数据结构

北京邮电大学信息安全中心
武斌 杨榆

上章内容

上一章（查找）内容：

- 理解“查找表”的结构特点以及各种表示方法的适用性
- 熟练掌握以顺序表或有序表表示静态查找表时的查找方法
- 熟练掌握二叉查找树的构造和查找方法
- 熟练掌握哈希表的构造方法，深刻理解哈希表与其它结构的表的实质性的差别
- 掌握描述查找过程的判定树的构造方法，以及按定义计算各种查找方法在等概率情况下查找成功时的平均查找长度

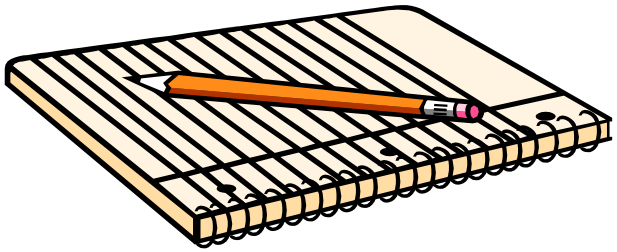
本次课程学习目标

学习完本次课程（内部排序(上)），

您应该能够：

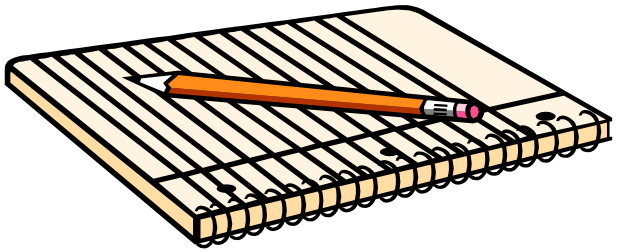
- 理解排序的定义和各种排序方法的特点，并能加以灵活应用
- 了解排序方法有不同的分类方法，基于“关键字间的比较”进行排序的方法可以按排序过程所依据的不同原则分为插入排序、交换排序、选择排序、归并排序和基数排序等五类
- 掌握各种排序方法的时间复杂度的分析方法，能从“关键字间的比较次数”分析排序算法的平均情况和最坏情况的时间性能。

本章课程内容（第十章 内部排序）



- 10.1 概述
- 10.2 插入排序
- 10.3 快速排序
- 10.4 选择排序
- 10.5 归并排序
- 10.6 基数排序
- 10.7 各种内部排序方法的比较讨论

本章课程内容（第十章 内部排序）



- 10.1 概述
- 10.2 插入排序
- 10.3 快速排序
- 10.4 选择排序
- 10.5 归并排序
- 10.6 基数排序
- 10.7 各种内部排序方法的比较讨论

概述

- 排序(Sorting)是计算机程序设计中的一种重要操作，它的功能是**将一个数据元素(或记录)的任意序列，重新排列成一个按关键字有序的序列**。
- 有序的顺序表采用折半查找法，平均查找长度为$\log_2(n+1)-1$，而无序的顺序表只能进行顺序查找，其平均查找长度为$(n+1)/2$。
- 排序可以对单个关键字进行，也可以对多个关键字的组合进行，可统称排序时所依赖的准绳为“**排序码**”。为讨论方便起，本章约定排序只对**单个关键字**进行，并约定排序结果为记录按关键字“**非递减**”的顺序进行排列。

概述

- 假设含有n个记录的序列为：$\{R_1,R_2,\ldots,R_n\}$ (10-1)

  其对应的关键字序列为： $\{K_1,K_2,\ldots,K_n\}$

  需确定$1,2,\ldots,n$的一种排列$p_1,p_2,\ldots,p_n$，使其相应的关键字满足如下的非递减关系：$K_{p_1} \le K_{p_2} \le \ldots \le K_{p_n}$

  即使式(10-1)的序列成为一个**按关键字有序**的序列：$\{R_{p_1},R_{p_2},\ldots,R_{p_n}\}$ **这样一种操作叫做排序**。

概述

- 上述排序定义中的关键字K_i可以是记录$R_i(i=1,2,\ldots,n)$的主关键字，也可以是记录R_i的次关键字，甚至是若干数据项的组合。
- 若K_i是**主关键字**，则任何一个记录的无序序列经排序后得到的结果是**惟一**的；
- 若K_i是**次关键字**，则排序的结果**不惟一**，因为待排序的记录序列中可能存在两个或两个以上关键字相等的记录。
- 假设$K_i = K_j$ $(1\le i\le n,\ 1\le j\le n, i\ne j)$，且在排序前的序列中**$R_i$领先于$R_j$**(即i<j)。若在排序后的序列中**$R_i$仍然领先于$R_j$**，则称所用的**排序方法是稳定的**；反之，若可能使排序后的序列中R_j领先于R_i，则称所用的**排序方法是不稳定的**。

概述

- 根据在排序过程中涉及的存储器不同，可将排序方法分为两大类：
  - (1) 内部排序：指的是待排序记录存放在计算机随机存储器中进行的排序过程；
  - (2) 外部排序：指的是待排序记录的数量很大，以致内存一次不能容纳全部记录，在排序过程中尚需对外存进行访问的排序过程。
- 本章只学习内部排序

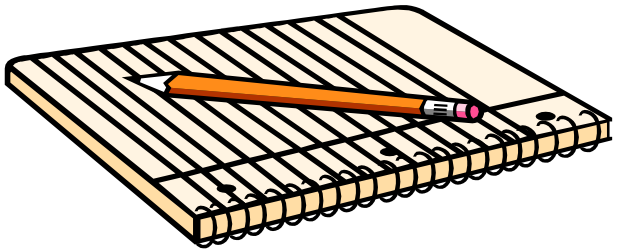
概述

- 内部排序的过程是一个**逐步扩大记录有序序列长度**的过程。
- 如果按排序过程中**依据的不同原则**对内部排序方法进行**分类**，则大致可分为**插入排序**、**交换排序**、**选择排序**、**归并排序**和**基数排序**等五类。
- 如果按内部排序过程中**所需的工作量**来区分，可分为3类：
  - (1) **简单的排序方法**，其时间复杂度为O (n^2)；
  - (2) **先进的排序方法**，其时间复杂度为O (nlogn)；
  - (3) **基数排序**，其时间复杂度为O (dn)；
- 通常，在排序的过程中需要进行下列**两种基本操作**：
  - (1) 比较两个关键字的大小；
  - (2) 将记录从一个位置移动至另一个位置。

概述

- 待排序的记录序列可有下列3种存储方式：
  - (1) 待排序的一组记录存放在地址连续的一组存储单元上。它类似于线性表的顺序存储结构，在序列中相邻的两个记录R_j和R_{j+1}(j=1,2,…,n-1)，它们的存储位置也相邻。在这种存储方式中，记录之间的次序关系由其存储位置决定，则实现排序必须借助移动记录。
  - (2) 一组待排序记录存放在静态链表中，记录之间的次序关系由指针指示，则实现排序不需要移动记录，仅需修改指针即可，称为(链)表排序。
  - (3) 待排序记录本身存储在一组地址连续的存储单元内，同时另设一个指示各个记录存储位置的地址向量，在排序过程中不移动记录本身，而移动地址向量中这些记录的“地址”，在排序结束之后再按照地址向量中的值调整记录的存储位置，称为地址排序。

本章课程内容（第十章 内部排序）



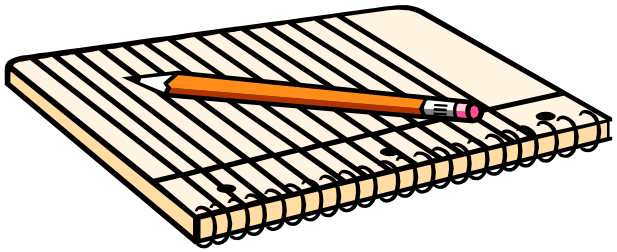
- 10.1 概述
- 10.2 插入排序
- 10.3 快速排序
- 10.4 选择排序
- 10.5 归并排序
- 10.6 基数排序
- 10.7 各种内部排序方法的比较讨论

