第7章 内部分类

7.0 术语和约定

一、分类及其目的

分类(Sorting)也叫排序(Ordering)，是将一组数据按照规定顺序进行排列，其目的是为了方便查询和处理。

二、分类的种类

- 按分类时分类对象存放的设备，分成内部分类(internal sorting)和外部分类(external sorting)。
- 分类过程中数据对象全部在内存中的分类，叫内部分类。
- 分类过程数据对象并非完全在内存中的分类，叫外部分类。

三、分类表的存储结构

```
struct  records {
            keytype   key ;
            fields   other ;
};
typedef  records  LIST[maxsize] ;
```


7.0 术语和约定

四、分类的稳定性

对于给定数组**A**，经过分类处理之后，满足关系：

$$A[1].key \leq A[2].key \leq \cdots\cdots \leq A[n].key$$

如果在分类之前存在关系

$$A[i].key \leq A[j].key \quad (1 \leq i < j \leq n)$$

经分类后，**A[i]**和**A[j]**分别被移至$A[i_1]$和$A[j_1]$，并且i_1和j_1满足关系

$$1 \leq i_1 < j_1 \leq n$$

我们称这种分类是稳定的，否则称其为不稳定的分类。

五、影响分类性能的因素

- 比较关键字的次数——如当关键字时字符串时，是主要因素。
- 交换记录位置和移动记录的次数——当记录很大时，是主要因素。

7.1 简单的分类算法

7.1.1 气泡分类

```
void  BubbleSort ( int  n , LIST  &A )
{  int  x, y ;
   for ( i =1; i <= n-1; i++ )
      for ( j =n; j >= i+1; j-- )
         if ( A[j].key < A[j-1] )
            swap (A[j], A[j-1])
}
void  swap(x, y)
records &x, records  &y
{   records   w ;
    w = x ;
     x = y ;
     y = w ;
}
```

时间复杂性：

$$f(n)=C_3n+\sum_{i=1}^{n-1} C_2\cdot(n-i)$$
$$=1/2\cdot C_2n^2+(C_3-1/2\cdot C_2)\cdot n$$
$$\leq (C_2/2+C_3)n^2 \quad \text{当} n^2 \geq 1 \text{时}$$
$$=Cn^2$$
$$T(n) = O(f(n)) = O(C\cdot n^2) = O(n^2)$$

	key	other
0		
1	P	
2	E	
3	K	
4	A	
5	V	
6	S	
7		

7.1.2 插入分类

```
void  InsertSort (int n, LIST A )
{  int  i, j ;
   A[0].key = -∞ ;
   for(i=1; i<=n; i++) {
      j=i;
      while(A[j].key<A[j-1].key) {
         swap(A[j],A[j-1]) ;
         j=j-1;
      }
   }
}
```

	key	other
0	-∞	
1	P	
2	E	
3	K	
4	A	
5	V	
6	S	
7		

时间复杂性：$T(n)=O(n^2)$

7.1.3 选择分类

```
void  SelectSort (int n, LIST A )
{   int  i, j，lowindex ;
    keytype lowkey;
    for(i=1; i<n; i++) {
       lowindex = i ;
       lowkey = A[i].key ;
       for ( j = i+1; j<=n; i++)
         if (A[j].key<lowkey) {
            lowkey=A[j] ;
            lowindex = j ;
         }
       swap(A[i],A[lowindex]) ;
    }
}
```

时间复杂性：$T(n)=O(n^2)$

	key	other
0		
1	P	
2	E	
3	K	
4	A	
5	V	
6	S	
7		

7.2 快速分类—划分交换排序

是**C.R.A.Hoare 1962**年提出的一种划分交换排序。采用的是分治策略**(**一般与递归技术结合使用**)**，以减少分类过程之中的比较次数。

一、分治法的基本思想

- **分解:** 将原问题分解为若干个与原问题相似的子问题，又称划分；
- **求解:** 递归地求解子问题，若子问题的规模足够小，则直接求解
- **组合:** 将每个子问题的解组合成原问题的解。

