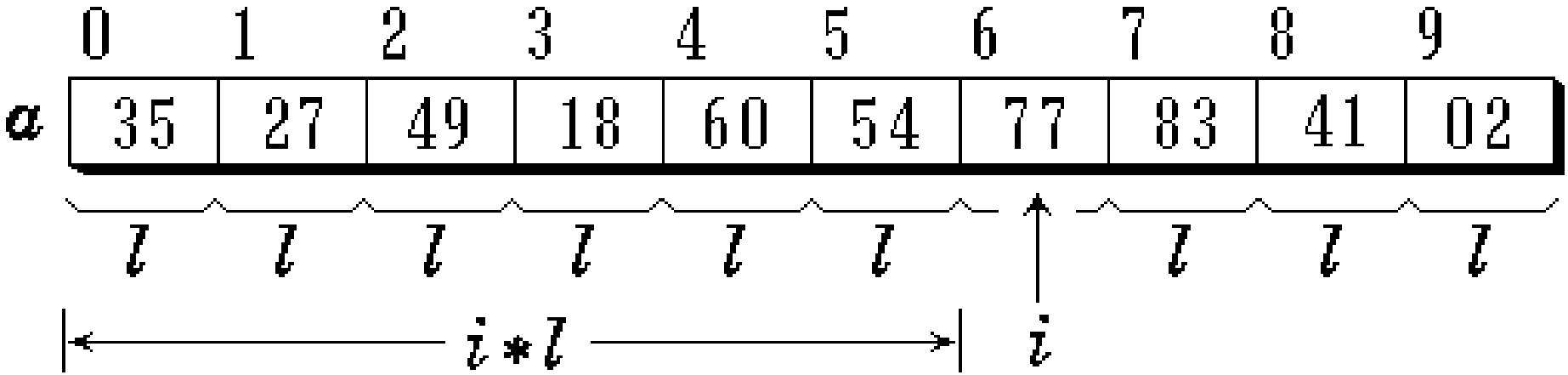
第二章　数组

- 作为抽象数据类型的数组
- 顺序表
- 多项式抽象数据类型
- 稀疏矩阵
- 字符串
- 小结

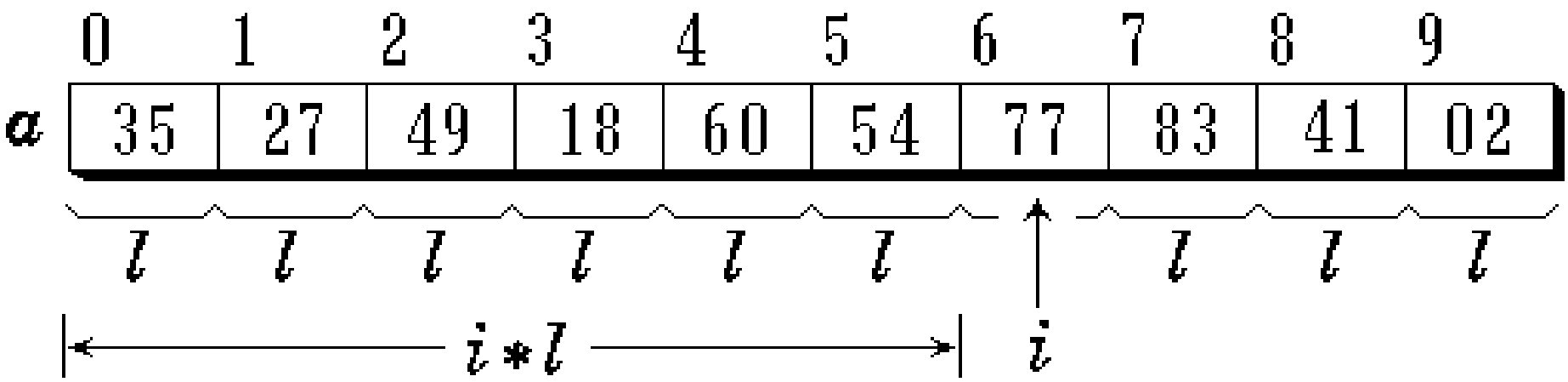
作为抽象数据类型的数组

- 一维数组
  - 一维数组的示例



一维数组的特点

- 连续存储的线性聚集（别名 向量）
- 除第一个元素外，其他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，其他每一个元素有一个且仅有一个直接后继。



数组的定义和初始化

```
#include <iostream.h>
 class szcl {
      int e;
 public:
      szcl ( ) { e = 0; }
      szcl ( int value ) { e = value; }
      int get_value ( ) { return e; }
 }
```

```
main ( ) {
   szcl a1[3] = { 3, 5, 7 }, *elem;
   for ( int i = 0; i < 3; i++ )
       cout << a1[i].get_value ( ) << "\n";
         //打印静态数组
   elem = a1;
   for ( int i = 0; i < 3; i++ ) {
       cout << elem→get_value( ) << "\n";
         //打印动态数组
       elem++;
    }
    return 0;
}
```

一维数组(Array)类的定义

```
#include <iostream.h>
#include <stdlib.h>
template <class Type>
class Array {
    Type *elements;        //数组存放空间
    int ArraySize;         //当前长度
    void getArray ( );     //建立数组空间
public:
    Array( int Size=DefaultSize );
    Array( const Array<Type>& x );
```

```
    ~Array( ) { delete [ ]elements;}
    Array<Type> & operator =   //数组复制
            ( const Array<Type> & A );
    Type& operator [ ] ( int i );   //取元素值
    Type* operator ( ) const        //指针转换
            { return elements; }
    int Length ( ) const                //取数组长度
            { return ArraySize; }
    void ReSize ( int sz );             //扩充数组
}
```

一维数组公共操作的实现

```
template <class Type>
void Array<Type>::getArray ( ) {
//私有函数：创建数组存储空间
    elements = new Type[ArraySize];
    if ( elements == 0 ) {
       arraySize = 0;
       cerr << "Memory Allocation Error" << endl;
       return;
}
```

```
template <class Type>
Array<Type>::Array ( int sz ) {
//构造函数
    if ( sz <= 0 ) {
        arraySize = 0;
        cerr << “非法数组大小” << endl;
        return;
    }
    ArraySize = sz;
    getArray ( );
}
```

```
template <class Type>  Array<Type>::
Array ( const Array<Type> & x ) {
//复制构造函数
    int n = ArraySize = x.ArraySize;
    elements = new Type[n];
    if ( elements == 0 ) {
        arraySize = 0;
        cerr << "存储分配错" << endl;
        return;
    }
    Type *srcptr = x.elements;
    Type *destptr = elements;
    while ( n-- ) * destptr++ = * srcptr++;
}
```

```
template <class Type>
Type & Array<Type>::operator [ ] ( int i ) {
 //按数组名及下标 i，取数组元素的值
    if ( i < 0 || i > ArraySize −1 ) {
        cerr << “数组下标超界” << endl;
        return NULL;
    return element[i];
 }
```

