第九章 排序

一、基本概念

- 排序的定义
- 数据表
- 排序码/排序关键字
- 排序目的：提高查找效率。

- 排序算法的稳定性:即相对次序没发生改变——稳定，否则不稳定。
- 内排序与外排序
- 内排序的排序方法
- 插入排序、选择排序、交换排序、归并排序和分配排序。

- 排序算法的评价标准：
- （1）排序的时间开销：数据比较次数与数据移动次数
- （2）分析情况：平均情况 最好情况 最坏情况
- 算法执行时所需的附加存储:即辅助排序算法完成所需的存储空间。

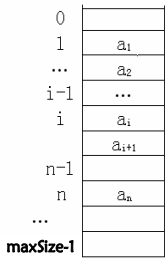
排序算法的实现

- 存储结构
- 算法的设计

存储结构

- (1)顺序存储方式：物理位置反映逻辑关系→调整物理位置→通过比较和移动记录来实现。
- (2)链式存储方式：逻辑关系由指针反映→无须移动记录，只须改变记录间指针的链接。
- (3)索引存储方式：在索引表设有指示各个记录物理位置的地址→无须移动记录，只须修改地址中相应分量的值。

顺序存储——顺序表



顺序表(SeqList)类的定义

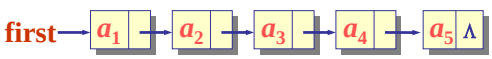
```
const int defaultSize = 100;
template <class T>
class SeqList {
protected:
    T *data;        //存放数组
    int maxSize; //最大可容纳表项的项数
    int last;    //数组中最后一个元素的下标
```

```
public:
    SeqList(int sz = defaultSize);      //构造函数
    int Size() const {return maxSize;}//求表最大容量
    int Length() const {return last+1;} //计算表长度
    int Search(T& x) const;
        //搜索x在表中位置，函数返回表项序号
    ……

    void Sort( ); //排序算法

};
```

排序方法

- 插入排序　void InsertSort( );
- 交换排序　void SwapSort( );
- 选择排序　void SelectSort( );
- 归并排序　void MergeSort( );
- 分配排序　void AssignSort( );
- 快速排序　void QuickSort( );

链式结构



链表的结点类

```
template <class T>

struct LinkNode {      //链表结点类的定义
    T data;                    //数据域
    LinkNode<T> * link;      //链指针域
};
```

单链表类

```
template <class T>
class List {    //单链表类定义
protected:
   LinkNode<T> * first;    //表头指针
public:
   List() { first = new LinkNode<T>; } //构造函数
   int Length() const;        //计算链表的长度
   ……

   void Sort( ); //排序算法

};
```

排序方法

- 插入排序　void InsertSort( );
- 交换排序　void SwapSort( );
- 选择排序　void SelectSort( );
- 归并排序　void MergeSort( );
- 分配排序　void AssignSort( );
- 快速排序　void QuickSort( );

排序的实现——顺序表

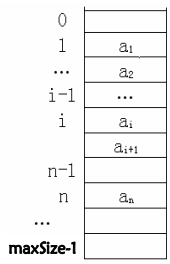
顺序表(SeqList)类的定义

```
const int defaultSize = 100;
template <class T>
class SeqList {
protected:
   T *data;          //存放数组
   int maxSize;  //最大可存放容量
   int last;  //最后一个元素的下标
```

说明：

(1)数组data[1..last]存放待排序序列中记录；

(2)data[0]作为临时单元



排序方法

二、插入排序

基本方法是:每步将一个待排序的对象,按其排序码大小,插入到前面已经排好序的一组对象的适当位置上,直到对象全部插入为止。

具体的实施方案有：

(1) 直接插入排序

(2) 折半插入排序

(3) 希尔排序

1.直接插入排序

- 基本思想：

直接插入排序——顺序表

- 基本思想：
- 当插入data[i] ($i \geq 2$) 对象时，前面的data[1], data[2], …, data[i-1]已经排好序。这时，用data[i]的排序码与data[i-1], data[i-2], …的排序码顺序进行比较，找到插入位置即将data[i]插入，原来位置上的对象向后顺移。

例如，已知一组待排序记录的排序码分别为 40，30，60，90，70，10，20，40，用直接插入法进行排序的过程图 9-1 所示：

单元编号	1	2	3	4	5	6	7	8
初始状态	40	30	60	90	70	10	20	40
第一趟	30	40	60	90	70	10	20	40
第二趟	30	40	60	90	70	10	20	40
第三趟	30	40	60	90	70	10	20	40
第四趟	30	40	60	70	90	10	20	40
第五趟	10	30	40	60	70	90	20	40
第六趟	10	20	30	40	60	70	90	40
第七趟	10	20	30	40	40	60	70	90

说明：有阴影部分为当前已排序的记录

下面给出其算法的描述：

```
void  SeqList<T>::InsertSort( )
{
 int i, j ;  T x;
 for (i=2 ; i<=last ; i++)//依次将data[2],...data[last]插入序列中
  {
    x=data[i];     // 暂存元素到x中
    j=i-1;
    while ( x < data[j] ) //查找x(即原data[i])的插入位置
       data[j+1]=data[j- -];//将序列中排序码大于x的记录往后移
    data[j+1]=x ;       // 填入data[i]
  } // for
} // InsertSort
```

发现问题

```
j=i-1;
while ( x<data[j] )//查找x的插入位置
   data[j+1]=data[j- -];//将排序码大于x的记录后移
```

单元编号	1	2	3	4	5	6	7	8
第四趟	30	40	60	70	90	10	20	40

```
while ( x<data[j] && j>0 )//查找x的插入位置
```

下面给出其算法的描述：

```
void  SeqList<T>::InsertSort( )
{
 int i, j ;  T x;
 for (i=2 ; i<=last ; i++)//依次将data[2],...data[last]插入序列中
  {
    x=data[i];     // 暂存元素到x中
    j=i-1;
    while ( x < data[j] && j>0 ) //查找x(即原data[i])的插入位置
       data[j+1]=data[j- -];//将序列中排序码大于x的记录往后移
    data[j+1]=x ;       // 填入data[i]
  } // for
} // InsertSort
```

发现问题

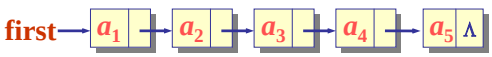
```
j=i-1;
while ( x<data[j] && j>0 )//查找x的插入位置
   data[j+1]=data[j- -];//将排序码大于x的记录后移
```

单元编号	1	2	3	4	5	6	7	8
第四趟	30	40	60	70	90	10	20	40

下面给出其算法的描述：

```
void  SeqList<T>::InsertSort( )
{
 int i, j ;
 for (i=2 ; i<=last ; i++)//依次将data[2],...data[last]插入序列中
  {
    data[0]=data[i];      // data[0]作为岗哨和暂存单元
    j=i-1;
    while ( data[0] < data[j] ) //查找data[i]的插入位置
       data[j+1]=data[j- -];// 将序列中排序码大于data[i]的记录后移
    data[j+1]=data[0] ;        // 填入data[i]
  } // for
} // InsertSort
```

