第二章　　线性表

1. 线性表的定义

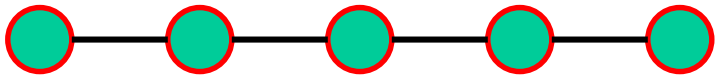
2. 线性表的顺序表示和实现

3. 线性表的链式表示和实现

4. 线性表的应用

线性结构的特点：

除第一个元素无直接前驱，最后一个元素无直接后继外，其他每个数据元素都有一个前驱和后继。



2.1 线性表的定义

线性表是一个具有相同特性的数据元素的有限序列。

- 相同特性：所有元素属于同一数据类型。
- 序列：相邻数据元素之间存在序偶关系。

线性表例：

(1). 一个整数线性表

1	2	3	4	5	6	7

(2). 一个图像线性表

(3). 一个学生成绩情况登记表

学号	姓名	高等数学	英语	C语言
1601001	李林	88	87	80
1601002	王东	76	90	66
1601003	张阳	86	75	83
1601004	赵海	69	80	78
……				

若将线性表记为：

$(a_1, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n)$

则a_i（$1 \leqslant i \leqslant n$）表示第i个数据元素。称$a_{i-1}$是$a_i$的直接前驱元素，$a_{i+1}$是$a_i$的直接后继元素。

线性表中元素的个数n(n>=0)定义为**线性表的长度**。当n=0时，称为**空表**。

线性表的抽象数据类型定义：

ADT List {

数据对象：$D=\{a_i \mid a_i \in ElemSet, i=1,2,\cdots,n, n\geq 0\}$

数据关系：$R1=\{\langle a_{i-1}, a_i\rangle \mid a_{i-1}, a_i \in D, i=2,\cdots,n\}$

基本操作：

InitList(&L)

操作结果：构造一个空的线性表L。

DestroyList(&L)

初始条件：线性表L已存在。

操作结果：销毁线性表L。

ClearList(&L)

初始条件：线性表L已存在。

操作结果：将L重置为空表。

ListEmpty(L)

初始条件：线性表L已存在。

操作结果：若L为空表，则返回TRUE，否则FALSE。

ListLength(L)

初始条件：线性表L已存在。

操作结果：返回L中元素个数。

GetElem(L, i, &e)

初始条件：线性表L已存在， $1 \le i \le$ ListLength (L)

操作结果：用e返回L中第i个元素的值。

LocateElem(L, e, compare( ))

初始条件：线性表L已存在，compare( )是元素判定函数。

操作结果：返回L中第1个与e满足关系compare( )的元素的位序。若这样的元素不存在，则返回值为0。

PriorElem(L, cur_e, &pre_e)

初始条件：线性表L已存在。

操作结果：若cur_e是L的元素，但不是第一个，则用pre_e 返回它的前驱，否则操作失败，pre_e无定义。

NextElem(L, cur_e, &next_e)

初始条件：线性表L已存在。

操作结果：若cur_e是L的元素，但不是最后一个，则用next_e返回它的后继，否则操作失败，next_e无定义。

ListInsert(&L, i, e)

初始条件：线性表L已存在，1≤i≤ListLength (L)+1

操作结果：在L的第i个元素之前插入新的元素e，L的长度增1。

ListDelete(&L, i, &e)

初始条件：线性表L已存在且非空，1≤i≤ListLength (L)

操作结果：删除L的第i个元素，并用e返回其值，L的长度减1。

ListTraverse(L, visit( ))

初始条件：线性表L已存在。

操作结果：依次对L的每个元素调用函数visit( )。一旦visit( )失败，则操作失败。

} ADT List

对上述定义的抽象数据类型线性表，还可进行更复杂的操作，如：

例2-1 假设有两个集合A和B分别用两个线性表LA和LB表示（即：线性表中的数据元素即为集合中的成员），现要求一个新的集合A＝A∪B。

上述问题可演绎为，要求对线性表作如下操作：扩大线性表LA，将存在于线性表LB中而不存在于线性表LA中的数据元素插入到线性表LA中去。

1. 从线性表LB中依次取得每个数据元素; **GetElem(LB, i,&e)**
2. 依值在线性表LA中进行查访; **LocateElem(LA, e, equal)**
3. 若不存在，则插入之。 **ListInsert(LA, n+1, e)**

```
void union(List &La, List Lb) {
// 将所有在线性表Lb中但不在La中的数据元素插入到La中
La_len = ListLength(La);
Lb_len =ListLength(Lb); // 求线性表的长度
for (i = 1; i <= Lb_len; i++) {
   GetElem(Lb, i, e);       // 取Lb中第i个数据元素赋给e
   if(!LocateElem(La, e, equal( ))   ListInsert(La, ++La_len, e);
     // La中不存在和 e 相同的数据元素，则插入之
   }
}
```

例2-3 归并两个“其数据元素按值非递减有序排列的”线性表LA和LB，求得线性表LC也具有同样特性。

设 $La = (a_1, \cdots, a_i, \cdots, a_n)$

$Lb = (b_1, \cdots, b_j, \cdots, b_m)$

$Lc = (c_1, \cdots, c_k, \cdots, c_{m+n})$

则 $C_k = \begin{cases} a_i & i = 1,2,\ldots,n \\ b_j & j = 1,2,\ldots,m \end{cases}$ $k = 1, 2, \cdots, m+n$

1. 分别从LA和LB中取得当前元素ai和bj；
2. 若ai≤bj，则将ai插入到LC中，否则将bj插入到LC中。

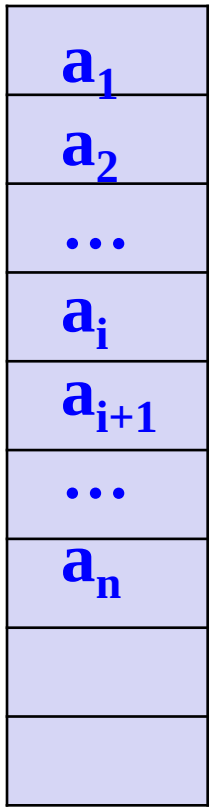
```
void MergeList(List La, List Lb, List &Lc) {
// 已知线性表La和Lb中的元素按值非递减123排列。归并La
// 和Lb得到新的线性表Lc，Lc的元素也按值非递减排列。
 InitList(Lc);       i = j = 1; k = 0;
 La_len = ListLength(La);   Lb_len = ListLength(Lb);
 while ((i <= La_len) && (j <= Lb_len)) {     // La和Lb均非空
     GetElem(La, i, ai); GetElem(Lb, j, bj);
     if (ai <= bj)      {ListInsert(Lc, ++k, ai); ++i; }
     else   {ListInsert(Lc, ++k, bj); ++j; }
   }
 while (i <= La_len) {
     GetElem(La, i++, ai);    ListInsert(Lc, ++k, ai);
   }
while (j <= Lb_len) {
     GetElem(Lb, j++, bj);    ListInsert(Lc, ++k, bj);
   }
}
```

2.2 线性表的顺序表示和实现

2.2.1 线性表的顺序存储表示

线性表的顺序存储结构:

用一组地址连续的存储单元依次存储线性表的数据元素。用这种方法存储的线表简称顺序表。

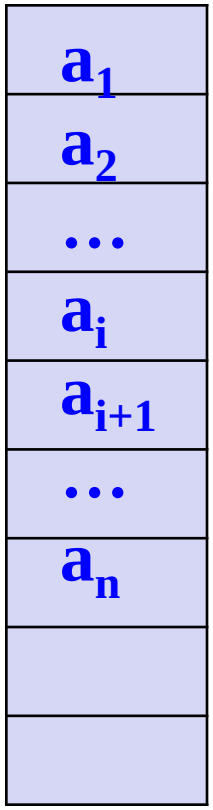


线性表（$a_1, a_2, a_3, \ldots a_n$）的顺序存储结构

特点:

逻辑上相邻的数据元素，其物理次序也相邻。

假设线性表的每个元素需占用L个存储单元，并以所占的第一个单元的存储地址作为数据元素的存储位置。则线性表中第i+1个数据元素的存储位置LOC(a_{i+1})和第i个数据元素的存储位置LOC(a_i)之间满足下列关系：

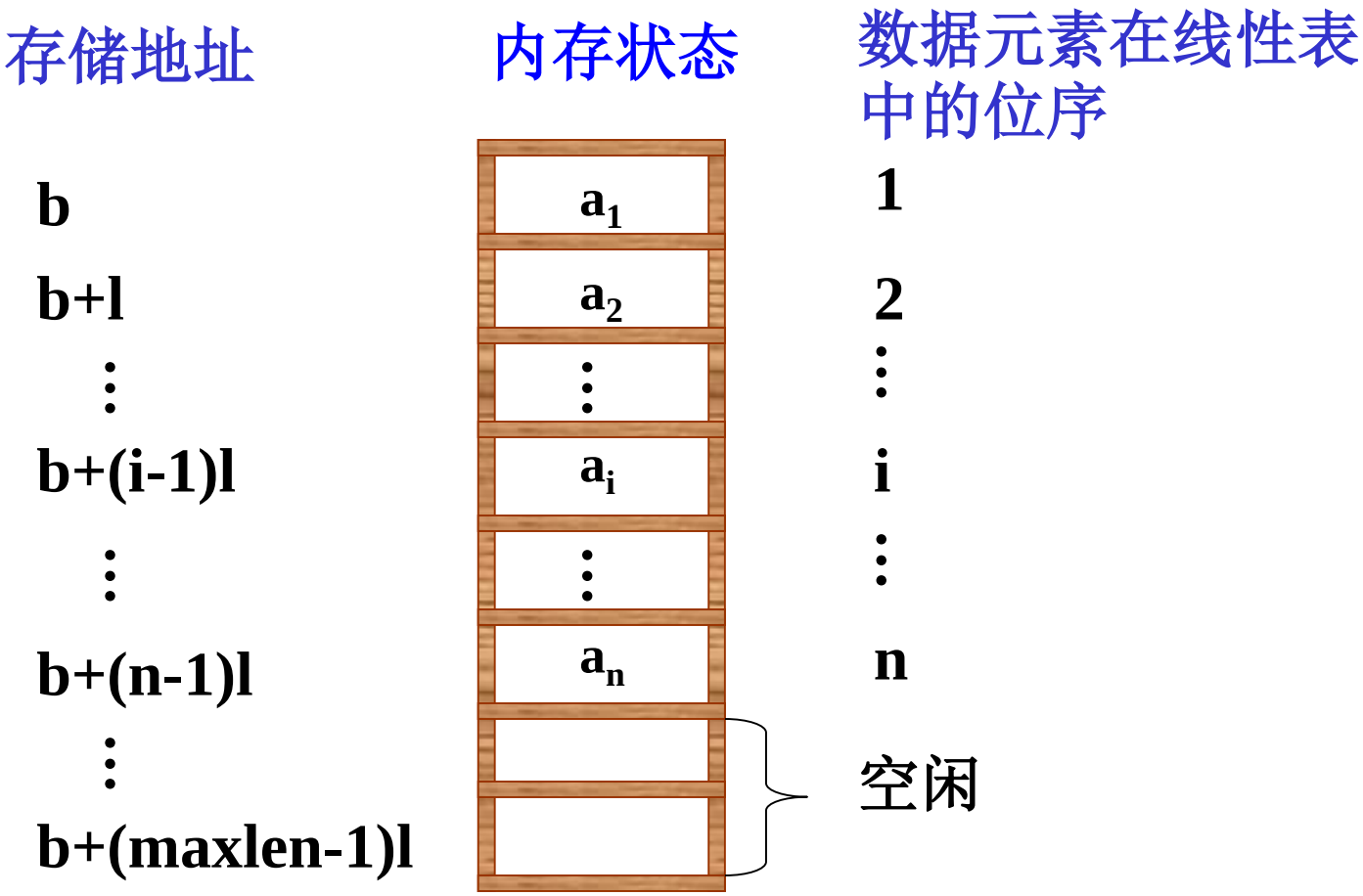


$$LOC(a_{i+1})=LOC(a_i)+L$$

线性表的第i个数据元素a_i的存储位置为：

$$LOC(a_i)=LOC(a_1)+(i-1)*L$$

基地址



线性表的顺序存储结构示意图

顺序存储结构的线性表类C语言表示：

由于C语言中的一维数组也是采用顺序存储表示，故可以用数组类型来描述顺序表。又因为除了用数组来存储线性表的元素之外，顺序表还应该用变量来表示线性表的长度属性等，所以我们用结构类型来定义顺序表类型。

```
#define LIST_INIT_SIZE  100  //存储空间的初始分配量
#define LISTINCREMENT  10 //分配增量

typedef struct {
     ElemType *elem; //存储空间基址
     int length;            //当前长度
     int listsize;         //当前分配的存储容量,以一数据元素存储
                            //长度为单位
}SqList;
```

顺序表类型如果这样表示：

```
#define MaxSize 100

typedef struct {

    ElemType  data[MaxSize]; //存放线性表的元素

    int length;              //线性表长度

}SqList;
```

- 数组指针elem指示顺序表的基地址。
- ElemType 数据类型是为了描述统一而自定的，在实际应用中，可根据实际需要具体定义表中元素的数据类型。

为了简单，假设ElemType为int类型，则可以使用以下自定义类型语句。

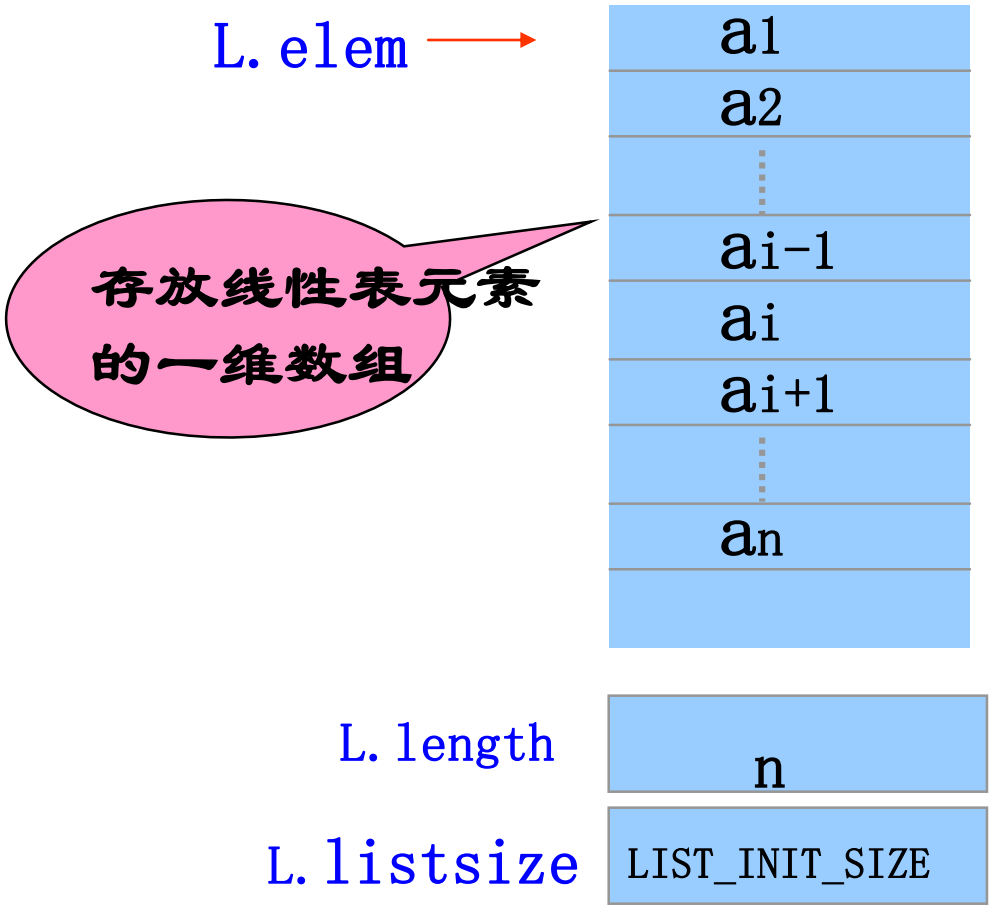
```
typedef  int ElemType;
```

有了上述定义后，可以通过变量定义语句

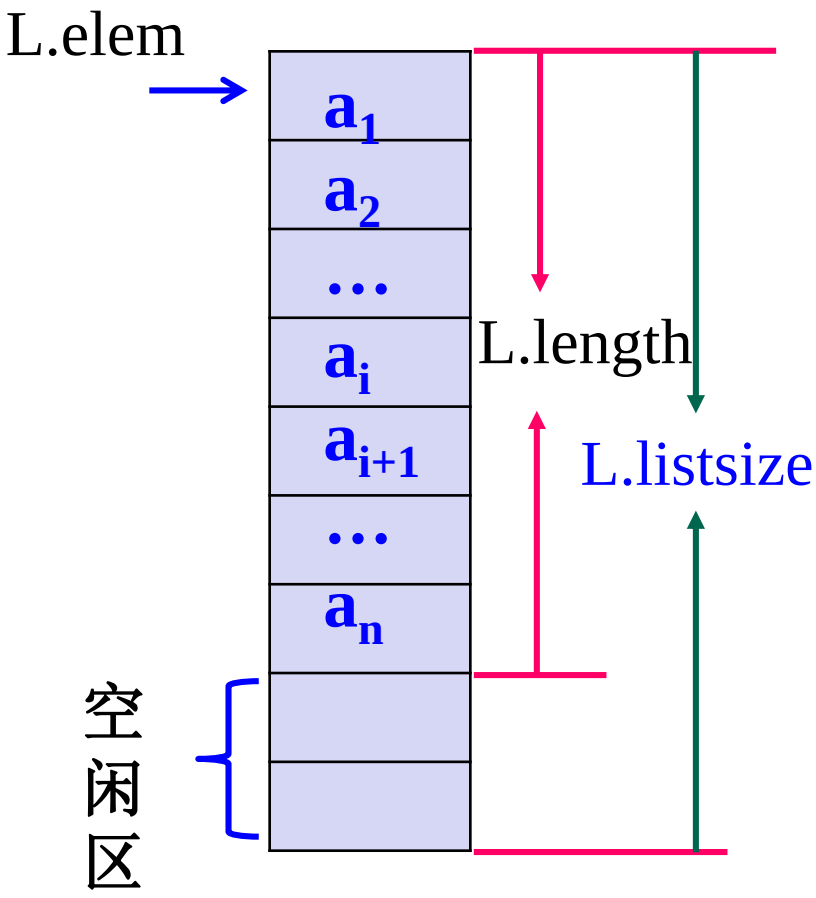
```
SqList  L;
```

将L定义为SqList类型的变量，便可以利用L.elem[i-1]访问表中位置序号为i的数据元素。

设A=(a_1, a_2 , a_3 , . . . a_n)是一线性表，L是SqList 类型的结构变量，用于存放线性表A，则L在内存中的状态如图所示：



顺序表图示


L.elem
a_1
a_2
…
a_i
a_{i+1}
…
a_n
L.length
L.listsize
空闲区

2.2.2 顺序表基本操作的实现

在顺序表存储结构中，很容易实现线性表的一些操作，如第i个元素的访问、线性表的构造。

注意：C语言中的数组下标从“0”开始，因此，若L是SqList类型的顺序表，则表中第i个元素是L.elem[i-1]。

在介绍基本操作的算法之前，先回顾一下本书算法中常用到的两个C函数

1） void *malloc(int size)　　if p==0分配失败

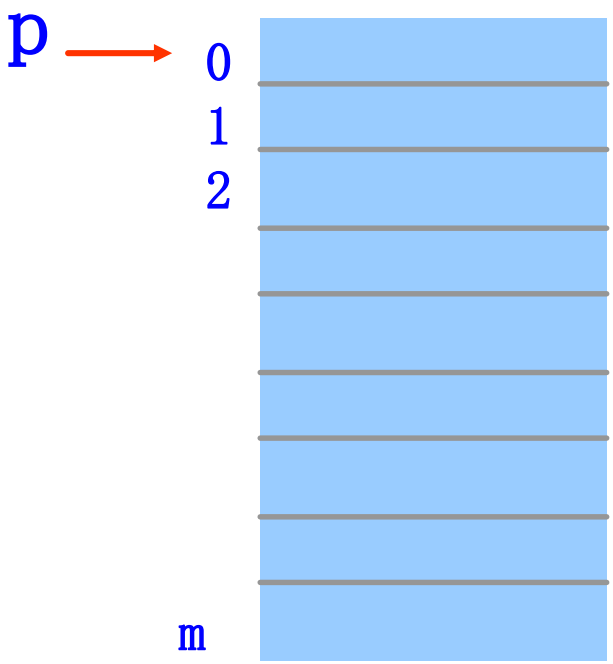
功能：在系统内存中分配size个的存储单元，并返回该空间的基址

使用方法：

```
...
int m = 100,
float  *p;
p = (float*) malloc(m*sizeof(float ));
```

sizeof操作符以字节形式给出了其操作数的存储大小

执行语句p = (float*) malloc(m*sizeof(float))，计算机将按float 类型变量所占空间的大小（一般为32bit）分配m*sizeof(float)个的存储单元,并将其基址赋值给指针变量p;



p = (float*) malloc(m*sizeof(float)) 图示

2) void free (p)

功能：将指针变量p所指示的存储空间，回收到系统内存空间中去；

使用方法：

调用free (p)

```
...
 int m = 100;                                   p ⟶
 float   *p;
  p = (float*) malloc(m*sizeof(float));
 // 一旦p所指示的内存空间不再使用
 //调用free( ) 回收之
free(p);
```

调用free (p) 图示

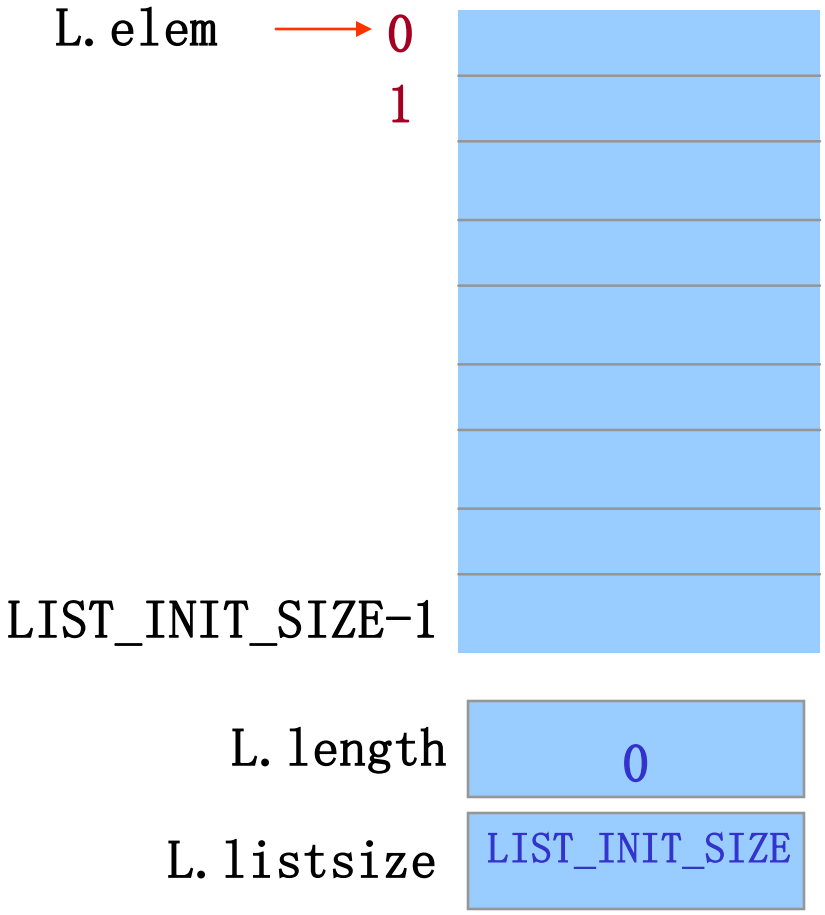
1. 顺序表的初始化操作 InitList_Sq(SqList &L)

