

优先级队列

概述：动机与接口

12-A7

I cannot choose the best.

The best chooses me.

邓俊辉

deng@tsinghua.edu.cn

循优先级访问

❖ 应用举例

- 离散事件模拟
- 操作系统：任务调度/中断处理/MRU/...
- 输入法：词频调整

❖ 极值元素：须反复地、快速地定位

集合组成：可动态变化

元素优先级：可动态变化

❖ 作为底层数据结构所支持的高效操作

是很多高效算法的基础

- 内部、外部、在线排序
- 贪心算法：Huffman编码、Kruskal
- 平面扫描算法中的事件队列
- ...

优先级队列

```
❖ template <typename T> struct PQ { //priority queue  
    virtual void insert( T ) = 0;  
  
    virtual T delMax() = 0;  
  
    virtual T & getMax() = 0;  
};
```

❖ Stack和Queue，都是PQ的特例——优先级完全取决于元素的插入次序

Steap和Queap，也是PQ的特例——插入和删除的位置受限

优先级队列

概述：基本实现

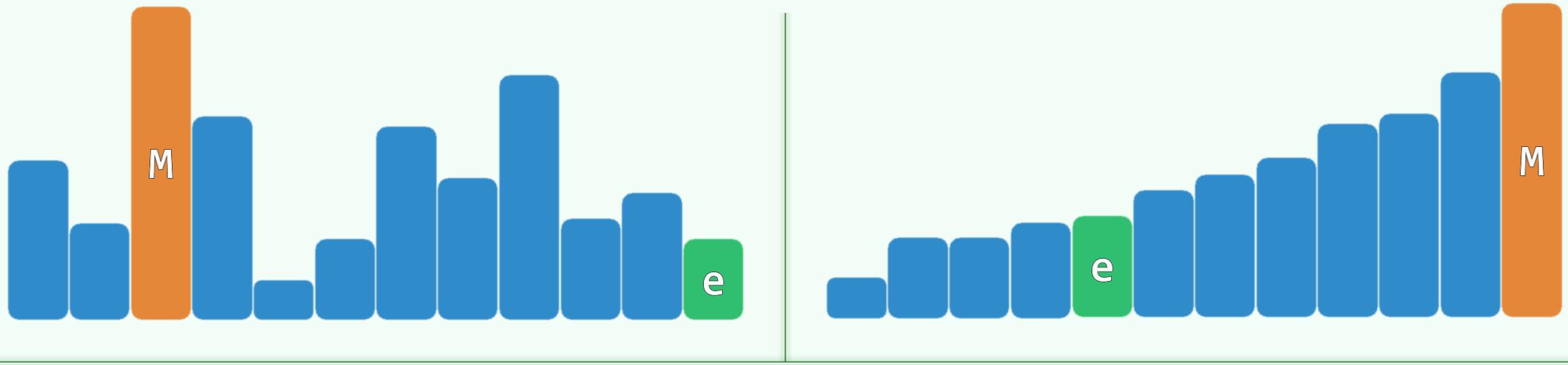
12-A2

邓俊辉

deng@tsinghua.edu.cn

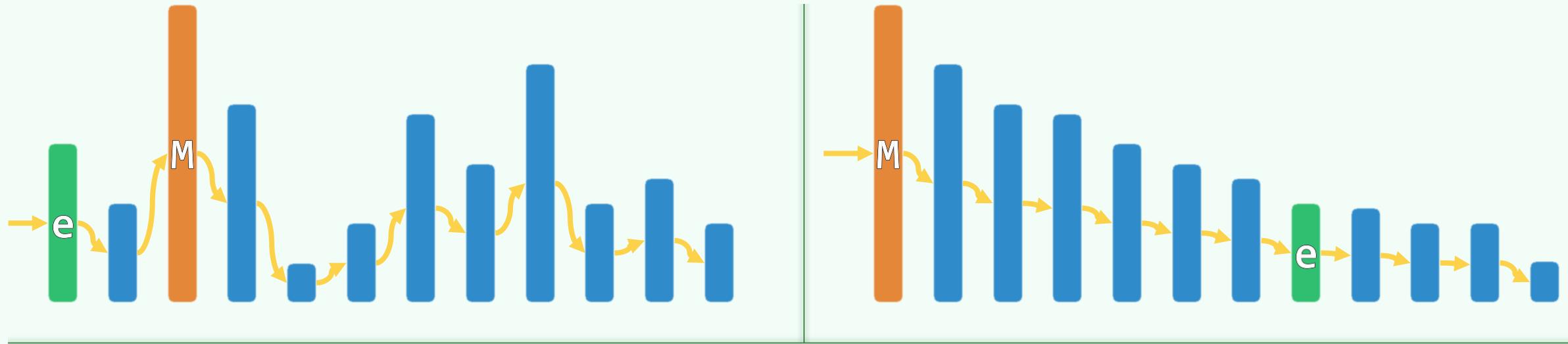
大兒孔文舉，小兒楊德祖；與子碌碌，莫足數也

Vector



Vector	getMax()	delMax()	insert()
Unsorted	traverse() $\Theta(n)$	remove(traverse()) $\Theta(n) + \Theta(n) = \Theta(n)$	insertLast(e) $\Theta(1)$
Sorted	[n-1] $\Theta(1)$	remove(n-1) $\Theta(1)$	insert(1 + search(e), e) $\Theta(\log n) + \Theta(n) = \Theta(n)$

List



List	getMax()	delMax()	insert()
Unsorted	traverse() $\Theta(n)$	remove(traverse()) $\Theta(n) + \Theta(1) = \Theta(n)$	insertFirst(e) $\Theta(1)$
Sorted	first() $\Theta(1)$	remove(first()) $\Theta(1)$	insert(search(e), e) $\Theta(n) + \Theta(1) = \Theta(n)$

❖ AVL、Splay、Red-black：三个接口均只需 $\mathcal{O}(\log n)$ 时间

但是，BBST的功能远远超出了PQ的需求：

$$PQ = \boxed{1 \times \text{insert}()} + \boxed{0.5 \times \text{search}()} + \boxed{0.5 \times \text{remove}()}$$

❖ 若只需查找极值元，则不必维护所有元素之间的全序关系，偏序足矣

❖ 因此有理由相信，存在某种更为简单、维护成本更低的实现方式

使得各功能接口的时间复杂度依然为 $\mathcal{O}(\log n)$ ，而且实际效率更高

❖ 当然，就最坏情况而言，这类实现方式已属最优——为什么？

统一测试

```
template <typename PQ, typename T> void testHeap( int n ) {
```

```
    T* A = new T[ 2*n/3 ]; //创建容量为2n/3的数组，并  
    for ( int i = 0; i < 2*n/3; i++ ) A[i] = dice( (T) 3*n ); //随机化
```

```
    PQ heap( A + n / 6, n / 3 ); delete [] A; //Robert Floyd
```

```
    while ( heap.size() < n ) //随机测试  
        if ( dice( 100 ) < 70 ) heap.insert( dice( (T) 3*n ) ); //70%概率插入  
        else if ( ! heap.empty() ) heap.delMax(); //30%概率删除  
    while ( ! heap.empty() ) heap.delMax(); //清空
```

```
}
```

优先级队列

完全二叉堆：结构

12-B1

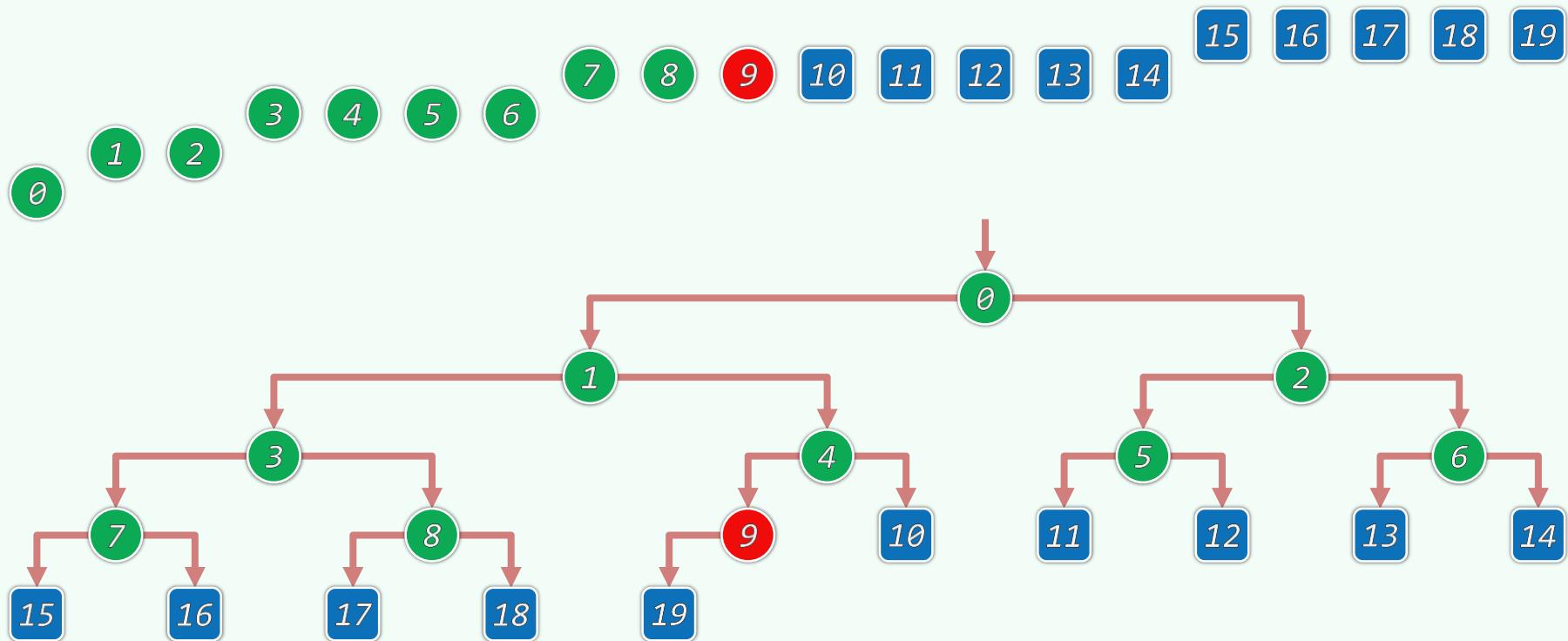
邓俊辉

deng@tsinghua.edu.cn

何人将乱石作堆？如何乱石堆中有杀气冲起？

结构性：逻辑上等同于完全二叉树，物理上直接借助向量实现

```
#define Parent(i)  ( ((i) - 1) >> 1 )      ❖ 内部节点的最大秩 = ⌊  $\frac{n-2}{2}$  ⌋ = ⌈  $\frac{n-3}{2}$  ⌉  
#define LChild(i) ( 1 + ((i) << 1) )  
#define RChild(i) ( (1 + (i)) << 1 )
```



PQ_CmplHeap = PQ + Vector

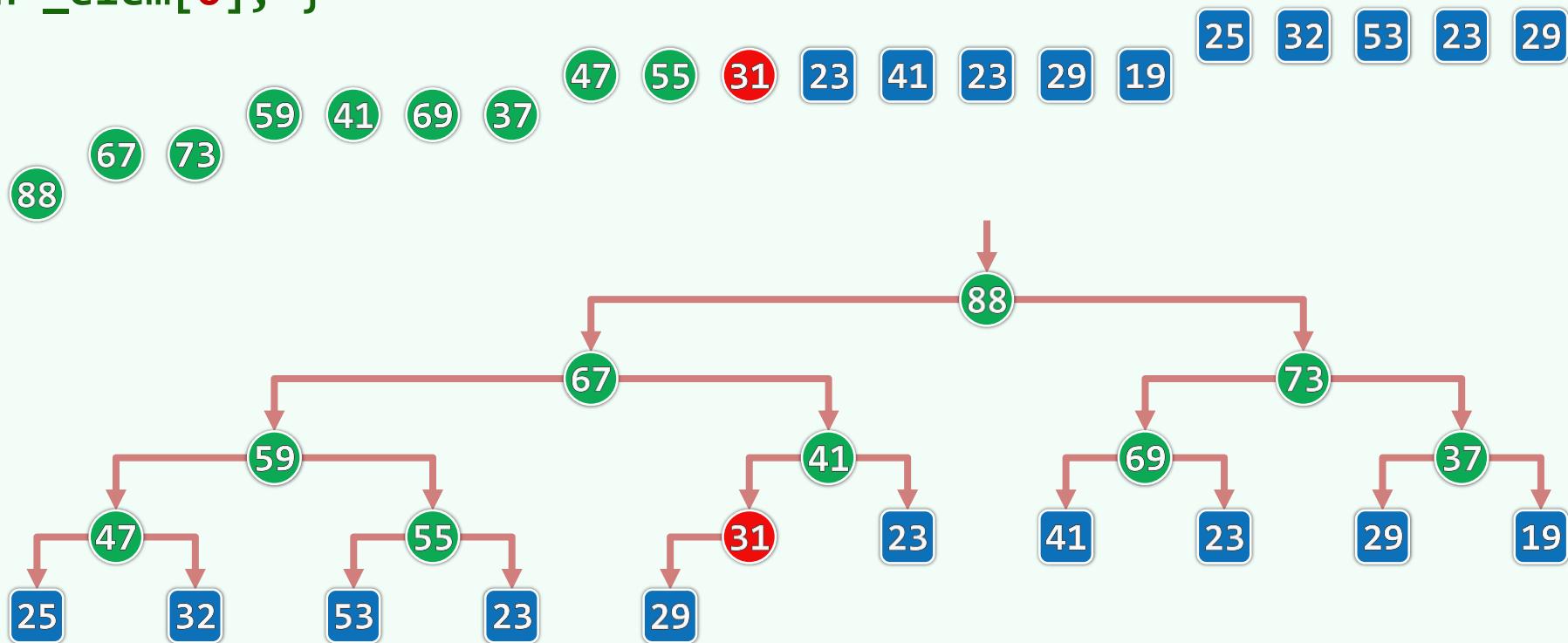
- ❖ template <typename T> struct PQ_CmplHeap : public PQ<T>, public Vector<T> {
 PQ_CmplHeap(T* A, Rank n) { copyFrom(A, 0, n); heapify(_elem, n); }
 void insert(T); T delMax(); T & getMax();
};
- ❖ template <typename T> Rank percolateDown(T* A, Rank n, Rank i); //下濾
template <typename T> Rank percolateUp(T* A, Rank i); //上濾
- ❖ template <typename T> void heapify(T* A, Rank n); //Floyd建堆算法

堆序性：处处满足 $H[i] \leq H[\text{Parent}(i)]$

❖ 故 $H[0]$ 即是全局最大者

❖ template <typename T> T & PQ_CmplHeap<T>::getMax()

```
{ return _elem[0]; }
```



优先级队列

完全二叉堆：插入

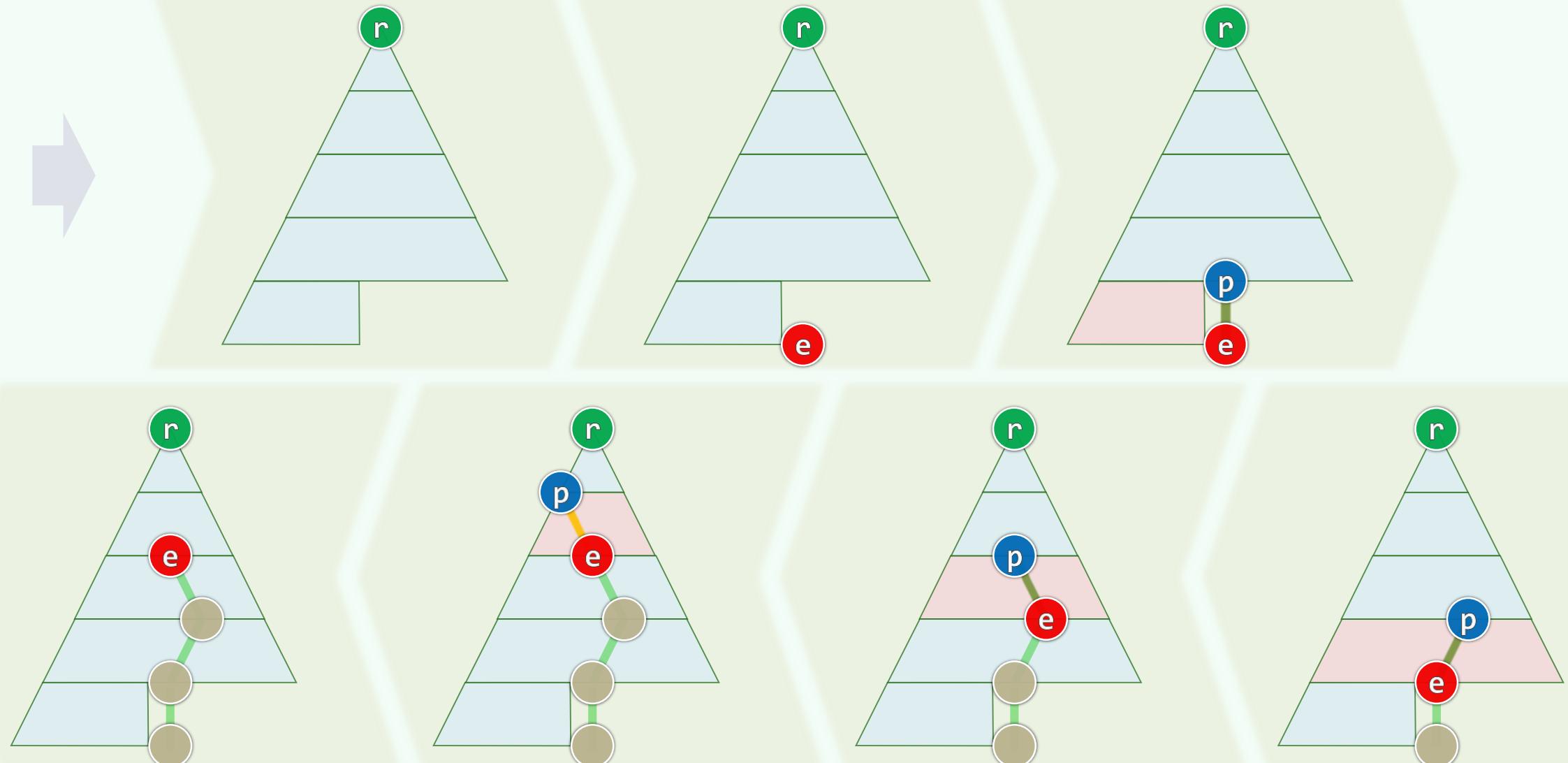
12-B2

时迁看见土地庙后一株大柏树，便把两只腿夹定，一
节节爬将上去树头顶，骑马儿坐在枝柯上

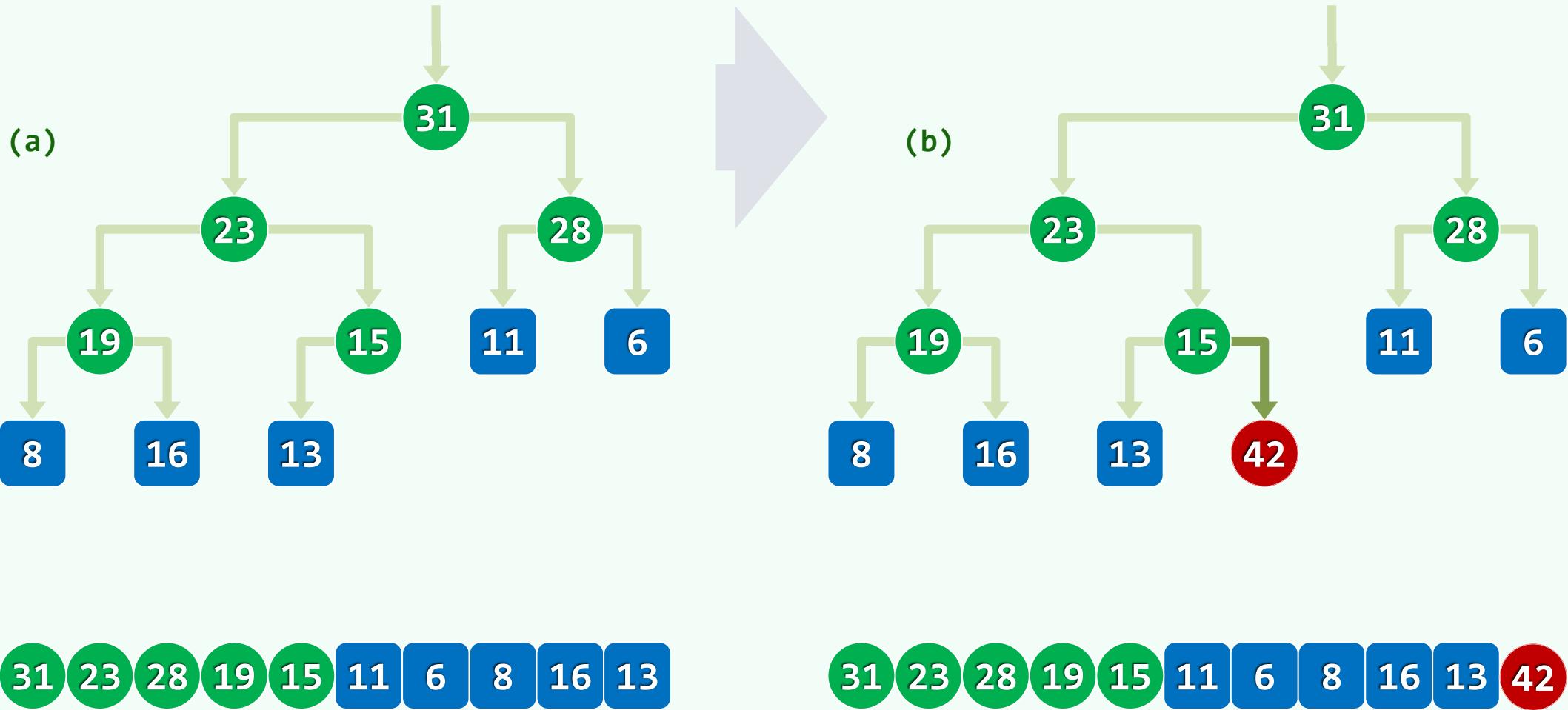
邓俊辉

deng@tsinghua.edu.cn

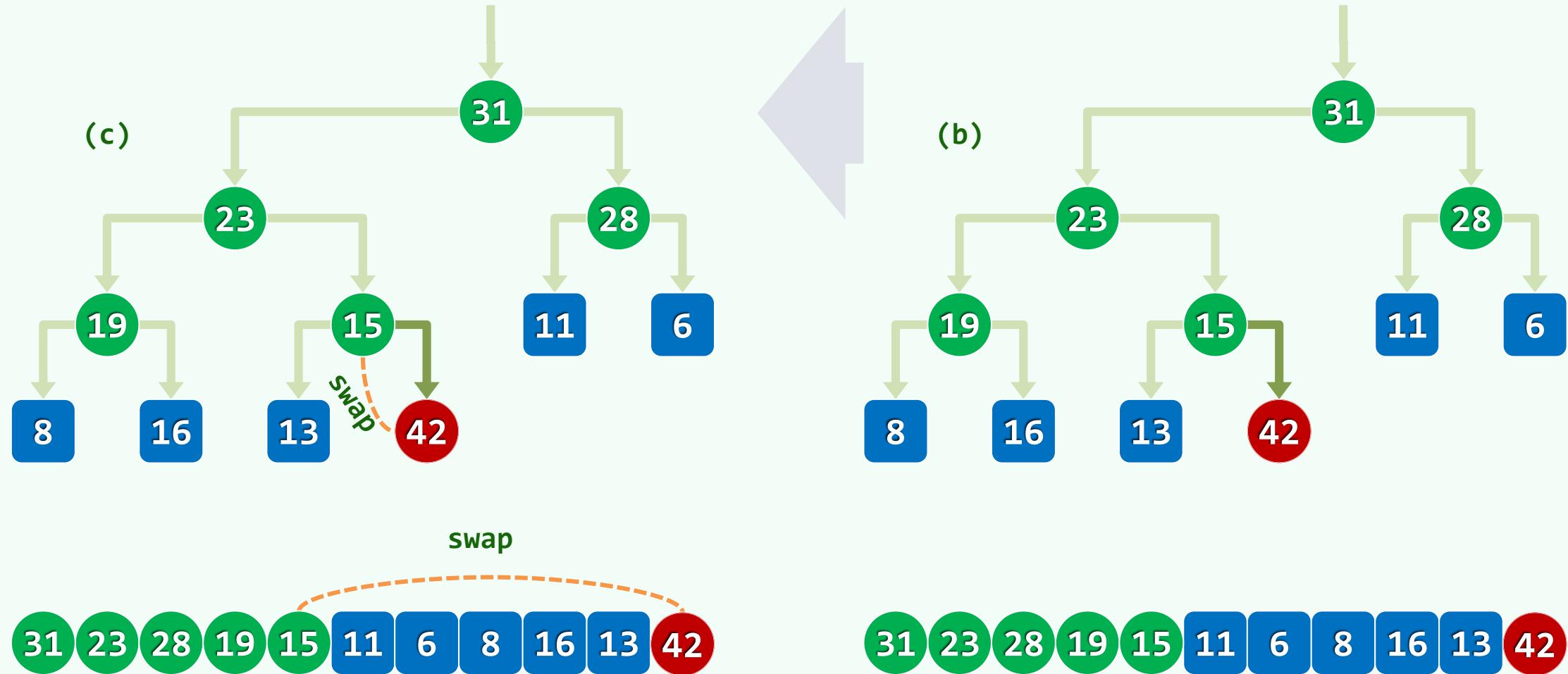
算法：逐层上滤



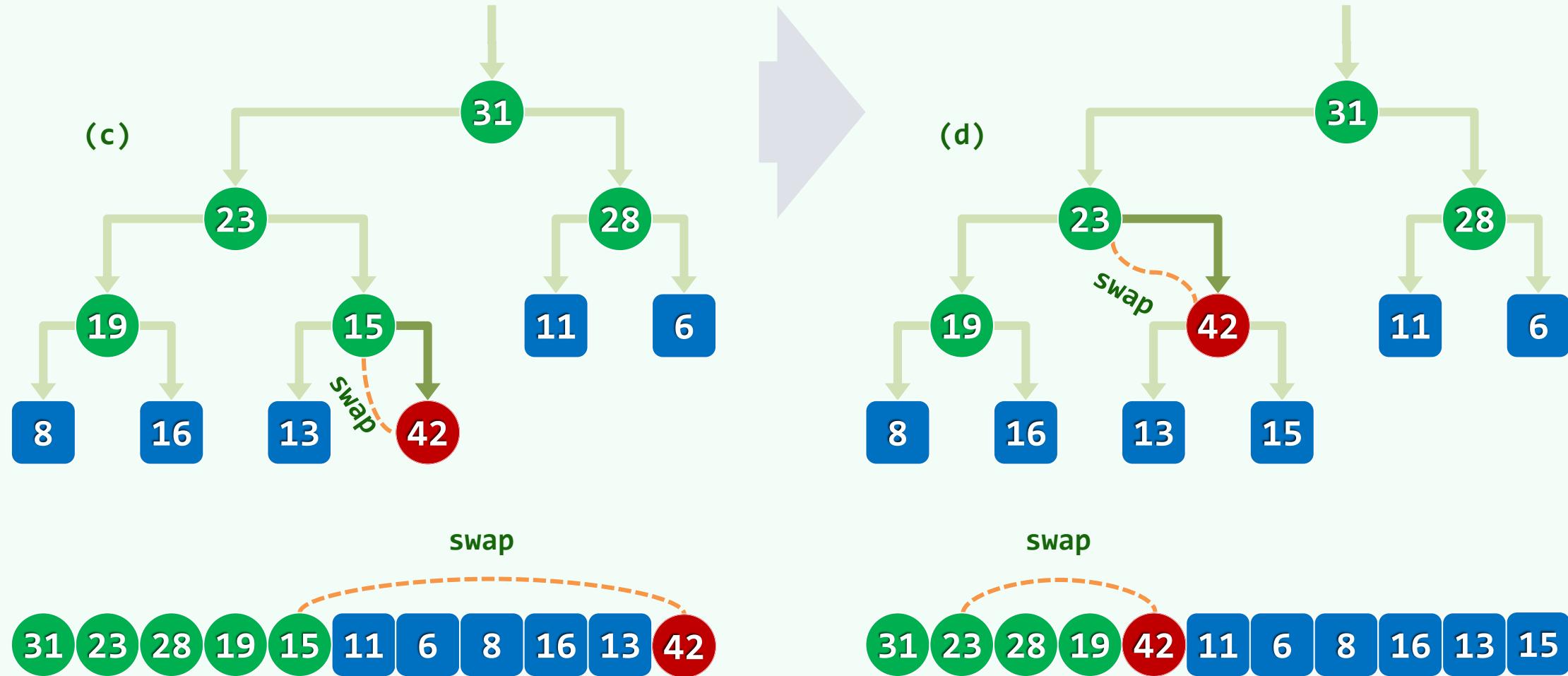
实例 (1/5)



