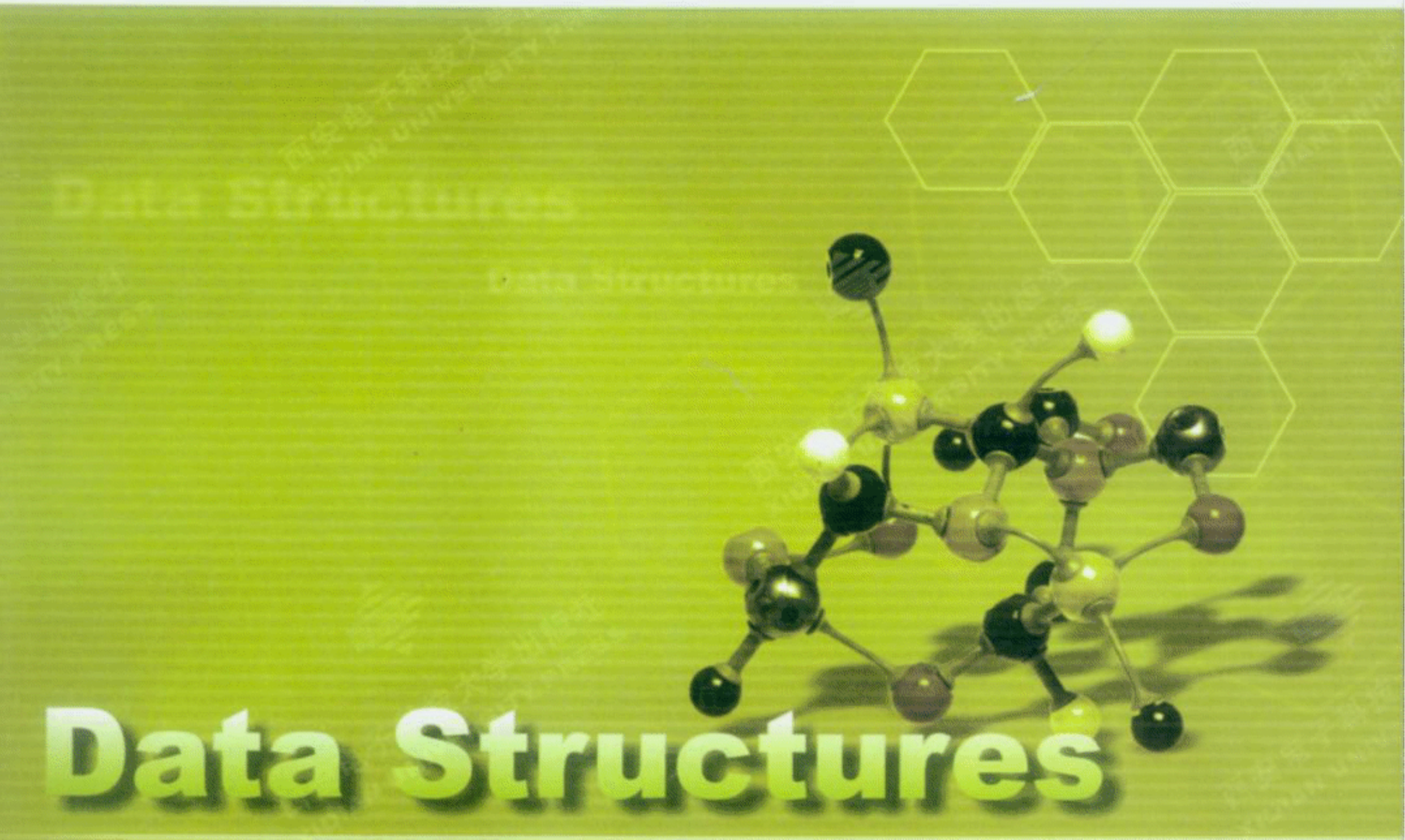
《数据结构》
算法实现及解析

——配合严蔚敏、吴伟民编著的《数据结构》（C语言版）

（第二版）

■ 高一凡 编著



西安电子科技大学出版社

http://www.xduph.com

《数据结构》算法实现及解析(第二版)

——配合严蔚敏、吴伟民编著的《数据结构》(C 语言版)

高一凡　编著

西安电子科技大学出版社

2004

内 容 简 介

本书是在第一版的基础上修订而成的。

本书为清华大学出版社出版、由严蔚敏和吴伟民编著的《数据结构》(C 语言版)(以下简称教科书)的学习辅导书。主要内容包括：教科书中的每一种数据存储结构的图示；教科书中每一种存储结构的基本操作函数及调用这些基本操作的主程序和程序运行结果；教科书中几乎每一种算法的实现。对于教科书中一些较复杂的算法，本书提供了详细的解析。有些在教科书中一带而过的存储结构(如第 2 章的静态链表和第 6 章的二叉树的三叉链表)，本书也提供了完整的基本操作函数及主程序和程序运行结果。本书配有光盘，光盘中包括书中所有程序及用标准 C 语言改写的程序。所有程序均在计算机上运行通过。

本书适用于使用教科书的大中专学生和自学者。书中的基本操作函数也可供从事计算机工程与应用工作的科技人员参考和采用。

图书在版编目（CIP）数据

《数据结构》算法实现及解析 / 高一凡编著. —2 版.

—西安：西安电子科技大学出版社，2004.10

ISBN 7–5606–1176–1

Ⅰ. 数… Ⅱ. 高… Ⅲ. 数据结构 – 算法分析 – 高等学校 – 教学参考资料

Ⅳ. TP311.12

中国版本图书馆 CIP 数据核字（2004）第 100807 号

策　　划　马武装

责任编辑　马武装

出版发行　西安电子科技大学出版社（西安市太白南路 2 号）

电　　话　(029)88242885　88201467　邮　编　710071

http://www.xduph.com　　E-mail: xdupfxb@pub.xaonline.com

经　　销　新华书店

印　　刷　西安文化彩印厂

版　　次　2002 年 10 月第 1 版　2004 年 10 月第 2 版　2004 年 10 月第 3 次印刷

开　　本　787 毫米× 1092 毫米　1/16　印张 29.75

字　　数　707 千字

印　　数　8001～12 000 册

定　　价　33.00(含光盘)元

ISBN 7 – 5606 – 1176 – 1 / TP· 0608

XDUP 1447012 – 3

第一版前言

“数据结构”并非一门纯数学课程。它要求学生能根据所学的“数据结构”理论完成较复杂的程序设计。而程序设计能力的提高有个学习、观摩、借鉴和实践的过程。

学生在学习“数据结构”课程时，虽然已学过 C 语言，但仅是初学，并不精通。对于抽象的数据类型、动态分配存储空间等概念，在理解上还是有一定困难的。如何理解数据存储结构，消化算法，将算法转化成 C 语言的函数并能编写出运行该函数的主程序，往往是摆在他们面前的一道难关。

作者多次讲授“数据结构”课，所用教科书为清华大学出版社出版的严蔚敏、吴伟民编著的《数据结构》(C 语言版)(以下简称为教科书)。该教科书内容较全面，有一定深度。但在叙述一些基本概念和算法时过于精炼，使学生在理解上有一定的困难。作者根据多年的授课经验，编写了教科书中各种数据存储结构示意图，并给出了基本操作函数以及调用这些基本操作的主程序。作者力图把抽象的问题具体化，使学生深刻、透彻地理解教科书中的各种存储结构和基于这种存储结构的算法，掌握数据结构基本操作函数的编写和应用，并在此基础上，能针对具体的工程问题选择甚至创建恰当的数据存储结构，正确应用基本操作函数编程解决之。

本书内容包括：

(1) 教科书中的每一种数据存储结构的图示；

(2) 教科书中每一种存储结构的基本操作函数及调用这些基本操作的主程序和程序运行结果。教科书中几乎每一种算法的实现。对于教科书中一些较复杂的算法，本书提供了详细的解析。有些在教科书中一带而过的存储结构(如第 2 章的静态链表和第 6 章的二叉树的三叉链表)，本书也提供了完整的基本操作函数及主程序和程序运行结果；

(3) 教科书中每一个程序在 Borland C++ Version 3.1 下的运行结果。

本书附带包含书中所有程序的光盘。所有程序(在光盘的\BC 子目录下)都在 Borland C++ Version 3.1 和 Microsoft Visual C++ 6.0 下运行通过。为了方便使用标准 C 语言的读者，光盘中也附有用标准 C 语言改写的所有程序(在光盘的\TC 子目录下)。用标准 C 语言改写的所有程序都在 Turbo C 2.0 下运行通过。要注意的是，光盘中文件的属性是“只读”的。

本书紧密配合教科书，故在章节编排上与教科书保持一致，以便读者对照查找。在引

用教科书中的算法和基本操作时，尽量与其保持一致，不做或少做修改。有些章节内容因易于理解和掌握，故未提供学习指导，只保留了章节目录。

本书曾以讲义的形式印过两次(第一次仅包括前 7 章)，受到学生的欢迎和好评。学生普遍反映本书对于理解教科书内容很有帮助，有的学生还建议正式出版。正是学生对本书的认可给了我极大的鼓励，谨在此对他们表示深深的感谢。

作者对于学习方法的建议：

对于每一种数据类型，要注重主要结构的基本操作。如第 2 章，要注重顺序表和单链表的基本操作。有余力再看次要结构的基本操作。

对于每一个程序，不应仅仅满足于运行出结果，应根据自己的研究目的修改主程序，或在函数中加一些输出语句，以便更好地理解各函数的意义。

各种数据类型的结构都有相通之处，可多做对比，达到融会贯通之效力。

尽管作者尽了最大努力，但限于水平，书中疏漏之处在所难免。希望读者不吝赐教，以便再版时修订。作者 E-mail：gyfan@chd.edu.cn。读者也可通过出版社与我取得联系。

作　者

2002 年 6 月

第二版前言

本书第一版面市以来，受到广大读者的欢迎并被部分院校选为教材。看到自己辛勤劳动的成果得到读者认可，欣慰之余更感到责任重大。本着对读者负责、精益求精的精神，为了更好地发挥本书的作用，对第一版进行了修订。修订内容主要有以下几个方面：

(1) 增加了一些叙述性的文字。如在 2.1 节，不仅说明算法 2.1 和 2.2 在两个程序中实现，而且提示其原因。

(2) 增加了一些图示。如在程序 algo4-3.cpp 中，增加了图 4–11 关于文本结构的示例。目的是克服仅通过文字描述存储结构不直观、不形象的缺点，便于读者掌握程序和算法的精髓。

(3) 增删了一些存储结构及其基本操作。在第 2 章增加了不带头结点的单链表的存储结构及其基本操作。另外，删去了诸如线性表的扩展操作等有些重复的程序。这样，既使得程序简明，又将本书前后内容有机结合，融会贯通。

(4) 修改了一些算法。如在程序 algo3-1.cpp 中加了一个常量，使 algo3-1.cpp 从将十进制的整数转换为八进制的功能扩展到将十进制的整数转换为二～九进制。

(5) 增加了几个算法。如在 7.4.3 节，教科书中描述了克鲁斯卡尔算法的原理，但没有给出相应的算法，程序 algo7-8.cpp 实现了克鲁斯卡尔算法。

(6) 增加了几个算法应用程序。如程序 algo7-9.cpp 将算法 7.16 应用于教科书的图 7.33（全国交通网），显示了算法的实用性，同时也增加了趣味性。

(7) 增加了 1 个可视化的应用程序。将 algo7-9.cpp 改编为 Visual C++ 6.0 下的可视化的程序(在光盘的\VC\shortest 子目录下)。其目的是增加趣味性，提高读者的学习兴趣。同时也说明，“算法与数据结构”不仅能用于传统的 DOS 界面，也能应用于视窗界面。

(8) 订正了第 1 版中出现的错误。如在程序 bo4-1.cpp 的 StrInsert()函数中将语句

```
for(i=MAXSTRLEN;i<=pos;i--)
  S[i]=S[i-T[0]];
```

改为

```
for(i=MAXSTRLEN;i>=pos+T[0];i--)
  S[i]=S[i-T[0]];
```

(9) 修改了一些程序。目的是使程序更加合理、简明。如将程序 bo7-1.cpp 的 DFS()函数中的

```
for(w=FirstAdjVex(G,v1);w>=0;w=NextAdjVex(G,v1,strcpy(w1,GetVex(G,w))))
```

改为

```
for(w=FirstAdjVex(G,G.vexs[v]);w>=0;w=NextAdjVex(G,G.vexs[v],G.vexs[w]))
```

虽然经过作者的精心修订，书中疏漏、错误及可改进之处恐仍在所难免，希望读者不吝赐教，以便有机会时改正。

作　者
2004 年 7 月

于长安大学

目　　录

第 1 章　绪　　论

本书主要由实现基本操作和算法的程序构成。这些程序文件有 6 类：

（1）数据存储结构。文件名第一个字母为 c，以 h 为扩展名。如 c1-1. h 是第 1 章的第 1 种存储结构。

（2）每种存储结构的一组基本操作函数。以 bo（bo 表示基本操作）开头，cpp 为扩展名。如 bo2-4. cpp 是第 2 章第 4 种存储结构的一组基本操作函数。教科书中涉及基本操作的算法也收到基本操作函数中且在函数中注明算法的编号。

（3）调用基本操作的主程序。以 main（main 表示主程序）开头，cpp 为扩展名。如 main3-5. cpp 是调用 bo3-5. cpp 的主程序。

（4）实现算法的程序。以 algo（algo 表示算法）开头，cpp 为扩展名。如 algo6-2. cpp 是实现算法 6. 13 的程序。

（5）不属于基本操作又被多次调用的函数。以 func 开头，cpp 为扩展名。如 func2-3. cpp 中的函数 equal()、comp()、print()等被许多程序调用。为节约篇幅，将这些函数放到 func2-3. cpp 中。

（6）数据文件。以 txt 为扩展名。如第 4 章的 bookinfo. txt 等。

只有以 main、algo 开头的程序文件有主函数 main()，而且是可执行的程序。其它程序都是被调用的程序，通过#include 命令包括在可执行程序中。它们可能被多次调用。为了节省篇幅，只出现在书中第 1 次调用前。

为了便于查找，附录 B 中给出了每个程序文件出现的章节号及其在光盘中的位置。

1.1　什么是数据结构

1.2　基本概念和术语

1.3　抽象数据类型的表示与实现

教科书定义 OK、ERROR 等为函数的结果状态代码，Status 为其类型。我们把这些信息放到头文件 c1. h 中。c1. h 还包含一些常用的头文件，如 string. h、stdio. h 等。为了操作方便，本书几乎每一个程序都把 c1. h 包含进去，也就把这些结果状态代码的定义和头文件包含了进去。对于某一个程序来说，有些结果状态代码和头文件并没有用到，不过这不

影响使用。头文件 c1.h 内容如下：

```
// c1.h (程序名)
#include<string.h>
#include<ctype.h>
#include<malloc.h> // malloc()等
#include<limits.h> // INT_MAX等
#include<stdio.h> // EOF(=^Z或F6),NULL
#include<stdlib.h> // atoi()
#include<io.h> // eof()
#include<math.h> // floor(),ceil(),abs()
#include<process.h> // exit()
#include<iostream.h> // cout,cin
// 函数结果状态代码
#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
// #define OVERFLOW -2 因为在math.h中已定义OVERFLOW的值为3,故去掉此行
typedef int Status; // Status是函数的类型,其值是函数结果状态代码, 如OK等
typedef int Boolean; // Boolean是布尔类型,其值是TRUE或FALSE
```

教科书中的例 1－7 定义了一个三元组的抽象数据类型 Triplet。通过例 1－7，给出了这个三元组的表示方法和所定义的基本操作函数的实现。

例 1－7 实际上是教科书第 2～7 章中各种存储结构的一个范例：定义一种存储结构及建立在这种存储结构上的一组基本操作，并给出基本操作的实现。下面就是例 1－7 在程序中的具体实现：

头文件 c1-1.h 定义了三元组的抽象数据类型 Triplet。它采用动态分配的顺序存储结构(见图 1–1)：

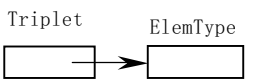


图 1－1　采用动态分配的顺序存储结构

```
// c1-1.h 采用动态分配的顺序存储结构
typedef ElemType *Triplet; // 由InitTriplet分配3个元素存储空间
// Triplet类型是ElemType类型的指针, 存放ElemType类型的地址
```

在 c1-1.h 中我们遇到 ElemType(元素类型)，在后面的章节中我们还会遇到 SElemType(栈元素类型)、QElemType(队列元素类型)和 TElemType(树元素类型)等。在诸如 c1-1.h 这类头文件中，它们是抽象的数据类型。也就是说，还没有确定它们是 C 语言的哪一种具体的类型。

bo1-1.cpp 是有关抽象数据类型 Triplet 和 ElemType 的 8 个基本操作函数。这 8 个函数返回值的类型都是 Status，即函数返回值只能是头文件 c1.h 中定义的 OK、ERROR 等。

```
// bo1-1.cpp 抽象数据类型Triplet和ElemType(由c1-1.h定义)的基本操作(8个)
Status InitTriplet(Triplet &T,ElemType v1,ElemType v2,ElemType v3)
{ // 操作结果: 构造三元组T, 依次置T的3个元素的初值为v1, v2和v3(见图1－2)
```

```
  if(!(T=(ElemType *)malloc(3*sizeof(ElemType))))
    exit(OVERFLOW);
  T[0]=v1,T[1]=v2,T[2]=v3;
  return OK;
}
```

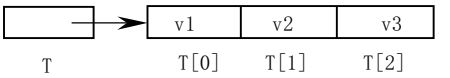


图 1－2　构造三元组 T

```
Status DestroyTriplet(Triplet &T)
{ // 操作结果：三元组T被销毁(见图1－3)
  free(T);
  T=NULL;
  return OK;
}
```

图 1－3　三元组 T 被销毁

```
Status Get(Triplet T,int i,ElemType &e)
{ // 初始条件：三元组T已存在，1≤i≤3。操作结果：用e返回T的第i元的值
  if(i<1||i>3)
    return ERROR;
  e=T[i-1];
  return OK;
}
Status Put(Triplet T,int i,ElemType e)
{ // 初始条件：三元组T已存在，1≤i≤3。操作结果：改变T的第i元的值为e
  if(i<1||i>3)
    return ERROR;
  T[i-1]=e;
  return OK;
}
Status IsAscending(Triplet T)
{ // 初始条件：三元组T已存在。操作结果：如果T的3个元素按升序排列，则返回1；否则返回0
  return(T[0]<=T[1]&&T[1]<=T[2]);
}
Status IsDescending(Triplet T)
{ // 初始条件：三元组T已存在。操作结果：如果T的3个元素按降序排列，则返回1；否则返回0
  return(T[0]>=T[1]&&T[1]>=T[2]);
}
Status Max(Triplet T,ElemType &e)
{ // 初始条件：三元组T已存在。操作结果：用e返回指向T的最大元素的值
  e=T[0]>=T[1]?T[0]>=T[2]?T[0]:T[2]:T[1]>=T[2]?T[1]:T[2];
  return OK;
}
Status Min(Triplet T,ElemType &e)
{ // 初始条件：三元组T已存在。操作结果：用e返回指向T的最小元素的值
  e=T[0]<=T[1]?T[0]<=T[2]?T[0]:T[2]:T[1]<=T[2]?T[1]:T[2];
  return OK;
}
```

main1-1.cpp 是检验 bo1-1.cpp 中的各基本操作函数是否正确的主函数。在 main1-1.cpp 中可根据需要定义抽象数据类型 ElemType 为 int、double 或其它数值型类型(只能是数值类型，因为其中有几个函数是比较大小的)，而不需改变基本操作 bo1-1.cpp，这使得 bo1-1.cpp 的通用性大大增强。在 main1-1.cpp 中，有些基本操作的实参(ElemType 类型)要根据 ElemType 的类型做相应改变，而另一些基本操作的实参(Triplet 类型，总是

ElemType 类型的指针类型)不需改变。

```
// main1-1.cpp 检验基本操作bo1-1.cpp的主函数
#include"c1.h" // 要将程序中所有#include命令所包含的文件拷贝到当前目录下
// 以下2行可根据需要选用一个(且只能选用一个)，而不需改变基本操作bo1-1.cpp
typedef int ElemType; // 定义抽象数据类型ElemType在本程序中为整型
//typedef double ElemType; // 定义抽象数据类型ElemType在本程序中为双精度型
#include"c1-1.h" // 在此命令之前要定义ElemType的类型
#include"bo1-1.cpp" // 在此命令之前要包括c1-1.h文件(因为其中定义了Triplet)
void main()
{
  Triplet T;
  ElemType m;
  Status i;
  i=InitTriplet(T,5,7,9); // 初始化三元组T，其3个元素依次为5，7，9
//i=InitTriplet(T,5.0,7.1,9.3); // 当ElemType为双精度型时，可取代上句
  printf("调用初始化函数后，i=%d(1:成功) T的3个值为",i);
  cout<<T[0]<<','<<T[1]<<','<<T[2]<<endl;
  // 为避免ElemType的类型变化的影响，用cout取代printf()。注意结尾要加endl
  i=Get(T,2,m); // 将三元组T的第2个值赋给m
  if(i==OK) // 调用Get()成功
    cout<<"T的第2个值为"<<m<<endl;
  i=Put(T,2,6); // 将三元组T的第2个值改为6
  if(i==OK) // 调用Put()成功
    cout<<"将T的第2个值改为6后，T的3个值为"<<T[0]<<','<<T[1]<<','<<T[2]<<endl;
  i=IsAscending(T); // 此类函数实参与ElemType的类型无关,当ElemType的类型变化时,实参不需改变
  printf("调用测试升序的函数后，i=%d(0:否 1:是)\n",i);
  i=IsDescending(T);
  printf("调用测试降序的函数后，i=%d(0:否 1:是)\n",i);
  if((i=Max(T,m))==OK) // 先赋值再比较
    cout<<"T中的最大值为"<<m<<endl;
  if((i=Min(T,m))==OK)
    cout<<"T中的最小值为"<<m<<endl;
  DestroyTriplet(T); // 函数也可以不带回返回值
  cout<<"销毁T后，T="<<T<<"(NULL)"<<endl;
}
```

 　程序运行结果：

```
调用初始化函数后，i=1(1:成功) T的3个值为5 7 9
T的第2个值为7
将T的第2个值改为6后，T的3个值为5 6 9
调用测试升序的函数后，i=1(0:否 1:是)
调用测试降序的函数后，i=0(0:否 1:是)
T中的最大值为9
T中的最小值为5
销毁T后，T=0x0000(NULL)
```

main1-1.cpp 是我们运行的第 1 个程序。通过运行 main1-1.cpp，要掌握这样几点：首先，main1-1.cpp 是提供本书程序风格的一个例子。它是教科书中例 1－7 的完整实现。通过它，我们把数据的存储结构、建立于此结构上的基本操作函数以及调用这些函数的主程序结合到一起；其次，抽象的数据类型如何具体化；第三，函数类型 Status 的应用：若函数类型为 Status，它的返回值只能是 OK、ERROR 等；第四，熟悉 C 语言中 malloc 函数的使用。在学习 C 语言时，malloc 函数使用得并不多，但在“数据结构”中它却几乎是使用最多的函数。注意：只有将 c1.h、c1-1.h 和 bo1-1.cpp 三个文件拷贝到 main1-1.cpp 的目录下，才能正确运行 main1-1.cpp。

本书中的 main 主程序是用于检验相应的 bo 程序中的基本操作各个函数程序是否正确的，它往往很长。若读者对某个基本操作函数特别关注，就可以对相应的 main 主程序进行删改，以适应自己的需要，又不需花太多的时间去输入程序。

在 bo1-1.cpp 中，有些基本函数的形参带有“&”，如第一个基本函数的声明：

```
Status InitTriplet(Triplet &T,ElemType v1,ElemType v2,ElemType v3);
```

其中，形参 T 前带有“&”，说明形参 T 是引用类型的。引用类型是 C++语言特有的。引用类型的变量，其值若在函数中发生变化，则变化的值会带回主调函数中。程序 algo1-3.cpp(algo 代表算法)说明了函数中引用类型变量和非引用类型变量的区别：

```
// algo1-3.cpp 变量的引用类型和非引用类型的区别
#include<stdio.h>
void fa(int a) // 在函数中改变a，将不会带回主调函数(主调函数中的a仍是原值)
{
  a++;
  printf("在函数fa中：a=%d\n",a);
}
void fb(int &a) // 由于a为引用类型，在函数中改变a，其值将带回主调函数
{
  a++;
  printf("在函数fb中：a=%d\n",a);
}
void main()
{
  int n=1;
  printf("在主程中，调用函数fa之前：n=%d\n",n);
  fa(n);
  printf("在主程中，调用函数fa之后,调用函数fb之前：n=%d\n",n);
  fb(n);
  printf("在主程中，调用函数fb之后：n=%d\n",n);
}
```

 　程序运行结果：

```
在主程中，调用函数fa之前：n=1
在函数fa中：a=2
在主程中，调用函数fa之后,调用函数fb之前：n=1
在函数fb中：a=2
在主程中，调用函数fb之后：n=2
```

标准 C 语言中没有引用类型，把 C++中含有引用类型的函数转换为标准 C 语言可执行的语句的方法详见附录 A。为了方便使用标准 C 语言的读者，本书所附光盘中 TC\子目录下有本书中所有程序的标准 C 语言版。

bo1-1.cpp 中的 exit()函数是 C 语言的库函数。它的作用是中断程序的运行。程序 algo1-4.cpp 说明了 exit()函数的作用：

```
// algo1-4.cpp 说明exit()函数作用的程序
#include"c1.h"
int a(int i)
{
  if(i==1)
  {
    printf("退出程序的运行\n");
    exit(1);
  }
  return i;
}
void main()
{
  int i;
  printf("请输入i：");
  scanf("%d",&i);
  printf("a(i)=%d\n",a(i));
}
```

 　程序运行结果：

```
请输入i：1↙
退出程序的运行
```

```
请输入i：2↙
a(i)=2
```

当输入为 1 时，执行 exit()语句，退出程序的运行，不执行主程序的 printf()语句；而当输入为 2 时，不执行 exit()语句，返回主程序后，执行 printf()语句。

1.4　算法和算法分析

1.4.1　算法

1.4.2　算法设计的要求

1.4.3　算法效率的度量

同样是计算 1-1/x+1/x*x…，algo1-1.cpp 的语句频度表达式为(1+n)*n/2，它的时间复杂度 T(n)=O(n^2)；而 algo1-2.cpp 的语句频度表达式为 n，它的时间复杂度 T(n)=O(n)。从两个程序的运行结果可以看出：当输入数据一样时，计算结果是一样的，但运行时间的差别很大。在算法正确的前提下，应该选择算法效率高的。

```
// algo1-1.cpp 计算1-1/x+1/x*x…
#include<stdio.h>
#include<sys/timeb.h>
void main()
{
  timeb t1,t2;
  long t;
  double x,sum=1,sum1;
  int i,j,n;
  printf("请输入x n: ");
  scanf("%lf%d",&x,&n);
  ftime(&t1); // 求得当前时间
  for(i=1;i<=n;i++)
  {
    sum1=1;
    for(j=1;j<=i;j++)
      sum1=sum1*(-1.0/x);
    sum+=sum1;
  }
  ftime(&t2); // 求得当前时间
  t=(t2.time-t1.time)*1000+(t2.millitm-t1.millitm); // 计算时间差
  printf("sum=%lf 用时%ld毫秒\n",sum,t);
}
```

程序运行结果(其中用时与计算机的配置有关，带下划线的字符为键盘输入)：

```
请输入x n: 123 10000↙
sum=0.991935 用时5440毫秒
```

```
// algo1-2.cpp 计算1-1/x+1/x*x…的更快捷的算法
#include<stdio.h>
#include<sys/timeb.h>
```

```
void main()
{
  timeb t1,t2;
  long t;
  double x,sum1=1,sum=1;
  int i,n;
  printf("请输入x n: ");
  scanf("%lf%d",&x,&n);
  ftime(&t1); // 求得当前时间
  for(i=1;i<=n;i++)
  {
    sum1*=-1.0/x;
    sum+=sum1;
  }
  ftime(&t2); // 求得当前时间
  t=(t2.time-t1.time)*1000+(t2.millitm-t1.millitm); // 计算时间差
  printf("sum=%lf 用时%ld毫秒\n",sum,t);
}
```

程序运行结果(其中用时与计算机的配置有关):

```
请输入x n: 123 10000↙
sum=0.991935 用时0毫秒
```

第2章 线 性 表

2.1 线性表的类型定义

线性表的基本操作共有 12 个。通过将基本操作有机地组合，还可以对线性表进行较复杂的处理。算法 2.1 和算法 2.2 就是这样的例子。

注意到算法 2.1 和算法 2.2 的形参 La、Lb 和 Lc 的类型是 List(表)，List 并不是稍后将要介绍的具体的线性表存储结构如 SqList 和 LinkList 等，它是抽象的线性表类型。算法 2.1 和算法 2.2 中所涉及的函数都是基本操作，如 GetElem()、ListLength()等，所涉及的变量类型也都是 C 语言的数据类型，不涉及具体的数据存储结构。这样，算法 2.1 和算法 2.2 就可以应用到任何一种具体的线性表存储结构中。

采用 SqList 和 LinkList 两种线性表存储结构实现算法 2.1 的程序分别在 algo2-1.cpp 和 algo2-12.cpp 中，实现算法 2.2 的程序分别在 algo2-2.cpp 和 algo2-13.cpp 中。

2.2 线性表的顺序表示和实现

顺序表存储结构容易实现随机存取线性表的第 i 个数据元素的操作，但在实现插入、删除的操作时要移动大量数据元素，所以，它适用于数据相对稳定的线性表，如职工工资表、学生学籍表等。

c2-1.h 是动态分配的顺序表存储结构，bo2-1.cpp 是基于顺序表的基本操作。由于 C++函数可重载，故去掉 bo2-1.cpp 中算法 2.3 等函数名中表示存储类型的后缀_Sq。c2-1.h 不采用固定数组作为线性表的存储结构，而采用动态分配的存储结构，这样可以合理地利用空间，使长表占用的存储空间多，短表占用的存储空间少。这些可通过 bo2-1.cpp 中基本操作函数 ListInsert()和图 2-4 清楚看出。

```
// c2-1.h 线性表的动态分配顺序存储结构(见图2-1)
#define LIST_INIT_SIZE 10 // 线性表存储空间的初始分配量
#define LIST_INCREMENT 2 // 线性表存储空间的分配增量
struct SqList
{
  ElemType *elem; // 存储空间基址
  int length; // 当前长度
  int listsize; // 当前分配的存储容量(以sizeof(ElemType)为单位)
};
```

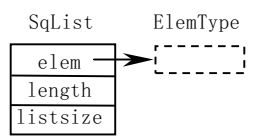


图 2-1 动态分配顺序存储结构

图 2－1 表示结构体 SqList 的存储结构。用一个始于 elem 的箭头表示 elem 是指针类型。指针又是分类型的。用该箭头止于 ElemType 表明 elem 指针的类型。因为此处并不是说明 ElemType，故用虚线表示。

```
// bo2-1.cpp 顺序表示的线性表(存储结构由c2-1.h定义)的基本操作(12个),包括算法2.3,2.4,2.5,2.6
void InitList(SqList &L) // 算法2.3
{ // 操作结果：构造一个空的顺序线性表L(见图2－2)
  L.elem=(ElemType*)malloc(LIST_INIT_SIZE*sizeof(ElemType));
  if(!L.elem)
    exit(OVERFLOW); // 存储分配失败
  L.length=0; // 空表长度为0
  L.listsize=LIST_INIT_SIZE; // 初始存储容量
}
```

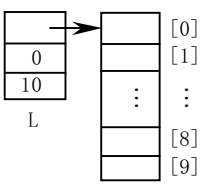


图 2－2　构造一个空的顺序线性表 L

```
void DestroyList(SqList &L)
{ // 初始条件：顺序线性表L已存在。操作结果：销毁顺序线性表L(见图2－3)
  free(L.elem);
  L.elem=NULL;
  L.length=0;
  L.listsize=0;
}
```

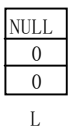
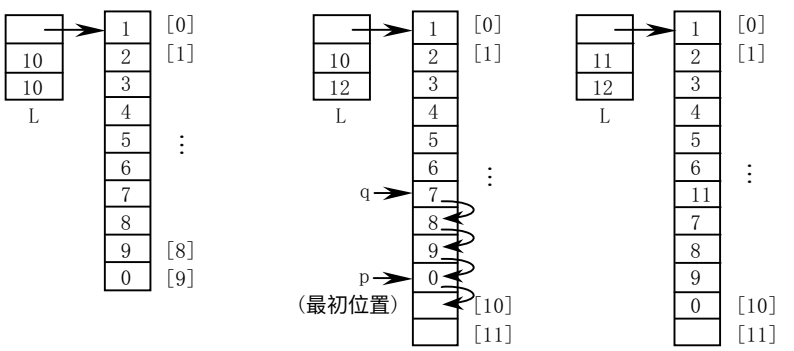


图 2－3　销毁顺序线性表 L

```
void ClearList(SqList &L)
{ // 初始条件：顺序线性表L已存在。操作结果：将L重置为空表
  L.length=0;
}
Status ListEmpty(SqList L)
{ // 初始条件：顺序线性表L已存在。操作结果：若L为空表，则返回TRUE；否则返回FALSE
  if(L.length==0)
    return TRUE;
  else
    return FALSE;
}
int ListLength(SqList L)
{ // 初始条件：顺序线性表L已存在。操作结果：返回L中数据元素的个数
  return L.length;
}
Status GetElem(SqList L,int i,ElemType &e)
{ // 初始条件：顺序线性表L已存在,1≤i≤ListLength(L)。操作结果：用e返回L中第i个数据元素的值
  if(i<1||i>L.length)
    return ERROR;
  e=*(L.elem+i-1);
  return OK;
}
int LocateElem(SqList L,ElemType e,Status(*compare)(ElemType,ElemType))
{ // 初始条件：顺序线性表L已存在，compare()是数据元素判定函数(满足为1，否则为0)
  // 操作结果：返回L中第1个与e满足关系compare()的数据元素的位序。
  //           若这样的数据元素不存在，则返回值为0。算法2.6
  ElemType *p;
  int i=1; // i的初值为第1个元素的位序
  p=L.elem; // p的初值为第1个元素的存储位置
  while(i<=L.length&&!compare(*p++,e))
    ++i;
```

```
  if(i<=L.length)
    return i;
  else
    return 0;
}
Status PriorElem(SqList L,ElemType cur_e,ElemType &pre_e)
{ // 初始条件：顺序线性表L已存在
  // 操作结果：若cur_e是L的数据元素，且不是第一个，则用pre_e返回它的前驱；
  //           否则操作失败，pre_e无定义
  int i=2;
  ElemType *p=L.elem+1;
  while(i<=L.length&&*p!=cur_e)
  {
    p++;
    i++;
  }
  if(i>L.length)
    return INFEASIBLE; // 操作失败
  else
  {
    pre_e=*--p;
    return OK;
  }
}
Status NextElem(SqList L,ElemType cur_e,ElemType &next_e)
{ // 初始条件：顺序线性表L已存在
  // 操作结果：若cur_e是L的数据元素，且不是最后一个，则用next_e返回它的后继；
  //           否则操作失败，next_e无定义
  int i=1;
  ElemType *p=L.elem;
  while(i<L.length&&*p!=cur_e)
  {
    i++;
    p++;
  }
  if(i==L.length)
    return INFEASIBLE; // 操作失败
  else
  {
    next_e=*++p;
    return OK;
  }
}
Status ListInsert(SqList &L,int i,ElemType e) // 算法2.4
{ // 初始条件：顺序线性表L已存在，1≤i≤ListLength(L)+1
  // 操作结果：在L中第i个位置之前插入新的数据元素e，L的长度加1(见图2－4)
  ElemType *newbase,*q,*p;
  if(i<1||i>L.length+1) // i值不合法
    return ERROR;
  if(L.length>=L.listsize) // 当前存储空间已满,增加分配
  {
    if(!(newbase=(ElemType *)realloc(L.elem,(L.listsize+LIST_INCREMENT)*sizeof(ElemType))))
```

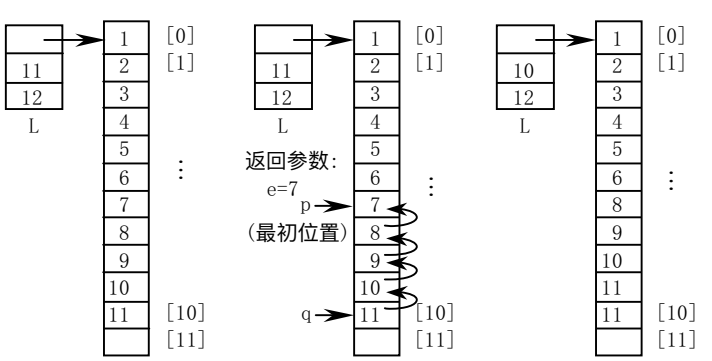
```
      exit(OVERFLOW); // 存储分配失败
    L.elem=newbase; // 新基址
    L.listsize+=LIST_INCREMENT; // 增加存储容量
  }
  q=L.elem+i-1; // q为插入位置
  for(p=L.elem+L.length-1;p>=q;--p) // 插入位置及之后的元素右移
    *(p+1)=*p;
  *q=e; // 插入e
  ++L.length; // 表长增1
  return OK;
}
```



(a) L 调用函数之前的状态　(b) L 增加存储容量、移动元素　(c) L 调用函数之后的状态

图 2－4　调用 ListInsert()示例(i=7, e=11)

```
Status ListDelete(SqList &L,int i,ElemType &e) // 算法2.5
{ // 初始条件: 顺序线性表L已存在, 1≤i≤ListLength(L)
  // 操作结果: 删除L的第i个数据元素, 并用e返回其值, L的长度减1(见图2－5)
  ElemType *p,*q;
  if(i<1||i>L.length) // i值不合法
    return ERROR;
  p=L.elem+i-1; // p为被删除元素的位置
  e=*p; // 被删除元素的值赋给e
  q=L.elem+L.length-1; // 表尾元素的位置
  for(++p;p<=q;++p) // 被删除元素之后的元素左移
    *(p-1)=*p;
```



(a) L 调用函数之前的状态　(b) L 移动元素　(c) L 调用函数之后的状态

图 2－5　调用 ListDelete()示例(i=7)

```
  L.length--; // 表长减1
  return OK;
}
void ListTraverse(SqList L,void(*vi)(ElemType&))
{ // 初始条件：顺序线性表L已存在
  // 操作结果：依次对L的每个数据元素调用函数vi()
  //           vi()的形参加'&'，表明可通过调用vi()改变元素的值
  ElemType *p;
  int i;
  p=L.elem;
  for(i=1;i<=L.length;i++)
    vi(*p++);
  printf("\n");
}
```

bo2-1.cpp 中的 ListTraverse()函数要调用 vi()函数，vi()是 ListTraverse()的形参。因为 vi()不是一个特定的函数，它是满足一定条件的一类函数，所以称其为函数类形参。在函数类形参的声明中指定了 vi()的函数类型，也就是函数返回值的类型(void)。在声明中还指定了 vi()函数形参的个数(1 个)和类型(ElemType 的引用类型)。所有满足条件的函数(返回值为 void 类型，有一个形参，且类型为 ElemType&)，都可以作为 ListTraverse()函数的实参。

LocateElem()函数也要调用一类函数 compare()。由 LocateElem()函数的声明可以看出：要求这类函数具有 2 个形参，其类型均为 ElemType；函数的返回值为 Status 类型。不仅如此，根据 LocateElem()函数的说明(初始条件)，当 compare()函数的 2 个形参满足给定条件时，返回值为 1；否则为 0。只有满足这种条件的函数才能作为替代 compare()函数的实参函数。

main2-1.cpp 是检验 bo2-1.cpp 中的各基本操作函数是否正确的主函数。其中 equal()和 sq()函数满足 LocateElem()函数对函数类形参 compare()的要求，可以作为 LocateElem()的调用函数。print1()函数和 dbl()函数满足 ListTraverse()函数对函数类形参 vi()的要求，可以作为 ListTraverse()的调用函数。其中，print1()函数只是向屏幕输出形参 c，并不改变 c，所以形参不需要是引用类型，但为了和 ListTraverse()函数的要求一致，其形参被定为引用类型。ListTraverse()函数之所以要求 vi()的形参为引用类型，是因为另一个实参函数 dbl()是给形参的值加倍，且要将形参值的改变带回主调函数，故必须是引用类型。

```
// func2-3.cpp 几个常用的函数
Status equal(ElemType c1,ElemType c2)
{ // 判断是否相等的函数
  if(c1==c2)
    return TRUE;
  else
    return FALSE;
}
int comp(ElemType a,ElemType b)
```

```
{ // 根据a<、=或>b，分别返回-1、0或1
  if(a==b)
    return 0;
  else
    return (a-b)/abs(a-b);
}
void print(ElemType c)
{
  printf("%d ",c);
}
void print2(ElemType c)
{
  printf("%c ",c);
}
void print1(ElemType &c)
{
  printf("%d ",c);
}

// main2-1.cpp 检验bo2-1.cpp的主程序
#include"c1.h"
typedef int ElemType;
#include"c2-1.h"
#include"bo2-1.cpp"
#include"func2-3.cpp" // 包括equal()、comp()、print()、print2()和print1()函数
Status sq(ElemType c1,ElemType c2)
{ // 数据元素判定函数(平方关系)，LocateElem()调用的函数
  if(c1==c2*c2)
    return TRUE;
  else
    return FALSE;
}
void dbl(ElemType &c)
{ // ListTraverse()调用的另一函数(元素值加倍)
  c*=2;
}
void main()
{
  SqList L;
  ElemType e,e0;
  Status i;
  int j,k;
  InitList(L);
  printf("初始化L后：L.elem=%u L.length=%d L.listsize=%d\n",L.elem,L.length,L.listsize);
  for(j=1;j<=5;j++)
    i=ListInsert(L,1,j);
  printf("在L的表头依次插入1～5后：*L.elem=");
  for(j=1;j<=5;j++)
    cout<<*(L.elem+j-1)<<' ';
  cout<<endl;
  printf("L.elem=%u L.length=%d L.listsize=%d ",L.elem,L.length,L.listsize);
```

```
i=ListEmpty(L);
printf("L是否空: i=%d(1:是 0:否)\n",i);
ClearList(L);
printf("清空L后: L.elem=%u L.length=%d L.listsize=%d ",L.elem,L.length,L.listsize);
i=ListEmpty(L);
printf("L是否空: i=%d(1:是 0:否)\n",i);
for(j=1;j<=10;j++)
  ListInsert(L,j,j);
printf("在L的表尾依次插入1～10后: *L.elem=");
for(j=1;j<=10;j++)
  cout<<*(L.elem+j-1)<<' ';
cout<<endl;
printf("L.elem=%u L.length=%d L.listsize=%d\n",L.elem,L.length,L.listsize);
ListInsert(L,1,0);
printf("在L的表头插入0后: *L.elem=");
for(j=1;j<=ListLength(L);j++) // ListLength(L)为元素个数
  cout<<*(L.elem+j-1)<<' ';
cout<<endl;
printf("L.elem=%u(有可能改变) L.length=%d(改变) L.listsize=%d(改变)\n",L.elem,L.length,
L.listsize);
GetElem(L,5,e);
printf("第5个元素的值为%d\n",e);
for(j=10;j<=11;j++)
{
  k=LocateElem(L,j,equal);
  if(k) // k不为0, 表明有符合条件的元素, 且其位序为k
    printf("第%d个元素的值为%d\n",k,j);
  else
    printf("没有值为%d的元素\n",j);
}
for(j=3;j<=4;j++)
{
  k=LocateElem(L,j,sq);
  if(k) // k不为0, 表明有符合条件的元素, 且其位序为k
    printf("第%d个元素的值为%d的平方\n",k,j);
  else
    printf("没有值为%d的平方的元素\n",j);
}
for(j=1;j<=2;j++) // 测试头两个数据
{
  GetElem(L,j,e0); // 把第j个数据赋给e0
  i=PriorElem(L,e0,e); // 求e0的前驱
  if(i==INFEASIBLE)
    printf("元素%d无前驱\n",e0);
  else
    printf("元素%d的前驱为%d\n",e0,e);
}
for(j=ListLength(L)-1;j<=ListLength(L);j++) // 最后两个数据
{
  GetElem(L,j,e0); // 把第j个数据赋给e0
  i=NextElem(L,e0,e); // 求e0的后继
```

```
    if(i==INFEASIBLE)
      printf("元素%d无后继\n",e0);
    else
      printf("元素%d的后继为%d\n",e0,e);
  }
  k=ListLength(L); // k为表长
  for(j=k+1;j>=k;j--)
  {
    i=ListDelete(L,j,e); // 删除第j个数据
    if(i==ERROR)
      printf("删除第%d个元素失败\n",j);
    else
      printf("删除第%d个元素成功，其值为%d\n",j,e);
  }
  printf("依次输出L的元素：");
  ListTraverse(L,print1); // 依次对元素调用print1()，输出元素的值
  printf("L的元素值加倍后：");
  ListTraverse(L,dbl); // 依次对元素调用dbl()，元素值乘2
  ListTraverse(L,print1);
  DestroyList(L);
  printf("销毁L后：L.elem=%u L.length=%d L.listsize=%d\n",L.elem,L.length,L.listsize);
}
```

程序运行结果：

```
初始化L后：L.elem=2558 L.length=0 L.listsize=10
在L的表头依次插入1～5后：*L.elem=5 4 3 2 1
L.elem=2558 L.length=5 L.listsize=10 L是否空：i=0(1:是 0:否)
清空L后：L.elem=2558 L.length=0 L.listsize=10 L是否空：i=1(1:是 0:否)
在L的表尾依次插入1～10后：*L.elem=1 2 3 4 5 6 7 8 9 10
L.elem=2558 L.length=10 L.listsize=10
在L的表头插入0后：*L.elem=0 1 2 3 4 5 6 7 8 9 10
L.elem=2582(有可能改变) L.length=11(改变) L.listsize=12(改变)
第5个元素的值为4
第11个元素的值为10
没有值为11的元素
第10个元素的值为3的平方
没有值为4的平方的元素
元素0无前驱
元素1的前驱为0
元素9的后继为10
元素10无后继
删除第12个元素失败
删除第11个元素成功，其值为10
依次输出L的元素：0 1 2 3 4 5 6 7 8 9
L的元素值加倍后：
0 2 4 6 8 10 12 14 16 18
销毁L后：L.elem=0 L.length=0 L.listsize=0
```

以下是采用线性表的动态分配顺序存储结构(c2-1.h)调用几个不是基本操作的算法的程序，它们都包含了 bo2-1.cpp 基本操作函数文件。

```
// algo2-1.cpp 实现算法2.1的程序
#include"c1.h"
typedef int ElemType;
#include"c2-1.h" // 采用线性表的动态分配顺序存储结构
#include"bo2-1.cpp" // 可以使用bo2-1.cpp中的基本操作
#include"func2-3.cpp" // 包括equal()、comp()、print()、print2()和print1()函数
void Union(SqList &La,SqList Lb) // 算法2.1
{ // 将所有在线性表Lb中但不在La中的数据元素插入到La中
  ElemType e;
  int La_len,Lb_len;
  int i;
  La_len=ListLength(La); // 求线性表的长度
  Lb_len=ListLength(Lb);
  for(i=1;i<=Lb_len;i++)
  {
    GetElem(Lb,i,e); // 取Lb中第i个数据元素赋给e
    if(!LocateElem(La,e,equal)) // La中不存在和e相同的元素，则插入之
      ListInsert(La,++La_len,e);
  }
}
void main()
{
  SqList La,Lb;
  int j;
  InitList(La); // 创建空表La。如不成功，则会退出程序的运行
  for(j=1;j<=5;j++) // 在表La中插入5个元素，依次为1、2、3、4、5
    ListInsert(La,j,j);
  printf("La= "); // 输出表La的内容
  ListTraverse(La,print1);
  InitList(Lb); // 创建空表Lb
  for(j=1;j<=5;j++) // 在表Lb中插入5个元素，依次为2、4、6、8、10
    ListInsert(Lb,j,2*j);
  printf("Lb= "); // 输出表Lb的内容
  ListTraverse(Lb,print1);
  Union(La,Lb); // 调用Union()，将Lb中满足条件的元素插入La
  printf("new La= "); // 输出新表La的内容
  ListTraverse(La,print1);
}
```

 　程序运行结果：

```
La= 1 2 3 4 5
Lb= 2 4 6 8 10
new La= 1 2 3 4 5 6 8 10
```

```
// algo2-2.cpp 实现算法2.2的程序
#include"c1.h"
typedef int ElemType;
#include"c2-1.h"
#include"bo2-1.cpp"
#include"func2-3.cpp" // 包括equal()、comp()、print()、print2()和print1()函数
void MergeList(SqList La,SqList Lb,SqList &Lc) // 算法2.2
{ // 已知线性表La和Lb中的数据元素按值非递减排列。
  // 归并La和Lb得到新的线性表Lc，Lc的数据元素也按值非递减排列
  int i=1,j=1,k=0;
  int La_len,Lb_len;
  ElemType ai,bj;
  InitList(Lc); // 创建空表Lc
  La_len=ListLength(La);
  Lb_len=ListLength(Lb);
  while(i<=La_len&&j<=Lb_len) // 表La和表Lb均非空
  {
    GetElem(La,i,ai);
    GetElem(Lb,j,bj);
    if(ai<=bj)
    {
      ListInsert(Lc,++k,ai);
      ++i;
    }
    else
    {
      ListInsert(Lc,++k,bj);
      ++j;
    }
  } // 以下两个while循环只会有一个被执行
  while(i<=La_len) // 表La非空且表Lb空
  {
    GetElem(La,i++,ai);
    ListInsert(Lc,++k,ai);
  }
  while(j<=Lb_len) // 表Lb非空且表La空
  {
    GetElem(Lb,j++,bj);
    ListInsert(Lc,++k,bj);
  }
}
void main()
{
  SqList La,Lb,Lc;
  int j,a[4]={3,5,8,11},b[7]={2,6,8,9,11,15,20};
  InitList(La); // 创建空表La
  for(j=1;j<=4;j++) // 在表La中插入4个元素
    ListInsert(La,j,a[j-1]);
  printf("La= "); // 输出表La的内容
  ListTraverse(La,print1);
  InitList(Lb); // 创建空表Lb
```

```
  for(j=1;j<=7;j++) // 在表Lb中插入7个元素
    ListInsert(Lb,j,b[j-1]);
  printf("Lb= "); // 输出表Lb的内容
  ListTraverse(Lb,print1);
  MergeList(La,Lb,Lc);
  printf("Lc= "); // 输出表Lc的内容
  ListTraverse(Lc,print1);
}
```

程序运行结果：

```
La= 3 5 8 11
Lb= 2 6 8 9 11 15 20
Lc= 2 3 5 6 8 8 9 11 11 15 20
```

```
// algo2-3.cpp 实现算法2.7的程序
#include"c1.h"
typedef int ElemType;
#include"c2-1.h"
#include"bo2-1.cpp"
#include"func2-3.cpp" // 包括equal()、comp()、print()、print2()和print1()函数
void MergeList(SqList La,SqList Lb,SqList &Lc) // 算法2.7
{ // 已知顺序线性表La和Lb的元素按值非递减排列。
  // 归并La和Lb得到新的顺序线性表Lc,Lc的元素也按值非递减排列
  ElemType *pa,*pa_last,*pb,*pb_last,*pc;
  pa=La.elem;
  pb=Lb.elem;
  Lc.listsize=Lc.length=La.length+Lb.length; // 不用InitList()创建空表Lc
  pc=Lc.elem=(ElemType *)malloc(Lc.listsize*sizeof(ElemType));
  if(!Lc.elem) // 存储分配失败
    exit(OVERFLOW);
  pa_last=La.elem+La.length-1;
  pb_last=Lb.elem+Lb.length-1;
  while(pa<=pa_last&&pb<=pb_last) // 表La和表Lb均非空
  { // 归并
    if(*pa<=*pb)
      *pc++=*pa++; // 将pa所指单元的值赋给pc所指单元后，pa和pc分别+1(指向下一个单元)
    else
      *pc++=*pb++; // 将pb所指单元的值赋给pc所指单元后，pa和pc分别+1(指向下一个单元)
  } // 以下两个while循环只会有一个被执行
  while(pa<=pa_last) // 表La非空且表Lb空
    *pc++=*pa++; // 插入La的剩余元素
  while(pb<=pb_last) // 表Lb非空且表La空
    *pc++=*pb++; // 插入Lb的剩余元素
}
void main()
{
  SqList La,Lb,Lc;
  int j;
  InitList(La); // 创建空表La
```

```
  for(j=1;j<=5;j++) // 在表La中插入5个元素，依次为1、2、3、4、5
    ListInsert(La,j,j);
  printf("La= "); // 输出表La的内容
  ListTraverse(La,print1);
  InitList(Lb); // 创建空表Lb
  for(j=1;j<=5;j++) // 在表Lb中插入5个元素，依次为2、4、6、8、10
    ListInsert(Lb,j,2*j);
  printf("Lb= "); // 输出表Lb的内容
  ListTraverse(Lb,print1); // 由按非递减排列的表La、Lb得到按非递减排列的表Lc
  MergeList(La,Lb,Lc);
  printf("Lc= "); // 输出表Lc的内容
  ListTraverse(Lc,print1);
}
```

