算法与数据结构

liuyubobobo

堆排序 Heap Sort

堆和优先队列 Heap and Priority Queue

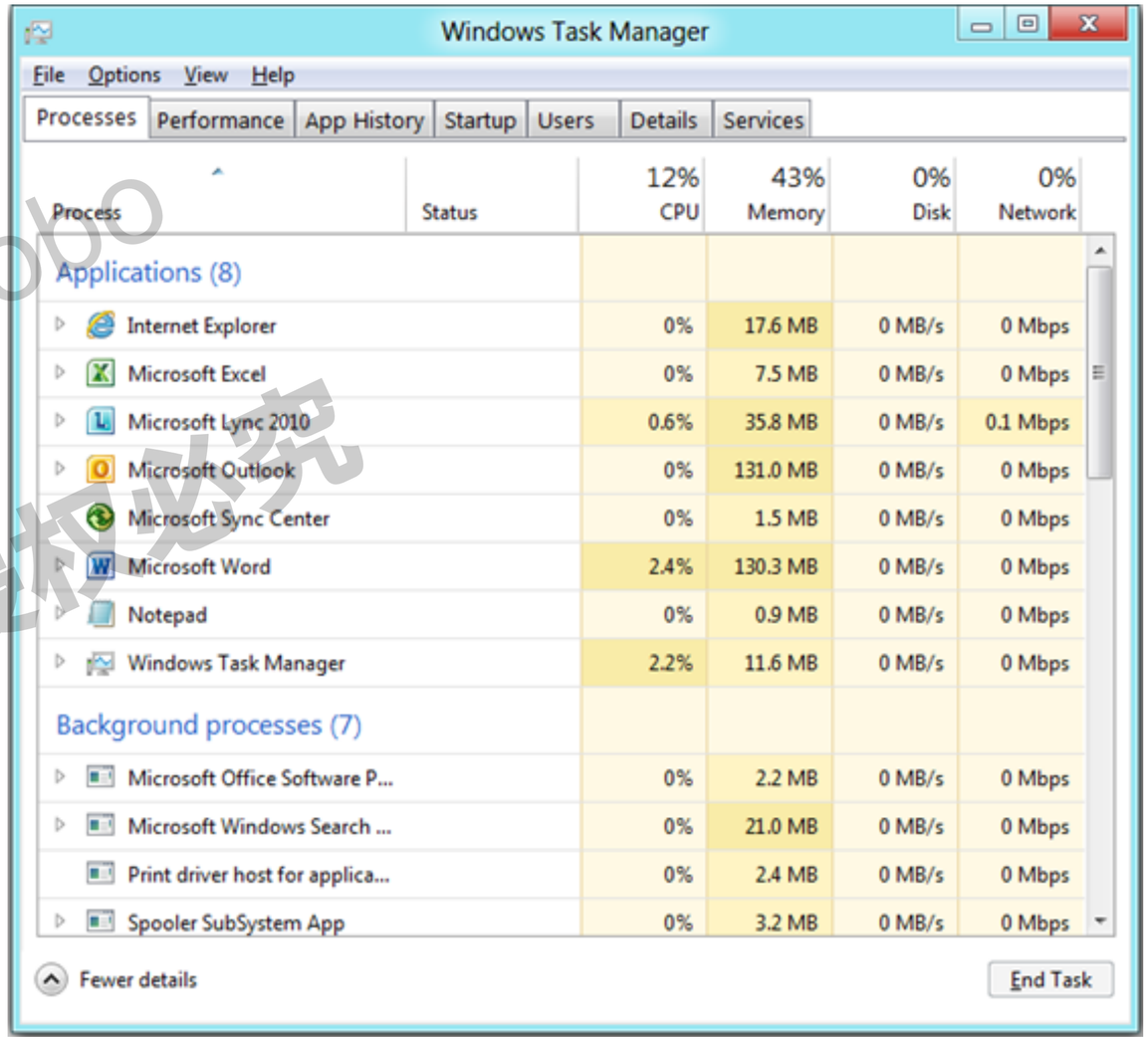
什么是优先队列?

普通队列：先进先出；后进后出

优先队列：出队顺序和入队顺序无关；和优先级相关

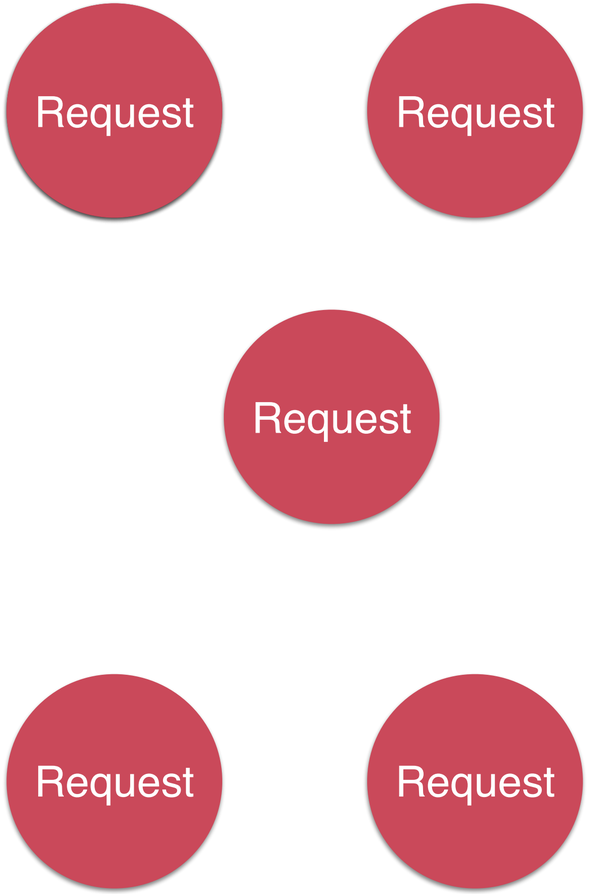
为什么使用优先队列?

动态选择优先级最高的任务执行

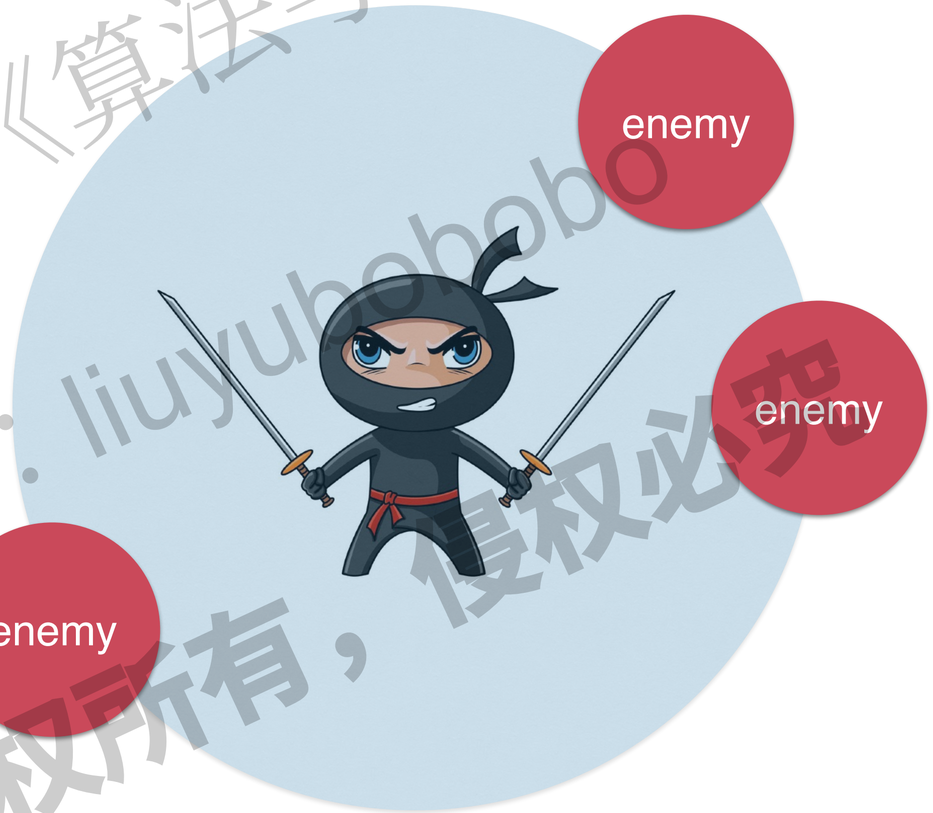


为什么使用优先队列?

关键词：动态



为什么使用优先队列?



enemy

enemy

enemy

enemy

enemy

enemy

enemy

为什么使用优先队列?

在1,000,000个元素中选出前100名?

在N个元素中选出前M个元素

排序? NlogN

使用优先队列? NlogM

优先队列主要操作

入队

出队（取出优先级最高的元素）

优先队列的实现

	入队	出队
普通数组	O(1)	O(n)
顺序数组	O(n)	O(1)
堆	O(lgn)	O(lgn)

使用堆实现优先队列

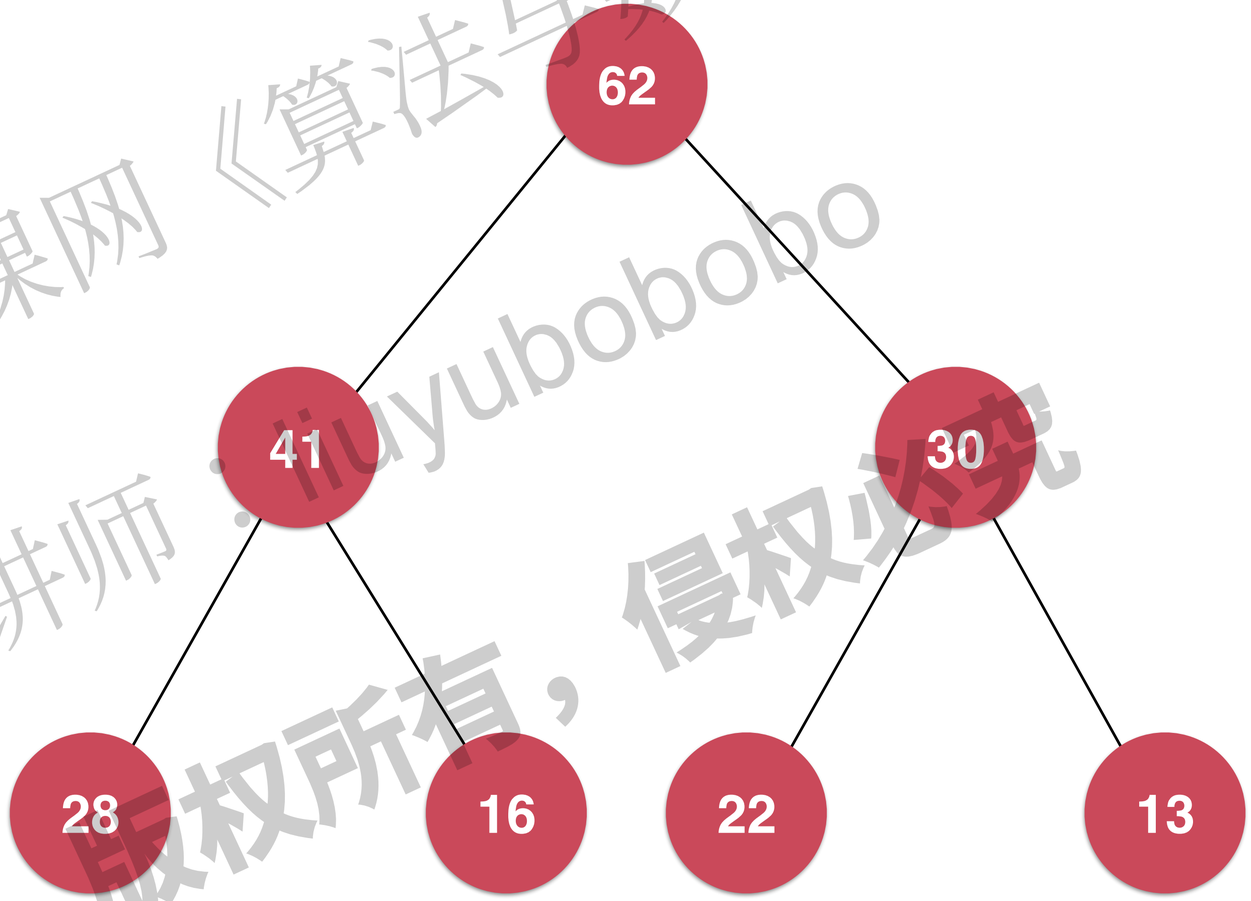
对于总共N个请求：

使用普通数组或者顺序数组，最差情况：O(n^2)

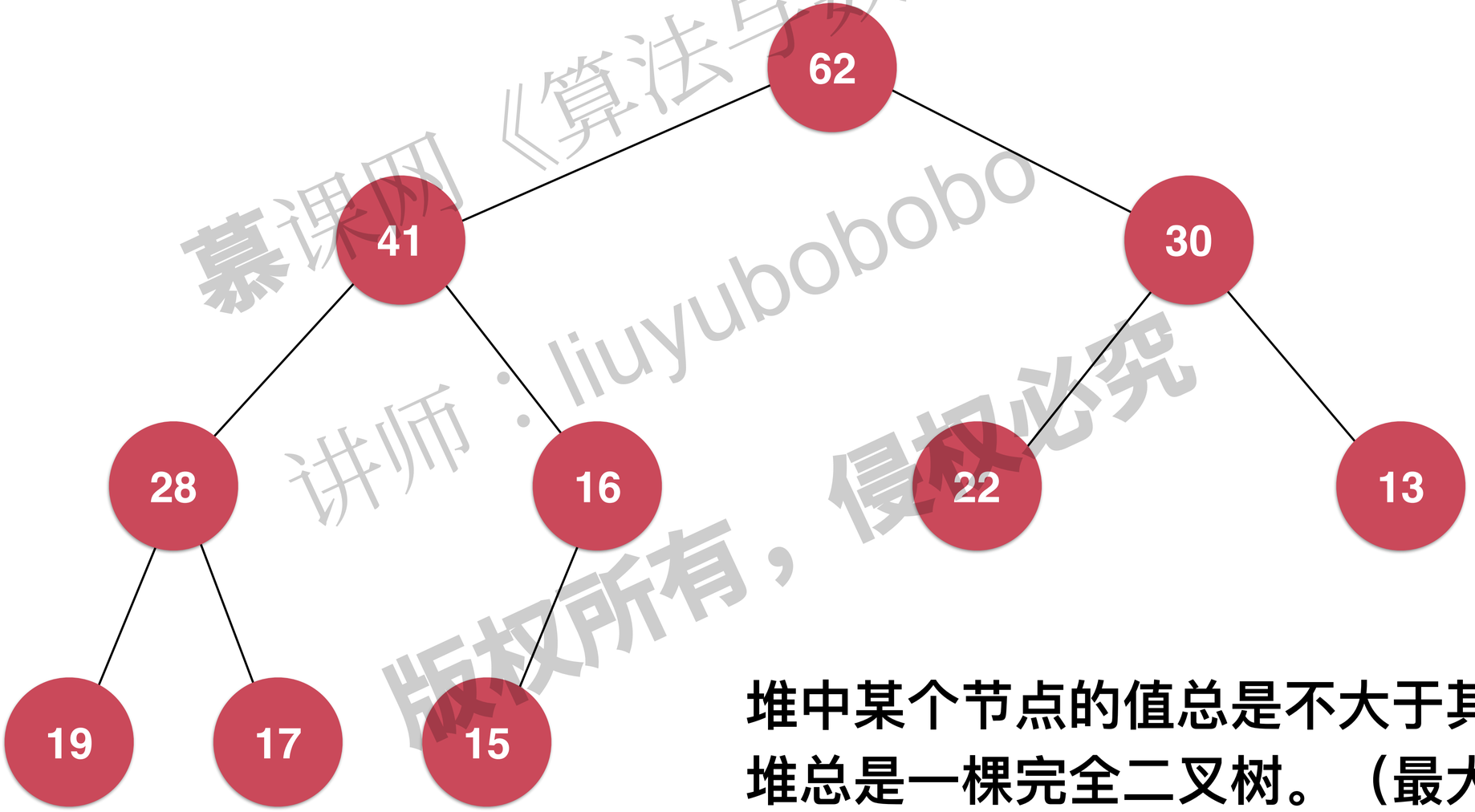
使用堆：O(nlgn)

堆的基本实现

二叉堆 Binary Heap

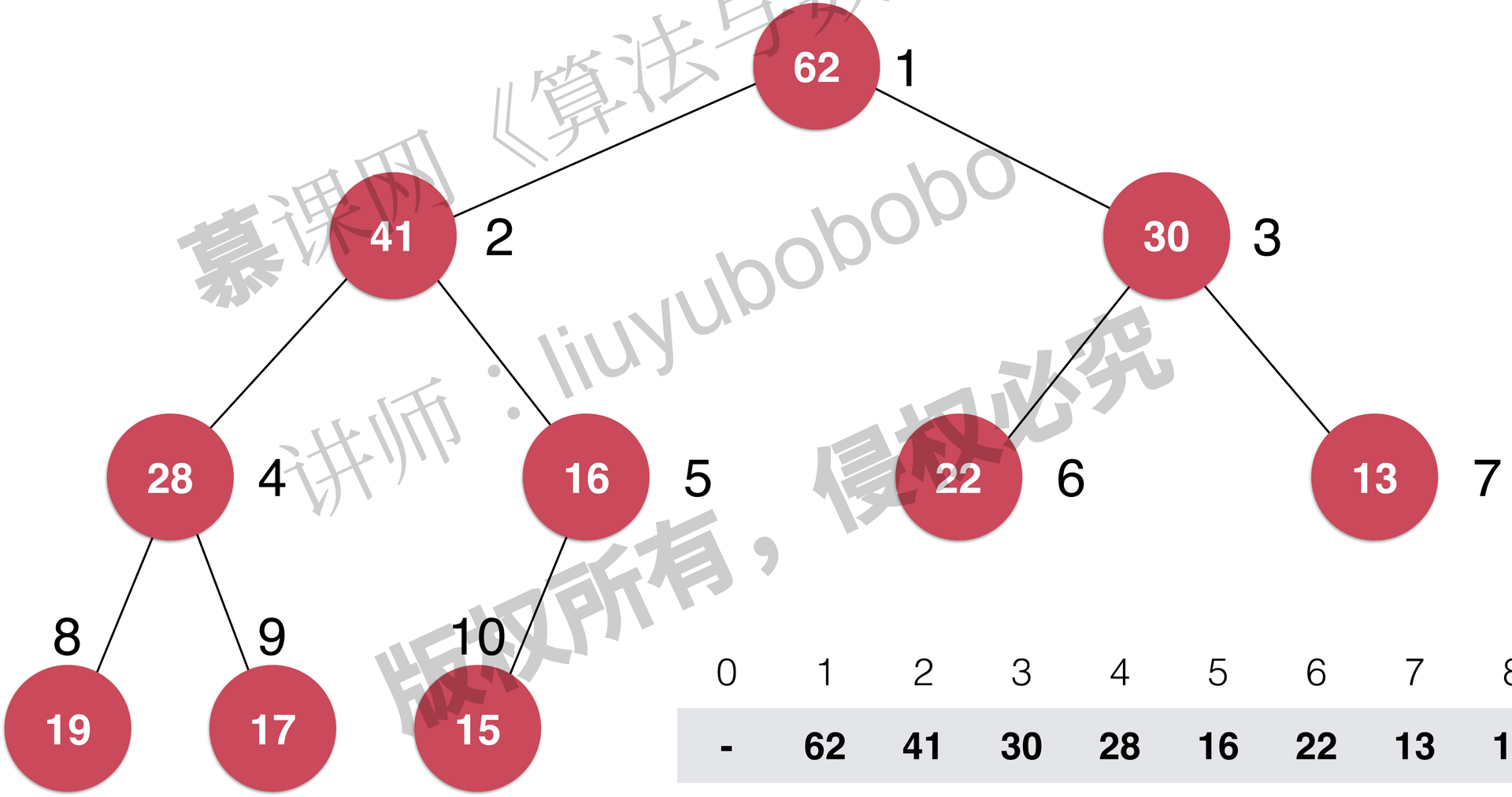


二叉堆 是一棵 完全二叉树



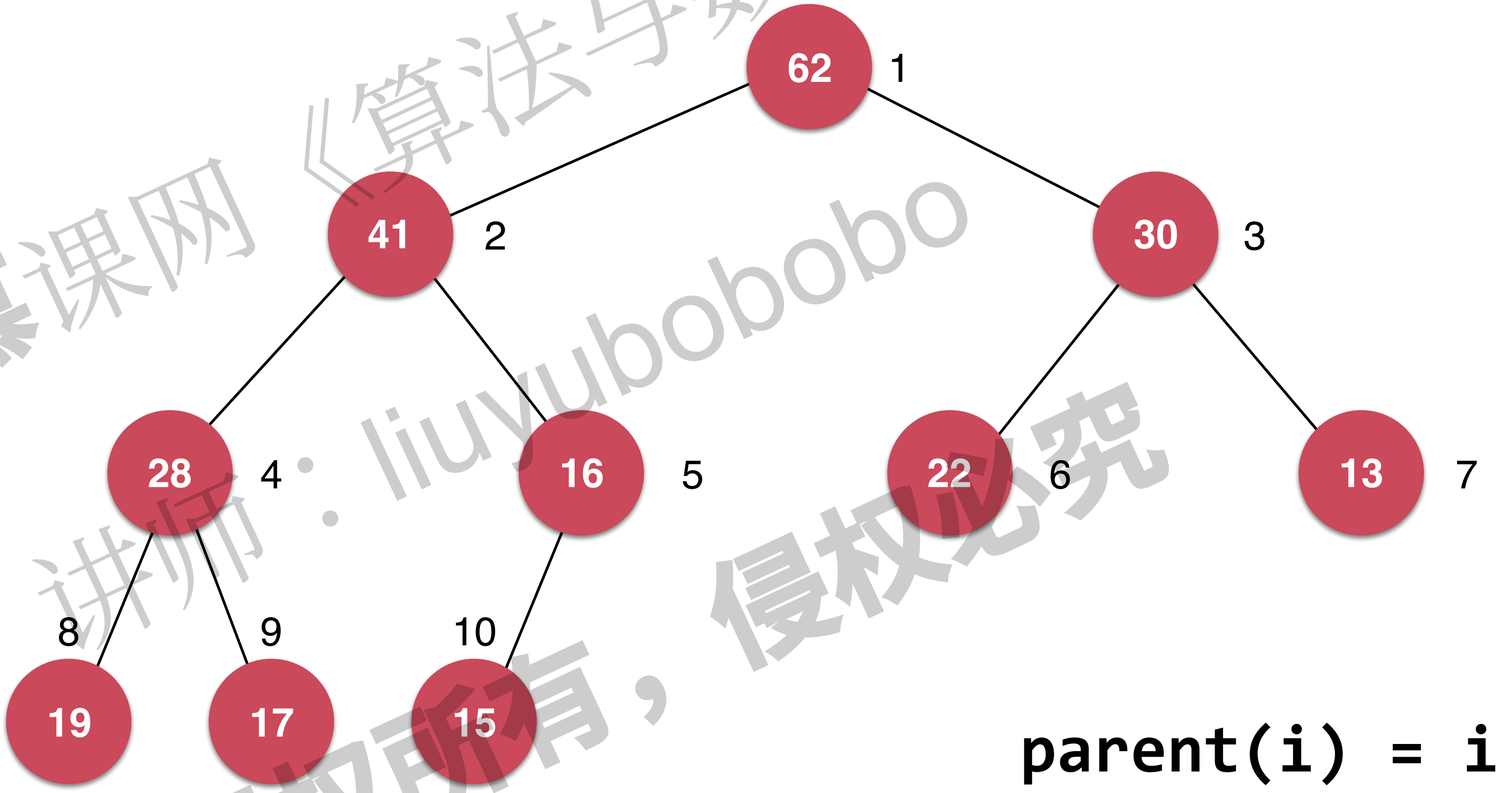
堆中某个节点的值总是不大于其父节点的值；
堆总是一棵完全二叉树。（最大堆）

用数组存储二叉堆



0	1	2	3	4	5	6	7	8	9	10
-	62	41	30	28	16	22	13	19	17	15

用数组存储二叉堆



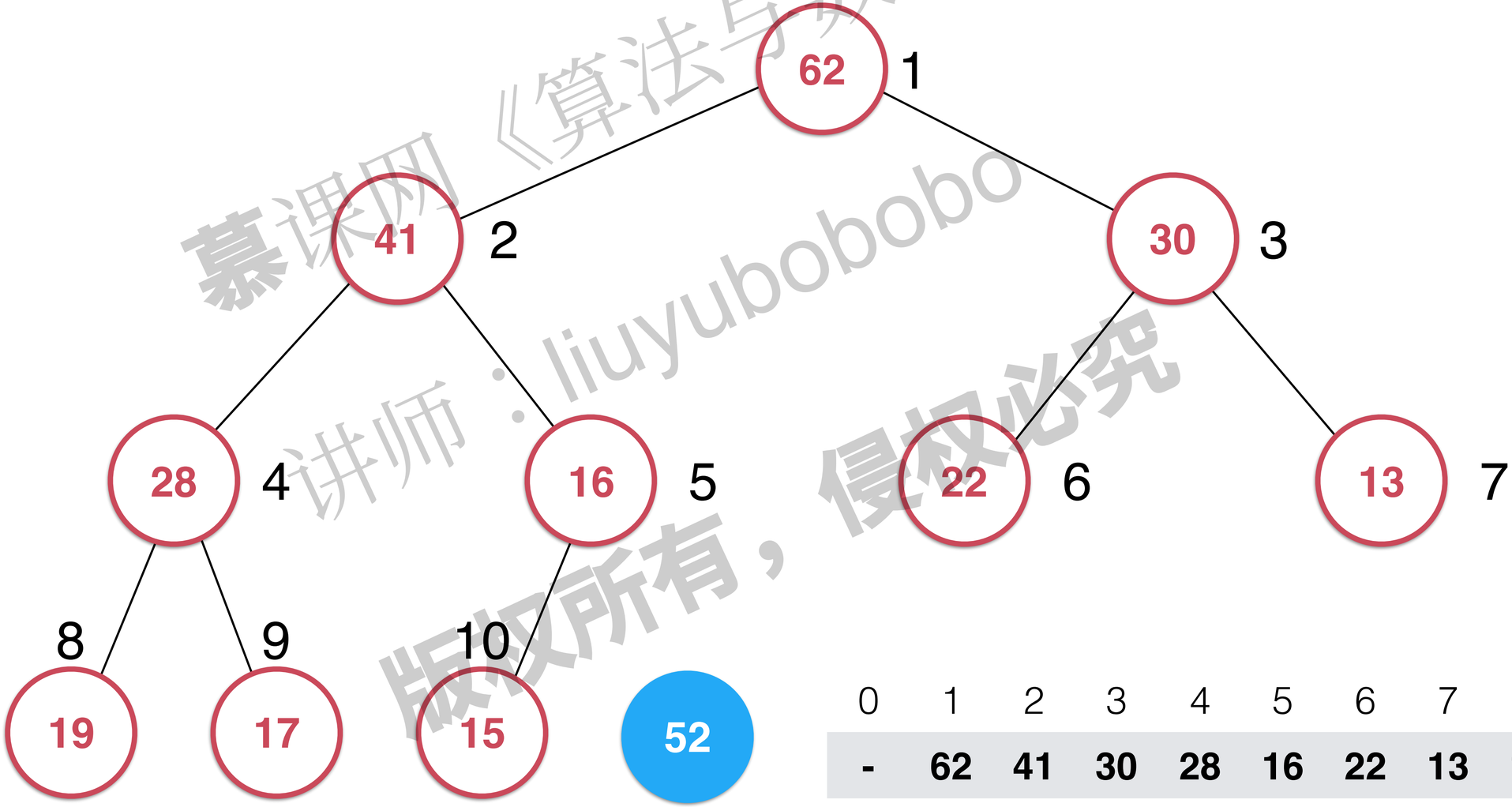
0	1	2	3	4	5	6	7	8	9	10
-	62	41	30	28	16	22	13	19	17	15

```
parent(i) = i/2

left child  (i) = 2*i
right child (i) = 2*i + 1
```

