

二叉堆

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>

码拉松



实力IT教育 www.520it.com

■ 设计一种数据结构，用来存放整数，要求提供 3 个接口

□ 添加元素

□ 获取最大值

□ 删除最大值

0	1	2	3	4	5	6
31	66	17	15	28	20	59

0	1	2	3	4	5	6
15	17	20	28	31	59	66

	获取最大值	删除最大值	添加元素	
动态数组\双向链表	$O(n)$	$O(n)$	$O(1)$	
有序动态数组\双向链表	$O(1)$	$O(1)$	$O(n)$	全排序有点浪费
BBST	$O(\log n)$	$O(\log n)$	$O(\log n)$	杀鸡用了牛刀

■ 有没有更优的数据结构？

□ 堆

✓ 获取最大值： $O(1)$ 、删除最大值： $O(\log n)$ 、添加元素： $O(\log n)$

Top K问题

- 什么是 Top K 问题
 - 从海量数据中找出前 K 个数据
- 比如
 - 从 100 万个整数中找出最大的 100 个整数
- Top K 问题的解法之一：可以用数据结构“堆”来解决

堆 (Heap)

■ 堆 (Heap) 也是一种树状的数据结构 (不要跟内存模型中的“堆空间”混淆), 常见的堆实现有

□ 二叉堆 (Binary Heap, 完全二叉堆)

□ 多叉堆 (D-heap、D-ary Heap)

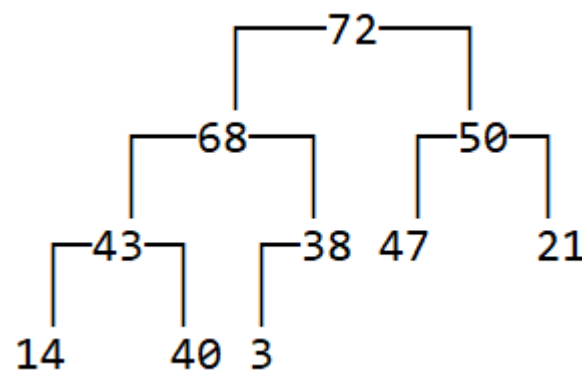
□ 索引堆 (Index Heap)

□ 二项堆 (Binomial Heap)

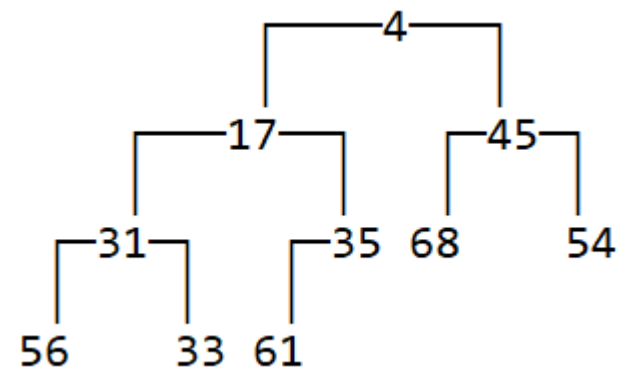
□ 斐波那契堆 (Fibonacci Heap)

□ 左倾堆 (Leftist Heap, 左式堆)

□ 斜堆 (Skew Heap)



最大堆_二叉堆



最小堆_二叉堆

■ 堆的一个重要性质: 任意节点的值总是 \geq (\leq) 子节点的值

□ 如果任意节点的值总是 \geq 子节点的值, 称为: 最大堆、大根堆、大顶堆

□ 如果任意节点的值总是 \leq 子节点的值, 称为: 最小堆、小根堆、小顶堆

■ 由此可见, 堆中的元素必须具备可比较性 (跟二叉搜索树一样)

堆的基本接口设计

- `int size();` // 元素的数量
- `boolean isEmpty();` // 是否为空
- `void clear();` // 清空
- `void add(E element);` // 添加元素
- `E get();` // 获得堆顶元素
- `E remove();` // 删除堆顶元素
- `E replace(E element);` // 删除堆顶元素的同时插入一个新元素

二叉堆 (Binary Heap)

■ 二叉堆的逻辑结构就是一棵完全二叉树，所以也叫完全二叉堆

■ 鉴于完全二叉树的一些特性，二叉堆的底层（物理结构）一般用数组实现即可

■ 索引 i 的规律（ n 是元素数量）

□ 如果 $i = 0$ ，它是根节点

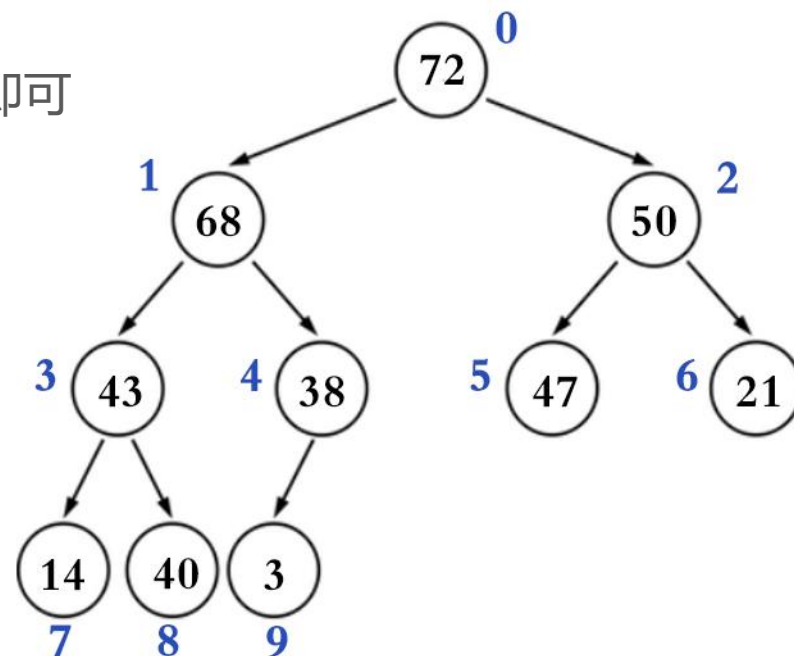
□ 如果 $i > 0$ ，它的父节点的索引为 $\text{floor}((i - 1) / 2)$