程序运行结果：

```
La= 1 2 3 4 5
Lb= 2 4 6 8 10
Lc= 1 2 2 3 4 4 5 6 8 10
```

```
// algo2-4.cpp 修改算法2.7的第一个循环语句中的条件语句为开关语句，且当
// *pa=*pb时，只将两者中之一插入Lc。此操作的结果和算法2.1相同
#include"c1.h"
typedef int ElemType;
#include"c2-1.h"
#include"bo2-1.cpp"
#include"func2-3.cpp" // 包括equal()、comp()、print()、print2()和print1()函数
void MergeList(SqList La,SqList Lb,SqList &Lc)
{ // 另一种合并线性表的方法(根据算法2.7下的要求修改算法2.7)，La、Lb和Lc均为按递增排列的表
  ElemType  *pa,*pa_last,*pb,*pb_last,*pc;
  pa=La.elem;
  pb=Lb.elem;
  Lc.listsize=La.length+Lb.length; // 此句与算法2.7不同
  pc=Lc.elem=(ElemType *)malloc(Lc.listsize*sizeof(ElemType));
  if(!Lc.elem)
    exit(OVERFLOW);
  pa_last=La.elem+La.length-1;
  pb_last=Lb.elem+Lb.length-1;
  while(pa<=pa_last&&pb<=pb_last) // 表La和表Lb均非空
    switch(comp(*pa,*pb)) // 此句与算法2.7不同
    {
      case  0: pb++;
      case -1: *pc++=*pa++;
               break;
      case  1: *pc++=*pb++;
    }
  while(pa<=pa_last) // 表La非空且表Lb空
    *pc++=*pa++;
```

```
  while(pb<=pb_last) // 表Lb非空且表La空
    *pc++=*pb++;
  Lc.length=pc-Lc.elem; // 加此句
}
void main()
{
  SqList La,Lb,Lc;
  int j;
  InitList(La); // 创建空表La
  for(j=1;j<=5;j++) // 在表La中插入5个元素，依次为1、2、3、4、5
    ListInsert(La,j,j);
  printf("La= "); // 输出表La的内容
  ListTraverse(La,print1);
  InitList(Lb); // 创建空表Lb
  for(j=1;j<=5;j++) // 在表Lb中插入5个元素，依次为2、4、6、8、10
    ListInsert(Lb,j,2*j);
  printf("Lb= "); // 输出表Lb的内容
  ListTraverse(Lb,print1);
  MergeList(La,Lb,Lc); // 由按递增排列的表La、Lb得到按递增排列的表Lc
  printf("Lc= "); // 输出表Lc的内容
  ListTraverse(Lc,print1);
}
```

程序运行结果：

```
La= 1 2 3 4 5
Lb= 2 4 6 8 10
Lc= 1 2 3 4 5 6 8 10
```

2.3 线性表的链式表示和实现

和顺序表相比，链表存储结构在实现插入、删除的操作时，不需要移动大量数据元素(但不容易实现随机存取线性表的第 i 个数据元素的操作)。所以，链表适用于经常需要进行插入和删除操作的线性表，如飞机航班的乘客表等。

2.3.1 线性链表

```
// c2-2.h 线性表的单链表存储结构(见图2-6)
struct LNode
{
  ElemType data;
  LNode *next;
};
typedef LNode *LinkList; // 另一种定义LinkList的方法
```

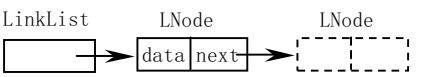


图2-6 线性表的单链表存储结构

图 2－7 是根据 c2-2.h 定义的带有头结点且具有两个结点的线性链表的结构。bo2-2.cpp 是这种带有头结点的线性链表基本操作。

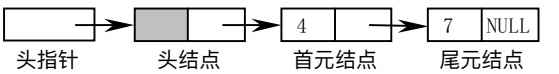


图 2－7　带有头结点且具有两个结点(4,7)的线性链表

```
// bo2-2.cpp 带有头结点的单链表(存储结构由c2-2.h定义)的基本操作(12个),包括算法2.8,2.9,2.10
void InitList(LinkList &L)
{ // 操作结果: 构造一个空的线性表L(见图2－8)
  L=(LinkList)malloc(sizeof(LNode)); // 产生头结点, 并使L指向此头结点
  if(!L) // 存储分配失败
    exit(OVERFLOW);
  L->next=NULL; // 指针域为空
}
```

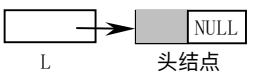


图 2－8　空(仅有头结点)的单链表 L

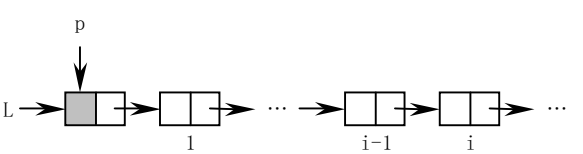
```
void DestroyList(LinkList &L)
{ // 初始条件: 线性表L已存在。操作结果: 销毁线性表L(见图2－9)
  LinkList q;
  while(L)
  {
    q=L->next;
    free(L);
    L=q;
  }
}
```

图 2－9　线性表 L 被销毁

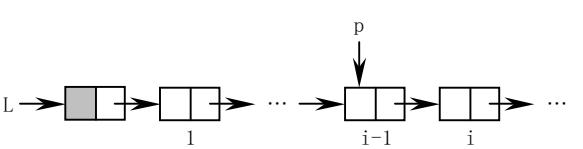
```
void ClearList(LinkList L) // 不改变L
{ // 初始条件: 线性表L已存在。操作结果: 将L重置为空表(见图2－8)
  LinkList p,q;
  p=L->next; // p指向第一个结点
  while(p) // 没到表尾
  {
    q=p->next;
    free(p);
    p=q;
  }
  L->next=NULL; // 头结点指针域为空
}
Status ListEmpty(LinkList L)
{ // 初始条件: 线性表L已存在。操作结果: 若L为空表, 则返回TRUE; 否则返回FALSE
  if(L->next) // 非空
    return FALSE;
  else
    return TRUE;
}
int ListLength(LinkList L)
{ // 初始条件: 线性表L已存在。操作结果: 返回L中数据元素的个数
  int i=0;
  LinkList p=L->next; // p指向第一个结点
  while(p) // 没到表尾
  {
    i++;
    p=p->next;
```

```
  }
  return i;
}
Status GetElem(LinkList L,int i,ElemType &e) // 算法2.8
{ // L为带头结点的单链表的头指针。当第i个元素存在时，其值赋给e并返回OK；否则返回ERROR
  int j=1; // j为计数器
  LinkList p=L->next; // p指向第一个结点
  while(p&&j<i) // 顺指针向后查找，直到p指向第i个元素或p为空
  {
    p=p->next;
    j++;
  }
  if(!p||j>i) // 第i个元素不存在
    return ERROR;
  e=p->data; // 取第i个元素
  return OK;
}
int LocateElem(LinkList L,ElemType e,Status(*compare)(ElemType,ElemType))
{ // 初始条件：线性表L已存在，compare()是数据元素判定函数(满足为1，否则为0)
  // 操作结果：返回L中第1个与e满足关系compare()的数据元素的位序。
  //           若这样的数据元素不存在，则返回值为0
  int i=0;
  LinkList p=L->next;
  while(p)
  {
    i++;
    if(compare(p->data,e)) // 找到这样的数据元素
      return i;
    p=p->next;
  }
  return 0;
}
Status PriorElem(LinkList L,ElemType cur_e,ElemType &pre_e)
{ // 初始条件：线性表L已存在
  // 操作结果：若cur_e是L的数据元素，且不是第一个，则用pre_e返回它的前驱，
  //           返回OK；否则操作失败，pre_e无定义，返回INFEASIBLE
  LinkList q,p=L->next; // p指向第一个结点
  while(p->next) // p所指结点有后继
  {
    q=p->next; // q为p的后继
    if(q->data==cur_e)
    {
      pre_e=p->data;
      return OK;
    }
    p=q; // p向后移
  }
  return INFEASIBLE;
}
Status NextElem(LinkList L,ElemType cur_e,ElemType &next_e)
{ // 初始条件：线性表L已存在
```

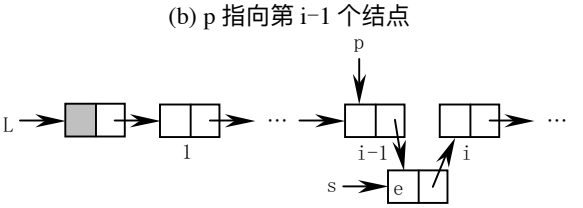
```
  // 操作结果：若cur_e是L的数据元素，且不是最后一个，则用next_e返回它的后继，
  //           返回OK;否则操作失败，next_e无定义，返回INFEASIBLE
  LinkList p=L->next; // p指向第一个结点
  while(p->next) // p所指结点有后继
  {
    if(p->data==cur_e)
    {
      next_e=p->next->data;
      return OK;
    }
    p=p->next;
  }
  return INFEASIBLE;
}
Status ListInsert(LinkList L,int i,ElemType e) // 算法2.9。不改变L
{ // 在带头结点的单链线性表L中第i个位置之前插入元素e(见图2－10)
  int j=0;
  LinkList p=L,s;
  while(p&&j<i-1) // 寻找第i-1个结点
  {
    p=p->next;
    j++;
  }
  if(!p||j>i-1) // i小于1或者大于表长
    return ERROR;
  s=(LinkList)malloc(sizeof(LNode)); // 生成新结点
  s->data=e; // 插入L中
  s->next=p->next;
  p->next=s;
  return OK;
}
```



(a) 初态，p 指向头结点



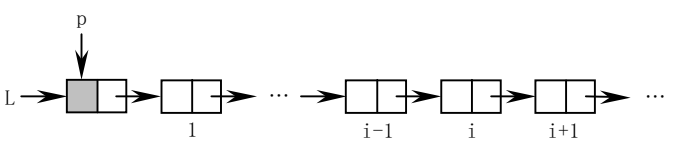
(b) p 指向第 i-1 个结点



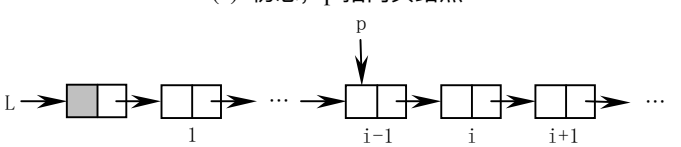
(c) 在第 i 个结点之前插入值为 e 的结点

图 2－10　在链表 L 的第 i 个位置之前插入元素 e

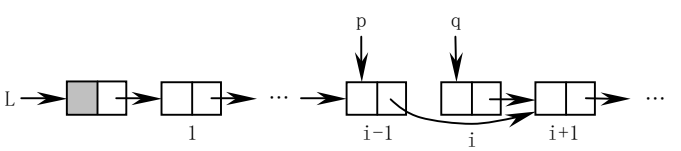
```
Status ListDelete(LinkList L,int i,ElemType &e) // 算法2.10。不改变L
{ // 在带头结点的单链线性表L中，删除第i个元素，并由e返回其值(见图2－11)
  int j=0;
  LinkList p=L,q;
  while(p->next&&j<i-1) // 寻找第i个结点，并令p指向其前驱
  {
    p=p->next;
    j++;
  }
  if(!p->next||j>i-1) // 删除位置不合理
    return ERROR;
  q=p->next; // 删除并释放结点
  p->next=q->next;
  e=q->data;
  free(q);
  return OK;
}
```



(a) 初态，p 指向头结点



(b) p 指向第 i-1 个结点



(c) 从链表上删除第 i 个结点

图 2－11　删除链表 L 的第 i 个结点

```
void ListTraverse(LinkList L,void(*vi)(ElemType))
// vi的形参类型为ElemType，与bo2-1.cpp中相应函数的形参类型ElemType&不同
{ // 初始条件：线性表L已存在。操作结果：依次对L的每个数据元素调用函数vi()
  LinkList p=L->next;
  while(p)
  {
    vi(p->data);
    p=p->next;
  }
  printf("\n");
}
```

main2-2.cpp 是验证基本操作 bo2-2.cpp 的主程序。注意到 bo2-2.cpp 中的

ListTraverse()函数的形参 vi()函数的形参类型是 ElemType，不是引用类型。这与 bo2-1.cpp 中的 ListTraverse()函数不同。因此在主程序中替代 vi()的实参函数 print()的形参类型也是 ElemType。由于 bo2-2.cpp 和 bo2-1.cpp 都是 12 个函数名、操作结果均相同的基本操作函数，仅变量类型、实现过程不同，而验证基本操作的主函数 main2-1.cpp 和 main2-2.cpp 的作用又基本相同，所以程序 main2-2.cpp 和程序 main2-1.cpp 很相像。

```
// main2-2.cpp 检验bo2-2.cpp的主程序(与main2-1.cpp很像)
#include"c1.h"
typedef int ElemType;
#include"c2-2.h" // 与main2-1.cpp不同
#include"bo2-2.cpp" // 与main2-1.cpp不同
#include"func2-3.cpp" // 包括equal()、comp()、print()、print2()和print1()函数
void main()
{ // 除了几个输出语句外，主程序和main2-1.cpp很像
  LinkList L; // 与main2-1.cpp不同
  ElemType e,e0;
  Status i;
  int j,k;
  InitList(L);
  for(j=1;j<=5;j++)
    i=ListInsert(L,1,j);
  printf("在L的表头依次插入1～5后: L=");
  ListTraverse(L,print); // 依次对元素调用print()，输出元素的值
  i=ListEmpty(L);
  printf("L是否空: i=%d(1:是 0:否)\n",i);
  ClearList(L);
  printf("清空L后: L=");
  ListTraverse(L,print);
  i=ListEmpty(L);
  printf("L是否空: i=%d(1:是 0:否)\n",i);
  for(j=1;j<=10;j++)
    ListInsert(L,j,j);
  printf("在L的表尾依次插入1～10后: L=");
  ListTraverse(L,print);
  GetElem(L,5,e);
  printf("第5个元素的值为%d\n",e);
  for(j=0;j<=1;j++)
  {
    k=LocateElem(L,j,equal);
    if(k)
      printf("第%d个元素的值为%d\n",k,j);
    else
      printf("没有值为%d的元素\n",j);
  }
  for(j=1;j<=2;j++) // 测试头两个数据
  {
    GetElem(L,j,e0); // 把第j个数据赋给e0
    i=PriorElem(L,e0,e); // 求e0的前驱
    if(i==INFEASIBLE)
```

```
      printf("元素%d无前驱\n",e0);
    else
      printf("元素%d的前驱为%d\n",e0,e);
  }
  for(j=ListLength(L)-1;j<=ListLength(L);j++) // 最后两个数据
  {
    GetElem(L,j,e0); // 把第j个数据赋给e0
    i=NextElem(L,e0,e); // 求e0的后继
    if(i==INFEASIBLE)
      printf("元素%d无后继\n",e0);
    else
      printf("元素%d的后继为%d\n",e0,e);
  }
  k=ListLength(L); // k为表长
  for(j=k+1;j>=k;j--)
  {
    i=ListDelete(L,j,e); // 删除第j个数据
    if(i==ERROR)
      printf("删除第%d个元素失败\n",j);
    else
      printf("删除第%d个元素成功，其值为%d\n",j,e);
  }
  printf("依次输出L的元素：");
  ListTraverse(L,print);
  DestroyList(L);
  printf("销毁L后：L=%u\n",L);
}
```

程序运行结果：

```
在L的表头依次插入1～5后：L=5 4 3 2 1
L是否空：i=0(1:是 0:否)
清空L后：L=
L是否空：i=1(1:是 0:否)
在L的表尾依次插入1～10后：L=1 2 3 4 5 6 7 8 9 10
第5个元素的值为5
没有值为0的元素
第1个元素的值为1
元素1无前驱
元素2的前驱为1
元素9的后继为10
元素10无后继
删除第11个元素失败
删除第10个元素成功，其值为10
依次输出L的元素：1 2 3 4 5 6 7 8 9
销毁L后：L=0
```

```
// algo2-12.cpp 用单链表实现算法2.1，仅有4句与algo2-1.cpp不同
#include"c1.h"
```

```
typedef int ElemType;
#include"c2-2.h" // 此句与algo2-1.cpp不同(因为采用不同的结构)
#include"bo2-2.cpp" // 此句与algo2-1.cpp不同(因为采用不同的结构)
#include"func2-3.cpp" // 包括equal()、comp()、print()、print2()和print1()函数
void Union(LinkList La,LinkList Lb) // 算法2.1，此句与algo2-1.cpp不同
{ // 将所有在线性表Lb中但不在La中的数据元素插入到La中
  ElemType e;
  int La_len,Lb_len;
  int i;
  La_len=ListLength(La); // 求线性表的长度
  Lb_len=ListLength(Lb);
  for(i=1;i<=Lb_len;i++)
  {
    GetElem(Lb,i,e); // 取Lb中第i个数据元素赋给e
    if(!LocateElem(La,e,equal)) // La中不存在和e相同的元素，则插入之
      ListInsert(La,++La_len,e);
  }
}
void main()
{
  LinkList La,Lb; // 此句与algo2-1.cpp不同(因为采用不同的结构)
  int j;
  InitList(La);
  if(i==1) // 创建空表La成功
    for(j=1;j<=5;j++) // 在表La中插入5个元素
      ListInsert(La,j,j);
  printf("La= "); // 输出表La的内容
  ListTraverse(La,print);
  InitList(Lb); // 也可不判断是否创建成功
  for(j=1;j<=5;j++) // 在表Lb中插入5个元素
    ListInsert(Lb,j,2*j);
  printf("Lb= "); // 输出表Lb的内容
  ListTraverse(Lb,print);
  Union(La,Lb);
  printf("new La= "); // 输出新表La的内容
  ListTraverse(La,print);
}
```

 　程序运行结果：

```
La= 1 2 3 4 5
Lb= 2 4 6 8 10
new La= 1 2 3 4 5 6 8 10
```

```
// algo2-13.cpp 采用单链表结构实现算法2.2的程序，仅有4句与algo2-2.cpp不同
#include"c1.h"
typedef int ElemType;
#include"c2-2.h" // 此句与algo2-2.cpp不同
#include"bo2-2.cpp" // 此句与algo2-2.cpp不同
```

```
#include"func2-3.cpp" // 包括equal()、comp()、print()、print2()和print1()函数
void MergeList(LinkList La,LinkList Lb,LinkList &Lc) // 算法2.2,此句与algo2-2.cpp不同
{ // 已知线性表La和Lb中的数据元素按值非递减排列。
  // 归并La和Lb得到新的线性表Lc, Lc的数据元素也按值非递减排列
  int i=1,j=1,k=0;
  int La_len,Lb_len;
  ElemType ai,bj;
  InitList(Lc); // 创建空表Lc
  La_len=ListLength(La);
  Lb_len=ListLength(Lb);
  while(i<=La_len&&j<=Lb_len) // 表La和表Lb均非空
  {
    GetElem(La,i,ai);
    GetElem(Lb,j,bj);
    if(ai<=bj)
    {
      ListInsert(Lc,++k,ai);
      ++i;
    }
    else
    {
      ListInsert(Lc,++k,bj);
      ++j;
    }
  }
  while(i<=La_len) // 表La非空且表Lb空
  {
    GetElem(La,i++,ai);
    ListInsert(Lc,++k,ai);
  }
  while(j<=Lb_len) // 表Lb非空且表La空
  {
    GetElem(Lb,j++,bj);
    ListInsert(Lc,++k,bj);
  }
}
void main()
{
  LinkList La,Lb,Lc; // 此句与algo2-2.cpp不同
  int j,a[4]={3,5,8,11},b[7]={2,6,8,9,11,15,20}; // 教科书例2-2的数据
  InitList(La); // 创建空表La
  for(j=1;j<=4;j++) // 在表La中插入4个元素
    ListInsert(La,j,a[j-1]);
  printf("La= "); // 输出表La的内容
  ListTraverse(La,print);
  InitList(Lb); // 创建空表Lb
  for(j=1;j<=7;j++) // 在表Lb中插入7个元素
    ListInsert(Lb,j,b[j-1]);
  printf("Lb= "); // 输出表Lb的内容
  ListTraverse(Lb,print);
  MergeList(La,Lb,Lc);
```

```
  printf("Lc= "); // 输出表Lc的内容
  ListTraverse(Lc,print);
}
```

 　程序运行结果：

```
La= 3 5 8 11
Lb= 2 6 8 9 11 15 20
Lc= 2 3 5 6 8 8 9 11 11 15 20
```

algo2-12.cpp 和 algo2-13.cpp 是采用链表结构实现算法 2.1 和 2.2 的程序，与采用顺序表结构实现算法 2.1 和 2.2 的程序 algo2-1.cpp 和 algo2-2.cpp 相比，只有 4 条语句不同：

(1) 包含了不同的存储结构文件；

(2) 包含了不同的基本操作函数文件；

(3) 函数的形参类型不同，是否为引用参数也不一定相同；

(4) 主程序中相应的变量类型不同。

原因是算法 2.1 和 2.2 是通过调用线性表的基本操作(如 ListLength()等)来完成的。算法中不包括针对某种具体的存储结构所特有的变量的操作。这类算法的可移植性好，从一种存储结构移植到另一种存储结构仅做少量修改即可。但由于没有考虑具体的存储结构，这类算法往往不是最高效的。而算法 2.7 的程序(在 algo2-3.cpp 中)涉及到顺序存储结构所特有的 L.length 等变量，不容易移植到其它存储结构中使用。但该程序可以充分考虑顺序存储结构的特点，从而做到高效。

```
// algo2-5.cpp 实现算法2.11、2.12的程序
#include"c1.h"
typedef int ElemType;
#include"c2-2.h"
#include"bo2-2.cpp"
#include"func2-3.cpp" // 包括equal()、comp()、print()、print2()和print1()函数
void CreateList(LinkList &L,int n) // 算法2.11
{ // 逆位序(插在表头)输入n个元素的值，建立带表头结构的单链线性表L
  int i;
  LinkList p;
  L=(LinkList)malloc(sizeof(LNode));
  L->next=NULL; // 先建立一个带头结点的单链表
  printf("请输入%d个数据\n",n);
  for(i=n;i>0;--i)
  {
    p=(LinkList)malloc(sizeof(LNode)); // 生成新结点
    scanf("%d",&p->data); // 输入元素值
    p->next=L->next; // 插入到表头
    L->next=p;
  }
}
```

```
void CreateList2(LinkList &L,int n)
{ // 正位序(插在表尾)输入n个元素的值, 建立带表头结构的单链线性表L
  int i;
  LinkList p,q;
  L=(LinkList)malloc(sizeof(LNode)); // 生成头结点
  L->next=NULL;
  q=L;
  printf("请输入%d个数据\n",n);
  for(i=1;i<=n;i++)
  {
    p=(LinkList)malloc(sizeof(LNode));
    scanf("%d",&p->data);
    q->next=p;
    q=q->next;
  }
  p->next=NULL;
}
void MergeList(LinkList La,LinkList &Lb,LinkList &Lc) // 算法2.12
{ // 已知单链线性表La和Lb的元素按值非递减排列。
  // 归并La和Lb得到新的单链线性表Lc, Lc的元素也按值非递减排列
  LinkList pa=La->next,pb=Lb->next,pc;
  Lc=pc=La; // 用La的头结点作为Lc的头结点
  while(pa&&pb)
    if(pa->data<=pb->data)
    {
      pc->next=pa;
      pc=pa;
      pa=pa->next;
    }
    else
    {
      pc->next=pb;
      pc=pb;
      pb=pb->next;
    }
  pc->next=pa?pa:pb; // 插入剩余段
  free(Lb); // 释放Lb的头结点
  Lb=NULL;
}
void main()
{
  int n=5;
  LinkList La,Lb,Lc;
  printf("按非递减顺序, ");
  CreateList2(La,n); // 正位序输入n个元素的值
  printf("La="); // 输出链表La的内容
  ListTraverse(La,print);
  printf("按非递增顺序, ");
  CreateList(Lb,n); // 逆位序输入n个元素的值
  printf("Lb="); // 输出链表Lb的内容
  ListTraverse(Lb,print);
```

```
  MergeList(La,Lb,Lc); // 按非递减顺序归并La和Lb，得到新表Lc
  printf("Lc="); // 输出链表Lc的内容
  ListTraverse(Lc,print);
}
```

程序运行结果：

```
按非递减顺序，请输入5个数据
1 2 2 3 7↙
La=1 2 2 3 7
按非递增顺序，请输入5个数据
9 8 8 7 5↙
Lb=5 7 8 8 9
Lc=1 2 2 3 5 7 7 8 8 9
```

单链表也可以不设头结点，如图 2－12 所示。显然，基于这种结构的基本操作和带有头结点的线性链表基本操作是不同的。bo2-8.cpp 是不带头结点的线性链表的基本操作。

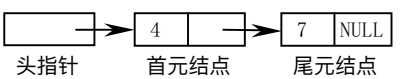


图 2－12　不带头结点且具有 2 个结点(4,7)的线性链表

```
// bo2-8.cpp 不带头结点的单链表(存储结构由c2-2.h定义)的部分基本操作(9个)
#define DestroyList ClearList // DestroyList()和ClearList()的操作是一样的
void InitList(LinkList &L)
{ // 操作结果：构造一个空的线性表L(见图2－13)
  L=NULL; // 指针为空
}
void ClearList(LinkList &L)
{ // 初始条件：线性表L已存在。操作结果：将L重置为空表(见图2－13)
  LinkList p;
  while(L) // L不空
  {
    p=L; // p指向首元结点
    L=L->next; // L指向第2个结点(新首元结点)
    free(p); // 释放首元结点
  }
}
Status ListEmpty(LinkList L)
{ // 初始条件：线性表L已存在。操作结果：若L为空表，则返回TRUE；否则返回FALSE
  if(L)
    return FALSE;
  else
    return TRUE;
}
int ListLength(LinkList L)
{ // 初始条件：线性表L已存在。操作结果：返回L中数据元素的个数
  int i=0;
  LinkList p=L;
```

图 2－13　空的和被销毁的线性表 L

```
  while(p) // p指向结点(没到表尾)
  {
    p=p->next; // p指向下一个结点
    i++;
  }
  return i;
}
Status GetElem(LinkList L,int i,ElemType &e)
{ // L为不带头结点的单链表的头指针。当第i个元素存在时，其值赋给e并返回OK；否则返回ERROR
  int j=1;
  LinkList p=L;
  if(i<1) // i值不合法
    return ERROR;
  while(j<i&&p) // 没到第i个元素，也没到表尾
  {
    j++;
    p=p->next;
  }
  if(j==i) // 存在第i个元素
  {
    e=p->data;
    return OK;
  }
  else
    return ERROR;
}
int LocateElem(LinkList L,ElemType e,Status(*compare)(ElemType,ElemType))
{ // 初始条件：线性表L已存在,compare()是数据元素判定函数(满足为1；否则为0)
  // 操作结果：返回L中第1个与e满足关系compare()的数据元素的位序。
  //           若这样的数据元素不存在，则返回值为0
  int i=0;
  LinkList p=L;
  while(p)
  {
    i++;
    if(compare(p->data,e)) // 找到这样的数据元素
      return i;
    p=p->next;
  }
  return 0;
}
Status ListInsert(LinkList &L,int i,ElemType e)
{ // 在不带头结点的单链线性表L中第i个位置之前插入元素e
  int j=1;
  LinkList p=L,s;
  if(i<1) // i值不合法
    return ERROR;
  s=(LinkList)malloc(sizeof(LNode));
              // 生成新结点
  s->data=e; // 给s的data域赋值
  if(i==1) // 插在表头(见图2－14)
  {
```

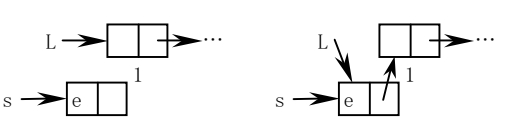
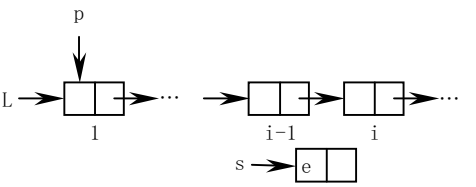


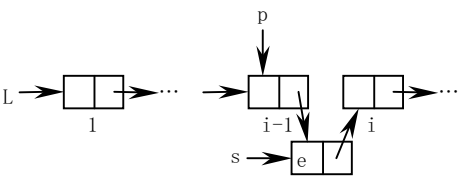
图 2－14 在链表 L 的第 1 个位置之前插入元素 e

```
    s->next=L;

    L=s; // 改变L
  }
  else
  { // 插在表的其余处(见图2－15)
    while(p&&j<i-1) // 寻找第i-1个结点
    {
      p=p->next;
      j++;
    }
    if(!p) // i大于表长+1
      return ERROR;
    s->next=p->next;
    p->next=s;
  }
  return OK;
}
Status ListDelete(LinkList &L,int i,ElemType &e)
{ // 在不带头结点的单链线性表L中,删除第i个元素,并由e返回其值
  int j=1;
  LinkList p=L,q;
  if(i==1) // 删除第1个结点(见图2－16)
  {
    L=p->next; // L由第2个结点开始
    e=p->data;
    free(p); // 删除并释放第1个结点
  }
  else(见图2－17)
  {
    while(p->next&&j<i-1) // 寻找第i个结点,并令p指向其前驱
    {
      p=p->next;
      j++;
    }
    if(!p->next||j>i-1) // 删除位置不合理
      return ERROR;
    q=p->next; // 删除并释放结点
    p->next=q->next;
    e=q->data;
    free(q);
  }
  return OK;
}
void ListTraverse(LinkList L,void(*vi)
  (ElemType))
{ // 初始条件：线性表L已存在
  // 操作结果：依次对L的每个数据元素调用
  // 函数vi()
  LinkList p=L;
  while(p)
```

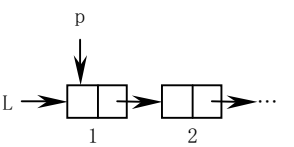


(a) 初态，p 指向第 1 个结点

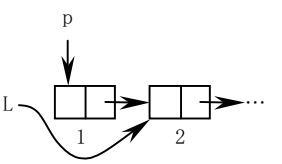


(b) 在第 i 个结点之前插入值为 e 的结点

图 2－15　在链表 L 的第 i 个位置之前插入元素 e

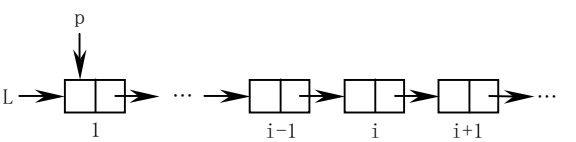


(a) p 指向第 1 个结点

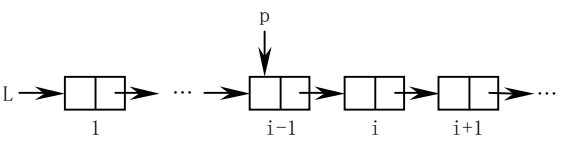


(b) L 指向第 2 个结点

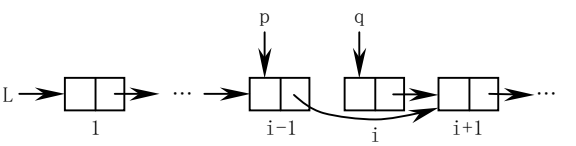
图 2－16　删除链表 L 的第 1 个结点



(a) 初态，p 指向第 1 个结点



(b) p 指向第 i-1 个结点



(c) 从链表上删除第 i 个结点

图 2－17　删除链表 L 的第 i 个结点(i!=1)

```
  {
    vi(p->data);
    p=p->next;
  }
  printf("\n");
}

// bo2-9.cpp 不带头结点的单链表(存储结构由c2-2.h定义)的部分基本操作(2个)
Status PriorElem(LinkList L,ElemType cur_e,ElemType &pre_e)
{ // 初始条件: 线性表L已存在
  // 操作结果: 若cur_e是L的数据元素, 且不是第一个, 则用pre_e返回它的前驱,
  //           返回OK; 否则操作失败, pre_e无定义, 返回INFEASIBLE
  LinkList q,p=L; // p指向第一个结点
  while(p->next) // p所指结点有后继
  {
    q=p->next; // q为p的后继
    if(q->data==cur_e)
    {
      pre_e=p->data;
      return OK;
    }
    p=q; // p向后移
  }
  return INFEASIBLE;
}
Status NextElem(LinkList L,ElemType cur_e,ElemType &next_e)
{ // 初始条件: 线性表L已存在
  // 操作结果: 若cur_e是L的数据元素, 且不是最后一个, 则用next_e返回它的后继,
  //           返回OK; 否则操作失败, next_e无定义, 返回INFEASIBLE
  LinkList p=L; // p指向第一个结点
  while(p->next) // p所指结点有后继
  {
    if(p->data==cur_e)
    {
      next_e=p->next->data;
      return OK;
    }
    p=p->next;
  }
  return INFEASIBLE;
}

// main2-8.cpp 检验bo2-8.cpp和bo2-9.cpp的主程序
#include"c1.h"
typedef int ElemType;
#include"c2-2.h"
#include"bo2-8.cpp"
#include"bo2-9.cpp"
#include"func2-3.cpp" // 包括equal()、comp()、print()、print2()和print1()函数
void main()
{
```

```
LinkList L;
ElemType e,e0;
Status i;
int j,k;
InitList(L);
for(j=1;j<=5;j++)
{
  i=ListInsert(L,1,j);
  if(!i) // 插入失败
    exit(ERROR);
}
printf("在L的表头依次插入1～5后: L=");
ListTraverse(L,print); // 依次对元素调用print(), 输出元素的值
i=ListEmpty(L);
printf("L是否空: i=%d(1:是 0:否)\n",i);
ClearList(L);
printf("清空L后: L=");
ListTraverse(L,print);
i=ListEmpty(L);
printf("L是否空: i=%d(1:是 0:否)\n",i);
for(j=1;j<=10;j++)
  ListInsert(L,j,j);
printf("在L的表尾依次插入1～10后: L=");
ListTraverse(L,print);
i=GetElem(L,5,e);
if(i==OK)
  printf("第5个元素的值为%d\n",e);
for(j=0;j<=1;j++)
{
  k=LocateElem(L,j,equal);
  if(k)
    printf("第%d个元素的值为%d\n",k,j);
  else
    printf("没有值为%d的元素\n",j);
}
for(j=1;j<=2;j++) // 测试头两个数据
{
  GetElem(L,j,e0); // 把第j个数据赋给e0
  i=PriorElem(L,e0,e); // 求e0的前驱
  if(i==INFEASIBLE)
    printf("元素%d无前驱\n",e0);
  else
    printf("元素%d的前驱为%d\n",e0,e);
}
for(j=ListLength(L)-1;j<=ListLength(L);j++) // 最后两个数据
{
  GetElem(L,j,e0); // 把第j个数据赋给e0
  i=NextElem(L,e0,e); // 求e0的后继
  if(i==INFEASIBLE)
    printf("元素%d无后继\n",e0);
  else
```

```
      printf("元素%d的后继为%d\n",e0,e);
  }
  k=ListLength(L); // k为表长
  for(j=k+1;j>=k;j--)
  {
    i=ListDelete(L,j,e); // 删除第j个数据
    if(i==ERROR)
      printf("删除第%d个元素失败\n",j);
    else
      printf("删除第%d个元素成功，其值为%d\n",j,e);
  }
  printf("依次输出L的元素：");
  ListTraverse(L,print);
  DestroyList(L);
  printf("销毁L后：L=%u\n",L);
}
```

程序运行结果：

```
在L的表头依次插入1～5后：L=5 4 3 2 1
L是否空：i=0(1:是 0:否)
清空L后：L=
L是否空：i=1(1:是 0:否)
在L的表尾依次插入1～10后：L=1 2 3 4 5 6 7 8 9 10
第5个元素的值为5
没有值为0的元素
第1个元素的值为1
元素1无前驱
元素2的前驱为1
元素9的后继为10
元素10无后继
删除第11个元素失败
删除第10个元素成功，其值为10
依次输出L的元素：1 2 3 4 5 6 7 8 9
销毁L后：L=0
```

和带有头结点的单链表(见图 2－7)相比，不带头结点的单链表(见图 2－12)显得更直观。但不带头结点的单链表在插入和删除第 1 个元素时与插入和删除其它元素时的操作不一样，要改变链表头指针的值。而带有头结点的单链表无论插入和删除第几个元素，其操作都是统一的。这从 bo2-2.cpp 和 bo2-8.cpp 两文件中的 ListInsert()及 ListDelete()函数的区别可看出。

algo2-6.cpp 是以结构体为数据元素(ElemType 类型)，利用不带头结点的单链表对学生健康登记表进行操作的一个实例。

```
// func2-1.cpp 不带头结点的单链表(存储结构由c2-2.h定义)的扩展操作(3个)
// algo2-6.cpp和bo7-2.cpp用到
```

```
void InsertAscend(LinkList &L,ElemType e,int(*compare)(ElemType,ElemType))
{ // 按关键字非降序将e插入表L。函数compare()返回值为形参1的关键字-形参2的关键字
  LinkList q=L;
  if(!L||compare(e,L->data)<=0) // 链表空或e的关键字小于等于首结点的关键字
    ListInsert(L,1,e); // 将e插在表头，在bo2-8.cpp中
  else
  {
    while(q->next&&compare(q->next->data,e)<0) // q不是尾结点且q的下一结点关键字<e的关键字
      q=q->next;
    ListInsert(q,2,e); // 插在q所指结点后(将q作为头指针)
  }
}
LinkList Point(LinkList L,ElemType e,Status(*equal)(ElemType,ElemType),LinkList &p)
{ // 查找表L中满足条件的结点。如找到，返回指向该结点的指针，p指向该结点的前驱(若该结点是
  // 首元结点，则p=NULL)。如表L中无满足条件的结点，则返回NULL，p无定义。
  // 函数equal()的两形参的关键字相等，返回OK；否则返回ERROR
  int i,j;
  i=LocateElem(L,e,equal);
  if(i) // 找到
  {
    if(i==1) // 是首元结点
    {
      p=NULL;
      return L;
    }
    p=L;
    for(j=2;j<i;j++)
      p=p->next;
    return p->next;
  }
  return NULL; // 没找到
}
Status DeleteElem(LinkList &L,ElemType &e,Status(*equal)(ElemType,ElemType))
{ // 删除表L中满足条件的结点，并返回TRUE；如无此结点，则返回FALSE。
  // 函数equal()的两形参的关键字相等，返回OK；否则返回ERROR
  LinkList p,q;
  q=Point(L,e,equal,p);
  if(q) // 找到此结点，且q指向该结点
  {
    if(p) // 该结点不是首元结点，p指向其前驱
      ListDelete(p,2,e); // 将p作为头指针,删除第2个结点
    else // 该结点是首元结点
      ListDelete(L,1,e);
    return TRUE;
  }
  return FALSE;
}

// algo2-6.cpp 利用无头结点的单链表结构处理教科书图2.1(学生健康登记表)
#include"c1.h"
#define NAMELEN 8 // 姓名最大长度
#define CLASSLEN 4 // 班级名最大长度
```

```
struct stud // 记录的结构
{
  char name[NAMELEN+1]; // 包括'\0'
  long num;
  char sex;
  int age;
  char Class[CLASSLEN+1]; // 包括'\0'
  int health;
};
typedef stud ElemType; // 链表结点元素类型为结构体(见图2-18)
#include"c2-2.h"
#include"bo2-8.cpp" // 无头结点单链表的部分基本操作
#include"func2-1.cpp" // 无头结点单链表的扩展操作
char sta[3][9]={"健康  ","一般  ","神经衰弱"}; // 健康状况(3类)(见图2-19)
FILE *fp; // 全局变量
void Print(stud e)
{ // 显示记录e的内容
  printf("%-8s %6ld",e.name,e.num);
  if(e.sex=='m')
    printf(" 男");
  else
    printf(" 女");
  printf("%5d  %-4s",e.age,e.Class);
  printf("%9s\n",sta[e.health]);
}
void ReadIn(stud &e)
{ // 由键盘输入结点信息
  printf("请输入姓名(<=%d个字符): ",NAMELEN);
  scanf("%s",e.name);
  printf("请输入学号: ");
  scanf("%ld",&e.num);
  printf("请输入性别(m:男 f:女): ");
  scanf("%*c%c",&e.sex);
  printf("请输入年龄: ");
  scanf("%d",&e.age);
  printf("请输入班级(<=%d个字符): ",CLASSLEN);
  scanf("%s",e.Class);
  printf("请输入健康状况(0:%s 1:%s 2:%s):",sta[0],sta[1],sta[2]);
  scanf("%d",&e.health);
}
void WriteToFile(stud e)
{ // 将结点信息写入fp指定的文件
  fwrite(&e,sizeof(stud),1,fp);
}
Status ReadFromFile(stud &e)
{ // 由fp指定的文件读取结点信息到e
  int i;
  i=fread(&e,sizeof(stud),1,fp);
  if(i==1) // 读取文件成功
    return OK;
  else
    return ERROR;
```

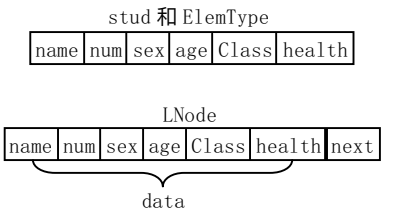


图 2-18 ElemType 和 LNode 类型

图 2-19 sta 类型

```
}
int cmp(ElemType c1,ElemType c2)
{
  return (int)(c1.num-c2.num);
}
void Modify(LinkList &L,ElemType e)
{ // 修改结点内容，并按学号将结点非降序插入链表L
  char s[80];
  Print(e); // 显示原内容
  printf("请输入待修改项的内容，不修改的项按回车键保持原值:\n");
  printf("请输入姓名(<=%d个字符): ",NAMELEN);
  gets(s);
  if(strlen(s))
    strcpy(e.name,s);
  printf("请输入学号: ");
  gets(s);
  if(strlen(s))
    e.num=atol(s);
  printf("请输入性别(m:男 f:女): ");
  gets(s);
  if(strlen(s))
    e.sex=s[0];
  printf("请输入年龄: ");
  gets(s);
  if(strlen(s))
    e.age=atoi(s);
  printf("请输入班级(<=%d个字符): ",CLASSLEN);
  gets(s);
  if(strlen(s))
    strcpy(e.Class,s);
  printf("请输入健康状况(0:%s 1:%s 2:%s):",sta[0],sta[1],sta[2]);
  gets(s);
  if(strlen(s))
    e.health=atoi(s); // 修改完毕
  InsertAscend(L,e,cmp); // 把q所指结点的内容按学号非降序插入L(func2-1.cpp)
}
#define N 4 // student记录的个数
Status EqualNum(ElemType c1,ElemType c2)
{
  if(c1.num==c2.num)
    return OK;
  else
    return ERROR;
}
Status EqualName(ElemType c1,ElemType c2)
{
  if(strcmp(c1.name,c2.name))
    return ERROR;
  else
    return OK;
}
void main()
```

```
{ // 表的初始记录
  stud student[N]={{"王小林",790631, 'm',18,"计91",0},{"陈红",790632, 'f',20,"计91",1},
                   {"刘建平",790633, 'm',21,"计91",0},{"张立立",790634, 'm',17,"计91",2}};
  int i,j,flag=1;
  char filename[13];
  ElemType e;
  LinkList T,p,q;
  InitList(T); // 初始化链表
  while(flag)
  {
    printf("1:将结构体数组student中的记录按学号非降序插入链表\n");
    printf("2:将文件中的记录按学号非降序插入链表\n");
    printf("3:键盘输入新记录, 并将其按学号非降序插入链表\n");
    printf("4:删除链表中第一个有给定学号的记录\n");
    printf("5:删除链表中第一个有给定姓名的记录\n");
    printf("6:修改链表中第一个有给定学号的记录\n");
    printf("7:修改链表中第一个有给定姓名的记录\n");
    printf("8:查找链表中第一个有给定学号的记录\n");
    printf("9:查找链表中第一个有给定姓名的记录\n");
    printf("10:显示所有记录 11:将链表中的所有记录存入文件 12:结束\n");
    printf("请选择操作命令: ");
    scanf("%d",&i);
    switch(i)
    {
      case 1: for(j=0;j<N;j++)
                InsertAscend(T,student[j],cmp); // 在func2-1.cpp中
              break;
      case 2: printf("请输入文件名: ");
              scanf("%s",filename);
              if((fp=fopen(filename,"rb"))==NULL)
                printf("打开文件失败!\n");
              else
              {
                while(ReadFromFile(e))
                  InsertAscend(T,e,cmp); // 在func2-1.cpp中
                fclose(fp);
              }
              break;
      case 3: ReadIn(e);
              InsertAscend(T,e,cmp); // 在func2-1.cpp中
              break;
      case 4: printf("请输入待删除记录的学号: ");
              scanf("%ld",&e.num);
              if(!DeleteElem(T,e,EqualNum)) // 在func2-1.cpp中
                printf("没有学号为%ld的记录\n",e.num);
              break;
      case 5: printf("请输入待删除记录的姓名: ");
              scanf("%*c%s",e.name); // %*c吃掉回车符
              if(!DeleteElem(T,e,EqualName)) // 在func2-1.cpp中
                printf("没有姓名为%s的记录\n",e.name);
              break;
      case 6: printf("请输入待修改记录的学号: ");
```

```
            scanf("%ld%*c",&e.num);
            if(!DeleteElem(T,e,EqualNum)) // 在链表中删除该结点，并由e返回(func2-1.cpp)
              printf("没有学号为%ld的记录\n",e.num);
            else
              Modify(T,e); // 修改e并按学号插入链表T
            break;
      case 7: printf("请输入待修改记录的姓名: ");
            scanf("%*c%s%*c",e.name); // %*c吃掉回车符
            if(!DeleteElem(T,e,EqualName)) // 在func2-1.cpp中
              printf("没有姓名为%s的记录\n",e.name);
            else
              Modify(T,e);
            break;
      case 8: printf("请输入待查找记录的学号: ");
            scanf("%ld",&e.num);
            if(!(q=Point(T,e,EqualNum,p))) // 在func2-1.cpp中
              printf("没有学号为%ld的记录\n",e.num);
            else
              Print(q->data);
            break;
      case 9: printf("请输入待查找记录的姓名: ");
            scanf("%*c%s",e.name);
            if(!(q=Point(T,e,EqualName,p))) // 在func2-1.cpp中
              printf("没有姓名为%s的记录\n",e.name);
            else
              Print(q->data);
            break;
      case 10:printf("  姓名     学号 性别 年龄 班级 健康状况\n");
            ListTraverse(T,Print);
            break;
      case 11:printf("请输入文件名: ");
            scanf("%s",filename);
            if((fp=fopen(filename,"wb"))==NULL)
              printf("打开文件失败!\n");
            else
              ListTraverse(T,WriteToFile);
            fclose(fp);
            break;
      case 12:flag=0;
    }
  }
}
```

程序运行结果：

```
1:将结构体数组student中的记录按学号非降序插入链表
2:将文件中的记录按学号非降序插入链表
3:键盘输入新记录，并将其按学号非降序插入链表
4:删除链表中第一个有给定学号的记录
5:删除链表中第一个有给定姓名的记录
```

```
6:修改链表中第一个有给定学号的记录
7:修改链表中第一个有给定姓名的记录
8:查找链表中第一个有给定学号的记录
9:查找链表中第一个有给定姓名的记录
10:显示所有记录 11:将链表中的所有记录存入文件 12:结束
请选择操作命令: 1↙(见图2-20)
1:将结构体数组student中的记录按学号非降序插入链表
2:将文件中的记录按学号非降序插入链表
3:键盘输入新记录，并将其按学号非降序插入链表
4:删除链表中第一个有给定学号的记录
5:删除链表中第一个有给定姓名的记录
6:修改链表中第一个有给定学号的记录
7:修改链表中第一个有给定姓名的记录
8:查找链表中第一个有给定学号的记录
9:查找链表中第一个有给定姓名的记录
10:显示所有记录 11:将链表中的所有记录存入文件 12:结束
请选择操作命令: 10↙
```

图 2-20 链表此时内容

```
  姓名      学号  性别  年龄  班级  健康状况
王小林    790631  男     18    计91     健康
陈红      790632  女     20    计91     一般
刘建平    790633  男     21    计91     健康
张立立    790634  男     17    计91  神经衰弱

1:将结构体数组student中的记录按学号非降序插入链表
2:将文件中的记录按学号非降序插入链表
3:键盘输入新记录，并将其按学号非降序插入链表
4:删除链表中第一个有给定学号的记录
5:删除链表中第一个有给定姓名的记录
6:修改链表中第一个有给定学号的记录
7:修改链表中第一个有给定姓名的记录
8:查找链表中第一个有给定学号的记录
9:查找链表中第一个有给定姓名的记录
10:显示所有记录 11:将链表中的所有记录存入文件 12:结束
请选择操作命令: 5↙
请输入待删除记录的姓名: 张立立↙(见图2-21)
1:将结构体数组student中的记录按学号非降序插入链表
2:将文件中的记录按学号非降序插入链表
3:键盘输入新记录，并将其按学号非降序插入链表
4:删除链表中第一个有给定学号的记录
5:删除链表中第一个有给定姓名的记录
6:修改链表中第一个有给定学号的记录
7:修改链表中第一个有给定姓名的记录
8:查找链表中第一个有给定学号的记录
9:查找链表中第一个有给定姓名的记录
10:显示所有记录 11:将链表中的所有记录存入文件 12:结束
请选择操作命令: 6↙
请输入待修改记录的学号: 790632↙
陈红      790632  女     20    计91     一般
请输入待修改项的内容，不修改的项按回车键保持原值:
请输入姓名(<=8个字符): ↙
请输入学号: ↙
```

图 2-21 链表此时内容

```
请输入性别(m:男 f:女): ↙
请输入年龄: ↙
请输入班级(<=4个字符): 计92↙
请输入健康状况(0:健康  1:一般  2:神经衰弱): ↙
1:将结构体数组student中的记录按学号非降序插入链表
2:将文件中的记录按学号非降序插入链表
3:键盘输入新记录，并将其按学号非降序插入链表
4:删除链表中第一个有给定学号的记录
5:删除链表中第一个有给定姓名的记录
6:修改链表中第一个有给定学号的记录
7:修改链表中第一个有给定姓名的记录
8:查找链表中第一个有给定学号的记录
9:查找链表中第一个有给定姓名的记录
10:显示所有记录 11:将链表中的所有记录存入文件 12:结束
请选择操作命令: 10↙(见图2－22)
  姓名     学号  性别 年龄 班级 健康状况
王小林    790631 男   18   计91   健康
陈红      790632 女   20   计92   一般
刘建平    790633 男   21   计91   健康

1:将结构体数组student中的记录按学号非降序插入链表
2:将文件中的记录按学号非降序插入链表
3:键盘输入新记录，并将其按学号非降序插入链表
4:删除链表中第一个有给定学号的记录
5:删除链表中第一个有给定姓名的记录
6:修改链表中第一个有给定学号的记录
7:修改链表中第一个有给定姓名的记录
8:查找链表中第一个有给定学号的记录
9:查找链表中第一个有给定姓名的记录
10:显示所有记录 11:将链表中的所有记录存入文件 12:结束
请选择操作命令: 11↙
请输入文件名: table.txt↙
1:将结构体数组student中的记录按学号非降序插入链表
2:将文件中的记录按学号非降序插入链表
3:键盘输入新记录，并将其按学号非降序插入链表
4:删除链表中第一个有给定学号的记录
5:删除链表中第一个有给定姓名的记录
6:修改链表中第一个有给定学号的记录
7:修改链表中第一个有给定姓名的记录
8:查找链表中第一个有给定学号的记录
9:查找链表中第一个有给定姓名的记录
10:显示所有记录 11:将链表中的所有记录存入文件 12:结束
请选择操作命令: 12↙
```

图 2－22　链表此时内容

结构体 stud 的 health 域的类型是整型，其值的范围是 0～2，通过数组 sta(见图 2－19)转换为 3 种健康状况。为直观，图 2－20～图 2－22 中显示的是转换后的结果。

图 2－22 的数据存到了当前目录下的 table. txt 文件中。在文本状态下打开 table. txt，其内容如下：

```
王小林□□□g□□m□□计91□□□陈红□□□□□□h□□f□□计92□  □刘建平□□□i□□m□□计
```

91□□□

其中，姓名、性别和班级可正常显示，因为它们是字符或字符串，是以 ASCII 码格式存储的。学号、年龄和健康状况显示不正常，因为它们是以整型或长整型数据的格式存储的。虽然在文本状态下它们不能正常显示，但当读入计算机时，可正常识别。

某些高级语言如 BASIC，没有“指针”数据类型，但有“结构体”数据类型。在这类高级语言中，若要使用链表结构，就需先开辟一个充分大的结构体数组。结构体的一个成员存放数据，另一个成员(“游标”)存放下一个数的位置(下标)，这称为静态链表。输出链表不是按数组的下标顺序输出的，而是由一个指定的位置开始根据游标依次输出的。c2-3.h 就是这样的一个数据结构。algo2-7.cpp 是以 c2-3.h 为数据结构，输出教科书中图 2.10 静态链表的示例程序：

```
// c2-3.h 线性表的静态单链表存储结构(见图2－23)
#define MAX_SIZE 100 // 链表的最大长度
typedef struct
{
  ElemType data;
  int cur;
}component,SLinkList[MAX_SIZE];

// algo2-7.cpp 教科书中图2.10静态链表示例
// 第1个结点的位置在[0].cur中。成员cur的值为0，则到链表尾
#include"c1.h"
#define N 6 // 字符串长度
typedef char ElemType[N];
#include"c2-3.h"
void main()
{
  SLinkList s={{"",1},{"ZHAO",2},{"QIAN",3},{"SUN",4},{"LI",5},{"ZHOU",6},{"WU",7},
  {"ZHENG",8},{"WANG",0}}; // 教科书中图2.10(a)的状态
  int i;
  i=s[0].cur; // i指示第1个结点的位置
  while(i)
  { // 输出教科书中图2.10(a)的状态
    printf("%s ",s[i].data); // 输出链表的当前值
    i=s[i].cur; // 找到下一个
  }
  printf("\n");
  s[4].cur=9; // 按教科书中图2.10(b)修改(在"LI"之后插入"SHI")
  s[9].cur=5;
  strcpy(s[9].data,"SHI");
  s[6].cur=8; // 删除"ZHENG"
  i=s[0].cur; // i指示第1个结点的位置
  while(i)
  { // 输出教科书中图2.10(b)的状态
    printf("%s ",s[i].data); // 输出链表的当前值
    i=s[i].cur; // 找到下一个
  }
  printf("\n");
```

