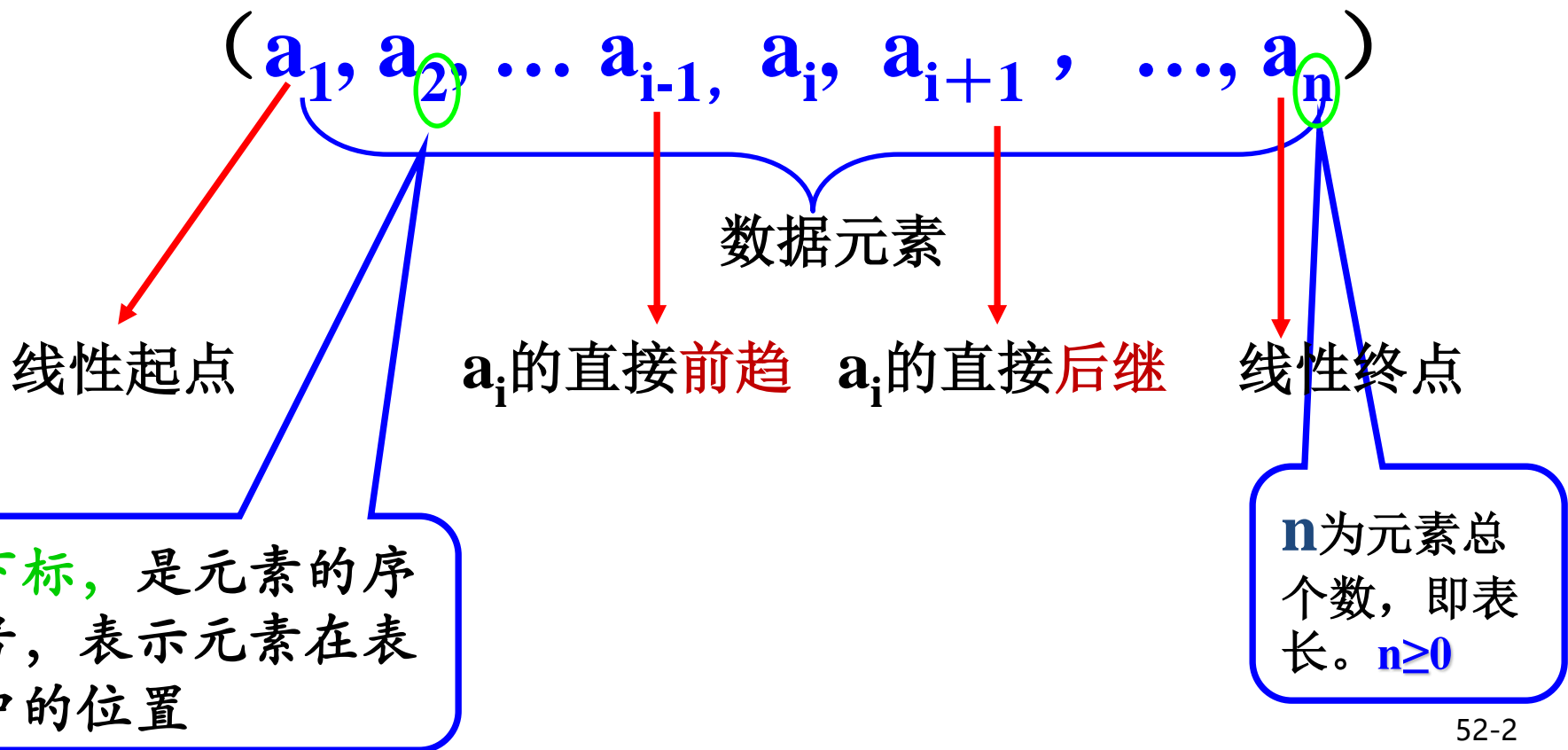


第二章 线性表

回顾

线性表逻辑结构

线性表的定义： 用数据元素的有限序列表示



回顾

顺序存储：用一组地址连续的存储单元依次存储线性表的元素。

```
#define MaxSize 100

typedef struct {
    ElemType data[MaxSize];
    int length;
} SqList;
```

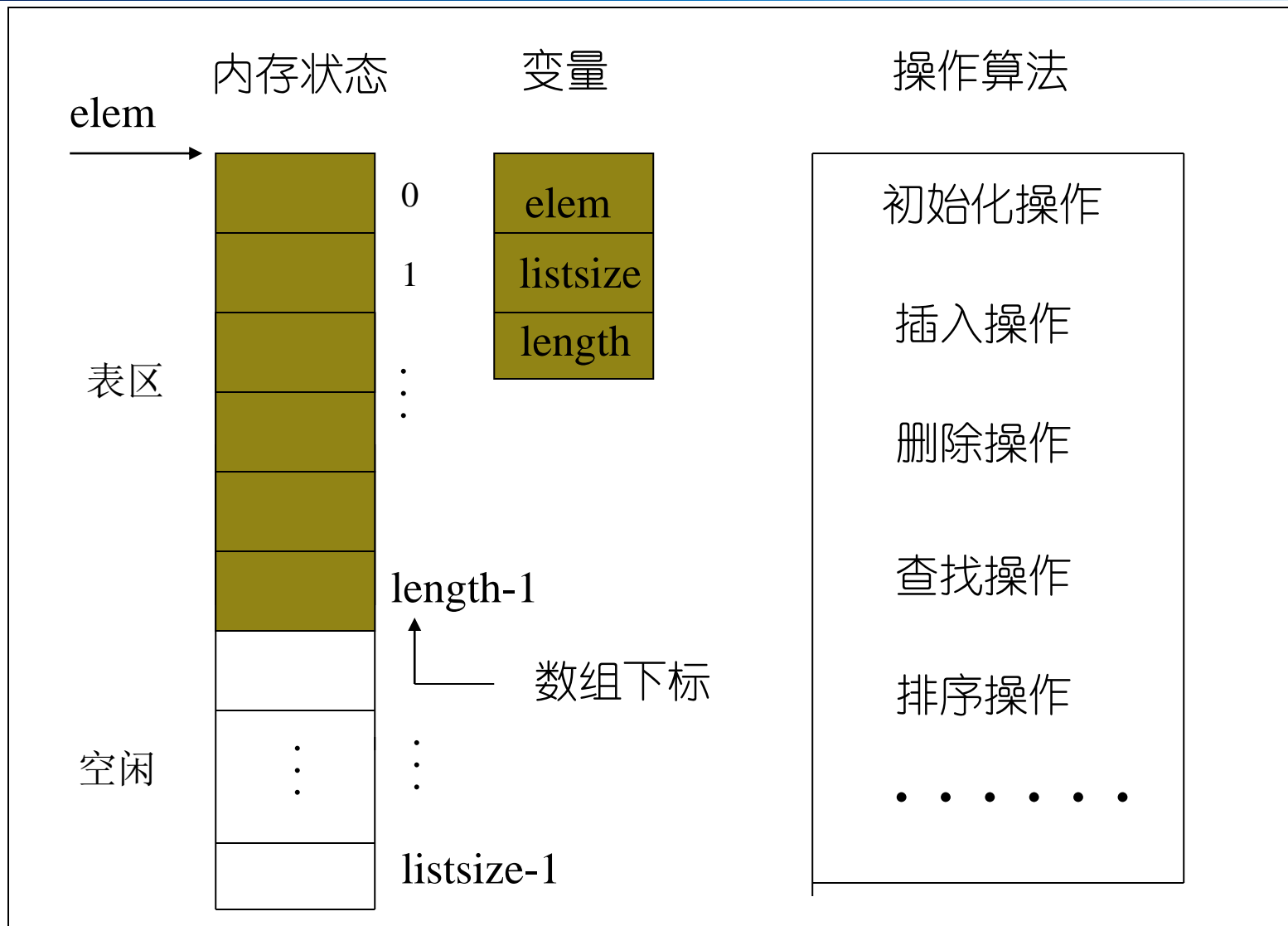
```
#define LIST_INIT_LENGTH 100
#define LISTINCREMENT 10

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

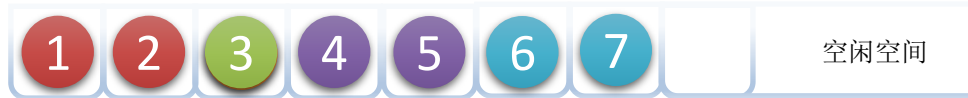
```
void *malloc(unsigned int size)
```

