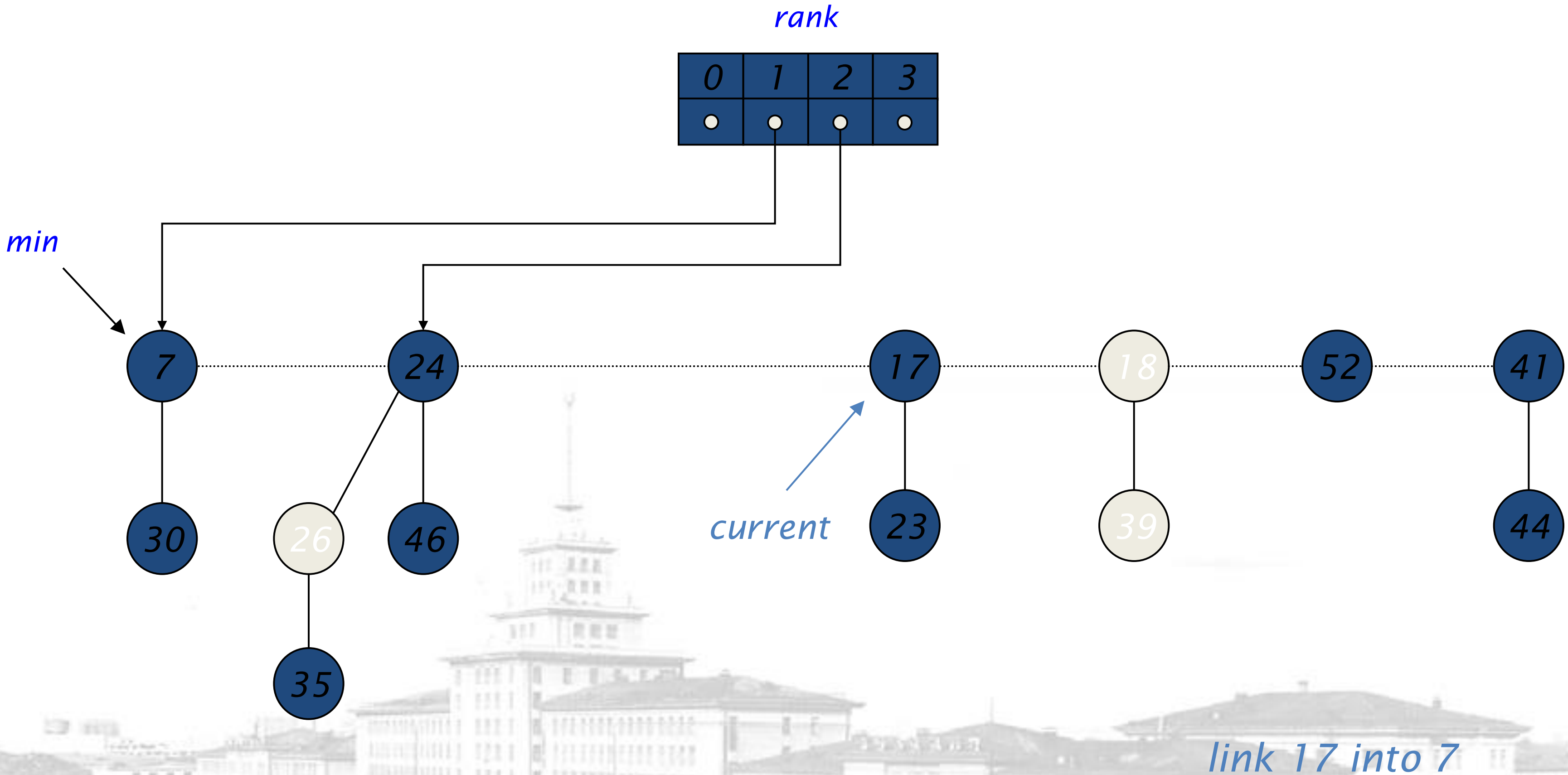
斐波那契堆：Delete Min

- Delete min.
  - 删除 min; 将其子节点放到根节点列表中; 更新 min.
  - 合并tree，使得没有两个根节点具有相同的 rank.



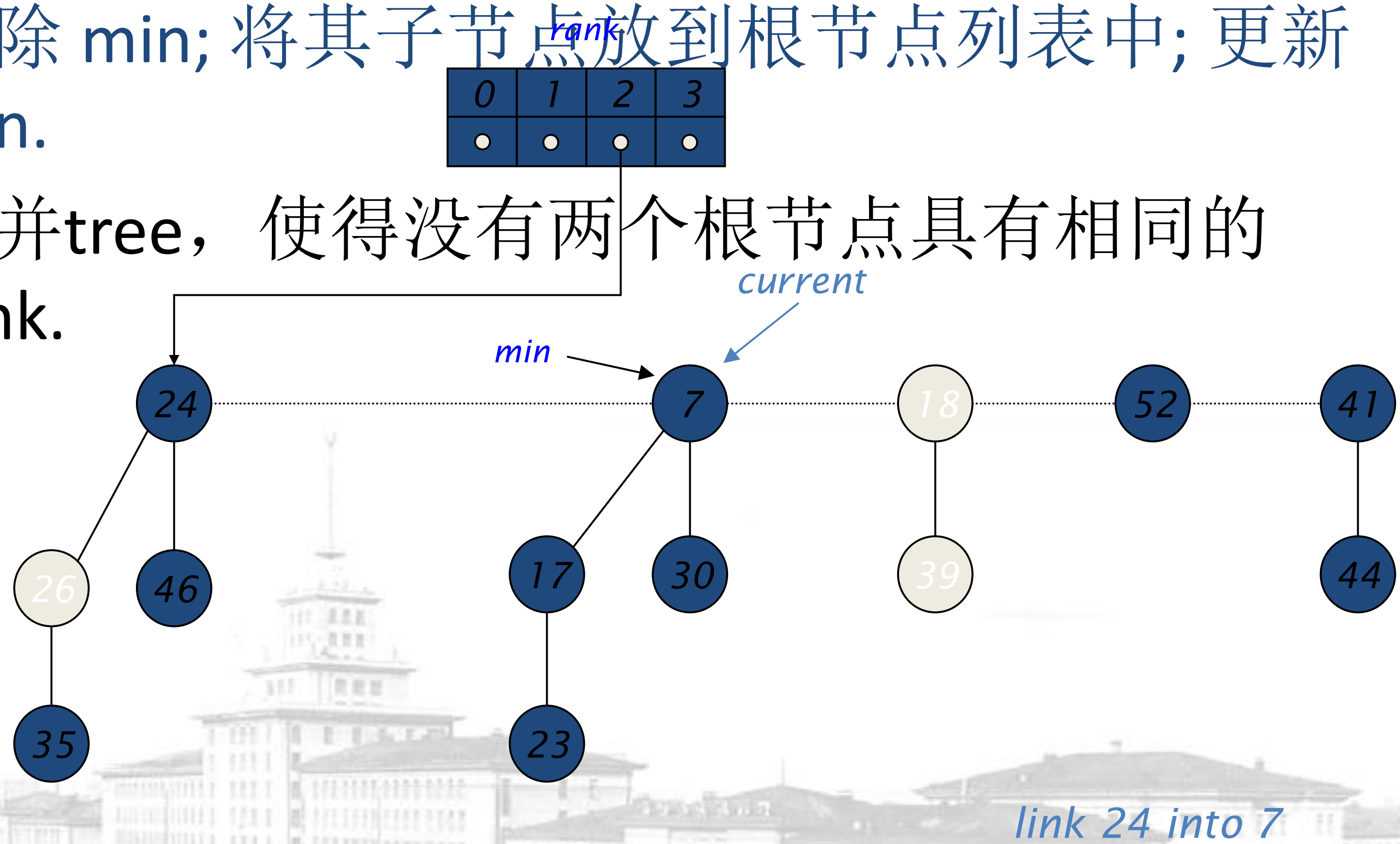
斐波那契堆：Delete Min

- Delete min.
  - 删除 min; 将其子节点放到根节点列表中; 更新 min.
  - 合并tree，使得没有两个根节点具有相同的rank.



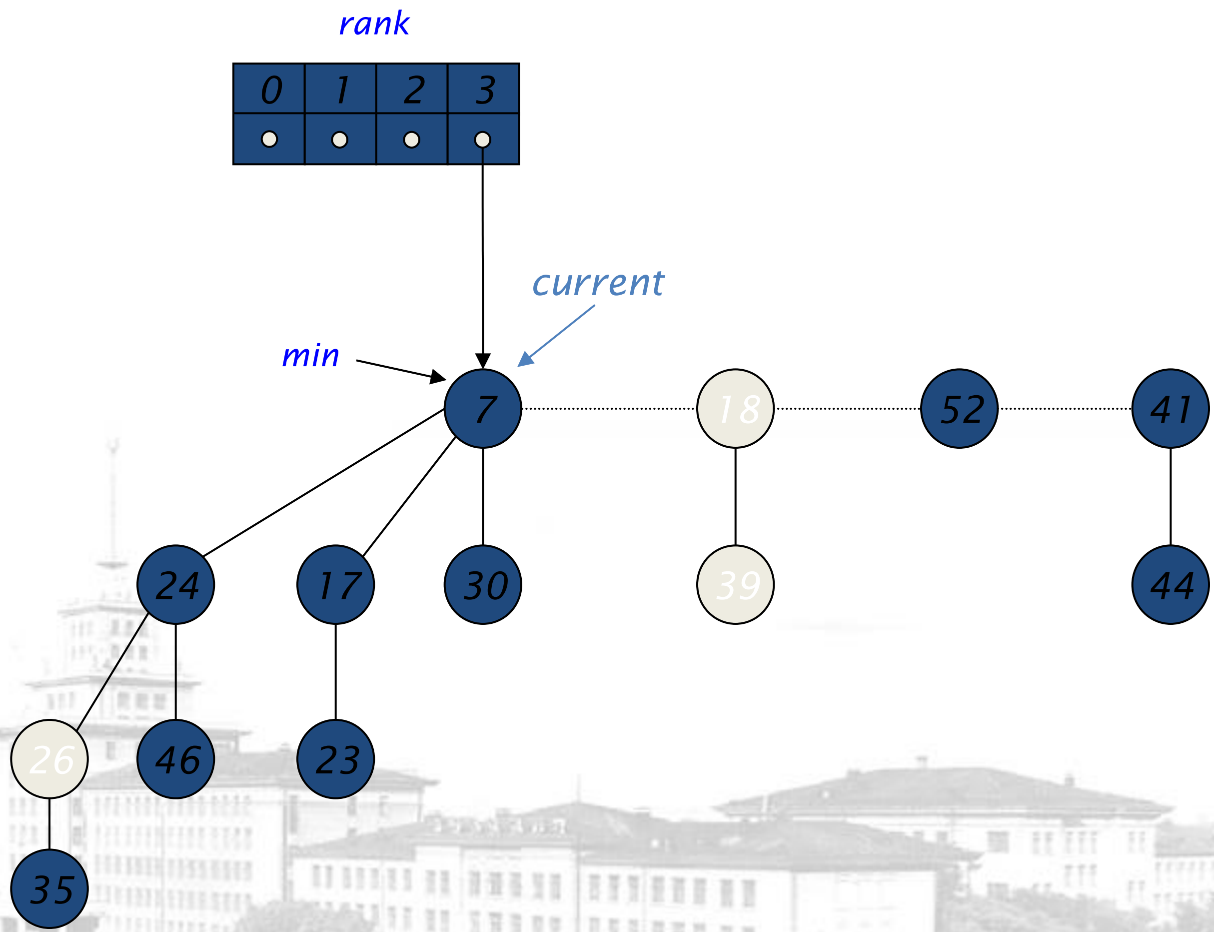
斐波那契堆：Delete Min

- Delete min.
  - 删除 min; 将其子节点放到根节点列表中; 更新 min.
  - 合并tree，使得没有两个根节点具有相同的 rank.



