第二章　线性表

* 2.1 线性表的类型定义
* 2.2 线性表的顺序表示和实现
* 2.3 线性表的链式表示和实现
* 2.3.1 线性链表
  2.3.2 循环链表
  2.3.3 双向链表

2.4 一元多项式的表示及相加*

2.1 线性表的类型定义

- 线性结构的特点：在数据元素的非空有限集中，
- (1)存在唯一的一个被称作“第一个”的数据元素；
- (2)存在唯一的一个被称作“最后一个”的数据元素；
- (3)除第一个之外，集合中的每个数据元素均只有一个前驱；
- (4)除最后一个之外，集合中每个数据元素均只有一个后继。

* 线性表中的数据元素可以是多种多样的，但同一线性表中的元素必定具有相同的特性，即属同一数据对象，相邻数据元素之间存在序偶关系。

* 若将线性标记为
* $(a_1, ..., a_{i-1}, a_i, a_{i+1}, ..., a_n)$
* $< a_{i-1}, a_i >$
* $<a_i, a_{i+1}>$
* a_{i-1}是a_i的直接前驱元素， a_{i+1}是a_i的直接后继元素。

例、学生健康情况登记表如下:

姓　名	学　　号	性　别	年龄	健康情况
王小林	790631	男	18	健康
陈　红	790632	女	20	一般
刘建平	790633	男	21	健康
张立立	790634	男	17	神经衰弱
……	……	……	……	……

- 线性表中元素的个数n（n>=0)定义为线性表的**长度**，n=0时称为**空表**。
- 在非空表中的每个数据元素都有一个确定的位置，ai是第i 个元素，称i 是数据元素ai在线性表中的**位序**。
- 线性表的长度可根据需要增长或缩短。（插入、删除操作）

* 抽象数据类型线性表的定义如下：
* ADT List{
* 数据对象：
* D={ai|ai∈Elemset，i=1，2，...n，n>=0}
* 数据关系：
* R1={< a_{i-1}, a_i >| a_{i-1}, a_i ∈D}
* 基本操作：
* InitList(&L);
* 操作结果：构造一个空的线性表L。

* DestroyList(&L);
* 初始条件：线性表L已存在。
* 操作结果：销毁线性表L。
* ClearList(&L);
* 初始条件：线性表L已存在。
* 操作结果：将L重置为空表。
* ListEmpty(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。
* ListLength(L)
* 初始条件：线性表L已存在。
* 操作结果：返回L中数据元素个数。

* GetElem(L, i, &e)
* 初始条件：线性表L已存在，1≤i≤ListLength(L)。
* 操作结果：用e返回L中第i 个数据元素的值。
* LocateElem(L, e, compare())
* 初始条件：线性表L已存在，compare()是数据元素判定函数。
* 操作结果：返回L中第1个与e满足关系compare()的数据元素的位序。若这样的数据元素不存在，则返回0。
* PriorElem(L, cur_e, &pre_e)
* 初始条件：线性表L已存在。
* 操作结果：若cur_e是L的数据元素，且不是第1个，则用pre_e返回它的前驱，否则操作失败， pre_e无定义。

* NextItem(L, cur_e, &next_e)
* 初始条件：线性表L已存在。
* 操作结果：若cur_e是L的数据元素，且不是最后一个，则用next_e返回它的后继，否则操作失败， next_e无定义。
* ListInsert(&L, i, e)
* 初始条件：线性表L已存在，1≤i≤ListLength(L)+1。
* 操作结果：在L中第i 个位置之前插入新的数据元素e，L的长度加1。
* ListDelete(&L, i, &e)
* 初始条件：线性表L已存在，1≤i≤ListLength(L) 。
* 操作结果：删除L的第i 个数据元素，并用e返回其值，L的长度减1。

* ListTraverse(L, visit())
* 初始条件：线性表L已存在。
* 操作结果：依次对L的每个数据元素调用函数visit（）。一旦visit（）失败，则操作失败。
* }ADT List

* 对上述定义的抽象数据类型线性表，还可进行一些更复杂的操作。
* 例：假设利用两个线性表LA和LB分别表示两个集合A和B（即线性表中的数据元素即为集合中的成员），现要求一个新的集合，使得A=A ∪ B。

需要对线性表进行如下操作：
扩大线性表A，将存在于线性表LB中而不存在于LA中的数据元素插入到线性表LA中。

LA	a	b	c	d	e	f		
LB	a	b	f	g	v			
LA	a	b	c	d	e	f	g	v

- 从LB中依次取得每个数据元素，并依值在LA中进行查访，若不存在，则执行插入操作。
- Void union(List &La, List Lb){
- La_len = ListLength(La);
- Lb_len = ListLength(Lb);
- for (i =1; i <=Lb_len; i ++) {
- GetElem(Lb, i, e);
- if (!LocateElem(La, e, equal()))
- ListInsert(La, ++La_len, e)
- }
- }

