Data Structures

Ch8

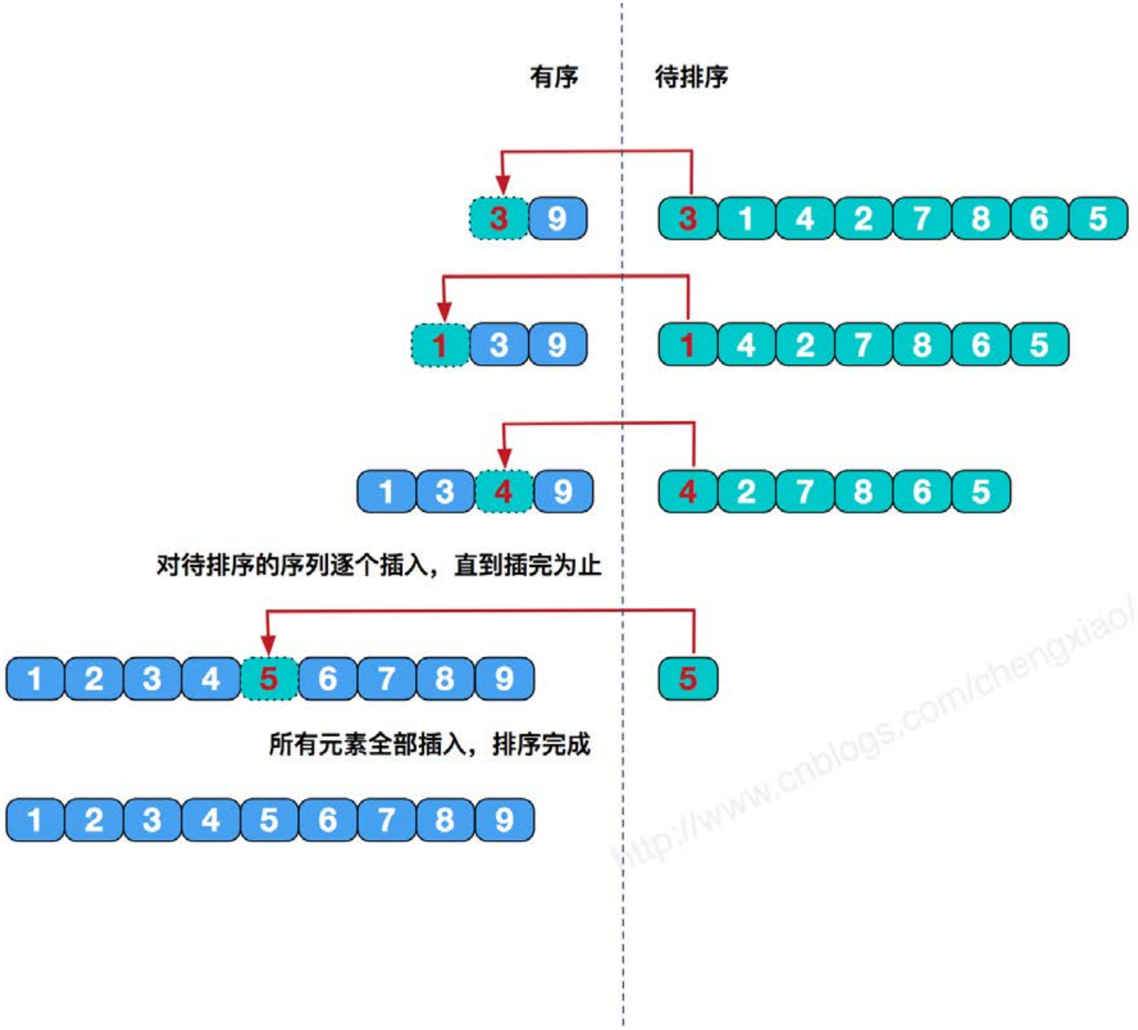
排序
Sort

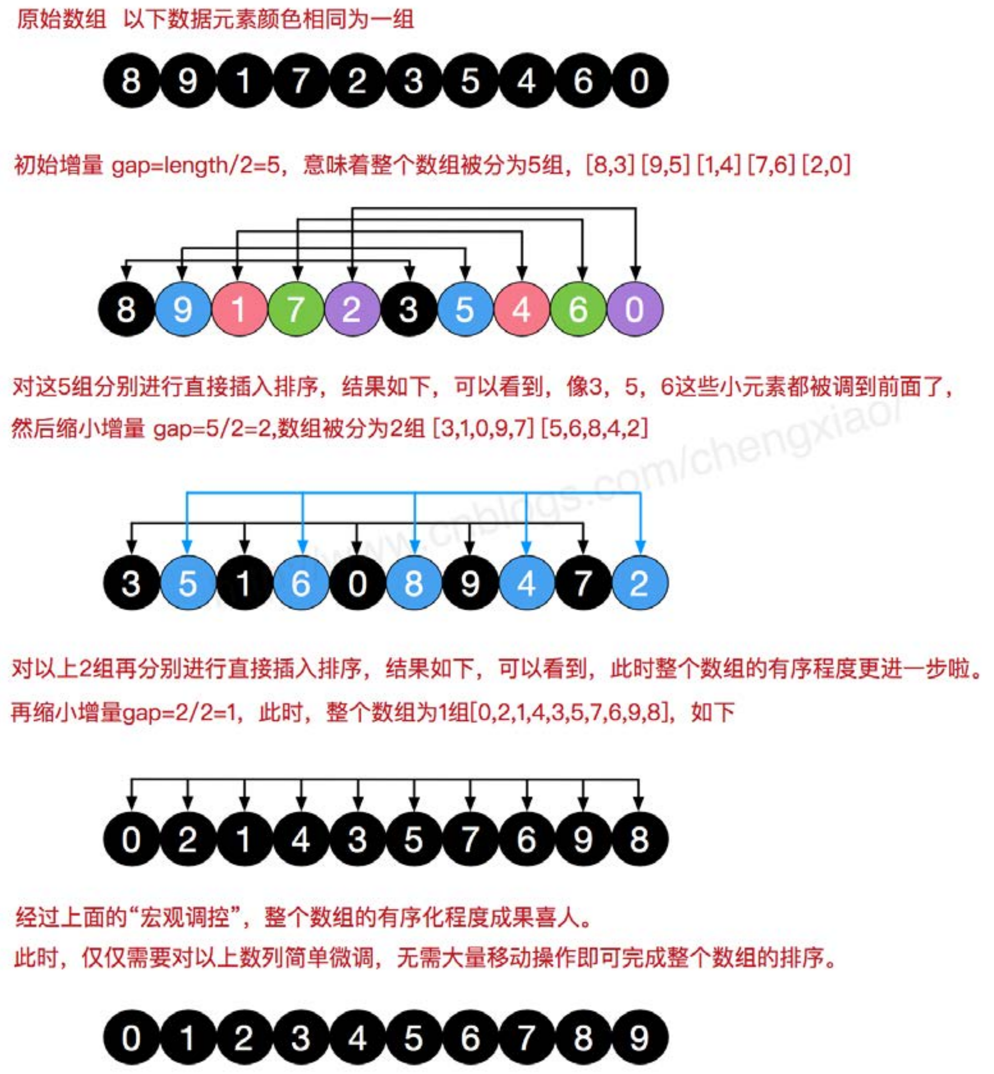
2023 年 12 月 14 日

Chapter 8 排序

- ☞ 8.1 概述
- ☞ 8.2 插入排序：直接插入排序、希尔排序
- ☞ 8.3 交换排序：起泡、快速排序
- ☞ 8.4 选择排序：简单选择、堆排序
- ☞ 8.5 归并排序：二路归并排序
- ☞ 8.6 各种排序方法比较
- ☞ 8.7 扩展与提高

