第2章 线性表

- 2．1 线性表的类型定义
- 2．2 线性表的顺序表示与实现
- 2．3 线性表的链式表示与实现
  - 2．3．1 线性链表
  - 2．3．2 循环链表
  - 2．3．3 双向链表

一、教学目的与要求

了解线性表的定义；学会线性表的顺序表示和实现、链式表示和实现

二、主要教学内容

线性表的类型定义；线性表的顺序表示和实现；线性表的链式表示和实现；一元多项式的表示及相加

三、教学重点、难点

线性表的顺序表示和实现、链式表示和实现

四、授课方法及手段

采用多媒体大屏幕投影授课

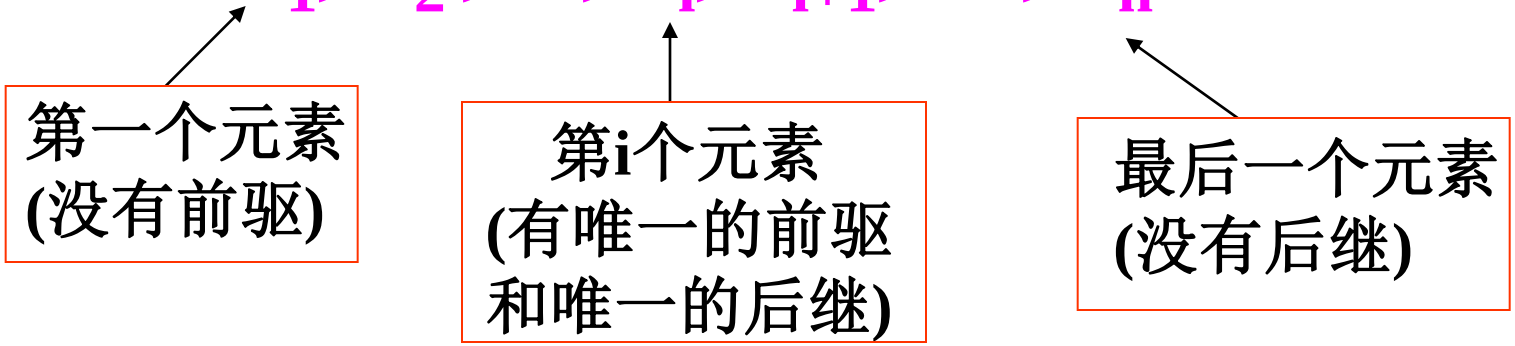
五、讲课具体内容（讲稿）

线性结构的特点是：

- 存在唯一的“第一个”数据元素；
- 存在唯一的“最后一个”数据元素；
- 除第一个外，每个数据元素均有且只有一个前驱元素；
- 除最后一个外，每个数据元素均有且只有一个后继元素。

2.1 线性表的类型定义

- 线性表举例:
  - 字母表　(A,B,C,…,X,Y,Z)
  - 数据序列　(6,17,28,50,92,188)
- n个元素的线性表:
  - $(a_1, a_2, \ldots, a_i, a_{i+1}, \ldots, a_n)$



线性表的抽象数据类型(ADT)

ADT List {

- 数据对象： **D = {a_i | a_i 属于 Elemset, (i=1, 2, …, n, n≥0)}**
- 数据关系： **R1 = {＜a_{i-1},a_i＞|a_{i-1},a_i 属于 D, (i=2,3,…,n)}**
- 基本操作：
  - **InitList(&L);　　DestroyList(&L);**
  - **ListInsert(&L,i,e);　ListDelete(&L,i,&e);　…**

} ADT List

基本操作(一)

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已经存在。

操作结果: 销毁线性表L。

ClearList(&L)

初始条件: 线性表L已经存在。

操作结果: 将线性表L重置为空表。

基本操作(二)

ListEmpty(L)

初始条件: 线性表L已经存在。

操作结果: 若线性表L为空表，则返回TURE;否则返回FALSE。

ListLength(L)

初始条件: 线性表L已经存在。

操作结果: 返回线性表L中的数据元素个数。

基本操作(三)

GetElem(L,i,&e);

初始条件: 线性表L已经存在，1<= I <= ListLength(L)。

操作结果: 用e返回线性表L中第i个数据元素的值。

LocateElem(L,e,compare())

初始条件：线性表L已经存在，compare()是数据元素判定函数。

操作结果：返回L中第1个与e满足compare()的数据元素的位序。若这样的数据元素不存在则返回值为0。

基本操作(四)

PriorElem(L, cur_e, &pre_e)

初始条件: 线性表L已经存在。

操作结果: 若cur_e是L的数据元素,且不是第一个,则用pre_e返回它的前驱,否则操作失败; pre_e无意义。

NextElem(L, cur_e, &next_e)

初始条件: 线性表L已经存在。

操作结果:若cur_e是L的数据元素, 且不是第最后个,则用next_e返回它的后继,否则操作失败, next_e无意义。

基本操作(五)

ListInsert(&L, i, e)

初始条件：线性表L已经存在，1<=i<= ListLength(L)+1。

操作结果：在L的第i个位置之前插入新的数据元素e，L的长度加一。

ListDelete(&L,i,&e)

初始条件：线性表L已经存在，1<=i<= ListLength(L)。

操作结果：删除L的第i个数据元素,并用e返回其值，L的长度减一。

例2-1 线性表La,Lb的合并

合并前：

La = (2,5,7,9)

Lb = (4,5,6,7)

合并后：

La = (2,5,7,9,4,6)

时间复杂度： O(ListLength(La)×ListLength(Lb))

```
void union(List &La, List Lb){
    La_len = ListLength(La);
    Lb_len = ListLength(Lb);
    for (i=1; i<=Lb_len; i++){
        GetElem(Lb,i, e);
        If(!LocateElem(La,e,equal))
            ListInsert(La, ++La_len, e);
    }
}//union
```

例：非递减线性表La, Lb的合并

合并前： La = (3,5,8,11)

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

合并后： Lc=(2,3,5,6,8,8,9,11,11,15,20)

```
void MergeList(List La, List Lb, List &Lc){
// 已知非递减线性表La,Lb
//将所有La和Lb中的数据元素归并到Lc中,
   使Lc 的数据元素也是非递减的。
}// MergeList
```

非递减线性表La, Lb的合并的C程序

```
InitList(&Lc);    i=j=1;    k=0;
La_len = ListLength(La);
Lb_len = ListLength(Lb);

While ((i<=La_len) && (j<=Lb_len)){
    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);
}
```

时间复杂度为:

$$O(ListLength(La) + ListLength(Lb))$$

2.2 线性表的顺序表示与实现

- 线性表（$a_1, a_2, \ldots, a_i, a_{i+1}, \ldots, a_n$）的顺序表示：

  用一组地址连续的存贮单元依次存贮线性表的数据元素.

- 设 l 为每个数据元素所需的存储单元（即：每个元素的长度）

$$Loc(a_{i+1}) = Loc(a_i) + l$$

$$Loc(a_i) = Loc(a_1) + (i-1)* l$$

线性表的顺序表示(图示)

b ——数据元素a1的存储地址

l ——每个数据元素所需的存储单元大小

C++补充知识1—指针

变量地址——系统分配给变量的内存单元的起始地址（&—取地址符）

```
#include <stdio.h>
int main(int argc, char* argv[])
{     int x=10;
      printf("%d\n",x);
      printf("%ld\n",&x);
      return 0;
}
```

