10.4.3 堆排序

➢ 对树型排序的进一步改进。

堆的定义:

- 堆是满足下列性质的记录$\{r_1, r_2, \ldots, r_n\}$:

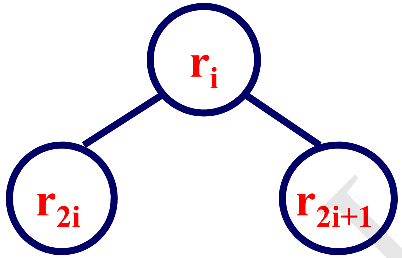
$$\begin{cases} r_i \le r_{2i} \\ r_i \le r_{2i+1} \end{cases} \text{(小根堆)} \quad \text{或} \quad \begin{cases} r_i \ge r_{2i} \\ r_i \ge r_{2i+1} \end{cases} \text{(大根堆)}$$

✓ 例如:

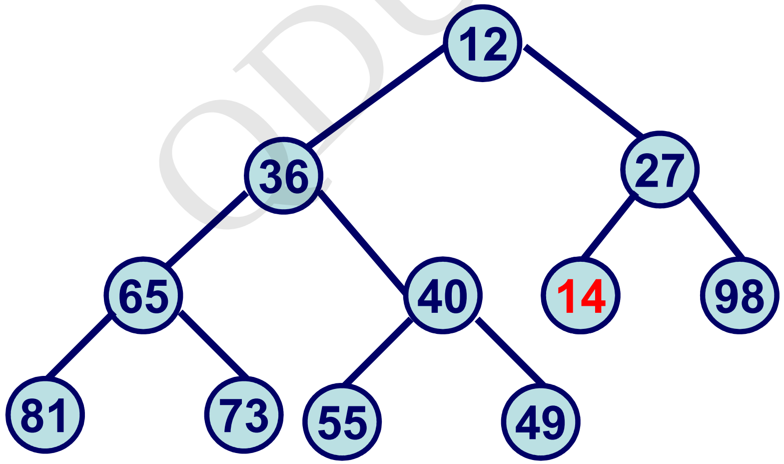
{12, 36, 27, 65, 40, 34, 98, 81, 73, 55, 49} 是小顶堆