直接插入排序——链式结构



算法设计

下面对该算法进行分析：

(1)空间复杂度：排序过程中只需要一个记录的附加空间，即data[0]。

(2)时间复杂度

（a）最好情况：原始序列中各记录已经按排序码递增的顺序排列。

比较次数=n-1

移动次数=2*(n-1)

（b）最坏情况：原始序列中各记录已经按排序码递减的顺序排序。

比较次数=2+3+...+n = (n-1)*(n+2)/2

移动次数=3+4+...+n+1 = (n-1)*(n+4)/2

（c）平均情况：若原始序列中各记录是随机的，即待排序序列中的记录可能出现的各种排列的概率相同。

比较次数=((n-1)*(n+2)/2 +n-1)/2 =(n2+n-2)/4

移动次数=((n-1)*(n+4)/2 +2*(n-1)) =(n2+7n-8)/4

因此，直接插入排序的时间复杂度为O(n*n).

(3)稳定性

直接插入排序是稳定的排序方法。因为具有同一排序码的后一记录必然插在具有同一排序码的前一个记录的后面，即相对次序保持不变。

发现问题

观察第7趟处理过程

例如，已知一组待排序记录的排序码分别为 40，30，60，90，70，10，20，40，用直接插入法进行排序的过程图 9-1 所示：

单元编号	1	2	3	4	5	6	7	8
初始状态	40	30	60	90	70	10	20	40
第一趟	30	40	60	90	70	10	20	40
第二趟	30	40	60	90	70	10	20	40
第三趟	30	40	60	90	70	10	20	40
第四趟	30	40	60	70	90	10	20	40
第五趟	10	30	40	60	70	90	20	40
第六趟	10	20	30	40	60	70	90	40
第七趟	10	20	30	40	40	60	70	90

说明：有阴影部分为当前已排序的记录

2.折半插入排序

- 基本思想：
- 设在顺序表中有一个对象序列data[1],data[2],…,data[*n*]。其中，data[1], *data*[2], …, data[*i*-1]是已经排好序的对象。在插入data[*i*]时，利用折半查找方法寻找data[*i*]的插入位置。

下面给出其算法的描述：

```
void SeqList<T>::Binary_InsertSort( )
{
int i, j ;
for (i=2 ; i<=last ; i++ ) // 依次将data[2],...data[last]插入序列中
 {
  data[0]=data[i];     // data[0]作为暂存单元
  low=1; high =i-1; // low,high分别作为折半查找的上、下界
  while ( low <=high ) //用折半方法查找data[i]的插入位置
  {
    mid=( low+high ) /2;
    if ( data[0] < data[mid] )
       high=mid-1;
    else
       low=mid+1;
  } // while
 for ( j=i-1 ; j>=low ; j-- )
     data[j+1]=data[j];  // 将序列中排序码大于data[i]的记录后移
 data[low]=data[0] ;    // 把暂存在data[0]的data[i]填入相应的位置中
} // for
} // Binary_InsertSort
```

下面对该算法进行分析：

(1) 空间复杂度

排序过程中也只需要一个比较记录的附加空间(data[0])，因此空间的占用量较少。

(2) 时间复杂度

(a) 比较次数：分析对比折半插入排序算法和直接插入排序算法，不难发现折半插入排序比直接插入排序减少了排序码间的比较次数。在折半插入排序中，为了插入第i个记录，最多只需比较logi次，所以总的比较次数最多为：

log2+ log3+.....+ logn = logn!<n*logn

因此，对有n个记录的序列进行折半插入排序所需要的时间数量级为O(nlogn)。

(b) 移动次数：折半插入排序与直接插入排序的记录移动次数是相同的，数量级为$O(n^2)$。

因此，折半插入排序算法的时间复杂度仍为$O(n^2)$。

(3) 稳定性

折半插入排序方法也是稳定的。

发现问题

- 对10 20 30 40 50 60 70 做直接插入排序

3.希尔排序

- 希尔排序（Shell Sort）又称缩小增量排序，是希尔（D.L.Shell）在1959年提出的一种改进的插入排序算法。

- 基本思想是：把记录按一定的增量分组，对每组用直接插入排序算法排序，随着增量的减少，各分组中包含的记录将越来越多，当增量减少至1时，所有记录序列变成一个组。由此可见，希尔排序也是巧妙地利用原有子序列的有序性。

- 具体做法：
- （1）分组 di
- （2）组内直接插入排序
- （3）缩小分组间隔di

- di有各种不同的取法，
- Shell ： d1=n/2 , di+1=di/2；
- Knuth: di+1=di-1/3；

例如，已知一组待排序记录的排序码分别为 40，30，60，90，70，10，20，40，用希尔排序法进行排序，取 $d_1=n/2=8/2=4$, $d_{i+1}=d_i/2$，排序过程如图 9-2 所示：

单元编号	1	2	3	4	5	6	7	8
初始态	40	30	60	90	70	10	20	40

第一次分组情况：

第一组：40 70	组内排序后有：	40 70
第二组：30 10	组内排序后有：	10 30
第三组：60 20	组内排序后有：	20 60
第四组：90 40	组内排序后有：	40 90

因此，第一趟希尔排序得到：

单元编号	1	2	3	4	5	6	7	8
第一趟	40	10	20	40	70	30	60	90

第二次分组情况：

第一组：40 20 70 60	组内排序后有：	20 40 60 70
第二组：10 40 30 90	组内排序后有：	10 30 40 90

因此，可依次得到剩余趟次希尔排序后的结果：

单元编号	1	2	3	4	5	6	7	8
第二趟	20	10	40	30	60	40	70	90
第三趟	10	20	30	40	40	60	70	90

说明：带同一阴影的记录表示为同一组中的记录

按d1=n/2 ,di+1=di/2来选择增量序列的希尔排序算法：

```
void SeqList<T>:: Shellsort( )
{ int i,j,k;
  k=last/2;
  while ( k>=1 )  // 直到增量值为1为止
  { for ( i=k+1;i<=last;i++)//分别对同组内的记录进行排序
    {
      data[0]=data[i];  // data[0]作为岗哨和暂存单元
      j=i-k;
      while ( data[0]<data[j] )
      { data[j+k]=data[j]; j=j-k; } // while
      data[j+k]=data[0];
    } // for
   k=k/2;
  }// while
}// Shellsort
```

算法分析

- 时间复杂度：
- 对特定的待排序对象序列，可以准确地估算排序码的比较次数和对象移动次数。但想要弄清排序码比较次数和对象移动次数与增量选择之间的依赖关系，并给出完整的数学分析，还没有人能够做到。
- Knuth利用大量实验统计资料得出：当 n 很大时，排序码平均比较次数和对象平均移动次数大约在 $n^{1.25}$ 到 $1.6n^{1.25}$ 的范围内。这是在利用直接插入排序作为子序列排序方法的情况下得到的。
- 稳定性：希尔排序是不稳定的。

三、交换排序

基本思想：

两两比较待排序对象的排序码,如果发生逆序(即排列顺序与排序后的次序正好相反)，则交换之。直到所有对象都排好序为止。

具体的实施方案有：

(1) 冒泡排序

(2) 快速排序

1.冒泡排序

- 基本方法：
- 设待排序对象序列中的对象个数为 n。最多作 $n-1$ 趟，$i = 1, 2, \ldots, n-1$。在第 i 趟中从前向后，$j = 1, 2, \ldots, n-i+1$，顺次两两比较data[j]和data[$j+1$]。如果发生逆序，则交换data[j]和data[$j+1$]。
- 方向可以是从前向后或从后向前。
- 下面以从前向后为例来进行说明.