C++补充知识1－指针

指针与指针变量

（1）指针——即地址

一个变量的地址称为该变量的指针。通过变量的指针能够找到该变量。

（2）指针变量——专门用于存储其它变量地址的变量

指针与指针变量的区别，就是变量值与变量的区别。

（3）为表示指针变量和它指向的变量之间的关系，用指针运算符“*”表示。

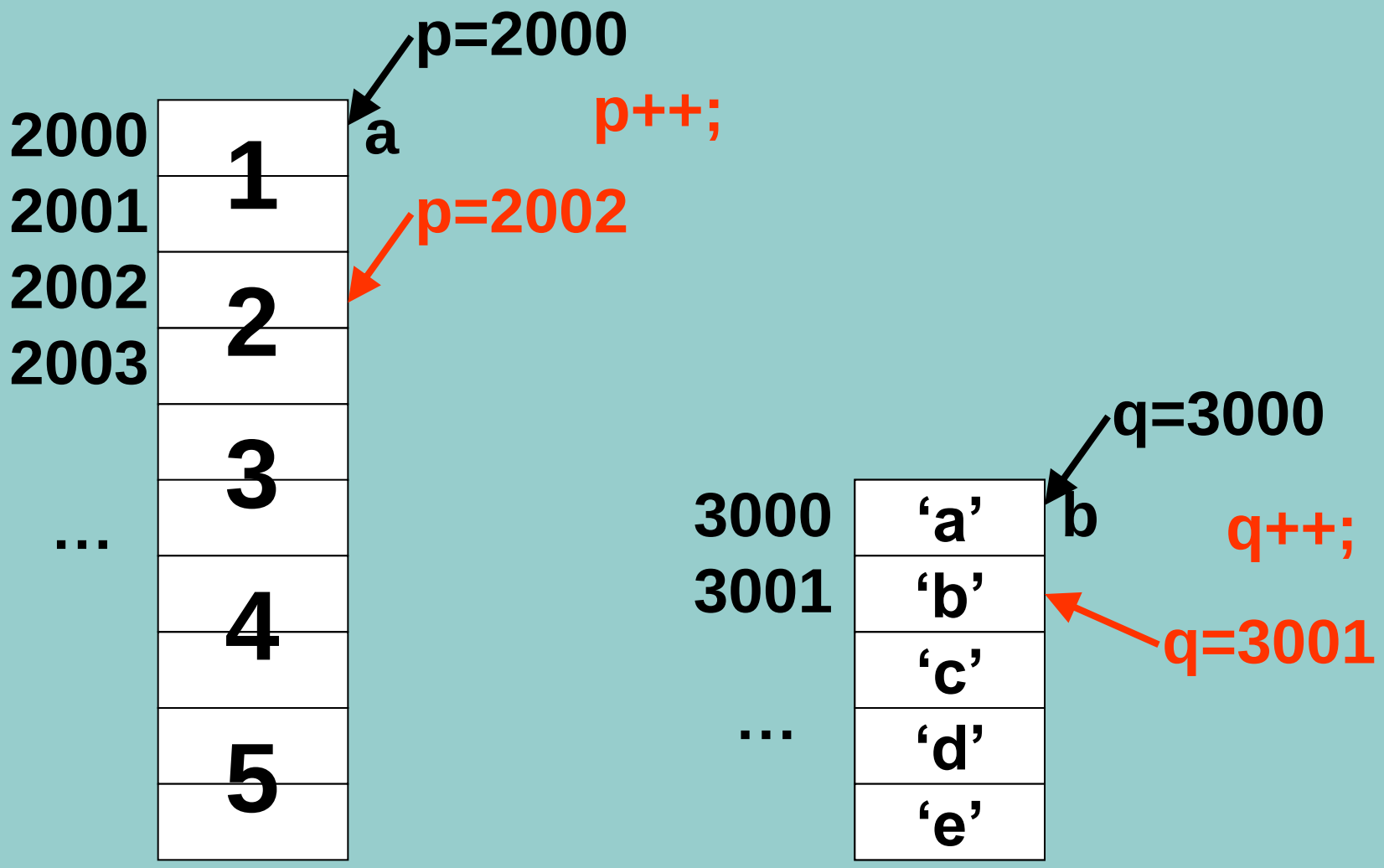
C++补充知识1—指针

```
#include <stdio.h>
int main(int argc, char* argv[])
{    int x, *y;
     x=10;                y=&x;
     printf("%d\n",x);
     printf("%ld\n",&x);
     printf("%d\n",*y);   printf("%ld\n",y);
     return 0;
}
```

C++补充知识1—指针

不同类型的指针变量的区别：

- int　　*p , a[5]={1,2,3,4,5};
- char　*q , b[5]={‘a’,’b’,’c’,’d’,’e’};
- p=a;　　q=b;
- printf(“%d\n”,*p);　printf(“%c\n”,*q);
- p++;　　q++;
- printf(“%d\n”,*p);　printf(“%c\n”,*q);


p=2000
p++;
2000
2001
2002
2003
1
2
3
4
5
a
p=2002
...
q=3000
3000
3001
‘a’
‘b’
‘c’
‘d’
‘e’
b
q++;
q=3001
...

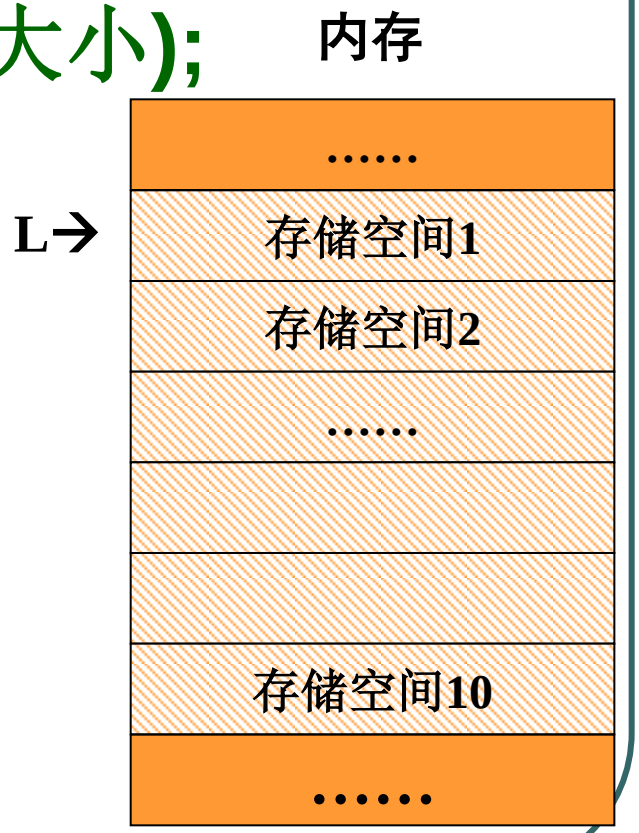
C++补充知识2—动态分配存储空间

首次分配存储空间：

- L=(返回值类型)malloc(空间大小);

例：

- L=(int *)malloc(10*sizeof(int));



malloc函数：

- 分配一定大小的内存空间；
- 以字节为分配单位；
- 与New功能相同。

C++补充知识2—动态分配存储空间

```
#include <stdio.h>
#include <malloc.h>
main()
{int *p,i;
 clrscr();
 p = new int[10];
 for (i=0;i<10;i++)
        {p[i]=i;
         printf("%d ",p[i]);
        }
}
```

C++补充知识2—动态分配存储空间

追加存储空间

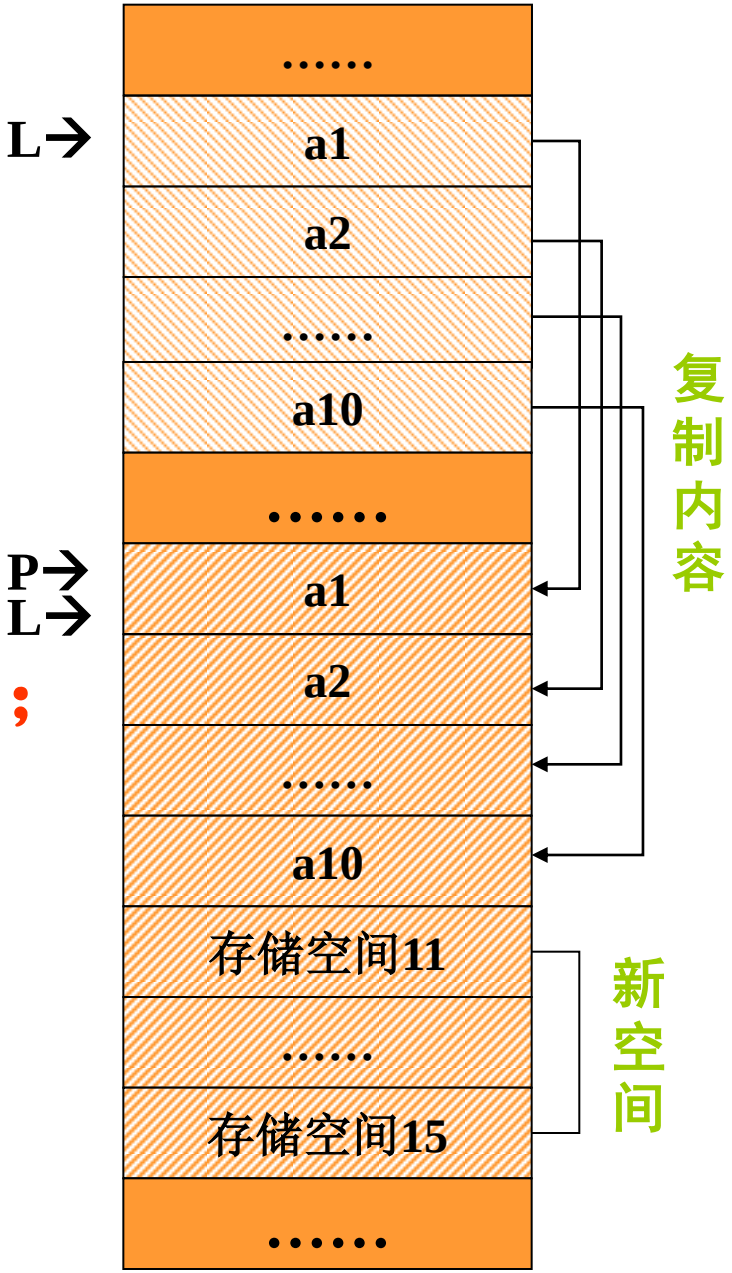
- **P = (返回值类型)realloc**
  (旧空间首地址，新空间大小);

例：

- **P=(int *)realloc(L,15*sizeof(int));**
- **L=P;**

realloc函数：

- 分配一定大小的内存空间；
- 以字节为分配单位；
- 将原空间内容复制到新空间；
- 与New功能稍有不同。



C++补充知识3—结构类型

- 结构类型定义

```
struct student
{
  int    num;          /* 学号 */
  float ensco;         /* 英语成绩 */
  float mathsco;       /* 数据成绩 */
};
```

C++补充知识3—结构类型

- 结构类型变量定义

  student stu1,stu2[10];

- 结构类型变量赋值

  stu1.num=1;

  stu1.ensco=73.5;

  stu1.mathsco=90.0;

  stu2[1].num=5;

