

10. 优先级队列

(b) 完全二叉堆

逊问曰：“何人将乱石作堆？
如何乱石堆中有杀气冲起？”

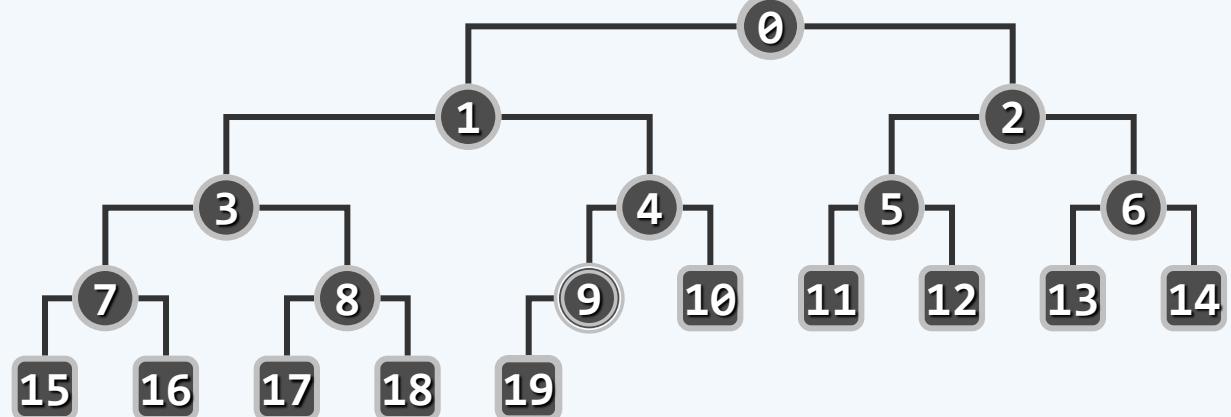
邓俊辉

deng@tsinghua.edu.cn

结构性

❖ 拓扑结构

等同于完全二叉树



❖ 借助向量实现，无需指针或引用

节点与元素，依层次遍历次序相互对应



```
#define Parent( i ) ( ( i - 1 ) >> 1 )
```

```
#define LChild( i ) ( 1 + ( ( i ) << 1 ) ) //奇数
```

```
#define RChild( i ) ( ( 1 + ( i ) ) << 1 ) //偶数
```