7.2 快速分类—划分交换排序

二、快速分类的基本思想**(**利用分治法**)**

设被分类的无序区为**A[i],……,A[j]**

1)选择表（文件**,**数据集合）中的一个记录**(**数据元素**)**的关键字**v** 作为基准元素**(**控制关键字**);(**怎么选择？**)**

2)通过基准元素**v** 把表**(**文件**,**数据集合**)**划分成左、右两部分，使得左边的各记录的关键字都小于**v** ；右边的各记录的关键字都大于等于**v**；**(**如何划分？**)**

3)重复**(1)~(2)**，分别对左边和右边部分递归的进行快速分类；

4)组合：左、右两部分均有序，整个表有序。

因此，快速分类的关键问题是：

- 基准元素的选取：

3	1	4	1	5	9	2	6	5	3

- 表的划分：**1.**扫描 **2.**测试 **3.**交换

7.2 快速分类—划分交换排序

三、基准元素选取

- 基准元素的选取是任意的，但不同的选取方法对算法性能影响很大；
- 一般原则：是每次都能将表划分为规模相等的两部分（最佳情况）。此时，划分次数为$\log_2 n$，全部比较次数$n\log_2 n$，交换次数$(n/6)\log_2 n$ 。
- 设 **FindPivot(i , j),**是求**A[i].key,…,A[j].key**的基准元素**v=A[k]**，返回其下标**k** 。

v =(A[i].key， A[(i+j)/2].key，A[j].key的中值 **)**

v =从**A[i].key**到**A[j].key** 最先找到的两个不同关键字中的最大者。 （若**A[i].key,…A[j].key**之中至少有两个关键字不相同）

优点：若无两个关键字不同，则**A[i]**到**A[j]**已有序，分类结束。

3	1	4	1	5	9	2	6	5	3

7.2 快速分类—划分交换排序

```
/* 设A是外部数组 */
int  FindPivot( int  i, int  j )
/*若A[i],…A[j]的关键字全部相同，则返回0；
   否则，左边两个不同关键字中的较大者的下标.
*/
{  keytype  firstkey ; /* 第1个关键字的值A[i].key */
   int  k ;                /* 从左到右查找不同的关键字 */
   firstkey = A[i].key ;
   for ( k=i+1 ;  k<=j; k++ ) /* 扫描不同的关键字 */
      if ( A[k].key > firstkey ) /* 选择较大的关键字 */
         return  k ;
      else if ( A[k].key < firstkey )
         return  i ;
   return  0 ;
}
```


7.2 快速分类—划分交换排序

四、表的划分(分割)

1) 扫描**:**

令游标 **l** 从左端(初始时 **l = i)**开始向右扫描，越过关键字小于**v**的所有记录，直到遇到**A[l].key**≥**v**；

又令游标 **r** 从右端(初始时 **r = j)**开始向左扫描，越过关键字大于等于**v** 的所有记录,直到遇到**A[r].key<v**；

2) 测试 **l** 和 **r**：若**l < r,**则转**(3)**，否则**(l > r,** 即 **l = r+1)**转**(4)**；

3) 交换：交换**A[l]**和**A[r]**，转**(1)**；**(**目的是使 **l** 和 **r** 都至少向其前进方向前进一步**)**

4) 此时**A[i],…A[j]**被划分成为满足条件的两部分**A[i],…A[l-1]**和**A[l],…,A[j]**。

3	1	4	1	5	9	2	6	5	3

7.2 快速分类—划分交换排序

四、表的划分(分割)

int Partition (int i , int j , keytype pivot) **P217**

```
/*划分A[i],…,A[j]，是关键字 < pivot 的在左子序列，
  关键字≥pivot 的在右子序列，返回有子序列的起始下标*/
{ int l , r ;
  do{
       for( l=i ; A[l].key  <  pivot ; l++ ) ;
       for( r=j ; A[l].key >= pivot ; r--) ;
       if( l < r )
            swap(A[l],A[r]);
  } while( l <= r );
  return  l ;
}
```

