

知识回顾

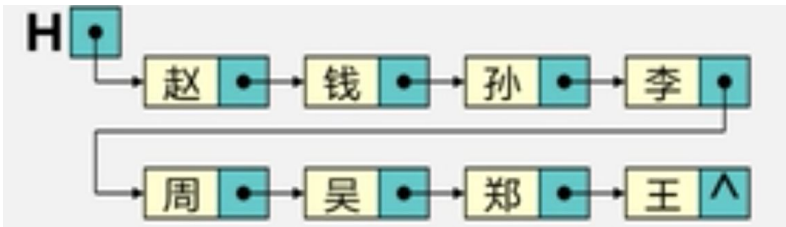
- 顺序表的特点:以物理位置相邻表示逻辑关系;
 - 顺序表的优点:任一元素均可随机存取;
 - 顺序表的缺点:进行插入和删除操作时,需移动大量的,存储空间不灵活;
 - 链式表示:用一组物理位置任意的存储单元来存放线性表的数据元素。
- 这组存储单元既可以是连续的,也可以是不连续的,甚至是零散分布在内存中的任意位置。
- 例:线性表:赵, 钱, 孙, 李, 周, 吴, 郑, 王)

顺序表

存储地址	存储状态
0031	赵
0033	钱
0035	孙
0037	李
0039	周
0041	吴
0043	郑
0045	王

链式表

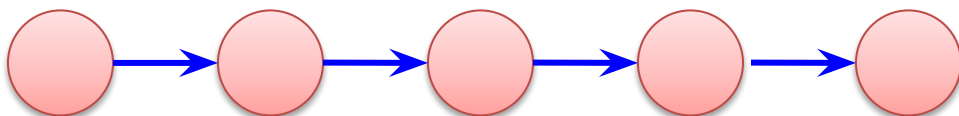
存储地址	存储状态
0001	李 0043
0007	钱 0013
0013	孙 0001
0019	王 NULL
0025	吴 0037
0031	赵 0007
0037	郑 0019
0043	周 0025



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。



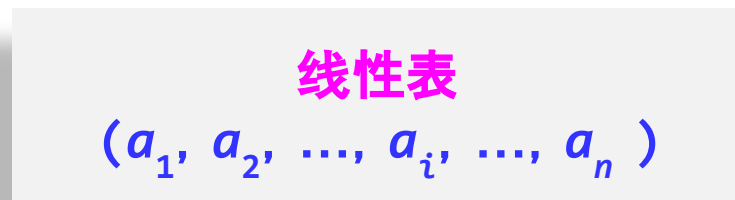
设计链式存储结构时，每个逻辑结点存储单独存储，为了表示逻辑关系，增加**指针域**。

数据域	指针域
-----	-----

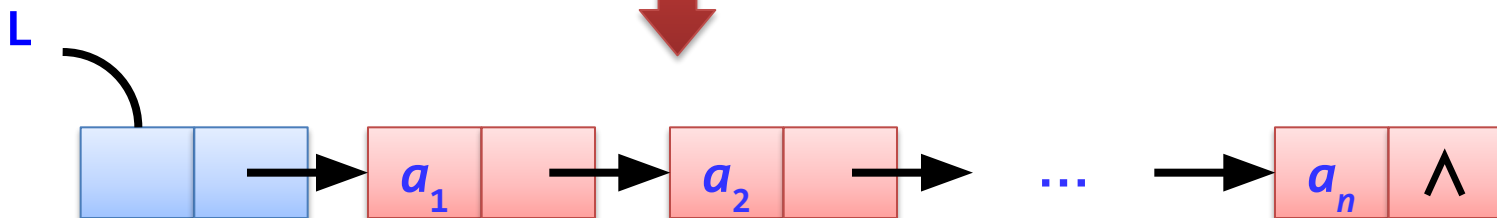
- 每个物理结点增加一个指向后继结点的指针域 ⇨ **单链表**。
- 每个物理结点增加一个指向后继结点的指针域和一个指向前驱结点的指针域 ⇨ **双链表**。

逻辑结构

存储结构



映射



带头结点单链表示意图

2.3.2 单链表

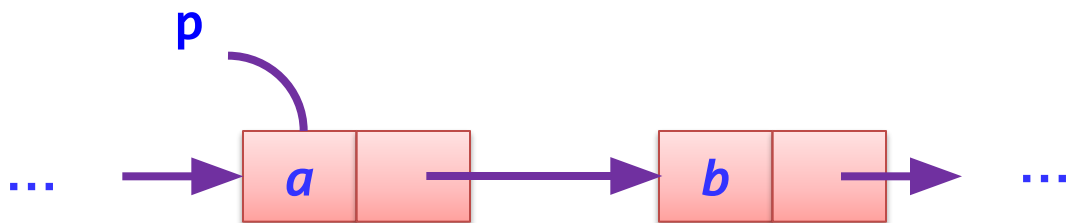
单链表中结点类型LinkNode的定义如下：

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



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。



2、建立单链表

先考虑如何整体建立单链表。

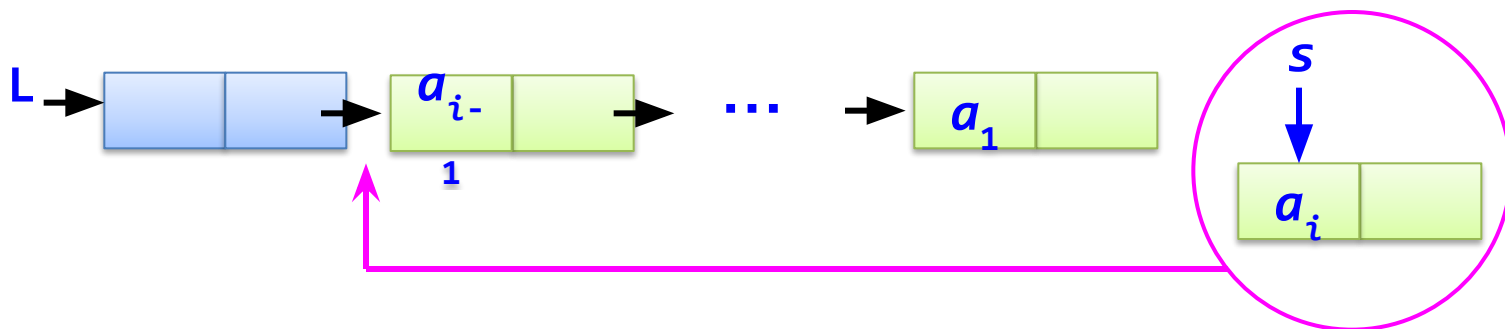


建立单链表的常用方法有两种。

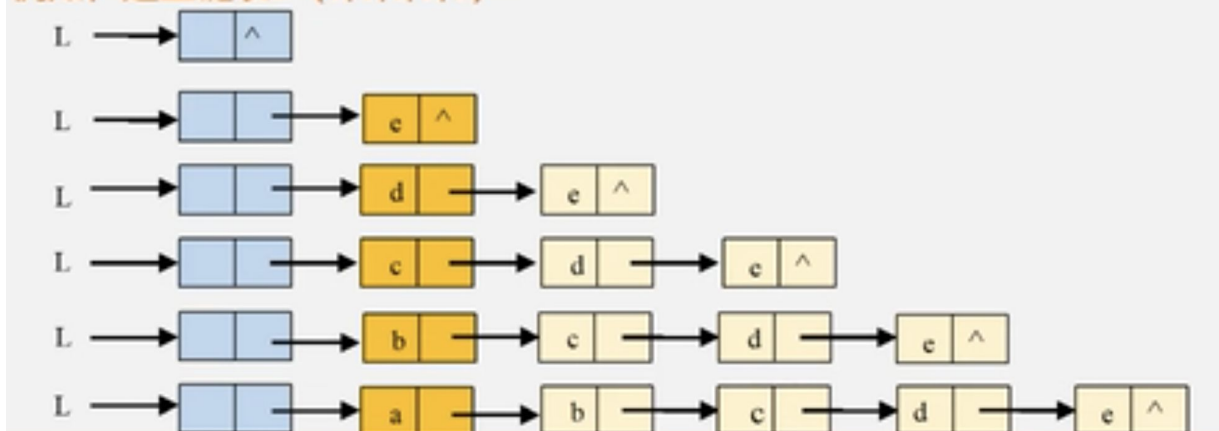
(1) 头插法建表

注意：链表的结点顺序与逻辑次序相反。

- 从一个空表开始，创建一个头结点。
- 依次读取字符数组 a 中的元素，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。



例如，建立链表 L (a,b,c,d,e)



头插法建表算法如下：

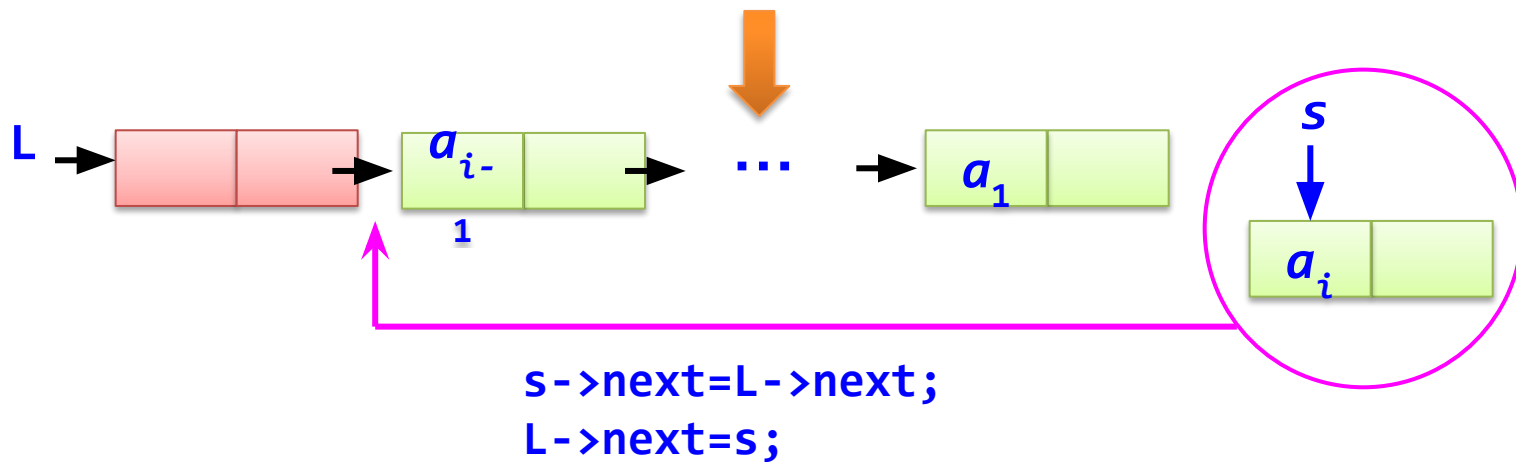
```
void CreateListF(LinkNode *&L, ElemType a[], int n)
{  LinkNode *s;
   int i;
   L=(LinkNode *)malloc(sizeof(LinkNode));
   L->next=NULL;      //创建头结点, 其next域置为NULL
```




```

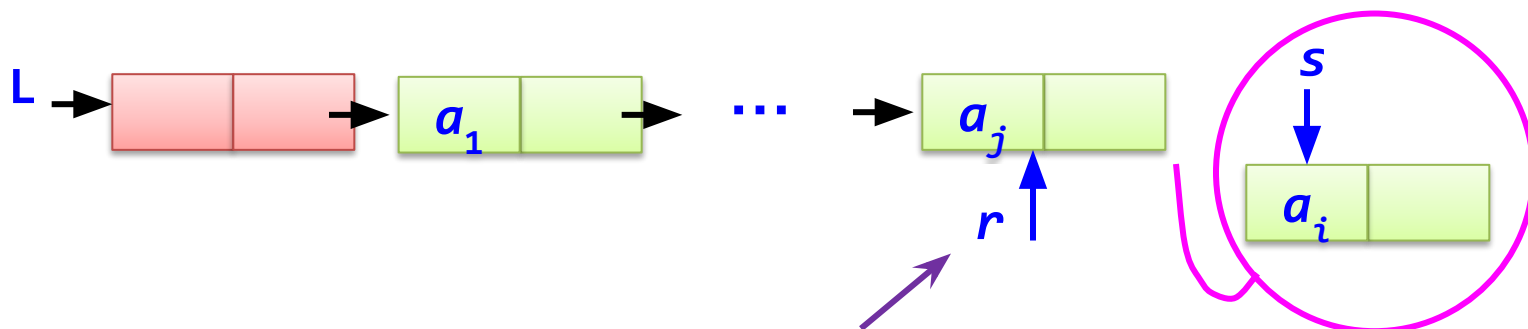
for (i=0;i<n;i++)           //循环建立数据结点
{
    s=(LinkNode *)malloc(sizeof(LinkNode));
    s->data=a[i];           //创建数据结点s
    s->next=L->next;        //将s插在原开始结点之前，头结点之后
    L->next=s;
}
}