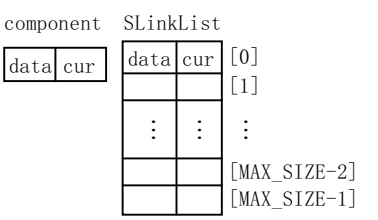


图 2－23 静态单链表存储结构

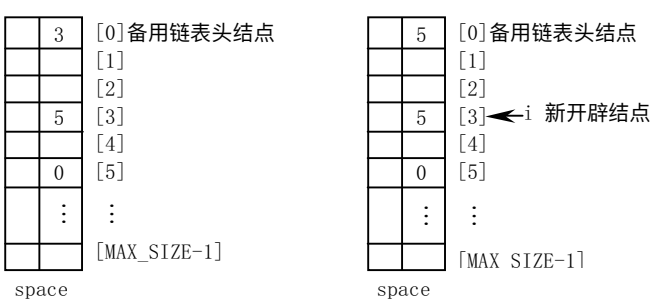
```
}
```

程序运行结果：

```
ZHAO QIAN SUN LI ZHOU WU ZHENG WANG
ZHAO QIAN SUN LI SHI ZHOU WU WANG
```

algo2-7.cpp 输出不是按数组下标的顺序输出的，而是像链表一样，由当前结点 cur 域的值决定下一个结点的位置确定输出。这就是链表的特点。但 algo2-7.cpp 插入新结点是完全靠人工判断该结点是否为空闲结点，而不是“自动地”找到空闲结点作为新结点，该方法不能满足实际应用的需要。为了解决这个问题，将所有空闲结点链接形成一个备用链表，数组下标为 0 的单元为备用链表的头结点(这时，链表的头结点就不能再是数组下标为 0 的单元了，需要另外定义)。静态数组实际有 2 个链表，一个链表上链接的是线性表的结点，另一个链表(备用链表)上链接的是所有没被使用的结点。静态数组的每一个元素都链接在这 2 个链表中的一个上。当线性表需要新结点时，把备用链表中的首元结点(由[0].cur 指示)从备用链表中删除，作为新结点，插入线性表。当删除线性表中的结点时，被删除的结点插入备用链表中，成为备用链表的首元结点。之所以从备用链表删除结点或向备用链表插入结点都在表头进行，是因为这样效率最高。开辟新结点和回收空闲结点的操作如算法 2.15、2.16 所示，在程序 func2-2.cpp 中实现。

```
// func2-2.cpp 实现算法2.15、2.16的程序，main2-31.cpp和main2-32.cpp调用
int Malloc(SLinkList space) // 算法2.15(见图2－24)
{ // 若备用链表非空，则返回分配的结点下标(备用链表的第一个结点)；否则返回0
  int i=space[0].cur;
  if(i) // 备用链表非空
    space[0].cur=space[i].cur; // 备用链表的头结点指向原备用链表的第二个结点
  return i; // 返回新开辟结点的坐标
}
```

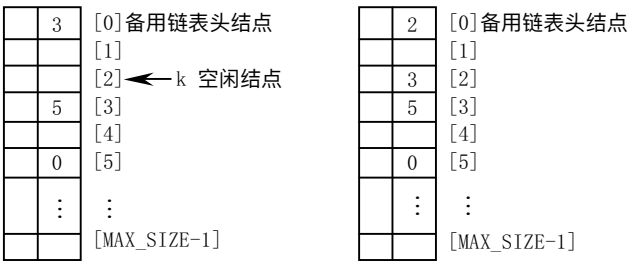


(a) 在调用 Malloc()之前，备用链表上共有 2 个结点：[3]和[5]　(b) 在调用 Malloc()之后，备用链表上只有 1 个结点[5]

图 2－24　调用 Malloc()示例

```
void Free(SLinkList space,int k) // 算法2.16(见图2－25)
{ // 将下标为k的空闲结点回收到备用链表(成为备用链表的第一个结点)
```

```
  space[k].cur=space[0].cur; // 回收结点的 " 游标 " 指向备用链表的第一个结点
  space[0].cur=k; // 备用链表的头结点指向新回收的结点
}
```



(a) 在调用 Free()之前，备用链表上有 2 个结点：[3]和[5]　　(b) 在调用 Free()之后，备用链表上有 3 个结点：[2][3][5]

图 2 - 25　调用 Free()示例

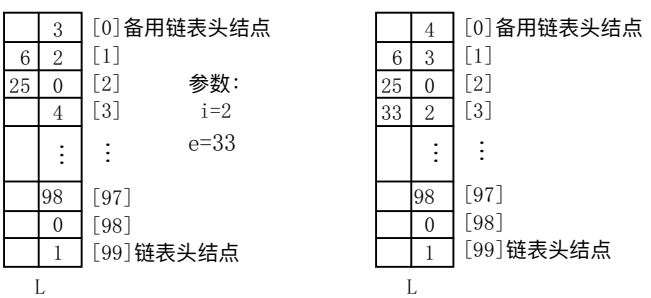
生成静态链表的方法可有两种：一种是在一个数组中只生成一个静态链表，这种情况可以固定静态链表的头指针位置，如最后一个单元([MAX_SIZE-1])；另一种是在一个数组中可根据需要生成若干个独立的链表，每个链表的头指针在生成链表时才指定。第一种方法指定数组名就指定了链表，函数的形参简单。但若在一个程序中用到多个链表，就需要定义多个数组，每个数组的备用链表不能互相调剂，空间浪费较大。第二种方法指定一个链表必须在指定数组名的同时指定链表的头指针位置，函数要多一个形参。bo2-31.cpp 是第一种情况的基本操作，main2-31.cpp 是验证 bo2-31.cpp 的主函数。

```
// bo2-31.cpp 一个数组只生成一个静态链表(数据结构由c2-3.h定义)的基本操作(11个)包括算法2.13
#define DestroyList ClearList // DestroyList()和ClearList()的操作是一样的
void InitList(SLinkList L)
{ // 构造一个空的链表L，表头为L的最后一个单元L[MAX_SIZE-1]，其余单元链成
  // 一个备用链表，表头为L的第一个单元L[0]，“ 0”表示空指针(见图2 - 26)
  int i;
  L[MAX_SIZE-1].cur=0; // L的最后一个单元为空链表的表头
  for(i=0;i<MAX_SIZE-2;i++) // 将其余单元链接成以L[0]为表头的备用链表
    L[i].cur=i+1;
  L[MAX_SIZE-2].cur=0;
}
void ClearList(SLinkList L)
{ // 初始条件：线性表L已存在。操作结果：将L重置为空表
  int i,j,k;
  i=L[MAX_SIZE-1].cur; // 链表第一个结点的位置
  L[MAX_SIZE-1].cur=0; // 链表空
  k=L[0].cur; // 备用链表第一个结点的位置
  L[0].cur=i; // 把链表的结点连到备用链表的表头
  while(i) // 没到链表尾
  {
    j=i;
    i=L[i].cur; // 指向下一个元素
  }
```

图 2 - 26　构造空链表 L

```
  L[j].cur=k; // 备用链表的第一个结点接到链表的尾部
}
Status ListEmpty(SLinkList L)
{ // 若L是空表，返回TRUE；否则返回FALSE
  if(L[MAX_SIZE-1].cur==0) // 空表
    return TRUE;
  else
    return FALSE;
}
int ListLength(SLinkList L)
{ // 返回L中数据元素个数
  int j=0,i=L[MAX_SIZE-1].cur; // i指向第一个元素
  while(i) // 没到静态链表尾
  {
    i=L[i].cur; // 指向下一个元素
    j++;
  }
  return j;
}
Status GetElem(SLinkList L,int i,ElemType &e)
{ // 用e返回L中第i个元素的值
  int l,k=MAX_SIZE-1; // k指向表头序号
  if(i<1||i>ListLength(L))
    return ERROR;
  for(l=1;l<=i;l++) // 移动到第i个元素处
    k=L[k].cur;
  e=L[k].data;
  return OK;
}
int LocateElem(SLinkList L,ElemType e) // 算法2.13(有改动)
{ // 在静态单链线性表L中查找第1个值为e的元素。若找到，则返回它在L中的
  // 位序；否则返回0。(与其它LocateElem()的定义不同)
  int i=L[MAX_SIZE-1].cur; // i指示表中第一个结点
  while(i&&L[i].data!=e) // 在表中顺链查找(e不能是字符串)
    i=L[i].cur;
  return i;
}
Status PriorElem(SLinkList L,ElemType cur_e,ElemType &pre_e)
{ // 初始条件：线性表L已存在
  // 操作结果：若cur_e是L的数据元素，且不是第一个，则用pre_e返回它的前驱；
  //           否则操作失败，pre_e无定义
  int j,i=L[MAX_SIZE-1].cur; // i指示链表第一个结点的位置
  do
  { // 向后移动结点
    j=i;
    i=L[i].cur;
  }while(i&&cur_e!=L[i].data);
  if(i) // 找到该元素
  {
    pre_e=L[j].data;
```

```
    return OK;
  }
  return ERROR;
}
Status NextElem(SLinkList L,ElemType cur_e,ElemType &next_e)
{ // 初始条件：线性表L已存在
  // 操作结果：若cur_e是L的数据元素，且不是最后一个，则用next_e返回它的后继；
  //           否则操作失败，next_e无定义
  int j,i=LocateElem(L,cur_e); // 在L中查找第一个值为cur_e的元素的位置
  if(i) // L中存在元素cur_e
  {
    j=L[i].cur; // cur_e的后继的位置
    if(j) // cur_e有后继
    {
      next_e=L[j].data;
      return OK; // cur_e元素有后继
    }
  }
  return ERROR; // L不存在cur_e元素，cur_e元素无后继
}
Status ListInsert(SLinkList L,int i,ElemType e)
{ // 在L中第i个元素之前插入新的数据元素e(见图2－27)
```

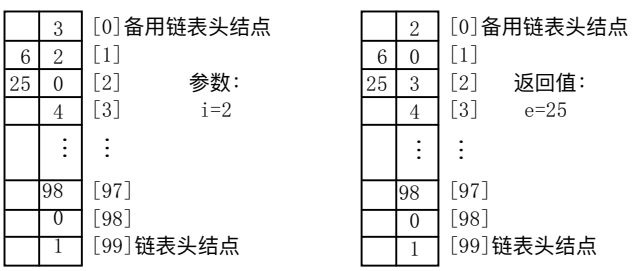


(a) 在调用 ListInsert()之前，链表 L 上有 2 个元素：6 和 25　　(b) 在调用 ListInsert()之后，链表 L 上有 3 个元素：6、33 和 25

图 2－27　调用 ListInsert()示例

```
  int l,j,k=MAX_SIZE-1; // k指向表头
  if(i<1||i>ListLength(L)+1)
    return ERROR;
  j=Malloc(L); // 申请新单元
  if(j) // 申请成功
  {
    L[j].data=e; // 赋值给新单元
    for(l=1;l<i;l++) // 移动i-1个元素
      k=L[k].cur;
    L[j].cur=L[k].cur;
    L[k].cur=j;
    return OK;
  }
```

```
  return ERROR;
}
Status ListDelete(SLinkList L,int i,ElemType &e)
{ // 删除在L中第i个数据元素e，并返回其值(见图2－28)
  int j,k=MAX_SIZE-1; // k指向表头
  if(i<1||i>ListLength(L))
    return ERROR;
  for(j=1;j<i;j++) // 移动i-1个元素
    k=L[k].cur;
  j=L[k].cur;
  L[k].cur=L[j].cur;
```



(a) 在调用 ListDelete()之前，链表 L 上有 2 个元素：6 和 25　　(b) 在调用 ListDelete()之后，链表 L 上有 1 个元素：6

图 2－28　调用 ListDelete()示例

```
  e=L[j].data;
  Free(L,j);
  return OK;
}
void ListTraverse(SLinkList L,void(*vi)(ElemType))
{ // 初始条件：线性表L已存在。操作结果：依次对L的每个数据元素调用函数vi()
  int i=L[MAX_SIZE-1].cur; // 指向第一个元素
  while(i) // 没到静态链表尾
  {
    vi(L[i].data); // 调用vi()
    i=L[i].cur; // 指向下一个元素
  }
  printf("\n");
}

// main2-31.cpp 检验func2-2.cpp和bo2-31.cpp的主程序
#include"c1.h"
typedef int ElemType;
#include"c2-3.h"
#include"func2-2.cpp" // 两种方法都适用的函数在此文件中
#include"bo2-31.cpp"
#include"func2-3.cpp" // 包括equal()、comp()、print()、print2()和print1()函数
void main()
{
  int j,k;
```

```
Status i;
ElemType e,e0;
SLinkList L;
InitList(L);
for(j=1;j<=5;j++)
  i=ListInsert(L,1,j);
printf("在L的表头依次插入1～5后: L=");
ListTraverse(L,print);
i=ListEmpty(L);
printf("L是否空:i=%d(1:是 0:否)表L的长度=%d\n",i,ListLength(L));
ClearList(L);
printf("清空L后: L=");
ListTraverse(L,print);
i=ListEmpty(L);
printf("L是否空:i=%d(1:是 0:否)表L的长度=%d\n",i,ListLength(L));
for(j=1;j<=10;j++)
  ListInsert(L,j,j);
printf("在L的表尾依次插入1～10后: L=");
ListTraverse(L,print);
GetElem(L,5,e);
printf("第5个元素的值为%d\n",e);
for(j=0;j<=1;j++)
{
  k=LocateElem(L,j);
  if(k)
    printf("值为%d的元素在静态链表中的位序为%d\n",j,k);
  else
    printf("没有值为%d的元素\n",j);
}
for(j=1;j<=2;j++) // 测试头两个数据
{
  GetElem(L,j,e0); // 把第j个数据赋给e0
  i=PriorElem(L,e0,e); // 求e0的前驱
  if(!i)
    printf("元素%d无前驱\n",e0);
  else
    printf("元素%d的前驱为%d\n",e0,e);
}
for(j=ListLength(L)-1;j<=ListLength(L);j++) // 最后两个数据
{
  GetElem(L,j,e0); // 把第j个数据赋给e0
  i=NextElem(L,e0,e); // 求e0的后继
  if(!i)
    printf("元素%d无后继\n",e0);
  else
    printf("元素%d的后继为%d\n",e0,e);
}
k=ListLength(L); // k为表长
for(j=k+1;j>=k;j--)
{
```

```
    i=ListDelete(L,j,e); // 删除第j个数据
    if(i)
      printf("第%d个元素为%d,已删除。\n",j,e);
    else
      printf("删除第%d个元素失败(不存在此元素)。\n",j);
  }
  printf("依次输出L的元素: ");
  ListTraverse(L,print); // 依次对元素调用print(), 输出元素的值
}
```

程序运行结果:

```
在L的表头依次插入1～5后: L=5 4 3 2 1
L是否空: i=0(1:是 0:否) 表L的长度=5
清空L后: L=
L是否空: i=1(1:是 0:否) 表L的长度=0
在L的表尾依次插入1～10后: L=1 2 3 4 5 6 7 8 9 10
第5个元素的值为5
没有值为0的元素
值为1的元素在静态链表中的位序为5
元素1无前驱
元素2的前驱为1
元素9的后继为10
元素10无后继
删除第11个元素失败(不存在此元素)。
第10个元素为10,已删除。
依次输出L的元素: 1 2 3 4 5 6 7 8 9
```

bo2-32.cpp 是第二种方法(在一个数组中可根据需要生成若干个独立的链表)的基本操作, 由整型形参 n 表示各个独立的链表表头的位序。main2-32.cpp 是验证 bo2-32.cpp 的主函数。

```
// bo2-32.cpp 一个数组可生成若干静态链表(数据结构由c2-3.h定义)的基本操作(12个),包括算法2.14
#define DestroyList ClearList // DestroyList()和ClearList()的操作是一样的
void InitSpace(SLinkList L) // 算法2.14。另加(见图2－29)
{ // 将一维数组L中各分量链成一个备用链表, L[0].cur为头指针。" 0"表示空指针。
  int i;
  for(i=0;i<MAX_SIZE-1;i++)
    L[i].cur=i+1;
  L[MAX_SIZE-1].cur=0;
}
int InitList(SLinkList L)
{ // 构造一个空链表, 返回值为空表在数组中的位序(见图2－30)
  int i;
  i=Malloc(L); // 调用Malloc(), 简化程序
  L[i].cur=0; // 空链表的表头指针为空(0)
  return i;
```

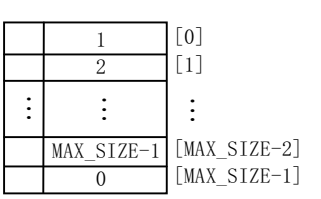
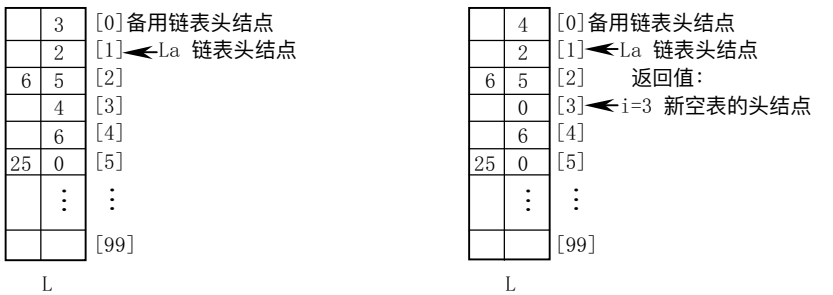


图 2－29　构造备用链表

```
}
```



(a) 在调用 InitList()之前，L 中有 1 个链表 La，上有 2 个元素：6 和 25

(b) 在调用 InitList()之后，增加了一个空表，它的头结点是L[3]。现在，L中有2个链表

图 2－30 调用 InitList()示例

```
void ClearList(SLinkList L,int n)
{ // 初始条件：L中表头位序为n的静态链表已存在。操作结果：将此表重置为空表
  int j,k,i=L[n].cur; // 链表第一个结点的位置
  L[n].cur=0; // 链表空
  k=L[0].cur; // 备用链表第一个结点的位置
  L[0].cur=i; // 把链表的结点连到备用链表的表头
  while(i) // 没到链表尾
  {
    j=i;
    i=L[i].cur; // 指向下一个元素
  }
  L[j].cur=k; // 备用链表的第一个结点接到链表的尾部
}
Status ListEmpty(SLinkList L,int n)
{ // 判断L中表头位序为n的链表是否空，若是空表返回TRUE；否则返回FALSE
  if(L[n].cur==0) // 空表
    return TRUE;
  else
    return FALSE;
}
int ListLength(SLinkList L,int n)
{ // 返回L中表头位序为n的链表的数据元素个数
  int j=0,i=L[n].cur; // i指向第一个元素
  while(i) // 没到静态链表尾
  {
    i=L[i].cur; // 指向下一个元素
    j++;
  }
  return j;
}
Status GetElem(SLinkList L,int n, int i,ElemType &e)
{ // 用e返回L中表头位序为n的链表的第i个元素的值
  int l,k=n; // k指向表头序号
  if(i<1||i>ListLength(L,n)) // i值不合理
    return ERROR;
  for(l=1;l<=i;l++) // 移动i-1个元素
```

```
    k=L[k].cur;
  e=L[k].data;
  return OK;
}
int LocateElem(SLinkList L,int n,ElemType e) // 算法2.13(有改动)
{ // 在L中表头位序为n的静态单链表中查找第1个值为e的元素。
  // 若找到，则返回它在L中的位序；否则返回0。(与其它LocateElem()的定义不同)
  int i=L[n].cur; // i指示表中第一个结点
  while(i&&L[i].data!=e) // 在表中顺链查找(e不能是字符串)
    i=L[i].cur;
  return i;
}
Status PriorElem(SLinkList L,int n,ElemType cur_e,ElemType &pre_e)
{ // 初始条件：在L中表头位序为n的静态单链表已存在
  // 操作结果：若cur_e是此单链表的数据元素，且不是第一个，
  //           则用pre_e返回它的前驱；否则操作失败，pre_e无定义
  int j,i=L[n].cur; // i为链表第一个结点的位置
  do
  { // 向后移动结点
    j=i;
    i=L[i].cur;
  }while(i&&cur_e!=L[i].data);
  if(i) // 找到该元素
  {
    pre_e=L[j].data;
    return OK;
  }
  return ERROR;
}
Status NextElem(SLinkList L,int n,ElemType cur_e,ElemType &next_e)
{ // 初始条件：在L中表头位序为n的静态单链表已存在
  // 操作结果：若cur_e是此单链表的数据元素，且不是最后一个，
  // 则用next_e返回它的后继；否则操作失败，next_e无定义
  int i;
  i=LocateElem(L,n,cur_e); // 在链表中查找第一个值为cur_e的元素的位置
  if(i) // 在静态单链表中存在元素cur_e
  {
    i=L[i].cur; // cur_e的后继的位置
    if(i) // cur_e有后继
    {
      next_e=L[i].data;
      return OK; // cur_e元素有后继
    }
  }
  return ERROR; // L不存在cur_e元素,cur_e元素无后继
}
Status ListInsert(SLinkList L,int n,int i,ElemType e)
{ // 在L中表头位序为n的链表的第i个元素之前插入新的数据元素e
  int l,j,k=n; // k指向表头
  if(i<1||i>ListLength(L,n)+1)
    return ERROR;
```

```
  j=Malloc(L); // 申请新单元
  if(j) // 申请成功
  {
    L[j].data=e; // 赋值给新单元
    for(l=1;l<i;l++) // 游标向后移动i-1个元素
      k=L[k].cur;
    L[j].cur=L[k].cur;
    L[k].cur=j;
    return OK;
  }
  return ERROR;
}
Status ListDelete(SLinkList L,int n,int i,ElemType &e)
{ // 删除在L中表头位序为n的链表的第i个数据元素e，并返回其值
  int j,k=n; // k指向表头
  if(i<1||i>ListLength(L,n))
    return ERROR;
  for(j=1;j<i;j++) // 游标向后移动i-1个元素
    k=L[k].cur;
  j=L[k].cur;
  L[k].cur=L[j].cur;
  e=L[j].data;
  Free(L,j);
  return OK;
}
void ListTraverse(SLinkList L,int n,void(*vi)(ElemType))
{ // 依次对L中表头位序为n的链表的每个数据元素调用函数vi()
  int i=L[n].cur; // 指向第一个元素
  while(i) // 没到静态链表尾
  {
    vi(L[i].data); // 调用vi()
    i=L[i].cur; // 指向下一个元素
  }
  printf("\n");
}

// main2-32.cpp 检验func2-2.cpp和bo2-32.cpp的主程序
#include"c1.h"
typedef int ElemType;
#include"c2-3.h"
#include"func2-2.cpp" // 两种方法都适用的函数在此文件中
#include"bo2-32.cpp"
#include"func2-3.cpp" // 包括equal()、comp()、print()、print2()和print1()函数
void main()
{
  int j,k,La,Lb;
  Status i;
  ElemType e,e0;
  SLinkList L;
  InitSpace(L); // 建立备用链表
  La=InitList(L); // 初始化链表La
```

```
Lb=InitList(L); // 初始化链表Lb
printf("La表空否? %d(1:空 0:否) La的表长=%d\n",ListEmpty(L,La),ListLength(L,La));
for(j=1;j<=5;j++)
  ListInsert(L,La,1,j);
printf("在空表La的表头依次插入1～5后: La=");
ListTraverse(L,La,print);
for(j=1;j<=5;j++)
  ListInsert(L,Lb,j,j);
printf("在空表Lb的表尾依次插入1～5后: Lb=");
ListTraverse(L,Lb,print);
printf("La表空否? %d(1:空 0:否) La的表长=%d\n",ListEmpty(L,La),ListLength(L,La));
ClearList(L,La);
printf("清空La后: La=");
ListTraverse(L,La,print);
printf("La表空否? %d(1:空 0:否) La的表长=%d\n",ListEmpty(L,La),ListLength(L,La));
for(j=2;j<8;j+=5)
{
  i=GetElem(L,Lb,j,e);
  if(i)
    printf("Lb表的第%d个元素的值为%d\n",j,e);
  else
    printf("Lb表不存在第%d个元素! \n",j,e);
}
for(j=0;j<=1;j++)
{
  k=LocateElem(L,Lb,j);
  if(k)
    printf("Lb表中值为%d的元素在静态链表中的位序为%d\n",j,k);
  else
    printf("Lb表中没有值为%d的元素\n",j);
}
for(j=1;j<=2;j++) // 测试头两个数据
{
  GetElem(L,Lb,j,e0); // 把第j个数据赋给e0
  i=PriorElem(L,Lb,e0,e); // 求e0的前驱
  if(!i)
    printf("Lb表中的元素%d无前驱\n",e0);
  else
    printf("Lb表中元素%d的前驱为%d\n",e0,e);
}
for(j=ListLength(L,Lb)-1;j<=ListLength(L,Lb);j++) // 最后两个数据
{
  GetElem(L,Lb,j,e0); // 把第j个数据赋给e0
  i=NextElem(L,Lb,e0,e); // 求e0的后继
  if(!i)
    printf("Lb表中元素%d无后继\n",e0);
  else
    printf("Lb表中元素%d的后继为%d\n",e0,e);
}
k=ListLength(L,Lb); // k为表长
for(j=k+1;j>=k;j--)
```

```
  {
    i=ListDelete(L,Lb,j,e); // 删除第j个数据
    if(i)
      printf("Lb表中第%d个元素为%d,已删除。\n",j,e);
    else
      printf("删除Lb表中第%d个元素失败(不存在此元素)。\n",j);
  }
  printf("依次输出Lb的元素: ");
  ListTraverse(L,Lb,print); // 依次对元素调用print(), 输出元素的值
}
```

程序运行结果:

```
La表空否? 1(1:空 0:否) La的表长=0
在空表La的表头依次插入1～5后: La=5 4 3 2 1
在空表Lb的表尾依次插入1～5后: Lb=1 2 3 4 5
La表空否? 0(1:空 0:否) La的表长=5
清空La后: La=
La表空否? 1(1:空 0:否) La的表长=0
Lb表的第2个元素的值为2
Lb表不存在第7个元素!
Lb表中没有值为0的元素
Lb表中值为1的元素在静态链表中的位序为8
Lb表中的元素1无前驱
Lb表中元素2的前驱为1
Lb表中元素4的后继为5
Lb表中元素5无后继
删除Lb表中第6个元素失败(不存在此元素)。
Lb表中第5个元素为5,已删除。
依次输出Lb的元素: 1 2 3 4
```

```
// algo2-8.cpp 实现算法2.17的程序
#include"c1.h"
#define N 2
typedef char ElemType;
#include"c2-3.h"
#include"func2-2.cpp"
#include"bo2-32.cpp"
#include"func2-3.cpp" // 包括equal()、comp()、print()、print2()和print1()函数
void difference(SLinkList space,int &S) // 算法2.17
{ // 依次输入集合A和B的元素, 在一维数组space中建立表示集合(A-B)∪(B-A)
  // 的静态链表, S为其头指针。假设备用空间足够大, space[0].cur为备用空间的头指针
  int r,p,m,n,i,j,k;
  ElemType b;
  InitSpace(space); // 初始化备用空间
  S=Malloc(space); // 生成S的头结点
  r=S; // r指向S的当前最后结点
  printf("请输入集合A和B的元素个数m,n:");
  scanf("%d,%d%*c",&m,&n); // %*c吃掉回车符
```

```
  printf("请输入集合A的元素（共%d个）:",m);
  for(j=1;j<=m;j++) // 建立集合A的链表
  {
    i=Malloc(space); // 分配结点
    scanf("%c",&space[i].data); // 输入A的元素值
    space[r].cur=i; // 插入到表尾
    r=i;
  }
  scanf("%*c"); // %*c吃掉回车符
  space[r].cur=0; // 尾结点的指针为空
  printf("请输入集合B的元素（共%d个）:",n);
  for(j=1;j<=n;j++)
  { // 依次输入B的元素，若不在当前表中，则插入；否则删除
    scanf("%c",&b);
    p=S;
    k=space[S].cur; // k指向集合A中的第一个结点
    while(k!=space[r].cur&&space[k].data!=b)
    { // 在当前表中查找
      p=k;
      k=space[k].cur;
    }
    if(k==space[r].cur)
    { // 当前表中不存在该元素，插入在r所指结点之后，且r的位置不变
      i=Malloc(space);
      space[i].data=b;
      space[i].cur=space[r].cur;
      space[r].cur=i;
    }
    else // 该元素已在表中，删除之
    {
      space[p].cur=space[k].cur;
      Free(space,k);
      if(r==k)
        r=p; // 若删除的是尾元素，则需修改尾指针
    }
  }
}
void main()
 {
   int k;
   SLinkList s;
   difference(s,k);
   ListTraverse(s,k,print2);
 }
```

程序运行结果(以教科书图 2.11 为例)：

```
请输入集合A和B的元素个数m,n:6,4↙
请输入集合A的元素（共6个）:cbegfd↙
请输入集合B的元素（共4个）:abnf↙
```

c e g d n a

algo2-9.cpp 是用静态链表的基本操作来实现算法 2.17 功能的。由于只用到 1 个链表，故采用 bo2-31.cpp 中的基本操作；又由于集合是与顺序无关的，而链表的插入以插在表头效率最高，故在 algo2-9.cpp 中插入元素时均插在表头。这只影响集合中元素的输出顺序。将 algo2-9.cpp 与 algo2-8.cpp 对比可见，采用基本操作可使程序简洁明了，思路清晰，因此在编程时应尽量采用已有的基本操作，以提高效率。

```
// algo2-9.cpp 尽量采用bo2-31.cpp中的基本操作实现算法2.17的功能
#include"c1.h"
#define N 2
typedef char ElemType;
#include"c2-3.h"
#include"func2-2.cpp"
#include"bo2-31.cpp"
#include"func2-3.cpp" // 包括equal()、comp()、print()、print2()和print1()函数
void difference(SLinkList space) // 改进算法2.17(尽量利用基本操作实现)
{ // 依次输入集合A和B的元素，在一维数组space中建立表示集合(A-B)∪(B-A)的静态链表
  int m,n,i,j;
  ElemType b,c;
  InitList(space); // 构造空链表
  printf("请输入集合A和B的元素个数m,n:");
  scanf("%d,%d%*c",&m,&n); // %*c吃掉回车符
  printf("请输入集合A的元素（共%d个）:",m);
  for(j=1;j<=m;j++) // 建立集合A的链表
  {
    scanf("%c",&b); // 输入A的元素值
    ListInsert(space,1,b); // 插入到表头
  }
  scanf("%*c"); // 吃掉回车符
  printf("请输入集合B的元素（共%d个）:",n);
  for(j=1;j<=n;j++)
  { // 依次输入B的元素，若不在当前表中，则插入；否则删除
    scanf("%c",&b);
    for(i=1;i<=ListLength(space);i++)
    {
      GetElem(space,i,c); // 依次求得表中第i个元素的值，并将其赋给c
      if(c==b) // 表中存在b,且其是第i个元素
      {
        ListDelete(space,i,c); // 删除第i个元素
        break; // 跳出i循环
      }
    }
    if(i>ListLength(space)) // 表中不存在b
      ListInsert(space,1,b); // 将b插在表头
  }
}
void main()
{
```

```
  SLinkList s;
  difference(s);
  ListTraverse(s,print2);
}
```

程序运行结果(以教科书图 2.11 为例):

```
请输入集合A和B的元素个数m,n:6,4↙
请输入集合A的元素（共6个）:cbegfd↙
请输入集合B的元素（共4个）:abnf↙
n a d g e c
```

2.3.2　循环链表

单链的循环链表结点的存储结构和单链表的存储结构一样，所不同的是：最后一个结点的 next 域指向头结点，而不是“空”。这样，由表尾很容易找到表头。但若链表较长，则由表头找到表尾较费时，因而，单循环链表往往设立尾指针而不是头指针，如图 2－31 所示。这在两个链表首尾相连合并成一个链表时非常方便。Bo2-4.cpp 是设立尾指针的单循环链表的基本操作。

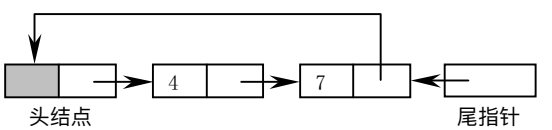


图 2－31　设立尾指针且具有 2 个结点(4,7)的单循环链表

```
// bo2-4.cpp 设立尾指针的单循环链表(存储结构由c2-2.h定义)的12个基本操作
void InitList(LinkList &L)
{ // 操作结果：构造一个空的线性表L(见图2－32)
  L=(LinkList)malloc(sizeof(LNode)); // 产生头结点，并使L指向此头结点
  if(!L) // 存储分配失败
    exit(OVERFLOW);
  L->next=L; // 指针域指向头结点
}
void DestroyList(LinkList &L)
{ // 操作结果：销毁线性表L(见图2－33)
  LinkList q,p=L->next; // p指向头结点
  while(p!=L) // 没到表尾
  {
    q=p->next;
    free(p);
    p=q;
  }
  free(L);
  L=NULL;
}
```

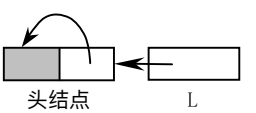


图 2－32　空(仅有头结点)的单循环链表 L

图 2－33　线性表 L 被销毁

```
void ClearList(LinkList &L) // 改变L
{ // 初始条件：线性表L已存在。操作结果：将L重置为空表(见图2－32)
  LinkList p,q;
  L=L->next; // L指向头结点
  p=L->next; // p指向第一个结点
  while(p!=L) // 没到表尾
  {
    q=p->next;
    free(p);
    p=q;
  }
  L->next=L; // 头结点指针域指向自身
}
Status ListEmpty(LinkList L)
{ // 初始条件：线性表L已存在。操作结果：若L为空表，则返回TRUE；否则返回FALSE
  if(L->next==L) // 空
    return TRUE;
  else
    return FALSE;
}
int ListLength(LinkList L)
{ // 初始条件：L已存在。操作结果：返回L中数据元素个数
  int i=0;
  LinkList p=L->next; // p指向头结点
  while(p!=L) // 没到表尾
  {
    i++;
    p=p->next;
  }
  return i;
}
Status GetElem(LinkList L,int i,ElemType &e)
{ // 当第i个元素存在时，其值赋给e并返回OK；否则返回ERROR
  int j=1; // 初始化，j为计数器
  LinkList p=L->next->next; // p指向第一个结点
  if(i<=0||i>ListLength(L)) // 第i个元素不存在
    return ERROR;
  while(j<i)
  { // 顺指针向后查找，直到p指向第i个元素
    p=p->next;
    j++;
  }
  e=p->data; // 取第i个元素
  return OK;
}
int LocateElem(LinkList L,ElemType e,Status(*compare)(ElemType,ElemType))
{ // 初始条件：线性表L已存在，compare()是数据元素判定函数
  // 操作结果：返回L中第1个与e满足关系compare()的数据元素的位序。
  //           若这样的数据元素不存在，则返回值为0
  int i=0;
```

```
  LinkList p=L->next->next; // p指向第一个结点
  while(p!=L->next)
  {
    i++;
    if(compare(p->data,e)) // 满足关系
      return i;
    p=p->next;
  }
  return 0;
}
Status PriorElem(LinkList L,ElemType cur_e,ElemType &pre_e)
{ // 初始条件: 线性表L已存在
  // 操作结果: 若cur_e是L的数据元素, 且不是第一个, 则用pre_e返回它的前驱;
  //           否则操作失败, pre_e无定义
  LinkList q,p=L->next->next; // p指向第一个结点
  q=p->next;
  while(q!=L->next) // p没到表尾
  {
    if(q->data==cur_e)
    {
      pre_e=p->data;
      return TRUE;
    }
    p=q;
    q=q->next;
  }
  return FALSE; // 操作失败
}
Status NextElem(LinkList L,ElemType cur_e,ElemType &next_e)
{ // 初始条件: 线性表L已存在
  // 操作结果: 若cur_e是L的数据元素, 且不是最后一个, 则用next_e返回它的后继;
  //           否则操作失败, next_e无定义
  LinkList p=L->next->next; // p指向第一个结点
  while(p!=L) // p没到表尾
  {
    if(p->data==cur_e)
    {
      next_e=p->next->data;
      return TRUE;
    }
    p=p->next;
  }
  return FALSE; // 操作失败
}
Status ListInsert(LinkList &L,int i,ElemType e) // 改变L
{ // 在L的第i个位置之前插入元素e
  LinkList p=L->next,s; // p指向头结点
  int j=0;
  if(i<=0||i>ListLength(L)+1) // 无法在第i个元素之前插入
    return ERROR;
  while(j<i-1) // 寻找第i-1个结点
```

```
  {
    p=p->next;
    j++;
  }
  s=(LinkList)malloc(sizeof(LNode)); // 生成新结点
  s->data=e; // 插入L中
  s->next=p->next;
  p->next=s;
  if(p==L) // 改变尾结点(见图2－34)
    L=s;
  return OK;
}
Status ListDelete(LinkList &L,int i,ElemType &e)
                // 改变L
{ // 删除L的第i个元素，并由e返回其值
  LinkList p=L->next,q; // p指向头结点
  int j=0;
  if(i<=0||i>ListLength(L)) // 第i个元素不存在
    return ERROR;
  while(j<i-1) // 寻找第i-1个结点
  {
    p=p->next;
    j++;
  }
  q=p->next; // q指向待删除结点
  p->next=q->next;
  e=q->data;
  if(L==q) // 删除的是表尾元素(见图2－35)
    L=p;
  free(q); // 释放待删除结点
  return OK;
}
void ListTraverse(LinkList L,void(*vi)(ElemType))
{ // 初始条件：L已存在
  // 操作结果：依次对L的每个数据元素调用函数vi()
  LinkList p=L->next->next; // p指向首元结点
  while(p!=L->next) // p不指向头结点
  {
    vi(p->data);
    p=p->next;
  }
  printf("\n");
}


// main2-4.cpp 单循环链表,检验bo2-4.cpp的主程序
#include"c1.h"
typedef int ElemType;
#include"c2-2.h"
#include"bo2-4.cpp"
#include"func2-3.cpp" // 包括equal()、comp()、print()、print2()和print1()函数
void main()
```

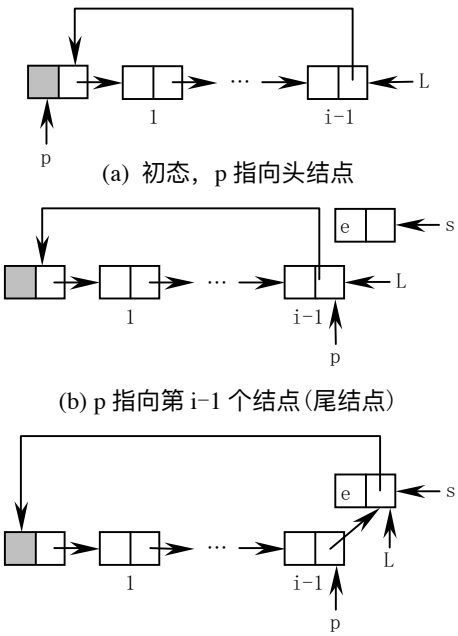


图 2－34 在链表 L 的表尾插入元素 e

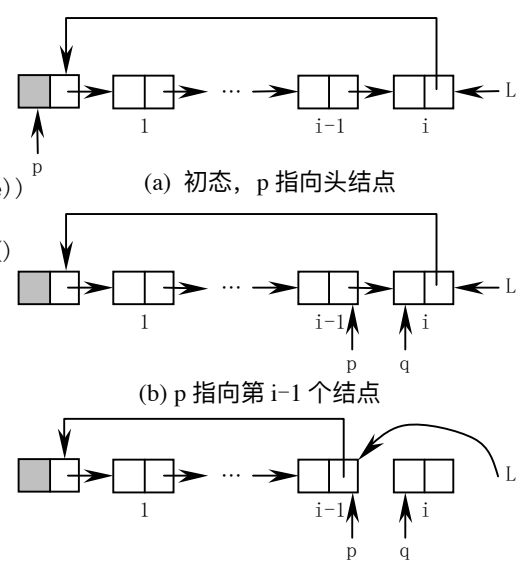


图 2－35 删除链表 L 的表尾结点

```
{
  LinkList L;
  ElemType e;
  int j;
  Status i;
  InitList(L); // 初始化单循环链表L
  i=ListEmpty(L);
  printf("L是否空 i=%d(1:空 0:否)\n",i);
  ListInsert(L,1,3); // 在L中依次插入3,5
  ListInsert(L,2,5);
  i=GetElem(L,1,e);
  j=ListLength(L);
  printf("L中数据元素个数=%d,第1个元素的值为%d。\n",j,e);
  printf("L中的数据元素依次为");
  ListTraverse(L,print);
  PriorElem(L,5,e); // 求元素5的前驱
  printf("5前面的元素的值为%d。\n",e);
  NextElem(L,3,e); // 求元素3的后继
  printf("3后面的元素的值为%d。\n",e);
  printf("L是否空 %d(1:空 0:否)\n",ListEmpty(L));
  j=LocateElem(L,5,equal);
  if(j)
    printf("L的第%d个元素为5。\n",j);
  else
    printf("不存在值为5的元素\n");
  i=ListDelete(L,2,e);
  printf("删除L的第2个元素: \n");
  if(i)
  {
    printf("删除的元素值为%d,现在L中的数据元素依次为",e);
    ListTraverse(L,print);
  }
  else
    printf("删除不成功! \n");
  ClearList(L);
  printf("清空L后, L是否空: %d(1:空 0:否)\n",ListEmpty(L));
  DestroyList(L);
}
```

程序运行结果:

```
L是否空 i=1(1:空 0:否)
L中数据元素个数=2,第1个元素的值为3。
L中的数据元素依次为3 5
5前面的元素的值为3。
3后面的元素的值为5。
L是否空 0(1:空 0:否)
L的第2个元素为5。
删除L的第2个元素:
删除的元素值为5,现在L中的数据元素依次为3
```

```
清空L后，L是否空：1(1:空 0:否)
```

```
// algo2-10.cpp 两个仅设表尾指针的循环链表的合并(教科书图2.13)
#include"c1.h"
typedef int ElemType;
#include"c2-2.h"
#include"bo2-4.cpp"
#include"func2-3.cpp" // 包括equal()、comp()、print()、print2()和print1()函数
void MergeList(LinkList &La,LinkList Lb)
{ // 将Lb合并到La的表尾，由La指示新表
  LinkList p=Lb->next;
  Lb->next=La->next;
  La->next=p->next;
  free(p);
  La=Lb;
}
void main()
{
  int n=5,i;
  LinkList La,Lb;
  InitList(La);
  for(i=1;i<=n;i++)
    ListInsert(La,i,i);
  printf("La="); // 输出链表La的内容
  ListTraverse(La,print);
  InitList(Lb);
  for(i=1;i<=n;i++)
    ListInsert(Lb,1,i*2);
  printf("Lb="); // 输出链表Lb的内容
  ListTraverse(Lb,print);
  MergeList(La,Lb);
  printf("La+Lb="); // 输出合并后的链表的内容
  ListTraverse(La,print);
}
```

程序运行结果：

```
La=1 2 3 4 5
Lb=10 8 6 4 2
La+Lb=1 2 3 4 5 10 8 6 4 2
```

2.3.3　双向链表

```
// c2-4.h 线性表的双向链表存储结构(见图2－36)
typedef struct DuLNode
{
  ElemType data;
  DuLNode *prior,*next;
}DuLNode,*DuLinkList;
```

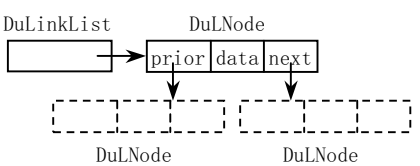


图 2－36　线性表的双向链表存储结构

双向链表(见图 2－37)每个结点有两个指针，一个指向结点的前驱，另一个指向结点的后继。所以，从链表的每一个结点出发，都可到达任意一个结点，有利于链表的查找。单链表的找前驱函数，除了有指向当前结点的指针外，还有一个紧跟其后，一直指向其前驱的指针。在双向链表中，不需要这个指向前驱的指针。

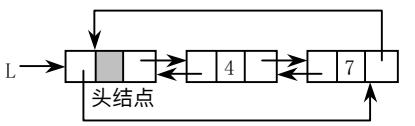


图 2－37　带有头结点且具有 2 个结点(4,7)的双向循环链表 L

```
// bo2-5.cpp 带头结点的双向循环链表(存储结构由c2-4.h定义)的基本操作(14个),包括算法2.18,2.19
void InitList(DuLinkList &L)
{ // 产生空的双向循环链表L(见图2－38)
  L=(DuLinkList)malloc(sizeof(DuLNode));
  if(L)
    L->next=L->prior=L;
  else
    exit(OVERFLOW);
}
```

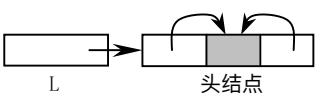


图 2－38　空的双向循环链表

```
void DestroyList(DuLinkList &L)
{ // 操作结果：销毁双向循环链表L(见图2－39)
  DuLinkList q,p=L->next; // p指向第一个结点
  while(p!=L) // p没到表头
  {
    q=p->next;
    free(p);
    p=q;
  }
  free(L);
  L=NULL;
}
```

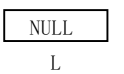


图 2－39　销毁双向循环链表 L

```
void ClearList(DuLinkList L) // 不改变L
{ // 初始条件：L已存在。操作结果：将L重置为空表(见图2－38)
  DuLinkList q,p=L->next; // p指向第一个结点
  while(p!=L) // p没到表头
  {
    q=p->next;
    free(p);
    p=q;
  }
  L->next=L->prior=L; // 头结点的两个指针域均指向自身
}
Status ListEmpty(DuLinkList L)
{ // 初始条件：线性表L已存在。操作结果：若L为空表，则返回TRUE；否则返回FALSE
  if(L->next==L&&L->prior==L)
    return TRUE;
  else
    return FALSE;
}
```

```
int ListLength(DuLinkList L)
{ // 初始条件: L已存在。操作结果: 返回L中数据元素个数
  int i=0;
  DuLinkList p=L->next; // p指向第1个结点
  while(p!=L) // p没到表头
  {
    i++;
    p=p->next;
  }
  return i;
}
Status GetElem(DuLinkList L,int i,ElemType &e)
{ // 当第i个元素存在时, 其值赋给e并返回OK; 否则返回ERROR
  int j=1; // j为计数器
  DuLinkList p=L->next; // p指向第1个结点
  while(p!=L&&j<i) // 顺指针向后查找, 直到p指向第i个元素或p指向头结点
  {
    p=p->next;
    j++;
  }
  if(p==L||j>i) // 第i个元素不存在
    return ERROR;
  e=p->data; // 取第i个元素
  return OK;
}
int LocateElem(DuLinkList L,ElemType e,Status(*compare)(ElemType,ElemType))
{ // 初始条件: L已存在, compare()是数据元素判定函数
  // 操作结果: 返回L中第1个与e满足关系compare()的数据元素的位序。
  //           若这样的数据元素不存在, 则返回值为0
  int i=0;
  DuLinkList p=L->next; // p指向第1个元素
  while(p!=L)
  {
    i++;
    if(compare(p->data,e)) // 找到这样的数据元素
      return i;
    p=p->next;
  }
  return 0;
}
Status PriorElem(DuLinkList L,ElemType cur_e,ElemType &pre_e)
{ // 操作结果: 若cur_e是L的数据元素, 且不是第一个, 则用pre_e返回它的前驱;
  //           否则操作失败, pre_e无定义
  DuLinkList p=L->next->next; // p指向第2个元素
  while(p!=L) // p没到表头
  {
    if(p->data==cur_e)
    {
      pre_e=p->prior->data;
      return TRUE;
    }
```

```
    p=p->next;
  }
  return FALSE;
}
Status NextElem(DuLinkList L,ElemType cur_e,ElemType &next_e)
{ // 操作结果：若cur_e是L的数据元素，且不是最后一个，则用next_e返回它的后继；
  //           否则操作失败，next_e无定义
  DuLinkList p=L->next->next; // p指向第2个元素
  while(p!=L) // p没到表头
  {
    if(p->prior->data==cur_e)
    {
      next_e=p->data;
      return TRUE;
    }
    p=p->next;
  }
  return FALSE;
}
DuLinkList GetElemP(DuLinkList L,int i) // 另加
{ // 在双向链表L中返回第i个元素的地址。i为0，返回头结点的地址。若第i个元素不存在，
  // 返回NULL(算法2.18、2.19要调用的函数)
  int j;
  DuLinkList p=L; // p指向头结点
  if(i<0||i>ListLength(L)) // i值不合法
    return NULL;
  for(j=1;j<=i;j++)
    p=p->next;
  return p;
}
Status ListInsert(DuLinkList L,int i,ElemType e)
{ // 在带头结点的双链循环线性表L中第i个位置之前插入元素e，i的合法值为1≤i≤表长+1
  // 改进算法2.18；否则无法在第表长+1个结点之前插入元素
  DuLinkList p,s;
  if(i<1||i>ListLength(L)+1) // i值不合法
    return ERROR;
  p=GetElemP(L,i-1); // 在L中确定第i个元素前驱的位置指针p
  if(!p) // p=NULL,即第i个元素的前驱不存在(设头结点为第1个元素的前驱)
    return ERROR;
  s=(DuLinkList)malloc(sizeof(DuLNode));
  if(!s)
    return OVERFLOW;
  s->data=e;
  s->prior=p; // 在第i-1个元素之后插入
  s->next=p->next;
  p->next->prior=s;
  p->next=s;
  return OK;
}
Status ListDelete(DuLinkList L,int i,ElemType &e) // 算法2.19
{ // 删除带头结点的双链循环线性表L的第i个元素，i的合法值为1≤i≤表长
```

```
  DuLinkList p;
  if(i<1) // i值不合法
    return ERROR;
  p=GetElemP(L,i);  // 在L中确定第i个元素的位置指针p
  if(!p) // p=NULL,即第i个元素不存在
    return ERROR;
  e=p->data;
  p->prior->next=p->next;
  p->next->prior=p->prior;
  free(p);
  return OK;
}
void ListTraverse(DuLinkList L,void(*visit)(ElemType))
{ // 由双链循环线性表L的头结点出发，正序对每个数据元素调用函数visit()
  DuLinkList p=L->next; // p指向头结点
  while(p!=L)
  {
    visit(p->data);
    p=p->next;
  }
  printf("\n");
}
void ListTraverseBack(DuLinkList L,void(*visit)(ElemType))
{ // 由双链循环线性表L的头结点出发，逆序对每个数据元素调用函数visit()。另加
  DuLinkList p=L->prior; // p指向尾结点
  while(p!=L)
  {
    visit(p->data);
    p=p->prior;
  }
  printf("\n");
}

// main2-5.cpp 检验bo2-5.cpp的主程序
#include"c1.h"
typedef int ElemType;
#include"c2-4.h"
#include"bo2-5.cpp"
#include"func2-3.cpp" // 包括equal()、comp()、print()、print2()和print1()函数
void main()
{
  DuLinkList L;
  int i,n;
  Status j;
  ElemType e;
  InitList(L);
  for(i=1;i<=5;i++)
    ListInsert(L,i,i); // 在第i个结点之前插入i
  printf("正序输出链表: ");
  ListTraverse(L,print); // 正序输出
  printf("逆序输出链表: ");
```

```
  ListTraverseBack(L,print); // 逆序输出
  n=2;
  ListDelete(L,n,e); // 删除并释放第n个结点
  printf("删除第%d个结点，值为%d，其余结点为",n,e);
  ListTraverse(L,print); // 正序输出
  printf("链表的元素个数为%d\n",ListLength(L));
  printf("链表是否空：%d(1:是 0:否)\n",ListEmpty(L));
  ClearList(L); // 清空链表
  printf("清空后，链表是否空：%d(1:是 0:否)\n",ListEmpty(L));
  for(i=1;i<=5;i++)
    ListInsert(L,i,i); // 重新插入5个结点
  ListTraverse(L,print); // 正序输出
  n=3;
  j=GetElem(L,n,e); // 将链表的第n个元素赋值给e
  if(j)
    printf("链表的第%d个元素值为%d\n",n,e);
  else
    printf("不存在第%d个元素\n",n);
  n=4;
  i=LocateElem(L,n,equal);
  if(i)
    printf("等于%d的元素是第%d个\n",n,i);
  else
    printf("没有等于%d的元素\n",n);
  j=PriorElem(L,n,e);
  if(j)
    printf("%d的前驱是%d\n",n,e);
  else
    printf("不存在%d的前驱\n",n);
  j=NextElem(L,n,e);
  if(j)
    printf("%d的后继是%d\n",n,e);
  else
    printf("不存在%d的后继\n",n);
  DestroyList(L);
}
```

 程序运行结果：

```
正序输出链表：1 2 3 4 5
逆序输出链表：5 4 3 2 1
删除第2个结点，值为2，其余结点为1 3 4 5
链表的元素个数为4
链表是否空：0(1:是 0:否)
清空后，链表是否空：1(1:是 0:否)
1 2 3 4 5
链表的第3个元素值为3
等于4的元素是第4个
4的前驱是3
```

4的后继是5

前面介绍的线性链表结构较简单，还不能满足实际应用的需要。其主要存在 3 个问题：第一，只有头指针，没有尾指针。如要在表尾插入结点，则效率很低。第二，求表长要从表头找到表尾效率也很低。第三，基本操作函数太少。c2-5.h 是从实际应用角度出发重新定义的线性链表类型，LinkList 类型增加了尾指针和表长 2 个成员，成为结构体类型。bo2-6.cpp 是基于 c2-5.h 的基本操作。

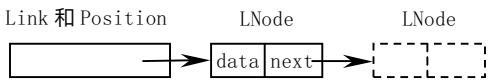


图 2-40 结点类型

```
// c2-5.h 带头结点的线性链表类型
typedef struct LNode // 结点类型(见图2-40)
{
  ElemType data;
  LNode *next;
}*Link,*Position;
struct LinkList // 链表类型(见图2-41)
{
  Link head,tail; // 分别指向线性链表中的头结点和最后一个结点
  int len; // 指示线性链表中数据元素的个数
};
```

