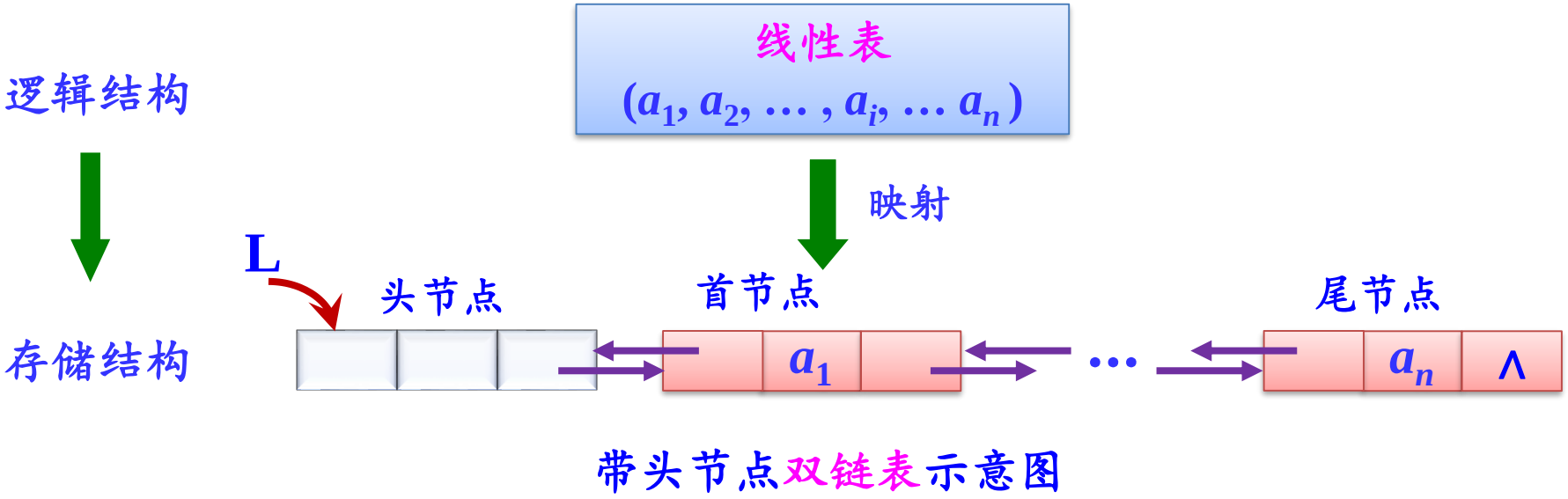
2.3.3 双链表

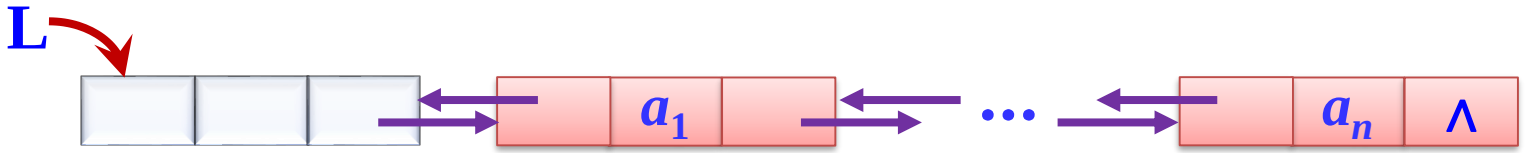
在线性表的链式存储结构中，每个物理节点增加一个指向后继节点的指针域和一个指向前趋节点的指针域 ⇨双链表。



