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

1.4.1 基本概念和术语

小结：数据结构的三个方面：

- 数据的逻辑结构
  - 线性结构
    - 线性表、栈和队列、串、数组和特殊矩阵
  - 非线性结构
    - 树形结构
    - 图形结构
- 数据的存储结构
  - 顺序存储
  - 链式存储
- 数据的运算：检索、排序、插入、删除、修改等

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

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
```

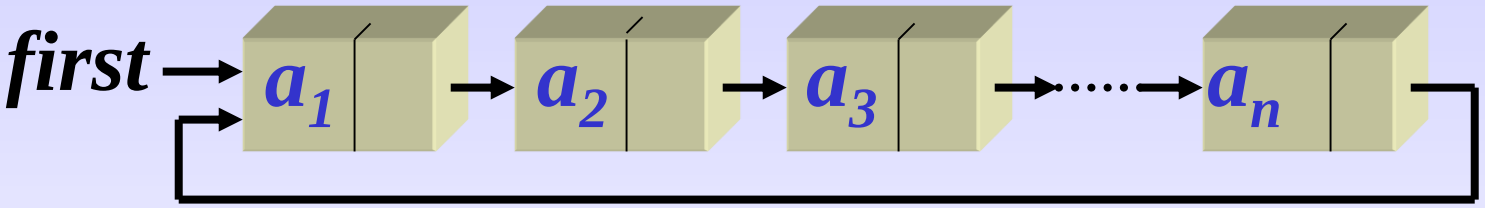
2.3 线性表的链式表示与实现

2.3.2 循环链表 (Circular List)

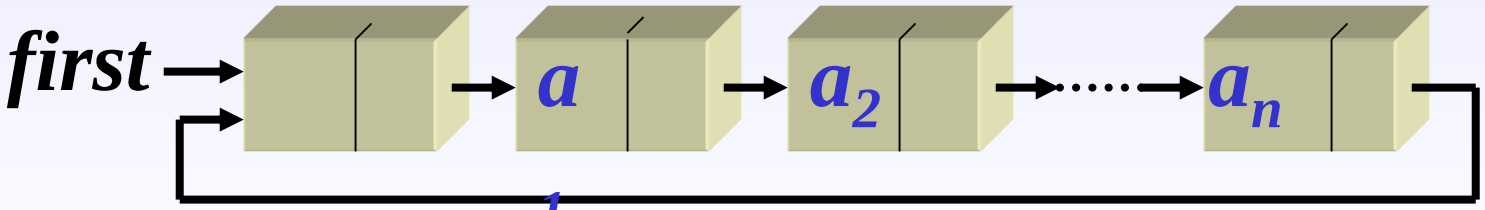
- 循环链表是单链表的变形。
- 循环链表最后一个结点的 link 指针不为NULL，而是指向了表的前端。
- 为简化操作，在循环链表中往往加入头结点。
- 循环链表的特点是：只要知道表中某一结点的地址，就可搜寻到所有其他结点的地址。

2.3 线性表的链式表示与实现

- 循环链表的示例

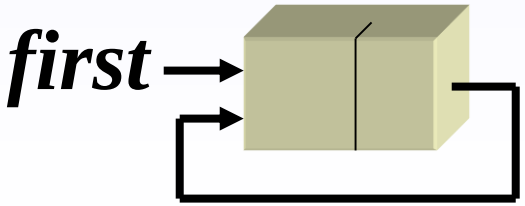


- 带头结点的循环链表



1

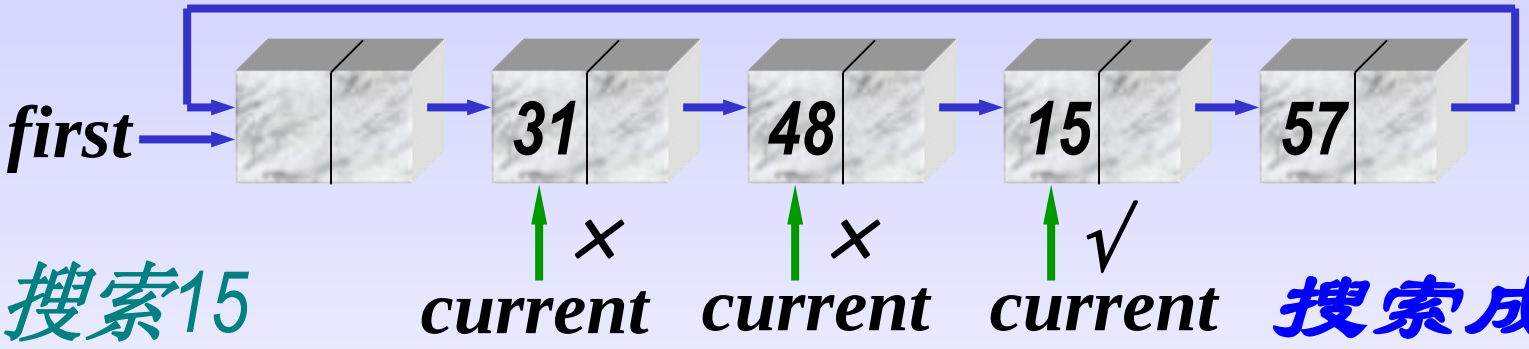
（非空表）



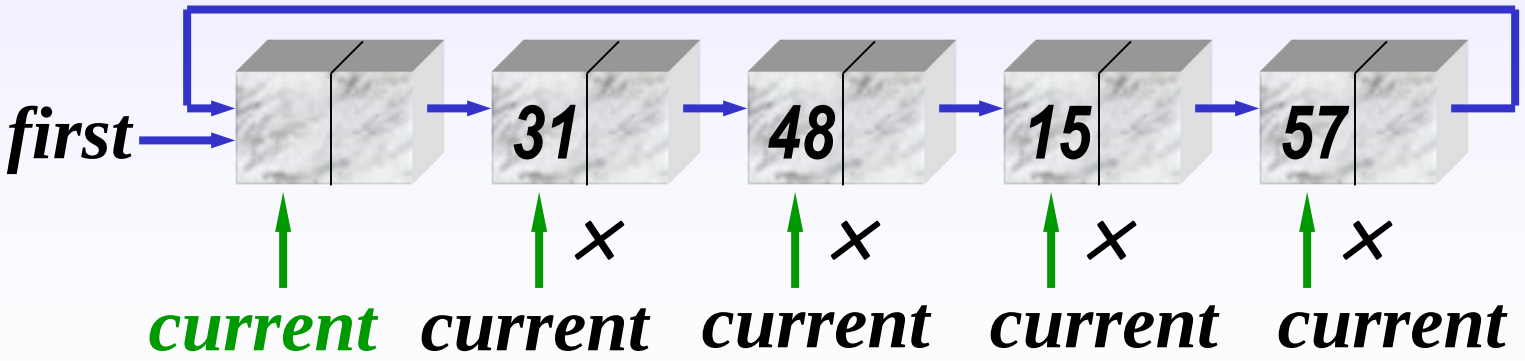
（空表）

2.3 线性表的链式表示与实现

循环链表的搜索算法



搜索15　　**搜索成功**



搜索25　　判定条件 current == first　　**搜索不成功**

2.3 线性表的链式表示与实现

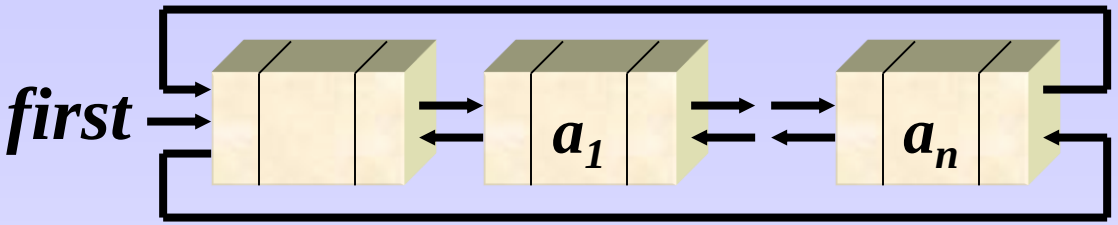
2.3.3 双向链表 (Doubly Linked List)

- 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：

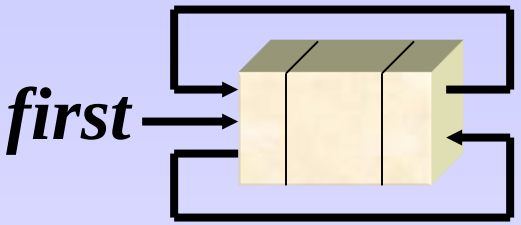
前驱方向 ← → 后继方向

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

2.3 线性表的链式表示与实现



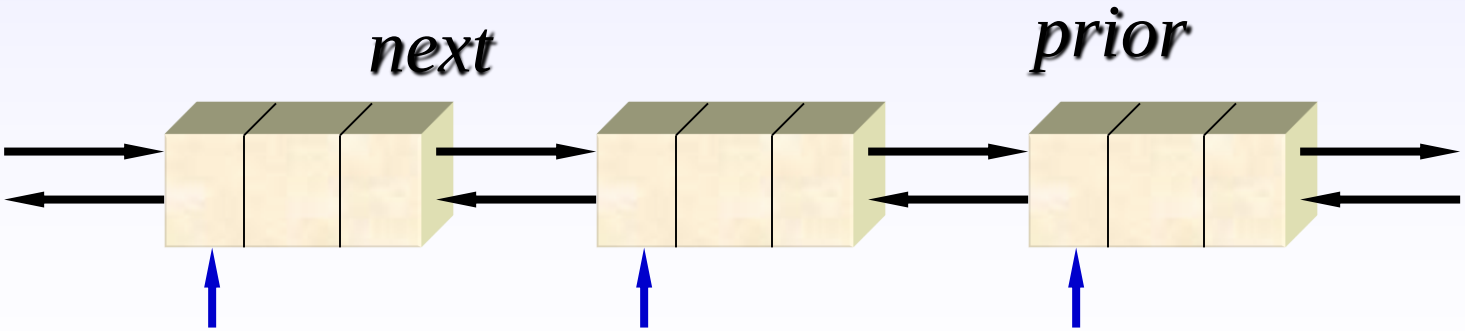
非空表



空表

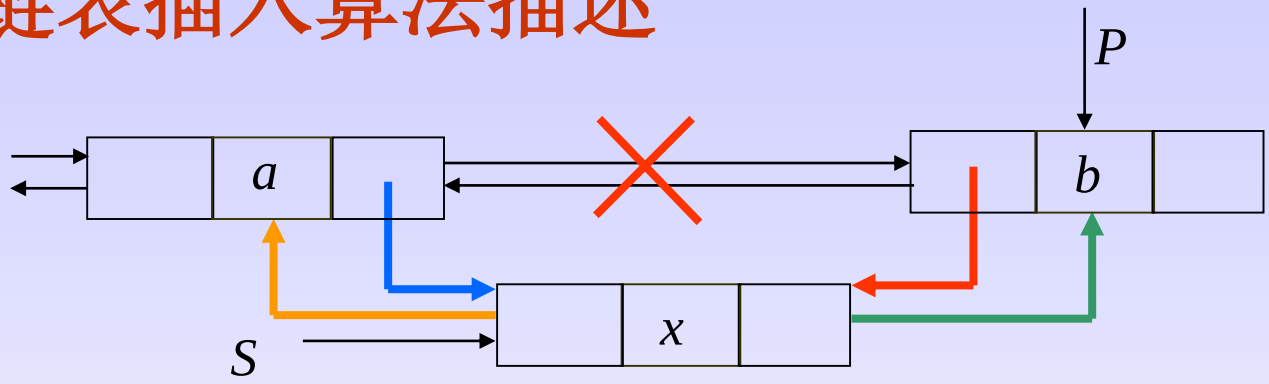
- 结点指向

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



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

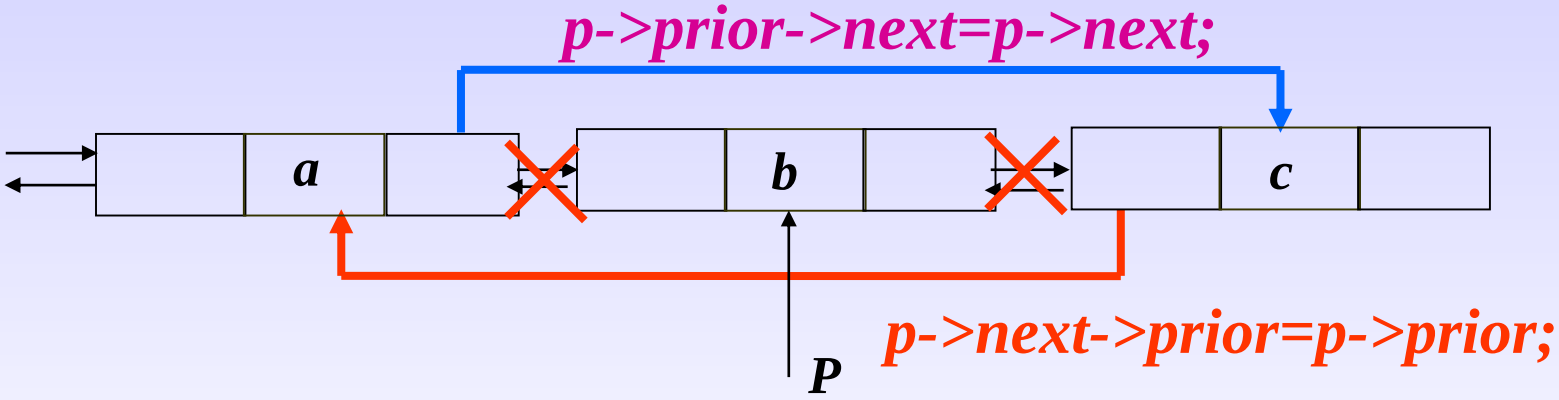
双向链表的插入操作算法:

```
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

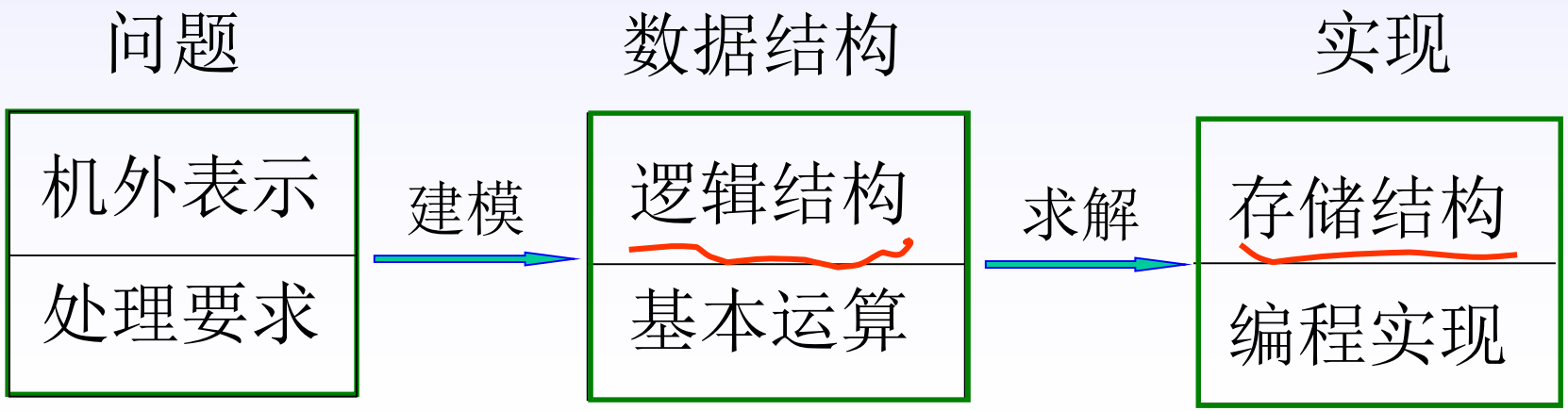
（2）基于时间的比较

- 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

1.2 什么是数据结构

计算机求解问题的步骤：

- 分析问题；
- 建立求解问题的数据结构并设计算法—通过算法来表示对象数据及其相互关系；
- 实现：编制程序模拟对象领域中的求解过程。



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？

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：

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

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。

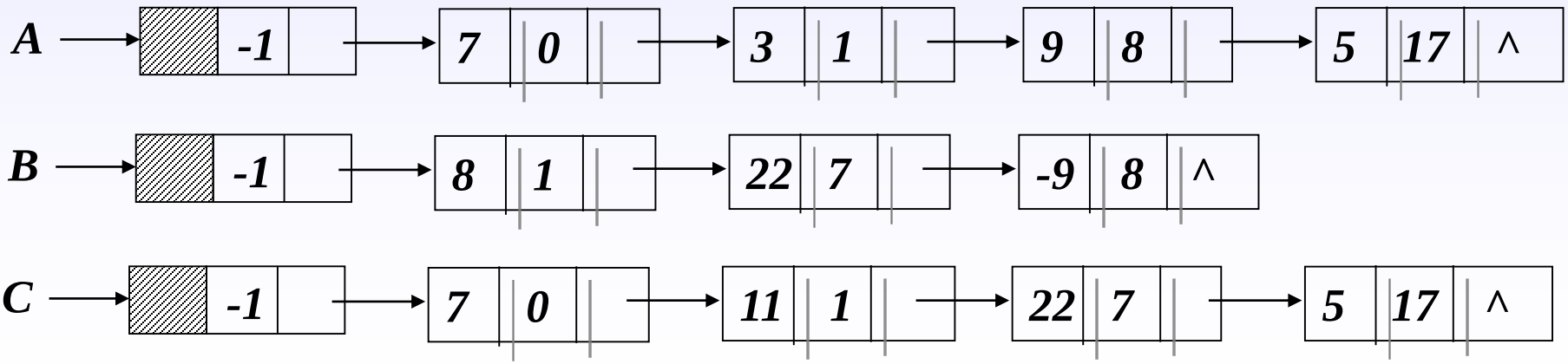
2.4 一元多项式的表示及相加

例: 一元多项式相加

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

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

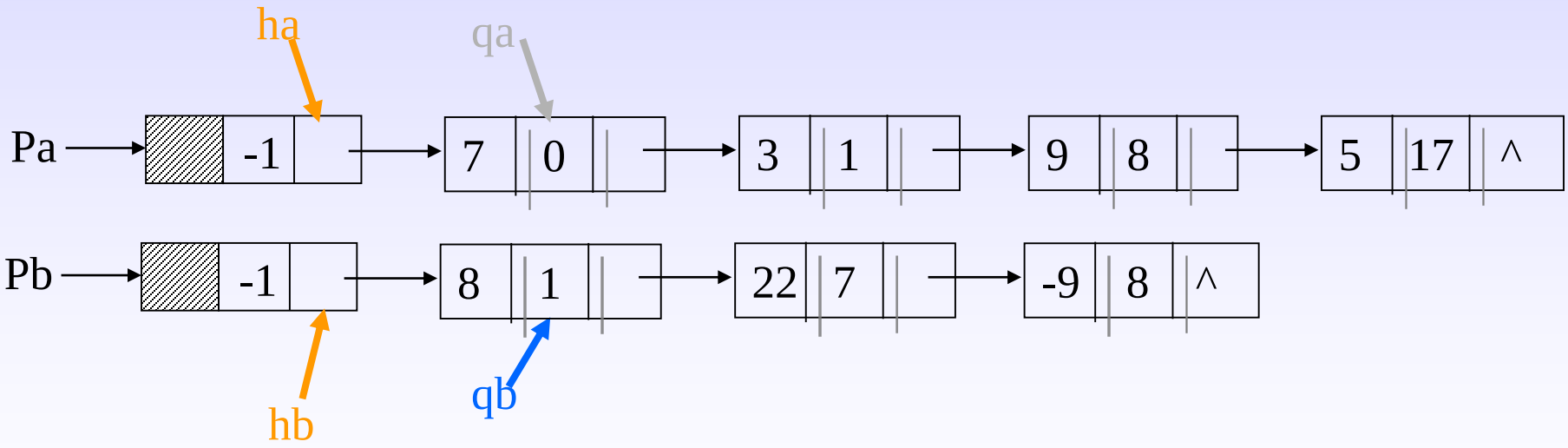
$$C(x)=A(x)+B(x)=7+11x+22x^{7}+5x^{17}$$



```
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
}
```