```
void *realloc(void *p, unsigned int size)
```

回顾



回顾



```
q=&(L.elem[i-1]);  
for (p=&(L.elem[L.length-1]);p>=q;--p)  
    *(p+1) = *p;  
*q=e;  
++L.length;
```



```
e=*p;  
q=L.elem+L.length-1;  
for (++p;p<=q;++p)  
    *(p-1)=*p;  
--L.length;
```

回顾

1) 优点

- 顺序表的结构简单
- 顺序表的存储效率高，是紧凑结构
- 顺序表是一个随机存储结构（直接存取结构）

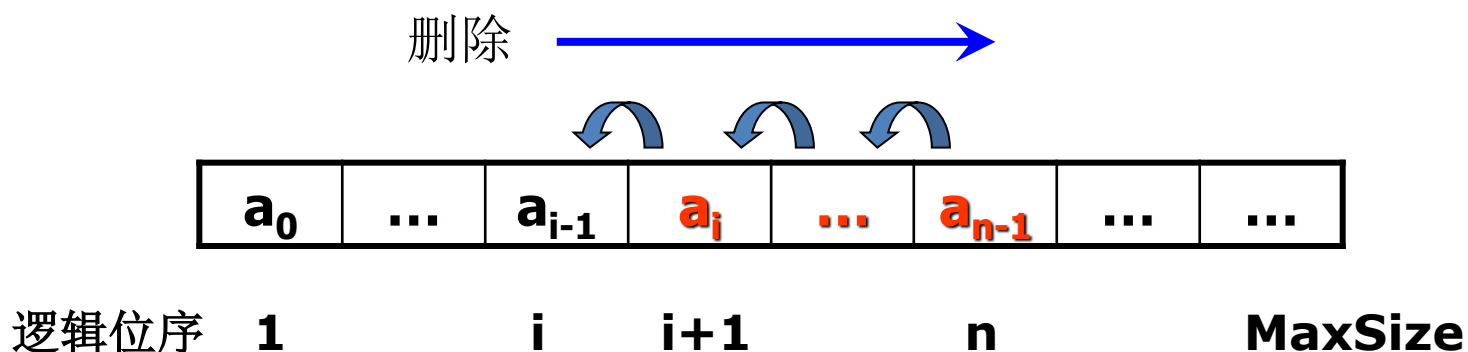
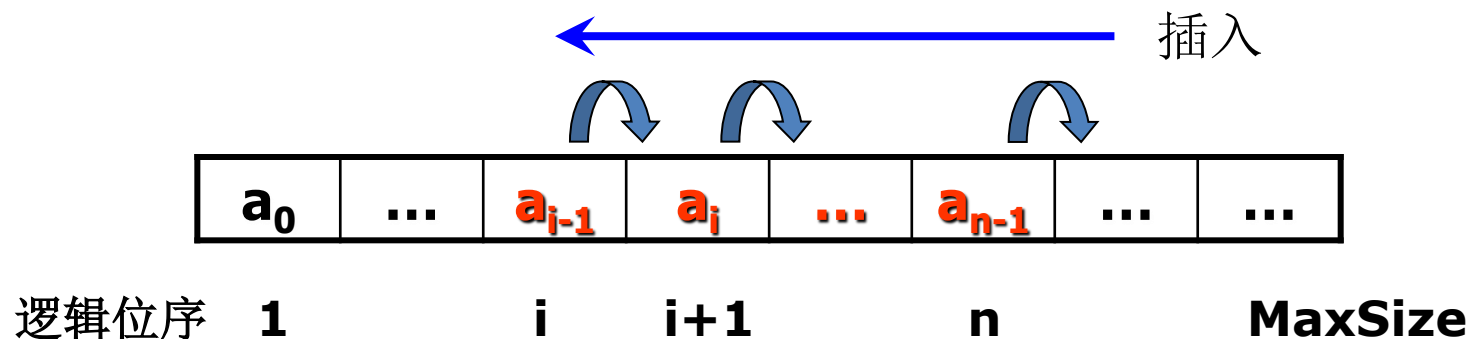
2) 缺点

- 在顺序表中进行插入和删除操作时，需要移动数据元素，算法效率较低。
- 对长度变化较大的线性表，或者要预先分配较大空间或者要经常扩充线性表，给操作带来不方便。
- 原因：数组的静态特性造成

2.3 线性表的链式表示和实现

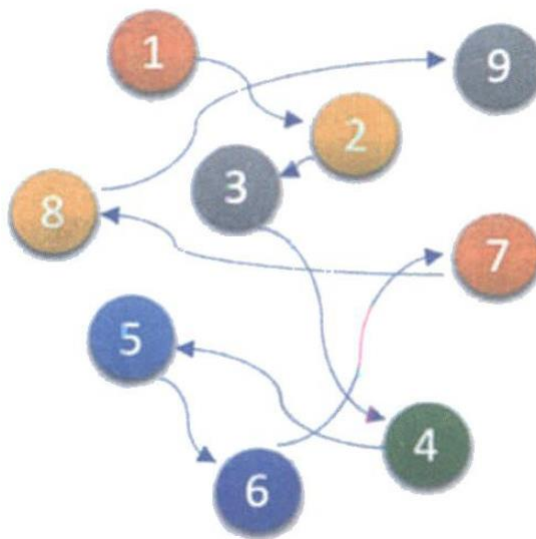
- **2.3.1 线性表的链式存储—链表**
- **2.3.2 单链表基本运算的实现**
- **2.3.3 静态链表**
- **2.3.4 循环链表**
- **2.3.5 双链表**

2.3.1 线性表的链式存储—链表



2.3.1 线性表的链式存储—链表

- **特点：**在内存中用一组任意的存储单元来存储线性表的数据元素，用每个数据元素所带的指针来确定其后继元素的存储位置。这两部分信息组成数据元素的存储映像，称作**结点**。



2.3.1 线性表的链式存储—链表

在每个结点中除包含有数据域外,只设置一个指针域,用以指向其后继结点,这样构成的链接表称为线性单向链接表,简称**单链表**;

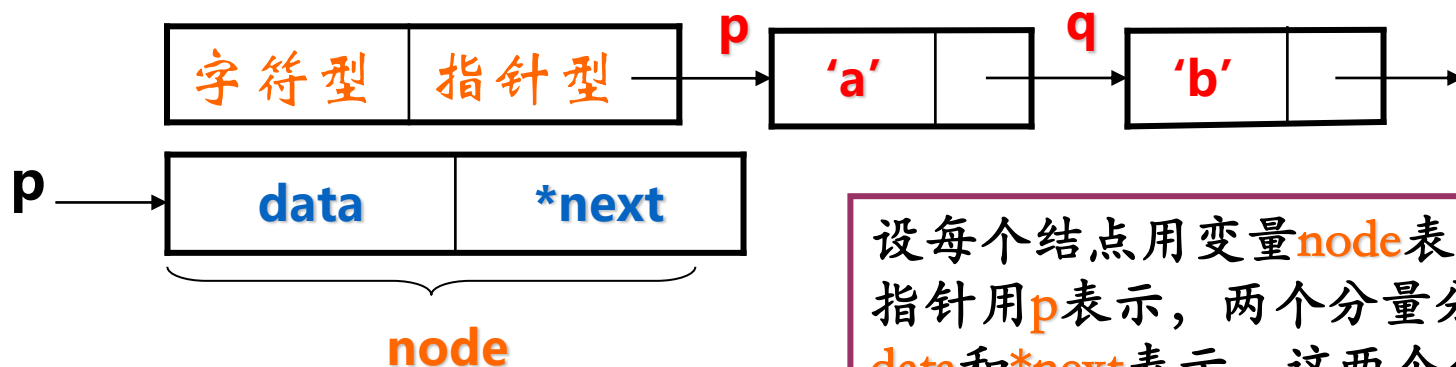


●**结点**：数据域 + 指针域（链域）

- 链式存储结构：n个结点链接成一个链表
- 线性链表（单链表）：链表的每个结点只包含一个指针域

2.3.1 线性表的链式存储—链表

以26个字母的链表为例，每个结点都有两个分量：

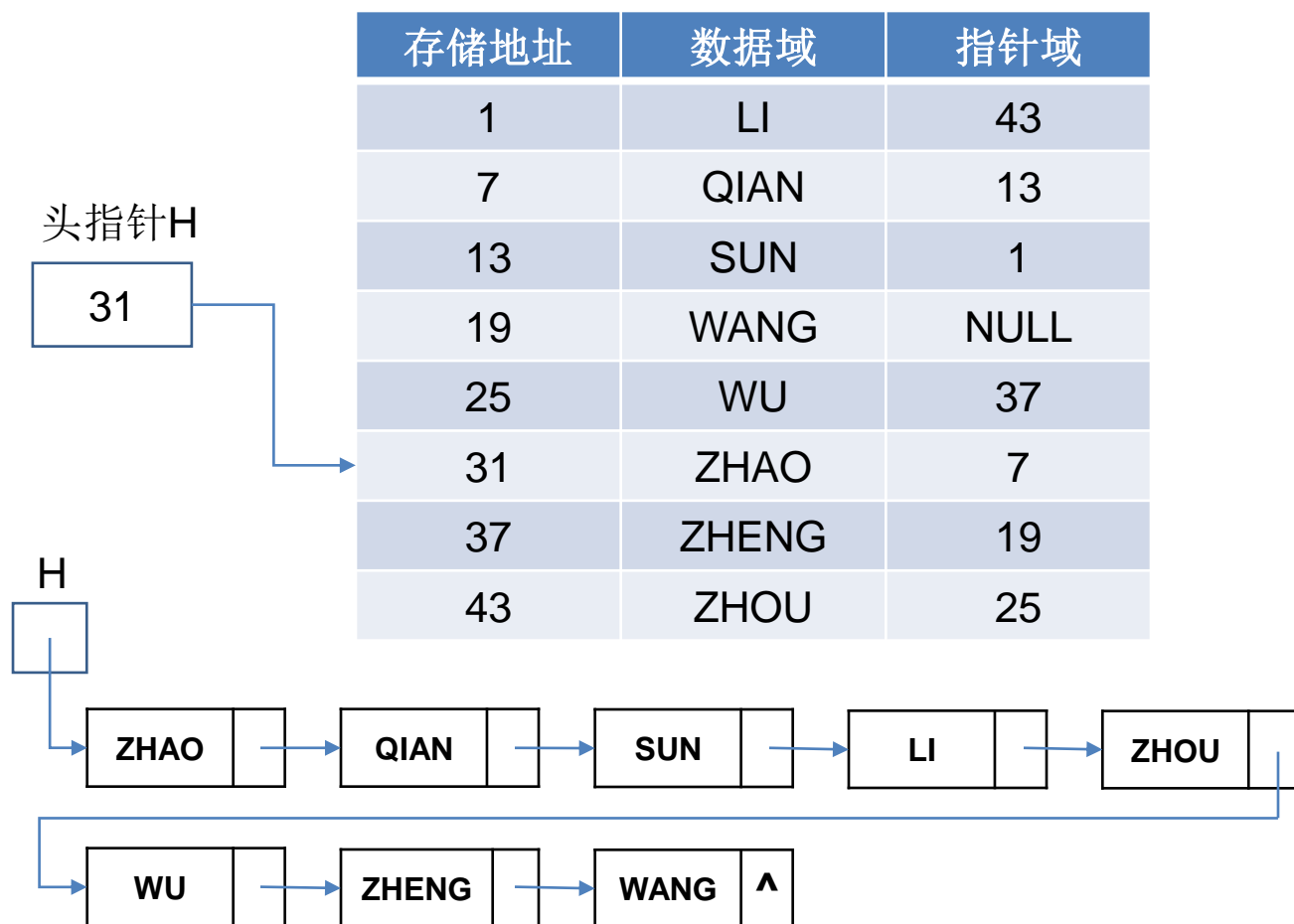


设每个结点用变量`node`表示，其指针用`p`表示，两个分量分别用`data`和`*next`表示，这两个分量如何赋值？

- 方式1： 直接表示为 `node.data = 'a'; node.next = q`
- 方式2： `p`指向结点首地址，然后 `p->data = 'a'; p->next = q;`
- 方式3： `p`指向结点首地址，然后 `(*p).data = 'a'; (*p).next = q`


2.3.1 线性表的链式存储—链表

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



2.3.1 线性表的链式存储—链表