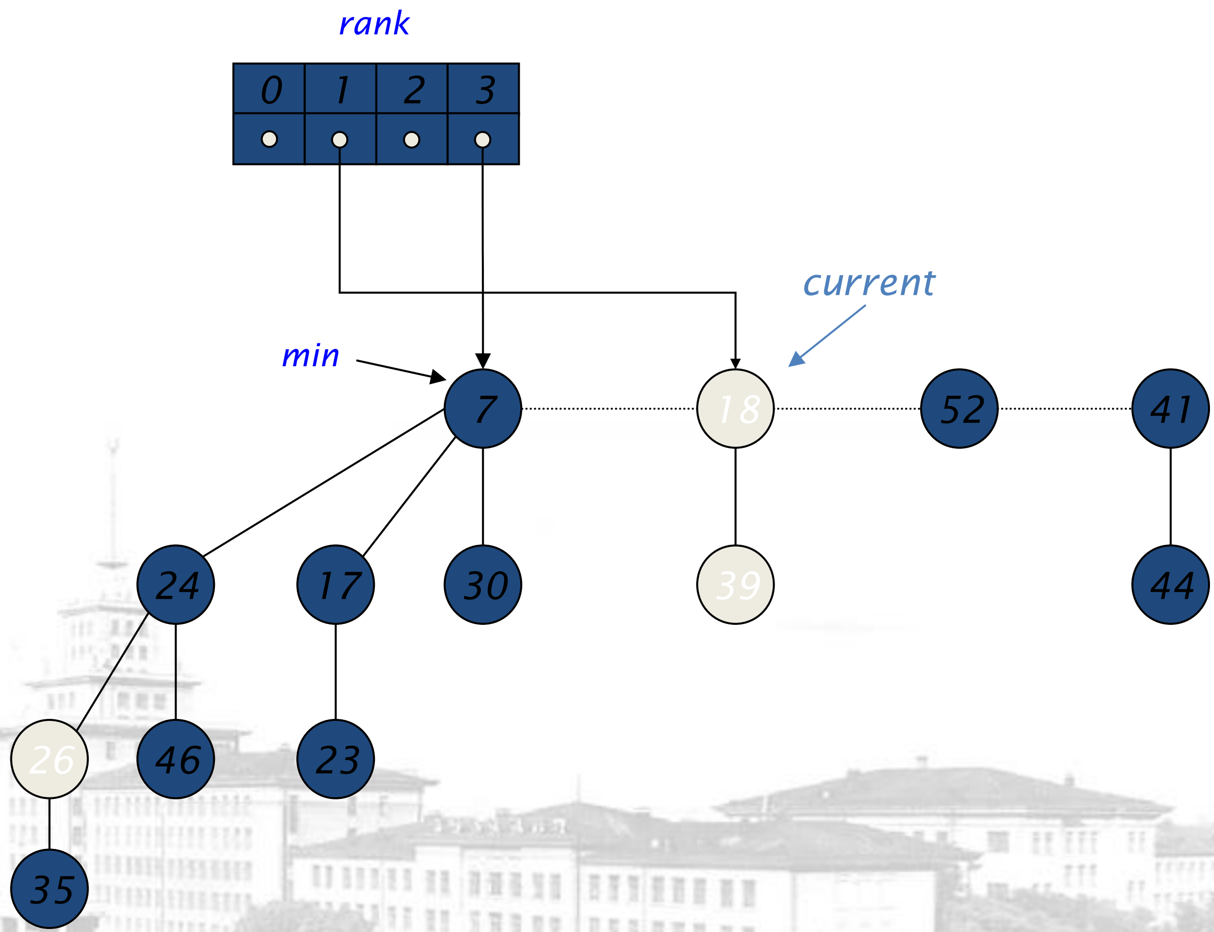
斐波那契堆：Delete Min

- Delete min.
  - 删除 min; 将其子节点放到根节点列表中; 更新 min.
  - 合并tree，使得没有两个根节点具有相同的rank.



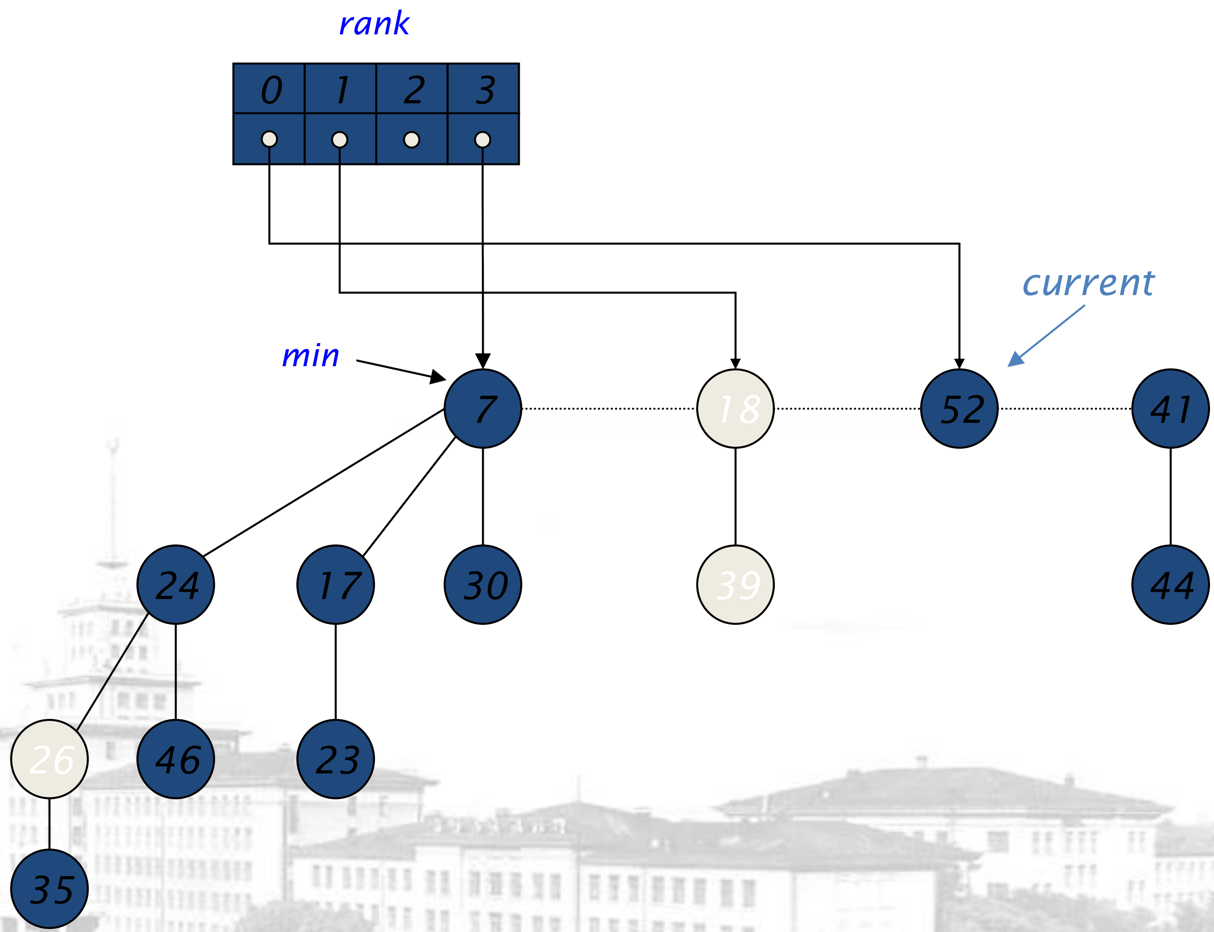
斐波那契堆：Delete Min

- Delete min.
  - 删除 min; 将其子节点放到根节点列表中; 更新 min.
  - 合并tree，使得没有两个根节点具有相同的rank.



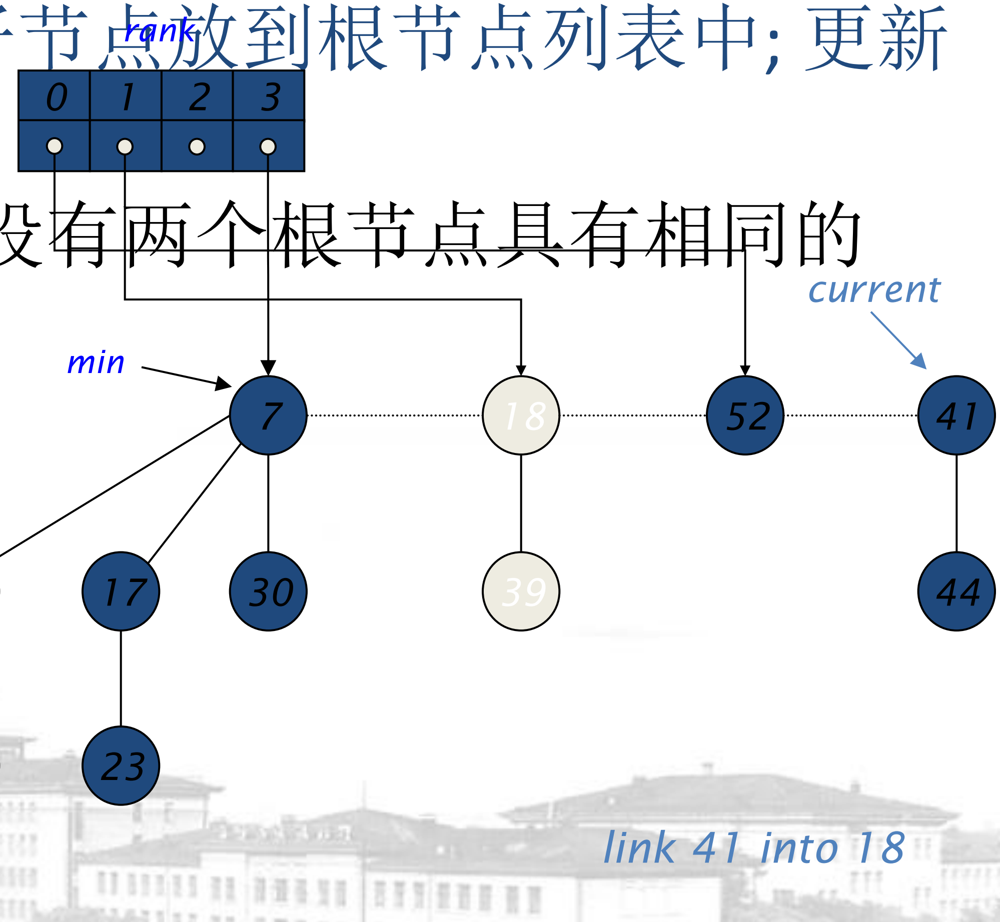
斐波那契堆：Delete Min

- Delete min.
  - 删除 min; 将其子节点放到根节点列表中; 更新 min.
  - 合并tree，使得没有两个根节点具有相同的rank.



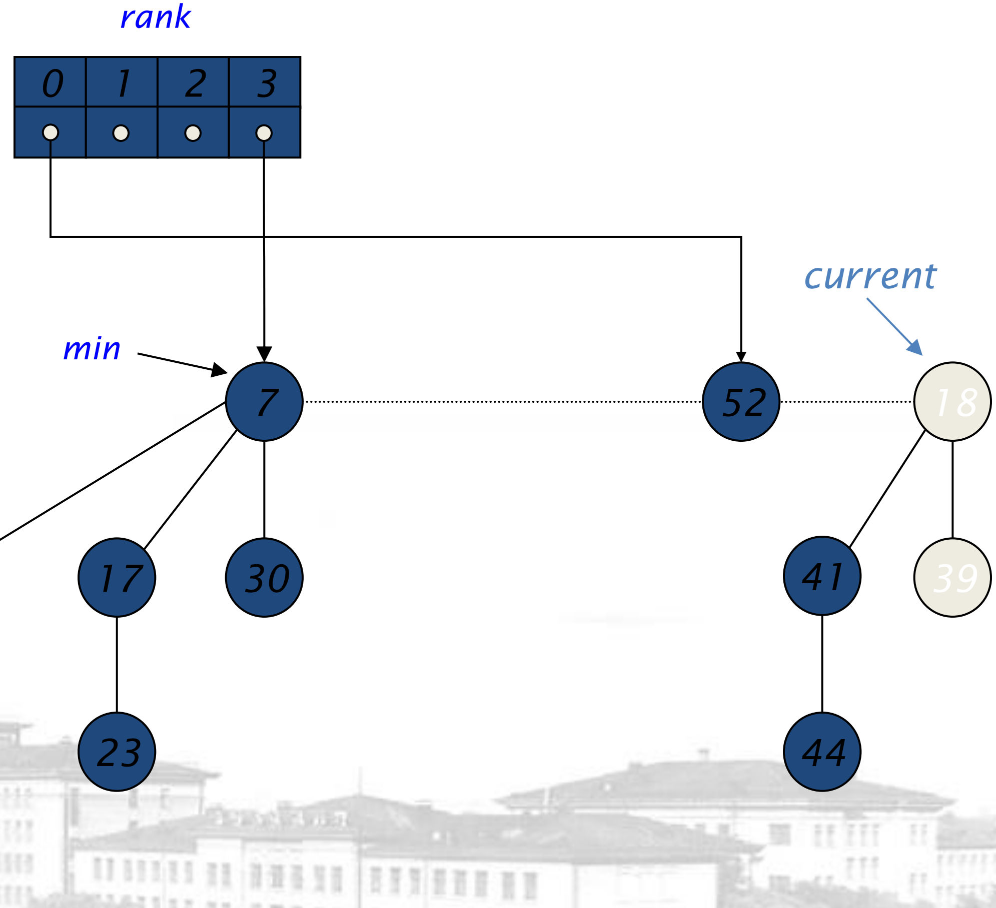
斐波那契堆：Delete Min

- Delete min.
  – 删除 min; 将其子节点放到根节点列表中; 更新 min.
  – 合并tree，使得没有两个根节点具有相同的 rank.



