数据结构与算法

第二章　线性表

线性表的类型定义
线性表的顺序表示和实现
线性表的链式表示和实现
一元多项式的表示及相加

第二章 线性表

线性结构特点：在数据元素的非空有限集中

- 存在唯一的一个被称作“第一个”的数据元素
- 存在唯一的一个被称作“最后一个”的数据元素
- 除第一个外，集合中的每个数据元素均只有一个前驱
- 除最后一个外，集合中的每个数据元素均只有一个后继

线性表的种类

链表、栈、队列、串、数组

2.1 线性表的类型定义

- 定义：一个线性表是n个数据元素的有限序列

  $(a_1, a_2, a_3, \ldots\ldots, a_n)$。

  (A, B, C, ……, Z)。

  (6, 17,28,50, 92, 188)。

结构数组

可用结构表示

数据项

记录

学号	姓名	性别	年龄	班级
0001	张三	男	19	物流02
0002	李四	男	18	物流02
0003	王岚	女	19	物流02

- 要求：
  - 元素同构，属于同一数据对象。通常可用C语言的数据类型描述
  - 不能出现缺项

线性表的形式定义

- 定义：含有n个元素的线性表是一个数据结构

  Linear_List = (D, R)

  其中，$D=\{a_i | a_i \in ElemSet, i=1, 2, \ldots, n \geq 0\}$

  $R=\{<a_{i-1}, a_i> | a_{i-1}, a_i \in D, i=1, 2, \ldots, n\}$

- 元素个数n — 称为**表的长度**，当n=0称为**空表**
- 当1<i<n时
  - a_i的直接**前驱**是a_{i-1}，a_1无直接前驱
  - a_i的直接**后继**是a_{i+1}，a_n无直接后继
- 线性表是一个相当灵活的数据结构，它的长度因需可长可短。因此对线性表的操作不仅有查询，还有插入和删除。

线性表的基本操作

- InitList(&L)
- DestroyList(&L)
- ClearList(&L)
- ListEmpty(L)
- ListLength(L)
- GetElem(L, I, &e)
- LocateElem(L, e, compare())
- PriorElem(L, cur_e, &pre_e)
- NextElem(L, cur_e, &next_e)
- ListInsert(&L, i, e)
- ListDelete(&L, i, &e)
- ListTraverse(L, visit())

注意:&表示传地址，与C语言不一样，它是C++的表示法

线性表的扩展操作

- 例:线性表La和Lb表示两个集合，现要求两集合的并，并存放在线性表La中
- 例:线性表La和Lb是按元素序非递减的，合并两个线性表为Lc，使Lc的元素亦为非递减序

分析：显然，要求La和Lb的元素是同类数据对象。Lc是否为空表也需考虑。若Lc为空表，仅需考虑La和Lb元素。用两个指示变量分别取La和Lb的元素，比较其大小，依次加入到Lc中。

算法：教材P21

进一步：

(1) 若Lc不为空?

则要求Lc的元素与La,Lb的元素属同类数据对象

(2) La、Lb无序呢?

考虑分为两步：

(a) MergeList – 合并表，与教材中例的算法含义不一样!

(b) SortList – 表排序

2.2 线性表的顺序表示和实现

- 线性表的顺序表示是指用一组地址连续的存储单元依次存储线性表的数据元素。这种线性表也叫**顺序表**。
- 元素地址计算方法：
  - $LOC(a_{i+1})=LOC(a_i)+L$
  - $LOC(a_i)=LOC(a_1)+(i-1)*L$