设 p 为指向链表的第 i 个元素的指针,则第 i 个元素的数据域写为 $p \rightarrow data$, 指针域写为 $p \rightarrow next$ 。



a_i 的值 a_{i+1} 的地址

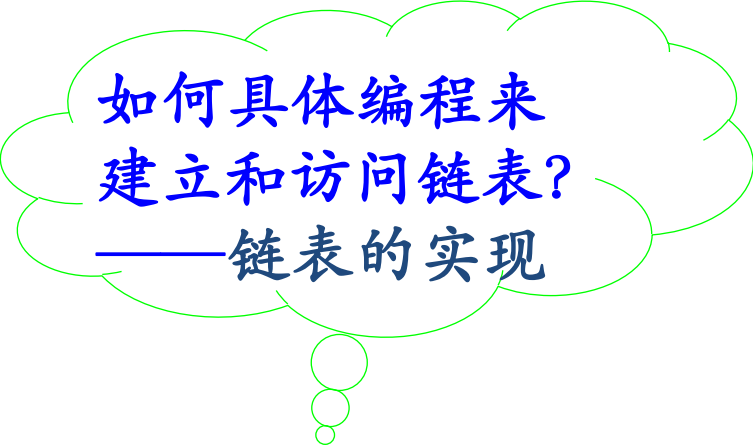
2.3.1 线性表的链式存储—链表

用结构类型和指针来表示顺序结构

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

Q1: 第一行的**Lnode** 与最后一行的**Lnode**是不是一回事?

A1: 不是。前者**Lnode**是结构名，后者**Lnode**是对整个struct类型的一种“缩写”，是一种“**新定义名**”，它只是对现有类型名的补充，而不是取代。



如何具体编程来
建立和访问链表?
——链表的实现

2.3.1 线性表的链式存储—链表

在线性表的链式存储中,为了便于插入和删除算法的实现,每个链表带有一个头结点,并通过头结点的指针惟一标识该链表。



带头结点单链表示意图

- 说明：头结点的next域指向链表中的第一个数据元素结点。
- 对于头结点数据域的处理：
 - a.加特殊信息
 - b.置空
 - c.如数据域为整型，则在该处存放链表长度信息

2.3.1 线性表的链式存储—链表

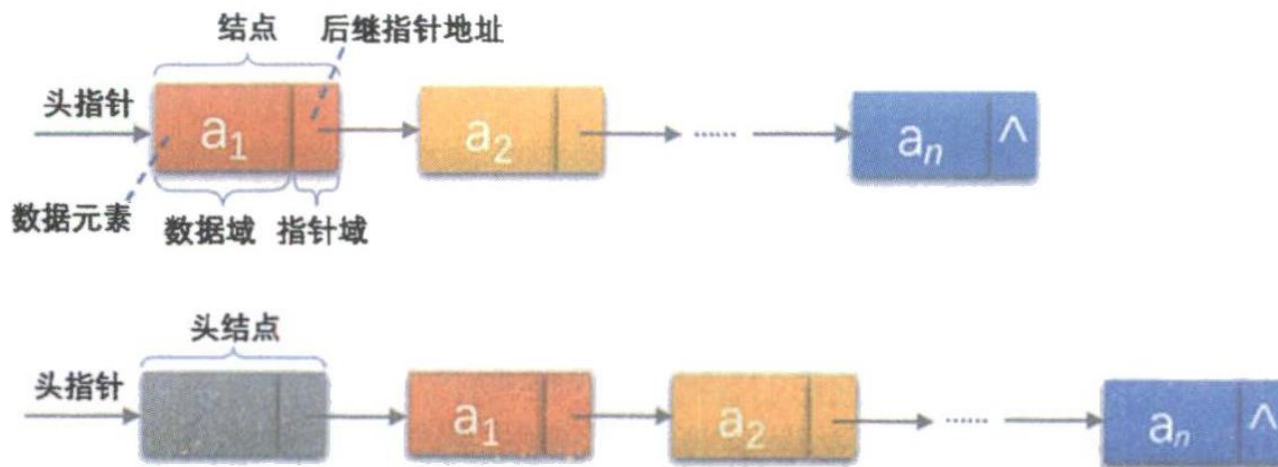
头指针

- 头指针是指链表指向第一个结点的指针，若链表有头结点，则是指向头结点的指针
- 头指针具有标志作用，所以常用头指针冠以链表的名字
- 无论链表是否为空，头指针均不为空。头指针是链表的必要元素

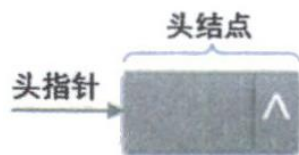
头结点

- 头结点是为了操作的统一和方便而设立的，放在第一元素的结点之前，其数据域一般无意义（也可存放链表的长度）
- 有了头结点，对在第一元素结点前插入结点和删除第一结点，其操作与其他结点的操作就统一了
- 头结点不一定是链表必需要素

2.3.1 线性表的链式存储—链表

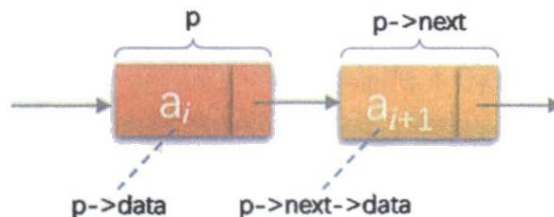


空链表:



$L \rightarrow \text{next} = \text{NIL}$

后续节点访问:



无论链表是否为空，其头指针是指向头结点的非空指针，因此对空表与非空表的处理也就统一了，简化了链表操作的实现

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

2.3.2 单链表基本运算的实现

- 取元素 p29 算法2.8
- 插入元素 p30 算法2.9
- 删除元素 p30 算法2.10
- 建立链表 p30~p31 算法2.11
- 有序链表的合并 p31 算法2.12
- 查找（按值查找）
- 求长度
- 集合的并运算

2.3.2 单链表基本运算的实现

(1)取元素

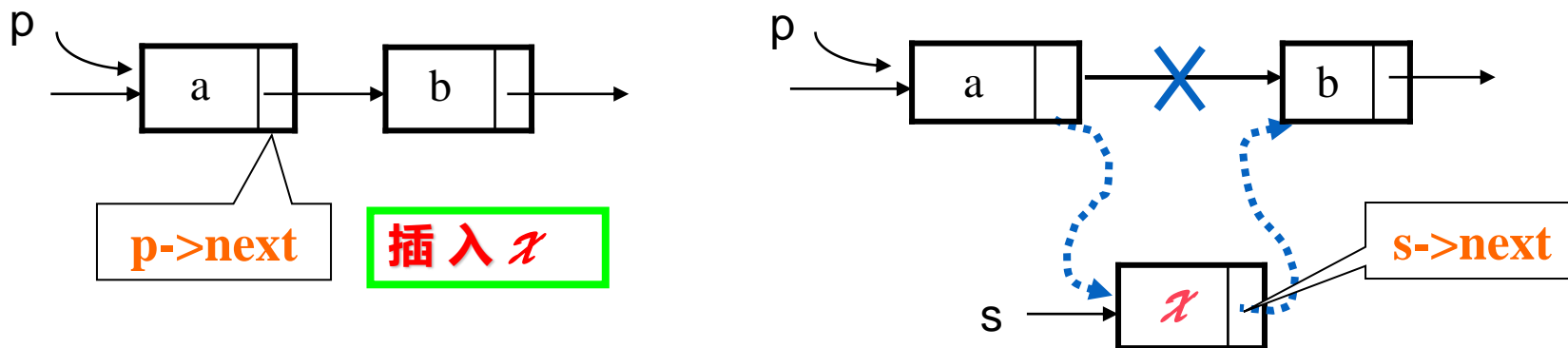
从链表的头指针出发，顺链域next逐个结点往下搜索，直至找到第i个结点为止 (j=i)