操作：堆的基本框架

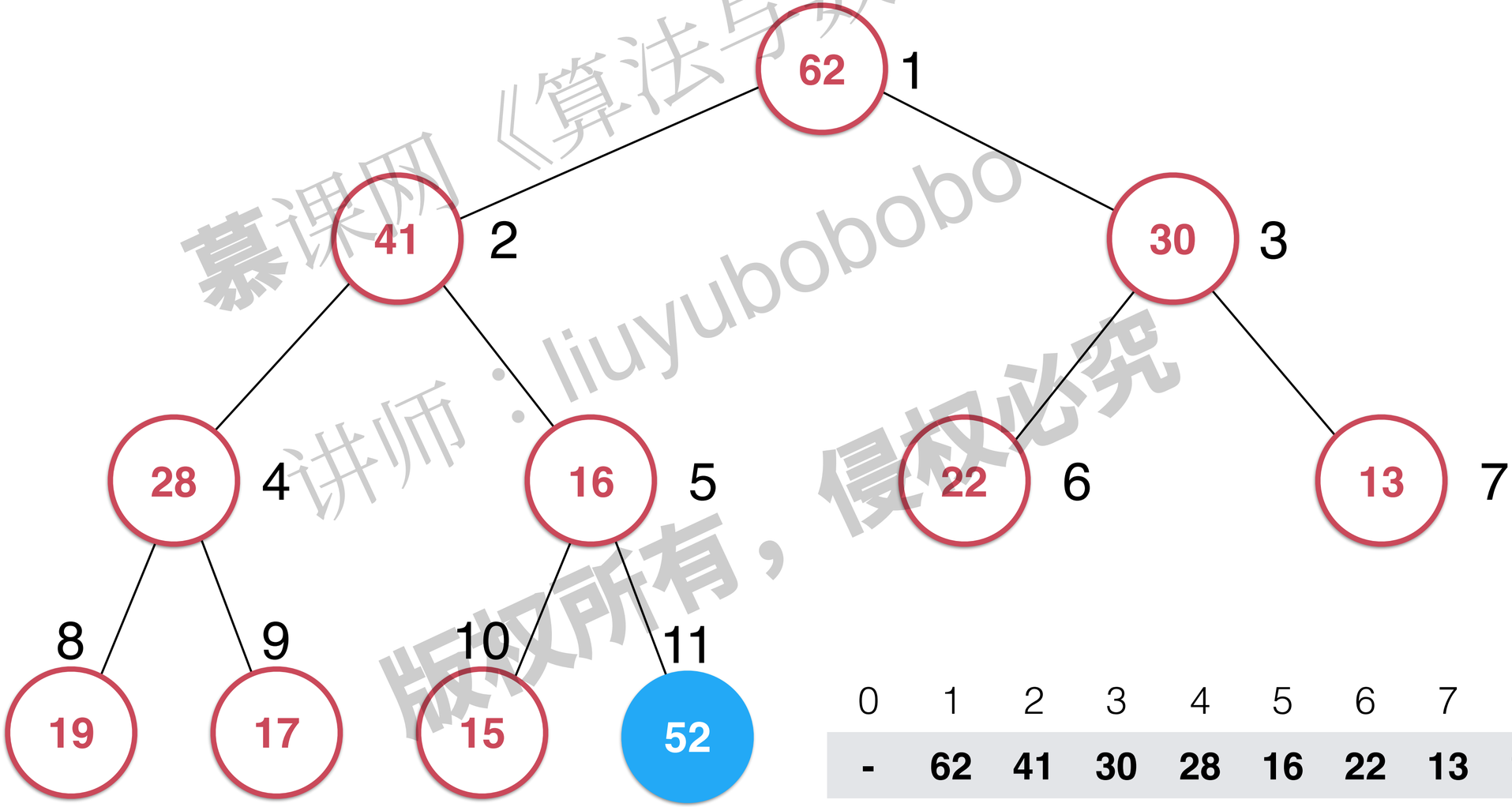
Shift Up

Shift Up



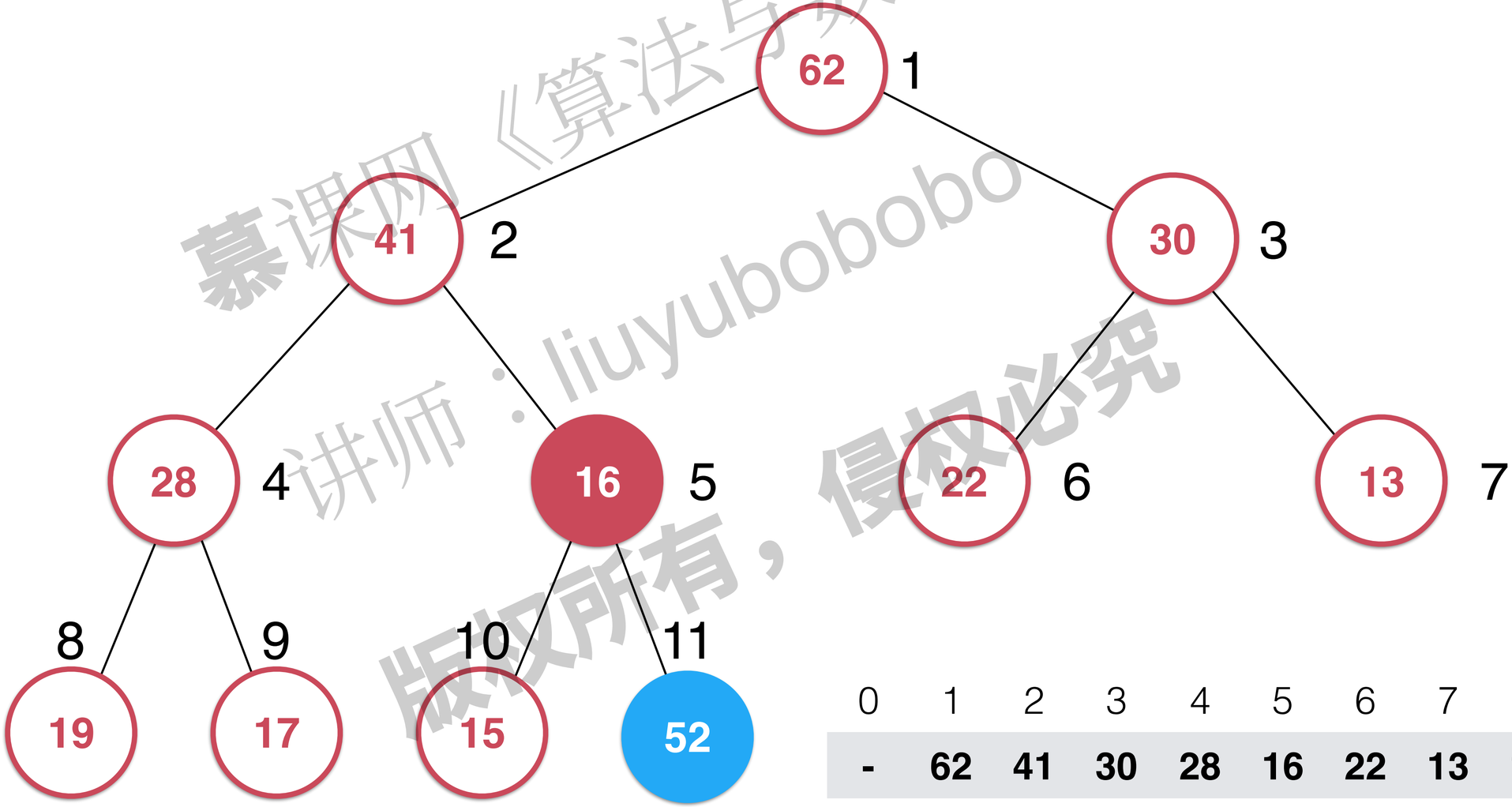
0	1	2	3	4	5	6	7	8	9	10
-	62	41	30	28	16	22	13	19	17	15

Shift Up



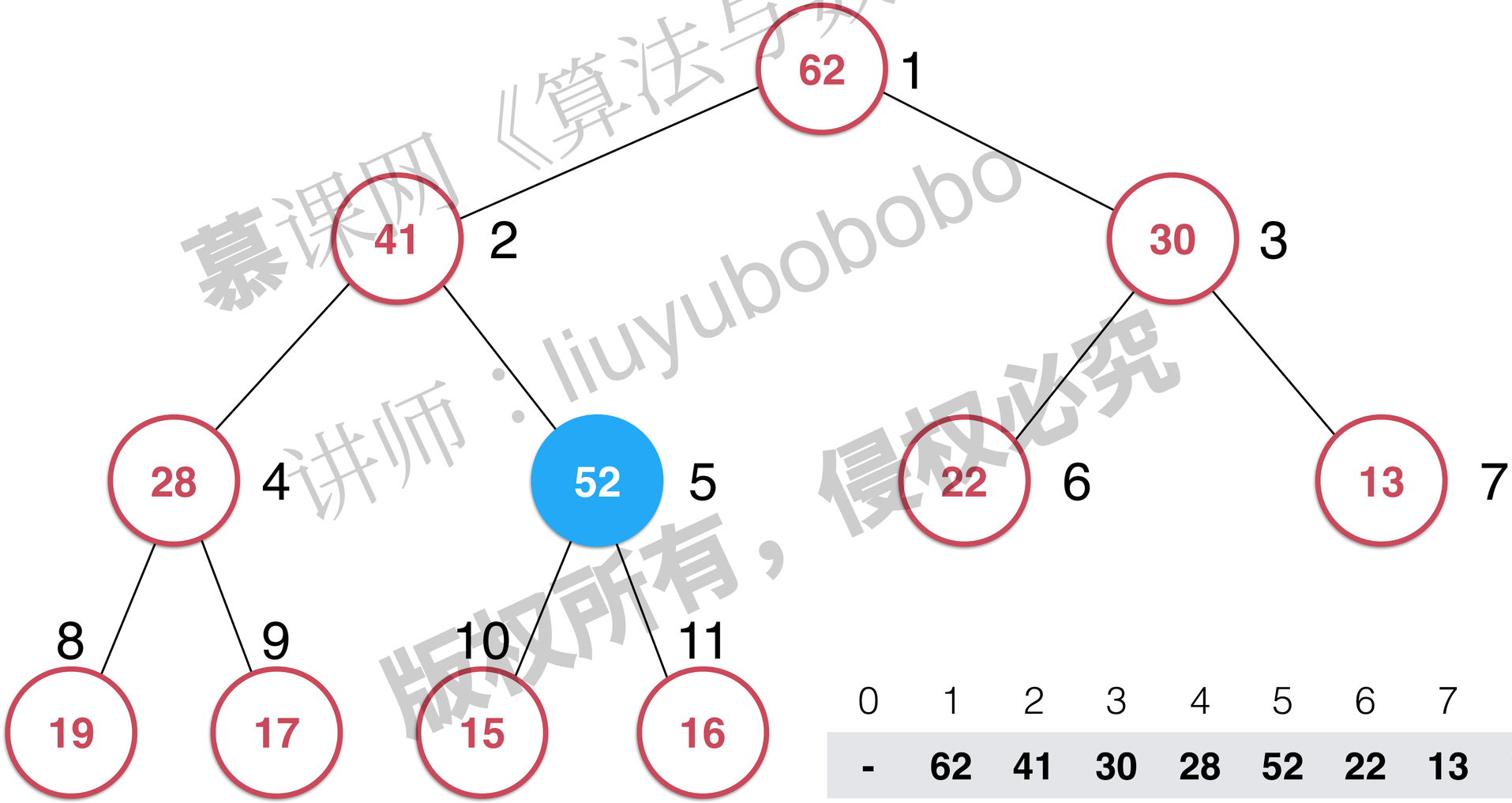
0	1	2	3	4	5	6	7	8	9	10	11
-	62	41	30	28	16	22	13	19	17	15	52

Shift Up



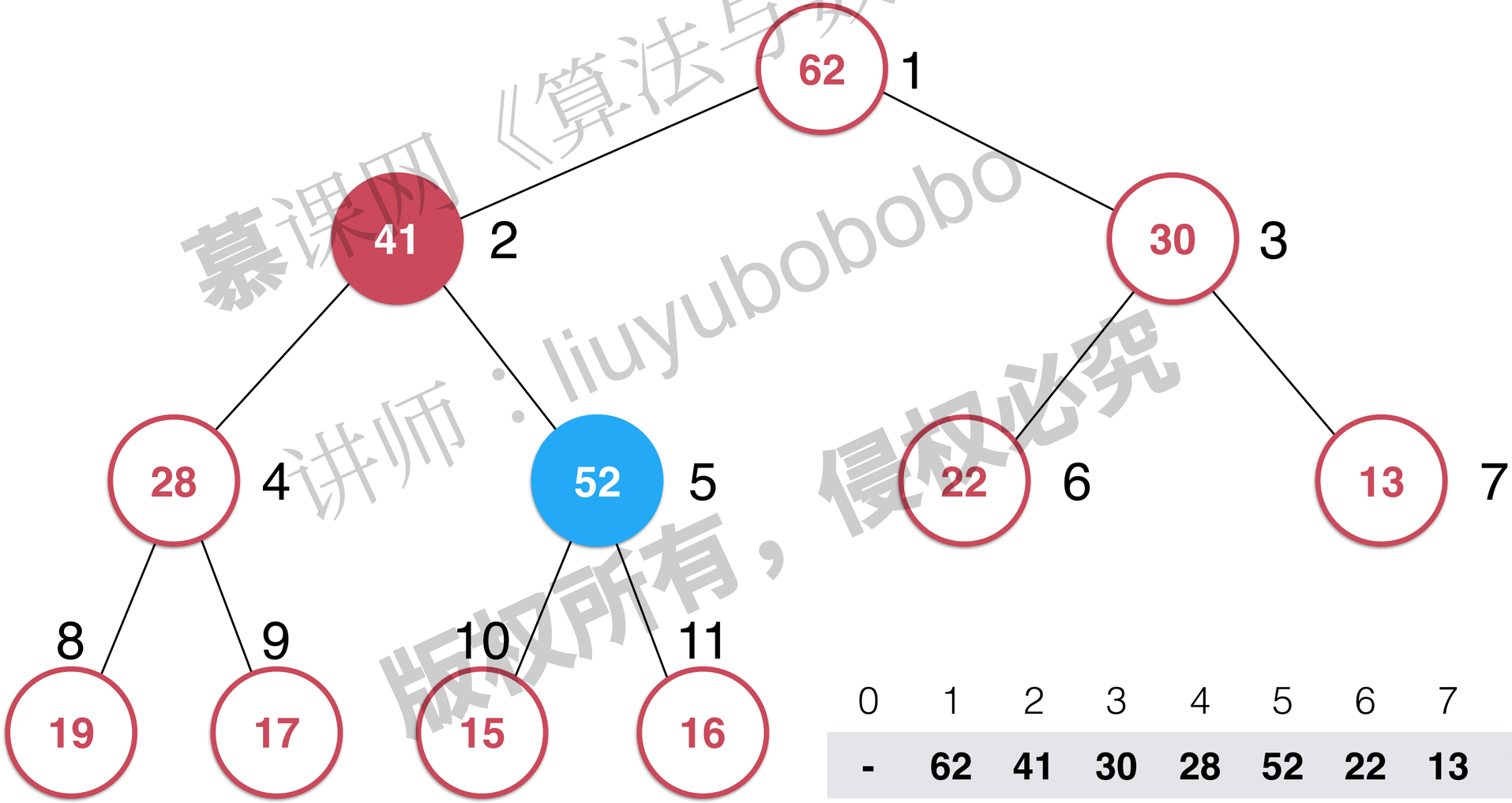
0	1	2	3	4	5	6	7	8	9	10	11
-	62	41	30	28	16	22	13	19	17	15	52

Shift Up



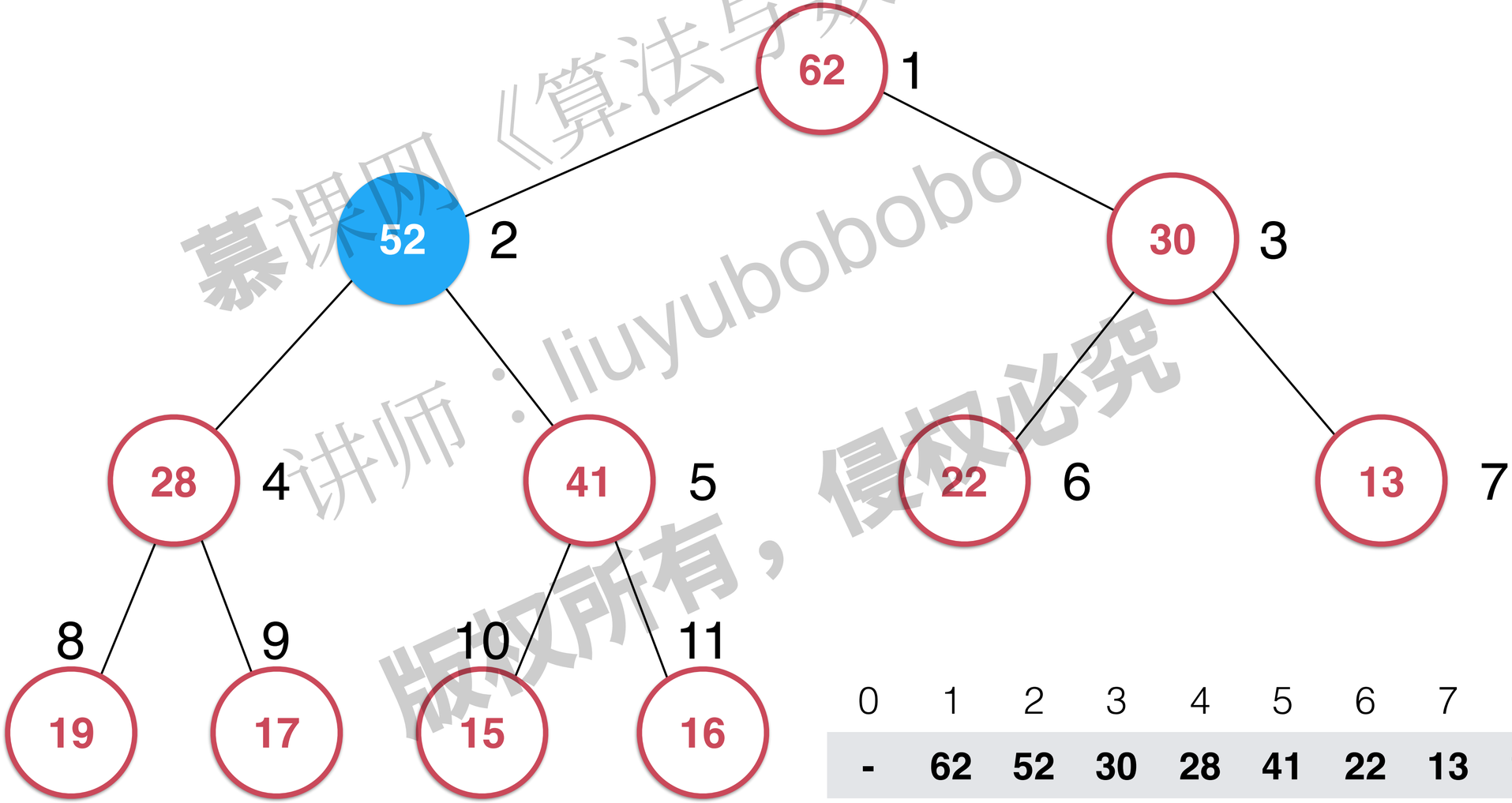
0	1	2	3	4	5	6	7	8	9	10	11
-	62	41	30	28	52	22	13	19	17	15	16

Shift Up



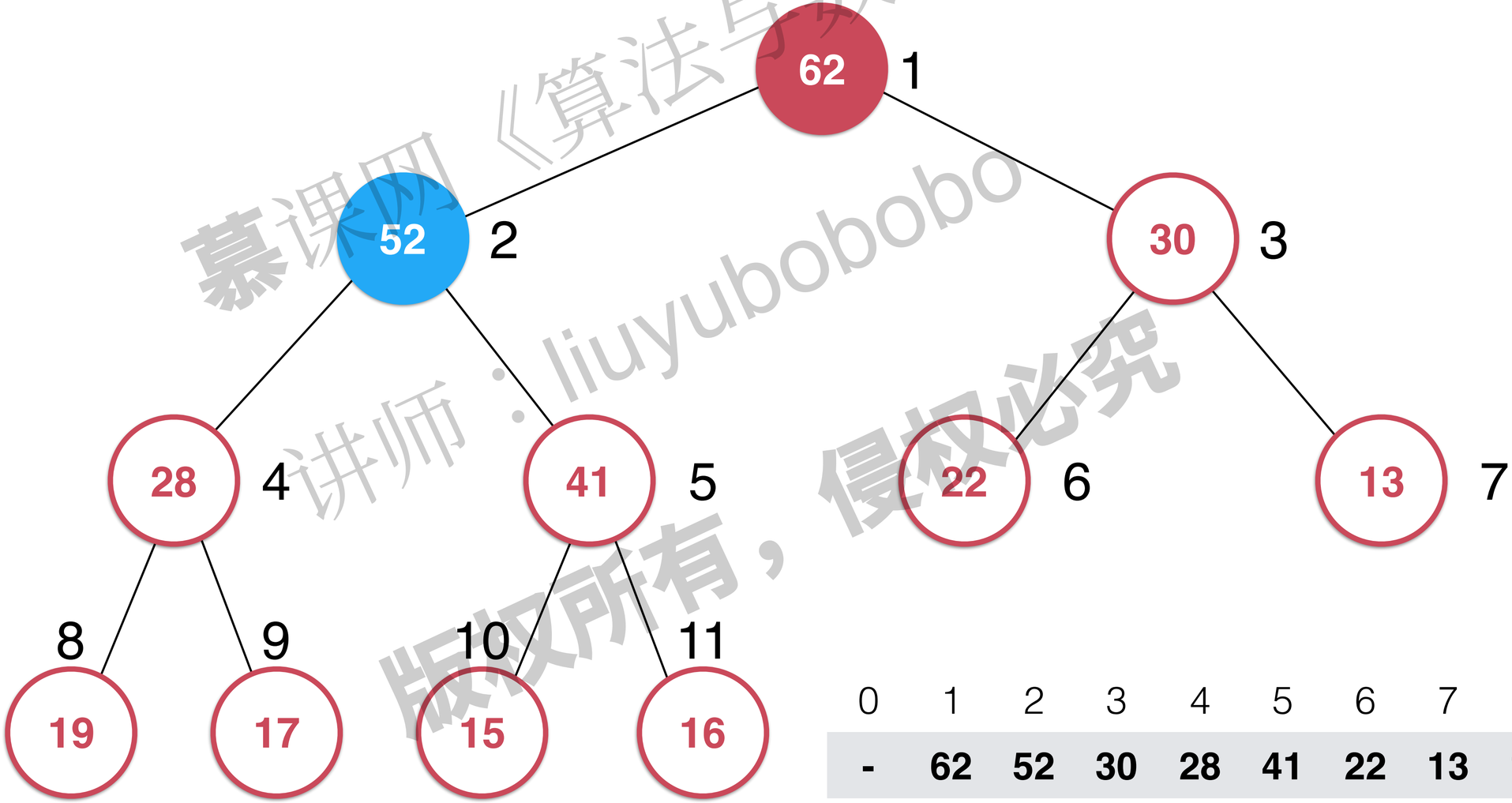
0	1	2	3	4	5	6	7	8	9	10	11
-	62	41	30	28	52	22	13	19	17	15	16

Shift Up



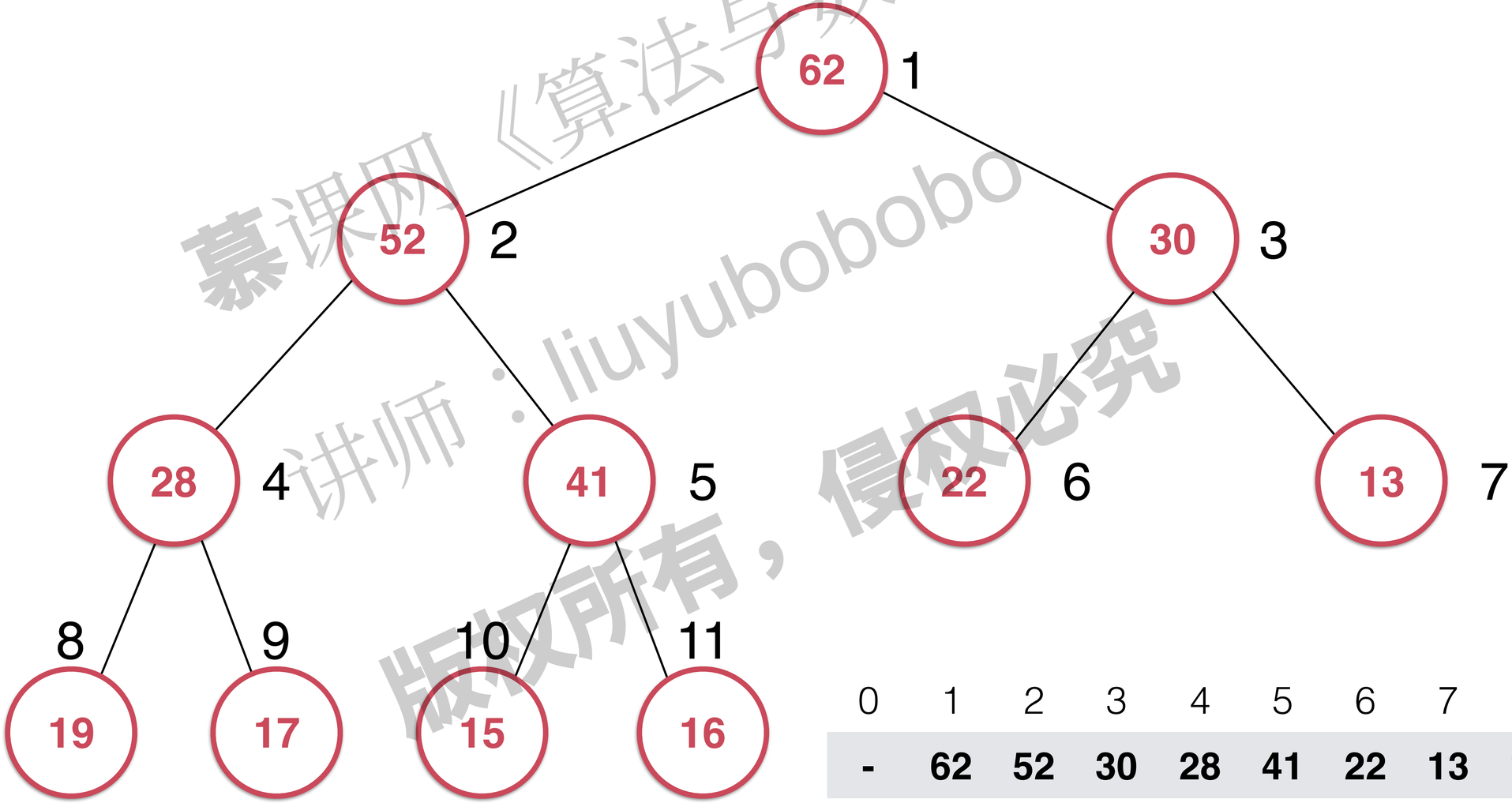
0	1	2	3	4	5	6	7	8	9	10	11
-	62	52	30	28	41	22	13	19	17	15	16

