



信息与软件工程学院

# 程序设计与算法基础II

主讲教师：陈安龙

# 第2章 线性表

2.1 线性表的概念及定义（**自学**）

2.2 线性表的顺序存储

2.3 线性表的链式存储

2.4 线性表应用（一元多项式的表示及相加）

2.5 顺序表与链表的综合比较（**自学**）

2.6 典型题例



请观察上述图形具有具有什么特征？

- ① 存在**唯一**的一个被称作“**第一个**”的数据元素
- ② 存在**唯一**的一个被称作“**最后一个**”的数据元素
- ③ 除第一个外，集合中的每个数据元素均**只有一个前驱**
- ④ 除最后一个外，集合中的每个数据元素均**只有一个后继**

请举例说明现实世界中还有哪些具有上述特征？

## 2.1 线性表的定义

线性表是一个具有相同特性的数据元素的有限序列。



- ✓ **相同特性**：所有元素属于同一数据类型。
- ✓ **有限**：数据元素个数是有限的。
- ✓ **序列**：数据元素由逻辑序号唯一确定。一个线性表中可以有相同值的元素。
- ✓ **结构**：除第一个元素**无前驱**、最后一个元素**无后继**外，其余每个元素都有**唯一前驱和唯一后继元素**

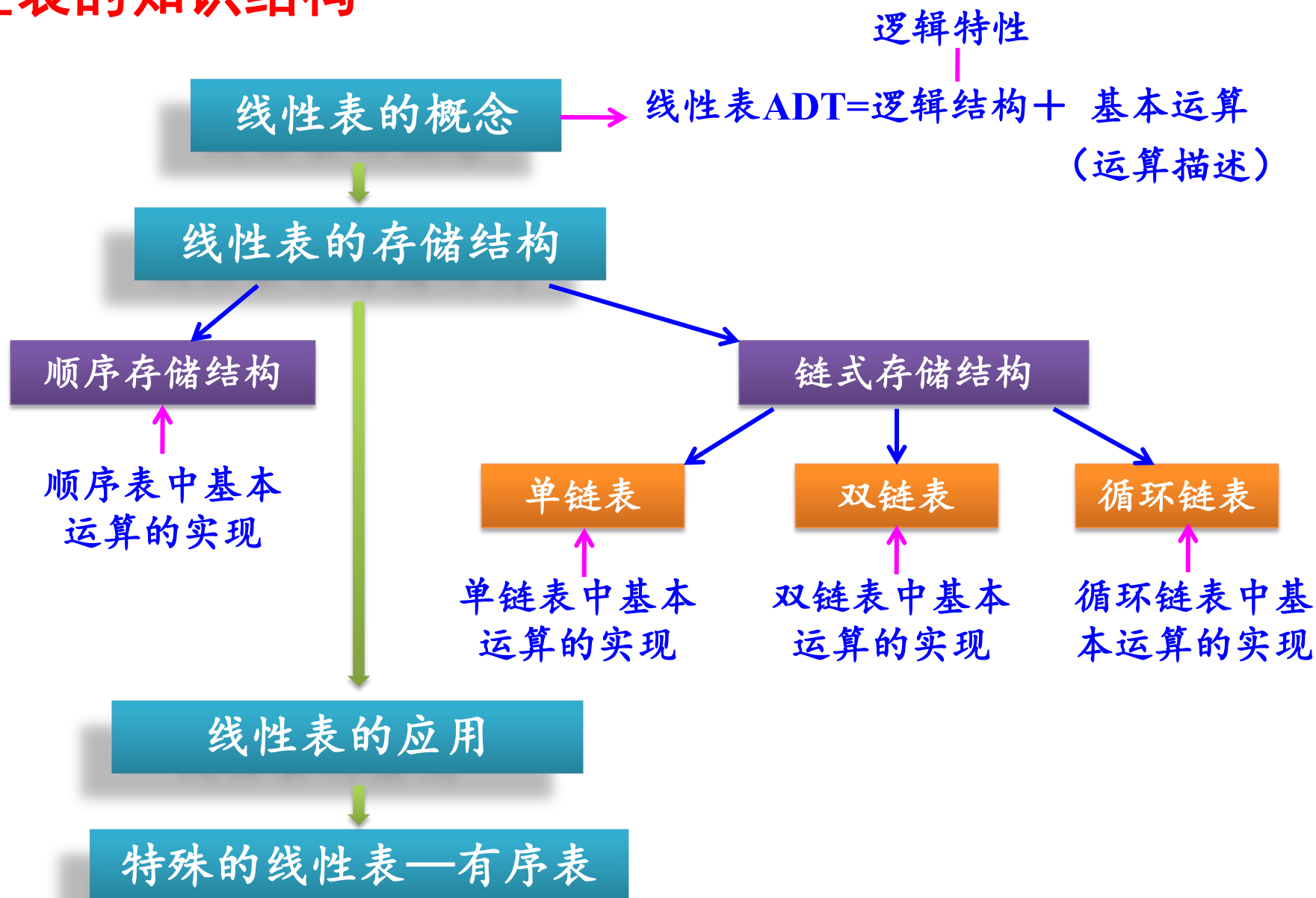
线性表中所含元素的个数叫做**线性表的长度**，用 $n$ 表示， $n \geq 0$ 。 $n=0$ 时，表示线性表是一个空表，即表中不包含任何元素。

线性表的逻辑表示为： $(a_1, a_2, \dots, a_i, a_{i+1}, \dots, a_n)$   $a_i$  ( $1 \leq i \leq n$ ) 表示第 $i$ 个元素。

表头元素

表尾元素

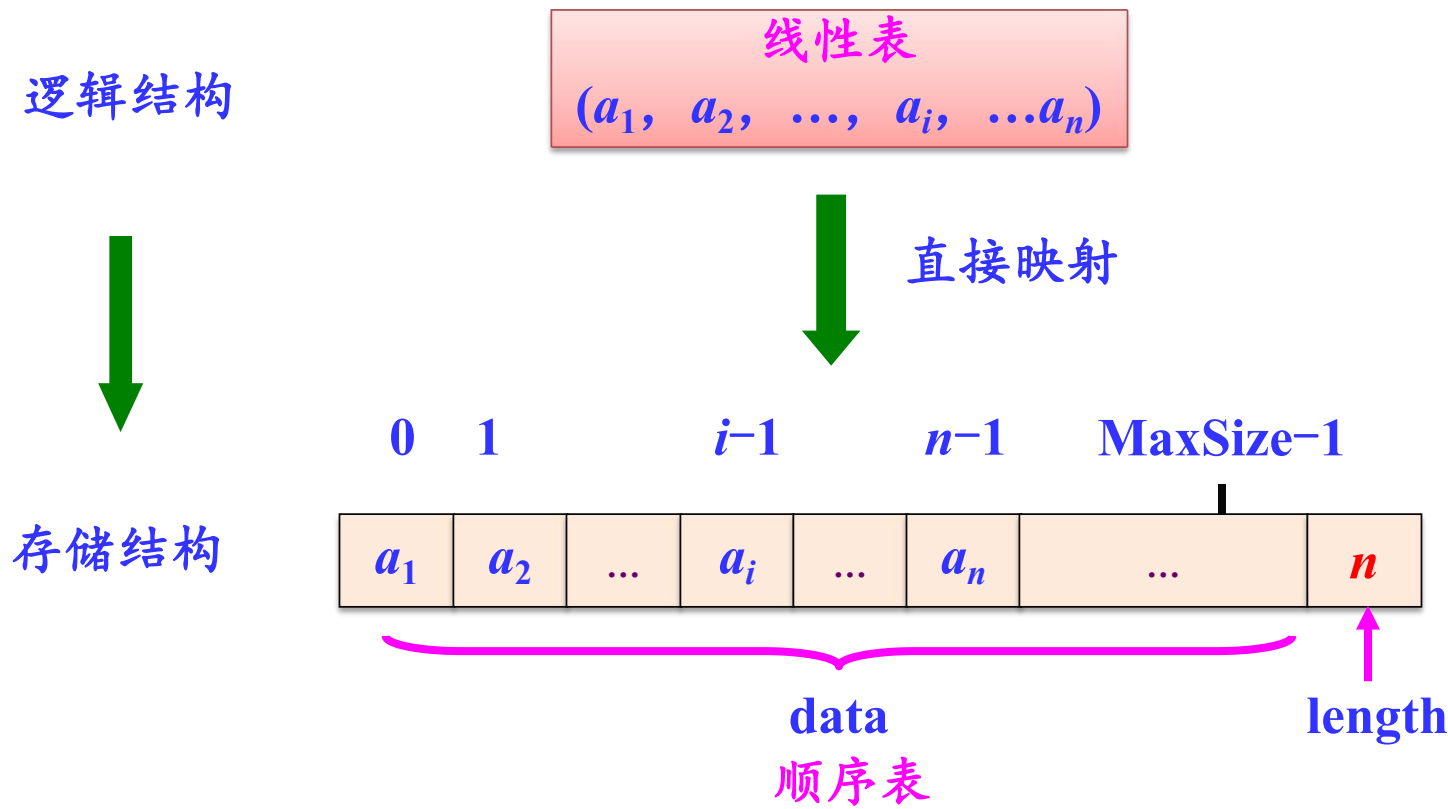
# 线性表的知识结构



# 2.2 线性表的顺序存储结构

线性表的顺序存储结构：把线性表中的所有元素按照顺序存储方法进行存储。

按逻辑顺序依次存储到存储器中一片连续的存储空间中。



## 顺序表类型定义：

**ElemType**为用户定义的类型，在后面的示例中假设为**int**类型

```
#define MAXSIZE 100
typedef struct
{
    ElemType elem[MAXSIZE];
    int last;
} SeqList; //顺序表类型
```

例如：定义  
学生类型

```
typedef struct bookinfo {
    int    No; /*图书编号*/
    char   *name /*图书名称*/
    char   *author /*作者姓名*/
} ElemType;
```

其中：elem成员存放元素，**last**最后一个元素在数组中的位置，元素总个数为**last+1**。

链表定义和使用有下列两种形式：

(1) 定义链表变量 L

```
SeqList L;
```

```
L.elem[i-1]; //访问链表第i个元素
```

(2) 定义链表指针变量 L

```
SeqList L1;
```

```
SeqList* L = &L1;
```

```
L->elem[i-1]; //访问链表第i个元素
```

## 2.2.2 线性表在顺序存储结构上的基本运算

(1) 求某个数据元素值在顺序表中的序号 **Locate**(SeqList L, ElemType e)

该运算返回元素值e在线性表中的序号。

```
int Locate(SeqList L, ElemType e)
{
    int i;
    for(i=0; i<=L.last; i++)
        if(L.elem[i]==e) return i+1;
    return -1;
}
```

本算法的时间复杂度为  $O(n)$ 。



查找某个值在顺序表中第1次出现的位置

下面是在主程序中调用示例

```
//.....
int Locate(SeqList L, ElemType e);
int main(int argc, char *argv[]) {
    SeqList SL;
    ElemType eval;

    //这里可以添加对线性表SL初始化语句
    printf("\nEnter locate position:\n");
    scanf("%d", &eval);
    printf("\n%d\n", Locate(SL, eval));
    system("pause");
    return 0;
}
//下面实现自己的算法函数
```



## 2.2.2 线性表在顺序存储结构上的基本运算

(2) 求序号为*i*的数据元素值 `GetData(SeqList *L, int i, ElemType *e)`

该运算返回L中第 *i* ( $1 \leq i \leq L.\text{last}+1$ ) 个元素的值, 存放在指针变量*e*所指的内存中。

```
int GetData(SeqList *L, int i, ElemType *e)
{
    if (i < 1 || i > L->last+1) return 0;
    *e = L->elem[i-1];
    return 1;
}
```

本算法的时间复杂度为  $O(1)$ 。



体现顺序表的随机存取特性

```
//.....
int GetData(SeqList *L, int i, ElemType *e);
int main(int argc, char *argv[]) {
    SeqList SL;
    ElemType eval;
    int pos;
    int boolval;
    //这里可以添加对线性表SL初始化语句
    printf("\nEnter locate position:\n");
    scanf("%d", &pos);
    boolval = GetData(&SL, pos, &eval);
    if (boolval == 1) printf("\n%d\n", eval);
    system("pause");
    return 0;
}
//下面实现自己的算法函数
```

## 2.2.2 线性表在顺序存储结构上的基本运算

求序号为  $i$  的数据元素值的另外一种实现方法 `GetData(SeqList *L, int i)`

该运算返回  $L$  中第  $i$  ( $1 \leq i \leq L.\text{last}+1$ ) 个元素的值。

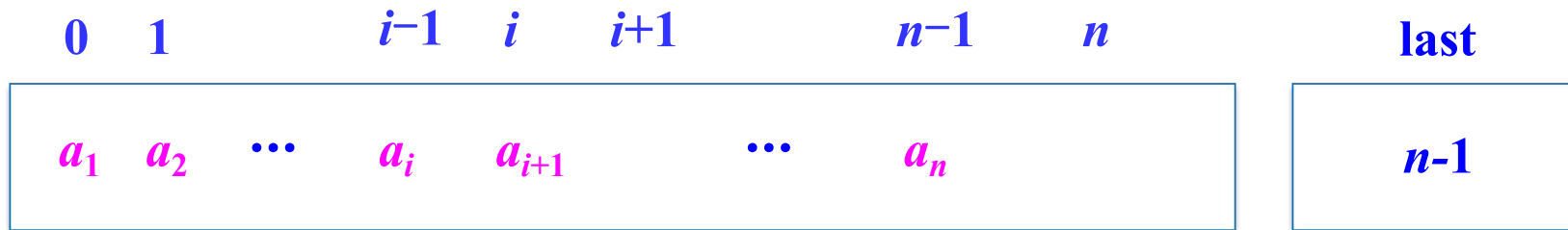
```
ElemType GetData(SeqList *L, int i)
{
    if(i >= 1 && i <= L.last+1)
        return L.elem[i-1];
    else
        return '\0';
}
```

算法时间复杂度同样为  $O(1)$ 。

## 2.2.2 线性表在顺序存储结构上的基本运算

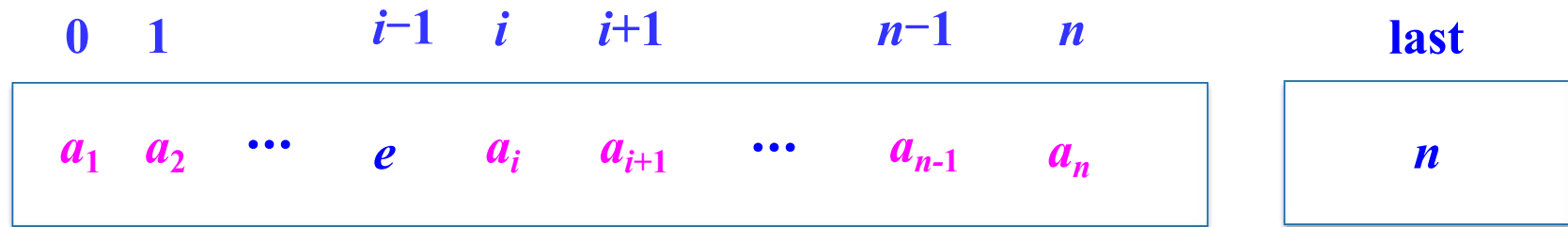
### (3) 顺序表插入数据元素 InsList (L, i, e)

该运算在顺序表L的第 $i$  ( $1 \leq i \leq \text{last}+2$ ) 个位置上插入新的元素 $e$ 。



$e$

插入完成后:



## 插入算法如下：

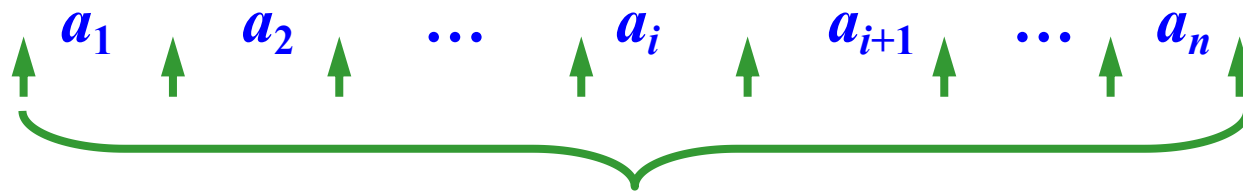
```
#define OK 1
#define ERROR 0
int InsListSeqList (*L, int i, ElemType e)
{   int k;
    if (i<1 || i>L->last+2) {
        printf("\nThe Insert Postion is invalide!");
        return ERROR;//参数错误时返回ERROR
    }
    if(L->last>=MAXSIZE-1) {
        printf("\nThe SeqList is full!");
        return ERROR;
    }
    for(k=L->last;k>=i-1;k--) L->elem[k+1]=L->elem[k];//将data[i..n]元素后移一个位置
    L->elem[i-1]=e;           //插入元素e
    L->last++;                //顺序表最后元素位置增1
    return OK;               //成功插入返回OK
}
```

该算法元素移动的次数不仅与表长 $L \rightarrow \text{last}+1=n$ 有关，而且与插入位置 $i$ 有关：

- 当 $i=n+1$ 时，移动次数为0；
- 当 $i=1$ 时，移动次数为 $n$ ，达到最大值。

算法最好时间复杂度为 $O(1)$     算法最坏时间复杂度为 $O(n)$

## 平均情况分析：



在线性表L中共有 $n+1$ 个可以插入元素的地方

在插入元素 $a_i$ 时，若为等概率情况，则 $p_i = \frac{1}{n+1}$

此时需要将 $a_i \sim a_n$ 的元素均后移一个位置，共移动 $n-i+1$ 个元素。

所以在长度为 $n$ 的线性表中插入一个元素时所需移动元素的平均次数为：

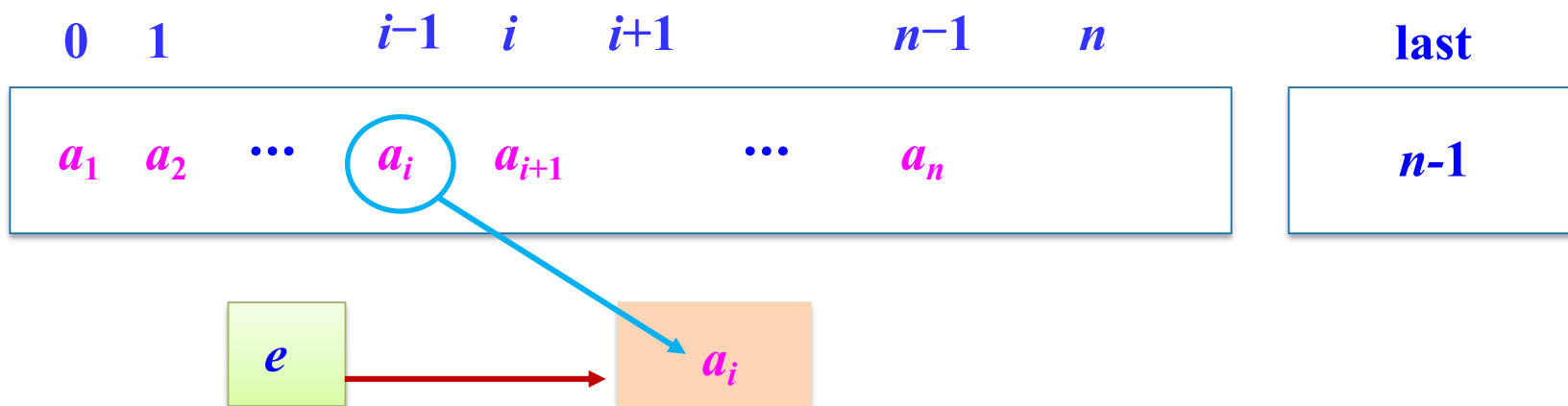
$$\sum_{i=1}^{n+1} p_i (n-i+1) = \sum_{i=1}^{n+1} \frac{1}{n+1} (n-i+1) = \frac{n}{2}$$

因此插入算法的平均时间复杂度为 $O(n)$ 。

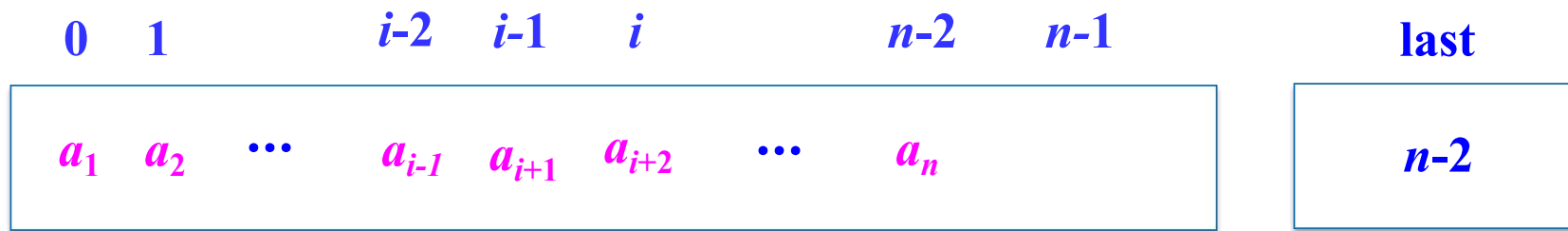
## 2.2.2 线性表在顺序存储结构上的基本运算

### (4) 顺序表删除数据元素 DelList (L, i, e)

在顺序表L中删除第 $i$  ( $1 \leq i \leq \text{last}+1$ ) 个位置上元素，并用指针参数 $e$ 返回。



删除完成后:



## 顺序表删除第i个元素的算法实现

```
int DelList(SeqList *L,int i,ElemType *e)
{ int k;
  if(i<1 || i>L->last+1)
  {printf("删除位置不合理");return 0;}

  if(L->last<0)
  {printf("线性表空！ ");return 0;}

  *e=L->elem[i-1];
  for(k=i;k<=L->last;k++)
    L->elem[k-1]=L->elem[k];

  L->last--;
  return 1;
}
```

## 下面是在主程序中调用示例

```
//.....
int DelList(SeqList *L,int i,ElemType *e);
int main(int argc, char *argv[]) {
  SeqList SL;
  ElemType eval;
  int pos;
  int boolval;
  //这里可以添加对线性表SL初始化语句
  printf("\nEnter locate position:\n" );
  scanf("%d",&pos);
  boolval=DelList(&SL,pos, &eval);
  if (boolval=1) printf("\n%d\n",eval);
  system("pause");
  return 0;
}
//下面将是实现算法的函数
```

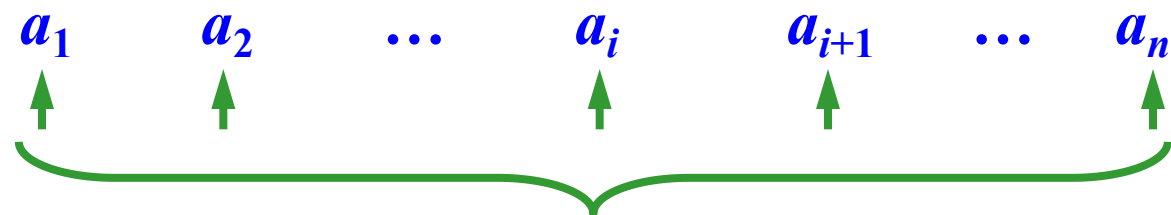


对于本算法来说，元素移动的次数也与表长 $n$ 和删除元素的位置 $i$ 有关：

- 当 $i=n$ 时，移动次数为0；
- 当 $i=1$ 时，移动次数为 $n-1$ 。

删除算法最好时间复杂度为 $O(1)$     删除算法最坏时间复杂度为 $O(n)$

## 平均情况分析：



在线性表L中共有 $n$ 个可以删除元素的地方

在删除元素 $a_i$ 时，若为等概率情况，则 $p_i = \frac{1}{n}$

此时需要将 $a_{i+1} \sim a_n$ 的元素均前移一个位置，共移动 $n - (i+1) + 1 = n - i$ 个元素。

所以在长度为 $n$ 的线性表中删除一个元素时所需移动元素的平均次数为：

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

因此删除算法的平均时间复杂度为 $O(n)$ 。

```
#define MAXSIZE 100
```

```
typedef struct
```

```
{   ElemType elem[MAXSIZE];
```

```
    int last;
```

```
} SeqList; //顺序表类型
```

这种存储称为顺序表的静态存储结构



顺序表的静态存储有什么缺陷？

如何解决？

# 顺序表的动态存储

- 将线性表的用指针表示为:

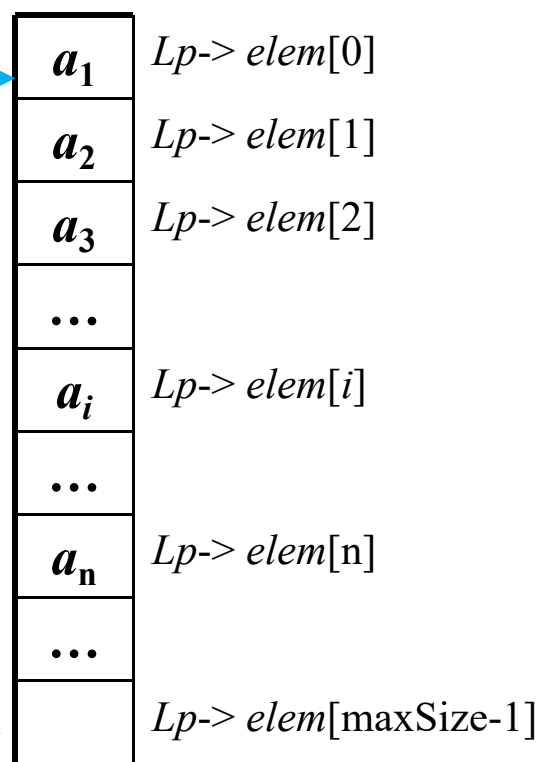
```
typedef struct  
{ ElemType *elem;  
  int last;  
  int maxSize;  
} SeqList, *SeqListPtr;
```

**Last+1**为线性表的当前长度。

**maxsize**为当前分配给线性表的存储空间

指针变量**elem**指向线性表的基地址。

数据区



这时：定义类型为线性表的变量：SeqList *La*;

定义类型为线性表的指针：SeqListPtr *Lp*;或 SeqList \**Lp*;

动态申请内存

*Lp*->elem = (ElemType \*)malloc(Maxsize\*sizeof(ElemType));

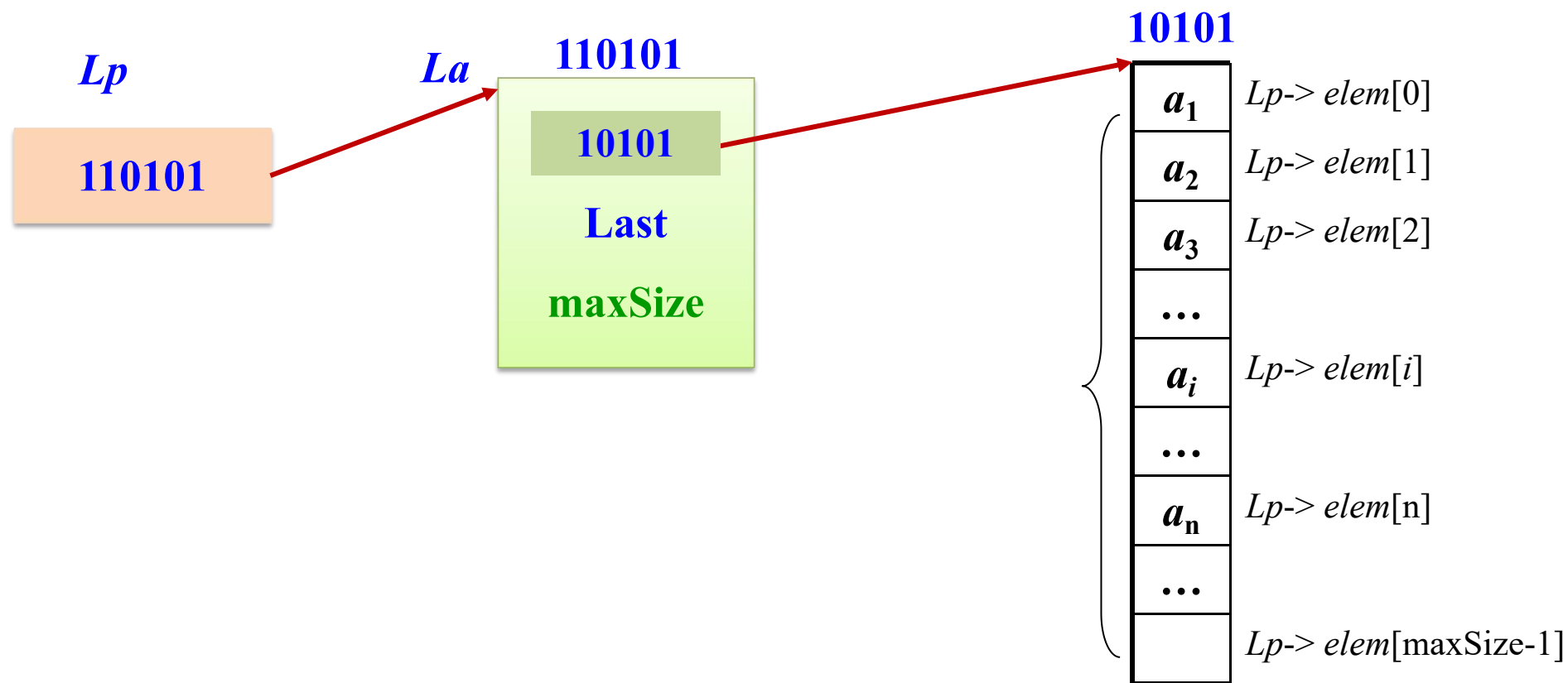
或 (\**Lp*).elem = (ElemType \*)malloc(Maxsize\*sizeof(ElemType));

第1个元素： *Lp*->elem[0], 第2个元素： *Lp*->elem[1] .....