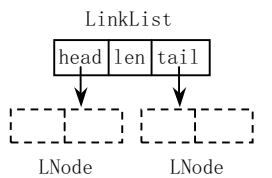


图 2-41 链表类型

图 2-42 是根据 c2-5.h 定义的具有 2 个结点的线性链表的结构。

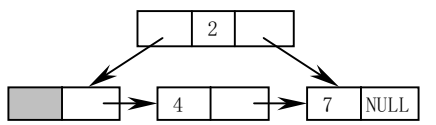


图 2-42 具有 2 个结点且带头结点的线性链表

```
// bo2-6.cpp 具有实用意义的线性链表(存储结构由c2-5.h定义)的24个基本操作
void MakeNode(Link &p,ElemType e)
{ // 分配由p指向的值为e的结点。若分配失败，则退出
  p=(Link)malloc(sizeof(LNode));
  if(!p)
    exit(ERROR);
  p->data=e;
}
void FreeNode(Link &p)
{ // 释放p所指结点
  free(p);
  p=NULL;
}
void InitList(LinkList &L)
{ // 构造一个空的线性链表L(见图2-43)
  Link p;
  p=(Link)malloc(sizeof(LNode)); // 生成头结点
  if(p)
  {
    p->next=NULL;
    L.head=L.tail=p;
    L.len=0;
  }
  else
    exit(ERROR);
```

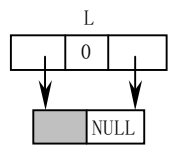


图 2-43 空的线性链表 L

```
}
void ClearList(LinkList &L)
{ // 将线性链表L重置为空表，并释放原链表的结点空间
  Link p,q;
  if(L.head!=L.tail) // 不是空表
  {
    p=q=L.head->next;
    L.head->next=NULL;
    while(p!=L.tail)
    {
      p=q->next;
      free(q);
      q=p;
    }
    free(q);
    L.tail=L.head;
    L.len=0;
  }
}
void DestroyList(LinkList &L)
{ // 销毁线性链表L，L不再存在(见图2－44)
  ClearList(L); // 清空链表
  FreeNode(L.head);
  L.tail=NULL;
  L.len=0;
}
void InsFirst(LinkList &L,Link h,Link s) // 形参增加L,因为需修改L
{ // h指向L的一个结点，把h当做头结点，将s所指结点插入在第一个结点之前
  s->next=h->next;
  h->next=s;
  if(h==L.tail) // h指向尾结点
    L.tail=h->next; // 修改尾指针
  L.len++;
}
Status DelFirst(LinkList &L,Link h,Link &q) // 形参增加L,因为需修改L
{ // h指向L的一个结点，把h当做头结点，删除链表中的第一个结点并以q返回。
  // 若链表为空(h指向尾结点)，q=NULL，返回FALSE
  q=h->next;
  if(q) // 链表非空
  {
    h->next=q->next;
    if(!h->next) // 删除尾结点
      L.tail=h; // 修改尾指针
    L.len--;
    return OK;
  }
  else
    return FALSE; // 链表空
}
void Append(LinkList &L,Link s)
{ // 将指针s(s->data为第一个数据元素)所指(彼此以指针相链，以NULL结尾)的
```

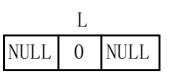


图 2－44　销毁后的线性链表 L

```
  // 一串结点链接在线性链表L的最后一个结点之后，并改变链表L的尾指针指向新的尾结点
  int i=1;
  L.tail->next=s;
  while(s->next)
  {
    s=s->next;
    i++;
  }
  L.tail=s;
  L.len+=i;
}
Position PriorPos(LinkList L,Link p)
{ // 已知p指向线性链表L中的一个结点，返回p所指结点的直接前驱的位置。若无前驱，则返回NULL
  Link q;
  q=L.head->next;
  if(q==p) // 无前驱
    return NULL;
  else
  {
    while(q->next!=p) // q不是p的直接前驱
      q=q->next;
    return q;
  }
}
Status Remove(LinkList &L,Link &q)
{ // 删除线性链表L中的尾结点并以q返回，改变链表L的尾指针指向新的尾结点
  Link p=L.head;
  if(L.len==0) // 空表
  {
    q=NULL;
    return FALSE;
  }
  while(p->next!=L.tail)
    p=p->next;
  q=L.tail;
  p->next=NULL;
  L.tail=p;
  L.len--;
  return OK;
}
void InsBefore(LinkList &L,Link &p,Link s)
{ // 已知p指向线性链表L中的一个结点，将s所指结点插入在p所指结点之前，
  // 并修改指针p指向新插入的结点
  Link q;
  q=PriorPos(L,p); // q是p的前驱
  if(!q) // p无前驱
    q=L.head;
  s->next=p;
  q->next=s;
  p=s;
  L.len++;
```

```
}
void InsAfter(LinkList &L,Link &p,Link s)
{ // 已知p指向线性链表L中的一个结点，将s所指结点插入在p所指结点之后，
  // 并修改指针p指向新插入的结点
  if(p==L.tail) // 修改尾指针
    L.tail=s;
  s->next=p->next;
  p->next=s;
  p=s;
  L.len++;
}
void SetCurElem(Link p,ElemType e)
{ // 已知p指向线性链表中的一个结点，用e更新p所指结点中数据元素的值
  p->data=e;
}
ElemType GetCurElem(Link p)
{ // 已知p指向线性链表中的一个结点，返回p所指结点中数据元素的值
  return p->data;
}
Status ListEmpty(LinkList L)
{ // 若线性链表L为空表，则返回TRUE；否则返回FALSE
  if(L.len)
    return FALSE;
  else
    return TRUE;
}
int ListLength(LinkList L)
{ // 返回线性链表L中元素个数
  return L.len;
}
Position GetHead(LinkList L)
{ // 返回线性链表L中头结点的位置
  return L.head;
}
Position GetLast(LinkList L)
{ // 返回线性链表L中最后一个结点的位置
  return L.tail;
}
Position NextPos(Link p)
{ // 已知p指向线性链表L中的一个结点，返回p所指结点的直接后继的位置。若无后继，则返回NULL
  return p->next;
}
Status LocatePos(LinkList L,int i,Link &p)
{ // 返回p指示线性链表L中第i个结点的位置，并返回OK，i值不合法时返回ERROR。i=0为头结点
  int j;
  if(i<0||i>L.len)
    return ERROR;
  else
  {
    p=L.head;
    for(j=1;j<=i;j++)
```

```
      p=p->next;
    return OK;
  }
}
Position LocateElem(LinkList L,ElemType e,Status (*compare)(ElemType,ElemType))
{ // 返回线性链表L中第1个与e满足函数compare()判定关系的元素的位置,
  // 若不存在这样的元素，则返回NULL
  Link p=L.head;
  do
    p=p->next;
  while(p&&!(compare(p->data,e))); // 没到表尾且没找到满足关系的元素
  return p;
}
void ListTraverse(LinkList L,void(*visit)(ElemType))
{ // 依次对L的每个数据元素调用函数visit()
  Link p=L.head->next;
  int j;
  for(j=1;j<=L.len;j++)
  {
    visit(p->data);
    p=p->next;
  }
  printf("\n");
}
void OrderInsert(LinkList &L,ElemType e,int (*comp)(ElemType,ElemType))
{ // 已知L为有序线性链表，将元素e按非降序插入在L中。(用于一元多项式)
  Link o,p,q;
  q=L.head;
  p=q->next;
  while(p!=NULL&&comp(p->data,e)<0) // p不是表尾且元素值小于e
  {
    q=p;
    p=p->next;
  }
  o=(Link)malloc(sizeof(LNode)); // 生成结点
  o->data=e; // 赋值
  q->next=o; // 插入
  o->next=p;
  L.len++; // 表长加1
  if(!p) // 插在表尾
    L.tail=o; // 修改尾结点
}
Status LocateElem(LinkList L,ElemType e,Position &q,int(*compare)(ElemType,ElemType))
{ // 若升序链表L中存在与e满足判定函数compare()取值为0的元素，则q指示L中
  // 第一个值为e的结点的位置，并返回TRUE；否则q指示第一个与e满足判定函数
  // compare()取值>0的元素的前驱的位置。并返回FALSE。(用于一元多项式)
  Link p=L.head,pp;
  do
  {
    pp=p;
    p=p->next;
```

```
  }while(p&&(compare(p->data,e)<0)); // 没到表尾且p->data.expn<e.expn
  if(!p||compare(p->data,e)>0) // 到表尾或compare(p->data,e)>0
  {
    q=pp;
    return FALSE;
  }
  else // 找到
  {
    q=p;
    return TRUE;
  }
}

// main2-6.cpp 检验bo2-6.cpp的主程序
#include"c1.h"
typedef int ElemType;
#include"c2-5.h"
#include"bo2-6.cpp"
#include"func2-3.cpp" // 包括equal()、comp()、print()、print2()和print1()函数
void main()
{
  Link p,h;
  LinkList L;
  Status i;
  int j,k;
  InitList(L); // 初始化空的线性表L
  for(j=1;j<=2;j++)
  {
    MakeNode(p,j); // 生成由p指向、值为j的结点
    InsFirst(L,L.tail,p); // 插在表尾
  }
  OrderInsert(L,0,comp); // 按升序插在有序表头
  for(j=0;j<=3;j++)
  {
    i=LocateElem(L,j,p,comp);
    if(i)
      printf("链表中有值为%d的元素。\n",p->data);
    else
      printf("链表中没有值为%d的元素。\n",j);
  }
  printf("输出链表: ");
  ListTraverse(L,print); // 输出L
  for(j=1;j<=4;j++)
  {
    printf("删除表头结点: ");
    DelFirst(L,L.head,p); // 删除L的首结点，并以p返回
    if(p)
      printf("%d\n",GetCurElem(p));
    else
      printf("表空，无法删除 p=%u\n",p);
  }
```

```
printf("L中结点个数=%d L是否空 %d(1:空 0:否)\n",ListLength(L),ListEmpty(L));
MakeNode(p,10);
p->next=NULL; // 尾结点
for(j=4;j>=1;j--)
{
  MakeNode(h,j*2);
  h->next=p;
  p=h;
} // h指向一串5个结点，其值依次是2 4 6 8 10
Append(L,h); // 把结点h链接在线性链表L的最后一个结点之后
OrderInsert(L,12,comp); // 按升序插在有序表尾头
OrderInsert(L,7,comp); // 按升序插在有序表中间
printf("输出链表: ");
ListTraverse(L,print); // 输出L
for(j=1;j<=2;j++)
{
  p=LocateElem(L,j*5,equal);
  if(p)
    printf("L中存在值为%d的结点。\n",j*5);
  else
    printf("L中不存在值为%d的结点。\n",j*5);
}
for(j=1;j<=2;j++)
{
  LocatePos(L,j,p); // p指向L的第j个结点
  h=PriorPos(L,p); // h指向p的前驱
  if(h)
    printf("%d的前驱是%d。\n",p->data,h->data);
  else
    printf("%d没前驱。\n",p->data);
}
k=ListLength(L);
for(j=k-1;j<=k;j++)
{
  LocatePos(L,j,p); // p指向L的第j个结点
  h=NextPos(p); // h指向p的后继
  if(h)
    printf("%d的后继是%d。\n",p->data,h->data);
  else
    printf("%d没后继。\n",p->data);
}
printf("L中结点个数=%d L是否空 %d(1:空 0:否)\n",ListLength(L),ListEmpty(L));
p=GetLast(L); // p指向最后一个结点
SetCurElem(p,15); // 将最后一个结点的值变为15
printf("第1个元素为%d 最后1个元素为%d\n",GetCurElem(GetHead(L)->next),GetCurElem(p));
MakeNode(h,10);
InsBefore(L,p,h); // 将10插到尾结点之前，p指向新结点
p=p->next; // p恢复为尾结点
MakeNode(h,20);
InsAfter(L,p,h); // 将20插到尾结点之后
k=ListLength(L);
```

```
  printf("依次删除表尾结点并输出其值: ");
  for(j=0;j<=k;j++)
    if(!(i=Remove(L,p))) // 删除不成功
      printf("删除不成功 p=%u\n",p);
    else
      printf("%d ",p->data);
  MakeNode(p,29); // 重建具有1个结点(29)的链表
  InsFirst(L,L.head,p);
  DestroyList(L); // 销毁线性链表L
  printf("销毁线性链表L之后: L.head=%u L.tail=%u L.len=%d\n",L.head,L.tail,L.len);
}
```

程序运行结果:

```
链表中有值为0的元素。
链表中有值为1的元素。
链表中有值为2的元素。
链表中没有值为3的元素。
输出链表: 0 1 2
删除表头结点: 0
删除表头结点: 1
删除表头结点: 2
删除表头结点: 表空, 无法删除 p=0
L中结点个数=0 L是否空 1(1:空 0:否)
输出链表: 2 4 6 7 8 10 12
L中不存在值为5的结点。
L中存在值为10的结点。
2没前驱。
4的前驱是2。
10的后继是12。
12没后继。
L中结点个数=7 L是否空 0(1:空 0:否)
第1个元素为2 最后1个元素为15
依次删除表尾结点并输出其值: 20 15 10 10 8 7 6 4 2 删除不成功 p=0
销毁线性链表L之后: L.head=0 L.tail=0 L.len=0
```

```
// algo2-11.cpp 实现算法2.20、2.21的程序
#include"c1.h"
typedef int ElemType;
#include"c2-5.h"
#include"bo2-6.cpp"
#include"func2-3.cpp" // 包括equal()、comp()、print()、print2()和print1()函数
Status ListInsert_L(LinkList &L,int i,ElemType e) // 算法2.20
{ // 在带头结点的单链线性表L的第i个元素之前插入元素e
  Link h,s;
  if(!LocatePos(L,i-1,h))
    return ERROR; // i值不合法
  MakeNode(s,e); // 结点分配失败则退出
  InsFirst(L,h,s); // 对于从第i个结点开始的链表, 第i-1个结点是它的头结点
```

```
  return OK;
}
void MergeList_L(LinkList &La,LinkList &Lb,LinkList &Lc,int(*compare)(ElemType,ElemType))
{ // 已知单链线性表La和Lb的元素按值非递减排列。归并La和Lb得到新的单链
  // 线性表Lc，Lc的元素也按值非递减排列。算法2.21
  Link ha,hb,pa,pb,q;
  ElemType a,b;
  InitList(Lc); // 存储空间分配失败则退出
  ha=GetHead(La); // ha和hb分别指向La和Lb的头结点
  hb=GetHead(Lb);
  pa=NextPos(ha); // pa和pb分别指向La和Lb的首元结点
  pb=NextPos(hb);
  while(pa&&pb) // La和Lb均非空
  {
    a=GetCurElem(pa); // a和b为两表中当前比较元素(第1个元素)
    b=GetCurElem(pb);
    if(compare(a,b)<=0) // a<=b
    {
      DelFirst(La,ha,q); // 移去La的首元结点并以q返回
      q->next=NULL; // 将q的next域赋值NULL，以便调用Append()
      Append(Lc,q); // 将q结点接在Lc的尾部
      pa=NextPos(ha); // pa指向La新的首元结点
    }
    else // a>b
    {
      DelFirst(Lb,hb,q); // 移去Lb的首元结点并以q返回
      q->next=NULL; // 将q的next域赋值NULL，以便调用Append()
      Append(Lc,q); // 将q结点接在Lc的尾部
      pb=NextPos(hb); // pb指向Lb新的首元结点
    }
  }
  if(pa) // La非空
    Append(Lc,pa); // 链接La中剩余结点
  else // Lb非空
    Append(Lc,pb); // 链接Lb中剩余结点
  free(ha); // 销毁La和Lb
  La.head=La.tail=NULL;
  La.len=0;
  free(hb);
  Lb.head=Lb.tail=NULL;
  Lb.len=0;
}
int diff(ElemType c1,ElemType c2)
{
  return c1-c2;
}
void main()
{
  LinkList La,Lb,Lc;
  int j;
  InitList(La);
```

```
  for(j=1;j<=5;j++)
    ListInsert_L(La,j,j); // 顺序插入 1、2、3、4、5
  printf("La=");
  ListTraverse(La,print);
  InitList(Lb);
  for(j=1;j<=5;j++)
    ListInsert_L(Lb,j,2*j); // 顺序插入 2、4、6、8、10
  printf("Lb=");
  ListTraverse(Lb,print);
  MergeList_L(La,Lb,Lc,diff); // 归并La和Lb，产生Lc
  printf("Lc=");
  ListTraverse(Lc,print);
  DestroyList(Lc);
}
```

程序运行结果：

```
La=1 2 3 4 5
Lb=2 4 6 8 10
Lc=1 2 2 3 4 4 5 6 8 10
```

2.4　一元多项式的表示及相加

```
// c2-6.h 抽象数据类型Polynomial的实现(见图2－45)
typedef struct // 项的表示，多项式的项作为LinkList的数据元素
{
  float coef; // 系数
  int expn; // 指数
}term,ElemType; // 两个类型名：term用于本ADT，ElemType为LinkList的数据对象名
```

term 和 ElemType

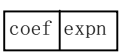


图 2－45　多项式的存储结构

图 2－46 是根据 c2-5.h 和 c2-6.h 定义的多项式 $7.3+22X^7$ 的存储结构。

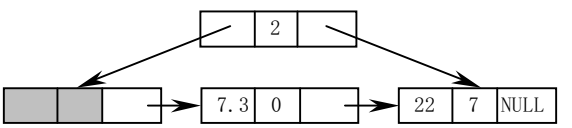


图 2－46　多项式 $7.3+22X^7$ 的存储结构

```
// bo2-7.cpp 多项式(存储结构由c2-6.h定义)的基本操作及算法2.22，2.23等(8个)
#include"c2-5.h"
#include"bo2-6.cpp"
typedef LinkList polynomial;
#define DestroyPolyn DestroyList // 与bo2-6.cpp中的函数同义不同名
#define PolynLength ListLength // 与bo2-6.cpp中的函数同义不同名
```

```
void OrderInsertMerge(LinkList &L,ElemType e,int(* compare)(term,term))
{ // 按有序判定函数compare()的约定，将值为e的结点插入或合并到升序链表L的适当位置
  Position q,s;
  if(LocateElem(L,e,q,compare)) // L中存在该指数项
  {
    q->data.coef+=e.coef; // 改变当前结点系数的值
    if(!q->data.coef) // 系数为0
    { // 删除多项式L中当前结点
      s=PriorPos(L,q); // s为当前结点的前驱
      if(!s) // q无前驱
        s=L.head;
      DelFirst(L,s,q);
      FreeNode(q);
    }
  }
  else // 生成该指数项并插入链表
  {
    MakeNode(s,e); // 生成结点
    InsFirst(L,q,s);
  }
}
int cmp(term a,term b) // CreatPolyn()的实参
{ // 依a的指数值<、=或>b的指数值，分别返回-1、0或+1
  if(a.expn==b.expn)
    return 0;
  else
    return (a.expn-b.expn)/abs(a.expn-b.expn);
}
void CreatPolyn(polynomial &P,int m) // 算法2.22
{ // 输入m项的系数和指数，建立表示一元多项式的有序链表P
  Position q,s;
  term e;
  int i;
  InitList(P);
  printf("请依次输入%d个系数，指数：\n",m);
  for(i=1;i<=m;++i)
  { // 依次输入m个非零项(可按任意顺序)
    scanf("%f,%d",&e.coef,&e.expn);
    if(!LocateElem(P,e,q,cmp)) // 当前链表中不存在该指数项，cmp是实参
    {
      MakeNode(s,e); // 生成结点并插入链表
      InsFirst(P,q,s);
    }
  }
}
void PrintPolyn(polynomial P)
{ // 打印输出一元多项式P
  Link q;
  q=P.head->next; // q指向第1个结点
  printf("  系数    指数\n");
  while(q)
```

```
  {
    printf("%f  %d\n",q->data.coef,q->data.expn);
    q=q->next;
  }
}
void AddPolyn(polynomial &Pa,polynomial &Pb) // 算法2.23
{ // 多项式加法：Pa=Pa+Pb，并销毁一元多项式Pb
  Position ha,hb,qa,qb;
  term a,b;
  ha=GetHead(Pa);
  hb=GetHead(Pb); // ha和hb分别指向Pa和Pb的头结点
  qa=NextPos(ha);
  qb=NextPos(hb); // qa和qb分别指向Pa和Pb中当前结点(现为第1个结点)
  while(!ListEmpty(Pa)&&!ListEmpty(Pb)&&qa)
  { // Pa和Pb均非空且ha没指向尾结点(qa!=0)
    a=GetCurElem(qa);
    b=GetCurElem(qb); // a和b为两表中当前比较元素
    switch(cmp(a,b))
    {
      case -1:ha=qa; // 多项式Pa中当前结点的指数值小
              qa=NextPos(ha); // ha和qa均向后移1个结点
              break;
      case 0: qa->data.coef+=qb->data.coef; // 两者的指数值相等，修改Pa当前结点的系数值
              if(qa->data.coef==0) // 删除多项式Pa中当前结点
              {
                DelFirst(Pa,ha,qa);
                FreeNode(qa);
              }
              else
                ha=qa;
              DelFirst(Pb,hb,qb);
              FreeNode(qb);
              qb=NextPos(hb);
              qa=NextPos(ha);
              break;
      case 1: DelFirst(Pb,hb,qb); // 多项式Pb中当前结点的指数值小
              InsFirst(Pa,ha,qb);
              ha=ha->next;
              qb=NextPos(hb);
    }
  }
  if(!ListEmpty(Pb))
  {
    Pb.tail=hb;
    Append(Pa,qb); // 链接Pb中剩余结点
  }
  DestroyPolyn(Pb); // 销毁Pb
}
void AddPolyn1(polynomial &Pa,polynomial &Pb)
{ // 另一种多项式加法的算法：Pa=Pa+Pb，并销毁一元多项式Pb
  Position qb;
```

```
  term b;
  qb=GetHead(Pb); // qb指向Pb的头结点
  qb=qb->next; // qb指向Pb的第1个结点
  while(qb)
  {
    b=GetCurElem(qb);
    OrderInsertMerge(Pa,b,cmp);
    qb=qb->next;
  }
  DestroyPolyn(Pb); // 销毁Pb
}
void Opposite(polynomial Pa)
{ // 一元多项式Pa系数取反
  Position p;
  p=Pa.head;
  while(p->next)
  {
    p=p->next;
    p->data.coef*=-1;
  }
}
void SubtractPolyn(polynomial &Pa,polynomial &Pb)
{ // 多项式减法: Pa=Pa-Pb, 并销毁一元多项式Pb
  Opposite(Pb);
  AddPolyn(Pa,Pb);
}
void MultiplyPolyn(polynomial &Pa,polynomial &Pb)
{ // 多项式乘法: Pa=Pa× Pb, 并销毁一元多项式Pb
  polynomial Pc;
  Position qa,qb;
  term a,b,c;
  InitList(Pc);
  qa=GetHead(Pa);
  qa=qa->next;
  while(qa)
  {
    a=GetCurElem(qa);
    qb=GetHead(Pb);
    qb=qb->next;
    while(qb)
    {
      b=GetCurElem(qb);
      c.coef=a.coef*b.coef;
      c.expn=a.expn+b.expn;
      OrderInsertMerge(Pc,c,cmp);
      qb=qb->next;
    }
    qa=qa->next;
  }
  DestroyPolyn(Pb); // 销毁Pb
  ClearList(Pa); // 将Pa重置为空表
```

```
  Pa.head=Pc.head;
  Pa.tail=Pc.tail;
  Pa.len=Pc.len;
}

// main2-7.cpp 检验bo2-7.cpp的主程序
#include"c1.h"
#include"c2-6.h"
#include"bo2-7.cpp"
void main()
{
  polynomial p,q;
  int m;
  printf("请输入第1个一元多项式的非零项的个数: ");
  scanf("%d",&m);
  CreatPolyn(p,m);
  printf("请输入第2个一元多项式的非零项的个数: ");
  scanf("%d",&m);
  CreatPolyn(q,m);
  AddPolyn(p,q);
  printf("2个一元多项式相加的结果: \n");
  PrintPolyn(p);
  printf("请输入第3个一元多项式的非零项的个数: ");
  scanf("%d",&m);
  CreatPolyn(q,m);
  AddPolyn1(p,q);
  printf(2个一元多项式相加的结果(另一种方法): \n");
  PrintPolyn(p);
  printf("请输入第4个一元多项式的非零项的个数: ");
  scanf("%d",&m);
  CreatPolyn(q,m);
  SubtractPolyn(p,q);
  printf("2个一元多项式相减的结果: \n");
  PrintPolyn(p);
  printf("请输入第5个一元多项式的非零项的个数: ");
  scanf("%d",&m);
  CreatPolyn(q,m);
  MultiplyPolyn(p,q);
  printf("2个一元多项式相乘的结果: \n");
  PrintPolyn(p);
  DestroyPolyn(p);
}
```

程序运行结果:

```
请输入第1个一元多项式的非零项的个数: 3↙
请依次输入3个系数, 指数:
1,2↙
5,4↙
```

```
3,3↙
请输入第2个一元多项式的非零项的个数：3↙
请依次输入3个系数，指数：
-3,3↙
4,2↙
7,1↙
2个一元多项式相加的结果：
   系数      指数
7.000000   1
5.000000   2
5.000000   4
请输入第3个一元多项式的非零项的个数：3↙
请依次输入3个系数，指数：
-5,2↙
3,3↙
-3,1↙
2个一元多项式相加的结果(另一种方法)：
   系数      指数
4.000000   1
3.000000   3
5.000000   4
请输入第4个一元多项式的非零项的个数：3↙
请依次输入3个系数，指数：
4,1↙
2,3↙
6,6↙
2个一元多项式相减的结果：
   系数      指数
1.000000   3
5.000000   4
-6.000000   6
请输入第5个一元多项式的非零项的个数：2↙
请依次输入2个系数，指数：
1,1↙
2,2↙
2个一元多项式相乘的结果：
   系数      指数
1.000000   4
7.000000   5
10.000000   6
-6.000000   7
-12.000000   8
```

第3章　栈 和 队 列

3.1　栈

3.1.1　抽象数据类型栈的定义

3.1.2　栈的表示和实现

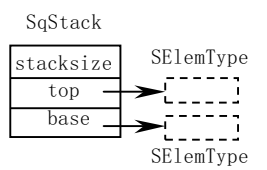


图 3－1　顺序栈存储结构

```
// c3-1.h 栈的顺序存储结构(见图3－1)
#define STACK_INIT_SIZE 10 // 存储空间初始分配量
#define STACK_INCREMENT 2 // 存储空间分配增量
struct SqStack
{
  SElemType *base; // 在栈构造之前和销毁之后，base的值为NULL
  SElemType *top; // 栈顶指针
  int stacksize; // 当前已分配的存储空间，以元素为单位
}; // 顺序栈
```

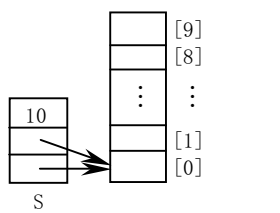


图 3－2　构造一个空的顺序栈 S

```
// bo3-1.cpp 顺序栈(存储结构由c3-1.h定义)的基本操作(9个)
void InitStack(SqStack &S)
{ // 构造一个空栈S(见图3－2)
  if(!(S.base=(SElemType *)malloc(STACK_INIT_SIZE*sizeof(SElemType))))
    exit(OVERFLOW); // 存储分配失败
  S.top=S.base;
  S.stacksize=STACK_INIT_SIZE;
}
void DestroyStack(SqStack &S)
{ // 销毁栈S，S不再存在(见图3－3)
  free(S.base);
  S.base=NULL;
  S.top=NULL;
  S.stacksize=0;
}
void ClearStack(SqStack &S)
{ // 把S置为空栈
  S.top=S.base;
}
Status StackEmpty(SqStack S)
{ // 若栈S为空栈，则返回TRUE；否则返回FALSE
  if(S.top==S.base)
```

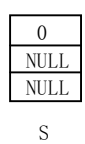
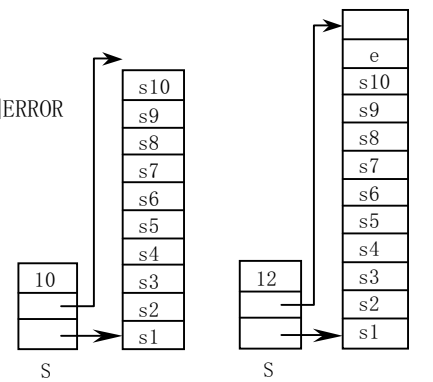


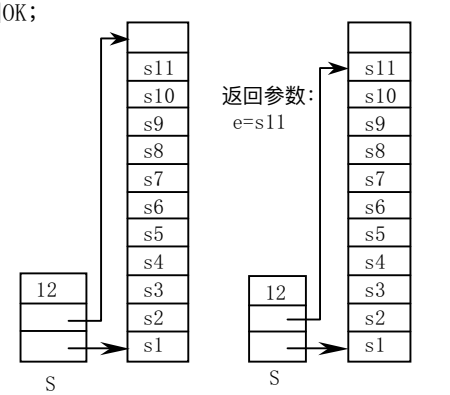
图 3－3　销毁顺序栈 S

```
    return TRUE;
  else
    return FALSE;
}
int StackLength(SqStack S)
{ // 返回S的元素个数，即栈的长度
  return S.top-S.base;
}
Status GetTop(SqStack S,SElemType &e)
{ // 若栈不空，则用e返回S的栈顶元素，并返回OK；否则返回ERROR
  if(S.top>S.base)
  {
    e=*(S.top-1);
    return OK;
  }
  else
    return ERROR;
}
void Push(SqStack &S,SElemType e)
{ // 插入元素e为新的栈顶元素(见图3-4)
  if(S.top-S.base>=S.stacksize) // 栈满，追加存储空间
  {
    S.base=(SElemType *)realloc(S.base,(S.stacksize+STACK_INCREMENT)*sizeof(SElemType));
    if(!S.base)
      exit(OVERFLOW); // 存储分配失败
    S.top=S.base+S.stacksize;
    S.stacksize+=STACK_INCREMENT;
  }
  *(S.top)++=e;
}
Status Pop(SqStack &S,SElemType &e)
{ // 若栈不空，则删除S的栈顶元素，用e返回其值，并返回OK；
  // 否则返回ERROR(见图3-5)
  if(S.top==S.base)
    return ERROR;
  e=*--S.top;
  return OK;
}
void StackTraverse(SqStack S,void(*visit)(SElemType))
{ // 从栈底到栈顶依次对栈中每个元素调用函数visit()
  while(S.top>S.base)
    visit(*S.base++);
  printf("\n");
}
```



(a) 调用 Push()之前　(b) 调用 Push()之后

图 3-4　调用 Push()示例



(a) 调用 Pop()之前　(b) 调用 Pop()之后

图 3-5　调用 Pop()示例

```
// main3-1.cpp 检验bo3-1.cpp的主程序
#include"c1.h"
typedef int SElemType; // 定义栈元素类型，此句要在c3-1.h的前面
#include"c3-1.h"
#include"bo3-1.cpp"
void print(SElemType c)
```

```
{
  printf("%d ",c);
}
void main()
{
  int j;
  SqStack s;
  SElemType e;
  InitStack(s);
  for(j=1;j<=12;j++)
    Push(s,j);
  printf("栈中元素依次为");
  StackTraverse(s,print);
  Pop(s,e);
  printf("弹出的栈顶元素 e=%d\n",e);
  printf("栈空否: %d(1:空 0:否)\n",StackEmpty(s));
  GetTop(s,e);
  printf("栈顶元素 e=%d 栈的长度为%d\n",e,StackLength(s));
  ClearStack(s);
  printf("清空栈后, 栈空否: %d(1:空 0:否)\n",StackEmpty(s));
  DestroyStack(s);
  printf("销毁栈后, s.top=%u s.base=%u s.stacksize=%d\n",s.top,s.base, s.stacksize);
}
```

程序运行结果:

```
栈中元素依次为1 2 3 4 5 6 7 8 9 10 11 12
弹出的栈顶元素 e=12
栈空否: 0(1:空 0:否)
栈顶元素 e=11 栈的长度为11
清空栈后, 栈空否: 1(1:空 0:否)
销毁栈后, s.top=0 s.base=0 s.stacksize=0
```

栈也是线性表，是操作受限的线性表。栈的操作是线性表操作的子集。因此，也可以将线性表的结构作为栈的结构。例如，可把不带头结点的线性单链表结构(见图 2－12)作为链栈的结构，如图 3－6 所示。这样，线性单链表的一些基本操作(在 bo2-8.cpp 中)就可以直接用于链栈的操作了。例如，初始化链表和初始化链栈的操作是一样的，就不必定义 InitStack()函数，可通过“#define InitStack InitList”命令直接把 InitList()函数当作 InitStack()函数使用。同时把栈元素 SElemType 定义为线性表元素 ElemType。线性表的另一些基本操作，如 ListInsert()也可以作为栈的基本操作 Push()来使用(取特例 i=1，即在第 1 个元素之前插入)。由于栈的操作被限定仅在栈顶进行，显然，令表头为栈顶可简化栈的操作。教科书对栈的定义是：限定仅在表尾进行插入或删除操作的线性表(教科书 44 页)。

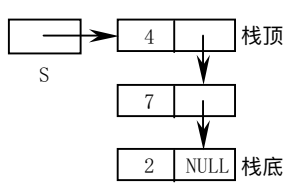


图 3－6　具有 3 个栈元素(4, 7, 2)的链栈 S

许多书上也都是这样定义的。对链栈，这样定义就不恰当。更准确的定义是：限定仅在表的一端进行插入或删除操作的线性表。bo3-5.cpp 是链栈的基本操作，其中有部分函数是通过 bo2-8.cpp 中的相关函数改名得来，还有部分函数是通过在特例下(如令形参 i=1，处理首元结点)调用 bo2-8.cpp 中的相关函数得来。在文件 main3-5.cpp 中，调用栈的基本操作函数仍可使用 StackLength()、Pop()等函数名。这就把线性表的一些操作函数的应用范围扩展到了栈的领域，减少了编写栈的基本操作函数的工作量。栈是一种线性表，因此对栈的操作是在一定条件下对线性表的操作。

```
// bo3-5.cpp 链栈(存储结构由c2-2.h定义)的基本操作(4个)
// 部分基本操作是由bo2-8.cpp中的函数改名得来
// 另一部分基本操作是由调用bo2-8.cpp中的函数(取特例)得来
typedef SElemType ElemType; // 栈结点类型和链表结点类型一致
#include"c2-2.h" // 单链表存储结构
typedef LinkList LinkStack; // LinkStack是指向栈结点的指针类型
#define InitStack InitList // InitStack()与InitList()作用相同，下同
#define DestroyStack DestroyList
#define ClearStack ClearList
#define StackEmpty ListEmpty
#define StackLength ListLength
#include"bo2-8.cpp" // 无头结点单链表的基本操作
Status GetTop(LinkStack S,SElemType &e)
{ // 若栈不空，则用e返回S的栈顶元素，并返回OK；否则返回ERROR
  return GetElem(S,1,e);
}
Status Push(LinkStack &S,SElemType e)
{ // 插入元素e为新的栈顶元素
  return ListInsert(S,1,e);
}
Status Pop(LinkStack &S,SElemType &e)
{ // 若栈不空，则删除S的栈顶元素，用e返回其值，并返回OK；否则返回ERROR
  return ListDelete(S,1,e);
}
void StackTraverse(LinkStack S,void(*visit)(SElemType))
{ // 从栈底到栈顶依次对栈中每个元素调用函数visit()
  LinkStack temp,p=S; // p指向栈顶元素
  InitStack(temp); // 初始化临时栈temp
  while(p)
  {
    Push(temp,p->data); // 由S栈顶到栈底，依次将栈元素入栈到temp栈
    p=p->next;
  }
  ListTraverse(temp,visit); // 遍历temp线性表
}

// main3-5.cpp 检验bo3-5.cpp的主程序
#include"c1.h"
typedef int SElemType; // 定义栈元素的类型
#include"bo3-5.cpp"
void print(SElemType c)
```

```
{
  printf("%d ",c);
}
void main()
{
  int j;
  LinkStack s;
  SElemType e;
  InitStack(s); // 初始化栈s
  for(j=1;j<=5;j++) // 将2,4,6,8,10入栈
    Push(s,2*j);
  printf("栈中的元素从栈底到栈顶依次为");
  StackTraverse(s,print);
  Pop(s,e);
  printf("弹出的栈顶元素为%d\n",e);
  printf("栈空否: %d(1:空 0:否)\n",StackEmpty(s));
  GetTop(s,e);
  printf("当前栈顶元素为%d, 栈的长度为%d\n",e,StackLength(s));
  ClearStack(s);
  printf("清空栈后, 栈空否: %d(1:空 0:否), 栈的长度为%d\n",StackEmpty(s),StackLength(s));
  DestroyStack(s);
}
```

程序运行结果:

```
栈中的元素从栈底到栈顶依次为 2 4 6 8 10
弹出的栈顶元素为10
栈空否: 0(1:空 0:否)
当前栈顶元素为8, 栈的长度为4
清空栈后, 栈空否: 1(1:空 0:否), 栈的长度为0
```

由于栈只在表的一端进行插入和删除的操作, 采用顺序存储结构(c3-1.h 定义), 在入栈和出栈时也不需要移动栈中元素。故顺序栈比链栈的效率要高一些。

3.2　栈的应用举例

3.2.1　数制转换

```
// algo3-1.cpp 调用算法3.1的程序
#define N 8 // 定义待转换的进制N(二进制～九进制)
typedef int SElemType; // 定义栈元素类型为整型
#include"c1.h"
#include"c3-1.h" // 采用顺序栈
#include"bo3-1.cpp" // 利用顺序栈的基本操作
void conversion() // 算法3.1
{ // 对于输入的任意一个非负十进制整数, 打印输出与其等值的N进制数
  SqStack s;
  unsigned n; // 非负整数
```

```
  SElemType e;
  InitStack(s); // 初始化栈
  printf("将十进制整数n转换为%d进制数，请输入：n(>=0)=",N);
  scanf("%u",&n); // 输入非负十进制整数n
  while(n) // 当n不等于0
  {
    Push(s,n%N); // 入栈n除以N的余数(N进制的低位)
    n=n/N;
  }
  while(!StackEmpty(s)) // 当栈不空
  {
    Pop(s,e); // 弹出栈顶元素且赋值给e
    printf("%d",e); // 输出e
  }
  printf("\n");
}
void main()
{
  conversion();
}
```

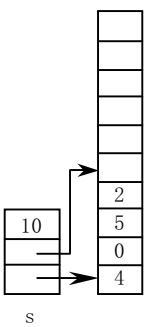


图 3－7　栈 s 在元素最多时的状态

程序运行结果(见图 3－7)：

```
将十进制整数 n 转换为 8 进制数，请输入：n(>=0)=1348↙
2504
```

如果将 N 定义为 2，algo3-1.cpp 就是将十进制数转换为二进制数的程序。

程序运行结果：

```
将十进制整数 n 转换为 2 进制数，请输入：n(>=0)=13↙
1101
```

algo3-1.cpp 能不能用于十进制到十六进制的转换呢？存在一个问题，要将余数 10～15 转换为 A～F 输出。algo3-2.cpp 实现了十进制到十六进制的转换。

```
// algo3-2.cpp 改算法3.1，十进制→十六进制
typedef int SElemType; // 定义栈元素类型为整型
#include"c1.h"
#include"c3-1.h" // 采用顺序栈
#include"bo3-1.cpp" // 利用顺序栈的基本操作
void conversion()
{ // 对于输入的任意一个非负十进制整数，打印输出与其等值的十六进制数
  SqStack s;
  unsigned n; // 非负整数
```

```
  SElemType e;
  InitStack(s); // 初始化栈
  printf("将十进制整数n转换为十六进制数，请输入：n(>=0)=");
  scanf("%u",&n); // 输入非负十进制整数n
  while(n) // 当n不等于0
  {
    Push(s,n%16); // 入栈n除以16的余数(十六进制的低位)
    n=n/16;
  }
  while(!StackEmpty(s)) // 当栈不空
  {
    Pop(s,e); // 弹出栈顶元素且赋值给e
    if(e<=9)
      printf("%d",e);
    else
      printf("%c",e+55); // 大于9的余数，输出相应的字符
  }
  printf("\n");
}
void main()
{
  conversion();
}
```

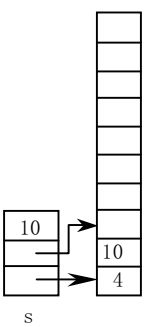


图 3－8　栈 s 在元素最多时的状态

程序运行结果(见图 3－8)：

```
将十进制整数n转换为十六进制数，请输入：n(>=0)=164↙
A4
```

3.2.2　括号匹配的检验

检验括号匹配的方法，就是对给定的字符串依次检验：若是左括号，入栈；若是右括号，出栈一个左括号判断是否与之相匹配；是其它字符，不检验。检验到字符串尾，还要检查栈是否空。只有栈空，整个字符串才匹配完。

```
// algo3-3.cpp 括号（()、[]和{}）匹配的检验
typedef char SElemType;
#include"c1.h"
#include"c3-1.h"
#include"bo3-1.cpp"
void check()
{ // 对于输入的任意一个字符串，检验括号是否配对
  SqStack s;
  SElemType ch[80],*p,e;
  InitStack(s); // 初始化栈成功
  printf("请输入带括号（()、[]和{}）的表达式\n");
```

```
  gets(ch);
  p=ch; // p指向字符串的首字符
  while(*p) // 没到串尾
    switch(*p)
    {
      case '(':
      case '[':
      case '{':Push(s,*p++); // 左括号入栈，且p++
               break;
      case ')':
      case ']':
      case '}':if(!StackEmpty(s)) // 栈不空
               {
                 Pop(s,e); // 弹出栈顶元素
                 if(!(e=='('&&*p==')'||e=='['&&*p==']'||e=='{'&&*p=='}'))
                 { // 出现3种匹配情况之外的情况
                   printf("左右括号不配对\n");
                   exit(ERROR);
                 }
               }
               else // 栈空
               {
                 printf("缺乏左括号\n");
                 exit(ERROR);
               }
      default: p++; // 其它字符不处理，指针向后移
    }
  if(StackEmpty(s)) // 字符串结束时栈空
    printf("括号匹配\n");
  else
    printf("缺乏右括号\n");
}
void main()
{
  check();
}
```

程序运行结果：

```
请输入带括号（()、[]和{}）的表达式
{[(5-2)*(7-3)+2]*4+8}*6↙
括号匹配
```

3.2.3　行编辑程序

```
// algo3-4.cpp 行编辑程序，实现算法3.2
typedef char SElemType;
```

```
#include"c1.h"
#include"c3-1.h"
#include"bo3-1.cpp"
FILE *fp;
void copy(SElemType c)
{ // 将字符c送至fp所指的文件中
  fputc(c,fp);
}
void LineEdit()
{ // 利用字符栈s，从终端接收一行并送至调用过程的数据区。算法3.2
  SqStack s;
  char ch;
  InitStack(s);
  printf("请输入一个文本文件,^Z结束输入:\n");
  ch=getchar();
  while(ch!=EOF)
  { // 当全文没结束(EOF为^Z键，全文结束符)
    while(ch!=EOF&&ch!='\n')
    { // 当全文没结束且没到行末(不是换行符)
      switch(ch)
      {
        case '#':if(!StackEmpty(s))
                   Pop(s,ch); // 仅当栈非空时退栈,c可由ch替代
                 break;
        case '@':ClearStack(s); // 重置s为空栈
                 break;
        default :Push(s,ch); // 其它字符进栈
      }
      ch=getchar(); // 从终端接收下一个字符
    }
    StackTraverse(s,copy); // 将从栈底到栈顶的栈内字符传送至文件
    fputc('\n',fp); // 向文件输入一个换行符
    ClearStack(s); // 重置s为空栈
    if(ch!=EOF)
      ch=getchar();
  }
  DestroyStack(s);
}
void main()
{
  fp=fopen("ed.txt","w"); // 在当前目录下建立ed.txt文件，用于写数据，
  if(fp)                  // 如已有同名文件则先删除原文件
  {
    LineEdit();
    fclose(fp); // 关闭fp所指的文件
  }
  else
    printf("建立文件失败!\n");
}
```

程序运行结果(以教科书 49 页下的输入为例)：

```
请输入一个文本文件,^Z结束输入:
whli##ilr#e(s#*s)↙

  outcha@putchar(*s=#++);↙

^Z↙
```

文件 ed.txt 的内容：

```
while(*s)
putchar(*s++);
```

3.2.4　迷宫求解

```
// func3-1.cpp、algo3-5.cpp、algo3-9.cpp和algo3-11.cpp要调用的函数、结构和全局变量
struct PosType // 迷宫坐标位置类型(见图3－9)
{
  int x; // 行值
  int y; // 列值
};
#define MAXLENGTH 25 // 设迷宫的最大行列为25
typedef int MazeType[MAXLENGTH][MAXLENGTH]; // 迷宫数组类型[行][列]
// 全局变量
MazeType m; // 迷宫数组
int x,y; // 迷宫的行数, 列数
PosType begin,end; // 迷宫的入口坐标,出口坐标
void Print()
{ // 输出迷宫的解(m数组)
  int i,j;
  for(i=0;i<x;i++)
  {
    for(j=0;j<y;j++)
      printf("%3d",m[i][j]);
    printf("\n");
  }
}
void Init(int k)
{ // 设定迷宫布局(墙为值0,通道值为k)
  int i,j,x1,y1;
  printf("请输入迷宫的行数,列数(包括外墙): ");
  scanf("%d,%d",&x,&y);
  for(i=0;i<x;i++) // 定义周边值为0(外墙)
  {
    m[0][i]=0; // 行周边
    m[x-1][i]=0;
  }
```

PosType

图 3－9　PosType 结构

```
  for(i=0;i<y-1;i++)
  {
    m[i][0]=0; // 列周边
    m[i][y-1]=0;
  }
  for(i=1;i<x-1;i++)
    for(j=1;j<y-1;j++)
      m[i][j]=k; // 定义除外墙，其余都是通道，初值为k
  printf("请输入迷宫内墙单元数: ");
  scanf("%d",&j);
  printf("请依次输入迷宫内墙每个单元的行数,列数: \n");
  for(i=1;i<=j;i++)
  {
    scanf("%d,%d",&x1,&y1);
    m[x1][y1]=0; // 修改墙的值为0
  }
  printf("迷宫结构如下:\n");
  Print();
  printf("请输入入口的行数,列数: ");
  scanf("%d,%d",&begin.x,&begin.y);
  printf("请输入出口的行数,列数: ");
  scanf("%d,%d",&end.x,&end.y);
}
```

由于把迷宫数组 m、迷宫的行数 x、列数 y、迷宫的入口坐标 begin 和出口坐标 end 作为全局变量，它们在各函数中都通用，不需要用形参来传递，减少了 Print()函数和 Init()函数中的形参数量。

```
// algo3-5.cpp 利用栈求解迷宫问题(只输出一个解，算法3.3)
#include"c1.h"
#include"func3-1.cpp"
int curstep=1; // 当前足迹，初值(在入口处)为1
struct SElemType // 栈的元素类型(见图3－10)
{
  int ord; // 通道块在路径上的"序号"
  PosType seat; // 通道块在迷宫中的"坐标位置"
  int di; // 从此通道块走向下一通道块的"方向"(0～3表示东～北)
};
#include"c3-1.h" // 采用顺序栈存储结构
#include"bo3-1.cpp" // 采用顺序栈的基本操作函数
// 定义墙元素值为0，可通过路径为1，不能通过路径为-1，通过路径为足迹
Status Pass(PosType b)
{ // 当迷宫m的b点的序号为1(可通过路径)，返回OK；否则，返回ERROR
  if(m[b.x][b.y]==1)
    return OK;
  else
    return ERROR;
}
void FootPrint(PosType a)
```

SElemType

ord	seat.x	seat.y	di

图 3－10　SElemType 结构

```
{ // 使迷宫m的a点的值变为足迹(curstep)
  m[a.x][a.y]=curstep;
}
void NextPos(PosType &c,int di)
{ // 根据当前位置及移动方向，求得下一位置
  PosType direc[4]={{0,1},{1,0},{0,-1},{-1,0}}; // {行增量,列增量},移动方向,依次为东南西北
  c.x+=direc[di].x;
  c.y+=direc[di].y;
}
void MarkPrint(PosType b)
{ // 使迷宫m的b点的序号变为-1(不能通过的路径)
  m[b.x][b.y]=-1;
}
Status MazePath(PosType start,PosType end) // 算法3.3
{ // 若迷宫m中存在从入口start到出口end的通道，则求得一条
  // 存放在栈中(从栈底到栈顶)，并返回TRUE；否则返回FALSE
  SqStack S; // 顺序栈
  PosType curpos; // 当前位置
  SElemType e; // 栈元素
  InitStack(S); // 初始化栈
  curpos=start; // 当前位置在入口
  do
  {
    if(Pass(curpos))
    { // 当前位置可以通过，即是未曾走到过的通道块
      FootPrint(curpos); // 留下足迹
      e.ord=curstep;
      e.seat=curpos;
      e.di=0;
      Push(S,e); // 入栈当前位置及状态
      curstep++; // 足迹加1
      if(curpos.x==end.x&&curpos.y==end.y) // 到达终点(出口)
        return TRUE;
      NextPos(curpos,e.di); // 由当前位置及移动方向，确定下一个当前位置
    }
    else
    { // 当前位置不能通过
      if(!StackEmpty(S)) // 栈不空
      {
        Pop(S,e); // 退栈到前一位置
        curstep--; // 足迹减1
        while(e.di==3&&!StackEmpty(S)) // 前一位置处于最后一个方向(北)
        {
          MarkPrint(e.seat); // 在前一位置留下不能通过的标记(-1)
          Pop(S,e); // 再退回一步
          curstep--; // 足迹再减1
        }
        if(e.di<3) // 没到最后一个方向(北)
        {
          e.di++; // 换下一个方向探索
```

```
            Push(S,e); // 入栈该位置的下一个方向
            curstep++; // 足迹加1
            curpos=e.seat; // 确定当前位置
            NextPos(curpos,e.di); // 确定下一个当前位置是该新方向上的相邻块
          }
        }
      }
    }while(!StackEmpty(S));
    return FALSE;
}
void main()
{
    Init(1); // 初始化迷宫，通道值为1
    if(MazePath(begin,end)) // 有通路
    {
      printf("此迷宫从入口到出口的一条路径如下:\n");
      Print(); // 输出此通路
    }
    else
      printf("此迷宫没有从入口到出口的路径\n");
}
```

程序运行结果(以教科书图 3.4 为例):

```
请输入迷宫的行数,列数(包括外墙): 10,10↙
请输入迷宫内墙单元数: 18↙
请依次输入迷宫内墙每个单元的行数,列数:
1,3↙
1,7↙
2,3↙
2,7↙
3,5↙
3,6↙
4,2↙
4,3↙
4,4↙
5,4↙
6,2↙
6,6↙
7,2↙
7,3↙
7,4↙
7,6↙
7,7↙
8,1↙
迷宫结构如下:
  0  0  0  0  0  0  0  0  0  0
  0  1  1  0  1  1  1  0  1  0
  0  1  1  0  1  1  1  0  1  0
```

```
  0  1  1  1  1  0  0  1  1  0
  0  1  0  0  0  1  1  1  1  0
  0  1  1  1  0  1  1  1  1  0
  0  1  0  1  1  1  0  1  1  0
  0  1  0  0  0  1  0  0  1  0
  0  0  1  1  1  1  1  1  1  0
  0  0  0  0  0  0  0  0  0  0
请输入入口的行数,列数: 1,1↙
请输入出口的行数,列数: 8,8↙
此迷宫从入口到出口的一条路径如下:(见图3-11)
  0  0  0  0  0  0  0  0  0  0
  0  1  2  0 -1 -1 -1  0  1  0
  0  1  3  0 -1 -1 -1  0  1  0
  0  5  4 -1 -1  0  0  1  1  0
  0  6  0  0  0  1  1  1  1  0
  0  7  8  9  0  1  1  1  1  0
  0  1  0 10 11 12  0  1  1  0
  0  1  0  0  0 13  0  0  1  0
  0  0  1  1  1 14 15 16 17  0
  0  0  0  0  0  0  0  0  0  0
```

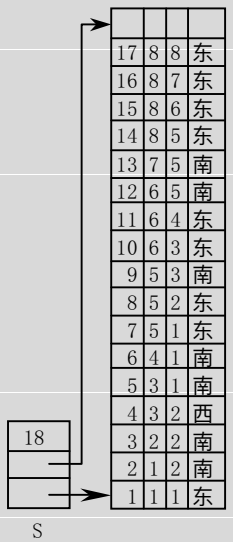


图 3-11　到达终点时栈 S 的内容

从入口到出口的一条路径由全局变量 m 数组显示。入口的值为 1，路径的值依次为 2, 3, …, 17。图 3-11 显示了到达终点时栈的内容。其中栈元素的 di 成员的值本是 0～3，为直观，用东～北代替。由于有数组 m，栈的主要作用并不是保存路径。它的主要作用是当试探失败时，通过退栈回到前一点，从前一点再继续试探。

这条路径共走了 16 步，从入口(第 1 步)到第 5 步其实只需走 2 步。原因是 NextPos()函数的第 1 句定义 direc 数组的移动方向依次为东南西北。程序由入口处先向东试探。既然有路，就不再向南试探了。因此走了弯路。如果将 direc[0] 和 direc[1]的值交换一下，新的移动方向依次为南东西北。这样，程序先向南试探，只在不成功时才向东试探。避免了走弯路，14 步就到了出口。以下是修改了 direc 数组的值(改 NextPos()函数的第 1 句为 PosType direc[4]={{1, 0}, {0, 1}, {0, -1}, {-1, 0}};)，不改变输入数据的情况下的运行结果。为节约篇幅，略去相同部分，只列出最后结果。