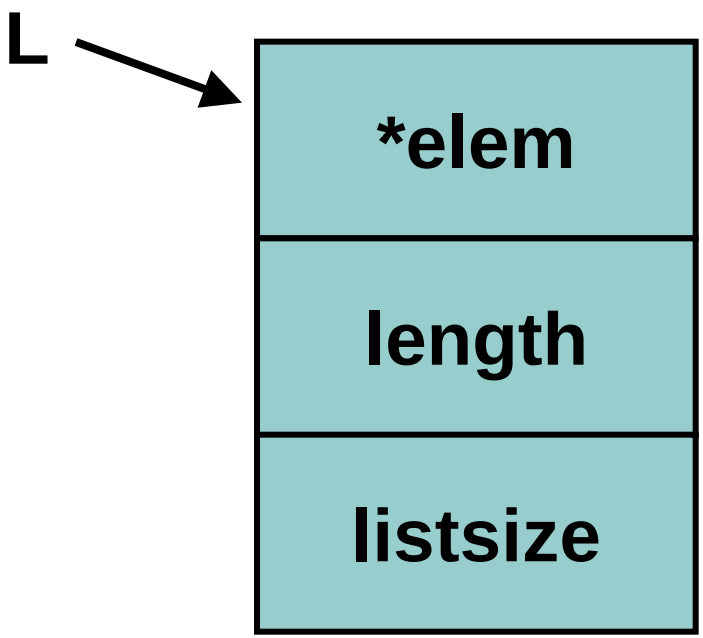
  stu2[1].ensco=88.0;

  stu2[1].mathsco=67.5;

线性表的顺序表示

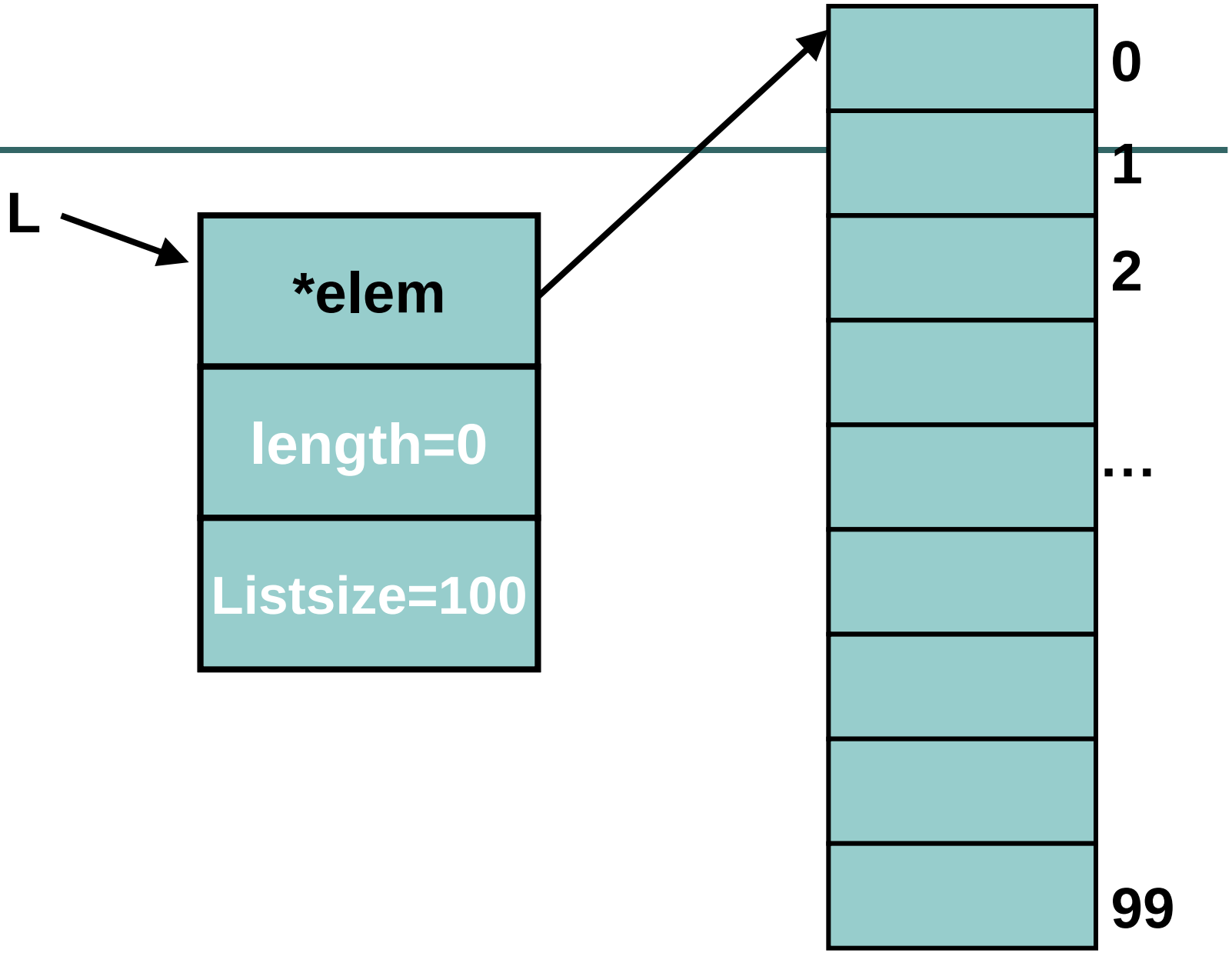
```
#define   L_I_S     100
#define   LI        10
struct SqList{
   ElemType   *elem;
   int        length;
   int        listsize;
} ;
```

定义变量：　　SqList　L;



线性表的初始化：

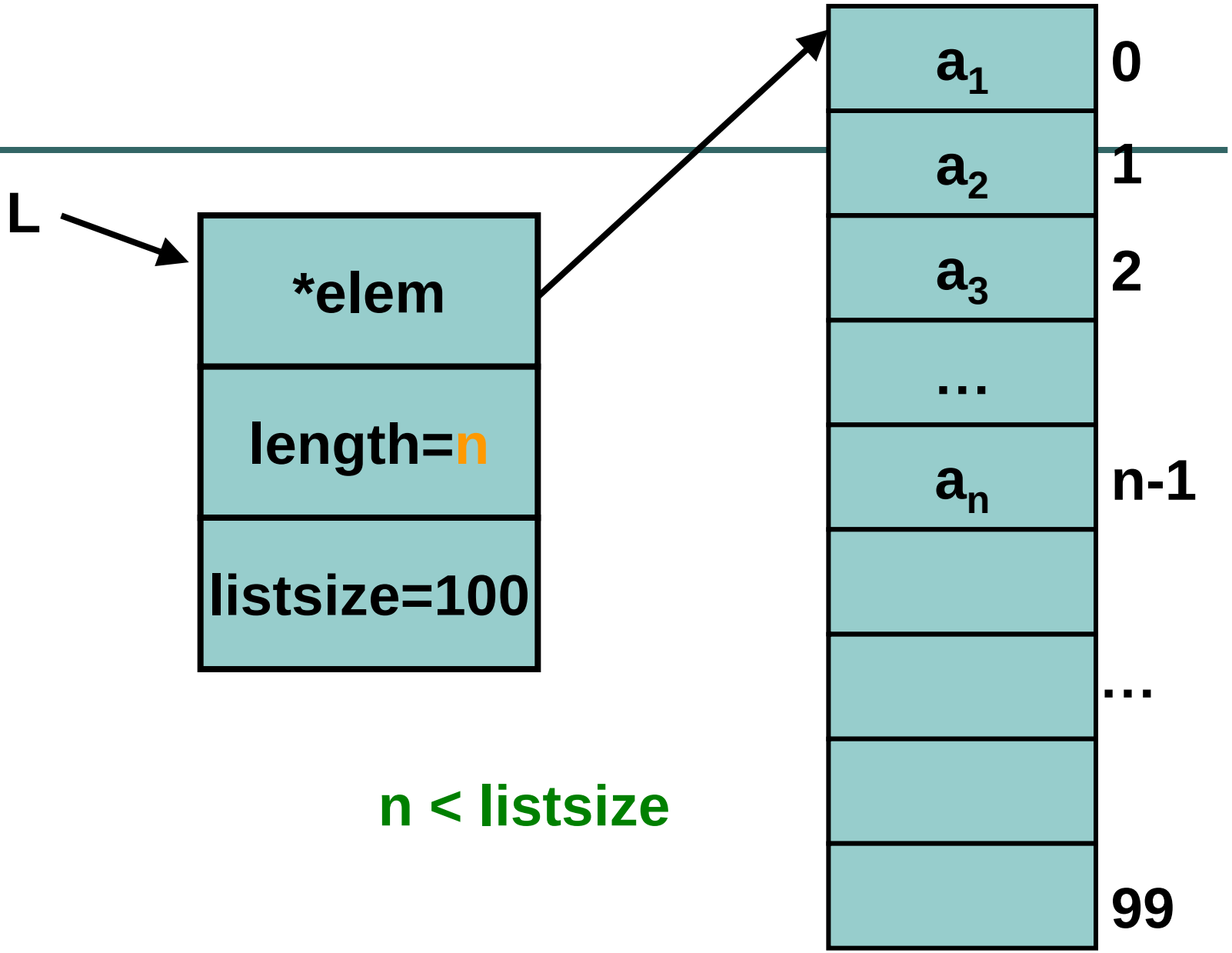
```
Status InitList_Sq(SqList &L) {
 //构造一个空的线性表L
 L.elem = new  ElemType[L_I_S];
 if (!L.elem)   return(OVERFLOW);
 L.length=0;
 L.listsize=L_I_S;
 return OK;
 } //InitList_Sq
```

