

第九章 排序

本章节目录

[9.1 排序的基本概念](#)

[9.2 插入排序](#)

[9.3 交换排序](#)


[9.4 选择排序](#)

[9.5 归并排序](#)

[9.6 基数排序](#)

9.1 排序的基本概念

- ❖ **排序**是将一组“无序”的记录序列，重新排列成一个**按关键字“有序”**的记录序列。
- ❖ 记录=数据元素
- ❖ 假设含 n 个记录的序列为 $\{R_1, R_2, \dots, R_n\}$ ，其相应的关键字序列为 $\{K_1, K_2, \dots, K_n\}$
- ❖ **递增排序**：将 n 个记录的序列重新排列为 $\{Rs_1, Rs_2, \dots, Rs_n\}$ ，使得 $K_{s_1} \leq K_{s_2} \leq \dots \leq K_{s_n}$ 。
- ❖ 若关键字 $K_i = K_j$ ($i \neq j$)，且在排序前的序列中 R_i 领先于 R_j 。经过排序后， R_i 与 R_j 的相对次序保持不变（即 R_i 仍领先于 R_j ），则称这种排序方法是**稳定的**，否则称之为**不稳定的**。

- 
- ❖ 排序的分类：内部排序与外部排序
 - ❖ 排序的方法：插入、交换、选择、归并、基数
 - ❖ 插入排序：直接插入排序和希尔排序
 - ❖ 交换排序：冒泡排序和快速排序
 - ❖ 选择排序：直接选择排序和堆排序
 - ❖ 归并
 - ❖ 基数



❖ 排序中涉及的操作：

❖ 比较

❖ 移动记录

❖ 排序中涉及的数据结构：

❖ 数组：比较和移动数据

❖ 链表：比较和修改指针，无需移动数据

❖ 待排序记录采用**顺序存储结构**，类型定义如下：

```
#define  n   30//待排序记录的个数

typedef struct

    { int key;

      InfoType otherinfo; // 记录其它数据域

    } RecType;

RecType R[n+1];
```


9.2 插入排序

- ❖ **插入排序的基本思想**是将第一个记录看作有序，从第2个记录开始，将待排序的记录插入到有序序列中，使有序序列逐渐扩大，直至所有记录都插入到有序序列中。
- ❖ 序列组成=**有序部分+无序部分**
- ❖ 步骤：找到位置（向后移动数据空出位置），插入数据
- ❖ **【例1】**有8个待排序的记录，其关键字分别为：52，35，68，96，85，17，25，52。对其进行直接插入排序。

9.2.1 直接插入排序

1. 基本思想

- (1) 有序子序列 $R[1.....i-1]$
- (2) 将记录 $R[i](2 \leq i \leq n)$ 插入到有序子序列 $R[1..i-1]$ 中，使记录的有序序列从 $R[1.....i-1]$ 变为 $R[1.....i]$ 。

2. 算法-----**递增顺序**，前面1.....i-1个数据已经排好，插入第i个数据

```
void StrOnePass(RecType R[], int i)
```

```
{ R[0]=R[i]; j=i-1; // 暂存待排序记录到R[0]
```

```
  x=R[0].key;
```

```
  // 从后向前查找插入位置，将大于待排序记录向后移动
```

```
  while (x< R[j].key)
```

```
    { R[j+1]=R[j]; j--; } // 记录后移 , 找到位置
```

```
  R[j+1]=R[ 0 ]; // 将待排序记录放到合适位置
```

```
} // StrOnePass
```



3. 效率分析

空间效率：仅用了一个辅助单元，辅助空间为 $O(1)$ 。

时间效率：向有序序列中逐个插入记录的操作，进行了 $n-1$ 趟，每趟操作分为比较关键字和移动记录。

当 n 很大时，其效率不高。

时间复杂度为 $O(n^2)$ 、最小为 $O(n)$ 。

适合场合：数量少，基本有序

特点：是一种**稳定的**排序方法

9.2.2 希尔排序

希尔排序又称**缩小增量排序**，它在时间效率上比直接插入排序有较大的改进，主要从**“减小记录个数”**和**“基本有序”**两方面着手。

1. 基本思想

- (1)将待排序的记录划分成**若干个子序列**，分别进行**直接插入排序**
- (2)再划分几个大的子序列，进行直接插入排序；**反复进行几遍**
- (3)最后作为一个完整的序列，实施**直接插入排序**

特点：当经过几次子序列的排序后，记录的排列已经基本有序

9.2.2 希尔排序

◆ 希尔排序不是将相邻记录分成一组子序列，而是将相隔一定距离 d 的记录分成一组子序列。

【例2】有10个待排序的记录，其关键字分别为：52，35，68，96，85，17，25，52，55，10。对其进行希尔排序，

第1趟排序 d 取5，第2趟排序 d 取3，第3趟采用直接插入排序。

52, 35, 68, 96, 85, 17, 25, 52, 55, 10

52 17 25

35 68 52 55

96 85 10

17, 25, 52, 55, 10, 52, 35, 68, 96, 85

17			55			35			85
	25			10			68		
		52			52			96	
17,	10,	52,	35,	25,	52,	55,	68,	96,	85
10,	17,	25,	35,	52,	55,	55,	68,	85,	96

特点：逆序的记录一个跳过多个记录，跳跃式移动

```
void ShellSort(RecType R[ ], int n)
```

```
{ // 以步长d/2分组，第一个步长取n/2，最后一个取1
```

```
for(d=n/2;d>=1;d=d/2)
```

```
{ for(i=1+d; i<=n; i=i++) //将R[i]插入到所属组的有序列中
```

```
{ R[0]=R[i]; j=i-d; //R[0]作为缓存，j指向前一个记录
```

```
while(j>0&&R[0].key<R[j].key)
```

```
{ R[j+d]=R[j]; //后移1个记录，空出当前位置
```

```
j=j-d; //查前1个记录
```

```
} // while
```

```
R[j+d]=R[0];
```

```
} // for i
```

```
} // for d
```

```
} // ShellSort
```

```
void StrOnePass(RecType R[ ], int i)
```

```
{ R[0]=R[i]; j=i-1;
```

```
x=R[0].key;
```

```
while (x< R[j].key)
```

```
{ R[j+1]=R[j];
```

```
j=j-1; }
```

```
R[j+1]=R[0];
```

```
} // StrOnePass
```

j i
R[0] R[1] R[i-2d]..... R[i-d]..... R[i]..... R[n]

3. 效率分析

- ❖ 希尔排序适用于待排序记录数量较大的情况，在此情况下，希尔排序方法比直接插入排序方法速度快。
- ❖ 希尔排序是一种不稳定的排序方法。

9.3 交换排序

- ❖ 基本思想：在排序过程中，通过对待排序记录序列中元素间关键字的比较，**发现逆序的**，则交换元素位置。

R[0]
R[1]
R[2]
R[3]
.....
.....
R[n]

9.3.1 冒泡排序

1. 基本思想-----**递增排列**

- ❖ 冒泡排序的处理过程为：

- 1) 将整个待排序的记录序列划分为**无序区**和**有序区**，初始状态有序区中数据元素个数为0，无序区为所有待排序的记录。
- 2) 对**无序区**从前向后依次将相邻记录的关键字进行比较，若逆序将其交换，从而使关键字值小的记录“上浮”，关键字值大的记录“下沉”。最终目前关键字最大的记录进入有序区。然后在余下的记录中再找出最大的记录，放入有序区



❖ 每经过一趟冒泡排序，都使无序区中关键字值最大的记录进入有序区

❖ 对于由 n 个记录组成的记录序列，最多经过 ? 趟冒泡排序，就可以将这 n 个记录按关键字大小排序。

❖ 第 i 趟比较 ? 次。

❖ 如何提前结束：在一趟冒泡排序过程中没有交换，说明待排序记录已全部有序，冒泡排序过程结束。

```
void BubbleSort(RecType R[ ], int n) // 排序 R [1.....n]
{
    i = n;    // i 指示无序序列中最后一个记录的位置
    while ( i > 1)
    {
        LastExchange=1; // 记最后一次交换发生的位置
        for(j=1; j<i; j++)
        {
            if(R[j].key>R[j+1].key)
            {
                temp=R[j]; R[j]=R[j+1]; R[j+1]=temp; // 逆
                序时交换
            }
            LastExchange=j;
        } // if
        i=i-1;
    } // while
}
```

```
void BubbleSort(RecType R[ ], int n) // 排序R [1.....n]
```

```
{ i = n;    // i 指示无序序列中最后一个记录的位置
```

```
while ( i > 1)
```

```
{LastExchange=1; // 记最后一次交换发生的位置
```

```
for(j=1; j<i; j++)
```

```
if(R[j].key>R[j+1].key)
```

```
{ temp=R[j]; R[j]=R[j+1]; R[j+1]=temp;
```

```
LastExchange=j;
```

```
} // if
```

```
i=LastExchange; //记录最后交换位置，此后的已经  
排好序
```

```
} // while
```

```
}
```

3. 效率分析

- ❖ 冒泡排序的比较次数和记录的交换次数与记录的初始顺序有关。在正序时，比较次数为 $n-1$ ，交换次数为0；在逆序时，比较次数和交换次数均为 $n(n-1)/2$ ，因此，总的时间复杂度为 $O(n^2)$ 。
- ❖ 冒泡排序是一种**稳定的排序方法**。