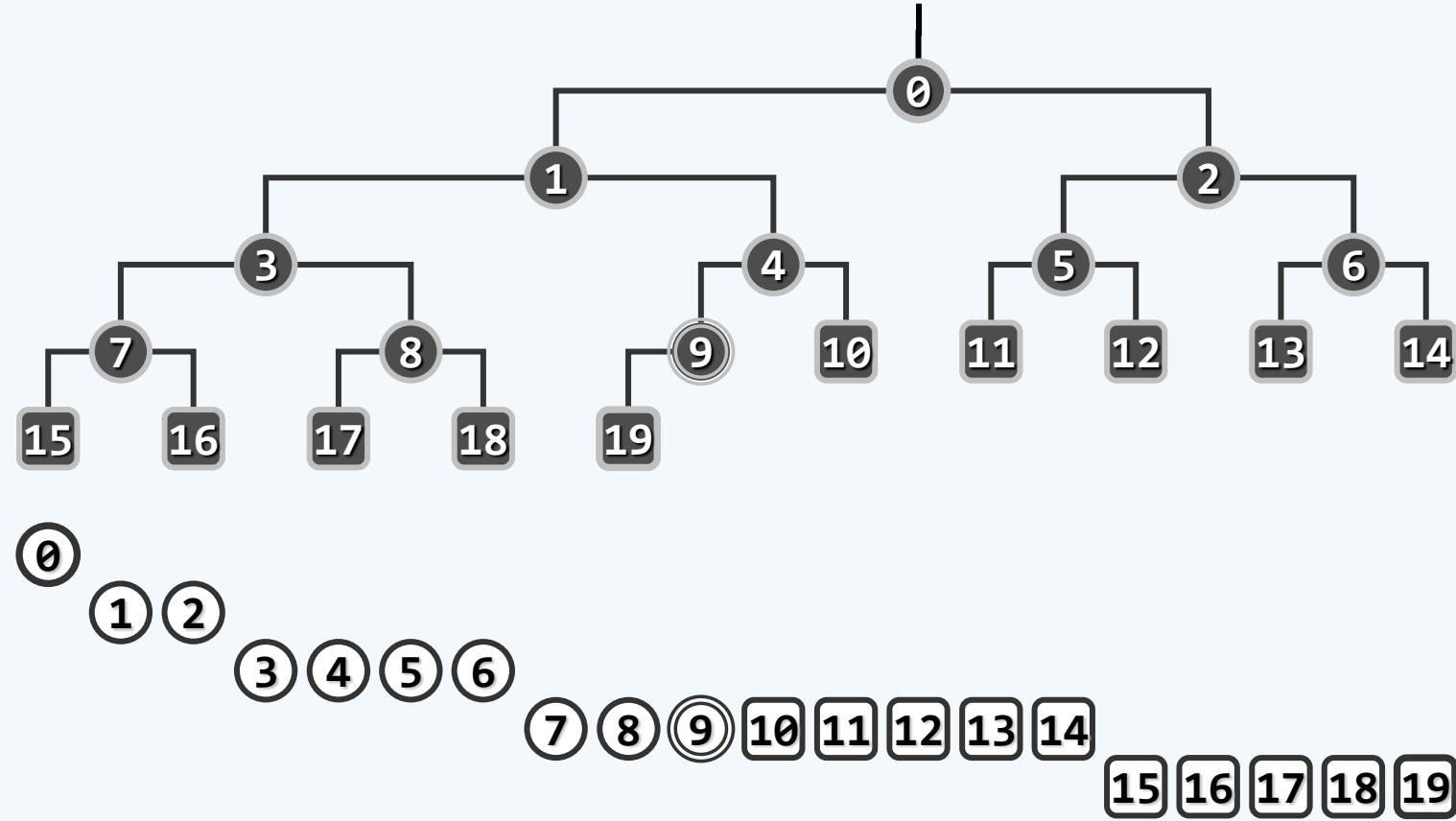
❖ 共n个节点时，内部节点的最大秩 = $\lfloor (n - 2)/2 \rfloor = \lceil (n - 3)/2 \rceil$ //比如，n = 2或3 ...

PQ_CmplHeap = PQ + Vector

```
❖ template <typename T> class PQ_CmplHeap : public PQ<T>, public Vector<T> {  
protected: Rank percolateDown( Rank n, Rank i ); //下滤  
Rank percolateUp( Rank i ); //上滤  
void heapify( Rank n ); //Floyd建堆算法  
public: PQ_CmplHeap( T* A, Rank n ) //批量构造  
{ copyFrom( A, 0, n ); heapify( n ); }  
void insert( T ); //按照比较器确定的优先级次序，插入词条  
T getMax() { return _elem[0]; } //读取优先级最高的词条  
T delMax(); //删除优先级最高的词条  
};
```

堆序性

- ❖ $H[i] \geq \max(H[lc(i)], H[rc(i)])$
- ❖ 堆顶 $H[0]$ 即是全局最大元素——于是 getMax() / delMax() 只需...



插入与上滤：算法

❖ 为插入词条e，只需将e作为末元素接入向量

//结构性自然保持

//若堆序性也亦未破坏，则完成

❖ 否则 //只能是e与其父节点违反堆序性

e与其父节点换位 //若堆序性因此恢复，则完成

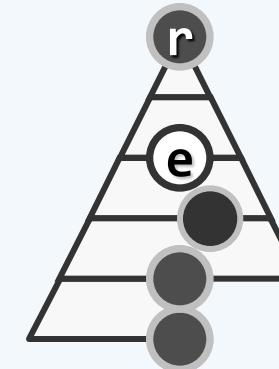
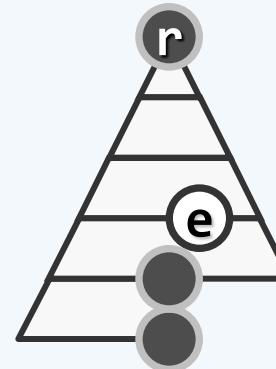
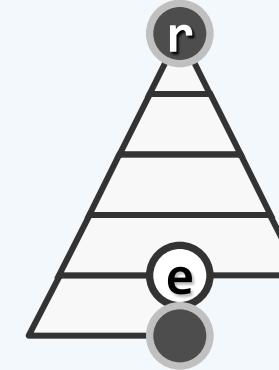
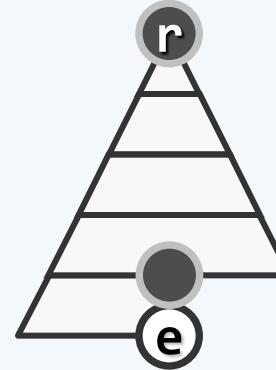
❖ 否则 //依然只可能是e与其（新的）父节点...

