数据结构—C++实现

沈 俊

jshen@t.shu.edu.cn

上海大学 计算机工程与科学学院

2020年12月

第5章 字符串、数组和广义表

- 字符串
- 数组
- 稀疏矩阵
- 广义表

5.1 字符串

字符串(string)是由n（n≥0）个字符组成的有限序列。字符串简称为串，一般记为：

$$s = "a_0\ a_1\ \ldots\ a_{n-1}"$$

其中s是串名；用双引号括起来的字符序列是串值；a_i (0≤i<n)可以是ASCII码字符中的可打印字符，通常是字母、数字等字符；i称为字符a_i 在串中的位置；n称为串的长度；n =0时，s称为空串。

空格串是由一个或多个空格组成的串，如4个空格组成的空格串“␣␣␣␣”，它的长度是4。

字符串

串中任意多个连续的字符组成的子序列称为该串的子串。子串在该串中的位置就是子串的首字符在该串中的位置。

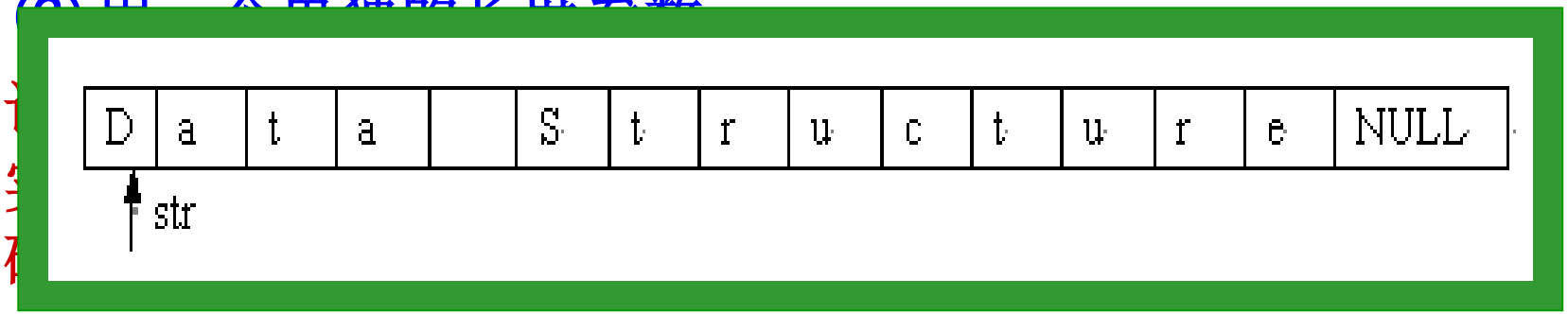
如果两个字符串对应位置的字符都相等，且它们长度相等，则称这两个字符串相等。

字符串的存储结构

采用顺序存储结构时，数组名指出了串在内存中的首地址。

有两种表示串长度的方法：

(1)在存储串的数组末尾添加一个结束标识符；



下述语句定义了一个字符数组并赋初值"Data Structure"：

```
char str[ ]= "Data Structure";
```

字符串"Data Structure"在内存中的存储形式如下：

字符串的操作

下面讨论串的操作，为了方便讨论，我们先定义如下几个串：

s1 = "It is a car"

s2 = "jeep"

s3 = "car"

串的操作主要有：

(1) 求串的长度。例如，s1的长度为11，s2的长度为4。

(2) 把一个串的值赋值给另一个串。若有s4＝s3，则s4 的值为 "car"。

字符串的操作

(3) 串的连接：把两个串连接形成一个长度为两个串长度之和的新串。设s5为s2和s3连接形成的新串，则s5="jeepcar"，即将后面的串连接在前面串的尾部。

(4) 串的比较：比较两个串的ASCII码值的大小。设str1和str2为两个串，按下述规则得到两个串的比较结果。

- 若str1小于str2，比较结果为－1；
- 若str1等于str2，比较结果为0；
- 若str1大于str2，比较结果为1。

字符串的操作

(5) 模式匹配：在一个串（称为主串）中查找是否存在和另一个串相等的子串。

若主串 s1 = "It is a car"
为str2 s2 = "jeep"
在str1 s3 = "car"

结果
字符

若主串str1中不存在和串str2相等的子串，则操作结果为－1。

例如，在串s1中查找到存在和串s3相等的子串，则操作结果为8。在串s1中未查找到和串s2相等的子串，则操作结果为－1。

字符串的操作

(6) 找字符：在一个串中是否存在一个字符。

设在串str中查找字符ch，若串str中存在字符ch，则操作结果为字符ch在串str中第一次出现的位置；

若串str中不存在字符ch，则操作结果为－1。

例如，在串s2中查找字符'e'，则操作结果为1；在串s2中查找字符'c'，则操作结果为－1。

s1 = "It is a car"
s2 = "jeep"
s3 = "car"

字符串的操作

(7) 取子串：截取子串形成一个新串。

设str为要截取的串，pos为要截取的起始位置，length为要截取的长度，则形成的新串长度为length。

例如：从s1串截取起始位置为3（这里是从0开始）、长度为2的子串放在s6中，则s6＝"is"。

s1 = "It is a car"
s2 = "jeep"
s3 = "car"

字符串的操作

(8) 在一个串中插入另一个串。设把串str2插入到串str1中，pos为要插入的起始位置，则操作结果形成的新串长度为str1和str2之和。

例如，把串s7=" not"，插入到串s1的位置5处，则操作结果形成的新串s8="It is not a car"。

s1 = "It is a car"
s2 = "jeep"
s3 = "car"

字符串的操作

(9) 从一个串中删除一个子串。设在串str中要删除长度为length的子串，pos为要删除的子串在str中的起始位置，则删除后的新串长度为原长度减去length。

例如，要在s8中删除长度为4的子串，起始位置为5，则删除后的新串为s1。

s1 = "It is a car"

s8 = "It is not a car"

(10) 从键盘输入一个字符序列。

(11) 在屏幕上显示一个字符序列。

常用的C++字符串函数

C++的串库(string.h)中提供了许多字符串操作函数。但C++的串库中提供的实现字符串操作的函数功能和前面讨论的字符串的基本操作功能不完全相同。

几个常用的C++字符串函数及其使用方法如下：

假设我们已有如下定义语句：

```
char s1[ ] = "It is a car";
char s2[ ] = "jeep";
char s3[ ] = "car";
int result;
char s4[20] , *p;
```

常用的C++字符串函数

(1) 串长度　int strlen(char *str);

```
cout << strlen(s1)<<endl;      //输出11
cout << strlen(s2)<<endl;      //输出4
```

(2) 串拷贝　char *strcpy(char *str1，char *str2);

```
strcpy(s4，s2);                  //s4 = "jeep"
```

(3) 串连接　char *strcat(char *str1，char *str2);

```
strcat(s2，s3);                 //s4 = "jeepcar"
```

(4) 串比较　char *strcmp(char *str1，char *str2);

```
result = strcmp(s2，s3);        //result>0
result = strcmp(s2，s2);        //result=0
result = strcmp(s3，s2);        //result<0
```

(5) 串中字符定位　char *strchr(char *str , char ch);

```
p = strchr(s1 , 'c'); //p指在s1中字符'c'首次出现的位置
strcpy(p , s2);                 //s1 = "It is a jeep"
```

常用的C++字符串函数

C++的流库（iostream.h）为字符输入流cin（键盘）和字符输出流cout（屏幕）提供I/O操作，使用方法如下：

(1) 读串 Stream Variable >>str

```
cin>>s4;    //若输入"Hello!"，则s4 = "Hello!"
```

(2) 写串 Stream Variable <<str

```
cout << s4;                    //输出"Hello!"
```

C++的串库提供的实现字符串操作的函数功能较强，但由于函数定位通常用字符指针，所以使用的难度较大。

常用的C++字符串函数

例5-1　名和姓的对换问题。英国和美国人的姓名是名在前姓在后，中间由一个空格符分隔，如Jeffer Offutt。但在有些情况下，需要把姓名写成姓在前名在后中间加一个逗号的形式。编写一个程序，把名在前姓在后的姓名表示法转换成姓在前名在后中间加一个逗号的姓名表示法。

常用的C++字符串函数

```
#include "assistance.h"
void reversename(char *name , char *newname)  {
    char *p;
    p = strchr(name , ' ');
    *p = '\0';
    strcpy(newname , p+1);
    strcat(newname , ",");
    strcat(newname , name);
    *p = ' ';
    return;
}
```

常用的C++字符串函数

```
int main() {
    char name[30], newname[30];
    cout << “输入一个人姓名：名在前，姓在后，中间有一个空格分隔。" << endl;
    cin.getline(name,30,'\n');
    reversename(name , newname);
    cout << “reversename:” << newname << endl;
    system("PAUSE");
    return 0;
}
```

程序运行如下：

```
Jeffer Offutt                          //键盘输入
reversename:Offutt, Jeffer  //屏幕输出
```

串类的定义及其实现

使用重载操作符的办法，使得C++中的一些运算符具有新的功能。如："=="具有串的比较功能，等等。可以定义满足如下要求的串类：

(1) 可以把串对象或C++串赋给另一串对象。
(2) 串对象的连接用"+"算符完成。
(3) 串比较是用关系算符执行的。
(4) 串对象可用C++串或另一串对象初始化。
(5) 串必须拥有任意可变长度。
(6) 可以把串对象转换为C++串。
(7) 可以查找子串。

串的类定义

```
class String {
protected:
        char *sVal;   // 串值
        int length;    // 串长
```

串类的定义

```
public:
        String();          // 构造函数
        virtual ~String();       // 析构函数
        String(const String &s);      // 复制构造函数
        String(const char *s);         // 转换构造函数
        String(LinkList<char> &s //线性表转换的构造函数
        int GetLength() const;         // 求串长度
        bool IsEmpty() const;          // 判断串是否为空
        String &operator =(const String &s);       // 赋值语句重载
        const char *CStr() const;// 将串转换成字符数组
        char &String::operator [](int p) const;      // 重载下标运算符
};
```

串类的实现

(1) 构造函数

操作结果：初始化一个空串

```
String::String()
{
        length = 0;             // 串长度为0
        sVal = NULL;// 空串
}
```

串类的实现

(2) 析构函数

操作结果：销毁串，释放串的存储空间

```
String::~String()
{
        delete []sVal;// 释放串的存储空间sVal
}
```

串类的实现

(3) 复制构造函数

操作结果：由串s构造一个新串

```
String::String(const String &s)
{
        length = strlen(s.CStr());     // 设置串长
        sVal = new char[length + 1];// 分配存储空间
        strcpy(sVal, s.CStr());                  // 复制串值
}
```

串类的实现

(4) 转换构造函数

操作结果：用C++的字符数组s转换构造新串

```
String::String(const char *s)
{
        length = strlen(s);              // 设置串长
        sVal = new char[length + 1];// 分配存储空间
        strcpy(sVal, s);                 // 复制串值
        sVal[length] = '\0';
}
```

串类的实现

(5) 转换构造函数

操作结果：从线性链表s转换构造新串

```
String::String(LinkList<char> &s)
{
        length = s.GetLength();      // 串长
        sVal = new char[length + 1];// 分配存储空间
        for (int i = 0; i < length; i++) // 复制串值
                s.GetElem(i + 1, sVal[i]);
        sVal[length] = '\0';              // 串值以'\0'结束
}
```

串类的实现

(6) 求串的长度

操作结果：返回串长度，不改变数据成员。

```
int String::GetLength() const
{
        return length;
}
```

串类的实现

(7) 判断是否为空串

操作结果：如果是空串则返回true，否则返回false。不改变数据成员

```
bool String::IsEmpty() const
{
        return length == 0;
}
```