⇦ 分割:

[i, … …, j]

⇩

[i, … k-1, k, k+1, … j]

3	1	4	1	5	9	2	6	5	3

7.2 快速分类—划分交换排序

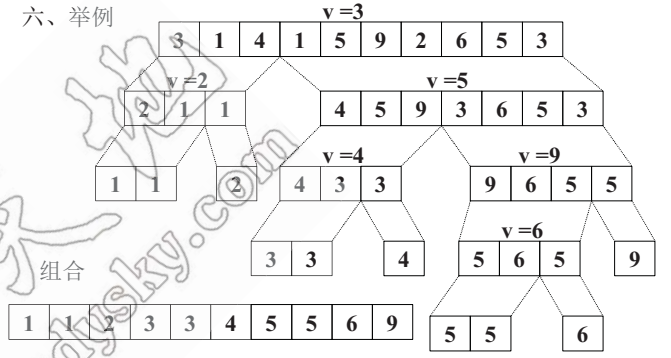
```
void QuickSort ( int i , int  j )        五、快速分类算法
/*对外部数组A 的元素A[i ],…,A[j]进行快速分类*/
{  keytype pivot;
   int k; //关键字大于等于pivot的记录在序列中的起始下标
   int pivotindex ; //关键字为pivot的记录在数组A中的下标
   pivotindex = FindPivot ( i , j );
   if( pivotindex != 0 ) {  //递归终止条件
        pivot=A[pivotindex].key;
        k=Partition ( i , j , pivot );
        QuickSort ( i , k -1);
        QuickSort ( k  , j );
   }
}//对数组A[1],…,A[n]进行快速分类可调用QuickSort(1,n)实现
```


7.2 快速分类—划分交换排序

五、时间复杂性和稳定性

时间复杂性：$T(n) = n\log_2 n$　　稳定性：是不稳定的分类

六、举例



7.3 归并分类—二路归并分类

- 归并技术：将若干个已排序的数据集合（子序列文件）合并成一个有序的数据集合（有序文件）。
- 归并分类（排序）：就是利用"归并"技术来进行分类。
- 基本思想：自顶向下和自底向上

7.3.1 合并两个分类序列（基础）

设A[l],…,A[m]是一个按关键字有序的序列，而A[m+1], …,A[n]是另一个按同一关键字的有序序列。现将它们合并在一起，形成一个分类序列B[l], …,B[n]，(类似玩扑克，两堆各自有序扑克牌，合并为一堆且有序)。实现函数Merge如下：

7.3 归并分类—二路归并分类

```
void Merge (int l , int m , int n , LIST  A , LIST  B)  //P219
/*将已分类序列A[ l ],…,A[m]和A[m+1], …,A[n]合并为一个分类序列
  B[ l ],…,B[n]*/
{  int i = l ;  j = m+1 , k = l ;//置初值
     /* 两个序列非空时，取小者输出到B[k]上 */
   while ( i <= m && j <= n )
       B[k++] = ( A[ i ].key <= A[ j ].key) ? A[i+1] : A[j++] ;
     /* 若第一个子序列非空(未处理完)，则复制剩余部分到B */
   while ( i <= m )   B[k++] = A[i++] ;
     /* 若第二个子序列非空(未处理完)，则复制剩余部分到B */
   while ( j <= n )    B[k++] = A[j++] ;
}
```

时间复杂性：O(n－l＋1)；　　空间复杂性：O(n－l＋1)

7.3.2 归并分类（自底向上）

算法要点：

1) 首先把每个记录看成是一个有序序列，共**n**个，将它们两两合并成⌈ **n/2** ⌉个分类序列，每个序列长度为**2**（当**n**为奇数时，最后一个序列长度为**1**）。