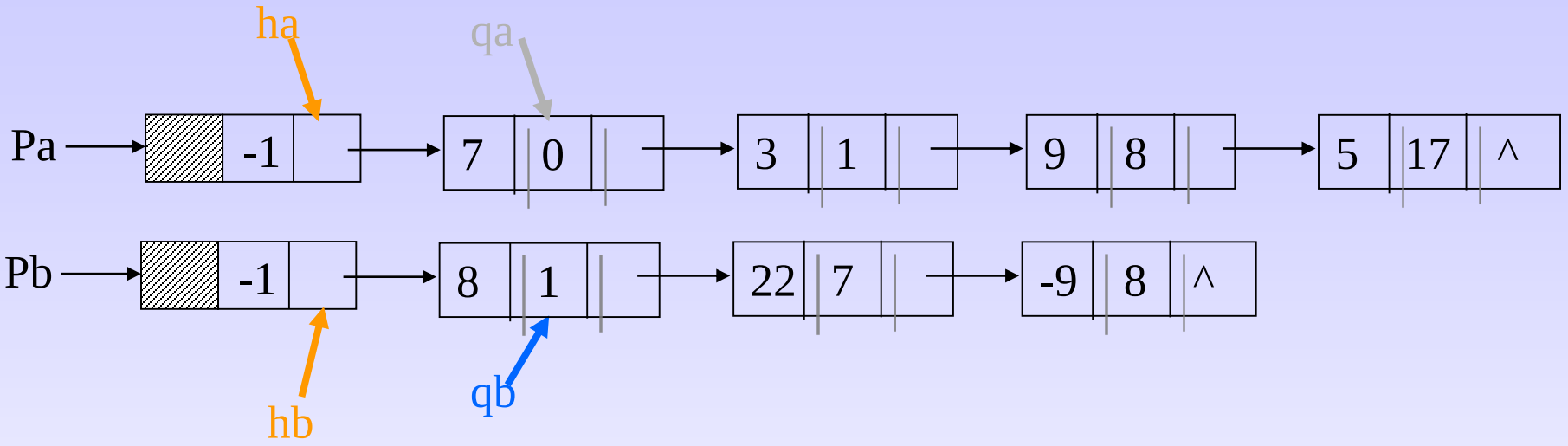
2.4 一元多项式的表示及相加

```
{ 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;
```



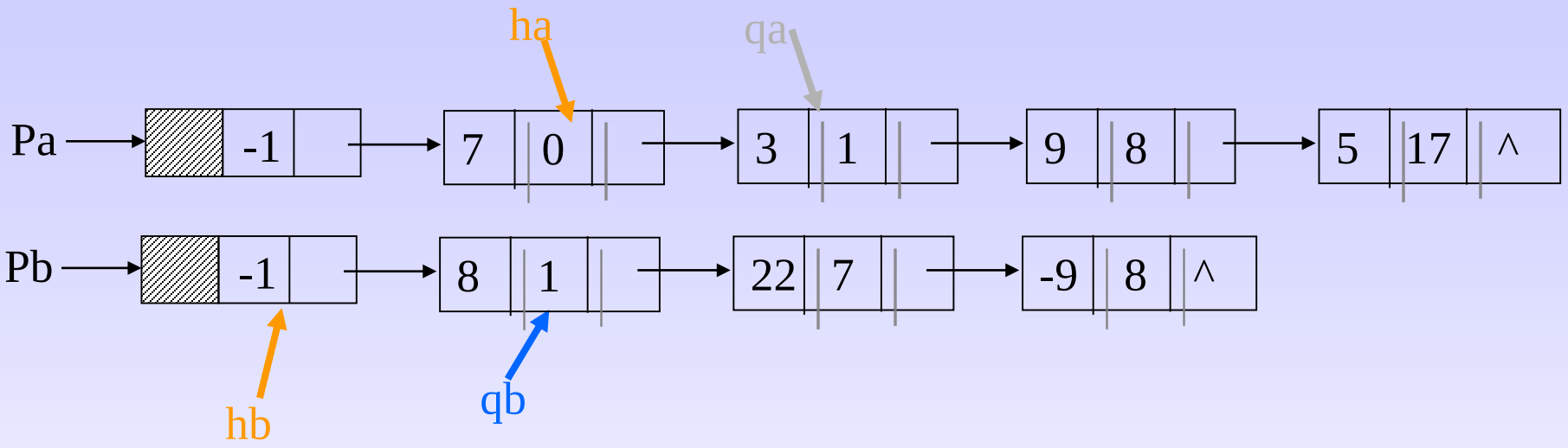
```
Void AddPoly(polynomial &Pa, polynomial &Pa)  // 多项式相加： Pa = Pa + Pb;
{ ha = GetHead(Pa); hb = GetHead(Pb);   // ha和hb分别指向Pa和Pb的头结点；
  qa = NextPos(Pa, ha), qb = NextPos(Pb, hb);   // qa和qb分别指向Pa和Pb的当前结点；
  while (qa &&qb) {                                  // qa和qb均非空；
      a = GetCurElem(qa); 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




```
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
                }
```

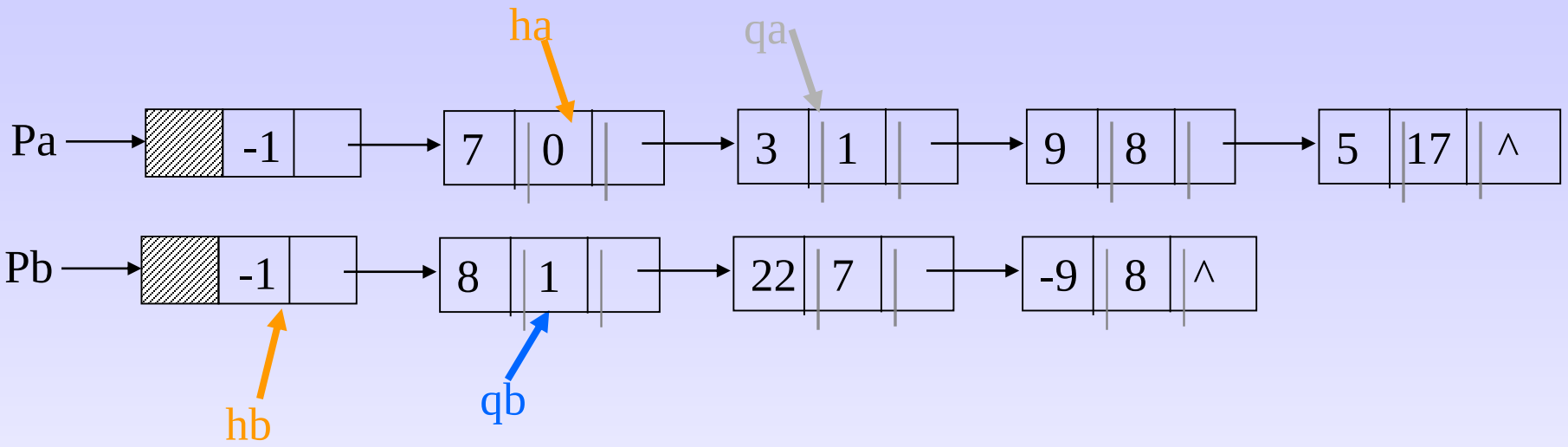
2.4 一元多项式的表示及相加




```
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
                }
```

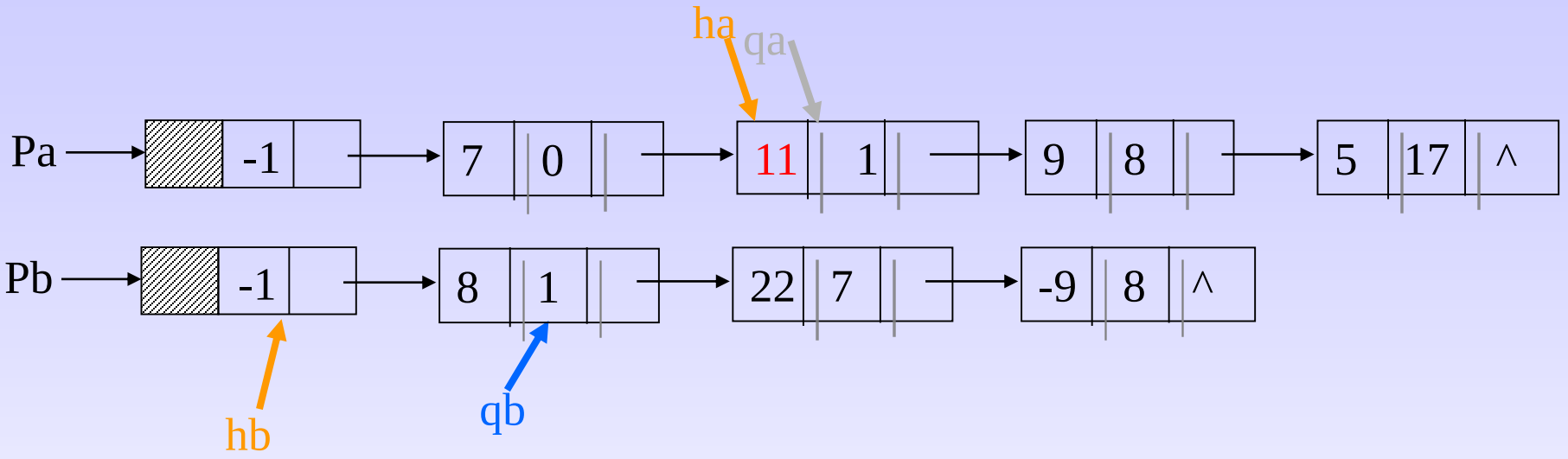
```
Void AddPoly(polynomial &Pa, polynomial &Pa)  // 多项式相加： Pa = Pa + Pb;
{ ha = GetHead(Pa); hb = GetHead(Pb);   // ha和hb分别指向Pa和Pb的头结点；
  qa = NextPos(Pa, ha), qb = NextPos(Pb, hb);   // qa和qb分别指向Pa和Pb的当前结点；
  while (qa &&qb) {                                  // qa和qb均非空；
      a = GetCurElem(qa); 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



