第二章 线性表

- 2.1 线性表的定义和基本操作
- 2.2 线性表的顺序存储结构
- 2.3 线性表的链接存储结构
- 2.4 复杂性分析

2.1 线性表的定义和基本操作

线性表定义：一个线性表是由零个或多个具有**相同类型**的结点组成的**有序集合**。用（a_0，a_1，…，a_{n-1}）来表示一个线性表。当n>0时，a_0称为表的**始结点**，a_{n-1}称为表的**终结点**，a_i为a_{i+1}的**前趋结点**，a_{i+1}为a_i的**后继结点**；当n=0时，线性表中有零个结点，称为**空表**。

线性表的逻辑结构：线性结构

线性表的形式定义

◆ 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)。

数组

数据项

记录

可用结构表示

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

❑ 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。

线性表的操作

（1）随机存取：存取下标为k的结点。

（2）插入：在下标为k的结点前（或后）

插入一 个新结点

（3）删除：删除下标为k的结点。

（4）查找：寻觅具有特定域值的结点。

（5）归并、分拆、复制、计数、排序。

线性表的操作

- 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)

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

```
void union(List &La, List Lb) {
  // 将所有在线性表 Lb 中但不在 La 中的数据元素插入到 La 中
  La_len = ListLength(La);  Lb_len = ListLength(Lb); // 求线性表的长度
  for (i = 1; i <= Lb_len; i++) {
    GetElem(Lb, i, e);                    // 取 Lb 中第 i 个数据元素赋给 e
    if (!LocateElem(La, e, equal)) ListInsert(La, ++La_len①, e);
                                  // La 中不存在和 e 相同的数据元素,则插入之
  }
}// union
```

- 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

```
void MergeList(List La, List Lb, List &Lc) {
    // 已知线性表 La 和 Lb 中的数据元素按值非递减排列。
    // 归并 La 和 Lb 得到新的线性表 Lc,Lc 的数据元素也按值非递减排列。
    InitList(Lc);
    i = j = 1;k = 0;
    La_len = ListLength(La); Lb_len = ListLength(Lb);
    while ((i <= La_len) && (j <= Lb_len)) {// La 和 Lb 均非空
      GetElem(La, i, ai);  GetElem(Lb, j, bj);
      if (ai <= bj)  {ListInsert(Lc, ++k, ai); ++i; }
      else {ListInsert(Lc, ++k, bj); ++j; }
    }
    while (i <= La_len) {
        GetElem(La, i++, ai); ListInsert(Lc, ++k, ai);
    }
    while (j <= Lb_len) {
        GetElem(Lb, j++, bj); ListInsert(Lc, ++k, bj);
    }
} // MergeList
```