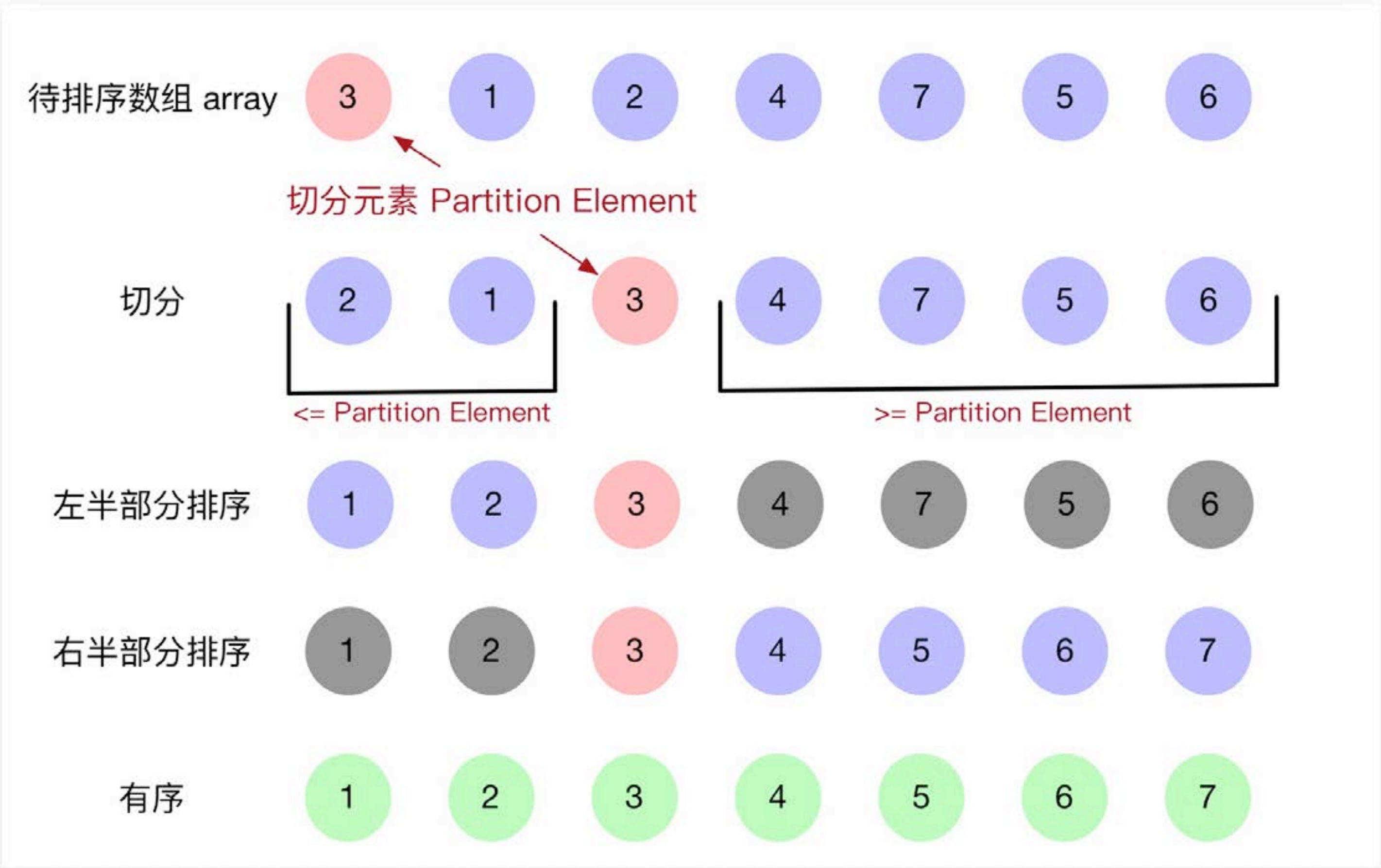
回顾





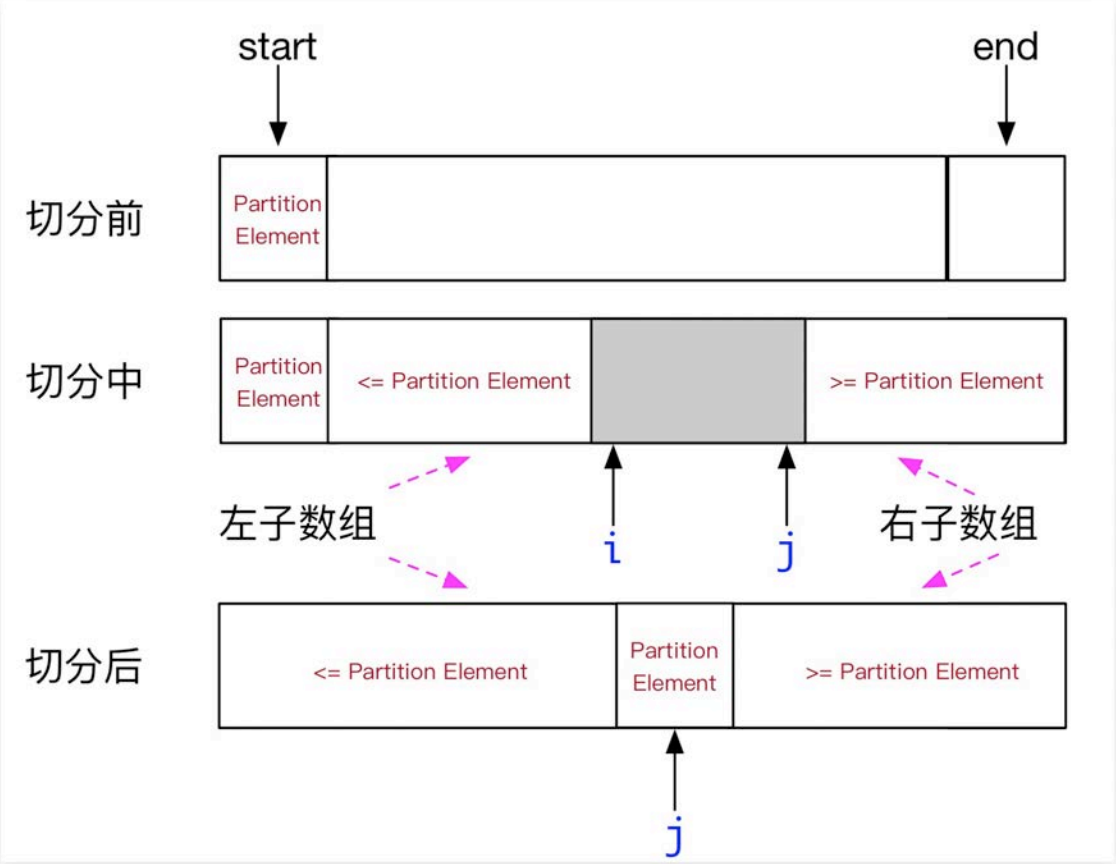
https://www.cnblogs.com/chengxiao/p/6103002.html

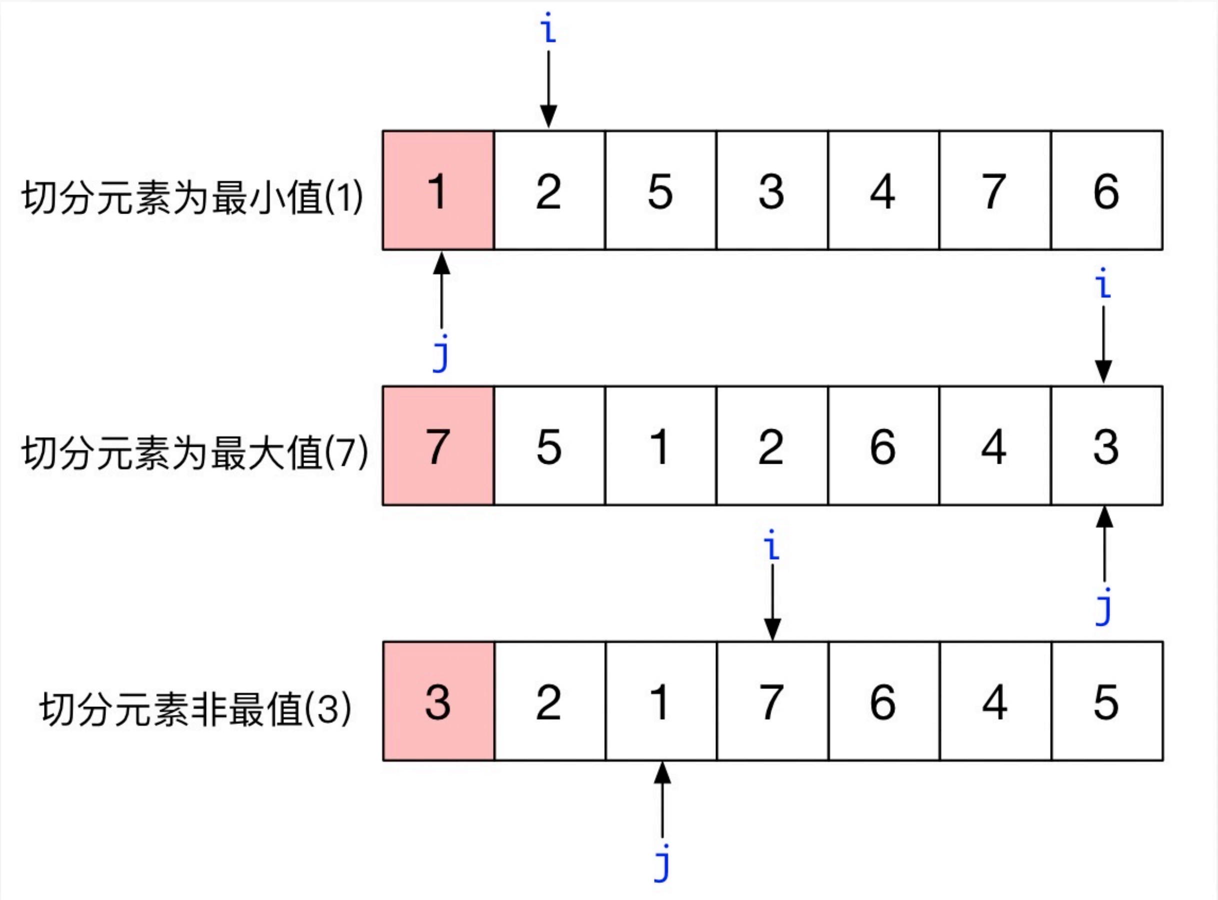
回顾



https://www.cnblogs.com/chengxiao/p/6103002.html

回顾


start
end
切分前
Partition Element
切分中
Partition Element
<= Partition Element
>= Partition Element
左子数组
右子数组
i
j
切分后
<= Partition Element
Partition Element
>= Partition Element
j

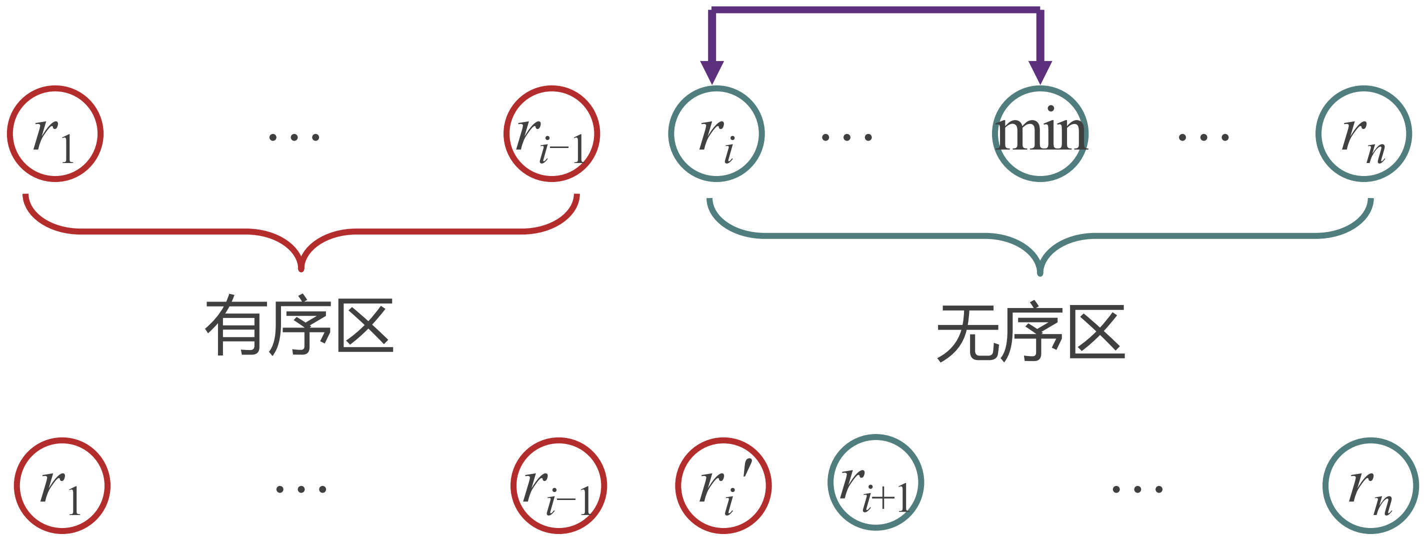

i
切分元素为最小值(1)
1 2 5 3 4 7 6
j
i
切分元素为最大值(7)
7 5 1 2 6 4 3
i
j
切分元素非最值(3)
3 2 1 7 6 4 5
j

8.4 选择排序

8-4-1　简单选择排序

1. 简单选择排序

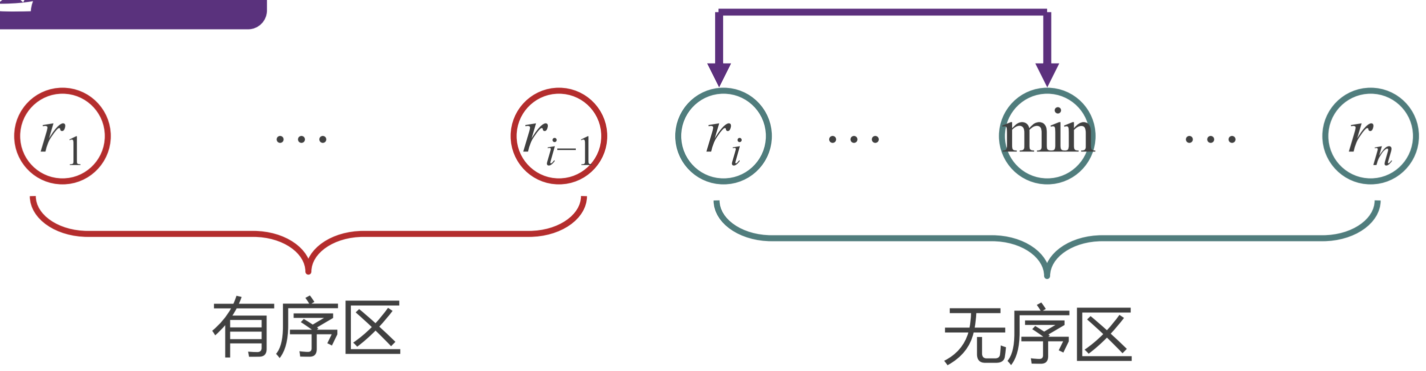
简单选择排序的基本思想：第 i 趟（$1 \le i \le n-1$）排序在待排序序列 r[i]～r[n] 中选取最小记录，并和第 i 个记录交换。



1. 简单选择排序

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

2. 关键问题

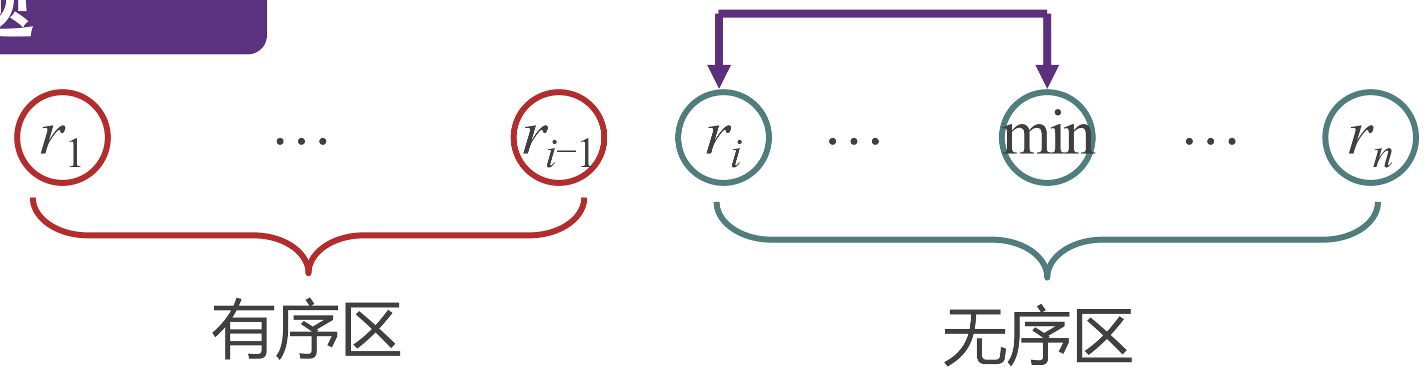


算法描述：

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

简单选择排序进行多少趟？ $n-1$趟

2. 关键问题



算法描述:

```
index = i;
for (j = i + 1; j < length; j++)
    if (data[j] < data[index]) index = j;
if (index != i) {
    交换data[i]和data[index];
}
```

第 i 趟简单选择排序完成什么工作?