概述

- **假设：**在本章的讨论中，待排序的一组记录以上述第一种方式存储，且记录的关键字均为整数。
- **因此**，待排记录的数据类型设为：

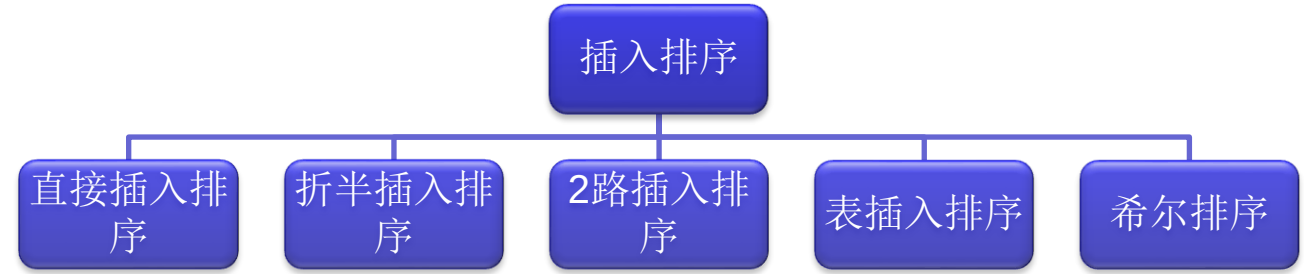
```
#define MAXSIZE  20   // 一个用作示例的小顺序表的最大长度
typedef  int  KeyType;    // 定义关键字类型为整数类型
typedef struct {
      KeyType key;                 // 关键字项
      InfoType otherinfo;          // 其它数据项
} RedType;                        // 记录类型并设定关键字为整型
typedef struct {
      RedType   r[MAXSIZE+1];     // r[0]闲置或作为判别标志的“哨兵”单元
      int       length;           // 顺序表长度
  } SqList;                       // 顺序表类型
```

本章课程内容（第十章 内部排序）



- 10.1 概述
- 10.2 插入排序
- 10.3 快速排序
- 10.4 选择排序
- 10.5 归并排序
- 10.6 基数排序
- 10.7 各种内部排序方法的比较讨论

插入排序

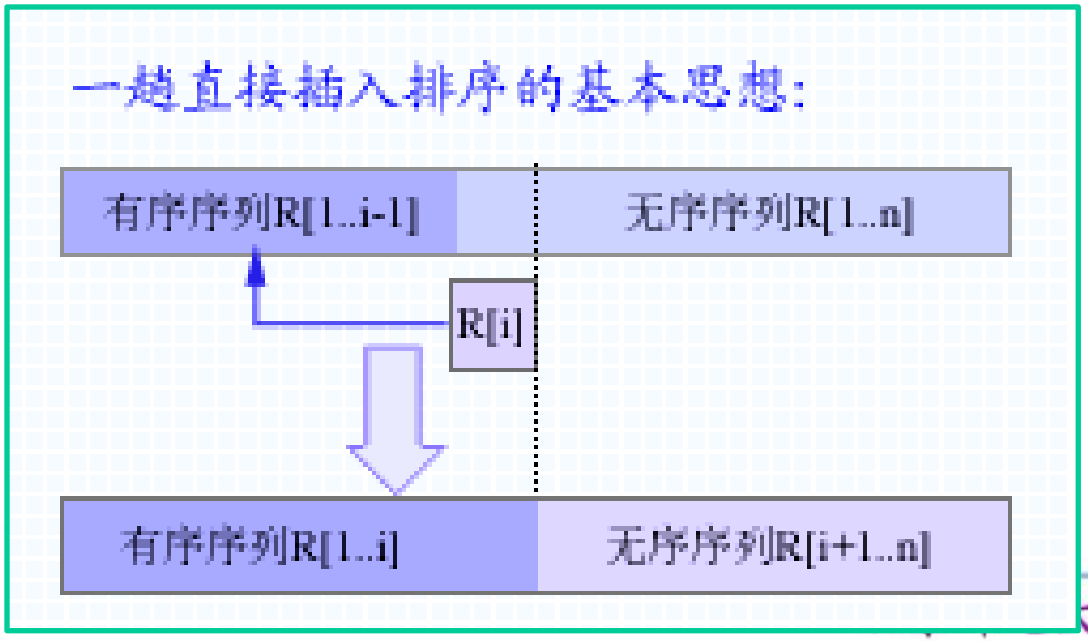


插入排序

●一、直接插入排序

插入排序的基本操作：将一个记录插入到已排好序的有序表中，从而得到一个新的、记录数增1的有序表。

即:在对记录序列R[1..n]的排序过程中，区段R[1..i-1]中的记录已按关键字非递减的顺序排列，将 R[i] 插入到有序序列 R[1..i-1] 中，使区段 R[1..i] 中的记录按关键字非递减顺序排列，如下图所示。演示flash\chap10\10-1-1.swf

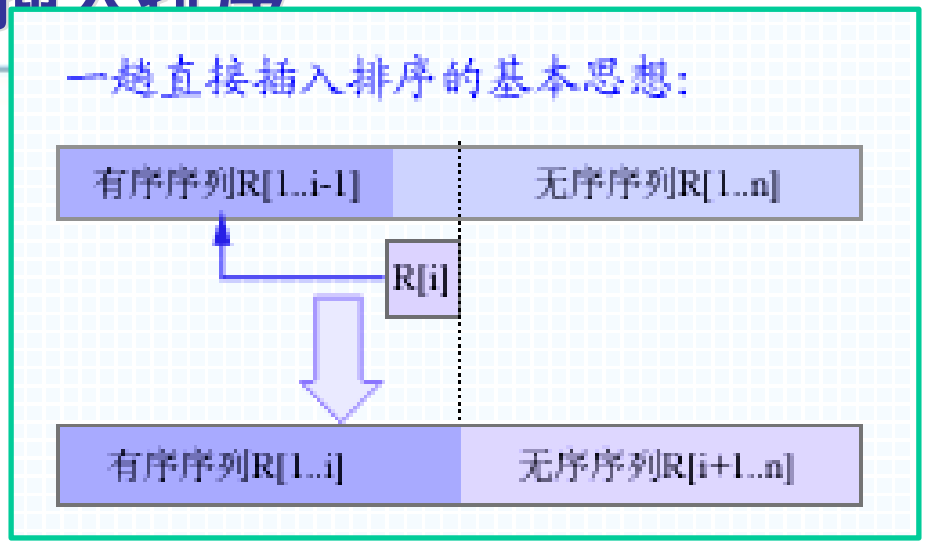


插入排序

- 由此实现一趟插入排序的步骤为：
  - 1) 在 R[1..i-1] 中查找 R[i] 的插入位置，即确定j(0≤j＜i)使得

    R[1..j].key≤R[i].key＜R[j+1..i-1].key
  - 2) 将 R[j+1..i-1] 中的记录后移一个位置；
  - 3) 将 R[i] 插入到 j+1 的位置。
- 和第九章中讨论的顺序查找类似，为了避免在查找过程中判别循环变量是否出界，**设置 R[0] 为监视哨**，并在查找的同时进行“**记录后移**” 演示flash\chap10\10-2-2.swf

插入排序



● 算法10.1

```
void InsertSort ( 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 ) {      // 将 L.r[i] 插入有序子表
             L.r[0] = L.r[i];                     // 复制为哨兵
             L.r[i]=L.r[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
```