串类的实现

(8) 赋值语句重载

操作结果：把赋值右边的串赋给串对象

```
String &String::operator =(const String &s)
{
        if (&s != this)  {
            delete []sVal;        // 释放原串存储空间
            length = strlen(s.CStr()); // 设置串长
            sVal = new char[length + 1];// 分配存储空间
            strcpy(sVal, s.CStr());    // 复制串值
        }
        return *this;
}
```

串类的实现

(9) 串转换为C风格串

操作结果：将串转换成C的字符数组返回

```
const char *String::CStr() const
{
        return (const char *)sVal;
        // 串值类型转换
}
```

串类的实现

(10) 重载下标运算符

操作结果：取串中下标为p的字符

```
char &String::operator [](int p) const
{
        return sVal[p];
}
```

串的相关函数定义

```
void Write(const String &s);  // 输出串
void Copy(String &s1, const String &s2);
      // 将串s2复制到串s1
void Copy(String &s1, const String &s2, int n);
      // 将串s2复制n个字符到串s1
Status Insert(String &s1, const String &s2, int p);
      // 将字符串s2插入到s1的p位置
Status Delete(String &s, int p, int n);
      // 删除字符串s中从p位置开始长度为n的字符串
String SubString(const String &s, int p, int n);
      // 求串s的第p个字符开始的长度为n的子串
```

串的相关函数定义

```
String operator +(const String &s1, const String &s2);
        // 重载连接运算符+
bool operator ==(const String &s1, const String &s2);
        // 重载关系运算符==
bool operator <(const String &s1, const String &s2);
        // 重载关系运算符<
bool operator >(const String &s1, const String &s2);
        // 重载关系运算符>
```

串的相关函数定义

```
bool operator <=(const String &s1, const String &s2);
      // 重载关系运算符<=
bool operator >=(const String &s1, const String &s2);
      // 重载关系运算符>=
bool operator !=(const String &s1, const String &s2);
      // 重载关系运算符!=
```

串的相关函数实现

(1) 串的输出

操作结果：输出串的值。

```
void Write(const String &str)
{
        cout << str.CStr() << endl;
        // 输出串值
}
```

串的相关函数实现

(2) 串的复制

操作结果：将串s2复制到串s1。

```
void Copy(String &s1, const String &s2)
{
        const char *cs2 = s2.CStr();            // 初始串
        char *cs1 = new char[strlen(cs2) + 1];    // 申请临时空间
        strcpy(cs1, cs2);                        // 复制串
        s1 = cs1;                                // 串赋值
        delete []cs1;                            // 释放临时空间
}
```

串的相关函数实现

(3) 取串的前缀

操作结果：将串s2长度为n的前缀复制字符到串s1。

```
void Copyn(String &s1, const String &s2, int n)
{
        const char *cs2 = s2.CStr();            // 初始串
        int len = strlen(cs2) < n ? strlen(cs2) : n; // 取目标串长
        char *cs1 = new char[len + 1];      // 申请临时空间
        strncpy(cs1, cs2, n);                   // 复制串
        cs1[len] = '\0';                    // 串值以'\0'结束
        s1 = cs1;                           // 串赋值
}
```

串的相关函数实现

(4) 插入子串

```
Status Insert(String &s1, const String &s2, int p)
{
        const char *cs1 = s1.CStr();                    // 取第一个串
        const char *cs2 = s2.CStr();                    // 取第二个串
        if (p >=0 && p < strlen(cs1)) {
            int len = strlen(cs1) + strlen(cs2);
            char *cs = new char[len + 1];     // 申请临时空间
            strncpy(cs, cs1, p);   // 复制第一个串前部分的串值
            cs[p] = '\0';
            strcat(cs, cs2);       // 连接第二个串
```

串的相关函数实现

(4) 插入子串

```
        int    j = p + strlen(cs2);
        for (int i = p; i < strlen(cs1); i++, j++)
                cs[j] = cs1[i];       // 复制第一个串后部分的串值
        cs[len] = '\0';
        s1 = cs;                                    // 串赋值
        return SUCCESS;
    }
    return RANGE_ERROR;
}
```

串的相关函数实现

(5) 删除子串

```
Status Delete(String &s, int p, int n)
{
        const char *cs = s.CStr();                  // 取原串值
        if (p >=0 && (p + n) < strlen(cs)) {
            int len = strlen(cs) - n;          // 求新串的长度
            char *news = new char[len + 1];  // 申请临时空间
            strncpy(news, cs, p);          // 复制原串前部分的串值
            int    j = p + n;
            for (int i = p; i < len; i++, j++)
                    news[i] = cs[j];
                    // 复制第一个串后部分的串值
```

串的相关函数实现

(5) 删除子串

```
            news[len] = '\0';
            s = news;                          // 串赋值
            delete []news;                     // 释放临时空间
            return SUCCESS;
        }
        return RANGE_ERROR;
}
```

串的相关函数实现

(6) 取子串

操作结果：当0 <= p < s.Length()且0 <= n <= s.Length() – p时返回串s的第p个字符开始的长度为n的子串，否则返回空串。

```
String SubString(const String &s, int p, int n)
{
        if  (0 <= p && p + n < s.Length() && 0 <= n)      {
            char *sub = new char[n + 1];    // 申请临时空间
            const char *strp = s.CStr();      // 生成字符数组
            strncpy(sub, strp + p, n);          // 复制串
            sub[n] = '\0';                          // 串值以'\0'结束
            String cs(sub);                        // 生成串对象
```

串的相关函数实现

```
        delete []sub;        // 释放临时空间
        return cs;
    }
    else    {                // 返回空串
        String cs("");       // 生成空串对象
        return cs;
    }
}
```

串的相关函数实现

(7) 串的连接

操作结果：重载字符串连接运算符+。

```
String operator +(const String &s1, const String &s2)
{
        const char *cs1 = s1.CStr();          // 取第一个串值
        const char *cs2 = s2.CStr();          // 取第二个串值
        char *cs = new char[strlen(cs1) + strlen(cs2) + 1];
        strcpy(cs, cs1);          // 复制第一个串
        strcat(cs, cs2);          // 连接第二个串
        String s(cs);             // 定义串对象并初始化
        delete []cs;              // 释放临时空间
        return s;
}
```

串的相关函数实现

(8) 判字符串是否相等

操作结果：重载字符串关系运算符==。

```
bool operator ==(const String &s1, const String &s2)
{
        return strcmp(s1.CStr(), s2.CStr()) == 0;
}
```

模式匹配

设有两个字符串ob和pat，若要在串ob中查找与串pat相等的子串，则称ob为主串（或称目标串），pat为模式串，并称查找模式串在主串中的匹配位置的运算为模式匹配。

该运算从ob的第一个字符开始查找一个与串pat（称作模式串）相同的子串。

如果在串ob中查找到一个与模式串pat相同的子串，则函数返回模式串pat的头一个字符在串ob中的位置；

如在主串中未查找到一个与模式串pat相同的子串，则函数返回-1。

Brute-Force算法

Brute-Forc算法的主要思想是：从主串ob = “s_0 s_1 … s_{m-1}”的第一个字符开始与模式串pat = “t_0 t_1 … t_{n-1}”的第一个字符比较：

若相等，则继续比较后续字符；

否则，从主串ob的第二个字符开始重新与模式串pat的第一个字符比较。

如此继续，若在主串ob中有一个与模式串相等的连续字符序列，则匹配成功，函数返回模式串pat的首字符在串ob中的位置；

否则，匹配失败，函数返回－1。

Brute-Force算法

看一个模式匹配的例子。设主串ob = “ZWWZWZ”，模式串pat = “ZWZ”，ob的长度为m = 6，pat的长度为n = 3，用变量i指示主串ob的当前比较字符的下标，用变量j指示模式串pat的当前比较字符的下标。

