2.2.3 顺序表算法设计

顺序表算法设计：数据采用顺序表存储，利用顺序表的基本操作来完成求解任务。

【例2-1】已知长度为n的线性表A采用顺序存储结构。设计一个时间复杂度为$O(n)$、空间复杂度为$O(1)$的算法，该算法删除线性表中所有值为x的数据元素。

以下两种方法都不满足要求：

- 如果每删除一个值为x的元素都进行移动，其时间复杂度为$O(n^2)$，空间复杂度为$O(1)$。
- 如果借助一个新的顺序表，存放将A中所有不为x的元素，其时间复杂度为$O(n)$，空间复杂度为$O(n)$。

解法一：设删除A中所有值等于x元素后的顺序表为$A1$，显然$A1$包含在A中，为此$A1$重用A的空间。

思路：扫描顺序表A，重建A只包含不等于x的元素。

删除顺序表中所有值为x的元素（方法1）演示

删除所有x=2的元素（k记录保留的元素个数，初值=0）：

k=3

k=3，L->length=k=3

删除完成

对应的算法如下：

```
void delnode1(SqList *&A, ElemType x)
{   int k=0, i;                    //k记录值不等于x的元素个数
    for (i=0;i<A->length;i++)
        if (A->data[i]!=x)      //若当前元素不为x，将其插入A中
        {   A->data[k]=A->data[i];
            k++;                   //不等于x的元素增1
        }
    A->length=k;                   //顺序表A的长度等于k
}
```

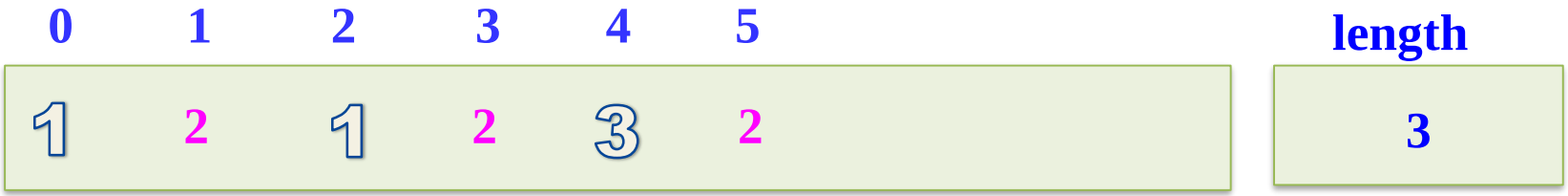
算法1：类似于建顺序表

解法二： 用k记录顺序表A中等于x的元素个数，一边扫描A一边统计k值。

思路： 将不为x的元素前移k个位置，最后修改A的长度。

删除顺序表中所有值为x的元素（方法2）演示

删除所有x=2的元素（k记录删除的元素个数，初值=0）



k=3

顺序表长度=6−k=3

删除完成

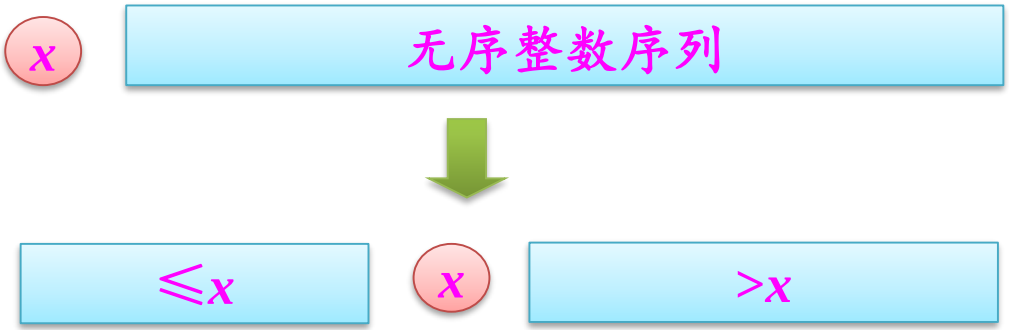
对应的算法如下：

```
void delnode2(SqList *&A,ElemType x)
{    int k=0, i=0;                  //k记录值等于x的元素个数
     while (i<A->length)
     {    if (A->data[i]==x)        //当前元素值为x时k增1
               k++;
          else                      //当前元素不为x时将其前移k个位置
               A->data[i-k] = A->data[i];
          i++;
     }
     A->length-=k;  //顺序表A的长度递减k
}
```

