第10章　内部排序

在信息处理过程中，最基本的操作是查找。从查找来说，效率最高的是折半查找，折半查找的前提是所有的数据元素(记录)是按关键字有序的。需要将一个无序的数据文件转变为一个有序的数据文件。

将任一文件中的记录通过某种方法整理成为按(记录)关键字有序排列的处理过程称为排序。

排序是数据处理中一种最常用的操作。

10.1 排序的基本概念

(1) 排序(Sorting)

排序是将一批(组)任意次序的记录重新排列成按关键字有序的记录序列的过程，其定义为：

给定一组记录序列：$\{R_1, R_2, \ldots, R_n\}$，其相应的关键字序列是$\{K_1, K_2, \ldots, K_n\}$。确定1, 2, … n的一个排列$p_1, p_2, \ldots, p_n$，使其相应的关键字满足如下非递减(或非递增)关系：$K_{p_1} \le K_{p_2} \le \ldots \le K_{p_n}$的序列$\{K_{p_1}, K_{p_2}, \ldots, K_{p_n}\}$，这种操作称为排序。

关键字K_i可以是记录R_i的主关键字，也可以是次关键字或若干数据项的组合。

◆ K_i是主关键字：排序后得到的结果是唯一的；

◆ K_i是次关键字：排序后得到的结果是不唯一的。

(2) 排序的稳定性

若记录序列中有两个或两个以上关键字相等的记录：$K_i = K_j (i \neq j,\ i, j=1, 2, \ldots n)$，且在排序前$R_i$先于$R_j (i<j)$，排序后的记录序列仍然是$R_i$先于$R_j$，称排序方法是稳定的，否则是不稳定的。

排序算法有许多，但就全面性能而言，还没有一种公认为最好的。每种算法都有其优点和缺点，分别适合不同的数据量和硬件配置。

评价排序算法的标准有：执行时间和所需的辅助空间，其次是算法的稳定性。

若排序算法所需的辅助空间不依赖问题的规模**n**，即空间复杂度是**O(1)**，则称排序方法是就地排序，否则是非就地排序。

(3) 排序的分类

待排序的记录数量不同，排序过程中涉及的存储器的不同，有不同的排序分类。

① 待排序的记录数不太多：所有的记录都能存放在内存中进行排序，称为内部排序；

② 待排序的记录数太多：所有的记录不可能存放在内存中，排序过程中必须在内、外存之间进行数据交换，这样的排序称为外部排序。

(4) 内部排序的基本操作

对内部排序地而言，其基本操作有两种：

◆ 比较两个关键字的大小；

◆ 存储位置的移动：从一个位置移到另一个位置。

第一种操作是必不可少的；而第二种操作却不是必须的，取决于记录的存储方式，具体情况是：

① 记录存储在一组连续地址的存储空间：记录之间的逻辑顺序关系是通过其物理存储位置的相邻来体现，记录的移动是必不可少的；

② 记录采用链式存储方式：记录之间的逻辑顺序关系是通过结点中的指针来体现，排序过程仅需修改结点的指针，而不需要移动记录；

③ 记录存储在一组连续地址的存储空间：构造另一个辅助表来保存各个记录的存放地址(指针)：排序过程不需要移动记录，而仅需修改辅助表中的指针，排序后视具体情况决定是否调整记录的存储位置。

①比较适合记录数较少的情况；而②、③则适合记录数较少的情况。

为讨论方便，假设待排序的记录是以①的情况存储，且设排序是按升序排列的；关键字是一些可直接用比较运算符进行比较的类型。

待排序的记录类型的定义如下：

```
#define  MAX_SIZE  100
Typedef  int  KeyType ;
typedef  struct  RecType
    { KeyType  key ;           /* 关键字码  */
        infoType  otherinfo ;     /* 其他域  */
    }RecType ;
typedef  struct Sqlist
    {  RecType  R[MAX_SIZE] ;
        int length   ;
    }Sqlist ;
```

10.2 插入排序

采用的是以“玩桥牌者”的方法为基础的。即在考察记录R_i之前，设以前的所有记录$R_1, R_2, ..., R_{i-1}$已排好序，然后将R_i插入到已排好序的诸记录的适当位置。

最基本的插入排序是直接插入排序(**Straight Insertion Sort**)。

10.2.1 直接插入排序

1 排序思想

将待排序的记录R_i，插入到已排好序的记录表R_1, R_2 ,…., R_{i-1}中，得到一个新的、记录数增加1的有序表。直到所有的记录都插入完为止。

设待排序的记录顺序存放在数组R[1…n]中，在排序的某一时刻，将记录序列分成两部分：

- R[1…i-1]：已排好序的有序部分；
- R[i…n]：未排好序的无序部分。

显然，在刚开始排序时，R[1]是已经排好序的。

例：设有关键字序列为：7, 4, -2, 19, 13, 6，直接插入排序的过程如下图10-1所示：

初始记录的关键字：[7] 4 -2 19 13 6

第一趟排序：[4 7] -2 19 13 6

第二趟排序：[-2 4 7] 19 13 6

第三趟排序：[-2 4 7 19] 13 6

第四趟排序：[-2 4 7 13 19] 6

第五趟排序：[-2 4 6 7 13 19]

图10-1 直接插入排序的过程

2 算法实现

```
void straight_insert_sort(Sqlist *L)
  {  int i, j ;
     for (i=2; i<=L->length; i++)
       { L->R[0]=L->R[i]; j=i-1;     /*  设置哨兵  */
         while( LT(L->R[0].key, L->R[j].key) )
           {  L->R[j+1]=L->R[j];
              j--;
           }        /*  查找插入位置  */
         L->R[j+1]=L->R[0];     /*  插入到相应位置  */
       }
  }
```

3 算法说明

算法中的**R[0]**开始时并不存放任何待排序的记录，引入的作用主要有两个：

① 不需要增加辅助空间：保存当前待插入的记录**R[i]**，**R[i]**会因为记录的后移而被占用；

② 保证查找插入位置的内循环总可以在超出循环边界之前找到一个等于当前记录的记录，起“哨兵监视”作用，避免在内循环中每次都要判断**j**是否越界。

4 算法分析

(1) 最好情况：若待排序记录按关键字从小到大排列(正序)，算法中的内循环无须执行，则一趟排序时：关键字比较次数1次，记录移动次数2次(R[i]→R[0], R[0]→R[j+1])。

则整个排序的关键字比较次数和记录移动次数分别是：

比较次数：$\sum_{i=2}^{n} 1 = n-1$　　　移动次数：$\sum_{i=2}^{n} 2 = 2(n-1)$

(2) **最坏情况**：若待排序记录按关键字从大到小排列(逆序)，则一趟排序时：算法中的内循环体执行i-1，关键字比较次数i次，记录移动次数i+1。

则就整个排序而言：

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

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

一般地，认为待排序的记录可能出现的各种排列的概率相同，则取以上两种情况的平均值，作为排序的关键字比较次数和记录移动次数，约为$n^2/4$，则复杂度为$O(n^2)$ 。

