

第二章 线性表



本章内容

- 2.1 线性表的基本概念
- 2.2 线性表的顺序存储结构
- 2.3 线性表的链式存储结构
- 2.4 循环链表及其运算
- 2.5 双向链表及其运算
- 2.6 链表的应用举例

2.1 线性表的基本概念

一、线性表的定义

$$A=(a_1, a_2, a_3, \dots, a_n)$$

1. 线性关系

- (1) 当 $1 < i < n$ 时,
 a_i 的直接前驱为 a_{i-1} , a_i 的直接后继为 a_{i+1} 。
- (2) 除了第一个元素与最后一个元素,序列中任何一个元素有且仅有一个直接前驱元素,有且仅有一个直接后继元素。
- (3) 数据元素之间的先后顺序为“一对一”的关系。

2. 线性表的定义

数据元素之间具有的逻辑关系为线性关系的数据元素集合称为**线性表**， n 为线性表的长度，长度为0的线性表称为空表。

几个线性表的例子

$A = (a_1, a_2, a_3, \dots, a_n)$

一个数据元素
为一个整数

① 数列: (25, 12, 78, 34, 100, 88)
 $a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_5 \quad a_6$

② 字母表: ('A', 'B', 'C',, 'Z')
 $a_1 \quad a_2 \quad a_3 \quad \dots \quad a_{26}$

一个数据元素
为一个字母

3

数据文件：

	学号	姓 名	性别	年龄	其 他
a_1	99001	张 华	女	17
a_2	99002	李 军	男	18
a_3	99003	王 明	男	17
\vdots
\vdots
\vdots
a_{50}	99050	刘 东	女	19

一个数据元素
为一个记录

二. 线性表的基本操作

1. 创建一个新的线性表。
2. 求线性表的长度。
3. 检索线性表中第*i*个数据元素。
4. 根据数据元素的某数据项(通常称为关键字)的值求该数据元素在线性表中的位置。
5. 在线性表的第*i*个位置上存入一个新的数据元素。
6. 在线性表的第*i*个位置上插入一个新的数据元素。
7. 删除线性表中第*i*个数据元素。
8. 对线性表中的数据元素按照某一个数据项的值的大小做升序或者降序排序。

$$1 \leq i \leq n+1$$

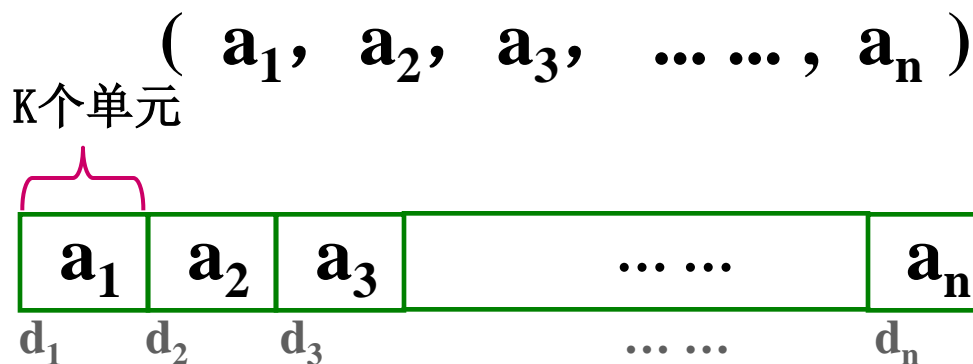
$$1 \leq i \leq n$$

9. 复制一个线性表。
10. 按照一定的原则，将两个或两个以上的线性表合并成为一个线性表。
11. 按照一定的原则，将一个线性表分解为两个或两个以上的线性表。
-

2.2 线性表的顺序存储结构

一、构造原理

用一组地址连续的存储单元依次存储线性表中的数据元素，数据元素之间的逻辑关系通过数据元素的存储位置直接反映。



$LOC(a_i)$

所谓一个元素的**地址**是指该元素占用的若干(连续的)存储单元的**第一个单元的地址**。

a_1	a_2	a_3	...	a_n
-------	-------	-------	-----	-------

结论:

若假设每个数据元素占用k个存储单元，并且已知第一个元素的存储位置 $LOC(a_1)$ ，则有

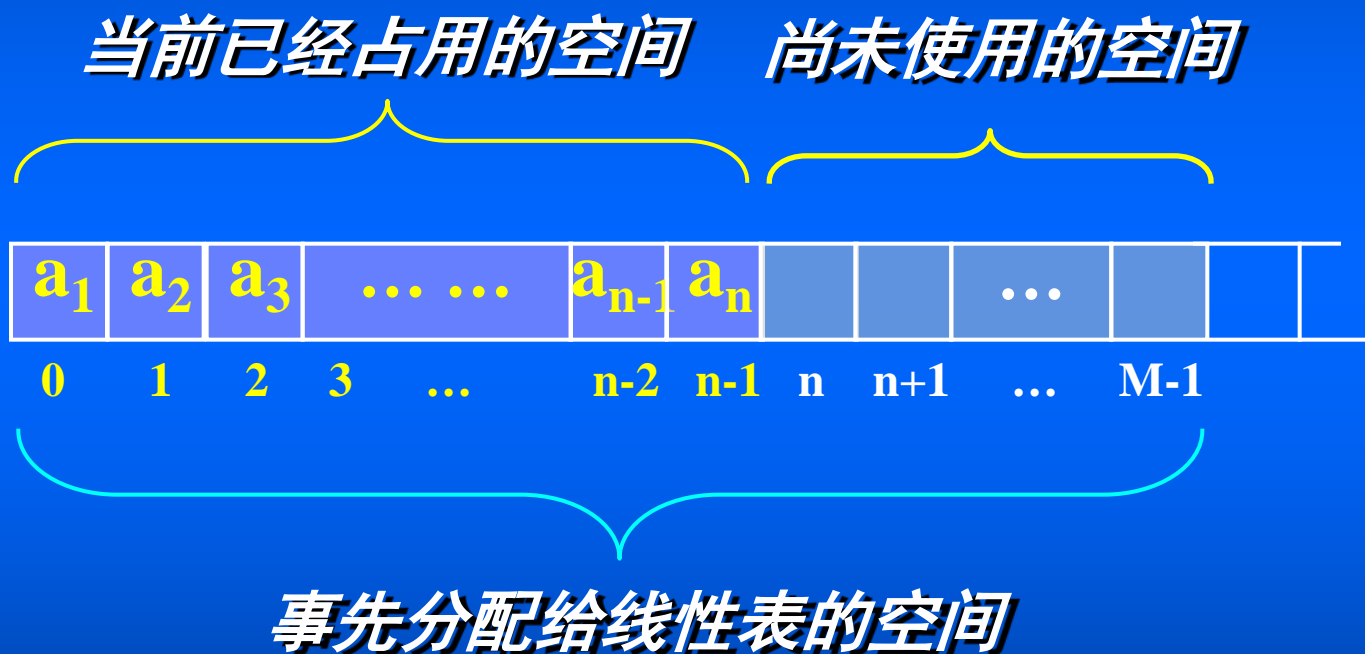
$$LOC(a_i) = LOC(a_1) + (i-1) \times k$$

例： $LOC(a_1)=100$ $k=4$ 求 $LOC(a_5)=?$

a_1	a_2	a_3	a_4	a_5	...	a_n
100	104	108	112	(116)		

$$LOC(a_5) = 100 + (5 - 1) \times 4 = 116$$

顺序存储结构示意图



预先分配给线性表的空间大小

在C语言中

```
#define MaxSize 100  
ElemType A[MaxSize];  
int n;
```

表的长度

顺序表（向量）

二、基本操作

1. 确定元素item在长度为n的线性表A中的位置

($a_1, a_2, a_3, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_{n-1}, a_n$)

顺序查找法

算法

```
int LOCATE( ElemType A[ ], int n, ElemType item )
{
    int i;
    for(i=0;i<n;i++)
        if (A[i]==item)
            return i+1;
    return(-1);
}
```

时间复杂度 $O(n)$

/* 查找成功, 返回在表中位置 */

/* 查找失败, 返回信息-1 */

2. 在长度为n的线性表A的第i个位置上插入一个新的数据元素item

该运算是在线性表的第*i*-1个数据元素与第*i*个数据元素之间插入一个由符号item表示的数据元素，使长度为n的线性表

$(a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_{n-1}, a_n)$

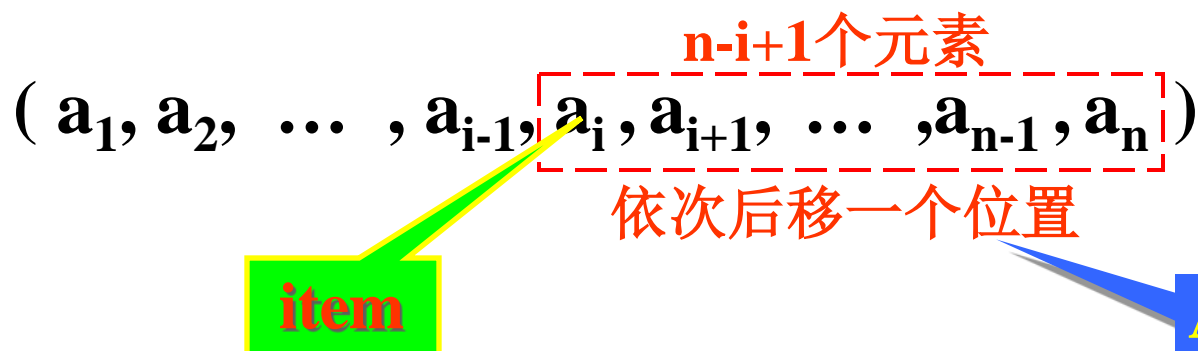
n个数据元素

转换成长度为n+1的线性表

$(a_1, a_2, \dots, a_{i-1}, \text{item}, a_i, \dots, a_{n-1}, a_n)$

n+1个数据元素





正常情况下需要做的工作:

- (1). 将第*i*个元素至第*n*个元素依次后移一个位置;
- (2). 将被插入元素插入表的第*i*个位置;
- (3). 修改表的长度(表长增1)。

需要考虑的异常情况:

- (1). 是否表满? $n = \text{MaxSize}$?
- (2). 插入位置是否合适? (正常位置: $1 \leq i \leq n+1$)

约定

若插入成功，算法返回1，
否则，算法返回-1。

算法

```
int INSERTLIST(ElemType A[], int &n, int i, ElemType item )
{
    int j;
    if (n==MaxSize || i<1 || i>n+1)
        return(-1);
    for( j=n-1; j>=i-1; j-- )
        A[j+1]=A[j];
    A[i-1]=item;
    n++;
    return(1);
}
```

测空间满否

/* 插入失败 */

/* 元素依次后移一个位置 */
/* 将item插入表的第i个位置 */
/* 线性表的长度加1 */
/* 插入成功 */

测插入位置合适否

算法时间复杂度分析

通常采用元素移动次数的平均值作为衡量插入和删除算法时间效率的主要指标。

若设 p_i 为插入一个元素于线性表第 i 个位置的概率(概率相等), 则在长度为 n 的线性表中插入一个元素需要移动其他的元素的平均次数为:

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

称该算法的时间复杂度是 $O(n)$ 。

3. 删除长度为n的线性表A的第i个数据元素

该运算是把线性表的第i个数据元素从线性表中去掉，使得长度为n的线性表

($a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_{n-1}, a_n$)

n个数据元素

转换成长度为 n-1 的线性表

($a_1, a_2, \dots, a_{i-1}, a_{i+1}, \dots, a_{n-1}, a_n$)

n-1个数据元素



($a_1, a_2, \dots, a_{i-1}, a_i, \underbrace{a_{i+1}, \dots, a_{n-1}, a_n}_{\text{n-i个元素}}$)
依次前移一个位置

$A[j-1]=A[j]$

正常情况下需要做的工作:

- (1). 将第 $i+1$ 个元素至第 n 个元素依次前移一个位置;
- (2). 修改表的长度(表长减1)。

需要考虑的异常情况:

- (1). 是否表空? ($n=0?$)
- (2). 删除位置是否合适? (正常位置: $1 \leq i \leq n$)

约定

若删除成功，算法返回1，
否则，算法返回-1。

算法

```
int DELETELIST( ElemType A[ ], int &n, int i )
{
    int j;
    if(n==0 || i<1 || i>n)
        return(-1);
    for( j=i; j<n; j++ )
        A[j-1]=A[j];
    n--;
    return(1);
}
```

测表空和位置合适否

/* 删除失败 */

/* 元素依次前移一个位置 */

/* 线性表的长度减1 */

/* 删除成功 */

算法时间复杂度分析

若 p_i 为删除线性表中第 i 个数据元素的概率
(设概率相等), 在长度为 n 的线性表中删除第 i
个数据元素需要移动其他的元素的平均次数为:

$$T_{ds} = \sum_{i=1}^n p_i(n-i) = \sum_{i=1}^n (n-i)/n = (n-1)/2$$

称该算法的时间复杂度为 $O(n)$ 。

三、线性表的顺序存储结构的特点

1. 优点

$$LOC(a_i) = LOC(a_1) + (i-1) \times k$$

- (1) 构造原理简单、直观，易理解。
- (2) 元素的存储地址可以通过一个简单的解析式计算出来。是一种随机存储结构，存储速度快。
- (3) 由于只需存放数据元素本身的信息，而无其他空间开销，相对链式存储结构而言，存储空间开销小(仅此而已!)