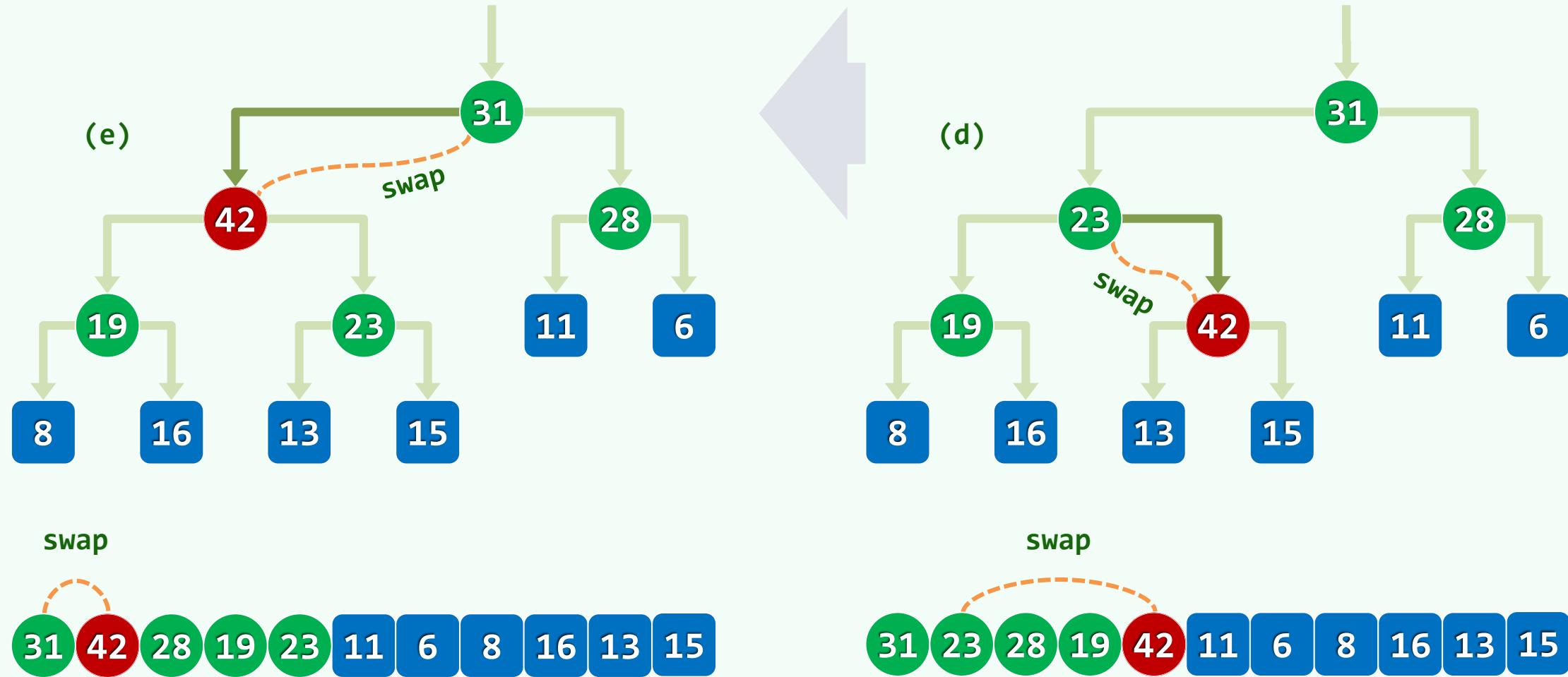
实例 (2/5)



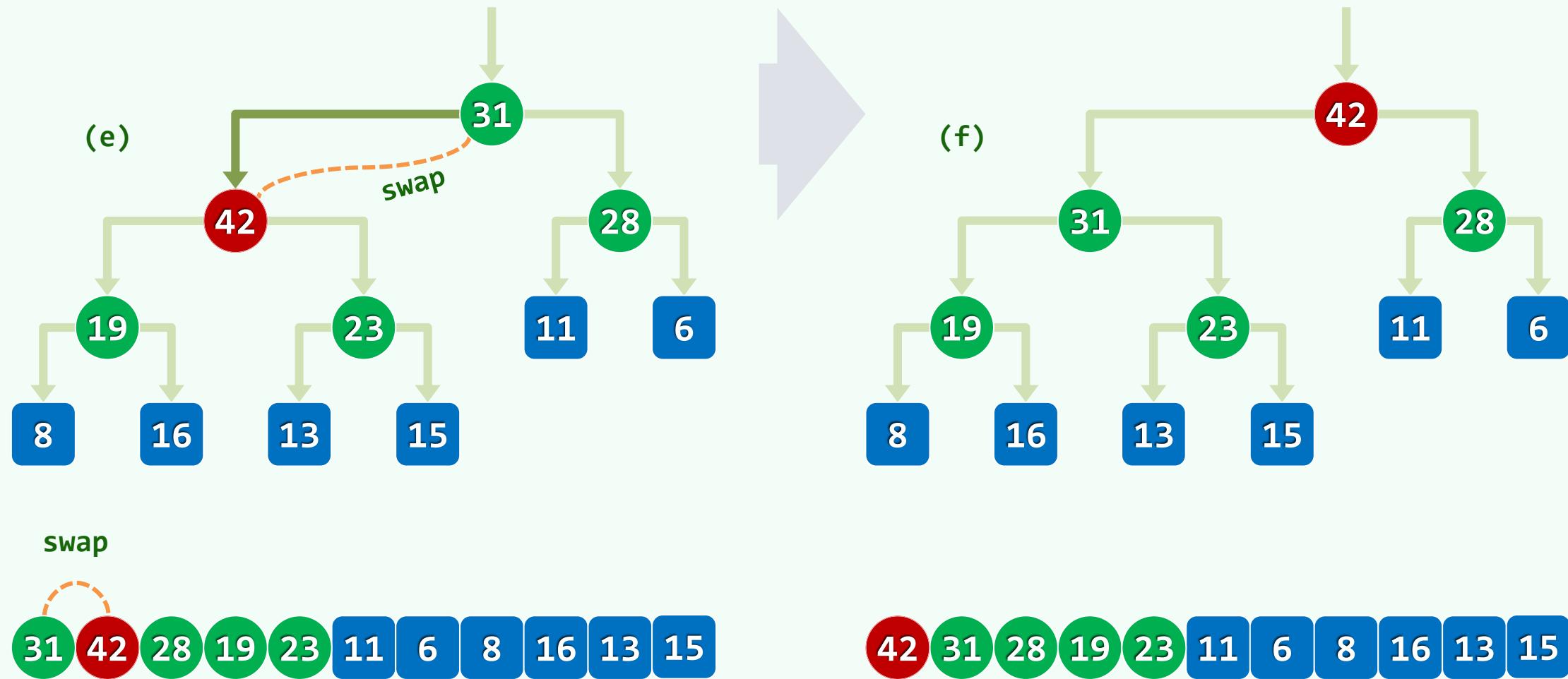
实例 (3/5)



实例 (4/5)



实例 (5/5)



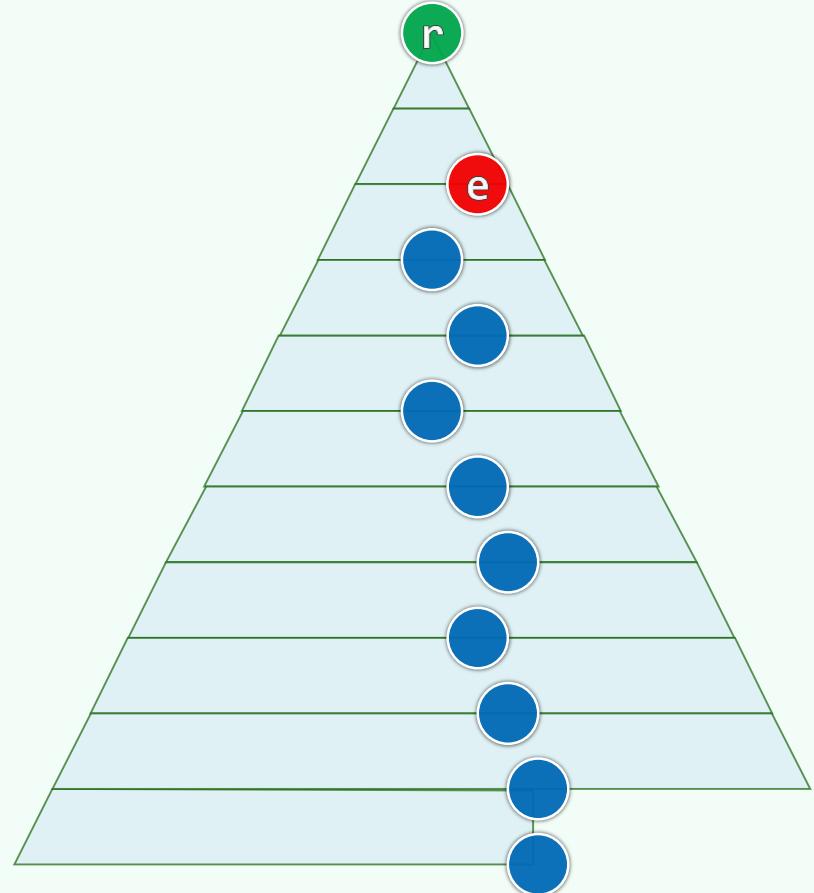
实现

```
❖ template <typename T> void PQ_CmplHeap<T>::insert( T e ) //插入
{ Vector<T>::insert( e ); percolateUp( _elem, _size - 1 ); } //先接入，再上滤

❖ template <typename T> Rank percolateUp( T* A, Rank i ) { //0 <= i < _size
    while ( 0 < i ) { //在抵达堆顶之前，反复地
        Rank j = Parent( i ); //考查[i]之父亲[j]
        if ( ! ( A[j] < A[i] ) ) break; //一旦父子顺序，上滤旋即完成；否则
        swap( A[i], A[j] ); i = j; //父子换位，并继续考查上一层
    } //while
    return i; //返回上滤最终抵达的位置
}
```

效率

- ❖ e在上滤过程中，只可能与祖先们交换
- ❖ 完全树必平衡，e的祖先不超过 $\mathcal{O}(\log n)$ 个
- ❖ 故知插入操作可在 $\mathcal{O}(\log n)$ 时间内完成
- ❖ 然而就数学期望而言
实际效率往往远远更高...



优先级队列

完全二叉堆：删除

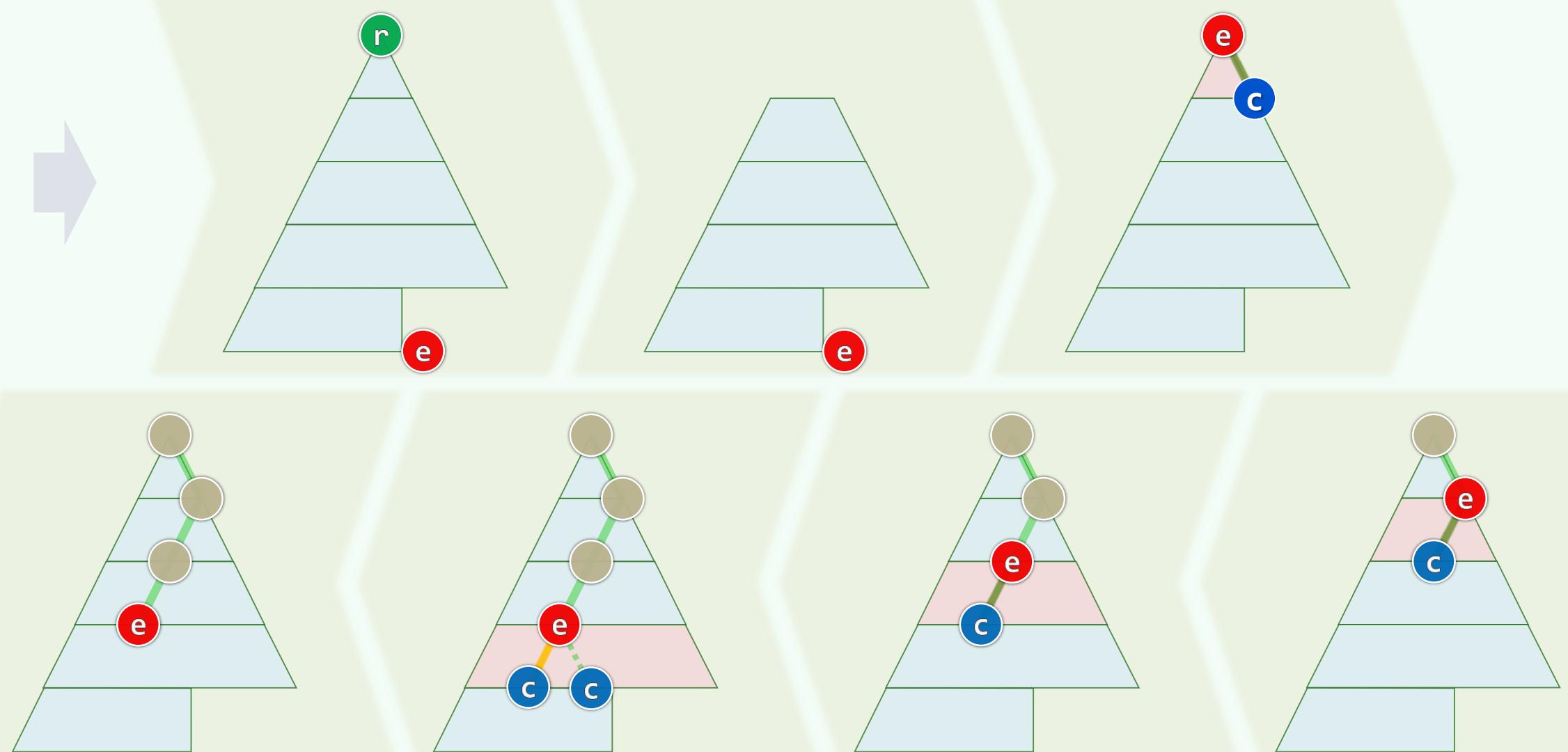
12-B3

邓俊辉

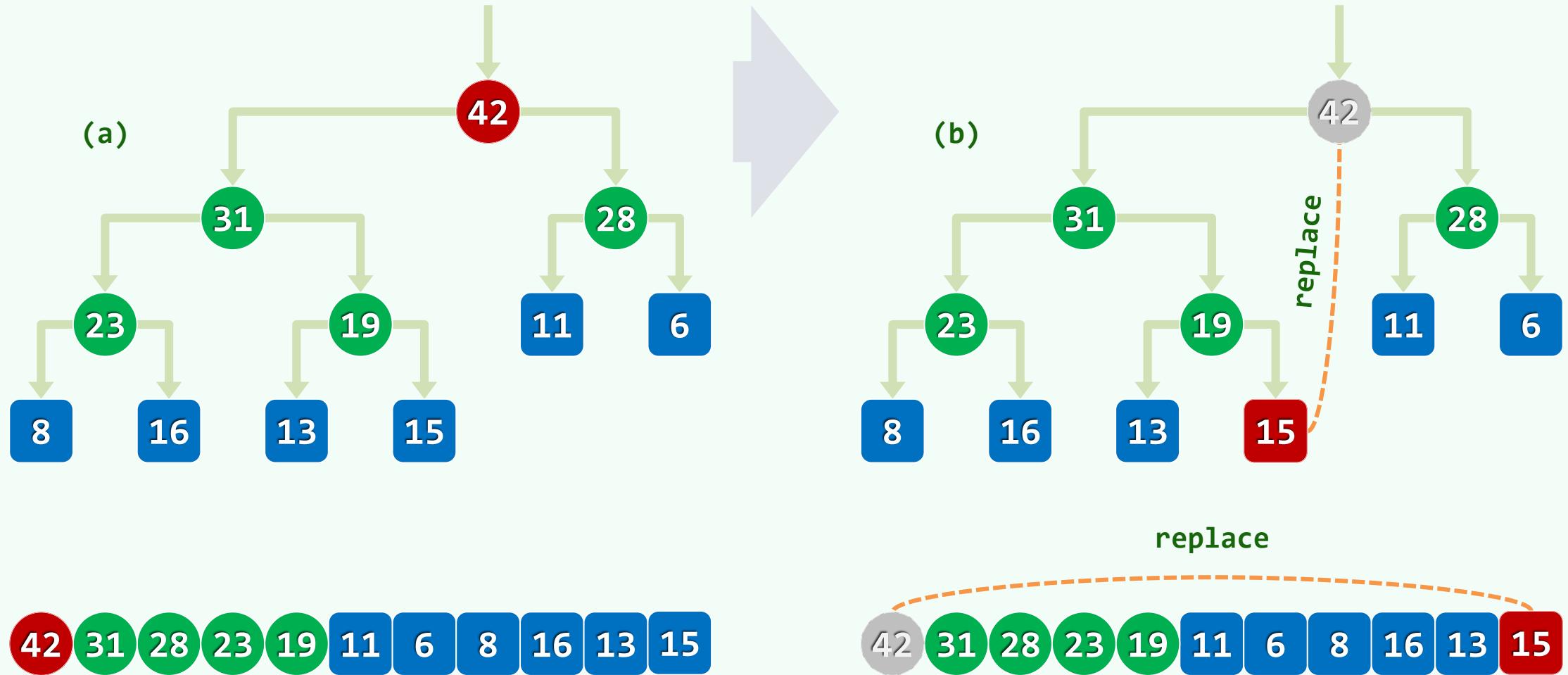
deng@tsinghua.edu.cn

I have scaled the peak and
found no shelter in
fame's bleak and barren height

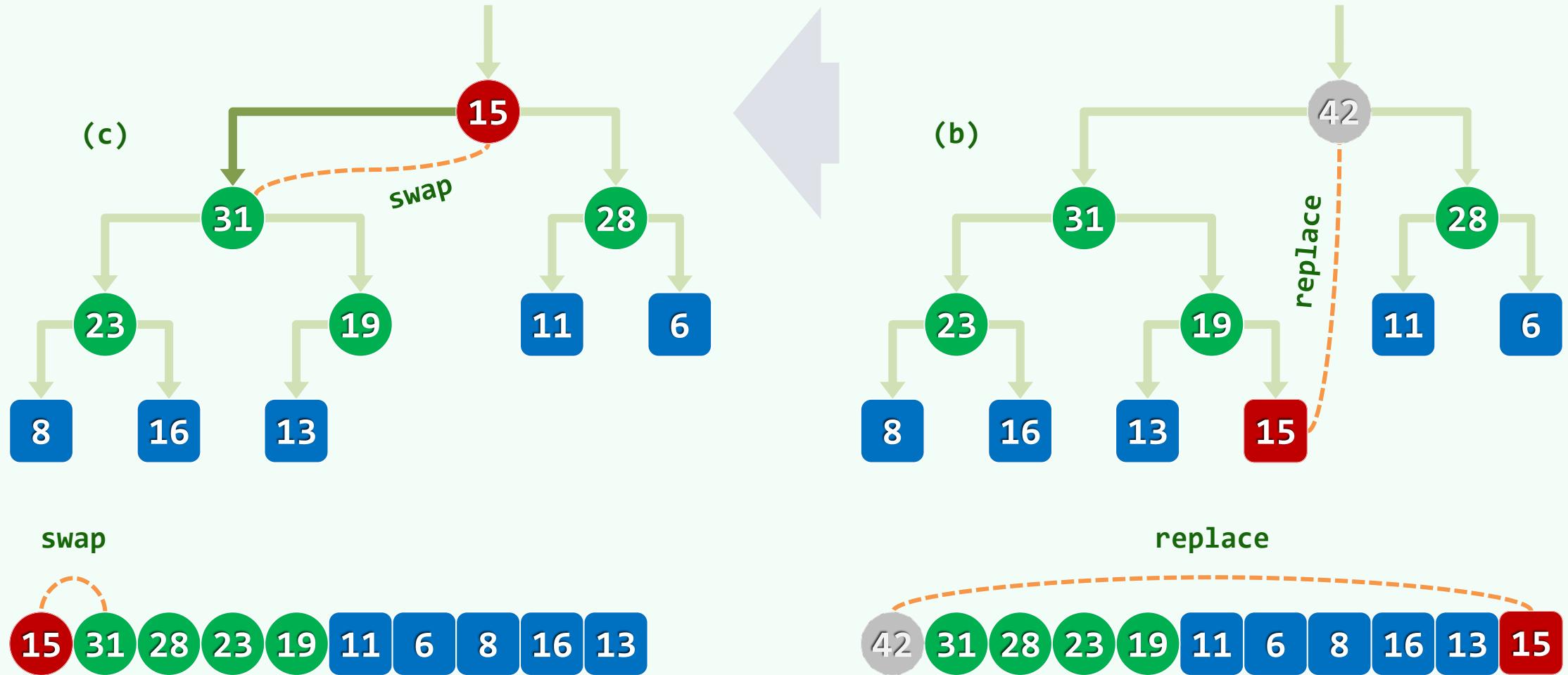
算法：割肉补疮 + 逐层下滤



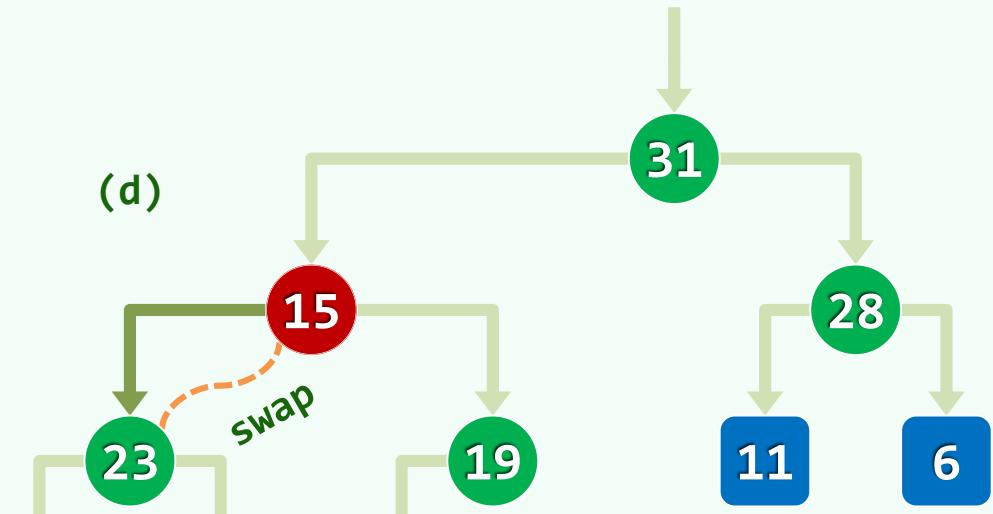
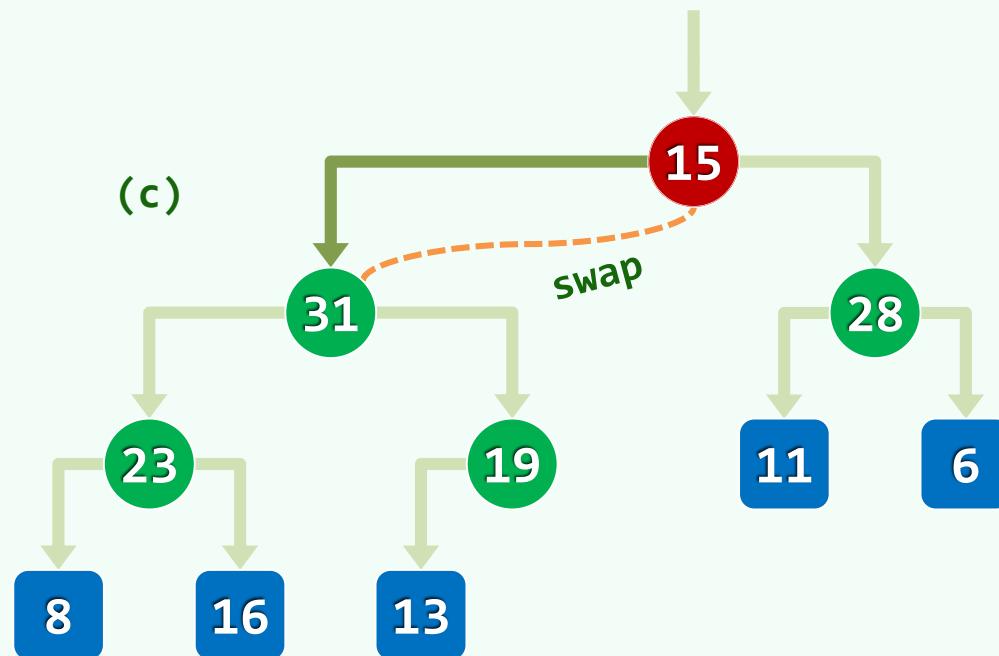
实例 (1/5)



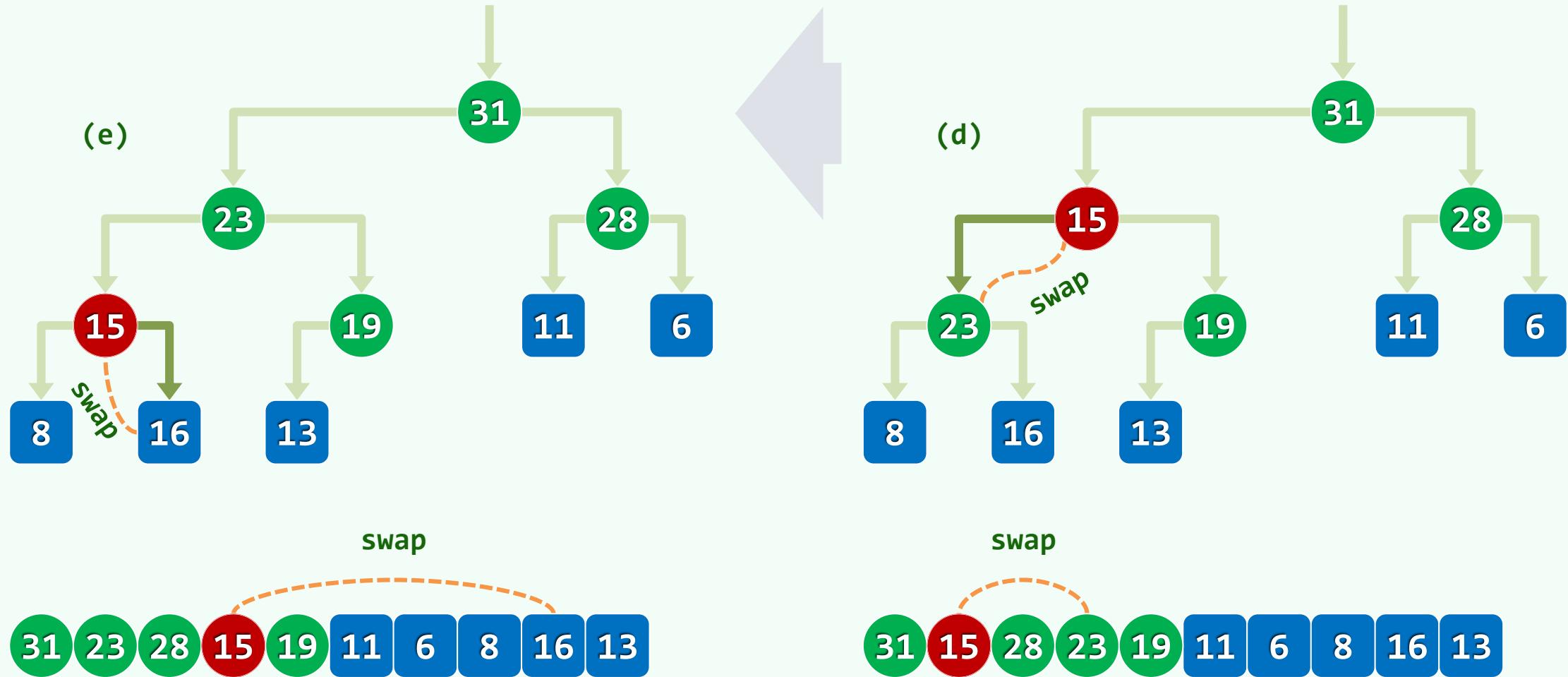
实例 (2/5)



