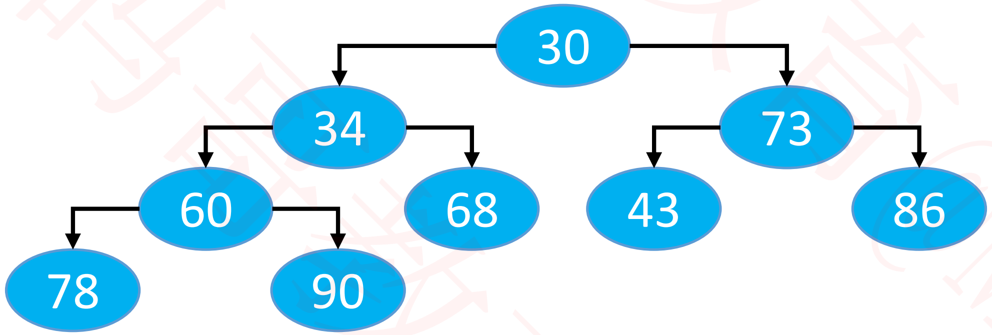
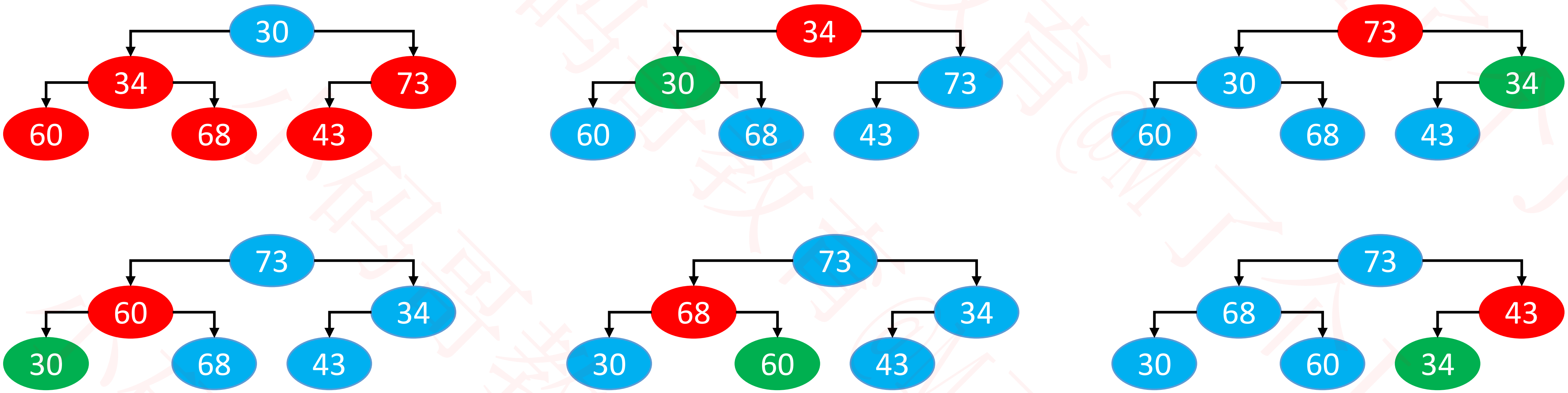
最大堆 – 批量建堆（Heapify）