参数：L是存放线性表的结构变量（称L为顺序表）, 因为初始化操作对顺序表L进行了修改,所以用了引用参数L;

功能：建立空的顺序表L

主要步骤：调用malloc ( )为顺序表分配一预定大小(LIST_INIT_SIZE）的空间,并将其基址赋值给L.elem;

初始化操作演示



顺序表初始化

初始化操作算法(算法2.3)

引用参数

```
Status InitList (SqList &L)
{   //构造一个空的顺序表L

L.elem=(ElemType*)malloc(LIST_INIT_SIZE*sizeof(ElemType));
     if   (! L.elem)   exit  (OVERFLOW);  //存储分配失败
     L. length=0;                          //空表长度为0
     L.listsize= LIST_INIT_SIZE;          //初始存储容量
    return OK;
}
```

2 销毁操作 DestroyList (SqList &L) {

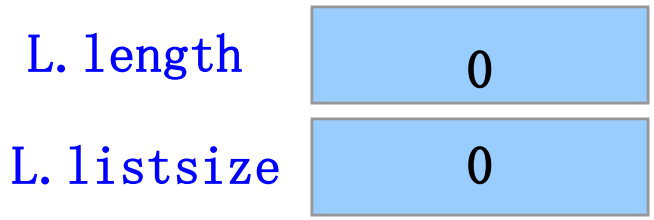
功能：　　回收为顺序表动态分配的存储空间

主要步骤：　调用free( ), 回收为顺序表动态分配的存储空间

销毁操作图示

L.elem = NULL

销毁顺序表



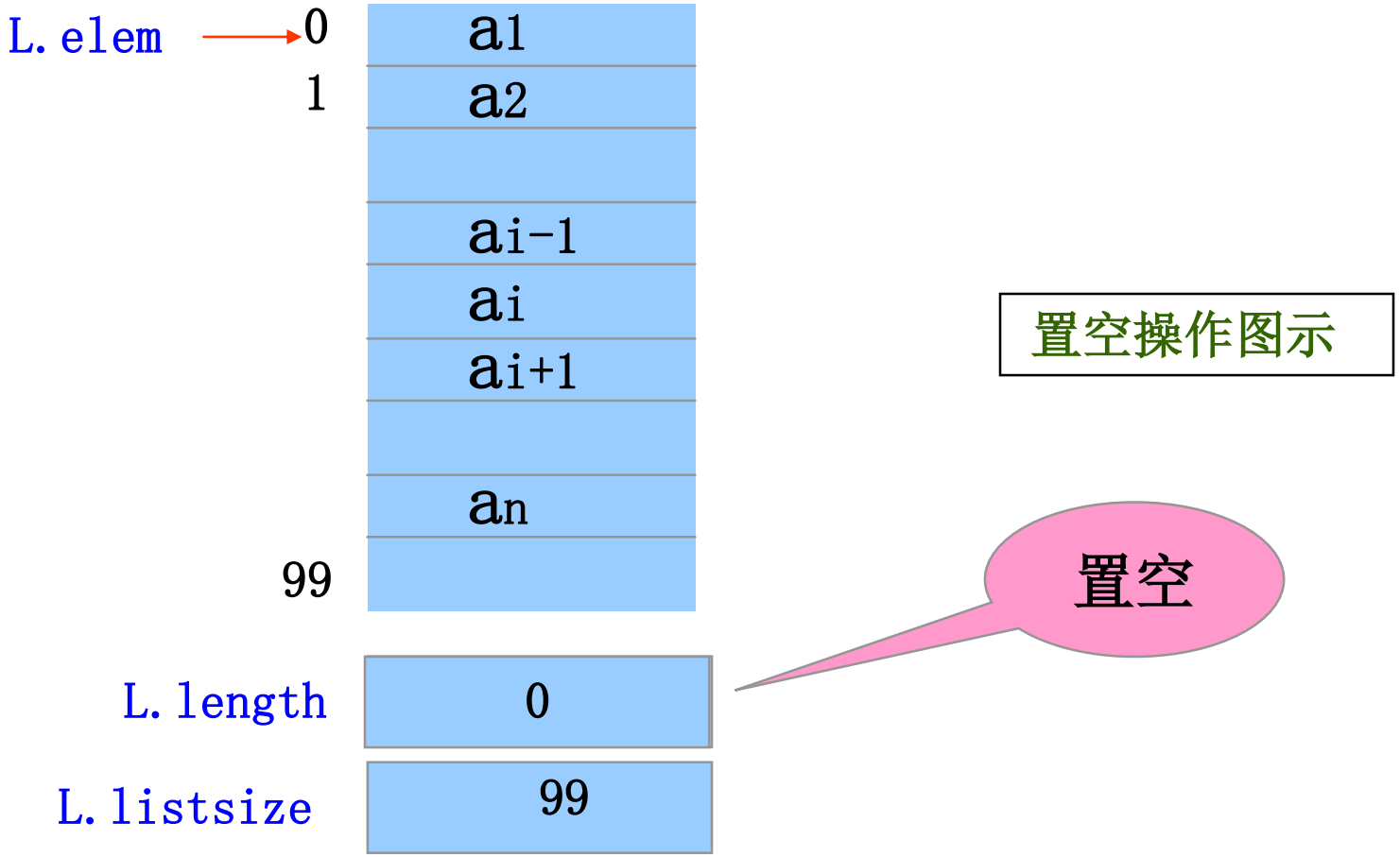
销毁操作算法：

```
Status  DestroyList ( SqList &L)
{
        If (!L.elem)   return ERROR;  // 若表L不存在
        free (L.elem);  // 若表L已存在，回收动态分配的存储空间
        L.elem = NULL;
        L.length = 0;
        L.Listsize = 0;
        return OK;
}
```

3、置空操作ClearList_Sq (SqList &L)

功能：若L已存在，重新将其置成空表；

置空操作图示


L.elem
0
1
99
a1
a2
ai-1
ai
ai+1
an
L.length
0
L.listsize
99
置空
置空操作图示

置空操作算法：

```
Status ClearList ( SqList &L)
{
        If (!L.elem) return ERROR;  // 若表L不存在
        L.length = 0;                          //若表L已存在，将L置空
        return OK;
}
```

以下主要讨论线性表的插入和删除两种运算。

1、插入

线性表的插入运算是指在表的第i (1≦i≦n+1个位置上，插入一个新结点x，使长度为n的线性表

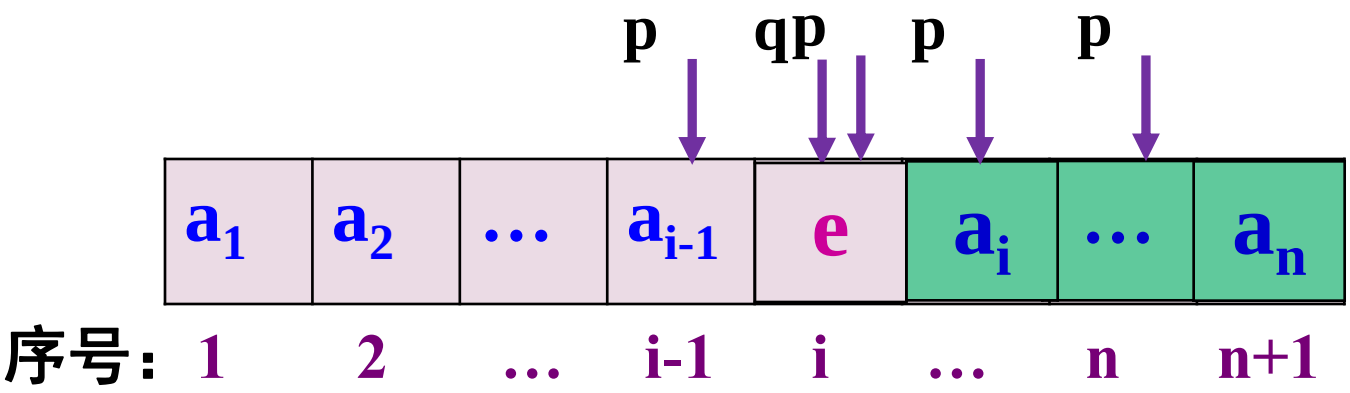
$(a_1, \ldots a_{i-1}, a_i, \ldots, a_n)$

变成长度为n+1的线性表

$(a_1, \ldots a_{i-1}, x, a_i, \ldots, a_n)$

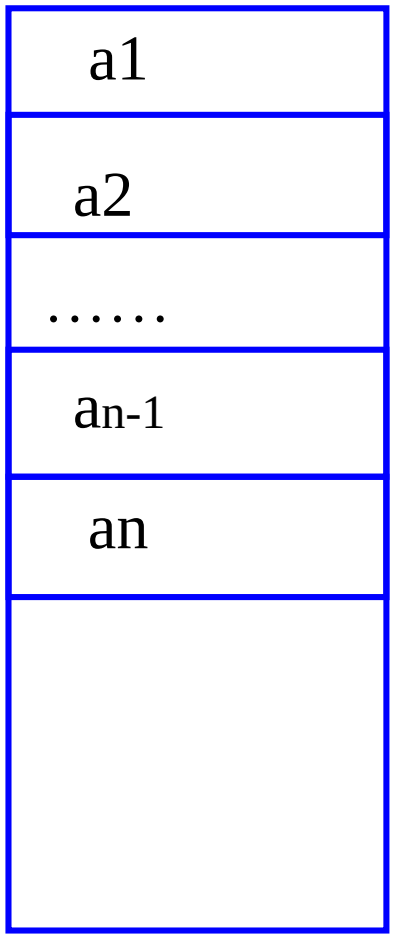
```
Status ListInsert (SqList &L, int i, ElemType e)
{// 在顺序线性表L的第i个元素之前插入新的元素e，1 ≤ i ≤ L.length+1

    if (i < 1 || i > L.length+1)  return ERROR;  // 插入位置不合法

    if (L.length >= L.listsize) {       // 当前存储空间已满，增加分配
        newbase = (ElemType *)realloc(L.elem,
                  (L.listsize+LISTINCREMENT)*sizeof (ElemType));
        if (!newbase) exit(OVERFLOW);    // 存储分配失败
        L.elem = newbase;           // 新基址
        L.listsize += INCREMENT ;  // 增加存储容量

      }
```

```
    q = &(L.elem[i-1]);     // q指示插入位置
    for (p = &(L.elem[L.length-1]); p >= q; --p)
          *(p+1) = *p;          // 插入位置及之后的元素右移
    *q = e;     // 插入e
     ++L.length; // 表长增1
     return OK;
}
```



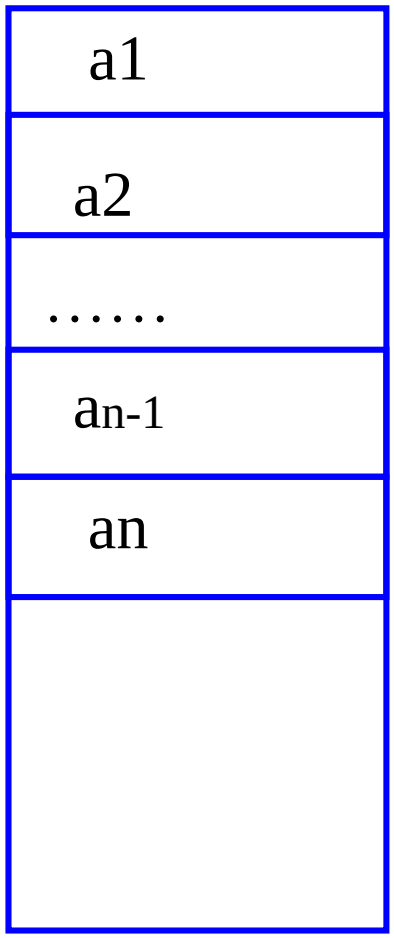
编写算法实现顺序表操作Locate(L, x).

若L中存在数据元素x，则返回x在顺序表L中第一次出现的位序；否则返回0.



编写算法实现顺序表操作Locate(L, x).

若L中存在数据元素x，则返回x在顺序表L中第一次出现的位序；否则返回0.



现在分析插入算法的时间复杂度:

这里的问题规模是表的长度n。该算法的时间主要花费在for循环语句上，所需移动结点的次数不仅依赖于表的长度n，而且还与插入位置i有关。

A.当i=L.length+1时，不需移动数据元素；这是最好情况，其时间复杂度O（1）；

B.当i=1时，需移动表中所有结点，这是最坏情况，其时间复杂度为O（n）。

由于插入可能在表中任何位置上进行，因此需分析算法的平均复杂度。

算法的平均复杂度：

$a_1, \ldots, a_{i-1}, a_i, \ldots, a_n$

设p_i为在第i个位置插入一个元素的概率，等概率情况下

$$p_i = \frac{1}{n+1}$$

此时需要移动$n-i+1$个元素。

因此在长度为n的线性表中插入一个元素时所需移动元素的平均次数为：

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

因此算法的平均时间复杂度为$O(n)$。

2、删除

线性表的删除运算是指将表的第$i(1 \leqq i \leqq n)$结点删除，使长度为n的线性表：

$$(a_1, \ldots a_{i-1}, a_i, a_{i+1} \ldots, a_n)$$

变成长度为$n-1$的线性表

$$(a_1, \ldots a_{i-1}, a_{i+1}, \ldots, a_n)$$

序号　　数据元素

1

2

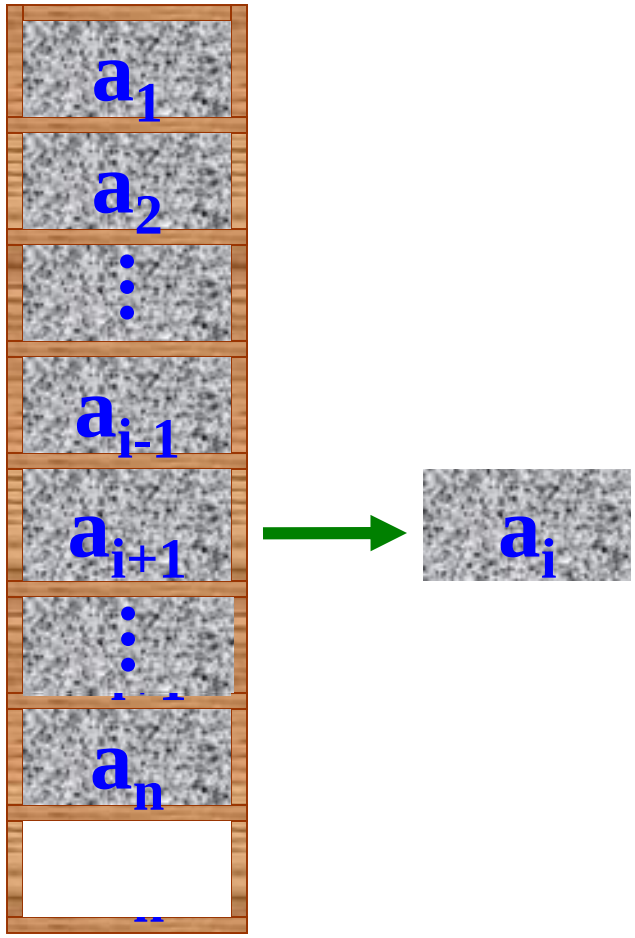
⋮

i-1

i

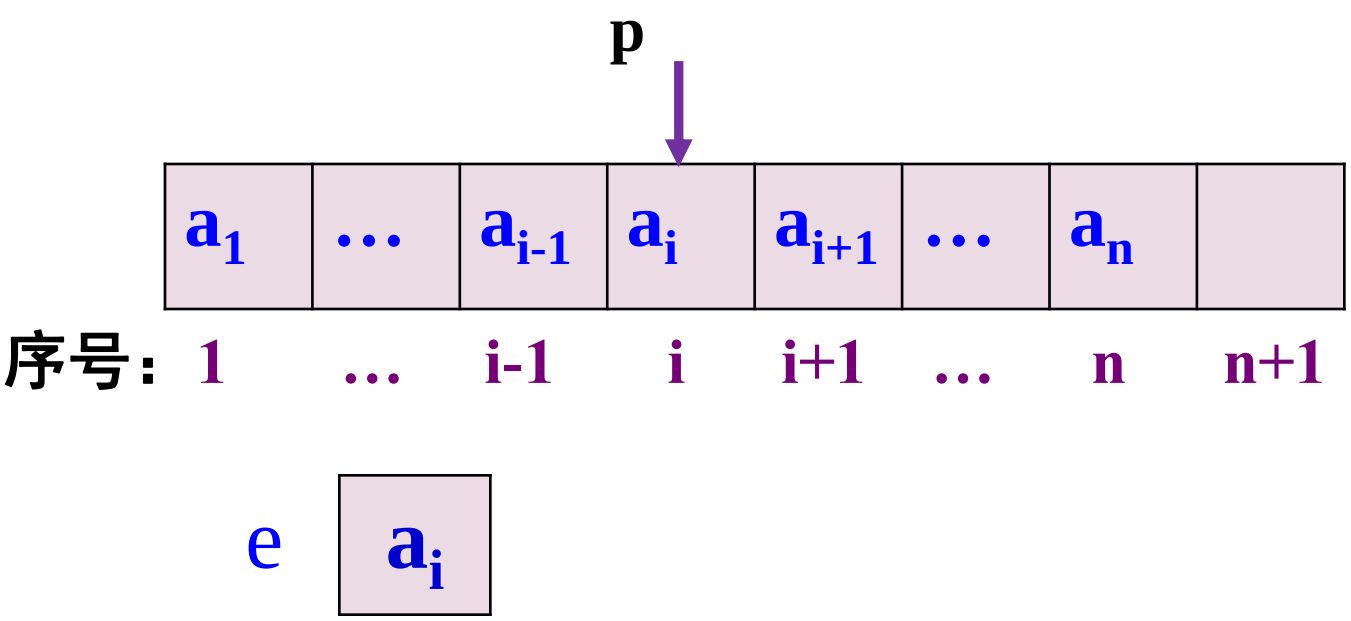
⋮

n-1

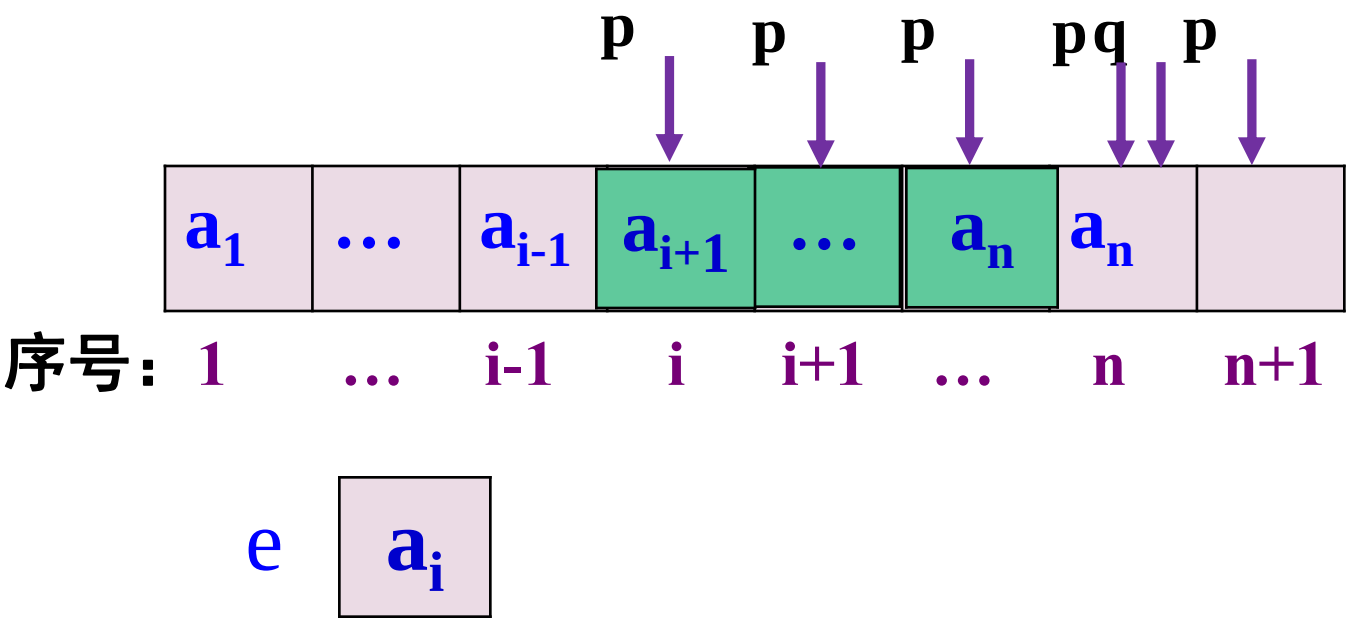


顺序表的删除算法:

```
Status ListDelete (SqList &L, int i, ElemType &e)
{// 在顺序表L中删除第i个元素，并用e返回其值，1≤i≤L.length
    if ((i < 1) || (i > L.length)) return ERROR; // 删除位置不合法
    p = &(L.elem[i-1]);      // p为被删除元素的位置
    e = *p;       // 被删除元素的值赋给e
```



```
    q = L.elem+L.length-1; // 表尾元素的位置

    for (++p; p <= q; ++p)     *(p-1) = *p;
                                    // 被删除元素之后的元素左移
     --L.length; // 表长减1

   return OK;
}
```



算法分析

本算法元素移动的次数也与表长n和删除元素的位置i有关:

- 当i=n时，无需移动结点，时间复杂度为O(1) ；
- 当i=1时，移动次数为n-1，时间复杂度为O(n)。

算法的平均复杂度：

$a_1, \ldots, a_{i-1}, a_i, \ldots, a_n$

设p_i为删除第i个元素的概率，等概率情况下

$$p_i = \frac{1}{n}$$

此时需要移动$n-i$个元素。

因此在长度为n的线性表中删除一个元素时所需移动元素的平均次数为：

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

因此算法的平均时间复杂度为$O(n)$。

即在顺序表上做删除运算，平均要移动表中约一半的结点，平均时间复杂度也是O(n)。

线性表的合并

已知两个线性表LA和LB中的数据元素按值非递减有序排列，要求将 LA和LB归并为一个新的线性表LC，且LC中的数据元素仍按值非递减有序排列。

例如，设

LA=(3, 5, 8, 11)

LB=(2, 6, 8, 9, 11, 15, 20)

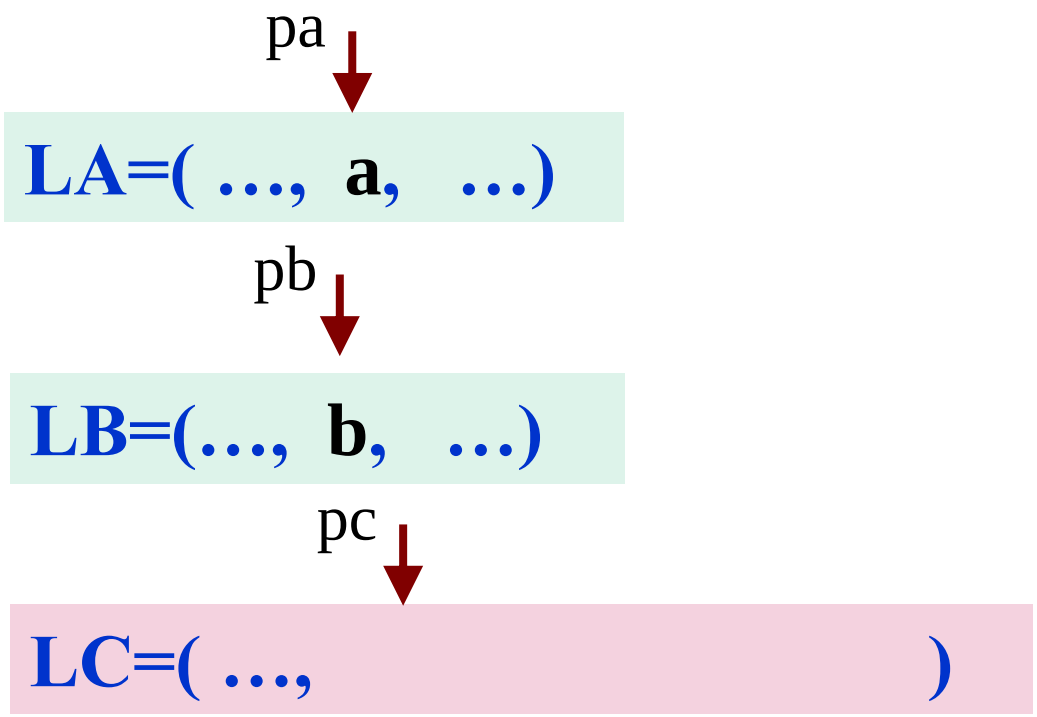
则

LC=(2, 3, 5, 6, 8, 8, 9, 11, 11, 15, 20)

由问题要求可知，LC中的数据元素或是LA中的元素，或是Lb中的元素，因此只要先设LC为空表，然后将LA或LB中的元素逐个插入到LC中即可。

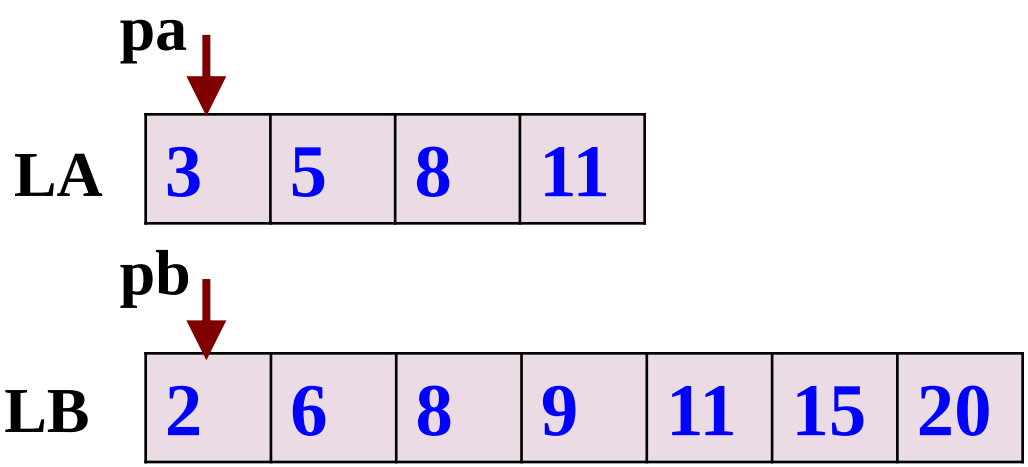
为使LC中的元素按值非递减有序排列，可设两个指针pa和pb分别指向LA和LB中的某个元素，若设pa当前所指的元素为a，pb当前所指的元素为b，则当前应插入到LC中的元素c为

$$c = \begin{cases} a & \text{当} a \le b \text{时} \\ b & \text{当} a > b \text{时} \end{cases}$$

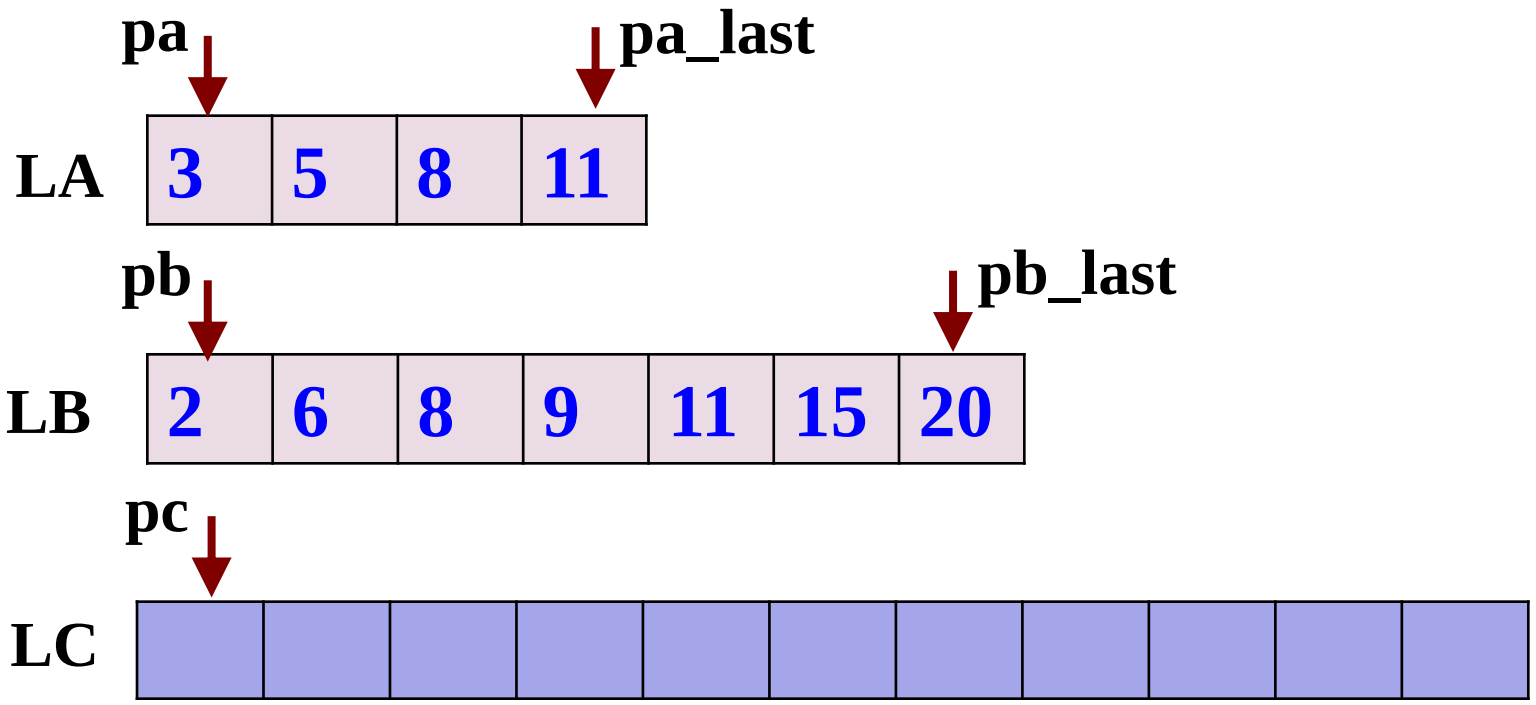


(1). 两个顺序有序表的合并

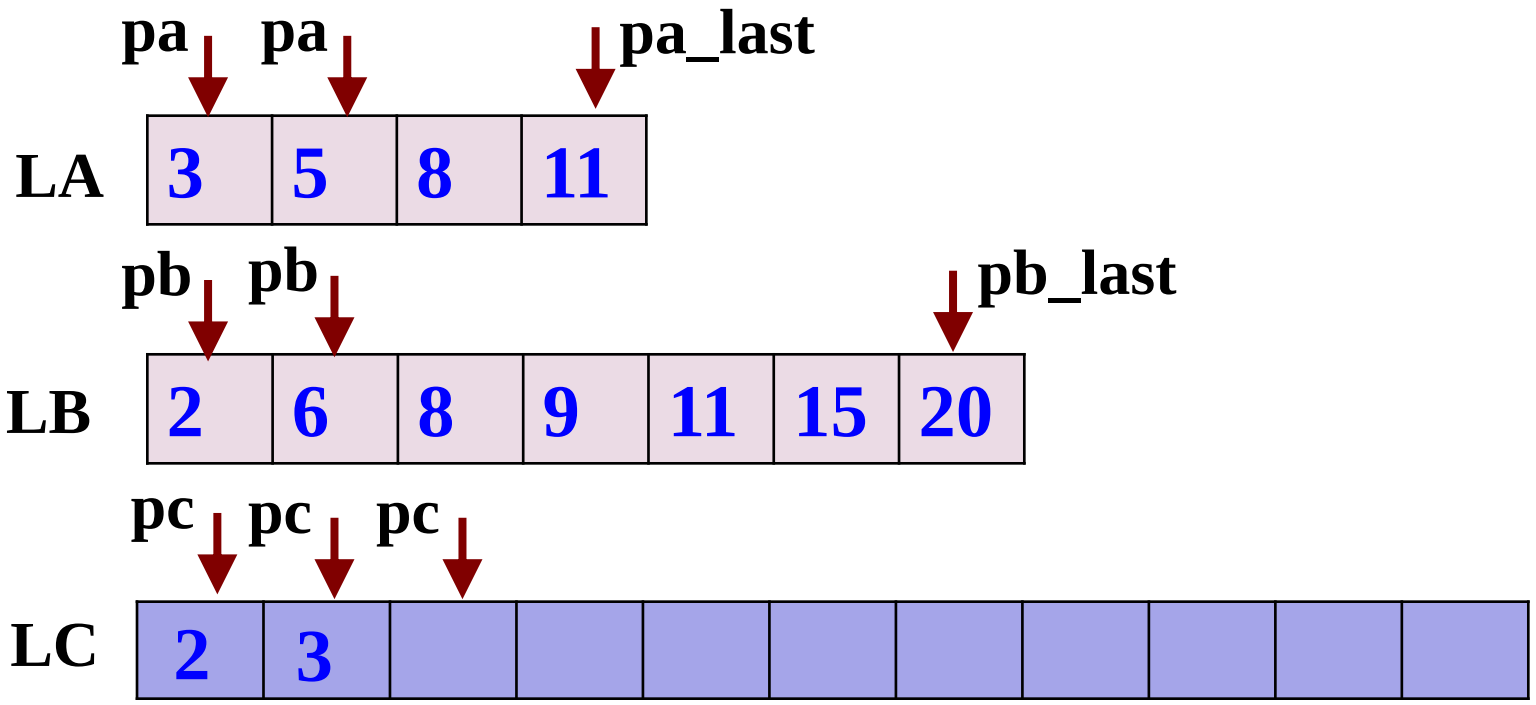
```
void MergeList_Sq(SqList la, SqList lb, SqList &lc)
 { // 已知顺序表La和Lb的元素按值非递减排列，归并La和Lb得
  //到新的顺序表Lc，Lc的元素也按值非递减排列。
      pa = la.elem;
      pb = lb.elem;      //pa和pb的初值分别指向表的第一个元素
```



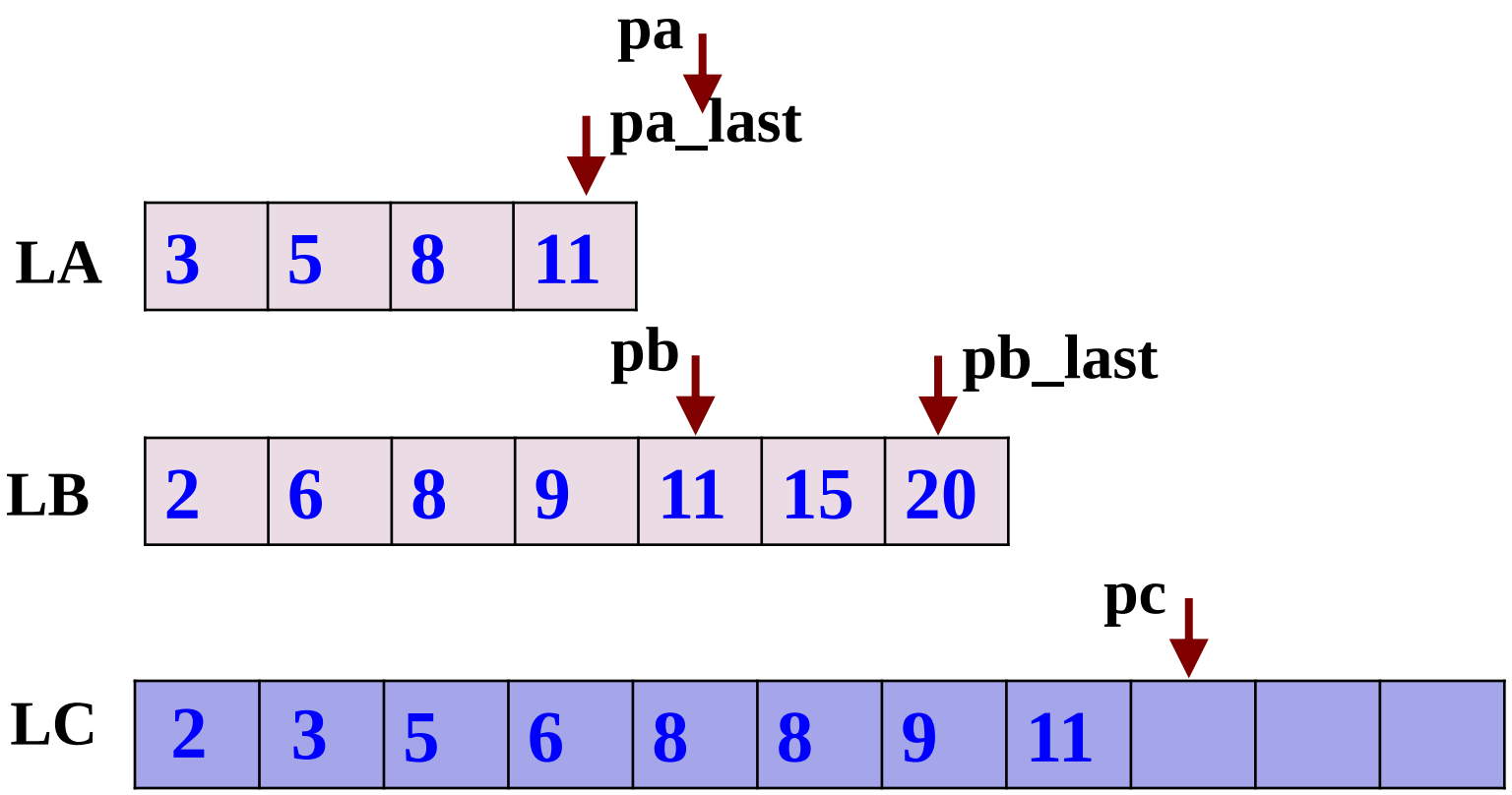
```
lc.listsize =lc.length= la.length + lb.length;
                                    //LC表的长度为LA和LB两表的长度之和
pc = lc.elem =  (ElemType*)malloc(sizeof(ElemType)*lc.listsize);
                                    //为LC表分配空间
pa_last = la.elem + la.length - 1;
                                    //指针pa_last指向LA的最后一个元素
pb_last = lb.elem + lb.length - 1;
                                    //指针pb_last指向LB的最后一个元素
```



```
while(pa <= pa_last && pb <= pb_last)   //La和LB均未到达表尾
{    if  (*pa <= *pb) //依次取两表中值较小的元素插入到LC的最后
          *pc++ = *pa++;
     else
          *pc++ = *pb++;
}
```



```
    while(pa <= pa_last)    *pc++ = *pa++;
        //LB已到表尾，依次将LA的剩余元素插入到LC的最后
    while(pb <= pb_last)    *pc++ = *pb++;
        //LA已到表尾，依次将LB的剩余元素插入到LC的最后
}
```



如果两个表长分别记为m和n，则算法的时间复杂度为O(m+n)。

```
void MergeList_Sq(SqList la, SqList lb, SqList &lc)
{ // 已知顺序表La和Lb的元素按值非递减排列。与P21算法2.2类似
  // 归并La和Lb得到新的顺序线性表Lc，Lc的元素也按值非递减排列。
    pa = la.elem;
    pb = lb.elem;
    lc.listsize =lc.length= la.length + lb.length;
    pc = lc.elem = (ElemType*)malloc(sizeof(ElemType)*lc.listsize);
    if (! lc.elem) exit(OVERFLOW);
    pa_last = la.elem + la.length - 1;
    pb_last = lb.elem + lb.length - 1;
    while(pa <= pa_last && pb <= pb_last)   //归并
     {    if(*pa <= *pb)    *pc++ = *pa++;
          else      *pc++ = *pb++;
     }
   while(pa <= pa_last)    *pc++ = *pa++; //插入la的剩余元素
   while(pb <= pb_last)    *pc++ = *pb++; //插入lb的剩余元素
 }
```

小　结

顺序表的特点：

1　通过元素的存储顺序反映线性表中数据元素之间的逻辑关系；

2　可随机存取顺序表的元素；

3　顺序表的插入删除操作要通过移动元素实现。

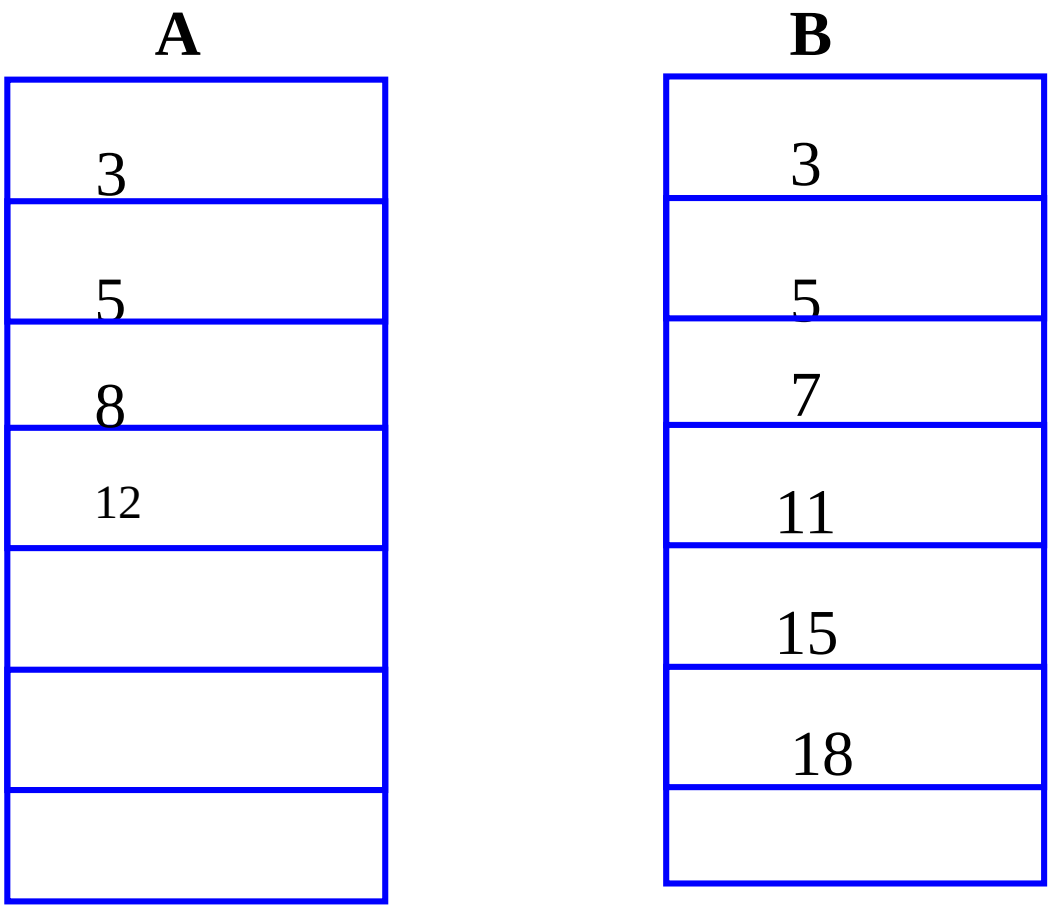
第二章　　习题一

习题取自

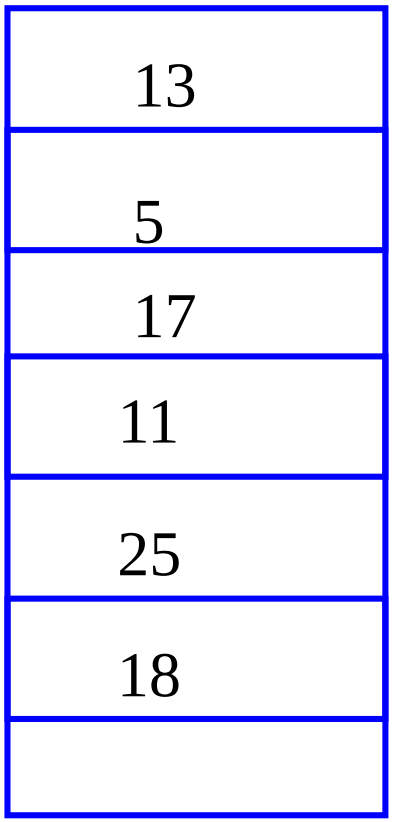
数据结构题集(C语言版)　严蔚敏等编　清华大学出版社

1. P17　　2. 12
2. P18　　2. 21

2.12设A=(a1,...,am)和B=(b1,...,bn)均为顺序表，A’和B’分别为A和B中除去最大共同前缀后的子表，若A’=B’=空表，则A=B;若A’=空表，而B’!=空表，或者两者都不为空表，且A’的首元小于B’的首元，则A<B;否则A >B.试写一个比较A,B大小的算法。



2.21试写一算法，实现顺序表的就地逆置，即利用原表的存储空间将线性表(a1, a2, ..., an)逆置为(an, an-1, ..., a1).