10.2.2 其它插入排序

1 折半插入排序

当将待排序的记录R[i] 插入到已排好序的记录子表R[1…i-1]中时，由于R_1, R_2 ,…, R_{i-1}已排好序，则查找插入位置可以用“折半查找”实现，则直接插入排序就变成为折半插入排序。

(1) 算法实现

```
void Binary_insert_sort(Sqlist *L)
  { int i, j, low, high, mid ;
     for (i=2; i<=L->length; i++)
       { L->R[0]=L->R[i];      /*  设置哨兵  */
```

```
        low=1 ; high=i-1 ;
        while (low<=high)
          {  if ( LT(L->R[0].key, L->R[mid].key) )
                 high=mid-1 ;
             else   low=mid+1 ;
          }        /*  查找插入位置  */
        for (j=i-1; j>=high+1; j--)
        L->R[j+1]=L->R[j];
        L->R[high+1]=L->R[0]; /*  插入到相应位置  */
    }
}
```

从时间上比较，折半插入排序仅仅减少了关键字的比较次数，却没有减少记录的移动次数，故时间复杂度仍然为$O(n^2)$。

(2) 排序示例

设有一组关键字**30, 13, 70, 85, 39, 42, 6, 20**，采用折半插入排序方法排序的过程如图**10-2**所示：

i=1 (30) 13 70 85 39 42 6 20

i=2 13 (13 30) 70 85 39 42 6 20

⋮

i=7 6 (6 13 30 39 42 70 85) 20

i=8 20 (6 13 30 39 42 70 85) 20

low mid high

i=8 20 (6 13 30 39 42 70 85) 20

low mid high

i=8 20 (6 13 30 39 42 70 85) 20

low mid high

i=8 20 (6 13 20 30 39 42 70 85)

图10-2 折半插入排序过程

2 2-路插入排序

是对折半插入排序的改进，以减少排序过程中移动记录的次数。附加n个记录的辅助空间，方法是：

① 另设一个和L->R同类型的数组d，L->R[1]赋给d[1]，将d[1]看成是排好序的序列中中间位置的记录；

② 分别将L->R[ ]中的第i个记录依次插入到d[1]之前或之后的有序序列中，具体方法：

- ◆ L->R[i].key<d[1].key： L->R[i]插入到d[1]之前的有序表中；
- ◆ L->R[i].key≥d[1].key： L->R[i]插入到d[1]之后的有序表中；

关键点：实现时将向量d看成是循环向量，并设两个指针first和final分别指示排序过程中得到的有序序列中的第一个和最后一个记录。

排序示例

设有初始关键字集合{49, 38, 65, 13, 97, 27, 76}，采用2-路插入排序的过程如右图10-3所示。

在2-路插入排序中，移动记录的次数约为n2/8 。但当L->R[1]是待排序记录中关键字最大或最小的记录时，2-路插入排序就完全失去了优越性。

图10-3　2-路插入排序过程

3 表插入排序

前面的插入排序不可避免地要移动记录，若不移动记录就需要改变数据结构。附加n个记录的辅助空间，记录类型修改为:

```
typedef struct  RecNode
   {  KeyType  key ;
      infotype  otherinfo ;
      int *next;
   }RecNode ;
```

初始化：下标值为0的分量作为表头结点，关键字取为最大值，各分量的指针值为空；

① 将静态链表中数组下标值为1的分量(结点)与表头结点构成一个循环链表；

② i=2 ，将分量R[i]按关键字递减插入到循环链表；

③ 增加i ，重复②，直到全部分量插入到循环链表。

例：设有关键字集合{49, 38, 65, 97, 76, 13, 27, 49}，采用表插入排序的过程如下图10-4所示。

	0	1	2	3	4	5	6	7	8	
	MAXINT	49	38	65	13	97	27	76	49	key域
	1	0	-	-	-	-	-	-	-	next域
i=2	MAXINT	49	38	65	13	97	27	76	49	
	2	0	1	-	-	-	-	-	-	
i=3	MAXINT	49	38	65	13	97	27	76	49	
	2	3	1	0	-	-	-	-	-	
i=4	MAXINT	49	38	65	13	97	27	76	49	
	4	3	1	0	2	-	-	-	-	
i=5	MAXINT	49	38	65	13	97	27	76	49	
	4	3	1	5	2	0	-	-	-	

i=6	MAXINT 4	49 3	38 1	65 5	13 6	97 0	27 2	76 -	49 -
i=7	MAXINT 4	49 3	38 1	65 7	13 6	97 0	27 2	76 5	49 -
i=8	MAXINT 4	49 8	38 1	65 7	13 6	97 0	27 2	76 5	49 3

图10-4　表插入排序过程

和直接插入排序相比，不同的是修改2n次指针值以代替移动记录，而关键字的比较次数相同，故时间复杂度为$O(n^2)$。

表插入排序得到一个有序链表，对其可以方便地进行顺序查找，但不能实现随即查找。根据需要，可以对记录进行重排，记录重排详见P_{268}。

10.2.3 希尔排序

希尔排序(Shell Sort，又称缩小增量法)是一种分组插入排序方法。

1 排序思想

① 先取一个正整数$d_1(d_1<n)$作为第一个增量，将全部n个记录分成d_1组，把所有相隔d_1的记录放在一组中，即对于每个k(k=1, 2, … d_1)，R[k], R[d_1+k], R[2d_1+k] , …分在同一组中，在各组内进行直接插入排序。这样一次分组和排序过程称为一趟希尔排序；