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

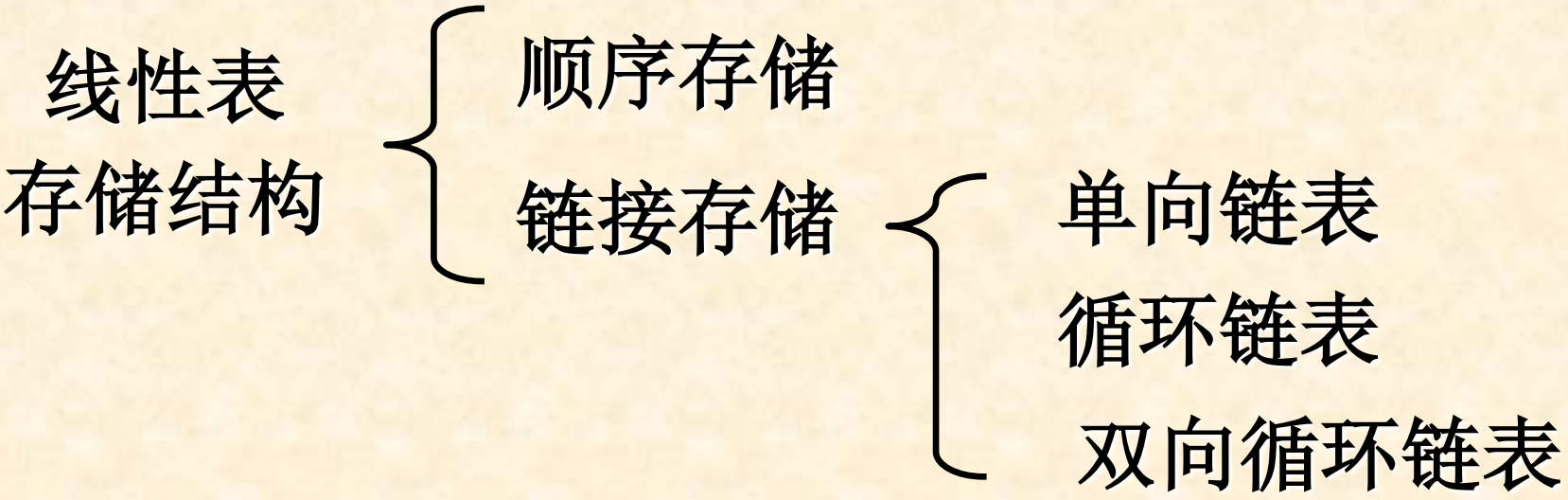
(2) La、Lb无序呢？

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 线性表的存储结构

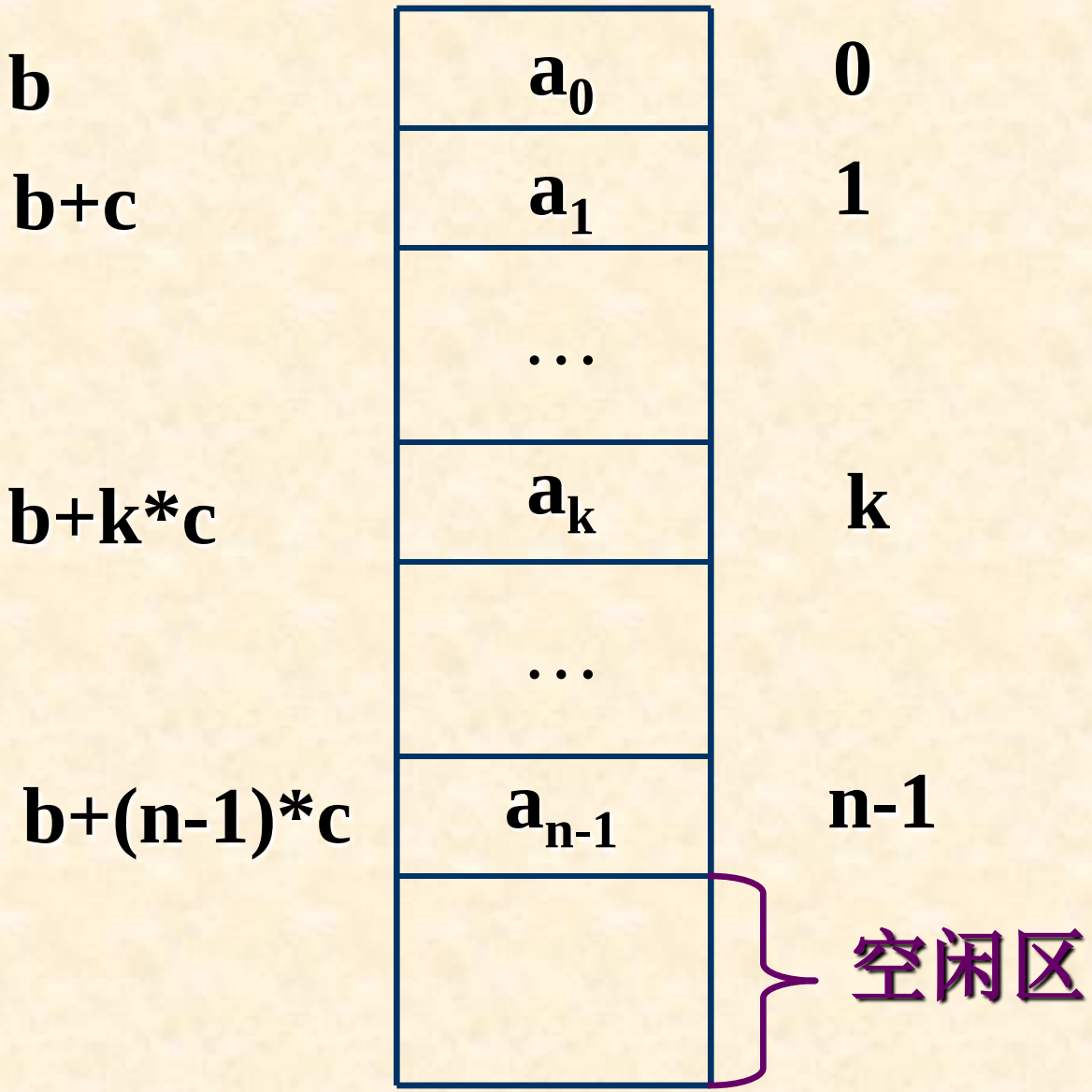


2.2.1 顺序存储结构

顺序存储：用一组连续的存储空间依次存储线性表的元素。

实现顺序存储的最有效方法是使用一维数组。

[例]：线性表（a_0，a_1，…，a_{n-1}）。可以使用一个数组A[n]来存放此线性表。



Loc(a[k])= Loc (a[0]) + k*c

顺序存储的线性表的基本运算

1、插入

[例] 在顺序表（12,13,21,24,28,30,42,77）中，插入元素 25。

问题：此时，线性表的逻辑结构发生什么变化？

位置关系发生变化

长度增1


序号
元素
0
12
1
13
2
21
插入25
3
24
4
28
5
30
6
42
7
77
序号
元素
0
12
1
13
2
21
3
24
4
25
5
28
6
30
7
42
8
77

时间复杂性分析：

插入操作的基本运算是：

元素移动

D_n中有多少种可能的输入？

n种（n个位置可以发生插入）

设每种输入发生的频率相等：1/n

则期望复杂性为：

$$E(n)=((n-1)+\cdots\cdots+1+0)/n=(n-1)/2$$

2、删除

[例] 在顺序表（12, 13, 21, 24, 28, 30, 42, 77）中，删除元素 24。

问题：此时，线性表的逻辑结构发生什么变化？

位置关系发生变化

长度减1


序号 元素
0 12
1 13
2 21
3 24
4 28
5 30
6 42
7 77
删除24
序号 元素
0 12
1 13
2 21
3 28
4 30
5 42
6 77

时间复杂性分析：

删除操作的基本运算是：

元素移动

D_n中有多少种可能的输入？

n 种（n个位置可以发生删除）

设每种输入发生的频率相等：1/n

则期望复杂性为：

$$E(n)=(n-1)/n+\ldots\ldots+1/n+0/n$$

$$=（n-1）/2$$

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(ai+1)=LOC(ai)+L
  - LOC(ai)=LOC(a1)+(i-1)*L

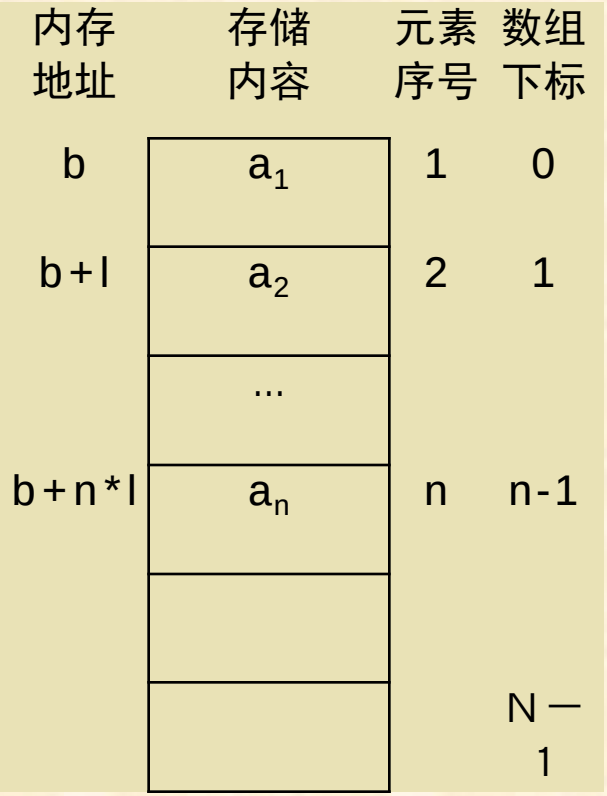
  其中：

  L — 一个元素占用的存储单元个数

  LOC(ai) — 线性表第i个元素的地址
- LOC(a1)是线性表第一个元素a1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
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
- 插入一个元素
- 删除一个元素
- 查找/定位一个元素
- 合并操作
- 顺序表的特点

顺序表定义及初始化操作

```
// - - - - - 线性表的动态分配顺序存储结构 - - - - -
#define  LIST_INIT_SIZE    100   // 线性表存储空间的初始分配量
#define  LISTINCREMENT      10   // 线性表存储空间的分配增量
typedef  struct {
    ElemType  * elem;          // 存储空间基址
    int       length;          // 当前长度
    int       listsize;        // 当前分配的存储容量(以 sizeof(ElemType)为单位)
}SqList;
```

```
Status InitList_Sq(SqList &L) {
    // 构造一个空的线性表 L。
    L.elem = (ElemType * )malloc(LIST_INIT_SIZE * sizeof(ElemType));
    if (! L.elem) exit(OVERFLOW);      // 存储分配失败
    L.length = 0;                      // 空表长度为 0
    L.listsize = LIST_INIT_SIZE;       // 初始存储容量
    return OK;
}// InitList_Sq
```

算法 2.3

顺序表插入操作

```
Status ListInsert_Sq(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;
}// ListInsert_Sq
```

算法 2.4

顺序表删除操作

```
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;
    return OK;
}// ListDelete_Sq
```

