

第二章 线性表

- 本章主要介绍下列内容
 - 线性表的定义和基本操作
 - 线性表的顺序存储结构
 - 线性表的链式存储结构
 - 线性表的应用举例

线性表

定义： n (≥ 0) 个数据元素的有限序列，
记作 $(a_0, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_{n-1})$

其中 a_i 是表中数据元素， n 是表长度。

特点：

- 同一线性表中元素具有相同特性。
- 相邻数据元素之间存在序偶关系。
- 除第一个元素外，其它每一个元素有且仅有一个直接前驱。
- 除最后一个元素外，其它每一个元素有且仅有一个直接后继。

举例:

L1= (34, 89, 765, 12, 90, -34, 22)

数据元素类型为int。

L2=("Hello","World", "China", "Welcome")

数据元素类型为string。

L3=(book₁,book₂,...,book₁₀₀)

数据元素类型为下列所示的结构类型:

```
struct bookinfo{  
    int No;           //图书编号  
    char *name;       //图书名称  
    char *author;     //作者名称  
}
```

线性表的基本操作

1. 初始化线性表L `InitList(L)`
2. 销毁线性表L `DestoryList(L)`
3. 清空线性表L `ClearList(L)`
4. 求线性表L的长度 `ListLength(L)`
5. 判断线性表L是否为空 `IsEmpty(L)`
6. 获取线性表L中的某个数据元素内容 `GetElem(L,i,e)`
7. 检索值为e的数据元素 `LocateElem(L,e)`
8. 返回线性表L中e的直接前驱元素 `PriorElem(L,e)`
9. 返回线性表L中e的直接后继元素 `NextElem(L,e)`
10. 在线性表L中插入一个数据元素 `ListInsert(L,i,e)`
11. 删除线性表L中第i个数据元素 `ListDelete(L,i,e)`

顺序表

定义： 将线性表中的元素相继存放在一个连续的存储空间中。

存储结构： 数组

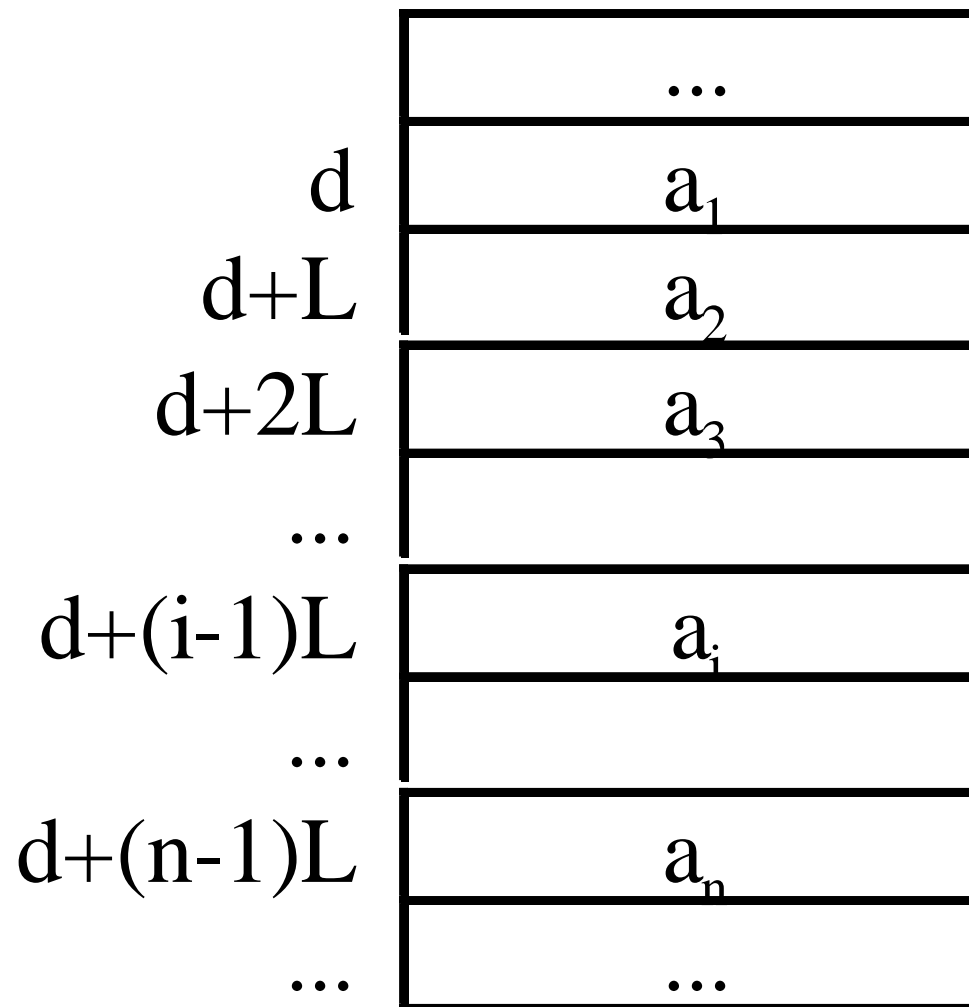
特点： 线性表的顺序存储方式。

存取方式： 顺序存取

顺序存储结构示意图



存储地址 内存单元

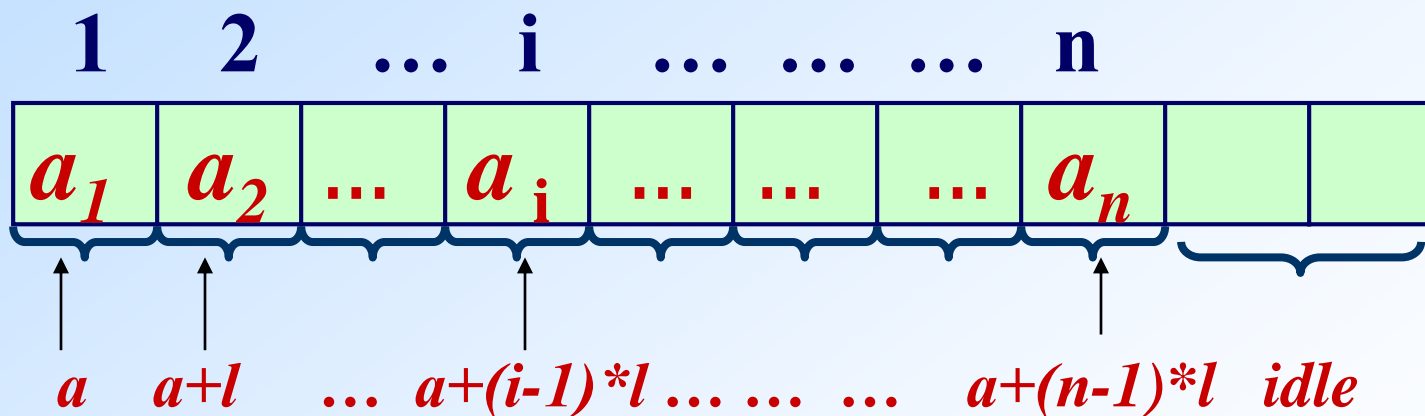


线性表顺序存储结构示意图

顺序表的存储方式:

$$\text{LOC}(a_{i+1}) = \text{LOC}(a_i) + l$$

$$\text{LOC}(a_i) = \text{LOC}(a_1) + (i-1) * l$$



顺序表 (SeqList) 的类型定义

```
#define ListSize 100    //最大允许长度
```

```
#define Listincrement 100
```

```
typedef struct {
```

```
    Elemtype * data;    //存储空间基址
```

```
    int length;         //当前元素个数
```

```
    int listsize;       //当前分配的存储容量
```

```
} SeqList
```


顺序表基本运算

■ 初始化 //动态分配数组空间

```
void InitList ( SeqList * L ) {  
    L->data = (Elemtype* ) malloc  
        ( ListSize * sizeof (Elemtype) );  
        //相对于c++中new运算符  
    if ( L->data == NULL ) {  
        printf ( “存储分配失败!\n” );  
        exit (1);  
    }  
    L->length = 0;  
}
```

建立顺序表

```
void creatlist(SeqList * L)
{ int i;
  printf("input the length is ");
  scanf("%d",&L->length);
  for (i=0;i<L->length;i++)
    {printf("input the data is ");
     scanf("%d",&(L->data[i]));
    }
}
```

- **按值查找：** 查找x在表中的位置，若查找成功，返回x的位置，否则返回-1

```
int Find ( SeqList *L, Elemtype x ) {  
    int i = 0;  
    while ( i < L->length && L->data[i] != x )  
        i++;  
    if ( i < L->length ) return i; // i计数从0起  
    else return -1;  
}
```

按值查找：判断x是否在表中

```
int IsIn ( SeqList *L, Elemtypex )  
{  
    int i = 0, found=0;  
    while ( i < L->length &&!found )  
        if (L->data[i] != x ) i++;  
        else found=1;  
    return found;  
}
```

- **求表的长度**

```
int Length ( SeqList * L ) {  
    return L->length;  
}
```

- **获取函数：在表中获取第 i 个元素的值**

```
Elemtype GetData ( SeqList *L, int i ) {  
    if ( i >= 1 && i <= L->length )  
        return L->data[i-1];  
    else printf ( “参数 i 不合理！ \n” );  
}
```

- 寻找x的后继

```
int Next ( SeqList *L, Elemtype x ) {  
    int i = Find(x);  
    if ( i >=0 && i < L->length-1 ) return i+1;  
    else return -1;  
}
```

- 寻找x的前驱

```
int prior( SeqList *L, Elemtype x ) {  
    int i = Find(x);  
    if ( i >0 && i < =L->length-1 ) return i-1;  
    else return -1;  
}
```

■ 插入

位置



顺序表插入时，在各表项插入概率相等时，
平均数据移动次数AMN

$$\begin{aligned}
 \text{AMN} &= \frac{1}{n+1} \sum_{i=1}^{n+1} (n-i+1) = \frac{1}{n+1} (n + \dots + 1 + 0) \\
 &= \frac{1}{(n+1)} \frac{n(n+1)}{2} = \frac{n}{2}
 \end{aligned}$$

▪ 顺序表的插入

```
int Insert ( SeqList *L, Elemtype x, int i ) {  
    //在表中第 i 个位置(1<=i<=n)插入新元素 x    // i 计数从1起  
    int *q,*p; int *newbase;  
    if (i < 1 || i > L->length+1) return 0;  
  
    if (L->length>=L->listsize){ //动态扩充数组空间  
        newbase=( Elemtype * ) realloc (L->data,  
            ( ListSize+listincrement) * sizeof ( Elemtype ) );  
        if (newbase== NULL ) {  
            printf ( “存储分配失败!\n” ); exit (1);}  
  
        L->data=newbase; //数组的基地址  
        L->listsize+=listincrement;    }  
  
        q=&(L->data[i-1]); //元素后移  
        for (p=&(L->data[L->length-1]); p>=q; --p) *(p+1)=*p;  
        *q=x;  
  
        ++L->length;  
        return 1;  
    }
```


- **装填因子**：线性表实际长度length与存储容量listsize的比值，是衡量空间利用率的重要指标。
- 如何实现扩容？“懒惰”策略：只有在即将发生溢出时，才将**容量加倍**，因此装填因子不低于50%。每次扩容元素的搬迁都需要花费额外时间，而单次操作所需的分摊运行时间为 $O(1)$ 。

■ 删除



顺序表删除在各表项删除概率相等时
平均数据移动次数 AMN

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

- **顺序表的删除** //在表中删除已有元素 x

```
int Delete ( SeqList *L, Elemtype x )
{ int j;
  int i = Find (L, x); //在表中查找 x , i计数从0起
  if ( i >= 0 ) {
    L->length -- ;
    for (j = i; j < L->length; j++ )
      L->data[j] = L->data[j+1]; //后续元素前移
    return 1;                    //成功删除
  }
  return 0;                      //表中没有 x
}
```

顺序表的应用:

集合的“并”运算

```
void Union ( SeqList *A, SeqList *B ) {  
    int n,m,x,k;  
    n = Length ( A );  
    m = Length ( B );  
    for ( int i = 0; i < m; i++ ) {  
        x = GetData ( B, i ); //在B中取一元素  
        k = Find ( A, x );    //在A中查找它  
        if ( k == -1 )        //若未找到插入它  
            { Insert ( A, x, n ); n++; }  
    }  
}
```

■ 集合的 “交” 运算

```
void Intersection ( SeqList *A, SeqList *B ) {  
    int n,m,x,k;  
    n = Length ( A );  
    m = Length ( B );  
    int i = 0;  
    while ( i < n ) {  
        x = GetData ( A, i ); //在A中取一元素  
        k = Find ( B, x );    //在B中查找它  
        if ( k == -1 ) { Delete ( A, i ); n--; }  
        else i++;             //未找到在A中删除它  
    }  
}
```

STL中的线性表顺序实现

向量类vector

- 在头文件<vector>中定义了模板类vector(向量类), 该类是一个容器。它实现的是顺序存储的线性表。
- 其中可以用v[i]来随机访问第i个元素。
- 在该模板类支持顺序线性表的基本操作。

vector底层与Array一样都是连续的内存空间, 但vector的空间是动态的, 随着更多元素的加入可以自动实现空间扩展。不是逐个元素向系统申请的低效扩展, 而是按照某种倍率来扩展, 有效减少因为扩容带来的效率降低问题。

练习1:

设计一个高效算法，将顺序表的所有元素逆置，要求算法的空间复杂度为 $O(1)$ 。

```
void reverse (SeqList *L)
{
    int i; Elemtypex;
    for (i=0;i<L->length/2;i++)
    {
        x=L->data[i];
        L->data[i]=L->data[L->length-i-1];
        L->data[L->length-i-1]=x;
    }
}
```

练习2： 设将 n 个整数存放到一维数组 R 中。试设计一个时间和空间两方面尽可能高效的算法，将 R 中整数序列循环左移 p 个位置，即将 R 中的序列 $(X_0, X_1, \dots, X_{n-1})$ 变换为 $(X_p, X_{p+1}, \dots, X_{n-1}, X_0, X_1, \dots, X_{p-1})$ 。

- 先将原始序列 $(X_0, X_1, \dots, X_p, X_{p+1}, \dots, X_{n-1})$ 原地逆置，得到 $(X_{n-1}, \dots, X_p, X_{p-1}, \dots, X_0)$ ，
- 然后再将前 $n-p$ 个元素 (X_{n-1}, \dots, X_p) 和后 p 个元素 (X_{p-1}, \dots, X_0) 分别原地逆置，得到最终结果
- $(X_p, X_{p+1}, \dots, X_{n-1}, X_0, X_1, \dots, X_{p-1})$ 。
- 算法的时间复杂度 $O(n)$, 空间复杂度 $O(1)$.