0	1	2	…	k	k+1	…	i-2	i-1	i

0	1	2	…	k	k+1	…	i-2	i-1	i

插入排序

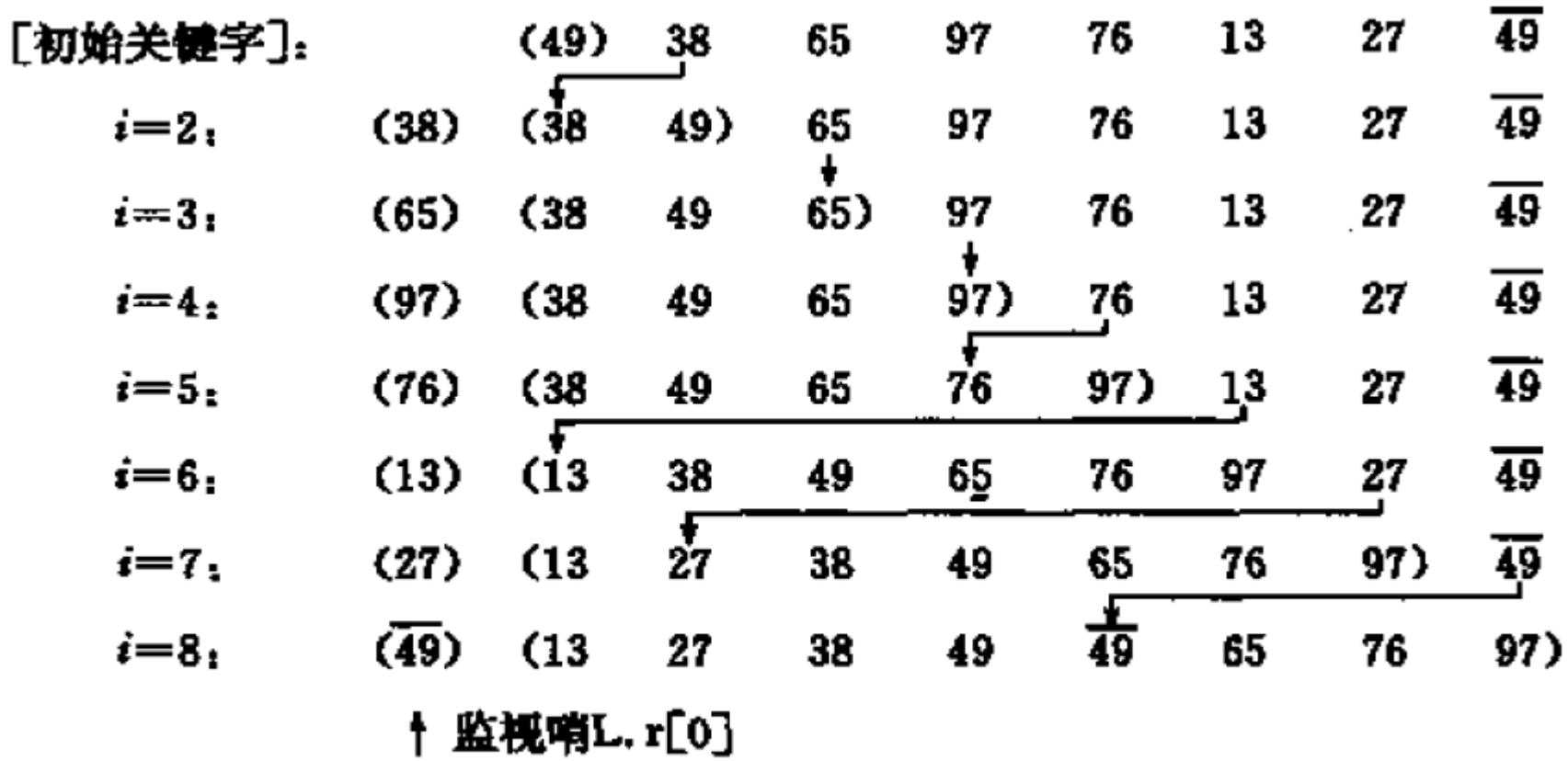


图 10.1　直接插入排序示例

插入排序的关键：

1.找位置（采用顺序查找的方式，设监视哨）；

2.为新元素“腾地”，边比较边移动

插入排序

●直接插入排序的时间复杂度

两个基本操作是：**(关键字间的)比较**和**(记录的)移动**。因此排序的时间性能取决于排序过程中这两个操作的次数，取决于待排记录序列的状态。

- 当待排记录处于“**正序**”(即记录按关键字从小到大的顺序排列)的情况时，所需进行的**关键字比较(n-1)**和记录**移动**的次数最少**(0)**。
- 当待排记录处于“逆序”(即记录按关键字从大到小的顺序排列)的情况时，关键字比较和记录移动的次数最多,分别为：

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

待排记录序列状态	"比较"次数	"移动"次数
正序	n-1	0
逆序	$\frac{(n+2)(n-1)}{2}$	$\frac{(n+4)(n-1)}{2}$

插入排序

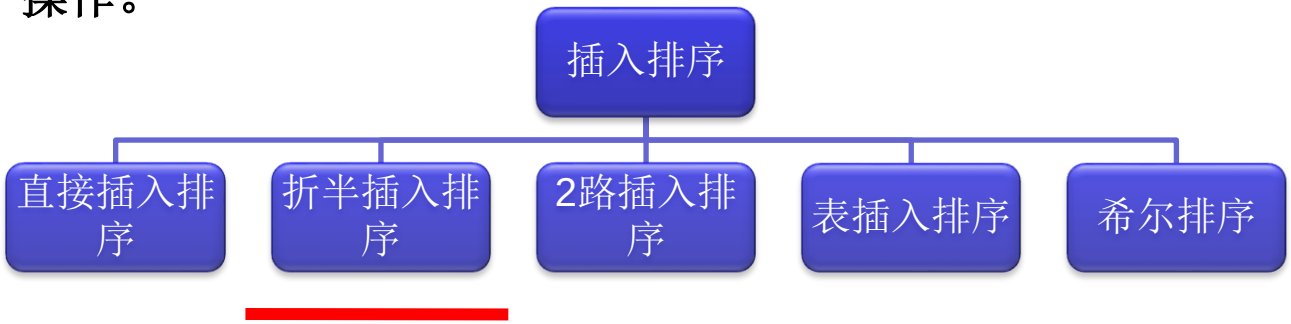
- 说明：
  - "移动记录"的次数包括监视哨的设置。
  - 先分析一趟直接插入排序的情况：
    - 若 L.r[i].key≥L.r[i-1].key，只进行“1”次比较，不移动记录；
    - 若 L.r[i].key＜L.r[1].key，需进行“i”次比较，i+1次移动。
  - 整个插入排序的过程中，i从2至 n，则得（正序时的比较和移动次数；逆序时的比较和移动次数）

$$\sum_{i=2}^{n}1=n-1;\sum_{i=2}^{n}0=0 \qquad \sum_{i=2}^{n}i=\frac{(n+2)(n-1)}{2} \qquad \sum_{i=2}^{n}(i+1)=\frac{(n+4)(n-1)}{2}$$

- 若待排记录序列处于随机状态，则可以最坏和最好的情况的平均值作为插入排序的时间性能的量度。一般情况下，**直接插入排序的时间复杂度为$O(n^2)$**。

插入排序

直接插入排序算法简单，适用于记录数n较小的情况。当n较大时，可以从减少“比较”和“移动”这两种操作的次数着眼改进算法。折半插入排序算法利用“折半查找”代替直接插入排序中的“顺序查找”操作。



插入排序

- 二、其他插入排序
  - 1、折半插入排序

由于插入排序的基本思想是在一个**有序序列**中插入一个新的记录，则可以**利用“折半查找”查询插入位置**，由此得到的插入排序算法为“折半插入排序”。演示flash\chap10\10-2-3.swf

从动画演示可见，折半插入排序过程中的折半查找的目的是查询插入点，因此不论是否存在和给定值相同的关键字，结束查找过程的条件都是high<low，并且插入位置为low指示的地方。

