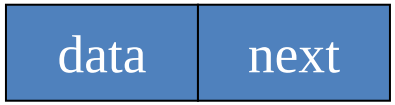
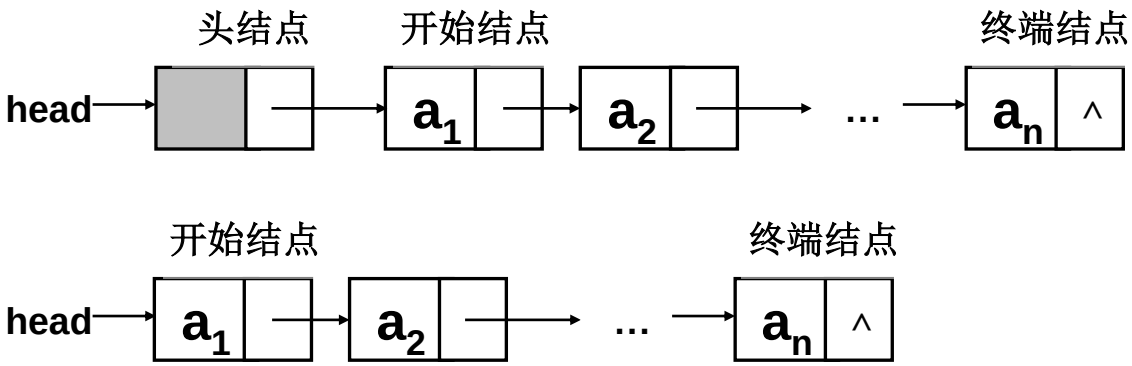
第二章 线性表

回顾



●**结点**：数据域 + 指针域（链域）

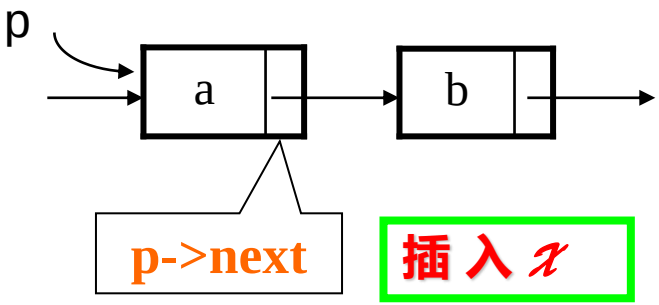
```
typedef struct Lnode {
    ElemType      data;     //数据域
    struct Lnode  *next;    //指针域
}Lnode, *LinkList;          // *LinkList为Lnode类型的指针
```

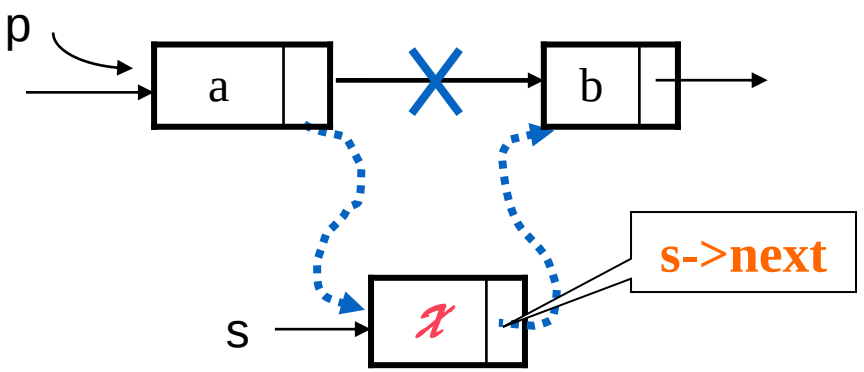


回顾

(1)插入元素

在链表中插入一个元素x的示意图如下：





链表插入的核心语句：

Step 1： s->next=p->next;
Step 2： p->next=s；

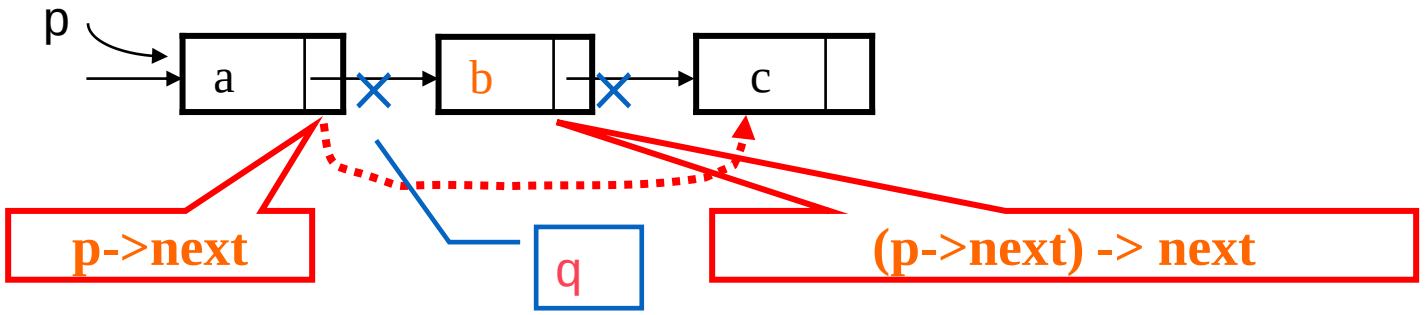
x 结点的生成方式：

```
s=(node*)malloc(m);
s->data=x;
s->next= ?
```