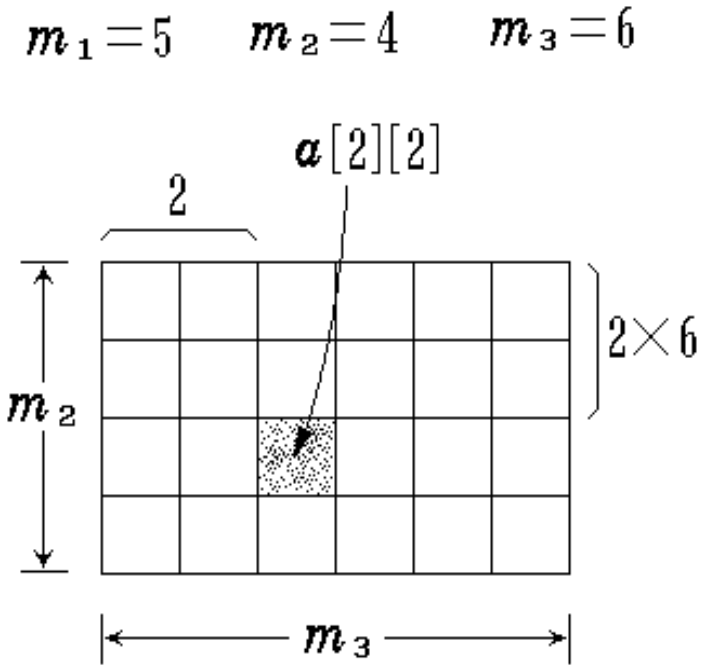
使用该函数于赋值语句

$Pos = Position[i-1] + Number[i-1]$

```
template <class Type>
void Array<Type>::Resize (int sz) {
  if ( sz >= 0 && sz != ArraySize ) {
    Type * newarray = new Type[sz];
    if ( newarray == 0 ) {
      cerr << "存储分配错" << endl;
      return;
    }
    int n = ( sz <= ArraySize ) ? sz : ArraySize;
    Type *srcptr = elements;
    Type *destptr = newarray;
    while ( n-- ) * destptr++ = * srcptr++;
```

```
        delete [ ] elements;
        elements = newarray;
        ArraySize = sz;
    }
}
```

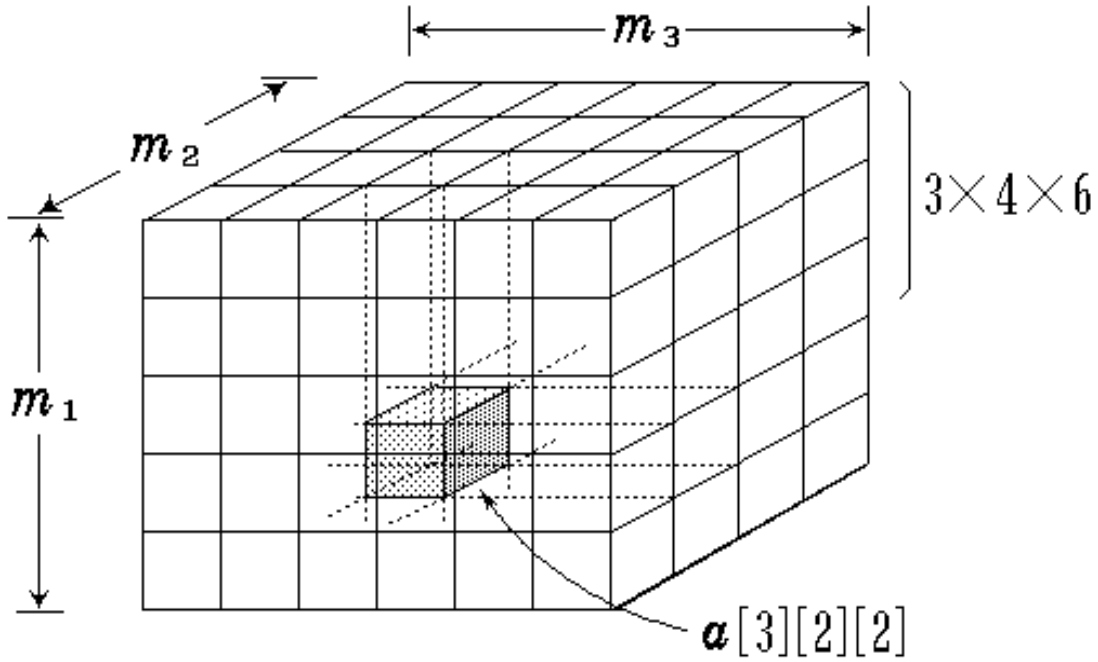
二维数组



行向量 下标 *i*

列向量 下标 *j*

三维数组



页向量 下标 *i*

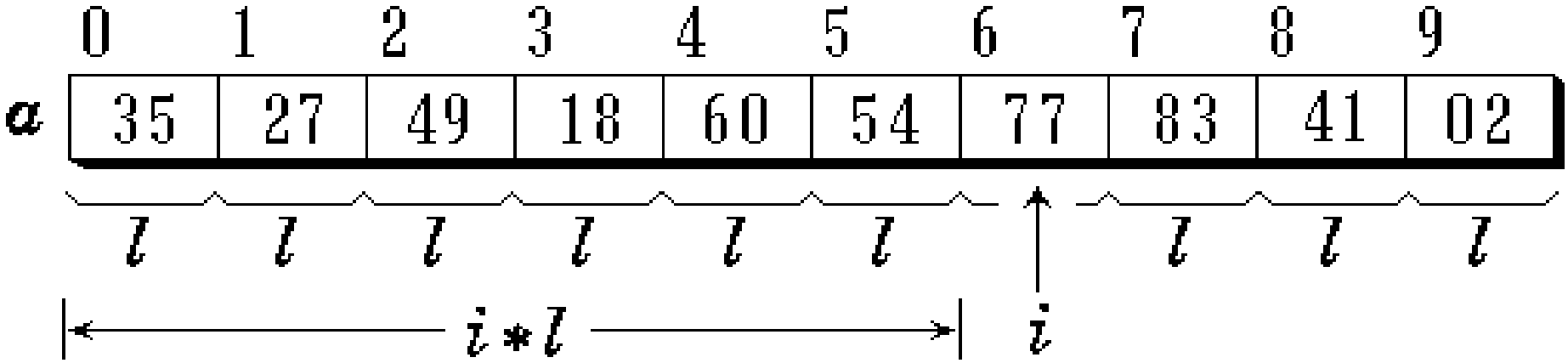
行向量 下标 *j*

列向量 下标 *k*

数组的连续存储方式

- 一维数组

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



$$LOC(i)=LOC(i-1)+l=\alpha+i*l$$

- 二维数组

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

行优先 $LOC(j,k) =$

$$= a + (j * m + k) * l$$

n维数组

- 各维元素个数为 $m_1, m_2, m_3, \ldots, m_n$
- 下标为 $i_1, i_2, i_3, \ldots, i_n$ 的数组元素的存储地址：

$$LOC(i_1, i_2, \ldots, i_n) = a + (i_1*m_2*m_3*\ldots*m_n + i_2*m_3*m_4*\ldots*m_n + \ldots\ldots + i_{n-1}*m_n + i_n) * l$$

$$= a + \left(\sum_{j=1}^{n-1} i_j * \prod_{k=j+1}^{n} m_k + i_n \right) * l$$

顺序表 (*Sequential List*)