```
Status GetElem_L(LinkList L,int i,ElemType &e) {  
    LinkList p;  
    p=L->next; int j=1;  
    while (p && j<i) { p=p->next; ++j; }  
    if (!p || j>i) return ERROR;  
    e=p->data;  
    return OK;  
}
```

2.3.2 单链表基本运算的实现

(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: $s \rightarrow next = p \rightarrow next;$

Step 2: $p \rightarrow next = s;$

x 结点的生成方式：

$s = (\text{node}^*) \text{malloc} (m);$

$s \rightarrow data = x;$

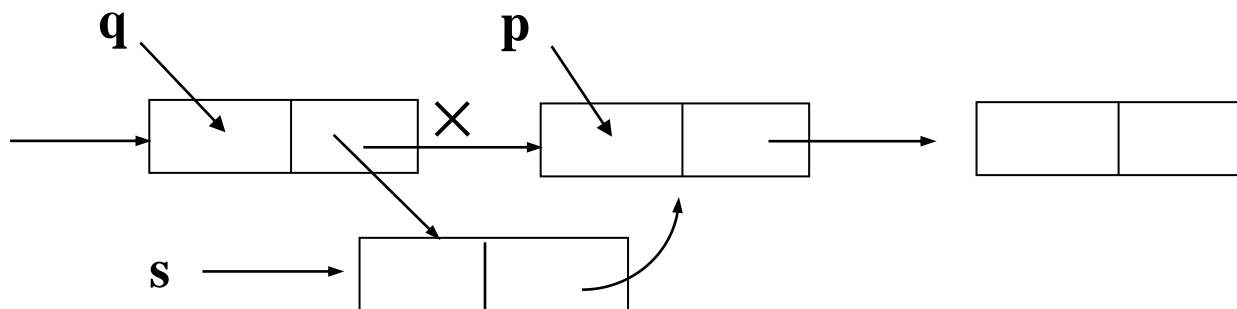
$s \rightarrow next = ?$

思考：Step1和Step2能互换么？

2.3.2 单链表基本运算的实现

前插结点：设 p 指向链表中某结点， s 指向待插入的值为 x 的新结点，将 s 插入到 p 的前面。

思路：首先要找到 p 的前驱 q ，然后再完成在 q 之后插入 s ，设单链表头指针为 L



$q = L;$

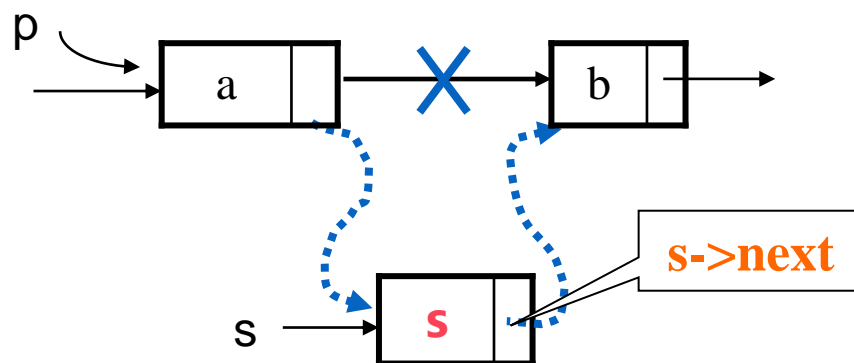
$\text{while } (q \rightarrow \text{next} \neq p) \quad q = q \rightarrow \text{next};$ /*找 p 的直接前驱*/

$s \rightarrow \text{next} = q \rightarrow \text{next};$

$q \rightarrow \text{next} = s;$

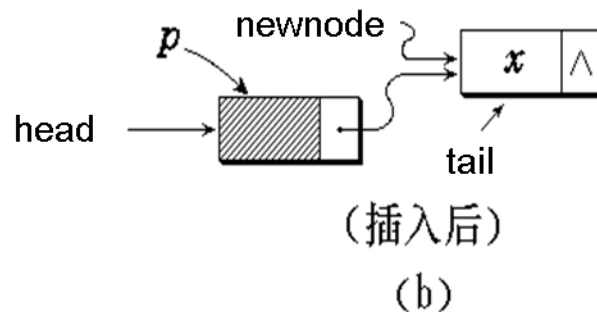
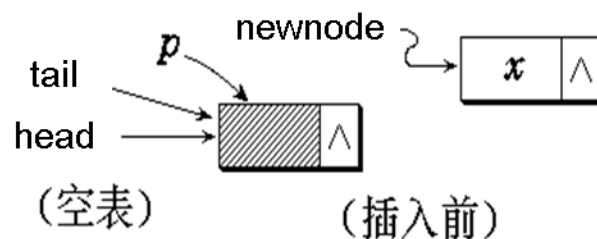
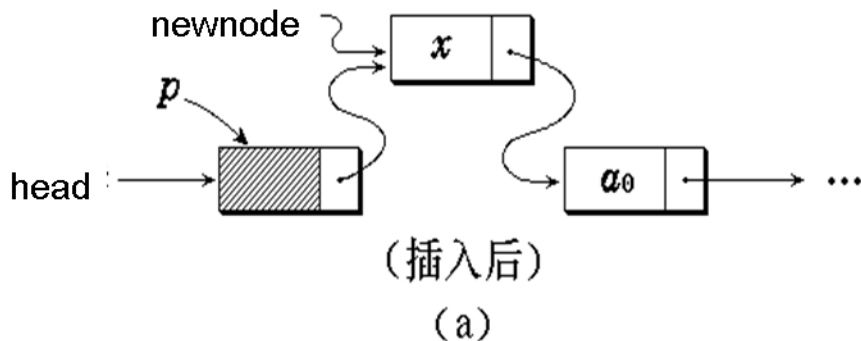
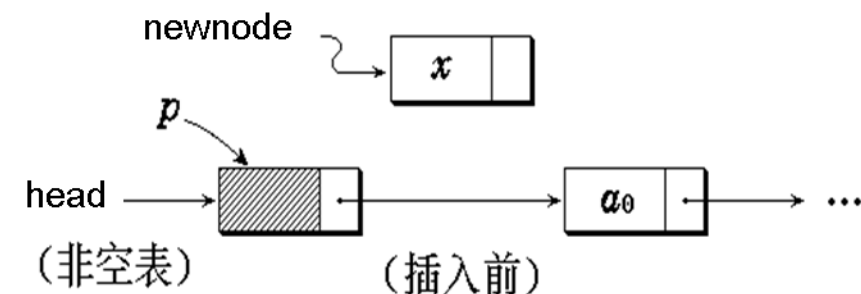
2.3.2 单链表基本运算的实现