带头节点双链表示意图

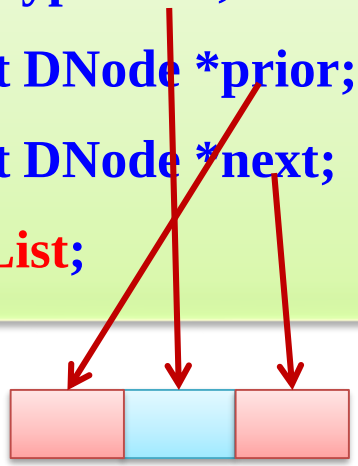
双链表的优点：

- 从任一节点出发可以快速找到其前趋节点和后继节点；
- 从任一节点出发可以访问其他节点。



对于双链表，采用类似于单链表的类型定义，其节点类型DLinkList定义如下：

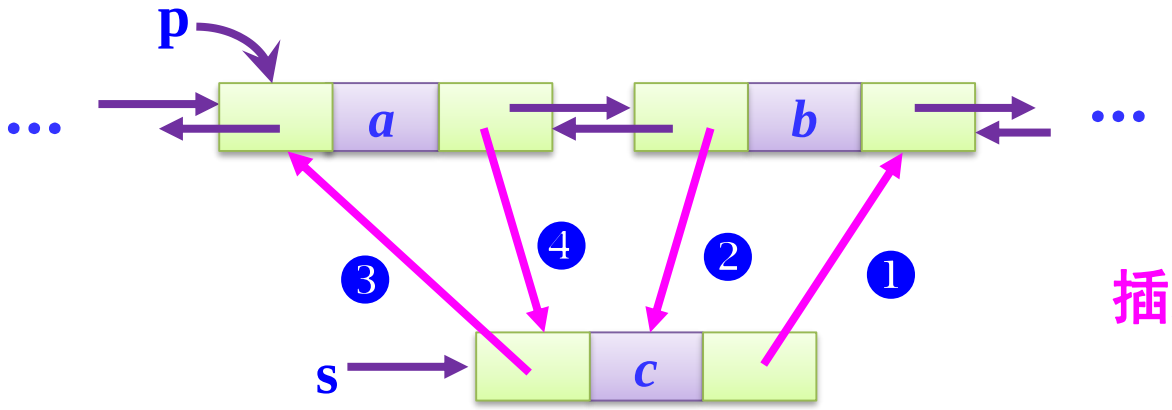
```
typedef struct DNode                    //双链表节点类型
{    ElemType data;
     struct DNode *prior;               //指向前趋节点
     struct DNode *next;                //指向后继节点
} DLinkList;
```