- 批量建堆，有 2 种做法
  - 自上而下的上滤
  - 自下而上的下滤

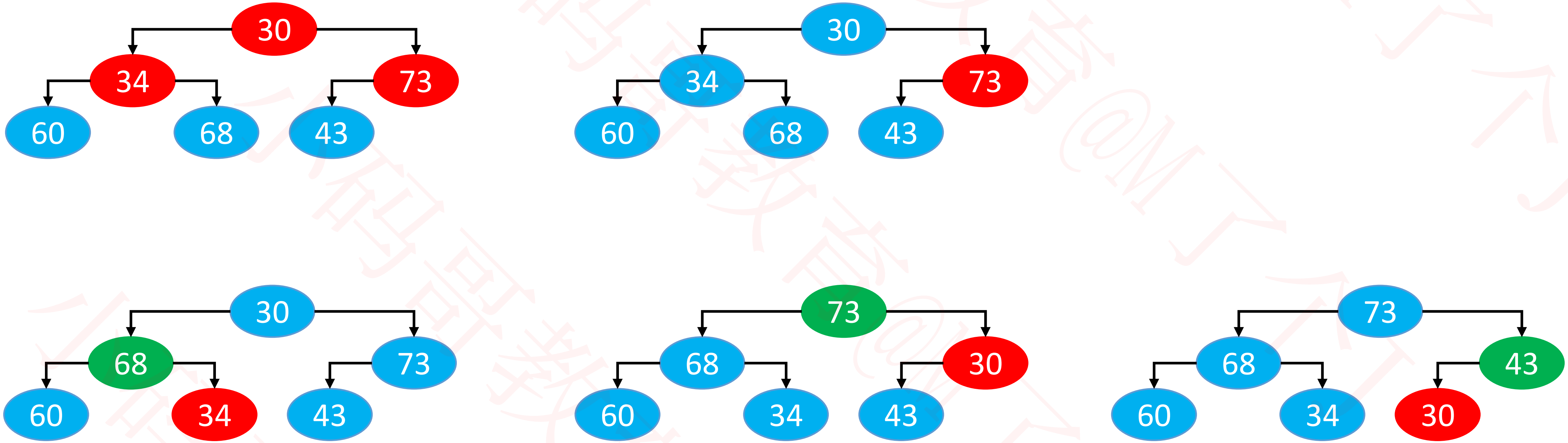


最大堆 – 批量建堆 – 自上而下的上滤




```
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

自下而上的下滤

所有节点的深度之和
O(nlogn)

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(logn) 级别的
  - 因此，在叶子节点这一块，就达到了 O(nlogn) 级别
  - O(nlogn) 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 $H(n) = 2^0 * (h-0) + 2^1 * (h-1) + 2^2 * (h-2) + \cdots + 2^{h-1} * [h-(h-1)]$
  - $H(n) = h * (2^0 + 2^1 + 2^2 + \cdots + 2^{h-1}) - [1 * 2^1 + 2 * 2^2 + 3 * 2^3 + \cdots + (h-1) * 2^{h-1}]$
  - $H(n) = h * (2^h - 1) - [(h-2) * 2^h + 2]$
  - $H(n) = h * 2^h - h - h * 2^h + 2^{h+1} - 2$
  - $H(n) = 2^{h+1} - h - 2 = 2 * (2^h - 1) - h = 2n - h = 2n - \log_2(n+1) = O(n)$

公式推导

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

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

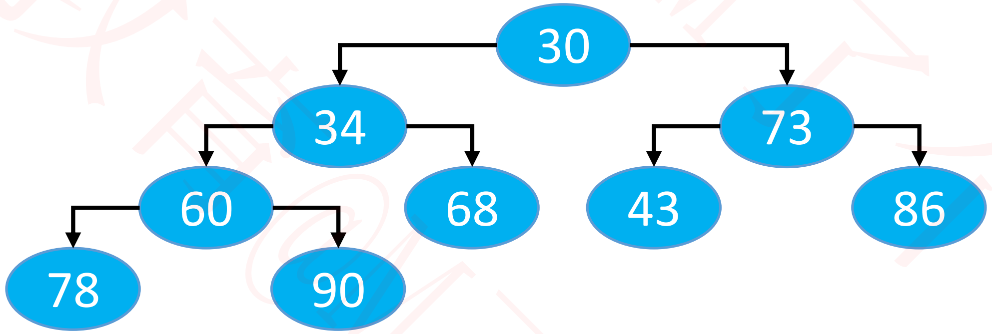
- $S(h) - 2S(h) = [2^1 + 2^2 + 2^3 + \cdots + 2^{h-1}] - (h-1) * 2^h = (2^h - 2) - (h-1) * 2^h$

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

疑惑

- 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



批量建堆

```java
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();
    }
}
```

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