```
Status ListInsert_L(LinkList &L, int i, ElemType e) {  
    LinkList p, s;  
    p=L;    int j=0;  
    while (p && j<i-1) { p=p->next; ++j;}  
    if (!p || j> i-1) return ERROR;  
    s = (LinkList) malloc( sizeof (LNode));  
    s->data = e;  
    s->next = p->next;  
    p->next = s;  
    return OK;  
}
```



2.3.2 单链表基本运算的实现

在带头结点的单链表第一个结点前插入新结点

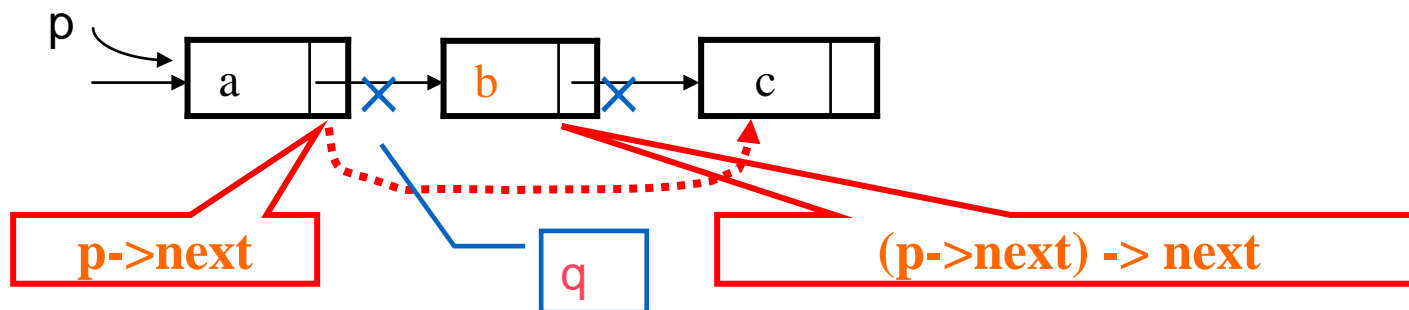