例如，已知一组待排序记录的排序码分别为 40，30，60，90，70，10，20，40，用冒泡排序方法进行排序，其过程如图 9-7 所示。

单元编号	1	2	3	4	5	6	7	8
初始状态	40	30	60	90	70	10	20	40
第一趟	30	40	60	70	10	20	40	90
第二趟	30	40	60	10	20	40	70	90
第三趟	30	40	10	20	40	60	70	90
第四趟	30	10	20	40	40	60	70	90
第五趟	10	20	30	40	40	60	70	90
第六趟	10	20	30	40	40	60	70	90
第七趟	10	20	30	40	40	60	70	90

冒泡排序算法描述如下：

```
void SeqList<T>:: BubbleSort( )
{ int i, j ;

 for (i=1; i<last ; i++ ) // 共做last-1趟排序
 {
  for ( j= 1 ; j <=last-i ; j++ )
  {
   if (data[j]>data[j+1])//data[j]与data[j+1]不符合排序要求则交换
    {
     data[0]=data[j]; // 进行交换，data[0]作为记录交换的暂存单元
     data[j]=data[j+1];  data[j+1]=data[0];
    } // if
  } // for
 }// for
} // Bubble Sort
```

发现问题

- 对10 20 90 30 40 50 60 做冒泡排序

冒泡排序改进算法描述如下：

```
void SeqList<T>:: BubbleSort( )
{ int i, j ;

 for (i=1; i<last ; i++ ) // 共做last-1趟排序
 {
  flag=1; // 设置交换标志，1表示无交换，0表示有交换
  for ( j= 1 ; j <=last-i ; j++ ) {
   if (data[j]>data[j+1])//data[j]与data[j+1]不符合排序要求则交换
   {
     flag=0 ; // 有交换存在
     data[0]=data[j]; // 进行交换，data[0]作为记录交换的暂存单元
     data[j]=data[j+1];  data[j+1]=data[0];
    } // if
   } // for
  if ( flag==1 ) break;//表示在该趟排序中不存在交换，排序结束
 }// for
} // Bubble Sort
```

下面对冒泡排序算法进行分析：

(1)空间复杂度

在冒泡排序过程中也只需要一个记录大小的辅助存储空间，作为交换记录的暂存单元。

(2)时间复杂度

（a）最好情况：原始序列中各记录已按排序码递增的顺序排列（即正序）的情况下：记录的比较次数为n-1次，记录的移动次数为0。

（b）最坏情况：原始序列中各记录已按排序码递减的顺序排列（即逆序）的情况下：

比较次数= (n-1)+(n-2)+ ...+(1)=n*(n-1)/2

移动次数=3*n*(n-1)/2

(c)平均情况：若原始序列中各记录是随机的，即待排序序列中的记录可能出现的各种排列的概率相同。

比较次数= (n*(n-1)/2+(n-1))/2 = (n2+n-2)/4

移动次数=(3*n*(n-1)/2 + 0)/2 = (3n2-3n)/4

因此，冒泡排序算法的时间复杂度为O(n*n)

(3)稳定性：冒泡排序方法是稳定的。

发现问题

- 对20 30 90 40 50 60 10 做冒泡排序

改进策略

- 双向冒泡

发现问题

例如，已知一组待排序记录的排序码分别为 40，30，60，90，70，10，20，40，用冒泡排序方法进行排序，其过程如图 9-7 所示。

单元编号	1	2	3	4	5	6	7	8
初始状态	40	30	60	90	70	10	20	40
第一趟	30	40	60	70	10	20	40	90
第二趟	30	40	60	10	20	40	70	90
第三趟	30	40	10	20	40	60	70	90
第四趟	30	10	20	40	40	60	70	90
第五趟	10	20	30	40	40	60	70	90
第六趟	10	20	30	40	40	60	70	90
第七趟	10	20	30	40	40	60	70	90

- 40 30 60 90 70 10 20 40

快速排序

2. 快速排序

- 基本思想：
- 是任取待排序对象序列中的某个对象（例如取第一个对象）作为基准，按照该对象的排序码大小，将整个对象序列划分为左右两个子序列：
  - 左侧子序列中所有对象的排序码都小于或等于基准对象的排序码；
  - 右侧子序列中所有对象的排序码都大于基准对象的排序码；
- 基准对象则排在这两个子序列中间（这也是该对象最终应安放的位置）。
- 然后分别对这两个子序列重复施行上述方法，直到所有的对象都排在相应位置上为止。

例如：已知一组待排序记录的排序码分别为 40，30，60，90，70，10，20，40，用快速排序方法进行排序，完成以第一个记录作为基准第一趟排序过程如图 9-8 所示：

单元编号	0	1	2	3	4	5	6	7	8
初始状态	40	40	30	60	90	70	10	20	40
		↑down							↑up
up 向左移动	40	40	30	60	90	70	10	20	40
		↑down						↑up	
第一次交换后	40	20	30	60	90	70	10	20	40
			↑down					↑up	
down 向右移动	40	20	30	60	90	70	10	20	40
				↑down				↑up	
第二次交换后	40	20	30	60	90	70	10	60	40
				↑down			↑up		
第三次交换后	40	20	30	10	90	70	10	60	40
					↑down		↑up		
第四次交换后	40	20	30	10	90	70	90	60	40
					↑down	↑up			
up 向左移动	40	20	30	10	90	70	90	60	40
					↑down ↑up				
填入基准记录	40	20	30	10	40	70	90	60	40
					↑down ↑up				