2.2线性表的顺序表示和实现

- 线性表的顺序表示指的是用一组地址连续的存储单元依次存储线性表的数据元素。

- 假如线性表每个元素占用L个存储单元，并以第一个单元的存储地址作为数据元素的存储位置，则
- $LOC(a_{i+1}) = LOC(a_i) + L$
- $LOC(a_i) = LOC(a_1) + (i-1) * L$

线性表第1个元素a1的存储位置，
称作线性表的起始位置或基地址。

* 线性表的动态分配顺序存储结构：

```
#define  LIST_INIT_SIZE  100
#define  LISTINCREMENT  10
Typedef  struct {
    ElemType   *elem;
    int          length;
    int          listsize;
} SqList;
```

← 线性表存储空间的初始分配量

← 线性表存储空间的分配增量

← 指示线性表基地址

← 指示线性表当前长度

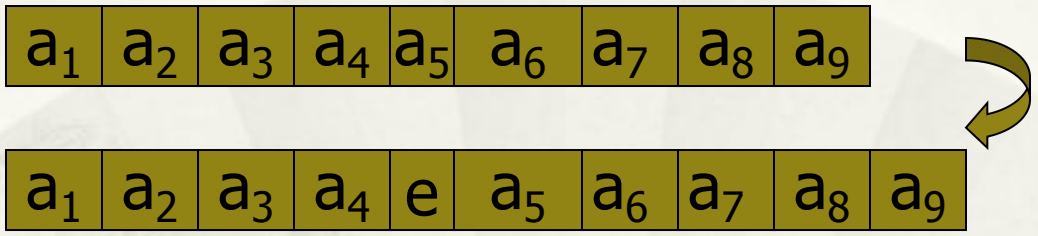
← **以sizeof(Elemtype)为单位**

指示顺序表当前分配的存储空间大小，一旦因插入元素而空间不足时，可进行再分配，即为顺序表增加一个大小为存储LISTINCREMENT个数据元素的空间

顺序表的初始化操作：

```
status InitList_Sq(SqList &L) {
    L.elem = (ElemType *)malloc(LIST_INIT_SIZE*
                                sizeof(ElemType));
    if(!L.elem)   exit (OVERFLOW);
    L.length = 0;
    L.listsize = LIST_INIT_SIZE;
    return OK;
}
```

线性表的插入和删除操作在顺序存储表示时的实现方法



* 一般情况下，在第i 个（1 ≤ i≤n）个元素之前插入一个元素时，需将第n至第i （共n-i+1）个元素向后移动一个位置。

```
status ListInsert_Sq(SqList &L, int i, ElemType e) {
    if (i<1 || i >L.length+1)  return ERROR; //异常处理
    if (L.length>=L.ListSize) { //存储空间不足
       newbase=(ElemType *)realloc(L.elem,
                (L.ListSize+LISTINCREMENT)*sizeof(ElemType));
       if(!newbase) exit(OVERFLOW);
       L.elem = newbase;
       L.ListSize+=LISTINCREMENT;
     }
     q=&(L.elem[i-1]);
     for(p=&(L.elem[L.length-1]);p>=q;--p)  *(p+1)=*p;//从n向i 依次移动
     *q=e; ++L.length; return OK;
}
```

对于删除操作，线性表的逻辑关系发生了变化，表长减1。

一般情况下，删除第i （1 ≤ i≤n）个元素时需将从第i+1至第n（共n-i ）个元素依次向前移动一个位置。

```
Status ListDelete_Sq(SqList &L, int i, ElemType &e){
    if ((i <1)||(i>L.length))   return  ERROR;
    p=&(L.elem[i-1]);  e=*p;
    q=L.elem+L.length-1;
    for(++p;p<=q;++p)  *(p-1)=*p;
    --L.length;
    return OK;
}
```

- 算法时间复杂度的分析：
- 假设pi是在第i 个元素之前插入一个元素的概率，则在长度为n的线性表中插入一个元素时，所需移动的元素的次数的期望值（平均次数）为
-

$$E_{is} = \sum_{i=1}^{n+1} P_i(n-i+1)$$

- 不失一般性，假定在线性表的任何位置上插入或删除元素是等概率的。即

$$P_i = \frac{1}{n+1}$$

$$E_{is} = \sum_{i=1}^{n+1} P_i(n-i+1) = \frac{1}{n+1}\frac{n(n+1)}{2} = \frac{n}{2}$$

- 类似的可以求出

$$E_{dl} = \sum_{i=1}^{n} P_i(n-i) = \frac{1}{n}\frac{n(n-1)}{2} = \frac{n-1}{2}$$

- 则顺序存储结构的线性表中插入或删除一个数据元素，平均约移动表中一半元素。若表长为n，则算法ListInsert_Sq和ListDelete_Sq的时间复杂度为O（n）。

