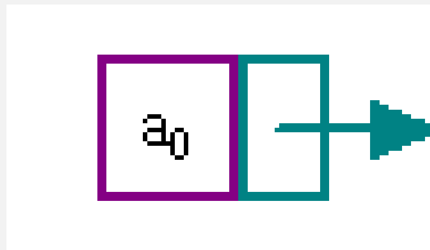
# 第四章　线性表的链式存储(Linked List)

- 线性表的链式存储
- 基于单链表的算法实现
- 链式存储的其他方法
- 基于带头结点的单循环链表算法
- 双向链表的基本算法实现
- 顺序存储与链式存储的比较
- 链式存储的应用举例

# 4.1、线性表的链式存储

- 为什么要使用链式存储结构？
  - 一次分配足够大的空间易造成浪费且容易失败
  - 当频繁进行插入和删除时，链式不需要移动元素

- 使用链式存储结构——链表（Linked List）
  - 结点(Node): 数据元素的存储映像，表示要处理的数据。
    - 数据
    - 指针



# 线性表的链式存储（单链表）

- 单链表 (singly linked list)
  - 结点：记录后继关系
  - 线性表（$a_0$, $a_1$, $a_2$,……$a_n$)

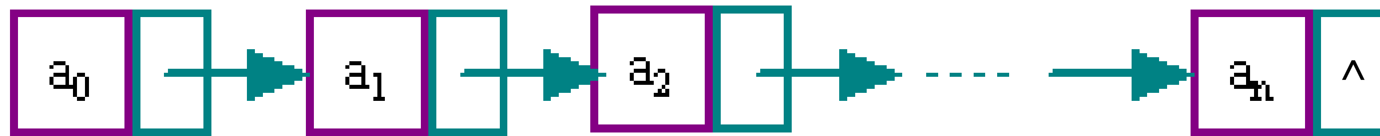
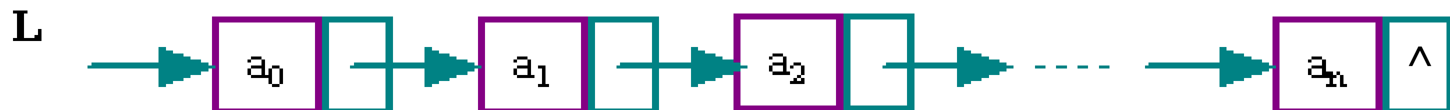



图4-1 单链表结构的逻辑图

不带头结点的单链表：



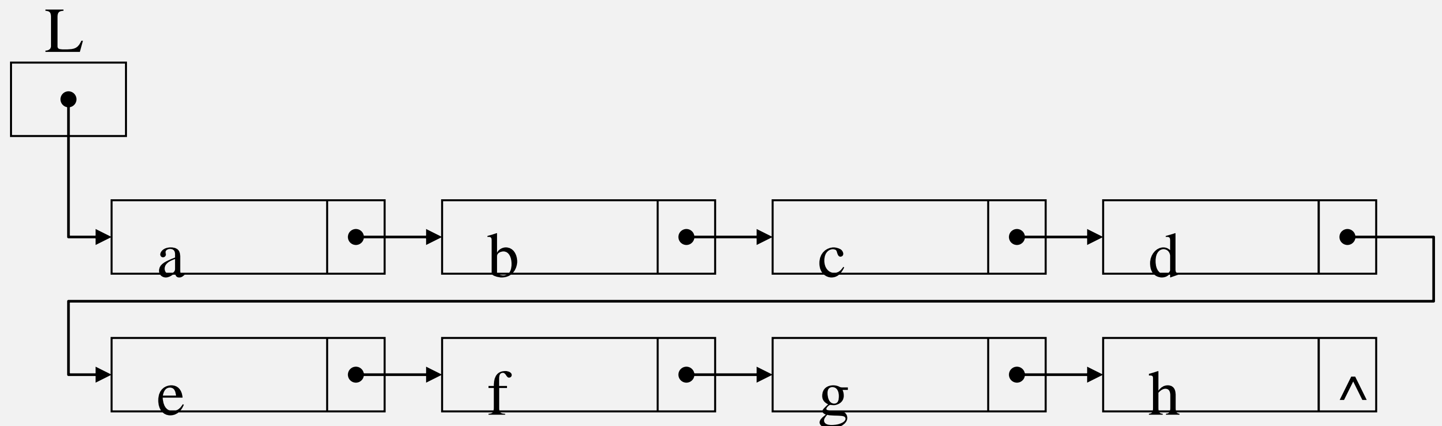
# 线性表的链式存储（举例）

头指针L

31

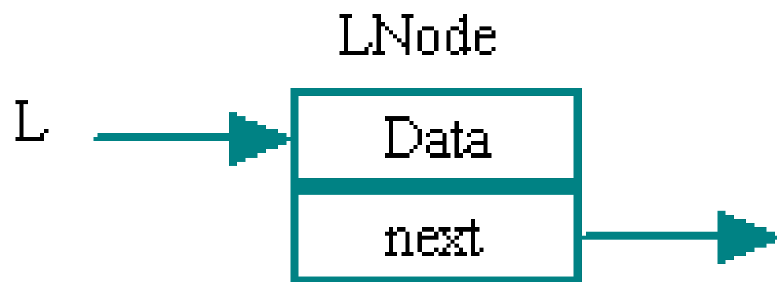
| 存储地址 | 数据域 | 指针域 |
| --- | --- | --- |
| 1 | d | 43 |
| 7 | b | 13 |
| 13 | c | 1 |
| 19 | h | NULL |
| 25 | f | 37 |
| 31 | a | 7 |
| 37 | g | 19 |
| 43 | e | 25 |