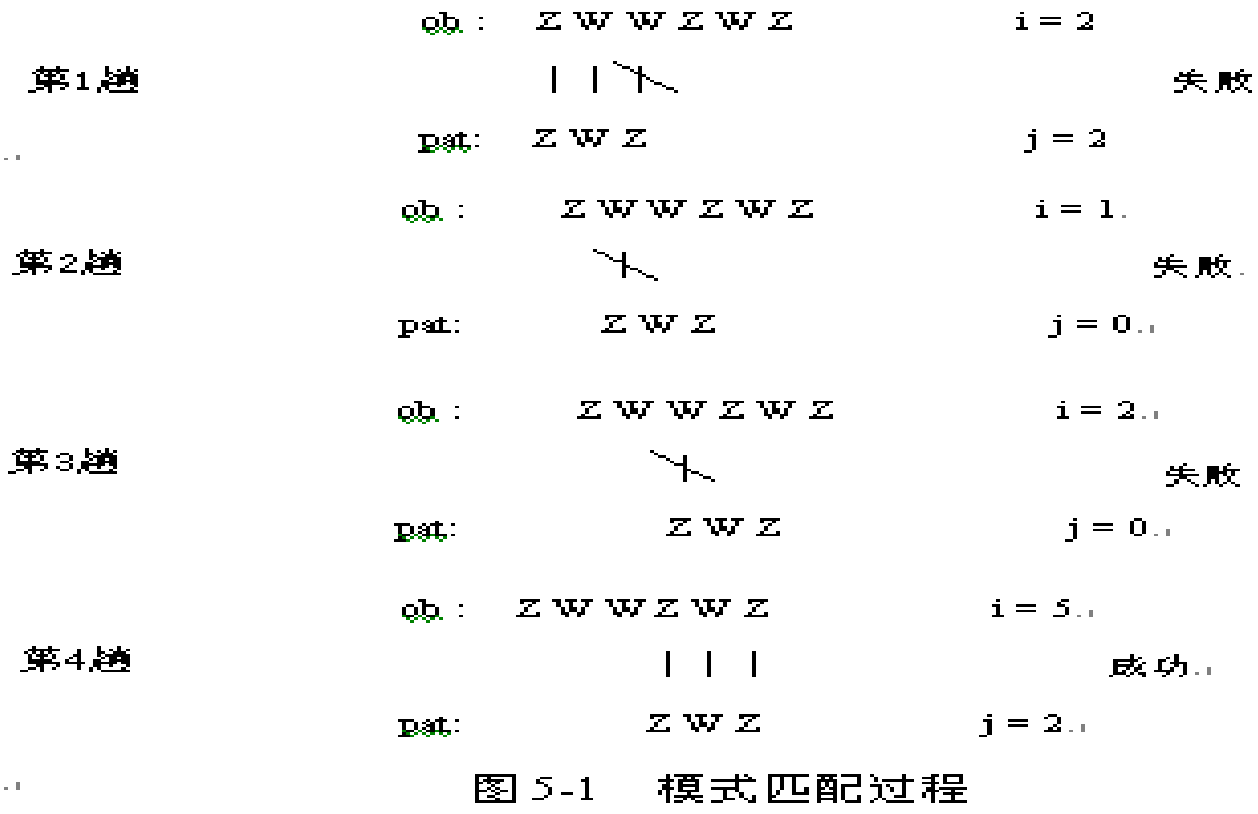


图 5-1　模式匹配过程

Brute-Force算法

对于一般的模式匹配过程如图5-2所示，这里给出了某一趟比较的状态和下一趟比较时下标变化的一般过程。

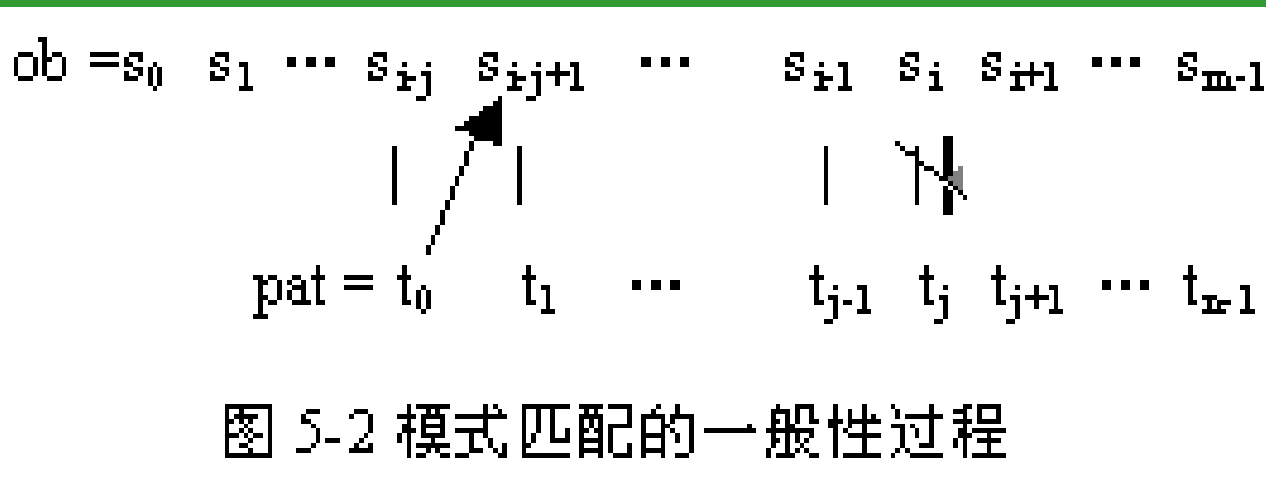


图 5-2 模式匹配的一般性过程

Brute-Force算法

```
int BF_find(const String &ob, const String &pat, const int p = 0)
{
    int i = p, j = 0;
    while (i < ob.Length() && j < pat.Length())
                    && pat.Length() - j <= ob.Length() - i)
        if (ob[i] == pat[j])          {        // 继续比较后续字符
                    i++; j++;
         }
        else      {                            // 指针回退,重新开始新的匹配
                    i = i - j + 1; j = 0;
         }
   if (j >= pat.Length())   return i - j;       // 匹配成功
   else                     return -1;          // 匹配失败
}
```

Brute-Force算法

Brute-Force算法是一种带回溯的算法，也叫朴素的模式匹配算法。在最坏情况下，最多要比较m－n＋1趟，每趟比较在最后才出现不等，要做n次比较，总比较次数要达到(m－n＋1)*n。通常n会远远小于m，因此，算法的最坏情况下运行时间为O(m*n)。

模式匹配的KMP算法

KMP算法是由D.E.Knuth、J.H.Morris和V.R.Pratt三人设计的。该算法是Brute-Force算法的改进。

它消除了Brute-Force算法中主串下标i在对应字符比较不相等时需要回退的现象。

真子串（最长的相等前缀和后缀）：在字符串"t_0 t_1 … t_{n-1}"中最长的相等前缀和后缀称为该字符串的真子串，也就是说在字符串"t_0 t_1 … t_{n-1}"中存在一个最大的k(0 < k < n)，使得"t_0 t_1 … t_{k-1}"= "$t_{n-k}t_{n-k+1}$ … t_{n-1}"，则"t_0 t_1 … t_{k-1}"就称为"t_0 t_1 … t_{n-1}"的真子串。需要注意的是真子串的前缀和后缀可以有重叠部分，但不能完成重叠。

模式匹配的KMP算法

(1) 模式串中无真子串

主串：　“ZWWZWX”

模式串：“ZWX”

当$s_0=t_0$，$s_1=t_1$，$s_2\neq t_2$时，算法中取i =1，j =0，使主串下标i值回退，然后比较s_1和t_0。

但是因为$t_1\neq t_0$，所以一定有$s_1\neq t_0$（ $s_1=t_1$），实际上接下来就可直接比较s_2和t_0。

模式匹配的KMP算法

(2) 模式串中有真子串

主串：　“ACACABAB”

模式串：“ACAB”

此时，因$t_0 \neq t_1$，$s_1 = t_1$，必有$s_1 \neq t_0$；又因$t_0 = t_2$，$s_2 = t_2$，必有$s_2 = t_0$，因此接下来可直接比较s_3和t_1。

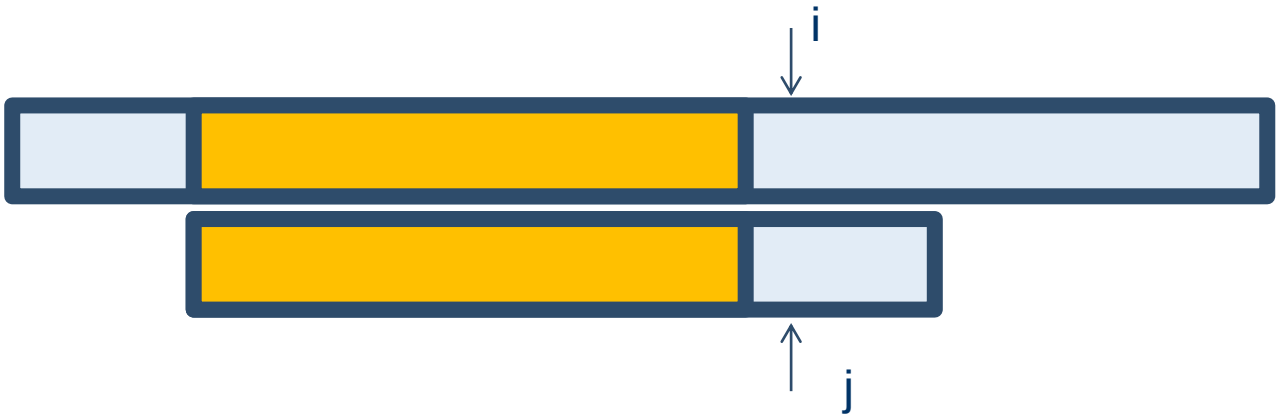
由此可见，一旦s_i和t_j比较不相等，主串ob的下标值可不必回退，主串的s_i（或s_{i+1}）可直接与模式串的t_k（$0 \leq k < j$）比较。

模式匹配的KMP算法

下面来讨论一般情况。设ob = “$s_0\ s_1\ \cdots\ s_{m-1}$”，pat = “$t_0\ t_1\ \cdots\ t_{n-1}$”，当$s_i \neq t_j$ ($0 \leqslant i < m$，$0 \leqslant j < n$)时，模式串“向右滑动”，可行的距离多远呢?

换句话说，当主串中第i个字符与模式串中第j个字符“失配”（即比较不等）时，主串中的s_i应与模式串中的哪个字符再比较？

因为已有："$s_{i-j}\ s_{i-j+1}\ \cdots\ s_{i-1}$" = "$t_0\ t_1\ \cdots\ t_{j-1}$"

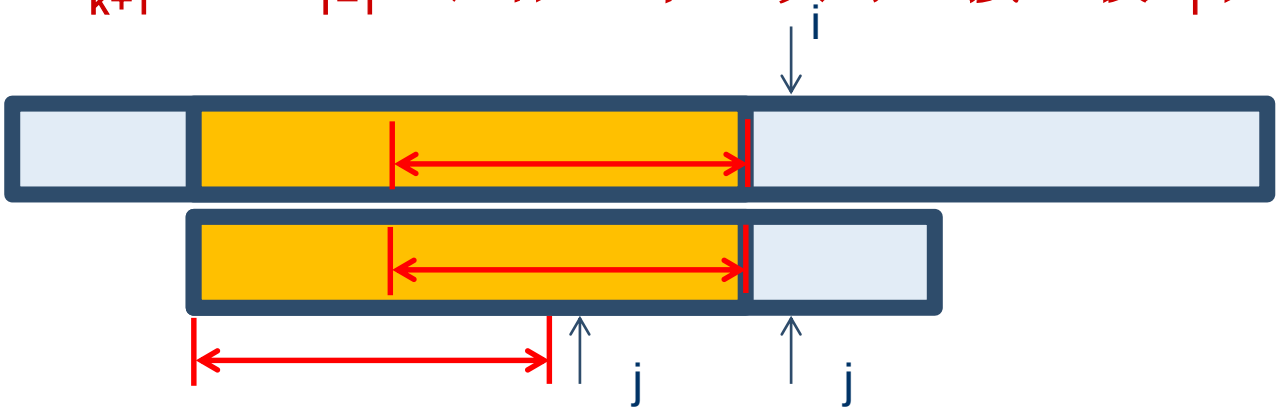


模式匹配的KMP算法

1、若模式串中t_j之前存在长度为k的真子串，即满足

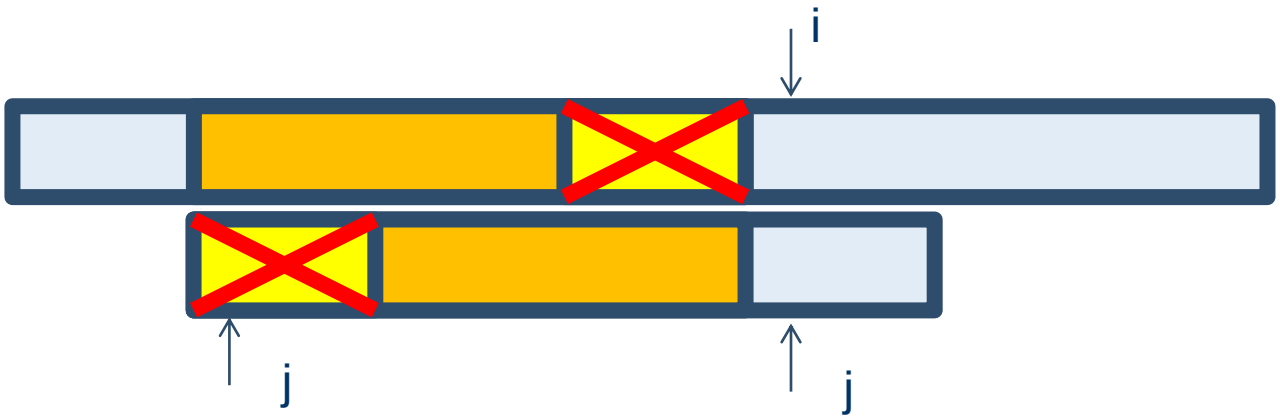
$$"t_0\ t_1\ \cdots\ t_{k-1}" = "t_{j-k}t_{j-k+1}\ \cdots\ t_{j-1}" \qquad (0<k<j)$$

则说明模式串中的子串"$t_0\ t_1\ \cdots\ t_{k-1}$"已和主串"$s_{i-k}s_{i-k+1}\ \cdots\ s_{i-1}$"匹配，下一次可直接比较$s_i$和$t_k$。



模式匹配的KMP算法

2、若模式串中t_j之前不存在相互重叠的真子串，则说明在模式串“$t_0\ t_1\ \cdots\ t_{j-1}$”中不存在任何以$t_0$为首字符的字符串与"$s_{i-j}\ s_{i-j+1}\ \cdots\ s_{i-1}$“ 中以$s_{i-1}$为末字符的字符串匹配，下一次可直接比较$s_i$和$t_0$。