说明：带阴影单元为当前已被腾空出来的单元。

根据上述一次快速排序的算法思想，其算法描述如下：

```
int SeqList<T>::QuickPass (int low,int high )
//对子序列data[low]到data[high] 作一趟的快速排序
{  int  down , up ;
   down=low ;  up=high ;   //  分别对位置指示器作初始化
   data[0]= data[low] ;     // 把基准记录暂存在data[0]中
   while ( down < up )
   {
     while((down<up)&&(data[up]>=data[0])) //从右向左扫描
          up- - ;
     if(down<up)//在被腾空出来的单元(由down来指向)中填入data[up]
        data[down++]=data[up] ;   // 记录，并将down往右移动一个单元
     while ((down<up)&&(data[down]<= data[0]))// 从左向右扫描
          down ++;
     if(down<up)// 在被腾空出来的单元(由up来指向)中填入data[down]
          data[up- -]=data[down] ;   // 记录，并将up往左移动一个单元
   }  // while
  data[down]=data[0];   //  在正确的位置上填入基准记录
  return  down ;   //  返回当前填入基准记录的所在位置
}  // QuickPass
```

例如，已知一组待排序记录的排序码分别为 40，30，60，90，70，10，20，40，用快速排序方法进行排序，每一趟的排序结果如图 9-9 所示：

存储单元	1	2	3	4	5	6	7	8
初始态	40	30	60	90	70	10	20	40
第一趟	20	30	10	40	70	90	60	40
第二趟	10	20	30	40	70	90	60	40
第三趟	10	20	30	40	40	60	70	90
第四趟	10	20	30	40	40	60	70	90

说明：带阴影单元的位置为该趟结束后基准记录的最终排序位置。

递归形式的快速排序算法如下：

```
void SeqList<T>:: QuickSort (int  low , int  high )
 // 对序列data[low]到 data[high]作快速排序
{
  int  mid ; // mid 为一趟快速排序后，基准记录所在位置指示器

  if(low<high)//当序列中的记录个数为1个或无记录时无须排序
  {
    mid=QuickPass ( R, low, high );
    // 对序列data[low]到data[high]作一趟快速排序，
    QuickSort(R, low , mid-1); //对左半部分作递归处理
    QuickSort(R, mid+1, high); //对右半部分作递归处理
  }  // if
}  // QuickSort
```

算法分析

- 时间复杂度：
- 如果每次划分对一个对象定位后，该对象的左侧子序列与右侧子序列的长度相同，则下一步将是对两个长度减半的子序列进行排序，这是最理想的情况。
- 在n个元素的序列中，对一个对象定位所需时间为$O(n)$。若设$T(n)$是对n个元素的序列进行排序所需的时间，而且每次对一个对象正确定位后，正好把序列划分为长度相等的两个子序列，此时，总的计算时间为$O(n\log_2 n)$.
- 实验结果表明：就平均计算时间而言，快速排序是所有内排序方法中最好的一个。

- 在最坏的情况，即待排序对象序列已经按其排序码从小到大排好序的情况下，其递归树成为单支树，每次划分只得到一个比上一次少一个对象的子序列。必须经过$n-1$趟才能把所有对象定位，而且第i趟需要经过$n-i$次排序码比较才能找到第i个对象的安放位置，总的排序码比较次数将达到：

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

- 此时，排序速度退化到简单排序的水平，比直接插入排序还慢。

- 空间复杂度：
- 快速排序是递归的，需要有一个栈存放每层递归调用时的指针和参数.最大递归调用层次数与递归树的高度一致，理想情况为$\lceil\log_2(n+1)\rceil$。因此，要求存储开销为$O(\log_2 n)$。
- 在最坏的情况下，所占用的附加存储(栈)将达到$O(n)$。
- 稳定性：快速排序是一种不稳定的排序方法。
- 结论：对于n较大的平均情况而言，快速排序是“快速”的，但是当n很小时，这种排序方法往往比其它简单排序方法还要慢。

四、选择排序

基本思想：

每一趟 (例如第 i趟, $i=1,2,\cdots,n-1$) 在后面 $n-i+1$个待排序对象中选出排序码最小或最大的对象,作为有序对象序列的第i个对象。待到第$n-1$趟作完，待排序对象只剩下1个,就不用再选了。

具体的实施方案有：

(1) 直接选择排序

(2) 锦标赛排序

(3) 堆排序

1.直接选择排序

- 直接选择排序是一种简单的排序方法,它的基本步骤是：

①在一组对象 data[i]～data[n] 中选择具有最小排序码的对象；

②若它不是这组对象中的第一个对象，则将它与这组对象中的第一个对象交换；

③在这组对象中剔除这个具有最小排序码的对象。在剩下的对象data[i+1]～data[n]中重复执行第①、②步，直到剩余对象只有一个为止。

直接选择排序

- 40 30 60 90 70 10 20 40

例如，已知一组待排序记录的排序码分别为 40，30，60，90，70，10，20，40，用直接选择排序法进行排序，其过程如图 9-3 所示：

单元编号	1	2	3	4	5	6	7	8
初始态	40	30	60	90	70	10	20	40
第一趟	10	30	60	90	70	40	20	40
第二趟	10	20	60	90	70	40	30	40
第三趟	10	20	30	90	70	40	60	40
第四趟	10	20	30	40	70	90	60	40
第五趟	10	20	30	40	40	90	60	70
第六趟	10	20	30	40	40	60	90	70
第七趟	10	20	30	40	40	60	70	90

说明：有同一行上的两个带阴影的单元表示要作记录交换的两个单元

下面给出直接选择排序算法的描述：

```
void SeqList<T>:: Selectsort( )
{  int i,j,pos;
   for ( i=1 ; i<last ; i++ )   // 共作n-1趟选择排序
  {
    pos=i;//pos用来记录当前排序码最小值的记录所在的位置
    for(j=i+1;j<=last;j++ )//在无序表中找出排序码最小的记录
      if ( data[j] <data[pos] )
          pos = j ;
    data[0]=data[i];   // data[0]作为交换的暂存单元
    data[i]=data[pos];
    data[pos]=data[0];
  } // for
} //  Selectsort
```

发现问题

- 对30 20 10 60 50 40 做直接选择排序

下面给出直接选择改进排序算法的描述：

```
void SeqList<T>:: Selectsort( )
{  int i,j,pos;
   for ( i=1 ; i<last ; i++ )   // 共作n-1趟选择排序
  {
    pos=i;//pos用来记录当前排序码最小值的记录所在的位置
    for(j=i+1;j<=last;j++ )//在无序表中找出排序码最小的记录
      if ( data[j] <data[pos] )
          pos = j ;
    if ( pos !=i )
    {
      data[0]=data[i];   // data[0]作为交换的暂存单元
      data[i]=data[pos];
      data[pos]=data[0];
    } // if
  } // for
} //  Selectsort
```