（1）在r[i]~r[n]中找最小值

（2）将最小记录与r[i]交换

3. 算法描述

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

交换语句之前的判断与效率有什么关系？

4. 时间性能分析

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

比较语句？执行次数？

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

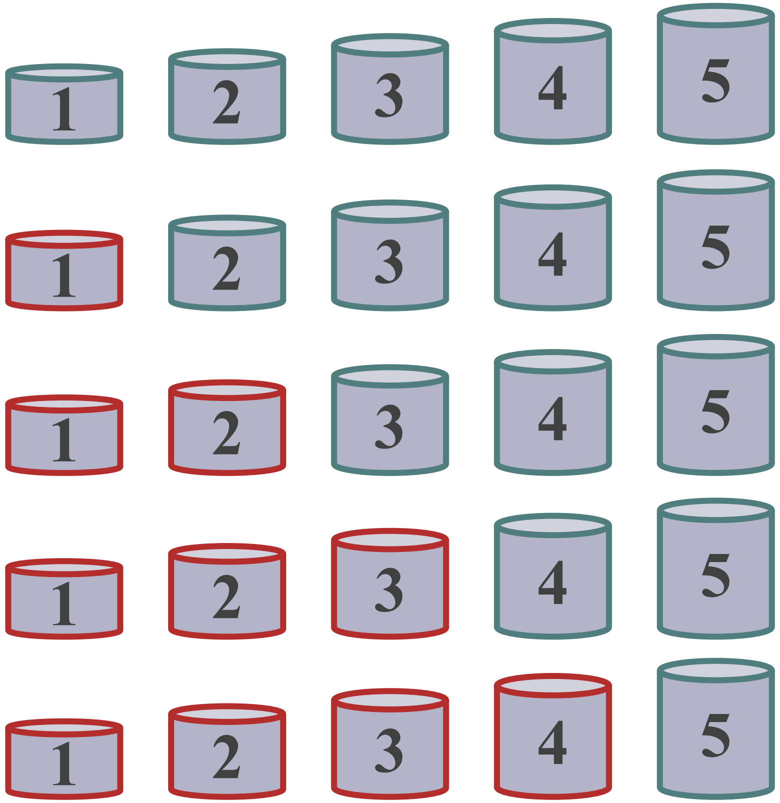
移动语句？执行次数？

4. 时间性能分析

- 比较次数： $O(n^2)$

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

- 移动次数：
  - 最好情况：0 次

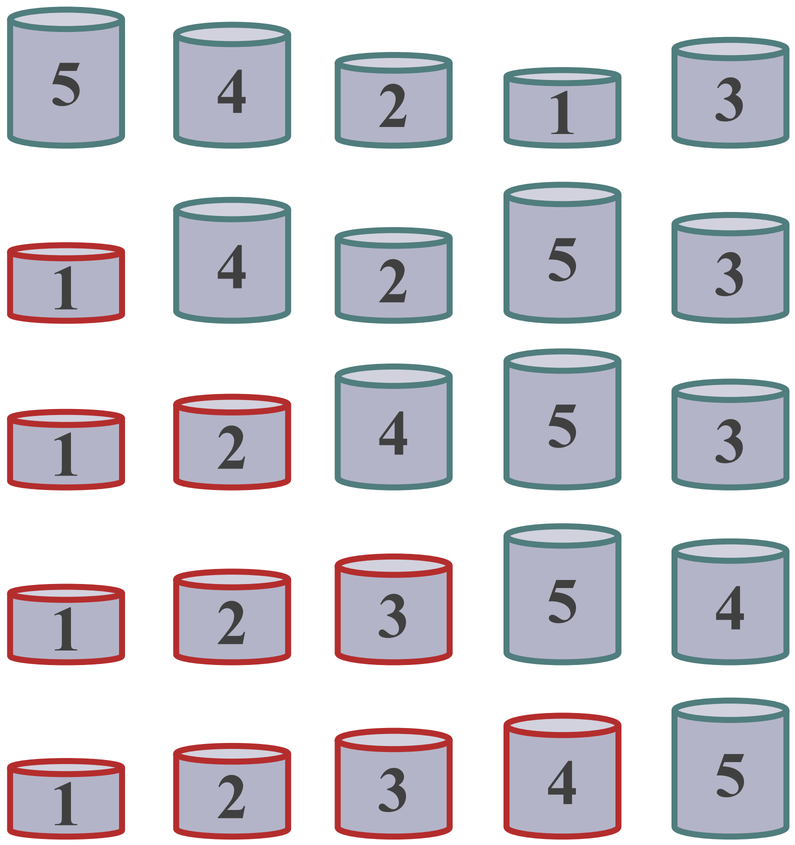


4. 时间性能分析

- 比较次数： $O(n^2)$

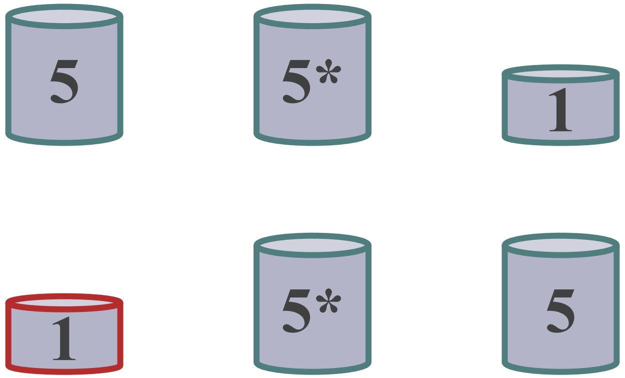
$$\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}1=(n-1+\cdots+2+1)=\frac{n(n-1)}{2}$$

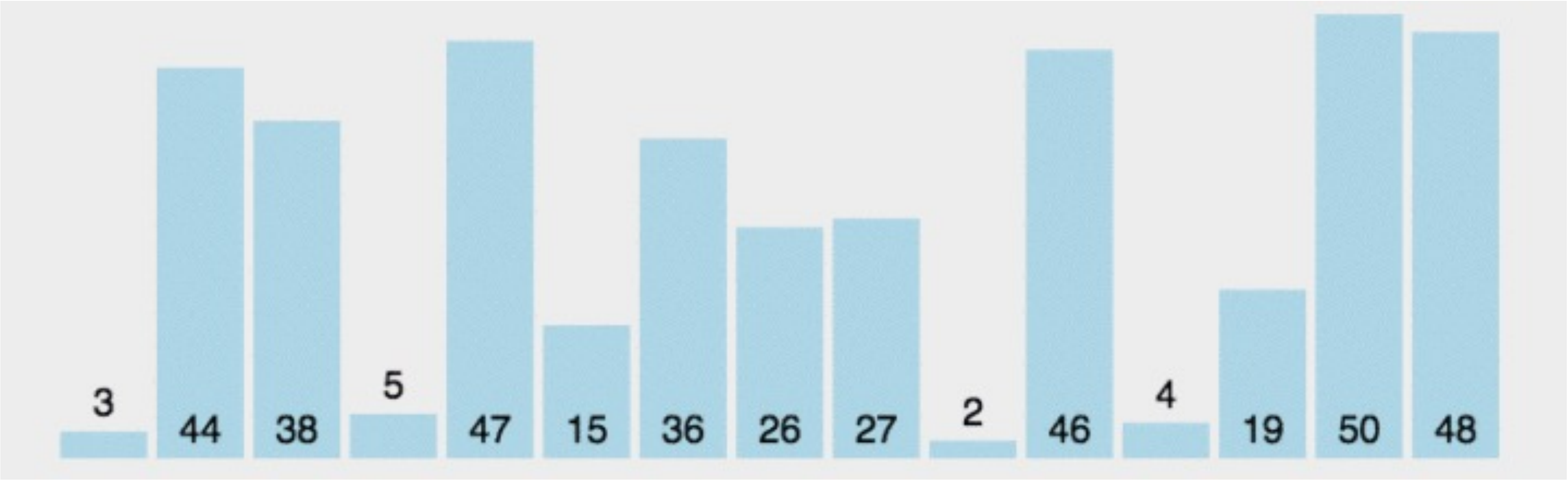
- 移动次数：
  - 最好情况：0 次
  - 最坏情况：$3(n-1)$次
- 最好、最坏、平均情况： $O(n^2)$



5. 空间性能分析

- 空间性能：$O(1)$
- 稳定性：不稳定

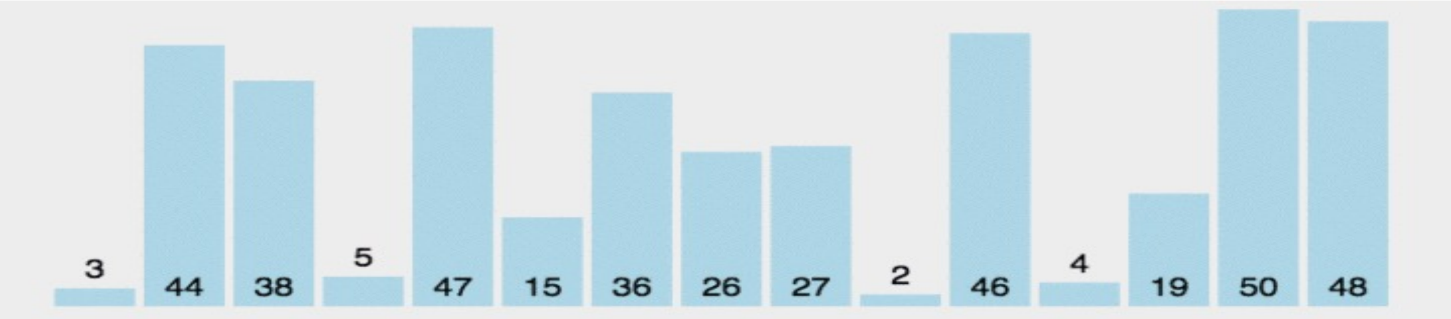




8.4 选择排序

8-4-2 堆排序 Heap Sort

简单选择排序



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$O(n)$

提高整个排序的效率 ⇐ 减少后面选择所用的比较次数

⇑

利用每趟比较后的结果 ⇒ 查找最小值的同时找出并保存较小值

1. 堆的定义

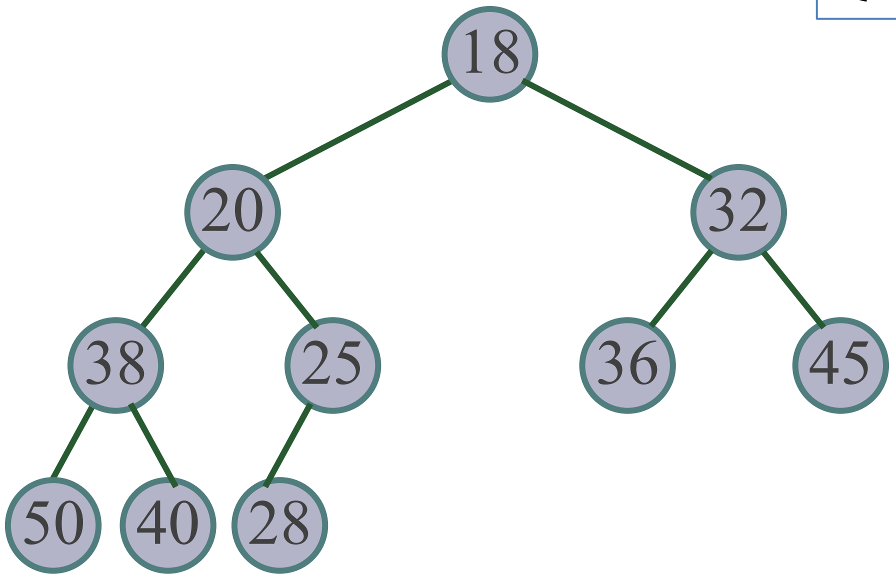
n个元素的序列{$k_1,k_2,\ldots,k_n$}，当且仅当满足下列关系时，称为堆：

