第二章 线性表

- 线性表及其基本运算
- 数组
- 线性表的顺序表示——顺序表
- 线性表的链式表示——链表
- 进阶导读
- 本章小结

2.1 线性表及其基本运算

- 线性表的定义和特点
  - 定义 n (≥ 0)个数据元素的有限序列，记作

    $$(k_0, k_1, \ldots, k_i, \ldots, k_{n-1})$$

    k_i是表中数据元素，n是表长度。
  - 原则上讲，线性表中表元素的数据类型可以不相同，但采用的存储表示可能会对其有限制。

线性表的特点

- 除第一个元素外，其它每一个元素有一个且仅有一个直接前驱。
- 除最后一个元素外，其它每一个元素有一个且仅有一个直接后继。

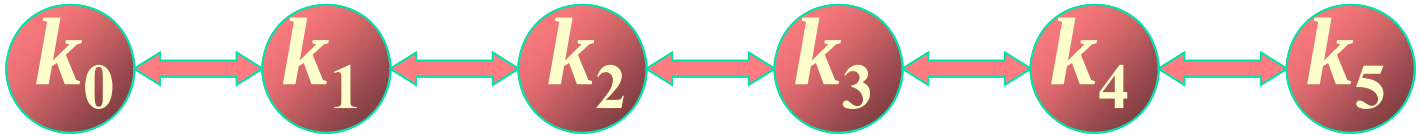


表2-1 一周内每天的平均气温记录表

星期	Mon	Tue	Wed	Thu	Fri	Sat	Sun
温度	15.5	16.0	16.5	15.7	15.0	16.1	16.4

表2-2 一个企业的职工工资表

职工号	姓名	性别	年龄	工资
001	Wang	male	35	160.5
002	Cai	male	32	150.00
003	Zhang	female	28	130.00
… …	… …	… …	… …	… …

线性表的操作

- 计算表的长度n；
- 从左到右（或从右到左）遍历表的元素；
- 访问第i个元素，$0 \le i < n$；
- 将新值赋予第i个元素，$0 \le i < n$；
- 将新元素插入第i个位置，$0 \le i < n$，使原来的第 i, $i+1, \ldots, n-1$个元素变为第$i+1, i+2, \ldots, n$个元素；
- 删除第i个元素，$0 \le i < n$，使原来的第$i+1, i+2, \ldots, n-1$个元素变为第$i, i+1, \ldots, n-2$个元素。

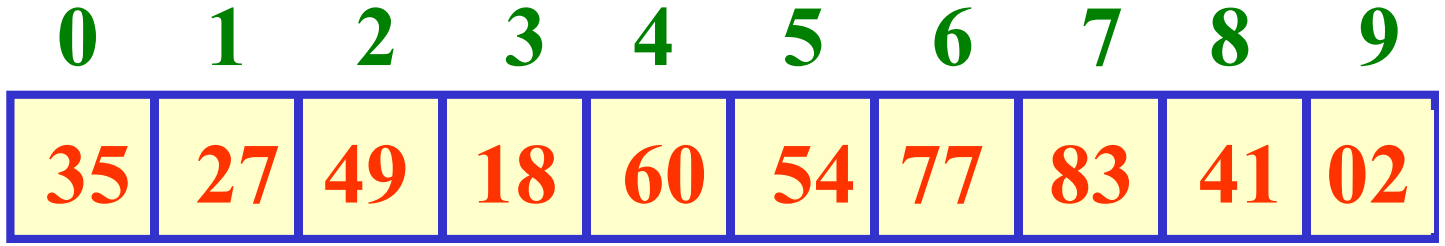
2.2 数组

一维数组

- 定义

  相同类型的数据元素的集合。

- 一维数组的示例



- 在高级语言中的一维数组只能按元素的下标直接存储和访问数组元素。

数组的定义和初始化

```
class Array {
    int e;
public:
    Array( ) { e = 0; }
    Array(int value) { e = value; }
    int Get_Value( ) { return e; }
};
```

```
int main( ) {
    Array array1[3]={1, 2, 3}, *array2;
    for (int i=0; i<3; i++)
        cout<<array1[i].Get_Value( )<<"\n" ;  //静态
    array2=array1;
    for (int i=0; i<3; i++) {
        cout<<array2->Get_Value( )<<"\n" ;  //动态
        array2++;
    }
    return 0;
}
```

程序2-1 一维数组(Array)类的定义

```
template <class Type> class Array {
  public:
    Array(int Size=DefaultSize);  //构造函数
    Array(const Array <Type> &x);  //复制构造函数
    ~Array( ) { delete [ ] elements; }  //析构函数
    Array <Type> & operator = (const Array <Type> &A);
    //数组复制
    Type & operator [ ] (int i);  //取下标为i的数组元素
    Type * operator * ( ) const { return elements; }
    //指针转换
    int Length( ) const { return ArraySize; }  //取数组长度
    void Resize(int sz);  //修改数组长度
  private:
     Type *elements;  //动态数组
     int ArraySize;  //数组元素个数
     void Get_Array( );  //动态分配数组元素存储空间
};
```

程序2-2 数组的构造函数

```
template <class Type> void Arrray <Type> :: Get_Array( )
{  //私有函数：动态分配数组的存储空间
        elements=new Type [ArraySize];
        //创建动态数组
        if (elements==0)
        {
                cerr<<“Memory Allocation Error”<<endl;
                ArraySize=0;
                return;
        }
}
```

```
template <class Type> void Arrray <Type> ::
Array(int Size)
{ //带形参构造函数：建立最大长度为Size的数组
        if (Size<=0)
        { //参数检查
                cerr<<“Invalid Array Size”<<endl;
                ArraySize=0; return;
        }
        ArraySize=Size; //数组长度
        Get_Array( ); //创建数组
}
```

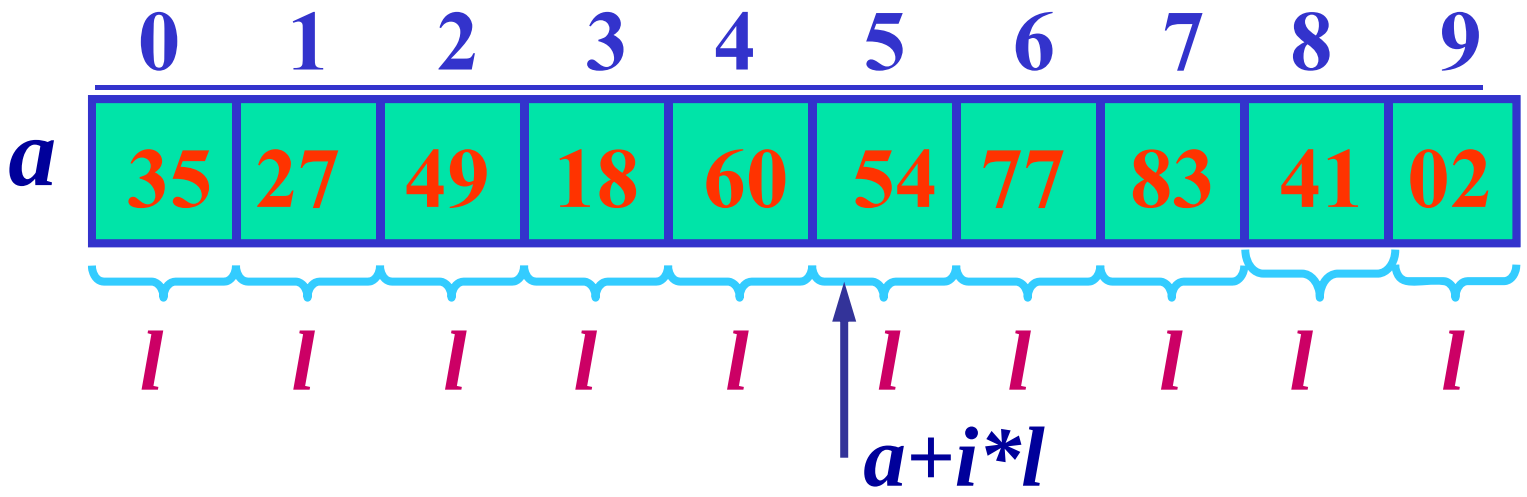
```
template <class Type> void Arrray <Type> ::
Array(const Array <Type> &x)
{ //复制构造函数：复制数组x以建立当前新数组
        int n=x.ArraySize;  ArraySize=n;  //目标数组的长度
        elements=new Type [n];
        //为目标数组动态分配存储空间
        if (elements==0)
        {
                cerr<<"Memory Allocation Error"<<endl;
                ArraySize=0;  return;
        }
        Type *srcptr=x.elements;  //源数组首地址
        Type *destptr=elements;  //目标数组首地址
        while (n--)  *destptr++=*srcptr++;  //传送，复制
}
```

数组的顺序存储方式

- 把数组中各个元素的值按一定的次序存放在计算机的一组连续存储单元中。
- 优点在于可以随机存取或者修改数组元素的值。
- 只要知道数组元素的下标值，就可以按相应的地址计算公式求得该元素的存放地址。

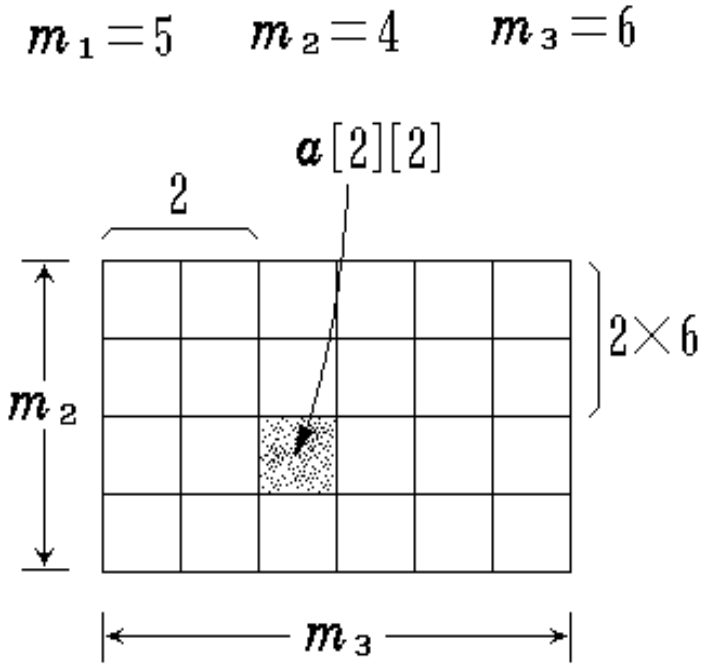
- 一维数组

$$LOC(i) = \begin{cases} a, & i=0 \\ LOC(i-1)+l=a+i*l, & i>0 \end{cases}$$



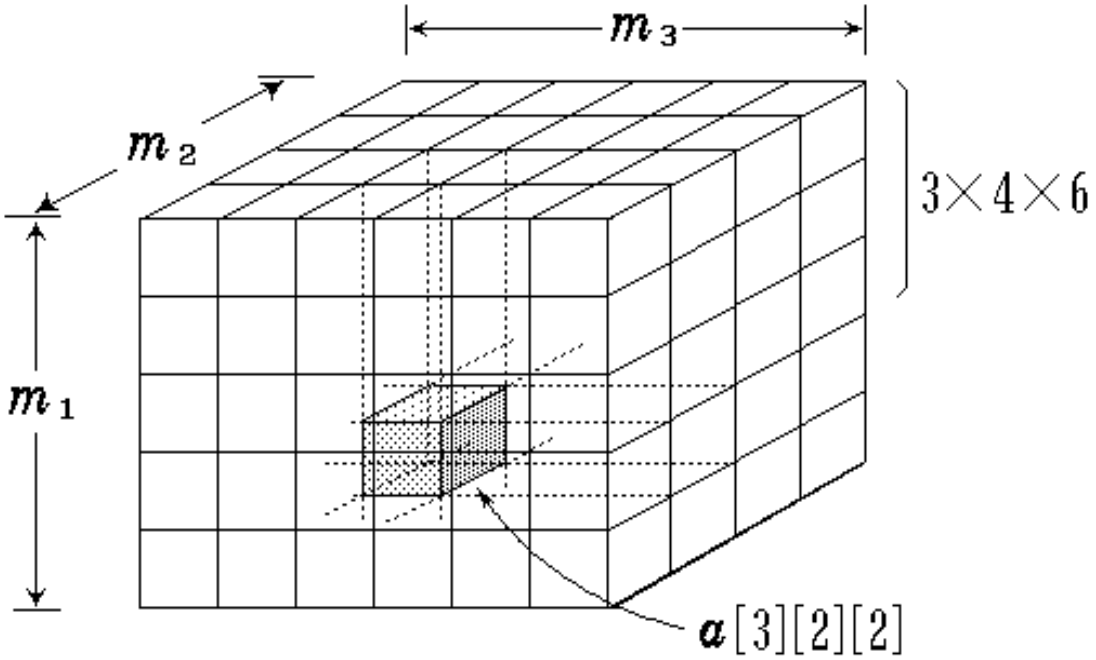
$$LOC(i)=LOC(i-1)+l=a+i*l$$

二维数组



行向量 下标 *i*
列向量 下标 *j*

三维数组



页向量 下标 *i*
行向量 下标 *j*
列向量 下标 *k*

- 二维数组

$$a=\begin{pmatrix} a[0][0] & a[0][1] & \cdots & a[0][t_2-1] \\ a[1][0] & a[1][1] & \cdots & a[1][t_2-1] \\ a[2][0] & a[2][1] & \cdots & a[2][t_2-1] \\ \vdots & \vdots & \ddots & \vdots \\ a[t_1-1][0] & a[t_1-1][1] & \cdots & a[t_1-1][t_2-1] \end{pmatrix}$$

行优先存放：

设数组开始存放位置 **LOC(0, 0)=&*a*[0][0]**，每个元素占用***s***个存储单元

LOC(*i*, *j*)=&*a*[0][0]+(*it_2+*j*)**s***

- 二维数组

$$a=\begin{pmatrix} a[0][0] & a[0][1] & \cdots & a[0][t_2-1] \\ a[1][0] & a[1][1] & \cdots & a[1][t_2-1] \\ a[2][0] & a[2][1] & \cdots & a[2][t_2-1] \\ \vdots & \vdots & \ddots & \vdots \\ a[t_1-1][0] & a[t_1-1][1] & \cdots & a[t_1-1][t_2-1] \end{pmatrix}$$

列优先存放：

设数组开始存放位置 LOC(0, 0)=&*a*[0][0]，每个元素占用*s*个存储单元

LOC(*i*, *j*)=&*a*[0][0]+(*j***t*$_1$+*i*)**s*

- 三维数组
  - 各维元素个数为t_1, t_2, t_3
  - 下标为i, j, k的数组元素的存储地址：

（按页/行/列存放）

$$LOC(i,j,k)=\&a[0][0][0]+(i*t_2*t_3+j*t_3+k)*s$$

$i*t_2*t_3$：前i页总元素个数

$j*t_3$：第i页的前j行总元素个数

k：第j行前k列元素个数

- ***n*** 维数组
  - 各维元素个数为 $t_1, t_2, t_3, \ldots, t_n$。
  - 下标为$i_1, i_2, i_3, \ldots, i_n$的数组元素的存储地址：

$$LOC(i_1, i_2, \ldots, i_n)=\&a[0][0]\ldots[0]+(i_1*t_2*t_3*\ldots*t_n+i_2*t_3*t_4*\ldots*t_n+\ldots\ldots+i_{n-1}*t_n+i_n)*s$$

$$=\&a[0][0]\ldots[0]+s*\sum_{j=1}^{n} i_j * \prod_{k=j+1}^{n} t_k$$

随堂练习

例1：二维数组A的每个元素是由6个字符组成的串，其行下标i=0, 1, ..., 8，列下标j=1, 2, ..., 10。若A按行先存储，元素A[8, 5]的起始地址与当A按列先存储时的元素的哪一个元素的起始地址相同。设每个字符占一个字节。

例2：若三维数组M[2..3, -4..2, -1..4]中，每个元素占用2个存储单元，起始地址为100，则如果按页优先顺序存储，M[3][-3][3]的存储地址为_____。

例1：二维数组A的每个元素是由6个字符组成的串，其行下标i=0, 1, ..., 8，列下标j=1, 2, ..., 10。若A按行先存储，元素A[8, 5]的起始地址与当A按列先存储时的元素的哪一个元素的起始地址相同。设每个字符占一个字节。

设二维数组A[c1..d1, c2..d2]，每个数据元素占L个字节，则：

行优先存储地址计算公式：

$$Loc(aij)=Loc(ac1c2)+[(i-c1)*(d2-c2+1)+(j-c2)]*L$$

列优先存储地址计算公式：

$$Loc(aij)=Loc(ac1c2)+[(j-c2)*(d1-c1+1)+(i-c1)]*L$$

$$(8-0)*(10-1+1)+(5-1)=(j-1)*(8-0+1)+(i-0)$$

即$9(j-1)+(i-0)=84$

推出A[3, 10]。

例2：若三维数组M[2..3, -4..2, -1..4]中，每个元素占用2个存储单元，起始地址为100，则如果按页优先顺序存储，M[3][-3][3]的存储地址为（204）。

设三维数组A[c1..d1, c2..d2, c3..d3]，每个数据元素占L个字节，则：

页优先存储地址计算公式：

$$Loc(aijk)=Loc(ac1c2c3)+[(i-c1)*(d2-c2+1)(d3-c3+1)+(j-c2)*(d3-c3+1)+(k-c3)]*L$$

行优先存储地址计算公式：

$$Loc(aijk)=Loc(ac1c2c3)+[(j-c2)*(d1-c1+1)(d3-c3+1)+(i-c1)*(d3-c3+1)+(k-c3)]*L$$

列优先存储地址计算公式：

$$Loc(aijk)=Loc(ac1c2c3)+[(k-c3)*(d1-c1+1)(d2-c2+1)+(i-c1)*(d2-c2+1)+(j-c1)]*L$$

特殊矩阵

- 特殊矩阵是指非零元素或零元素的分布有一定规律的矩阵。
- 特殊矩阵的压缩存储主要是针对阶数很高的特殊矩阵。为节省存储空间，对可以不存储的元素，如零元素或对称元素，不再存储。
  - 对称矩阵
  - 带状矩阵

对称矩阵的压缩存储

- 设有一个n×n的对称矩阵A：

$$A=\begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0n-1} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1n-1} \\ a_{20} & a_{21} & a_{22} & \cdots & a_{2n-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n-10} & a_{n-11} & a_{n-12} & \cdots & a_{n-1n-1} \end{bmatrix}$$

在矩阵中，$a_{ij}=a_{ji}$

- 为节约存储，只存对角线及对角线以上的元素，或者只存对角线或对角线以下的元素。前者称为上三角矩阵，后者称为下三角矩阵。

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0n-1} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1n-1} \\ a_{20} & a_{21} & a_{22} & \cdots & a_{2n-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n-10} & a_{n-11} & a_{n-12} & \cdots & a_{n-1n-1} \end{bmatrix}$$

下三角矩阵

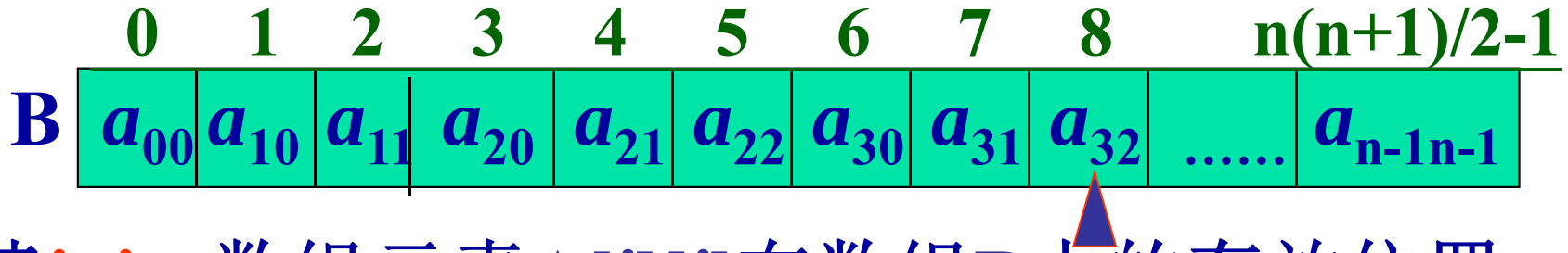
$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0n-1} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1n-1} \\ a_{20} & a_{21} & a_{22} & \cdots & a_{2n-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n-10} & a_{n-11} & a_{n-12} & \cdots & a_{n-1n-1} \end{bmatrix}$$

上三角矩阵

- 把它们按行存放于一个一维数组**B**中， 称之为对称矩阵**A**的压缩存储方式。
- 数组**B**共有$n+(n-1)+\cdots+1=$
  $n*(n+1)/2$个元素。

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0n-1} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1n-1} \\ a_{20} & a_{21} & a_{22} & \cdots & a_{2n-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n-10} & a_{n-11} & a_{n-12} & \cdots & a_{n-1n-1} \end{bmatrix}$$

下三角矩阵



若$i \geq j$，数组元素A[i][j]在数组B中的存放位置为$1+2+\cdots+i+j=\underbrace{(i+1)*i/2}_{\text{前}i\text{行元素总数}}+\underbrace{j}_{\text{第}i\text{行第}j\text{个元素前元素个数}}$

若$i<j$，数组元素A[i][j]在矩阵的上三角部分，在数组B中没有存放，可以找它的对称元素A[j][i]=$j*(j+1)/2+i$。

若已知某矩阵元素位于数组B的第k个位置，可寻找满足

$$i(i+1)/2 \le k < (i+1)*(i+2)/2$$

的i，此即为该元素的行号。

$$j=k-i*(i+1)/2$$

此即为该元素的列号。

例，当$k=8$，$3*4/2=6 \le k < 4*5/2=10$，取$i=3$，则 $j=8-3*4/2=2$。

$$n=4 \qquad \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix}$$

上三角矩阵

	0	1	2	3	4	5	6	7	8	9
B	a_{00}	a_{01}	a_{02}	a_{03}	a_{11}	a_{12}	a_{13}	a_{22}	a_{23}	a_{33}

若$i \le j$，数组元素A[*i*][*j*]在数组B中的存放位置为

$$\underbrace{n+(n-1)+(n-2)+\cdots+(n-i+1)}_{\text{前}i\text{行元素总数}}+\underbrace{j-i}_{\text{第}i\text{行第}j\text{个元素前元素个数}}$$

若$i \le j$，数组元素A[i][j]在数组B中的存放位置为

$$n+(n-1)+(n-2)+\cdots+(n-i+1)+j-i$$

$$=(2*n-i+1)*i/2+j-i$$

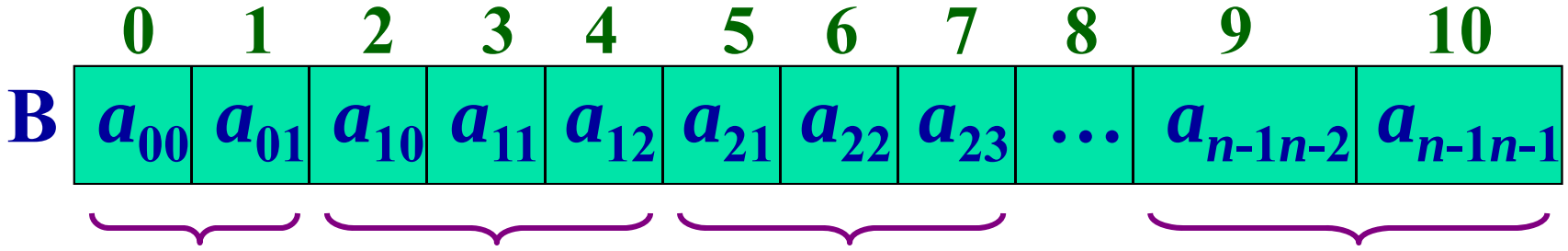
$$=(2*n-i-1)*i/2+j$$

若$i>j$，数组元素A[i][j]在矩阵的下三角部分，在数组B中没有存放。因此，找它的对称元素A[j][i]。

A[j][i]在数组B的第$(2*n-j-1)*j/2+i$的位置中找到。

带状（三对角）矩阵的压缩存储

$$A=\begin{bmatrix} a_{00} & a_{01} & 0 & 0 & 0 & 0 \\ a_{10} & a_{11} & a_{12} & 0 & 0 & 0 \\ 0 & a_{21} & a_{22} & a_{23} & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & a_{n-2n-3} & a_{n-2n-2} & a_{n-2n-1} \\ 0 & 0 & 0 & 0 & a_{n-1n-2} & a_{n-1n-1} \end{bmatrix}$$



- 三对角矩阵中除主对角线及在主对角线上下最临近的两条对角线上的元素外，所有其它元素均为**0**，总共有**3*n*-2**个非零元素。
- 将三对角矩阵**A**中三条对角线上的元素按行存放在一维数组**B**中，且a_{00}存放于**B[0]**。
- 在三条对角线上的元素a_{ij}满足

  $0 \le i \le n-1,\ i-1 \le j \le i+1$
- 在一维数组**B**中**A[*i*][*j*]**在第*i*行，它前面有**3*i*-1**个非零元素，在本行中第*j*列前面有**j-i+1**个，所以元素**A[*i*][*j*]**在**B**中位置为 $k=2*i+j$。

- 若已知三对角矩阵中某元素A[*i*][*j*]在数组B[ ]存放于第*k*个位置，则有

  $i=\lfloor (k+1)/3 \rfloor$

  $j=k-2*i$

- 例如，当*k*=8时，

  $i=\lfloor (8+1)/3 \rfloor=3$, $j=8-2*3=2$

  当*k*=10时，

  $i=\lfloor (10+1)/3 \rfloor=3$, $j=10-2*3=4$

随堂练习

例1：将一个A[1..100，1..100]的三对角矩阵，按行优先存入一维数组B[1..298]中，A中元素A[66,65]（即该元素下标i=66，j=65），在B数组中的位置K为_____。

例2：五对角矩阵中k与i、j的关系。

例1：将一个A[1..100，1..100]的三对角矩阵，按行优先存入一维数组B[1..298]中，A中元素A[66,65]（即该元素下标i=66，j=65），在B数组中的位置K为 195 。

例2：五对角矩阵中k与i、j的关系。

$$k=\begin{cases}4(i-1)+j & i=1\\ 4(i-1)+j-1 & 1<i<n\\ 4(i-1)+j-2 & i=n\end{cases}$$

稀疏矩阵 (Sparse Matrix)

$$A_{6\times7}=\begin{pmatrix}0&0&0&22&0&0&15\\0&11&0&0&0&17&0\\0&0&0&-6&0&0&0\\0&0&0&0&0&39&0\\91&0&0&0&0&0&0\\0&0&28&0&0&0&0\end{pmatrix}$$

非零元素个数远远少于矩阵元素个数

稀疏矩阵的定义

- 设矩阵$A_{m\times n}$中有t个非零元素，若t远远小于矩阵元素的总数$m\times n$，则称矩阵A为稀疏矩阵。
- 为节省存储空间，应只存储非零元素。
- 非零元素的分布一般没有规律，应在存储非零元素时，同时存储该非零元素的行下标*row*、列下标*col*、值*value*。
- 每一个非零元素由一个三元组唯一确定：

  (行号 *row*, 列号 *col*, 值 *value*)

稀疏矩阵类定义

```
template <class Type> class SparseMatrix <Type>;
//稀疏矩阵类的前向引用声明
template <class Type> class Trituple
{  //三元组类定义
        friend class SparseMatrix <Type>;
        private:
            int row, col;  //非零元素的行号与列号
            Type data;  //非零元素的值
};
```

```
template <class Type> class SparseMatrix {
//稀疏矩阵类定义
   public:
      SparseMatrix(int MaxRow, int MaxCol);
      //构造函数：建立一个MaxRow行与MaxCol列的稀疏矩阵
      SparseMatrix <Type> Mat_Transpose
                                          (SparseMatrix <Type> b);
      //对*this指示的三元组数组中各个三元组交换其行、列的值，
      //得到其转置矩阵
      SparseMatrix <Type> Mat_Fast_Transpose
                                          (SparseMatrix <Type> b);
      //快速转置
      SparseMatrix <Type> Multiply(SparseMatrix <Type> b);
      //实现两矩阵相乘
    private:
       int Rows, Cols, NonZero_Terms;
       Trituple <Type> SMArray[MaxTerms];
};
```