e再与父节点换位

❖ 不断重复...直到

e与其父亲满足堆序性，或者

e到达堆顶（没有父亲）

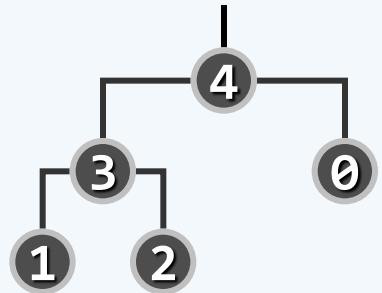


插入与上滤：实现

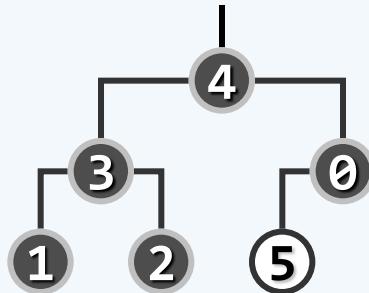
```
❖ template <typename T> void PQ_CmplHeap<T>::insert( T e ) //插入
{ Vector<T>::insert( e ); percolateUp( _size - 1 ); }

❖ template <typename T> //对第i个词条实施上滤，i < _size
Rank PQ_CmplHeap<T>::percolateUp( Rank i ) {
    while ( ParentValid( i ) ) { //只要i有父亲（尚未抵达堆顶），则
        Rank j = Parent( i ); //将i之父记作j
        if ( lt( _elem[i], _elem[j] ) ) break; //一旦父子不再逆序，上滤旋即完成
        swap( _elem[i], _elem[j] ); i = j; //否则，交换父子位置，并上升一层
    } //while
    return i; //返回上滤最终抵达的位置
}
```