  其中：

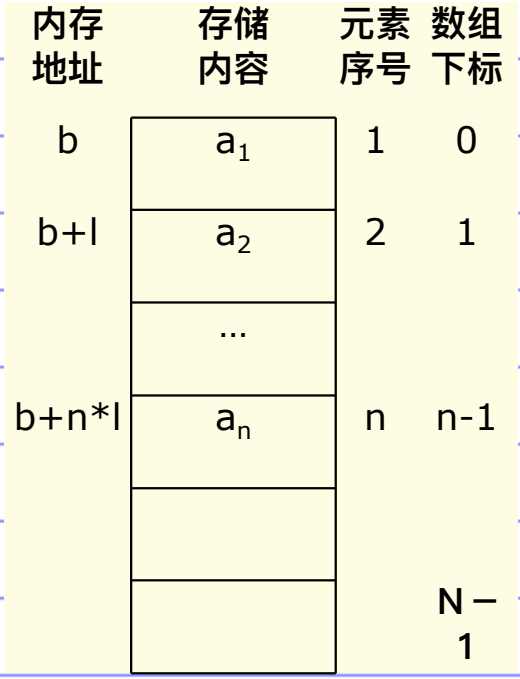
  L — 一个元素占用的存储单元个数

  $LOC(a_i)$ — 线性表第i个元素的地址
- $LOC(a_1)$是线性表第一个元素a_1的存储地址，也叫线性表的起始位置或基地址
- 线性表的这种内存存储表示也叫线性表的顺序存储结构，或顺序映像(Sequential Mapping)

顺序表的内存映像

- 特点：
  - 实现逻辑上相邻—物理地址相邻
  - 实现随机存取
- 实现：可用C语言的一维数组实现

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



注意：元素用指针，是考虑到要动态分配内存

顺序表基本操作

- 关于数据元素的定义
- 初始化
- 撤销
- 插入一个元素
- 删除一个元素
- 插入/删除算法的复杂性
- 查找/定位一个元素
- 顺序表的特点

关于数据元素的定义

- 在前面的例子中

$(a_1, a_2, a_3, \ldots\ldots, a_n)$。

(A, B, C, ……, Z)。

(6, 17,28,50, 92, 188)。

数组

结构数组

学号	姓名	性别	年龄	班级
0001	张三	男	19	物流02
0002	李四	男	18	物流02
0003	王岚	女	19	物流02

顺序表初始化操作

```
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;
}
```

顺序表插入操作

```
Status InsertList (SqList &L, int i, ElemType e) {
    // 在顺序线性表 L 中第 i 个位置之前插入新的元素 e,
    // i 的合法值为 1≤i≤ListLength_Sq(L) + 1
    if (i < 1 || i > L.length + 1) return ERROR;   // i 值不合法
    if (L.length >= L.listsize) {          // 当前存储空间已满,增加分配
      newbase = (ElemType *)realloc(L.elem,
                    (L.listsize + LISTINCREMENT) * sizeof (ElemType));
      if (!newbase)exit(OVERFLOW);         // 存储分配失败
      L.elem = newbase;                    // 新基址
      L.listsize += LISTINCREMENT;         // 增加存储容量
    }
    q = &(L.elem[i-1]);                    // q 为插入位置

    for (p = &(L.elem[L.length-1]); p >= q; --p)
      *(p+1) = *p; // 插入位置及之后的元素右移
    *q = e;                // 插入 e
    ++L.length;            // 表长增 1
    return OK;
} // InsertList
```

再次注意这里：给定了表长度，若表长度不够，仍重分配内存

顺序表删除操作

- 删除第i(1≤i≤n) 个元素时，需前移(n-i)个元素
- 平均移动次数 (n-1)/2

```
Status ListDelete_Sq(SqList &L, int i, ElemType &e) {
    // 在顺序线性表 L 中删除第 i 个元素，并用 e 返回其值
    // i 的合法值为 1≤i≤ListLength_Sq(L)
    if ((i < 1) || (i > L.length))
                return ERROR;          // i 值不合法
    p = &(L.elem[i-1]);                // p 为被删除元素的位置
    e = *p;                            // 被删除元素的值赋给 e
    q = L.elem + L.length - 1;         // 表尾元素的位置
    for (++p; p <= q; ++p)
        *(p-1) = *p;                   // 被删除元素之后的元素左移
    --L.length;                        // 表长减 1
    return OK;
} // ListDelete_Sq
```

顺序表插入删除操作复杂性

假设 p_i 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元素次数的期望值(平均次数)为

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

假设 q_i 是删除第 i 个元素的概率，则在长度为 n 的线性表中删除一个元素时所需移动元素次数的期望值(平均次数)为

$$E_{dl} = \sum_{i=1}^{n} q_i(n - i)$$

不失一般性，我们可以假定在线性表的任何位置上插入或删除元素都是等概率的，即

$$p_i = \frac{1}{n+1}, \quad q_i = \frac{1}{n}$$

则式(2-3)和(2-4)可分别简化为式(2-5)和(2-6)：

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

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

由式可见，在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 $O(n)$。

顺序表查找操作

- 给出一个位置i(1≤i≤n)，直接定位
- 给出一个元素e

```
int LocateElem_Sql(SqlList L, ElemType e,
        Status (*compare)(ElemType, ElemType) {
        // 在顺序表L中查找第1个值与e满足
        // compare()的元素的位置，若找到，
        // 则返回其在L中的位置，否则返回0
        i = 1;          // i 的初值为第1个元素的位置
        p = L.elem;  // p 的初值为第1个元素的存储位置
        while (i<=L.length && !(*compare)(*p++, e))
              ++i;
        if (i<=L.length) return i;
        return 0;
} // LocateElem_Sql
```