回顾

(2)删除元素

设p指向单链表中某结点，删除*p。操作示意图如图所示。要实现对结点*p的删除，首先要找到*p的前驱结点*q，然后完成指针的操作即可。

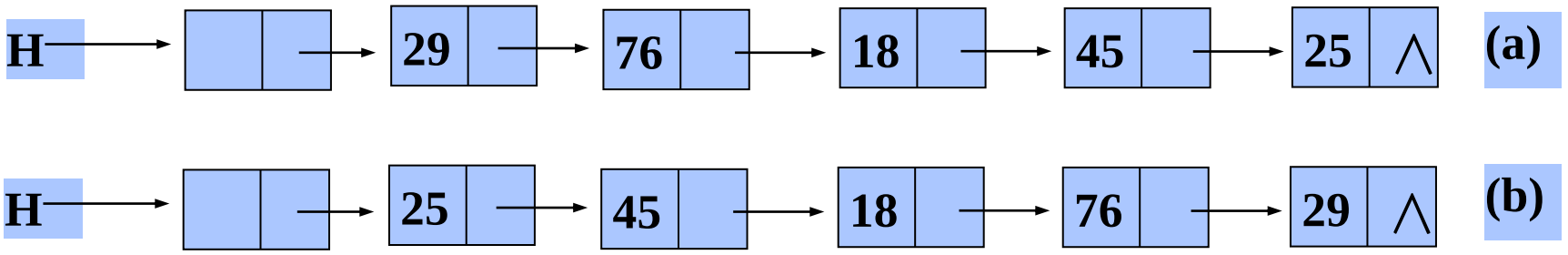


删除动作的核心语句（要借助辅助指针变量q）：

```
q = p->next;          //首先保存b的指针，靠它才能找到c;
p->next=q->next;  //将a、c两结点相连，淘汰b结点;
free(q)；                 //彻底释放b结点空间
```

回顾

例 已知单链表H，写一算法将其倒置。即实现如图2.22的操作。(a)为倒置前，(b)为倒置后。



算法思路：

依次取原链表中的每个结点，将其作为第一个结点插入到新链表中去，指针p用来指向当前结点，p为空时结束。

回顾

```
void  reverse (Linklist H) {
   LNode  *p;
   p=H->next;
   H->next=NULL;
   while (p) {
       q =p;
       p=p->next;
       q->next=H->next; H->next=q;
    }
}
```

回顾

例：已知L为带头结点的单链表,请依照递归思想实现下列运算。

1、求链表中的最大整数

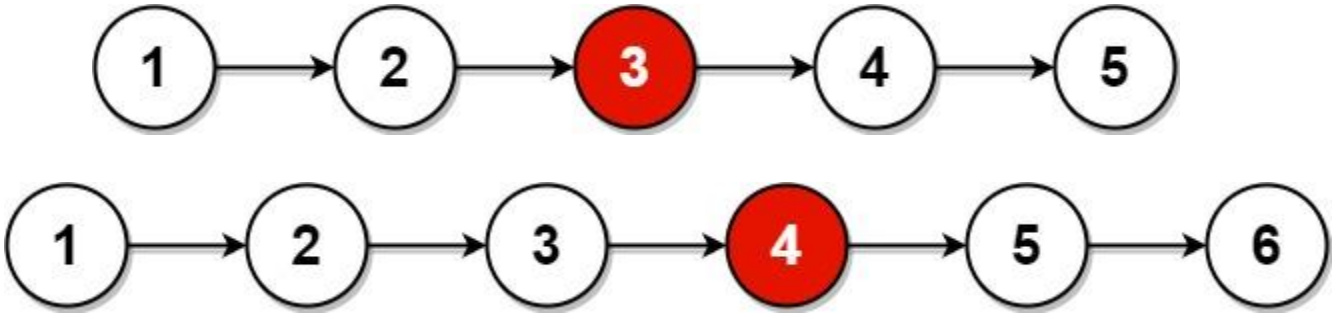
2、求表中的节点数?

3、求链表所有元素的平均值?

```
int getmax(Linklist L) {
   if (L == NULL) {
      return MIN;//宏定义MIN为一个很小的数例如-999999
   }
   int tmp = getmax(L->next);//获得下一个元素
   return tmp > L->data ? tmp : L->data;//比较两个元素大小
}
```

回顾

例 给你单链表的头结点 head ，请你找出并返回链表的中间结点。如果有两个中间结点，则返回第二个中间结点。



思路：快慢指针的典型运用

快慢指针：慢指针每次移动一个结点，快指针每次移动两个结点，当快指针到达NULL时，慢指针刚好就在中间结点的位置，即2/n的位置。

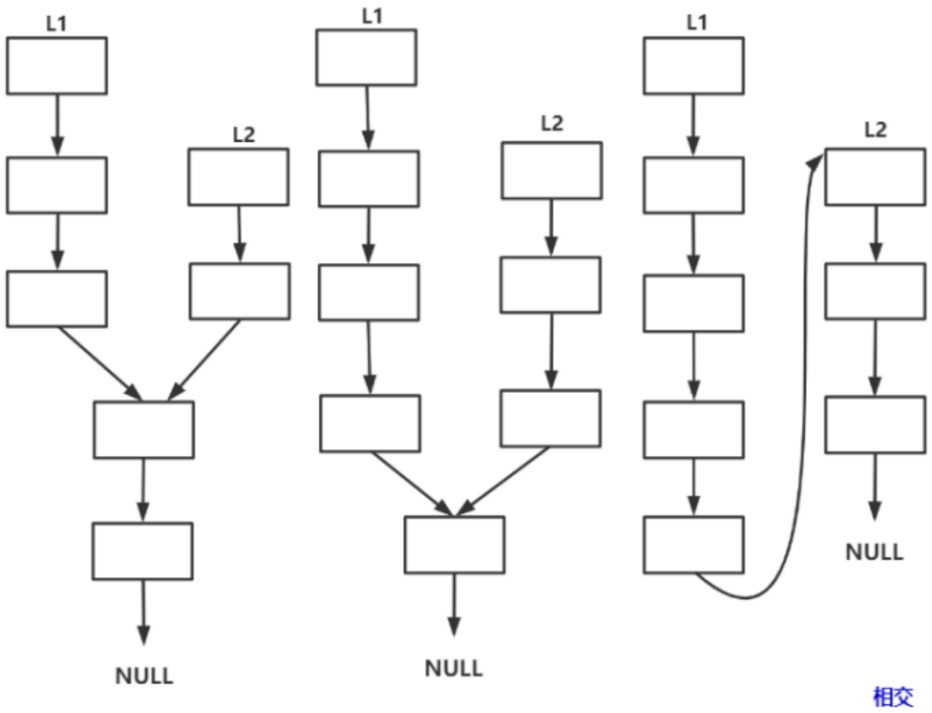
回顾

```
Lnode* middleNode(Linklist head) {
    Lnode* slow = head;
    Lnode* fast = head;
    while(fast && fast->next)
    //偶数个结点的判断条件是fast == NULL;
    //奇数个结点的判断条件是fast->next == NULL;
    {
        slow = slow->next;
        fast = fast->next->next;
    }
    return slow;
}
```

