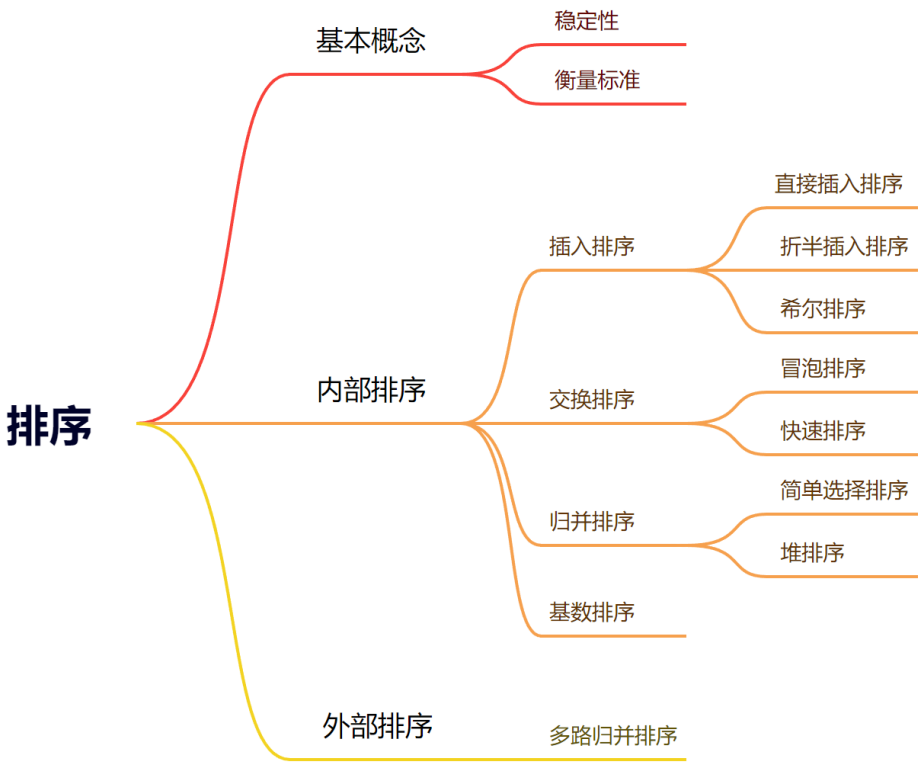
第八章 排序



1.排序的基本概念

排序:重新排列表中的元素,使表中的元素满足按关键字有序的过程

算法的稳定性:

1. **稳定的**:假设某个排序表中,有两个相同的关键字A和B(A在B的前面),使用某一排序算法后,A仍然在B的前面
2. **不稳定的:**假设某个排序表中,有两个相同的关键字A和B(A在B的前面),使用某一排序算法后,B在A的前面

例如:在一组数据 1 2 5 3 1 4中在经过某个排序算法排序后,前面的1的仍然是在后面的1前面则说明这个算法是稳定的,反之

排序算法的分类(看数据元素是否完全在内存中):