练习3：寻找主元素（众数）

$A = (0, 5, 5, 3, 5, 7, 5, 5)$

- 主元素问题，有时间复杂度 $O(n \log n)$ 的算法，但更快的有 $O(n)$ 的算法。基本思想是先将第一个元素保存到变量中，然后依次向后遍历，当遍历到的元素与变量值相同时，就将计数+1，不同则-1。当计数减到0，修改变量等于当前遍历到的元素值。
- 如果存在主元素，那么主元素的个数一定是大于 $n/2$ 的，所以最后抵消下来，剩下的一定是主元素。因此需再遍历一遍，判断最后的变量值是不是主元素。

- `#include <stdio.h>`
- `const int M = 100;`
- `int a[M];`
- `int main() {`
- `int n;`
- `scanf ("%d", &n);`
- `for (int i = 0; i < n; i++)`
- `scanf("%d", &a[i]);`
- `int cur_value = a[0]; cur_count = 1; //cur_value变量值, cur_count计数 */`
- `for (int i = 1; i < n; i++)`
- `{ if (cur_count == 0) //如果抵消完, 将它的值改成当前遍历到元素的值 */`
- `{ cur_value = a[i]; cur_count = 1; }`
- `else {`
- `if (cur_value == a[i])`
- `cur_count ++;`
- `else cur_count --;`
- `}`
- `}`

- `int cnt = 0;`
- `for (int i = 0; i < n; i++)` //判断是否存在主元素
- `if (cur_value == a[i]) cnt++;`
- `if (cnt > n / 2)`
- `printf("%d\n", cur_value);`
- `else`
- `puts("NO");`
- `}`
- `return 0;`
- `}`

线性表的链式存储结构

线性表顺序存储结构的特点，是一种简单、方便的存储方式，要求线性表的数据元素依次存放在连续的存储单元中，属于静态存储形式。暴露的问题：

- 在插入或删除元素时，会产生大量的数据元素移动；
- 对于长度变化较大的线性表，要一次性地分配足够的存储空间，但这些空间常常又得不到充分的利用；
- 线性表的容量扩充花费额外时间。

链表 (Linked List)

链表是线性表的链接存储表示

- 单链表
- 静态链表
- 循环链表
- 双向链表

单链表(Singly Linked List)

定义：用一组地址任意的存储单元存放线性表中的数据元素。

头指针

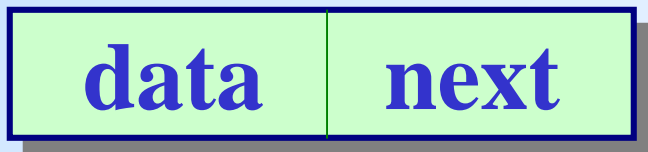
31

存储地址	数据域	指针域
1	ZHANG	13
7	WANG	1
13	LI	null
19	ZHAO	37
25	WU	7
31	ZHOU	19
37	SUN	25

单链表结构

每个元素由结点(Node)构成, 它包括两个域:数据域Data和指针域next

Node



存储结构：链式存储结构

特点：存储单元可以不连续

存取方式：非随机存取

■ 单链表的类型定义

```
typedef struct node {  
    Elemtype data;  
    struct node *next;  
} ListNode;
```

//链表结点

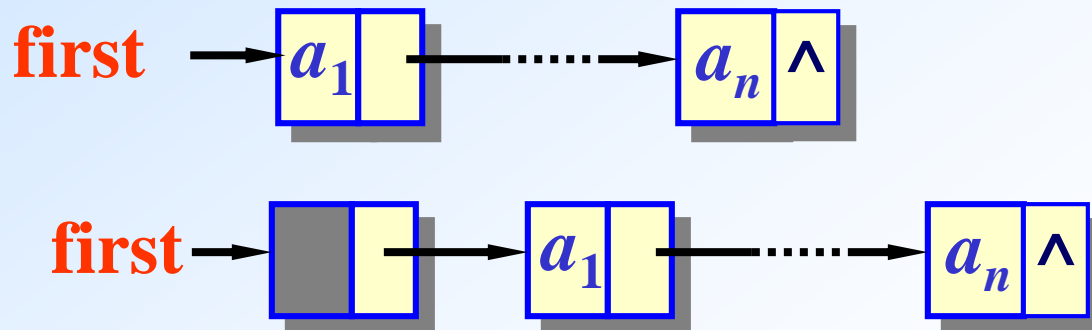
//结点数据域

//结点的指针域

```
typedef ListNode * LinkList;
```

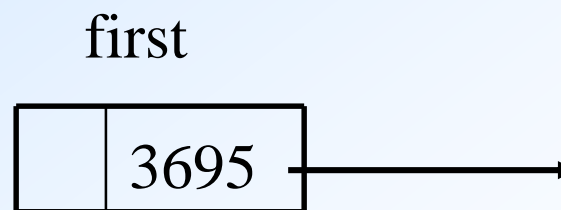
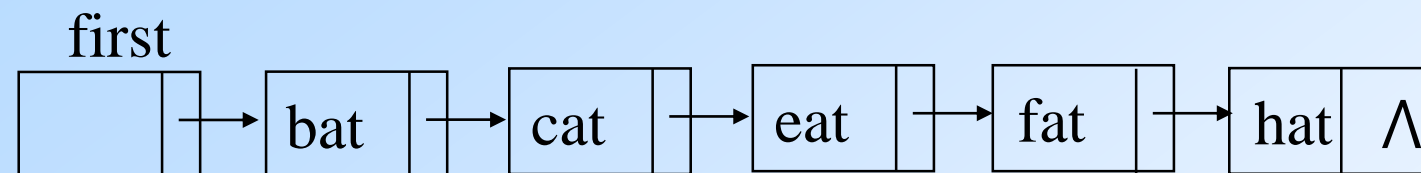
```
LinkList first;
```

//链表头指针first

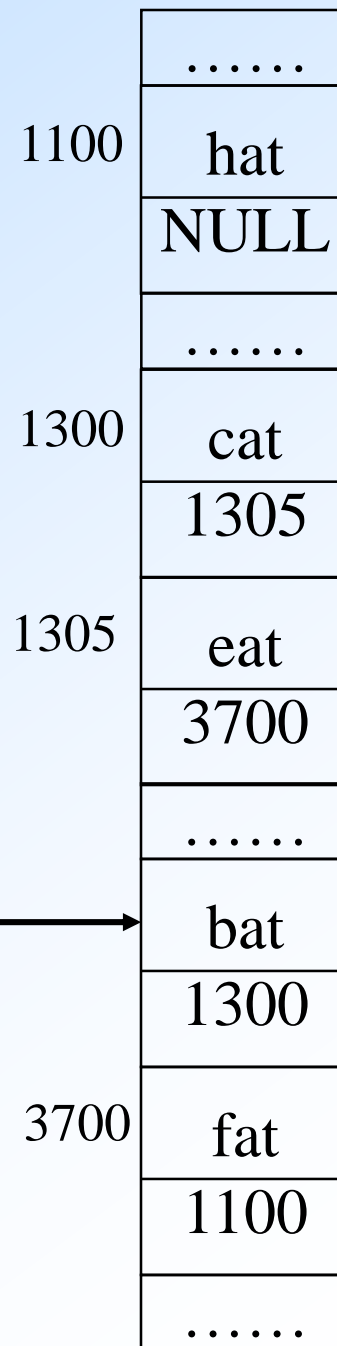


线性表L=(bat, cat, eat, fat, hat)

例： 逻辑结构

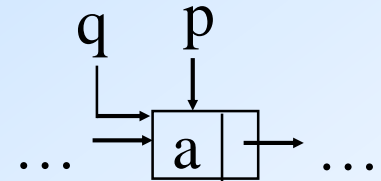
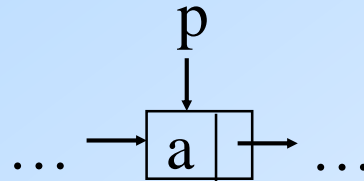


物理存储结构

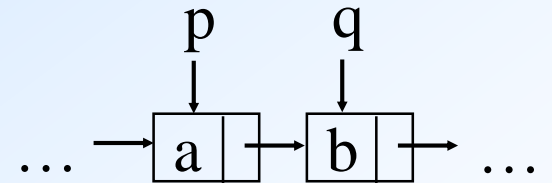
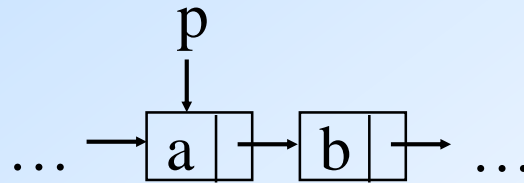


常见的指针操作

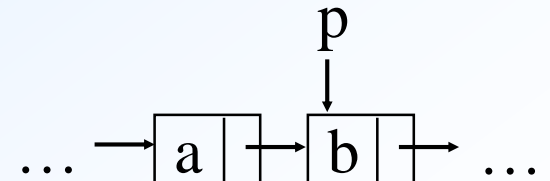
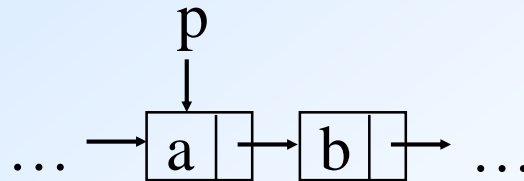
① $q=p;$



② $q=p \rightarrow \text{next};$

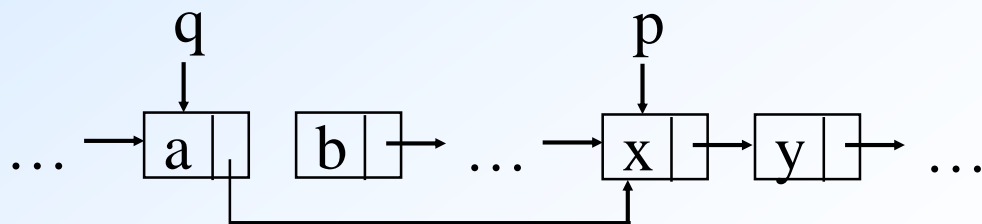
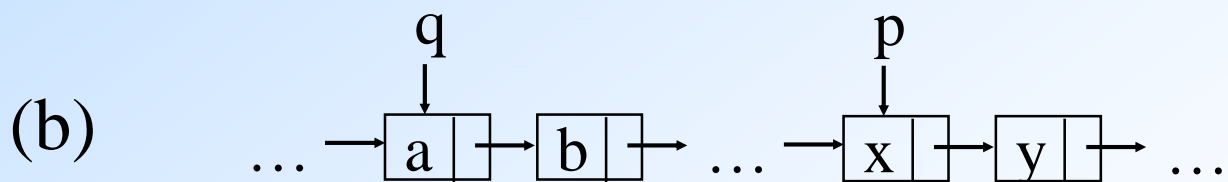
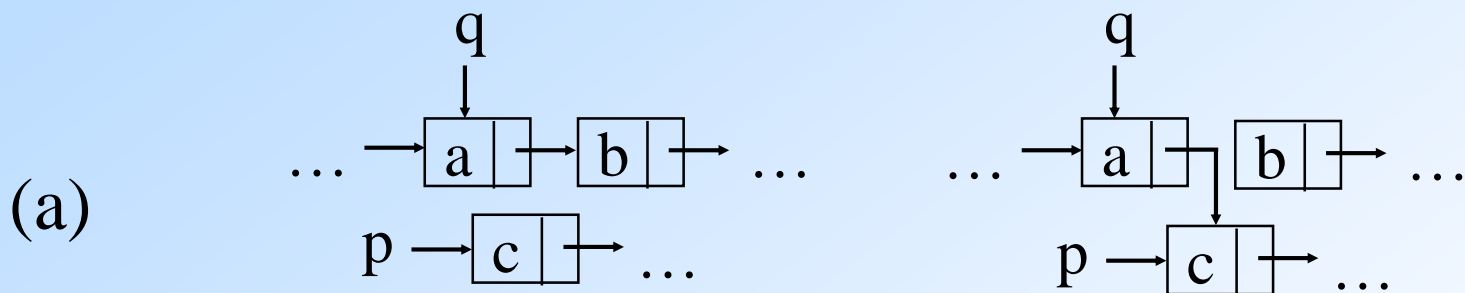


③ $p=p \rightarrow \text{next};$

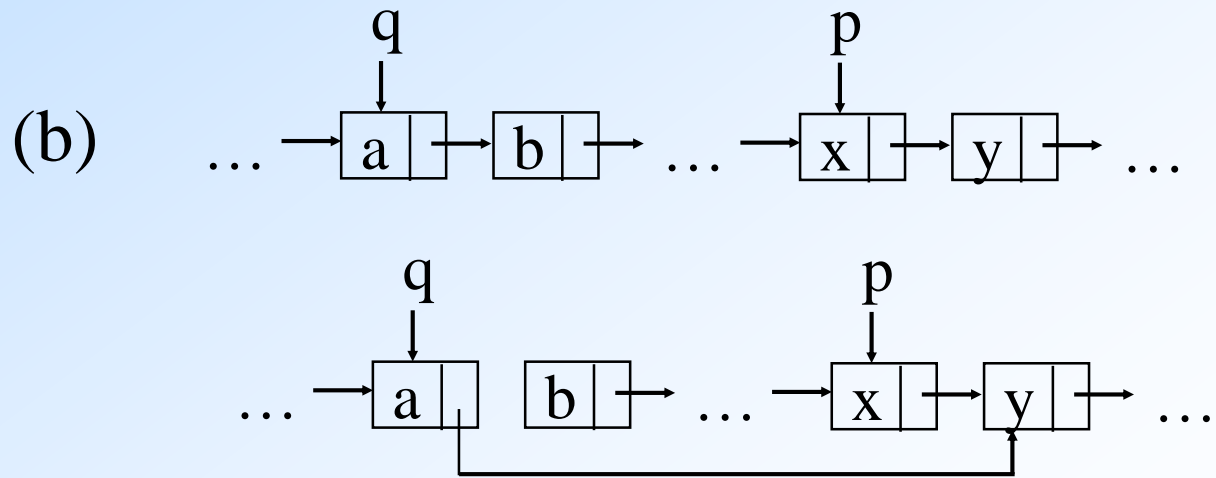
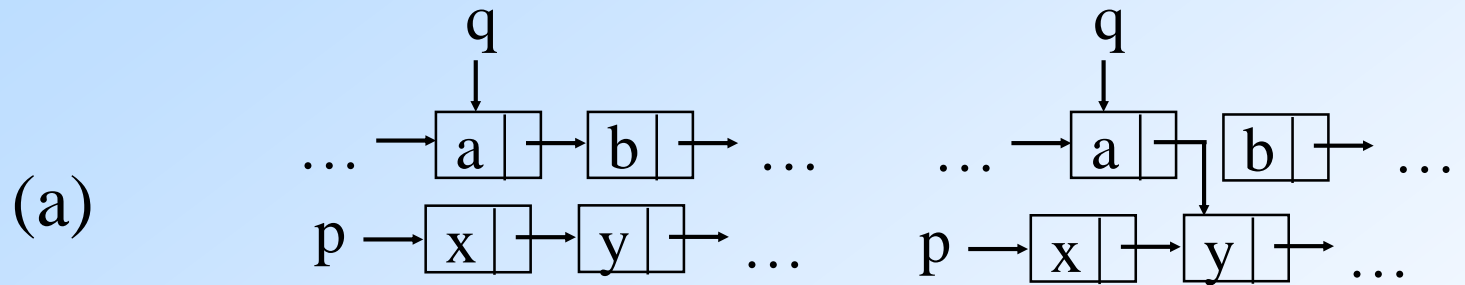