算法分析

- 时间复杂度：
- 直接选择排序的排序码比较次数与对象的初始排列无关。
- 设整个待排序对象序列有 n 个对象，则第 i 趟选择具有最小排序码对象所需的比较次数总是 $n-i-1$ 次。因此，总的排序码比较次数为：

$$KCN = \sum_{i=0}^{n-2}(n-i-1) = \frac{n(n-1)}{2}$$

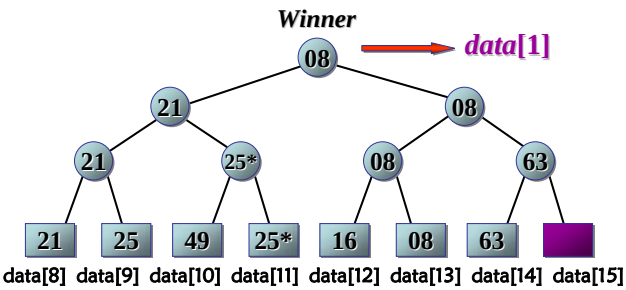
- 对象的移动次数与对象序列的初始排列有关。当这组对象的初始状态是按其排序码从小到大有序的时候，对象的移动次数＝ 0，达到最少。
- 最坏情况是每一趟都要进行交换，总的对象移动次数为 ＝ $3(n-1)$。
- 稳定性：直接选择排序是一种不稳定的排序方法。

• 问题分析

- 直接选择排序有何缺陷？
- 每次从n-i+1个记录中选一个排序码最小的记录，从而需要进行n-i次比较(其中i=1,2,...,n)当n充分大时，其效率很低。
- 如何提高排序速度？

锦标赛排序 (Tournament Tree Sort)

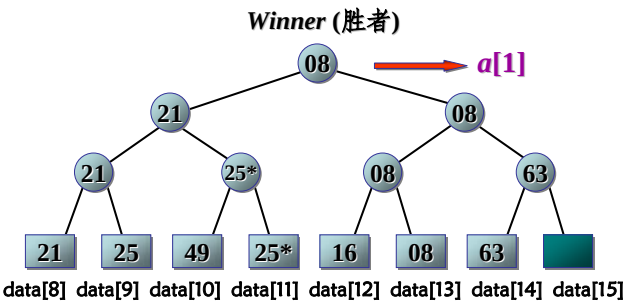
- 它的思想与体育比赛时的淘汰赛类似。首先取得 n 个对象的关键码，进行两两比较，得到 $\lceil n/2 \rceil$ 个比较的优胜者(关键码小者)，作为第一步比较的结果保留下来。然后对这 $\lceil n/2 \rceil$ 个对象再进行关键码的两两比较，...，如此重复，直到选出一个关键码最小的对象为止。



在图例中，最下面是对象排列的初始状态，相当于一棵满二叉树的叶结点，它存放的是所有参加排序的对象的关键码。

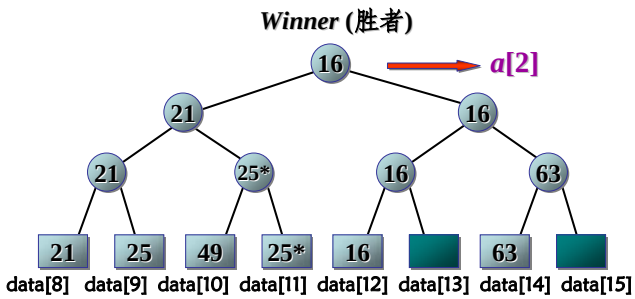
胜者树

- 每次两两比较的结果是把关键码小者作为优胜者上升到双亲结点，称这种比赛树为胜者树。



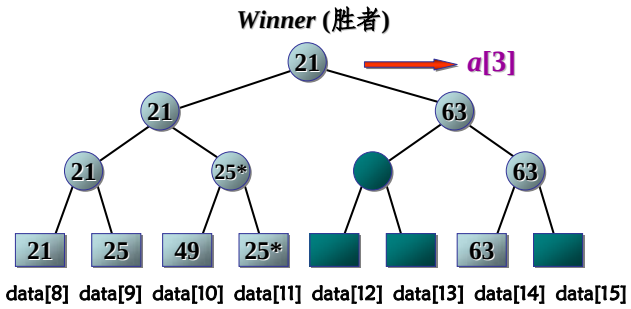
形成初始胜者树（最小关键码上升到根）

关键码比较次数：6



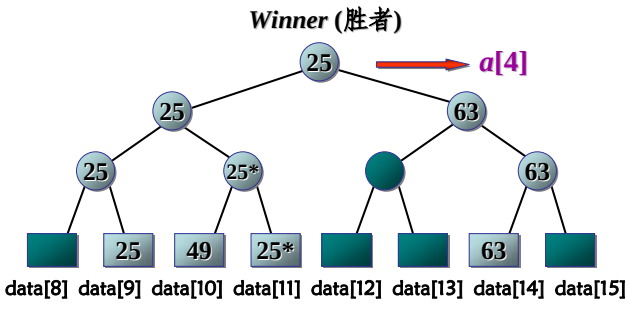
输出冠军并调整胜者树后树的状态

关键码比较次数：2



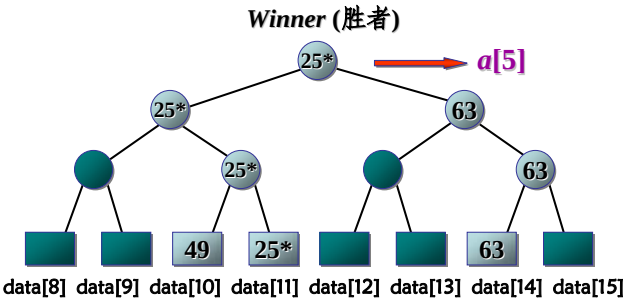
输出亚军并调整胜者树后树的状态

关键码比较次数：2



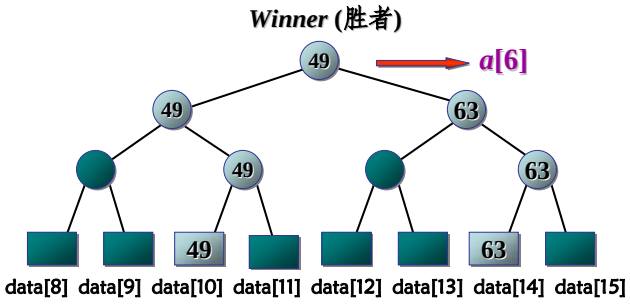
输出第三名并调整胜者树后树的状态

关键码比较次数：2



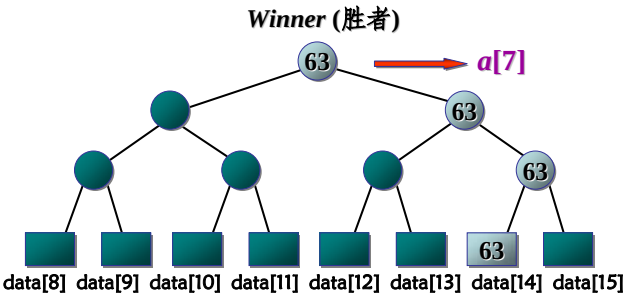
输出第四名并调整胜者树后树的状态

关键码比较次数：2



输出第四名并调整胜者树后树的状态

关键码比较次数：2



全部比赛结果输出时树的状态

关键码比较次数：2

锦标赛排序算法分析

- 时间
- 空间

发现问题

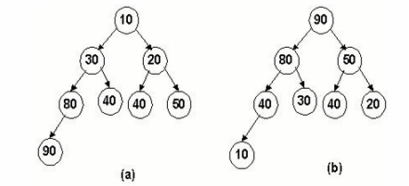
- 直接选择排序有何缺陷？
- 每次从n-i+1个记录中选一个排序码最小的记录，从而需要进行n-i次比较（其中i=1,2,...,n）当n充分大时，其效率很低。
- 是否能利用以前的比较结果来选择后一个值，则可提高排序速度。