稀疏矩阵的转置

- 一个$m \times n$的矩阵**A**，其转置矩阵**B**是一个$n \times m$的矩阵，且**A**[*i*][*j*]=**B**[*j*][*i*]。即矩阵**A**的行成为矩阵**B**的列，矩阵**A**的列成为矩阵**B**的行。
- 在稀疏矩阵的三元组表中，非零矩阵元素按行存放。当行号相同时，按列号递增的顺序存放。
- 稀疏矩阵的转置运算要转化为对应三元组表的转置。

稀疏矩阵

$$\begin{pmatrix} 0 & 0 & 0 & 22 & 0 & 0 & 15 \\ 0 & 11 & 0 & 0 & 0 & 17 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 39 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 & 0 \end{pmatrix}$$

	行 (row)	列 (col)	值 (value)
[0]	0	3	22
[1]	0	6	15
[2]	1	1	11
[3]	1	5	17
[4]	2	3	−6
[5]	3	5	39
[6]	4	0	91
[7]	5	2	28

转置矩阵

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 91 & 0 \\ 0 & 11 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 28 \\ 22 & 0 & -6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 17 & 0 & 39 & 0 & 0 \\ 15 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

	行 (row)	列 (col)	值 (value)
[0]	0	4	91
[1]	1	1	11
[2]	2	5	28
[3]	3	0	22
[4]	3	2	−6
[5]	5	1	17
[6]	5	3	39
[7]	6	0	16

用三元组表表示的稀疏矩阵及其转置

	行 (row)	列 (col)	值 (value)
[0]	0	3	22
[1]	0	6	15
[2]	1	1	11
[3]	1	5	17
[4]	2	3	−6
[5]	3	5	39
[6]	4	0	91
[7]	5	2	28

	行 (row)	列 (col)	值 (value)
[0]	0	4	91
[1]	1	1	11
[2]	2	5	28
[3]	3	0	22
[4]	3	2	−6
[5]	5	1	17
[6]	5	3	39
[7]	6	0	16

稀疏矩阵转置算法思想

- 设矩阵列数为*Cols*，对矩阵三元组表扫描*Cols*次，第*k*次检测列号为*k*的项。
- 第*k*次扫描找寻所有列号为*k*的项，将其行号变列号、列号变行号，顺次存于转置矩阵三元组表。

程序2-3 稀疏矩阵的转置

```
template <class Type> SparseMatrix <Type>
SparseMatrix <Type> ::
                Mat_Transpose(SparseMatrix <Type> b)
{ //将矩阵a（由*this指示）转置，结果在稀疏矩阵b中
      int CurrentPosition; //存放位置指针
      b.Rows=Cols; //矩阵b的行数=矩阵a的列数
      b.Cols=Rows; //矩阵b的列数=矩阵a的行数
      b.NonZero_Terms=NonZero_Terms;
       //矩阵b的非零元素个数=矩阵a的非零元素个数
      if (NonZero_Terms>0)
      { //非零元素个数不为零
          CurrentPosition=0; //存放位置指针清零
          for (int k=0; k<Cols; k++)
           //按列号做扫描，做Cols趟
              for (int i=0; i<NonZero_Terms; i++)
               //在数组中寻找列号为k的三元组
```

稀疏矩阵的转置

```
                    if (SMArray[i].col==k)
                    {  //第i个三元组中元素的列号为k
                        b.SMArray[CurrentPosition].row=k;
                        //新三元组的行号
                        b.SMArray[CurrentPosition].col
                                                =SMArray[i].row;
                        //新三元组的列号
                        b.SMArray[CurrentPosition].data
                                                =SMArray[i].data;
                        //新三元组的值
                        CurrentPosition++;
                        //存放指针进1
                    }
                }
            }
        return b;
}
```

快速转置算法

- 设矩阵三元组表总共有*t*项，上述算法的时间代价为**O(*n**t)**。
- 若矩阵有**200**行、**200**列、**10,000**个非零元素，总共有**2,000,000**次处理。
- 为加速转置速度，建立辅助数组**rowSize**和**rowStart**，记录矩阵转置后各行非零元素个数和各行元素在转置三元组表中开始存放位置。
- 扫描矩阵三元组表，根据某项列号，确定它转置后的行号，查**rowStart**表，按查到的位置直接将该项存入转置三元组表中。

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	语 义
rowSize	1	1	1	2	0	2	1	矩阵 A 各列非零元素个数
rowStart	0	1	2	3	5	5	7	矩阵 B 各行开始存放位置

三元组	(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)
行row	0	0	1	1	2	3	4	5
列col	3	6	1	5	3	5	0	2
值value	22	15	11	17	-6	39	91	28

程序2-4 稀疏矩阵的快速转置

```
template <class Type> SparseMatrix <Type>
SparseMatrix <Type> ::
        Mat_Fast_Transpose (SparseMatrix <Type> b)
{ //对稀疏矩阵（由*this指示）进行快速转置
  //结果放在b中
    int i, j;
    int *x=new int [Cols];
    //辅助数组，统计各列非零元素个数
    int *y=new int [Cols];
    //辅助数组，预计转置后各行存放位置
    b.Rows=Cols;
    b.Cols=Rows;
    b.NonZero_Terms=NonZero_Terms;
```

稀疏矩阵的快速转置

```
if (NonZero_Terms>0)
    {
        for (i=0; i<Cols; i++)  x[i]=0;
        //统计矩阵b中第i行非零元素个数
        for (i=1; i<=NonZero_Terms; i++)
            x[SMArray[i].col]++;
        //根据矩阵a中第i个非零元素的列号
        //将x相当该列的计数加1
        y[0]=0;  //计算矩阵b第i行的开始存放位置
        for (i=1; i<=Cols; i++)  y[i]=y[i-1]+x[i-1];
        //y[i]等于矩阵b的第i行的开始存放位置
```

稀疏矩阵的快速转置

```
        for (i=1; i<=NonZero_Terms; i++)
        {  //从a向b传送
           j=y[SMArray[i].col];
           //j为第i个非零元素在b中应存放的位置
           b.SMArray[j].row=SMArray[i].col;
           b.SMArray[j].col=SMArray[i].row;
           b.SMArray[j].data=SMArray[i].data;
           y[SMArray[i].col]++;
           //矩阵b第i行非零元素的存放位置
        }
    }
    delete [ ] x; delete [ ] y;
    return b;
}
```

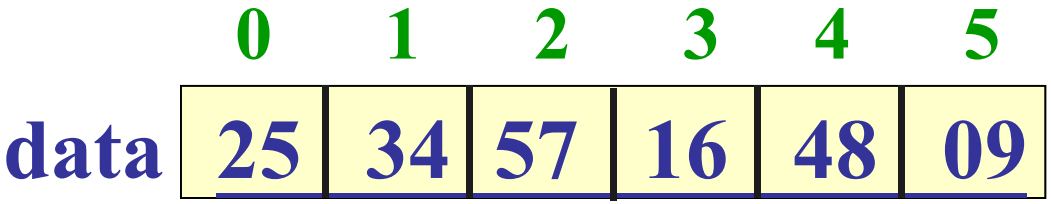
- 在函数中，共有四个循环，它们分别执行*Cols*、*NonZero_Terms*、*Cols*-1和*NonZero_Terms*次，且执行时间分别为O(*Cols*)、O(*NonZero_Terms*)、O(*Cols*-1)和O(*NonZero_Terms*)。
- 而循环外面的几个赋值语句的执行时间为O(1)。所以，该函数的执行时间为：

$$O(1)+O(Cols)+O(NonZero\_Terms)+O(Cols-1)+O(NonZero\_Terms)$$

- 当*NonZero_Terms* = *Rows***Cols*时，函数的执行时间为O(*Cols*+*Rows***Cols*)=O(*Rows***Cols*)，这与不采用三元组数组表示矩阵时的转置时间相同。

2.3 线性表的顺序表示——顺序表

- 顺序表的定义和特点
  - 定义　将线性表中的元素相继存放在一个连续的存储空间中。
  - 可利用一维数组描述存储结构。
  - 特点　线性表的数组存储方式。
  - 限制　所有元素有相同数据类型。



程序2-5 顺序表(SeqList)类定义

```
template <class Type> class SeqList {
    public:
        SeqList(int MaxSize=defaultSize);  //构造函数
        ~SeqList( ) { delete [ ] list; }  //析构函数
        int length( ) const { return last+1; }  //计算表长度
        int Find(Type &x) const;  //查找
        int IsIn(Type &x);  //判断x是否在表中
        int Insert(Type &x, int i);  //插入x在表中第i个位置处
        int Remove(int i);  //删除第i个位置处的表项
        int Next(Type &x);  //寻找x的后继
        int Prior(Type &x);  //寻找x的前驱
        int IsEmpty( ) { return last==-1; }  //判断表空否
        int IsFull( ) { return last==MaxSize-1; }  //判断表满否
        Type Get(int i) {return i<0||i>last?NULL:data[i]; }
        //取第i个元素的值
```

程序2-5 顺序表(SeqList)类定义

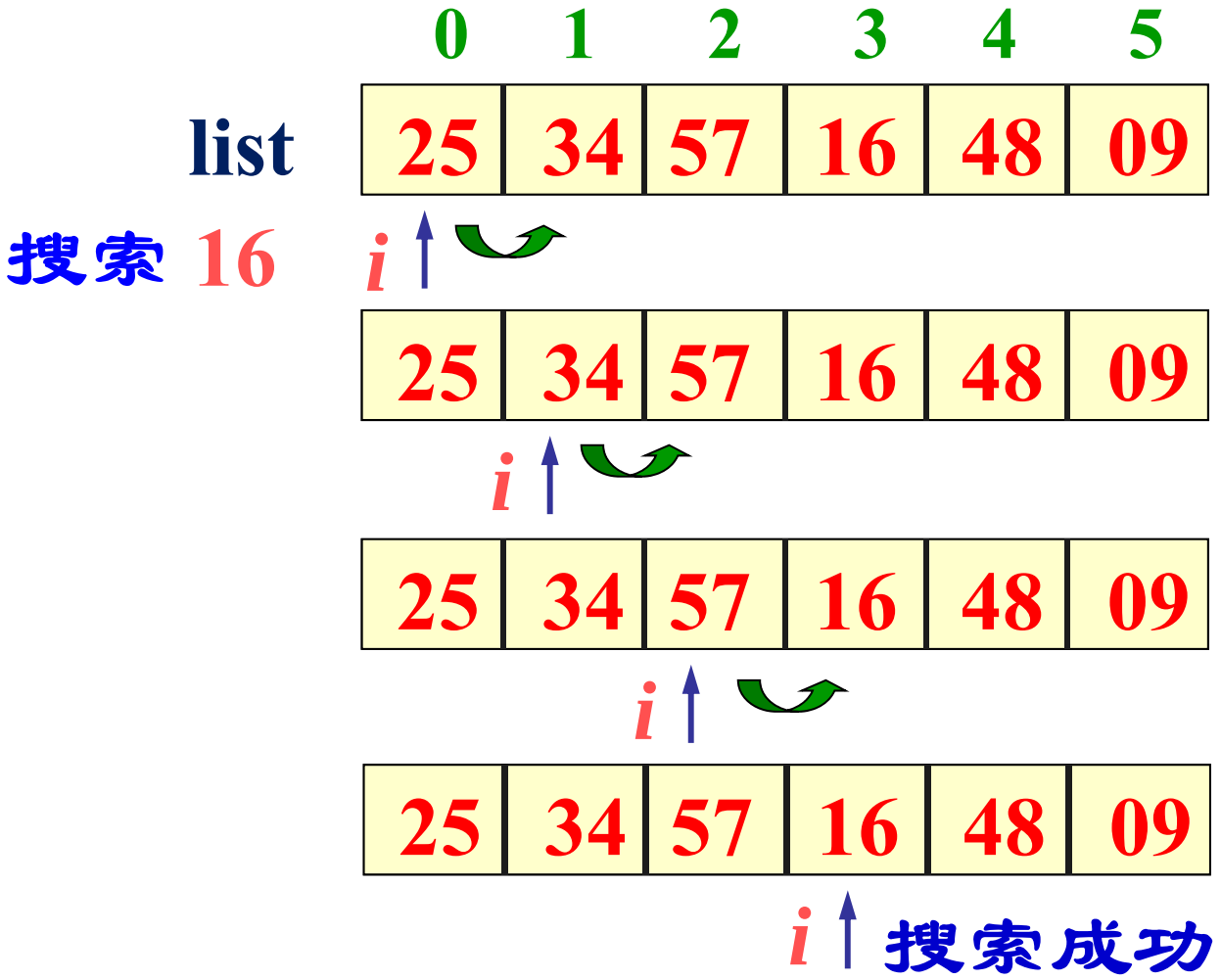
```
    private:
        Type *list;  //表的存放数组
        int MaxSize;  //表的最大可容纳项数
        int *p_n;  //当前已存表项的最后位置
};
```

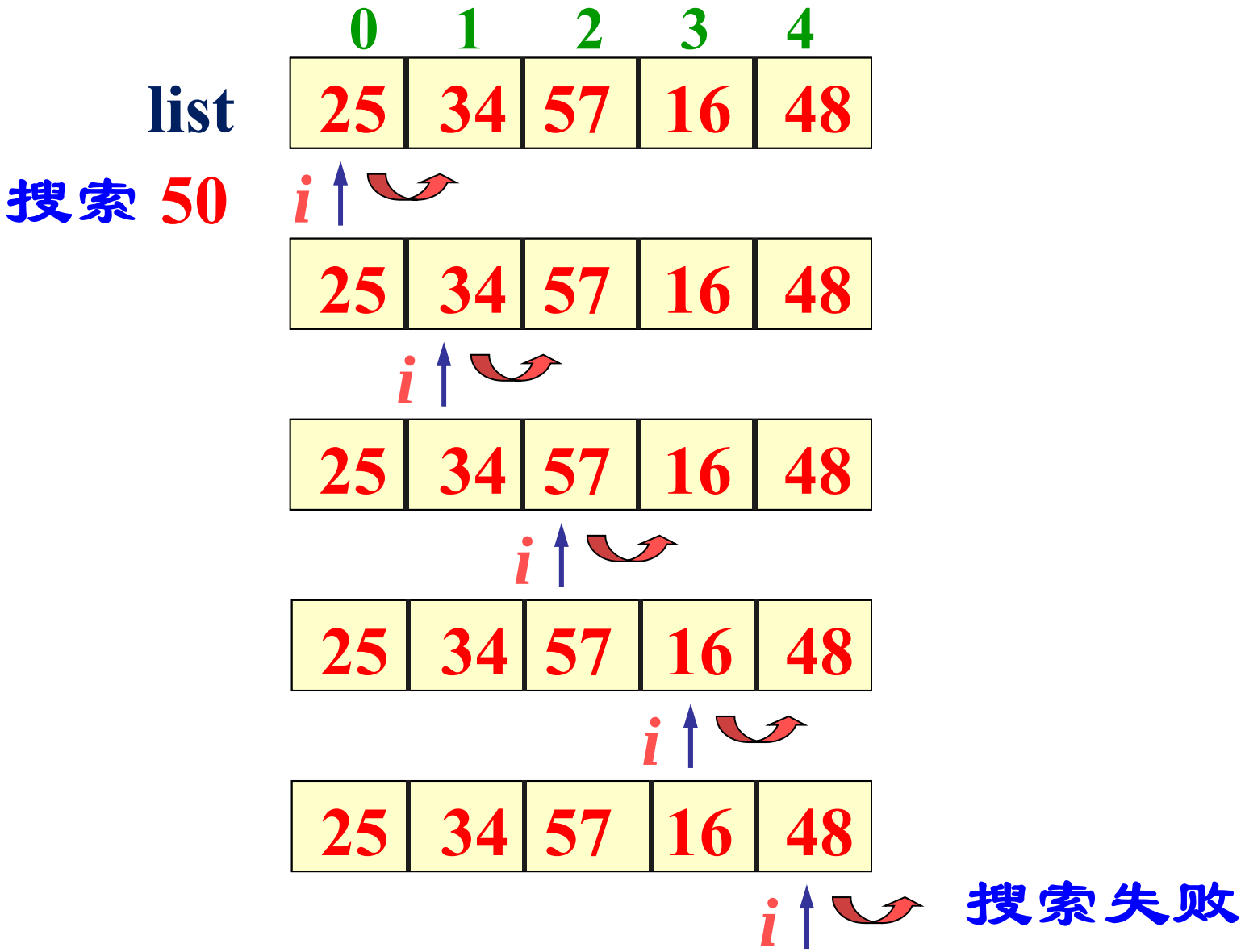
```
template <class Type>
SeqList <Type> :: SeqList(int sz) { //构造函数
    if (sz>0) {
        maxSize=sz;  last=-1;
        list=new Type [maxSize];
        if (list==NULL) {
            cerr<<“存储分配错误！ ”<<endl;
            exit(1);
        }
    }
}
```

```
template <class Type>
SeqList <Type> :: SeqList(SeqList <Type> &L) {
    MaxSize=L.Size( );  last=L.Length( )-1;
    list=new Type [maxSize];
    if (list==NULL) {
        cerr<<“存储分配错误！ ”<<endl;
        exit(1);
    }
    for (int i=1; i<=last+1; i++)
        list[i-1]=L.GetData(i);
}
```

```
template <class Type>
SeqList <Type> :: ReSize(int newSize) {
    if (newSize<=0)
      { cerr<<"无效的数组大小"<<endl; return; }
    if (newSize!=maxSize) {
        Type *NewArray=new Type [newSize];
        if (NewArray==NULL)
          { cerr<<"存储分配错误"<<endl; exit(1); }
        int n=last+1;
        Type *srcptr=list;
        Type *destptr=NewArray;
        while (n--)  *destptr++=*srcptr++;
        delete [ ] list;
        list=newArray;  maxSize=newSize;
    }
}
```

顺序搜索图示


0 1 2 3 4 5
list
25 34 57 16 48 09
搜索 16
i
25 34 57 16 48 09
i
25 34 57 16 48 09
i
25 34 57 16 48 09
i 搜索成功


0 1 2 3 4
list
25 34 57 16 48
搜索 50
i
25 34 57 16 48
i
25 34 57 16 48
i
25 34 57 16 48
i
25 34 57 16 48
i
搜索失败

```
template <class Type>
int SeqList <Type> :: Search(Type &x) const {
//搜索函数：在表中顺序搜索与给定值x匹配的表项
//找到则函数返回该表项是第几个元素
//否则函数返回0，表示搜索失败
    for (int i=0; i<=last; i++)
        if (list[i]==x)  return i+1;
    return 0;
}
```

搜索成功的平均比较次数

$$ACN = \sum_{i=0}^{n-1} p_i \times c_i$$

若搜索概率相等，则

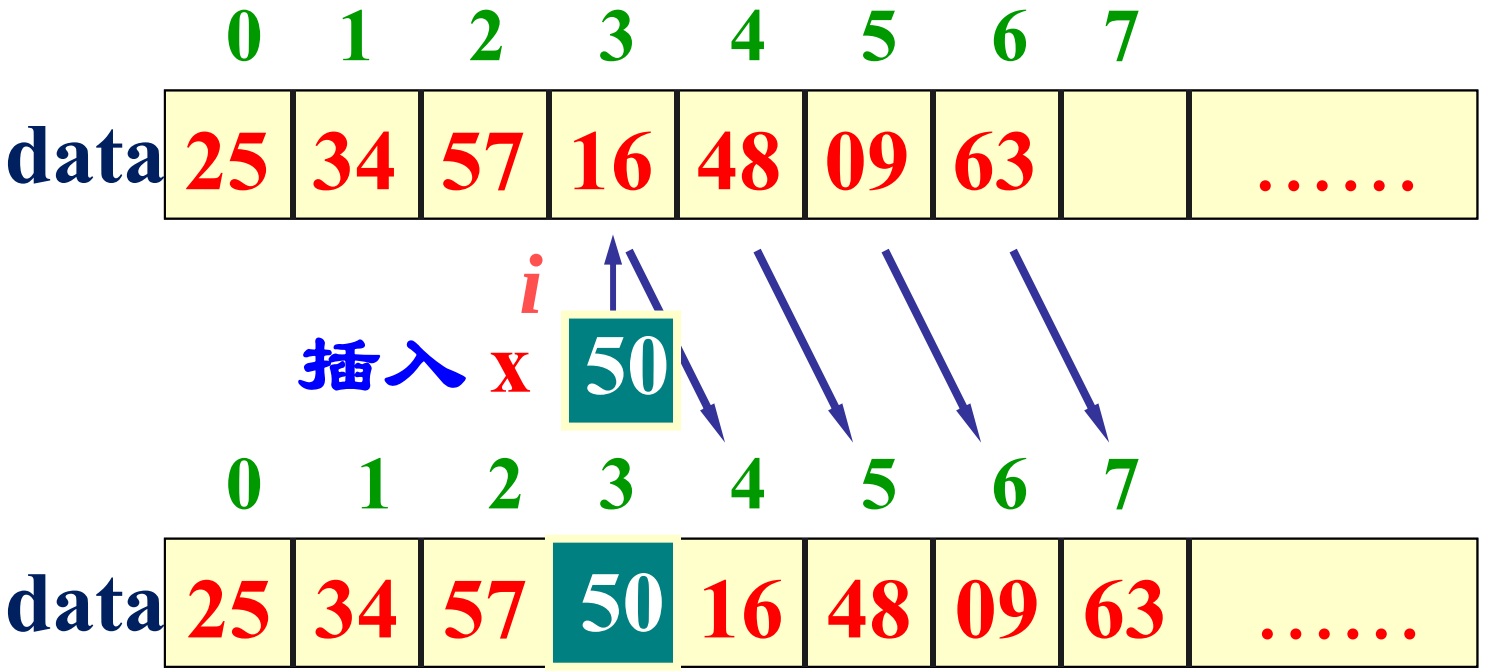
$$ACN = \frac{1}{n}\sum_{i=0}^{n-1}(i+1) = \frac{1}{n}(1+2+\cdots+n) =$$

$$= \frac{1}{n} * \frac{(1+n)*n}{2} = \frac{1+n}{2}$$

搜索不成功　数据比较n次。

```
template <class Type>
int SeqList <Type> :: Locate(int i) const {
//定位函数：函数返回第i个表项的位置
//否则函数返回0，表示定位失败
    if (i>=1&&i<=last+1)  return i;
    else return 0;
}
```

表项的插入



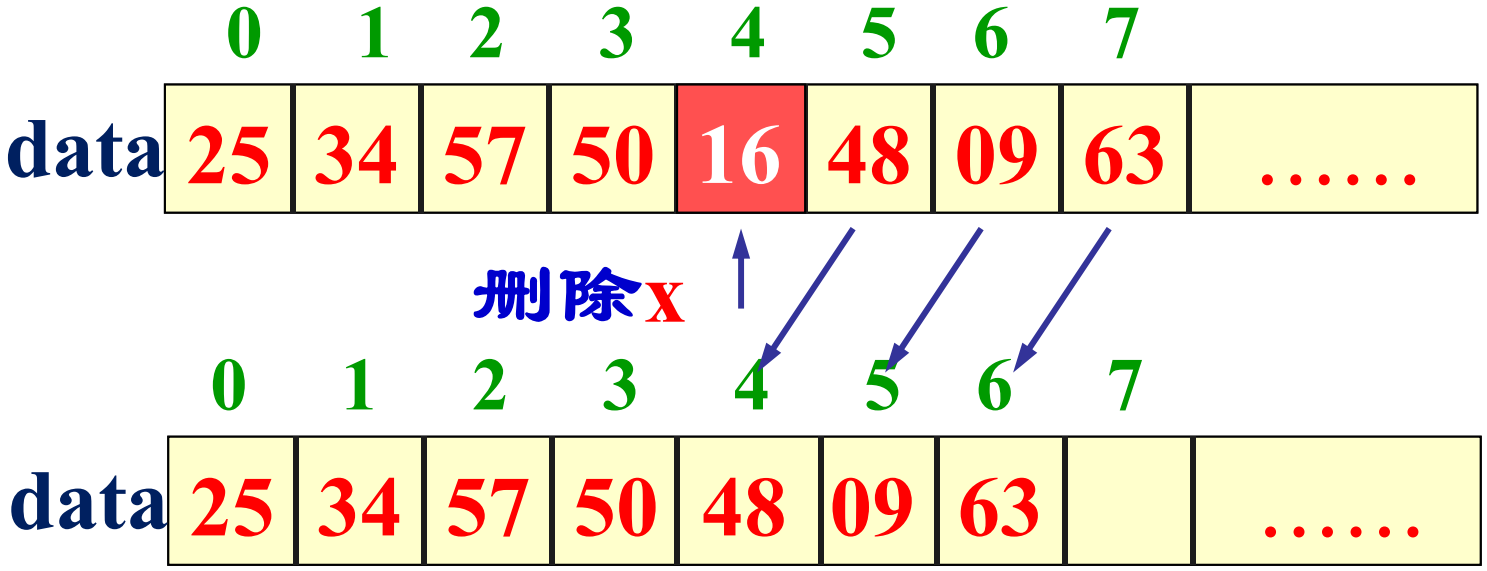
$$\text{AMN} = \frac{1}{n+1}\sum_{i=0}^{n}(n-i) = \frac{1}{n+1}(n+\cdots+1+0)$$

$$= \frac{1}{(n+1)}\frac{n(n+1)}{2} = \frac{n}{2}$$

程序2-6 顺序表的插入

```
template <class Type> int SeqList <Type> ::
        Insert(Type &x, int i)
{  //插入项x在顺序表中第i个位置处
   //函数返回插入是否成功的信息
      int j;
      if (i<0||i>*p_n)  return 1;
       //插入位置不合理，不能插入
      if (*p_n==MaxSize)  return 2;
       //无空闲可用存储单元，不能插入
      for (j=*p_n; j>i; j--)
           list[j]=list[j-1];  //依次后移
      list[i]=x;  //插入
      (*p_n)++;  //顺序表长度增1
      return 0;
}
```

表项的删除



$$\mathrm{AMN} = \frac{1}{n}\sum_{i=0}^{n-1}(n-i-1) = \frac{1}{n}\frac{(n-1)n}{2} = \frac{n-1}{2}$$

程序2-7 顺序表的删除

```
template <class Type> int SeqList <Type> ::
Remove(int i)
{ //删除第i个位置处的表项
      int j;
      if (i<0||i>=*p_n)  return 1;
       //删除位置不合理，不能删除
      for (j=i+1; j<*p_n; j++)
            list[j-1]=list[j];  //依次前移
      (*p_n)--;  //表长度减1
      return 0;
}
```

随堂练习

例1：用线性表的顺序存储结构来描述一个城市的设计和规划是否合适？为什么？

例2：在包含n个元素的顺序表中删除一个元素，需要平均移动_____个元素，其中具体移动的元素个数与_____有关。

例3：线性表的顺序存储结构具有三个弱点：其一，在作插入或删除操作时，需移动大量元素；其二，由于难以估计，必须预先分配较大的空间，往往使存储空间不能得到充分利用；其三，表的容量难以扩充。线性表的链式存储结构是否一定都能够克服上述三个弱点。试讨论之。

例1：用线性表的顺序存储结构来描述一个城市的设计和规划是否合适？为什么？

不合适。因为一个城市的设计和规划涉及非常多的项目，比较复杂，需要经常改动、扩充和删除各种信息，这样才适应不断发展的需要，所以顺序表不能很好地适应其需要。

例2：在包含n个元素的顺序表中删除一个元素，需要平均移动(n-1)/2 个元素，其中具体移动的元素个数与具体删除位置有关。

例3：线性表的顺序存储结构具有三个弱点：其一，在作插入或删除操作时，需移动大量元素；其二，由于难以估计，必须预先分配较大的空间，往往使存储空间不能得到充分利用；其三，表的容量难以扩充。线性表的链式存储结构是否一定都能够克服上述三个弱点。试讨论之。

不一定。由于链式存储需要额外的空间来存储指针，所以要比顺序存储多占用空间。在空间允许的情况下，链式存储结构可以克服顺序存储结构的弱点，但空间不允许时，链式存储结构会出现新问题。

多项式 (Polynomial)

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

$$= \sum_{i=0}^{n} a_i x^i$$

- n阶多项式$P_n(x)$有$n+1$项。
  - 系数$a_0, a_1, a_2, \ldots, a_n$。
  - 指数$0, 1, 2, \ldots, n$，按升幂排列。

多项式(Polynomial)的抽象数据类型

```
class Polynomial {
  public:
    Polynomial( );  //构造函数
    int operator ! ( );  //判是否零多项式
    float Coef(int e);
    int LeadExp( );  //返回最大指数
    Polynomial Add(Polynomial poly);
    Polynomial Mult(Polynomial poly);
    float Eval(float x);  //求值
};
```