思考题

为什么说上述两个算法都能够满足题目的要求？

【例2-2】设顺序表L有10个整数。设计一个算法，以第一个元素为分界线（基准），将所有小于等于它的元素移到该元素的前面，将所有大于它的元素移到该元素的后面。



解法1：

pivot=L->data[0]（基准）

*j*从后向前找≤pivot的元素 ➡ 两者交换

*i*从前向后找>pivot的元素

pivot

1 0 2 3 3 5 7 4 6 8

```
void move1(SqList *&L)
{    int i=0, j=L->length-1;  ElemType tmp;
     ElemType pivot=L->data[0];       //以data[0]为基准
     while (i<j)
     {    while (i<j && L->data[j]>pivot)
              j--;                    //从后向前扫描，找一个≤pivot的元素
          while (i<j && L->data[i]<=pivot)
              i++;                    //从前向后扫描，找一个>pivot的元素
          if  (i<j)
          {    tmp=L->data[i];        //L->data[i] ⇔ L->data[j]
               L->data[i]=L->data[j];
               L->data[j]=tmp;
          }
     }
     tmp=L->data[0];                  //L->data[0] ⇔ L->data[j]
     L->data[0]=L->data[j];   L->data[j]=tmp;
}
```

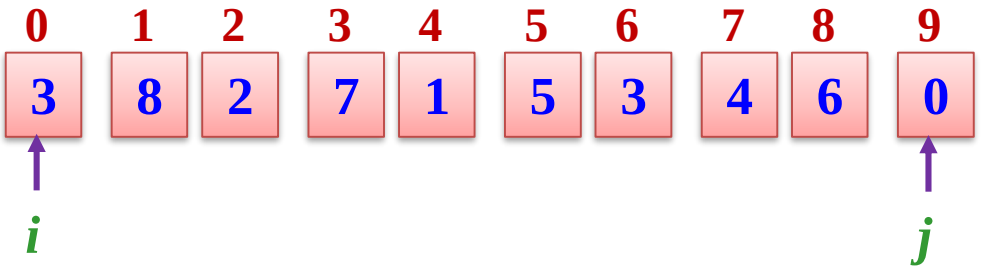
解法2：

pivot=L->data[0]（基准）

*j*从后向前找小于等于pivot的元素：前移

*i*从前向后找大于pivot的元素：后移

pivot（基准）



0 3 2 1 3 5 7 4 6 8

算法时间复杂度为O(*n*)。

```
void move2(SqList *&L)
{    int i=0,  j=L->length-1;
     ElemType  pivot=L->data[0];          //以data[0]为基准
     while (i<j)
     {    while (j>i && L->data[j]>pivot)
               j--;                       //从右向左扫描,找一个≤pivot的data[j]
          L->data[i]=L->data[j];          //将其放入data[i]处
          while (i<j && L->data[i]<=pivot)
               i++;                       //从左向右扫描,找一个>pivot的记录data[i]
          L->data[j]=L->data[i];          //将其放入data[j]处
     }
     L->data[i]=pivot;                    //放置基准
}
```

为什么解法2比解法1更好？

- 两个记录*a*、*b*交换：tmp=*a*; *a*=*b*; *b*=tmp;　需要3次移动
- 多个相邻记录连续交换，如*a*、*b*、*c*：

❶ 位置1和位置2的元素交换 ⇨ *b*、*a*、*c*　需要3次移动

❷ 位置2和位置3的元素交换 ⇨ *b*、*c*、*a*　需要3次移动

共6次移动

而采用：

tmp=*a*; *a*=*b*; *b*=*c*; *c*=tmp;　4次移动

性能得到提高。

——本讲完——