常见的指针操作

④ $q \rightarrow \text{next} = p$;



⑤ $q \rightarrow \text{next} = p \rightarrow \text{next}$;



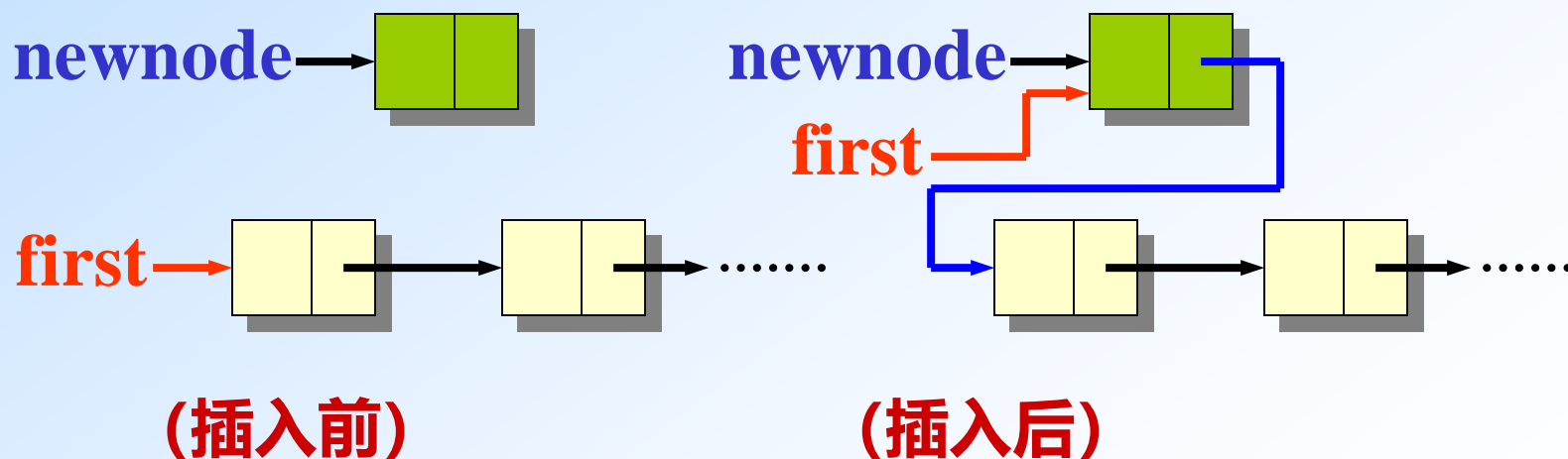
■ 单链表的基本运算

■ 插入（三种情况）

• 第一种情况：在第一个结点前插入

$\text{newnode} \rightarrow \text{next} = \text{first};$

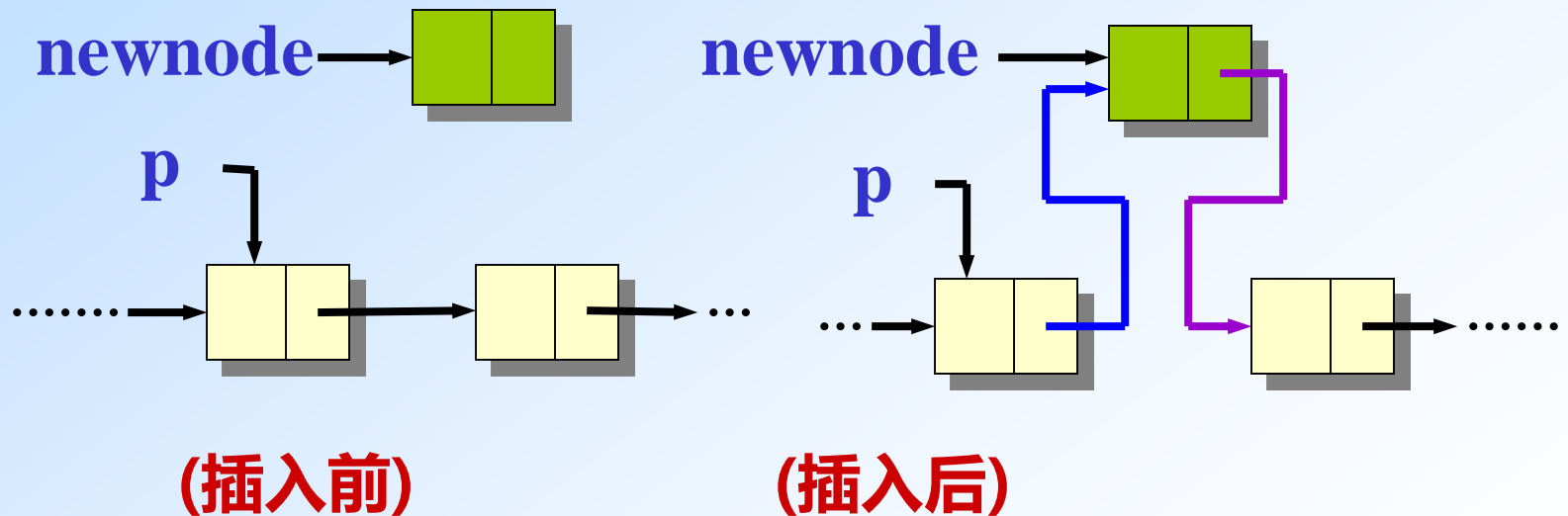
$\text{first} = \text{newnode};$



- **第二种情况：在链表中间插入**

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

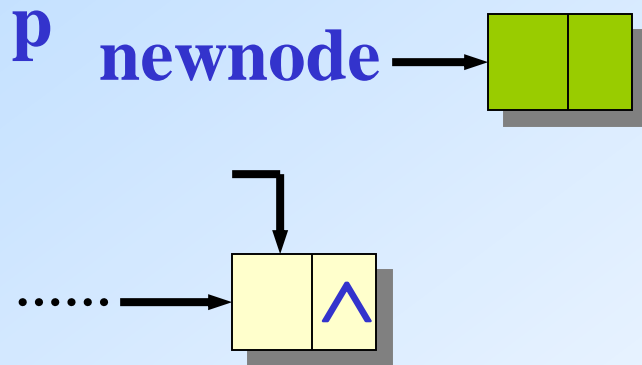
$\text{p} \rightarrow \text{next} = \text{newnode};$



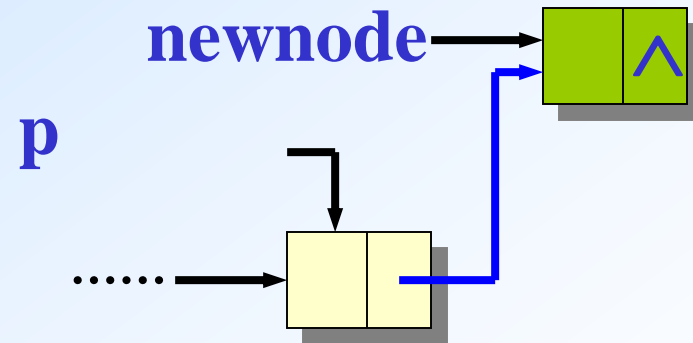
- **第三种情况：在链表末尾插入**

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

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



(插入前)



(插入后)

算法描述(略)

```
int Insert ( LinkList& first, int x, int i ) {  
    //在链表第 i 个结点前插入新元素 x  
    ListNode * p = first;  int k = 0;  
    while ( p != NULL && k < i -1 )  
        { p = p->next; k++; }    //找第 i-1 个结点  
    if ( p == NULL && first != NULL ) {  
        printf ( “无效的插入位置!\n” ); //终止插入  
        return 0;  
    }  
    ListNode * newnode =          //创建新结点  
        (ListNode *) malloc ( sizeof (ListNode) );
```



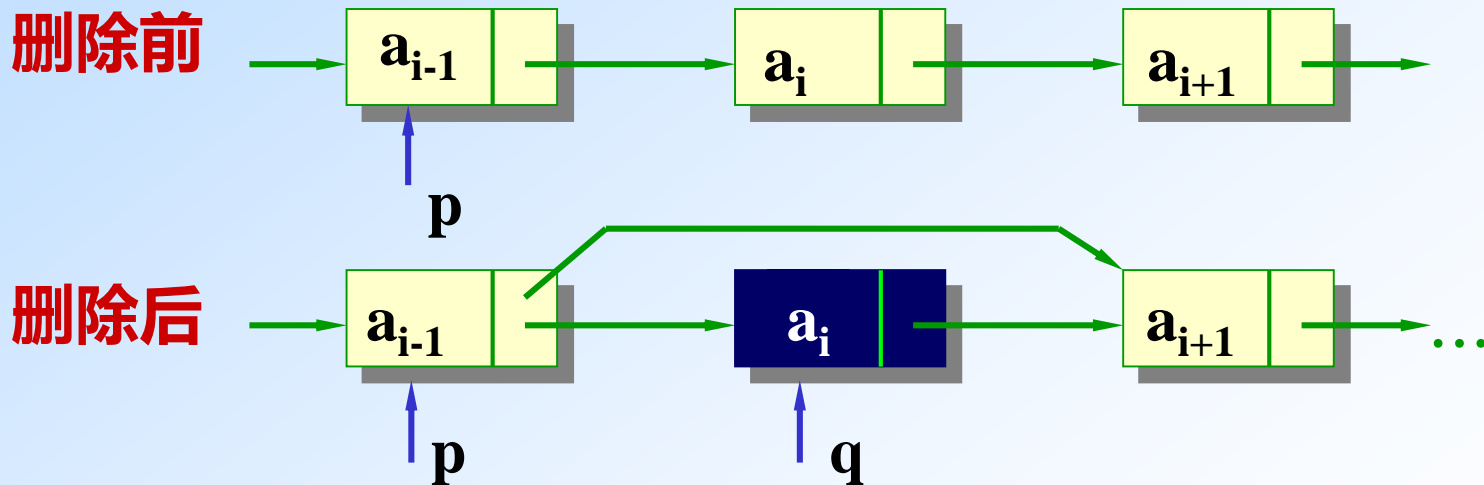
```
newnode->data = x;
if ( first == NULL || i == 1 ) { //插入空表或非空表第一个
    结点之前
        newnode->next = first;//新结点成为第一个结点
        if(first==NULL)last=newnode;//若是空表，表尾指针
        指向新结点
        first = newnode;
    }
    else { //插在表中间或末尾
        newnode->next = p->next;
        if(p->next ==NULL)last=newnode;
        p->next = newnode;
    }
    return 1;
}
```

■ 删除

在单链表中删除 a_i 结点

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

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



```
(略) int Delete ( LinkList& first, int i ) {  
//在链表中删除第 i 个结点  
    ListNode *p, *q;  
    if ( i == 0 ) //删除表中第 1 个结点  
        { q = first; first = first->next; }  
    else {  
        p = first; int k = 0;  
        while ( p != NULL && k < i-1 )  
            { p = p->next; k++; } //找第 i-1 个结点
```

```
if ( p == NULL || p->next == NULL ) {//找不到第  
i-1个结点
```

```
    printf ( “无效的删除位置!\n” );  
    return 0;
```

```
}
```

```
else //删除中间结点或尾结点元素
```

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

```
}
```

```
if (q==last) last=p;//删除表尾结点
```

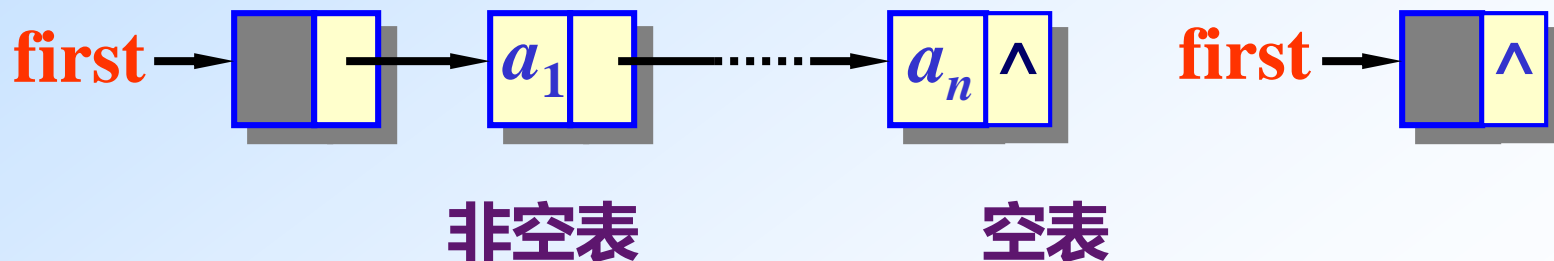
```
k= q->data; free ( q ); return k; //取出被删结点数据并释  
放q
```

```
}
```

```
}
```

带表头结点的单链表

- 表头结点位于表的最前端，本身不带数据，仅标志表头。
- 设置表头结点的目的：
简化链表操作的实现。

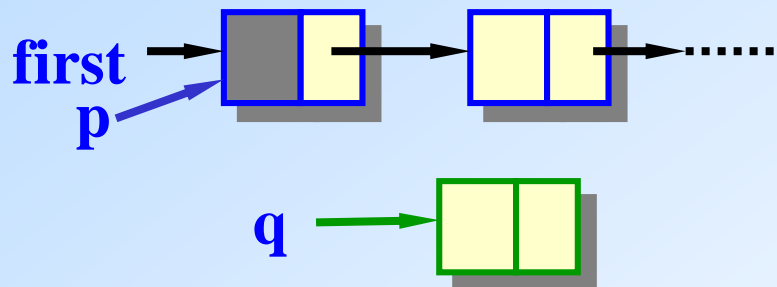


■ 插入

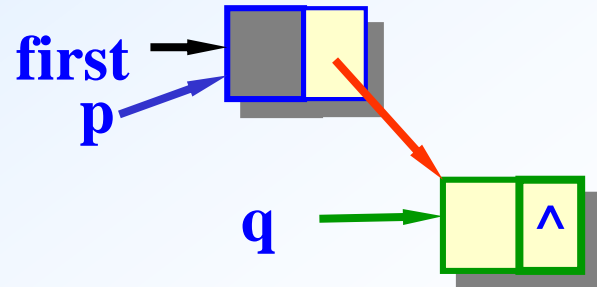
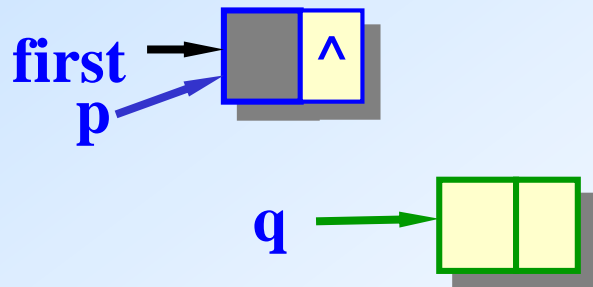
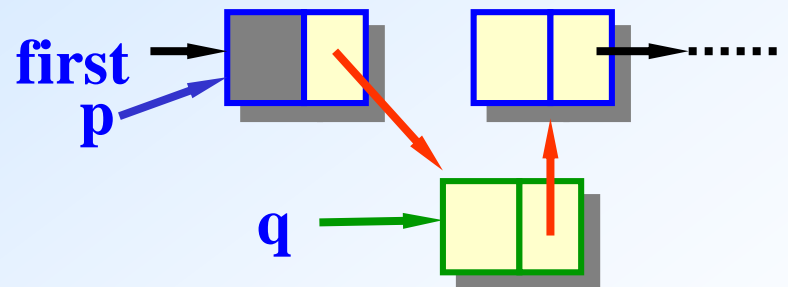
$q \rightarrow \text{next} = p \rightarrow \text{next};$