9.3.2 快速排序

1. 基本思想—递增

从排序序列中任选一记录作为**枢轴（或支点）**，凡其关键字小于枢轴的记录均移动至该记录之前，而关键字大于枢轴的记录均移动至该记录之后。

一趟排序后“枢轴”到位，并将序列分成两部分，前一部分比“枢轴”，后一部分比“枢轴”大，再分别对这两部分排序。

$R[1] \dots R[i-1] R[i] R[i+1] \dots R[n-1] R[n]$

$R[1] \dots R[k] \dots R[i-1] R[i] R[i+1] \dots R[m] \dots R[n-1] R[n]$

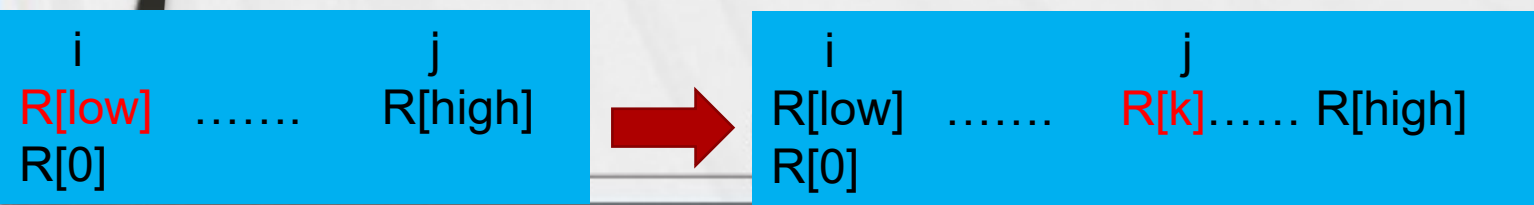
❖ 设待排序序列的下界和上界分别为low和high，选R[low]是枢轴元素，一趟快速排序的具体做法是：

①首先将R[low]中的记录保存到R[0]变量中，用两个整型变量i、j分别指向low和high所在位置上的记录。

②先从j所指的记录起自右向左逐一将关键字和R[0].key进行比较，当找到第1个关键字小于R[0].key的记录时，将此记录复制到i所指的位置上去。

③然后从i+1所指的记录起自左向右逐一将关键字和R[0].key进行比较，当找到第1个关键字大于R[0].key的记录时，将该记录复制到j所指的位置上去。

④接着再从j-1所指的记录重复以上的（2）、（3）两步，直到*i=j*为止，此时将R[0]中的记录放回到i的位置上，一趟快速排序完成。



R[0] R[1] R[2] R[3] R[4] R[5] R[6] R[7] R[8]

52	52	35	68	96	85	17	25	52
----	----	----	----	----	----	----	----	----

i j

0	1	2	3	4	5	6	7	8
52		35	68	96	85	17	25	52

i j

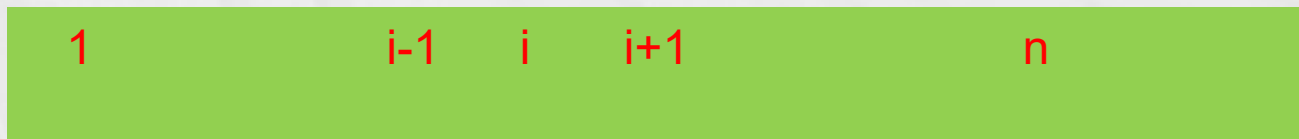
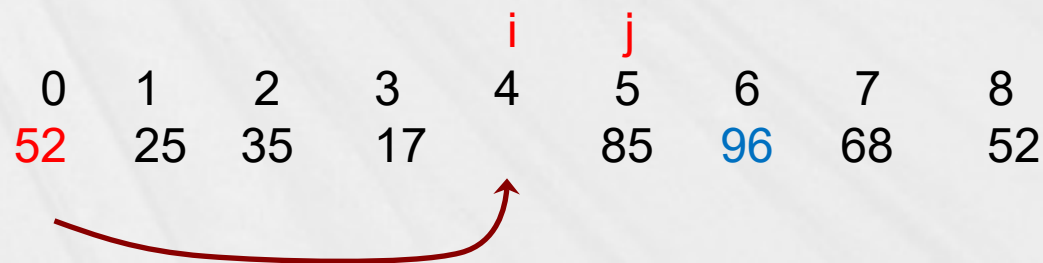
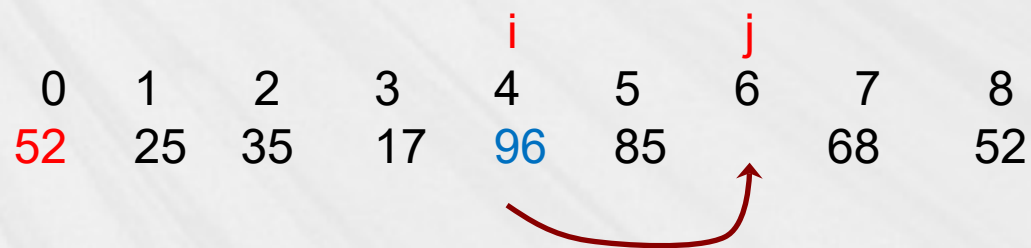
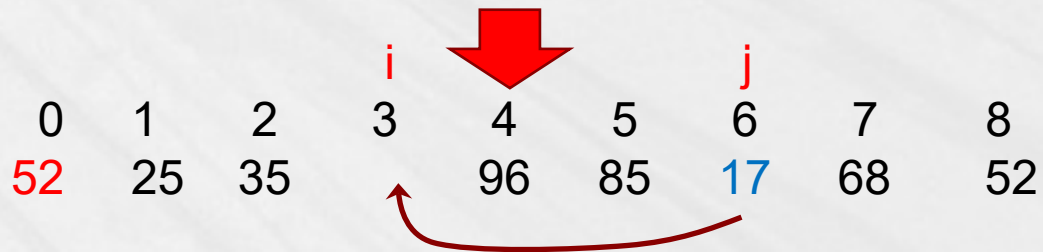
0	1	2	3	4	5	6	7	8
52		35	68	96	85	17	25	52

i j

0	1	2	3	4	5	6	7	8
52	25	35	68	96	85	17		52

i j

0	1	2	3	4	5	6	7	8
52	25	35		96	85	17	68	52



2. 算法 ---- 一遍排序算法，返回枢轴最终所在位置

```
int Partition(RecType R[ ], int l, int h)
```

```
{ int i=l; j=h; // 用变量i, j记录待排序记录首尾位置
```

```
  R[0] = R[i]; // 以子表的第一个记录作枢轴，将其暂存到记录R[0]中
```

```
  x = R[i].key; // 用变量x存放枢轴记录的关键字
```

```
  while(i<j) // 从表的两端交替地向中间扫描
```

```
    { while(i<j && R[j].key>=x)  j--;
```

```
      R[i++] = R[j]; // 将比枢轴小的记录移到低端
```

```
      while(i<j && R[i].key<=x)  i++;
```

```
      R[j--] = R[i]; // 将比枢轴大的记录移到高端
```

```
    } // while
```

```
    R[i] = R[0]; // 枢轴记录到位
```

```
    return i; // 返回枢轴位置
```

```
} // Partition
```

i		j
R[low]	R[high]
R[0]		

递归算法:

```
void QuickSort(RecType R[ ], int s, int t)
{ // 对记录序列R[s..t]进行快速排序
    if(s<t)
    { k=Partition(R,s,t);
      QuickSort(R,s,k-1);
      QuickSort(R,k+1,t);
    } // if
} // QuickSort
```

R[0]	R[1]	R[2]	R[3]	R[4]	R[5]	R[6]	R[7]	R[8]
	52	35	68	96	85	17	25	52

0	1	2	3	4	5	6	7	8
25	35	17	52	85	96	68	52	

3. 效率分析

- ◆ **快速排序**的平均时间复杂度为 **$O(n\log_2 n)$** ，若待排记录的初始状态为按关键字有序，快速排序将蜕化为冒泡排序，其时间复杂度为 $O(n^2)$ 。
- ◆ 快速排序是**不稳定排序**。

9.4 选择排序

- ❖ **选择排序的基本思想**是依次从待排序记录序列中选择出关键字值最小（或最大）的记录、关键字值次之的记录、.....，并分别将它们交换到序列的第1个位置、第2个位置、.....，从而使记录成为按关键字值由小到大（或由大到小）顺序排列。

9.4.1 直接选择排序

1. 基本思想-----递增

共有 n 个记录，假设前面 $i-1$ 个记录已经**递增有序**，第 i 趟从序列 $R[i.....n]$ 的 $n-i+1$ 记录中选出关键字最小的记录，与 $R[i]$ 互换。

$R[1]$ $R[2]$ $R[i-1]$ $R[i]$ $R[i+1]$ $R[n]$



$R[1]$ $R[2]$ $R[i-1]$ $R[i]$ $R[i+1]$ $R[n]$

$R[1]$ $R[2]$ $R[i-1]$ $R[i]$ $R[i+1]$ $R[n]$

$R[1]$ $R[2]$ $R[i-1]$ $R[i]$ $R[n]$

9.4.1 直接选择排序

举例：

21 19 23 13 8 17 33 45

21 19 23 13 8 17 33 45

8 19 23 13 21 17 33 45

8 19 23 13 21 17 33 45

8 13 23 19 21 17 33 45

8 13 17 19 21 23 33 45



2. 算法 // 对记录序列 $R[1.....n]$ 作直接选择排序。

```
void SelectSort(RecType R[ ], int n)
```

```
{
```

```
    for(i=1; i<n; i++) // 选择第i小的记录，并交换到位
```

```
        { k=i; // k记录关键字最小元素的下标
```

```
            for(j=i+1; j<=n; j++) // 找最小元素的下标->k
```

```
                if(R[j].key<R[k].key) k=j;
```

```
                if(i!=k) R[i]↔R[k]; // 与第i个记录交换
```

```
            } // for
```

```
    } // SelectSort
```

R[1]

R[2]

...

