数据结构与算法

Data Structure and Algorithms

西安交通大学自动化系

蔡忠闽 周亚东

第二章　线 性 表

第二章　线 性 表

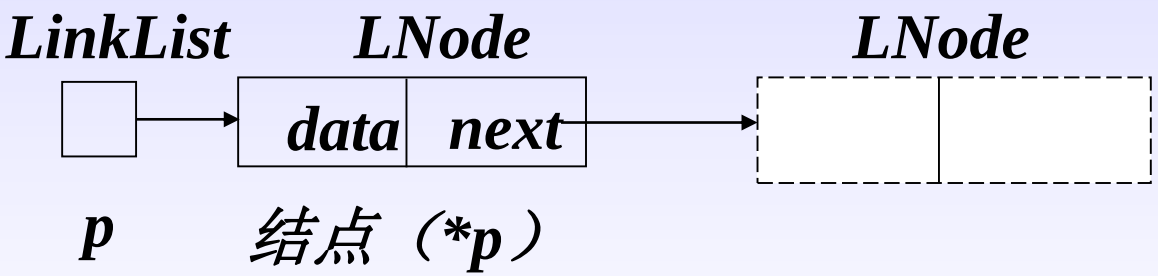
- 2.1 线性表的类型定义
- 2.2 线性表的顺序表示和实现
- 2.3 线性表的链式表示和实现
  - 2.3.1　线性链表
  - 2.3.2　循环链表
  - 2.3.3　双向链表
- 2.4 一元多项式的表示及相加

2.3 线性表的链式表示与实现

4. 单链表存储结构的实现

——用C语言中的“结构指针”来描述

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



注意结点p与结点a_i的区别

(*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

2.3 线性表的链式表示与实现

单链表特点：

- 它是一种动态结构，整个存储空间为多个链表共用
- 不需预先分配空间
- 指针占用额外存储空间
- 不能随机存取，查找速度慢

生成一个LNode型新结点：

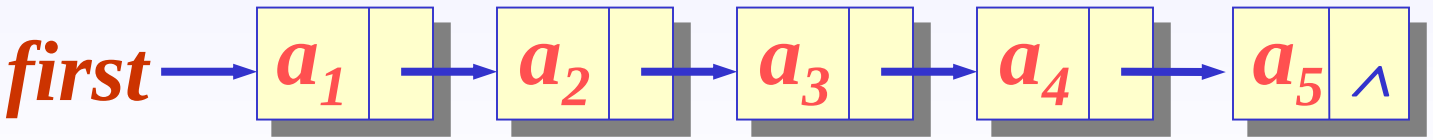
```
p=(LinkList)malloc(sizeof(LNode));
```

