选择类排序

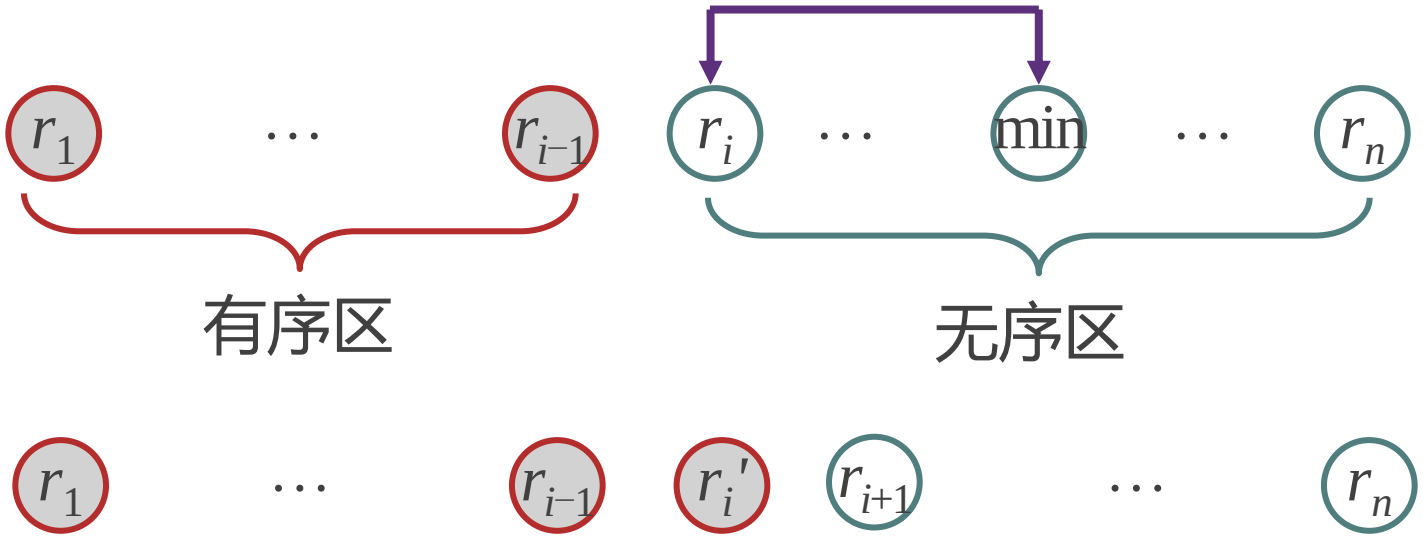
8-3-1　简单选择排序

讲什么？

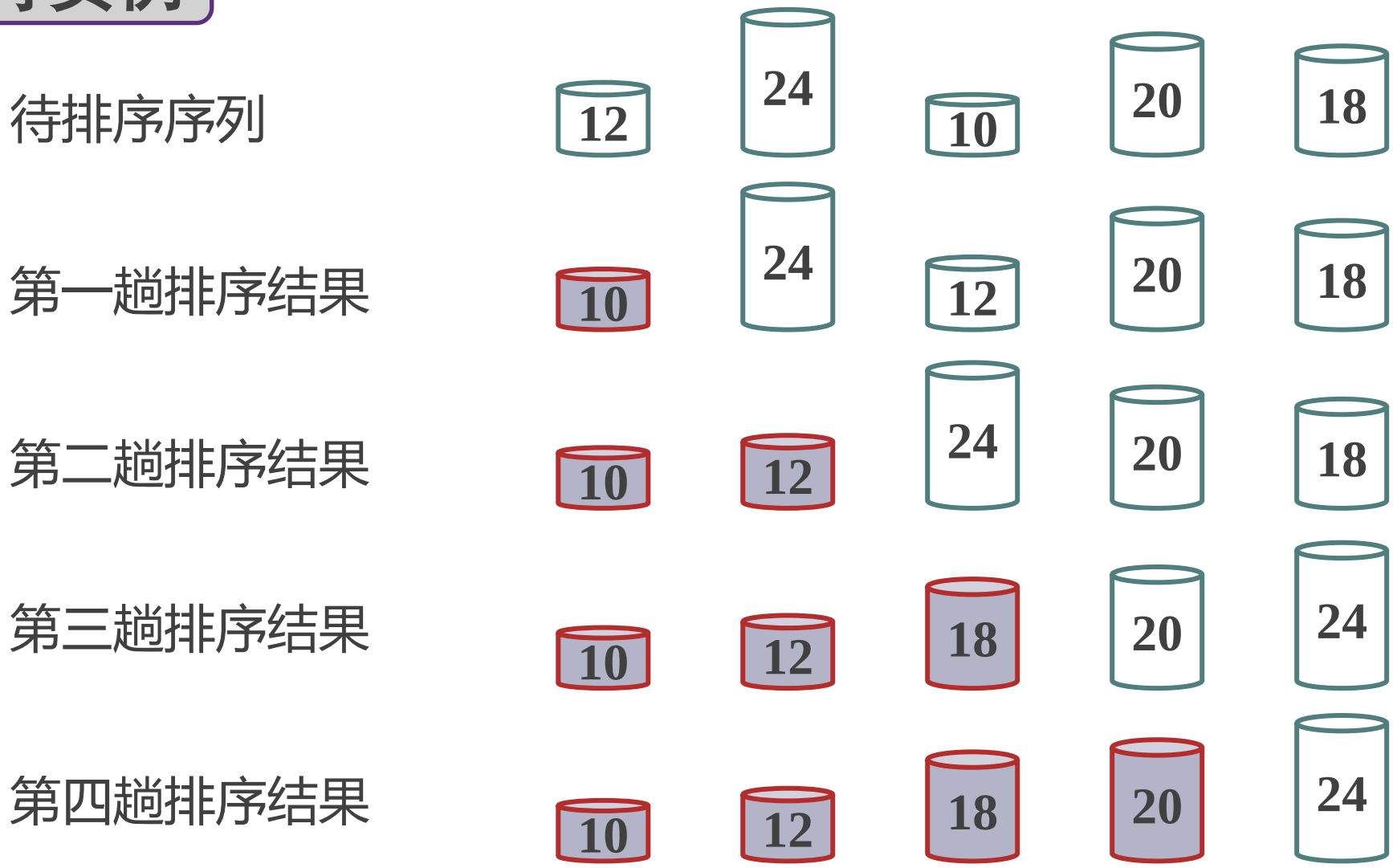
- 简单选择排序的基本思想
- 简单选择排序的算法
- 简单选择排序的时空性能
- 简单选择排序的稳定性