2) 对⌈ **n/2** ⌉个分类序列，再两两归并在一起；

3) 如此进行，直到归并成一个长度为**n** 的分类序列为止。

{26} {5} {77} {1} {61} {11} {59} {15} {48} {19}

{ 5 26 } { 1 77 } { 11 61 } { 15 59 } { 19 48 } #

{ 1 5 26 77 } { 11 15 59 61 } { 19 48 } #

{ 1 5 11 15 26 59 61 77 } { 19 48 } *

{ 1 5 11 15 19 26 48 59 61 77 }

7.3.2 归并分类（自底向上）

▪第**1**遍归并的子序列长度为2^0，第**2**遍为2^1，…，第**i**遍为2^{i-1}，所以由 $2^{i-1} >= n$ 知，对**n** 个记录的数据集合，总共需要归并$\log_2 n$次。

▪合并两个分类序列所需要的时间正比于其中的记录数，所以每遍归并正比于**n**，对于$\log_2 n$遍归并，总时间复杂性为$O(n\log_2 n)$。

{26} {5} {77} {1} {61} {11} {59} {15} {48} {19}

{ 5 26 } { 1 77 } { 11 61 } { 15 59 } { 19 48 } #

{ 1 5 26 77 } { 11 15 59 61 } { 19 48 } #

{ 1 5 11 15 26 59 61 77 } { 19 48 } *

{ 1 5 11 15 19 26 48 59 61 77 }

7.3.2 归并分类（自底向上）

假设第**i**遍归并的长度$2^{i-1}= l$，则

▪当子序列的个数为偶数时，最后一个子序列的长度可能小于**l**，此时应注意最后一个子序列的下标上界为 n 。

▪当子序列的个数为奇数时，则最后一个子序列无需与其它子序列归并（即轮空），直接复制到归并序列之后即可。

▪总之，每遍归并时，必须对子序列的个数是奇数、以及最后一个子序列长度可能小于 **l** 两种特殊情况进行特殊处理。

{26} {5} {77} {1} {61} {11} {59} {15} {48} {19}

{ 5 26 } { 1 77 } { 11 61 } { 15 59 } { 19 48 } #

{ 1 5 26 77 } { 11 15 59 61 } { 19 48 } #

{ 1 5 11 15 26 59 61 77 } { 19 48 } *

{ 1 5 11 15 19 26 48 59 61 77 }

7.3.2 归并分类（自底向上）

执行一遍归并的算法 **Mpass**

```
/*把A中长度为 l 的相邻序列归并成长度为 2l 的序列的函数 220页*/
void  Mpass (int n , int l , LIST A , LIST B)
/* 把A中长度均为 l 的相邻两个分类子序列归并入B， n 为A的记录总数 */
{ int i , t ;
for ( i=1 ; i+2*l-1<= n ; i+=2*l )
  Merge ( i , i+l-1, i+2*l-1, A , B ) ;/* 归并长度为 l 的两个分类子序列 */
if ( i+l-1< n )   /* 尚有两个子序列，其中最后一个长度小于 l */
  Merge ( i , i+l-1 , n , A , B ) ;    /* 归并最后两个子序列 */
else   /* 若i<= n且i+l-1>= n 时，则剩余一个子序列轮空，直接复制 */
  for ( t= i ; t<= n ; t++ )
    B[ t ] = A[ t ] ;
}  /* Mpass */
```

7.3.2 归并分类（自底向上）

归并分类算法：(用Mpass写)

```
void MergeSort ( int n , LIST A )
{   /* 二路归并分类 */
    int l = 1 ;/* 当前归并子序列的长度，初始为1 */
    LIST B ;
    while (l< n){
        Mpass (n , l , A , B) ;
        l = 2*l ;
        Mpass (n , l , B , A) ; /* A、B互换位置 */
        l = 2*l ;
    }
}/* MergeSort */
```

- 自底向上的归并算法效率较高，但可读性差。
- 若采用分治技术自顶向下的算法，形式简洁。

7.3.3 归并分类的分治算法（自顶向下）

基本思想：

- 分解：将当前表区间一分为二，即求分裂点 **mid = (low + high) / 2** ；
- 求解：递归地对表分区 **A[low], …, A[mid]**和**A[mid+1], …, A[high]**进行归并分类；
- 组合：将已分类的两个子序列区间归并为一个有序序列区间。