插入与上滤：实例

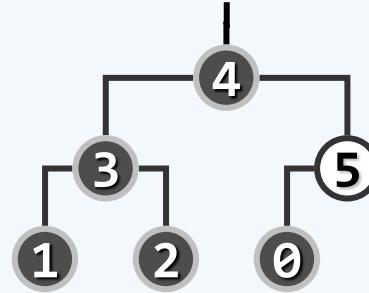


4 3 0 1 2



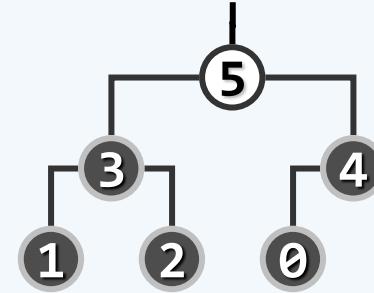
4 3 0 1 2 5

Swap



4 3 5 1 2 0

Swap



5 3 4 1 2 0

插入与上滤：效率

❖ e与父亲的交换，每次只需 $\Theta(1)$ 时间，且

每经过一次交换，e都会上升一层

❖ 在插入新节点e的整个过程中，

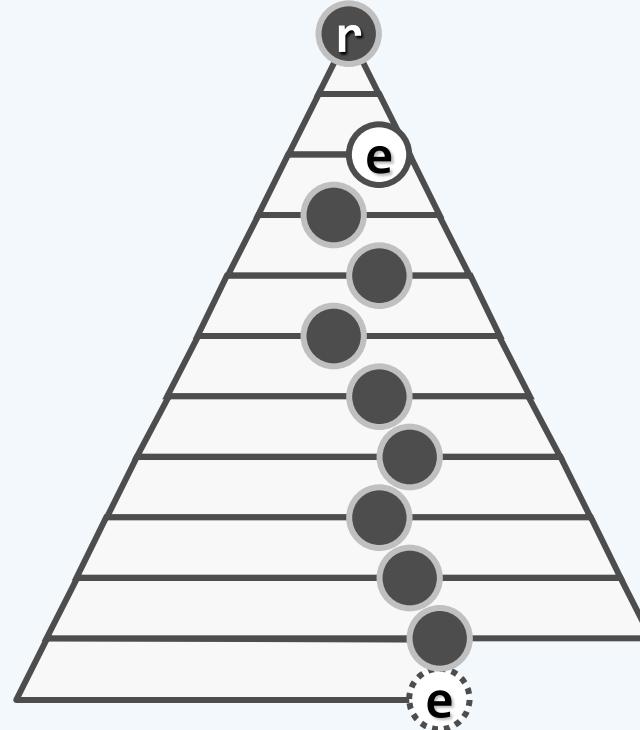
只有e的祖先们，才有可能需要与之交换

❖ 这里的堆以完全树实现，必平衡，故

e的祖先至多 $\Theta(\log n)$ 个

❖ 结论：通过上滤，可在 $\Theta(\log n)$ 时间内

插入一个新节点，并整体地重新调整为堆

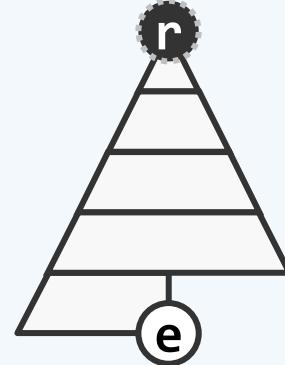


删除与下滤：算法

❖ 最大元素始终在堆顶，故为删除之，只需...

❖ 摘除向量首元素，代之以末元素e

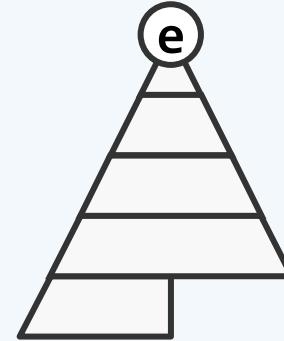
//结构性保持；若堆序性依然保持则完成



❖ 否则 //即e与孩子们违背堆序性

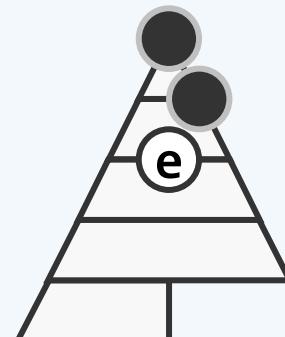
e与孩子中的大者换位

//若堆序性因此恢复，则完成



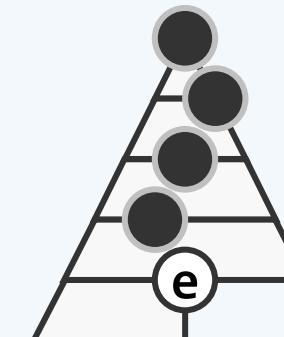
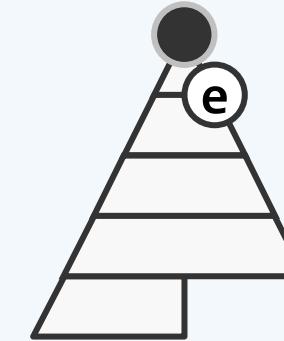
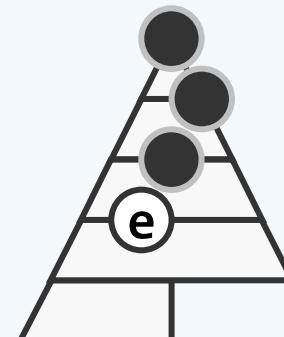
❖ 否则 //即e与其（新的）孩子们...