线性链表的逻辑状态：



# 线性表的链式存储（单链表定义）

- C语言的实现：




- 单链表的结点定义：

```
typedef struct node {        //node为结点类型预定义符
      ElemType  data;        //数据域
      struct  node *next;    //指针域
}  node, *LinkList;          //结点类型符、结点指针类型符
```

# 4.2、基于单链表的算法实现

- 假设链表中存储的是课程成绩（C语言的实现）：

```
typedef  int  ElemType;    //元素类型定义
typedef struct Node {      //node为结点类型预定义符
      ElemType  data;      //数据域
      struct  Node *next;  //指针域
} Node, *LinkList;         //结点类型符、结点指针类型符

LinkList   Head;   //定义一个单链表头指针为Head的变量
```

# 单链表的算法---构造一个空链表

## 1、构造一个空链表

不带头结点的单链表为空表的判断条件：Head = = NULL

方法一：参数采用二级指针，间接修改单链表头指针变量的值。

```
void  Init_LinkList(LinkList  *Head_pointer)
{    *Head_pointer = NULL; }
```

调用方式：Init_LinkList( &Head );

方法二：参数采用变参专递要被初始化的单链表头指针，从而修改其值。

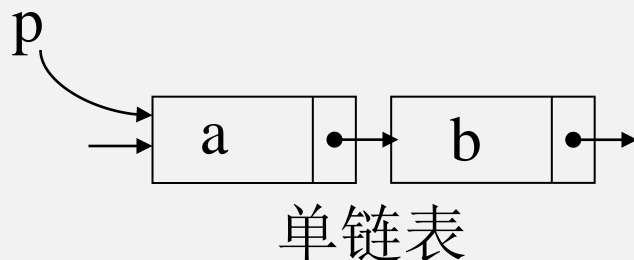
```
void  Init_LinkList(LinkList  &Head_pointer)
{    Head_pointer = NULL;  }
```

调用方式：Init_LinkList( Head );

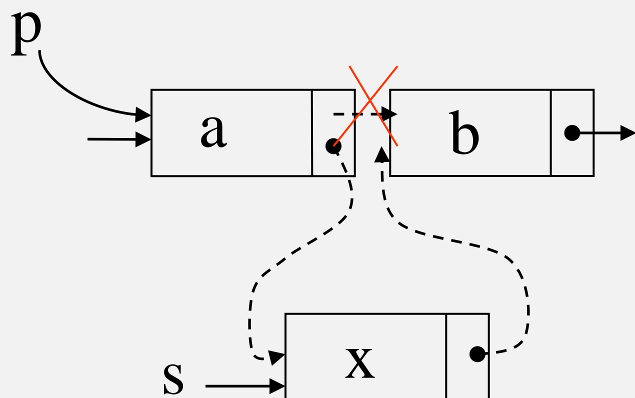
算法复杂度为：O(1)。

# 单链表的算法---插入与删除

单链表结点插入和删除时的情形：




单链表

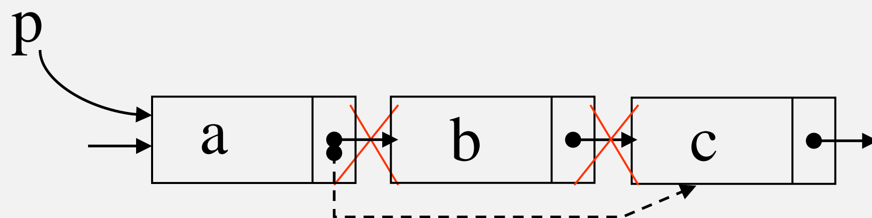



单链表插入结点时的情况

s->next = p->next;

p->next = s;

注：顺序不能反



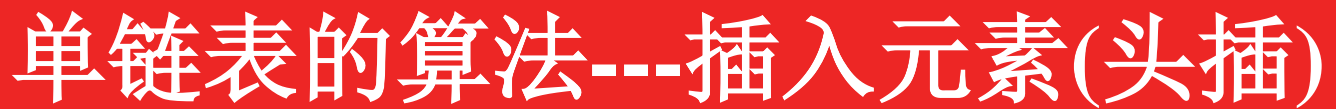

单链表删除结点时的情况

q=p->next;

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

free(q);

注：注意语句关系




## 2、插入一个元素（头上插入）

```
bool Insert_First(LinkList  &L,  ElemType  x ) //线性链表头指针，待插元素
{   Node  *p;                                  //定义一个临时的结点指针变量
    p = ( LinkList )malloc( sizeof(Node) );    // 动态申请分配一个结点存储空间
    if （ p==NULL）return false;               // 返回结点分配失败
    p->data = x;              // 给新结点的数据域填值
    p->next = L;              // 让新节点的后继指向原来的头结点
    L = p;                    // 修改头指针指向新插入的结点
    return  true;             // 插入结束，返回成功
}
```