② 取新的增量$d_2<d_1$，重复①的分组和排序操作；直至所取的增量$d_i=1$为止，即所有记录放进一个组中排序为止。

2 排序示例

设有10个待排序的记录，关键字分别为9, 13, 8, 2, 5, 13, 7, 1, 15, 11，增量序列是5, 3, 1，希尔排序的过程如图10-5所示。

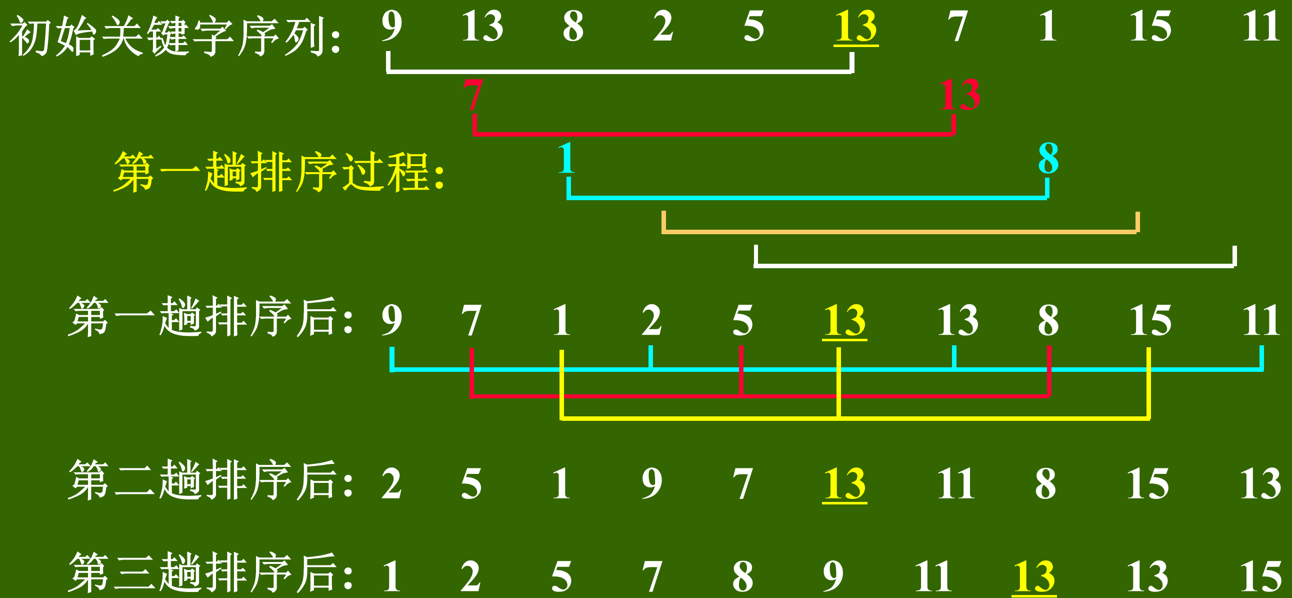

图10-5 希尔排序过程

3 算法实现

先给出一趟希尔排序的算法，类似直接插入排序。

```
void shell_pass(Sqlist *L, int d)
    /*  对顺序表L进行一趟希尔排序, 增量为d  */
  {  int j, k ;
     for (j=d+1; j<=L->length; j++)
        {  L->R[0]=L->R[j] ;        /*  设置监视哨兵  */
           k=j-d ;
           while (k>0&&LT(L->R[0].key, L->R[k].key) )
              {   L->R[k+d]=L->R[k] ; k=k-d ;   }
           L->R[k+j]=L->R[0] ;
        }
  }
```

然后在根据增量数组dk进行希尔排序。

```
void shell_sort(Sqlist *L, int dk[], int t)
    /*  按增量序列dk[0 … t-1],对顺序表L进行希尔排序  */
    { int m ;
       for (m=0; m<=t; m++)
           shll_pass(L, dk[m]) ;
    }
```

希尔排序的分析比较复杂，涉及一些数学上的问题，其时间是所取的“增量”序列的函数。

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，原因是：

- 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。

10.3 快速排序

是一类基于交换的排序，系统地交换反序的记录的偶对，直到不再有这样一来的偶对为止。其中最基本的是冒泡排序(**Bubble Sort**)。

10.3.1 冒泡排序

1 排序思想

依次比较相邻的两个记录的关键字，若两个记录是反序的(即前一个记录的关键字大于后前一个记录的关键字)，则进行交换，直到没有反序的记录为止。

① 首先将L->R[1]与L->R[2]的关键字进行比较，若为反序(L->R[1]的关键字大于L->R[2]的关键字)，则交换两个记录；然后比较L->R[2]与L->R[3]的关键字，依此类推，直到L->R[n-1]与L->R[n]的关键字比较后为止，称为一趟冒泡排序，L->R[n]为关键字最大的记录。

② 然后进行第二趟冒泡排序，对前n-1个记录进行同样的操作。

一般地，第i趟冒泡排序是对L->R[1 … n-i+1]中的记录进行的，因此，若待排序的记录有n个，则要经过n-1趟冒泡排序才能使所有的记录有序。

2 排序示例

设有9个待排序的记录，关键字分别为23, 38, 22, 45, 23, 67, 31, 15, 41， 冒泡排序的过程如图10-6所示。

3 算法实现

```
#define FALSE 0
#define TRUE 1
```

初始关键字序列:	23	38	22	45	23	67	31	15	41
第一趟排序后:	23	22	38	23	45	31	15	41	67
第二趟排序后:	22	23	23	38	31	15	41	45	67
第三趟排序后:	22	23	23	31	15	38	41	45	67
第四趟排序后:	22	23	23	15	31	38	41	45	67
第五趟排序后:	22	23	15	23	31	38	41	45	67
第六趟排序后:	22	15	23	23	31	38	41	45	67
第七趟排序后:	15	22	23	23	31	38	41	45	67

图10-6 冒泡排序过程

```
void Bubble_Sort(Sqlist *L)
  {  int j ,k , flag ;
     for (j=0; j<L->length; j++)       /*   共有n-1趟排序   */
        {  flag=TRUE ;
           for (k=1; k<=L->length-j; k++)   /*   一趟排序   */
              if (LT(L->R[k+1].key, L->R[k].key ) )
                 {   flag=FALSE ; L->R[0]=L->R[k] ;
                     L->R[k]=L->R[k+1] ;
                     L->R[k+1]=L->R[0] ;
                  }
              if  (flag==TRUE)  break ;
        }
  }
```

4 算法分析

时间复杂度

- 最好情况(正序)：比较次数：n-1；移动次数：0；
- 最坏情况(逆序)：

比较次数：$\sum_{i=1}^{n-1}(n-i)=\frac{n(n-1)}{2}$

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

故时间复杂度：$T(n)=O(n^2)$

空间复杂度：$S(n)=O(1)$

10.3.2 快速排序

1 排序思想

通过一趟排序，将待排序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，再分别对这两部分记录进行下一趟排序，以达到整个序列有序。

2 排序过程

设待排序的记录序列是**R[s…t]**，在记录序列中任取一个记录(一般取**R[s]**)作为参照(又称为基准或枢轴)，以**R[s].key**为基准重新排列其余的所有记录，方法是：

◆ 所有关键字比基准小的放R[s]之前；

◆ 所有关键字比基准大的放R[s]之后。

以R[s].key最后所在位置i作为分界，将序列R[s…t]分割成两个子序列，称为一趟快速排序。

3 一趟快速排序方法

从序列的两端交替扫描各个记录，将关键字小于基准关键字的记录依次放置到序列的前边；而将关键字大于基准关键字的记录从序列的最后端起，依次放置到序列的后边，直到扫描完所有的记录。

设置指针low，high，初值为第1个和最后一个记录的位置。

设两个变量i，j，初始时令i=low，j=high，以R[low].key作为基准(将R[low]保存在R[0]中) 。

① 从j所指位置向前搜索：将R[0].key与R[j].key进行比较：

- ◆ 若R[0].key≤R[j].key ：令j=j-1，然后继续进行比较，直到i=j或R[0].key>R[j].key为止；
- ◆ 若R[0].key>R[j].key ：R[j]⇒R[i]，腾空R[j]的位置，且令i=i+1；