e再次与孩子中的大者换位



❖ 不断重复...直到

e满足堆序性，或者已是叶子

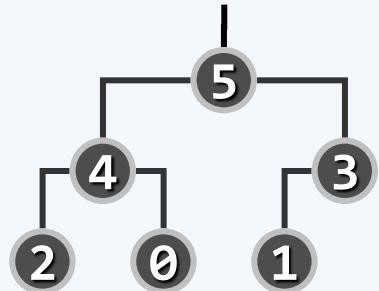


删除与下滤：实现

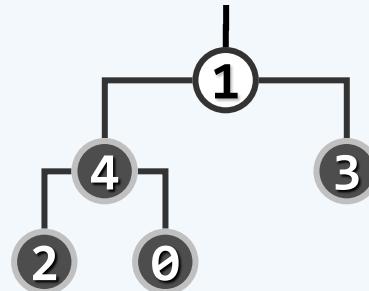
```
❖ template <typename T> T PQ_CmplHeap<T>::delMax() { //删除
    T maxElem = _elem[0]; _elem[0] = _elem[ --_size ]; //摘除堆顶，代之以末词条
    percolateDown( _size, 0 ); //对新堆顶实施下滤
    return maxElem; //返回此前备份的最大词条
}

❖ template <typename T> //对前n个词条中的第i个实施下滤，i < n
Rank PQ_CmplHeap<T>::percolateDown( Rank n, Rank i ) {
    Rank j; //i及其（至多两个）孩子中，堪为父者
    while ( i != ( j = ProperParent( _elem, n, i ) ) ) //只要i非j，则
        { swap( _elem[i], _elem[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）
}
```