2. 缺点

- (1) 存储分配需要事先进行。
- (2) 需要一片地址连续的存储空间。
- (3) 基本操作(如插入、删除)的时间效率较低。

$O(n)$

例

已知长度为 n 的线性表 A 采用顺序存储结构, 并且数据元素按值大小非递减排列, 写一算法, 在该线性表中插入一个数据元素 $item$, 使得线性表仍然保持按值非递减排列。

如何找到插入位置

$$a_i \leq a_{i+1} \quad 1 \leq i \leq n-1$$

$a_1, a_2, a_3, \dots, \overbrace{a_i, a_{i+1}, a_{i+2}, \dots, a_n}^{item}$

↑ 依次后移一个位置

特殊情况

```
for(j=n-1; j>=i-1; j--)  
    A[j+1]=A[j];  
A[i]=item;  
n++;
```

$A[n] \leftarrow item$

需要做的工作

1. 特殊位置：直接将item插入表的末尾。
2. 寻找插入位置：
 - (1) 从表的第一个元素开始进行比较，若有关系 $item < A[i]$
则找到插入位置为表的第i个位置。
 - (2) 将第i个元素至第n个元素依次后移一个位置；
 - (3) 将item插入表的第i个位置；
3. 表的长度增1。

算法

```
void INSERT( ElemType A[ ], int &n, ElemType item )
```

```
{
```

```
    int i,j;
```

```
    if (item >= A[n-1])
```

```
        A[n]=item;
```

```
    else {
```

```
        i=0;
```

```
        while (item >= A[i])
```

```
            i++;
```

```
        for (j=n-1; j>=i; j--)
```

```
            A[j+1]=A[j];
```

```
        A[i]=item;
```

```
    }
```

```
    n++;
```

```
}
```

特殊情况

/* 插入表的末尾 */

/* 寻找item的合适位置 */

确定插入位置

/* 插入表的第i个位置 */

表长加1

关于参数表中的&

```
void FUN1( ... )  
{  
    .....  
    n=10;  
    FUN2(n);  
    printf("\n%d",n);  
    .....  
}
```

调用算法

```
void FUN2( int &n )  
{  
    .....  
    n++;  
    .....  
}
```

被调用算法

调用

2.3 线性表的链式存储结构

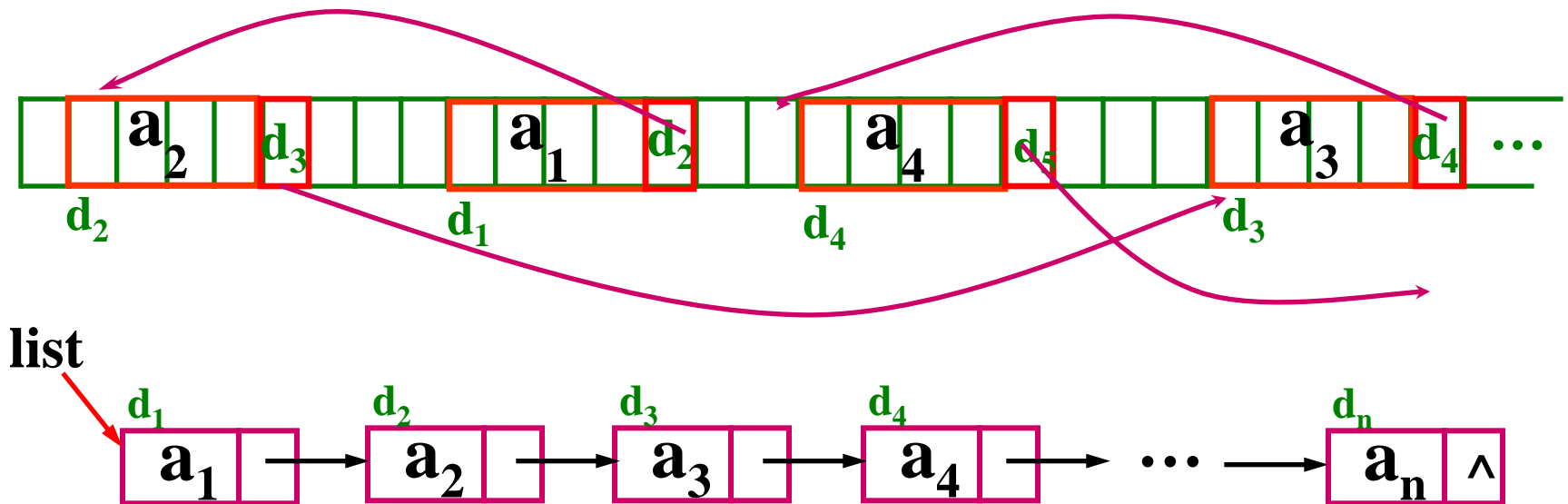
将要讨论的内容:

1. 线性链表的构造原理
2. 几个常用符号的说明
3. 线性链表的有关(操作)算法

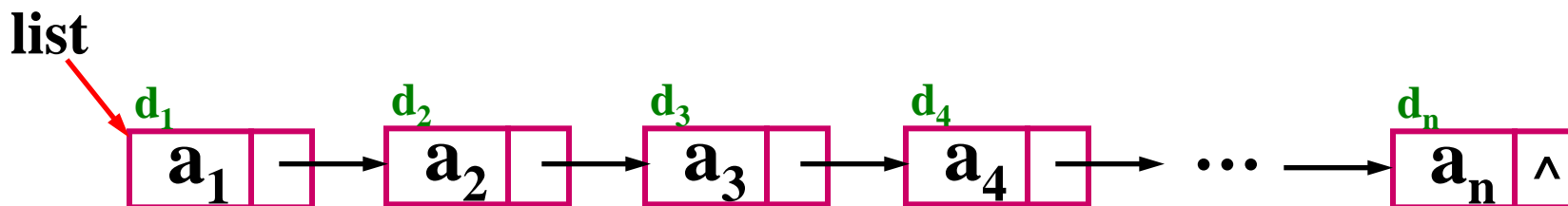
一. 构造原理

用一组地址任意的存储单元(连续的或不连续的)依次存储表中各个数据元素, 数据元素之间的逻辑关系通过**指针**间接地反映出来。

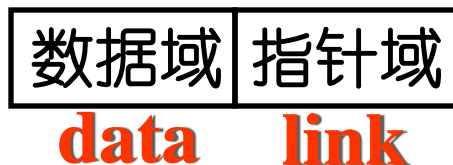
$(a_1, a_2, a_3, a_4, \dots, a_{n-1}, a_n)$



线性表的这种存储结构又称为**单链表**,
或者**线性链表**, 其一般形式为:



一个链结点



链结点:



data link

```
typedef struct node {  
    ElemType data;  
    struct node *link;  
} LNode, *LinkList;
```

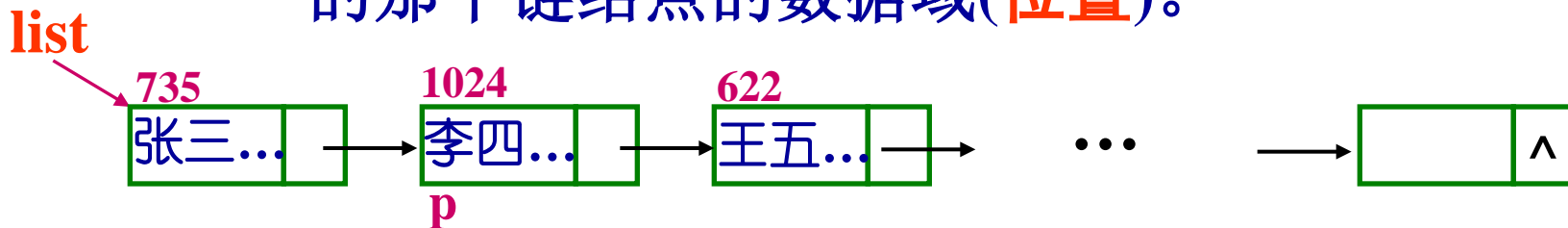
LinkList list, p;

二. 关于符号说明

若指针变量p为指向链表中某结点的指针(即p的内容为链表中某链结点的地址), 则

p->data

若出现在表达式中, 表示由p指的链结点的数据域中的**内容**; 否则, 表示由p所指的那个链结点的数据域(**位置**)。



$X = P \rightarrow data$; $X = ?$ $X = \text{“李四”}$

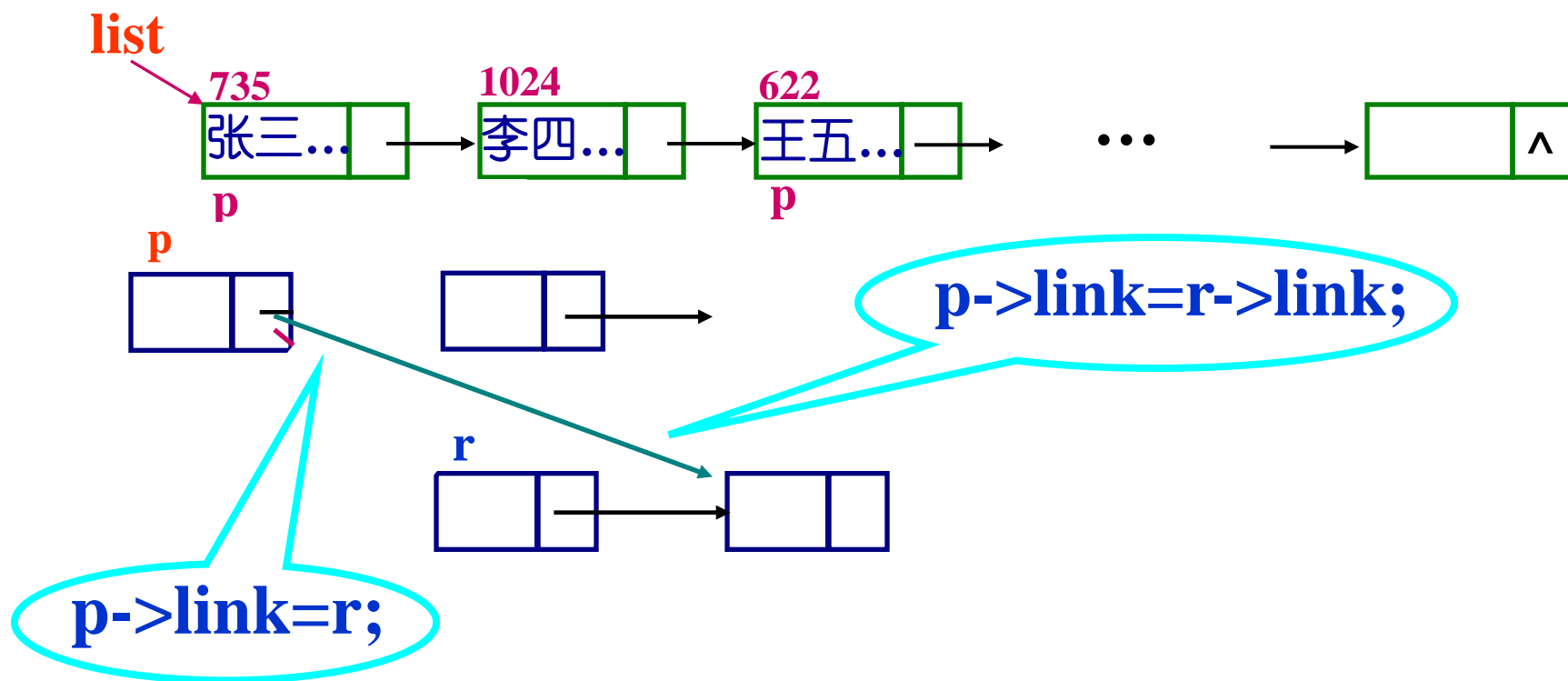
$P \rightarrow data = \text{“刘涛”}$; 1024单元data域中的内容是?