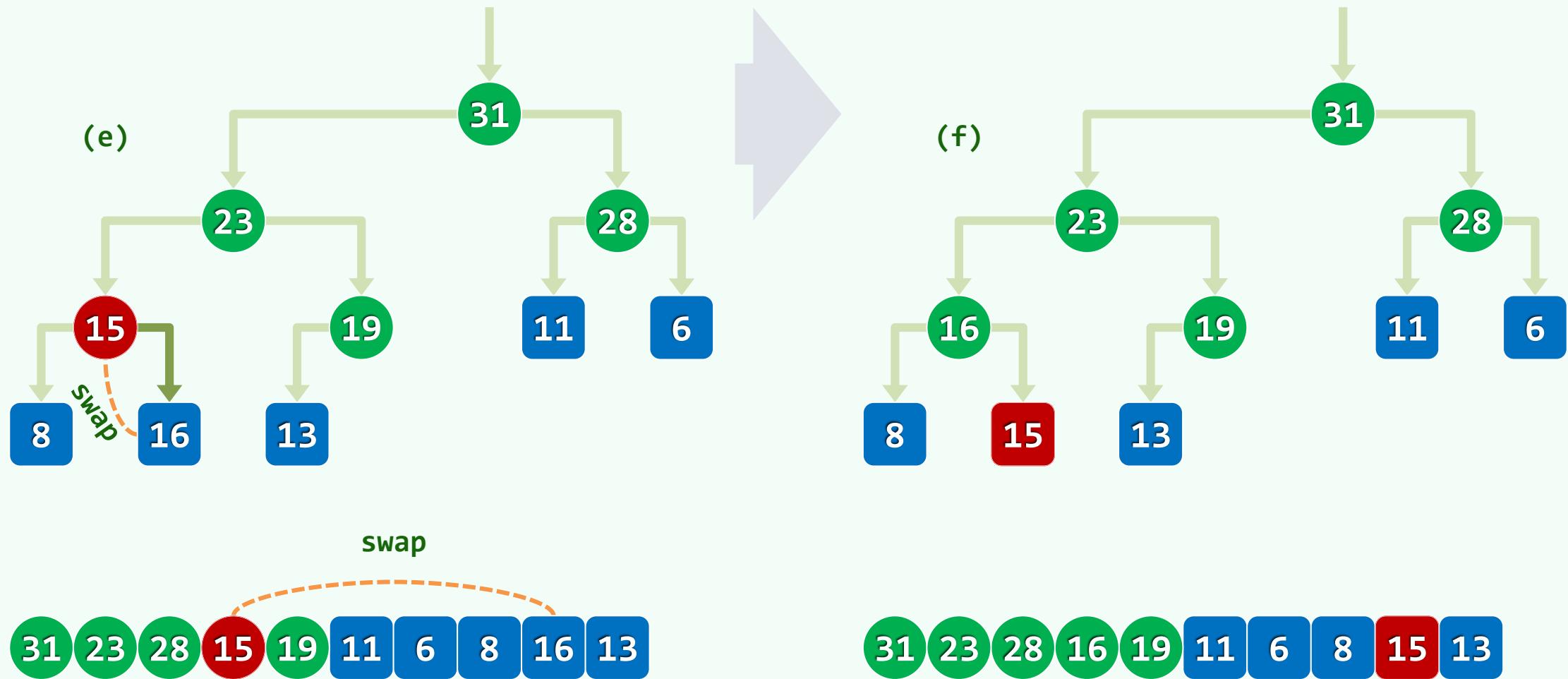
实例 (3/5)



实例 (4/5)



实例 (5/5)



实现

```
❖ template <typename T> T PQ_CmplHeap<T>::delMax() { //取出最大词条
    swap( _elem[0], _elem[ --_size ] ); //堆顶、堆尾互换 (_size递减不致引发shrink())
    percolateDown( _elem, _size, 0 ); //新堆顶下滤
    return _elem[ _size ]; //返回原堆顶
}

❖ template <typename T> Rank percolateDown( T* A, Rank n, Rank i ) { //0 <= i < n
    Rank j; //i及其(至多两个)孩子中，堪为父者
    while ( i != ( j = ProperParent( A, n, i ) ) ) //只要i非j，则
        swap( A[i], A[j] ), i = j; //换位，并继续考察i
    return i; //返回下滤抵达的位置(亦i亦j)
}
```

效率

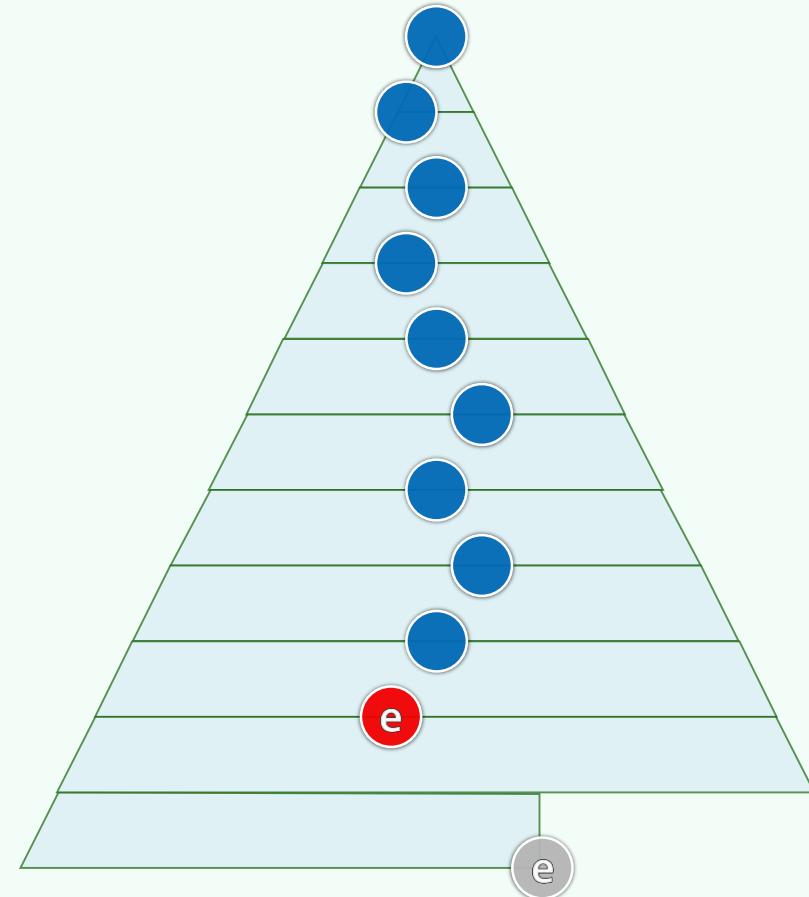
❖ e在每一高度至多交换一次

累计交换不超过 $\mathcal{O}(\log n)$ 次

❖ 通过下滤，可在 $\mathcal{O}(\log n)$ 时间内

- 删除堆顶节点，并
- 整体重新调整为堆

❖ 数学期望呢？



优先级队列

完全二叉堆：批量建堆

12-B4

今世号通人者，务为艰深之文，陈过高之义，以为士大夫劝，而独不为彼什
伯千万倍里巷乡间之子计，则是智益智，愚益愚，智日少，愚日多也

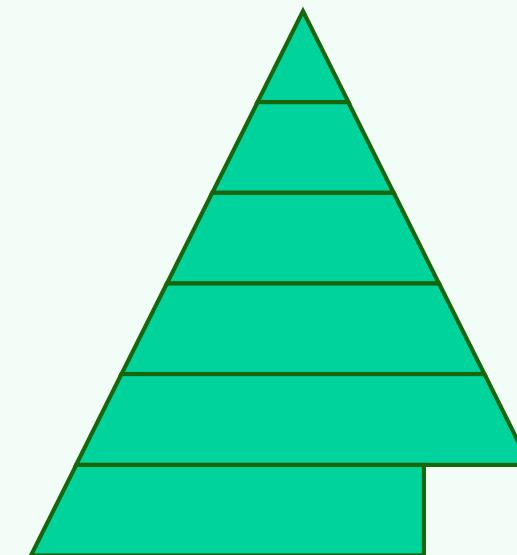
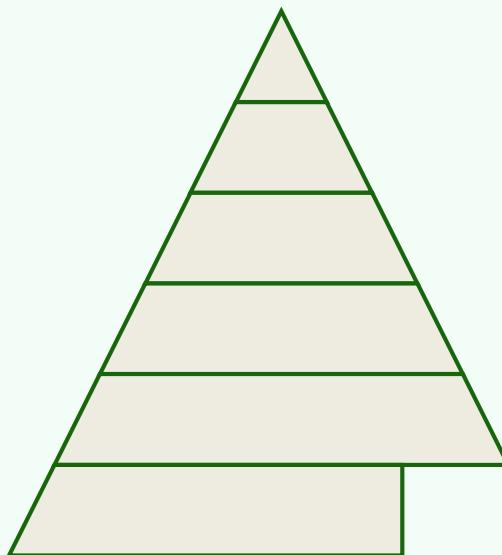
现在，培训富人应对贫穷要比教育穷人获得财富更切合实际，因为从
人数比例上来说，富人破产的越来越多，而穷人变富的越来越少

邓俊辉

deng@tsinghua.edu.cn

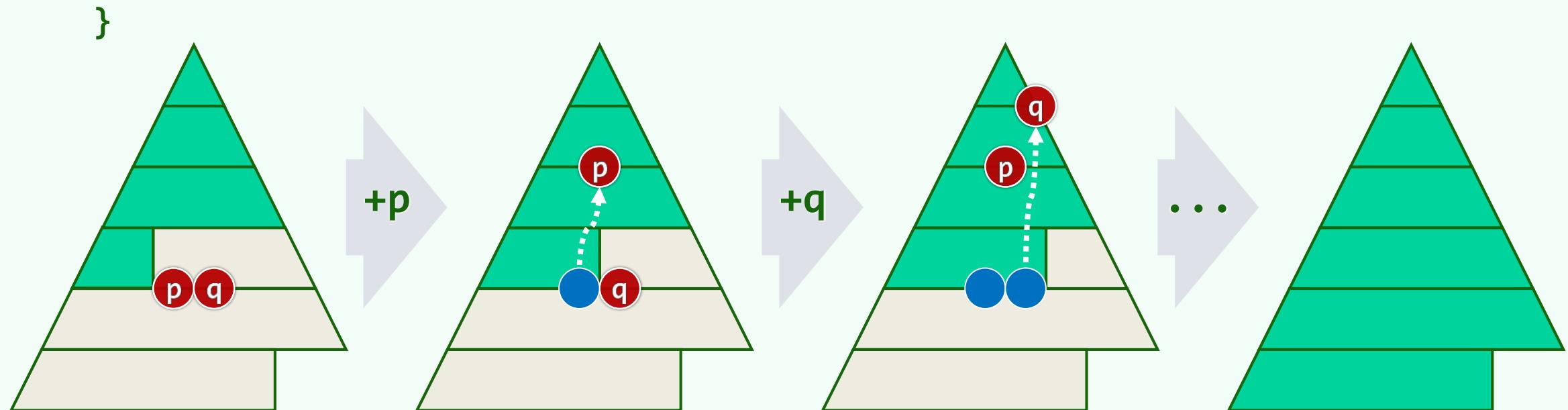
自上而下的上滤 (1/2)

```
PQ_CmplHeap( T* A, Rank n ) {  
    copyFrom( A, 0, n ); //O(n)  
    heapify( _elem, n ); //O(?)  
}
```



自上而下的上滤 (2/2)

```
template <typename T> void heapify( T* A, const Rank n ) { //蛮力  
    for ( Rank i = 1; i < n; i++ ) //按照逐层遍历次序逐一  
        percolateUp( A, i ); //经上滤插入各节点  
}
```



效率

❖ 最坏情况下

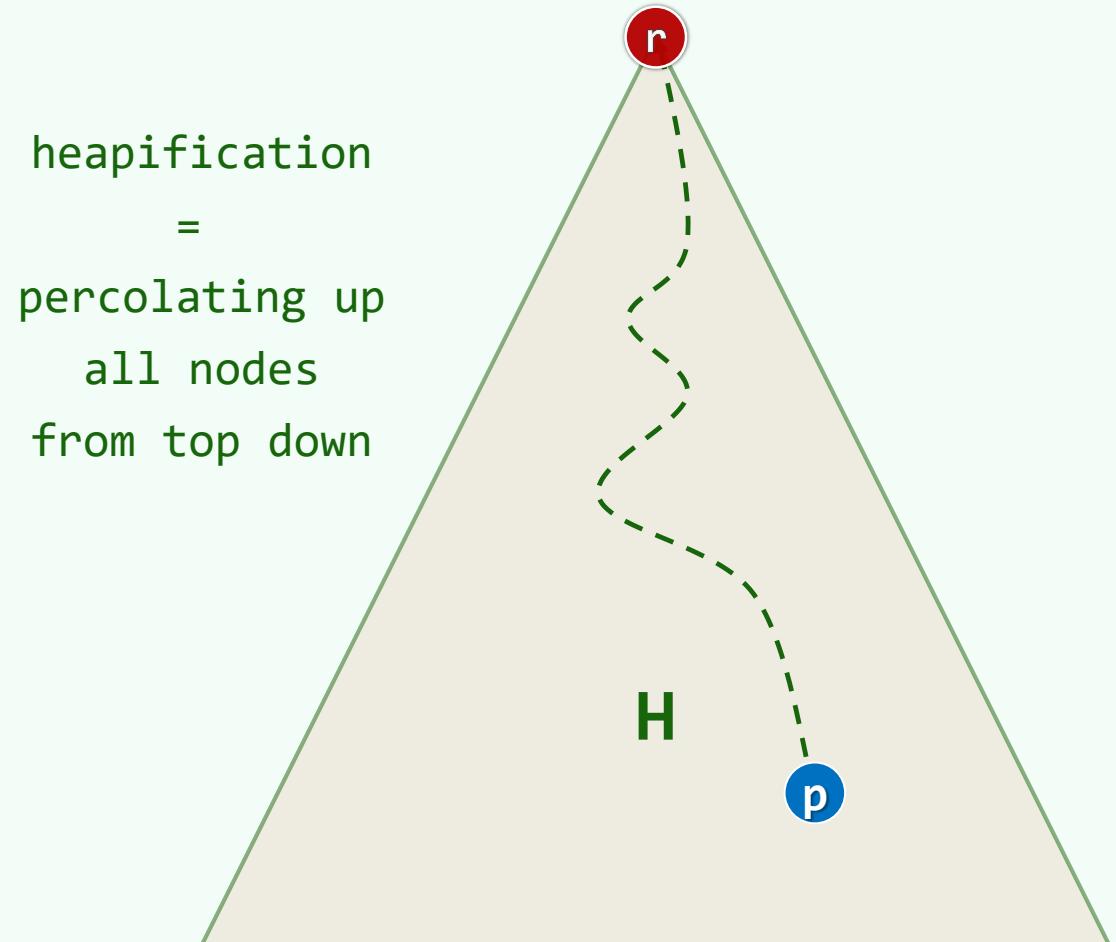
- 每个节点都需上滤至根
- 所需成本线性正比于其深度

❖ 即便只考虑底层

- $n/2$ 个叶节点，深度均为 $\mathcal{O}(\log n)$
- 亦累计耗时 $\mathcal{O}(n \log n)$

❖ 这样长的时间，本足以全排序！

所以...应该能够...更快的...



自下而上的下滤

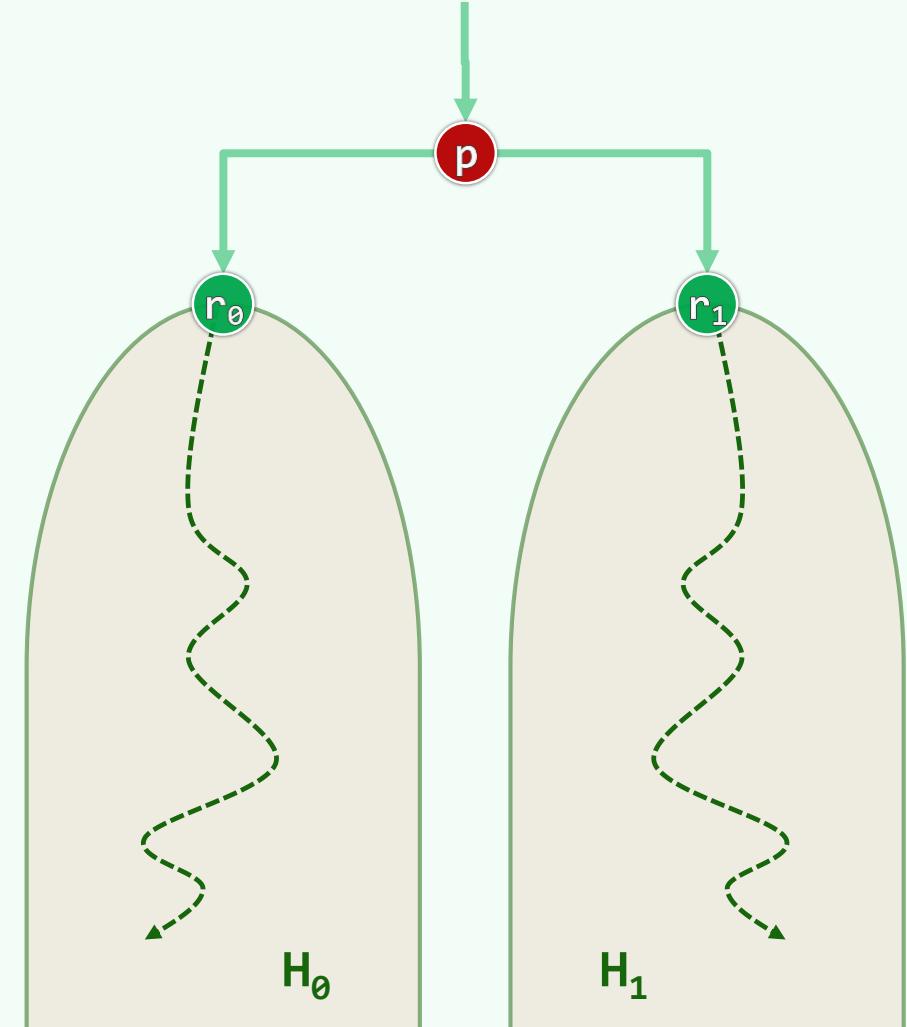
❖ 任意给定堆 \mathcal{H}_0 和 \mathcal{H}_1 ，以及节点 p

❖ 为得到堆 $\mathcal{H}_0 \cup \{p\} \cup \mathcal{H}_1$ ，只需

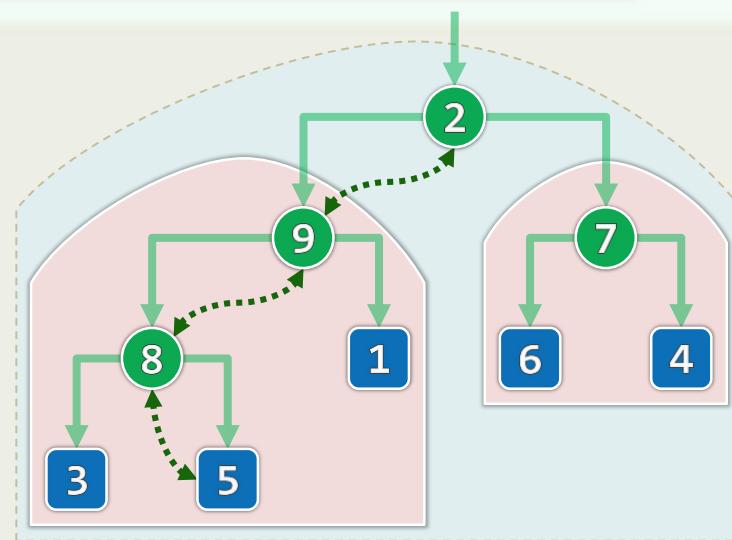
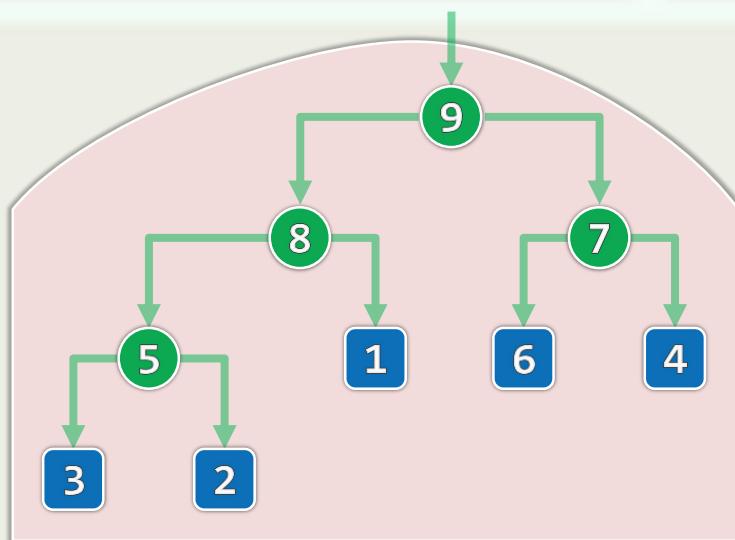
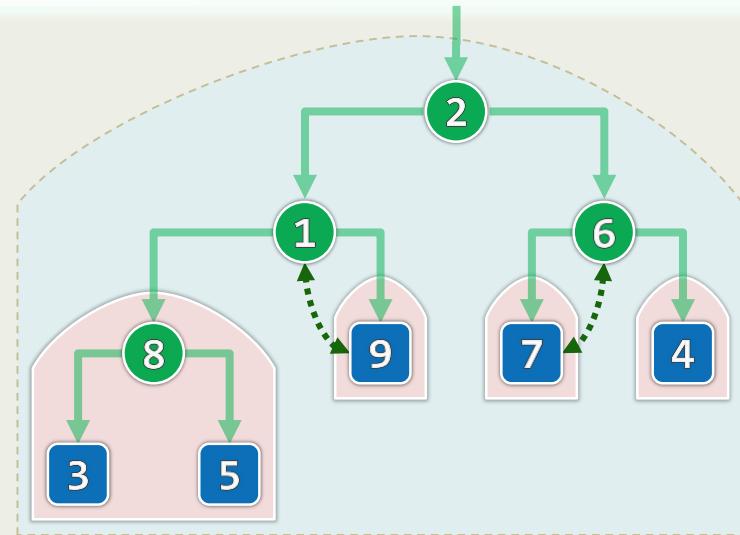
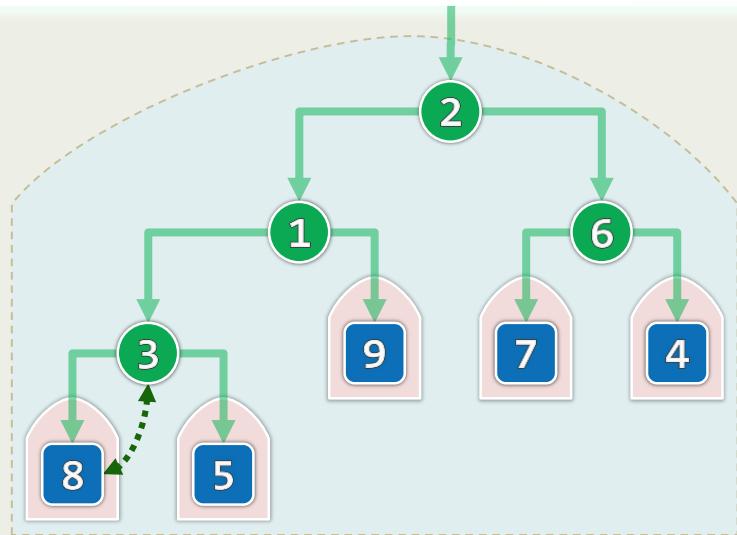
将 r_0 和 r_1 当作 p 的孩子，再对 p 下滤

❖ template <typename T> //Robert Floyd, 1964

```
void heapify( T* A, Rank n ) { //自下而上  
    for ( Rank i = n/2-1; -1 != i; i-- ) //依次  
        percolateDown( A, n, i ); //经下滤合并子堆  
} //可理解为子堆的逐层合并，堆序性最终必在全局恢复
```



实例



效率

❖ 每个内部节点所需的调整时间，正比于其高度而非深度

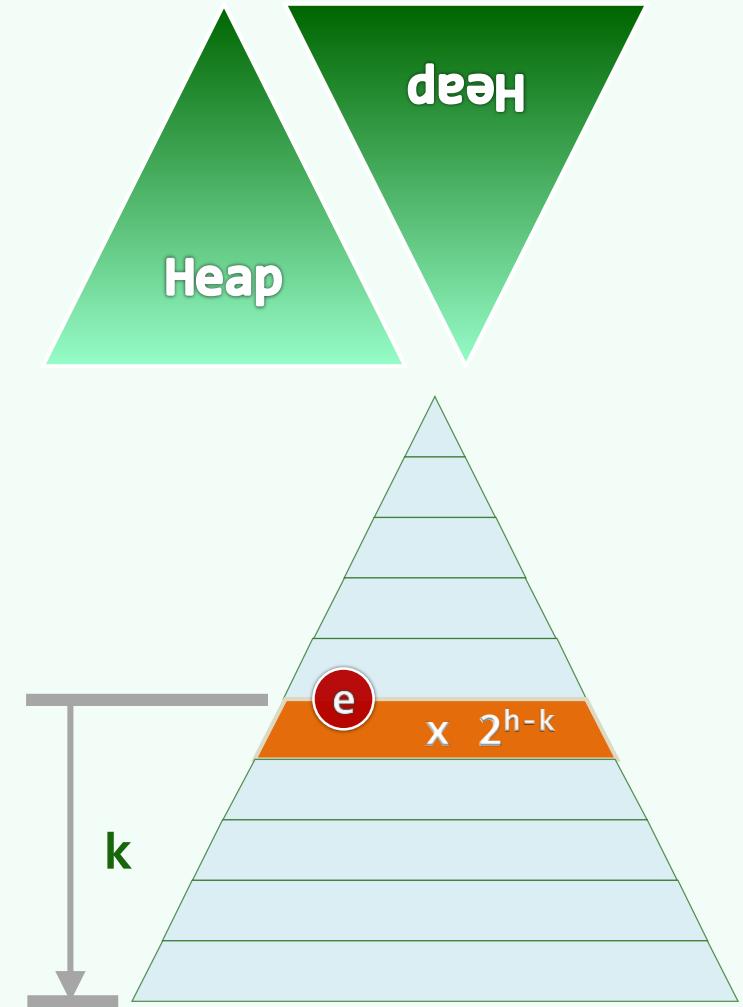
❖ 不失一般性，考查满树： $n = 2^{h+1} - 1$

❖ 所有节点的高度总和

$$S(n) = \sum_{k=1}^h k \cdot 2^{h-k} = \sum_{k=1}^h \sum_{i=1}^k 2^{h-k} = \sum_{i=1}^h \sum_{k=i}^h 2^{h-k}$$

$$= \sum_{i=1}^h \sum_{k=0}^{h-i} 2^k = \sum_{i=1}^h \{2^{h-i+1} - 1\} = \sum_{i=1}^h 2^{h-i+1} - h$$

$$= \sum_{i=1}^h 2^i - h = 2^{h+1} - 2 - h = \mathcal{O}(n)$$



- ❖ insert(): 最坏情况下效率为 $\mathcal{O}(\log n)$, 平均情况呢?
- ❖ heapify(): 构造次序颠倒后, 为什么复杂度会有实质降低?
这一算法在哪些场合不适用?
- ❖ 扩充接口: `decrease(i, delta)` //任一元素`_elem[i]`的数值减小`delta`
`increase(i, delta)` //任一元素`_elem[i]`的数值增加`delta`
`remove(i)` //删除任一元素`_elem[i]`
- ❖ 借助完全堆, 在 $\mathcal{O}(n \log n)$ 时间内构造Huffman树
- ❖ 大顶堆的delMin()操作, 能否也在 $\mathcal{O}(\log n)$ 时间内完成?
难道, 为此需要同时维护一个小顶堆?