顺序表查找操作

```
int LocateElem_Sq(SqList L, ElemType e,
                  Status ( * compare)(ElemType, ElemType)) {
    // 在顺序线性表 L 中查找第 1 个值与 e 满足 compare()的元素的位序

    // 若找到,则返回其在 L 中的位序,否则返回 0
    i = 1;                  // i 的初值为第 1 个元素的位序
    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;
    else return 0;
}// LocateElem_Sq
```

算法 2.6

顺序表合并操作

```
void MergeList_Sq(SqList La, SqList Lb, SqList &Lc) {
    // 已知顺序线性表 La 和 Lb 的元素按值非递减排列
    // 归并 La 和 Lb 得到新的顺序线性表 Lc,Lc 的元素也按值非递减排列
    pa = La.elem; pb = Lb.elem;
    Lc.listsize = Lc.length = La.length + Lb.length;
    pc = Lc.elem = (ElemType * )malloc(Lc.listsize * sizeof(ElemType));
    if (!Lc.elem)exit(OVERFLOW);  // 存储分配失败
    pa_last = La.elem + La.length - 1;
    pb_last = Lb.elem + Lb.length - 1;
    while (pa <= pa_last && pb <= pb_last) {     // 归并
      if (*pa <= *pb) *pc++ = *pa++;
      else *pc++ = *pb++;
    }
    while (pa <= pa_last) *pc++ = *pa++;          // 插入 La 的剩余元素
    while (pb <= pb_last) *pc++ = *pb++;          // 插入 Lb 的剩余元素
}  // MergeList_Sq
```

算法 2.7

● 结论：

线性表的顺序存储结构优点：

① 易于实现，随机存取速度快；

② 逻辑相邻，物理相邻；

③ 存储空间使用紧凑。

缺点：1）插入和删除结点时间复杂性高，需移动一批元素；

2）空间利用率低：预先分配空间需按最大空间分配；

3）表容量难以扩充。

问题：由于线性表中元素数目可变，定义数组时要需要考虑什么？

定义足够大的静态数组。

使用动态数组。

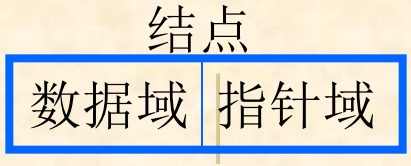
第二章 线性表

- 2.1 线性表的定义和基本操作
- 2.2 线性表的顺序存储结构
- 2.3 线性表的链接存储结构
- 2.4 复杂性分析

- 链接存储：用任意一组存储单元存储线性表，一个存储单元除包含结点数据(或信息)字段的值，还必须存放其逻辑相邻结点(前驱或后继结点)的地址信息，即指针字段。
- 随结点指针域的不同，链表主要有三种实现方式：单链表、循环链表和双向链表。

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)

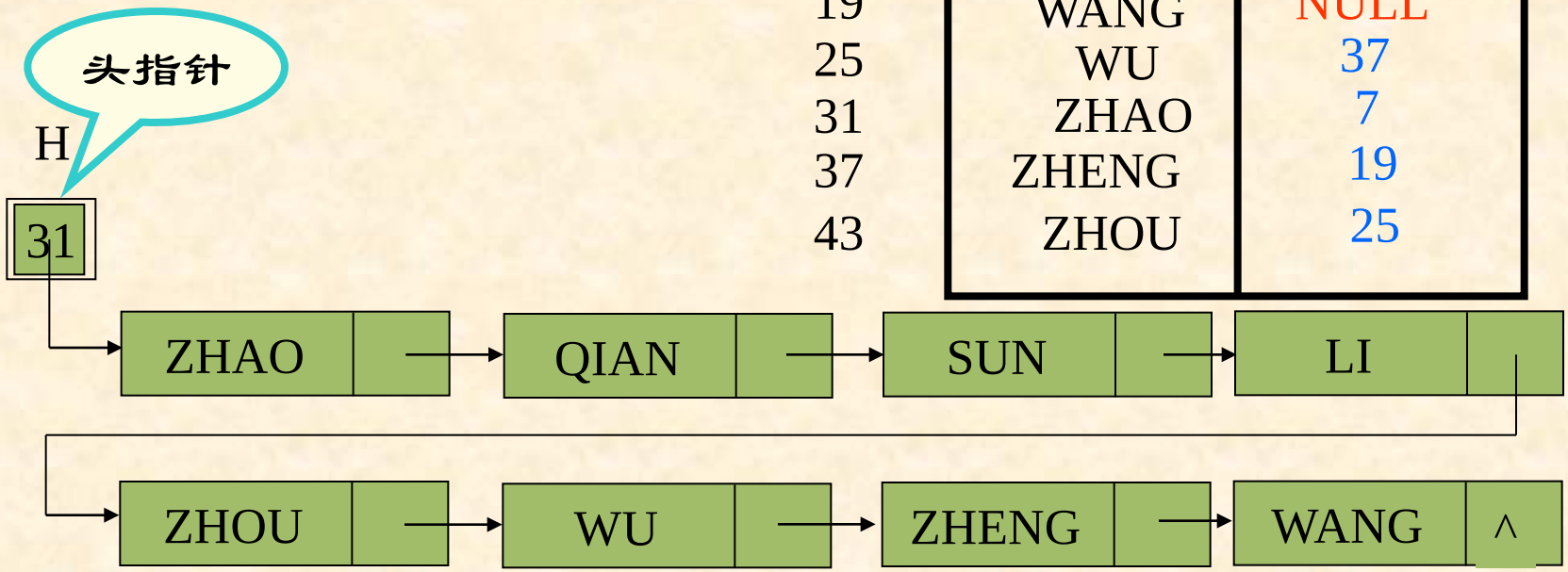



```
struct Linkage {
        DataType Data;
        struct Linkage *Next;
};
```

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



2.3.1 线性表的链接存储结构

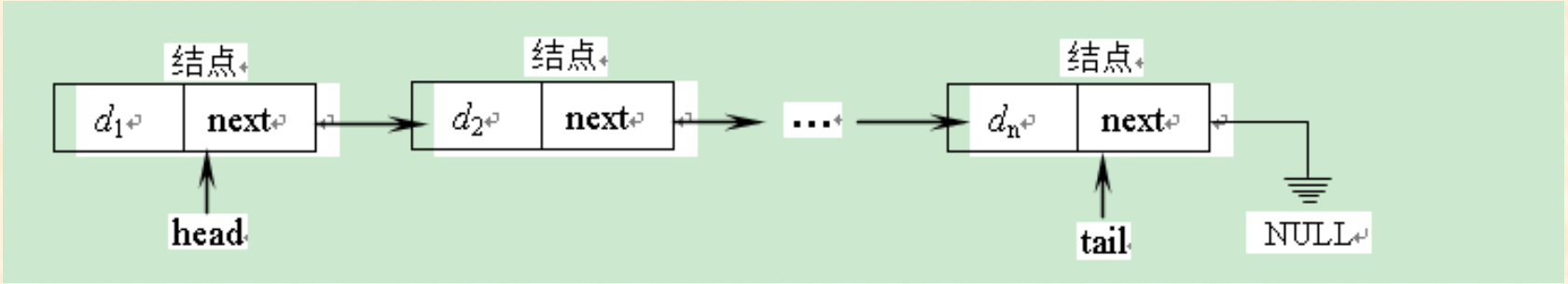
————单链表

1、单链表的定义

2、单链表的特点

3、单链表的实现

◆单链表的**结点结构**：



◆链表的第一个结点被称为头结点(也称为表头)，指向头结点的指针被称为头指针(head).

