

第2章 线性表

线性结构是最常用、最简单的一种数据结构。而线性表是一种典型的线性结构。其基本特点是线性表中的数据元素是有序且是有限的。在这种结构中：

- ① 存在一个唯一的被称为“第一个”的数据元素；
- ② 存在一个唯一的被称为“最后一个”的数据元素；
- ③ 除第一个元素外，每个元素均有唯一的一个直接前驱；
- ④ 除最后一个元素外，每个元素均有唯一的一个直接后继。

2.1 线性表的逻辑结构

2.1.1 线性表的定义

线性表(**Linear List**)：是由 $n(n \geq 0)$ 个数据元素(结点) a_1, a_2, \dots, a_n 组成的有限序列。该序列中的所有结点具有**相同的数据类型**。其中数据元素的个数 n 称为线性表的**长度**。

当 $n=0$ 时，称为空表。

当 $n>0$ 时，将非空的线性表记作： (a_1, a_2, \dots, a_n)

2.1.1 线性表的定义

a_1 称为线性表的第一个(首)结点, a_n 称为线性表的最后一个(尾)结点。

a_1, a_2, \dots, a_{i-1} 都是 a_i ($2 \leq i \leq n$)的前驱, 其中 a_{i-1} 是 a_i 的直接前驱;

$a_{i+1}, a_{i+2}, \dots, a_n$ 都是 a_i ($1 \leq i \leq n-1$)的后继, 其中 a_{i+1} 是 a_i 的直接后继。

2.1.2 线性表的逻辑结构

线性表中的数据元素 a_i 所代表的具体含义随具体应用的不同而不同，在线性表的定义中，只不过是一个抽象的表示符号。

◆ 线性表中的**结点**可以是**单值元素**(每个元素只有一个数据项)。

例1: 26个英文字母组成的字母表: (A, B, C, ..., Z)

例2：某小学从2010年到2015年各种型号的计算机拥有量的变化情况：(6, 17, 28, 50, 92, 188)

例3：一副扑克的点数 (2, 3, 4, ..., J, Q, K, A)

◆ 线性表中的结点可以是记录型元素，每个元素含有多个数据项，每个项称为结点的一个域。每个元素有一个可以唯一标识每个结点的数据项组，称为关键字。

例4： 某校2022级同学的基本情况：

{('2022414101', '张里户', '男',
06/24/2003), ('2022414102', '张化司',
'男', 08/12/2002) ..., ('2022414102', '李
利辣', '女', 08/12/2002) }

◆ 若线性表中的结点是**按值**(或按关键字值)由小到大(或由大到小)**排列**的，称线性表是**有序的**。

◆ 线性表是一种相当灵活的数据结构，其长度可根据需要增长或缩短。

◆ 对线性表的数据元素可以访问、插入和删除。

2.1.3 线性表的抽象数据类型定义

ADT List{

数据对象: $D = \{ a_i \mid a_i \in \text{ElemSet}, i=1,2,\dots,n, n \geq 0 \}$

数据关系: $R = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i=2,3,\dots,n \}$

基本操作:

InitList(&L)

操作结果: 构造一个空的线性表L;

ListLength(L)

初始条件：线性表L已存在；

操作结果：若L为空表，则返回**TRUE**，否则返回**FALSE**；

....

GetElem(L, i, &e)

初始条件：线性表L已存在， $1 \leq i \leq \text{ListLength}(L)$ ；

操作结果：用e返回L中第i个数据元素的值；

ListInsert (L, i, &e)

初始条件：线性表L已存在， $1 \leq i \leq \text{ListLength}(L)$ ；

操作结果：在线性表L中的第i个位置插入元素e；

...

} ADT List

2.2 线性表的顺序存储

2.2.1 线性表的顺序存储结构

顺序存储：把线性表的结点**按逻辑顺序依次存放**在一组地址连续的**存储单元**里。用这种方法存储的线性表简称**顺序表**。

顺序存储的线性表的特点：

- ◆ 线性表的逻辑顺序与物理顺序一致；
- ◆ 数据元素之间的关系是以元素在计算机内“**物理位置相邻**”来体现。

设有非空的线性表： (a_1, a_2, \dots, a_n) 。顺序存储如图2-1所示。

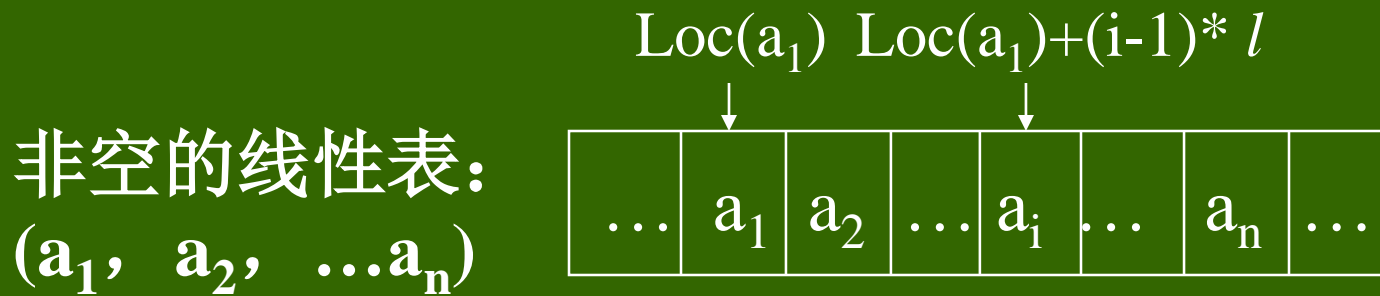


图2-1 线性表的顺序存储表示

在具体的机器环境下： 设线性表的每个元素需占用 l 个存储单元，以所占的第一个单元的存储地址作为数据元素的存储位置。则线性表中第 $i+1$ 个数据元素的存储位置 $LOC(a_{i+1})$ 和第 i 个数据元素的存储位置 $LOC(a_i)$ 之间满足下列关系：

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

线性表的第 i 个数据元素 a_i 的存储位置为：

$$LOC(a_i) = LOC(a_1) + (i-1) * l$$

在高级语言(如C语言)环境下：数组具有随机存取的特性，因此，借助数组来描述顺序表。除了用数组来存储线性表的元素之外，顺序表还应该有表示线性表的长度属性，所以用结构类型来定义顺序表类型。

```
#define OK 1
#define ERROR -1
#define MAX_SIZE 100
typedef int Status ;
typedef int ElemType ;
typedef struct sqlist
{   ElemType Elem_array[MAX_SIZE] ;
    int   length ;
} SqList ;
```

2.2.2 顺序表的基本操作

顺序存储结构中，很容易实现线性表的一些操作：初始化、赋值、查找、修改、插入、删除、求长度等。以下将对几种主要的操作进行讨论。

1 顺序线性表初始化

```
Status Init_SqList( SqList *L )
```

```
{ L->elem_array=( ElemType  
* )malloc(MAX_SIZE*sizeof( ElemType ) ) ;  
    if ( !L -> elem_array ) return ERROR ;  
    else { L->length= 0 ;    return OK ; }  
}
```