系统回收p结点： ***free(p)***

2.3 线性表的链式表示与实现

插入

- 第一种情况：在第一个结点前插入
- 第二种情况：在链表中间插入
- 第三种情况：在链表末尾插入

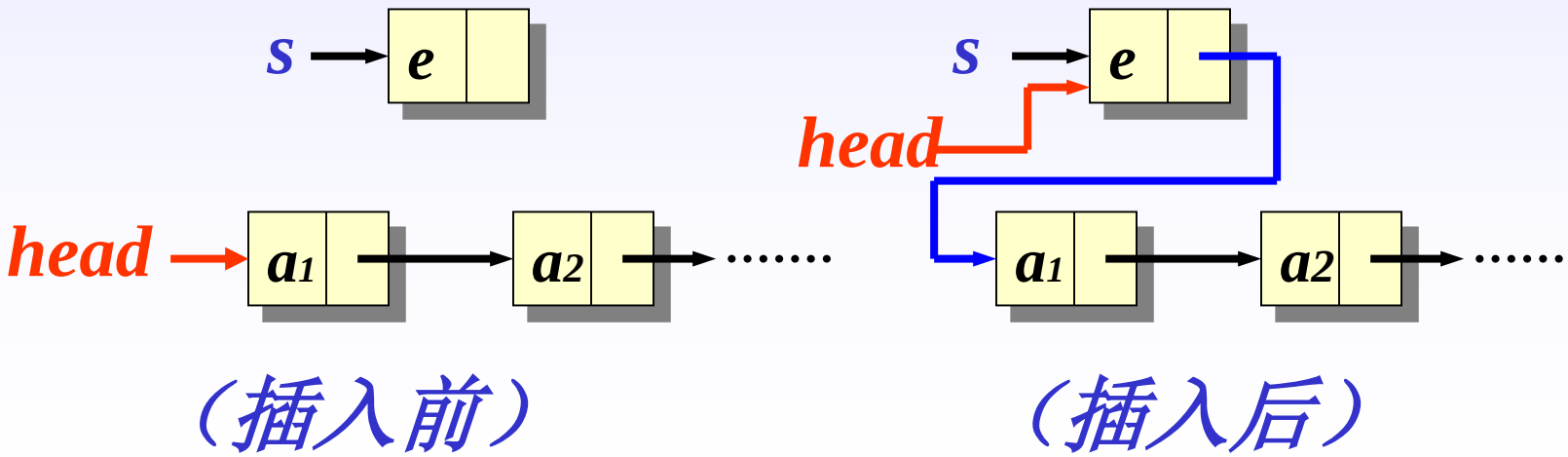


2.3 线性表的链式表示与实现

插入

- 第一种情况：在第一个结点前插入

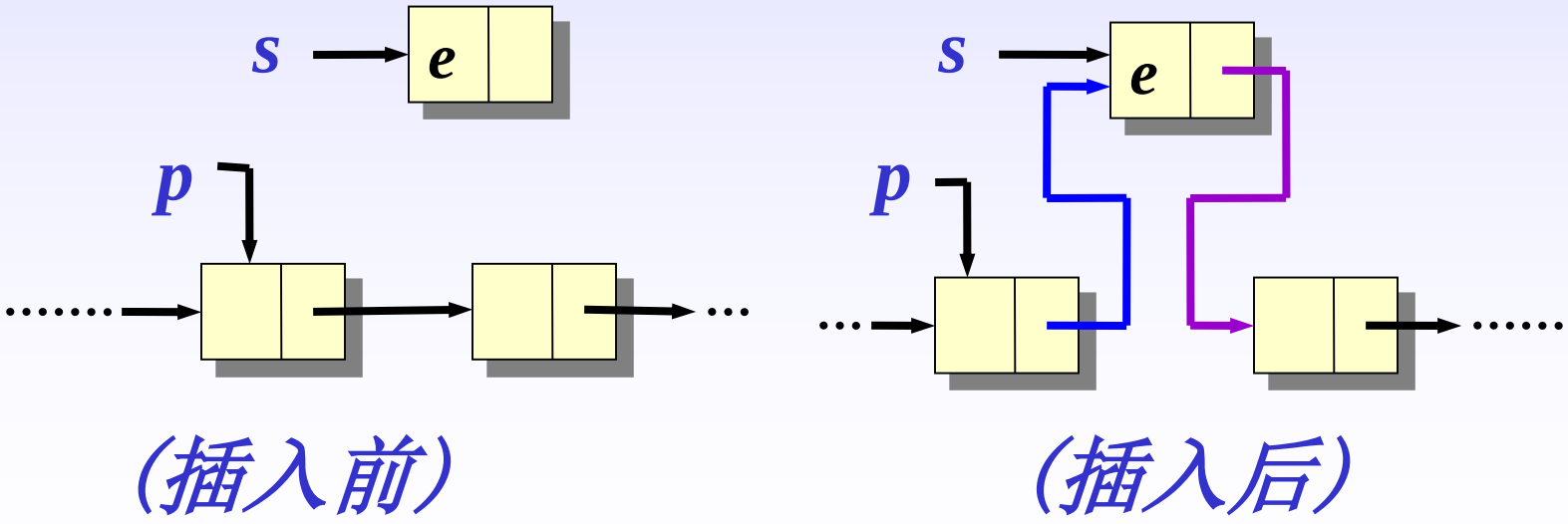
```
s->next = head ;
head = s;
```



2.3 线性表的链式表示与实现

- 第二种情况：在链表中间插入

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

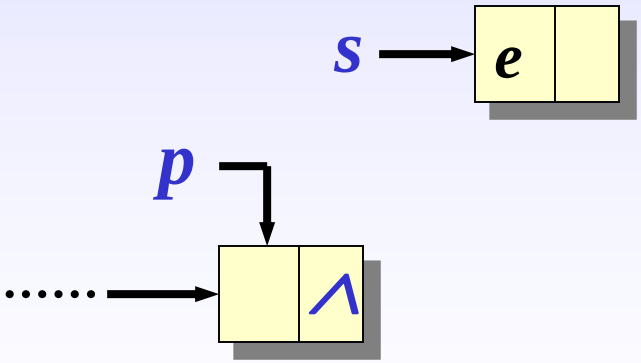


(插入前)　　(插入后)