```



(2) 尾插法建表

- 从一个空表开始, 创建一个头结点。
- 依次读取字符数组 a 中的元素, 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。

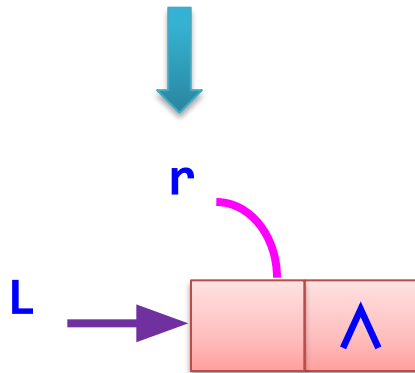


增加一个尾指针 r , 使其始终指向当前链表的尾结点

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：

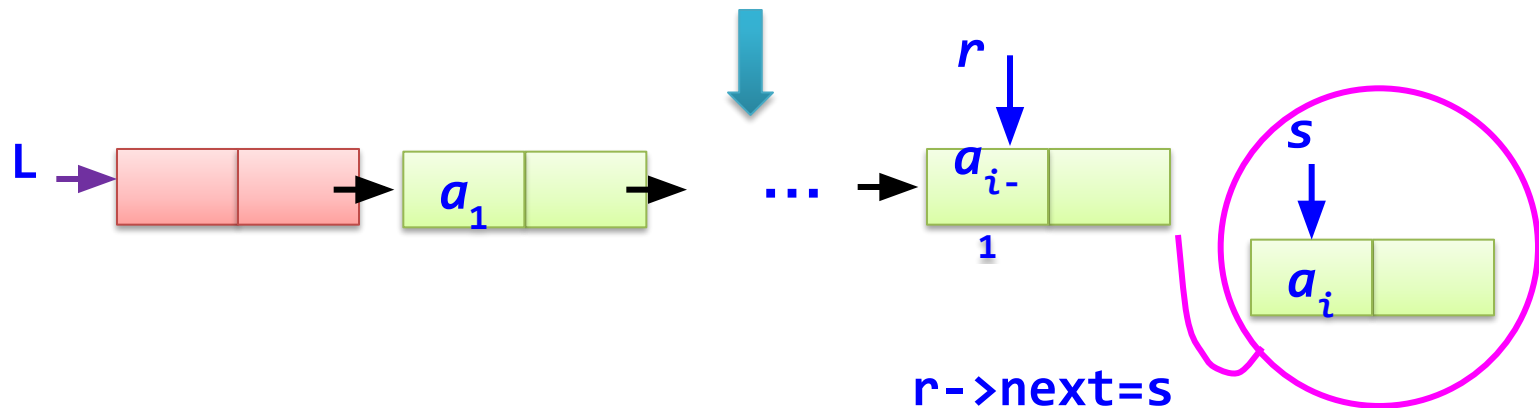
```
void CreateListR(LinkNode *&L, ElemType a[], int n)
{  LinkNode *s, *r;
   int i;
   L=(LinkNode *)malloc(sizeof(LinkNode));  //创建头结点
   r=L;                                     //r始终指向尾结点, 开始时指向头结点
```



```

for (i=0;i<n;i++)      //循环建立数据结点
{
    s=(LinkNode *)malloc(sizeof(LinkNode));
    s->data=a[i];      //创建数据结点s
    r->next=s;         //将s插入r之后
    r=s;
}
r->next=NULL;         //尾结点next域置为NULL
}

```

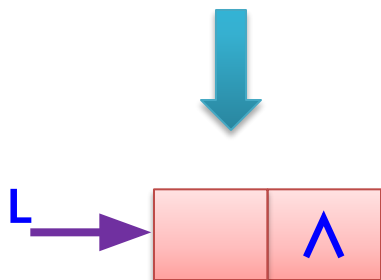


3、线性表基本运算在 单链表上的实现

(1)初始化线性表InitList(L)

建立一个空的 单链表, 即创建一个头结点。

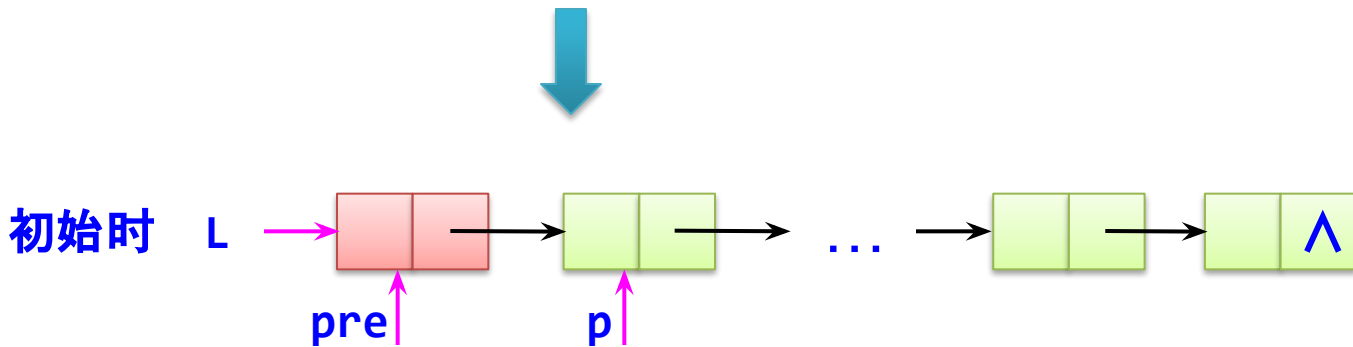
```
void InitList(LinkNode *&L)
{
    L=(LinkNode *)malloc(sizeof(LinkNode));    //创建头结点
    L->next=NULL;
}
```



(2) 销毁线性表 DestroyList(L)

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。

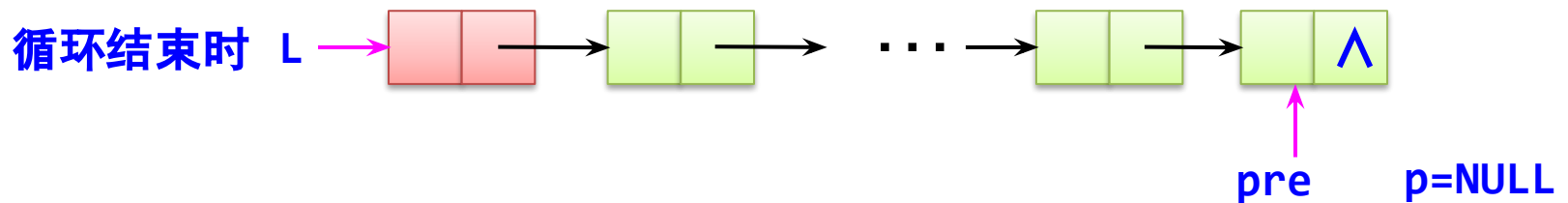
```
void DestroyList(LinkNode *&L)
{
    LinkNode *pre=L, *p=L->next;    //pre指向p的前驱结点
```



```

while (p!=NULL) //扫描单链表L
{
    free(pre); //释放pre结点
    pre=p;     //pre、p同步后移一个结点
    p=pre->next;
}
free(pre); //循环结束时, p为NULL, pre指向尾结点, 释放它
}

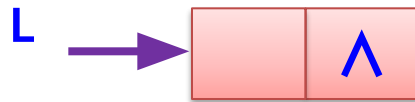
```



(3)判线性表是否为空表ListEmpty(L)

若单链表L没有数据结点, 则返回真, 否则返回假。

```
bool ListEmpty(LinkNode *L)
{
    return(L->next==NULL);
}
```

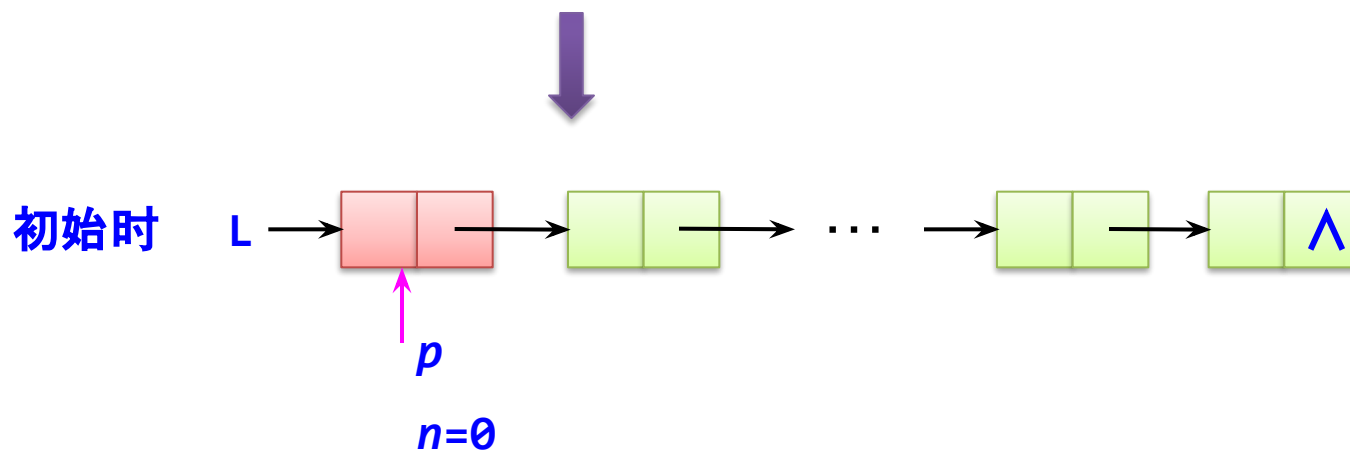


空表的情况

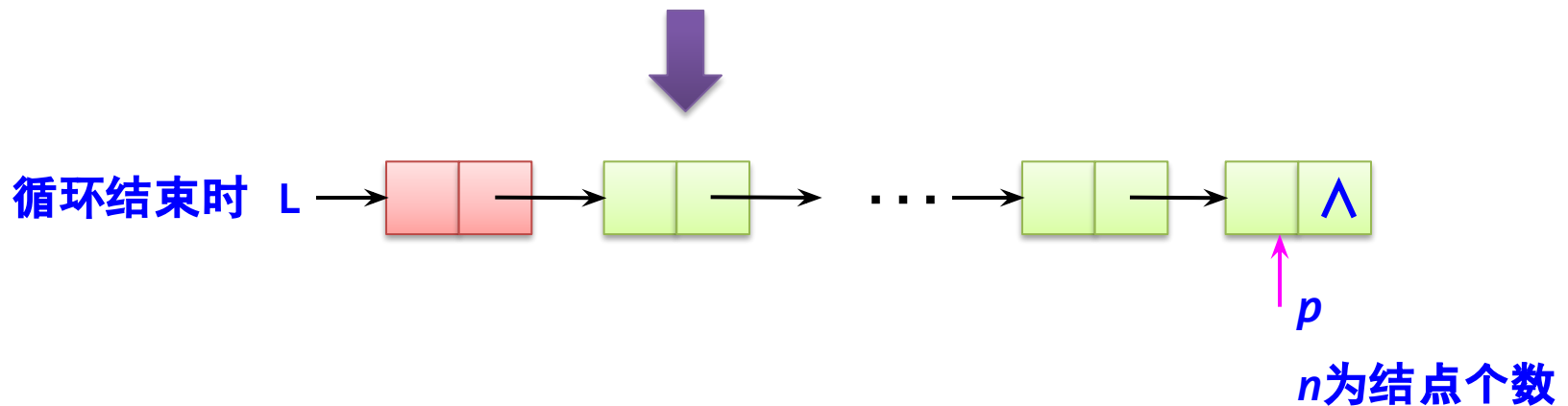
(4)求线性表的长度ListLength(L)

返回单链表L中数据结点的个数。

```
int ListLength(LinkNode *L)
{
    int n=0;
    LinkNode *p=L; //p指向头结点, n置为0(即头结点的序号为0)
```