程序运行结果：

```
此迷宫从入口到出口的一条路径如下:
  0  0  0  0  0  0  0  0  0  0
  0  1  1  0  1  1  1  0  1  0
  0  2  1  0  1  1  1  0  1  0
  0  3  1  1  1  0  0  1  1  0
  0  4  0  0  0  1  1  1  1  0
  0  5  6  7  0  1  1  1  1  0
  0 -1  0  8  9 10  0  1  1  0
  0 -1  0  0  0 11  0  0  1  0
  0  0  1  1  1 12 13 14 15  0
  0  0  0  0  0  0  0  0  0  0
```

m 数组中，-1 表示曾试探过，但不能通行又退回去的路径；除入口以外，1 表示没有走过的路径；大于 1 的值表示由入口通向出口的足迹。上面 m 数组第 5 步的下面有 2 个-1。它们的值本是 1(通道)。当走到第 5 步时，入栈{5,5,1,0}，说明第 5 步走在 5 行 1 列，且下一步是向南走。向南走是通路，将 m 数组 6 行 1 列的值改为 6(留下足迹)，入栈{6,6,1,0}，当前足迹 curstep 加 1 为 7。说明第 6 步走在 6 行 1 列，且下一步也是向南走。还是通路，将 m 数组 7 行 1 列的值改为 7(将 curstep 赋给 m[7][1])，入栈{7,7,1,0}，当前足迹 curstep 加 1 为 8。说明第 7 步走在 7 行 1 列，且下一步也是向南走。这时不再是通路，出栈{7,7,1,0}。改变方向，入栈{7,7,1,1}。说明第 7 步仍是走在 7 行 1 列，下一步改为向东走。依然没有通路。7 行 1 列的 4 个方向上都不是 1(走投无路)。最后，将 m 数组 7 行 1 列的值改为-1，出栈{7,7,1,3}，当前足迹 curstep 减 1 为 6。再出栈{6,6,1,0}。改变方向，入栈{6,6,1,1}。6 行 1 列的 4 个方向上也都不是 1。最后，将 m 数组 6 行 1 列的值改为-1，出栈{6,6,1,3}，当前足迹 curstep 减 1 为 5。再出栈{5,5,1,0}。改变方向，入栈{5,5,1,1}。向第 5 步的东面试探第 6 步，……，直到终点或没有通路。

3.2.5　表达式求值

```
// func3-2.cpp algo3-6.cpp和algo3-7.cpp要调用的函数
char Precede(SElemType t1,SElemType t2)
{ // 根据教科书表3.1，判断t1，t2两符号的优先关系('#'用'\n'代替)
  char f;
  switch(t2)
  {
    case '+':
    case '-':if(t1=='('||t1=='\n')
               f='<'; // t1<t2
             else
               f='>'; // t1>t2
             break;
    case '*':
    case '/':if(t1=='*'||t1=='/'||t1==')')
               f='>'; // t1>t2
             else
               f='<'; // t1<t2
             break;
    case '(':if(t1==')')
             {
               printf("括号不匹配\n");
               exit(ERROR);
             }
             else
               f='<'; // t1<t2
             break;
    case ')':switch(t1)
             {
```

```
                case '(':f='<'; // t1=t2
                        break;
                case '\n':printf("缺乏左括号\n");
                        exit(ERROR);
                        default :f='>'; // t1>t2
              }
              break;
    case'\n':switch(t1)
              {
                case'\n':f='='; // t1=t2
                        break;
                case'(' :printf("缺乏右括号\n");
                        exit(ERROR);
                default :f='>'; // t1>t2
              }
  }
  return f;
}
Status In(SElemType c)
{ // 判断c是否为7种运算符之一
  switch(c)
  {
    case'+':
    case'-':
    case'*':
    case'/':
    case'(':
    case')':
    case'\n':return TRUE;
    default :return FALSE;
  }
}
SElemType Operate(SElemType a,SElemType theta,SElemType b)
{ // 做四则运算a theta b，返回运算结果
  switch(theta)
  {
    case'+':return a+b;
    case'-':return a-b;
    case'*':return a*b;
  }
  return a/b;
}

// algo3-6.cpp 表达式求值(输入的值在0～9之间，中间结果和输出的值在-128～127之间)，算法3.4
typedef char SElemType; // 栈元素为字符型
#include"c1.h"
#include"c3-1.h"
#include"bo3-1.cpp"
#include"func3-2.cpp"
```

```
SElemType EvaluateExpression() // 算法3.4，有改动
{ // 算术表达式求值的算符优先算法。设OPTR和OPND分别为运算符栈和运算数栈
  SqStack OPTR,OPND;
  SElemType a,b,c,x;
  InitStack(OPTR); // 初始化运算符栈OPTR和运算数栈OPND
  InitStack(OPND);
  Push(OPTR,'\n'); // 将换行符压入运算符栈OPTR的栈底(改)
  c=getchar(); // 由键盘读入1个字符到c
  GetTop(OPTR,x); // 将运算符栈OPTR的栈顶元素赋给x
  while(c!='\n'||x!='\n') // c和x不都是换行符
  {
    if(In(c)) // c是7种运算符之一
      switch(Precede(x,c)) // 判断x和c的优先权
      {
        case'<':Push(OPTR,c); // 栈顶元素x的优先权低，入栈c
                c=getchar();  // 由键盘读入下一个字符到c
                break;
        case'=':Pop(OPTR,x); // x='('且c=')'情况，弹出'('给x(后又扔掉)
                c=getchar(); // 由键盘读入下一个字符到c(扔掉')')
                break;
        case'>':Pop(OPTR,x); // 栈顶元素x的优先权高，弹出运算符栈OPTR的栈顶元素给x(改)
                Pop(OPND,b); // 依次弹出运算数栈OPND的栈顶元素给b，a
                Pop(OPND,a);
                Push(OPND,Operate(a,x,b)); // 做运算a x b，并将运算结果入运算数栈
      }
    else if(c>='0'&&c<='9') // c是操作数
    {
      Push(OPND,c-48); // 将该操作数的值(不是ASCII码)压入运算数栈OPND
      c=getchar(); // 由键盘读入下一个字符到c
    }
    else // c是非法字符
    {
      printf("出现非法字符\n");
      exit(ERROR);
    }
    GetTop(OPTR,x); // 将运算符栈OPTR的栈顶元素赋给x
  }
  Pop(OPND,x); // 弹出运算数栈OPND的栈顶元素(运算结果)给x(改此处)
  if(!StackEmpty(OPND)) // 运算数栈OPND不空(运算符栈OPTR仅剩'\n')
  {
    printf("表达式不正确\n");
    exit(ERROR);
  }
  return x;
}
void main()
{
  printf("请输入算术表达式(输入的值要在0～9之间、中间运算值和输出结果在-128～127之间)\n");
  printf("%d\n",EvaluateExpression()); // 返回值(8位二进制，1个字节)按整型格式输出
}
```

程序运行结果(以教科书例 3－1 为例)：

```
请输入算术表达式(输入的值要在0～9之间、中间运算值和输出结果在-128～127之间)
3*(7-2)↙
15
```

algo3-7.cpp 对 algo3-6.cpp 做了些改进，把连续输入的几个数值型字符作为一个整数处理。使数值范围扩大为整型，更具有实用性。

```
// algo3-7.cpp 表达式求值(范围为int类型，输入负数要用(0-正数)表示)
typedef int SElemType; // 栈元素类型为整型，改algo3-6.cpp
#include"c1.h"
#include"c3-1.h" // 顺序栈的存储结构
#include"bo3-1.cpp" // 顺序栈的基本操作
#include"func3-2.cpp"
SElemType EvaluateExpression()
{ // 算术表达式求值的算符优先算法。设OPTR和OPND分别为运算符栈和运算数栈
  SqStack OPTR,OPND;
  SElemType a,b,d,x; // 改algo3-6.cpp
  char c; // 存放由键盘接收的字符，改algo3-6.cpp
  char z[11]; // 存放整数字符串，改algo3-6.cpp
  int i; // 改algo3-6.cpp
  InitStack(OPTR); // 初始化运算符栈OPTR和运算数栈OPND
  InitStack(OPND);
  Push(OPTR,'\n'); // 将换行符压入运算符栈OPTR的栈底(改)
  c=getchar(); // 由键盘读入1个字符到c
  GetTop(OPTR,x); // 将运算符栈OPTR的栈顶元素赋给x
  while(c!='\n'||x!='\n') // c和x不都是换行符
  {
    if(In(c)) // c是7种运算符之一
      switch(Precede(x,c)) // 判断x和c的优先权
      {
        case'<' :Push(OPTR,c); // 栈顶元素x的优先权低，入栈c
                 c=getchar();  // 由键盘读入下一个字符到c
                 break;
        case'=' :Pop(OPTR,x); // x='('且c=')'情况，弹出'('给x(后又扔掉)
                 c=getchar(); // 由键盘读入下一个字符到c(扔掉')')
                 break;
        case'>' :Pop(OPTR,x); // 栈顶元素x的优先权高，弹出运算符栈OPTR的栈顶元素给x(改)
                 Pop(OPND,b); // 依次弹出运算数栈OPND的栈顶元素给b，a
                 Pop(OPND,a);
                 Push(OPND,Operate(a,x,b)); // 做运算a x b，并将运算结果入运算数栈
      }
    else if(c>='0'&&c<='9') // c是操作数，此语句改algo3-6.cpp
    {
      i=0;
      while(c>='0'&&c<='9') // 是连续数字
```

```
      {
        z[i++]=c;
        c=getchar();
      }
      z[i]=0; // 字符串结束符
      d=atoi(z); // 将z中保存的数值型字符串转为整型存于d
      Push(OPND,d); // 将d压入运算数栈OPND
    }
    else // c是非法字符，以下同algo3-6.cpp
    {
      printf("出现非法字符\n");
      exit(ERROR);
    }
    GetTop(OPTR,x); // 将运算符栈OPTR的栈顶元素赋给x
  }
  Pop(OPND,x); // 弹出运算数栈OPND的栈顶元素(运算结果)给x(改此处)
  if(!StackEmpty(OPND)) // 运算数栈OPND不空(运算符栈OPTR仅剩'\n')
  {
    printf("表达式不正确\n");
    exit(ERROR);
  }
  return x;
}
void main()
{
  printf("请输入算术表达式,负数要用(0-正数)表示\n");
  printf("%d\n",EvaluateExpression());
}
```

程序运行结果：

```
请输入算术表达式,负数要用(0-正数)表示
(0-12)*((5-3)*3+2)/(2+2)↙
-24
```

3.3　栈与递归的实现

函数中有直接或间接地调用自身函数的语句，这样的函数称为递归函数。递归函数用得好，可简化编程工作。但函数自己调用自己，有可能造成死循环。为了避免死循环，要做到两点：

(1) 降阶。递归函数虽然调用自身，但并不是简单地重复。它的实参值每次是不一样的。一般逐渐减小，称为降阶。如教科书式(3－3)的 Ackerman 函数，当 m≠0 时，求 Ack(m,n)可由 Ack(m-1,…)得到，Ack()函数的第 1 个参数减小了。

(2) 有出口。即在某种条件下，不再进行递归调用。仍以教科书式(3－3)的 Ackerman

函数为例，当 m=0 时，Ack(0,n)=n+1，终止了递归调用。所以，递归函数总有条件语句。m=0 的条件是由逐渐降阶形成的。如取 Ack(m,n)函数的实参 m=-1，即使通过降阶也不会出现 m=0 的情况，这就造成了死循环。

编译软件在遇到函数调用时，会将调用函数的实参、返回地址等信息存入软件自身所带的栈中，再去运行被调函数。从被调函数返回时，再将调用函数的信息出栈，接着运行调用函数。编译软件开辟的栈空间是有限的，当递归调用时，嵌套的层次往往很多，就有可能使栈发生溢出现象，从而出现不可预料的后果。运行 algo3-8.cpp 时，m、n 的取值就不可过大。

```
// algo3-8.cpp 用递归调用求Ackerman(m,n)的值
#include<stdio.h>
int ack(int m,int n)
{
  int z;
  if(m==0)
    z=n+1; // 出口
  else if(n==0)
    z=ack(m-1,1); // 对形参m降阶
  else
    z=ack(m-1,ack(m,n-1)); // 对形参m、n降阶
  return z;
}
void main()
{
  int m,n;
  printf("Please input m,n:");
  scanf("%d,%d",&m,&n);
  printf("Ack(%d,%d)=%d\n",m,n,ack(m,n));
}
```

程序运行结果(m、n 不可取值过大)：

```
Please input m,n:3,9↙
Ack(3,9)=4093
```

```
// algo3-9.cpp 用递归函数求解迷宫问题(求出所有解)
#include"c1.h" // 根据《PASCAL程序设计》(郑启华编著)中的程序改编
#include"func3-1.cpp" // 定义墙元素值为0，可通过路径为-1，通过路径为足迹
void Try(PosType cur,int curstep)
{ // 由当前位置cur、当前步骤curstep试探下一点
  int i;
  PosType next; // 下一个位置
  PosType direc[4]={{0,1},{1,0},{0,-1},{-1,0}}; // {行增量,列增量},移动方向,依次为东南西北
  for(i=0;i<=3;i++) // 依次试探东南西北四个方向
  {
    next.x=cur.x+direc[i].x; // 根据移动方向，给下一位置赋值
```

```
    next.y=cur.y+direc[i].y;
    if(m[next.x][next.y]==-1) // 下一个位置是通路
    {
      m[next.x][next.y]=++curstep; // 将下一个位置设为足迹
      if(next.x!=end.x||next.y!=end.y) // 没到终点
        Try(next,curstep); // 由下一个位置继续试探(降阶递归调用，离终点更近)
      else // 到终点
      {
        Print(); // 输出结果(出口，不再递归调用)
        printf("\n");
      }
      m[next.x][next.y]=-1; // 恢复为通路，以便在另一个方向试探另一条路
      curstep--; // 足迹也减1
    }
  }
}
void main()
{
  Init(-1); // 初始化迷宫，通道值为-1
  printf("此迷宫从入口到出口的路径如下:\n");
  m[begin.x][begin.y]=1; // 入口的足迹为1
  Try(begin,1); // 由第1步入口试探起
}
```

程序运行结果：

```
请输入迷宫的行数,列数(包括外墙): 5,5↙
请输入迷宫内墙单元数: 1↙
请依次输入迷宫内墙每个单元的行数,列数:
2,2↙
迷宫结构如下:
  0  0  0  0  0
  0 -1 -1 -1  0
  0 -1  0 -1  0
  0 -1 -1 -1  0
  0  0  0  0  0
请输入入口的行数,列数: 1,1↙
请输入出口的行数,列数: 3,3↙
此迷宫从入口到出口的路径如下:
  0  0  0  0  0
  0  1  2  3  0
  0 -1  0  4  0
  0 -1 -1  5  0
  0  0  0  0  0

  0  0  0  0  0
  0  1 -1 -1  0
  0  2  0 -1  0
  0  3  4  5  0
  0  0  0  0  0
```

此迷宫从入口到出口有 2 条路径。由入口(1 行 1 列足迹 1)处向 4 个方向试探，只有 2 个方向(东、南)是通路(-1)。首先朝东继续试探。足迹 2～4 都只有 1 个方向是通路，足迹 5 到达出口。输出路径且逐一恢复足迹为-1(通路)。恢复后的情况是除入口为 1 外，迷宫其它各点与初态相同，以便第 2 条路径(由入口向南)继续试探。

与 algo3-5.cpp 相比，algo3-9.cpp 程序简短，且没使用栈(递归函数使用了编译软件内设的栈)。

```
// algo3-10.cpp Hanoi塔问题，调用算法3.5的程序
#include<stdio.h>
int c=0; // 全局变量，搬动次数
void move(char x,int n,char z)
{ // 第n个圆盘从塔座x搬到塔座z
  printf("第%i步：将%i号盘从%c移到%c\n",++c,n,x,z);
}
void hanoi(int n,char x,char y,char z) // 算法3.5
{ // 将塔座x上按直径由小到大且自上而下编号为1至n的n个圆盘
  // 按规则搬到塔座z上。y可用作辅助塔座
  if(n==1) // (出口)
    move(x,1,z); // 将编号为1的圆盘从x移到z
  else
  {
    hanoi(n-1,x,z,y); // 将x上编号为1至n-1的圆盘移到y，z作辅助塔(降阶递归调用)
    move(x,n,z); // 将编号为n的圆盘从x移到z
    hanoi(n-1,y,x,z); // 将y上编号为1至n-1的圆盘移到z，x作辅助塔(降阶递归调用)
  }
}
void main()
{
  int n;
  printf("3个塔座为a、b、c，圆盘最初在a座，借助b座移到c座。请输入圆盘数：");
  scanf("%d",&n);
  hanoi(n,'a','b','c');
}
```

 　程序运行结果：

```
3个塔座为a、b、c，圆盘最初在a座，借助b座移到c座。请输入圆盘数：3↙
第1步：将1号盘从a移到c
第2步：将2号盘从a移到b
第3步：将1号盘从c移到b
第4步：将3号盘从a移到c
第5步：将1号盘从b移到a
第6步：将2号盘从b移到c
第7步：将1号盘从a移到c
```

3.4　队　列

3.4.1　抽象数据类型的定义

3.4.2　链队列——队列的链式表示和实现

```
// c3-2.h 单链队列－－队列的链式存储结构
typedef struct QNode // (见图3－12)
{
  QElemType data;
  QNode *next;
}*QueuePtr;
struct LinkQueue // (见图3－13)
{
  QueuePtr front,rear; // 队头、队尾指针
};
```

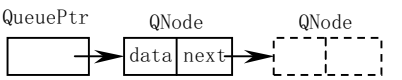


图 3－12　单链队列的结点类型

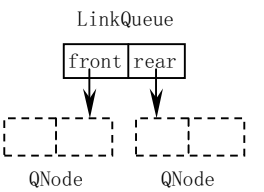


图 3－13　LinkQueue 类型

和栈一样，队列也是操作受限的线性表，只允许在队尾插入元素，在队头删除元素。对于链队列结构，为了便于插入元素，设立了队尾指针。这样，插入元素的操作与队列长度无关。图 3－14 是具有两个元素的链队列示例。

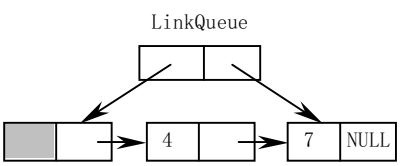


图 3－14　具有两个元素(4,7)的链队列

```
// bo3-2.cpp 链队列(存储结构由c3-2.h定义)的基本操作(9个)
void InitQueue(LinkQueue &Q)
{ // 构造一个空队列Q(见图3－15)
  if(!(Q.front=Q.rear=(QueuePtr)malloc(sizeof(QNode))))
    exit(OVERFLOW);
  Q.front->next=NULL;
}
void DestroyQueue(LinkQueue &Q)
{ // 销毁队列Q(无论空否均可)(见图3－16)
  while(Q.front)
  {
    Q.rear=Q.front->next;
    free(Q.front);
    Q.front=Q.rear;
  }
```

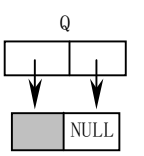


图 3－15　空队列 Q

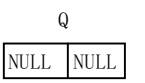
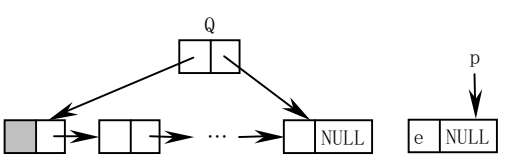
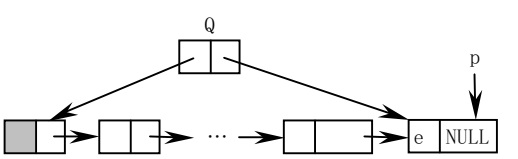


图 3－16　销毁后的队列 Q

```
}
void ClearQueue(LinkQueue &Q)
{ // 将Q清为空队列
  QueuePtr p,q;
  Q.rear=Q.front;
  p=Q.front->next;
  Q.front->next=NULL;
  while(p)
  {
    q=p;
    p=p->next;
    free(q);
  }
}
Status QueueEmpty(LinkQueue Q)
{ // 若Q为空队列，则返回TRUE；否则返回FALSE
  if(Q.front->next==NULL)
    return TRUE;
  else
    return FALSE;
}
int QueueLength(LinkQueue Q)
{ // 求队列的长度
  int i=0;
  QueuePtr p;
  p=Q.front;
  while(Q.rear!=p)
  {
    i++;
    p=p->next;
  }
  return i;
}
Status GetHead(LinkQueue Q,QElemType &e)
{ // 若队列不空，则用e返回Q的队头元素，并返回OK；否则返回ERROR
  QueuePtr p;
  if(Q.front==Q.rear)
    return ERROR;
  p=Q.front->next;
  e=p->data;
  return OK;
}
void EnQueue(LinkQueue &Q,QElemType e)
{ // 插入元素e为Q的新的队尾元素(见图3－17)
  QueuePtr p;
  if(!(p=(QueuePtr)malloc(sizeof(QNode))))
                          // 存储分配失败
    exit(OVERFLOW);
  p->data=e;
  p->next=NULL;
  Q.rear->next=p;
```



(a) p 生成队尾结点



(b) 将队尾结点插入队列

图 3－17　在队列 Q 的尾部插入元素 e

```
  Q.rear=p;
}
Status DeQueue(LinkQueue &Q,QElemType &e)
{ // 若队列不空，删除Q的队头元素，用e返回其值，
  // 并返回OK；否则返回ERROR(见图3－18)
  QueuePtr p;
  if(Q.front==Q.rear)
    return ERROR;
  p=Q.front->next;
  e=p->data;
  Q.front->next=p->next;
  if(Q.rear==p)
    Q.rear=Q.front;
  free(p);
  return OK;
}
void QueueTraverse(LinkQueue Q,void(*vi)(QElemType))
{ // 从队头到队尾依次对队列Q中每个元素调用函数vi()
  QueuePtr p;
  p=Q.front->next;
  while(p)
  {
    vi(p->data);
    p=p->next;
  }
  printf("\n");
}
```

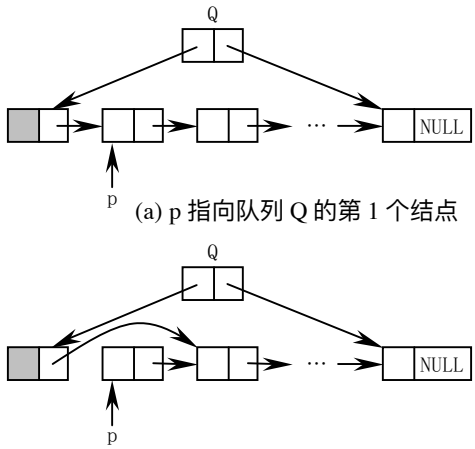


图 3－18　删除队列 Q 的第 1 个元素

```
// main3-2.cpp 检验bo3-2.cpp的主程序
#include"c1.h"
typedef int QElemType;
#include"c3-2.h"
#include"bo3-2.cpp"
void print(QElemType i)
{
  printf("%d ",i);
}
void main()
{
  int i;
  QElemType d;
  LinkQueue q;
  InitQueue(q);
  printf("成功地构造了一个空队列!\n");
  printf("是否空队列? %d(1:空 0:否)  ",QueueEmpty(q));
  printf("队列的长度为%d\n",QueueLength(q));
  EnQueue(q,-5);
  EnQueue(q,5);
  EnQueue(q,10);
  printf("插入3个元素(-5,5,10)后,队列的长度为%d\n",QueueLength(q));
  printf("是否空队列? %d(1:空 0:否)  ",QueueEmpty(q));
```

```
  printf("队列的元素依次为");
  QueueTraverse(q,print);
  i=GetHead(q,d);
  if(i==OK)
    printf("队头元素是: %d\n",d);
  DeQueue(q,d);
  printf("删除了队头元素%d\n",d);
  i=GetHead(q,d);
  if(i==OK)
    printf("新的队头元素是: %d\n",d);
  ClearQueue(q);
  printf("清空队列后,q.front=%u  q.rear=%u  q.front->next=%u\n",q.front,q.rear,q.front->next);
  DestroyQueue(q);
  printf("销毁队列后,q.front=%u  q.rear=%u\n",q.front,  q.rear);
}
```

程序运行结果:

```
成功地构造了一个空队列!
是否空队列? 1(1:空 0:否)  队列的长度为0
插入3个元素(-5,5,10)后,队列的长度为3
是否空队列? 0(1:空 0:否)  队列的元素依次为-5 5 10
队头元素是: -5
删除了队头元素-5
新的队头元素是: 5
清空队列后,q.front=2216 q.rear=2216 q.front->next=0
销毁队列后,q.front=0 q.rear=0
```

由 c3-2.h 和 c2-2.h 对比可见，单链队列和单链表的结构有相同之处。单链队列也是带有头结点的单链表，它的队头指针相当于单链表的头指针。因为队列操作是线性表操作的子集，所以 bo3-2.cpp 中的基本操作也可以用单链表的基本操作来代替。这样既可以充分利用现有资源，减小编程工作量，又可以更清楚地看出队列和线性表的内在联系和共性。bo3-6.cpp 是利用单链表的基本操作实现单链队列基本操作的程序。

```
// bo3-6.cpp 用单链表的基本操作实现链队列(存储结构由c3-2.h定义)的基本操作(9个)
typedef QElemType ElemType;
#define LinkList QueuePtr // 定义单链表的类型与相应的链队列的类型相同
#define LNode QNode
#include"bo2-2.cpp" // 单链表的基本操作
void InitQueue(LinkQueue &Q)
{ // 构造一个空队列Q
  InitList(Q.front); // 调用单链表的基本操作
  Q.rear=Q.front;
}
void DestroyQueue(LinkQueue &Q)
{ // 销毁队列Q(无论空否均可)
  DestroyList(Q.front);
```

```
  Q.rear=Q.front;
}
void ClearQueue(LinkQueue &Q)
{ // 将Q清为空队列
  ClearList(Q.front);
  Q.rear=Q.front;
}
Status QueueEmpty(LinkQueue Q)
{ // 若Q为空队列，则返回TRUE；否则返回FALSE
  return ListEmpty(Q.front);
}
int QueueLength(LinkQueue Q)
{ // 求队列的长度
  return ListLength(Q.front);
}
Status GetHead(LinkQueue Q,QElemType &e)
{ // 若队列不空，则用e返回Q的队头元素，并返回OK；否则返回ERROR
  return GetElem(Q.front,1,e);
}
void EnQueue(LinkQueue &Q,QElemType e)
{ // 插入元素e为Q的新的队尾元素
  QueuePtr p;
  if(!(p=(QueuePtr)malloc(sizeof(QNode)))) // 存储分配失败
    exit(OVERFLOW);
  p->data=e;
  p->next=NULL;
  Q.rear->next=p;
  Q.rear=p;
}
Status DeQueue(LinkQueue &Q,QElemType &e)
{ // 若队列不空，删除Q的队头元素，用e返回其值，并返回OK；否则返回ERROR
  if(Q.front->next==Q.rear) // 队列仅有1个元素(删除的也是队尾元素)
    Q.rear=Q.front; // 令队尾指针指向头结点
  return ListDelete(Q.front,1,e);
}
void QueueTraverse(LinkQueue Q,void(*vi)(QElemType))
{ // 从队头到队尾依次对队列Q中每个元素调用函数vi()
  ListTraverse(Q.front,vi);
}
```

用单链表的操作代替单链队列的操作又和代替链栈的操作情况不同。链栈和单链表的结构完全相同，许多栈的基本操作仅是单链表基本操作改了个名。而单链队列和单链表的结构并不完全相同，只能是在单链队列的基本操作中调用单链表的基本操作。

main3-6.cpp 是检验 bo3-6.cpp 的程序。只有 1 句和 main3-2.cpp 不同。程序运行结果也和 main3-2.cpp 的完全相同。从 main3-6.cpp 中完全看不出 bo3-6.cpp 和 bo3-2.cpp 的区别。

```
// main3-6.cpp 检验bo3-6.cpp的主程序
#include"c1.h"
```

```
typedef int QElemType;
#include"c3-2.h"
#include"bo3-6.cpp" // 仅此句与main3-2.cpp不同
void print(QElemType i)
{
  printf("%d ",i);
}
void main()
{
  int i;
  QElemType d;
  LinkQueue q;
  InitQueue(q);
  printf("成功地构造了一个空队列!\n");
  printf("是否空队列? %d(1:空 0:否)  ",QueueEmpty(q));
  printf("队列的长度为%d\n",QueueLength(q));
  EnQueue(q,-5);
  EnQueue(q,5);
  EnQueue(q,10);
  printf("插入3个元素(-5,5,10)后,队列的长度为%d\n",QueueLength(q));
  printf("是否空队列? %d(1:空 0:否)  ",QueueEmpty(q));
  printf("队列的元素依次为");
  QueueTraverse(q,print);
  i=GetHead(q,d);
  if(i==OK)
    printf("队头元素是: %d\n",d);
  DeQueue(q,d);
  printf("删除了队头元素%d\n",d);
  i=GetHead(q,d);
  if(i==OK)
    printf("新的队头元素是: %d\n",d);
  ClearQueue(q);
  printf("清空队列后,q.front=%u q.rear=%u q.front->next=%u\n",q.front,q.rear,q.front->next);
  DestroyQueue(q);
  printf("销毁队列后,q.front=%u q.rear=%u\n",q.front, q.rear);
}
```

程序运行结果:

```
成功地构造了一个空队列!
是否空队列? 1(1:空 0:否)  队列的长度为0
插入3个元素(-5,5,10)后,队列的长度为3
是否空队列? 0(1:空 0:否)  队列的元素依次为-5 5 10
队头元素是: -5
删除了队头元素-5
新的队头元素是: 5
清空队列后,q.front=2216 q.rear=2216 q.front->next=0
销毁队列后,q.front=0 q.rear=0
```

3.4.3　循环队列——队列的顺序表示和实现

队列的顺序表示为什么要采用循环方式？首先分析两种非循环顺序队列的表示和实现以及它们存在的问题。

```
// c3-5.h 队列的顺序存储结构(非循环队列,队列头元素在[0]单元)
#define QUEUE_INIT_SIZE 10 // 队列存储空间的初始分配量
#define QUEUE_INCREMENT 2 // 队列存储空间的分配增量
struct SqQueue1(见图3－19)
{
  QElemType *base; // 初始化的动态分配存储空间
  int rear; // 尾指针,若队列不空,指向队列尾元素的下一个位置
  int queuesize; // 当前分配的存储容量(以sizeof(QElemType)为单位)
};
```

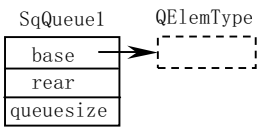


图 3－19　队列的顺序存储结构

图 3－20 是根据 c3-5.h 定义的有两个元素的非循环顺序队列。

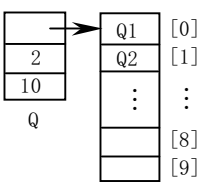


图 3－20　有两个元素(Q1,Q2)的顺序队列 Q

```
// bo3-7.cpp 顺序非循环队列(存储结构由c3-5.h定义)的基本操作(9个)
void InitQueue(SqQueue1 &Q)
{ // 构造一个空队列Q(见图3－21)
  if(!(Q.base=(QElemType*)malloc
            (QUEUE_INIT_SIZE*sizeof(QElemType))))
    exit(ERROR); // 存储分配失败
  Q.rear=0; // 空队列，尾指针为0
  Q.queuesize=QUEUE_INIT_SIZE; // 初始存储容量
}
void DestroyQueue(SqQueue1 &Q)
{ // 销毁队列Q，Q不再存在(见图3－22)
  free(Q.base); // 释放存储空间
  Q.base=NULL;
  Q.rear=Q.queuesize=0;
}
void ClearQueue(SqQueue1 &Q)
{ // 将Q清为空队列
  Q.rear=0;
}
Status QueueEmpty(SqQueue1 Q)
{ // 若队列Q为空队列，则返回TRUE；否则返回FALSE
  if(Q.rear==0)
    return TRUE;
  else
    return FALSE;
}
int QueueLength(SqQueue1 Q)
{ // 返回Q的元素个数，即队列的长度
  return Q.rear;
}
```

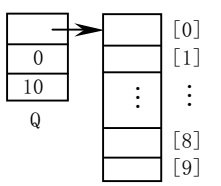


图 3－21　构造一个空的顺序队列 Q

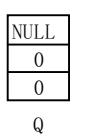
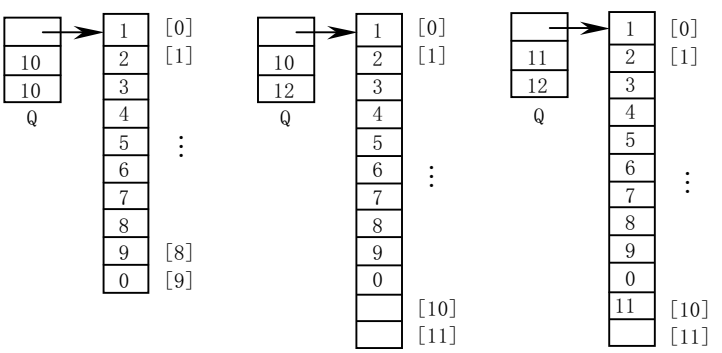


图 3－22　销毁顺序队列 Q

```
Status GetHead(SqQueue1 Q,QElemType &e)
{ // 若队列不空，则用e返回Q的队头元素，并返回OK；否则返回ERROR
  if(Q.rear)
  {
    e=*Q.base;
    return OK;
  }
  else
    return ERROR;
}
void EnQueue(SqQueue1 &Q,QElemType e)
{ // 插入元素e为Q的新的队尾元素(见图3－23)
  if(Q.rear==Q.queuesize) // 当前存储空间已满
  { // 增加分配
    Q.base=(QElemType*)realloc(Q.base,(Q.queuesize+QUEUE_INCREMENT)*sizeof(QElemType));
    if(!Q.base) // 分配失败
      exit(ERROR);
    Q.queuesize+=QUEUE_INCREMENT; // 增加存储容量
  }
  Q.base[Q.rear++]=e; // 入队新元素，队尾指针+1
}
```

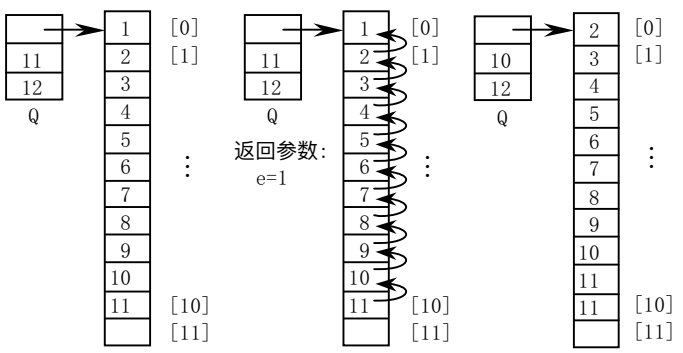


(a) Q 调用函数之前的状态　(b) Q 增加存储容量　(c) Q 调用函数之后的状态

图 3－23　调用 EnQueue()示例(e=11)

```
Status DeQueue(SqQueue1 &Q,QElemType &e)
{ // 若队列不空，则删除Q的队头元素，用e返回其值，并返回OK；否则返回ERROR(见图3－24)
  int i;
  if(Q.rear) // 队列不空
  {
    e=*Q.base;
    for(i=1;i<Q.rear;i++)
      Q.base[i-1]=Q.base[i]; // 依次前移队列元素
    Q.rear--; // 尾指针前移
    return OK;
  }
  else
    return ERROR;
}
```

```
void QueueTraverse(SqQueue1 Q,void(*vi)(QElemType))
{ // 从队头到队尾依次对队列Q中每个元素调用函数vi()
  int i;
  for(i=0;i<Q.rear;i++)
    vi(Q.base[i]);
  printf("\n");
}
```



(a) Q 调用函数之前的状态　(b) 移动元素　(c) Q 调用函数之后的状态

图 3－24　调用 DeQueue()示例

```
// main3-7.cpp 检验bo3-7.cpp的主程序
#include"c1.h"
typedef int QElemType;
#include"c3-5.h"
#include"bo3-7.cpp"
void print(QElemType i)
{
  printf("%d ",i);
}
void main()
{
  Status j;
  int i,k=5;
  QElemType d;
  SqQueue1 Q;
  InitQueue(Q);
  printf("初始化队列后，队列空否? %u(1:空 0:否)\n",QueueEmpty(Q));
  for(i=1;i<=k;i++)
    EnQueue(Q,i); // 依次入队k个元素
  printf("依次入队%d个元素后，队列中的元素为",k);
  QueueTraverse(Q,print);
  printf("队列长度为%d，队列空否? %u(1:空 0:否)\n",QueueLength(Q),QueueEmpty(Q));
  DeQueue(Q,d);
  printf("出队一个元素，其值是%d\n",d);
  j=GetHead(Q,d);
  if(j)
    printf("现在队头元素是%d\n",d);
  ClearQueue(Q);
```

```
  printf("清空队列后，队列空否？%u(1:空 0:否)\n",QueueEmpty(Q));
  DestroyQueue(Q);
}
```

程序运行结果：

```
初始化队列后，队列空否？1(1:空 0:否)
依次入队5个元素后，队列中的元素为1 2 3 4 5
队列长度为5，队列空否？0(1:空 0:否)
出队一个元素，其值是1
现在队头元素是2
清空队列后，队列空否？1(1:空 0:否)
```

c3-5.h 定义的队列顺序存储结构，队头元素总在[0]单元，所以不必设头指针。它的缺点是出队元素时要移动大量元素，尤其在队列较长时，效率很低。这由 DeQueue()函数和图 3－24 可以看出。

c3-4.h 定义了队列的另一种顺序存储结构，它克服了 c3-5.h 定义的存储结构的缺点，在出队元素时不移动元素，只是改变头指针的位置。

```
// c3-4.h 队列的顺序存储结构(出队元素时不移动元素，只改变队头元素的位置)
#define QUEUE_INIT_SIZE 10 // 队列存储空间的初始分配量
#define QUEUE_INCREMENT 2 // 队列存储空间的分配增量
struct SqQueue2(见图3－25)
{
  QElemType *base; // 初始化的动态分配存储空间
  int front; // 头指针，若队列不空,指向队列头元素
  int rear; // 尾指针，若队列不空,指向队列尾元素的下一个位置
  int queuesize; // 当前分配的存储容量(以sizeof(QElemType)为单位)
};
```

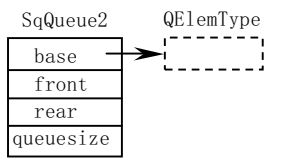


图 3－25　队列的顺序存储结构

图 3－26 是根据 c3-4.h 定义的有 3 个元素的非循环队列。bo3-4.cpp 和 bo3-9.cpp 是这种结构的基本操作。因为后面的程序还要调用 bo3-4.cpp，所以将 9 个基本操作分别放在两个文件中。

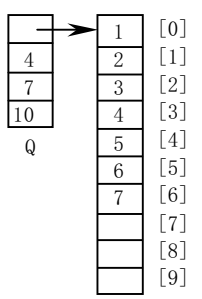


图 3－26　有 3 个元素(5, 6, 7)的非循环队列 Q

```
// bo3-4.cpp 顺序队列(存储结构由c3-4.h定义)的基本操作(5个)
void InitQueue(SqQueue2 &Q)
{ // 构造一个空队列Q(见图3－27)
  if(!(Q.base=(QElemType *)malloc(QUEUE_INIT_SIZE*sizeof(QElemType)))) // 存储分配失败
    exit(ERROR);
  Q.front=Q.rear=0;
  Q.queuesize=QUEUE_INIT_SIZE;
}
void DestroyQueue(SqQueue2 &Q)
{ // 销毁队列Q，Q不再存在(见图3－28)
  if(Q.base)
    free(Q.base);
  Q.base=NULL;
  Q.front=Q.rear=Q.queuesize=0;
}
void ClearQueue(SqQueue2 &Q)
{ // 将Q清为空队列
  Q.front=Q.rear=0;
}
Status QueueEmpty(SqQueue2 Q)
{ // 若队列Q为空队列，则返回TRUE；否则返回FALSE
  if(Q.front==Q.rear) // 队列空的标志
    return TRUE;
  else
    return FALSE;
}
Status GetHead(SqQueue2 Q,QElemType &e)
{ // 若队列不空，则用e返回Q的队头元素，并返回OK；否则返回ERROR
  if(Q.front==Q.rear) // 队列空
    return ERROR;
  e=Q.base[Q.front];
  return OK;
}

// bo3-9.cpp 顺序非循环队列(存储结构由c3-4.h定义)的基本操作(4个)
int QueueLength(SqQueue2 Q)
{ // 返回Q的元素个数，即队列的长度
  return(Q.rear-Q.front);
}
void EnQueue(SqQueue2 &Q,QElemType e)
{ // 插入元素e为Q的新的队尾元素(见图3－29)
  if(Q.rear==Q.queuesize)
  { // 队列满，增加存储单元
    Q.base=(QElemType *)realloc(Q.base,(Q.queuesize+QUEUE_INCREMENT)*sizeof(QElemType));
    if(!Q.base) // 增加单元失败
      exit(ERROR);
  }
  Q.base[Q.rear++]=e;
}
```

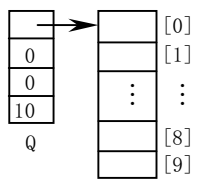


图 3－27　构造一个空的顺序队列 Q

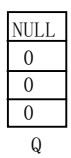
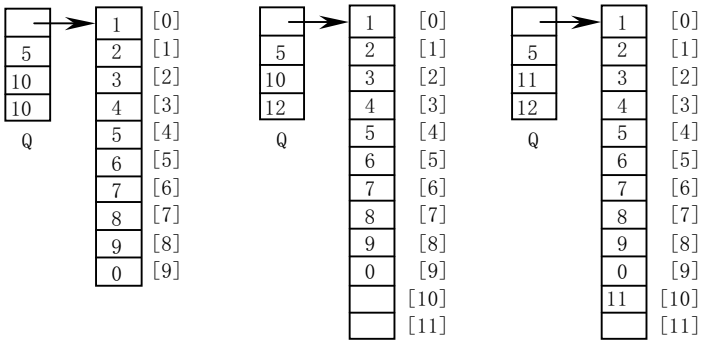


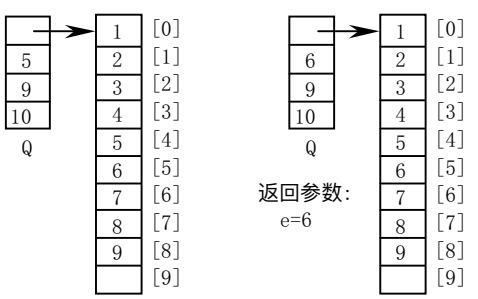
图 3－28　销毁顺序队列 Q



(a) Q 调用函数之前的状态 (b) Q 增加存储容量 (c) Q 调用函数之后的状态

图 3-29 调用 EnQueue()示例(e=11)

```
Status DeQueue(SqQueue2 &Q,QElemType &e)
{ // 若队列不空，则删除Q的队头元素，用e返回其值，并返回OK；否则返回ERROR(见图3-30)
  if(Q.front==Q.rear) // 队列空
    return ERROR;
  e=Q.base[Q.front++];
  return OK;
}
```



(a) Q 调用函数之前的状态 (b) Q 调用函数之后的状态

图 3-30 调用 DeQueue()示例

```
void QueueTraverse(SqQueue2 Q,void(*vi)(QElemType))
{ // 从队头到队尾依次对队列Q中每个元素调用函数vi()
  int i=Q.front;
  while(i!=Q.rear)
    vi(Q.base[i++]);
  printf("\n");
}

// main3-4.cpp 顺序队列(非循环)，检验bo3-4.cpp和bo3-9.cpp的主程序
#include"c1.h"
typedef int QElemType;
#include"c3-4.h"
#include"bo3-4.cpp" // 基本操作(1)
#include"bo3-9.cpp" // 基本操作(2)
void print(QElemType i)
{
```

```
  printf("%d ",i);
}
void main()
{
  Status j;
  int i,n=11;
  QElemType d;
  SqQueue2 Q;
  InitQueue(Q);
  printf("初始化队列后，队列空否? %u(1:空 0:否)\n",QueueEmpty(Q));
  printf("队列长度为%d\n",QueueLength(Q));
  printf("请输入%d个整型队列元素:\n",n);
  for(i=0;i<n;i++)
  {
    scanf("%d",&d);
    EnQueue(Q,d);
  }
  printf("队列长度为%d\n",QueueLength(Q));
  printf("现在队列空否? %u(1:空 0:否)\n",QueueEmpty(Q));
  printf("现在队列中的元素为\n");
  QueueTraverse(Q,print);
  DeQueue(Q,d);
  printf("删除队头元素%d\n",d);
  printf("队列中的元素为\n");
  QueueTraverse(Q,print);
  j=GetHead(Q,d);
  if(j)
    printf("队头元素为%d\n",d);
  else
    printf("无队头元素(空队列)\n");
  ClearQueue(Q);
  printf("清空队列后，队列空否? %u(1:空 0:否)\n",QueueEmpty(Q));
  j=GetHead(Q,d);
  if(j)
    printf("队头元素为 %d\n",d);
  else
    printf("无队头元素(空队列)\n");
  DestroyQueue(Q);
}
```

程序运行结果:

```
初始化队列后，队列空否? 1(1:空 0:否)
队列长度为0
请输入11个整型队列元素:
0 1 2 3 4 5 6 7 8 9 10↙
队列长度为11
现在队列空否? 0(1:空 0:否)
现在队列中的元素为
```

```
0 1 2 3 4 5 6 7 8 9 10
删除队头元素0
队列中的元素为
1 2 3 4 5 6 7 8 9 10
队头元素为1
清空队列后，队列空否？1(1:空 0:否)
无队头元素(空队列)
```

c3-4.h 定义的队列顺序存储结构，在出队元素时，只是改变头指针的位置，不移动元素，可简化操作，节约时间，这从 DeQueue()函数和图 3－30 可看出。但这种队列顺序存储结构也有它的缺点，队列的每个存储空间自始至终只能存一个队列元素。即使这个队列元素出队后，其它的队列元素也不能占用这个存储空间。尤其在队列长度不长，入队出队频繁的情况下，存储空间浪费较大。由于没有其它数据覆盖，当队头元素出队后，其值还保留在队列中。后面的 algo3-11.cpp(另一种求迷宫方法)就利用了 c3-4.h 的这个特点。

c3-3.h 克服了 c3-4.h 存储空间浪费较大的缺点。c3-3.h 采用了循环队列：当队尾元素占据了存储空间的最后一个单元时，如再有新的元素入队，不是申请新的存储空间，而是将新元素插到存储空间的第一个单元，只要这个单元为空(元素已出队)。通过头尾指针对存储空间 MAX_QSIZE 求余做到这一点，形成循环队列。

在循环队列中，队尾指针可能小于队头指针。入队元素时，队尾指针加 1。当队列满时，队尾指针等于队头指针，和队列空的条件一样。为了区别队满和队空，在循环队列中，少用一个存储单元。也就是在存储空间为 MAX_QSIZE 的循环队列中，最多只能存放 MAX_QSIZE-1 个元素。这样，队列空的条件仍为队尾指针等于队头指针，队列满的条件改为(队尾指针+1)对 MAX_QSIZE 求余等于队头指针。

```
// c3-3.h 队列的顺序存储结构(循环队列)(见图3－31)
#define MAX_QSIZE 5 // 最大队列长度+1
struct SqQueue
{
  QElemType *base; // 初始化的动态分配存储空间
  int front; // 头指针，若队列不空，指向队列头元素
  int rear; // 尾指针，若队列不空，指向队列尾元素的下一个位置
};

// bo3-3.cpp 循环队列(存储结构由c3-3.h定义)的基本操作(9个)
void InitQueue(SqQueue &Q)
{ // 构造一个空队列Q(见图3－32)
  Q.base=(QElemType *)malloc(MAX_QSIZE*sizeof(QElemType));
  if(!Q.base) // 存储分配失败
    exit(OVERFLOW);
  Q.front=Q.rear=0;
}
void DestroyQueue(SqQueue &Q)
{ // 销毁队列Q，Q不再存在(见图3－33)
  if(Q.base)
    free(Q.base);
```

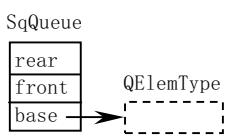


图 3－31　循环队列的存储结构

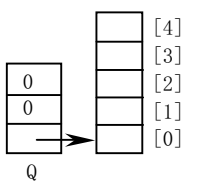


图 3－32　空队列 Q

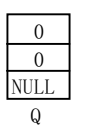
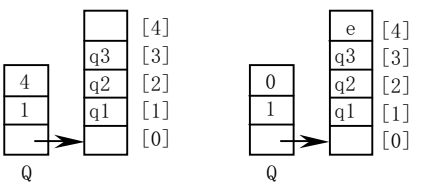


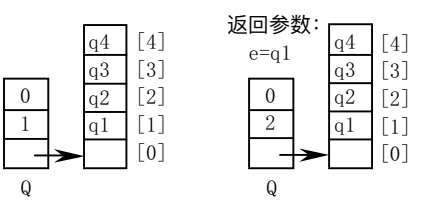
图 3－33　销毁队列 Q

```
  Q.base=NULL;
  Q.front=Q.rear=0;
}
void ClearQueue(SqQueue &Q)
{ // 将Q清为空队列(见图3-32)
  Q.front=Q.rear=0;
}
Status QueueEmpty(SqQueue Q)
{ // 若队列Q为空队列，则返回TRUE；否则返回FALSE
  if(Q.front==Q.rear) // 队列空的标志
    return TRUE;
  else
    return FALSE;
}
int QueueLength(SqQueue Q)
{ // 返回Q的元素个数，即队列的长度
  return(Q.rear-Q.front+MAX_QSIZE)%MAX_QSIZE;
}
Status GetHead(SqQueue Q,QElemType &e)
{ // 若队列不空，则用e返回Q的队头元素，并返回OK；否则返回ERROR
  if(Q.front==Q.rear) // 队列空
    return ERROR;
  e=Q.base[Q.front];
  return OK;
}
Status EnQueue(SqQueue &Q,QElemType e)
{ // 插入元素e为Q的新的队尾元素(见图3-34)
  if((Q.rear+1)%MAX_QSIZE==Q.front) // 队列满
    return ERROR;
  Q.base[Q.rear]=e;
  Q.rear=(Q.rear+1)%MAX_QSIZE;
  return OK;
}
Status DeQueue(SqQueue &Q,QElemType &e)
{ // 若队列不空，则删除Q的队头元素，用e返回其值，并返回OK；否则返回ERROR(见图3-35)
  if(Q.front==Q.rear) // 队列空
    return ERROR;
  e=Q.base[Q.front];
  Q.front=(Q.front+1)%MAX_QSIZE;
  return OK;
}
void QueueTraverse(SqQueue Q,void(*vi)(QElemType))
{ // 从队头到队尾依次对队列Q中每个元素调用函数vi()
  int i;
  i=Q.front;
  while(i!=Q.rear)
  {
    vi(Q.base[i]);
    i=(i+1)%MAX_QSIZE;
  }
  printf("\n");
```



(a) 调用 EnQueue()前　(b) 调用 EnQueue()后

图 3-34　调用 EnQueue()示例



(a) 调用 DeQueue()前　(b) 调用 DeQueue()后

图 3-35　调用 DeQueue()示例

```
}

// main3-3.cpp 循环队列 检验bo3-3.cpp的主程序
#include"c1.h"
typedef int QElemType;
#include"c3-3.h"
#include"bo3-3.cpp"
void print(QElemType i)
{
  printf("%d ",i);
}
void main()
{
  Status j;
  int i=0,l;
  QElemType d;
  SqQueue Q;
  InitQueue(Q);
  printf("初始化队列后, 队列空否? %u(1:空 0:否)\n",QueueEmpty(Q));
  printf("请输入整型队列元素(不超过%d个),-1为提前结束符: ",MAX_QSIZE-1);
  do
  {
    scanf("%d",&d);
    if(d==-1)
      break;
    i++;
    EnQueue(Q,d);
  }while(i<MAX_QSIZE-1);
  printf("队列长度为%d\n",QueueLength(Q));
  printf("现在队列空否? %u(1:空 0:否)\n",QueueEmpty(Q));
  printf("连续%d次由队头删除元素,队尾插入元素:\n",MAX_QSIZE);
  for(l=1;l<=MAX_QSIZE;l++)
  {
    DeQueue(Q,d);
    printf("删除的元素是%d,请输入待插入的元素: ",d);
    scanf("%d",&d);
    EnQueue(Q,d);
  }
  l=QueueLength(Q);
  printf("现在队列中的元素为\n");
  QueueTraverse(Q,print);
  printf("共向队尾插入了%d个元素\n",i+MAX_QSIZE);
  if(l-2>0)
    printf("现在由队头删除%d个元素: \n",l-2);
  while(QueueLength(Q)>2)
  {
    DeQueue(Q,d);
    printf("删除的元素值为%d\n",d);
  }
  j=GetHead(Q,d);
  if(j)
```

```
    printf("现在队头元素为%d\n",d);
  ClearQueue(Q);
  printf("清空队列后, 队列空否? %u(1:空 0:否)\n",QueueEmpty(Q));
  DestroyQueue(Q);
}
```