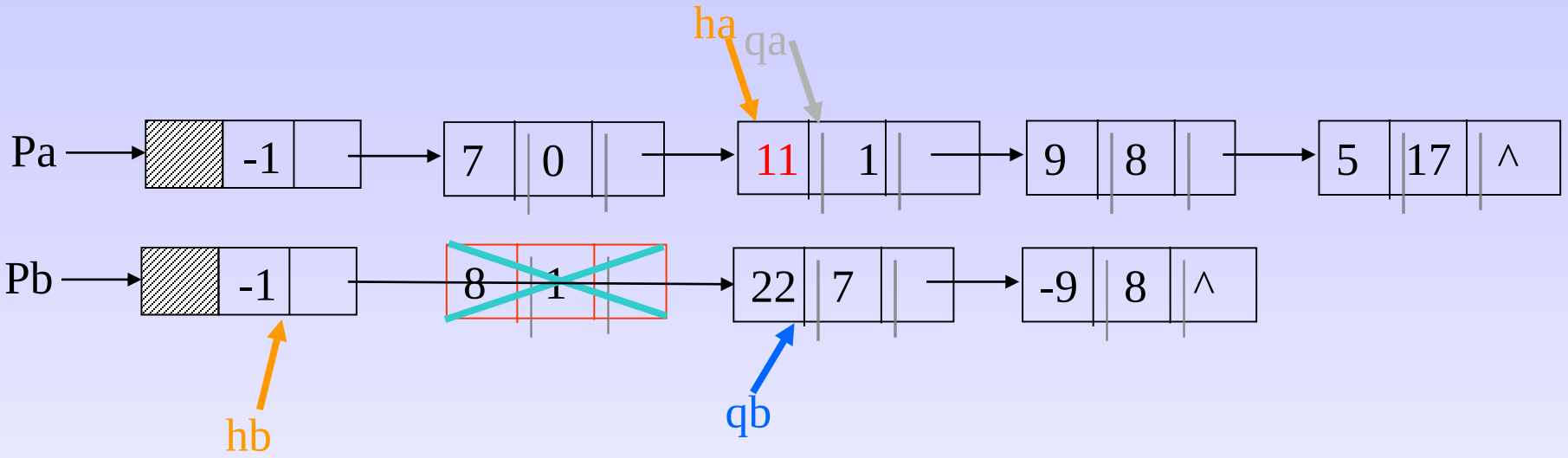

```
switch (*cmpexp(a,b)){
                case -1          // a中的指数小
                    ha = qa; qa = NextPos(Pa, qa); break;
                case 0           // a和b的指数相等
                    sum = a.coef + b.coef;
                    if (sum != 0.0)   // 修改多项式PA中当前节点的系数值
                        SetCurElem(qa, 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
                }
```

2.4 一元多项式的表示及相加



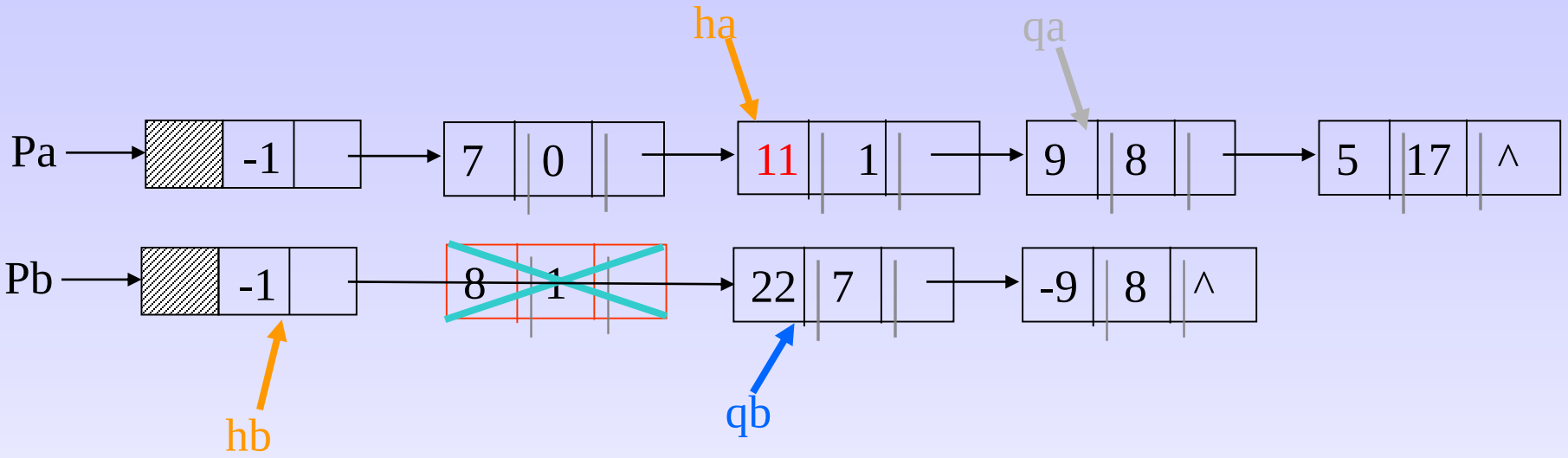

```
switch (*cmpexp(a,b)){
                case -1          // a中的指数小
                    ha = qa; qa = NextPos(Pa, qa); break;
                case 0           // a和b的指数相等
                    sum = a.coef + b.coef;
                    if (sum != 0.0)   // 修改多项式PA中当前节点的系数值
                        SetCurElem(qa, 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
                }
```

2.4 一元多项式的表示及相加



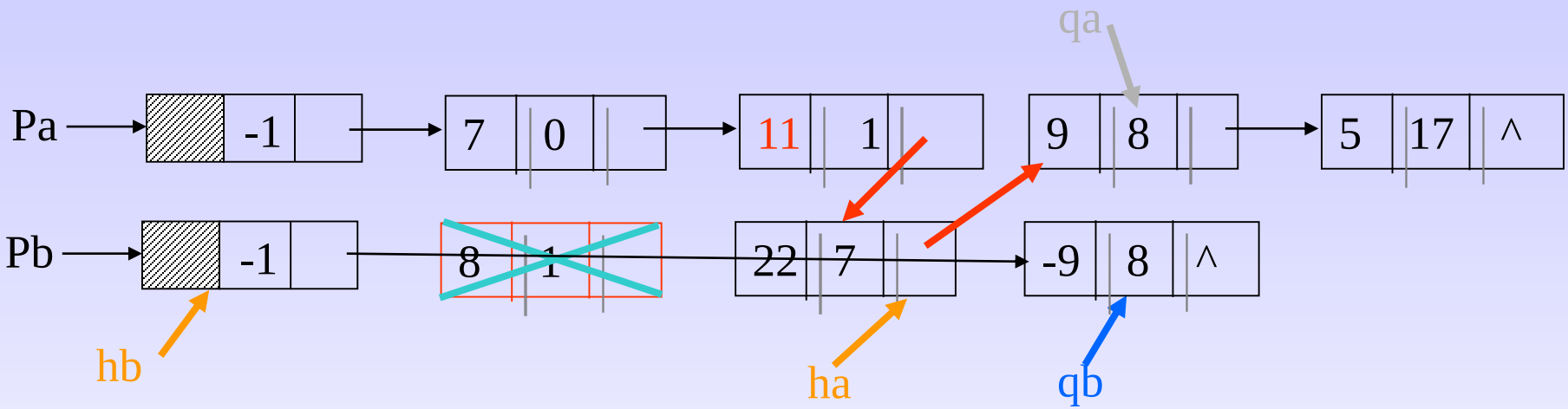

```
switch (*cmpexp(a,b)){
                case -1          // a中的指数小
                    ha = qa; qa = NextPos(Pa, qa); break;
                case 0           // a和b的指数相等
                    sum = a.coef + b.coef;
                    if (sum != 0.0)   // 修改多项式PA中当前节点的系数值
                        SetCurElem(qa, 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
                }
```

2.4 一元多项式的表示及相加




```
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
                }
```