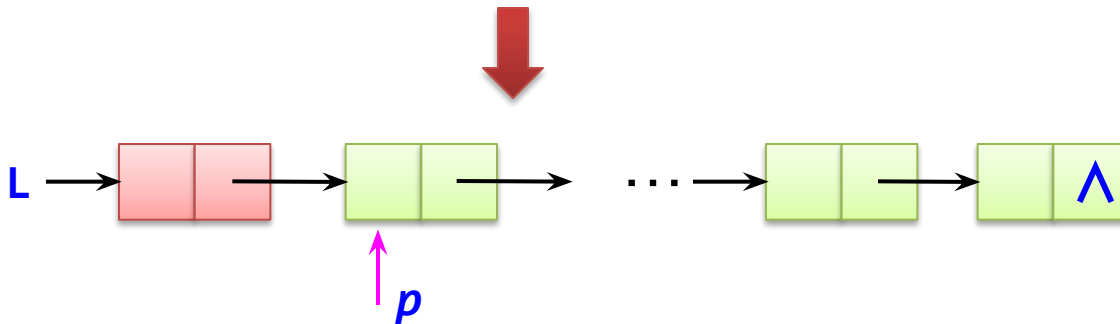
```
while (p->next!=NULL)
{
    n++;
    p=p->next;
}
return(n);    //循环结束, p指向尾结点, 其序号 n为结点个数
}
```



(5)输出线性表DispList(L)

逐一扫描单链表L的每个数据结点, 并显示各结点的data域值。

```
void DispList(LinkNode *L)
{
    LinkNode *p=L->next;    //p指向开始结点
    while (p!=NULL)         //p不为NULL, 输出p结点的data域
    { printf("%d ", p->data);
      p=p->next;             //p移向下一个结点
    }
    printf("\n");
}
```



(6)求线性表L中位置 i 的数据元素 $\text{GetElem}(L, i, \&e)$

在单链表L中从头开始找到第 i 个结点, 若存在第 i 个数据结点, 则将其 data域值赋给变量 e 。

```

bool GetElem(LinkNode *L, int i, ElemType &e)
{
    int j=0;
    LinkNode *p=L;  //p指向头结点, j置为0(即头结点的序号为0)

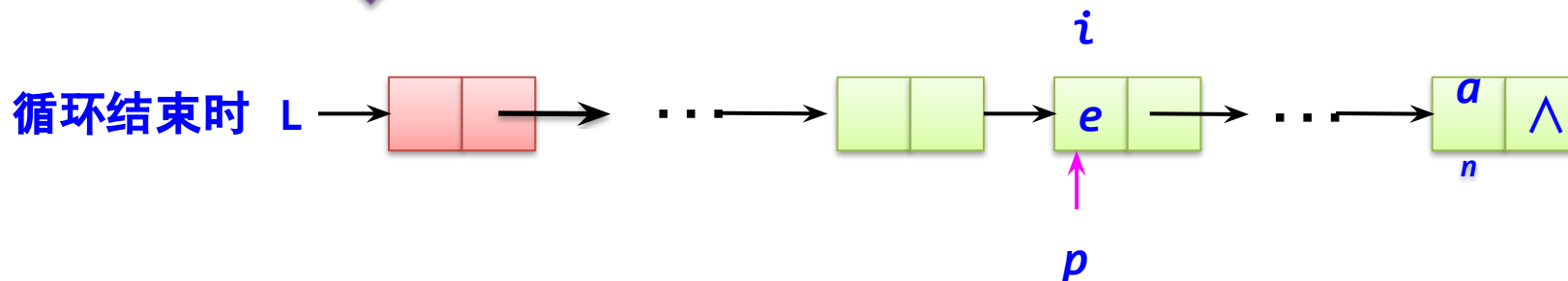
```

```

while (j<i && p!=NULL)
{
    j++;
    p=p->next;
}

```

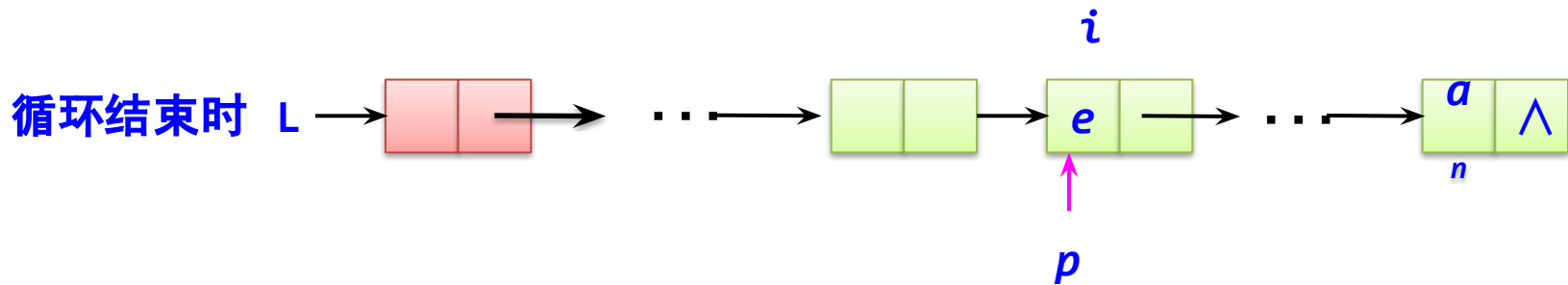
找第 i 个结点 p



```

if (p==NULL)    //不存在第 i 个数据结点, 返回 false
    return false;
else           //存在第 i 个数据结点, 返回 true
{
    e=p->data;
    return true;
}
}

```

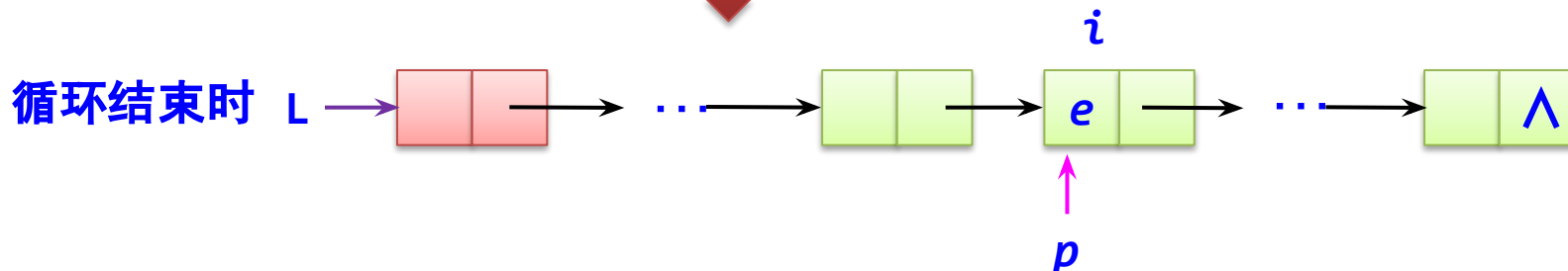


(7)按元素值查找LocateElem(L, e)

在单链表L中从头开始找第一个值域与 e 相等的结点, 若存在这样的结点, 则返回位置, 否则返回0。

```
int LocateElem(LinkNode *L, ElemType e)
{
    int i=1;
    LinkNode *p=L->next;    //p指向开始结点, i置为1

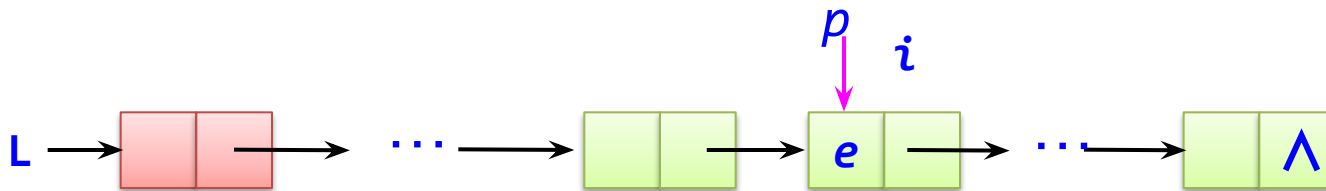
    while (p!=NULL && p->data!=e)
    {
        p=p->next;          //查找data值为e的结点, 其序号为i
        i++;
    }
}
```



```

if (p==NULL)    //不存在元素 值为 $e$ 的结点, 返回0
    return 0;
else            //存在元素 值为 $e$ 的结点, 返回其 逻辑序号 $i$ 
    return i;
}

```



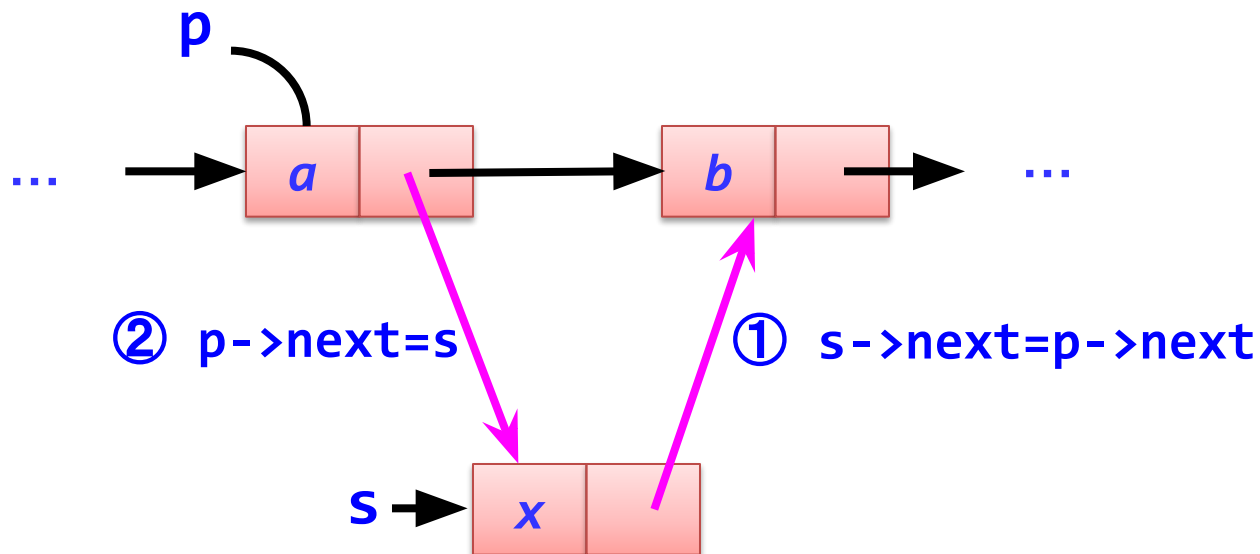
算法的时间复杂度为 $O(n) \Rightarrow$ 不具有随机存取特性

(1)插入结点

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, 不需要移动结点。

单链表插入结点演示



插入操作语句描述如下:

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

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

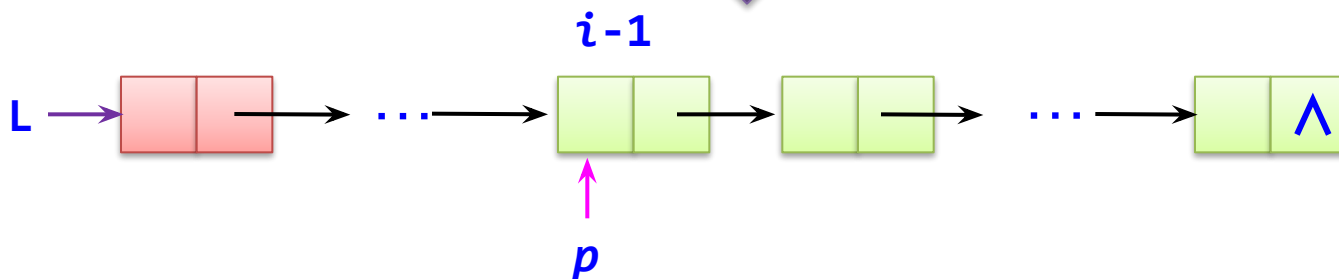
(8)插入数据元素 ListInsert(&L, i, e)

先在单链表L中找到第 $i-1$ 个结点p, 若存在这样的结点, 将值为e的结点s插入到其后。

```
bool ListInsert(LinkNode *&L, int i, ElemType e)
{   int j=0;
    LinkNode *p=L, *s;           //p指向头结点, j置为0
```

```
while (j<i-1 && p!=NULL)
{
    j++;
    p=p->next;
}
```

