第2章 线性表

提纲

CONTENTS

2.1 线性表的定义

2.1.1 什么是线性表

线性表是具有相同特性的数据元素的一个有限序列。

- 所有数据元素类型相同。
- 线性表是有限个数据元素构成的。
- 线性表中数据元素与位置相关，即每个数据元素有唯一的序号。

线性表的逻辑结构表示

$$(a_0, a_1, \cdots, a_i, a_{i+1}, \cdots, a_{n-1})$$

用图形表示的逻辑结构：

$$a_0 \to a_1 \to \cdots \to a_i \to a_{i+1} \to \cdots \to a_{n-1}$$

说明

线性表中每个元素a_i的唯一位置通过序号或者索引i表示，为了算法设计方便，将逻辑序号和存储序号统一，均假设从0开始，这样含n个元素的线性表的元素序号i满足$0 \leqslant i \leqslant n-1$。

2.1.2 线性表的抽象数据类型描述

```
ADT List
{
数据对象：
    D={a_i | 0≤i≤n-1, n≥0 }
数据关系：
    r={<a_i, a_{i+1}> | a_i, a_{i+1}∈D, i=0, …, n-2}
基本运算：
    CreateList( a)：由(E类型)数组a中的全部元素建立线性表的相应存储结构。
    Add(e)：将元素e添加到线性表末尾。
    getsize()：求线性表的长度。
    GetElem(int i)：求线性表中序号为i的元素。
    SetElem(int i, E e)：设置线性表中序号i的元素值为e。
    GetNo(E e)：求线性表中第一个值为e的元素的序号。
    Insert(int i, E e)：在线性表中插入数据元素e作为第i个元素。
    Delete(int i)：在线性表中删除第i个数据元素。
    display()：输出线性表的所有元素。
}
```

2.2 线性表的顺序存储结构

2.2.1 线性表的顺序存储结构—顺序表

长度为n的线性表存放在顺序表中

数组下标	0	1	…	i-1	i	…	n-1	capacity-1
data数组	a_0	a_1	…	a_{i-1}	a_i	…	a_{n-1}	…

data为Python中的列表

- Data列表存放线性表元素
- Data列表的容量（存放最多的元素个数）为capacity。
- 线性表中实际数据元素个数size

```
class SqList:                                    #顺序表类
  def __init__(self):                            #构造方法
    self.initcapacity=5;                         #初始容量设置为5
    self.capacity=self.initcapacity              #容量设置为初始容量
    self.data=[None]*self.capacity               #设置顺序表的空间
    self.size=0                                  #长度设置为0
  #线性表的基本运算算法
```

2.2.2 线性表基本运算算法在顺序表中的实现

在动态分配顺序表的空间时，初始容量设置为**initcapacity**，当添加或者插入元素可能需要扩大容量，在删除元素时可能需要减少容量。

```
def resize(self, newcapacity):    #改变顺序表的容量为newcapacity
  assert newcapacity>=0           #检测参数正确性的断言
  olddata=self.data
  self.data=[None]*newcapacity
  self.capacity=newcapacity
  for i in range(self.size):
    self.data[i]=olddata[i]
```

1.整体建立顺序表

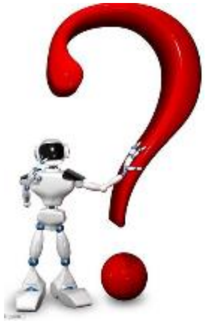
由含若干个元素的数组*a*的全部元素整体创建顺序表，即依次将*a*中的元素添加到**data**数组的末尾，当出现上溢出时按实际元素个数**size**的两倍扩大容量。

```
def CreateList(self, a):                    #由数组a中元素整体建立顺序表
  for i in range(0,len(a)):
    if self.size==self.capacity:            #出现上溢出时
      self.resize(2*self.size);             #扩大容量
    self.data[self.size]=a[i]
    self.size+=1                            #添加后元素个数增加1
```

2.顺序表基本运算算法

（1）将元素*e*添加的线性表末尾Add(*e*)

```
def Add(self, e):                      #在线性表的末尾添加一个元素e
    if self.size==self.capacity:       #顺序表空间满时倍增容量
        self.resize(2*self.size)
    self.data[self.size]=e             #添加元素e
    self.size+=1                       #长度增1
```



时间复杂度是多少?