回顾

例：输入两个链表，找出它们的第一个公共结点。

检测两个链表是否相交。分别找到两个链表中的最后一个节点，然后检测这两个节点的地址是否相同即可如果地址相同则相交，否则不相交



2.3 线性表的链式表示和实现

- 2.3.1 线性表的链式存储—链表
- 2.3.2 单链表基本运算的实现
- 2.3.5 静态链表
- 2.3.3 循环链表
- 2.3.4 双链表

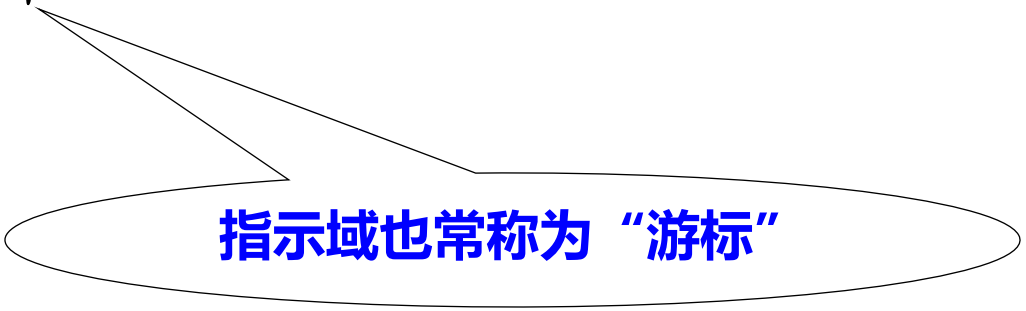
2.3.5 静态链表

问题：若某种高级语言没有指针类型，能否实现链式存储和运算？如何实现？

答：能！虽然链表通常用动态级联方式存储，但也可以用一片连续空间（一维数组）实现链式存储，这种方式称为静态链表。

2.3.5 静态链表

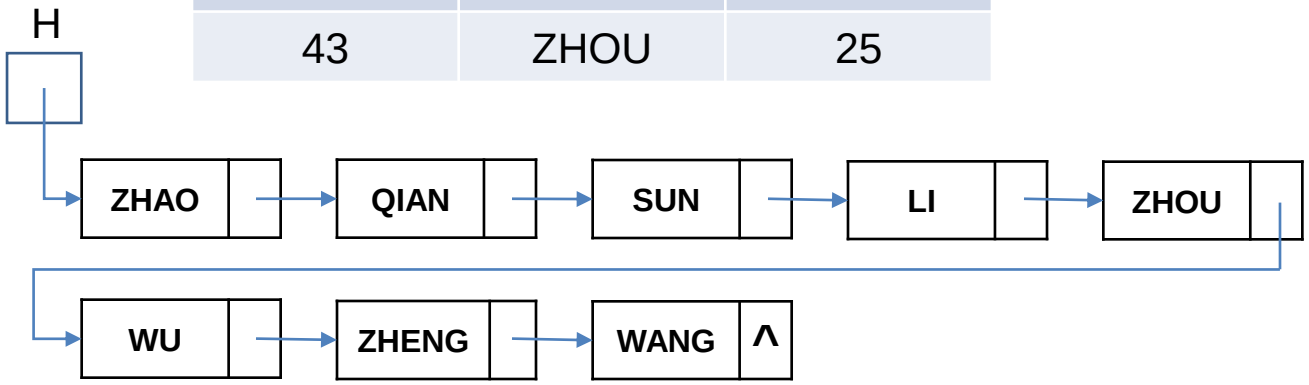
具体实现方法：定义一个结构型数组（每个元素都含有数据域和指示域），就可以完全描述链表，指示域就相当于动态链表中的指针。



2.3.5 静态链表

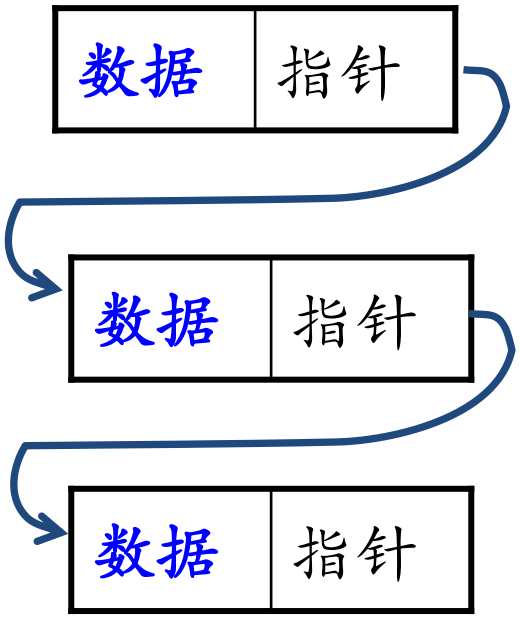
例： (ZHAO, QIAN, SUN, LI, ZHOU, WU, ZHENG, WANG)

存储地址	数据域	指针域
1	LI	43
7	QIAN	13
13	SUN	1
19	WANG	NULL
25	WU	37
31	ZHAO	7
37	ZHENG	19
43	ZHOU	25