- 顺序表的定义和特点
  - 定义 n（≥ 0）个表项的有限序列

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

    a_i是表项，n是表长度。
  - 特点　顺序存取
  - 遍历　逐项访问

    从前向后 从后向前 从两端向中间

顺序表(*SeqList*)类的定义

```
template <class Type>      class SeqList {
    Type *data;            //顺序表存储数组
    int MaxSize;           //最大允许长度
    int last;              //当前最后元素下标
public:
    SeqList ( int MaxSize = defaultSize );
    ~SeqList ( ) { delete [ ] data; }
    int Length ( ) const { return last+1; }
    int Find ( Type & x ) const;
```

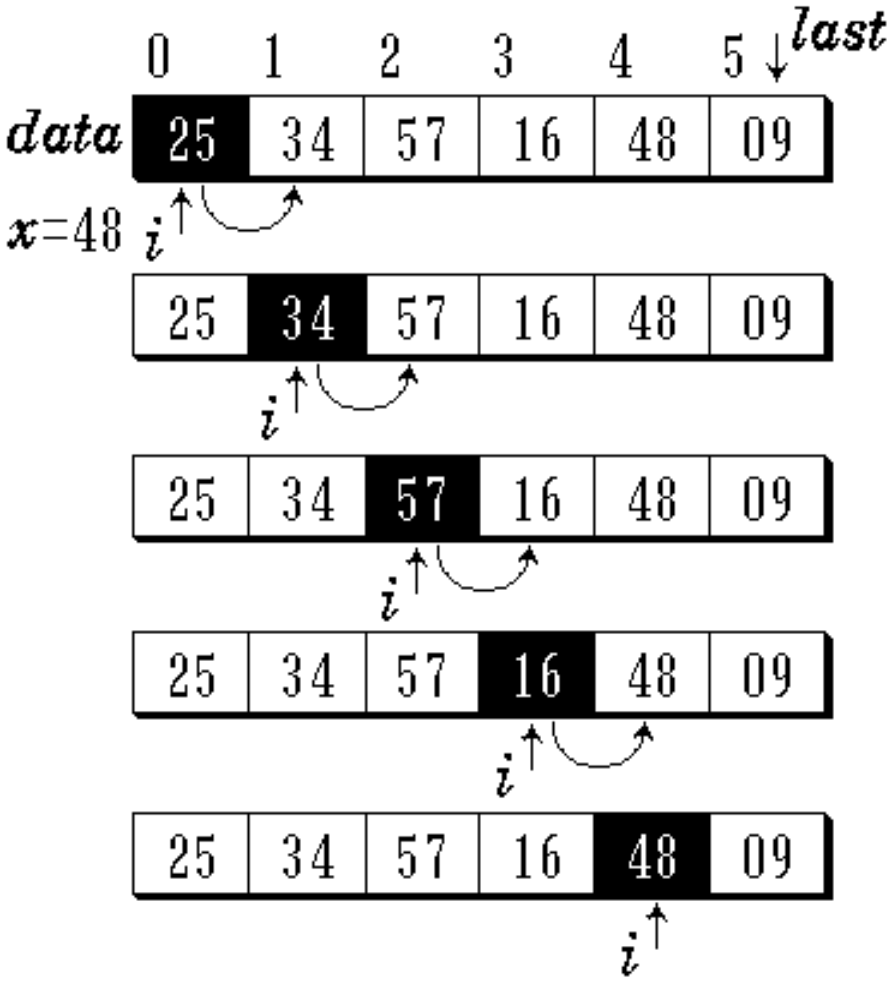
```
    int IsIn ( Type & x );
    int Insert ( Type & x, int i );
    int Remove ( Type & x );
    int Next ( Type & x ) ;
    int Prior ( Type & x ) ;
    int IsEmpty ( ) { return last == -1; }
    int IsFull ( ) { return last == MaxSize-1; }
    Type Get ( int i ) {
        return i < 0 || i > last? NULL : data[i];
    }
}
```

顺序表部分公共操作的实现

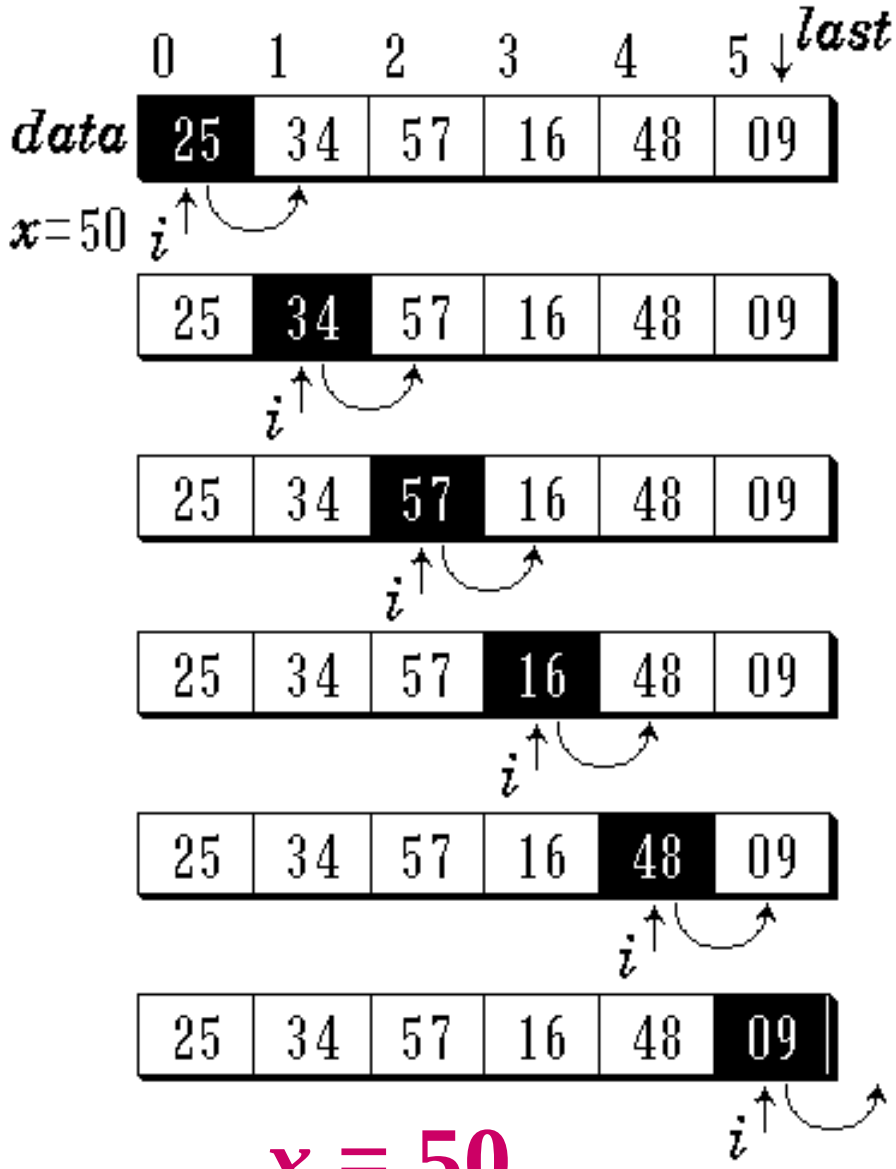
```
template <class Type>
SeqList<Type> :: SeqList ( int sz ) {    //构造函数
    if ( sz > 0 ) {
        MaxSize = sz;  last = −1;
        data = new Type[MaxSize];
        if ( data == NULL ) {
            MaxSize = 0;   last = −1;
            return;
        }
    }
}
```

```
template <class Type>
int SeqList<Type>::Find ( Type & x ) const {
 //搜索函数：在表中从前向后顺序查找 x
    int  i = 0;
    while ( i <= last && data[i] != x )
        i++;
    if ( i > last ) return -1;
    else return i;
}
```