完全二叉树结构

- 问题：如何保存这个比较结果？

2.堆排序

- 堆的定义：设有n个数据元素组成的序列$\{a_1, a_2, \ldots, a_n\}$，若它满足下面的条件：
- （1）这些数据元素是一棵完全二叉树中的结点，且a_i(i=1,2,...n)是该完全二叉树中编号为i的结点；
- （2）若$2i \le n$，有$a_{2i} \ge a_i$；
- （3）若$2i+1 \le n$，有$a_{2i+1} \ge a_i$；
- 则称该序列为一个堆。



小根堆　　大根堆

堆排序

- 40 30 60 90 70 10 20 40

堆排序

- 把待排序元素依次插入初始化为空的小根堆中。
- 从堆中依次取出(删除)根结点值，放入排序序列中，直到堆为空为止。

堆排序

- 基本思想：
- （1）如何将待排序序列构成为一个初始堆。（即如何挑选一个最小值出来，但又需要保存每趟的比较结果）
- （2）在输出堆顶记录之后，如何将剩余记录重新调整为一个新堆。

堆知识　回顾

最小堆的类定义

```
#define  DefaultSize = 10
template <class T>
class MinHeap
{    T *heap;
     int CurrentSize;
     int MaxHeapSize;
 public:
    MinHeap ( int sz=DefaultSize );  // 初始化为空堆
    MinHeap ( T arr[ ], int n );
    ~MinHeap ( ) { delete [ ] heap; }
    const MinHeap<T> & operator =
        ( const MinHeap &R );
```

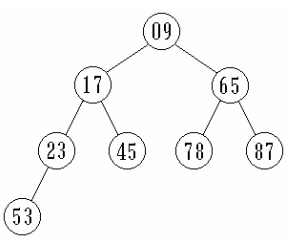
```
  bool  Insert ( const T &x );  // 插入操作
  bool  RemoveMin ( T &x );  // 删除操作
  bool  IsEmpty ( ) const
        { return CurrentSize == 0; }
  bool IsFull ( ) const
        { return CurrentSize == MaxHeapSize; }
  void MakeEmpty ( ) { CurrentSize = 0; }
private:
  void siftDown ( int start, int m );
  void siftUp ( int start );
}
```

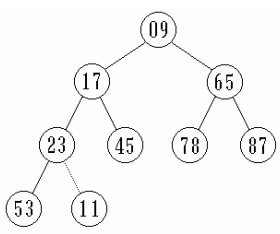
初始化操作→生成空堆

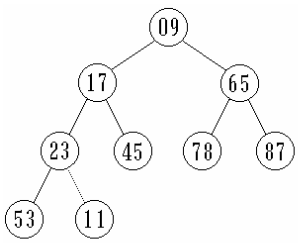
```
template <class T>
MinHeap <T> ::MinHeap ( int sz )
{
  //根据给定大小maxSize,建立堆对象
   MaxHeapSize=DefaultSize<sz ?maxSize :DefaultSize ;//确定堆大小
  heap = new T[MaxHeapSize]; //创建堆空间
  CurrentSize = 0;                    //初始化
}
```

小根堆的插入操作

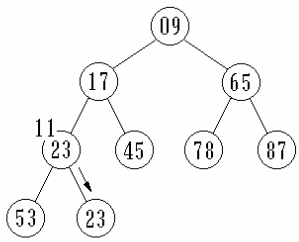
- 插入11



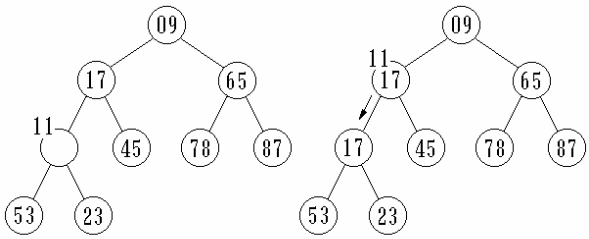




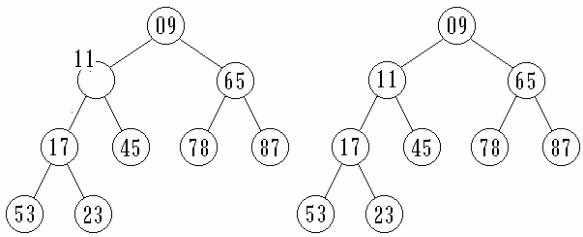
(a) 初始 尾部加11



(b) 双亲关键码23下降



(c) 双亲关键码17下降



(d) 11回填 调整完成

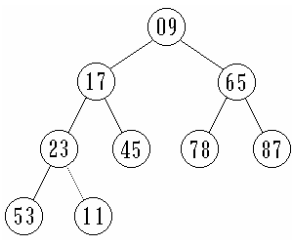
操作的归纳

- 向上调整的操作

void siftUp (int start);

最小堆的向上调整算法

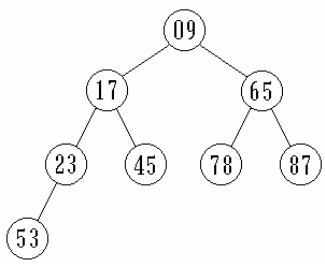
```
template <class T>
void MinHeap<T> ::siftUp ( int start )
{    //从 start 开始,向上直到0,调整堆
   int j = start, i = (j-1)/2;   // i 是 j 的双亲
   T temp = heap[j];
   while ( j > 0 )
   {  if( heap[i]<=temp) break;
      else
      { heap[j] = heap[i];
        j = i; i = (i -1)/2;
      }
   }
   heap[j] = temp;
}
```

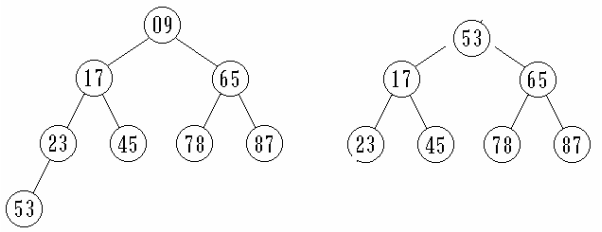


堆插入算法

```
template <class T>
bool MinHeap<T> ::Insert ( const T &x )
{ //在堆中插入新元素 x
   if ( CurrentSize == MaxHeapSize ) //堆满
    { cout << "堆已满" << endl; return 0; }
   heap[CurrentSize] = x;        //插在表尾
   siftUp (CurrentSize);        //向上调整为堆
   CurrentSize++;                //堆元素增一
   return 1;
}
```