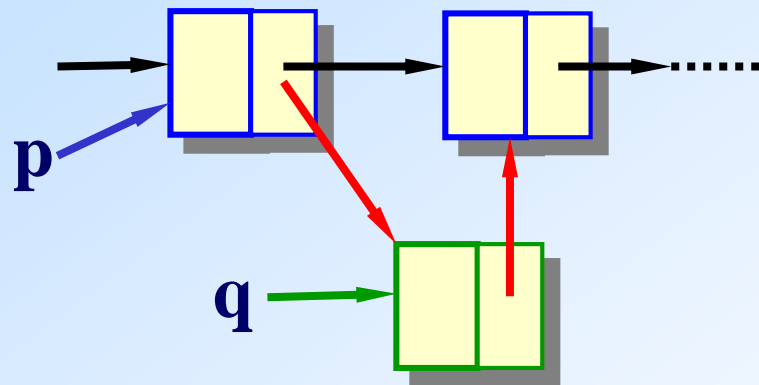
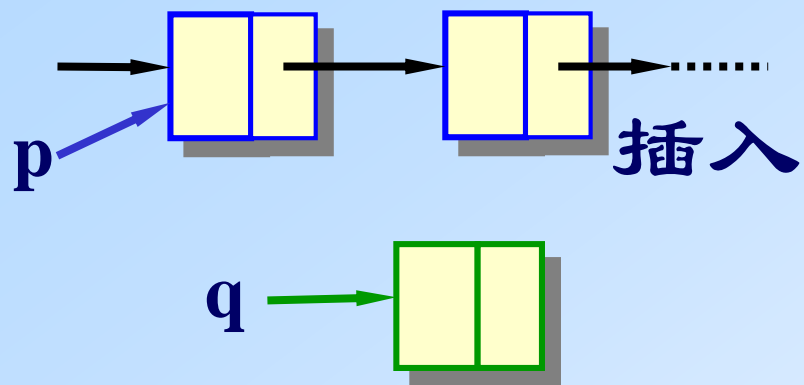
$p \rightarrow \text{next} = q;$

插入前



插入后





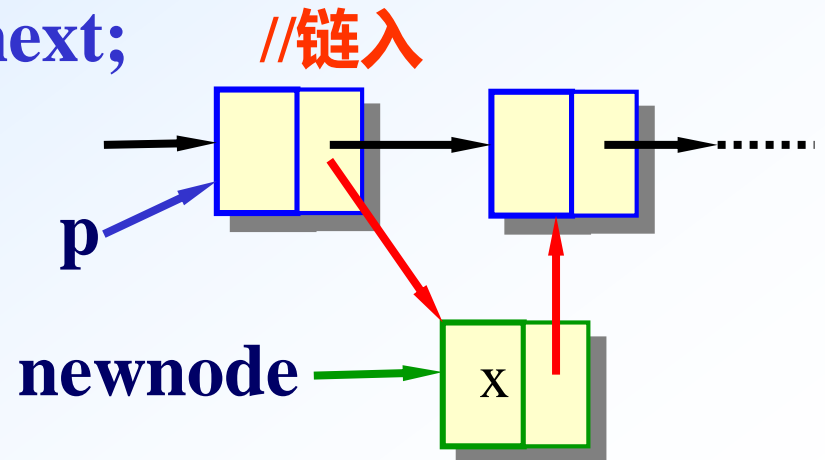
```

int Insert (LinkList first, Elemtype x, int i ) {
//将新元素 x 插入在链表中第 i 号结点位置
    ListNode * p;
    p = Locate ( first, i-1 );
    if ( p == NULL ) return 0;    //参数i值不合理返回0
    ListNode * newnode =         //创建新结点
        (ListNode *) malloc (sizeof (ListNode) );
    newnode->data = x;

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

    return 1;
}

```



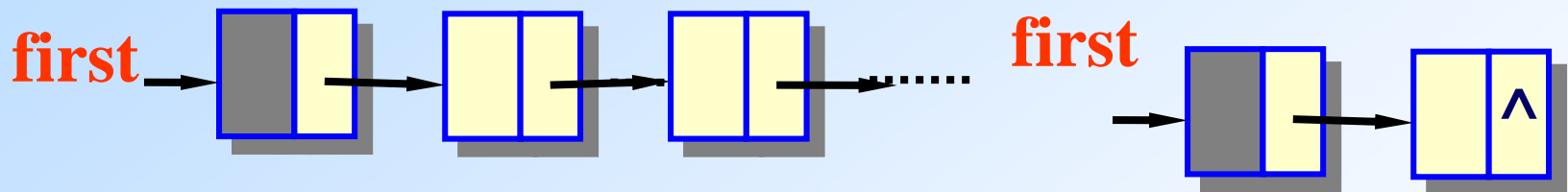
■ 删除

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

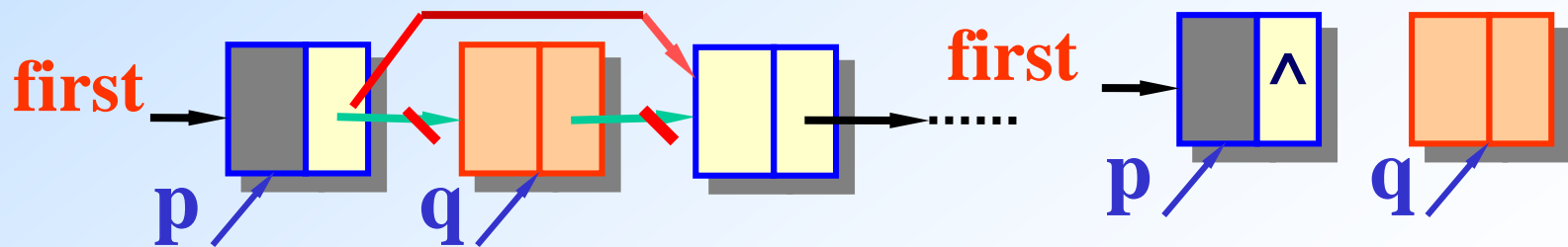
$p \rightarrow \text{next} = q \rightarrow \text{next};$

$\text{free}(q);$ //释放空间, c++ 操作符 delete

删除前



删除后



(非空表)

(空表)

```
int delete ( LinkList first, int i ) {  
    //将链表第 i 号元素删去  
    ListNode * p, * q  
    p = Locate ( first, i-1 ); //寻找第i-1个结点  
    if ( p == NULL || p->next == NULL)  
        return 0; //i值不合理或空表  
  
    q = p->next;  
    p->next = q->next; //删除结点  
  
    free (q); //释放  
  
    return 1;  
}
```

计算单链表长度

```
int Length ( LinkList first ) {  
    ListNode *p;  
    p = first->next; //指针 p 指示第一个结点  
    int count = 0;  
    while ( p != NULL ) {  
        p = p->next; count++;  
    }  
    return count;  
}
```

单链表清空

```
void makeEmpty ( LinkList first ) {  
    //删去链表中除表头结点外的所有其它结点  
    ListNode *q;  
    while ( first->next != NULL ) {  
        //当链不空时，循环逐个删去所有结点  
        q = first->next; first->next = q->next;  
        free( q );  
    }  
}
```

按值查找

```
ListNode * Find ( LinkList first, Elemtype value ) {  
    //在链表中从头搜索其数据值为value的结点  
    ListNode * p ;  
    p= first->next;      //指针 p 指示第一个结点  
    while ( p != NULL && p->data != value )  
        p = p->next;  
    return p;  
}
```

按序号查找 (定位)

```
ListNode * Locate ( LinkList first, int i ) {
```

```
//返回表中第 i 个元素的地址
```

```
    ListNode * p ;    int k ;
```

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

```
    p= first; k= 0;
```

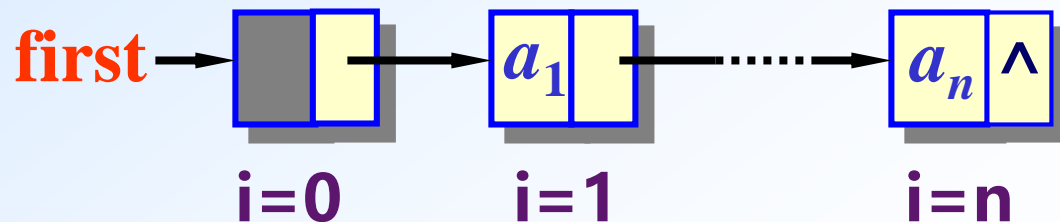
```
    while ( p != NULL && k < i )
```

```
        { p = p->next; k++; }           //找第 i 个结点
```

```
    if ( k == i ) return p; //返回第 i 个结点的地址
```

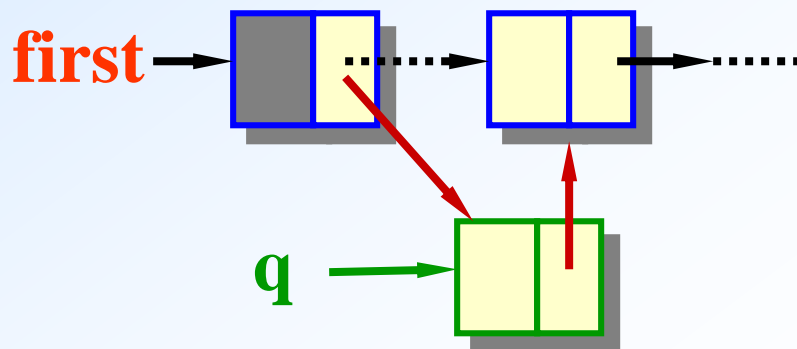
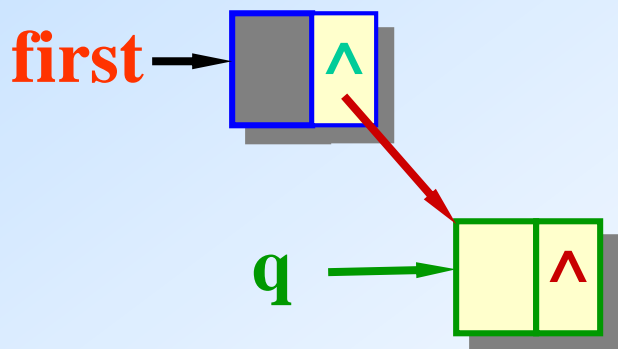
```
    else return NULL;
```

```
}
```



头插法建立单链表

- 从一个空表开始，重复读入数据：
 - 生成新结点
 - 将读入数据存放到新结点的数据域中
 - 将该新结点插入到链表的前端
- 直到读入结束符为止。



```
Void createListF ( Linklist *first,Elemtype a[],int n)
```

```
//形式参数first为指针参数，（指向指针的指针）
```

```
{ int i; ListNode *q;
```

```
 (*first)= (LinkList) malloc (sizeof (ListNode));
```

```
                //建立表头结点
```

```
 (*first)->next = NULL;
```

```
for (i=n-1;i>=0;i--)
```

```
 { q = (listNode *) malloc (sizeof(ListNode));
```

```
   q->data = a[i];
```

```
   q->next = (*first)->next;
```

```
   (*first)->next = q;
```

```
 }
```

```
}
```

```
Main( )
```

```
{ Linklist L;
```

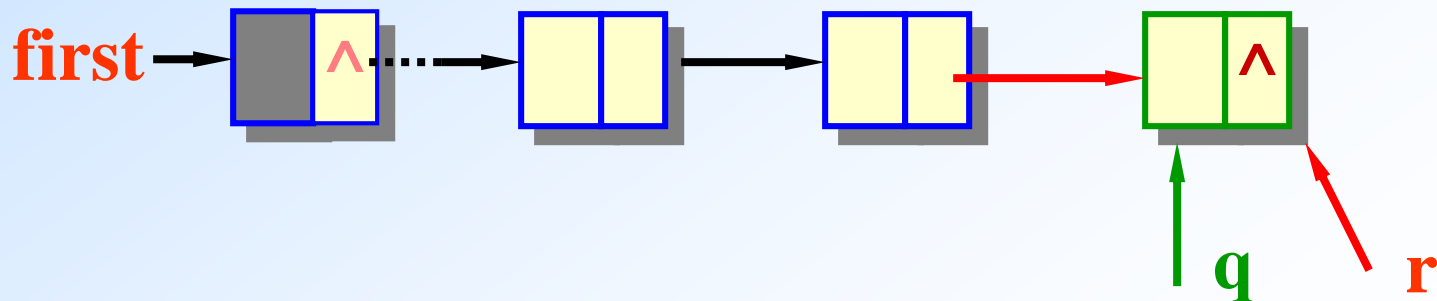
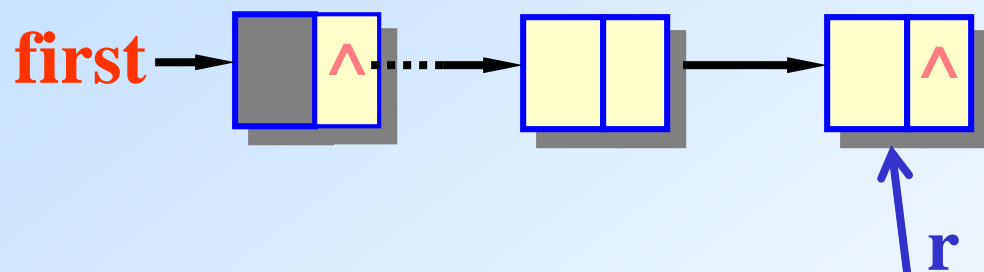
```
    ...
```

```
    creatListF(&L, , );
```

```
}
```