2.3 线性表的链式表示和实现

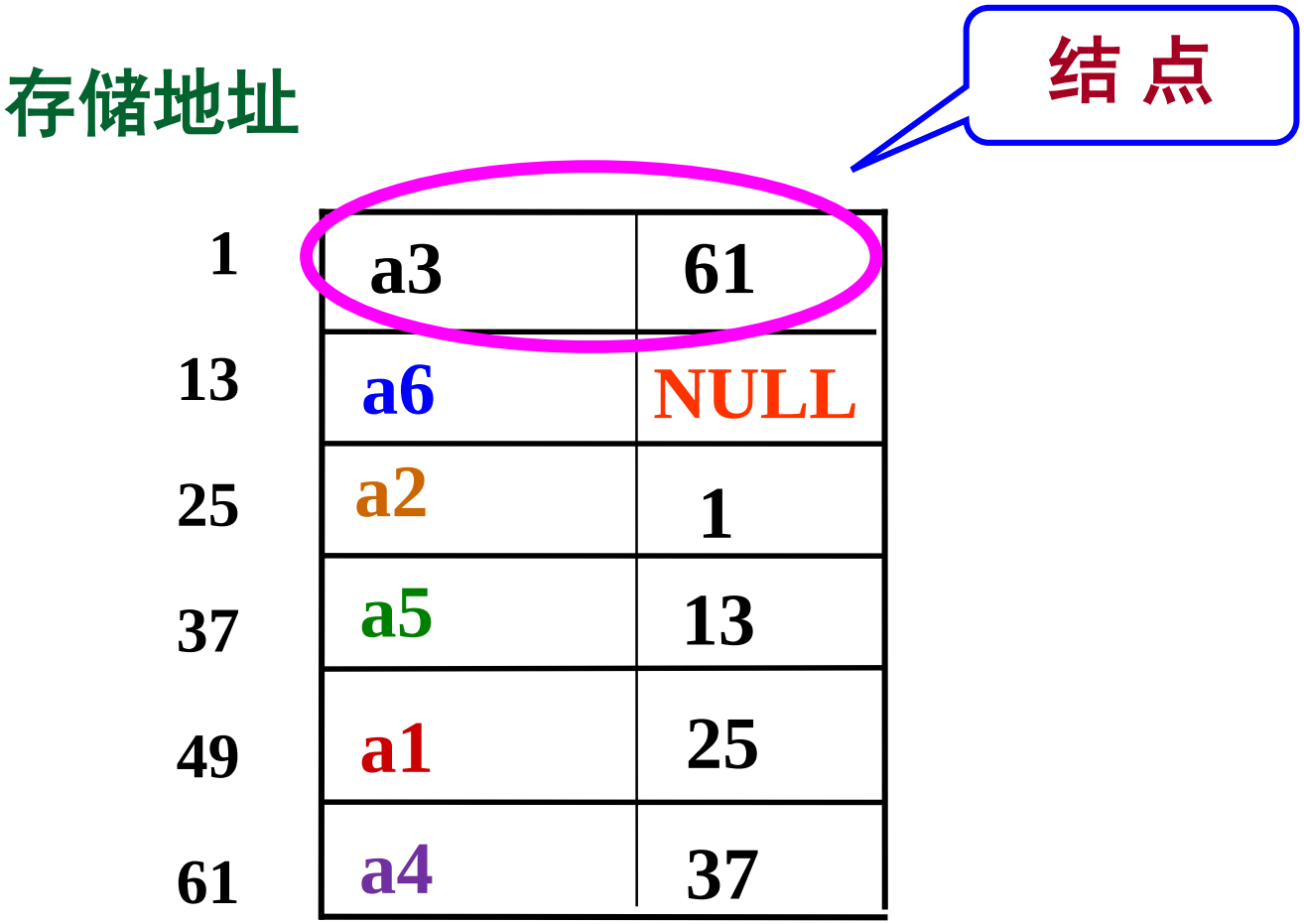
2.3.1 单链表的定义和表示

线性表链式存储结构的特点：

用一组任意的存储单元来存放线性表的数据元素。

如线性表：

（a_1，a_2，a_3，a_4，a_5，a_6）的单链表示意图如下：

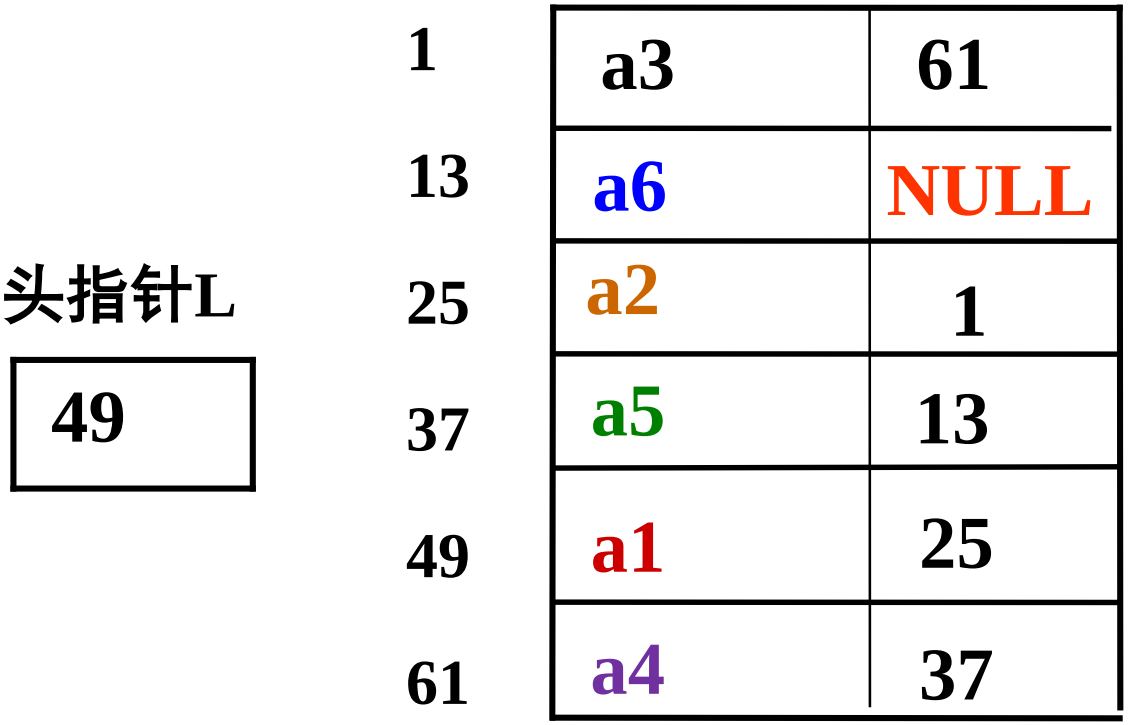


为了能正确表示数据元素间的逻辑关系，在存储每个数据元素其本身信息的同时，还必须存储一个指示其直接后继的信息。这两部分信息组成数据元素的存储映象，称为结点。它包括两个域：

数据域：存储数据元素信息的域。
指针域：存储直接后继存储位置的域。

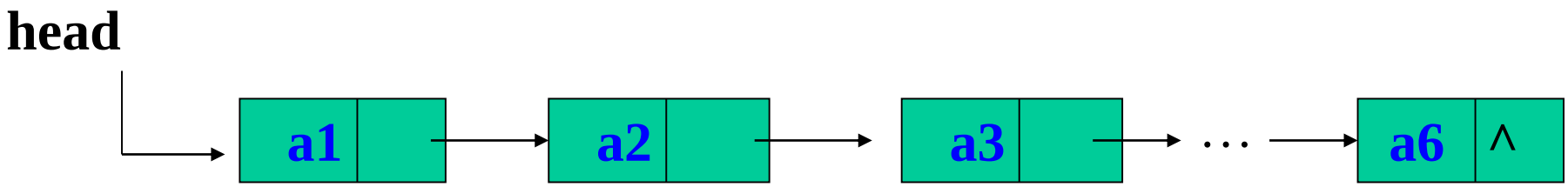
链表正是通过每个结点的指针域将线性表的n个结点按其逻辑次序链接在一起的。由于上述链表的每一个结点中只包含一个指针域，故将这种链表称为单链表（Single Linked）。

例、线性表(a_1，a_2，a_3，a_4，a_5，a_6)的单链表示意图如下：



显然，单链表中每个结点的存储地址是存放在其前趋结点指针域中，而第一个无前趋，故应设头指针head指向第一个结点。同时，由于最后一个结点无后继，故最后一个结点的指针域为空，即NULL（图示中也可用^表示）。

通常我们把链表画成用箭头相链接的结点的序列，结点之间的箭头表示链域中的指针。如：



例如：若头指针名是head，则把链表称为表head。

单链表可由头指针唯一确定，因此单链表可以用头指针的名字来命名。

上面用自然语言描述线性表的一种链式存储结构——线性链表，怎样在计算机上实现线性链表？显然，可以用C语言的结构体表示线性链表的结点，可以用指针存放直接后继的存储地址。

单链表的结点类型声明如下:

```
typedef   struct  Node {
        ElemType          data;        // 数据域
        struct   Node     *next;       // 指针域
}LNode,  *LinkList;    //LinkList为指向结构体LNode的指针类型
```

·LNode: 结构体类型名;

LNode类型结构体变量用于表示线性链表中的一个结点;

·LinkList: 指针类型名;

LinkList类型指针变量用于存放LNode类型结构体变量的地址;

注意区分指针变量和结点变量两个不同的概念，若定义:

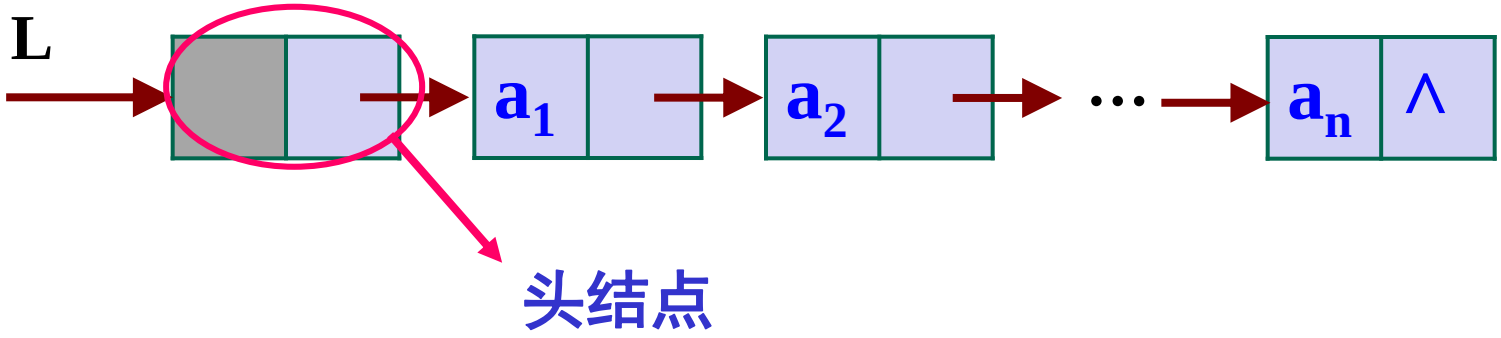
LinkList p;

或 LNode *p;

则 p为指向某结点的指针变量，表示该结点的地址；

*p为对应的结点变量，表示该结点的名称。

带头结点的单链表：



头结点：

在单链表的第一个结点之前附设的一个结点，它的数据域可以放数据元素，也可以放线性表的长度等附加信息，也可以不存储任何信息。

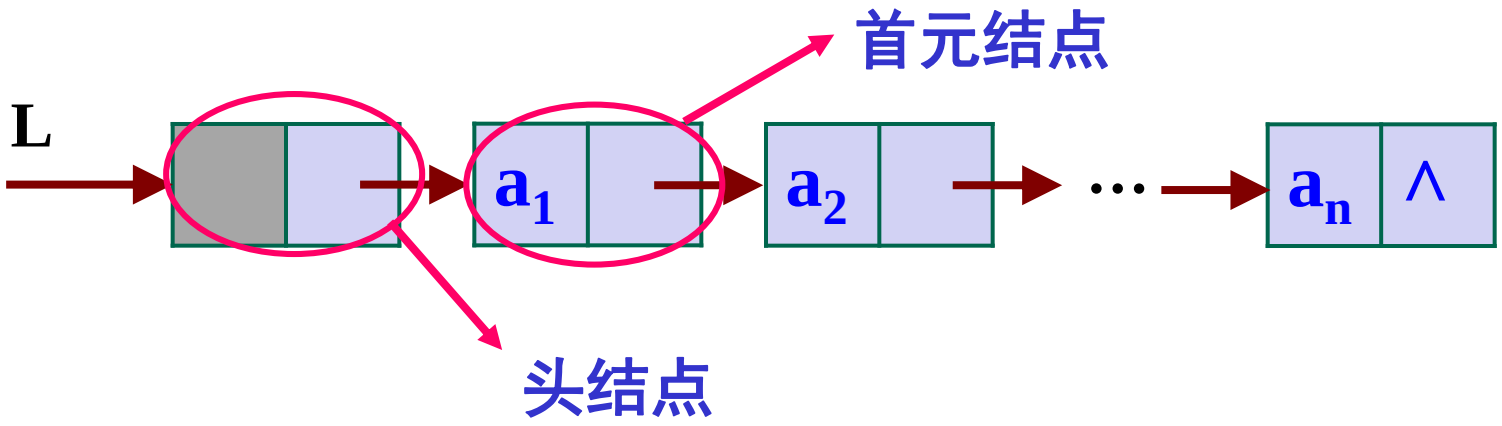
头指针、首元结点、头结点：

头指针：

是指向链表中第一个结点的指针。若链表设有头结点，则头指针所指结点为单链表的头结点。如下图的L指针。

首元结点：指链表中存储第一个数据元素的结点。如下图的中存储数据元素a1的结点。

头结点：在首元结点之前附设的一个结点。



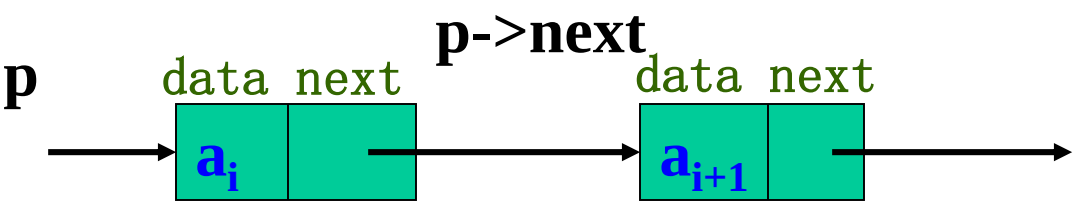
单链表增加一个头结点的优点：

- 首元结点的操作和表中其他结点的操作相一致，无需进行特殊处理；
- 无论链表是否为空，都有一个头结点，因此空表和非空表的处理也就可以统一。

设p是指向线性表中第i个数据元素(结点a_i) 的指针，则p->next是指向第个i+1个数据元素(结点a_{i+1}) 的指针。即

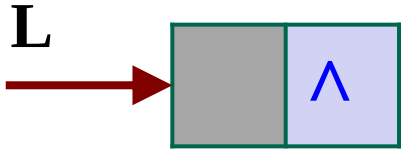
若　　p->data= a_i

则　　p->next->data= a_{i+1}



2.3.2单链表基本操作的实现

1. 单链表的初始化 InitList (&L)

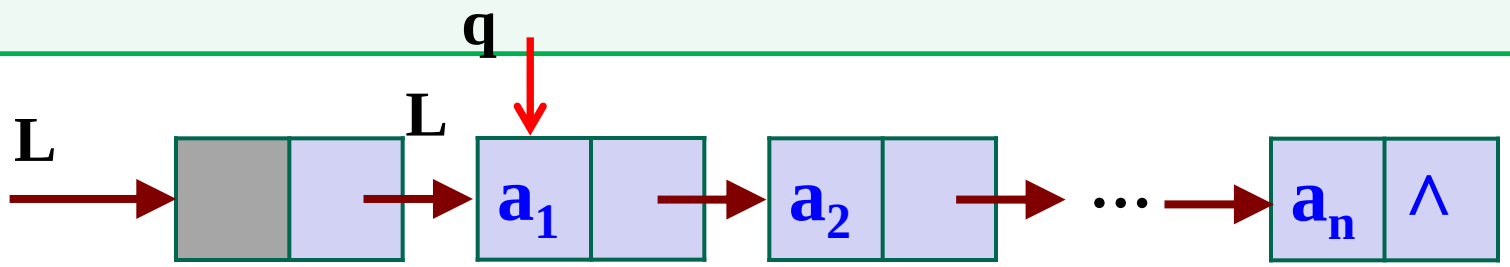



```
void InitList(LinkList &L)
{ //构造一个空的单链表L
   L=(LinkList)malloc(sizeof(LNode));
        //生成头结点，并使L指向此头结点
   L→next=NULL;
}
```

2. 销毁单链表DestroyList(&L)

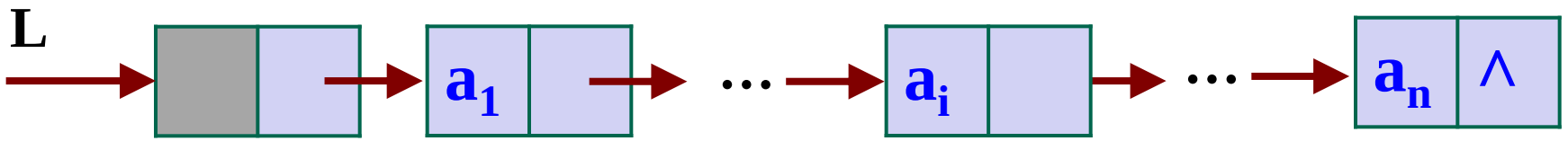
调用free函数，回收链表的所有结点的存储空间。

```
void DestroyList(LinkList &L)
{   while (L)
    {
        q=L->next;
        free(L);   //  回收L结点的存储空间
        L=q;
    }
}
```

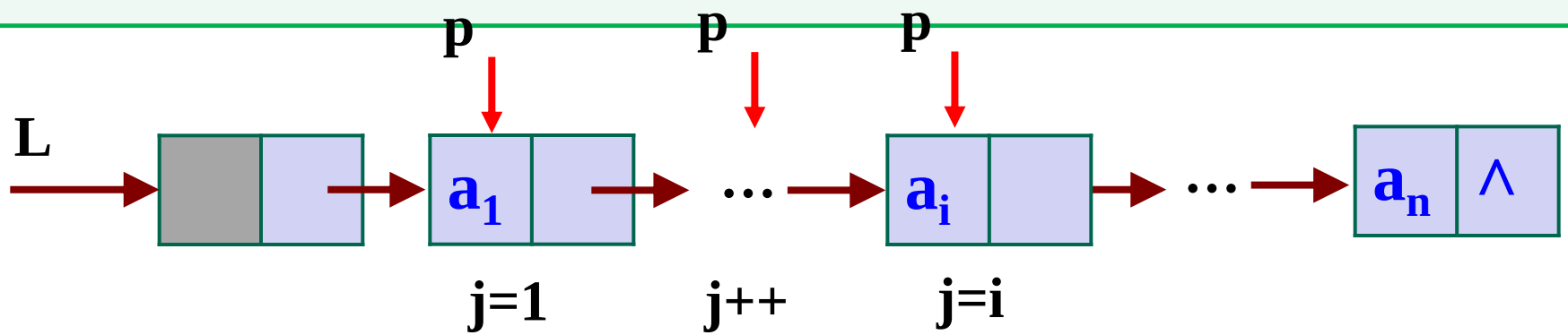


3. 单链表的取值GetElem (L, i, &e)

在链表中，根据给定的结点位置序号i，在链表中获取该结点的值只能从链表的头结点出发，顺着链域next逐个结点往下访问。



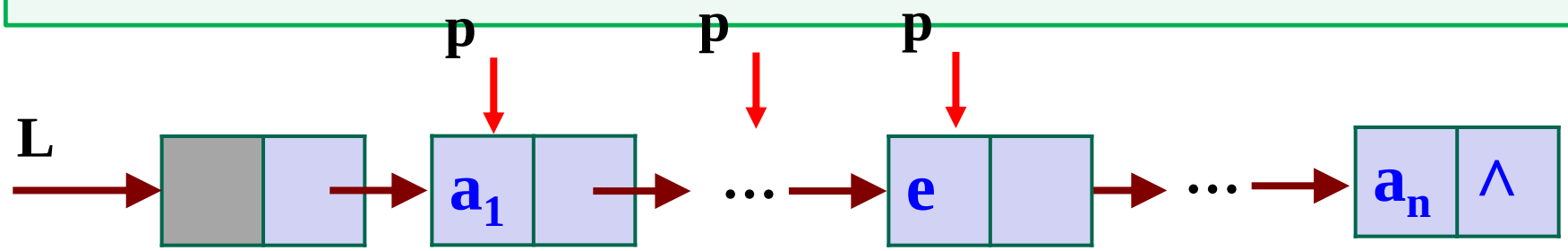
```
Status GetElem (LinkList L, int i, ElemType &e)
{// 在带头结点的单链表L中根据序号获取元素的值，用e返回L中第i个元素的值
    p = L->next; j = 1;    // 初始化，p指向第一个结点，j为计数器
    while (p && j<i) {     // 顺指针向后查找，直到p指向第i个元素或p为空
        p = p->next; ++j;  }
    if ( !p || j>i )    return ERROR; // i值不合法，i>n或i ≤0
    e = p->data;   // 取第i个结点的数据域
    return OK;
}
```



4. 查找LocateElem (L, e)