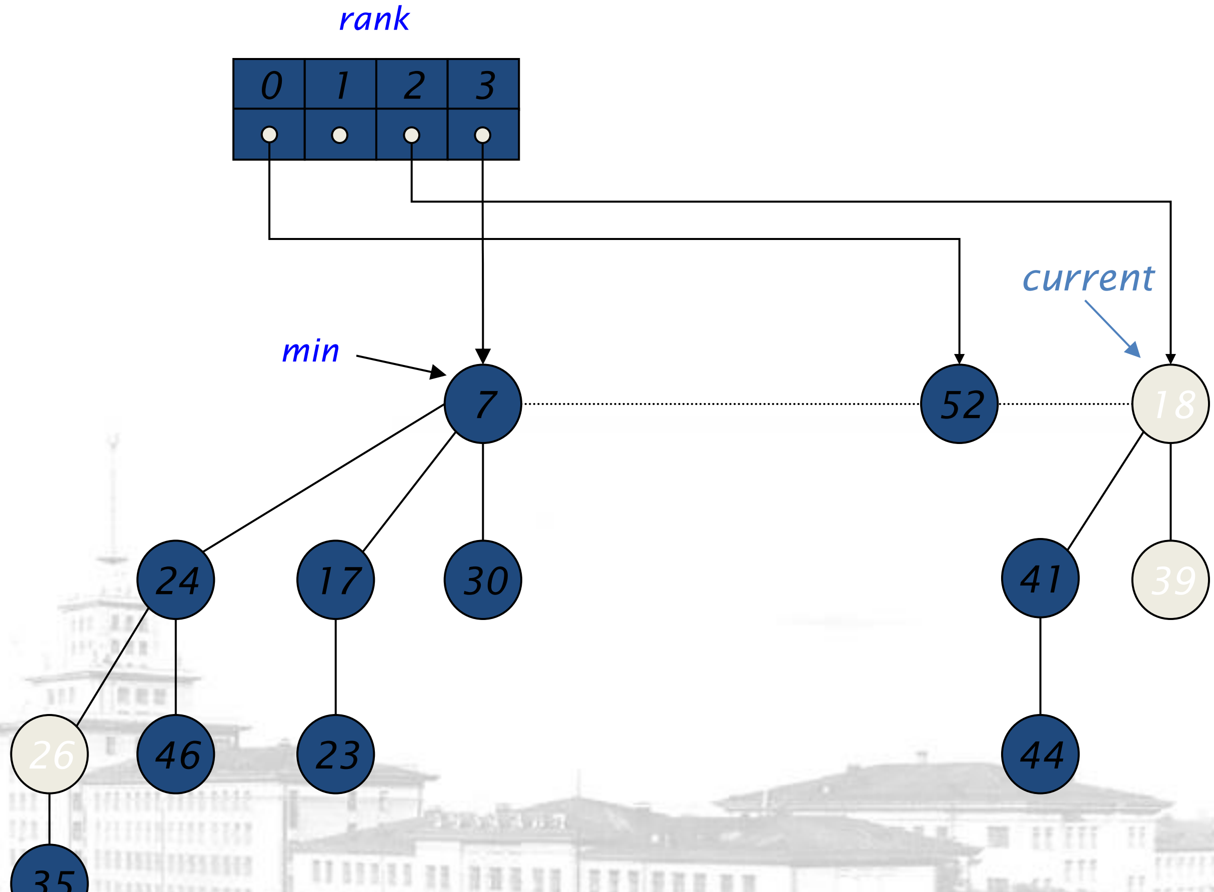
斐波那契堆：Delete Min

- Delete min.
  - 删除 min; 将其子节点放到根节点列表中; 更新 min.
  - 合并tree，使得没有两个根节点具有相同的rank.



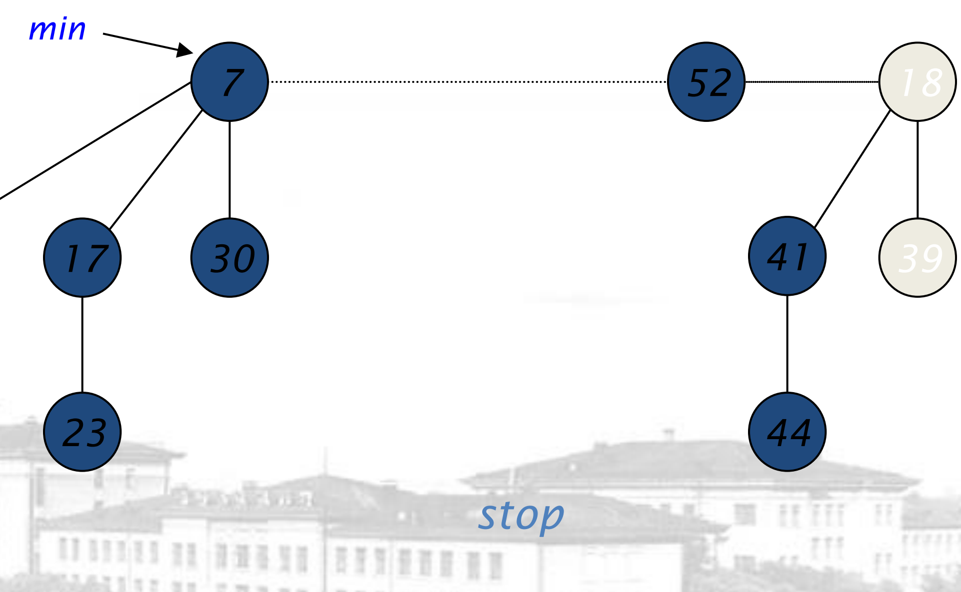
斐波那契堆：Delete Min

- Delete min.
  - 删除 min; 将其子节点放到根节点列表中; 更新 min.
  - 合并tree，使得没有两个根节点具有相同的rank.



斐波那契堆：Delete Min

- Delete min.
  - 删除 min; 将其子节点放到根节点列表中; 更新 min.
  - 合并tree，使得没有两个根节点具有相同的rank.



斐波那契堆：Delete Min 的分析

$$\Phi(H) = trees(H) + 2 \cdot marks(H)$$

势函数

- Delete min.

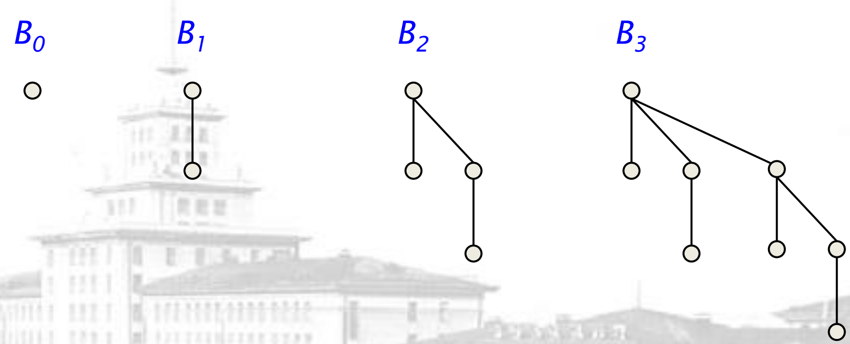
- 实际代价. $O(rank(H)) + O(trees(H))$
  - $O(rank(H))$ 用于将min的子节点放大到根节点列表中.
  - $O(rank(H)) + O(trees(H))$ 用于更新min.
  - $O(rank(H)) + O(trees(H))$ 用于合并trees.

斐波那契堆：Delete Min 的分析

- Q. 平摊代价为O(*rank*(*H*))，这是好事吗?

- A. 是的, 如果只有 *insert* and *delete-min* 操作.
  - 在这种情况下，所有的树都是二项树.
  - 这说明 *rank*(*H*) ≤ lg *n*.

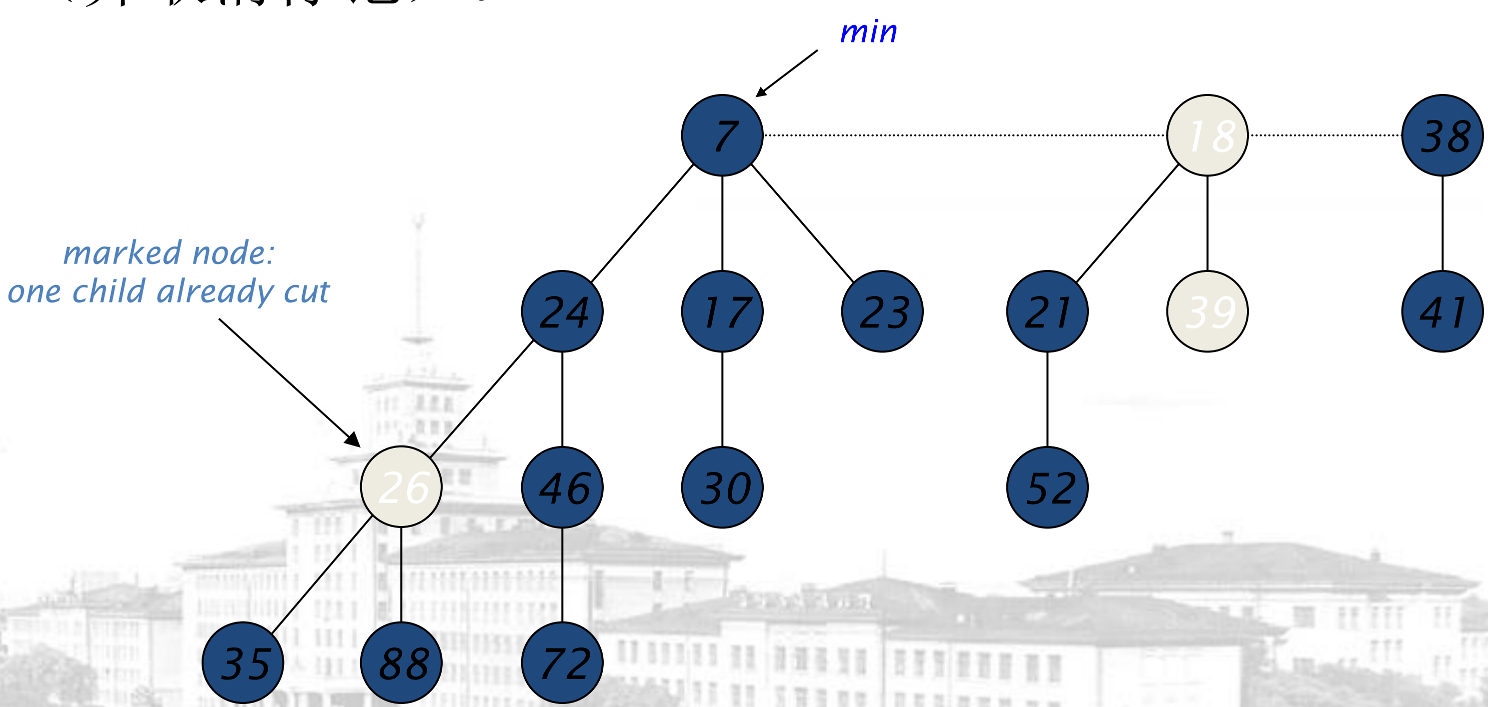
我们只链接那些有相同*rank*的树



Decrease Key

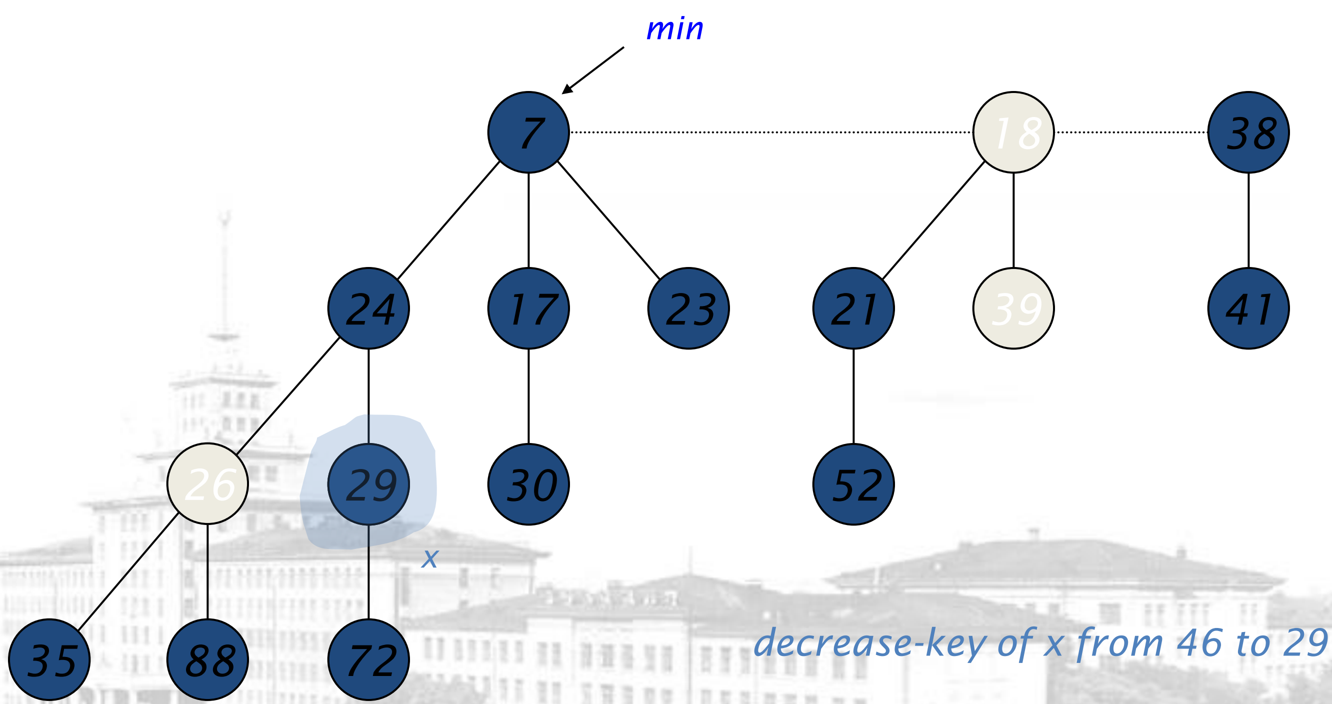
斐波那契堆: *Decrease Key*

- 直观上来看，删除节点*x*的key.
  - 如果不违背堆的顺序，只需要减小*x*的key.
  - 除此之外，需要将*x*为根节点的树剪下来（cut），然后加入根节点列表中.
  - 为了保证树的flat：只要一个节点中执行了两次子节点的裁剪（cut），就将这个节点剪下来，然后加入到根节点列表中（并取消标记）。