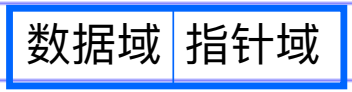
顺序存储结构的优缺点

- 优点
  - 逻辑相邻，物理相邻
  - 可随机存取任一元素
  - 存储空间使用紧凑
- 缺点
  - 插入、删除操作需要移动大量的元素
  - 预先分配空间需按最大空间分配，利用不充分
  - 表容量难以扩充

2.3 线性表的链式表示和实现

- 特点：
  - 用一组任意的存储单元存储线性表的数据元素
  - 利用指针实现了用不相邻的存储单元存放逻辑上相邻的元素
  - 每个数据元素a_i，除存储本身信息外，还需存储其直接后继的信息——结点
    - ✧ 数据域：元素本身信息
    - ✧ 指针域：指示直接后继的存储位置(链)

结点



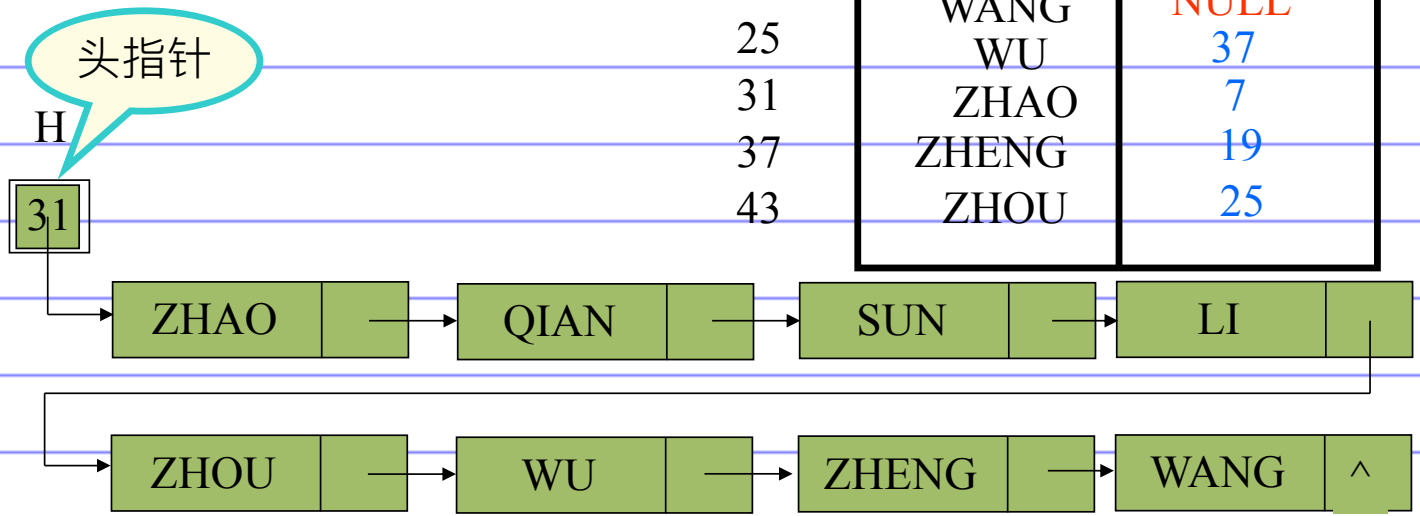

```
typedef struct LNode {
        ElemType Data;
        struct LNode *Next;
} LNode, *LinkList;
```

线性表的链式表示例

例：线性表

(ZHAO,QIAN,SUN,
LI,ZHOU,WU,
ZHENG,WANG)

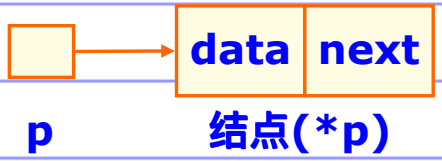
存储地址	数据域	指针域
1	LI	43
7	QIAN	13
13	SUN	1
19	WANG	NULL
25	WU	37
31	ZHAO	7
37	ZHENG	19
43	ZHOU	25