多项式的存储表示

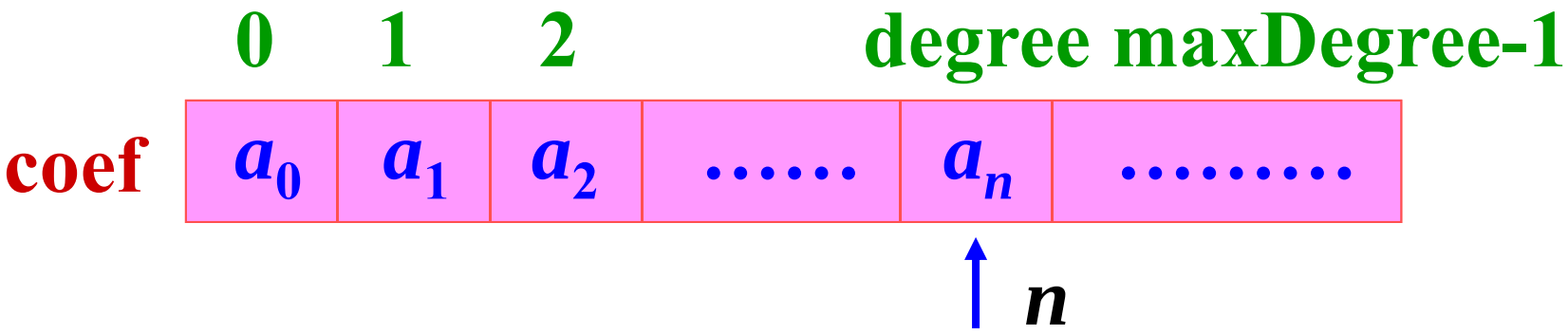
第一种：（静态数组表示）

```
private:
    int degree;
    float coef[maxDegree+1];
```

$P_n(x)$可以表示为：

pl.degree=n

pl.coef[i]=a_i, $0 \le i \le n$



第二种：**private:**

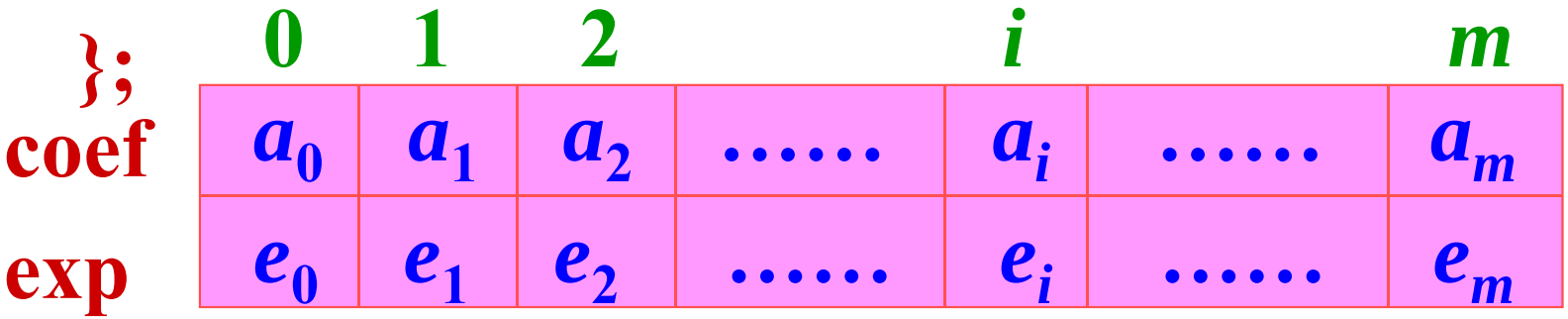
（动态数组表示）

```
    int degree;
    float *coef;
Polynomial :: Polynomial(int sz) {
    degree=sz;
    coef=new float [degree+1];
};
```

以上两种存储表示适用于指数连续排列的多项式。但对于指数不全的多项式如 $P_{101}(x)=3+5x^{50}-14x^{101}$, 不经济。

第三种：

```
class Polynomial;

class Term {  //多项式的项定义
   friend Polynomial;
   private:
      float coef;  //系数
      int exp;  //指数
};
```



程序2-8 多项式的类定义

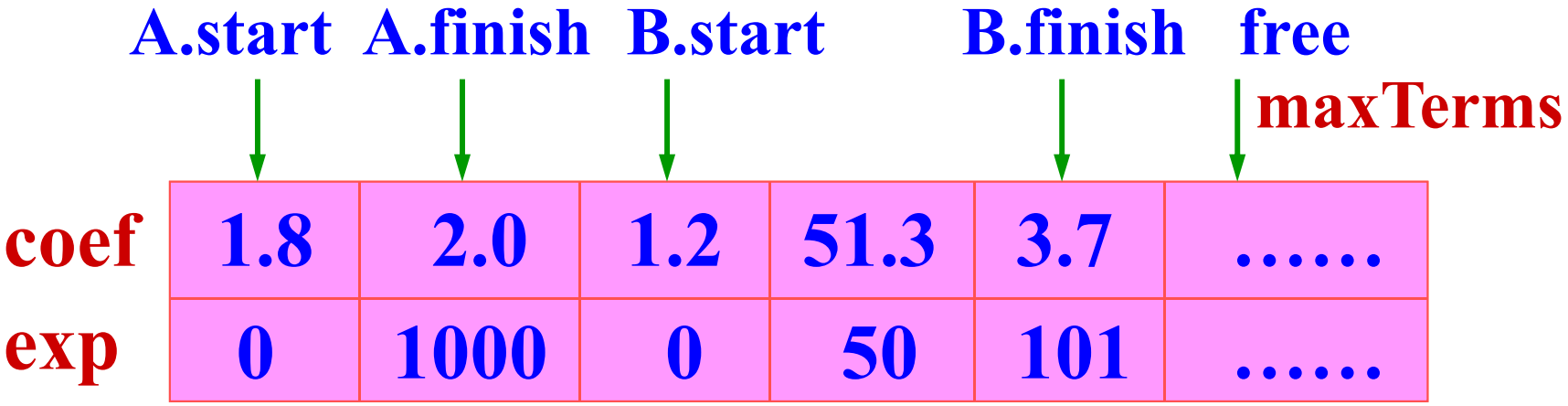
```
class Polynomial; //多项式类的前向引用声明
class term { //多项式中项的类定义
   friend Polynomial; //定义多项式类为term类的友元类
   private:
      float coef; //系数
      int exp; //指数
};
class Polynomial { //多项式类定义
   public:
      Polynomial( ); //无形参构造函数，返回多项式A(x)=0
      Polynomial(int size); //带形参构造函数
      int operator ! ( ); //若*this是零多项式则返回1，否则返回0
      int Poly_Append(int c, int e);
      //插入系数为c、指数为e的项至指针free所指示位置
```

多项式的类定义

```
    Polynomial Poly_Add(Polynomial poly);
    //返回多项式*this与多项式poly的和
    Polynomial Poly_Subtract(Polynomial poly);
    //返回多项式*this与多项式poly的差
    Polynomial Poly_Mult(Polynomial poly);
    //返回多项式*this与多项式poly的乘积
    Polynomial Poly_Division(Polynomial poly);
    //返回多项式*this与多项式poly的商
    ……
  private:
    static Term *Poly_Array;  //存放多项式的数组
    //要求在类外定义
    //Polynomial :: Poly_Array= new Term [MAXN+1];
    static int free;  //多项式有效项数的下一位置
    int start, finish;
};
```

$A(x)=2.0x^{1000}+1.8$

$B(x)=1.2+51.3x^{50}+3.7x^{101}$



两个多项式存放在termArray中。

两个多项式的相加

- 结果多项式另存。
- 扫描两个相加多项式，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。
- 若有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。

程序2-9 多项式的项加

```
int Polynomial :: Poly_Append(int c, int e);
{  //增加一项在起始空闲存储单元处
      if (free>MAXN)
         return 1;
         //无可用空闲存储单元，不能插入
      poly[free].coef=c;
      poly[free++].exp=e;
      return 0;
}
```

程序2-9 多项式的项加

```
Polynomial Polynomial :: Poly_Add(Polynomial B)
{ //返回两个多项式A(x)（在*this中）与B(x)的和
    Poynomial C;
    int ah=start, bh=B.start;
    //ah与bh分别为两个多项式的检测指针
    C.start=free;
    float c;  //c为结果多项式的存放指针
    while (ah<=finish && bh<B.finish)
     switch (Poly_Array[ah].exp, Poly_Array[bh].exp)
     { //比较对应项指数
        case '=':  c=Poly_Array[ah].coef+Poly_Array[bh].coef;
                   //相等，系数相加
                   if (c)  Poly_Append(c, Poly_Array[ah].exp);
                   //和非零，建立新项
                   ah++;  bh++;
                   break;
```

程序2- 9 多项式的项加

```
        case '>':  Poly_Append(Poly_Array[bh].coef ,
                                    Poly_Array[bh].exp);
                   //指数不等，建立新项
                   bh++;
                   break;
         case '<':  Poly_Append(Poly_Array[ah].coef ,
                                    Poly_Array[ah].exp);
                   //指数不等，建立新项
                   ah++;
      }
    for ( ; ah<=finish; ah++)
    //将A(x)中剩余项加入结果多项式中
          Poly_Append(Poly_Array[ah].coef , Poly_Array[ah].exp);
    for ( ; bh<=B.finish; ah++)
    //将B(x)中剩余的项加入结果多项式中
          Poly_Append(Poly_Array[bh].coef , Poly_Array[bh].exp);
    C.finish=free-1;
    return C;
}
```

- 执行函数Poly_Append的时间是O(1)。
- 在函数Poly_Add中，执行三个循环之外的几个语句的时间也是O(1)。所以，执行时间主要花费在三个循环上。
- 设A(x)和B(x)所含非零项的个数分别为*m*和*n*，对于第一个循环，当A(x)和B(x)的各项的次数都不相同时，执行循环的次数最多，共执行(*m*+*n*)次，故其执行时间最多为O(*m*+*n*)；第二和第三个循环最多分别执行*m*次和*n*次，故其执行时间最多分别为O(*m*)和O(*n*)。
- 执行函数Poly_Add的时间为O(1)+O(*m*+*n*)+O(*m*)+O(*n*)=O(*m*+*n*)。

2.4 线性表的链式表示——链表

为什么学习链表来表示线性表？

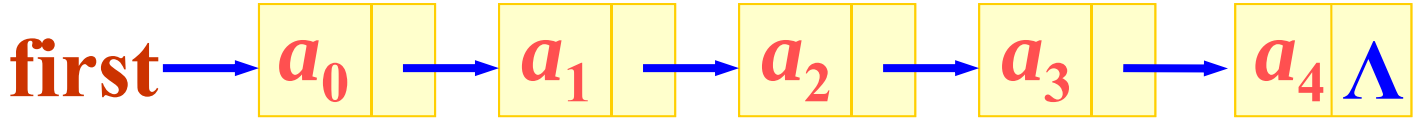
- 用数组或顺序映射表示线性表，相邻元素存储地址的间距是常数。因此，可用**O(1)**的时间访问任何元素。但是，删除或往其中插入元素将导致表中其它元素的移动，代价太高**O(*n*)**。
- 链表表示可以有效地解决上述数据移动问题。在链表表示中，表的数据元素可存放在存储器中的任何位置，每一个元素都与其下一个元素的地址（指针）相关联。

- 利用数组或顺序方式来组织数据结构
  - 优点
    - 存储利用率高，存取速度快。
  - 缺点
    - 数组元素个数不能自由扩充（除动态数组外）
    - 效率很低
    - 造成空间很大浪费
- ***链表(Linked List)***
  - 适用于插入删除频繁、存储空间需求不定的情形。
  - 不但可表示线性聚集，还可表示非线性聚集。

单链表 (Singly Linked List)

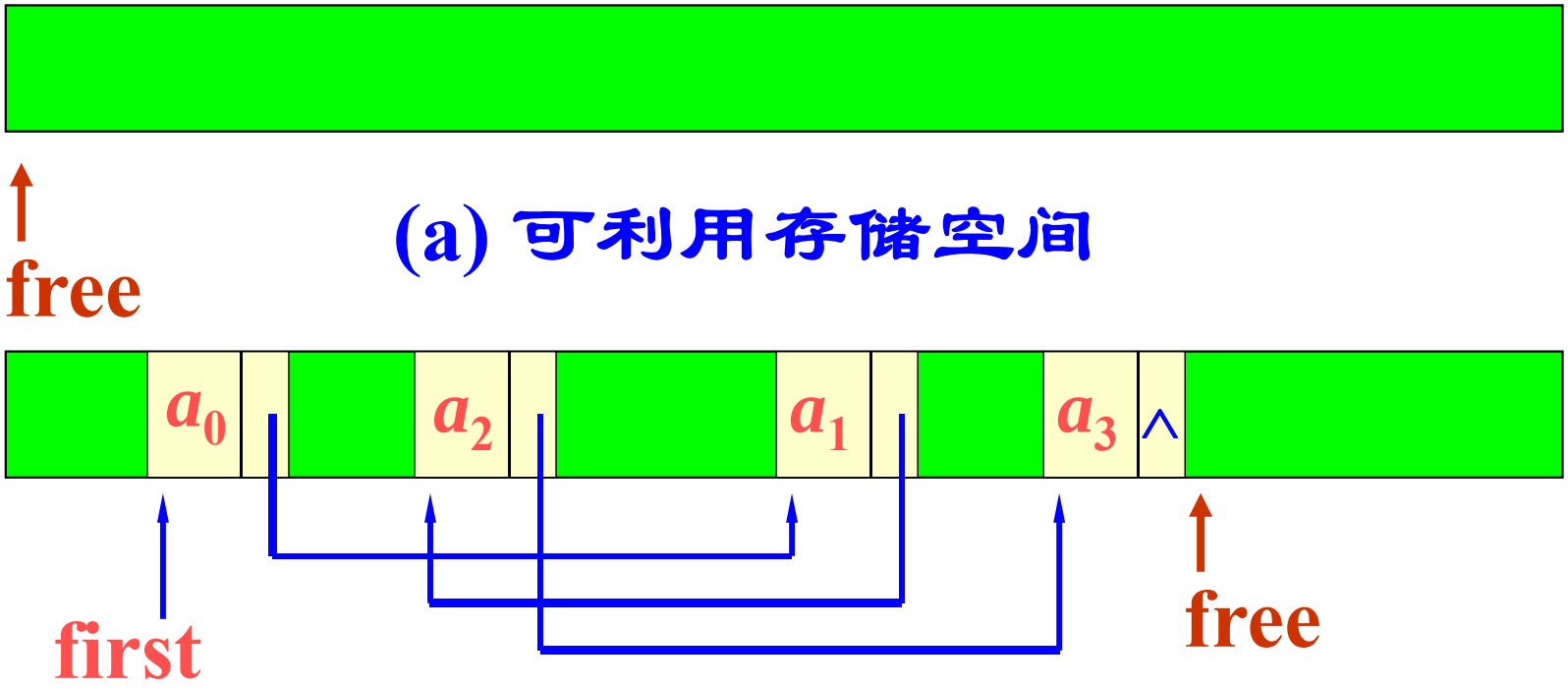
- 特点
  - 每个元素（表项）由结点(*Node*)构成。

  - 线性结构



  - 结点之间可以连续，可以不连续存储;
  - 结点的逻辑顺序与物理顺序可以不一致;
  - 表可扩充。

单链表的存储映像



单链表类定义

- 多个类表达一个概念（单链表）
  - 链表结点(LinkNode)类
  - 链表(List)类
- 定义方式
  - 复合方式
  - 嵌套方式
  - 继承方式
  - 结构方式

```
class List;  //链表类定义（复合方式）
class LinkNode {  //链表结点类
 friend class List;  //链表类为其友元类
 private:
    int data;  //结点数据，整型
    LinkNode *link;  //结点指针
};
class List {  //链表类
 private:
    LinkNode *first, *current;  //表头指针
 public:
    ..... //链表操作
};
```

```
class List {  //链表类定义（嵌套方式）
 private:
    class LinkNode {  //嵌套链表结点类
     public:
        int data;
        LinkNode *link;
    };
    LinkNode *first, *current;  //表头指针
 public:
    ……  //链表操作
};
```

```
class LinkNode {  //链表结点类
   protected:
        int data;
        LinkNode *link;
};
class List : public LinkNode
{  //链表类定义（继承方式）
    //链表类，继承链表结点类的数据和操作
   private:
        LinkNode *first, *current;  //表头指针
  public:
        ..... //链表操作
};
```

```
struct LinkNode {  //用结构定义链表结点类
    int data;
    LinkNode *link;
};
class List
{
    private:
        LinkNode *first, *current;  //表头指针
    public:
        ..... //链表操作
};
```

- 在复合方式中，链表结点类中声明链表类是它的友元类，这样可以“奉献”它的私有成员给链表类，这种方式灵活。
- 在嵌套方式中，链表结点类是链表类的私有成员，这样限制了链表结点类的应用范围。
- 在继承方式中，链表类声明为链表结点类的派生类，这在实现上是可行的，但在逻辑上是有问题的。如
  - ✓ 三角形 **is** 多边形（继承关系）
  - × 链表 **is** 链表结点（显然概念有误）

程序2-10 单链表的复合类定义

```
template <class Type> class List;  //List类的前向引用声明
template <class Type> class ListNode {  //链表结点类定义
    friend class List <Type>;  //链表类作为友元类定义
    public:
        ListNode( ) { link=NULL; }  //不带形参的构造函数
        ListNode(const Type &item);  //带形参的构造函数
        ListNode <Type> *NextNode( ) { return link; }
        //给出当前结点的后继结点地址
        void InsertAfter(ListNode <Type> *p);  //当前结点插入
        ListNode <Type> * GetNode(const Type &item,
                                  ListNode <Type> *next);
        //建立一个新结点
        ListNode <Type> * RemoveAfter( );
        //删除当前结点下一结点
    private:
            Type data;  //数据域
            ListNode <Type> *link;  //后继指针域
};
```

程序2-10 单链表的复合类定义

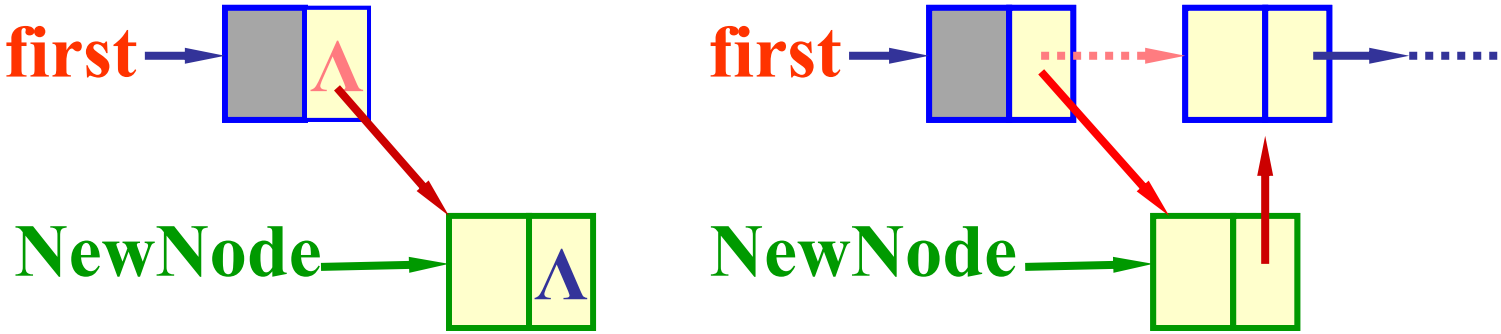
```
template <class Type> class List {  //单链表类定义
    public:
        List(const Type &value)
                        { tail=head=new ListNode <Type> (value); }
        //构造函数
        ~List( );  //析构函数
        ListNode <Type> * Create_List(int n);  //建立一个单链表
        void MakeEmpty( );  //将链表置为空表
        int Length( ) const;  //计算链表的长度
        ListNode <Type> * Find(Type value);
        //在链表中搜索含数据value的结点
        int Insert(Type value, int i);
        //将新元素插在链表中第i个位置
        Type * Delete(int i);  //将链表中的第i个元素删去
        Type * Get(int i);  //取出链表中第i个元素
     private:
        ListNode <Type> *head, *tail;
        //链表的表头指针、表尾指针
};
```

程序2-11 单链表的建立

```
template <class Type> ListNode <Type> * Create_List(int n)
{
        ListNode *p, *q;
        if (n==0)
                return NULL;
        head=new ListNode <Type> (NULL);
        p=head;
        for (i=1; i<n; i++)
        {
                cin>>p->data;
                q=new ListNode <Type> (NULL);
                p->link=q;
                p=q;
        }
        cin>>p->data;
        p->link=NULL;
        return head;
}
```

前插法建立单链表

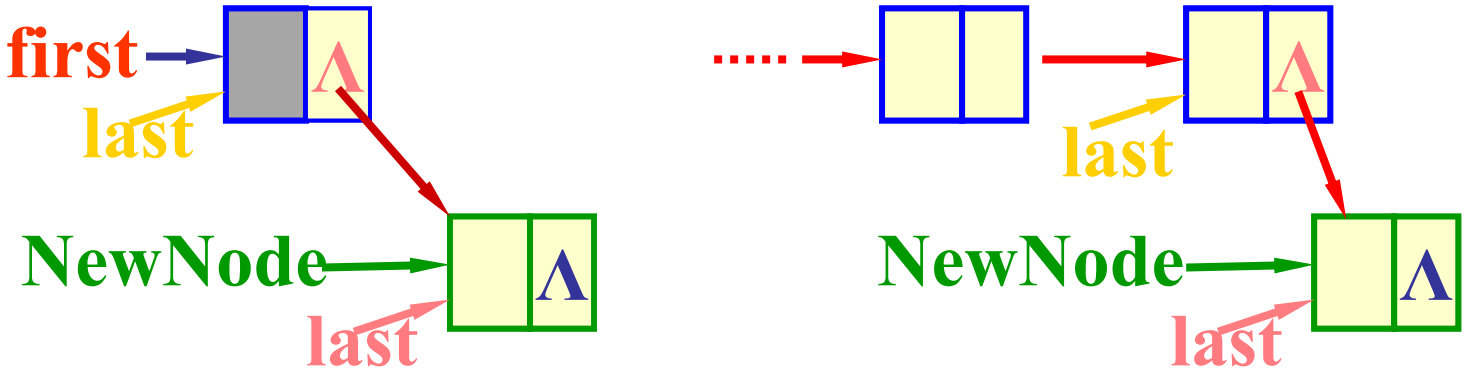
- 从一个空表开始，重复读入数据：
  - 生成新结点；
  - 将读入数据存放到新结点的数据域中；
  - 将该新结点插入到链表的前端。
- 直到读入结束符为止。



```
template <class Type>
void List <Type> :: CreateListF(Type endTag)
{
    Type val;  LinkNode <Type> *NewNode;
    first=new LinkNode <Type>;  //表头结点
    cin>>val;
    while (val!=endTag) {
        NewNode=new LinkNode <Type> (val);
        NewNode->link=first->link;
        first->link=NewNode;  //插入到表前端
        cin>>val;
    }
    current=first;
}
```

后插法建立单链表

- 每次将新结点插到链表的表尾；
- 设置一个尾指针**last**，总是指向表中最后一个结点，新结点插在它的后面；
- 尾指针**last**初始时置为指向表头结点地址。



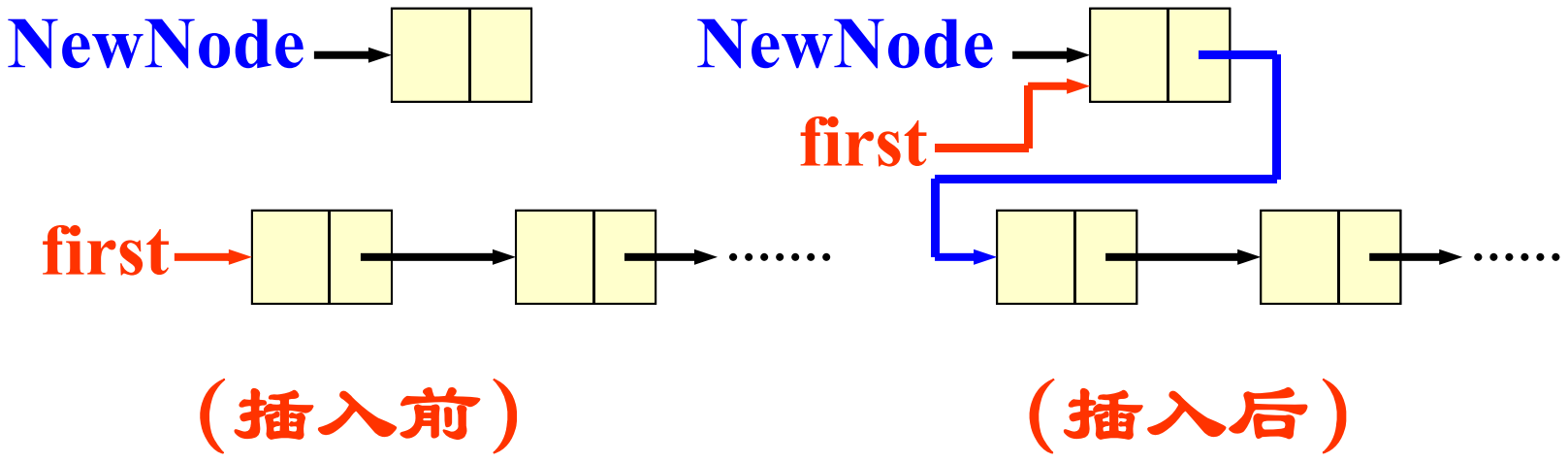
```
template <class Type>
void List <Type> :: CreateListR(Type endTag) {
    Type val;  LinkNode <Type> *NewNode, *last;
    first=new LinkNode <Type>;  //表头结点
    cin>>val;  current=last=first;
    while (val!=endTag) {  //last指向表尾
        NewNode=new LinkNode <Type> (val);
        last->link=NewNode;  last=NewNode;
        cin>>val;  //插入到表末端
    }
    last->link=NULL;  //表收尾
}
```

单链表中的插入与删除

- 插入
  - 第一种情况：在链表最前端插入

NewNode->link=first;

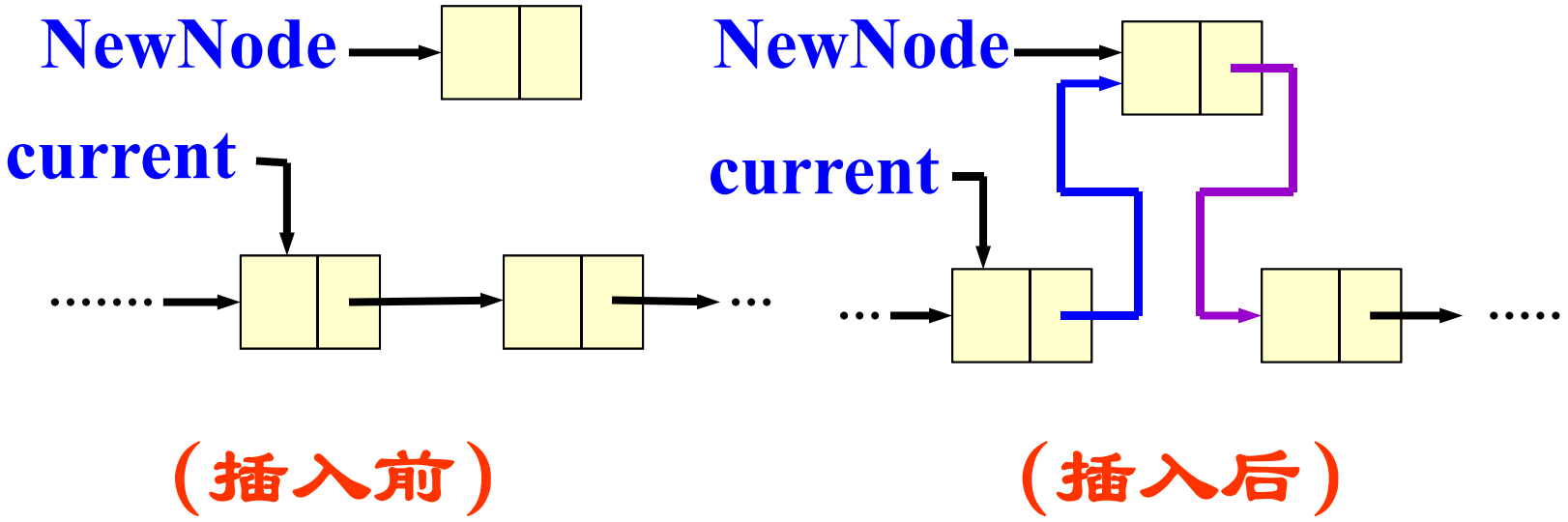
first=NewNode;