Shift Up



0	1	2	3	4	5	6	7	8	9	10	11
-	62	52	30	28	41	22	13	19	17	15	16

Shift Up

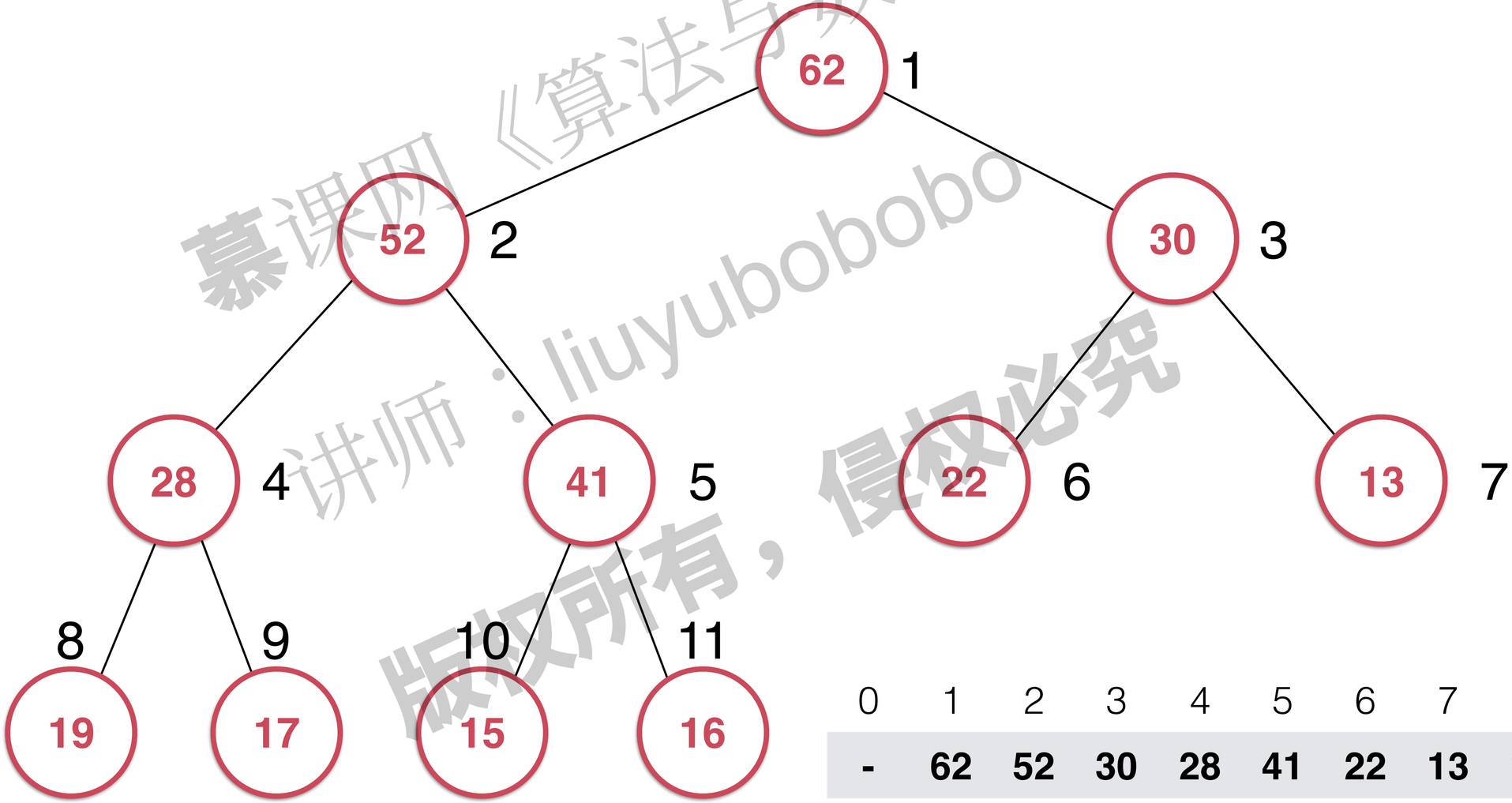


0	1	2	3	4	5	6	7	8	9	10	11
-	62	52	30	28	41	22	13	19	17	15	16

操作： Shift Up 和 insert

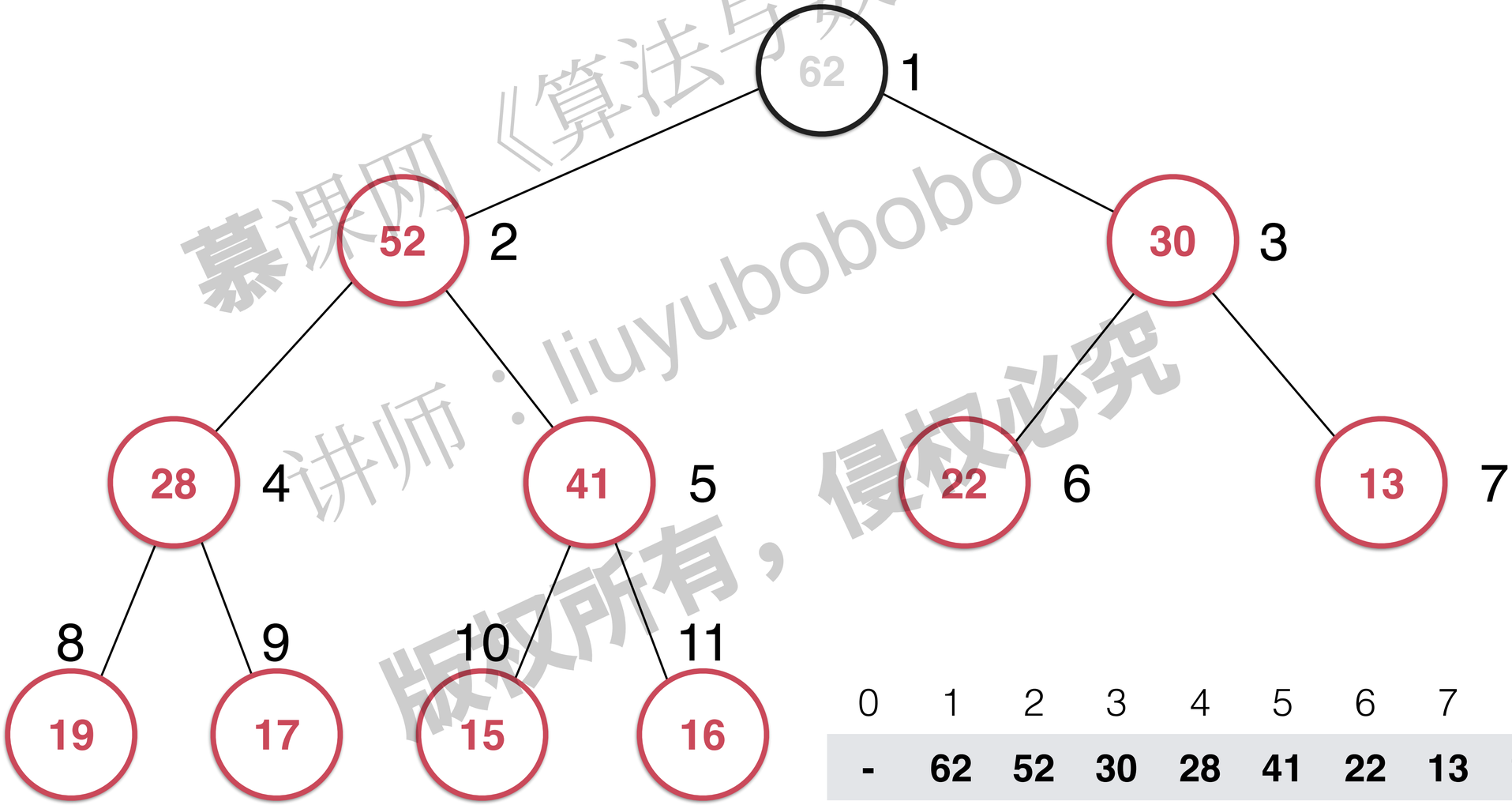
Shift Down

Shift Down



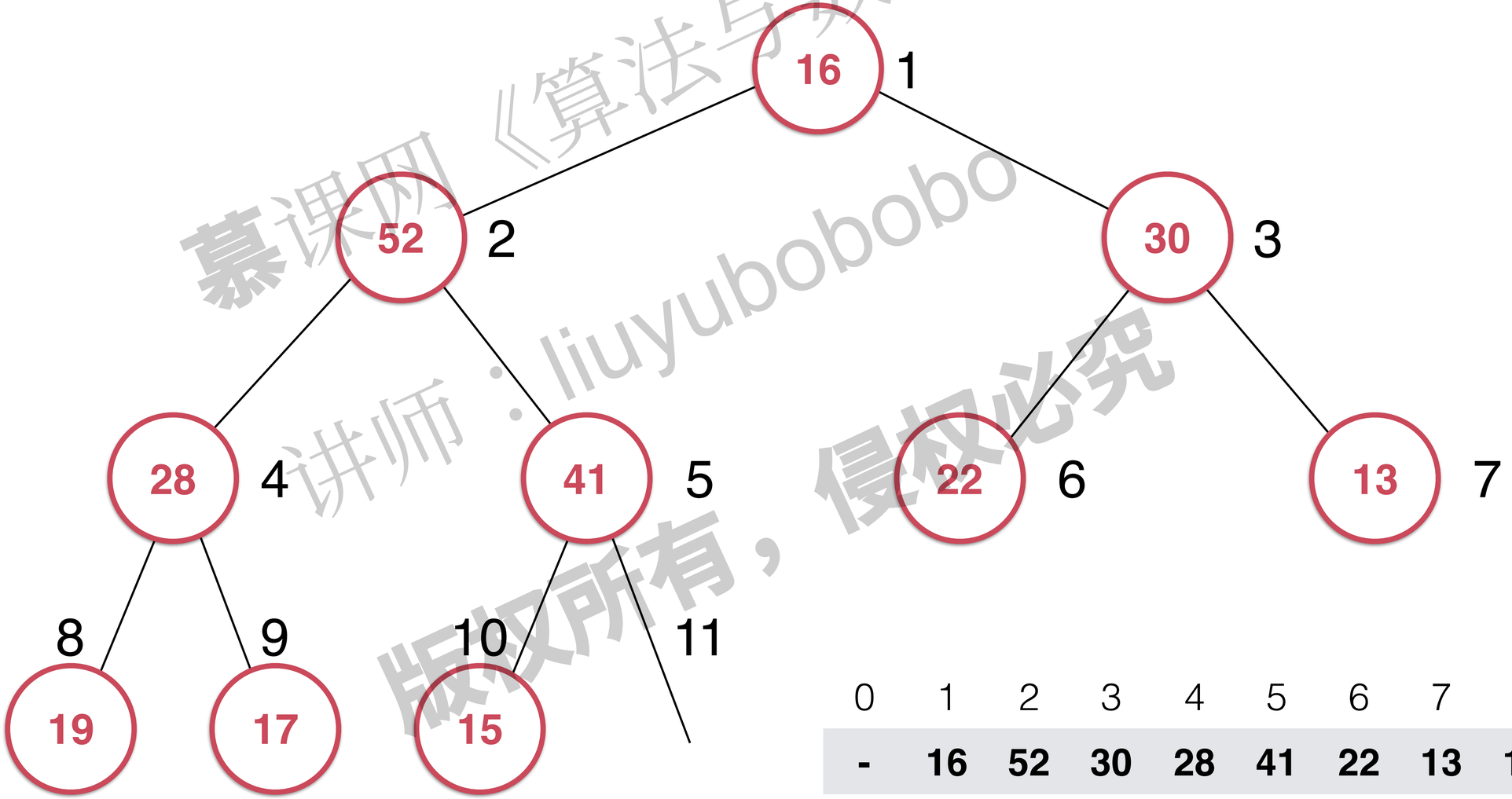
0	1	2	3	4	5	6	7	8	9	10	11
-	62	52	30	28	41	22	13	19	17	15	16

Shift Down



0	1	2	3	4	5	6	7	8	9	10	11
-	62	52	30	28	41	22	13	19	17	15	16

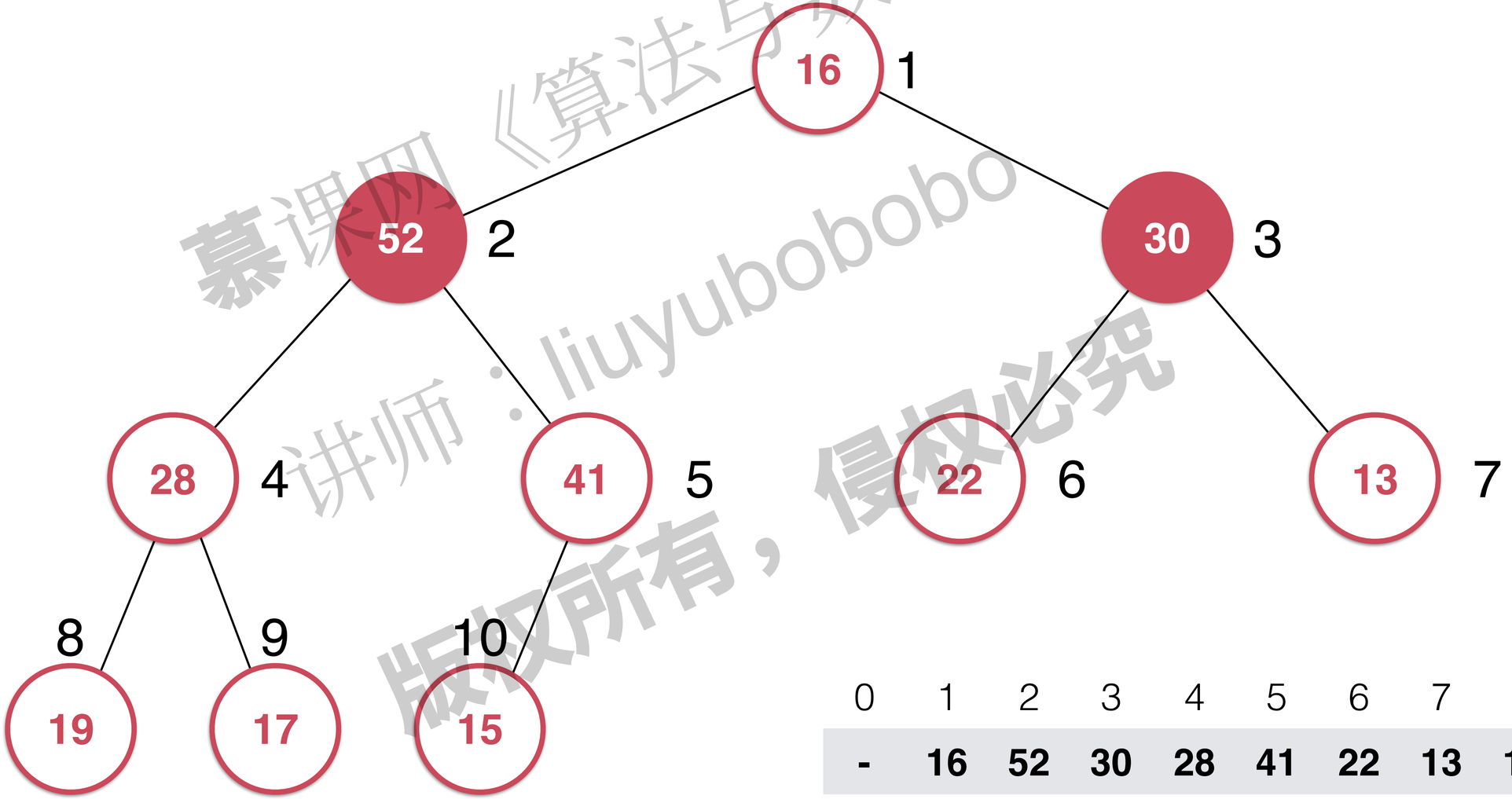
Shift Down



count

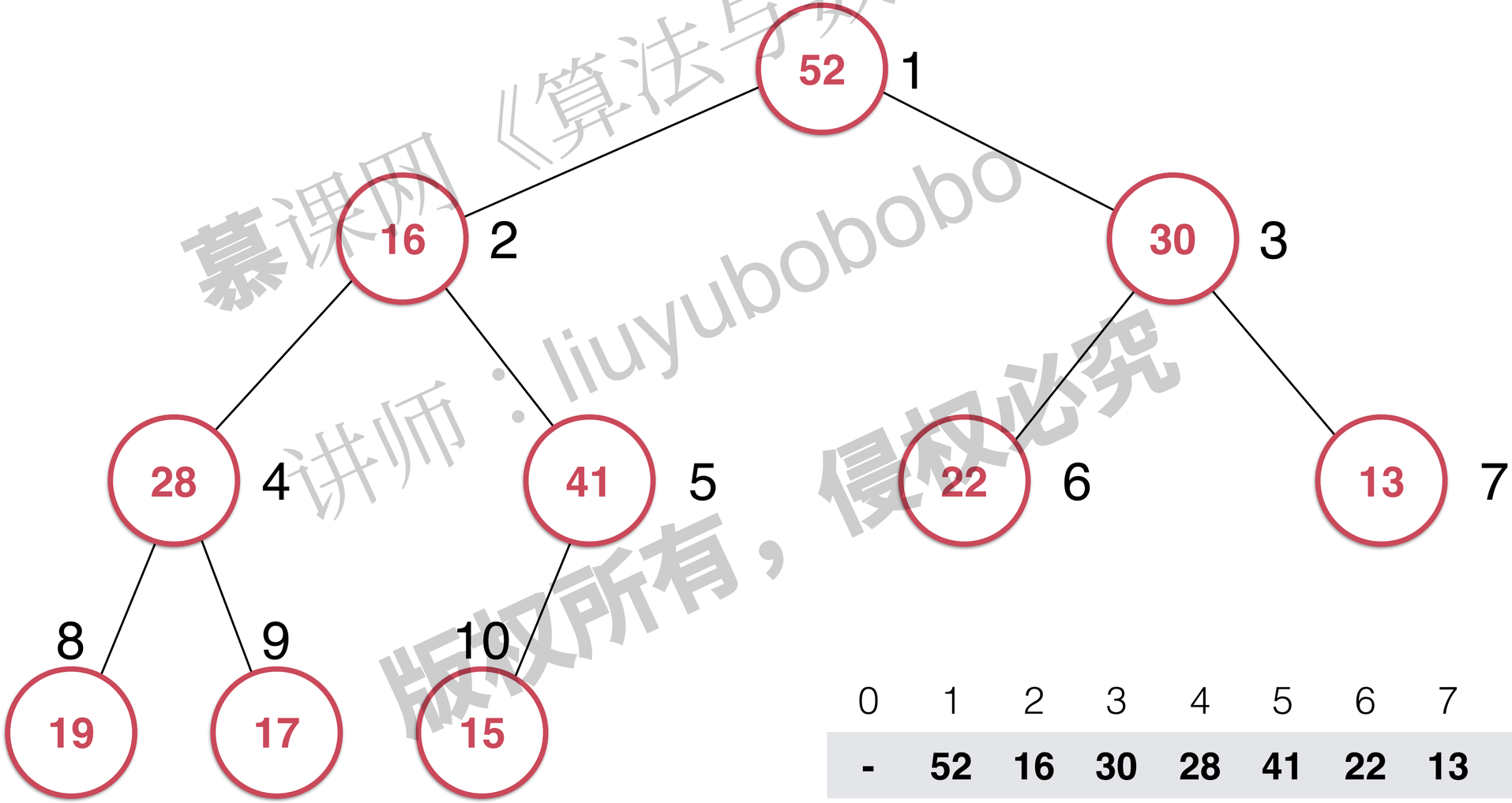
0	1	2	3	4	5	6	7	8	9	10	11
-	16	52	30	28	41	22	13	19	17	15	16

Shift Down



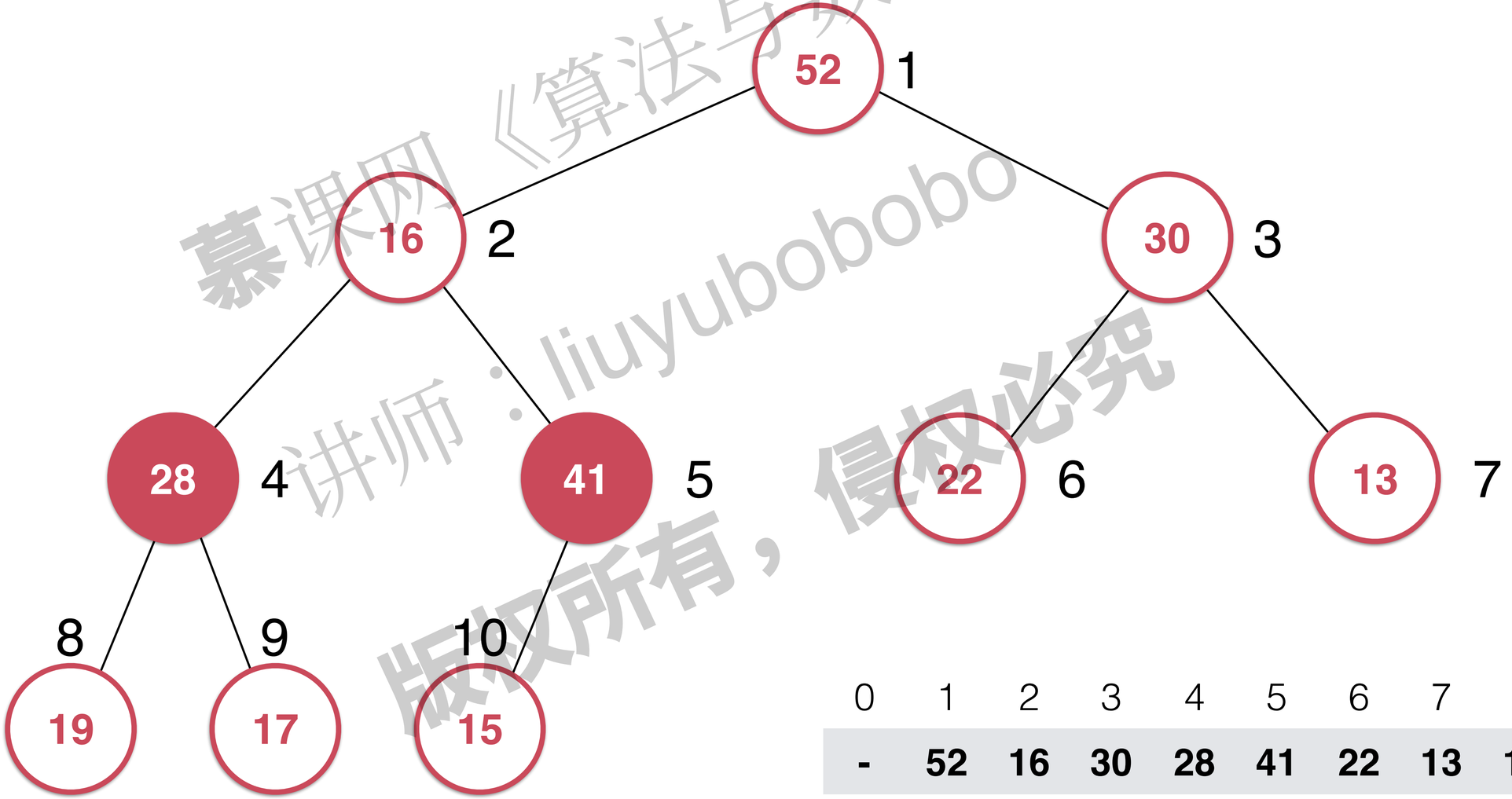
0	1	2	3	4	5	6	7	8	9	10
-	16	52	30	28	41	22	13	19	17	15

Shift Down



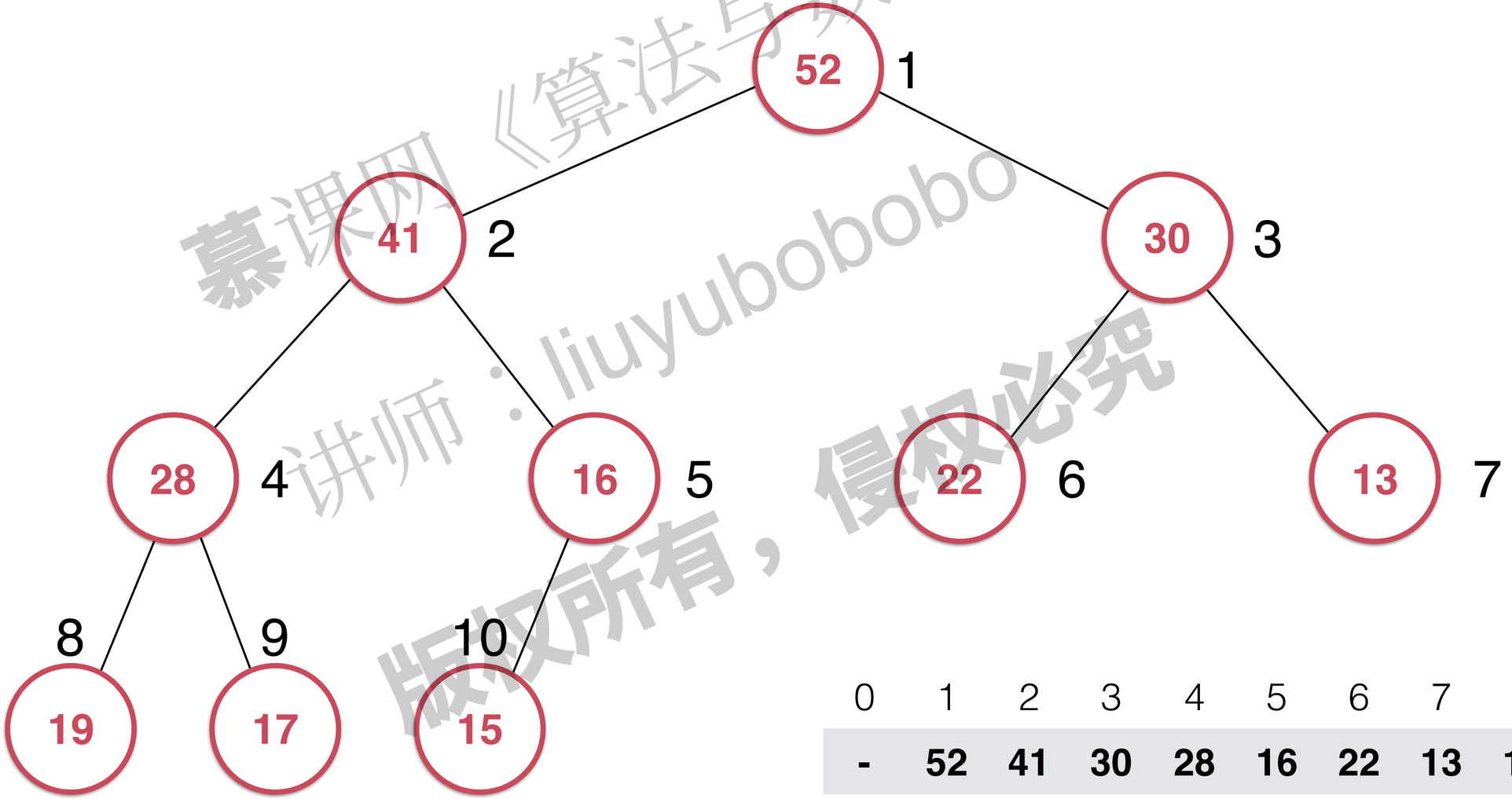
0	1	2	3	4	5	6	7	8	9	10
-	52	16	30	28	41	22	13	19	17	15

