

2.3 线性表的链式表示和实现

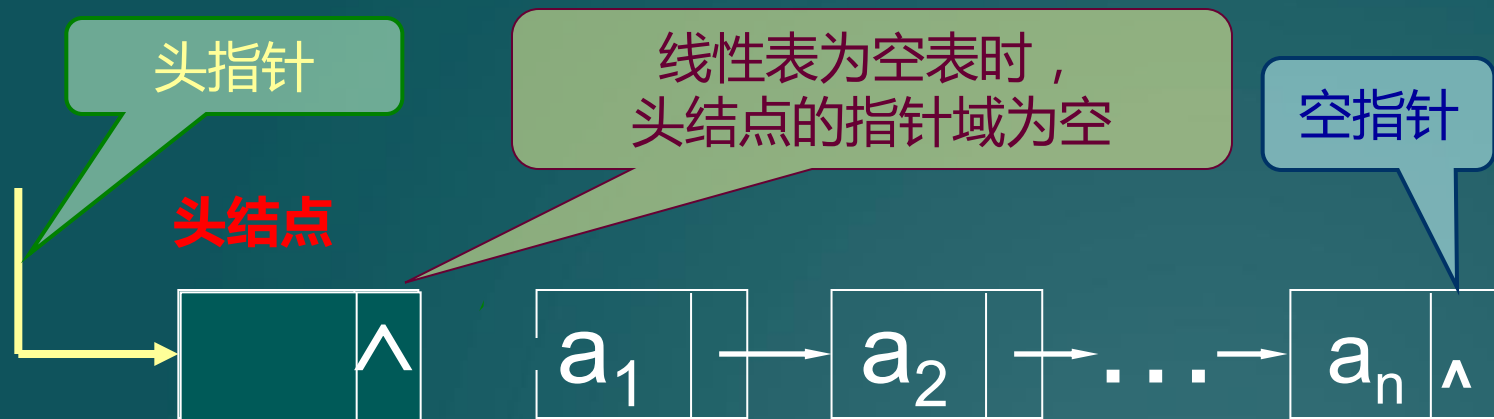
2.3.1 线性链表

用一组地址任意的存储单元存放线性表中的数据元素。

以元素（数据元素的映象）+ 指针（指示后继元素存储位置）= 结点（表示数据元素或数据元素的映象）

以“结点的序列”表示线性表——称作链表

由于此链表的每个结点中只包含一个指针域，故称为线性链表或单链表



以线性表中第一个数据元素 a_1 的存储地址作为线性链表的地址，称作线性链表的头指针。

有时为了操作方便，在第一个结点之前虚加一个“头结点”，以指向头结点的指针为链表的头指针。

结点和单链表的 C 语言描述

```
Typedef struct LNode {  
    ElemType    data; // 数据域  
    struct LNode *next; // 指针域  
} LNode, *LinkList;  
  
LinkList L ; // L 为单链表的头指针，它指向表中第一个  
            个结点。
```