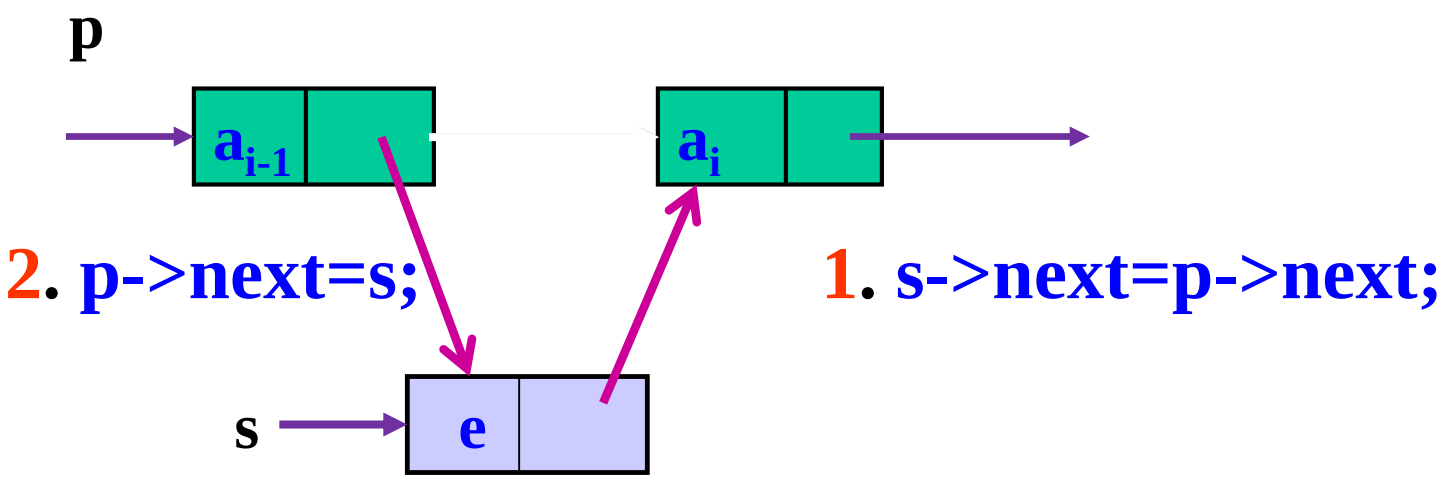
从链表的首元结点(第一个元素结点)开始，依次将结点值和给定值e进行比较，返回查找结果。

```
Lnode *LocateElem (LinkList L, ElemType e)
{    // 在带头结点的单链表L中查找值为e的元素
     p = L->next;      //p指向首元结点
     while (p && p->data!=e)
      // 顺指针域向后查找，直到p为空或p所指结点的数据域为e
          p = p->next;
     return p;  //查找成功返回值为e的结点地址p，查找失败p为NULL
}
```



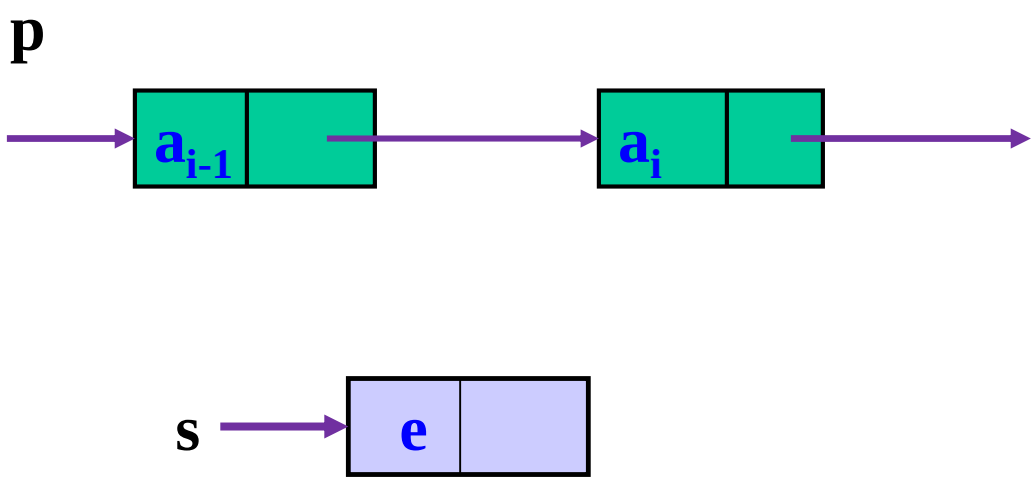
5. 插入数据元素ListInsert (&L, i, e)

假设要在数据元素a_{i-1}和a_i之间插入一个数据元素e，p为指向结点a_{i-1}的指针。

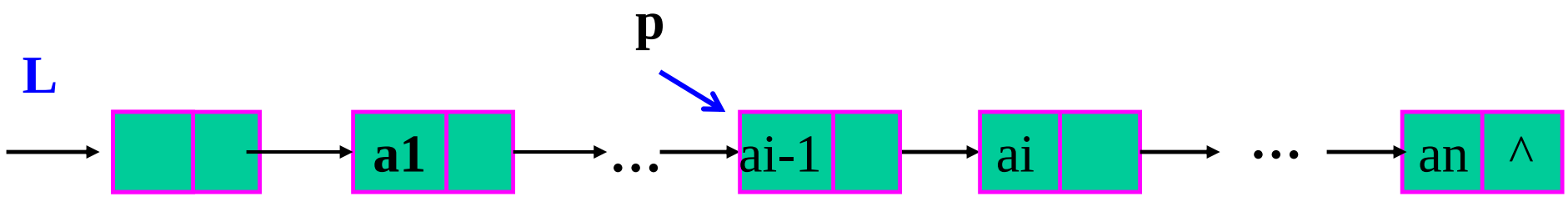


5. 插入数据元素ListInsert (&L, i, e)

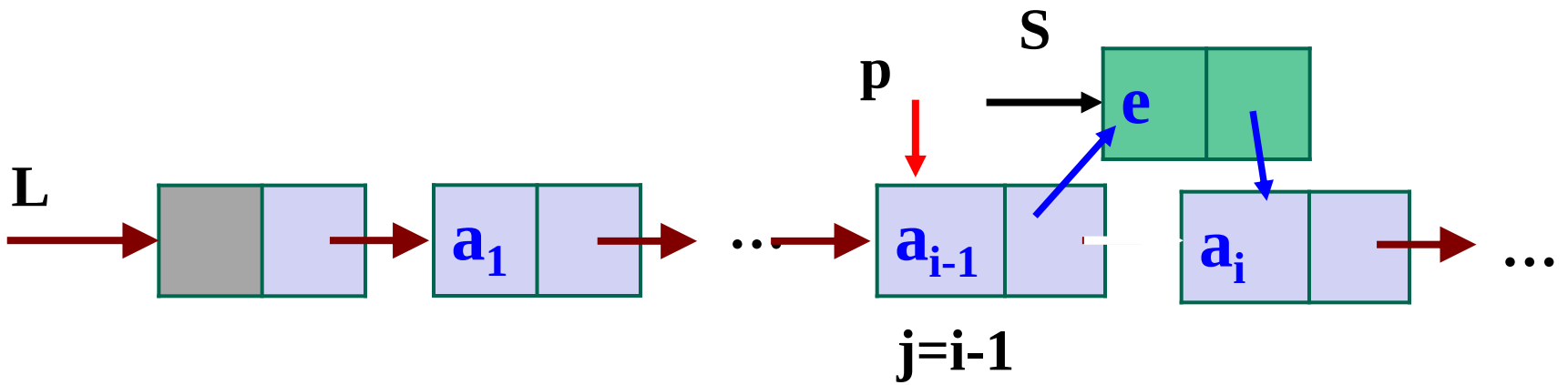
假设要在数据元素a_{i-1}和a_i之间插入一个数据元素x，p为指向结点a_{i-1}的指针。



在带头结点的单链表L中第i个数据元素之前插入数据元素x，先要找到结点a_{i-1}，并将指针p指向该结点。



```
Status ListInsert (LinkList &L, int i, ElemType e)
{  // 在带头结点的单链表L中第i个位置插入值为e的新结点
    p = L;  j = 0;
    while (p && j < i-1)
          { p = p->next;  ++j; }    // 寻找第i-1个结点，p指向该结点
    if (!p || j > i-1)    return ERROR;    // i>n+1或i<1
    s = (LinkList) malloc ( sizeof (LNode)); // 生成新结点
    s->data = e;
    s->next = p->next;      p->next = s; // 插入L中
    return OK;   }
```



如果链表的长度为n，则插入算法合法的插入位置有n+1个，即1≦i≦n+1，当i=n+1时，新结点插入在链表尾部。

插入算法的平均时间复杂度为O(n)。

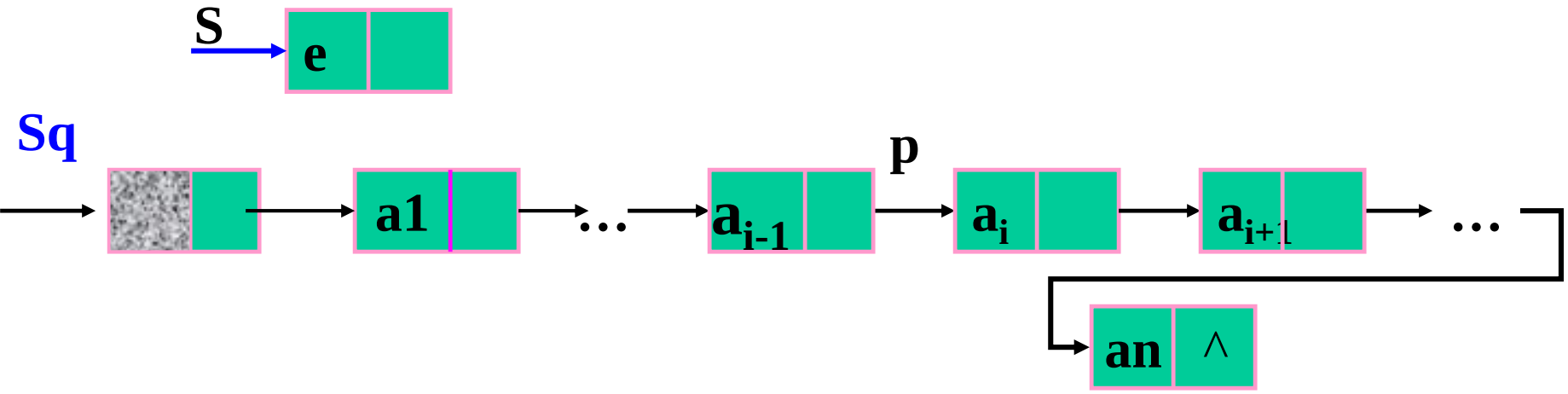
已知Sq是带头结点的非空单链表，且p结点既不是第一个结点，也不是最后一个结点，则：

(1) 在p结点后插入S结点的语句序列是：

(2) 在p结点前插入S结点的语句序列是：

(3) 在表尾插入S结点的语句序列是：

(4) 在表头插入S结点的语句序列是：



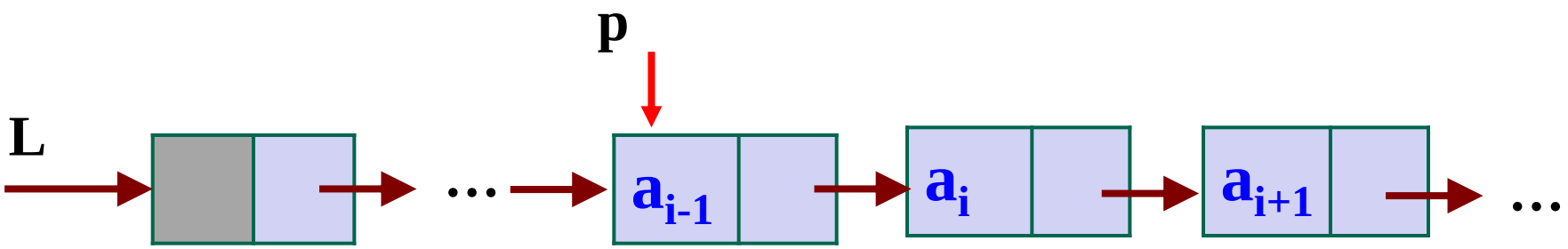
6. 删除数据元素ListDelete (&L，i，&e)

在单链表中删除指定位置的元素a_i，首先要找到其前驱结点a_{i-1}。

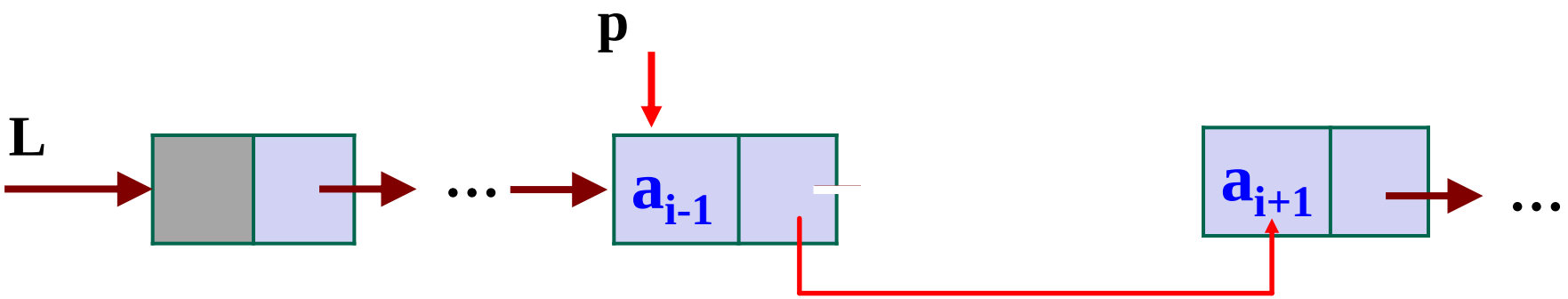
p->next=p->next->next;

单链表的删除算法如下：

```
Status ListDelete (LinkList &L，int i，ElemType &e)
{//在带头结点的单链表L中，删除第i个元素，并由e返回其值
  p = L; j = 0;
  while (p->next && j < i-1) // 查找第i-1个结点，p指向该结点
  {  p = p->next; ++j; }
  if (!(p->next) || j > i-1)    return ERROR;
                          //当 i>n或i<1时，删除位置不合理
```



```
    q = p->next;          //临时保存被删结点的地址以备释放
    p->next = q->next; //改变删除结点前驱结点的指针域
    e = q->data;
    free(q);    // 释放结点

    return OK;
}
```



删除算法的循环条件：

```
while (p->next && j < i-1)
```

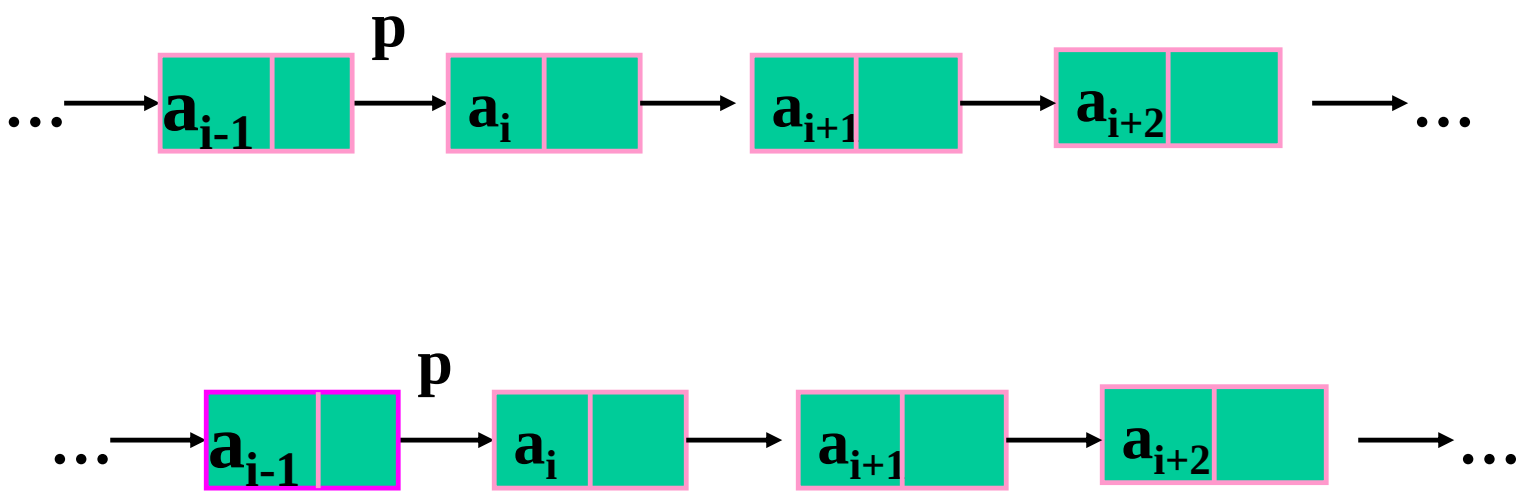
插入算法的循环条件：

```
while (p && j < i-1)
```

设单链表的长度为n，则删去第i个结点仅当$1 \leqq i \leqq n$时是合法的。显然此算法的时间复杂度也是$O(n)$。

从上面的讨论可以看出，链表上实现插入和删除运算，无须移动结点，仅需修改指针。

删除结点a_i



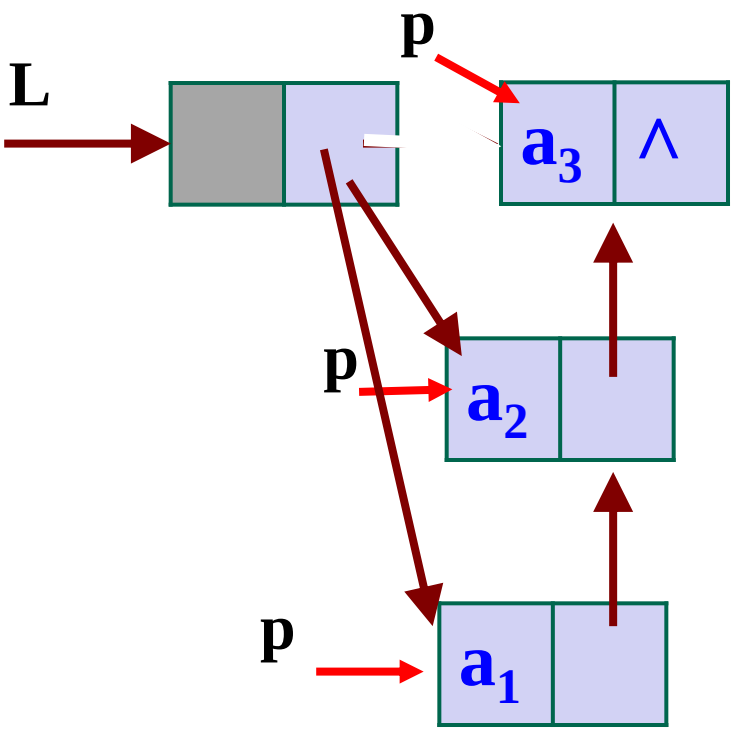
7. 创建单链表CreateList (&L, n)

单链表是一种动态结构，每个链表占用的空间不需要预先分配，而是由系统按需即时生成。因此，建立单链表过程就是一个动态生成链表的过程。即从“空表”的初始状态起，依此建立各元素结点，并逐个插入链表。

动态地建立单链表的常用方法有两种：

（1）头插法建立单链表

- 创建一个只有头结点的空链表；
- 依次读取相应的数据元素值，生成新结点；
- 将新结点插入到当前链表的头结点之后，直到结束。

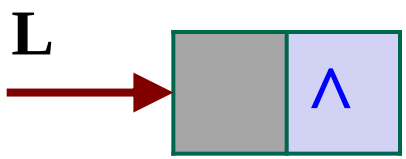


…

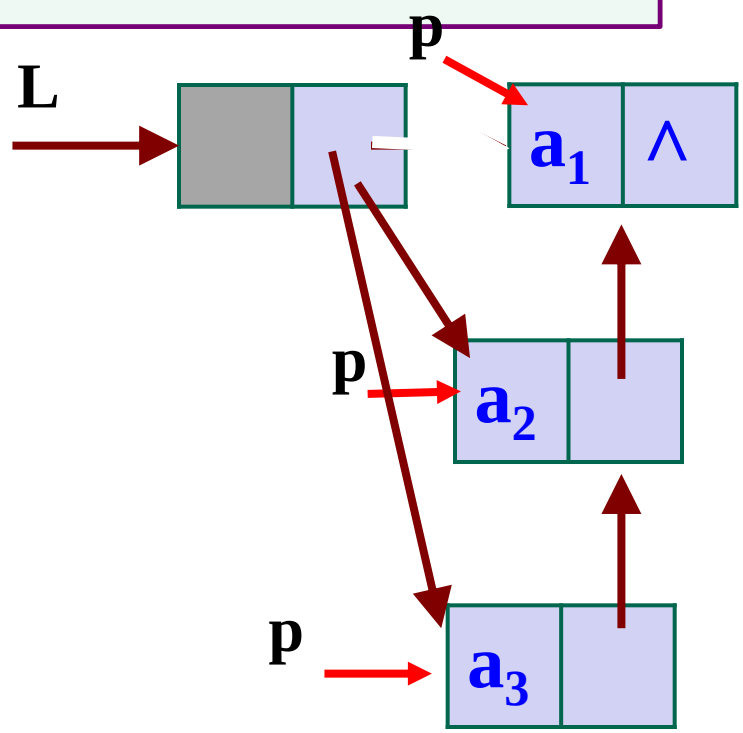
特点：元素输入顺序与逻辑顺序是相反的。

头插法建表算法如下：

```
void CreateList (LinkList &L,int n)
{  // 逆位序输入n个元素的值，建立带表头结点的单链表L。
   L = (LinkList) malloc (sizeof (LNode));
   L->next = NULL;      // 先建立一个带头结点的单链表
```

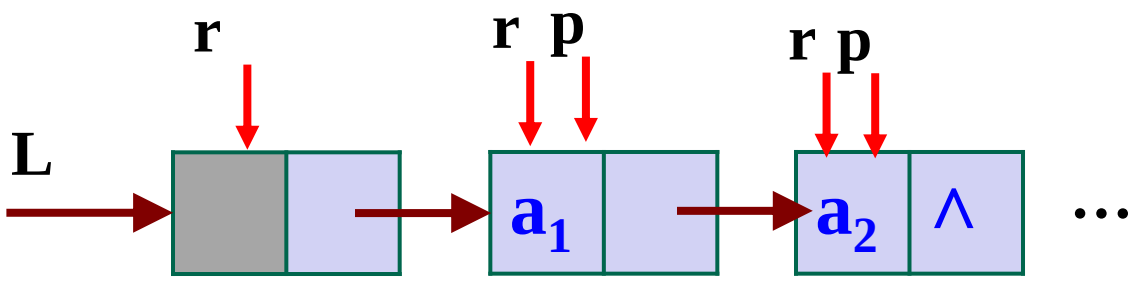


```
for (i = 0; i<n; i++)
{
    p = (LinkList) malloc (sizeof (LNode));  // 生成新结点
    scanf(&p->data);   // 输入元素值
    p->next = L->next;
    L->next = p; //将新结点 插入到头结点之后
}
```