（2）求线性表的长度getsize()

```
def getsize(self):                      #求线性表长度
  return self.size
```

（3）求线性表中序号为i的元素GetElem(i)

```
def __getitem__(self,i):             #求序号为i的元素
  assert 0<=i<self.size              #检测参数i正确性的断言
  return self.data[i]                #返回data[i]
```

（4）设置线性表中序号为i的元素SetElem(i, e)

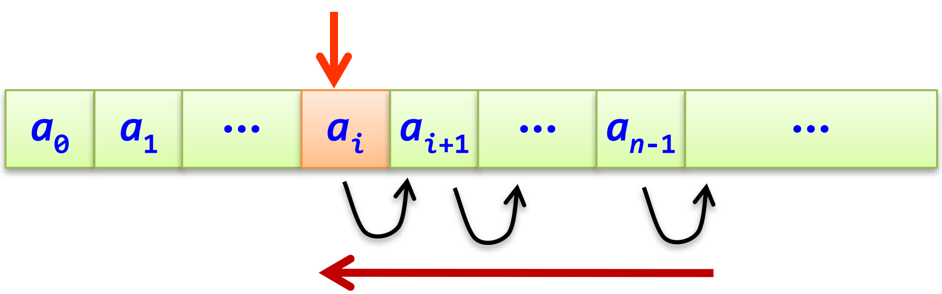
```
def __setitem__(self, i, x):                    #设置序号为i的元素
  assert 0<=i<self.size                         #检测参数i正确性的断言
  self.data[i]=x
```

（5）求线性表中第一个值为e的元素的逻辑序号GetNo(e)

```
def GetNo(self, e):                    #查找第一个为e的元素的序号
  i=0;
  while i<self.size and self.data[i]!=e:
    i+=1                               #查找元素e
  if (i>=self.size):                   #未找到时返回-1
    return -1;
  else:
    return i;                          #找到后返回其序号
```

（6）在线性表中插入e作为第i个元素Insert(i，e)

```
def Insert(self, i, e):                  #在线性表中序号i位置插入元素e
  assert 0<=i<=self.size                 #检测参数i正确性的断言
  if self.size==self.capacity:           #满时倍增容量
    self.resize(2*self.size)
  for j in range(self.size,i,-1):        #将data[i]及后面元素后移一个位置
    self.data[j]=self.data[j-1]
  self.data[i]=e                         #插入元素e
  self.size+=1                           #长度增1
```



从a_{n-1}元素开始移动起

a_0	a_1	…	a_i	a_{i+1}	…	a_{n-1}	…

主要时间花在元素移动上。有效插入位置i的取值是0～n，共有n+1个位置可以插入元素：

- 当i=0时，移动次数为n，达到最大值。
- 当i=n时，移动次数为0，达到最小值。
- 其他情况，需要移动data[i..n-1]的元素，移动次数为(n-1)-i+1=n-i。

$p_i = \frac{1}{n+1}$ 所需移动元素的平均次数为：

$$\sum_{i=0}^{n} p_i\,(n-i) = \frac{1}{n+1}\sum_{i=0}^{n}(n-i) = \frac{1}{n+1} \times \frac{n(n+1)}{2} = \frac{n}{2}$$

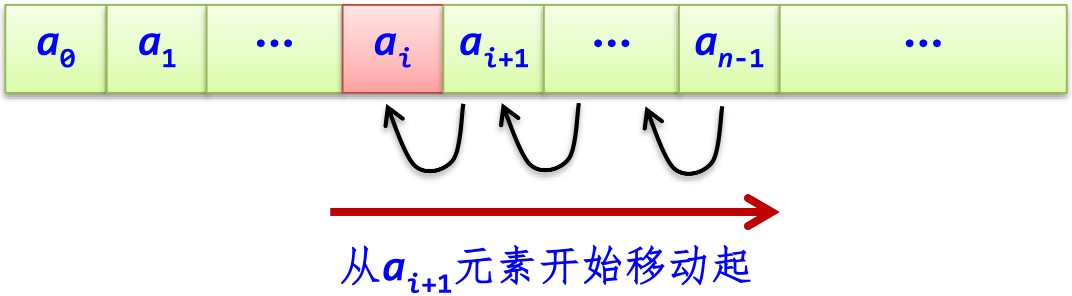
插入算法的平均时间复杂度为O(n)。

说明

扩容运算resize()在n次插入中仅仅调用一次，其平摊时间为O(1)，上述算法时间分析中可以忽略它。

（7）在线性表中删除第i个数据元素Delete(i)

```
def Delete(self, i):                    #在线性表中删除序号i的元素
  assert 0<=i<=self.size-1              #检测参数i正确性的断言
  for j in range(i,self.size-1):
    self.data[j]=self.data[j+1]         #将data[i]之后的元素前移一个位置
  self.size-=1                          #长度减1
  if self.capacity>self.initcapacity and self.size<=self.capacity/4:
   self.resize(self.capacity//2)        #满足缩容条件则容量减半
```