单链表操作的实现

GetElem(L, i, e) // 取第i个数据元素

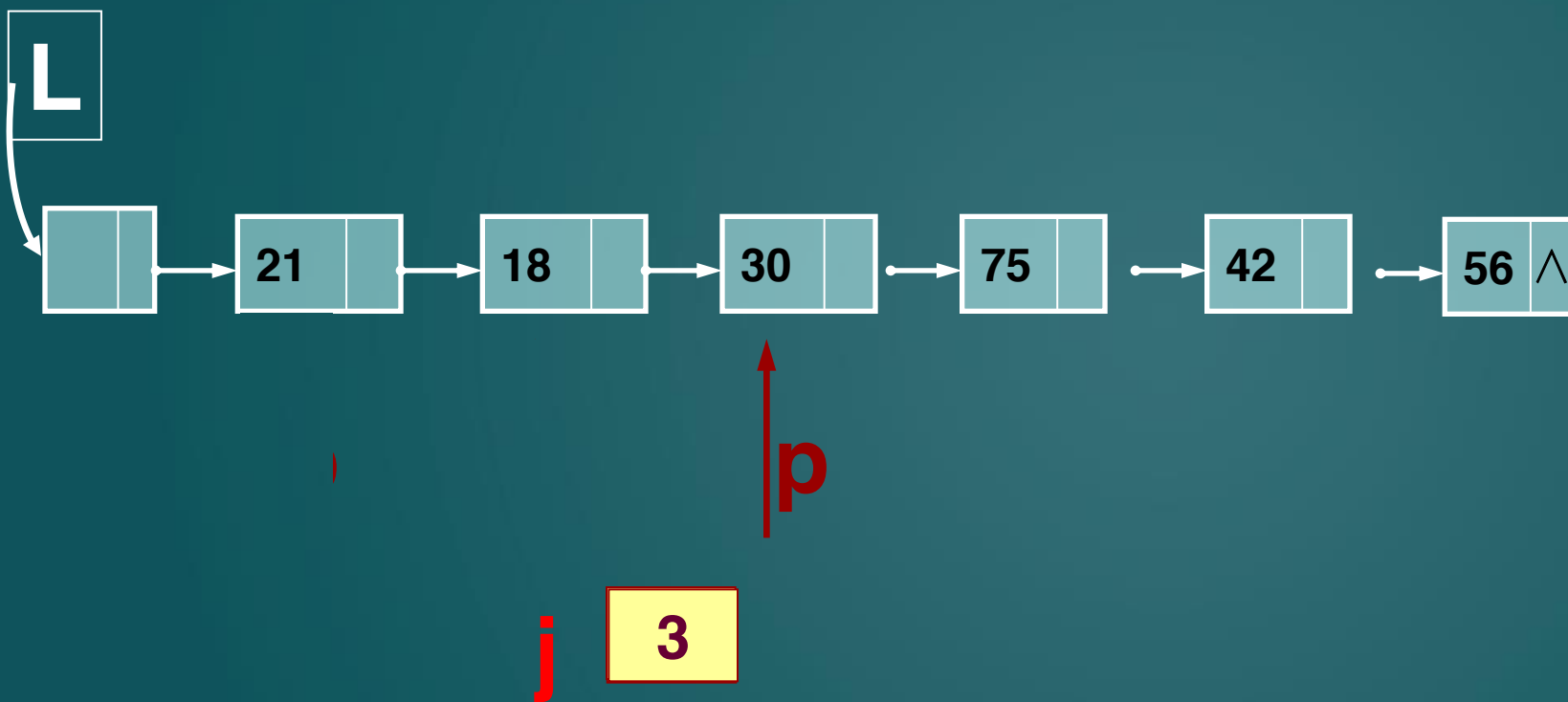
ListInsert(&L, i, e) // 插入数据元素

ListDelete(&L, i, e) // 删除数据元素

ClearList(&L) // 重置线性表为空表

CreateList(&L, n) // 生成含 n 个数据元素的链表

线性表的操作 **GetElem(L, i, &e)**在单链表中的实现:



单链表是一种顺序存取的结构，为找第 i 个数据元素，必须先找到第 $i-1$ 个数据元素。

因此，查找第 i 个数据元素的基本操作为：移动指针，比较 j 和 i 。

令指针 p 始终指向线性表中第 j 个数据元素。

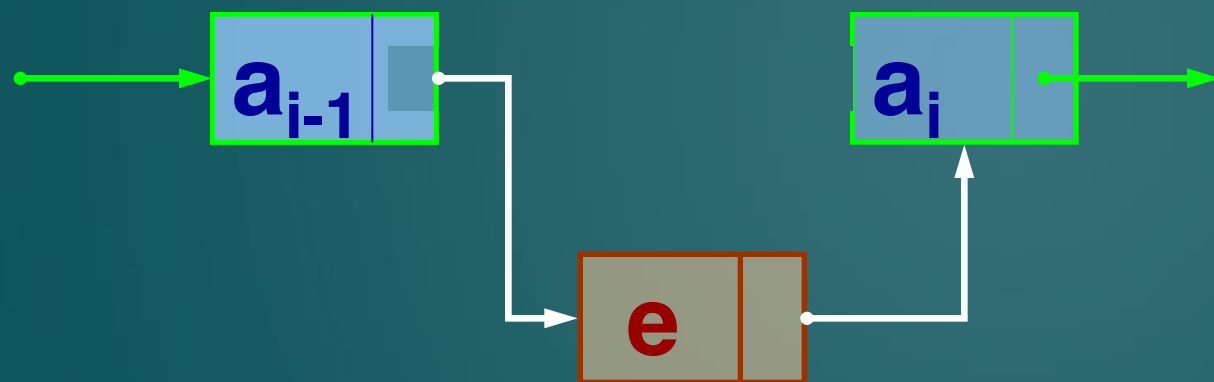
```
Status GetElem_L(LinkList L, int i, ElemType &e) {  
    // L是带头结点的链表的头指针，以 e 返回第 i 个元素  
    p = L->next ;   j = 1 ;   // p指向第一个结点，j为计数器  
    while (p && j<i) { p = p->next ;   ++j ;   }  
    // 顺指针向后查找，直到 p 指向第 i 个元素或 p 为空  
    if ( !p || j>i )  
        return ERROR ;   // 第 i 个元素不存在  
    e = p->data ;           // 取得第 i 个元素  
    return OK ;  
} // GetElem_L
```

算法时间复杂度为： $O(\text{ListLength}(L))$

算法：2.8

线性表的操作 **ListInsert(&L, i, e)** , 在单链表中的实现 :

有序对 $\langle a_{i-1}, a_i \rangle$ 改变为 $\langle a_{i-1}, e \rangle$ 和 $\langle e, a_i \rangle$



可见，在链表中插入结点只需要修改指针。但同时，若要在第 i 个结点之前插入元素，修改的是第 $i-1$ 个结点的指针。

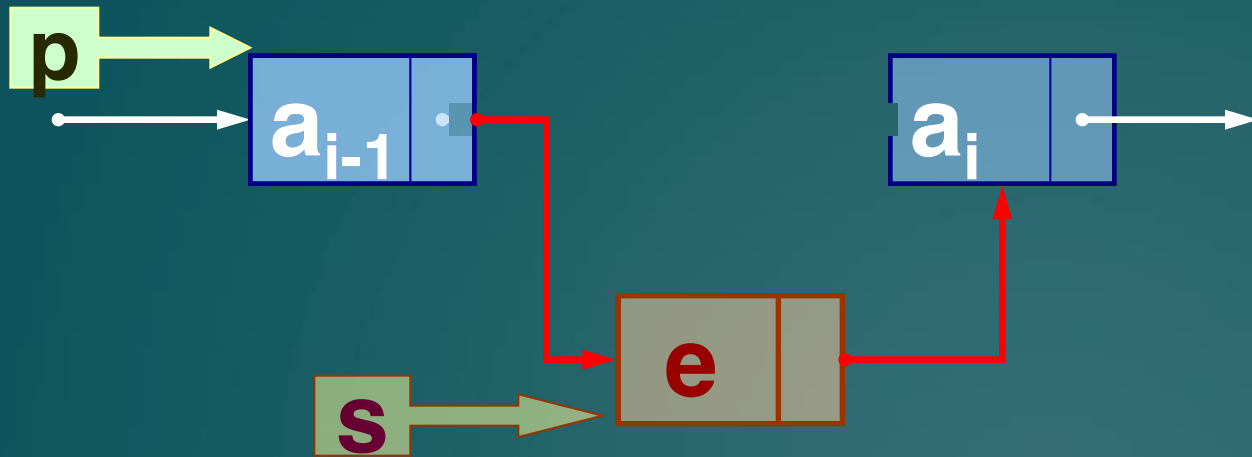
因此，在单链表中第 i 个结点之前进行插入的基本操作为：

找到线性表中第 $i-1$ 个结点，然后修改其指向后继的指针。

```
Status ListInsert_L(LinkList L, int i, ElemType e) {  
    // L 为带头结点的单链表的头指针，本算法在链表中第i 个  
    //结点之前插入新的元素 e  
    p = L ;    j = 0 ;  
    while (p && j < i-1)  
        { p = p->next ; ++j ; }    // 寻找第 i-1 个结点  
    if (!p || j > i-1) return ERROR ;    // i 大于表长或者小于1  
    s = (LinkList) malloc ( sizeof (LNode)) ;    // 生成新结点  
    s->data = e ;  
    s->next = p->next ;    p->next = s ;    // 插入  
    return OK ;  
} // ListInsert_L
```