(2) 尾插法建立单链表

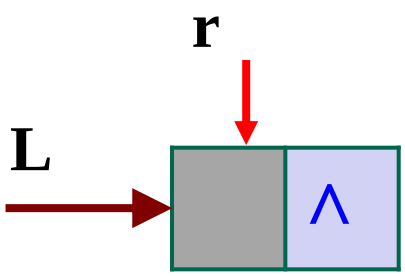
- 创建一个只有头结点的空链表；
- 依次读取相应的数据元素值，生成新结点；
- 将新结点插入到当前链表的尾结点之后，直到结束。



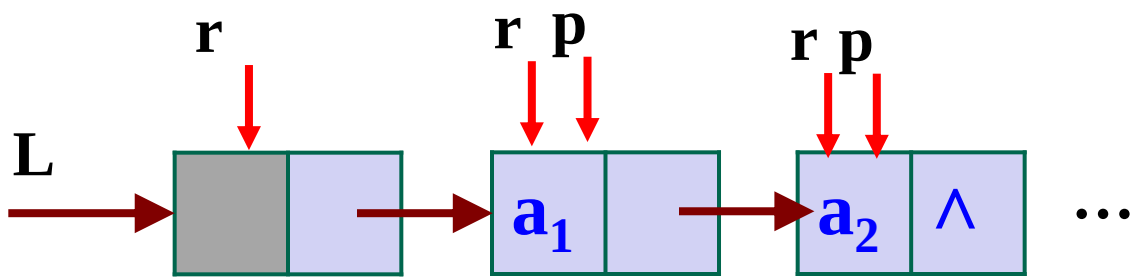
特点：元素输入顺序与逻辑顺序是相同的。

尾插法建表算法如下：

```
void CreateList (LinkList &L,int n)
{  // 输入n个元素的值，建立带表头结点的单链表L。
   L = (LinkList) malloc (sizeof (LNode));
   L->next = NULL;      // 先建立一个带头结点的单链表
   r=L;        //尾指针r指向头结点
```



```
    for (i = 0; i<n; i++)
     {
          p = (LinkList) malloc (sizeof (LNode));  // 生成新结点
          scanf(&p->data);   // 输入元素值
          p->next =NULL;
          r->next = p;   //将新结点 插入到尾结点*r之后
          r=p;          //r指向新的尾结点
      }
}
```

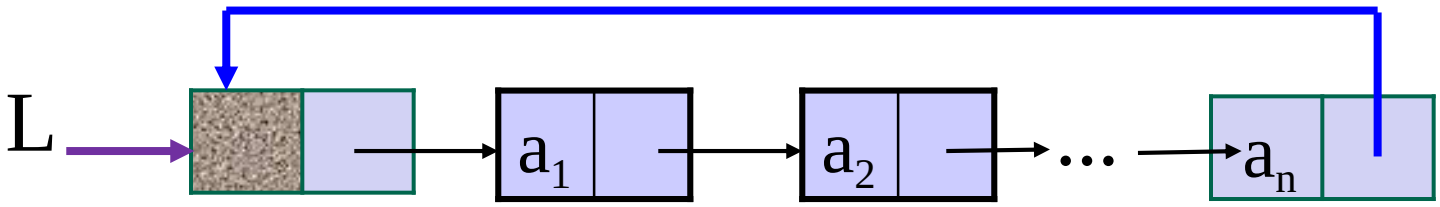


2.3.3　循环链表和双向链表

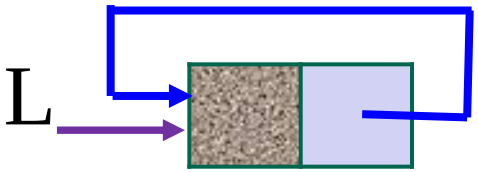
1. 循环链表

表中最后一个结点的指针域指向头结点，整个链表形成一个环。

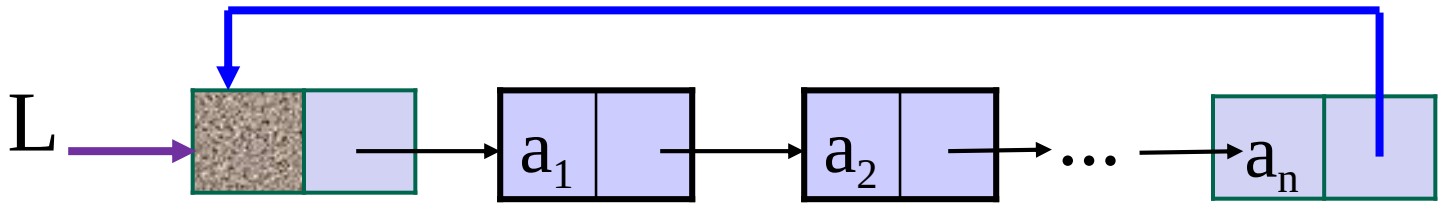
如下图所示：



a．非空循环单链表



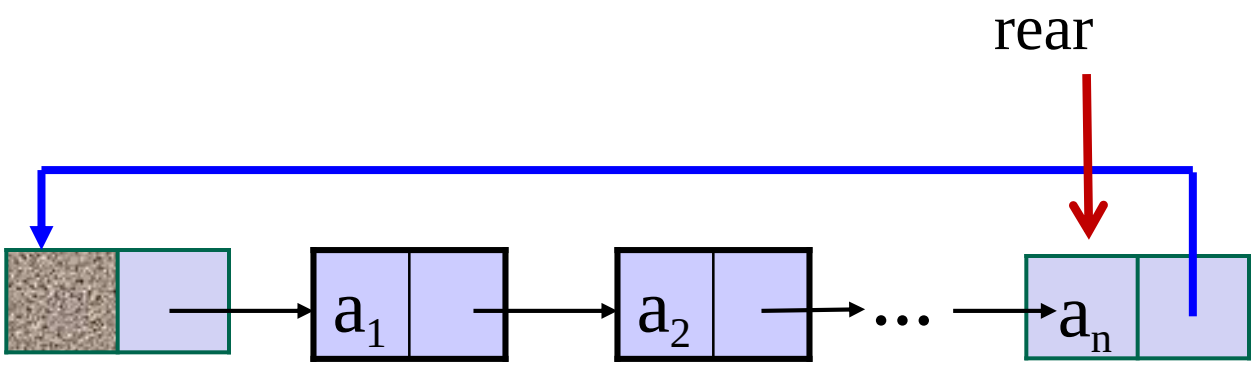
b．空循环单链表



❖循环单链表的操作与单链表基本一致，差别仅在于：遍历链表时，判断当前指针是否指向表尾结点的终止条件不同。

- 单链表　p！=NULL或p->next！=NULL
- 循环链表　p！=L或p->next！=L

在很多实际问题中，表的操作常常是在表的首尾位置上进行，此时头指针表示的循环单链表就显得不够方便.



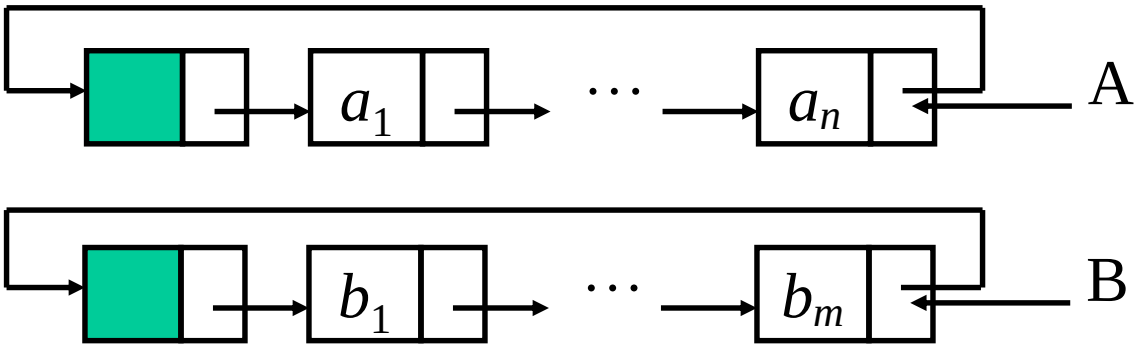
如果改用尾指针rear来表示循环单链表，

首元结点a1存储位置为：　rear->next->next

终端结点an的存储位置为：rear

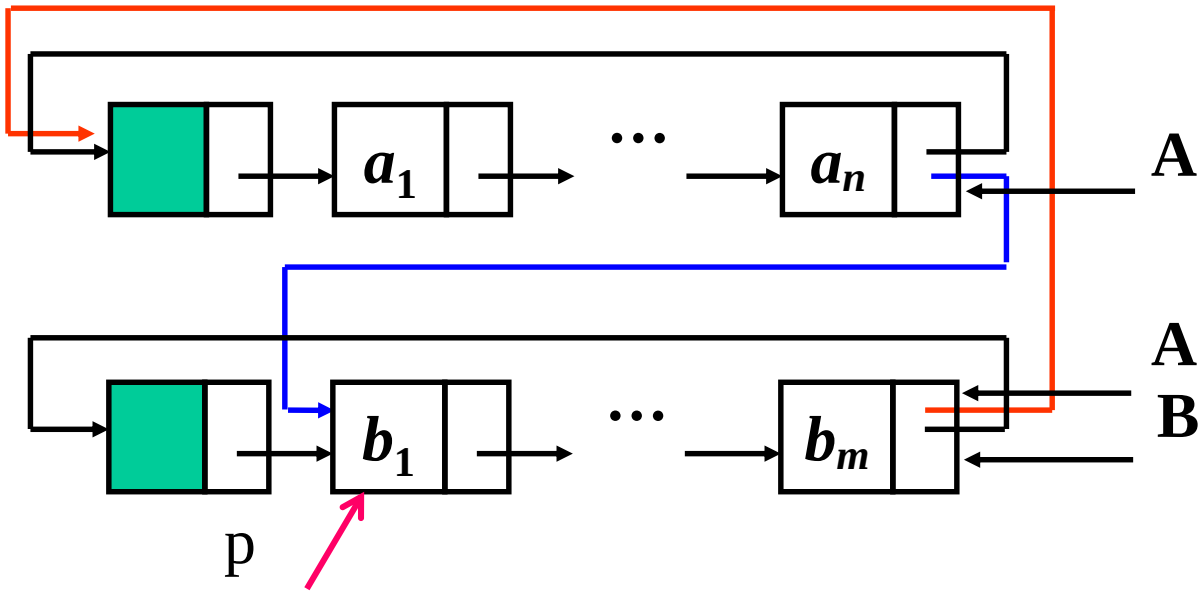
因此，实际中多采用尾指针表示循环单链表。

若在循环链表中设立尾指针而不设头指针，可使一些操作简化，例如：将两个线性表合并成一个线性表。



将两个设立尾指针的线性表合并成一个线性表时，仅需将第一个表的尾指针指向第二个表的第一个结点，第二个表的尾指针指向第一个表的头结点，然后释放第二个表的头结点。

操作过程如图所示：



主要语句段如下：

```
p = B –>next->next;
B –>next = A->next;
A –>next = p;
```

当线性表由以上图的循环链表作存储结构时，此操作仅需改变两个指针即可，因此时间复杂度是 $O(1)$。

2. 双向链表(Double linked list)

单链表:

查找某结点的直接后继结点的执行时间为$O(1)$。

查找某结点的直接前驱结点的执行时间为$O(n)$。

可用双向链表来克服单链表的这种单向性的缺点。

在双向链表的结点中有两个指针域，一个指向直接后继，另一个指向直接前趋。

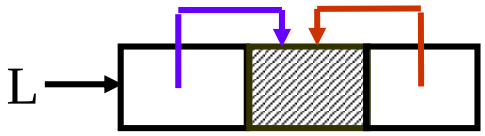

```
//———————双向链表存储结构———————————
typedef struct DuLNode {
    ElemType data;                  // 数据域
    struct DuLNode *prior;          // 指向直接前驱的指针域
    struct DuLNode *next;           // 指向直接后继的指针域
} DuLNode, *DuLinkList;
```

对指向双向链表任一结点的指针d，有下面的关系：

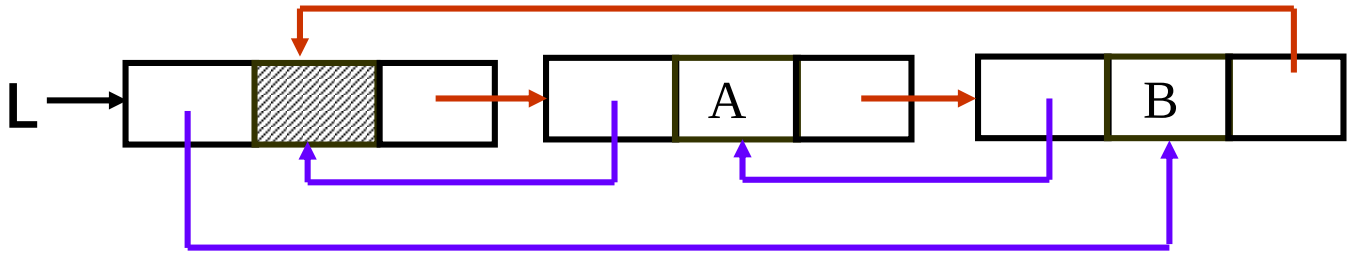
d->next->prior=d->prior->next=d

即：当前结点后继的前趋是自身，当前结点前趋的后继也是自身。

和单链的循环表类似，双向链表也有循环表，如下图：



a. 空的双向循环链表：

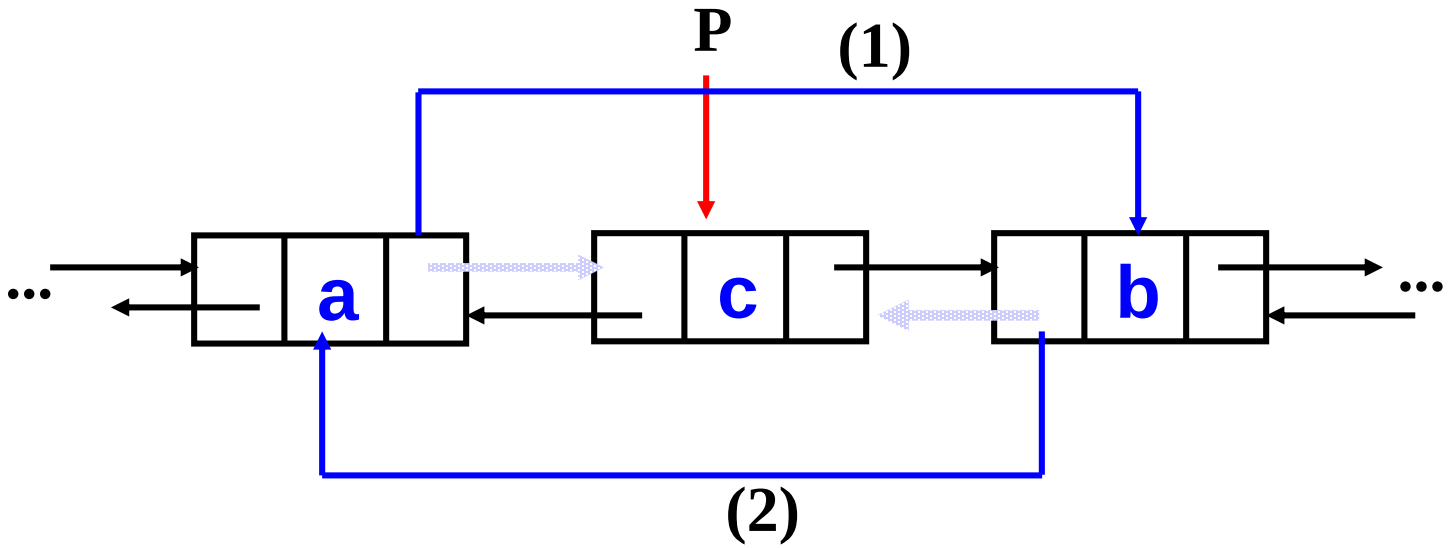


b. 非空双向循环链表：

(1). 双向链表的删除操作:

删除 p 结点

(1) P->prior->next=p->next

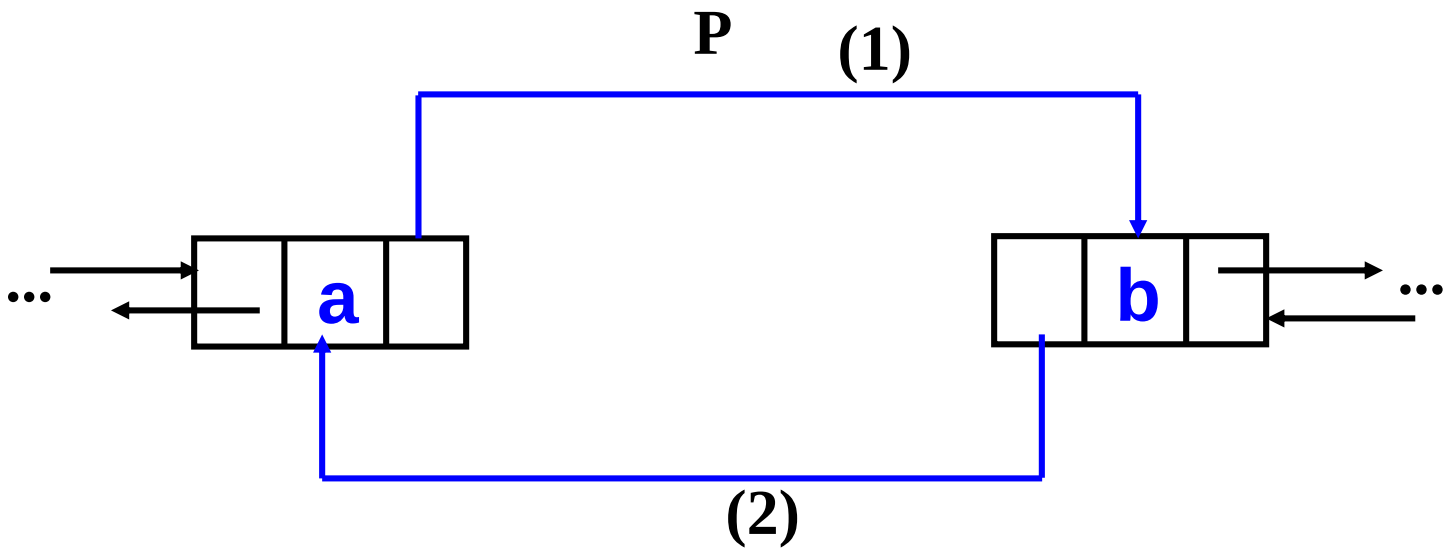


(2) p->next->prior = p->prior

(1). 双向链表的删除操作:

删除 p 结点

(1) P->prior->next=p->next

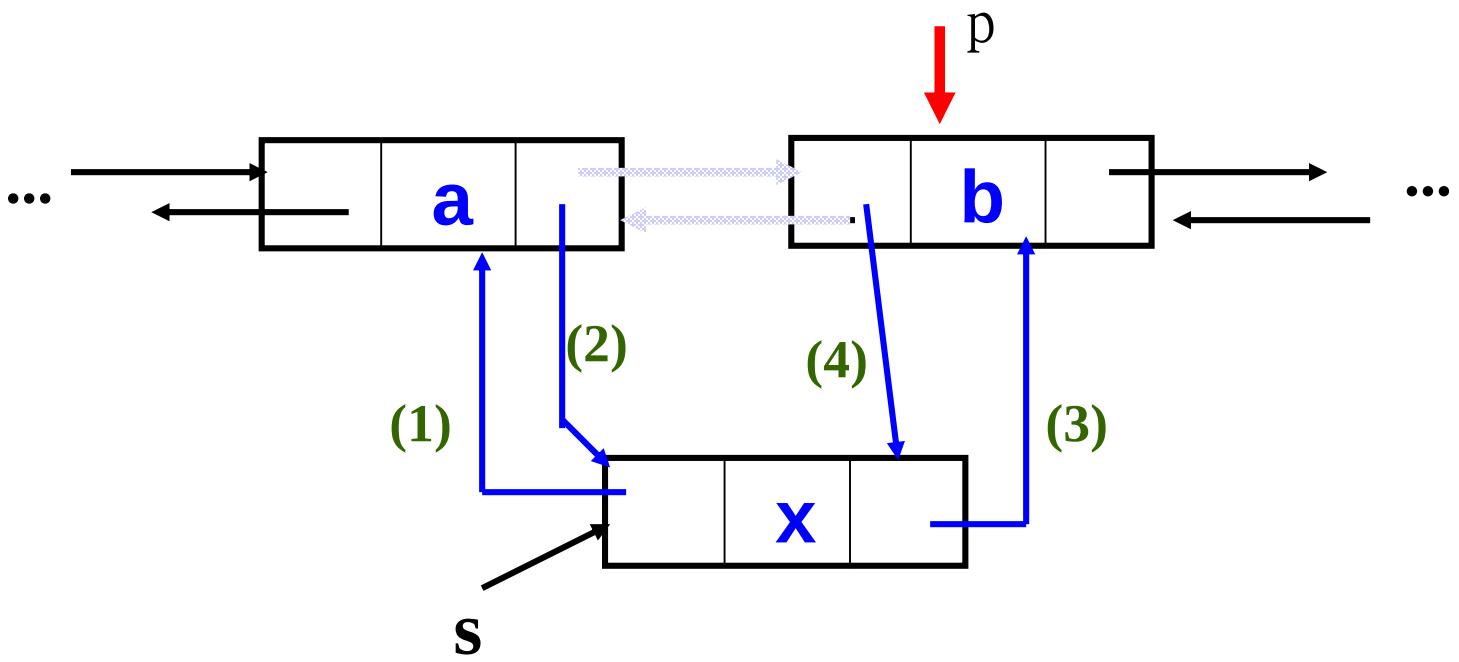


(2) p->next->prior = p->prior

(3) free(p)

(2). 双向链表的插入操作

在 p 结点之前插入 s 结点



(1) s->prior=p->prior

(2) p->prior->next=s

(3) s->next=p

(4) p->prior=s

2.4 线性表的应用

1. 一元多项式的表示及相加

一元多项式的表示：

$$P_n(x)=P_0+P_1x+P_2x^2+\cdots\cdots+P_nx^n$$

可用线性表P表示

$$P=(P_0,P_1,P_2,\cdots\cdots,P_n)$$

$$S(x)=1+3x^{1000}+2x^{20000}$$

但对S(x)这样的多项式浪费空间

一般情况 $$P_n(x)=P_1x^{e1}+P_2x^{e2}+\cdots\cdots+P_mx^{em}$$

其中 $0\le e1<e2\cdots\cdots<em=n$（$P_i$为非零系数）

用每个元素含两个数据项的线性表表示

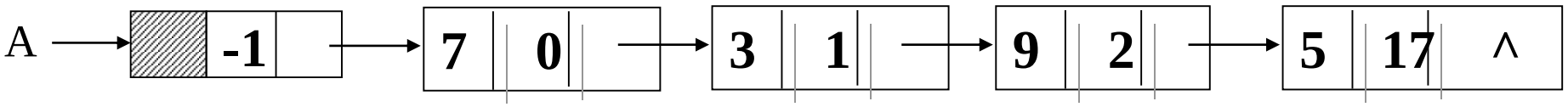
$$((P_1,\ e1),(P_2,\ e2),\cdots\cdots(P_m,\ em))$$