顺序搜索图示



$x = 48$



$x = 50$

搜索成功

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

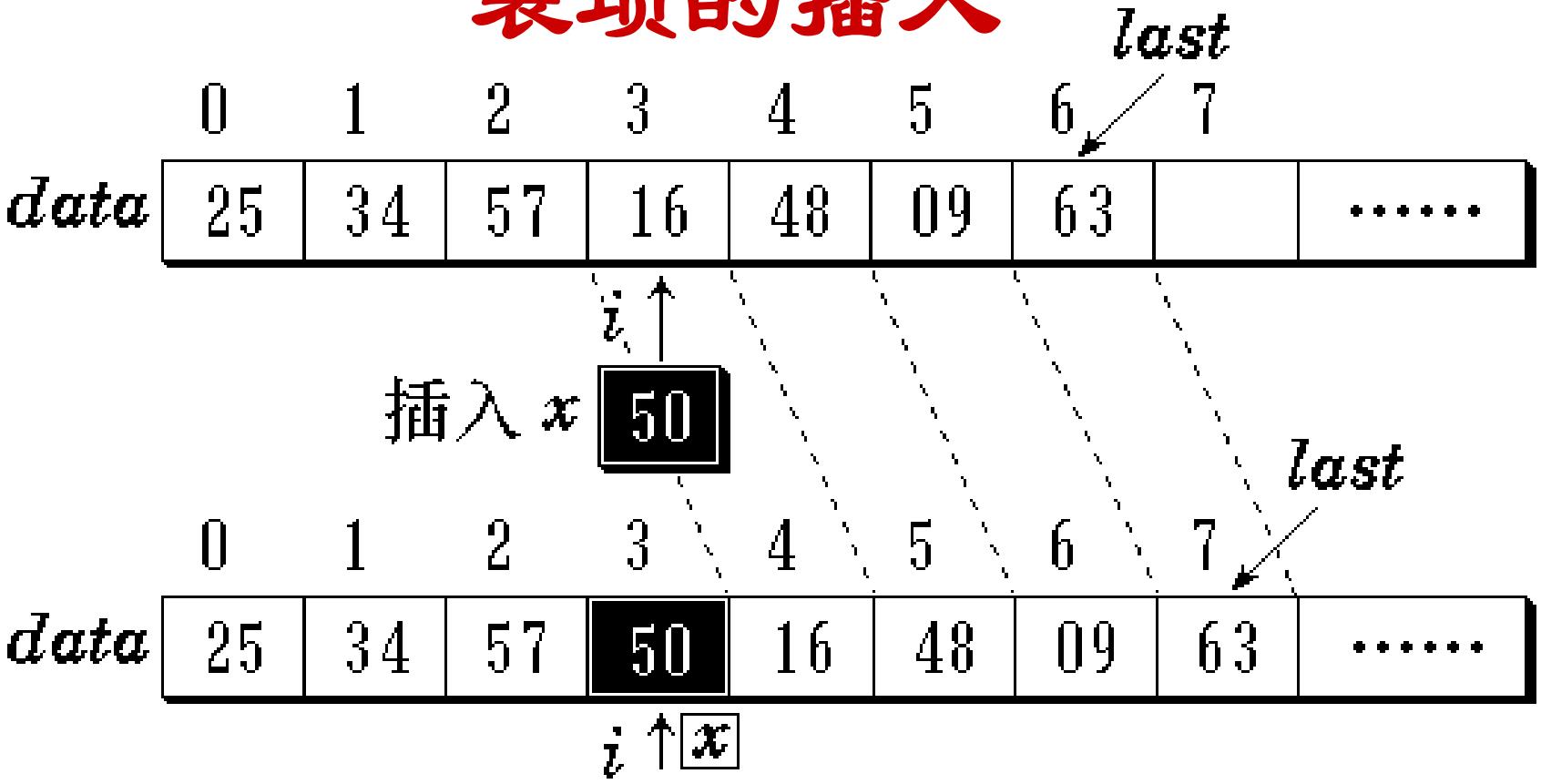
若搜索概率相等，则

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

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

搜索不成功　数据比较 n 次

表项的插入

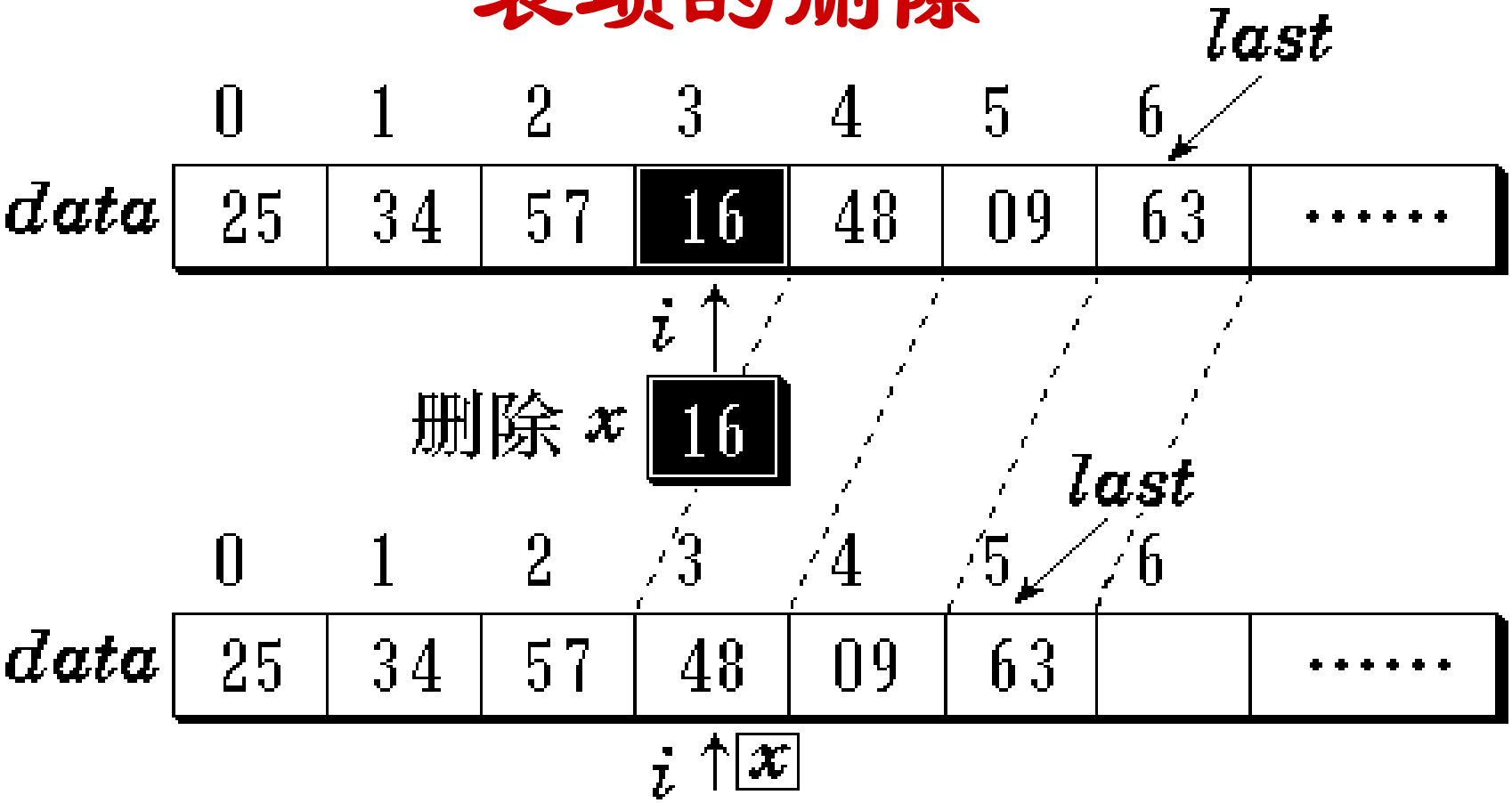


$$AMN = \frac{1}{n+1}\sum_{i=0}^{n}(n-i) = \frac{1}{n+1}(n+\cdots+1+0)$$

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

```
template <class Type>
int SeqList<Type>::Insert ( Type & x, int i ){
//在表中第 i 个位置插入新元素 x
    if ( i < 0 || i > last+1 || last == MaxSize−1 )
         return 0;                    //插入不成功
    else {
        last++;
        for ( int j = last; j > i; j−− )
                        data[j] = data[j −1];
        data[i] = x;
        return 1;                     //插入成功
    }
}
```

表项的删除



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

```
template <class Type>
int SeqList<Type>::Remove ( Type & x ) {
//在表中删除已有元素 x
    int i = Find (x);          //在表中搜索 x
    if ( i >= 0 ) {
        last--;
        for ( int j = i; j <= last; j++ )
            data[j] = data[j+1];
        return 1;              //成功删除
    }
    return 0;                  //表中没有 x
}
```

顺序表的应用：集合的“并”运算

```
template <class Type>
void Union ( SeqList<Type> & LA,
                        SeqList<Type> & LB ) {
    int n = LA.Length ( );
    int m = LB.Length ( );
    for ( int i = 1; i <= m; i++ ) {
        Type x = LB.Get(i);      //在LB中取一元素
        int k = LA.Find (x);     //在LA中搜索它
        if ( k == −1 )           //若未找到插入它
          { LA.Insert (n+1, x);  n++; }
    }
}
```

顺序表的应用：集合的“交”运算

```
template <class Type>
void Intersection ( SeqList<Type> & LA,
                    SeqList<Type> & LB ) {
    int n = LA.Length ( );
    int m = LB.Length ( );  int i = 0;
    while ( i < n ) {
        Type x = LA.Get (i);      //在LA中取一元素
        int k = LB.Find (x);      //在LB中搜索它