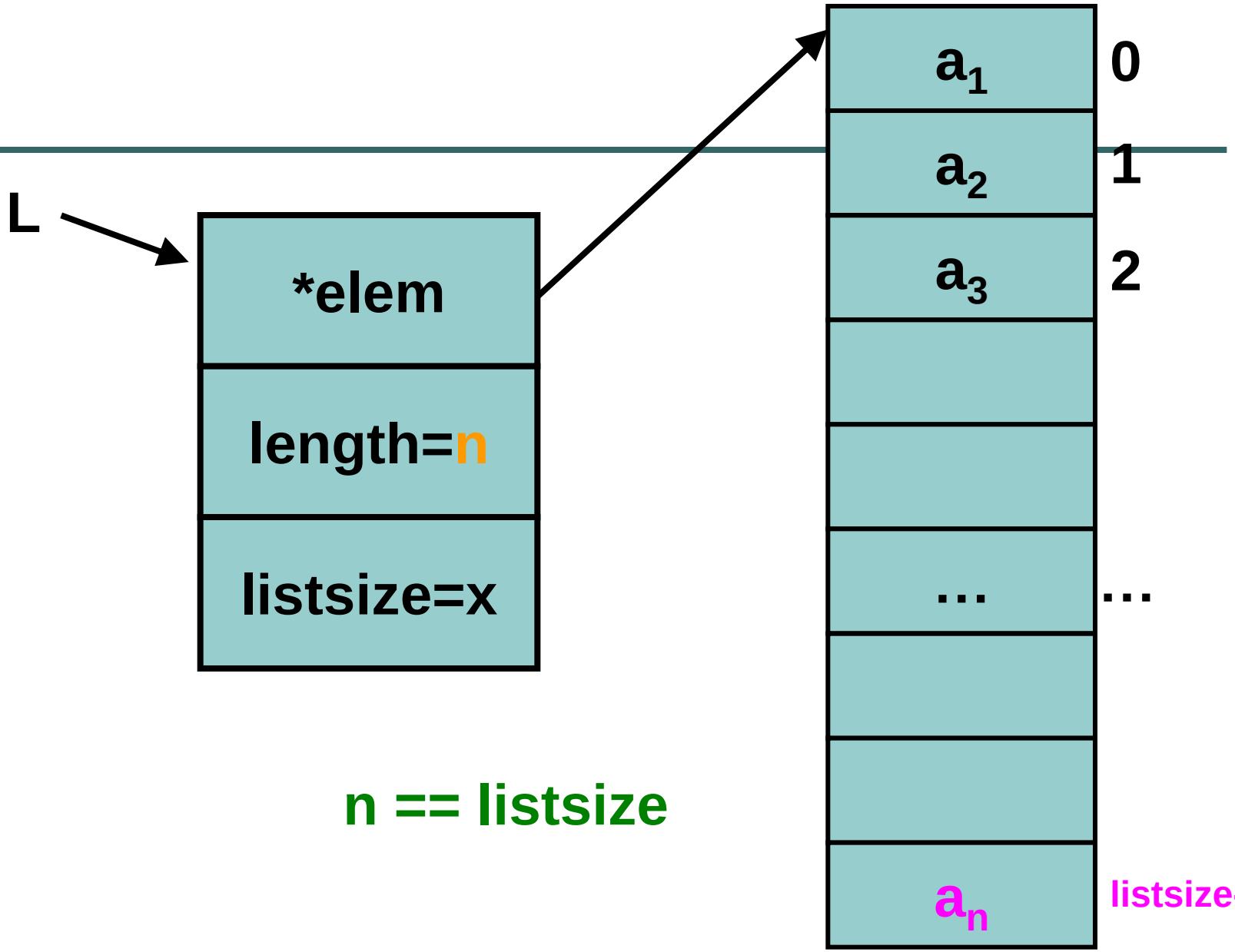


插入操作：

```
Status ListInsert_Sq(Sqlist &L, int i,
ElemType e)
{ // 在顺序表L的第i个位置之前插入新的元素e
  // i的合法值为1<=i<=ListLength_Sq(L) + 1

   // 判断i值是否合法
  if(i<1 || i>L.length+1) return ERROR;
```

```
    q=&(L.elem[i-1]);          // q为插入位置
    for(p=&(L.elem[L.length-1]); p>=q; --p)
        *(p+1)=*p
        // 插入位置及之后元素后移
    *q=e;                      // 插入e
    ++L.length;                // 表长增1
    return OK;
} //ListInsert_Sq
```




L
*elem
length=n
listsize=x
a1
a2
a3
…
an
0
1
2
…
listsize-1
n == listsize

添加程序段：

```
If  (L.length>=L.listsize){
      // 当前存储空间已满,增加分配
      newbase = new  ElemType[L.listsize+LI];
      if (!newbase)  exit(OVERFLOW);
                        // 存储分配失败
      L.elem=newbase;
      L.listsize+=LI;              // 增加存储容量
 }
```

算法2.4的时间复杂度：

- 插入一个元素所需平均移动次数E_{is}：

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

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

因此, 算法2.4的时间复杂度为： O(n)

删除操作1：

```
Status ListDelete_Sq (Sqlist &L,int i,ElemType *e)
{  if((i<1)||(i>L.length)) return ERROR;
                                        //i值不合法
    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;
} //ListDelete_Sq
```

删除操作2：

```
Status ListDelete_Sq (Sqlist &L,int i,ElemType *e)
{  if((i<1)||(i>L.length)) return ERROR;
                                    //i值不合法
    *e=L.elem[i-1];          // 被删除元素的值赋给e
    for(j=i;j<=L.length-1;j++)
       L.elem[j-1]=L.elem[j];
                              // 被删除元素之后的元素上移
    --L.length;              // 表长减1
    return OK;
} //ListDelete_Sq
```

算法2.5的时间复杂度：

- 删除一个元素所需平均移动次数E_{ds}：

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

因此, 算法2.5的时间复杂度为： O(n)

算法2.6：线性表定位函数（P25）

算法2.7：归并非递减线性表La、Lb到非递减线性表Lc（P26）

思考两种方式定义线性表的区别：

```
typedef  struct {
   ElemType   *elem;
   int              length;
   int              listsize;
} SqList ;
```

```
typedef  struct {
   ElemType   elem[100];
   int              length;
   int              listsize;
} SqList ;
```

2.3 线性表的链式表示与实现

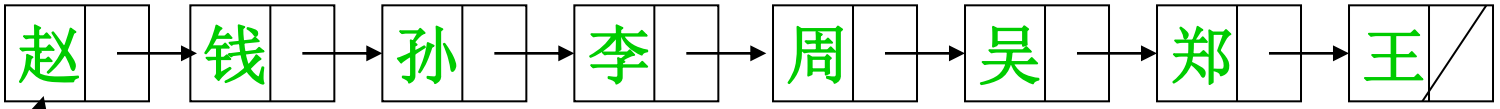
- 顺序表示的优点是可随机存取表中的任意元素；
- 顺序表示的弱点是在作插入或删除操作时，需移动大量元素。

链式表示——没有顺序表示的弱点，但同时也失去了顺序表示的优点。

2.3.1 线性链表

- 线性表的链式表示：用一组任意的存储单元（可连续也可不连续）存储线性表的数据元素。

- 例： 线性表
  - （赵，钱，孙，李，周，吴，郑，王）的链式表示



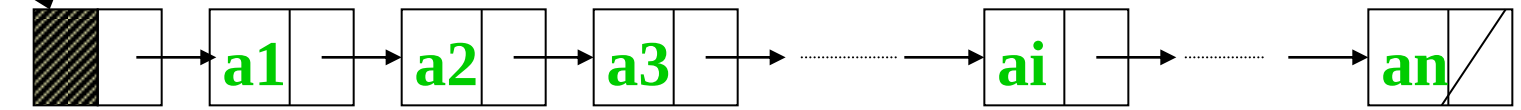
H

头指针H：

31

存储地址	数据域	指针域
1	李	43
7	钱	13
13	孙	1
19	王	NULL
25	吴	37
31	赵	7
37	郑	19
43	周	25

链表中相关名词与其结构定义：



- 结点、数据域、指针域、头结点、头指针
- 链表、单链表（线性链表）

C++中的定义：

```
struct LNode  {
   ElemType data;
   LNode *next;
 };
 typedef LNode *LinkList;
```

结构变量中“.”和“->”的区别：

```
#include <stdio.h>
struct stud {
      int num ;
      float scoEn;
      float scoMa;
 };
typedef struct stud stud1,*stud2;

int main()
{stud1 student1;        //结构体变量
 stud2 p;               //结构体指针变量

 p=&student1;
```

```
student1.num=1;
student1.scoEn= 90.5;
student1.scoMa=84;

printf("%d,%f,%f\n“ , student1.num , student1.scoEn , student1.scoMa);

printf("%d,%f,%f\n“ , p->num , p->scoEn , p->scoMa);

return 0;
}
```

初始化单链表的操作：

```
Status InitList_L(LinkList L)
{  //创建以L为头指针的空单链表
   L = (Lnode *)malloc(sizeof(Lnode));
   if( !L ) return OVERFLOW;
   L->next=NULL;
   return OK;
} // InitList_L
```

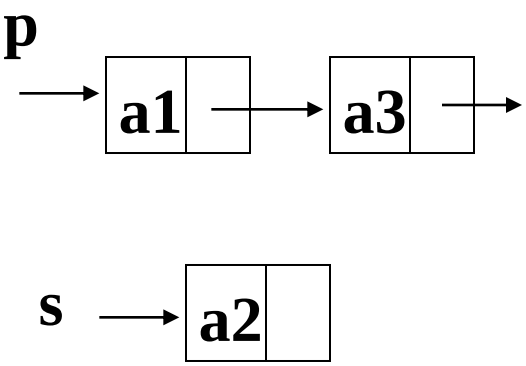
从单链表中取元素的操作：

```
Status GetElem_L(LinkList L,int i,ElemType &e)
{  //L为带头结点的单链表的头指针;当第i个元素
   //存在时，其值赋给e并返回OK,否则返回ERROR
   p = L->next;        j = 1;
   while ( p && j<i ){
         p = p->next;        ++j;
   }
   if( !p || j>i ) return ERROR;
   e = p->data;
   return OK;
} // GetElem_L
```