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**。

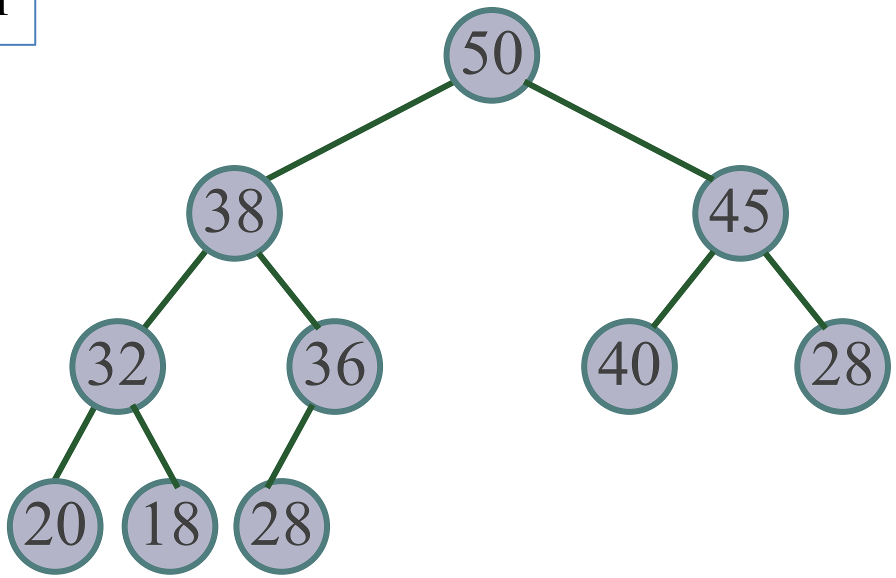
$$\begin{cases} k_i \le k_{2i} \\ k_i \le k_{2i+1} \end{cases} \text{或} \begin{cases} k_i \ge k_{2i} \\ k_i \ge k_{2i+1} \end{cases}$$

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**。

小根堆和大根堆统称为堆。



（1）根结点（称为**堆顶**）的值是所有结点的最大值；



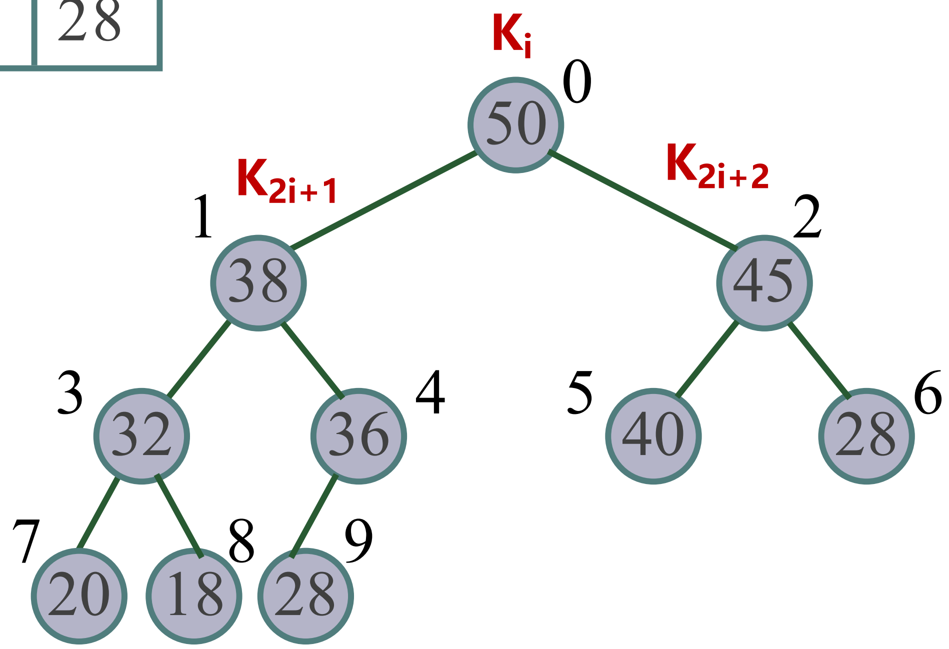
（2）较大值的结点靠近根结点，但不绝对。

2. 堆与序列的关系

堆采用顺序存储，则对应一个（**无序**）序列

0	1	2	3	4	5	6	7	8	9
50	38	45	32	36	40	28	20	18	28

顺序存储，以编号作为下标



3. 堆排序的基本思想

如何建?

如何调整?

基本思想:

- ✓将无序序列**建成**一个堆
- ✓输出**堆顶**的最小(大)值
- ✓使剩余的n-1个元素又**调整**成一个堆,则可得到n个元素的次小值
- ✓**重复**执行,得到一个有序序列

$$\underbrace{r_1 \quad \cdots \quad r_i}_{\text{无序区,调整为大根堆}} \quad \underbrace{r_{n-i+1} \quad \cdots \quad r_n}_{\text{有序区}}$$

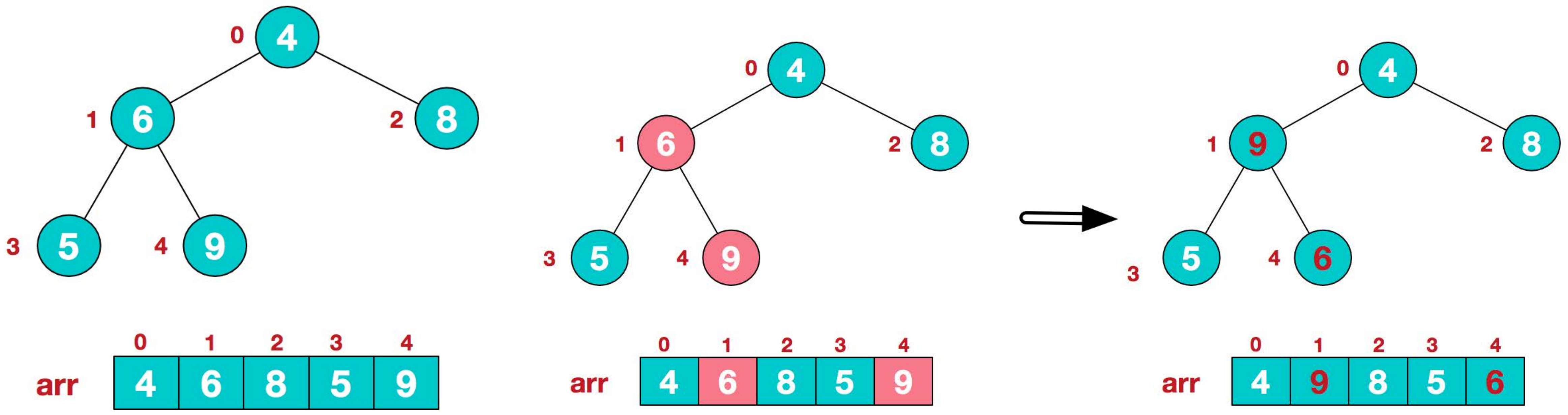
第 i 趟堆排序将 $r_1 \sim r_i$ 调整成大根堆,再将堆顶与 r_i 交换