优先级队列

堆排序

12-C

邓俊辉

deng@tsinghua.edu.cn

原理不变，还是逐个选取

❖ 在 selectionSort() 中

将 U 替换为 H...

❖ J. Williams, 1964

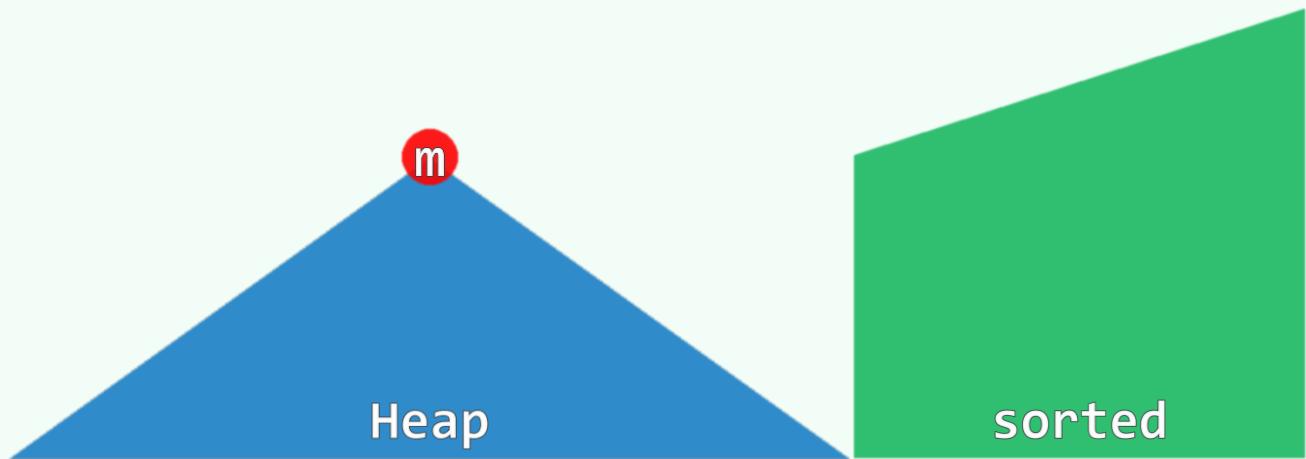
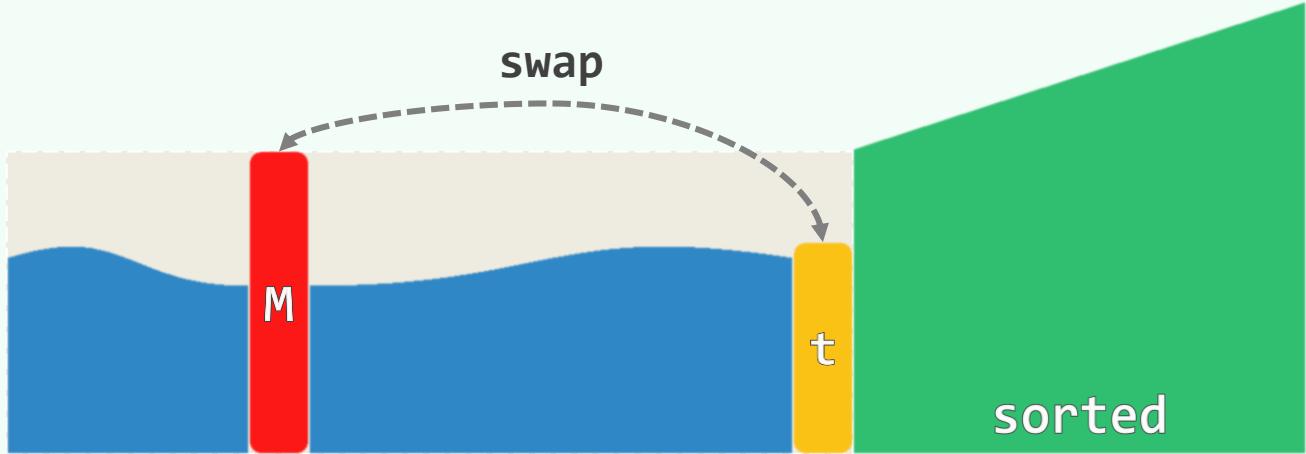
初始化 : heapify(), $\mathcal{O}(n)$

迭代 : delMax(), $\mathcal{O}(\log n)$

不变性 : $H \leq S$

❖ $\mathcal{O}(n) + n \cdot \mathcal{O}(\log n)$

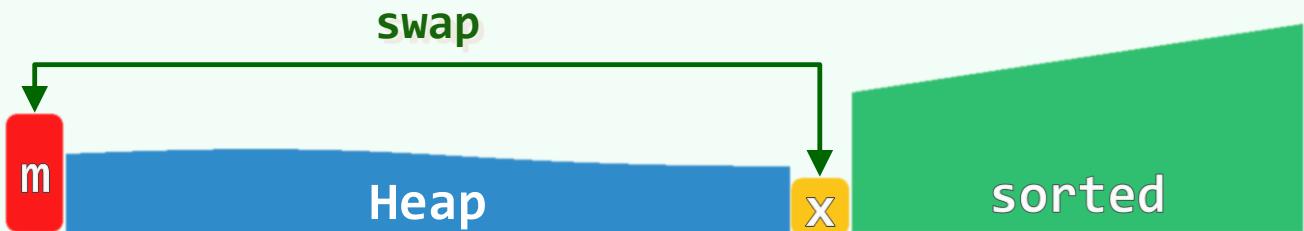
$$= \mathcal{O}(n \cdot \log n)$$



就地

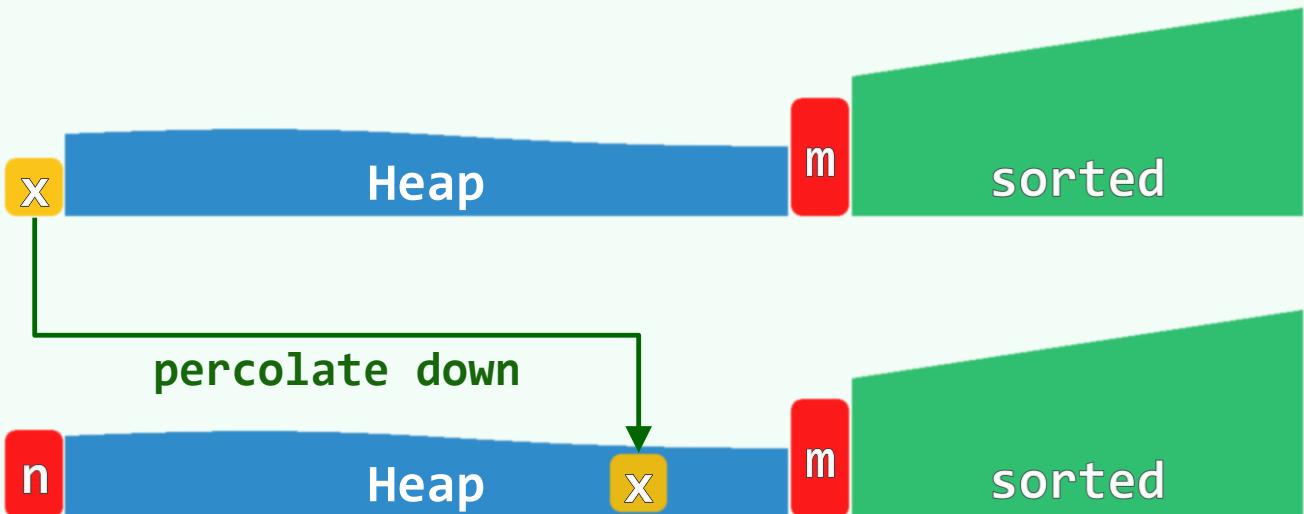
❖ 在物理上

完全二叉堆即是向量



❖ 既然此前有：

- $m = H[0]$
- $x = H[n - 1]$

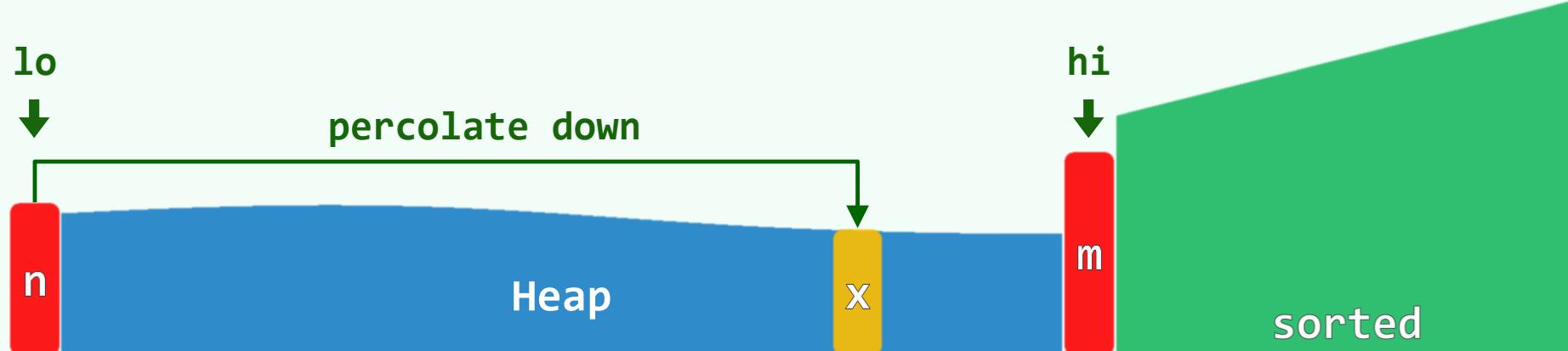


不妨随即就：

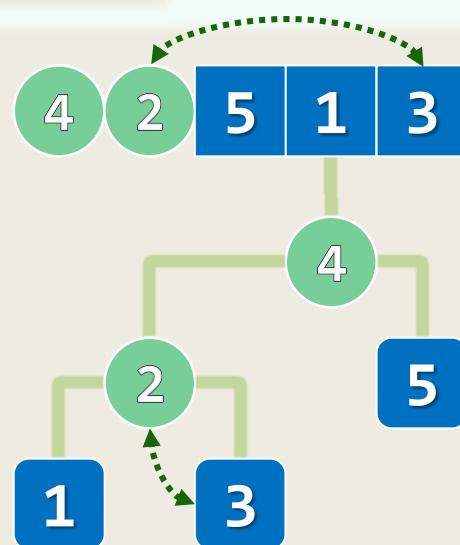
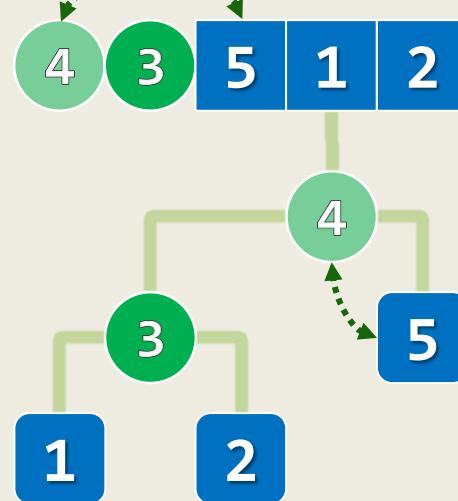
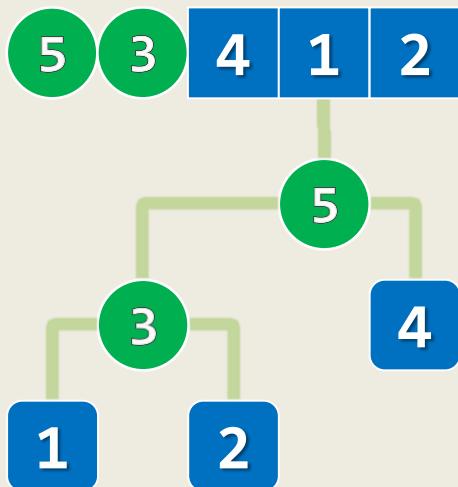
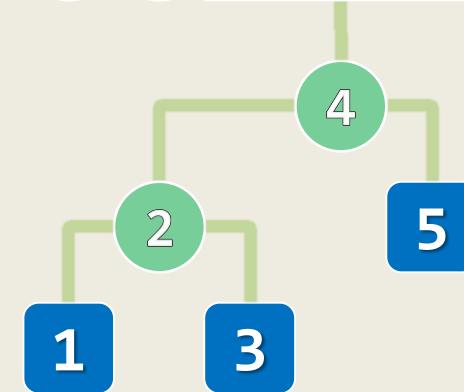
- $\text{swap}(m, x) = H.\text{insert}(x) + S.\text{insert}(m)$

实现

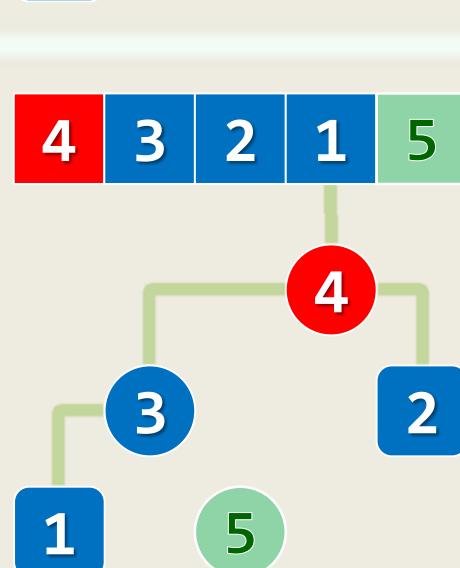
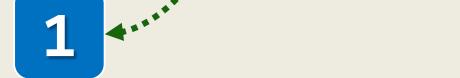
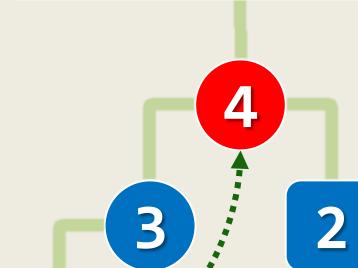
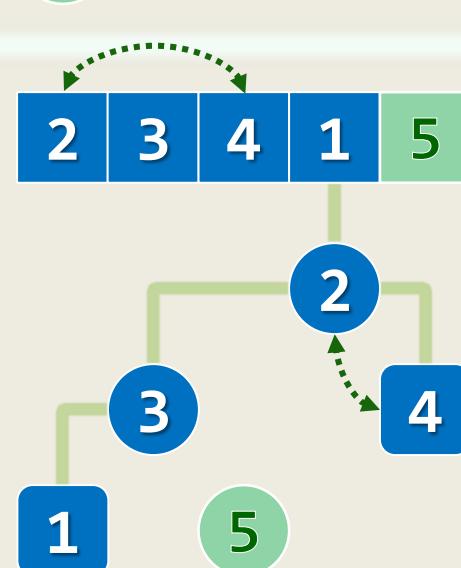
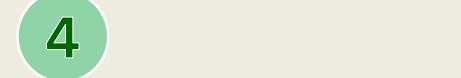
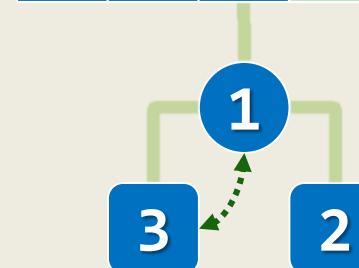
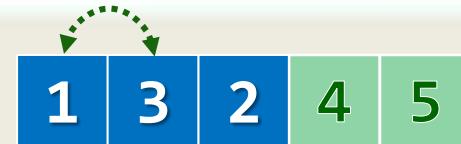
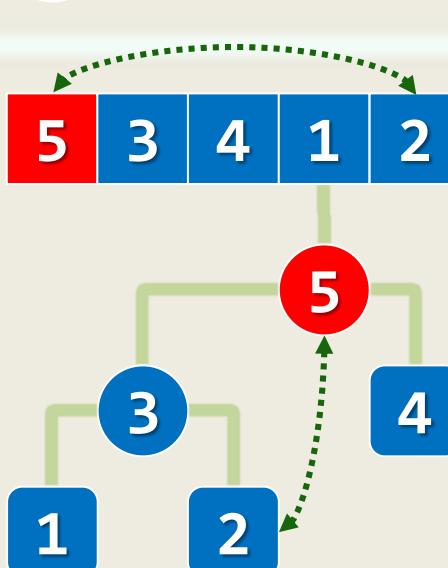
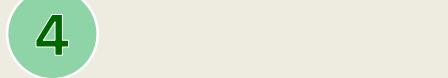
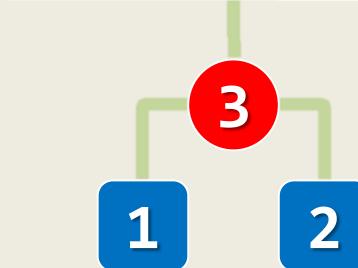
```
template <typename T> void Vector<T>::heapSort( Rank lo, Rank hi ) { //就地堆排序  
    T* A = _elem + lo; Rank n = hi - lo; heapify( A , n ); //待排序区间建堆,  $\mathcal{O}(n)$   
  
while ( 0 < --n ) //反复地摘除最大元并归入已排序的后缀, 直至堆空  
{ swap( A[0], A[n] ); percolateDown( A, n, 0 ); } //堆顶与末元素对换后下滤  
}
```



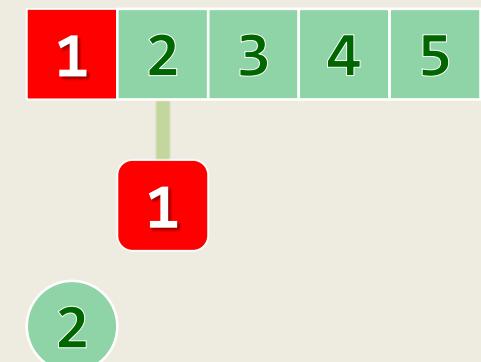
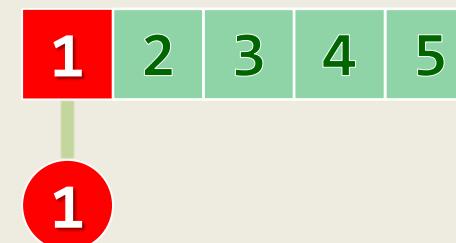
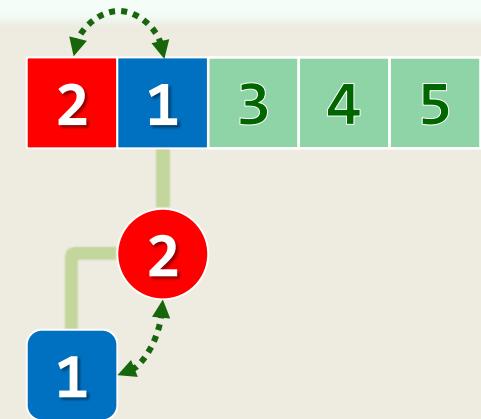
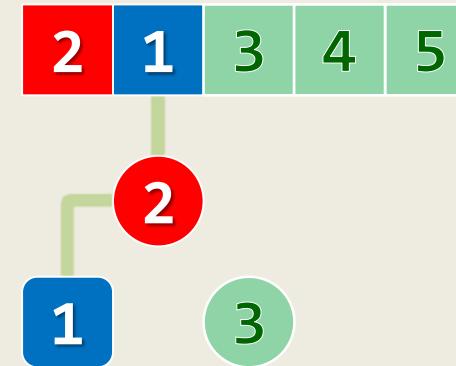
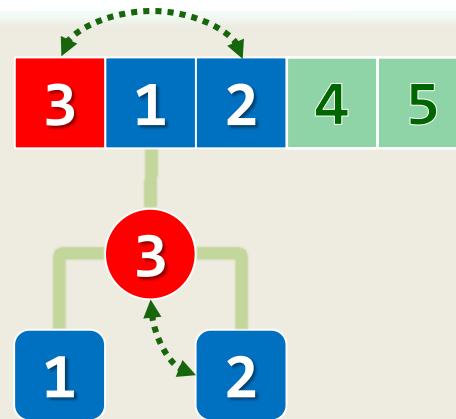
实例：建堆



实例：选取 + 调整 (1/2)



实例：选取 + 调整 (2/2)



优先级队列

锦标赛树：胜者树

12 - D1

怎么叫做分瓣梅花计？

如今把洞中大小群妖，点将起来，千中选百，百中选十，十中只选三个...

邓俊辉

deng@tsinghua.edu.cn

锦标赛树/Tournament Tree

❖ 完全二叉树

- 叶节点： 参赛选手/团队
- 内部节点： 孩子对决后之胜者

可能有重复（连胜）

❖ `create()` // $\Theta(n)$

`remove()` // $\Theta(\log n)$

`insert()` // $\Theta(\log n)$

❖ 树根总是全局冠军：类似于堆顶

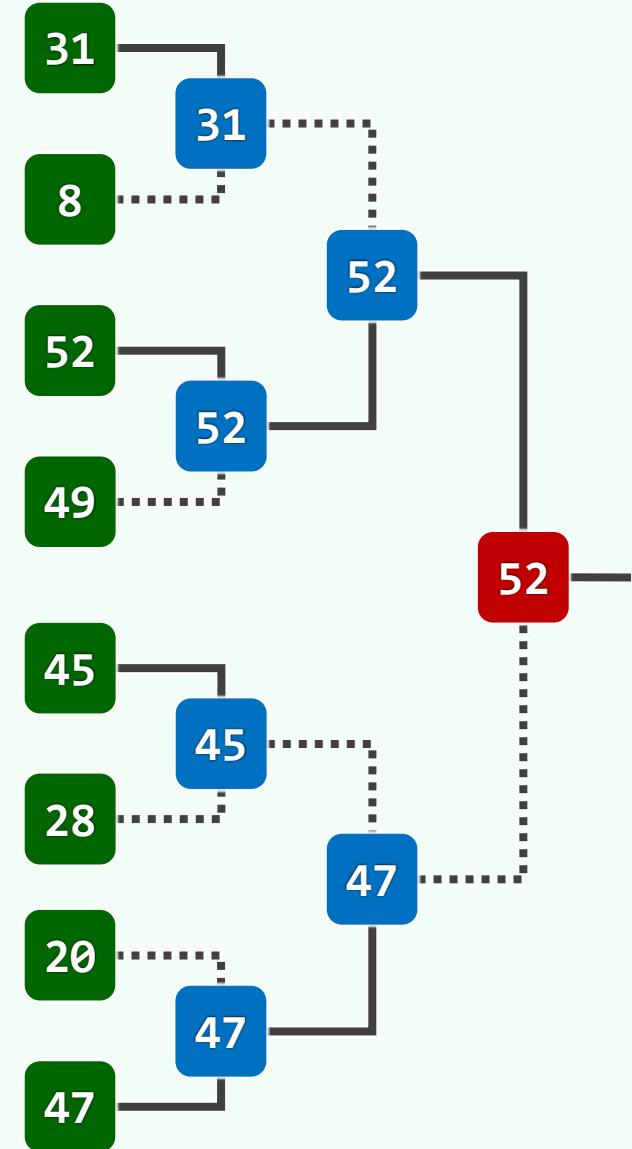


Tournamentsort()

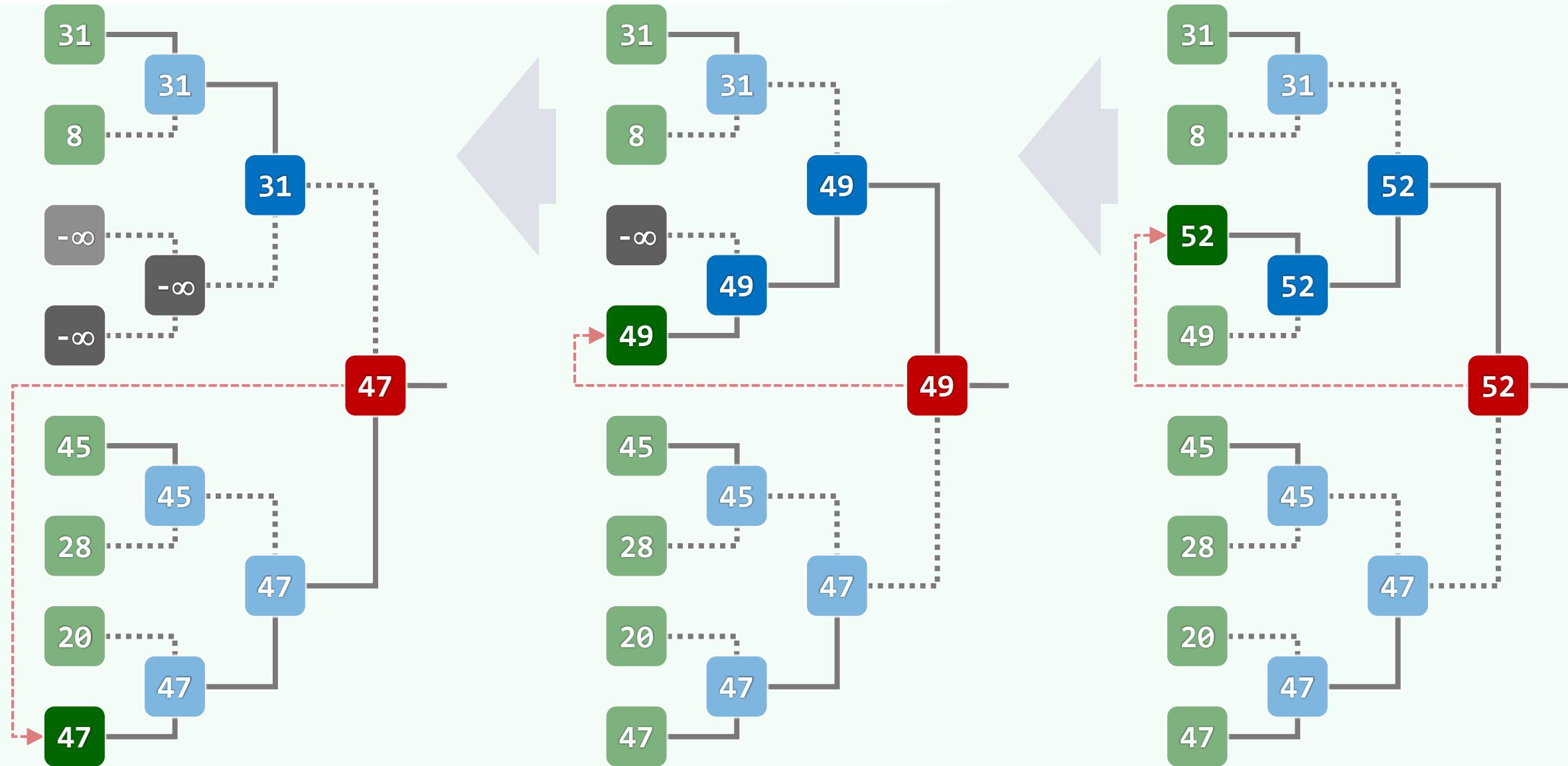
CREATE a tournament tree for the input list

while there are active leaves

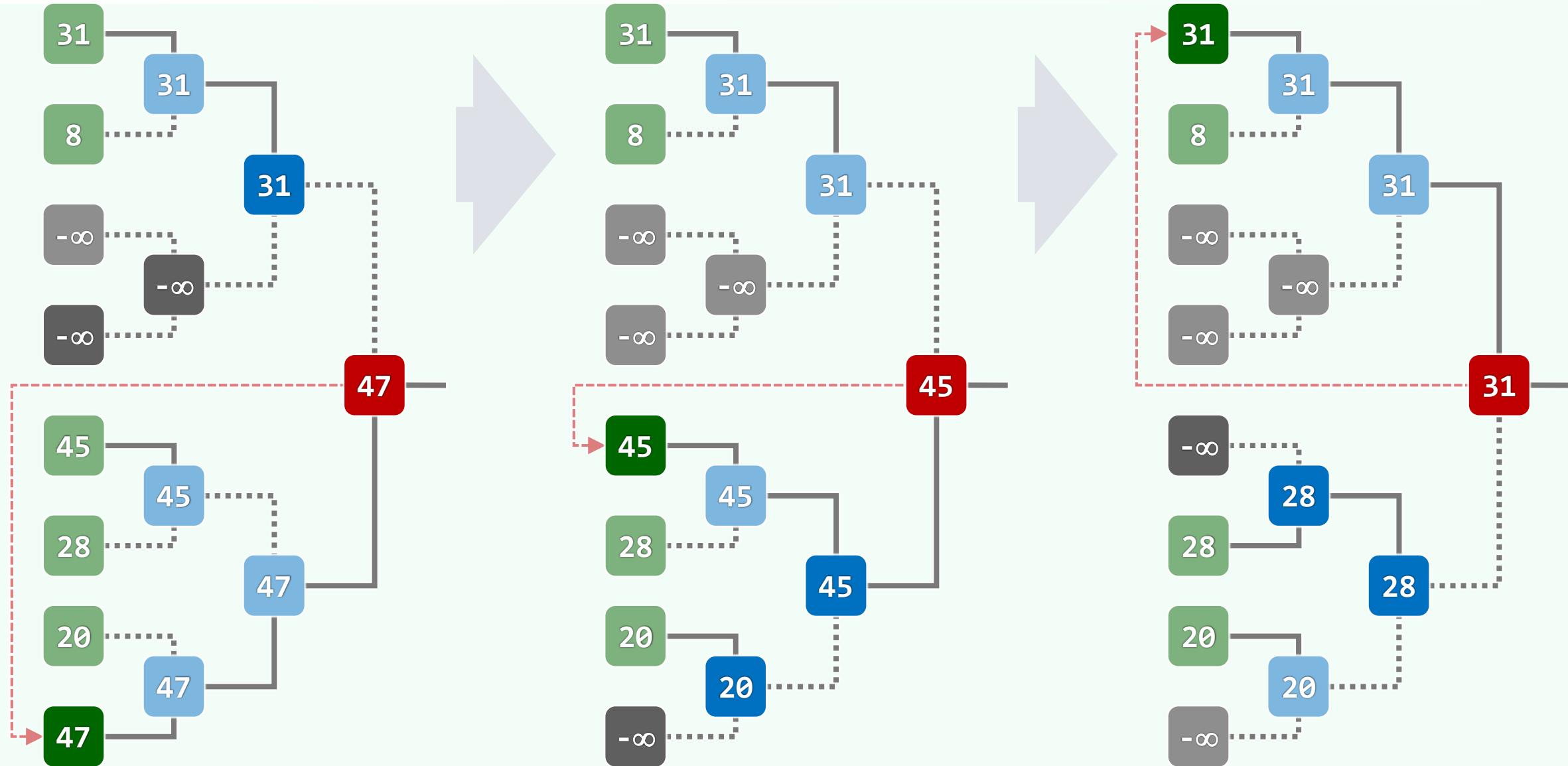
- **REMOVE** the root
- **RETRACE** the root down to its leaf
- **DEACTIVATE** the leaf
- **REPLAY** along the path back to the root