a_0	a_1	…	a_i	a_{i+1}	…	a_{n-1}	…

主要时间花在元素移动上。有效删除位置i的取值是0～n-1，共有n个位置可以删除元素：

- 当i=0时，移动次数为n-1，达到最大值。
- 当i=n-1时，移动次数为0，达到最小值。
- 其他情况，需要移动data[i+1..n-1]的元素，移动次数为(n-1)-(i+1)+1=n-i-1。

$p_i = \frac{1}{n}$ 所需移动元素的平均次数为：

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

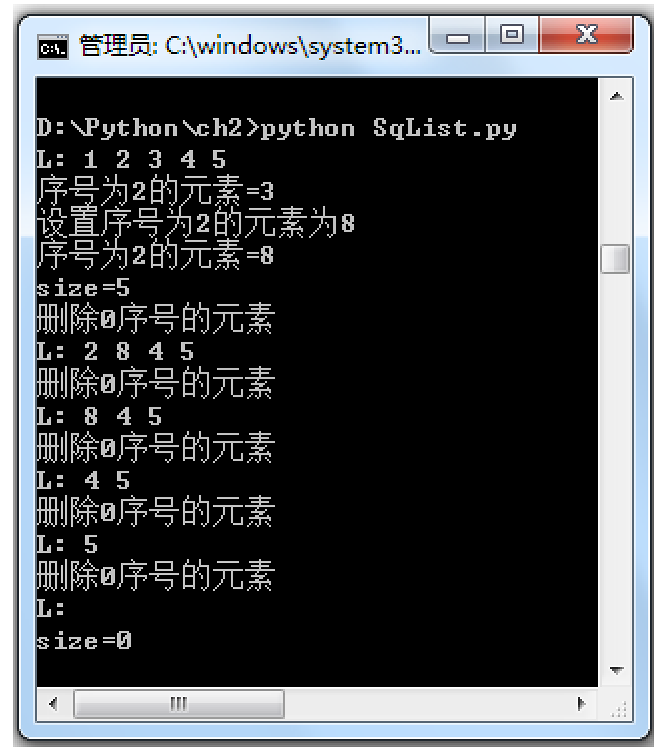
删除算法的平均时间复杂度为O(n)。

(8) 输出线性表所有元素display()

```
def display(self):                          #输出线性表
  for i in range(0,self.size):
    print(self.data[i],end='  ')
  print()
```

程序验证

```python
if __name__ == '__main__':
  L=SqList()
  for i in range(1,6):
    L.Add(i)
  print("L: ",end=''),L.display()
  print("序号为2的元素=%d" %(L[2]))
  print("设置序号为2的元素为8")
  L[2]=8
  print("序号为2的元素=%d" %(L[2]))
  n=L.getsize()
  print("size=%d" %(n))
  for i in range(0,n):
    print("删除%d序号的元素" %(0))
    L.Delete(0)
    print("L: ",end=''),L.display()
  print("size=%d" %(L.getsize()))
```