- 在顺序存储表示的线性表中插入或删除一个数据元素，平均约需移动表中**一半**元素。这个数目在线性表的长度较大时是很可观的。这个缺陷完全是由于顺序存储要求线性表的元素依次紧挨存放造成的。因此，这种顺序存储表示仅适用于不常进行插入和删除操作、表中元素相对稳定的线性表。

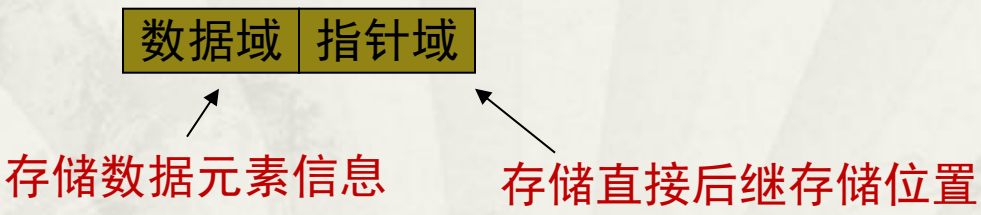
2.3线性表的链式表示和实现

- 线性表的顺序映象结构的特点：逻辑关系上相邻的两个元素，在物理位置上也相邻，可以随机存取表中任一元素。
- 弱点：在做插入或删除操作时，需移动大量元素。

- 线性表的链式存储结构的特点是用一组任意的存储单元存储线性表的数据元素。（这些存储单元可以是连续的，也可以是不连续的）

2.3.1　线性链表

* 有关线性链表的基本概念：
* 结点：包括两个域



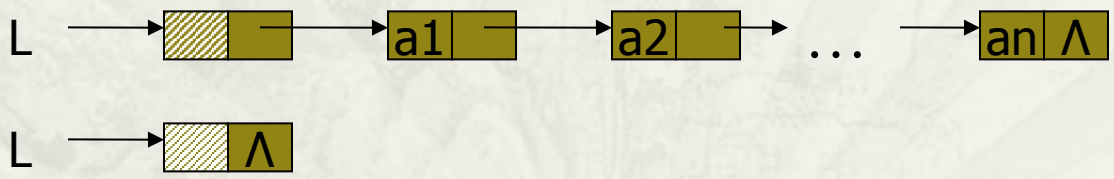
* 结点中只包含一个指针域的链表称为线性链表或单链表。
* 整个链表的存取必须从头指针开始进行，头指针指示链表中第一个结点的存储位置，线性链表中最后一个结点的指针域为“空”。

例、线性表(a1,a2,a3,a4,a5,a6,a7,a8,a9,a10)的单链表示意图如下：

- 单链表的存储结构可以用“结构指针”描述为
- typedef struct Lnode {
- 　　ElemType　data;
- 　　struct Lnode *next;
- } Lnode,*LinkList;

- 为了便于处理一些特殊情况，在第一个结点之前附加一个“头结点”，令该结点中指针域的指针指向第一个元素结点，并令头指针指向头结点

* 值得注意的是，若线性表为空，在不带头结点的情况下，头指针为空(NULL)，但在带头结点的情况下，链表的头指针不为空，而是其头结点中指针域的指针为空

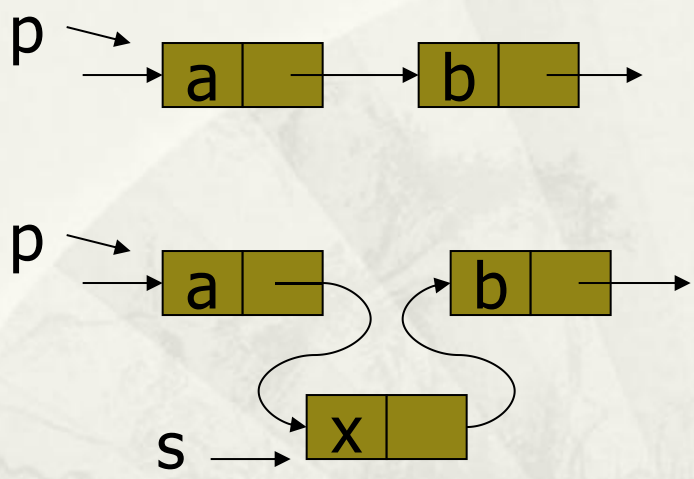


单链表的查访操作如何实现？

单链表是一种"顺序存取"的结构，即：为取第 i 元素，首先必须先找到第 i-1 个元素。因此不论 i 值为多少，都必须从头结点开始起"点数"，直数到第 i 个为止。头结点可看成是第0个结点。

```
status  GetElem_L(LinkList L, int i, ElemType &e) {
        p=L->next; j=1;
        while (p && j < i ) {
                p=p->next; ++j;
        }
        if ( !p || j>i )  return ERROR;
        e = p->data;
        return OK;
}
```