模式匹配的KMP算法

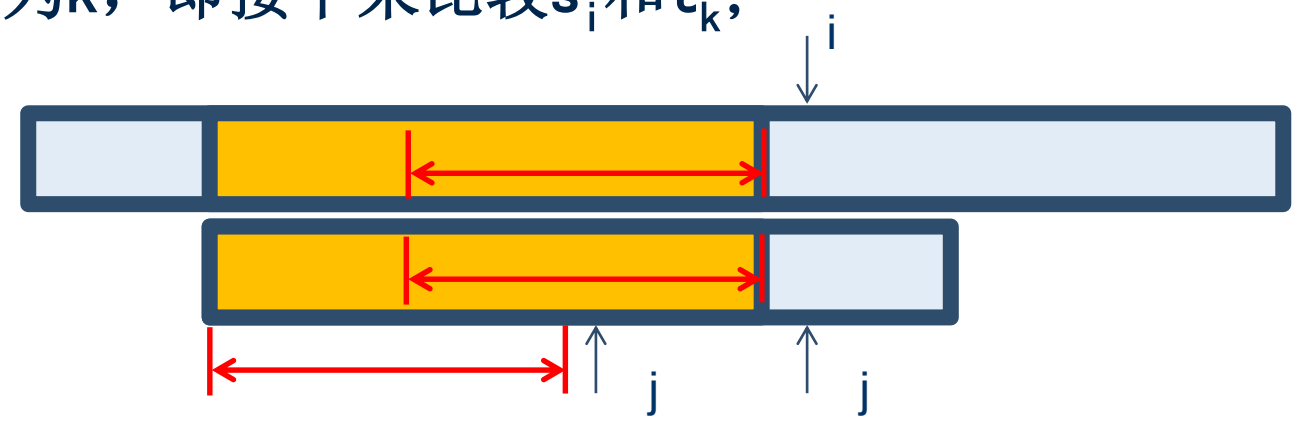
f[j]表示模式串中t_j之前的真子串的长度。即：

$$f[j]=\begin{cases}\max\{k \mid 0<k<j \text{且} "t_0t_1\cdots t_{k-1}"="t_{j-k}\cdots t_{j-1}"\} & \text{当此集合非空时}\\ -1 & \text{当 } j=0 \text{ 时}\\ 0 & \text{其它情况时}\end{cases}$$

模式匹配的KMP算法

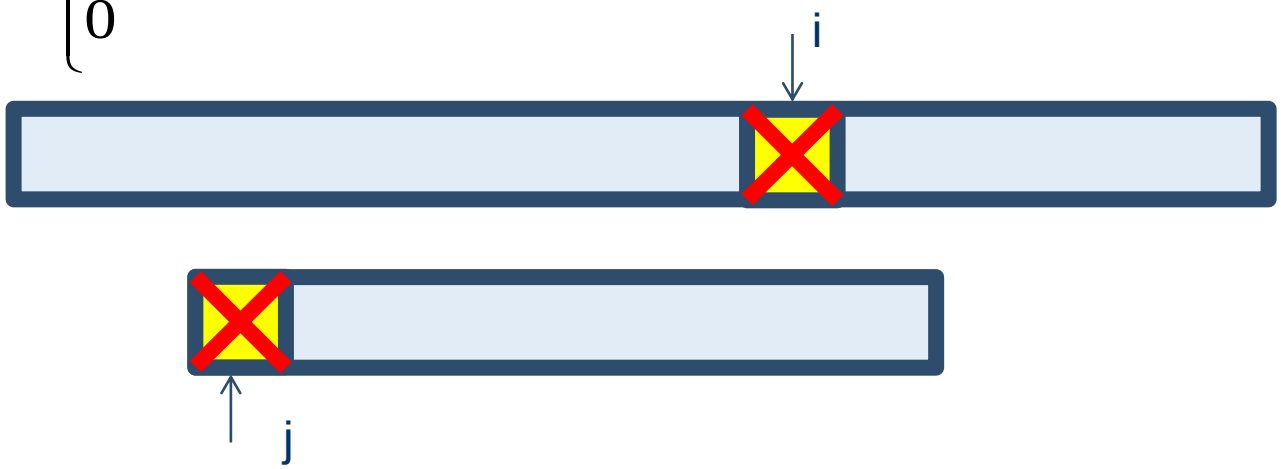
$$f[j]=\begin{cases}\max\{k \mid 0<k<j \text{且} "t_0t_1\cdots t_{k-1}"="t_{j-k}\cdots t_{j-1}"\} & \text{当此集合非空时}\\ -1 & \text{当 } j=0 \text{ 时}\\ 0 & \text{其它情况时}\end{cases}$$

1、若模式串中t_j存在真子串"t_0 t_1 … t_{k-1}"= "$t_{j-k}t_{j-k+1}$ … t_{j-1}"且满足0＜k＜j，则当模式串pat中的t_j与主串ob的s_i比较不相等时，模式串pat中需重新与主串ob的s_i比较的字符下标为k，即接下来比较s_i和t_k；



模式匹配的KMP算法

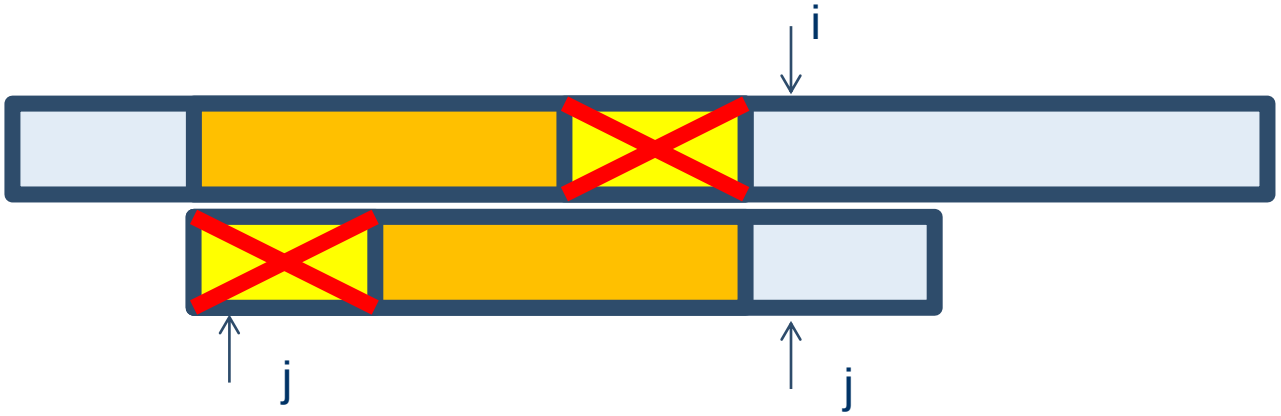
$$f[j]=\begin{cases}\max\{k \mid 0<k<j \text{且} "t_0t_1\cdots t_{k-1}"="t_{j-k}\cdots t_{j-1}"\} & \text{当此集合非空时}\\ -1 & \text{当 } j=0 \text{时}\\ 0 & \text{其它情况时}\end{cases}$$



2. 当j=0时，f[j]=-1，此处-1为一标记，表示接下来比较s_{i+1}和t_0；

模式匹配的KMP算法

$$f[j]=\begin{cases}\max\{k \mid 0<k<j \text{且} "t_0t_1\cdots t_{k-1}"="t_{j-k}\cdots t_{j-1}"\} & \text{当此集合非空时}\\ -1 & \text{当 } j=0 \text{时}\\ 0 & \text{其它情况时}\end{cases}$$



3. 若模式串pat中不存在如上所说的真子串，有f[j]=0，则接下来比较s_i和t_0。

模式匹配的KMP算法

$$f[j]=\begin{cases}\max\{k \mid 0<k<j \text{且} "t_0t_1\cdots t_{k-1}"="t_{j-k}\cdots t_{j-1}"\} & \text{当此集合非空时}\\ -1 & \text{当 } j=0 \text{时}\\ 0 & \text{其它情况时}\end{cases}$$

这里的函数f[j]被称为失效函数(failure function)，又称为失败函数。

总之，KMP算法对Brute-Force算法的改进就是利用已经得到的部分匹配结果将模式串pat右滑一段距离再继续比较，从而无需回退主串ob的下标值

模式匹配的KMP算法

```
int KMP_find(const String &ob, const String &pat, int p = 0)
{
    int *f = new int[pat.GetLength()];
    GetFailure(pat, f);                    // 求模式串pat的f数组的元素值
    int i = p, j = 0;
    while (i < ob.GetLength() && j < pat.GetLength()
                              && pat.GetLength() - j <= ob.GetLength() - i)
        if (j == -1 || pat[j] == ob[i])  {   i++; j++;     }
        else          j = f[j];         // 寻找新的模式串pat的匹配字符位置
     delete []f;
     if (j < pat.GetLength())        return -1;   // 匹配失败
     else        return i - j;                    // 匹配成功
}
```

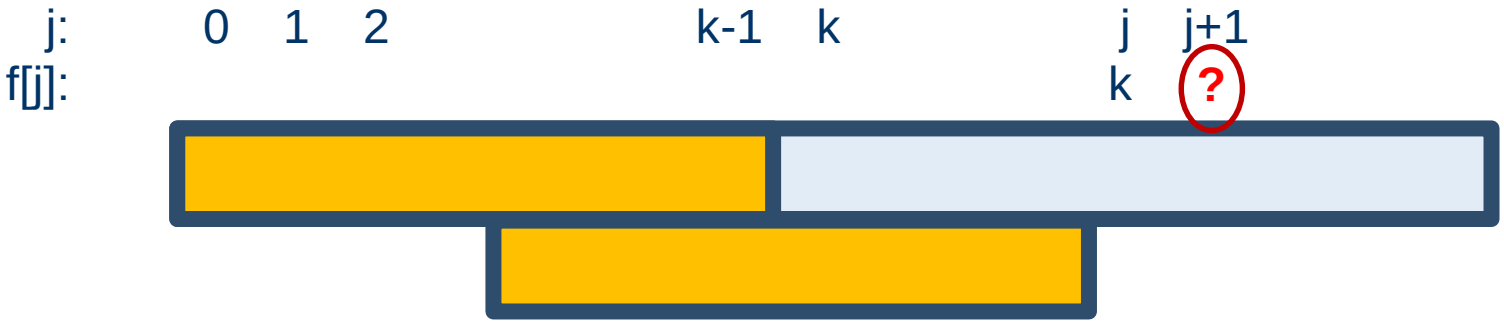
模式匹配的KMP算法

该算法的时间复杂度取决于其中的while循环，由于该算法中无回溯，在进行对应的字符比较后，要么是ob的下标值加1，要么是调整模式串的下标值，继续向后比较。字符比较的次数最多为0(ob.size)。

模式匹配的KMP算法

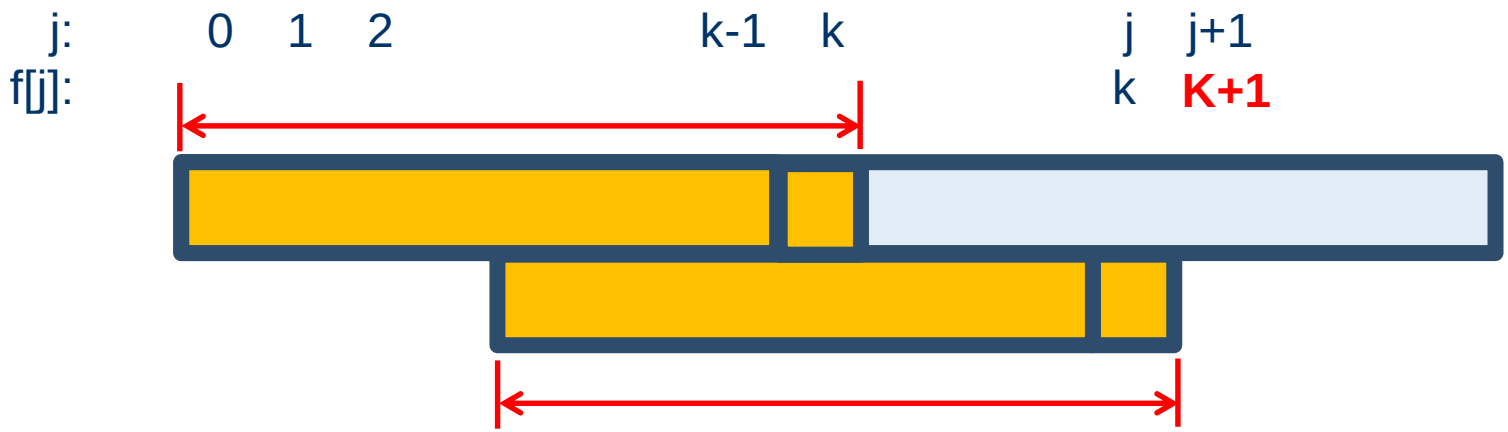
下面再来讨论求失效函数f[j]的算法。

从上面计算f[j]的例子可以看出f[j]的计算是一个递推计算问题。设有f[j] = k，即在模式串pat中存在"t_0 t_1 … t_{k-1}“ = "t_{j-k} t_{j-k+1} … t_{j-1}"（0＜k＜j），其中k为满足等式的最大值，则计算f[j+1]的值有两种情况：