其存储结构可以用顺序存储结构，也可以用单链表

类型定义：

```
typedef struct  node{
        float     coef;    //系数
        int       expn;    //指数
      struct   node    * next;
} polynomial;
```

$$A(x)=7+3x+9x^2+5x^{17}$$

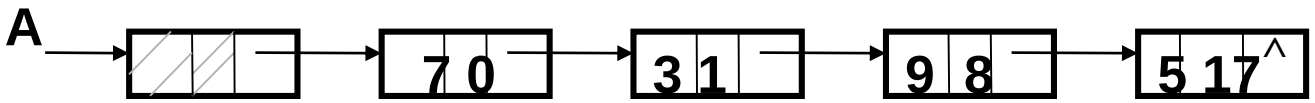


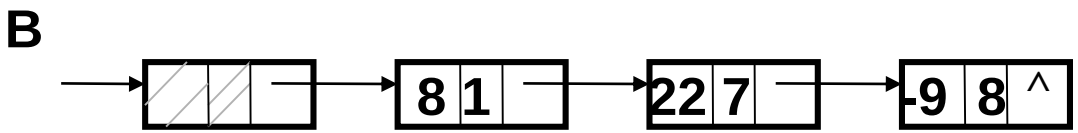
一元多项式相加

·如：多项式：$A(x) = 7+3x+9x^8+5x^{17}$

$B(x) = 8x+22x^7-9x^8$ ；

结果保留在 A 的单链表之中。



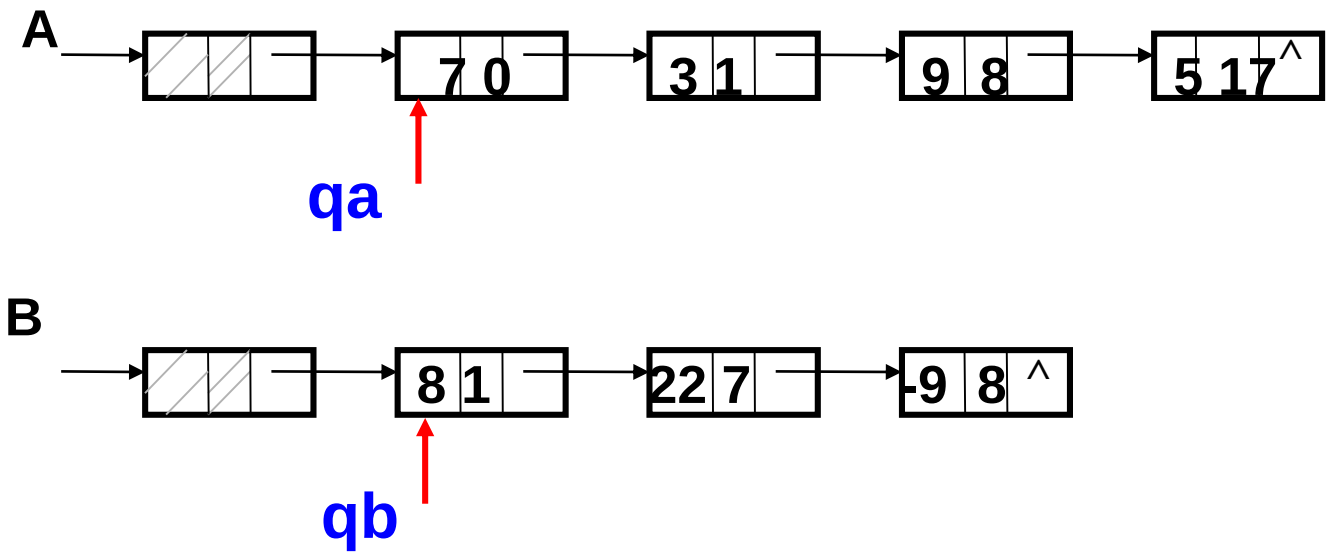


一元多项式相加

·如：多项式：$A(x) = 7+3x+9x^8+5x^{17}$

$B(x) = 8x+22x^7-9x^8$ ；

结果保留在 A 的单链表之中。

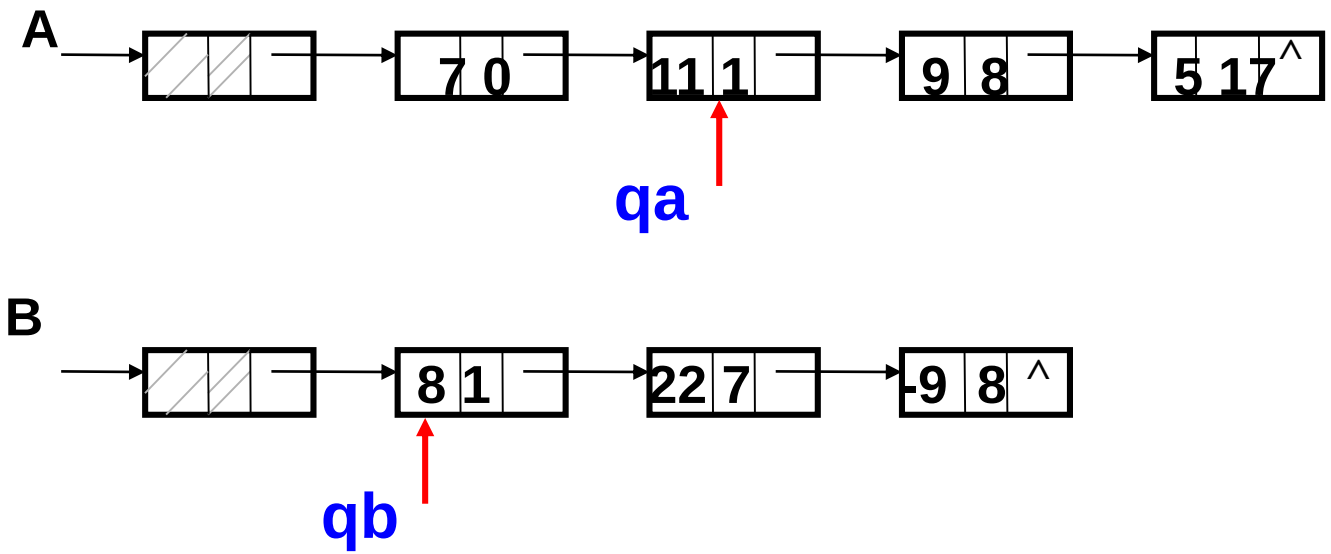


一元多项式相加

·如：多项式：$A(x) = 7+3x+9x^8+5x^{17}$

$B(x) = 8x+22x^7-9x^8$ ；

结果保留在 A 的单链表之中。

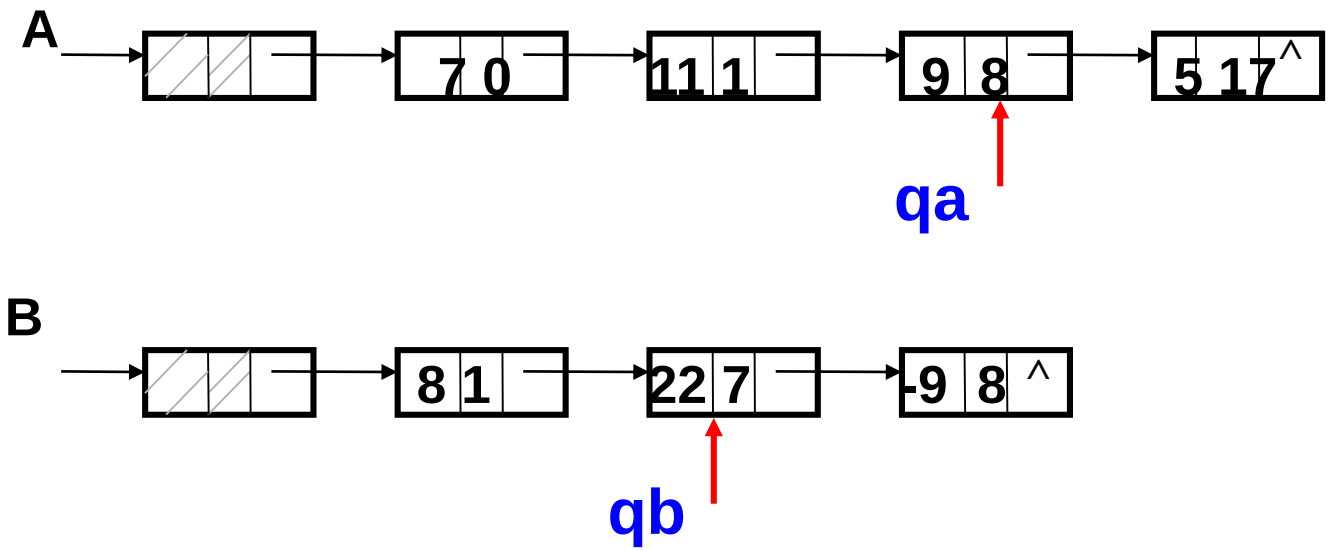


一元多项式相加

·如：多项式：$A(x) = 7+3x+9x^8+5x^{17}$

$B(x) = 8x+22x^7-9x^8$ ；

结果保留在 A 的单链表之中。

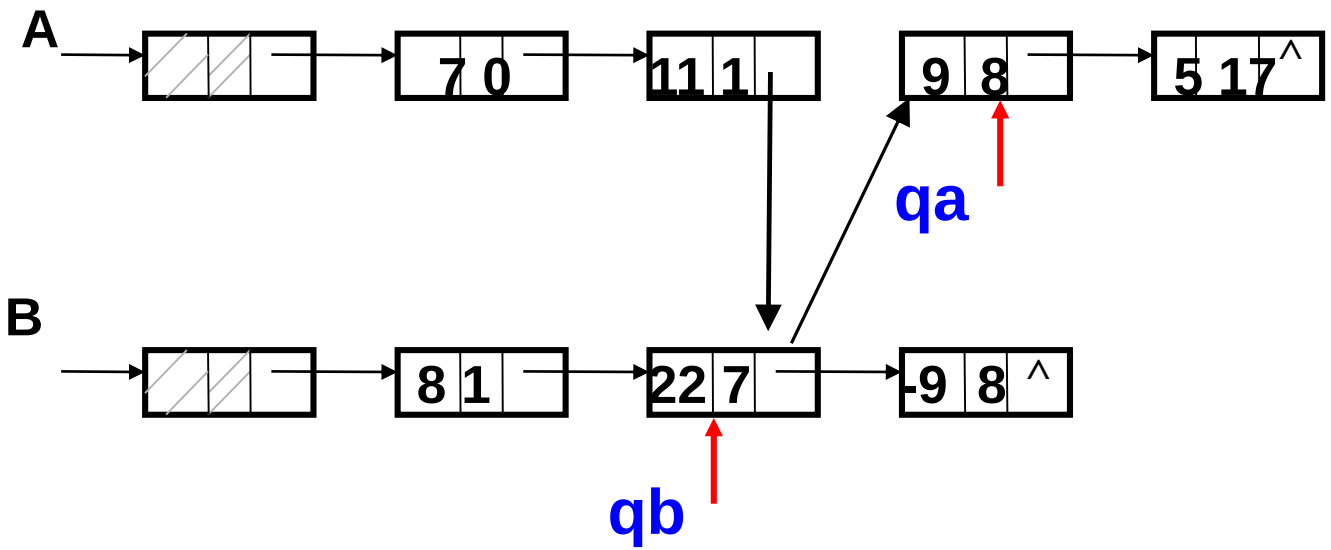


一元多项式相加

·如：多项式：$A(x) = 7+3x+9x^8+5x^{17}$

$B(x) = 8x+22x^7-9x^8$ ；

结果保留在 A 的单链表之中。

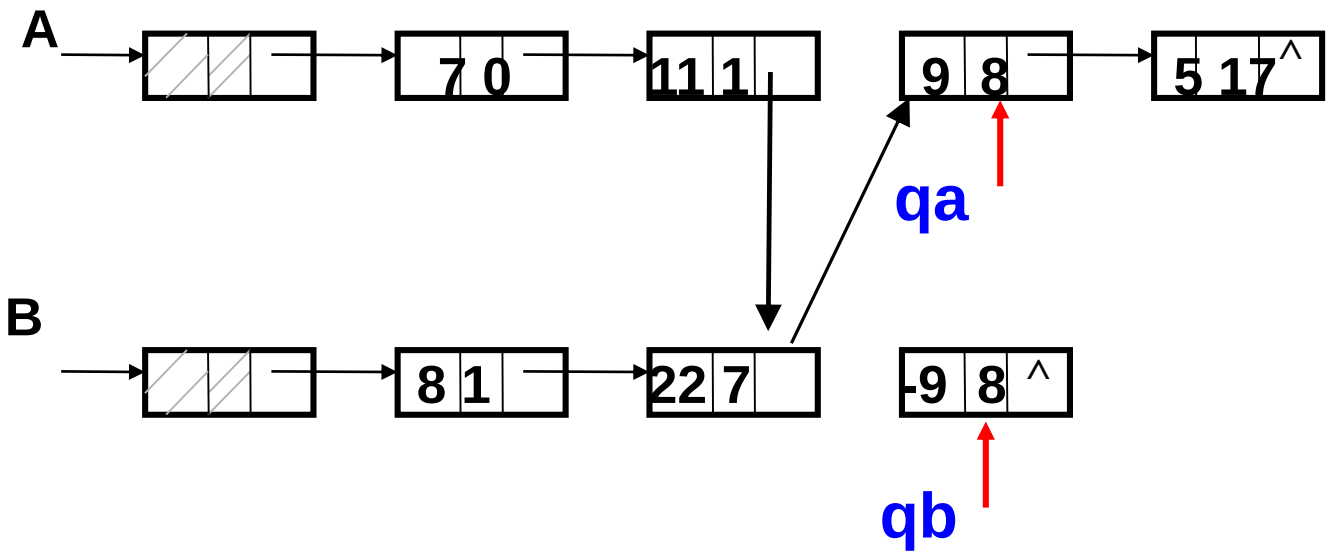


一元多项式相加

·如：多项式：$A(x) = 7+3x+9x^8+5x^{17}$

$B(x) = 8x+22x^7-9x^8$ ；

结果保留在 A 的单链表之中。

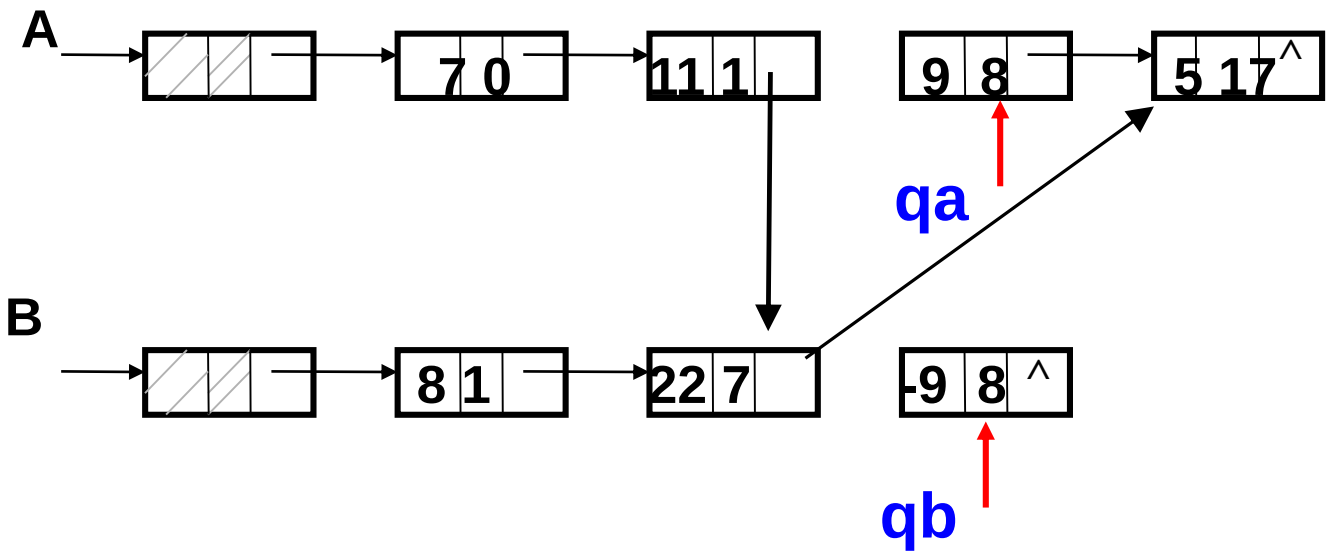


一元多项式相加

·如：多项式：$A(x) = 7+3x+9x^8+5x^{17}$

$B(x) = 8x+22x^7-9x^8$ ；

结果保留在 A 的单链表之中。



运算规则

设qa,qb分别指向A,B中某一结点，qa,qb初值是第一结点

比较 qa->exp与qb->exp

- qa->exp < qb->exp: qa结点是和多项式中的一项
  qa后移,qb不动
- qa->exp > qb->exp: qb结点是和多项式中的一项
  将qb插在qa之前,qb后移,qa不动
- qa->exp = qb->exp: 系数相加
  - 0：从A表中删去qa,
    释放qa,qb，qa,qb后移
  - ≠0：修改qa系数域,
    释放qb，qa,qb后移

直到qa或qb为NULL

- 若qb==NULL，结束
- 若qa==NULL，将B中剩余部分连到A上即可

2. 线性表的合并

已知两个线性表LA和LB中的数据元素按值非递减有序排列，要求将 LA和LB归并为一个新的线性表LC，且LC中的数据元素仍按值非递减有序排列。

例如，设

LA=(3, 5, 8, 11)

LB=(2, 6, 8, 9, 11, 15, 20)

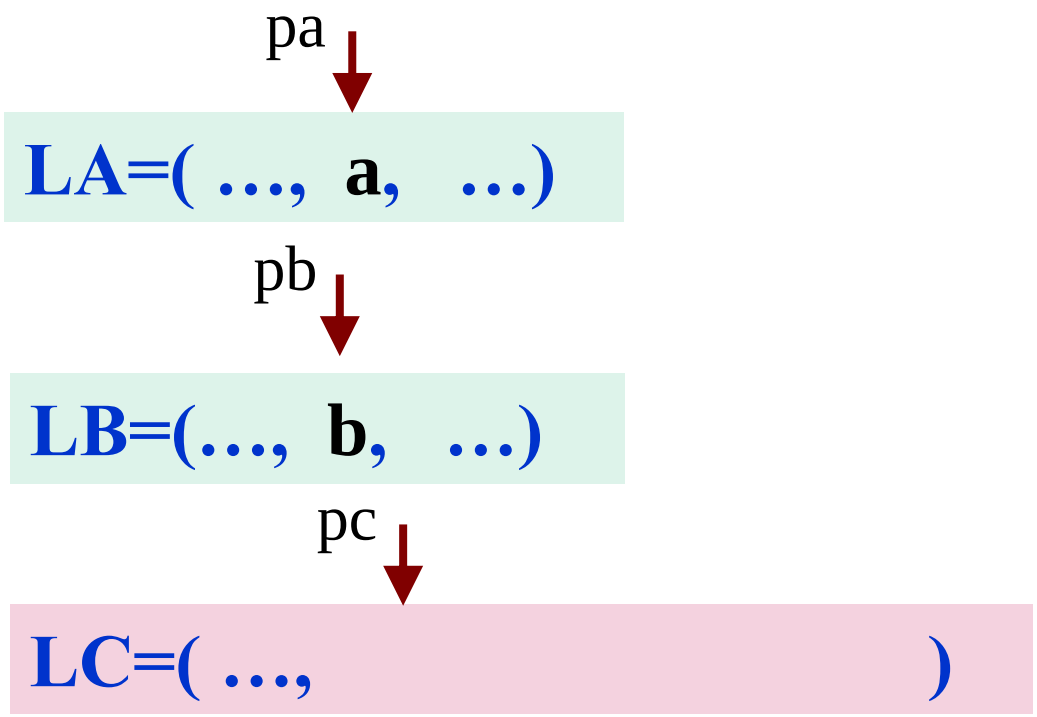
则

LC=(2, 3, 5, 6, 8, 8, 9, 11, 11, 15, 20)

由问题要求可知，LC中的数据元素或是LA中的元素，或是Lb中的元素，因此只要先设LC为空表，然后将LA或LB中的元素逐个插入到LC中即可。

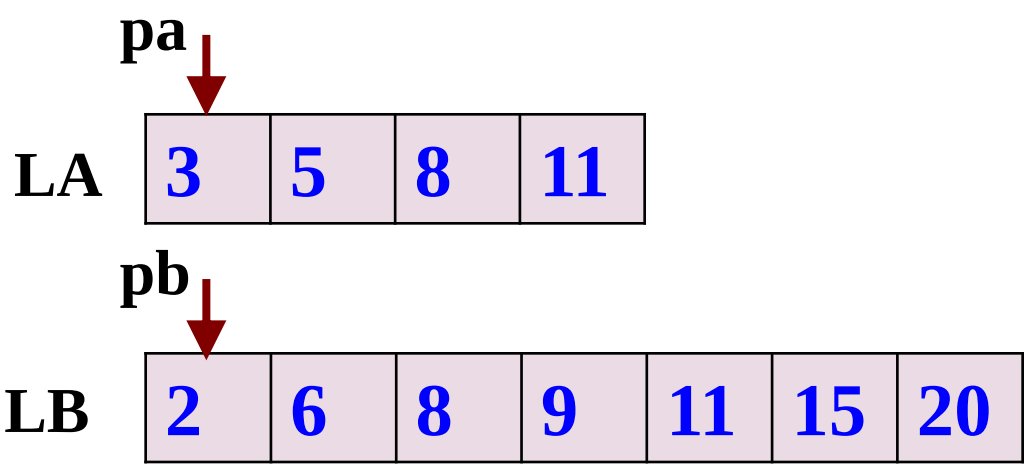
为使LC中的元素按值非递减有序排列，可设两个指针pa和pb分别指向LA和LB中的某个元素，若设pa当前所指的元素为a，pb当前所指的元素为b，则当前应插入到LC中的元素c为

$$c = \begin{cases} a & \text{当} a \le b \text{时} \\ b & \text{当} a > b \text{时} \end{cases}$$