* 对单链表执行插入操作：



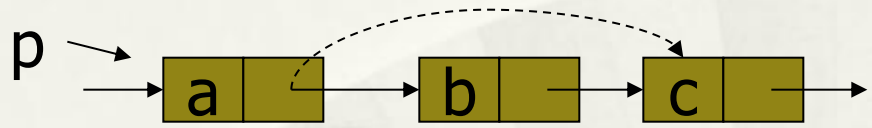
假设S为指向结点x的指针，则：

s->next = p->next;

P->next=s;

```
status ListInsert_L(LinkList &L, int i, ElemType e){
    p=L; j=0;
    while (p && j< i-1 ) {
          p = p->next;  ++j;
    }
    if (!p || j>i-1 )   return ERROR;
    s=(LinkList)malloc(sizeof(Lnode));
    s->data=e; s->next=p->next;
    p->next=s;
    return OK;
}
```

* 在单链表中删除结点时指针变化情况：



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

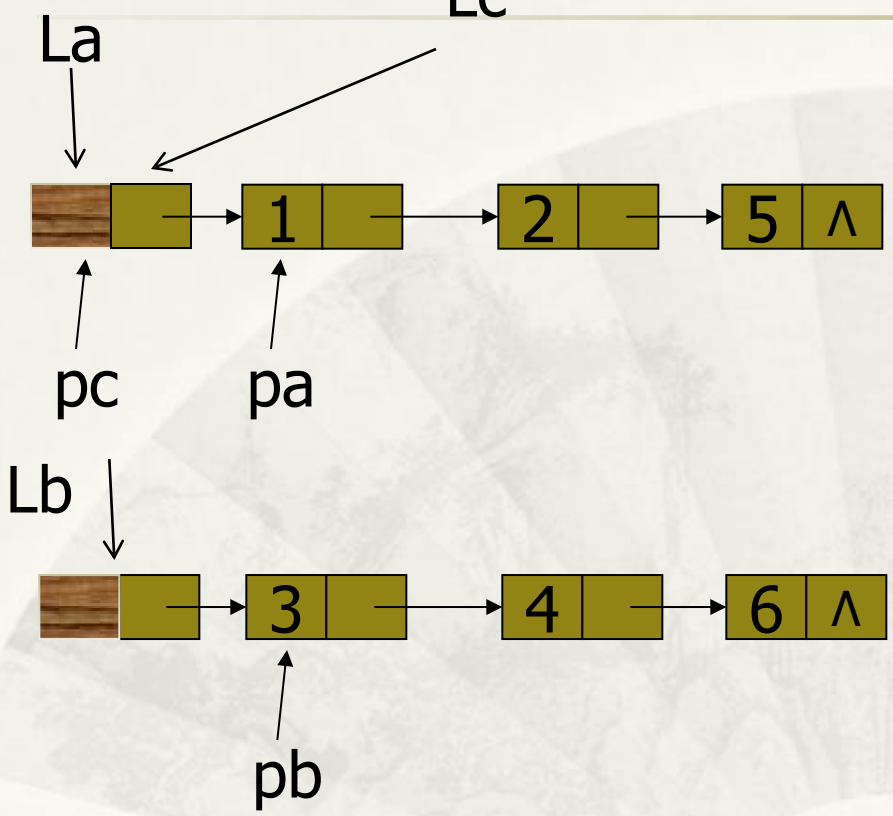
```
status ListDelete_L(LinkList &L, int i; ElemType &e){
    p = L; j=0;
    while ( p->next && j < i-1 ) {
        p=p->next; ++j;
    }
    if ( !(p->next) || j > i-1 ) return ERROR;
    q=p->next; p->next=q->next;
    e=q->data; free(q);
    return  OK;
}
```

- 逆序创建链表
- 链表的动态生成：从“空表”的初始状态起，依次建立各元素结点，并逐个插入链表。

```
void CreateList_L(LinkList &L, int n){
    L = (LinkList)malloc(sizeof(Lnode));
    L->next = NULL;
    for ( i =n;i>0;--i){
        p=(LinkList)malloc(sizeof(Lnode));
        scanf(&p->data);
        p->next=L->next;
        L->next=p;
    }
}
```

如何将两个有序链表合并为一个有序链表：

```
void MergeList_L(LinkList &La,LinkList &Lb,LinkList &Lc){
      pa=La->next; pb=Lb->next;
      Lc=pc=La;
      while(pa && pb ){
            if (pa->data <= pb->data ){
                  pc->next=pa; pc=pa; pa=pa->next;
            }
            else { pc->next=pb;pc=pb;pb=pb->next;}
       }
       pc->next=pa?pa:pb;
       free(Lb);
}
```