② 从i所指位置起向后搜索：将R[0].key与R[i].key进行比较：

- ◆ 若R[0].key≥R[i].key ：令i=i+1，然后继续进行比较，直到i=j或R[0].key<R[i].key为止；

◆ 若R[0].key<R[i].key： R[i]⇒R[j]，腾空R[i]的位置， 且令j=j-1；

③重复①、②，直至i=j为止，i就是R[0](基准)所应放置的位置。

4 一趟排序示例

设有6个待排序的记录，关键字分别为29, 38, 22, 45, 23, 67，一趟快速排序的过程如图10-7所示。

5 算法实现

(1) 一趟快速排序算法的实现

```
int  quick_one_pass(Sqlist  *L , int low, int high)
    {  int i=low, j=high ;
```

	0							
初始关键字序列:	29	29	38	22	45	23	67	31
		i						j
j前移2个位置后，R[j]放R[i]的位置:	29	23	38	22	45		67	31
		i				j		
i后移1个位置后，R[i]放R[j]的位置:	29	23		22	45	38	67	31
			i			j		
j前移2个位置后，R[j]放R[i]的位置:	29	23	22		45	38	67	31
			i	j				
i后前移1个位置后，i和j的位置重合:	29	23	22	29	45	38	67	31
				i j				

图10-7　一趟快速排序过程

```
    L->R[0]=L->R[i] ;       /*  R[0]作为临时单元和哨兵 */
    do
       {   while (LQ(L->R[0].key, L->R[j].key)&&(j>i))
                 j-- ;
           if  (j>i)  {  L->R[i]=L->R[j] ; i++;   }
           while (LQ(L->R[i].key, L->R[0].key)&&(j>i))
                 i++ ;
           if  (j>i)  {  L->R[j]=L->R[i] ; j--;   }
       } while(i!=j) ;    /*  i=j时退出扫描 */
    L->R[i]=L->R[0] ;
    return(i) ;
}
```

(2) 快速排序算法实现

当进行一趟快速排序后，采用同样方法分别对两个子序列快速排序，直到子序列记录个为1为止。

① 递归算法

```
void  quick_Sort(Sqlist  *L , int low, int high)
  {  int k ;
     if  (low<high)
        {  k=quick_one_pass(L, low, high);
           quick_Sort(L, low, k-1);
           quick_Sort(L, k+1, high);
        }    /*  序列分为两部分后分别对每个子序列排序  */
  }
```

② 非递归算法

```
# define  MAX_STACK  100
void  quick_Sort(Sqlist  *L , int low, int high)
   {  int k , stack[MAX_STACK] ,  top=0;
      do {  while  (low<high)
              {  k=quick_one_pass(L,low,high);
                 stack[++top]=high ;  stack[++top]=k+1 ;
                     /*  第二个子序列的上,下界分别入栈  */
                 high=k-1 ;
              }
              if (top!=0)
                {  low=stack[top--] ; high=stack[top--] ;  }
```

```
        }while (top!=0&&low<high) ;
    }
```

6 算法分析

快速排序的主要时间是花费在划分上，对长度为k的记录序列进行划分时关键字的比较次数是k-1 。设长度为n的记录序列进行排序的比较次数为C(n)，则$C(n)=n-1+C(k)+C(n-k-1)$ 。

◆ **最好情况**：每次划分得到的子序列大致相等，则

$C(n) \le n+2 \times C(n/2)+C(n-k-1)$

$\le n+2 \times [n/2+ 2 \times C((n/2)/2) \le 2n+4 \times C(n/4)$

$\le \ldots$

$\le h \times n+2^h \times C(n/2^h)$ ，当$n/2^h=1$时排序结束。

即$C(n) \le n \times \log_2 n + n \times C(1)$，$C(1)$看成常数因子，

即$C(n) \le O(n \times \log_2 n)$；

◆ 最坏情况：每次划分得到的子序列中有一个为空，另一个子序列的长度为n-1。即每次划分所选择的基准是当前待排序序列中的最小(或最大)关键字。

比较次数：$\sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2}$　　即$C(n)=O(n^2)$

◆ 一般情况：对n个记录进行快速排序所需的时间T(n)组成是：

① 对n个记录进行一趟划分所需的时间是：$n \times C$，C是常数；

② 对所得到的两个子序列进行快速排序的时间：

$T_{avg}(n)=C(n)+T_{avg}(k-1)+T_{avg}(n-k)$　　…… (1)

若记录是随机排列的，k取值在1~n之间的概率相同，则：

$$T_{avg}(n)=n\times C+\frac{1}{n}\sum_{k=0}^{n}[T_{avg}(k-1)+T_{avg}(n-k)]$$

$$=n\times C+\frac{2}{n}\sum_{k=0}^{n-1}T_{avg}(k) \quad \cdots\cdots (2)$$

当n>1时，用n-1代替(2)中的n，得到：

$$T_{avg}(n)=(n-1)\times C+\frac{2}{n}\sum_{k=0}^{n-2}T_{avg}(k) \quad \cdots\cdots (3)$$

$\therefore$ $nT_{avg}(n)-(n-1)T_{avg}(n-1)=(2n-1)\times C+2T_{avg}(n-1)$，即

$$T_{avg}(n)=(n+1)/n\times T_{avg}(n-1)+(2n-1)/n\times C$$

$$<(n+1)/n\times T_{avg}(n-1)+2C$$

$$<(n+1)/n\times [n/(n-1)\times T_{avg}(n-2)+2C]+2C$$

$=(n+1)/(n-1)\times T_{avg}(n-2)+2(n+1)[1/n+1/(n+1)]\times C$

$< \ldots$

$\therefore\ T_{avg}(n) < \frac{n+1}{2}T_{avg}(1)+2(n+1)\times C[\frac{1}{3}+\frac{1}{4}+\ldots+\frac{1}{n}+\frac{1}{n+1}]$

只有1个记录的排序时间是一个常数，

∴ 快速排序的平均时间复杂度是：$T(n)=O(n\log_2 n)$

从所需要的附加空间来看，快速排序算法是递归调用，系统内用堆栈保存递归参数，当每次划分比较均匀时，栈的最大深度为[log2n]+1 。

∴ 快速排序的空间复杂度是：$S(n)=O(\log 2n)$

从排序的稳定性来看，快速排序是不稳定的。

10.4　选择排序

选择排序(**Selection Sort**)的基本思想是：每次从当前待排序的记录中选取关键字最小的记录表，然后与待排序的记录序列中的第一个记录进行交换，直到整个记录序列有序为止。

10.4.1 简单选择排序

简单选择排序(Simple Selection Sort ， 又称为直接选择排序)的基本操作是：通过n-i次关键字间的比较，从n-i+1个记录中选取关键字最小的记录，然后和第i个记录进行交换，i=1, 2, … n-1 。

1 排序示例

例：设有关键字序列为：7, 4, -2, 19, 13, 6，直接选择排序的过程如下图10-8所示。

初始记录的关键字： 7 4 -2 19 13 6

第一趟排序：-2 4 7 19 13 6

第二趟排序： -2 4 7 19 13 6

第三趟排序： -2 4 6 19 13 7

第四趟排序： -2 4 6 7 13 19

第五趟排序： -2 4 6 7 13 19

第六趟排序： -2 4 6 7 13 19

图10-8 直接选择排序的过程

2 算法实现

```
void simple_selection_sort(Sqlist *L)
  {  int m, n , k;
     for (m=1; m<L->length; m++)
       {  k=m ;
          for  (n=m+1; n<=L->length; n++)
             if  ( LT(L->R[n].key, L->R[k].key) )  k=n ;
          if  (k!=m)      /*   记录交换   */
             {  L->R[0]=L->R[m]; L->R[m]=L->R[k];
                L->R[k]=L->R[0];
             }
       }
  }
```