**调用方式：** Insert_First( Head, 80 )

**算法复杂度为：** O(1)

# 单链表的算法---插入元素(中间插入)

## 3、插入一个元素（中间插入：指定元素之后）

```
bool Insert_Mid( LinkList  q, ElemType  x ) // 待插位置指针，待插元素值
{   Node  *p;                                  //定义一个临时的结点指针变量
    p = ( LinkList )malloc( sizeof(Node) );    // 动态申请分配一个结点存储空间
    if （ p==NULL）return false;               // 返回结点分配失败
    p->data = x;               // 给新结点的数据域填值
    p->next = q->next;         // 让新节点的后继指向P的后继结点
    q->next = p;               // 修改p指针的后继指向新插入的结点
    return  true;               // 插入结束，返回成功
}
```

**调用方式**：  Insert_Mid( pos, 80 )    其中：pos指向链表中某一个结点

**算法复杂度为**：  O(1)

# 单链表的算法---插入元素(尾部插入)

## 4、插入一个元素（尾部插入）

```
bool Insert_Tail( LinkList  &L, ElemType  x ) // 待插位置指针，待插元素值
{   Node  *p, *q;                                   //定义临时的结点指针变量
    p = ( LinkList )malloc( sizeof(Node) );         // 动态申请分配一个结点存储空间
    if （ p==NULL）return false;                    // 返回结点分配失败
    p->data = x; p->next = NULL;                    // 给新结点的数据域和链域填值
    q = L;                               // 让q指向表头结点
    if (q == NULL) L = p;                // 空表插入元素就是第一个唯一元素
    else {
        while(q->next != NULL ) q = q->next;  //循环查找直到链尾
        q->next = p;                     // 将p指向的新结点插入链尾
    }
    return  true;                  // 插入结束，返回成功
}
```

调用方式： Insert_Tail( Head, 80 )    其中：Head是链表的头指针

算法复杂度为： O(n)

# 单链表的算法---查找指定元素

## 5、查找指定元素（查找值为x的元素）

```
LinkList Search( LinkList  L, ElemType  x )
/* 头指针，待查找元素的值*/
{   LinkList  p;    //定义一个临时的结点指针变量
    p = L;          // 让p指向第一个结点
    while( p != NULL）{                  // 链表未结束，则继续
        if ( p->data == x ) break;   // 检查p指向元素的值是否为要查找值
        else  p = p->next;           //  当前元素不是要找的值，则p指针后移
    }
    return  p;       // 查找结束，成功则返回找到位置指针，否则返回NULL
}
```

调用方式： Search( Head, 85 )　其中：Head为链表头指针

算法复杂度为：O(n)（计算循环的执行频度）

1） Worst Case: n-1

2） Average Case: $\sum_{i=1}^{n} P_i(i-1) = \frac{1}{n}\sum_{i=1}^{n}(i-1) = \frac{n-1}{2}$

# 单链表的算法---删除指定元素

## 6、删除指定元素（查找值为x的元素，并删除之）

```
bool Delete(LinkList &L, ElemType x)  //头指针，待删除元素值
{   LinkList  p, q;             //定义临时的结点指针变量
    p = L; q = NULL;                     // 让p指向第一个结点；q指向其前驱
    while( p != NULL） {                 // 链表未结束，则继续
        if ( p->data == x ) break;   // 检查p指向元素的值是否为要查找值
        else  {q = p; p = p->next;} //  当前不是，则q指向p，p指针后移
    }
    if (p ==NULL)  return FALSE;  // 未找到待删除元素，返回失败
    else if ( q==NULL) { free(p); L = NULL; }         //找到的是第一个元素
    else { q->next = p->next; free(p); }              //找到的是中间元素
    return  TURE;                                     // 删除结束，返回成功
}
```

调用方式：  Delete( Head, 80 )     其中：Head为链表头指针

算法复杂度为：最好情况O(1)；最坏情况O(n)；  平均情况：O((n-1)/2)=O(n)

## 7、遍历线性表元素

```
void  Show_List(LinkList  L)                        //头指针
{   LinkList  p;                  //定义临时的结点指针变量
    p = L;                        // 让p指向第一个结点
    printf("\n 线性表元素如下: \n");
    if ( p == NULL) printf("\n 空表！ " );
    while( p != NULL）{                  // 链表未结束，则继续
          printf("  %d ", p->data );    // 输出当前元素的值
          p = p->next;                  //  p指针后移
    }
}
```

调用方式：  Show_List( Head )    其中：Head为链表头指针

算法复杂度为：O(n)

# 单链表的算法---计算表长度

## 8、计算线性表的长度

```
int  Length_List(LinkList  L)          //头指针
{    LinkList  p;                      //定义临时的结点指针变量
     int sum = 0;                      //定义计数器，并设初值为0
     p = L;                            // 让p指向第一个结点
     while( p != NULL）{               // 链表未结束，则继续
          sum++;                       // 统计线性表的长度
          p = p->next;                 //  p指针后移
     }
     return sum;                        // 返回线性表的长度
}
```

调用方式： Length_List( Head )　其中：Head为链表头指针

算法复杂度为：O(n)

# 单链表的应用算法---多项式相加

## 9、一元多项式相加（单链表的应用）

一元多项式的表示：

$$P_n(x) = p_0 + p_1 x + p_2 x^2 + \cdots + p_n x^n$$

$$= \sum_{i=0}^{n} p_i x^i$$

- $n$ 阶多项式 $P_n(x)$ 有 $n$+1 项。
  - 指数 0, 1, 2, …, $n$。按升幂排列
  - 系数 $p_0, p_1, p_2, \ldots, p_n$
- 可以用系数序列来表示一个多项式
  - $(p_0, p_1, p_2, \ldots, p_n)$