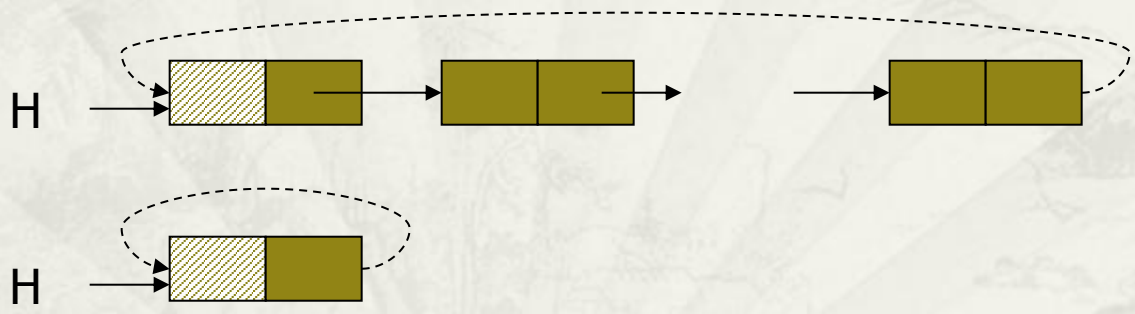



```
if (pa->data <= pb->data ){
        pc->next=pa;
        pc=pa;
        pa=pa->next;            }
else {
        pc->next=pb;
        pc=pb;
        pb=pb->next;}
```

```
pc->next=pa?pa:pb;
```

2.3.2 循环链表

- 循环链表（Circular Linked List）
- 特点：表中最后一个结点的指针域指向头结点，整个链表形成一个环。由此，从表中任一结点出发均可找到表中其他结点。

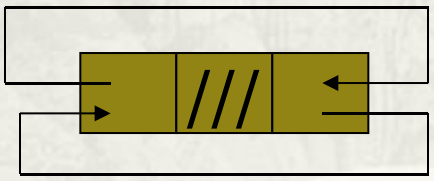


- 循环条件的判断不是p或p->next是否为空，而是它们是否等于头指针。

* 在单链表中只有一个指示直接后继的指针域，从某个结点出发只能顺时针往后寻查其他结点。若要寻查结点的直接前驱，则需从表头指针出发，为此，引出了双向链表。
* 在**双向链表**的结点中有两个指针域，其一指向直接后继，另一指向直接前驱，用C语言描述为：

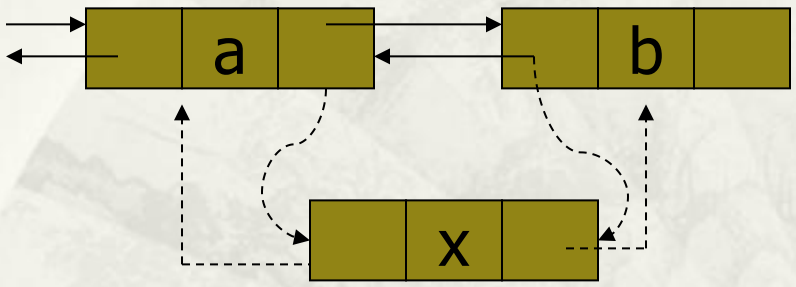
```
typedef struct DulNode{
    ElemType data;
    struct DulNode *prior;
    struct DulNode *next;
}DulNode, *DuLinkList;
```

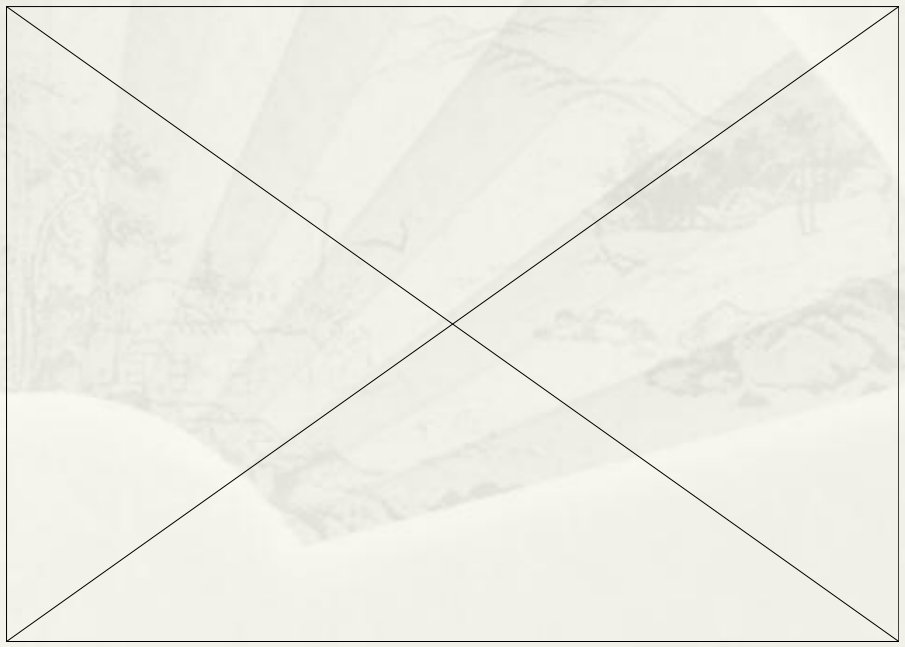
* 与单链表类似，双向链表也是由指向头结点的头指针唯一确定，若将头尾结点链接起来则构成**双向循环链表**。空的双向循环链表则由只含一个自成双环的头结点表示。