基本思想

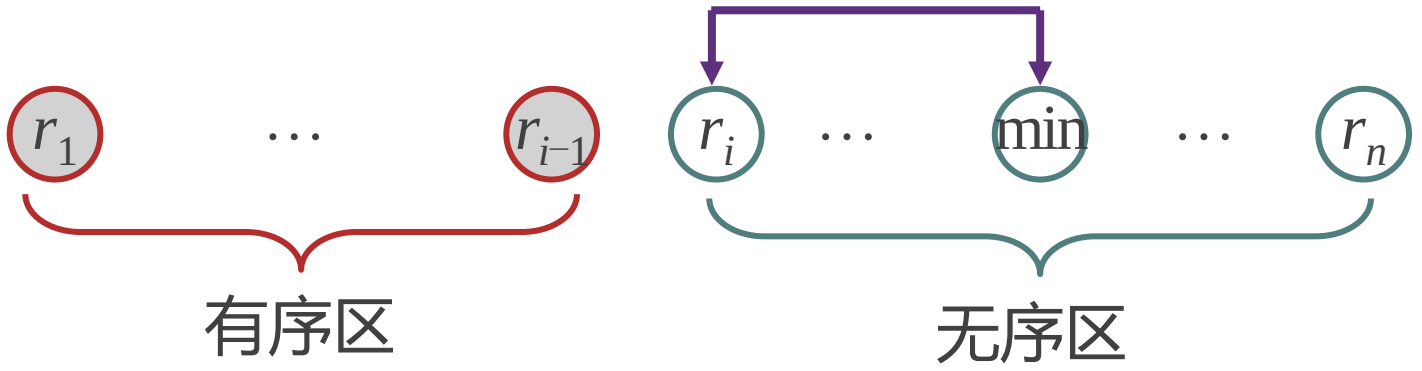
简单选择排序的基本思想：第 i 趟（$1 \le i \le n-1$）排序在待排序序列 $r_i \sim r_n$ 中选取最小记录，并与记录 r_i 交换



运行实例


待排序序列
12
24
10
20
18
第一趟排序结果
10
24
12
20
18
第二趟排序结果
10
12
24
20
18
第三趟排序结果
10
12
18
20
24
第四趟排序结果
10
12
18
20
24

关键问题



简单选择排序进行多少趟?

```
for (i = 0; i < length − 1; i++)
{
    第 i 趟简单选择排序;
}
```

```
index = i;
for (j = i + 1; j <= n; j++)
    if (r[j] < r[index]) index = j;
交换r[i]和r[index];
```

第 i 趟简单选择排序完成什么工作? （1）找最小值; （2）交换

算法描述

```
void Sort::SelectSort( )
{
    int i, j, index, temp;
    for (i = 0; i < length - 1; i++)
    {
        index = i;
        for (j = i + 1; j < length; j++)
            if (data[j] < data[index])
                index = j;
        if (index != i) {
            temp = data[i];
            data[i] = data[index];
            data[index] = temp;
        }
    }
}
```

时间性能

比较语句？执行次数？

$$\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}1=(n-1+\cdots+2+1)=\frac{n(n-1)}{2}$$

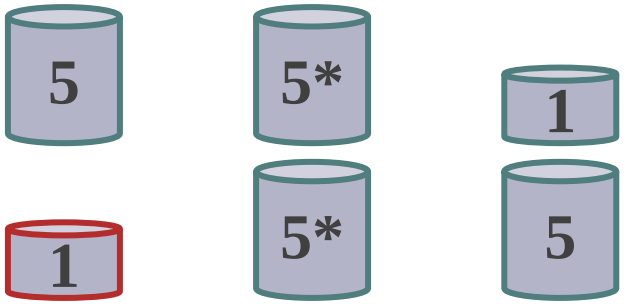
移动语句？执行次数？

- 最好情况：0 次

- 最坏情况：$(n-1)$次

空间性能

- 空间性能：$O(1)$
- 稳定性：不稳定



选择类排序

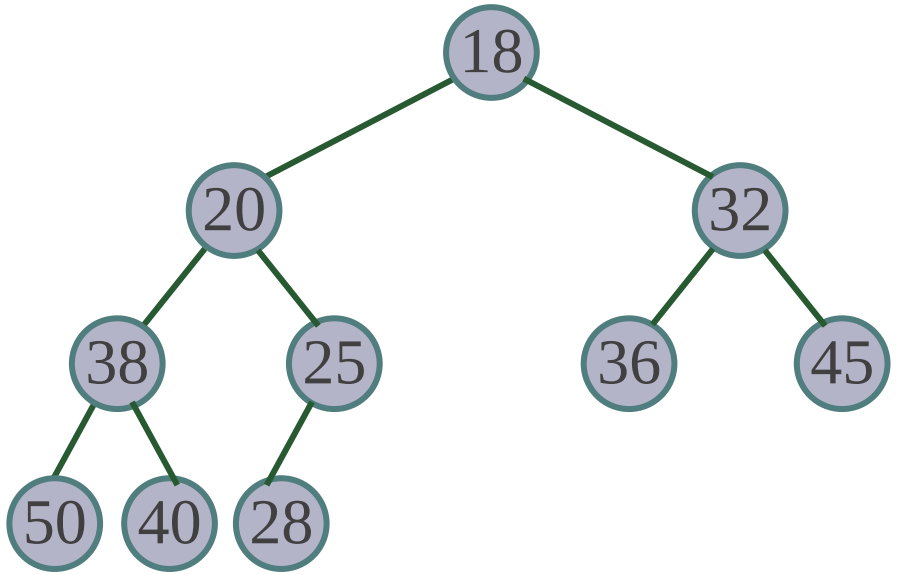
8-3-2　堆排序

讲什么？

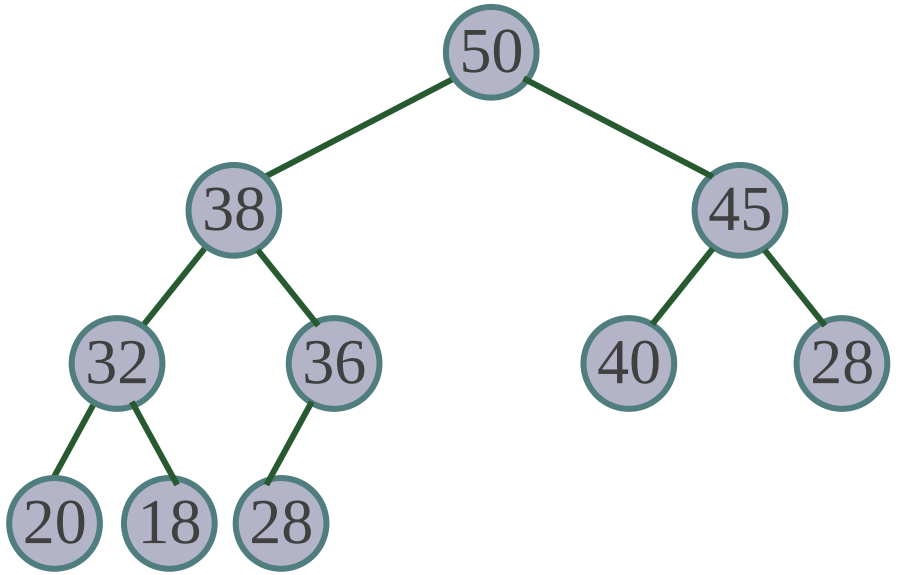
- 堆的定义
- 选择排序改进的着眼点
- 堆排序的基本思想
- 堆调整的算法
- 初始建堆的算法
- 堆排序的算法
- 堆排序的时空性能、稳定性