Shift Down



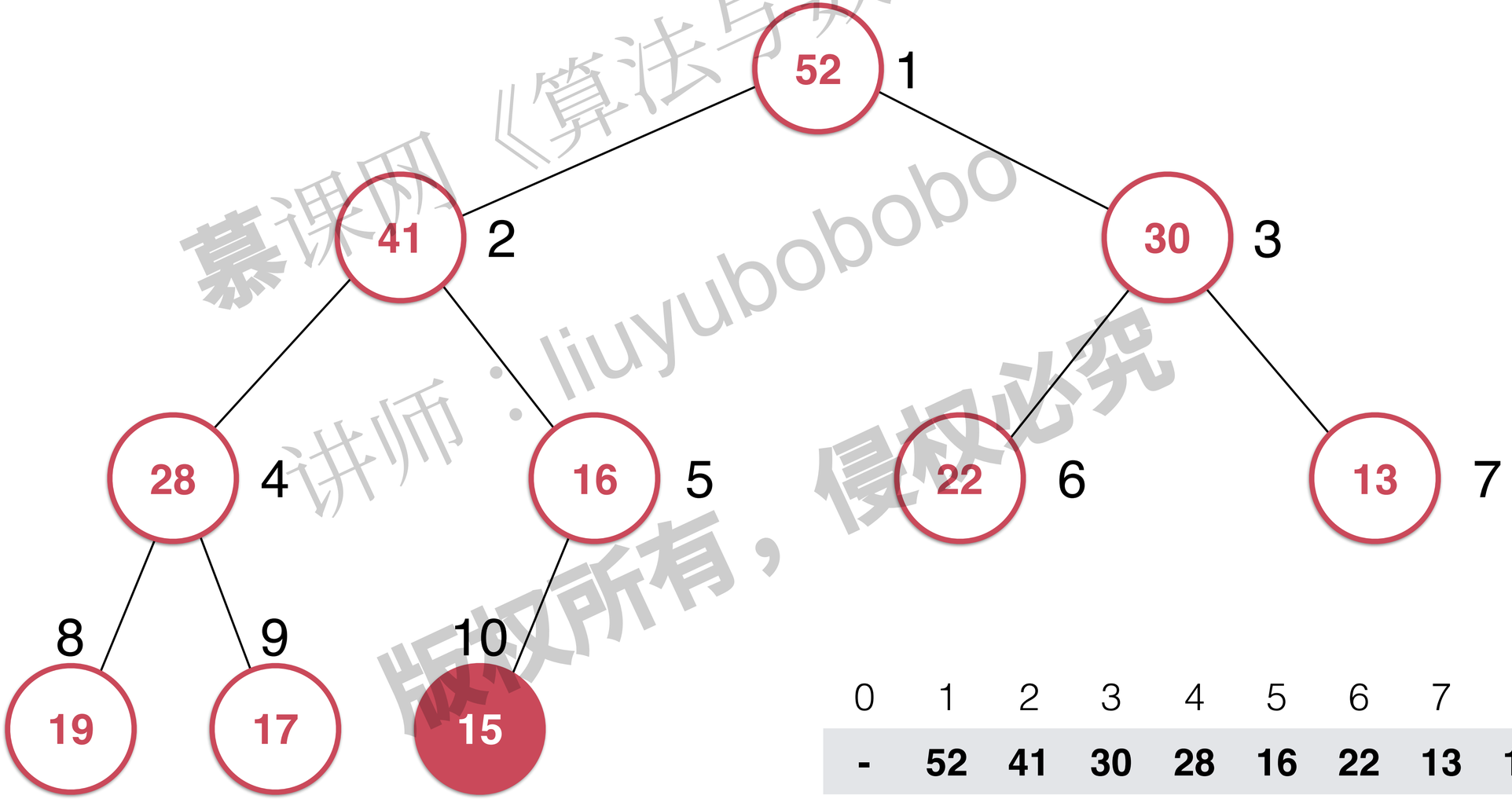
0	1	2	3	4	5	6	7	8	9	10
-	52	16	30	28	41	22	13	19	17	15

Shift Down



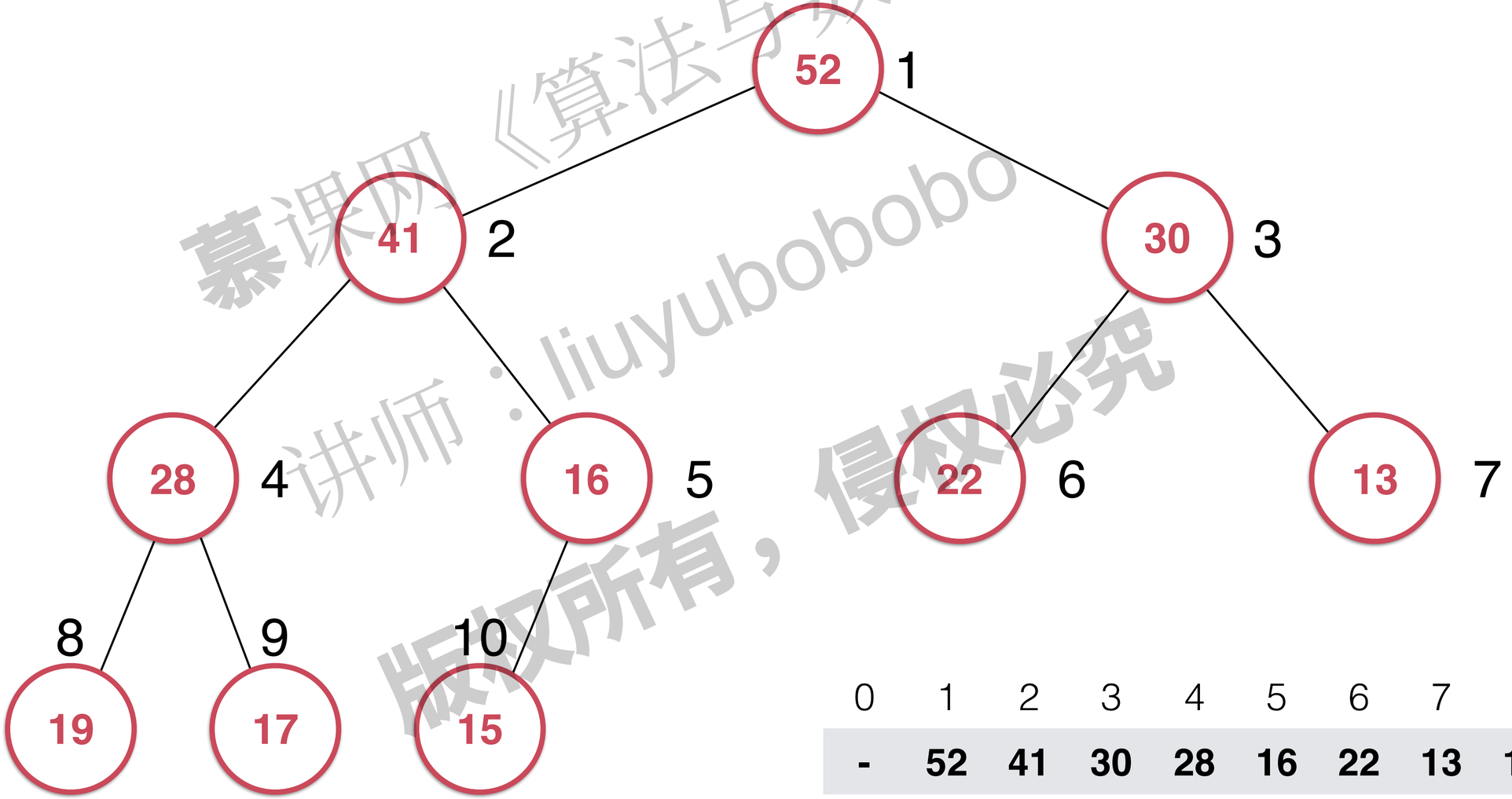
0	1	2	3	4	5	6	7	8	9	10
-	52	41	30	28	16	22	13	19	17	15

Shift Down



0	1	2	3	4	5	6	7	8	9	10
-	52	41	30	28	16	22	13	19	17	15

Shift Down



0	1	2	3	4	5	6	7	8	9	10
-	52	41	30	28	16	22	13	19	17	15

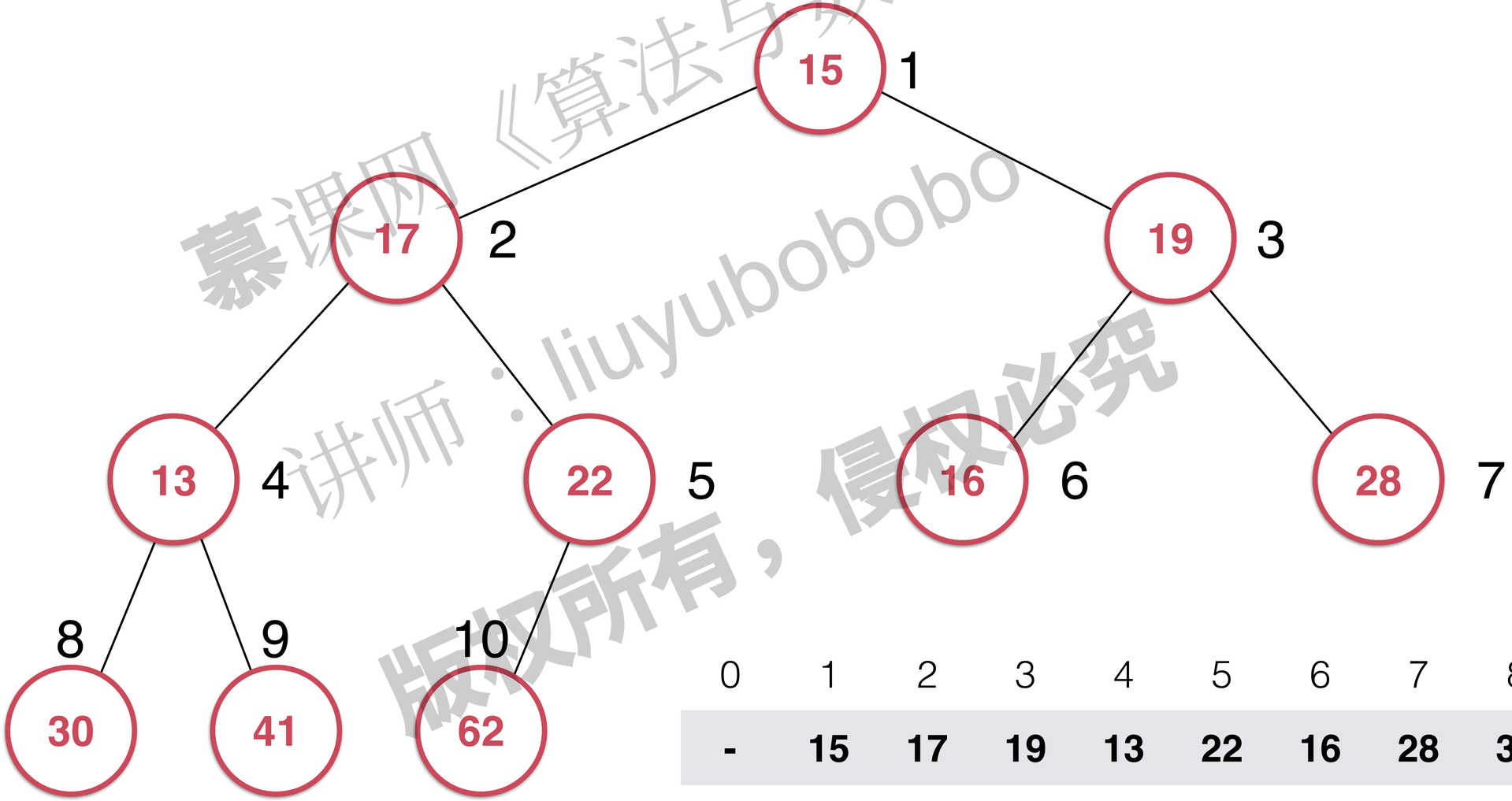
操作：Shift Down 和 extract Max

操作：Basic Heap Sort

操作：Basic Heap Sort 和 Merge Sort, Quick Sort 的比较

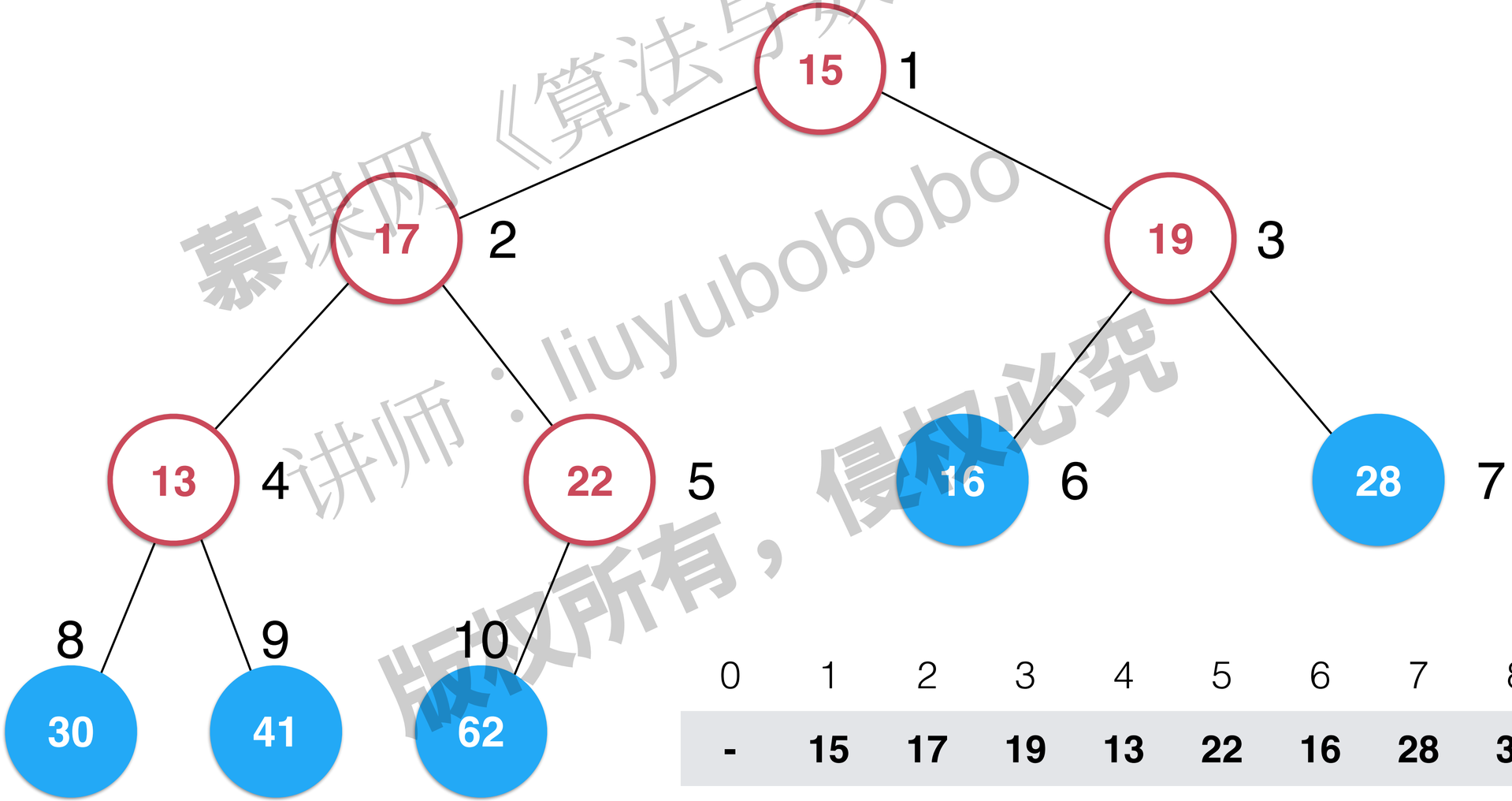
Heapify

Heapify



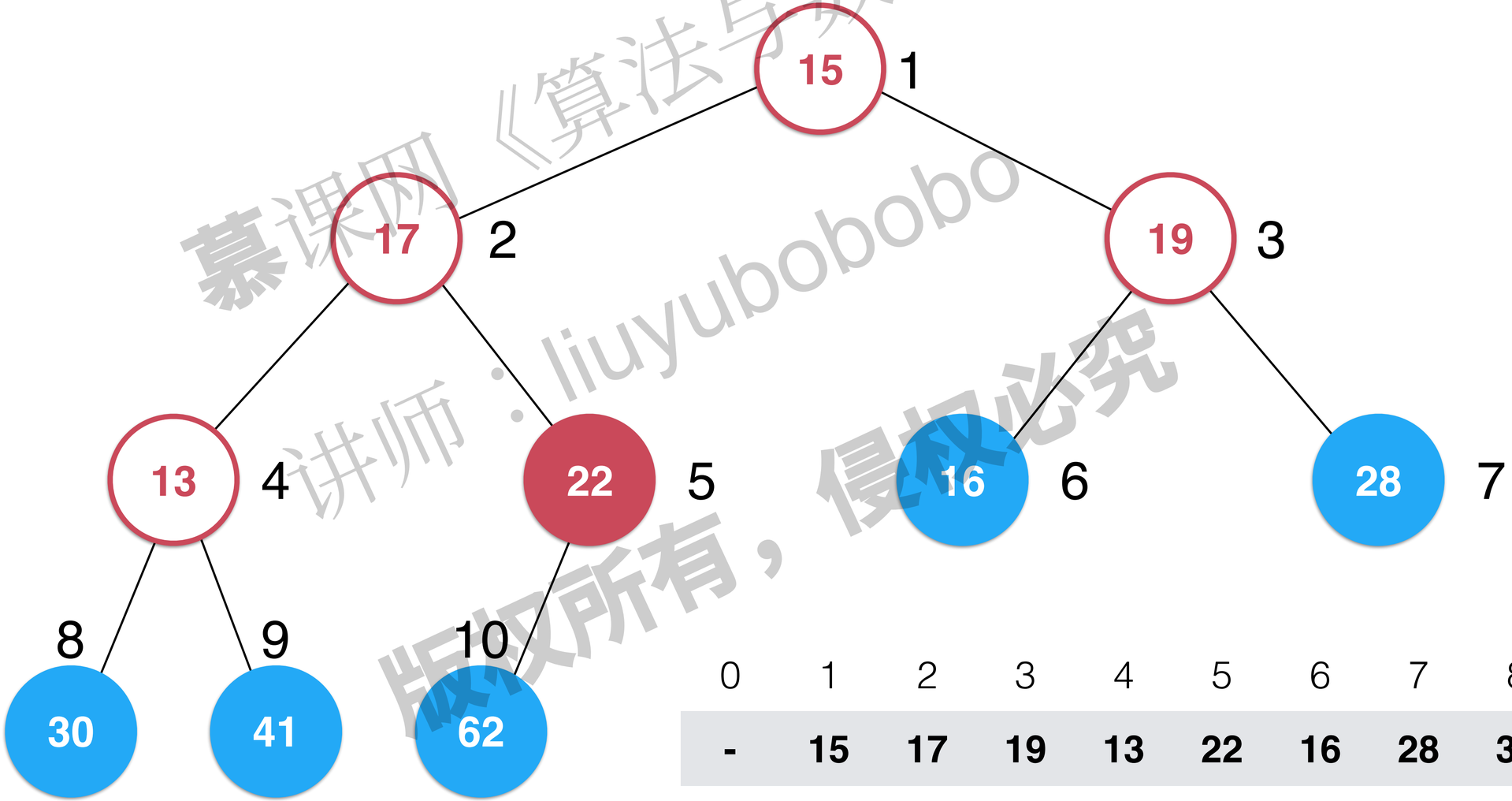
0	1	2	3	4	5	6	7	8	9	10
-	15	17	19	13	22	16	28	30	41	62

Heapify



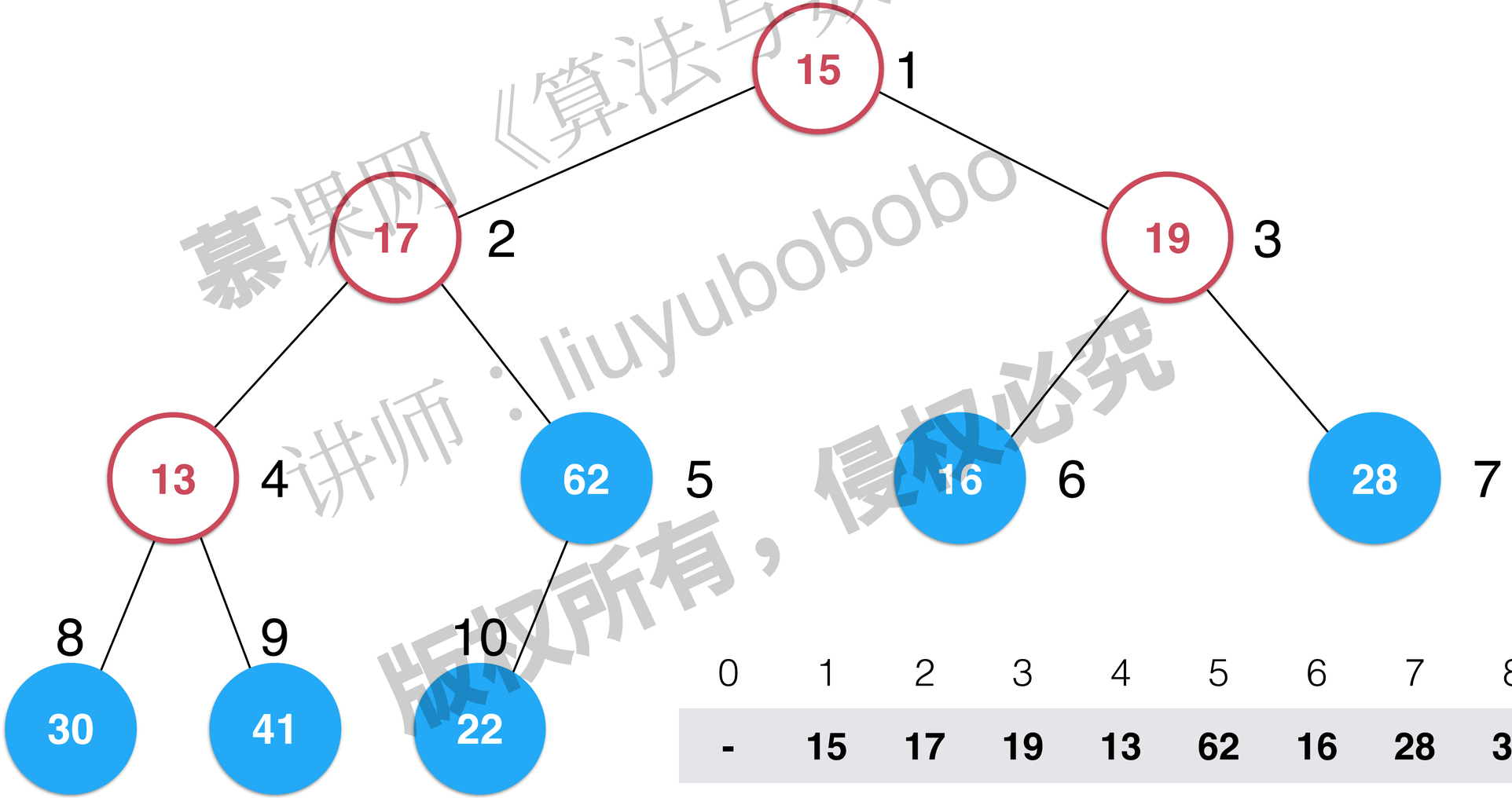
0	1	2	3	4	5	6	7	8	9	10
-	15	17	19	13	22	16	28	30	41	62

Heapify



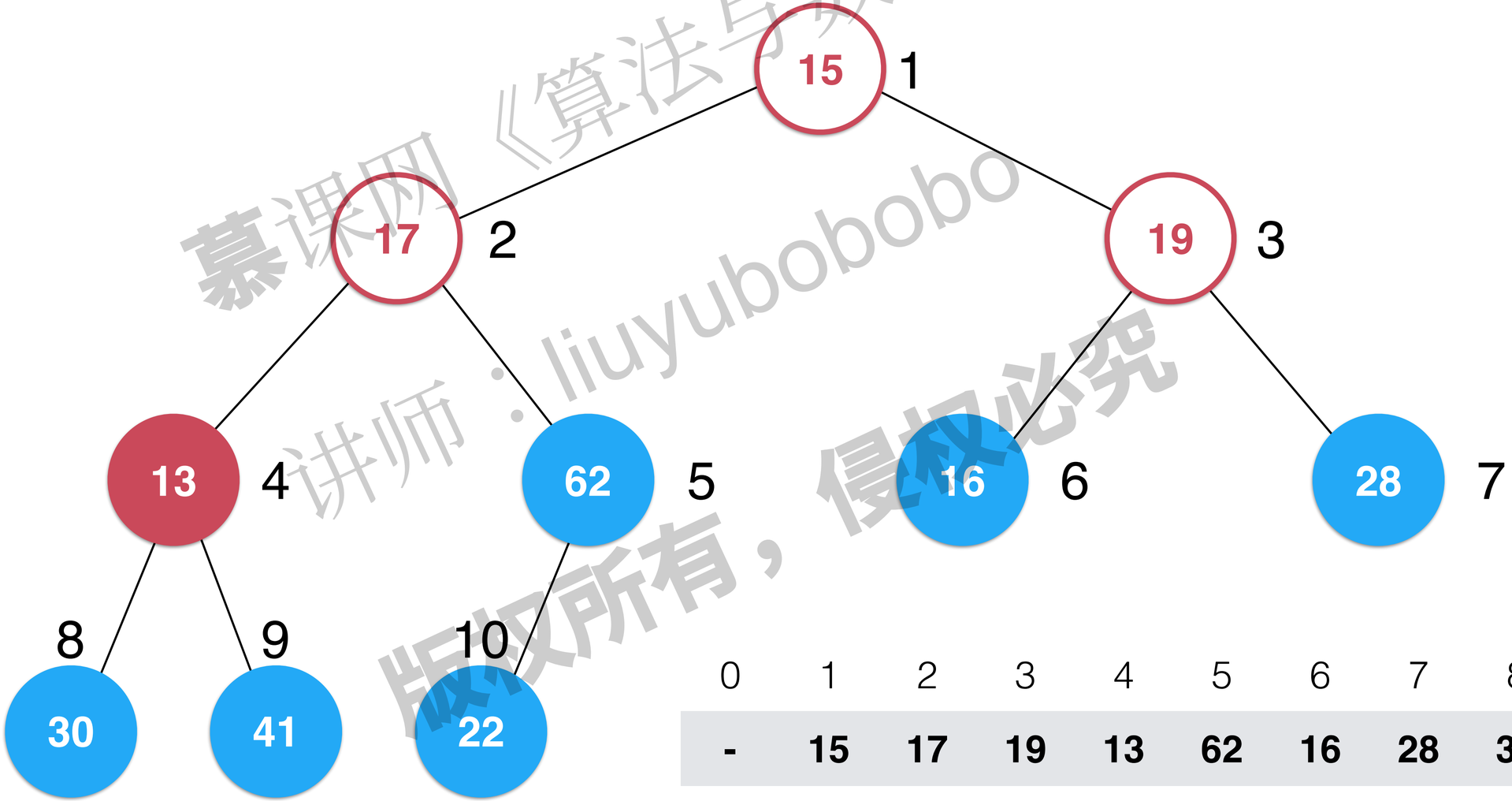
0	1	2	3	4	5	6	7	8	9	10
-	15	17	19	13	22	16	28	30	41	62