3 算法分析

整个算法是二重循环：外循环控制排序的趟数，对n个记录进行排序的趟数为n-1趟；内循环控制每一趟的排序。

进行第i趟排序时，关键字的比较次数为n-i，则：

比较次数：$\sum_{i=1}^{n-1}(n-i)=\frac{n(n-1)}{2}$

∴ 时间复杂度是：$T(n)=O(n^2)$

空间复杂度是：$S(n)=O(1)$

从排序的稳定性来看，直接选择排序是不稳定的。

10.4.2 树形选择排序

借助“淘汰赛”中的对垒就很容易理解树形选择排序的思想。

首先对n个记录的关键字两两进行比较，选取⌈n/2⌉个较小者；然后这⌈n/2⌉个较小者两两进行比较，选取⌈n/4⌉个较小者… 如此重复，直到只剩1个关键字为止。

该过程可用一棵有n个叶子结点的完全二叉树表示，如图10-9所示。

每个枝结点的关键字都等于其左、右孩子结点中较小的关键字，根结点的关键字就是最小的关键字。

输出最小关键字后，根据关系的可传递性，欲选取次小关键字，只需将叶子结点中的最小关键字改为“最大值”，然后重复上述步骤即可。

含有n个叶子结点的完全二叉树的深度为$\lceil \log_2 n \rceil+1$，则总的时间复杂度为$O(n\log_2 n)$。

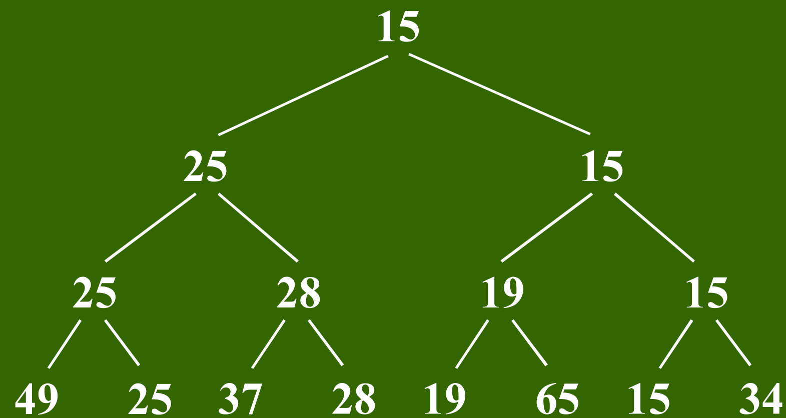

图10- 9 “淘汰赛”过程示意图

10.4.3 堆排序

1 堆的定义

是n个元素的序列H={k_1, k_2 , … k_n}，满足：

$k_i \le k_{2i}$ 当$2i \le n$时　　或　　$k_i \ge k_{2i}$ 当$2i \le n$时

$k_i \le k_{2i+1}$ 当$2i+1 \le n$时　　　　$k_i \ge k_{2i+1}$ 当$2i+1 \le n$时

其中：$i=1,2, \ldots, \lfloor n/2 \rfloor$

由堆的定义知，堆是一棵以k_1为根的完全二叉树。若对该二叉树的结点进行编号(从上到下，从左到右)，得到的序列就是将二叉树的结点以顺序结构存放，堆的结构正好和该序列结构完全一致。

2 堆的性质

① 堆是一棵采用顺序存储结构的完全二叉树，k_1是根结点；

② 堆的根结点是关键字序列中的最小(或最大)值，分别称为小(或大)根堆；

③ 从根结点到每一叶子结点路径上的元素组成的序列都是按元素值(或关键字值)非递减(或非递增)的；

④堆中的任一子树也是堆。

利用堆顶记录的关键字值最小(或最大)的性质，从当前待排序的记录中依次选取关键字最小(或最大)的记录，就可以实现对数据记录的排序，这种排序方法称为堆排序。

3 堆排序思想

① 对一组待排序的记录，按堆的定义建立堆；

② 将堆顶记录和最后一个记录交换位置，则前n-1个记录是无序的，而最后一个记录是有序的；

③ 堆顶记录被交换后，前n-1个记录不再是堆，需将前n-1个待排序记录重新组织成为一个堆，然后将堆顶记录和倒数第二个记录交换位置，即将整个序列中次小关键字值的记录调整(排除)出无序区；

④ 重复上述步骤，直到全部记录排好序为止。

结论：排序过程中，若采用小根堆，排序后得到的是非递减序列；若采用大根堆，排序后得到的是非递增序列。

堆排序的关键

① 如何由一个无序序列建成一个堆？

② 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？

4 堆的调整——筛选

(1) 堆的调整思想

输出堆顶元素之后，以堆中最后一个元素替代之；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，直到是叶子结点或其关键字值小于等于左、右子树的关键字的值，将得到新的堆。称这个从堆顶至叶子的调整过程为“筛选”，如图**10-10**所示。