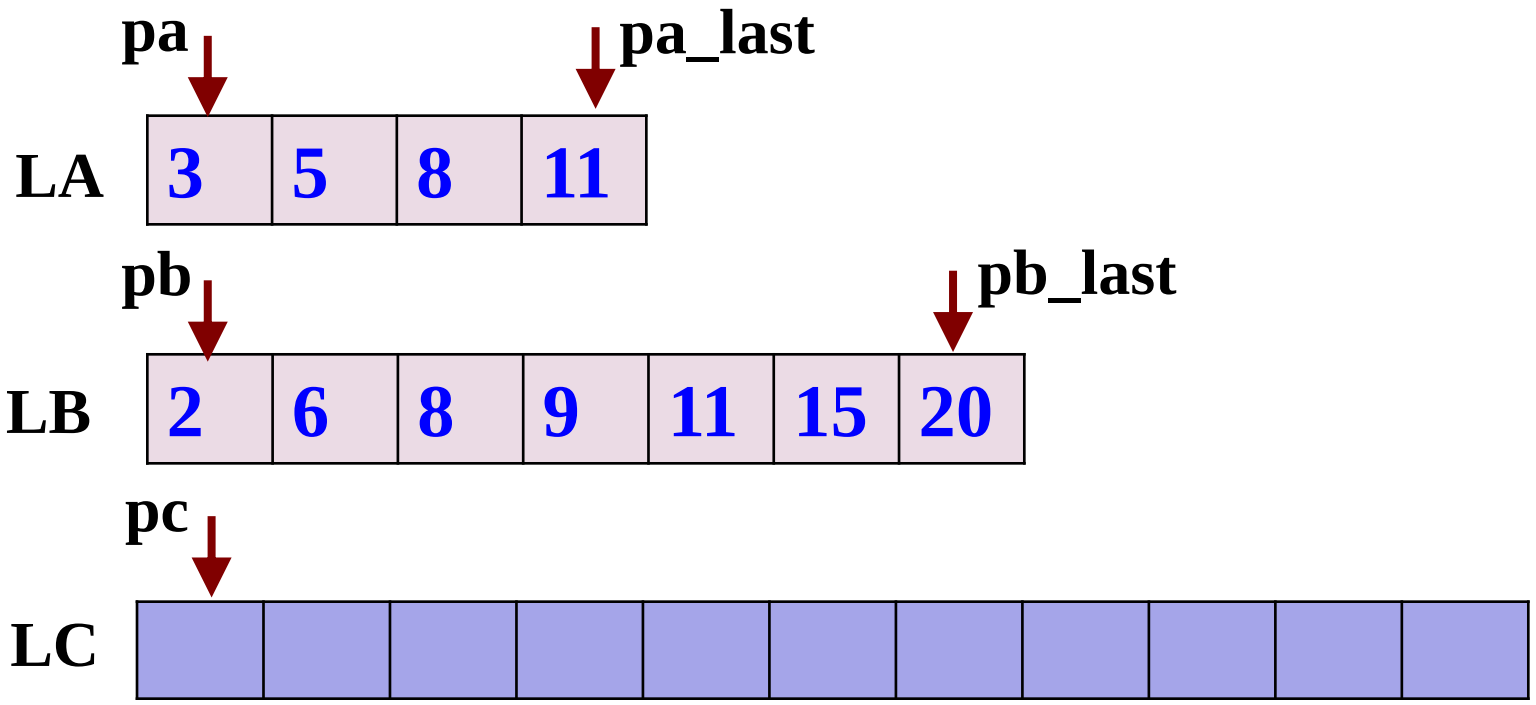


(1). 两个顺序有序表的合并

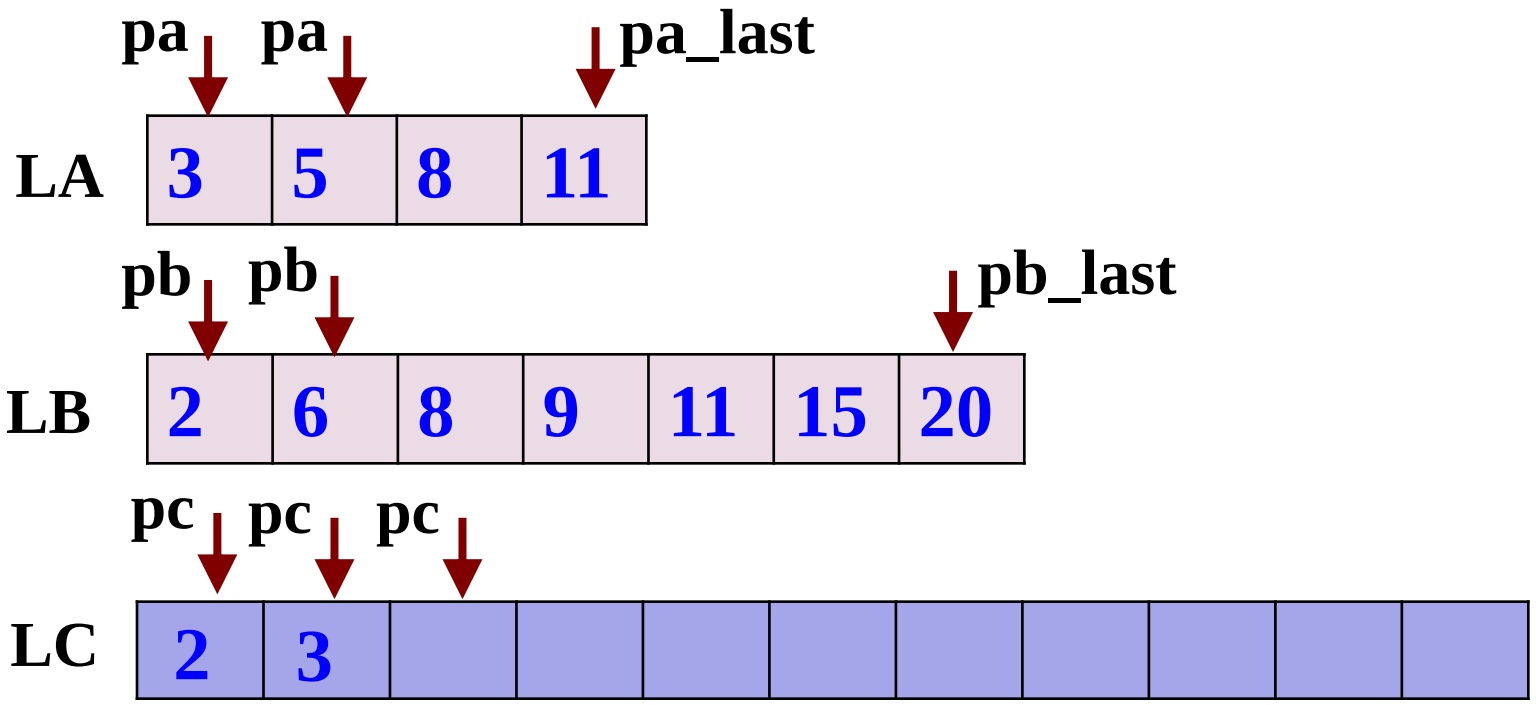
```
void MergeList_Sq(SqList    la, SqList lb, SqList &lc)
 { // 已知顺序表La和Lb的元素按值非递减排列，归并La和Lb得
  //到新的顺序表Lc，Lc的元素也按值非递减排列。
      pa = la.elem;
      pb = lb.elem;      //pa和pb的初值分别指向表的第一个元素
```



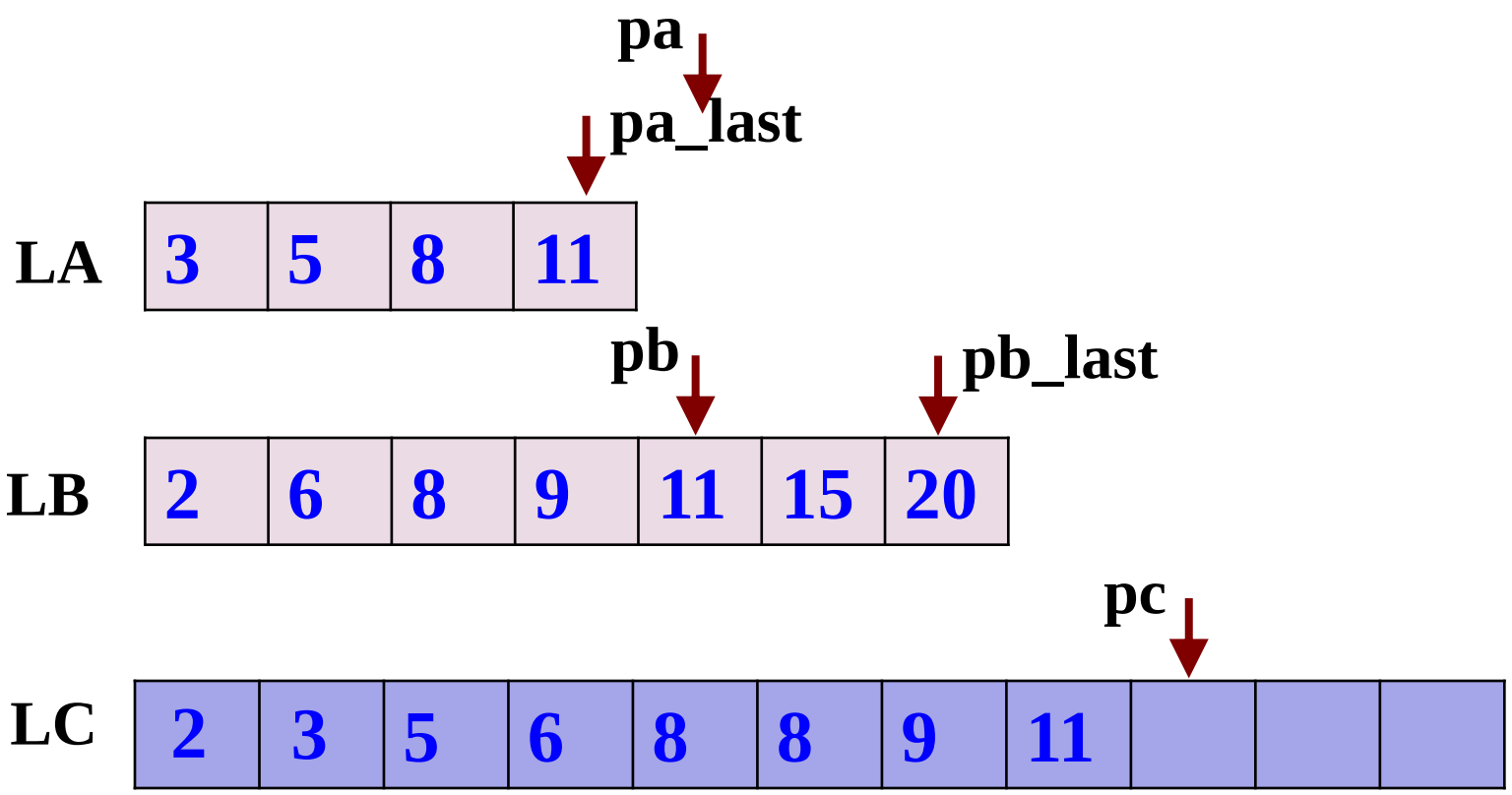
```
lc.listsize =lc.length= la.length + lb.length;
                                    //LC表的长度为LA和LB两表的长度之和
pc = lc.elem =  (ElemType*)malloc(sizeof(ElemType)*lc.listsize);
                                    //为LC表分配空间
pa_last = la.elem + la.length - 1;
                                    //指针pa_last指向LA的最后一个元素
pb_last = lb.elem + lb.length - 1;
                                    //指针pb_last指向LB的最后一个元素
```



```
while(pa <= pa_last && pb <= pb_last)   //La和LB均未到达表尾
{    if  (*pa <= *pb) //依次取两表中值较小的元素插入到LC的最后
          *pc++ = *pa++;
     else
          *pc++ = *pb++;
}
```



```
    while(pa <= pa_last)    *pc++ = *pa++;
        //LB已到表尾，依次将LA的剩余元素插入到LC的最后
    while(pb <= pb_last)    *pc++ = *pb++;
        //LA已到表尾，依次将LB的剩余元素插入到LC的最后
}
```



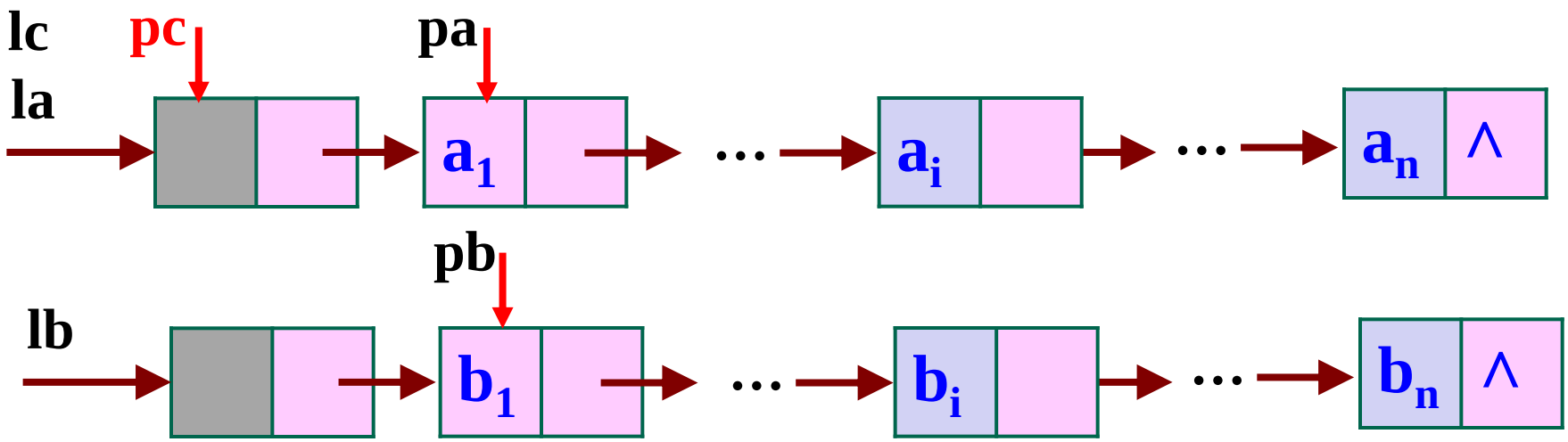
如果两个表长分别记为m和n，则算法的时间复杂度为O(m+n)。

(2). 两个链式有序表的合并

因为链表结点之间的关系是通过指针指向建立起来的，所以用链表进行合并不需要另外开辟存储空间，可以直接利用原来两个表的存储空间，合并过程中只需要把LA和LB两个表中的结点重新进行链接即可。

算法描述：

```
void MergeList_L(LinkList &la, LinkList &lb, LinkList &lc)
 { // 已知单链表La和Lb的元素按值非递减排列，归并La和Lb得
  //到新的单链表Lc，Lc的元素也按值非递减排列。
      pa = la->next;   pb = lb->next;
                          //pa和pb的初值分别指向表的第一个元素结点
      lc=la;              //用LA的头结点作为LC的头结点
      pc =lc;             //pc的初值指向LC的头结点
```



```
while(pa&pb)
{ //LA和LB均未到达表尾，依次“摘取”两表中值较小的结点插入到LC的最后
   if (pa->data<=pb->data)     //“摘取”pa所指结点
      {
           pc->next=pa;
           pc=pa;
           pa=pa->next;
      }
      else         //“摘取”pb所指结点
      {
           pc->next=pb;
           pc=pb;
           pb=pb->next;
      }
 }
  pc->next=pa?pa:pb;  //将非空表的剩余段插入到pc所指结点之后
  free(lb);                    //释放LB的头结点
}
```

第二章　　习题二

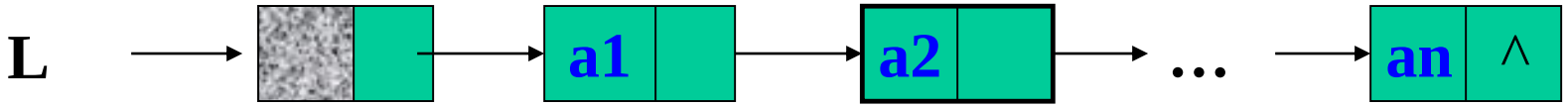
习题取自

数据结构题集　C语言版　严蔚敏等编　清华大学出版社

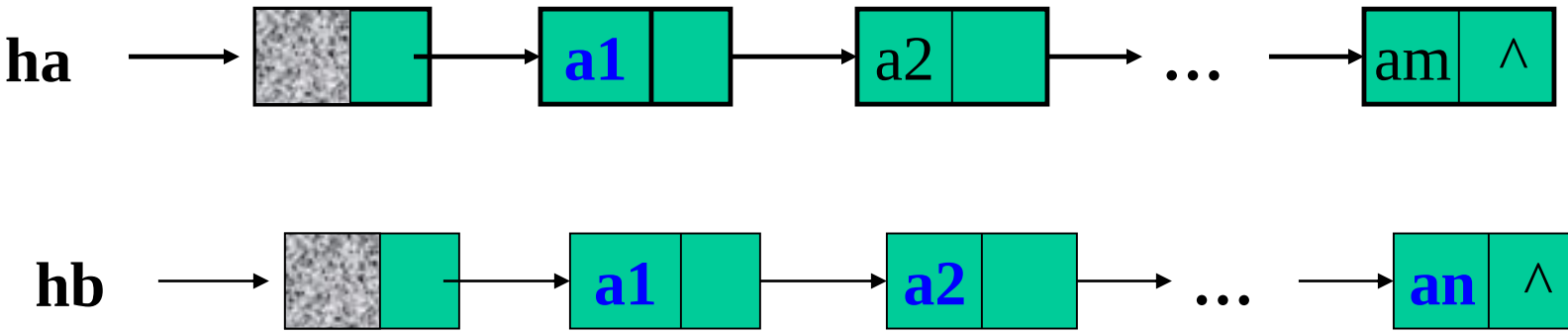
1 P17　2.14　2.15　2.19

2 P18　2.22　2.31

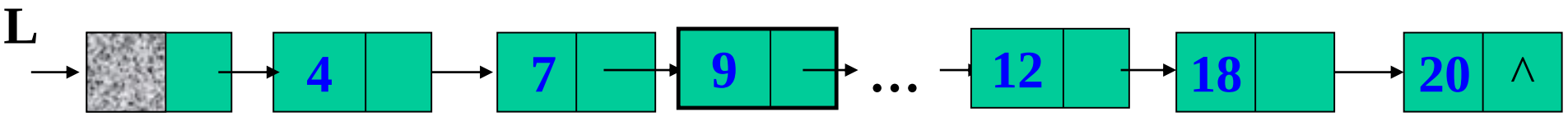
2.14 试写一算法在带头结点的单链表结构上实现线性表操作LENGTH(L).



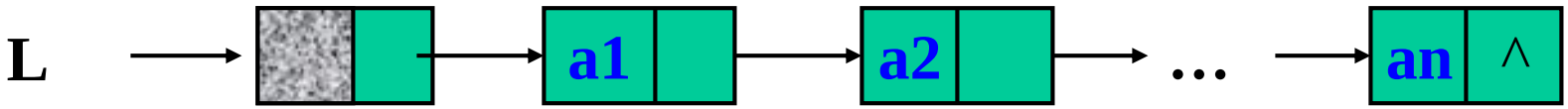
2.15已知指针ha和hb分别指向两个单链表的头结点，并且已知两个链表的长度分别为m和n，试写一算法将这两个链表连接在一起(即令其中一个表的首元结点连在另一个表的最后一个结点之后)。假设指针hc指向连接后的链表的头结点，并要求算法以尽可能短的时间完成连接运算。请分析你的算法的时间复杂度。



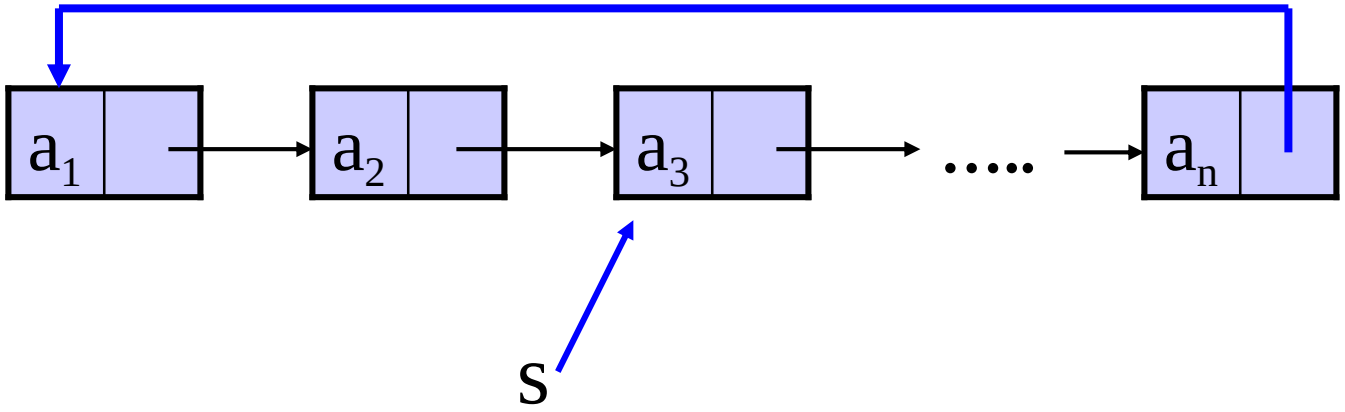
2.19已知线性表的元素以值递增有序排列，并以单链表作存储结构，试写一高效的算法，删除表中所有值大于mink且小于maxk的元素(若表中存在这样的元素)，同时释放被删结点空间，并分析你的算法的时间复杂度。



2.22 试写一算法，对单链表实现就地逆置。



2. 31假设某个单向循环链表的长度大于1，且表中既无头结点也无头指针。已知s指向链表中某个结点的指针，试编写算法在链表中删除指针s所指结点的前驱结点。



约瑟夫环　　见习题集P79

[问题描述]

约瑟夫(Joseph)问题的一种描述是：编号为1，2，…, n的n个人按顺时针方向围坐一圈，每人持有一个密码（正整数），一开始任选一个整数作为报数上限m, 从第一人开始按顺时针方向从自1开始顺序报数，报到m时停止报数。报m的人出列，将他的密码作为新的m值，从他的顺时针方向上的下一个人开始重新从1报数，如此下去，直至所有人全部出列为止，设计一个程序求出出列顺序。

[基本要求]

采用单向循环链表模拟此过程, 按照出列的顺序印出各人的编号

[测试数据]

n=7, 7个人的密码依次为：3，1，7，2，4，8，4，首先m的值为6（正确的出列顺序应为6, 1, 4, 7, 2, 3, 5）

```
typedef struct LNode
{
    int code;   //定义整型变量code用来存放序号
    int key;   //定义整型变量key用来存放密码
    struct LNode *next;
}LNode,*LinkList;
```