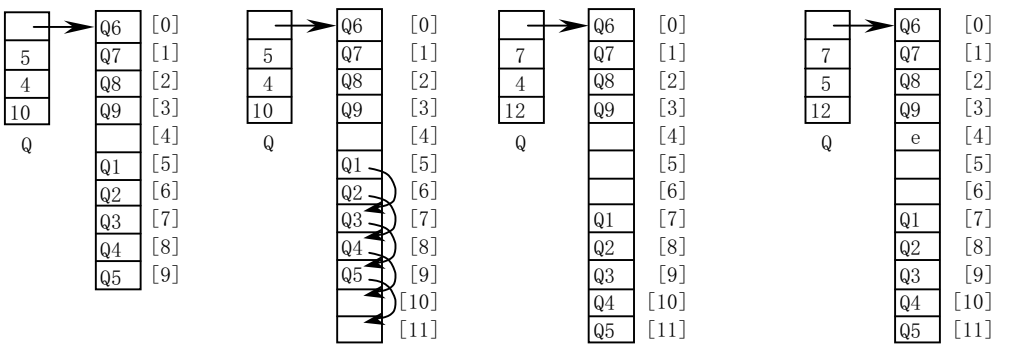
程序运行结果:

```
初始化队列后, 队列空否? 1(1:空 0:否)
请输入整型队列元素(不超过4个),-1为提前结束符: 1 2 3 -1↙
队列长度为3
现在队列空否? 0(1:空 0:否)
连续5次由队头删除元素,队尾插入元素:
删除的元素是1,请输入待插入的元素: 4↙
删除的元素是2,请输入待插入的元素: 5↙
删除的元素是3,请输入待插入的元素: 6↙
删除的元素是4,请输入待插入的元素: 7↙
删除的元素是5,请输入待插入的元素: 8↙
现在队列中的元素为
6 7 8
共向队尾插入了8个元素
现在由队头删除1个元素:
删除的元素值为6
现在队头元素为7
清空队列后, 队列空否? 1(1:空 0:否)
```

c3-3.h 所采用的循环顺序队列存储结构，当队列长度大于 MAX_QSIZE-1 时，无法动态地增加存储空间，原因是 MAX_QSIZE 是固定于 c3-3.h 中的常量。为了使循环队列也能动态地增加存储空间，不固定队列长度，把队列长度也作为结构体的一个成员。前述 c3-4.h 就可以作为这种循环顺序队列的存储结构，bo3-8.cpp 是基于 c3-4.h 结构的循环顺序队列的基本操作。

```
// bo3-8.cpp 循环队列(存储结构由c3-4.h定义)的基本操作(4个)
int QueueLength(SqQueue2 Q)
{ // 返回Q的元素个数, 即队列的长度
  return(Q.rear-Q.front+Q.queuesize)%Q.queuesize;
}
void EnQueue(SqQueue2 &Q,QElemType e)
{ // 插入元素e为Q的新的队尾元素(见图3－36)
  int i;
  if((Q.rear+1)%Q.queuesize==Q.front)
  { // 队列满, 增加存储单元
    Q.base=(QElemType *)realloc(Q.base,(Q.queuesize+QUEUE_INCREMENT)*sizeof(QElemType));
    if(!Q.base) // 增加单元失败
      exit(ERROR);
    if(Q.front>Q.rear) // 形成循环
    {
```

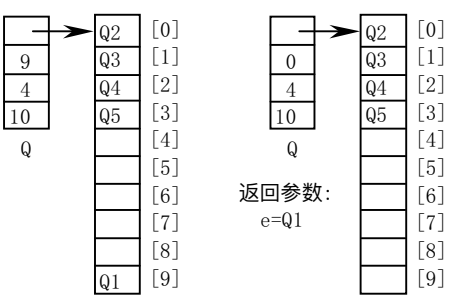
```
      for(i=Q.queuesize-1;i>=Q.front;i--)
        Q.base[i+QUEUE_INCREMENT]=Q.base[i]; // 移动高端元素到新的高端
      Q.front+=QUEUE_INCREMENT; // 移动队头指针
    }
    Q.queuesize+=QUEUE_INCREMENT; // 增加队列长度
  }
  Q.base[Q.rear]=e; // 将e插入队尾
  Q.rear=++Q.rear%Q.queuesize; // 移动队尾指针
}
```



(a) Q 调用函数之前的状态 (b) Q 增加存储容量 (c) 移动高端元素到新的高端 (d) Q 调用函数之后的状态

图 3 - 36　调用 EnQueue()示例

```
Status DeQueue(SqQueue2 &Q,QElemType &e)
{ // 若队列不空，则删除Q的队头元素，用e返回其值，并返回OK；否则返回ERROR(见图3 - 37)
  if(Q.front==Q.rear) // 队列空
    return ERROR;
  e=Q.base[Q.front]; // 用e返回队头元素
  Q.front=++Q.front%Q.queuesize; // 移动队头指针
  return OK;
}
```



(a) Q 调用函数之前的状态 (b) Q 调用函数之后的状态

图 3 - 37　调用 DeQueue()示例

```
void QueueTraverse(SqQueue2 Q,void(*vi)(QElemType))
{ // 从队头到队尾依次对队列Q中每个元素调用函数vi()
  int i=Q.front; // i指向队头
  while(i!=Q.rear) // 没到队尾
  {
    vi(Q.base[i]); // 调用函数vi()
    i=++i%Q.queuesize; // 向后移动i指针
  }
```

```
  printf("\n");
}

// main3-8.cpp 循环且可增加存储空间的顺序队列，检验bo3-8.cpp的主程序
#include"c1.h"
typedef int QElemType;
#include"c3-4.h"
#include"bo3-4.cpp" // 基本操作(1)，与非循环同
#include"bo3-8.cpp" // 基本操作(2)，与非循环不同
void print(QElemType i)
{
  printf("%d ",i);
}
void main()
{
  Status j;
  int i,n=11;
  QElemType d;
  SqQueue2 Q;
  InitQueue(Q);
  printf("初始化队列后，队列空否? %u(1:空 0:否)\n",QueueEmpty(Q));
  printf("队列长度为%d\n",QueueLength(Q));
  printf("请输入%d个整型队列元素:\n",n);
  for(i=0;i<n;i++)
  {
    scanf("%d",&d);
    EnQueue(Q,d);
  }
  printf("队列长度为%d\n",QueueLength(Q));
  printf("现在队列空否? %u(1:空 0:否)\n",QueueEmpty(Q));
  printf("现在队列中的元素为 \n");
  QueueTraverse(Q,print);
  for(i=1;i<=3;i++)
    DeQueue(Q,d);
  printf("由队头删除3个元素，最后删除的元素为%d\n",d);
  printf("现在队列中的元素为 \n");
  QueueTraverse(Q,print);
  j=GetHead(Q,d);
  if(j)
    printf("队头元素为%d\n",d);
  else
    printf("无队头元素(空队列)\n");
  for(i=1;i<=5;i++)
    EnQueue(Q,i);
  printf("依次向队尾插入1～5，现在队列中的元素为\n");
  QueueTraverse(Q,print);
  ClearQueue(Q);
  printf("清空队列后，队列空否? %u(1:空 0:否)\n",QueueEmpty(Q));
  j=GetHead(Q,d);
  if(j)
    printf("队头元素为 %d\n",d);
```

```
  else
    printf("无队头元素(空队列)\n");
  DestroyQueue(Q);
}
```

程序运行结果：

```
初始化队列后，队列空否? 1(1:空 0:否)
队列长度为0
请输入11个整型队列元素:
1 2 3 4 5 6 7 8 9 10 11↙
队列长度为11
现在队列空否? 0(1:空 0:否)
现在队列中的元素为
1 2 3 4 5 6 7 8 9 10 11
由队头删除3个元素，最后删除的元素为3
现在队列中的元素为
4 5 6 7 8 9 10 11
队头元素为 4
依次向队尾插入1～5，现在队列中的元素为
4 5 6 7 8 9 10 11 1 2 3 4 5
清空队列后，队列空否? 1(1:空 0:否)
无队头元素(空队列)
```

```
// algo3-11.cpp 利用非循环顺序队列采用广度搜索法求解迷宫问题(一条路径)
#include"c1.h"
#include"func3-1.cpp"
#define D 8 // 移动方向数，只能取4和8。(8个，可斜行；4个，只可直走)
typedef struct // 定义队列元素和栈元素为同类型的结构体(见图3－38)
{
  PosType seat; // 当前点的行值，列值
  int pre; // 前一点在队列中的序号
}QElemType,SElemType; // 栈元素和队列元素
#include"c3-1.h" // 栈的存储结构
#include"bo3-1.cpp" // 栈的基本操作
#include"c3-4.h" // 队列的存储结构
#include"bo3-4.cpp" // 非循环顺序队列的基本操作(1)
#include"bo3-9.cpp" // 非循环顺序队列的基本操作(2)
struct // 移动数组，移动方向由正东起顺时针转
{
  int x,y;
}move[D]={
#if D==8
   {0,1},{1,1},{1,0},{1,-1},{0,-1},{-1,-1},{-1,0},{-1,1}};
#endif
#if D==4
   {0,1},{1,0},{0,-1},{-1,0}};
#endif
void Path()
```

SElemType　QElemType

seat.x	seat.y	pre

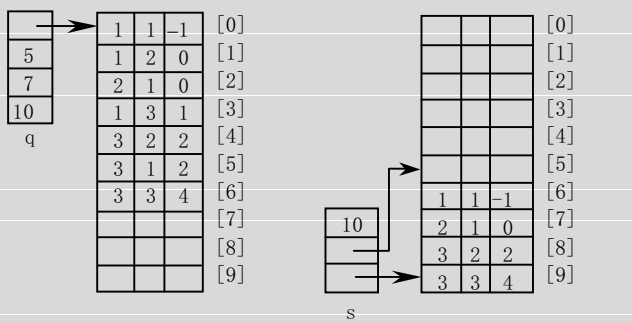
图 3－38　SElemType 和 QElemType 结构

```
{ // 广度搜索法求一条迷宫路径
  SqQueue2 q; // 采用非循环顺序队列
  QElemType qf,qt; // 当前点和下一点
  SqStack s; // 采用顺序栈
  int i,flag=1; // 当找到出口, flag=0
  qf.seat.x=begin.x; // 将入口作为当前点
  qf.seat.y=begin.y;
  qf.pre=-1; // 设入口(第一点)的上一点的序号=-1
  m[qf.seat.x][qf.seat.y]=-1; // 初始点设为-1(标记已访问过)
  InitQueue(q);
  EnQueue(q,qf); // 起点入队
  while(!QueueEmpty(q)&&flag)
  { // 队列中还有没被广度搜索过的点且还没找到出口
    DeQueue(q,qf); // 出队qf为当前点
    for(i=0;i<D;i++) // 向各个方向尝试
    {
      qt.seat.x=qf.seat.x+move[i].x; // 下一点的坐标
      qt.seat.y=qf.seat.y+move[i].y;
      if(m[qt.seat.x][qt.seat.y]==1)
      { // 此点是通道且不曾被访问过
        m[qt.seat.x][qt.seat.y]=-1; // 标记已访问过
        qt.pre=q.front-1; // qt的前一点处于队列中现队头减1的位置(没删除)
        EnQueue(q,qt); // 入队qt
        if(qt.seat.x==end.x&&qt.seat.y==end.y) // 到达终点
        {
          flag=0;
          break;
        }
      }
    }
  }
  if(flag) // 搜索完整个队列还没到达终点
    printf("没有路径可到达终点! \n");
  else // 到达终点
  {
    InitStack(s); // 初始化s栈
    i=q.rear-1; // i为待入栈元素在队列中的位置
    while(i>=0) // 没到入口
    {
      Push(s,q.base[i]); // 将队列中的路径入栈(栈底为出口, 栈顶为入口)
      i=q.base[i].pre; // i为前一元素在队列中的位置
    }
    i=0; // i为走出迷宫的足迹
    while(!StackEmpty(s))
    {
      Pop(s,qf); // 依照由入口到出口的顺序弹出路径
      i++;
      m[qf.seat.x][qf.seat.y]=i; // 标记路径为足迹(标记前的值为-1)
    }
    printf("走出迷宫的一个方案: \n");
    Print(); // 输出m数组
```

```
  }
}
void main()
{
  Init(1); // 初始化迷宫，通道值为1
  Path(); // 求一条迷宫路径
}
```

程序运行结果(见图 3－39)：

```
请输入迷宫的行数,列数(包括外墙): 5,5↙
请输入迷宫内墙单元数: 2↙
请依次输入迷宫内墙每个单元的行数,列数:
2,2↙
2,3↙
迷宫结构如下:
  0  0  0  0  0
  0  1  1  1  0
  0  1  0  0  0
  0  1  1  1  0
  0  0  0  0  0
请输入入口的行数,列数: 1,1↙
请输入出口的行数,列数: 3,3↙
走出迷宫的一个方案:
  0  0  0  0  0
  0  1 -1 -1  0
  0  2  0  0  0
  0 -1  3  4  0
  0  0  0  0  0
```



(a) 到达终点时队列 q 的状态　(b) 栈 s 在元素最多时的状态

图 3－39　程序运行期间栈和队列的状态

algo3-11.cpp 也是一种求迷宫的算法。该算法先将入口坐标及-1 入队，-1 表示入口坐标是第 1 点(无前驱)。出队入口点(1, 1, -1)，找 m[1][1]周边值为 1 的点(m[1][2]和 m[2][1])，入队这 2 点，且令它们的 pre 成员值为 0，因为它们的前一步是 m[1][1]，而 m[1][1]在队列中的序号为[0]；再出队(1, 2, 0)，找 m[1][2]周边值为 1 的点(m[1][3])，入队该点，且令它的 pre 成员值为 1；……；直到入队(3, 3, 4)。而 m[3][3]恰是出口，说明找到了一条由入口到出口的路径。为了输出这条路径，将(3, 3, 4)入栈 s，由(3, 3, 4)得知，出口的前一步在队列中序号为[4]处。再入栈队列中序号为[4]的元素(3, 2, 2)。依此类推，再入栈队列中序号为[2]的元素(2, 1, 0)、序号为[0]的元素(1, 1, -1)。出栈 s 时，依次将 m[1][1]、m[2][1]、m[3][2]和 m[3][3]赋值 1、2、3 和 4(足迹)。

在到达终点时，队列的状态如图 3－39(a)所示。这时队列中只有 2 个元素，队头元素的序号为[5]，队尾元素的序号为[6]。序号从[0]到[4]的那些元素虽然出队，但它们在队列中的存储状态并没有遭到破坏，仍完好地保存。所以才可以调出队列的“历史记录”。这也算是这种存储结构的一个优点吧。

在 algo3-11.cpp 的第 4 行，我们定义 D 为 8，它可以斜行(从足迹 2 到足迹 3)。如果定义 D 为 4，输入相同，则需 4 步走到出口。运行结果如下(省略相同部分)：

```
走出迷宫的一个方案:
  0  0  0  0  0
  0  1 -1 -1  0
  0  2  0  0  0
  0  3  4  5  0
  0  0  0  0  0
```

由于队列要在表的一端插入元素，在表的另一端删除元素，故采用链式存储结构比较好。可以从根本上免除移动队列元素的操作，且节约存储空间。

3.5　离散事件模拟

```
// func3-3.cpp、algo3-12.cpp和algo3-13.cpp用到的函数及变量等
#include"c1.h"
typedef struct // 定义ElemType为结构体类型
{
  int OccurTime; // 事件发生时刻
  int NType; // 事件类型，Qu表示到达事件，0至Qu-1表示Qu个窗口的离开事件
}Event,ElemType; // 事件类型，有序链表LinkList的数据元素类型(见图3-40)
#include"c2-5.h" // 从实际应用角度出发重新定义的线性链表结构
typedef LinkList EventList; // 事件链表指针类型，定义为有序链表
#include"bo2-6.cpp" // 基于c2-5.h存储结构的基本操作
typedef struct
{
  int ArrivalTime; // 到达时刻
  int Duration; // 办理事务所需时间
}QElemType; // 定义队列的数据元素类型QElemType为结构体类型(见图3-41)
#include"c3-2.h" // 链队列存储结构
#include"bo3-2.cpp" // 链队列基本操作
// 程序中用到的主要变量(全局)
EventList ev; // 事件表头指针
Event en,et; // 事件,临时变量
//FILE *fp; // 文件型指针，用于指向b.txt或d.txt文件
long int TotalTime=0; // 累计客户逗留时间(初值为0)
int CloseTime,CustomerNum=0; // 银行营业时间(单位是分),客户数(初值为0)
int cmp(Event a,Event b)
{ // 依事件a的发生时刻<、=或>事件b的发生时刻分别返回-1、0或1
  if(a.OccurTime==b.OccurTime)
    return 0;
  else
    return (a.OccurTime-b.OccurTime)/abs(a.OccurTime-b.OccurTime);
}
void Random(int &d,int &i)
{ // 生成两个随机数
  d=rand()%Blsj+1; // 1到Blsj之间的随机数(办理业务的时间)
```

Event ElemType

OccurTime	NType

图 3-40　Event 和 ElemType 结构

QElemType

ArrivalTime	Duration

图 3-41　QElemType 结构

```
  i=rand()%(Khjg+1); // 0到Khjg之间的随机数(客户到达的时间间隔)
}
void OpenForDay();
void CustomerArrived();
void CustomerDeparture();
void Bank_Simulation()
{ // 银行业务模拟函数
  Link p;
  OpenForDay(); // 初始化事件表ev且插入第1个到达事件，初始化队列
  while(!ListEmpty(ev)) // 事件表ev不空
  {
    DelFirst(ev,ev.head,p); // 删除事件表ev的第1个结点，并由p返回其指针，在bo2-6.cpp中
//  if(p->data.OccurTime<50) // 输出前50分钟内发生的事件到文件d.txt中
//    fprintf(fp,"p->data.OccurTime=%3d  p->data.NType=%d\n",p->data.OccurTime,p->data.NType);
    en.OccurTime=GetCurElem(p).OccurTime;
    // GetCurElem()在bo2-6.cpp中，返回p->data(ElemType类型)
    en.NType=GetCurElem(p).NType;
    if(en.NType==Qu) // 到达事件
      CustomerArrived(); // 处理客户到达事件
    else // 由某窗口离开的事件
      CustomerDeparture(); // 处理客户离开事件
  } // 计算并输出平均逗留时间
  printf("窗口数=%d 两相邻到达的客户的时间间隔=0～%d分钟 每个客户办理业务的时间=
  1～%d分钟\n",Qu,Khjg,Blsj);
  printf("客户总数:%d, 所有客户共耗时:%ld分钟,平均每人耗时:%d分钟,",CustomerNum,TotalTime,
  TotalTime/CustomerNum);
  printf("最后一个客户离开的时间:%d分\n",en.OccurTime);
}

// algo3-12.cpp 银行业务模拟。实现算法3.6、3.7的程序
#define Qu 4 // 客户队列数
#define Khjg 5 // 两相邻到达的客户的时间间隔最大值
#define Blsj 30 // 每个客户办理业务的时间最大值
#include"func3-3.cpp" // 包含algo3-12.cpp和algo3-13.cpp共同用到的函数和变量等
LinkQueue q[Qu]; // Qu个客户队列
QElemType customer; // 客户记录，临时变量
//FILE *fq; // 文件型指针，用于指向a.txt文件
void OpenForDay()
{ // 初始化事件链表ev且插入第1个到达事件，初始化Qu个队列
  int i;
  InitList(ev); // 初始化事件链表ev为空(见图3-42)
  en.OccurTime=0; // 设定第1位客户到达时间为0
                  // (银行一开门，就有客户来)
//fprintf(fq,"首位客户到达时刻=%3d,",en.OccurTime);
  en.NType=Qu; // 到达
  OrderInsert(ev,en,cmp); // 将第1个到达事件en有序插入事件表ev中，在bo2-6.cpp中(见图3-43)
  for(i=0;i<Qu;++i) // 初始化Qu个队列(见图3-44)
    InitQueue(q[i]);
}
```

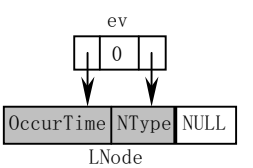


图3-42 初始化事件链表ev

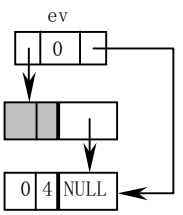


图 3－43　在 ev 中插入第 1 个事件

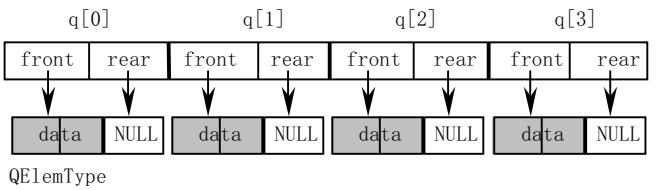


图 3－44　初始化队列数组 q

```
int Minimum(LinkQueue Q[])
{ // 返回最短队列的序号，若有并列值，返回队列序号最小的
  int l[Qu];
  int i,k=0;
  for(i=0;i<Qu;i++)
    l[i]=QueueLength(Q[i]);
  for(i=1;i<Qu;i++)
    if(l[i]<l[0])
    {
      l[0]=l[i];
      k=i;
    }
  return k;
}
void CustomerArrived()
{ // 处理客户到达事件en(en.NType=Qu)
  QElemType f;
  int durtime,intertime,i;
  ++CustomerNum; // 客户数加1
  Random(durtime,intertime);//生成当前客户办理业务的时间和下一个客户到达的时间间隔2个随机数
  et.OccurTime=en.OccurTime+intertime; // 下一客户et到达时刻等于当前客户en的到达
                                       // 时间加时间间隔
  et.NType=Qu; // 下一客户到达事件
  i=Minimum(q); // 求长度最短队列的序号，等长为最小的序号(到达事件将入该队)
//if(CustomerNum<=20) // 输出前20个客户到达信息到文件a.txt中
//  fprintf(fq,"办理业务的时间=%2d,所排队列=%d\n下一客户到达时刻=%3d,",durtime,i,
    et.OccurTime);
  if(et.OccurTime<CloseTime) // 下一客户到达时银行尚未关门
    OrderInsert(ev,et,cmp); // 按升序将下一客户到达事件et插入事件表ev中，在bo2-6.cpp中
  f.ArrivalTime=en.OccurTime; // 将当前客户到达事件en赋给队列元素f
  f.Duration=durtime;
  EnQueue(q[i],f); // 将f入队到第i队列(i=0～Qu-1)
  if(QueueLength(q[i])==1) // 该元素为队头元素
  {
    et.OccurTime=en.OccurTime+durtime; // 设定一个离开事件et
    et.NType=i;
    OrderInsert(ev,et,cmp); // 将此离开事件et按升序插入事件表ev中
  }
}
void CustomerDeparture()
{ // 处理客户离开事件en(en.NType<Qu)
  int i;
```

```
  i=en.NType; // 确定离开事件en发生的队列序号i
  DeQueue(q[i],customer); // 删除第i队列的排头客户
  TotalTime+=en.OccurTime-customer.ArrivalTime;
  // 客户逗留时间=离开事件en的发生时刻-该客户的到达时间
  if(!QueueEmpty(q[i]))
  { // 删除第i队列的排头客户后，第i队列仍不空
    GetHead(q[i],customer); // 将第i队列新的排头客户赋给customer
    et.OccurTime=en.OccurTime+customer.Duration;
    // 设定离开事件et,新排头的离开时间等于原排头的离开时间加新排头办理业务的时间
    et.NType=i; // 第i个队列的离开事件
    OrderInsert(ev,et,cmp); // 将此离开事件et按升序插入事件表ev中
  }
}
void main()
{
//fq=fopen("a.txt","w"); // 打开a.txt文件，用于写入客户到达信息
//fp=fopen("b.txt","w"); // 打开b.txt文件，用于写入有序事件表的历史记录
  printf("请输入银行营业时间长度(单位:分): ");
  scanf("%d",&CloseTime);
//srand(time(0));
  // 设置随机数种子，以使每次运行程序产生的随机数不同(time(0)是长整型数，与调用时间有关)
  Bank_Simulation();
//fclose(fq); // 关闭a.txt
//fclose(fp); // 关闭b.txt
}
```

程序运行结果:

```
请输入银行营业时间长度(单位:分): 480↙
窗口数=4 两相邻到达的客户的时间间隔=0～5分钟 每个客户办理业务的时间=1～30分钟
客户总数:209, 所有客户共耗时:42224分钟,平均每人耗时:202分钟,最后一个客户离开的时间:872分
```

文件 a. txt 的内容(见图 3 - 45):

```
首位客户到达时刻=  0,办理业务的时间=17,所排队列=0
下一客户到达时刻=  4,办理业务的时间= 3,所排队列=1
下一客户到达时刻=  8,办理业务的时间=17,所排队列=1
下一客户到达时刻=  9,办理业务的时间=16,所排队列=2
下一客户到达时刻= 10,办理业务的时间=29,所排队列=3
下一客户到达时刻= 14,办理业务的时间= 5,所排队列=0
下一客户到达时刻= 16,办理业务的时间= 2,所排队列=1
下一客户到达时刻= 17,办理业务的时间=13,所排队列=2
下一客户到达时刻= 21,办理业务的时间=13,所排队列=0
下一客户到达时刻= 24,办理业务的时间=24,所排队列=0
下一客户到达时刻= 27,办理业务的时间=30,所排队列=1
下一客户到达时刻= 28,办理业务的时间=21,所排队列=1
下一客户到达时刻= 33,办理业务的时间=15,所排队列=2
下一客户到达时刻= 38,办理业务的时间=13,所排队列=0
```

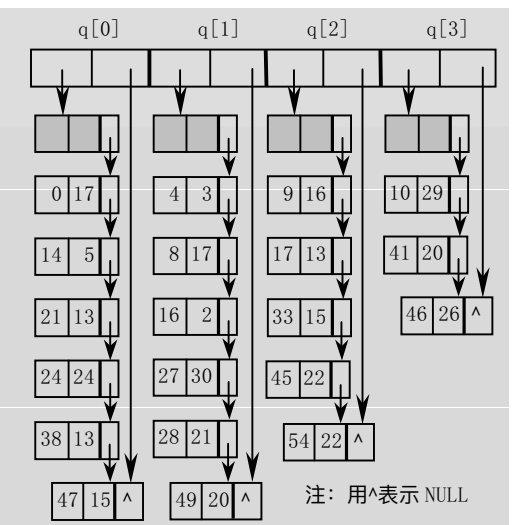


图 3 - 45　4 个队列的历史记录

```
下一客户到达时刻= 41,办理业务的时间=20,所排队列=3
下一客户到达时刻= 45,办理业务的时间=22,所排队列=2
下一客户到达时刻= 46,办理业务的时间=26,所排队列=3
下一客户到达时刻= 47,办理业务的时间=15,所排队列=0
下一客户到达时刻= 49,办理业务的时间=20,所排队列=1
下一客户到达时刻= 54,办理业务的时间=22,所排队列=2
下一客户到达时刻= 54,
```

文件 b. txt 的内容(见图 3－46)：

```
p->data.OccurTime=  0 p->data.NType=4
p->data.OccurTime=  4 p->data.NType=4
p->data.OccurTime=  7 p->data.NType=1
p->data.OccurTime=  8 p->data.NType=4
p->data.OccurTime=  9 p->data.NType=4
p->data.OccurTime= 10 p->data.NType=4
p->data.OccurTime= 14 p->data.NType=4
p->data.OccurTime= 16 p->data.NType=4
p->data.OccurTime= 17 p->data.NType=4
p->data.OccurTime= 17 p->data.NType=0
p->data.OccurTime= 21 p->data.NType=4
p->data.OccurTime= 22 p->data.NType=0
p->data.OccurTime= 24 p->data.NType=4
p->data.OccurTime= 25 p->data.NType=2
p->data.OccurTime= 25 p->data.NType=1
p->data.OccurTime= 27 p->data.NType=1
p->data.OccurTime= 27 p->data.NType=4
p->data.OccurTime= 28 p->data.NType=4
p->data.OccurTime= 33 p->data.NType=4
p->data.OccurTime= 35 p->data.NType=0
p->data.OccurTime= 38 p->data.NType=4
p->data.OccurTime= 38 p->data.NType=2
p->data.OccurTime= 39 p->data.NType=3
p->data.OccurTime= 41 p->data.NType=4
p->data.OccurTime= 45 p->data.NType=4
p->data.OccurTime= 46 p->data.NType=4
p->data.OccurTime= 47 p->data.NType=4
p->data.OccurTime= 49 p->data.NType=4
```

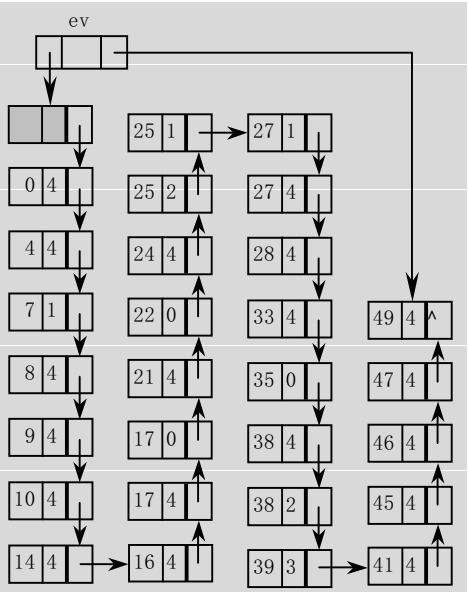


图 3－46　事件表的历史记录

文件 a. txt 的内容是每位客户的到达时刻和他办理业务所需的时间(二者根据随机函数依次生成)以及他所排的队列等信息。排队的原则是排在人数最少的队列，如果有不止一队的人数都同为最少，则排在序号最小的队列。由文件 a. txt 的第 3 条记录可知，第 8 分钟来了一个办理业务需要 17 分钟的客户，这时除 0 队有 1 人外(1 队在第 4 分钟来的人，已在第 7 分钟离去)，1～3 队均无人，客户排在了序号最小的 1 队。

文件 b. txt 的内容是事件表 ev 的历史记录。ev 就象一个安装在门口的监视器，按照时间顺序，记录客户的到达和离去。对于离去的客户，还要记录是由哪个窗口离去。文件 b. txt 的第 3 条记录记下了第 7 分钟在 1 队(1 号窗口)发生了一个离去事件。这和由文件

a.txt 推算的结果(见图 3-45)相吻合。

主程序 main()的第 5 行语句的作用是根据程序运行当前时间的不同，产生不同的随机函数。否则，随机数总是一样的。在调试程序时，希望每次产生相同的随机数，以便分析程序运行结果。而在实际应用时，则希望每次产生不同的随机数。

在运行 algo3-12.cpp 时不会产生 a.txt 和 b.txt 文件，因为有关产生这两个文件的语句已标为注释。在调试程序时，可加一些输出语句帮助分析，最后提交的程序应只输出需要的结果。实际上，迷宫求解问题中，图 3-11(到达终点时栈 S 的内容)的数据也是根据增加的输出语句得出的。

以上是银行开 4 个服务窗口，两相邻客户到达的时间间隔为 0～5 分钟，每个客户办理业务所需的时间为 1～30 分钟的模拟结果。模拟结果显示，客户等候耗时太多。最后一个客户离开时，已 14 个半小时了。如果开 6 个窗口(修改程序 algo3-12.cpp 的第 1 行，定义 Qu 为 6，6 个队列)，程序运行结果如下：

```
请输入银行营业时间长度(单位:分): 480↙
窗口数=6 两相邻到达的客户的时间间隔=0～5分钟 每个客户办理业务的时间=1～30分钟
客户总数:209, 所有客户共耗时:14311分钟,平均每人耗时:68分钟,最后一个客户离开的时间:602分
```

平均每位客户等候耗时大大缩短。

目前银行大多使用排队机，即 1 个队列，多个窗口。algo3-13.cpp 是根据这种情况编制的程序。其中 OpenForDay()、CustomerArrived()和 CustomerDeparture()等 3 个函数与 algo3-12.cpp 中的相应函数是同名函数，但内容不同。而 Bank_Simulation()，甚至 main()函数(除了打开的文件名不同之外)都是相同的。

```
// algo3-13.cpp 使用排队机的银行业务模拟
#define Qu 4 // 窗口数
#define Khjg 5 // 两相邻到达的客户的时间间隔最大值
#define Blsj 30 // 每个客户办理业务的时间最大值
#include"func3-3.cpp" // 包含algo3-12.cpp和algo3-13.cpp共同用到的函数和变量等
LinkQueue q; // 排队机队列q
QElemType customer[Qu]; // Qu个客户队列元素，存放正在窗口办理业务的客户的信息
//FILE *fq; // 文件型指针，用于指向c.txt文件
//int j=0; // 计数器，产生c.txt文件用到
Boolean chk[Qu]; // 窗口状态，1为闲，0为忙
void OpenForDay()
{ // 初始化事件链表ev且插入第1个到达事件，初始化排队机q，初始化Qu个窗口为1(空闲)
  int i;
  InitList(ev); // 初始化事件链表ev为空(见图3-42)
  en.OccurTime=0; // 设定第1位客户到达时间为0(银行一开门，就有客户来)
  en.NType=Qu; // 到达
  OrderInsert(ev,en,cmp); // 将第1个到达事件en有序插入事件表ev中，在bo2-6.cpp中(见图3-43)
  InitQueue(q); // 初始化排队机队列q(见图3-47)
  for(i=0;i<Qu;i++)
    chk[i]=1; // 初始化Qu个窗口为1(空闲) (见图3-48)
}
```

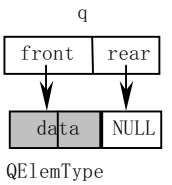


图 3－47　初始化队列 q

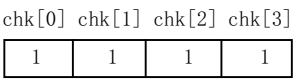


图 3－48　初始化 4 个窗口 chk

```
int ChuangKou()
{ // 返回空闲窗口的序号(0～Qu-1)，若有多个窗口空闲，返回序号最小的。若无空闲窗口，返回Qu
  int i;
  for(i=0;i<Qu;i++)
    if(chk[i])
      return i;
  return i;
}
void CustomerArrived()
{ // 处理客户到达事件en(en.NType=Qu)，与algo3-12.cpp不同
  QElemType f;
  int durtime,intertime,i;
  ++CustomerNum; // 客户数加1
  Random(durtime,intertime);//生成当前客户办理业务的时间和下一个客户到达的时间间隔2个随机数
  et.OccurTime=en.OccurTime+intertime; // 下一客户et到达时刻=当前客户en的到达时间+时间间隔
  et.NType=Qu; // 下一客户到达事件
  if(et.OccurTime<CloseTime) // 下一客户到达时银行尚未关门
    OrderInsert(ev,et,cmp); // 按升序将下一客户到达事件et插入事件表ev中，在bo2-6.cpp中
  f.ArrivalTime=en.OccurTime; // 将当前客户到达事件en赋给队列元素f
  f.Duration=durtime;
  EnQueue(q,f); // 将当前客户f入队到排队机
  i=ChuangKou(); // 求空闲窗口的序号
  if(i<Qu) // 有空闲窗口
  {
    DeQueue(q,customer[i]); // 删去排队机的排头客户(也就是刚入队的f由排队机到i号窗口)
//  if(j++<20) // 输出前20个客户到达信息及处理业务窗口信息到文件c.txt中
//    fprintf(fq,"客户到达时刻=%3d,办理业务的时间=%2d,所在窗口=%d\n",
      customer[i].ArrivalTime,customer[i].Duration,i);
    et.OccurTime=en.OccurTime+customer[i].Duration; // 设定一个i号窗口的离开事件et
    et.NType=i; // 第i号窗口的离开事件
    OrderInsert(ev,et,cmp); // 将此离开事件et按升序插入事件表ev中
    chk[i]=0; // i号窗口状态变忙
  }
}
void CustomerDeparture()
{ // 处理客户离开事件en(en.NType<Qu)，与algo3-12.cpp不同
  int i;
  i=en.NType; // 确定离开事件en发生的窗口序号i
  chk[i]=1; // i号窗口状态变闲
  TotalTime+=en.OccurTime-customer[i].ArrivalTime;
  // 客户逗留时间=离开事件en的发生时刻-该客户的到达时间
```

```
  if(!QueueEmpty(q))
  { // 第i窗口的客户离开后，排队机仍不空
    DeQueue(q,customer[i]);
    // 删去排队机的排头客户并将其赋给customer[i](排队机的排头客户去i窗口办理业务)
//  if(j++<20) // 输出前20个客户到达信息及处理业务窗口信息到文件c.txt中
//    fprintf(fq,"客户到达时刻=%3d,办理业务的时间=%2d,所在窗口=%d\n",
      customer[i].ArrivalTime,customer[i].Duration,i);
    chk[i]=0; // i号窗口状态变忙
    et.OccurTime=en.OccurTime+customer[i].Duration;
    // 设定customer[i]的离开事件et,客户的离开时间等于原客户的
    // 离开时间加当前客户办理业务的时间
    et.NType=i; // 第i号窗口的离开事件
    OrderInsert(ev,et,cmp); // 将此离开事件et按升序插入事件表ev中
  }
}
void main()
{
//fq=fopen("c.txt","w"); // 打开c.txt文件，用于写入客户到达信息
//fp=fopen("d.txt","w"); // 打开d.txt文件，用于写入有序事件表的历史记录
  printf("请输入银行营业时间长度(单位:分): ");
  scanf("%d",&CloseTime);
//srand(time(0));
  // 设置随机数种子,以使每次运行程序产生的随机数不同(time(0)是长整型数,与调用时间有关)
  Bank_Simulation();
//fclose(fq); // 关闭c.txt
//fclose(fp); // 关闭d.txt
}
```

程序运行结果：

```
请输入银行营业时间长度(单位:分): 480↙
窗口数=4 两相邻到达的客户的时间间隔=0～5分钟 每个客户办理业务的时间=1～30分钟
客户总数:209, 所有客户共耗时:41815分钟,平均每人耗时:200分钟,最后一个客户离开的时间:835分
```

文件 c.txt 的内容(见图 3－49)：

```
客户到达时刻=  0,办理业务的时间=17,所在窗口=0
客户到达时刻=  4,办理业务的时间= 3,所在窗口=1
客户到达时刻=  8,办理业务的时间=17,所在窗口=1
客户到达时刻=  9,办理业务的时间=16,所在窗口=2
客户到达时刻= 10,办理业务的时间=29,所在窗口=3
客户到达时刻= 14,办理业务的时间= 5,所在窗口=0
客户到达时刻= 16,办理业务的时间= 2,所在窗口=0
客户到达时刻= 17,办理业务的时间=13,所在窗口=0
客户到达时刻= 21,办理业务的时间=13,所在窗口=2
客户到达时刻= 24,办理业务的时间=24,所在窗口=1
客户到达时刻= 27,办理业务的时间=30,所在窗口=0
客户到达时刻= 28,办理业务的时间=21,所在窗口=2
```

chk[0]	chk[1]	chk[2]	chk[3]
0 17	4 3	9 16	10 29
14 5	8 17	21 13	33 15
16 2	24 24	28 21	41 20
17 13	38 13	45 22	49 20
27 30	46 26	54 22	
47 15			

图 3－49　4 个窗口的历史记录

```
客户到达时刻= 33,办理业务的时间=15,所在窗口=3
客户到达时刻= 38,办理业务的时间=13,所在窗口=1
客户到达时刻= 41,办理业务的时间=20,所在窗口=3
客户到达时刻= 45,办理业务的时间=22,所在窗口=2
客户到达时刻= 46,办理业务的时间=26,所在窗口=1
客户到达时刻= 47,办理业务的时间=15,所在窗口=0
客户到达时刻= 49,办理业务的时间=20,所在窗口=3
客户到达时刻= 54,办理业务的时间=22,所在窗口=2
```

文件 d.txt 的内容(见图 3-50):

```
p->data.OccurTime=  0 p->data.NType=4
p->data.OccurTime=  4 p->data.NType=4
p->data.OccurTime=  7 p->data.NType=1
p->data.OccurTime=  8 p->data.NType=4
p->data.OccurTime=  9 p->data.NType=4
p->data.OccurTime= 10 p->data.NType=4
p->data.OccurTime= 14 p->data.NType=4
p->data.OccurTime= 16 p->data.NType=4
p->data.OccurTime= 17 p->data.NType=4
p->data.OccurTime= 17 p->data.NType=0
p->data.OccurTime= 21 p->data.NType=4
p->data.OccurTime= 22 p->data.NType=0
p->data.OccurTime= 24 p->data.NType=0
p->data.OccurTime= 24 p->data.NType=4
p->data.OccurTime= 25 p->data.NType=2
p->data.OccurTime= 25 p->data.NType=1
p->data.OccurTime= 27 p->data.NType=4
p->data.OccurTime= 28 p->data.NType=4
p->data.OccurTime= 33 p->data.NType=4
p->data.OccurTime= 37 p->data.NType=0
p->data.OccurTime= 38 p->data.NType=4
p->data.OccurTime= 38 p->data.NType=2
p->data.OccurTime= 39 p->data.NType=3
p->data.OccurTime= 41 p->data.NType=4
p->data.OccurTime= 45 p->data.NType=4
p->data.OccurTime= 46 p->data.NType=4
p->data.OccurTime= 47 p->data.NType=4
p->data.OccurTime= 49 p->data.NType=4
p->data.OccurTime= 49 p->data.NType=1
```

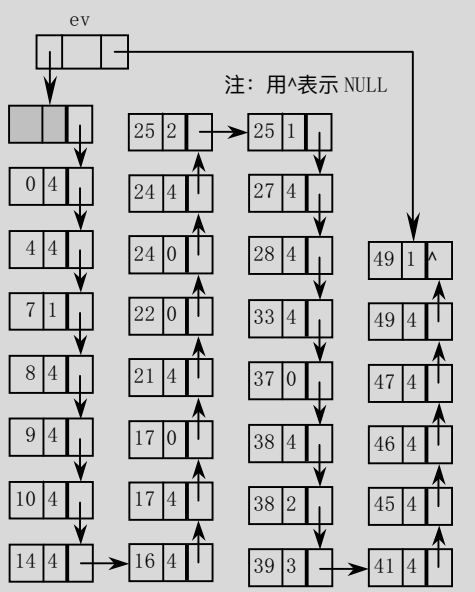


图 3-50　事件表的历史记录

和 algo3-12.cpp 的运行结果相比，在营业时间、营业窗口数和客户总数相同条件下，algo3-13.cpp 的平均每人耗时和最后一个客户离开的时间比 algo3-12.cpp 要略小。因为没有调用 srand()函数，两程序产生的随机数是一样的(通过比较文件 a.txt 和 c.txt 的内容可看出)。不同的是，客户所排队列或窗口不完全相同。algo3-12.cpp 在关门后(不再有客户进入)可能出现某队已空，而其他队还有若干人的情况。这就延长了最后一个客户离开的时间，也增大了平均每人耗时。

修改程序 algo3-13. cpp 的第 1 行，定义 Qu 为 6，6 个队列，程序运行结果如下：

```
请输入银行营业时间长度(单位:分): 480↙
窗口数=6 两相邻到达的客户的时间间隔=0～5分钟 每个客户办理业务的时间=1～30分钟
客户总数:209, 所有客户共耗时:13946分钟,平均每人耗时:66分钟,最后一个客户离开的时间:566分
```

第 4 章　串

4.1　串类型的定义

在 C 语言中，字符串存于字符型数组中。无论数组有多大，用数值 0 表示串结束。图 4－1 表示了“ but”字符串在 C 语言中的存储结构。其中数组 a 的定义为

```
char a[10];
```

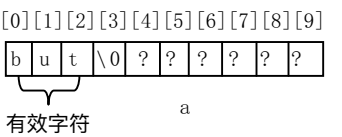


图 4－1　“but”在 C 语言中的存储结构

C 语言还在库函数 string.h 中提供了许多串处理的基本操作，如求串长函数 strlen()、串拷贝函数 strcpy()等。

算法语言本身提供的字符串存储结构及其基本操作不一定能满足实际应用的需要，我们往往还要根据具体情况另外定义字符串的存储结构及基于该存储结构的基本操作。

4.2　串的表示和实现

4.2.1　定长顺序存储结构

```
// c4-1.h 串的定长顺序存储结构(见图4－2)
#define MAX_STR_LEN 40 // 用户可在255(1个字节)以内定义最大串长
typedef char SString[MAX_STR_LEN+1]; // 0号单元存放串的长度
```

```
// bo4-1.cpp 串采用定长顺序存储结构(由c4-1.h定义)的基本操作(13个)，包括算法4.2，4.3，4.5
// SString是数组，故不需引用类型
#define DestroyString ClearString // DestroyString()与ClearString()作用相同
Status StrAssign(SString T,char *chars)
{ // 生成一个其值等于chars的串T
  int i;
  if(strlen(chars)>MAX_STR_LEN)
    return ERROR;
  else
  {
    T[0]=strlen(chars);
    for(i=1;i<=T[0];i++)
      T[i]=*(chars+i-1);
    return OK;
  }
```

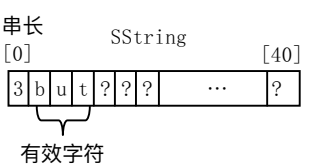


图 4－2　SString 类型

```
}
void StrCopy(SString T,SString S)
{ // 由串S复制得串T
  int i;
  for(i=0;i<=S[0];i++)
    T[i]=S[i];
}
Status StrEmpty(SString S)
{ // 若S为空串，则返回TRUE；否则返回FALSE
  if(S[0]==0)
    return TRUE;
  else
    return FALSE;
}
int StrCompare(SString S,SString T)
{// 初始条件：串S和T存在。操作结果：若S>T，则返回值>0；若S=T，则返回值=0；若S<T，则返回值<0
  int i;
  for(i=1;i<=S[0]&&i<=T[0];++i)
    if(S[i]!=T[i])
      return S[i]-T[i];
  return S[0]-T[0];
}
int StrLength(SString S)
{ // 返回串S的元素个数
  return S[0];
}
void ClearString(SString S)
{ // 初始条件：串S存在。操作结果：将S清为空串(见图4－3)
  S[0]=0; // 令串长为零
}
Status Concat(SString T,SString S1,SString S2) // 算法4.2改
{ // 用T返回S1和S2联接而成的新串。若未截断，则返回TRUE；否则返回FALSE
  int i;
  if(S1[0]+S2[0]<=MAX_STR_LEN)
  { // 未截断
    for(i=1;i<=S1[0];i++)
      T[i]=S1[i];
    for(i=1;i<=S2[0];i++)
      T[S1[0]+i]=S2[i];
    T[0]=S1[0]+S2[0];
    return TRUE;
  }
  else
  { // 截断S2
    for(i=1;i<=S1[0];i++)
      T[i]=S1[i];
    for(i=1;i<=MAX_STR_LEN-S1[0];i++)
      T[S1[0]+i]=S2[i];
    T[0]=MAX_STR_LEN;
    return FALSE;
  }
```

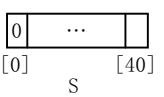


图4－3　空串S

```
}
Status SubString(SString Sub,SString S,int pos,int len)
{ // 用Sub返回串S的自第pos个字符起长度为len的子串。算法4.3
  int i;
  if(pos<1||pos>S[0]||len<0||len>S[0]-pos+1)
    return ERROR;
  for(i=1;i<=len;i++)
    Sub[i]=S[pos+i-1];
  Sub[0]=len;
  return OK;
}
int Index(SString S,SString T,int pos)
{ // 返回子串T在主串S中第pos个字符之后的位置。若不存在，则函数值为0。
  // 其中，T非空，1≤pos≤StrLength(S)。算法4.5
  int i,j;
  if(1<=pos&&pos<=S[0])
  {
    i=pos;
    j=1;
    while(i<=S[0]&&j<=T[0])
      if(S[i]==T[j]) // 继续比较后继字符
      {
        ++i;
        ++j;
      }
      else // 指针后退重新开始匹配
      {
        i=i-j+2;
        j=1;
      }
    if(j>T[0])
      return i-T[0];
    else
      return 0;
  }
  else
    return 0;
}
Status StrInsert(SString S,int pos,SString T)
{ // 初始条件：串S和T存在，1≤pos≤StrLength(S)+1
  // 操作结果：在串S的第pos个字符之前插入串T。完全插入返回TRUE，部分插入返回FALSE
  int i;
  if(pos<1||pos>S[0]+1)
    return ERROR;
  if(S[0]+T[0]<=MAX_STR_LEN)
  { // 完全插入
    for(i=S[0];i>=pos;i--)
      S[i+T[0]]=S[i];
    for(i=pos;i<pos+T[0];i++)
      S[i]=T[i-pos+1];
    S[0]+=T[0];
```

```
    return TRUE;
  }
  else
  { // 部分插入
    for(i=MAX_STR_LEN;i>=pos+T[0];i--)
      S[i]=S[i-T[0]];
    for(i=pos;i<pos+T[0]&&i<=MAX_STR_LEN;i++)
      S[i]=T[i-pos+1];
    S[0]=MAX_STR_LEN;
    return FALSE;
  }
}
Status StrDelete(SString S,int pos,int len)
{ // 初始条件：串S存在，1≤pos≤StrLength(S)-len+1
  // 操作结果：从串S中删除自第pos个字符起长度为len的子串
  int i;
  if(pos<1||pos>S[0]-len+1||len<0)
    return ERROR;
  for(i=pos+len;i<=S[0];i++)
    S[i-len]=S[i];
  S[0]-=len;
  return OK;
}
Status Replace(SString S,SString T,SString V) // 此函数与串的存储结构无关
{ // 初始条件：串S，T和V存在，T是非空串
  // 操作结果：用V替换主串S中出现的所有与T相等的不重叠的子串
  int i=1; // 从串S的第一个字符起查找串T
  Status k;
  if(StrEmpty(T)) // T是空串
    return ERROR;
  do
  {
    i=Index(S,T,i); // 结果i为从上一个i之后找到的子串T的位置
    if(i) // 串S中存在串T
    {
      StrDelete(S,i,StrLength(T)); // 删除该串T
      k=StrInsert(S,i,V); // 在原串T的位置插入串V
      if(!k) // 不能完全插入
        return ERROR;
      i+=StrLength(V); // 在插入的串V后面继续查找串T
    }
  }while(i);
  return OK;
}
void StrPrint(SString T)
{ // 输出字符串T。另加
  int i;
  for(i=1;i<=T[0];i++)
    printf("%c",T[i]);
  printf("\n");
}
```

```
// main4-1.cpp 检验bo4-1.cpp的主程序
#include"c1.h"
#include"c4-1.h"
#include"bo4-1.cpp"
void main()
{
  int i,j;
  Status k;
  char s,c[MAX_STR_LEN+1];
  SString t,s1,s2;
  printf("请输入串s1: ");
  gets(c);
  k=StrAssign(s1,c);
  if(!k)
  {
    printf("串长超过MAX_STR_LEN(=%d)\n",MAX_STR_LEN);
    exit(0);
  }
  printf("串长为%d 串空否? %d(1:是 0:否)\n",StrLength(s1),StrEmpty(s1));
  StrCopy(s2,s1);
  printf("拷贝s1生成的串为");
  StrPrint(s2);
  printf("请输入串s2: ");
  gets(c);
  k=StrAssign(s2,c);
  if(!k)
  {
    printf("串长超过MAX_STR_LEN(%d)\n",MAX_STR_LEN);
    exit(0);
  }
  i=StrCompare(s1,s2);
  if(i<0)
    s='<';
  else if(i==0)
    s='=';
  else
    s='>';
  printf("串s1%c串s2\n",s);
  k=Concat(t,s1,s2);
  printf("串s1联接串s2得到的串t为");
  StrPrint(t);
  if(k==FALSE)
    printf("串t有截断\n");
  ClearString(s1);
  printf("清为空串后,串s1为");
  StrPrint(s1);
  printf("串长为%d 串空否? %d(1:是 0:否)\n",StrLength(s1),StrEmpty(s1));
  printf("求串t的子串,请输入子串的起始位置,子串长度: ");
  scanf("%d,%d",&i,&j);
  k=SubString(s2,t,i,j);
  if(k)
  {
```

```
    printf("子串s2为");
    StrPrint(s2);
  }
  printf("从串t的第pos个字符起,删除len个字符, 请输入pos,len: ");
  scanf("%d,%d",&i,&j);
  StrDelete(t,i,j);
  printf("删除后的串t为");
  StrPrint(t);
  i=StrLength(s2)/2;
  StrInsert(s2,i,t);
  printf("在串s2的第%d个字符之前插入串t后,串s2为\n",i);
  StrPrint(s2);
  i=Index(s2,t,1);
  printf("s2的第%d个字母起和t第一次匹配\n",i);
  SubString(t,s2,1,1);
  printf("串t为");
  StrPrint(t);
  Concat(s1,t,t);
  printf("串s1为");
  StrPrint(s1);
  k=Replace(s2,t,s1);
  if(k) // 替换成功
  {
    printf("用串s1取代串s2中和串t相同的不重叠的串后,串s2为");
    StrPrint(s2);
  }
  DestroyString(s2); // 销毁操作同清空
}
```

程序运行结果:

```
请输入串s1: ABCD↙
串长为4 串空否? 0(1:是 0:否)
拷贝s1生成的串为ABCD
请输入串s2: 123456↙
串s1>串s2
串s1联接串s2得到的串t为ABCD123456
清为空串后,串s1为
串长为0 串空否? 1(1:是 0:否)
求串t的子串,请输入子串的起始位置,子串长度: 3,7↙
子串s2为CD12345
从串t的第pos个字符起,删除len个字符, 请输入pos,len: 4,4↙
删除后的串t为ABC456
在串s2的第3个字符之前插入串t后,串s2为
CDABC45612345
s2的第3个字母起和t第一次匹配
串t为C
串s1为CC
用串s1取代串s2中和串t相同的不重叠的串后,串s2为CCDABCC45612345
```

4.2.2　堆分配存储结构

```
// c4-2.h 串的堆分配存储(见图4－4)
struct HString
{
  char *ch; // 若是非空串，则按串长分配存储区；否则ch为NULL
  int length; // 串长度
};
```

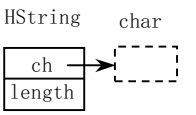


图 4－4　堆分配存储结构

```
// bo4-2.cpp 串采用堆分配存储结构(由c4-2.h定义)的基本操作(14个)。包括算法4.1，4.4
#define DestroyString ClearString // DestroyString()与ClearString()作用相同
void StrAssign(HString &T,char *chars)
{ // 生成一个其值等于串常量chars的串T(见图4－5)
  int i,j;
  if(T.ch)
    free(T.ch); // 释放T原有空间
  i=strlen(chars); // 求chars的长度i
  if(!i)
  { // chars的长度为0
    T.ch=NULL;
    T.length=0;
  }
  else
  { // chars的长度不为0
    T.ch=(char*)malloc(i*sizeof(char)); // 分配串空间
    if(!T.ch) // 分配串空间失败
      exit(OVERFLOW);
    for(j=0;j<i;j++) // 拷贝串
      T.ch[j]=chars[j];
    T.length=i;
  }
}
void StrCopy(HString &T,HString S)
{ // 初始条件：串S存在。操作结果：由串S复制得串T
  int i;
  if(T.ch)
    free(T.ch); // 释放T原有空间
  T.ch=(char*)malloc(S.length*sizeof(char)); // 分配串空间
  if(!T.ch) // 分配串空间失败
    exit(OVERFLOW);
  for(i=0;i<S.length;i++) // 拷贝串
    T.ch[i]=S.ch[i];
  T.length=S.length;
}
Status StrEmpty(HString S)
{ // 初始条件：串S存在。操作结果：若S为空串，则返回TRUE；否则返回FALSE
  if(S.length==0&&S.ch==NULL)
    return TRUE;
  else
    return FALSE;
```