{12, 36, 27, 65, 40, 14, 98, 81, 73, 55, 49} 不是堆

■ 若将该记录序列视作完全二叉树，则：r_{2i} 是 r_i 的左孩子；r_{2i+1} 是 r_i 的右孩子。



✓ 例如：{12, 36, 27, 65, 40, 14, 98, 81, 73, 55, 49}



不是小根堆

10.4.3 堆排序

※ 堆排序即是利用堆的特性对记录序列进行排序。

➢ 堆排序采用堆结构选择最大（最小）元素。

※ 设排序记录为R[1..n]。

➢ 将R[1..n]看成是一棵完全二叉树的顺序存储结构。

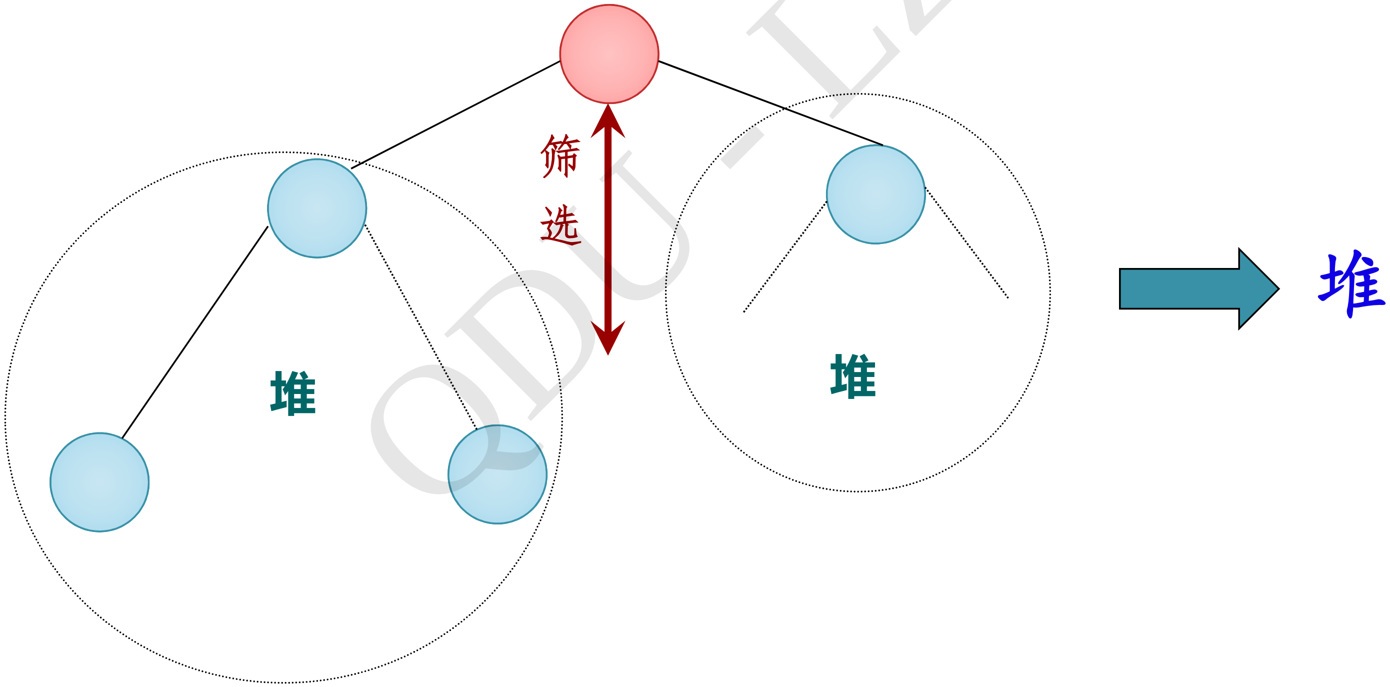