堆的定义

- 小根堆：每个结点的值都小于等于其左右孩子结点的完全二叉树
- 大根堆：每个结点的值都大于等于其左右孩子结点的完全二叉树
- 堆：小根堆和大根堆统称为堆



小根堆



大根堆

堆的特点

(1) 根（堆顶）是所有结点的最大值

大根堆有什么特点？ (2) 较大值的结点靠近根结点

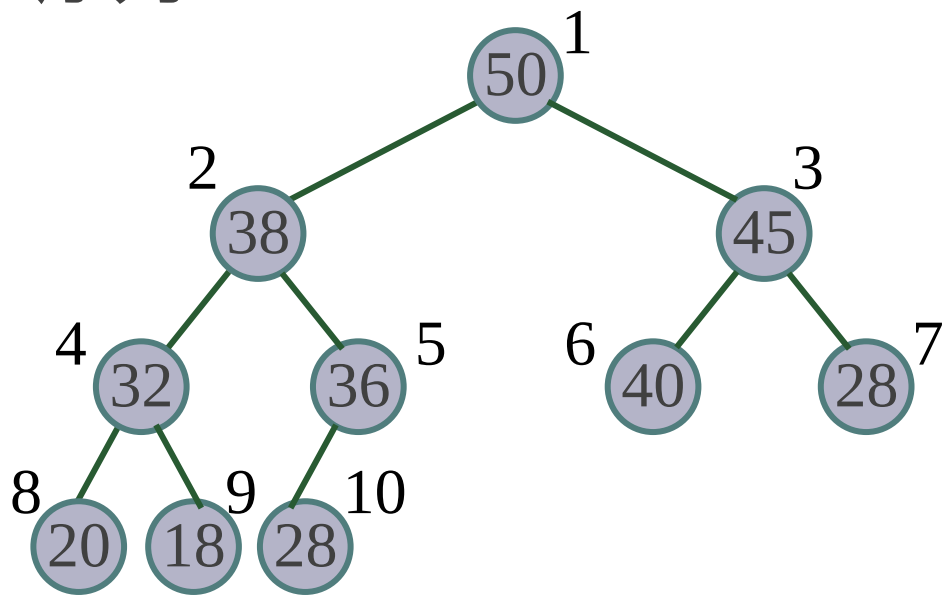
将堆按层序编号，有什么特点？

$$\begin{cases} k_i \geq k_{2i} \\ k_i \geq k_{2i+1} \end{cases} \quad (1 \leq i \leq \lfloor i/2 \rfloor)$$

堆采用顺序存储，则对应一个（无序）序列

1	2	3	4	5	6	7	8	9	10
50	38	45	32	36	40	28	20	18	28

顺序存储，
以编号作为下标



堆的基本操作

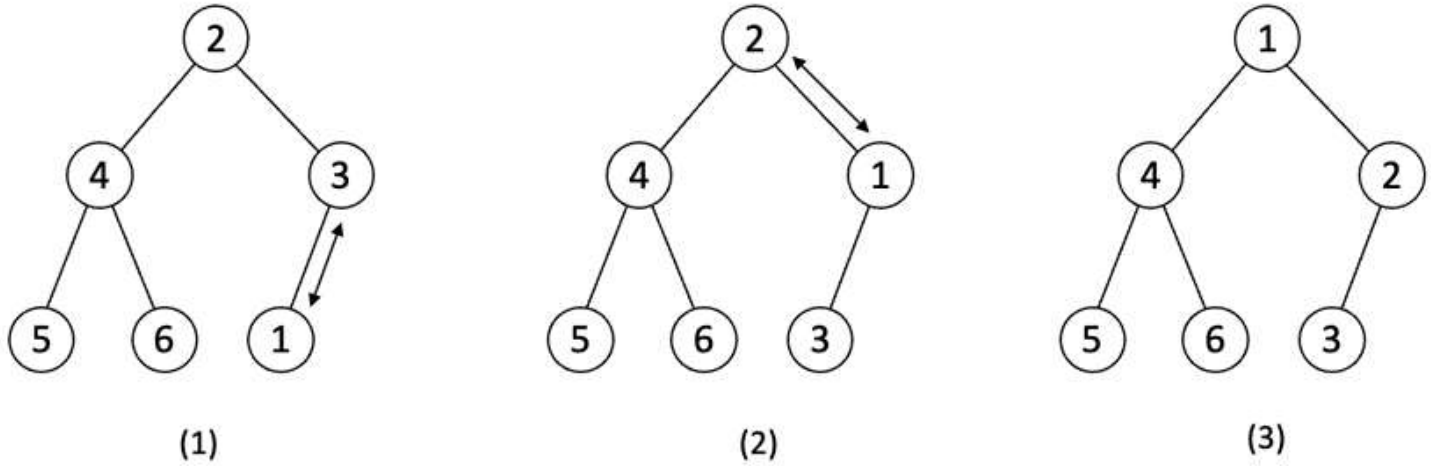
- **堆的基本操作是堆元素的上调和下调（这里的“上”和“下”是指用一般习惯画出二叉堆的树表示后元素在调整过程中的走向）。**
- **在上调和下调操作的基础上，可实现堆元素的插入、删除，以及建堆操作。下面以小顶堆为例介绍。**

二叉堆的上调操作（SiftUp）

如果堆中某结点 i 小于其父结点 p，此时可以交换结点 i 和结点 p 的元素，也就是把结点 i 沿着堆的这棵树往“上”调整。

此时，再看新的父结点与它的大小关系。重复该过程，直到结点 i 被调到根结点位置或者和新的父结点大小关系满足条件。

如图演示了一次二叉堆的上调操作的过程，元素1从开始的叶结点位置一直调整到了根结点。



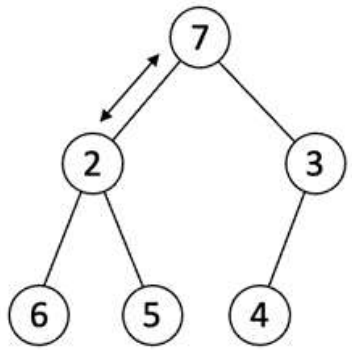
二叉堆的下调操作（SiftDown）

如果堆中某结点 i 大于其子结点，则要将其向“下”调整。

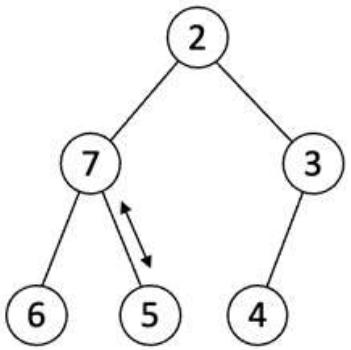
调整时需要注意，对于有两个子结点的情况，如果两个子结点均小于结点 i，交换时应选取它们中的较小者，只有这样才能保证调整之后三者的关系能够满足堆的性质。

重复该过程，直到结点 i 被调到叶结点位置或者和新的子结点大小关系满足条件。

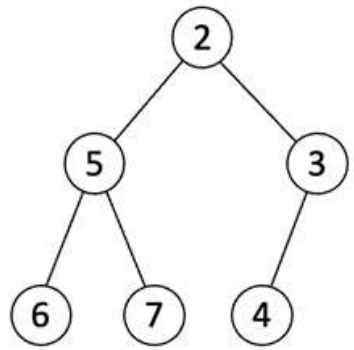
如图演示了一次二叉堆的下调操作的过程，元素7从开始的根结点位置一直调整到了叶结点。



(1)



(2)



(3)

SiftUp上调操作的伪代码实现

设数组 h.data[] 中保存着堆中的元素。

为避免多次交换，可以先将结点 i 的元素保存在临时变量中，随着调整将父结点的元素往“下”移动，最后再将原来结点 i 的元素填入合适的位置，实现插入元素的“上调”（SiftUp函数）。由此得到“上调”操作的算法如下：