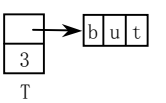


图 4－5　“ but”的堆存储结构

```
}
int StrCompare(HString S,HString T)
{ // 若S>T，则返回值>0；若S=T，则返回值=0；若S<T，则返回值<0
  int i;
  for(i=0;i<S.length&&i<T.length;++i)
    if(S.ch[i]!=T.ch[i])
      return S.ch[i]-T.ch[i];
  return S.length-T.length;
}
int StrLength(HString S)
{ // 返回S的元素个数，称为串的长度
  return S.length;
}
void ClearString(HString &S)
{ // 将S清为空串(见图4－6)
  free(S.ch);
  S.ch=NULL;
  S.length=0;
}
void Concat(HString &T,HString S1,HString S2)
{ // 用T返回由S1和S2联接而成的新串
  int i;
  if(T.ch)
    free(T.ch); // 释放旧空间
  T.length=S1.length+S2.length;
  T.ch=(char *)malloc(T.length*sizeof(char));
  if(!T.ch)
    exit(OVERFLOW);
  for(i=0;i<S1.length;i++)
    T.ch[i]=S1.ch[i];
  for(i=0;i<S2.length;i++)
    T.ch[S1.length+i]=S2.ch[i];
}
Status SubString(HString &Sub, HString S,int pos,int len)
{ // 用Sub返回串S的第pos个字符起长度为len的子串。
  // 其中,1≤pos≤StrLength(S)且0≤len≤StrLength(S)-pos+1
  int i;
  if(pos<1||pos>S.length||len<0||len>S.length-pos+1)
    return ERROR;
  if(Sub.ch)
    free(Sub.ch); // 释放旧空间
  if(!len) // 空子串
  {
    Sub.ch=NULL;
    Sub.length=0;
  }
  else
  { // 完整子串
    Sub.ch=(char*)malloc(len*sizeof(char));
    if(!Sub.ch)
```

图 4－6　空串 S 的结构

```
      exit(OVERFLOW);
    for(i=0;i<=len-1;i++)
      Sub.ch[i]=S.ch[pos-1+i];
    Sub.length=len;
  }
  return OK;
}
void InitString(HString &T)
{ // 初始化(产生空串)字符串T。另加
  T.length=0;
  T.ch=NULL;
}
int Index(HString S,HString T,int pos) // 算法4.1
{ // T为非空串。若主串S中第pos个字符之后存在与T相等的子串,
  // 则返回第一个这样的子串在S中的位置；否则返回0
  int n,m,i;
  HString sub;
  InitString(sub);
  if(pos>0)
  {
    n=StrLength(S);
    m=StrLength(T);
    i=pos;
    while(i<=n-m+1)
    {
      SubString(sub,S,i,m);
      if(StrCompare(sub,T)!=0)
        ++i;
      else
        return i;
    }
  }
  return 0;
}
Status StrInsert(HString &S,int pos,HString T) // 算法4.4
{ // 1≤pos≤StrLength(S)+1。在串S的第pos个字符之前插入串T
  int i;
  if(pos<1||pos>S.length+1) // pos不合法
    return ERROR;
  if(T.length) // T非空，则重新分配空间，插入T
  {
    S.ch=(char*)realloc(S.ch,(S.length+T.length)*sizeof(char));
    if(!S.ch)
      exit(OVERFLOW);
    for(i=S.length-1;i>=pos-1;--i) // 为插入T而腾出位置
      S.ch[i+T.length]=S.ch[i];
    for(i=0;i<T.length;i++)
      S.ch[pos-1+i]=T.ch[i]; // 插入T
    S.length+=T.length;
  }
```

```
  return OK;
}
Status StrDelete(HString &S,int pos,int len)
{ // 从串S中删除第pos个字符起长度为len的子串
  int i;
  if(S.length<pos+len-1)
    return ERROR;
  for(i=pos-1;i<=S.length-len;i++)
    S.ch[i]=S.ch[i+len];
  S.length-=len;
  S.ch=(char*)realloc(S.ch,S.length*sizeof(char));
  return OK;
}
Status Replace(HString &S,HString T,HString V) // 此函数与串的存储结构无关
{ // 初始条件: 串S, T和V存在, T是非空串
  // 操作结果: 用V替换主串S中出现的所有与T相等的不重叠的子串
  int i=1; // 从串S的第一个字符起查找串T
  if(StrEmpty(T)) // T是空串
    return ERROR;
  do
  {
    i=Index(S,T,i); // 结果i为从上一个i之后找到的子串T的位置
    if(i) // 串S中存在串T
    {
      StrDelete(S,i,StrLength(T)); // 删除该串T
      StrInsert(S,i,V); // 在原串T的位置插入串V
      i+=StrLength(V); // 在插入的串V后面继续查找串T
    }
  }while(i);
  return OK;
}
void StrPrint(HString T)
{ // 输出T字符串。另加
  int i;
  for(i=0;i<T.length;i++)
    printf("%c",T.ch[i]);
  printf("\n");
}

// main4-2.cpp 检验bo4-2.cpp的主程序
#include"c1.h"
#include"c4-2.h"
#include"bo4-2.cpp"
void main()
{
  int i;
  char c,*p="God bye!",*q="God luck!";
  HString t,s,r;
  InitString(t); // HString类型必须初始化
  InitString(s);
  InitString(r);
```

```
  StrAssign(t,p);
  printf("串t为");
  StrPrint(t);
  printf("串长为%d 串空否? %d(1:空 0:否)\n",StrLength(t),StrEmpty(t));
  StrAssign(s,q);
  printf("串s为");
  StrPrint(s);
  i=StrCompare(s,t);
  if(i<0)
    c='<';
  else if(i==0)
    c='=';
  else
    c='>';
  printf("串s%c串t\n",c);
  Concat(r,t,s);
  printf("串t联接串s产生的串r为");
  StrPrint(r);
  StrAssign(s,"oo");
  printf("串s为");
  StrPrint(s);
  StrAssign(t,"o");
  printf("串t为");
  StrPrint(t);
  Replace(r,t,s);
  printf("把串r中和串t相同的子串用串s代替后, 串r为");
  StrPrint(r);
  ClearString(s);
  printf("串s清空后, 串长为%d 空否? %d(1:空 0:否)\n",StrLength(s),StrEmpty(s));
  SubString(s,r,6,4);
  printf("串s为从串r的第6个字符起的4个字符, 长度为%d 串s为",s.length);
  StrPrint(s);
  StrCopy(t,r);
  printf("复制串t为串r,串t为");
  StrPrint(t);
  StrInsert(t,6,s);
  printf("在串t的第6个字符前插入串s后, 串t为");
  StrPrint(t);
  StrDelete(t,1,5);
  printf("从串t的第1个字符起删除5个字符后,串t为");
  StrPrint(t);
  printf("%d是从串t的第1个字符起, 和串s相同的第1个子串的位置\n",Index(t,s,1));
  printf("%d是从串t的第2个字符起, 和串s相同的第1个子串的位置\n",Index(t,s,2));
  DestroyString(t); // 销毁操作同清空
```

程序运行结果:

```
串t为God bye!
串长为8 串空否? 0(1:空 0:否)
串s为God luck!
```

```
串s>串t
串t联接串s产生的串r为God bye!God luck!
串s为oo
串t为o
把串r中和串t相同的子串用串s代替后，串r为Good bye!Good luck!
串s清空后，串长为0 空否? 1(1:空 0:否)
串s为从串r的第6个字符起的4个字符，长度为4 串s为bye!
复制串t为串r,串t为Good bye!Good luck!
在串t的第6个字符前插入串s后，串t为Good bye!bye!Good luck!
从串t的第1个字符起删除5个字符后,串t为bye!bye!Good luck!
1是从串t的第1个字符起，和串s相同的第1个子串的位置
5是从串t的第2个字符起，和串s相同的第1个子串的位置
```

串的堆分配存储结构(由 c4-2.h 定义)根据串的长度，动态地分配存储空间。这样既保证满足需要，又不浪费空间，对于串长也没有限制。串的定长存储结构(由 c4-1.h 定义)就没有这样灵活了。在 Concat()、StrInsert()和 Replace()中，总要检查串是否被截断，且对于短串的情况，空间浪费较大。故堆分配存储结构较好。

4.2.3　串的块链存储结构

```
// c4-3.h 串的块链存储结构(见图4－7)
#define CHUNK_SIZE 4 // 可由用户定义的块大小
struct Chunk
{
  char ch[CHUNK_SIZE];
  Chunk *next;
};
struct LString
{
  Chunk *head,*tail; // 串的头和尾指针
  int curlen; // 串的当前长度
};
```

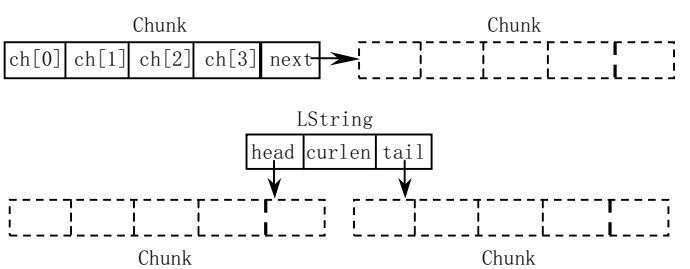


图 4－7　串的块链存储结构

图 4－8 是根据 c4-3.h 定义的串“ABCDEFGHI”的一种可能的存储形式(“#”作为填补空余的字符，不计为串的字符)。

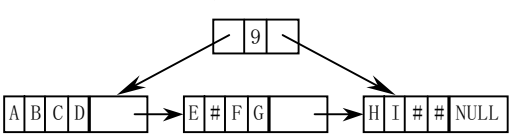


图 4－8　串“ABCDEFGHI”的一种块链存储结

```
// bo4-3.cpp 串采用块链存储结构(由c4-3.h定义)的基本操作(15个)
#define DestroyString ClearString // DestroyString()与ClearString()作用相同
void InitString(LString &T)
{ // 初始化(产生空串)字符串T。另加(见图4-9)
  T.curlen=0;
  T.head= T.tail=NULL;
}
```

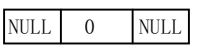


图 4-9　空串结构

```
Status StrAssign(LString &T,char *chars)
{ // 生成一个其值等于chars的串T(要求chars中不包含填补空余的字符)。成功返回OK；否则返回ERROR
  int i,j,k,m;
  Chunk *p,*q;
  i=strlen(chars); // i为串的长度
  if(!i||strchr(chars,blank)) // 串长为0或chars中包含填补空余的字符
    return ERROR;
  T.curlen=i;
  j=i/CHUNK_SIZE; // j为块链的结点数
  if(i%CHUNK_SIZE)
    j++;
  for(k=0;k<j;k++)
  {
    p=(Chunk*)malloc(sizeof(Chunk)); // 生成块结点
    if(!p) // 生成块结点失败
      return ERROR;
    for(m=0;m<CHUNK_SIZE&&*chars;m++) // 给块结点的数据域赋值
      *(p->ch+m)=*chars++;
    if(k==0) // 第一个链块
      T.head=q=p; // 头指针指向第一个链块
    else
    {
      q->next=p;
      q=p;
    }
    if(!*chars) // 最后一个链块
    {
      T.tail=q;
      q->next=NULL;
      for(;m<CHUNK_SIZE;m++) // 用填补空余的字符填满链表
        *(q->ch+m)=blank;
    }
  }
  return OK;
}
Status ToChars(LString T,char* &chars)
{ // 将串T的内容转换为字符串，chars为其头指针。成功返回OK；否则返回ERROR。另加
  Chunk *p=T.head; // p指向第1个块结点
  int i;
  char *q;
  chars=(char*)malloc((T.curlen+1)*sizeof(char));
  if(!chars||!T.curlen) // 生成字符串数组失败或串T长为0
    return ERROR;
  q=chars; // q指向chars的第1个字符
```

```
  while(p) // 块链没结束
  {
    for(i=0;i<CHUNK_SIZE;i++)
      if(p->ch[i]!=blank) // 当前字符不是填补空余的字符
        *q++=(p->ch[i]); // 赋给q所指字符空间
    p=p->next;
  }
  chars[T.curlen]=0; // 串结束符
  return OK;
}
Status StrCopy(LString &T,LString S)
{ // 初始条件：串S存在
  // 操作结果：由串S复制得串T，去掉填补空余的字符。成功返回OK；否则返回ERROR
  char *c;
  Status i;
  if(!ToChars(S,c)) // c为串S的内容
    return ERROR;
  i=StrAssign(T,c); // 将串S的内容赋给T
  free(c); // 释放c的空间
  return i;
}
Status StrEmpty(LString S)
{ // 初始条件：串S存在。操作结果：若S为空串，则返回TRUE；否则返回FALSE
  if(S.curlen) // 非空
    return FALSE;
  else
    return TRUE;
}
int StrCompare(LString S,LString T)
{ // 若S>T，则返回值>0；若S=T，则返回值=0；若S<T，则返回值<0
  char *s,*t;
  Status i;
  if(!ToChars(S,s)) // s为串S的内容
    return ERROR;
  if(!ToChars(T,t)) // t为串T的内容
    return ERROR;
  i=strcmp(s,t); // 利用C的库函数
  free(s); // 释放s，t的空间
  free(t);
  return i;
}
int StrLength(LString S)
{ // 返回S的元素个数，称为串的长度
  return S.curlen;
}
void ClearString(LString &S)
{ // 初始条件：串S存在。操作结果：将S清为空串
  Chunk *p,*q;
  p=S.head;
  while(p)
  {
```

```
    q=p->next;
    free(p);
    p=q;
  }
  S.head=S.tail=NULL;
  S.curlen=0;
}
Status Concat(LString &T,LString S1,LString S2)
{ // 用T返回由S1和S2联接而成的新串(中间可能有填补空余的字符)
  LString a1,a2;
  Status i,j;
  InitString(a1);
  InitString(a2);
  i=StrCopy(a1,S1);
  j=StrCopy(a2,S2);
  if(!i||!j) // 至少有1个串拷贝不成功
    return ERROR;
  T.curlen=S1.curlen+S2.curlen; // 生成串T
  T.head=a1.head;
  a1.tail->next=a2.head; // a1,a2两串首尾相连
  T.tail=a2.tail;
  return OK;
}
Status SubString(LString &Sub, LString S,int pos,int len)
{ // 用Sub返回串S的第pos个字符起长度为len的子串。
  // 其中，1≤pos≤StrLength(S)且0≤len≤StrLength(S)-pos+1
  char *b,*c;
  Status i;
  if(pos<1||pos>S.curlen||len<0||len>S.curlen-pos+1) // pos或len值不合法
    return ERROR;
  if(!ToChars(S,c)) // c为串S的内容
    return ERROR;
  b=c+pos-1; // b指向串c中串Sub内容的首地址
  b[len]=0; // Sub结束处赋0(字符串结束符)
  i=StrAssign(Sub,b); // 将串b的内容赋给Sub
  free(c);
  return i;
}
int Index(LString S,LString T,int pos)
{ // T为非空串。若主串S中第pos个字符之后存在与T相等的子串，
  // 则返回第一个这样的子串在S中的位置，否则返回0
  int i,n,m;
  LString sub;
  if(pos>=1&&pos<=StrLength(S)) // pos满足条件
  {
    n=StrLength(S); // 主串长度
    m=StrLength(T); // 串T长度
    i=pos;
    while(i<=n-m+1)
    {
      SubString(sub,S,i,m); // sub为从S的第i个字符起，长度为m的子串
```

```
      if(StrCompare(sub,T)) // sub不等于T
        ++i;
      else
        return i;
    }
  }
  return 0;
}
Status StrInsert(LString &S, int pos,LString T)
{ // 1≤pos≤StrLength(S)+1。在串S的第pos个字符之前插入串T
  char *b,*c;
  int i,j;
  Status k;
  if(pos<1||pos>S.curlen+1) // pos的值超出范围
    return ERROR;
  if(!ToChars(S,c)) // c为串S的内容
    return ERROR;
  if(!ToChars(T,b)) // b为串T的内容
    return ERROR;
  j=(int)strlen(c); // j为串S的最初长度
  c=(char*)realloc(c,(j+strlen(b)+1)*sizeof(char)); // 增加c的存储空间
  for(i=j;i>=pos-1;i--)
    c[i+strlen(b)]=c[i]; // 为插入串b腾出空间
  for(i=0;i<(int)strlen(b);i++) // 在串c中插入串b
    c[pos+i-1]=b[i];
  InitString(S); // 释放S的原有存储空间
  k=StrAssign(S,c); // 由c生成新的串S
  free(b);
  free(c);
  return k;
}
Status StrDelete(LString &S,int pos,int len)
{ // 从串S中删除第pos个字符起长度为len的子串
  char *c;
  int i;
  Status k;
  if(pos<1||pos>S.curlen-len+1||len<0) // pos, len的值超出范围
    return ERROR;
  if(!ToChars(S,c)) // c为串S的内容
    return ERROR;
  for(i=pos+len-1;i<=(int)strlen(c);i++)
    c[i-len]=c[i]; // c为删除后串S的内容
  InitString(S); // 释放S的原有存储空间
  k=StrAssign(S,c); // 由c生成新的串S
  free(c);
  return k;
}
Status Replace(LString &S,LString T,LString V) // 此函数与串的存储结构无关
{ // 初始条件: 串S, T和V存在, T是非空串
  // 操作结果: 用V替换主串S中出现的所有与T相等的不重叠的子串
  int i=1; // 从串S的第一个字符起查找串T
```

```
  if(StrEmpty(T)) // T是空串
    return ERROR;
  do
  {
    i=Index(S,T,i); // 结果i为从上一个i之后找到的子串T的位置
    if(i) // 串S中存在串T
    {
      StrDelete(S,i,StrLength(T)); // 删除该串T
      StrInsert(S,i,V); // 在原串T的位置插入串V
      i+=StrLength(V); // 在插入的串V后面继续查找串T
    }
  }while(i);
  return OK;
}
void StrPrint(LString T)
{ // 输出字符串T。另加
  int i=0,j;
  Chunk *h;
  h=T.head;
  while(i<T.curlen)
  {
    for(j=0;j<CHUNK_SIZE;j++)
      if(*(h->ch+j)!=blank) // 不是填补空余的字符
      {
        printf("%c",*(h->ch+j));
        i++;
      }
    h=h->next;
  }
  printf("\n");
}

// main4-3.cpp 检验bo4-3.cpp的主程序
char blank='#'; // 全局变量，用于填补空余
#include"c1.h"
#include"c4-3.h"
#include"bo4-3.cpp"
void main()
{
  char *s1="ABCDEFGHI",*s2="12345",*s3="",*s4="asd#tr",*s5="ABCD";
  Status k;
  int pos,len;
  LString t1,t2,t3,t4;
  InitString(t1);
  InitString(t2);
  printf("初始化串t1后，串t1空否? %d(1:空 0:否) 串长=%d\n",StrEmpty(t1),StrLength(t1));
  k=StrAssign(t1,s3);
  if(k==ERROR)
    printf("出错\n"); // 不能生成空串
  k=StrAssign(t1,s4);
  if(k==ERROR)
```

```
  printf("出错\n"); // 不能生成含有变量blank所代表的字符的串
k=StrAssign(t1,s1);
if(k==OK)
{
  printf("串t1为");
  StrPrint(t1);
}
else
  printf("出错\n");
printf("串t1空否? %d(1:空 0:否) 串长=%d\n",StrEmpty(t1),StrLength(t1));
StrAssign(t2,s2);
printf("串t2为");
StrPrint(t2);
StrCopy(t3,t1);
printf("由串t1拷贝得到串t3,串t3为");
StrPrint(t3);
InitString(t4);
StrAssign(t4,s5);
printf("串t4为");
StrPrint(t4);
Replace(t3,t4,t2);
printf("用t2取代串t3中的t4串后, 串t3为");
StrPrint(t3);
ClearString(t1);
printf("清空串t1后, 串t1空否? %d(1:空 0:否) 串长=%d\n",StrEmpty(t1),StrLength(t1));
Concat(t1,t2,t3);
printf("串t1(=t2+t3)为");
StrPrint(t1);
pos=Index(t1,t3,1);
printf("pos=%d\n",pos);
printf("在串t1的第pos个字符之前插入串t2, 请输入pos: ");
scanf("%d",&pos);
k=StrInsert(t1,pos,t2);
if(k)
{
  printf("插入串t2后, 串t1为");
  StrPrint(t1);
}
else
  printf("插入失败! \n");
printf("求从t1的第pos个字符起,长度为len的子串t2,请输入pos,len: ");
scanf("%d,%d",&pos,&len);
SubString(t2,t1,pos,len);
printf("串t2为");
StrPrint(t2);
printf("StrCompare(t1,t2)=%d\n",StrCompare(t1,t2));
printf("删除串t1中的子字符串: 从第pos个字符起删除len个字符。请输入pos,len: ");
scanf("%d,%d",&pos,&len);
k=StrDelete(t1,pos,len);
if(k)
{
```

```
    printf("从第%d位置起删除%d个元素后串t1为",pos,len);
    StrPrint(t1);
  }
  DestroyString(t1); // 销毁操作同清空
}
```

程序运行结果：

```
初始化串t1后，串t1空否？1(1:空 0:否) 串长=0
出错
出错
串t1为ABCDEFGHI
串t1空否？0(1:空 0:否) 串长=9
串t2为12345
由串t1拷贝得到串t3,串t3为ABCDEFGHI
串t4为ABCD
用t2取代串t3中的t4串后，串t3为12345EFGHI
清空串t1后，串t1空否？1(1:空 0:否) 串长=0
串t1(=t2+t3)为12345123456789EFGHI
pos=6
在串t1的第pos个字符之前插入串t2，请输入pos：1↙
插入串t2后，串t1为123451234512345EFGHI
求从t1的第pos个字符起,长度为len的子串t2,请输入pos,len：3,2↙
串t2为34
StrCompare(t1,t2)=-2
删除串t1中的子字符串：从第pos个字符起删除len个字符。请输入pos,len：6,15↙
从第6位置起删除15个元素后串t1为12345
```

由 bo4-3.cpp 可见，串的块链存储结构优势很少，一般不用。

4.3　串的模式匹配算法

4.3.1　求子串位置的定位函数 Index(S, T, pos)

串的模式匹配的一般方法如算法 4.5(在 bo4-1.cpp 中)所示：由主串 S 的第 pos 个字符起，检验是否存在子串 T。首先令 i 等于 pos(i 为 S 中当前待比较字符的位序)，j 等于 1(j 为 T 中当前待比较字符的位序)，如果 S 的第 i 个字符与 T 的第 j 个字符相同，则 i、j 各加 1 继续比较，直至 T 的最后一个字符(找到)。如果还没到 T 的最后一个字符，比较就出现了不同(没找到)，则令 i 等于 pos+1，j 等于 1，由 pos 的下一个位置起，继续查找是否存在子串 T。这个过程如图 4-10 所示。

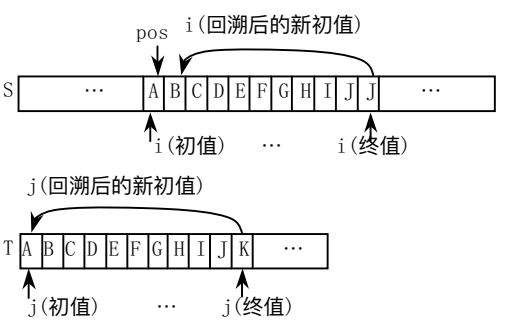


图 4-10　模式匹配的一般方法

4.3.2　模式匹配的一种改进算法

在算法 4.5 中，主串 S 的指针 i 总要回溯，特别是在如图 4－10 所示的有较多字符匹配而又不完全匹配的情况下，回溯得更多。这时，主串 S 的一个字符要进行多次比较，显然效率较低。

如果能使主串 S 的指针 i 不回溯，在有些情况下效率则会大为提高。这是可以做到的，因为主串 S 中位于 i-1，i-2，… 的字符恰和子串 T 中位于 j-1，j-2，… 的字符相等，如图 4－10 所示。仍以图 4－10 为例，当 S 和 T 在第 i(终值)个字符处字符不相符时，i 仍保持在终值处不动，j 回溯到第 1 个字符与 i 的当前字符继续进行比较。j 回溯到第几个字符是由子串 T 的模式决定的。算法 4.7 根据子串 T 生成的 next 数组指示 j 回溯到第几个字符。next 数组的意义是这样的：如果 next[j]=k，当子串 T 的第 j 个字符与主串 S 的第 i 个字符“失配”时，S 的第 i 个字符继续与 T 的第 k 个字符进行比较，T 的第 k 个字符之前的那些字符均与 S 的第 i 个字符之前的字符匹配。以教科书中图 4.5 为例，设子串 T 为“abaabcac”。当 T 的第 5 个字符与 S 的第 i 个字符失配时，S 的第 i-1 个字符一定是 a，和 T 的第 4 个字符相等。它和 T 的第 1 个字符相等。这样，S 的第 i 个字符和 T 的第 2 个字符开始比较即可。所以，对于模式串“abaabcac”，next[5]=2，详见图 4－11。

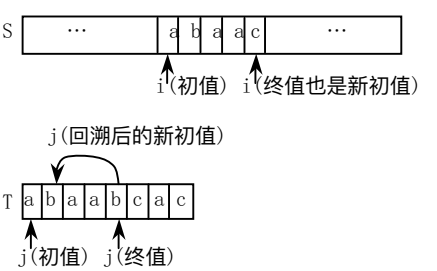


图 4－11　串"abaabcac"的数组 next[5]=2 的由来

算法 4.7 求子串的数组 next[]还有可改进之处。以图 4－11 为例：如果 T 的第 5 个字符与 S 的第 i 个字符失配，则 S 的第 i 个字符一定不是 b。这样，尽管 S 的第 i-1 个字符是 a，和 T 的第 1 个字符相等，但 S 的第 i 个字符肯定和 T 的第 2 个字符 b 不相等。所以可令 next[5]=1，使 S 的第 i 个字符和 T 的第 1 个字符开始比较。这样使得模式串又向右移了一位，提高了匹配的效率。算法 4.8 是改进的求数组 next[](在算法 4.8 中的形参是 nextval[])的算法。

算法 4.6 是改进的模式匹配算法。它利用算法 4.7 或算法 4.8 求得的数组 next[]，提高了算法的效率。algo4-1.cpp 是实现改进的模式匹配算法的程序。函数 get_next()和 get_nextval()分别求得给定的模式串的数组 next[]和 nextval[]，函数 Index_KMP()利用数组 next[]或 nextval[]求出模式串在主串中的位置。其中，next[j]=0，并不是将主串的当前字符与模式串的第 0 个字符进行比较(模式串也没有第 0 个字符)，而是主串当前字符的下一个字符与模式串的第 1 个字符进行比较。

```
// algo4-1.cpp 实现算法4.6、4.7、4.8的程序
#include"c1.h"
#include"c4-1.h"
#include"bo4-1.cpp"
void get_next(SString T,int next[])
{ // 求模式串T的next函数值并存入数组next。算法4.7
  int i=1,j=0;
  next[1]=0;
```

```
  while(i<T[0])
    if(j==0||T[i]==T[j])
    {
      ++i;
      ++j;
      next[i]=j;
    }
    else
      j=next[j];
}
void get_nextval(SString T,int nextval[])
{ // 求模式串T的next函数修正值并存入数组nextval。算法4.8
  int i=1,j=0;
  nextval[1]=0;
  while(i<T[0])
    if(j==0||T[i]==T[j])
    {
      ++i;
      ++j;
      if(T[i]!=T[j])
        nextval[i]=j;
      else
        nextval[i]=nextval[j];
    }
    else
      j=nextval[j];
}
int Index_KMP(SString S,SString T,int pos,int next[])
{ // 利用模式串T的next函数求T在主串S中第pos个字符之后的位置的KMP算法。
  // 其中，T非空，1≤pos≤StrLength(S)。算法4.6
  int i=pos,j=1;
  while(i<=S[0]&&j<=T[0])
    if(j==0||S[i]==T[j]) // 继续比较后继字符
    {
      ++i;
      ++j;
    }
    else // 模式串向右移动
      j=next[j];
  if(j>T[0]) // 匹配成功
    return i-T[0];
  else
    return 0;
}
void main()
{
  int i,*p;
  SString s1,s2; // 以教科书算法4.8之上的数据为例
  StrAssign(s1,"aaabaaaab");
  printf("主串为");
  StrPrint(s1);
```

```
  StrAssign(s2,"aaaab");
  printf("子串为");
  StrPrint(s2);
  p=(int*)malloc((StrLength(s2)+1)*sizeof(int)); // 生成s2的next数组空间
  get_next(s2,p); // 利用算法4.7，求得next数组，存于p中
  printf("子串的next数组为");
  for(i=1;i<=StrLength(s2);i++)
    printf("%d ",*(p+i));
  printf("\n");
  i=Index_KMP(s1,s2,1,p); // 利用算法4.6求得串s2在s1中首次匹配的位置i
  if(i)
    printf("主串和子串在第%d个字符处首次匹配\n",i);
  else
    printf("主串和子串匹配不成功\n");
  get_nextval(s2,p); // 利用算法4.8，求得next数组，存于p中
  printf("子串的nextval数组为");
  for(i=1;i<=StrLength(s2);i++)
    printf("%d ",*(p+i));
  printf("\n");
  printf("主串和子串在第%d个字符处首次匹配\n",Index_KMP(s1,s2,1,p));
}
```

程序运行结果：

```
主串为aaabaaaab
子串为aaaab
子串的next数组为0 1 2 3 4
主串和子串在第5个字符处首次匹配
子串的nextval数组为0 0 0 0 4
主串和子串在第5个字符处首次匹配
```

4.4　串操作应用举例

4.4.1　文本编辑

```
// algo4-2.cpp 文本行编辑
#include"c1.h"
#include"c4-2.h" // 采用串的堆分配存储结构
#include"bo4-2.cpp" // 串的堆分配基本操作
#define MAX_LEN 25 // 文件最大行数
#define LINE_LEN 75 // 每行字符数最大值+1
#define NAME_LEN 20 // 文件名最大长度(包括盘符、路径)+1
// 全局变量(见图4－12)
HString T[MAX_LEN];
char str[LINE_LEN],filename[NAME_LEN]="";
FILE *fp;
```

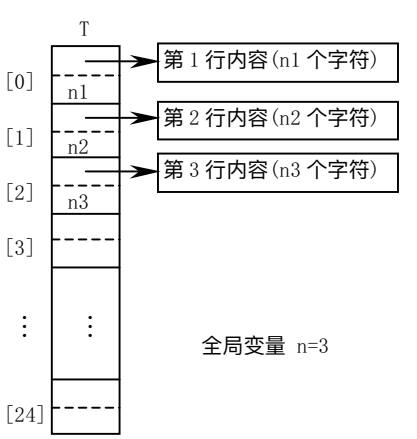


图 4－12　一个 3 行文本存储结构示例

```
int n=0; // 文本行数
void Open()
{ // 打开文件(新或旧)
  if(filename[0]) // 文件已打开
    printf("已存在打开的文件\n");
  else
  {
    printf("请输入文件名(可包括盘符、路径, 不超过%d个字符): ",NAME_LEN-1);
    scanf("%s",filename);
    fp=fopen(filename,"r"); // 以读的方式打开文件
    if(fp) // 已存在此文件
    {
      while(fgets(str,LINE_LEN,fp)) // 由文件读入1行字符成功
      {
        str[strlen(str)-1]=0; // 将换行符10强制改为串结束符0
        if(n>=MAX_LEN)
        {
          printf("文件太大\n");
          return;
        }
        StrAssign(T[n],str);
        n++;
      }
      fclose(fp); // 关闭文件
    }
    else
      printf("新文件\n");
  }
}
void List()
{ // 显示文本内容
  int i;
  for(i=0;i<n;i++)
  {
    printf("%d: ",i+1);
    StrPrint(T[i]);
  }
}
void Insert()
{ // 插入行
  int i,l,m;
  printf("在第l行前插m行,请输入l m: ");
  scanf("%d%d%*c",&l,&m);
  if(n+m>MAX_LEN)
  {
    printf("插入行太多\n");
    return;
  }
  if(n>=l-1&&l>0)
  {
```

```
    for(i=n-1;i>=l-1;i--)
      T[i+m]=T[i];
    n+=m;
    printf("请顺序输入待插入内容:\n");
    for(i=l-1;i<l-1+m;i++)
    {
      gets(str);
      InitString(T[i]);
      StrAssign(T[i],str);
    }
  }
  else
    printf("行超出范围\n");
}
void Delete()
{ // 删除行
  int i,l,m;
  printf("从第l行起删除m行,请输入l m: ");
  scanf("%d%d",&l,&m);
  if(n>=l+m-1&&l>0)
  {
    for(i=l-1+m;i<n;i++)
    {
      free(T[i-m].ch);
      T[i-m]=T[i];
    }
    for(i=n-m;i<n;i++)
      InitString(T[i]);
    n-=m;
  }
  else
    printf("行超出范围\n");
}
void Copy()
{ // 拷贝行
  int i,l,m,k;
  printf("把第l行开始的m行插在原k行之前,请输入l m k: ");
  scanf("%d%d%d",&l,&m,&k);
  if(n+m>MAX_LEN)
  {
    printf("拷贝行太多\n");
    return;
  }
  if(n>=k-1&&n>=l-1+m&&(k>=l+m||k<=l))
  {
    for(i=n-1;i>=k-1;i--)
      T[i+m]=T[i];
    n+=m;
    if(k<=l)
      l+=m;
```

```
    for(i=l-1;i<l-1+m;i++)
    {
      InitString(T[i+k-l]);
      StrCopy(T[i+k-l],T[i]);
    }
  }
  else
    printf("行超出范围\n");
}
void Modify()
{ // 修改行
  int i;
  printf("请输入待修改的行号: ");
  scanf("%d%*c",&i);
  if(i>0&&i<=n) // 行号合法
  {
    printf("%d: ",i);
    StrPrint(T[i-1]);
    printf("请输入新内容: ");
    gets(str);
    StrAssign(T[i-1],str);
  }
  else
    printf("行号超出范围\n");
}
void Search()
{ // 查找字符串
  int i,k,f=1; // f为继续查找标志
  char b[2];
  HString s;
  printf("请输入待查找的字符串: ");
  scanf("%s%*c",str);
  InitString(s);
  StrAssign(s,str);
  for(i=0;i<n&&f;i++) // 逐行查找
  {
    k=1; // 由每行第1个字符起查找
    while(k)
    {
      k=Index(T[i],s,k); // 由本行的第k个字符开始查找
      if(k) // 找到
      {
        printf("第%d行: ",i+1);
        StrPrint(T[i]);
        printf("第%d个字符处找到。继续查找吗(Y/N)? ",k);
        gets(b);
        if(b[0]!='Y'&&b[0]!='y') // 中断查找
        {
          f=0;
          break;
```

```
        }
        else
          k++;
      }
    }
  }
  if(f)
    printf("没找到\n");
}
void Replace()
{ // 替换字符串
  int i,k,f=1; // f为继续替换标志
  char b[2];
  HString s,t;
  printf("请输入待替换的字符串: ");
  scanf("%s%*c",str);
  InitString(s);
  StrAssign(s,str);
  printf("替换为");
  scanf("%s%*c",str);
  InitString(t);
  StrAssign(t,str);
  for(i=0;i<n&&f;i++) // 逐行查找、替换
  {
    k=1; // 由每行第1个字符起查找
    while(k)
    {
      k=Index(T[i],s,k); // 由本行的第k个字符开始查找
      if(k) // 找到
      {
        printf("第%d行: ",i+1);
        StrPrint(T[i]);
        printf("第%d个字符处找到。是否替换(Y/N)? ",k);
        gets(b);
        if(b[0]=='Y'||b[0]=='y')
        {
          StrDelete(T[i],k,StrLength(s));
          StrInsert(T[i],k,t);
        }
        printf("继续替换吗(Y/N)?");
        gets(b);
        if(b[0]!='Y' &&b[0]!='y') // 中断查找、替换
        {
          f=0;
          break;
        }
        else
          k+=StrLength(t);
      }
    }
```

```
  }
  if(f)
    printf("没找到\n");
}
void Save()
{ // 存盘
  int i,j;
  fp=fopen(filename,"w"); // 以写的方式重新打开文件
  if(fp) // 打开文件成功
  {
    for(i=0;i<n;i++)
    { // 依次将每行存入文件
      for(j=0;j<T[i].length;j++) // 依次存入每个字符
        fputc(T[i].ch[j],fp);
      fputc(10,fp); // 存入换行符
      ClearString(T[i]); // 释放串空间
    }
    fclose(fp); // 关闭文件
  }
  else
    printf("存盘失败\n");
}
void main()
{
  Status s=TRUE;
  int i,k;
  for(i=0;i<MAX_LEN;i++) // 初始化串
    InitString(T[i]);
  while(s)
  {
    printf("请选择: 1.打开文件(新或旧)  2.显示文件内容\n");
    printf("        3.插入行  4.删除行  5.拷贝行  6.修改行\n");
    printf("        7.查找字符串        8.替换字符串\n");
    printf("        9.存盘退出          0.放弃编辑\n");
    scanf("%d",&k);
    switch(k)
    {
      case 1: Open();
              break;
      case 2: List();
              break;
      case 3: Insert();
              break;
      case 4: Delete();
              break;
      case 5: Copy();
              break;
      case 6: Modify();
              break;
      case 7: Search();
```

```
            break;
      case 8: Replace();
            break;
      case 9: Save();
      case 0: s=FALSE;
    }
  }
}
```

文件 file.txt 的内容：

```
11111
22222
33333
44444
```

程序运行结果：

```
请选择: 1.打开文件(新或旧)  2.显示文件内容
        3.插入行  4.删除行  5.拷贝行  6.修改行
        7.查找字符串        8.替换字符串
        9.存盘退出          0.放弃编辑
1↙
请输入文件名(可包括盘符、路径，不超过19个字符): file.txt↙
请选择: 1.打开文件(新或旧)  2.显示文件内容
        3.插入行  4.删除行  5.拷贝行  6.修改行
        7.查找字符串        8.替换字符串
        9.存盘退出          0.放弃编辑
2↙
1: 11111
2: 22222
3: 33333
4: 44444
请选择: 1.打开文件(新或旧)  2.显示文件内容
        3.插入行  4.删除行  5.拷贝行  6.修改行
        7.查找字符串        8.替换字符串
        9.存盘退出          0.放弃编辑
9↙
```

4.4.2　建立词索引表

algo4-3.cpp 是以教科书图 4.9 为例建立书名关键词索引表的程序。教科书图 4.9(a)所示的书目文件存于 BookInfo.txt 中，其内容如下：

```
005 Computer Data Structures
010 Introduction to Data Structures
023 Fundamentals of Data Structures
034 The Design and Analysis of Computer Algorithms
050 Introduction to Numerical Analysis
067 Numerical Analysis
```

程序首先将存于 BookInfo.txt 文件中的每种书的书号及书名中的关键词提取出来。书号存储于整型变量 BookNo 中，书名中的每个关键词先临时存储于全局变量 wdlist 中，wdlist 的结构如图 4－13 所示。

在提取书名的关键词时，要注意剔除 the、a、of 等非索引词。这些词在书名中一般是小写，对于首字母是小写的单词不作处理即可。但这些非索引词若是书名的第 1 个单词，则首字母也为大写。为了解决这个问题，将非索引词存入文件 NoIdx.txt 中，内容如下：

```
5
A
An
In
Of
The
```

NoIdx.txt 文件的内容可通过文字编辑软件自行输入。第 1 行为非索引词个数，单词按字典顺序排列。程序将文件 NoIdx.txt 的内容读入全局变量 noidx 中。noidx 的结构和 wdlist 相同，其内容如图 4－14 所示。对于书名中的每个单词，首先检索是否出现在 noidx 中，如果是，则不将该单词插入 wdlist 中作为关键词。图 4－13 在显示 wdlist 结构的同时也显示了 wdlist 存储书目文件第 4 本书(书号为 34)时的内容。

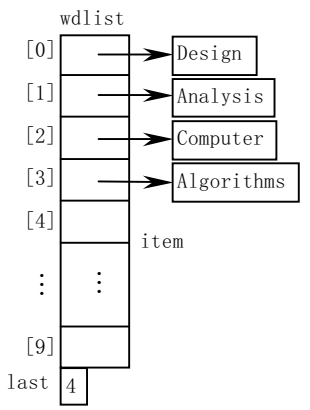


图 4－13　wdlist 的结构示例

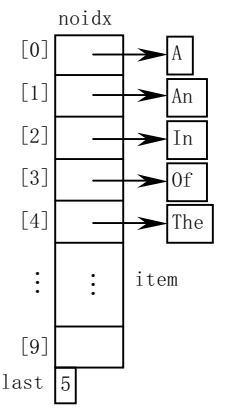


图 4－14　noidx 的结构

程序依次根据临时存储于 wdlist 中的每个书名的关键词，产生和充实关键词索引表。方法是：首先建一个空的关键词索引表变量 idxlist，然后依次查询 wdlist 中的各个关键词是否存在于 idxlist 中。如果已存在，则仅把该关键词的书号按升序插入相应的链表中。否

则先按字母顺序将 wdlist 中的关键词插入 idxlist，再插入书号。idxlist 的结构及内容如图 4－15 所示。

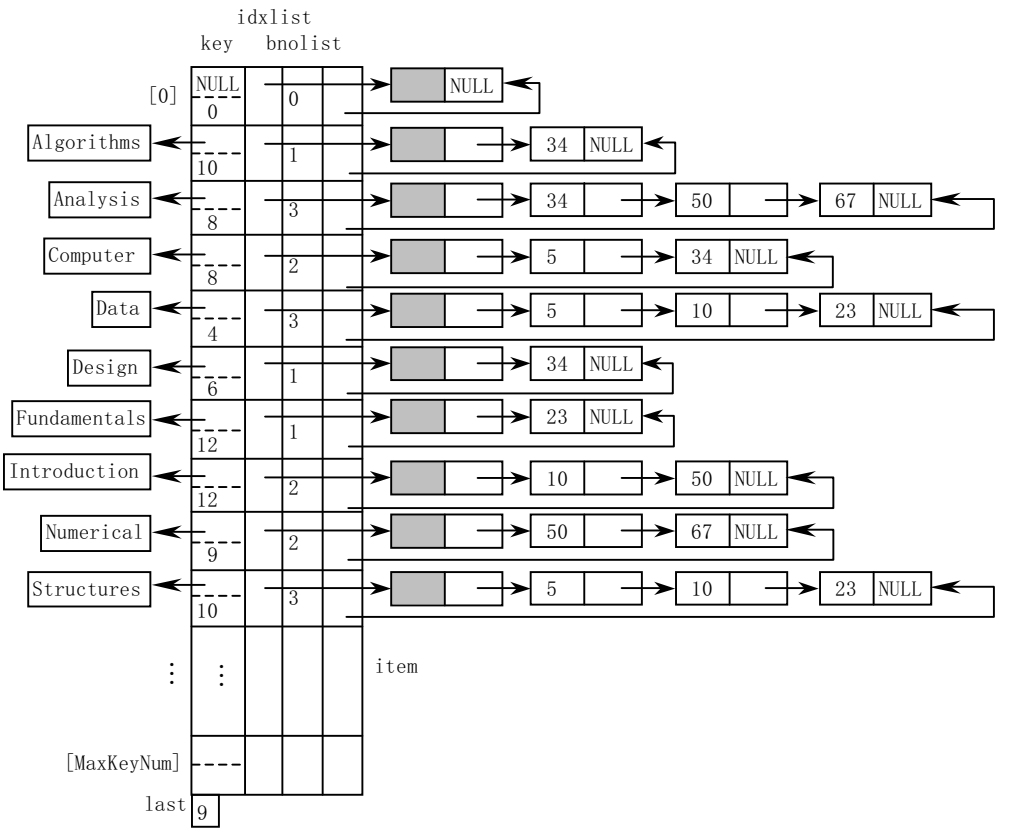


图 4－15　algo4-3.cpp 建立的书名关键词索引表 idxlist 的结构

书名关键词索引表仅在增加新书时有变化，因此不必在每次检索书名时都建立一次。可以把 algo4-3.cpp 生成的书名关键词索引表(idxlist)存成文件(BookIdx.txt)，由索引查询图书的程序 algo4-4.cpp 调用。algo4-3.cpp 程序运行的结果就是生成 BookIdx.txt 文件，其内容如下：

```
9
Algorithms
1
34
Analysis
3
34 50 67
Computer
2
5 34
Data
3
5 10 23
```

```
Design
1
34
Fundamentals
1
23
Introduction
2
10 50
Numerical
2
50 67
Structures
3
5 10 23
```

```
// algo4-3.cpp 根据书目文件bookinfo.txt生成书名关键词索引文件bookidx.txt,
// 为运行algo4-4.cpp做准备, 算法4.9～4.14
#include"c1.h"
typedef int ElemType;
#include"c2-5.h"
#include"bo2-6.cpp"
#include"c4-2.h"
#include"bo4-2.cpp"
#define MaxKeyNum 25 // 索引表的最大容量(关键词的最大数目)
#define MaxLineLen 52 // 书目串(书目文件的1行)buf的最大长度
#define MaxNoIdx 10 // 非索引词(也是一个书目中关键词)的最大数目
struct WordListType // 一个书目的词表(顺序表)和非索引词表(有序表)共用类型
{
  char *item[MaxNoIdx]; // 词表(字符串)指针数组
  int last; // 词的数量
};
struct IdxTermType // 索引项类型
{
  HString key; // 关键词(堆分配类型, c4-2.h)
  LinkList bnolist; // 存放书号索引的链表(c2-5.h)
};
struct IdxListType // 索引表类型(有序表)
{
  IdxTermType item[MaxKeyNum+1]; // 索引项数组类型
  int last; // 关键词的个数
};
// 全局变量
char buf[MaxLineLen+1]; // 当前书目串(包括'\0')
WordListType wdlist,noidx; // 暂存一种书的词表, 非索引词表
void InitIdxList(IdxListType &idxlist)
{ // 初始化操作, 置索引表idxlist为空表, 且在idxlist.item[0]设一空串
  idxlist.last=0;
  InitString(idxlist.item[0].key); // 初始化[0]单元, 函数在bo4-2.cpp中
  InitList(idxlist.item[0].bnolist); // 初始化[0]单元, 函数在bo2-6.cpp中
}
```

```
void ExtractKeyWord(int &BookNo)
{ // 从buf中提取书名关键词到词表wdlist，书号存入BookNo
  int i,l,f=1; // f是字符串buf结束标志 0：结束 1：未结束
  char *s1,*s2;
  for(i=1;i<=wdlist.last;i++)
  { // 释放上一个书目在词表wdlist的存储空间
    free(wdlist.item[i]);
    wdlist.item[i]=NULL;
  }
  wdlist.last=0; // 初始化词表wdlist的词数量
  BookNo=atoi(buf); // 将前几位数字转化为整数，赋给书号BookNo
  s1=&buf[4]; // s1指向书名的首字符
  while(f)
  { // 提取书名关键词到词表wdlist
    s2=strchr(s1,' '); // s2指向s1后的第一个空格，如没有，返回NULL
    if(!s2) // 到串尾(没空格)
    {
      s2=strchr(s1,'\12'); // s2指向buf的最后一个字符(回车符10)
      f=0;
    }
    l=s2-s1; // 单词长度
    if(s1[0]>='A'&&s1[0]<='Z') // 单词首字母为大写
    { // 写入词表
      wdlist.item[wdlist.last]=(char *)malloc((l+1)*sizeof(char)); // 生成串空间(包括'\0')
      for(i=0;i<l;i++)
        wdlist.item[wdlist.last][i]=s1[i]; // 写入词表
      wdlist.item[wdlist.last][l]=0; // 串结束符
      for(i=0;i<noidx.last&&(l=strcmp(wdlist.item[wdlist.last],noidx.item[i]))>0;i++);
      // 查找是否为非索引词
      if(!l) // 是非索引词
      {
        free(wdlist.item[wdlist.last]); // 从词表中删除该词
        wdlist.item[wdlist.last]=NULL;
      }
      else
        wdlist.last++; // 词表长度+1
    }
    s1=s2+1; // s1移动到下一个单词的首字符处
  };
}
void GetWord(int i,HString &wd)
{ // 用wd返回词表wdlist中第i个关键词，算法4.11
  StrAssign(wd,wdlist.item[i]); // 生成关键字字符串 bo4-2.cpp
}
int Locate(IdxListType &idxlist,HString wd,Status &b)
{ // 在索引表idxlist中查询是否存在与wd相等的关键词。若存在，则返回其
  // 在索引表中的位置，且b取值TRUE；否则返回插入位置，且b取值FALSE，算法4.12
  int i,m;
  for(i=idxlist.last;(m=StrCompare(idxlist.item[i].key,wd))>0;--i); // bo4-2.cpp
  if(m==0) // 找到
  {
```

```
    b=TRUE;
    return i;
  }
  else
  {
    b=FALSE;
    return i+1;
  }
}
void InsertNewKey(IdxListType &idxlist,int i,HString wd)
{ // 在索引表idxlist的第i项上插入新关键词wd，并初始化书号索引的链表为空表，算法4.13
  int j;
  for(j=idxlist.last;j>=i;--j) // 后移索引项
    idxlist.item[j+1]=idxlist.item[j];
  InitString(idxlist.item[i].key); // bo4-2.cpp
  StrCopy(idxlist.item[i].key,wd); // 串拷贝插入新的索引项 bo4-2.cpp
  InitList(idxlist.item[i].bnolist); // 初始化书号索引表为空表 bo2-6.cpp
  idxlist.last++;
}
void InsertBook(IdxListType &idxlist,int i,int bno)
{ // 在索引表idxlist的第i项中插入书号为bno的索引，算法4.14
  Link p;
  MakeNode(p,bno); // 分配结点 bo2-6.cpp
  p->next=NULL;
  Append(idxlist.item[i].bnolist,p); // 插入新的书号索引 bo2-6.cpp
}
void InsIdxList(IdxListType &idxlist,int bno)
{ // 将书号为bno的关键词插入索引表，算法4.10
  int i,j;
  Status b;
  HString wd;
  InitString(wd); // bo4-2.cpp
  for(i=0;i<wdlist.last;i++)
  {
    GetWord(i,wd);
    j=Locate(idxlist,wd,b); // 关键词的位置或待插入的位置(当索引表中不存在该词)
    if(!b) // 索引表中不存在关键词wd
      InsertNewKey(idxlist,j,wd); // 在索引表中插入新的索引项
    InsertBook(idxlist,j,bno); // 插入书号索引
  }
}
void PutText(FILE *f,IdxListType idxlist)
{ // 将生成的索引表idxlist输出到文件f
  int i,j;
  Link p;
  fprintf(f,"%d\n",idxlist.last);
  for(i=1;i<=idxlist.last;i++)
  {
    for(j=0;j<idxlist.item[i].key.length;j++)
      fprintf(f,"%c",idxlist.item[i].key.ch[j]); // HString类型串尾没有\0，只能逐个字符输出
    fprintf(f,"\n%d\n",idxlist.item[i].bnolist.len);
```

```
    p=idxlist.item[i].bnolist.head;
    for(j=1;j<=idxlist.item[i].bnolist.len;j++)
    {
      p=p->next;
      fprintf(f,"%d ",p->data);
    }
    fprintf(f,"\n");
  }
}
void main()
{ // 算法4.9
  FILE *f; // 任何时间最多打开一个文件
  IdxListType idxlist; // 索引表
  int BookNo; // 书号变量
  int k;
  if(!(f=fopen("NoIdx.txt","r"))) // 打开非索引词文件
    exit(OVERFLOW);
  fscanf(f,"%d",&noidx.last); // 读取非索引词个数
  for(k=0;k<noidx.last;k++) // 把非索引词文件的内容依次拷到noidx中
  {
    fscanf(f,"%s",buf);
    noidx.item[k]=(char*)malloc((strlen(buf)+1)*sizeof(char));
    strcpy(noidx.item[k],buf);
  }
  fclose(f); // 关闭非索引词文件
  if(!(f=fopen("BookInfo.txt","r"))) // 打开书目文件
    exit(FALSE);
  InitIdxList(idxlist); // 设索引表idxlist为空，并初始化[0]单元
  while(fgets(buf,MaxLineLen,f)) // 由书目文件读取1行信息到buf成功
  {
    ExtractKeyWord(BookNo);//将buf中的书号存入BookNo,关键词提取到词表(当前书目的关键词表)中
    InsIdxList(idxlist,BookNo); // 将书号为BookNo的关键词及书号插入索引表idxlist中
  }
  fclose(f); // 关闭书目文件
  if(!(f=fopen("BookIdx.txt","w"))) // 打开书名关键词索引文件
    exit(INFEASIBLE);
  PutText(f,idxlist); // 将生成的索引表idxlist输出到书名关键词索引文件
  fclose(f); // 关闭书名关键词索引文件
}
```

程序运行结果：生成 BookIdx.txt 文件。

algo4-4.cpp 根据 algo4-3.cpp 产生的文件 BookIdx.txt 索引查询图书。首先将存有书名关键词索引表信息的文件 BookIdx.txt 的内容恢复到变量 idxlist(见图 4－15)中。为方便查询，将关键词一律存为小写。再将书目文件 BookInfo.txt 的内容存入到变量 booklist(见图 4－16)中。程序运行时，从键盘输入一个书名关键词，并将此单词转为小写，在 idxlist 中查找。如果找到，则根据相应的链表中所存储的书号，依次查询 booklist 并输出相应的书名。