```
newnode→next = p→next;  
if ( p→next ==NULL ) tail = newnode;  
p→next = newnode;
```


2.3.2 单链表基本运算的实现

(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 ）：

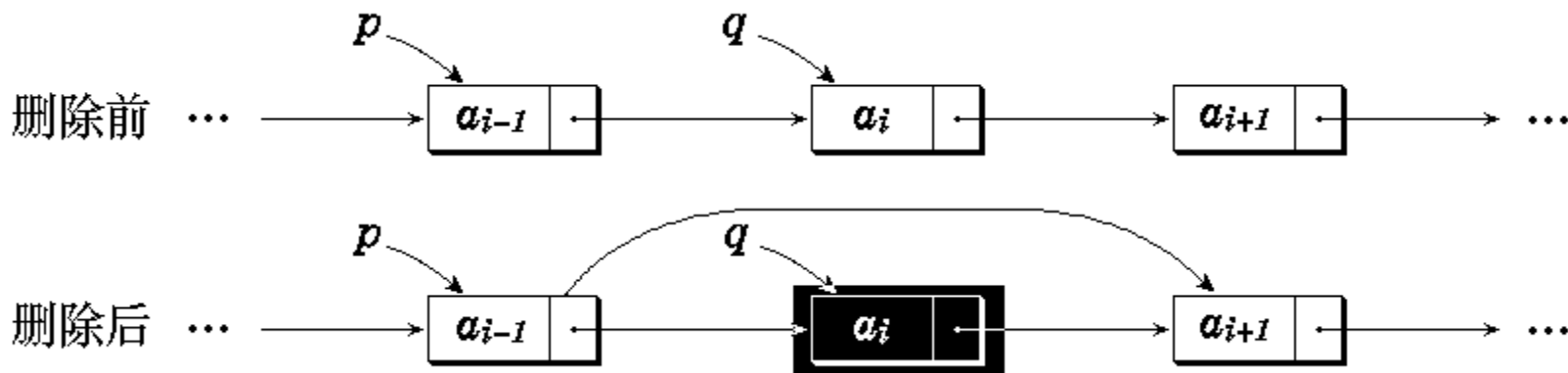
```
q = p->next;      //首先保存b的指针，靠它才能找到c；  
p->next=q->next;  //将a、c两结点相连，淘汰b结点；  
free(q);          //彻底释放b结点空间
```

思考：省略`free(q)`语句行不行？

2.3.2 单链表基本运算的实现

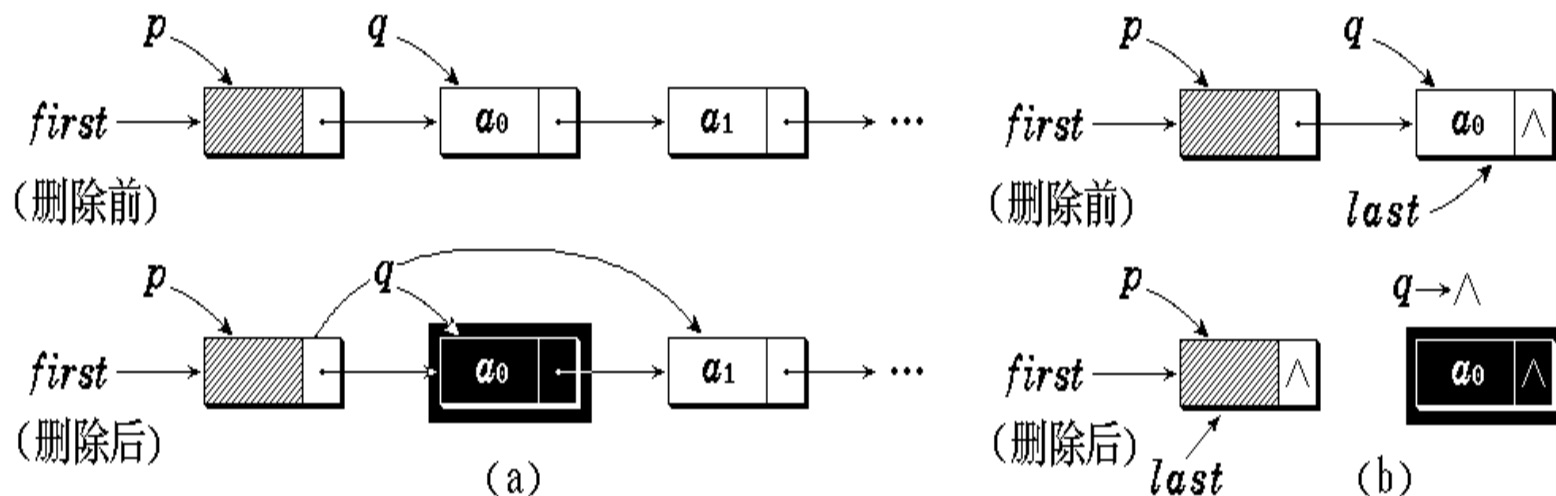
(2)删除元素

```
Status ListDelete_L(LinkList &L, int i, ElemType &e) {  
    LinkList p, q;  
    p=L;    int j=0;  
    while ( p->next && j<i-1 ) { p=p->next; ++j;}  
    if (!(p->next) || j> i-1) return ERROR;  
    q=p->next; p->next = q->next;  
    e=q->data; free(q);  
    return OK;  
}
```



2.3.2 单链表基本运算的实现

从带表头结点的单链表中删除第一个结点



```
q = p→next;  
p→next = q→next;  
delete q;  
if ( p→next == NULL ) last = p;
```

2.3.2 单链表基本运算的实现

插入和删除

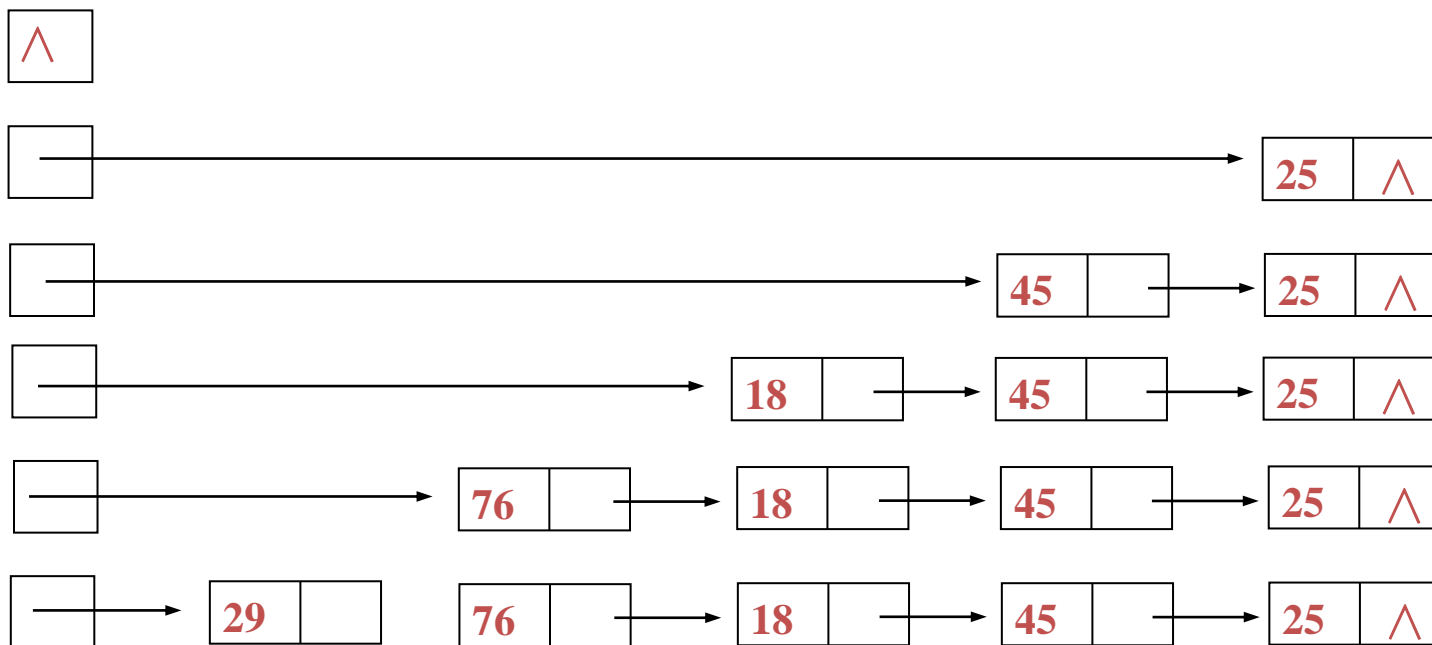
因线性链表不需要移动元素，只要修改指针，一般情况下时间复杂度为 $O(1)$ 。

如果要在单链表中进行前插或删除操作，因为要从头查找前驱结点，所耗时间复杂度将是 $O(n)$ 。

2.3.2 单链表基本运算的实现

(3)建立单链表

□ 头插法建表：读取字符数组a中的字符,生成新结点,将读取的数据放到新结点的数据域中,然后将新结点插入到当前链表的表头上,直到结束为止:



线性表：（25,45,18,76,29）之链表的建立过程

2.3.2 单链表基本运算的实现

(3)建立单链表

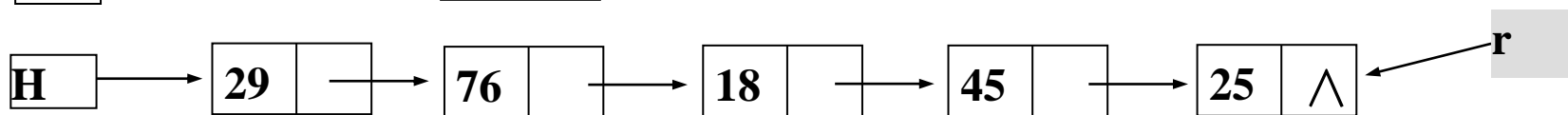
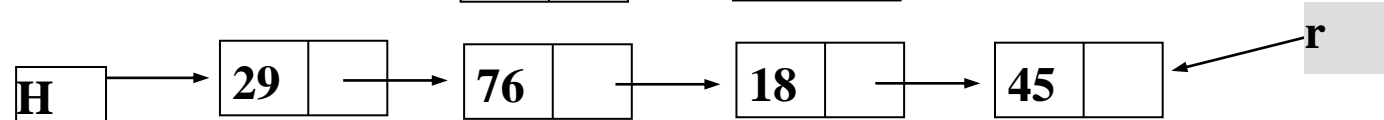
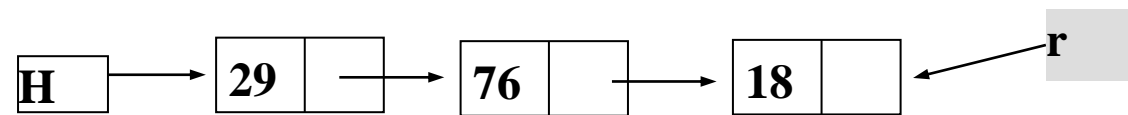
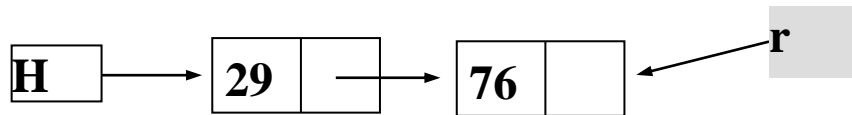
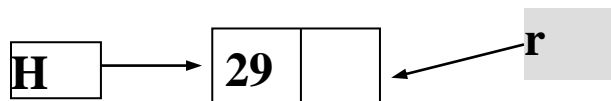
```
void CreateListL(LinkList &L,ElemType a[],int n) {  
    LNode *s;int i;  
    L=(LNode *)malloc(sizeof(LNode)); /*创建头结点*/  
    L->next=NULL;  
    for (i=0;i<n;i++){  
        s=(LNode *)malloc(sizeof(LNode));  
        /*创建新结点*/  
        s->data=a[i]; s->next=L->next;  
        /*将*s插入在原开始结点之前,头结点之后*/  
        L->next=s;  
    }  
}
```

2.3.2 单链表基本运算的实现

□ 尾插法建表

将新结点插到当前链表的表尾上,为此必须增加一个尾指针 r ,使其始终指向当前链表的尾结点:

$H=NULL$ $r=NULL$ /*初始状态*/



线性表: (25,45,18,76,29) 之链表的建立过程

2.3.2 单链表基本运算的实现

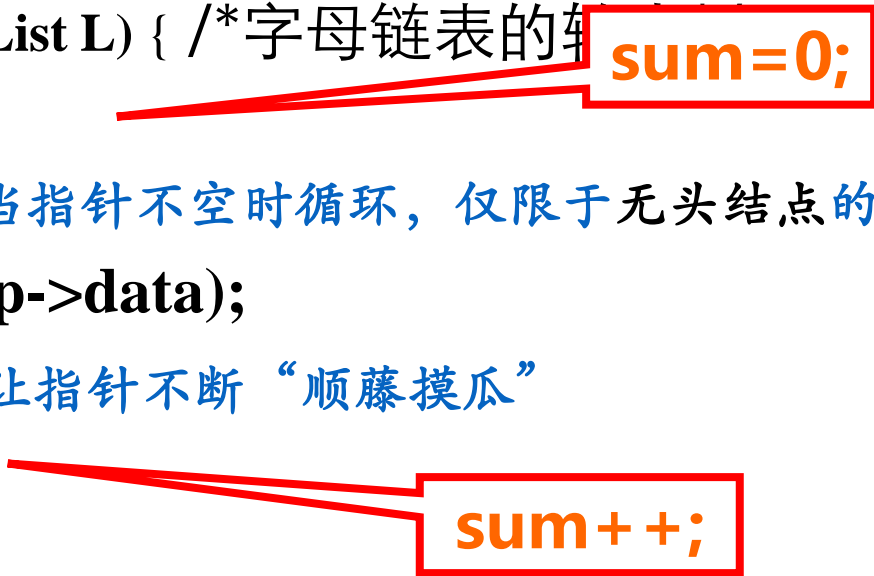
算法思路：初始状态，头指针 $H=NULL$ ，尾指针 $r=NULL$ ；按线性表中元素的顺序依次读入数据元素，不是结束标志时，申请结点，将新结点插入到 r 所指结点的后面，然后 r 指向新结点。