算法6-1： 二叉堆的上调操作 SiftUp(*h*, *i*)
输入： 堆 *h* 和上调起始位置 *i*
输出： 上调后满足堆性质的*h*

```
1.  elem← h.data[i]
2.  while i > 1 且 elem < h[i / 2] do  //当前结点小于其父结点
3.  |  h.data[i] ← h.data[i / 2]  //将i的父结点元素下移
4.  |  i ← i / 2  //i指向原结点的父结点，即向上调整
5.  end
6.  h.data[i] ← elem
```

对于上调操作而言，循环的次数不会超过树的高度，因此时间复杂度为$O(\log_2 n)$。

SiftDown下调操作的伪代码实现

下调操作同样可以使用上调操作的方法来避免交换操作，使用该方法实现的下调操作算法如下。对于下调操作而言，循环的次数也不会超过树的高度，因此时间复杂度为$O(\log_2 n)$。

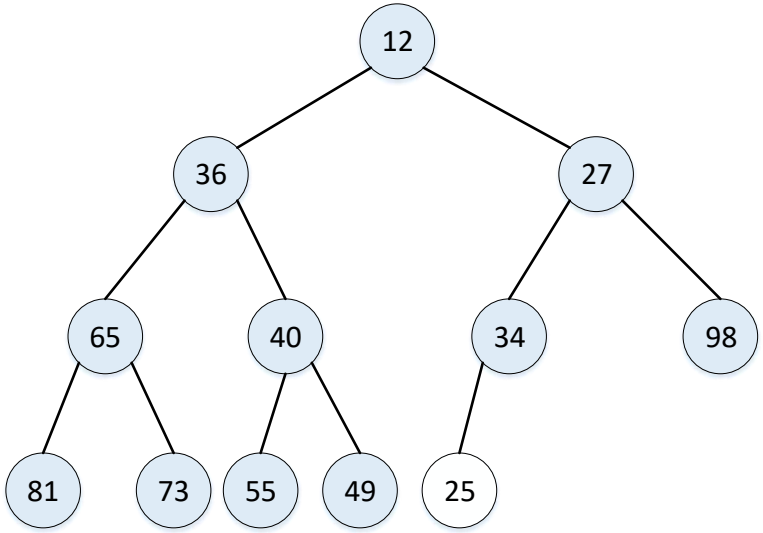
算法6-2： 二叉堆的下调操作 SiftDown(*h*, *i*)
输入： 堆 *h* 和下调起始位置 *i*
输出： 下调后满足堆性质的*h*

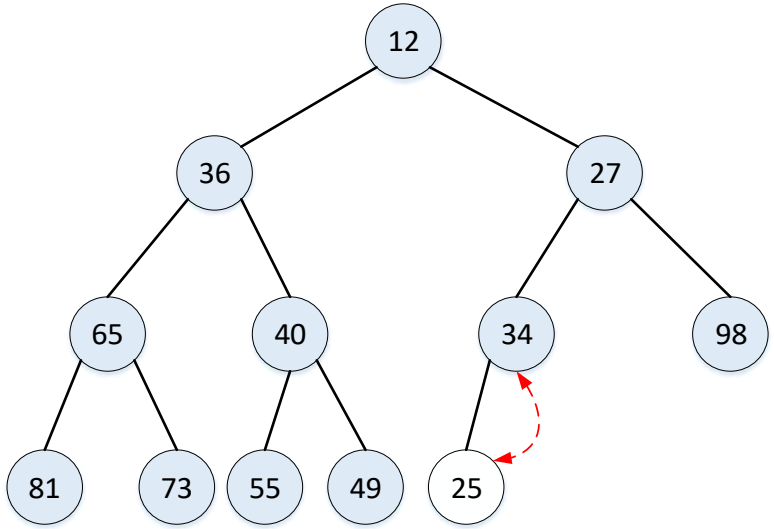
```
1.  last ← h.size  //这是最后一个元素的位置
2.  elem ← h.data[i]
3.  while true do
4.  | child ← 2i  //child当前是i的左孩子的位置
5.  | if child<last 且h.data[child+1]<h.data[child] then //如果i有右孩子并且右孩子更小
6.  | | child ← child + 1  //child更新为i的右孩子的位置
7.  | else if child>last  //如果i是叶子结点
8.  | | break  //已经调整到底，跳出循环
9.  | end
10. | if h.data[child] < elem then  //若较小的孩子比elem小
11. | | h.data[i] ← h.data[child]  //将较小的孩子结点上移
12. | | i ← child  //i指向原结点的孩子结点，即向下调整
13. | else  //若所有孩子都不比elem小
14. | | break  //则找到了elem的最终位置，跳出循环
15. | end
16. end
17. h.data[i] ← elem
```

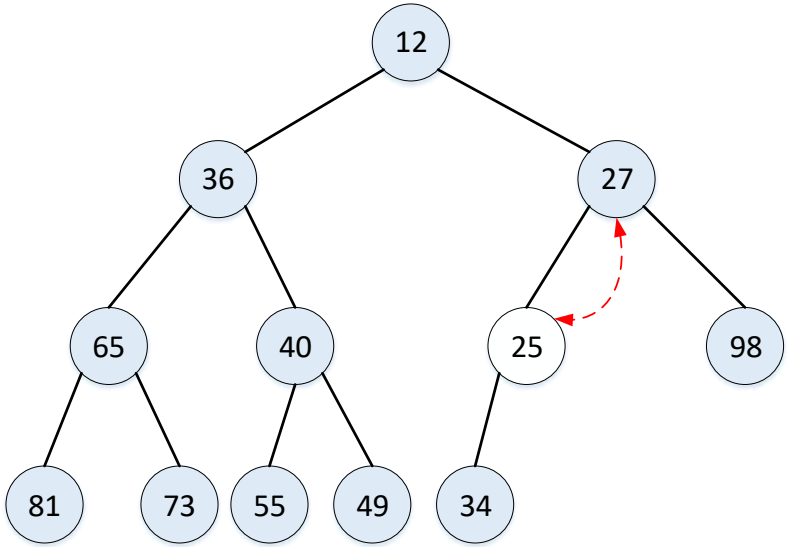
用堆的上调和下调操作实现堆的插入（构建）和删除

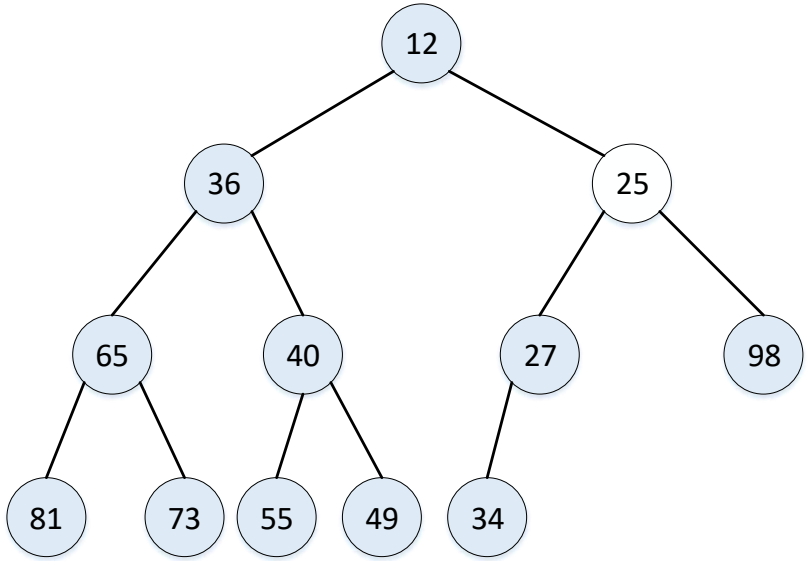
堆的插入操作(以小根堆为例)