- 第二种情况：在链表中间插入

NewNode->link=current->link;

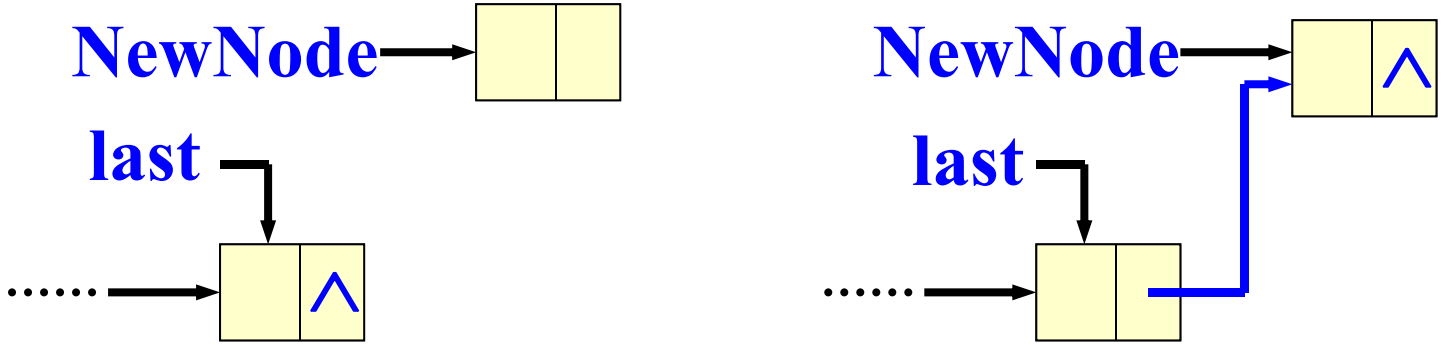
current->link=NewNode;



- 第三种情况：在链表末尾插入

NewNode->link=last->link;

last->link=NewNode;



（插入前）　　（插入后）

last (≡current)

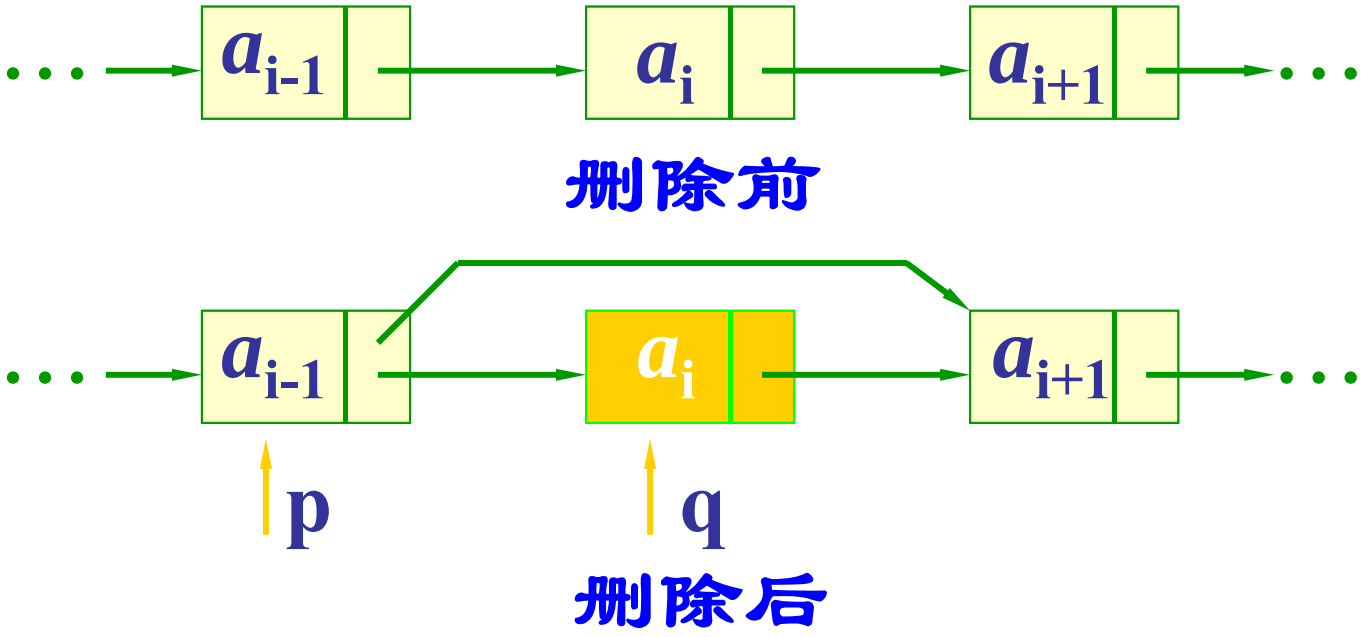
程序2-12 单链表的插入

```
template <class Type> int List <Type> :: Insert(Type x, int i)
{  //将新元素x插入到第i结点之前
   //其中i是结点编号，从0开始计数
      ListNode *p, *q;
      int j=0;
      p=head;
      q=new ListNode <Type> (x, NULL);
      //建立数据域为x的新结点，由q指示
      while (p!=NULL && j<i-1)
      {  //循链找第i-1个结点
           p=p->link;
           j++;
       }
       if (p==NULL && head!=NULL)
       {  //非空表且链短，找不到第i-1个结点
           cout<<"Invalid position for Insertion!\n";
           return 0;  //插入失败，返回0
       }
```

程序2-12 单链表的插入

```
    if (head==NULL||i==0)
    { //将新结点插入至空表或非空表第一个结点之前
      q->link=head; //原表首结点成为新结点的后继结点
      if (head==NULL)
          tail=q;
          //在空表中插入新结点时，表尾指针指向该结点
          head=q; //新结点q成为表首结点 }
      else { //将新结点插入至链表中间或者表尾
          q->link=p->link;
          if (p->link==NULL)
              tail=q; //如果插入新结点至表尾
                      //则修改表尾指针指向新结点
          p->link=q;
      }
      return 1; //插入成功，返回1
}
```

- 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

程序2-13 单链表的删除

```
template <class Type> int List <Type> :: Delete(int i)
{ //删除链表中的第i个结点并将其返回
  //其中i是结点编号，从0开始计数
      ListNode *p, *q;
      int j=0;
      p=head;
      while (p!=NULL && j<i-1)
      { //循链找第i-1个结点
          p=p->link;
          j++;
       }
       if (p==NULL && head!==NULL)
       { //非空表且链短，找不到第i-1个结点
           cout<<“Invalid position for Deletion!\n”;
           return 0; //删除失败，返回0
       }
```

程序2-13 单链表的删除

```
    if (i==0)
    { //删除表首结点
        q=head;
        p=head=head->link;
        //重置表首结点为被删结点的后继结点 }
    else { //删除表中间或者表尾结点
        q=p->link;
        p->link=q->link;
    }
    if (q==tail)
        tail=p; //如果删除表尾结点，修改表尾指针
    delete q;
    return 1; //删除成功，返回1
}
```

随堂练习

例1：(1) 对一个线性表分别进行遍历和逆置运算，求解其最好的渐进时间复杂度表示。

(2) 求解求顺序表和单链表长度的渐进时间复杂度表示。

(3) 若给定n个元素的向量，则求解建立一个有序单链表的渐进时间复杂度表示。

例2：设有头结点的单链表L，编程对表中任一值只保留一个结点，删除其余值相同的结点。

例3：有一带头结点的单链表，编程将链表颠倒过来，要求不用另外的数组或结点完成。

例1：(1) 对一个线性表分别进行遍历和逆置运算，求解其最好的渐进时间复杂度表示。

遍历时间的渐进时间复杂度表示为O(n)；进行逆置运算时，顺序表的时间要少于单链表时间，其量级也为O(n)。

(2) 求解求顺序表和单链表长度的渐进时间复杂度表示。

顺序表的长度可以直接从最后一个元素的位置获得，而单链表的长度则必须遍历整个表后获得，即O(1)与O(n)。

(3) 若给定n个元素的向量，则求解建立一个有序单链表的渐进时间复杂度表示。

单纯建立单链表的渐进时间复杂度为O(n)，而建立一个有序单链表还涉及查找插入到其正确位置，故渐进时间复杂度表示为$O(n^2)$。

例 2：设有头结点的单链表 L，编程对表中任一值只保留一个结点，删除其余值相同的结点。

(1) 首先用指针 p 指向链表中第一个数据结点，然后用指针 t 搜索整个链表以寻找值相同的结点直到链尾；在搜索中，指针 s 指向 t 所指结点前驱结点，当 t->data=p->data 时则删除 t 所指结点，即 s->link=t->link。

(2) 修改指针 p，使 p 指向链表的下一个结点（即 p=p->link），然后重复(1)的操作直至 p=NULL。

```
DelElem(lklist  *L)
{
    Pointer *p, *t, *pre;
    p=L->link;
    t=p;
    while (p!=NULL)
    {
        pre=t;
        t=t->link;
        do
        {
            while ((t!=NULL) && (t->data!=p->data))
            {
                pre=t;
                t=t->link;
            }
            if (t!=NULL)
            {
                pre->link=t->link;
                free(t);
                t=pre->link;
            }
        }while (t!=NULL);
        p=p->link;
        t=p;
    }
}
```

例 3：有一带头结点的单链表，编程将链表颠倒过来，要求不用另外的数组或结点完成。

在遍历原单链表每个结点的同时摘下该结点插入到新链表的表头。这样，先遍历到的结点先插入，后遍历到的结点后插入；由于插入是在表头进行，所以先插入的结点成为表尾，后插入的结点成为表头；也即实现了链表的逆置。

```
void reverse (lklist *h)
{
    lklist *p, *q;
    p=h->link;      /*p 指向原链表的第一个数据结点*/
    h->link=NULL;     /*新链表初始为空*/
    while (p)
    {
        q=p;       /*q 指向将摘下来插入到新链表的结点*/
        p=p->link;     /*p 指向下一个待摘下来的原链表结点*/
        q->link=h->link;          /*将*q 插入到新链表表头*/
        h->link=q;
    }
}
```

- 单链表用于表示一个顺序表时，经常需要执行的操作包括：
  - 打印一个单链表中所有数据的值；
  - 当链表中所有的数据为整数、浮点数或双精度数时，计算所有数据的累加和、乘积，求它们最大值、最小值、中值、平均值、方差；
  - 检索链表中所有满足某种条件的数据，以及对它们进行排序等。
- 为实现上述操作，都需要前后搜索整个链表。
  - 如果将所有操作都加入到链表类中，会使类的成员函数集变得很大。
  - 对于将来出现的大量新的需求，必须不断地修改类的定义，不利于建立类库以及类的复用。

链表的游标类 (Iterator)

链表游标(Iterator)是一种用于遍历链表的全部元素的对象。从first出发，很容易遍历链表的全部元素，为什么还需要游标呢？

- 设链表L的元素都是整型数，考虑下列操作：
  - 打印L中的所有元素。
  - 计算L中所有元素的最大值、最小值和平均值。
  - 计算L中所有元素的和与积。
  - 求L中满足谓词P(x)的元素（例如，P(x)为：x是某个整数的平方）。
  - 求L中的元素x，使得对于某个函数f，f(x)达到最大值。

- 这启发我们将遍历(Traversal)移到List <Type>定义之外。

- 进一步观察，以上操作都不改动链表内容，但需要访问List <Type>和LinkNode <Type>的私有数据成员。为此，我们定义第三个类ListIterator <Type>。

- ListIterator <Type>处理遍历链表的细节并对外提供链表中元素的值。

- 游标类主要用于单链表的搜索。
- 游标类的定义原则
  - **Iterator**类是**List**类和**ListNode**类的友元类。
  - **Iterator**对象引用已有的**List**类对象。
  - **Iterator**类有一数据成员**current**，记录对单链表最近处理到哪一个结点。
  - **Iterator**类提供若干测试和搜索操作。

表示链表的三个类的模板定义

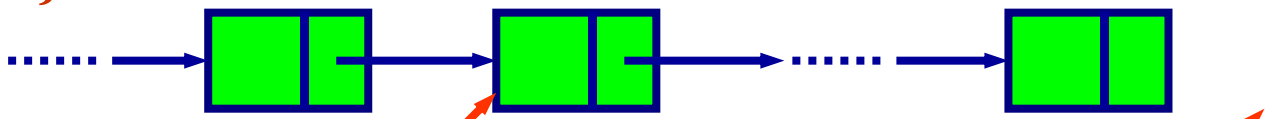
```
template <class Type> class List;
template <class Type> class ListIterator;
template <class Type> class LinkNode {  //表结点
    friend class List <Type>;
    friend class ListIterator <Type>;
    public:
        ……
    private:
        Type data;
        LinkNode <Type> *link;
};
```

```
template <class Type> class List {  //链表类
    public:
        ……
    private:
        LinkNode <Type> *first, *last;
};

template <class Type> class ListIterator {
    private:
        const List <Type> &list;  //引用已有链表
        LinkNode <Type> *current;  //当前结点指针
```

```
public:
    ListIterator(const List <Type> &l)
                    : list(l), current(l.first) { }
    //构造函数: 引用链表l, 表头为当前结点
    LinkNode <Type> * Firster( )
      { current=first; return first; }
    //当前指针置于表头, 返回表头结点地址
    bool NotNull( ); //检查当前指针空否
    bool NextNotNull( ); //检查链表中下一结点是否非空
    LinkNode <Type> *First( ); //返回第一个结点
    LinkNode <Type> *Next( );
    //返回链表当前结点的下一个结点的地址
};
```

链表的游标类成员函数的实现

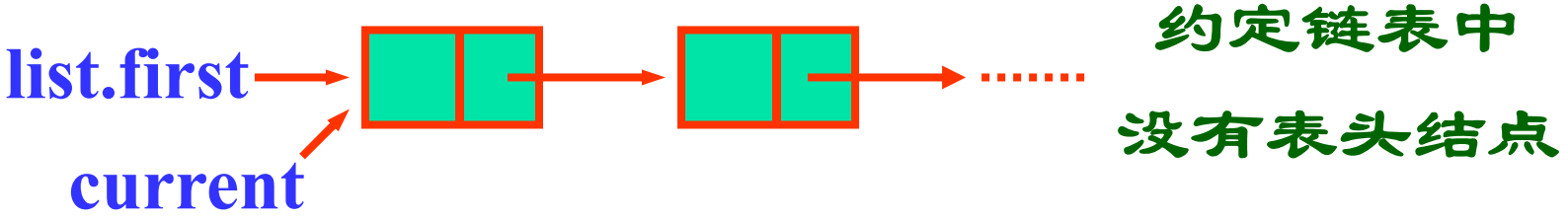
```
template <class Type>
bool ListIterator <Type> :: NotNull( ) {
//检查链表中当前元素是否非空
    if (current!=NULL) return true;
    else return false;
}
```



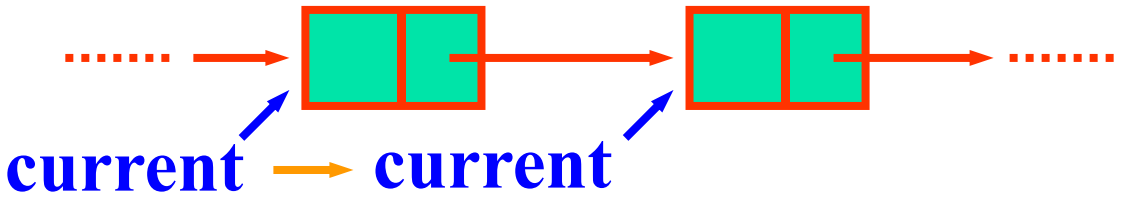
current　　current

情况 1　返回true　　情况 2　返回false

```
template <class Type>
LinkNode <Type> * ListIterator <Type> :: First( ) {
//返回链表中第一个结点的地址
    if (list.first!=NULL) {
        current=list.first;
        return &current->data;  }
    else { current=NULL;  return NULL; }
}
```



```
template <class Type>
LinkNode <Type> * ListIterator <Type> :: Next( ) {
//返回链表中当前结点的下一个结点的地址
    if (current!=NULL
          && current->link!=NULL ) {
       current=current->link;
       return &current->data;
    }
    else { current=NULL;  return NULL; }
}
```



利用游标类(Iterator)计算元素的和

```
int sum(const List <int> &L) {
    ListIterator <int> li(L);
    //定义游标对象， current指向li.first
    if (!li.NotNull( ))  return 0;
    //链表为空时返回
    int retval=*li.First( );  //第一个元素
    while (li.nextNotNull( ))  //链表未扫描完
        retval+=*li.Next( );  //累加
    return retval;
}
```

多项式的链表表示

- 在多项式的链表表示中每个结点三个数据成员：

Data≡Term

- 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。

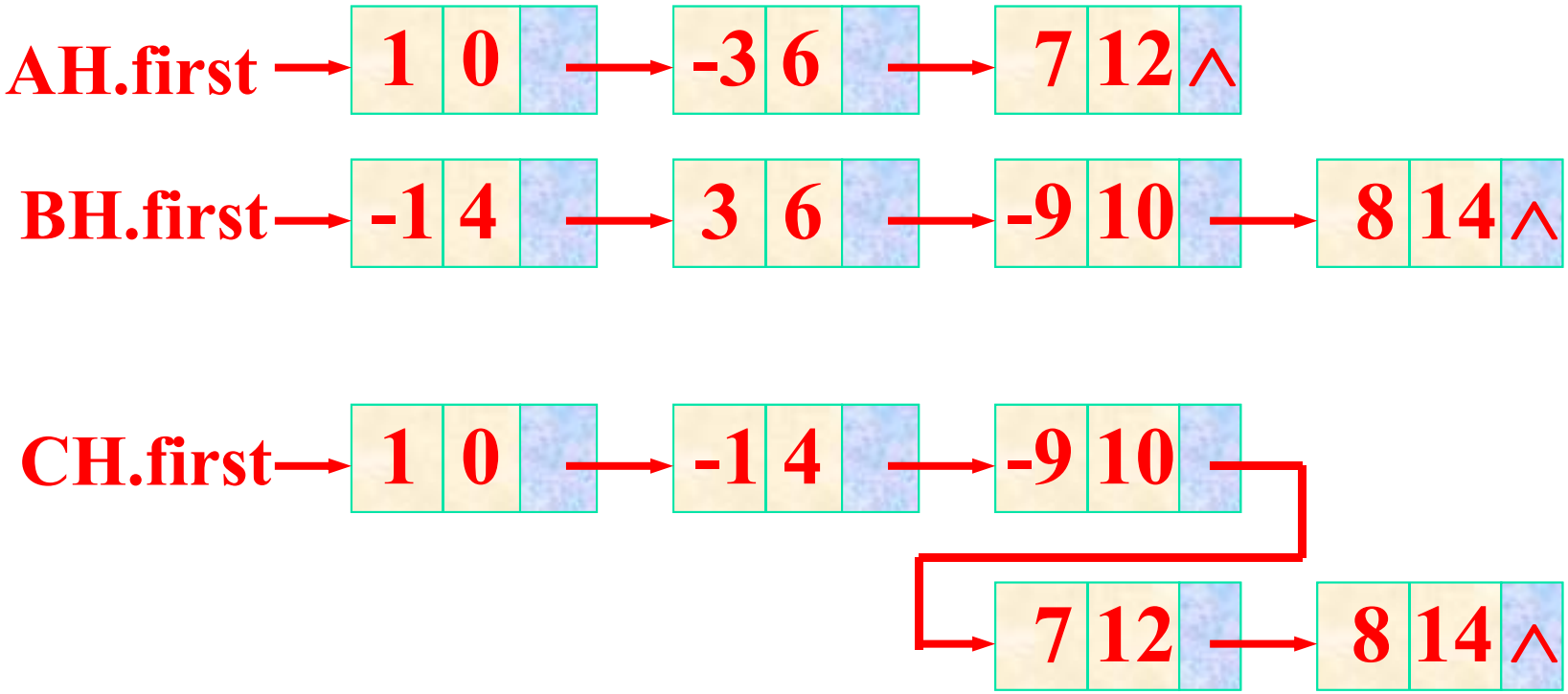
多项式(Polynomial)类的链表定义

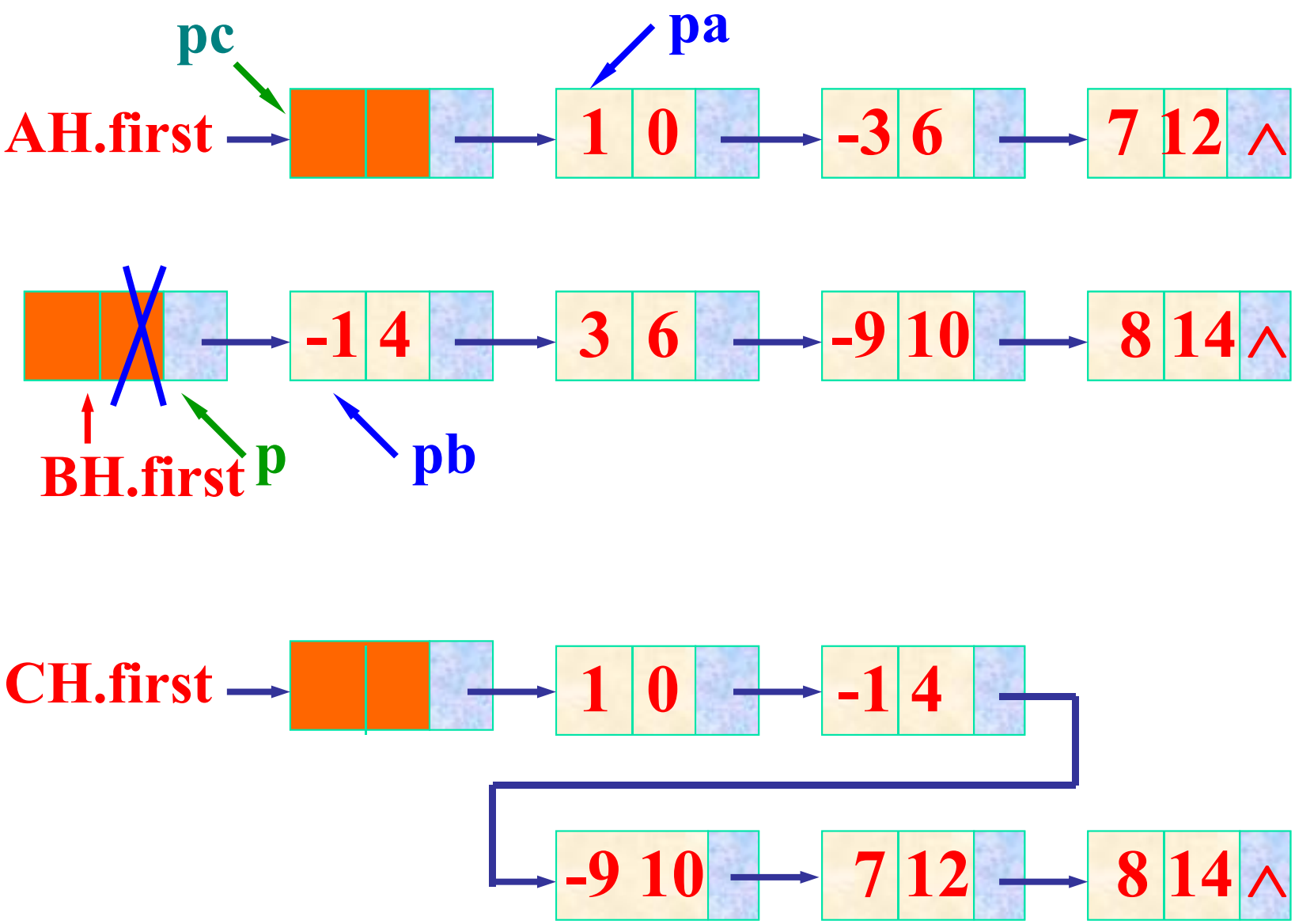
```
struct Term {  //多项式结点定义
    float coef;  //系数
    int exp;  //指数
    Term *link;
    Term(float c, int e) { coef=c;  exp=e; }
};
class Polynomial  {  //多项式类的定义
    List <Term> poly;
    friend Polynomial operator +
        (Polynomial &, Polynomial &);  //加法
};
```