3.1 构造初始堆

步骤一 构造初始堆。将给定无序序列构造成一个大顶堆（一般升序采用大顶堆，降序采用小顶堆)。

Step1：从最后一个非叶子结点开始从下至上进行调整

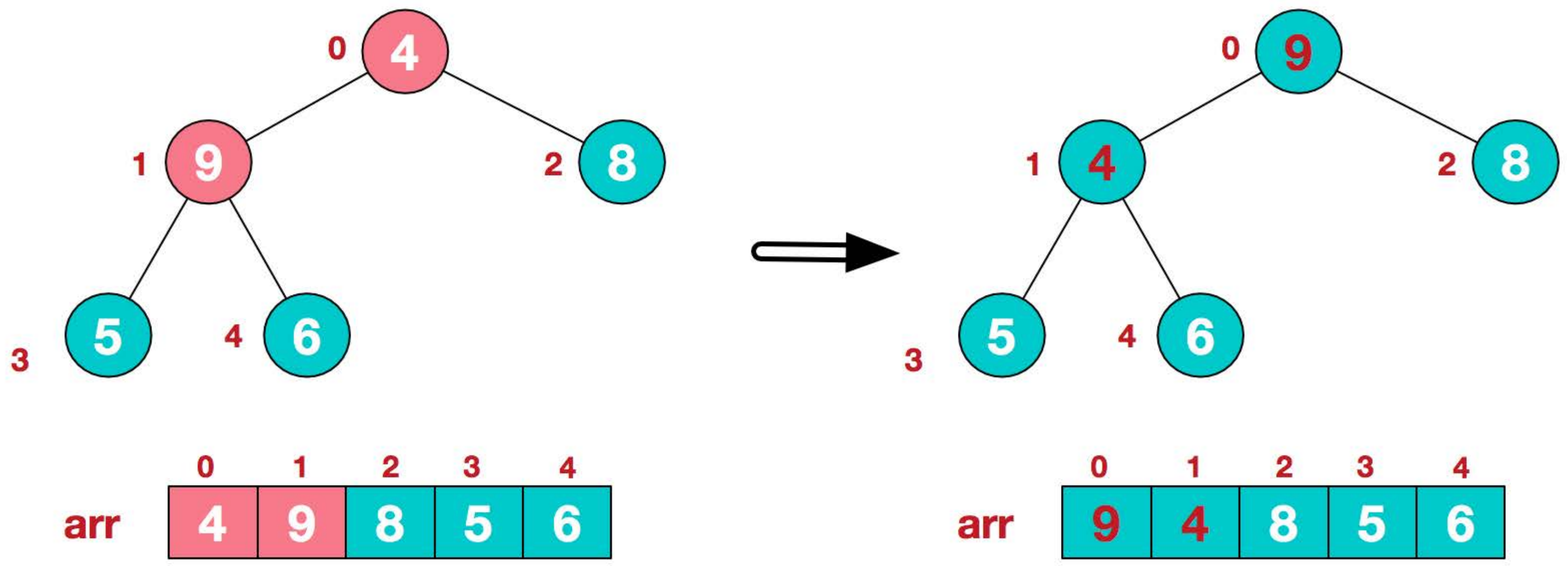
给定无序序列



3.1 构造初始堆

步骤一 构造初始堆。将给定无序序列构造成一个大顶堆（一般升序采用大顶堆，降序采用小顶堆)。

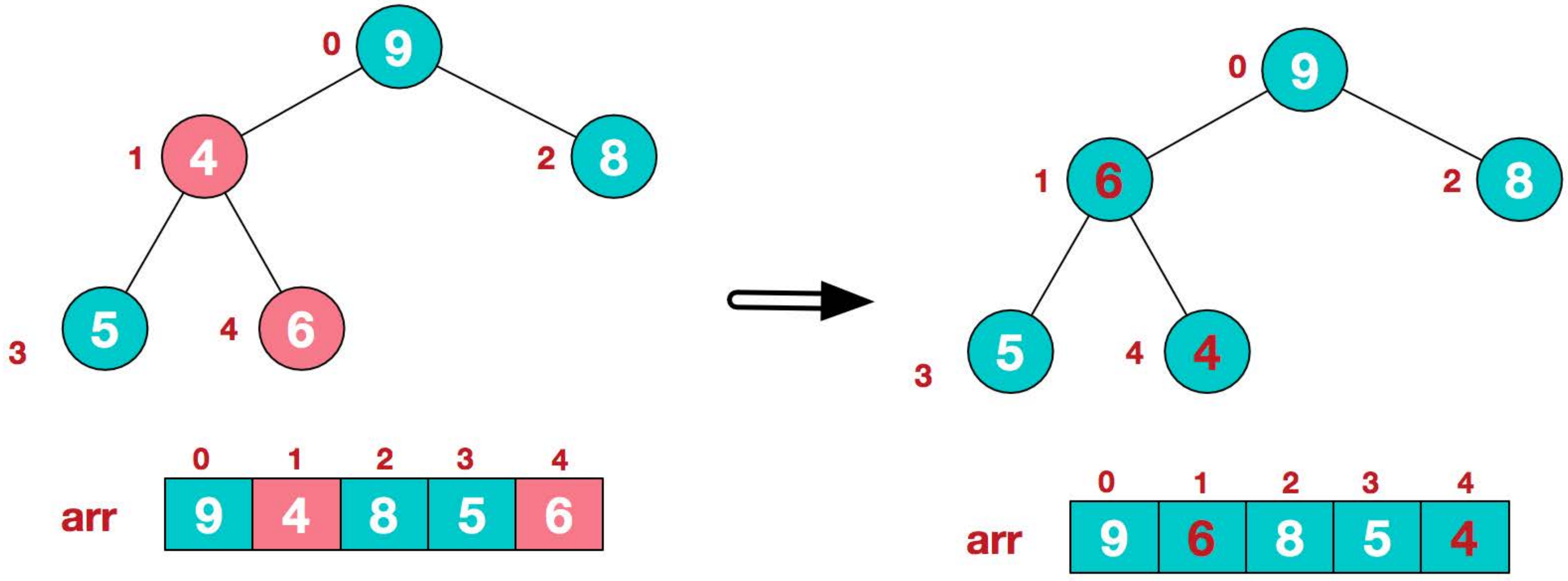
Step2： 找到第二个非叶节点4，由于[4,9,8]中9元素最大，4和9交换。



3.1 构造初始堆

步骤一 构造初始堆。将给定无序序列构造成一个大顶堆（一般升序采用大顶堆，降序采用小顶堆)。

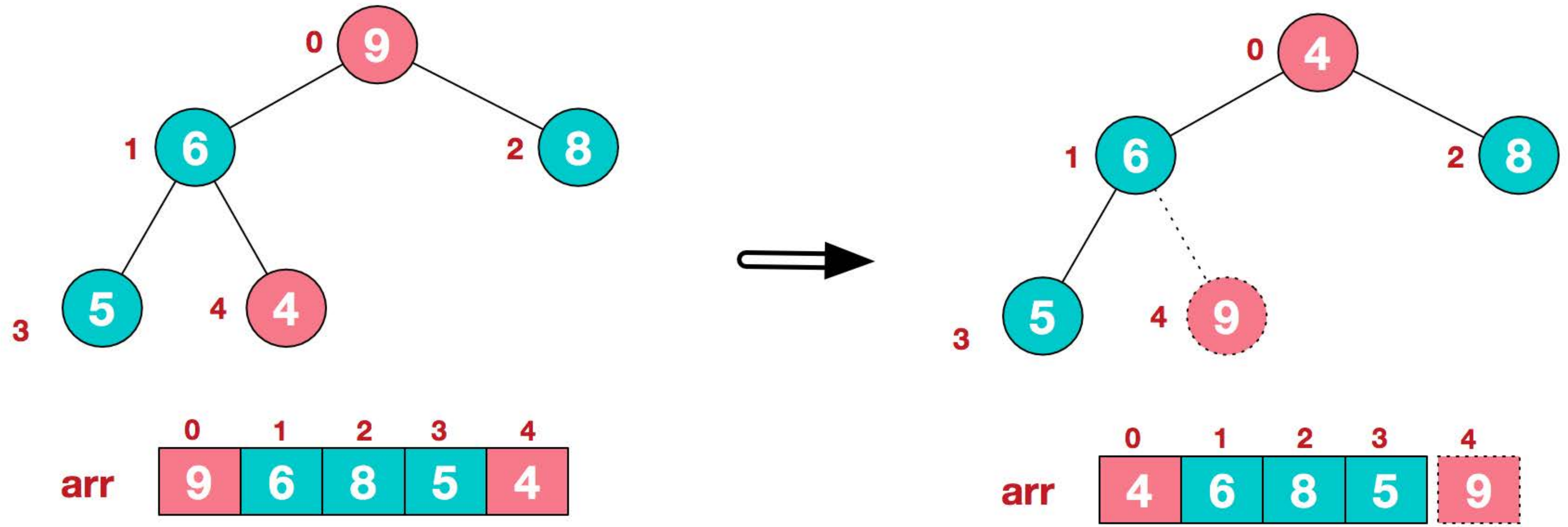
Step3：交换导致了子根[4,5,6]结构混乱，继续调整，[4,5,6]中6最大，交换4和6。