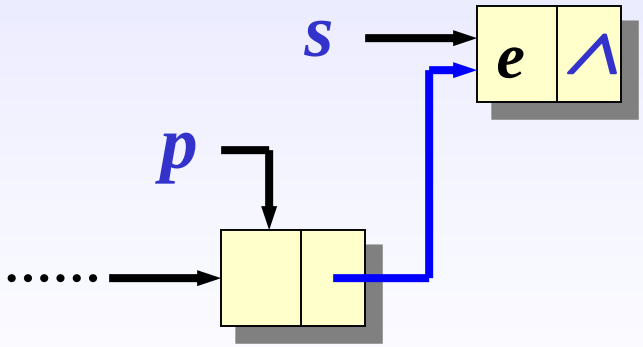
2.3 线性表的链式表示与实现

- 第三种情况：在链表末尾插入

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



（插入前）



（插入后）

2.3 线性表的链式表示与实现

插入

- 第一种情况：在第一个结点前插入

```
s->next = head ;
head = s;
```

- 第二种情况：在链表中间插入

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

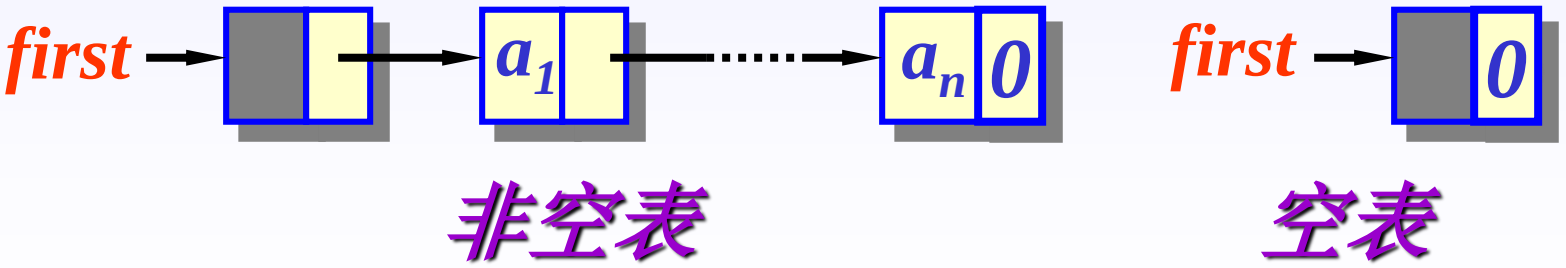
- 第三种情况：在链表末尾插入

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

2.3 线性表的链式表示与实现

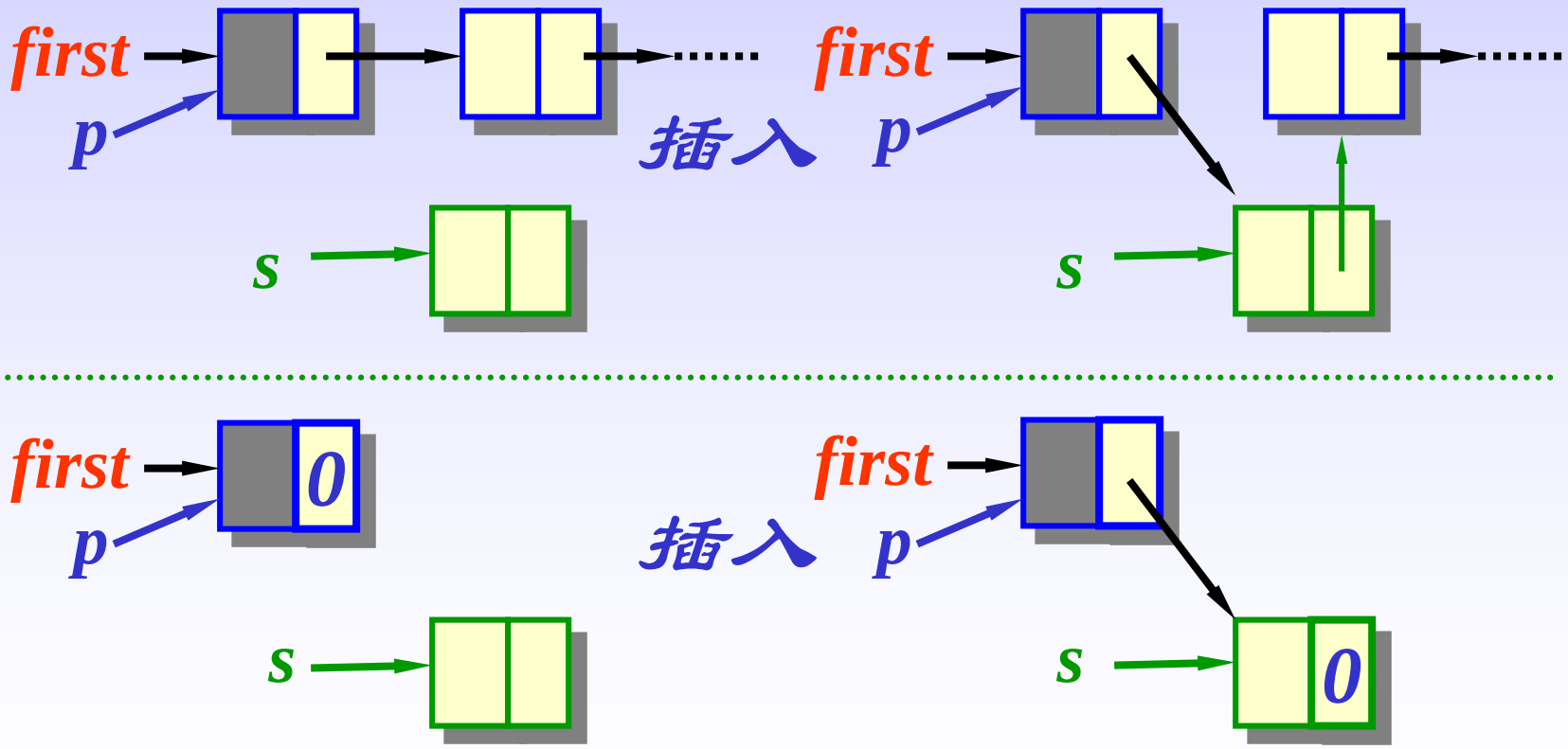
5. 带头结点的单链表

- 头结点位于表的最前端，本身不带数据，仅标志表头。
- 设置头结点的目的是统一空表与非空表的操作，简化链表操作的实现。



2.3 线性表的链式表示与实现

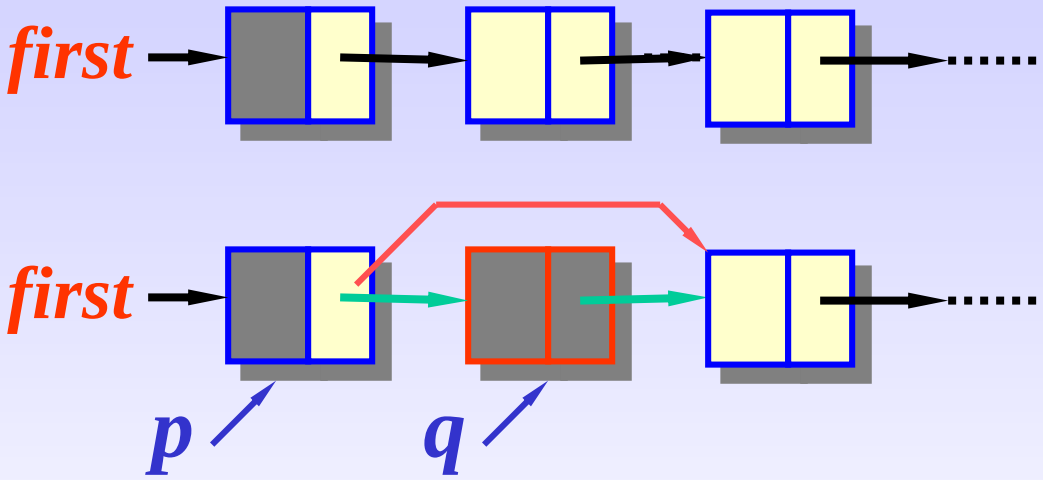
(1)在带头结点的单链表第一个结点前插入新结点



s->next = p->next; p->next = s;

2.3 线性表的链式表示与实现

(2)从带头结点的单链表中删除第一个结点




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

2.3 线性表的链式表示与实现

6. 单链表的若干算法及操作

(1) 根据已有数据生成一个链表

CreateList_L( )

(2) 插入

ListInsert_L ( )

(3) 删除

ListDelete_L ( )

(4) 归并两个链表

MergeList_L ( )

2.3 线性表的链式表示与实现

(1) 根据已有数据生成一个链表

设线性表n个元素已存放在数组a中，建立一个单链表，h为头指针。

算法思路：

（1）创建头结点

（2）逆序创建结点n，修改结点指针，再创建结点n-1,……直到结点1

2.3 线性表的链式表示与实现

```
void CreateList_L ( LinkList &L, int n ) {
   // 逆位序输入 n 个数据元素的值，建立带头结点的单链表 L
   L = ( LinkList ) malloc ( sizeof ( LNode ) );
   L->next = NULL; // 先建立一个带头结点的空链表
   for ( i = n; i > 0; --i ) {
         p = ( LinkList ) malloc ( sizeof ( LNode ) ); // 生成新结点
         scanf ( &p->data ); // 按照逆位序输入数据元素值
         p->next = L->next; // 将新结点插入到单链表的头
         L->next = p; // 修改单链表头结点的指针域
   } // for 结束
} CreatList_L
```