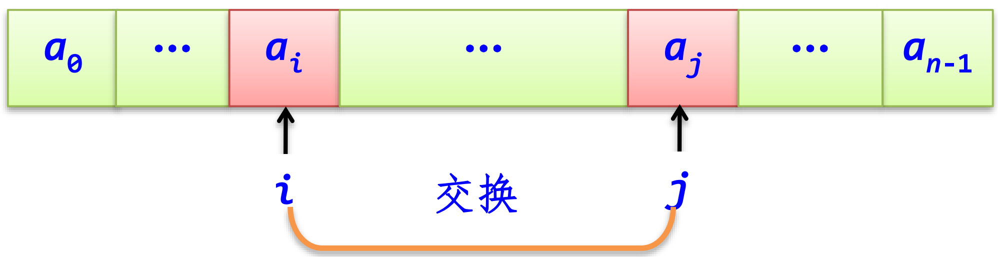
2.2.3 顺序表的应用算法设计示例

1.基于顺序表基本操作的算法设计

【例2.1】对于含有*n*个整数元素的顺序表L，设计一个算法将其中所有元素逆置。

例如L=(1，2，3，4，5)，逆置后L=(5，4，3，2，1)。并给出算法的时间复杂度和空间复杂度。

【例2.1】对于含有n个整数元素的顺序表L，设计一个算法将其中所有元素逆置。例如L=(1，2，3，4，5)，逆置后L=(5，4，3，2，1)。并给出算法的时间复杂度和空间复杂度。

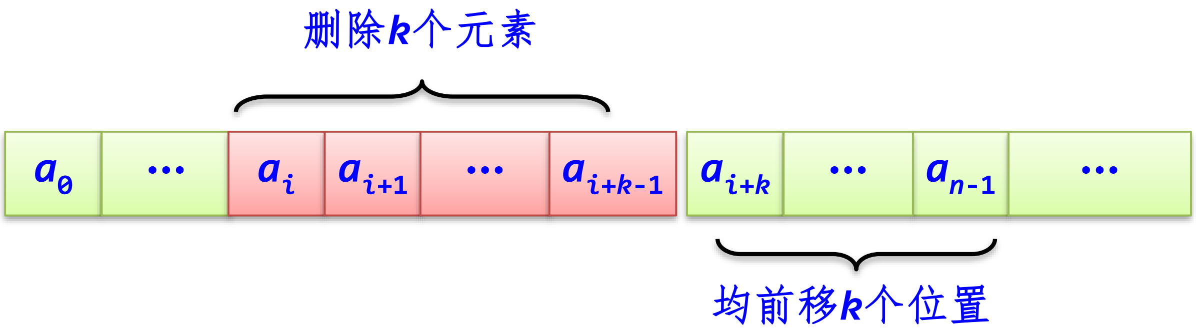



```
def Reverse(L):                    #求解算法
  i=0
  j=L.getsize()-1
  while i<j :
     L[i],L[j]=L[j],L[i]           #序号为i和j的两个元素交换
     i+=1
     j-=1
```