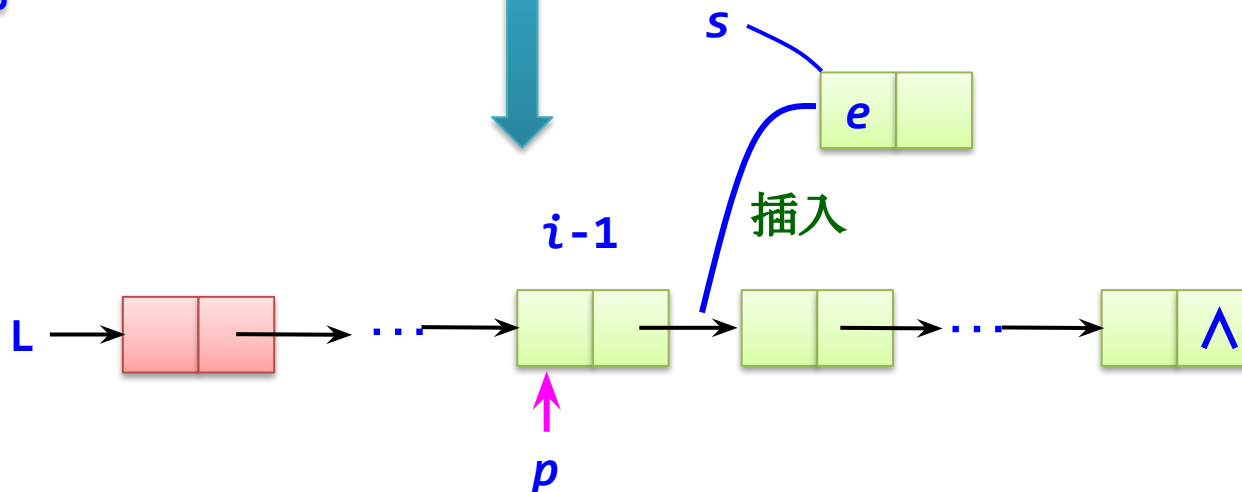
查找第 $i-1$ 个结点



```

if (p==NULL)           //未找到第 i-1 个结点, 返回 false
    return false;
else                   //找到第 i-1 个结点 p, 插入并返回 true
{
    s=(LinkNode *)malloc(sizeof(LinkNode));
    s->data=e;          //创建新结点 s, 其 data 域置为 e
    s->next=p->next;     //将 s 插入到 p 之后
    p->next=s;
    return true;
}
}

```

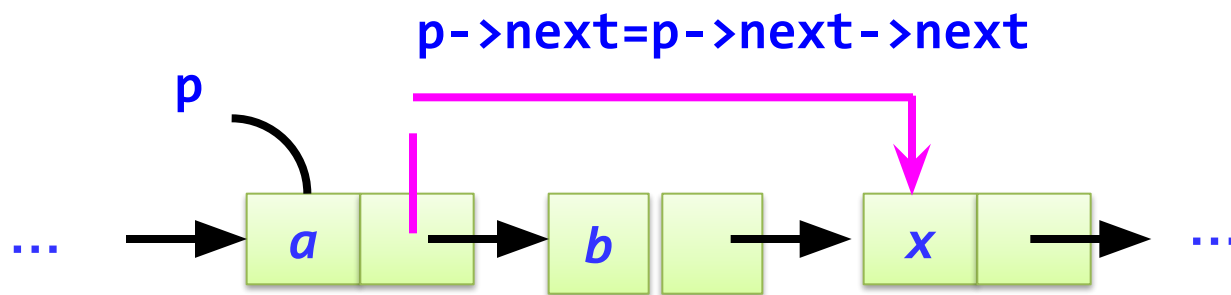


(2) 删除结点

删除操作：删除 p 结点之后的一个结点。

特点：只需修改相关结点的指针域，不需要移动结点。

单链表删除结点演示



删除操作语句描述如下：

$p \rightarrow next = p \rightarrow next \rightarrow next;$

(9) 删除数据元素 ListDelete(&L, i, &e)

先在单链表L中找到第 $i-1$ 个结点 p , 若存在这样的结点, 且也存在后继结点, 则删除该后继结点。

```
bool ListDelete(LinkNode *&L, int i, ElemType &e)
```

```
{  int j=0;
```

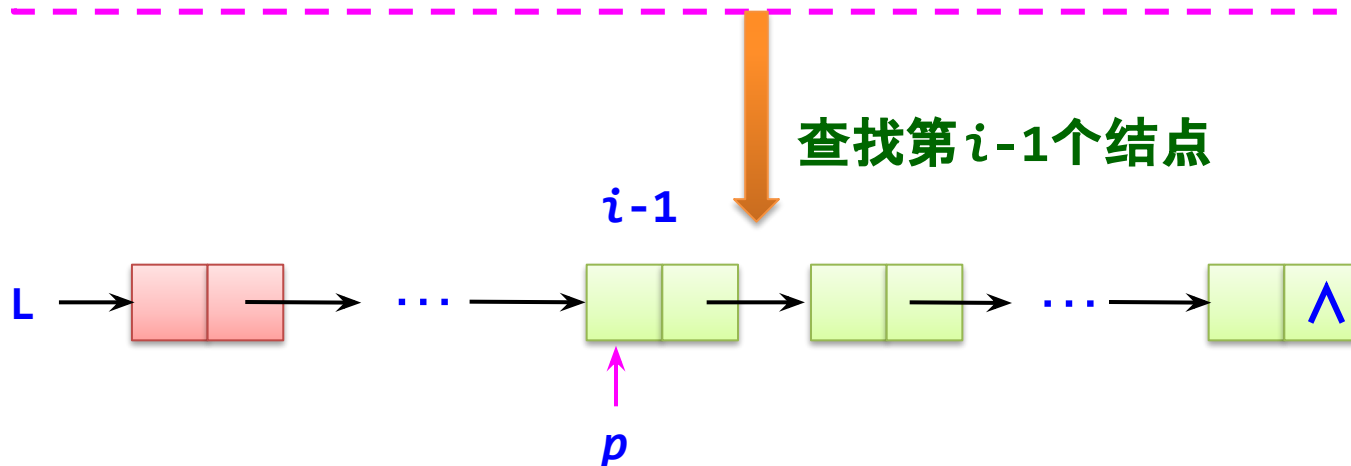
```
    LinkNode *p=L, *q;    //p指向头结点, j置为0
```

```
    while (j<i-1 && p!=NULL)    //查找第  $i-1$  个结点
```

```
    {    j++;
```

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

```
    }
```

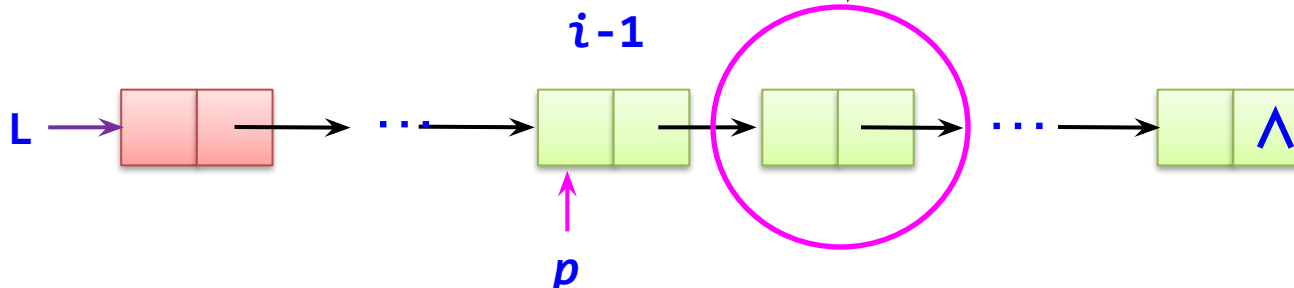


```

if (p==NULL)           //未找到第 i-1 个结点, 返回 false
    return false;
else                   //找到第 i-1 个结点 p
{
    q=p->next;          //q 指向第 i 个结点
    if (q==NULL)        //若不存在第 i 个结点, 返回 false
    {
        return false;
    }
    e=q->data;
    p->next=q->next;    //从单链表中删除 q 结点
    free(q);            //释放 q 结点
    return true;        //返回 true 表示成功删除第 i 个结点
}
}

```

删除第 i 个结点



4、单链表的算法设计方法

单链表的算法设计是线性表链式存储结构算法设计的基础,是需要重点掌握的内容。这里总结一般的算法设计方法。

① 以建表算法为基础的算法设计

- 单链表有尾插法和头插法两种建表算法。
- 很多算法是以这两个建表算法为基础进行设计的。

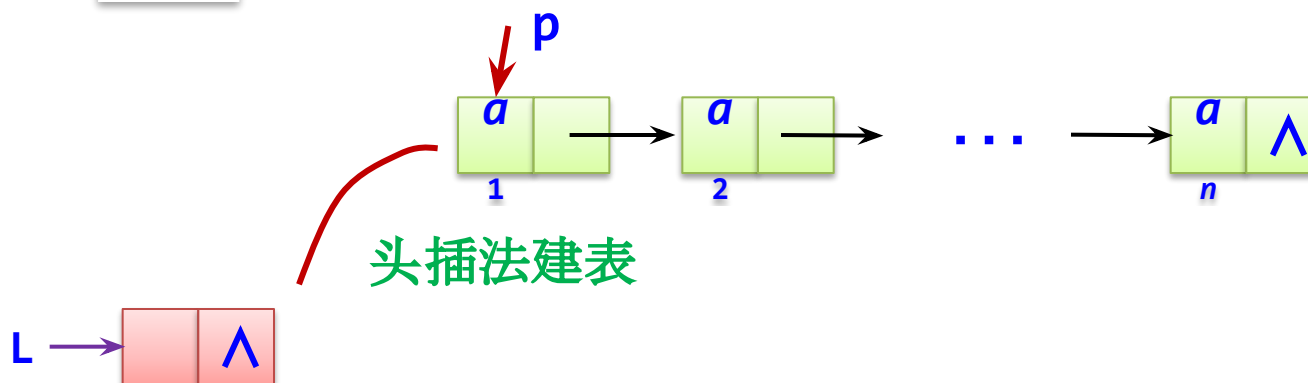
【例(补充)】 假设有一个带头结点的单链表

$$L = (a_1, a_2, \dots, a_n)。$$

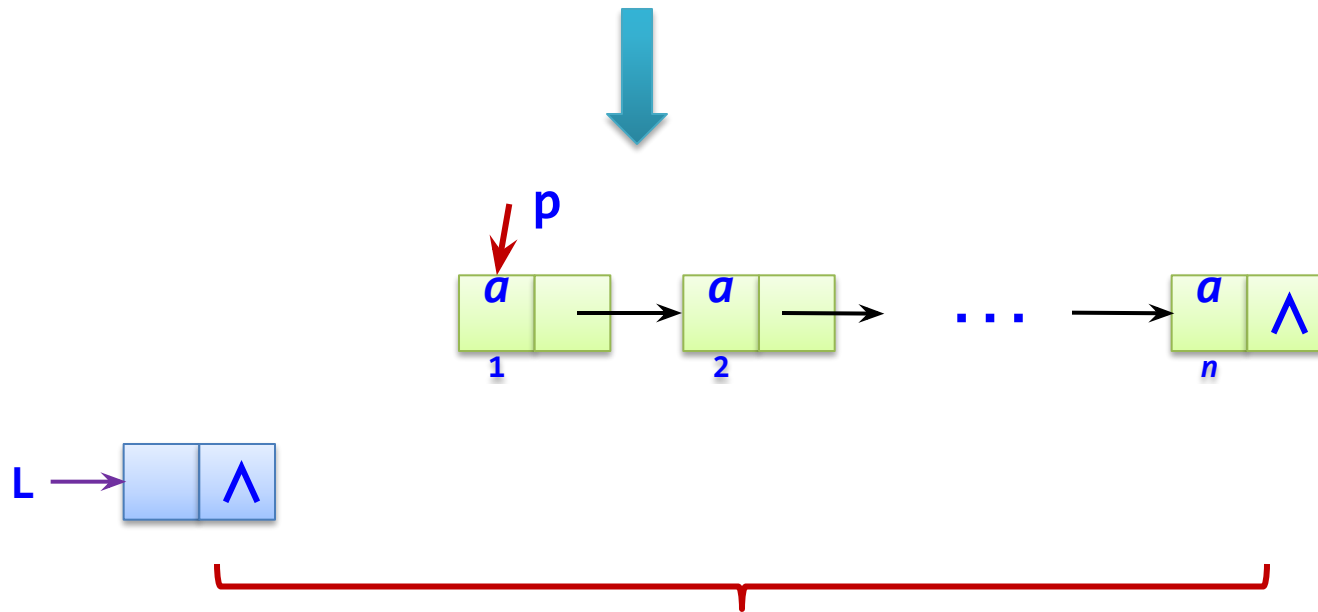
设计一个算法将所有结点逆置, 即:

$$L = (a_n, a_{n-1}, \dots, a_1)$$

解



```
void Reverse(LinkNode *&L)
{
    LinkNode *p=L->next, *q;
    L->next=NULL;
```