$$T(n) = O(n)$$

2.3 线性表的链式表示与实现

(2) 插入

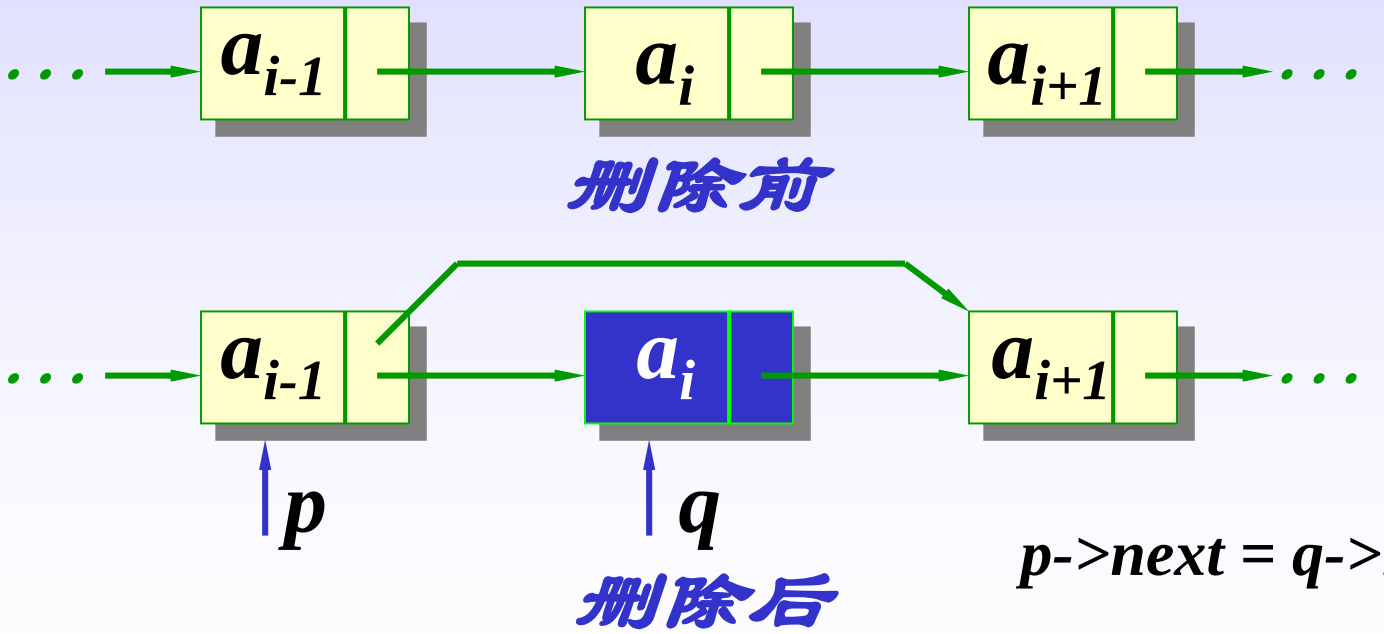
```
Status ListInsert_L ( LinkList &L, int i, ElemType e ) {
// 在带头结点的单链线性表L中第 i 个位置之前插入元素 e
  p = L; j = 0;
  while ( p && j<i-1 ) { p = p->next; ++j; } // 寻找第 i-1 个结点
  if ( ! p || j >i-1 ) return ERROR; // i 小于 1 或 i 大于表长
  s = ( LinkList ) malloc ( sizeof ( LNode ) ); // 生成新结点
  s->data = e; // 使新结点数据域的值为 e
  s->next = p->next; // 将新结点插入到单链表 L 中
  p->next = s; // 修改第 i-1 个结点指针
  return OK;
} // LinkInsert_L
```

$T(n)=O(n)$

2.3 线性表的链式表示与实现

(3) 删除

- 第一种情况: 删除表中第一个元素
- 第二种情况: 删除表中或表尾元素



在单链表中删除含a_i的结点

2.3 线性表的链式表示与实现

```
Status ListDelete_L ( LinkList &L, int i, ElemType &e ) {
// 在带头结点的单链线性表 L 中，删除第 i 个元素，并由 e 返回其值
  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;
} // LinkDelete_L
```

$T(n)=O(n)$

2.3 线性表的链式表示与实现

```
Status ListDelete_L ( LinkList &L, int i, ElemType &e ) {
// 在带头结点的单链线性表 L 中，删除第 i 个元素，并由 e 返回其值
  p = L; j = 0;
  while ( p->next && j<i-1 ) { // 寻找第 i-1 个结点
    p = p->next; ++j;
  }
  if ( ! p || j >i-1 ) return ERROR; // 删除位置不合理
  q = p->next;   // 用指针 q 指向被删除结点
  p->next = q->next;   // 删除第 i 个结点
  e = q->data;    // 取出第 i 个结点数据域值
  free (q);     // 释放第 i 个结点
  return OK;
} // LinkDelete_L
```

$T(n)=O(n)$

2.3 线性表的链式表示与实现

(4) 归并两个链表

```
void MergeList_L ( LinkList &La, LinkList &Lb, LinkList &Lc, )
{// 已知单链表 La 和 Lb 的元素按非递减排列，
    // 归并 La 和 Lb 得到新的单链表 Lc ， Lc 的元素也按非递减排列
    pa = La->next; pb = Lb->next;
    Lc = pc = La; // 用 La 的头结点作为 Lc 的头结点
    while ( pa && pb ) {
        if ( pa->data <= pb->data ) // 如果 pa->data≤pb->data
        {       pc->next = pa;   pc = pa;   pa = pa->next;     }
        else // 如果 pa->data > pb->data
        {       pc->next = pb;    pc = pb;   pb = pb->next;    }
    }
    pc->next = pa ? pa : pb; // 插入剩余段
    free (Lb); // 释放 Lb 的头结点
} // MergeList_L
```

2.3 线性表的链式表示与实现

7. 静态链表

定义：用数组描述的链表叫静态链表

存储结构：

```
#define  MAXSIZE = 100;     //静态链表的最大长度
typedef struct {
    ElemType  data;
    int   cur;    //游标,代替指针指示结点在数组中的位置
} component, SLinkList[MAXSIZE];
```

目的是为了在不设指针类型的高级程序设计语言中使用链表结构。

2.3 线性表的链式表示与实现

2.3 线性表的链式表示与实现

在静态链表中查找第1个具有给定值e的结点

```
int LocateElem_SL ( SLinkList S, ElemType e )
//若找不到，则返回0
{  i = S [0].cur;   //i 指向表第一个结点
   while ( i && S [i].data != e)
        i = S [i].cur; //顺链查找,相当于p=p->next
   return i;
} //LocateElem_SL
```