实例 (1/2)

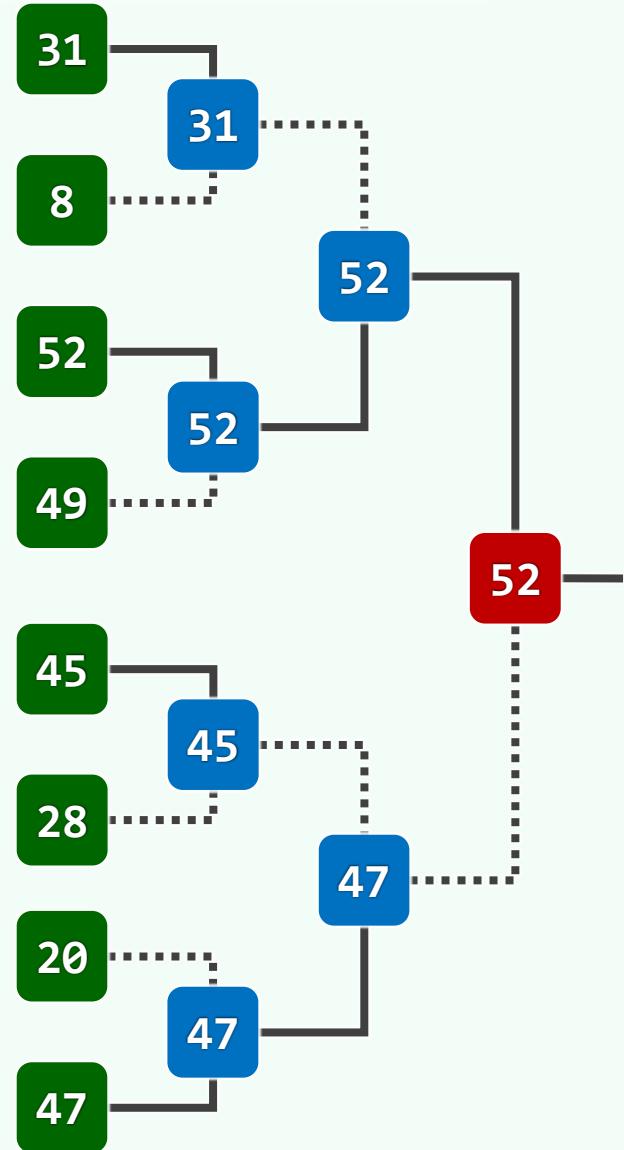


实例 (2/2)



效率

- ❖ 空间: $\mathcal{O}(\text{节点数}) = \mathcal{O}(\text{叶节点数}) = \mathcal{O}(n)$
- ❖ 构造: 仅需 $\mathcal{O}(n)$ 时间
- ❖ 更新: 每次都须全体重赛 replay?
上一优胜者的祖先们, 才有必要!
- ❖ 为此
如此 为确定各轮优胜者, 总共所需时间仅 $\mathcal{O}(\log n)$
- ❖ 时间: $n\text{轮} \times \mathcal{O}(\log n) = \mathcal{O}(n \log n)$, 达到下界



不完全选取

❖ 借助锦标赛树，从 n 个元素中找出最小的 k 个， $k \ll n$

- 初始话： $\Theta(n)$ // $n-1$ 次比较

- 迭代 k 步： $\Theta(k \cdot \log n)$ //每步 $\log n$ 次比较

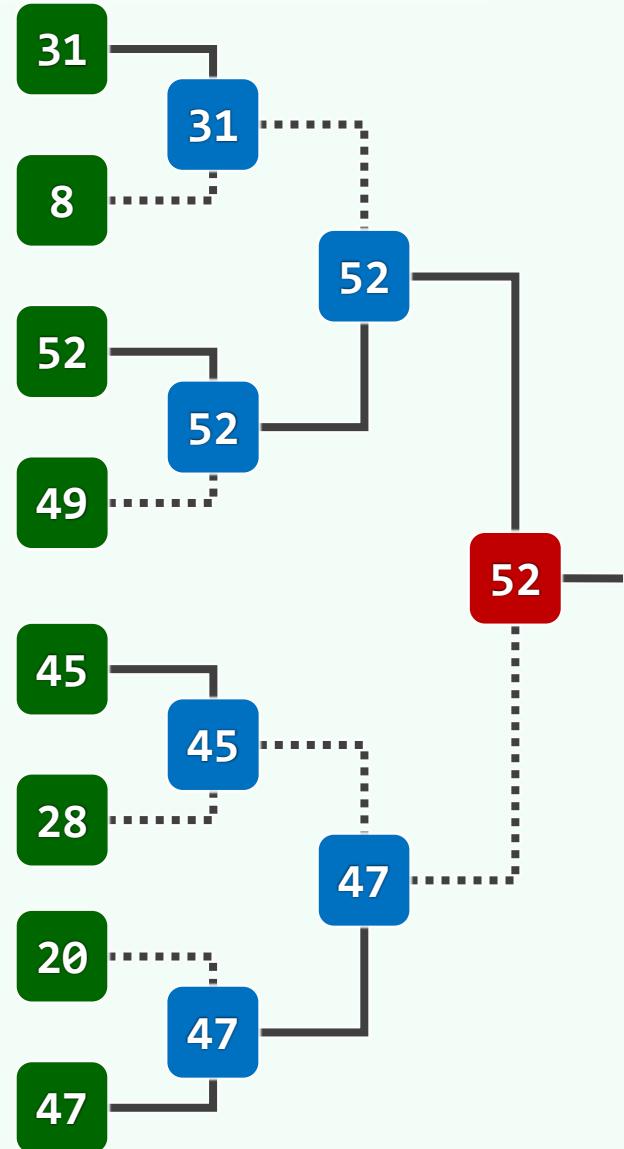
与小顶堆旗鼓相当？

❖ 渐近意义上，的确如此

但就常系数而言，区别不小...

❖ Floyd算法、delMax()中的percolateDown()

在每一层需做2次比较，累计大致 $2 \cdot \log n$ 次



优先级队列

锦标赛树：败者树

12-D2

We are the champions.

No time for losers

'Cause we are the champions of the world

善胜者不阵，善阵者不战，善战者不败，善败者终胜

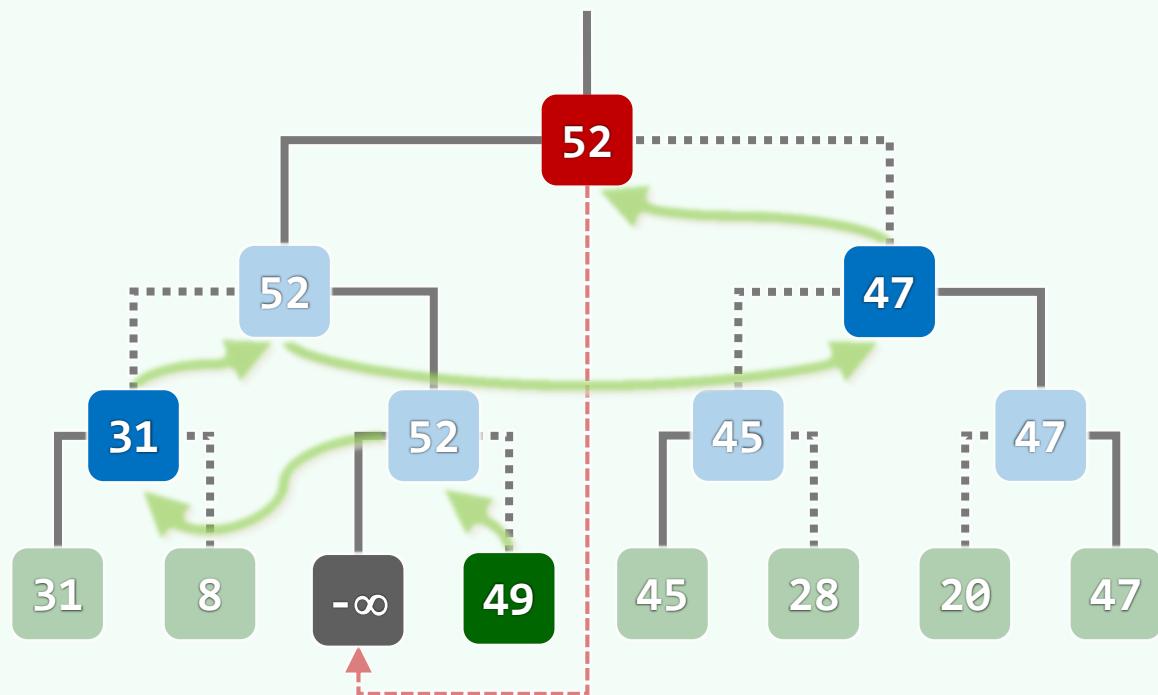
邓俊辉

deng@tsinghua.edu.cn

结构 + 重赛算法

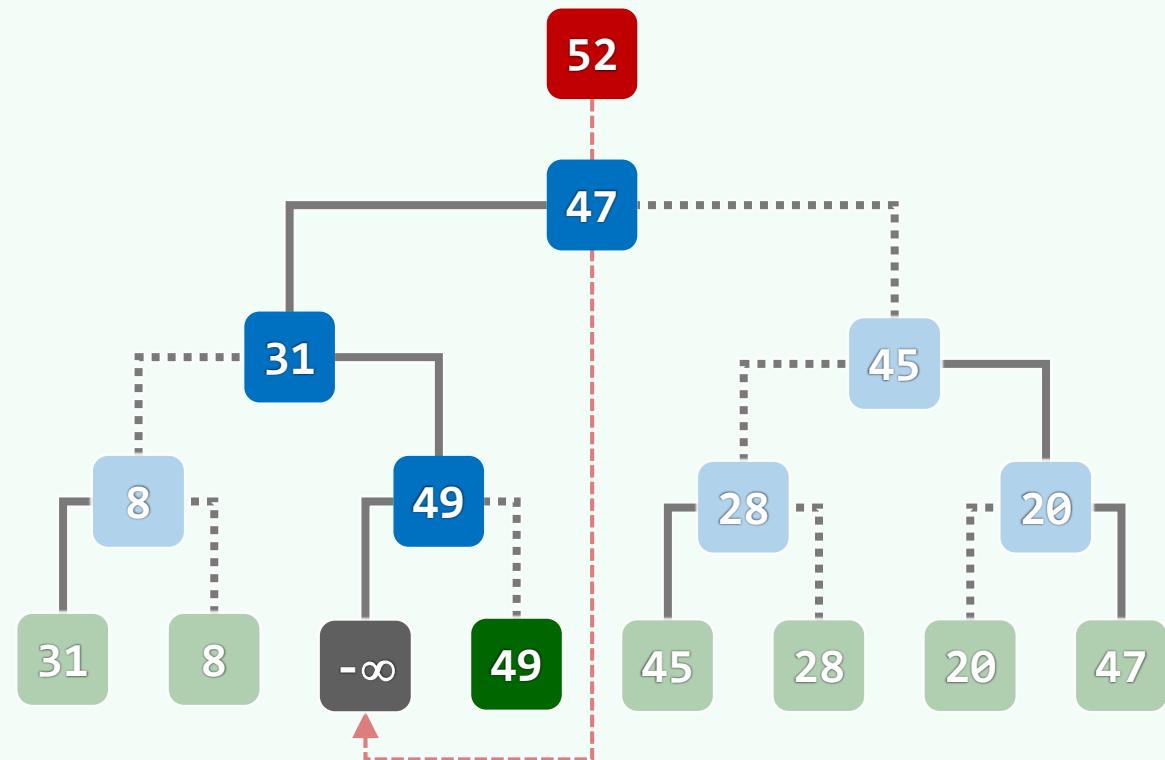
❖ 重赛过程中，须交替访问沿途节点及其兄弟

如何避免这类迂回？

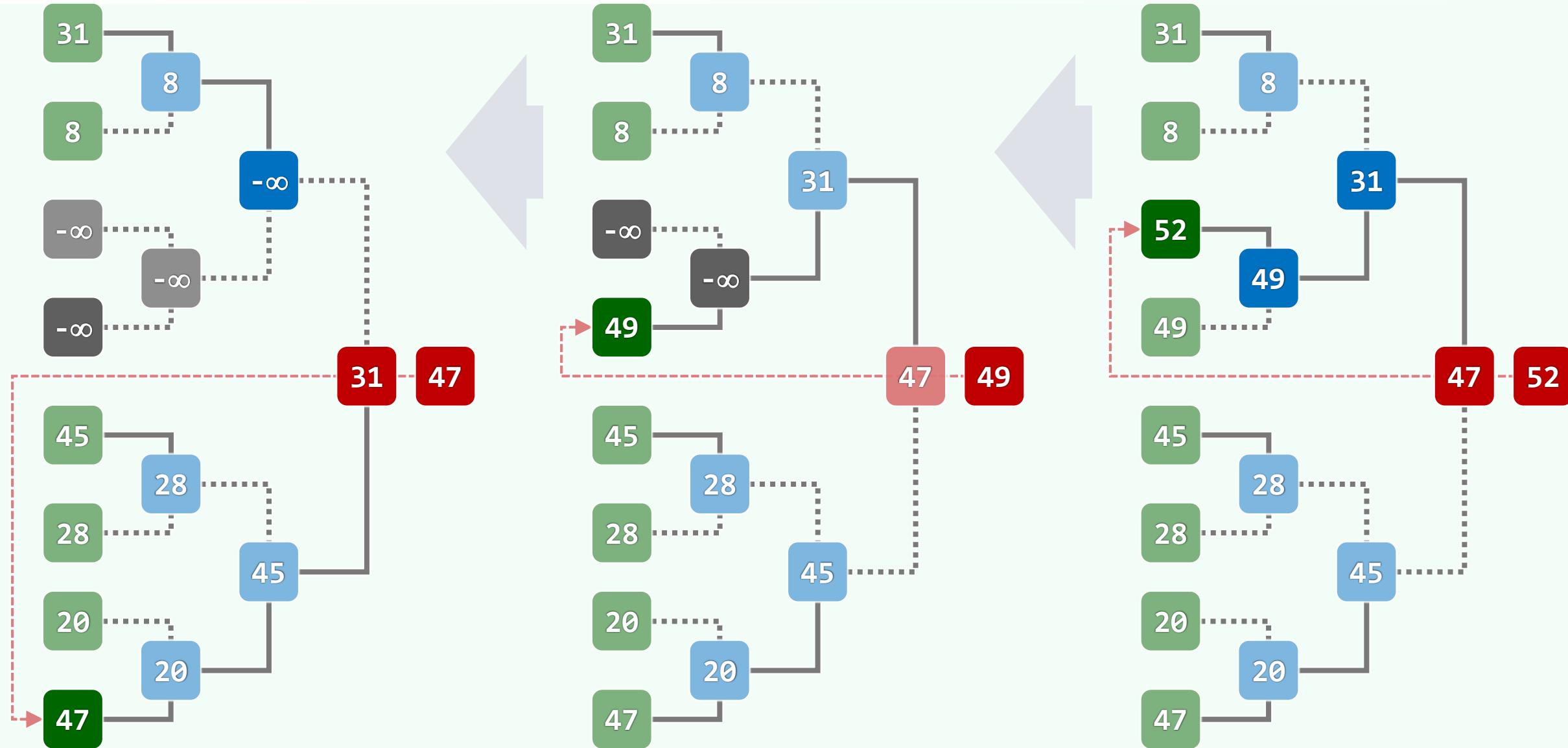


❖ 内部节点，记录对应比赛的败者

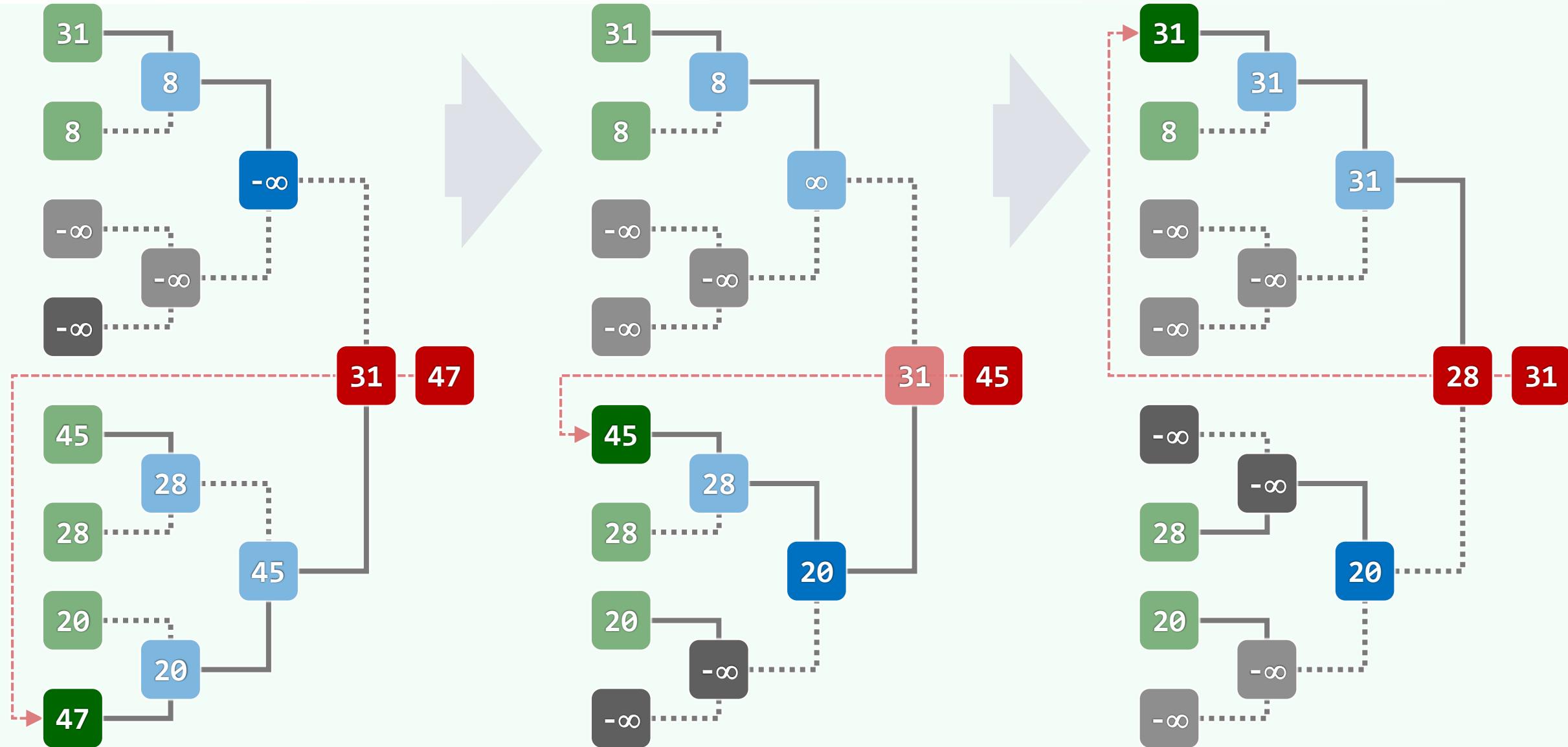
增设根的“父节点”，记录冠军



实例 (1/2)



实例 (2/2)



优先级队列

多叉堆

12-E

他说了一件事，时常萦绕我的心头，就是每个人若能窥清其他人心意，那么愿意下来的人会多于愿意高升的人

匏有苦葉，濟有深涉
深則厲，淺則揭

邓俊辉

deng@tsinghua.edu.cn

优先级搜索

- ❖ 回顾图的PFS以及统一框架: `g->pfs()`...
- ❖ 无论何种算法, 差异仅在于所采用的优先级更新器`prioUpdater()`
 - Prim算法: `g->pfs(0, PrimPU());`
 - Dijkstra算法: `g->pfs(0, DijkPU());`
- ❖ 每一节点引入遍历树后, 都需要
 - 更新树外顶点的优先级 (数), 并
 - 选出新的优先级最高者
- ❖ 若采用邻接表, 两类操作的累计时间, 分别为 $\mathcal{O}(n + e)$ 和 $\mathcal{O}(n^2)$...能否更快呢?

优先级队列

❖ 自然地，PFS中的各顶点可组织为**优先级队列**

❖ 为此需要使用PQ接口

heapify() ~ percolateDown() : 由 n 个顶点创建初始PQ, $\mathcal{O}(n)$

delMax() ~ percolateDown() : 取优先级最高（极短）跨边(u, w), $\mathcal{O}(n \cdot \log n)$

increase() ~ percolateUp() : 更新（缩短）所有关联顶点到 u 的距离, $\mathcal{O}(e \cdot \log n)$

❖ 总体运行时间 = $\mathcal{O}(n + e \cdot \log n)$

- 对于**稀疏图**, 处理效率很高
- 对于**稠密图**, 反而不如常规实现的版本

❖ 有无更好的办法?

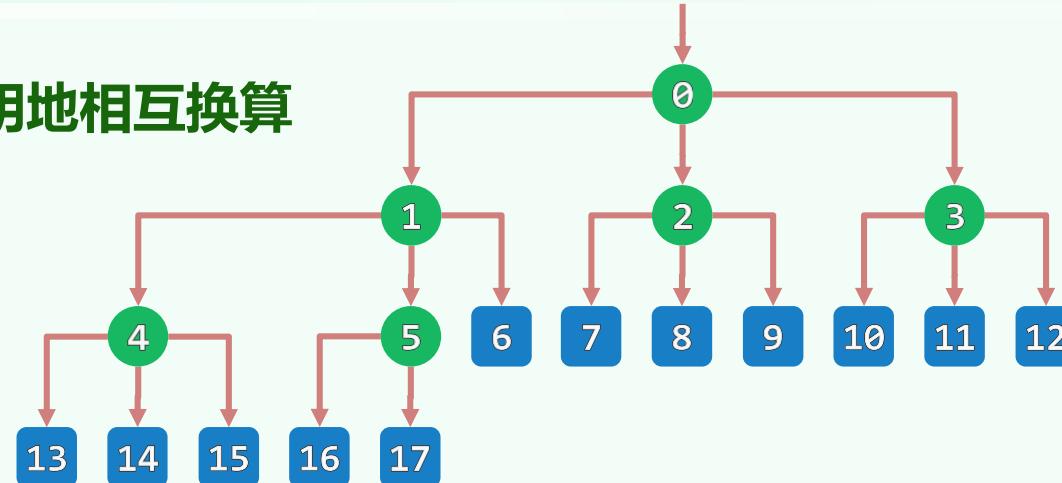
多叉堆

❖ 仍可基于向量实现，且父、子节点的秩可简明地相互换算

$$- \text{parent}(k) = \lfloor (k - 1)/d \rfloor$$

$$- \text{child}(k, i) = k \cdot d + i, 0 < i \leq d$$

// d 不是2的幂时，不能借助移位运算加速



❖ heapify(): $\theta(n)$

//不可能再快了



❖ delMax(): $\theta(\log n)$

//实质就是percolateDown()，已是极限了——为什么？

❖ increase(): $\theta(\log n)$

//实质就是percolateUp()——似乎仍有改进空间...