        if ( k == −1 ) { LA.Remove (i);  n−−; }
        else i++;                 //未找到在LA中删除它
    }
}
```

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
}
```

创建*power*类，计算*x*的幂

```
#include <iostream.h>
class power {
    double x;
    int e;
    double mul;
public:
    power (double val, int exp);            //构造函数
    double get_power ( ) { return mul; }  //取幂值
};
```

```
power::power (double val, int exp) {
//按val值计算乘幂
    x = val;   e = exp;   mul = 1.0;
    if ( exp == 0 ) return;
    for ( ; exp>0; exp--) mul = mul * x;
}

main ( ) {
    power pwr ( 1.5, 2 );
    cout << pwr.get_power ( ) << "\n";
}
```

多项式的存储表示

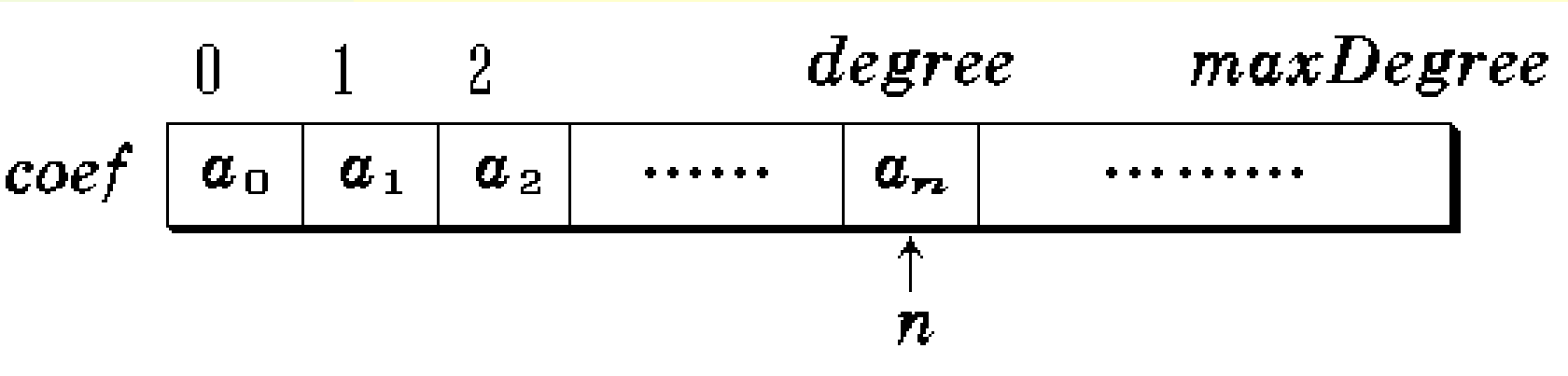
第一种：(静态数组表示)

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

$P_n(x)$可以表示为：

$pl.degree = n$

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



第二种：(动态数组表示)

```
private:
    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
};
```

	0	1	2	·····	i	·····	m
coef	a_0	a_1	a_2	·····	a_i	·····	a_m
exp	e_0	e_1	e_2	·····	e_i	·····	e_m

```
class Polynomial {          //多项式定义
public:
    ……
private:
    static term termArray[MaxTerms];    //项数组
    static int free;                    //当前空闲位置指针
    // term Polynomial::termArray[MaxTerms];
    // int Polynomial::free = 0;
    int start, finish;                  //多项式始末位置
}
```

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

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

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。

```
Polynomial Polynomial::operator+(Polynomial B) {
  Polynomial C;
  int a = start; int b = B.start;  C.start = free;
  float c;
  while ( a <= finish && b <= B.finish )
     Switch ( compare ( termArray[a].exp,
       termArray[b].exp) ) {          //比较对应项指数
     case '=' :                       //指数相等
       c = termArray[a].coef +        //系数相加
                          termArray[b].coef;
       if ( c ) NewTerm ( c, termArray[a].exp );
       a++;  b++;  break;
```