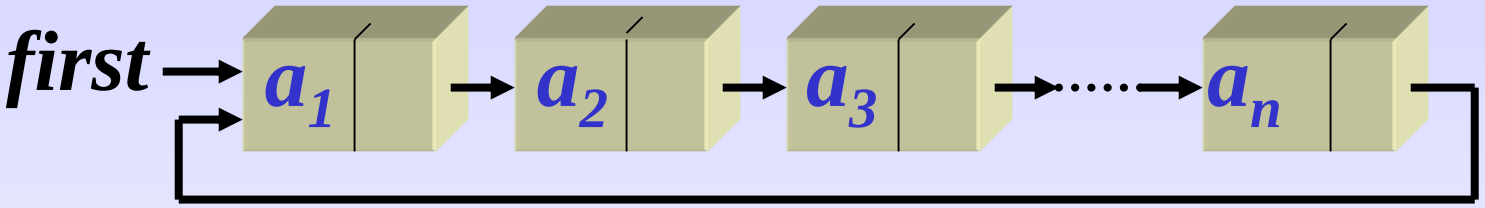
2.3 线性表的链式表示与实现

2.3.2 循环链表 (Circular List)

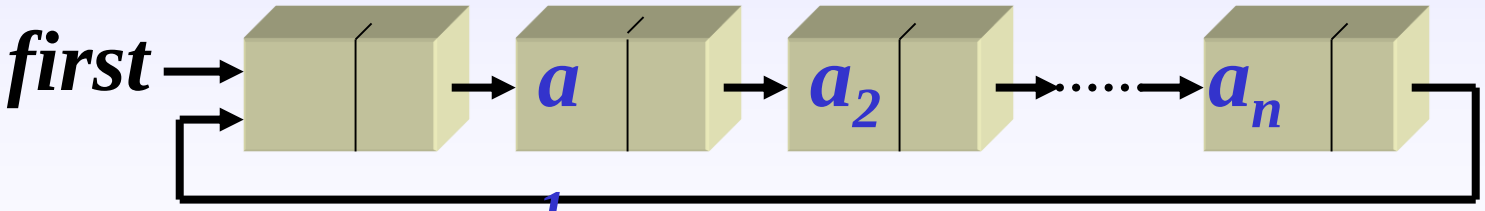
- 循环链表是单链表的变形。
- 循环链表最后一个结点的 link 指针不为NULL，而是指向了表的前端。
- 为简化操作，在循环链表中往往加入头结点。
- 循环链表的特点是：只要知道表中某一结点的地址，就可搜寻到所有其他结点的地址。