◆链表的最后一个结点被称为尾结点(也称为表尾)，指向尾结点的指针被称为尾指针(tail).

- 为了对表头结点插入、删除等操作的方便，通常在表的前端增加一个特殊的表头结点，称其为哨位结点

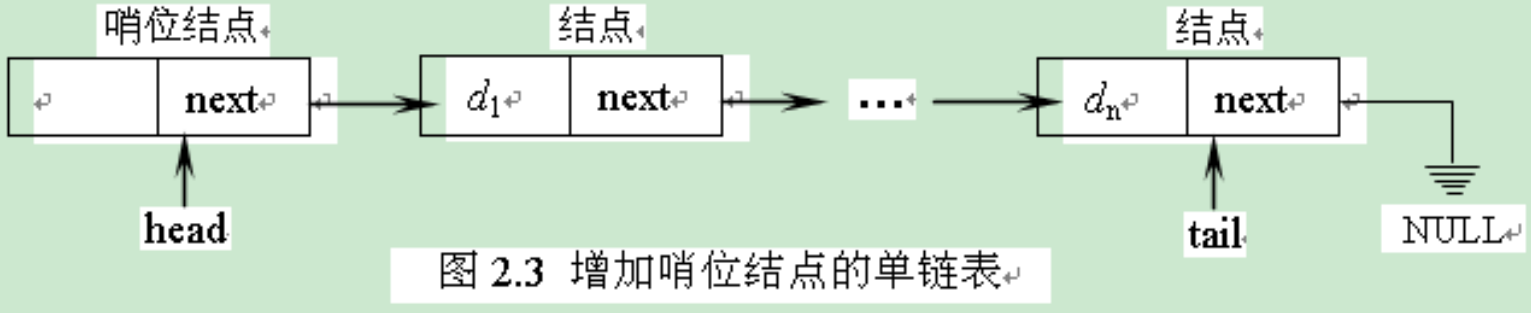


图 2.3 增加哨位结点的单链表

2.3.1 线性表的链接存储结构

————单链表

1、单链表的定义

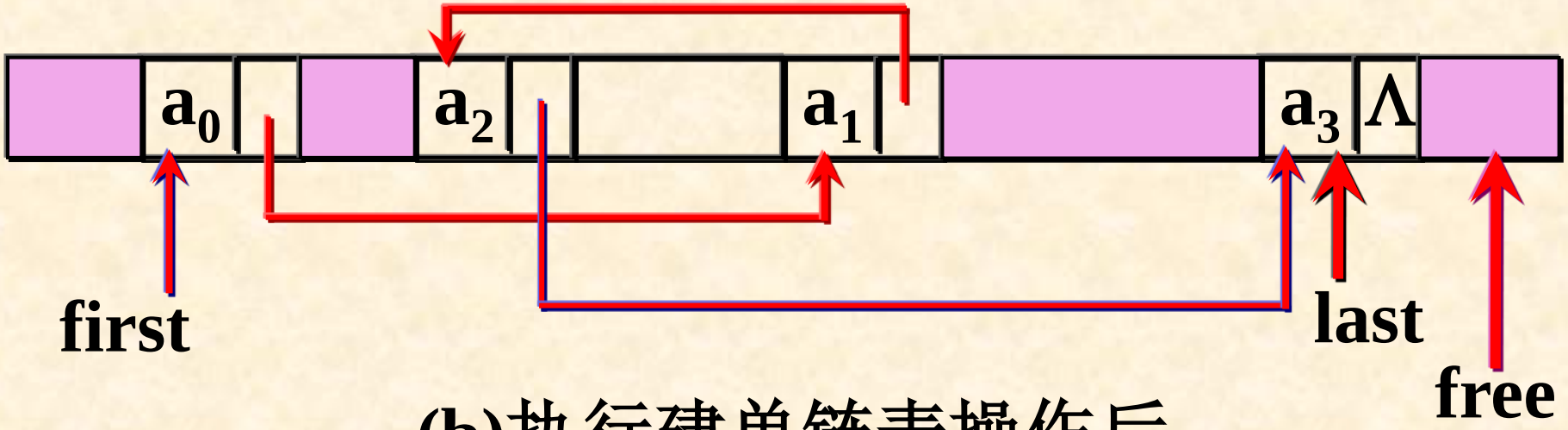
2、单链表的特点

3、单链表的实现

特点:逻辑上相邻的节点在物理上不必相邻

单链表的存储映像

(a)可用存储空间



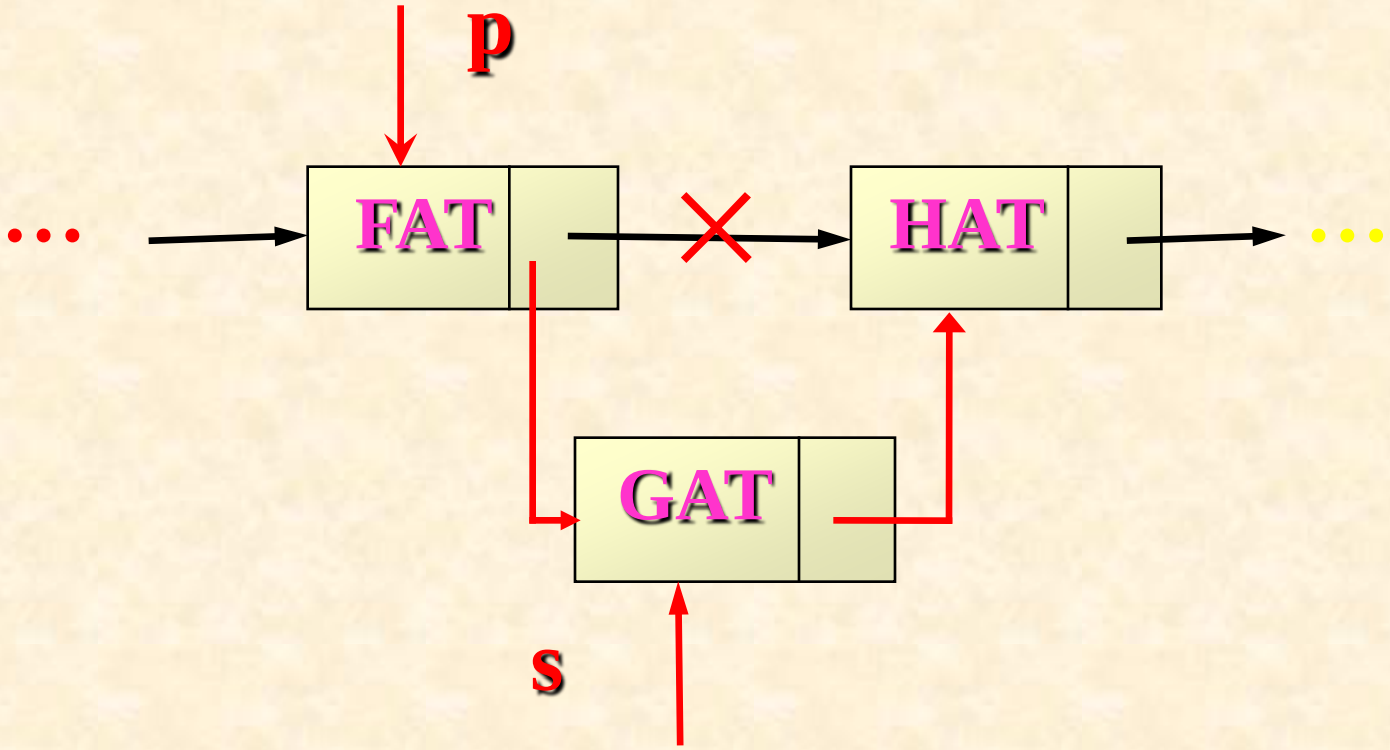
(b)执行建单链表操作后

在链表中，插入或删除一个结点，只需改变一个或两个相关结点的指针，不对其它结点产生影响。

- 插入：

next(s) ← next(p)

next(p) ← s

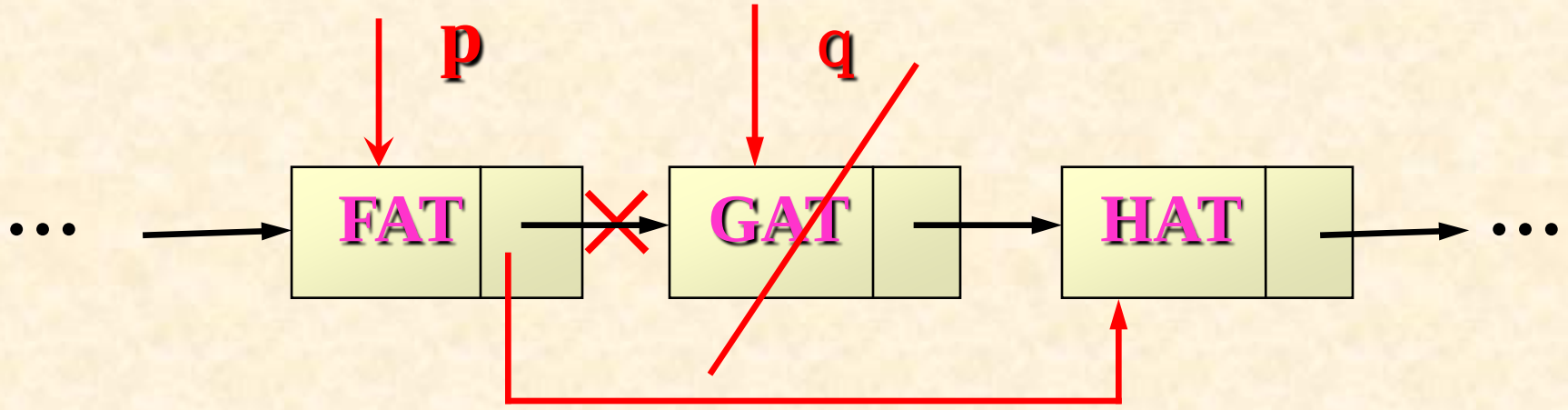


在链表中，插入或删除一个结点，只需改变一个或两个相关结点的指针，不对其它结点产生影响。

q ← next (p). next (p) ← next (q) .

AVAIL ⇐ q .

- 删除：



● 单链表的特性：

① 利用链接域实现线性表元素的逻辑关系。