注意：递归的终止条件是子区间长度为 **1**，因为一个记录自然有序。

7.3.3 归并分类的分治算法（自顶向下）

归并分类的分治递归算法

```
void MergeSort ( LIST A , LIST B , int low , int high )
/* 用分治法对A[low], …, A[high]进行二路归并 */
{   int mid = (low+high)/2 ;
    if (low<high){ /* 区间长度大于 1 ， high-low>0 */
        MergeSort (A , B , low , mid) ;
        MergeSort (A , B , mid+1 , high) ;
        Merge (low , mid , hight , A , B) ;
    }
}   /* MergeSort */
```

注：归并分类是稳定分类。

7.4 堆（Heap）分类

定义：把具有如下性质的数组**A**表示的完全二元树称为堆：

(1) 若**2*i ≤ n**，则**A[i].key ≤ A[2*i].key ;**

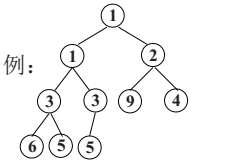
(2) 若**2*i+1 ≤ n**，则**A[i].key ≤ A[2*i+1].key ;** 小顶堆

或： 把具有如下性质的数组**A**表示的完全二元树称为堆：

(1) 若**2*i ≤ n**，则**A[i].key ≥ A[2*i].key ;**

(2) 若**2*i+1 ≤ n**，则**A[i].key ≥ A[2*i+1].key ;** 大顶堆

例：



1	1	2	3	3	9	4	6	5	5

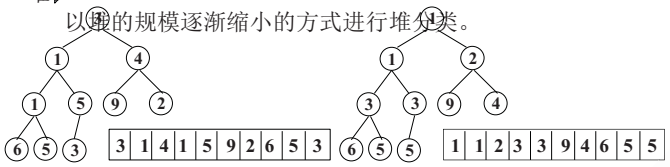
7.4 堆（Heap）分类

注意：上述二条性质说明：

- 对于任意一个非叶结点上的关键字，都不大于其左、右儿子结点上的关键字。即**A[i/2].key ≤ A[i].key　1 ≤ i/2 <i ≤ n**。
- 在堆中，以任意结点为根的子树仍然是堆。特别地，每个叶也点也可视为堆。每个结点都代表(是)一个堆。

  ⇨ 以堆（的数量）不断扩大的方式进行初始建堆。
- 在堆中（包括各子树对应的堆），其根结点的关键字是最小的。去掉堆中下标最大（或编号最大）的叶结点，仍然是堆。

  ⇨ 以堆的规模逐渐缩小的方式进行堆分类。



7.4 堆（Heap）分类

二、堆分类的基本思想：

- 把待分类的数据集合（序列）用完全二元树的数组存储结构**A**表示。
- 初始建堆：把数组所对应的完全二元树以堆不断扩大的方式整理成堆。令 **i= n/2,…,2,1** 分别把以**n/2 ,…,2,1** 为根的完全二元树整理成堆，即执行算法**PushDown (i , n)**。
- 堆分类：令**i = n, n -1 ,…, 2**，

  1.交换:把堆顶元素（当前最小的）与位置 **i**（当前最大的叶结点下标）的元素交换，即执行**swap(A[1],A[i])**；

  2.整理:把剩余的**i-1**个元素整理成堆，即执行**PushDown(1,i-1)**；

  3.重复执行完这一过程之后，则**A[1],A[2],…,A[n]**是按关键字不增顺序的有序序列。

待分类的数据集合 → 表示 → 完全二元树 → 整理 → 堆(初始建堆) → 分类 → 按关键字有序的数据集合

7.4 堆（Heap）分类

三、堆分类算法：

```
void HeapSort ( int n , LIST A )
{  int i;
   for(i=n/2;i<=1;i++)  /*初始建堆，从最右非叶结点开始*/
      PushDown(i,n);  /*整理堆，把以i为根，最大下标的叶为n*/
   for(i=n;i<=2;i--) {
      swap(A[1],A[i]); /*堆顶与当前堆中的下标最大的叶结点交换*/
      PushDown(1,i-1);
      /*整理堆把以1为根，最大叶下标为i-1的完全二元树整理成堆*/
    }
}
```