3.2 堆调整

步骤二 将堆顶元素与末尾元素进行交换，使末尾元素最大。然后继续调整堆，再将堆顶元素与末尾元素交换，得到第二大元素。如此反复进行交换、重建、交换。

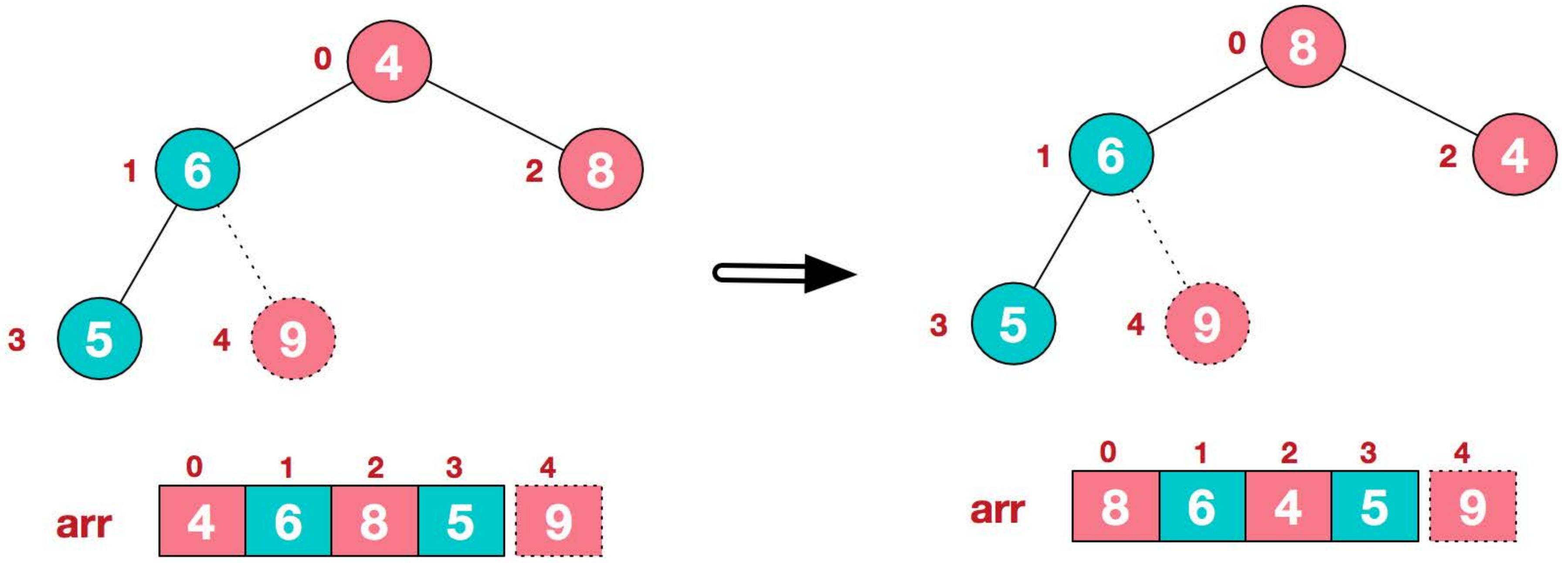
Step1：将堆顶元素9和末尾元素4进行交换



3.2 堆调整

步骤二 将堆顶元素与末尾元素进行交换，使末尾元素最大。然后继续调整堆，再将堆顶元素与末尾元素交换，得到第二大元素。如此反复进行交换、重建、交换。

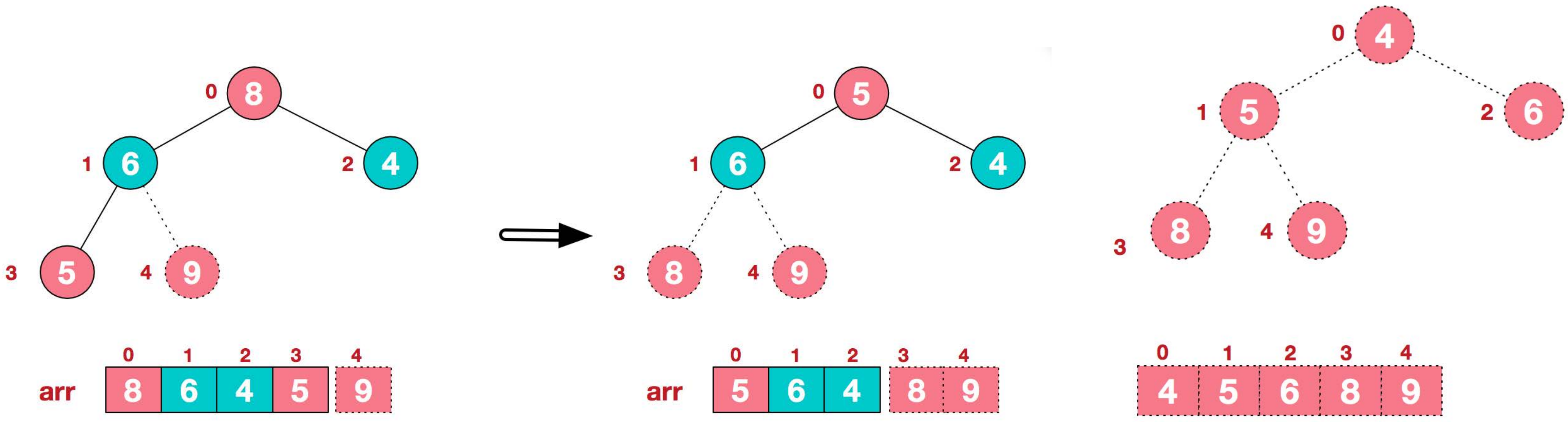
Step2：重新调整结构，使其继续满足堆定义



3.2 堆调整

步骤二 将堆顶元素与末尾元素进行交换，使末尾元素最大。然后继续调整堆，再将堆顶元素与末尾元素交换，得到第二大元素。如此反复进行交换、重建、交换。

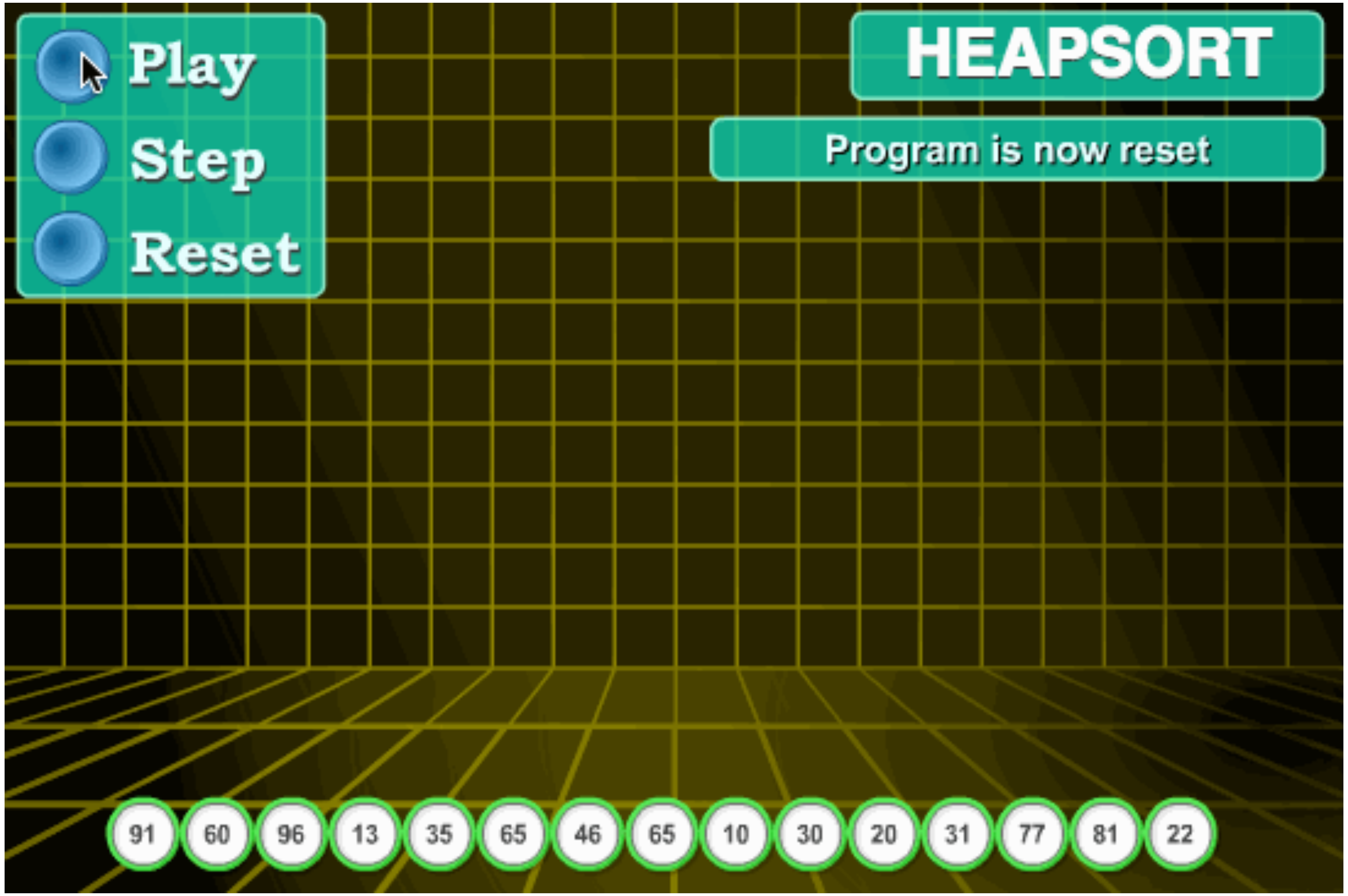
Step3：再将堆顶元素8与末尾元素5进行交换，得到第二大元素8.



3. 堆排序的基本思想

基本思想：

✓将无序序列**建成**一个堆
✓输出**堆顶**的最小（大）值
✓使剩余的n-1个元素又**调整**成一个堆，则可得到n个元素的次小值
✓**重复**执行，得到一个有序序列



https://zhuanlan.zhihu.com/p/34644389

4 堆调整的实现

堆调整：在一棵完全二叉树中，根结点的左右子树均是堆，调整根结点使整个完全二叉树成为一个堆的过程。

如何设计函数接口？

由于初始建堆和重建堆均调用此函数，因此，设置形参 k 和 last

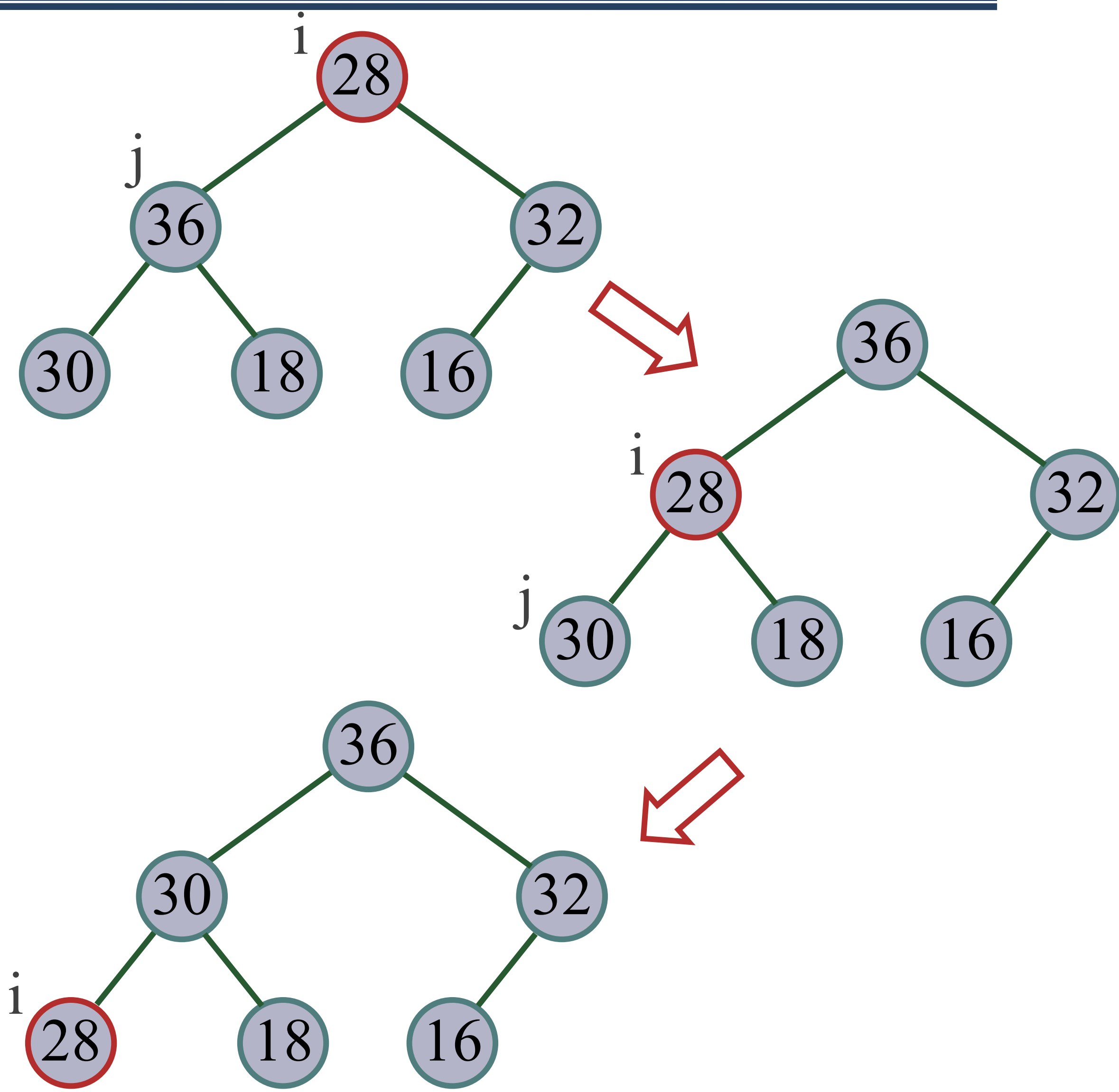
```
void Sort :: Sift(int k, int last) //根结点的编号为k，最后一个结点的编号为last
{



}
```

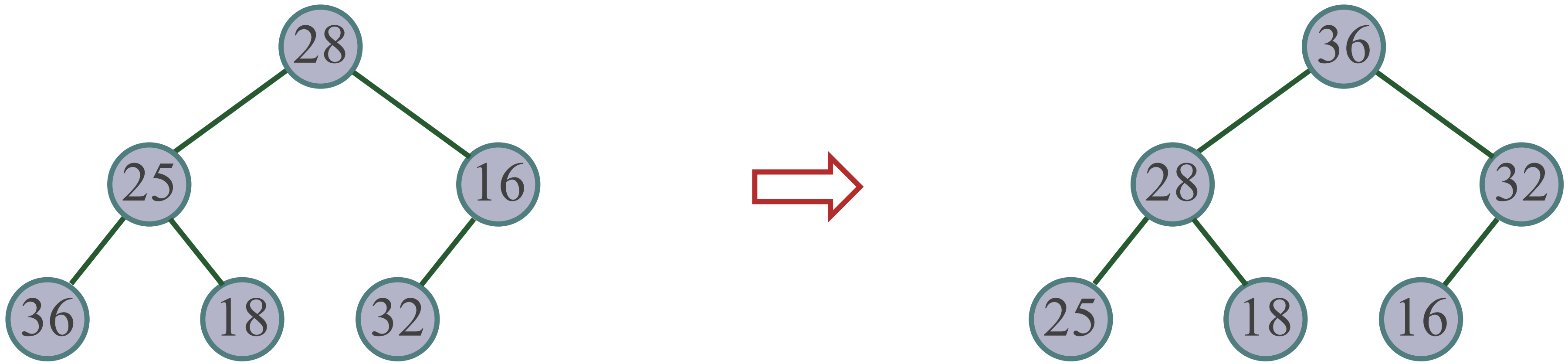
4 堆调整的实现

```
void Sort :: Sift(int k, int last)
{
    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
    {
        // j指向左右孩子的较大者
        if (j < last && data[j] < data[j+1]) j++;
        if (data[i] > data[j]) break;  //已经是堆
        else {
            temp = data[i];  data[i] = data[j];  data[j] = temp;
            //被调整结点位于结点j的位置
            i = j; j = 2 * i + 1;
        }
    }
}
```



5 初始建堆的实现

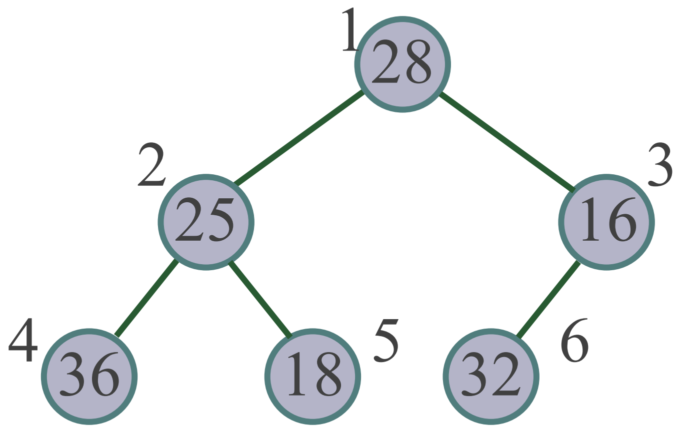
待排序序列 {28, 25, 16, 36, 18, 32} ⇨ 初始建堆结果 {36, 28, 32, 25, 18, 16}



如何将一个无序序列建成一个大根堆——初始建堆?