模式匹配的KMP算法

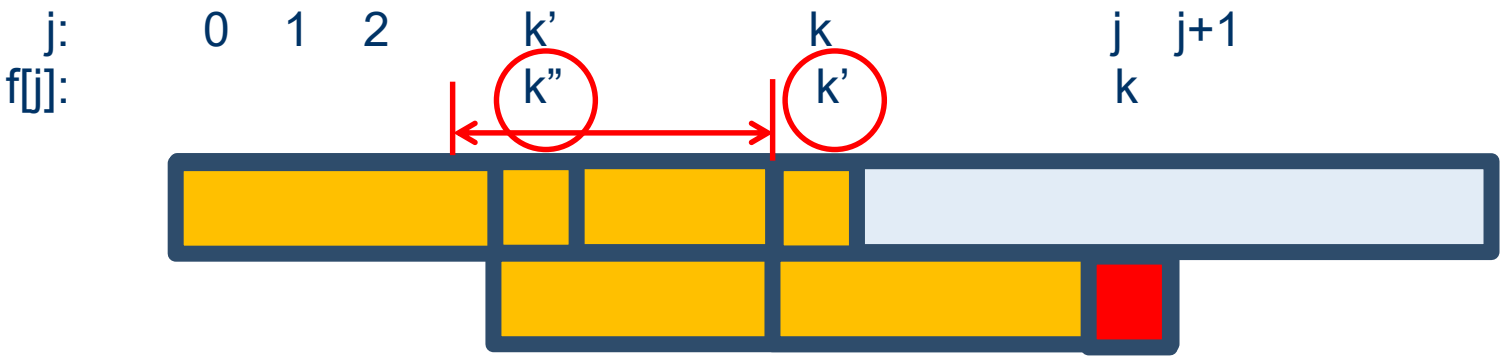
1、若$t_k = t_j$，则表明在模式串pat中有"$t_0\ t_1\ \cdots\ t_k$" = "$t_{j-k}t_{j-k+1}\ \cdots\ t_j$"，且不可能存在任何k'＞k满足上式，因此有：



$$f[j+1] = f[j] + 1 = k+1$$

模式匹配的KMP算法

2、若$t_k \neq t_j$，设：f[k]=k' （0<k'<k<j），若$t_{k'} = t_j$，则表明在模式串中有"t_0 t_1 … $t_{k'}$"= "$t_{j-k'}t_{j-k'+1}$ … t_j" （0<k'<k<j)，因此有：



f[j+1] = f [k] +1 = k’+1

若$t_{k'} \neq t_j$，则将模式串pat继续右滑到k‘’=f[k‘]。依此类推，直到某次匹配成功或匹配失败，最后一次匹配失败为

f[j+1] = f[0]+1 = -1+1=0

模式匹配的KMP算法

计算pat= "ABCABCAAA"的f[j]。

模式	A	B	C	A	B	C	A	A	A
j	0	1	2	3	4	5	6	7	8
f[j]									

模式匹配的KMP算法

```
void GetFailure(const String &pat, int f[])
{
    int j = 0, k = -1;
    f[0]=-1;                          // 初始f[0]的值为-1
    while (j < pat.Length() - 1)
        if (k == -1 || pat[k] == pat[j])
                f[++j]=++k;
        else                           // pat[k]与pat[j]不匹配
                k = f[k];              // 寻求新的匹配字符
}
```

模式匹配的KMP算法

如果主串的长度为m，模式串的长度为n，则包括计算失效函数在内的整个模式匹配KMP算法的时间复杂度为O(m+n)。

这里再看一个例子，模式串pat=“aaaaab”，其失效函数值为：-1、0、1、2、3、4，假设主串ob=“aaaacaaaaab”，当i=4、j=4时$s_4 \neq t_4$（s_4=’c’、t_4=’a’），由于f[4]=3，因此会用t_3和s_4进行比较。但实际上t_3==t_4，所以肯定有$s_4 \neq t_3$，同样t_2、t_1、t_0与s_4的比较也是无用比较。由此可见，上面求失效函数的算法还可以进一步改进。如何改进，请读者自己考虑。

5.2 数组

数组是读者已经很熟悉的一种数据结构，几乎所有的程序设计语言都把数组类型设定为固有类型。事实上，在前几章中，已经使用了C++的一维数组来存放线性表。在本节中介绍数组的定义以及相应的存储结构。

数组的基本概念

通常，一维数组A(array)是n（n≥0)个相同数据类型的数据元素a_0，a_1，…，a_{n-1}构成的有限线性序列。其中n叫做数组长度或数组大小，若n＝0就是空数组。当每一个数组元素a_i（0≤i≤n-1）本身又是一个一维数组时，则A就是一个二维数组。类似地，我们可以构成一个多维数组,一个m(m≥2)维数组中的每一个数组元素是一个m-1维的数组。

可见在一个m(m≥2)维数组中，每一个数组元素受m个线性关系的约束，如果一个元素在每一维中的序号分别为i_1、i_2、…、i_m，则称该元素的下标为：i_1、i_2、…、i_m。如果一个数组名为a，则$a_{i1i2\cdots im}$表示下标为i_1、i_2、…、i_m的数组元素。

数组的顺序存储结构

采用顺序存储结构存储数组的元素，就是按某种顺序将数组元素依次存放在内存中的一片连续的存储单元中。

数组的每个元素的数据类型都相同，因而占有相同的存储空间。对于一维数组，相邻元素的起始地址之差为一常数。

例如，一维数组a[5]的5个元素顺序存储的示意图。

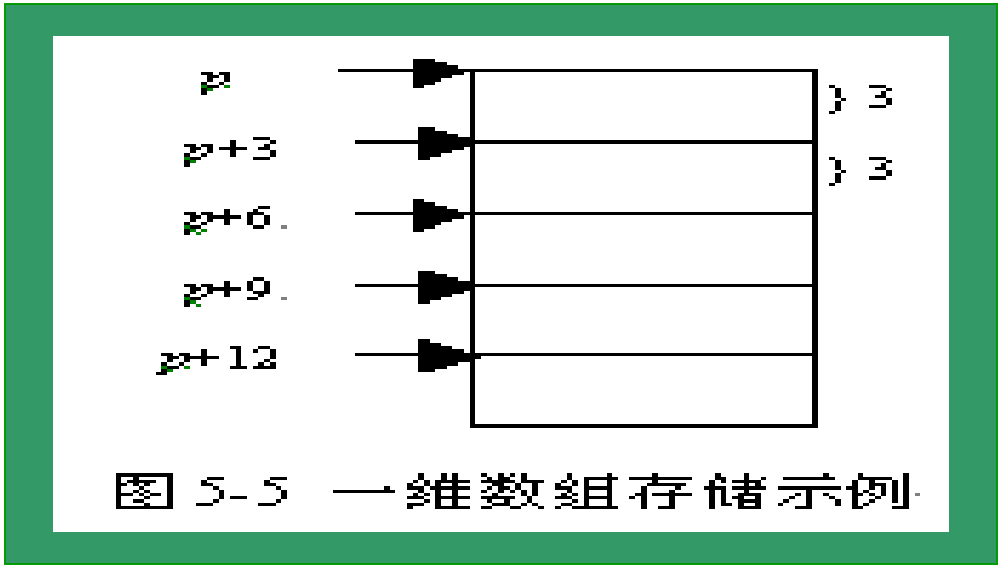


图 5-5 一维数组存储示例

数组的顺序存储结构

每个元素占3个存储单元，数组第1个元素a[0]的起始地址为loc(0) = p，则该数组的任一数组元素a[i]的起始地址loc(i)可由以下的递推公式计算：

$$loc(i)=\begin{cases} p & i=0 \\ loc(i-1)+3 & i>0 \end{cases}$$

实际上即有：loc（i）= p+i*3　　（i = 0，1，2，3，4）

数组的顺序存储结构

可有两种存储方式：一种是以行序为主序的存储方式。如，C，C++，PASCAL等语言都采用这种存储方式。

另一种是以列序为主序的存储方式，例如FORTRAN语言就采用这种存储方式。

对于二维数组a[m][n]，用a_{ij}来表示数组元素a[i][j]，则二维数组a可以表示为：

$$a=\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0\,n-1} \\ a_{10} & a_{11} & \cdots & a_{1\,n-1} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m-1\,0} & a_{m-1\,1} & \cdots & a_{m-1\,n-1} \end{bmatrix}$$

数组的顺序存储结构

图5-6(a) 显示了以行序为主序的存储方式，图5-6(b)显示了以列序为主序的存储方式。

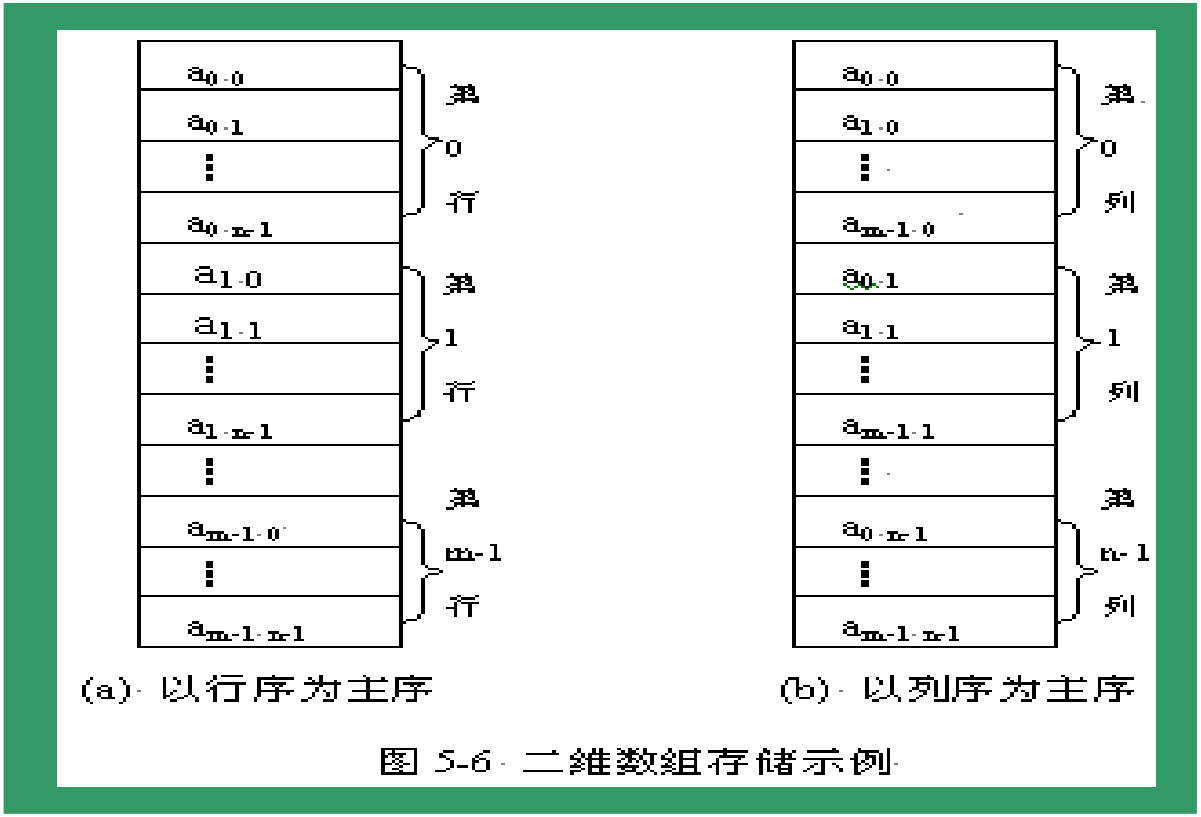


图 5-6 二维数组存储示例

数组的顺序存储结构

假设二维数组a[m][n]的首地址为p，即a[0][0]的起始地址为p，每个元素占l个存储单元，以行序为主序存储方式来存储数组a，计算数组元素a[i][j]的起始地址loc(i，j)。因为对于一维数组a中的第k个元素的起始地址是：