$$T(n) = O(n \cdot \log_2 n)$$

7.4 堆（Heap）分类

四、整理堆算法：**PushDown(first , last)**

该操作的功能是把以**A[first]**为根，以**A[last]**为最右边叶的完全二元树整理成堆。根据堆的定义，它要完成的功能是，把完全二元树中的关键字最小的元素放到堆顶，而把原堆顶元素下推到适当的位置，使**(A[first],…,A[last])**成为堆。

那么，怎样把关键字最小的元素放到堆顶，把堆顶元素下推到适当位置呢？

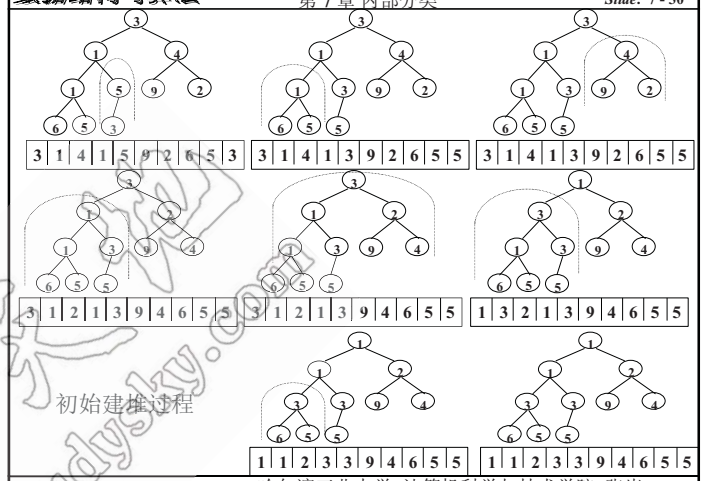
具体操作**(**要点**)**如下：

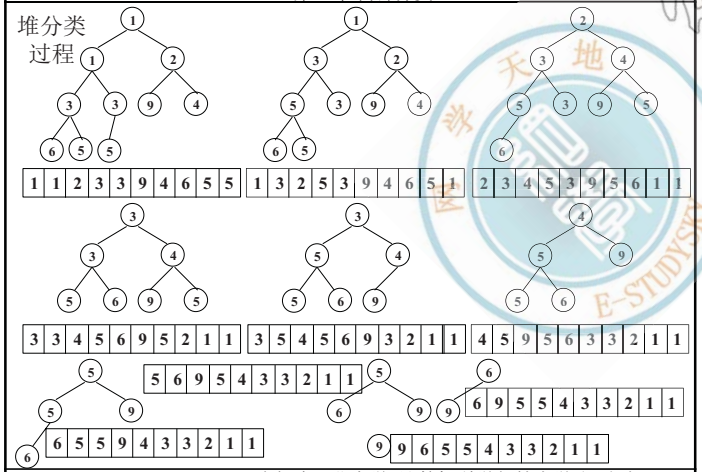
把完全二元树的根或者子树的根与其左、右儿子比较，如果它比其左 / 右儿子大，则与其中较小者交换（若左、右儿子相等，则与其左儿子交换）。重复上述过程，直到以**A[first]**为根的完全二元树是堆为止。