第*n*个元素： *Lp*->elem[*n*-1]

## 动态分配的示意图

```
SeqList La;  
SeqList *Lp=&La;  
Lp->elem = (ElemType *)malloc(Maxsize*sizeof(ElemType));
```



# 创建动态存储的顺序表示例

指定将创建的线性表最多可存储的元素个数

```
int DynSeqListInit (SeqList *L,int maxSize)
{
    L->elem= (ElemType *)malloc(maxSize*sizeof(ElemType));
    if (L->elem)
    {
        L->last=-1;
        L->maxSize=maxsize;
        return 1;
    }
    return 0;
}
```

**【例2-3】** 假设有两个顺序表LA和LB均为非递减有序序列（称为有序表）。设计一个算法，将它们合并成一个有序表LC，称为二路归并。



例如：LA = (1, 3, 5)，LB = (2, 4, 6, 8)，其二路归并过程如下：



LA、LB中每个元素恰好遍历一次，时间复杂度为 $O(m+n)$ 。

采用顺序表存放有序表时，二路归并算法如下：

```
void merge(SeqList *LA, SeqList *LB, SeqList *LC)
{
    int i,j,k;
    i=0;j=0;k=0;
    while(i<=LA->last&& j<=LB->last)
        if(LA->elem[i]<=LB->elem[j])
        {
            LC->elem[k]= LA->elem[i];
            i++; k++;
        } else {
            LC->elem[k]=LB->elem[j];
            j++;k++;
        }
    while(i<=LA->last)      /*当表LA有剩余元素时，则将表LA余下的元素赋给表LC*/
    {
        LC->elem[k]= LA->elem[i];
        i++;k++;
    }
    while(j<=LB->last) /*当表LB有剩余元素时，则将表LB余下的元素赋给表LC*/
    {
        LC->elem[k]= LB->elem[j];
        j++; k++;
    }
    LC->last=LA->last+LB->last+1;
}
```

两个有序表中  
没有遍历完



## 思考题

① 假如有一个学生表，每个学生包含学号、姓名和分数。

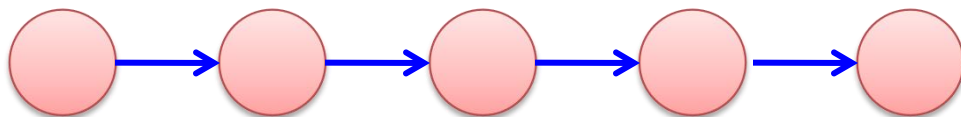
你如何设计相应的学生顺序存储？

② 如果需要对该学生表进行插入、修改和删除运算，你如何实现相关算法？

## 2.3 线性表的链式存储结构

### 2.3.1 线性表的链式存储—链表

线性表中每个中间结点有**唯一**的前驱结点和后继结点。



设计链式存储结构时，每个逻辑结点存储单独存储，为了表示逻辑关系，增加**指针域**。

- 每个物理结点增加一个指向后继结点的指针域 ⇨ **单链表**。
- 每个物理结点增加一个指向后继结点的指针域和一个指向前驱结点的指针域 ⇨ **双向链表**。

逻辑结构

线性表  
 $(a_1, a_2, \dots, a_i, \dots, a_n)$

映射

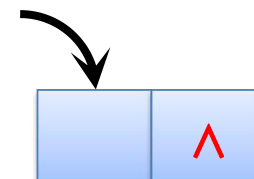
存储结构

L



带头结点单链表示意图

带头结点空单链表



不带头结点单链表



单链表增加一个头结点的优点如下：

- 第一个结点的操作和表中其他结点的操作相一致，无需进行特殊处理；
- 无论链表是否为空，都有一个头结点，因此空表和非空表的处理也就统一了。

## 思考题：

线性表的顺序存储结构和链式存储结构的各自的优缺点？

## 2.3.2 单链表

单链表中结点类型LinkedList的定义如下:

```
typedef struct Node      //定义单链表结点类型
{
    ElemType data;
    struct Node *next;    //指向后继结点
} Node, *LinkedList;
```

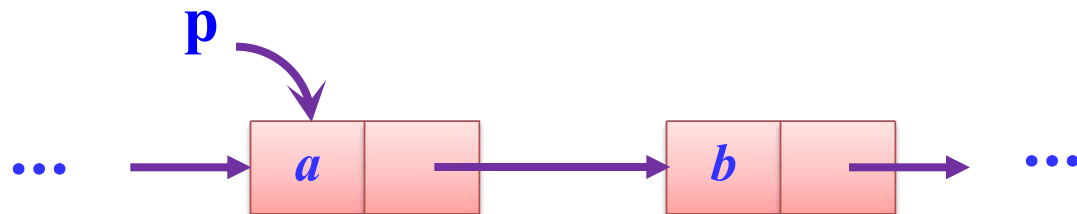


**Node\*** 和 **LinkedList** 为同一结构。

例如: **Node\* Lp1**和 **LinkedList Lp2** 都是定义了指向单链表结点指针

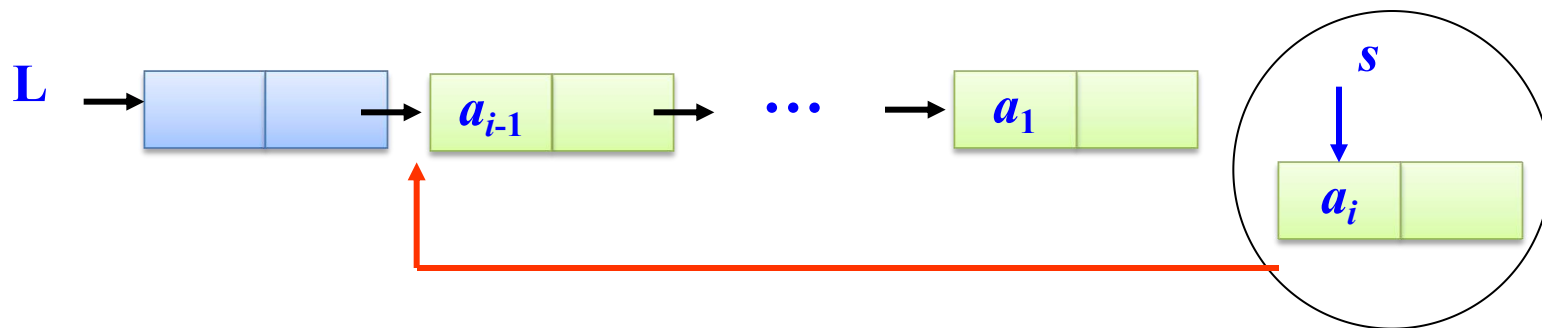
## 单链表的特点

当访问过一个结点后，只能接着访问它的后继结点，而无法访问它的前驱结点。



## (1) 头插法建表

- 从一个空表开始，创建一个头结点。
- 依次读取字符的元素，生成新结点
- 将新结点插入到当前链表的表头上，直到结束为止。



**注意：**链表的结点顺序与逻辑次序相反。

# 单链表的初始化

```
int InitList(LinkList *L)
{
    *L=(LinkList)malloc(sizeof(Node)); //创建头结点
    (*L)->next=NULL;
    return 0;
}
```

```
typedef struct Node    //定义单链表结点类型
{
    ElemType data;
    struct Node *next; //指向后继结点
} Node, *LinkList;
```

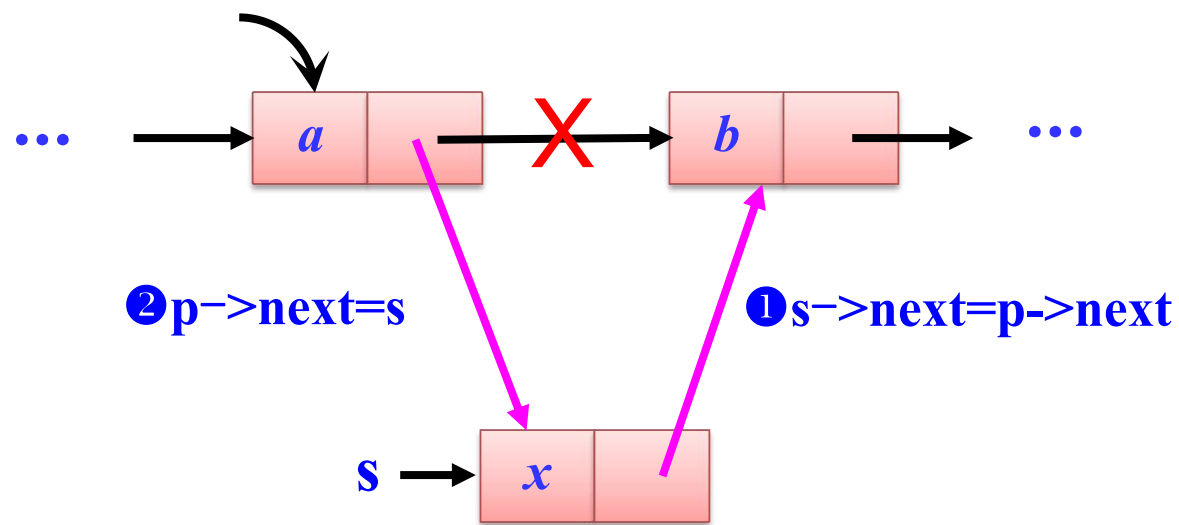




## 插入结点操作

插入操作：将值为 $x$ 的新结点 $*s$ 插入到 $*p$ 结点之后。

特点：只需修改相关结点的指针域 $p$ ，不需要移动结点。



插入操作语句描述如下：

- ①  $s \rightarrow \text{next} = p \rightarrow \text{next};$
- ②  $p \rightarrow \text{next} = s;$

## 头插法建单链表算法如下：

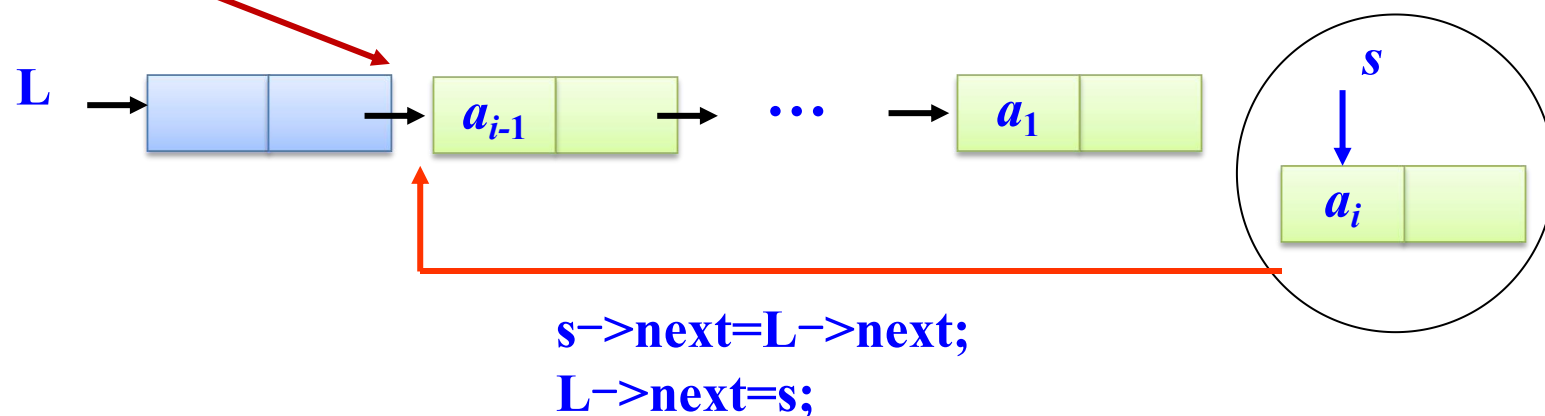
```
void CreateFromHead(LinkList L)
{
    Node *s;
    char c;
    int flag=1;
    while(flag) {
        c=getchar();
        if(c!='$') {
            s=(Node*)malloc(sizeof(Node)); //创建数据结点
            s->data=c;
            s->next=L->next;
            L->next=s;
        }
        else flag=0;
    }
}
```

请分析该算法实现有什么问题？

在函数里包含多重功能：

- (1) 由键盘交互输入结点数据；
- (2) 创建链表结点

设计函数尽量让操作单一化



## 头插法建单链表的改进算法如下：

```
void CreateFromHead(LinkList L,ElemType data[], int n)
```

```
{
```

```
    Node *s;
```

```
    int i=0;
```

```
    while(i<n) {
```

```
        s=(Node*)malloc(sizeof(Node)); //创建数据结点
```

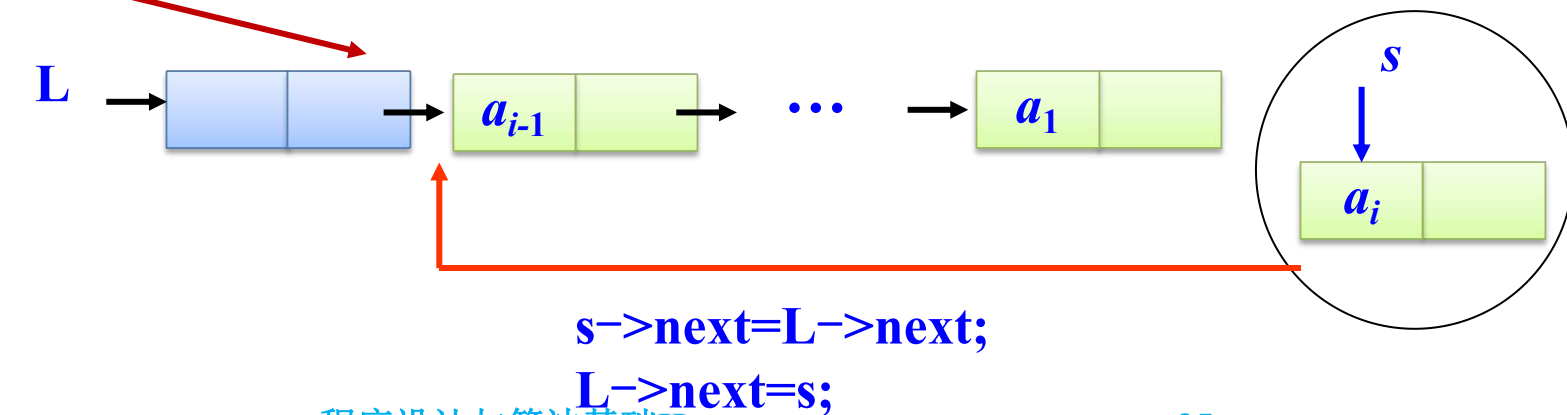
```
        s->data=data[i++];
```

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

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

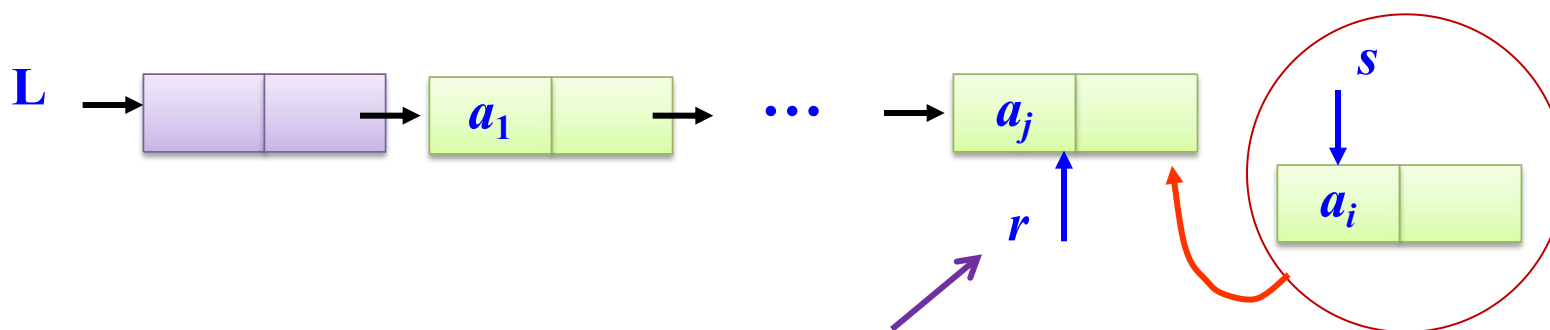
```
    }
```

```
}
```



## (2) 尾插法建表

- 从一个空表开始，创建一个头结点。
- 依次读取字符元素，生成新结点
- 将新结点插入到当前链表的表尾上，直到结束为止。



增加一个尾指针 $r$ ，使其始终指向当前链表的尾结点

**注意：**链表的结点顺序与逻辑次序相同。

## 尾插法建表算法如下：

```
void CreateFromTail(LinkList L)
```

```
{    Node *r, *s;
```

```
    int  flag =1;
```

```
    r=L;
```

```
    while(flag)
```

```
    {    c=getchar();
```

```
        if(c!='$')
```

```
        {
```

```
            s=(Node*)malloc(sizeof(Node));
```

```
            s->data=c;
```

```
            r->next=s;
```

```
            r=s;
```

```
        }
```

```
    } else
```

```
    {
```

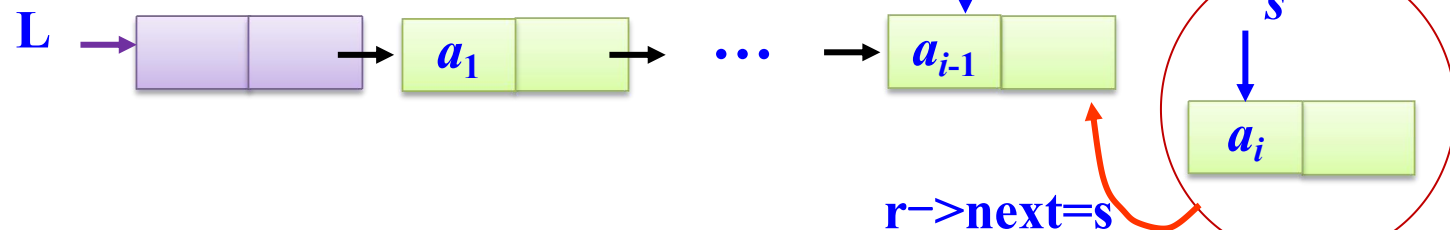
```
        flag=0;
```

```
        r->next=NULL;
```

```
    }
```