p->link

若出现在表达式中, 表示由p指的链结点的指针域的**内容**, 即p所指的链结点的下一个链结点的地址, 否则, 表示由p所指的那个链结点的指针域(**位置**)。

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

将指针变量 p 由指向链表中某一个链结点改变为指向下一个链结点。



指针变量注意事项：

- 1) 使用新指针变量或生成一个新结点前先执行申请新结点操作
- 2) 不要引用NULL指针（先判断再使用）
- 3) 不要引用已经FREE了的指针

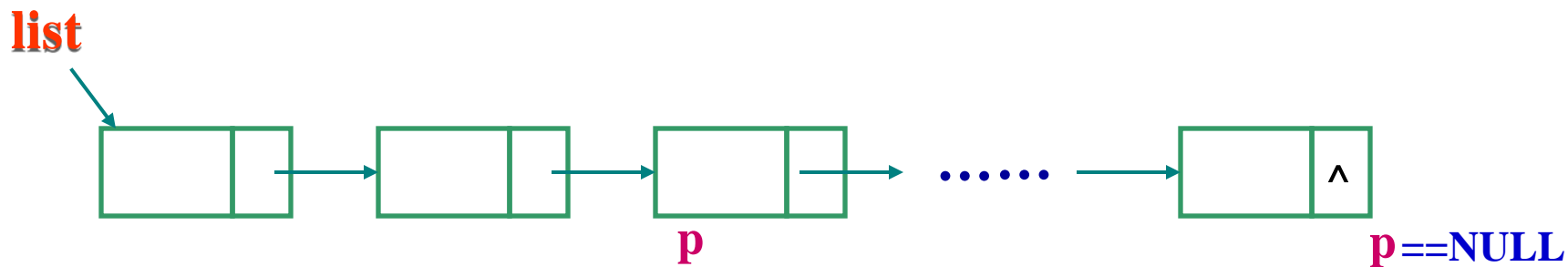
三. 链表的基本操作

- 求线性表的长度。
- 建立一个线性链表。
- 在非空线性链表的第一个结点前插入一个数据信息为`item` 的新结点。
- 在线性链表中由指针`q` 指出的结点之后插入一个数据信息为`item`的链结点。
- 在线性链表中第`i`个结点后面插入一个数据信息为`item`的链结点。
- 从非空线性链表中删除链结点`q`(`q`为指向被删除链结点的指针)。

- 删除线性链表中满足某个条件的链结点。
- 线性链表的逆转。
- 将两个线性链表合并为一个线性链表。
- 检索线性链表中的第 i 个链结点。

.....

1. 求线性链表的长度



初始:

n=0;

p=p->link;

n++;

链表长度

算法

非递归算法

```
int LENGTH( LinkList list )
{
    LinkList p=list;
    int n=0;                /* 链表的长度置初值0 */
    while (p!=NULL) {
        p=p->link;
        n++;
    }
    return(n);              /* 返回链表的长度n */
}
```

时间复杂度 $O(n)$

```
typedef struct node {
    datatype data;
    struct node *link;
} Lnode, *LinkList;
```

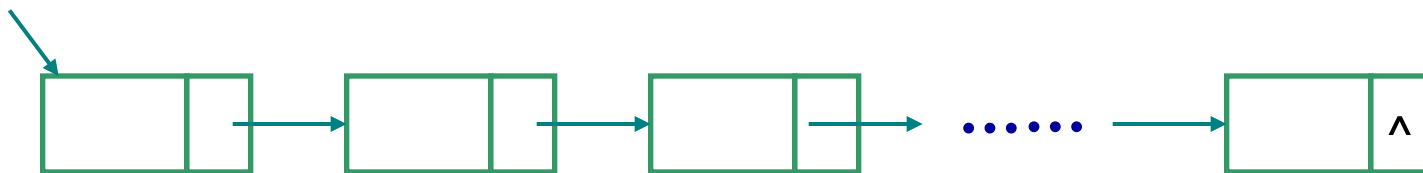
递归算法

递归算法的时间效率通常比非递归算法要低！



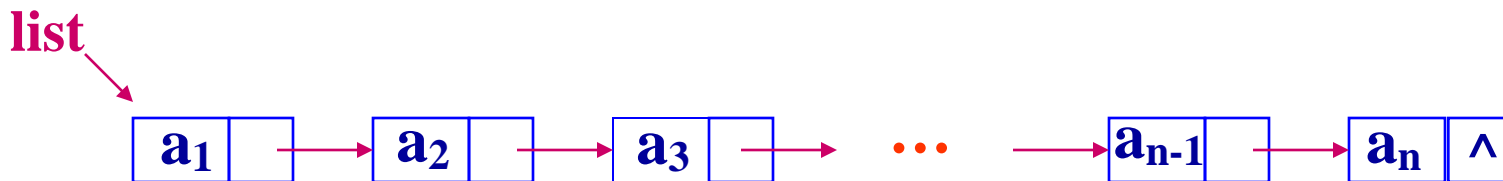
```
int LENGTH( LinkList list)
{
    if(list!=NULL)
        return 1+LENGTH(list->link);
    else
        return 0;
}
```

list



2. 建立一个线性链表

$(a_1, a_2, a_3, \dots, a_{n-1}, a_n)$



申请一个链结点的空间

```
p=(LinkedList)malloc( sizeof(LNode));
```

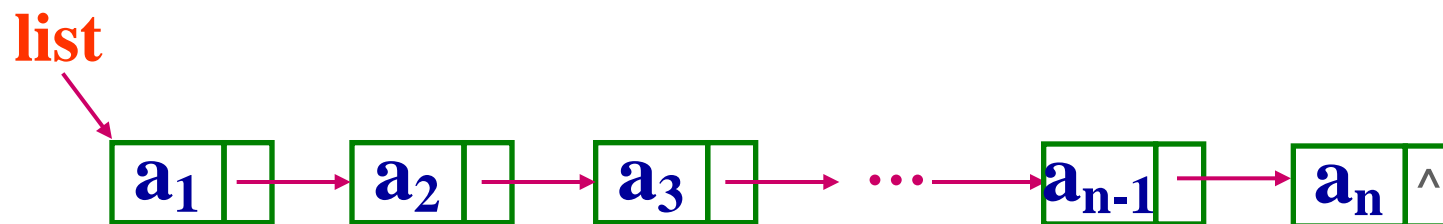
释放一个链结点的空间

```
free(p);
```

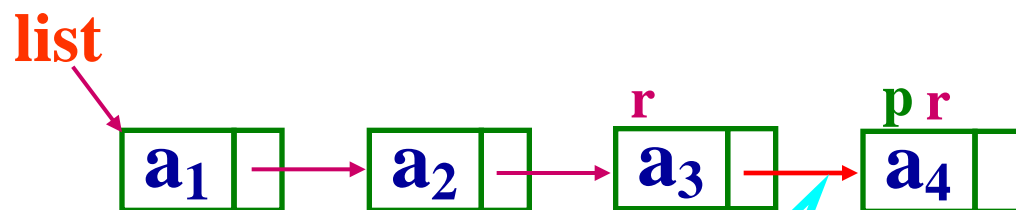
```
#include <alloc.h>
```

```
或#include <stdlib.h>
```


$(a_1, a_2, a_3, a_4, \dots, a_{n-1}, a_n)$



一个结点的插入过程



$r \rightarrow \text{link} = p;$



```
LinkedList CREATE( int n )
```

```
{
```

```
    LinkedList list, p, r;
```

```
    list=NULL;
```

```
    datatype a;
```

```
    for(i=1;i<=n;i++){
```

```
        READ(a);
```

```
        p=(LinkedList)malloc(sizeof(LNode));
```

```
        p->data=a;
```

```
        p->link=NULL;
```

```
        if (list==NULL)
```

```
            list=p;
```

```
        else
```

```
            r->link=p;
```

```
        r=p;
```

```
    }
```

```
    return(list);
```

```
}
```