□ 如果 $2i + 1 \leq n - 1$ ，它的左子节点的索引为 $2i + 1$

□ 如果 $2i + 1 > n - 1$ ，它无左子节点

□ 如果 $2i + 2 \leq n - 1$ ，它的右子节点的索引为 $2i + 2$

□ 如果 $2i + 2 > n - 1$ ，它无右子节点



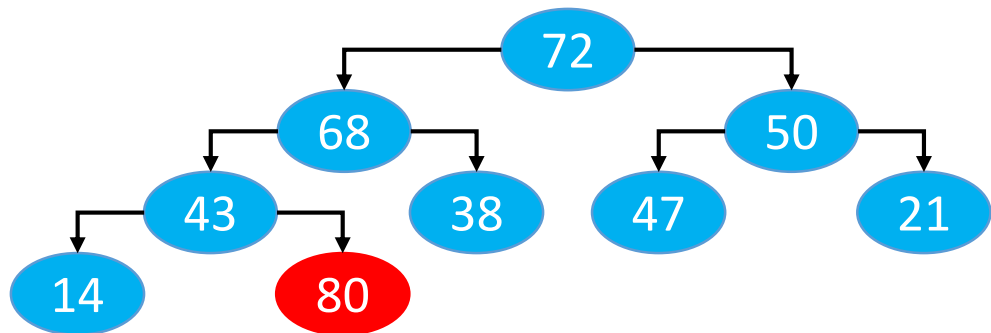
0	1	2	3	4	5	6	7	8	9
72	68	50	43	38	47	21	14	40	3

获取最大值

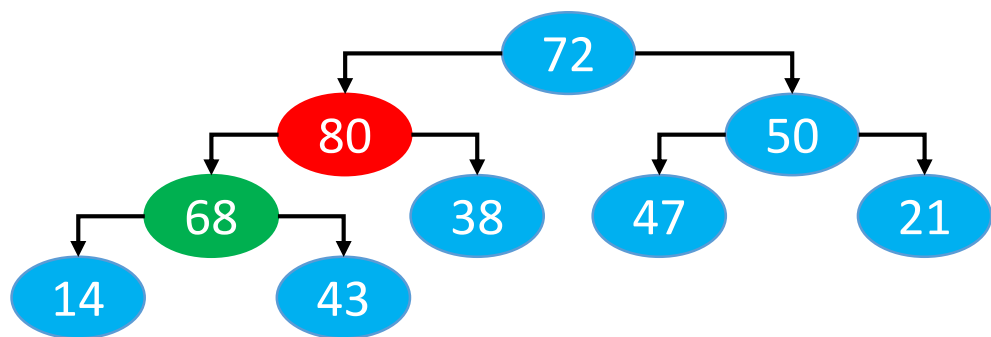
```
public E get() {  
    emptyCheck();  
    return elements[0];  
}
```

```
private void emptyCheck() {  
    if (size == 0) {  
        throw new IndexOutOfBoundsException("Heap is empty");  
    }  
}
```

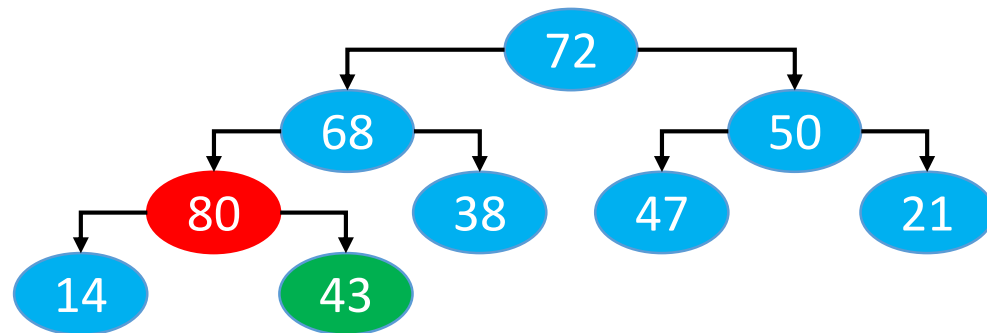
最大堆 - 添加



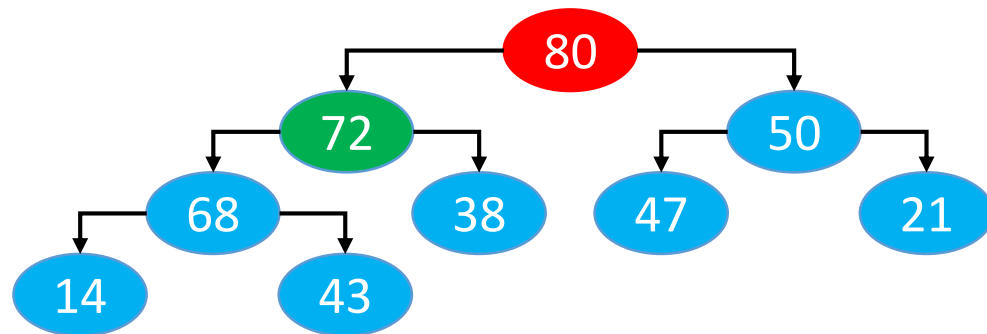
0	1	2	3	4	5	6	7	8	9
72	68	50	43	38	47	21	14	80	