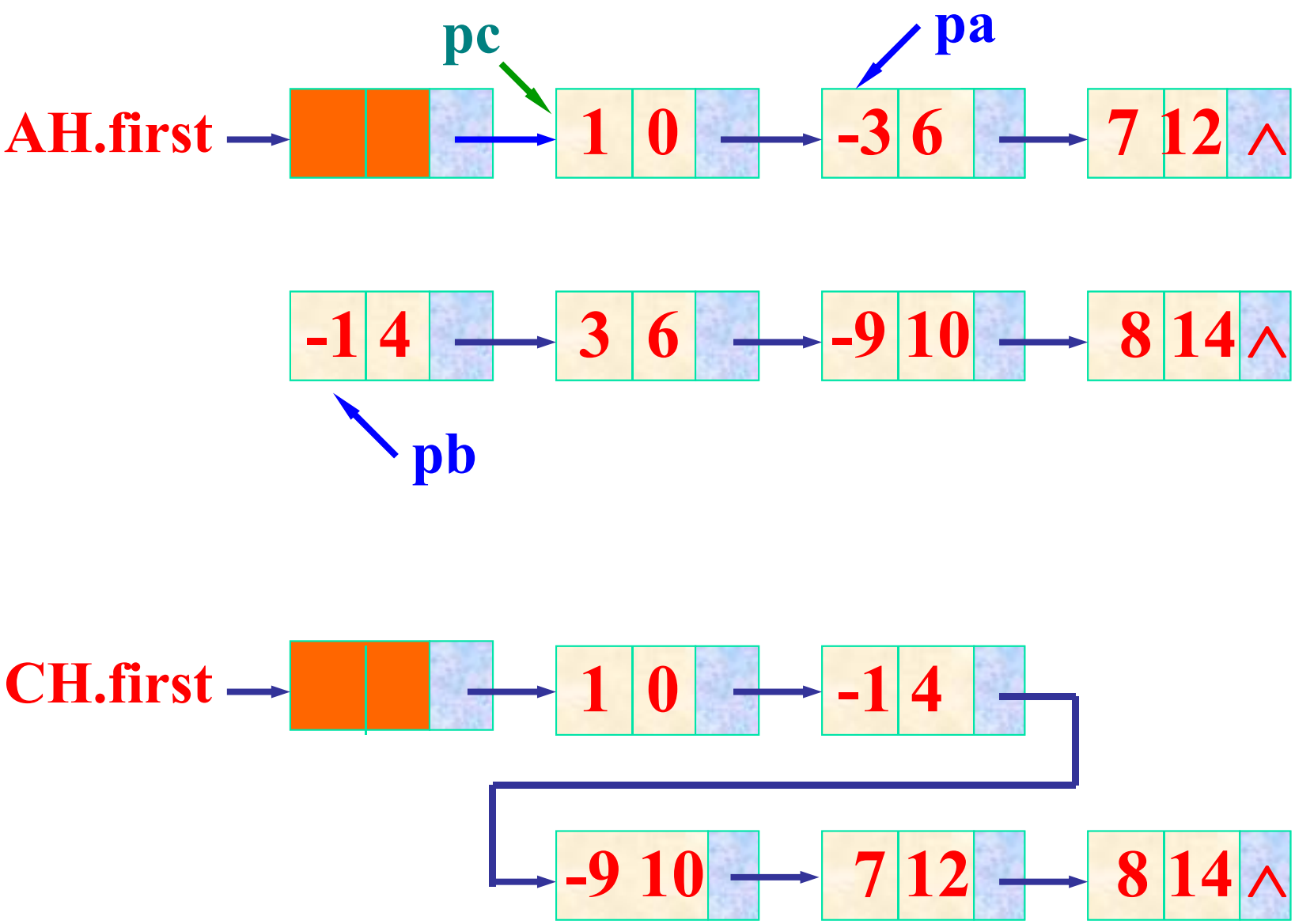
多项式链表的相加

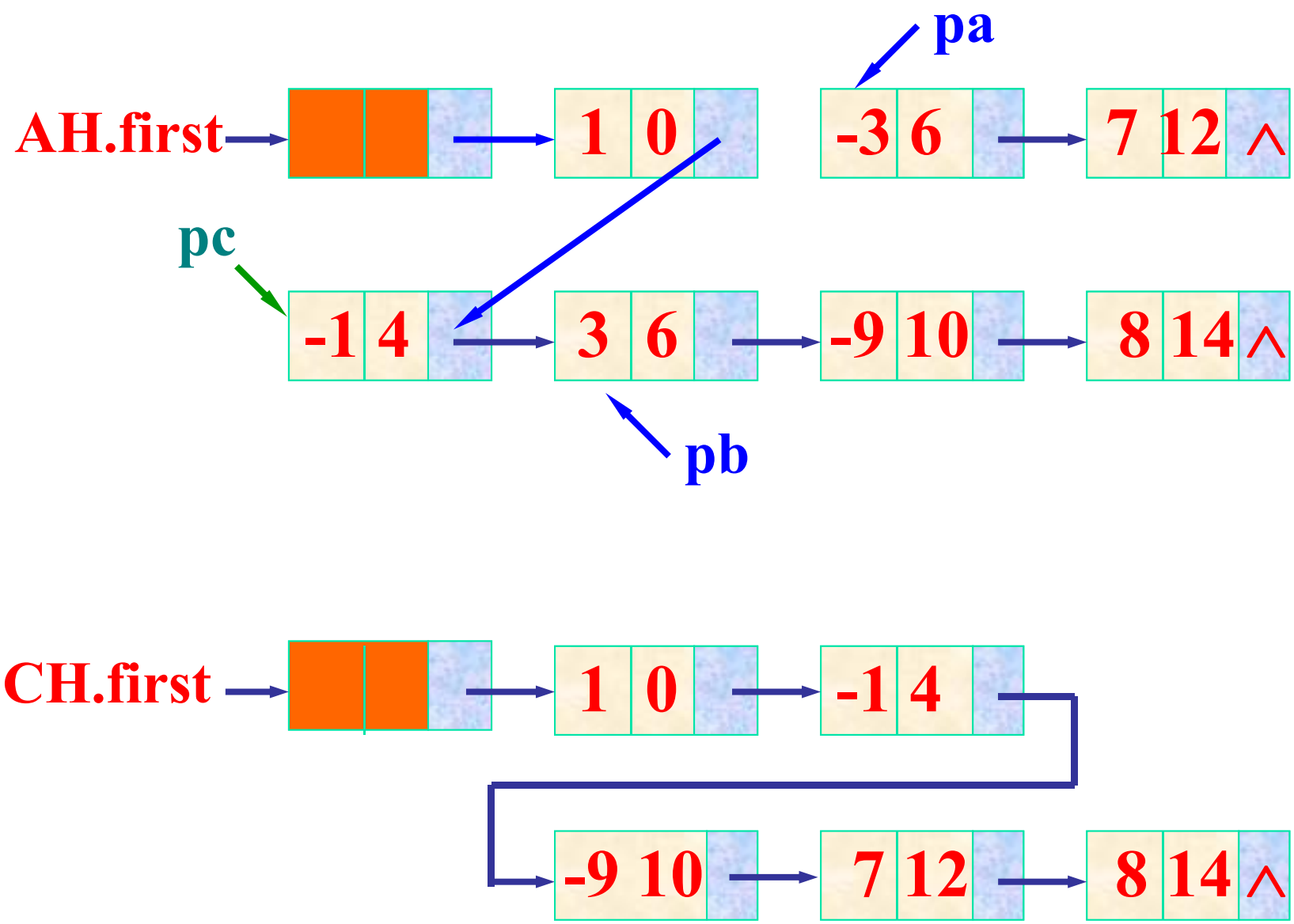
$AH=1-3x^6+7x^{12}$

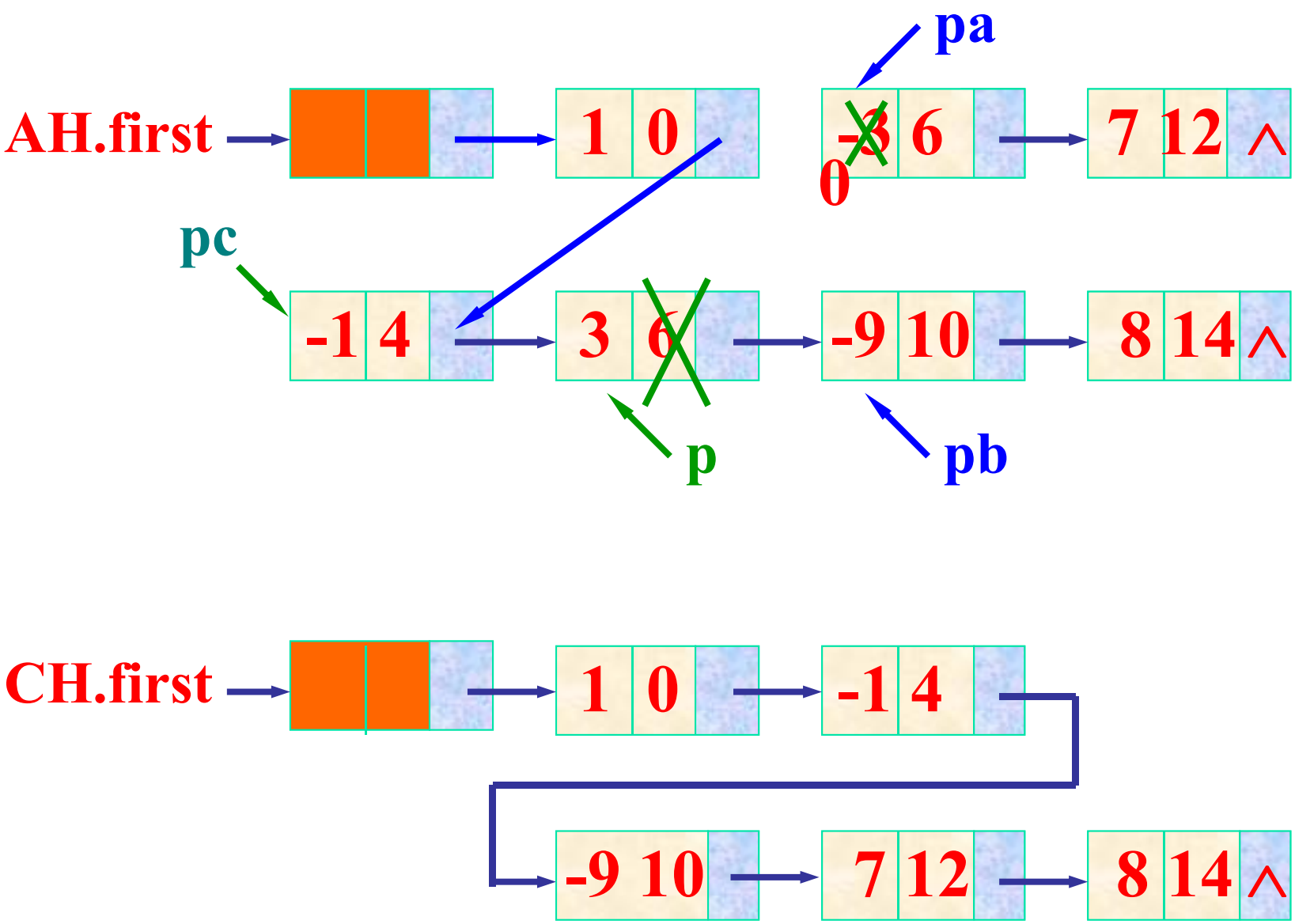
$BH=-x^4+3x^6-9x^{10}+8x^{14}$

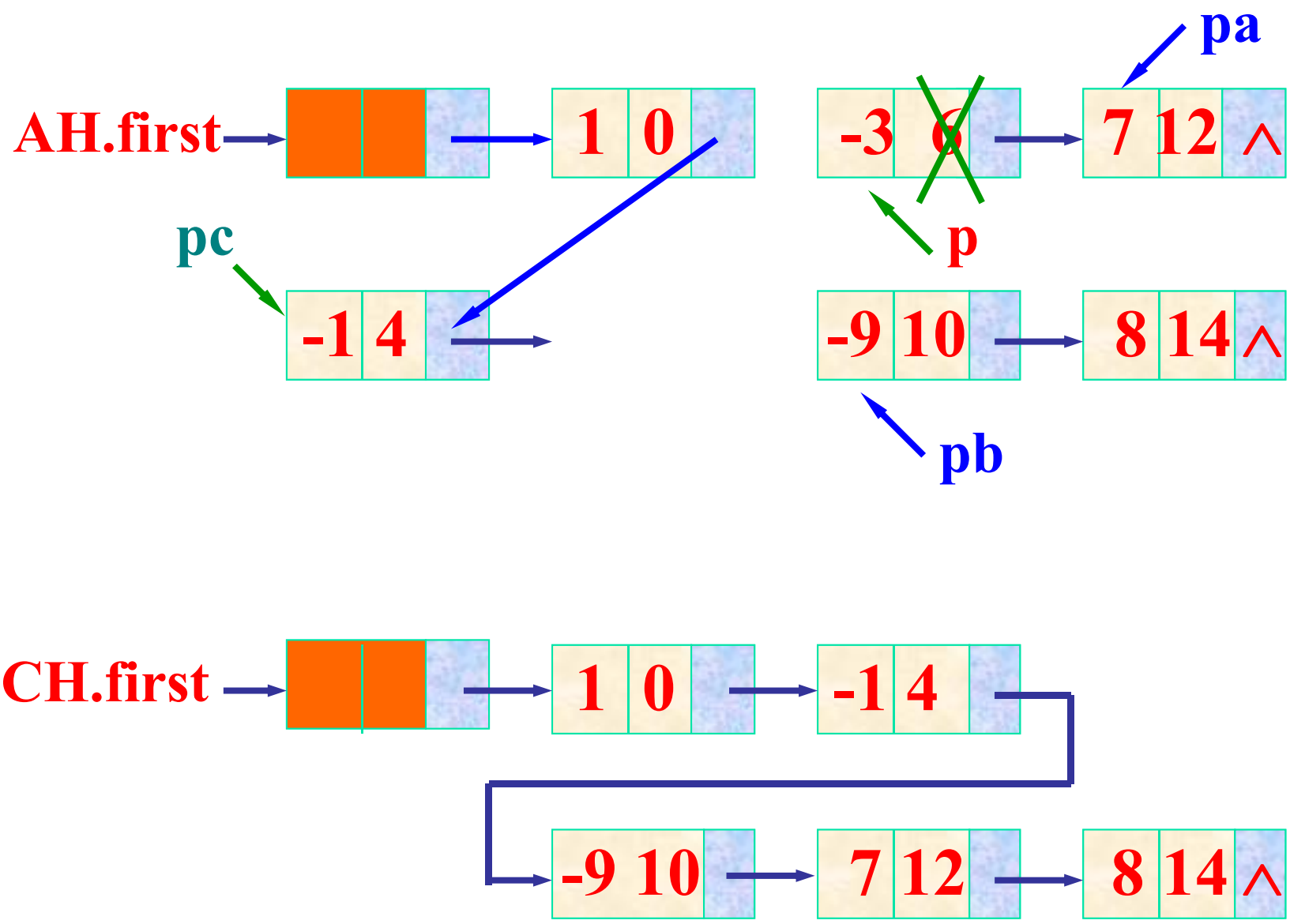


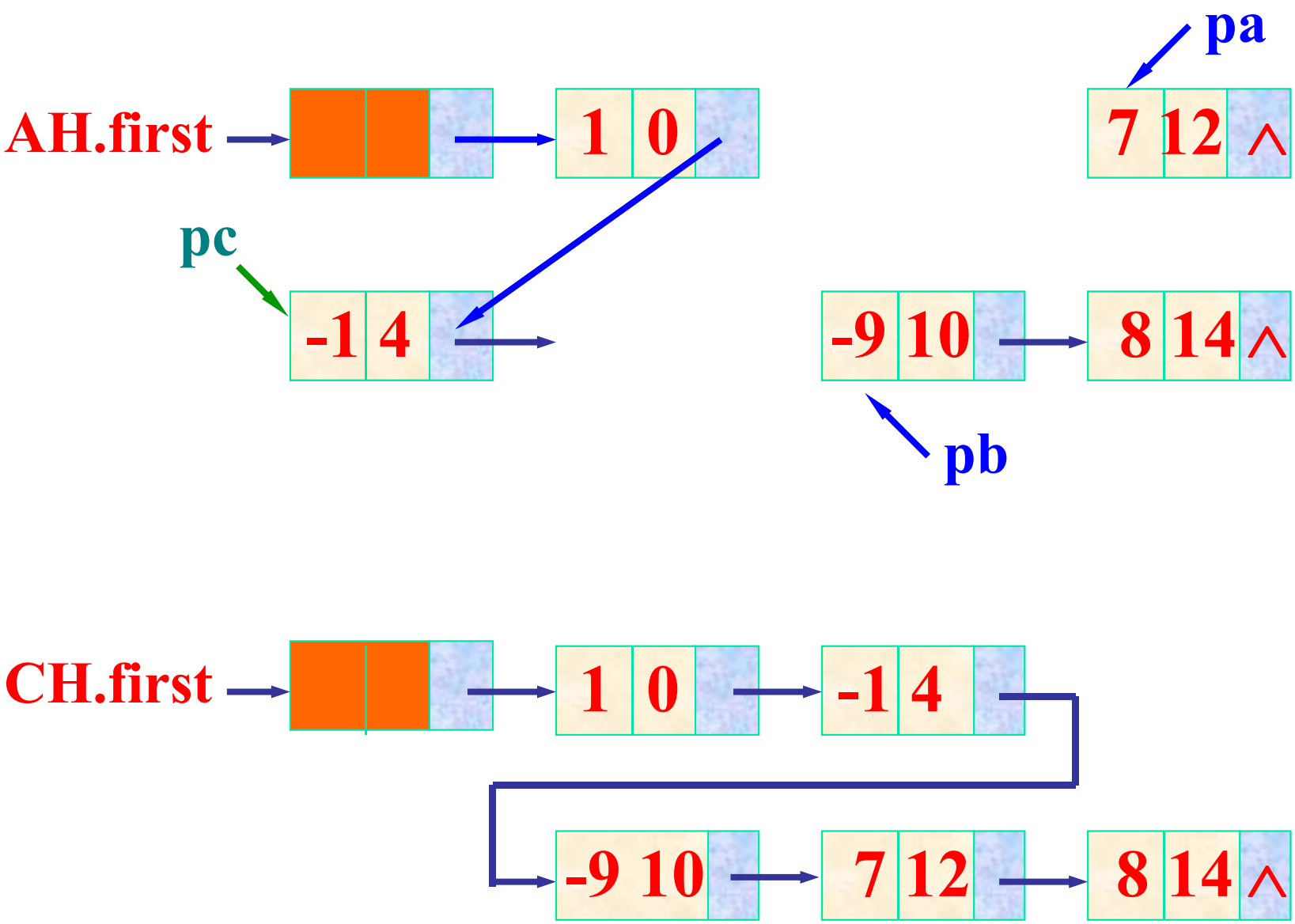

pc
pa
AH.first
1 0
-3 6
7 12 ∧
-1 4
3 6
-9 10
8 14 ∧
BH.first
p
pb
CH.first
1 0
-1 4
-9 10
7 12
8 14 ∧

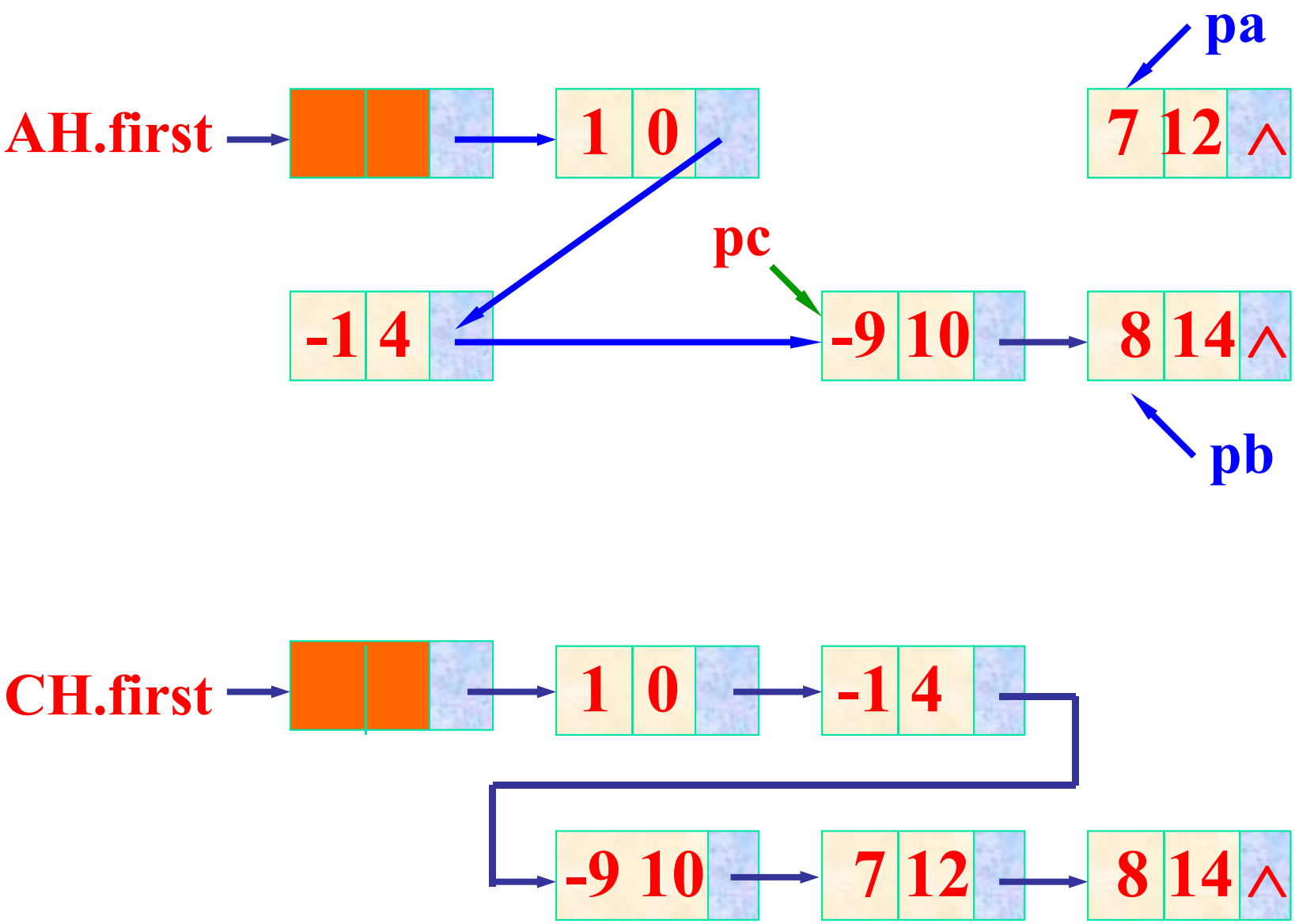

pc
pa
AH.first
1 0
-3 6
7 12 ∧
-1 4
3 6
-9 10
8 14 ∧
pb
CH.first
1 0
-1 4
-9 10
7 12
8 14 ∧

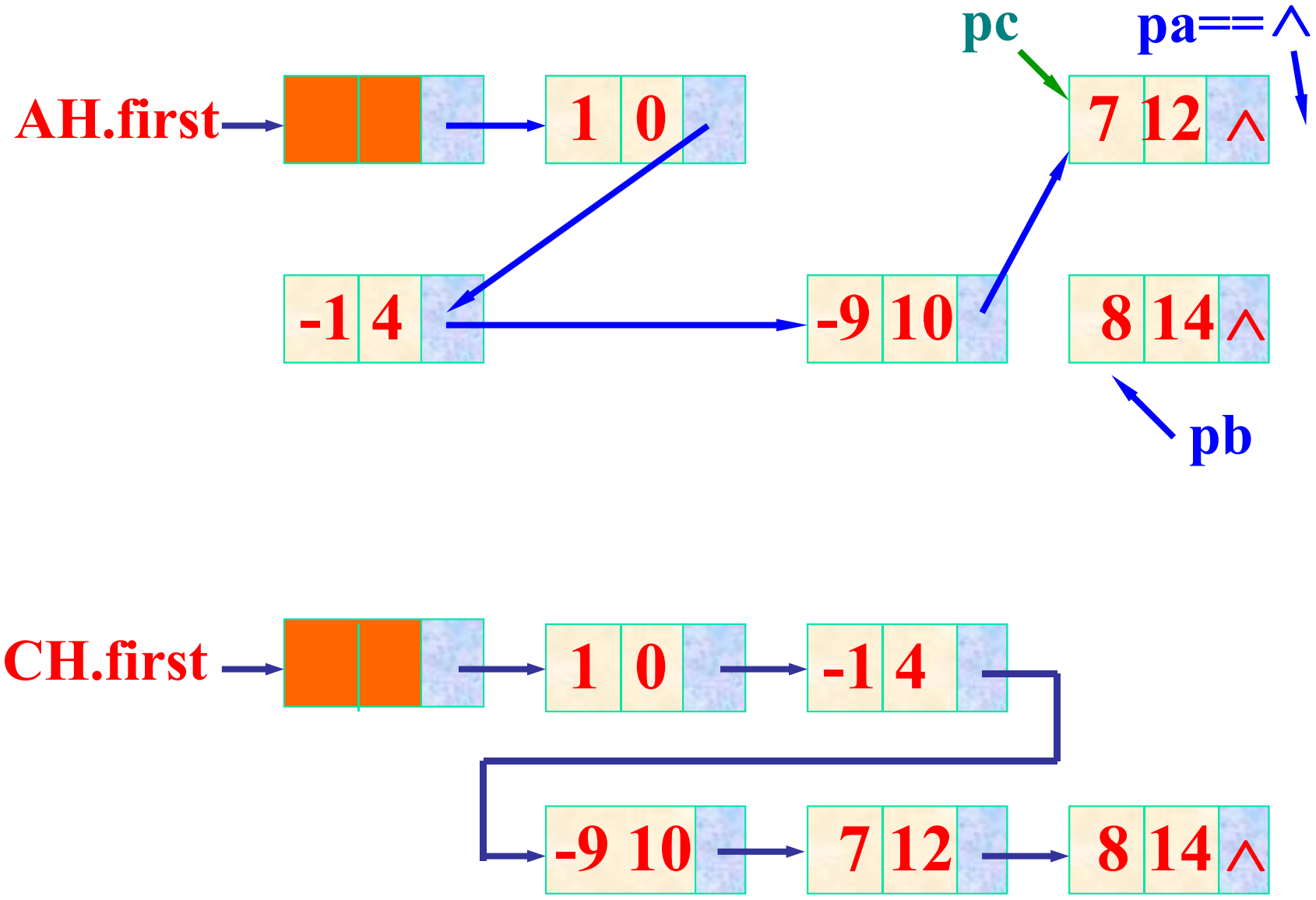

pa
AH.first
1 0
-3 6
7 12 ∧
pc
-1 4
3 6
-9 10
8 14 ∧
pb
CH.first
1 0
-1 4
-9 10
7 12
8 14 ∧

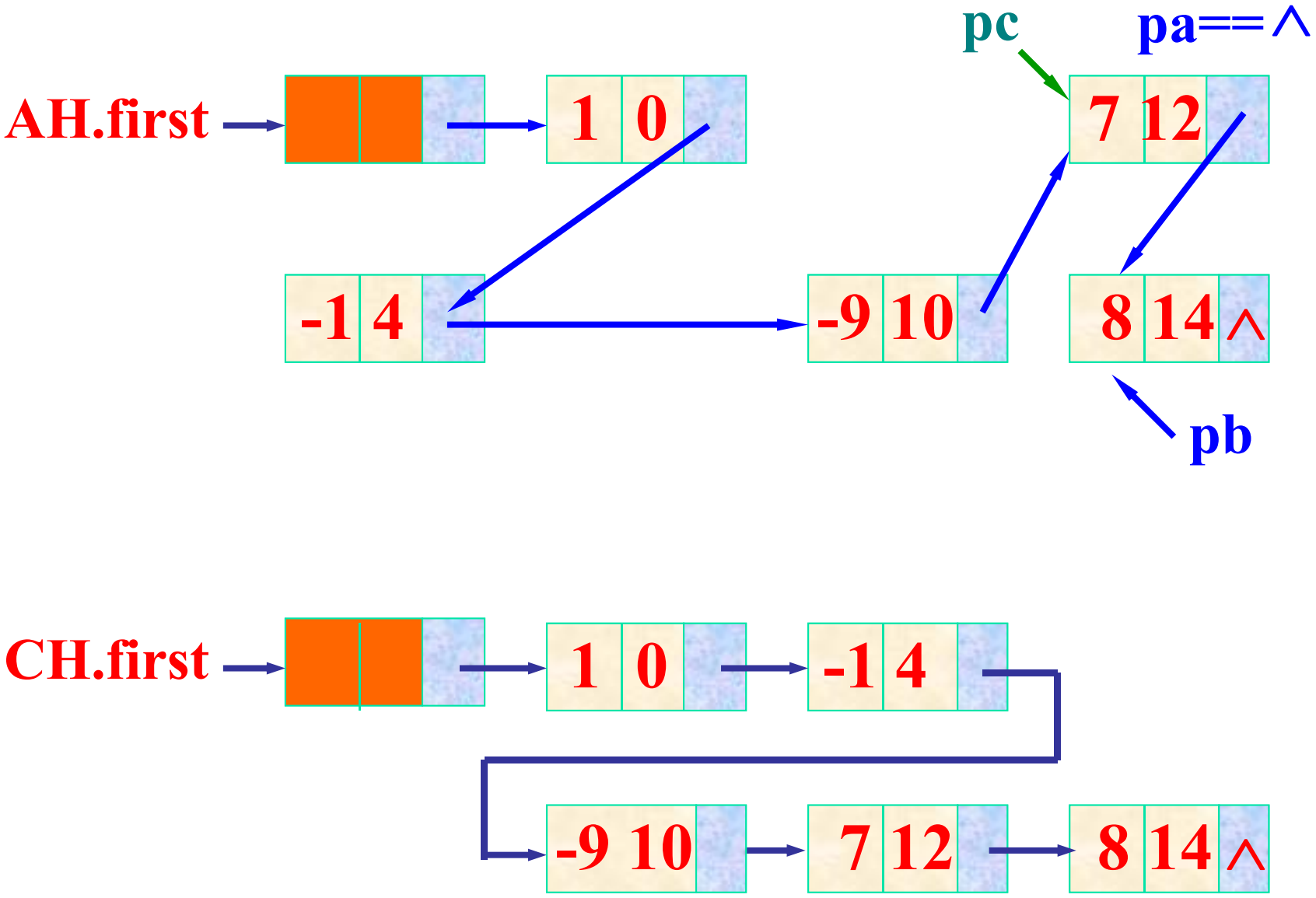

pa
AH.first
1 0
-3 6
0
7 12 ∧
pc
-1 4
3 6
-9 10
8 14 ∧
p
pb
CH.first
1 0
-1 4
-9 10
7 12
8 14 ∧


AH.first
1 0
-3 6
7 12 ∧
pa
pc
p
-1 4
-9 10
8 14 ∧
pb
CH.first
1 0
-1 4
-9 10
7 12
8 14 ∧


pa
AH.first
1 0
7 12 ∧
pc
-1 4
-9 10
8 14 ∧
pb
CH.first
1 0
-1 4
-9 10
7 12
8 14 ∧


pa
AH.first
1 0
7 12 ∧
pc
-1 4
-9 10
8 14 ∧
pb
CH.first
1 0
-1 4
-9 10
7 12
8 14 ∧


pc
pa==^
AH.first
1 0
7 12 ^
-1 4
-9 10
8 14 ^
pb
CH.first
1 0
-1 4
-9 10
7 12
8 14 ^


pc
pa==^
AH.first
1 0
7 12
-1 4
-9 10
8 14 ^
pb
CH.first
1 0
-1 4
-9 10
7 12
8 14 ^

程序2-14 多项式的类定义及相加

```
class PolyLinkedList;  //多项式链表类的前向引用声明
class TermNode {  //多项式链表中项结点的类定义
    friend PolyLinkedList;
    //定义多项式链表类为TermNode类的友元类
    public:
        TermNode( ) { link=NULL; }  //构造函数
    private:
        float coef;  //系数
        int exp;  //指数
        TermNode *link;  //后继指针
};
```