/* 创建一个空链表 */

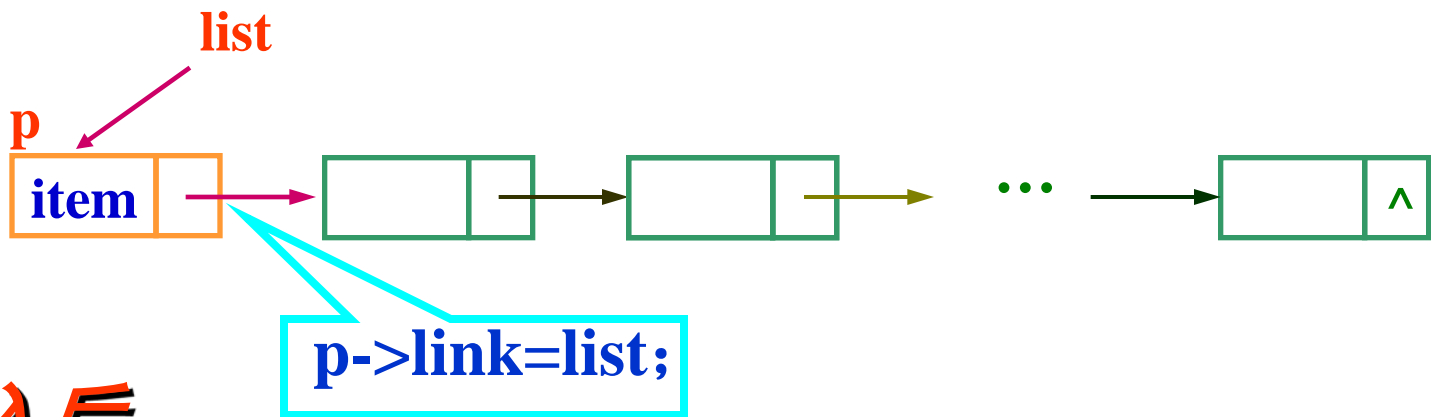
/* 取一个数据元素 */

申请一个新的链结点

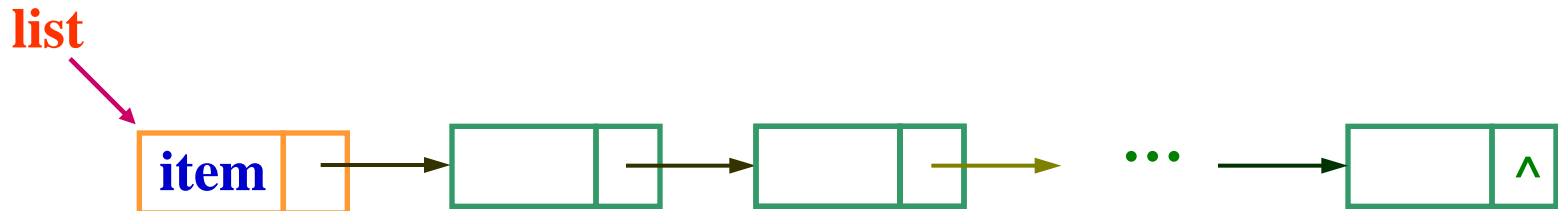
/* 将新结点链接在链表尾部 */

时间复杂度 $O(n)$

3. 在非空线性链表的第一个结点前插入一个数据信息为item的新结点



插入后



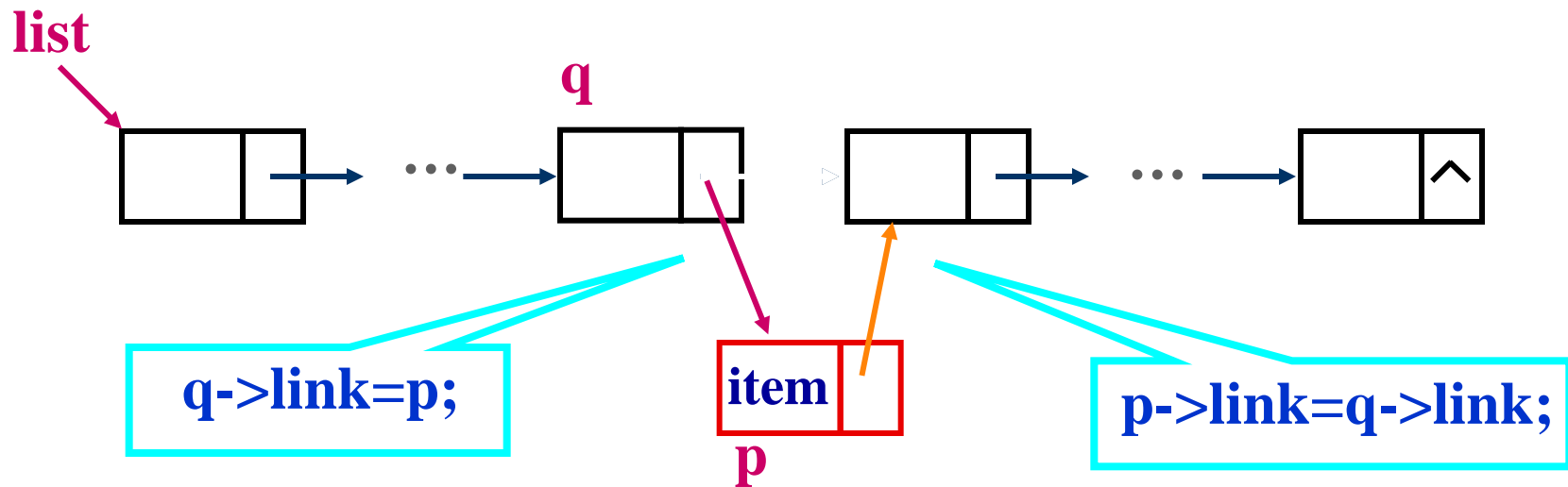
算法

```
void INSERTLINK1( LinkList &list, ElemType item )
{
    // list指向链表第一个链结点 //
    p=(LinkList)malloc(sizeof(LNode));
    p->data=item;                /* 将item送新结点数据域 */
    p->link=list;                /* 将list送新结点指针域 */
    list=p;                      /* 修改指针list的指向 */
}
```

时间复杂度 $O(1)$

4. 在线性链表中由指针q 指的链结点之后插入一个数据信息为item 的链结点

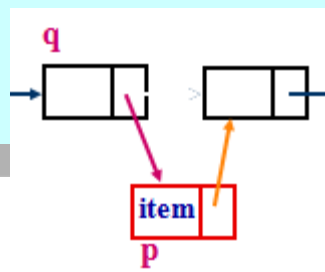
插入过程



算法

```
void INSERTLINK2( LinkList &list, LinkList q,  
                  ElemType item )  
{   LinkList p;  
    p=(LinkList)malloc(sizeof(LNode));  
    p->data=item;                        /* 将item送新结点数据域 */  
    if (list==NULL) {                  /* 若原链表为空 */  
        list=p;  
        p->link=NULL;  
    }  
    else{                               /* 若原链表为非空 */  
        p->link=q->link;  
        q->link=p;  
    }  
}
```

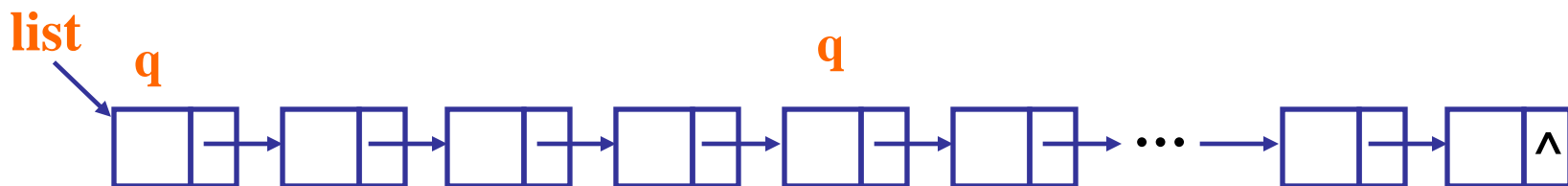
时间复杂度 $O(1)$



5. 在线性链表中第 i ($i > 0$) 个结点后面插入一个数据信息为item的新结点

寻找第 i 个结点

如何找到第 i 个结点?



$q = q \rightarrow \text{link};$

执行 $i-1$ 次



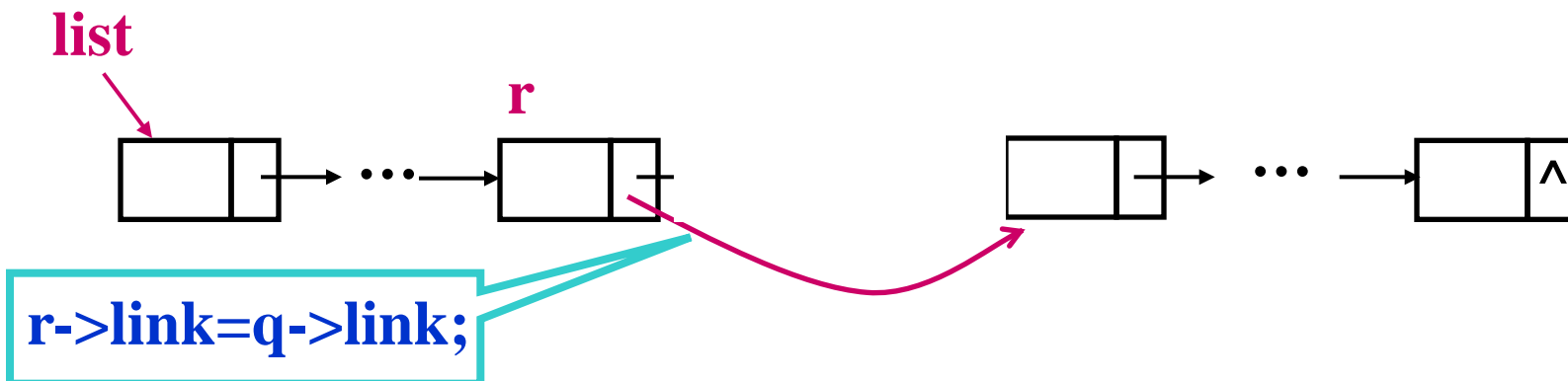
```
void INSERTLINK3( LinkList list, int i,
                  ElemType item )
{
    LinkList p,q=list;
    int j;
    if(q==NULL)
        return -1; /*空表*/
    for(j=1;j<=i-1;j++){ /*寻找第i个结点*/
        q=q->link;
        if(q==NULL)
            return -1; } /*不存在第i个结点*/
    p=(LinkList)malloc(sizeof(LNode));
    p->data=item; /*将item送新结点数据域*/
    p->link=q->link;
    q->link=p; /*将新结点插入到第i个结点之后*/
}
```

时间复杂度 $O(n)$

6. 从非空线性链表中删除q指的链结点, 设q的直接前驱结点由r指出



情况1：删除链表的第一个结点



情况2：删除链表中非第一个结点



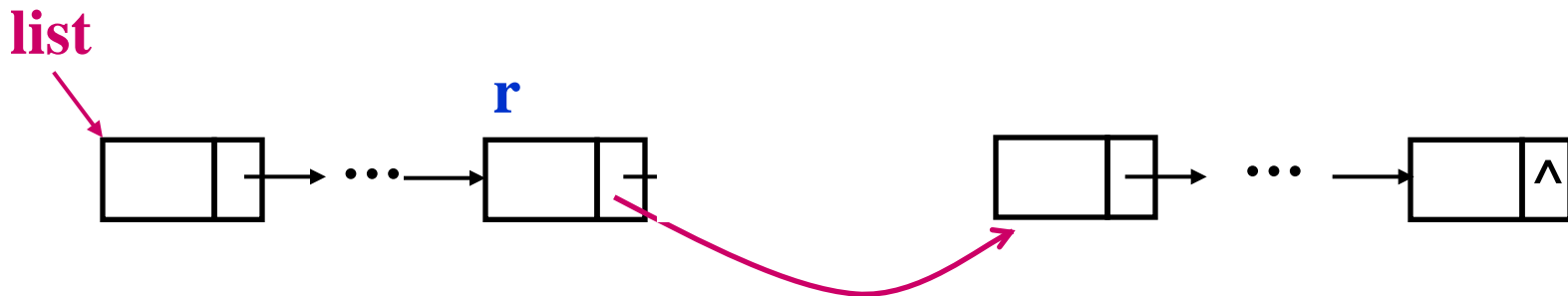
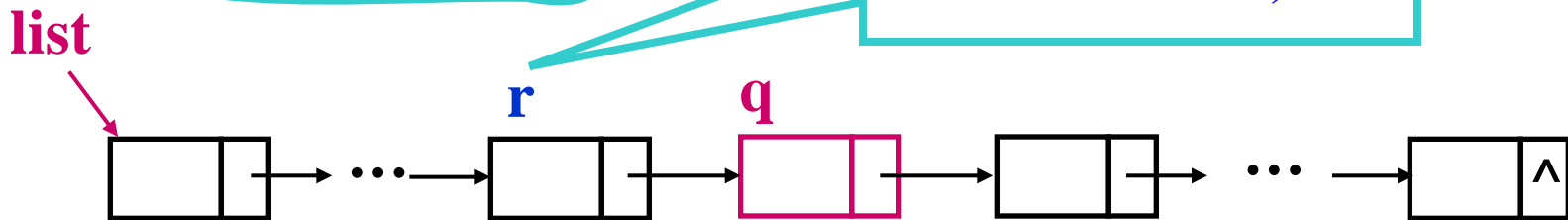
```
void DELETEDLINK1( LinkList &list, LinkList r,
                  LinkList q )
{
    if (q==list)
        list=q->link;           /* 删除链表的第一个链结点*/
    else
        r->link=q->link;        /* 删除q指的链结点*/
    free(q);                     /* 释放被删除的结点空间*/
}
```

时间复杂度 $O(1)$

7. 从非空线性链表中删除q指的链结点, 设q的直接前驱结点由r指出

```
r=list;  
while(r->link!=q)  
    r=r->link;
```

```
r=r->link;
```





```
void DELETELINK2( LinkList &list, LinkList q )
```

```
{  
    LinkList r;  
    if (q==list) {                               /*当删除链表第一个结点*/  
        list=list->link;  
        free(q);                                /*释放被删除结点的空间*/  
    }  
    else {  
        r=list;  
        while ( r->link!=q && r->link!=NULL)    /*移向下一个链结点*/  
            r=r->link;  
        if (r->link!=NULL) {  
            { r->link=q->link;  
              free(q);                            /*释放被删除结点的空间*/  
            }  
            寻找q结点的直接前驱r  
        }  
    }  
}
```