【例2.2】 假设有一个整数顺序表L，设计一个算法用于删除从序号i开始的k个元素。

例如L=（1，2，3，4，5），删除i=1开始的k=2个元素后L=（1，4，5）。

解： 在参数正确时，直接将 a_{i+k} ~ a_{n-1} 的所有元素依次前移 k 个位置。

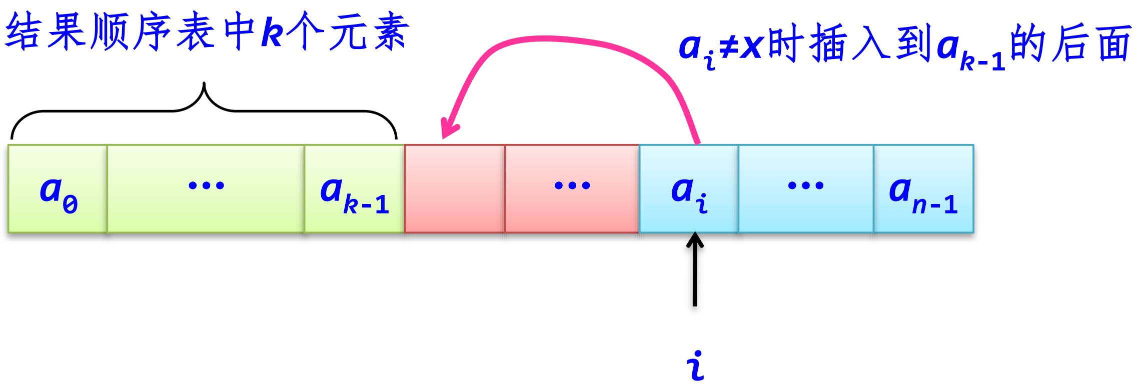


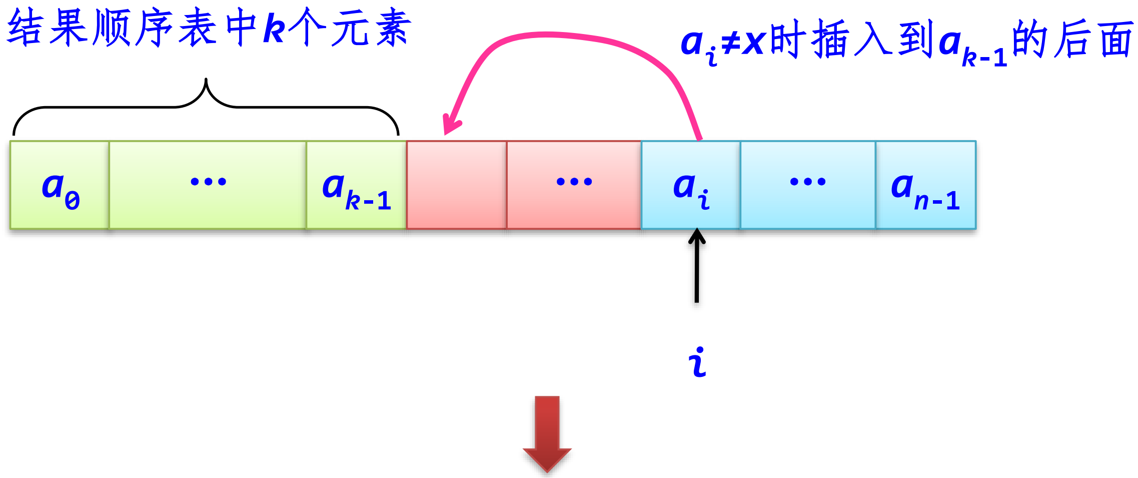

```
def Deletek(L, i, k):                           #求解算法
  assert i>=0 and k>=1 and i+k>=1 and i+k<=L.getsize()
  for j in range(i+k,L.getsize()):              #将要删除的元素均前移k个位置
     L[j-k]=L[j]
  L.size-=k                                     #长度减k
```

【例2.3】对于含有n个整数元素的顺序表L，设计一个算法用于删除其中所有值为x的元素。

例如L=(1，2，1，5，1)，若x=1，删除后L=(2，5)。并给出算法的时间复杂度和空间复杂度。

解法1：对于整数顺序表L，删除其中所有x元素后得到的结果顺序表可以与原L共享，所以求解问题转化为新建结果顺序表。





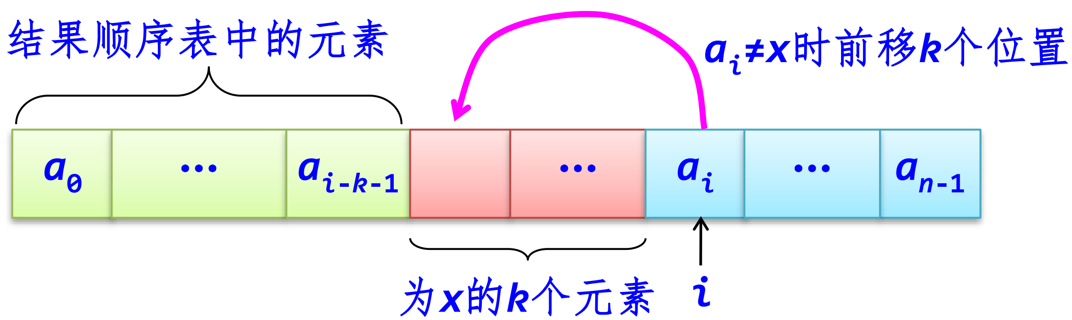

```
def Deletex1(L, x):                         #求解算法1
  k=0
  for i in range(L.getsize()):
    if L[i]!=x:                             #将不为x的元素插入到data中
      L[k]=L[i]
      k+=1
  L.size=k                                  #重置长度为k
```