② 单链表有头结点、尾结点、头指针。

● 单链表的优点：

① 插入、删除方便；

② 空间利用率高：结点可不连续存储，易于扩充

2.3.1 线性表的链接存储结构

————单链表

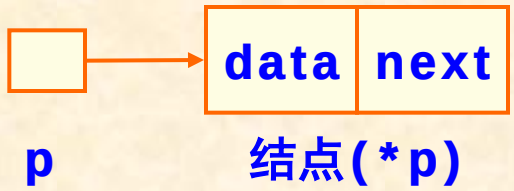
1、单链表的定义

2、单链表的特点

3、单链表的实现

单链表的实现

```
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

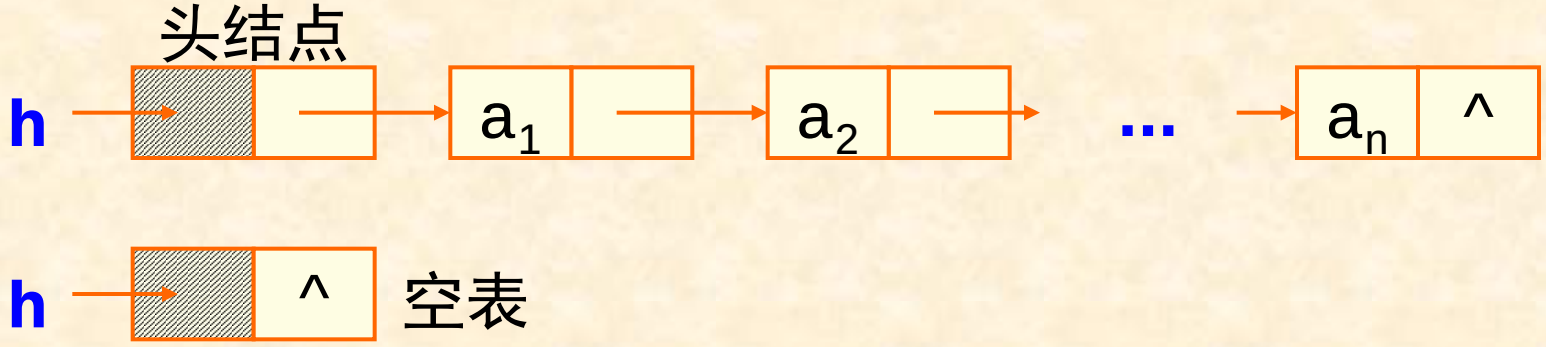
生成一个Node型新结点：

```
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;
        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 $T(n) = O(n)$

- 获取第i个位置的元素赋给e:

```
Status GetElem_L(LinkList L, int i, ElemType &e){
    LinkList * 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 $T(n) = O(n)$

- 获取第i个位置的元素赋给e:

```
Status GetElem_L(LinkList L, int i, ElemType &e){
    LinkList * 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
```

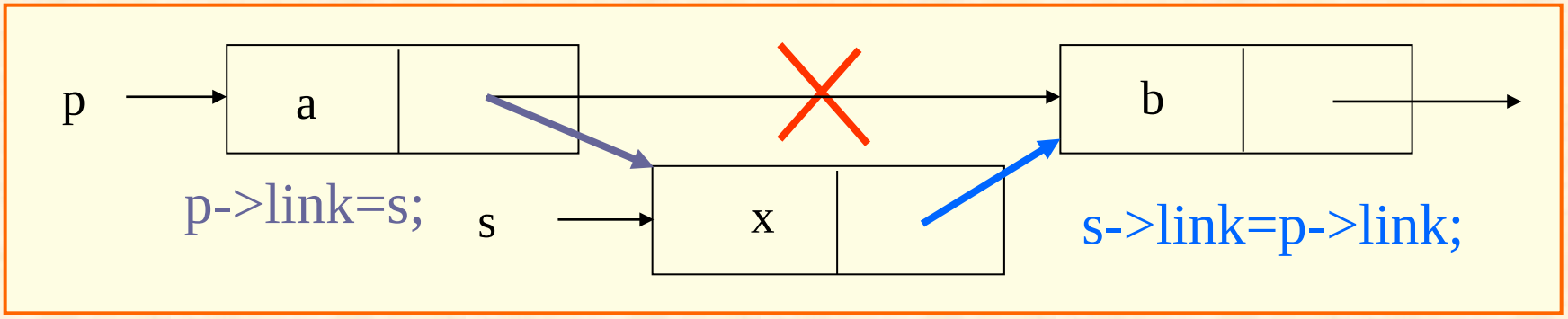
- 算法评价:

  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:

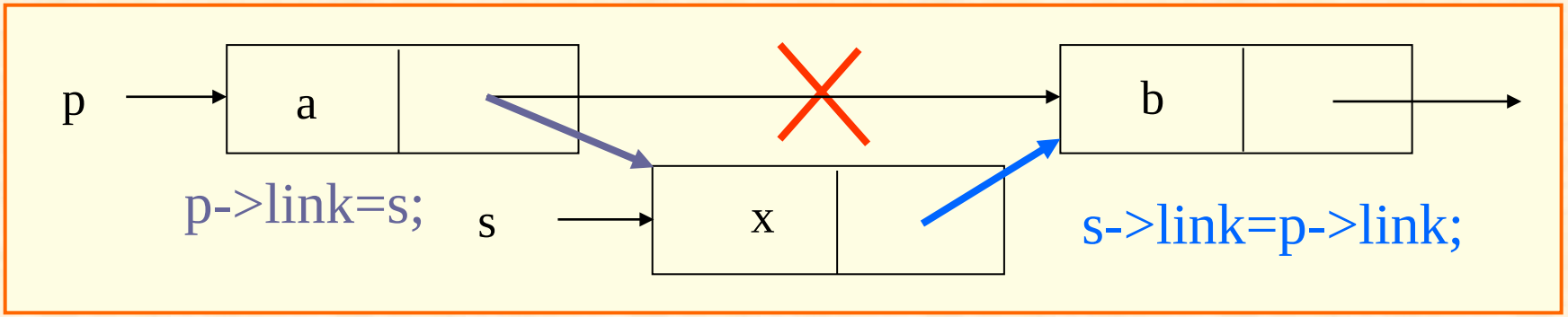
```
Status ListInsert_L(LinkList &L, int i, ElemType e) {
   LinkList * p = L; j = 0;
   while(p && j<i-1) {p = p->next; ++j;}
   if(!p || j>i-1) return ERROR;
   s = (LinkList)malloc(sizeof(LNode));
   s->data = e; s->next = p->next; p->next = s;
   return OK;
}
```



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) {
   LinkList * 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;
}
```



单链表中插入结点

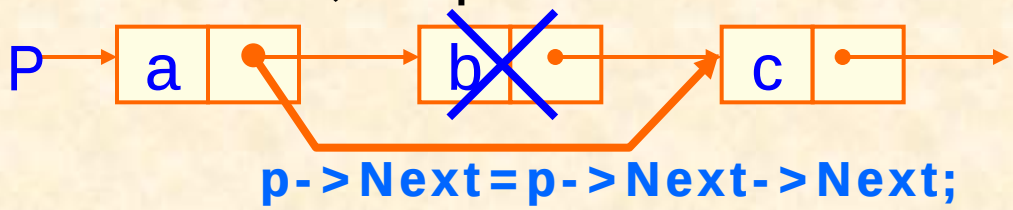
- 在第i个位置之前插入元素e:

```
Status ListInsert_L(LinkList &L, int i, ElemType e) {
   LinkList * 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;
}
```