1. 内部排序:排序期间元素全部存放在内存中的排序
2. 外部排序:排序期间元素无法全部同时存放在内存中,必须在排序的过程中要求不断地在内外存之间移动的排序

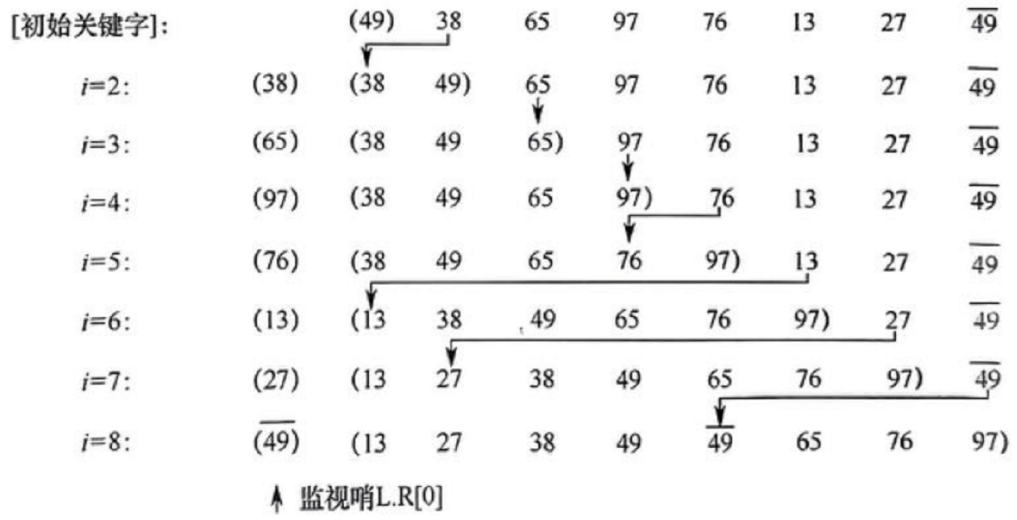
注:内部排序算法一般有两个操作:比较和移动,大部分内部排序算法只适用于顺序存储的线性表

2.插入排序

思想:每次将一个待排序的记录按关键字大小插入前面已排好序的子序列

2.1直接插入排序

将一个待排序的记录按关键字大小插入前面已排好序的子序列




```
void InsertSort(ElemType A[],int n){
    int i,j;
    for(i=2;i<=n;i++){//依次查找关键字,从2下标开始
        if(A[i]<A[i-1]){//若关键字小于前驱时
            A[0]=A[i];//记录下关键字
            for(j=i-1;A[0]<A[j];j--){//将关键字插入的位置到原关键字的位置向后移动一个
                A[j+1]=A[j];//移动
            }
            A[j+1]=A[0];//将空出的位置的关键字补上
        }
    }
}
```

空间复杂度O(1)

时间复杂度

$$O(n^2)$$

最好的情况,表中元素已经有序,不需要移动元素,只需要比较n次,O(n)

最坏的情况,表中元素正好逆序

适用性:顺序存储和链式存储的线性表

2.2折半插入排序

将一个待排序的记录按关键字大小插入前面已排好序的子序列,但查找插入的位置利用折半查找(因为前面的子序列有序所以可以使用折半查找)

```
void InsertSort(ElemType A[],int n){
    int i,j,low,high,mid;
    for(i=2;i<=n;i++){
        A[0]=A[i];
        low=1;
        high=i-1;
        while(low<=high){
            mid=(low+high)/2;
            if(A[mid]>A[0]) high=mid-1;
            else low=mid+1;
        }
        for(j=i-1;j>=high+1;j--){
            A[j+1]=A[j];
        }
        A[j+1]=A[0];
    }
}
```

时间复杂度仍然为

$$O(n^2)$$

但减少了比较次数(折半查找的时间复杂度),对于数据量不是很大的排序表,折半插入排序会表现出更好的性能

2.3希尔排序

又上述可知插入排序当表基本有序时,时间效率会有显著提高,因此我们可以基于这样的思想完善插入排序得到希尔排序

希尔排序的思想:将排列表分割成若干个子表,然后对各个子表进行直接插入排序

```
void ShellSort(ElemType A[],int n){
    int dk,i,j;
    for(dk=n/2;dk>=1;dk=dk/2)//增量变化(无同一规定)
        for(i=dk+1;i<=dk;i++)
            if(A[i]<A[i-dk]){
                A[0]=A[i];
                for(j=i-dk;j>0&&A[0]<A[j];j-=dk)
                    A[j+dk]=A[j];
                A[j+dk]=A[0];
            }
}
```

空间效率为:O(1)