5 初始建堆的实现

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

算法描述：

```
for (i = ceil(length/2) - 1; i >= 0; i--)
    Sift(i, length-1);              //调整结点 i
```

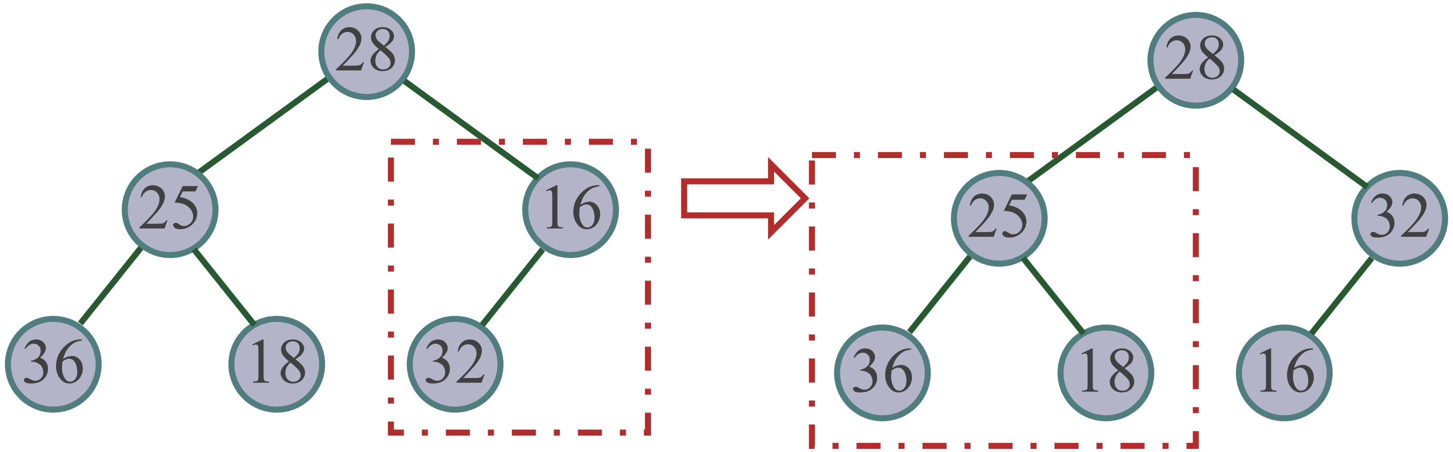
```
void Sort :: Sift (int k, int last)
//根结点的编号为 k，最后一个结点的编号为 last
```

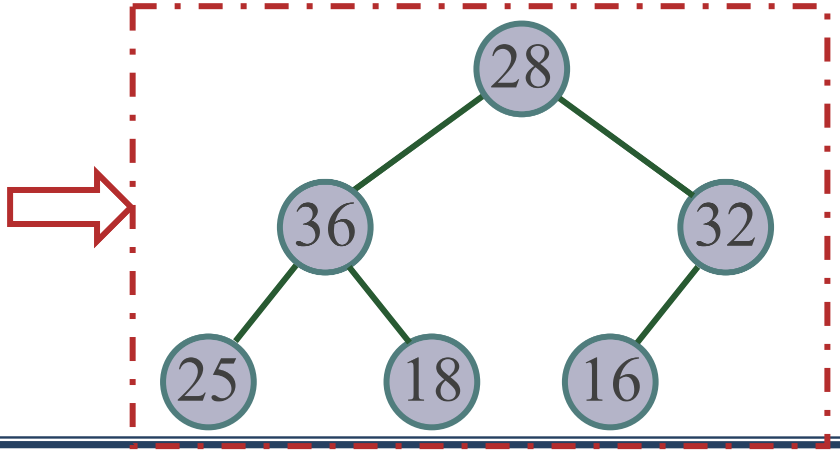
需要调整叶子结点吗？分支结点中编号最大的是多少？

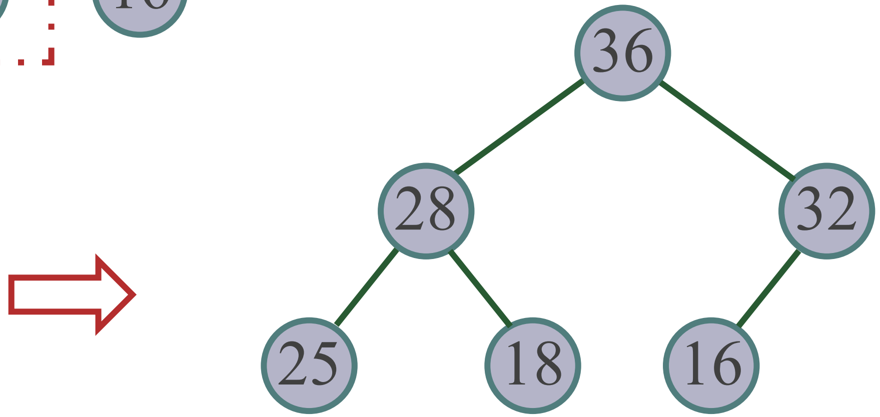
5 初始建堆的实现

待排序序列 {28, 25, 16, 36, 18, 32}

```
for (i = ceil(length/2) - 1; i >= 0; i--)
    Sift(i, length-1);                 //调整结点 i
```







初始建堆结果 {36, 28, 32, 25, 18, 16}

6 堆排序算法实现

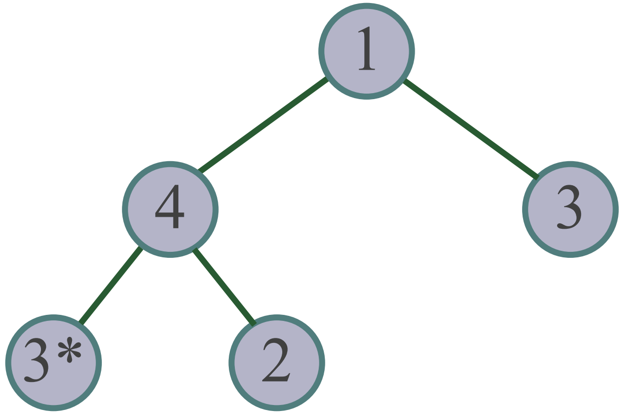
```
void Sort :: HeapSort( )
{
    int i, temp;
    //从最后一个分支结点至根结点
    for (i = ceil(length/2) - 1; i >= 0; i--)
        Sift(i, length-1) ;
    for (i = 1; i < length; i++)
    {
        temp = data[0]; data[0] = data[length-i]; data[length-i] = temp;
        Sift(0, length-i-1); //重建堆
    }
}
```

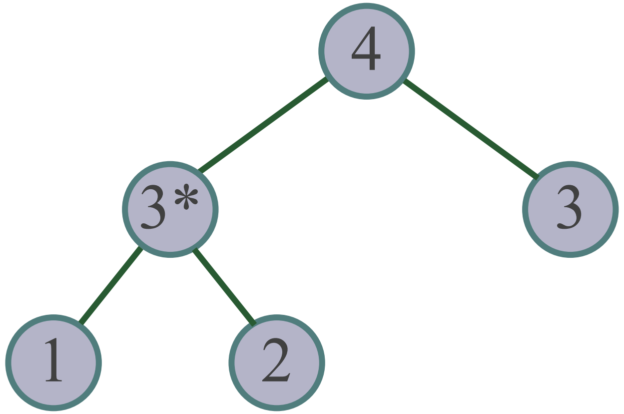
- 初始建堆： $O(n\log_2 n)$
- 重建堆次数： $n-1$
- 重建堆： $O(\log_2 i)$
- 最好、最坏、平均情况： $O(n\log_2 n)$

7 空间性能分析

待排序序列

初始建堆





- 空间性能：$O(1)$
- 稳定性：不稳定

运行实例2

待排序序列

初始建堆

交换r[0]和r[4]

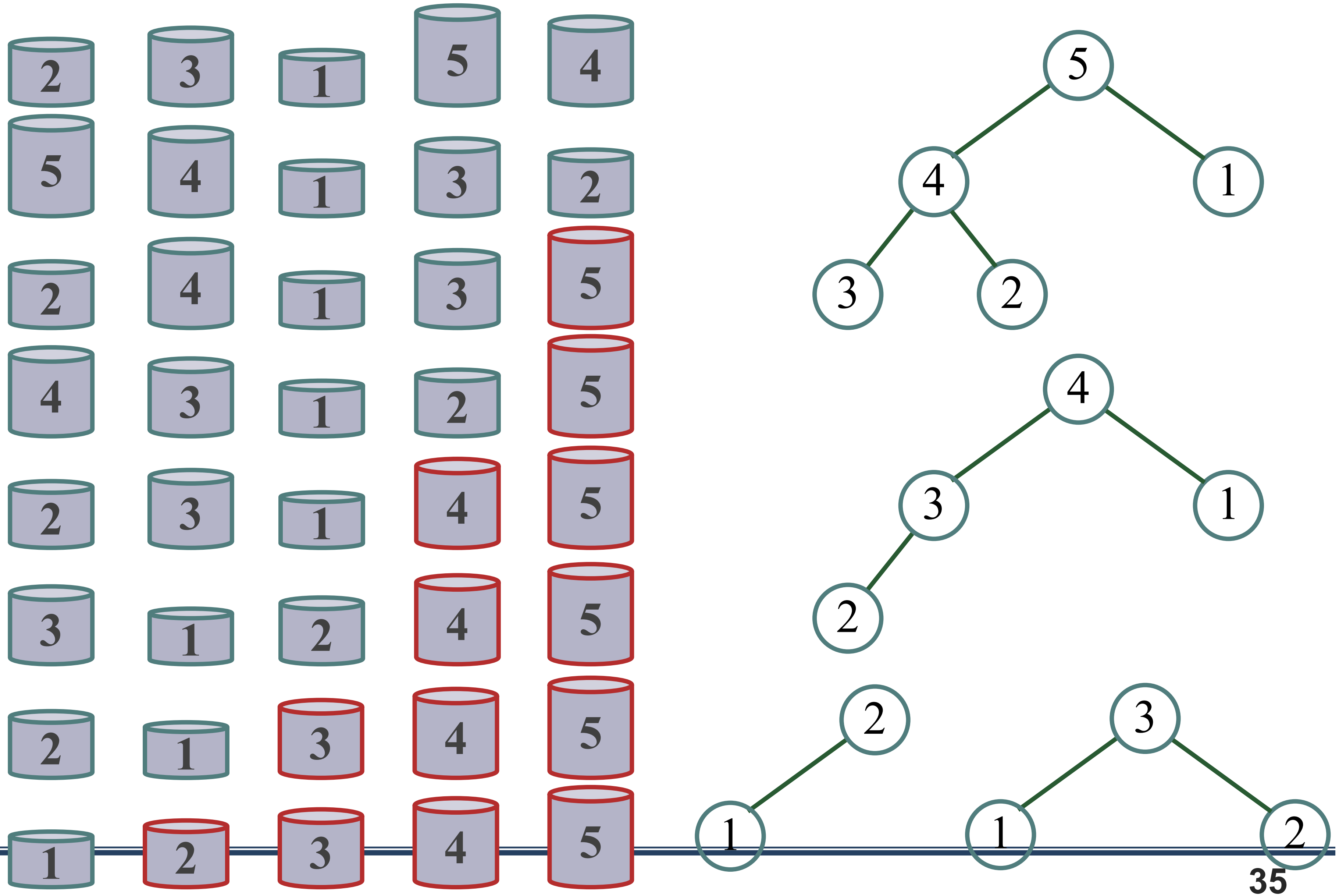
无序区重建堆

交换r[0]和r[3]

无序区重建堆

交换r[0]和r[2]

最后一趟



小结

1. 掌握**简单选择排序**算法及实现
2. 掌握**堆排序**算法及实现

作业

1．已知关键字序列(3, 26, 38, 5, 47, 15, 36, 26*, 2, 4, 19, 50)，使用直接插入排序、希尔排序、起泡排序、快速排序、简单选择排序、堆排序、二路归并排序七种排序算法进行排序，请分别给出七种排序算法每一趟排序的结果，并给出时间和空间复杂度量级。

Thank You !