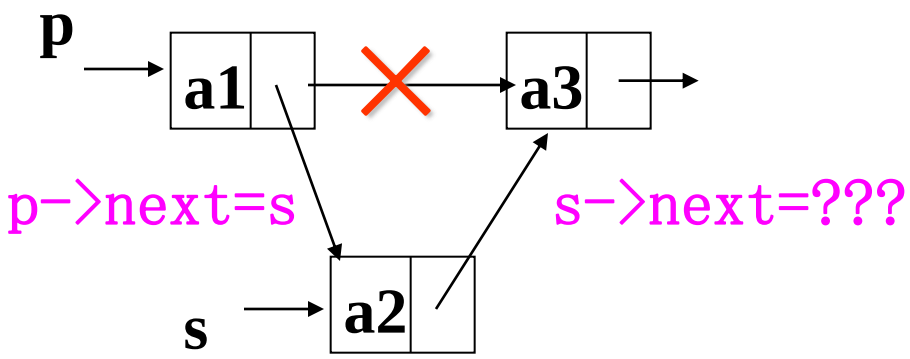
单链表中结点的插入

（在指针p所指的结点后插入指针s所指的结点）

插入前：



插入后：

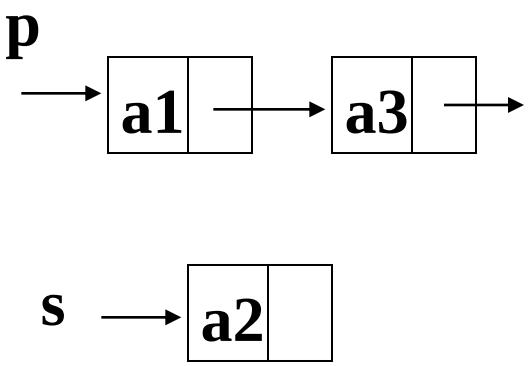


- ~~p->next = s;~~
- ~~s->next = ? ? ?~~

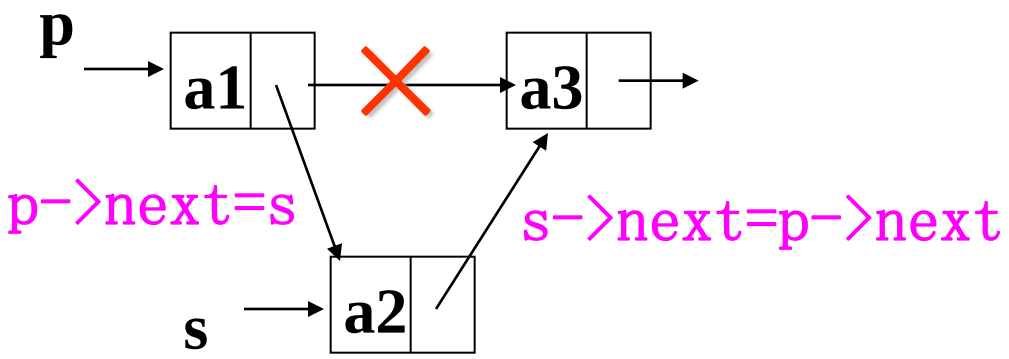
单链表中结点的插入

（在指针p所指的结点后插入指针s所指的结点）

插入前：



插入后：

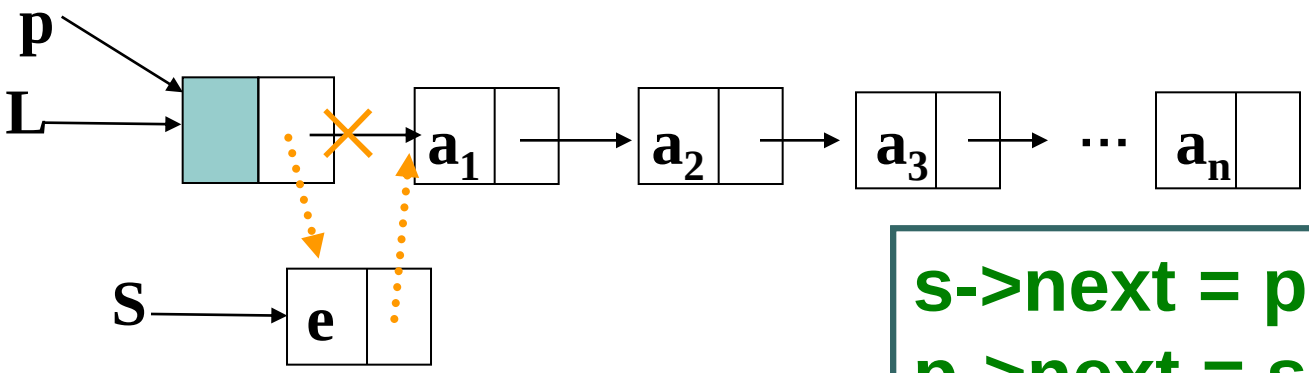



```
s->next = p->next;
p->next = s;
```

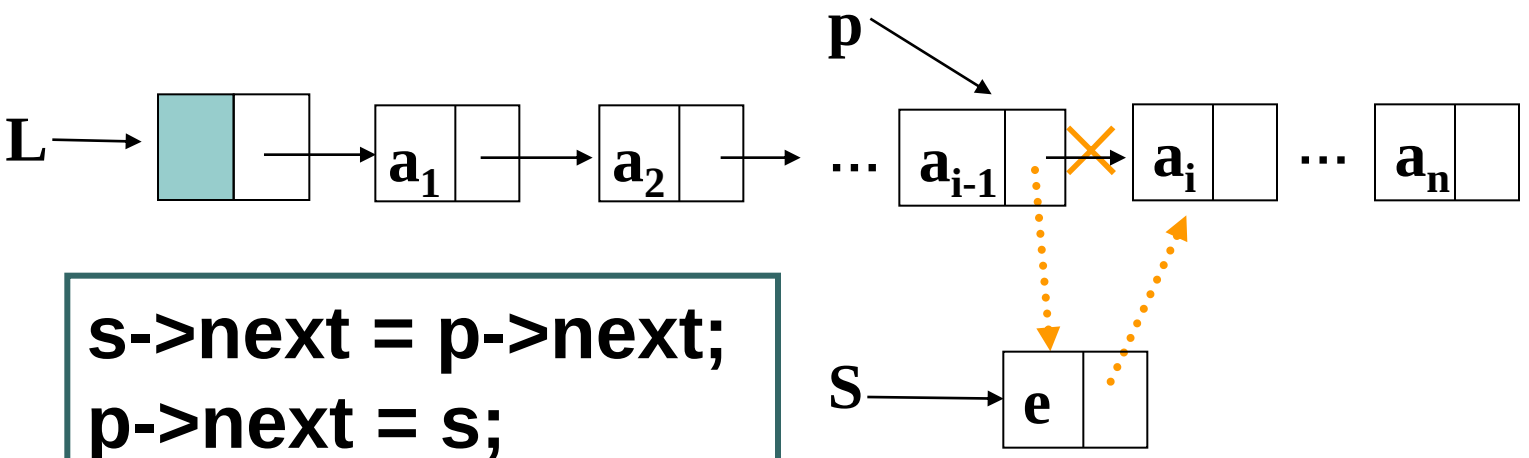
算法2.9：插入结点程序

```
Status ListInsert_L(LinkList &L, int i, ElemType e)
{  //在带头结点的单链表L中第i个位置之前插入元素e
   p = L;   j = 0;
   while(p&&j<i-1)  {p=p->next;  ++j}
   if(!p||j>i-1) return ERROR;
   s = (Lnode *)malloc(sizeof(Lnode));
   if(!s) return OVERFLOW;
   s->data = e;
   s->next = p->next;   p->next = s;
   return OK;
}//LinstInsert_L
```

带头结点的单链表插入结点方法：

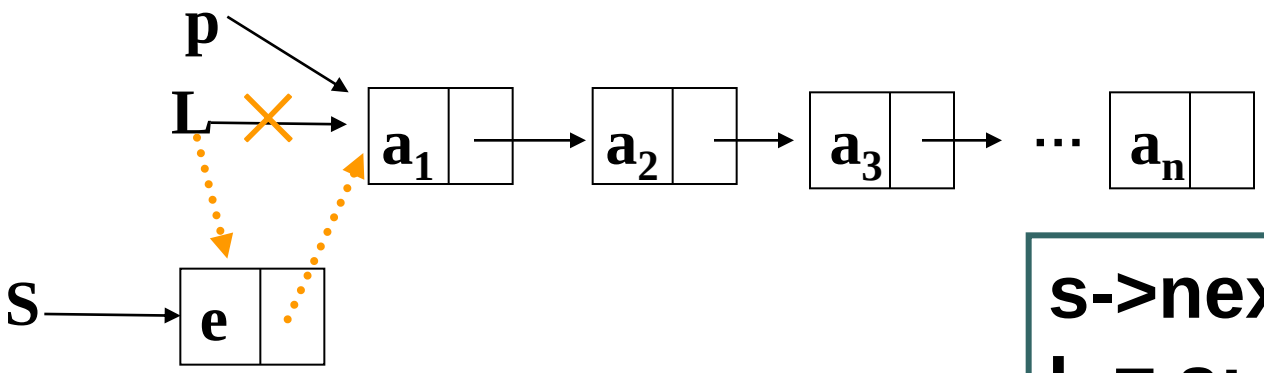



```
s->next = p->next;
p->next = s;
```