2.3.5 静态链表

动态链表样式：



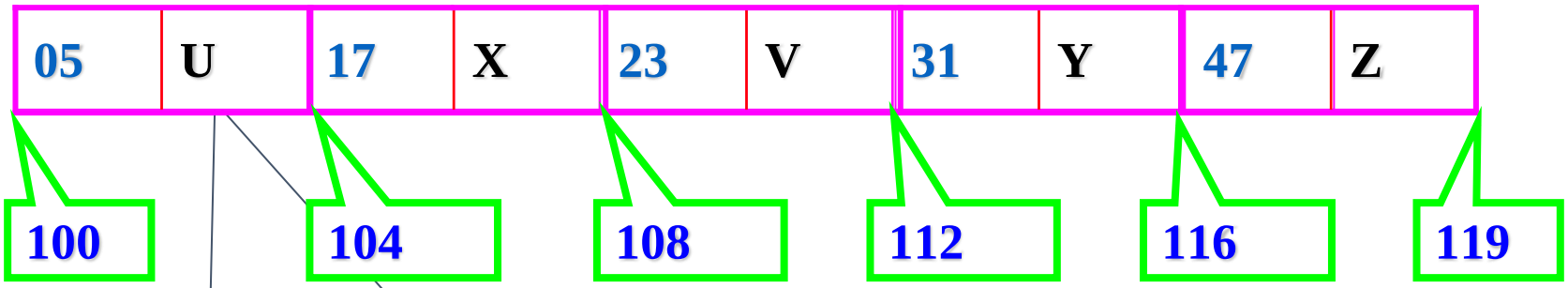
静态链表样式：

数据	指示
数据	指示
数据	指示
数据	指示
数据	指示

数组中每个元素都至少有两个分量，属于结构型数组。

常用于无指针类型的高级语言中。

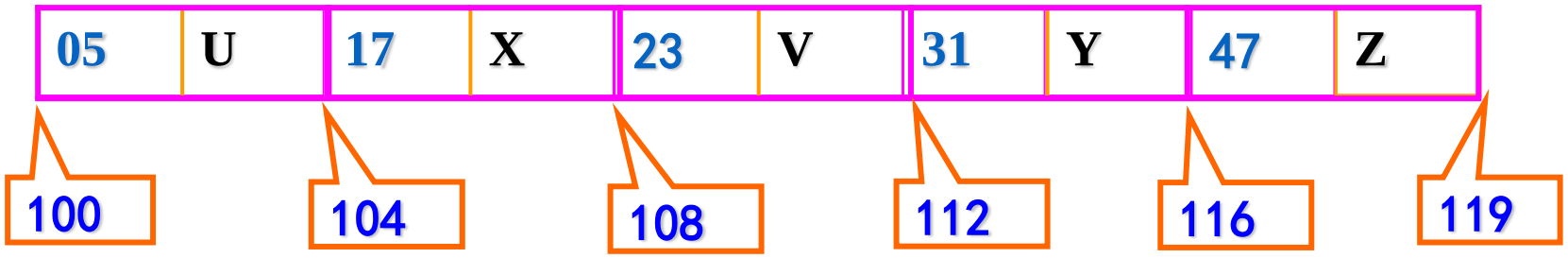
2.3.5 静态链表



若此分量定义为指针型，属于动态链表；
若此分量定义为整型（通常存放下标），则属于静态链表。

2.3.5 静态链表

现有一个具有五个元素的线性表L={23，17，47，05，31}，若它以下列方式存储在下列100～119号地址空间中，每个结点由数据（占2个字节）和指针（占2个字节）组成。



其中指针X，Y，Z的值分别为多少？该线性表的首结点起始地址为多少？末结点的起始地址为多少？

答：X= 116 Y= NULL Z= 100，

首址= 108 末址= 112。

2.3.5 静态链表

静态单链表的类型定义如下:

```
#define MAXSIZE   1000
   //预分配最大的元素个数（连续空间）
   typedef   struct  {
          ElemType   data ;    //数据域
          int            cur ;     //指示域
   }component , SLinkList[MAXSIZE] ;
//这是一维结构型数组
```

2.3.5 静态链表

前例：一线性表 S = (ZHAO, QIAN, SUN, LI, ZHOU, WU)，用静态链表如何表示?

i	data	cur
0	头结点	1
1	ZHAO	3
2	LI	5
3	QIAN	6
4	WU	0
5	ZHOU	4
6	SUN	2
…	……	……
1000		

说明1：假设S为SLinkList型变量，则

S[MAXSIZE] 为一个静态链表；

S[0].cur则表示第1个结点在数组中的位置。

说明2：如果数组的第i个分量表示链表的第k个结点，则：

S[i].data表示第k个结点的数据；

S[i].cur 表示第k+1个结点(即k的直接后继)的位置。

2.3.5 静态链表

- **初始化静态链表**

```
int InitSpace_ SL(SLinkList &space) {
    for(i = O; i<MAXSIZE-1; ++i)
        space[i].cur = i+l;
    space[MAXSIZE- 1]. cur = 0;
}
```

- **静态链表查找**

```
int LocateElem_ SL(SLink:List S, ElemType e) {
    i = S[0].cur;
    while (i && S[i].data != e)
        i = S[i].cur;
    return i;
}
```

2.3.5 静态链表

例如：在线性表 S = (ZHAO, QIAN, SUN, LI, ZHOU, WU)的QIAN, SUN之间插入新元素LIU，可以这样实现：

i	data	cur
0	头结点	1
1	ZHAO	3
2	LI	5
3	QIAN	7
4	WU	0
5	ZHOU	4
6	SUN	2
7	LIU	6
1000	…	…

Step1:将QIAN的游标值存入LIU的游标中：

S[7].cur = S[3].cur;