- 算法评价: $T(n) = O(n)$
- 若给出的不是结点位置，而是结点数据，如何在给定的数据前/后插入数据X呢 

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;
}
```

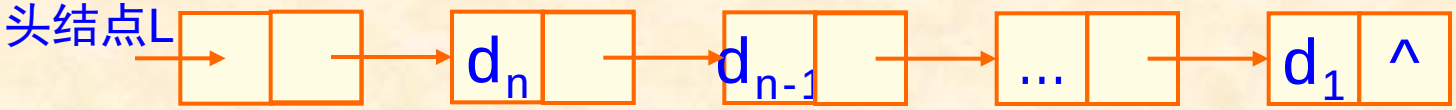
- 算法评价：$T(n) = O(n)$

插入、删除操作的时间复杂性分析

- 插入、删除操作的最好情况的时间复杂度为$O(1)$；
- 插入、删除操作最坏情况下的时间复杂度为$O(n)$；
- 平均情况下，时间复杂度也是$O(n)$。

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。

单链表实现的特点

- 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?

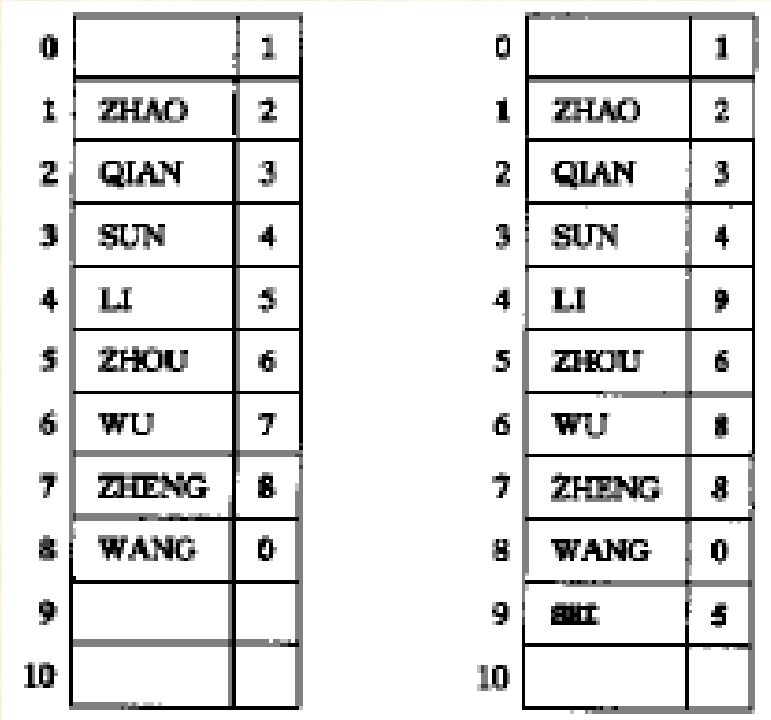
静态链表

- 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		

2.3 线性表的链接存储结构

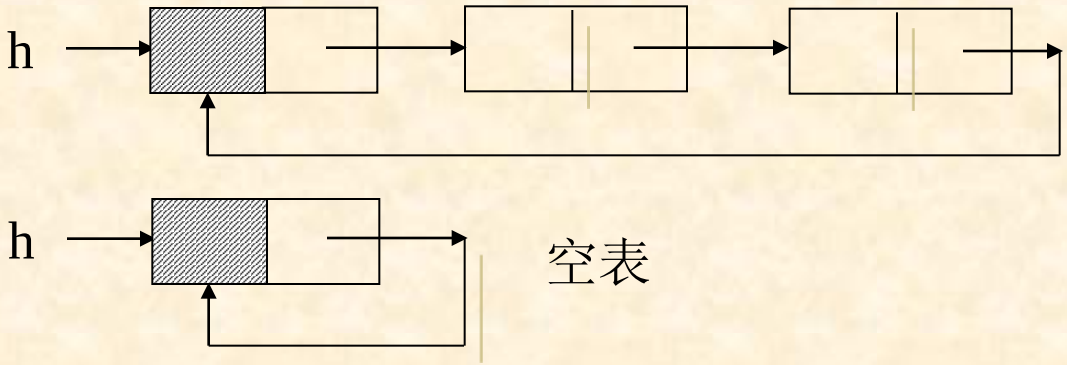
2.3.1 单链表

2.3.2 循环链表

2.3.3 双向链表

循环链表(circular linked list)

- 把链接结构“循环化”，即是把表中最后一个结点的指针（next域）指向头结点，使链表构成环状，而不是存放空指针NULL，称为循环链表；
- 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



2.3 线性表的链接存储结构

2.3.1 单链表

2.3.2 循环链表

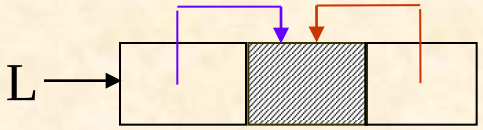
2.3.3 双向链表

双向链表(bi-linked list)

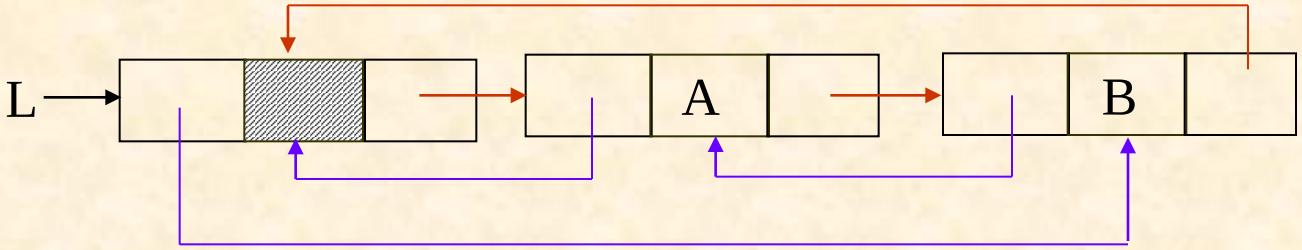
- 单链表具有单向性的缺点，只能找后继，不能找前驱。为查找一个结点总得从头结点处开始。双向链表是在结点中增加一个指向前驱的指针：

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

空双向循环链表

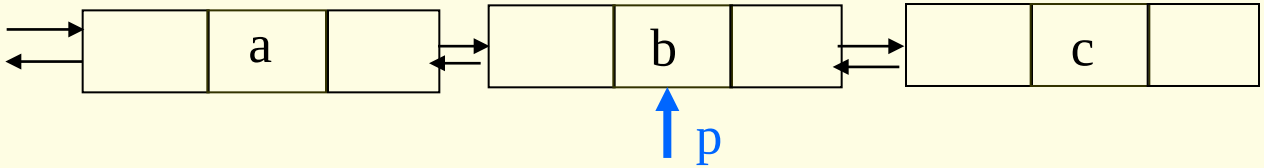


非空双向循环链表



双向链表特征

- 其前驱的后继就是它自己

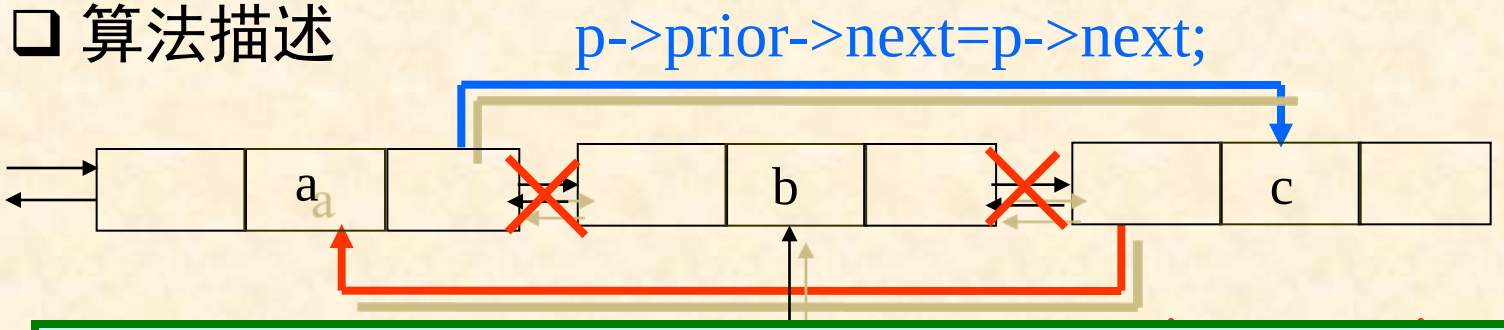


p->prior->next= p= p->next->prior;

- 双向链表的操作：
  - 插入
  - 删除

双向链表的删除操作

- 算法描述

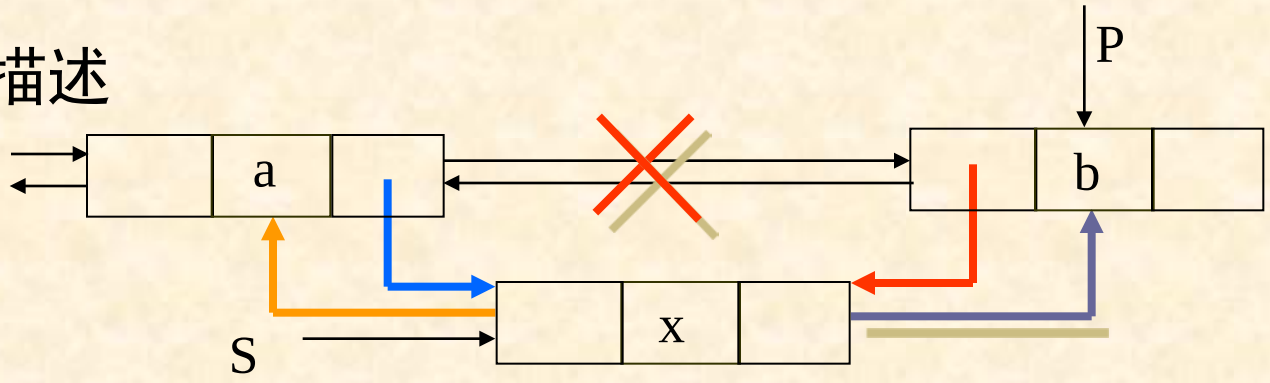