删除与下滤：实例

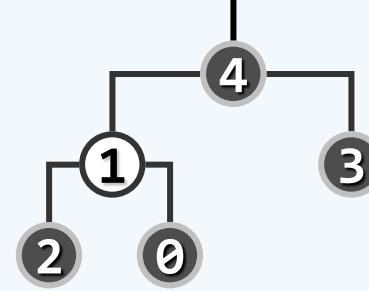


5 4 3 2 0 1



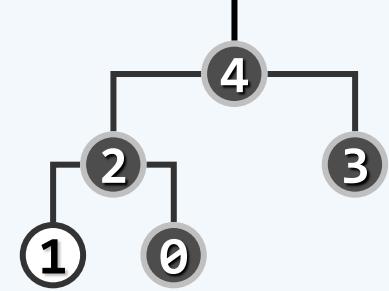
1 4 3 2 0

swap



4 1 3 2 0

swap



4 2 3 1 0

删除与下滤：效率

❖ 在下滤的过程中

每经过一次交换

e的高度都降低一层

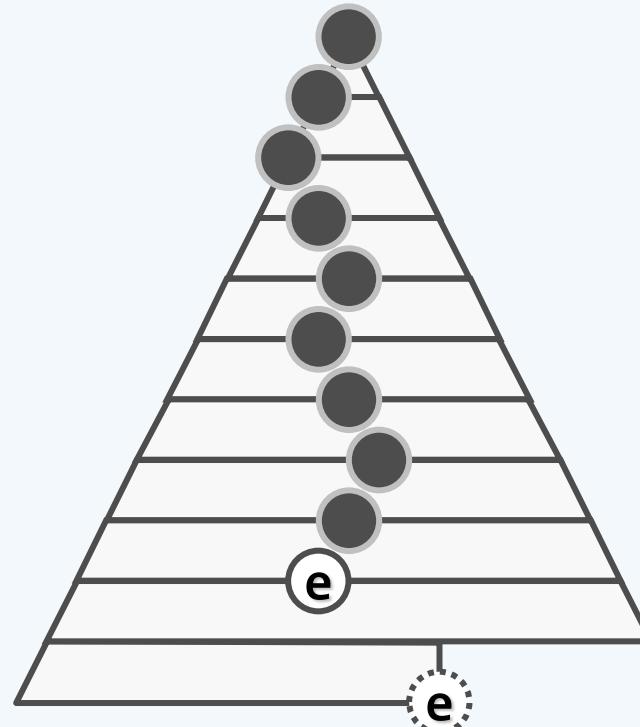
❖ 故此，在每层至多需要一次交换

❖ 再次地，由于堆是完全树，故其高度为 $\Theta(\log n)$

❖ 结论：通过下滤，可在 $\Theta(\log n)$ 时间内

删除堆顶节点，并

整体重新调整为堆



批量构造：自上而下的上滤：蛮力

❖ PQ_CmplHeap(T* A, Rank n) //批量构造
{ copyFrom(A, 0, n); heapify(n); } //如何实现？

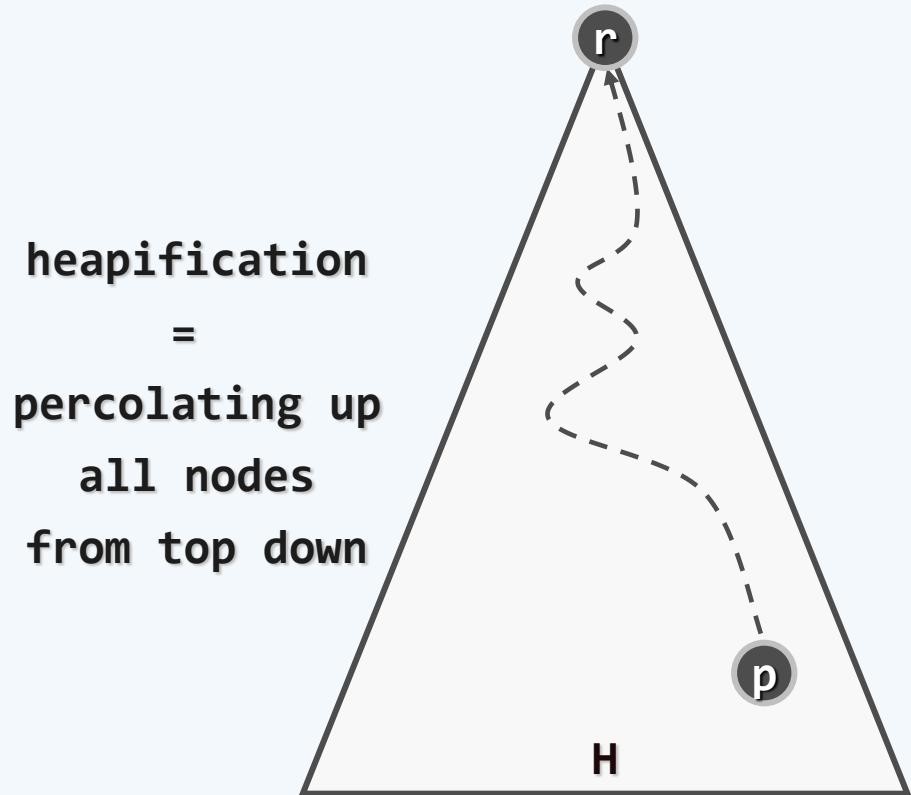
❖ 蛮力：从空堆开始，**依次插入各节点**

❖ 最坏情况，**每个新节点都需上滤至根**

累计耗时 $\Theta(n \log n)$

❖ 这样长的时间，本足以**全排序**！

❖ 应该，能够更快的...



批量构造：自下而上的下滤：算法及实现

❖ 任意给定堆 H_0 和 H_1 ，以及节点 p

❖ 为得到堆 $H_0 \cup \{p\} \cup H_1$ ，只需

将 r_0 和 r_1 当作 p 的孩子，对 p 下滤

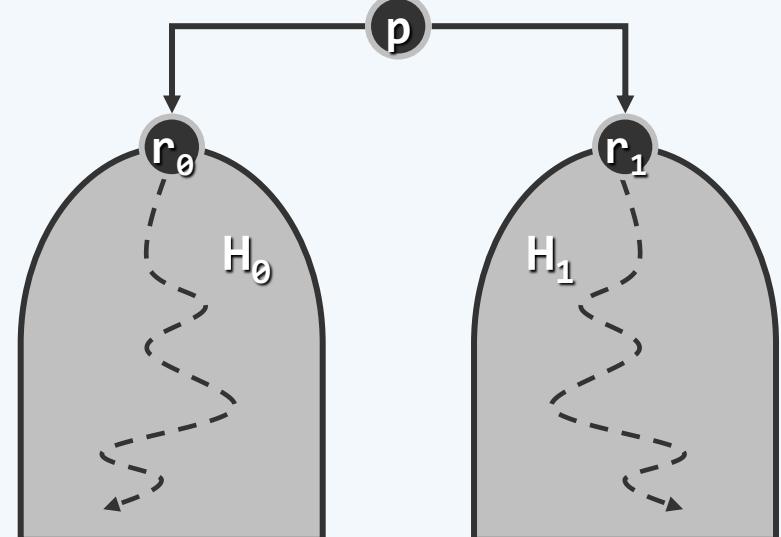
❖ template <typename T>

```
void PQ_CmplHeap<T>::heapify( Rank n ) { //Robert Floyd, 1964
```