2 顺序线性表的插入

在线性表 $L = (a_1, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$ 中的第 i ($1 \leq i \leq n$) 个位置上插入一个新结点 e , 使其成为线性表:

$$L = (a_1, \dots, a_{i-1}, e, a_i, a_{i+1}, \dots, a_n)$$

实现步骤

- (1) 将线性表 L 中的第 i 个至第 n 个结点后移一个位置。
- (2) 将结点 e 插入到结点 a_{i-1} 之后。
- (3) 线性表长度加1。

算法描述

```
Status Insert_SqList(SqList *L, int i , ElemType e)
{
    int j ;
    if ( i<0||i>L->length-1) return ERROR ;
    if (L->length>=MAX_SIZE)
        { printf(“线性表溢出!\n”); return ERROR ; }
    for ( j=L->length-1; j>=i-1; --j )
        L->Elem_array[j+1]=L->Elem_array[j];
        /* i-1位置以后的所有结点后移 */
    L->Elem_array[i-1]=e; /* 在i-1位置插入结点 */
    L->length++ ;
    return OK ;
}
```

时间复杂度分析

在线性表L中的第i个元素之前插入新结点，其时间主要耗费在表中结点的移动操作上，因此，可用结点的移动来估计算法的时间复杂度。

设在线性表L中的第i个元素之前插入结点的概率为 P_i ，不失一般性，设各个位置插入是等概率，则 $P_i=1/(n+1)$ ，而插入时移动结点的次数为 $n-i+1$ 。

总的平均移动次数： $E_{\text{insert}}=\sum p_i*(n-i+1) \quad (1 \leq i \leq n)$

$\therefore E_{\text{insert}}=n/2$ 。

即在顺序表上做插入运算，平均要移动表上一半结点。当表长n较大时，算法的效率相当低。因此算法的平均时间复杂度为 $O(n)$ 。

3 顺序线性表的删除

在线性表 $L=(a_1, \dots a_{i-1}, a_i, a_{i+1}, \dots, a_n)$ 中删除结点 $a_i (1 \leq i \leq n)$, 使其成为线性表:

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

实现步骤

- (1) 将线性表L中的第i+1个至第n个结点依此向前移动一个位置。
- (2) 线性表长度减1。

算法描述

```
ElemType Delete_SqList(SqList *L, int i)
```

```
{ int k; ElemType x;
```



```
if (L->length==0)
    { printf(“线性表L为空!\n”); return ERROR; }
else if ( i<1||i>L->length )
    { printf(“要删除的数据元素不存在!\n”);
      return ERROR ; }
else { x=L->Elem_array[i-1]; /*保存结点的值*/
      for ( k=i ; k<L->length ; k++)
          L->Elem_array[k-1]=L->Elem_array[k];
          /* i位置以后的所有结点前移 */
      L->length--; return (x);
    }
}
```

时间复杂度分析

删除线性表L中的第i个元素，其时间主要耗费在表中结点的移动操作上，因此，可用结点的移动来估计算法的时间复杂度。

设在线性表L中删除第i个元素的概率为 P_i ，不失一般性，设删除各个位置是等概率，则 $P_i=1/n$ ，而删除时移动结点的次数为 $n-i$ 。

则总的平均移动次数： $E_{\text{delete}}=\sum p_i*(n-i) \quad (1 \leq i \leq n)$

$$\therefore E_{\text{delete}}=(n-1)/2。$$

即在顺序表上做删除运算，平均要移动表上一半结点。当表长n较大时，算法的效率相当低。因此算法的平均时间复杂度为 $O(n)$ 。

4 顺序线性表的查找定位删除

在线性表 $L = (a_1, a_2, \dots, a_n)$ 中删除值为 x 的第一个结点。

实现步骤

- (1) 在线性表 L 查找值为 x 的第一个数据元素。
- (2) 将从找到的位置至最后一个结点依次向前移动一个位置。
- (3) 线性表长度减1。

算法描述

Status Locate_Delete_SqList(SqList *L, ElemType x)

/* 删除线性表 L 中值为 x 的第一个结点 */

{ int i=0, k;

```
while (i<L->length)    /*查找值为x的第一个结点*/
{   if (L->Elem_array[i]!=x ) i++ ;
    else
        {   for ( k=i+1; k< L->length; k++)
                L->Elem_array[k-1]=L->Elem_array[k];
            L->length--; break ;
        }
}

if (i>L->length)
{   printf(“要删除的数据元素不存在!\n”) ;
    return ERROR ; }

return OK;

}
```

时间复杂度分析

时间主要耗费在数据元素的比较和移动操作上。

首先，在线性表L中查找值为x的结点是否存在；
其次，若值为x的结点存在，且在线性表L中的位置为i，
则在线性表L中删除第i个元素。

设在线性表L删除数据元素概率为 P_i ，不失一般性，
设各个位置是等概率，则 $P_i=1/n$ 。

◆ 比较的平均次数： $E_{\text{compare}}=\sum p_i*i \quad (1 \leq i \leq n)$

$\therefore E_{\text{compare}}=(n+1)/2$ 。

◆ 删除时平均移动次数： $E_{\text{delete}}=\sum p_i*(n-i) \quad (1 \leq i \leq n)$

$\therefore E_{\text{delete}}=(n-1)/2$ 。 平均时间复杂度： $E_{\text{compare}}+E_{\text{delete}}=n$ ，
即为 $O(n)$

2.3 线性表的链式存储

2.3.1 线性表的链式存储结构

链式存储：用一组任意的存储单元存储线性表中的数据元素。用这种方法存储的线性表简称**线性链表**。

存储链表中结点的一组任意的存储单元可以是连续的，也可以是不连续的，甚至是零散分布在内存中的任意位置上的。

链表中结点的逻辑顺序和物理顺序不一定相同。

为了正确表示结点间的逻辑关系，在存储每个结点值的同时，还必须存储指示其直接后继结点的**地址(或位置)**，称为**指针(pointer)或链(link)**，这两部分组成了链表中的结点结构，如图2-2所示。

链表是通过每个结点的**指针域**将线性表的 n 个结点按其**逻辑次序链接在一起**的。

每一个结只包含一个指针域的链表，称为**单链表**。

为操作方便，总是在链表的第一个结点之前附设一个**头结点(头指针) head**指向第一个结点。头结点的数据域可以不存储任何信息(或链表长度等信息)。



data：数据域，存放结点的值。**next**：指针域，存放结点的直接后继的地址。

图2-2 链表结点结构

单链表是由表头唯一确定，因此单链表可以用头指针的名字来命名。

例1、线性表L=(bat, cat, eat, fat, hat)