# 单链表的应用算法---多项式相加

例 4-2-1

| | |
|---|---|
| $7 + 8x$ | ( 7 8 ) |
| $3 + x^2$ | ( 3 0 2 ) |
| $5 + 2x + 3x^2$ | ( 5 2 3 ) |
| $8 + 6x + 9x^2 + 2x^4$ | ( 8 6 9 0 2 ) |

设想：

1、一元多项式可以采用顺序表存储；

2、两个一元多项式相加，可用两个顺序表实现。

# 一元多项式：$4 + 3x^{2001}$

( 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 )

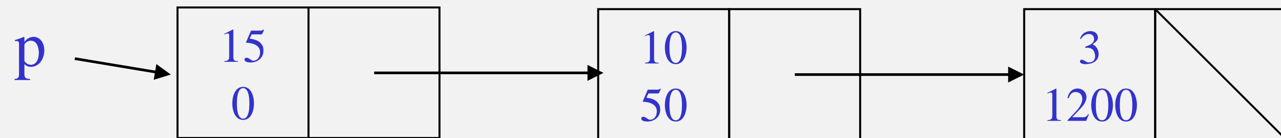
# 单链表的应用算法---多项式相加

改进：只记录系数不为0的指数项：$4 + 3x^{2001}$

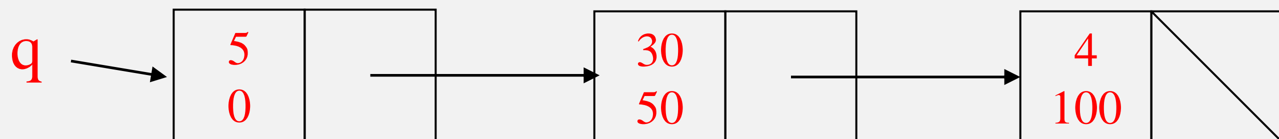
(<4,0>  <3,2001>)

多项式的相加：

$15+10x^{50}+3x^{1200}$




$5+30x^{50}+4x^{100}$



```
typedef  ElemType{      //元素类型定义
         int  ratio；
         int  index；
}
typedef struct Node {                   //node为结点类型预定义符
         ElemType  data;                //数据域
         struct  Node *next;            //指针域
} Node, *LinkList;                      //结点类型符、结点指针类型符

LinkList Head1, Head2, Head3; //定义一个单链表头指针为Head的变量
```

```
void Sum_Polynomial(LinkList L1, LinkList L2, LinkList &L3){ //相加对象及结果
    LinkList  p, q, w;
    p = L1;  q=L2; w=NULL;
    while (p != NULL && q != NULL){
            w = ( LinkList )malloc( sizeof(Node) );  w->next = NULL;
            if (p->data.index == q->data.index) {
                        w->data.index = p->data.index;
                        w->data.ratio  = p->data.ratio + q->data.ratio;
                        p = p->next;  q = q->next;
            } else  if (p->data.index < q->data.index){
                        w->data.index = p->data.index;
                        w->data.ratio  = p->data.ratio ;
                        p = p->next;
            }else {     w->data.index = q->data.index;
                        w->data.ratio  = q->data.ratio ;
                        q = q->next;
            }
            Append_List(&L3,  w);
    }       // end of while (p != NULL && q != NULL)
```

```
    while (p != NULL ){
          w = ( LinkList )malloc( sizeof(Node) );
          w->data.index = p->data.index;
          w->data.ratio  = p->data.ratio ;
          Append_List(&L3,  w);
          p = p->next;
    }
    while (q != NULL ){
          w = ( LinkList )malloc( sizeof(Node) );
          w->data.index = q->data.index;
          w->data.ratio  = q->data.ratio ;
          Append_List(&L3,  w);
          q = q->next;
    }
}  // end of Sum_Polynomial
```

```
void Append_List(LinkList &L, LinkList p){ //把p指向的结点插入到单链表L的尾
    LinkList  q;
    if (L == NULL) { L=p; }
    else {
        q=L;
        while (q->next != NULL) q = q->next;
        q->next = p;
    }
}
```

**假设：** Head1和Head2已经建好两个一元多项式，Head3=NULL;

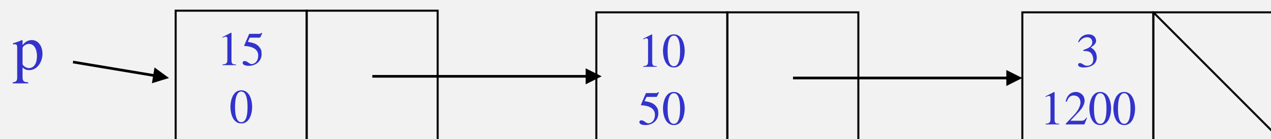
**调用方式：** Sum_Polynomial(Head1, Head2, &Head3)