```
Status ListDelete_DuL(DuLinkList &L, int i, ElemType &e) {
    if(!(p = GetElemP_DuL(L, i)))
        return ERROR;
    e = p->data;
    p->prior->next = p->next;
    p->next->prior = p->prior;
    free(p);  return OK;
}
```

- 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;
  s->next = p;
  p->prior->next = s;
  p->prior = s;
  return OK;
}
```

❑ 算法评价：$T(n)=O(n)$

第二章 线性表、堆栈和队列

- 2.1 线性表的定义和基本操作
- 2.2 线性表的顺序存储结构
- 2.3 线性表的链接存储结构
- 2.4 复杂性分析

§2.4 顺序存储和链式存储的复杂性分析

1、空间效率的比较

- 顺序表所占用的空间来自于申请的数组空间，数组大小是事先确定的，当表中的元素较少时，顺序表中的很多空间处于闲置状态，造成了空间的浪费；
- 链表所占用的空间是根据需要动态申请的，不存在空间浪费问题，但链表需要在每个结点上附加一个指针，从而产生额外开销。

2、时间复杂性的比较

- 线性表的基本操作是存取、插入和删除。对于顺序表，随机存取是非常容易的，但是每插入或者删除一个元素，都需要移动若干元素。对于链表，无法实现随机存取，必须要从表头开始遍历链表，直到找到要存取的元素，但是链表的插入和删除操作却非常简便，只需要修改一个或者两个指针值。
- 当线性表经常需要进行插入、删除操作时，链表的时间复杂性较小，效率较高；当线性表经常需要存取，且存取操作比插入删除操作频繁的情况下，则是顺序表的时间复杂性较小，效率较高。

单链表应用:一元多项式及其相加

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

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

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

coef	Exp	Link

■**优点：**

- 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。

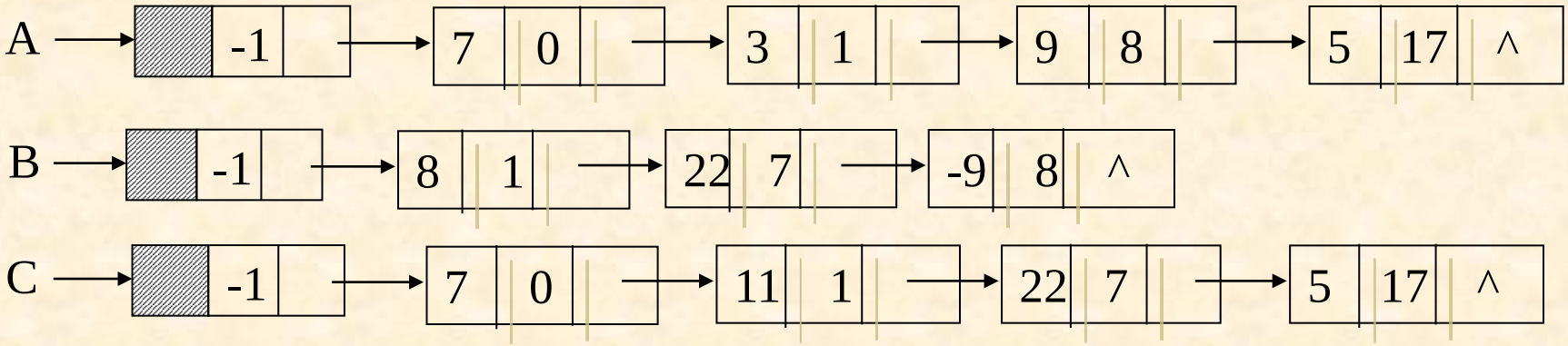
单链表的结点定义

```
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

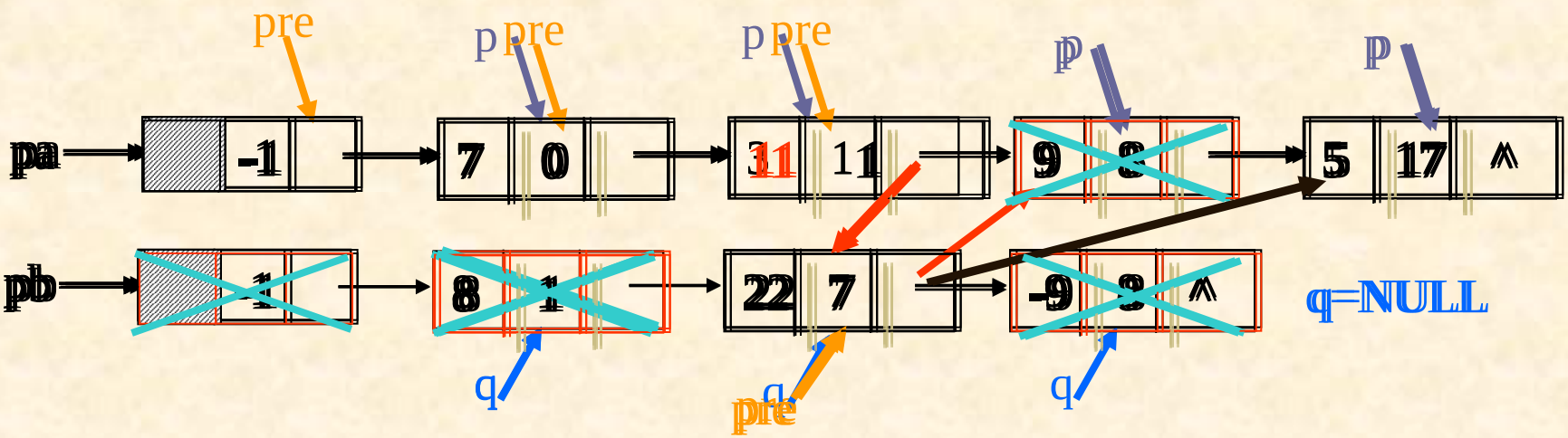
- 设p, q分别指向A,B中某一结点，p, q初值是第一结点
- 比较 p->exp与q->exp
  - p->exp < q->exp：p结点是和多项式中的一项，p后移，q不动
  - p->exp > q->exp：q结点是和多项式中的一项将q插在p之前, q后移, p不动
  - p->exp = q->exp：系数相加

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
8
5
17
^
pb
-1
8
1
22
7
-9
8
^
q=NULL
q
q
pre
q


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