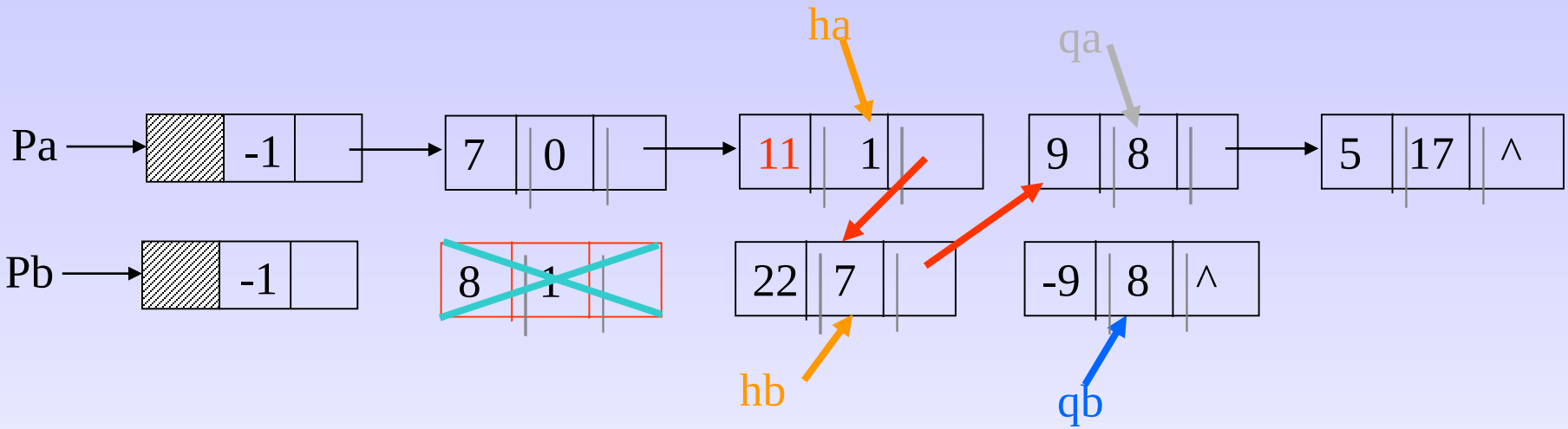
2.4 一元多项式的表示及相加




```
switch (*cmp(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
                }
```

```
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
}
```

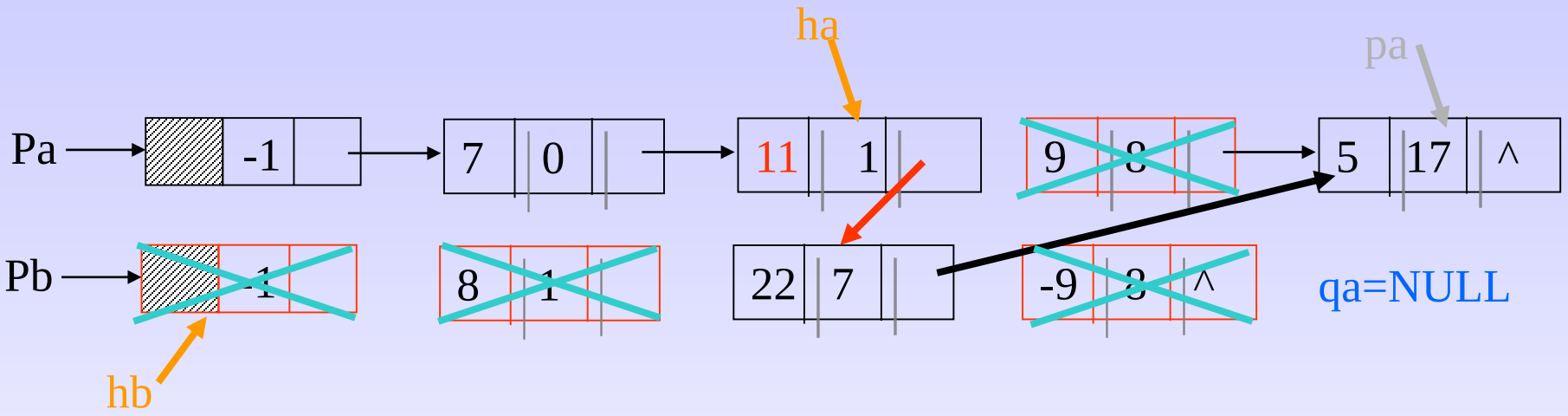
2.4 一元多项式的表示及相加




```
switch (*cmp(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
                }
```

```
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
}
```

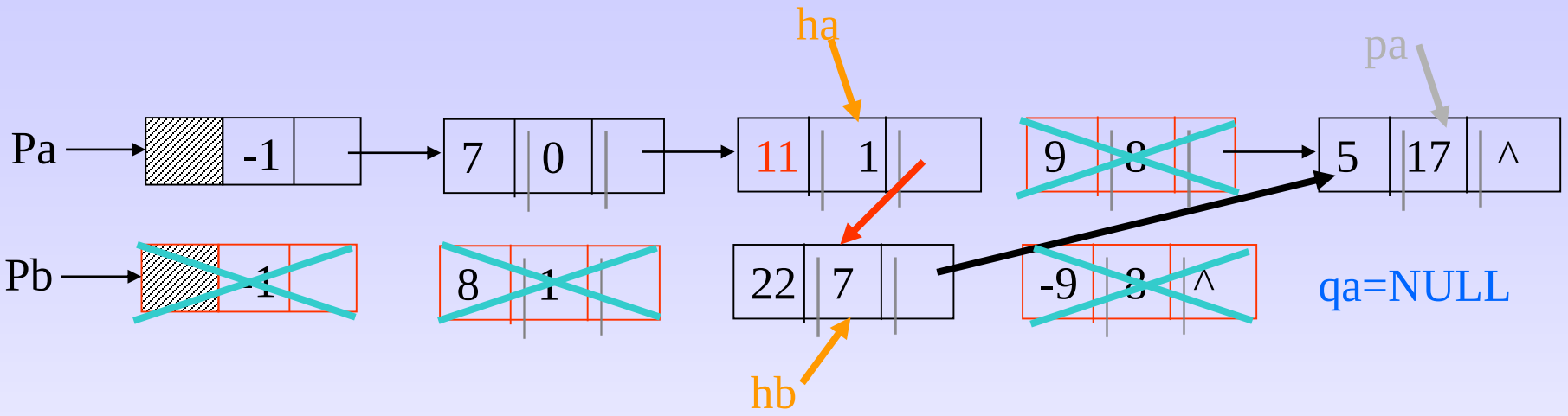
2.4 一元多项式的表示及相加

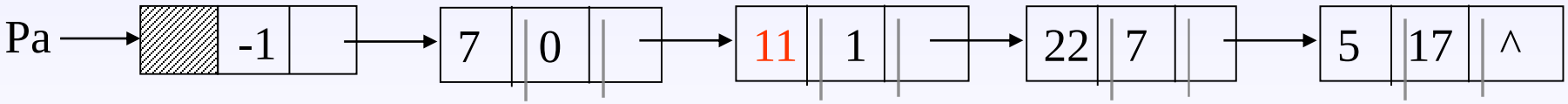



```
switch (*cmp(a,b)){
                case -1          // a中的指数小
                    ha = qa; qa = NextPos(Pa, qa); break;
                case 0           // a和b的指数相等
                    sum = a.coef + b.coef;
                    if (sum != 0.0)   // 修改多项式PA中当前节点的系数值
                        SetCurElem(qa, 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
                }
```

```
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

第三章 栈和队列

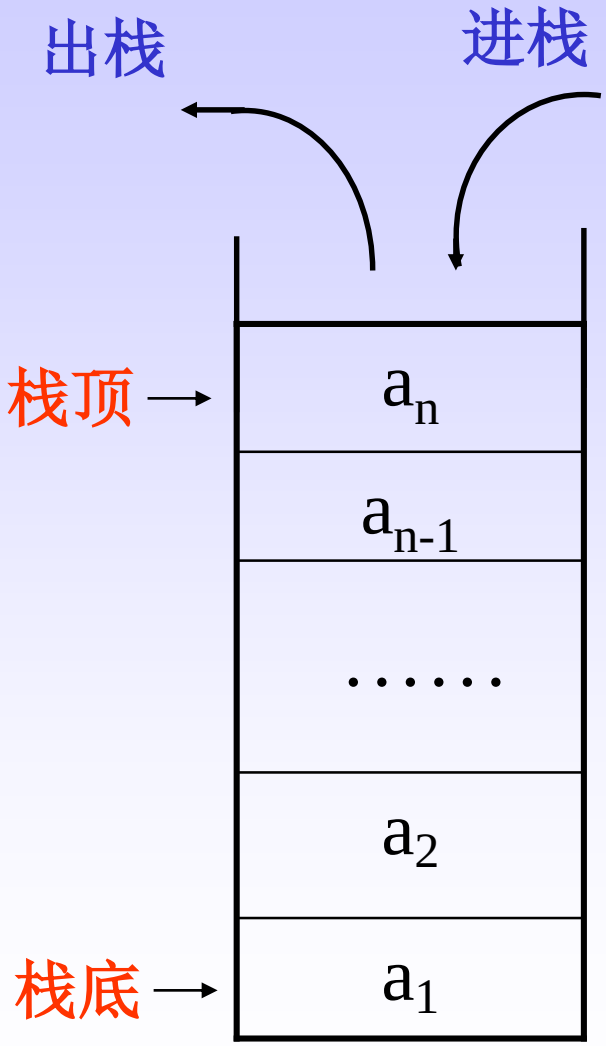
- 3.1、栈
- 3.2、栈的应用举例
- 3.3、队列

3.1 栈(stack)

3.1.1 定义

栈(Stack)是限制在表的一端进行插入和删除运算的线性表，通常称插入、删除的这一端为栈顶(Top)，另一端为栈底(Bottom)。当表中没有元素时称为空栈。

假设栈S=(a1，a2，a3，…an)，则a1称为栈底元素，an为栈顶元素。栈中元素按a1，a2，a3，…an的次序进栈，退栈的第一个元素应为栈顶元素。换句话说，栈的修改是按后进先出的原则进行的。因此，栈称为后进先出表（Last In First Out，LIFO）。



3.1 栈(stack)

- ADT Stack {

  数据对象：D＝{ ai | ai ∈ElemSet, i=1,2,...,n, n≥0 }

  数据关系：R1＝{ <ai-1 ,ai >| ai-1, ai ∈D, i=2,...,n }

  约定a_n端为栈顶，a_1端为栈低

  基本操作：

  InitStack (&S) //构造一个空栈

  DestroyStack (&S) //销毁栈

  ClearStack (&S) //将S清为空栈

  StackEmpty(S) //判断是否为空栈，是则返回True

  StackLength(S) //返回栈的长度

  GetTop (S, &e) // 返回栈顶元素

  Push (&S, e) //插入元素e为新的栈顶元素

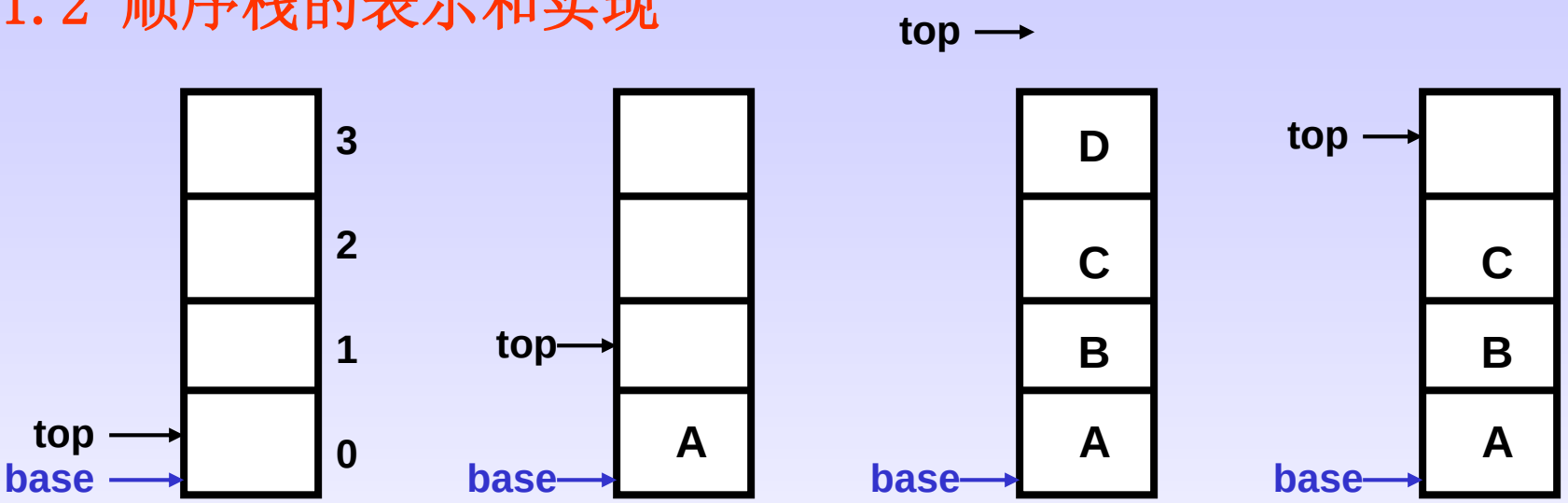
  Pop (&S, &e) // 删除栈顶元素，并用e返回

  StackTraverse(S, visit()) //对每个元素都调用visit函数，如调用失败，则操作失效

  }

3.1 栈

3.1.2 顺序栈的表示和实现



利用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素，同时附设指针top指示栈顶元素在顺序栈的位置

