第8章 排序
Sorting

本章内容

- 8.1 概述
- 8.2 插入排序
- 8.3 选择排序
- 8.4 交换排序
- 8.5 归并排序
- 8.6 基数排序
- 8.7 外部排序

8.1 概述

- 什么是排序？
- 排序是计算机内经常进行的一种操作，其目的是将一组“无序”的记录序列调整为“有序”的记录序列。

- 例如：
- 52, 49, 80, 36, 14, 58, 61, 23, 97, 75
- 14, 23, 36, 49, 52, 58, 61 ,75, 80, 97

- 排序：
  - 假设含n个记录的序列为{ R_1, R_2, …， R_n }
  - 其相应的关键字序列为 { K_1, K_2, …， K_n }
  - 这些关键字相互之间可以进行比较，即在它们之间存在着这样一个关系： $K_{p1} \le K_{p2} \le \dots \le K_{pn}$
  - 按此固有关系将上式记录序列重新排列为

    { R_{p1}, Rp2, …， Rpn }

    的操作称作排序。

排序的稳定性

- 在待排记录序列中，任何两个关键字相同的记录，用某种排序方法排序后相对位置不变，则称这种排序方法是稳定的，否则称为不稳定的。
- 例如：
- 待排序列：　49,38,65,97,76,13,27,<u>49</u>
- 排序后：　　13,27,38,49,<u>49</u>,65,76,97 — 稳定
- 排序后：　　13,27,38,<u>49</u>,49,65,76,97—不稳定

排序方法的分类

- 按照是否访问外存：
  - 内部排序：整个排序过程不需要访问外存；
  - 外部排序：若参加排序的记录数量很大，整个序列的排序过程不可能在内存中完成。

内部排序方法的分类

- 根据设置有序序列的方式的不同，分为：
  - 插入排序：直接插入排序、折半插入排序、希尔排序
  - 交换排序：冒泡排序、快速排序
  - 选择排序：简单选择排序、堆排序
  - 归并排序：2-路归并排序
  - 基数排序

数据结构

- 待排记录的数据类型定义如下:

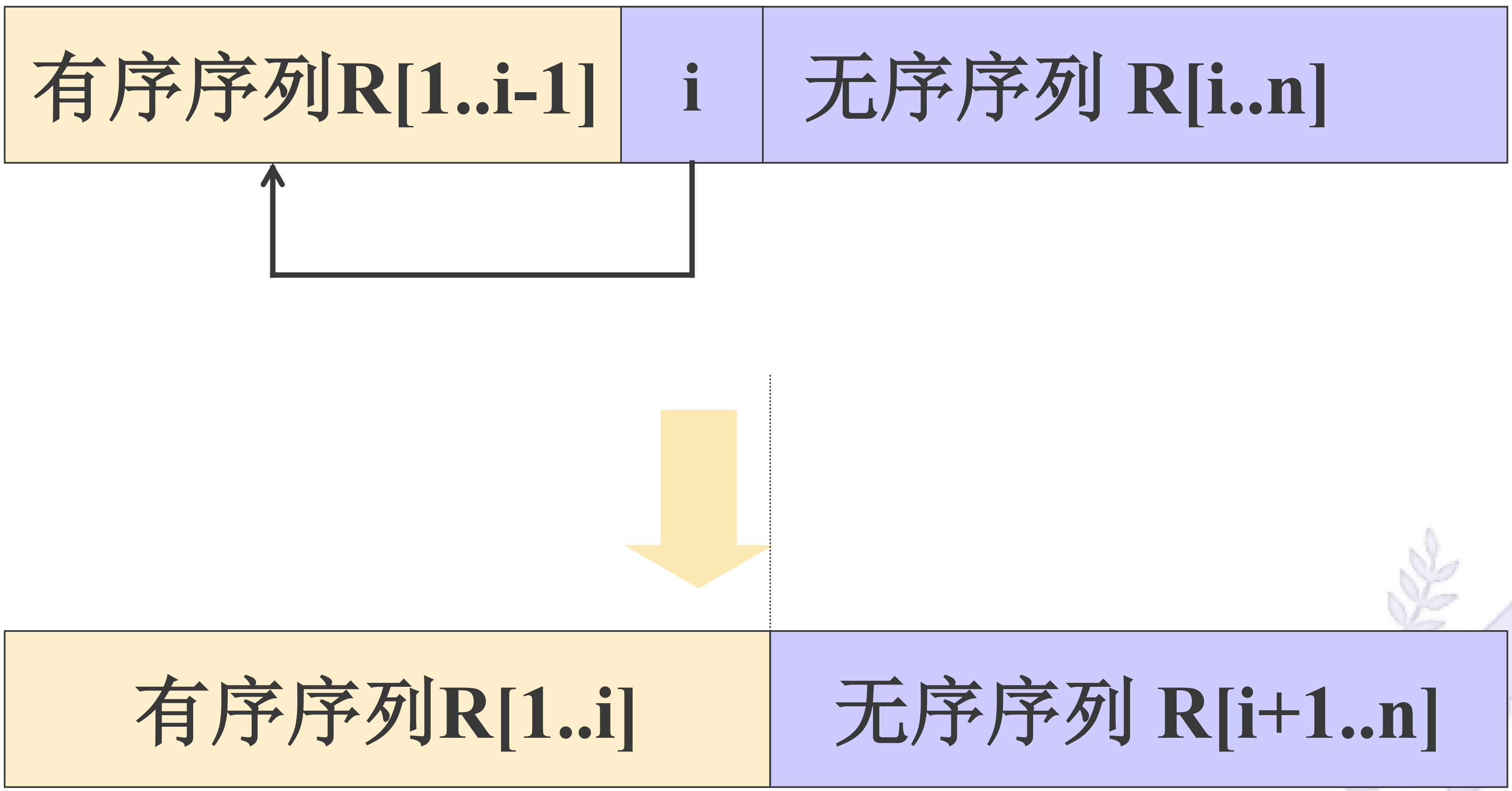
```
#define MAXSIZE  1000 // 待排顺序表最大长度
typedef  int  KeyType;  // 关键字类型为整数类型

typedef  struct {// 记录类型
   KeyType   key;             // 关键字项
   InfoType  otherinfo;  // 其它数据项
} RcdType;

typedef  struct {// 顺序表类型
    RcdType    r[MAXSIZE+1]; // r[0]闲置
    int             length;            // 顺序表长度
} SqList;
```

8.2 插入排序(Insertion Sort)

♦ **8.2.1 插入排序的基本思想**



8.2.1 插入排序的基本思想

1. 在R[1..i-1]中查找R[i]的插入位置：
   R[1..j].key ≤ R[i].key < R[j+1..i-1].key；

2. 将R[j+1..i-1]中的所有记录均后移一个位置；

3. 将R[i] 插入(复制)到R[j+1]的位置上。

8.2.1 插入排序的基本思想

- 不同的具体实现方法导致不同的算法描述
  - 直接插入排序（基于顺序查找）
  - 折半插入排序（基于折半查找）
  - 希尔排序（基于逐趟缩小增量）

8.2.2 直接插入排序

例：R:	0	1	2	3	4	5	6	7	8
		49	38	65	97	76	13	27	49
		（49）	38	65	97	76	13	27	49
	38	（38	49）	65	97	76	13	27	49
	65	（38	49	65）	97	76	13	27	49
	97	（38	49	65	97）	76	13	27	49
	76	（38	49	65	76	97）	13	27	49
	13	（13	38	49	65	76	97）	27	49
	27	（13	27	38	49	65	76	97）	49
	49	（13	27	38	49	49	65	76	97）

直接插入排序的基本思想

- 利用 “顺序查找”实现“在R[1..i-1]中查找R[i]的插入位置”
- 基本思想：
  - 监视哨设置在R[0]； R[0] = R[i];
  - 从R[i-1]起向前顺序查找，直到找到插入位置

```
for (j=i-1; R[0].key < R[j].key; --j);
插入位置是：j+1
```

	j						*i*				
0	1	2	3	4	5	6	7	8			m−1
27	13	38	49	65	76	97	27	49			

直接插入排序的基本思想

- 基本思想：
  - 把从j+1到i-1的所有数据向后移动一位

```
for (j=i-1; R[0].key<R[j].key; --j)
        R[j+1] = R[j];
```

移位和查找可以同时进行

  - 将R[i]插入到j＋1的位置

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

	j						*i*				
0	1	2	3	4	5	6	7	8			m−1
27	13	27	38	49	65	76	97	<u>49</u>			

```
void InsertionSort ( SqList &L ) {
  // 对顺序表 L 作直接插入排序。
   for ( i=2; i<=L.length; ++i ) {
        L.r[0] = L.r[i];            // 复制为监视哨
        for ( j=i-1; L.r[0].key < L.r[j].key;  -- j )
              L.r[j+1] = L.r[j];        // 查找并后移
        L.r[j+1] = L.r[0];        // 插入到正确位置
   }
} // InsertSort
```

直接插入排序的时间分析

- 实现内部排序的基本操作有两个：
  - （1）“比较”序列中两个关键字的大小；
  - （2）“移动”记录。

- 最好情况（关键字在记录序列中顺序有序）：
  - 比较的次数为

$$\sum_{i=2}^{n} 1 = n-1$$

$$\sum_{i=2}^{n} 2 = 2n - 2$$

- 移动的次数为 **0**

◆ 最坏的情况（关键字在记录序列中逆序有序）

¶ 比较的次数

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

– 移动的次数

$$\sum_{i=2}^{n} (i+1) = \frac{(n+4)(n-1)}{2}$$

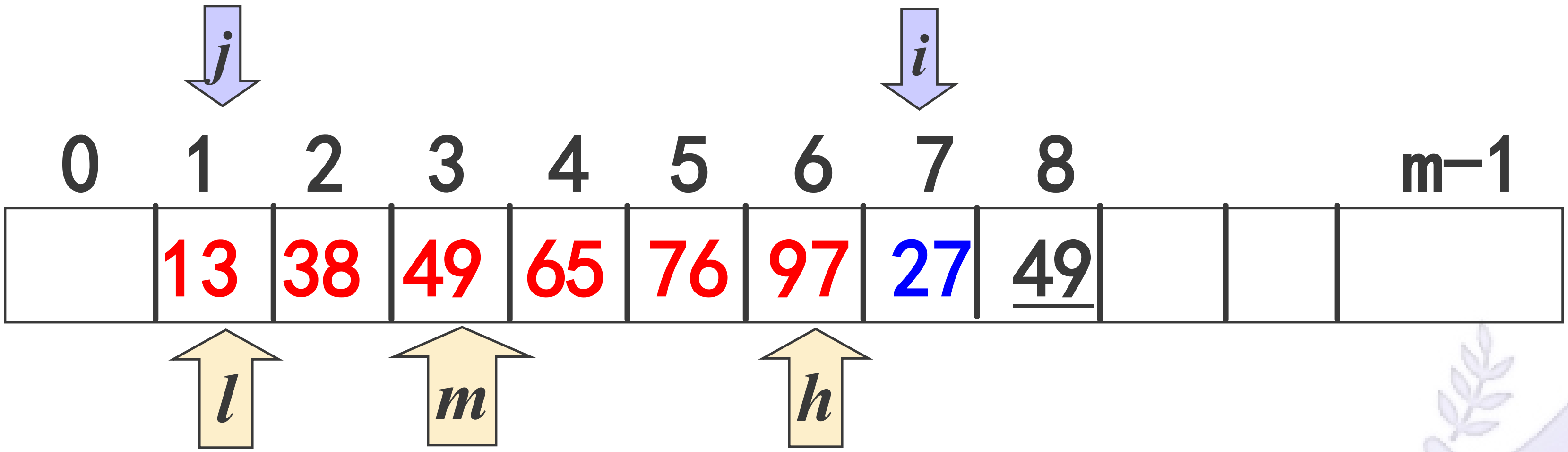
直接插入排序的平均复杂度为O(n^2)

直接插入排序特点

- 算法简单
- 存储结构：顺序、链式
- 时间复杂度为O(n^2)
- 空间复杂度为O(1)
- 稳定
- 适用于：
- （1）若待排序记录按关键码基本有序时，直接插入排序的效率可以大大提高；
- （2）由于直接插入排序算法简单，则在待排序记录数量 n较小 时效率也很高

8.2.3 折半插入排序

- 基本思想：利用折半查找实现“在R[1..i-1]中查找R[i]的插入位置”



8.2.3 折半插入排序

插入位置

i

	1	2	3	4	5	6	7	8	9	10
L.r	14	36	49	52	80	58	61	23	97	75

low m high

low m

low m

high

插入位置：low

8.2.3 折半插入排序

插入位置

i

	1	2	3	4	5	6	7	8	9	10
L.r	14	36	49	52	80	49	61	23	97	75

low m high

low m

high

插入位置：low

北京理工大学

```
void BInsertionSort ( SqList &L ) {
  // 对顺序表 L 作 折半 插入排序。
   for ( i=2; i<=L.length; ++i ) {
       L.r[0] = L.r[i];            // 复制为监视哨
       low = 1; high = i – 1; //折半查找
       while ( low <= high) {
            m = (low+high)/2;
            if (L.r[0].key<L.r[m].key) high = m-1;
            else low = m+1;
       }//while( low <= high)
       for ( j=i-1; j>=low;  -- j )// 后移
             L.r[j+1] = L.r[j];
       L.r[low] = L.r[0];        // 插入到正确位置
   }
} // InsertSort
```

时间复杂度为$O(n^2)$

8.2.4 希尔排序

- **Shell's sort**
- 又称**缩小增量排序(Diminishing Increment Sort)**

- **基本思想**：对待排记录序列先作“宏观”调整，再作“微观”调整。
  - 1） 对数据分组，在各组内进行直接插入排序；
  - 2） 作若干次使待排记录基本有序；
  - 3） 对全部记录进行一次顺序插入排序；

- 分组的方式
- 将 n 个记录分成 d 个子序列：
  - { R[1]，R[1+d]，R[1+2d]，…，R[1+kd] }
  - { R[2]，R[2+d]，R[2+2d]，…，R[2+kd] }
  - …
  - { R[d]，R[2d]，R[3d]，…，R[kd]，R[(k+1)d] }
- d 称为增量，它的值在排序过程中从大到小逐渐缩小，直至最后一趟排序减为 1