➢ 如果每个结点的关键字均大于等于其所有孩子结点的关键字，称为**大根堆**。

➢ 如果每个结点的关键字均小于等于其所有孩子结点的关键字，称为**小根堆**。

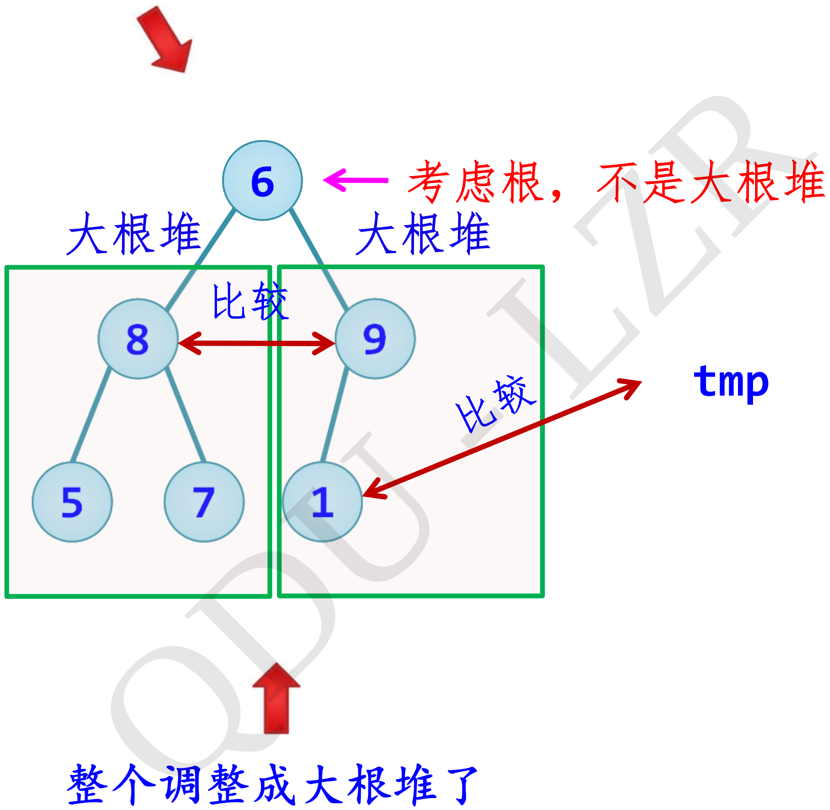
※ 本节的堆排序采用的大根堆。

※ 堆排序的关键是构造堆,这里采用筛选算法建堆。

※ 所谓“筛选”指的是，对一棵左、右子树均为堆的完全二叉树，“调整”根结点使整个二叉树也成为一个堆。



例如：（6,8,9,5,7,1）



堆排序过程：

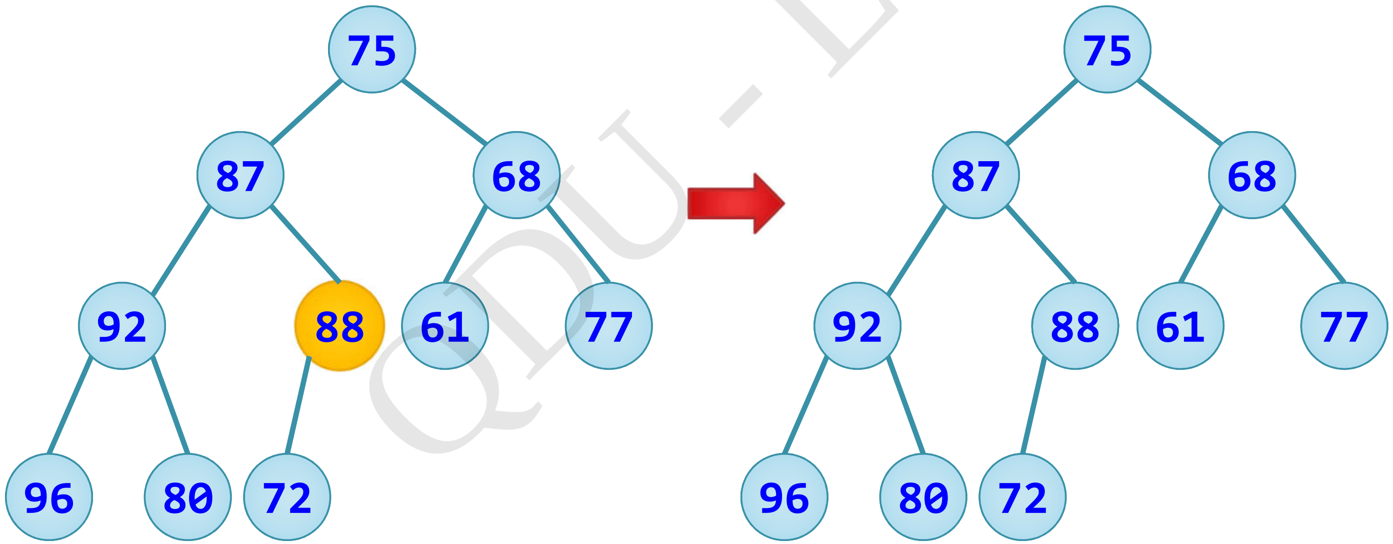
※ 从最后一个分支结点（编号为n/2）开始到根结点（编号为1）通过多次调用筛选算法建立初始堆。