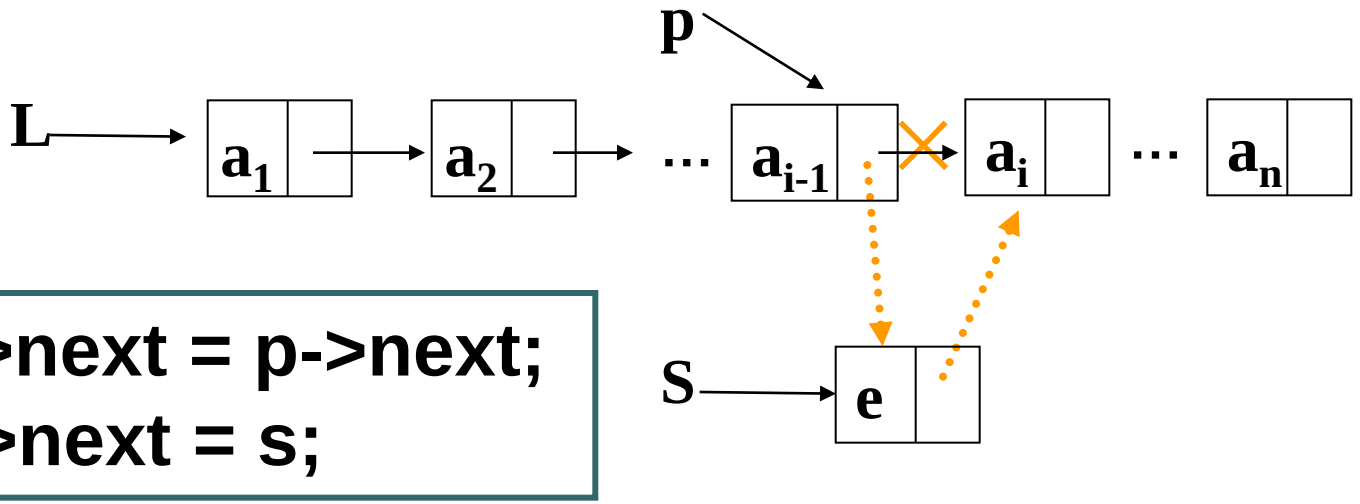



```
s->next = p->next;
p->next = s;
```

不带头结点的单链表插入结点方法：



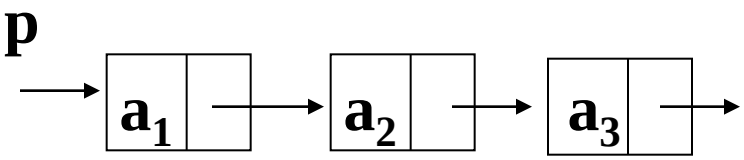

```
s->next = L;
L = s;
```



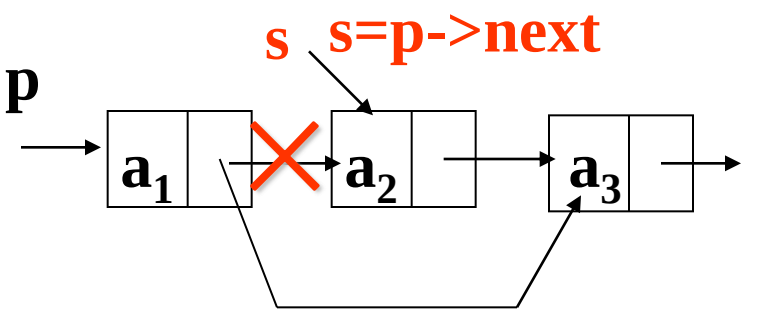

```
s->next = p->next;
p->next = s;
```

单链表中结点的删除（删除指针p后面的结点）

删除前：



删除后：



释放结点后：

free(s);

- p->next = p->next ->next;　　free(s);

算法2.10：删除结点程序

```
Status ListDlete_L(LinkList L, int i, ElemType &e)
{  //删除带头结点的单链表L中第i个元素，并由e带回其值
   p = L;          j = 0;
   while(p->next && j<i-1)  {p=p->next;   ++j}
   if(!(p->next)||j>i-1) return ERROR;
   s = p->next;
   p->next = s->next;
   e = s->data;
   free(s);
   return OK;
}
```

试比较不带头结点和带头结点的单链表删除结点方法的不同

建立链表方法1—手工链表：

```
#include <stdio.h>
struct stud {
    int num ;
    struct stud *next;
 };
typedef struct stud stud1,*stud2;

int main()
{stud1 a1,a2,a3;
 stud2 L,p;
 int i;

 a1.num=1;    a2.num=2;    a3.num=3;
```

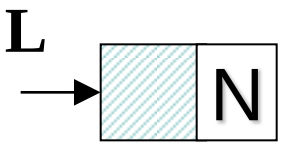
```
//建立链表：
L=&a1;
 a1.next=&a2;    a2.next=&a3;   a3.next=NULL;

//循环遍历链表：
 p=L;
 while (p!=NULL)
   {printf("%d\n",p->num);
    p=p->next;
   }
 return 0;
}
```

建立链表方法2—调用基本操作（函数）：

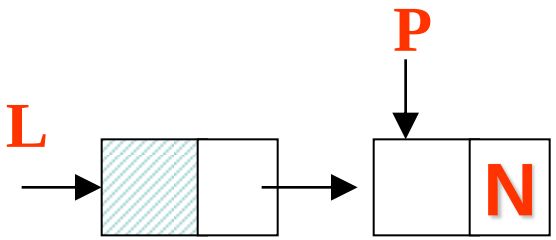
```
#include <stdio.h>
int main()
{ LinkList L;
  int i,a[5]={1,2,3,4,5};

  InitList_L(L);
  for (i=0;i<5;i++)
        ListInsert_L(L , i , a[i]);
 return 0;
}
```

建立链表方法3：逆位序输入n个元素的值，建立带头结点的单链线性表L

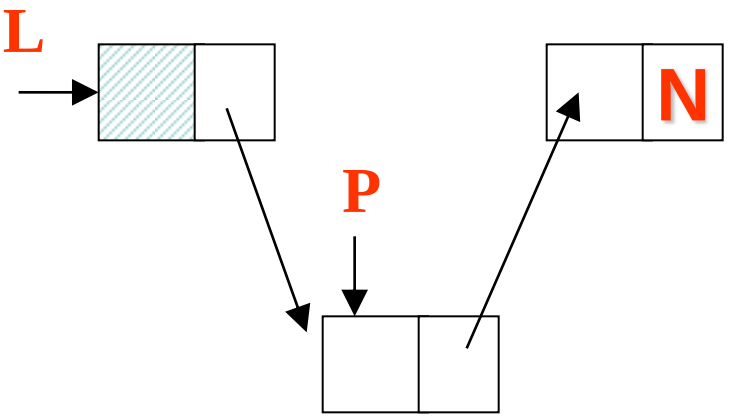
初始状态：



插入第一个结点：

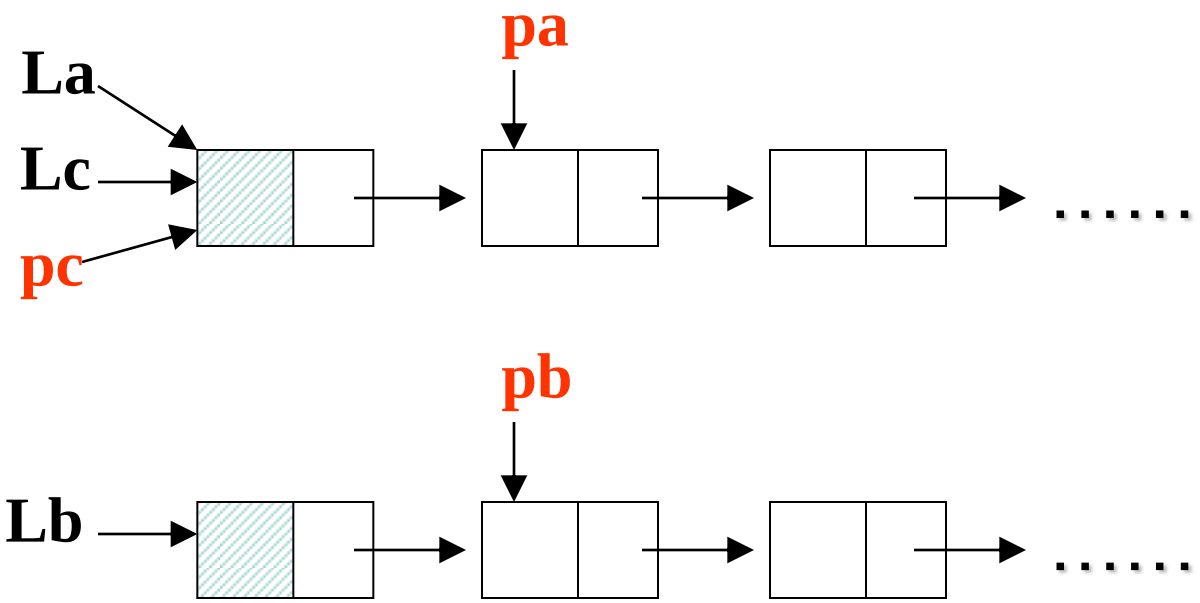


插入第二个结点：




```
P->next=L->next;
L->next=P;
```

算法2.12：已知单链线性表La和Lb的元素非递减排列，归并La和Lb得到新的单链线性表Lc，Lc的元素也非递减排列。



注意：在该算法中，pa总是指向La中待比较的结点
pb总是指向Lb中待比较的结点
pc总是指向Lc中最后一个结点

静态链表：借用一维数组来描述线性链表

```
//----------线性表的静态单链表存储结构----------
#define MAXSIZE 100     // 链表的最大长度
typedef struct {
    ElemType     data;
    int                cur;       // 游标
}coponent, Slinklist[MAXSIZE];
```