时间效率:平均下来

$$O(n^{1.3})$$

最差

$$O(n^2)$$

稳定性:不稳定

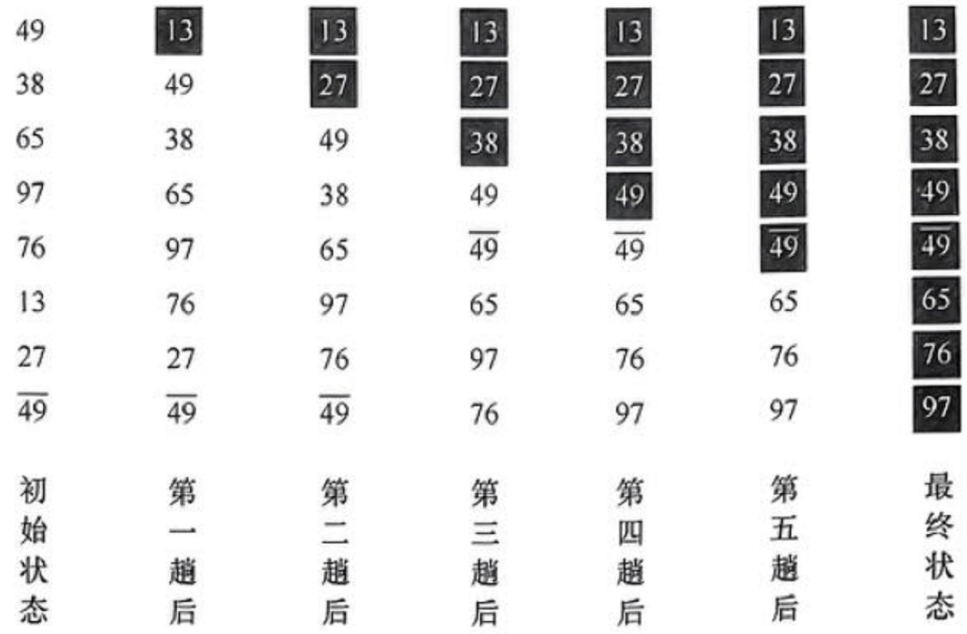
适用性:仅适用于线性表为顺序存储的情况

3.交换排序

交换:将两个元素关键字的比较结果来对换这两个记录在序列中的位置

3.1冒泡排序

思想:从后往前(或从前往后)两两比较相邻元素的值,若为逆序则交换它们




```
void BubbleSort(ElemType A[],int n){
    for(int i=0;i<n-1;i++){
        bool flag=false;//表示本次冒泡是否发生交换的标志
        for(int j=n-1;j>i;j--){//一趟冒泡排序
            if(A[j-1]>A[j]){//如果相邻元素为逆序
                swap(A[j-1],A[j]);//交换
                flag=true;
            }
        }
        if(flag==false) return;//若没有发生交换则说明已经有序
    }
}
```

空间效率:O(1)

时间效率:

$$O(n^2)$$

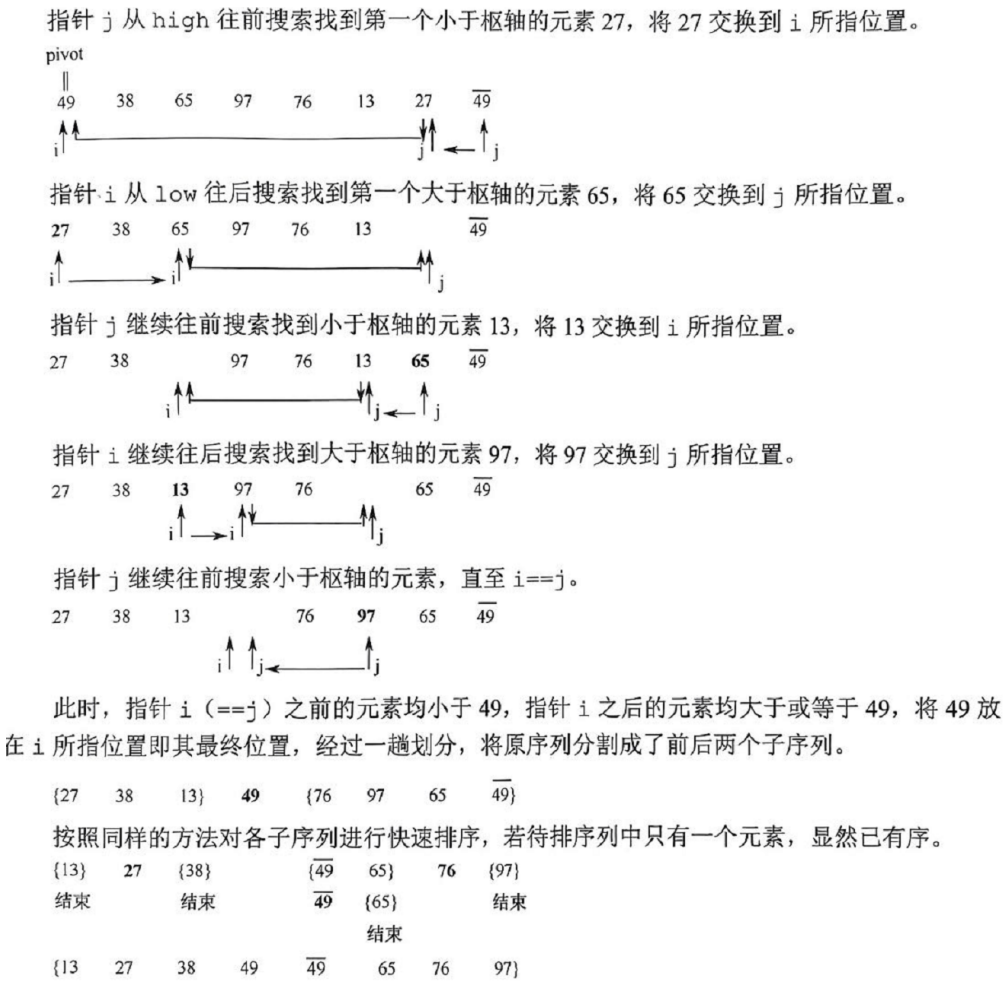
最好的情况是O(n),表中已经有序只需要比较,不需要交换

3.2快速排序

基本思想:分治法

在排序表中任取一个元素pivot作为枢轴,将排序表分为两部分,使前面的一部分全部小于pivot,后面的一部分全部大于pivot,这样不断划分

例如,取第一个元素49作为pivot




```
void QuickSort(ElemType A[],int low,int high){
    if(low<high){
        int pivotpos=Partition(A,low,high);//划分
        QuickSort(A,low,pivotpos-1);//依次对两个子表进行划分
        QuickSort(A,pivotpos+1,high);
    }
}

int Partition(ElemType A[],int low,int high){
```

```
    ElemType pivot=A[low];//取第一个元素作为枢轴
    while(low<high){//跳出循环条件
        while(low<high&&A[high]>=pivot) --high;//找到比枢轴小的元素,并将指针指向它
        A[low]=A[high];//将枢轴小的元素移动到左端
        while(low<high&&A[low]<=pivot) ++low;//找到比枢轴大的元素,并将指针指向它
        A[high]=A[low];//将枢轴大的元素移动到右端
    }
    A[low]=pivot;//枢轴元素存放到最终的位置
    return low;//返回最终位置
}
```

空间效率:需要借助一个递归工作栈,其**容量与递归调用的最大深度一致**

最好的情况下为

$$O(log_2n)$$

最坏的情况下要调用n-1次

平均情况下是

$$O(log_2n)$$

时间效率:**快速排序是所有内部排序算法中平均性能最优的排序算法**,其时间复杂度为

$$O(nlog_2n)$$

最坏的情况下,初始排序表基本有序或逆序时,时间复杂度为

$$O(n^2)$$

稳定性:不稳定

4.选择排序

选择:在每一趟待排序的元素中选取关键字最小的元素,作为有序子序列的第i个元素,直到结束

4.1简单选择排序

```
void SelectSort(ElemType A[],int n){
    for(int i=0;i<n-1;i++){//遍历
        int min=i;//记录最小元素位置
        for(int j=i+1;j<n;j++){//在i到n-1中选择最小的元素
            if(A[j]<A[min]) min=j;//更新最小的元素
        }
        if(min!=i) swap(A[i],A[min]);//两个元素交换
    }
}
```