小根堆的删除操作





操作的归纳

- 向下调整的操作

void siftDown (int start, int m);

结点向下调整的代码

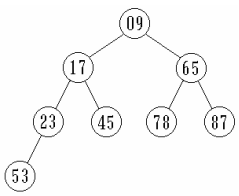
```
template <class T>
void MinHeap<T> :: siftDown ( int start, int m )
{
   int i = start,  j = 2*i+1;      //j 是 i 的左子女
   T temp = heap[i];
   while ( j <= m )
   {
      if ( j < m && heap[j] >heap[j+1] )
            j++;   //两子女中选小者
      if ( temp <= heap[j] ) break;
      else { heap[i] = heap[j]; i = j ;  j = 2*j+1; }
   }
   heap[i] = temp;
}
```

最小堆的删除算法

```
template <class T>
int MinHeap <T> ::RemoveMin ( T &x )
{
   if ( !CurrentSize )
    {   cout << " 堆已空 " << endl; return 0;  }
   x = heap[0];          //最小元素出队列
   heap[0] = heap[CurrentSize-1];
   CurrentSize--;      //用最小元素填补
   siftDown ( 0, CurrentSize-1 );
   //从0号位置开始自顶向下调整为堆
   return 1;
}
```

堆排序

- 把待排序元素依次插入初始化为空的小根堆中。
- 从堆中依次取出(删除)根结点值，放入排序序列中，直到堆为空。

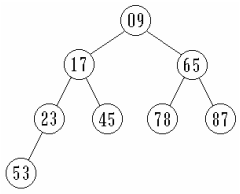


最小堆的类定义

```
Template<class T>
Void SeqList<T>::HeapSort(void)
{ int i ; T x;
  MinHeap<T>  MyHeap;
  for(i=1;i<=last;i++)   // 把待排序元素插入堆中
     MyHeap.Insert(data[i]);
 i=1;
 while ( ! MyHeap.IsEmpty( ) )  // 堆非空
 {    MyHeap.Remove(x); // 从堆中删除根结点
      data[i]=x;
      i++;
  }
}
```

算法分析

- 时间复杂度
- （1）堆插入
- （2）堆删除



算法分析

(1) 时间复杂度

(a) 比较次数

由于堆排序过程分为建立初始堆和对剩余记录重建堆两个部分。因此，记录排序码的比较次数应为这两部分比较次数之和。

对于具有n个记录的完全二叉树深度是$h=\lfloor \log_2 n \rfloor+1$。在建立初始堆时，对于每个非叶子都要自上而下作“筛选”建堆，由于在二叉树中第i层上的结点个数至多为2^{i-1}，且第i层上的结点的最大下移深度为h-i，根据算法siftdown可知，每下移一层要作两次比较，即每一个位于第i层上结点所要进行的比较次数至多为2*(h-i)。由于在堆中每个非叶子结点所处的层数只可能为h-1, h-2, . . . 1，则它们作下移时，最大的下移层数分别为1, 2, . . . , h-1,

因此，建立初始堆所要进行的比较总次数满足：

总的比较次数 $\leq 2^0*2*(h-1)+2^1*2*(h-2)+\ldots+2^{h-2}*2*(1)$

$\leq 4*(n-\log_2^{n-1})$

对于堆排序过程中，总共要进行n-1次重建堆，每次重建堆都将根结点“下沉”到合适的位置。现考虑第j次重建堆，堆中有n-j个结点，此时二叉树的深度为$\lfloor \log_2^{n-j} \rfloor+1$，对于根结点最大的下移层数为$\lfloor \log_2^{n-j} \rfloor$，而每下移一层要作两次比较，则所需的比较次数至多为$2*\lfloor \log_2^{n-j} \rfloor$。因此，n-1趟排序过程中重建堆的比较总次数应满足：

总的比较次数

$\leq 2*\lfloor \log_2^{n-1} \rfloor+2*\lfloor \log_2^{n-2} \rfloor+\ldots+2*\lfloor \log_2^{2} \rfloor+2*\lfloor \log_2^{1} \rfloor$

$< 2*n*\lfloor \log_2^{n} \rfloor$

因此，用堆排序进行排序时，总的比较次数为$O(n+n\log^n)$，即$O(n\log^n)$。

(b) 移动次数

用类似的分析方法可以得到，堆排序中总的移动次数也是$O(n\log^n)$。

由此可见，堆排序在最坏和平均的情况下，其时间复杂度均为O(nlogn)。

由于建立初始堆所需的比较次数较多，所以堆排序对记录数较少的序列并不值得提倡，但对于数目较大的序列来说还是很有效的。

(2) 空间复杂度

堆排序过程中仅需要一个记录大小的辅助存储空间，以作为两个记录交换的暂存单元。

(3) 稳定性

堆排序是不稳定的。

五、归并排序

- 归并：是将两个或两个以上的有序表合并成一个新的有序表。
- 两路归并

例如，已知一组待排序记录的排序码分别为 40，30，60，90，70，10，20，40，则进行二路归并排序的过程如图 9-10 所示。

存储单元	1	2	3	4	5	6	7	8
初始态	40	30	60	90	70	10	20	40
第一趟	30	40	60	90	10	70	20	40
第二趟	30	40	60	90	10	20	40	70
第三趟	10	20	30	40	40	60	70	90

说明：带同一阴影的相邻单元中的记录表示为同一个有序子序列中的记录

算法分析

(1) 空间复杂度

由于在算法中需要利用与存放待排序记录的数组R一样大小的一个辅助数组T，所以其空间复杂度为O(n)。可见它明显高于前面所讨论的所有排序算法的空间复杂度。

(2) 时间复杂度

通过分析二路归并排序算法可知，其时间复杂度应等于归并趟数与每一趟的时间复杂度的乘积。

在二路归并排序算法中，第一趟归并将记录个数为1的子序列进行合并；第二趟归并将记录个数为2的子序列进行合并；第i 趟归并将记录个数为2i的子序列进行合并；因此，对于具有n个记录的待排序序列来说，要完成归并排序总共需要进行 ⌈ logn ⌉ 趟归并。

对于每一趟归并就是将两个两个的有序子序列进行归并，而每一对有序子序列归并时，记录的比较次数和移动次数（指的是从一个数组复制到另一个数组的记录个数）均等于这一对有序子序列的记录个数之和，所以每一趟归并的比较次数和移动次数均等于待排序序列中的记录个数n ，显然每一趟归并的时间复杂度为O(n)。

因此，二路归并排序的时间复杂度为O(n*logn)。

(3) 稳定性：二路归并排序是稳定的。

六、分配排序

问题1：排序时能否不比较也可以调整位置？

如何不比较将一位数字5，9，7，3进行排序

```
cin>>x;
data[x]=x;
```

问题2：如果待排序的排序码有重复值，则如何解决呢？如数字5，9，7，9，3?