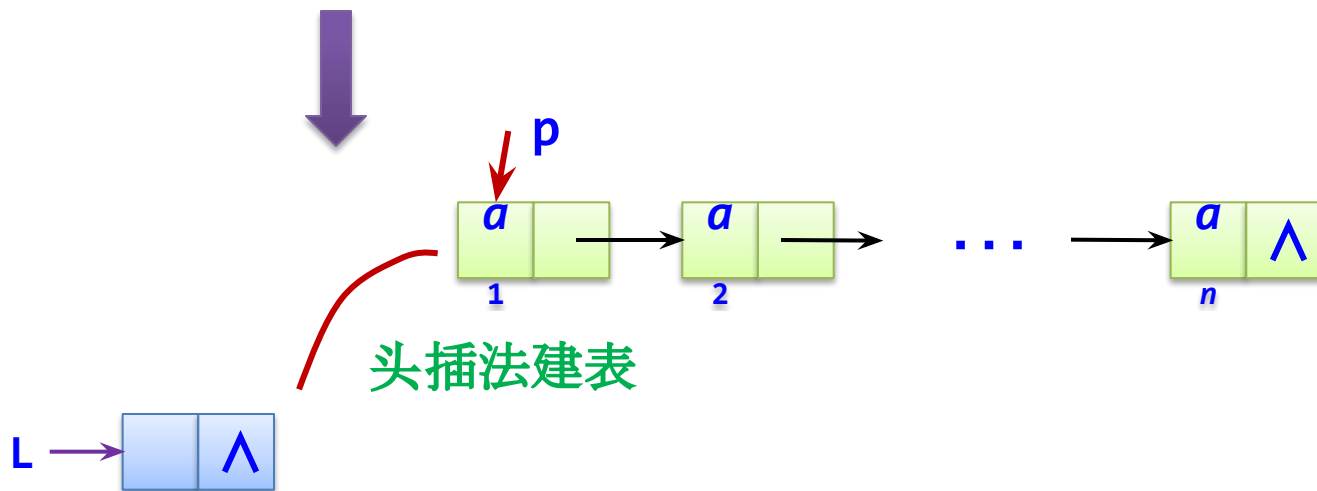


将L拆分为两个部分

```

while (p!=NULL)
{
    q=p->next;    //临时保存p的后继结点
    p->next=L->next; //将p结点采用头插法连接
    L->next=p;
    p=q;
}
}

```

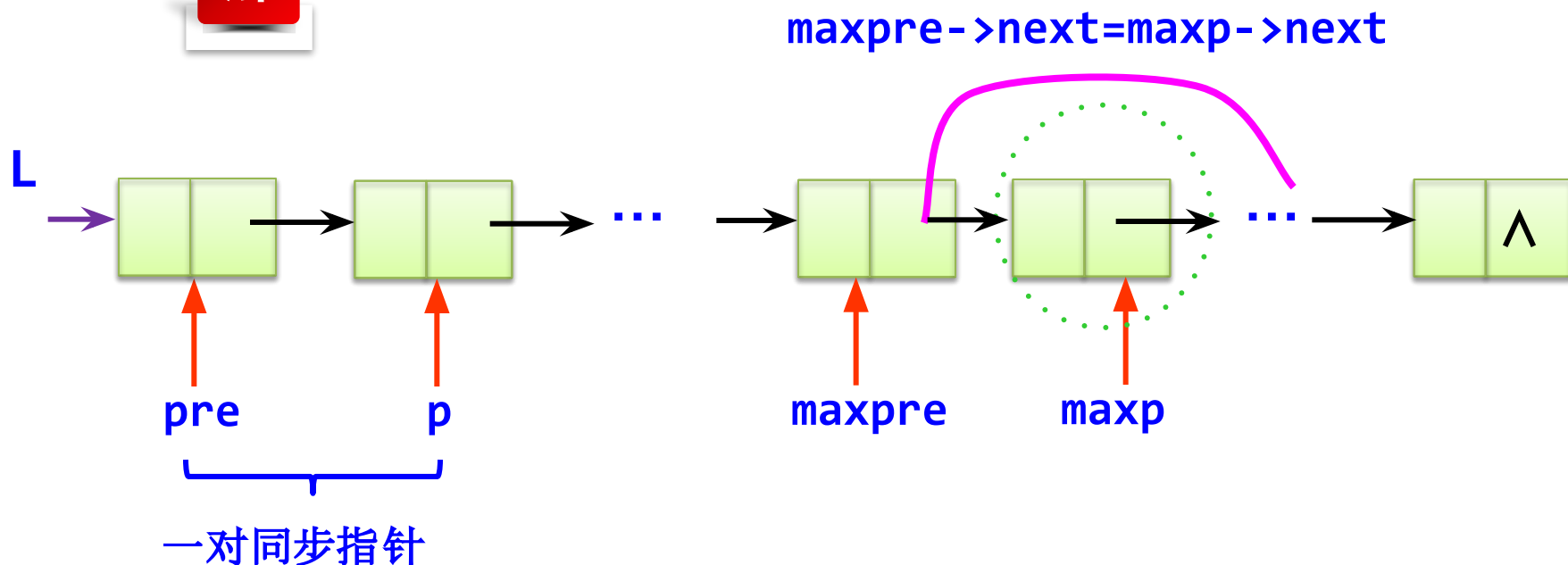


② 以查找为基础的算法设计

- 按照条件进行结点查找;
- 进行插入或者删除操作。

【例2.7】设计一个算法，删除一个单链表L中元素值最大的结点(假设最大值结点是唯一的)。

解



```

void delmaxnode(LinkNode *&L)
{
    LinkNode *p=L->next, *pre=L, *maxp=p, *maxpre=pre;

    while (p!=NULL)
    {
        if (maxp->data<p->data)    //若找到一个更大的 结点
        {
            maxp=p;                //更改maxp
            maxpre=pre;            //更改maxpre
        }
        pre=p;                    //p、pre同步后移一个 结点
        p=p->next;
    }

    maxpre->next=maxp->next;    //删除maxp结点
    free(maxp);                //释放maxp结点
}  该算法的时间复杂度为 $O(n)$ 。

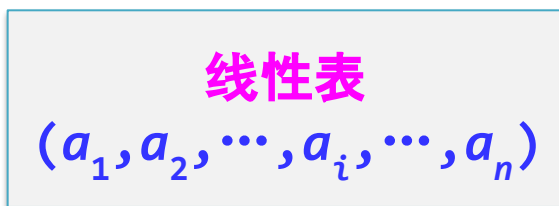
```

2.3.3 双链表

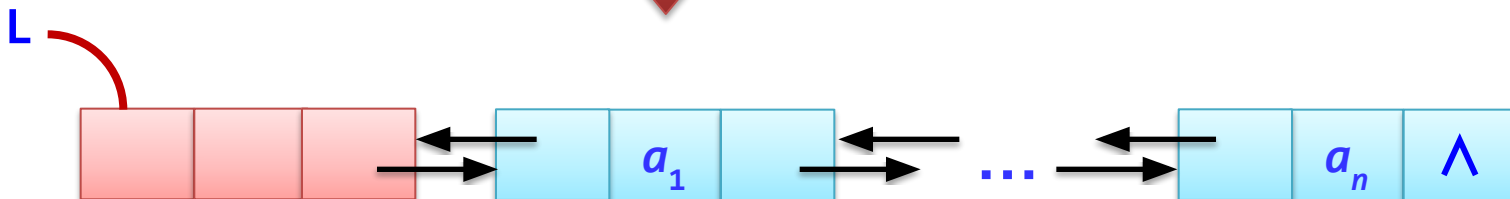
在线性表的链式存储结构中, 每个物理 结点增加一个指向后 继结点的指针域和一个指向前 驱结点的指针域 ⇨ 双链表。

逻辑结构

存储结构



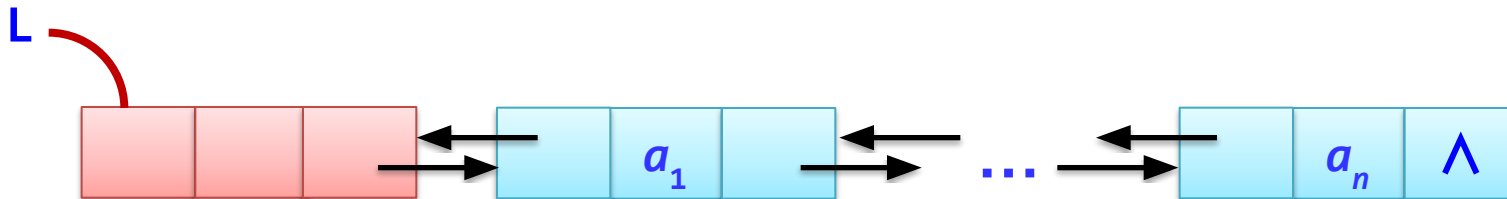
映射



带头结点双链表示意图

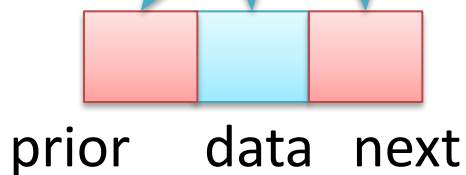
双链表的优点：

- 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 DLinkNode 声明如下:

```
typedef struct DNode          //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
} DLinkNode;
```

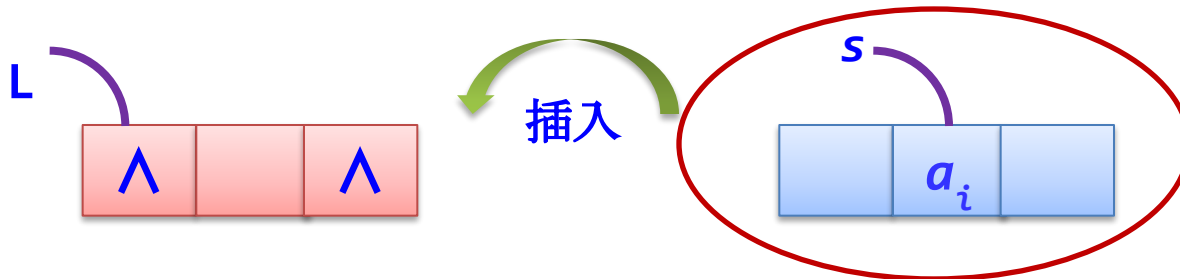


1、建立双链表

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。

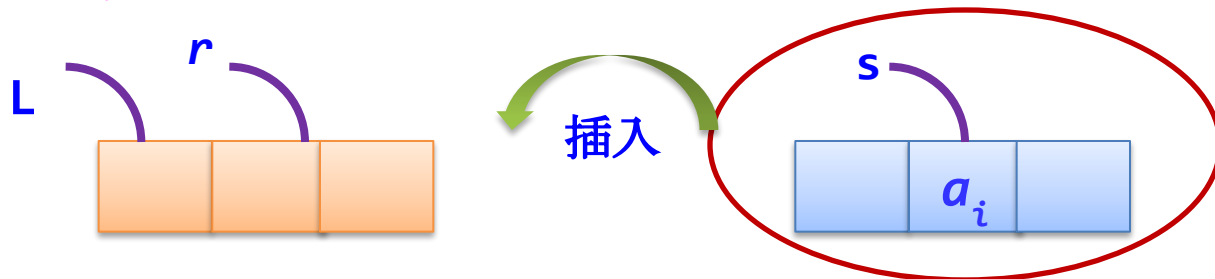
头插法建立双链表:由含有 n 个元素的数组 a 创建带头结点的双链表 L 。

```
void CreateListF(DLinkNode *&L, ElemType a[], int n)
{
    DLinkNode *s; int i;
    L=(DLinkNode *)malloc(sizeof(DLinkNode)); //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=(DLinkNode *)malloc(sizeof(DLinkNode));
        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;
    }
}
```