```
void CreateListR(LinkList &L,ElemType a[],int n){
    LNode *s,*r;int i;
    L=(LNode *)malloc(sizeof(LNode)); /*创建头结点*/
    r=L; /*r始终指向终端结点,开始时指向头结点*/
    for (i=0;i<n;i++){
        s=(LNode *)malloc(sizeof(LNode)); /*创建新结点*/
        s->data=a[i];r->next=s; /*将*s插入*r之后*/
        r=s;
    }
    r->next=NULL; /*终端结点next域置为NULL*/
}
```


2.3.2 单链表基本运算的实现

(4)显示单链表

```
void DispList(LinkList L) { /*字母链表的输出*/
    p = L;
    while (p) { //当指针不空时循环，仅限于无头结点的情况
        printf("%c",p->data);
        p=p->next; //让指针不断“顺藤摸瓜”
    }
}
```



讨论：要统计链表中数据元素的个数，该如何改写？

2.3.2 单链表基本运算的实现

(5)单链表的修改(或读取)

思路：要修改第*i*个数据元素，必须从头指针起一直找到该结点的指针*p*，然后才能执行 `p->data=new_value`。

修改第*i*个数据元素的操作函数可写为：

```
Status GetElem_L(LinkList L, int i, ElemType &e) {  
    p=L->next; j=1;    //带头结点的链表  
    while(p&& j<i) {p=p->next; ++j;}  
    if( !p ) return ERROR;  
    p->data =e; //若是读取则为： e=p->data;  
    return OK;  
} // GetElem_L
```

缺点：想寻找单链表中第*i*个元素，只能从头指针开始逐一查询（顺藤摸瓜），无法随机存取。

2.3.2 单链表基本运算的实现

(6)初始化带头结点的线性表

该运算建立一个空的单链表，即创建一个头结点。

```
void InitList(LinkList &L)
{
    L=(LNode *)malloc(sizeof(LNode)); /*创建头结点*/
    L->next=NULL;
}
```

2.3.2 单链表基本运算的实现

(7)销毁带头结点的线性表

释放单链表L占用的内存空间。即逐一释放全部结点的空间。

```
void DestroyList(LinkList &L) {  
    LNode *p=L,*q=p->next;  
    while (q!=NULL)  
    {    free(p);  
        p=q;q=p->next;  
    }  
    free(p);  
}
```

2.3.2 单链表基本运算的实现

(8)判断带头结点的线性表是否为空表

```
int ListEmpty(LinkList L) {  
    return(L->next==NULL);  
}
```

2.3.2 单链表基本运算的实现

(9)求带头结点的线性表长度

```
int ListLength(LinkList L) {  
    LNode *p=L;int i=0;  
    while (p->next!=NULL) {  
        i++;  
        p=p->next;  
    }  
    return(i);  
}
```

2.3.2 单链表基本运算的实现

(10)查找带头结点线性表中的元素