11个元素

1	2	3	4	5	6	7	8	9	10	11
16	25	12	30	47	11	23	36	9	18	31

第一趟希尔排序，设 **d =5**

16	25	12	30	47	11	23	36	9	18	31
11	23	12	9	18	16	25	36	30	47	31

第二趟希尔排序，设 **d = 3**

11	23	12	9	18	16	25	36	30	47	31
9	18	12	11	23	16	25	31	30	47	36

第三趟希尔排序，设 **d = 1**

9	18	12	11	23	16	25	31	30	47	36
9	11	12	16	18	23	25	30	31	36	47

Shell's sort

```
int dlta[] = {5, 3, 1}; //增量序列
int t = 3; //增量数
void ShellSort (SqList &L)
{   // 增量为dlta[]的希尔排序
    for (k=0; k<t; ++k)
        ShellInsert(L, dlta[k]);  //一趟插入排序
} // ShellSort
```

Shell’s sort

```
void ShellInsert ( SqList &L, int dk ) {//dk为增量
//一趟希尔插入排序（顺序插入排序）
    for ( i=dk+1; i<=L.length; ++i ) {
        L.r[0] = L.r[i];            // 暂存在R[0]
        for (j=i-dk;  j>0&&(L.r[0].key<L.r[j].key);  j-=dk)
                L.r[j+dk] = L.r[j];  //查找并后移
        L.r[j+dk] = L.r[0];              // 插入
    }
} // ShellInsert
```

8.3 选择排序(Selection Sort)

◆ 8.3.1 简单选择排序

有序序列R[1..i-1] | 无序序列 R[i..n]

从中选出
关键字最小的记录R[j]

第 i 趟
简单选择排序

有序序列R[1..i] | 无序序列 R[i+1..n]

[49 38 65 97 76 13 27 49]

13[38 65 97 76 49 27 49]

13 27[65 97 76 49 38 49]

13 27 38[97 76 49 65 49]

13 27 38 49[76 97 65 49]

13 27 38 49 49[97 65 76]

13 27 38 49 49 65[97 76]

13 27 38 49 49 65 76 97

简单选择排序是不稳定的！

```
void SelectSort (Elem R[], int n ) {
  // 对记录序列R[1..n]作简单选择排序。
  for (i=1; i<n; ++i) {
     // 选择第 i 小的记录，并交换到位
     j = SelectMinKey(R, i, n);
            // 在 R[i..n] 中选择关键字最小的记录
     if (i!=j)  R[i]←→R[j];
            // 与第 i 个记录交换
  }
} // SelectSort
```

简单选择排序时间性能分析

- 对 **n** 个记录进行简单选择排序：
- 关键字间的比较次数 总计为

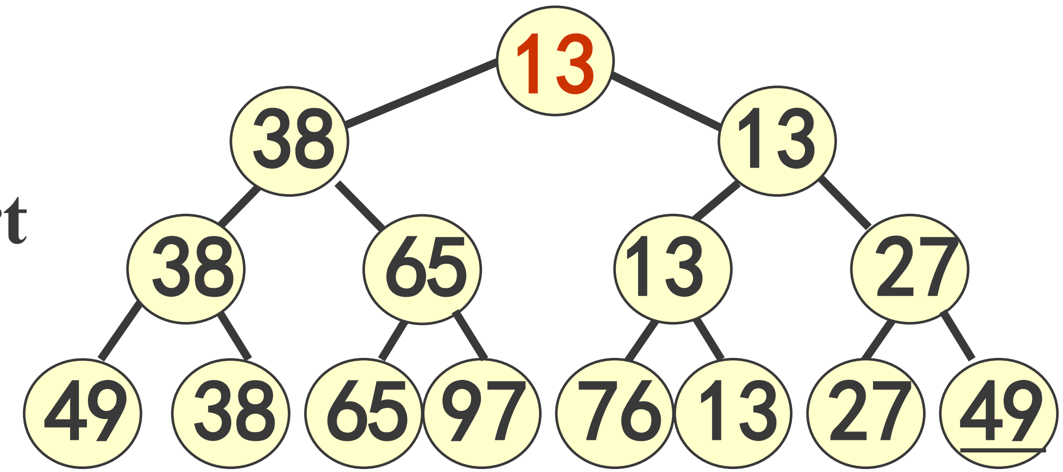
$$\sum_{i=1}^{n-1}(n-i)=\frac{n(n-1)}{2}$$

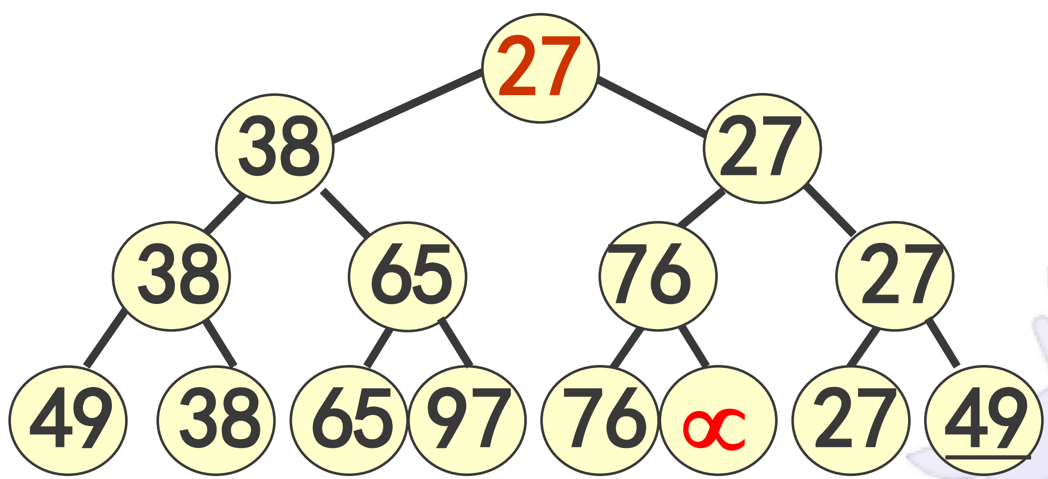
- 移动记录的次数
  - 最小值为 **0**, 最大值为**3(n-1)** 。
- 时间复杂度为**O(n²)**
- 如何减少比较次数？
  - 利用之前比较的结果

8.3.2 树型选择排序 Tree Selection Sort

锦标赛排序

Tournament Sort

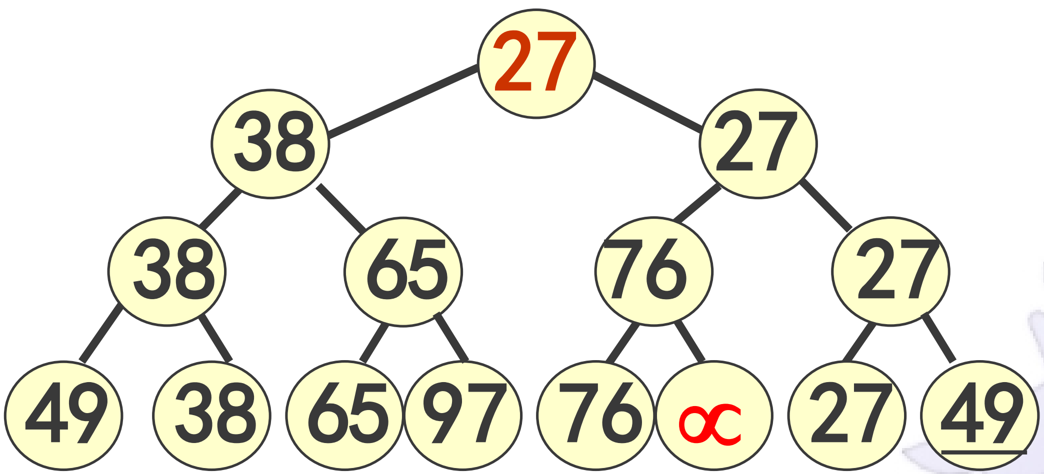




8.3.2 树型选择排序 Tree Selection Sort

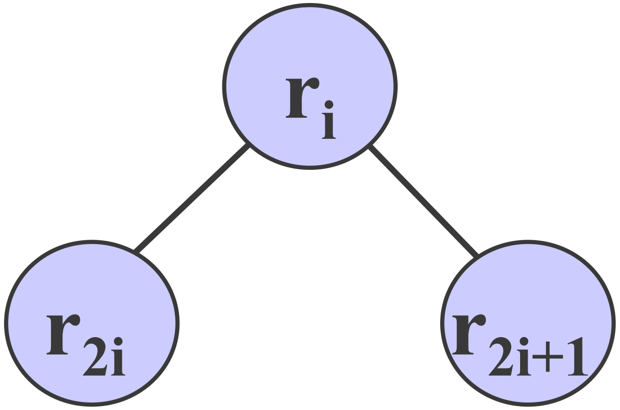
- 选择最小关键字：比较n-1次
- 选择其它当前最小关键字：$\lceil logn \rceil$
- 排序的时间复杂度：O(nlogn)
- 空间复杂度 O(n)

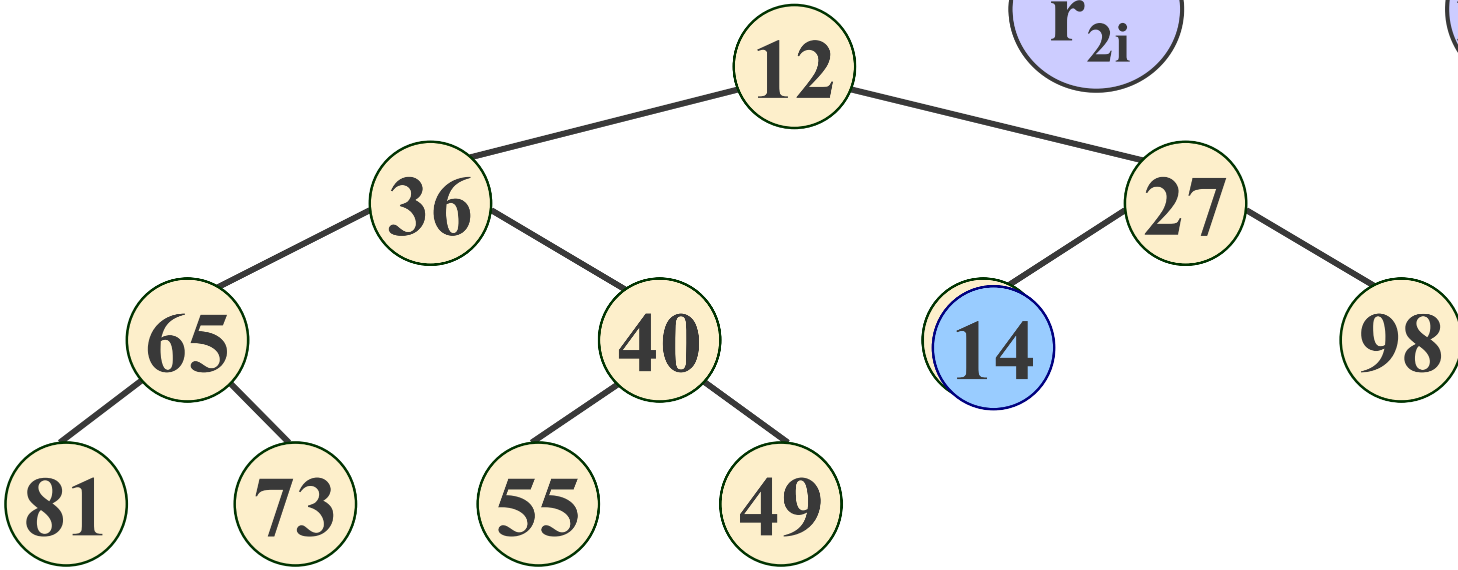
完全二叉树



8.3.3 堆排序Heap Sort

- 完全二叉树:
  - r_{2i} 是 r_i 的左孩子;
  - r_{2i+1} 是 r_i 的右孩子。





堆：R[11]{12, 36, 27, 65, 40, 34, 98, 81, 73, 55, 49}

R[11]{12, 36, 27, 65, 40, 14, 98, 81, 73, 55, 49}不是堆

8.3.3 堆排序

◆ 堆：是满足下列性质的数列{r1, r2, …， rn}：

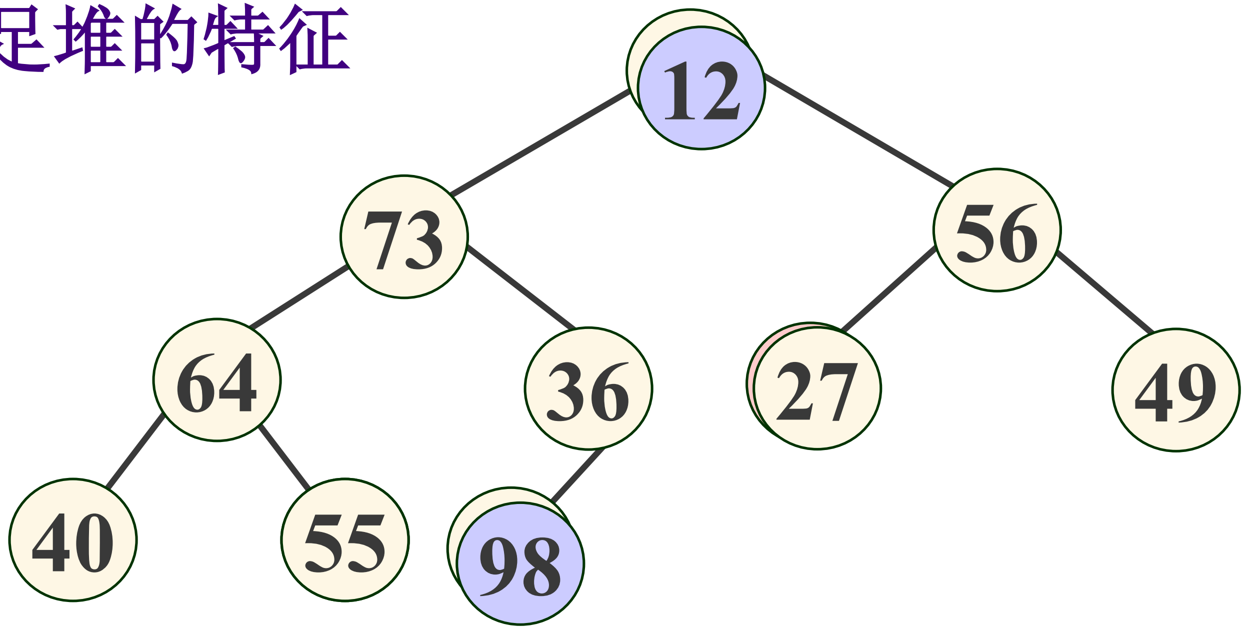
$$\begin{cases} r_i \le r_{2i} \\ r_i \le r_{2i+1} \end{cases} \text{(小顶堆)} \quad \text{或} \quad \begin{cases} r_i \ge r_{2i} \\ r_i \ge r_{2i+1} \end{cases} \text{(大顶堆)}$$

```
typedef  SqList HeapType;
                         // 堆采用顺序表表示
```

```
typedef  struct {
    ElemType *R;        // 存储空间基址
    int      length;   // 当前长度
    int      listsize; // 当前分配的存储容量
                       // (以sizeof(ElemType)为单位)
} SqList;  // 俗称顺序表
```