算法如下：

```
void PushDown(int first,int last)
{/*整理堆:A是外部数组,把A[first]下推到完全二元树的适当位置*/
  int r=first; /* r是被下推到的适当位置，初始值为根first*/
  while(r<=last/2) /* A[r]不是叶，否则是堆 */
    if((r==last/2) && (last%2==0)) {/* r有一个儿子在2*r上且为左儿子*/
        if(A[r].key>A[2*r].key)
            swap(A[r],A[2*r]);/*下推*/
        r=last;  /*A[r].key小于等于A[2*r].key或者"大于"，交换后到叶，循环结束*/
    } else if((A[r].key>A[2*r].key)&&(A[2*r].key<=A[2*r+1].key)) {
        /*根大于左儿子，且左儿子小于或等于右儿子*/
        swap(A[r],A[2*r]);       /*与左儿子交换*/
        r=2*r;              /*下推到的位置也是下次考虑的根*/
    } else if((A[r].key>A[2*r+1].key)&&(A[2*r+1].key<A[2*r].key)) {
        /*根大于右儿子，且右儿子小于左儿子*/
        swap(A[r],A[2*r+1]);     /*与右儿子交换*/
        r=2*r+1;          /*下推到的位置也是下次考虑的根*/
    }else /*A[r]符合堆的定义，不必整理，循环结束*/
        r=last;
}/*PushDown*/
```

$O(\log_2(\text{last/first}))=O(\log_2 n)$





7.4 堆（Heap）分类

五、时间复杂性

PutDown函数中，执行一次**while** 循环的时间是一个常数。因为**r** 每次至少为原来的两倍，假设**while**循环执行次数为 **i**，则当**r** 从**first** 变为**first*2^i** 时循环结束。此时**r=first*2^i >last/2**，即 **i >log(last/first)-1)**。所以**while**循环体最多执行**$\log_2$ (last/first)** 次，即**PushDown**时间复杂性**O(log(last/first))=O($\log_2$ n)**。

∴HeapSort时间复杂性**O(n $\log_2$ n)**

六、稳定性

不稳定的，举出反例

7.5 基数分类——多关键字分类

理论上可以证明，对于基于关键字之间比较的分类，无论用什么方法都至少需要进行$\log_2 n!$次比较。

由Stirling公式可知，$\log_2 n! \approx n\log_2 n - 1.44n + O(\log_2 n)$。所以基于关键字比较的分类时间的下界是$O(n\log_2 n)$。因此不存在时间复杂性低于此下界的基于关键字比较的分类!

只有不通过关键字比较的分类方法,才有可能突破此下界.
一、基数分类（时间复杂性可达到线性级O(n)）

不比较关键字的大小，而根据构成关键字的每个分量的值，排列记录顺序的方法，称为分配法分类（基数分类）。

而把关键字各个分量所有可能的取值范围的最大值称为基数或桶或箱，因此基数分类又称为桶分类。

显然，要求关键字分量的取值范围必须是有限的，否则可能要无限的箱。(基数分类的适用范围)

7.5 基数分类——多关键字分类

二、算法的基本思想

设被分类的数据的关键字都是位相同的整数(不相同时,取位数的最大值)，其位数为figure，每个关键字可以各自含有figure个分量，每个分量的值取值范围为0,1,…,9即基数为10。依次从低位考查,每个分量。

首先把全部数据装入一个排队A，然后按下列步骤进行:

1.初态:设置10个排队，分别为Q[0],Q[1],…,Q[9]，并且均为空。

2.分配:依次从排队中取出每个数据data；第pass遍处时，考查data.key右起第pass位数字，设其为r，把data插入排队Q[r]，取尽A，则全部数据被分配到Q[0],Q[1],…,Q[9].

3.收集:从Q[0]开始，依次取出Q[0],Q[1],…,Q[9]中的全部数据，并按照取出顺序，把每个数据插入排队A。

4.重复1,2,3步，对于关键字中有figure位数字的数据进行figure遍处理，即可得到按关键字有序的序列。

7.5 基数分类——多关键字分类 ♥ ♠ ♦ ♣

321 986 123 432 543 018 765 678 987 789 098 890 109 901 210 012

Q[0]:890 210	Q[0]:901 109	Q[0]:012 018 098
Q[1]:321 901	Q[1]:210 012 018	Q[1]:109 123
Q[2]:432 012	Q[2]:321 123	Q[2]:210
Q[3]:123 543	Q[3]:432	Q[3]:321
Q[4]:	Q[4]:543	Q[4]:432
Q[5]:765	Q[5]:	Q[5]:543
Q[6]:986	Q[6]:765	Q[6]:678
Q[7]:987	Q[7]:678	Q[7]:765 789
Q[8]:018 678 098	Q[8]:986 987 789	Q[8]:890
Q[9]:789 109	Q[9]:890 098	Q[9]: 901 986 987