```
int LocateElem(LinkList L,ElemType e) {  
    LNode *p=L->next;int n=1;  
    while (p!=NULL && p->data!=e) {  
        p=p->next; n++;  
    }  
    if (p==NULL) return(0);  
    else return(n);  
}
```

2.3.2 单链表基本运算的实现

(11)合并有序链表

算法要求：

已知：线性表 A 和 B，分别由单链表 L_a 和 L_b 存储，其中数据元素按值非递减有序排列（即已经有序）；

要求：将 A 和 B 归并为一个新的线性表 C，C 的数据元素仍按值非递减排列。设线性表 C 由单链表 L_c 存储。

假设：A = (3, 5, 8, 11)，B = (2, 6, 8, 9, 11)

预测：合并后的 C = (2, 3, 5, 6, 8, 8, 9, 11, 11)

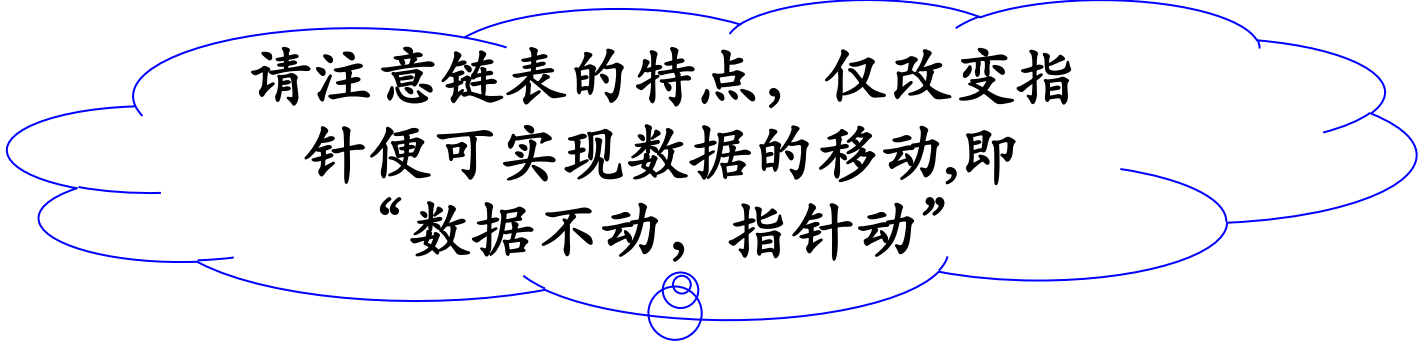
2.3.2 单链表基本运算的实现

算法主要包括搜索、比较、插入三个操作：

搜索：需要设立三个指针来指向La、Lb和Lc链表；

比较：比较La和Lb表中结点数据的大小；

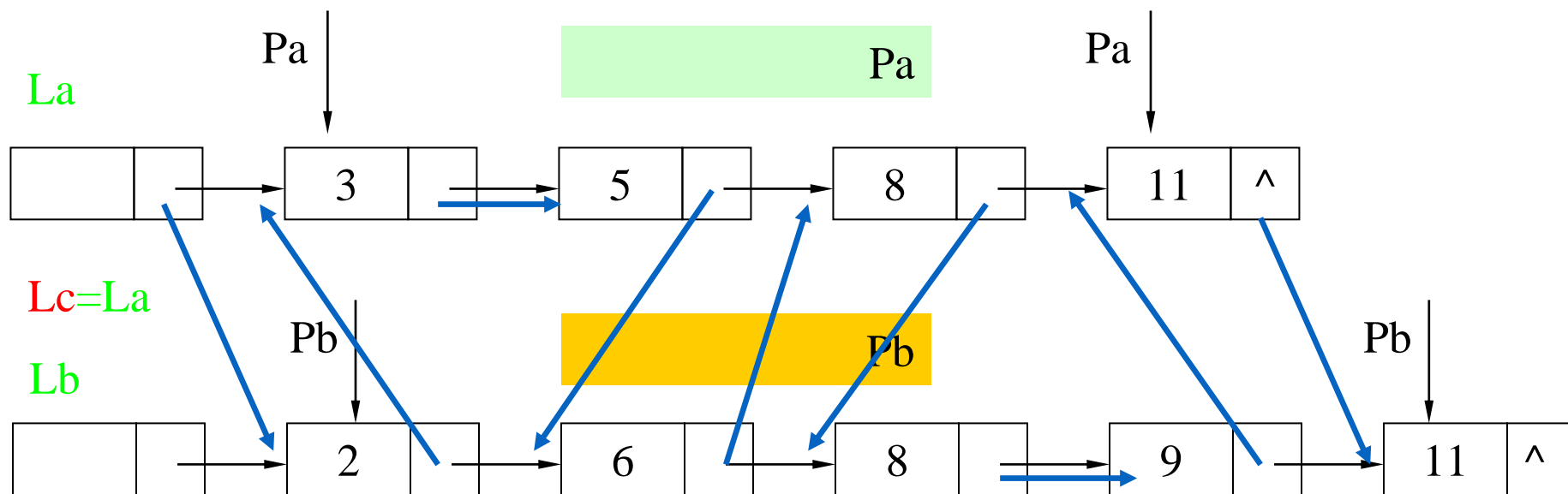
插入：将La和Lb表中数据较小的结点插入新链表Lc。



请注意链表的特点，仅改变指针便可实现数据的移动，即
“数据不动，指针动”

2.3.2 单链表基本运算的实现

P_a 、 P_b 用于搜索 L_a 和 L_b ， P_c 指向新链表当前结点。



2.3.2 单链表基本运算的实现

```
void MergeList_L(LinkList &La, LinkList &Lb, LinkList &Lc) {  
    LinkList pa, pb, pc;  
    pa = La->next; pb = Lb->next;  
    Lc = pc = La;  
    while (pa && pb) {  
        if (pa->data <= pb->data) {  
            pc->next = pa; pc = pa; pa = pa->next;  
        }  
        else {pc->next = pb; pc = pb; pb = pb->next;}  
    }  
    pc->next = pa ? pa : pb;  
    free(Lb);  
}
```

不用Lc，直接把La表插到Lb表中；或者把Lb表插到La表中，怎么修改？

2.3.2 单链表基本运算的实现

合并 $A=A \cup B$

```
void UnionList_L(LinkList &La, LinkList Lb) {
    LinkList p, q, first;  int x;
    first = La->next;  p=Lb->next;
    while (p) {
        x=p->data;  p=p->next;  q=first;
        while (q && q->data !=x)  q=q->next;
        if (!q) {
            q=(LinkList)malloc(sizeof(LNode));
            q->data = x;  q->next = La->next;
            La->next = q;
        }
    }
}
```

2.3.2 单链表基本运算的实现

例 设 $C=\{a_1, b_1, a_2, b_2, \dots, a_n, b_n\}$ 为一线性表,采用带头结点的hc单链表存放,编写一个算法,将其拆分为两个线性表,使得:
 $A=\{a_1, a_2, \dots, a_n\}, B=\{b_1, b_2, \dots, b_n\}$


```
typedef struct LNode {  
    ElemType    data;    //数据域  
    struct Lnode *next;  //指针域  
}LNode, *LinkList;
```

2.3.2 单链表基本运算的实现

```
void fun(LinkList hc, LinkList &ha, LinkList &hb) {
    LNode *p=hc->next,*ra,*rb;
    ha=hc;          /*ha的头结点利用hc的头结点*/
    ra=ha;          /*ra始终指向ha的末尾结点*/
    hb=(LNode *)malloc(sizeof(LNode)); /*创建hb头结点*/
    rb=hb;          /*rb始终指向hb的末尾结点*/
    while (p!=NULL) {
        ra->next=p;ra=p; /*将*p链到ha单链表末尾*/
        p=p->next;
        if (p!=NULL)
        {   rb->next=p;
            rb=p;   /*将*p链到hb单链表末尾*/
            p=p->next;
        }
    }
    ra->next=rb->next=NULL; /*两个尾结点的next域置空*/
}
```

2.3.2 单链表基本运算的实现

下述哪一条是顺序存储结构的优点?()

-  ☒ A 存储密度大
- ☐ B 插入运算方便
- ☐ C 删除运算方便
- ☐ D 可方便地用于各种逻辑结构的存储表示

2.3.2 单链表基本运算的实现

链表不具有的特点是()

- ☐ A 插入、删除不需要移动元素
- ☒ B 可随机访问任一元素
- ☐ C 不必事先估计存储空间
- ☐ D 所需空间与线性长度成正比

正在答疑