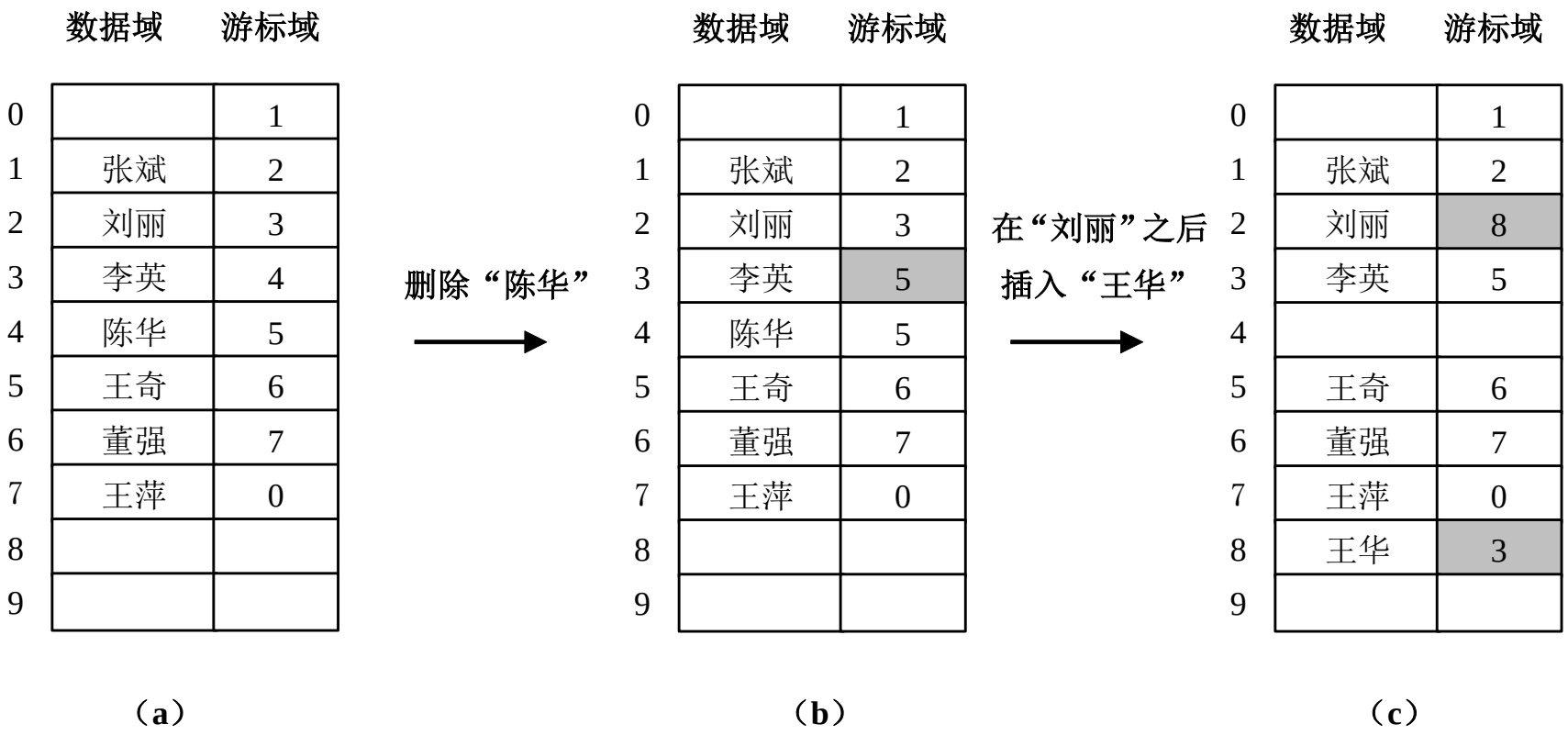
Step2:将QIAN的游标换为新元素LIU的下标：

S[3].cur = 7

说明3：静态链表的插入与删除操作与普通链表一样，不需要移动元素，只需修改指示器就可以了。

2.3.5 静态链表

下图给出了一个静态链表的示例。图(a)是一个修改之前的静态链表，图(b)是删除数据元素“陈华”之后的静态链表，图(c)插入数据元素“王华”之后的静态链表，图中用阴影表示修改的游标。



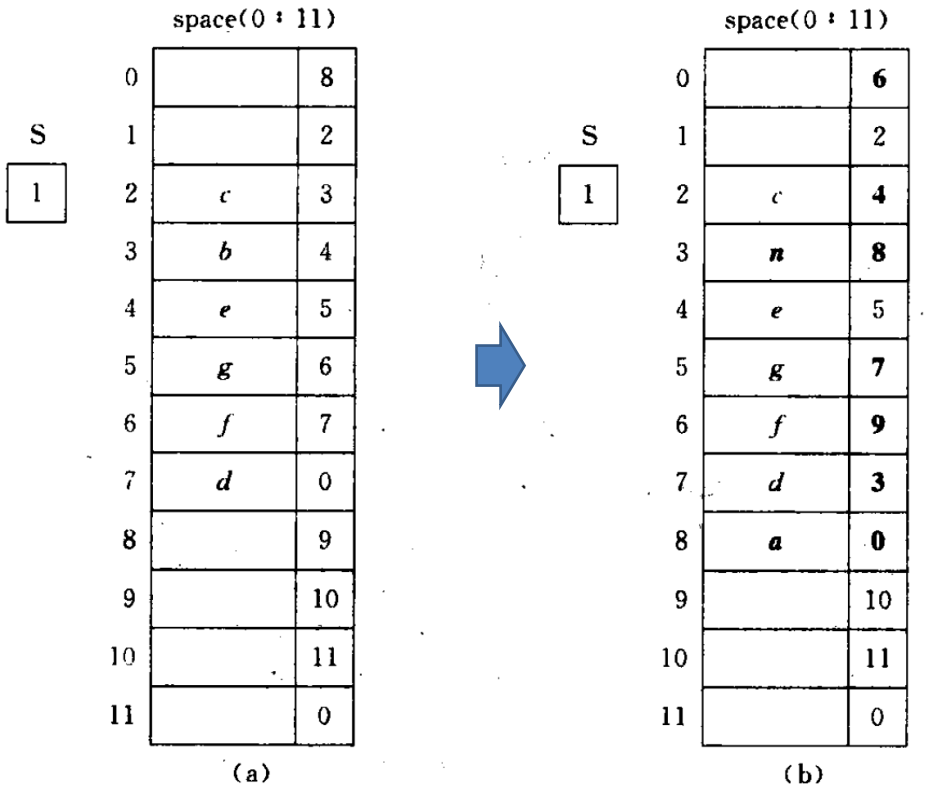
2.3.5 静态链表

例：集合运算(A-B) U (B-A)

假设由终端输入集合元素，先建立表示集合A的静态链表S, 而后在输入集合B的元素的同时查找S表，若存在和B相同的元素，则从S表中删除之，否则将此元素插人S表。

A= (c, b, e, g, f, d)

B= (a, b, n, f)



2.3.5 静态链表

- **初始化静态链表**

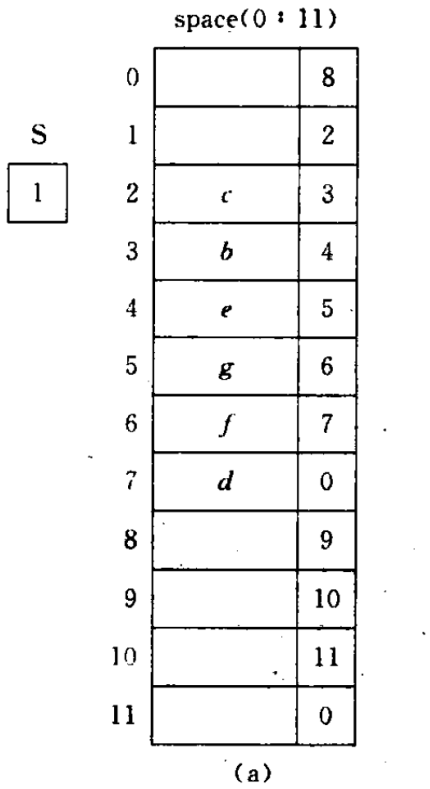
```
void InitSpace_ SL(SLinkList &space){
    for(i = 0; i<MAXSIZE-1; ++i)     space[i].cur = i+l;
    space[MAXSIZE- 1]. cur = 0;
}
```