1、双链表中节点的插入和删除

双链表插入节点的演示

在*p节点之后插入节点*s



插入完毕

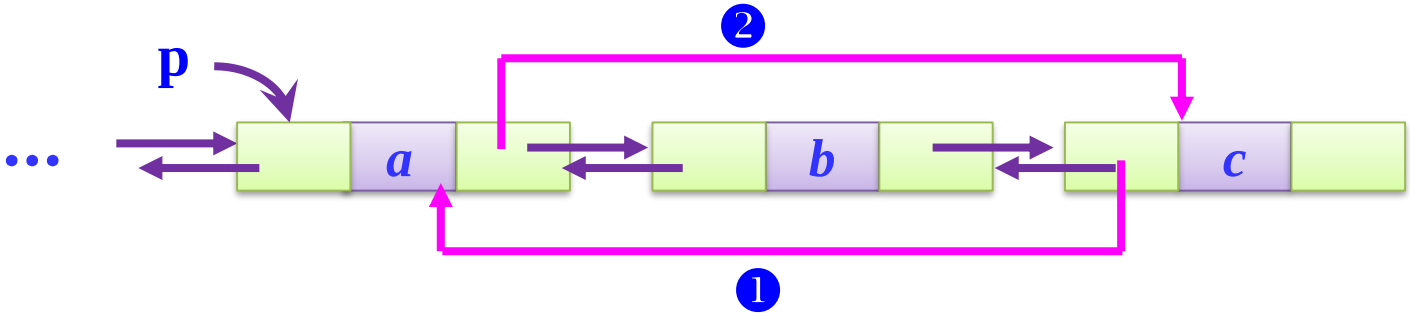
操作语句：

❶ s->next = p->next　　❷ p->next->prior = s

❸ s->prior = p　　❹ p->next = s

双链表删除节点的演示

删除*p节点之后的一个节点



操作语句：

❶ p->next->next->prior = p

❷ p->next = p->next->next

删除完毕

思考题

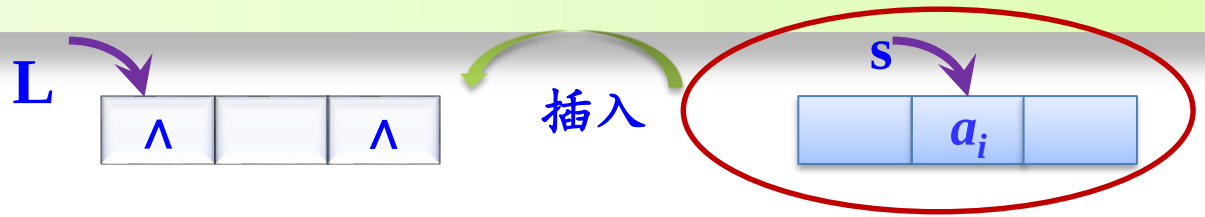
一个带头节点的双链表L（节点个数大于2），插入一个非尾节点的节点，需要修改_____个指针域，删除一个非尾节点的节点，需要修改_____个指针域。

2、建立双链表

整体建立双链表也有两种方法：头插法和尾插法。与单链表的建表算法相似，主要是插入和删除的不同。

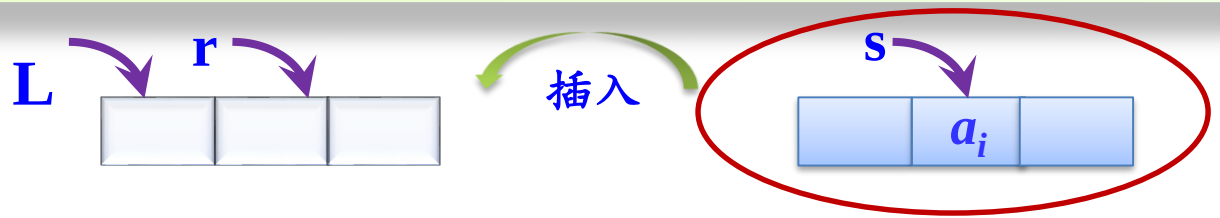
❶ 头插法建立双链表：由数组$a[0..n-1]$创建带头节点的双链表L。

```
void CreateListF(DLinkList *&L,ElemType a[],int n)
{    DLinkList *s; int i;
     L=(DLinkList *)malloc(sizeof(DLinkList));    //创建头节点
     L->prior=L->next=NULL;                       //前后指针域置为NULL
     for (i=0;i<n;i++)                            //循环建立数据节点
     {      s=(DLinkList *)malloc(sizeof(DLinkList));
            s->data=a[i];                         //创建数据节点*s
            s->next=L->next;                      //将*s插入到头节点之后
            if (L->next!=NULL)                    //若L存在数据节点,修改前趋指针
                 L->next->prior=s;
            L->next=s;
            s->prior=L;
     }
}
```



❷ 尾插法建立双链表：由数组$a[0..n-1]$创建带头节点的双链表L。

```
void CreateListR(DLinkList *&L,ElemType a[],int n)
{    DLinkList *s,*r;
     int i;
     L=(DLinkList *)malloc(sizeof(DLinkList));   //创建头节点
     r=L;                                        //r始终指向尾节点,开始时指向头节点
     for (i=0;i<n;i++)                           //循环建立数据节点
     {       s=(DLinkList *)malloc(sizeof(DLinkList));
             s->data=a[i];                       //创建数据节点*s
             r->next=s;s->prior=r;               //将*s插入*r之后
             r=s;                                //r指向尾节点
     }
     r->next=NULL;                               //尾节点next域置为NULL
}
```



3、线性表基本运算在双链表中的实现

和单链表相比，双链表主要是插入和删除运算不同。

❶ 双链表的插入算法：

```
bool ListInsert(DLinkList *&L,int i,ElemType e)
{   int j=0;
    DLinkList *p=L,*s;                  //p指向头节点，j设置为0
    while (j<i-1 && p!=NULL)            //查找第i-1个节点
    {       j++;
            p=p->next;
    }
```

查找第*i*-1个节点*p

```
    if (p==NULL)                          //未找到第i-1个节点,返回false
        return false;
    else                                  //找到第i-1个节点*p,在其后插入新节点*s
    {
        s=(DLinkList *)malloc(sizeof(DLinkList));
        s->data=e;                        //创建新节点*s
        s->next=p->next;                  //在*p之后插入*s节点
        if (p->next!=NULL)                //若存在后继节点,修改其前趋指针
            p->next->prior=s;
        s->prior=p;    p->next=s;
        return true;
    }
}
```