```
typedef struct {
    ElemType *base;
    ElemType *top;
    int      stacksize;
} SqStack;
```

注意：因为 **base == top** 是栈空标志，所以 **top** 指针只能指示真正的栈顶元素之上的数组元素的下标地址。否则造成矛盾。

栈满时的处理方法：

1、提示出错，返回操作系统。

2、分配更大的空间。

3.1 栈

1 顺序栈的初始化实现：

```
#define STACK_INIT_SIZE   100;    //基本容量
#define STACK_INCREMENT 10;       //增量
status InitStack(SqStack &s)
{ s.base=(SElemType *)malloc
            (STACK_INIT_SIZE*sizeof(SElemType));
  if ( !s.base )   exit ( OVERFLOW ) ;
  s.top = s.base ;   // 空栈标志
  s.stacksize = STACK_INIT_SIZE;
  return OK;
} // InitStack;
```

3.1 栈

2 顺序栈的 Push 操作实现：

```
Status Push (SqStack &s, SElemType e)
{  if ( s.top - s.base >= s.stacksize )  // 判断是否栈满
   { s.base=(SElemType *) realloc( s.base,
            (s.stacksize+STACK_INCREMENT)*sizeof(SElemType));
     if ( !s.base )   exit ( OVERFLOW );
     s.top = s.base + s.stacksize ;
     s.stacksize  += STACK_INCREMENT;
   }
   *s.top ++=  e;    // 相当于： * s.top = e;  s.top ++ 两条指令;
   return OK;
}  // Push;
```

3.1 栈

3 顺序栈的 pop 操作实现：

```
Status Pop (SqStack &s, SElemType &e)
{    if ( s.top == s.base ) // 判断是否栈空
         return ERROR;
     s.top--;
     e = *s.top
     return OK ;
} // Pop
```

→ e = * - - s.top

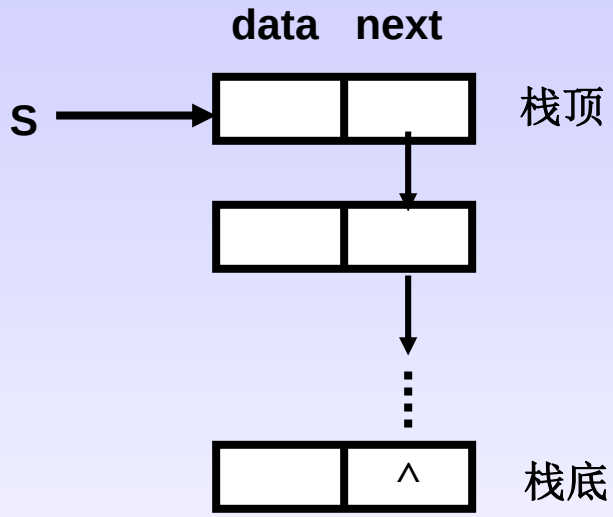
Note:

e = * (s.top - 1) → GetTop 操作

3.1 栈

3.1.2 链式栈的表示和实现

```
typedef struct Snode {
        SElemType          data;
        struct   Snode     *next;
} Snode,  *LinkStack;
```



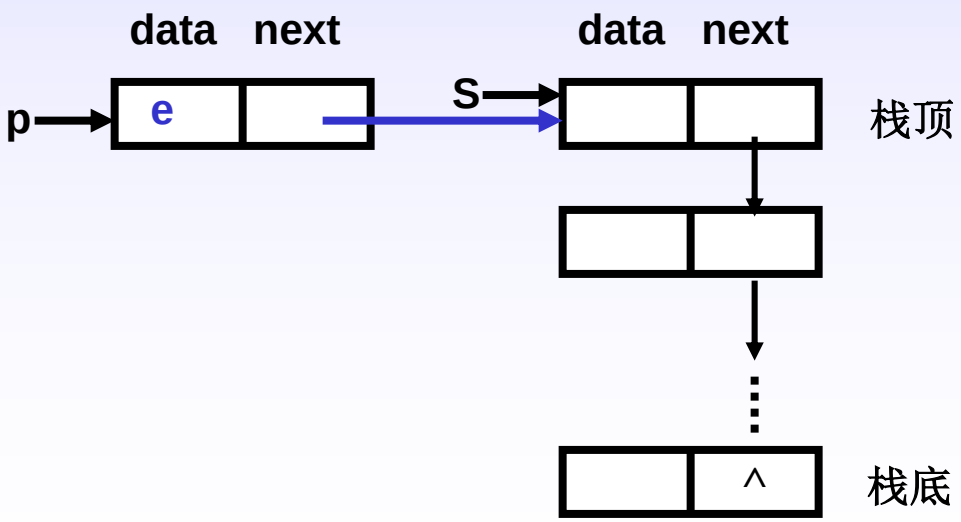
1 链式的栈的初始化实现：

```
void InitlinkStack(LinkStack &s)
{ s = NULL; }  // InitlinkStack，没有头结点，顶，底？
```

3.1 栈

2 链式栈的 Push 操作实现：

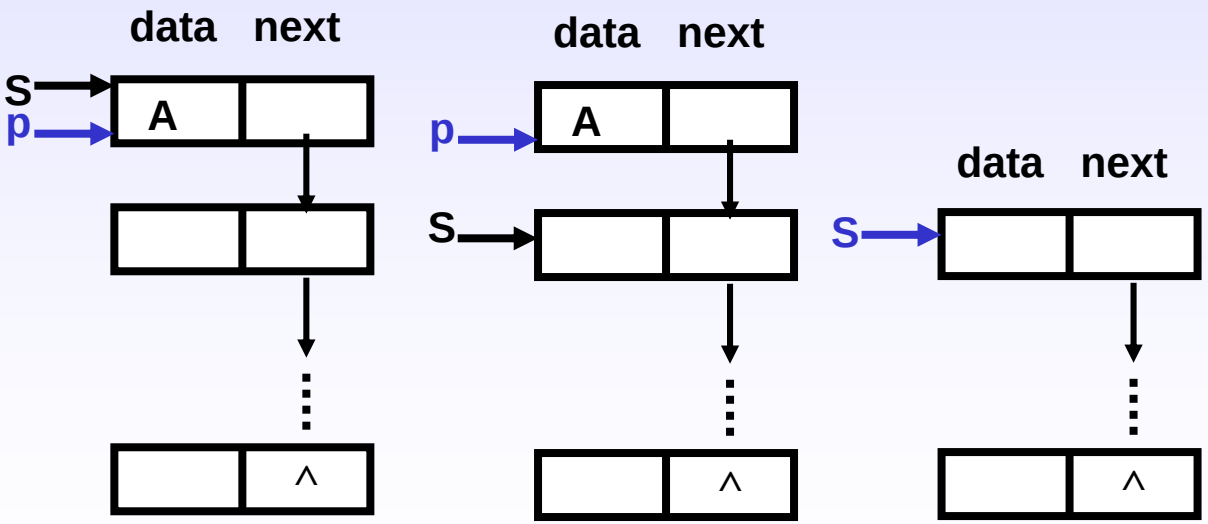
```
Status Push(LinkStack &s, SElemType e)
{  p = ( Snode * ) malloc (sizeof(Snode) );
   p->data = e;    p->next = s;
   s = p;
   return OK;
 }  // Push;
```



3.1 栈

3 链式栈的 Pop 操作实现：