其带头结点的单链表的逻辑状态和物理存储方式如图2-3所示。

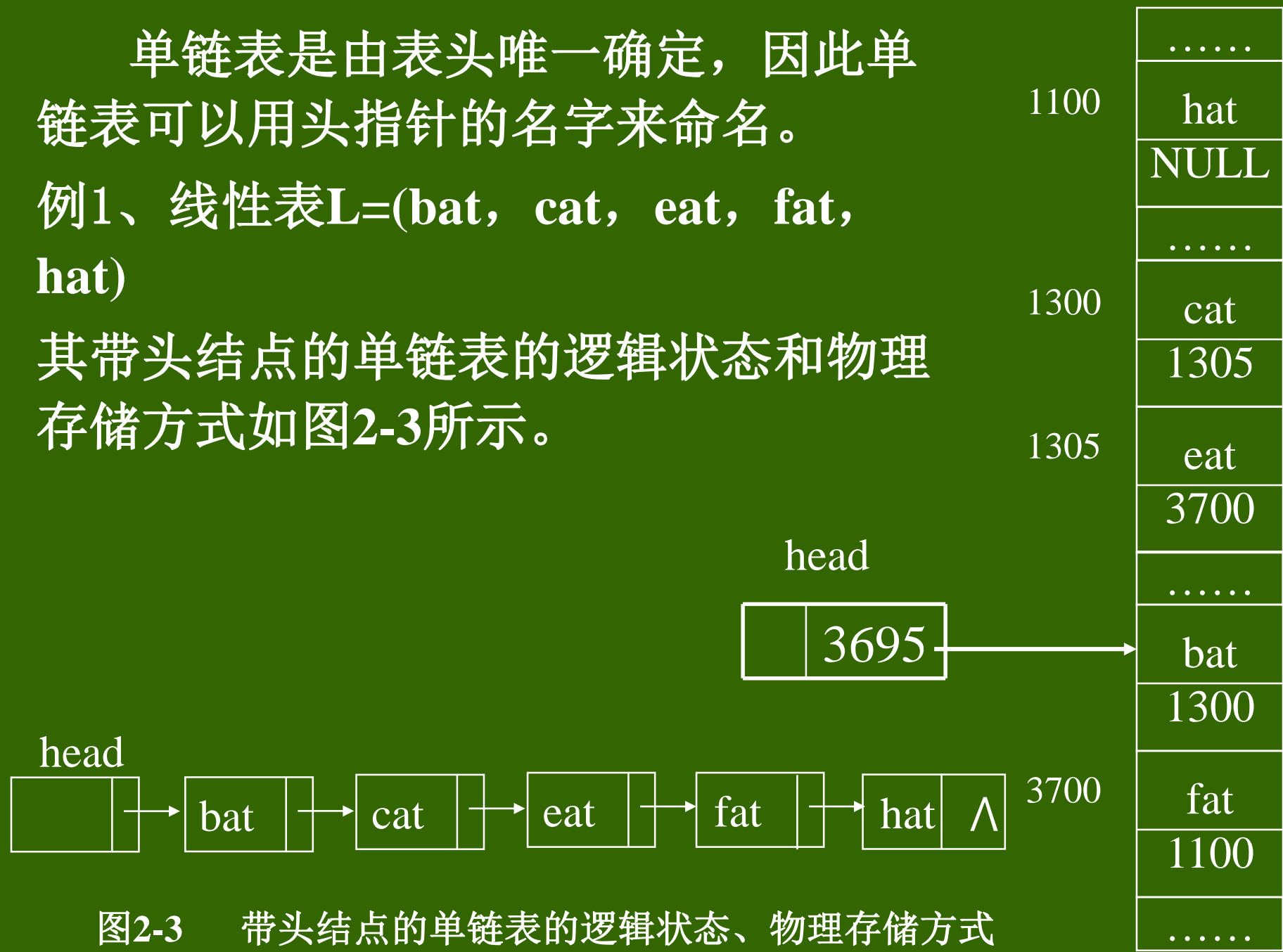


图2-3 带头结点的单链表的逻辑状态、物理存储方式

1 结点的描述与实现

C语言中用带指针的结构体类型来描述

```
typedef struct Lnode
```

```
{ ElemType data;    /*数据域，保存结点的值 */
```

```
    struct Lnode *next;    /*指针域*/
```

```
}LNode;    /*结点的类型 */
```

2 结点的实现

结点是通过动态分配和释放来的实现，即需要时分配，不需要时释放。实现时是分别使用C语言提供的标准函数：`malloc()`，`realloc()`，`sizeof()`，`free()`。

动态分配 `p=(LNode*)malloc(sizeof(LNode));`

函数`malloc`分配了一个类型为`LNode`的结点变量的空间，并将其首地址放入指针变量`p`中。

动态释放 `free(p);`

系统回收由指针变量`p`所指向的内存区。`P`必须是最近一次调用`malloc`函数时的返回值。

3 最常用的基本操作及其示意图

(1) 结点的赋值

```
LNode *p;
```

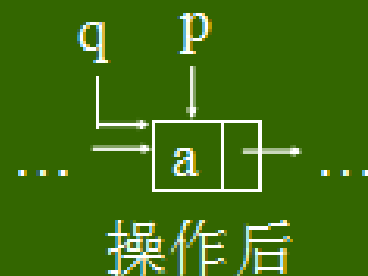
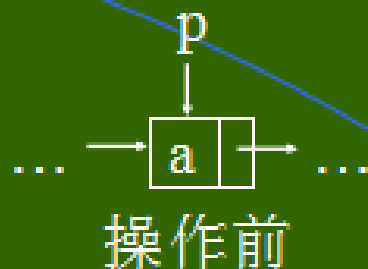
```
p=(LNode*)malloc(sizeof(LNode));
```

```
p->data=20; p->next=NULL;
```

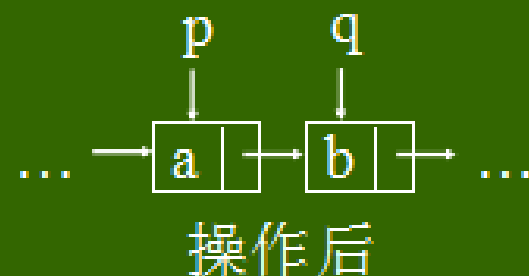
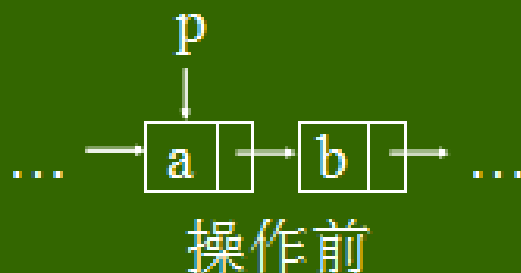


(2) 常见的指针操作

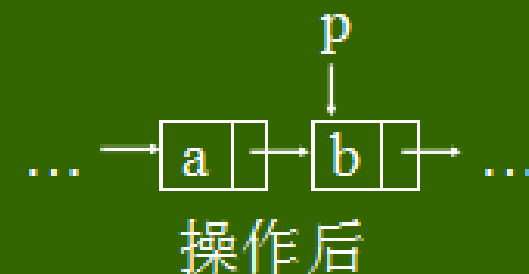
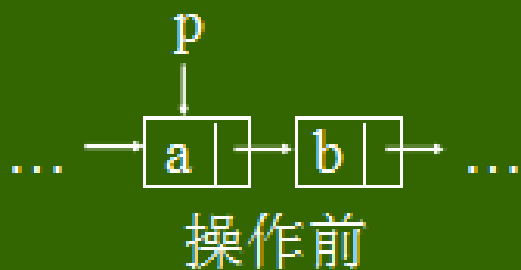
① $q=p;$