2.3 线性表的链式表示与实现

- 循环链表的示例

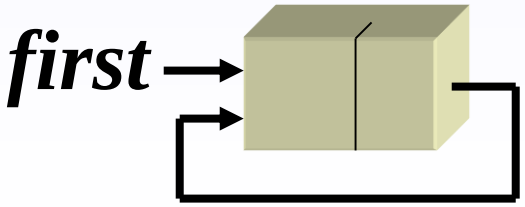


- 带头结点的循环链表



1

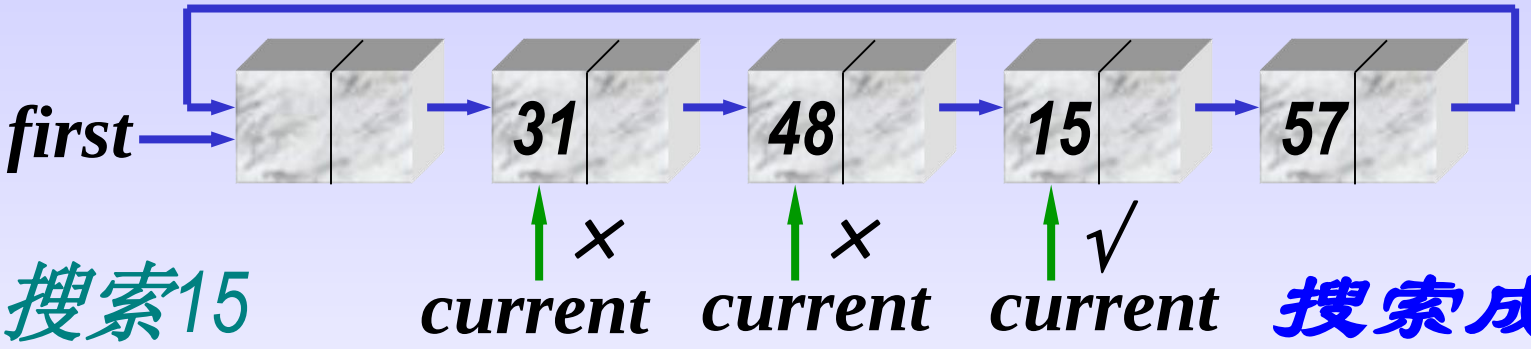
（非空表）

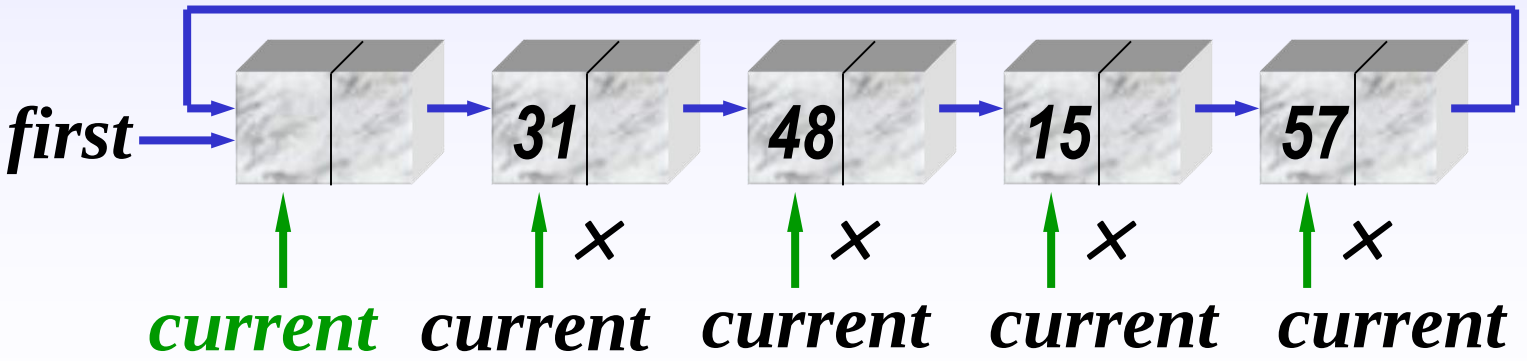


（空表）

2.3 线性表的链式表示与实现

循环链表的搜索算法





2.3 线性表的链式表示与实现

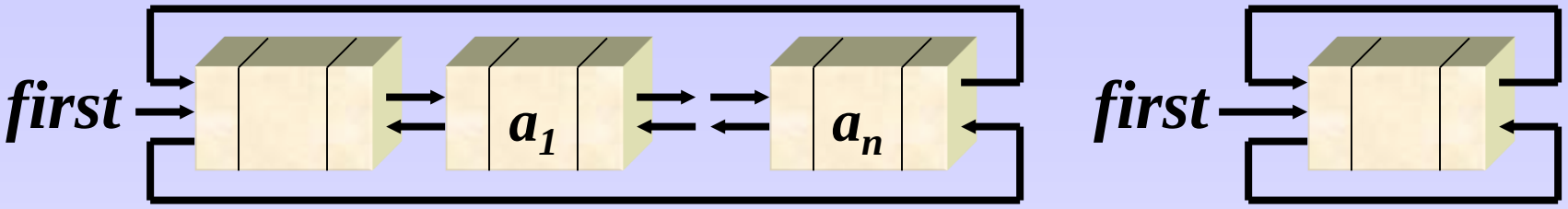
2.3.3 双向链表 (Doubly Linked List)

- 双向链表是指在前驱和后继方向都能游历(遍历)的线性链表。
- 双向链表每个结点有两个指针域, 结构如下：

前驱方向 ← → 后继方向

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

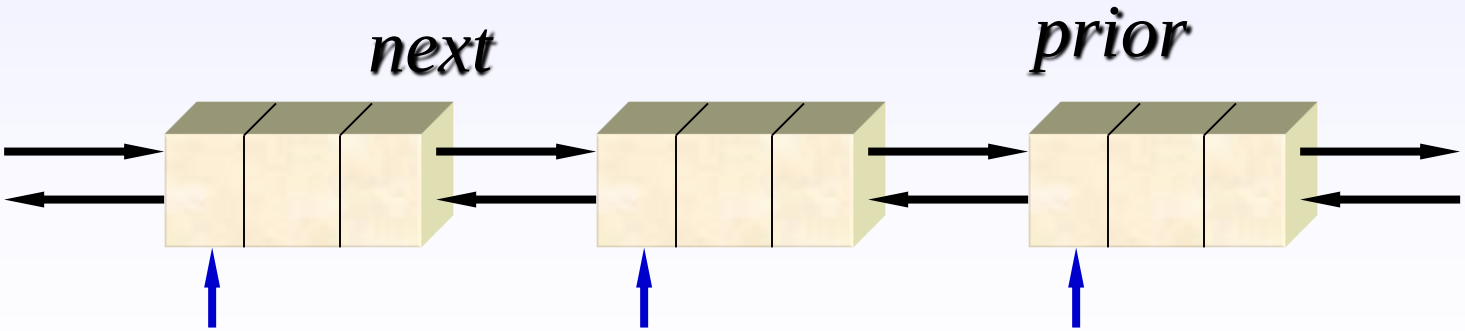
2.3 线性表的链式表示与实现



非空表　　空表

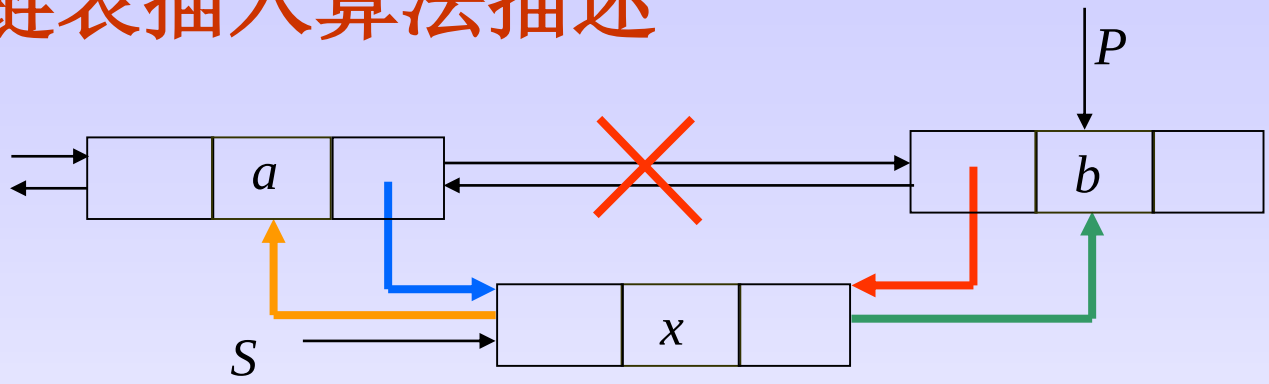
- 结点指向

$p == p->prior->next == p->next->prior$



2.3 线性表的链式表示与实现

双向链表插入算法描述




```
s->data=x;
s->prior=p->prior;
p->prior->next=s;
s->next=p;
p->prior=s;
```