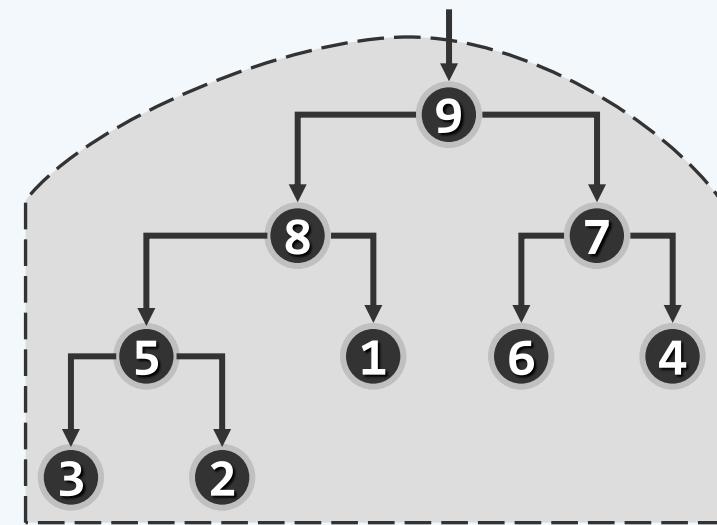
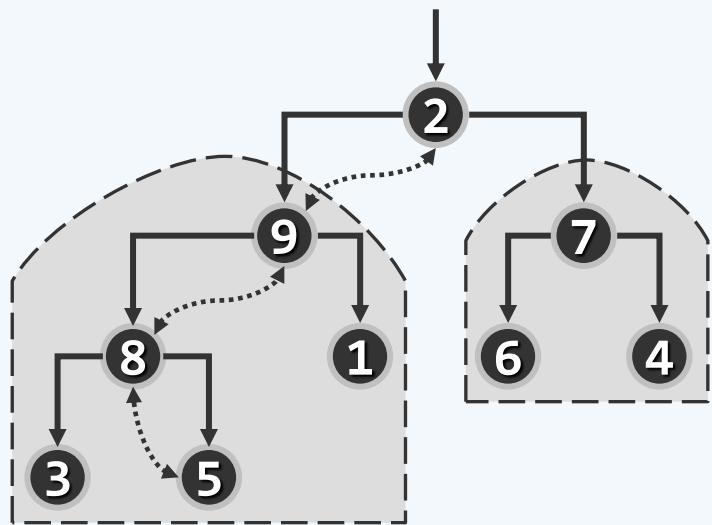
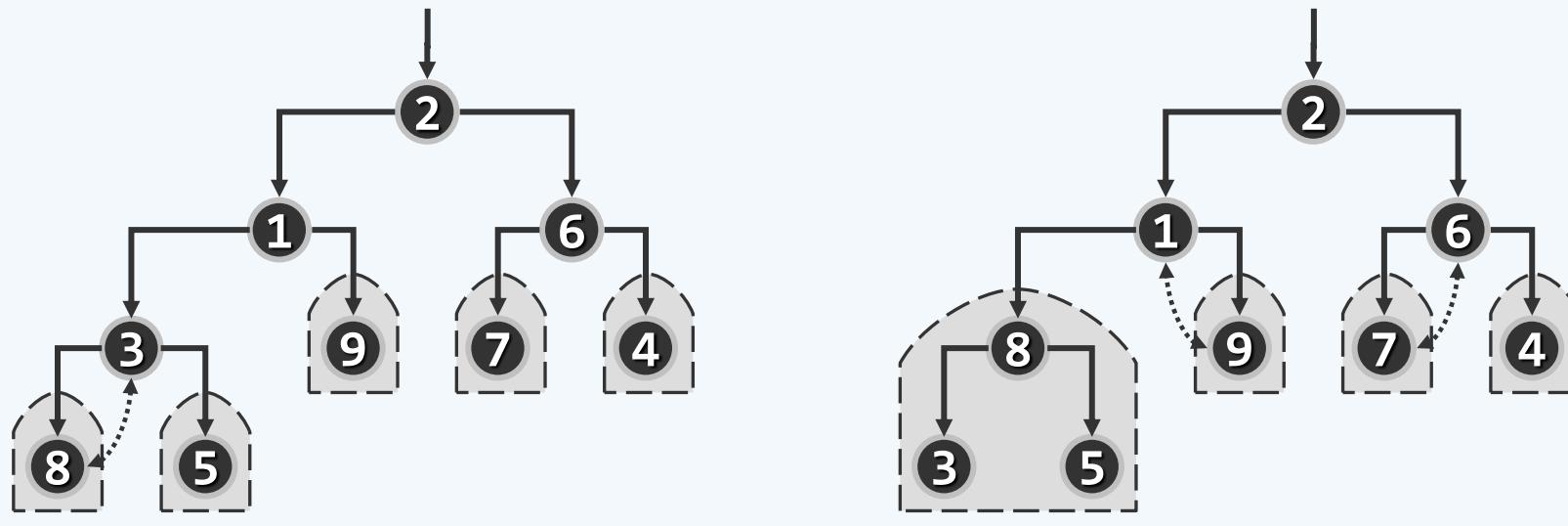
```
for ( int i = LastInternal(n); i >= 0; i-- ) //自下而上，依次
```

```
percolateDown( n, i ); //下滤各内部节点
```

```
} //可理解为子堆的逐层合并，——由以上性质，堆序性最终必然在全局恢复
```