"折半插入"不失为是一条减少关键字比较次数的途径，它是"归并插入排序“(可使排序过程中的关键字比较次数达到最少的一种排序方法)的基本依据。

Idx	0	1	2	3	4	5	6	7	8
i=0		49	38	65	97	76	13	27	49’
i=7	27	(13	38	49	65	76	97）		

- 算法10.2 **void** BInsertSort (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];                          // 将L.r[i]暂存到L.r[0]
      low = 1; high = i-1;
      while (low<=high) {// 在r[low..high]中折半查找有序插入的位置
        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
      } // while
      for ( j=i-1; j>=high+1; --j ) L.r[j+1] = L.r[j];   // 记录后移
      L.r[high+1] = L.r[0];                       // 插入
    }
} // BInsertSort
```

- 折半插入排序只能减少排序过程中关键字比较的时间，并不能减少记录移动的时间，因此**折半插入排序的时间复杂度仍为$O(n^2)$**。

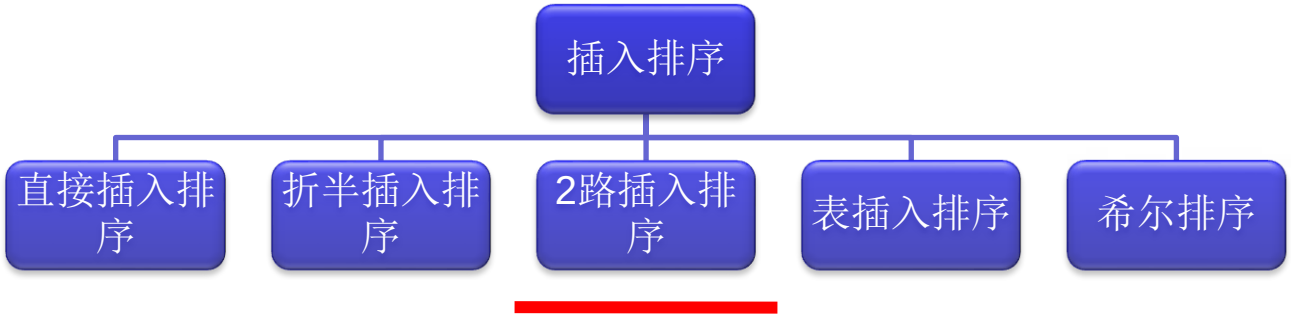
插入排序

2-路插入排序是在折半排序基础上改进，目的是减少排序过程中移动记录的次数。需要增加n个记录的辅助空间d。

以d[1]（值同L.r[1]）为参照，关键字小于参照记录的记录，则插入参照记录之前的有序表，否则插入之后的有序表。

综上，相对于直接插入排序和折半插入排序，2-路插入排序时，插入对象变为两个有序表，问题规模降低，操作次数减少。

实现时，可将d视为循环数组。



插入排序

2、2-路插入排序

2-路插入排序是在折半插入排序的基础上再改进之，其目的是减少排序过程中移动记录的次数，但为此需要n个记录的辅助空间。

- 具体做法是： 另设一个和L.r同类型的数组d，首先将L.r[1]赋值给d[1]，并将d[1]看成是排好序的序列中处于中间位置的记录，然后从L.r中第2个记录起依次插入到d[1]之前或之后的有序序列中。先将待插记录的关键字和d[1]的关键字进行比较，若L.r[i].key< d[1].key，则将L.r[i]插入到d[1]之前的有序表中。反之，则将L.r[i]插入到d[1]之后的有序表中。
- 在2-路插入排序中，移动记录的次数约为$n^2/8$。因此，2-路插入排序只能减少移动记录的次数，而不能绝对避免移动记录。并且，当L.r[1]是待排序记录中关键字最小或最大的记录时，2-路插入排序就完全失去它的优越性。

插入排序

Idx	1	2	3	4	5	6	7	8
i=0	49	38	65	97	76	13	27	49’
i=1	(49)							
	f(1), t(1)							
i=2	(49)							(38)
	t(1)							f(8)
i=3	(49	65)						(38)
		t(2)						f(8)
i=4	(49	65	97)					(38)
			t(3)					f(8)
i=5	(49	65	76	97)				(38)
				t(4)				f(8)
i=6	(49	65	76	97)			(13	38)
				t(4)			f(7)	

Idx为记录在数组中的位置；**i=0**为初始状况，此时所有记录尚未排序；**f**和**t**分别为参考元素前有序表表头位置，以及参考元素后（含参考元素）有序表表尾位置。

i=2:插入**38,38<49**，应插入**49**之前有序表；插入后，表头指针**f**循环前移到**f=8**；

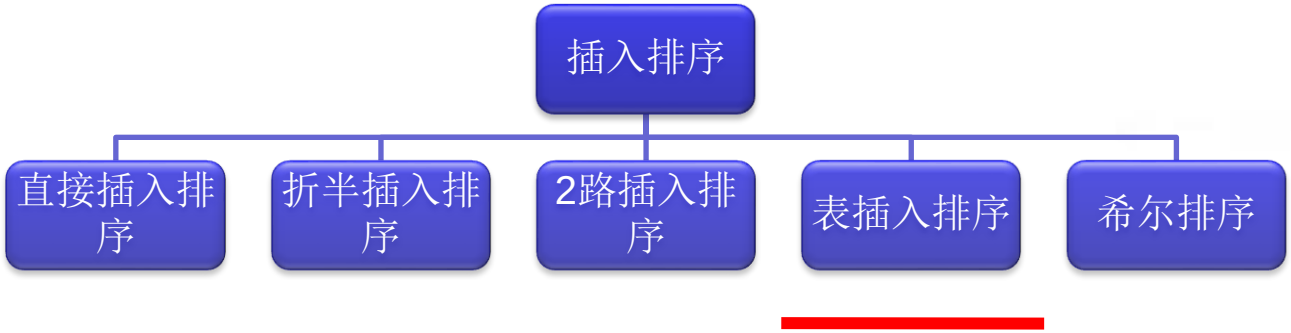
i=5:76大于**49**，应插入**49**后有序表；采用折半查找确定插入位置为**idx=3**，插入位置开始到表尾的所有元素向“后”搬移一个位置；表尾指针循环后移到**t=4**;

插入排序

2-路插入排序不能绝对避免记录的移动，当操作记录L.r[1]是所有记录中关键字最大或最小的记录时，2-路插入排序完全失去优越性。要避免移动记录，只能改变存储结构，进行表插入排序。

表插入排序中，记录通过“指针”链接在一起。插入记录时，只要修改链接指向，不用移动记录。

同时，排序后生成的是有序链表，不能对其进行随机查找。要支持折半查找等操作，还需要重排记录。



插入排序

3、表插入排序

表插入排序是以静态链表作待排记录序列的存储结构实现的插入排序。这个静态链表由**存储记录的顺序表**和**附加的指针数组**构成，静态链表中的指针实际上指的是数组的下标。

- 表插入排序分**两步**进行：首先构造一个有序链表；然后按照“附加指针”的指示将结点中的记录重新排列成一个有序序列。演示flash\chap10\10-2-4.swf
- 从例子中可见，构成有序链表的过程和前面讨论的直接插入排序的过程基本相同，先生成一个只含一个记录的有序链表，之后将从第2个至最后一个记录逐个插入，差别仅在于查找插入位置是从前到后进行查询，直至找到一个记录的关键字大于当前待插入的记录的关键字，因此，在查询过程中应该保持一个指"前驱"结点的指针。

入排序

- 所谓“**重排记录**”是将记录移动到正确位置（即按关键字从小至大有序排列）。在重排的过程中设置了三个指针：
  - 其中 **i 指示当前需要“归位”的记录的正确位置**；
  - **p 指示当前需要“归位”的记录的原始位置**；
  - **q 指示第 i+1 个记录（即指针 p 所指记录的后继）的原始位置**。
- 显然，重排的主要操作是互换 p 和 i 所指记录，以便使第 i 小关键字记录归位至正确位置，但由于互换记录**破坏了链表的链接关系**，可利用和当前已归位记录相应的指针予以**修补**。
- 如何找需要“归位”记录的原始位置？记前一条记录的next指针为p。由于 i 值从小至大变化，因此第 i条需要“归位”记录的原始位置 p 必定大于i；否则，说明该位置的记录已被移走，应根据next指针定位。

Idx		i		p		q
key		k_i		k_p		k_q
next		n_i		n_p		n_q

重排

Idx		i		p		q
key		k_p		k_i		k_q
next		n_p		n_i		n_q

修正

Idx		i		p		q
key		k_p		k_i		k_q
next		**p**		n_i		n_q

插入排序

Idx	1	2	3	4	5	6	7	8
	49	38	65	97	76	13	27	52
6	8	1	5	0	4	7	2	3
i=1	13	38	65	97	76	49	27	52
p=6 q=7	6(7)	1	5	0	4	8	2	3
i=2	13	27	65	97	76	49	38	52
p=7 q=2	6(7)	7(2)	5	0	4	8	1	3
i=3	13	27	38	97	76	49	65	52
p=(2),7 q=1	6(7)	7(2)	7(1)	0	4	8	5	3
i=4	13	27	38	49	76	97	65	52
p=(1),6 q=8	6(7)	7(2)	7(1)	6(8)	4	0	5	3

重排记录时，可以充分利用现有结构，不增加额外存储空间。

初始时表头为**L.r[6]**（**L.r[0].next**指示表头位置），表尾为**L.r[4]**（**next**为**0**）。**p**保存当前元素**i**更新后的**next**，**q**保存当前元素**i**更新前的**next**。

i=1时,重排后，**L.r[1]**应保存链表中的第一个结点（即表头），所以**L.r[1]**应该保存**L.r[6]**信息。

那么**L.r[1]**怎么办？对调**L.r[6]**和**L.r[1]**。

对调操作不应改变链表结点顺序，所以**L.r[1]**被调换到新位置后，其**next**值不变。

而**L.r[6]**被换到**L.r[1]**后，若在后续遍历中被访问，其实应是原链表结点**L.r[1]**被访问（被调到位置**6**）。为维持结点访问顺序，位置对调后，**L.r[1]**的**next**域应指向**6**（**p=6**）.

同时，为了访问**L.r[1]**的后续结点，在更新其**next**指向前，应保存其**next**值（**q=7**）。

● 算法10.3容易看出，在重排过程中至多进行3(n-1)次移动记录，然而整个表插入排序过程也仅仅是减少了移动记录的时间，所以它的时间复杂度仍为$O(n^2)$。