```
Status Pop(LinkStack &s, SElemType & e)
{  if ( !s )
        return ERROR;
   e = s->data;
   p = s;
   s = s->next;
   free(p);
   return OK;
 }  // Pop;
```



3.2 栈的应用--数制转换

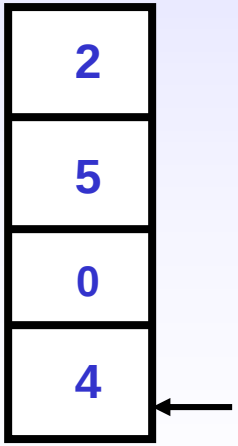
公式： **N=(N div d)*d + N mod d** （div为整除, mod为求余）

```
void conversion ( )
{  InitStack( S );
   scanf( "%d", &N );
   while (N)
   {  Push(S, N%8);
      N = N/8;
   }
   while ( !Stackempty(S))
   {  Pop(S, e);
      printf ("%d" ,  e );
   }
}
```

例如10 进制和 8 进制之间的数的转换。

$(1348)_{10} = 8^3 * a_3 + 8^2 * a_2 + 8 * a_1 + 8^0 * a_0$

N	N div 8	N % 8	
1348	168	4	a0
168	21	0	a1
21	2	5	a2
2	0	2	a3



3.2 栈的应用--括号匹配检查

括号匹配检查

括号匹配的就近原则：

一定是最后一个左括号同当前的右括号进行匹配的。

3.2 栈的应用--括号匹配检查

```
Status  clarity(Sqlist  L)  // L中存放需要检查的表达式
{  Initstack(S);
   for(i=1; i<=Listlength(L); i++)
   {  GetElem(L, i, e1);
      if(( e1 == '(' ) || ( e1 == '[' ))   Push(S, e1); //左括号入栈
      else if(( e1 == ')' ) || ( e1==']' ))   //右括号出栈
           if(( Pop(S, e2) == ERROR )   // 左括号少于右括号
              || ( e1== ')'  && e2 != '(' ) // 左右括号不匹配
             || ( e1== ']'  && e2 != '[' ) )
            return ERROR;
   }
   if  StackEmpty(S)   return OK;
   else  return ERROR;                    // 左括号多于右括号
}
```

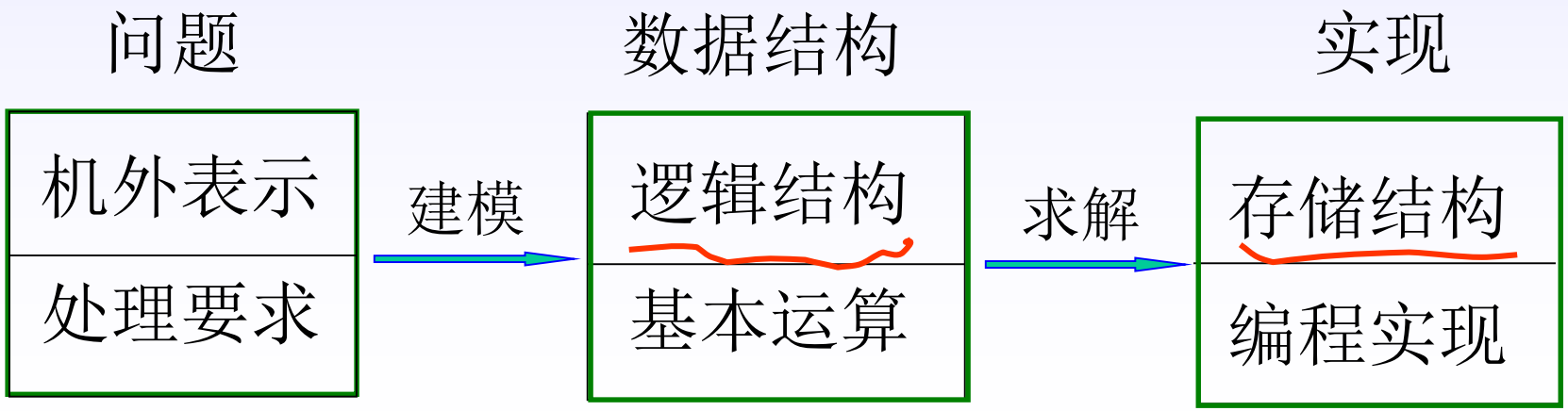
3.2 栈的应用--括号匹配检查