...

R[i-1]

R[i]

R[i+1]

...

...

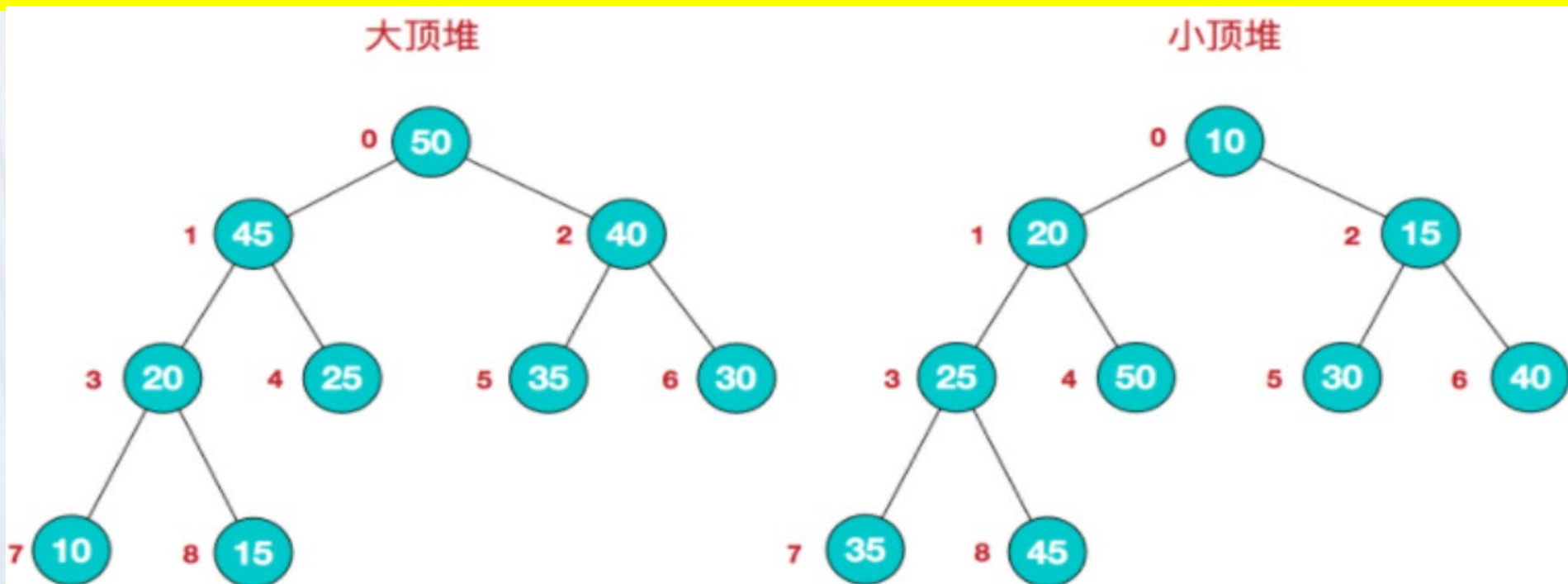
R[n]

3.效率分析

- ❖ 直接选择排序比较次数与关键字初始排序无关。
总的比较次数为 $n * (n-1) / 2$ ，所以时间复杂度为 $O(n^2)$ 。
- ❖ 直接选择排序是**不稳定的排序方法**。

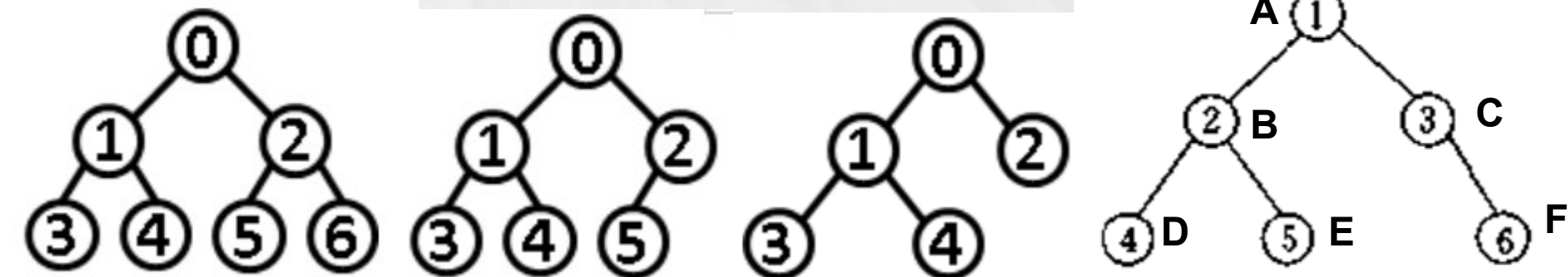
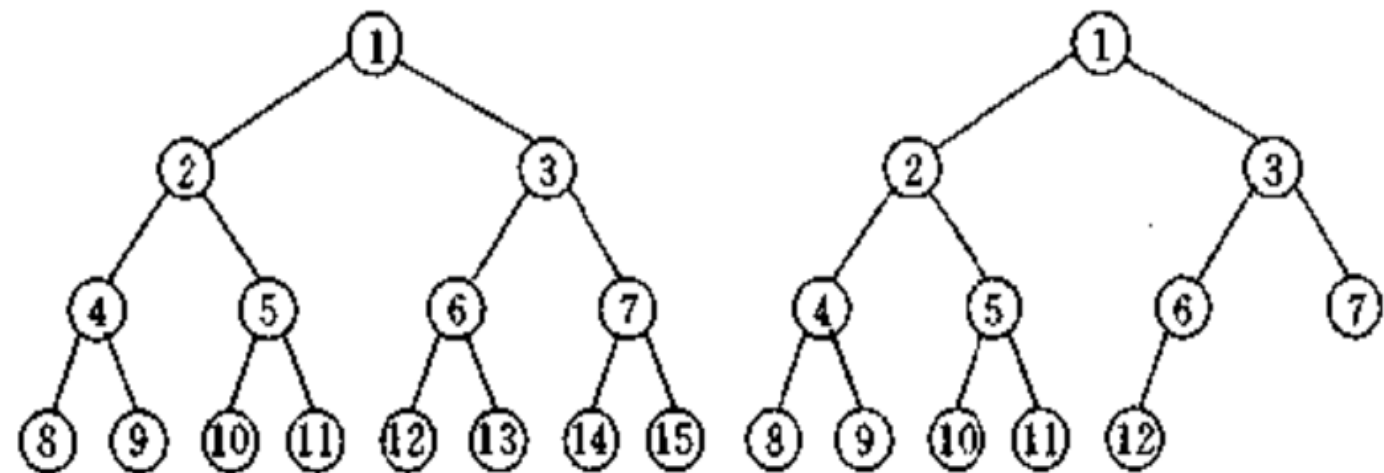
9.4.2 堆排序-----Heapsort

- ❖ 如何快速地找出最大值或者最小值
- ❖ 堆的定义：堆是关键字满足下列性质的记录序列 $\{K_1, K_2, \dots, K_n\}$ ： $K_i \leq K_{2i}$, $K_i \leq K_{2i+1}$, 或者 $K_i \geq K_{2i}$, $K_i \geq K_{2i+1}$ ($i=1, 2, \dots, n/2$)
- ❖ 若上述序列是堆，则 K_1 必是序列中的最小值，或者最大值，分别称作小顶堆或大顶堆(小根堆或大根堆)



9.4.2 堆排序

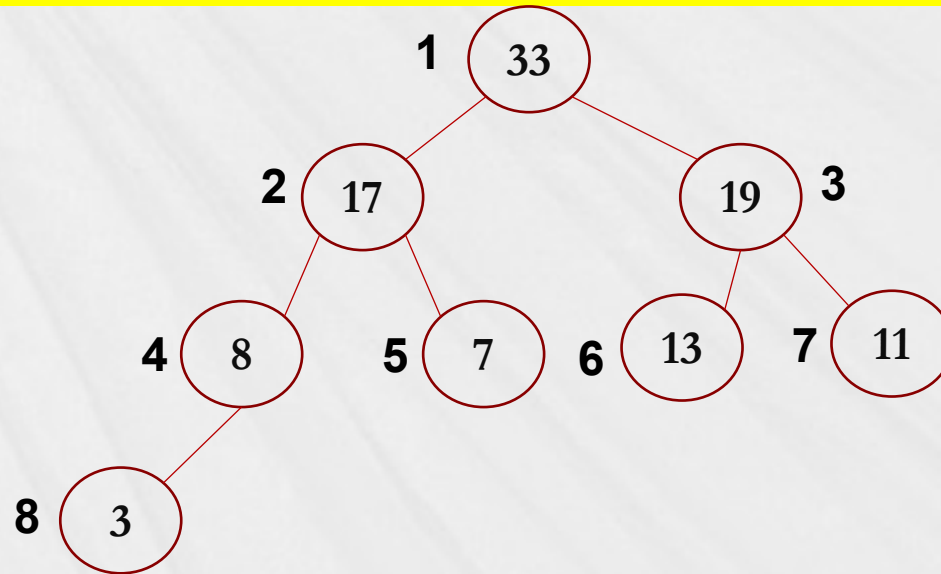
- ❖ 满二叉树
- ❖ 完全二叉树
- ❖ 一般二叉树
- ❖ 顺序存储结构
- ❖ 双亲与孩子节点的关系