程序2-14 多项式的类定义及相加

```
class PolyLinkedList {  //多项式链表类定义
    public:
        TermNode * Get_NewTermNode(TermNode *pc, int c, int e);
        //为新项建立新结点加入至多项式链表中
        TermNode * PolyLinkedList_Add(TermNode *ah, TermNode *bh);
        //返回分别以ah与bh为头结点的两个多项式链表求和之后的结果
        TermNode * PolyLinkedList_Subtract(TermNode *ah, TermNode *bh);
        //返回分别以ah与bh为头结点的两个多项式链表求差之后的结果
        TermNode * PolyLinkedList_Mult(TermNode *ah, TermNode *bh);
        //返回分别以ah与bh为头结点的两个多项式链表求积之后的结果
        TermNode * PolyLinkedList_Division(TermNode *ah, TermNode *bh);
        //返回分别以ah与bh为头结点的两个多项式链表求商之后的结果
                    ……
    private:
                    ……
};
```

程序2-14 多项式的类定义及相加

```
TermNode * PolyLinkedList ::
                Get_NewTermNode(TermNode *pc, int c, int e)
{  //建立系数为c且指数为e的新结点
   //并把其插入至pc指针所指示结点后面
     TermNode *t;
     t=new <TermNode> (NULL);
     t->coef=c;
     t->exp=e;
     pc->link=t;
     return t;
}
```

程序2-14 多项式的类定义及相加

```
TermNode * PolyLinkedList ::
   PolyLinkedList_Add(TermNode *ah, TermNode *bh)
{  //返回分别以ah与bh为头结点的两个多项式链表
  //求和之后的结果
    TermNode *pa, *pb, *ch, *pc;
    char c;
    ch=new TermNode(NULL);
    //建立结果多项式链表的头结点
    pc=ch;  //以ch为头结点的结果多项式链表检测指针
    pa=ah;  //以ah为头结点的多项式链表检测指针
    pb=bh;  //以bh为头结点的多项式链表检测指针
```

程序2-14 多项式的类定义及相加

```
while (pa!=NULL && pb!=NULL)
{  //对应项两两比较
    if (pa->exp==pb->exp)  c='=';  //对应项指数相等
    else if (pa->exp>pb->exp)  c='>';  //对应项指数不等
    else c='<';
    switch (c)
    {
        case '=':  //pa->exp==pb->exp
                   if (pa->coef+pb->coef!=0)
                   //对应项系数相加非零
                       pc=Get_NewTermNode(pc, pa->coef
                                              +pb->coef, pa->exp);
                       //形成系数为和值的新结点加入至
                       //结果多项式链表中
                   pa=pa->link;  //检测指针后移
                   pb=pb->link;
                   break;
```

程序2-14 多项式的类定义及相加

```
            case '>':  //pa->exp>pb->exp
                       pc=Get_NewTermNode(pc, pb->coef,
                                                  pb->exp);
                       pb=pb->link;
                       break;
            case '<':  //pa->exp<pb->exp
                       pc=Get_NewTermNode(pc, pa->coef,
                                                  pa->exp);
                       pa=pa->link;
                       break;
        }
    }
    while (pa!=NULL)
    { //以ah为头结点的多项式链表中的剩余结点
      //链入结果链表中
       pc=Get_NewTermNode(pc, pa->coef, pa->exp);
       pa=pa->link;
    }
```

程序2-14 多项式的类定义及相加

```
    while (pb!=NULL)
    { //以bh为头结点的多项式链表中的剩余结点
      //链入结果链表中
        pc=Get_NewTermNode(pc, pb->coef, pb->exp);
        pb=pb->link;
    }
    pc->link=NULL;  //结果多项式链表收尾
    pc=ch;
    ch=ch->link;
    delete pc;  //释放最初设置的辅助结点
    return ch;
}
```

```
Polynomial operator + (const Polynomial &ah,
                       const Polynomial &bh) {
    LinkNode <Term> *pa, *pb, *pc, *p;
    ListIterator <Term> Aiter (ah.poly);
    ListIterator <Term> Biter(bh.poly);
    //建立两个多项式对象Aiter、Biter
    pa=pc=Aiter.First( );
    pb=p=Biter.First( );
    //取得ah与bh的表头结点
    pa=Aiter.Next( );  pb=Biter.Next( );
    //pa与pb是链表ah、bh的检测指针
    delete p;  //保留ah的表头结点，删去bh的表头结点
```

```
while (Aiter.NotNull( ) && Biter.NotNull( ))
 //两两比较
    switch (compare(pa->exp, pb->exp)) {
       case '=' :  //pa->exp==pb->exp
           pa->coef=pa->coef+pb->coef;  //系数相加
           p=pb;   pb=Biter.Next( );  delete p;
           //删去原pb所指结点
           if (!pa->coef) {  //相加为0，不要
               p=pa;  pa=Aiter.Next( );  delete p; }
           else {  //相加不为0，加入ch链
               pc->link=pa;  pc=pa;
               pa=Aiter.Next( );  }
           break;
```

```
        case '>':
        //pa->exp>pb->exp
              pc->link=pb;  pc=pb;
              pb=Biter.Next( );   break;
        case '<':
        //pa->exp<pb->exp
             pc->link=pa;  pc=pa;
             pa=Aiter.Next( );
    }
    if (Aiter.NotNull( ))  pc->link=pa;
    //剩余部分链入 ch链
    else  pc->link=pb;
}
```

- 设两个多项式链表的长度分别为***m***与***n***，则总的比较次数为**O(*m*+*n*)**。

- 有了多项式的加法，则可容易地定义多项式的减法、乘法和除法。
  - 两个多项式的乘法可以转化为一系列多项式的加法来实现。

线性链表的其他变形

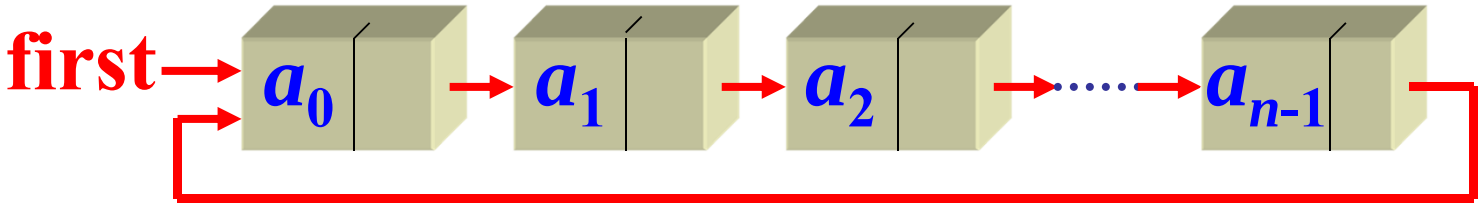
循环链表 (Circular List)

- 循环链表是单链表的变形。
- 循环链表的最后一个结点的**link**指针不为**NULL**，而是指向表的前端。
- 为简化操作，在循环链表中往往加入表头结点。
- 循环链表的特点是：只要知道表中某一结点的地址，就可搜寻到所有其它结点的地址。
- 检查指针**current**是否到达链表的链尾时：

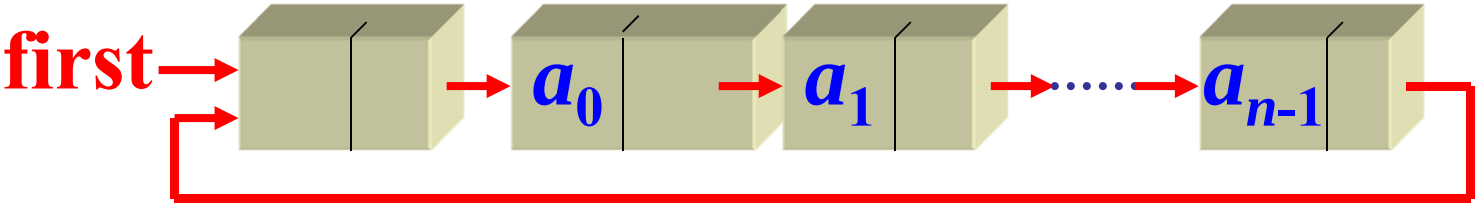
  **不是判断是否current->link==NULL;

  **而是判断是否current->link==first。

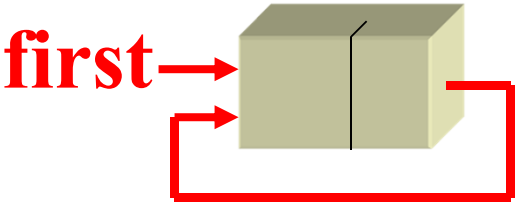
- 循环链表的示例



- 带表头结点的循环链表



(非空表)



(空表)

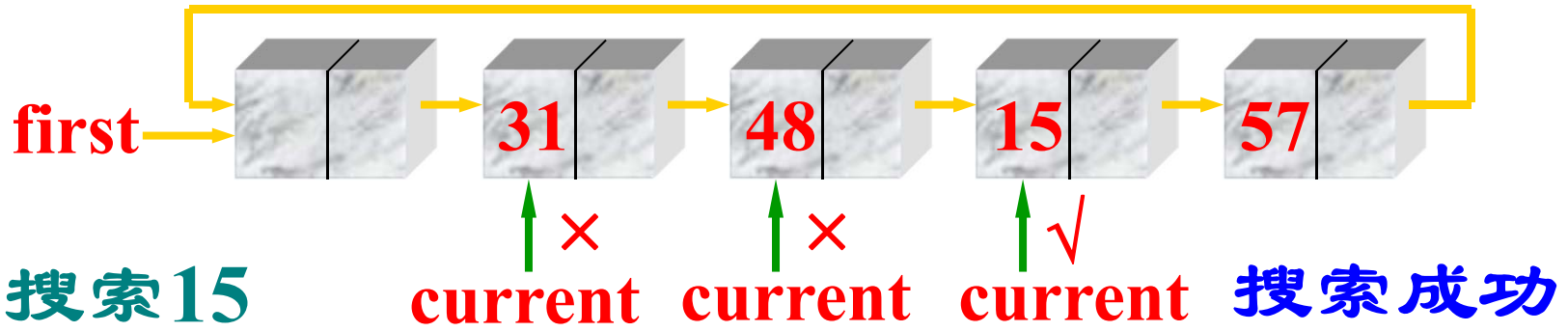
循环链表类定义

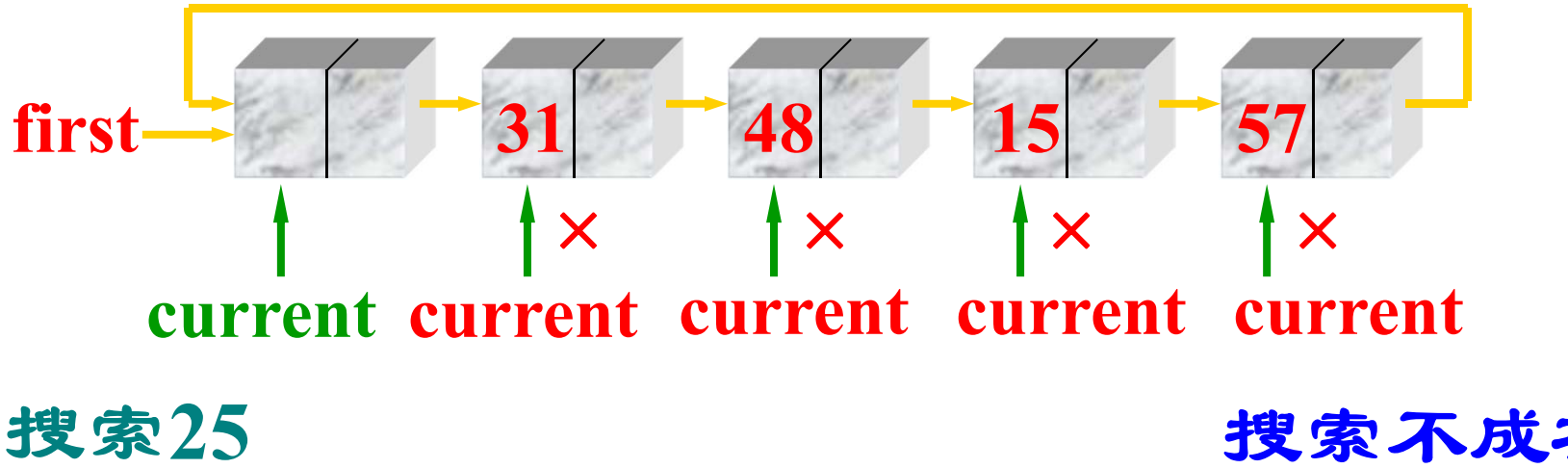
```
template <class Type> class CircList;
template <class Type> class CircLinkNode {
    friend class CircList <Type>;
    public:
        CircLinkNode(Type d=0,
                CircLinkNode <Type> *next=NULL)
                    : data(d), link(next) { }  //构造函数
    private:
        Type data;  //结点数据
        CircLinkNode <Type> *link;  //链接指针
};
```

```
template <class Type> class CircList {
//循环链表类
   private:
      LinkNode <Type> *first, *last, *current;
      //循环链表的表头指针、尾指针和当前元素指针
   public:
      CircList(const Type &value);
      CircList(CircList <Type> &L);
      ~CircList( );
      int Length( ) const;
      CircLinkNode <Type> * GetHead( ) const;
      void SetHead(CircLinkNode <Type> *p)
      { first=p; }
```

```
    CircLinkNode <Type> * Search(Type value);
    CircLinkNode <Type> * Locate(int i);
    Type * GetData(int i);
    void SetData(int i, Type &x);
    bool Insert(int i, Type &x);
    bool Remove(int i, Type &x);
    bool IsEmpty( ) const
          { return first->link==first ? true : false; }
    void Sort( );
    void Input( );
    void Output( );
    CircList <Type> operator = (CircList <Type> &L);
};
```

循环链表的搜索算法


first
31
48
15
57
×
×
√
搜索15
current
current
current
搜索成功

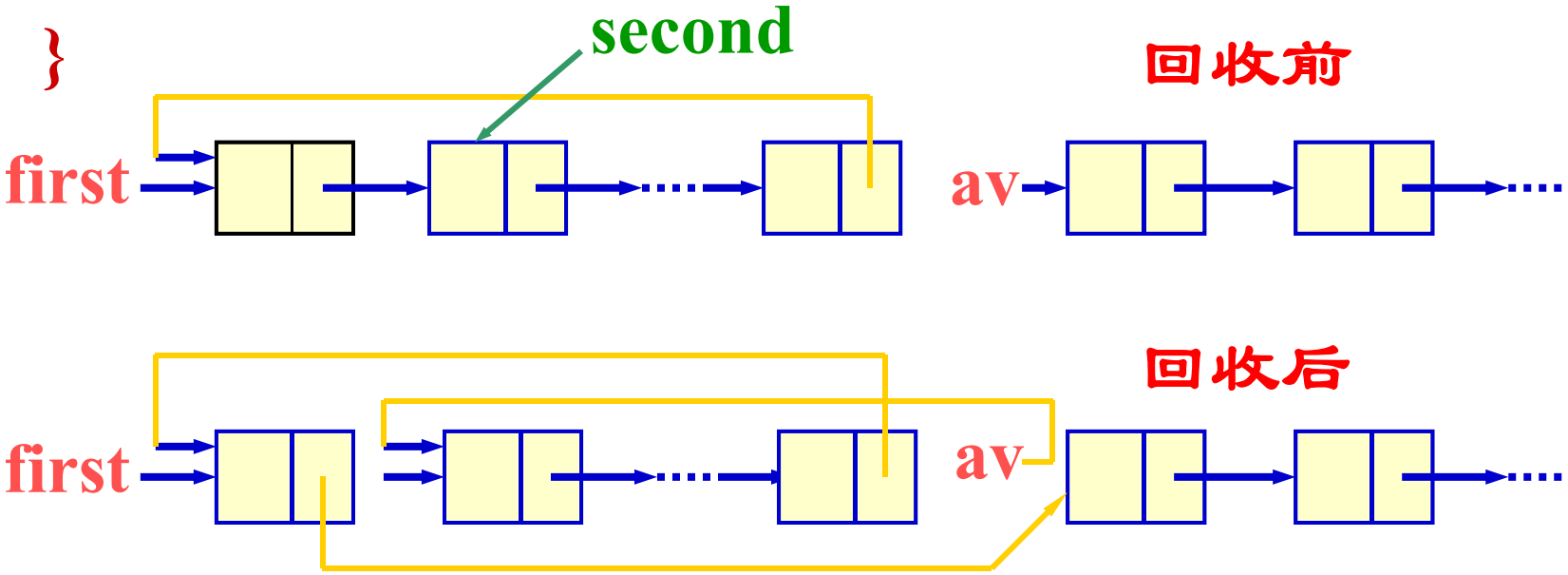

first
31
48
15
57
current
current
current
current
current
×
×
×
×
搜索25
搜索不成功

循环链表的搜索算法

```
template <class Type> CircLinkNode <Type>
        * CircList <Type> :: Search (Type value)
{ //在链表中从头搜索其数据值为value的结点
    current=first->link;
    while (current!=first &&
                    current->data!=value)
       current=current->link;
    return current;
}
```

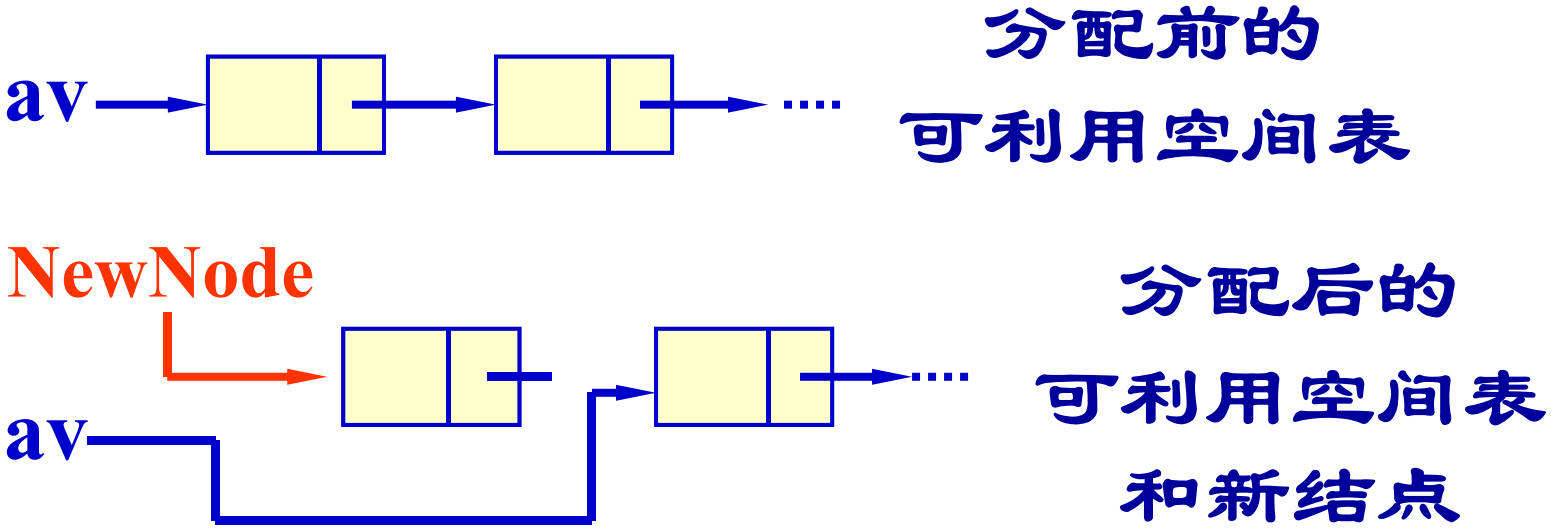
利用可利用空间表回收循环链表

```
if ( first!=NULL ) {
    CircLinkNode <Type> *second=first->link;
    first->link=av;  av=second;
    first=NULL;
}
```

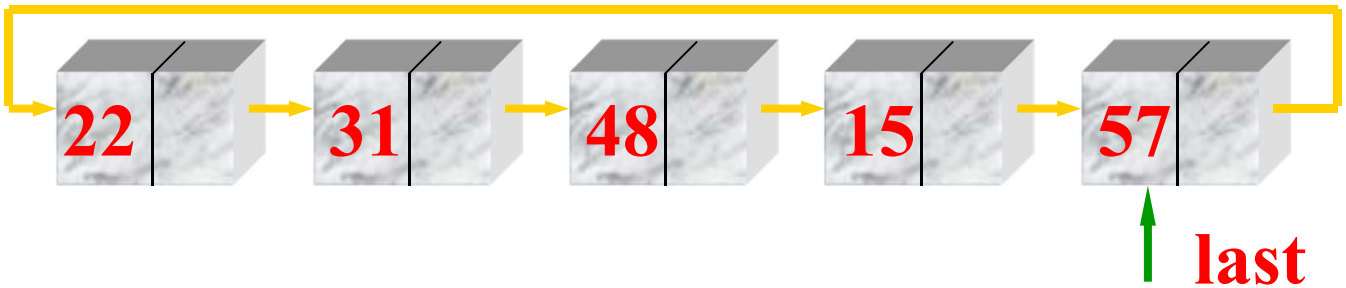


从可利用空间表分配结点

```
if (av==NULL)
    Newnode=new CircLinkNode <Type>;
else { NewNode=av;  av=av->link; }
```



带尾指针的循环链表



如果插入与删除仅在链表的两端发生，可采用带表尾指针的循环链表结构。

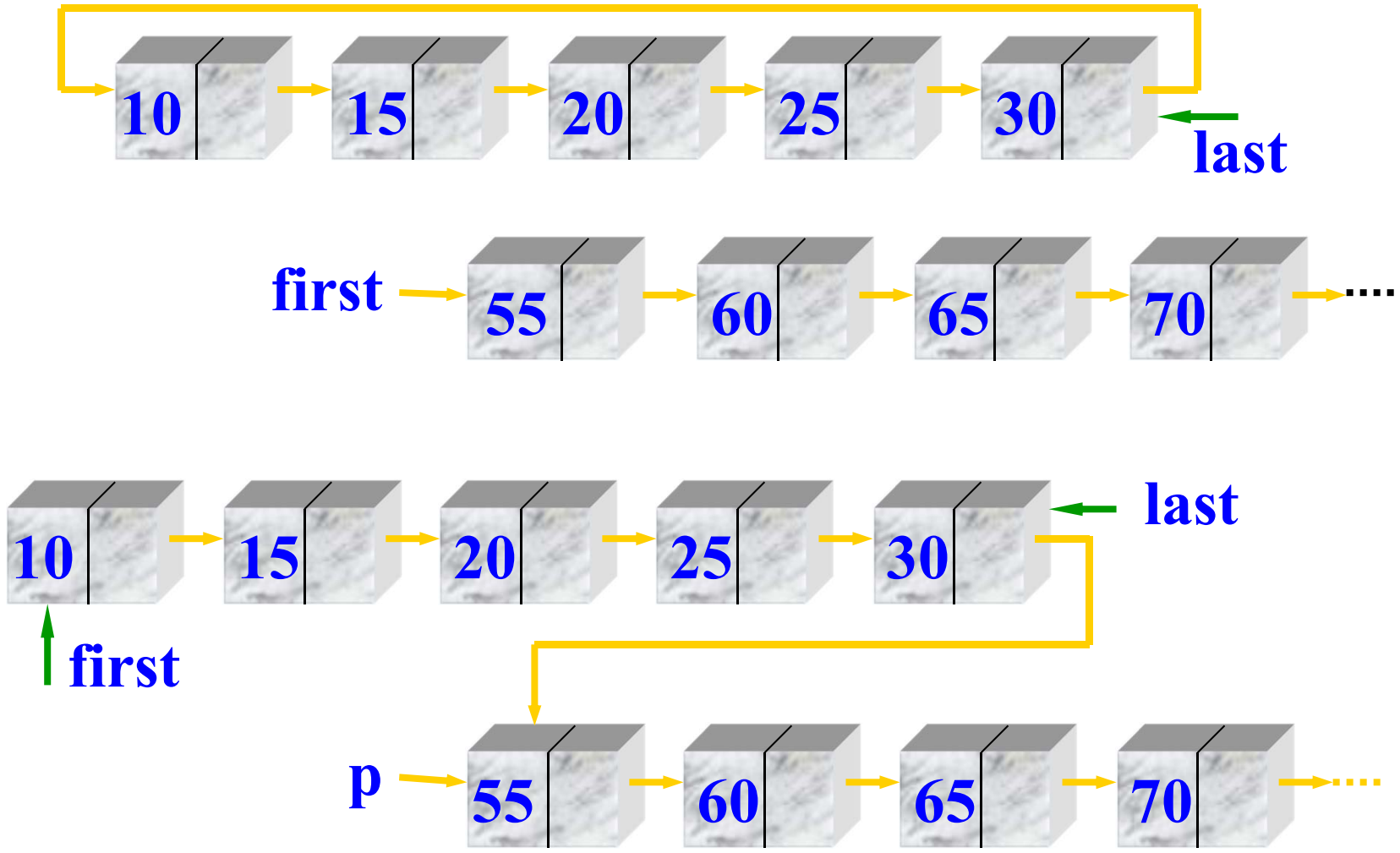
在表尾插入，时间复杂性O(1)；

在表尾删除，时间复杂性O(*n*)；

在表头插入，相当于在表尾插入；

在表头删除，时间复杂性O(1)。

将循环链表链入单链表链头

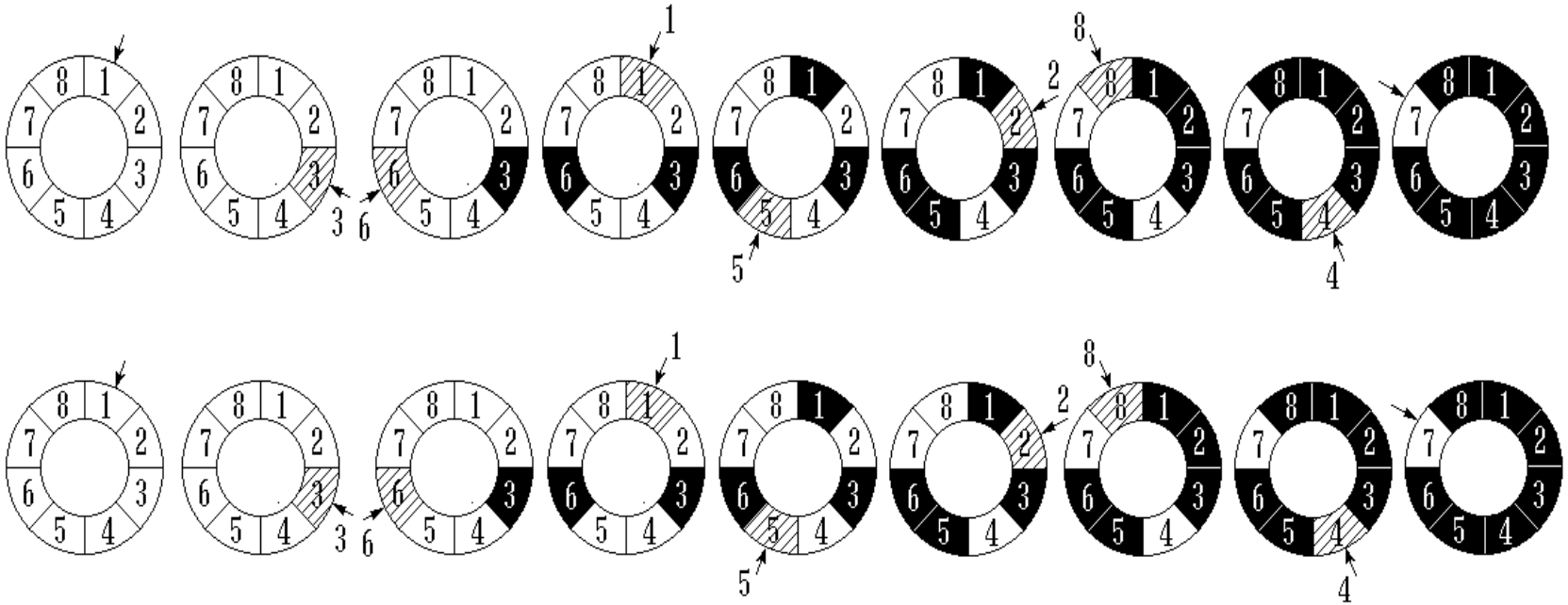


用循环链表求解约瑟夫问题

- 约瑟夫问题的提法

  n个人围成一个圆圈，首先第**1**个人从**1**开始一个人一个人顺时针报数，报到第***m***个人，令其出列。然后再从下一 个人开始，从**1**顺时针报数，报到第***m***个人，再令其出列，…，如此下去，直到圆圈中只剩一个人为止。此人即为优胜者。