空间效率:O(1)

时间复杂度为

$$O(n^2)$$

最好的情况,已经有序,只需要比较,不需要交换,时间复杂度仍然不变

稳定性:不稳定

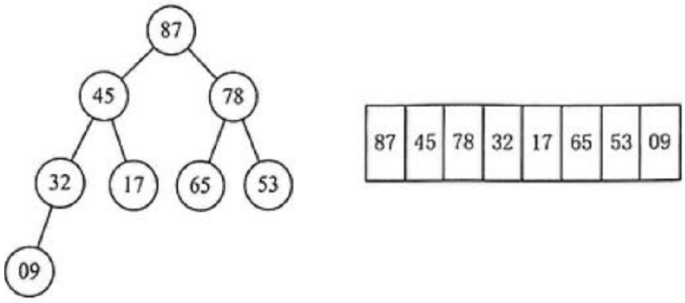
4.2堆排序

堆:满足以下条件的n个关键字序列L[1...n]:

1. L(i)>=L(2i)且L(i)>=L(2i+1)或
2. L(i)<=L(2i)且L(i)<=L(2i+1)

满足条件1的堆叫**大根堆**,满足条件2的堆叫**小根堆**

大根堆的示意图:

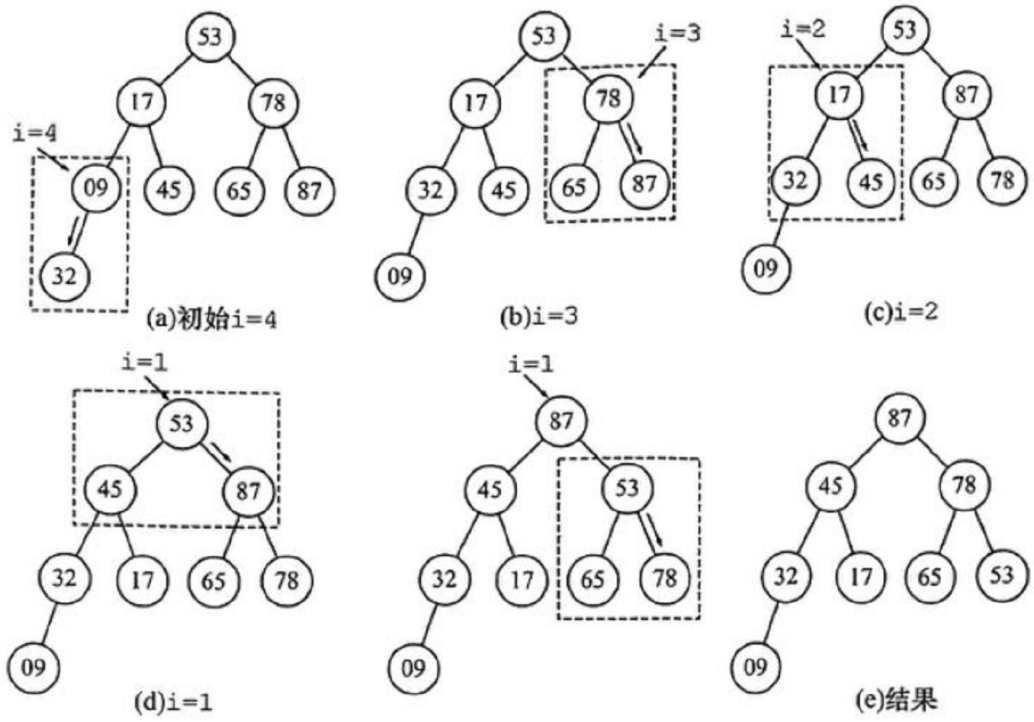


堆排序的思路:输出堆顶元素(最大或最小),将栈底元素送入堆顶,调整堆,使之满足大根堆或小根堆,依次重复

构造初始堆:

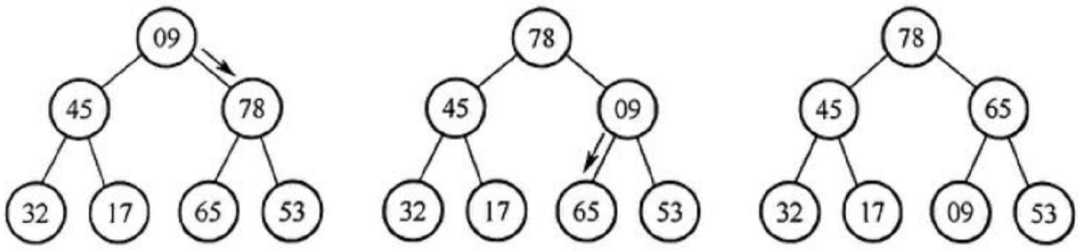
堆排序的关键是构造初始堆。n 个结点的完全二叉树，最后一个结点是第 $\lfloor n/2 \rfloor$ 个结点的孩子。对第 $\lfloor n/2 \rfloor$ 个结点为根的子树筛选（对于大根堆，若根结点的关键字小于左右孩子中关键字较大者，则交换），使该子树成为堆。之后向前依次对各结点（$\lfloor n/2 \rfloor - 1 \sim 1$）为根的子树进行筛选，看该结点值是否大于其左右子结点的值，若不大于，则将左右子结点中的较大值与之交换，交换后可能会破坏下一级的堆，于是继续采用上述方法构造下一级的堆，直到以该结点为根的子树构成堆为止。反复利用上述调整堆的方法建堆，直到根结点。

自下往上调整大根堆:



输出堆顶元素后调整大根堆:

将上面的大根堆的堆顶元素输出,然后将堆中的最后一个元素09与堆顶元素交换,选取左右孩子较大者交换,然后同样的调整



建立大根堆的算法:

```
void BuildMaxHeap(ElemType A[],int len){
    for(int i=len/2;i?0;i--) HeadAdjust(A,i,len);
}

void HeadAdjust(ElemType A[],int k,int len){
    A[0]=A[k];//A[0]暂存子树根结点
    for(int i=2*k;i<=len;i*=2){
        if(i<len&&A[i]<A[i+1]) i++;//取key较大的子结点下标
        if(A[0]>=A[i]) break;//筛选结束
        else{
            A[k]=A[i];//将A[i]调整到双亲结点上
            k=i;//修改k值,以便继续向下筛选
        }
    }
    A[k]=A[0];//被筛选结点的值放入最终位置
}
```

```
void HeapSort(ElemType A[],int len){
    BuildMaxHeap(A,len);//构建堆
    for(int i=len;i>1;i--){//n-1趟交换和建堆的过程
        Swap(A[i],A[1]);//输出堆顶元素
        HeadAdjust(A,1,i-1);//调整
    }
}
```

堆同时也能进行插入操作,将新结点放在堆的末端,然后一样的进行调整操作

空间效率:O(1)

时间效率:

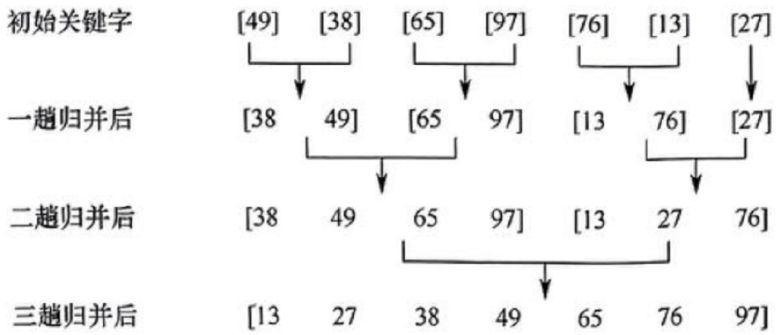
$$O(nlog_2n)$$

稳定性:不稳定

5.归并排序和基数排序

5.1归并排序

思想:将两个或两个以上的有序表合并成一个新的有序表,当两两归并时又称二路归并排序




```
ElemType *B=(ElemType *)malloc((n+1)*sizeof(ElemType));//辅助数组B
void Merge(ElemType A[],int low,int mid,int high){
    int i,j,k;
    for(k=low;k<=high;k++) B[k]=A[k];//将A中元素全部复制到B中
    for(i=low;j=mid+1,k=i;i<=mid&&j<=high;k++){
        if(B[i]<=B[j]) A[k]=B[i++];//将较小值复制到A中去
        else A[k]=B[j++];
    }
    while(i<=mid) A[k++]=B[i++];//若第一个表未检测完,复制
    while(j<=high) A[k++]=B[j++];//若第二个表未检测完,复制
}

void MergeSort(ElemType A[],int low,int high){
    if(low<high){
        int mid=(low+high)/2;
```

```
        MergeSort(A,low,mid);//左侧子序列递归排序
        MergeSort(A,mid+1,high);//右侧子序列递归排序
        Merge(A,low,mid,high);//归并
    }
}
```

空间效率:O(n)

时间效率:

$$O(nlog_2n)$$

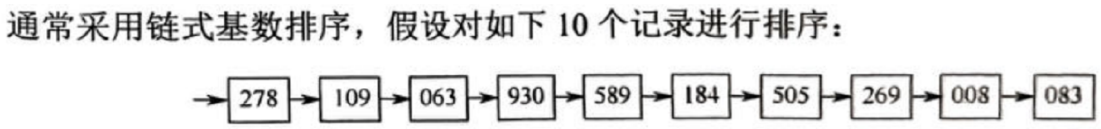
稳定性:稳定

5.2基数排序

不基于移动和比较

思想:基于关键字各位的大小

例如:一个三位的数字,先比较个位数进行排序,再比较十位数排序,再比较百位数排序,每个 位数相等的记录分配到同一个队列,然后收集



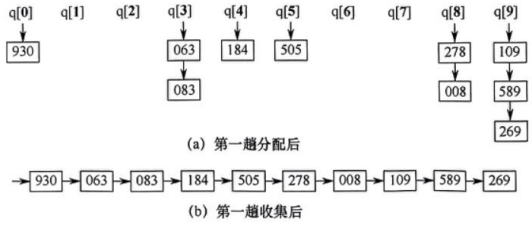


图 8.9　第一趟链式基数排序操作

第二趟分配用次低位子关键字 K^2 进行，将所有次低位子关键字（十位）相等的记录分配到同一个队列，如图 8.10(a)所示，第二趟收集后的结果如图 8.10(b)所示。

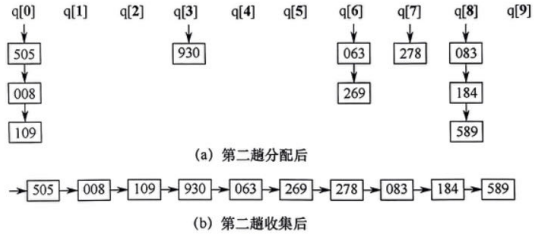


图 8.10　第二趟链式基数排序操作

第三趟分配用最高位子关键字 K^1 进行，将所有最高位子关键字（百位）相等的记录分配到同一个队列，如图 8.11(a)所示，第三趟收集后的结果如图 8.11(b)所示，至此整个排序结束。

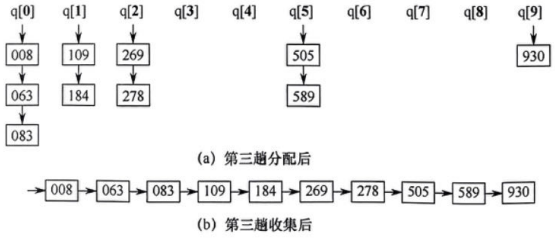


图 8.11　第三趟链式基数排序操作

空间效率:一趟排序需要辅助空间为r(r个队列,r个队头指针,r个队尾指针),空间复杂度为O(r)

时间效率:需要进行d趟分配和收集,每次分配需要O(n),一趟收集需要O(r),时间复杂度为O(d(n+r))

稳定性:稳定