- 在双向链表中，若d为指向表中某一结点的指针，则有

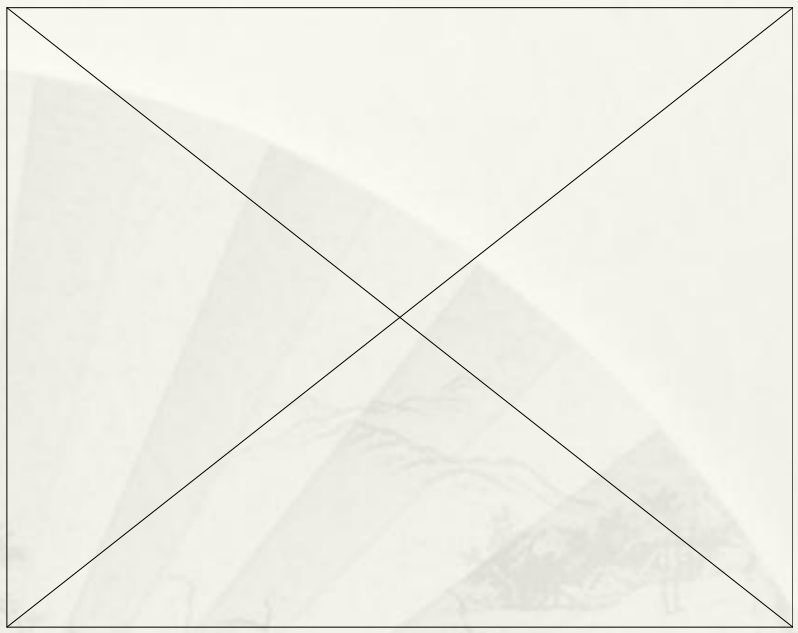
  d->next->prior=d->prior->next=d
- 在对双向链表进行插入和删除操作时，需同时修改两个方向上的指针。





```
status ListInsert_Dul(DulLinkList &L, int i, ElemType e){
    if (!(p=GetElemp_Dul(L,i)))   return  ERROR;//空表
    if(!(s=(DuLinkList)malloc(sizeof(DulNode)))) return
    ERROR;//无足够空间分配
    s->data=e;
    s->prior=p->prior;p->prior->next=s;
    s->next=p;
    p->prior=s;
    return OK;
}
```

```
void  ddeletenode(dlistnode *p)
    {
      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，在双链表中插入和删除必须同时修改两个方向上的指针。上述两个算法的时间复杂度均为O(1)。

多项式相加问题

- 存储结构的选取

任一一元多项式可表示为$P_n(x)=P_0+P_1x+P_2x^2+...+P_nx^n$，显然，由其n+1个系数可惟一确定该多项式。故一元多项式可用一个仅存储其系数的线性表来表示，多项式指数i隐含于P_i的序号中。

$$P=(P_0，P_1，P_2，...，P_n)$$

若采用顺序存储结构来存储这个线性表，那么多项式相加的算法实现十分容易，同位序元素相加即可。

- 但当多项式的次数很高而且变化很大时，采用这种顺序存储结构极不合理。例如，多项式$S(x)= 1+ 3x+12x^{999}$需用一长度为1000的线性表来表示，而表中仅有三个非零元素，这样将大量浪费内存空间。
- 此时可考虑另一种表示方法，如线性表S(x)可表示成S=((1,0)，(3,1), (12,999))，其元素包含两个数据项：系数项和指数项。

- 第二种多项式表示方法在计算机内同样对应两种存储方式：
  - 顺序存储结构
  - 链式存储结构
- 当只对多项式进行访问、求值等不改变多项式指数(即表的长度不变化)的操作时，宜采用顺序存储结构；当要对多项式进行加法、减法、乘法等改变多项式指数的操作时，宜采用链式存储结构。