```
Status  clarity(Sqlist  L)
{  Initstack(S);
   for(i=1; i<=Listlength(L); i++)
   {  GetElem(L, i, e1);
      if(( e1 == '(' ) || ( e1 == '[' ))   Push(S, e1);
      else if(( e1 == ')' ) || ( e1==']' ))
            if(( Pop(S, e2) == ERROR )
               || ( e1== ')'  && e2 != '(' )
              || ( e1== ']'  && e2 != '[' ) )
             return ERROR;
   }
   if  StackEmpty(S)   return OK;
   else  return ERROR;
}
```

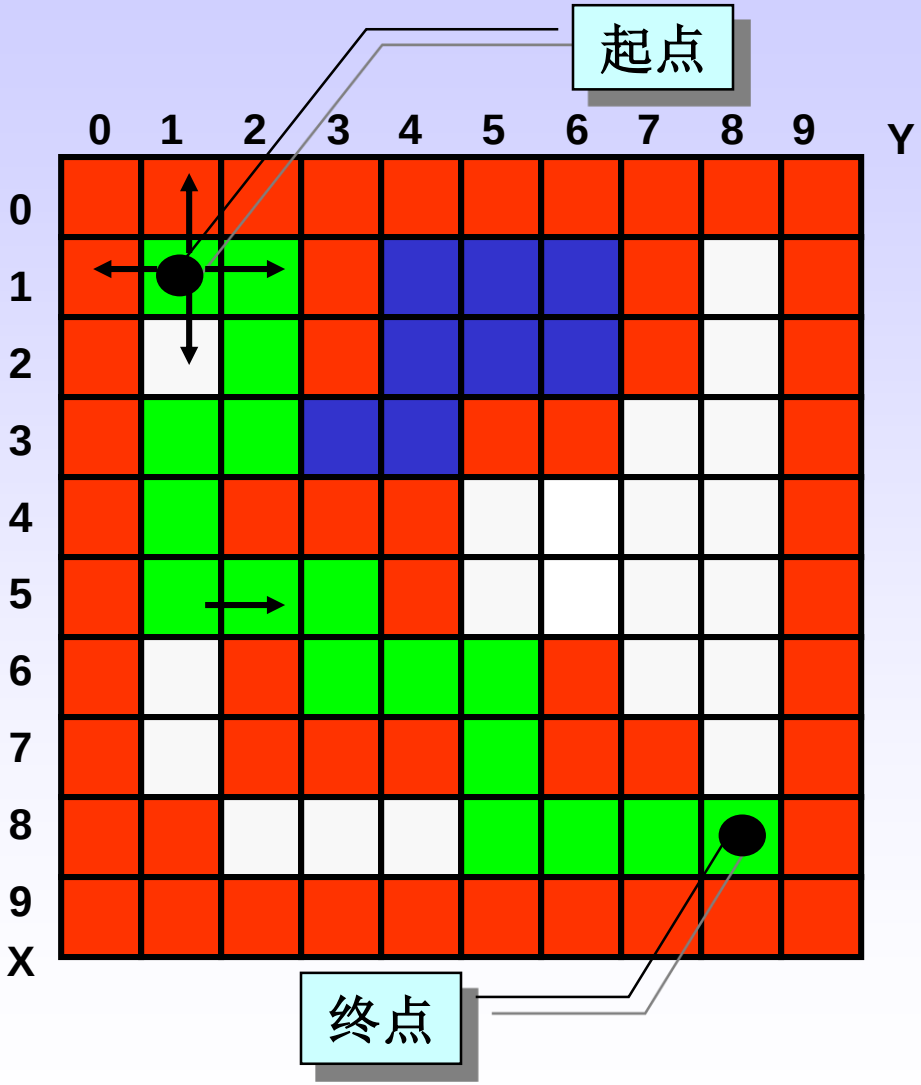
1.2 什么是数据结构

计算机求解问题的步骤：

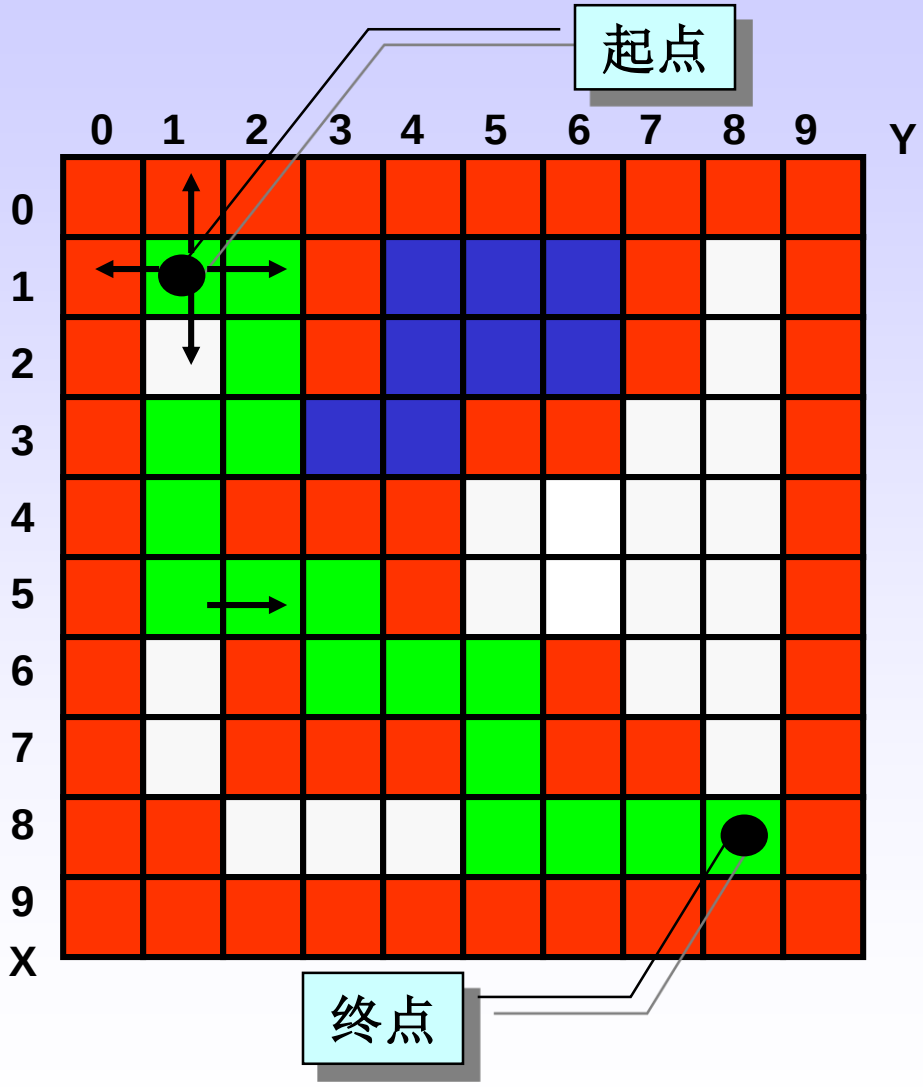
- 分析问题；
- 建立求解问题的数据结构并设计算法—通过算法来表示对象数据及其相互关系；
- 实现：编制程序模拟对象领域中的求解过程。



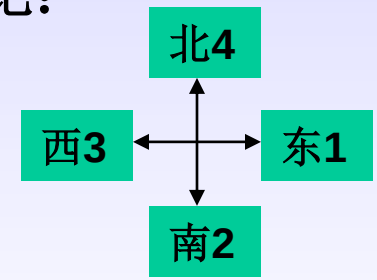
3.2 栈的应用——求从起点到终点的简单路径



3.2 栈的应用——求从起点到终点的简单路径



- 方格的四种类型：
  - ·非墙且未经试探的方格
  - ·墙
  - ·已在路径上的方格
  - ·已试探过的无发展前途的方格
- 方向标记：



- 起点：(x=1,y=1);

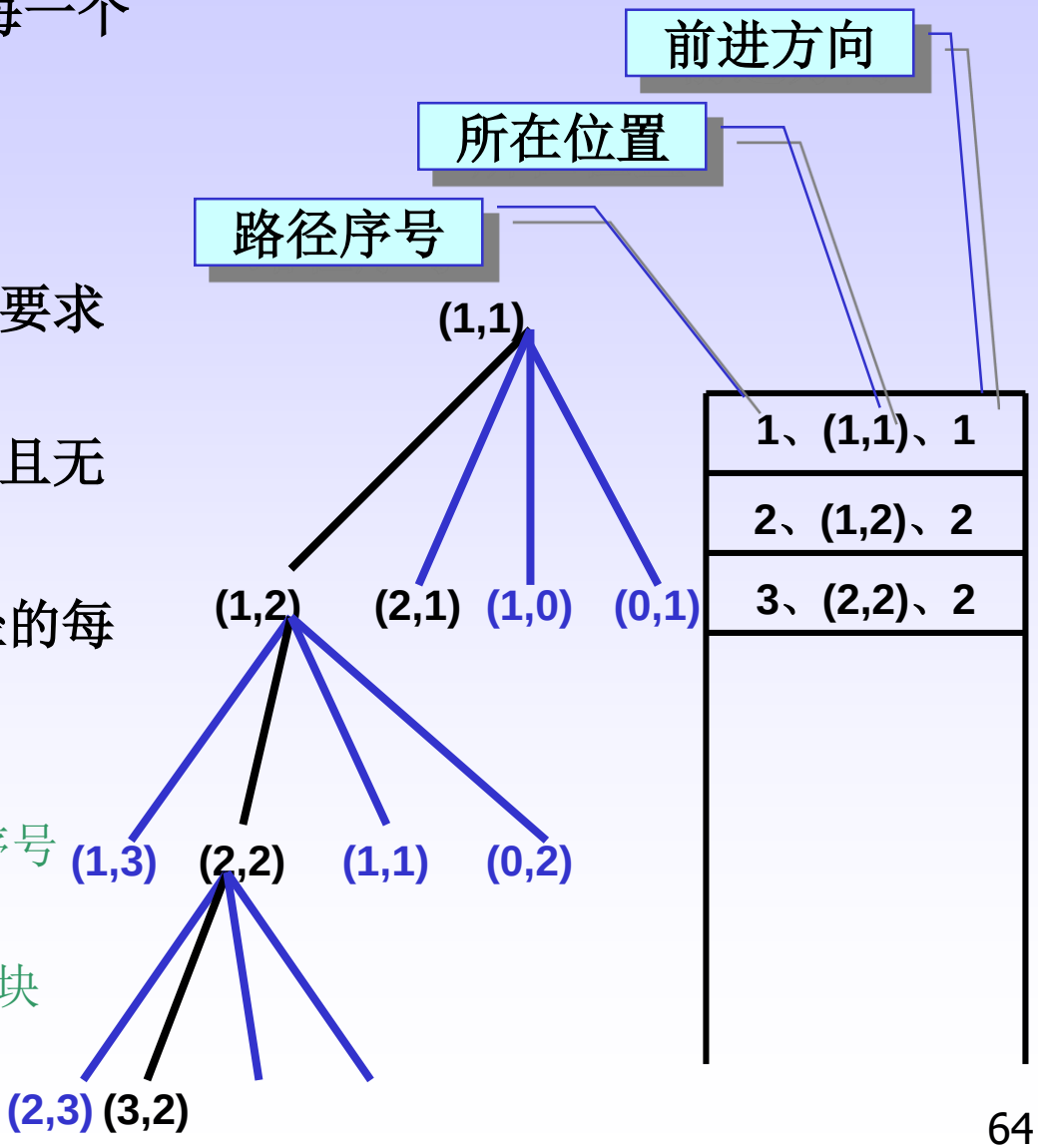
  东(1,2) 南(2,1) 西(1,0) 北(0,1)

演示

3.2 栈的应用--求从起点到终点的简单路径

- 试探方法： 穷举求解，试探每一个可能的方向。
- 可能的方向：
  - 非墙新方格
  - 不是在路径上的方格（因为要求从起点到终点的简单路径）
  - 不是曾经纳入过路径的方格且无发展前途的
- 堆栈中记录的数据（组成路径的每个点）类型：

```
typedef struct
 { int        step; 块在路经的序号
   PosType seat; 块的坐标位置
   int        di; 从此块走向下一块
                  的方向
 } ElemType;
```



3.2 栈的应用--求从起点到终点的简单路径

- 坐标位置类型：

```
typedef struct
 { int  x, y;
 } PosType;
```

- 迷宫类型：

```
typedef struct
 { int   m, n;
    char arr[MAX][MAX];
 } MazeType;
```

```
Status Pass(PosType curpos )      //判断当前位置能否通过
 { if( maze.arr[curpos.x][curpos.y]=='W' )  return TRUE;
    else  return FALSE;
 }
void FootPrint(PosType curpos )  // 标记已访问过的位置
{ maze.arr[curpos.x][curpos.y]='G';
}
```

3.2 栈的应用--求从起点到终点的简单路径

```
void MarkPrint(PosType curpos )    // 设置非路径标志,
{  maze.arr[curpos.x][curpos.y]='B';
}
PosType NextPos( PosType curpos, int dir ) //返回下一个方向的位置
{  PosType  cpos=curpos;
   switch(dir){
         case 1: cpos.y+=1; break;
         case 2: cpos.x+=1; break;
         case 3: cpos.y-=1;  break;
         case 4: cpos.x-=1;  break;
   }
   return(cpos);
}
```

```
Status MazePath ( MazeType maze, PosType start, PosType end  )
{  Initstack(S);  curpos = start; curstep = 1;
   do{ if ( Pass(curpos) )                    // 判断当前位置能否通过
       { FootPrint(curpos);                   // 标记已访问过的位置
         e = ( curstep,curpos,1 ); Push( S, e );  // 加入路径
         if  ( curpos == end  )  return ( TRUE  );  // 已到达终点
         curpos = NextPos( curpos, 1);  // 得到东邻位置，注意1代表东。
         curstep ++;
       } // if
       else {  // 当前位置不能通过
       if  ( !StackEmpty( S ) ) {
         Pop( S,e );                          // 退回到上次来的位置
        while ( e.di == 4 && !StackEmpty(S) )  // 当前位置不在路径上
          { MarkPrint(maze, e.seat); Pop(S,e); }  // 设置非路径标志, 后退一步
        if ( e.di < 4) { e.di++; Push(S,e); curpos = NextPos(e.seat, e.di); }
       }  // 在下一个方向继续探索
   } while ( !StackEmpty(S));
   return ( FALSE );  //  不存在从起点到终点的路径
} // MazePath
```