时间复杂度 $O(n)$

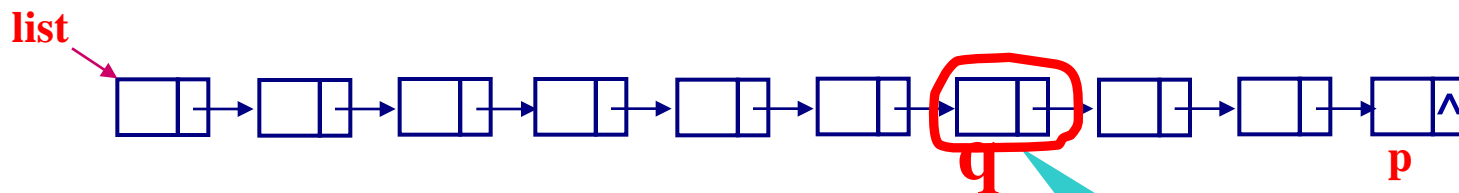
例

2009年硕士研究生入学考试计算机专业基础综合全国联考试题

请写一算法，该算法用尽可能高的时间效率找到由list所指的线性链表的倒数第k个结点。若找到这样的结点，算法给出该结点的地址，否则，给出NULL。

限制

1. 算法中不得求出链表长度；
2. 不允许使用除指针变量和控制变量以外的其他辅助空间。



`p=p->link;`

`q=q->link;`

步骤

1. 设置一个指针变量 p ，初始时指向链表的第1个结点；
2. 然后令 p 后移指向链表的第 k 个结点；
3. 再设置另一个指针变量 q ，初始时指向链表的第1个结点；
4. 利用一个循环让 p 与 q 同步沿链表向后移动；当 p 指向链表最后那个结点时， q 指向链表的倒数第 k 个结点。

LinkedList SEARCHNODE(LinkedList list,int k)

```
{
    LinkedList p,q;
    int i;
    if(list!=NULL && k>0){
        p=list;
        for(i=1;i<k;i++){    /* 循环结束时， p指向链表的第k个结点 */
            p=p->link;
            if(p==NULL){
                printf("链表中不存在倒数第k个结点！ ")
                return NULL;
            }
        }
        q=list;
        while(p->link!=NULL){
            p=p->link;
            q=q->link;
        }
        return q;    /* p指向链表最后那个结点， q指向倒数第k个结点 */
    }
}
```

/* 给出链表倒数第k个结点(q指向的那个结点)的地址 */



步骤

$O(n)$

1. 设置一个指针变量 p ，初始时指向链表的第1个结点；
- ② 然后令 p 后移指向链表的第 k 个结点； 执行 $k-1$ 次
3. 再设置另一个指针变量 q ，初始时指向链表的第1个结点； 执行 $n-k$ 次
- ④ 利用一个循环让 p 与 q 同步沿链表向后移动；当 p 指向链表最后那个结点时， q 指向链表的倒数第 k 个结点。

若用 n 表示链表中结点的个数，
对于任意 k ($1 \leq k \leq n$)

四、链式存储结构的特点

1. 优点

- (1) 存储空间动态分配，可以根据实际需要使用。
- (2) 不需要地址连续的存储空间。
- (3) 插入/删除操作只须通过修改指针实现，不必移动数据元素，操作的时间效率较高



无论位于链表何处，无论链表的长度如何，插入和删除操作的时间都是 $O(1)$

2. 缺点

- (1) 每个链结点需要设置指针域（存储密度小）。
- (2) 是一种非随机存储结构，查找、定位等操作要通过顺序扫描链表实现，时间效率较低。

时间为 $O(n)$

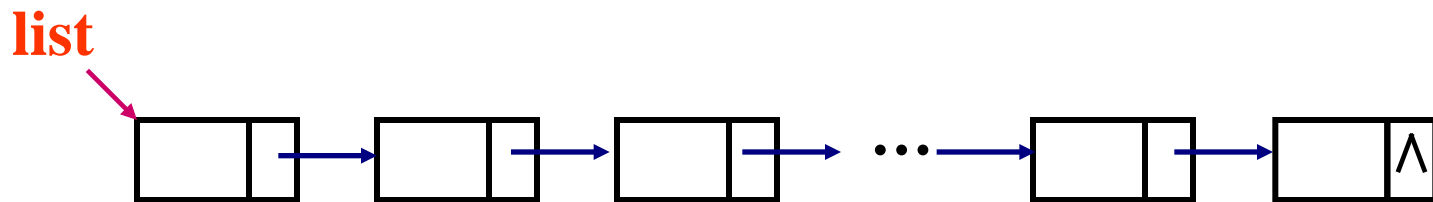
思考题

1. 有人说，线性表的顺序存储结构比链式存储结构的存储开销要少，也有人说，线性表的链式存储结构比顺序存储结构的存储开销要少，你是如何看待这两种说法的？
2. 线性表可以采用顺序存储结构，也可以采用链存储结构，在实际问题中，应该根据什么原则来选择其中最合适的一种存储结构？

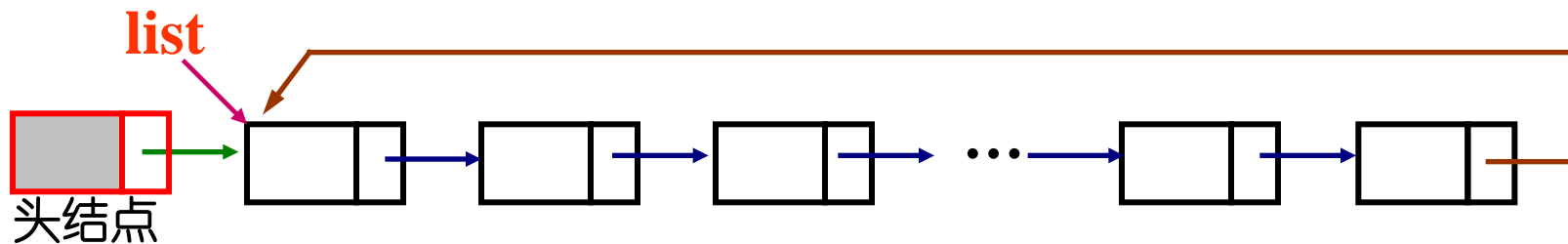
2.4. 循环链表

循环链表 是指链表中最后那个链结点的指针域存放指向链表最前面那个结点的指针，整个链表形成一个环。

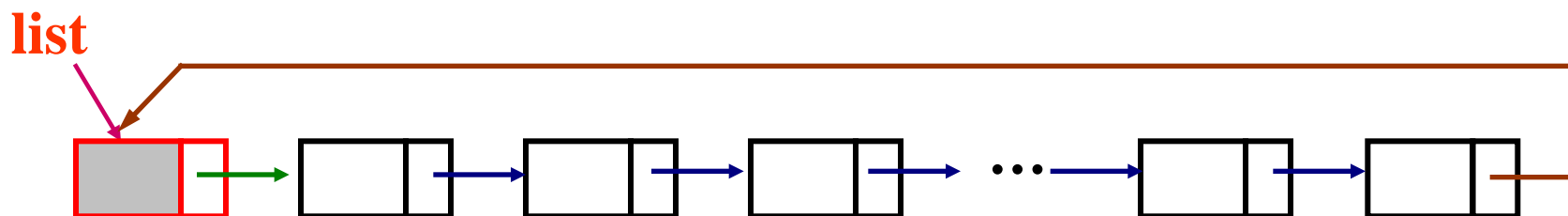
线性链表



循环链表



带有头结点的循环链表

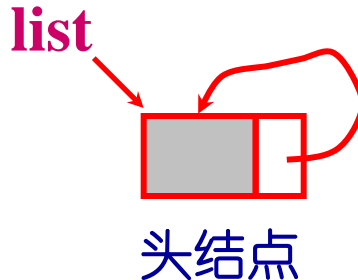


头结点

说明

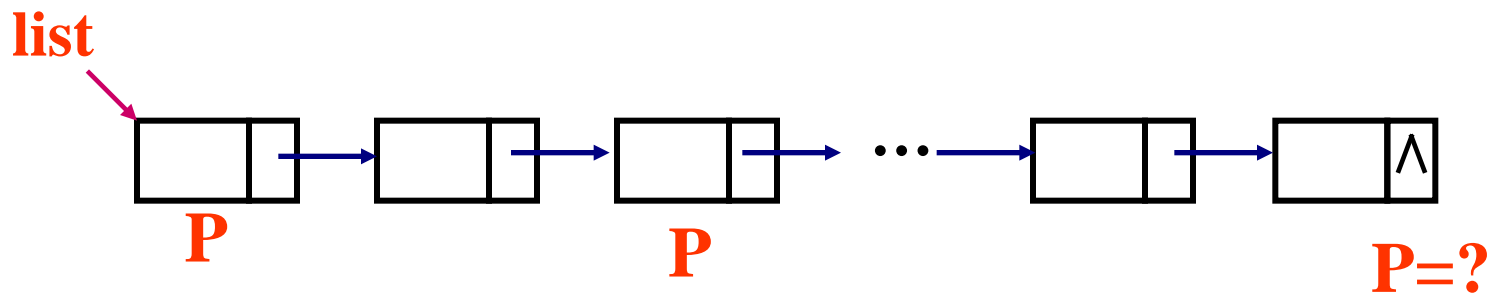
1. 头结点的设置要根据实际需要确定；
2. 对于采用循环链表作为存储结构的线性表，若链表设置了头结点，则判断空表的条件是

`list->link=list`



3. 对于循环链表，如何判断是否遍历了链表一周？

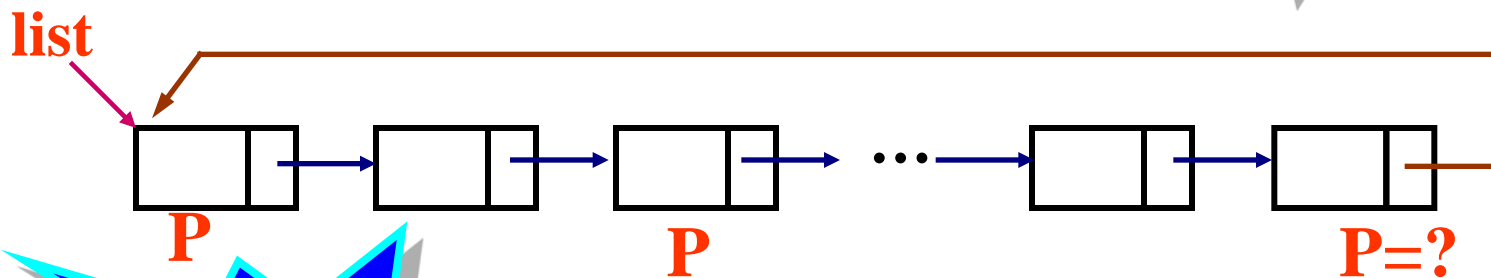
线性链表



P=P->Link;

P=NULL

循环链表



P=List

求线性链表的长度

```
int LENGTH( LinkList list )
{
    LinkList p=list;
    int n=0;          /* 链表的长度置初值0 */
    while (p!=NULL) {
        p=p->link;
        n++;
    }
    return(n);        /* 返回链表的长度n */
}
```

非
循
环
链
表

循
环
链
表

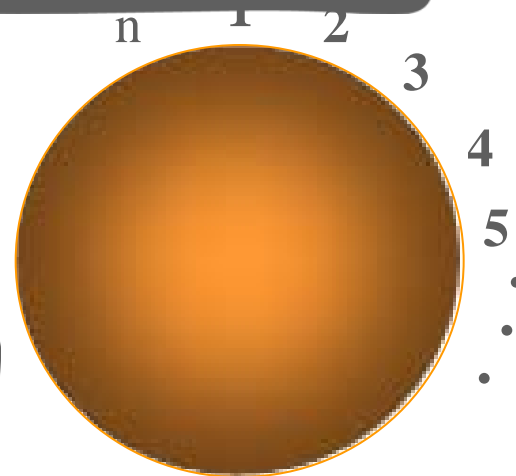
```
int LENGTH( LinkList list )
{
    LinkList p=list;
    int n=0;          /* 链表的长度置初值0 */
    if (p!=NULL){
        do {
            p=p->link;
            n++;
        } While ( p!=List );
    }
    return(n);        /* 返回链表的长度n */
}
```