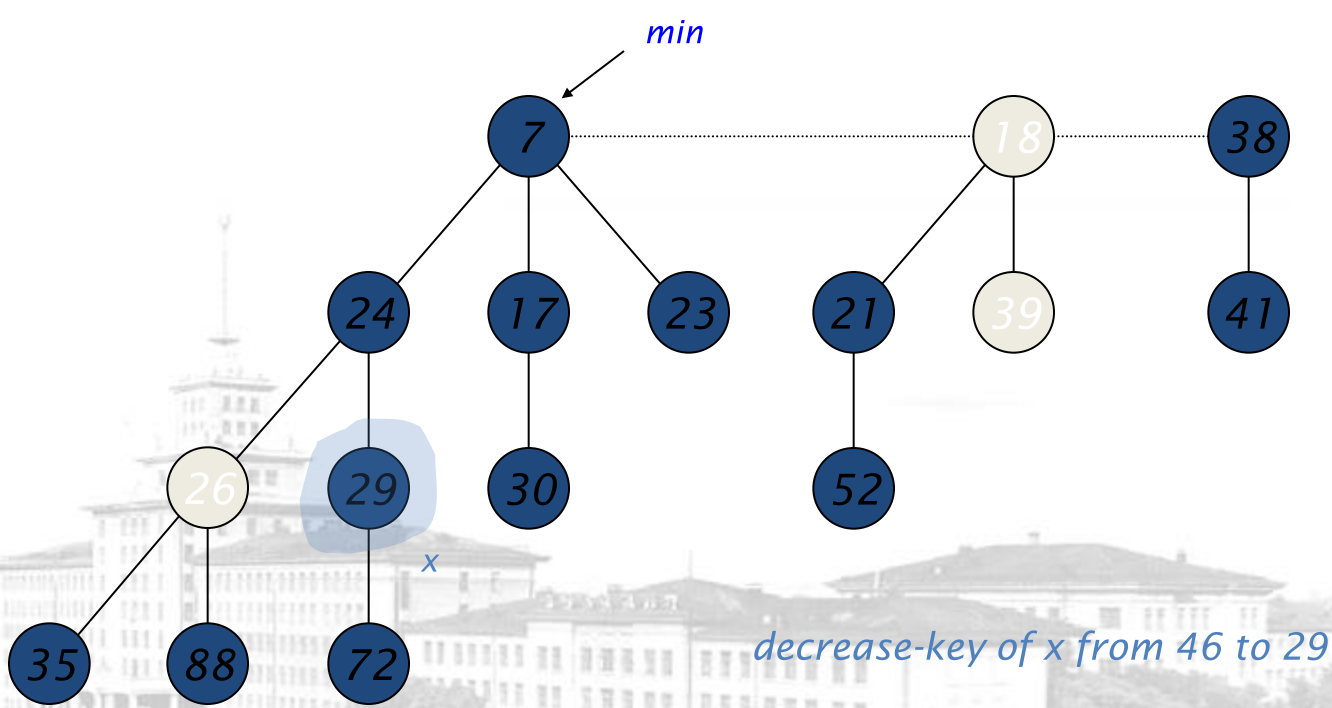
斐波那契堆: *Decrease Key*

- 情况 1. [没有违背堆的顺序]
  - 减小*x*的key.
  - 改变堆中min指向的指针 (如果有必要的话).



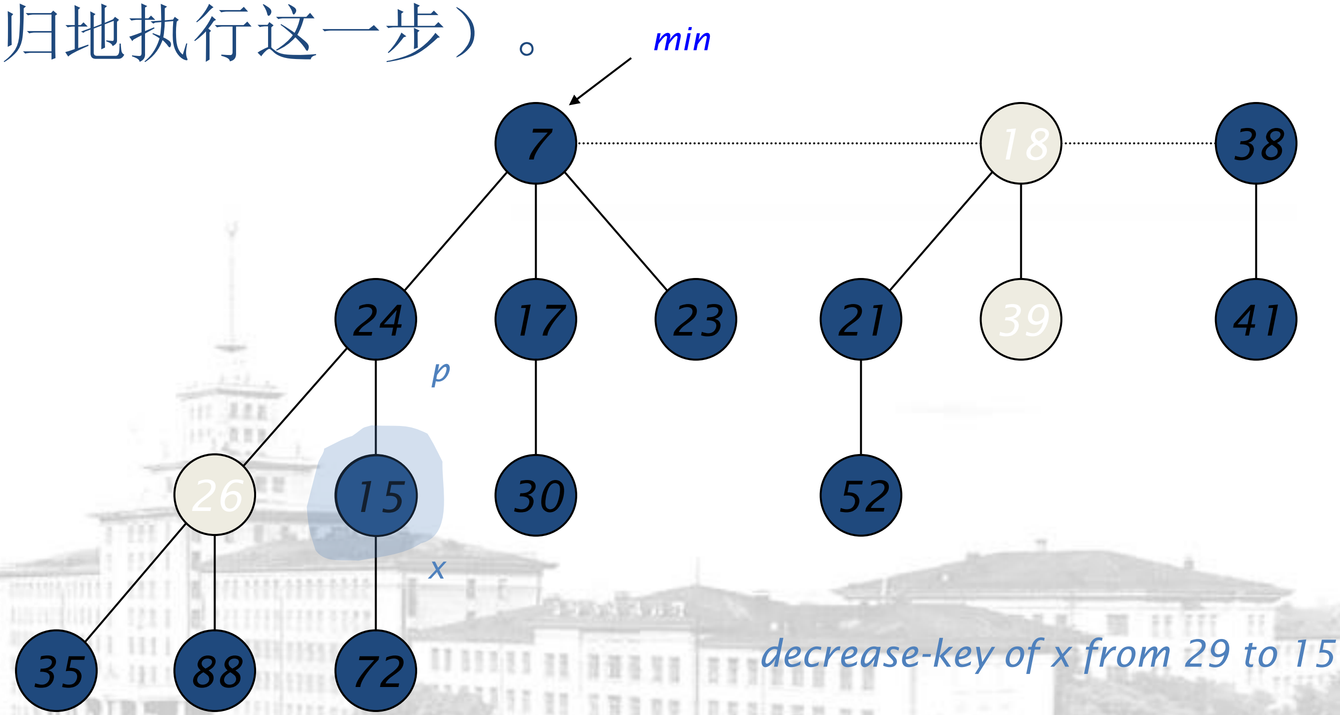
斐波那契堆: *Decrease Key*

- 情况 2. [没有违背堆的顺序]
  - 减小x的key.
  - 改变堆中min指向的指针 (如果有必要的话).



斐波那契堆: Decrease Key

- 情况 2a. [违背了堆的顺序]
  - 减小*x* 的*key*.
  - 将根节点为*x*的树剪下来，然后添加到根节点列表中并取消标记.
  - 如果*x* 的父节点*p* 是未标记的（从未失去过一个子节点），就标记它；否则，将*p* 剪下来，添加到子节点列表中，并取消标记（对于所有失去了第二个子节点的祖先，递归地执行这一步）。

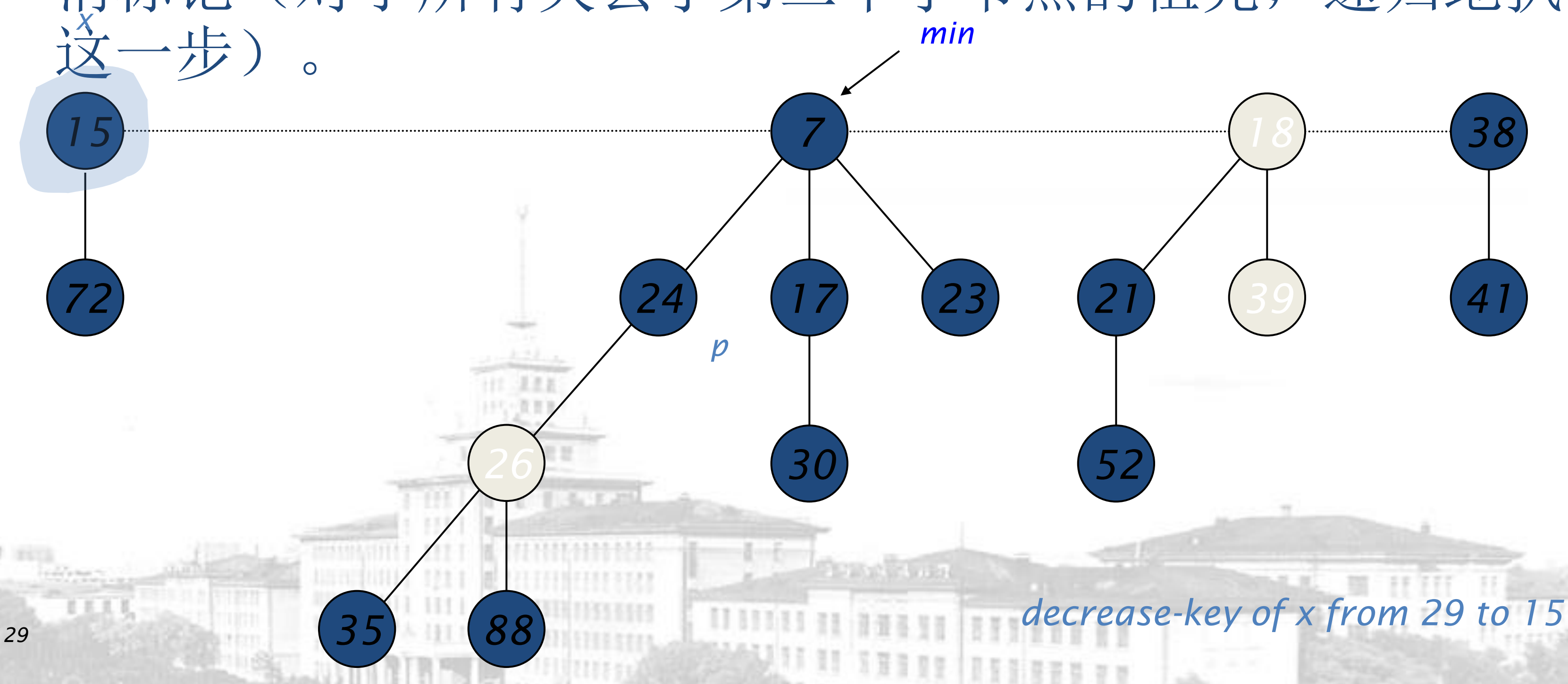


- 情况 2a. [违背了堆的顺序]
  - *减小x 的key.*
  - 将根节点为x的树剪下来，然后添加到根节点列表中并取消标记.
  - 如果*x* 的父节点*p* 是未标记的（从未失去过一个子节点），就标记它；否则，将*p* 剪下来，添加到子节点列表中，并取消标记（对于所有失去了第二个子节点的祖先，递归地执行这一步）。