0	1	2	3	4	5	6	7	8	9
72	80	50	68	38	47	21	14	43	

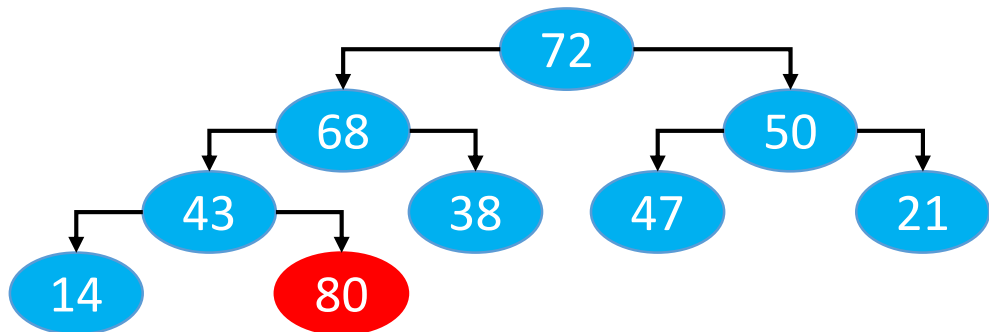


0	1	2	3	4	5	6	7	8	9
72	68	50	80	38	47	21	14	43	



0	1	2	3	4	5	6	7	8	9
80	72	50	68	38	47	21	14	43	

最大堆 - 添加 - 总结



■ 循环执行以下操作 (图中的 80 简称为 node)

□ 如果 $node > 父节点$

✓ 与父节点交换位置

□ 如果 $node \leq 父节点$, 或者 node 没有父节点

✓ 退出循环

■ 这个过程, 叫做上滤 (Sift Up)

□ 时间复杂度: $O(\log n)$

```

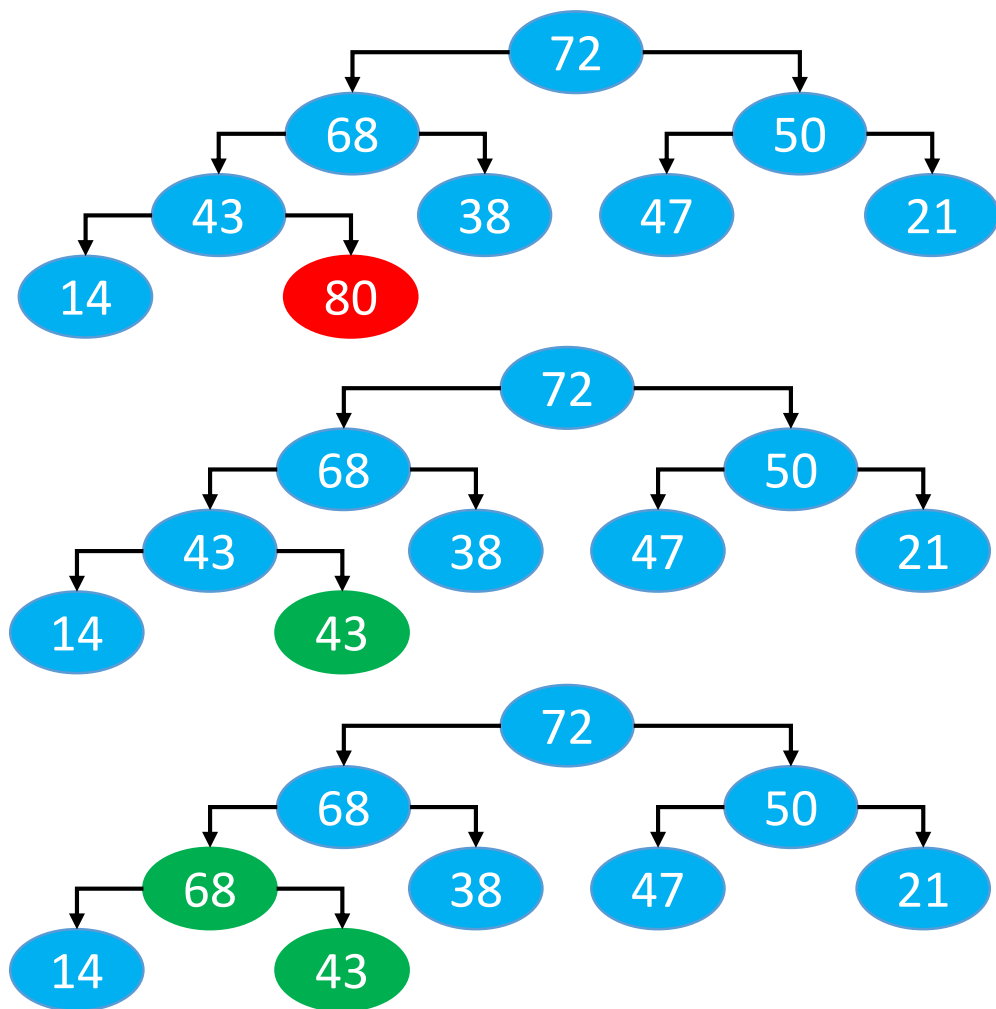
public void add(E element) {
    elementNotNullCheck(element);
    ensureCapacity(size + 1);
    elements[size++] = element;
    siftUp(size - 1);
}
    
```

```

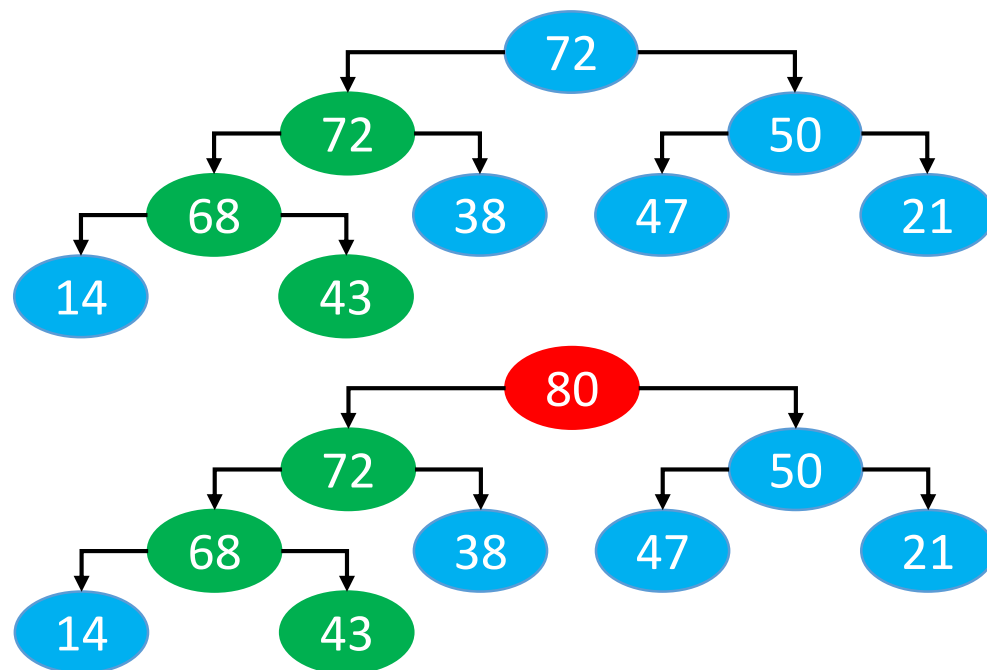
private void siftUp(int index) {
    E element = elements[index];
    while (index > 0) {
        int parentIndex = (index - 1) >> 1;
        E parent = elements[parentIndex];
        // 小于父节点
        if (compare(parent, element) >= 0) break;
        // 将父元素安排到index位置
        elements[index] = parent;
        index = parentIndex;
    }
    elements[index] = element;
}
    
```