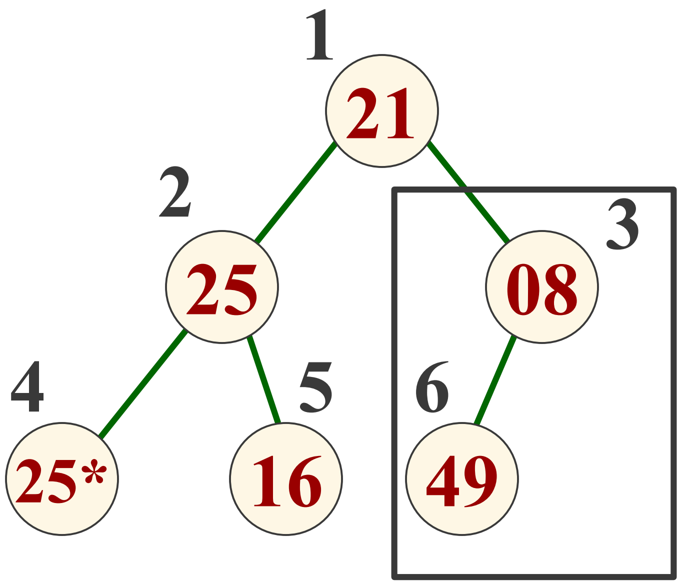
堆排序的一般过程（大顶堆）

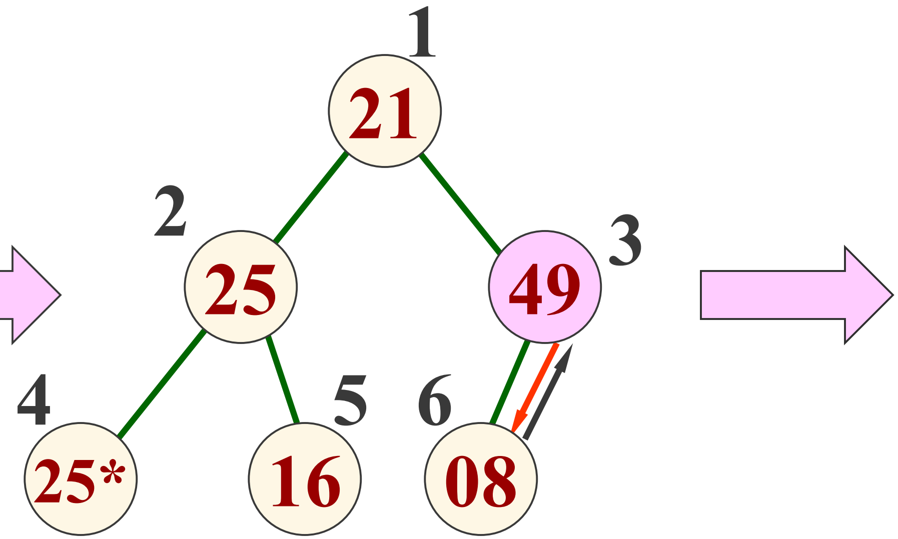
- 1）建立初始堆；
- 2）将堆顶元素与最后一个元素对换；
- 3）调整堆
  - 即假设H.R[s..m]中记录的关键字除 R[s] 之外均满足堆的特征



堆：R[10]{12, 73, 56, 64, 36, 27, 49, 40, 55} {98}

建立初始的最大堆





21	25	08	25*	16	49
1	2	3	4	5	6

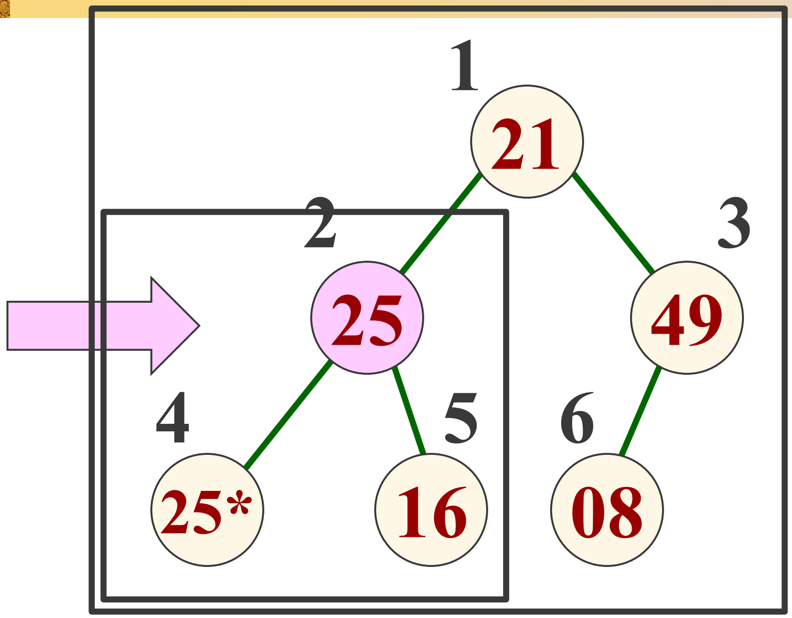
初始排序码集合

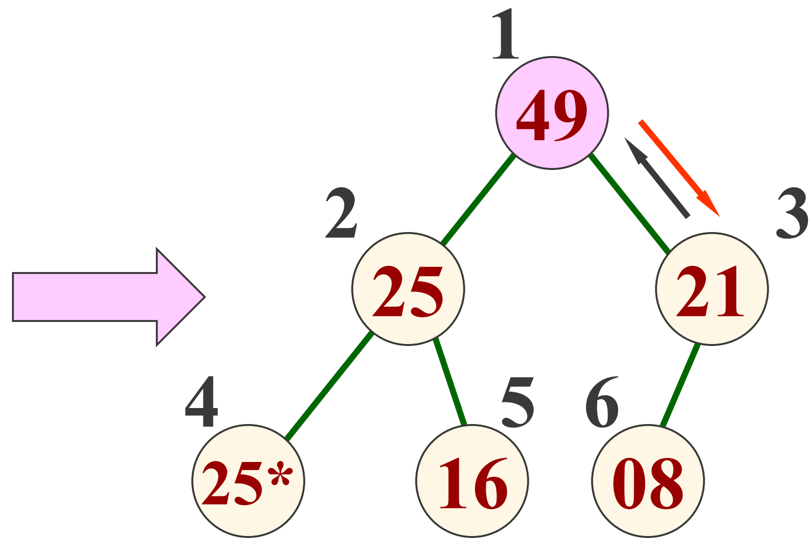
21	25	49	25*	16	08
1	2	3	4	5	6

i = 3

HeapAdjust (H, 3, H.length)

建立初始的最大堆





21	25	49	25*	16	08
1	2	3	4	5	6

i = 2

HeapAdjust (H, 2, H.length)

49	25	21	25*	16	08
1	2	3	4	5	6

i = 1.形成最大堆

HeapAdjust (H, 1, H.length)

建立初始的最大堆

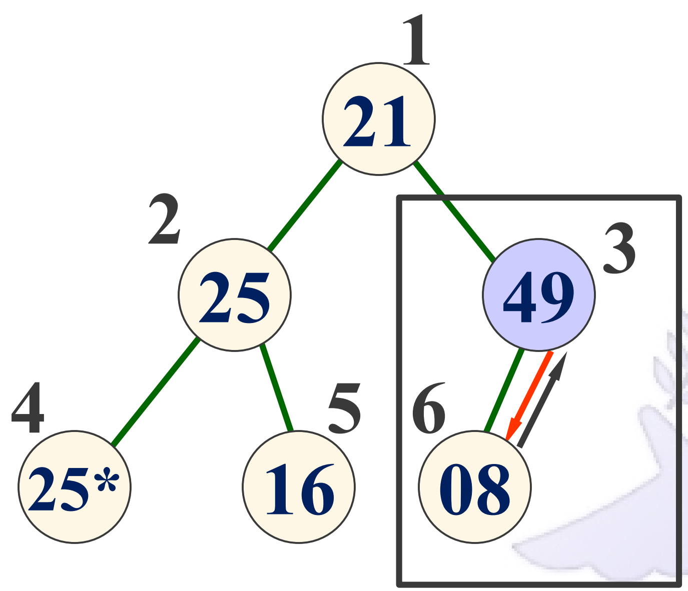
◆ 建立最大堆：

```
for ( i=H.length/2;   i>0;   --i )

        HeapAdjust ( H.R, i, H.length );
```

i = 3

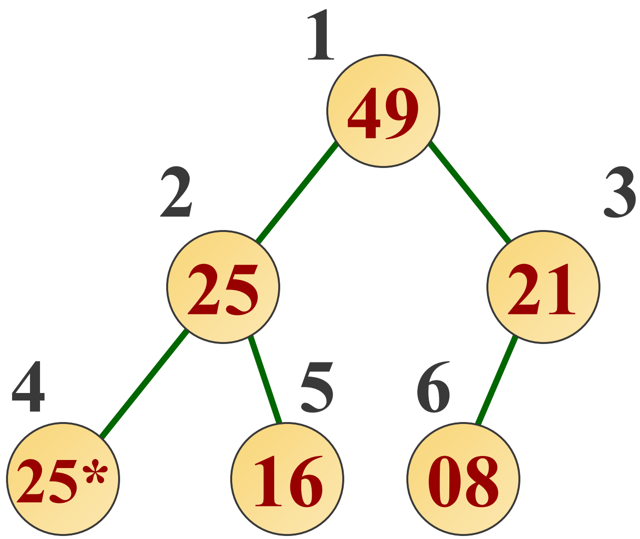
21	25	49	25*	16	08
1	2	3	4	5	6

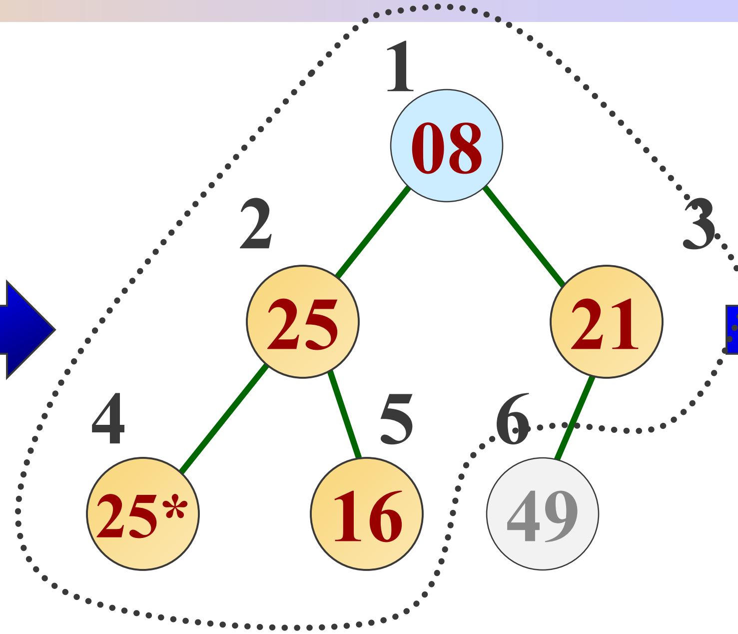


基于初始堆进行堆排序

1. 最大堆堆顶r[1]具有最大的排序码, 将r[1]与 r[n]对调, 把具有最大排序码的对象交换到最后
2. 对前面的n-1个对象, 使用堆的调整算法重新建立最大堆, 具有次最大排序码的对象上浮到r[1]位置。称为筛选。
3. 对调r[1]和r[n-1],调用调整算法, 对前n-2个对象重新调整。
4. 如此反复执行，最后得到全部排序好的对象序列。