Heapify



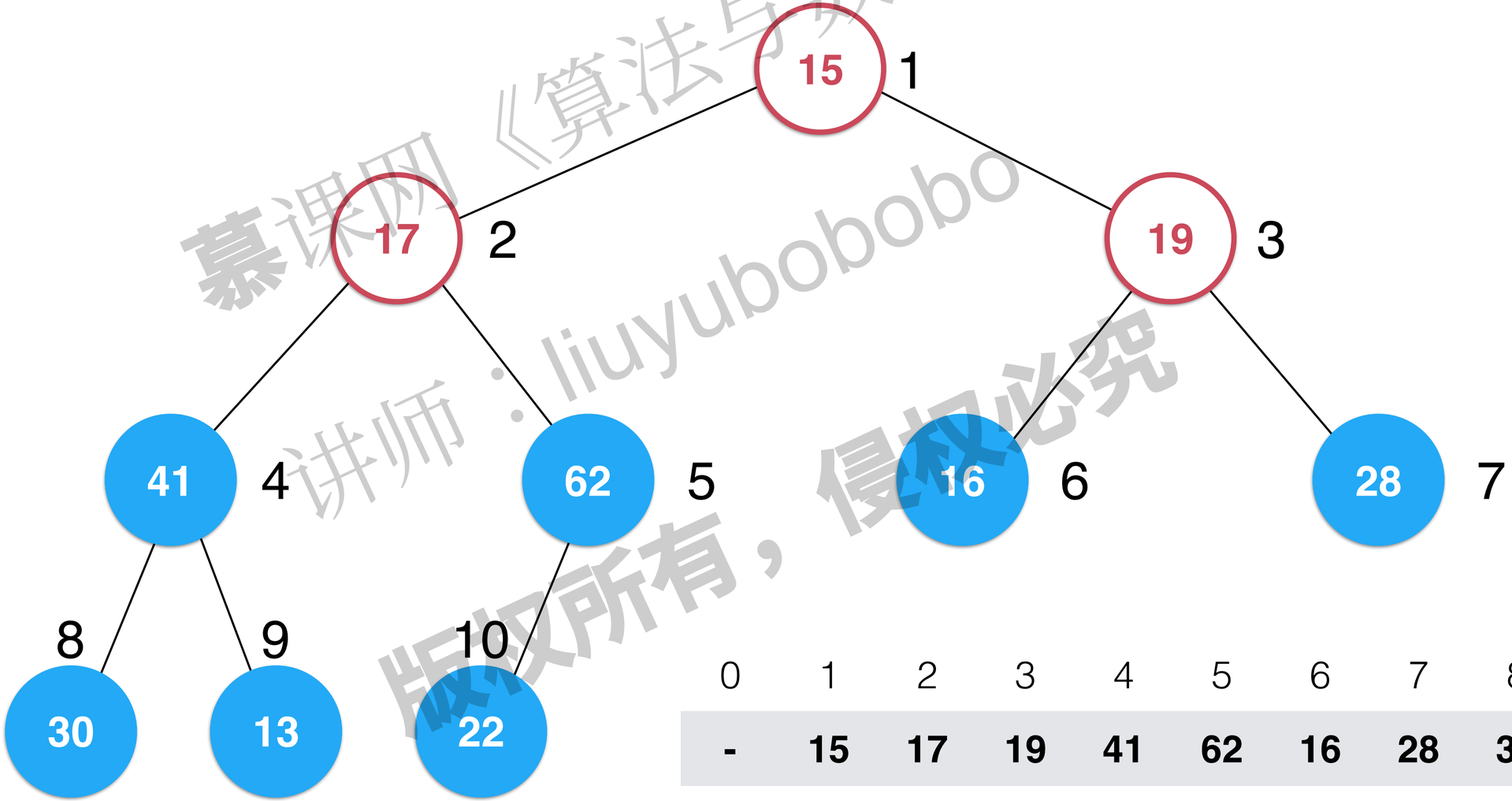
0	1	2	3	4	5	6	7	8	9	10
-	15	17	19	13	62	16	28	30	41	22

Heapify



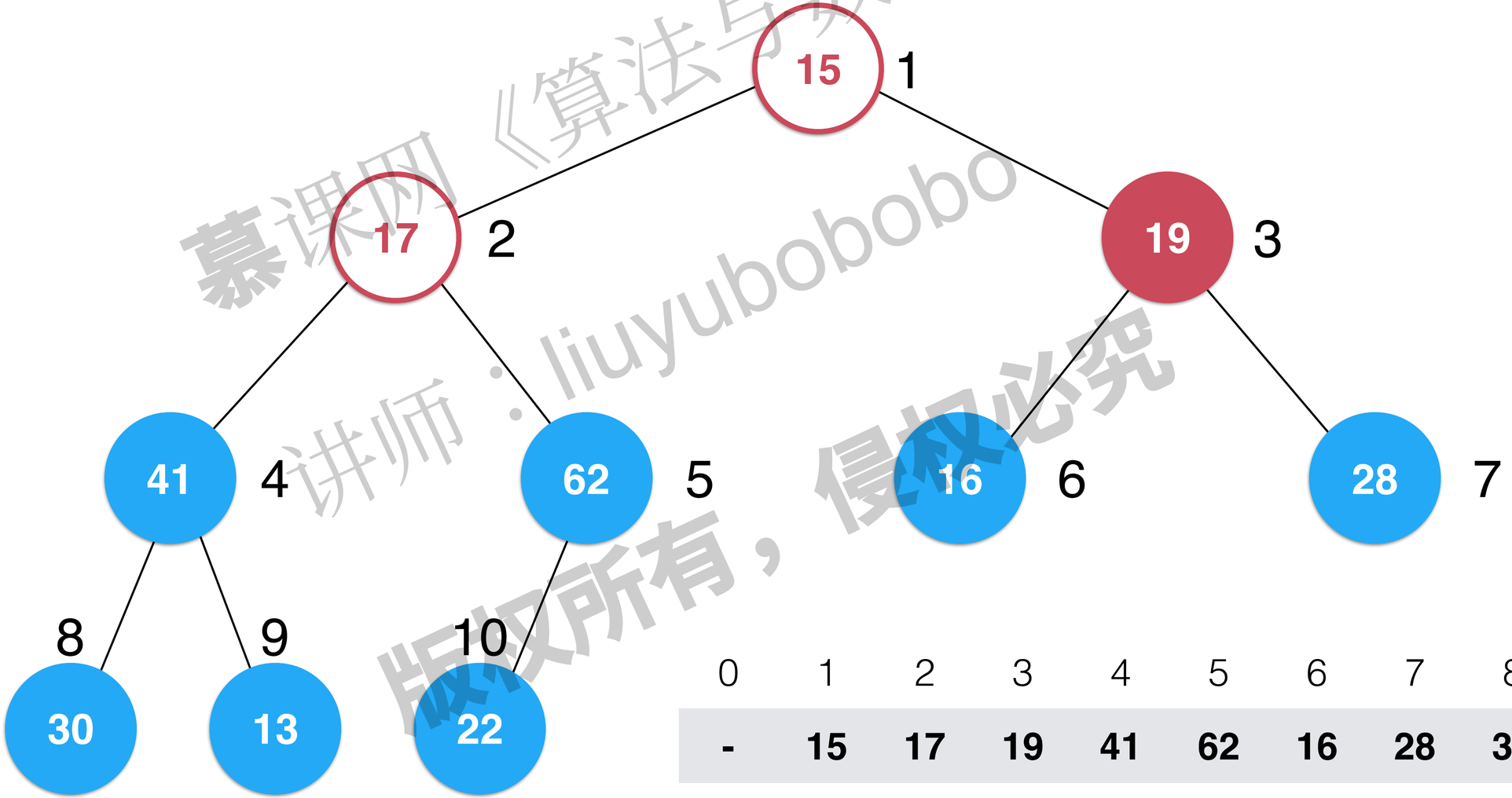
0	1	2	3	4	5	6	7	8	9	10
-	15	17	19	13	62	16	28	30	41	22

Heapify



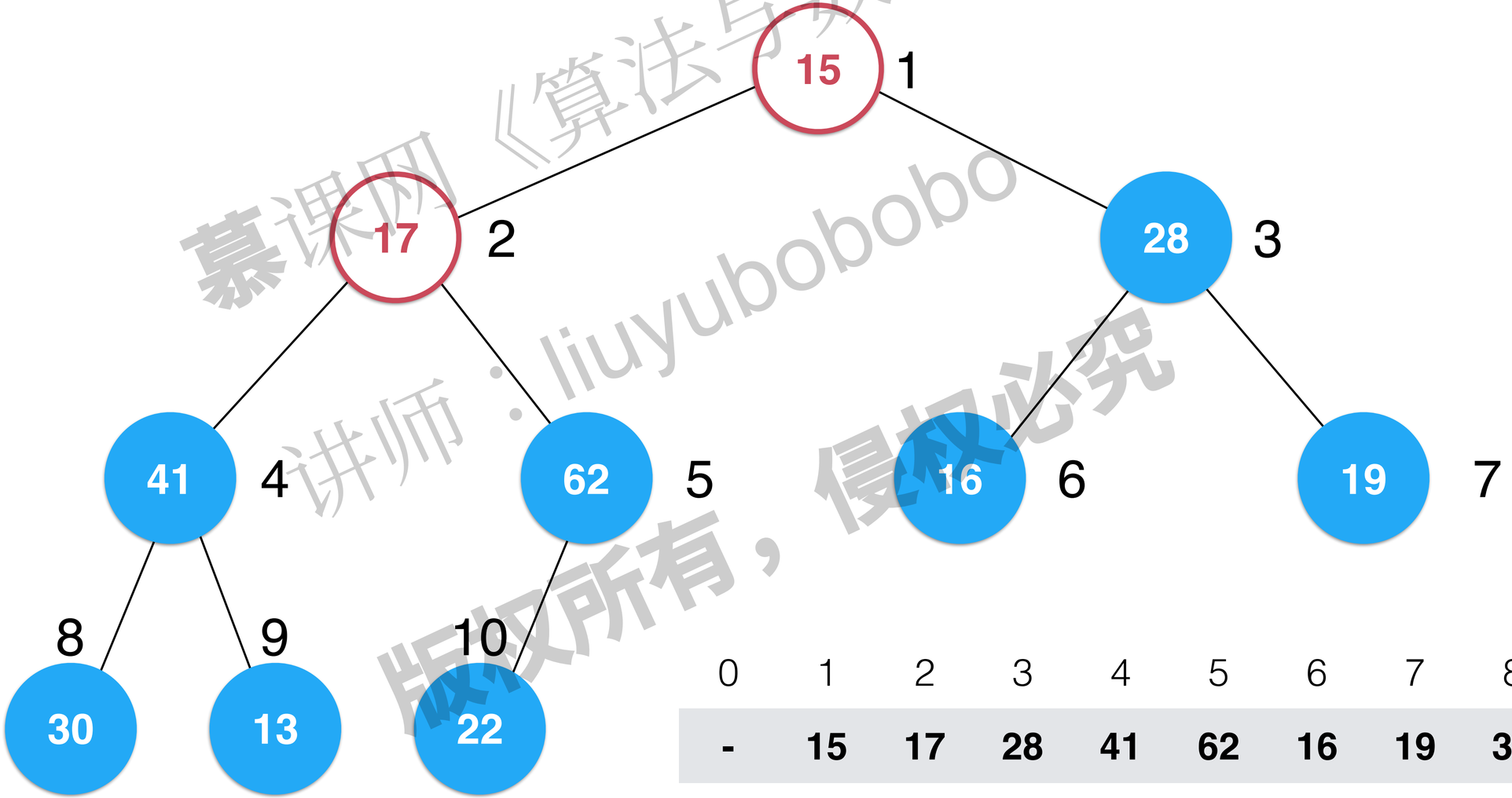
0	1	2	3	4	5	6	7	8	9	10
-	15	17	19	41	62	16	28	30	13	22

Heapify



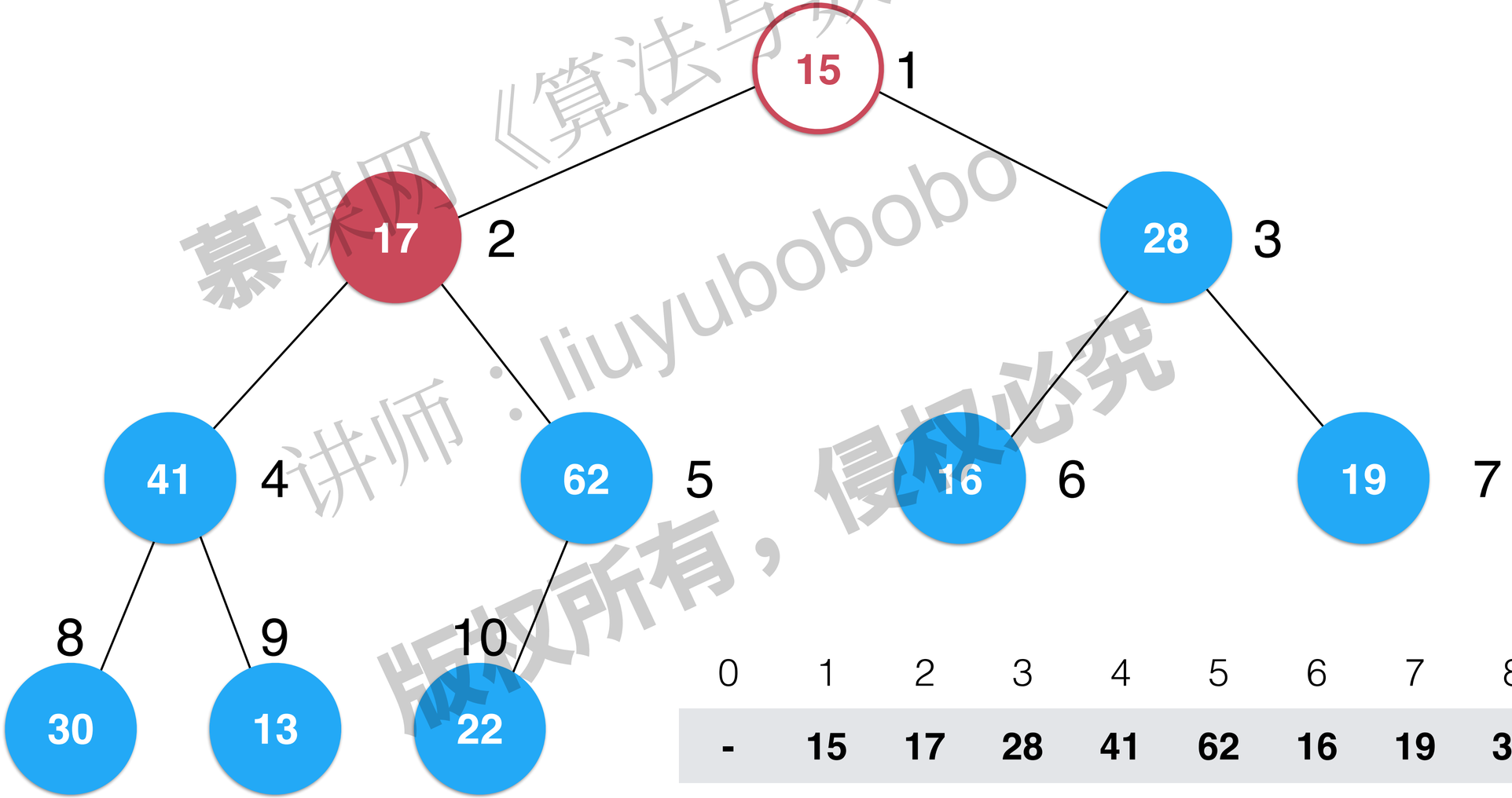
0	1	2	3	4	5	6	7	8	9	10
-	15	17	19	41	62	16	28	30	13	22

Heapify



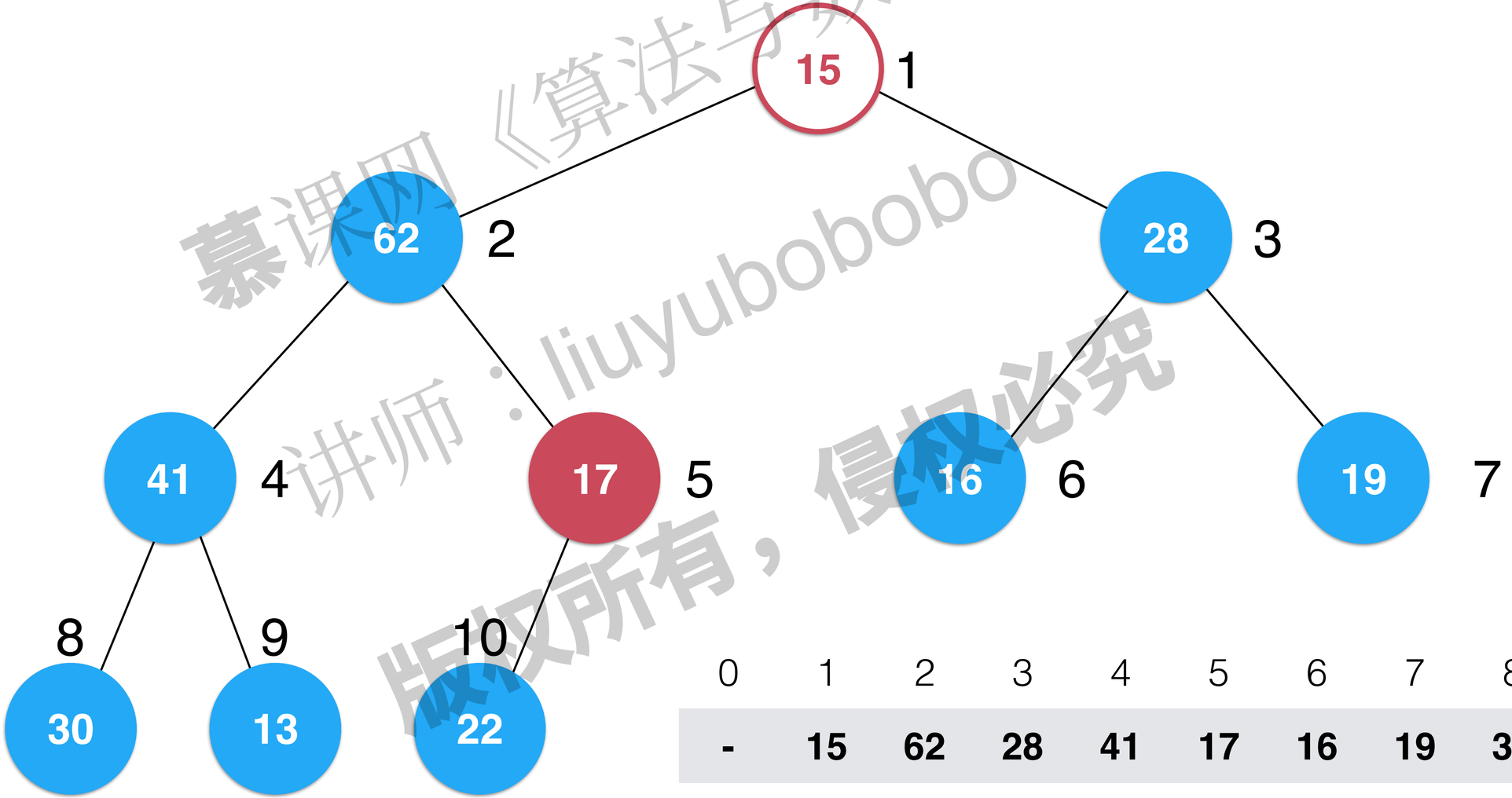
0	1	2	3	4	5	6	7	8	9	10
-	15	17	28	41	62	16	19	30	13	22

Heapify



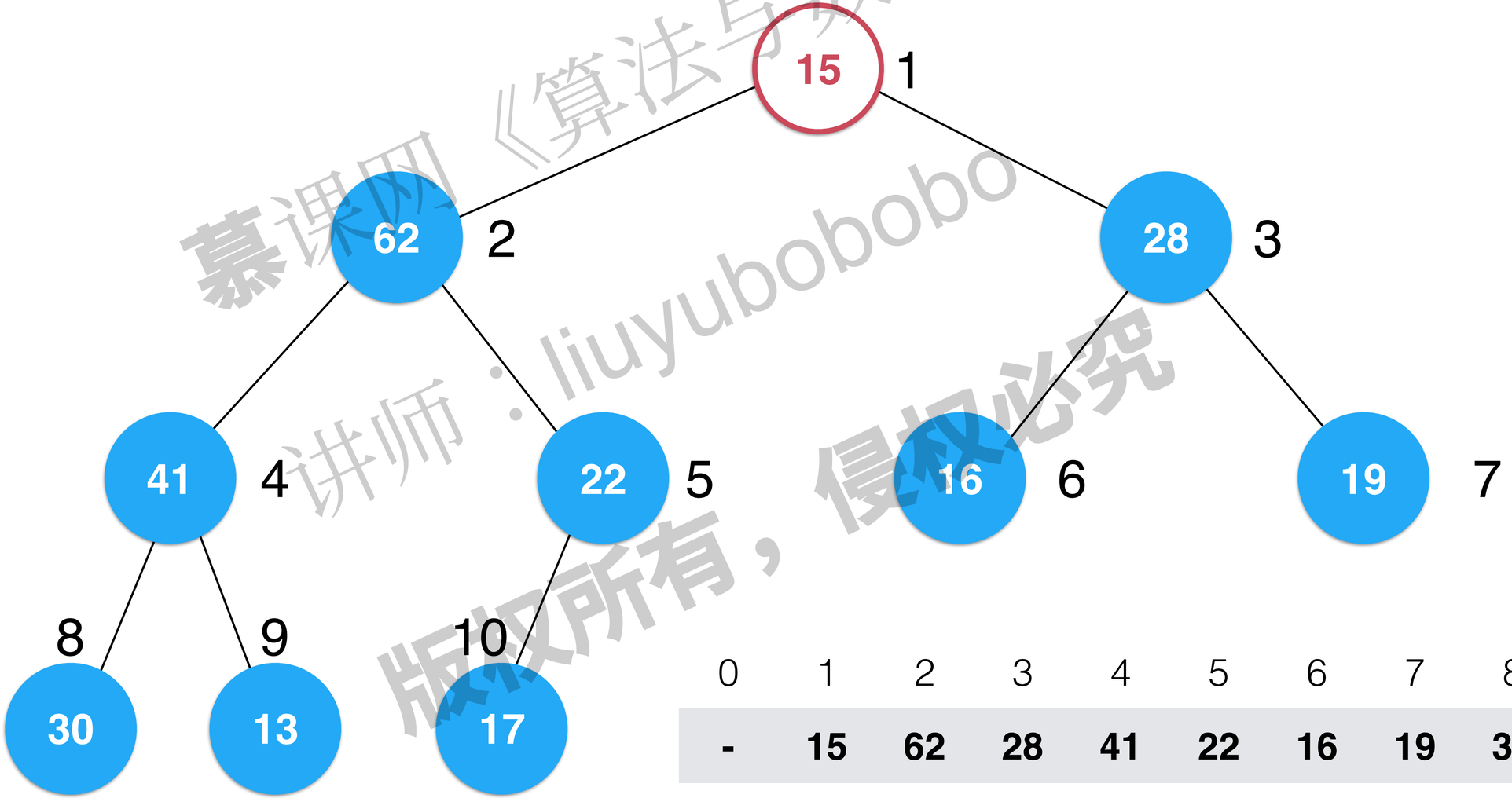
0	1	2	3	4	5	6	7	8	9	10
-	15	17	28	41	62	16	19	30	13	22