线性链表

- 定义：结点中只含一个指针域的链表叫~，也叫单链表
- 实现：

```
typedef struct LNode {
    ElemType data;
    struct LNode  *next;
} LNode, *LinkList;
LNode   *h,*p;
```



(*p)表示p所指向的结点

(*p).data⇔p->data表示p指向结点的数据域

(*p).next⇔p->next表示p指向结点的指针域

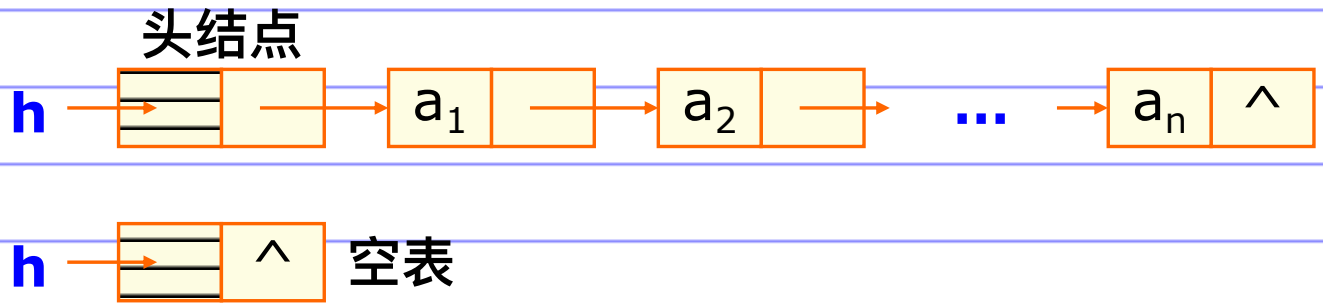
生成一个Node型新结点：

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

系统回收p结点：free(p)

头结点

- 头结点：在单链表第一个结点前附设一个结点叫~
- 头结点指针域为空表示线性表为空
- 在单链表中，任何两个元素的存储位置之间没有固定的关系。然而，每个元素的存储位置都包含在其直接前驱结点的信息之中。



单链表中查找与定位元素

- 查找：查找单链表中是否存在数据为X的结点，若有则返回指向含X的结点指针；否则返回NULL。算法描述如下：

```
LinkList *LinkListSearch(LinkList L, ElemType X){
        LinkList *p = L->next;
        while(p!=NULL && p->data!=X)
            p=p->next;
        return (p);
 }
```

- 算法评价：
  - 最好情况首节点即是，最坏情况是未找到，平均循环和比较次数为(n+1)/2.
  - 时间复杂性 $T(n) = O(n)$

```
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;
} // GetElem_L
```

- **算法评价:**

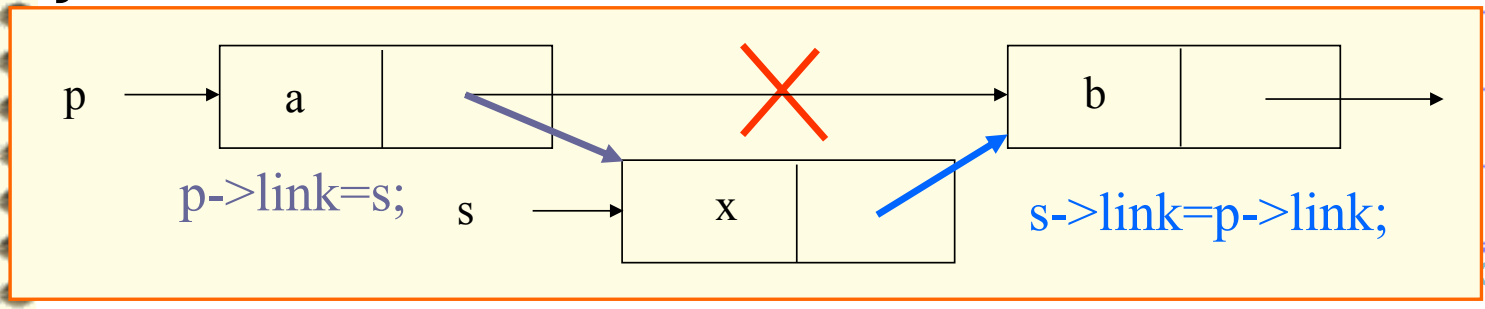
  时间复杂性 $T(n) = O(n)$

单链表中插入结点

- 在第i个位置之前插入元素e:

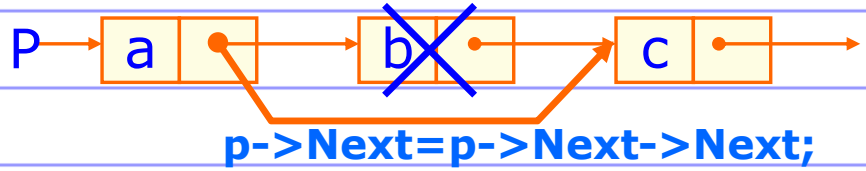
```
Status ListInsert_L(LinkList &L, int i, ElemType e) {
   p = L; j = 0;
   while(p && j<i-1) {p = p->next; ++j;}
   if(!p || j>i-1) return ERROR;
   s = (LinkList)malloc(sizeof(LNode));
   s->data = e; s->next = p->next;p->next = s;
   return OK;
}
```

↳此时P在(i-1)这个位置



单链表中删除结点

- 删除：单链表中删除b，设p指向a 。算法描述如下：

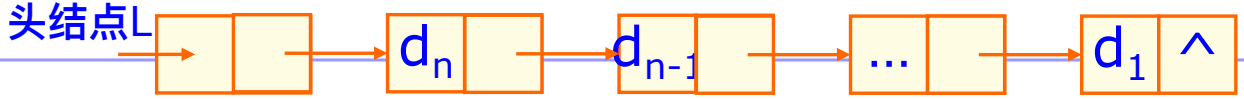



```
Status ListDelete_L(LinkList &L, int i, ElemType &e) {
  p = L; j = 0;
  while(p->next && j<i-1) { p = p->next; ++j; }
  if(!(p->next) || j>i-1) return ERROR;
  q = p->next; p->next = q->next; e = q->data;
 free(q);
  return OK;
}
```

- 算法评价：$T(n) = O(n)$

单链表的动态建立

- 动态建立单链表算法：从表尾到表头逆向建立单链表




```
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;
    }
}
```

- 算法评价：$T(n) = O(n)$

链表的合并/连接

- 把两个链表La和Lb合并
  - 若简单的合并两个链表，只需将第一个链表未尾元素的空指针改变指向第二个链表的头元素即可。
  - 若要合并两个有序表?
    - 算法见教材p31　MergeList_L
- 特点：时间复杂性同一般顺序表，但空间需求减少。

单链表特点

- 它是一种动态结构，整个可用存储空间为多个链表共用
- 每个链表占用的空间不需预先分配，应需即时生成
  - 建立线性表的链式存储结构的过程就是一个动态生成链表的过程。即从“空表”的初始状态起，依次建立各元素的结点，并逐个插入到链表中。
- 指针占用额外存储空间
- 不能随机存取，查找速度慢？

静态链表

- 在一些无指针类型的高级语言中使用
- 通常使用一如下结构的数组

```
typedef struct {
    ElemType        Data;
    int             cur;
} SLinkList[MAXSIZE];
```

称作静态链表

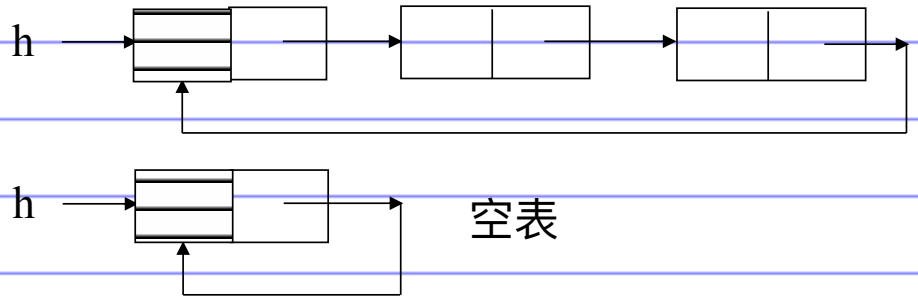
- 与顺序表用数组存储相比，在插入或删除时无需移元素

0		1
1	ZHAO	2
2	QIAN	3
3	SUN	4
4	LI	5
5	ZHOU	6
6	WU	7
7	ZHENG	8
8	WANG	0
9		
10		

0		1
1	ZHAO	2
2	QIAN	3
3	SUN	4
4	LI	9
5	ZHOU	6
6	WU	8
7	ZHENG	8
8	WANG	0
9	[illegible]	5
10		