算法的时间复杂度为： $O(\text{ListLength}(L))$

算法：2.9



$s \rightarrow \text{next} = p \rightarrow \text{next}$

$p \rightarrow \text{next} = s$

线性表的操作**ListDelete** (&L, i, &e)在链表中的实现:

有序对 $\langle a_{i-1}, a_i \rangle$ 和 $\langle a_i, a_{i+1} \rangle$ 改变为 $\langle a_{i-1}, a_{i+1} \rangle$



在单链表中删除第 i 个结点的基本操作为:找到线性表中第 $i-1$ 个结点, 修改其指向后继的指针。

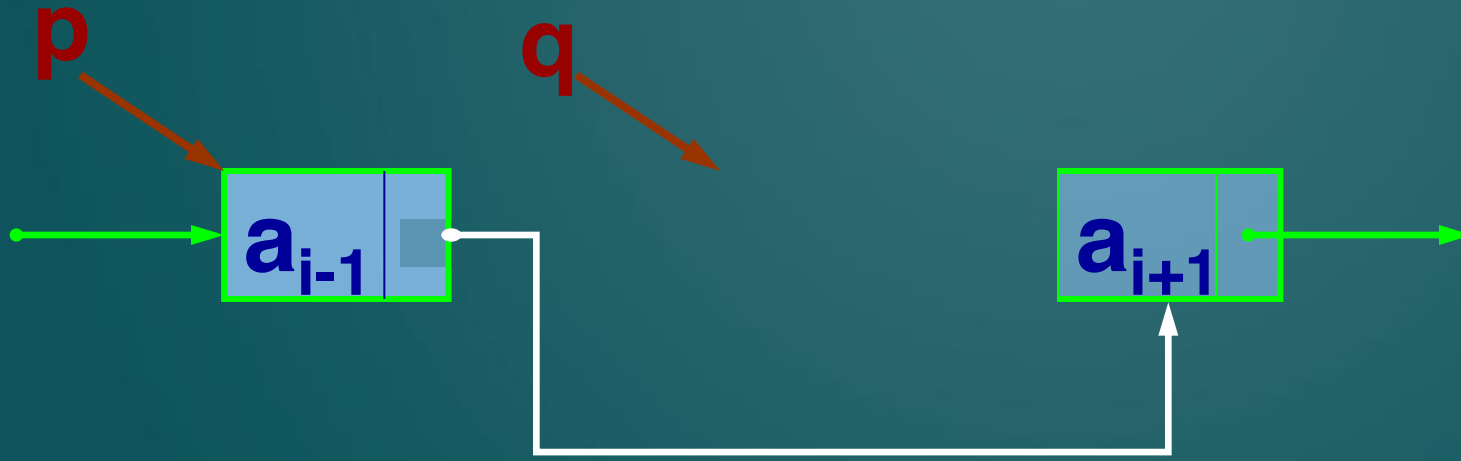
```
q = p->next; p->next = q->next;  


---

e = q->data; free(q);  


---


```



```
Status ListDelete_L(LinkList L, int i, ElemType &e) {  
    // 删除以 L 为头指针（带头结点）的单链表中第 i 个结点  
    p = L ;    j = 0 ;  
    while (p->next && j < i-1) { p = p->next ;    ++j ; }  
        // 寻找第 i 个结点，并令 p 指向其前驱  
    if (!(p->next) || j > i-1)  
        return ERROR ; // 删除位置不合理  
    q = p->next ;    p->next = q->next ; // 删除并释放结点  
    e = q->data ;    free(q) ;  
    return OK ;  
} // ListDelete_L
```