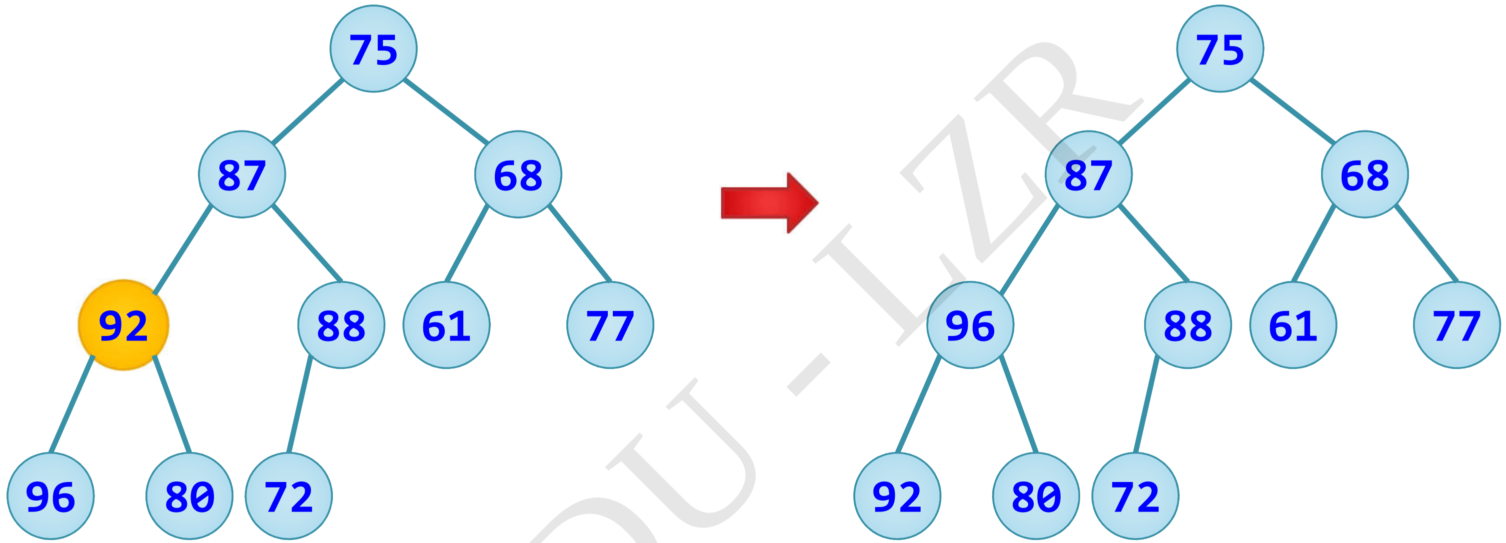
※ 排序过程：

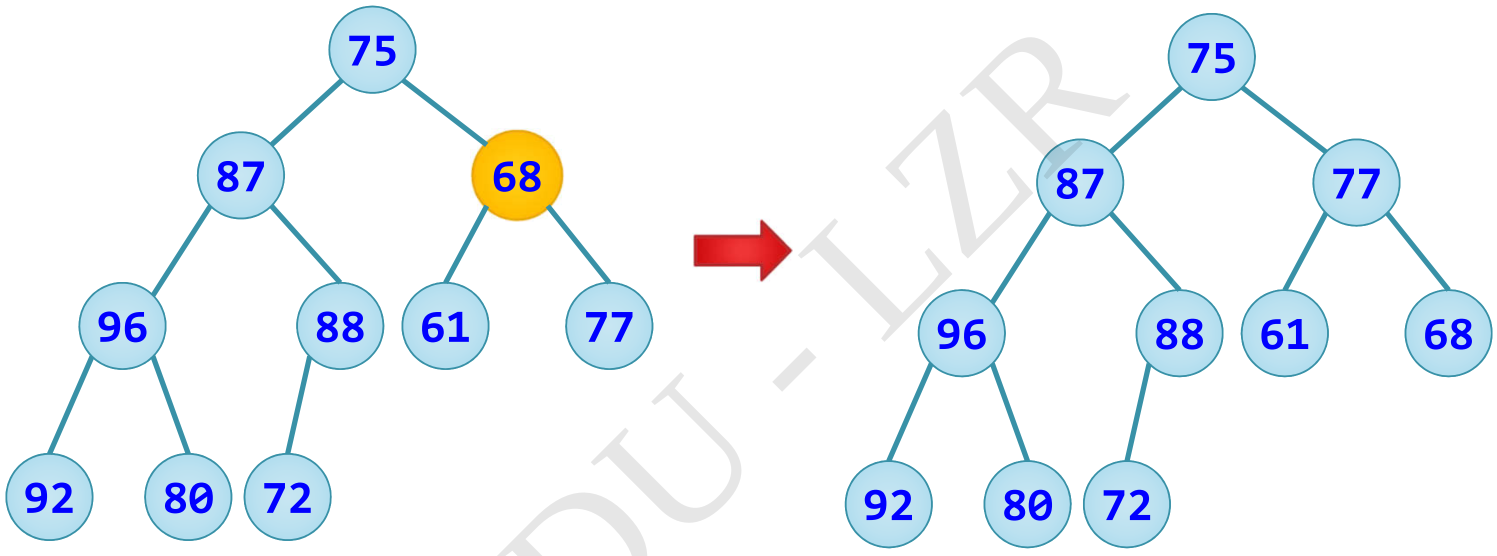
- 将R[1]（无序区中最大记录）与无序区最后一个记录交换，归位无序区最后一个记录，无序区减少一个记录。
- 再筛选，重复进行，直到无序区只有一个记录。