堆的插入操作为向堆中追加待插入的元素，可以用“上调”操作将其调整到合适的位置来完成堆的调整。



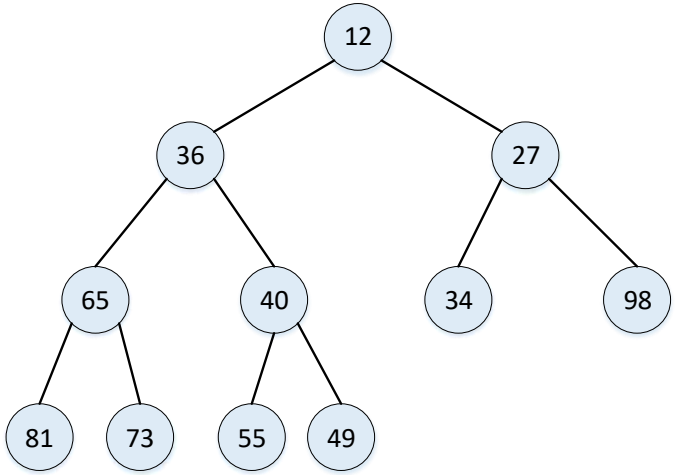


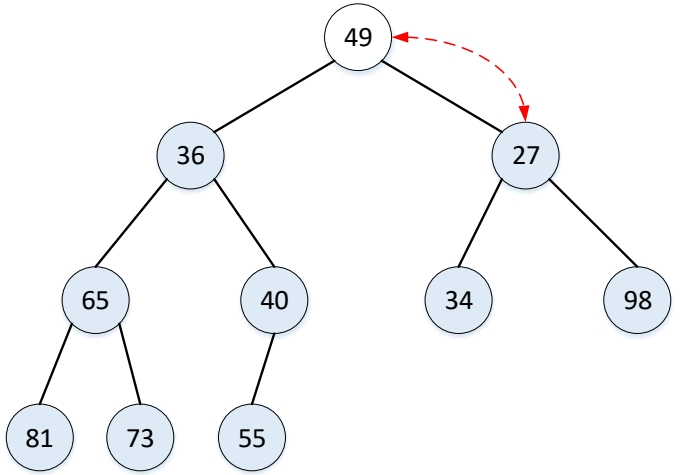


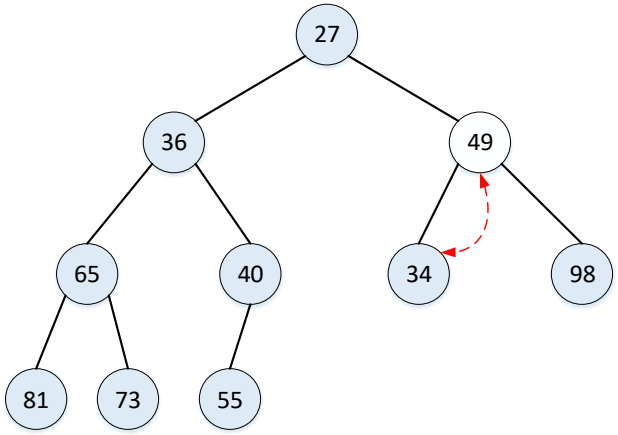


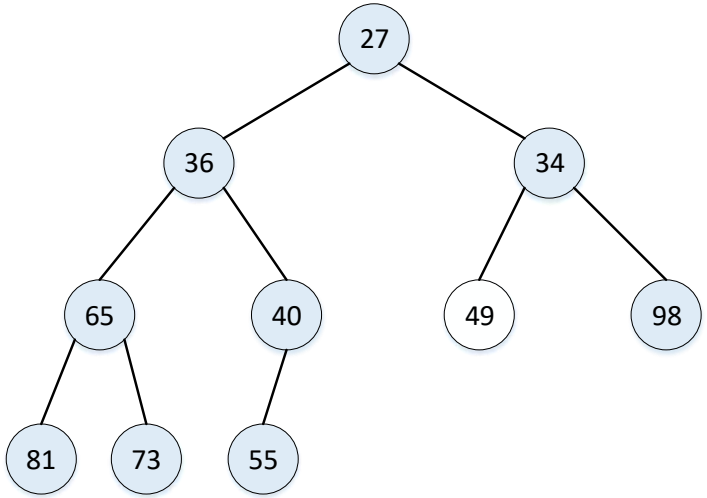
堆的删除操作(以小根堆为例)