上山容易下山难

❖ 若将二叉堆改成多叉堆 (d-heap)

则堆高降至

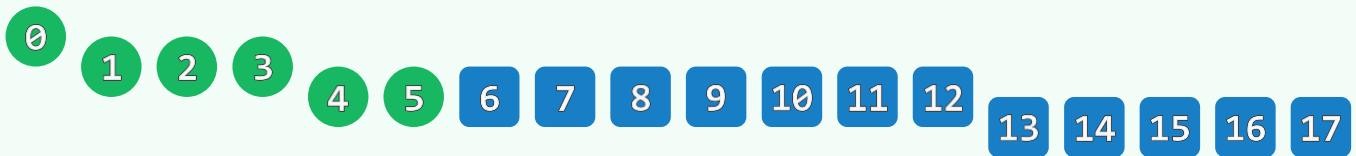
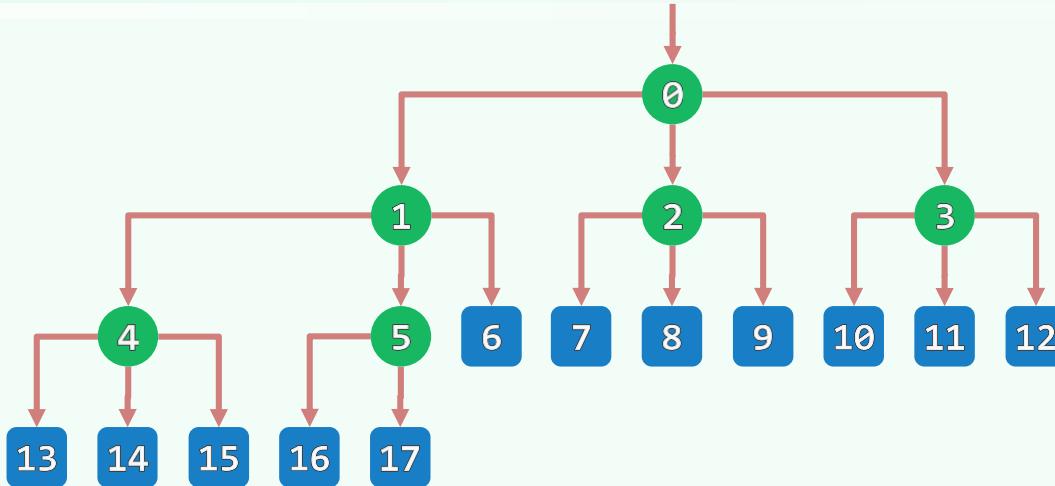
$$\log_d n$$

❖ 相应地，上滤成本降至

$$\log_d n$$

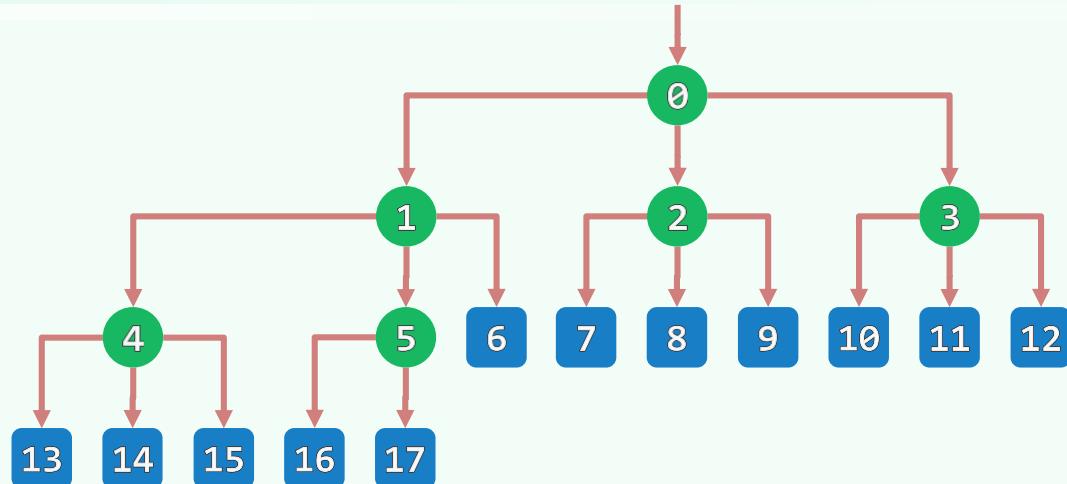
❖ 但 (只要 $d > 4$) 下滤成本却增至

$$d \cdot \log_d n = \frac{d}{\ln d} \cdot \ln n$$



❖ 如此，PFS的运行时间将是：

$$\begin{aligned} & n \cdot d \cdot \log_d n + e \cdot \log_d n \\ &= (n \cdot d + e) \cdot \log_d n \end{aligned}$$



❖ 取 $d \approx e/n + 2$ 时

总体性能达到最优： $\mathcal{O}(e \cdot \log_{(e/n+2)} n)$



❖ 对于稀疏图保持高效： $e \cdot \log_{(e/n+2)} n \approx n \cdot \log_{(n/n+2)} n = \mathcal{O}(n \log n)$

对于稠密图改进极大： $e \cdot \log_{(e/n+2)} n \approx n^2 \cdot \log_{(n^2/n+2)} n \approx n^2 = \mathcal{O}(e)$

对于一般的图，会自适应地实现最优

优先级队列

左式堆：沿藤合并

12-F1

God's right hand is gentle
But terrible is his left hand

邓俊辉

deng@tsinghua.edu.cn

堆合并

❖ $H = \text{merge}(A, B)$: 将堆A和B合二为一

//不妨设 $|A| = n \geq m = |B|$

❖ 方法一: $A.\text{insert}(B.\text{delMax}())$

$$m \cdot (\log m + \log(n+m))$$

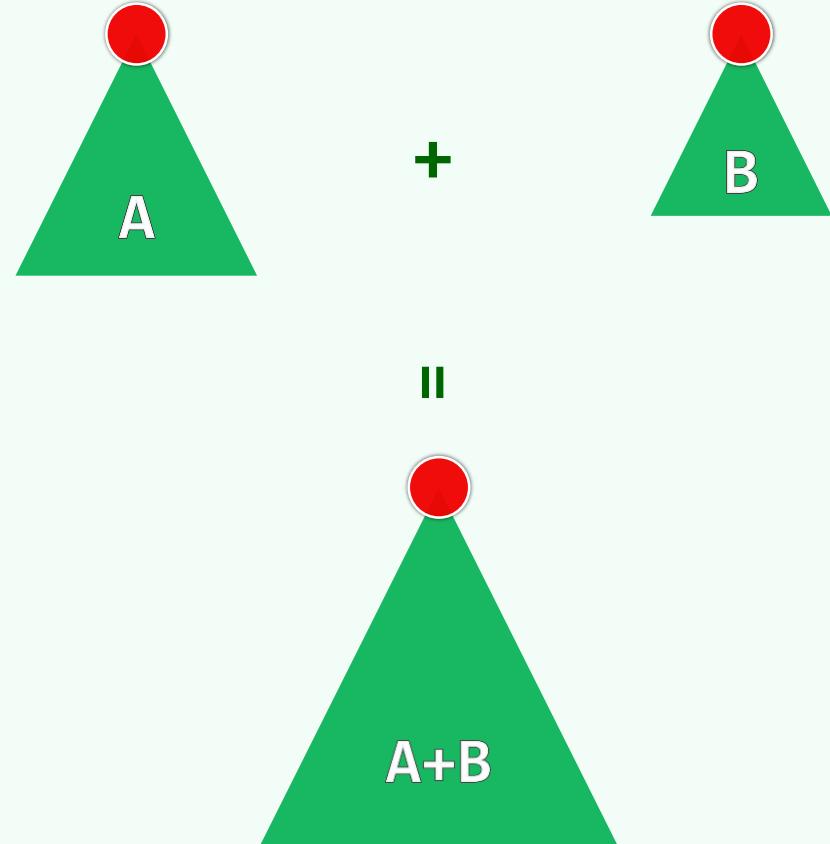
$$= \mathcal{O}(m \cdot \log(n+m))$$

❖ 方法二: $\text{union}(A, B).\text{heapify}(n + m)$

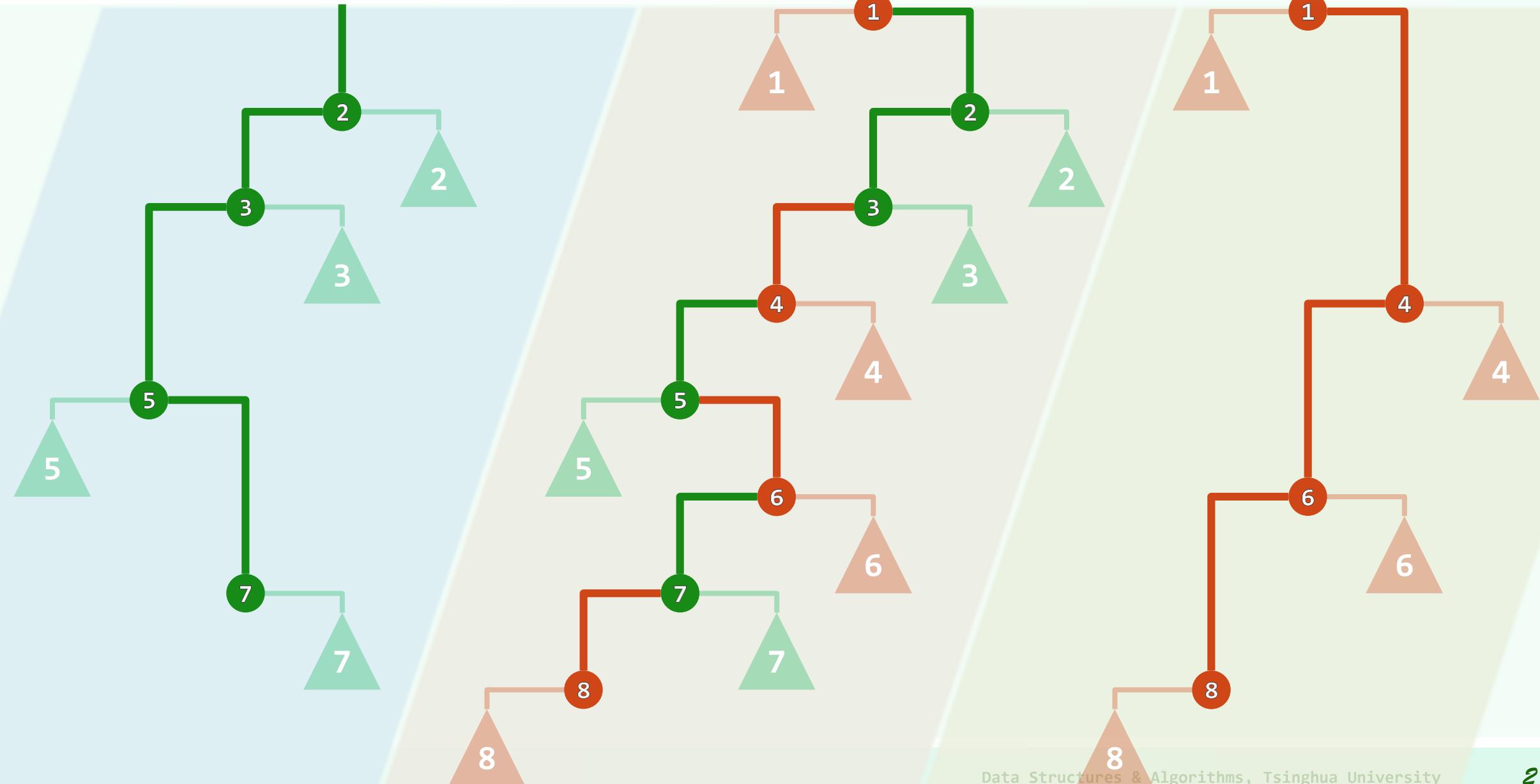
$$= \mathcal{O}(n+m)$$

❖ 有没有更好的办法? 比如...

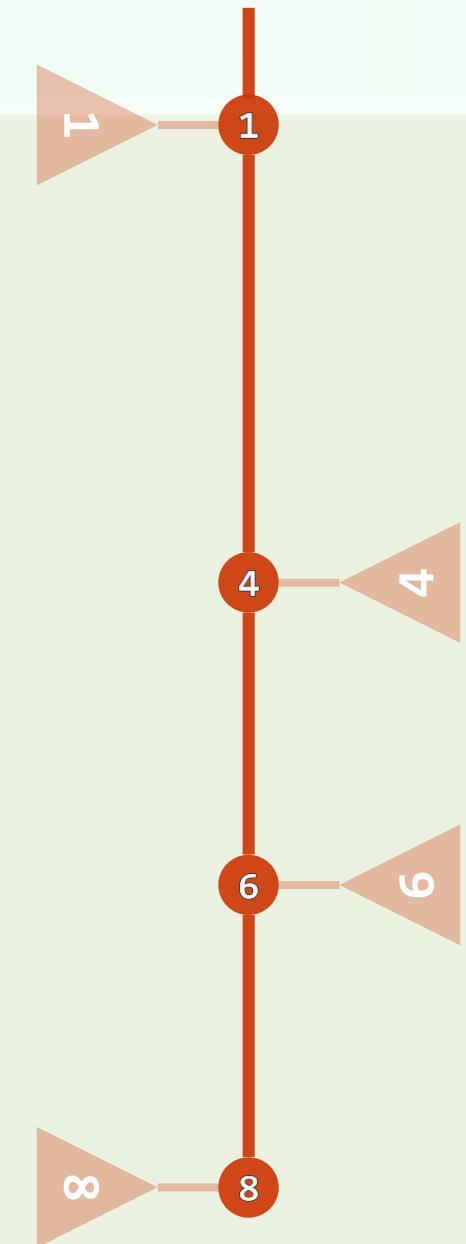
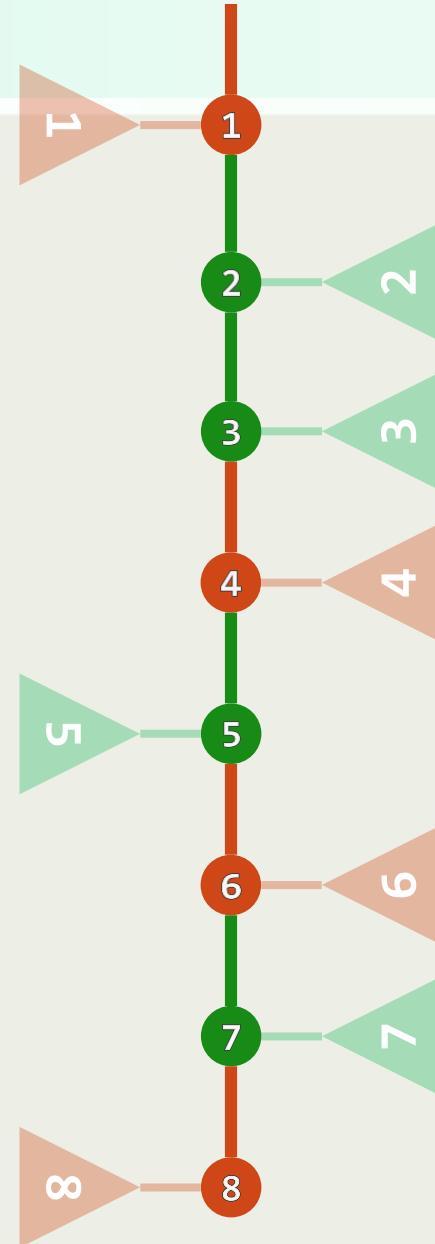
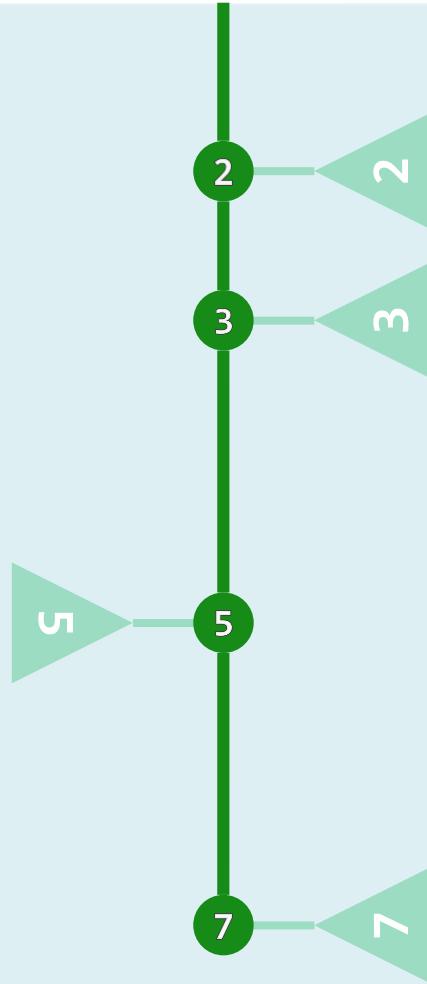
❖ 可否奢望在... $\mathcal{O}(\log n)$...时间内实现 merge()?



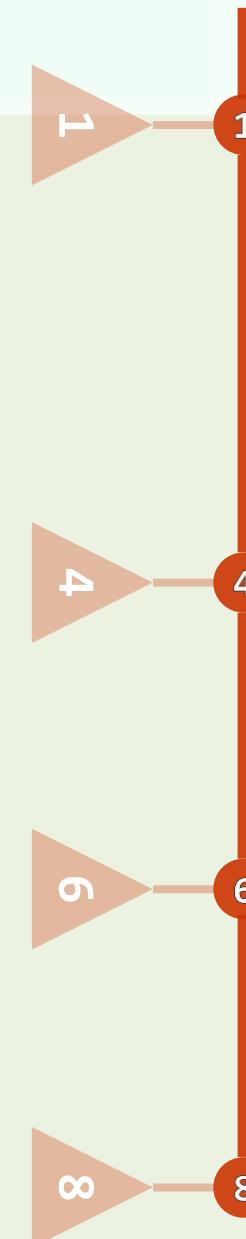
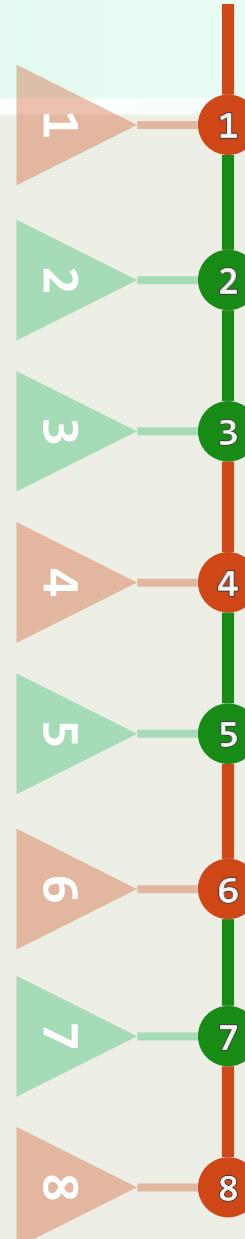
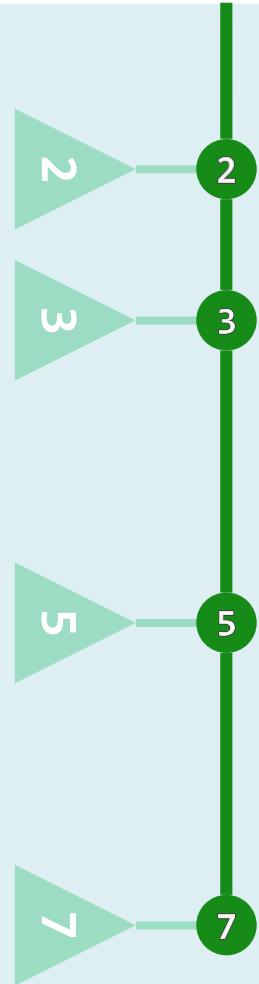
两堆合并 = 二路归并 (1/2)



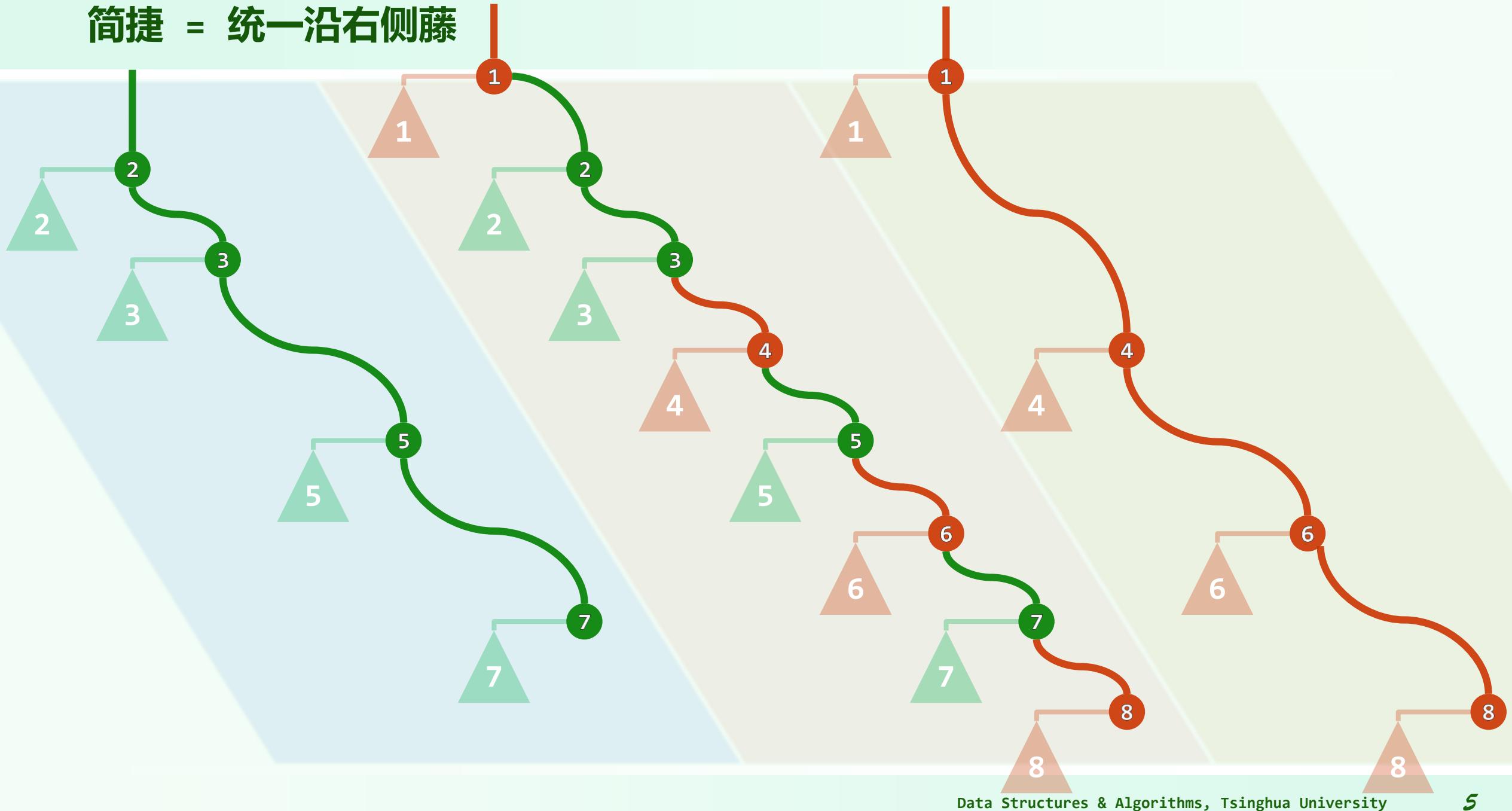
两堆合并 = 二路归并 (2/2)



简捷 = 统一右侧藤



简捷 = 统一沿右侧藤



LeftHeap = PQ + BinTree

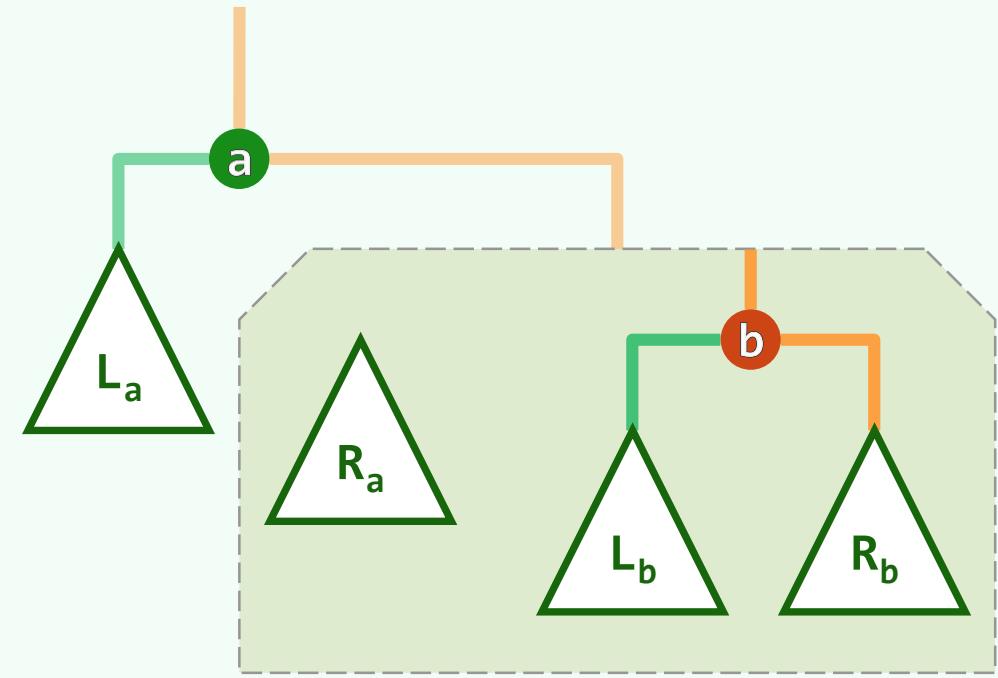
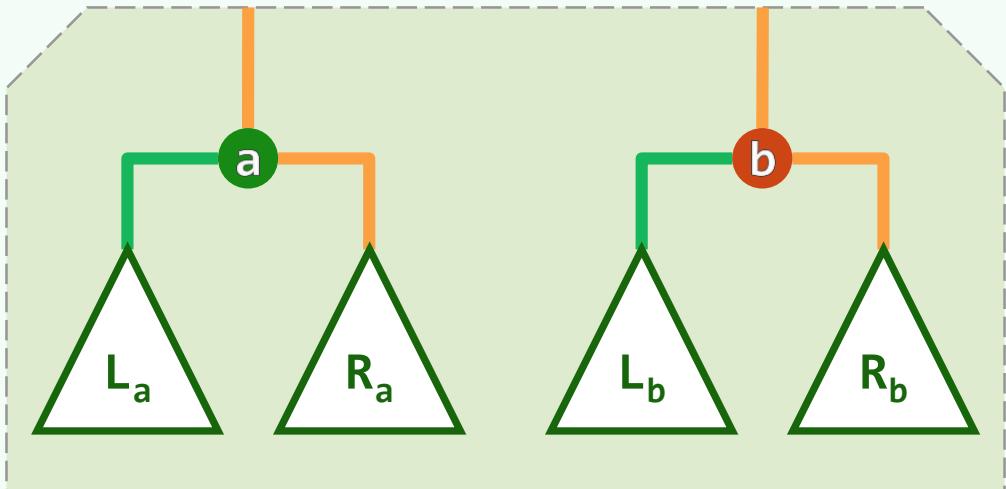
```
template <typename T> class PQ_LeftHeap : public PQ<T>, public BinTree<T> {  
  
public:  T & getMax() { return _root->data; }  
        void insert(T); T delMax(); //均基于统一的合并操作实现...  
  
PQ_LeftHeap( PQ_LeftHeap & A, PQ_LeftHeap & B ) {  
    _root = merge( A._root, B._root ); _size = A._size + B._size;  
    A._root = B._root = NULL; A._size = B._size = 0;  
}  
};  
  
template <typename T> BinNodePosi<T> merge( BinNodePosi<T>, BinNodePosi<T> );
```

递归实现

所需时间 \propto 右侧藤总长

$\cancel{O(n)}$

$O(\log n)$



如何...控制藤长
以...持续高效地合并?

优先级队列

左式堆：NPL与控制藤长

12-F2

君子居则贵左，用兵则贵右

邓俊辉

deng@tsinghua.edu.cn

可持续 = 单侧倾斜

❖ C. A. Crane, 1972:

保持堆序性，附加新条件，使得

在堆合并过程中，只涉及少量节点： $\mathcal{O}(\log n)$

❖ 新条件 = 单侧倾斜：

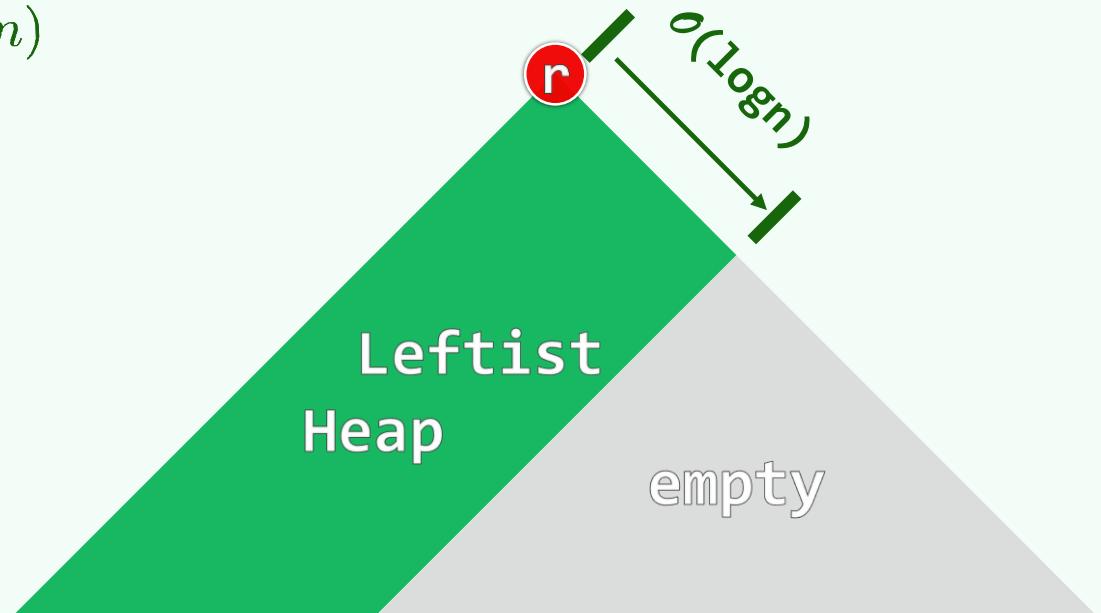
节点分布偏向于左侧

合并操作只涉及右侧

❖ 可是，果真如此，则拓扑上...