* 抽象数据类型一元多项式的定义如下：
* ADT List{
* 数据对象：
* D={ai|ai∈Termset，i=1，2，...m，m>=0, Termset中的每个元素包含一个表示系数的实数和表示指数的实数}
* 数据关系：
* R1={< a_{i-1}, a_i >| a_{i-1}, a_i ∈D，且a_{i-1} 中的指数< a_i 中的指数}
* 基本操作：
* CreatePoly(&P, m);
* 操作结果：输入m项系数和指数，建立一元多项式

* DestroyPolyn(&P);
* 初始条件：一元多项式P已存在。
* 操作结果：销毁一元多项式P。
* PrintPolyp(P);
* 初始条件：一元多项式P已存在。
* 操作结果：打印输出一元多项式。
* PolynLength(P)
* 初始条件：一元多项式P已存在。
* 操作结果：返回一元多项式P中的系数。
* AddPolyn(&Pa, &Pb)
* 初始条件：一元多项式Pa和Pb已存在。
* 操作结果：完成多项式相加运算，即Pa = Pa + Pb, 并销毁一元多项式Pb。

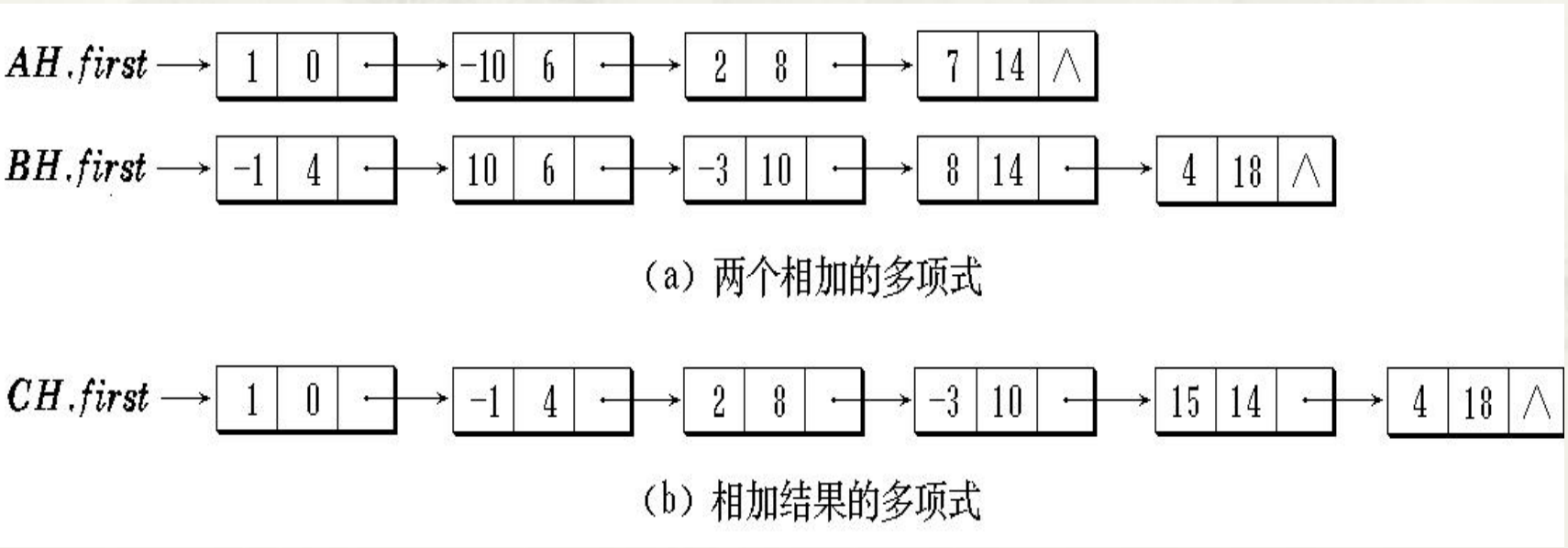
* SubstractPolyn(&Pa, &Pb);
* 初始条件：一元多项式Pa和Pb已存在。
* 操作结果：完成多项式相减运算，即Pa=Pa-Pb，并且销毁一元多项式Pb。
* MultiPolyp(&Pa, &Pb);
* 初始条件：一元多项式Pa和Pb已存在。
* 操作结果：完成多项式相乘运算，即Pa=Pa * Pb，并销毁一元多项式Pb。
* }ADT Polynomial.

例　$AH = 1 - 10x^6 + 2x^8 + 7x^{14}$

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

$CH(x)=AH(x)+BH(x)$

它的单链表表示如图



（a）两个相加的多项式

（b）相加结果的多项式

一元多项式相加预算规则（假设指针qa和qb分别指向多项式A和B中的某个点）：

1. 指针qa所指结点的指数值<指针qb所指结点的指数值，则摘取qa指针所指的结点插入到“和多项式”链表中；
2. 指针qa所指结点的指数值>指针qb所指结点的指数值，则摘取qb指针所指的结点插入到“和多项式”链表中；
3. 指针qa所指结点的指数值=指针qb所指结点的指数值，则将两个结点中的系数相加，若和数不为零，则修改qa所指结点的系数值，同时释放qb所指结点；反之，从多项式A中删除相应结点，释放指针qa和qb所指结点；