注意：筛选过程中，根结点的左、右子树都是堆，因此，筛选是从根结点到某个叶子结点的一次调整过程。

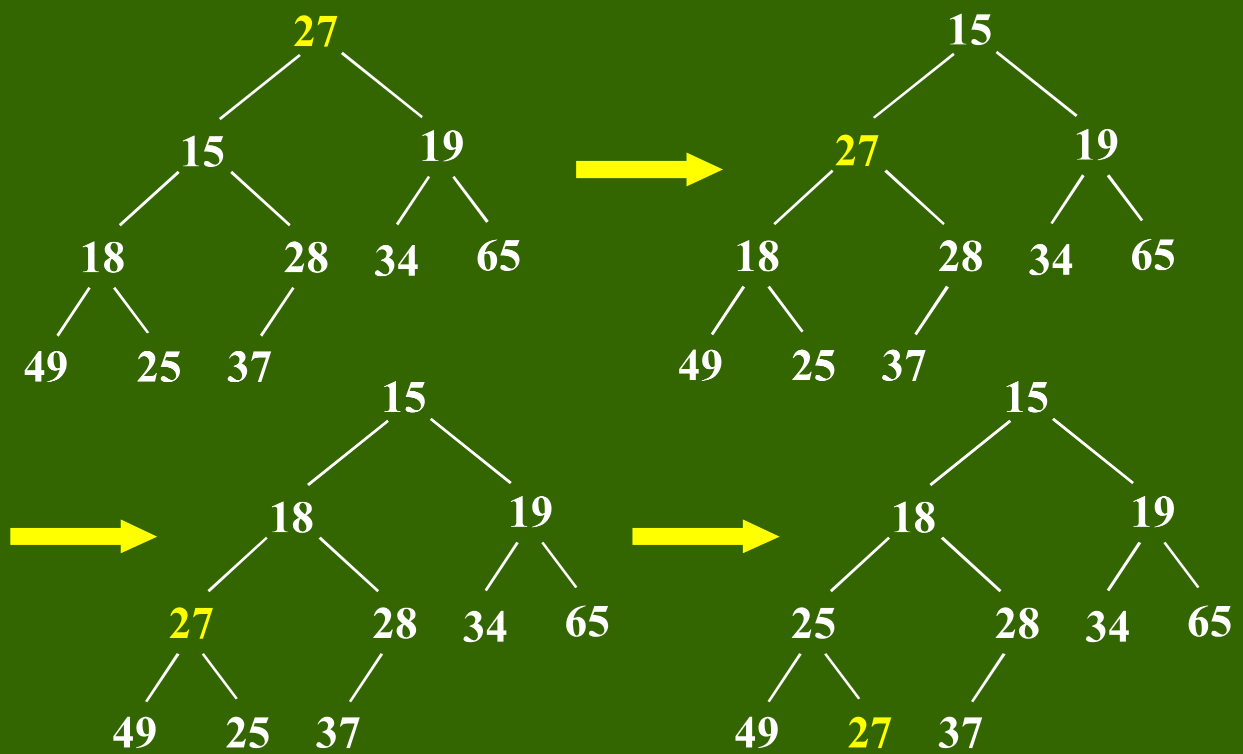

图10-10 堆的筛选过程

(2) 堆调整算法实现

```
void Heap_adjust(Sqlist *H, int s, int m)
   /*  H->R[s…m]中记录关键字除H->R[s].key均满足堆定义  */
   /*  调整H->R[s]的位置使之成为小根堆  */
   {  int j=s, k=2*j ;      /*  计算H->R[j]的左孩子的位置  */
      H->R[0]=H->R[j] ;        /*  临时保存H->R[j]  */
      for (k=2*j; k<=m; k=2*k)
         {  if ((k<m)&&(LT(H->R[k+1].key, H->R[k].key))
                k++ ;      /*  选择左、右孩子中关键字的最小者  */
            if  ( LT(H->R[k].key, H->R[0].key) )
                {   H->R[j]=H->R[k] ; j=k ; k=2*j  }
            else  break ;
         }
```

```
    H->R[j]=H->R[0] ;
}
```

5 堆的建立

利用筛选算法，可以将任意无序的记录序列建成一个堆，设R[1],R[2], …,R[n]是待排序的记录序列。

将二叉树的每棵子树都筛选成为堆。只有根结点的树是堆。第$\lfloor n/2 \rfloor$个结点之后的所有结点都没有子树，即以第$\lfloor n/2 \rfloor$个结点之后的结点为根的子树都是堆。因此，以这些结点为左、右孩子的结点，其左、右子树都是堆，则进行一次筛选就可以成为堆。同理，只要将这些结点的直接父结点进行一次筛选就可以成为堆…。

只需从第$\lfloor n/2 \rfloor$个记录到第1个记录依次进行筛选就可以建立堆。

可用下列语句实现：

```
for (j=n/2; j>=1; j--)
    Heap_adjust(R, j , n) ;
```

6 堆排序算法实现

堆的根结点是关键字最小的记录，输出根结点后，是以序列的最后一个记录作为根结点，而原来堆的左、右子树都是堆，则进行一次筛选就可以成为堆。

```
void  Heap_Sort(Sqlist *H)
  {  int j ;
     for (j=H->length/2; j>0; j--)
         Heap_adjust(H, j , H->length) ;  /*  初始建堆  */
```

```
    for (j=H->length/2; j>=1; j--)
        {  H->R[0]=H->R[1] ; H->R[1]=H->R[j] ;
           H->R[j]=H->R[0] ;  /*  堆顶与最后一个交换  */
           Heap_adjust(H, 1, j-1) ;
        }
}
```

7 算法分析

主要过程：初始建堆和重新调整成堆。设记录数为n，所对应的完全二叉树深度为h 。

◆ 初始建堆：每个非叶子结点都要从上到下做“筛选”。第i层结点数≤2i-1，结点下移的最大深度是h-i，而每下移一层要比较2次，则比较次数C1(n)为：

$$C_1(n) \le 2\sum_{i=1}^{h-1}(2^{i-1}\times(h-i)) \le 4(2^h-h-1)$$

∵ $h=\lfloor \log_2 n \rfloor+1$， ∴ $C_1(n) \le 4(n-\log_2 n-1)$

◆ 筛选调整：每次筛选要将根结点“下沉”到一个合适位置。第i次筛选时：堆中元素个数为n-i+1；堆的深度是$\lfloor \log_2(n-i+1) \rfloor+1$，则进行n-1次“筛选”的比较次数$C_2(n)$为：

$$C_2(n) \le \sum_{i=1}^{n-1}(2\times \log_2(n-i+1))$$

∴ $C_2(n) < 2n\log_2 n$

∴ 堆排序的比较次数的数量级为： $T(n)=O(n\log_2 n)$；而附加空间就是交换时所用的临时空间，故空间复杂度为： $S(n)=O(1)$ 。

10.5 归并排序

归并(Merging)：是指将两个或两个以上的有序序列合并成一个有序序列。若采用线性表(无论是那种存储结构)易于实现，其时间复杂度为O(m+n)。

归并思想实例：两堆扑克牌，都已从小到大排好序，要将两堆合并为一堆且要求从小到大排序。

- 将两堆最上面的抽出(设为C_1，C_2)比较大小，将小者置于一边作为新的一堆(不妨设$C_1 < C_2$)；再从第一堆中抽出一张继续与C_2进行比较，将较小的放置在新堆的最下面；
- 重复上述过程，直到某一堆已抽完，然后将剩下一堆中的所有牌转移到新堆中。