$$loc(k) = p+k*l$$

因此只要计算出a[i][j]是数组的第几个元素就可计算出loc(i，j)。

若设其为k，则有k = i*n+j，所以：

$$loc(i，j) = p+ (i*n+j) *l$$

5.3 稀疏矩阵

矩阵本身就是二维数组。对于一个矩阵，如果零元素较多，还是采用上一节所述的存储方式来存储的话，就会使得大量的存储空间存放同一个值零，从而造成事实上的存储空间的浪费。

本节，将讨论这种矩阵如何进行压缩存储，以及基本操作的实现。

像这种零元素非常多的矩阵称为稀疏矩阵。显然，关于“稀疏”的定义是无法精确给出的。因为稀疏矩阵是非零元素很少的矩阵，我们只要存储非零元素就行了。

整个稀疏矩阵的存储结构既可以采用顺序结构存储，也可以采用链式结构存储。

非零元素的三元组定义

```
template<class ElemType>
struct Triple
{
// 数据成员:
        int row, col;           // 非零元素的行下标与列下标
        ElemType value;         // 非零元素的值
// 构造函数:
        Triple(){};
        Triple(int r, int c, ElemType v);
};
```

三元组顺序表

$$M = \begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -5 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 7 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix}$$

(a) 矩阵 M

	row	col	value
triElems [0]	0	0	15
[1]	0	3	22
[2]	0	5	-5
[3]	1	1	11
[4]	1	2	3
[5]	2	3	6
[6]	4	0	91
[7]	5	1	7
[8]	6	2	28

(a.rows=7, a.cols=6, a.num=9)

三元组顺序表类的定义

```
template<class ElemType>
class TriSparseMatrix{
protected:
// 稀疏矩阵三元组顺序表的数据成员:
        Triple<ElemType> *triElems;
        int maxSize;
        int rows, cols, num;
```

三元组顺序表类的定义

```
public:
        TriSparseMatrix(int rs = DEFAULT_SIZE, int cs =
DEFAULT_SIZE, int size = DEFAULT_SIZE);
        ~TriSparseMatrix(); // 析构函数
        int GetRows() const;// 返回稀疏矩阵行数
        int GetCols() const; // 返回稀疏矩阵列数
        int GetNum() const; // 返回稀疏矩阵非零元个数
        Status SetElem(int r, int c, const ElemType &v);
        Status GetElem(int r, int c, ElemType &v);
```

三元组顺序表类的定义

```
TriSparseMatrix<ElemType> &operator =(const
        TriSparseMatrix<ElemType>       &copy);
        // 赋值运算符重载
void SimpleTranspose(const TriSparseMatrix<ElemType>
        &source, TriSparseMatrix<ElemType> &dest);
        // 将稀疏矩阵source转置成稀疏矩阵dest的简单算法
void FastTranspose(const TriSparseMatrix<ElemType>
        &source, TriSparseMatrix<ElemType> &dest);
        // 将稀疏矩阵source转置成稀疏矩阵dest的快速算法
.......
};
```

矩阵转置

$$M=\begin{bmatrix}15&0&0&22&0&-5\\0&11&3&0&0&0\\0&0&0&6&0&0\\0&0&0&0&0&0\\91&0&0&0&0&0\\0&7&0&0&0&0\\0&0&28&0&0&0\end{bmatrix}$$

(a) 矩阵 M

$$N=\begin{bmatrix}15&0&0&0&91&0&0\\0&11&0&0&0&7&0\\0&3&0&0&0&0&28\\22&0&6&0&0&0&0\\0&0&0&0&0&0&0\\-5&0&0&0&0&0&0\end{bmatrix}$$

(b)M 的转置矩阵 N

图 5-7 稀疏矩阵

	row	col	value
triElems[0]	0	0	15
[1]	0	3	22
[2]	0	5	-5
[3]	1	1	11
[4]	1	2	3
[5]	2	3	6
[6]	4	0	91
[7]	5	1	7
[8]	6	2	28

(a.rows=7, a.cols=6, a.num=9)

	row	col	value
triElems[0]	0	0	15
[1]	0	4	91
[2]	1	1	11
[3]	1	5	7
[4]	2	1	3
[5]	2	6	28
[6]	3	0	22
[7]	3	2	6
[8]	5	0	5

(b.rows=6, b.cols=7, b.num=9)

矩阵转置

一个简单的方法就是把三元组顺序表a中各三元组的row与col的内容互换，然后再按照新的row中的行号从小到大重新排放。

该转置算法实际上就是按照M的列序来进行转置，即把矩阵M的第0列的所有元素找出来，并转置，把结果存到矩阵N的第0行；再把矩阵M的第1列的所有元素找出来，并转置，把结果存到矩阵N的第1行；依次执行之，直到所有元素转换完毕为止。

为了找到M的某一列中所有的非零元素，需要对M的a.smarray从第0行起整个扫描一遍，共需扫描a.cols遍。由于a.smarray是以行序为主序来存放非零元素的，由此得到的恰是b.smarray应有的顺序。

矩阵转置

```
template<class ElemType>  void TriSparseMatrix<ElemType>::
SimpleTranspose(TriSparseMatrix<ElemType> &b){
    b.rows = cols;  b.cols = rows;  b.num = num; b.maxSize = maxSize;
    delete []b.triElems;
    b.triElems = new Triple<ElemType>[b.maxSize];
    if (b.num > 0){
    int i = 0;       // 稀疏矩阵b的下一个三元组的存放位置
    for (int col = 0; col < cols; col++)
      for (int j = 0; j < num; j++)   // 查找原矩阵中第col列的三元组
        if (triElems[j].col == col) {
            b.triElems[i].row = triElems[j].col;
            b.triElems[i].col = triElems[j].row;
            b.triElems[i].value = triElems[j].value;     i++;
        }
   }
}
```

矩阵转置

矩阵转置的快速算法：

按照a.smarray中三元组的次序进行转置，并将转置后的三元组置b中恰当的位置。

如果能预先确定矩阵M中每一列（即N中每一行）的头一个非零元在b.smarray中应有的位置，那么在对a.smarray中的三元组依次作转置时，便可直接放到b.smarray中恰当的位置上去。

为了确定这些位置，在转置前，应先求得M的每一列中非零元的个数，进而就可求得M的每一列的头一个非零元在b.smarray中应有的位置。

矩阵转置

```
template<class ElemType>void TriSparseMatrix<ElemType>::
FastTranspose(TriSparseMatrix<ElemType> &b){
    b.rows = cols;b.cols = rows;  b.num = num;    b.maxSize = maxSize;
    delete []b.triElems;
    b.triElems = new Triple<ElemType>[b.maxSize];
    int *cNum = new int[cols + 1]; // 存放原矩阵中每一列的非零元个数
    int *cPos = new int[cols + 1];
    int col, i;
    if (b.num > 0)          {
        for (col = 0; col < cols; col++) cNum[col] = 0;       // 初始化cNum
            for (i = 0; i < a.num; i++)
            ++cNum[triElems[i].col];// 统计原矩阵中每一列的非零元个数
```

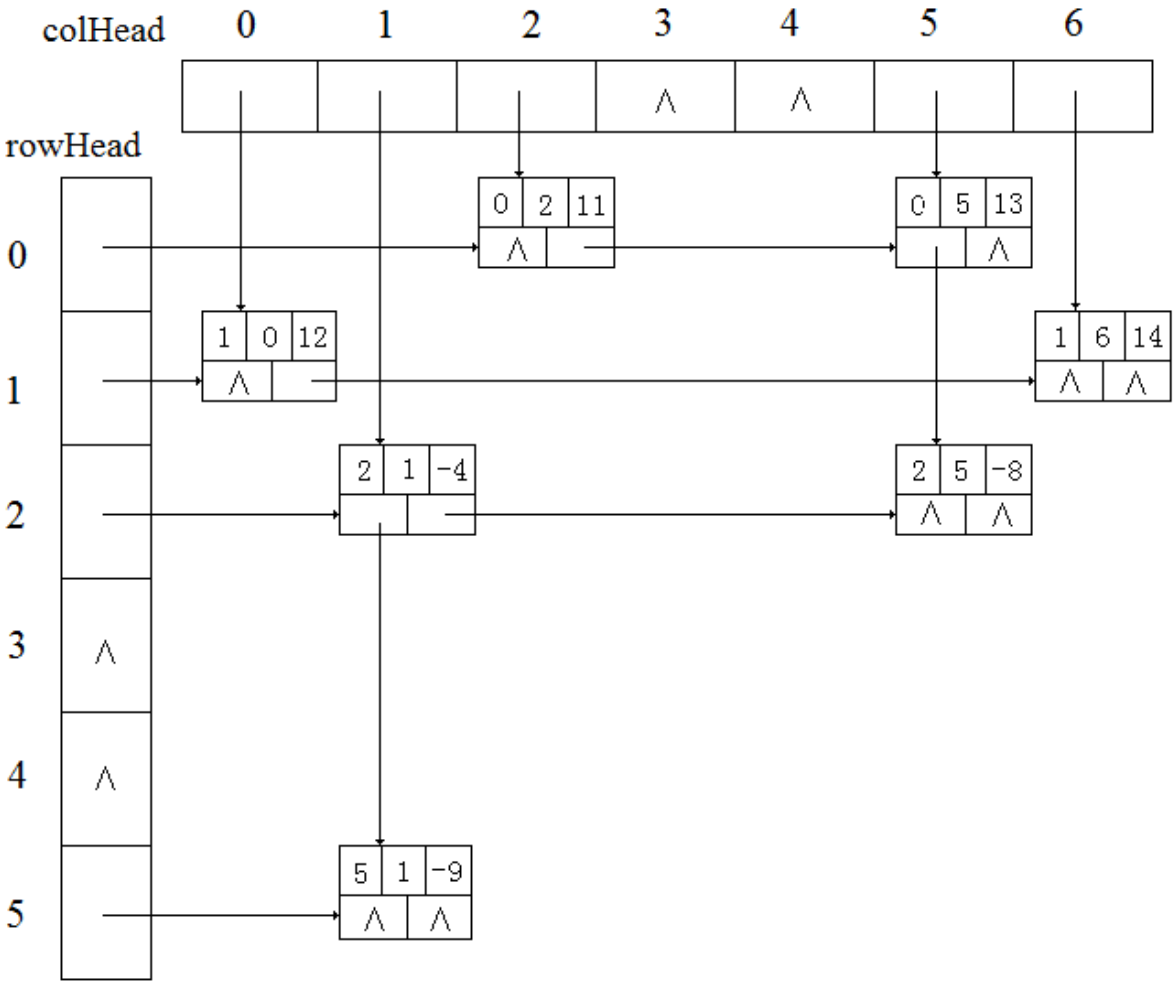
矩阵转置

```
    cPos[0] = 0;   // 第一列的第一个非零元在b存储的起始位置
    for (col = 1; col < cols; col++)
        cPos[col] = cPos[col - 1] + cNum[col - 1];
    for (i = 0; i < num; i++){          // 循环遍历原矩阵中的三元组
        int j = cPos[triElems[i].col];
        b.triElems[j].row = triElems[i].col;
        b.triElems[j].col = triElems[i].row;
        b.triElems[j].value = triElems[i].value;
        ++cPos[triElems[i].col];
     }
   }
   delete []cNum;
   delete []cPos;
}
```