算法的时间复杂度为： $O(\text{ListLength}(L))$

算法：2.10

操作 **ClearList(&L)** 在链表中的实现:

```
void ClearList(&L) {  
    // 将单链表重新置为一个空表  
    while (L->next) {  
        p=L->next ;   L->next=p->next ;  
        free(p) ;  
    }  
} // ClearList
```



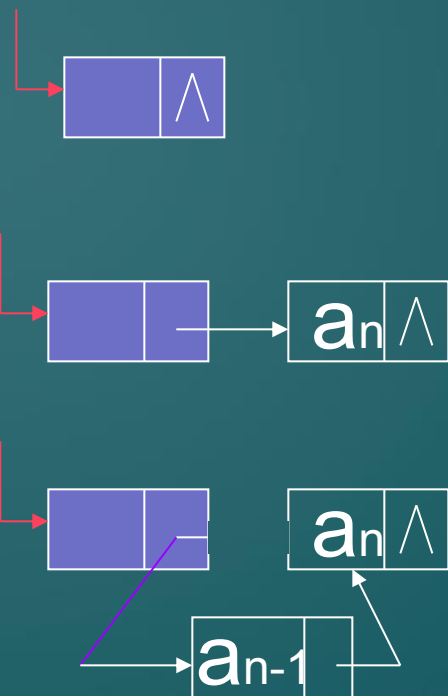
如何从线性表得到单链表？

链表是一个动态的结构，它不需要预分配空间，因此生成链表的过程是一个结点“逐个插入”的过程。

例如：逆位序输入 n 个数据元素的值，建立带头结点的单链表。

操作步骤：

- 一、建立一个“空表”；
- 二、输入数据元素 a_n ，建立结点并插入；
- 三、输入数据元素 a_{n-1} ，建立结点并插入；
- 四、依次类推，直至输入 a_1 为止。




```
void CreateList_L(LinkList &L, int n) {  
    // 逆序输入 n 个数据元素，建立带头结点的单链表  
    L = (LinkList) malloc (sizeof (LNode)) ;  
    L->next = NULL ;    // 先建立一个带头结点的单链表  
    for ( i = n ; i > 0 ; --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 ; // 插入到表头  
    }  
} // CreateList_L
```

算法的时间复杂度为： $O(\text{Listlength}(L))$

算法：2.11

用上述定义的单链表实现线性表的操作时，存在的问题：

- 1．单链表的表长是一个隐含的值；
- 2．在单链表的最后一个元素之后插入元素时，需遍历整个链表；
- 3．在链表中，元素的“位序”概念淡化，结点的“位置”概念加强。

改进链表的设置：

- 1．增加“表长”、“表尾指针”和“当前位置的指针”三个数据域；
- 2．将基本操作中的“位序 i ” 改变为“指针 p ”。

```
void MergeList_L(LinkList &La, LinkList &Lb, LinkList &Lc) {  
    // 已知单链线性表La和 Lb的元素按值非递减排列归并La和 Lb  
    //得到新的单链线性表Lc , Lc的元素也按值非递减排列  
    pa = La->next ;   pb = Lb->next ;  
    Lc = pc = La ;    //用的La头结点作为Lc的头结点  
    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;    //插入剩余段  
    free (Lb) ;              //释放Lb的头结点  
} // MergeList_L
```

算法：2.13

有时也可借用一维数组来描述线性链表，其类型说明如下：

//-----线性表的静态单链表存储结构-----

```
#define MAXSIZE1000 //链表的最大长度
```

```
typedef struct {
```

```
    ElemType data;
```

```
    int    cur;
```