不见得是完全二叉树，结构性无法保证！？

❖ 是的，实际上，结构性并非堆结构的本质要求



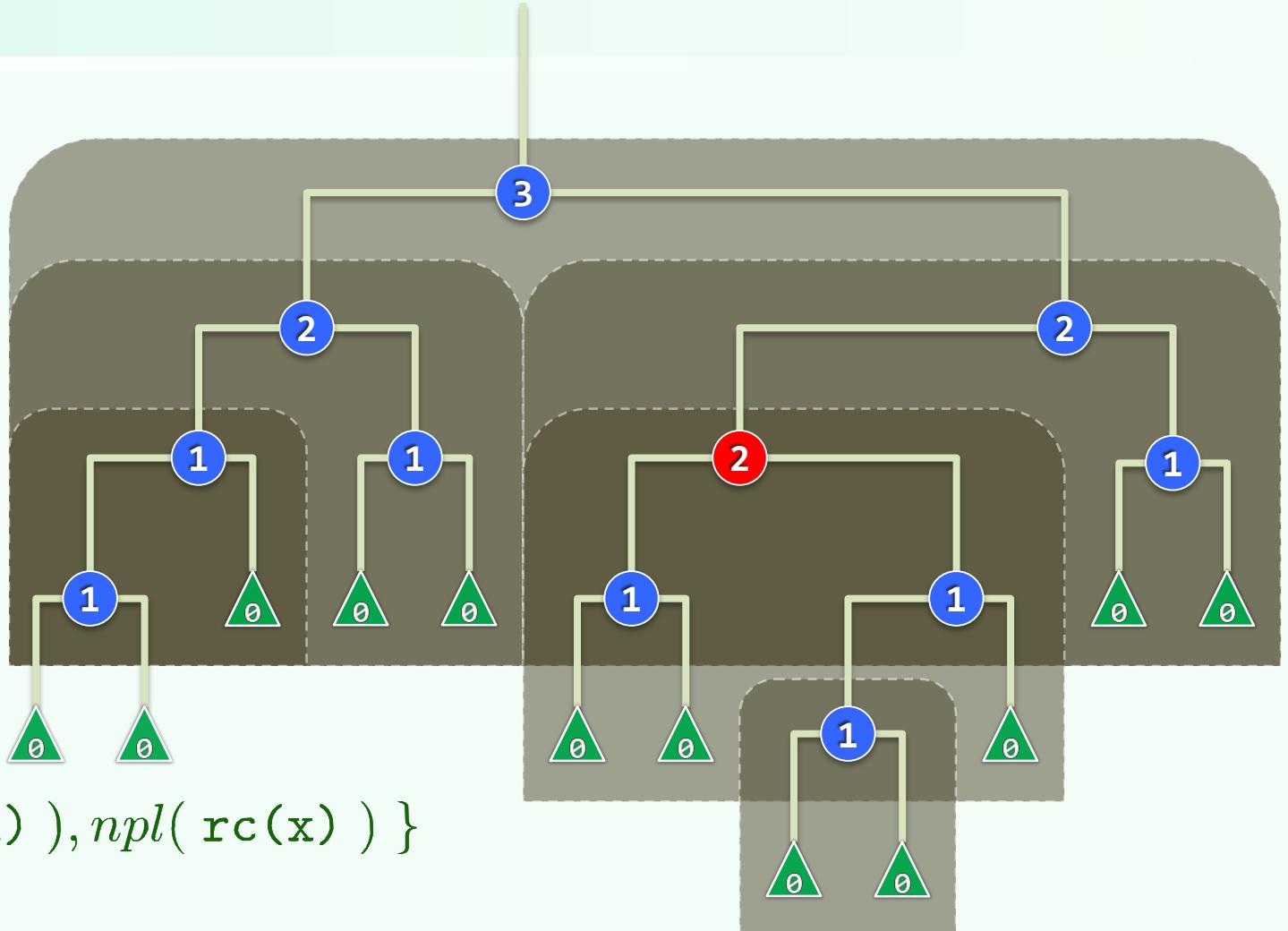
空节点路径长度

❖ 引入所有的外部节点

- 消除一度节点
- 转为真二叉树

❖ Null Path Length

- $npl(\text{NULL}) = 0$
- $npl(x) = 1 + \min\{ npl(\text{lc}(x)), npl(\text{rc}(x)) \}$



❖ 验证: $npl(x) = x$ 到外部节点的最近距离 = 以x为根的最大满子树的高度

左式堆 = 处处左倾

❖ 对任何内节点x，都有：

$$npl(\text{lc}(x)) \geq npl(\text{rc}(x))$$

❖ 推论：

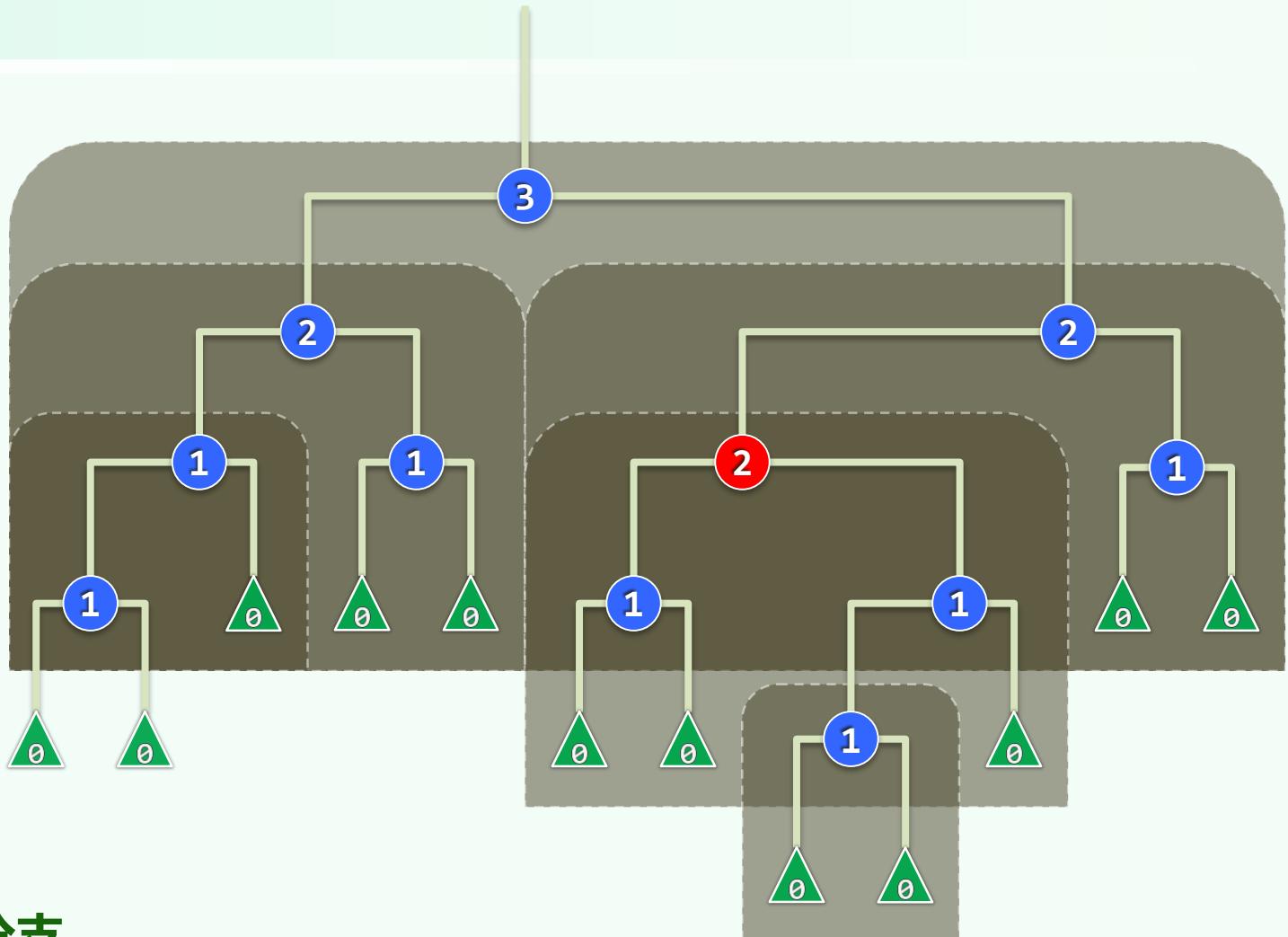
$$npl(x) = 1 + npl(\text{rc}(x))$$

❖ 左倾性与堆序性，相容而不矛盾

❖ 左式堆的子堆，必是左式堆

❖ 左式堆倾向于更多节点分布于左侧分支

这是否意味着，左子堆的规模和高度必然大于右子堆？



右侧链

❖ $rChain(x)$: 从节点 x 出发, 一直沿右分支前进

❖ 特别地, $rChain(r)$ 的终点, 即全堆中最浅的外部节点

- $npl(r) \equiv |rChain(r)| = d$

- 存在一棵以 r 为根、高度为 d 的满子树

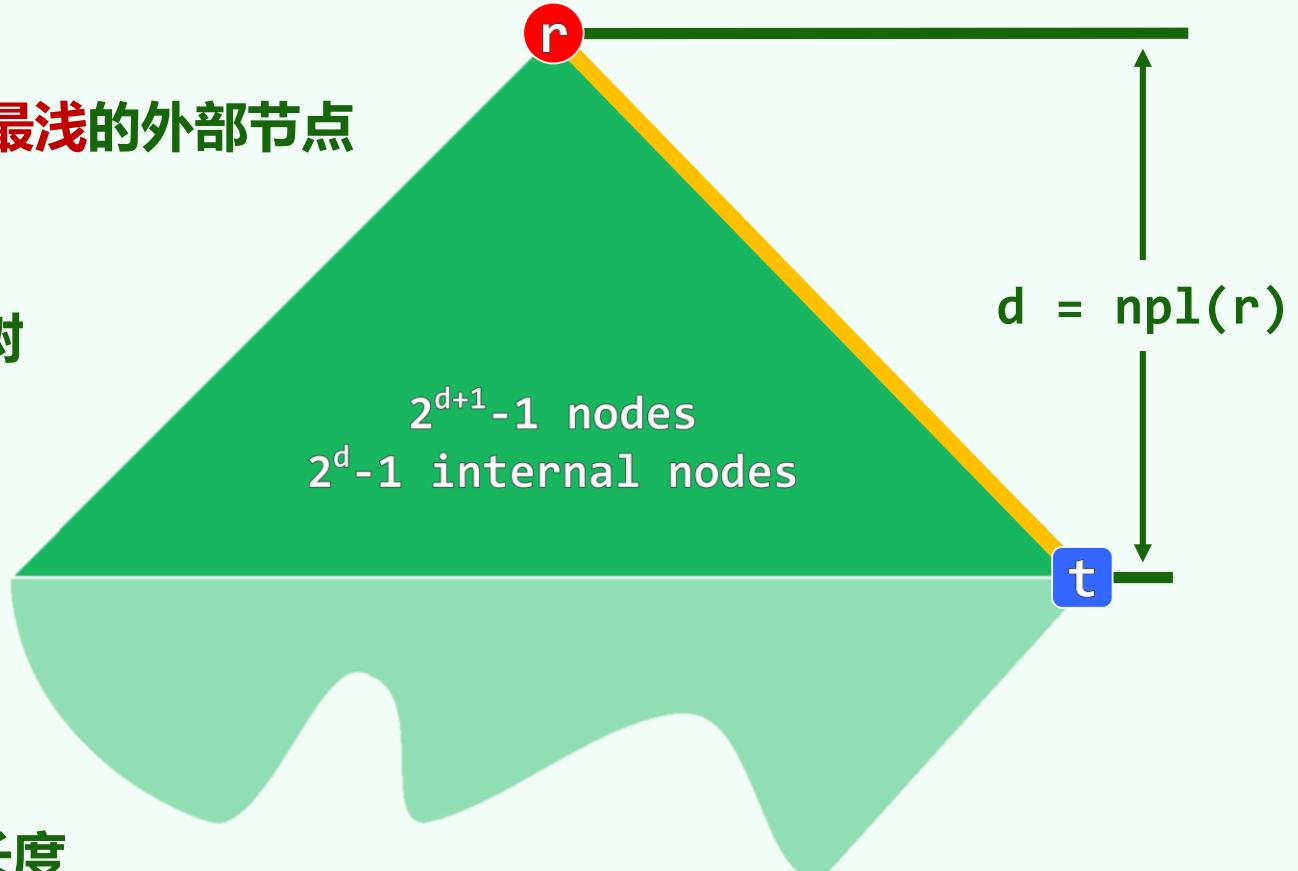
❖ 右侧链长为 d 的左式堆, 至少包含

- $2^d - 1$ 个内部节点

- $2^{d+1} - 1$ 个节点

❖ 反之, 包含 n 个节点的左式堆, 右侧链长度

$$d \leq \lfloor \log_2 (n + 1) \rfloor - 1 = \mathcal{O}(\log n)$$



优先级队列

左式堆：合并算法

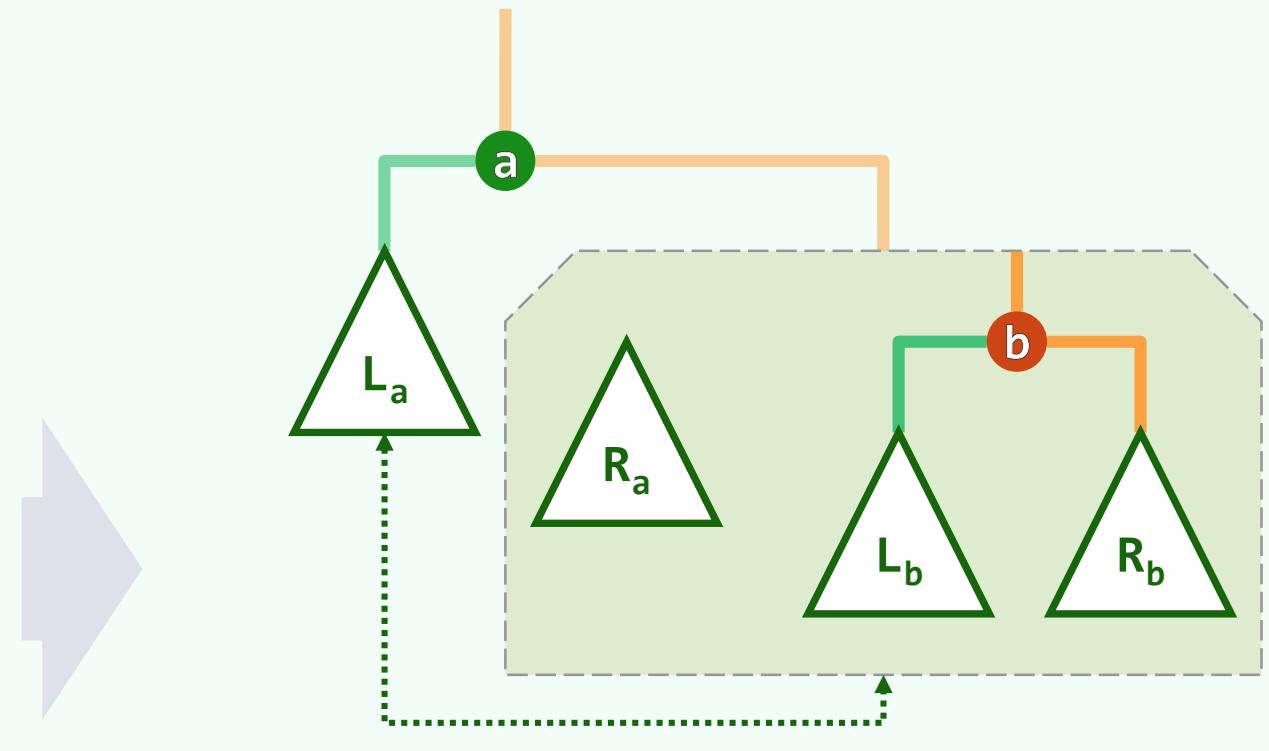
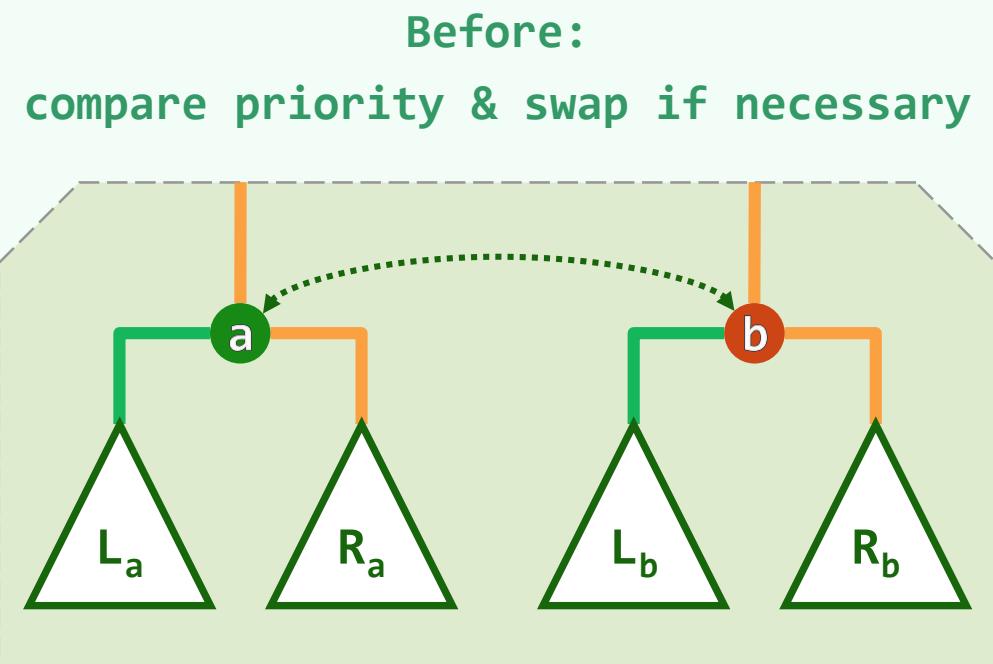
12-F3

邓俊辉

deng@tsinghua.edu.cn

左之左之，君子宜之；右之右之，君子有之

递归：前处理 + 后处理

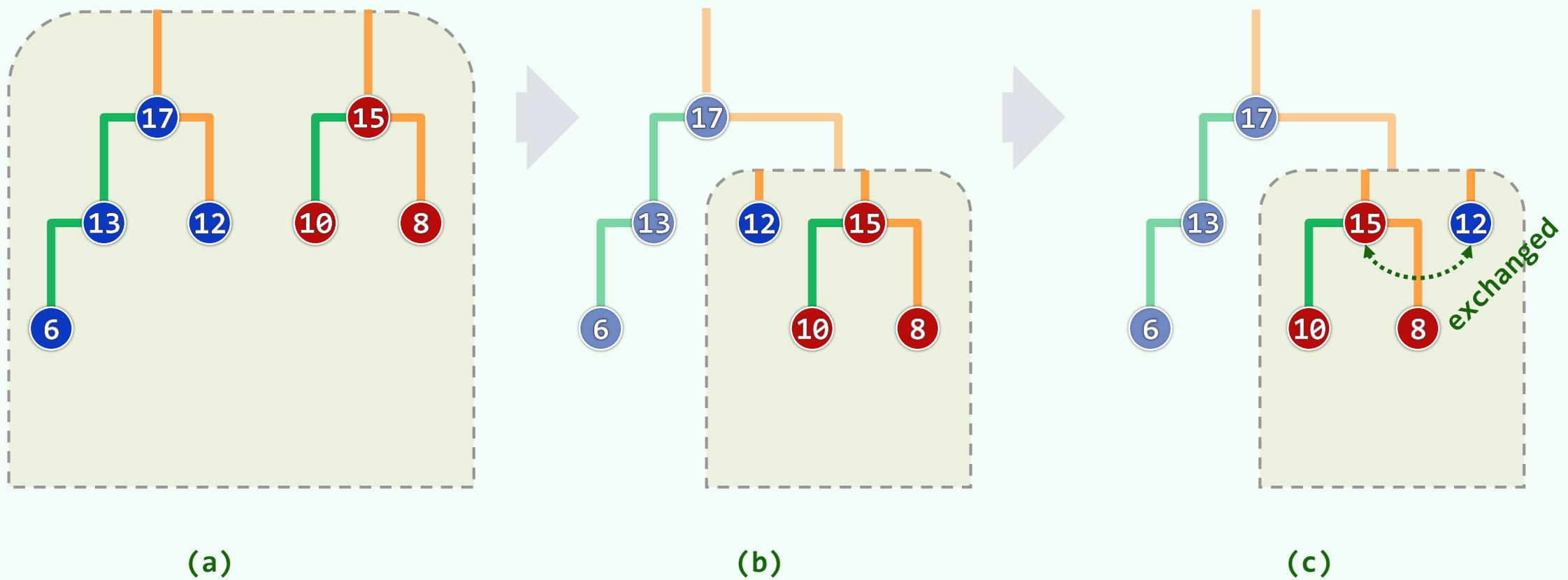


After:
compare NPL & flip if necessary

递归实现

```
template <typename T> BinNodePosi<T> merge( BinNodePosi<T> a, BinNodePosi<T> b ) {  
  
    if ( !a ) return b; if ( !b ) return a; //递归基  
    if ( *a < *b ) swap( a, b ); //确保a>=b: BinNode已重载定义过 “<”  
  
    ( a->rc = merge( a->rc, b ) )->parent = a; //将a的右子堆，与b合并  
  
    if ( !a->lc || (a->lc->npl < a->rc->npl) ) //若有必要  
        swap( a->lc, a->rc ); //交换a的左、右子堆，以确保左子堆的npl不小  
    a->npl = a->rc ? 1 + a->rc->npl : 1; //更新a的npl  
  
    return a; //返回合并后的堆顶  
}
```

实例 (1/5)

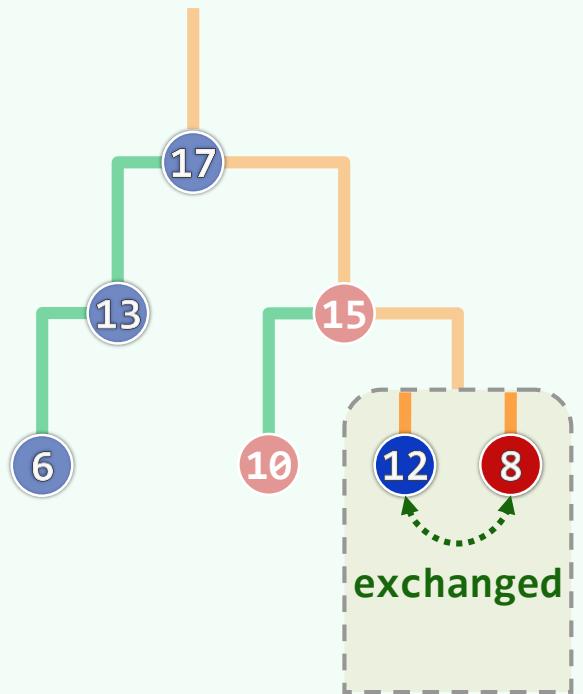


(a)

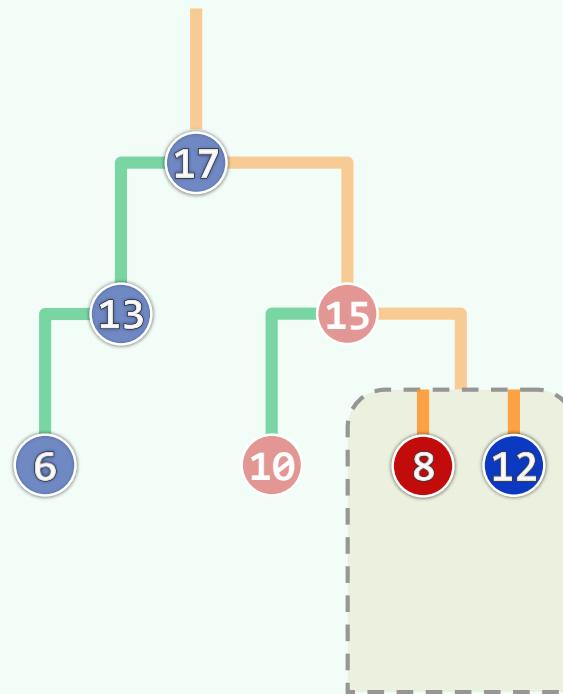
(b)

(c)

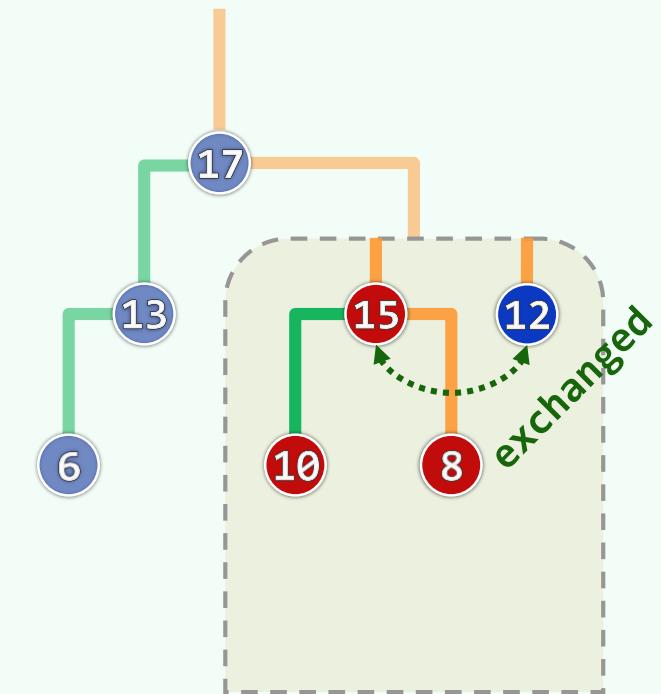
实例 (2/5)



(e)

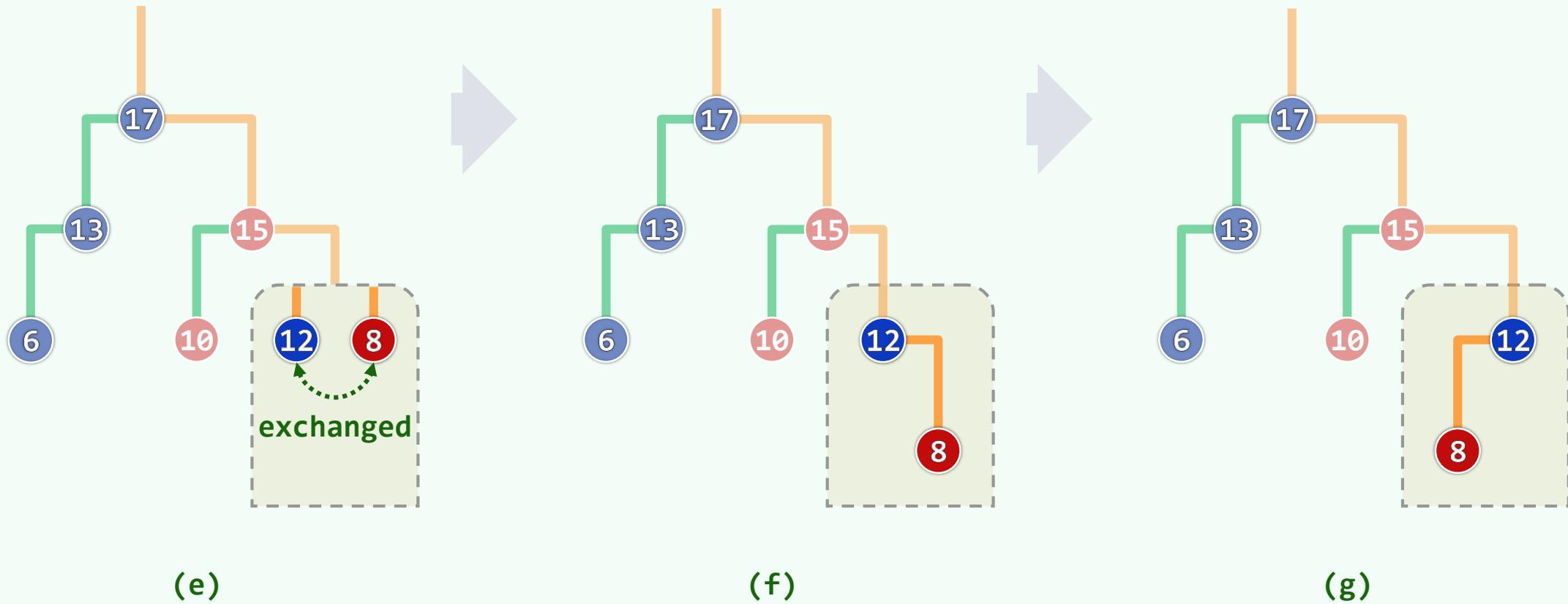


(d)