十字链表

$$\begin{bmatrix} 0 & 0 & 11 & 0 & 0 & 13 & 0 \\ 12 & 0 & 0 & 0 & 0 & 0 & 14 \\ 0 & -4 & 0 & 0 & 0 & -8 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -9 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(a) 稀疏矩阵



十字链表中非零元素结点类

```
#include "triple.h"                                    // 三元组类
// 十字链表非零元素结点类
template<class ElemType>
struct CrossNode   {
// 数据成员:
        Triple<ElemType> triElem;          // 三元组
        CrossNode<ElemType> *right, *down;
// 构造函数:
        CrossNode();
        CrossNode(const Triple<ElemType> &e,
         CrossNode<ElemType> *rLink = NULL,
              CrossNode<ElemType> *dLink = NULL);
};
```

十字链表类的定义

```
template<class ElemType>  class CrossList  {
protected:
    CrossNode<ElemType> **rowHead, **colHead;// 行列链表头数组
    int rows, cols, num;   // 稀疏矩阵的行数,列数及非零元个数
public:
    CrossList(int rs = DEFAULT_SIZE, int cs = DEFAULT_SIZE);
                        // 构造一个rs行cs列的空稀疏矩阵
     ~CrossList();                                          // 析构函数
    void Clear();                                  // 清空稀疏矩阵
    int GetRows() const { return rows; };   // 返回稀疏矩阵行数
    int GetCols() const { return cols; };      // 返回稀疏矩阵列数
    int GetNum() const { return num; };     // 返回稀疏矩阵非零元个数
```

十字链表类的定义

```
    Status SetElem(int r, int c, const ElemType &v);
        // 设置指定位置的元素值
    Status GetElem(int r, int c, ElemType &v);// 取指定位置的元素值
    CrossList(const CrossList<ElemType> &b);    // 复制构造函数
    CrossList<ElemType> &operator =(const CrossList<ElemType> &b);
                    // 重载赋值运算符
    CrossList<ElemType> operator +(const CrossList<ElemType> &b);
                    // 重载加法运算符
};
```

修改指定位置的元素值

```
template <class ElemType>
Status CrossList<ElemType>::SetElem(int r, int c, const ElemType &v) {
    if (r >= rows || c >= cols || r < 0 || c < 0)  return RANGE_ERROR;
    CrossNode<ElemType> *pre, *p;
    if (v == 0) {
        pre=NULL; p=rowHead[r];
        while (p != NULL && p->triElem.col < c) {
            pre=p;          p=p->right;
        }
       if (p != NULL && p->triElem.col == c) {
          if (rowHead[r] == p)     rowHead[r]=p->right;
          else                pre->right=p->right;
```

修改指定位置的元素值

```
        if (colHead[c] == p)     colHead[c]=p->down;
        else {
            pre=colHead[c];
            while (pre->down != p)       pre=pre->down;
            pre->down=p->down;
        }
        delete p;            num--;
    }
}
else { // 把第r行、第c列的元素值修改为非零元素
    pre=NULL; p=rowHead[r];
    while (p != NULL && p->triElem.col < c)          {
        pre=p;p=p->right;
    }
```

修改指定位置的元素值

```
        if (p != NULL && p->triElem.col == c)      p->triElem.value == v;
        else {      // 原结点为0元素，则需要插入结点
            Triple<ElemType> e(r, c, v);
            CrossNode<ElemType>*ePtr=new CrossNode<ElemType>(e);
            if (rowHead[r] == p)    rowHead[r]=ePtr;
            else        pre->right=ePtr;
            ePtr->right=p;  pre=NULL;   p=colHead[c];
            while (p != NULL && p->triElem.row < r)    {
                pre=p;   p=p->down;
            }
            if (colHead[c] == p)     colHead[c]=ePtr;
            else        pre->down=ePtr;
            ePtr->down=p;        num++;
        }
    }
    return SUCCESS;
}
```

取指定位置的元素值

```
template <class ElemType>
Status CrossList<ElemType>::GetElem(int r, int c, ElemType &v)  {
        if (r >= rows || c >= cols || r < 0 || c < 0)
                return RANGE_ERROR;
        CrossNode<ElemType> *p;
        for (p=rowHead[r]; p != NULL && p->triElem.col < c; p=p->right);
        if (p != NULL && p->triElem.col == c)          // 找到三元组
                v=p->triElem.value;
        else     // 未找到三元组
                v=0;
        return SUCCESS;
}
```

加法运算

```
template <class ElemType>
CrossList<ElemType> CrossList<ElemType>::operator +(const
CrossList<ElemType> &b)  {
    if (rows != b.rows || cols != b.cols) throw Error("行数或列数不相等!");
    CrossList<ElemType> temp(b.rows, b.cols);
    ElemType v;    CrossNode<ElemType> *p, *q;
    for (int i=0; i < rows; i++) {
        p=rowHead[i]; q=b.rowHead[i];
        while (p != NULL && q != NULL)
          if (p->triElem.col < q->triElem.col) {
            temp.SetElem(p->triElem.row, p->triElem.col, p->triElem.value);
            p=p->right;
          }
         else  if (p->triElem.col > q->triElem.col) {
            temp.SetElem(q->triElem.row, q->triElem.col,
            q->triElem.value);    q=q->right;
          }
```

加法运算

```
        else  {
            v=p->triElem.value + q->triElem.value;
            if (v != 0) temp.SetElem(q->triElem.row, q->triElem.col, v);
            p=p->right;    q=q->right;
        }
      while (p != NULL) {
          temp.SetElem(p->triElem.row, p->triElem.col, p->triElem.value);
          p=p->right;
      }
      while (q != NULL) {
          temp.SetElem(q->triElem.row, q->triElem.col, q->triElem.value);
          q=q->right;
      }
   }
   return temp;
}
```

5.4 广义表

广义表LS是由n≥0个表元素 α_1，α_2，…，α_n 组成的有限序列，其中表元素 α_i（1≤i≤n）或者是一个数据元素（可称为单元素或原子），或者是一个表(称为子表)。记作

$$LS = (\alpha_1, \alpha_2, \ldots, \alpha_n)$$

其中LS是表名，表的长度为n。长度为0的广义表为空表。一般用大写字母表示表名，用小写字母表示数据元素。如果n≥1，则称 α_1 为广义表LS的表头(head)，称(α_2，…，α_n)为广义表LS的表尾(tail)。

5.4 广义表

广义表的定义是递归的，因为在表的描述中又用到了表。

广义表的例子：

(1) A = ( ) 空表，它的长度为零。

(2) B = (e)

(3) C = (a , (b , c , d)) 长度为2，两个表元素分别为单元素a和子表(b , c , d)。 head(C) = a , tail(C) = ((b , c , d))。

(4) D = (A , B , C) 长度为3，三个表元素都是子表。head (D) = A , tail (D) = (B , C)。

(5) E = (a , E) 长度为2，E是一个递归表，它对应于无限表E = (a , (a , (a ,...))))。 head (E) = a , tail (E) = (E)。

广义表结点类的定义

广义表采用3个域的结点结构表示。

第一个域是tag域，tag =0，表示此结点是专用表头结点；tag=1，表示是单元素结点；tag=2，表示是子表结点。

第二个域ref/data/hlink由其tag域确定，当tag=0时，ref中存放引用计数；当tag=1时，即为data域，此处假设存放字符型数据；当tag=2时，即为hlink域，用来存放指向子表表头的指针。

第三个域是tlink域，当tag=0时，该指针域存放指向该表表头元素的指针；当tag≠0时，用来存放指向同一层下一个结点的指针。

tag=0/1/2	ref/data/hlink	tlink

广义表结点类的定义

```
enum GenListNodeType {HEAD, ATOM, LIST};
template<class ElemType>struct GenListNode  {
    GenListNodeType tag;
     // 标志域,HEAD(0):头结点, ATOM(1):原子结构, LIST(2):表结点
    GenListNode<ElemType> *tLink;
    union{
            int ref;   // tag=HEAD,头结点,存放引用数
            ElemType atom;            // tag=ATOM,存放原子结点的数据域
            GenListNode<ElemType> *hLink;
     };
     GenListNode(GenListNodeType tg = HEAD,
          GenListNode<ElemType> *next = NULL);
          // 由标志tg和指针next构造引用数法广义表结点
};
```

广义表的存储结构

A = ( )

B = (e)

C = (a , (b , c , d))

D = (A , B , C)

E = (a , E)

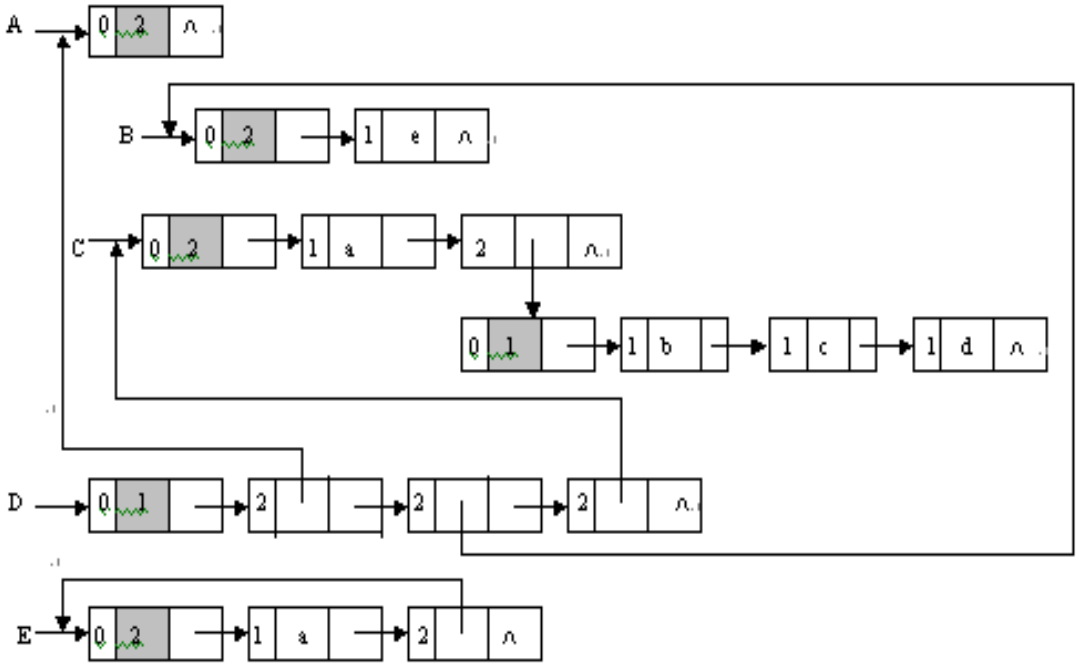


图 5-13　广义表的存储结构示例

广义表类的定义

```
template<class ElemType>
class GenList  {
protected:
    GenListNode<ElemType> *head;
    void ShowHelp(GenListNode<ElemType> *hd) const;
    int DepthHelp(const GenListNode<ElemType> *hd);
    void ClearHelp(GenListNode<ElemType> *hd);
    void CopyHelp(const GenListNode<ElemType> *sourceHead,
                  GenListNode<ElemType> *&destHead);
    static void CreateHelp(GenListNode<ElemType> *&first);
```