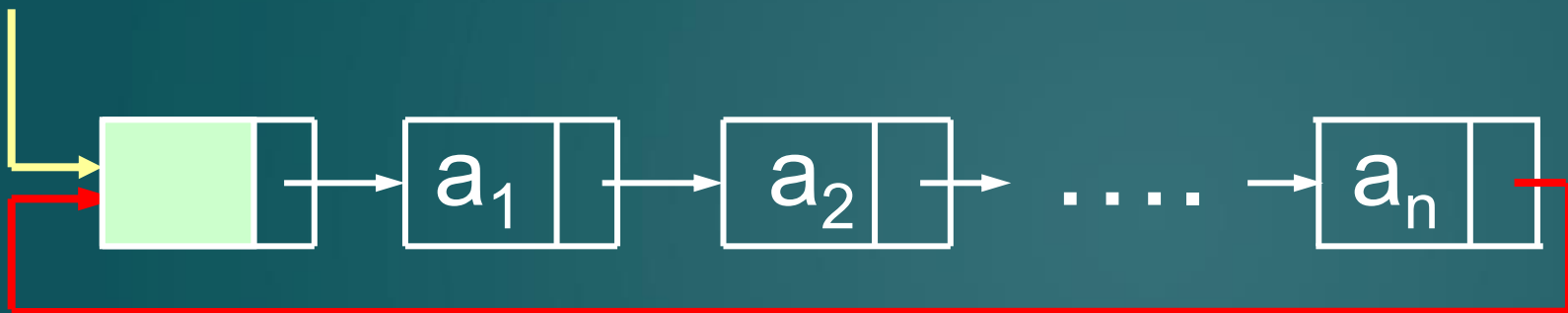
```
}component, SLinkList [ MAXSIZE ];
```

这种用数组描述的链表起名叫**静态链表**。

关于静态链表的内容，讲义P31 – P35

2.3.2 循环链表

其特点是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。



和单链表的差别仅在于，判别链表中最后一个结点的条件不再是“后继是否为空”，而是“后继是否为头结点”。



空表

2.3.3 双向链表

在每个结点中有两个指针域，其一指向直接后继，另一指向直接前驱。

在C语言中可描述如下：

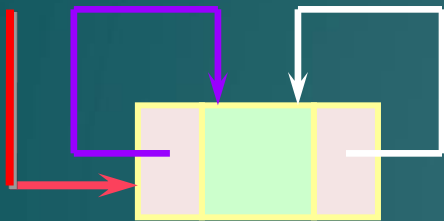
```
typedef struct DuLNode {  
    ElemType    data ;    // 数据域  
    struct DuLNode *prior ;  
                        // 指向前驱的指针域  
    struct DuLNode *next ;  
                        // 指向后继的指针域  
} DuLNode, *DuLinkList ;
```