**算法复杂度：** O(n+m)　　其中：n为Head1长度，m为Head2的长度。

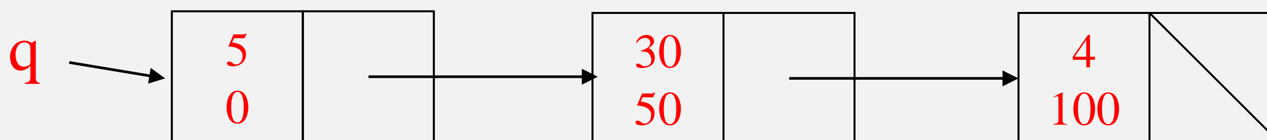
# 单链表的应用算法---多项式相加

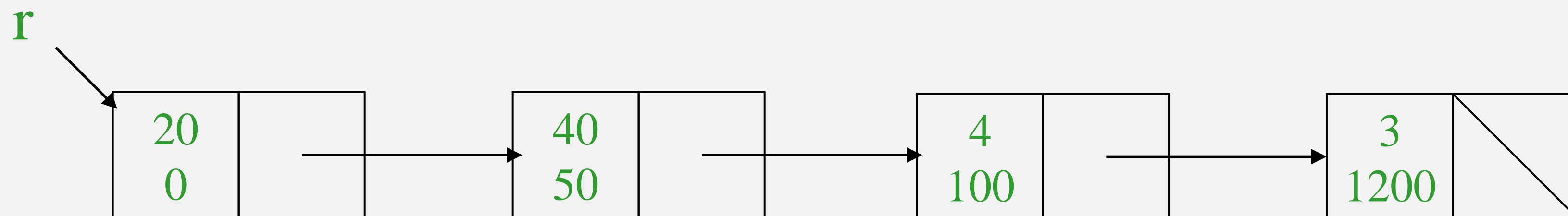
遍历每个链表：θ(n+m)

$15+10x^{50}+3x^{1200}$




$5+30x^{50}+4x^{100}$

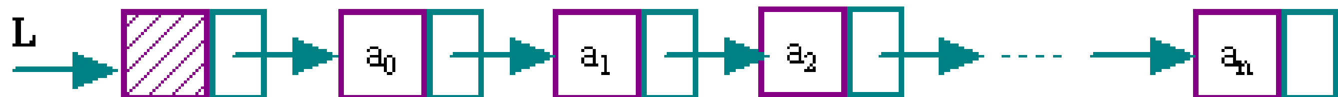


# 4.3 链式存储的其他方法

1、单循环链表
2、双向链表
3、双向循环链表
4、带头结点的单链表
5、带头结点的单循环链表
6、带头结点的双向链表
7、带头结点的双向循环链表

例如：带头结点的单链表如下：



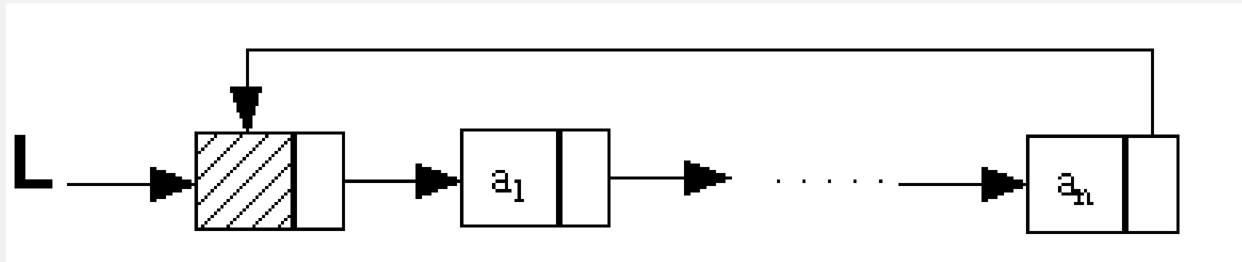
# 4.3 链式存储的其他方法

**1、不带头结点的链表为空条件（前三种）**

Head == NULL

**2、带头结点的单循环链表**

例如：带头结点的循环单链表如下：

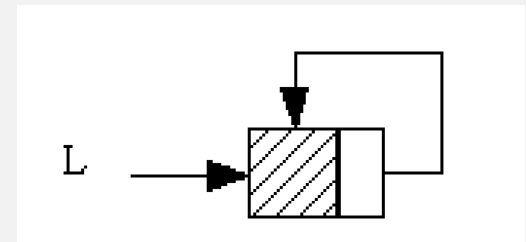



带头结点的单循环链表为“空”的条件：

**Head->next == Head**

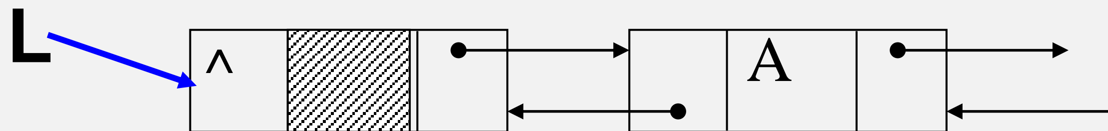
注：带头结点的单链表为空条件：

**Head->next == NULL**



# 4.3 链式存储的其他方法

## 3、带头结点的双向链表




双向链表为“空”的条件：

Head->next == NULL

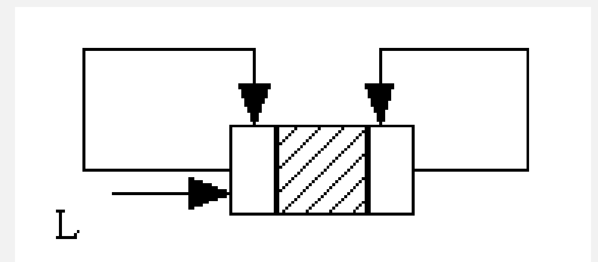
或者 Head->prior == NULL

## 4、带头结点的双向循环链表

判断为空表的条件：

Head->next == Head

或者 Head->prior == Head



# 4.4、基于带头结点的单循环链表算法

- 带头结点的单循环链表的数据定义与单链表一样：

```
typedef  int  ElemType;    //元素类型定义
typedef struct Node {       //node为结点类型预定义符
        ElemType  data;     //数据域
        struct  Node *next; //指针域
} Node, *LinkList;          //结点类型符、结点指针类型符

LinkList   Head;   //定义一个单链表头指针为Head的变量
```