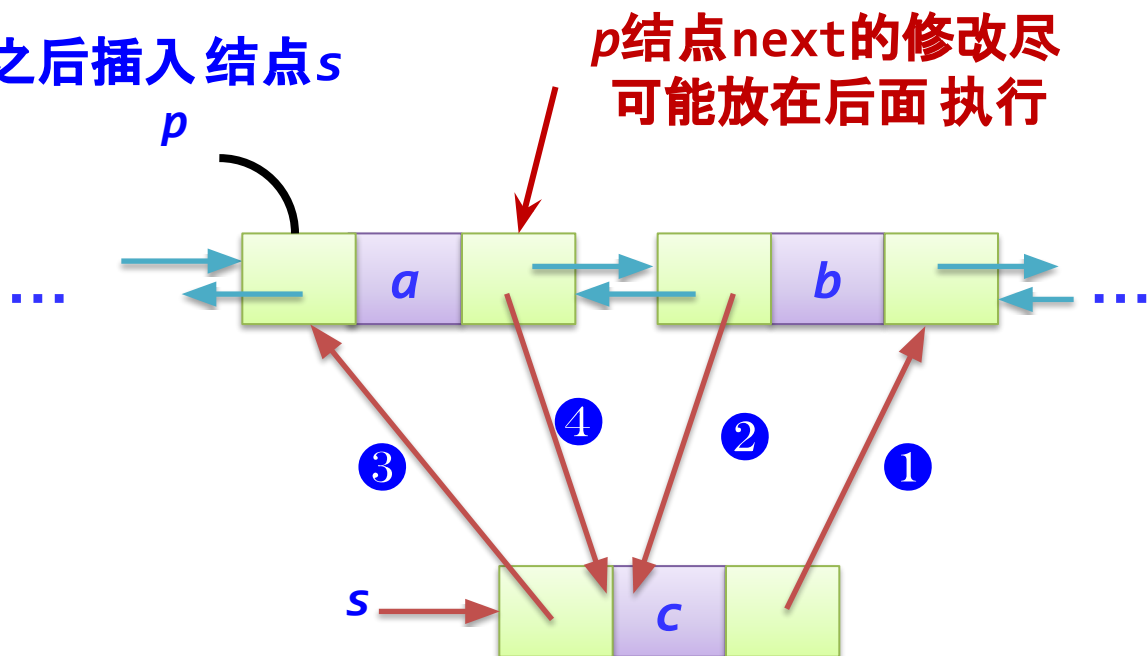
尾插法建立双链表:由含有 n 个元素的数组 a 创建带头结点的双链表 L 。

```
void CreateListR(DLinkNode *&L, ElemType a[], int n)
{  DLinkNode *s, *r;
   int i;
   L=(DLinkNode *)malloc(sizeof(DLinkNode));    //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=(DLinkNode *)malloc(sizeof(DLinkNode));
      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
}
```



(1) 双链表插入结点操作

在 p 结点之后插入结点 s



操作语句:

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

插入完毕

2、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。

① 双链表的插入算法：

```
bool ListInsert(DLinkNode *&L, int i, ElemType e)
{   int j=0;
    DLinkNode *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个结点
    {   j++;
        p=p->next;
    }
```

查找第 $i-1$ 个结点 p


```

if (p==NULL)           //未找到第 i-1 个结点, 返回 false
    return false;
else                   //找到第 i-1 个结点 p, 在其后插入新 结点 s
{
    s=(DLinkNode *)malloc(sizeof(DLinkNode));
    s->data=e;          //创建新结点 s
    s->next=p->next;     //在 p 之后插入 s 结点
    if (p->next!=NULL) //若存在后继结点, 修改其前驱指针
        p->next->prior=s;
    s->prior=p;
    p->next=s;
    return true;
}
}

```



新建结点 s , 将其插入到 p 结点之后

另外解法: 在双链表中, 可以查找第 i 个结点, 并在它前面插入一个 结点。

② 双链表的删除算法:

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

查找第 $i-1$ 个结点 p

```

if (p==NULL)                //未找到第i-1个结点
    return false;
else                        //找到第i-1个结点p
{
    q=p->next;              //q指向第i个结点
    if (q==NULL)            //当不存在第i个结点时返回false
        return false;
    e=q->data;
    p->next=q->next;         //从双单链表中删除q结点
    if (q->next!=NULL)       //若q结点存在后继结点
        q->next->prior=p;    //修改q结点后继结点的前驱指针
    free(q);                //释放q结点
    return true;
}
}

```


删除q结点并释放其空间

另外解法：在双链表中，可以查找第i个结点，并将它删除。


2.3.4 循环链表

循环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。
- **循环双链表**:形成两个环。

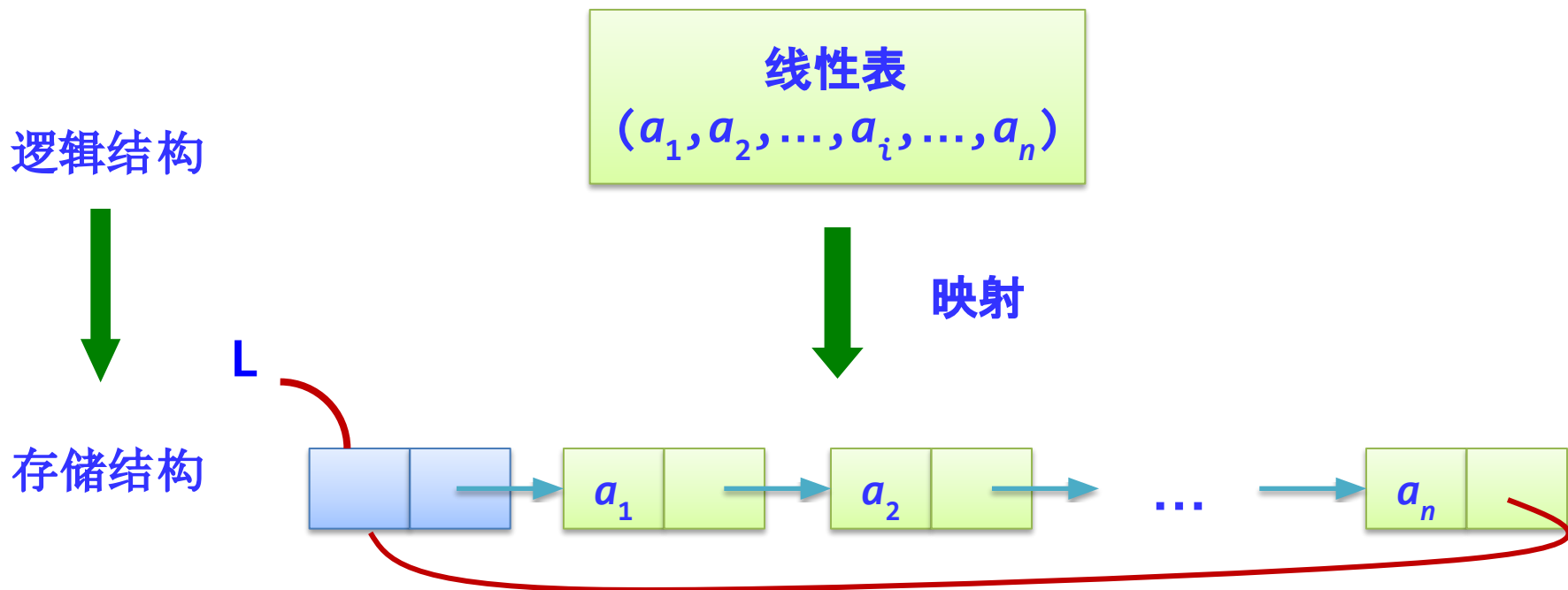


结点类型与非循环单链表的相同



结点类型与非循环双链表的相同

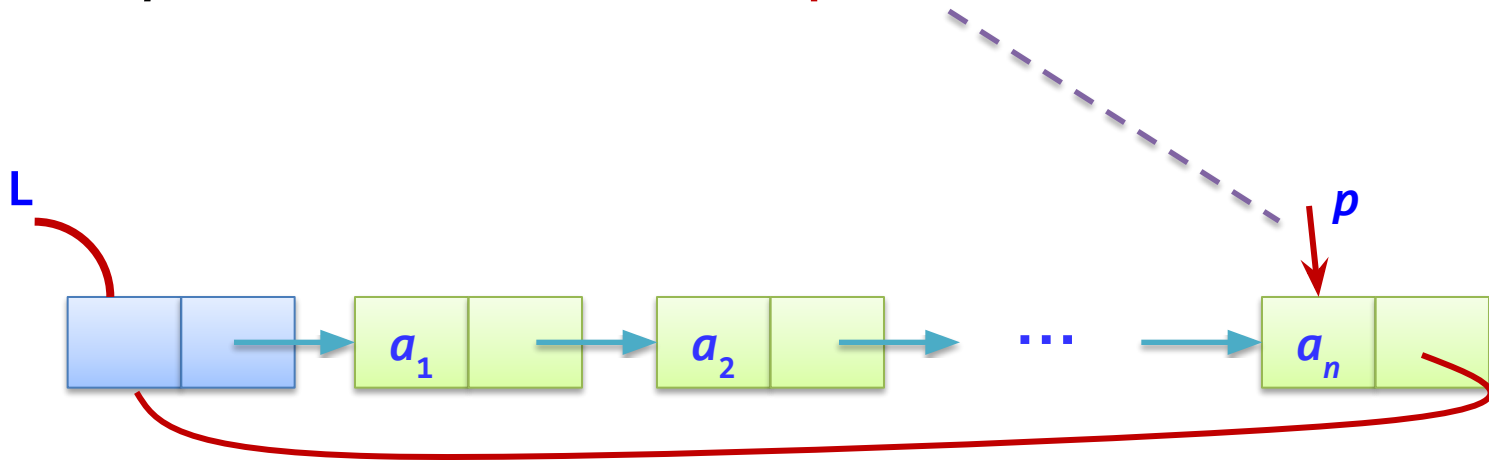
1、循环单链表



带头结点循环单链表示意图

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: $p \rightarrow \text{next} == L$



2、循环双链表

逻辑结构

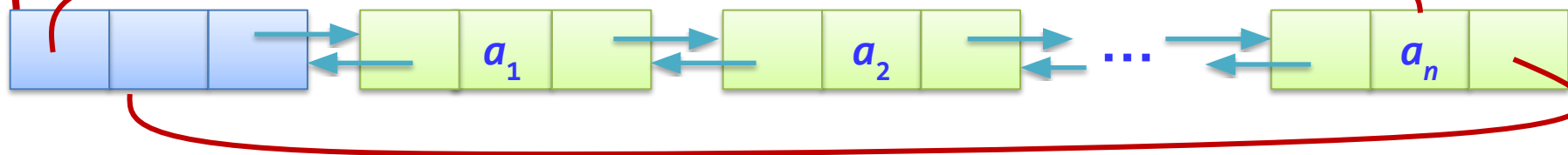


存储结构

线性表

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

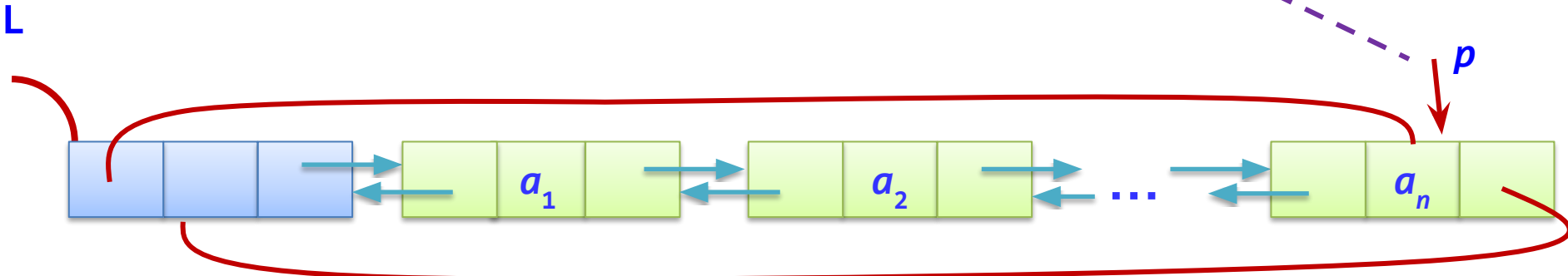
映射



带头结点循环双链表示意图

与非循环双链表相比, 循环双链表:

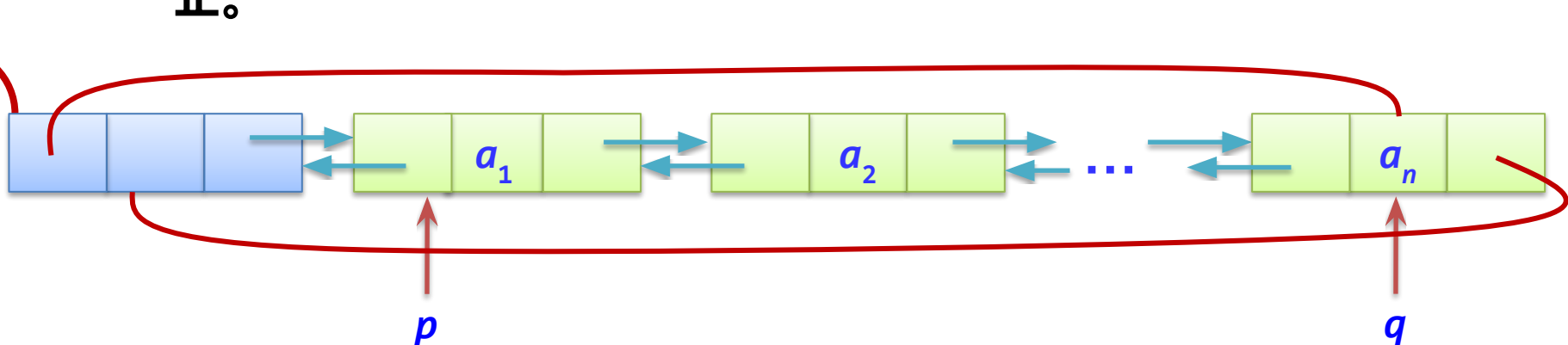
- 链表中没有空指针域
- p 所指结点为尾结点的条件: $p \rightarrow \text{next} == L$
- 一步操作即 $L \rightarrow \text{prior}$ 可以找到尾结点



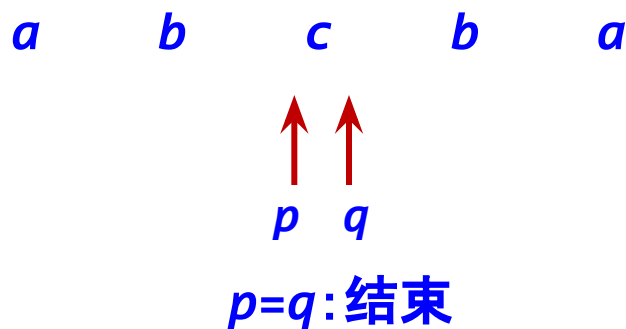
【例2.13】设计判断带头结点的循环双链表L(含两个以上的结点)是否对称相等的算法。

解

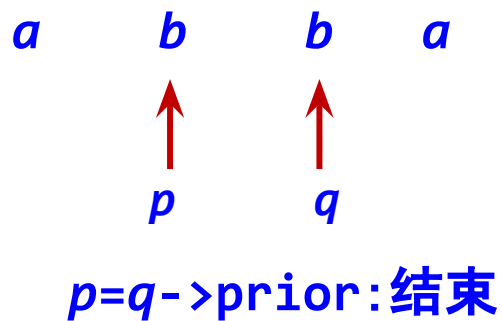
- p 从左向右扫描L, q 从右向左扫描L
- 若对应数据结点的data域不相等, 则退出循环
- 否则继续比较, 直到 p 与 q 相等或 p 的下一个结点为 q 为止。



① 数据结点为奇数的情况:



② 数据结点为偶数的情况:



```

bool Symm(DLinkNode *L)
{
    bool same=true;
    DLinkNode *p=L->next;           //p指向首结点
    DLinkNode *q=L->prior;          //q指向尾结点
    while (same)
    {
        if (p->data!=q->data)
            same=false;
        else
        {
            if (p==q || p==q->prior) break;
            q=q->prior;               //q前移
            p=p->next;                //p后移
        }
    }
    return same;
}

```

2.4 线性表的应用

问题描述

两个表自然连接问题

- 表: m 行、 n 列。假设所有元素为整数。


第1列	第2列	第3列
1	2	3
2	3	3
1	1	1

一个3行3列的表

- 两个表自然连接

A

1	2	3
2	3	3
1	1	1



3=1

B

3	5
1	6
3	4

C

1	2	3	3	5
1	2	3	3	4
2	3	3	3	5
2	3	3	3	4
1	1	1	1	6

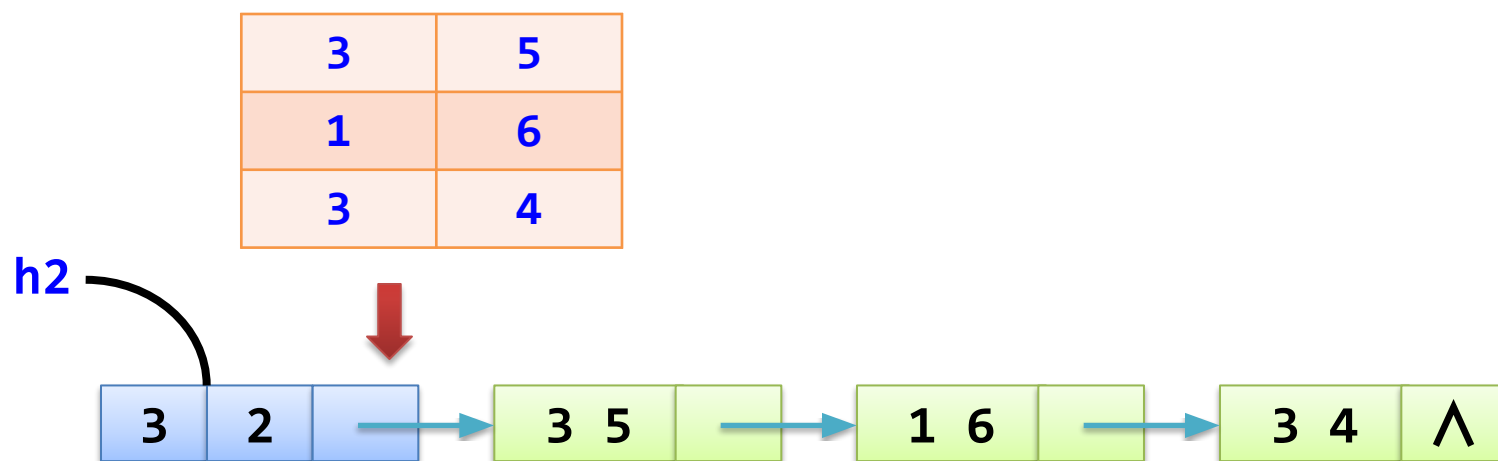
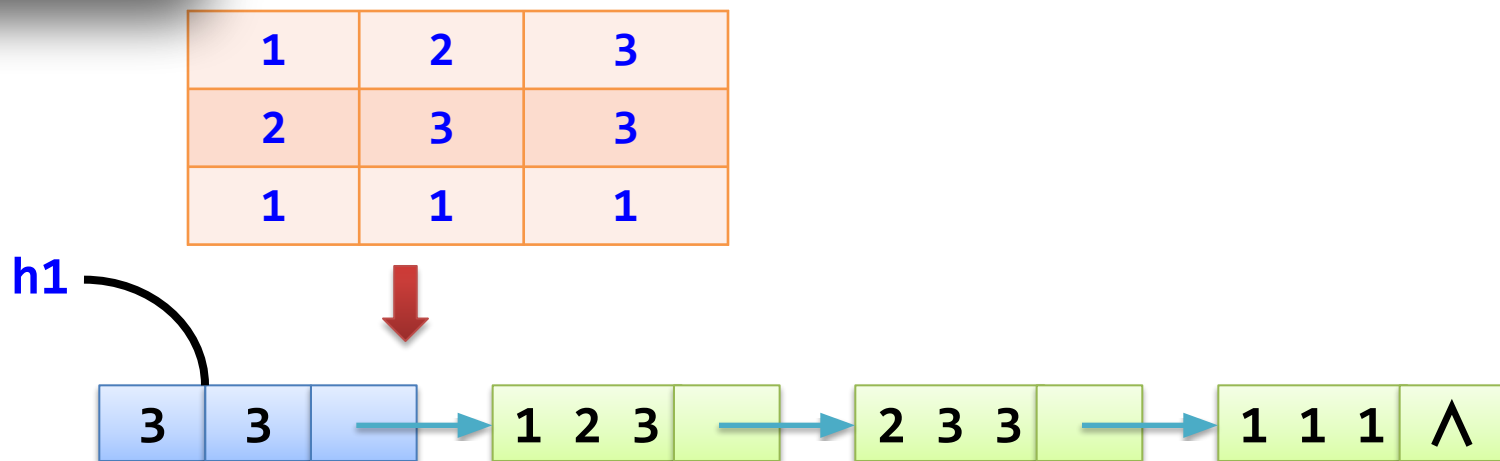
$C=A \bowtie B$

3=1

连接结果

一般格式: $C=A \bowtie_{i=j} B$

数据组织



注意：头结点和数据结点的类型不同！！！！

单链表中数据结点类型声明如下：

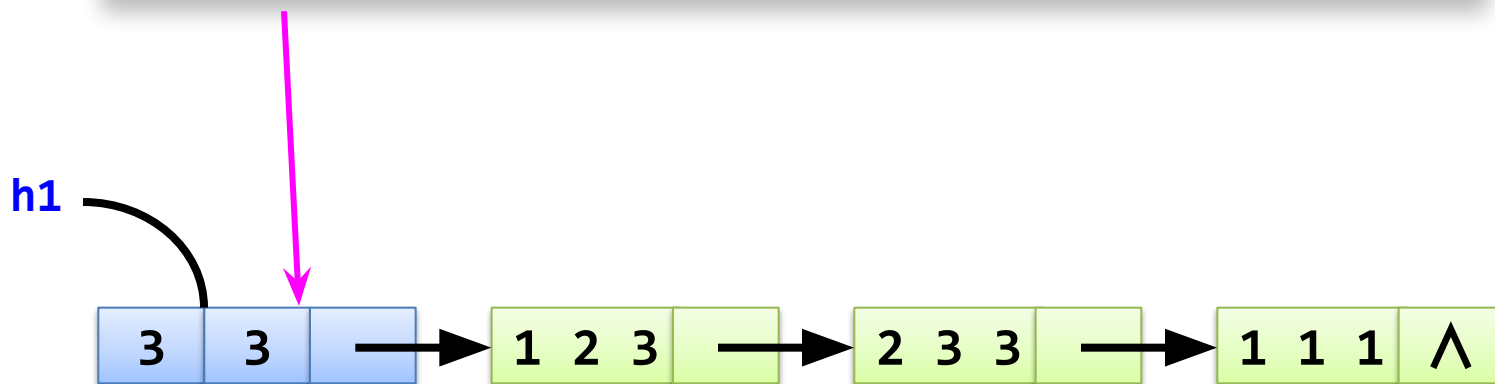
```
#define MaxCol 10          //最大列数  
typedef struct Node1       //定义数据结点类型  
{ ElemType data[MaxCol];  
  struct Node1 *next;      //指向后继数据结点  
} DList;
```

h1



头结点类型声明如下：

```
typedef struct Node2 //定义头结点类型
{
    int Row, Col;      //行数和列数
    DList *next;       //指向第一个数据 结点
} HList;
```



顺序表和链表混合使用！！！！

设计基本运算算法

1. CreateTable(HList *&h):交互式创建单链表。
2. DestroyTable(HList *&h) :销毁单链表。
3. DispTable (HList *h):输出单链表。
4. LinkTable(HList *h1, HList *h2, HList *&h):实现两个单链表的自然连接运算。

主程序



```
CreateTable(HList *&h)
DestroyTable(HList *&h)
DispTable (HList *h)
LinkTable(HList *h1, HList *h2, HList *&h)
```

(1)交互式创建单链表算法

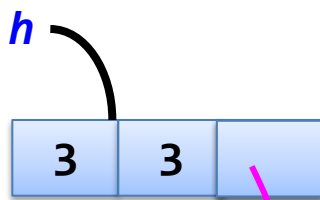
```
void CreateTable(HList *&h)
{  int i, j;  DList *r, *s;
   h=(HList *)malloc(sizeof(HList));    //创建头结点
   h->next=NULL;
   printf("表的行数, 列数 :");
   scanf("%d%d", &h->Row, &h->Col);    //输入表的行数和列数
   for (i=0;i<h->Row;i++)              //输入所有行的数据
   {  printf("  第%d行:", i+1);
      s=(DList *)malloc(sizeof(DList)); //创建数据结点
      for (j=0;j<h->Col;j++)           //输入一行的数据
      scanf("%d", &s->data[j]);
      if (h->next==NULL)                //插入第一个数据 结点
          h->next=s;
      else                               //插入其他数据 结点
          r->next=s;                     //将s插入到r结点之后
      r=s;                              //r始终指向尾结点
   }
   r->next=NULL;                        //尾结点next域置空
```