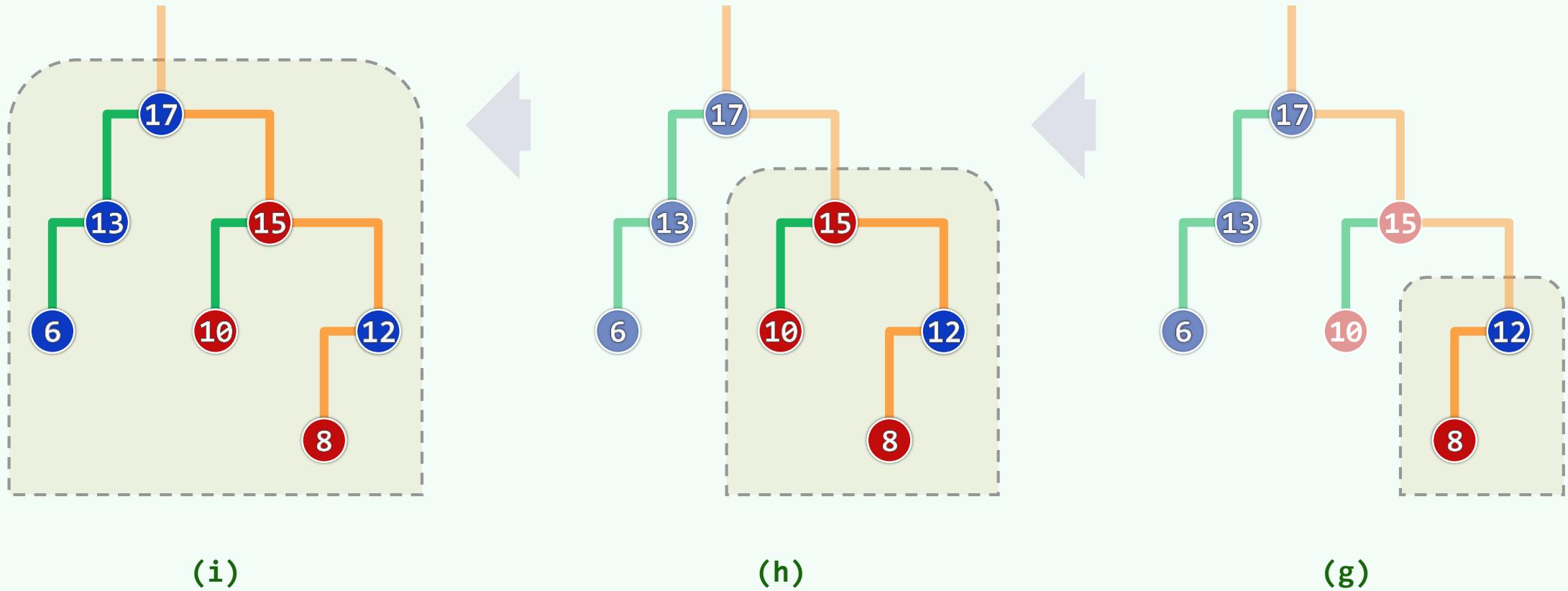


(c)

实例 (3/5)



实例 (4/5)

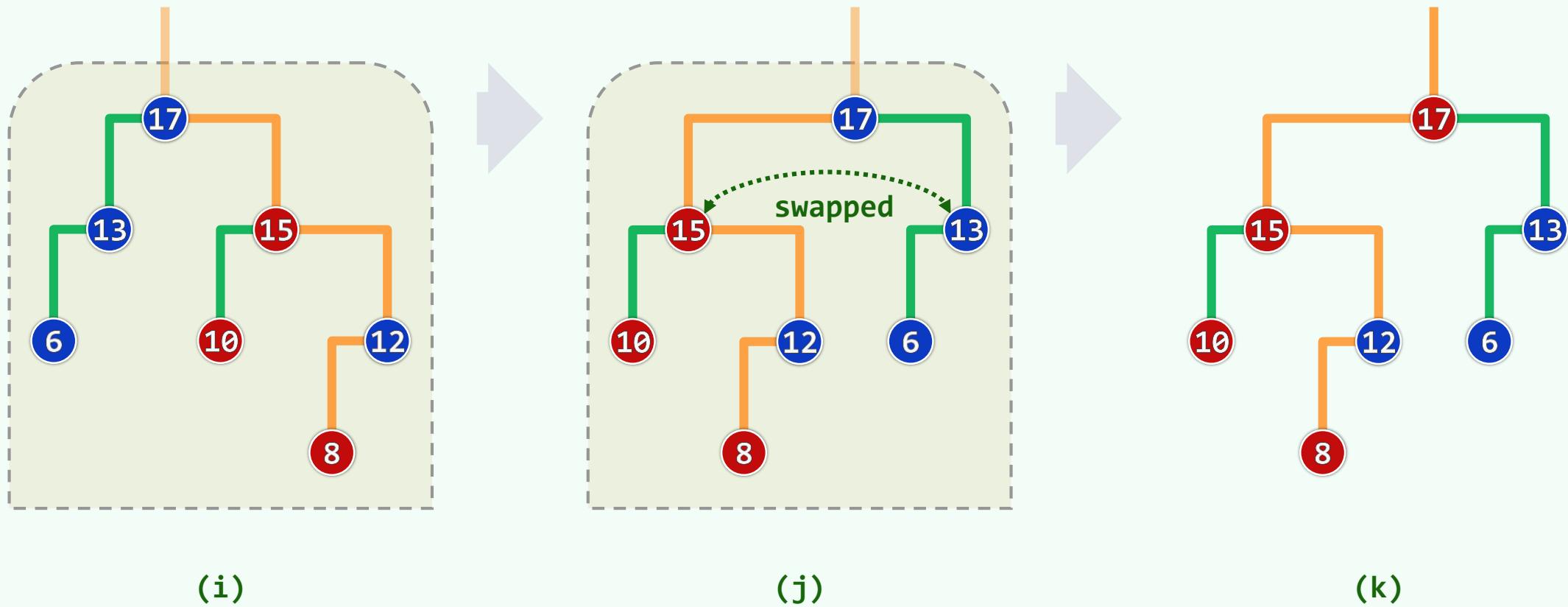


(i)

(h)

(g)

实例 (5/5)



迭代实现

```
template <typename T> BinNodePosi<T> merge( BinNodePosi<T> a, BinNodePosi<T> b ) {  
    if ( !a ) return b; if ( !b ) return a; if ( *a < *b ) swap( a, b );  


---

for ( ; a->rc; a = a->rc ) //沿右侧链做二路归并，直至堆a->rc先于b变空  
    if ( *(a->rc) < *b ) { b->parent = a; swap( a->rc, b ); } //接入b  
(a->rc = b)->parent = a; //直接接入b的残余部分（必然非空）  


---

for ( ; a; b = a, a = a->parent ) { //从a出发沿右侧链逐层回溯 (b == a->rc)  
    if ( !a->lc || (a->lc->npl < a->rc->npl) ) swap( a->lc, a->rc ); //确保npl合法  
    a->npl = a->rc ? a->rc->npl + 1 : 1; //更新npl  
}  


---

return b; //返回合并后的堆顶
```

优先级队列

左式堆：插入与删除

12-F4

一招鲜，吃遍天

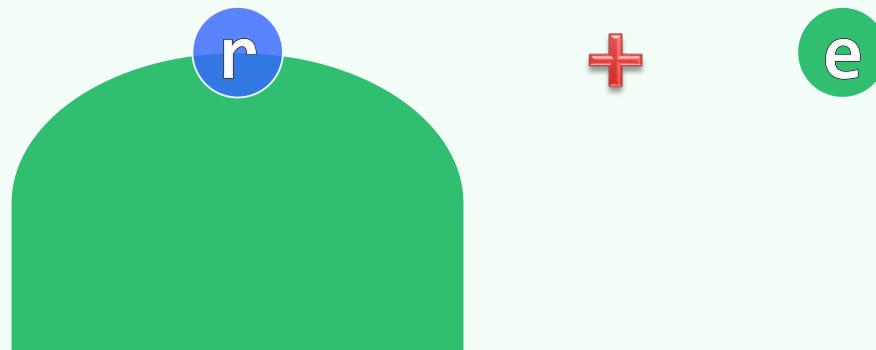
良田千顷，不如薄技在身

邓俊辉

deng@tsinghua.edu.cn

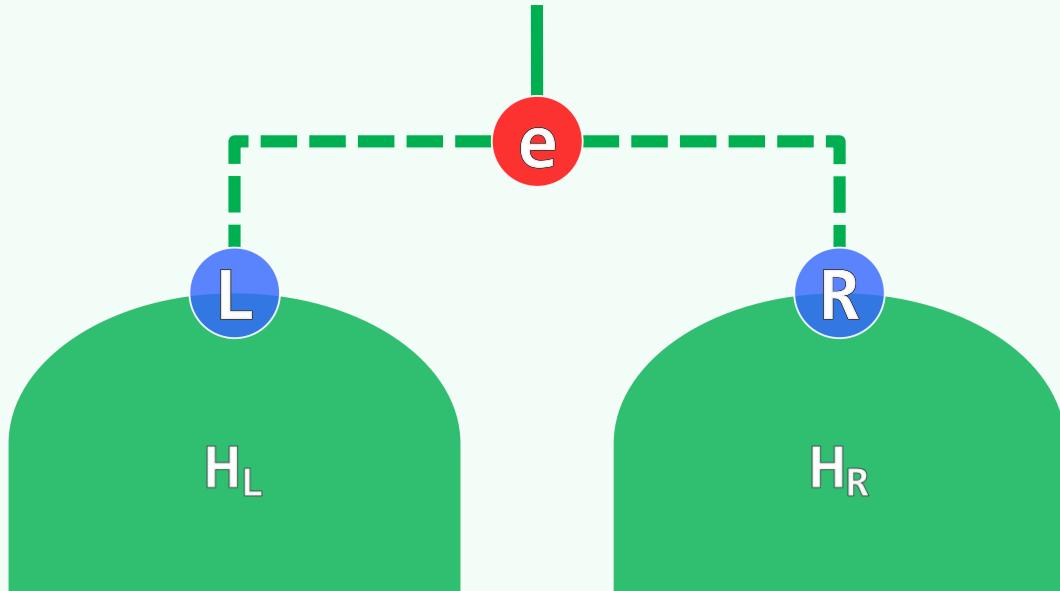
插入

```
template <typename T> void PQ_LeftHeap<T>::insert( T e ) { //O(logn)  
    _root = merge( _root, new BinNode<T>( e, NULL ) );  
  
    _size++;  
}
```



删除

```
template <typename T> T PQ_LeftHeap<T>::delMax() { //O(logn)
    BinNodePosi<T> lHeap = _root->lc; if (lHeap) lHeap->parent = NULL;
    BinNodePosi<T> rHeap = _root->rc; if (rHeap) rHeap->parent = NULL;
    T e = _root->data;
    delete _root; _size--;
    _root = merge( lHeap, rHeap );
    return e;
}
```



12-XA

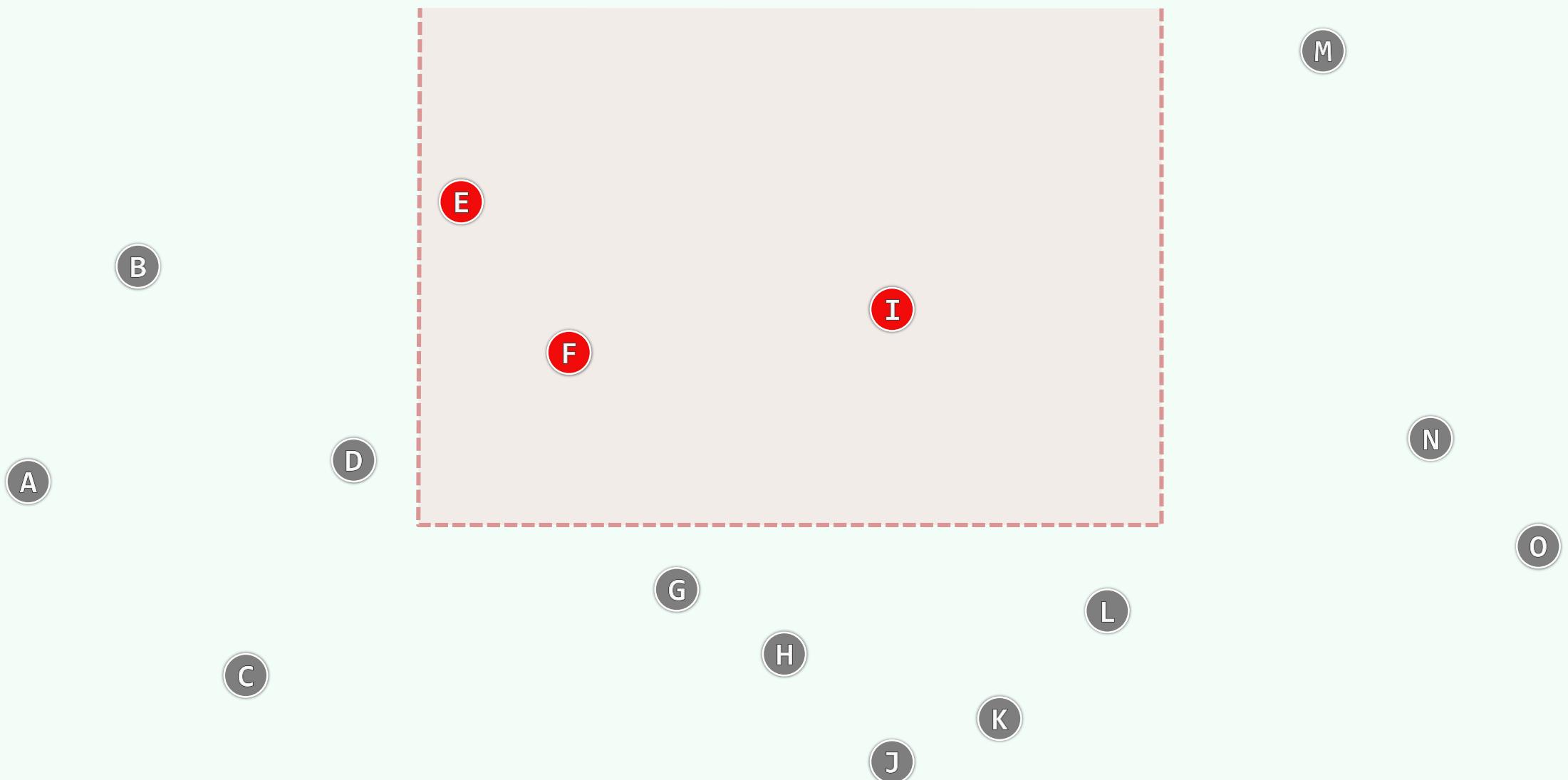
优先级队列

优先级搜索树

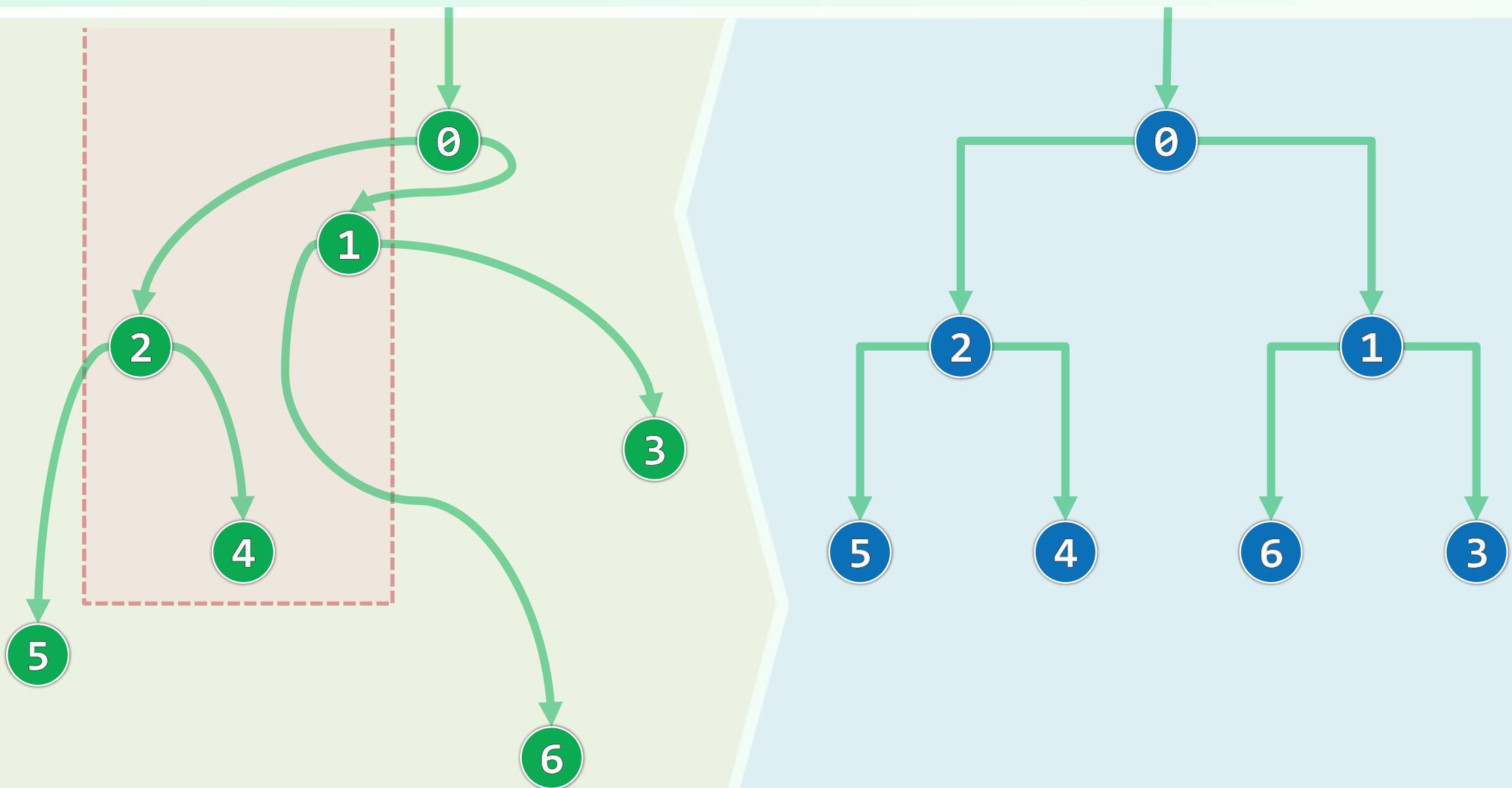
邓俊辉

deng@tsinghua.edu.cn

Grounded Range Query

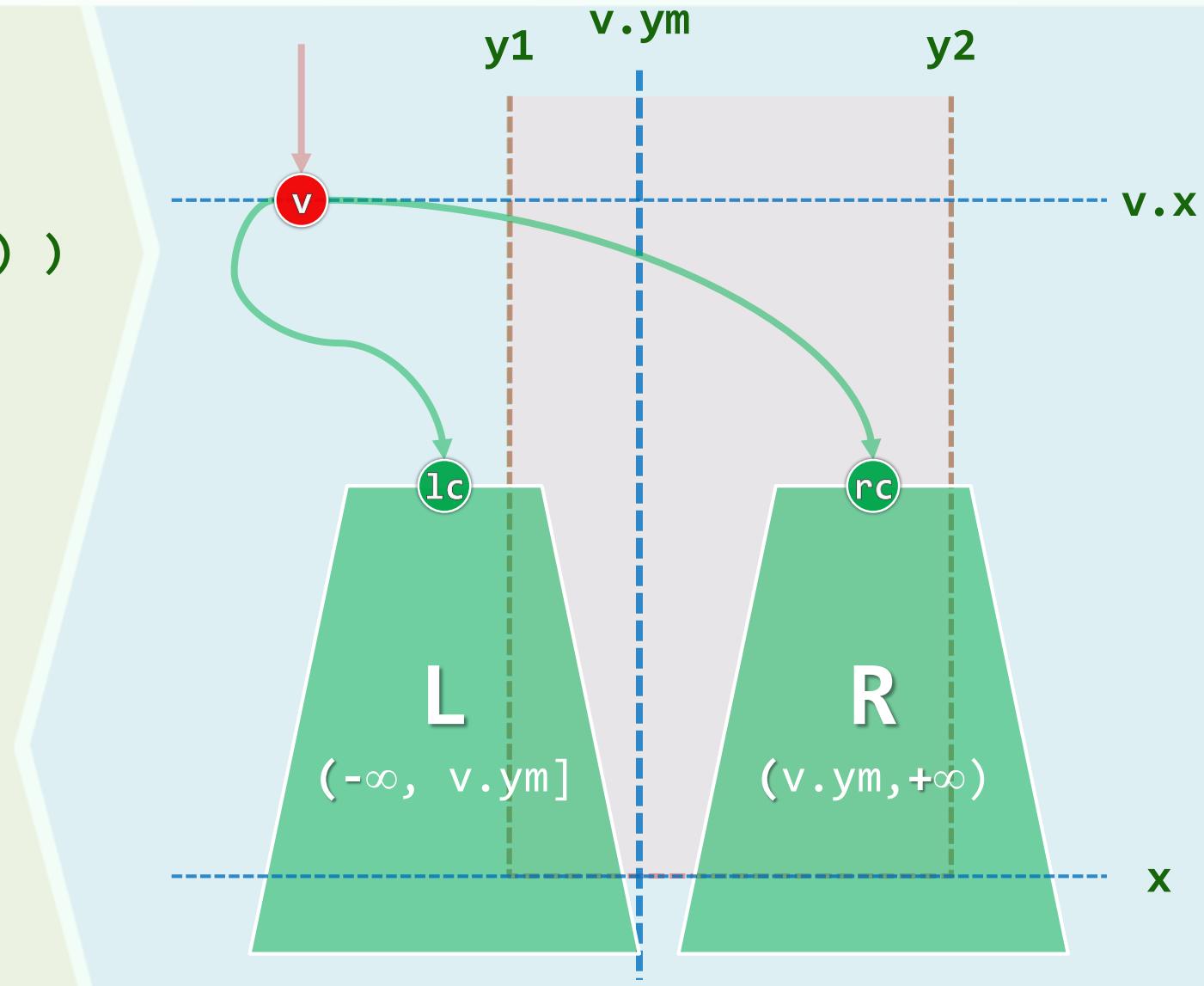


Priority Search Tree = BST + PQ

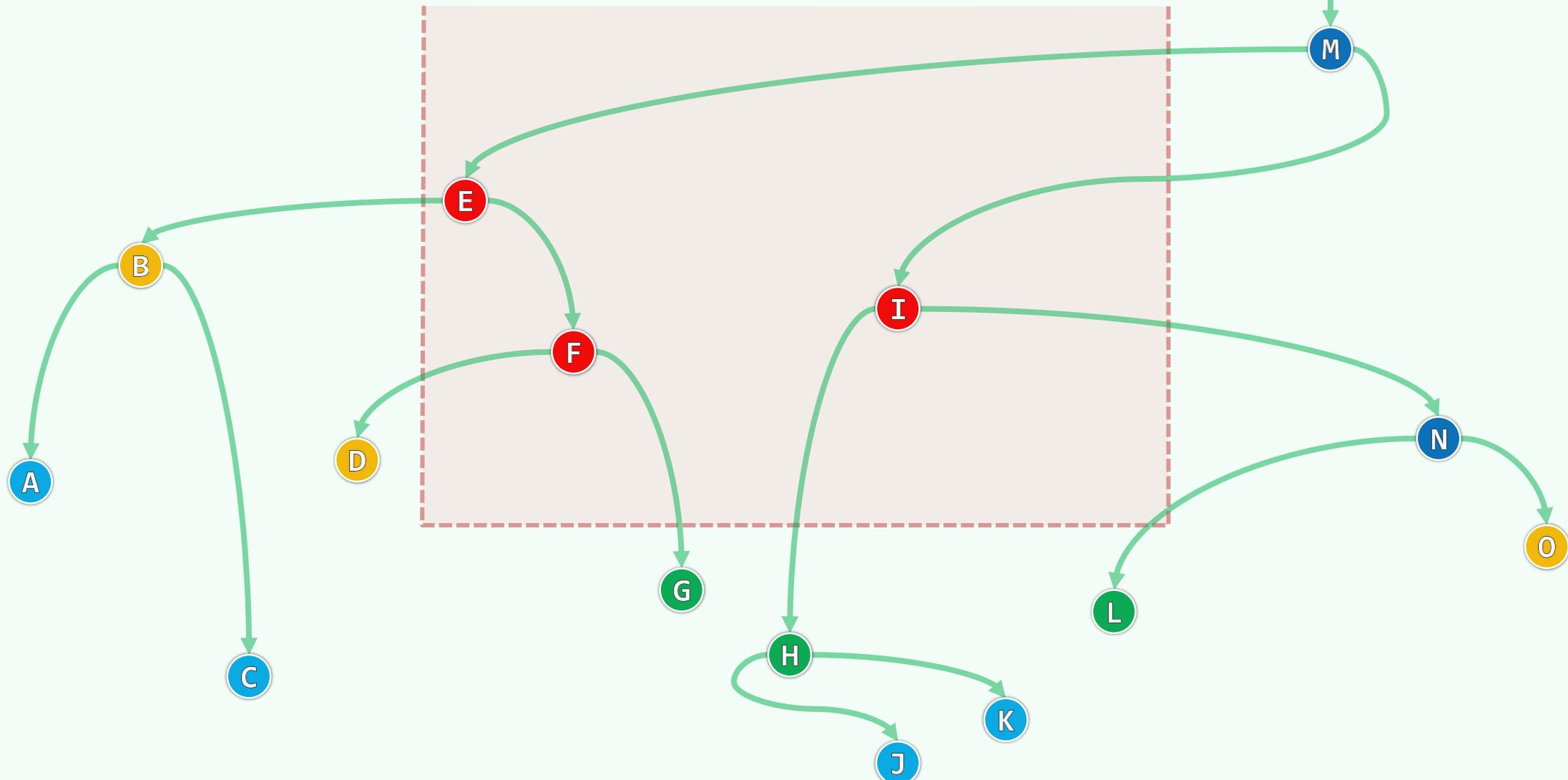


queryPST(PSTNode v, int x, int y1, int y2)

```
if ( !v || x < v.x )
    return //pruned for bad X
if ( ( y1 < v.y ) && ( v.y < y2 ) )
    output(v) //accepted
//else rejected for bad Y
if ( y1 ≤ v.ym )
    queryPST( v.lc, x, y1, y2 )
//else pruned for bad Y
if ( v.ym < y2 )
    queryPST( v.rc, x, y1, y2 )
//else pruned for bad Y
```



Example



Query Time

P: Pruned with descendants due to bad Y

- no more time cost

A: Visited and accepted

- exactly $r = \text{output size}$

BY: Visited but rejected due to bad Y

- no more than 2 for each level
- altogether $\mathcal{O}(\log n)$

BX: Visited but rejected due to bad X

- having an A or BY parent
- no more than $\mathcal{O}(r + \log n)$