例：约瑟夫(JOSEPHU)问题

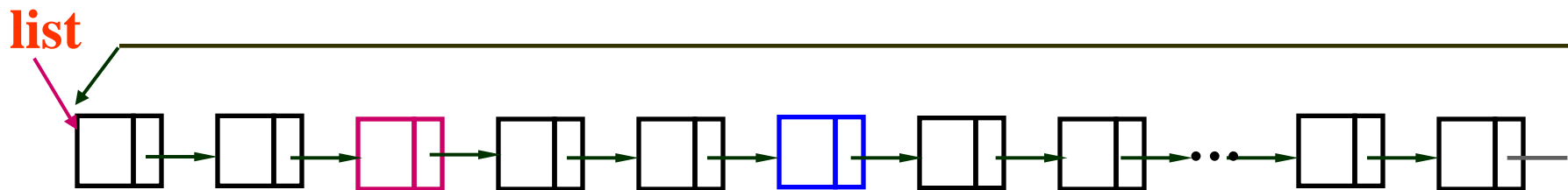
圆桌问题

Flavius Josephus 是公元一世纪的一个著名的历史学家。据传说，如果Josephus没有他的数学才能的话，也许他根本不会出名就死去（活不到他出名）。在犹太人和古罗马人战争期间，39个犹太人与Josephus及他的朋友躲到一个洞中，39个犹太人决定宁愿死也不要被敌人捉到，于是决定了一个自杀方式，41个人排成一个圆圈，由第1个人开始报数，每报数到第3人该人就必须自杀，然后再由下一个重新报数，直到所有人都自杀身亡为止。然而Josephus和他的朋友并不想遵从，Josephus 要他的朋友先假装遵从，他将朋友与自己安排在第16个与第31个位置，于是逃过了这场死亡游戏。

已知 n 个人(不妨分别以编号1, 2, 3, ..., n 代表)围坐在一张圆桌周围，编号为 k 的人从1开始报数，数到 m 的那个人出列，他的下一个人又从1开始继续报数，数到 m 的那个人出列，..., 依此重复下去，直到圆桌周围的人全部出列。直到圆桌周围只剩一个人。



利用一个不带头结点的 循环链表

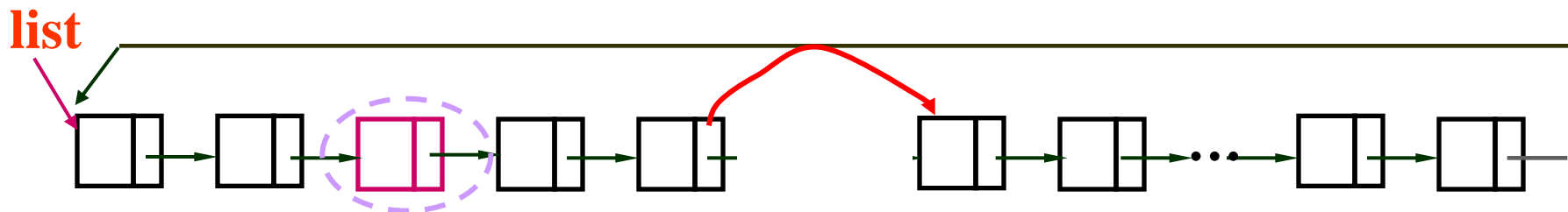


若假设 $k=3$, $m=4$

n : 链表中链结点的个数;
 k : 第一个出发点;
 m : 报数。

需要做的工作：



1. 建立一个不带头结点的循环链表；
2. 找到第一个出发点；
3. 反复删除一个链结点。



若假设 $k=3$, $m=4$

建立一个无头结点的线性链表

```
void CREATE( int n, LinkList &list )
{
    LinkList p,r;
    list=NULL;                                /* 创建一个空链表 */
    datatype a;
    for(i=1;i<=n;i++){
        READ(a);                             /* 取一个数据元素 */
        p=(LinkList)malloc(sizeof(LNode));
        p->data=a;
        p->link=NULL;
        if (list==NULL)
            list=p;
        else
            r->link=p;                         /* 将新结点链接在链表尾部 */
        r=p;
    }
}
```



p->link=list;

算法

```
void JOSEPHU( int n, int k, int m )
```

```
{  LinkList list,p,r;
```

```
  int i;
```

```
  list=NULL;
```

```
  for(i=1;i<=n;i++) {
```

```
    p=(LinkList)malloc(sizeof(LNode));
```

```
    p->data=i;
```

```
    if (list==NULL)
```

```
        list=p;
```

```
    else
```

```
        r->link=p;
```

```
    r=p;
```

```
  }
```

```
  p->link=list;
```

```
  p=list;
```

```
  for(i=1;i<=1;i++) {
```

```
    r=p;
```

```
    p=p->link;
```

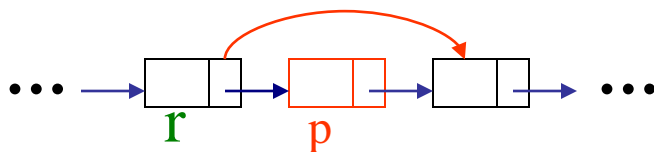
```
  }
```

```
/* 找到第一个点 */
```

当 $k \neq 1$, $m=1$ 时

反复寻找并删除结点

```
while(p->link!=p) {
    for(i=1;i<=m-1;i++){
        r=p;
        p=p->link;
    }
    r->link=p->link;
    printf("%3d",p->data);
    free(p);
    p=r->link;
}
printf("%3d", p->data );
free(p);
}
```



主函数

```
#include <alloc.h>
```

```
main( )
```

```
{
```

```
    int n, k, m;
```

```
    printf( “ \nInput n, k, m: ” );
```

```
    scanf( “ %d %d %d ”, &n, &k, &m );
```

```
    JOSEPHU( n, k, m );
```

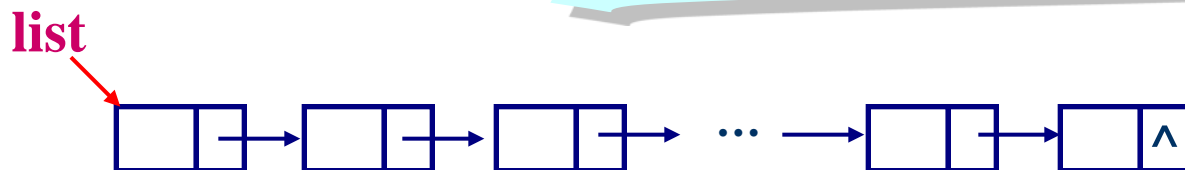
```
}
```

输入链结点总数n、
报数的起始位置k与
报数m。

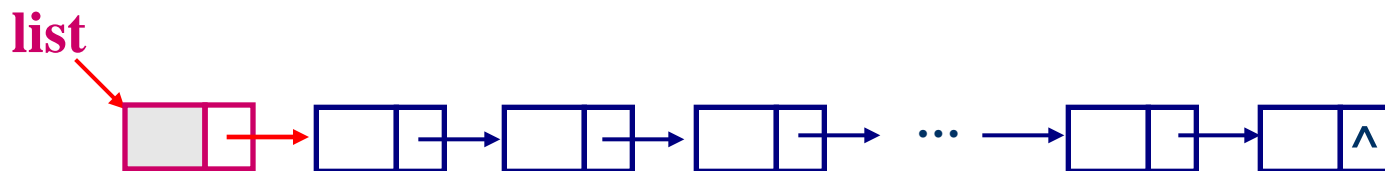
调用约瑟夫函数

线性表的链式存储结构

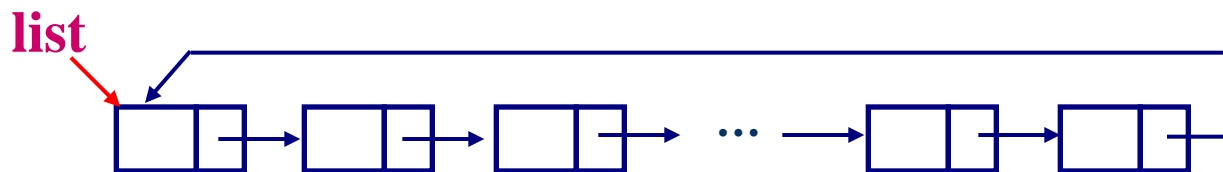
线性链表(单链表)