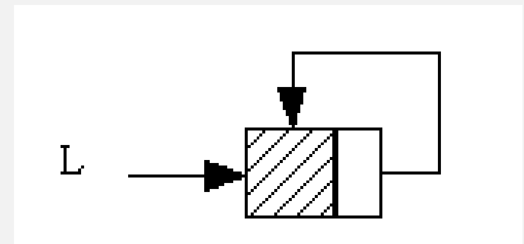
# 带头结点的单循环链表 (建空表)

## 1、构造一个空链表（带头结点的单循环链表）

带头结点的单循环链表为空表的判断条件：Head->next == NULL

方法一：参数采用二级指针，间接修改单链表头指针变量的值。

```
bool  Init_LLinkList(LinkList  *LHead )
{    LinkList   p;
     p = (LinkList ) malloc( sizeof(Node) );
     if  (p == NULL ) return FALSE;
     p->next = p;
     *LHead = p;
     return  TRUE;
 }
```




调用方式：Init_LLinkList( &Head );

算法复杂度为：O(1)。

# 带头结点的单循环链表 (建空表)

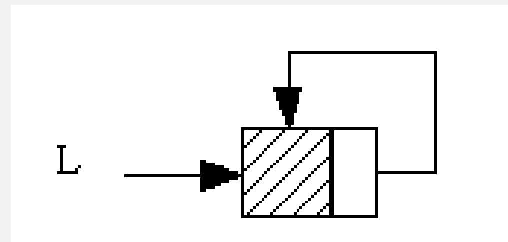
## 构造一个空链表（带头结点的单循环链表）

方法二：参数采用变参专递要被初始化的单链表头指针，从而修改其值。

```
bool  Init_LLinkList(LinkList  &LHead )
{    LinkList    p;
     p = (LinkList ) malloc( sizeof(Node) );
     if  (p == NULL ) return FALSE;
     p->next = p;
     LHead = p;
     return  TRUE;
 }
```

调用方式：Init_LLinkList( Head );

算法复杂度为：O(1)。



# 带头结点的单循环链表 (第i之后插入)

## 2、插入一个元素（在第i个结点之后插入）

```
bool Insert_Li(LinkList L, ElemType x, int i) // 头指针，待插元素值及位置
{   Node  *p, *q;                                    //定义临时的结点指针变量
    p = ( LinkList )malloc( sizeof(Node) );     // 动态申请分配一个结点存储空间
    if （ p==NULL）return FALSE;                // 返回结点分配失败
    p->data = x;                 // 给新结点的数据域填值
    q = L;                       // 让q指向头结点(L指向的表头结点）
    while (q->next != H && i>0) { q=q->next; i--; }   // q指针按照i计数后移
    if ( i==0 ) {                // q指向表中结点，i计数结束
        p->next = q->next;          // 让新插结点p的后继指向q后继结点
        q->next = p;                // 修改q后继指向新插入的结点
        return  TRUE;               // 插入结束，返回成功
    } else  return  FALSE;          // 插入结束，返回失败
}
```

注：书上p118算法4.15有问题，它不能实现在插入表头插入元素。

**调用方式**： Insert_Li(Head, 80, m )    其中：Head指向头结点

**算法复杂度为**： O(n)  ---根据i定位的循环语句频度

# 带头结点的单循环链表 (查找x)

## 3、查找指定元素（带头结点单循环表中查找值x元素）

```
LinkList Search_LList( LinkList  L, ElemType  x )
/* 头指针，待查找元素的值*/
{   LinkList  p;    //定义一个临时的结点指针变量
    p = L->next;   // 让p指向第一个结点
    while( p != L）{                          // 链表未结束，则继续
           if ( p->data == x ) break;   // 检查p指向元素的值是否为要查找值
           else  p = p->next;              //  当前元素不是要找的值，则p指针后移
    }
    if (p != L ) return  p;       // 查找结束，成功则返回找到元素结点指针
    else  return NULL;          // 否则查找失败，返回NULL
}
```

调用方式： Search_LList( Head, 85 )    其中：Head为链表头指针

算法复杂度为：O(n) （计算循环的执行频度）

# 带头结点的单循环链表 (删除值x)

## 4、删除指定元素（查找值为x的元素，并删除之）

```
bool Delete_LList(LinkList L, ElemType x)  //头指针，待删除元素值
{   LinkList  p, q;                          //定义临时的结点指针变量
    q = L; p = q->next;                      // 让p指向第一个结点；q指向头结点
    while( p != L）{                         // 链表未结束，则继续
          if ( p->data == x ) break;         // 检查p指向元素的值是否为要查找值
          else  {q = p; p = p->next;}        //  当前不是，则q指向p，p指针后移
    }
    if (p == L )  return FALSE;              // 未找到待删除元素，返回失败
    else  { q->next = p->next; free(p); }          //找到的是链表中元素
    return  TRUE;              // 删除结束，返回成功
}
```