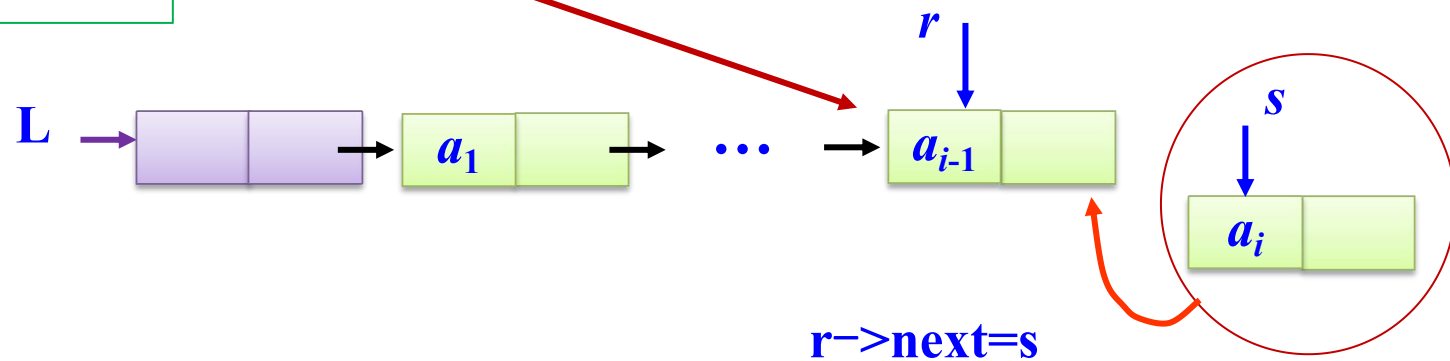
```
}
```

```
}
```



## 尾插法建表算法改进版

```
void CreateFromTail(LinkList L, ElemType data[], int n)
{
    Node *r, *s;
    int i=0;
    r=L;
    while(i<n)
    {
        s=(Node*)malloc(sizeof(Node));
        s->data = data[i++];
        s->next = r->next;
        r->next = s;
        r = s;
    }
}
```



### (3) 求线性表L中位置*i*的数据元素GetData (L, i)

**思路：**在带头单链表L中从头开始找到第*i*个结点，若存在第*i*个数据结点，则将返回该结点的指针。

```
Node* GetData(LinkList L, int i)
```

```
{
```

```
    int j=0;
```

```
    Node *p=L;  //p指向头结点，j置为0（即头结点的序号为0）
```

```
    if (i<=0) return NULL;
```

```
    while (j<i && p->next !=NULL)
```

```
    {    p=p->next;
```

```
        j++;
```

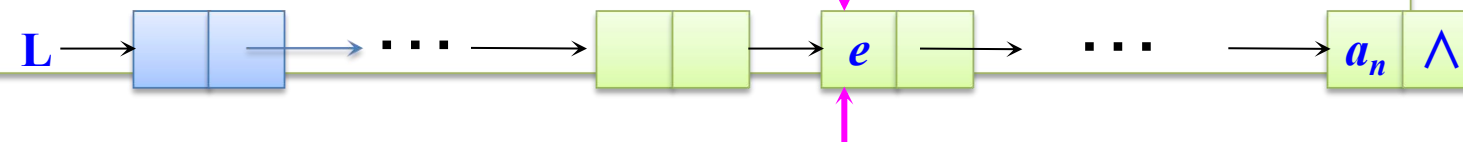
```
    }
```

```
    if (i==j) return p; //找第i个结点*p
```

```
    else return NULL;
```

```
}
```

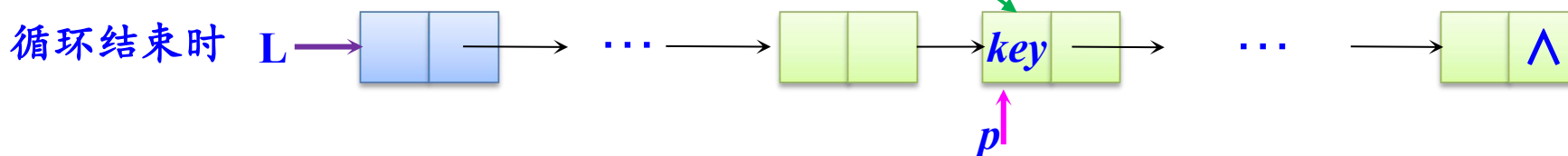
循环结束时



#### (4) 按元素值查找LocateElem(L, key)

**思路：**在单链表L中从头开始找第1个值域与key相等的结点，若存在这样的结点，则返回结点指针p，否则返回NULL。

```
Node* Locate(LinkList L, ElemType key)
{
    Node *p=L->next;    //p指向开始结点
    while (p!=NULL)
        if (p->data!=key)
            p=p->next;    //查找data值为key的结点
        else break;
    return p;
}
```



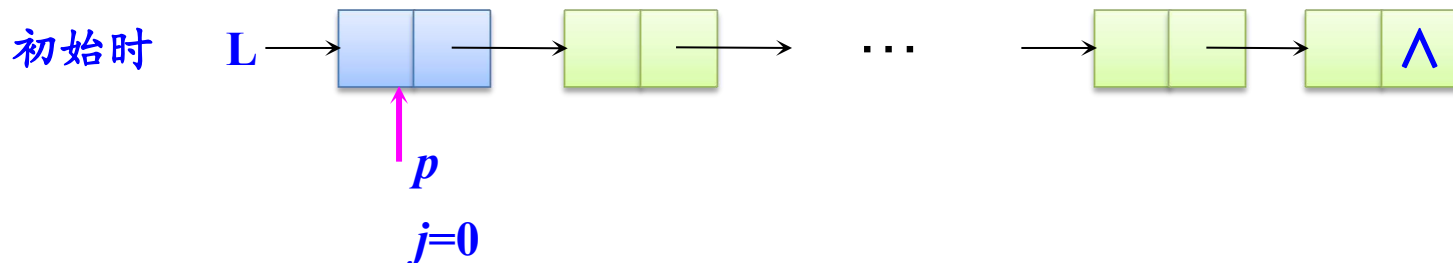
算法的时间复杂度为  $O(n)$   $\Rightarrow$  不具有随机存取特性



## (5) 求带头单链表的长度ListLength(L)

返回单链表L中数据结点的个数。

```
int ListLength(LinkList L)
{
    int j=0;
    Node *p=L->next;    //p指向头结点，n置为0（即头结点的序号为0）
    while (p!=NULL)
    {
        p=p->next;
        j++;
    }
    return j;
}
```



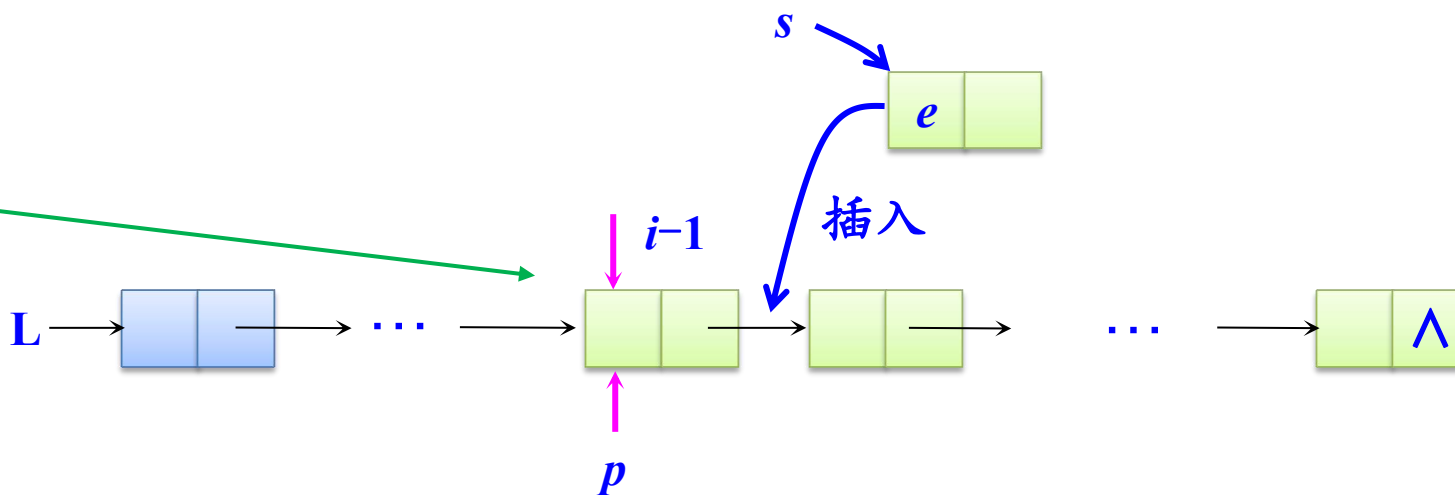
## (6) 插入数据元素 InsList(&L, i, e)

思路：先在单链表L中找到第*i*-1个结点\*p，若存在，将值为*e*的结点\*s插入到其后。

```
int InsList( LinkList L,int i,ElemType e)
```

```
{
    Node *pre,*s;
    int k;
    pre=L; k=0;
    while(pre!=NULL&& k<i-1)
    { pre=pre->next; k=k+1;}
    if(pre==NULL) {
        printf("插入位置不合理！ ");
        return -1;
    }
```

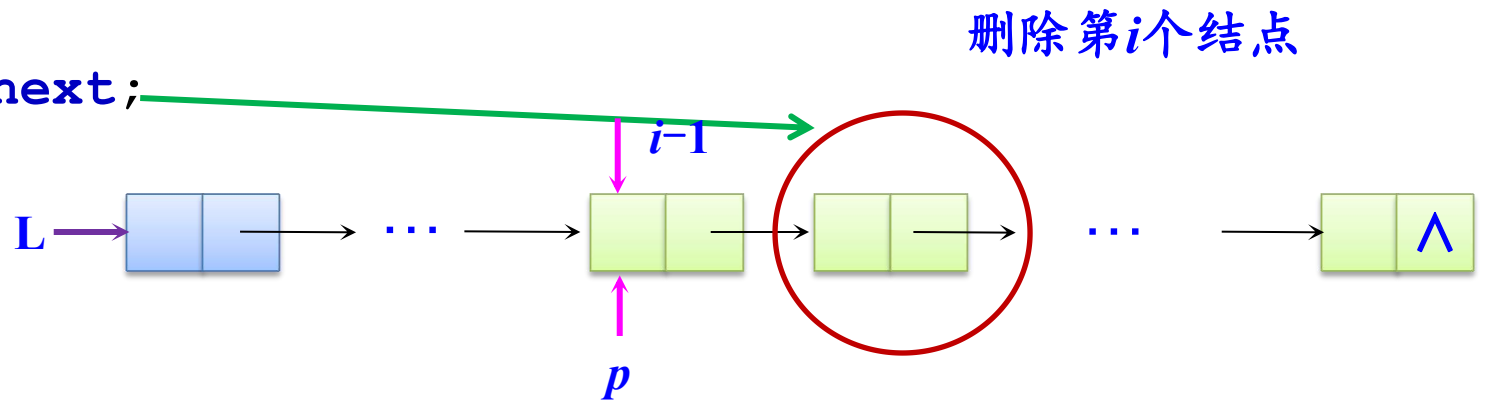
```
    s=(Node*)malloc(sizeof(Node));
    s->data=e;
    s->next=pre->next;
    pre->next=s;
    return 0;
}
```



## (7) 删除数据元素 DelList (L, i, \*e)

**思路：**在单链表L中找到第*i*-1个结点\*p，若存在，且存在后继结点，则删除该后继结点。

```
int DelList(LinkList L,int i,ElemType *e)
{ Node *pre,*r;
  int k;
  pre=L;k=0;
  while (pre->next!=NULL&&k<i-1) {
    pre=pre->next;
    k=k+1;
  }
  if (pre->next==NULL) {
    printf("删除结点的位置i不合理！");
    return -1;
  }
  r=pre->next;
  pre->next=pre->next->next;
  *e=r->data;
  free(r);
  return 0;
}
```



## (8) 假设有有序表LA和LB采用单链表存储，设计算法将它们合并成一个有序表单链表LC

LinkedList MergeLinkedList(LinkedList LA, LinkedList LB)

```
{
    LinkedList LC;
    Node *pa,*pb,*r;
    pa=LA->next;
    pb=LB->next;
    LC=LA;
    LC->next=NULL;r=LC;
    while(pa!=NULL&&pb!=NULL)
    {
        if(pa->data<=pb->data)
            {r->next=pa;r=pa;pa=pa->next;}
        else
            {r->next=pb;r=pb;pb=pb->next;}
    }
    if(pa) r->next=pa; //pa表中没有遍历完, pb表已遍历完
    else r->next=pb; //pb表中没有遍历完, pa表已遍历完
    free(LB);
    return(LC);
}
```

该算法的空间复杂度为 $O(1)$

下列单链表存储的有序表的合并算法，保留原有的链表LA和LB，新生成合并链表返回

```
LinkedList UnionList (LinkedList LA,LinkedList LB)
{
    Node *pa=LA->next, *pb=LB->next, *r, *s;
    LinkedList LC=(LinkedList )malloc(sizeof(Node)); //创建LC的头结点
    r=LC; //r始终指向LC的尾结点
    while (pa!=NULL && pb!=NULL)
    {
        if (pa->data<pb->data)
        {
            s=(LinkedList )malloc(sizeof(Node)); //复制结点
            s->data=pa->data;
            r->next=s;r=s; //采用尾插法将*s插入到LC中
            pa=pa->next;
        }
        else
        {
            s=(LinkedList )malloc(sizeof(Node));//复制结点
            s->data=pb->data;
            r->next=s;r=s; //采用尾插法将*s插入到LC中
            pb=pb->next;
        }
    }
}
```

```
while (pa!=NULL)
{
    s=(LinkedList)malloc(sizeof(Node)); //复制结点
    s->data=pa->data;
    r->next=s;r=s;                      //采用尾插法将*s插入到LC中
    pa=pa->next;
}
```

若LA没有扫描完，  
将余下结点复制到LC中

```
while (pb!=NULL)
{
    s=(LinkedList)malloc(sizeof(Node)); //复制结点
    s->data=pb->data;
    r->next=s;r=s;                      //采用尾插法将*s插入到LC中
    pb=pb->next;
}
r->next=NULL;                          //尾结点的next域置空
return LC;
```