Q & A

本文一共总结了10种排序算法，其中

基于比较的排序算法有

冒泡排序，插入排序，希尔排序，选择排序，归并排序，堆排序，快速排序；

线性时间排序算法包括

计数排序，基数排序，桶排序；

前边有提到过，基于比较的排序算法，时间复杂度最差达到O(nlogn)O(nlogn)，无法突破这个界限，只有线性时间排序能够突破，达到O(n)O(n)，所以说，如果满足了线性时间排序算法的限制条件，使用线性时间排序将会使排序性能得到极大提升。

二、实际测试数据

下面对以上涉及到的每种算法做一个简单的实际测试对比：利用随机数，随机生成区间0 ~ K之间的序列，共计N个数字，利用各种算法进行排序，记录排序所需时间，测试环境为i7+vs2015+Debug版本。

算法\输入数据	N=50 K=50	N=200 K=100	N=500 K=500	N=2000 K=2000	N=5000 K=8000	N=10000 K=20000	N=20000 K=20000	N=20000 K=200000
冒泡排序	0ms	15ms	89ms	1493ms	9363ms	36951ms	147817ms	143457ms
插入排序	1ms	13ms	82ms	1402ms	8698ms	34731ms	134817ms	134836ms
希尔排序	0ms	1ms	6ms	30ms	110ms	257ms	599ms	606ms
选择排序	0ms	5ms	31ms	461ms	2888ms	11736ms	45308ms	44838ms
堆排序	0ms	3ms	9ms	40ms	124ms	247ms	525ms	527ms
归并排序	2ms	6ms	18ms	75ms	199ms	392ms	778ms	793ms
快速排序	0ms	1ms	2ms	14ms	36ms	84ms	196ms	163ms
计数排序	0ms	1ms	1ms	5ms	15ms	32ms	51ms	62ms
基数排序	0ms	1ms	4ms	19ms	47ms	114ms	237ms	226ms
桶排序	0ms	2ms	6ms	25ms	68ms	126ms	254ms	251ms

三、性能对比小结

1. 传统简单排序确实当数据量很小的时候也表现不错，但当数据量增大，其耗时也增大十分明显；
2. 冒泡，插入，选择三种排序中，当数据量很大时，选择排序性能会更好；
3. 堆排，希尔，归并，快排几种排序算法也表现不错，源于其时间复杂度达到了O(nlogn)O(nlogn)；
4. 随机快速排序性能确实表现十分亮眼，甚至有时比基数排序和桶排序还好，这可能也是快排如此流行的原因；
5. 线性排序中计数排序表现最好，但他们的限制也比较明显，只能处理范围内的正整数。

https://www.cnblogs.com/luoahong/p/9685904.html

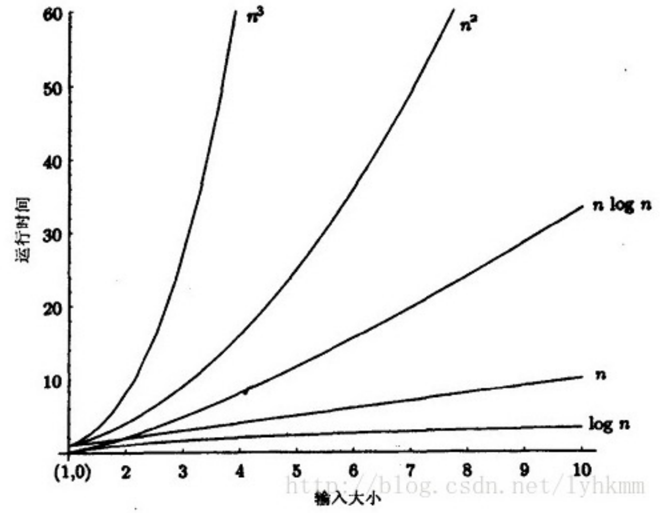
时间复杂度

类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均情况	最好情况	最坏情况	辅助存储	
插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	Shell排序	$O(n^{1.3})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定
选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定
交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	不稳定
归并排序		$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定
基数排序		$O(d(r+n))$	$O(d(n+rd))$	$O(d(r+n))$	$O(rd+n)$	稳定

注：基数排序的复杂度中，r代表关键字的基数，d代表长度，n代表关键字的个数。

http://blog.csdn.net/lyhkmm

复杂度函数时间大小比较



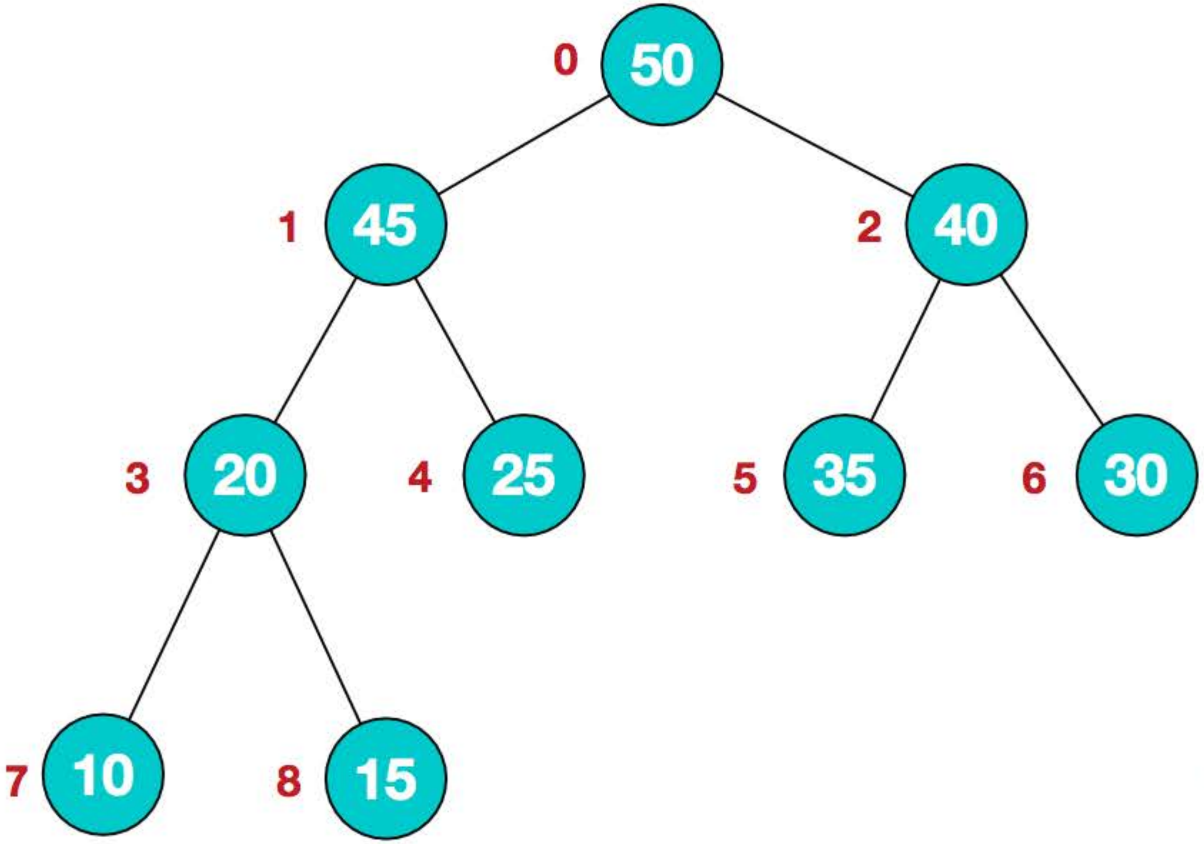
http://blog.csdn.net/lyhkmm

GitHub地址

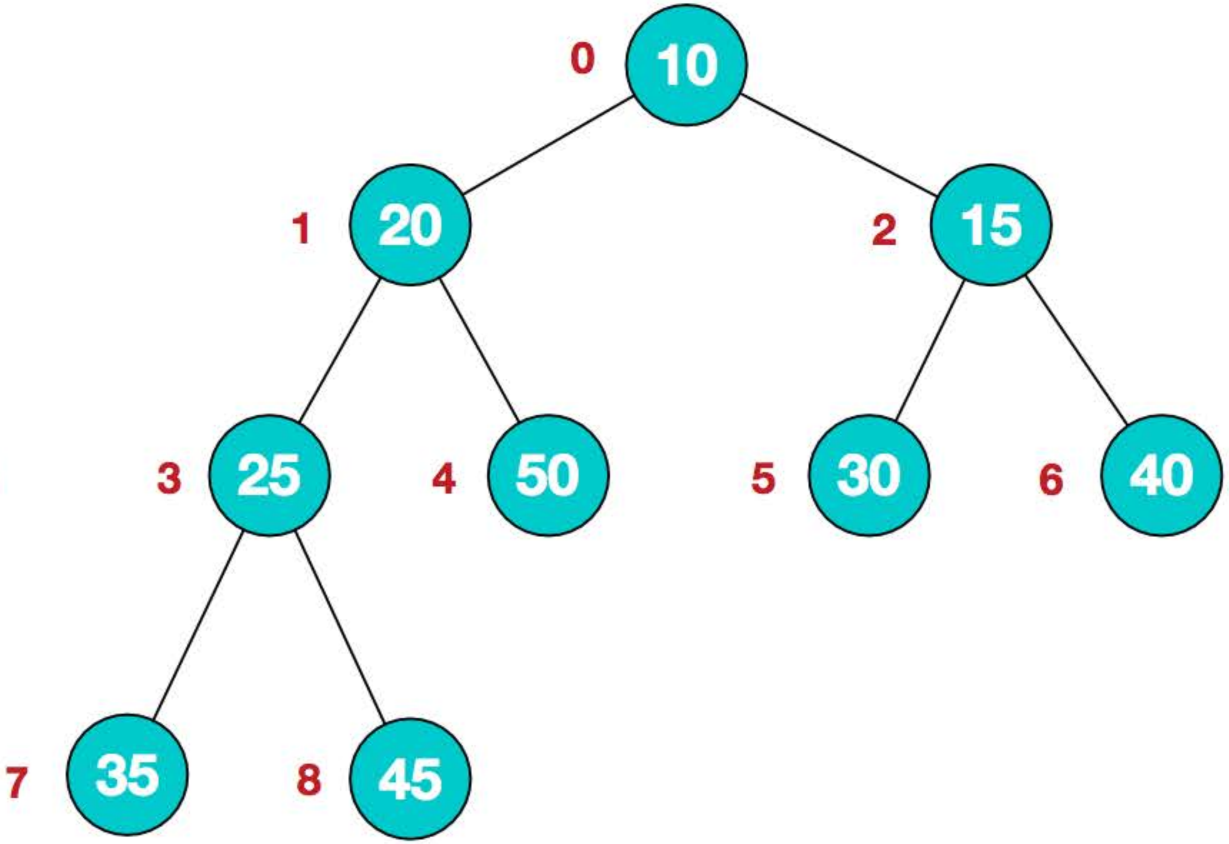
https://github.com/lyhkmm/algorithm

https://blog.csdn.net/lyhkmm/article/details/78920769

大顶堆


0
50
1
45
2
40
3
20
4
25
5
35
6
30
7
10
8
15

小顶堆


0
10
1
20
2
15
3
25
4
50
5
30
6
40
7
35
8
45