```
void Arrange (SLinkListType &SL)
{    // 根据静态链表SL中各结点的指针值调整记录位置，使得SL中记录
     // 按关键字非递减有序顺序排列
     p = SL.r[0].next;                    // p 指示第一个记录的当前位置
     for ( i=1; i<L.length-1; ++i )
      {// SL.r[1..i-1]中记录已按关键字有序排列，第 i 个记录 在L中的当前位置应不小于 i
        while (p<i) p = SL.r[p].next;     // 找到第i个记录，并用p指示其在L中当前位置
        q = SL.r[p].next;                 // q 指示尚未调整的后继记录
        if ( p!= i ) {
           SL.r[p]←→SL.r[i];              // 交换记录，使第 i 个记录到位
           SL.r[i].next = p;              // 指向被移走的记录，使得以后可由while循环找回
        } // if
        p = q;                            // p 指向下一个将要调整的记录
     } // for
} // Arrange
```

Idx		i		p		q
key		k_i		k_p		k_q
next		n_i		n_p		n_q

Idx	重排	i		p		q
key		k_p		k_i		k_q
next		n_p		n_i		n_q

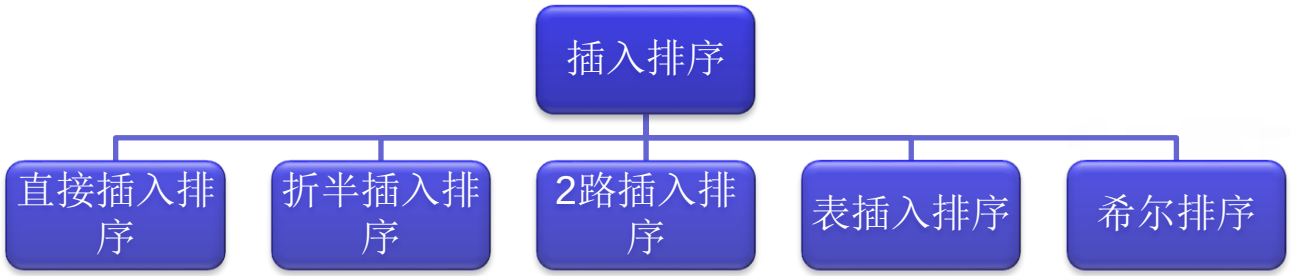
Idx	修正	i		p		q
key		k_p		k_i		k_q
next		**p**		n_i		n_q

插入排序

当待排记录为正序时，直接插入排序时间复杂度从$O(n^2)$提升到$O(n)$。可见，若待排记录按关键字“基本有序”时，排序算法时间效率能够提升。基于此，设计得到希尔排序算法，又称“缩小增量排序”。

希尔排序的基本思想是：先将整个待排记录序列分割成子序列进行直接插入排序，待序列中的记录“基本有序”时再对全体记录进行一次直接插入排序。

分割子序列意味着降低问题规模，对子序列的直接插入排序效率高。随着增量的缩小，虽然子序列的长度增加，但每个子序列经先前排序操作，接近或已经基本有序，所以时间效率也是良好的。



插入排序

●三、希尔排序

希尔排序又称**“缩小增量排序”**，它的基本思想是，先对待排序列进行“宏观调整”，待序列中的记录“基本有序”时再进行直接插入排序。

- 所谓“基本有序"是指，在序列中的各个关键字之前，只存在少量关键字比它大的记录。
- 例如一个含11个关键字的序列 (16，25，12，30，47，11，23，36，9，18，31)(直接插入排序演示flash\chap10\10-2-4-1.swf)，先对它进行“增量为5”的插入排序，即分别使(R_1,R_6,R_{11})、(R_2,R_7)、(R_3,R_8)、(R_4,R_9)和(R_5,R_{10})为有序序列，然后将增量“缩小到3”，排序结果使(R_1,R_4, R_7,R_{10})、 (R_2,R_5,R_8,R_{11})和(R_3,R_6,R_9)分别成为有序序列，此时序列中在关键字18，23，25，31和47之前的关键字均比它们小，即在进行最后一趟排序时这几个关键字都不需要“往前进行”插入，之后经过最后一趟插入排序即得到有序序列。演示flash\chap10\10-2-5.swf。

插入排序

- 可以看出：希尔排序在前几趟的排序过程中，**关键字较大的记录是“跳跃式”地往后移动**，从而使得在进行最后一趟插入排序之前序列中记录的关键字已基本有序，由此**减少**了整个排序过程中所需进行的**“比较”和“移动”的次数**。

● 算法 10.4

```
void ShellInsert ( SqList &L，int dk)
{     // 对顺序表L作一趟增量为dk的希尔排序
    for ( i=dk+1; i<=L.length; ++i )
      if (LT(L.r[i].key, L.r[i-dk].key )) {     // 需将L.r[i]插入有序增量子表
        L.r[0] = L.r[i];                        // 暂存在L.r[0]
        for ( j=i-dk; j>0 &&LT(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
        L.r[j+dk] = L.r[0];                     // 插入到正确位置
      } // if
} // ShellInsert
```

Idx	1	2	3	4	5	6	7	8
	49	38	65	97	76	13	27	52

插入排序

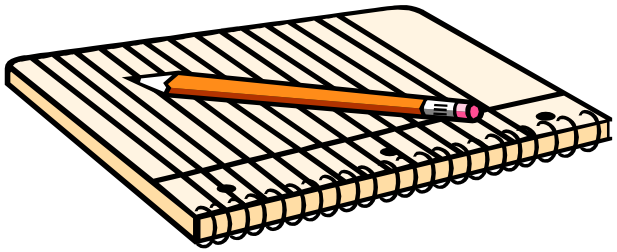
- 算法10.5