分配排序方法

是按某种方法将记录分配，然后再按另一种方法将记录收集，如此反复进行分配与收集，直到将所有记录排好序为止。

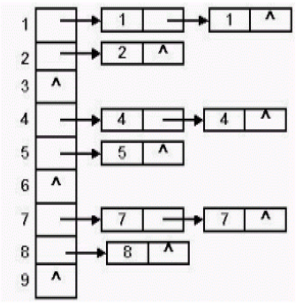
具体的实施方案有：

(1) 桶排序

(2) 基数排序

桶排序

例如：已知一组待排序记录的排序码分别为8，4，4，7，5，1，2，1，7，则进行桶排序的过程如图 9-11 所示。



下面给出桶排序的算法：

```
void SeqList<T>:: Bucket_Sort(LinkNode *B[])
{   int  i ;

   for ( i=1 ; i<=last ; i++ )
   //用尾插入的方法将待排序记录分别装入相应的桶中
   B[data[i]].InsertTail ( data[i] );
   for ( i=1 ; i< key ; i++ )
    // 将所有的桶首尾连接相连
   B[i].tail=B[i+1].head ;
}  // Bucket_sort
```

算法分析

(1) 空间复杂度

该排序算法是利用key个桶来完成的，而所有桶中的记录总数刚好为n个，所以该算法所用的辅助空间为key+n，即其空间复杂度为O(key+n)。如果key与n的数量级相同，则此时算法的时间复杂度为O(n)。

(2) 时间复杂度

由于算法中首先要将n个待排序的记录依次装入到相应的桶中，而在实现桶间的连接时共做key-1次首尾指针相连的操作，所以在整个桶排序共用n+key-1，即其时间复杂度为O(n+key-1)。如果key与n的数量级相同，则此时算法的时间复杂度为O(n)。

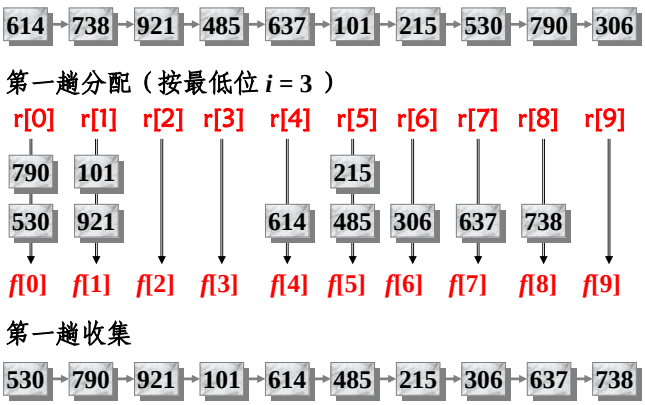
(3) 稳定性：桶排序是稳定的。

基数排序

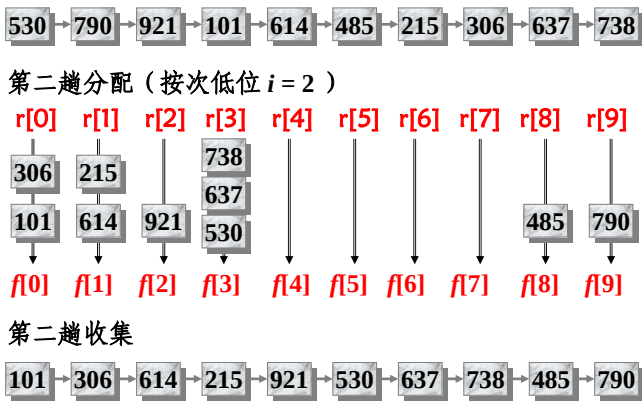
- 桶排序的缺陷？多位数怎办？
- 基数排序正是一个尽量利用减少桶的数目来缩短排序时间的分配排序方法。

- 基数排序的基本思想：
- 初始化：以基数安排相应个数的桶。
- 排序过程（从低位向高位做如下的工作）
- 分配：
- 收集：

基数排序的“分配”与“收集”过程 第一趟



基数排序的“分配”与“收集”过程 第二趟



基数排序的“分配”与“收集”过程 第三趟

算法分析

- (1) 时间复杂度

在该算法中，没有进行排序码的比较和记录的移动，只是进行指针赋值，因此，算法的执行时间主要耗费在指针的修改上。开始时分配出d个桶并初始化为空链所许的时间为O(n)。对于具有n个记录的待排序序列，执行一趟分配和收集所需的时间为O(n+d)。如果每个记录的排序码都有m位，则算法总共需要进行d趟的排序，所以链式基数排序算法的时间复杂度为O(m*(n+d))。显然，若m是常数，d是常数或不大于O(n)，则基数排序的时间复杂度都为O(n)。因此当n较小，m较大时，基数排序并不实用。只有当n较大、m较小时，基数排序才特别有效。

- (2) 空间复杂度

  该排序算法中增加了一个容量为d的辅助数组B来作为编号从0到d-1的d个桶，桶中的每一个记录均增加了一个指针域，而所有桶的记录总数刚好为n个，故算法的辅助存储空间开销为O(n+d)。

- (3) 稳定性：基数排序是稳定的。

各种排序方法的比较

排序方法	平均情况	最坏情况	辅助空间	稳定性
直接插入排序	$O(n^2)$	$O(n^2)$	O(1)	稳定
折半插入排序	$O(n^2)$	$O(n^2)$	O(1)	稳定
希尔排序	$O(n^{1.3})$	$O(n^{1.3})$	O(1)	不稳定
直接选择排序	$O(n^2)$	$O(n^2)$	O(1)	不稳定
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	O(1)	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	O(1)	稳定
快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)\sim O(n)$	不稳定
归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	O(n)	稳定
桶排序	O(n)	O(n)	O(n)	稳定
基数排序	O(n)	O(m*(n+d))	O(n)	稳定

结 论

- (1) 当待排序序列中的记录数n较大，记录排序码分布较随机，且要求排序不稳定时，则采用快速排序方法为宜。
- (2) 当待排序序列中的记录数n较大，有足够的内存空间，且要求排序稳定时，则采用归并排序方法为宜。
- (3) 当待排序序列中的记录数n较大，记录排序码分布可能会出现正序或逆序的情况，且对稳定性不作要求时，则采用堆排序（或归并排序）方法为宜。
- (4) 当待排序序列中的记录数n较小（如n≤100），记录排序码分布可能会出现正序或分布较随机，且要求排序稳定时，则采用直接插入排序为宜。
- (5) 当待排序序列中的记录数n较小（如n≤100），对稳定性不作要求时，则采用直接选择排序为宜。