```
case '>' :                          //b指数小,建立新项
    NewTerm ( termArray[b].coef,
                        termArray[b].exp );
    b++;  break;
case '<':                           //a指数小,建立新项
    NewTerm ( termArray[a].coef,
                        termArray[a].exp );
    a++;
}
for (  ; a <= finish; a++ )      //a未检测完时
    NewTerm ( termArray[a].coef,
                        termArray[a].exp );
```

```
    for ( ; b <= B.finish; b++ )        //b未检测完时
        NewTerm ( termArray[b].coef,
                                termArray[b].exp );
    C.finish = free−1;
    return C;
}
```

在多项式中加入新的项

```
void Polynomial::NewTerm ( float c, int e ) {
//把一个新的项加到多项式C(x)中
   if ( free >= maxTerms ) {
      cout << "Too many terms in polynomials"
           << endl;
      return;
   }
   termArray[free].coef = c;
   termArray[free].exp = e;
   free++;
}
```

稀疏矩阵 (Sparse Matrix)

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

行数$m=6$，列数$n=7$，非零元素个数$t=6$

稀疏矩阵(*SparseMatrix*)的抽象数据类型

```
template <class Type>
class SparseMatrix {
    int Rows, Cols, Terms;    //行/列/非零元素数
    Trituple<Type> smArray[MaxTerms];
                              //三元组表
public:
    SparseMatrix ( int MaxRow, int Maxcol );
    SparseMatrix<Type> Transpose ( );   //转置
    SparseMatrix<Type>      //相加
          Add ( SparseMatrix<Type> b );
    SparseMatrix<Type>      //相乘
          Multiply ( SparseMatrix<Type> b );
}
```

三元组 (*Trituple*) 类的定义

```
template<class Type> class SparseMatrix<Type>;

template<class Type> class Trituple {
friend class SparseMatrix <Type>
private:
    int row, col;       //非零元素所在行号/列号
    Type value;         //非零元素的值
}
```

row	col	value

稀疏矩阵

$$A_{6\times 7}=\begin{pmatrix} 0 & 0 & 0 & 22 & 0 & 0 & 15 \\ 0 & 11 & 0 & 0 & 0 & 17 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 39 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 & 0 \end{pmatrix}$$

转置矩阵

$$B_{7\times 6}=\begin{pmatrix} 0 & 0 & 0 & 0 & 91 & 0 \\ 0 & 11 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 28 \\ 22 & 0 & -6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 17 & 0 & 39 & 0 & 0 \\ 15 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

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。
- 设矩阵三元组表总共有 *Terms* 项，其时间代价为 O (*Cols* Terms*)。
- 若矩阵有 200 行，200 列，10,000 个非零元素，总共有 2,000,000 次处理。

稀疏矩阵的转置

```
template <class Type>
SparseMatrix<Type>
SparseMatrix<Type>:: Transpose ( ) {
    SparseMatrix<Type> b ( Cols, Rows );
    b.Rows = Cols;   b.Cols = Rows;
    b.Terms = Terms;
    //转置矩阵的列数,行数和非零元素个数
    if ( Terms > 0 ) {
        int CurrentB = 0;
        //转置三元组表存放指针
```

```
        for ( int k = 0; k < Cols; k++ )
            for ( int i = 0; i < Terms; i++ )
                if ( smArray[i].col == k ) {
                    b.smArray[CurrentB].row = k;
                    b.smArray[CurrentB].col =
                                    smArray[i].row;
                    b.smArray[CurrentB].value=
                                    smArray[i].value;
                    CurrentB++;
                }
    }
    return b;
}
```

快速转置算法

- 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，根据某项的列号，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。
- 转置时间代价为 O(*max*(*Terms*, *Cols*))。若矩阵有 200 列，10000 个非零元素，总共需要 10000 次处理。

	[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

```
for ( int i = 0; i < Cols; i++ ) rowSize[i] = 0;
for ( i = 0; i < Terms; i++ )
    rowSize[smArray[i].col]++;
rowStart[0] = 0;
for ( i = 1; i < Cols; i++ )
    rowStart[i] = rowStart[i-1]+rowSize[i-1];
```

稀疏矩阵的快速转置

```
template <class Type> SparseMatrix<Type>
SparseMatrix<Type>::FastTranspos ( ) {
    int *rowSize = new int[Cols];
    int *rowStart = new int[Cols];
    SparseMatrix<Type> b ( Cols, Rows );
    b.Rows = Cols;   b.Cols = Rows;
    b.Terms = Terms;
    if ( Terms > 0 ) {
        for (int i = 0; i < Cols; i++) rowSize[i] = 0;
    for ( i = 0; i < Terms; i++ )
        rowSize[smArray[i].col]++;
```

```
        rowStart[0] = 0;
        for ( i = 1; i < Cols; i++ )
            rowStart[i] = rowStart[i−1]+rowSize[i−1];
        for ( i = 0; i < Terms; i++ ) {
            int j = rowStart[smArray[i].col];
            b.smArray[j].row = smArray[i].col;
            b.smArray[j].col = smArray[i].row;
            b.smArray[j].value = smArray[i].value;
            rowStart[smArray[i].col]++;
        }
    }
    delete [ ] rowSize;    delete [ ] rowStart;
    return b;
}
```

字符串 (*String*)

字符串是 n (≥ 0) 个字符的有限序列，

记作 S : “$c_1c_2c_3\ldots c_n$”

其中，S 是串名字

“$c_1c_2c_3\ldots c_n$”是串值

c_i 是串中字符

n 是串的长度。

字符串抽象数据类型和类定义

```
const int maxLen = 128;
class String {
    int curLen;          //串的当前长度
    char *ch;            //串的存储数组
public:
    String ( const String & ob );
    String ( const char *init );
    String ( );
    ~String ( ) { delete [ ] ch; }
    int Length ( ) const { return curLen; }
```