广义表类的定义

```
public:
    GenList();
    GenList(GenListNode<ElemType> *hd);
    ~GenList();
    GenListNode<ElemType> *First() const;
    GenListNode<ElemType> *Next(GenListNode<ElemType> *p) const;
    bool IsEmpty() const;
    void Insert(const ElemType &e);
    void Insert(GenList<ElemType> &subList);
    Status Delete(int i);
    int GetDepth();
    int GetLength();
    void Input(void);
    void Show(void);
};
```

部分成员函数的实现

```
template <class ElemType>
int GenList<ElemType>::DepthHelp(const GenListNode<ElemType> *hd)
// 操作结果：返回以hd为表头的广义表的深度
{
    if (hd->tLink == NULL) return 1;
    int subMaxDepth = 0;                                    // 子表最大深度
    for (GenListNode<ElemType> *p = hd->tLink; p != NULL; p = p->tLink){
        if (p->tag == LIST) {       // 子表
                int curSubDepth = DepthHelp(p->hLink); // 子表深度
                if (subMaxDepth < curSubDepth) subMaxDepth = curSubDepth;
        }
    }
    return subMaxDepth + 1;
}
```

部分成员函数的实现

```
template <class ElemType>
void GenList<ElemType>::ClearHelp(GenListNode<ElemType> *hd) {
// 操作结果：释放以hd为表头的广义表结构
        hd->ref--;
        if (hd->ref == 0)      {
            GenListNode<ElemType> *pre = hd, *p;
            for (p = hd->tLink; p != NULL; p = p->tLink){
                        delete pre;          pre = p;
                        if (p->tag == LIST)          ClearHelp(p->hLink);      // 释放子表
            }
            delete pre;
        }
}
```

部分成员函数的实现

```
template <class ElemType>
void GenList<ElemType>::CopyHelp(const GenListNode<ElemType> *sourceHea
        GenListNode<ElemType> *&destHead)       {
        destHead = new GenListNode<ElemType>(HEAD);
        GenListNode<ElemType> *destPtr = destHead;
        destHead->ref = 1;
        for (GenListNode<ElemType> *p = sourceHead->tLink; p != NULL;
            p = p->tLink)      {
            destPtr = destPtr->tLink = new GenListNode<ElemType>(p->tag);
            if (p->tag == LIST)          CopyHelp(p->hLink, destPtr->hLink);
            else       destPtr->atom = p->atom;
        }
}
```

n元多项式的表示

如果用线性表表示n元多项式，会有两个问题：

- 一是对于变元数较少的项，也要按n个变元分配存储空间，将造成浪费；
- 二是对n值不同的多项式，存储线性表时结点大小也不同，存储管理也就不便。

为了避免上述问题，我们将用广义表表示n元多项式。

例如三元多项式：

$$P(x, y, z)=x^{10}y^3z^2+2x^8y^3z^2+3x^8y^2z^2+x^4y^4z+6x^3y^4z+2yz$$

把P(x , y , z)重新写作：

$$P(x, y, z)=((x^{10}+2x^8)y^3+3x^8y^2)z^2+((x^4+6x^3)y^4+2y)z$$

n元多项式的表示

P(x , y , z)可以看作是z的多项式，即Az^2+Bz。A和B本身又是(x , y)的二元多项式，如A(x , y) = Cy^3+Dy^2，C和D又为x的一元多项式。继续这样分析，可知P中的每一个子多项式由一个变元和若干个系数、指数偶对组成，其中每个系数本身可以又是一个多项式。

上述三元多项式可以表示为下面的广义表P：

P = ((A, 2), (B, 1))

P的变元为z，其中：

A = ((C, 3), (D, 2))　　B = ((E, 4), (2,1))

A、B的变元为y

C = ((1, 10), (2, 8))　　D = ((3, 8))

E = ((1, 4), (6, 3))

C、D、E的变元为x

n元多项式的表示

每个结点有4个域，第1个域为tag，第二个域称为nodename，依据tag的不同取值，nodename亦不同,如图5-14所示。第一种tag 为 var，表示该结点是链表的头结点，则nodename为vble域，存变元名，且exp = 0；第二种tag 为 ptr，则系数本身又是一个多项式，nodename为hlink域，hlink中存放指向那个多项式子链表的指针，exp为变元的指数；第三种是tag 为 num，nodename为coef，表示系数是一个实数，exp为变元的指数值。

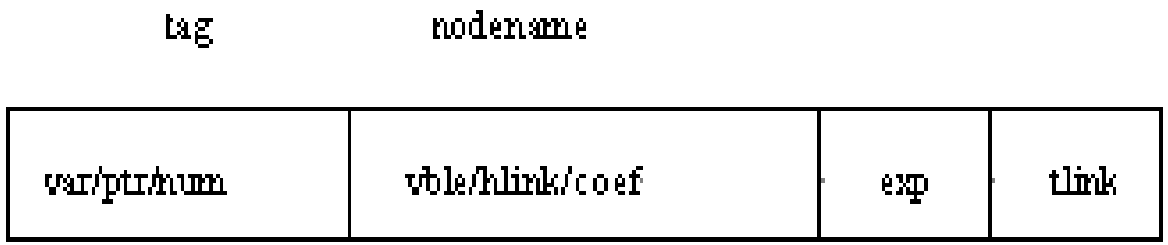


图 5-14 多项式链表的结点结构

polynode类的定义

```
enum triple { var , ptr , num };
class polynode{//多项式结点类定义
        polynode * tlink ;          //同一层下一结点指针
        int exp ;                   //指数
        triple tag ;
        union       {
            char vble ;
            polynode * hlink ;
            int coef ;
        };
};
```

n元多项式的表示

这里给出一个简单的多项式

$$q(x,\ y)=y^3+3x^2y^2+1.5y+x^3-4$$

设表头指针q指示的是一个表头结点，它标明以它打头的一层链表是基于变元y的，其tlink指示了多项式链表的第一项。

对基于y的链表中第二个结点的hlink所指示的链表，表头结点中tag=var，该链表是基于变量x的，tlink指向的结点的tag为num。

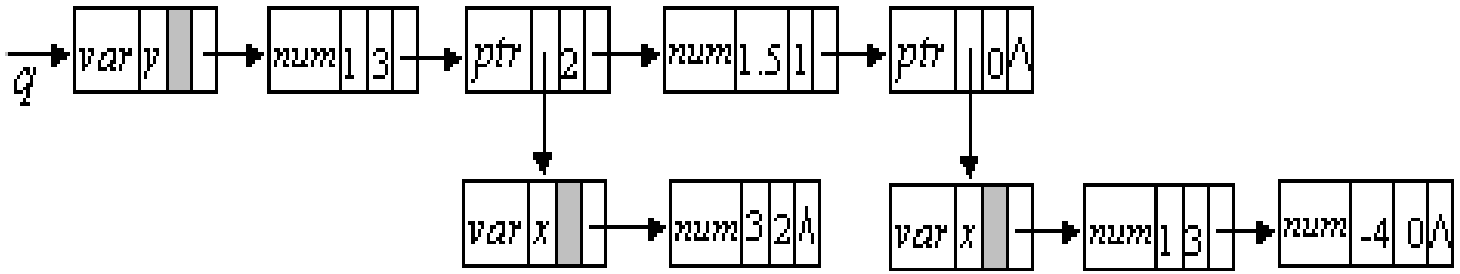


图5-15 Q(x, y)的链表表示

n元多项式的表示

$P(x, y, z)=((x^{10}+2x^8)y^3+3x^8y^2)z^2+((x^4+6x^3)y^4+2y)z$的链表表示，由于tag在该链表中是很明了的，因此在图中省略了tag域，这样该链表表示更清晰。

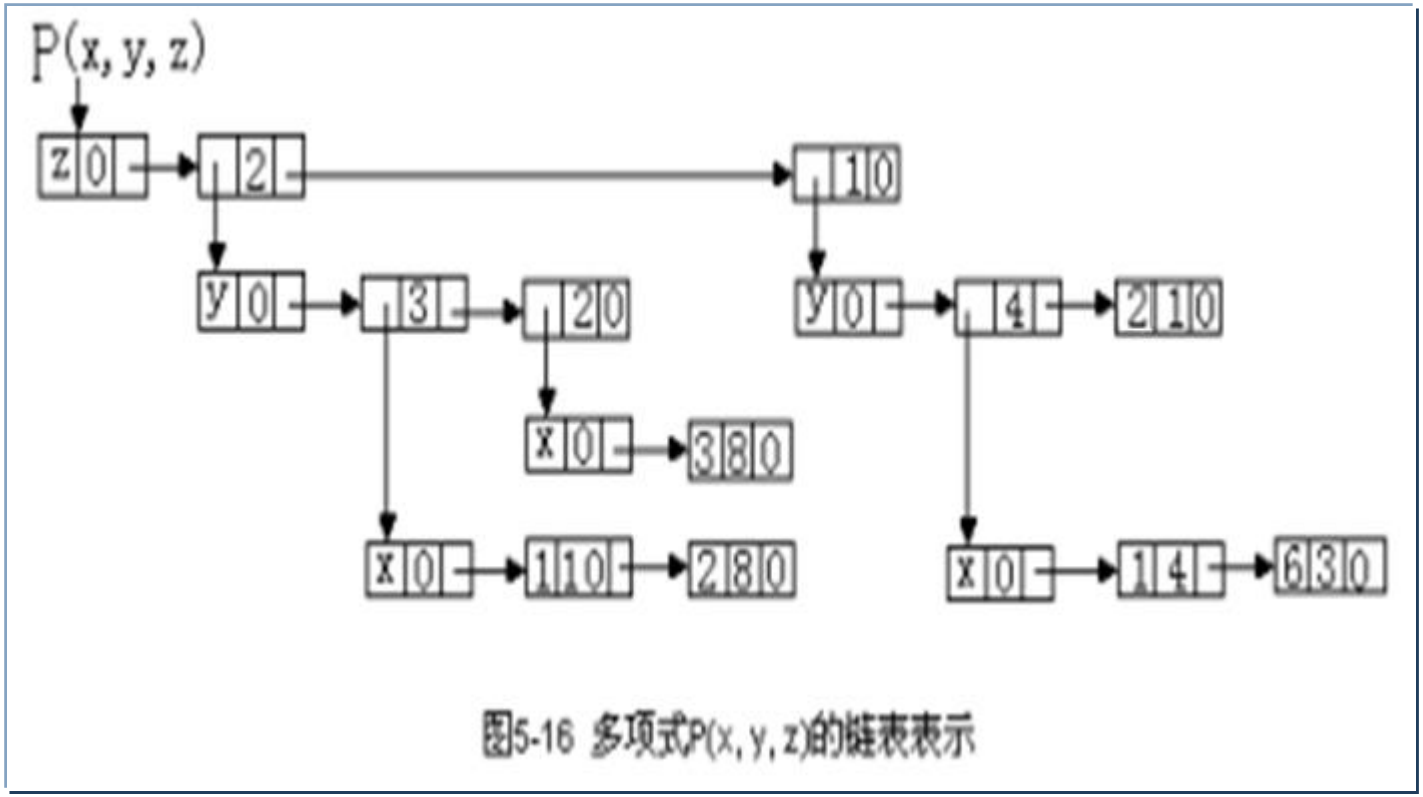


图5-16 多项式P(x, y, z)的链表表示


祝你成功！