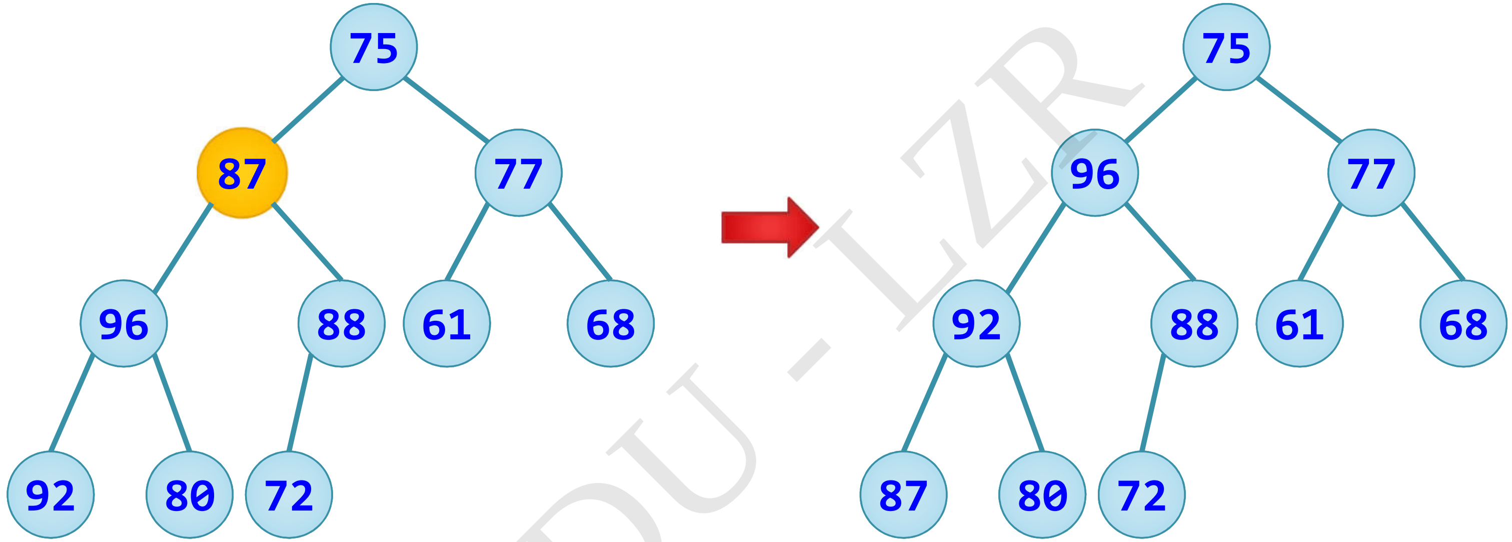
【示例-1】 已知有10个待排序的记录，它们的关键字序列为（75，87，68，92，88，61，77，96，80， 72），给出用堆排序法进行排序的过程。