1 排序思想

① 初始时，将每个记录看成一个单独的有序序列，则n个待排序记录就是n个长度为1的有序子序列；

② 对所有有序子序列进行两两归并，得到⌈n/2⌉个长度为2或1的有序子序列——一趟归并；

③ 重复②，直到得到长度为n的有序序列为止。

上述排序过程中，子序列总是两两归并，称为**2-路归并排序**。其核心是如何将相邻的两个子序列归并成一个子序列。设相邻的两个子序列分别为：

{R[k], R[k+1], …, R[m]}和**{R[m+1], R[m+2],…, R[h]}**，将它们归并为一个有序的子序列：

{DR[*l*], DR[*l*+1], …, DR[m], DR[m+1], …, DR[h] }

例：设有9个待排序的记录，关键字分别为23, 38, 22, 45, 23, 67, 31, 15, 41，归并排序的过程如图10-11所示。

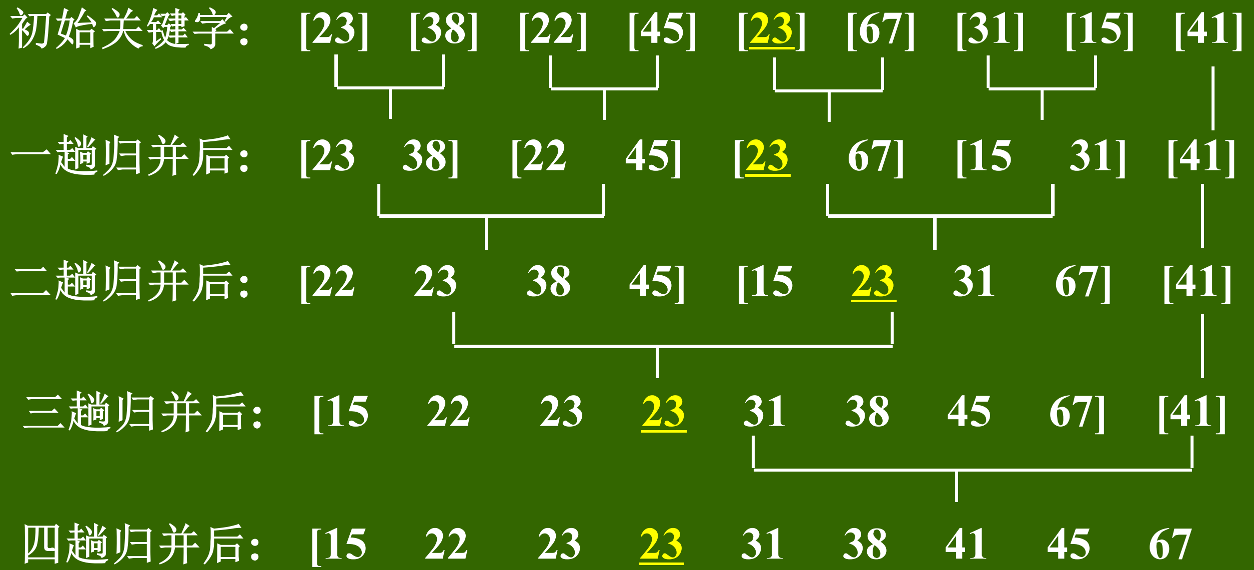

图10-11 归并排序过程

归并的算法

```
void Merge(RecType R[], RecType DR[], int k, int m, int
h)
  {  int p, q, n ; p=n=k, q=m+1 ;
     while ((p<=m)&&(q<=h))
       {   if (LQ(R[p].key, R[q].key) )  /*  比较两个子序列 */
              DR[n++]=R[p++] ;
           else   DR[n++]=R[q++] ;
       }
     while (p<=m)     /*   将剩余子序列复制到结果序列中  */
       DR[n++]=R[p++] ;
     while (q<=h)   DR[n++]=R[q++] ;
  }
```

2 一趟归并排序

一趟归并排序都是从前到后，依次将相邻的两个有序子序列归并为一个，且除最后一个子序列外，其余每个子序列的长度都相同。设这些子序列的长度为**d**，则一趟归并排序的过程是：

从**j=1**开始，依次将相邻的两个有序子序列**R[j…j+d-1]**和**R[j+d…j+2d-1]**进行归并；每次归并两个子序列后，**j**后移动**2d**个位置，即**j=j+2d**；若剩下的元素不足两个子序列时，分以下两种情况处理：

① 剩下的元素个数>**d**：再调用一次上述过程，将一个长度为**d**的子序列和不足**d**的子序列进行归并；

② 剩下的元素个数≤**d**：将剩下的元素依次复制到归并后的序列中。

(1) 一趟归并排序算法

```
void Merge_pass(RecType R[], RecType DR[], int d, int n)
  {  int j=1 ;
     while ((j+2*d-1)<=n)
        {   Merge(R, DR, j, j+d-1, j+2*d-1) ;
            j=j+2*d ;
        }     /*  子序列两两归并  */
     if (j+d-1<n)      /*  剩余元素个数超过一个子序列长度d  */
        Merge(R, DR, j, j+d-1, n) ;
     else  Merge(R, DR, j, n, n) ;/*  剩余子序列复制  */
  }
```

(2) 归并排序的算法

开始归并时，每个记录是长度为1的有序子序列，对这些有序子序列逐趟归并，每一趟归并后有序子序列的长度均扩大一倍；当有序子序列的长度与整个记录序列长度相等时，整个记录序列就成为有序序列。算法是：

```
void Merge_sort(Sqlist *L, RecType DR[])
  {  int d=1 ;
      while(d<L->length)
        {   Merge_pass(L->R, DR, d, L->length) ;
            Merge_pass(DR, L->R, 2*d, L->length) ;
            d=4*d ;
        }
  }
```

3 算法分析

具有n个待排序记录的归并次数是$\log_2 n$，而一趟归并的时间复杂度为O(n)，则整个归并排序的时间复杂度无论是最好还是最坏情况均为$O(n\log_2 n)$。在排序过程中，使用了辅助向量DR，大小与待排序记录空间相同，则空间复杂度为O(n)。归并排序是稳定的。

10. 6 基数排序

基数排序(**Radix Sorting**) 又称为桶排序或数字排序：按待排序记录的关键字的组成成分(或“位”)进行排序。

基数排序和前面的各种内部排序方法完全不同，不需要进行关键字的比较和记录的移动。借助于多关键字排序思想实现单逻辑关键字的排序。

10.6.1 多关键字排序

设有n个记录$\{R_1, R_2, …,R_n\}$， 每个记录R_i的关键字是由若干项(数据项)组成，即记录R_i的关键字Key是若干项的集合：$\{K_i^1, K_i^2, …,K_i^d\}(d>1)$ 。

记录$\{R_1, R_2, …,R_n\}$有序的，指的是$\forall i, j \in [1,n]$，$i<j$ ，若记录的关键字满足：

$\{K_i^1, K_i^2, …K_i^d\}<\{K_j^1, K_j^2, …K_j^d\}$，

即$K_i^p \le K_j^p$ $(p=1, 2, … d)$

多关键字排序思想

先按第一个关键字K^1进行排序，将记录序列分成若干个子序列，每个子序列有相同的K^1值；然后分别对每个子序列按第二个关键字K^2进行排序，每个子序列又被分成若干个更小的子序列；如此重复，直到按最后一个关键字K^d进行排序。

最后，将所有的子序列依次联接成一个有序的记录序列，该方法称为最高位优先(**Most Significant Digit first**)。

另一种方法正好相反，排序的顺序是从最低位开始，称为最低位优先(**Least Significant Digit first**)。

10.6.2 链式基数排序

若记录的关键字由若干确定的部分(又称为“位”)组成，每一位(部分)都有确定数目的取值。对这样的记录序列排序的有效方法是基数排序。

设有n个待排序记录$\{R_1, R_2, \ldots, R_n\}$，(单)关键字是由d位(部分)组成，每位有r种取值，则关键字R[i].key可以看成一个d元组：$R[i].key=\{K_i^1, K_i^2, \ldots, K_i^d\}$。