```
    String &operator ( ) ( int pos, int len );
    int operator == ( const String &ob )
        const { return strcmp (ch, ob.ch) == 0; }
    int operator != ( const String &ob )
        const { return strcmp (ch, ob.ch) != 0; }
    int operator ! ( )
        const { return curLen == 0; }
    String &operator = ( const String &ob );
    String &operator += ( const String &ob );
    char &operator [ ] ( int i );
    int Find ( String& pat ) const;
}
```

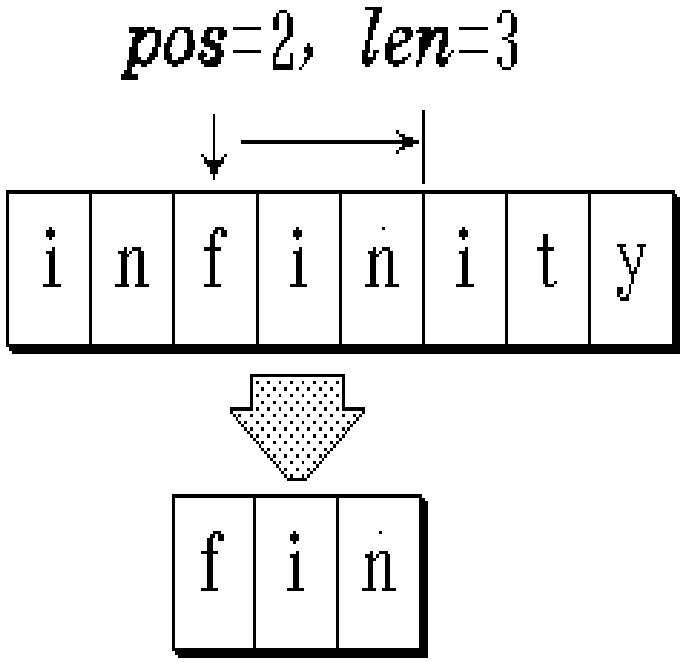
字符串部分操作的实现

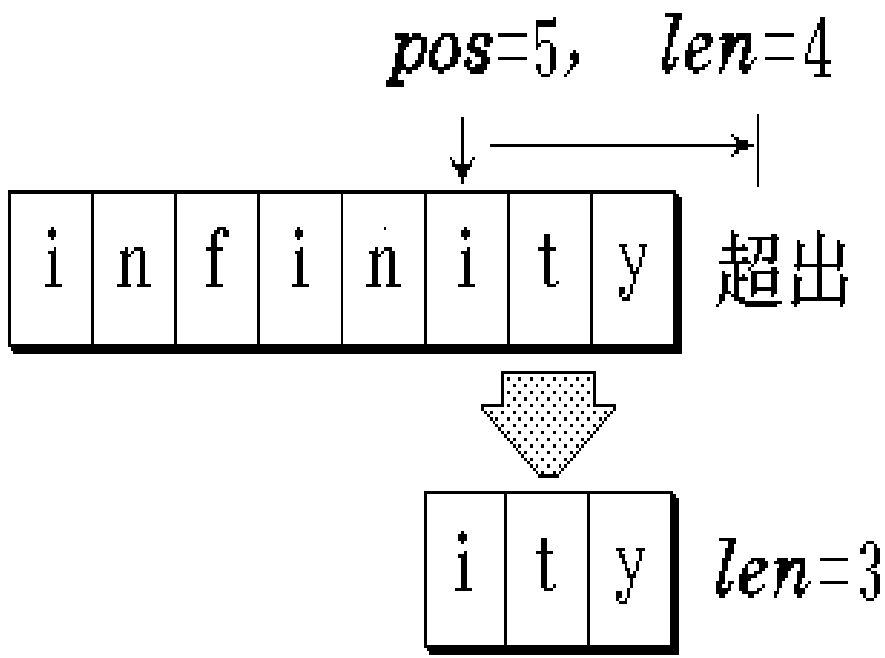
```
String::String ( const String &ob ) {
//复制构造函数：从已有串ob复制
    ch = new char[maxLen+1];
    if ( !ch ) {
        cerr << "存储分配错 \n";
        exit(1);
    }
    curLen = ob.curLen;
    strcpy ( ch, ob.ch );
}
```

```
String::String ( const char *init ) {
//复制构造函数: 从已有字符数组*init复制
    ch = new char[maxLen+1];
    if ( !ch ){
        cerr << "存储分配错 \n";
        exit(1);
    }
    curLen = strlen ( init );
    strcpy ( ch, init );
}
```

```
String::String ( ) {
//构造函数: 创建一个空串
    ch = new char[maxLen+1];
    if ( !ch ) {
        cerr << "存储分配错\n";
        exit(1);
    }
    curLen = 0;
    ch[0] = '\0';
}
```

提取子串的算法示例





$pos+len-1 \leq curLen-1$

$pos+len-1 \geq curLen$

```
String &String:: operator ( ) ( int pos, int len ) {
//从串中第 pos 个位置起连续提取 len 个字符形
//成子串返回
    String *temp = new String;   //动态分配
    if ( pos < 0 || pos+len−1 >= maxLen || len < 0 ) {
        temp→curLen = 0;          //返回空串
        temp→ch[0] = '\0';
    }
    else {                                  //提取子串
        if ( pos+len −1 >= curLen )
            len = curLen −pos;
        temp→curLen = len;        //子串长度
```

```
        for ( int i = 0, j = pos; i < len; i++, j++ )
            temp→ch[i] = ch[j];      //传送串数组
        temp→ch[len] = '\0';         //子串结束
    }
    return *temp;
}
```

例：串 ***st*** = **“university”，*pos* = 3，*len* = 4**

使用示例　***subSt* = *st* (3, 4)**

提取子串　***subSt* = “vers”**

```
String &String::operator = ( const String &ob ) {
//串赋值：从已有串ob复制
    if ( &ob != this ) {
        delete [ ] ch;
        ch = new char [maxLen+1];  //重新分配
        if ( ! ch ) {
            cerr << "内存不足!\n ";  exit (1);
         }
         curLen = ob.curLen;              //串复制
         strcpy ( ch, ob.ch );
    }
    else  cout << "字符串自身赋值出错! \n";
    return *this;
}
```

```
char &String::operator [ ] ( int i ) {
//按串名提取串中第i个字符
   if ( i < 0 && i >= curLen ) {
      cout << "字符串下标超界!\n "; exit (1) ;
   }
   return ch[i];
}
```

例： 串 ***st*** = **“university”,**

使用示例　***newSt = st;　newChar = st[1];***

数组赋值　***newSt* = “university”**

提取字符　***newChar* = ‘n’**

```
String &String::operator += ( const String &ob ) {
//串连接
    char * temp =ch;              //暂存原串数组
    curLen += ob.curLen;          //串长度累加
    ch = new char [maxLen+1];
    if ( ! ch ) {
        cerr << "字符串下标超界!\n "; exit (1) ;
    }
    strcpy ( ch, temp );          //拷贝原串数组
    strcat ( ch, ob.ch );         //连接ob串数组
    delete [ ] temp; return *this;
}
```

例： 串 *st1* = “beijing ”,

st2 = “university”,

使用示例 *st1* += *st2*;

连接结果 *st1* = “beijing university”

st2 = “university”

串的模式匹配

- 定义 在串中寻找子串（第一个字符）在串中的位置
- 词汇 在模式匹配中，子串称为模式，串称为目标。
- 示例 目标 T：“Beijing”

  模式 P：“jin”

  匹配结果 = 3

穷举的模式匹配过程

第1趟

T a b b a b a

P a b a

第2趟

T a b b a b a

P a b a

第3趟

T a b b a b a

P a b a

第4趟

T a b b a b a

P a b a

√