2.3 线性表的链式表示与实现

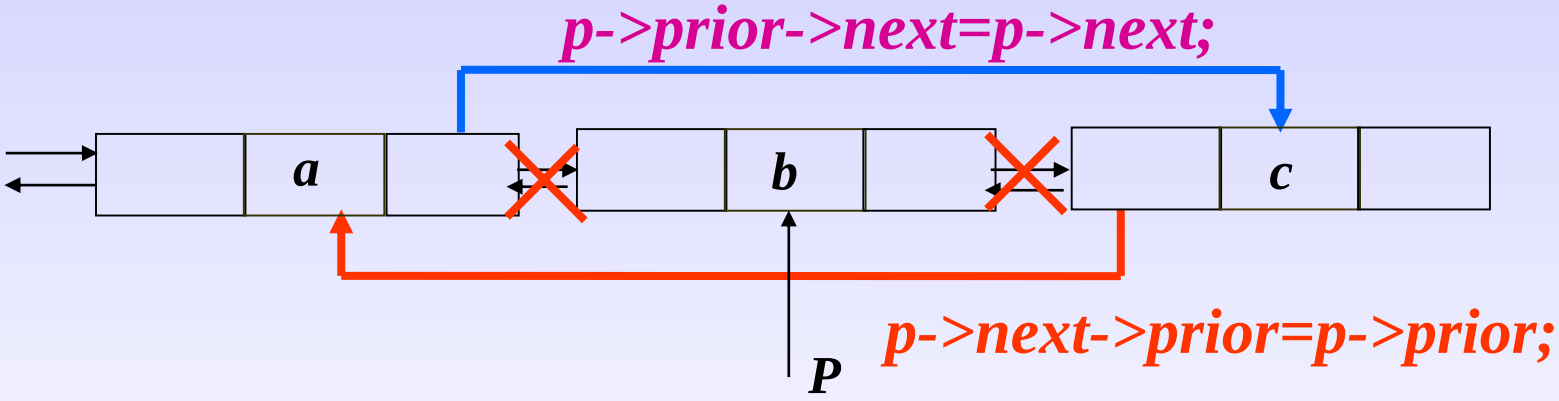
双向链表的插入操作算法:

```
Status ListInsert_DuL ( DuLinkList &L, int i, ElemType e ) {
// 在带头结点的双向循环线性表L 中第i 个位置之前插入元素e，1≤i≤表长+1
if ( ! ( p = GetElemP_DuL ( L, i ) ) ) return ERROR;
// 在L 中确定第i 个元素的位置指针p，若p = NULL，则不存在
if ( ! ( s = ( DuLinkList ) malloc ( sizeof ( DuLNode ) ) ) ) return
ERROR
s->data = e; // 将数据放入新结点的数据域
s->prior = p->prior; // 将p 的前驱结点指针放入新结点的前向指针域
s->next = p; //将p 的放入新结点的/反向指针域
p->prior->next = s; // 修改p 的前驱结点的反向指针
p->prior = s; // 修改p 的前向指针
return OK;
} // ListInsert_DuL
```

– 算法评价：$T(n)=O(n)$

2.3 线性表的链式表示与实现

双向链表删除算法描述




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

2.3 线性表的链式表示与实现

双向链表的删除操作算法:

```
Status ListDelete_DuL ( DuLinkList &L, int i, ElemType &e )
{ // 删除带头结点的双向循环链表L 中第i 个元素并返回其值，1≤i≤表长
if ( ! ( p = GetElemP_DuL ( L, i ) ) ) return ERROR;
// 在L 中确定第i 个元素，p 为指向该结点的指针；
// 若i < 1 或i > 表长，则p 为NULL，第i 个元素不存在
e = p->data; // 将p 指向的结点数据域中的值取出
p->prior->next = p->next; // 修改p 的前驱结点的反向指针
p->next->prior = p->prior; // 修改p 的后继结点的前向指针
free (p); // 释放p 结点
return OK;
} // ListDelete_DuL
```

算法评价：T(n)=O(n)

2.3 线性表的链式表示与实现

2.3.4 顺序表与链表的比较

（1）基于空间的比较

- 存储分配的方式
  - 顺序表的存储空间是静态分配的
  - 链表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

2.3 线性表的链式表示与实现

（2）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
  - 若插入/删除仅发生在表的两端，宜采用带尾指针的循环链表

2.4 一元多项式的表示及相加

1. 一元多项式的表示

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

- **n 阶多项式 $P_n(x)$ 有 n+1 项**
  - 系数 $P_0, P_1, P_2, \ldots, P_n$
  - 指数 0, 1, 2, …, n。按升幂排列

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

2.4 一元多项式的表示及相加

若：

$$S(x)=1+3x^{1000}+2x^{20000}$$

对S(x)这样的多项式采用全部存储的方式则浪费空间。

怎么办？

2.4 一元多项式的表示及相加

若：

$$S(x)=1+3x^{1000}+2x^{20000}$$

对S(x)这样的多项式采用全部存储的方式则浪费空间。

怎么办？

一般n次多项式可以写成：

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

其中 $0\le e1<e2<\cdots\cdots<em=n$

P_i 为非零系数。

因此可以用数据域含两个数据项的线性表来表示：

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

其存储结构可以用顺序存储结构，也可以用单链表。

2.4 一元多项式的表示及相加

2. 多项式的抽象数据类型

ADT Polynomial {

数据对象：

$D = \{ a_i \mid a_i \in TermSet,\ i=1,2,...,m,\ m \geq 0$

TermSet 中的每个元素包含一个表示系数的实数和表示指数的整数}

数据关系：

$R = \{ <a_{i-1}, a_i> \mid a_{i-1}, a_i \in D,\quad i=2,...,n$

且a_{i-1}中的指数值<a_i中的指数值}

2.4 一元多项式的表示及相加

基本操作：

CreatPolyn (&P, m)

操作结果：输入 m 项的系数和指数，建立一元多项式 P。

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。

PrintPolyn (&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 P。

2.4 一元多项式的表示及相加

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 中的项数。

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。

SubtractPolyn (&Pa, &Pb)

… …

} ADT Polynomial

2.4 一元多项式的表示及相加

3. 多项式的链式存储表示

- 在多项式的链表表示中每个结点三个数据成员：

```
typedef   struct LNode
{   int coef, exp;
    struct LNode  *next;
}LNode;
```

- 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。

2.4 一元多项式的表示及相加

4. 一元多项式的相加算法

- 扫描两个多项式，若都未检测完：
  - 若当前被检测项指数相等，系数相加。若未变成 0，则将结果加到结果多项式。
  - 若当前被检测项指数不等，将指数小者加到结果多项式。
- 若一个多项式已检测完，将另一个多项式剩余部分复制到结果多项式。