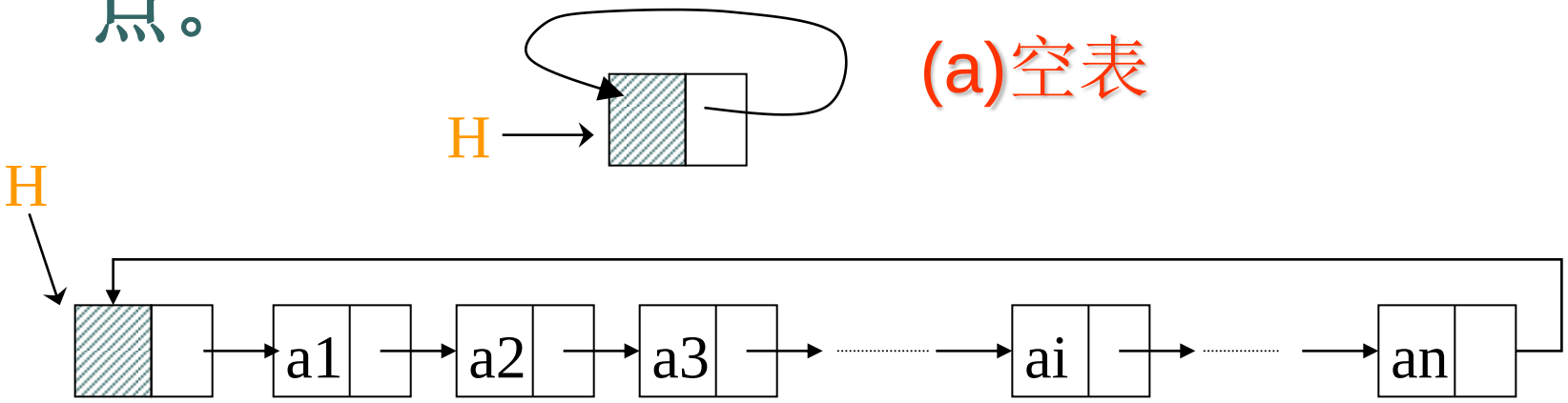
```
//------------线性表的单链表存储结构------------
typedef struct Lnode{
    ElemType     data;
    struct Lnode *next;
}Lnode, *Linklist;
```


相当于头结点
第一个结点下标
0 1
1 赵 2
2 钱 3
3 孙 4
4 李 5
5 周 6
6 吴 7
7 郑 8
8 王 0
9
10
(a)初始状态
0 1
1 赵 2
2 钱 3
3 孙 4
4 李 9
5 周 6
6 吴 7
7 郑 8
8 王 0
9 史 5
10
(b)在“李”后插入“史”
0 1
1 赵 2
2 钱 3
3 孙 4
4 李 9
5 周 6
6 吴 8
7 郑 8
8 王 0
9 史 5
10
(c)删除“郑”

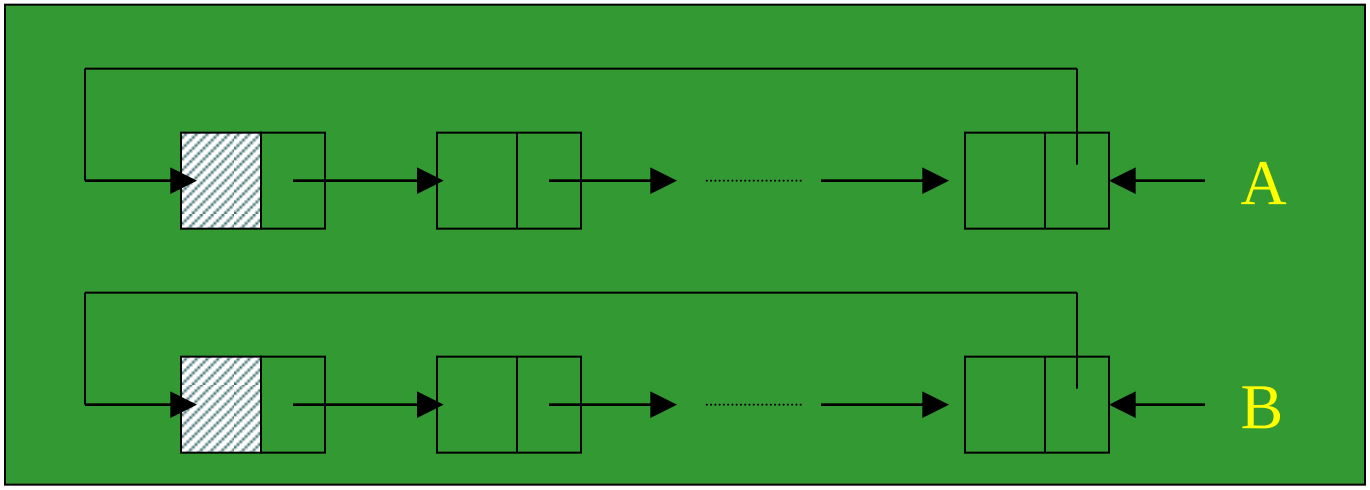
2.3.2 循环链表

- 循环链表的特点：表中的最后一个结点的指针域指向头结点，整个链表形成一个环。
- 从表的任意结点出发可以找到表中其它结点。

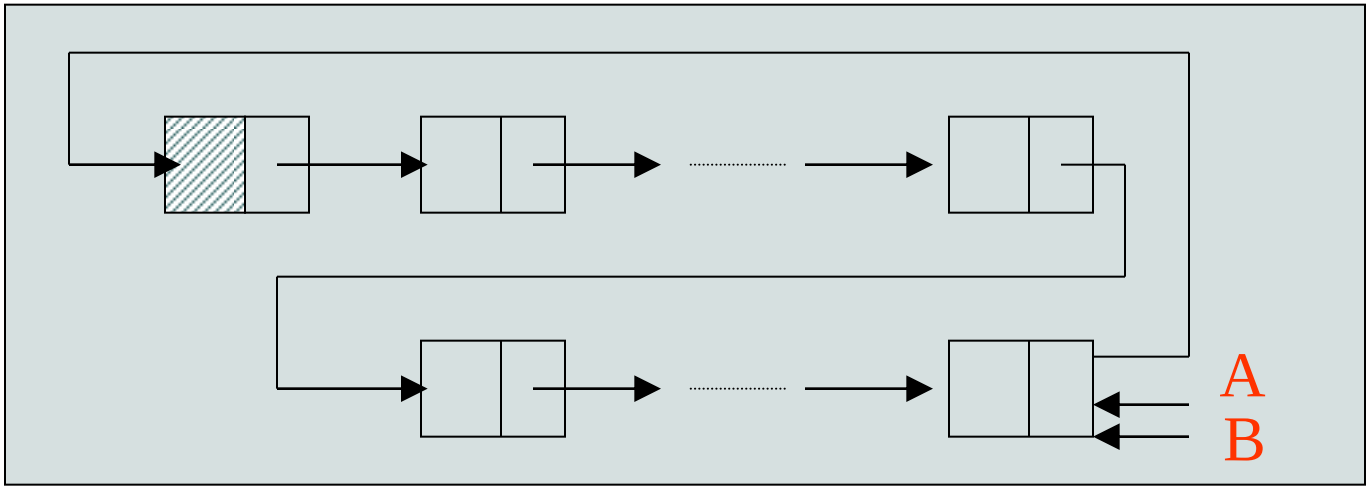
循环链表和线性表的操作差别：仅在于算法中的循环条件不是p或p->next是否为空，而是它们是否等于头指针。



- 有时，在循环链表中设立尾指针而不设头指针可简化某些操作。如：合并两个线性表



(a)合并前



(b)合并后

思考：合并算法

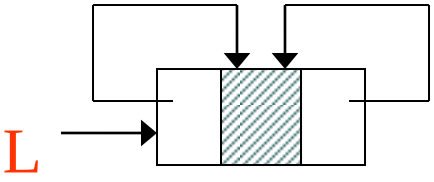
2.3.3 双向链表

- 双向链表的特点——表中的每个结点有两个指针域，一个指向后继结点，一个指向前趋结点， 整个链表形成两个环。

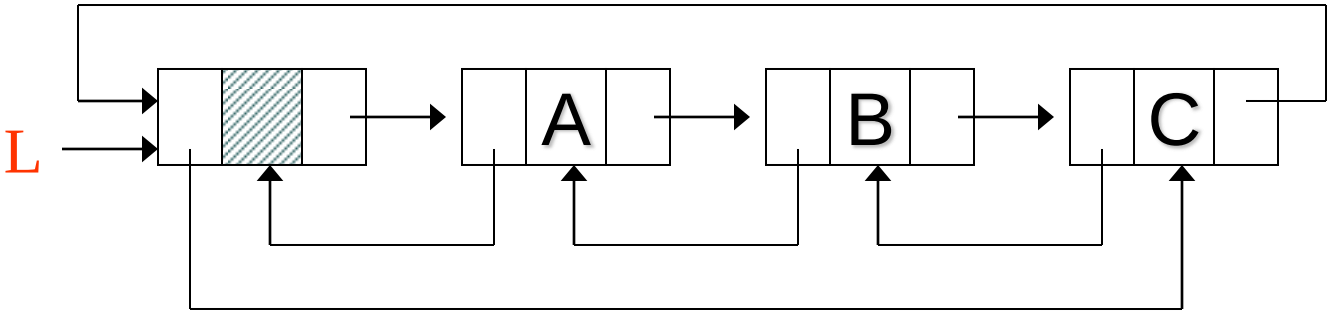
- 从表的任意结点出发可以通过正向环（或反向环）找到表中其它结点。

```
//----线性表的双链表存储结构----
typedef struct DuLnode{
    ElemType          data;
    Struct DuLnode  *prior;
    Struct DuLnode  *next;
}DuLnode, *DuLinklist;
```