- **分配存储区域**

```
int Malloc_ SL(SLinkList &space){
    i = space[0].cur;
    if (space[0].cur)     space[0].cur = space[i].cur;
    return i;
}
```

- **释放存储区域**

```
int Free_ SL(SLinkList &space, int k){
    space[k].cur = space[0].cur;     space[0].cur = k;
}
```



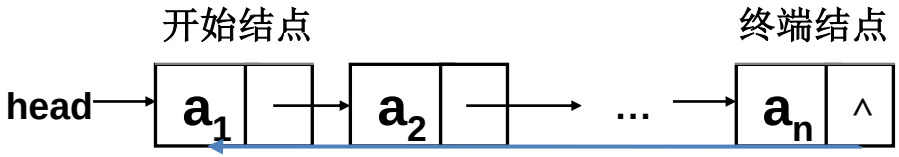
2.3.5 静态链表

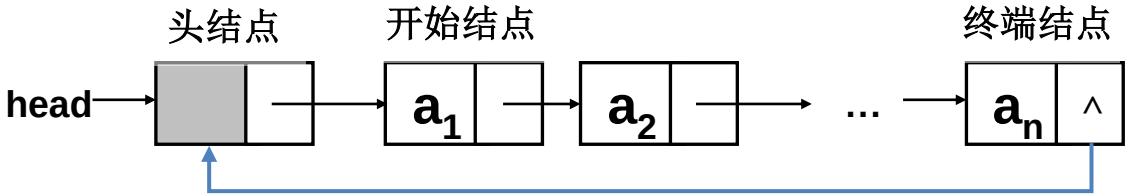
```
void difference(SLinkList &space, int &S){
    InitSpace_SL(space); S = Malloc_SL(space); r = S;
    scanf(m, n);
    for (j = 1; j <= m; ++ j) {
        i = Malloc_SL(space); scanf(space[i].data); space[r].cur= i; r = i;
     }
     space[r].cur = 0;
     for(j = 1; j <= n; ++ j) {
        scanf(b); p = S; k = space[S].cur;
        while (k != space[r].cur && space[k].data != b){
            p = k; k = space[k].cur;
        }
        if (k == space[r].cur) {
            i = Malloc_SL(space);  space[i].data = b; space[i].cur = space[r].cur;
            space[r].cur = i;
        }
        else{ space[p].cur = space[k].cur; Free_SL(space, k); if (r == k) r = p;}
    }
}
```

2.3 线性表的链式表示和实现

- 2.3.1 线性表的链式存储—链表
- 2.3.2 单链表基本运算的实现
- 2.3.5 静态链表
- 2.3.3 循环链表
- 2.3.4 双链表

2.3.3 循环链表





特点

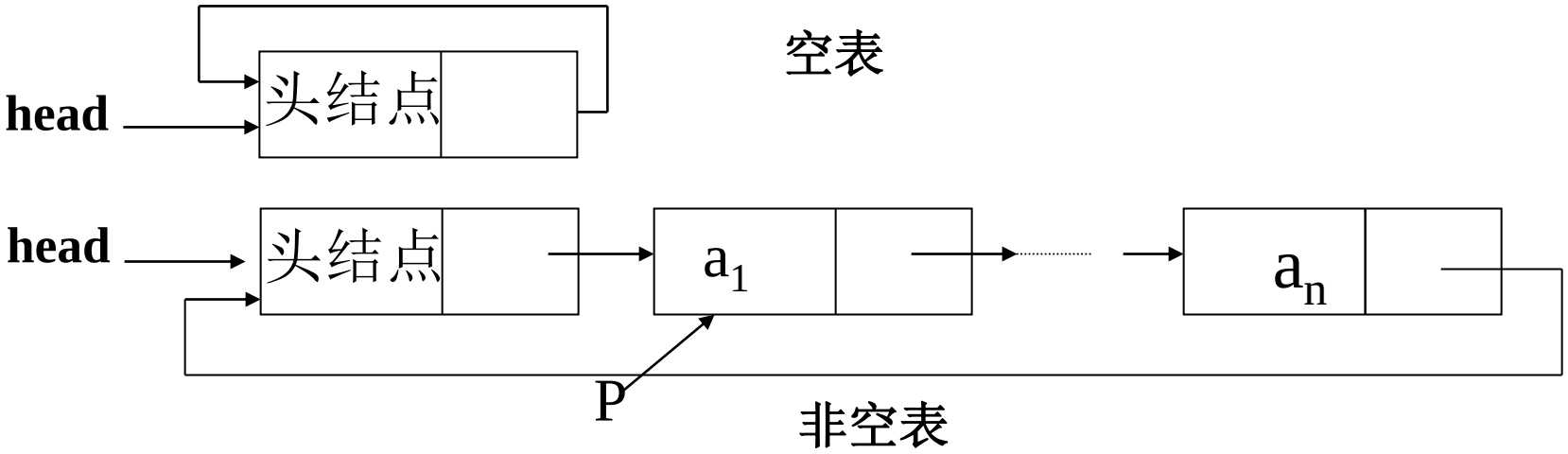
(1) 尾结点指针域指向头结点

(2) 结点结构与线性链表相同

(3) 有关操作与线性链表基本相同，仅有判断链表尾方法不同

(4) 知道任何一个指向某一结点的指针就可以访问链表中的所有结点

2.3.3 循环链表



(1) 判断链表空吗?

p->next==p，则链表为空或者只有一个元素

(2) 判断链表遍历完否?