```
void ShellSort (SqList &L, int dlta[], int t)
{      // 按增量序列 dlta[0..t-1] 对顺序表L作希尔排序
    for (k=0; k<t; ++t)
        ShellInsert(L, dlta[k]);     // 一趟增量为 dlta[k] 的插入排序
} // ShellSort
```

- 希尔排序的时间复杂度和所取增量序列相关，例如已有学者证明，当增量序列为dlta[k]=$2^{t-k-1}-1(1\le k\le t\le \lfloor \log_2(n+1) \rfloor)$时，**希尔排序的时间复杂度为$O(n^{3/2})$**。

- 增量序列可以有各种取法，但需**注意**：应使增量序列中的值没有除1之外的公因子，并且最后一个增量值必须等于1。如dlta[]=……,9,5,3,2,1或dlta[]=……,31,15,7,3,1或dlta[]=……,40,13,4,1等。

本章课程内容（第十章 内部排序）



- 10.1 概述
- 10.2 插入排序
- 10.3 快速排序
- 10.4 选择排序
- 10.5 归并排序
- 10.6 基数排序
- 10.7 各种内部排序方法的比较讨论

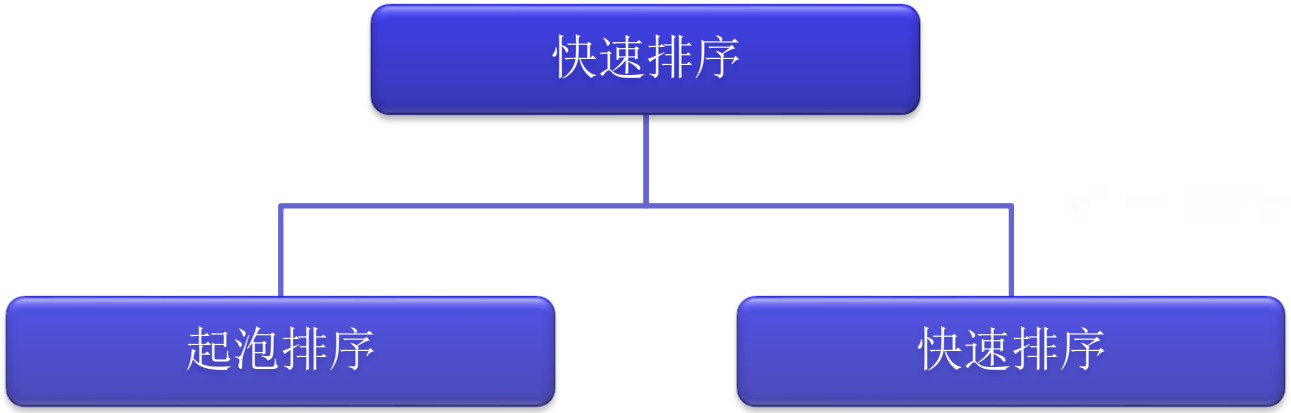
快速排序

快速排序的典型操作是交换记录。

起泡排序（冒泡法）是最简单的快速排序。起泡排序的基本思路是：每轮依次比较相邻记录，若它们依关键字是逆序，则交换这两条记录。

每轮操作将使得关键字较小的记录向前移动，好像“起泡在上升”，整个序列的也朝着“有序”的方向变化。

当一轮比较完毕后，都不需要进行“交换”操作，意味着完成了排序操作。



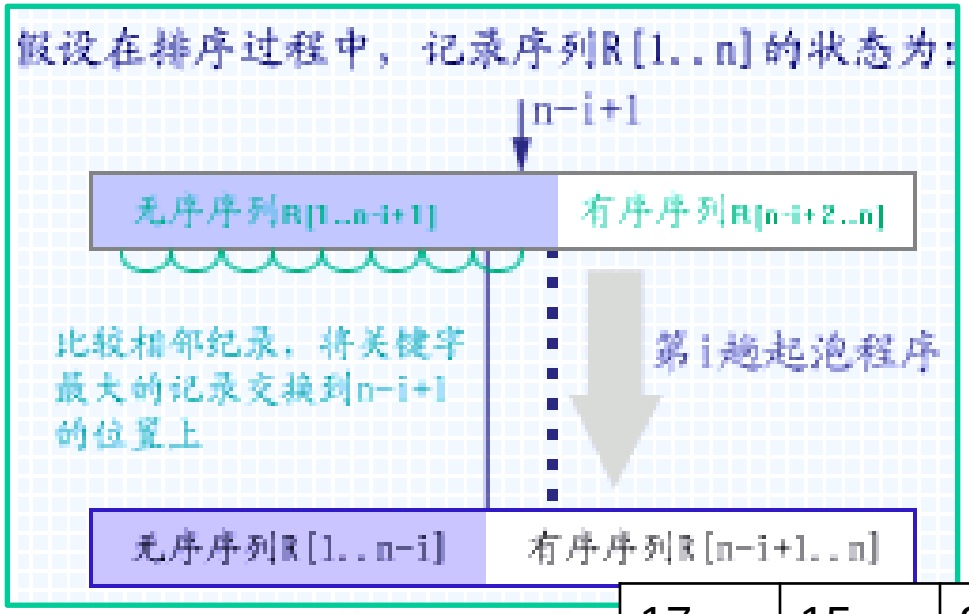
快速排序

●1、起泡排序

起泡排序是交换类排序方法中的一种简单排序方法。其基本思想为：依次比较相邻两个记录的关键字，若和所期望的相反，则交换这两个记录。

- 如下图所示，在第i趟起泡排序之前，区段R[n-i+2..n]中的记录已按关键字从小到大有序排列，而区段R[1..n-i+1]中的记录不一定有序，但该区段中所有记录的关键字均不大于有序序列中记录的关键字(即小于或等于R[n-i+2].key)。

21	17	15	30	6	39	66	70



第4趟起泡排序

17	15	21	6	30	39	66	70

快速排序

- 第i趟起泡排序的操作为，从第1个记录起，逐个和相邻记录的关键字进行比较，若第j(1≤j≤n-i)个记录的关键字大于第j+1个记录的关键字，则互换记录，由此可将区段R[1..n-i+1]中关键字最大的记录“交换”到R[n-i+1]的位置上，从而使有序序列的长度增1。
- 显然，如果第i趟起泡排序的过程中，没有进行任何记录的交换，则表明区段R[1..n-i+1]中的记录已经按关键字从小到大有序排列，由此不再需要进行下一趟的起泡，即起泡排序已经完成，可见**排序结束的条件是(i=n-1)或者(第i趟的起泡中没有进行记录交换)。**
- 演示flash\chap10\10-3-1.swf

快速排序

- 算法10.6 **void** BubbleSort(SqList **&**L)