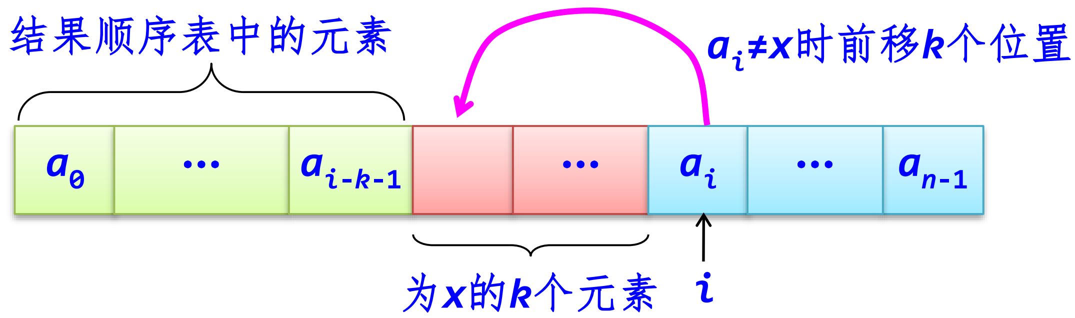
解法2： 前移法，对于整数顺序表L，从头开始扫描L，用k累计当前为止值为x的元素个数（初始值为0），处理当前序号为i的元素a_i：

（1）若a_i是不为x的元素，此时前面有k个为x的元素，将a_i前移k个位置，继续处理下一个元素。

（2）若是为x的元素，置k++，继续处理下一个元素。

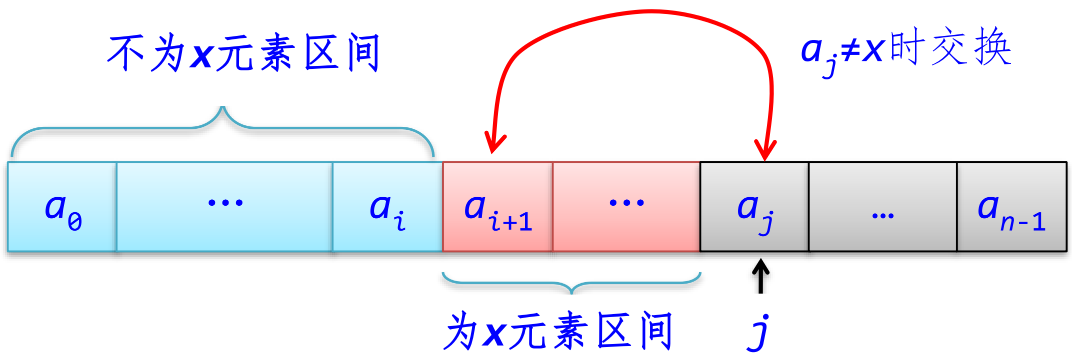
最后将L的长度减少k。





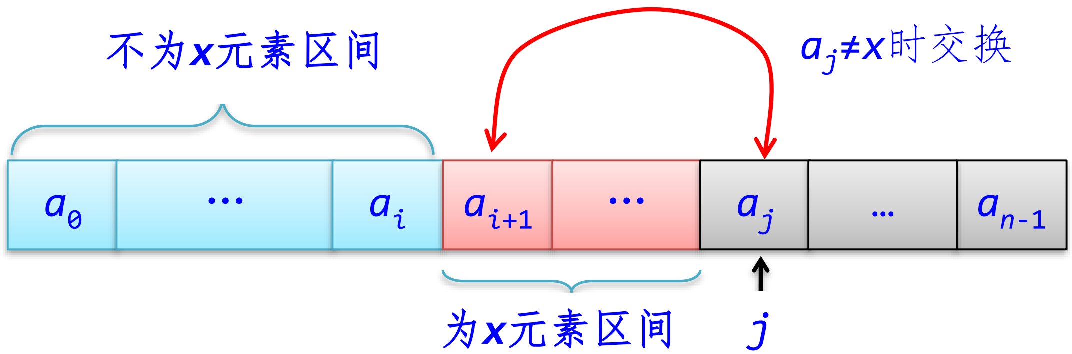

```
def Deletex2(L, x):                              #求解算法2
  k=0;
  for i in range(L.getsize()):
    if L[i]!=x:                                  #将不为x的元素前移k个位置
      L[i-k]=L[i]
    else:                                        #累计删除的元素个数k
      k+=1
  L.size-=k                                      #重置长度减少k
```

解法3： 由解法2延伸出区间划分法



初始时，“不为x的区间”为空 ⇨ i=-1，j从0开始遍历，“为x的区间”是a[i+1..j-1]