堆排序示例





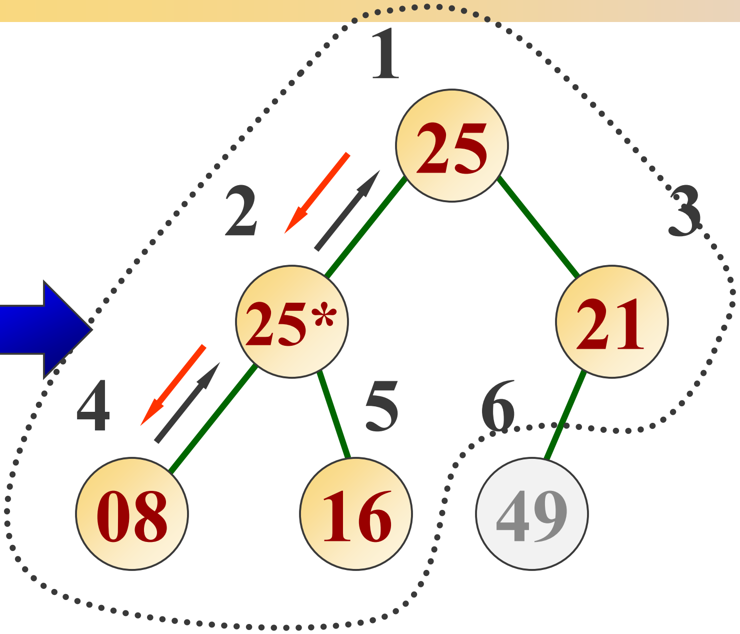
1	2	3	4	5	6
49	25	21	25*	16	08

初始最大堆

1	2	3	4	5	6
08	25	21	25*	16	49

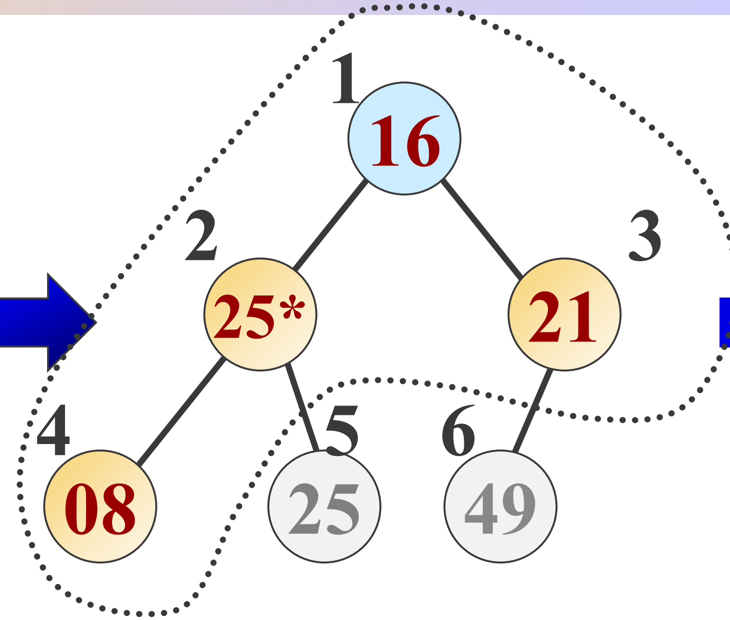
交换 1 号与 6 号对象，
6 号对象就位

堆排序示例



25	25*	21	08	16	49
1	2	3	4	5	6

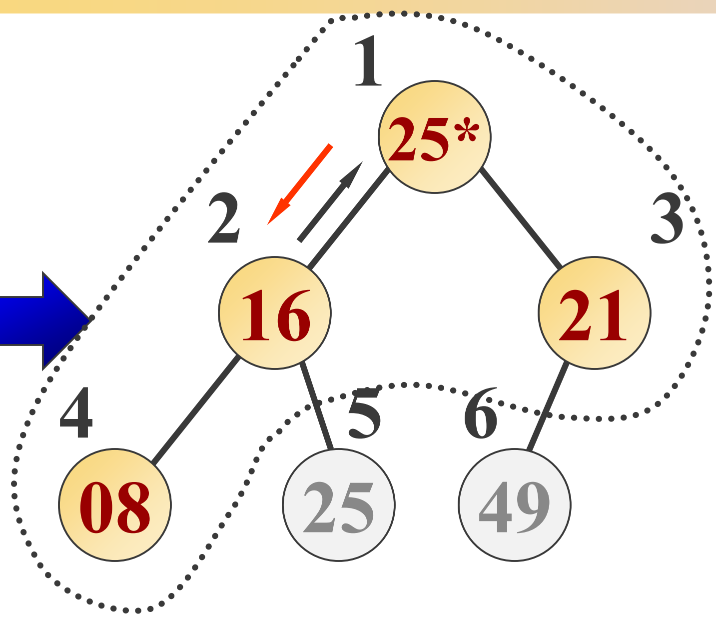
从 1 号到 5 号 重新调整为最大堆



16	25*	21	08	25	49
1	2	3	4	5	6

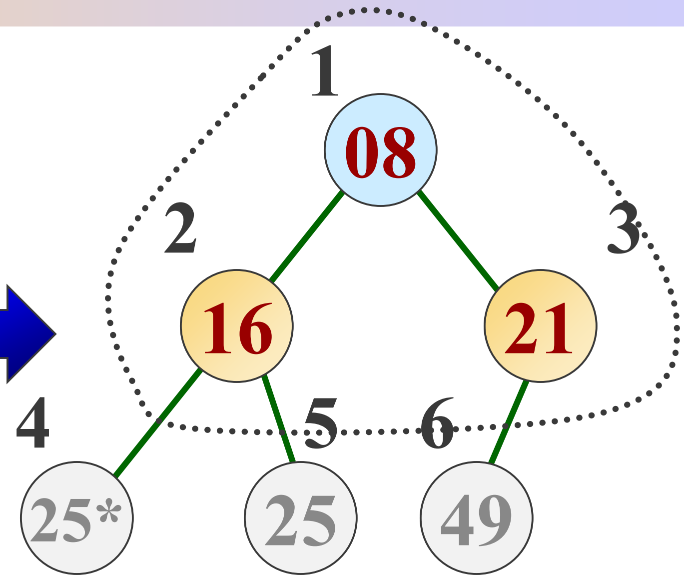
交换 1 号与 5 号对象，5 号对象就位

堆排序示例



25*	16	21	08	25	49
1	2	3	4	5	6

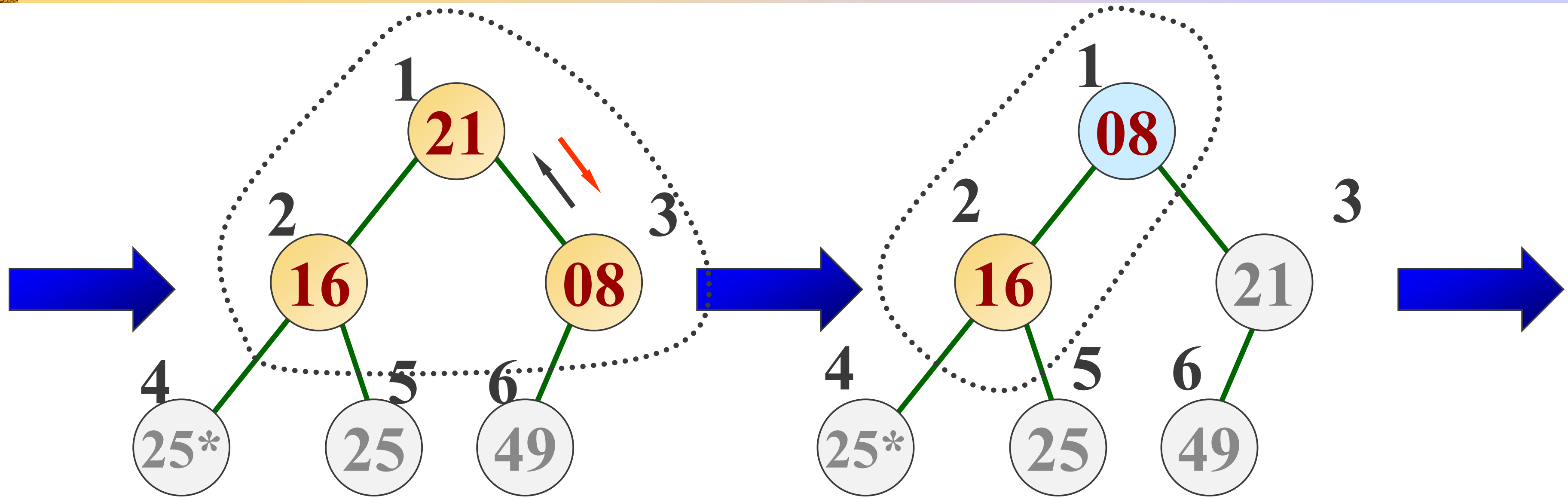
从 1 号到 4 号 重新调整为最大堆



08	16	21	25*	25	49
1	2	3	4	5	6

交换 1 号与 4 号对象，4 号对象就位

堆排序示例



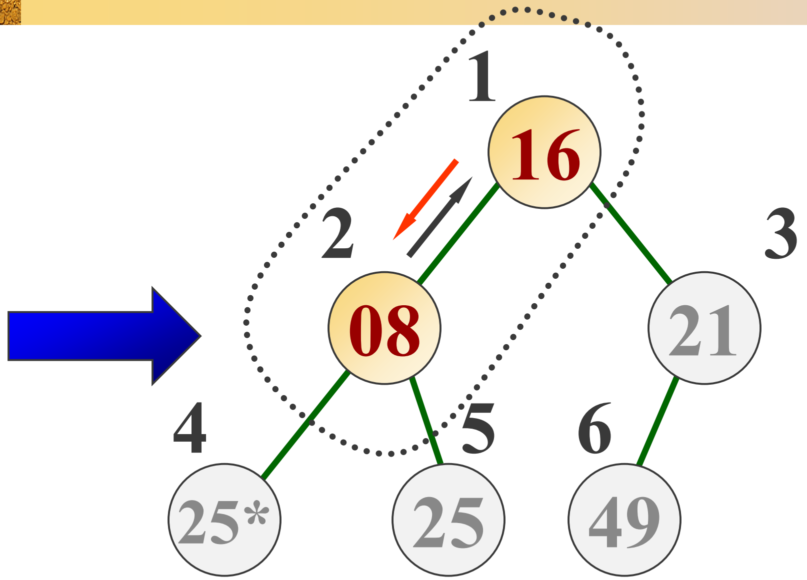
21	16	08	25*	25	49
1	2	3	4	5	6

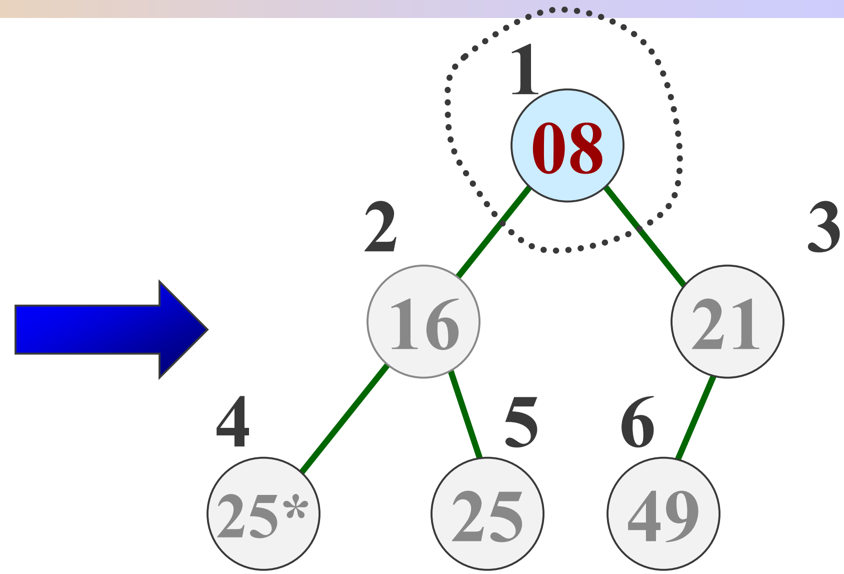
从 1 号到 3 号 重新调整为最大堆

08	16	21	25*	25	49
1	2	3	4	5	6

交换 1 号与 3 号对象，3 号对象就位

堆排序示例





16	08	21	25*	25	49
1	2	3	4	5	6

08	16	21	25*	25	49
1	2	3	4	5	6

从 1 号到 2 号 重新调整为最大堆

交换 1 号与 2 号对象，2 号对象就位

```
void HeapSort ( HeapType &H ) {
  // 对顺序表 H 进行堆排序
   for ( i=H.length/2;   i>0;   --i ) // 建大顶堆

        HeapAdjust ( H.R, i, H.length );
   for ( i=H.length; i>1; --i ) {
         // 将堆顶记录和当前未经排序子序列
         //  最后一个记录相互交换
       H. R[1]←→H.R[i];
       HeapAdjust(H.R, 1, i-1);  // 对 H.R[1] 进行筛选
   }
} // HeapSort
```

堆排序的时间复杂度分析

- 1. 含有n 个关键字的完全二叉树的深度h=$\lfloor \log_2 n \rfloor$+1。
- 2. 对深度为 h (h=$\lfloor \log_2 n \rfloor$+1)的堆，“筛选”所需进行的关键字比较的次数至多为2(h-1) = 2$\lfloor \log_2 n \rfloor$；
- 3. 建堆的复杂度：对 n 个关键字，建立深度为h (h=$\lfloor \log_2 n \rfloor$+1)的堆，所需进行的关键字比较的次数t：
  - 建立初始堆需要对$\lfloor n/2 \rfloor$棵子树进行调整
  - 第i层上的节点至多为2^{i-1}个，以它们为根的子树深度为：h-i+1

$$t=\sum_{i=h-1}^{1}2^{i-1}\cdot 2(h-i)=\sum_{i=h-1}^{1}2^{i}\cdot(h-i)$$

（令$j=h-i$）

$$=\sum_{j=1}^{h-1}2^{h-j}\cdot j=\sum_{j=1}^{h-1}2^{h}\cdot 2^{-j}\cdot j\le(2n)\sum_{j=1}^{h-1}j/2^{j}\le 4n$$

数列1/2，2/4，3/8，……n/2^n，……的前n项和 = 2-(n+2)/2^n

堆排序的时间复杂度分析

- 0. 对于有n个节点的堆，“筛选”所需进行的关键字比较的次数至多为 $2\lfloor \log_2 n \rfloor$；
- 1. 建立初始的最大堆的比较次数<=4n;
- 2. 排序：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，堆排序的最坏时间复杂度为O(nlogn)。

堆排序的特点

- 堆排序的最坏时间复杂度为O(nlogn)。
- 空间复杂度：1个记录空间，O(1)
- 稳定性：不稳定
- 适合于n较大的情况。

堆与优先队列

- 优先队列的应用
  - **Huffman树**
  - **最小生成树**
  - **迪杰斯特拉算法、A*算法**
  - **…**

堆与优先队列

- .1 优先队列的定义
  - 优先队列中的每一个元素都有一个优先级值。
  - 通常约定优先级值小的优先级高
- 优先队列支持的基本运算有：
  - **(1)Size( )**：返回优先队列中元素个数。
  - **(2)Min( )**：返回优先队列中最小优先级值元素。
  - **(3)Insert(x)**：将元素x插入优先队列。
  - **(4)DeleteMin(x)**：删除优先队列中具有最小优先级值的元素，并保存到x中。

堆与优先队列

- .2 优先队列的实现方式:
  - 顺序表
  - 链表
  - 二叉排序树、AVL树
  - 优先级树
  - 堆
  - 左偏树
  - …

8.3 交换排序

- **基本思想：** 将待排记录中两两记录关键字进行比较，若逆序则交换位置。

- 8.3.1 起泡排序
- 49 38 65 97 76 13 27 <u>49</u>
  - 38 49 65 76 13 27 <u>49</u> 97
  - 38 49 65 13 27 <u>49</u> 76 97
  - 38 49 13 27 <u>49</u> 65 76 97
  - 38 13 27 49 <u>49</u> 65 76 97
  - 13 27 38 49 <u>49</u> 65 76 97
  - 13 27 38 49 <u>49</u> 65 76 97

8.3.1 起泡排序

```
Void bubble-sort (int a[]， int n)
{ //起泡排序，从小到大排列
    for( i=n-1,change=TURE;  i>1 && change;- -i)
    {
        change=false;
        for( j=0;j<i; ++j)
          if ( a[j]>a[j+1]) {
              a[j] ←→a[j+1];
              change=TURE;
          }
    }
}
```

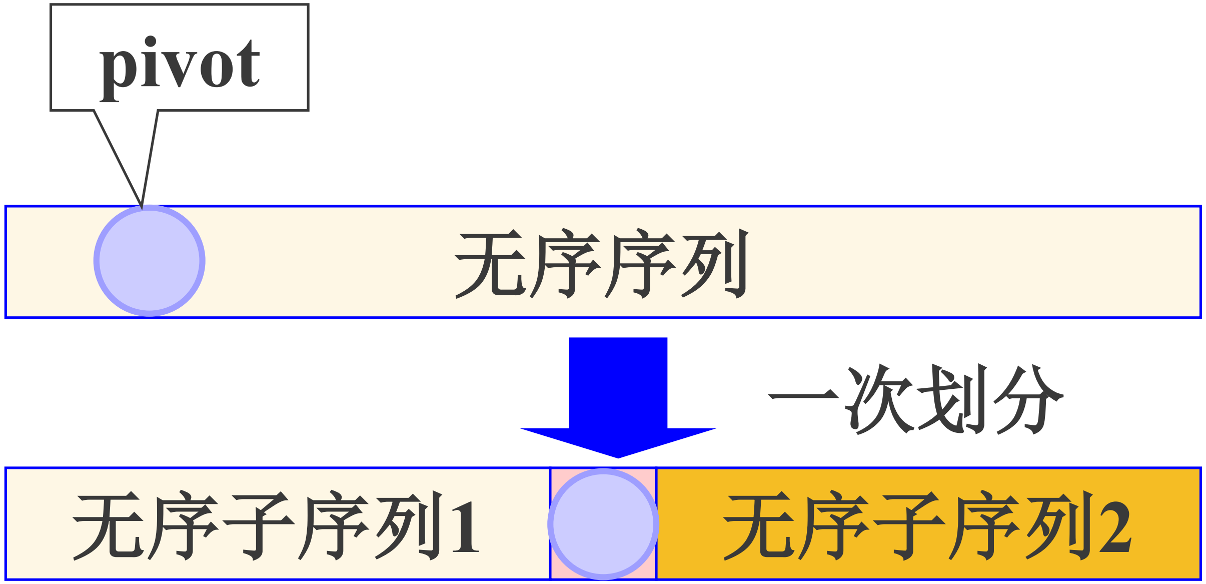
起泡排序分析

- 正序：
  - 比较(n-1)次
  - 不移动记录
- 逆序：
  - 比较(n-1)+(n-2)+…+1=n(n-1)/2次
  - 交换n(n-1)/2次
- 时间复杂度$O(n^2)$
- 空间复杂度O(1)
- 稳定性：稳定

8.3.2 快速排序

- **基本思想：**
  - 通过一趟排序将待排序记录分割成两个部分
  - 一部分记录的关键字比另一部分的小。
  - 选择一个关键字作为分割标准，称为pivot
- **基本操作：**
  - 选定一记录R（pivot），将所有其他记录关键字k’与该记录关键字k比较
    - 若 k’<k则将记录换至R之前；
    - 若k’ >k 则将记录换至R之后;
  - 继续对R前后两部分记录进行快速排序，直至排序范围为1;

8.3.2 快速排序



49 38 65 97 76 13 27 <u>49</u>

<u>27 38 13</u> 49 <u>76 97 65 49</u>

继续划分

用第一个记录作pivot

$\boxed{49}$ 49 38 65 97 76 13 27 <u>49</u>

l h h

27 38 65 97 76 13 27 <u>49</u>

l l h

27 38 65 97 76 13 65 <u>49</u>

l h h

27 38 13 97 76 13 65 <u>49</u>

l l h

27 38 13 97 76 97 65 <u>49</u>

l h h

27 38 13 49 76 97 65 <u>49</u>

l h

一趟快速排序

[27 38 13] **49** [76 97 65 <u>49</u>] 一趟快速排序

[13] **27** [38] **49** [<u>49</u> 65] **76** [97] 两趟快速排序

13 27 38 49 <u>49</u> 65 76 97 三趟快速排序

快速排序

8.3.2 快速排序

- 设初始时
  - low指针指向第一个记录;
  - high指针指向最后一个记录;
- 一趟快速排序的算法过程:

1. 将第一个记录设置为pivot
2. 从表的两端交替地向中间扫描，直到两个指针相遇
   - 先从高端扫描
     - 找到第一个比pivotkey小的记录
     - 将该记录移动到low指针指向的地方;
   - 再从低端扫描
3. 将pivot移动到low指针位置，并返回该位置

8.3.2 快速排序

```
int Partition(SqList &L, int low, int high)
{ /*对顺序表L中子表r[low..high]的记录
  作一趟快速排序，并返回pivot记录所在位置。*/
    L.r[0]=L.r[low]; //用第一个记录作pivot记录
    pivotkey=L.r[low].key; // pivotkey是pivot关键字

    while(low<high)
    { //从表的两端交替地向中间扫描
         while(low<high && L.r[high]. Key>=pivotkey)
                  --high;
         L.r[low]=L.r[high];
         while (low<high && L.r[low]. Key<=pivotkey)
                 ++low;
         L.r[high]=L.r[low];        } // 交替扫描结束

    L.r[low]=L.r[0]; //pivot位置
    return low;         //返回pivot位置

}//Partition
```

8.3.2 快速排序

```
void Qsort(SqList &L, int low, int high)
{//对顺序表L中的子序列L.r[low.. high]作快速排序
  if (low<high)//递归结束条件
  {      pivotloc=Partition(L, low, high);
         QSort(L, low, pivotloc-1);
         Qsort(L, pivotloc+1, high);
  }
}
```

```
void QuickSort(SqList &L )
{//对顺序表L快速排序
   QSort(L, 1, L.length);
}
```

快速排序特点

- 存储结构：顺序
- 时间复杂度
  - 最坏情况：每次划分选择pivot是最小或最大元素
  - 最坏情况： $O(n^2)$
  - 最好情况（每次划分折半）： $O(n\log_2 n)$
  - 平均时间复杂度为$O(n\log_2 n)$
- 空间复杂度
  - 最坏情况：$O(n)$
  - 最好情况（每次划分折半）： $O(\log_2 n)$
  - 平均空间复杂度$O(\log_2 n)$
- 稳定性
  - 不稳定

改进的快速排序

- 改进1：小序列用直接插入排序

```
void  Quicksort( ElementType A[ ], int N )
{        Qsort( A, 0, N - 1 );
}// Quicksort
```

```
void  Qsort( ElementType A[ ], int Left, int Right )
{   int  i,  j;
    ElementType  Pivot;
    if ( Left + Cutoff >= Right )
         InsertionSort( A + Left, Right - Left + 1 );
    else
         {/*改进的快速排序*/}
}
```

/* if the sequence is too short */

改进的快速排序

- 改进2：尽量将pivot取在中间位置
  - 三平均分区法（median-of-three） Low, center, high 指示的记录关键字“三值”取中

L　　　　　　　　C　　　　　　　　R

99　17　65　97　45　13　27　87　34

对L、C、R指示三记录进行排序

34　17　65　97　45　13　27　87　99

交换C和R-1指示记录

34　17　65　97　87　13　27　45　99

将R-1作为pivot

只需对L+1到R-2进行比较

L

R

找第一个比pivot大的记录

34 17 65 97 87 13 27 45 99

i j

找第一个比pivot小的记录

34 17 65 97 87 13 27 45 99

i j

交换i、j记录

34 17 27 97 87 13 65 45 99

i j

交换i、j记录

34 17 27 13 87 97 65 45 99

j i

将pivot移到*i*位

34 17 27 13 45 97 65 87 99

一趟快速排序

递归的对左右两个部分继续进行快速排序

```
/*改进的快速排序*/
Pivot = Median3( A, Left, Right );  /* 选择枢轴*/
i = Left+1;     j = Right – 2;
for( ; ; ) {
        while ( A[i + +] < Pivot ) { }  /* 从左扫描 */
        while ( A[j - - ] > Pivot ) { }  /* 从右扫描 */
        if ( i < j )
                Swap( &A[ i ], &A[ j ] );  /* 部分调整 */
         else     break;  /* 调整完成 */
}
Swap( &A[ i ], &A[ Right - 1 ] ); /* 存储pivot*/
Qsort( A, Left, i - 1 ); /* 递归的对左边进行快速排序 */
Qsort( A, i + 1, Right ); /*递归对右边进行快速排序 */
```

改进的快速排序

- 对于三平均分区法还可以进一步扩展
  - median-of-(2t+1)：在选取中轴值时，可以从由左中右三个中选取扩大到五个元素中或者更多元素(2t+1)中选取。
  - median
- 改进3：当序列中有许多相同元素时，某些分区的所有元素值可能都相等
  - 划分三个区间
  - 一块是小于中轴值的所有元素；
  - 一块是等于中轴值的所有元素；
  - 另一块是大于中轴值的所有元素
- ……

8.3.2 快速排序分析

- 快速排序的基本思想是基于分治策略的。对于输入的子序列L[p..r]，如果规模足够小则直接进行排序（比如用前述的冒泡、选择、插入排序均可），否则分三步处理：
- 1、分解(Divide)：将待排序列L[p..r]划分为两个非空子序列L[p..q]和L[q+1..r]，使前面任一元素的值不大于后面元素的值。
- 途径实现：在序列L[p..r]中选择数据元素L[q]，经比较和移动后，L[q]将处于L[p..r]中间的适当位置，使得数据元素L[q]的值小于L[q+1..r]中任一元素的值。

- **2、递归求解(Conquer)**：通过递归调用快速排序算法，分别对L[p..q]和L[q+1..r]进行排序。
- **3、合并(Merge)**：由于对分解出的两个子序列的排序是就地进行的，所以在L[p..q]和L[q+1..r]都排好序后不需要执行任何计算L[p..r]就已排好序，即自然合并。
- 这个解决流程是符合分治法的基本步骤的。因此，快速排序法是分治法的经典应用实例之一。

内部排序方法的分类

- 根据设置有序序列的方式的不同，分为：
  - 插入排序：直接插入排序、折半插入排序、希尔排序
  - 交换排序：冒泡排序、快速排序
  - 选择排序：简单选择排序、堆排序
  - 归并排序：2-路归并排序
  - 基数排序

8.5 归并排序(Merging Sort)

- 归并：
  - 将两个或两个以上有序表组合成一个新的有序表。
- 2-路归并排序：
  - 设初始序列含有n个记录，则可看成 n 个有序的子序列，每个子序列长度为1。
  - 两两合并，得到 $\left\lceil \frac{n}{2} \right\rceil$个长度为 2 或1的有序子序列。
  - 再两两合并，……如此重复，直至得到一个长度为 n 的有序序列为止。

8.5 归并排序

例

初始关键字： [49] [38] [65] [97] [76] [13] [27]

一趟归并后： [38 49] [65 97] [13 76] [27]

二趟归并后： [38 49 65 97] [13 27 76]

三趟归并后： [13 27 38 49 65 76 97]

8.5 归并排序

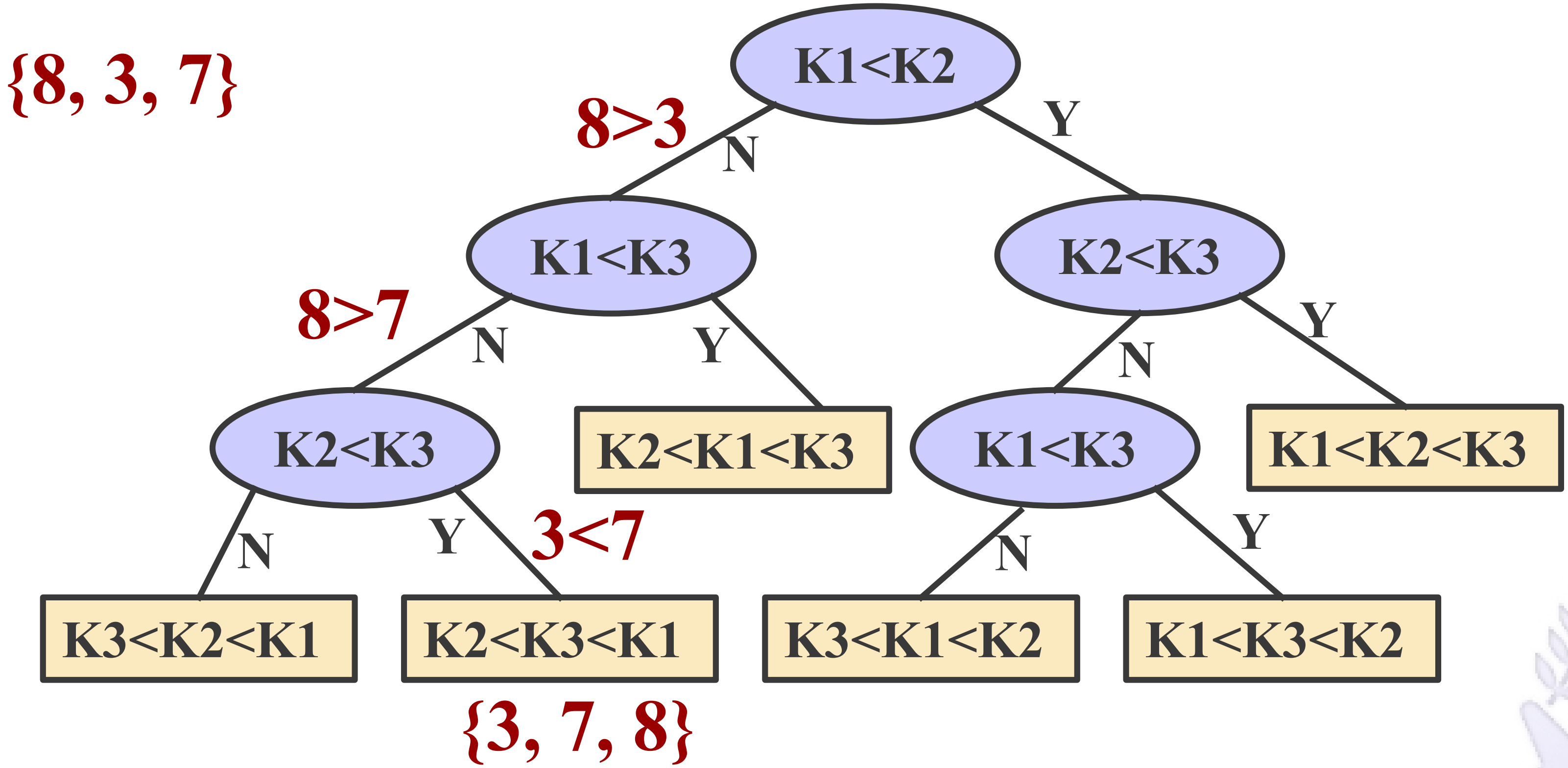
- 时间复杂度：
  - 共进行$\lceil \log_2 n \rceil$趟归并，每趟对n个记录进行归并
  - 所以时间复杂度是O(nlogn)
- 空间复杂度：
  - O(n)
- 稳定性：
  - 稳定

基于比较操作的内排算法分析

	排序方法	最好时间	最坏时间	平均时间	辅助空间	稳定性
1	直接插入	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
2	希尔排序	——	——	——	$O(1)$	不稳定
3	冒泡排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
4	快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	$O(\log_2 n)$	不稳定
5	简单选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
6	堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定
7	归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定

基于比较的内排最快速度是多少？

- 基于关键字比较操作的排序方法可以等价于判断树



3个记录排序，有3! (=6)种可能的排列。

基于比较的内排最快速度是多少？

- n个记录排序，有n!种可能的排列。
- 排序是找到某个叶子节点对应的路径的过程。
- 具有n!个叶子节点的完全二叉树的深度h为：

$$h <= \lceil \log_2 n! \rceil = O(n \log_2 n)$$

- 基于比较操作的排序算法的最坏复杂度最好为：

  $O(n\log_2 n)$

- 在n<11时的比较次数等于$\lceil \log_2 n! \rceil$。

注：斯特林公式： $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

8.6 基数排序(Radix Sorting)

- 基数排序：
  - 借助多关键字排序的方法对单关键字排序。

- 包含多位 k = k1,k2,…,kd 的单关键字
  - 多关键字排序
- 最高位优先（MSD：Most Significant Digit first）
- 最低位优先（LSD：Least Significant Digit first）

8.6 基数排序

例：对52张扑克牌排序，花色优先

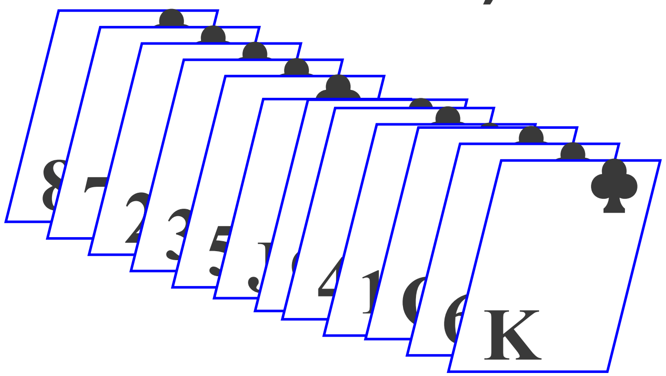
< ♣,2> < ♣,3> … < ♣,K> < ♣,A>
< ♦,2> < ♦,3> … < ♦,K> < ♦,A>
< ♥,2> < ♥,3> … < ♥,K> < ♥,A>
< ♠,2> < ♠,3> … < ♥,K> < ♠,A>

排序方法

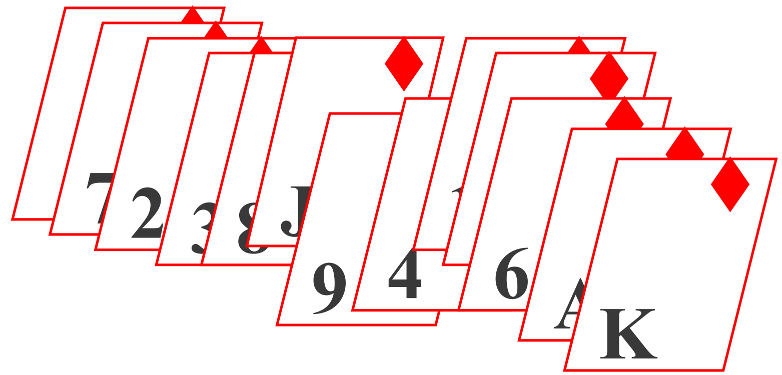
- 先按花色分类，再按面值分类——最高位优先
- 先按面值分类，再按花色分类——最低位优先

8.6 基数排序

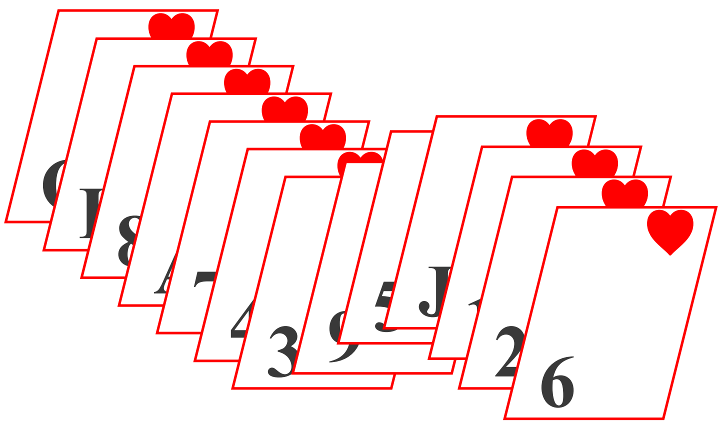
- 最高位优先: 1) 先按照花色分为4堆



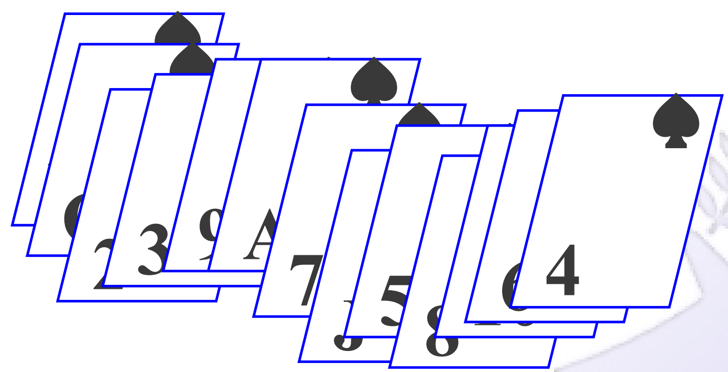
1) 梅花：13 张



2) 方块：13 张



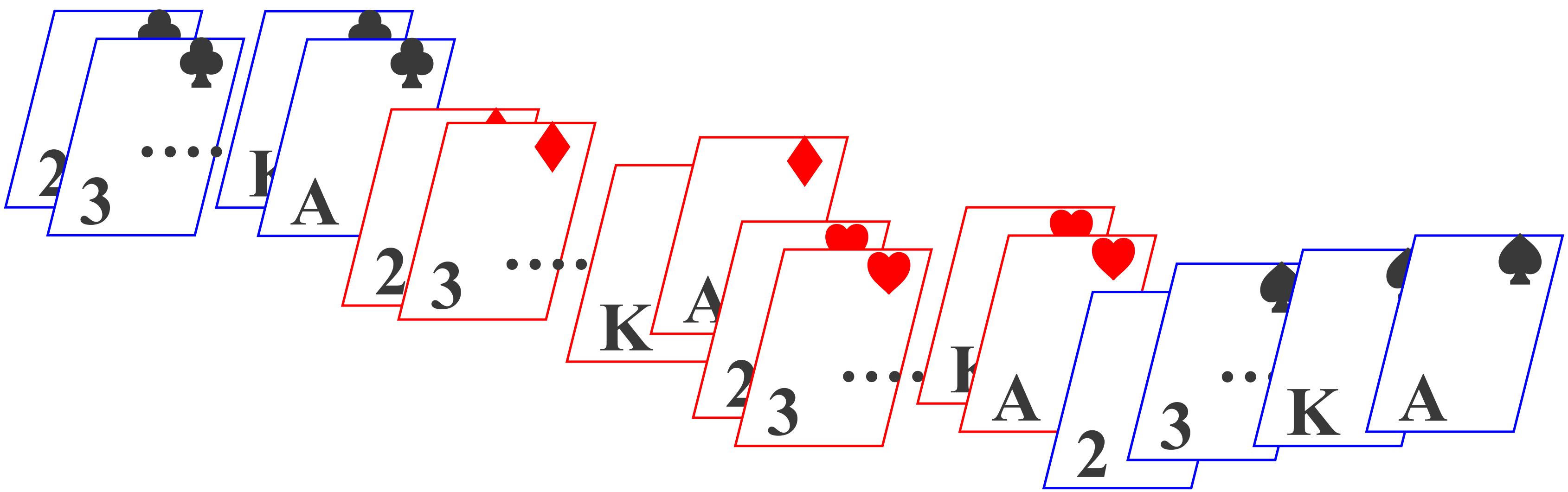
3) 红桃：13 张



4) 黑桃：13 张

8.6 基数排序

◆ 2) 每一堆按面值从小到大排列



< ♣,2> < ♣,3> … < ♣,K> < ♣,A>

< ♦,2> < ♦,3> … < ♦,K> < ♦,A>

< ♥,2> < ♥,3> … < ♥,K> < ♥,A>

< ♠,2> < ♠,3> … < ♥,K> < ♠,A>

8.6 基数排序

- 最低位优先

1) 按照面值分配（分成13组）

< ♣,2>	< ♦,3>		<♥,K>	<♠,A>
< ♦,2>	< ♣,3>		<♦,K>	<♣,A>
< ♥,2>	< ♠,3>	......	<♠,K>	<♥,A>
< ♠,2>	< ♥,3>		<♣,K>	<♦,A>

收集（按面值有序）

< ♣,2>, <♦,2>, <♥,2>,<♠,2>,<♦,3>,< ♣,3>, <♠,3>, <♥,3>,

....

<♥,K>,<♦,K>,<♠,K>,<♣,K>,<♠,A>,<♣,A>,<♥,A>,<♦,A>

8.6 基数排序

< ♣,2>, <♦,2>, <♥,2>,<♠,2>,<♦,3>,< ♣,3>, <♠,3>, <♥,3>,

……

<♥,K>,<♦,K>,<♠,K>,<♣,K>,<♠,A>,<♣,A>,<♥,A>,<♦,A>

2) 按花色分配（分成4组）

按照上述顺序放入4组中

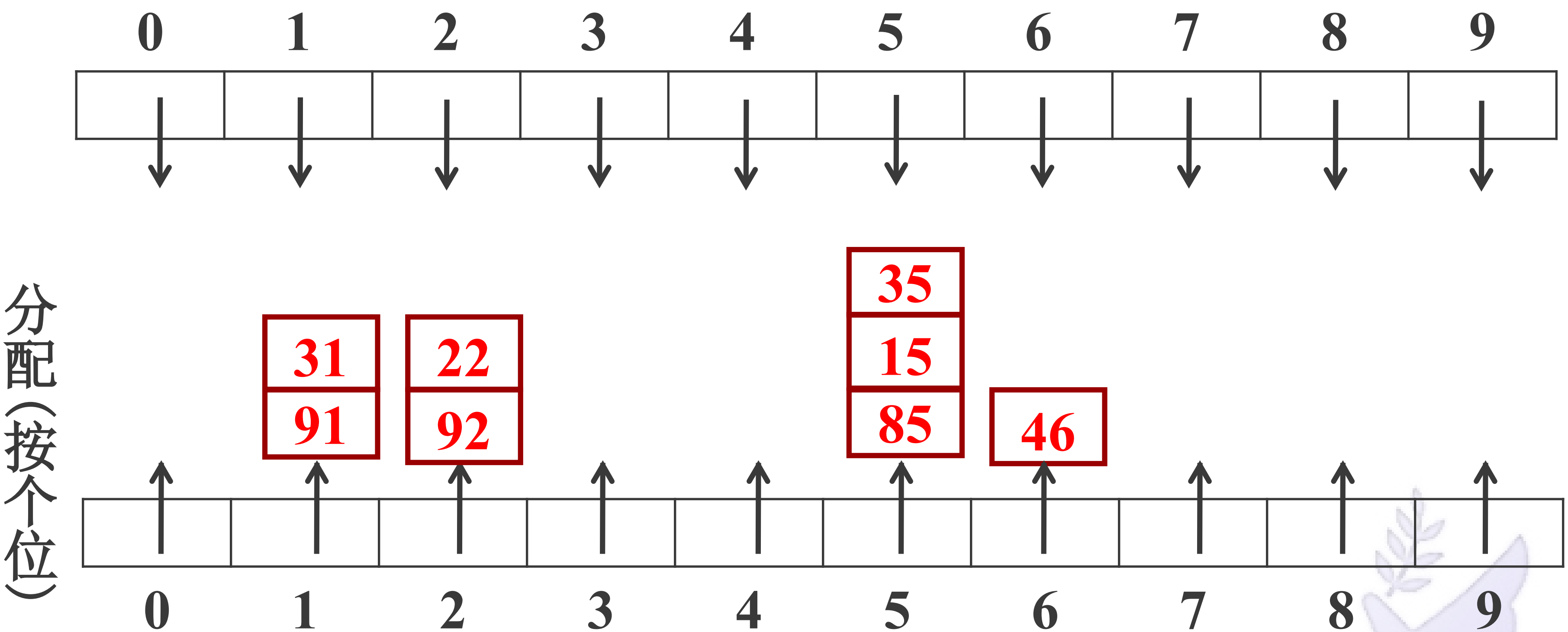
♣	♦	♥	♠
< ♣,2>	<♦,2>	<♥,2>	<♠,2>
< ♣,3>	<♦,3>	<♥,3>	<♠,3>
……	……	……	……
< ♣, A>	<♦,A>	<♥,A>	<♠,A>