注：书上p119算法4.17有问题，不能删除表尾结点。

调用方式： Delete_LList( Head, 80 )　 其中：Head为链表头指针

算法复杂度为：O(n) ---计算循环的执行频度

# 带头结点的单循环链表 (遍历链表)

## 5、遍历线性表元素（带头结点的单循环链表）

```
void  Show_LList(LinkList  L)                    //头指针
{   LinkList  p;                 //定义临时的结点指针变量
    p = L->next;                 // 让p指向第一个结点
    printf("\n 线性表元素如下: \n");
    if ( p == L) printf("\n 空表！ " );
    while( p != L）{                    // 链表未结束，则继续
        printf("  %d ", p->data );     // 输出当前元素的值
        p = p->next;                    //  p指针后移
    }
}
```

调用方式：  Show_LList( Head )    其中：Head为链表头指针

算法复杂度为：O(n)

# 带头结点的单循环链表 (计算表长)

## 6、计算线性表的长度

```
int  Length_LList(LinkList  L)        //头指针
{   LinkList  p;                      //定义临时的结点指针变量
    int sum = 0;                      //定义计数器，并设初值为0
    p = L->next;                      // 让p指向第一个结点
    while( p != L）{                  // 链表未结束，则继续
          sum++;                      // 统计线性表的长度
          p = p->next;                //  p指针后移
    }
    return sum;                       // 返回线性表的长度
}
```

调用方式： Length_LList( Head )    其中：Head为链表头指针

算法复杂度为：O(n)

# 带头结点的单循环链表 (清空表)

## 7、清空线性表（带头结点的单循环链表）

```
void  Clear_LList(LinkList  L)          //头指针
{    LinkList  p, q;                    //定义临时的结点指针变量
     p = L->next;                       // 让p指向第一个结点
     while( p != L）{                   // 链表未结束，则继续
            q = p;                      // 保存待删除结点指针q
            p = p->next;                //  p指针后移
            free( q);                   // 删除q指向的待删除结点
     }
     L-next = L;                        // 设置头结点为空表
}
```

调用方式： Length_LList( Head )　其中：Head为链表头指针

算法复杂度为：O(n)

## 带与不带头结点的单循环链表比较：

1、**带头结点的单循环链表中一个冗余的表头结点**，因此，除了初始化链表需要通过间接变量引用或传递一个二级指针，其他函数不会修改头指针，也就只需要值传递即可。

2、带头结点的任何形式链表操作中插入和删除函数实现时，**对空表和实表处理是一样处理的**；最重要的目的是可以简化操作实现。

**唯一缺点是带来一个结点空间的浪费**。

3、**不带头结点**的单循环链表在插入和删除函数实现时，**要考虑处理由实表变成空表或由空表变成实表的特殊处理**，此时都需要修改头指针。

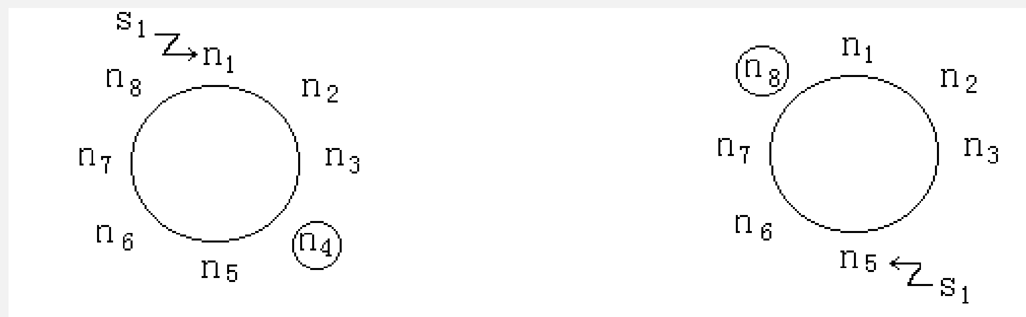
4、**带头结点或不带头结点的链表没有好坏之分**，采用哪种存储方式取决于实际问题的需要。

5、**一般的循环链表大都采用带头结点存储方式**，因为此时空表已经构成循环链表方式，操作实现变得简单方便。

# 不带头结点的单循环链表 (应用)

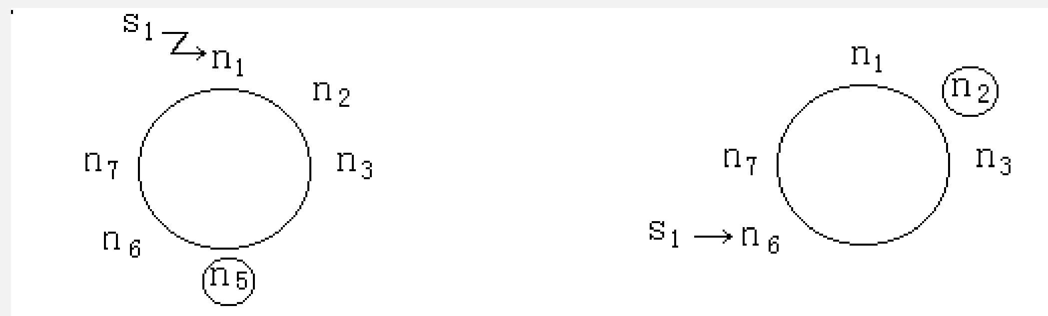
## 8、 Josephus约瑟夫问题（不带头结点的单循环链表）

设有n个人围坐在一个圆桌周围，现从第s个人开始报数，数到第m的人出列，然后从出列的下一个人重新开始报数，数到第m的人又出列，…，如此反复直到所有的人全部出列为止。Josephus问题是：对于任意给定的n,s和m，求出按出列次序得到的n个人员的序列