9.4.2 堆排序

❖ 满二叉树、完全二叉树：

❖ 如：33，17，19，8，7，13，11，3是**大顶堆**，若将此序列看成是一棵**完全二叉树**，则**堆**或是**空树**或是满足下列特性的**完全二叉树**：其左、右子树分别是堆，任何一个结点的值不小于左、右孩子结点（若存在）的值。



顺序存储结构：

R[1]	R[2]	R[3]	R[4]	R[5]	R[6]	R[7]	R[8]
33	17	19	8	7	13	11	3

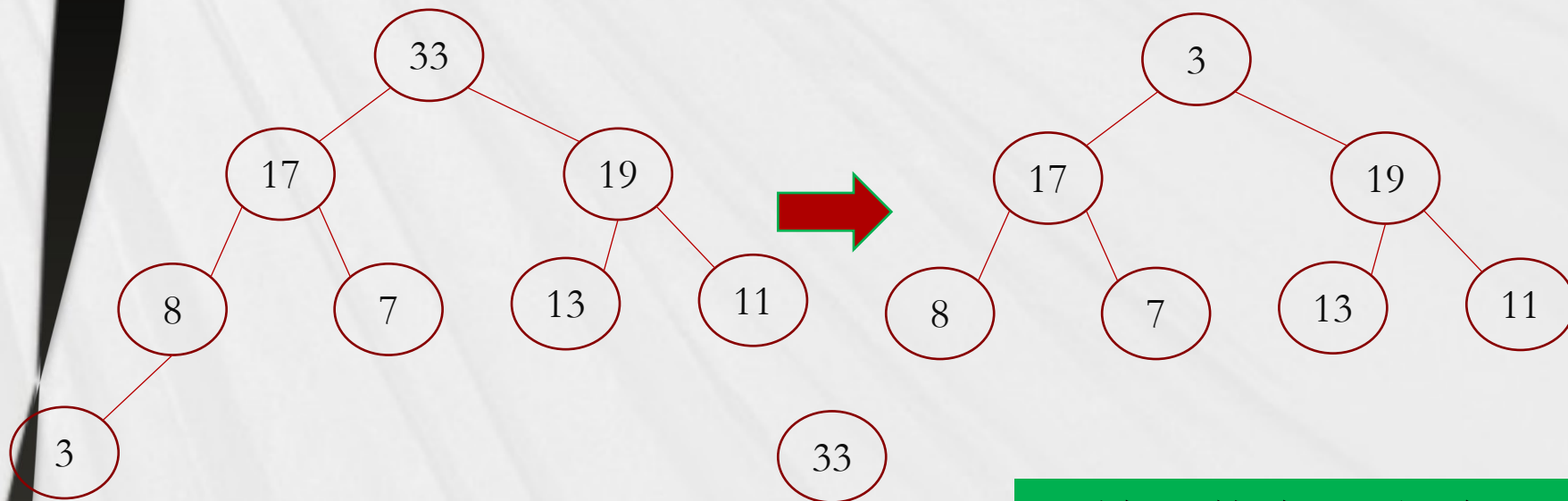
9.4.2 堆排序---大顶堆

❖ 待排序序列=无序区+有序区

R[1] R[2] R[3] R[4] R[5] R[6] R[7] R[8] 无序区
33 17 19 8 7 13 11 3



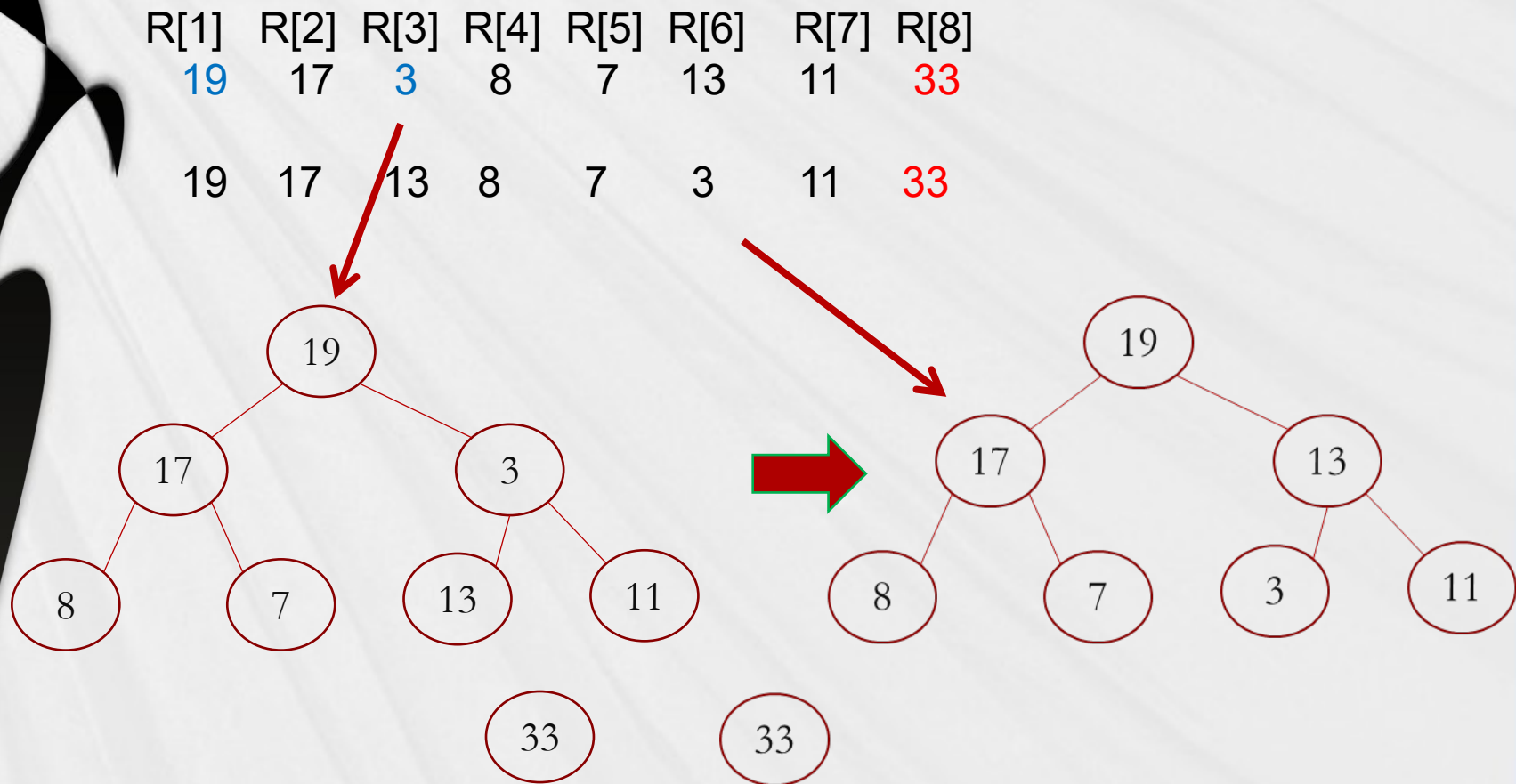
R[1] R[2] R[3] R[4] R[5] R[6] R[7] R[8] 无序区+有序区
3 17 19 8 7 13 11 33