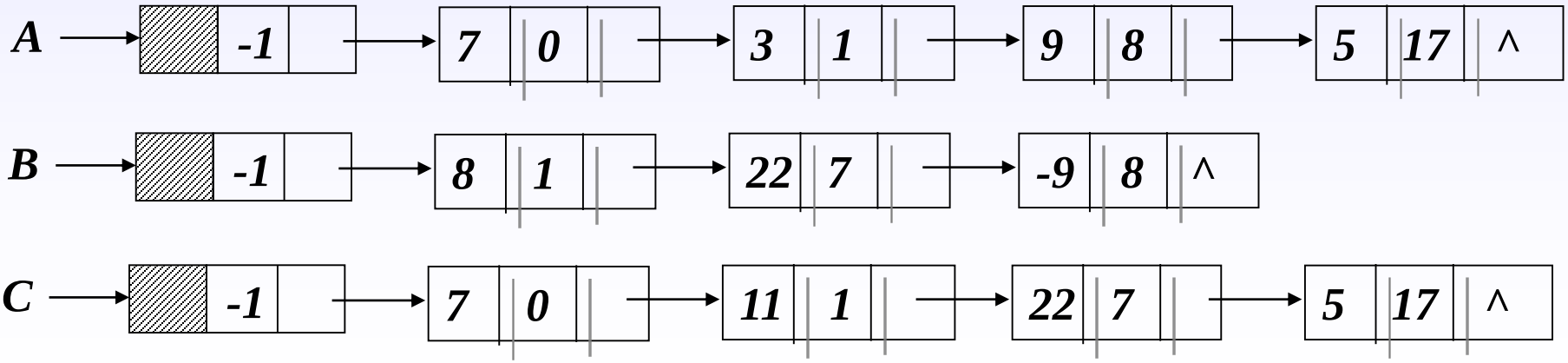
2.4 一元多项式的表示及相加

例: 一元多项式相加

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

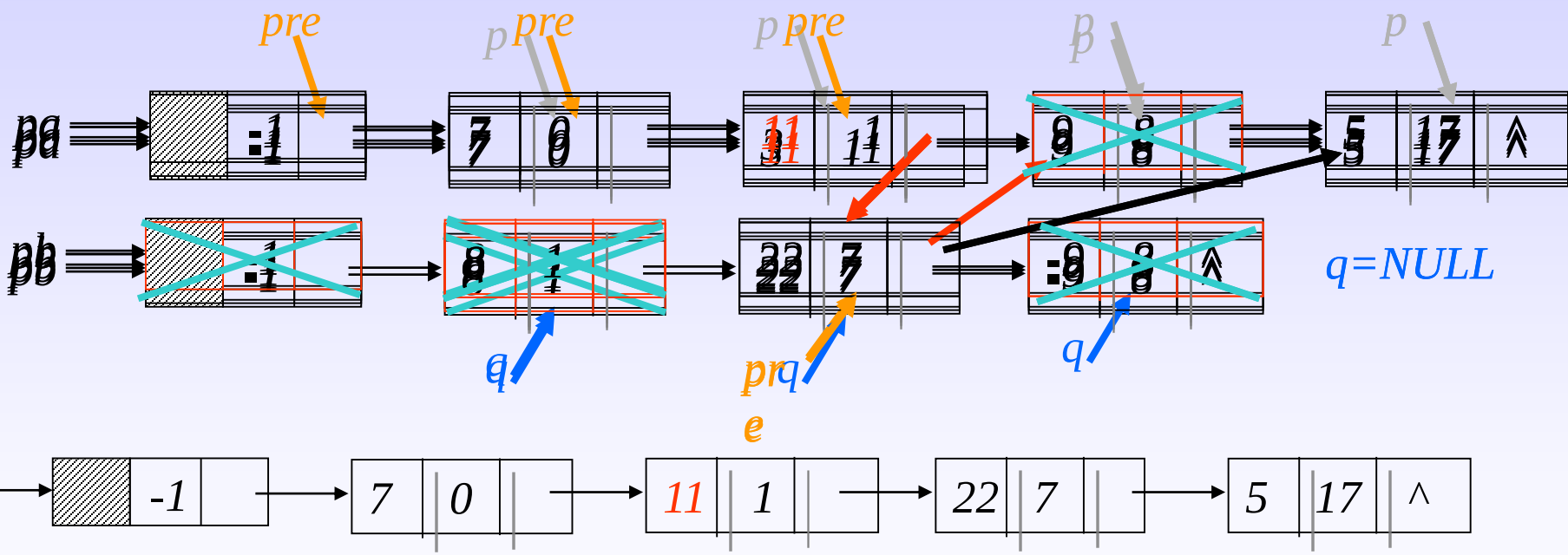
$$B(x)=8x+22x^{7}-9x^{8}$$

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



```
Void AddPoly(polynomial &Pa, polynomial &Pa)  // 多项式相加： Pa = Pa + Pb;
{ ha = GetHead(Pa); hb = GetHead(Pb);   // ha和hb分别指向Pa和Pb的头结点；
  qa = NextPos(Pa, ha), qb = NextPos(Pb, hb);   // qa和qb分别指向Pa和Pb的当前结点；
  while (qa &&qb) {                                  // qa和qb均非空；
      a = GetCurElem(qa); b = GetCurElem(qb); // a和b分别为两表中当前比较元素；
      switch (*cmpexp(a,b)){
          case -1        // a中的指数小
              ha = qa; qa = NextPos(Pa, qa); break;
          case 0         // a和b的指数相等
              sum = a.coef + b.coef;
              if (sum != 0.0)  // 修改多项式PA中当前节点的系数值
                  SetCurElem(qa, sum); ha = qa; }
              else { DelFirst (ha, qa); FreeNode(qa); } // 删除多项式PA中的当前节点
              DelFirst (hb, qb); FreeNode(qb); qb = NextPos(Pb, hb));
              qa = NextPos(Pa, ha); break;
          case 1:        // b中的指数小
              delFirst (hb, qb); InsFirst (ha, qb);
              qb = NextPos(Pb, hb); ha = NextPos(Pa, ha); break
          }
      }
      if (!ListEmpty(Pb)) Append(Pa, qb);  //链接Pb中的剩余节点
      FreeNode(hb);  //释放Pb的头结点
}
```

2.4 一元多项式的表示及相加



本章小结

- 线性表的抽象数据类型
- 线性表的顺序存储和实现
- 线性表的链式存储和实现
  - 单链表
  - 双向链表
  - 循环链表
  - 静态链表
- 一元多项式的表示和相加

第二章　作业

- 《数据结构题集》P.15 - P.20
  - 2.9
  - 2.12
  - 2.14
  - 2.39
- 下周六（10月8日）上课时交