- 若a[j]=x，跳过，j++。
- 若a[j]≠x，操作是，先执行i++，将a[j]与a[i]进行交换，再执行j++继续遍历其余元素。




```
def Deletex3(L, x):                         #求解算法3
  i=-1
  j=0
  while j<L.getsize():                      #j遍历所有元素
    if L[j]!=x:                             #找到不为x的元素a[j]
      i+=1                                  #扩大不为x的区间
      if i!=j:
        L[i],L[j]=L[j],L[i]                 #将序号为i和j的两个元素交换
    j+=1                                    #继续扫描
  L.size=i+1                                #重置长度为i+1
```

扩展

各种顺序表的高效算法设计

- 设计一个算法，从一给定的顺序表L中删除元素值在x到y（$x \leqslant y$）之间的所有元素，要求算法的时间复杂度为$O(n)$，空间复杂度为$O(1)$。
- 设计一个算法从有序顺序表中删除重复的元素，并使剩余元素间的相对次序保持不变。
- …

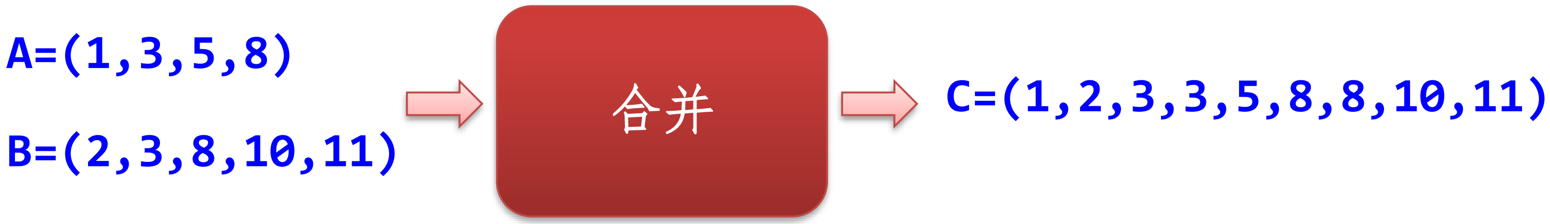
3.有序顺序表的算法设计

- 有序表是指按元素值或者某属性值递增或者递减排列的线性表，有序表是线性表的一个子集。
- 有序顺序表是有序表的顺序存储结构。
- 对于有序表可以利用其元素的有序性提高相关算法的效率，二路归并就是有序表的一种经典算法。

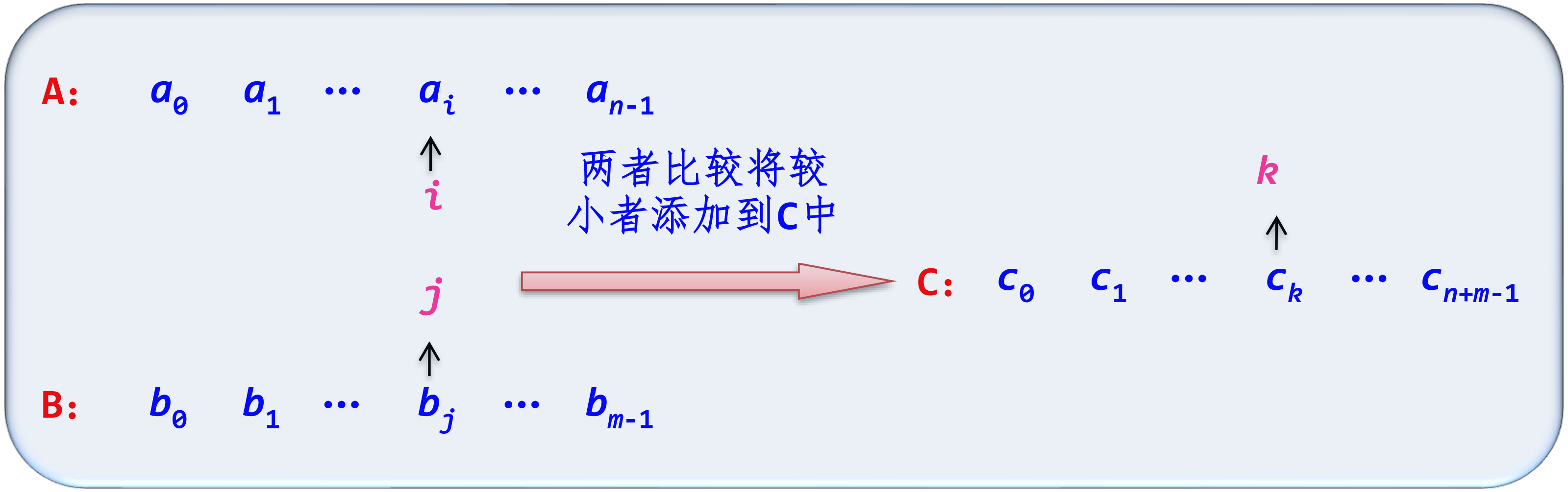
A=(1,3,8,23,30)