② $q=p->next;$



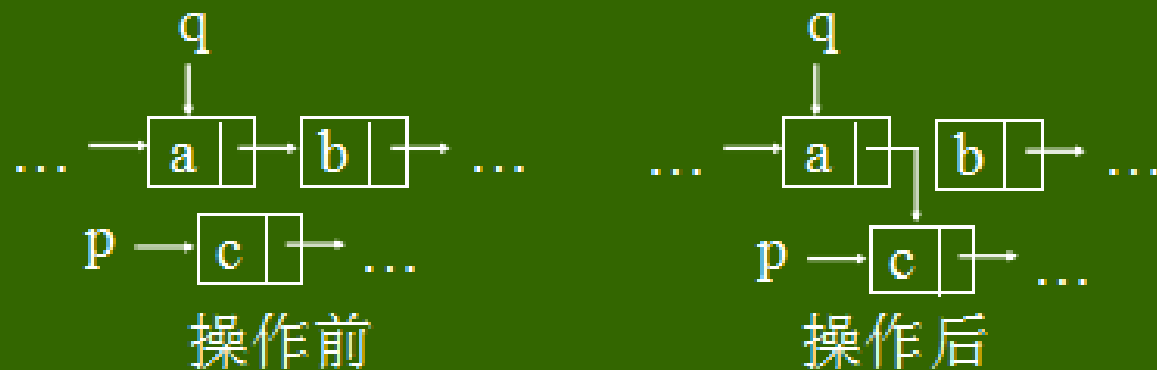
③ $p=p->next;$



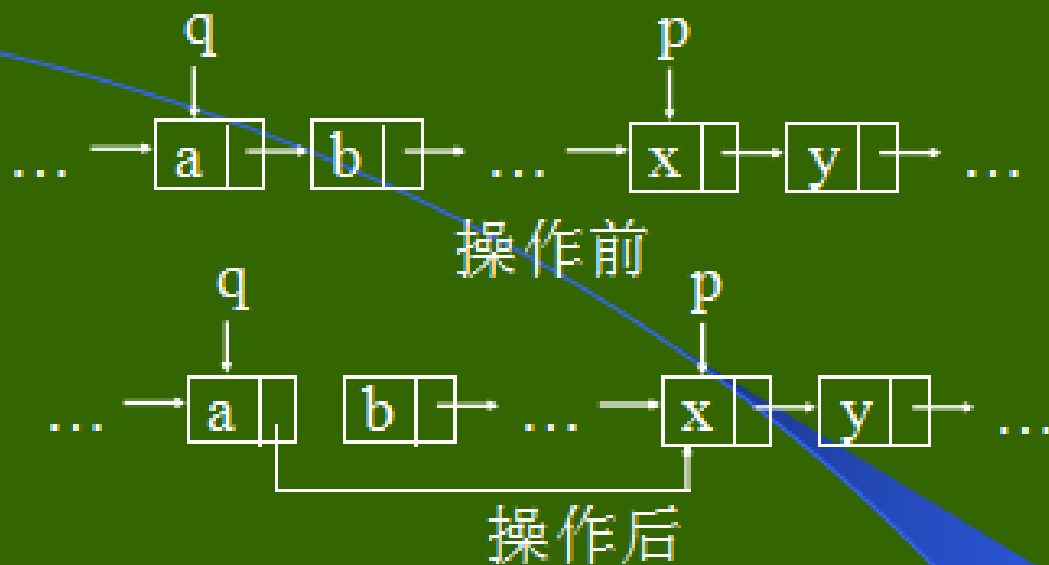
(2) 常见的指针操作

④ $q \rightarrow \text{next} = p$;

(a)

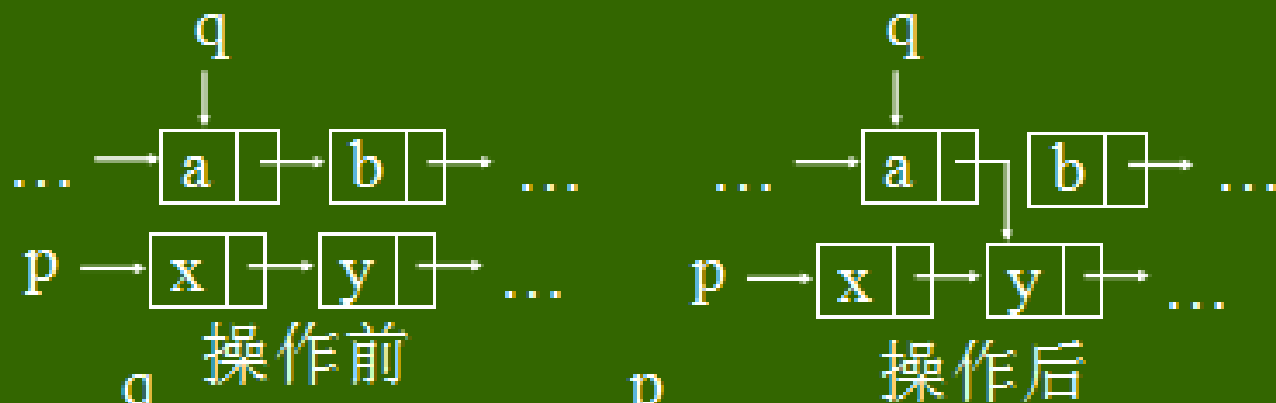


(b)

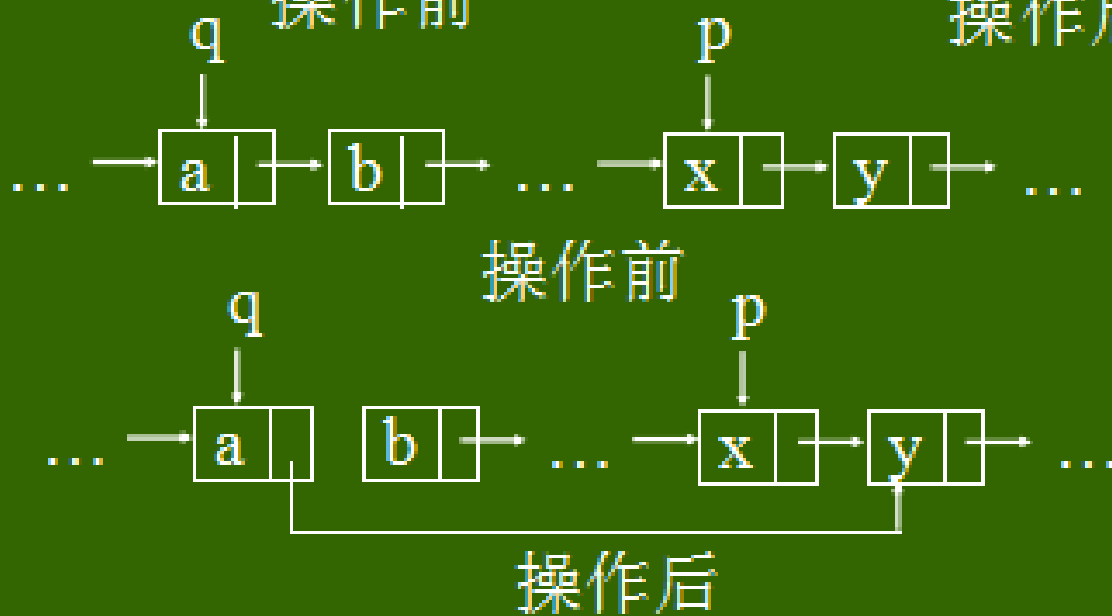


⑤ $q \rightarrow \text{next} = p \rightarrow \text{next}$;