调堆：找左、右孩子的最大者与之交换

9.4.2 堆排序

❖ 待排序序列=无序区+有序区



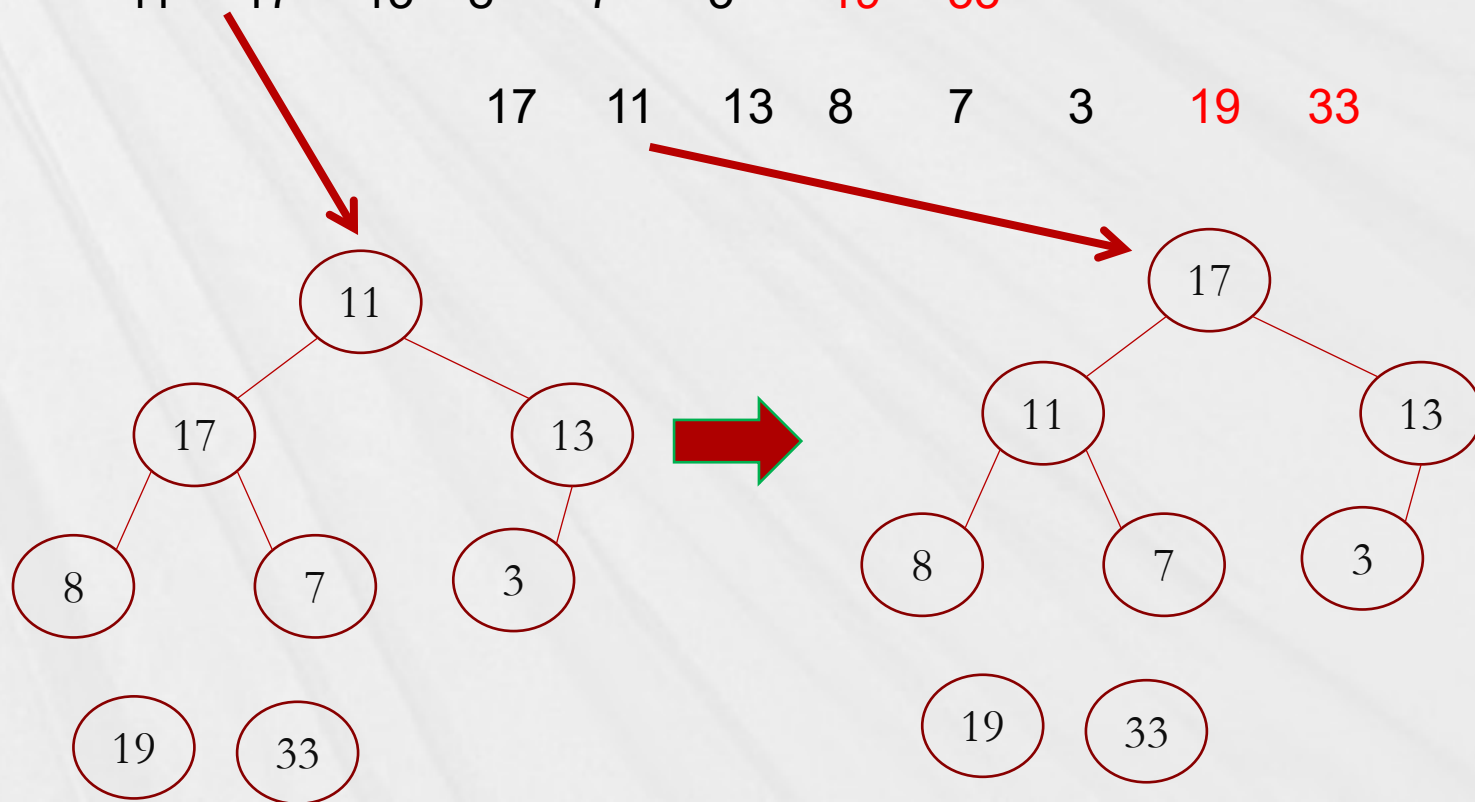
调堆：找左、右孩子的最大者与之交换

9.4.2 堆排序

❖ 待排序序列=无序区+有序区

R[1]	R[2]	R[3]	R[4]	R[5]	R[6]	R[7]	R[8]
19	17	13	8	7	3	11	33

R[1]	R[2]	R[3]	R[4]	R[5]	R[6]	R[7]	R[8]
11	17	13	8	7	3	19	33



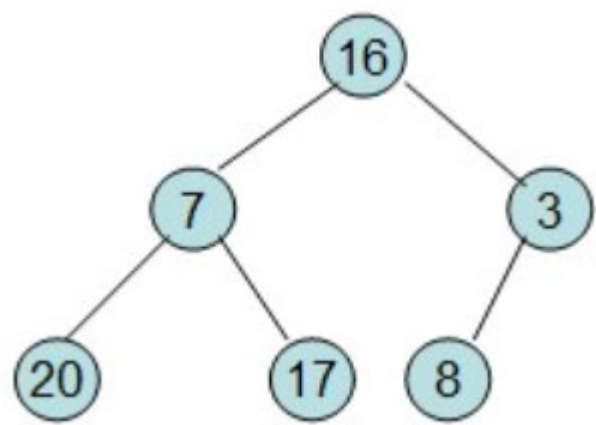
- ① 先将初始 $R[1...n]$ 建成一个大根堆，此堆为初始的无序区
- ② 再将关键字最大的记录 $R[1]$ （即堆顶）和无序区的最后一个记录 $R[n]$ 交换，由此得到新的无序区 $R[1...n-1]$ 和有序区 $R[n]$
- ③ 当前无序区 $R[1...n-1]$ 调整为堆。然后再次将 $R[1...n-1]$ 中关键字最大的记录 $R[1]$ 和该区间的最后一个记录 $R[n-1]$ 交换，由此得到新的无序区 $R[1...n-2]$ 和有序区 $R[n-1...n]$

同样要将 $R[1...n-2]$ 调整为堆。……

直到无序区只有一个元素为止。

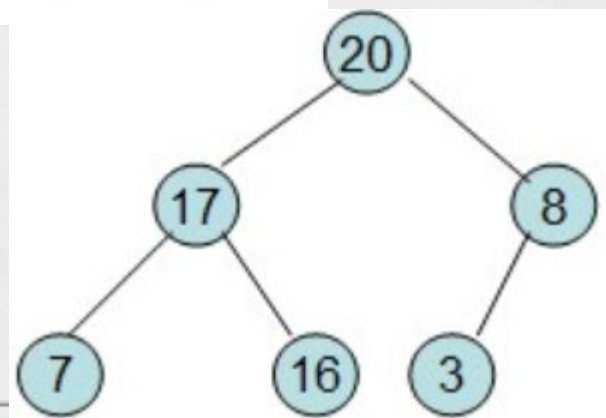
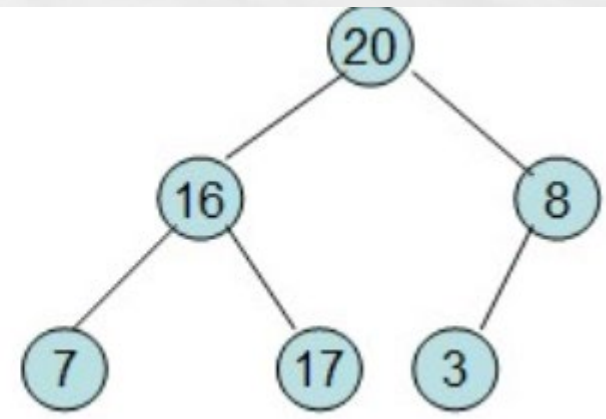
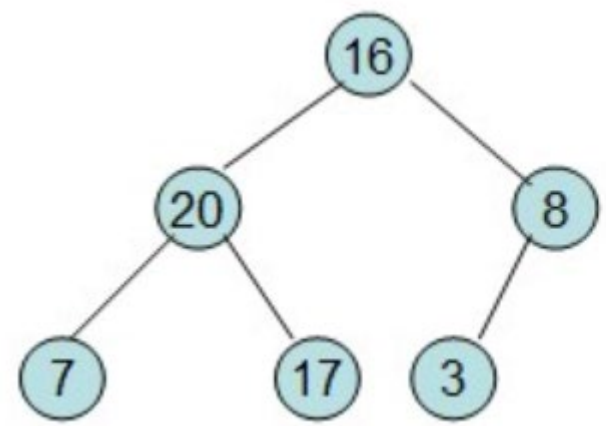
调堆的过程称为“筛选”

给定一个列表 $\text{array}=[16,7,3,20,17,8]$ ，对其进行堆排序。



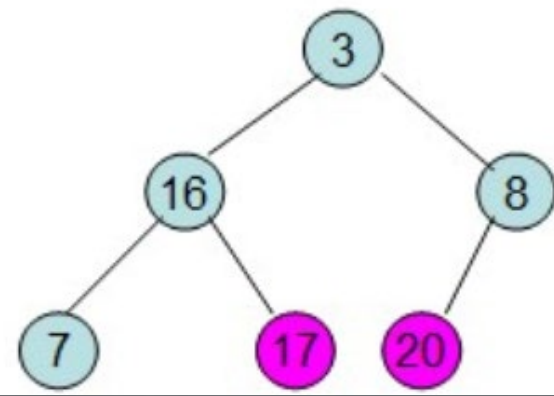
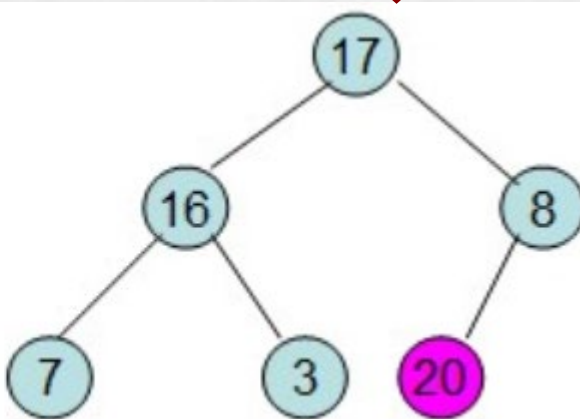
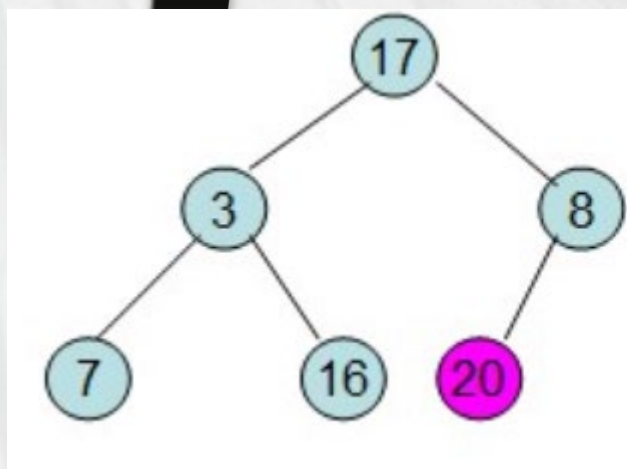
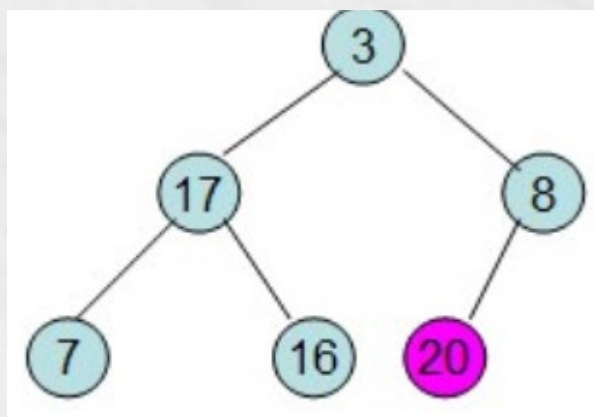
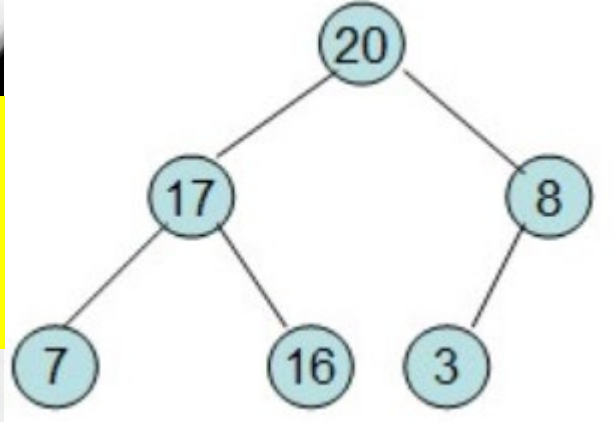
↓
初始堆