删除操作通常是删顶操作，所要提取的最小元素就是堆中的第一个元素，可以把堆中的最后一个元素挪到第一个位置，并通过“下调”操作将这个元素调整到合适的位置。









- 回到堆排序

改进的着眼点

缺点：简单选择排序的时间主要耗费在哪了呢？

- 对无序序列扫描一趟（n-1 次比较）只做了一件事——找最小值

优点：移动次数较少，最坏情况$O(n)$

改进的着眼点：利用简单选择排序的思想，同时减少比较次数

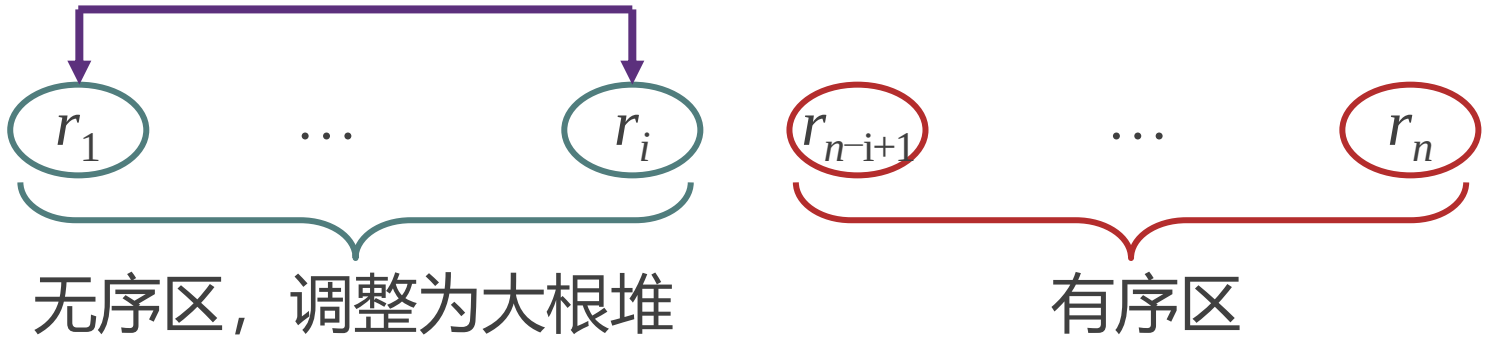
减少后面选择所用的比较次数 ⇦ 提高整个排序的效率

⇧

查找最小值的同时找出并保存较小值 ⇨ 利用每趟比较后的结果

基本思想

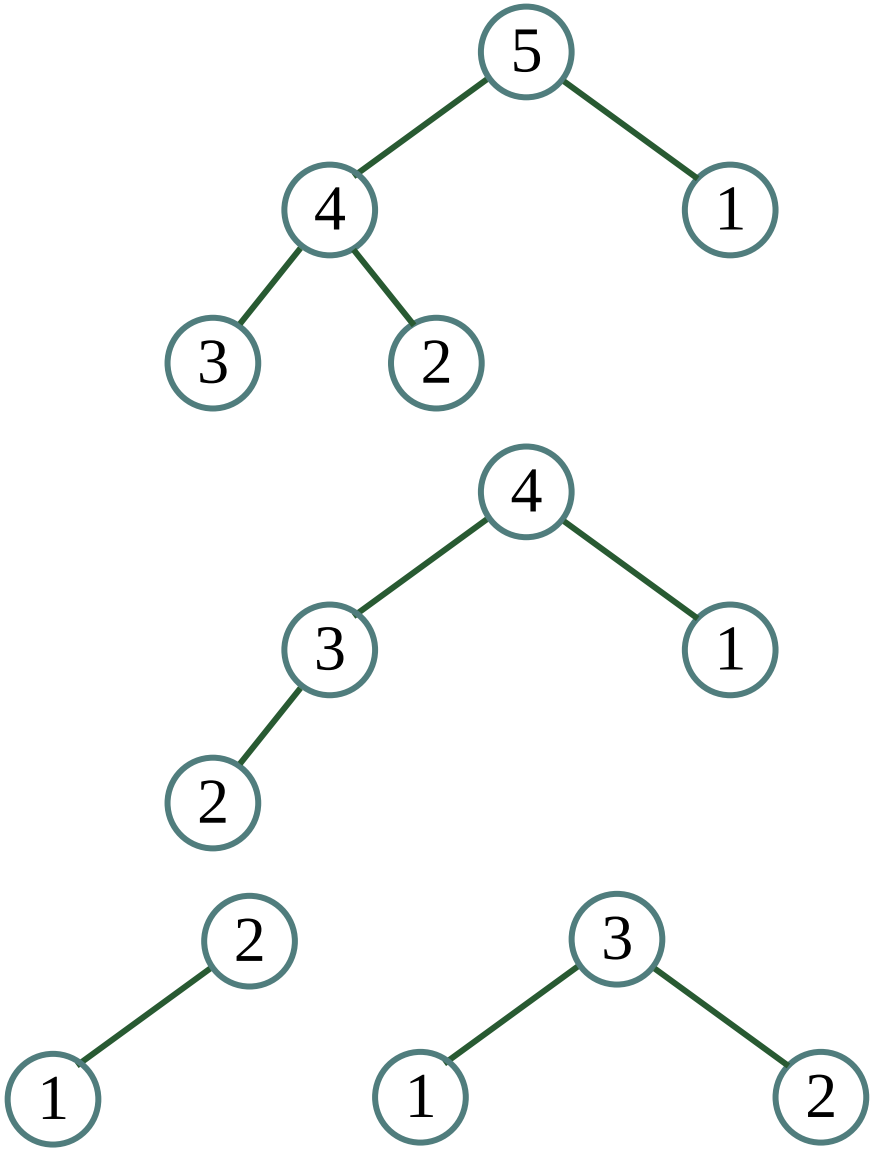
堆排序的基本思想：首先将待排序序列构造成一个大根堆，即选出了堆中所有记录的最大者，将它从堆中移走，并将剩余的记录再调整成堆，这样又找出了次小的记录，以此类推，直到堆中只有一个记录。



第 i 趟堆排序将 $r_1 \sim r_i$ 调整成大根堆，再将堆顶与 r_i 交换

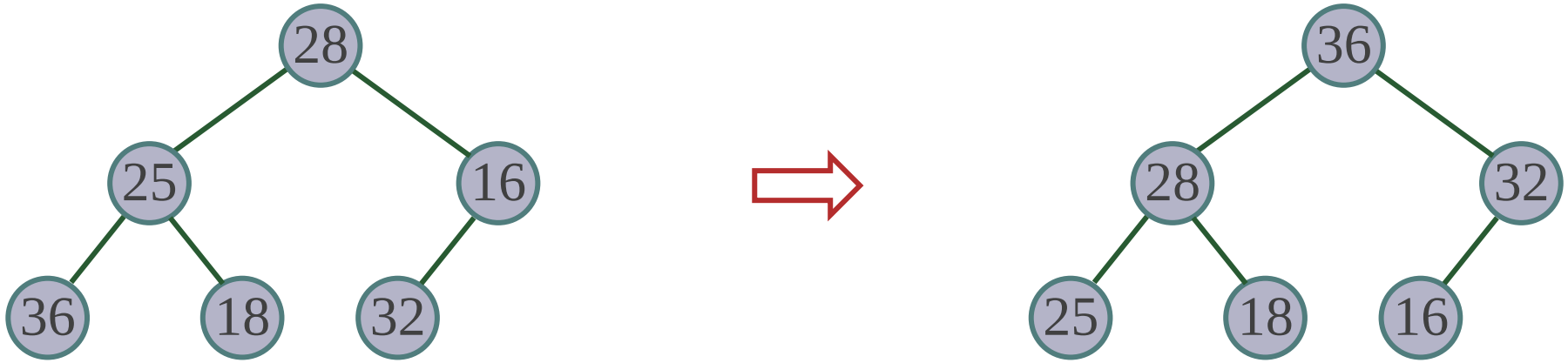
运行实例


待排序序列
2 3 1 5 4
初始建堆
5 4 1 3 2
交换r[1]和r[5]
2 4 1 3 5
无序区重建堆
4 3 1 2 5
交换r[1]和r[4]
2 3 1 4 5
无序区重建堆
3 1 2 4 5
交换r[1]和r[3]
2 1 3 4 5
最后一趟
1 2 3 4 5


5
4 1
3 2
4
3 1
2
2
1 3
1 2

初始建堆

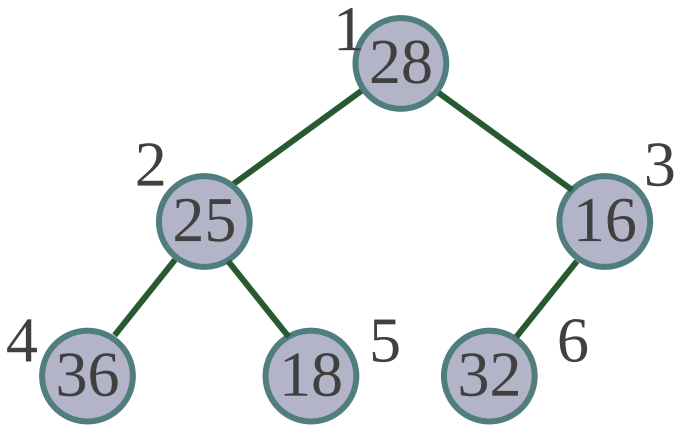
待排序序列　{28, 25, 16, 36, 18, 32}　⇨　初始建堆结果　{36, 28, 32, 25, 18, 16}



如何将一个无序序列建成一个大根堆——初始建堆?

关键问题

待排序序列　{28, 25, 16, 36, 18, 32}



解决办法：

从编号最大的**分支结点**到根结点进行调整

分支结点中编号最大的是多少?