结点：



(a)空表



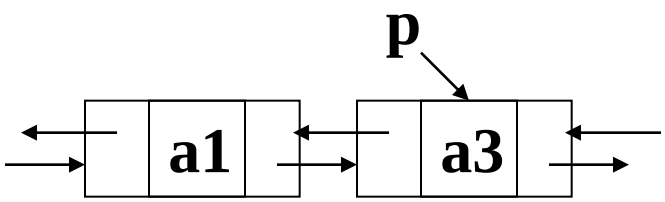
(b)非空表

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

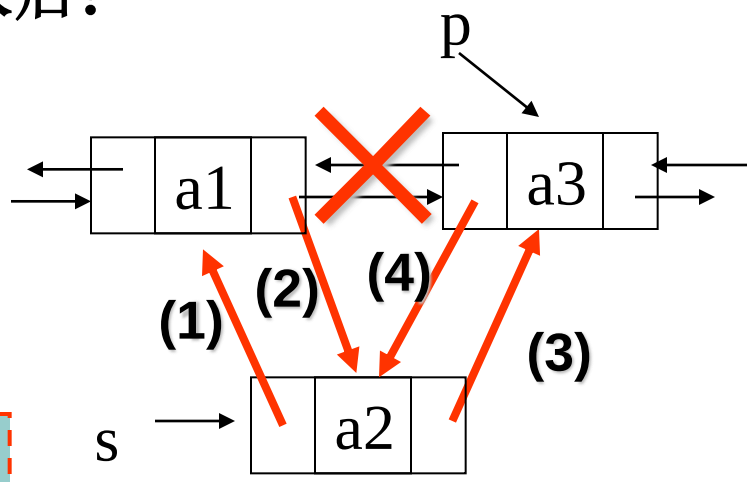
双向链表中结点的插入

（在指针p所指的结点前插入指针s所指的结点）

插入前：



插入后：



- s->prior = p->prior ;
- p->prior->next = s ;
- s->next = p ;
- p->prior = s ;

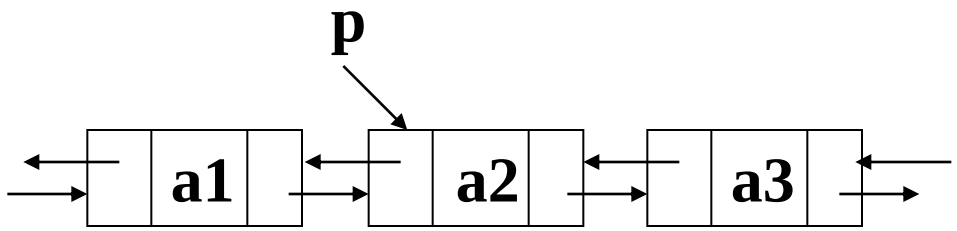
算法2.18：插入结点程序

```
Status ListInsert_DuL(DuLinklist L, int i, ElemType e)
{DuLinklist s,p;
  if (!(p=GetElemP_DuL(L,i)))
                    // 在L中确定第i个元素的位置指针p
      return ERROR;
  if  (!(s = (DuLinklist)malloc(sizeof(DuLNode))))
      return ERROR;
  s->data = e;              // 构造数据为e的结点s
  s->prior = p->prior;          p-> prior ->next = s;
  s->next = p;                  p->prior = s;
  return OK;
} // ListInsert_DuL
```

双向链表中结点的删除

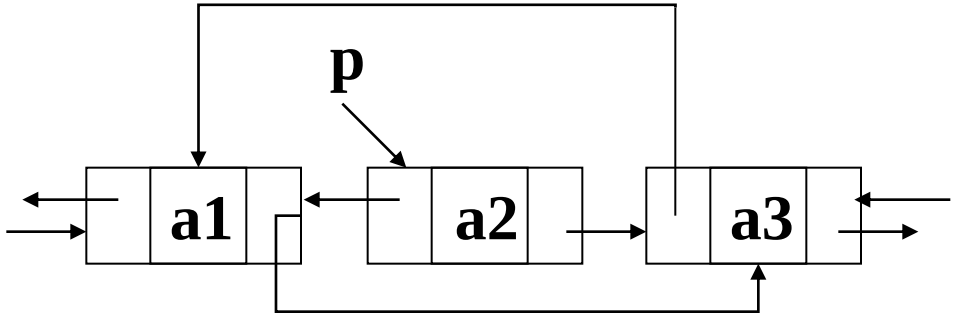
（删除指针p所指的结点）

删除前：



p->next->proir =p->prior

删除后：



p->prior->next =p->next

释放结点： free(p) ;

算法2.19：删除结点程序

```
Status ListDelete_DuL(DuLinklist L, int i, ElemType &e)
{DuLinklist p ;
  if (!(p=GetElemP_DuL(L,i)))
     return ERROR ; // 确定第i个元素的位置指针p
  e = p->data ;          // 删除的结点的值存入e
  p-> prior ->next = p->next ;
  p->next->prior = p->prior ;
  free(p) ;
  return OK ;
} // ListDelete_DuL
```

2.4 一元多项式的表示及相加

一元多项式按升幂可写为：

$$P_n(x) = p_0 + p_1 x + p_2 x^2 + \cdots + p_n x^n$$

因此，在计算机中，可用一个线性表P来表示：

$$P = (p_0, p_1, p_2, \cdots, p_n)$$

假设$Q_m(x)$是一元m次多项式，同样可以表示为：

$$Q = (q_0, q_1, q_2, \cdots, q_m)$$

设m<n，则两个多项式P、Q相加的结果可用线性表R来表示：

$$R = (p_0+q_0, p_1+q_1, p_2+q_2, \cdots, p_m+q_m, \cdots, p_n)$$

显然，对P、Q和R采用顺序存储结构即可解决多项式相加问题。

但是，在处理形如 $S(x)=1+3x^{10000}+2x^{20000}$

的稀疏多项式（有很多项系数为0）时，用顺序存储结构存储数据会造成大量的空间浪费。如果只存储非零系数项则必须存储相应的指数。

一般情况下，一元n次多项式可写为：

$$P_n(x) = p_1 x^{e_1} + p_2 x^{e_2} + \cdots + p_m x^{e_m}$$

其中，$0 \le e_1 < e_2 < \cdots e_m = n$

若用一个长度为m且每个元素有两个数据项（系数项和指数项）的线性表

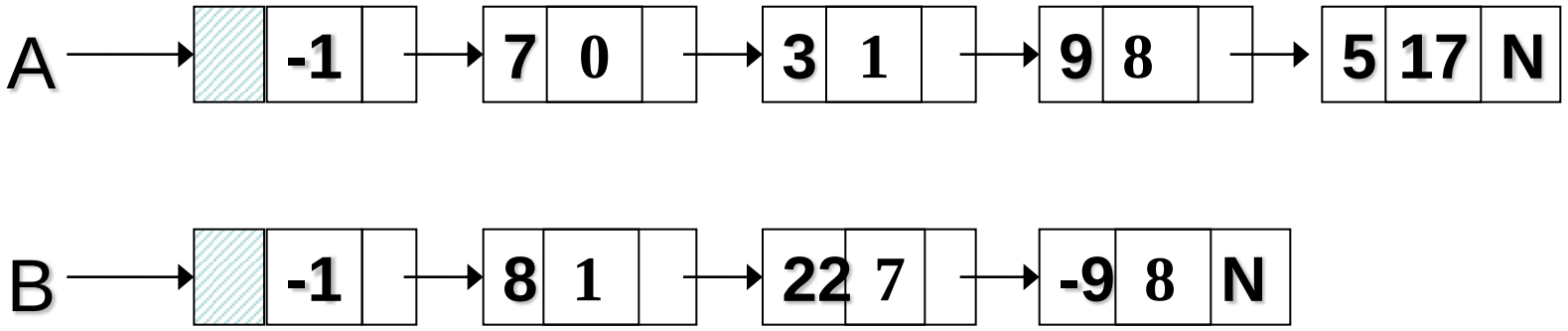
$$((p_1, e_1), (p_2, e_2), \cdots, (p_m, e_m))$$

便可唯一确定多项式Pn(x)。

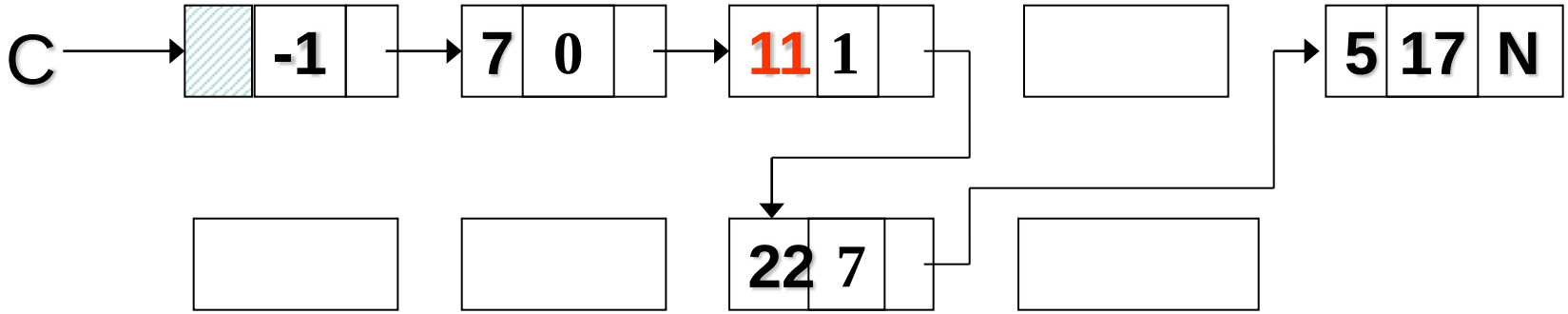
可有两种存储表示方法：顺序表示和链式表示

采用链式结构存储的两个一元多项式相加：

相加前：



相加后：



思考：如何求多项式A的导数？