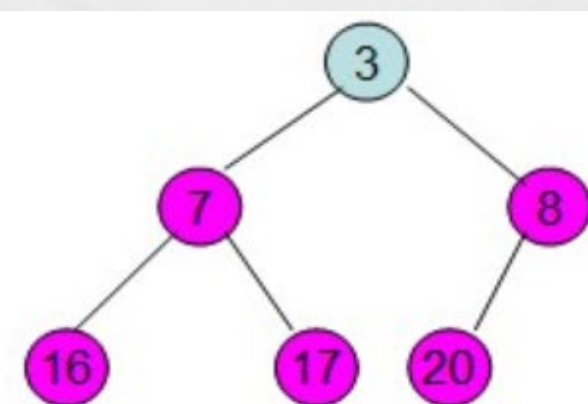
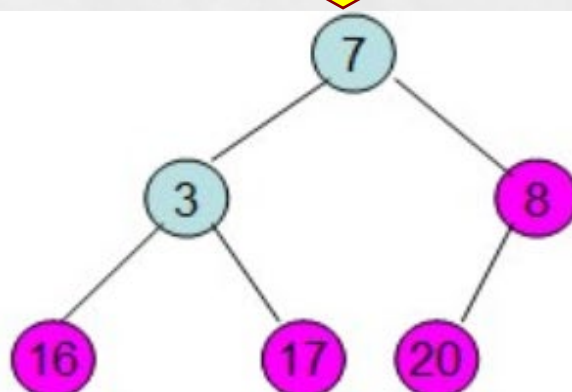
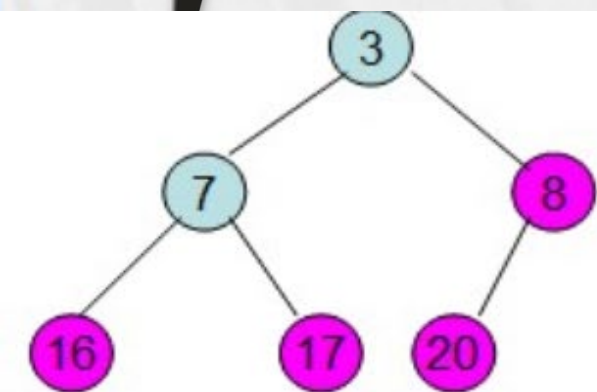
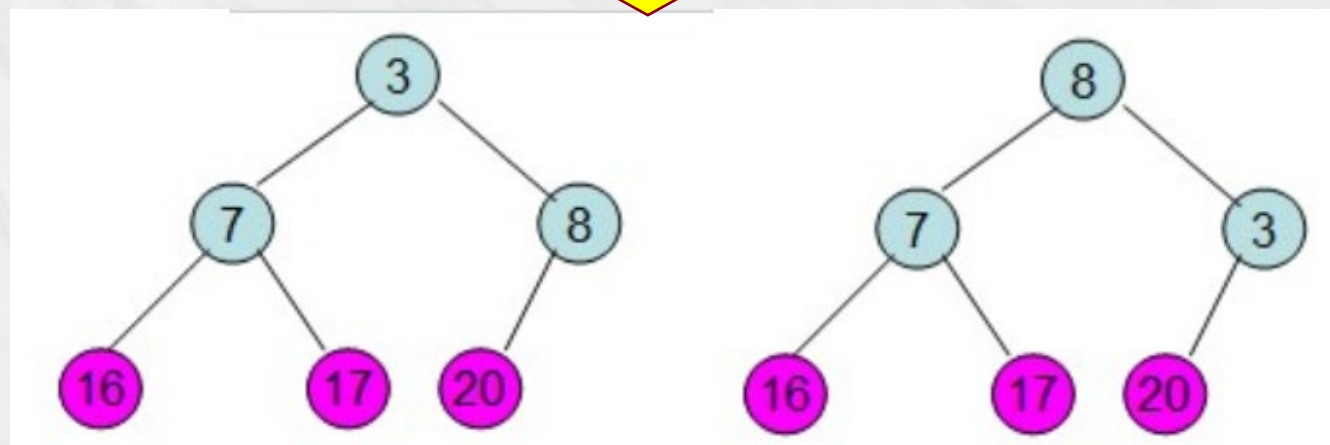
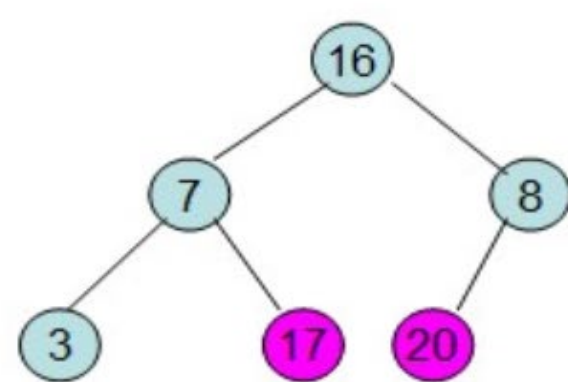
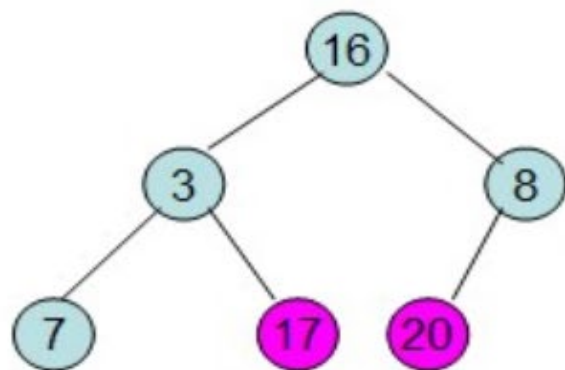
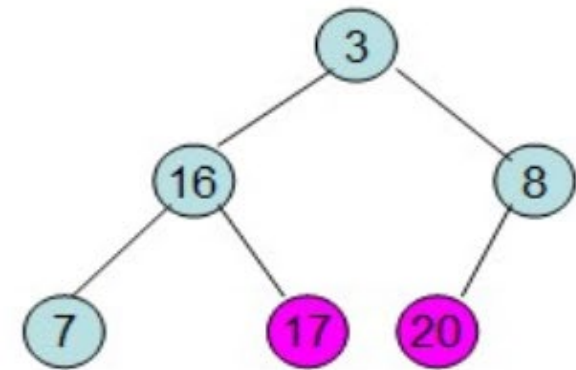
(从最后一个有子节点开始往上调整最大堆)



堆顶元素 $R[1]$ 与最后一个元素 $R[n]$ 交换，交换后堆长度减一

初始堆





大根堆排序算法的基本操作:

①初始化建堆，建堆是不断调整堆的过程，从 $n/2$ 处节点开始调整，一直到第一个节点，此处 n 是堆中元素的个数。建堆的过程是线性的过程，从 $n/2$ 到1处一直调用调整堆的过程。

②将堆的根节点取出(一般是与最后一个节点进行交换)，将前面 $n-1$ 个节点继续调整成堆。

③调整堆：比较节点 i 和它的孩子节点 $\text{left}(i)$, $\text{right}(i)$ ，选出三者最大者，如果最大值不是节点 i 而是它的一个孩子节点，交互节点 i 和该节点；重复上述操作，直到.....