尾插法建立单链表

- 每次将新结点加在链表的表尾；
- 设置尾指针 r ,总是指向表中最后一个结点, 新结点插在它的后面；

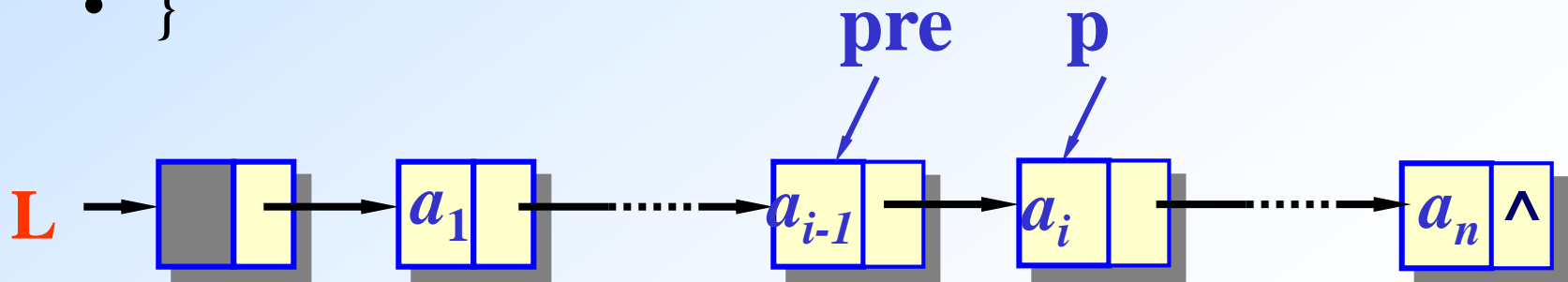


```
LinkedList createListR ( ) {//函数值返回头结点指针
    char ch;
    LinkedList first = //建立表头结点
        (LinkedList) malloc (sizeof (ListNode));

    ListNode *q, *r = first;
    while ( (ch = getchar( ) ) != '\n' ) {
        q = (ListNode *) malloc (sizeof(ListNode));
        q->data = ch;
        r ->next = q; r =q; //插入到表末端
    }
    r ->next = NULL;
    return first; {//返回函数值
}
```

练习1：设计一个算法，在带头结点的单链表L中删除所有值为x的结点并释放空间，假设这样的结点不是唯一的。

- Void del(linklist L,elemtype x)
- {ListNode * p=L->next ,*pre=L,*q;
- While (p!=NULL)
- {if (p->data==x)
- {q=p; p=p->next;
- pre->next=p; free(q);
- }
- Else { pre=p ; p=p->next ;}
- }
- }



练习2：设计一个算法，在带头结点的单链表L中删除一个最小值结点，假设其唯一。

Void delminnode(Linklist L)

```
{Listnode *pre=L, *p=pre->next, *minp=p, *minpre=pre;
```

```
while (p!=NULL)
```

```
    //查找最小值节点 *minp及其前趋节点 *minpre
```

```
    { if (p->data<minp->data)
```

```
        { minp=p; minpre=pre;}
```

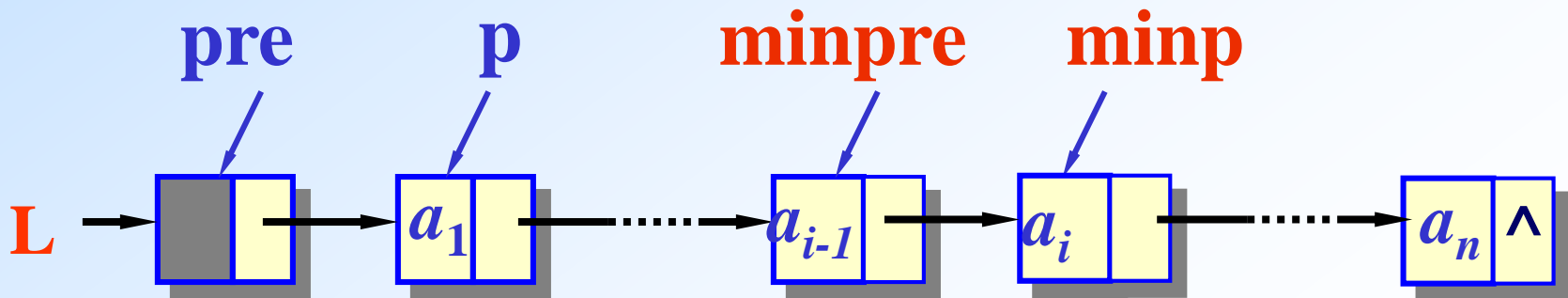
```
    pre=p;
```

```
    p=p->next; //p、pre同步后移一个节点
```

```
}
```

```
minpre->next=minp->next ; free (minp);
```

```
}
```



练习3： 设 $C=\{a_1, b_1, a_2, b_2, \dots, a_n, b_n\}$ 为一线性表，采用带头结点的hc单链表存放，设计一个就地算法，将其拆分为两个单链表ha和hb。

```
Void split(Linklist hc, Linklist *ha, Linklist *hb)
```

```
{ Listnode *p=hc->next,*ra,*rb;
```

```
  * ha=hc; ra=hc;
```

```
  *hb=(Linklist)malloc(sizeof(linknode));  rb=*hb;
```

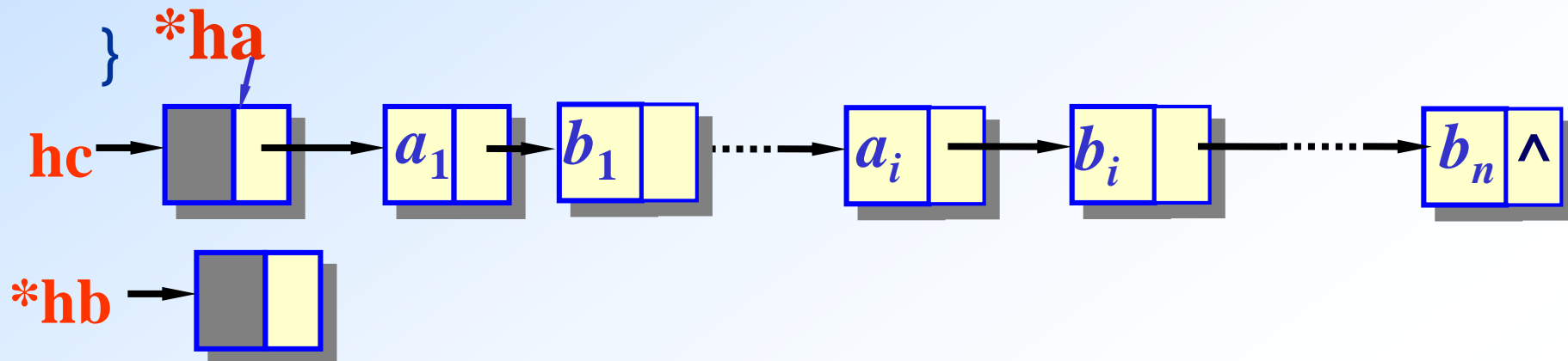
```
  while(p!=NULL)
```

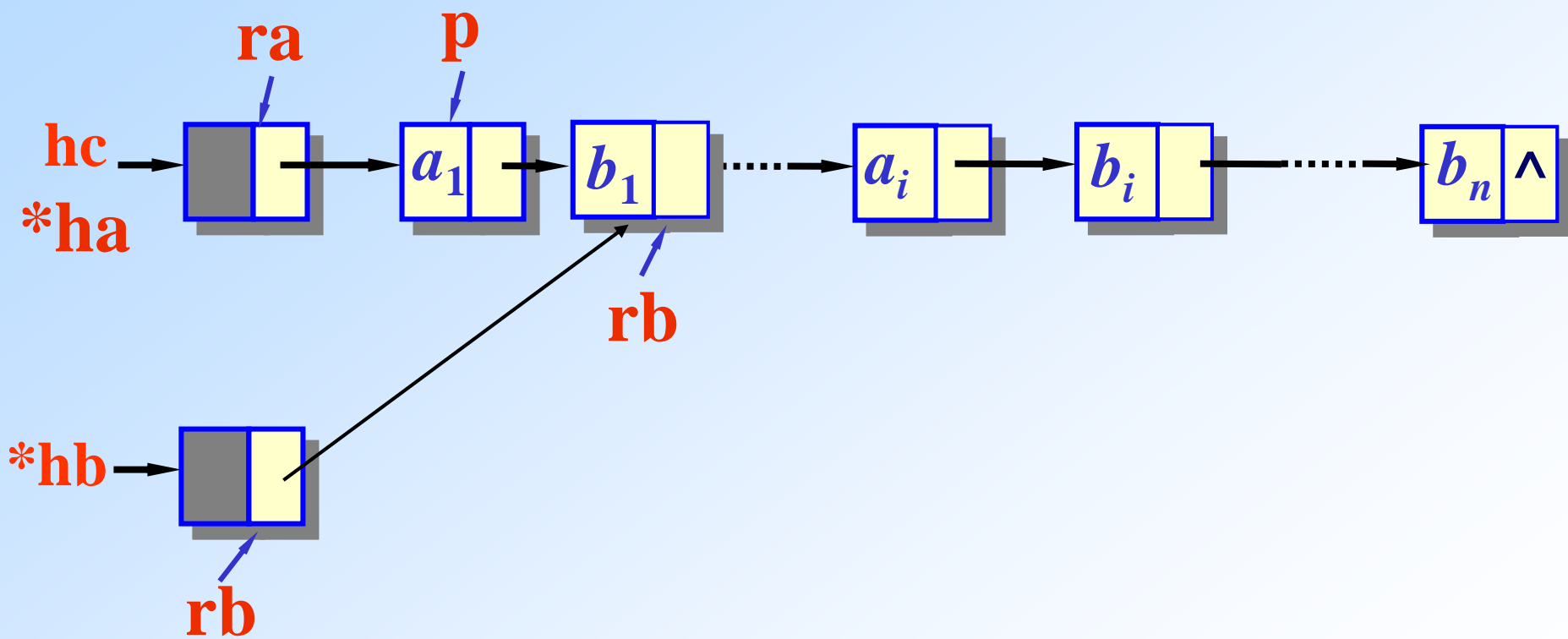
```
  { ra->next=p; ra=p;  p=p->next;
```

```
    if (p!=NULL){ rb->next=p; rb=p ; p=p->next ; }
```

```
  }
```

```
  ra->next=rb->next=NULL;
```





```
main()
```

```
{lnode *l,*la,*lb;
```

```
    l=(lnode *)malloc(sizeof(lnode));
```

```
    createListR(l);
```

```
    splitl(l,&la,&lb);
```

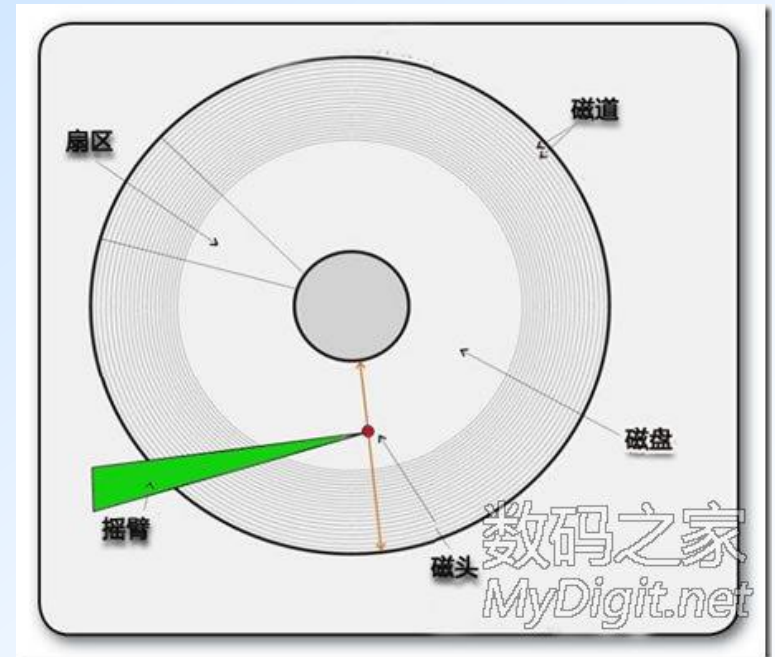
```
}
```

在一个长度为 n 的带头结点的单链表 h 上，另设尾指针 r ，执行——^{**b**}——操作与链表的长度有关。

- (a) 删除单链表中的首结点
- (b) 删除单链表中的尾结点
- (c) 在单链表首结点前插入一个新结点
- (d) 在单链表尾结点后插入一个新结点

FAT（File Allocation Table，文件分配表）文件系统是windows操作系统所使用的一种文件系统

- 扇区：磁盘上的每个磁道被等分为若干个弧段，这些弧段便是磁盘的扇区，每个扇区可以存放512个字节的信息。
- 磁盘是由一个一个扇区组成的，若干个扇区合为一个簇，硬盘每簇的扇区数与硬盘的总容量大小有关，可能是4、8、16、32、64……文件存取是以簇为单位的。



文件分配表FAT

- FAT表(文件分配表)，是文件系统中用于磁盘数据索引和定位而引进的一种链式结构。在FAT文件系统中，文件的存储依照FAT表制定的簇链式数据结构来进行。
- FAT表示磁盘文件的空间分配信息，不真正存储文件内容。同一个文件的数据不一定完整地存放在磁盘的一个连续的区域內，而往往会分成若干段，像一条链子一样存放。这种存储方式称为文件的链式存储。由于硬盘上保存着段与段之间的连接信息（即FAT），操作系统在读取文件时，总是能够准确地找到各段的位置并正确读出。

- 为了安全起见，FAT有一个备份。**文件目录表FDT**表是用于登记管理磁盘文件的名称、类型、文件属性、文件建立或修改时间和日期、文件的首簇号以及文件长度等信息的表格。从文件目录表（FDT）中查得该文件的起始簇号。

表1 FAT32短文件目录项32个字节的表示定义			
字节偏移(16进制)	字节数	定义	
0x0~0x7	8	文件名	
0x8~0xA	3	扩展名	
0xB*	1	属性字节	00000000(读写) 00000001(只读) 00000010(隐藏) 00000100(系统) 00001000(卷标) 00010000(子目录) 00100000(归档)
0xC	1	系统保留	
0xD	1	创建时间的10毫秒位	
0xE~0xF	2	文件创建时间	
0x10~0x11	2	文件创建日期	
0x12~0x13	2	文件最后访问日期	
0x14~0x15	2	文件起始簇号的高16位	
0x16~0x17	2	文件的最近修改时间	
0x18~0x19	2	文件的最近修改日期	
0x1A~0x1B	2	文件起始簇号的低16位	
0x1C~0x1F	4	表示文件的长度	

*此字段在短文件目录项中不可取值0FH,如果设值为0FH,目录段为长文件名目录段

文件分配表FAT

- FAT表有两个重要作用：描述簇的分配状态以及标明文件或目录的下一簇的簇号。为了实现文件的链式存储，硬盘上必须准确地记录哪些簇已经被文件占用，还必须为每个已经占用的簇指明存储后继内容的下一个簇的簇号。
- 如果该文件结束于该簇，则在它的FAT表项中记录的是一个文件结束标记，对于FAT32而言，代表文件结束的FAT表项值为0x0FFFFFFF。