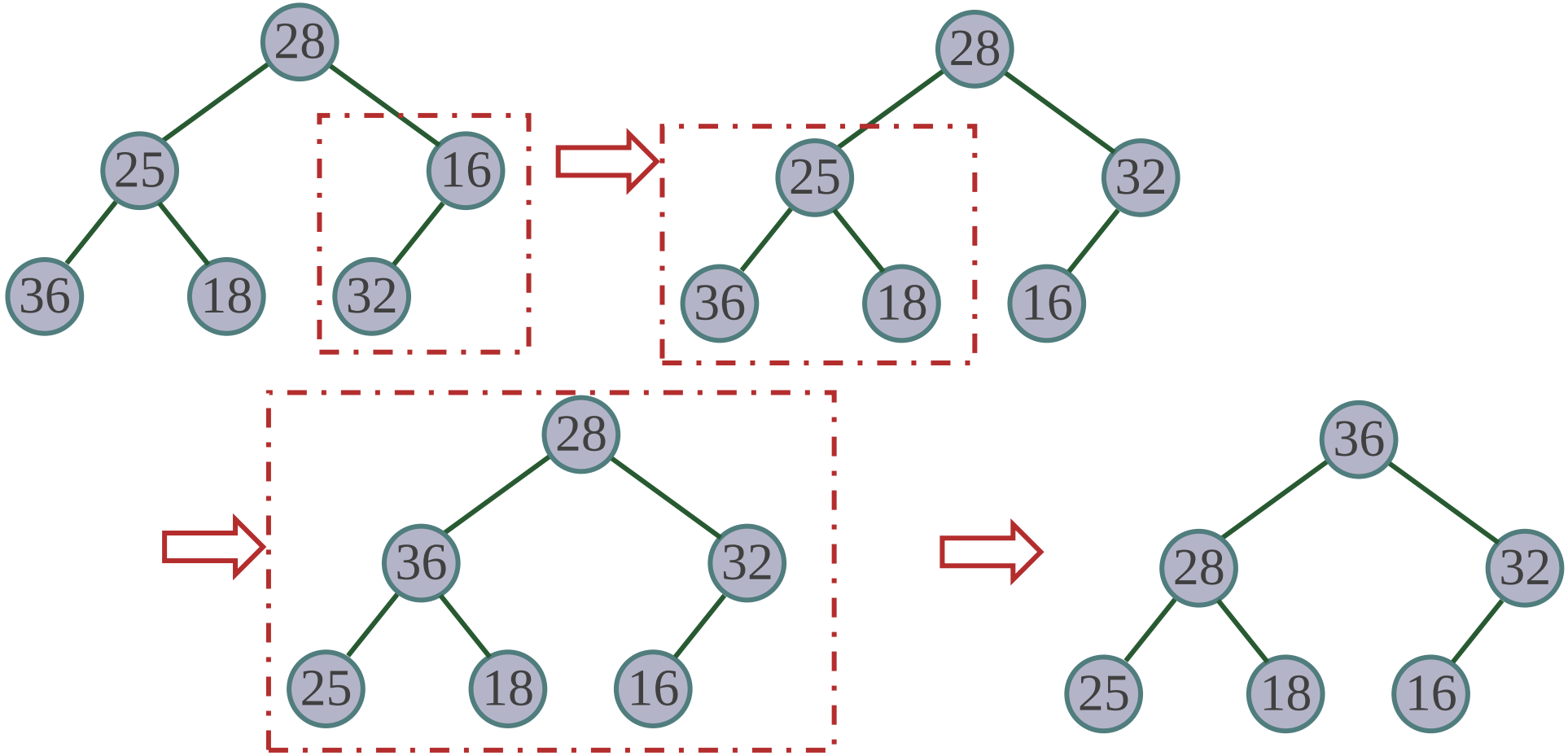
假设待排序序列长度是n，即含n个结点，最大编号的分支结点就是n/2, 所以只要调整以n/2, n/2-1,...,1的结点为根的子树就可以了。

需要调整叶子结点吗?

每个叶子结点就是一个堆，所以不需要调整

运行实例

待排序序列　{28, 25, 16, 36, 18, 32}

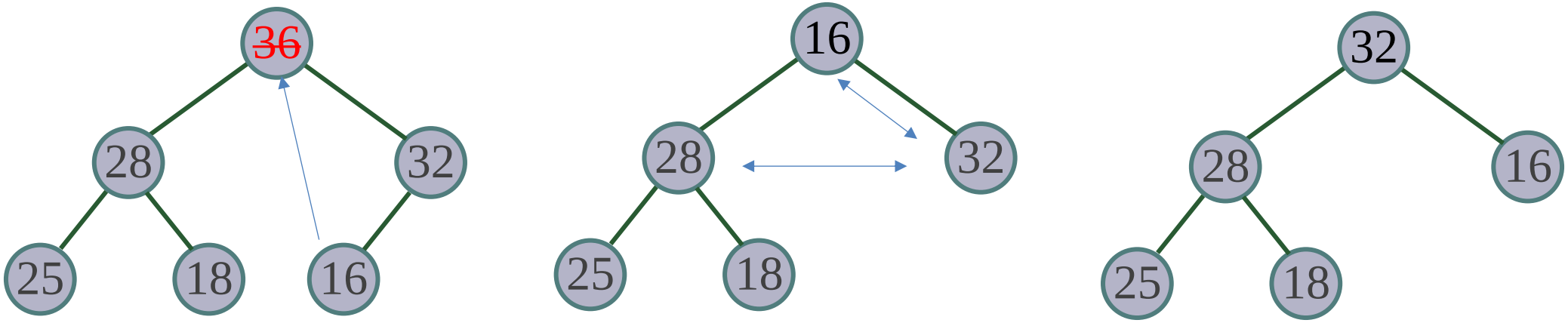


初始建堆结果　{36, 28, 32, 25, 18, 16}

运行实例

初始建堆完成后，删顶，把堆顶元素与无序区的最后一个元素交换，然后--》

重建堆



重建堆　{32,28, 16, 25, 18}+{36}

算法描述

```
void Sort::HeapSort( )
{
    int i, temp;
     //初始建堆
    for (i = ceil(length/2) －  1; i >= 0; i−−)
        Sift(i, length-1) ;
   //建堆后，  data[0]放的是堆顶元素
    for (i = 1; i < length; i++)
    {
        temp = data[0];
        data[0] = data[length-i];
        data[length-i] = temp;
        Sift(0, length-i-1);   //重建堆
    }
}
```

```
void Sort :: Sift(int k, int last)       //k是根，last是最后一个结点
{
    int i, j, temp;
    i = k; j =2 * i + 1;                          // i是被调整结点，j是i的左孩子
    while (j <= last)                             //还没有进行到叶子
    {
        if (j < last && data[j] < data[j+1]) j++;       // j指向左右孩子的较大者
        if (data[i] > data[j]) break;                   //已经是堆
        else {
            temp = data[i]; data[i] = data[j]; data[j] = temp;
            i = j; j = 2 * i + 1;                       //被调整结点位于结点j的位置
        }
    }
}
```

时间性能

- 初始建堆*：$O(n)$

注：该ppt里的初始建堆，采用的是101教材P196的快速建堆方法，即从最大编号的分支结点，逐个对子树的根结点的进行从上到下的调整，时间复杂度是O（n）；若采用P196的朴素建堆，初始建堆时间复杂度会达到$O(n\log_2 n)$