Heapify



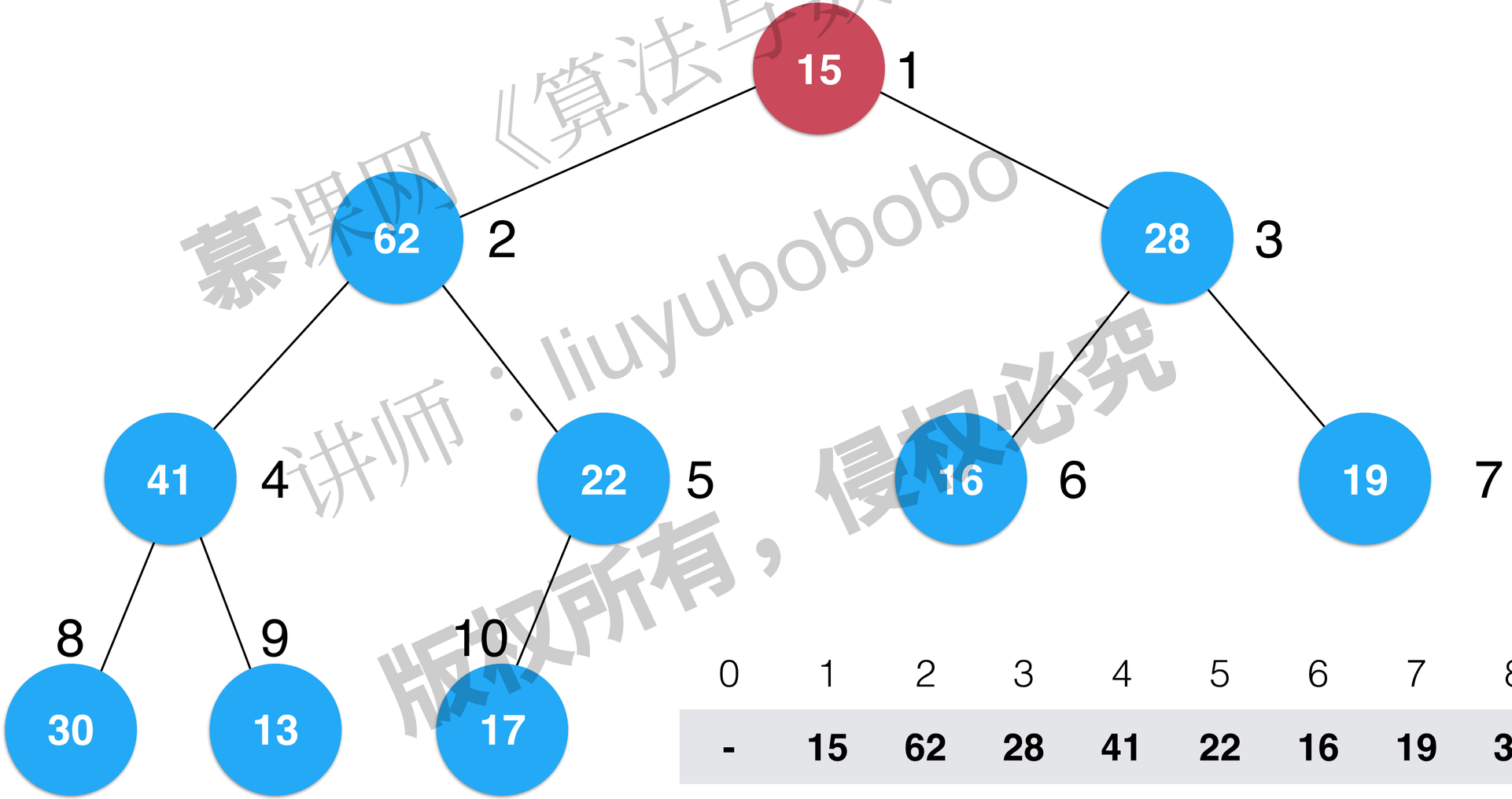
0	1	2	3	4	5	6	7	8	9	10
-	15	62	28	41	17	16	19	30	13	22

Heapify



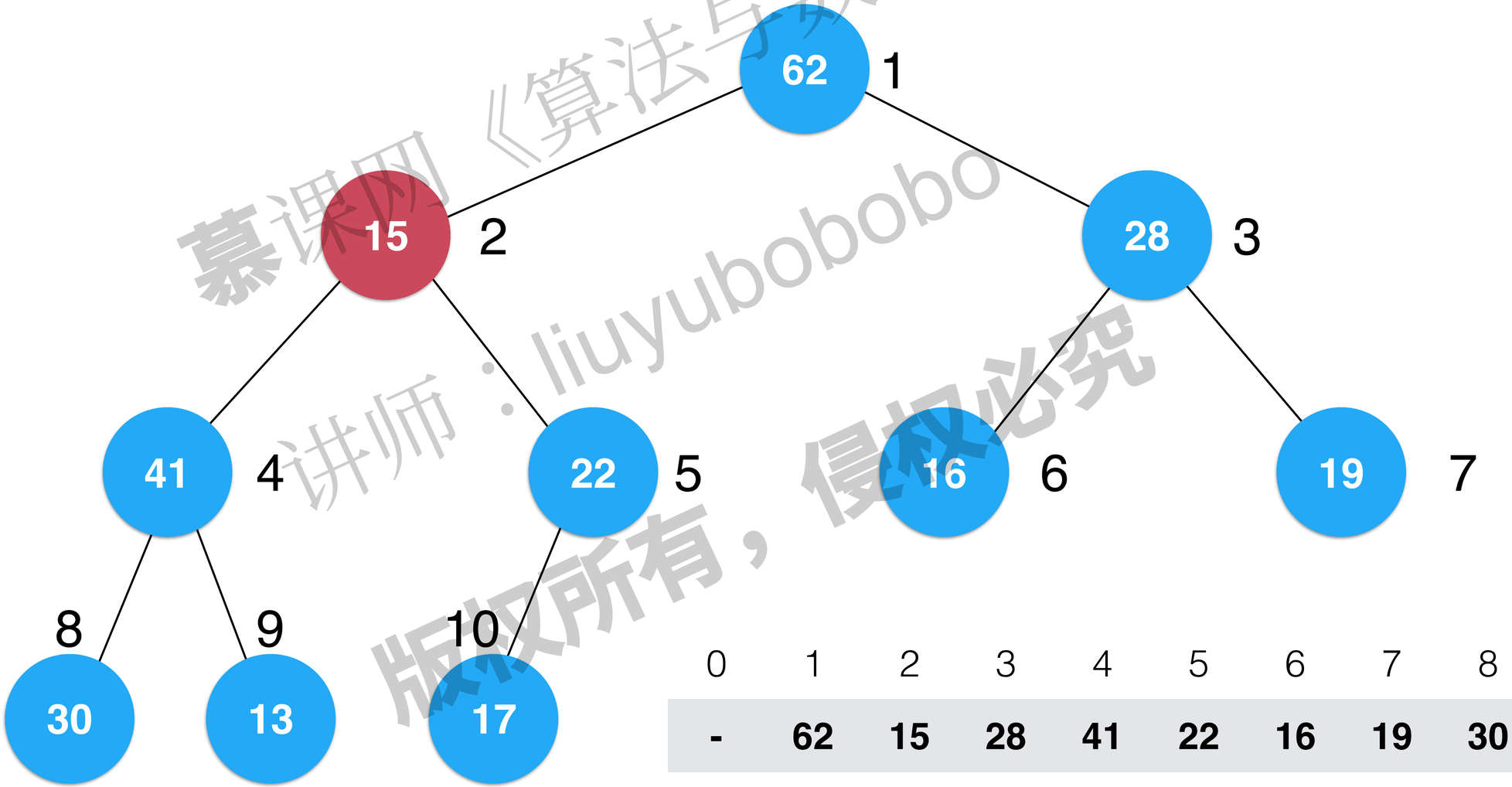
0	1	2	3	4	5	6	7	8	9	10
-	15	62	28	41	22	16	19	30	13	17

Heapify



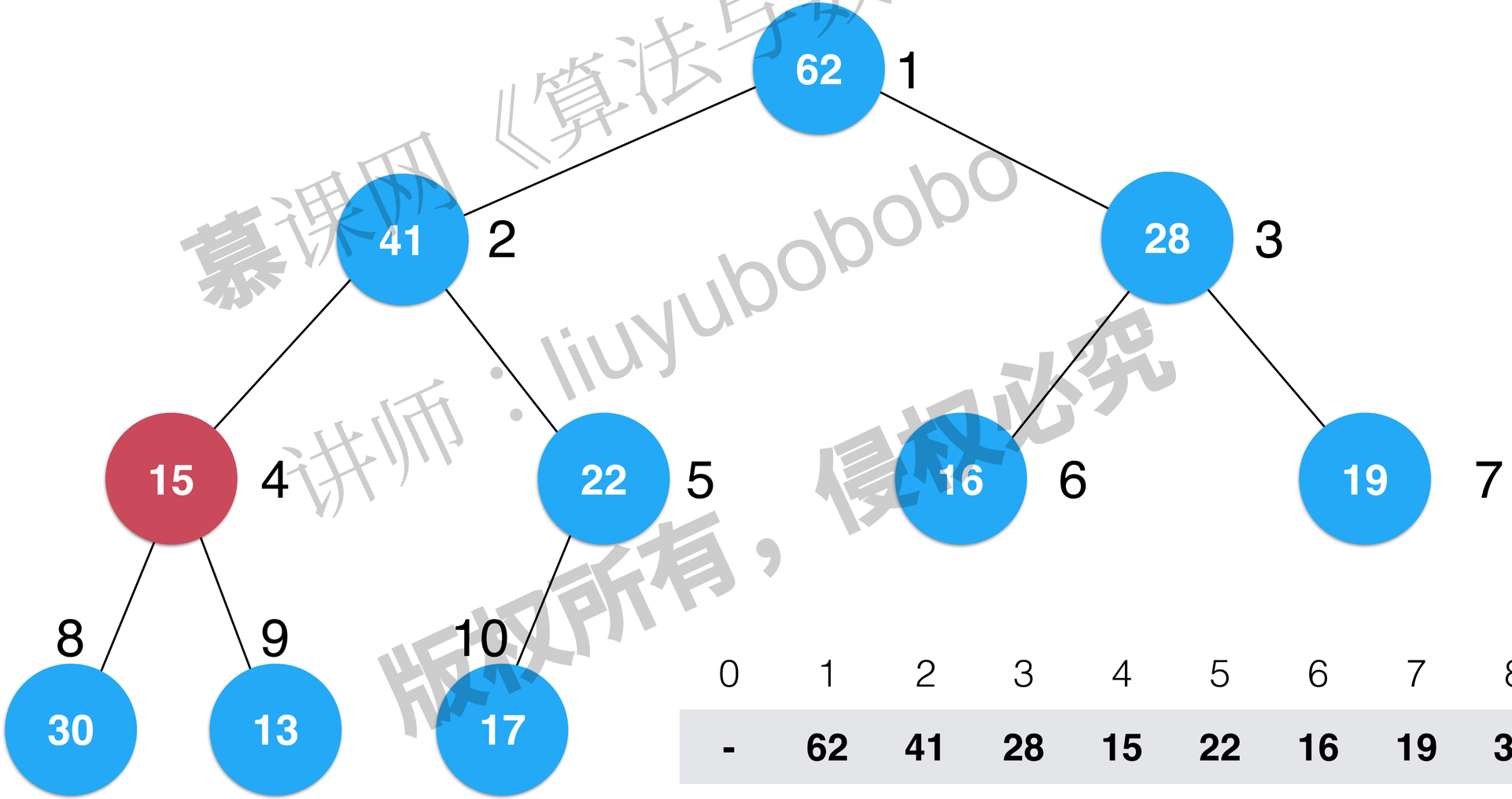
0	1	2	3	4	5	6	7	8	9	10
-	15	62	28	41	22	16	19	30	13	17

Heapify



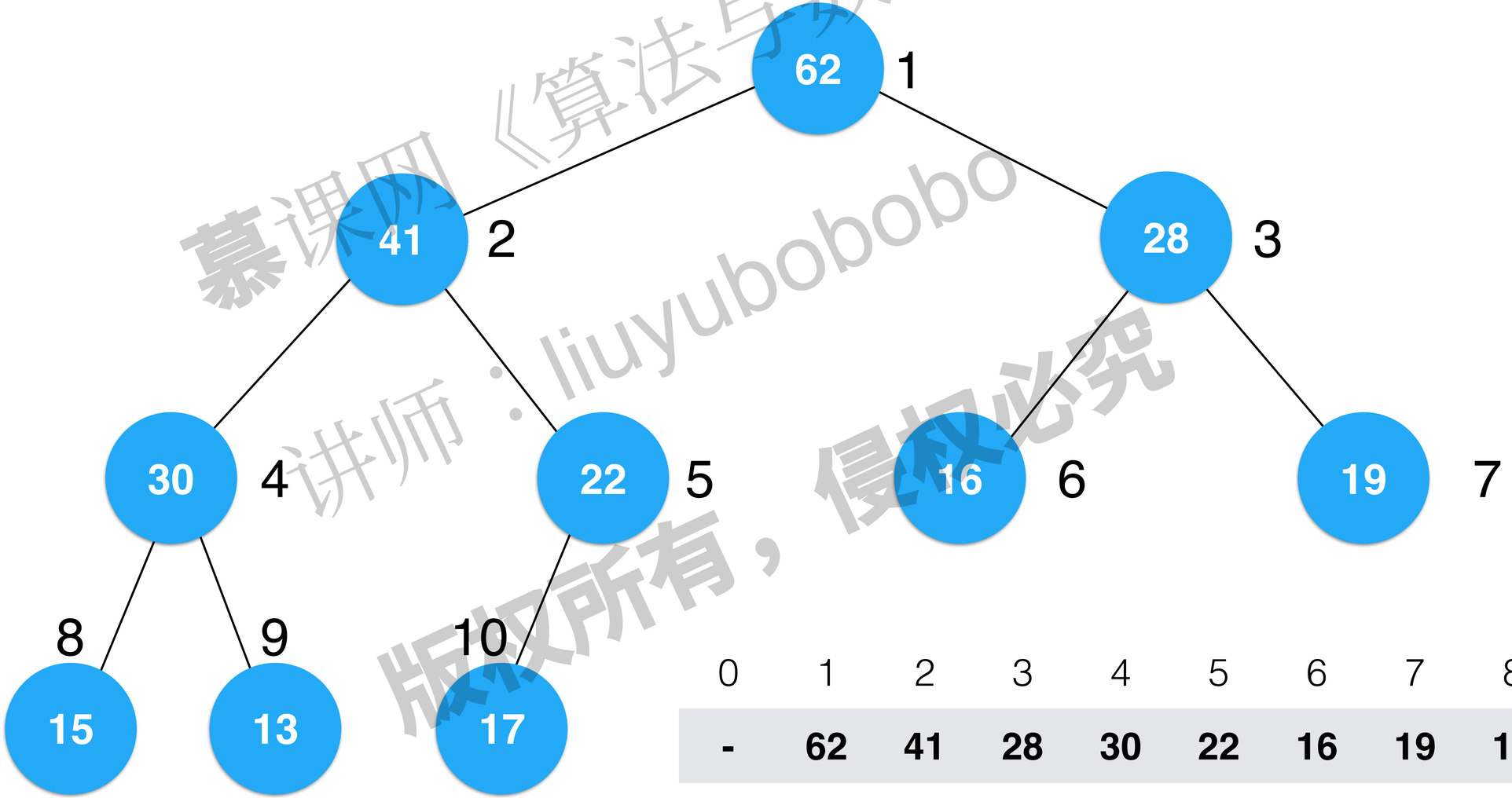
0	1	2	3	4	5	6	7	8	9	10
-	62	15	28	41	22	16	19	30	13	17

Heapify



0	1	2	3	4	5	6	7	8	9	10
-	62	41	28	15	22	16	19	30	13	17

Heapify



0	1	2	3	4	5	6	7	8	9	10
-	62	41	28	30	22	16	19	15	13	17

操作：Heapify

操作：使用Heapify的Heap Sort 和 Basic Heap Sort 的比较

Heapify 的算法复杂度

将n个元素逐个插入到一个空堆中，算法复杂度是O(nlogn)

heapify的过程，算法复杂度为O(n)

原地堆排序

原地堆排序

原地堆排序

原地堆排序

原地堆排序

原地堆排序

原地堆排序

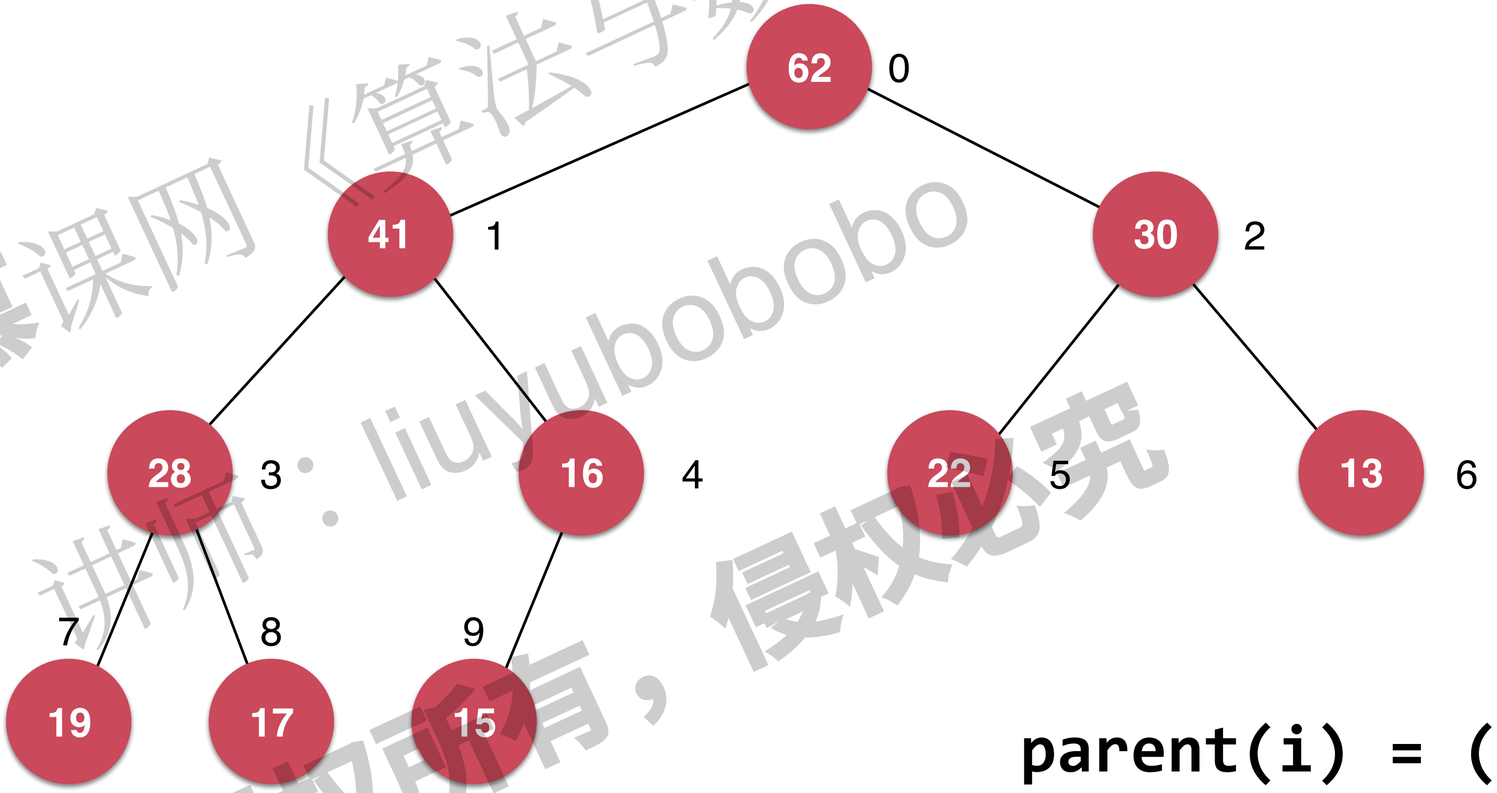
原地堆排序

原地堆排序

原地堆排序

Max Heap

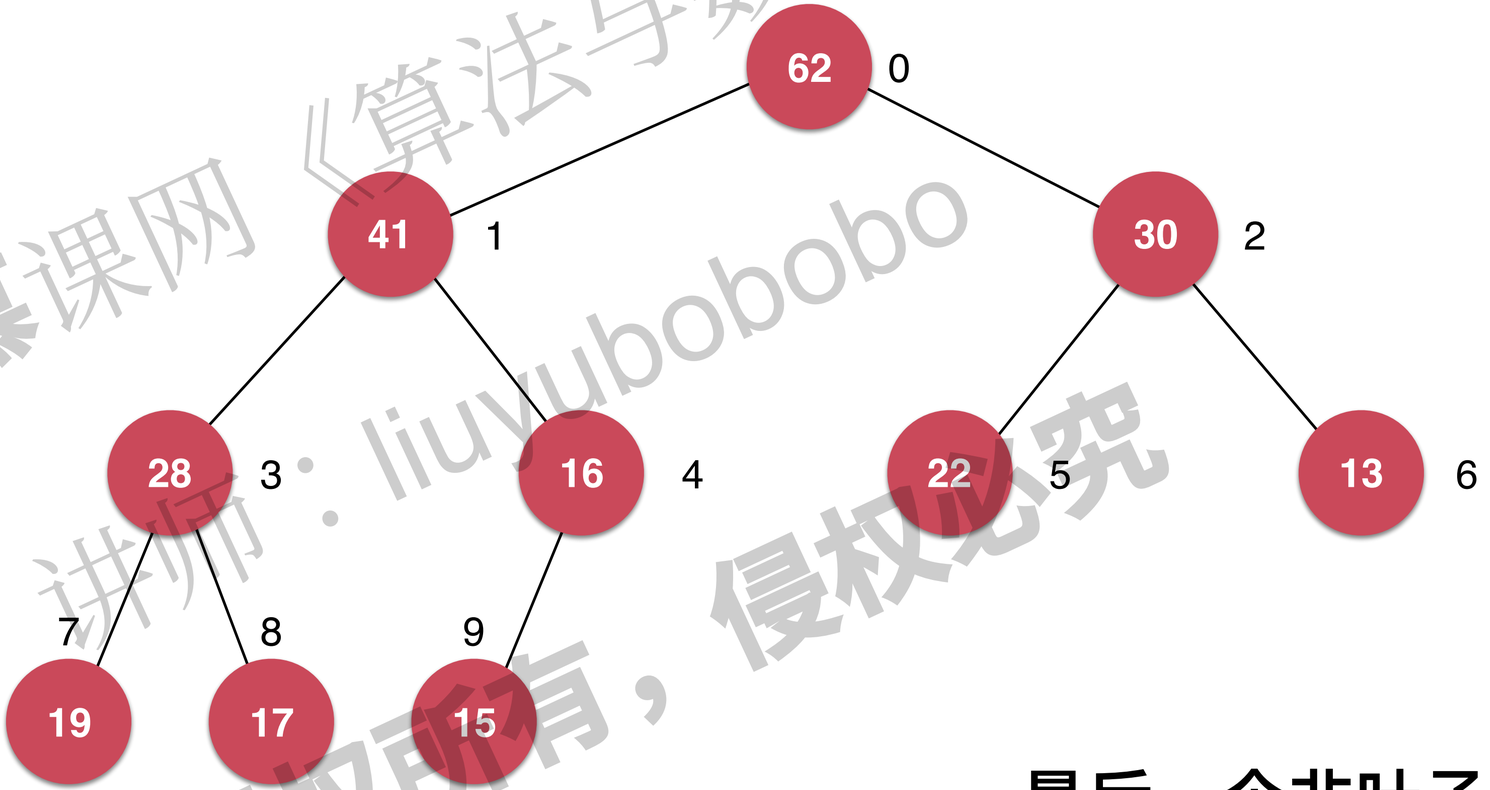
用数组存储二叉堆



0	1	2	3	4	5	6	7	8	9
62	41	30	28	16	22	13	19	17	15

```
parent(i) = (i-1)/2

left child  (i) = 2*i + 1
right child (i) = 2*i + 2
```

用数组存储二叉堆



0	1	2	3	4	5	6	7	8	9
62	41	30	28	16	22	13	19	17	15

最后一个非叶子节点的索引

(count-2)/2

操作：原地堆排序

排序算法总结

排序算法总结

	平均时间复杂度
插入排序 Insertion Sort	O(n^2)
归并排序 Merge Sort	O(nlogn)
快速排序 Quick Sort	O(nlogn)
堆排序 Heap Sort	O(nlogn)

排序算法总结

	平均时间复杂度	原地排序
插入排序 Insertion Sort	O(n^2)	✓
归并排序 Merge Sort	O(nlogn)	×
快速排序 Quick Sort	O(nlogn)	✓
堆排序 Heap Sort	O(nlogn)	✓

排序算法总结

	平均时间复杂度	原地排序	额外空间
插入排序 Insertion Sort	O(n^2)	✓	O(1)
归并排序 Merge Sort	O(nlogn)	×	O(n)
快速排序 Quick Sort	O(nlogn)	✓	O(logn)
堆排序 Heap Sort	O(nlogn)	✓	O(1)

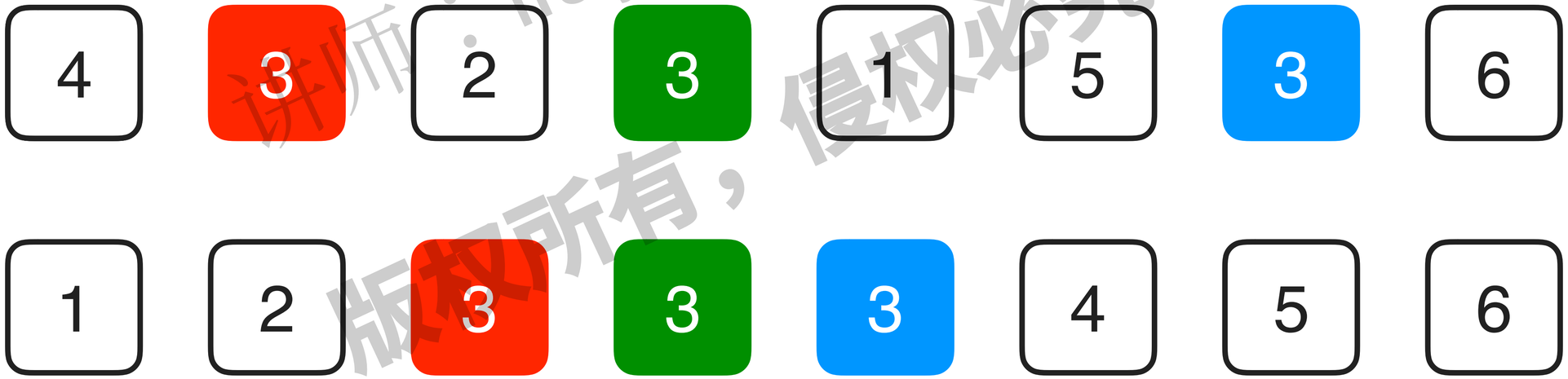
排序算法总结

	平均时间复杂度	原地排序	额外空间	稳定排序
插入排序 Insertion Sort	O(n^2)	✓	O(1)	✓
归并排序 Merge Sort	O(nlogn)	×	O(n)	✓
快速排序 Quick Sort	O(nlogn)	✓	O(logn)	×
堆排序 Heap Sort	O(nlogn)	✓	O(1)	×