p->next==head，则遍历完毕

(3) 插入或删除结点

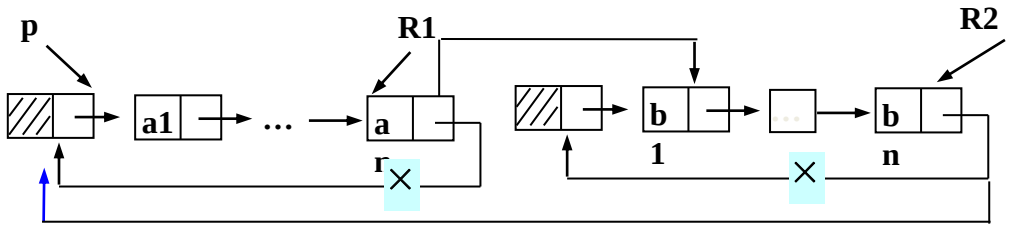
与一般的单向链表相同

2.3.3 循环链表

(3) 合并两个链表

单链表：

```
p=La;
while (p->next)  p=p->next;
p->next = Lb->next;
free(Lb);
```



循环链表：

```
p=La;
while (p->next!=La)  p=p->next;
p->next=Lb->next;
p=p->next;
while (p->next!=Lb) p=p->next;
p->next=La;
free(Lb);
```

2.3.3 循环链表

(4) 建立循环链表

```
void CreateList_L(LinkList &L) {
   Lnode* p;  int x;
   L= (LinkList)malloc(sizeof(LNode));
   L->next=L;
   while (scanf("%d,",&x), x!=0 ) {
      p=(Lnode*)malloc(sizeof(LNode));
      p->data=x; p->next = L->next;
      L->next = p;
   }
}
```

2.3.3 循环链表

(5) 显示输出循环链表

```
void PrintList_LC(LinkList L)
 { LinkList p;
    p=L->next;  printf("L->");
    while (p!=L) {
         printf("%d->",p->data);
         p=p->next;  }
   printf("L\n");
 }
```

2.3.3 循环链表

例 将一循环单链表就地逆置。

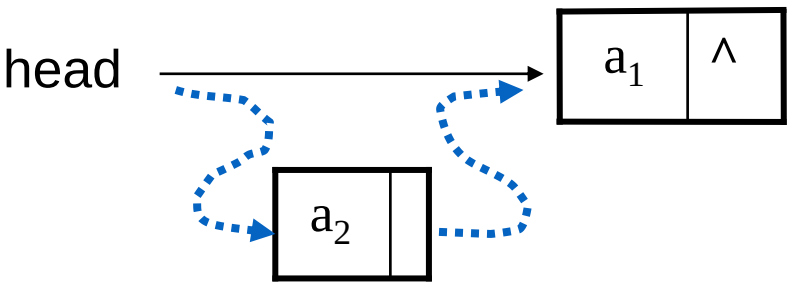
分析：要想让a_n指向a_{n-1},……a_2指向a_1，一般有两种算法：

①替换法：扫描a_1……a_n，将每个a_i的指针域指向a_{i-1}。

思路：后继变前驱

②插入法：扫描a_1……a_n，将每个a_i插入到链表首部即可。

思路：头部变尾部



实际上是链栈的概念

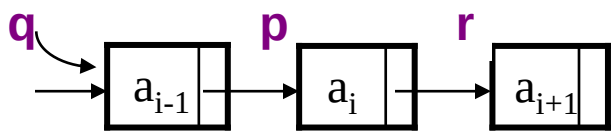
2.3.3 循环链表

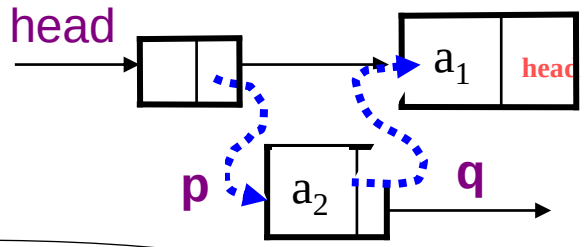
替换法的核心语句：

```
q=head;//逆置后的第一个结点
p=head->next; //有头结点
while(p!=head) //循环单链表
{  r=p->next;
   p->next=q;  //前驱变后继
   q=p;
   p=r; }  //准备处理下一结点
head->next=q; // 以an为首
```

插入法的核心语句：

```
p=head->next; //有头结点
if(p!=head){r=p->next;
  p->next =head;p=r}; //处理a1
while(p!=head) //循环单链表
{ r=p ->next    //保存原后继
   p ->next= head->next;
   head->next=p;
  p=r;} //准备处理下一结点
```





请上机验证并分析效率！

2.3 线性表的链式表示和实现

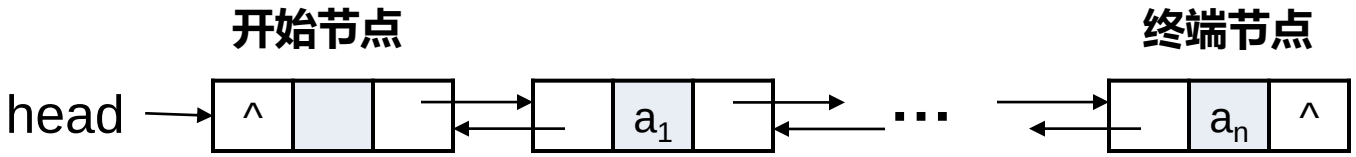
- 2.3.1 线性表的链式存储—链表
- 2.3.2 单链表基本运算的实现
- 2.3.5 静态链表
- 2.3.3 循环链表
- 2.3.4 双链表

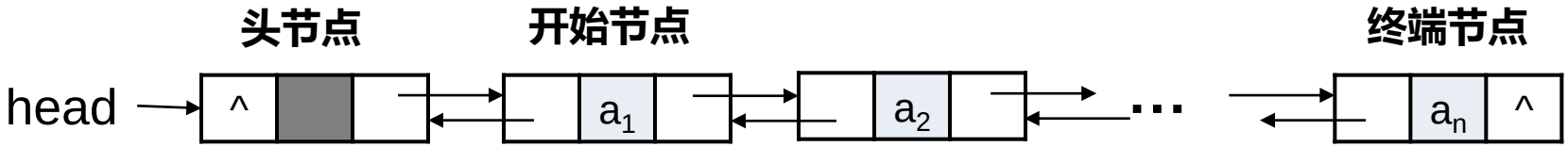
2.3.4 双链表

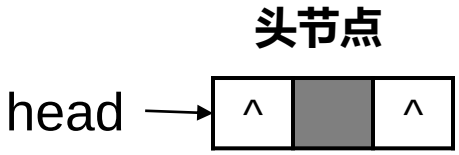
双向链表(Double linked list):在单链表的每个结点里再增加一个指向其直接前趋的指针域prior。这样就形成的链表中有两个方向不同的链，故称为双向链表。


```
typedef struct DNode   {  /*定义双链表结点类型*/
    ElemType data;
    struct DNode *prior;    /*指向前驱结点*/
    struct DNode *next;     /*指向后继结点*/
} DLinkList;
```