静态链表

用一维数组描述线性链表

0		1
1	ZHANG	2
2	WANG	3
3	LI	4
4	ZHAO	5
5	WU	-1
6		
7		

0		1
1	ZHANG	2
2	WANG	6
3	LI	5
4	ZHAO	?
5	WU	-1
6	CHEN	3
7		

(插入chen,删除zhao)修改后



■ 定义

```
#define MaxSize 100    //静态链表大小
```

```
typedef int Elemtyp;
```

```
typedef struct node {    //静态链表结点  
    Elemtyp data;  
    int next;  
} SNode;
```

```
typedef struct {    //静态链表  
    SNode Nodes[MaxSize];  
    int newptr;    //当前可分配空间首地址  
} SLinkList;
```



静态链表

		data	next
L.Nodes[0] →	0		1
	1	ZHANG	2
	2	WANG	3
	3	LI	4
	4	ZHAO	5
	5	WU	-1
L.newptr →	6		
	7		



静态链表

		data	next
L.Nodes[0] →	0		1
	1	ZHANG	2
	2	WANG	6
	3	LI	5
L.newptr →	4	ZHAO	?
	5	WU	-1
	6	CHEN	3
	7		

(插入chen,删除zhao)修改后



链表空间初始化

```
void InitList ( SLinkList *SL ) {  
    int i;  
    (*SL).Nodes[0].next = 1;  
    (*SL).newptr = 1; //当前可分配空间从 1 开始  
    //建立带表头结点的空链表  
    for (i = 1; i < MaxSize-1; i++ )  
        (*SL).Nodes[i].next = i+1; //构成空闲链接表  
    (*SL).Nodes[MaxSize-1].next = 0; //链表收尾  
}
```

- main()
- {SLinkList l;
- InitList(&l);}



在静态链表中查找具有给定值的结点

```
int Find ( SLinkList SL, Elemtypex ) {  
    int p = SL.Nodes[0].next;  
    //指针 p 指向链表第一个结点  
    while ( p != -1 )//逐个查找有给定值的结点  
        if ( SL.Nodes[p].data != x)  
            p = SL.Nodes[p].next;  
    else break;  
    return p;  
}
```



在静态链表中查找第 i 个结点

```
int Locate ( SLinkList SL, int i ) {  
    int j, p;  
    if ( i < 0 ) return -1; //参数不合理  
    j = 1; p = SL.Nodes[0].next;  
    while ( p != -1 && j < i ) { //循环查找第 i 号结点  
        p = SL.Nodes[p].next;  
        j++;  
    }  
    if ( i == 0 ) return 0;  
    return p;  
}
```



在静态链表第 i 个结点处插入一个新结点

```
int Insert ( SLinkList *SL, int i, Elemtype x ) {  
    int p, q;  
    p = Locate ( *SL, i-1 );  
    if ( p == -1 ) return 0;           //找不到结点  
    q = (*SL). newptr;                 //分配结点  
    (*SL). newptr = (*SL). Nodes[(*SL). newptr].next;  
    (*SL). Nodes[q].data = x;  
    (*SL). Nodes[q].next = (*SL). Nodes[p].next;  
    (*SL). Nodes[p].next = q;         //插入  
    return 1;  
}
```



在静态链表中释放第 i 个结点

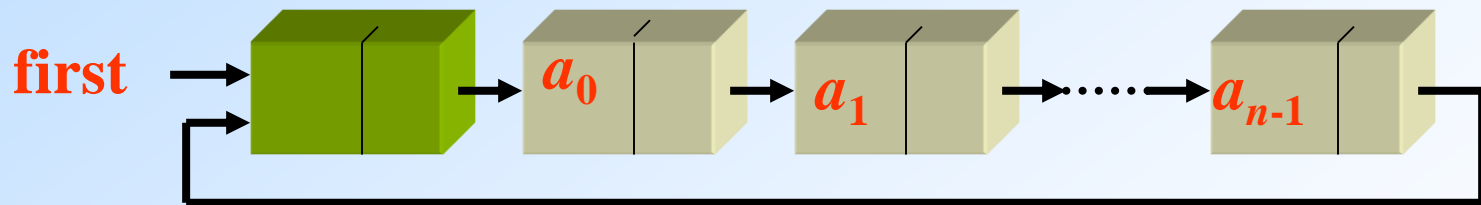
```
int Remove ( SLinkList *SL, int i ) {  
    int p = Locate ( *SL, i-1 );  
    if ( p == -1 ) return 0;      //找不到结点  
    int q = (*SL).Nodes[p].next; //第 i 号结点  
    (*SL). Nodes[p].next = (*SL). Nodes[q].next;  
    (*SL). Nodes[q].next = (*SL). newptr;  //释放  
    (*SL). newptr = q;  
    return 1;  
}
```



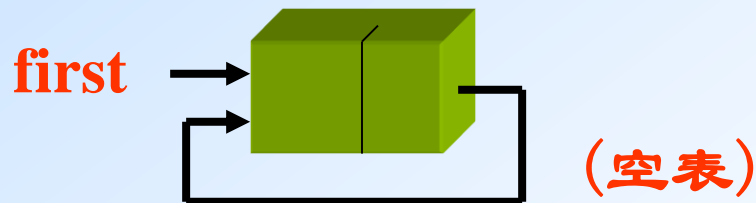
循环链表 (Circular List)

- **特点:**最后一个结点的 next 指针不为NULL, 而是指向头结点。只要已知表中某一结点的地址, 就可搜寻所有结点的地址。
- **存储结构:**链式存储结构

带表头结点的循环链表



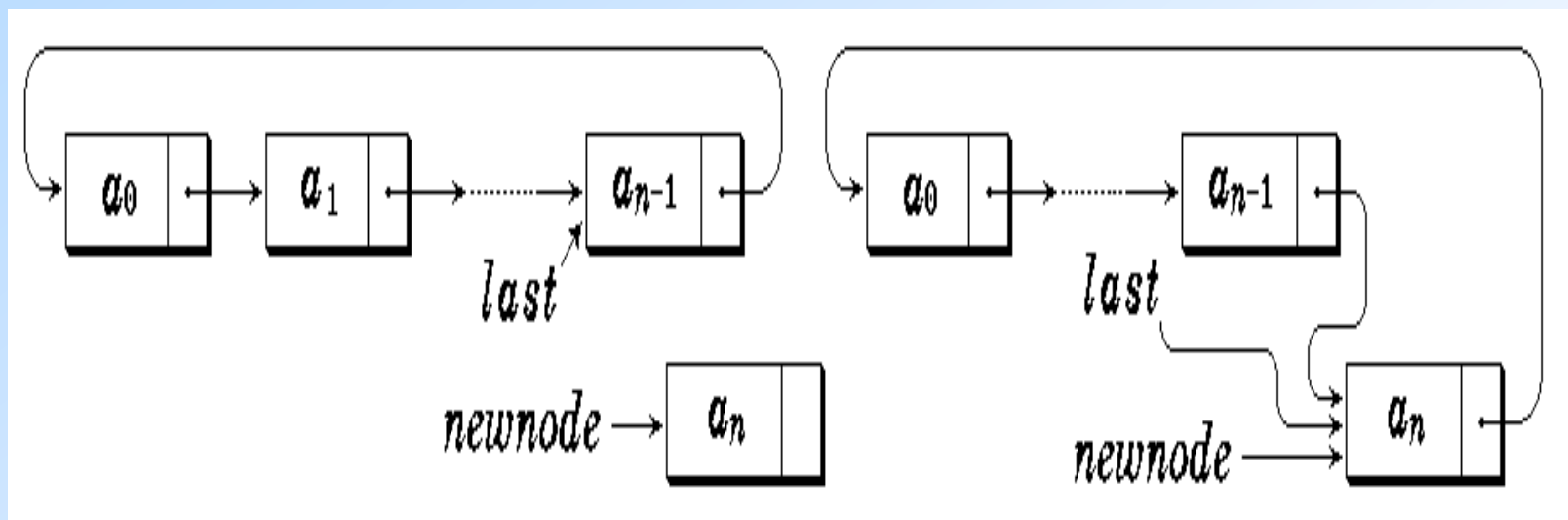
(非空表)



(空表)



循环链表的插入

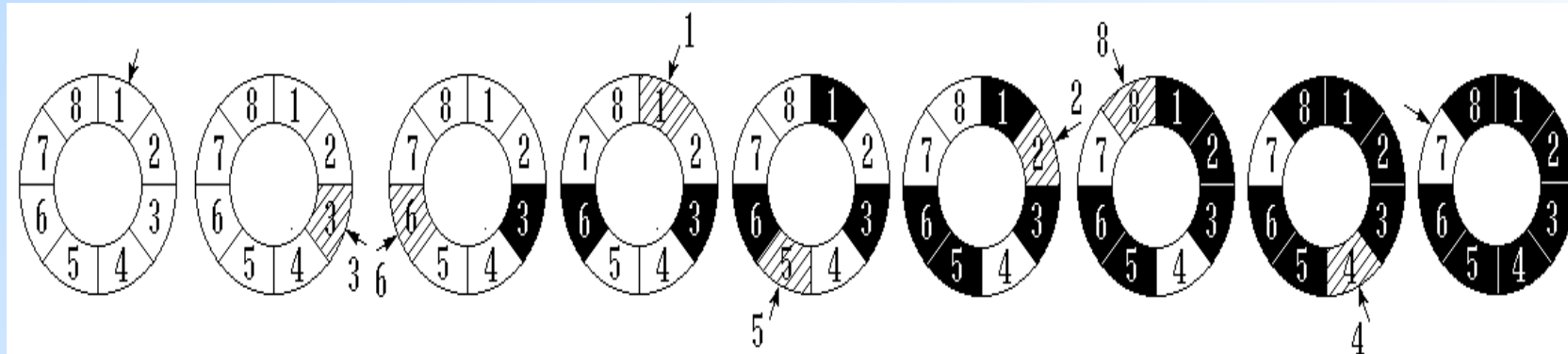


- 某些情况需要把结点组成环形链表。例如，多个进程在一段时间内访问同样的资源，为了保证每一个进程可以公平地分享这个资源，可以把这些进程组织在循环链表结构中。可以通过指针访问该结构，指针指向的结点即为将要激活的进程。随着指针的移动，可以激活每一个进程。



约瑟夫问题(附：叶周课件)

- 用循环链表求解约瑟夫问题



约瑟夫问题

- 这是17世纪的法国数学家加斯帕在《数目的游戏问题》中讲的一个故事：
- 15个教徒和15个非教徒在深海上遇险，必须将一半的人投入海中，其余的人才能幸免于难，于是想了一个办法：30个人围成一圆圈，从第一个人开始依次报数，每数到第九个人就将他扔入大海，如此循环进行直到仅余15个人为止。问怎样排法，才能使每次投入大海的都是非教徒。



约瑟夫问题

数学问题描述：设有 n 个人围成一圈，每个人的编号依次为 $1, 2, 3, \dots, n$ 。现从编号为 k （ $1 \leq k \leq n$ ）的人开始报数，报到 m （ $1 \leq m \leq n$ ）的人出列，接着从出列的下一个开始重新报数，报到 m 的人又出列，如此下去，直到所有要出列的人都出列为止。设计程序求这 n 个人的出列次序及最后的出列者。





分析：我们构造一单向循环链表，每个节点的数据为1~10，从第一个节点开始计数，设置一个节点变量指向第一个节点，每计一个数该节点变量指向下一个，计满3则删除当前指向的节点，并指向下一个节点重新开始计数，只要计满3就删除当前节点，如此继续直至删除所有节点，并在中间记录下依次删除的节点。

如图所示，黑色数字为每个人的编号，红色数字为依次退出的次序。



```
#include<stdio.h>
#include<malloc.h>

//定义单向循环链表节点的存储结构
typedef struct lnode
{
int num; //数据域
struct lnode *next; //指针域
}node,*L; //节点类型，节点指针

void main()
{
int amount,mount,start,circle,n,c;
L head,p,q; //定义头节点及中间辅助节点变量

printf("一共几个人围成一圈： "); //输入一圈总人数
scanf("%d",&amount);
printf("一共需要退出的人数： "); //输入需退出人数
scanf("%d",&mount);
printf("从第几个人开始计数： "); //设置计数开始点
scanf("%d",&start);
printf("每几人一次循环报数： "); //输入循环报数值
scanf("%d",&circle);
```




```
head=(L)malloc(sizeof(node)); //生成头节点
head->next=NULL;
head->num=0;
q=head; //用来链接下一节点
n=0; //用来给每个人记标号 (1~amount)
while(n++<amount)
{
    p=(L)malloc(sizeof(node)); //生成新节点
    p->next=NULL;
    p->num=n;
    q->next=p;
    q=p;
}
q->next=head->next; //形成循环链表
//以上完成单向循环链表的建立
```

```
p=head->next; //p赋为第一个节点
q=head; //q赋为p的上一节点，即头节点
n=1; //用来寻找开始计数点的位置
while(n++<start) //在未找到开始计数点前，p，q依次后移
{
    p=p->next;
    q=q->next;
}
//当退出循环时p，q分别位于指定位置，q为p的上一节点
```