排序算法的稳定性 Stable

稳定排序：对于相等的元素，在排序后，原来靠前的元素依然靠前。

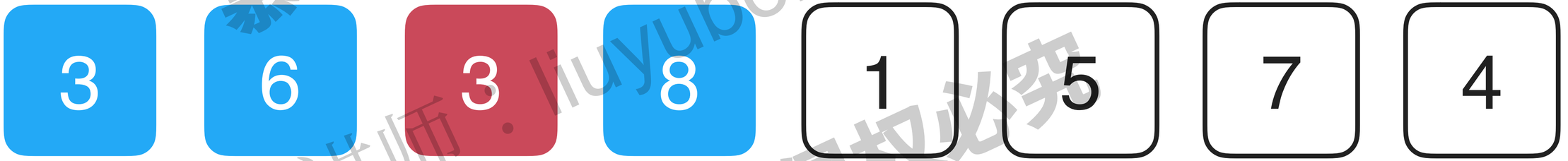
相等元素的相对位置没有发生改变。



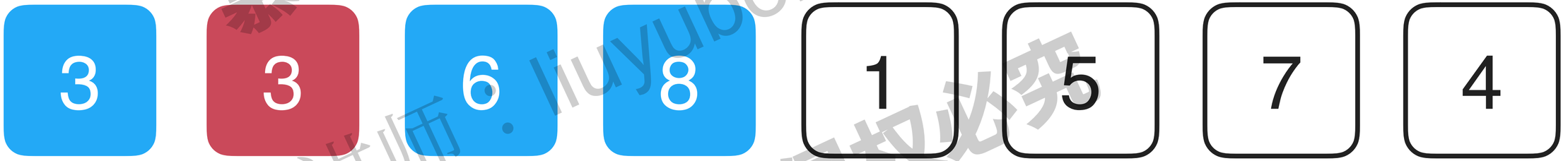
插入排序 Insertion Sort

插入排序 Insertion Sort

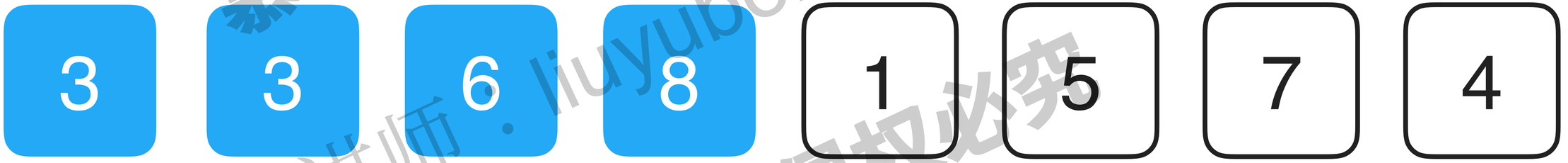
插入排序 Insertion Sort



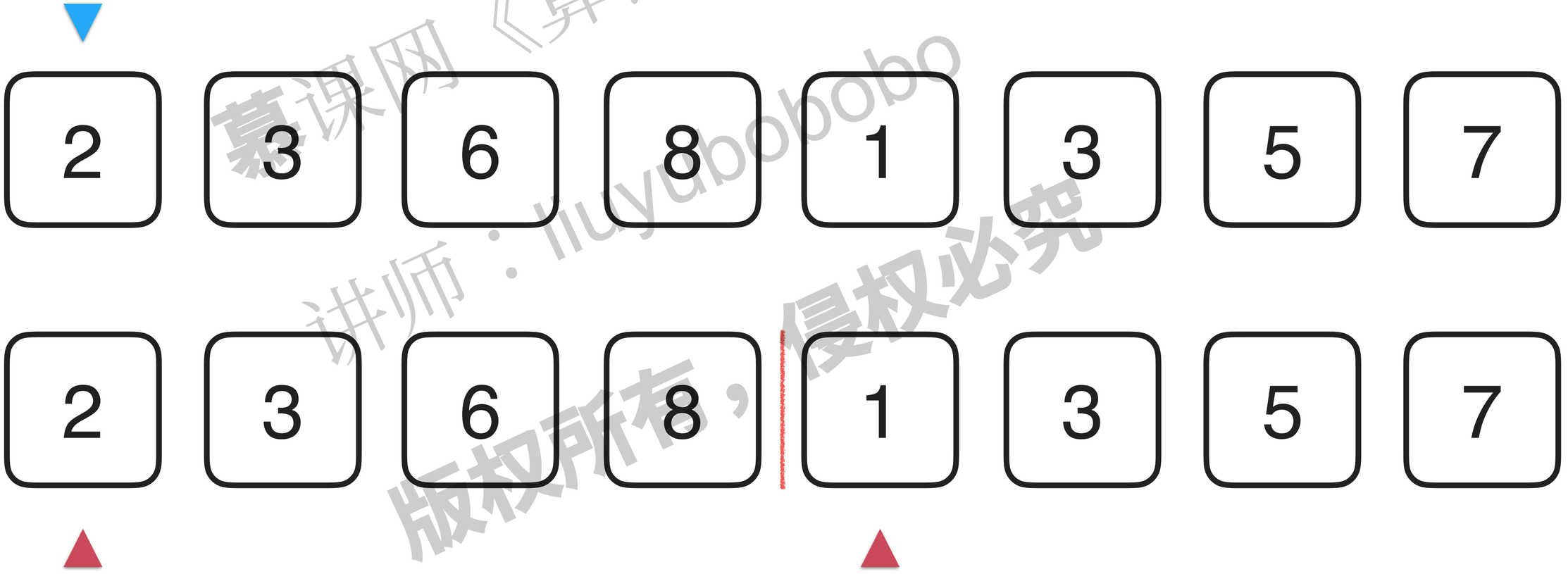
插入排序 Insertion Sort



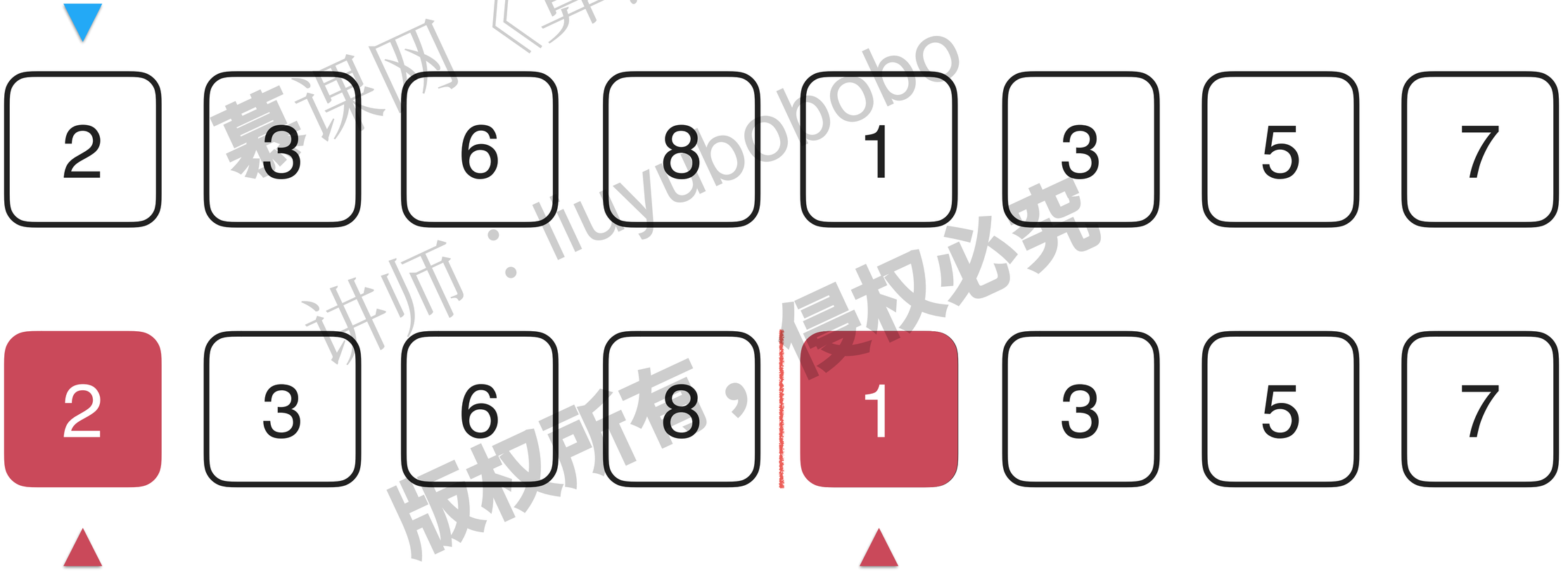
插入排序 Insertion Sort



归并过程 Merge



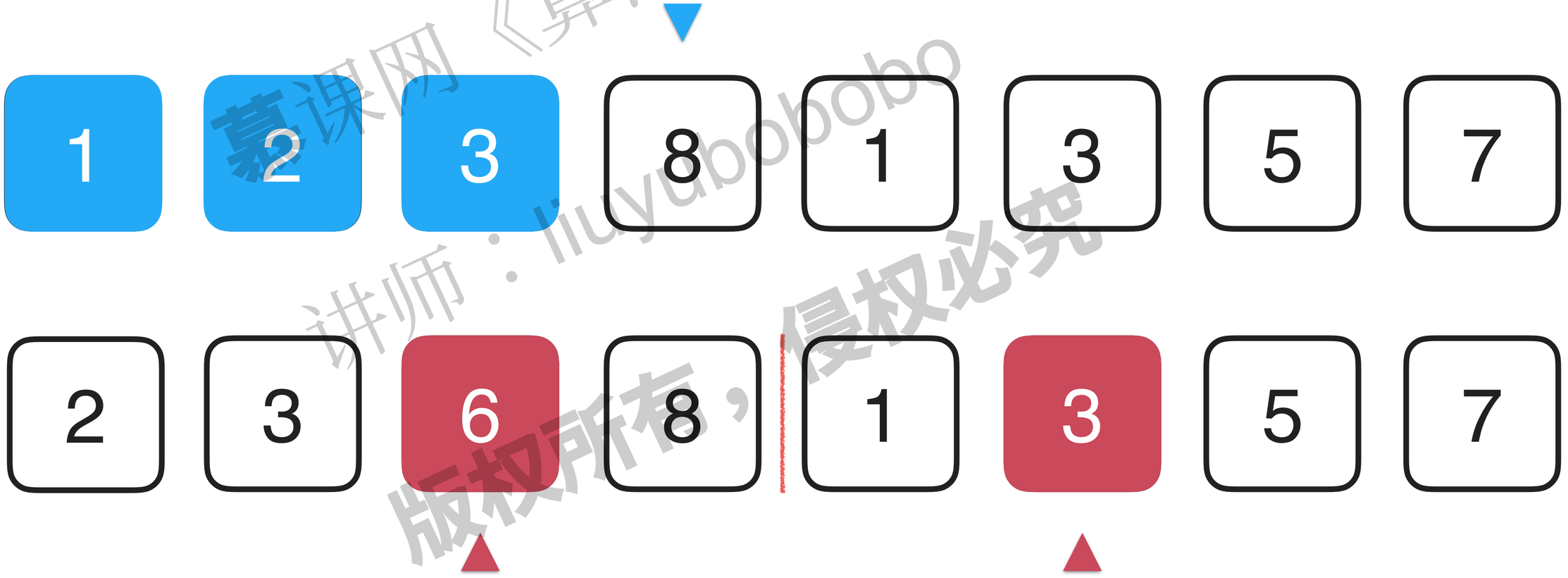
归并过程 Merge



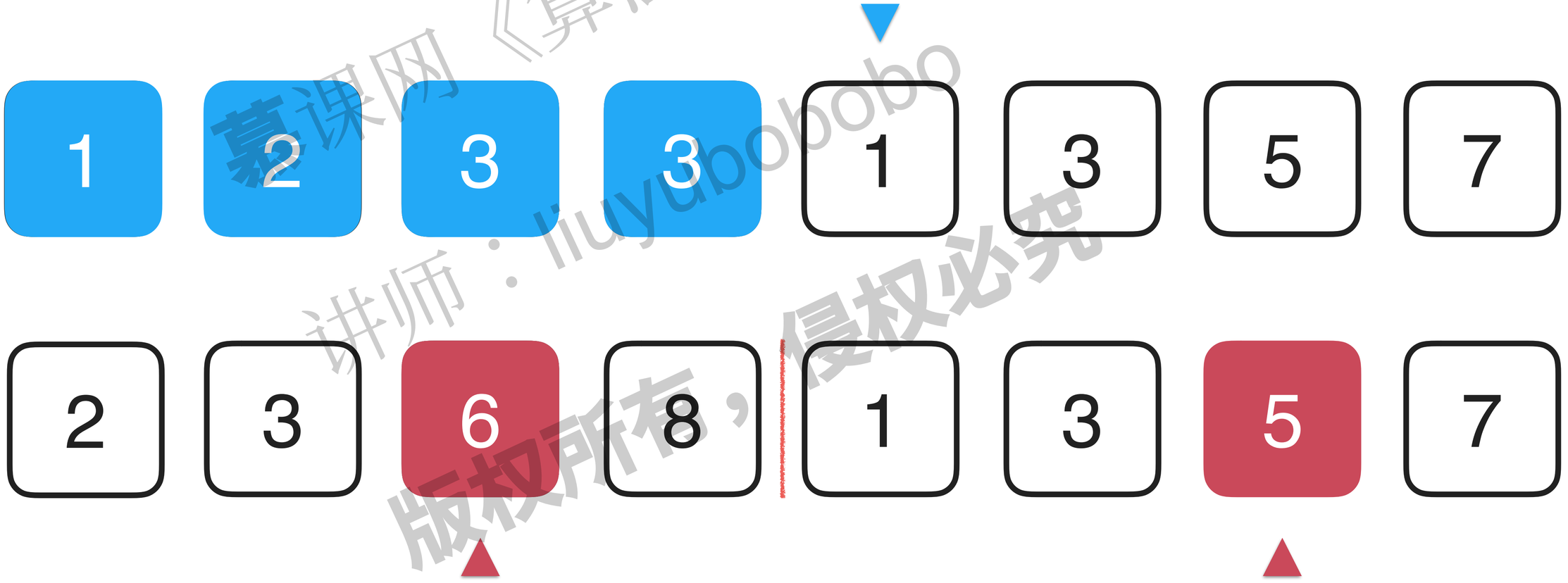
归并过程 Merge

归并过程 Merge

归并过程 Merge



归并过程 Merge



排序算法的稳定性 Stable

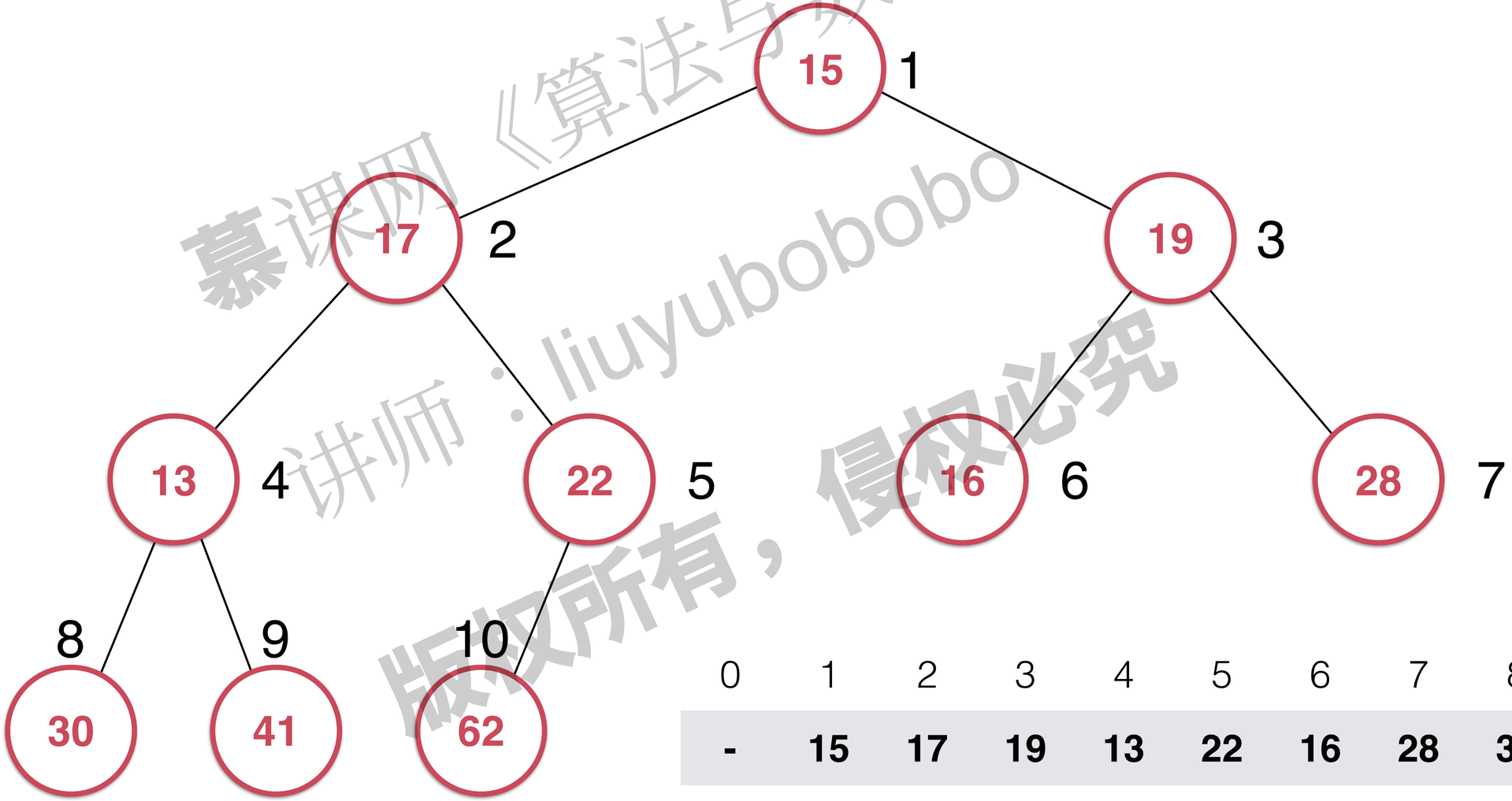
可以通过自定义比较函数，让排序算法不存在稳定性的问题。

```
bool operator<(const Student& otherStudent){

    return score != otherStudent.score ?
                score > otherStudent.score :
                 name < otherStudent.name;
}
```

	平均时间复杂度	原地排序	额外空间	稳定排序
插入排序 Insertion Sort	O(n^2)	✓	O(1)	✓
归并排序 Merge Sort	O(nlogn)	×	O(n)	✓
快速排序 Quick Sort	O(nlogn)	✓	O(logn)	×
堆排序 Heap Sort	O(nlogn)	✓	O(1)	×
神秘的排序算法?	O(nlogn)	✓	O(1)	✓

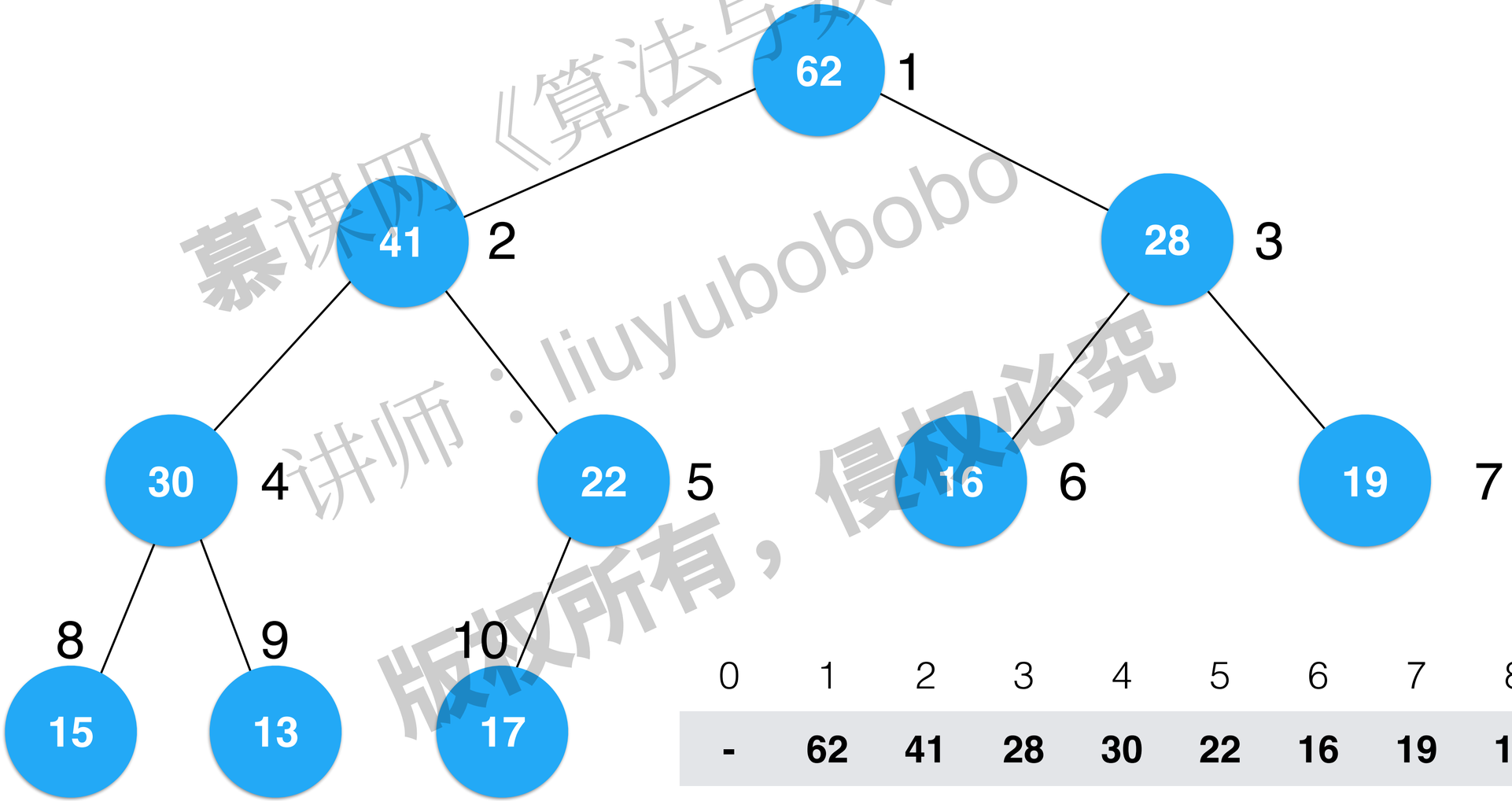
索引堆 Index Heap

Heap



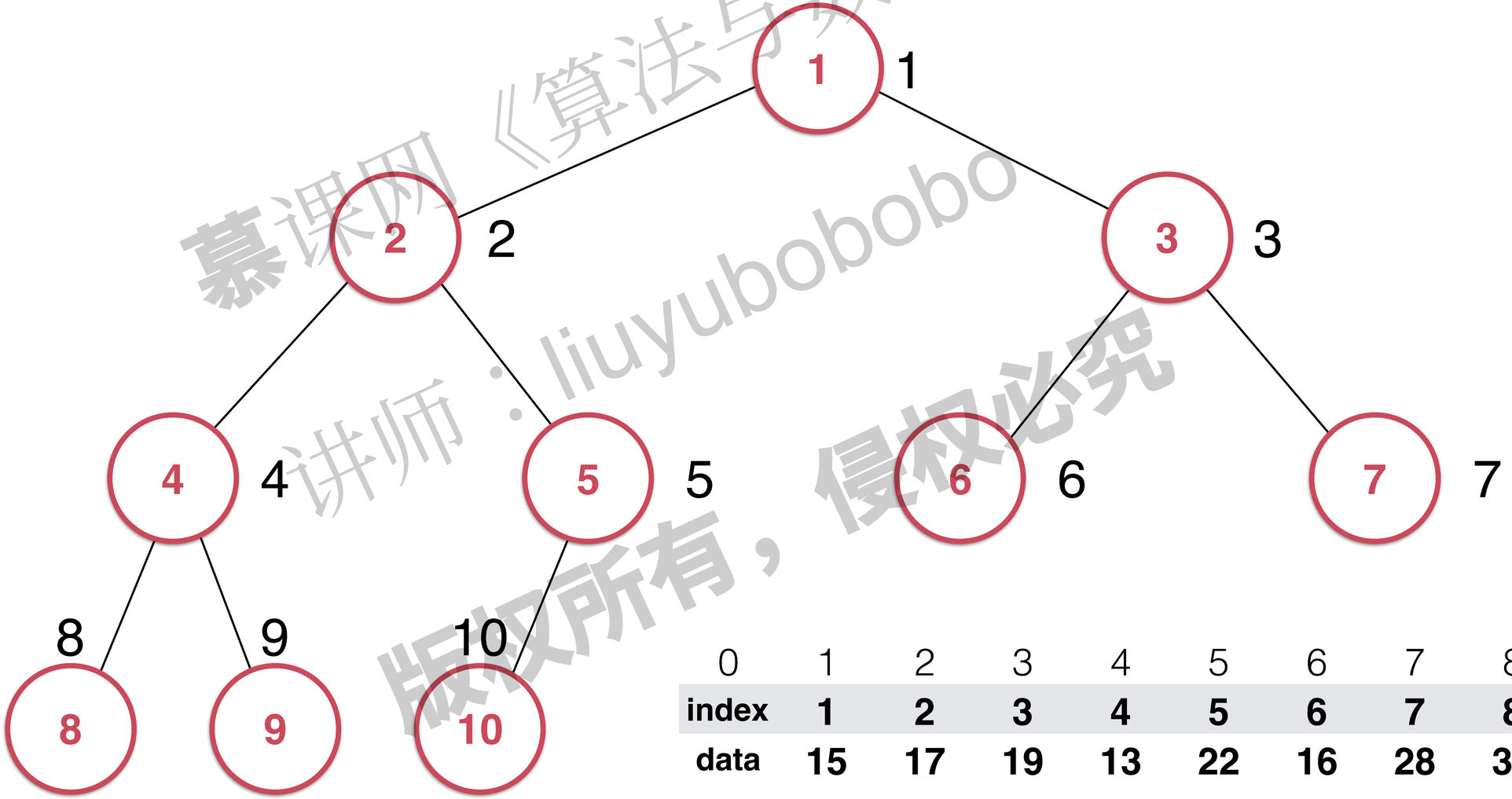
0	1	2	3	4	5	6	7	8	9	10
-	15	17	19	13	22	16	28	30	41	62

Heap



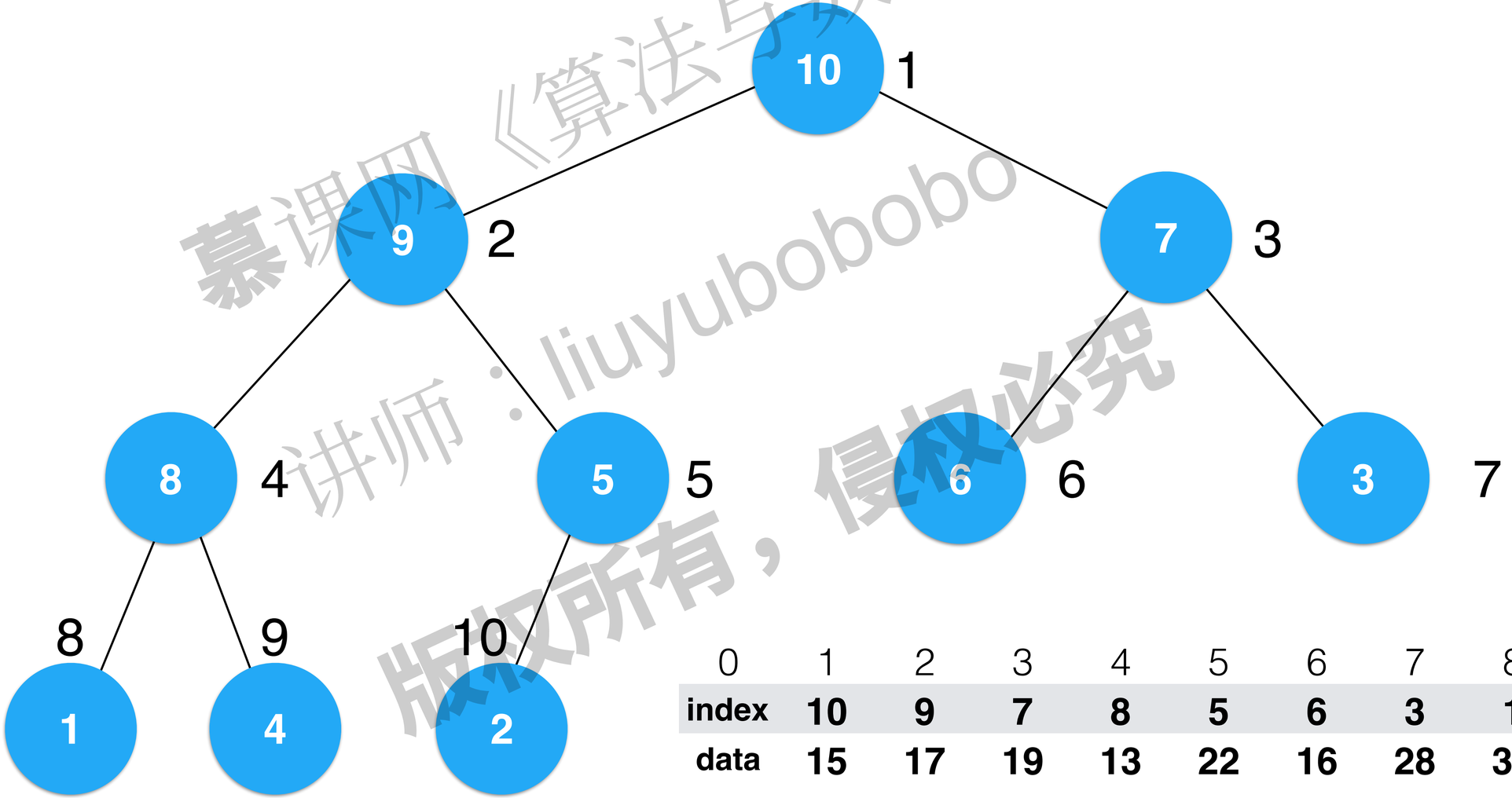
0	1	2	3	4	5	6	7	8	9	10
-	62	41	28	30	22	16	19	15	13	17