批量构造：自下而上的下滤：实例



批量构造：自下而上的下滤：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： $n = 2^{d+1} - 1$

❖ $S(n) = \text{所有节点的} \boxed{\text{高度}} \text{总和}$

$$= \sum_{i=0..d} ((d - i) \times 2^i)$$

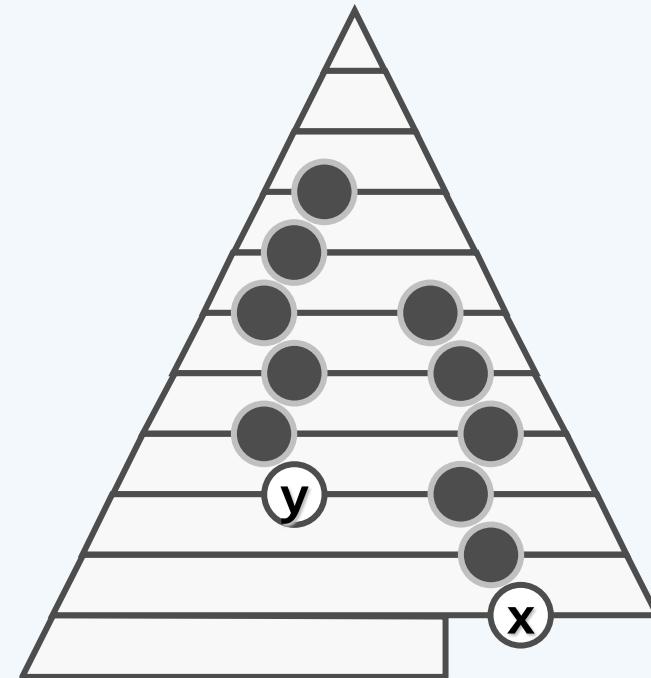
$$= d \times \sum_{i=0..d} 2^i - T(n)$$

$$= d \times (2^{d+1} - 1) - [(d - 1) \times 2^{d+1} + 2]$$

$$= 2^{d+1} - (d + 2)$$

$$= n - \log_2(n + 1)$$

$$= O(n)$$



- ❖ `insert()` : 最坏情况下效率为 $\Theta(n \log n)$, 平均情况呢 ?
- ❖ `heapify()` : 构造次序颠倒后 , 为什么复杂度会实质性的降低 ?
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 $\Theta(n \log n)$ 时间内构造 Huffman 树
- ❖ 在大顶堆中 , `delMin()` 操作能否也在 $\Theta(n \log n)$ 时间内完成 ?
难道 , 为此需要同时维护一个 小顶堆 ?