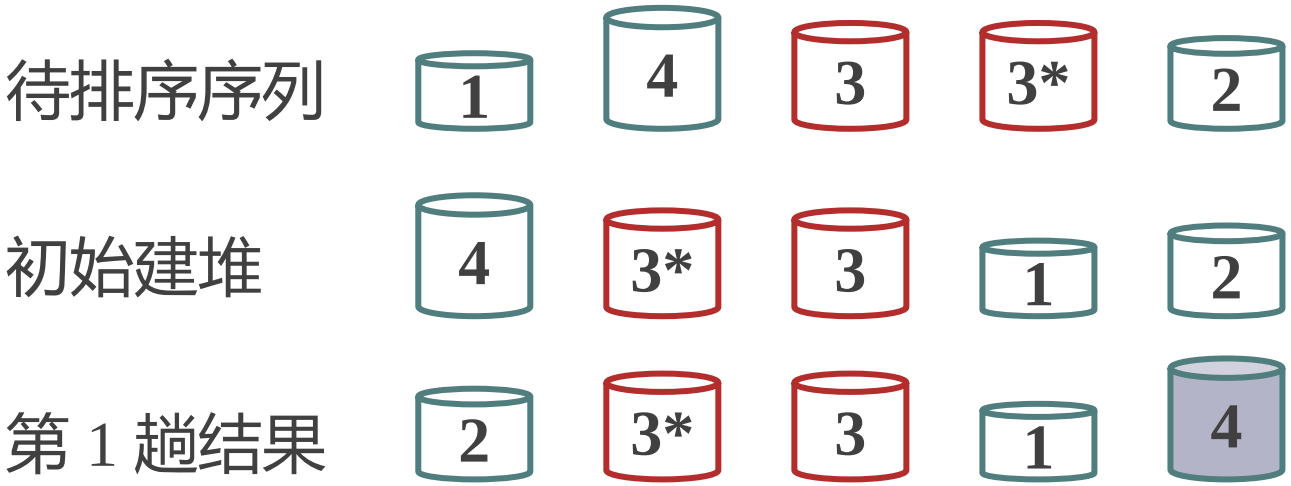
- 重建堆次数：$n-1$
- 重建堆：$O(\log_2 i)$
- 最好、最坏、平均：$O(n\log_2 n)$

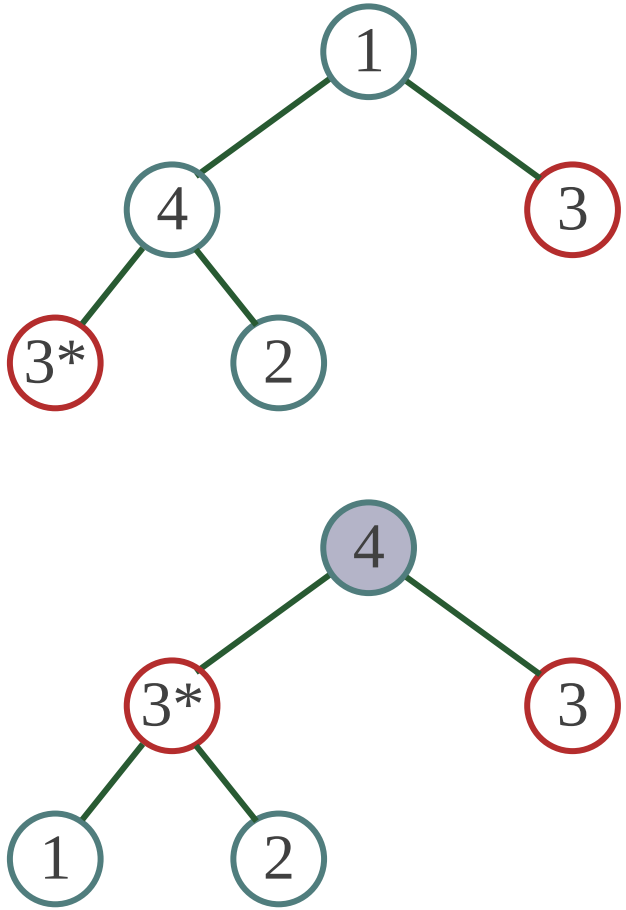
性能分析

- （1）对高度为h的堆，一趟筛选最多比较2(h-1)次，最多交换h-1次；
- （2）n个关键字建成的堆的高度$h = \lfloor \log_2 n \rfloor + 1$，在建好堆后，排序过程中n-1次筛选总的比较次数不超过$2(\lfloor \log_2(n-1) \rfloor + \lfloor \log_2(n-2) \rfloor + \cdots + \lfloor \log_2 2 \rfloor) < 2n\log_2 n$；
- （3）建初始堆时，做$\lfloor n/2 \rfloor$次筛选，但由于每次筛选所针对的堆高度为2到h不等，总的比较次数至多为4n。

空间性能

- 空间性能：$O(1)$
- 稳定性：不稳定





单选题 5分

设置

1. 每一趟简单选择排序只能确定一个记录的最终位置。

A 正确

B 错误

提交

单选题 5分

设置

2. 无论待排序序列的初始状态如何，简单选择排序都执行n-1趟。

A 正确

B 错误

提交

单选题 5分

设置

3. 简单选择排序是稳定的排序方法。

A 正确

B 错误

提交

单选题 5分

设置

4. 相对于其他基于比较的内排序，简单选择排序记录的比较次数较多，但是移动次数较少。

A 正确

B 错误

提交

主观题 10分

设置

5. 对于待排序记录序列{25, 30, 18, 10, 15, 35}，给出简单选择每一趟的结果。

正常使用主观题需2.0以上版本雨课堂

作答

单选题 5分

设置

1. 设有键值序列（k1, k2, ..., kn），当i>n/2时，任何一个子序列（ki, ki+1,... , kn）一定是堆。

A 正确

B 错误

提交

单选题 5分

设置

2. 在大根堆中，最小值结点一定是叶子结点。

A 正确

B 错误

提交

单选题 5分

设置

3. 在大根堆中，某结点的值一定大于其所有子孙结点的值。

A 正确

B 错误

注，这题不严谨，大根堆，根结点可以大于等于子孙结点，极端情况堆中所有元素都相等，但这种情况显然不建议用堆排序。

提交

单选题 5分

设置

4. 堆排序执行的趟数取决于待排序序列的初始状态。

A 正确

B 错误

提交

单选题 5分

设置

5. 堆排序所需的时间与待排序的记录个数无关。

A 正确

B 错误

提交

单选题 5分

设置

6. 堆排序将整个待排序序列划分为无序区和有序区，每一趟的关键是将无序区调整为堆。

A 正确

B 错误

提交

单选题 5分

设置

7. 对于堆排序算法，初始建堆的时间性能是（ ）。

A $O(n)$

B $O(n^2)$

C $O(n\log_2 n)$

D $O(\log_2 n)$

注，这题不严谨，采用教材196，算法6-5和6-6初始建堆分别可达n和nlog2n。

提交

主观题 10分

设置

8. 对于待排序记录序列{10, 25, 15, 18, 35, 20, 30, 12, 20*}，写出初始建堆的结果（写出结果序列并画出大根堆）

正常使用主观题需2.0以上版本雨课堂

作答