新建节点*s，将其插入到*p节点之后

另外解法：在双链表中，可以查找第*i*个节点，并在它前面插入一个节点。

❷ 双链表的删除算法：

```
bool ListDelete(DLinkList *&L,int i,ElemType &e)
{    int j=0;   DLinkList *p=L,*q;                    //p指向头节点,j设置为0
     while (j<i-1 && p!=NULL)                         //查找第i-1个节点
     {      j++;
            p=p->next;
     }
```

查找第*i*-1个节点*p

```
    if (p==NULL)                    //未找到第i-1个节点
        return false;
    else                            //找到第i-1个节点*p
    {   q=p->next;                  //q指向第i个节点
        if (q==NULL)                //当不存在第i个节点时返回false
            return false;
        e=q->data;
        p->next=q->next;            //从双链表中删除*q节点
        if (p->next!=NULL)          //修改其前趋指针
            p->next->prior=p;
        free(q);                    //释放*q节点
        return true;
    }
}
```

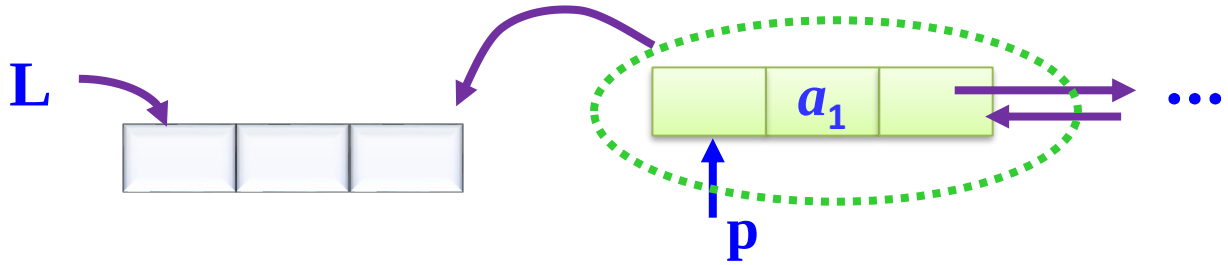
删除*q节点并释放其空间

另外解法：在双链表中，可以查找第*i*个节点，并将它删除。

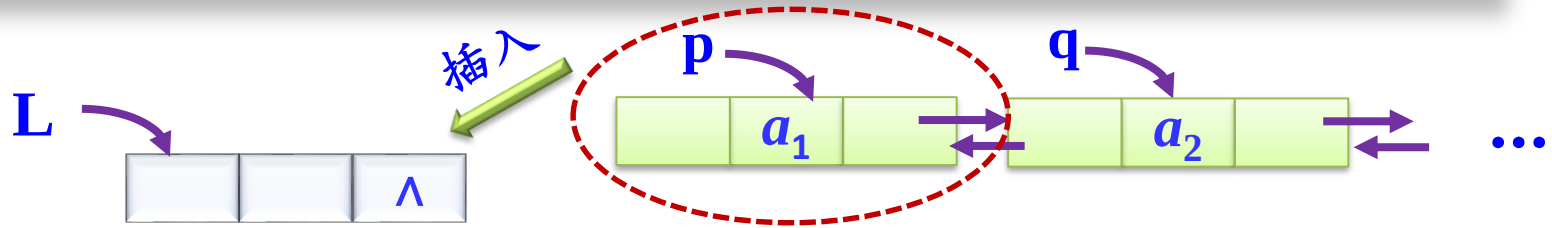
【例2-7】有一个带头节点的双链表L，设计一个算法将其所有元素逆置，即第1个元素变为最后一个元素，第2个元素变为倒数第2个元素，…，最后一个元素变为第1个元素。

算法设计思路：

采用头插法建表。



```
void  Reverse(DLinkList *&L)          //双链表节点逆置
{    DLinkList *p=L->next,*q;         //p指向开好节点
     L->next=NULL;                    //构造只有头节点的双链表L
     while (p!=NULL)                  //扫描L的数据节点
     {       q=p->next;               //用q保存其后继节点
             p->next=L->next;         //采用头插法将*p节点插入
             if (L->next!=NULL)       //修改其前趋指针
                 L->next->prior=p;
             L->next=p;
             p->prior=L;
             p=q;                     //让p重新指向其后继节点
     }
}
```



——本讲完——