双向循环链表

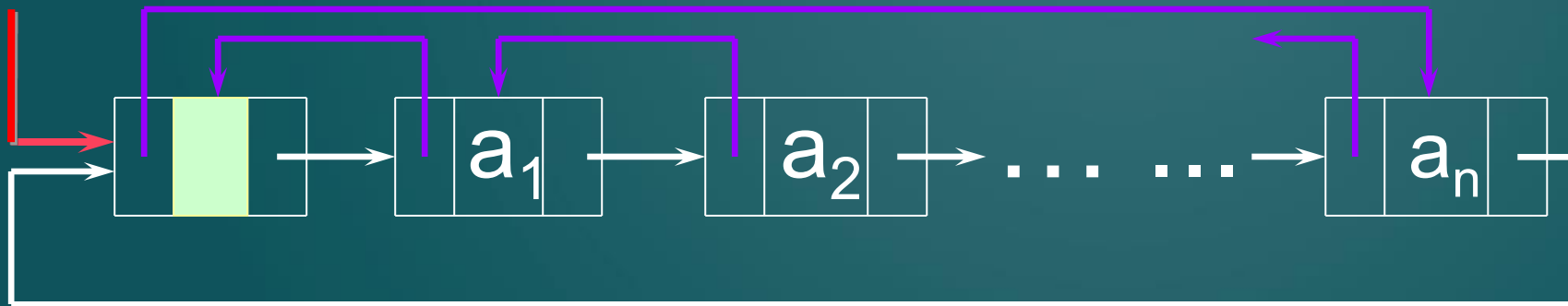
结点结构：



空表



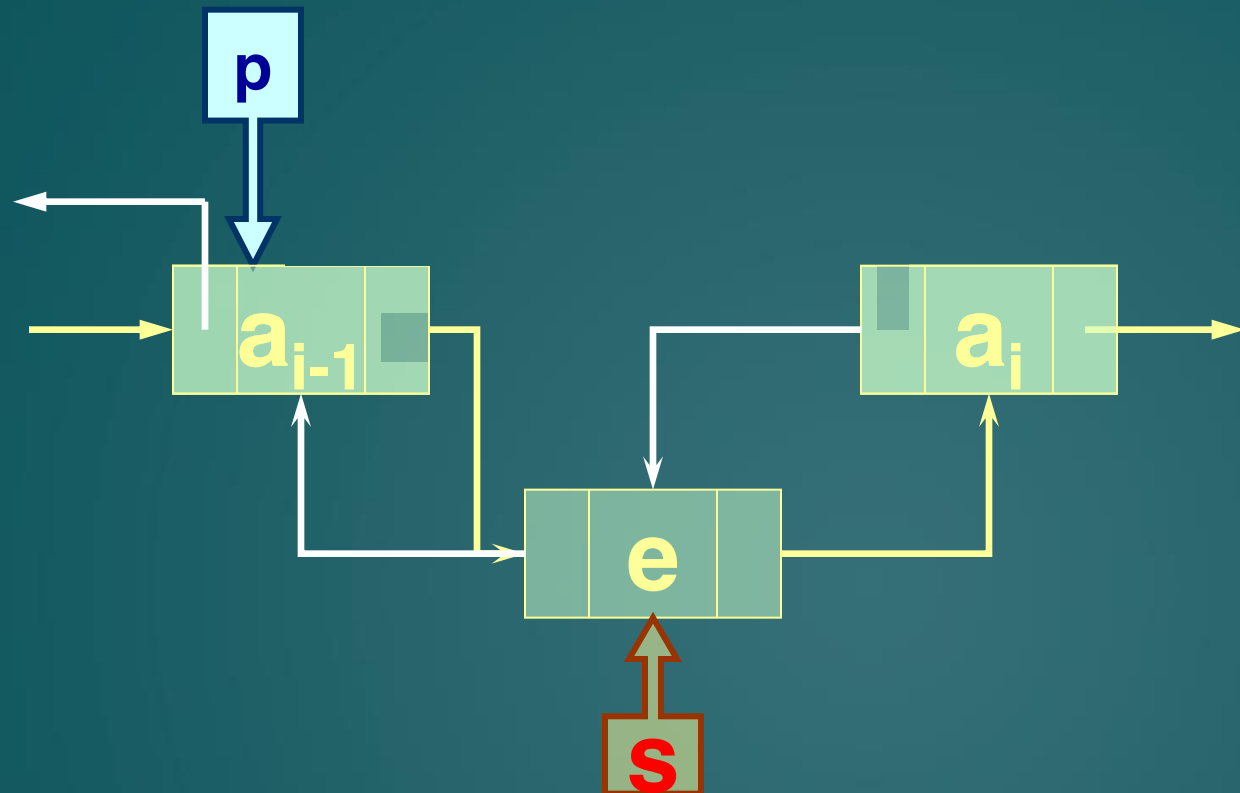
非空表



双向链表的操作特点：

- 1、“查询” 和单链表相同。
- 2、“插入” 和 “删除” 时需要同时修改两个方向上的指针。

插入：



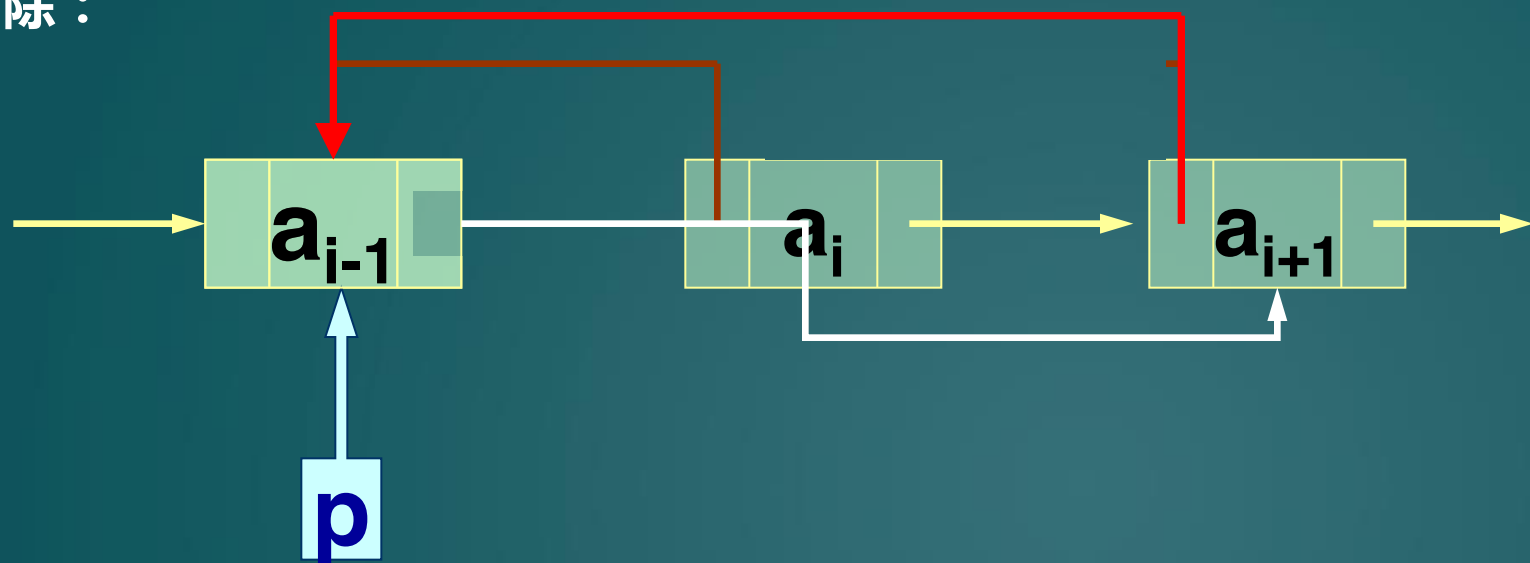
$s \rightarrow \text{next} = p \rightarrow \text{next};$ $p \rightarrow \text{next} = s;$

$s \rightarrow \text{next} \rightarrow \text{prior} = s;$ $s \rightarrow \text{prior} = p;$

```
Status ListInsert_ DuL (DuLinkList &L, int i, ElemType e) {  
    //在带头结点的双链循环线性表L中第 i 个位置之前插入 元素e  
    // i 的合法值为 $1 \leq i \leq \text{表长} + 1$ .  
    if (!(p=GetElemP_ DuL(L,i))) //在L中确定插入位置指针p  
        return ERROR; // i 等于表长加1时 , p 指向头结点 ;  
                        // i 大于表长加1时 , p=NUL  
    if (!(s=(DuLinkList) malloc(sizeof (DuLNode))))  
        return ERROR ;  
    s->data=e ;  
    s->prior=p->prior ; p->prior->next=s ;  
    s->next=p ;      p->prior=s ;  
    return OK ;  
} // ListInsert_ DuL
```

算法 : 2.18

删除：



$p \rightarrow \text{next} = p \rightarrow \text{next} \rightarrow \text{next};$

$p \rightarrow \text{next} \rightarrow \text{prior} = p;$

```
Status ListDelete_ DuL (DuLinkList &L, int i, ElemType &e) {  
    //删除带头结点的双链循环线性表L中第 i 个元素 , i 的合法值为  
    //1≤i≤表长  
    if (!(p=GetElemP_ DuL(L,i)))  
        //在L中确定第 i 个元素的位置指针p  
        return ERROR ; //p=NULL , 即第 i 个元素不存在  
    e=p->data ;  
    p->prior->next=p->next ;  
    p->next->prior=p->prior ;  
    free (p) ;    return OK ;  
} // ListDelete_ DuL
```

算法 : 2.19