循环链表(circular linked list)

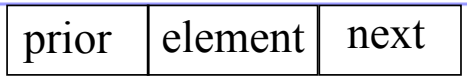
- 循环链表是表中最后一个结点的指针指向头结点，使链表构成环状
- 特点：从表中任一结点出发均可找到表中其他结点，提高查找效率
- 操作与单链表基本一致,循环条件不同
  - 单链表p或p->link=NULL
  - 循环链表p或p->link=H



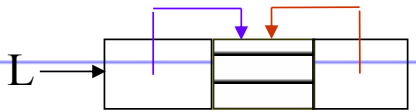
双向链表(bi-linked list)

- 单链表具有单向性的缺点，只能找后继，不能找前驱。为查找一个结点总得从头结点处开始。双向链表是在结点中增加一个指向前驱的指针：

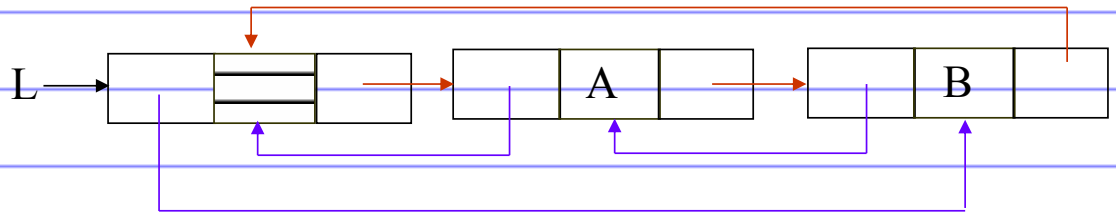
```
typedef struct DuLNode {
    ElemType data;
    struct DuLNode *prior,*next;
}DuLNode; *DuLinkList;
```