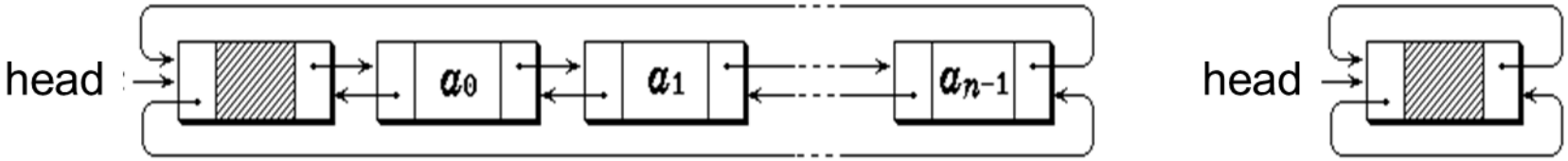
2.3.4 双链表







2.3.4 双链表



(b) 非空双向循环链表　　(c) 空表

- 对双向循环链表中任一结点的指针，有：

  p == p→prior→next == p→next→prior

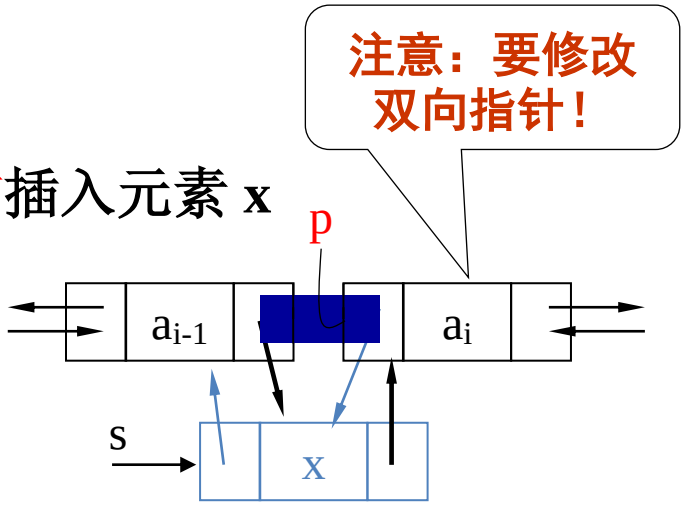
- 置空表：

  p→prior = p ; p→next = p;

2.3.4 双链表

（1）双向链表插入

设p已指向第 i 元素，请在第 i 元素前插入元素 x



指针域的变化：

① a_{i-1}的后继从 a_i (指针是p)变为 x（指针是s) :

s->next = p ； p->prior->next = s ;

② a_i 的前驱从 a_{i-1} (指针是p->prior)变为 x (指针是s);

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

2.3.4 双链表

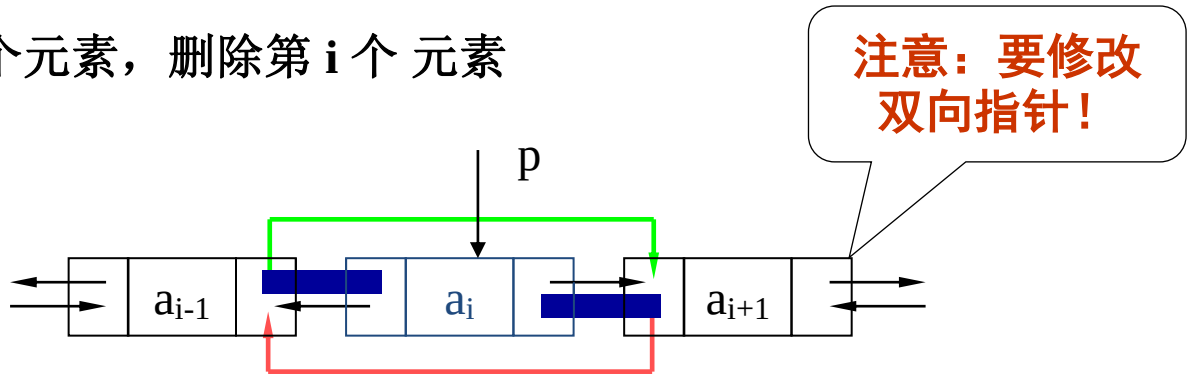
（1）双向链表插入

```
void ListInsert(DuLNode* p, ElemType x)
{
    //申请新节点
    DuLNode* s = (DuLNode*)malloc(sizeof(DuLNode));
    //1·将新节点的next改为当前节点
    s->next = p;
    //2·将当前节点的上一节点的next改为新节点
    p->prior->next = s;
    //3·将新节点的prior改为当前节点的上一节点
    s->prior = p->prior;
    //4·将当前节点的prior改为新节点
    p->prior = s;
}
```

2.3.4 双链表

（2）双向链表删除

设p指向第 i 个元素，删除第 i 个 元素



指针域的变化：

后继方向：a_{i-1}的后继由 a_i (指针p)变为 a_{i+1}(指针 p ->next);

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

前驱方向：a_{i+1} 的前驱由 a_i (指针p)变为a_{i-1} (指针 p -> prior);

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

2.3.4 双链表

(2) 双向链表删除

```
void ListDelete(DuLNode* p)
{
    //禁止删除头节点
    assert(p->next != p);
    //1·当前位置的上一节点的next变为当前节点的下一节点
    p->prior->next = p->next;
    //2·当前位置的下一节点的prior变为当前节点的上一节点
    p->next->prior = p->prior;
    //释放当前位置的节点
    free(p);
    p = NULL;
}
```

2.3.4 双链表

删除第一个数据域为x的节点

```
void ListDeleteByData(DuLNode* L, ElemType x){
    DuLNode* p,q,r;
    p = L;
    while(p->next && p->next->data != x){
        p = p->next;
    }
    if(!(p->next)){
        printf("The char '%c' does't exist.\n", x);
        return ;
    }
    q = p->next;
    r = q->next;
    p->next = r;
    if(r){
        r->prior = p;
    }
    free(q);
}
```

正在答疑