堆排序是利用上面过程来进行的。首先是根据元素构建初始堆。然后将堆的根节点取出(一般是与最后一个节点进行交换)，将前面 $n-1$ 个节点继续进行堆调整的过程，然后再将根节点取出，这样一直到所有节点都取出。

2. 算法: $R[i+1.....m]$ 为大根堆, 加入 $R[i]$ 后调整为大根堆

//找到孩子中最大关键字的记录, 然后互换

```
void Sift(RecType R[ ], int i, int m)
```

```
{  $R[0]=R[i]$ ;
```

```
  for( $j=2*i$ ;  $j \leq m$ ;  $j*=2$ )
```

```
    { if( $j < m$  &&  $R[j].key < R[j+1].key$ )  $j++$ ; //沿大者方向筛选
```

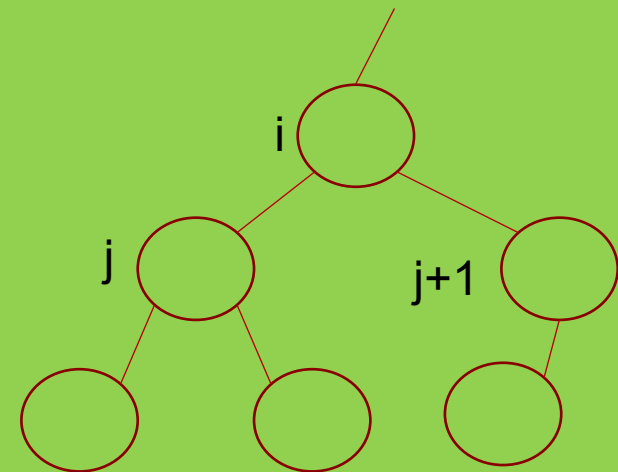
```
      if( $R[0].key < R[j].key$ ) {  $R[i]=R[j]$ ;  $i=j$ ; }
```

```
      else break;
```

```
    } // for
```

```
     $R[i]=R[0]$ ;
```

```
} // Sif
```



```
void HeapSort(RecType R[ ], int n)
```

```
{ // 对记录序列R[1..n]进行堆排序。
```

```
    for(i=n/2; i>0; i--) // 把R[1..n]建成大顶堆，从最后一个非叶子节点开始
```

```
        Sift(R, i, n);
```

```
    for(i=n; i>1; i--) // 输出并调堆
```

```
    { R[1]  $\longleftrightarrow$  R[i];
```

```
        Sift(R, 1, i-1); // 将R[1..i-1]重新调整为大顶堆
```

```
    } }
```

3. 效率分析

- ❖ 堆排序的时间复杂度为 $O(n \log_2 n)$ 。
- ❖ 堆排序只需要一个记录的辅助存储空间。
- ❖ 堆排序方法是不稳定的。

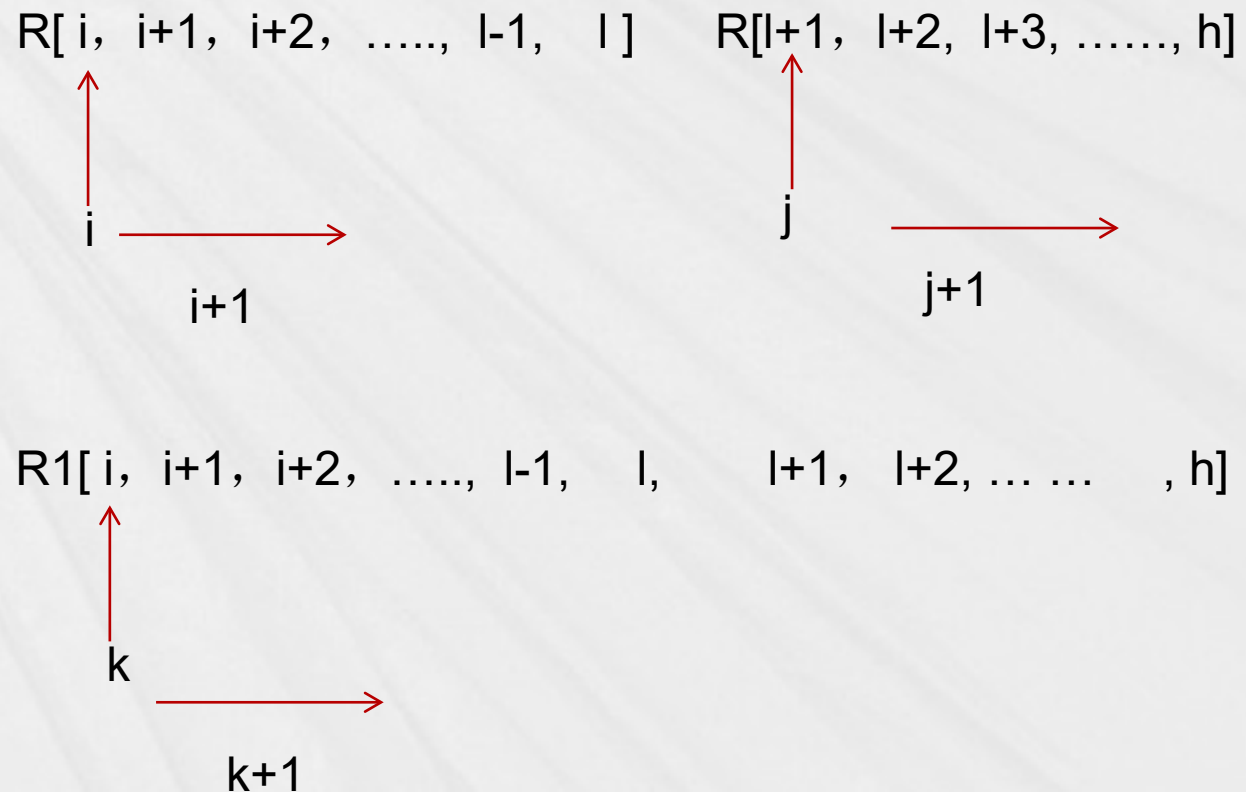
9.5 归并排序

1. 基本思想

- ❖ **归并排序**是将两个或两个以上的**有序序列**合并成一个**新的有序序列**.
- ❖ 基本思想：将具有 n 个待排序记录的序列看成是 n 个长度为1的有序序列，进行两两归并，得到 $\lceil n/2 \rceil$ 个长度为2的有序序列，再进行两两归并，得到 $\lceil n/4 \rceil$ 个长度为4的有序序列，如此重复，直至得到一个长度为 n 的有序序列为止。

两个有序序列合并成一个新的有序序列.

均为有序



另外一个
数组

2. 算法

```
void Merge(RecType R[ ],RecType R1[ ], int i, int l, int h)
{ // 将有序的R[i.....l]和R[l+1.....h]归并为有序的R1[i.....h]

    for(k=i, j=l+1; i<=l&& j<=h; k++)

        if(R[i].key<=R[j].key)    R1[k]=R[i++];

        else R1[k]=R[j++];

    if(i<=l) R1[k..h]=R[i..l];    // 将剩余的R[i..l]复制到R1

    if(j<=h) R1[k..h]=R[j..h];    // 将剩余的R[j..h]复制到R1

} // Merge
```



```
void Msort(RecType R[ ], RecType R1[ ], int s, int t)
{ // 将R[s..t]进行2-路归并排序为R1[s..t], 递归算法

    if(s==t) R1[s]=R[s];

    else

        { m=(s+t)/2;          // 将R[s..t]平分为R[s..m]和R[m+1..t]


          Msort(R,R2,s,m); // 递归地将R[s..m]归并为有序的R2[s..m]

          Msort(R,R2,m+1,t); // 递归地R[m+1..t]归并为有序的R2[m+1..t]

          Merge(R2,R1,s,m,t); // 将R2[s..m]和R2[m+1..t]归并到R1[s..t]

        }

} // MSort
```



```
void MergeSort(RecType R[ ], int n)
{ // 对记录序列R[1..n]作2-路归并排序。

    MSort(R, R, 1, n);

} // MergeSort
```

3. 效率分析

- ◆ 2-路归并排序的时间复杂度为 $O(n \log_2 n)$ 。
- ◆ 2-路归并排序是一种稳定的排序方法。

9.6 基数排序

1. 基本思想

- ❖ 基数排序就是一种借助“多关键字排序”的思想来实现“单关键字排序”的算法。
- ❖ 扑克牌：花色、点数
- ❖ 对多关键字 $K_0 K_1 \dots K_{d-1}$ 排序通常有两种方法：
 - (1) **最高位优先MSD法**：先对 K_0 进行排序，并按 K_0 的不同值将记录序列分成若干子序列之后，分别对 K_1 进行排序，...，依次类推，直至最后对最次位关键字排序完成为止。
 - (2) **最低位优先LSD法**：先对 K_{d-1} 进行排序，然后对 K_{d-2} 进行排序，依次类推，直至对最主位关键字 K_0 排序完成为止。

- ❖ 假设多关键字的记录序列中，**每个关键字的取值范围相同**，则**按LSD进行排序**时，可以采用“**分配—收集**”的方法。
- ❖ 对于**数字型或字符型的单关键字**，若可以看成是由d个分量（ $K_{i0}, K_{i1}, \dots, K_{id-1}$ ）构成的，每个分量取值范围相同 $C_1 \leq K_{ij} \leq C_{rd}$ （ $0 \leq j < d$ ）（可能取值的个数 **r_d 称为基数**），可以采用分配—收集的方法进行排序，这种方法称**基数排序法**。

【例4】已知关键字序列

{278, 109, 063, 930, 589, 184, 505, 269, 008, 083},

写出基数排序的排序过程。

最低位排序:

278 109 063 930 589 184 505 269 008 083

0:

1:

2:

3:

4:

5:

6:

7:

8:

9:

桶

最低位排序:

278 109 063 930 589 184 505 269 008 083

0: 930

1:

2:

3: 063 083

4: 184

5: 505

6:

7:

8: 278 008

9: 109 589 269

桶

尾

首

收集

930 063 083 184 505 278 008 109 589 269

次低位:

930 063 083 184 505 278 008 109 589 269

分配

0: 505 008 109

1:

2:

3: 930

4:

5:

6: 063 269

7: 278

8: 083 184 589

9:

收集 → 505 008 109 930 063 269 278 083 184 589

最高位:

505 008 109 930 063 269 278 083 184 589

0: 008 063 083

1: 109 184

2: 269 278

3:

4:

5: 505 589

6:

7:

8:

9: 930

每个桶相当与
什么数据结构?