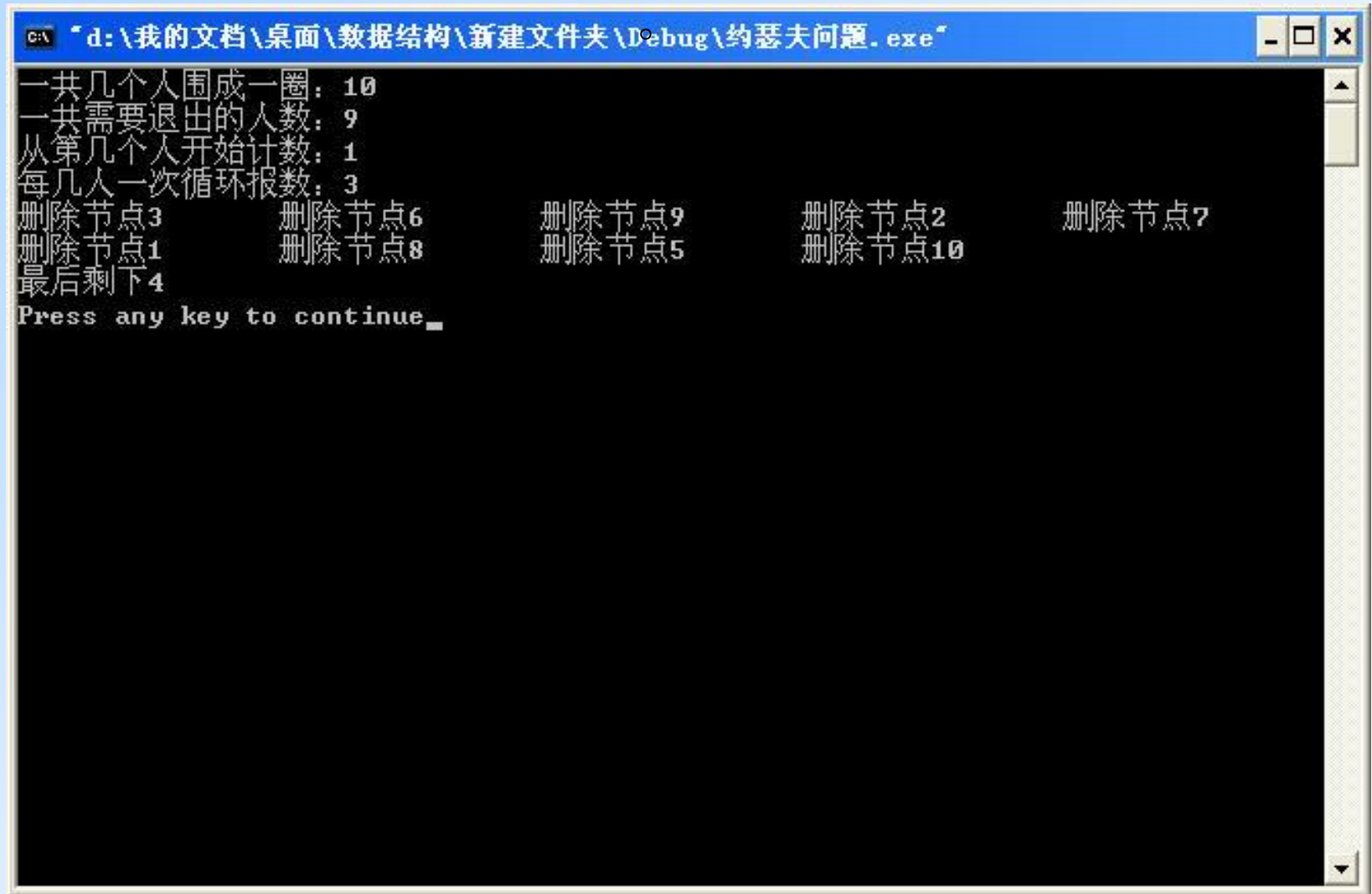


```
//接下来进行周期性节点删除，直到删除数目达到mount为止
n=1; //n计算被删除的节点数目，当n=mount时删除结束
while(n++<=mount)
{
    c=1; //c作为循环报数值临时变量
    while(c++<circle)
    {
        p=p->next;
        q=q->next;
    }
    //删除当前p指向的节点
    printf("删除节点%d\t",p->num); //输出删除节点的标号
    q->next=p->next; //删除节点p
    p=p->next; //p赋新值，原节点的下一节点
}
printf("\n");
n=1;
while(n++<=amount-mount) //n计算输出剩余节点的数目，当n=amount-mount时全部输出完毕
{
    printf("最后剩下%d\t",p->num); //输出剩余节点的标号
    p=p->next; //p下移
}
printf("\n");
}
```



运行结果

这里令amount=10, mount=9, start=1, circle=3



```
C:\ "d:\我的文档\桌面\数据结构\新建文件夹\Debug\约瑟夫问题.exe"
一共几个人围成一圈: 10
一共需要退出的人数: 9
从第几个人开始计数: 1
每几人一次循环报数: 3
删除节点3      删除节点6      删除节点9      删除节点2      删除节点7
删除节点1      删除节点8      删除节点5      删除节点10
最后剩下4
Press any key to continue_
```



现在我们可以来解决最原始的约瑟夫问题了，这里令amount=30，
mount=15，start=1，circle=9，欲使投入大海的15个人都是非
教徒，则只需将15个教徒安排在第25，28，29，1，2，3，4，
10，11，13，14，15，17，20，21个位置就可以了



```
C:\ "d:\我的文档\桌面\数据结构\新建文件夹\Debug\约瑟夫问题.exe"
一共几个人围成一圈: 30
一共需要退出的人数: 15
从第几个人开始计数: 1
每几人次循环报数: 9
删除节点9      删除节点18      删除节点27      删除节点6      删除节点16
删除节点26      删除节点7      删除节点19      删除节点30     删除节点12
删除节点24      删除节点8      删除节点22      删除节点5      删除节点23

最后剩下25      最后剩下28      最后剩下29      最后剩下1      最后剩下2
最后剩下3      最后剩下4      最后剩下10     最后剩下11     最后剩下13
最后剩下14      最后剩下15     最后剩下17     最后剩下20     最后剩下21

Press any key to continue
```



*** 链式存储结构 ***

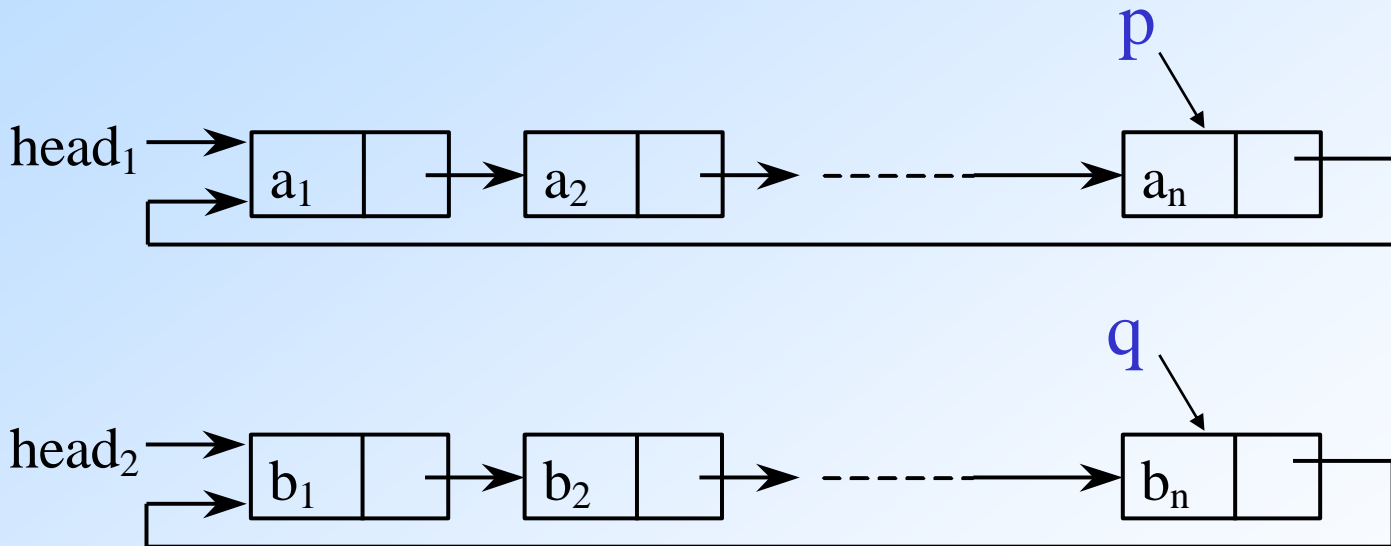
使用一个不带头结点的循环单链表结构。结点结构为：

num	code	next
-----	------	------



练习1:

有两个循环单链表，头指针分为head1和head2，
编写函数将链表head2链接到链表head1之后，
链接后的链表仍保持是循环链表的形式。



先分别找到两个链表的表尾，将head2放入链表 head1的表尾，将两个链表链接起来，然后将head1放入原head2链表的表尾，构成新的循环链表。

```
link(listnode *head1, listnode *head2)
{
    listnode *p,*q;
    p=head1;
    while(p->next!=head1)
        p=p->next;

    q=head2;
    while(q->next!=head2)
        q=q->next;

    p->next=head2;
    q->next=head1;
}
```



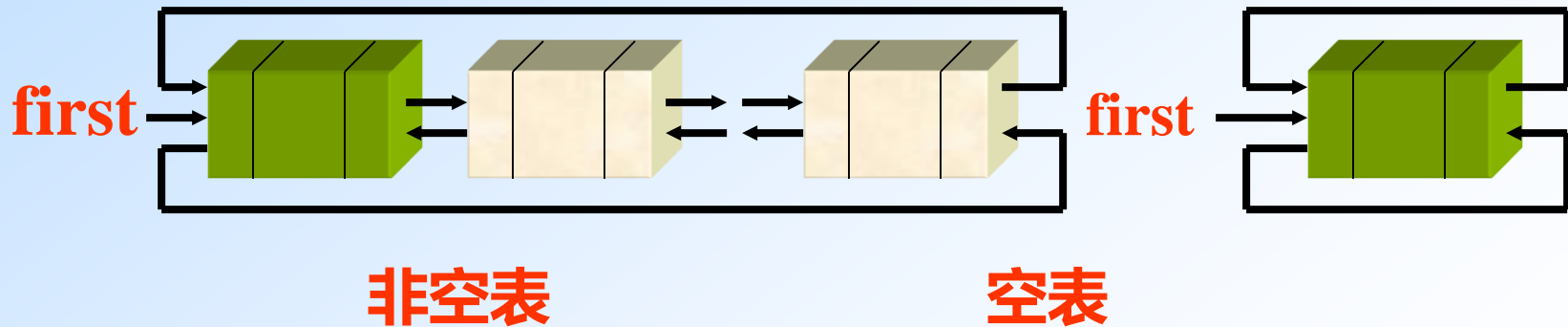
双向循环链表

- 在循环链表中，访问后继结点，只需要向后走一步，而访问前驱结点，就需要转一圈。循环链表并不适用于经常访问前驱结点的情况。
- 在需要频繁地同时访问前驱和后继结点的时候，使用双向链表。所谓**双向链表**就是每个结点有两个指针域:一个指向后继结点，另一个指向前驱结点。



双向链表 (Doubly Linked List)

- 双向链表结点结构：



双向循环链表的定义

```
typedef int Elemtyp;
typedef struct dnode {
    ListNode data;
    struct dnode * prior, * next;
} DbListNode;

typedef DbListNode * DbList;
```



建立空的双向循环链表

```
void CreateDbllist ( Dbllist * first ) {  
    (*first) = ( DbllNode * ) malloc  
        ( sizeof ( DbllNode ) );  
    if ( (*first) == NULL )  
        { print ( “存储分配错!\n” ); exit (1); }  
    (*first) ->prior = (*first) ->next = (*first) ;  
        //表头结点的链指针指向自己  
}
```



计算双向循环链表的长度

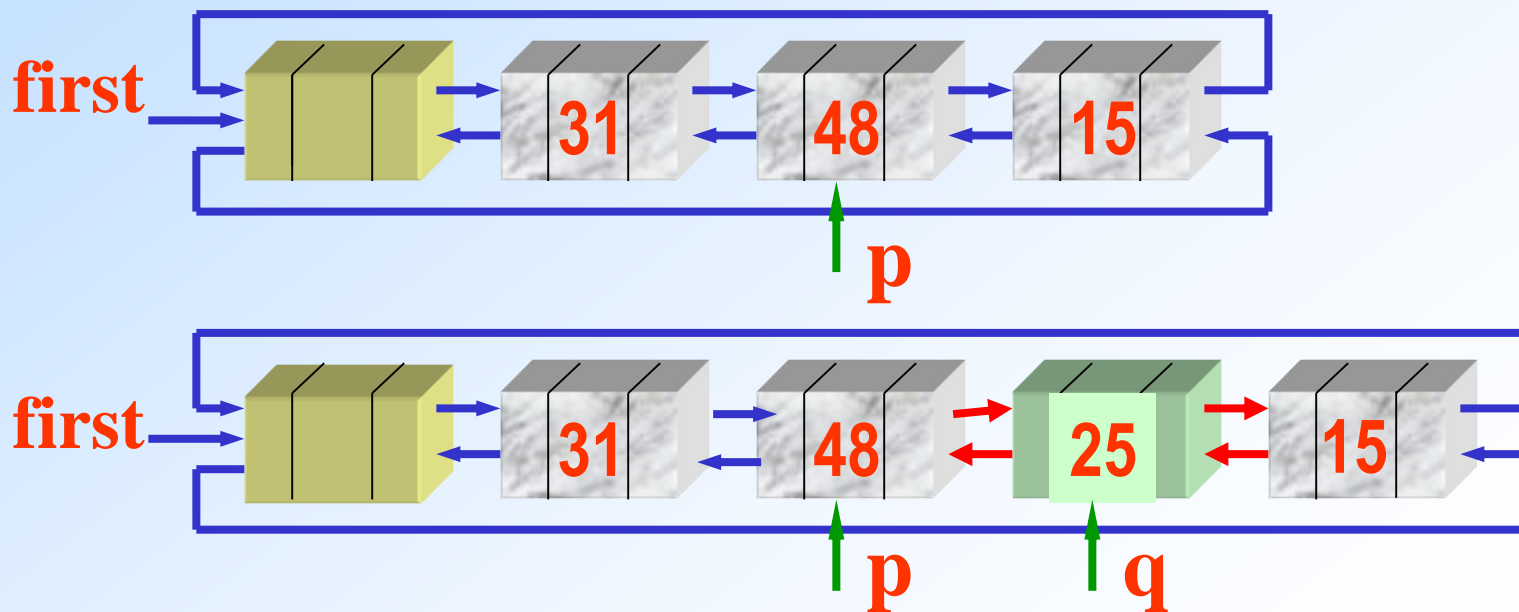
```
int Length ( Dbllist first ) {  
    //计算带表头结点的双向循环链表的长度  
    DbllNode * p = first->next;  
    int count = 0;  
    while ( p != first )  
        { p = p->next; count++; }  
    return count;  
}
```



双向循环链表的插入 (非空表)

在结点 *p 后插入25

```
q->prior = p;  
q->next = p->next;  
p->next = q;  
q->next->prior = q;
```



双向循环链表的插入 (空表)

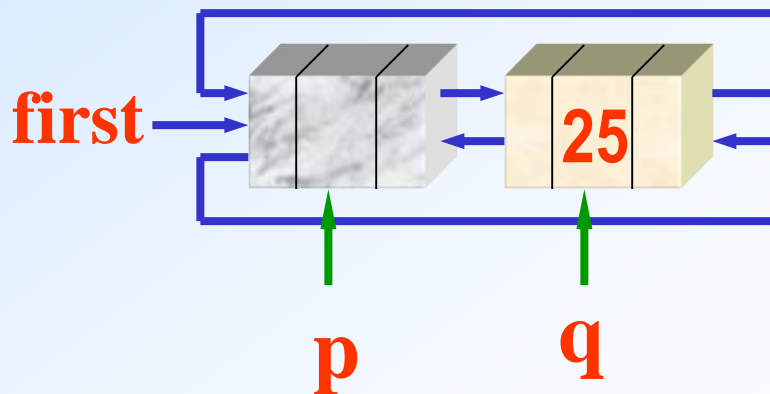
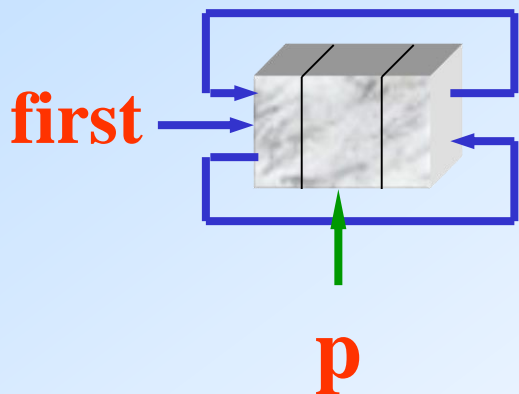
在结点 *p 后插入25