890 210 321 901 432 012 123 543 765 986 987 018 678 098 789 019

901 109 210 012 018 321 123 432 543 765 678 986 987 789 890 098

012 018 098 109 123 310 321 432 543 678 765 789 890 901 986 987


```
void RadixSort ( int figure , QUEUE &A )
{ QUEUE Q[10] ;
  records data ;
  int pass, r, i ;
  for ( pass=1 ; pass <=figure ; pass+ ) {
    for ( i=0 ; i<=9 ; i++ ) /*置空队列*/
        MAKENULL( Q[i] ) ;
    while ( !EMPTY( A ) ){/* 分配 */
       data = FRONT ( A ) ;
       DEQUEUE ( A ) ;
       r = Radix ( data. key , pass ) ;
       ENQUEUE( data , Q[r] ) ; }
    for ( i=0 ; i <=9 ; i++ ) /* 收集 */
       while ( !EMPTY( Q[i] ) ) {
            data = FRONT ( Q[i] ) ;
            DEQUEUE( Q[i] ) ;
            ENQUEUE( data , A ) ; }
  }
}
```

```
/ *求整数 k 的第 p 位*/
int Radix ( int k, int p )
{int power , i ;
 power = 1 ;
 for ( i=1; i<=p-1 ; i++ )
       power = power * 10 ;
 return (( k%(power*10))/power) ;
}
```

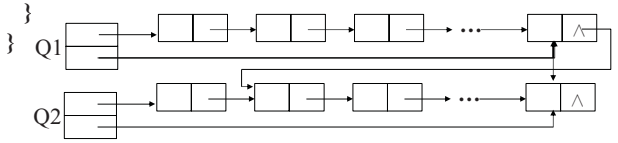
```
for (i=0;i<=9;i++) {
   Concatenate(Q[1], Q[i]);
   A=Q[0];
}/*大大缩短收集操作的时间*/
```

7.5 基数分类——多关键字分类

四、算法改进

由于每个桶（箱）存放多少个关键字分量相同的记录个数无法预料，即队列**Q[0]，Q[1],…,Q[9]**长度很难确定，故桶一般设计成链式排队，两个排队链在一起的方法如下：**(P227)**

```
void Concatenate(QUEUE Q1,QUEUE Q2)
{ if ( !EMPTY( Q2 ) ) {
      Q1.rear->next=Q2.front->next;
      Q1.rear=Q2.rear;
  }
}
```



于是**RadixSort**算法改进见上页

7.5 基数分类——多关键字分类

五、推广和应用

- 若被分类的数据关键字由若干域组成，可以把每个域看成一个分量按照每个域进行基数分类。
- 若关键字各分量不是整数，则把各分量所有可以取值与一组自然数对应。

六、时间复杂性

主要花在分配操作上：**O(d• n)**，是线性的。

其中**d** 是关键字分量位数

七、稳定性：是的

小结——各种分类的比较

分类方法	平均时间	最坏情况	辅助空间	稳定性
简单分类	**O(n^2)**	**O(n^2)**	**O(1)**	选择NO
快速分类	**O(n·logn)**	**O(n^2)**	**O(logn)**	**NO**
堆分类	**O(n·logn)**	**O(n·logn)**	**O(1)**	**NO**
归并分类	**O(n·logn)**	**O(n·logn)**	**O(n)**	**YES**
基数分类	**O(d· (n+r·d))**	**O(d· (n+r·d))**	**O(r·d)**	**YES**

(1) 平均时间性能

(2) 当序列¡°基本有序¡±时，简单分类中的插入分类最佳

(3) 基数分类适用于**n**值很大而关键字较小的序列

(4) 稳定性以基数分类为最佳

(5) 大多数分类算法的策略是局部有序逐步达到全局有序