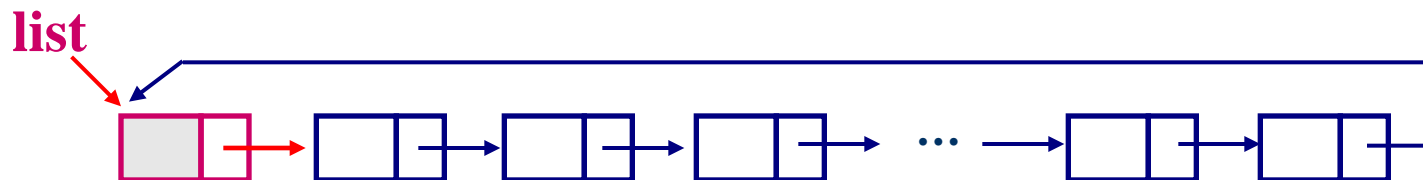
带头结点的线性链表



循环链表



带头结点的循环链表



2.5 双向链表及其操作

本节内容

1. 双向链表的构造
2. 双向链表的插入与删除

一. 双向链表的构造

所谓**双向链表**是指链表的每一个结点中除了数据域以外设置两个指针域，其中之一指向结点的直接前驱结点，另外一个指向结点的直接后继结点。

链结点的实际构造可以形象地描述如下：

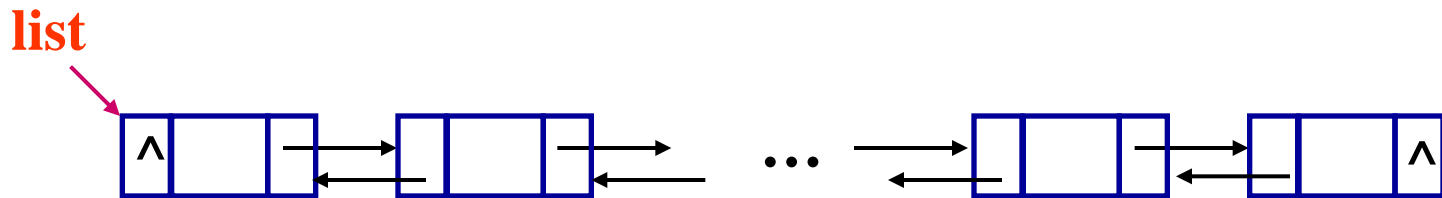


其中，**data** 为数据域

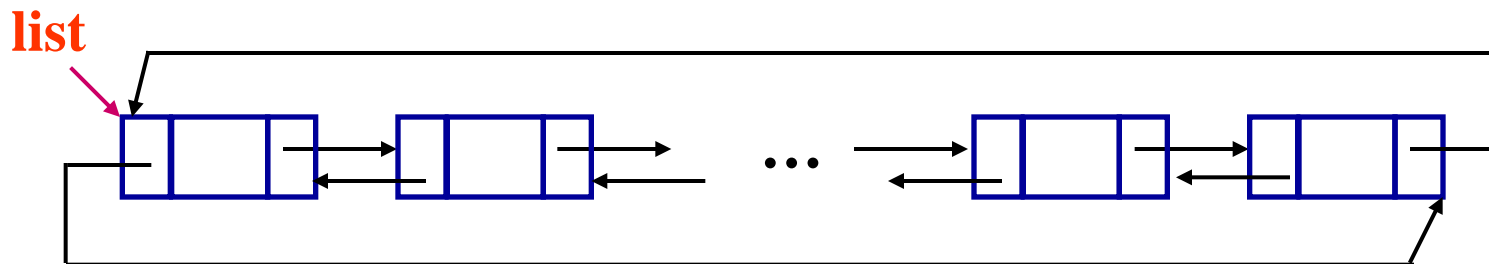
llink, rlink 分别为指向该结点的直接前驱结点与直接后继结点的指针域

分别称为左指针和右指针

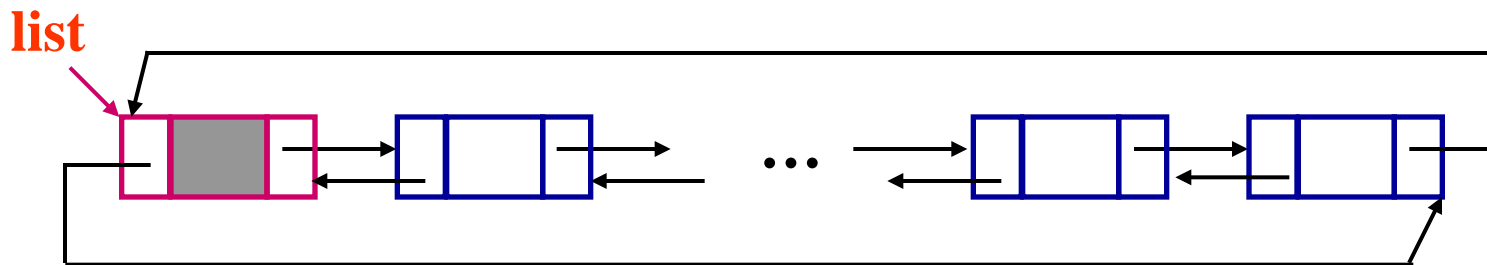
双向链表的几种形式



无头结点的双向链表



无头结点的双向循环链表



带头结点的双向循环链表

类型定义

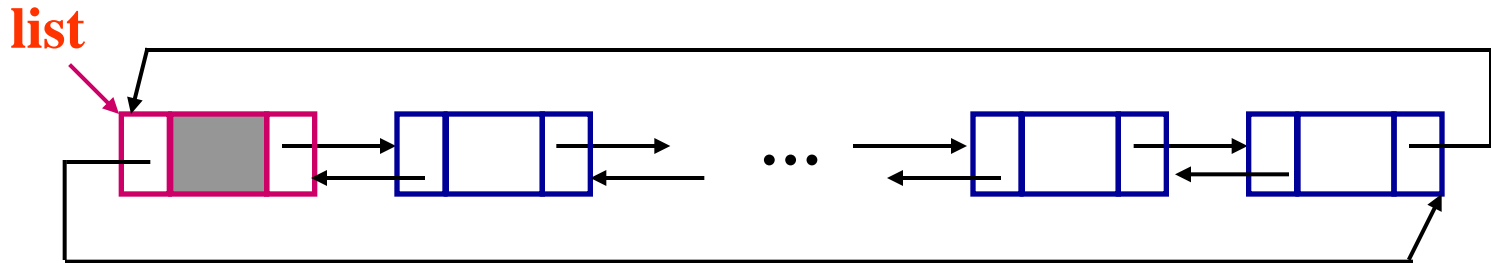
```
typedef struct node {  
    ElemType data;  
    struct node *llink, *rlink;  
}DNode, *DLinkList;
```



二. 双向链表的插入

功能

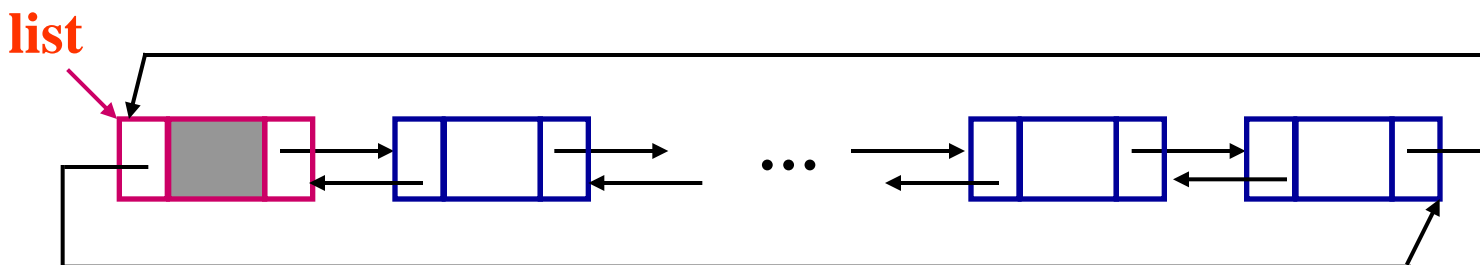
在带有头结点的非空双向循环链表中第一个数据域的内容为 x 的链结点右边插入一个数据信息为 $item$ 的新结点。



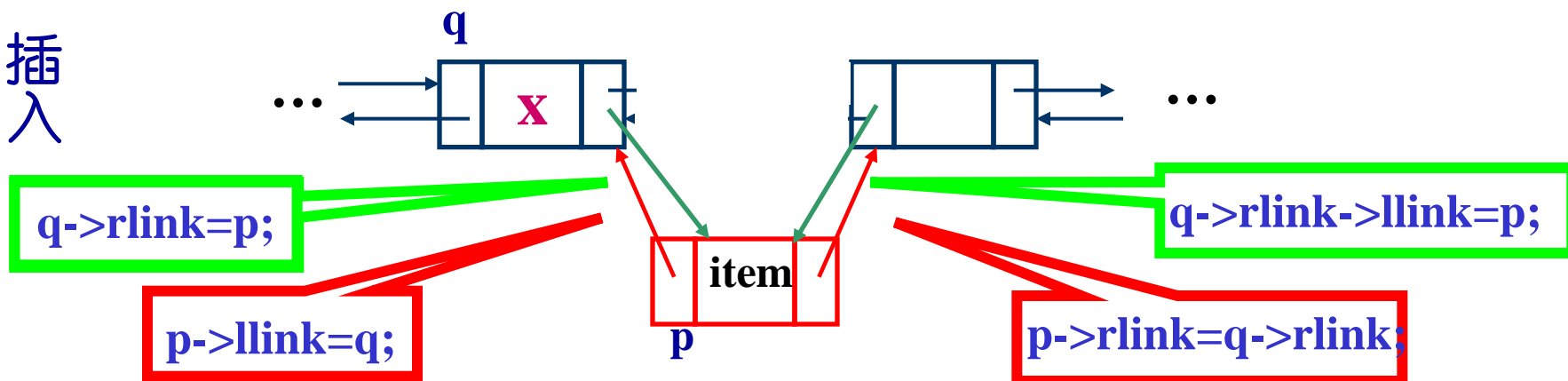
需要做的工作:

1. 找到满足条件的结点;
2. 若找到, 申请一个新的链结点;
3. 将新结点插到满足条件的结点后面。

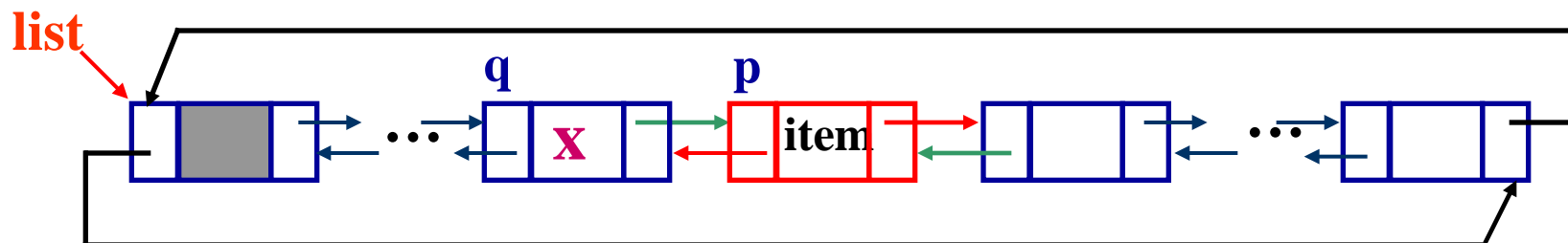
插入前



插入



插入后

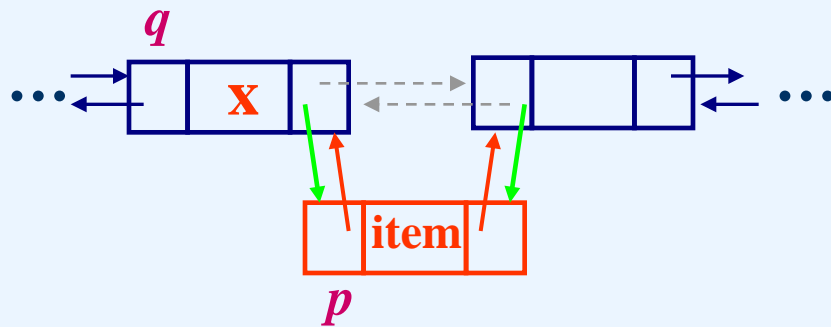




时间复杂度 $O(n)$

```
int INSERTD( DLinkList list, ElemType x, ElemType item )
{
    DLinkList p,q;
    q=list->rlink; /* q 初始指向头结点的下一个结点 */
    while (q!=list && q->data!=x) /* 寻找满足条件的链结点 */
        q=q->rlink;
    if (q==list) /* 没有找到满足条件的结点 */
        return(-1);
    p=(DLinkList)malloc(sizeof(DNode)); /* 申请一个新的结点 */
    p->data=item;
    p->llink=q;
    p->rlink=q->rlink;
    q->rlink->llink=p;
    q->rlink=p;
    return(1); /* 插入成功 */
}
```

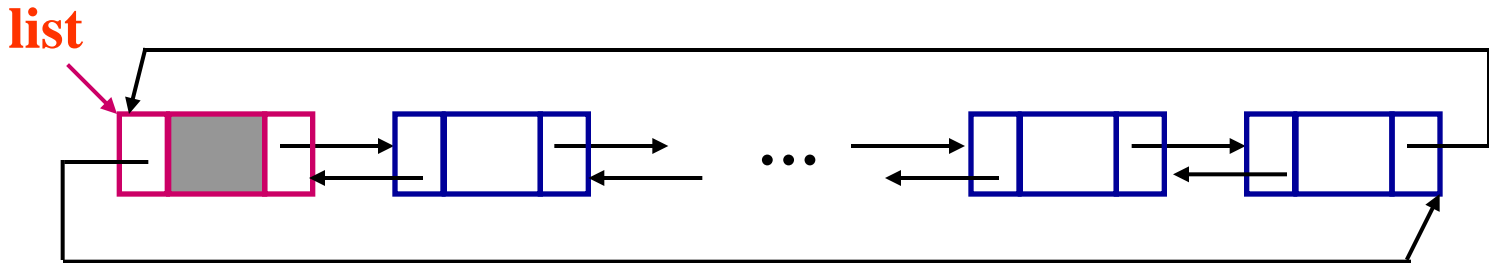
寻找满足条件的结点



二. 双向链表的删除

功能

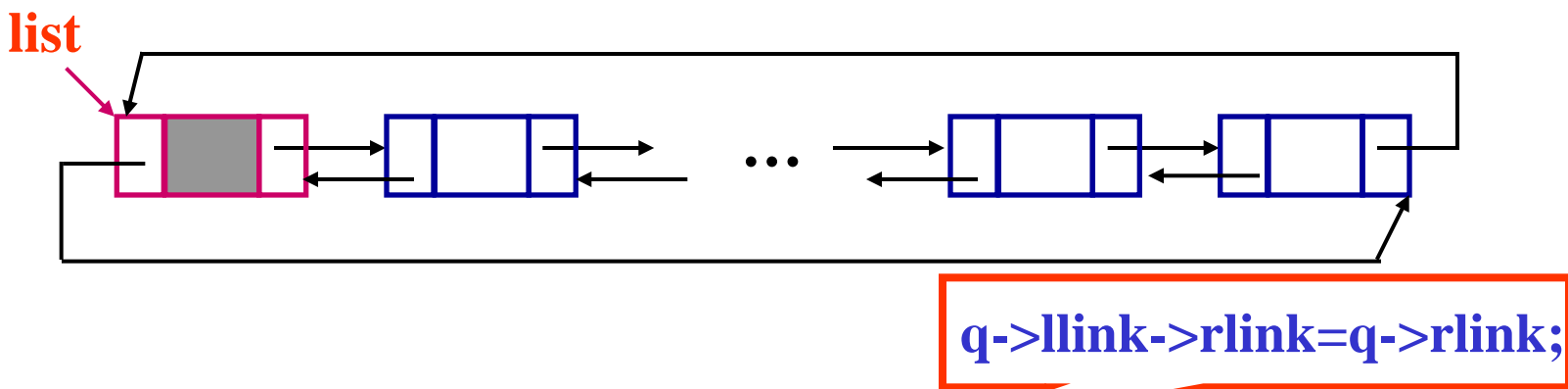
删除带有头结点的非空双向循环链表中第一个数据域的内容为 x 的链结点。



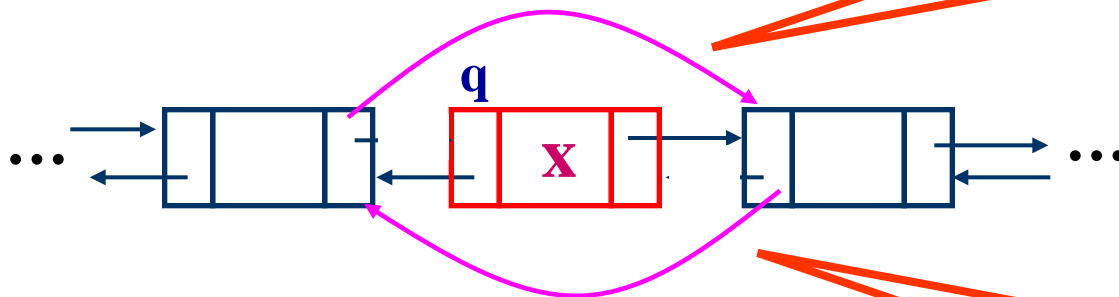
需要做的工作:

1. 找到满足条件的结点;
2. 若找到, 删除(并释放)满足条件的结点。

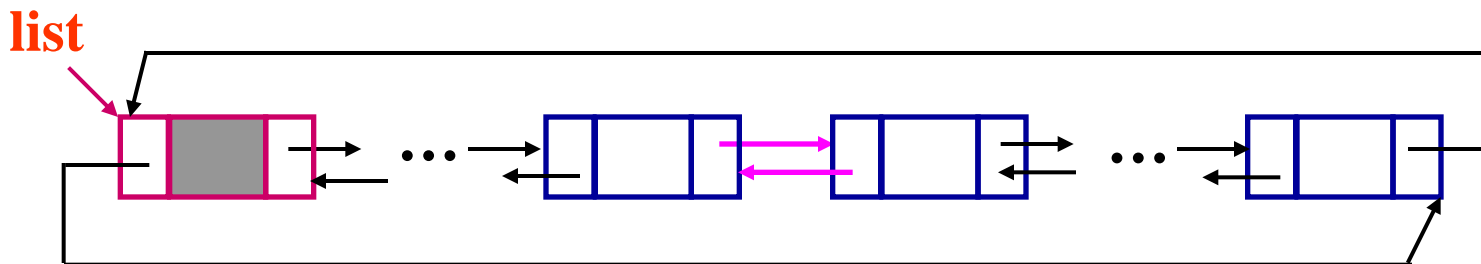
删除前



删除



删除后

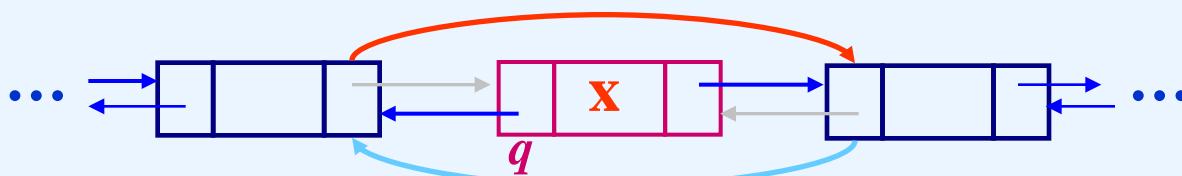


算法

时间复杂度 $O(n)$

寻找满足条件的结点

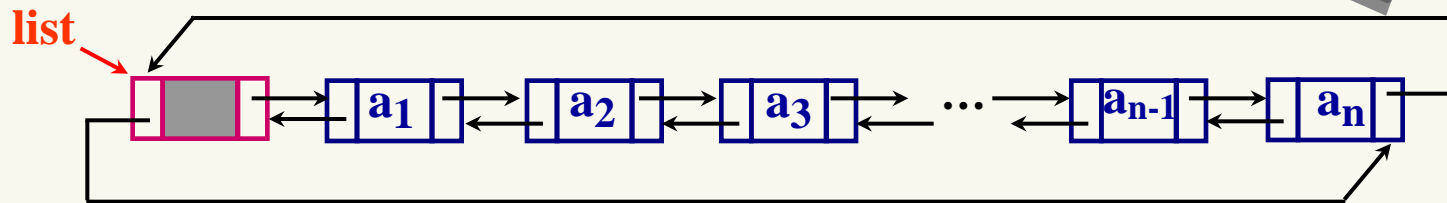
```
int DELETED( DLinkList list, ElemType x )
{
    DLinkList q;
    q=list->rlink;          /* q初始指向头结点的下一个结点*/
    while (q!=list && q->data!=x) /* 找满足条件的链结点 */
        q=q->rlink;
    if (q==list)
        return(-1);          /* 没有找到满足条件的结点 */
    q->llink->rlink=q->rlink;
    q->rlink->llink=q->llink;
    free(q);                 /* 释放被删除的结点的存储空间 */
    return(1);               /* 删除成功 */
}
```



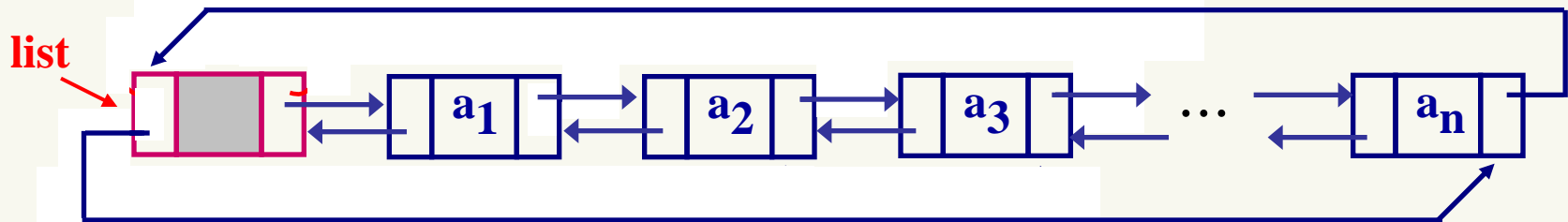
练习

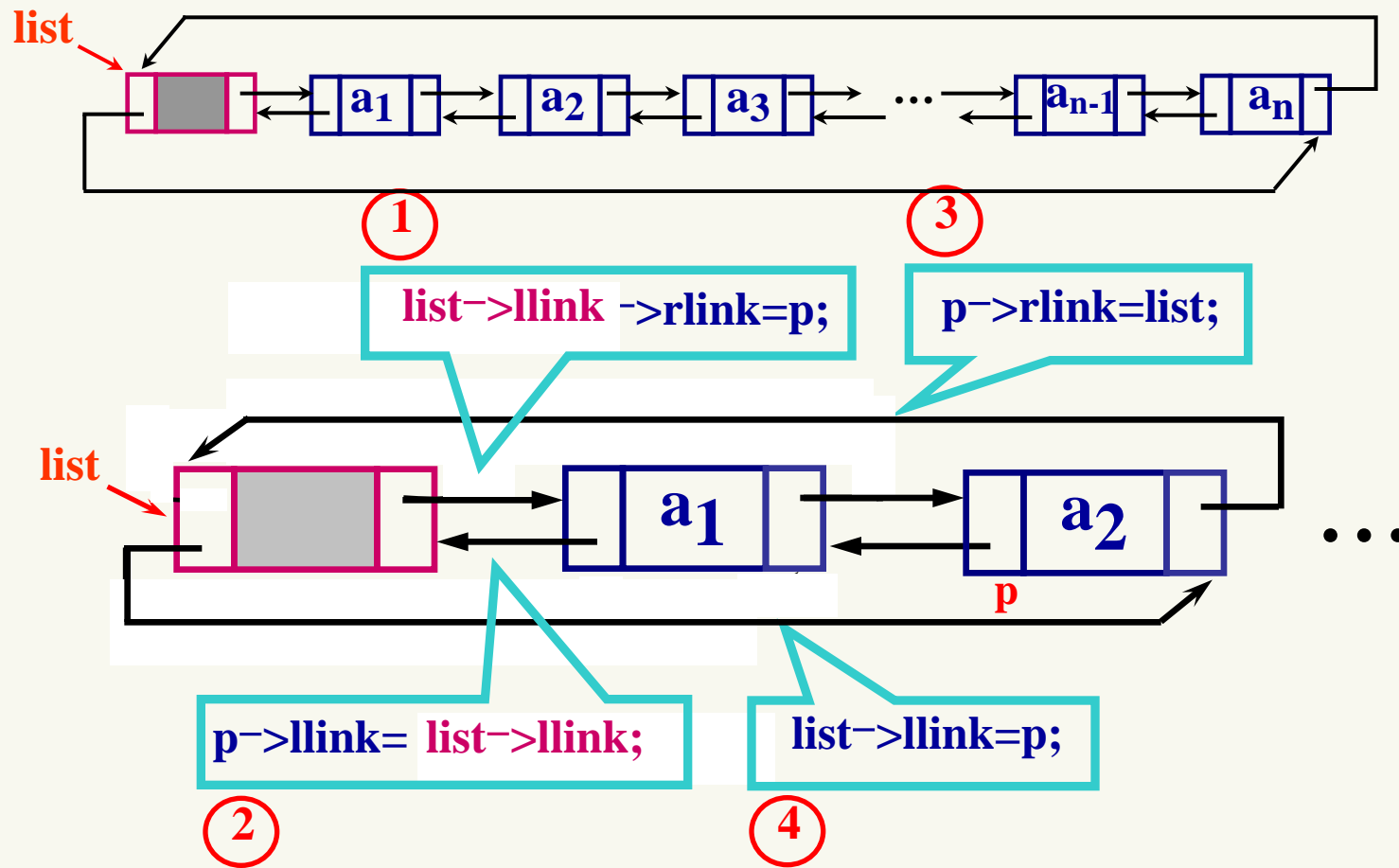
构造一个带头结点的双向循环链表

$(a_1, a_2, a_3, \dots, a_{n-1}, a_n)$



插入

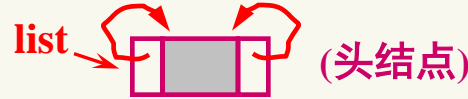




首先构造一个头结点

DLinkedList INITIALDLINK(int n)

```
{
    int i;
    DLinkedList list, p;
    list = (DLinkedList) malloc(sizeof(DNode));
    list->llink = list;
    list->rlink = list;
    for(i=0; i<n; i++){
        p = (DLinkedList) malloc(sizeof(DNode));
        READ(p->data);          /* 读入一元素 */
        INSERTNODE(list, p);
    }
    return list;
}
```



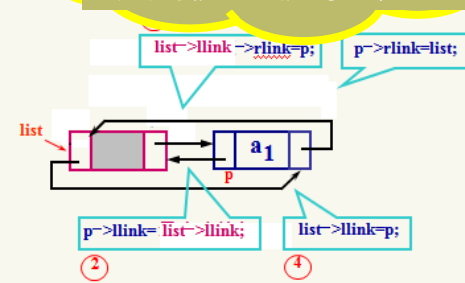
时间复杂度
 $O(n)$

void INSERTNODE(DLinkedList list, List p)

```
{
    list->llink->rlink = p;
    p->llink = list->llink;
    p->rlink = list;
    list->llink = p;
}
```

功能

将指针为p的结点插入到头结点指针为list的双向循环链表中



本章内容小结



线性表的基本概念

- 什么是线性关系？
- 什么是线性表？
- 线性表的基本操作有哪些？其中最重要的有哪些？

线性表的顺序存储结构

- 构造原理。
- 插入、删除操作对应的算法设计。
- 优点、缺点

线性表的链式存储结构

- 线性链表、循环链表和双向链表的构造原理。
- 各种链表中进行插入、删除操作对应的算法设计
- 头结点问题