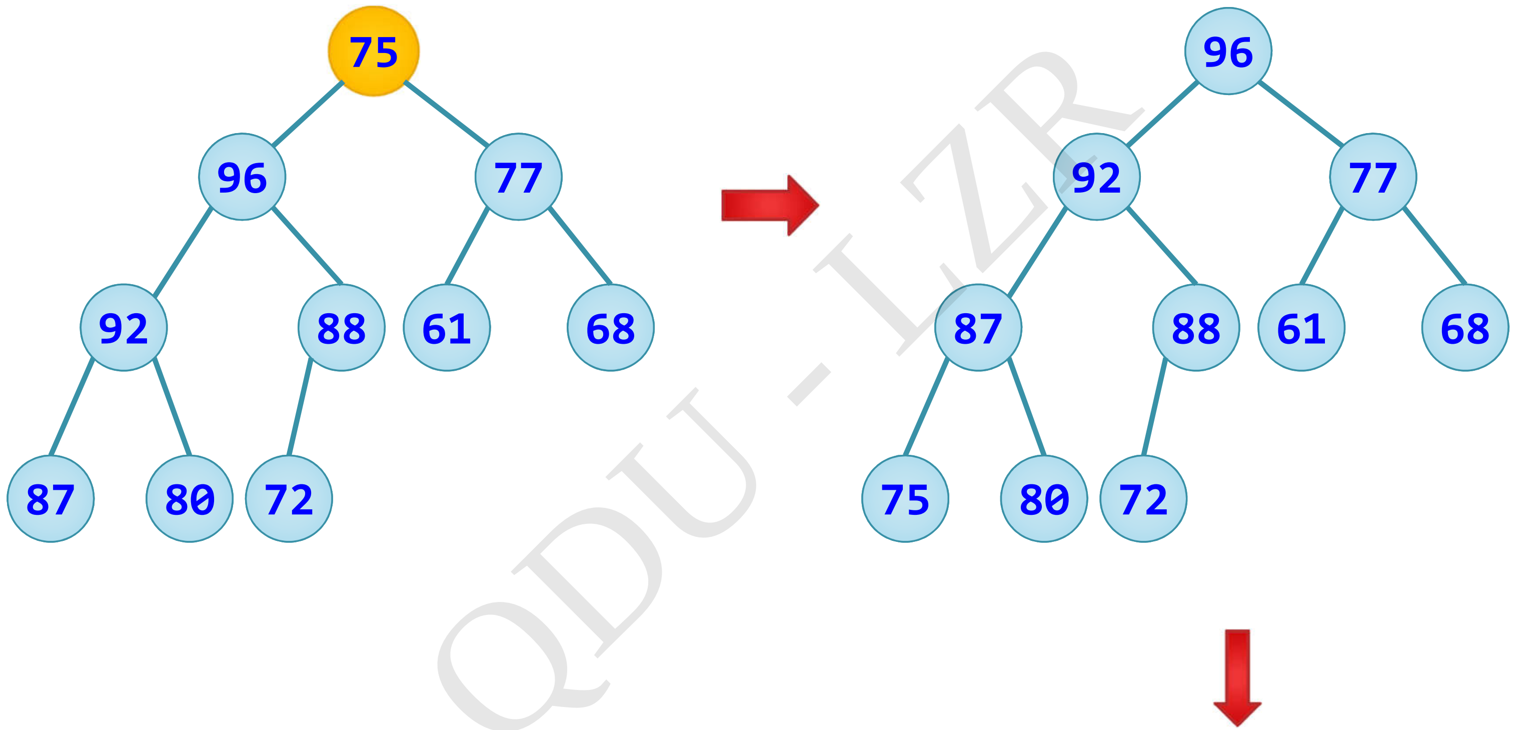
（1）建立初始堆




75
87
68
92
88
61
77
96
80
72
75
87
68
96
88
61
77
92
80
72

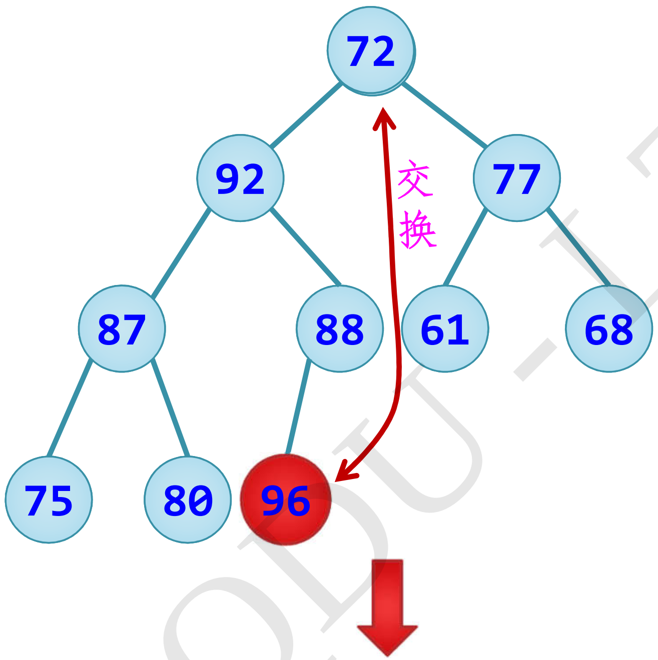

75
87
68
96
88
61
77
92
80
72
75
87
77
96
88
61
68
92
80
72


75
87
77
96
88
61
68
92
80
72
75
96
77
92
88
61
68
87
80
72

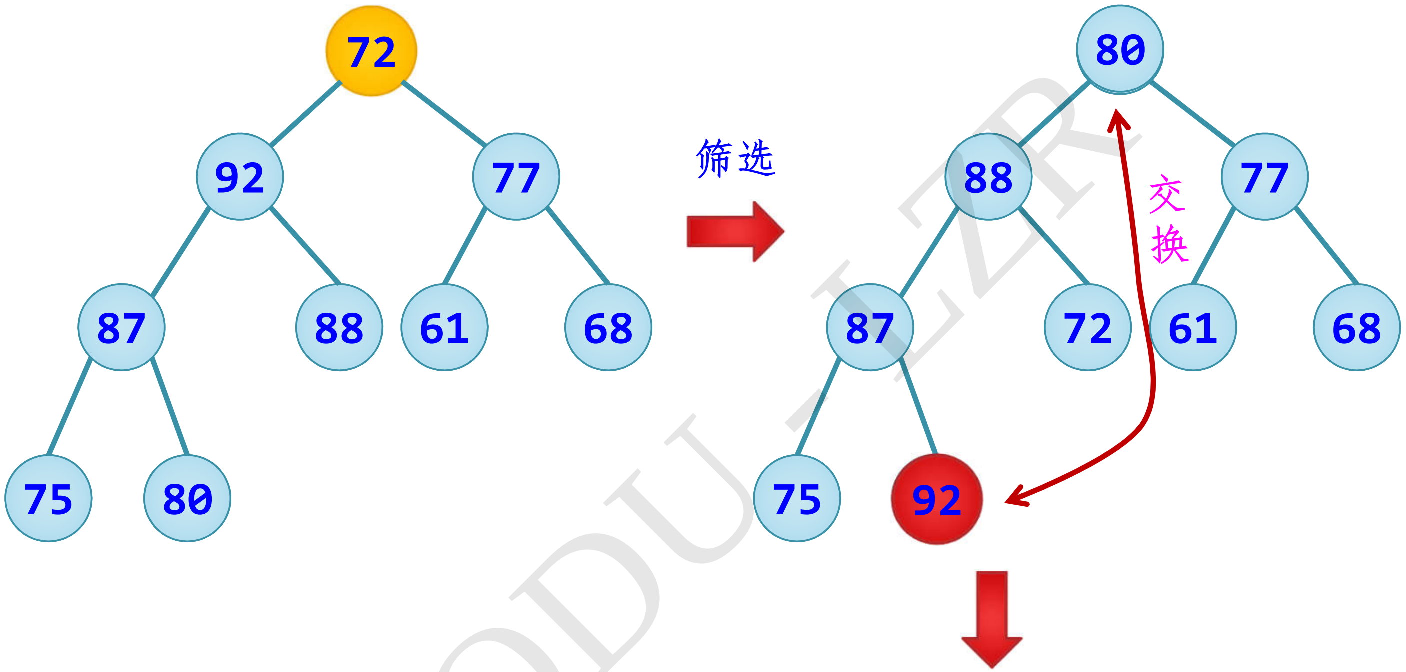


初始堆：
96,92,77,87,88,61,68,75,80,72

(2) 排序过程



第1趟排序结果（i=10）初始堆:
72,92,77,87,88,61,68,75,80,96



第2趟排序结果（*i*=9）初始堆：

80,88,77,87,72,61,68,75,92,96

以此
类推

最终结果：61,68,72,75,77,80,87,88,92,90

假设对*R*[low..high]进行堆调整，它是一棵满足筛选条件的完全二叉树，即以*R*[low]为根结点的左子树和右子树均为堆，其调整堆的算法如下：

```
typedef SqList HeapType; // 堆采用顺序表存储表示