```
{      // 对顺序表L作起泡排序
    for (i=L.length, change=TRUE; i>1 && change; --i) {
        change = FALSE;
        for (j=1; j<i; ++j)
            if (L.r[j].key > L.r[j+1].key)
                { L.r[j]←→L.r[j+1]; change = TRUE }
    }// for i
} // BubbleSort
```

15	30	44	35	62	79

- 算法中设立了一个标志一趟起泡中是否进行了交换记录操作的布尔型变量change，在每一趟起泡之前均将它设为"FALSE"，一旦进行记录交换，则将它改为"TRUE"，因此 change=TRUE 是进行下一趟起泡的必要条件。

快速排序

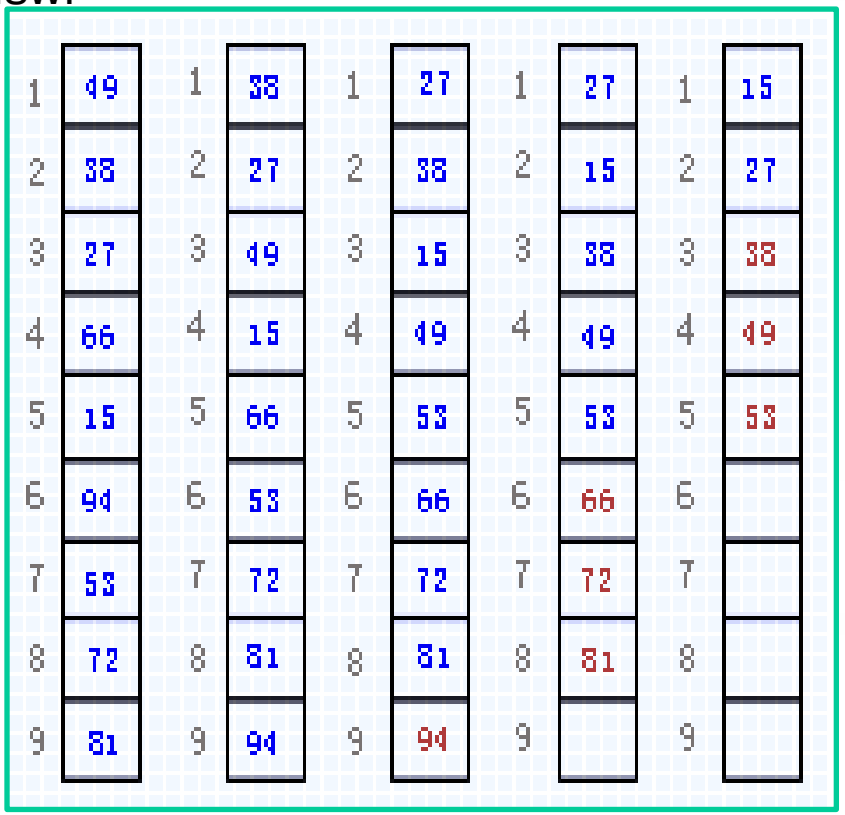
- 分析**起泡排序的时间复杂度**：和直接插入相似，排序过程中所进行的“比较” 和 “移动” 操作的次数取决于待排记录序列的状态，在待排记录处于“正序”时取最小值，此时只需进行一趟起泡排序，反之，在待排记录处于“逆序”时取最大值，此时需进行 n-1趟起泡，列表如下：

待排记录状态	“比较”次数	“移动”次数
正序	**n-1**	**0**
逆序	$\frac{n(n-1)}{2}$	$\frac{3n(n-1)}{2}$

快速排序

- 在算法10.6中，经过每一趟起泡，待排记录序列的上界 i 都只是减1。但实际上，有的时候起泡的上界可以缩减不止1个记录的位置，例如右下侧所示例子。演示flash\chap10\10-3-2.swf

- 从这个例子可见，在一趟起泡的过程中，有可能只是在区段的前端进行记录的交换，而其后端记录已经按关键字有序排列，由此应在算法中设置一个指示“最后一个进行交换的记录的位置”的变量。**改进后的起泡算法**如算法10.7所示。



快速排序

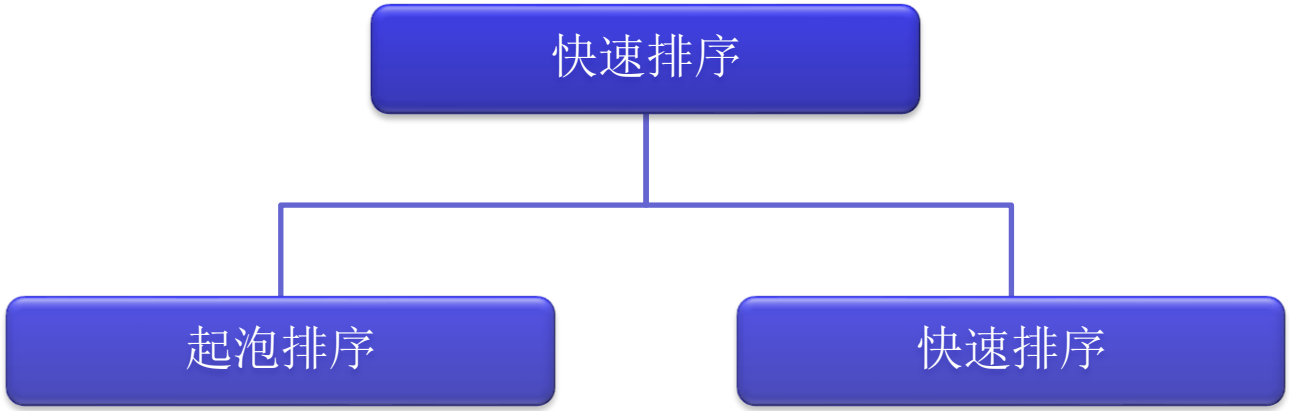
1	2	3	4	5	6	7	8	9
38	27	49	15	66	53	72	81	94

- 算法10.7