```
int String::Find ( String &pat ) const {
//穷举的模式匹配
    char * p = pat.ch,  * s = ch;    int i = 0;
    if ( *p && *s )                  //当两串未检测完
        while ( i <= curLen - pat.curLen )
            if ( *p++ == *s++ )      //比较串字符
                if ( !*p ) return i;     //相等
            else { i++;  s = ch + i;  p = pat.ch; }
                //对应字符不相等，对齐目标的
                //下一位置，继续比较
        return -1;
    }
```

改进的模式匹配

目标 T　t_0 t_1 t_2 …… t_{m-1} … t_{n-1}

↕ ↕ ↕ ↕

模式 pat　p_0 p_1 p_2 …… p_{m-1}

目标 T　t_0 t_1 t_2 …… t_{m-1} t_m … t_{n-1}

↕ ↕ ↕ ↕

模式 pat　p_0 p_1 …… p_{m-2} p_{m-1}

目标 T　t_0 t_1 …… t_i t_{i+1}…… t_{i+m-2} t_{i+m-1}… t_{n-1}

‖ ‖ ‖ ‖

模式 pat　p_0 p_1 …… p_{m-2} p_{m-1}

- 穷举的模式匹配算法时间代价:
  最坏情况比较$n-m+1$趟，每趟比较m次，总比较次数达$(n-m+1)*m$
- 原因在于每趟重新比较时，目标串的检测指针要回退。改进的模式匹配算法可使目标串的检测指针每趟不回退。
- 改进的模式匹配(KMP)算法的时间代价:
  - 若每趟第一个不匹配，比较$n-m+1$趟，总比较次数最坏达$(n-m)+m = n$
  - 若每趟第m个不匹配，总比较次数最坏亦达到 n

$T \quad t_0 \; t_1 \; \cdots \; t_{s-1} \; t_s \; t_{s+1} \; t_{s+2} \; \cdots \; t_{s+j-1} \; t_{s+j} \; t_{s+j+1} \; \cdots \; t_{n-1}$

$\qquad\qquad\qquad\quad \| \quad \| \quad \| \qquad\quad \| \quad\quad \| \quad\quad \times$

$P \qquad\qquad\qquad\quad p_0 \; p_1 \; p_2 \; \cdots \; p_{j-1} \; p_j \; p_{j+1}$

则有 $\quad t_s \, t_{s+1} \, t_{s+2} \cdots t_{s+j} = p_0 \, p_1 \, p_2 \cdots p_j \qquad (1)$

为使模式 *P* 与目标 *T* 匹配，必须满足

$$p_0 \, p_1 \, p_2 \cdots p_{j-1} \cdots p_{m-1} = t_{s+1} \, t_{s+2} \, t_{s+3} \cdots t_{s+j} \cdots t_{s+m}$$

如果 $\quad p_0 \, p_1 \cdots p_{j-1} \neq p_1 \, p_2 \cdots p_j \qquad (2)$

则立刻可以断定

$$p_0 \, p_1 \cdots p_{j-1} \neq t_{s+1} \, t_{s+2} \cdots t_{s+j}$$

下一趟必不匹配

同样，若 $p_0 p_1 \cdots p_{j-2} \neq p_2 p_3 \cdots p_j$

则再下一趟也不匹配，因为有

$$p_0 p_1 \cdots p_{j-2} \neq t_{s+2} t_{s+3} \cdots t_{s+j}$$

直到对于某一个“k”值，使得

$$p_0 p_1 \cdots p_{k+1} \neq p_{j-k-1} p_{j-k} \cdots p_j$$

且 $$p_0 p_1 \cdots p_k = p_{j-k} p_{j-k+1} \cdots p_j$$

则 $$p_0 p_1 \cdots p_k = t_{s+j-k} t_{s+j-k+1} \cdots t_{s+j}$$

$$\| \quad \| \quad \|$$

$$p_{j-k} \; p_{j-k+1} \cdots \; p_j$$

k 的确定方法

当比较到模式第 j 个字符失配时，k 的值与模式的前 j 个字符有关，与目标无关。

利用失效函数 $f(j)$ 可描述。

利用失效函数 $f(j)$ 的匹配处理

如果 $j = 0$，则目标指针加 1，模式指针回到 p_0。

如果 $j > 0$，则目标指针不变，模式指针回到 $p_{f(j-1)+1}$。

若设 模式 $P = p_0 p_1 \cdots p_{m-2} p_{m-1}$

$$f(j) = \begin{cases} k, & \text{当} 0 \le k < j \text{且使得 } p_0 p_1 \cdots p_k = p_{j-k} p_{j-k+1} \cdots p_j \text{ 的最大整数.} \\ -1, & \text{其它情况.} \end{cases}$$

示例 ： 确定失效函数 $f(j)$

j	0	1	2	3	4	5	6	7
P	a	b	a	a	b	c	a	c
$f(j)$	-1	-1	0	0	1	-1	0	-1

运用KMP算法的匹配过程

第1趟 目标 a c a b a a b a a b c a c a a b c

模式 a b a a b c a c

× $j = 1 \Rightarrow j = f(j-1)+1 = 0$

第2趟 目标 a c a b a a b a a b c a c a a b c

模式 a b a a b c a c

× $j = 0$ 目标指针加 1

第3趟 目标 a c a b a a b a a b c a c a a b c

模式 a b a a b c a c

× $j = 5$

$\Rightarrow j = f(j-1)+1 = 2$

第4趟 目标 a c a b a a b a a b c a c a a b c

模式 (a b) a a b c a c √

```
int String :: fastFind ( String pat ) const {
 //带失效函数的KMP匹配算法
    int posP = 0,  posT = 0;
    int lengthP = pat.curLen,   lengthT = curLen;
    while ( posP < lengthP && posT < lengthT )
        if ( pat.ch[p osP] == ch[posT] ) {
              posP++;  posT++;   //相等继续比较
        }
        else if ( posP == 0 ) posT++;     //不相等
              else posP = pat.f [posP−1]+1;
    if ( posP < lengthP ) return −1;
    else return posT − lengthP;
 }
```

计算失效函数 $f[j]$ 的方法

首先确定 $f[0]=-1$，再利用 $f[j]$ 求 $f[j+1]$。

$$f[j+1]=\begin{cases} f^{(m)}[j]+1, & \text{可找到令 } p^{(k)}_{f[j]+1}=p_j \text{ 的最小正整数 } k \\ -1, & \text{否则或 } j=0 \end{cases}$$

其中，$f^{(1)}[j]=f[j]$，

$f^{(m)}[j]=f[f^{(m-1)}[j]]$

```
void String::fail ( ) {
//计算失效函数
    int lengthP = curLen;
    f [0] = −1;                                    //直接赋值
    for ( int j=1; j<lengthP; j++ ) {     //依次求f [j]
        int i = f [j−1];
        while ( *(ch+j) != *(ch+i+1) && i >= 0 )
            i = f [i];                  //递推
        if ( *(ch+j) == *(ch+i+1) )   f [j] = i+1;
        else f [j] = −1;
    }
}
```

小结：需要复习的知识点

- 作为抽象数据类型的数组
  - 数组的定义和初始化；
  - 作为抽象数据类型的数组；
  - 对称矩阵按上三角或下三角压缩存储时的地址转换公式
- 顺序表
  - 顺序表的定义和特点

- 顺序表的类定义
- 顺序表的查找、插入和删除算法
- 在顺序表中插入及删除时计算平均移动元素个数
- 使用顺序表的事例

稀疏矩阵

- 稀疏矩阵的三元组表表示;
- 稀疏矩阵的转置算法;

- 字符串
  - 字符串的抽象数据类型;
  - 字符串常用操作的实现;
  - 字符串模式匹配方法(不要求算法)