void HeapAdjust(HeapType &H, int s, int m)
{ // 已知H.r[s..m]中记录的关键字除H.r[s].key之外均满足堆的定义，本函数
  // RedType rc;调整H.r[s]的关键字，使H.r[s..m]成为一个大顶堆(对其中记
录的关键字而言)。算法10.10
    int j;
    rc = H.r[s];
    for(j = 2 * s; j <= m; j *= 2) {
        // 沿key较大的孩子结点向下筛选
        if(j < m && LT(H.r[j].key, H.r[j + 1].key))
            ++j; // j为key较大的记录的下标
        if(!LT(rc.key, H.r[j].key))
            break; // rc应插入在位置s上
        H.r[s] = H.r[j];
        s = j;
    }
    H.r[s] = rc; // 插入
}
```

堆排序的算法如下：

```
void HeapSort(HeapType &H)
{
    // 对顺序表H进行堆排序。算法10.11
    RedType t;
    int i;
    // 把H.r[1..H.length]建成大顶堆
    for(i = H.length / 2; i > 0; --i)
        HeapAdjust(H, i, H.length);
    // 将堆顶记录和当前未经排序子序列H.r[1..i]中最后一个记录相互交换
    for(i = H.length; i > 1; --i) {
        t = H.r[1];
        H.r[1] = H.r[i];
        H.r[i] = t;
        HeapAdjust(H, 1, i - 1); // 将H.r[1..i-1]重新调整为大顶堆
    }
}
```

堆排序的时间复杂度分析：

- 对高度为k的堆，“筛选”所需进行的关键字比较的次数至多为$2(k-1)$；
- 对n个关键字，建成高度为$h(=\lfloor log_2 n \rfloor+1)$的堆，所需进行的关键字比较的次数不超过$4n$
- 调整“堆顶”$n-1$ 次，总共进行的关键字比较的次数不超过：

  $$2(\lfloor log_2(n-1) \rfloor+\lfloor log_2(n-2) \rfloor+\ldots+log_2 2)<2n(\lfloor log_2 n \rfloor)$$

因此，堆排序的时间复杂度为$O(n\log_2 n)$。

归纳起来，堆排序算法的性能如表所示。

时间复杂度			空间复杂度	稳定性
最好情况	最坏情况	平均情况		
$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定

—— END ——