(a)



(b)



2.3.2 单线性链式的基本操作

1 建立单链表

假设线性表中结点的数据类型是整型，以32767作为结束标志。动态地建立单链表的常用方法有如下两种：头插入法，尾插入法。

(1) 头插入法建表

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，然后将新结点插入到当前链表的表头上，直到读入结束标志为止。即每次插入的结点都作为链表的第一个结点。

算法描述

LNode *create_LinkList(void)

```
/* 头插入法创建单链表,链表的头结点head作为返回值 */
{   int data ;
    LNode *head, *p;
    head= (LNode *) malloc( sizeof(LNode));
    head->next=NULL;    /* 创建链表的表头结点head */
    while (1)
    {   scanf("%d", &data) ;
        if (data==32767) break ;
        p= (LNode *)malloc(sizeof(LNode));
        p->data=data;    /* 数据域赋值 */
```

```
        p->next=head->next ; head->next=p ;  
        /* 钩链，新创建的结点总是作为第一个结点 */  
    }  
    return (head);  
}
```

(2) 尾插入法建表

头插入法建立链表虽然算法简单，但生成的链表中结点的次序和输入的顺序相反。若希望二者次序一致，可采用尾插法建表。该方法是将新结点插入到当前链表的表尾，使其成为当前链表的尾结点。

算法描述

LNode *create_LinkList(void)

```
/* 尾插入法创建单链表,链表的头结点head作为返回值 */
{  int data ;
    LNode *head, *p, *q;
    head=p=(LNode *)malloc(sizeof(LNode));
    p->next=NULL;      /* 创建单链表的表头结点head */
    while (1)
    {   scanf("%d",& data);
        if (data==32767) break ;
        q= (LNode *)malloc(sizeof(LNode));
        q->data=data;    /* 数据域赋值 */
        q->next=p->next; p->next=q; p=q ;
    }
```

/*钩链，新创建的结点总是作为最后一个结点*/

}

return (head);

}

无论是哪种插入方法，如果要插入建立的单线性链表的结点是 n 个，算法的时间复杂度均为 $O(n)$ 。

对于单链表，无论是哪种操作，只要涉及到钩链(或重新钩链)，如果没有明确给出直接后继，钩链(或重新钩链)的次序必须是“先右后左”。

2 单链表的查找

(1) **按序号查找** 取单链表中的第*i*个元素。

对于单链表，不能象顺序表中那样直接按序号*i*访问结点，而只能从链表的头结点出发，沿链域next逐个结点往下搜索，直到搜索到第*i*个结点为止。因此，**链表不是随机存取结构**。

设单链表的长度为*n*，要查找表中第*i*个结点，仅当 $1 \leq i \leq n$ 时，*i*的值是合法的。

算法描述

```
ElemType  Get_Elem(LNode *L ,  int  i)
{  int j ;  LNode *p;
    p=L->next; j=1;    /* 使p指向第一个结点 */
    while (p!=NULL && j<i)
        {  p=p->next; j++; }      /* 移动指针p , j计数 */
    if (j!=i) return(-32768) ;
    else    return(p->data);
        /*  p为NULL 表示i太大; j>i表示i为0 */
}
```

移动指针p的频度:

$i < 1$ 时: 0次; $i \in [1, n]$: $i-1$ 次; $i > n$: n 次。

\therefore 时间复杂度: $O(n)$ 。

(2) 按值查找

按值查找是在链表中，查找是否有结点值等于给定值key的结点？若有，则返回首次找到的值为key的结点的存储位置；否则返回NULL。查找时从开始结点出发，沿链表逐个将结点的值和给定值key作比较。

算法描述

```
LNode *Locate_Node(LNode *L, int key)
```

```
/* 在以L为头结点的单链表中查找值为key的第一个结点 */
```

```
{  LNode *p=L->next;
```

```
    while ( p!=NULL&& p->data!=key)  p=p->next;
```

```
    if (p->data==key)  return p;
```

```
    else
```

```
        {  printf(“所要查找的结点不存在!!\n”);
```

```
            retutn(NULL);
```

```
        }
```

```
    }
```

算法的执行与形参key有关，平均时间复杂度为 $O(n)$ 。

3 单链表的插入

插入运算是将值为 e 的新结点插入到表的第 i 个结点的位置上，即插入到 a_{i-1} 与 a_i 之间。因此，必须首先找到 a_{i-1} 所在的结点 p ，然后生成一个数据域为 e 的新结点 q ， q 结点作为 p 的直接后继结点。

算法描述

```
void Insert_LNode(LNode *L, int i, ElemType e)
```

```
/* 在以L为头结点的单链表的第i个位置插入值为e的结点 */
```

```
{ int j=0; LNode *p, *q;
```

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

```
  while ( p!=NULL&& j<i-1)
```

```
    { p=p->next; j++; }
```

```
if (j!=i-1)    printf(“i太大或i为0!!\n ”);
else
    { q=(LNode *)malloc(sizeof(LNode));
      q->data=e;  q->next=p->next;
      p->next=q;
    }
}
```

设链表的长度为 n ，合法的插入位置是 $1 \leq i \leq n$ 。算法的时间主要耗费移动指针 p 上，故时间复杂度亦为 $O(n)$ 。

4 单链表的删除

(1) 按序号删除

删除单链表中的第 i 个结点。

为了删除第 i 个结点 a_i ，必须找到结点的存储地址。该存储地址是在其直接前趋结点 a_{i-1} 的`next`域中，因此，必须首先找到 a_{i-1} 的存储位置 p ，然后令 $p \rightarrow \text{next}$ 指向 a_i 的直接后继结点，即把 a_i 从链上摘下。最后释放结点 a_i 的空间，将其归还给“**存储池**”。

设单链表长度为 n ，则删去第 i 个结点仅当 $1 \leq i \leq n$ 时是合法的。则当 $i = n + 1$ 时，虽然被删结点不存在，但其前趋结点却存在，是终端结点。故判断条件之一是 $p \rightarrow \text{next} \neq \text{NULL}$ 。显然此算法的时间复杂度也是 $O(n)$ 。

算法描述

```
void Delete_LinkList(LNode *L,  int i)
    /* 删除以L为头结点的单链表中的第i个结点  */
{  int j=1; LNode *p, *q;
    p=L; q=L->next;
    while ( p->next!=NULL&& j<i)
        { p=q; q=q->next; j++; }
    if (j!=i)    printf(“i太大或i为0!!\n ”);
    else
        { p->next=q->next;  free(q);  }
}
```

(2) 按值删除

删除单链表中值为key的第一个结点。

与按值查找相类似，首先要查找值为key的结点是否存在？若存在，则删除；否则返回NULL。

算法描述

```
void Delete_LinkList(LNode *L, int key)
```

```
/* 删除以L为头结点的单链表中值为key的第一个结点 */
```

```
{   LNode *p=L, *q=L->next;
```

```
    while ( q!=NULL&& q->data!=key)
```

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

```
    if (q->data==key)
```

```
        { p->next=q->next; free(q); }
```

```
    else
```

```
        printf(“所要删除的结点不存在!!\n”);
```

```
}
```