decrease-key of x from 29 to 15

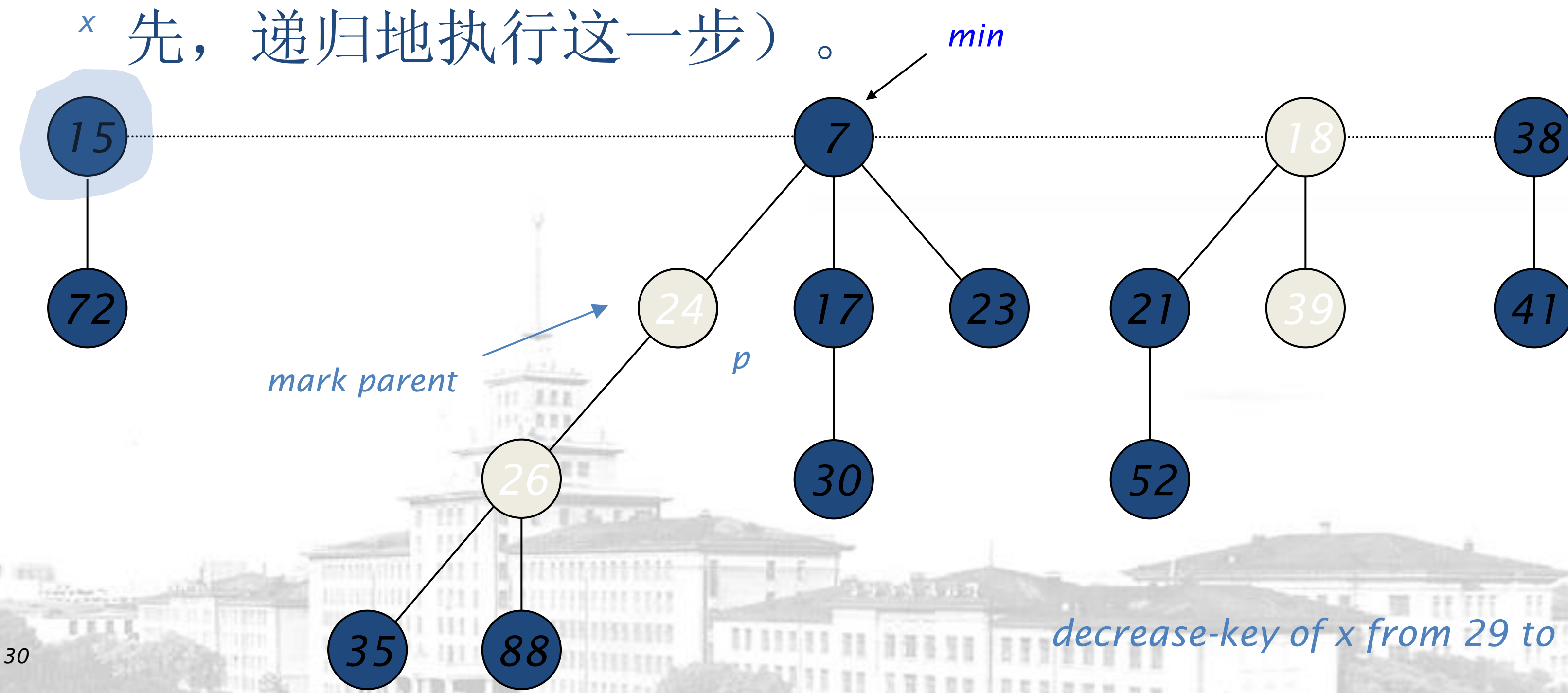
斐波那契堆: Decrease Key

- 情况 2a. [违背了堆的顺序]
  - *减小x 的key.*
  - 将根节点为x的树剪下来，然后添加到根节点列表中并取消标记.
  - 如果*x* 的父节点*p* 是未标记的（从未失去过一个子节点），就标记它；否则，将*p* 剪下来，添加到子节点列表中，并取消标记（对于所有失去了第二个子节点的祖先，递归地执行这一步）。



斐波那契堆: *Decrease Key*

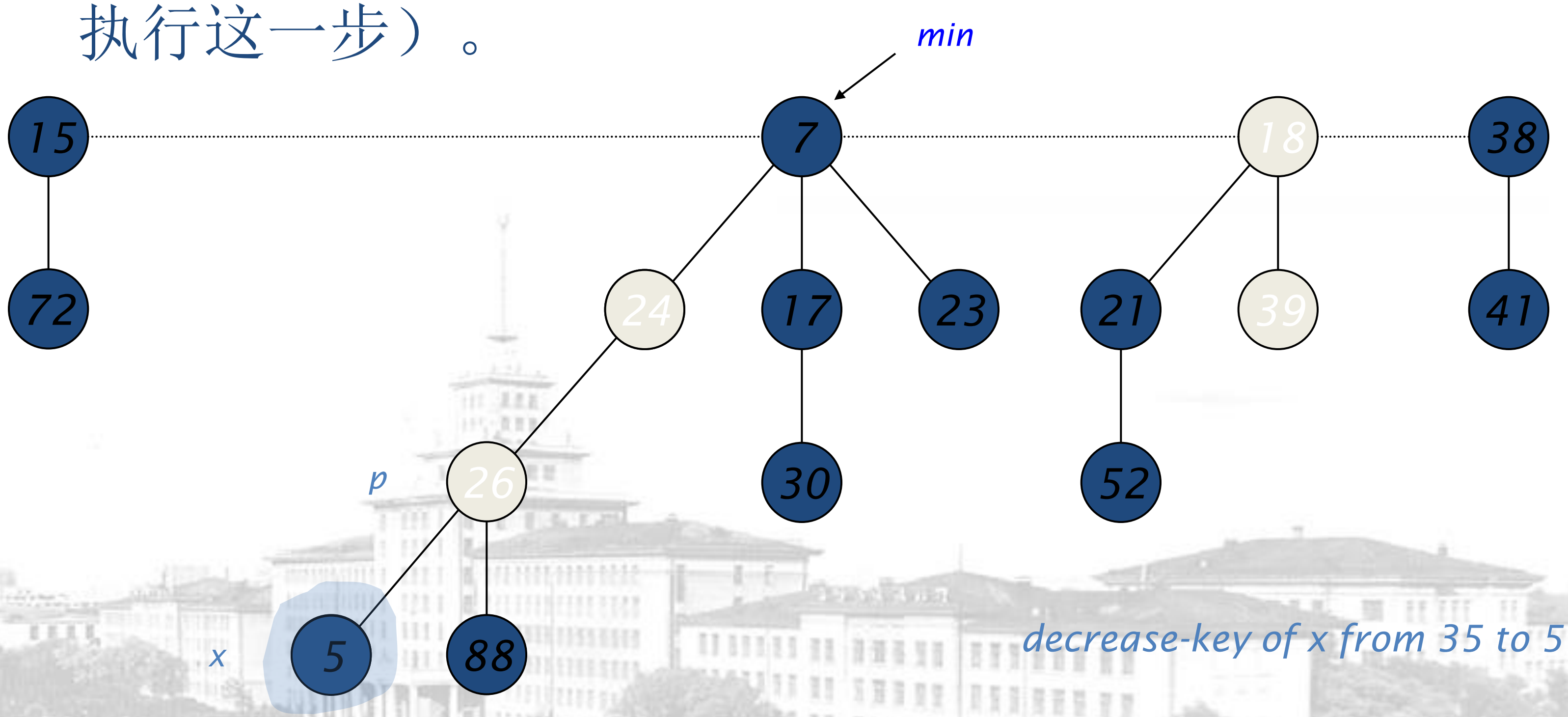
- 情况 2a. [违背了堆的顺序]
  - *减小x 的key.*
  - 将根节点为*x*的树剪下来，然后添加到根节点列表中并取消标记.
  - 如果x 的父节点p 是未标记的（从未失去过一个子节点），就标记它；否则，将*p* 剪下来，添加到子节点列表中，并取消标记（对于所有失去了第二个子节点的祖先，递归地执行这一步）。



decrease-key of x from 29 to 15

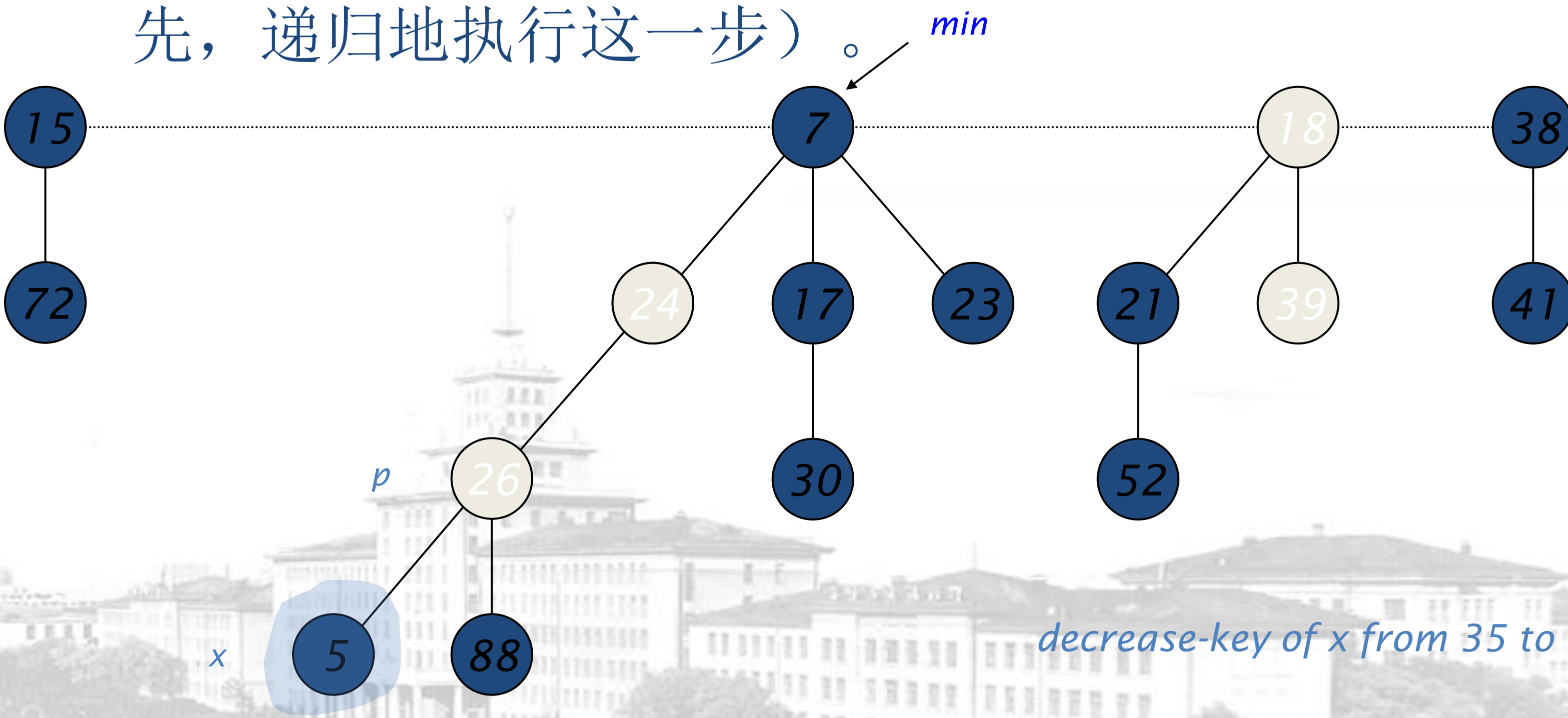
斐波那契堆: Decrease Key

- 情况 2b. [违背了堆的顺序]
  - 减小x 的key.
  - 将根节点为*x*的树剪下来，然后添加到根节点列表中并取消标记.
  - 如果x 的父节点p 是未标记的（从未失去过一个子节点），就标记它；否则，将*p* 剪下来，添加到子节点列表中，并取消标记（对于所有失去了第二个子节点的祖先，递归地执行这一步）。



斐波那契堆: *Decrease Key*

- 情况 2b. [违背了堆的顺序]
  - 减小x 的key.
  - 将根节点为x的树剪下来，然后添加到根节点列表中并取消标记.
  - 如果x 的父节点p 是未标记的（从未失去过一个子节点），就标记它；否则，将*p* 剪下来，添加到子节点列表中，并取消标记（对于所有失去了第二个子节点的祖先，递归地执行这一步）。