$q \rightarrow \text{prior} = p;$

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

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

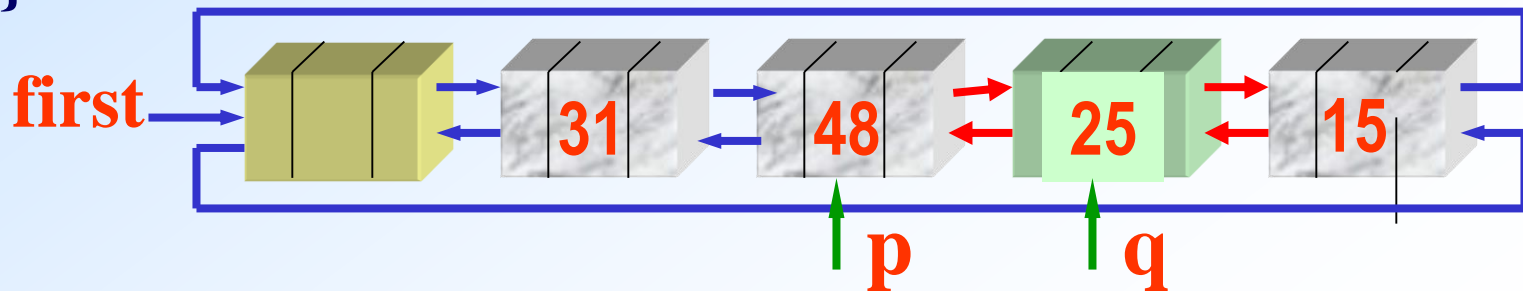
$q \rightarrow \text{next} \rightarrow \text{prior} = q;$



```

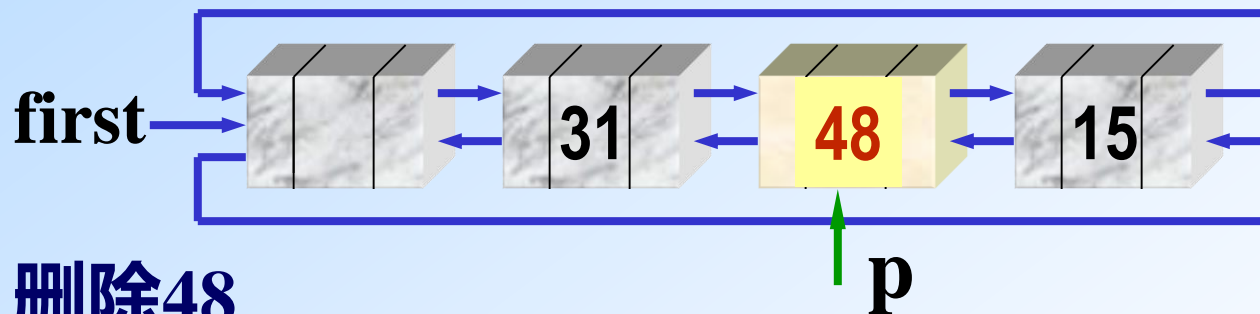
int Insert ( DbList first, int i, Elemtype x ) {
    DbNode * p = Locate ( first, i-1 );
    //指针定位于插入位置
    if ( p == first && i != 1) return 0;
    DbNode * q = ( DbNode * ) malloc
        ( sizeof ( DbNode ) ); //分配结点
    q->data = x;
    q->prior = p;
    q->next = p->next;
    p->next = q;
    q->next->prior = q;
    return 1;
}

```

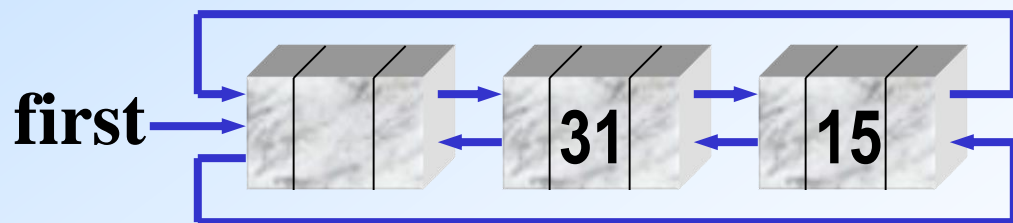


双向循环链表的删除

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



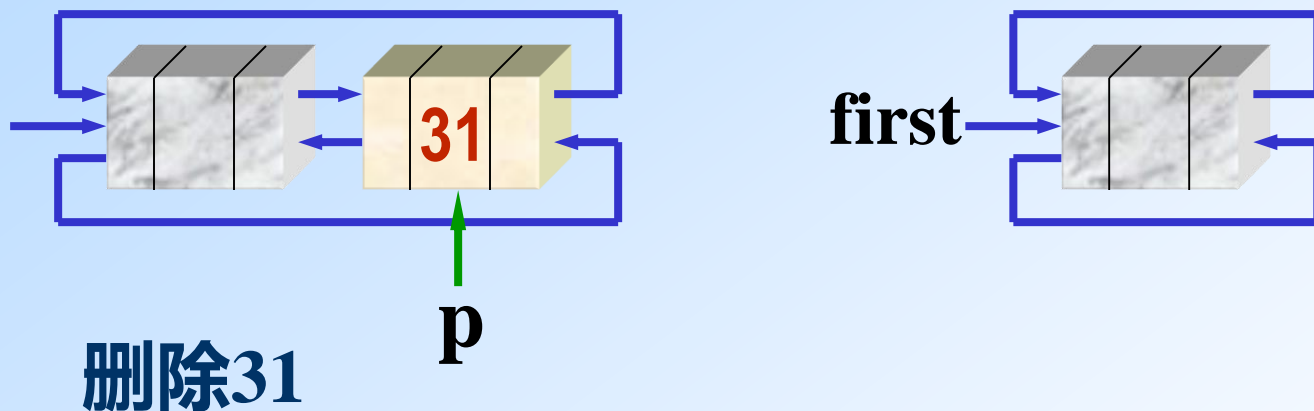
删除48



(非空表)



双向循环链表的删除



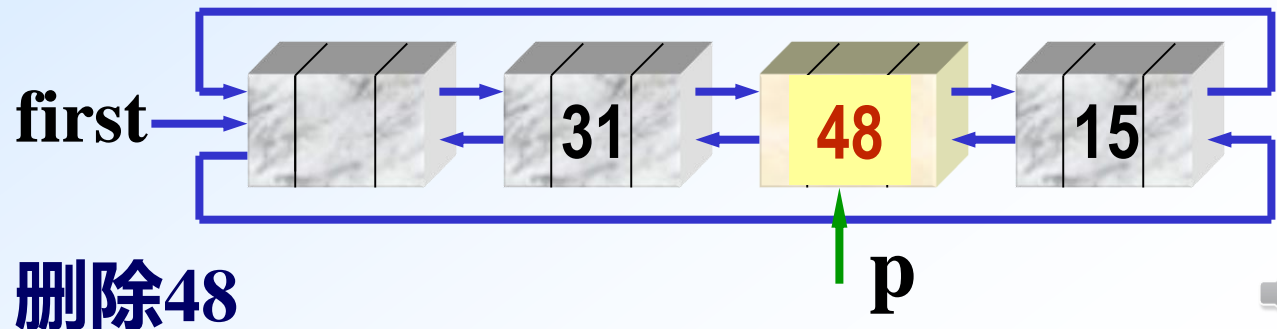
$p \rightarrow next \rightarrow prior = p \rightarrow prior;$
 $p \rightarrow prior \rightarrow next = p \rightarrow next;$



```

int Remove ( DbList first, int i ) {
    DbNode * p = Locate ( first, i );
    //指针定位于删除结点位置
    if ( p == first ) return 0;
    p->next->prior = p->prior;
    p->prior->next = p->next;
    //删除结点 p
    free ( p );      //释放
    return 1;
}

```



- 在linux内核中，有大量的数据结构需要用到双循环链表，例如进程、文件、模块、页面等。



顺序表与链表的比较

基于空间的比较

- 存储分配的方式
 - ◆ 顺序表结点的存储空间是初始化分配的
 - ◆ 链表结点的存储空间是动态分配的
- 存储密度 = 结点数据本身所占的存储量/结点结构所占的存储总量
 - ◆ 顺序表的存储密度 = 1
 - ◆ 链表的存储密度 < 1



顺序表与链表的比较

基于时间的比较

- 存取方式
 - ◆ 顺序表可以随机存取，也可以顺序存取
 - ◆ 链表是顺序存取的
- 插入/删除时移动元素个数
 - ◆ 顺序表平均需要移动近一半元素
 - ◆ 链表不需要移动元素，只需要修改指针



STL中的线性表实现

(1) 向量类vector

- ① 在头文件<vector>中定义了模板类vector(向量类), 该类是一个容器。它实现的是顺序存储的线性表。
- ② 其中可以用v[i]来随机访问第i个元素。
- ③ 在该模板类支持顺序线性表的基本操作。

(2) 链表类list

- ① 在头文件<list>中定义了模板类list(双向链表类), 该类是一个容器, 它实现的是双向链表存储的线性表。
- ② 其中的元素用迭代器类iterator来访问。
- ③ 该模板类支持双向链表的基本操作。



❖ 多项式及其相加

- 在多项式的链表表示中每个结点增加了一个数据成员 `next`，作为链接指针。

data \equiv *Term*

<i>coef</i>	<i>exp</i>	<i>link</i>
-------------	------------	-------------

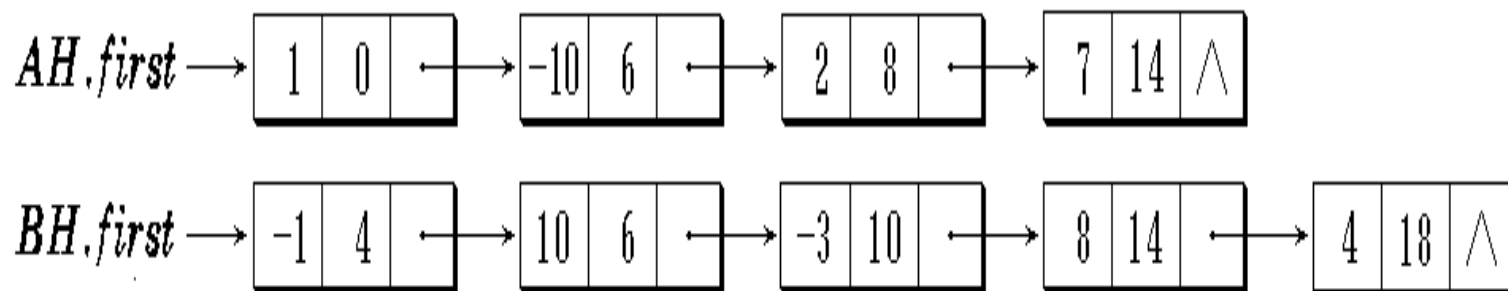
- 优点是：
 - 多项式的项数可以动态地增长，不存在存储溢出问题。
 - 插入、删除方便，不移动元素。



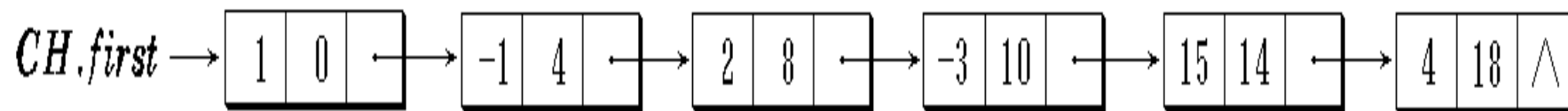
多项式链表的相加

$$AH = 1 - 10x^6 + 2x^8 + 7x^{14}$$

$$BH = -x^4 + 10x^6 - 3x^{10} + 8x^{14} + 4x^{18}$$



(a) 两个相加的多项式



(b) 相加结果的多项式



思考题 1:

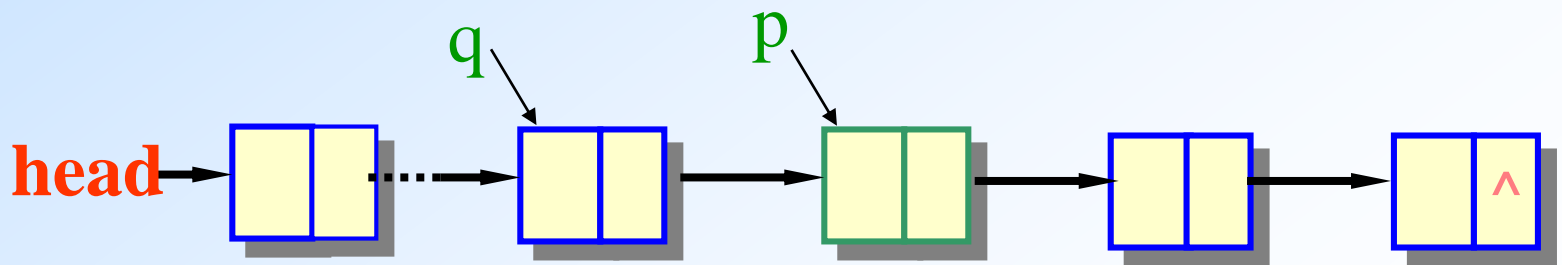
有一个单链表L(至少有一个结点), 其表头结点指针为head, 编写一个函数将L逆置, 即最后一个结点变成第1个结点, 原来倒数第二个结点变成第二个结点.....如此等等。



```

void invert(listnode *head)
{
    listnode *p,*q;
    p=head->next;
    head->next=NULL;
    while(p!=NULL) /*没有后继时停止*/
    {
        q=p;
        p=p->next;
        q->next=head->next;
        head->next=q;
    }
}

```



思考题2：已知带头结点的单链表, list为表头指针，在不改变链表的前提下，请设计一个尽可能高效的算法，查找链表中倒数第K个位置上的结点。



- `int findelem(LNode*head,int k)`
- `{ p=head; q=head->next; i=1;`
- `while(q!=NULL)`
- `{ q=q->next;`
- `++i;`
- `if (i>k) p=p->next;`
- `}`
- `if (p==head) return 0;`
- `else {cout<<p->data;`
- `return 1;}`
- `}`



- 思考题3： 已知带头结点的单链表, L为表头指针。试设计一个高效算法，删除单链表中所有重复的结点（对于多个重复结点，只保留第一个）。