若LB没有扫描完，  
将余下结点复制到LC中

```
}
```

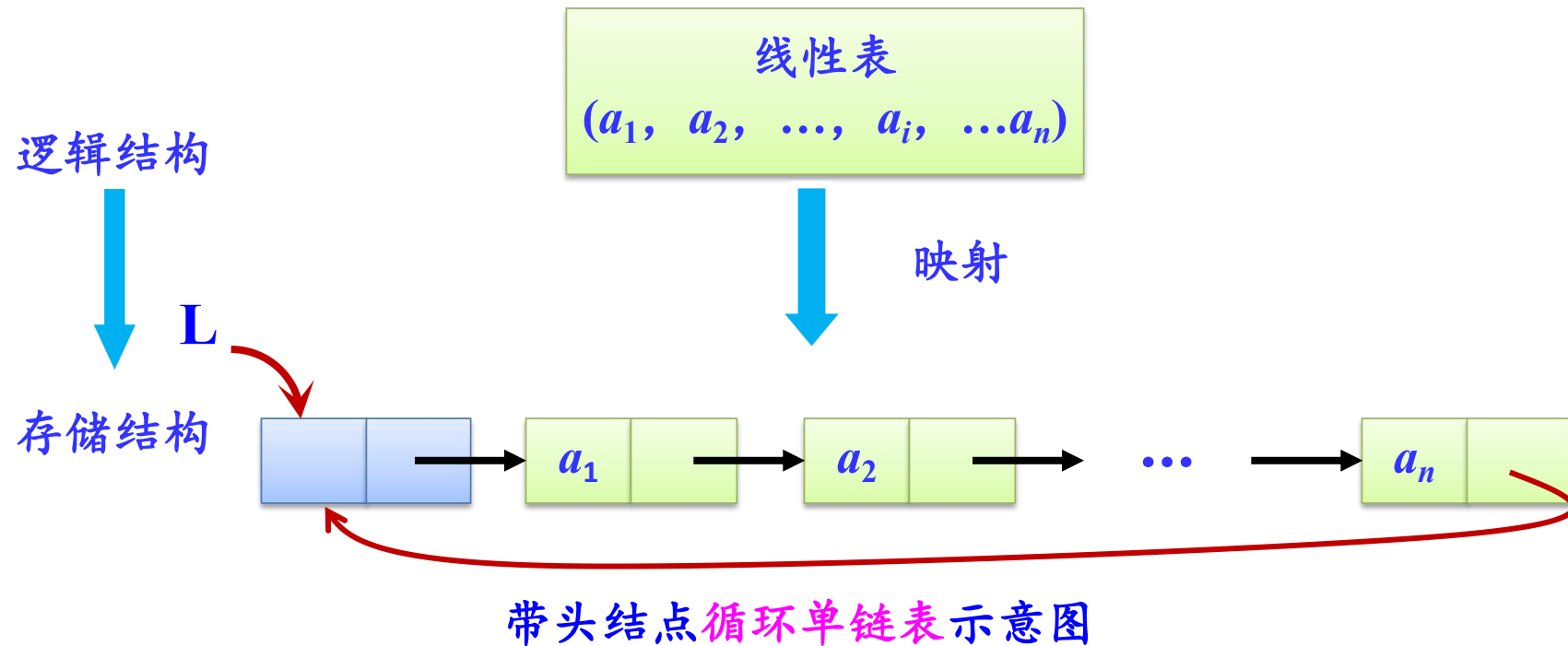
本算法的时间复杂度为 $O(m+n)$ ，空间复杂度为 $O(m+n)$ 。

## 2.3.3 循环单链表

循环单链表是另一种形式的链式存储结构形式。

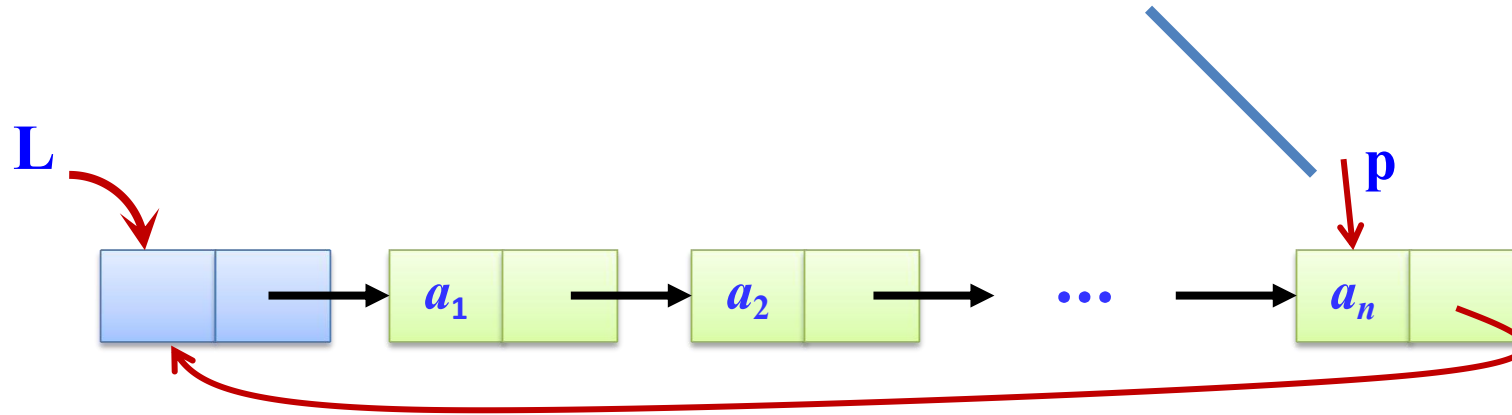
**循环单链表**：将表中尾结点的指针域改为指向表头结点，整个链表形成一个环。由此从表中任一结点出发均可找到链表中其他结点。

结点类型与非循环单链表的相同



与非循环单链表相比，循环单链表：

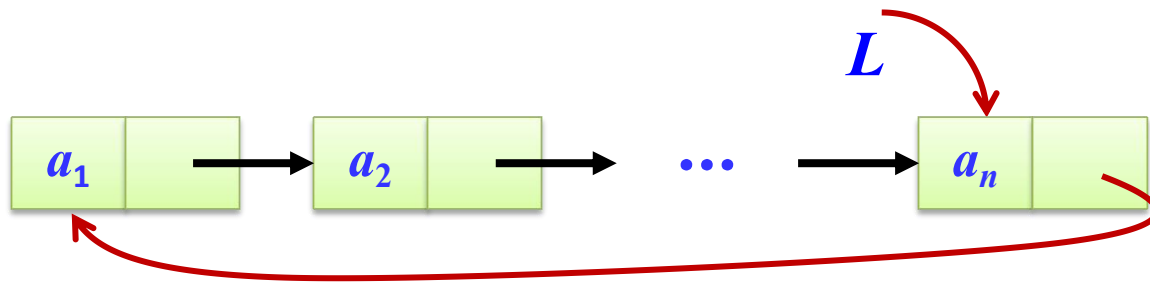
- 链表中没有空指针域
- $p$ 所指结点为尾结点的条件： $p \rightarrow \text{next} == L$





【例（补充）】某线性表最常用的操作是在尾元素之后插入一个元素和删除第一个元素，故采用\_\_\_\_\_存储方式最节省运算时间。

- A. 单链表
- B. 仅有头结点指针的循环单链表
- C. 双链表
- D. 仅有尾结点指针的循环单链表



- ✓ 在尾元素之后插入一个元素
- ✓ 删除第一个元素



时间复杂度均为 $O(1)$

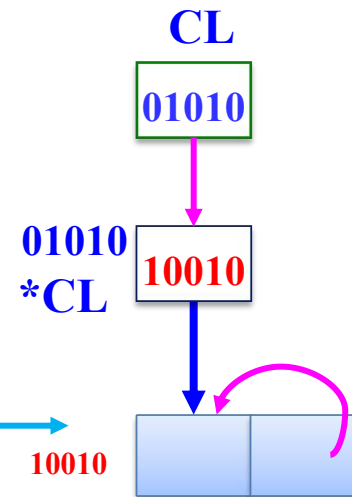
# 循环单链表的算法

循环单链表和普通单链表的结点类型LinkedList的定义相同:

```
typedef struct Node    //定义单链表结点类型
{
    ElemType data;
    struct Node *next;  //指向后继结点
} Node, *LinkedList;
```

## 1) 初始化循环单链表

```
void InitCLinkList (LinkedList *CL)
{
    *CL = (LinkedList) malloc(sizeof(Node));
    (*CL)->next = *CL;
}
```



# 循环单链表的算法

## 2) 创建循环单链表

```
void CreateCLinkList (LinkList CL)
```

```
{  Node *rear, *s;
```

```
  char c;
```

```
  rear = CL;
```

```
  c = getchar();
```

```
  while (c != '$') {
```

```
    s = (Node *) malloc(sizeof(Node));
```

```
    s->data = c;
```

```
    rear ->next = s ;
```

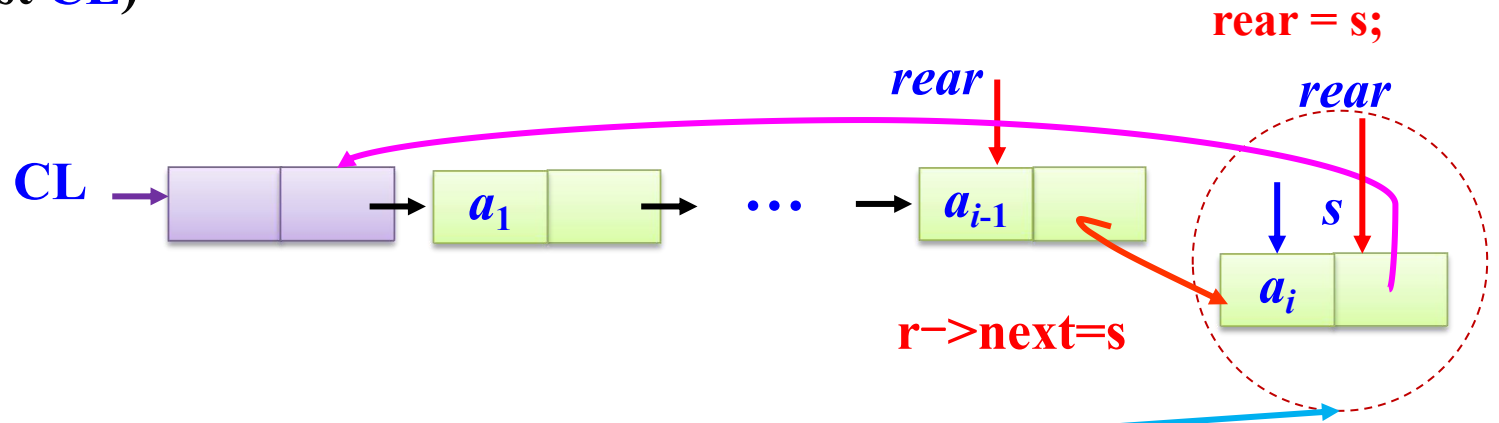
```
    rear = s;
```

```
    c = getchar();
```

```
  }
```

```
  rear->next = CL;
```

```
}
```



## 创建循环单链表的另一种实现

```
void CreateCLinkList (LinkList CL, ElemType data[], int n)
```

```
{  Node *rear, *s;
```

```
  int i=0;
```

```
  rear = CL;
```

```
  while ( i < n ) {
```

```
    s = (Node *) malloc(sizeof(Node));
```

```
    s->data = data[i];
```

```
    rear->next = s ;
```

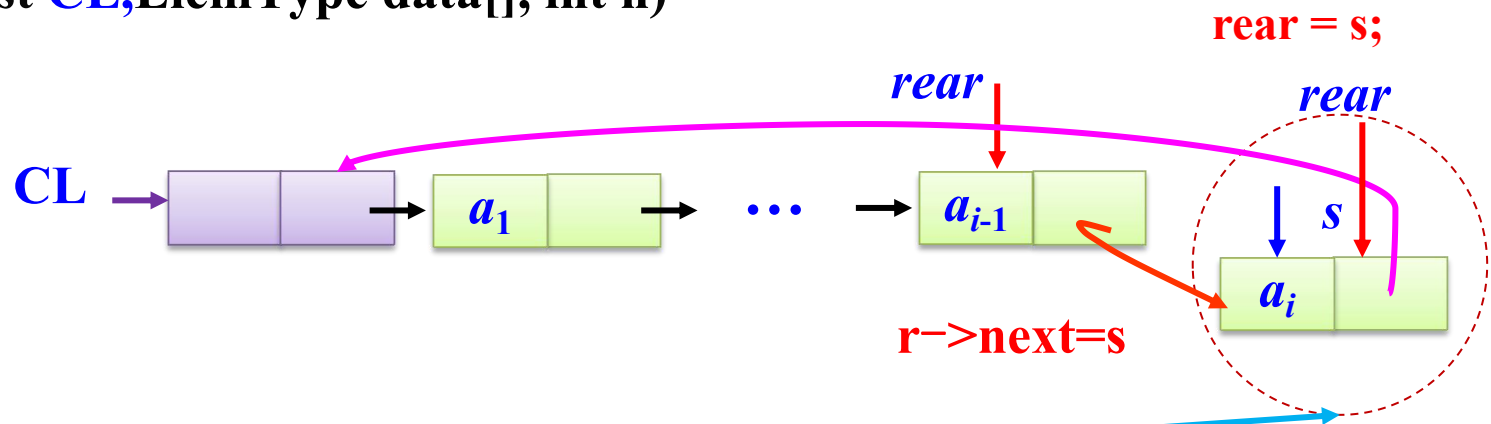
```
    rear = s;
```

```
    i++;
```

```
  }
```

```
  rear->next = CL;
```

```
}
```



# 循环单链表的算法

## 3) 循环单链表的合并 (传入两个循环单链表头指针)

LinkList merge1\_CLinkList (LinkList LA, LinkList LB)

```
{ Node *p, *q;
```

```
  p = LA;
```

```
  q = LB;
```

```
  char c;
```

```
  while (p->next != LA) p=p->next; //找尾结点
```

```
  while (q->next != LB) q=q->next; //找尾结点
```

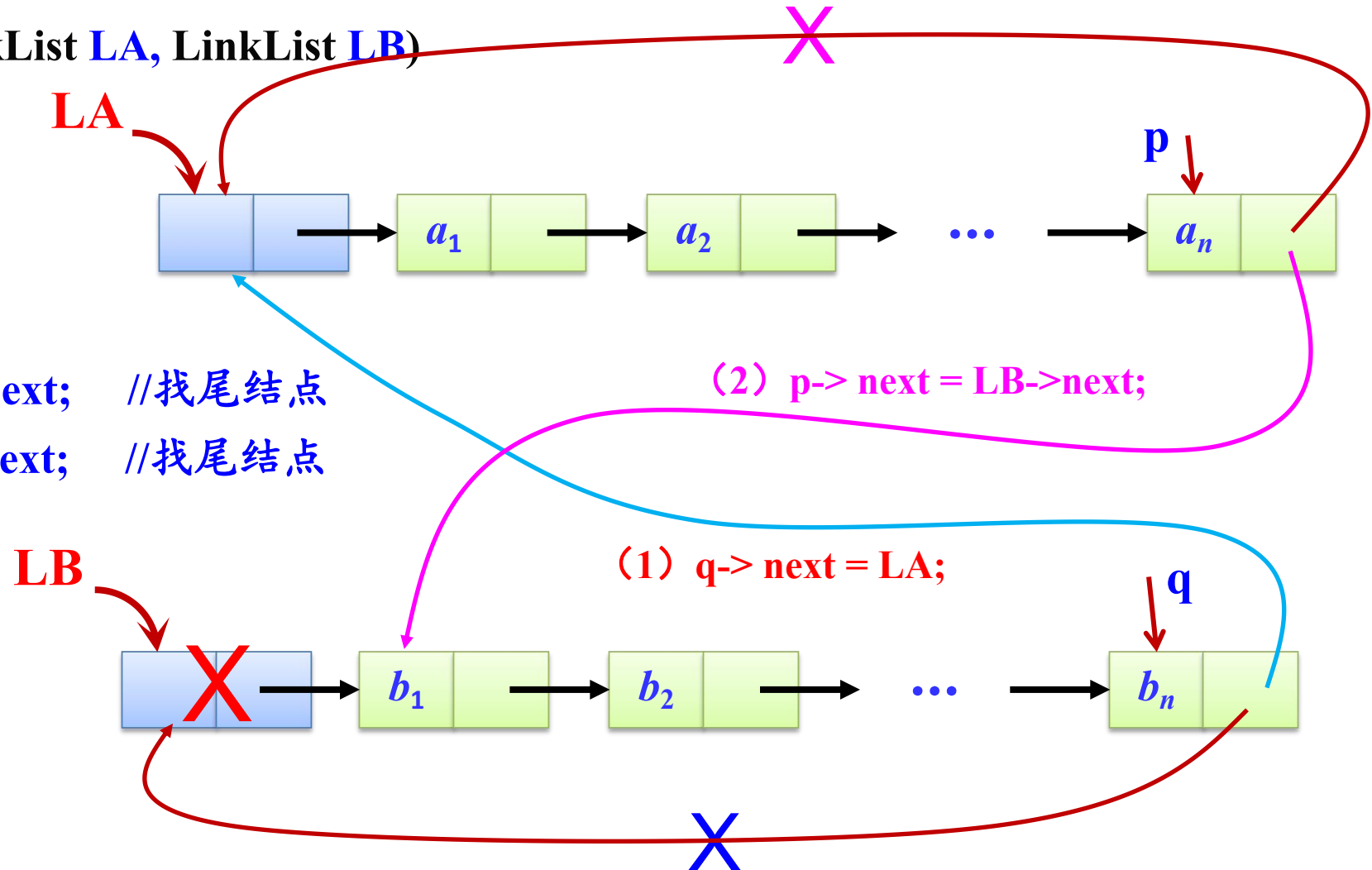
```
  q->next = LA;
```

```
  p->next = LB->next;
```

```
  free(LB);
```

```
  return(LA);
```

```
}
```



# 循环单链表的算法

## 3) 循环单链表的合并 (传入两个循环单链表的尾指针)

LinkList merge2\_CLinkList (LinkList RA, LinkList RB)