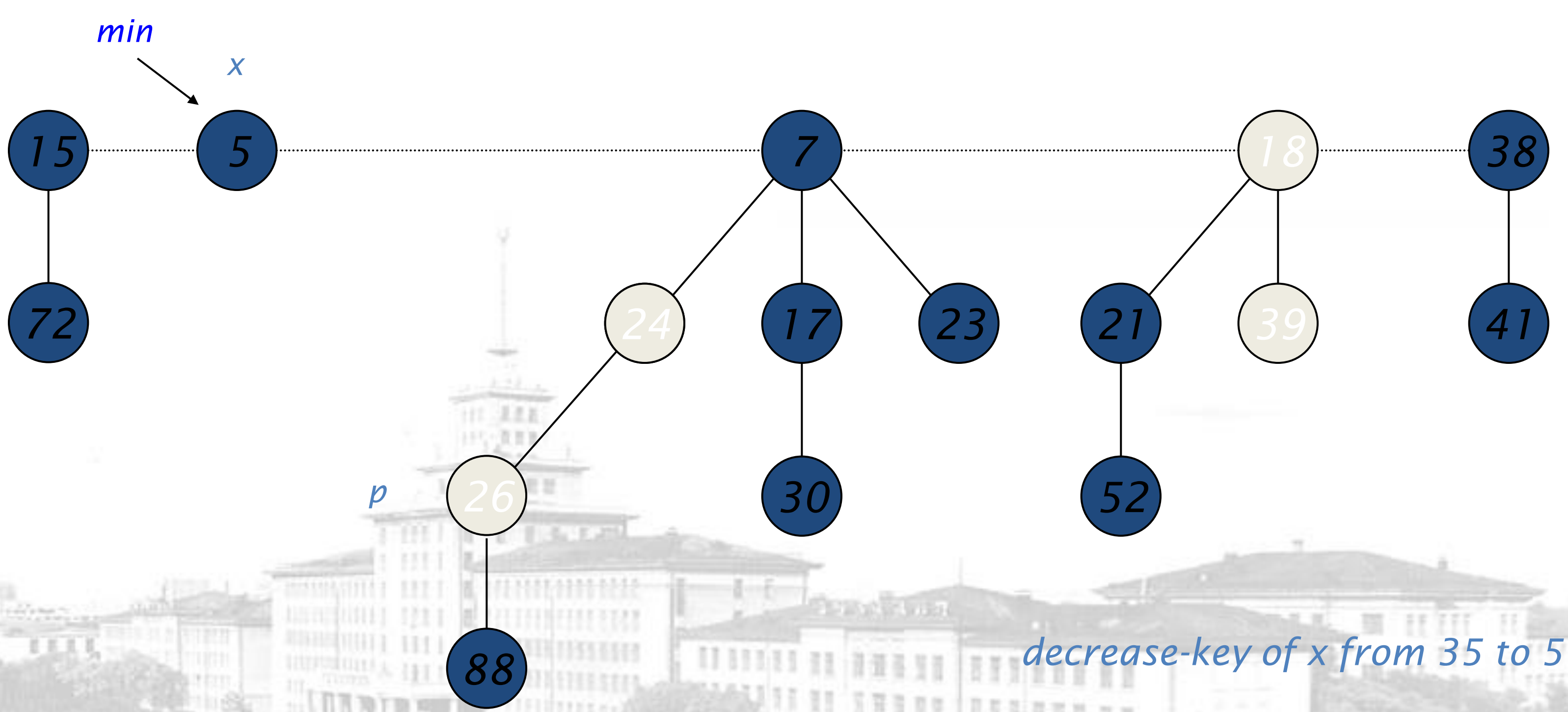


decrease-key of x from 35 to 5

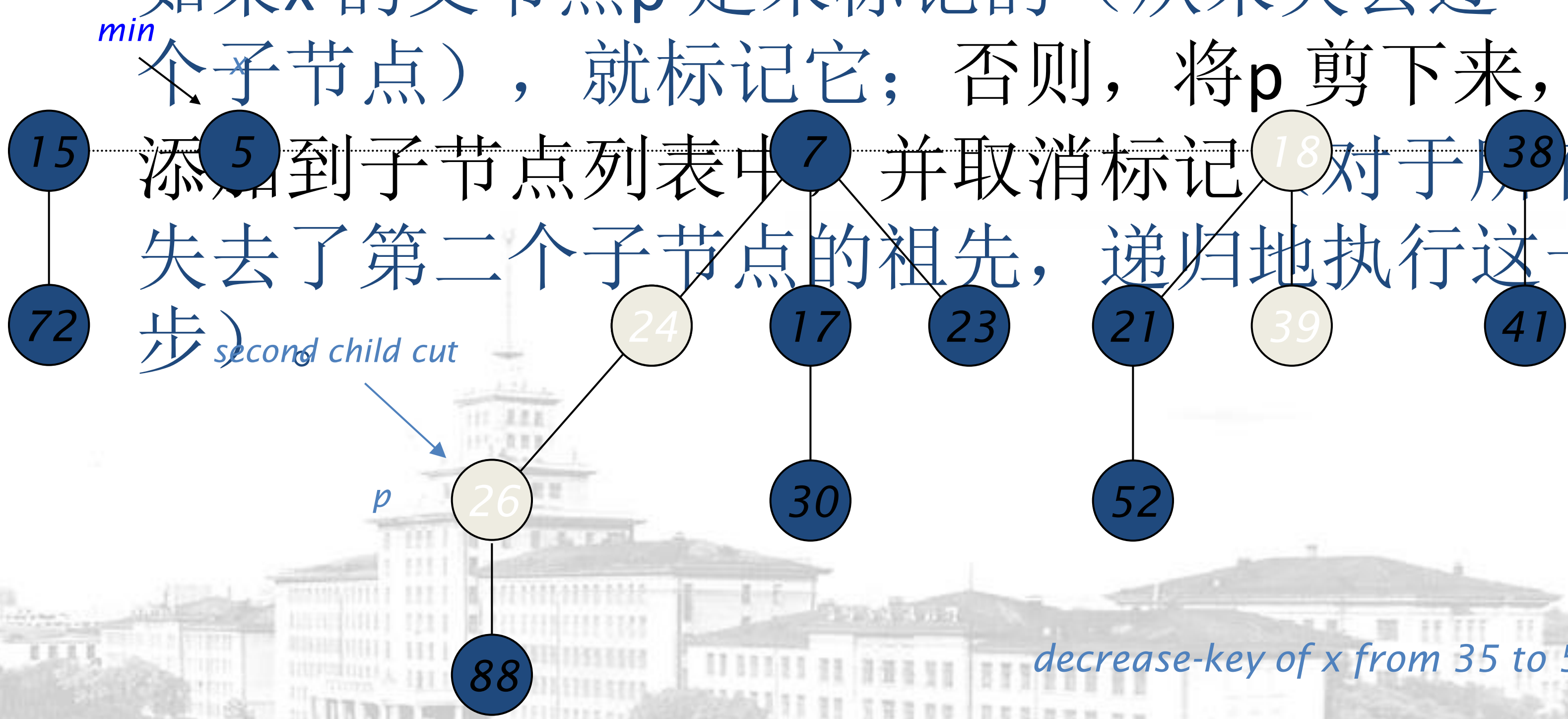
斐波那契堆: *Decrease Key*

情况 2b. [违背了堆的顺序]

– 减小x 的key.
– 将根节点为x的树剪下来，然后添加到根节点列表中并取消标记.
– 如果x 的父节点p 是未标记的（从未失去过一个子节点），就标记它；否则，将*p* 剪下来，添加到子节点列表中，并取消标记（对于所有失去了第二个子节点的祖先，递归地执行这一步）。

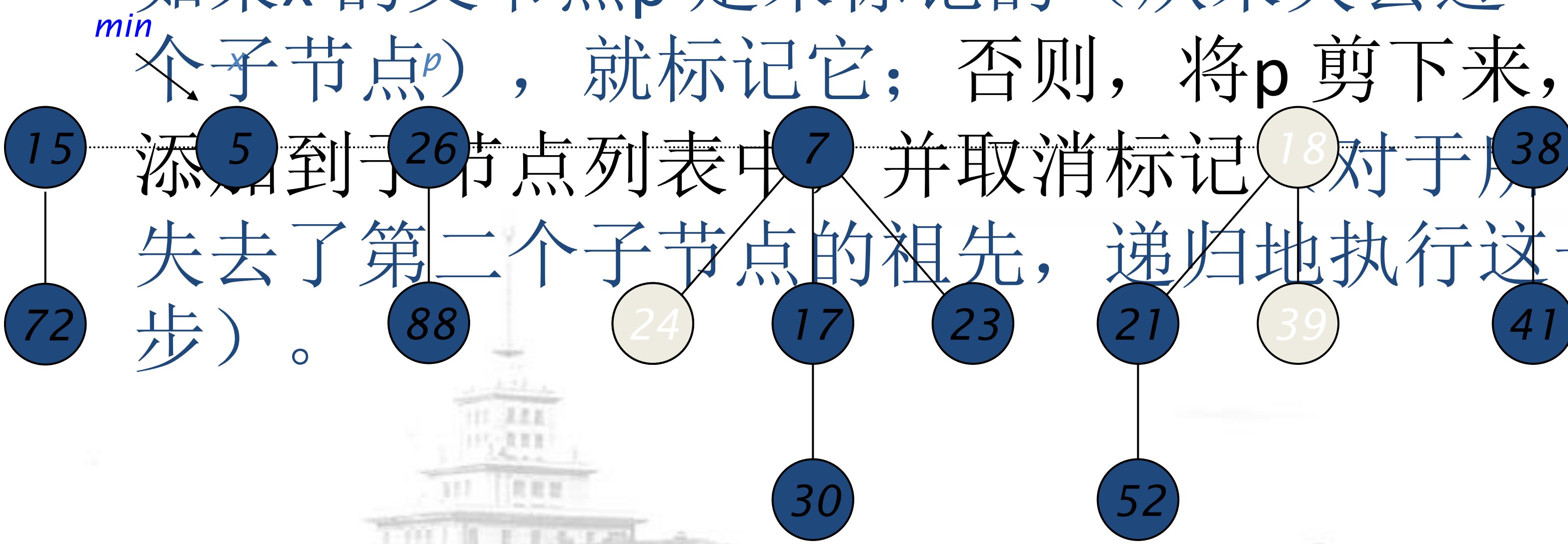


- 情况 2b. [违背了堆的顺序]
  - 减小x 的key.
  - 将根节点为*x*的树剪下来，然后添加到根节点列表中并取消标记.
  - 如果x 的父节点p 是未标记的（从未失去过一个子节点），就标记它；否则，将p 剪下来，添加到子节点列表中，并取消标记（对于所有失去了第二个子节点的祖先，递归地执行这一步）.



斐波那契堆: *Decrease Key*

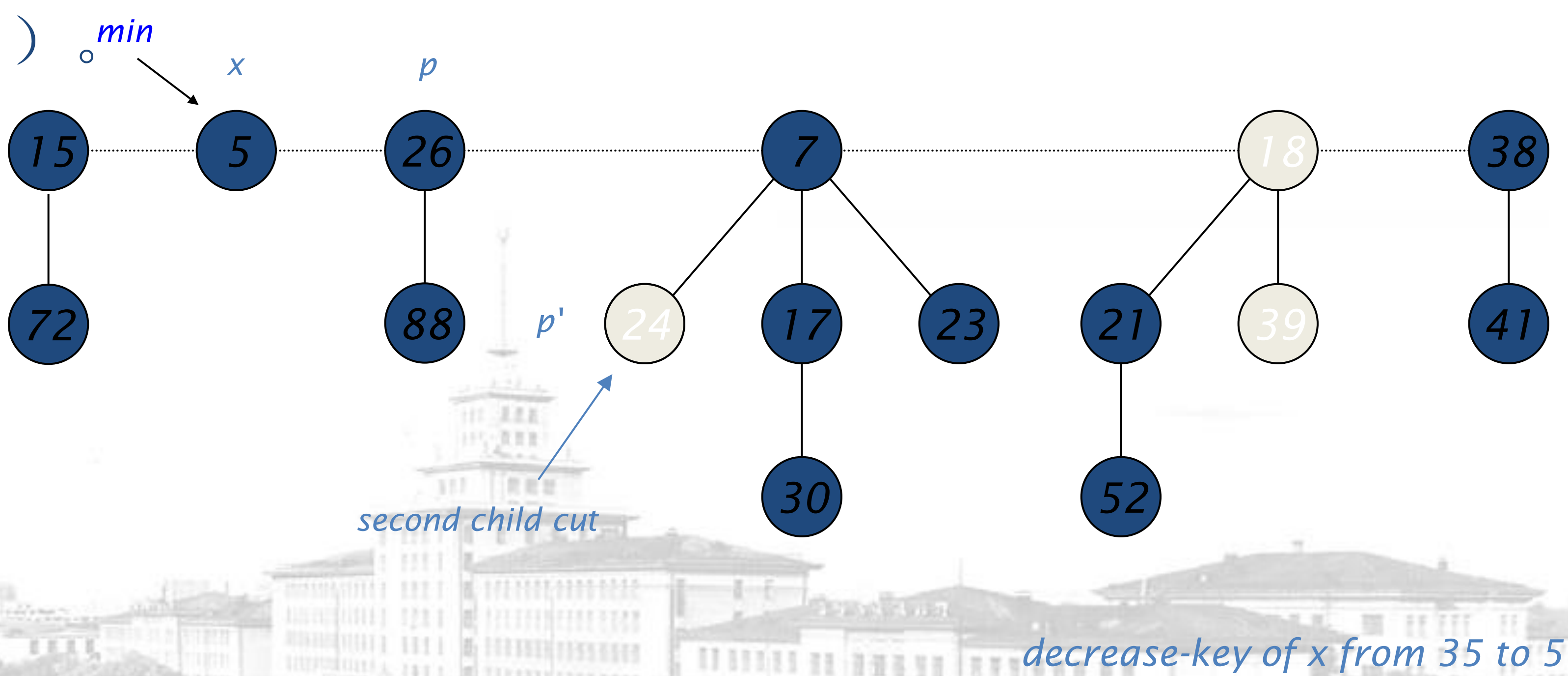
- 情况 2b. [违背了堆的顺序]
  - – 减小x 的key.
  - – 将根节点为*x*的树剪下来，然后添加到根节点列表中并取消标记.
  - – 如果x 的父节点p 是未标记的（从未失去过一个子节点），就标记它；否则，将p 剪下来，添加到根节点列表中，并取消标记（对于所有失去了第二个子节点的祖先，递归地执行这一步）。