```
void BubbleSort( SqList &L )
{   // 对顺序表L作起泡排序
      i = L.length;
      while (i >1) {                          // i>1 表明上一趟曾进行过记录的交换
         lastExchangeIndex = 1;
         for (j = 1; j < i; j++){
            if (L.r[j+1].key < L.r[j].key) {
               L.r[j] ←→L.r[j+1];             // 互换记录
               lastExchangeIndex = j;         // 记下进行交换的记录的位置
            } // if
          } // for
         i = lastExchangeIndex;      // 一趟起泡后仍处于无序状态的最后一个记
                                        录的位置
      } // while
} // BubbleSort
```

快速排序

快速排序是起泡排序的一种改进，其基本思想：通过一趟排序将待排记录分割成独立的两部分，其中一部分记录的关键字均比另一部分记录的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序。



快速排序

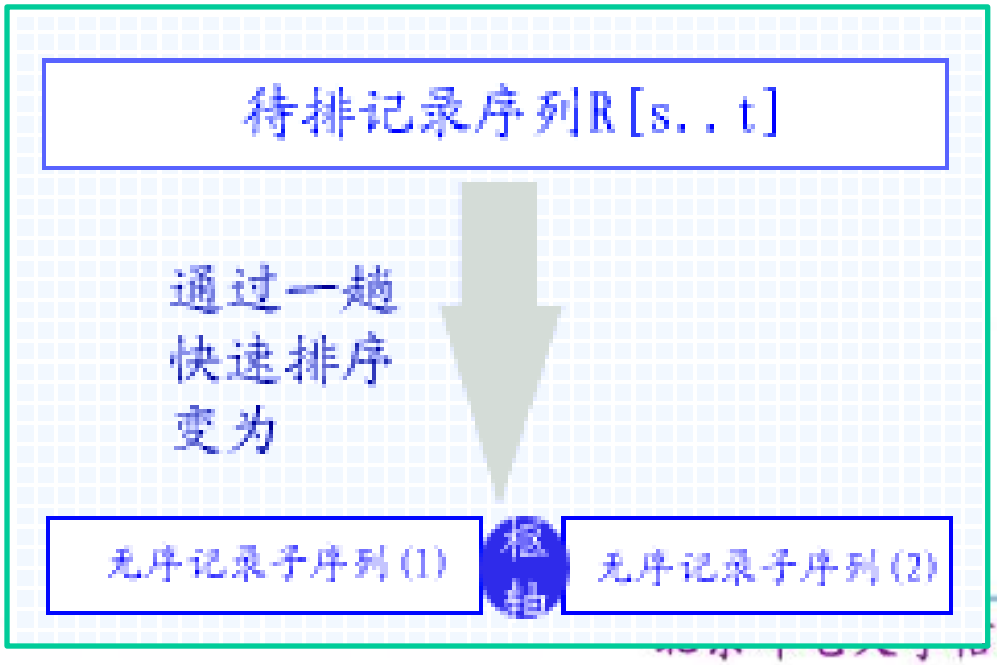
●2、快速排序

快速排序是起泡排序的一种改进，其基本思想：**通过一趟排序将待排记录分割成独立的两部分，其中一部分记录的关键字均比另一部分记录的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序**。

- **例如**，关键字序列 (52, 49, 80, 36, 14, 75, 58, 97, 23, 61)
  - 以52为**枢轴**，剩余记录，关键字比它小记录排在它之前，关键字较大的记录放在它之后。以枢轴位置为界，将序列分为两个子序列。这个过程称为一趟快速排序（或一次划分）
  - 经第1趟快速排序之后为 (23, 49, 14, 36) 52 (75, 58, 97, 80, 61)
  - 经多趟快速排序之后为 (14) 23 (49, 36) 52 (61, 58) 75 (80, 97)
  - 快速排序完成后为 (14, 23, 36, 49, 52, 58, 61, 75, 80, 97)

快速排序

- 选定一个关键字介于“中间”的记录，从而使剩余记录可以分成两个子序列分别继续排序，通常称该记录为“**枢轴**”。
  - 如下图所示，假设一趟快速排序之后枢轴记录的位置为 i，则得到的无序记录子序列R[s..i-1]中记录的关键字均小于枢轴记录的关键字，反之，得到的无序记录子序列R[i+1..t]中记录的关键字均大于枢轴记录的关键字，由此这两个子序列可分别独立进行快速排序。



快速排序

- 一趟快排也称“一次划分”，即将待排序列 R[s..t]“划分”为两个子序列R[s..i-1] 和 R[i+1..t]，i 为一次划分之后的枢轴位置。可以取待排序列中任何一个记录作为枢轴，但为方便起见，**通常取序列中第一个记录 R[s] 为枢轴**，以它的关键字作为划分的依据。
  - 划分可如下进行：设置两个指针 low 和 high，分别指向待排序列的低端 s 和高端 t。若 R[high].key<R[s].key，则将它移动至枢轴记录之前；反之，若 R[low].key>R[s].key，则将它移动至枢轴记录之后，并为避免枢轴来回移动，可先将枢轴 R[s] 暂存在数组的闲置分量 R[0] 中。如动画演示为“一次划分”过程的一个例子。
    演示flash\chap10\10-3-4.swf

快速排序

Idx	1	2	3	4	5	6	7	8
初始	49	38	65	97	76	13	27	49’
49	L=1							H=8
1次	27	38	65	97	76	13	27	49’
49			L=3				H=7	
2次	27	38	65	97	76	13	65	49’
49			L=3			H=6		
3次	27	38	13	97	76	13	65	49’
49				L=4		H=6		
4次	27	38	13	49	76	97	65	49’
49				L=4 H=4				

一趟快速排序（一次划分）

```
while (low<high && L.r[high].key>=pivotkey)    --high;
L.r[low] = L.r[high];                          // 将比枢轴记录小
```

快速排序中，根据枢轴，将初始序列划分为两个序列。关键字小于枢轴的记录放到枢轴之前。如何放？快速算法采用交换记录的策略。

初始时，设置两个指针**L**和**H**分别指向两个子序列可存放记录位置。将枢轴记录**49**放入**L.r[0]**，为交换记录腾出空地。

1.检查**H**指向记录，关键字**49’**等于枢轴，无需交换该记录，**H**前移一位。

2.检查**H**指向记录，关键字**27**小于枢轴，交换该记录到**L**所指向位置（**L**所指向记录在初始化时备份到**L.r[0]**）。交换后，**H**指向位置腾空，可保存新元素，所以，从另一端扫描，**L**后移直到其指向记录的关键字大于枢轴。

3.交换**L**所指记录到**H**指向位置。此时**L**所指位置腾空，从另一端扫描，**H**前移直到其所指记录关键字小于枢轴。

4.如此交替扫描，交换元素。当**L=H**时，存放枢轴于此位置，序列一分为二。

快速排序

- 算法10.10

```
int Partition (SqList &L, int low, int high)
{  // 交换顺序表中子表L.r[low..high]的记录，枢轴记录到位，并返回其所在位置，
   // 此时，在它之前（后）的记录的关键字均不大（小）于它的关键字。
    L.r[0] = L.r[low];                          // 用子表的第一个记录作枢轴记录
    pivotkey = L.r[low].key;                    // 枢轴记录关键字
    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];                     // 将比枢轴记录小的记录移到低端
      while (low<high && L.r[low].key<=pivotkey)    ++low;
      L.r[high] = L.r[low];                     // 将比枢轴记录大的记录移到高端
    } // while
    L.r[low] = L.r[0];                          // 枢轴记录移到正确位置
    return low;                                 // 返回枢轴位置
} // Partition
```

快速排序

- 算法10.8

```
void QSort (SqList &L, int low, int high )
{   // 对顺序表L中 的子序列L.r[low…high] 进行快速排序
    if (low < high) {                    // 长度大于1
       pivotloc = Partition(L, low, high);   // 将L.r[low…high]一分为二
       QSort(L, low, pivotloc-1);    // 对低子序列递归进行排序，pivotloc是枢轴位置
       QSort(L, pivotloc+1, high);   // 对高子序列递归进行排序
    } // if
} // Qsort
```

- 算法10.9

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

快速排序

- 为避免出现枢轴记录关键字为“最大”或“最小”的情况，通常进行的快速排序采用“三者取中”的改进方案，即以**L.r[low]**、**L.r[high]**和 **L.r[(low+high)/2]** 三者中关键字介于中值者为枢轴。只要将它和 L.r[low] 互换，一次划分的算法仍不变。
- 可以推证，快速排序的平均时间复杂度为***O*(nlogn)**，在三者取中的前提下，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，如果借用起泡排序中设置记录“交换与否”的布尔变量的作法，快速排序也适用于已经有序的记录序列。

本章小结

- 本章主要讨论各种内部排序的方法。学习本章的目的是了解各种排序方法的原理以及各自的优缺点，以便在编制软件时能按照情况所需合理选用。
- 一般来说，在选择排序方法时，可有下列几种选择：
  - 若待排序的记录个数n值较小（例如n<30），则可选用插入排序法，但若记录所含数据项较多，所占存储量大时，应选用选择排序法。
  - 反之，若待排序的记录个数n值较大时，应选用快速排序法。但若待排序记录关键字有“有序”倾向时，就慎用快速排序，而宁可选用堆排序或归并排序，而后两者的最大差别是所需辅助空间不等。

本章小结

- 快速排序和归并排序在n值较小时的性能不及直接插入排序，因此在实际应用时，可将它们和插入排序“混合”使用。如在快速排序划分子区间的长度小于某值时，转而调用直接插入排序；或者对待排记录序列先逐段进行直接插入排序，然后再利用“归并操作”进行两两归并直至整个序列有序为止。

本章知识点与重点

- **知识点**

排序、直接插入排序、折半插入排序、表插入排序、希尔排序、起泡排序、快速排序

- **重点和难点**

希尔排序、快速排序等高效方法是本章的学习重点和难点

练习题

1. 请总结排序算法特点和时间复杂度。

插入排序（直接插入排序，折半插入排序，2-路插入排序，表插入排序，希尔排序）；快速排序（起泡排序，改进起泡排序，快速排序）。

2. 若对序列（72，35，12，18，26，29，41，15）按从小到大排序，请写出：（过程：即每趟排序结果）

① 2-路插入排序、表插入排序和希尔排序过程；

② 改进起泡排序的第一趟结果；

③ 堆排序过程和结果。