- 例如，*n*=8　*m*=3



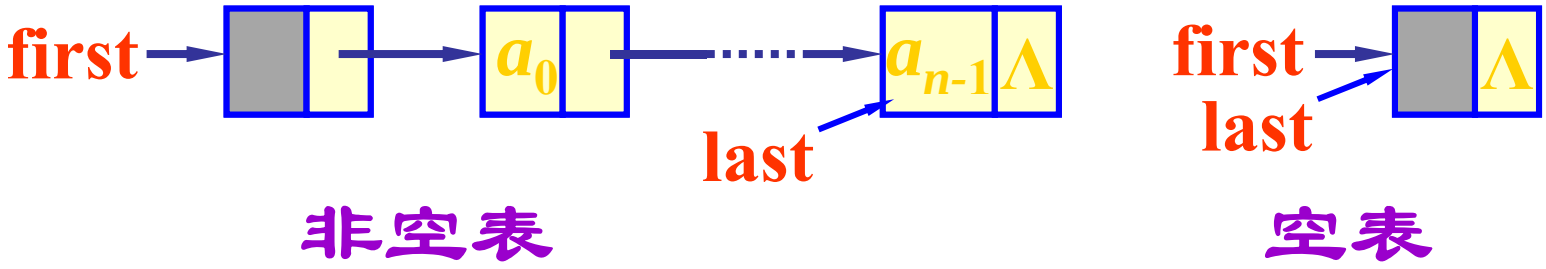
约瑟夫问题的解法

```
#include <iostream.h>
#include "CircList.h"
template <Type> void Josephus (CircList <Type> &Js,
                                        int n, int m)
{
   CircLinkNode <Type> *p=Js.GetHead( ), *pre=NULL;
   for (int i=0; i<n-1; i++) {  //执行n-1次
      for (int j=0; j<m-1; j++)
      { pre=p;  p=p->link; }
      cout<<"出列的人是"<<p->data<<endl;
      pre->link=p->link;  delete p;
      p=pre->link;
   }
}
```

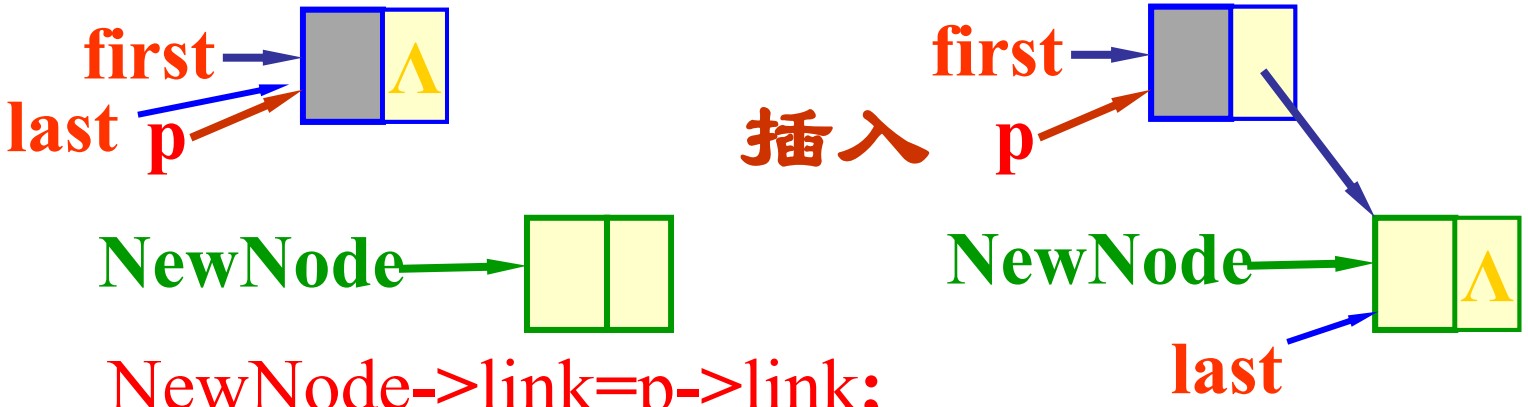
```
void main( ) {
    CircList <int> clist;
    int n, m;
    cout<<“输入游戏者人数和报数间隔：”;
    cin>>n>>m;  //形成约瑟夫环
    for (int i=1; i<=n; i++) clist.Insert(i);
    clist.Josephus(n, m);  //解决约瑟夫问题
}
```

带表头结点的单链表

- 表头结点位于表的最前端，本身不带数据，仅标志表头。
- 设置表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。

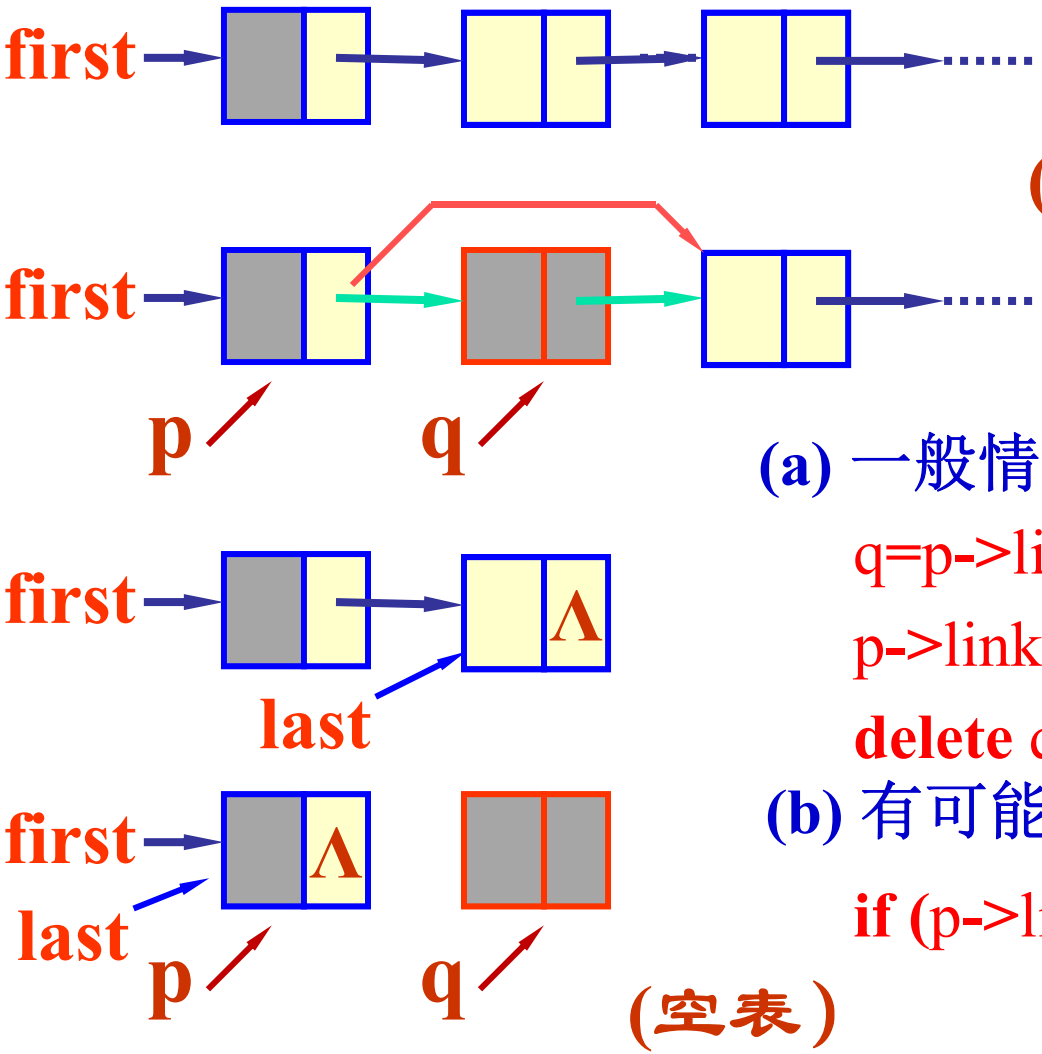


在带表头结点的单链表最前端插入新结点




```
NewNode->link=p->link;
if (p->link==NULL)  last=NewNode;
p->link=NewNode;
```

从带表头结点的单链表中删除最前端的结点



(a) 一般情况：

```
q=p->link;
p->link=q->link;
delete q;
```

(b) 有可能删除后链表变为空表：

```
if (p->link==NULL)  last=p;
```

用模板定义的单链表类

```
template <class Type> class List;
template <class Type> class LinkNode {
    friend class List <Type>;
    private:
        Type data;  //结点数据
        LinkNode <Type> *link;  //结点链接指针
    public:
        LinkNode( ) : link(NULL) { }  //构造函数
        LinkNode(Type item) : data(item), link(NULL) { }
```

```
    LinkNode <Type> * NextNode( ) { return link; }
    //取得结点的下一结点地址
    Type InsertAfter(ListNode <Type> *p);
    //当前结点插入
    LinkNode <Type> * GetNode(const Type &item,
                              LinkNode <Type> *next);
    //以item和next建立一个新结点
    LinkNode <Type> * RemoveAfter( );
    //删除当前结点的下一结点
};
```

```
template <class Type> class List {
//链表类
    private:
        LinkNode <Type> *first, *last, *current;
        //链表的表头指针、尾指针和当前元素指针
    public:
        List( ) { first=new LinkNode <Type>; }
        List(const Type &value)
              { last=first=new LinkNode <Type> (value); }
        List (List <Type> &L);
        ~List( ) { MakeEmpty( );  delete first; }
        void MakeEmpty( );
        int Length( ) const;
       LinkNode <Type> * GetHead( ) const { return first; }
        void SetHead(LinkNode <Type> *p) { first=p; }
```

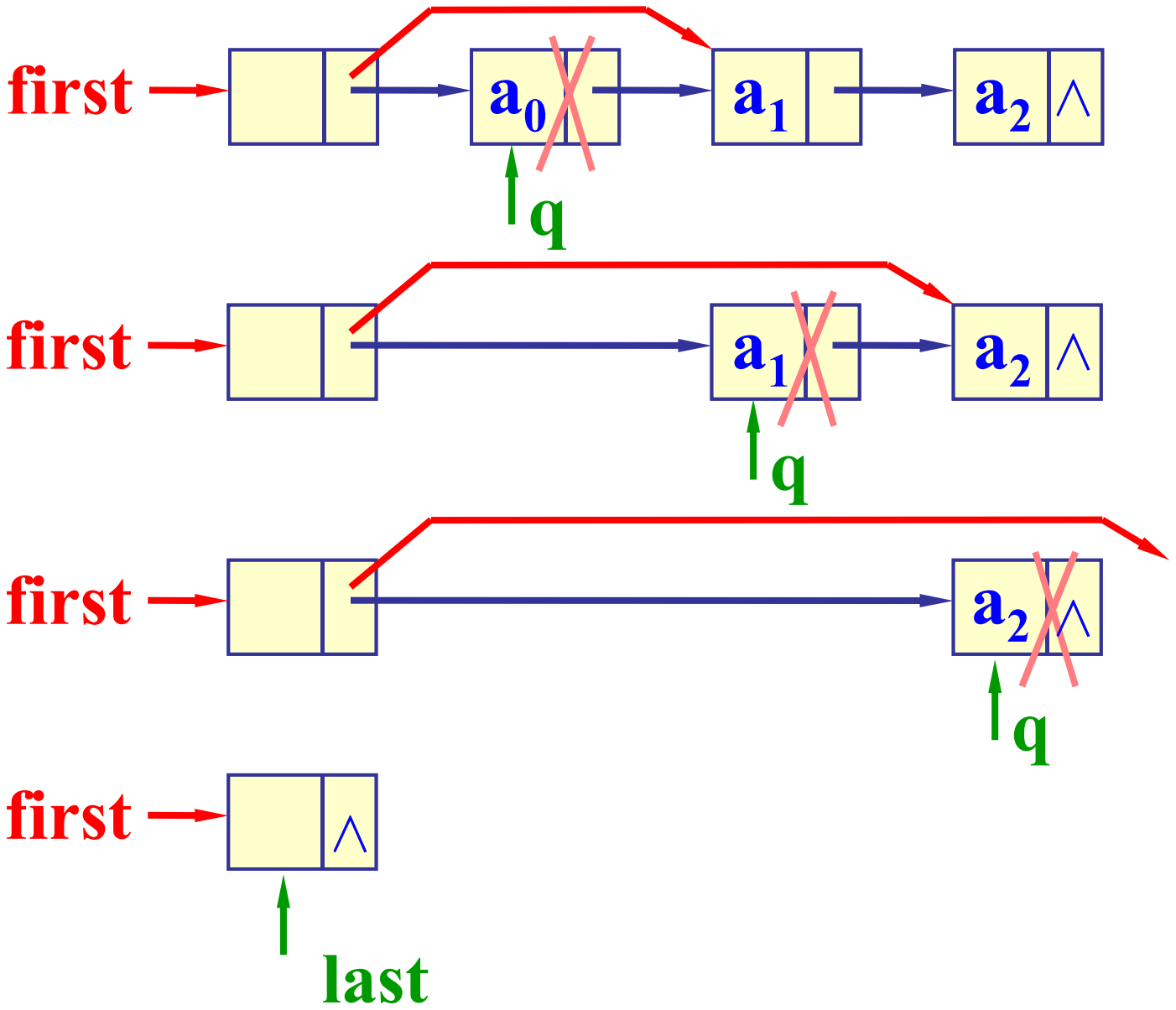
```
    LinkNode <Type> * Search(Type x);
    LinkNode <Type> * Locate(int i);
    Type * GetData (int i);
     void SetData(int i, Type &x);
    bool Insert(int i, Type &x);
    bool Remove(int i, Type &x);
    bool IsEmpty( ) const
            { return first->link==NULL ? true : false; }
    bool IsFull( ) const  { return false; }
    void Sort( );
    void Input( );
    void Output( );
    List <Type> operator = (List <Type> &L);
};
```

链表类部分操作的实现

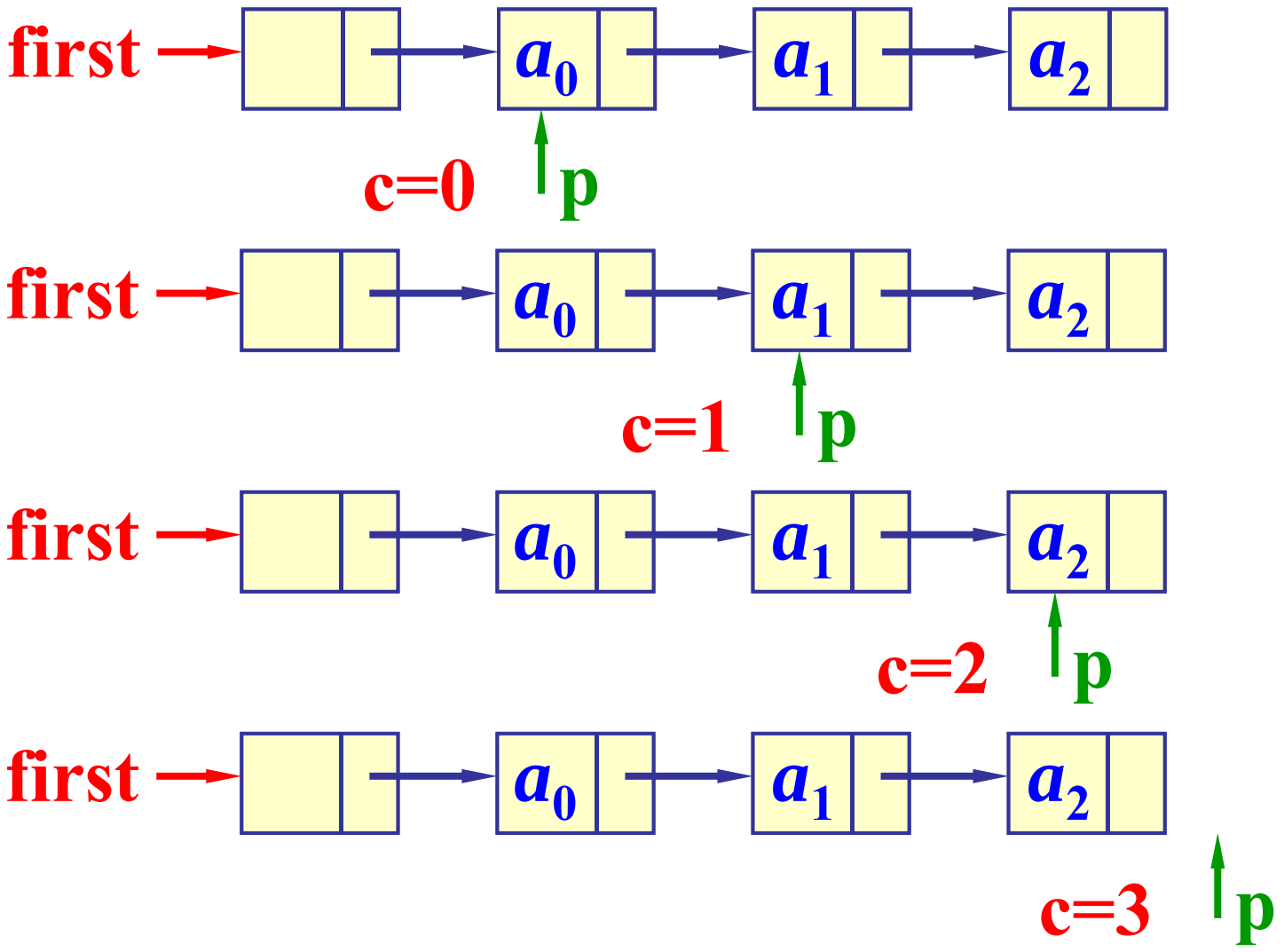
```
template <class Type> void LinkNode <Type> ::
    InsertAfter(LinkNode <Type> *p) {
//将p所指示结点链接成为当前结点(*this)的后继结点
    p->link=link;  link=p;
}

template <class Type> LinkNode <Type>
          * LinkNode <Type> :: GetNode ( const Type &item,
                    LinkNode <Type> *next=NULL) {
//以item和next为参数，建立新结点并返回新结点地址
    LinkNode <Type> *NewNode=
                              new LinkNode <Type> (item);
    NewNode->link=next;  return NewNode;
}
```

```
template <class Type> * LinkNode <Type> ::
        RemoveAfter( ) {
//从链中摘下当前结点的下一结点删除
//并返回其地址
   LinkNode <Type> *tempptr=link;
   //保存被删除结点的地址
   if (link==NULL)  return NULL;
   //当结点无后继，返回NULL
   link=tempptr->link;  //将被删结点从链中摘下
   return tempptr;
}
```

```
template <class Type>
        List <Type> :: List(List <Type> &L) {
    Type value;
    LinkNode <Type> *srcptr=L.GetHead( );
    LinkNode <Type> *destptr=first=new LinkNode <Type>;
    while (srcptr->link!=NULL) {
        value=srcptr->link->data;
        destptr->link=new LinkNode <Type> (value);
        destptr=destptr->link;
        srcptr=srcptr->link;
    }
    destptr->link=NULL;
}
```


first
a_0
a_1
a_2 ∧
q
first
a_1
a_2 ∧
q
first
a_2 ∧
q
first
∧
last

```
template <class Type>
        void List <Type> :: MakeEmpty ( ) {
//删去链表中除表头结点外的所有其它结点
//将链表置为空表
   LinkNode <Type> *q;
   while (first->link!=NULL) {
   //当链不空时，删去链中所有结点
       q=first->link;  first->link=q->link;
       delete q;  //循链逐个删除，保留一个表头结点
   }
   last=first;  //表尾指针指向表头结点
}
```


first
a_0
a_1
a_2
c=0
p
first
a_0
a_1
a_2
c=1
p
first
a_0
a_1
a_2
c=2
p
first
a_0
a_1
a_2
c=3
p

```
template <class Type>
        int List <Type> :: Length( ) const {
//求单链表的长度
    LinkNode <Type> *p=first->link;
    //检测指针p指示第一个数据结点
    int count=0;
    while (p!=NULL) {  //循链扫描，  寻找链尾
        p=p->link;  count++;
    }
    return count;
}
```

```
template <class Type> LinkNode <Type>
        * List <Type> :: Search (Type x) {
//在链表中从头搜索其数据值为x的结点
//搜索成功时函数返回该结点地址，否则返回NULL
    LinkNode <Type> *p=first->link;
    while (p!=NULL&&p->data!=x)
        p=p->link;
    //循链找含x结点
    return p;
}
```

```
template <class Type> LinkNode <Type>
        * List <Type> :: Locate(int i) {
//定位函数：返回表中第i个元素的地址
//若i<0或i超出表中结点个数，则返回NULL
   if (i<-1)  return NULL;  //i值不合理
   LinkNode <Type> *p=first;  int j=0;
   //检测指针p指向表中第一个结点
   while (p!=NULL && j<i)
   { p=p->link;  j++; }  //寻找第i个结点的地址
   return p;
   //返回第i个结点地址，若返回NULL，表示i值太大
}
```

```
template <class Type>
        bool List <Type> :: Insert(int i, Type &x) {
//将新元素x插入在链表中第i个位置
    LinkNode <Type> *p=Locate(i);
    //定位第i个元素
    if (p==NULL)  return false;
    //参数i的值不合理，函数返回false
    ListNode <Type> *NewNode=p->GetNode(x, p->link);
    //创建含x的结点
    if (p->link==NULL)   last=NewNode;
    p->link=NewNode;
    return true;  //插入成功，函数返回true
}
```

```
template <class Type>
        bool  List <Type> :: Remove(int i, Type &x) {
//将链表中的第i个元素删去
    LinkNode <Type> *p=Locate(i-1), *q;
     //p定位于第i-1个元素
     if (p==NULL || p->link==NULL )  return false;
     //i的值不合理或空表，返回flase
     q=p->link;  p->link=q->link;
     //q指向被删结点，重新拉链
     x=q->data;
     //取出被删结点中的数据值
     if (q==last)  last=p;  //删除尾结点时，表尾指针修改
     delete q;
     return true;
}
```

```
template <class Type> List <Type> & List <Type> ::
operator = (List <Type> &L) {
    Type value;
    LinkNode <Type> *srcptr=L.GetHead( );
    LinkNode <Type> *destptr=first
                                    =new LinkNode <Type>;
    while (srcptr->link!=NULL) {
        value=srcptr->link->data;
        destptr->link=new LinkNode <Type> (value);
        destptr=destptr->link;
        srcptr=srcptr->link;
    }
    destptr->link=NULL;
    return *this;
}
```

程序2-15 带表头结点的循环链表表示的多项式相加

```
TermNode * PolyLinkedList ::
        PolyLinkedList_Add(TermNode *ah, TermNode *bh)
{ //返回分别以ah与bh为表头结点的两个多项式循环链表
  //求和之后的结果
    TermNode *pa, *pb, *ch, *pc;
    char c;
    ch=new TermNode (NULL);
    //建立结果多项式循环链表的表头结点
    ch->exp=-1;
    pc=ch;  //以ch为表头结点的结果多项式循环检测指针
    pa=ah->link;
    //以ah为表头结点的多项式循环链表检测指针
    pb=bh->link;
    //以bh为表头结点的多项式循环链表检测指针
```

程序2-15 带表头结点的循环链表表示的多项式相加

```
while (pa->exp!=-1 || pb->exp!=-1)  //(pa!=ah || pb!=bh)
{  //对应项两两比较
    if (pa->exp==pb->exp)  c='=';  //对应项指数相等
    else if (pa->exp>pb->exp)   c='>';  //对应项指数不等
    else c='<';
    switch (c)
    {
        case '=':  //pa->exp==pb->exp
           if (pa->coef+pb->coef!=0)  //对应项系数相加非零
              pc=Get_NewTermNode(pc, pa->coef+pb->coef,
                                                    pa->exp);
              //形成系数为和值的新结点加入至
              //结果多项式循环链表中
              pa=pa->link;  //检测指针后移
              pb=pb->link;
              break;
```

程序2-15 带表头结点的循环链表表示的多项式相加

```
            case '>':  //pa->exp>pb->exp
                pc=Get_NewTermNode(pc, pb->coef, pb->exp);
                pb=pb->link;
                break;
            case '<':  //pa->exp<pb->exp
                pc=Get_NewTermNode(pc, pa->coef, pa->exp);
                pa=pa->link;
                break;
        }
    }
    while (pa->exp!=-1)  //(pa!=ah)
    {  //以ah为头结点的多项式循环链表中的剩余结点
       //链入结果多项式循环链表中
        pc=Get_NewTermNode(pc, pa->coef, pa->exp);
        pa=pa->link;
    }
```

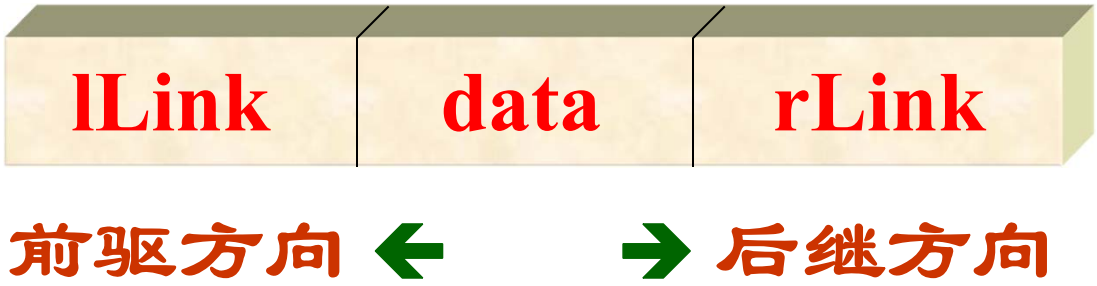
程序2-15 带表头结点的循环链表表示的多项式相加

```
    while (pb->exp!=-1)  //(pb!=bh)
    { //以bh为头结点的多项式循环链表中的剩余结点
      //链入结果多项式循环链表中
        pc=Get_NewTermNode(pc, pb->coef, pb->exp);
        pb=pb->link;
    }
    pc->link=ch;  //结果多项式循环链表收尾
    return ch;
}
```

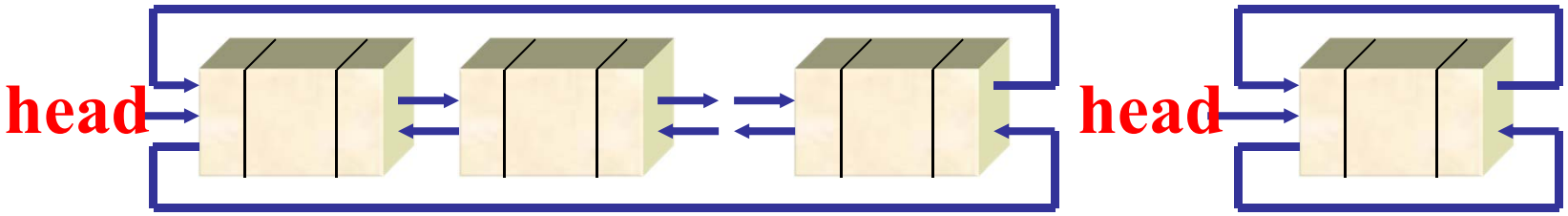
- 在单链表中，搜索一个指定结点的后继结点非常方便。
  - 只要该结点的***link***域的内容不为空，就可通过***link***域找到该结点的后继结点地址。
- 在单链表中，搜索一个指定结点的前驱结点非常不容易。
  - 必须从链头开始，沿***link***链顺序检测，直到某一结点的后继结点为该指定结点，则此结点即为该指定结点的前驱结点。
- 在一个应用问题中，经常要求检测指针向前驱和后继方向移动。
  - 需要保证移动的时间复杂度达到最小。