decrease-key of x from 35 to 5

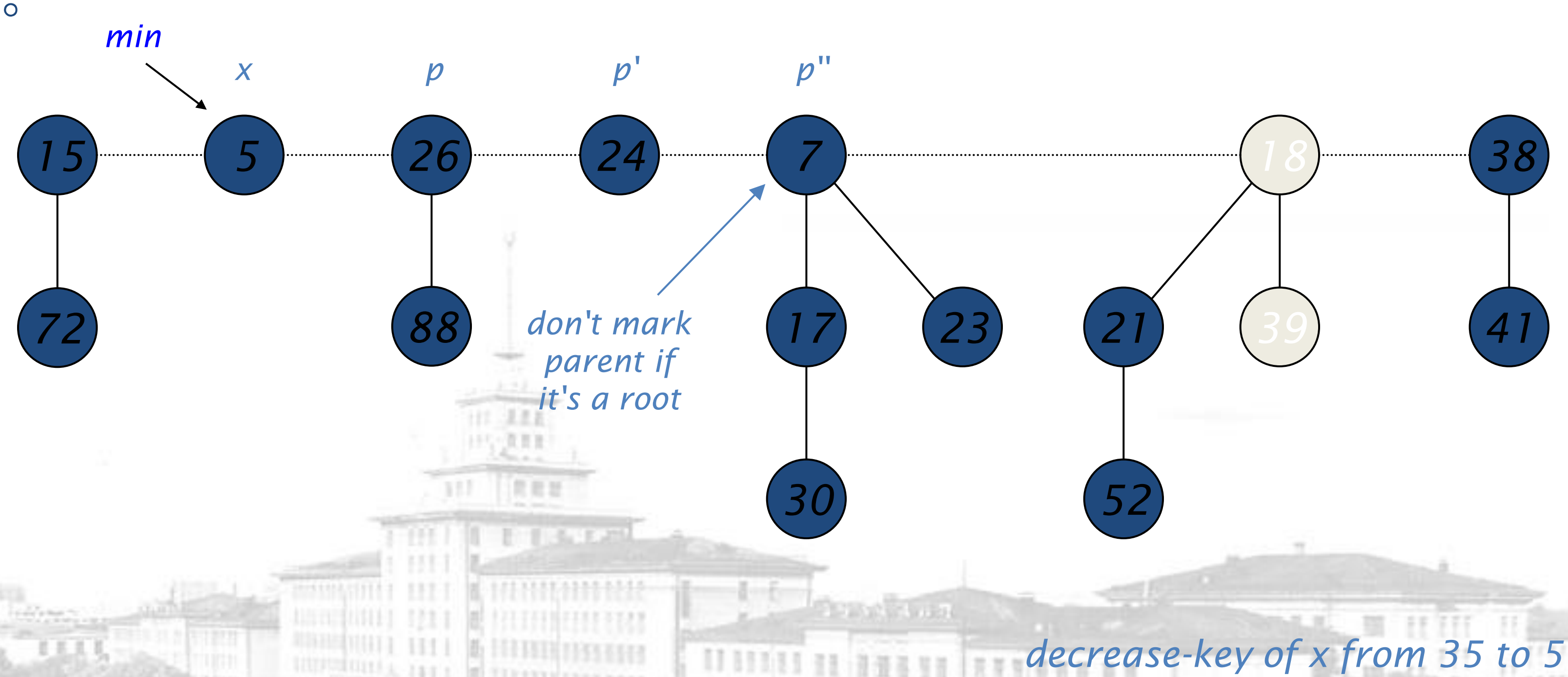
斐波那契堆: *Decrease Key*

- 情况 2b. [违背了堆的顺序]
  - 减小x 的key.
  - 将根节点为x的树剪下来，然后添加到根节点列表中并取消标记.
  - 如果x 的父节点p 是未标记的（从未失去过一个子节点），就标记它；否则，将p 剪下来，添加到子节点列表中，并取消标记（对于所有失去了第二个子节点的祖先，递归地执行这一步）。



斐波那契堆: Decrease Key

- 情况 2b. [违背了堆的顺序]
  - 减小x 的key.
  - 将根节点为x的树剪下来，然后添加到根节点列表中并取消标记
  - 如果x 的父节点p 是未标记的（从未失去过一个子节点），就标记它；否则，将p 剪下来，添加到子节点列表中，并取消标记（对于所有失去了第二个子节点的祖先，递归地执行这一步）。



斐波那契堆: *Decrease Key* 的分析

- Decrease-key.

$$\Phi(H) = trees(H) + 2 \cdot marks(H)$$

势函数

- 实际代价. O(c)
  - O(1) 的时间用于改变 key.
  - c个裁剪中每个需要花费O(1), 这包括 "加入根节点列表" 这一个操作.

- 势的改变. O(1) - c
  - $trees(H') = trees(H) + c.$

分析

分析汇总

- *Insert.* O(1)
- *Delete-min.* O(*rank*(*H*)) †
- *Decrease-key.* O(1) †

† 平摊

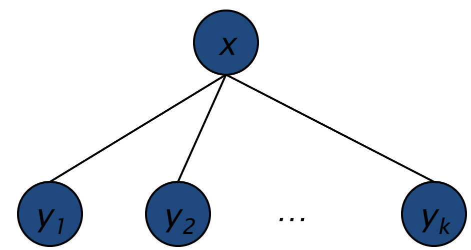
- 关键引理. *rank*(*H*) = O(log *n*).

节点的数量是*rank*的指数级别

斐波那契堆: *Bounding the Rank*

- 引理. Fix a point in time.,设x是一个节点，并设y_1， …， y_k表示x的子节点，它们是按与x链接的顺序排列的。 于是有

$$rank\ (y_i) \ \geq \begin{cases} 0 & \text{if} \quad i = 1 \\ i - 2 & \text{if} \quad i \geq 1 \end{cases}$$

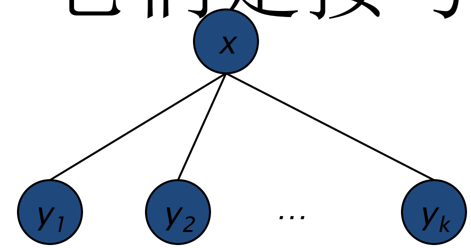


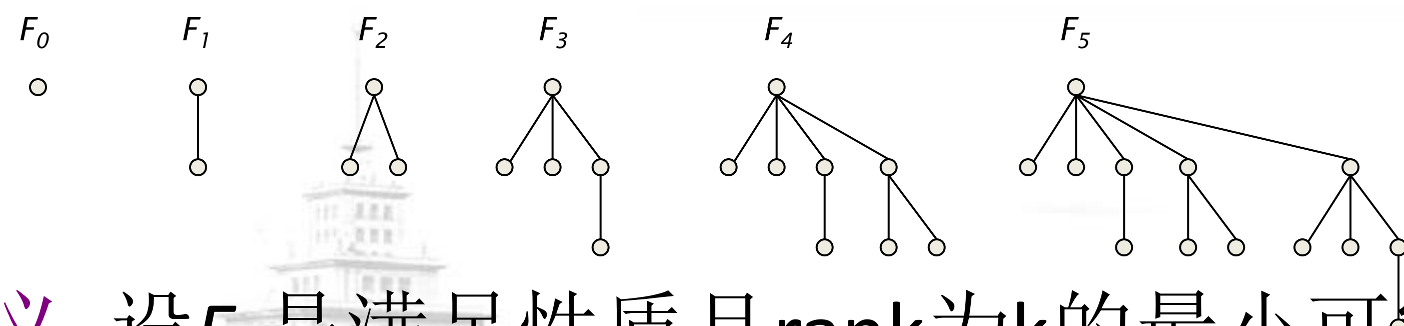
- 证明.
  - 当 y_i 被链接到*x*时, *x* 至少有 *i* -1 子节点，分别为 $y_1, \ldots, y_{i-1}$.
  - 因为这个时候，只有那些具有相等rank的树会被链接到一起，所以$rank(y_i) = rank(x_i) \geq i - 1$.
  - 从那以后, y_i 有至多失去一个子节点.
  - 因此, 现在有 $rank(y_i) \geq i - 2$.

或者 y_i 已经被裁剪

- 引理. Fix a point in time.,设x是一个节点，并设y_1，…，y_k表示x的子节点，它们是按与x链接 于是有

$$rank\ (y_i) \geq \begin{cases} 0 & \text{if} \quad i = 1 \\ i-2 & \text{if} \quad i \geq 1 \end{cases}$$



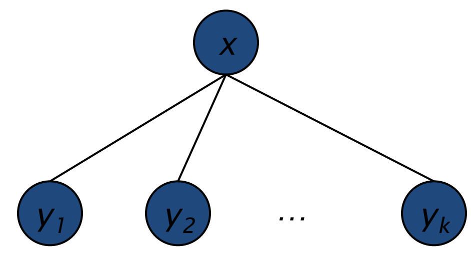


- 定义. 设F_k是满足性质且rank为k的最小可能树。

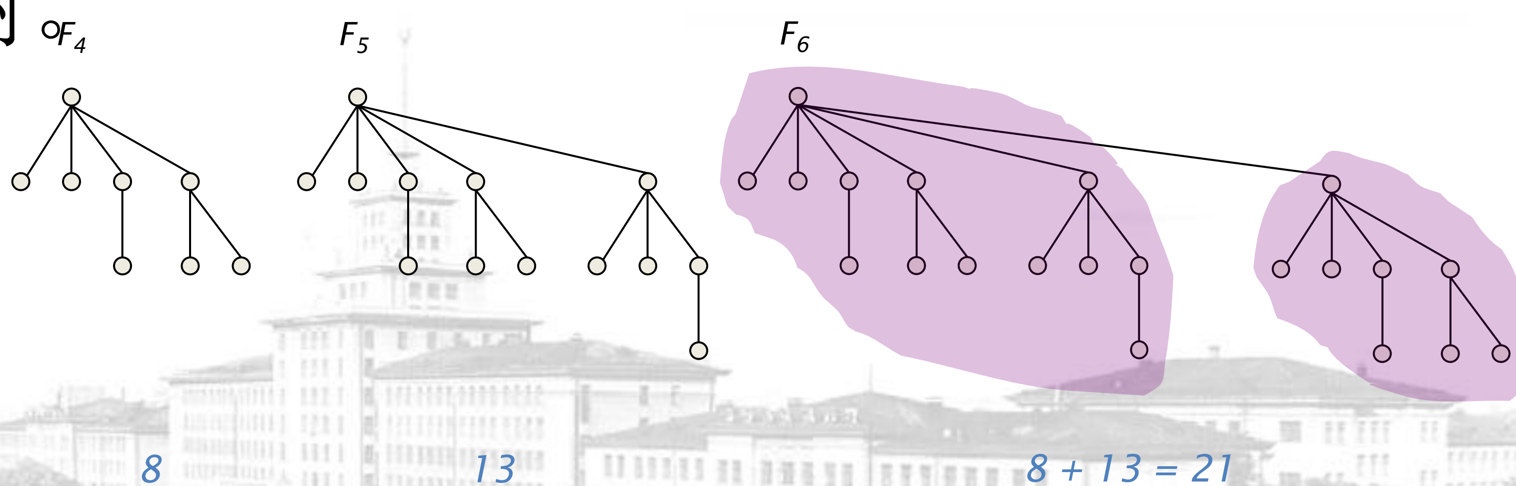
斐波那契堆: *Bounding the Rank*

- 引理. Fix a point in time.,设x是一个节点，并设y_1， …， y_k表示x的子节点，它们是按与x链接的顺序排列的。 于是有

$$rank\ (y_i) \geq \begin{cases} 0 & \text{if } \ i = 1 \\ i-2 & \text{if } \ i \geq 1 \end{cases}$$



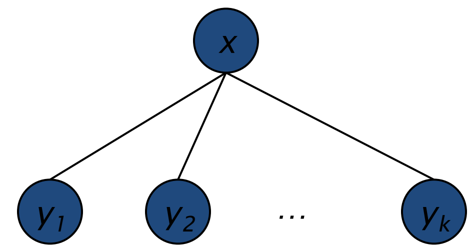
- 定义. 设F_k是满足性质且rank为k的最小可能树



斐波那契堆: *Bounding the Rank*

- 引理. Fix a point in time.,设x是一个节点，并设y_1，…，y_k表示x的子节点，它们是按与x链接的顺序排列的。 于是有

$$rank\ (y_i)\ \geq \begin{cases} 0 & \text{if}\ \ i=1 \\ i-2 & \text{if}\ \ i\geq 1 \end{cases}$$



- 定义. 设F_k是满足性质且rank为k的最小可能树。
- Fibonacci fact. $F_k \geq \phi^k$, 其中 $\phi = (1 + \sqrt{5}) / 2 \approx 1.618$.

黄金比例

- 推论. $rank(H) \leq \log_\phi n$.