```
typedef struct
{
    float coef;    //系数
    int            len;     //指数
}term, ElemType; //两个类型名
Typedef LinkList polynomial;
//完成多项式相加运算，即：Pa = Pa + Pb, 并销毁一元多项式
void AddPolyn(polynomial &Pa, polynomial &Pb) {
    ha = GetHead (Pa); hb = GetHead (Pb); //头结点位置
    qa = NextPos(ha);  qb = NextPos(hb);  //开始结点位置
    while( qa && qb )  {
       a = GetCurElem(qa);  b = GetCurElem(qb);
    switch (*cmp(a,b)) {
          case -1: // 多项式PA中当前结点的指数小
             ha = qa; qa=NextPos(Pa, qa); break;
```

```
        case 0:  // 两者的指数值相等
          sum = a.coef + b.coef;
          if( sum != 0.0) {  SetCurElem(qa, sum); ha=qa;          }
          else {  DelFirst(ha, qa); FreeNode(qa); }
          DelFirst(hb, qb); FreeNode(qb); qb=NextPos(Pb, hb);
          qa=NextPos(Pa, qa) ;break;
        case 1: //多项式PB中当前结点的指数小
           DelFirst(hb, qb); InsFirst(ha, qb);
          qb=NextPos(Pb, hb); ha=NextPos(Pa, ha); break;
        }          // End of Switch
    }  // End of while
    if ( ! ListEmpty (Pb) ) Append(Pa, qb); //链接Pb中剩余结点
    FreeNode(hb);    // 释Pb头结点
} // End of AddPolyn()
```

小结

- 线性表是 n(n≥0) 个数据元素的序列
- 因此线性表中除了第一个和最后一个元素之外都只有一个前驱和一个后继。线性表中每个元素都有自己确定的位置，即“位序”。

- n=0时的线性表称为"空表"，它是线性表的一种特殊状态，因此在写线性表的操作算法时一定要考虑你的算法对空表的情况是否也正确。

* 顺序表是线性表的顺序存储结构的一种别称。它的特点是以"存储位置相邻"表示两个元素之间的前驱后继关系。因此，顺序表的优点是可以随机存取表中任意一个元素，其缺点是每作一次插入或删除操作时，平均来说必须移动表中一半元素。常应用于主要是为查询而很少作插入和删除操作，表长变化不大的线性表。

* 链表是线性表的链式存储结构的别称。它的特点是以"指针"指示后继元素，因此线性表的元素可以存储在存储器中任意一组存储单元中。它的优点是便于进行插入和删除操作，但不能进行随机存取，每个元素的存储位置都存放在其前驱元素的指针域中，为取得表中任意一个数据元素都必须从第一个数据元素起查询。由于它是一种动态分配的结构，结点的存储空间可以随用随取，并在删除结点时随时释放，以便系统资源更有效地被利用。这对编制大型软件非常重要，作为一个程序员在编制程序时必须养成这种习惯。

复习提要

- 在这一章，第一次系统性地引入链式存储的概念，链式存储概念将是整个数据结构学科的重中之重，无论哪一章都涉及到了这个概念。
- 1.线性表的相关基本概念，如：前驱、后继、表长、空表、首元结点，头结点，头指针等概念。
-
- 2.线性表的结构特点，主要是指：除第一及最后一个元素外，每个结点都只有一个前趋和只有一个后继。
-
- 3.线性表的顺序存储方式及其在具体语言环境下的两种不同实现：表空间的静态分配和动态分配。静态链表与顺序表的相似及不同之处。

* 4.线性表的链式存储方式及以下几种常用链表的特点和运算：单链表、循环链表，双向链表，双向循环链表。在链表的小题型中，经常考到一些诸如：判表空的题。在不同的链表中，其判表空的方式是不一样的，请大家注意。

* 5.线性表的顺序存储及链式存储情况下，其不同的优缺点比较，即其各自适用的场合。单链表中设置头指针、循环链表中设置尾指针而不设置头指针以及索引存储结构*的各自好处。

作业

- 1。描述以下三个概念的区别：头指针，头结点，首元结点(第一个元素结点)。

  2. 简述线性表的两种存储结构顺序表和链表的优缺点。

- 3。已知 P 是指向双向链表中某个结点的指针，试写出删除 P 所指结点的前驱结点的语句序列。

- **4。简述以下算法的功能。**

(1) Status A(LinkedList L) { // L 是无表头结点的单链表

```
if (L && L->next){
    Q =L; L =L->next; P =L ;
    while ( P->next) P =P->next ;
    P->next =Q; Q->next = NULL;
    }
return OK;
} // A
```

- (2) **void** BB(LNode *s, LNode *q) {
  p =s ;
  while (p->next!=q) p =p->next ;
  p->next =s;
  } //BB
  void AA(LNode *pa, LNode *pb) {
  // pa 和 pb 分别指向单循环链表中的两个结点
  BB(pa, pb);
  BB(pb, pa);
  } //AA

经典面试题

- 题目1：给定链表的头指针和一个结点指针，在O(1)时间删除该结点。
- 题目2：设计算法判断单链表中是否存在环。算法的时间复杂度要求是O(n)，空间复杂度要求是O(1)。