算法的执行与形参key有关，平均时间复杂度为 $O(n)$ 。

从上面的讨论可以看出，链表上实现插入和删除运算，无需移动结点，仅需修改指针。解决了顺序表的插入或删除操作需要移动大量元素的问题。

变形之一：

删除单链表中值为key的所有结点。

与按值查找相类似，但比前面的算法更简单。

基本思想：从单链表的第一个结点开始，对每个结点进行检查，若结点的值为key，则删除之，然后检查下一个结点，直到所有的结点都检查。

算法描述

```
void Delete_LinkList_Node(LNode *L, int key)
/* 删除以L为头结点的单链表中值为key的第一个结点 */
{   LNode *p=L, *q=L->next;
    while ( q!=NULL)
        { if (q->data==key)
            { p->next=q->next; free(q); q=p->next; }
          else
            { p=q; q=q->next; }
        }
}
```

变形之二:

删除单链表中所有值重复的结点，使得所有结点的值都不相同。

与按值查找相类似，但比前面的算法更复杂。

基本思想：从单链表的第一个结点开始，对每个结点进行检查：检查链表中该结点的所有后继结点，只要有值和该结点的值相同，则删除之；然后检查下一个结点，直到所有的结点都检查。

算法描述

```
void Delete_Node_value(LNode *L)
```

```
    /* 删除以L为头结点的单链表中所有值相同的结点 */
```

```
    {   LNode *p=L->next, *q, *ptr;
```

```
        while ( p!=NULL) /* 检查链表中所有结点 */
```

```
        {   *q=p, *ptr=p->next;
```

```
            /* 检查结点p的所有后继结点ptr */
```

```
            while (ptr!=NULL)
```

```
                {   if (ptr->data==p->data)
```

```
                    {   q->next=ptr->next; free(ptr);
```

```
                        ptr=q->next;   }
```

```
                else {   q=ptr; ptr=ptr->next;   }
```

```
            }
```