Index Max Heap



0	1	2	3	4	5	6	7	8	9	10
index	1	2	3	4	5	6	7	8	9	10
data	15	17	19	13	22	16	28	30	41	62

Index Max Heap

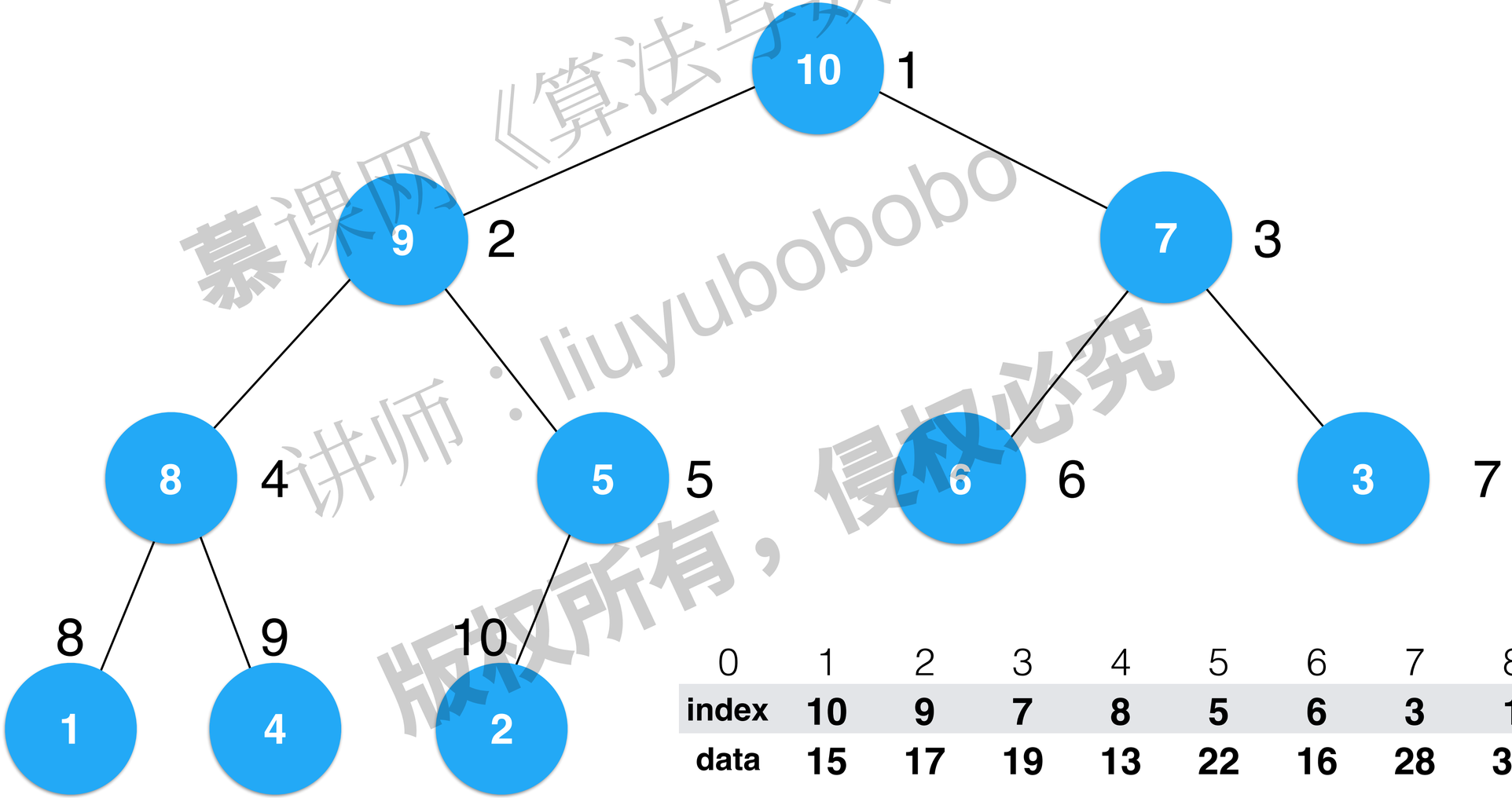


0	1	2	3	4	5	6	7	8	9	10
index	10	9	7	8	5	6	3	1	4	2
data	15	17	19	13	22	16	28	30	41	62

操作：编写基础 Index Max Heap

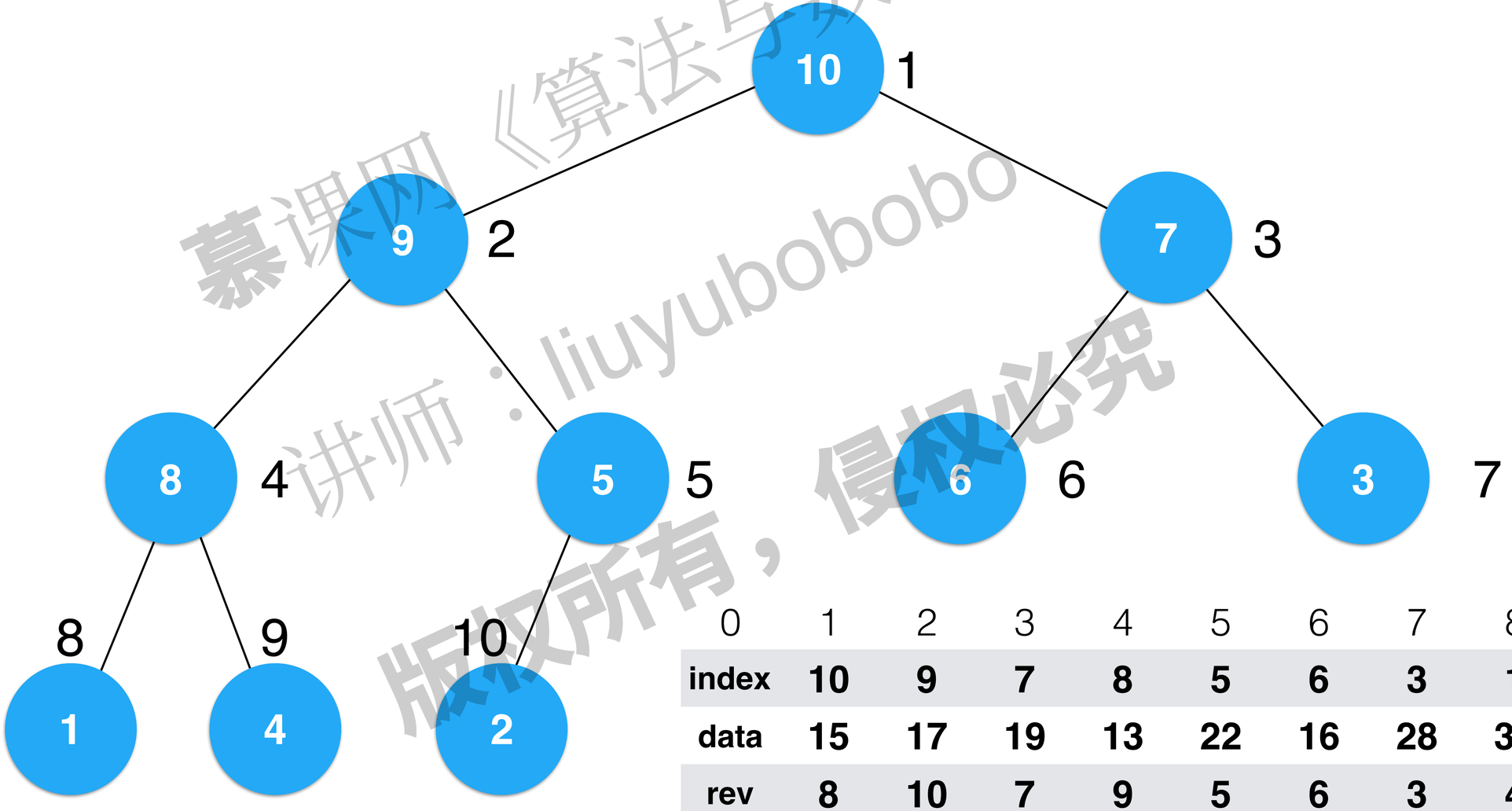
使用reverse数组反向查找

Index Max Heap



0	1	2	3	4	5	6	7	8	9	10
index	10	9	7	8	5	6	3	1	4	2
data	15	17	19	13	22	16	28	30	41	62

Index Max Heap



0	1	2	3	4	5	6	7	8	9	10
index	10	9	7	8	5	6	3	1	4	2
data	15	17	19	13	22	16	28	30	41	62
rev	8	10	7	9	5	6	3	4	2	1

Index Max Heap

0	1	2	3	4	5	6	7	8	9	10
index	10	9	7	8	5	6	3	1	4	2
data	15	17	19	13	22	16	28	30	41	62
rev	8	10	7	9	5	6	3	4	2	1

reverse[i] 表示索引i在indexes(堆)中的位置

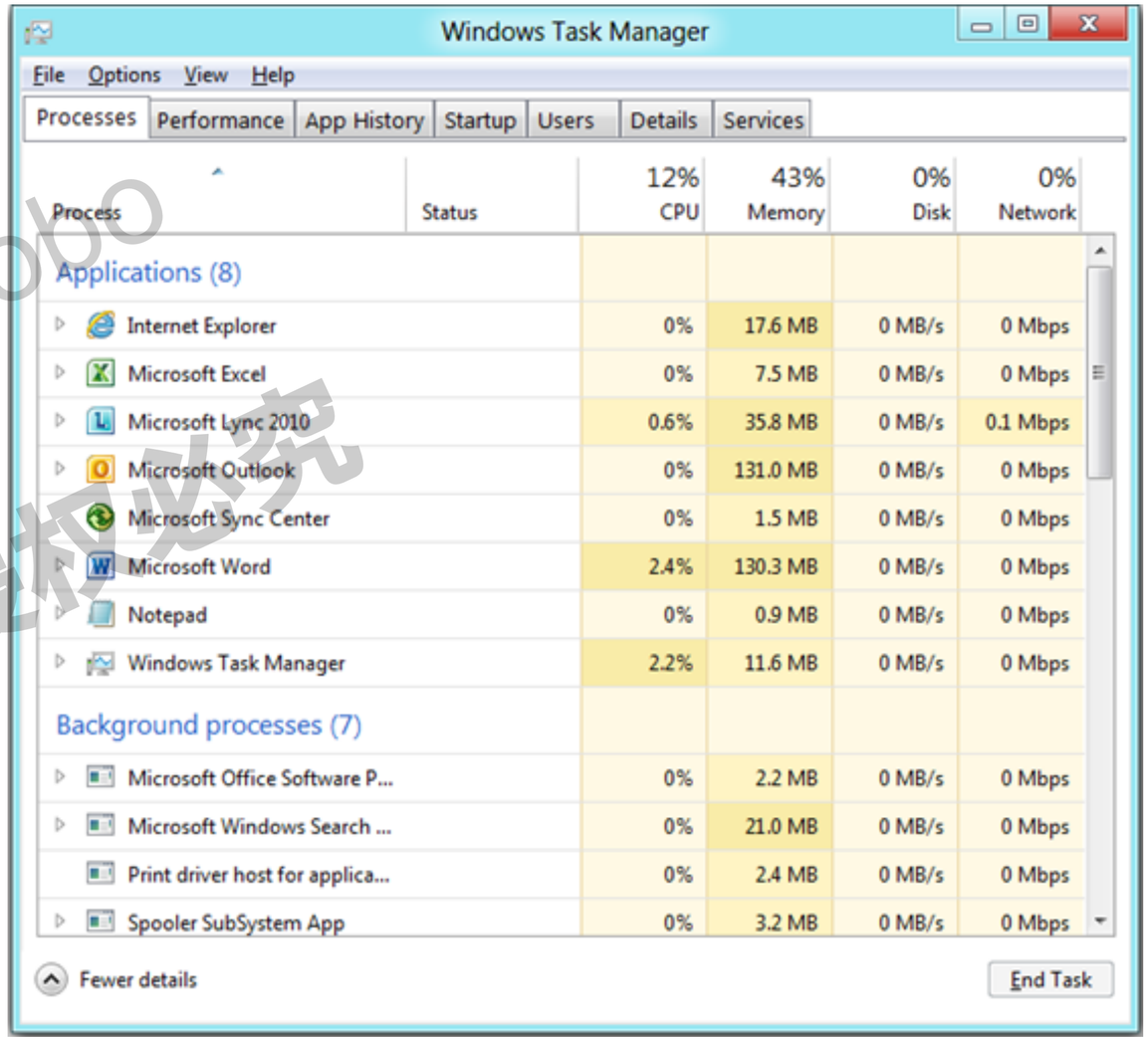
```
indexes[i] = j
reverse[j] = i

indexes[reverse[i]] = i
reverse[indexes[i]] = i
```

操作：编写拥有reverse反向查找的Index Max Heap

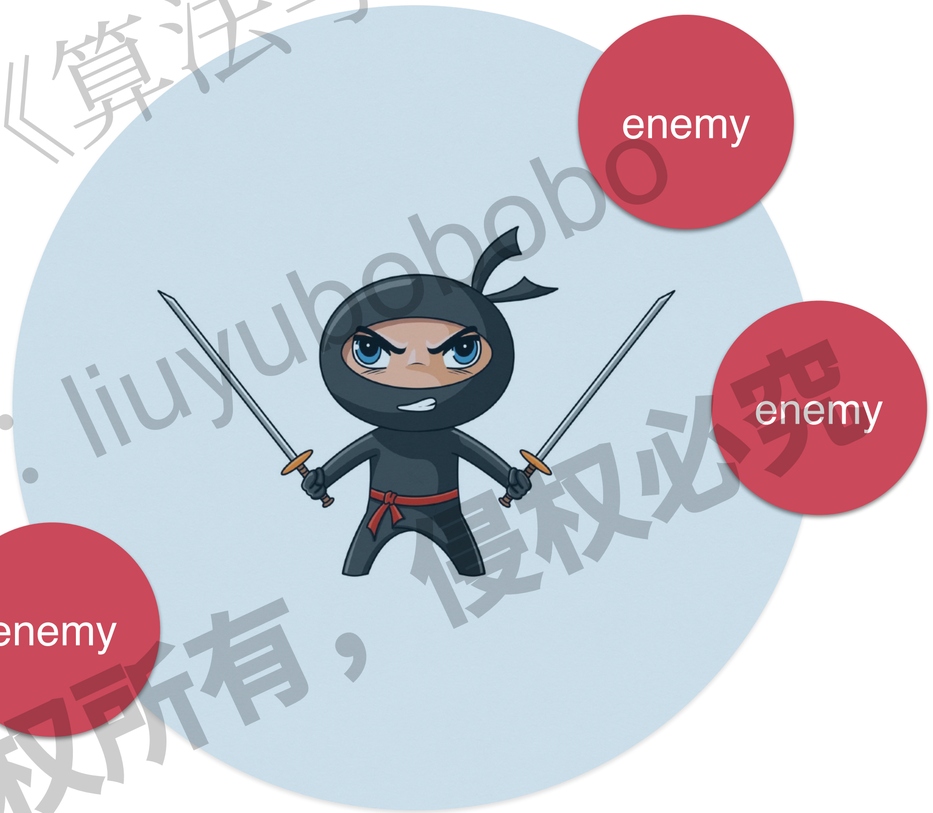
和堆相关的问题

使用堆实现优先队列

动态选择优先级最高的任务执行



使用堆实现优先队列



enemy

enemy

enemy

enemy

enemy

enemy

enemy

使用堆实现优先队列

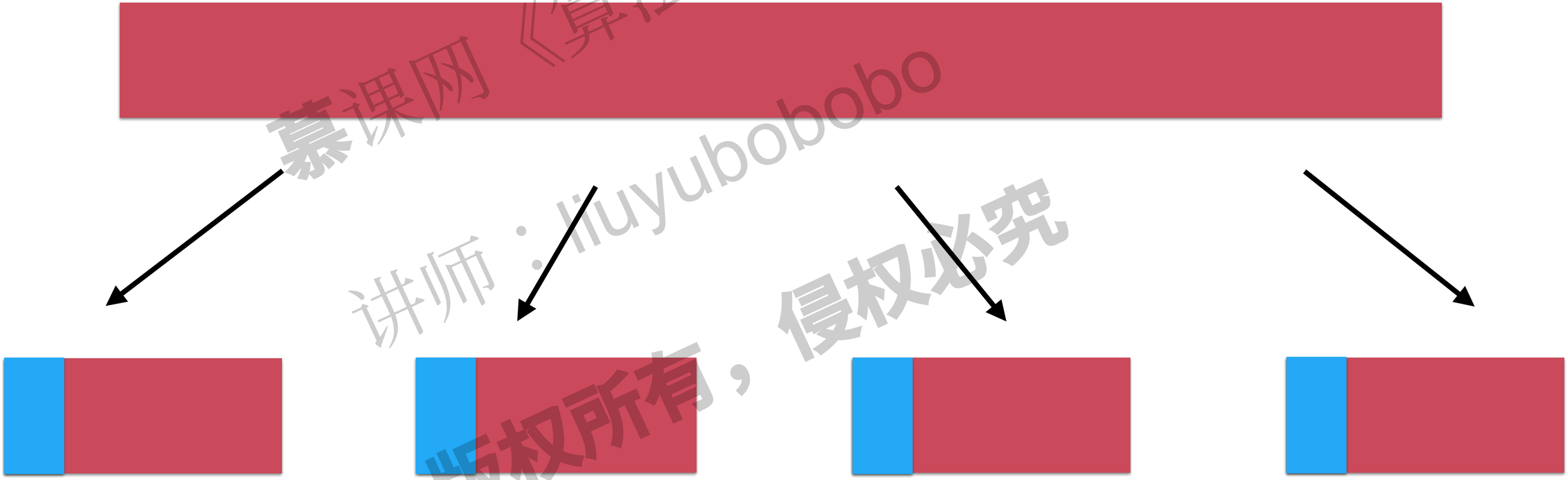
在1,000,000个元素中选出前100名?

在N个元素中选出前M个元素

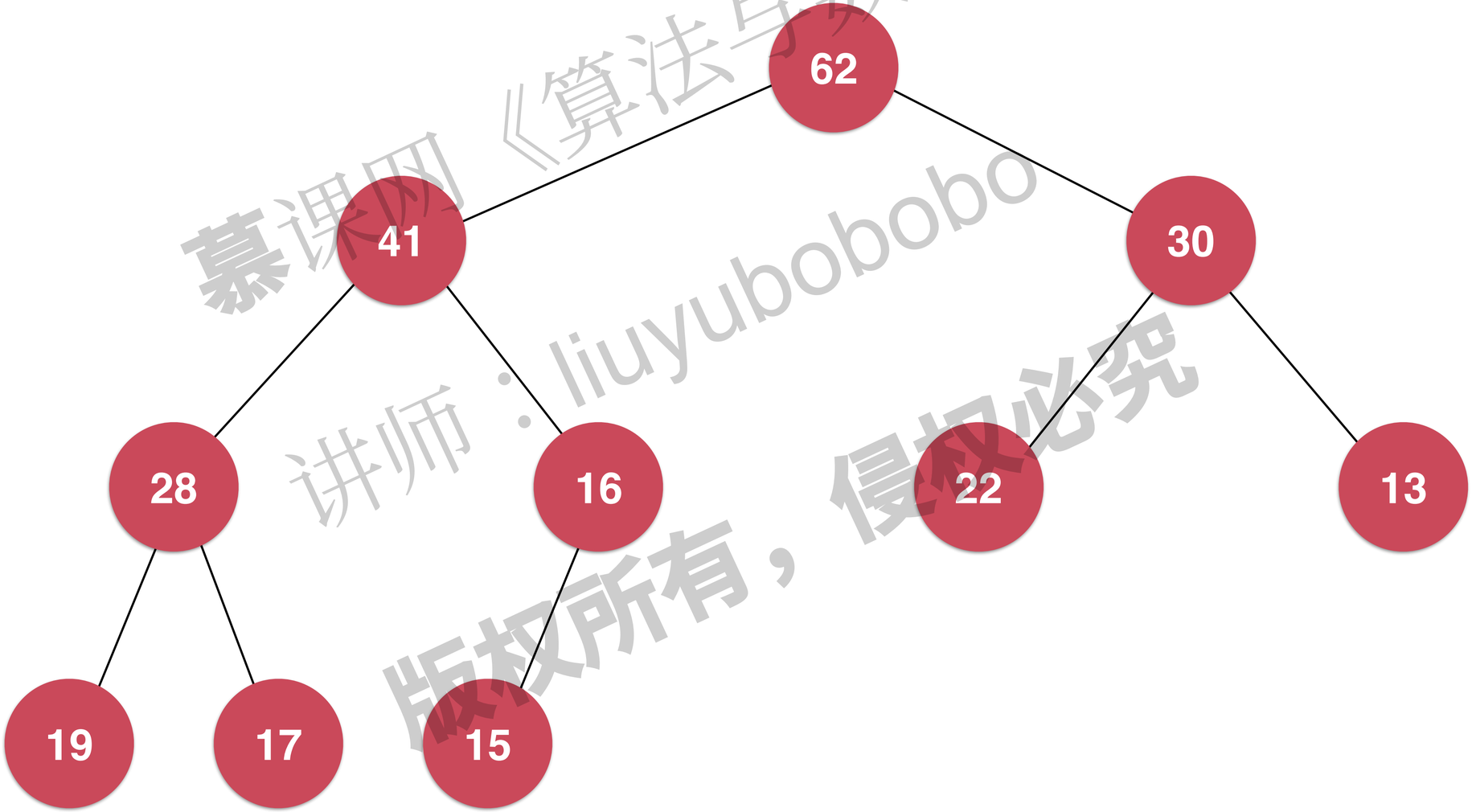
使用优先队列? NlogM

多路归并排序

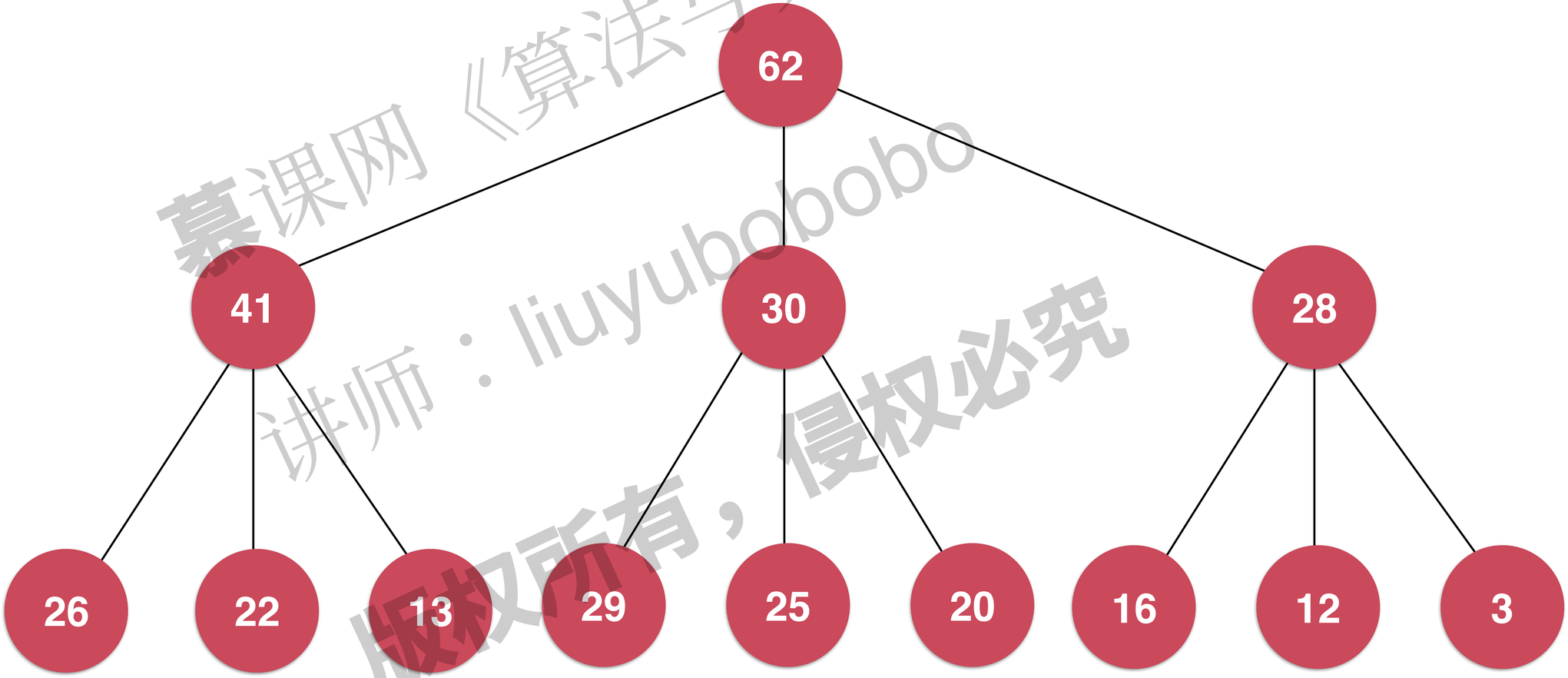
多路归并排序



二叉堆 Binary Heap



d 叉堆 d-ary heap



最大堆 最大索引堆

最小堆 最小索引堆

堆的实现细节优化

ShiftUp 和 ShiftDown 中使用赋值操作替换swap操作

表示堆的数组从0开始索引

没有capacity的限制，动态的调整堆中数组的大小

最大最小队列

二项堆

斐波那契堆

其他

欢迎大家关注我的个人公众号：是不是很酷

算法与数据结构

liuyubobobo