收集 →

008 063 083 109 184 269 278 505 589 930

数据结构：用结构体数组建立静态链表

struct

```
{ int key[d];
```

```
    int next;        //相邻的下一个数据元素的下标
```

```
    int other; } R[n+1];
```

初始数据：

key	21	6	2	87	33	55	//排序过程中各个元素的位置不同
-----	----	---	---	----	----	----	------------------

next	2	3	4	5	6	-1	//下一个数据元素的下标
------	---	---	---	---	---	----	--------------

排好序后数据： 排号序后返回第一个数据元素的下标，如： 3

下标	1	2	3	4	5	6
----	---	---	---	---	---	---

key	21	6	2	87	33	55
-----	----	---	---	----	----	----

next	5	1	2	-1	6	4	//下一个数据元素的下标
------	---	---	---	----	---	---	--------------

数据结构：数组 $R[n+1]$

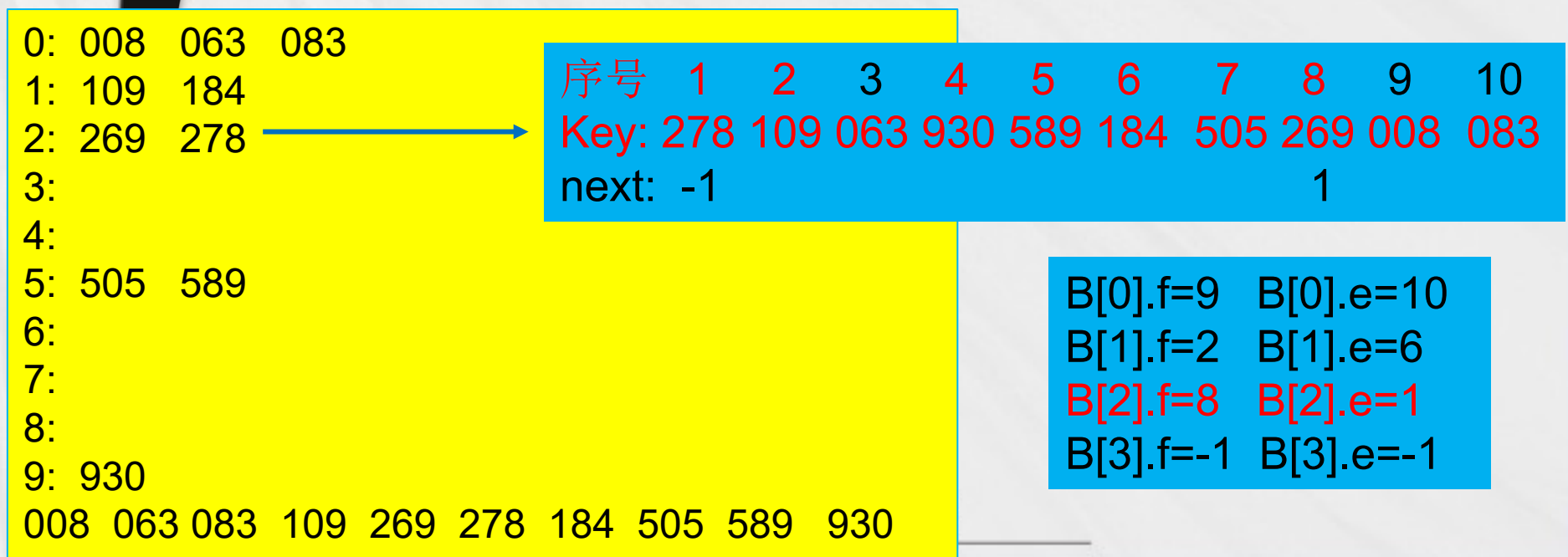
```
struct { int key[d];  
        int next;  
        int other; } R[n+1];
```

```
struct  
{ int f, e; } B[m]; // key[i] from 0 to m-1
```

0: 008 063 083 →
1: 109 184
2: 269 278
3:
4:
5: 505 589
6:
7:
8:
9: 930
008 063 083 109 269 278 184 505 589 930

序号	1	2	3	4	5	6	7	8	9	10
Key:	278	109	063	930	589	184	505	269	008	083
next:			10						3	-1

B[0].f=9 B[0].e=10
B[1].f=2 B[1].e=6
B[2].f=8 B[2].e=1
B[3].f=-1 B[3].e=-1



0: 008 063 083 →
 1: 109 184
 2: 269 278
 3:
 4:
 5: 505 589
 6:
 7:
 8:
 9: 930
 008 063 083 109 269 278 184 505 589 930

序号 1 2 3 4 5 6 7 8 9 10
 Key: 278 109 063 930 589 184 505 269 008 083
 next: 10 3 -1

B[0].f=9 B[0].e=10
 B[1].f=2 B[1].e=6
 B[2].f=8 B[2].e=1
 B[3].f=-1 B[3].e=-1
 B[4].f=-1 B[4].e=-1
 B[5].f=7 B[5].e=5
 B[6].f=-1 B[6].e=-1
 B[7].f=-1 B[7].e=-1
 B[8].f=-1 B[8].e=-1
 B[9].f=4 B[9].e=4



序号 1 2 3 4 5 6 7 8 9 10
 Key: 278 109 063 930 589 184 505 269 008 083
 next: 7 6 10 -1 4 8 5 1 3 2

返回9

数据结构：数组 $R[n+1]$

```
typedef struct {  
    int key[d];  
    int next;  
    int other; } SLRectype ;
```

```
SLRectype R[n+1];
```

```
struct { int f, e; } B[m]; // key[i] from 0 to m-1
```

```
int Radixsort(SLRectype R[ ], int n) // 返回第一个元素的下标
```

```
{ int i, j, p, t;
```

```
for(i=1; i<n; i++)
```

```
    R[i].next=i+1;
```

```
R[n].next=-1;
```

初始数据:

key	21	6	2	87	33	55
next	2	3	4	5	6	-1

从最低位开始
分配各个记录
到相应的桶里

```
p=1; //R[p] to a barrel
for(j=d-1; j>=0; j--) //key[j]
{ for(i=0; i<m; i++)
    B[i].f=B[i].e=-1;
While(p != -1 )
{ k=R[p].key[j];
  if(B[k].f ==-1) B[k].f=p;
  else R[B[k].e].next=p;
  B[k].e=p;
  p=R[p].next;
}
```

顺序收集各个桶
中的记录，组成
一个链表

```
i=0;
while(B[i].f== -1) i++;
p=B[i].f; t=B[i].e;
while(i<m-1)
{ i++;
  if(B[i].f != -1) {
    R[t].next=B[i].f;
    t=B[i].e;      }
}
R[t].next =-1; } //for(j=d-1;;)
Return p;
}
```

初始数据:

key	21	6	2	87	33	55
next	2	3	4	5	6	-1

3. 效率分析

- ❖ 基数排序的时间复杂度是 $O(d \cdot (r_d + n))$ 。
- ❖ 基数排序存储空间复杂度为 $O(r_d)$ 。
- ❖ 基数排序是稳定的。

本章小结

- ❖ 内部排序主要有**直接插入排序**、**希尔排序**、**冒泡排序**、**快速排序**、**直接选择排序**、**堆排序**、归并排序和基数排序。
- ❖ **当 n 较小时**，可采用直接插入排序和直接选择排序。当待排序记录的初始状态已是按关键字**基本有序**时，可选择**直接插入排序或冒泡排序**。
- ❖ **当 n 较大时**，若关键字有明显结构特征，且关键字位数较少，易于分解时，采用基数排序较好。若关键字无明显特征时，可采用快速排序、堆排序或归并排序。
- ❖ 直接插入排序、冒泡排序、归并排序和基数排序方法是稳定的
- ❖ 希尔排序、快速排序、直接选择排序、堆排序方法是不稳定的

本章习题

1. 在各种排序方法中，哪些是稳定的？哪些是不稳定的？
2. 假设关键字从小到大排序，问在什么情况下，冒泡排序算法关键字交换的次数最多？
3. 假设待排序的关键字序列为{15, 20, 8, 32, 28, 20, 40, 18}，试分别写出使用以下排序方法每趟排序后的结果。
 - (1) 希尔排序
 - (2) 归并排序
 - (3) 快速排序

本章习题

4. 判断下列的关键字序列是否是一个大根堆，如果不是，把它调整成堆。

(1) 95, 86, 60, 85, 20, 25, 10, 70

(2) 95, 70, 60, 20, 85, 25, 10, 86

5. 假设待排序的关键字序列为{268, 109, 023, 930, 547, 505, 328, 240, 118}，试写出使用基数排序方法地1趟分配和收集后的结果。

下标: 1 2 3 4 5 6 7 8 9

Key: 268, 109, 023, 930, 547, 505, 328, 240, 118

Next: