第十章 内部排序

王 勇
计算机学院&软件学院 大数据分析与信息安全团队
21#518 电话:13604889411
Email： wangyongcs@hrbeu.edu.cn

哈爾濱工程大學
Harbin Engineering University

本章内容

1. 概述
2. 插入排序
3. 快速排序
4. 堆排序
5. 归并排序
6. 基数排序
7. 各种方法的比较

本章说明

- 学习目标
  - 理解排序的定义和**各种排序方法**的特点，并能灵活应用。
  - 排序方法有不同的分类方法，本课程主要采用基于“**关键字之间的比较**”进行排序的方法。
  - 按排序过程所依据的不同原则分为：**插入排序、交换排序、选择排序、归并排序和基数排序**五类。

本章说明

- 学习目标
  - 掌握各种排序方法**时间复杂度**的分析方法。能从“关键字间的比较次数”分析排序算法的**平均情况和最坏情况**时的时间性能。
  - 按平均时间复杂度划分，内部排序可分为三类：$O(n^2)$ 的简单排序方法，$O(n \cdot \log_2 n)$ 的高效排序方法和$O(d \cdot n)$的基数排序方法。
  - 理解排序方法"稳定"或"不稳定"的含义，弄清楚在什么情况下要求应用的排序方法必须是稳定的。

本章说明

- 知识点

  排序、直接插入排序、折半插入排序、希尔排序、起泡排序、快速排序、简单选择排序、堆排序、2-路归并排序、基数排序、排序方法的综合比较

- 重点和难点

  希尔排序、快速排序、堆排序和归并排序等高效方法是本章的学习重点和难点

本章说明

- **学习指导**

  本章学习主要掌握各种排序方法实现时所依据的**原则**以及它们的**主要操作**（“关键字间的**比较**”和“记录的**移动**”）的时间分析。

- 学习中应注意**掌握**各种**排序方法实现的要点**，可通过对相关算法的手工执行和比较分析，切实掌握各种排序过程的排序特点所在，注意同一排序方法在不同的教科书上可能有不同书写形式描述的算法。

本章内容

1. 概述
2. 插入排序
3. 快速排序
4. 堆排序
5. 归并排序
6. 基数排序
7. 各种方法的比较

- 排序的定义
  - 内部排序和外部排序
    - 内部排序方法的分类

一、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：将下列关键字序列

52, 49, 80, 36, 14, 58, 61, 23, 97, 75

调整为

14, 23, 36, 49, 52, 58, 61 ,75, 80, 97

一般情况下，

假设含n个记录的序列为$\{ R_1, R_2, \cdots, R_n \}$，其相应的关键字序列为 $\{ K_1, K_2, \cdots, K_n \}$

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：

$$K_{p1} \le K_{p2} \le \cdots \le K_{pn}$$

按此固有关系将上式记录序列重新排列为：

$$\{ R_{p1}, R_{p2}, \cdots, R_{pn} \}$$

的**操作**称作**排序**。

二、内部排序和外部排序

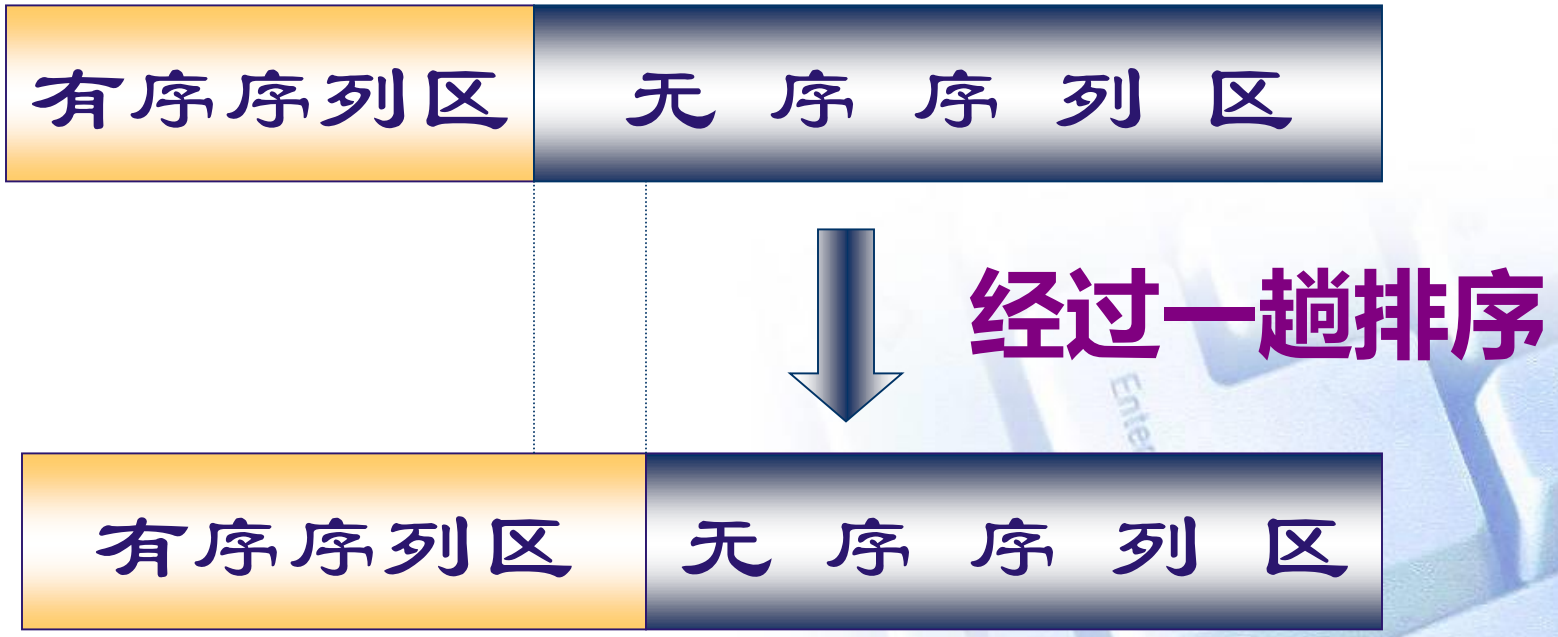
若整个排序过程不需要访问外存便能完成，则称此类排序问题为内部排序。

反之，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，则称此类排序问题为外部排序。

由于外部排序涉及内外存交换，比较复杂，本章主要研究内部排序。

三、内部排序的方法

内部排序的过程实质是一个逐步扩大记录中**有序**序列长度的过程。



基于不同的“扩大” 有序序列长度的方法，内部排序方法大致可分下列几种类型：

插入类　　交换类　　选择类

归并类　　其它方法

1. 插入类

将**无序子序列**中的一个或几个记录“**插入**”到有序序列中，从而增加记录的有序子序列的长度。

2. 交换类

通过“交换”无序序列中的记录从而得到其中关键字最小或最大的记录，并将它加入到有序子序列中，以此方法增加记录的有序子序列的长度。

3. 选择类

从记录的**无序子序列**中“选择”关键字**最小或最大**的记录，并将它加入到有序子序列中，以此方法增加记录的有序子序列的长度。

4. 归并类

通过“归并”两个或两个以上的有序子序列记录，逐步增加记录有序序列的长度。

5. 其它方法

上述分类不是一成不变的，存在多种分类原则和算法思想综合运用的改进算法，以及上述中没有列出的其他算法。

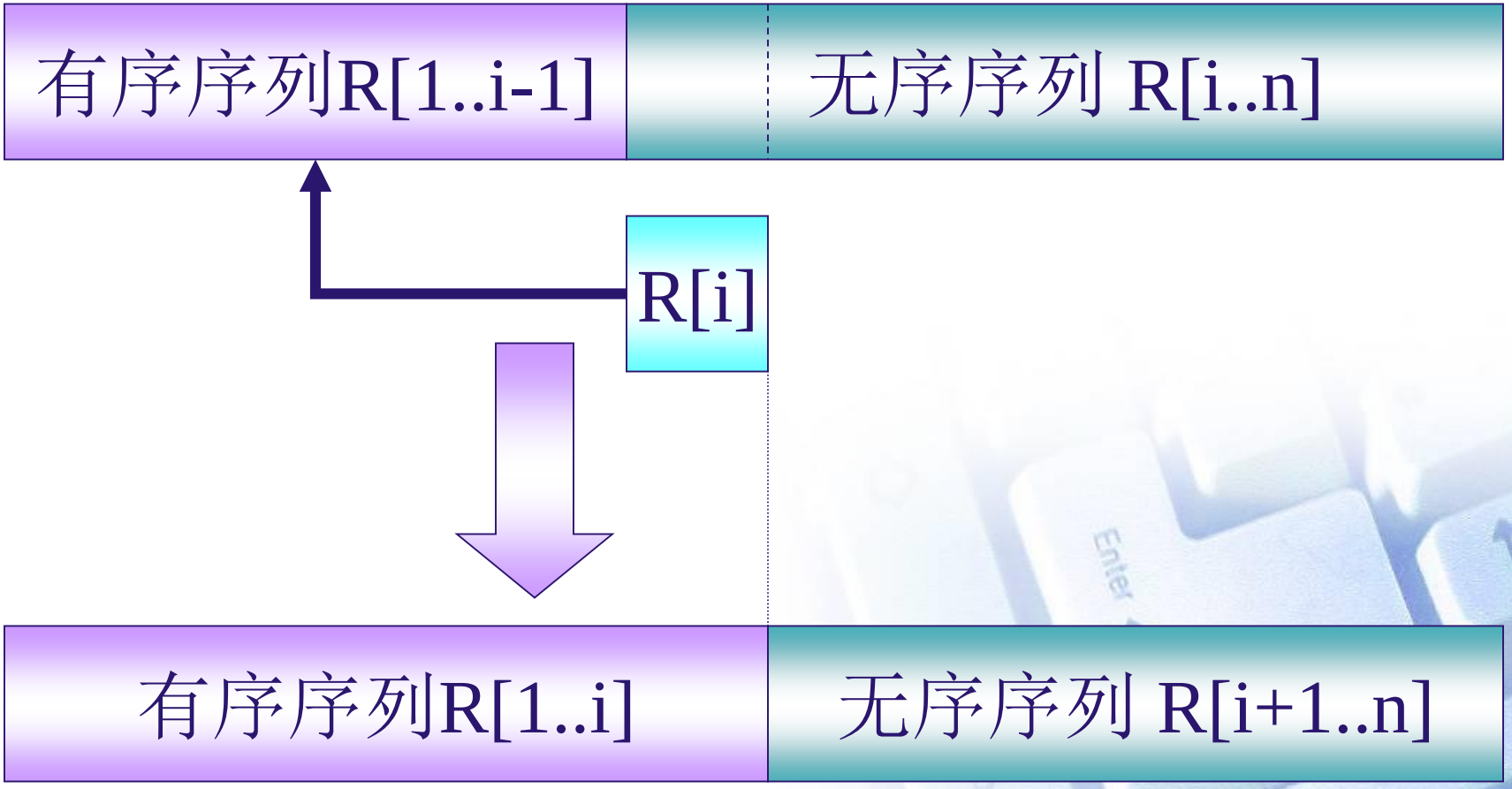
10.7 各种方法比较

排序方法	
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	
基数排序	

本章内容

1. 概述
2. 插入排序
3. 快速排序
4. 堆排序
5. 归并排 序
6. 基数排序
7. 各种方法的比较

一趟直接插入排序的基本思想：



实现“一趟插入排序”可分三步进行：

1．在R[1..i-1]中查找R[i]的插入位置 j ；

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

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）

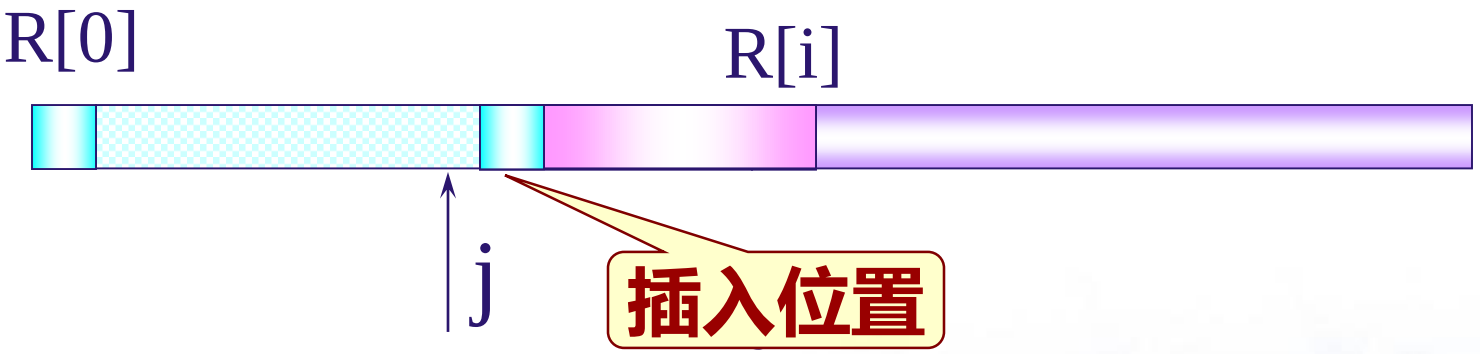
一、直接插入排序

利用“顺序查找”实现

“在R[1..i-1]中查找R[i]的插入位置”

算法的实现要点：

- 从R[i-1]起向前进行顺序查找，找插入位置，监视哨设置在R[0]；



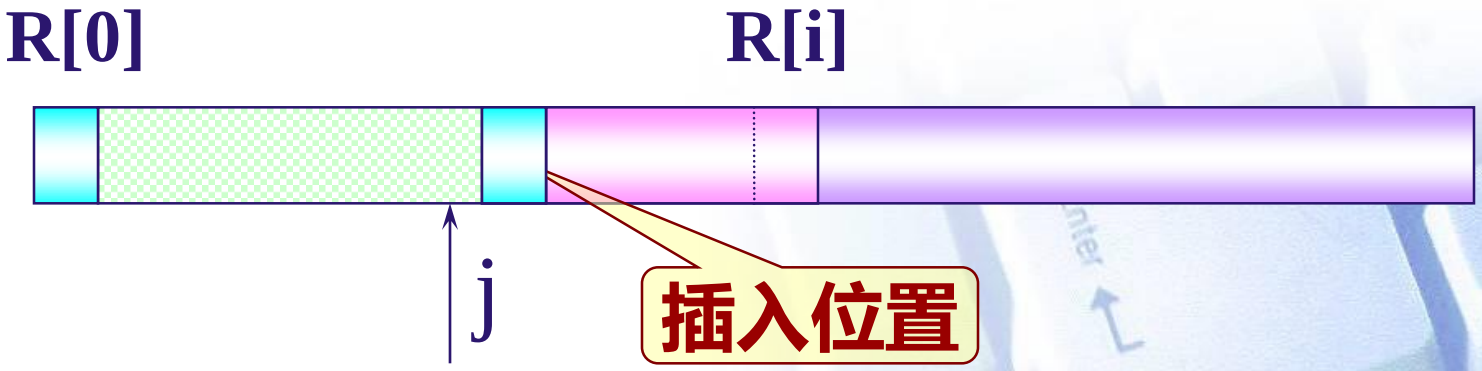

```
R[0] = R[i];              // 设置“哨兵”

for (j=i-1; R[0].key<R[j].key; --j);  // 从后往前找
```

循环结束表明R[i]的插入位置为 *j +1*

- 对于在查找过程中找到的那些关键字不小于R[i].key的记录，并在查找的同时实现记录向后移动；

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



上述循环结束后可以直接进行“插入”

- 令 i = 2，3，…, n,

  实现整个序列的排序

```
for ( i=2;  i<=n;  ++i )
  if (R[i].key<R[i-1].key)
    {
      在 R[1..i-1]中查找R[i]的插入位置;
      插入R[i] ;
    }
```

稳定的排序方法

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

内部排序的时间分析：

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。

- 若待排序文件中有关键字相等的记录，排序后，其前后相对次序与排序前未发生变化，这种排序称为“稳定”排序；否则是“不稳定”排序
- 稳定排序与不稳定排序只是表示所用的排序方法的特性，并不说明哪种方法好与差

对于直接插入排序：

最好的情况（关键字在记录序列中顺序有序）：

“比较”的次数： $\sum_{i=2}^{n} 1 = n-1$

“移动”的次数： **0**

最坏的情况（关键字在记录序列中逆序有序）：

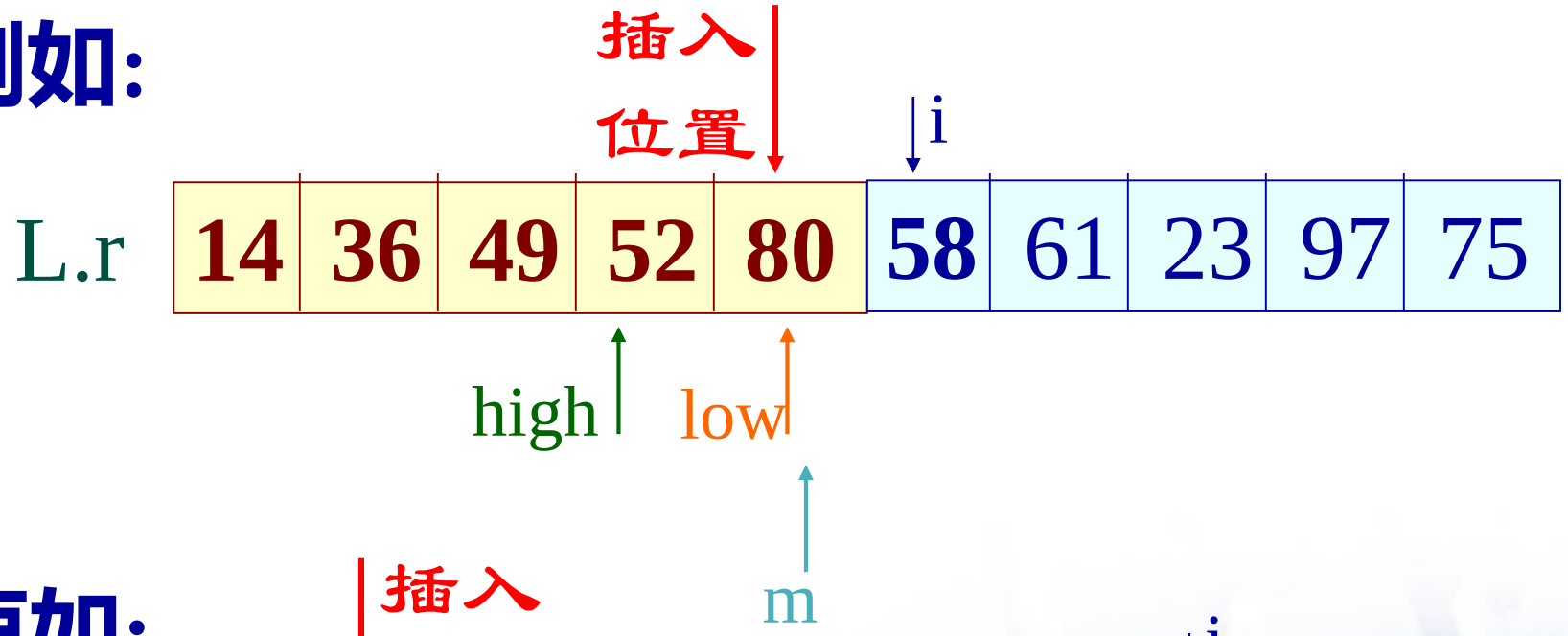
“比较”的次数： $\sum_{i=2}^{n} (i) = \frac{(n+2)(n-1)}{2}$

“移动”的次数： $\sum_{i=2}^{n} (i+1) = \frac{(n+4)(n-1)}{2}$

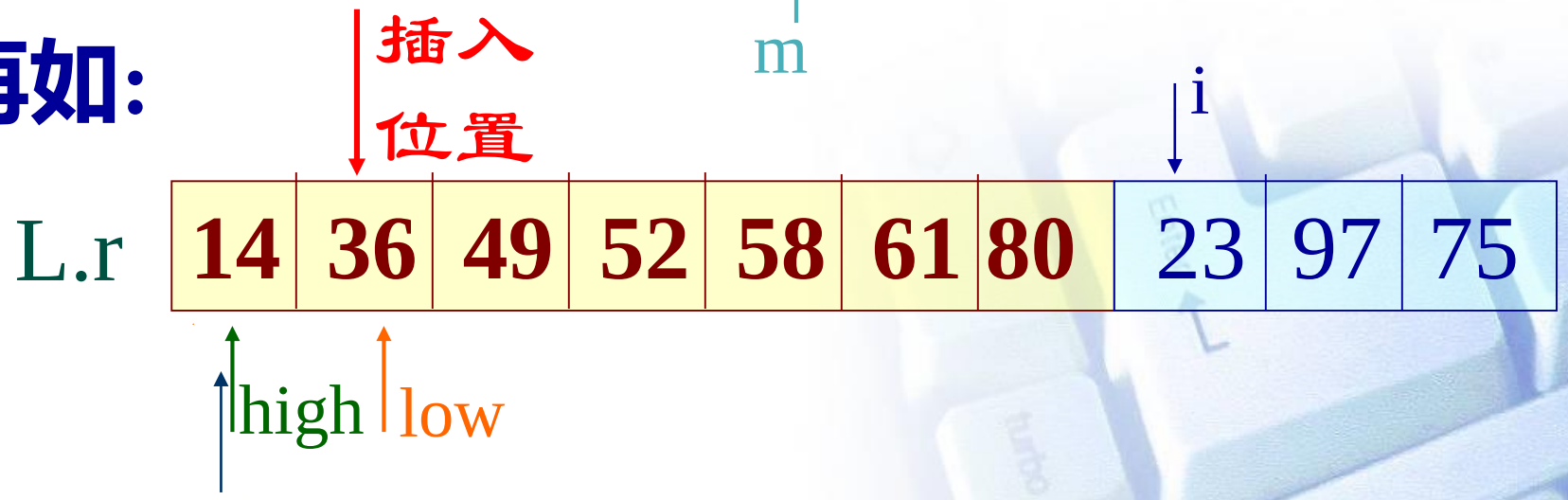
二、折半插入排序

因为 R[1..i-1] 是一个按关键字有序的有序序列，则可以利用折半查找实现“在R[1..i-1]中查找R[i]的插入位置”，如此实现的插入排序为折半插入排序。

例如:


插入
位置
i
L.r
14 36 49 52 80 58 61 23 97 75
high
low
m

再如:


插入
位置
i
L.r
14 36 49 52 58 61 80 23 97 75
high
low
m

```
void BiInsertionSort ( SqList &L )
{
  for ( i=2; i<=L.length; ++i )
  {   L.r[0] = L.r[i];      // 将 L.r[i] 暂存到 L.r[0]
      在 L.r[1..i-1]中折半查找插入位置；
      for ( j=i-1;  j>=low;  --j )
          L.r[j+1] = L.r[j];      // 记录后移
         L.r[low] = L.r[0];  // 插入
  } // for
} // BInsertSort
```

```
low = 1;   high = i-1;
while (low<=high)
{  m = (low+high)/2;          // 折半
   if (L.r[0].key < L.r[m].key)
         high = m-1;   // 插入点在低半区
   else  low = m+1;   // 插入点在高半区
}
```

三、希尔排序（又称缩小增量排序）

❖基本思想：对待排记录序列先作“宏观”调整，再作“微观”调整

- 是对插入排序的一个改进，也称缩小增量排序
- 先将整个待排序记录序列分割成若干个子序列，分别对这些子序列进行直接插入排序；待整个序列中的记录“基本有序”时，再对全体记录进行一次直接插入排序

- 排序过程
  - 取一个正整数$d_1<n$，把所有相隔d_1的记录放一组，组内进行直接插入排序；
  - 然后取$d_2<d_1$，重复上述分组和排序操作；
  - 直至$d_i=1$，即所有记录放进一个组中排序为止

例如：将 n 个记录分成 d 个子序列：

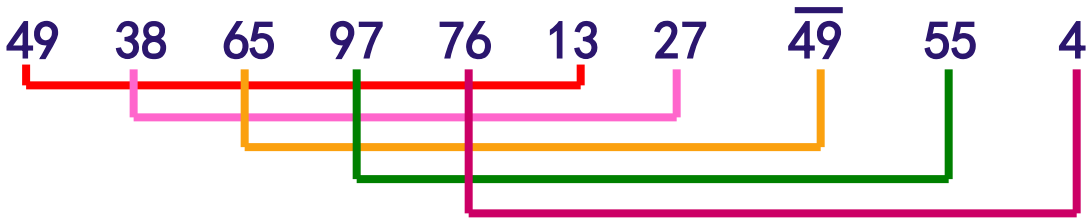
{ R[1]，R[1+d]，R[1+2d]，…，R[1+kd] }
{ R[2]，R[2+d]，R[2+2d]，…，R[2+kd] }
…
{ R[d]，R[2d]，R[3d]，…，R[kd]，R[(k+1)d] }

其中，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。

不稳定的排序方法

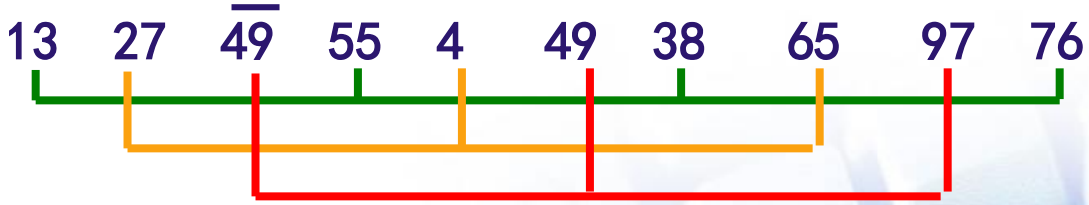
初始关键字：49, 38, 65, 97, 76, 13, 27, 49, 55, 4

取d1=5
一趟分组：



一趟排序结果：13 27 49 55 4 49 38 65 97 76

取d2=3
二趟分组：



二趟排序结果：13 4 49 38 27 49 55 65 97 76

取d3=1
三趟分组：

三趟排序结果：4 13 27 38 49 49 55 65 76 97

```
void ShellInsert ( SqList &L, int dk )
 {
   for ( i=dk+1; i<=n; ++i )
     if ( L.r[i].key< L.r[i-dk].key)
    {//将L.r[i]插入有序子序列
       L.r[0] = L.r[i];            // 暂存在L.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]; // 记录后移，查找插入位置
```

本算法是对顺序表L作一趟希尔插入排序,和一趟直接插入排序相比，作了如下修改：

(1)前后记录位置的增量是d_k，不是1；

(2)r[0]只是暂存单元，不是哨兵。当j<=0时，插入位置已找到。

```
void ShellSort (SqList &L, int dlta[], int t)
{   // 增量为dlta[]的希尔排序
    for (k=0; k<t; ++t)
        ShellInsert(L, dlta[k]);
}
```

增量数组

数组元素个数

希尔排序的时间复杂度和所取增量序列相关，已有学者证明，当增量序列为

2^{t-k-1} (k=0,1, …，t-1)

时，希尔排序的时间复杂度为$O(n^{3/2})$。

- 希尔排序特点
  - 子序列的构成不是简单的“逐段分割”，而是将相隔某个增量的记录组成一个子序列。
  - 希尔排序可提高排序速度，因为
    - 分组后n值减小，n^2更小，而$T(n)=O(n^2)$,所以T(n)从总体上看是减小了；
    - 关键字较小的记录跳跃式前移，在进行最后一趟增量为1的插入排序时，序列已基本有序。
  - 增量序列取法
    - 无除1以外的公因子；
    - 最后一个增量值必须为1；
    - 如……,9,5,3,2,1或……,31,15,7,3,1或……,40,13,4,1等。

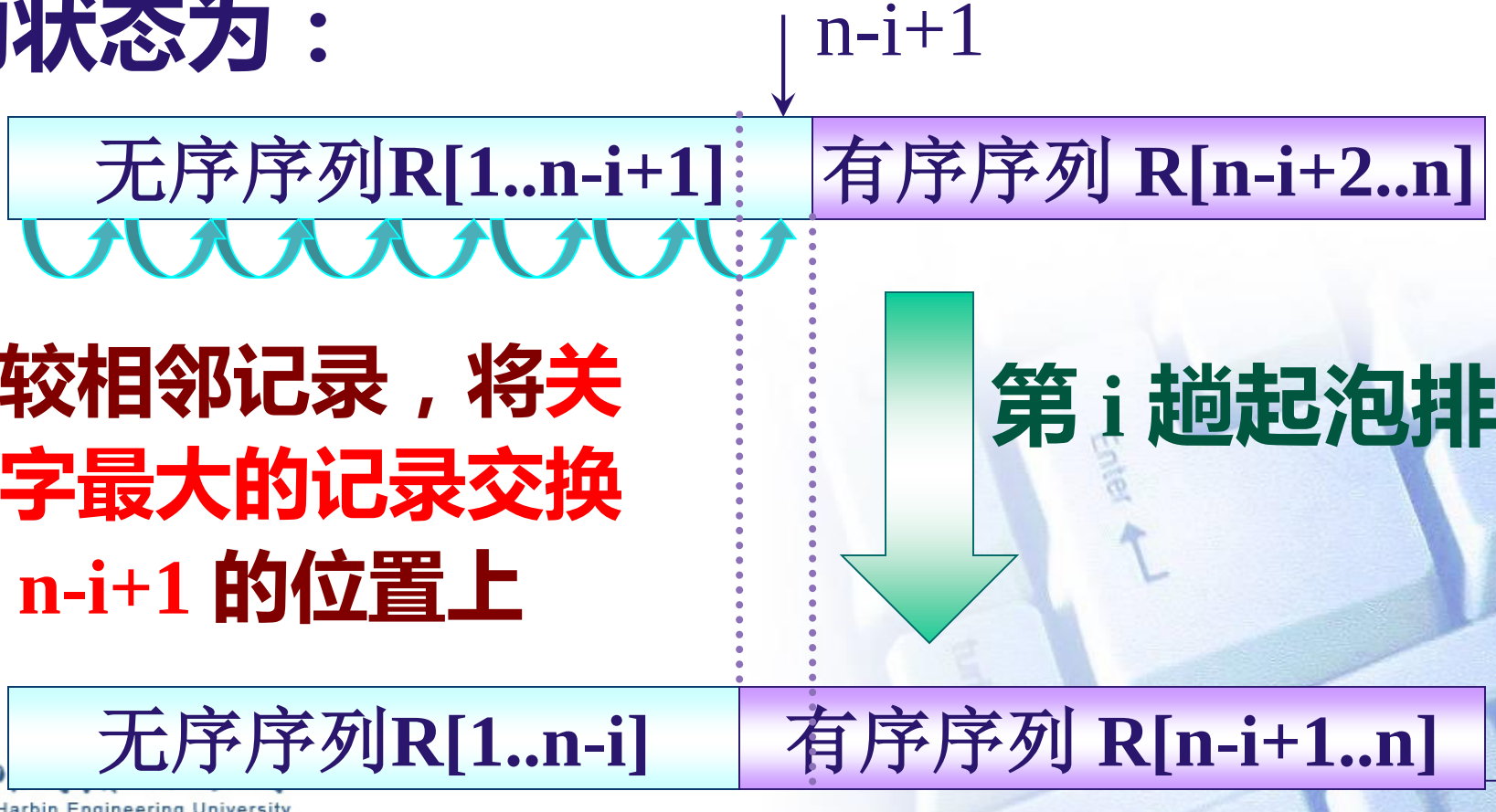
本章内容

1. 概述
2. 插入排序
3. 快速排序
4. 堆排序
5. 归并排序
6. 基数排序
7. 各种方法的比较

- 起/冒泡排序
- 一趟快速排序
- 快速排序
- 时间分析

一、起泡排序

假设在排序过程中，记录序列R[1..n]的状态为：



❖排序过程

- 将第一个记录的关键字与第二个记录的关键字进行比较，若为逆序r[1].key>r[2].key，则交换；然后比较第二个记录与第三个记录；依次类推，直至第n-1个记录和第n个记录比较为止——**第一趟冒泡排序**，结果关键字最大的记录被安置在最后一个记录上。
- 对前n-1个记录进行第二趟冒泡排序，结果使关键字次大的记录被安置在第n-1个记录位置
- 重复上述过程，直到“在一趟排序过程中没有进行过交换记录的操作”为止。

初始关键字	第一趟后	第二趟后	第三趟后	第四趟后	第五趟后	第六趟后
38	38	38	38	13	13	13
49	49	49	13	27	27	27
65	65	13	27	38	38	38
76	13	27	49	49	49	
13	27	49	49	49		
27	49	65	65			
49	76	76				
97	97					

时间分析：

最好的情况（关键字在记录序列中顺序有序）：

只需进行一趟起泡

“比较”的次数： **n-1**

“移动”的次数： **0**

最坏的情况（关键字在记录序列中逆序有序）：

需进行n-1趟起泡

“比较”的次数：

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

“移动”的次数：

$$3\sum_{i=n}^{2}(i-1)=\frac{3n(n-1)}{2}$$

二、一趟快速排序（一次划分）

思想：找一个记录，以它的关键字作为“枢轴（划分元）”，凡其关键字小于枢轴的记录均移动至该记录之前，反之，凡关键字大于枢轴的记录均移动至该记录之后

致使一趟排序之后，记录的无序序列R[s..t]将分割成两部分：R[s..i-1]和R[i+1..t], 且

$R[j].key \le R[i].key \le R[j].key$

$(s \le j \le i-1)$ 枢轴 $(i+1 \le j \le t)$

❑排序过程

对r[s……t]中记录进行一趟快速排序，附设两个指针i（low)和j(high)，设枢轴记录rp=r[s]，x=rp.key，初始时令i=s,j=t

- 首先，从j所指位置向前搜索第一个关键字小于x的记录，并和rp交换；
- 再从i所指位置起向后搜索，找到第一个关键字大于x的记录，和rp交换；
- 重复上述两步，直至i==j为止；

10.3 快速排序

x 49

初始关键字： 27 38 13 49 76 97 65 $\overline{49}$

完成一趟排序： 27 38 13 49 76 97 65 $\overline{49}$

初始关键字： 49 38 65 97 76 13 27 $\overline{49}$

一次划分后：（ 27 38 13 ）49（ 76 97 65 $\overline{49}$ ）

分别进行快速排序：（ 13 ）27（ 38 ）49（ $\overline{49}$ 65 ）76（ 97 ）

（ 13 ）27（ 38 ）49 $\overline{49}$ (65) 76（ 97 ）

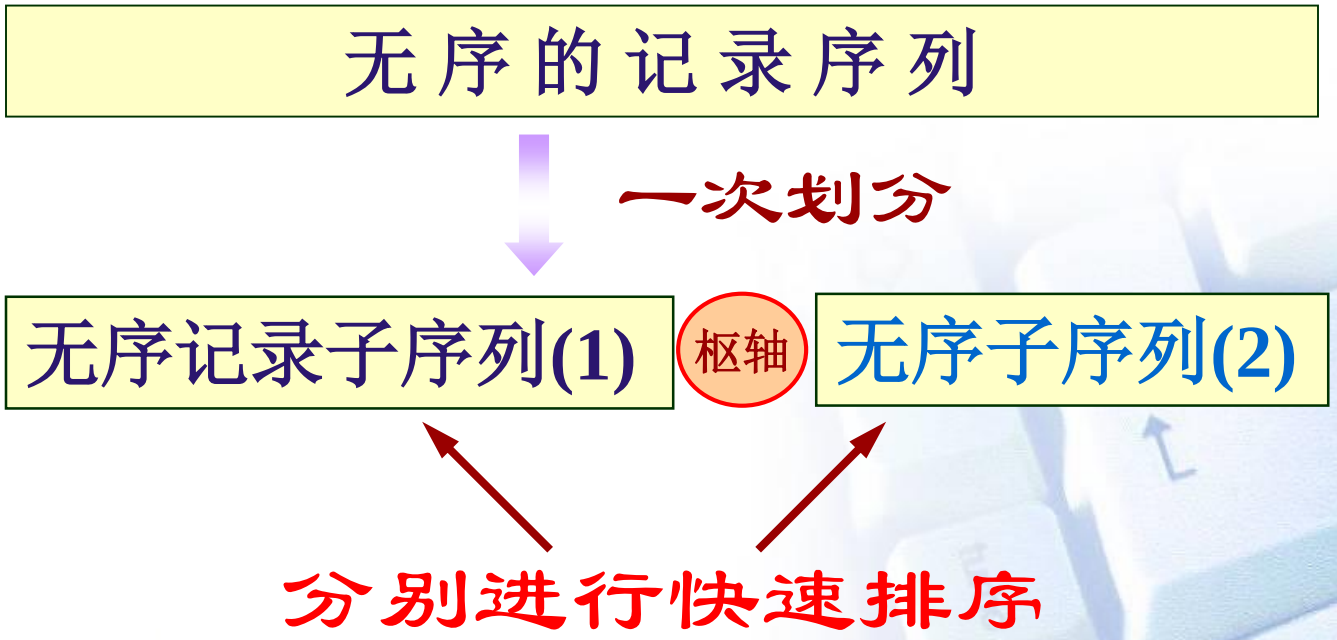
快速排序结束： 13 27 38 49 $\overline{49}$ 65 76 97

```
int Partition (RedType& R[], int low, int high)
{
    pivotkey = R[low].key;
   while (low<high)
  {
     while (low<high && R[high].key>=pivotkey)
          --high;
     R[low]←→R[high];
    while (low<high && R[low].key<=pivotkey)
         ++low;
    R[low]←→R[high];
  }
 return low;          // 返回枢轴所在位置
} // Partition
```

```
int Partition (RedType R[], int low, int high) {
  R[0] = R[low];  pivotkey = R[low].key;  //选第1个为枢轴
   while (low<high)
   {
       while(low<high&& R[high].key>=pivotkey)
             -- high;      // 从右向左搜索
       R[low] = R[high];
       while (low<high && R[low].key<=pivotkey)
             ++ low;      // 从左向右搜索
       R[high] = R[low];
   }
  R[low] = R[0];     return low;
}// Partition
```

三、快速排序

首先对无序的记录序列进行“一次划分”，之后分别对分割所得两个子序列“递归”进行快速排序。



```
void QSort (RedType& R[],  int s,  int  t )
 { // 对记录序列R[s..t]进行快速排序
     if (s < t)
       {            // 长度大于1
          pivotloc = Partition(R, s, t);
                              // 对 R[s..t] 进行一次划分
          QSort(R, s, pivotloc-1);
               // 对低子序列递归排序，pivotloc是枢轴位置
          QSort(R, pivotloc+1, t); // 对高子序列递归排序
       }
} // QSort
```

第一次调用函数 Qsort 时，待排序记录序列的上、下界分别为 1 和 L.length。

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

四、快速排序的时间分析

- 时间复杂度
  - 最好情况（每次总是选到中间值作枢轴）$T(n)=O(n\log_2 n)$
  - 最坏情况（每次总是选到最小或最大元素作枢轴）$T(n)=O(n^2)$
  - 为防止最差情况的出现，一般采取“三者取中”法来确定枢轴。即在第一个记录和最后一个记录，以及中间位置的记录中，选取值为中间的那个来作枢轴，这样就能防止最差情况的出现。
- 空间复杂度：需栈空间以实现递归
  - 最坏情况：$S(n)=O(n)$
  - 一般情况：$S(n)=O(\log_2 n)$
- 对随机的关键字序列，快速排序是目前被认为是最好的排序方法。

本章内容

1. 概述
2. 插入排序
3. 快速排序
4. **堆排序**
5. 归并排序
6. 基数排序
7. 各种方法的比较

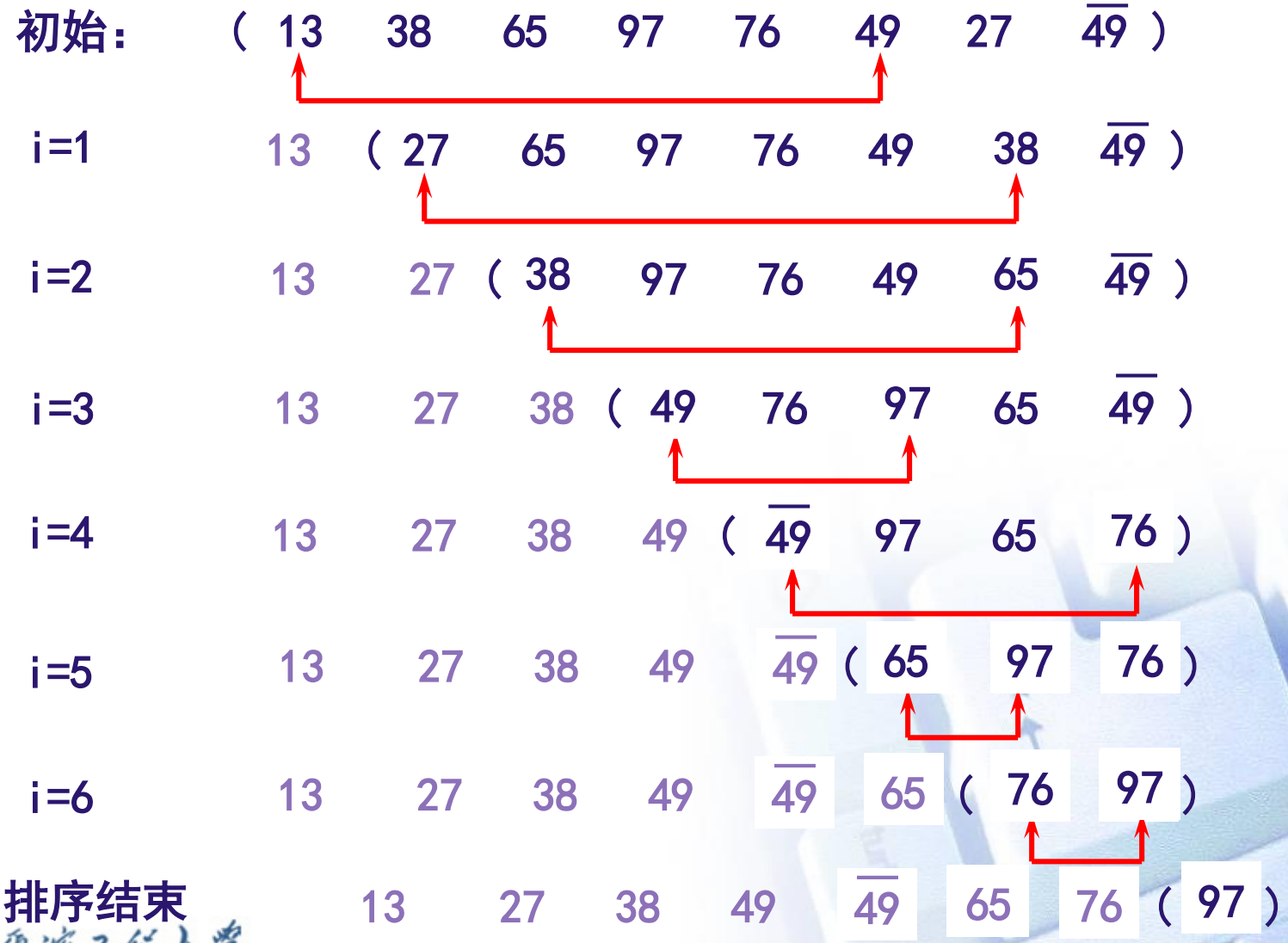
- 简单选择排序
- 堆 排 序

一、简单选择排序

假设排序过程中，待排记录序列的状态为：

- **基本思想**
  - 每一趟在n-i+1（i=1,2,…,n-1）个记录中选取关键字最小的记录，并将它和第i个记录进行互换，从而使其成为有序序列中第i个记录
- **排序过程**
  - 首先，通过n-1次关键字比较，从n个记录中找出关键字最小的记录，将它与第一个记录交换
  - 再通过n-2次比较，从剩余的n-1个记录中找出关键字次小的记录，将它与第二个记录交换
  - 重复上述操作，共进行n-1趟排序后排序结束

10.4 堆排序


初始： (13 38 65 97 76 49 27 49)
i=1 13 (27 65 97 76 49 38 49)
i=2 13 27 (38 97 76 49 65 49)
i=3 13 27 38 (49 76 97 65 49)
i=4 13 27 38 49 (49 97 65 76)
i=5 13 27 38 49 49 (65 97 76)
i=6 13 27 38 49 49 65 (76 97)
排序结束 13 27 38 49 49 65 76 (97)

简单选择排序的算法描述如下：

```
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);
              // 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
```

简单选择排序算法 10.9

```
void SelectSort (SqList &L)
{   // 对顺序表L作简单选择排序
    for (i=1; i<L.length; ++i)
    {  // 选择第 i 小的记录，并交换到位
          j = i;   //j类似于min，初始化为i，与i+1到L.length比较
          for ( k=i+1; k<=L.length; k++ )
// 在L.r[i..L.length]中选择关键字最小的记录
            if ( LT( L.r[k].key , L.r[j].key ))   j =k;
          if ( i!=j ) L.r[j] ↔ L.r[i];
       // 与第 i 个记录互换
    }// for
} // SelectSort
```

时间复杂度T(n)=O(n²)

时间性能分析

对 n 个记录进行简单选择排序，所需进行的关键字间的比较次数，与i+1到n进行比较， 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) //互换

❑堆的定义

堆是满足下列性质的数列$\{r_1, r_2, \cdots, r_n\}$：

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

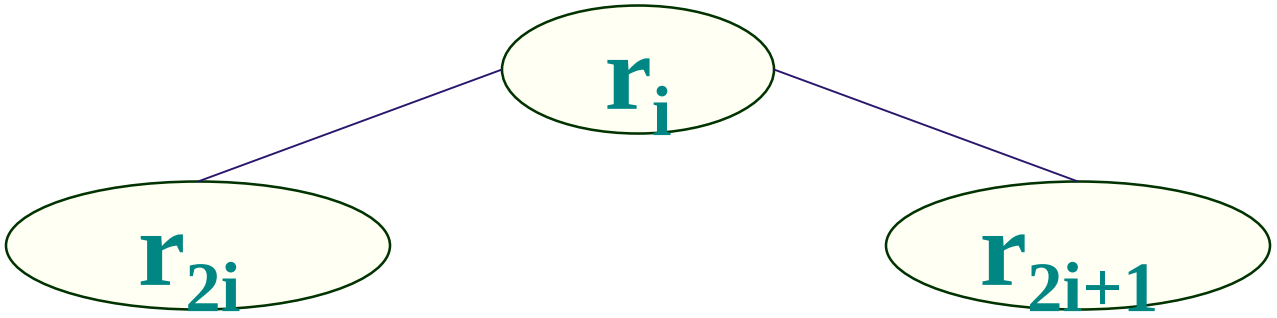
❑若将此序列对应的一维数组看成是一个完全二叉树，则r_i为二叉树的根结点，r_{2i}和r_{2i+1}分别为r_i的“左子树根”和“右子树根”。

例如:

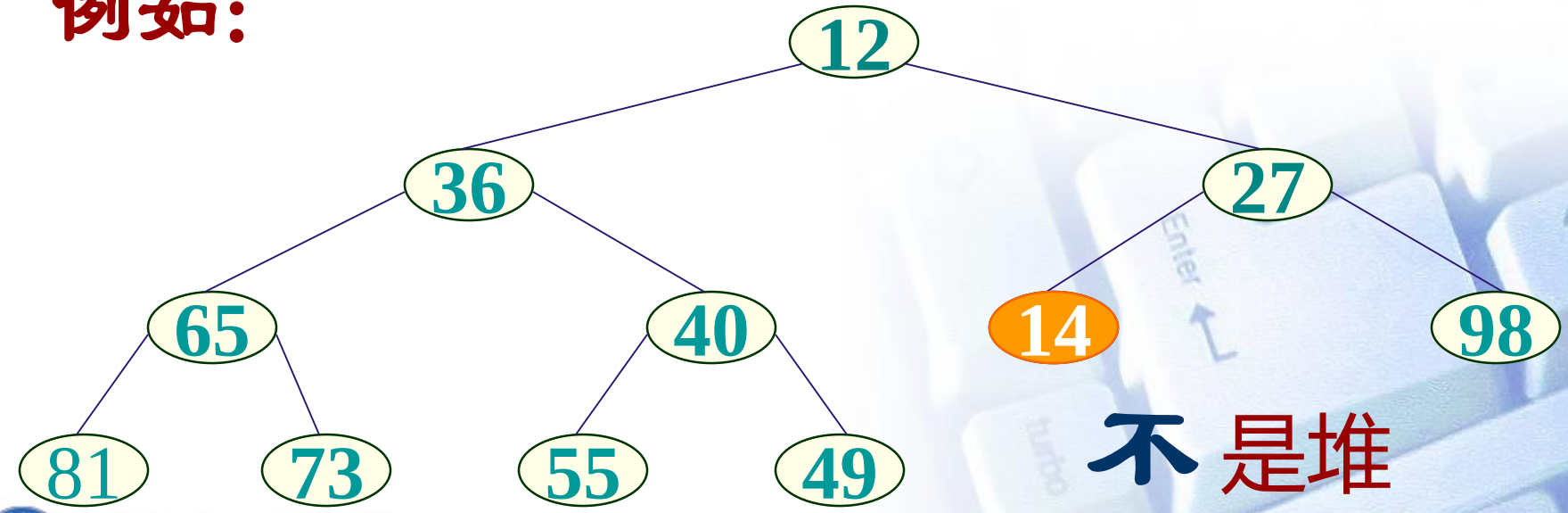
{12, 36, 27, 65, 40, 34, 98, 81, 73, 55, 49} 是小顶堆

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

若将该数列视作完全二叉树，则 r_{2i} 是 r_i 的左孩子; r_{2i+1} 是 r_i 的右孩子



例如：



不是堆

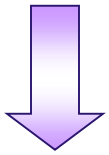
堆排序即是利用堆的特性对记录序列进行排序的一种排序方法。

❑堆排序

- 将无序序列建成一个堆，得到关键字**最小（或最大）**的记录；输出堆顶的最小（大）值后，使剩余的**n-1个元素重又建成一个堆**，则可得到n个元素的次小值；重复执行，得到一个有序序列的过程

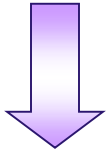
例如：

{ 40, 55, 49, 73, 12, 27, 98, 81, 64, 36 }



建大顶堆

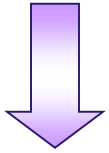
{ 98, 81, 49, 73, 36, 27, 40, 55, 64, 12 }



交换 98 和 12

{ 12, 81, 49, 73, 36, 27, 40, 55, 64, 98 }

经过筛选



重新调整为大顶堆

{ 81, 73, 49, 64, 36, 27, 40, 55, 12, 98 }

定义堆类型为:

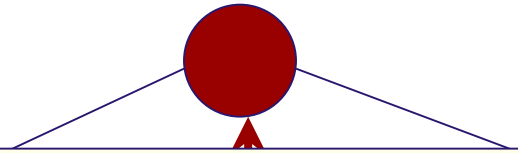
```
typedef  SqList HeapType;
                    // 堆采用顺序表表示
```

两个问题:

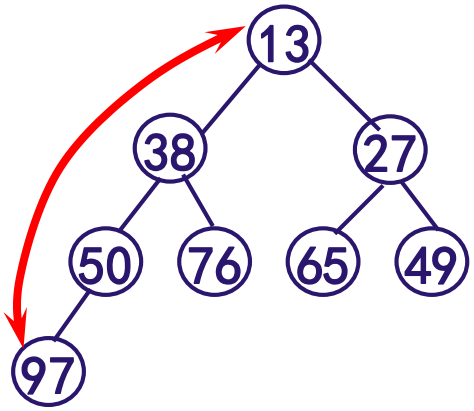
- 如何由一个无序序列建成一个堆？(初始建堆)
- 如何在输出堆顶元素之后，调整剩余元素，使之成为一个新的堆？

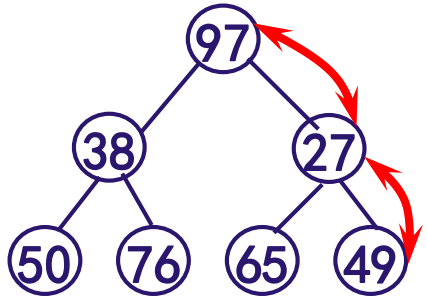
如何“建堆”？　　如何“筛选”？

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。



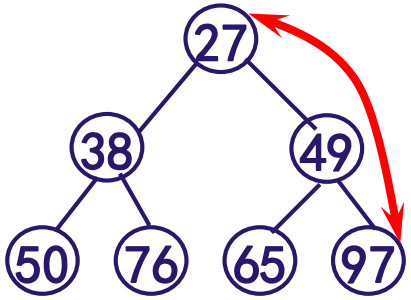
- 第二个问题解决方法——筛选（以小顶堆为例）
  - 输出堆顶元素之后，以堆中最后一个元素替代
  - 然后将根结点值与左、右子树的根结点值进行比较，并与其中小者进行交换；
  - 重复上述操作，直至叶子结点，将得到新的堆





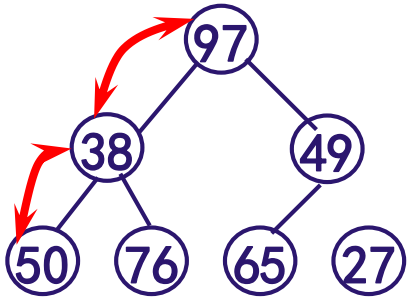
13

输出：13



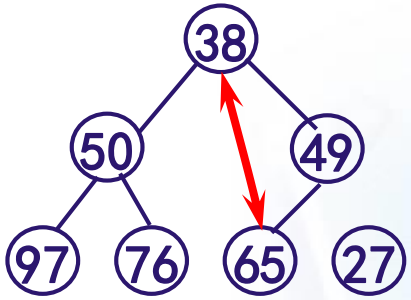
13

输出：13



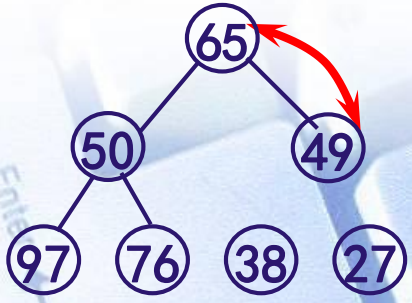
13

输出：13, 27



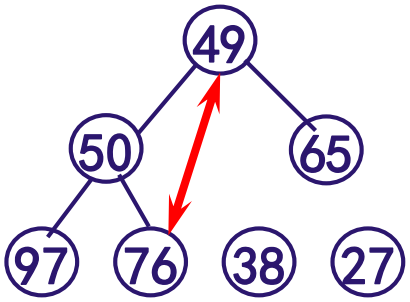
13

输出：13, 27



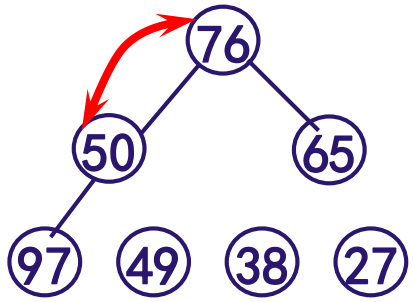
13

输出：13, 27, 38



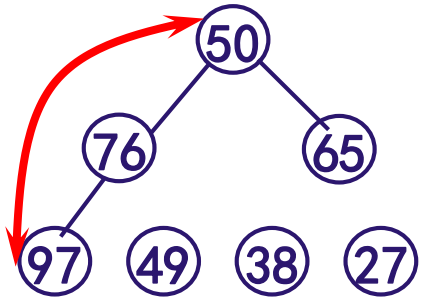
13

输出：13, 27, 38



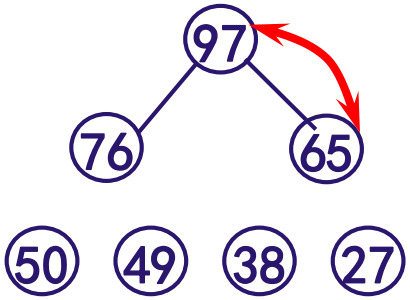
13

输出：13, 27, 38, 49



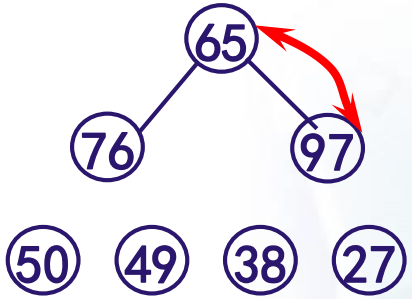
13

输出：13, 27, 38, 49



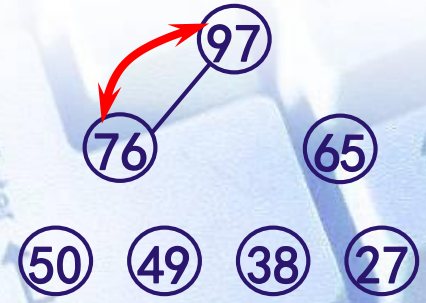
13

输出：13, 27, 38, 49, 50



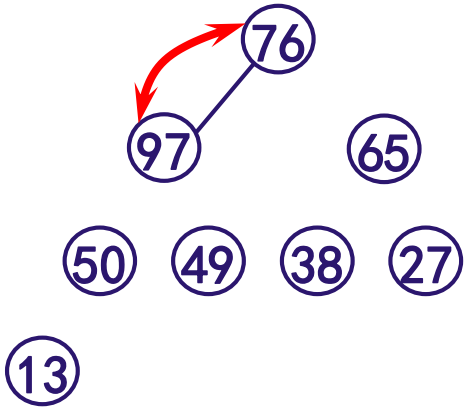
13

输出：13, 27, 38, 49, 50

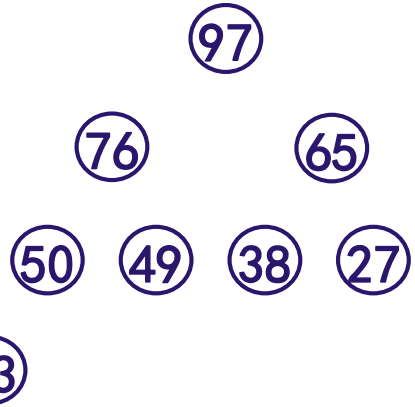


13

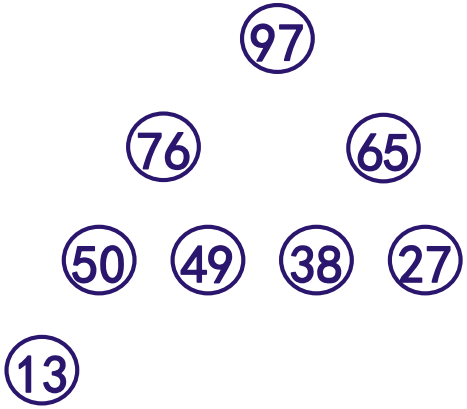
输出：13, 27, 38, 49, 50, 65



输出：13, 27, 38, 49, 50, 65



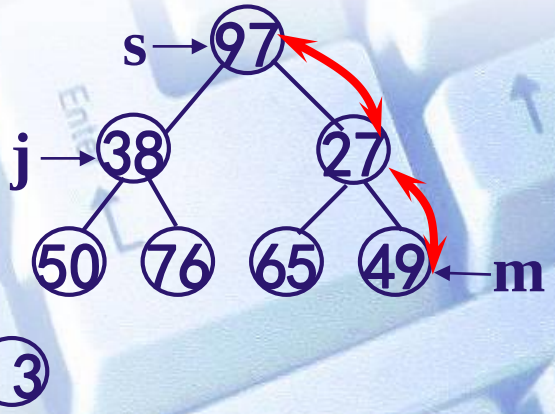
输出：13, 27, 38, 49, 50, 65, 76



输出：13, 27, 38, 49, 50, 65, 76, 97

筛选算法思想

- 设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
- **设指针j为s左孩子，指针 j+1为s右孩子，指针rc=H.r[s]**
- **如果H.r[j+1].key<H.r[j].key, j++，选两者小的与rc.key比较，j始终指向最小的节点下标**
- **如果rc.key<H.r[j].key,则筛选结束，**
- **否则 H.r[s]、H.r[j]互换**
- **s、j下移，即s=j, j=2*j**
- **重复上述过程，直到j>=m为止**



若改为大顶堆，
该如何修改语句？

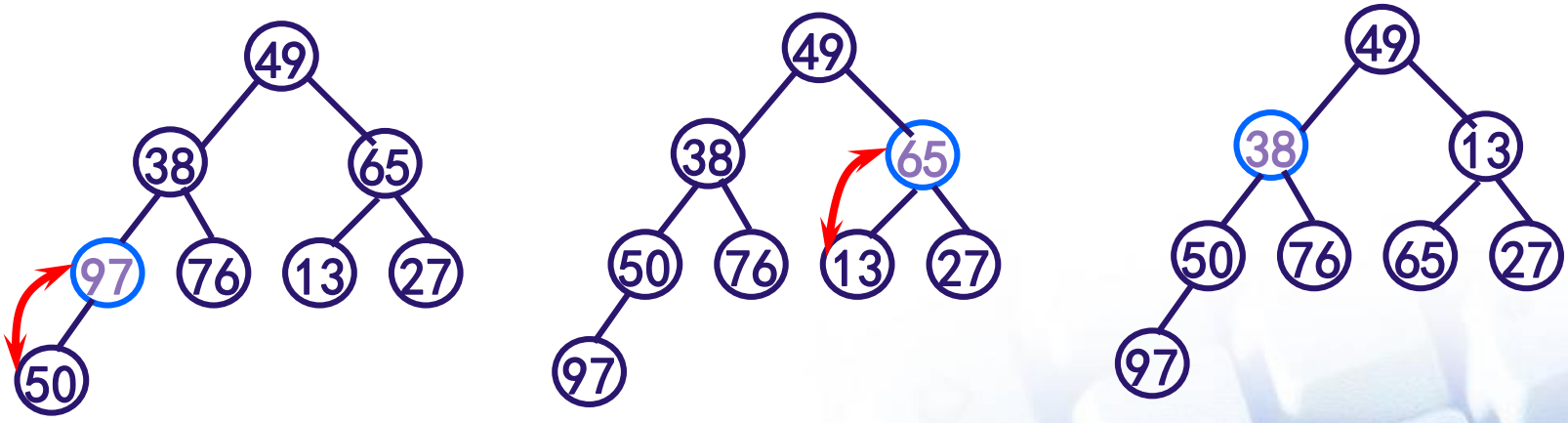
筛选算法：

```
void HeapAdjust (SqList &H, int s, int m){
   // 已知H.r[s..m]中除H.r[s].key之外均满足小顶堆定义，调整H.r[s]
  //使H.r[s..m]成为一个小顶堆（对其中记录的关键字而言）
   rc = H.r[s];                          // 暂存根结点的记录
   for(j=2*s; j<=m; j*=2 ) {             // 沿关键字较小的孩子结点向下筛选
        if ( j  j<m && LT(H.r[j].key,H.r[j+1].key)
   // j 为关键字较小的孩子记录的下标
        if (L  !LT(rc.key,  H.r[j].key )
                                  // 不需要调整，跳出循环
        H.r[s] = H.r[j]; s = j;   // 将小关键字记录往上调
   }// for
   H.r[s] = rc;                          // 回移筛选下来的记录
} // HeapAdjust
```

- 第一个问题（如何建初始堆）解决方法？

例如：含8个元素的无序序列

（49，38，65，97，76，13，27，50）

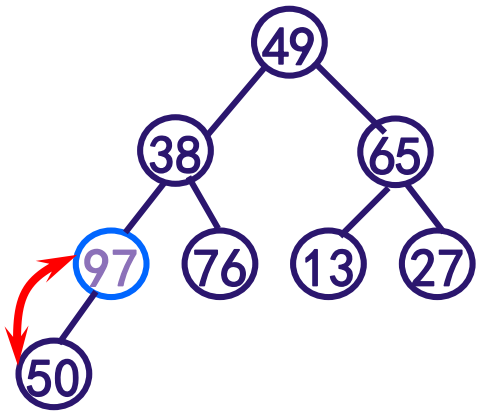


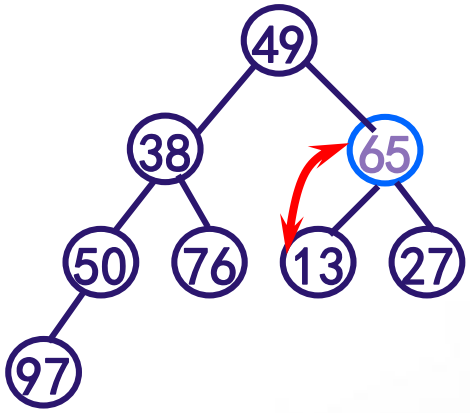
❑解决方法

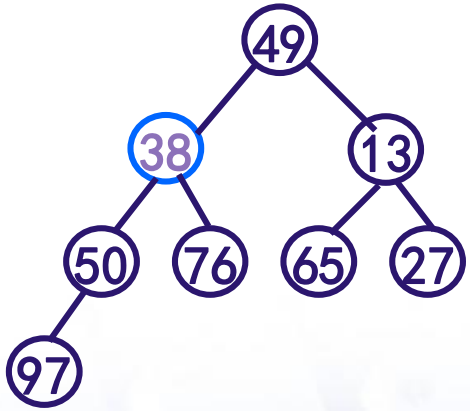
❖从无序序列的第⌊n/2⌋个元素（即此无序序列对应的完全二叉树的最后一个非终端结点）起，至第一个元素止，进行反复筛选。

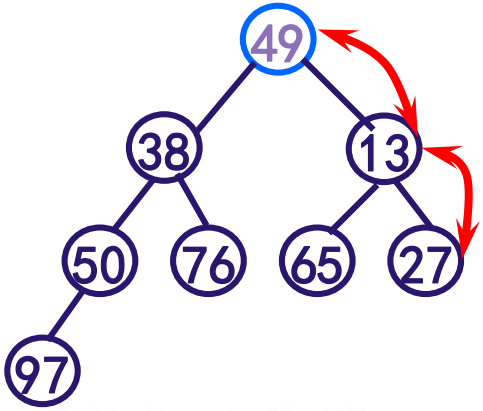
例如：含8个元素的无序序列

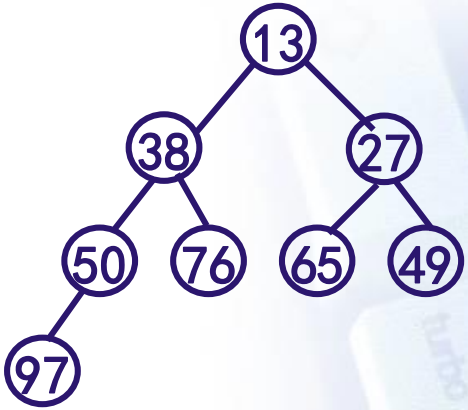
（49，38，65，97，76，13，27，50）











堆排序算法10. 11

```
void HeapSort ( SqList &H )
{   // 对顺序表H进行堆排序
    for ( i=H.length/2; i>0; --i )
                              // 将 H.r[1..H.length] 建成大/小顶堆
        HeapAdjust ( H, i, H.length );
    for ( i=H.length; i>1; --i )
      {
        H.r[1] ↔ H.r[i];  // 将堆顶记录和当前未经排序的子序列
                              H.r[1..i]中最后一个记录互相交换
        HeapAdjust(H, 1, i-1);  // 将H.r[1..i-1]重新调整//为大/小顶堆
      }//for
} // HeapSort
```

- 算法评价
  - **时间复杂度**：最坏情况下T(n)=O(nlogn)
    - 筛选算法：最多从第1层筛到最底层，为完全二叉树的深度
      $\lfloor \log_2 n \rfloor + 1$;
    - 初始建堆：**调用**筛选算法⌊n/2⌋次；
    - 重建堆：调用筛选算法 n-1 次（输出）。
  - **空间复杂度**：S(n)=O(1) //交换
  - **记录较少时，不提倡使用。**
  - **不稳定的排序方法**

本章内容

1. 概述
2. 插入排序
3. 快速排序
4. 堆排序
5. 归并排序
6. 基数排序
7. 各种方法的比较

归并排序的过程基于下列**基本思想**进行：

将两个或两个以上的有序子序列 “归并” 为一个有序序列。

在内部排序中，通常采用的是2-路归并排序。

即：将两个位置相邻的记录有序子序列

有序子序列 R[*l*..*m*]	有序子序列 R[*m*+1..*n*]

归并为一个记录的有序序列。

有 序 序 列 R[*l*..*n*]

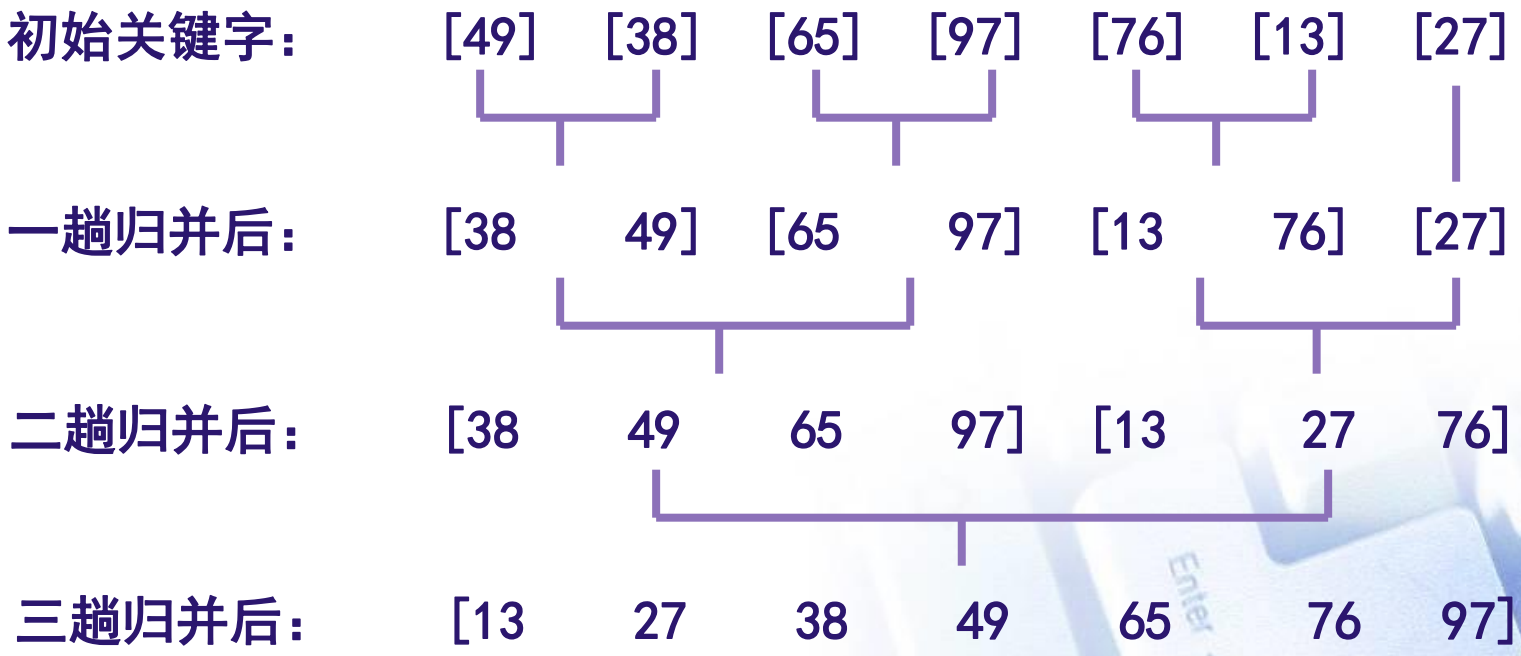
这个操作对顺序表而言，是轻而易举的。

❖排序过程

- 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⌈n/2⌉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。

10.5 归并排序

稳定的排序方法



两个有序序列归并为一个有序序列的算法 10.12

```
void Merge(RcdType SR[],RcdType &TR[], int i, int m, int n)
{
    // 将有序的SR[i..m]和SR[m+1..n]归并为有序的TR[i..n]
    for (k=i, j=m+1; i<=m && j<=n; ++k)
    {
    // 将SR中记录按关键字从小到大地复制到TR中
        if ( LQ(SR[i].key, SR[j].key))     TR[k] = SR[i++];
        else TR[k] = SR[j++];
    }
    while (i<=m) TR[k++] = SR[i++]; // 将剩余的 SR[i..m] 复制到TR
    while (j<=n) TR[k++] = SR[j++]; // 将剩余的 SR[j..n] 复制到TR
} // Merge
```

每趟两两归并需调用merge算法⌈n/2h⌉次（h为有序段长度）

```
void MSort(RcdType SR[], RcdType &TR1[], int s, int t)
    // 将SR[s..t] 归并排序为TR1[s..t]算法 10.13
{
     if (s==t)    TR1[s]=SR[s]
       else
     {    m=(s+t)/2; //将SR[s..t]平分为SR[s..m]和SR[m+1..t]
          MSort(SR,TR2,s,m);   //将SR[s..m]归并TR2[s..m]
          MSort(SR,TR2,m+1,t);//将SR[m+1..t]归并TR2[m+1..t]
          Merge(TR2,TR1,s,m,t);//将TR2[s..m]、TR2[m+1..t]
      }                                  归并为TR1[s..t]
} // MSort
Void MergeSort(Sqlist &L)   //将顺序表L归并排序10.14
{
    MSort(L.r, L.r, 1, L.Length);
}//MergeSort
```

- 算法评价
  - 时间复杂度：$T(n)=O(n\log_2 n)$
    - 每趟两两归并需调用merge算法$\lceil n/2h \rceil$次（h为有序段长度）；
    - 整个归并要进行$\lceil \log_2 n \rceil$趟。
  - 空间复杂度：$S(n)=O(n)$
    - 等量的辅助存储空间

本章内容

基数排序是一种借助“多关键字排序”的思想来实现“单关键字排序”的内部排序算法。

- 多关键字的排序
- 链式基数排序

一、多关键字的排序

例 对52张扑克牌按以下次序排序

两个关键字：花色（♣<♦<♥<♠）

面值（2<3<……<A）

并且“花色”地位高于“面值”

主关键字

次关键字

♣2<♣3<……<♣A<♦2<♦3<……<♦A<

♥2<♥3<……<♥A<♠2<♠3<……<♠A

实现多关键字排序通常有两种作法:

最高位优先MSD法

最低位优先LSD法

先对最高位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，每个子序列有相同的K^0值，然后分别对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。

例如:学生记录含三个关键字:

系别、班号和班内的序列号，其中以系别为“最主” 位关键字。

MSD的排序过程如下:

无序序列	3,2,30	1,2,15	3,1,20	2,3,18	2,1,20
对K^0排序	1,2,15	2,3,18	2,1,20	3,2,30	3,1,20
对K^1排序	1,2,15	2,1,20	2,3,18	3,1,20	3,2,30
对K^2排序	1,2,15	2,1,20	2,3,18	3,1,20	3,2,30

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。

排序过程中不需要根据“前一个”关键字的排序结果，将记录序列分割成若干个(“前一个”关键字不同的)子序列

例如:学生记录含三个关键字:系别、班号和班内的序列号，其中以系别为“最主” 位关键字。

LSD的排序过程如下:

无序序列	3,2,30	1,2,15	3,1,20	2,3,18	2,1,20
对K^2排序	1,2,15	2,3,18	3,1,20	2,1,20	3,2,30
对K^1排序	3,1,20	2,1,20	1,2,15	3,2,30	2,3,18
对K^0排序	1,2,15	2,1,20	2,3,18	3,1,20	3,2,30

基数排序的时间复杂度为O(d(n+rd))

其中，分配为O(n);

收集为O(rd)(rd为“基”),

d为“分配-收集”的趟数。

本章内容

1. 概述
2. 插入排序
3. 快速排序
4. 堆排序
5. 归并排 序
6. 基数排序
7. 各种方法的比较

一、时间性能

1. 平均的时间性能

时间复杂度为 $O(n\log n)$：

快速排序、堆排序和归并排序

时间复杂度为 $O(n^2)$：

直接插入排序、起泡排序和简单选择排序

时间复杂度为 $O(n)$:

基数排序

2. 当待排记录序列按关键字顺序有序时

直接插入排序和**起泡排序**能达到$O(n)$的时间复杂度;

快速排序的时间性能蜕化为$O(n^2)$

3. 简单选择排序、堆排序和归并排序的时间性能**不随**记录序列中关键字的分布而改变。

二、空间性能

指的是排序过程中所需的辅助空间大小

1. 所有的**简单排序方法**(包括：直接插入、起泡和简单选择) 和**堆排序**的空间复杂度为**O(1)**；

2. **快速排序为O(log*n*)，为递归程序执行过程中，栈所需的辅助空间;**

3. **归并排序**所需辅助空间最多，其空间复杂度为 $O(n)$;

三、排序方法的稳定性能

1. 稳定的排序方法指的是，对于两个关键字相等的记录，它们在序列中的相对位置，在排序之前和经过排序之后，没有改变。

排序之前： $\{\cdots\cdots R_i(K)\cdots\cdots R_j(K)\cdots\cdots\}$

排序之后： $\{\cdots\cdots R_i(K)\ R_j(K)\cdots\cdots\cdots\cdots\}$

例如：

排序前 (56, 34, 47, 23, 66, 18, 82, 47)

若排序后得到结果

(18, 23, 34, 47, 47, 56, 66, 82)

则称该排序方法是稳定的;

若排序后得到结果

(18, 23, 34, 47, 47, 56, 66, 82)

则称该排序方法是不稳定的;

2. 对于不稳定的排序方法，只要能举出一个实例说明即可。

例如： 对{ 4, 3, 4, 2 }进行快速排序，得到{ 2, 3, 4, 4 }

3.希尔排序、快速排序和堆排序是不稳定的排序方法。

四、关于“排序方法的时间复杂度的下限”

本章讨论的各种排序方法，除基数排序外，其它方法都是基于“比较关键字”进行排序的排序方法。

可以证明， 这类排序法可能达到的最快的时间复杂度为O(nlogn)。 （基数排序不是基于“比较关键字”的排序方法，所以它不受这个限制）

10.7 各种方法比较

排序方法	最好情况	最坏情况	平均情况	辅助存储	稳定排序
直接插入排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	√
折半插入排序	$O(n\log_2 n)$	$O(n^2)$	$O(n^2)$	$O(1)$	√
希尔排序	$O(n^{3/2})$	$O(n^2)$	$O(n^{1.3})$	$O(1)$	×
冒泡排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	√
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	×
快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	$O(\log_2 n)$	×
堆 排 序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	×
归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	√
基数排序	$O(d(n+rd))$	$O(d(n+rd))$	$O(d(n+rd))$	$O(n+rd)$	√

本章小结

1. 介绍了排序的定义和各种排序方法的特点，各种方法的排序过程及其依据的原则。基于“关键字间的比较”进行排序的方法可以按排序过程所依据的不同原则分为插入排序、交换排序、选择排序、归并排序和基数排序等五类。

本章小结

2. 分析了各种排序方法的**时间复杂度**。能从**“关键字间的比较次数”**分析排序算法的**平均**情况和**最坏**情况的时间性能。

按平均时间复杂度划分，内部排序可分为三类：**$O(n^2)$**的简单排序方法，**O(nlogn)**的高效排序方法 和 **O(dn)**的基数排序方法。

本章小结

3．排序方法分为“**稳定**”或“**不稳定**”，选择使要分清楚在**什么情况下**要求应用的排序方法必须是稳定的。

课程结束！