收集（按花色有序）：得到正确序列

最高位优先和最低位优先的区别

- 高位优先
  - 先通过一次分配将数据分成多个组，然后对各组数据分别进行排序
- 低位优先
  - 通过多次对全体数据集的分配和收集即可实现排序

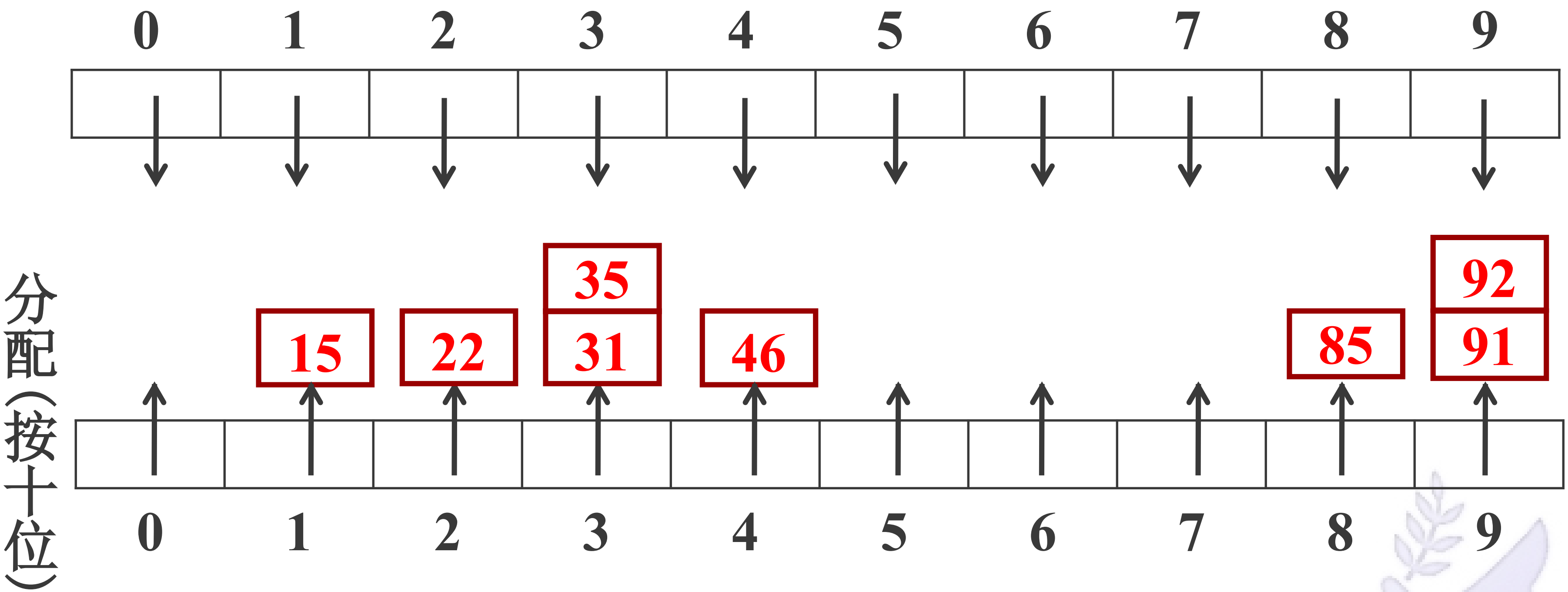
Initial list：46 91 85 15 92 35 31 22，<u>最低位优先</u>



第一次收集：91 31 92 22 85 15 35 46

8.6 基数排序

第一次收集：91 31 92 22 85 15 35 46



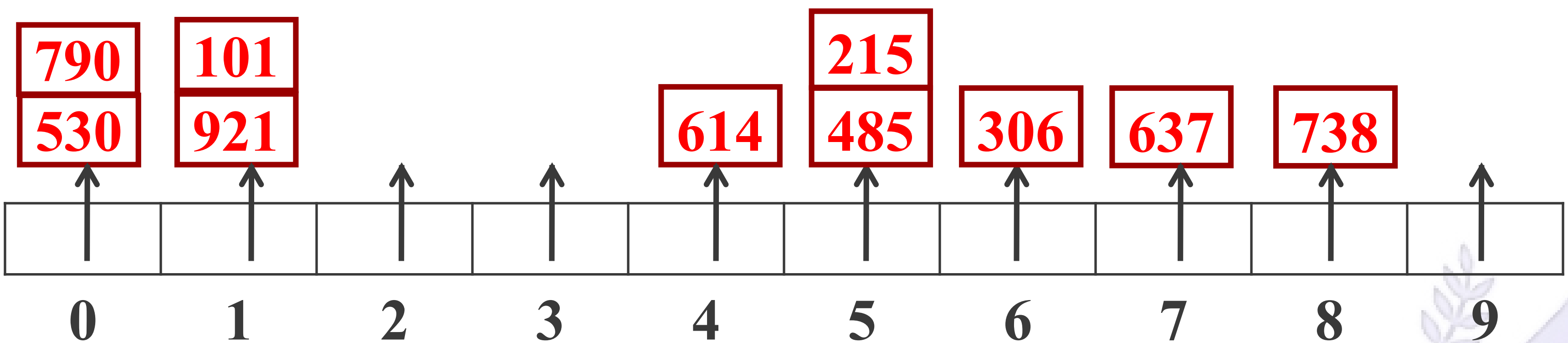
第二次收集：15 22 31 35 46 85 91 92

8.6 基数排序

初始序列：614 738 921 485 637 101 215 530 790 306

第一次分配和收集：个位

序列：614 738 921 485 637 101 215 530 790 306



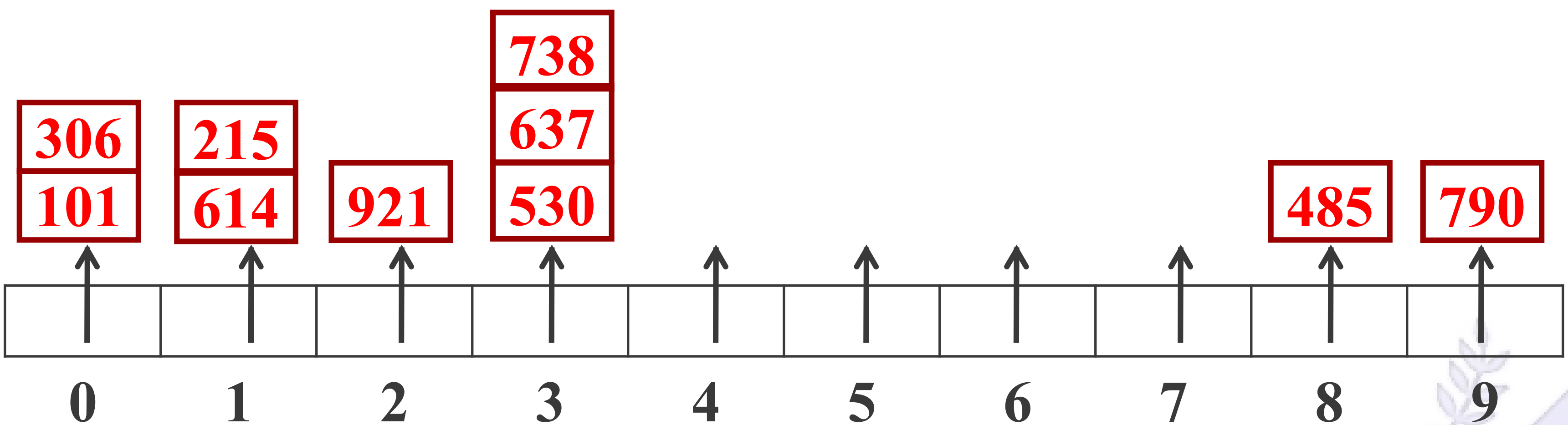
收集：530 790 921 101 614 485 215 306 637 738

8.6 基数排序

初始序列：614 738 921 485 637 101 215 530 790 306

序列：530 790 921 101 614 485 215 306 637 738

第二次分配和收集：十位



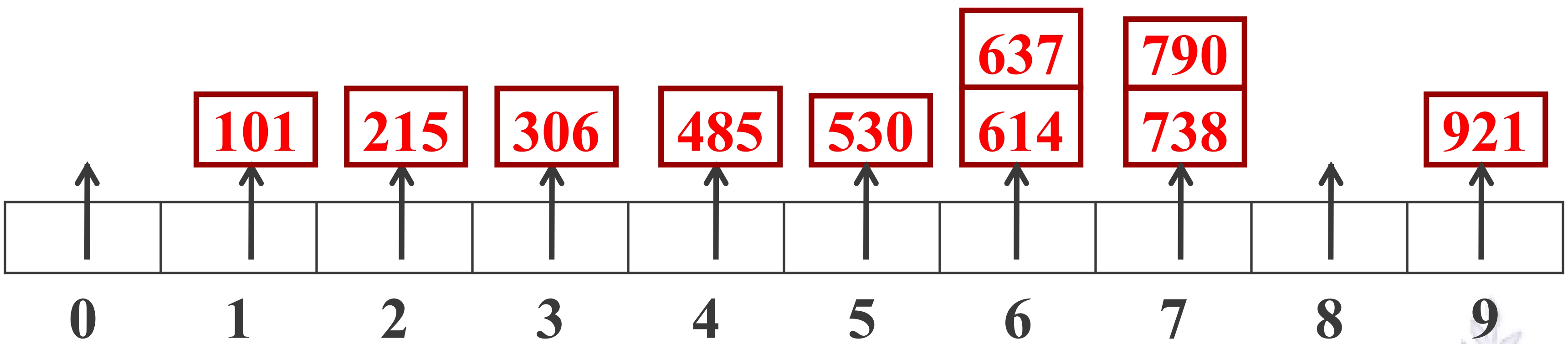
收集：101 306 614 215 921 530 637 738 485 790

8.6 基数排序

初始序列：614 738 921 485 637 101 215 530 790 306

序列： 101 306 614 215 921 530 637 738 485 790

第三次分配和收集：百位



收集：101 215 306 485 530 614 637 738 790 921

8.6 基数排序

一种实现方法：静态链表

```
# define MAX_NUM_KEY 3  //关键字个数
# define RADIX 10 //关键字的基数
# define MAXSIZE 10000
```

静态链表的节点类型

```
typedef sturct {
    KeyType   key[ MAX_NUM_KEY ];//关键字
    int next;
} SLCell; //静态链表的节点类型
```

8.6 基数排序

一种实现方法：静态链表

静态链表类型

```
typedef sturct {
    SLCell r[ MAXSIZE ] // 静态链表空间
    int bitnum;    //关键字位数
    int rednum;  //记录个数
} SLList;
typedef int ArrType[ RADIX ]; // 指针数组类型
```

8.6 基数排序

- 基数排序的特点
  - 关键字包含d位 $k = k_1, k_2, \ldots, k_d$.
  - 每个关键字最多包含r个不同关键字
  - 基本步骤：分配、收集
  - 时间复杂度 $O(d(n+r))$
  - 空间复杂度 $O(n+r)$
  - 稳定

其它排序方法1

- 已知一个含有n个记录的序列，其关键字为整数，其取值范围是[0, n)。若不存在关键字相同的记录，怎样排序最快？

哈希排序

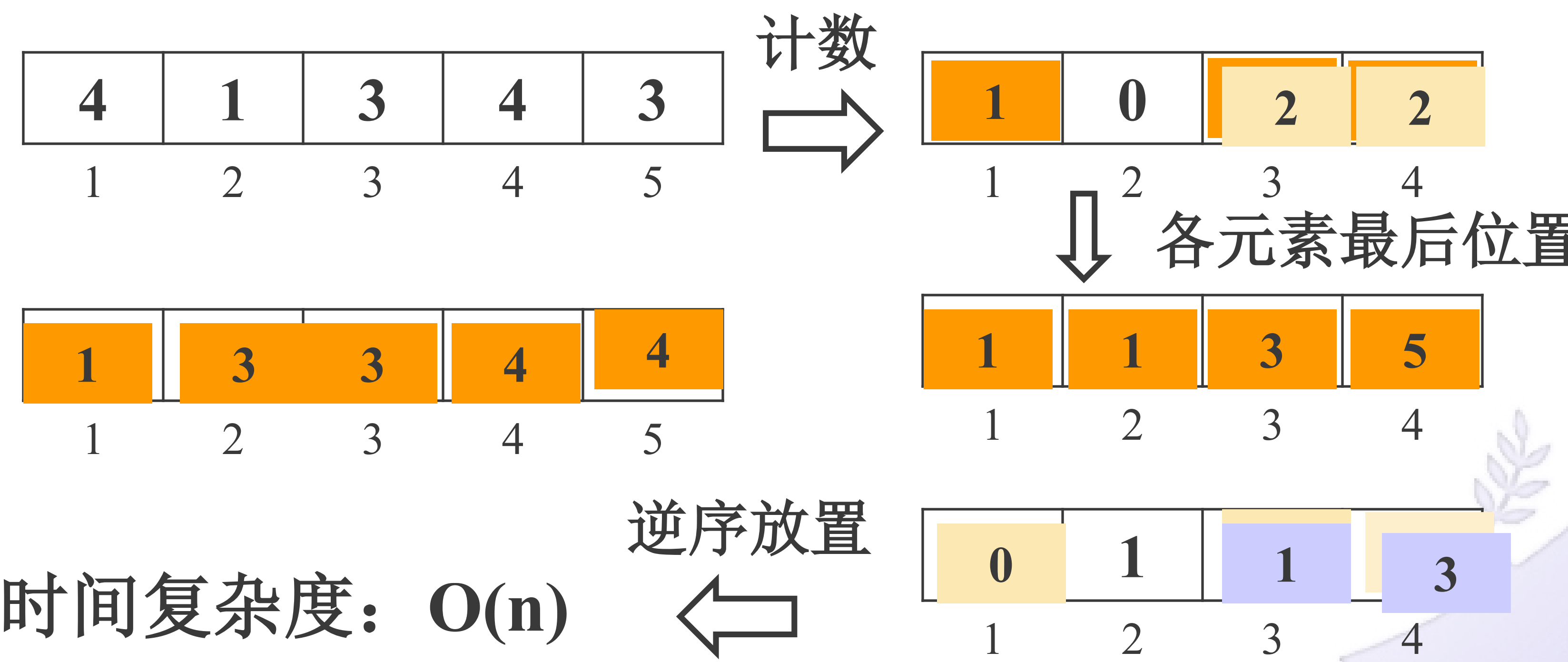
6	4	3	1	7	2	0	5

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7

其它排序方法2

计数排序

- 2、已知一个含有n个记录的序列，其关键字为整数，其取值范围是[0, n)。若<u>存在</u>关键字相同的记录，怎样排序最快？



时间复杂度：O(n)

稳定性：稳定

其它排序方法2

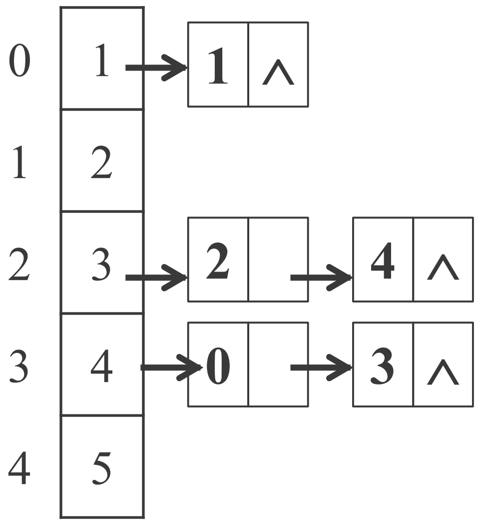
计数排序

- 实例：一年的全国高考考生人数为500万，分数最低100，最高900，没有小数。现对这500万元素的数据集合排序。
  - 一共可出现的分数可能有多少种呢？
  - 一共有900-100+1=801。
  - 对801种不同的成绩计数然后移动记录即可。