{ Node \*p;

**p = RA->next;**

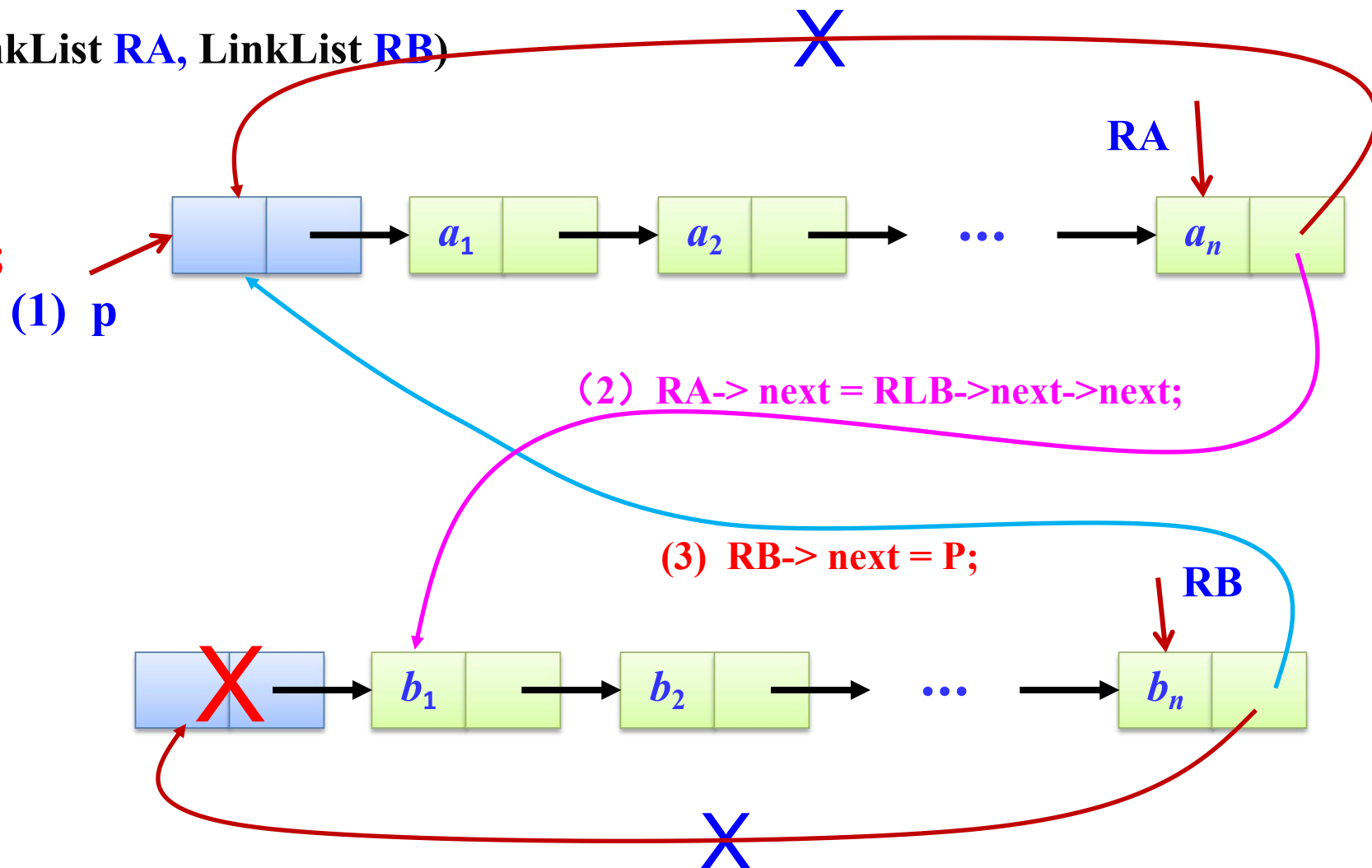
**RA->next = RB->next->next;**

free(RB->next);

RB->next = p;

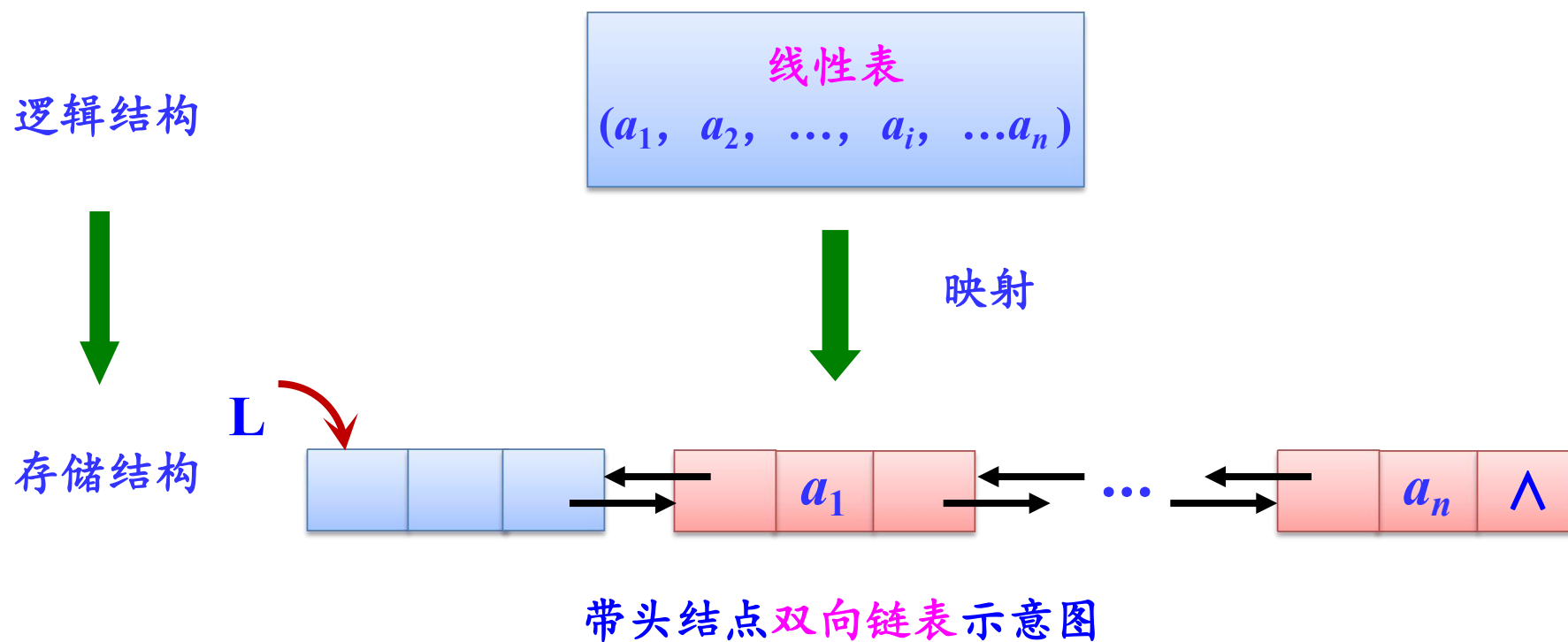
return(RB);

}



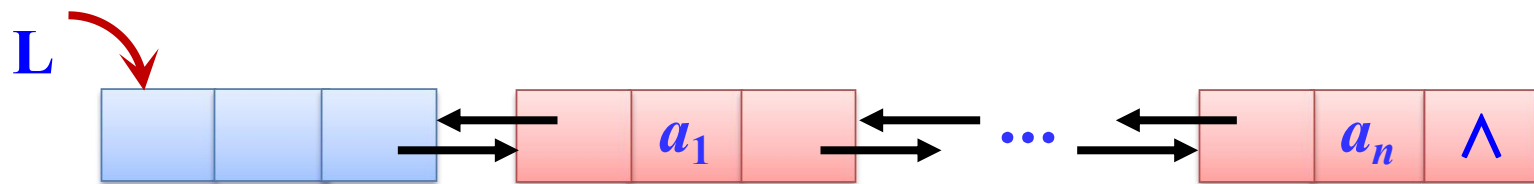
## 2.3.4 双向链表

在线性表的链式存储结构中，每个物理结点增加一个指向后继结点的指针域和一个指向前驱结点的指针域  $\Rightarrow$  双向链表。



## 双向链表的优点:

- 从任一结点出发可以快速找到其前驱结点和后继结点;
- 从任一结点出发可以访问其他结点。



对于双链表，采用类似于单链表的类型定义，其结点类型**DoubleList**定义如下：

```
typedef struct Dnode           //双链表结点类型
{
    ElemType data;
    struct Dnode *prior;
    struct Dnode *next;
} Dnode, *DoubleList;
```

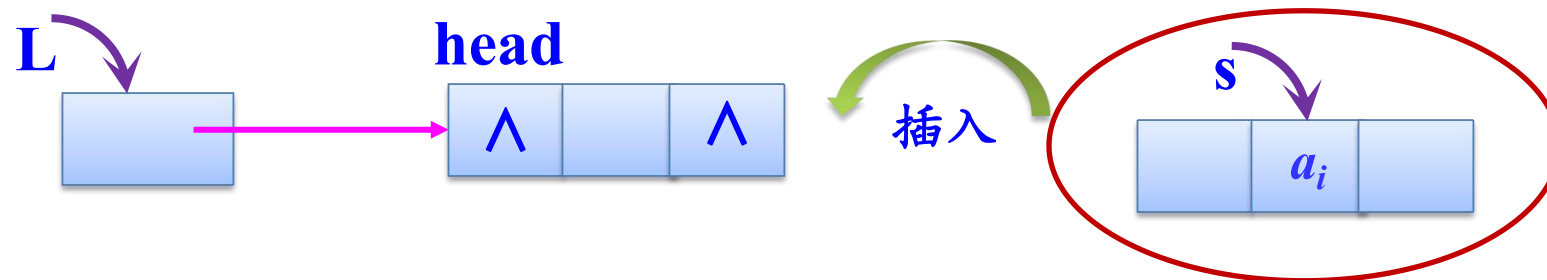




# 1、建立双向链表

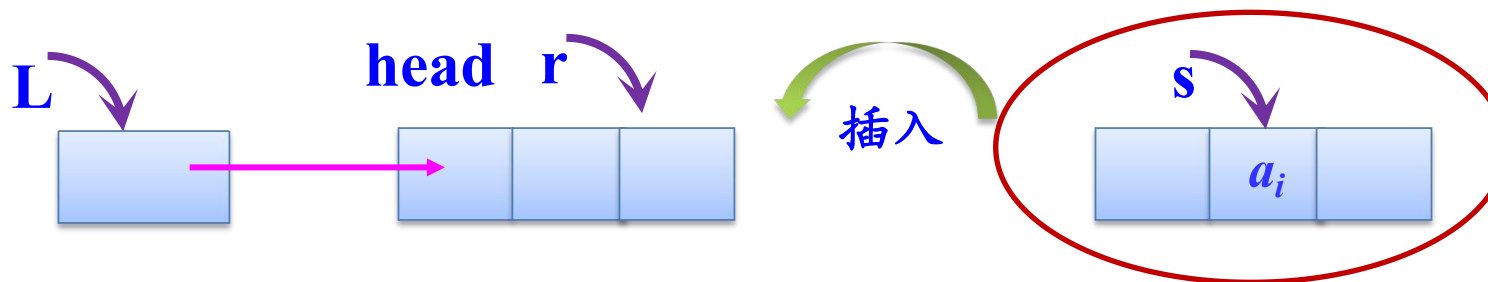
**头插法**建立双向链表：由含有 $n$ 个元素的数组 $a$ 创建带头结点的双向链表 $L$ 。

```
void CreateDListHead(DoubleList *L, ElemType a[], int n)
{   Dnode *s;int i;
    *L=(Dnode*)malloc(sizeof(Dnode)); //创建头结点
    (*L)->prior = (*L)->next=NULL;   //前后指针域置为NULL
    for (i=0;i<n;i++)                 //循环建立数据结点
    {   s=(Dnode *)malloc(sizeof(Dnode));
        s->data=a[i];                  //创建数据结点*s
        s->next = (*L)->next;         //将*s插入到头结点之后
        if ( (*L)->next!=NULL)        //若*L存在数据结点，修改前驱指针
            (*L)->next->prior=s;
        (*L)->next=s;
        s->prior = *L;
    }
}
```



**尾插法**建立双链表：由含有 $n$ 个元素的数组 $a$ 创建带头结点的双链表 $L$ 。

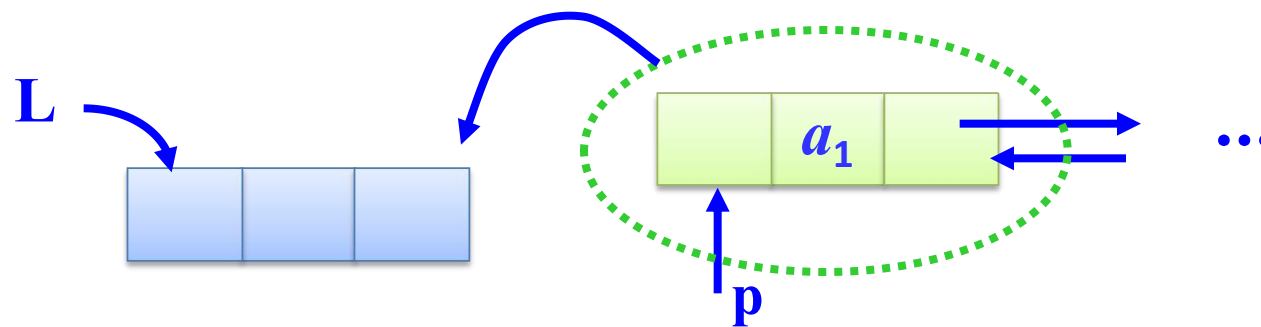
```
void CreateDListRear(DoubleList *L, ElemType a[], int n)
{   Dnode *s, *r;
    int i;
    (*L)=(Dnode *)malloc(sizeof(Dnode)); //创建头结点
    r=*L;                                //r始终指向尾结点, 开始时指向头结点
    for (i=0;i<n;i++)                    //循环建立数据结点
    {   s=(Dnode *)malloc(sizeof(Dnode));
        s->data=a[i];                    //创建数据结点*s
        r->next=s;s->prior=r;            //将*s插入*r之后
        r=s;                             //r指向尾结点
    }
    r->next=NULL;                        //尾结点next域置为NULL
}
```