最大堆 – 添加 – 交换位置的优化

- 一般交换位置需要3行代码，可以进一步优化
- 将新添加节点备份，确定最终位置才摆放上去

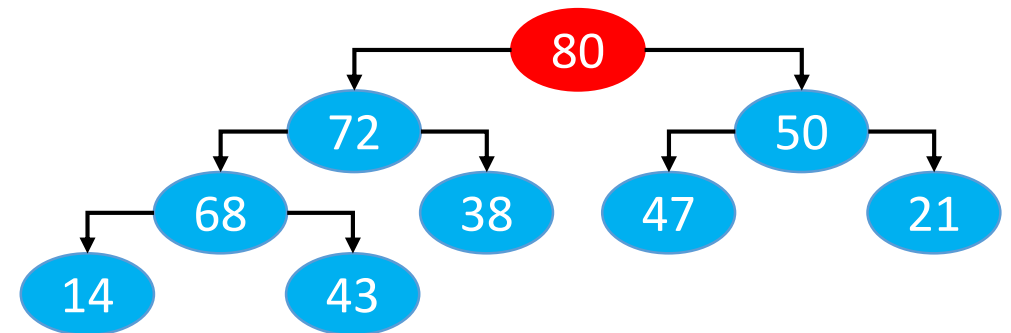


80

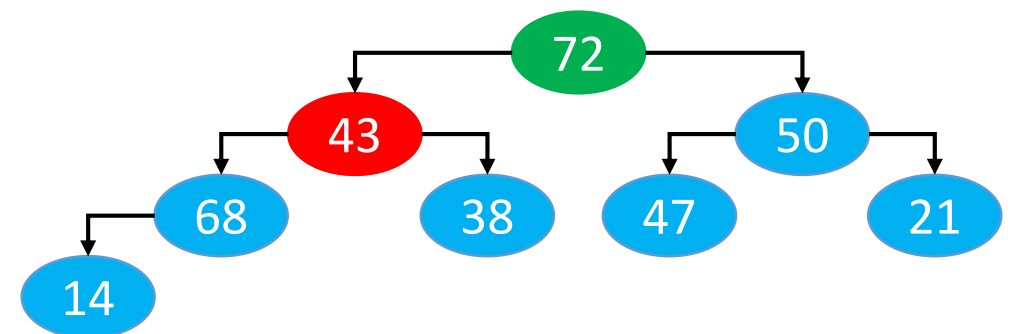


- 仅从交换位置的代码角度看
- 可以由大概的 $3 * O(\log n)$ 优化到 $1 * O(\log n) + 1$

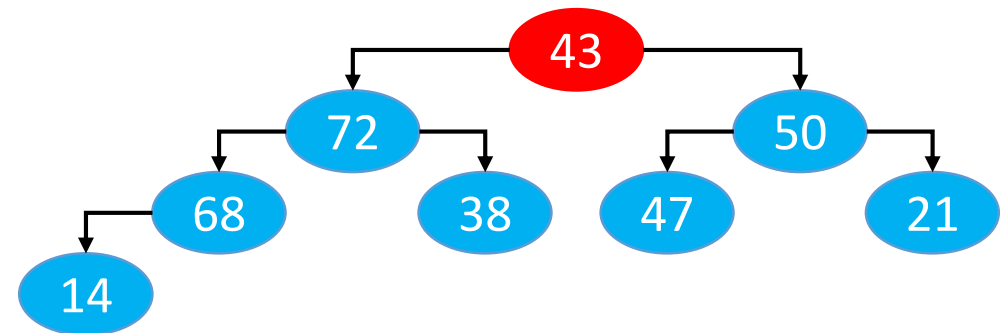
最大堆 - 删除



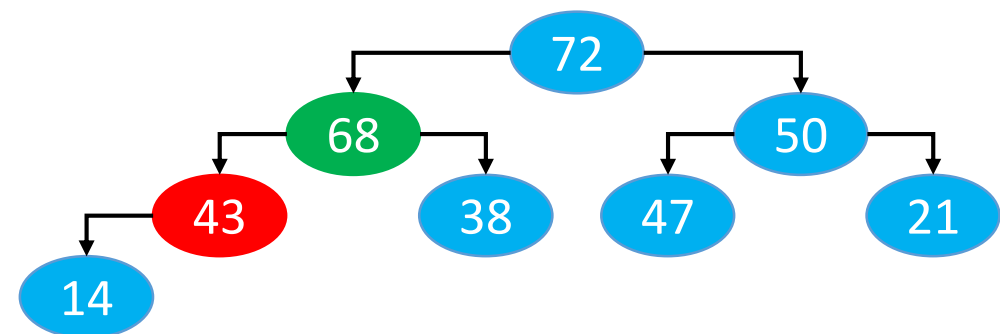
0	1	2	3	4	5	6	7	8	9
80	72	50	68	38	47	21	14	43	



0	1	2	3	4	5	6	7	8	9
72	43	50	68	38	47	21	14		

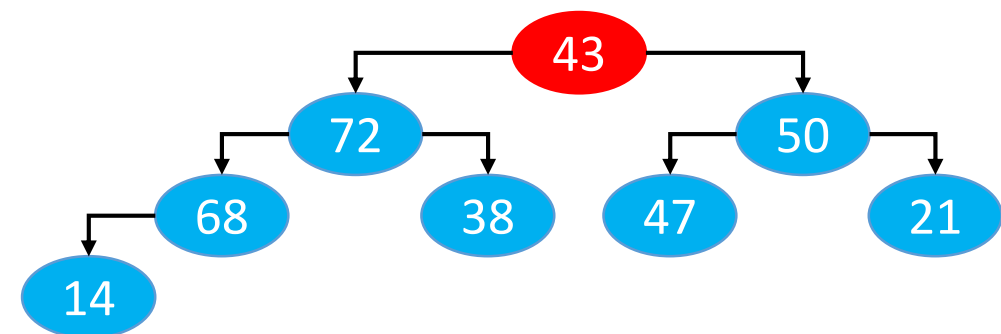
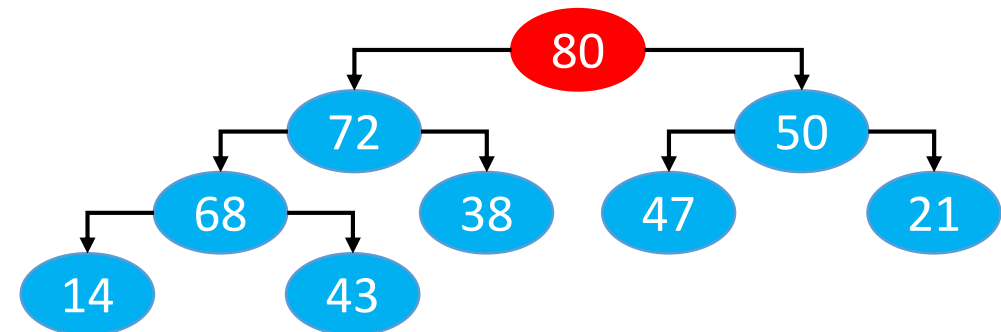


0	1	2	3	4	5	6	7	8	9
43	72	50	68	38	47	21	14		



0	1	2	3	4	5	6	7	8	9
72	68	50	43	38	47	21	14		

最大堆 – 删除 – 总结



1. 用最后一个节点覆盖根节点
 2. 删除最后一个节点
 3. 循环执行以下操作 (图中的 43 简称为 node)
 - ❑ 如果 $\text{node} < \text{最大的子节点}$
 - ✓ 与最大的子节点交换位置
 - ❑ 如果 $\text{node} \geq \text{最大的子节点}$, 或者 node 没有子节点
 - ✓ 退出循环
- 这个过程, 叫做下滤 (Sift Down), 时间复杂度: $O(\log n)$
- 同样的, 交换位置的操作可以像添加那样进行优化

最大堆 - 删除

```
public E remove() {  
    emptyCheck();  
    E first = elements[0];  
    int lastIndex = --size;  
    elements[0] = elements[lastIndex];  
    elements[lastIndex] = null;  
    siftDown(0);  
    return first;  
}
```

```
private void siftDown(int index) {  
    E element = elements[index];  
  
    int half = size >> 1;  
    while (index < half) { // index必须是非叶子节点  
        // 默认是左边跟父节点比  
        int childIndex = (index << 1) + 1;  
        E child = elements[childIndex];  
  
        int rightIndex = childIndex + 1;  
        // 右子节点比左子节点大  
        if (rightIndex < size &&  
            compare(elements[rightIndex], child) > 0) {  
            child = elements[childIndex = rightIndex];  
        }  
  
        // 大于等于子节点  
        if (compare(element, child) >= 0) break;  
  
        elements[index] = child;  
        index = childIndex;  
    }  
    elements[index] = element;  
}
```

replace

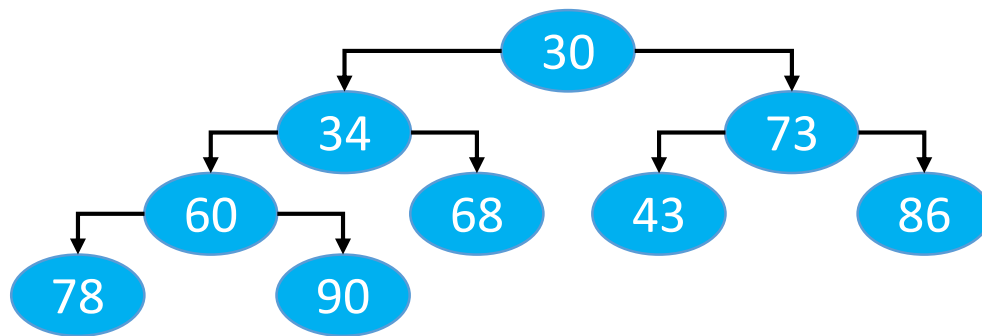
```
public E replace(E element) {  
    E top = null;  
    if (size == 0) {  
        elements[size++] = element;  
    } else {  
        top = elements[0];  
        elements[0] = element;  
        siftDown(0);  
    }  
    return top;  
}
```

最大堆 – 批量建堆 (Heapify)

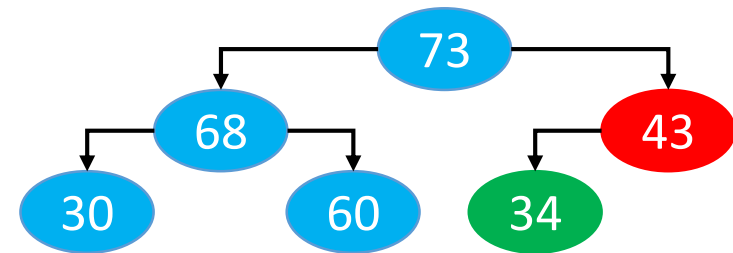
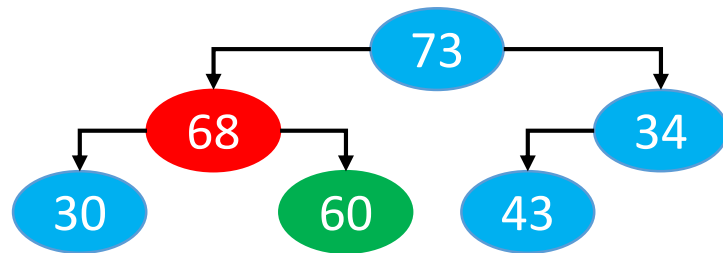
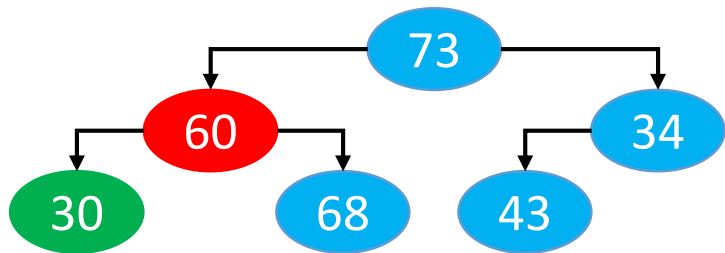
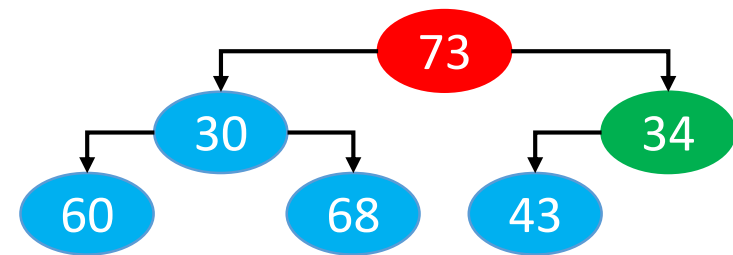
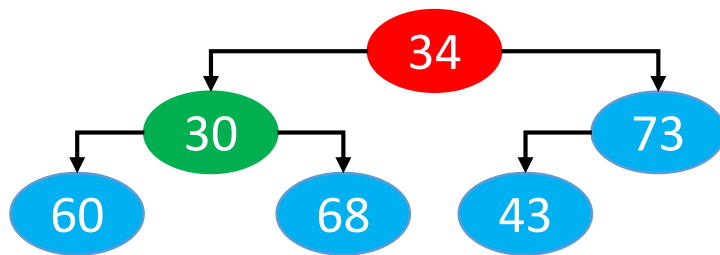
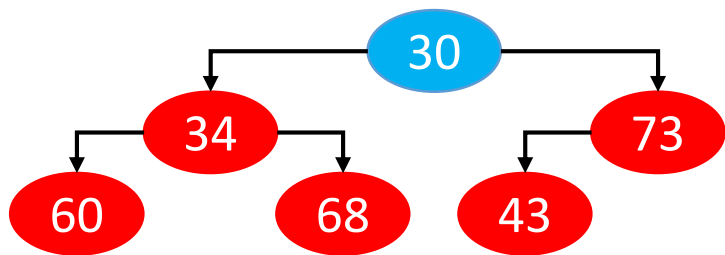
■ 批量建堆，有 2 种做法

□ 自上而下的上滤

□ 自下而上的下滤

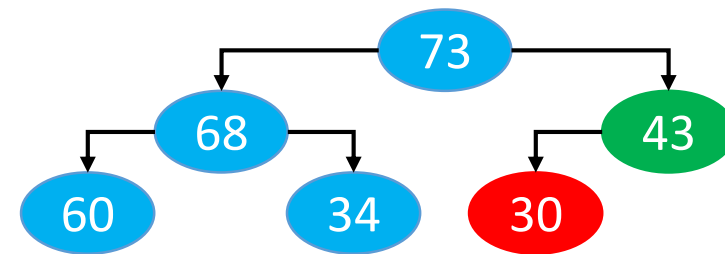
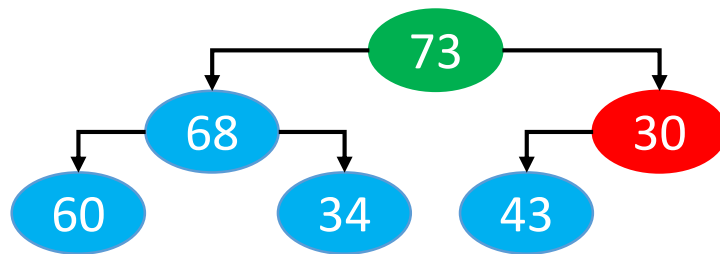
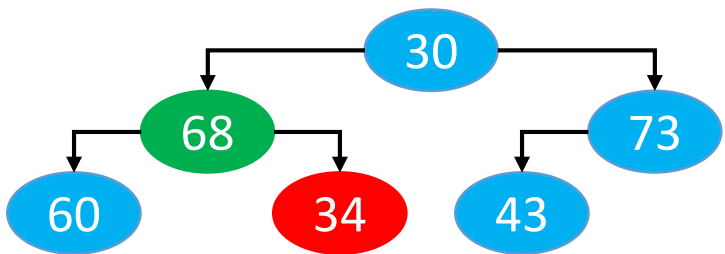
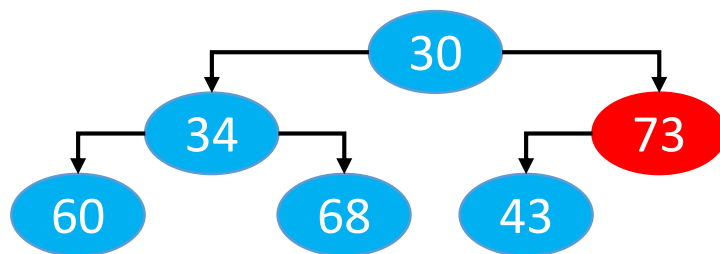
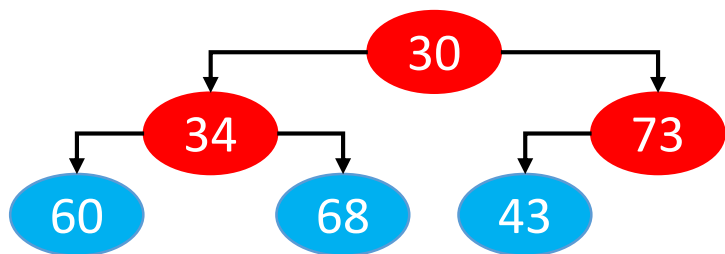


最大堆 – 批量建堆 – 自上而下的上滤



```
for (int i = 1; i < size; i++) {  
    siftUp(i);  
}
```

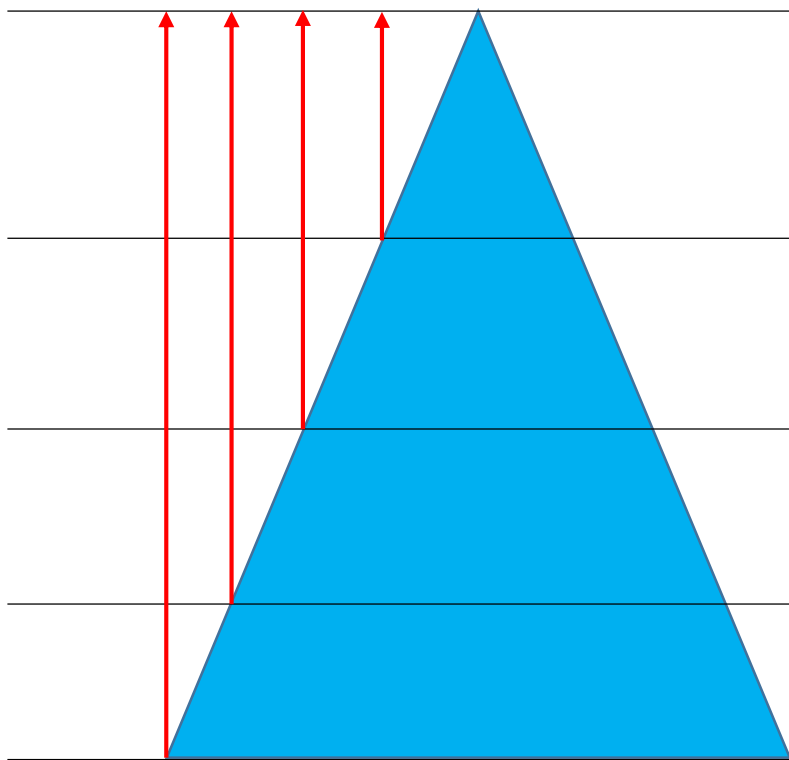

最大堆 – 批量建堆 – 自下而上的下滤



```
for (int i = (size >> 1) - 1; i >= 0; i--) {  
    siftDown(i);  
}
```

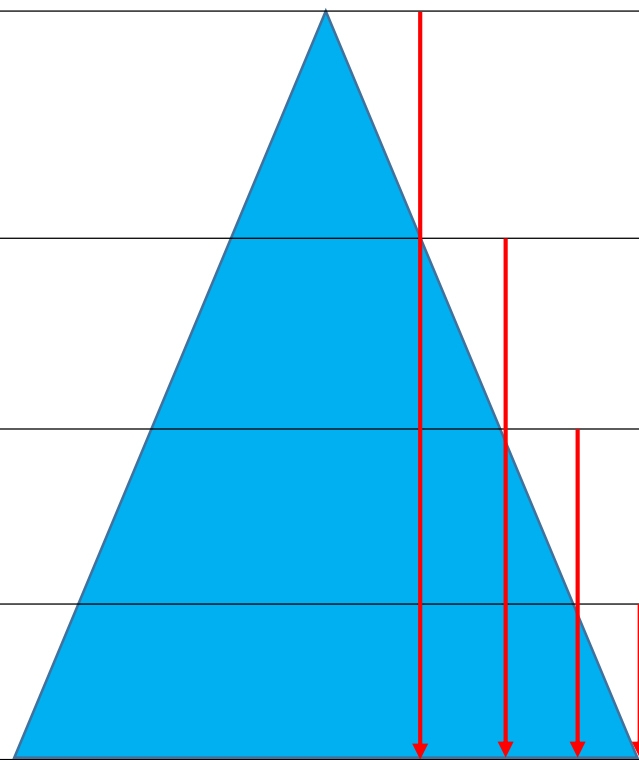
最大堆 – 批量建堆 – 效率对比

自上而下的上滤



所有节点的深度之和
 $O(n \log n)$

自下而上的下滤



所有节点的高度之和
 $O(n)$

最大堆 – 批量建堆 – 效率对比

■ 所有节点的深度之和

- 仅仅是叶子节点，就有近 $n/2$ 个，而且每一个叶子节点的深度都是 $O(\log n)$ 级别的
- 因此，在叶子节点这一块，就达到了 $O(n \log n)$ 级别
- $O(n \log n)$ 的时间复杂度足以利用排序算法对所有节点进行全排序

■ 所有节点的高度之和

- 假设是满树，节点总个数为 n ，树高为 h ，那么 $n = 2^h - 1$
- 所有节点的树高之和 $H(n) = 2^0 * (h - 0) + 2^1 * (h - 1) + 2^2 * (h - 2) + \dots + 2^{h-1} * [h - (h - 1)]$
- $H(n) = h * (2^0 + 2^1 + 2^2 + \dots + 2^{h-1}) - [1 * 2^1 + 2 * 2^2 + 3 * 2^3 + \dots + (h - 1) * 2^{h-1}]$
- $H(n) = h * (2^h - 1) - [(h - 2) * 2^h + 2]$
- $H(n) = h * 2^h - h - h * 2^h + 2^{h+1} - 2$
- $H(n) = 2^{h+1} - h - 2 = 2 * (2^h - 1) - h = 2n - h = 2n - \log_2(n + 1) = O(n)$

$$\blacksquare S(h) = 1 * 2^1 + 2 * 2^2 + 3 * 2^3 + \dots + (h-2) * 2^{h-2} + (h-1) * 2^{h-1}$$

$$\blacksquare 2S(h) = 1 * 2^2 + 2 * 2^3 + 3 * 2^4 + \dots + (h-2) * 2^{h-1} + (h-1) * 2^h$$

$$\blacksquare S(h) - 2S(h) = [2^1 + 2^2 + 2^3 + \dots + 2^{h-1}] - (h-1) * 2^h = (2^h - 2) - (h-1) * 2^h$$

$$\blacksquare S(h) = (h-1) * 2^h - (2^h - 2) = (h-2) * 2^h + 2$$

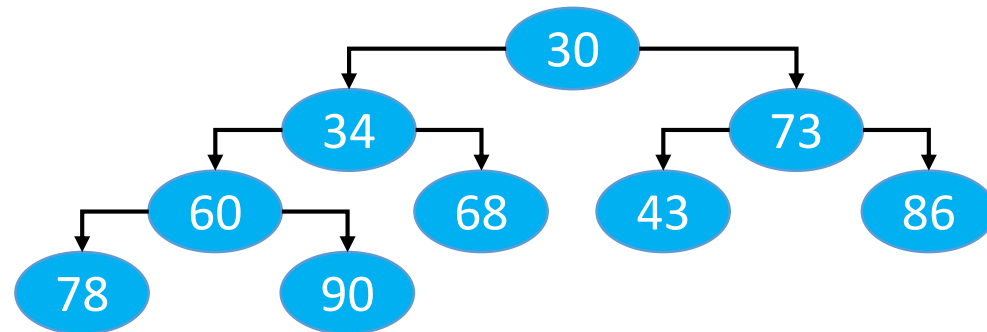
■ 以下方法可以批量建堆么

□ 自上而下的下滤

□ 自下而上的上滤

■ 上述方法不可行，为什么？

□ 认真思考【自上而下的上滤】、【自下而上的下滤】的本质



```
public BinaryHeap(E[] elements, Comparator<E> comparator) {
    super(comparator);

    if (elements == null || elements.length == 0) {
        this.elements = (E[]) new Object[DEFAULT_CAPACITY];
    } else {
        int capacity = Math.max(DEFAULT_CAPACITY, elements.length);
        this.elements = (E[]) new Object[capacity];
        this.size = elements.length;
        for (int i = 0; i < elements.length; i++) {
            this.elements[i] = elements[i];
        }
        heapify();
    }
}
```

```
private void heapify() {
    for (int i = (size >> 1) - 1; i >= 0; i--) {
        siftDown(i);
    }
}
```

如何构建一个小顶堆?

```
Integer[] data = {  
    70, 30, 34, 73, 60, 68, 43,  
    86, 15, 1, 64, 65, 54, 77,  
    25, 72, 78, 90, 57};  
BinaryHeap<Integer> heap = new BinaryHeap<>(data, new Comparator<Integer>() {  
    public int compare(Integer o1, Integer o2) {  
        return o2 - o1;  
    }  
});
```

Top K问题

- 从 n 个整数中，找出最大的前 k 个数（ k 远远小于 n ）
- 如果使用排序算法进行全排序，需要 $O(n\log n)$ 的时间复杂度
- 如果使用二叉堆来解决，可以使用 $O(n\log k)$ 的时间复杂度来解决
 - 新建一个小顶堆
 - 扫描 n 个整数
 - ✓ 先将遍历到的前 k 个数放入堆中
 - ✓ 从第 $k + 1$ 个数开始，如果大于堆顶元素，就使用 `replace` 操作（删除堆顶元素，将第 $k + 1$ 个数添加到堆中）
 - 扫描完毕后，堆中剩下的就是最大的前 k 个数
- 如果是找出最小的前 k 个数呢？
 - 用大顶堆
 - 如果小于堆顶元素，就使用 `replace` 操作

```
for (int i = 0; i < n; i++) {  
    int value = data[i];  
    if (minHeap.size() < k) {  
        minHeap.add(value);  
    } else if (value > minHeap.get()) {  
        minHeap.replace(value);  
    }  
}
```

```
for (int i = 0; i < n; i++) {  
    int value = data[i];  
    if (maxHeap.size() < k) {  
        maxHeap.add(value);  
    } else if (value < maxHeap.get()) {  
        maxHeap.replace(value);  
    }  
}
```


- 了解和实现堆排序
- 使用堆排序将一个无序数组转换成一个升序数组
- 空间复杂度能否下降至 $O(1)$?