一个递增有序整数顺序表

【例2.4】有两个按元素值递增有序的整数顺序表A和B，设计一个算法将顺序表A和B的全部元素合并到一个递增有序顺序表C中。并给出算法的时间复杂度和空间复杂度。



二路归并：



i遍历A，j遍历B，均从0开始

while i,j都没有超界

　　a_i与b_j比较：较小元素添加到C中，后移相应指针

将没有遍历完的元素添加到C中

```python
def Merge2(A, B):                            #求解算法
  C=SqList()                                 #新建顺序表C
  i=j=0                                      #i用于遍历A,j用于遍历B
  while i<A.getsize() and j<B.getsize():     #两个表均没有遍历完毕
    if A[i]<B[j]:
      C.Add(A[i])                            #将较小的A[i]添加到C中
      i+=1
    else:
      C.Add(B[j])                            #将较小的B[j]添加到C中
      j+=1
  while i<A.getsize():                       #若A没有遍历完毕
    C.Add(A[i])
    i+=1
  while j<B.getsize():                       #若B没有遍历完毕
    C.Add(B[j])
    j+=1
  return C                                   #返回C
```

- 算法中尽管有多个**while**循环语句，但恰好对顺序表**A**、**B**中每个元素均访问一次，所以时间复杂度为**O(*n*+*m*)** 。
- 算法中需要在临时顺序表**C**中添加***n*+*m***个元素，所以算法的空间复杂度也是**O(*n*+*m*)**。

二路归并中，若两个有序表的长度分别为*n*、*m*，算法的主要时间花费在元素比较上，那么比较次数是多少呢?

- 最好的情况下，整个归并中仅仅是较长表的第一个元素与较短表每个元素比较一次，此时元素比较次数为MIN(*n*，*m*)（为最少元素比较次数），如*A*=(1，2，3)，*B*=(4，5，6，7，8)，只需比较3次。
- 最坏的情况下，这*n*+*m*个元素均两两比较一次，比较次数为*n*+*m*-1（为最多元素比较次数），如*A*=(1，3，5，7)，*B*=(2，4，6)，需要比较6次。

2009年全国计算机学科考研题

【例2.5】一个长度为L（L≥1）的升序序列S，处在第$\lceil L/2 \rceil$个位置的数称为S的**中位数**。

例如：若序列S_1=(11，13，15，17，19)，则S_1的中位数是15。

两个序列的中位数是含它们所有元素的升序序列的中位数。例如，若S_2=(2，4，6，8，20)，则S_1和S_2的中位数是11。

现有两个等长的升序序列A和B，试设计一个在时间和空间两方面都尽可能高效的算法，找出两个序列A和B的中位数。要求：

（1）给出算法的基本设计思想。

（2）根据设计思想，采用C、C++或Java语言描述算法，关键之处给出注释。

（3）说明你所设计算法的时间复杂度和空间复杂度。

思路

S_1=(11, 13, 15, 17, 19)

S_2=(2, 4, 6, 8, 20)

二路归并

S=(2, 4, 6, 8, 11, 13, 15, 17, 19, 20)

中位数

实际上，不需要求出S的全部元素，用*k*记录当前归并的元素个数，当*k*=*n*时，归并的那个元素就是中位数。

```python
def Middle(A,B):                             #求解算法
  i=j=k=0
  while i<A.getsize() and j<B.getsize(): #两个有序顺序表均没有扫描完
    k+=1                                     #元素比较次数增1
    if A[i]<B[j]:                            #A中当前元素为较小的元素
      if k==A.getsize():                     #恰好比较了n次
        return A[i]                          #返回A中的当前元素
      i+=1
    else:                                    #B中当前元素为较小的元素
      if k==B.getsize():                     #恰好比较了n次
        return B[j];                         #返回B中的当前元素
      j+=1
```

算法的时间复杂度为O(n)，空间复杂度为O(1)。