**【例】** 有一个带头结点的双向链表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， $\cdots$ ，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(DoubleList L)
{
    Dnode *p=L->next, *q;
    L->next=NULL;
    while (p!=NULL)
    {
        q=p->next;

        p->next=L->next;
        if (L->next!=NULL)
            L->next->prior=p;
        L->next=p;
        p->prior=L;

        p=q;
    }
}

```

//双向链表结点逆置

//p指向开好结点

//构造只有头结点的双链表L

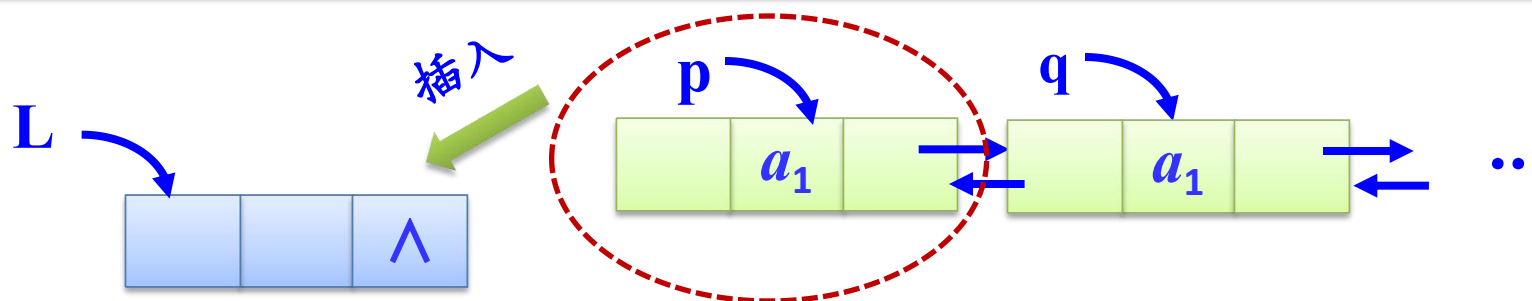
//扫描L的数据结点

//用q保存其后继结点

//采用头插法将\*p结点插入

//修改其前驱指针

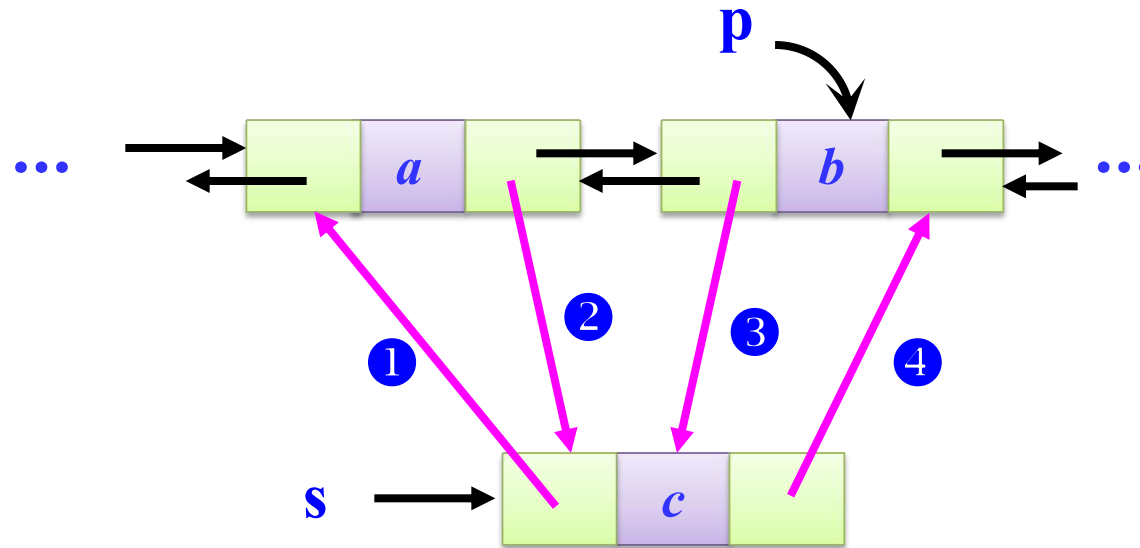
//让p重新指向其后继结点



## 2、双向链表中结点的插入和删除

### 双向链表插入结点的演示

在\*p结点之前插入结点\*s



操作语句：

- ①  $s \rightarrow \text{prior} = p \rightarrow \text{prior}$
- ②  $p \rightarrow \text{prior} \rightarrow \text{next} = s$
- ③  $p \rightarrow \text{prior} = s$
- ④  $s \rightarrow \text{next} = p$

插入完毕


请思考：在\*p结点之后插入结点\*s，如何操作？

# 线性表在双向链表中插入结点的实现

和单向链表相比，双向链表主要是插入运算不同。

## ① 双向链表的插入算法：

在双链表中，可以查找第 $i$ 个结点，并在它前面插入结点 $e$ 。

```
int ListInsert(DoubleList *L, int i, ElemType e)
{
    int j=0;
    Dnode *p=*L,*s,*q;           //p指向头结点，j设置为0
    while (j<i && p!=NULL)        //查找第i个数据结点
    {
        if (j==i-1) q=p;         //用q记住第i-1个结点
        j++;
        p=p->next;
    }
    
    查找第 $i$ 个结点 $*p$ 
}
```

```

if (j<i-1 )           //链表中小于i-1个结点，返回0
    return 0;
else                 //链表原有结点数大于等于i-1个
{
    s=(Dnode *)malloc(sizeof(Dnode));
    if (!s) return 0;
    s->data=e;        //创建新结点*s
    if (p!=NULL) {   //若存在第i个结点，则p不为NULL，则新结点*s插入在*p之前
        s->prior = p->prior; //在*p之前插入*s结点
        p->prior->next = s; //修改其前驱结点的后继指针
        p->prior=s;        //修改p的前驱指针
        s->next=p;
    }
    else { //若原链表只有i-1结点，新结点*s插入到最后
        q->next=s;
        s->prior=q;
        s->next=NULL;
    }
    return 1;
}
}

```

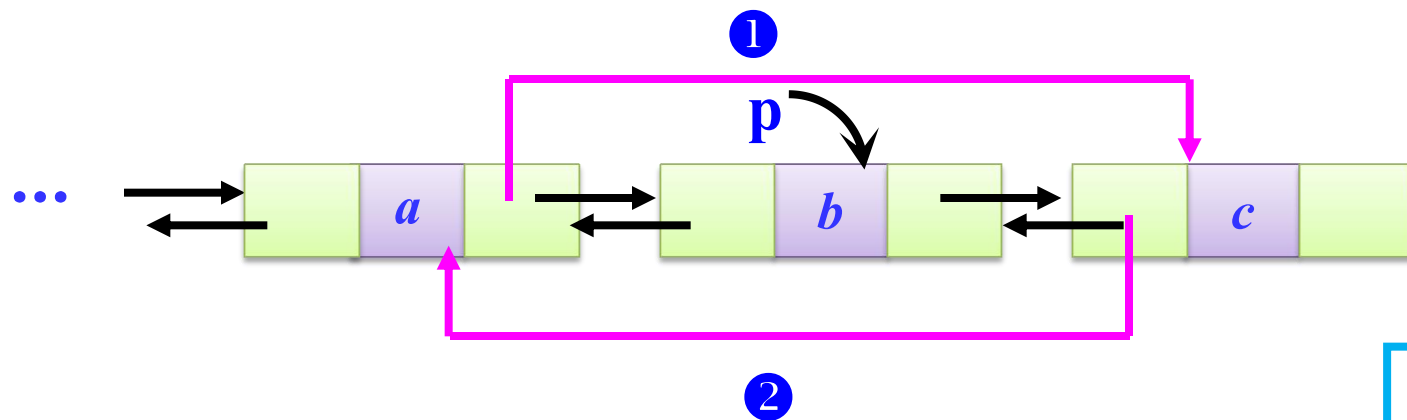
**另外解法：**  
在双链表中，可以查找第*i*-1个结点，并在它后面插入一个结点。

```
int ListInsert(DoubleList *L, int i, ElemType e)
{   int j=0;
    Dnode *p=*L,*s;                //p指向头结点，j设置为0
    while (j<i-1 && p!=NULL)        //查找第i-1个数据结点
    {   j++;
        p=p->next;
    }
    if (j<i-1 ) return 0;            //链表中小于i-1个结点，返回0
    else                             //链表原有结点数大于等于i-1个
    {   s=(Dnode *)malloc(sizeof(Dnode));
        if (!s) return 0;
        s->data=e;                  //创建新结点s
        s->prior = p;                //在p成为s结点的前驱
        s->next = p->next;           //p的后继成为s的后继
        p->next=s;                  //s成为p的后继
        if (s->next !=NULL) {       //若s的后继不为NULL
            s->next->prior=s;        //修改s的后继的前驱指针
        }
    }
    return 1;
}
```



## 双向链表删除结点的演示

删除\*p结点



操作语句：

①  $p \rightarrow \text{prior} \rightarrow \text{next} = p \rightarrow \text{next}$

②  $p \rightarrow \text{next} \rightarrow \text{prior} = p \rightarrow \text{prior}$

删除完毕

请思考：删除\*p结点之后（或者之前）的一个结点，如何删除？

## ② 双向链表的删除算法:

```
int ListDelete(DoubleList *L, int i, ElemType *e)
{   int j=0; Dnode *p=*L;           //p指向头结点, j设置为0
    while (j<i && p!=NULL)           //查找第i个结点
    {   j++;
        p=p->next;
    }
    if (j<i || p==NULL)               //未找到第i个结点
        return 0;
    else                               //找到第i个结点*p
    {   *e=p->data;
        p->prior->next=p->next;       //修改p的前驱结点的后继指针
        if (p->next!=NULL)            //p的后继指针不为空
            p->next->prior=p->prior;   //修改p的后继结点前驱指针
        free(p);                      //释放*q结点
        return 1;
    }
}
```

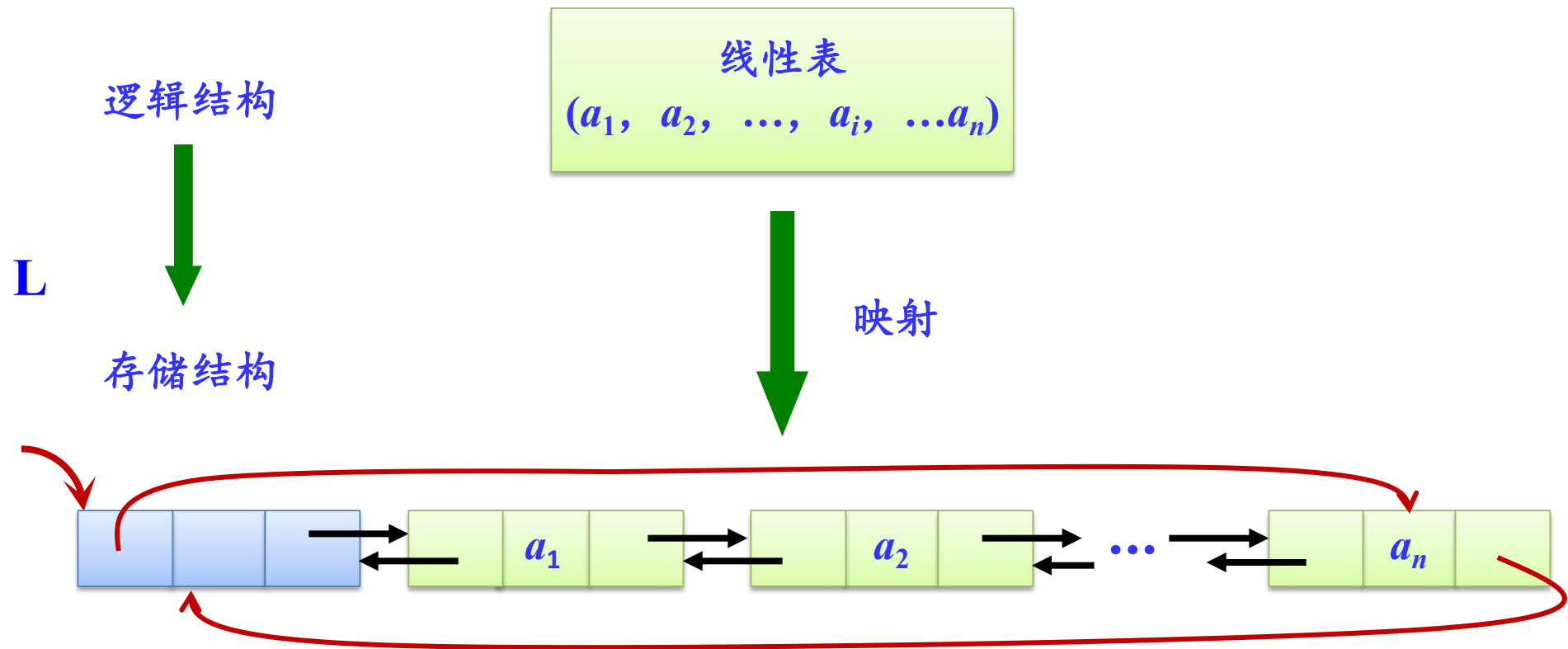
删除\*p结点并释放其空间

**另外解法:** 在双链表中, 可以查找第*i*-1个结点, 并将它后面结点删除。

## 思考题

一个带头结点的双向链表L（结点个数大于2），插入一个非尾结点的结点，需要修改\_\_\_\_\_个指针域，删除一个非尾结点的结点，需要修改\_\_\_\_\_个指针域。

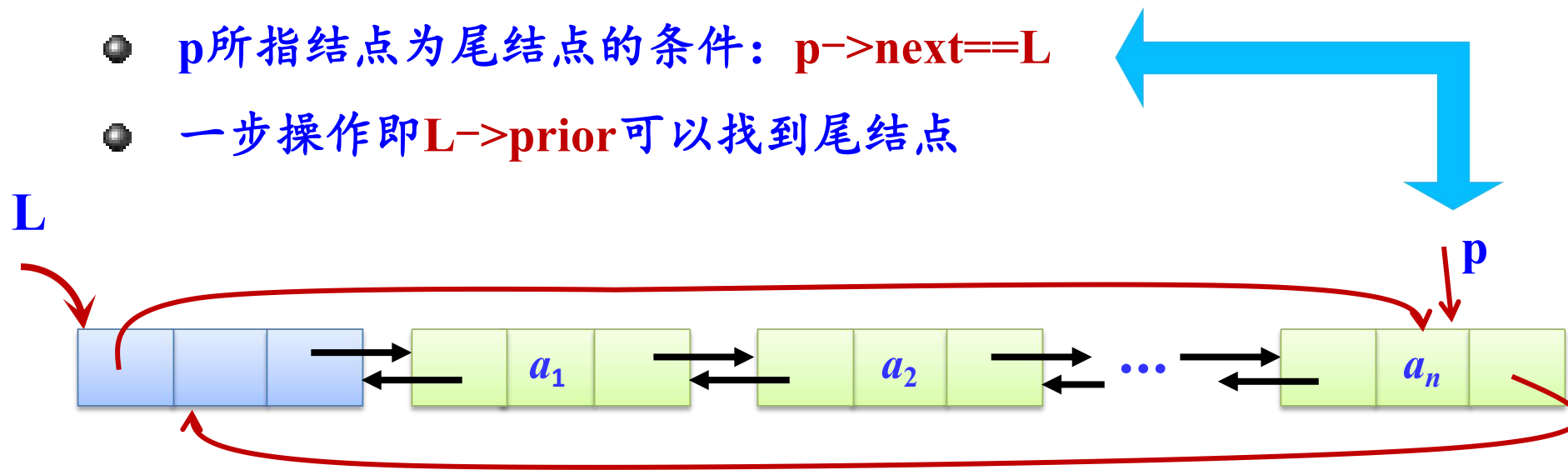
### 3、循环双向链表



带头结点循环双向链表示意图

与非循环双向链表相比，循环双向链表：

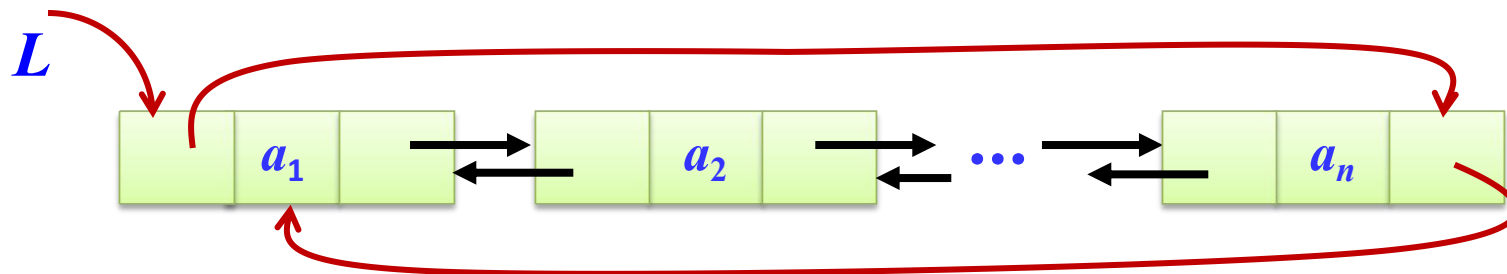
- 链表中没有空指针域
- $p$ 所指结点为尾结点的条件： $p \rightarrow \text{next} == L$
- 一步操作即 $L \rightarrow \text{prior}$ 可以找到尾结点



**【例】** 假设对含有 $n$  ( $n>1$ ) 个元素的线性表的运算只有4种，即删除第一个元素、删除尾元素、在第一个元素前面插入新元素、在尾元素的后面插入新元素，则最好使用\_\_\_\_\_。

- A.只有尾结点指针没有头结点的循环单链表
- B.只有尾结点指针没有头结点的非循环双链表
- C.只有首结点指针没有尾结点指针的循环双链表
- D.既有头指针也有尾指针的循环单链表

## 只有首结点指针没有尾结点指针的循环双链表



- 删除第一个元素
- 删除尾元素
- 在第一个元素前面插入新元素
- 在尾元素的后面插入新元素

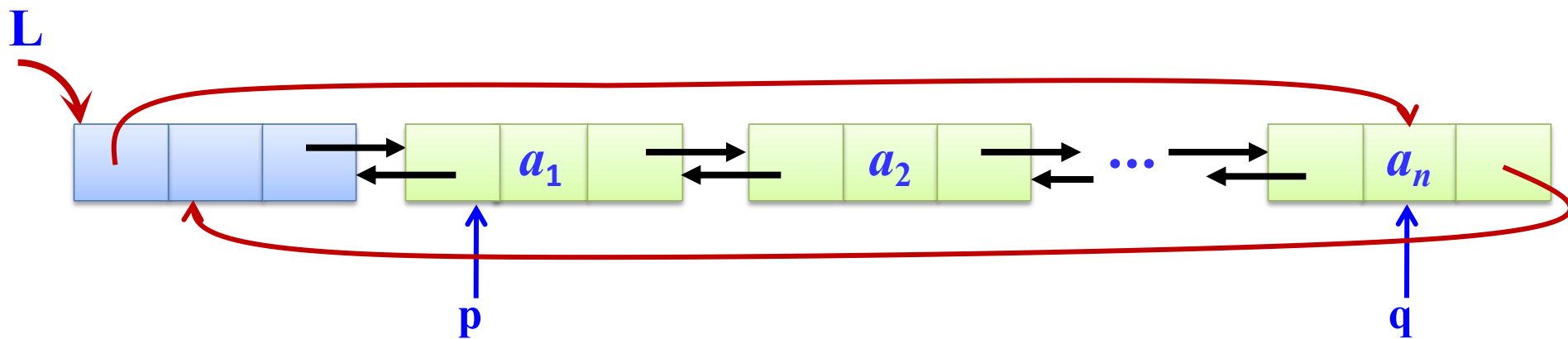


时间复杂度均为 $O(1)$

【例】设计判断带头结点的循环双链表L（含两个以上的结点）是否对称相等的算法。

### 算法思路

- p从左向右扫描L，q从右向左扫描L
- 若对应数据结点的data域不相等，则退出循环
- 否则继续比较，直到p与q相等或p的下一个结点为\*q为止。





① 数据结点为奇数的情况:

*a*    *b*    *c*    *b*    *a*



*p*   *q*

*p*=*q*: 结束

② 数据结点为偶数的情况:

*a*    *b*    *b*    *a*



*p*    *q*

*p*=*q*->*prior*: 结束