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

```
}
```

```
}
```

5 单链表的合并

设有两个有序的单链表，它们的头指针分别是La、Lb，将它们合并为以Lc为头指针的有序链表。合并前的示意图如图2-4所示。

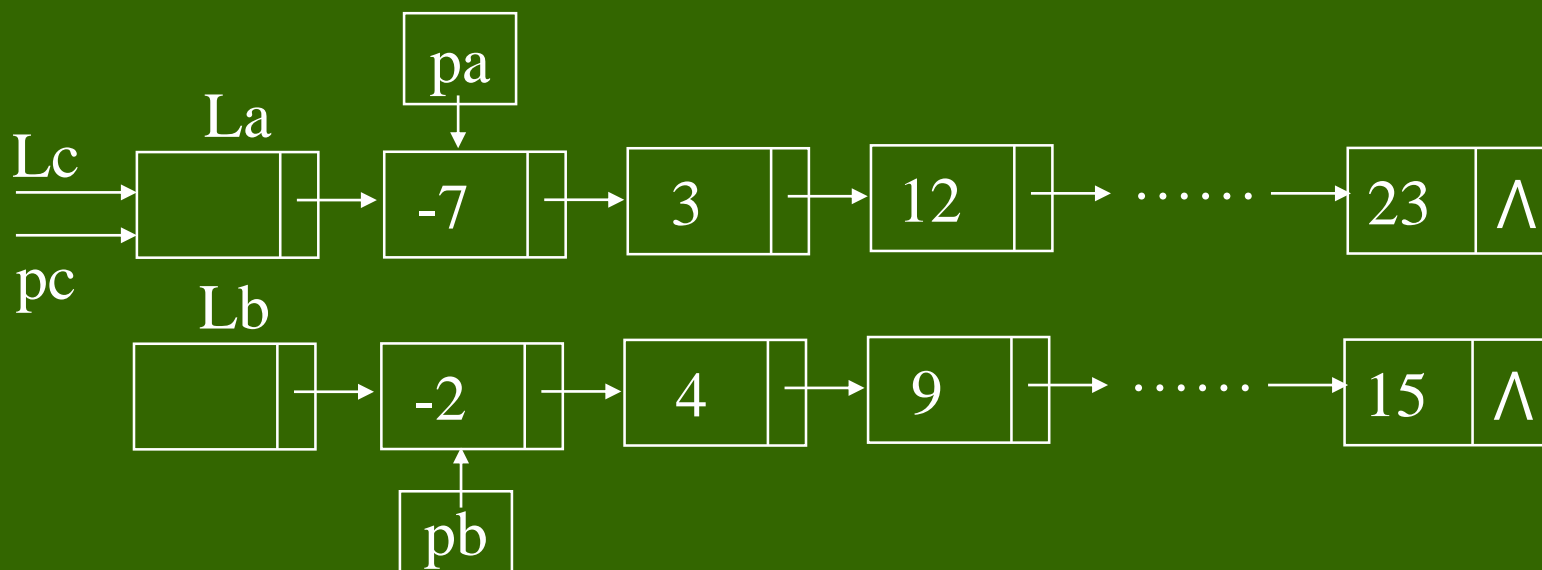


图2-4 两个有序的单链表La，Lb的初始状态

合并了值为-7， -2的结点后示意图如图2-5所示。

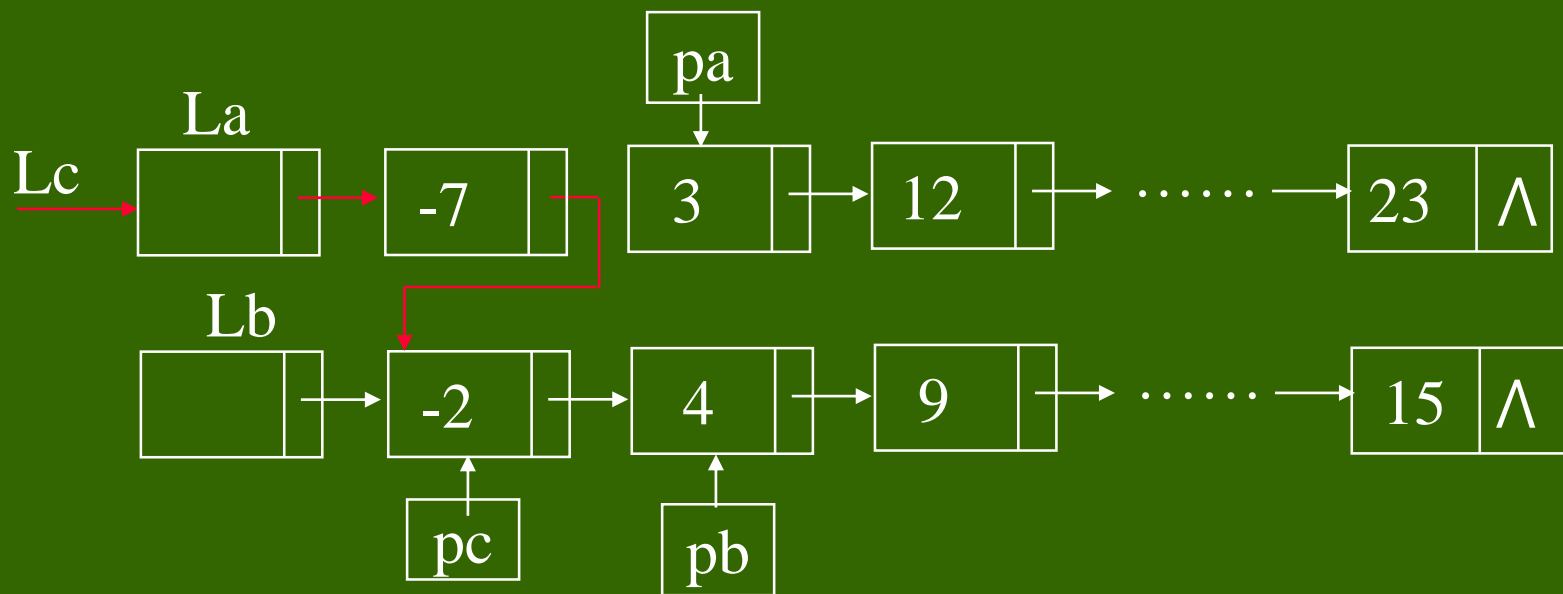


图2-5 合并了值为-7， -2的结点后的状态

算法说明

算法中pa， pb分别是待考察的两个链表的当前结点， pc是合并过程中合并的链表的最后一个结点。

算法描述

```
LNode *Merge_LinkList(LNode *La, LNode *Lb)
/* 合并以La,Lb为头结点的两个有序单链表 */
{  LNode *Lc, *pa , *pb , *pc, *ptr ;
   Lc=La ; pc=La ;  pa=La->next ; pb=Lb->next ;
   while (pa!=NULL && pb!=NULL)
       { if (pa->data<pb->data)
           { pc->next=pa ; pc=pa ; pa=pa->next ; }
         /* 将pa所指的结点合并, pa指向下一个结点 */
         if (pa->data>pb->data)
             { pc->next=pb ; pc=pb ; pb=pb->next ; }
         /* 将pa所指的结点合并, pa指向下一个结点 */
```

```
    if (pa->data==pb->data)
        { pc->next=pa ; pc=pa ; pa=pa->next ;
          ptr=pb ; pb=pb->next ; free(ptr) ; }
    /* 将pa所指的结点合并，pb所指结点删除 */
}

if (pa!=NULL) pc->next=pa ;
else pc->next=pb ; /*将剩余的结点链上*/
free(Lb) ;
return(Lc) ;
}
```

算法分析

若La，Lb两个链表的长度分别是m，n，则链表合并的时间复杂度为 $O(m+n)$ 。

2.3.3 循环链表

循环链表(Circular Linked List)：是一种头尾相接的链表。其特点是最后一个结点的指针域指向链表的头结点，整个链表的指针域链接成一个环。

从循环链表的任意一个结点出发都可以找到链表中的其它结点，使得表处理更加方便灵活。

图2-6是带头结点的单循环链表的示意图。

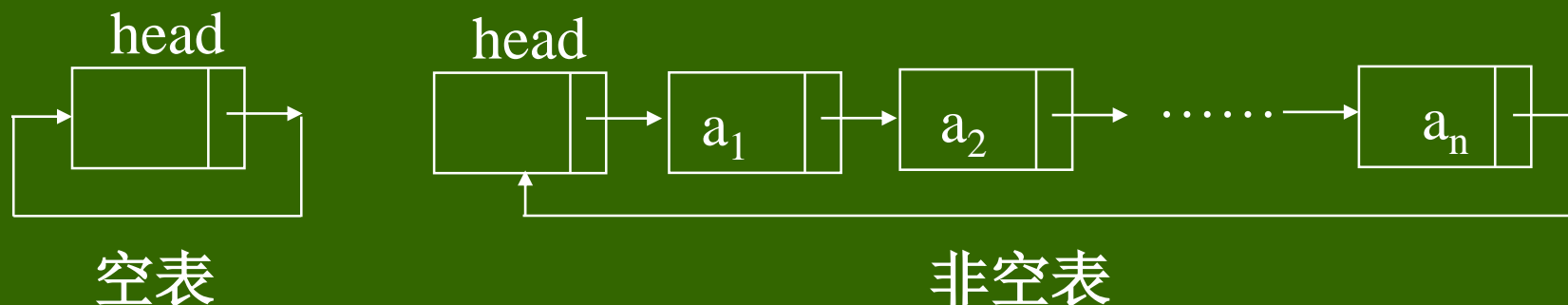


图2-6 单循环链表示意图

循环链表的操作

对于单循环链表，除链表的合并外，其它的操作和单线性链表基本上一致，仅仅需要在单线性链表操作算法基础上作以下简单修改：

- (1) 判断是否是空链表： `head->next==head` ；
- (2) 判断是否是表尾结点： `p->next==head` ；

2.4 双向链表

双向链表(Double Linked List) :指的是构成链表的每个结点中设立两个指针域：一个指向其直接前趋的指针域prior，一个指向其直接后继的指针域next。这样形成的链表中有两个方向不同的链，故称为**双向链表**。

和单链表类似，双向链表一般增加头指针也能使双链表上的某些运算变得方便。

将头结点和尾结点链接起来也能构成循环链表，并称之为双向循环链表。

双向链表是为了克服单链表的单向性的缺陷而引入的。

1 双向链表的结点及其类型定义

双向链表的结点的类型定义如下。其结点形式如图2-7所示，带头结点的双向链表的形式如图2-8所示。

```
typedef struct Dulnode
{
    ElemType data ;
    struct Dulnode *prior , *next ;
}DulNode ;
```



图2-7 双向链表结点形式



空双向链表



非空双向链表

图2-8 带头结点的双向链表形式

双向链表结构具有对称性，设p指向双向链表中的某一结点，则其对称性可用下式描述：

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

结点p的存储位置存放在其直接前趋结点p->prior的直接后继指针域中，同时也存放在其直接后继结点p->next的直接前趋指针域中。

2 双向链表的基本操作

(1) **双向链表的插入** 将值为e的结点插入双向链表中。插入前后链表的变化如图2-9所示。



图2-9 双向链表的插入

① 插入时仅仅指出直接前驱结点，钩链时必须注意先后次序是：“先右后左”。部分语句组如下：

```
S=(DulNode *)malloc(sizeof(DulNode));
```

```
S->data=e;
```

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

```
p->next=S; S->prior=p; /* 钩链次序非常重要 */
```

② 插入时同时指出直接前驱结点p和直接后继结点q，钩链时无须注意先后次序。部分语句组如下：

```
S=(DulNode *)malloc(sizeof(DulNode));
```

```
S->data=e;
```

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

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

(2) 双向链表的结点删除

设要删除的结点为p，删除时可以不引入新的辅助指针变量，可以直接先断链，再释放结点。部分语句组如下：

```
p->prior->next=p->next;  
p->next->prior=p->prior;  
free(p);
```

注意：

与单链表的插入和删除操作不同的是，在双向链表中**插入和删除必须同时修改两个方向上的指针域的指向**。

2.5 一元多项式的表示和相加

1 一元多项式的表示

一元多项式 $p(x)=p_0+p_1x+p_2x^2+ \dots +p_nx^n$ ，由 $n+1$ 个系数唯一确定。则在计算机中可用线性表(p_0 ， p_1 ， p_2 ， \dots ， p_n)表示。既然是线性表，就可以用顺序表和链表来实现。两种不同实现方式的元素类型定义如下：