空双向循环链表

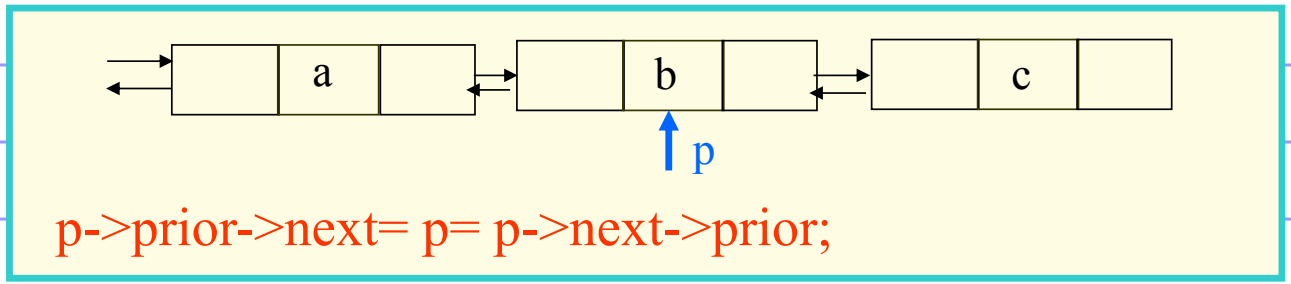


非空双向循环链表



双向链表特征

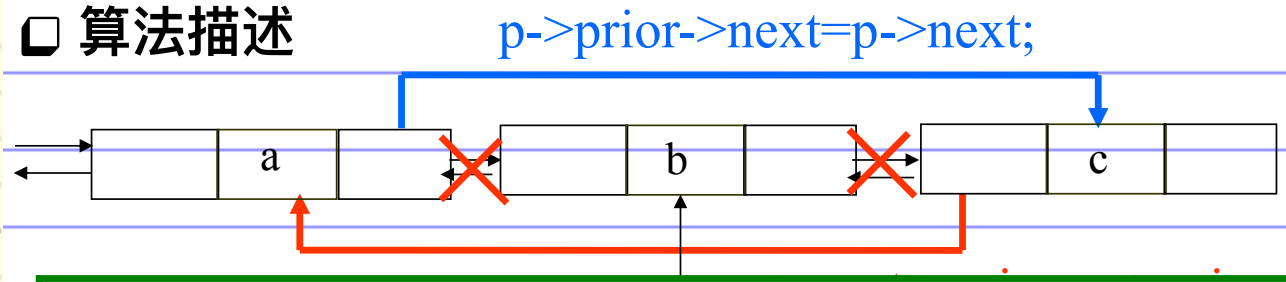
- 其前驱的后继就是它自己



- 双向链表的操作：
  - 插入
  - 删除

双向链表的删除操作

- 算法描述

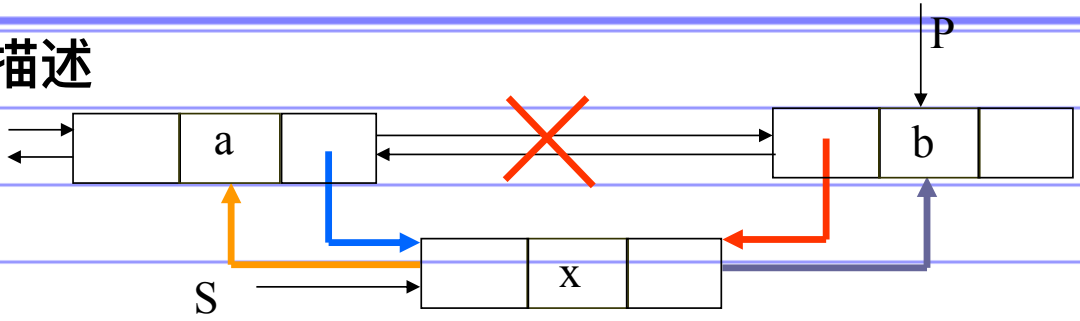



```
Status ListDelete_DuL(DuLinkList &L, int i, ElemType &e) {
    if(!(p = GetElemP_DuL(L, i)))
        return ERROR;
    e = p_data;
    p->prior->next = p->next;
    p->next->prior = p->prior;
    free(p);  return OK;
}
```

- 算法评价：$T(n)=O(n)$

双向链表的插入操作

- 算法描述




```
Status ListInsert_DuL(DuLinkList &L, int i, ElemType e) {
  if(!(p = GetElemP_DuL(L, i))) return ERROR;
  if(!(s = (DuLinkList)malloc(sizeof(DuLNode)))) return ERROR;
  s->data = e;
  s->prior = p->prior;   p->prior->next = s;
  s->next = p;     p->prior = s;
  return OK;
}
```

- 算法评价：$T(n)=O(n)$

2.4 线性表的应用举例

一元多项式的表示及相加

$$P_n(x) = P_0 + P_1x + P_2x^2 + \cdots\cdots + P_nx^n$$

可用线性表P表示 $P = (P_0, P_1, P_2, \cdots\cdots, P_n)$

但对S(x)这样的多项式浪费空间 $S(x) = 1 + 3x^{1000} + 2x^{20000}$

一般 $P_n(x) = P_1x^{e1} + P_2x^{e2} + \cdots\cdots + P_mx^{em}$

其中 $0 \le e1 \le e2 \cdots\cdots \le em$ （P_i为非零系数）

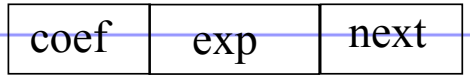
用数据域含两个数据项的线性表表示

$((P_1, e1), (P_2, e2), \cdots\cdots (P_m, em))$

其存储结构可以用顺序存储结构，也可以用单链表

单链表的结点定义

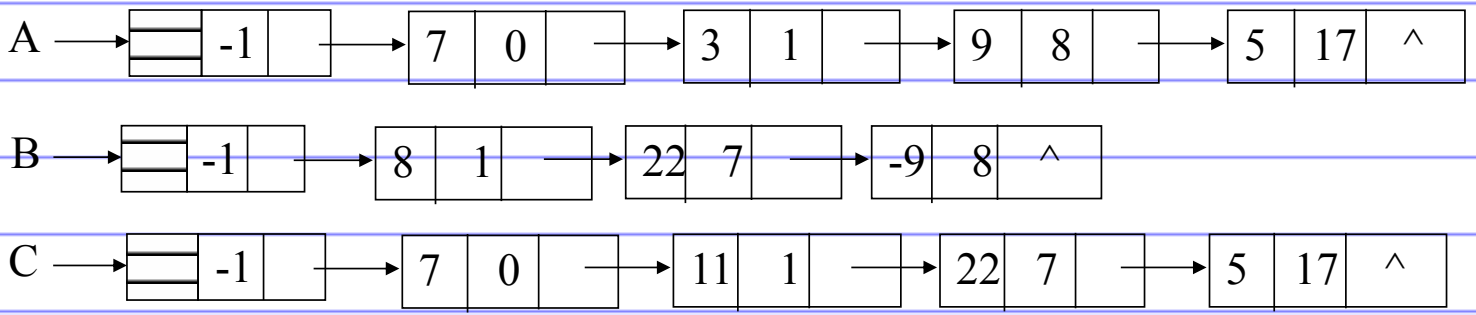
```
typedef   struct node
{   float coef;
    int exp;
    struct node  *next;
}Node;
```