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。
- 双向链表每个结点结构



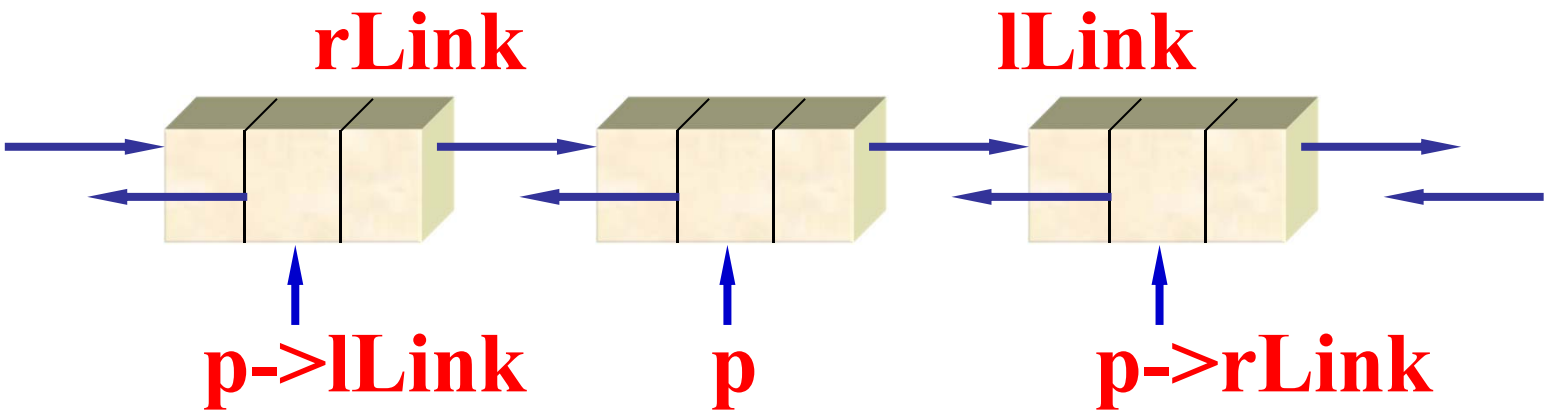
- 双向链表通常采用带表头结点的循环链表形式。



非空表　　空表

- 结点指向

p=p->lLink->rLink=p->rLink->lLink



程序2-16 循环双向链表的类定义

```
template <class Type> class DblList;
//循环双向链表类的前向引用声明
template <class Type> class DblNode {
//循环双向链表结点类定义
    friend class DblList <Type>;
    private:
       Type data;  //数据域
       DblNode <Type> *llink, rlink;
       //前驱（左）指针域与后继（右）指针域
    public:
       DblNode(Type value) : data(value), llink(NULL),
                                          rlink(NULL) { }  //构造函数
       DblNode(Type value, DblNode <Type> *left_pointer,
            DblNode <Type> *right_pointer) : data(value),
               llink(left_pointer), rlink(right_pointer) { }
       //构造函数
       Type GetData( ) { return data; }  //获取结点数据域内容
};
```

程序2-16 循环双向链表的类定义

```
template <class Type> class DblList {  //循环双向链表类定义
    public:
        DblList( );  //构造函数
        ~DblList( );  //析构函数
        int Length( ) const;  //计算循环双向链表的长度
        int IsEmpty( ) { return head->rlink==head; }
        //判断循环双向链表是否为空表
        int Find(const Type &x);
        //在循环双向链表搜索其数据域为给定值的结点
        int Head( );  //定位于表头结点
        int Next( );  //定位于当前结点的后继结点
        int Prior( );  //定位于当前结点的前驱结点
        int Insert(const Type &x, const Type &y);
        //建立数据域值为y的新结点
        //并将其插入至数据域值为x的结点后面
        int Delete(const Type &x);  //删除数据域值为x的结点
    private:
        DblNode <Type> *head, *last, *current;  //表头指针
};
```

程序2-17 循环双向链表的插入与删除

```
template <class Type> int DblList <Type> ::
                    Insert(const Type &x, const Type &y)
{  //建立数据域值为y的新结点
   //并将其插入至数据域值为x的结点后面
     DblNode *p, *q;
     p=head->rlink;
     while (p!=head && p->data!=x)  //搜索数据域值为x的结点
          p=p->rlink;
     if (p==head)
          return 0;  //搜索不成功，插入失败，返回0
     q=new DblNode (y, NULL, NULL);
     //建立数据域值为y的新结点
     q->rlink=p->rlink;  //插入新结点在适当位置
                         //并修改相应指针
     p->rlink=q;
     q->rlink->llink=q;
     q->llink=p;
     return 1;  //插入成功，返回1
}
```

程序2-17 循环双向链表的插入与删除

```
template <class Type> int DblList <Type> ::
                                    Delete(const Type &x)
{ //删除数据域值为x的结点
    DblNode *p;
    p=head->rlink;
    while (p!=head &&p->data!=x)  //搜索数据域值为x的结点
        p=p->rlink;
    if (p==head)
        return 0;  //搜索不成功，删除失败，返回0
    p->llink->rlink=p->rlink;
    //删除数据域值为x的结点，并修改相应指针
    p->rlink->llink=p->llink;
    delete p;
    return 1;  //删除成功，返回1
}
```

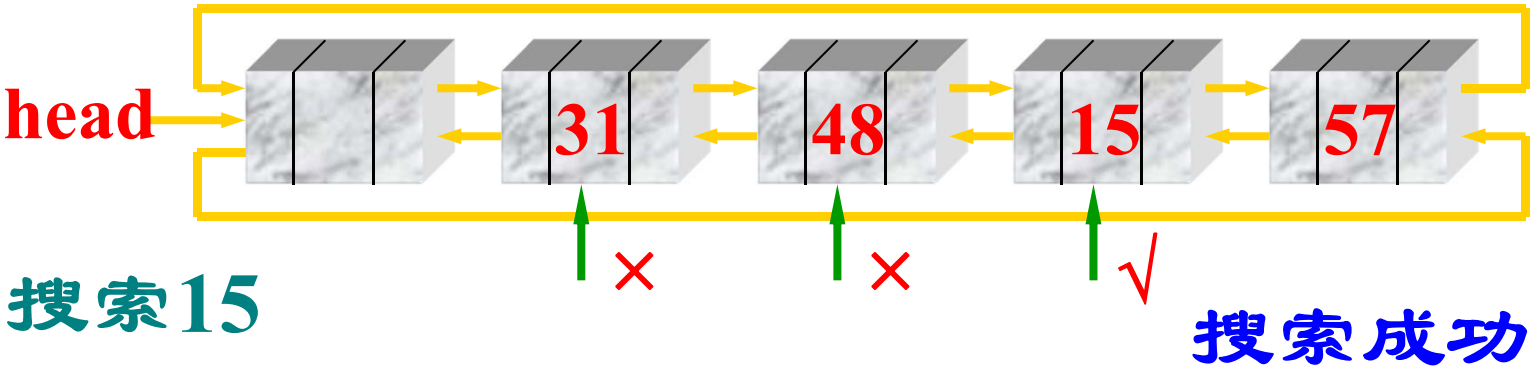
```
template <class Type>
DblList <Type> :: DblList(Type uniqueVal) {
//构造函数：建立双向循环链表的表头结点
   head=current=
         new DblNode <Type> ( uniqueVal);
   if (head==NULL)
      { cerr << "存储分配错！ \n" ; exit(1); }
   head->rLink=head->lLink=head;
}
```

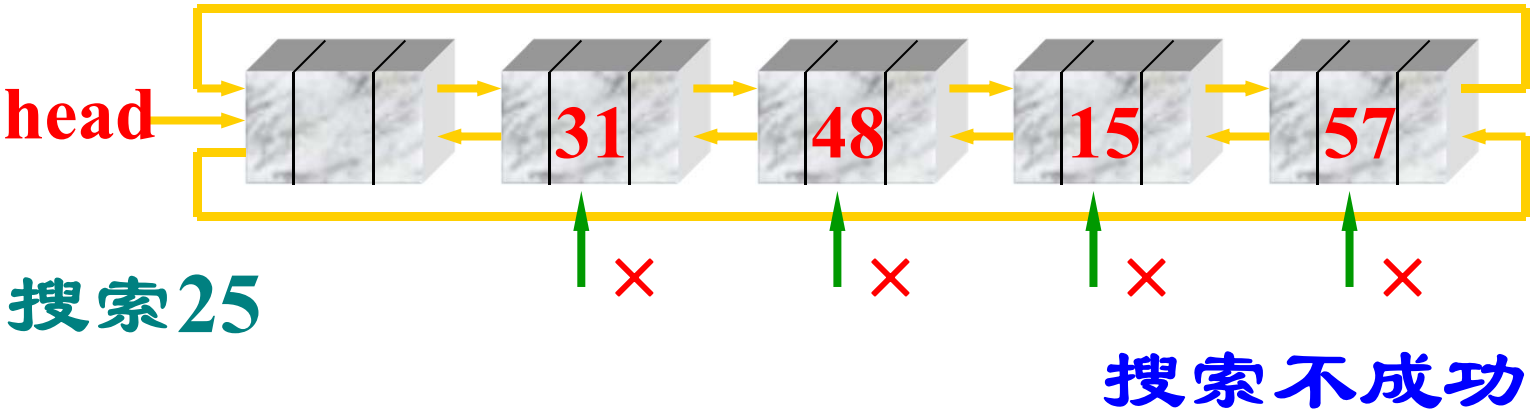
```
template <class Type>
DblList <Type> :: DblLIst(DblList <Type> &RL)
{ //复制构造函数：用已有链表RL初始化
   head=new DblNode <Type> (RL->head->data);
   if (head==NULL)
     { cerr<<"存储分配错！ \n" ; exit(1); }
   head->lLink=head->rLink=head;
   if (RL->current==RL->head)  current=head;
   ListNode <Type> *p=RL->head->rLink;
   ListNode <Type> *last=head;
```

```
    while (p!=RL->head)
    {  //逐个结点复制
       last->rLink=new DblNode <Type>
                                (p->data, last, last->rLink);
       if (last->rLink==NULL)  //分配新结点
         { cerr<<"存储分配错！ \n" ; exit(1); }
       last=last->rLink;
       last->rLink->lLink=last;  //链结完成
       if (RL->current==p)  current=last;
       p=p->rLink;
    }
}
```

```
template <class Type>
int DblList <Type> :: Length( ) const {
//计算带表头结点的双向循环链表的长度
    DblNode <Type> *p=head->rLink;
    int count=0;
    while (p!=head)
        { p=p->rLink;  count++; }
    return count;
}
//按前驱方向计算只需左、右互换即可
```

双向循环链表的搜索算法

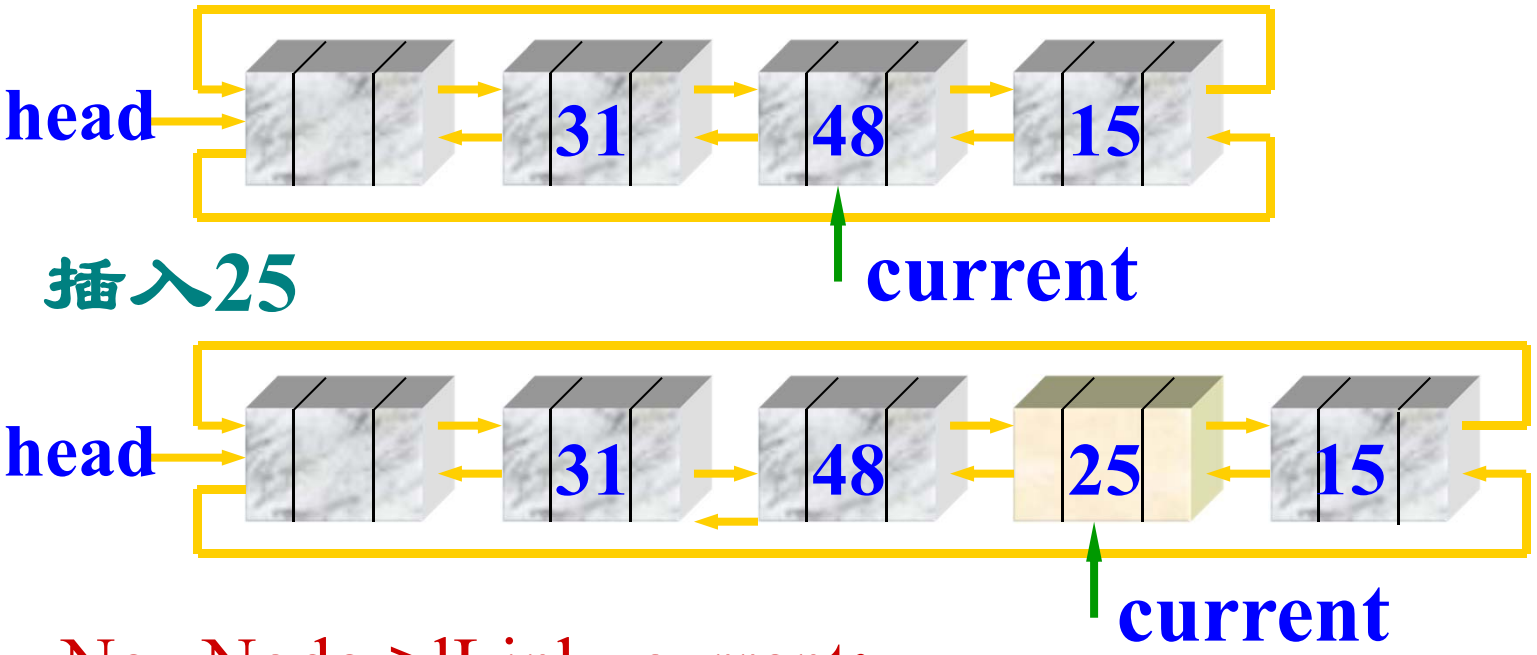




```
template <class Type> DblNode <Type>
        * DblList <Type> :: Search(const Type &x) {
//在双向循环链表中搜索含x的结点
    current=head->rLink;
    while (current!=head && current->data!=x)
        current=current->rLink;
    if (current!=head)  return current;
    else return NULL;
}
```

```
template <class Type> DblNode <Type>
        * DblList <Type> :: Locate(int i, int d) {
    if (head->rlink==head || i==0)  return head;
    if (d==0)  current=head->lLink;
    else current=head->rLink;
    for (int j=1; j<i; j++)
        if (current==head)  break;
        else if (d==0)  current=current->lLink;
         else current=current->rLink;
    if (current!=head)  return current;
    else return NULL;
}
```

双向循环链表的插入算法（非空表）




```
NewNode->lLink=current;
NewNode->rLink=current->rLink;
current->rLink=NewNode;
current=current->rLink;
current->rLink->lLink=current;
```

双向循环链表的插入算法（空表）

NewNode->lLink=current;

NewNode->rLink=current->rLink; (=head)

current->rLink=NewNode;

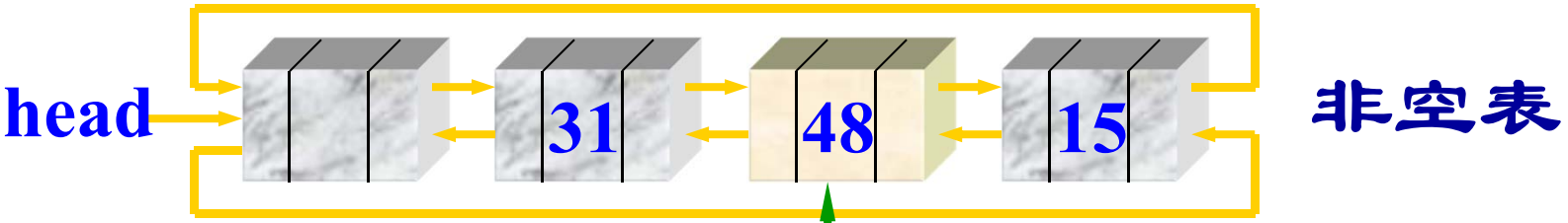
current=current->rLink;

current->rLink->lLink=current;

(head->lLink=current)

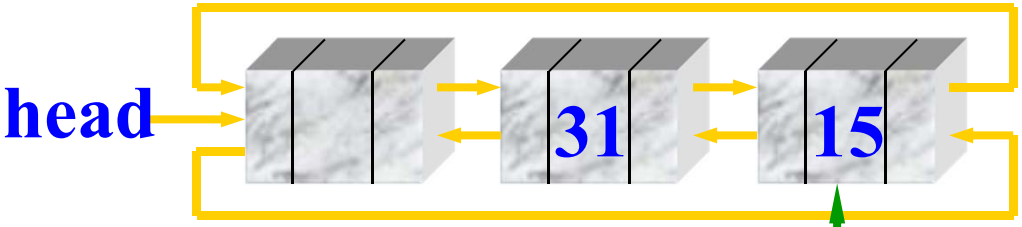
```
template <class Type> int DblList <Type> ::
                        Insert(int i, const Type &x, int d) {
    current=Locate(i, d);
    if (current==NULL)  return false;
    DblNode <Type> *NewNode=new DblNode <Type> (x);
    if (NewNode==NULL)
      { cerr<<"存储分配失败！ "<<endl; exit(1); }
    if (d==0) {
       NewNode->lLink=current->lLink;
       current->lLink=NewNode;
       NewNode->lLink->rLink=NewNode;
       NewNode->rLink=current;   }
    else {
       NewNode->rLink=current->rLink;
       current->rLink=NewNode;
       NewNode->rLink->lLink=NewNode;
       NewNode->lLink=current;   }
    return true;
}
```

双向循环链表的删除算法



删除48

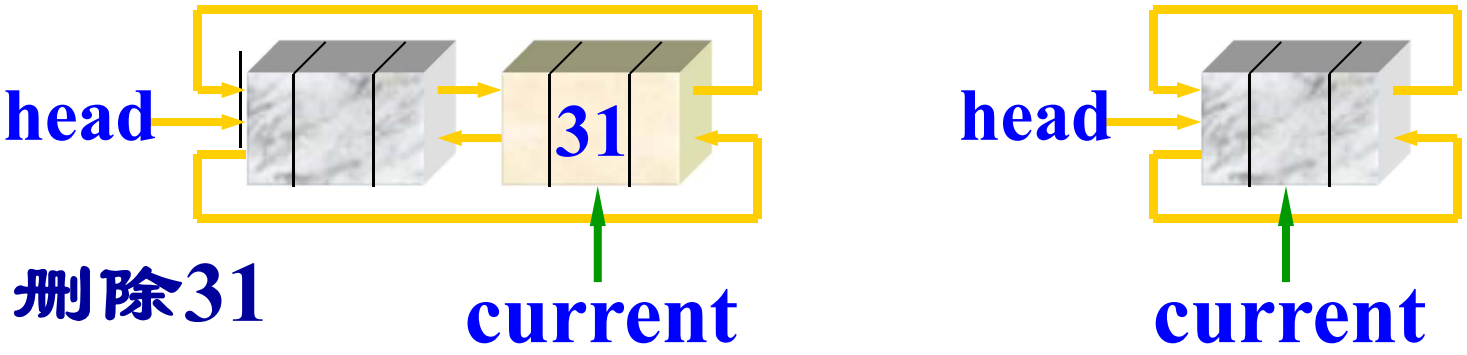
current



current

```
current->rLink->lLink=current->lLink;
current->lLink->rLink=current->rLink;
```

双向循环链表的删除算法



current->rLink->lLink=current->lLink;
current->lLink->rLink=current->rLink;

```
template <class Type> bool DblList <Type> ::
Remove(int i, Type &x, int d) {
    DblNode <Type> *current=Locate(i, d);
    if (current==NULL)  return false;
    current->rLink->lLink=current->lLink;  //摘下
    current->lLink->rLink=current->rLink;
    x=current->data;  delete current;
    return true;
}
```

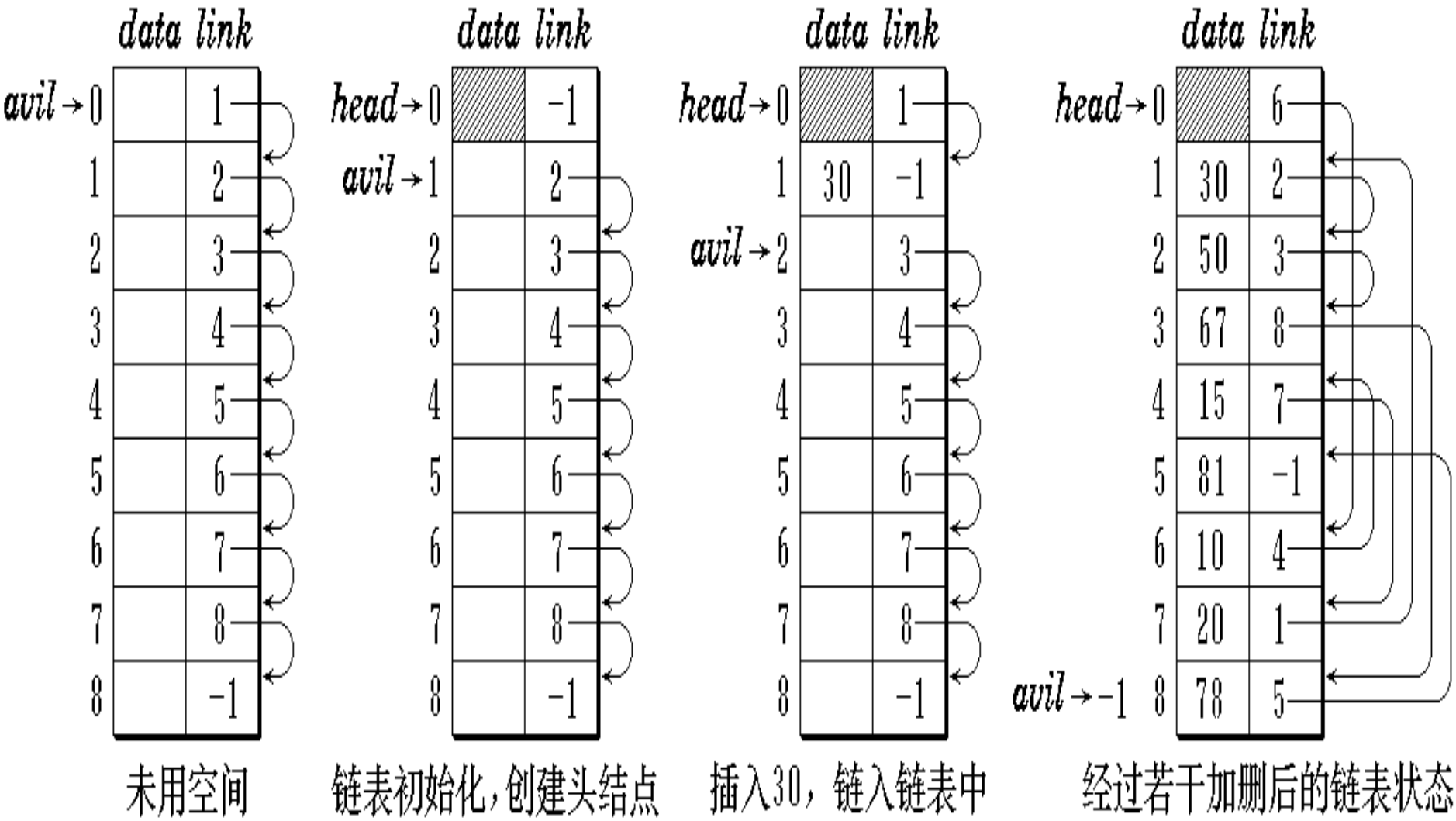
2.5 进阶导读

- 线性表是一种最简单且最常用的数据结构，也是软件设计中最基础的数据结构。
- 用顺序方法存储的线性表称为顺序表，当线性表中很少做插入和删除操作，线性表的长度变化不大，易于事先确定其大小时，可以采用顺序表作为存储结构。
- 用链接方法存储的线性表称为线性链表，当线性表的长度变化较大，难以估计其存储规模，另外对线性表频繁进行插入和删除操作时，则采用链表作为存储结构可能会更好一些。
- 除动态链表外，还有静态链表。
- 在C++的STL库中提供关于线性表的几种类库，如vector、list等。

静态链表

- 为数组中每一个元素附加一个链接指针。
- 允许不改变各元素的物理位置，只要重新链接就能够改变这些元素的逻辑顺序。
- 利用数组定义，在整个运算过程中存储空间的大小不会变化——静态链表。
  - 每个结点由两个数据成员构成；
    - **data**域存储数据
    - **link**域存放链接指针
  - 所有结点形成一个结点数组；
  - 可带有表头结点。

静态链表结构


data link
avil → 0
1 2 3 4 5 6 7 8
-1
未用空间
data link
head → 0
avil → 1
-1 2 3 4 5 6 7 8 -1
链表初始化，创建头结点
data link
head → 0
1 30 avil → 2 3 4 5 6 7 8
-1
插入30，链入链表中
data link
head → 0
1 2 3 4 5 6 7 avil → -1 8
6 30 50 67 15 81 10 20 78
2 3 8 7 -1 4 1 5
经过若干加删后的链表状态

- 需要分配一个结点时，从**avil**为头指针的链中摘下第一个结点分配出去，并用指针**j**指示；
- 头指针**avil**退到可利用空间表的第二个结点，即下一次可分配的结点地址。

  j=avil; avil=elem[avil].link;
- 需要释放由指针**i**指示的结点时，将其链入**avil**所指示的链中的最前端，并由**avil**指示。

  elem[i].link=avil; avil=i;
- 表头结点一般在第**0**个位置，其**link**指针指向链表中第一个结点。
- 最后一个结点的**link**指针为**-1**，表示链表终止。

静态链表类定义

```
const int maxSize=100;  //静态链表大小
template <class Type> class StaticList;
template <class Type> class SLinkNode {
    friend class StaticList <Type>;
    private:
        Type data;  //结点数据
        int link;  //结点链接指针
};
```

```
template <class Type> class StaticList {
        SLinkNode <Type> elem[maxSize];
        int avil;  //当前可分配空间首地址
    public:
        void InitList( );
        int Length( );
        int Search(Type x);
        int Locate(int i);
        bool Append(Type x);
        bool Insert(int i, Type x);
        bool Remove(int i);
        bool IsEmpty( );
};
```

将链表空间初始化

```
template <class Type>
void StaticList <Type> :: InitList( ) {
    elem[0].link=-1;
    avil=1;  //当前可分配空间从1开始
             //建立带表头结点的空链表
    for (int i=1; i<maxSize-1; i++)
        elem[i].link=i+1;  //构成空闲链接表
    elem[maxSize-1].link=-1;  //链表收尾
}
```

```
template <class Type>
int StaticList <Type> :: Length( ) {
//计算静态链表的长度
   int p=elem[0].link;  int i=0;
   while (p!=-1) {
        p=elem[p].link;  i++;   }
   return i;
}
template <class Type>
bool StaticList <Type> :: IsEmpty( ) {
//判断表空否
   if (elem[0].link==-1)  return true;
   else return false;
}
```

在静态链表中查找具有给定值的结点

```
template <class Type>
int StaticList <Type> :: Search(Type x) {
    int p=elem[0].link;
    //指针p指向链表第一个结点
    while (p!=-1)
        if (elem[p].data==x)  break;
        else p=elem[p].link;
    //逐个结点检测查找具有给定值的结点
    return p;
}
```

在静态链表中查找第i个结点

```
template <class Type>
int StaticList <Type> :: Locate (int i) {
    if (i<0)  return -1;  //参数不合理
    if (i==0)  return 0;
    int j=1, p=elem[0].link;
    while (p!= -1&&j<i)
        { p=elem[p].link;  j++; }
        //循链查找第i号结点
    return p;
}
```

在静态链表的表尾追加一个新结点

```
template <class Type>
bool StaticList <Type> :: Append(Type x) {
    if (avil==-1)  return false;  //追加失败
    int q=avil;  //分配结点
    avil=elem[avil].link;
    elem[q].data=x;   elem[q].link=-1;
    int p=0;  //查找表尾
    while (elem[p].link!=-1)  p=elem[p].link;
    elem[p].link=q;  //追加
    return true;
}
```

在静态链表第i个结点处插入新结点

```
template <class Type>
bool StaticList <Type> :: Insert(int i, Type x) {
    int p=Locate(i);
    if (p==-1)  return false;  //找不到结点
    int q=avil;  //分配结点
    avil=elem[avil].link;
    elem[q].data=x;
    elem[q].link=elem[p].link;  //链入
    elem[p].link=q;
    return true;
}
```

在静态链表中释放第i个结点

```
template <class Type>
bool StaticList <Type> :: Remove(int i) {
    int p=Locate(i-1);
    if (p==-1)  return false;  //找不到结点
    int q=elem[p].link;  //第i号结点
    elem[p].link=elem[q].link;
    elem[q].link=avil;  //释放
    avil=q;
    return true;
}
```

本章小结

- 知识点
  - 线性表
  - 数组及其压缩存储方式
  - 顺序表及其三种重要操作
  - 单链表、带表头结点的单链表
  - 循环链表、双向链表与静态链表
  - 多项式的应用

课程习题

- 笔做题——**2.2, 2.3, 2.10, 2.13, 2.15, 2.19**
  （以作业形式提交）
- 上机题——**2.16**
- 思考题——**2.1, 2.6, 2.8, 2.14, 2.17**

Any Suggestion or Question

联系方式：

- 张玥杰

Email: yjzhang@fudan.edu.cn