(1) 顺序存储表示的类型

```
typedef struct
```

```
{ float coef; /*系数部分*/  
  int expn; /*指数部分*/  
} ElemType ;
```

(2) 链式存储表示的类型

```
typedef struct ploy
```

```
{ float coef ; /*系数部分*/  
  int expn ; /*指数部分*/  
  struct ploy *next ;  
} Ploy ;
```

2 一元多项式的相加

不失一般性，设有两个一元多项式：

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

$$Q(x)=q_0+q_1x+q_2x^2+\dots+q_mx^m \quad (m<n)$$

$$R(x)=P(x)+Q(x)$$

$R(x)$ 由线性表 $R((p_0+q_0), (p_1+q_1), (p_2+q_2), \dots, (p_m+q_m), \dots, p_n)$ 唯一表示。

(1) 顺序存储表示的相加

线性表的定义

```
typedef struct
```

```
{ ElemType a[MAX_SIZE] ;
```

```
    int  length ;
```

```
}Sqlist ;
```

用顺序表示的相加非常简单。访问第5项可直接访问：`L.a[4].coef`，`L.a[4].expn`

(2) 链式存储表示的相加

当采用链式存储表示时，根据结点类型定义，凡是系数为0的项不在链表中出现，从而可以大大减少链表的长度。

一元多项式相加的实质是：

- ☞ 指数不同： 是链表的合并。
- ☞ 指数相同： 系数相加，和为0，去掉结点，和不为0，修改结点的系数域。

算法之一：

就在原来两个多项式链表的基础上进行相加，相加后原来两个多项式链表就不存在了。当然再要对原来两个多项式进行其它操作就不允许了。

算法描述

Ploy *add_ploy(ploy *La, ploy *Lb)

/* 将以La , Lb为头指针表示的一元多项式相加 */

{ ploy *Lc , *pc , *pa , *pb , *ptr ; float x ;

Lc=pc=La ; pa=La->next ; pb=Lb->next ;

while (pa!=NULL&&pb!=NULL)

{ if (pa->expn<pb->expn)

{ pc->next=pa ; pc=pa ; pa=pa->next ; }

/* 将pa所指的结点合并, pa指向下一个结点 */

if (pa->expn>pb->expn)

{ pc->next=pb ; pc=pb ; pb=pb->next ; }

/* 将pb所指的结点合并, pb指向下一个结点 */

else

```
{ x=pa->coef+pb->coef ;
```

```
  if (abs(x)<=1.0e-6)
```

```
    /* 如果系数和为0，删除两个结点 */
```

```
    { ptr=pa ; pa=pa->next ; free(ptr) ;
```

```
      ptr=pb ; pb=pb->next ; free(ptr) ; }
```

```
  else /* 如果系数和不为0，修改其中一个结点的系数域，删除另一个结点 */
```

```
    { pc->next=pa ; pa->coef=x ;
```

```
      pc=pa ; pa=pa->next ;
```

```
      ptr=pb ; pb=pb->next ; free(pb) ;
```

```
    }
```

```
}
```

```
}    /* end of while */
```

```
if (pa==NULL) pc->next=pb ;
```

```
else pc->next=pa ;
```

```
return (Lc) ;
```

```
}
```

算法之二:

对两个多项式链表进行相加，生成一个新的相加后的结果多项式链表，原来两个多项式链表依然存在，不发生改变，如果要再对原来两个多项式进行其它操作也不影响。

算法描述

Ploy *add_ploy(ploy *La, ploy *Lb)

/* 将以La , Lb为头指针表示的一元多项式相加, 生成一个新的结果多项式 */

```
{ ploy *Lc , *pc , *pa , *pb , *p ; float x ;  
  Lc=pc=(ploy *)malloc(sizeof(ploy)) ;  
  pa=La->next ; pb=Lb->next ;  
  while (pa!=NULL&&pb!=NULL)  
  { if (pa->expn<pb->expn)  
    { p=(ploy *)malloc(sizeof(ploy)) ;  
      p->coef=pa->coef ; p->expn=pa->expn ;  
      p->next=NULL ;
```

```
        /* 生成一个新的结果结点并赋值 */
        pc->next=p ; pc=p ; pa=pa->next ;
    }

    /* 生成的结点插入到结果链表的最后， pa指向下一个结点 */
    if (pa->expn>pb->expn)
    { p=(ploy *)malloc(sizeof(ploy)) ;
      p->coef=pb->coef ; p->expn=pb->expn ;
      p->next=NULL ;
      /* 生成一个新的结果结点并赋值 */
      pc->next=p ; pc=p ; pb=pb->next ;
    } /* 生成的结点插入到结果链表的最后， pb指向下一个结点 */
```

```
if (pa->expn==pb->expn)
    { x=pa->coef+pb->coef ;
      if (abs(x)<=1.0e-6)
          /* 系数和为0， pa, pb分别直接后继结点 */
          { pa=pa->next ; pb=pb->next ; }
      else /* 若系数和不为0，生成的结点插入到结果链表的最后， pa, pb分别直接后继结点 */
          { p=(ploy *)malloc(sizeof(ploy)) ;
            p->coef=x ; p->expn=pb->expn ;
            p->next=NULL ;
            /* 生成一个新的结果结点并赋值 */
            pc->next=p ; pc=p ;
            pa=pa->next ; pb=pb->next ;
```

```
        }  
    }  
}  
/* end of while */  
if (pb!=NULL)  
while(pb!=NULL)  
{   p=(ploy *)malloc(sizeof(ploy)) ;  
    p->coef=pb->coef ; p->expn=pb->expn ;  
    p->next=NULL ;  
    /* 生成一个新的结果结点并赋值 */  
    pc->next=p ; pc=p ; pb=pb->next ;  
}
```



```
if (pa!=NULL)
```

```
while(pa!=NULL)
```

```
{ p=(p1 * )malloc(sizeof(p1)) ;
```

```
p->coef=p->coef ; p->expn=p->expn ;
```

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

```
/* 生成一个新的结果结点并赋值 */
```

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

```
}
```

```
return (p) ;
```

```
}
```

习题二

- 1 简述下列术语：线性表，顺序表，链表。
- 2 何时选用顺序表，何时选用链表作为线性表的存储结构合适？各自的主要优缺点是什么？
- 3 在顺序表中插入和删除一个结点平均需要移动多少个结点？具体的移动次数取决于哪两个因素？
- 4 链表所表示的元素是否有序？如有序，则有序性体现于何处？链表所表示的元素是否一定要在物理上是相邻的？有序表的有序性又如何理解？
- 5 设顺序表L是递增有序表，试写一算法，将x插入到L中并使L仍是递增有序表。

- 6 写一求单链表的结点数目ListLength(L)的算法。
- 7 写一算法将单链表中值重复的结点删除，使所得的结果链表中所有结点的值均不相同。
- 8 写一算法从一给定的向量A删除值在x到y($x \leq y$)之间的所有元素(注意：x和y是给定的参数，可以和表中的元素相同，也可以不同)。
- 9 设A和B是两个按元素值递增有序的单链表，写一算法将A和B归并为按按元素值递减有序的单链表C，试分析算法的时间复杂度。