基数排序可以采用前面介绍的MSD或LSD方法。以下以LSD方法讨论链式基数排序。

1 排序思想

⑴ 首先以静态链表存储n个待排序记录，头结点指针指向第一个记录结点；

⑵ 一趟排序的过程是：

① 分配： 按K^d值的升序顺序，改变记录指针，将链表中的记录结点分配到r个链表(桶)中，每个链表中所有记录的关键字的最低位(K^d)的值都相等，用f[i]、e[i]作为第i个链表的头结点和尾结点；

② 收集：改变所有非空链表的尾结点指针，使其指向下一个非空连表的第一个结点，从而将r个链表中的记录重新链接成一个链表；

⑶ 如此依次按K^{d-1}, K^{d-2}, … K^1分别进行，共进行d趟排序后排序完成。

2 排序示例

设有关键字序列为1039, 2121, 3355, 4382, 66, 118的一组记录，采用链式基数排序的过程如下图10-12所示。

初始链表

head| 1039| 2121| 3355| 4382| 0066| 0118 |∧

分配：

f[0]	f[1]	f[2]	f[3]	f[4]	f[5]	f[6]	f[7]	f[8]	f[9]
	2121	4382			3355	0066		0118	1039
e[0]	e[1]	e[2]	e[3]	e[4]	e[5]	e[6]	e[7]	e[8]	e[9]

第一趟收集结果：

head| 2121| 4382| 3355| 0066| 0118| 1039|∧

分配：

f[0]	f[1]	f[2]	f[3]	f[4]	f[5]	f[6]	f[7]	f[8]	f[9]
	0118	2121	1039		3355	0066		4382	
e[0]	e[1]	e[2]	e[3]	e[4]	e[5]	e[6]	e[7]	e[8]	e[9]

第二趟收集结果：

head| 0118| 2121| 1039| 3355| 0066| 4382|∧

分配：

f[0]	f[1]	f[2]	f[3]	f[4]	f[5]	f[6]	f[7]	f[8]	f[9]
1039	0118		3355						
0066	2121		4382						
e[0]	e[1]	e[2]	e[3]	e[4]	e[5]	e[6]	e[7]	e[8]	e[9]

第三趟收集结果：

head| 1039| 0066| 0118| 2121| 3355| 4382|∧

分配：f[0] f[1] f[2] f[3] f[4] f[5] f[6] f[7] f[8] f[9]

0066 1039 2121 3355 4382

0118

e[0] e[1] e[2] e[3] e[4] e[5] e[6] e[7] e[8] e[9]

第四趟收集结果：

head | 0066 | 0118 | 1039 | 2121 | 3355 | 4382 | ∧

图10-12 以LSD方法进行链式基数排序的过程

3 链式基数排序算法

为实现基数排序，用两个指针数组来分别管理所有的缓存(桶)，同时对待排序记录的数据类型进行改造，相应的数据类型定义如下：

```
#define BIT_key  8    /*  指定关键字的位数d  */
#define RADIX  10    /*  指定关键字基数r  */
typedef  struct  RecType
  {  char  key[BIT_key] ;      /*  关键字域  */
     infoType  otheritems ;
     struct  RecType  *next ;
  }SRecord, *f[RADIX] ,*e[RADIX] ;
     /*   桶的头尾指针数组   */
```

```
void Radix_sort(SRecord *head )
  {   int j, k, m ;
      SRecord  *p, *q, *f[RADIX], *e[RADIX] ;
      for (j=BIT_key-1; j>=0; j--)
              /*  关键字的每位一趟排序  */
        {  for (k=0; k<RADIX; k++)
               f[k]=e[k]=NULL ;   /*  头尾指针数组初始化  */
           p=head ;
           while (p!=NULL)       /*  一趟基数排序的分配  */
             {   m=ord(p->key[j]) ;   /*  取关键字的第j位kj   */
                 if  (f[m]==NULL)  f[m]=p ;
                 else  e[m]->next=p ;
                 p=p->next ;
```

```
            }
        head=NULL ;    /* 以head作为头指针进行收集 */
        q=head ;       /* q作为收集后的尾指针 */
        for (k=0; k<RADIX; k++)
          { if (f[k]!=NULL)   /* 第k个队列不空则收集 */
              { if (head!=NULL) q->next=f[k] ;
                else head=f[k] ;
                q=e[k] ;
              }
          }   /* 完成一趟排序的收集 */
        q->next=NULL ;    /* 修改收集链尾指针 */
      }
}
```

4 算法分析

设有n个待排序记录，关键字位数为d，每位有r种取值。则排序的趟数是d；在每一趟中：

- 链表初始化的时间复杂度：O(r)；
- 分配的时间复杂度：O(n)；
- 分配后收集的时间复杂度：O(r)；

则链式基数排序的时间复杂度为：**O(d(n+r))**

在排序过程中使用的辅助空间是：2r个链表指针，n个指针域空间，则空间复杂度为：**O(n+r)**

基数排序是稳定的。

10.7 各种内部排序的比较

各种内部排序按所采用的基本思想(策略)可分为：插入排序、交换排序、选择排序、归并排序和基数排序，它们的基本策略分别是：

1 插入排序：依次将无序序列中的一个记录，按关键字值的大小插入到已排好序一个子序列的适当位置，直到所有的记录都插入为止。具体的方法有：直接插入、表插入、2-路插入和shell排序。

2 交换排序：对于待排序记录序列中的记录，两两比较记录的关键字，并对反序的两个记录进行交换，直到整个序列中没有反序的记录偶对为止。具体的方法有：冒泡排序、快速排序。

3 选择排序：不断地从待排序的记录序列中选取关键字最小的记录，放在已排好序的序列的最后，直到所有记录都被选取为止。具体的方法有：简单选择排序、堆排序。

4 归并排序：利用"归并"技术不断地对待排序记录序列中的有序子序列进行合并，直到合并为一个有序序列为止。

5 基数排序：按待排序记录的关键字的组成成分("位")从低到高(或从高到低)进行。每次是按记录关键字某一"位"的值将所有记录分配到相应的桶中，再按桶的编号依次将记录进行收集，最后得到一个有序序列。

各种内部排序方法的性能比较如下表。

表7-1 主要内部排序方法的性能

方法	平均时间	最坏所需时间	附加空间	稳定性
直接插入	$O(n^2)$	$O(n^2)$	$O(1)$	稳定的
Shell排序	$O(n^{1.3})$		$O(1)$	不稳定的
直接选择	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定的
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定的
冒泡排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定的
快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)$	不稳定的
归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定的
基数排序	$O(d(n+r))$	$O(d(n+r))$	$O(n+r)$	稳定的

讲稿中讨论的排序方法是在顺序存储结构上实现的，在排序过程中需要移动大量记录。当记录数很多、时间耗费很大时，可以采用静态链表作为存储结构。但有些排序方法，若采用静态链表作存储结构，则无法实现表排序。

选取排序方法的主要考虑因素：

- 待排序的记录数目**n**；
- 每个记录的大小；
- 关键字的结构及其初始状态；
- 是否要求排序的稳定性；
- 语言工具的特性；
- 存储结构的初始条件和要求；
- 时间复杂度、空间复杂度和开发工作的复杂程