斐波那契数

斐波那契数：指数增长

- 定义. 斐波那契序列: 1, 2, 3, 5, 8, 13, 21, …

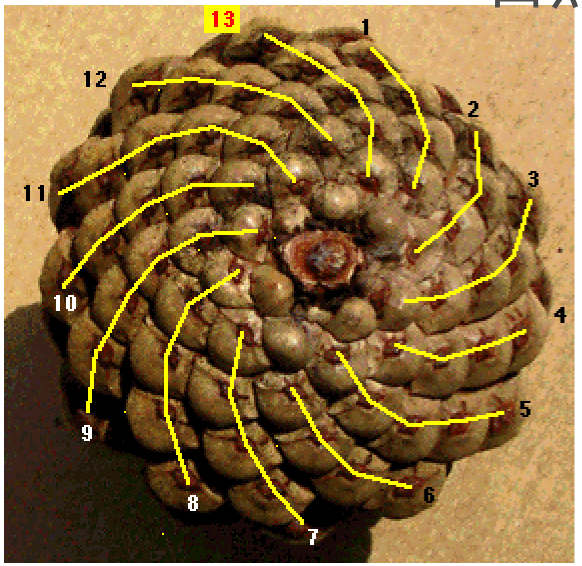
$$F_k = \begin{cases} 1 & \text{if } k = 0 \\ 2 & \text{if } k = 1 \\ F_{k-1} + F_{k-2} & \text{if } k \geq 2 \end{cases}$$

略有不规范的定义

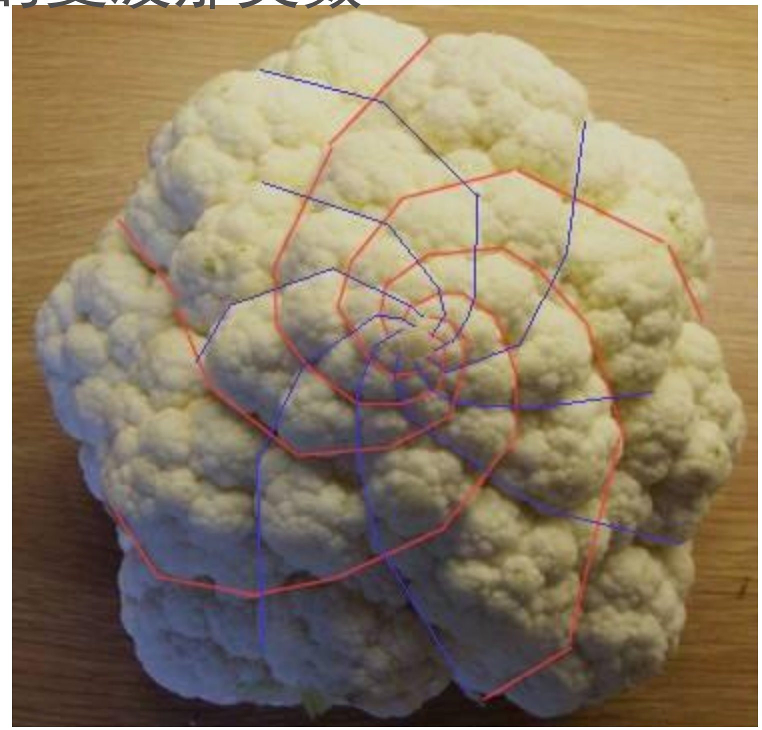
$$\begin{aligned} F_{k+2} &= F_k + F_{k+1} \\ &\geq \phi^k + \phi^{k+1} \\ &= \phi^k (1 + \phi) \\ &= \phi^k (\phi^2) \\ &= \phi^{k+2} \end{aligned}$$

- 引理. $F_k \geq \phi^k$, 其中 $\phi = (1 + \sqrt{5}) / 2 \approx 1.618$.
- 证明. [对k进行归纳]
  - 基本情况: $F_0 = 1 \geq 1$, $F_1 = 2 \geq \phi$.
  - 归纳假设: $F_k \geq \phi^k$ 且 $F_{k+1} \geq \phi^{k+1}$

自然界中的斐波那契数



松果

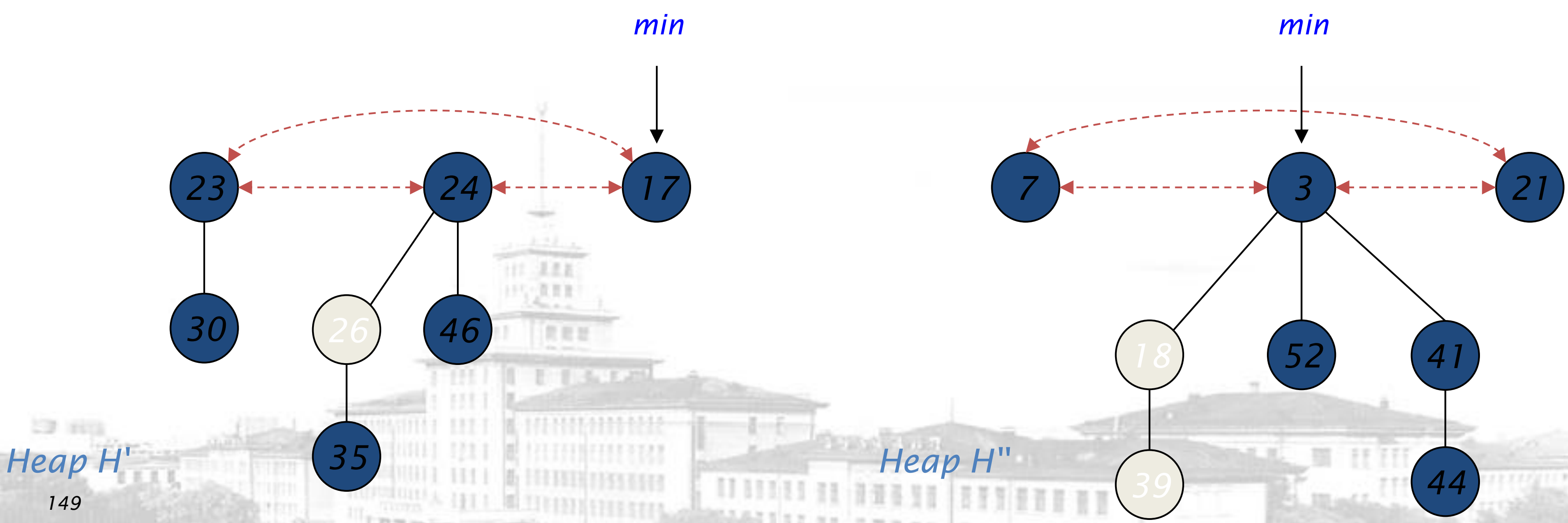


花椰菜

Union

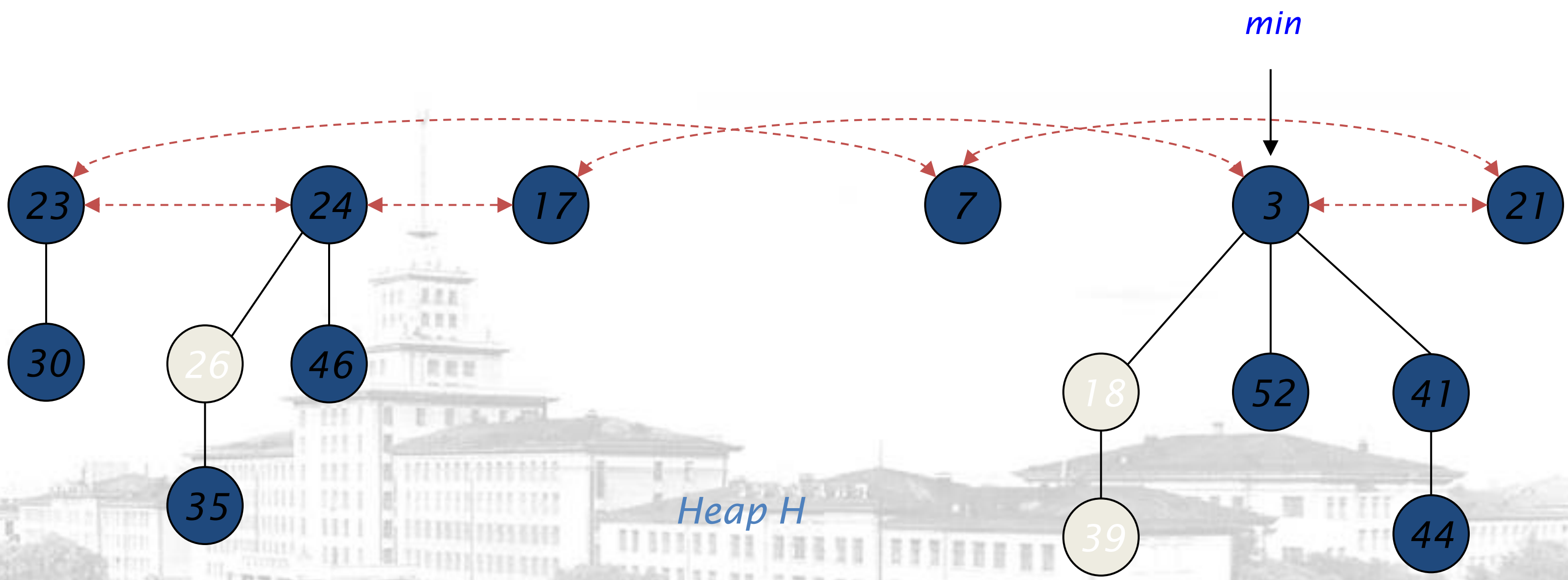
斐波那契堆：*Union*

- Union. 合并两个斐波那契堆。
- Representation. 根节点列表是循环的，也就是双向链表.



斐波那契堆：*Union*

- Union. 合并两个斐波那契堆。
- Representation. 根节点列表是循环的，也就是双向链表.



斐波那契堆：*Union*

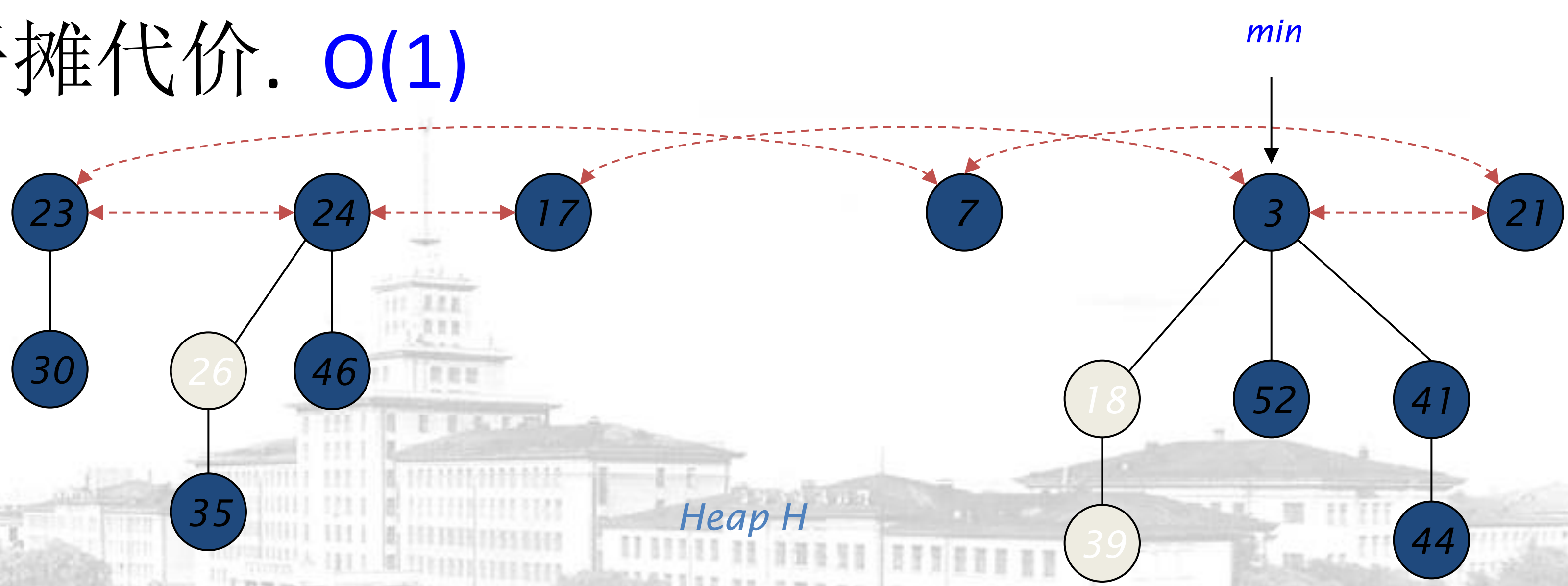
$$\Phi(H) = trees(H) + 2 \cdot marks(H)$$

势函数

- 实际代价. O(1)

- Change in potential. 0

- 平摊代价. O(1)



Delete

斐波那契堆：Delete

$$\Phi(H) = trees(H) + 2 \cdot marks(H)$$

势函数

- 删除节点*x*.
  - 将*x*的*key*减小为-∞.
  - 在堆中删除*min*元素.

- 平摊代价. O(*rank*(*H*))
  - *decrease-key*的平摊代价是O(1)
  - *delete-min*的平摊代价是O(*rank*(*H*))

优点队列的性能汇总

操作	Linked List	Binary Heap	Binomial Heap	Fibonacci Heap †	Relaxed Heap
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 是优先队列中的元素数量

† 平摊后