$$A(x) = 7 + 3x + 9x^8 + 5x^{17}$$

$$B(x) = 8x + 22x^7 - 9x^8$$

$$C(x) = A(x) + B(x) = 7 + 11x + 22x^7 + 5x^{17}$$



运算规则

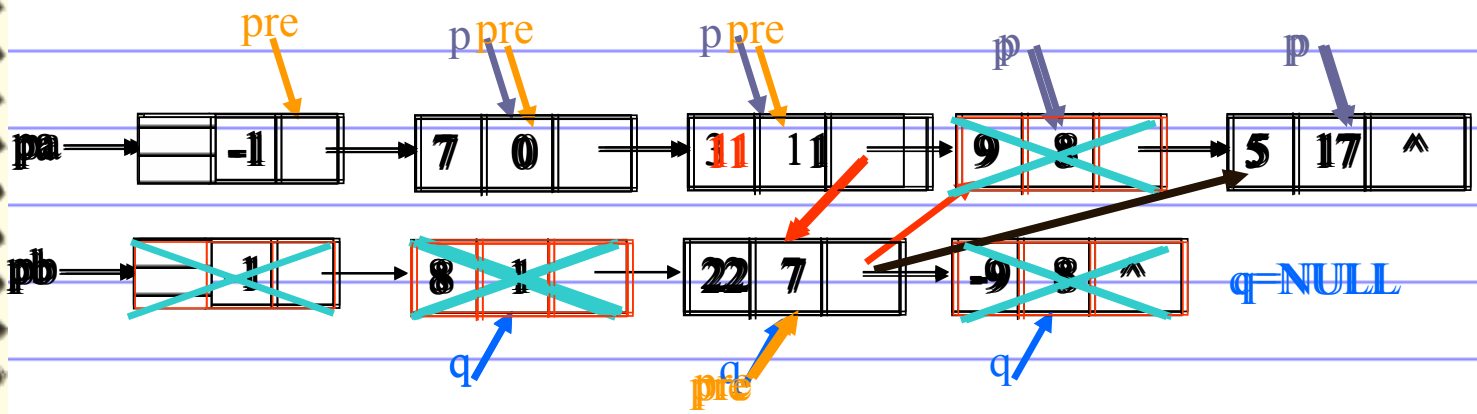
- 设p,q分别指向A,B中某一结点，p,q初值是第一结点

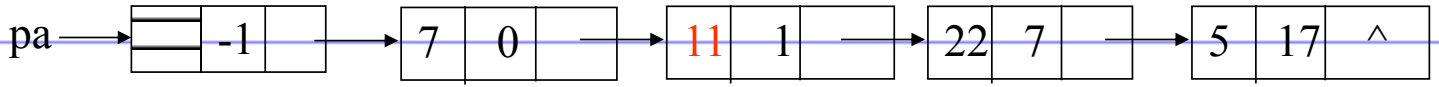
比较 p->exp与q->exp

- p->exp < q->exp: p结点是和多项式中的一项 p后移,q不动
- p->exp > q->exp: q结点是和多项式中的一项 将q插在p之前,q后移,p不动
- p->exp = q->exp: 系数相加
  - 0：从A表中删去p, 释放p,q，p,q后移
  - ≠0：修改p系数域, 释放q，p,q后移

算法描述

- 算法描述


pre
p
pre
p
pre
p
p
pa
-1
7
0
11
1
9
5
17
^
pb
1
8
1
22
7
-9
^
q=NULL
q
pre
q
q


pa
-1
7
0
11
1
22
7
5
17
^