```

int Symmetry (DoublyList L)
{
    int same=1;
    Dnode *p=L->next;           //p指向第一个数据结点
    Dnode *q=L->prior;          //q指向最后数据结点
    while (same==1)
    {
        if (p->data!=q->data)
            same=0;
        else
        {
            if (p==q || p==q->prior) break;
            q=q->prior;           //q前移
            p=p->next;            //p后移
        }
    }
    return same;
}

```

**思考题：** 循环链表的作用？  
在什么情况下使用循环链表？

# 一元多项式的加法运算

## 1、一元多项式的运算

$$p_n(x) = p_0 + p_1x + p_2x^2 + \dots + p_nx^n$$

在计算机中，可以用一个线性表来表示：

$$P = (p_0, p_1, \dots, p_n)$$

但是对于形如

$$S(x) = 1 + 3x^{10000} - 2x^{20000}$$

的多项式，上述表示方法是否合适？

一般情况下的一元稀疏多项式可写成

$$P_n(x) = p_1x^{e_1} + p_2x^{e_2} + \dots + p_mx^{e_m}$$

其中：  $p_i$  是指数为  $e_i$  的项的非零系数，

$$0 \leq e_1 < e_2 < \dots < e_m = n$$

可以下列线性表表示：

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

例如：

$$P(x) = 7x^3 - 2x^{12} - 8x^{999}$$

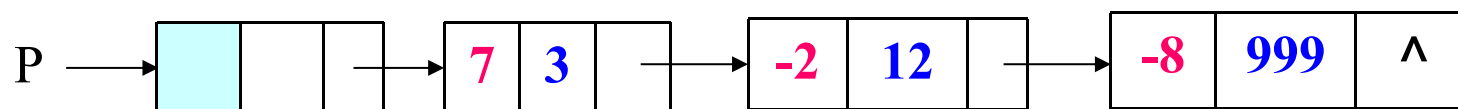
可用线性表可以表示为：  $((7, 3), (-2, 12), (-8, 999))$

## ✓ 单链表的结点定义

```
typedef struct Polynode
{
    int coef;
    int exp;
    struct Polynode *next;
}Polynode, *Polylist;
```

coef	exp	next
------	-----	------

例如:  $P(x) = 7x^3 - 2x^{12} - 8x^{999}$



# 创建一元多项式链式存储的算法

【算法思想】通过键盘输入多项式的系数和指数，用尾插法建立一元多项式的链表。以输入系数0为结束标志，约定建立多项式链表时，总是按指数从小到大的顺序排列。

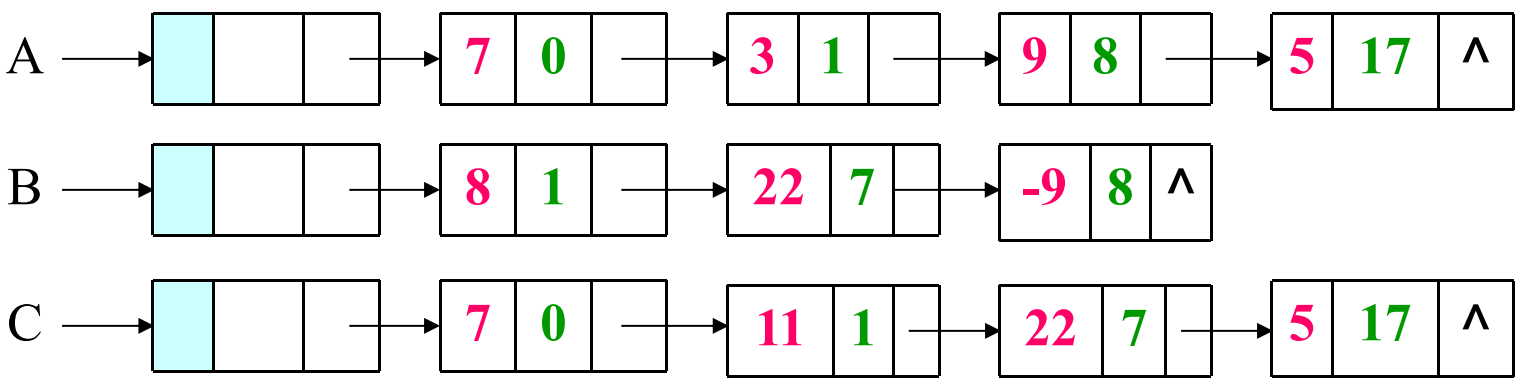
```
Polylist polycreate()
{ Polynode *head, *rear, *s;
  int c,e;
  rear=head=(Polynode *)malloc(sizeof(Polynode)); //建立头结点， rear 指向单链表的尾结点
  scanf("%d,%d",&c,&e); //键入多项式的系数和指数项
  while(c!=0) //若c=0，则代表多项式的输入结束
  { s=(Polynode *)malloc(sizeof(Polynode)); //申请新的结点
    s->coef=c ; s->exp=e ; rear->next=s ; //在当前表尾做插入
    rear=s;
    scanf("%d,%d",&c,&e);
  }
  rear->next=NULL; //将表的最后一个结点的next置NULL，以示表结束
  return(head);
}
```

# 两个一元多项式相加

$$A(x) = 7 + 3x + 9x^8 + 5x^{17}$$

$$B(x) = 8x + 22x^7 - 9x^8$$

$$C(x) = A(x) + B(x) = 7 + 11x + 22x^7 + 5x^{17}$$



两个一元多项式polya和polyb相加，存储在新的链表中，返回头指针head

**Polynode\* PolyAdd( Polylist polya, Polylist polyb)** //将多项式a与b相加，结果存在第三个链表中

```
{  Polynode *p,*q, *head , *L,*s;
    p = polya->next;
    q = polyb->next;
    head=(Polynode *)malloc(sizeof(Polynode));
    head->next=NULL;
    L=head;
    while(p&&q)                //遍历两表，根据情况判断表的多项式项，建立新的链表来存取内容
    {
        if(p->exp > q->exp)    //比较指数大小，将指数和系数较小的存入新表
        {
            s=(Polynode *)malloc(sizeof(Polynode));
            L->next=s;
            s->coef=q->coef;
            s->exp=q->exp;
            L=s;
            q=q->next;
        }
    }
```



```

if(p->exp < q->exp)    //比较指数大小，将指数和系数较小的存入新表
{
    s=(Polynode *)malloc(sizeof(Polynode));
    L->next=s;
    s->coef=p->coef;
    s->exp=p->exp;
    L=s;
    p=p->next;
}
if(p->exp == q->exp)    //相等可以直接相加
{
    if(p->coef + q->coef !=0) {
        s=(Polynode *)malloc(sizeof(Polynode));
        L->next=s;
        s->coef=p->coef + q->coef;
        s->exp=p->exp;
        L=s;
    }
    p=p->next;
    q=q->next;
}
} //当至少遍历了其中一个多项式，结束while循环

```

```
while(p) //p有剩余
```

```
{
```

```
    s=(Polynode *)malloc(sizeof(Polynode));
```

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

```
    s->coef=p->coef;
```

```
    s->exp=p->exp;
```

```
    L=s;
```

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

```
}
```

```
while(q) //q有剩余
```

```
{
```

```
    s=(Polynode *)malloc(sizeof(Polynode));
```

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

```
    s->coef=q->coef;
```

```
    s->exp=q->exp;
```

```
    L=s;
```

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

```
}
```

```
L->next=NULL;
```

```
return head;
```

```
}
```

```
//算法结束
```

**例2.6:** 设计一个高效的算法，从**顺序表L**中删除所有值为x的元素，并要求算法的时间复杂度为 **$O(n)$** ，空间复杂度为 **$O(1)$** 。

```
void delx(SeqList *L, ElemType x)
{
    i=0;
    while(i<=L->last&&L->elem[i]!=x) i++;
    j=i+1;
    while(j<=L->last)
        if(L->elem[j]!=x) {
            L->elem[i]=L->elem[j];
            i++;j++;
        }
        else j++;
    L->last=i-1;
}
```

## 例2.7：算法实现带头结点单链表的就地逆置问题。

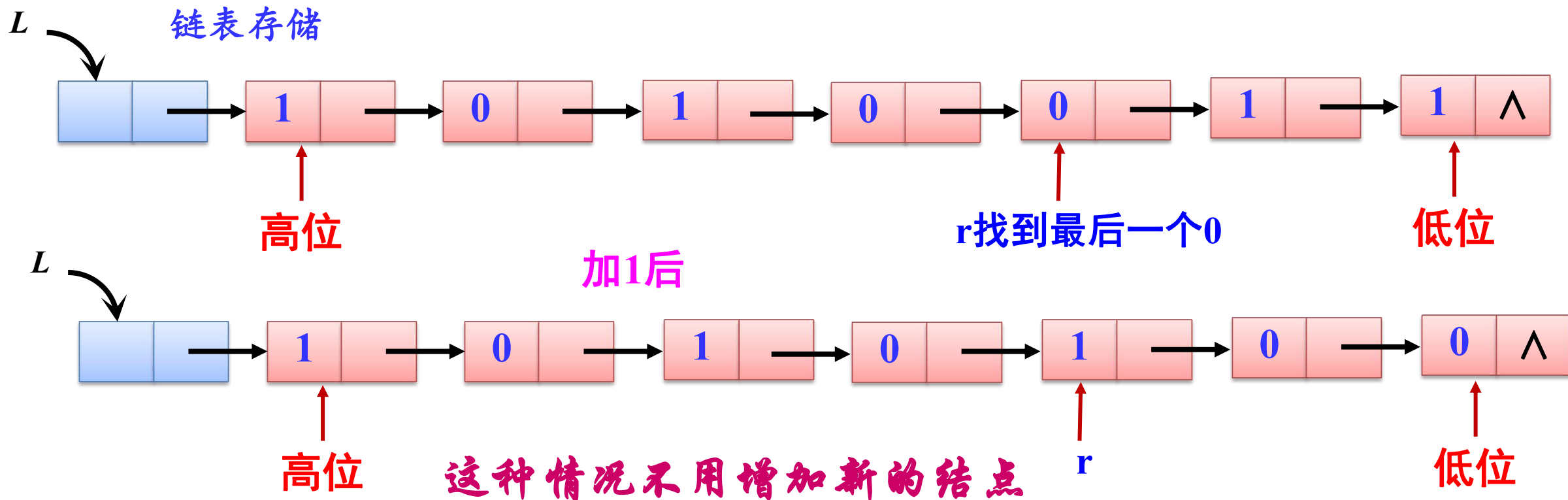
```
void ReverseList(LinkList L)  //逆置带头结点的单链表
{
    LinkList p=L->next, q=NULL;           // P为原链表的当前处理结点
    L->next=NULL;                          //逆置单链表初始为空表
    while(p!=NULL)                         //当原链表未处理完
    {
        q=p->next;                          // q指针保留原链表当前处理结点的下一个结点
        p->next=L->next;
        L->next=p;                          //将当前处理结点p插入到逆置表L的表头
        p=q;                               // p指向下一个待插入的结点
    } /*end of while*/
} /*end of ReverseList*/
```

**例2.8:** 已知带头结点单链表L，设计算法实现：以表中第一元素作为标准，将表中所有值小于第一个元素的结点均放在第一结点之前，所有值大于第一元素的结点均放在第一元素结点之后。

```
int changelist(LinkList L)
{ if (L->next==NULL) return 0;
  p1=L->next; // p1指向表中第一元素
  pre=p1;
  p=p1->next;
  while (p) /*顺次从p开始取结点，比p1->data小的插在头结点之后，
            比p1->data大的结点不作处理，继续检测其后继结点*/
  { q=p->next;
    if(p->data>=p1->data){ pre=p; p=q;}
    else{ pre->next=p->next;
          p->next=L->next;
          L->next=p; p=q ;} //当小于第1个结点，进行头插
    }
}
```

**例2.9：**建立一个带头结点的线性链表，用以存放输入的二进制数，链表中每个结点的data域存放一个二进制位。并在此链表上实现对二进制数加1的运算。

**【问题分析】**     **1010011**



请思考什么时候需  
要创建新结点？

只有所有位都为1时，  
加1后需要创建新结点

## 例2.9：程序代码

```
void BinAdd(LinkList L) /*用带头结点的单链表L存储二进制数，实现加1运算*/
{
    Node *q,*r,*temp,*s;
    q=L->next;
    r=L;
    while(q!=NULL) /*查找最后一个值域为0的结点*/
    {
        if(q->data == 0)
            r = q;
        q = q->next;
    }
}
```

## 例2.9的程序：（续）

```
if (r != L)
    r->data = 1;    //将最后一个值域为0的结点的值域赋为1
else                //未找到值域为0的结点
{
    temp = r->next;
    s=(Node*)malloc(sizeof(Node)); //申请新结点
    s->data=1;                //值域赋为1
    s->next=temp;
    r->next = s;              //插入到头结点之后
    r = s;
}
r = r->next;
while(r!=NULL)        //将后面的所有结点的值域赋为0
{
    r->data = 0;
    r = r->next;
}
} //BinAdd结束
```



**例：**假设线性表采用顺序表存储，编程实现删除所有大于等于x且小于等于y的元素。

```
void del_x2y(SeqList* L, ElemType x, ElemType y)
{
    int i=0, j=L->last;
    while (i<j)
    {
        if ((L->elem[i] < x) || (L->elem[i] > y)) i++; //从左向右找到需要删除的数据的位置i

        if ((L->elem[j] >= x) && (L->elem[j] <= y)) j--; //从右向左找到不需要删除的数据的位置j

        if ((L->elem[i] >= x) && (L->elem[i] <= y) && (L->elem[j] < x) && (L->elem[j] > y)){

            L->elem[i]=L->elem[j]; //将右边不需要删除的数据写入位置i
            i++;
            j--;
        }

        L->Last=j;
        return;
    }
}
```

本章结束