初值：N=8,s=1,m=4

结果：$n_4,n_8,n_5,n_2,n_1,n_3,n_7,n_6$

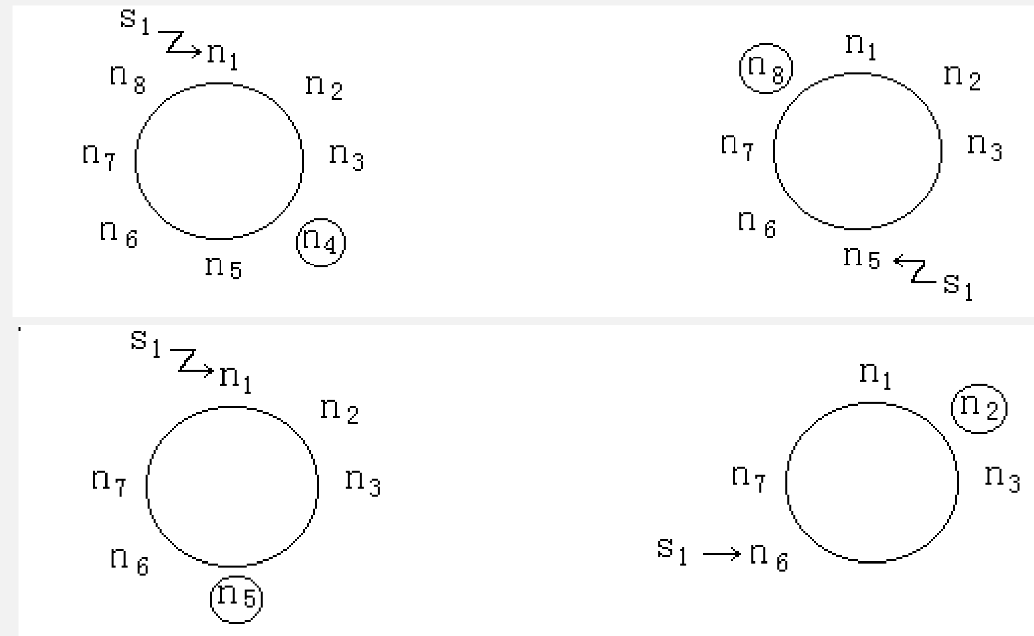


# 不带头结点的单循环链表 (应用)

Josephus约瑟夫问题（不带头结点的单循环链表）

初值：N=8,s=1,m=4

结果：$n_4,n_8,n_5,n_2,n_1,n_3,n_7,n_6$




分析：

– 选用数据结构?

1、选用的数据结构应该能够从任何位置开始访问其他所有的结点；

2 、从最后一个数据循环到第一个数据应该比较方便。

结论：

– 选择不带头结点的单向循环链表

# Josephus约瑟夫问题（算法实现）

```
LinkList Head=NULL;     //定义不带头结点的循环单链表：头指针初始化
Status Josephus(int n, int s, int m)
{   LinkList p, q;              //定义临时的单链表指针
    CreateList(&Head,  n); //创建n个结点的单循环链表
    p = GetElem(Head, s); //找出第s个结点
    while(Head)
    {
        q = GetElem(p, m);          //从p所指的结点开始，找到第m个结点
        p = q->next;                //下一次过程的起始位置
        DeleteElem(&Head, q, &e);   //删除q指向的结点，返回其值在e中
        printf(e);
    }
}
```

1、删除结点的时候，需要改变前驱结点的关系

2、注意最后一个结点的处理。

3、建立单链表的时候注意第一个结点的处理。

# Josephus约瑟夫问题（算法实现）

采用不带头结点LinkList:

```
Void CreateList(LinkList &L, int n)   //读取并创建n个结点的单循环表
{  LinkList p, Tail;                  //定义临时的单链表指针
        for(i=n; i>0; i--) {       //循环插入n的结点
                p = ( LinkList )malloc( sizeof( Lnode ) );
                scanf( &p->data );      //读取当前结点的值
                if(!L){ L=p; Tail=p; //循环单链表为空，插入第1个结点
                        p->next = L; //第一个单结点构成循环
                }else { p->next = Tail->next;  //已有元素，插入到末尾
                        Tail->next = p;
                        Tail = p;          //修改尾指针
                }
        }
}
```

# Josephus约瑟夫问题（算法实现）

```
LinkList  GetElem(LinkList p, int m)
{   //求从循环链表上p所指的结点开始，第m个结点
    j=1;
    if(!p) return NULL;                    //p为NULL，返回失败
    while(j<m){ p=p->next; j++;}    //按m计数移动p指针
    return p;                              //返回完成计数后移到的结点指针p
}
Status DeleteElem(LinkList &L, LinkList p, ElemType &e)
{   //删除循环链表中p所指的节点
    LinkList pre=L;            //删除p指向结点，先要找到其前驱pre
    if(!L) return FALSE;    //空表，返回失败
    while(pre->next != p){ pre = pre->next;} //找p结点的前驱结点pre
    e = p->data;                           //填写返回被删结点的参数值
    if( pre = p){ L = NULL; }          //p是最后一个结点，删除后为空表
    else{ pre->next = p->next;}       //从单循环链表中摘除p结点
    free(p);                               //释放p指向结点空间
    return  TURE;                          //删除结束，返回成功
}                       《本片完》
```