其它排序方法3

◆ 已知一个含有n个记录的序列，其关键字为整数，其取值范围是[0, n)。若存在关键字相同的记录，怎样排序最快？

4	**1**	**3**	**4**	**3**
0	1	2	3	4

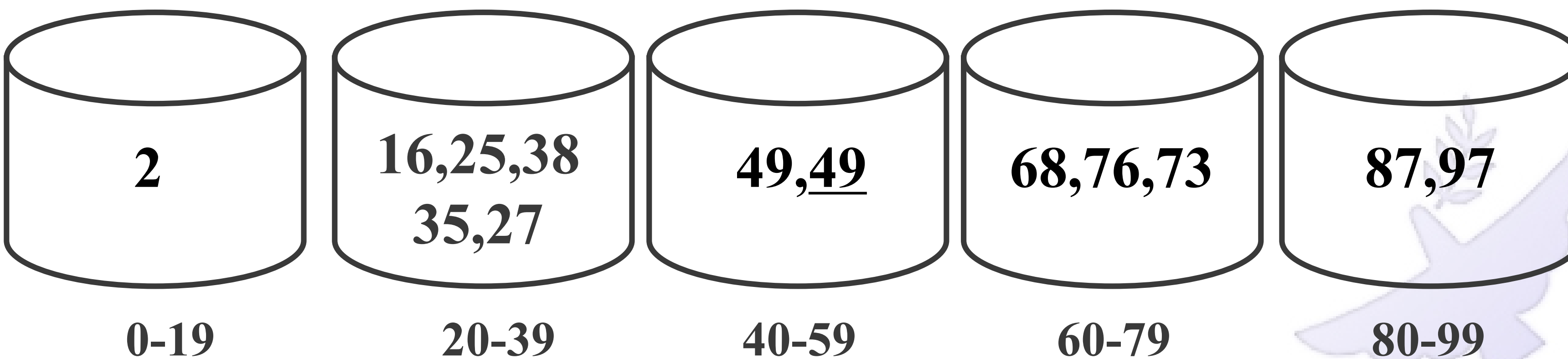


1	**3**	**3**	**4**	**4**
0	1	2	3	4

其它排序方法4

桶排序

- 已知一个含有m个记录的序列，其关键字为整数，其取值范围是[0, n)。
- 假如 {49、2、16、87、25、68、 38 、 35、 97 、 76、73 、 27、 49 }。
- 数据在[1, 100)内 。
- 若分成5组，每组数据放入一个桶：



- 再分别对桶内数据进行排序

其它排序方法5

基数排序

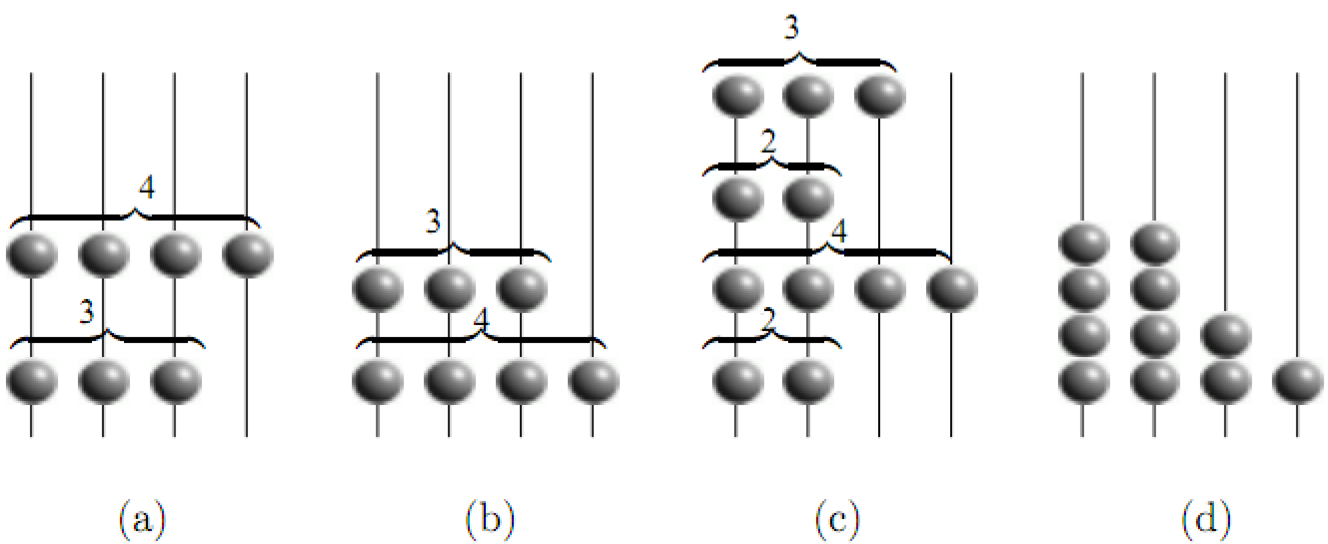
- 已知一个含有n个记录的序列，其关键字为整数，其取值范围是[0, n^2)。怎样排序最快？
- 将每个关键字K认为
  - $K=K_1*n + K_2$,
  - 其中K_1和K_2都是在[0, n)范围内的整数。
- 利用基数排序，则排序的复杂度为O(n)

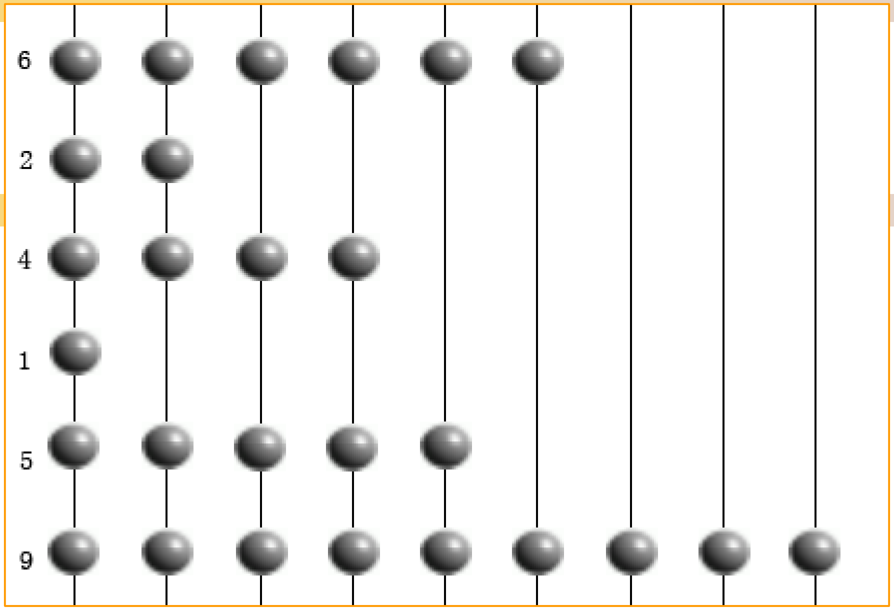
其它排序方法6

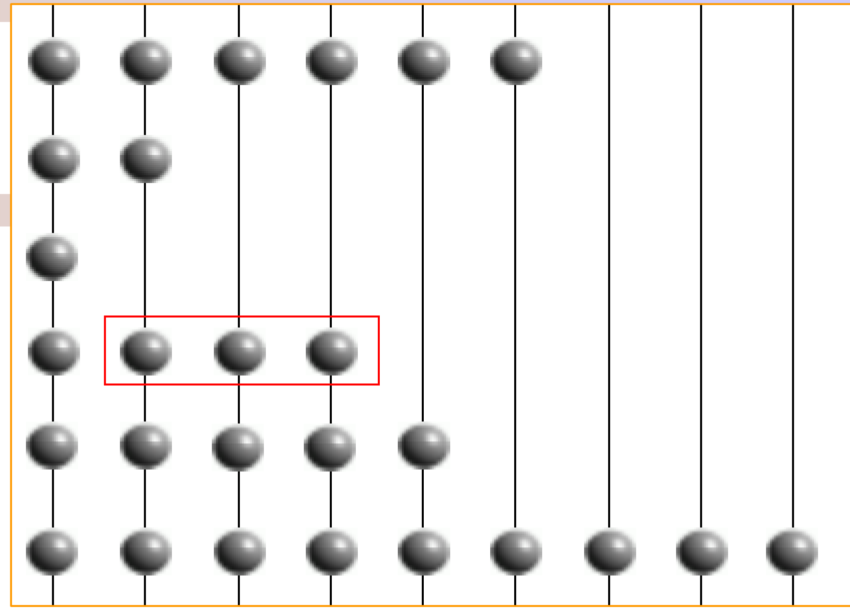
珠排序

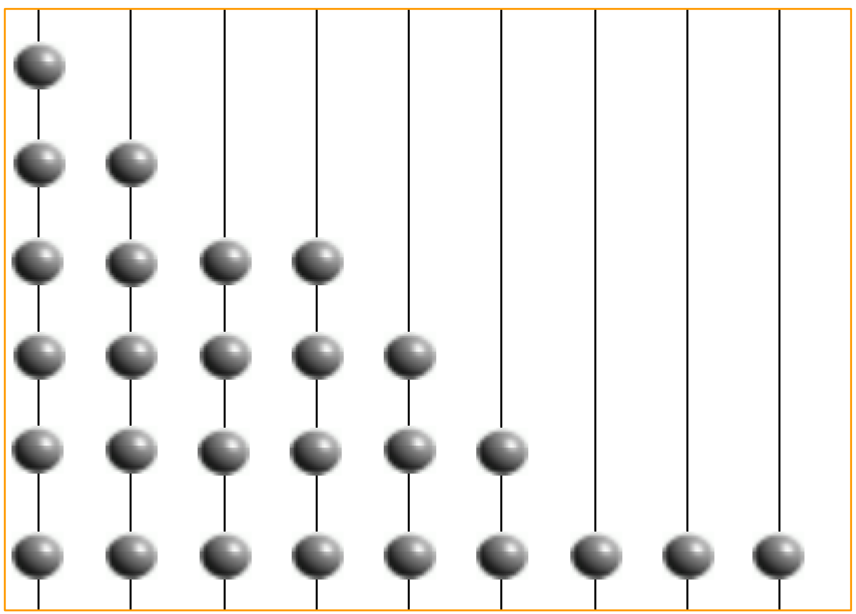
◆ 一个数字用对应的个数珠子表示

❖ 把珠子向算盘珠一样串在一起，然后让珠子自由下落









珠排序可以是以下复杂度级别：
O(1)：即所有珠子都同时移动，无法在计算机中实现。
O(√n)：在真实的物理世界中用引力实现，时间正比于珠子最大高度的平方根，而最大高度正比于**n**。
O(n)：一次移动一列珠子，可以用模拟和数字的硬件实现。
O(S)，**S**是所有输入数据的和：一次移动一个珠子，能在软件中实现。

睡眠排序

- 构造n个线程，它们和这n个数一一对应。
- 设置线程Th_i在n_i时刻醒来。
- 初始化后，线程们开始睡眠，等到对应的数那么多个时间单位后各自醒来，然后输出它对应的数。
- 这样最小的数对应的线程最早醒来，这个数最早被输出。等所有线程都醒来，排序就结束了。

Bogo 排序/猴子排序/随机排序

- 算法思想：
  - 把元素随机排列。
  - 如果没有排好序，再次把元素随机排列。
  - 如果还没有排好序，继续随机排列
  - 直到得到一个有序数组
- 示例：
  - 4, 7, 9, 6, 5, 5, 2, 1 (未排序)
  - 2, 5, 4, 7, 5, 9, 6, 1 (随机排列)
  - 1, 4, 5, 6, 9, 7, 5, 2 (再次随机排列)
  - 1, 2, 4, 5, 5, 6, 7, 9 (天呐，真幸运)

排序算法小结

	排序方法	最好时间	最坏时间	平均时间	辅助空间	稳定性
1	直接插入	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
2	希尔排序	——	——	——	$O(1)$	不稳定
3	冒泡排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
4	快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	$O(\log_2 n)$	不稳定
5	简单选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
6	堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定
7	归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定
8	基数排序	$O(d(n+r))$	$O(d(n+r))$	$O(d(n+r))$	$O(n+r)$	稳定

排序算法小结

- 几点说明

1. 几种简单的排序算法（1、3）的最好时间复杂度都为O(n). 说明算法的输入在接近有序时的效率比较高。
2. 三种平均时间复杂度为$O(n\log_2 n)$的算法中
   - 快速排序的平均效率高，但是最坏时间复杂度为$O(n^2)$，且空间复杂度为$O(\log_2 n)$
   - 堆排序的最坏时间复杂度为$O(n\log_2 n)$，且空间复杂度仅为O(1)，但是不稳定
   - 归并排序的最坏时间复杂度也为$O(n\log_2 n)$，而且是稳定算法，但是空间复杂度为O(n)

排序算法小结

- 不同的排序方法适应不同的应用环境和要求

1. 若n较小，可采用直接插入或简单选择排序
   - 当记录规模较小时，直接插入排序较好，它会比选择更少的比较次数，且是稳定的；
   - 当记录规模稍大时，因为简单选择移动的记录数少于直接插入，所以宜用选简单选择排序。
2. 若初始状态基本有序，则应选用直接插入、冒泡或随机的快速排序为宜；
3. 若n较大，则应采用时间复杂度为$O(n\log_2 n)$的排序方法：快速排序、堆排序或归并排序。
4. 特殊的基数排序

- As of Perl 5.8, merge sort is its default sorting algorithm (it was quicksort in previous versions of Perl).
- In Java, the Arrays.sort() methods use merge sort or a tuned quicksort depending on the datatypes and for implementation efficiency switch to insertion sort when fewer than seven array elements are being sorted.[11]
- Python uses timsort, another tuned hybrid of merge sort and insertion sort, that has become the standard sort algorithm in Java SE 7,[12] on the Android platform,[13] and in GNU Octave.[14]

STL提供的Sort算法

函数名	功能描述
sort	对给定区间所有元素进行排序
stable_sort	对给定区间所有元素进行稳定排序
partial_sort	对给定区间所有元素部分排序
partial_sort_copy	对给定区间部分元素复制并排序
nth_element	找出给定区间的某个位置对应的元素
is_sorted	判断一个区间是否已经排好序
partition	使得符合某个条件的元素放在前面
stable_partition	相对稳定的使得符合某个条件的元素放在前面

- 若在10^8个记录中找最小的两个记录，
- 采取哪种排序算法所需用的关键字比较次数最少？
- 用该算法需用比较多少次?

- **荷兰国旗问题：已知一个含有n个记录的序列，其关键字的取值为{red, white, blue}。请给出一个时间复杂度为O(n)的算法，将这个序列按{red- white- blue}的顺序排好。**

END