采用尾插法建表

为什么与一般尾插法建表不一样？

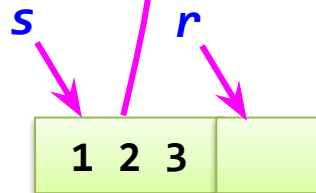
原因是头结点与数据结点类型不同！ r 不能指向头结点。

① 创建头结点 h



② 创建第一个数据结点 s

```
if
(h->next==NULL)
{ h->next=s;
  r=s;
}
```



③ 创建其他数据结点 s ，直接链接到 r 结点的后面

```
if
(h->next!=NULL)
{ r->next=s;
  r=s;
}
```

(2) 销毁单链表算法

```
void DestroyTable(HList *&h)
{
    DList *pre=h->next, *p=pre->next;
    while (p!=NULL)
    {
        free(pre);
        pre=p;
        p=p->next;
    }
    free(pre);
    free(h);
}
```

(3)输出单链表算法

```
void DispTable(HList *h)
{   int j;
    DList *p=h->next;           //p指向开始行 结点
    while (p!=NULL)             //扫描所有行
    {   for (j=0;j<h->Col;j++) //输出一行的数据
        printf("%4d", p->data[j]);
        printf("\n");
        p=p->next;               //p指向下一行 结点
    }
}
```

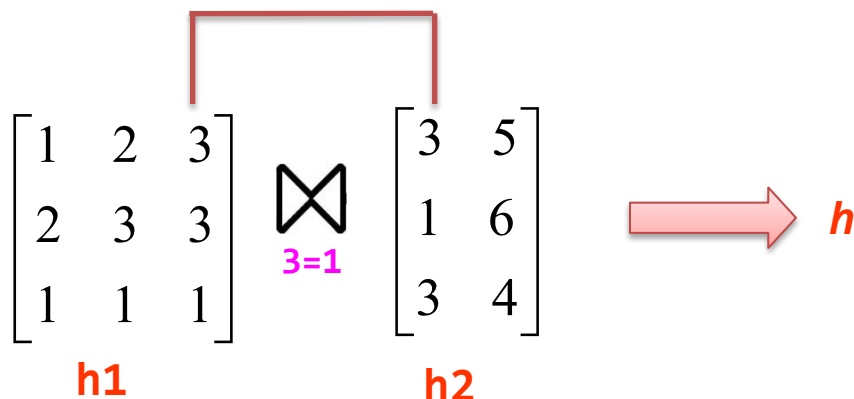
例如，输出一个表：

1	2	3	3	5
1	2	3	3	4
2	3	3	3	5
2	3	3	3	4
1	1	1	1	6

(4) 表连接运算算法

p 扫描 $h1$ 的数据结点, q 扫描 $h2$ 的数据结点。

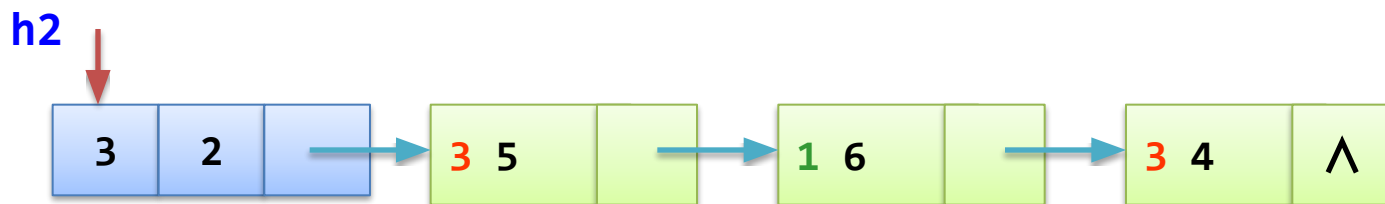
$p \rightarrow data[i-1] == q \rightarrow data[j-1]$



- 一旦条件成立, 就新建一个 结点插入到单链表 h 中。
- 单链表 h 采用尾插法建表方法 创建。

两表条件连接实现的演示

连接条件为 $3=1$



```
void LinkTable(HList *h1, HList *h2, HList *&h)
{
    int i, j, k;
    DList *p=h1->next, *q, *s, *r;

    printf("连接字段是:第1个表序号, 第2个表序号:");
    scanf("%d%d", &i, &j);

    h=(HList *)malloc(sizeof(HList)); //创建结果表头结点
    h->next=NULL;                      //置next域为NULL
    h->Row=0;                          //置行数为0
    h->Col=h1->Col+h2->Col;            //置列数为表1和表2的列数和
}
```



```
while (p!=NULL)                                //扫描表1
{  q=h2->next;                                  //q指向表2的开始结点
  while (q!=NULL)                              //扫描表2
  {  if (p->data[i-1]==q->data[j-1])  //对应字段值相等
    {
      s=(DList *)malloc(sizeof(DList)); //创建结点
      for (k=0;k<h1->Col;k++)           //复制表1的当前行
        s->data[k]=p->data[k];
      for (k=0;k<h2->Col;k++)           //复制表2的当前行
        s->data[h1->Col+k]=q->data[k];
```

```

    if (h->next==NULL) //若插入第一个数据 结点
        h->next=s;      //将s插入到头结点之后
    else                //若插入其他数据 结点
        r->next=s;      //将s插入到r结点之后
    r=s;                //r始终指向尾结点

    h->Row++;           //表行数增1
    }
    q=q->next;          //表2下移一个 记录
}
p=p->next;             //表1下移一个 记录
}
r->next=NULL;         //表尾结点next域置空
}

```

求解程序

建立如下主函数 调用上述算法：

```
void main()
{   HList *h1, *h2, *h;
    printf("表1:\n");
    CreateTable(h1);           //创建表1
    printf("表2:\n");
    CreateTable(h2);           //创建表2
    LinkTable(h1, h2, h);      //连接两个表
    printf("连接结果表:\n");
    DispTable(h);              //输出连接结果
    DestroyTable(h1);           //销毁单链表h1
    DestroyTable(h2);           //销毁单链表h2
    DestroyTable(h);            //销毁单链表h
}
```

运行结果

表1:

表的行数, 列数 : 3 3✓

第1行: 1 2 3✓

第2行: 2 3 3✓

第3行: 1 1 1✓

表2:

表的行数, 列数 : 3 2✓

第1行: 3 5✓

第2行: 1 6✓

第3行: 3 4✓

连接字段是 : 第1个表位序, 第2个表位序 : 3 1✓

连接结果表:

1 2 3 3 5

1 2 3 3 4

2 3 3 3 5

2 3 3 3 4

1 1 1 1 6

2.5 有序表

2.5.1 有序表的概念

所谓**有序表**，是指这样的线性表，其中所有元素以**递增或递减**方式有序排列。

后面讨论的有序表默认元素是**以递增方式排列**。

例如： $L=(1, 5, 8, 10, 15, 20)$ 就是一个整数有序表。

有序表是线性表的一个子集。

有序表 \subset 线性表

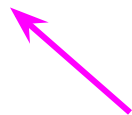
有序表和线性表中元素之间的逻辑关系相同, 其区别是运算实现的不同。

注意

有序表和顺序表的区别

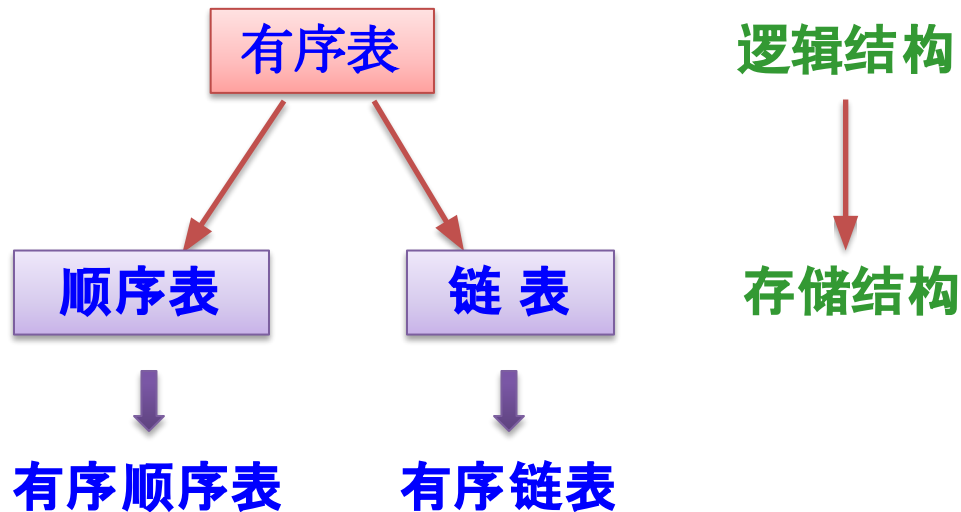
逻辑层面的概念

物理层面的概念



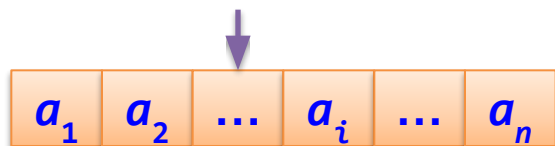
2.5.2 有序表的存储结构及其基本运算算法

既然有序表中元素逻辑关系与线性表的相同, 有序表可以采用与线性表相同的存储结构。



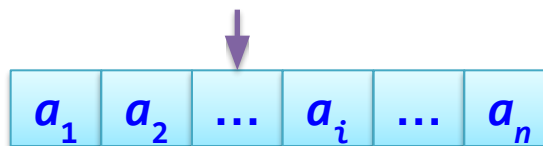
- 在以顺序表或者链表存储有序表时, 许多基本运算算法与线性顺序表或者链表的算法相同(不考虑优化)。
- 只有插入运算-ListInsert()算法有所差异。

ListInsert(L, i , e)



线性表

ListInsert(L, e)



有序表

若以顺序表存储有序表, ListInsert()算法如下:

```
void ListInsert(SqList *&L, ElemType e)
{
    int i=0, j;
    while (i<L->length && L->data[i]<e)
        i++;
        //查找值为e的元素
    for (j=ListLength(L);j>i;j--) //将data[i..n]后移一个位置
        L->data[j]=L->data[j-1];
    L->data[i]=e;
    L->length++;
    //有序顺序表长度增1
}
```

若以单链表存储有序表, ListInsert()的算法如下:

```
void ListInsert(LinkNode *&L, ElemType e)
{  LinkNode *pre=L, *p;

  while (pre->next!=NULL && pre->next->data<e)
    pre=pre->next;  //查找插入结点的前驱结点pre

  p=(LinkNode *)malloc(sizeof(LinkNode));
  p->data=e;        //创建存放e的数据结点p
  p->next=pre->next; //在pre结点之后插入p结点
  pre->next=p;
```

在pre之后插入p结点

查找插入的位置 pre

2.5.3 有序表的归并算法

【例2.14】假设有两个有序表 LA 和 LB。设计一个算法, 将它们合并成一个有序表 LC。



二路归并示意图

顺序表二路归并示例的演示

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 UnionList(SqList *LA, SqList *LB, SqList *&LC)
{  int i=0, j=0, k=0;    //i、j分别为LA、LB的下标, k为LC中元素个数
   LC=(SqList *)malloc(sizeof(SqList)); //建立有序顺序表LC
   while (i<LA->length && j<LB->length)
   {  if (LA->data[i]<LB->data[j])
      {  LC->data[k]=LA->data[i];
         i++;k++;
      }
      else //LA->data[i]>LB->data[j]
      {  LC->data[k]=LB->data[j];
         j++;k++;
      }
   }
}
```



两个有序表
均没有遍历
完

```
    while (i<LA->length) //LA尚未扫描完, 将其余元素插入 LC中
    { LC->data[k]=LA->data[i];
      i++;k++;
    }
    while (j<LB->length) //LB尚未扫描完, 将其余元素插入 LC中
    { LC->data[k]=LB->data[j];
      j++;k++;
    }
    LC->length=k;
}
```

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

2.5.4 有序表的应用

- 利用有序表元素的有序性提高 查找效率
- 利用二路归并过程提高算法效率

【例2.16】已知一个有序单链表L(允许出现值重复的结点), 设计一个高效算法删除值重复的结点。并分析算法的时间复杂度。

解

- 由于是有序单链表, 所以相同值的结点都是相邻的。
- 用 p 扫描递增单链表, 若 p 所指结点的值等于其后继点的值, 则删除后者。

```
void dels(LinkNode *&L)
{  LinkNode *p=L->next,*q;
   while (p->next!=NULL)
   {  if (p->data==p->next->data)    //找到重复值的结点
      {  q=p->next;                //q指向这个重复值的结点
         p->next=q->next;          //删除q结点
         free(q);
      }
      else                          //不是重复结点,p指针下移
         p=p->next;
   }
}
```

算法的时间复杂度为 $O(n)$