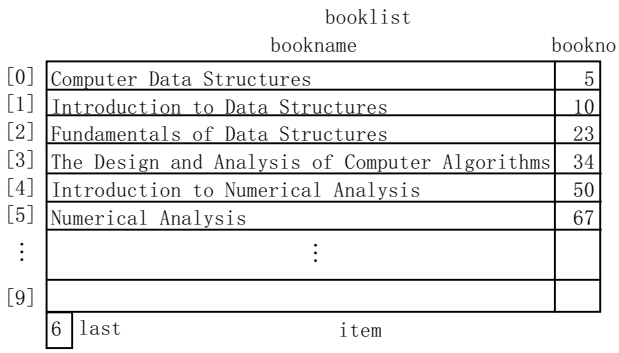


图 4 - 16　booklist 的结构

```
// algo4-4.cpp 根据algo4-3.cpp产生的文件,索引查询图书
#include"c1.h"
typedef int ElemType;
#include"c2-5.h"
#include"bo2-6.cpp"
#include"c4-2.h"
#include"bo4-2.cpp"
#define MaxBookNum 10
// 假设只对10个书名建索引表
#define MaxKeyNum 25
// 索引表的最大容量(关键词的最大数目)
#define MaxLineLen 46 // 书名的最大长度
struct IdxTermType // 索引项类型
{
  HString key; // 关键词(堆分配类型, c4-2.h)
  LinkList bnolist; // 存放书号索引的链表(c2-5.h)
};
struct IdxListType // 索引表类型(有序表)
{
  IdxTermType item[MaxKeyNum+1]; // 索引项数组类型
  int last; // 关键词的个数
};
struct BookTermType // 书目项类型
{
  char bookname[MaxLineLen+1]; // 书名串(包括'\0')
  int bookno; // 书号
};
struct BookListType // 书目表类型(有序表)
{
  BookTermType item[MaxBookNum]; // 书目项数组类型
  int last; // 书目的数量
};
void main()
{
  FILE *f; // 任何时间最多打开一个文件
  IdxListType idxlist; // 索引表
  BookListType booklist; // 书目表
  char buf[MaxLineLen+5]; // 当前书目串(包括书号和'\0')
```

```
HString ch; // 索引字符串
int BookNo; // 书号
Link p; // 链表指针
int i,j,k,flag=1; // flag是继续查询的标志
InitString(ch); // 初始化HString类型的变量
if(!(f=fopen("BookIdx.txt","r"))) // 打开书名关键词索引表文件
  exit(OVERFLOW);
fscanf(f,"%d",&idxlist.last); // 书名关键词个数
for(k=0;k<idxlist.last;k++) // 把关键词文件的内容拷到idxlist中
{
  fscanf(f,"%s",buf);
  i=0;
  while(buf[i])
    buf[i++]=tolower(buf[i]); // 字母转为小写
  InitString(idxlist.item[k].key);
  StrAssign(idxlist.item[k].key,buf);
  InitList(idxlist.item[k].bnolist); // 初始化书号链表, bo2-6.cpp
  fscanf(f,"%d",&i);
  for(j=0;j<i;j++)
  {
    fscanf(f,"%d",&BookNo);
    MakeNode(p,BookNo); // 产生新的书号结点, bo2-6.cpp
    p->next=NULL; // 给书号结点的指针域赋值
    Append(idxlist.item[k].bnolist,p); // 在表尾插入新的书号结点, bo2-6.cpp
  }
}
fclose(f);
if(!(f=fopen("BookInfo.txt","r"))) // 打开书目文件
  exit(FALSE);
i=0;
while(fgets(buf,MaxLineLen,f))
{ // 把书目文件的内容拷到booklist中
  booklist.item[i].bookno=atoi(buf); // 前几位数字为书号
  strcpy(booklist.item[i++].bookname,&buf[4]); // 将buf由书名开始的字符串拷贝到booklist中
}
booklist.last=i;
while(flag)
{
  printf("请输入书目的关键词(一个): ");
  scanf("%s",buf);
  i=0;
  while(buf[i])
    buf[i++]=tolower(buf[i]); // 字母转为小写
  StrAssign(ch,buf);
  i=0;
  do
  {
    k=StrCompare(ch,idxlist.item[i++].key); // bo4-2.cpp
  }while(k&&i<=idxlist.last);
  if(!k) // 索引表中有此关键词
  {
```

```
      p=idxlist.item[--i].bnolist.head->next; // p指向索引表中此关键词相应链表的首元结点
      while(p)
      {
        j=0;
        while(j<booklist.last&&p->data!=booklist.item[j].bookno) //在booklist中找相应的书号
          j++;
        if(j<booklist.last)
          printf("%3d %s",booklist.item[j].bookno,booklist.item[j].bookname);
        p=p->next; // 继续向后找
      }
    }
    else
      printf("没找到\n");
    printf("继续查找请输入1，退出查找请输入0：");
    scanf("%d",&flag);
  }
}
```

程序运行结果：

```
请输入书目的关键词(一个)：DATA↙
  5 Computer Data Structures
 10 Introduction to Data Structures
 23 Fundamentals of Data Structures
继续查找请输入1，退出查找请输入0：1↙
请输入书目的关键词(一个)：structure↙
没找到
继续查找请输入1，退出查找请输入0：0↙
```

第 5 章　数组和广义表

5.1　数组的定义

5.2　数组的顺序表示和实现

```
// c5-1.h 数组的顺序存储表示(见图5－1)
#include<stdarg.h> // 标准头文件，提供宏va_start，va_arg和va_end，用于存取变长参数表
#define MAX_ARRAY_DIM 8 // 假设数组维数的最大值为8
struct Array
{
  ElemType *base; // 数组元素基址，由InitArray分配
  int dim; // 数组维数
  int *bounds; // 数组维界基址，由InitArray分配
  int *constants; // 数组映象函数常量基址，由InitArray分配
};
```

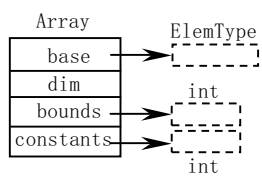


图 5－1　数组的顺序存储结构

```
// bo5-1.cpp 顺序存储数组(存储结构由c5-1.h定义)的基本操作(5个)
Status InitArray(Array &A,int dim,...)
{ // 若维数dim和各维长度合法，则构造相应的数组A，并返回OK(见图5－2)
  int elemtotal=1,i; // elemtotal是数组元素总数，初值为1(累乘器)
  va_list ap;
  if(dim<1||dim>MAX_ARRAY_DIM)
    return ERROR;
  A.dim=dim;
  A.bounds=(int *)malloc(dim*sizeof(int));
  if(!A.bounds)
    exit(OVERFLOW);
  va_start(ap,dim);
  for(i=0;i<dim;++i)
  {
    A.bounds[i]=va_arg(ap,int);
    if(A.bounds[i]<0)
      return UNDERFLOW; // 在math.h中定义为4
    elemtotal*=A.bounds[i];
  }
  va_end(ap);
  A.base=(ElemType *)malloc(elemtotal*sizeof(ElemType));
  if(!A.base)
```

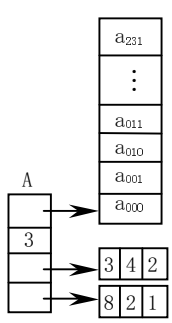


图 5－2　A[3][4][2]数组的存储结构

```
    exit(OVERFLOW);
  A.constants=(int *)malloc(dim*sizeof(int));
  if(!A.constants)
    exit(OVERFLOW);
  A.constants[dim-1]=1;
  for(i=dim-2;i>=0;--i)
    A.constants[i]=A.bounds[i+1]*A.constants[i+1];
  return OK;
}
void DestroyArray(Array &A)
{ // 销毁数组A(见图5-3)
  if(A.base)
    free(A.base);
  if(A.bounds)
    free(A.bounds);
  if(A.constants)
    free(A.constants);
  A.base= A.bounds=A.constants=NULL;
  A.dim=0;
}
Status Locate(Array A,va_list ap,int &off) // Value()、Assign()调用此函数
{ // 若ap指示的各下标值合法，则求出该元素在A中的相对地址off
  int i,ind;
  off=0;
  for(i=0;i<A.dim;i++)
  {
    ind=va_arg(ap,int);
    if(ind<0||ind>=A.bounds[i])
      return OVERFLOW;
    off+=A.constants[i]*ind;
  }
  return OK;
}
Status Value(ElemType &e,Array A,...) // 在VC++中，...之前的形参不能是引用类型
{ // ...依次为各维的下标值，若各下标合法，则e被赋值为A的相应的元素值
  va_list ap;
  int off;
  va_start(ap,A);
  if(Locate(A,ap,off)==OVERFLOW) // 调用Locate()
    return ERROR;
  e=*(A.base+off);
  return OK;
}
Status Assign(Array A,ElemType e,...) // 变量A的值不变，故不需要&
{ // ...依次为各维的下标值，若各下标合法，则将e的值赋给A的指定的元素
  va_list ap;
  int off;
  va_start(ap,e);
  if(Locate(A,ap,off)==OVERFLOW) // 调用Locate()
    return ERROR;
```

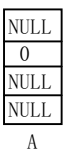


图 5-3　销毁数组 A

```
  *(A.base+off)=e;
  return OK;
}

// main5-1.cpp 检验bo5-1.cpp的主程序
#include"c1.h"
typedef int ElemType;
#include"c5-1.h"
#include"bo5-1.cpp"
void main()
{
  Array A;
  int i,j,k,*p,dim=3,bound1=3,bound2=4,bound3=2; // A[3][4][2]数组
  ElemType e,*p1;
  InitArray(A,dim,bound1,bound2,bound3); // 构造3× 4× 2的3维数组A(见图5－2)
  p=A.bounds;
  printf("A.bounds=");
  for(i=0;i<dim;i++) // 顺序输出A.bounds
    printf("%d ",*(p+i));
  p=A.constants;
  printf("\nA.constants=");
  for(i=0;i<dim;i++) // 顺序输出A.constants
    printf("%d ",*(p+i));
  printf("\n%d页%d行%d列矩阵元素如下:\n",bound1,bound2,bound3);
  for(i=0;i<bound1;i++)
  {
    for(j=0;j<bound2;j++)
    {
      for(k=0;k<bound3;k++)
      {
        Assign(A,i*100+j*10+k,i,j,k); // 将i× 100+j× 10+k赋值给A[i][j][k]
        Value(e,A,i,j,k); // 将A[i][j][k]的值赋给e
        printf("A[%d][%d][%d]=%2d ",i,j,k,e); // 输出A[i][j][k]
      }
      printf("\n");
    }
    printf("\n");
  }
  p1=A.base;
  printf("A.base=\n");
  for(i=0;i<bound1*bound2*bound3;i++) // 顺序输出A.base
  {
    printf("%4d",*(p1+i));
    if(i%(bound2*bound3)==bound2*bound3-1)
      printf("\n");
  }
  printf("A.dim=%d\n",A.dim);
  DestroyArray(A);
}
```

 　程序运行结果：

```
A.bounds=3 4 2
A.constants=8 2 1
3页4行2列矩阵元素如下:
A[0][0][0]= 0 A[0][0][1]= 1
A[0][1][0]=10 A[0][1][1]=11
A[0][2][0]=20 A[0][2][1]=21
A[0][3][0]=30 A[0][3][1]=31

A[1][0][0]=100 A[1][0][1]=101
A[1][1][0]=110 A[1][1][1]=111
A[1][2][0]=120 A[1][2][1]=121
A[1][3][0]=130 A[1][3][1]=131

A[2][0][0]=200 A[2][0][1]=201
A[2][1][0]=210 A[2][1][1]=211
A[2][2][0]=220 A[2][2][1]=221
A[2][3][0]=230 A[2][3][1]=231

A.base=
   0   1  10  11  20  21  30  31
 100 101 110 111 120 121 130 131
 200 201 210 211 220 221 230 231
A.dim=3
```

bo5-1.cpp 中有些函数的形参有“...”，它代表变长参数表，即“...”可用若干个实参取代。这很适合含有维数不定的数组的函数。因为如果是两维数组，参数中要包括两维的长度，两个整型量；而如果是三维数组，则参数中要包括三维的长度，三个整型量。随着所构造的数组的维数不同，参数的个数也不同。这就必须使用变长参数表才能解决参数个数不定的问题。C 语言教材中介绍变长参数表的不多，有兴趣的读者可参阅西安交通大学出版社出版的《精讲多练 C 语言》(冯博琴、刘路放主编）相关章节。algo5-2.cpp 是采用变长参数表的一个实例。

```
// algo5-2.cpp 变长参数表(函数的实参个数可变)编程示例
#include"c1.h"
#include<stdarg.h> // 实现变长参数表要包括的头文件
typedef int ElemType;
ElemType Max(int num,...) // ...表示变长参数表,位于形参表的最后，前面必须至少有一个固定参数
{ // 函数功能：返回num个数中的最大值
  va_list ap; // 定义ap是变长参数表类型(C语言的数据类型)
  int i;
  ElemType m,n;
  if(num<1)
    exit(ERROR);
  va_start(ap,num); // ap指向固定参数num后面的实参表
```

```
  m=va_arg(ap,ElemType);//依次读取ap所指的实参(以逗号为分隔符)作为ElemType类型实参,ap向后移
  for(i=1;i<num;++i)
  {
    n=va_arg(ap,ElemType); // 同上
    if(m<n)
      m=n;
  }
  va_end(ap); // 与va_start()配对，结束对变长参数表的读取，ap不再指向变长参数表
  return m;
}
void main()
{
  printf("1.最大值为%d\n",Max(4,7,9,5,8)); // 在4个数中求最大值，ap最初指向“ 7,9,5,8”
  printf("2.最大值为%d\n",Max(3,17,36,25)); // 在3个数中求最大值，ap最初指向“ 17,36,25”
}
```

程序运行结果：

```
1.最大值为9
2.最大值为36
```

其实，printf()就是含有变长参数表的 C 语言库函数，它的第 1 个形参是字符串常量或字符型指针，第 2 个形参是变长参数表。因此，我们才可以在 1 个 printf()函数中输出任意个变量。

5.3　矩阵的压缩存储

5.3.1　特殊矩阵

5.3.2　稀疏矩阵

```
// c5-2.h 稀疏矩阵的三元组顺序表存储表示(见图5－4)
#define MAX_SIZE 100 // 非零元个数的最大值
struct Triple
{
  int i,j; // 行下标,列下标
  ElemType e; // 非零元素值
};
struct TSMatrix
{
  Triple data[MAX_SIZE+1]; // 非零元三元组表，data[0]未用
  int mu,nu,tu; // 矩阵的行数、列数和非零元个数
};
```

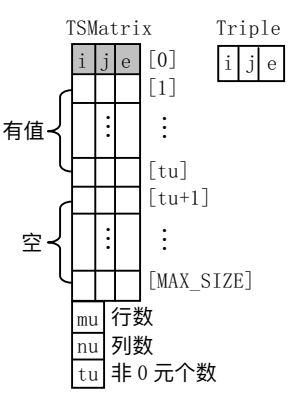
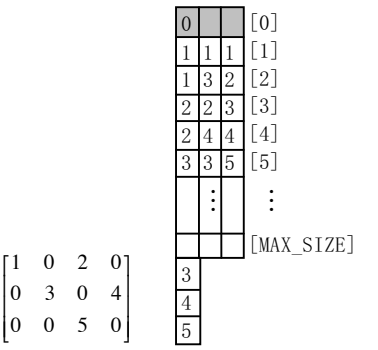


图 5－4　三元组顺序表存储结构

图 5－5 是采用三元组顺序表存储稀疏矩阵的例子。为简化算法，在创建稀疏矩阵输入非零元时，要按行、列的顺序由小到大输入。



(a) 稀疏矩阵　(b) 存储表示

图 5－5　三元组存储稀疏矩阵示例

```
// bo5-2.cpp 三元组稀疏矩阵的基本操作(8个),包括算法5.1
Status CreateSMatrix(TSMatrix &M)
{ // 创建稀疏矩阵M
  int i,m,n;
  ElemType e;
  Status k;
  printf("请输入矩阵的行数,列数,非零元素数: ");
  scanf("%d,%d,%d",&M.mu,&M.nu,&M.tu);
  if(M.tu>MAX_SIZE)
    return ERROR;
  M.data[0].i=0; // 为以下比较顺序做准备
  for(i=1;i<=M.tu;i++)
  {
    do
    {
      printf("请按行序顺序输入第%d个非零元素所在的行(1～%d),列(1～%d),元素值:",i,M.mu,M.nu);
      scanf("%d,%d,%d",&m,&n,&e);
      k=0;
      if(m<1||m>M.mu||n<1||n>M.nu) // 行或列超出范围
        k=1;
      if(m<M.data[i-1].i||m==M.data[i-1].i&&n<=M.data[i-1].j) // 行或列的顺序有错
        k=1;
    }while(k);
    M.data[i].i=m;
    M.data[i].j=n;
    M.data[i].e=e;
  }
 return OK;
}
void DestroySMatrix(TSMatrix &M)
{ // 销毁稀疏矩阵M(见图5－6)
  M.mu=M.nu=M.tu=0;
}
void PrintSMatrix(TSMatrix M)
{ // 输出稀疏矩阵M
  int i;
  printf("%d行%d列%d个非零元素。\n",M.mu,M.nu,M.tu);
  printf("行　列　元素值\n");
  for(i=1;i<=M.tu;i++)
    printf("%2d%4d%8d\n",M.data[i].i,M.data[i].j,M.data[i].e);
}
void PrintSMatrix1(TSMatrix M)
{ // 按矩阵形式输出M
  int i,j,k=1;
  Triple *p=M.data;
  p++; // p指向第1个非零元素
  for(i=1;i<=M.mu;i++)
```

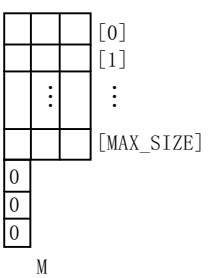


图 5－6　销毁稀疏矩阵 M

```
  {
    for(j=1;j<=M.nu;j++)
      if(k<=M.tu&&p->i==i&&p->j==j) // p指向非零元，且p所指元素为当前处理元素
      {
        printf("%3d",p->e); // 输出p所指元素的值
        p++; // p指向下一个元素
        k++; // 计数器+1
      }
      else // p所指元素不是当前处理元素
        printf("%3d",0); // 输出0
    printf("\n");
  }
}
void CopySMatrix(TSMatrix M,TSMatrix &T)
{ // 由稀疏矩阵M复制得到T
  T=M;
}
int comp(int c1,int c2)
{ // AddSMatrix函数要用到，另加
  if(c1<c2)
    return -1;
  if(c1==c2)
    return 0;
  return 1;
}
Status AddSMatrix(TSMatrix M,TSMatrix N,TSMatrix &Q)
{ // 求稀疏矩阵的和Q=M+N
  int m=1,n=1,q=0;
  if(M.mu!=N.mu||M.nu!=N.nu) // M、N两稀疏矩阵行或列数不同
    return ERROR;
  Q.mu=M.mu;
  Q.nu=M.nu;
  while(m<=M.tu&&n<=N.tu) // 矩阵M和N的元素都没处理完
  {
    switch(comp(M.data[m].i,N.data[n].i))
    {
      case -1: Q.data[++q]=M.data[m++]; // 将矩阵M的当前元素值赋给矩阵Q
               break;
      case  0: switch(comp(M.data[m].j,N.data[n].j)) // M、N矩阵当前元素的行相等,继续比较列
               {
                 case -1: Q.data[++q]=M.data[m++];
                          break;
                 case  0: Q.data[++q]=M.data[m++]; // M、N矩阵当前非零元素的行列均相等
                          Q.data[q].e+=N.data[n++].e; //矩阵M、N的当前元素值求和并赋给矩阵Q
                          if(Q.data[q].e==0) // 元素值为0，不存入压缩矩阵
                            q--;
                          break;
                 case  1: Q.data[++q]=N.data[n++];
               }
               break;
      case  1: Q.data[++q]=N.data[n++]; // 将矩阵N的当前元素值赋给矩阵Q
```

```
    }
  }
  while(m<=M.tu) // 矩阵N的元素全部处理完毕
    Q.data[++q]=M.data[m++];
  while(n<=N.tu) // 矩阵M的元素全部处理完毕
    Q.data[++q]=N.data[n++];
  Q.tu=q; // 矩阵Q的非零元素个数
  if(q>MAX_SIZE) // 非零元素个数太多
    return ERROR;
  return OK;
}
Status SubtSMatrix(TSMatrix M,TSMatrix N,TSMatrix &Q)
{ // 求稀疏矩阵的差Q=M-N
  int i;
  for(i=1;i<=N.tu;i++)
    N.data[i].e*=-1;
  return AddSMatrix(M,N,Q);
}
void TransposeSMatrix(TSMatrix M,TSMatrix &T)
{ // 求稀疏矩阵M的转置矩阵T。算法5.1改
  int p,q,col;
  T.mu=M.nu;
  T.nu=M.mu;
  T.tu=M.tu;
  if(T.tu)
  {
    q=1;
    for(col=1;col<=M.nu;++col)
      for(p=1;p<=M.tu;++p)
        if(M.data[p].j==col)
        {
          T.data[q].i=M.data[p].j;
          T.data[q].j=M.data[p].i;
          T.data[q].e=M.data[p].e;
          ++q;
        }
  }
}
Status MultSMatrix(TSMatrix M,TSMatrix N,TSMatrix &Q)
{ // 求稀疏矩阵的乘积Q=M× N
  int i,j;
  ElemType *Nc,*Tc;
  TSMatrix T; // 临时矩阵
  if(M.nu!=N.mu)
    return ERROR;
  T.nu=M.mu; // 临时矩阵T是Q的转秩矩阵
  T.mu=N.nu;
  T.tu=0;
  Nc=(ElemType*)malloc((N.mu+1)*sizeof(ElemType));//Nc为矩阵N一列的临时数组(非压缩,[0]不用)
  Tc=(ElemType*)malloc((M.nu+1)*sizeof(ElemType));//Tc为矩阵T一行的临时数组(非压缩,[0]不用)
  if(!Nc||!Tc) // 创建临时数组不成功
```

```
    exit(ERROR);
  for(i=1;i<=N.nu;i++) // 对于N的每一列
  {
    for(j=1;j<=N.mu;j++)
      Nc[j]=0; // 矩阵Nc的初值为0
    for(j=1;j<=M.mu;j++)
      Tc[j]=0; // 临时数组Tc的初值为0, [0]不用
    for(j=1;j<=N.tu;j++) // 对于N的每一个非零元素
      if(N.data[j].j==i) // 属于第i列
        Nc[N.data[j].i]=N.data[j].e; // 根据其所在行将其元素值赋给相应的Nc
    for(j=1;j<=M.tu;j++) // 对于M的每一个值
      Tc[M.data[j].i]+=M.data[j].e*Nc[M.data[j].j]; // Tc中存N的第i列与M相乘的结果
    for(j=1;j<=M.mu;j++)
      if(Tc[j]!=0)
      {
        T.data[++T.tu].e=Tc[j];
        T.data[T.tu].i=i;
        T.data[T.tu].j=j;
      }
  }
  if(T.tu>MAX_SIZE) // 非零元素个数太多
    return ERROR;
  TransposeSMatrix(T,Q); // 将T的转秩赋给Q
  DestroySMatrix(T); // 销毁临时矩阵T
  free(Tc); // 释放动态数组Tc和Nc
  free(Nc);
  return OK;
}

// main5-2.cpp 检验bo5-2.cpp的主程序
#include"c1.h"
typedef int ElemType;
#include"c5-2.h"
#include"bo5-2.cpp"
void main()
{
  TSMatrix A,B,C;
  printf("创建矩阵A: ");
  CreateSMatrix(A);
  PrintSMatrix(A);
  printf("由矩阵A复制矩阵B:\n");
  CopySMatrix(A,B);
  PrintSMatrix1(B);
  DestroySMatrix(B);
  printf("销毁矩阵B后:\n");
  PrintSMatrix1(B);
  printf("创建矩阵B2:(与矩阵A的行、列数相同, 行、列分别为%d,%d)\n",A.mu,A.nu);
  CreateSMatrix(B);
  PrintSMatrix1(B);
  AddSMatrix(A,B,C);
  printf("矩阵C1(A+B):\n");
```

```
  PrintSMatrix1(C);
  SubtSMatrix(A,B,C);
  printf("矩阵C2(A-B):\n");
  PrintSMatrix1(C);
  TransposeSMatrix(A,C);
  printf("矩阵C3(A的转置):\n");
  PrintSMatrix1(C);
  printf("创建矩阵A2: ");
  CreateSMatrix(A);
  PrintSMatrix1(A);
  printf("创建矩阵B3:(行数应与矩阵A2的列数相同=%d)\n",A.nu);
  CreateSMatrix(B);
  PrintSMatrix1(B);
  MultSMatrix(A,B,C);
  printf("矩阵C5(A× B):\n");
  PrintSMatrix1(C);
}
```

程序运行结果：

```
创建矩阵A: 请输入矩阵的行数,列数,非零元素数: 3,3,2↙
请按行序顺序输入第1个非零元素所在的行(1～3),列(1～3),元素值:1,2,1↙
请按行序顺序输入第2个非零元素所在的行(1～3),列(1～3),元素值:2,2,2↙
3行3列2个非零元素。(见图5－7)
行  列  元素值
 1   2       1
 2   2       2
由矩阵A复制矩阵B:
  0  1  0
  0  2  0
  0  0  0
销毁矩阵B后:
创建矩阵B2:(与矩阵A的行、列数相同, 行、列分别为3,3)
请输入矩阵的行数,列数,非零元素数: 3,3,1↙
请按行序顺序输入第1个非零元素所在的行(1～3),列(1～3),元素值:1,2,1↙
  0  1  0
  0  0  0
  0  0  0
矩阵C1(A+B):
  0  2  0
  0  2  0
  0  0  0
矩阵C2(A-B):
  0  0  0
  0  2  0
  0  0  0
矩阵C3(A的转置):
  0  0  0
  1  2  0
```

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

图 5－7　矩阵 A 图示

```
  0  0  0
创建矩阵A2: 请输入矩阵的行数,列数,非零元素数: 2,3,2↙
请按行序顺序输入第1个非零元素所在的行(1～2),列(1～3),元素值:1,1,1↙
请按行序顺序输入第2个非零元素所在的行(1～2),列(1～3),元素值:2,3,2↙
  1  0  0
  0  0  2
创建矩阵B3:(行数应与矩阵A2的列数相同=3)
请输入矩阵的行数,列数,非零元素数: 3,2,2↙
请按行序顺序输入第1个非零元素所在的行(1～3),列(1～2),元素值:2,2,1↙
请按行序顺序输入第2个非零元素所在的行(1～3),列(1～2),元素值:3,1,2↙
  0  0
  0  1
  2  0
矩阵C5(A× B):
  0  0
  4  0
```

```
// algo5-1.cpp 实现算法5.2的程序
#include"c1.h"
typedef int ElemType;
#include"c5-2.h"
#include"bo5-2.cpp"
void FastTransposeSMatrix(TSMatrix M,TSMatrix &T)
{ // 快速求稀疏矩阵M的转置矩阵T。算法5.2改
  int p,q,t,col,*num,*cpot;
  num=(int *)malloc((M.nu+1)*sizeof(int)); // 存M每列(T每行)非零元素个数([0]不用)
  cpot=(int *)malloc((M.nu+1)*sizeof(int)); // 存T每行的下1个非零元素的存储位置([0]不用)
  T.mu=M.nu; // 给T的行、列数与非零元素个数赋值
  T.nu=M.mu;
  T.tu=M.tu;
  if(T.tu) // 是非零矩阵
  {
    for(col=1;col<=M.nu;++col)
      num[col]=0; // 计数器初值设为0
    for(t=1;t<=M.tu;++t) // 求M中每一列含非零元素个数
      ++num[M.data[t].j];
    cpot[1]=1; // T的第1行的第1个非零元在T.data中的序号为1
    for(col=2;col<=M.nu;++col)
      cpot[col]=cpot[col-1]+num[col-1]; // 求T的第col行的第1个非零元在T.data中的序号
    for(p=1;p<=M.tu;++p) // 从M的第1个元素开始
    {
      col=M.data[p].j; // 求得在M中的列数
      q=cpot[col]; // q指示M当前的元素在T中的序号
      T.data[q].i=M.data[p].j;
      T.data[q].j=M.data[p].i;
      T.data[q].e=M.data[p].e;
      ++cpot[col]; // T第col行的下1个非零元在T.data中的序号
    }
  }
  free(num);
  free(cpot);
```

```
}
void main()
{
  TSMatrix A,B;
  printf("创建矩阵A: ");
  CreateSMatrix(A);
  PrintSMatrix1(A);
  FastTransposeSMatrix(A,B);
  printf("矩阵B(A的快速转置):\n");
  PrintSMatrix1(B);
}
```

程序运行结果:

```
创建矩阵A: 请输入矩阵的行数,列数,非零元素数: 3,2,4↙
请按行序顺序输入第1个非零元素所在的行(1～3),列(1～2),元素值:1,1,1↙
请按行序顺序输入第2个非零元素所在的行(1～3),列(1～2),元素值:2,1,2↙
请按行序顺序输入第3个非零元素所在的行(1～3),列(1～2),元素值:3,1,3↙
请按行序顺序输入第4个非零元素所在的行(1～3),列(1～2),元素值:3,2,4↙
  1  0
  2  0
  3  4
矩阵B(A的快速转置):
  1  2  3
  0  0  4
```

由于稀疏矩阵的三元组顺序表存储结构要求先按行、同行再按列顺序存储非零元素，算法 5.1(在 bo5-2.cpp 中)采用了双重循环求转置矩阵，对于外层循环 col(列)的每一个值，对所有的非零元素，如果其列数与 col 相等，则按顺序存入转秩矩阵。这就保证了转秩矩阵也是先按行、同行再按列顺序存储非零元素。所以它的时间复杂度为 O(列数× 非零元素数)。而算法 5.2(在 algo5-1.cpp 中)采用了 2 个单循环。第 1 个循环，对所有的非零元素，计算其所在列并计数，得到每列的非零元素数 num[col]及每列第 1 个非零元素在转秩矩阵中的存储位置 cpot[col]。第 2 个循环，对所有的非零元素，根据其列数和 cpot[col]的当前值，存入转秩矩阵。由于还要给 num[col]和 cpot[col]赋初值，它的时间复杂度为 O(列数+非零元素数)。

```
// c5-3.h 稀疏矩阵的三元组行逻辑链接的顺序表存储表示(见图5－8)
#define MAX_SIZE 100 // 非零元个数的最大值
#define MAX_RC 20 // 最大行列数
struct Triple // 同c5-2.h
{
  int i,j; // 行下标,列下标
  ElemType e; // 非零元素值
};
struct RLSMatrix
```

```
{
  Triple data[MAX_SIZE+1]; // 非零元三元组表, data[0]未用
  int rpos[MAX_RC+1]; // 各行第一个非零元素的位置表, 比c5-2.h增加的项
  int mu,nu,tu; // 矩阵的行数、列数和非零元个数
};
```

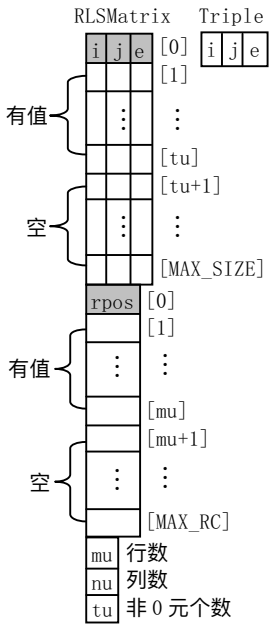


图 5－8　三元组行逻辑链接顺序表存储结构

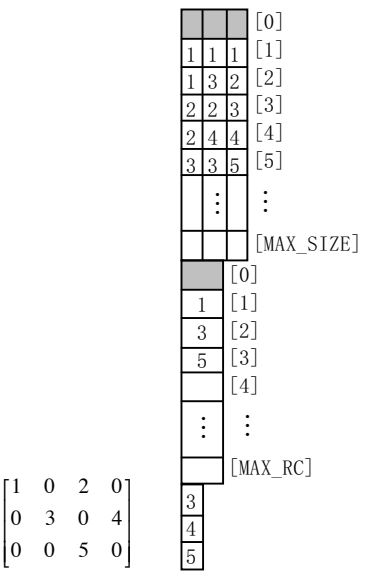


图 5－9　采用三元组行逻辑链接存储稀疏矩阵的实例

三元组行逻辑链接的顺序表存储表示(c5-3.h)比三元组顺序表存储表示(c5-2.h)增加了 rpos 数组，用以存放各行的第一个非零元素在 data 数组中的位置。这样，就可以迅速地找到某一行的元素。图 5－9 是采用三元组行逻辑链接的顺序表存储稀疏矩阵的实例。和 c5-2.h 存储结构一样，创建稀疏矩阵输入非零元时，也要按行、列的顺序由小到大输入。

```
// bo5-3.cpp 行逻辑链接稀疏矩阵(存储结构由c5-3.h定义)的基本操作(8个),包括算法5.3
Status CreateSMatrix(RLSMatrix &M)
{ // 创建稀疏矩阵M
  int i,j;
  Triple T;
  Status k;
  printf("请输入矩阵的行数,列数,非零元素数: ");
  scanf("%d,%d,%d",&M.mu,&M.nu,&M.tu);
  if(M.tu>MAX_SIZE||M.mu>MAX_RC)
    return ERROR;
  M.data[0].i=0; // 为以下比较做准备
  for(i=1;i<=M.tu;i++)
  {
    do
    {
      printf("请按行序顺序输入第%d个非零元素所在的行(1～%d),列(1～%d),元素值:",i,M.mu,M.nu);
```

```
      scanf("%d,%d,%d",&T.i,&T.j,&T.e);
      k=0;
      if(T.i<1||T.i>M.mu||T.j<1||T.j>M.nu) // 行、列超出范围
        k=1;
      if(T.i<M.data[i-1].i||T.i==M.data[i-1].i&&T.j<=M.data[i-1].j)//没有按顺序输入非零元素
        k=1;
    }while(k); // 当输入有误，重新输入
    M.data[i]=T;
  }
  for(i=1;i<=M.mu;i++) // 给rpos[]赋初值0
    M.rpos[i]=0;
  for(i=1;i<=M.tu;i++) // 计算每行非零元素数并赋给rpos[]
    M.rpos[M.data[i].i]++;
  for(i=M.mu;i>=1;i--) // 计算rpos[]
  {
    M.rpos[i]=1; // 赋初值1
    for(j=i-1;j>=1;j--)
      M.rpos[i]+=M.rpos[j];
  }
  return OK;
}
void DestroySMatrix(RLSMatrix &M)
{ // 销毁稀疏矩阵M(使M为0行0列0个非零元素的矩阵)
  M.mu=M.nu=M.tu=0;
}
void PrintSMatrix(RLSMatrix M)
{ // 输出稀疏矩阵M
  int i;
  printf("%d行%d列%d个非零元素。\n",M.mu,M.nu,M.tu);
  printf("行　列　元素值\n");
  for(i=1;i<=M.tu;i++)
    printf("%2d%4d%8d\n",M.data[i].i,M.data[i].j,M.data[i].e);
  for(i=1;i<=M.mu;i++)
    printf("第%d行的第一个非零元素是本矩阵第%d个元素\n",i,M.rpos[i]);
}
void PrintSMatrix1(RLSMatrix M)
{ // 按矩阵形式输出M
  int i,j,k=1;
  Triple *p=M.data;
  p++; // p指向第1个非零元素
  for(i=1;i<=M.mu;i++)
  {
    for(j=1;j<=M.nu;j++)
      if(k<=M.tu&&p->i==i&&p->j==j) // p指向非零元，且p所指元素为当前处理元素
      {
        printf("%3d",p->e); // 输出p所指元素的值
        p++; // p指向下一个元素
        k++; // 计数器+1
      }
      else // p所指元素不是当前处理元素
        printf("%3d",0); // 输出0
```

```
    printf("\n");
  }
}
void CopySMatrix(RLSMatrix M,RLSMatrix &T)
{ // 由稀疏矩阵M复制得到T
  T=M;
}
Status AddSMatrix(RLSMatrix M,RLSMatrix N,RLSMatrix &Q)
{ // 求稀疏矩阵的和Q=M+N
  int k,p,q,tm,tn;
  if(M.mu!=N.mu||M.nu!=N.nu)
    return ERROR;
  Q.mu=M.mu; // Q矩阵行数
  Q.nu=M.nu; // Q矩阵列数
  Q.tu=0; // Q矩阵非零元素数初值
  for(k=1;k<=M.mu;++k) // 对于每一行，k指示行号
  {
    Q.rpos[k]=Q.tu+1; // Q矩阵第k行的第1个元素的位置
    p=M.rpos[k]; // p指示M矩阵第k行当前元素的序号
    q=N.rpos[k]; // q指示N矩阵第k行当前元素的序号
    if(k==M.mu) // 是最后一行
    {
      tm=M.tu+1; // tm，tn分别是p，q的上界
      tn=N.tu+1;
    }
    else
    {
      tm=M.rpos[k+1];
      tn=N.rpos[k+1];
    }
    while(p<tm&&q<tn) // M，N矩阵均有第k行元素未处理
      if(M.data[p].j==N.data[q].j) // M矩阵当前元素的列=N矩阵当前元素的列
      {
        if(M.data[p].e+N.data[q].e!=0) // 和不为0，存入Q
        {
          Q.data[++Q.tu]=M.data[p];
          Q.data[Q.tu].e+=N.data[q].e;
        }
        p++;
        q++;
      }
      else if(M.data[p].j<N.data[q].j) // M矩阵当前元素的列<N矩阵当前元素的列
        Q.data[++Q.tu]=M.data[p++]; // 将M的当前值赋给Q
      else // M矩阵当前元素的列>N矩阵当前元素的列
        Q.data[++Q.tu]=N.data[q++]; // 将N的当前值赋给Q
    while(p<tm) // M矩阵还有第k行的元素未处理
      Q.data[++Q.tu]=M.data[p++]; // 将M的当前值赋给Q
    while(q<tn) // N矩阵还有k行的元素未处理
      Q.data[++Q.tu]=N.data[q++]; // 将N的当前值赋给Q
  }
  if(Q.tu>MAX_SIZE)
```

```
    return ERROR;
  else
    return OK;
}
Status SubtSMatrix(RLSMatrix M,RLSMatrix N,RLSMatrix &Q)
{ // 求稀疏矩阵的差Q=M-N
  int i;
  if(M.mu!=N.mu||M.nu!=N.nu)
    return ERROR;
  for(i=1;i<=N.tu;++i) // 对于N的每一元素，其值乘以-1
    N.data[i].e*=-1;
  AddSMatrix(M,N,Q); // Q=M+(-N)
  return OK;
}
Status MultSMatrix(RLSMatrix M,RLSMatrix N,RLSMatrix &Q)
{ // 求稀疏矩阵乘积Q=M× N。算法5.3改
  int arow,brow,p,q,ccol,ctemp[MAX_RC+1],t,tp;
  if(M.nu!=N.mu) // 矩阵M的列数应和矩阵N的行数相等
    return ERROR;
  Q.mu=M.mu; // Q初始化
  Q.nu=N.nu;
  Q.tu=0;
  if(M.tu*N.tu==0) // M和N至少有一个是零矩阵
    return ERROR;
  for(arow=1;arow<=M.mu;++arow)
  { // 从M的第一行开始，到最后一行，arow是M的当前行
    for(ccol=1;ccol<=Q.nu;++ccol)
      ctemp[ccol]=0; // Q的当前行的各列元素累加器清零
    Q.rpos[arow]=Q.tu+1; // Q当前行的第1个元素位于上1行最后1个元素之后
    if(arow<M.mu)
      tp=M.rpos[arow+1];
    else
      tp=M.tu+1; // 给最后1行设界
    for(p=M.rpos[arow];p<tp;++p)
    { // 对M当前行中每一个非零元
      brow=M.data[p].j; // 找到对应元在N中的行号(M当前元的列号)
      if(brow<N.mu)
        t=N.rpos[brow+1];
      else
        t=N.tu+1; // 给最后1行设界
      for(q=N.rpos[brow];q<t;++q)
      {
        ccol=N.data[q].j; // 乘积元素在Q中列号
        ctemp[ccol]+=M.data[p].e*N.data[q].e;
      }
    } // 求得Q中第arow行的非零元
    for(ccol=1;ccol<=Q.nu;++ccol) // 压缩存储该行非零元
      if(ctemp[ccol]!=0)
      {
        if(++Q.tu>MAX_SIZE)
          return ERROR;
```

```
        Q.data[Q.tu].i=arow;
        Q.data[Q.tu].j=ccol;
        Q.data[Q.tu].e=ctemp[ccol];
      }
  }
  return OK;
}
void TransposeSMatrix(RLSMatrix M,RLSMatrix &T)
{ // 求稀疏矩阵M的转置矩阵T
  int p,q,t,col,*num;
  num=(int *)malloc((M.nu+1)*sizeof(int));
  T.mu=M.nu;
  T.nu=M.mu;
  T.tu=M.tu;
  if(T.tu)
  {
    for(col=1;col<=M.nu;++col)
      num[col]=0;  // 设初值
    for(t=1;t<=M.tu;++t) // 求M中每一列非零元个数
      ++num[M.data[t].j];
    T.rpos[1]=1;
    for(col=2;col<=M.nu;++col) // 求M中第col中第一个非零元在T.data中的序号
      T.rpos[col]=T.rpos[col-1]+num[col-1];
    for(col=1;col<=M.nu;++col)
      num[col]=T.rpos[col];
    for(p=1;p<=M.tu;++p)
    {
      col=M.data[p].j;
      q=num[col];
      T.data[q].i=M.data[p].j;
      T.data[q].j=M.data[p].i;
      T.data[q].e=M.data[p].e;
      ++num[col];
    }
  }
  free(num);
}

// main5-3.cpp 检验bo5-3.cpp的主程序(与main5-2.cpp很相像)
#include"c1.h"
typedef int ElemType;
#include"c5-3.h" // 此行与main5-2.cpp不同
#include"bo5-3.cpp" // 此行与main5-2.cpp不同
void main()
{
  RLSMatrix A,B,C; // 此行与main5-2.cpp不同
  printf("创建矩阵A: ");
  CreateSMatrix(A);
  PrintSMatrix(A);
  printf("由矩阵A复制矩阵B:\n");
  CopySMatrix(A,B);
```

```
  PrintSMatrix1(B);
  DestroySMatrix(B);
  printf("销毁矩阵B后:\n");
  PrintSMatrix1(B);
  printf("创建矩阵B2:(与矩阵A的行、列数相同，行、列分别为%d,%d)\n",A.mu,A.nu);
  CreateSMatrix(B);
  PrintSMatrix1(B);
  AddSMatrix(A,B,C);
  printf("矩阵C1(A+B):\n");
  PrintSMatrix1(C);
  SubtSMatrix(A,B,C);
  printf("矩阵C2(A-B):\n");
  PrintSMatrix1(C);
  TransposeSMatrix(A,C);
  printf("矩阵C3(A的转置):\n");
  PrintSMatrix1(C);
  printf("创建矩阵A2:\n");
  CreateSMatrix(A);
  PrintSMatrix1(A);
  printf("创建矩阵B3:(行数应与矩阵A2的列数相同=%d)\n",A.nu);
  CreateSMatrix(B);
  PrintSMatrix1(B);
  MultSMatrix(A,B,C);
  printf("矩阵C5(A× B):\n");
  PrintSMatrix1(C);
}
```

程序运行结果：

```
创建矩阵A: 请输入矩阵的行数,列数,非零元素数: 3,3,2↙
请按行序顺序输入第1个非零元素所在的行(1～3),列(1～3),元素值:1,2,1↙
请按行序顺序输入第2个非零元素所在的行(1～3),列(1～3),元素值:2,2,2↙
3行3列2个非零元素。(见图5－7)
行  列  元素值
 1   2       1
 2   2       2
第1行的第一个非零元素是本矩阵第1个元素
第2行的第一个非零元素是本矩阵第2个元素
第3行的第一个非零元素是本矩阵第3个元素
由矩阵A复制矩阵B:
  0  1  0
  0  2  0
  0  0  0
销毁矩阵B后:
创建矩阵B2:(与矩阵A的行、列数相同，行、列分别为3,3)
请输入矩阵的行数,列数,非零元素数: 3,3,1↙
请按行序顺序输入第1个非零元素所在的行(1～3),列(1～3),元素值:1,2,1↙
  0  1  0
  0  0  0
```

```
  0  0  0
矩阵C1(A+B):
  0  2  0
  0  2  0
  0  0  0
矩阵C2(A-B):
  0  0  0
  0  2  0
  0  0  0
矩阵C3(A的转置):
  0  0  0
  1  2  0
  0  0  0
创建矩阵A2:
请输入矩阵的行数,列数,非零元素数: 2,3,2↙
请按行序顺序输入第1个非零元素所在的行(1～2),列(1～3),元素值:1,1,1↙
请按行序顺序输入第2个非零元素所在的行(1～2),列(1～3),元素值:2,3,2↙
  1  0  0
  0  0  2
创建矩阵B3:(行数应与矩阵A2的列数相同=3)
请输入矩阵的行数,列数,非零元素数: 3,2,2↙
请按行序顺序输入第1个非零元素所在的行(1～3),列(1～2),元素值:2,2,1↙
请按行序顺序输入第2个非零元素所在的行(1～3),列(1～2),元素值:3,1,2↙
  0  0
  0  1
  2  0
矩阵C5(A× B):
  0  0
  4  0
```

```
// c5-4.h 稀疏矩阵的十字链表存储表示(见图5－10)
struct OLNode
{
  int i,j; // 该非零元的行和列下标
  ElemType e; // 非零元素值
  OLNode *right,*down;
          // 该非零元所在行表和列表的后继链域
};
typedef OLNode *OLink;
struct CrossList
{
  OLink *rhead,*chead;
  // 行和列链表头指针向量基址，由CreatSMatrix_OL()分配
  int mu,nu,tu; // 稀疏矩阵的行数、列数和非零元个数
};
```

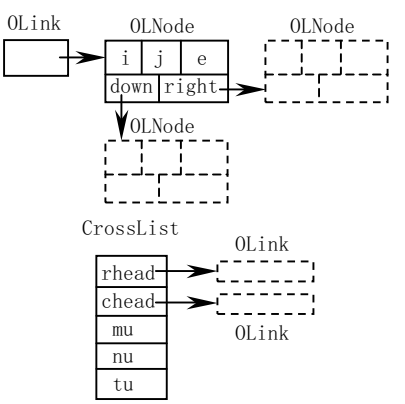
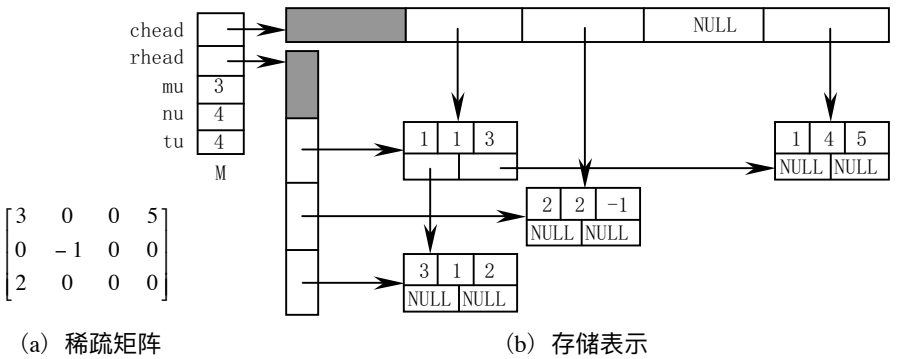


图 5－10　稀疏矩阵的十字链表存储结构

图 5－11 是采用十字链表存储稀疏矩阵的实例(教科书图 5.6)。由于十字链表存储结构中的非零元素是按其所在行、列插入相应的链表的，所以，在创建稀疏矩阵输入非零元时，可以按任意顺序输入非零元素。每个非零元结点按升序被插入到两个没有头结点的单链表中：一个是所在行链表；另一个是所在列链表。当插入或删除结点时，只要修改相关

的行、列链表即可，比较灵活。



(a) 稀疏矩阵　　(b) 存储表示

图 5-11　采用十字链表存储稀疏矩阵的示例

```
// bo5-4.cpp 稀疏矩阵的十字链表存储(存储结构由c5-4.h定义)的基本操作(9个)，包括算法5.4
void InitSMatrix(CrossList &M)
{ // 初始化M(CrossList类型的变量必须初始化，否则创建、复制矩阵将出错)。加(见图5-12)
  M.rhead=M.chead=NULL;
  M.mu=M.nu=M.tu=0;
}
void InitSMatrixList(CrossList &M)
{ // 初始化十字链表表头指针向量。加(见图5-13)
  int i;
  if(!(M.rhead=(OLink*)malloc((M.mu+1)*sizeof(OLink))))
  // 生成行表头指针向量
    exit(OVERFLOW);
  if(!(M.chead=(OLink*)malloc((M.nu+1)*sizeof(OLink))))
  // 生成列表头指针向量
    exit(OVERFLOW);
  for(i=1;i<=M.mu;i++)
  // 初始化矩阵T的行表头指针向量，各行链表为空
    M.rhead[i]=NULL;
  for(i=1;i<=M.nu;i++)
  // 初始化矩阵T的列表头指针向量，各列链表为空
    M.chead[i]=NULL;
}
```

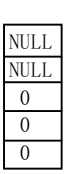


图 5-12　初始化和销毁稀疏矩阵

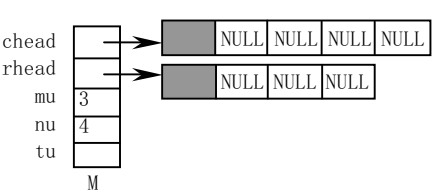
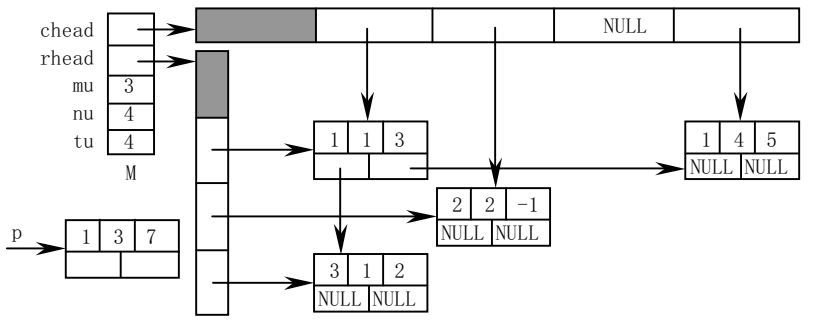
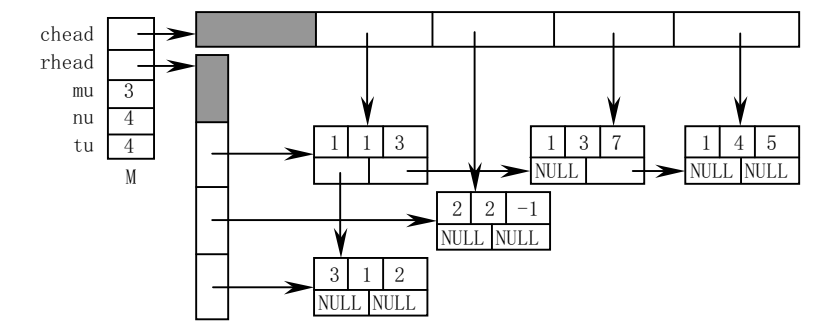


图 5-13　调用 InitSMatrixList()的示例

```
void InsertAscend(CrossList &M,OLink p)
{ // 初始条件：稀疏矩阵M存在,p指向的结点存在。
  // 操作结果：按行列升序将p所指结点插入M(见图5-14)
```



(a) 插入前



(b) 插入后

图 5 - 14　按升序将 p 所指结点插入十字链表存储稀疏矩阵 M 示例

```
OLink q=M.rhead[p->i]; // q指向待插行表
if(!q||p->j<q->j) // 待插的行表空或p所指结点的列值小于首结点的列值
{
  p->right=M.rhead[p->i]; // 插在表头
  M.rhead[p->i]=p;
}
else
{
 while(q->right&&q->right->j<p->j)//q所指不是尾结点且q的下一结点的列值小于p所指结点的列值
    q=q->right; // q向后移
  p->right=q->right; // 将p插在q所指结点之后
  q->right=p;
}
q=M.chead[p->j]; // q指向待插列表
if(!q||p->i<q->i) // 待插的列表空或p所指结点的行值小于首结点的行值
{
  p->down=M.chead[p->j]; // 插在表头
  M.chead[p->j]=p;
}
else
{
  while(q->down&&q->down->i<p->i) //q所指不是尾结点且q的下一结点的行值小于p所指结点的行值
    q=q->down; // q向下移
  p->down=q->down; // 将p插在q所指结点之下
```

```
    q->down=p;
  }
  M.tu++;
}
void DestroySMatrix(CrossList &M)
{ // 初始条件: 稀疏矩阵M存在。操作结果: 销毁稀疏矩阵M(见图5-12)
  int i;
  OLink p,q;
  for(i=1;i<=M.mu;i++) // 按行释放结点
  {
    p=*(M.rhead+i);
    while(p)
    {
      q=p;
      p=p->right;
      free(q);
    }
  }
  free(M.rhead);
  free(M.chead);
  InitSMatrix(M);
}
void CreateSMatrix(CrossList &M)
{ // 创建稀疏矩阵M, 采用十字链表存储表示。算法5.4改
  int i,k;
  OLink p;
  if(M.rhead)
    DestroySMatrix(M);
  printf("请输入稀疏矩阵的行数 列数 非零元个数: ");
  scanf("%d%d%d",&M.mu,&M.nu,&i);
  InitSMatrixList(M); // 初始化M的表头指针向量
  printf("请按任意次序输入%d个非零元的行 列 元素值:\n",M.tu);
  for(k=0;k<i;k++)
  {
    p=(OLink)malloc(sizeof(OLNode)); // 生成结点
    if(!p)
      exit(OVERFLOW);
    scanf("%d%d%d",&p->i,&p->j,&p->e); // 给结点的3个成员赋值
    InsertAscend(M,p); // 将结点p按行列值升序插到矩阵M中
  }
}
void PrintSMatrix(CrossList M)
{ // 初始条件: 稀疏矩阵M存在。操作结果: 按行或按列输出稀疏矩阵M
  int i,j;
  OLink p;
  printf("%d行%d列%d个非零元素\n",M.mu,M.nu,M.tu);
  printf("请输入选择(1.按行输出 2.按列输出): ");
  scanf("%d",&i);
  switch(i)
  {
```

```
    case 1: for(j=1;j<=M.mu;j++)
            {
              p=M.rhead[j];
              while(p)
              {
                cout<<p->i<<"行"<<p->j<<"列值为"<<p->e<<endl;
                p=p->right;
              }
            }
            break;
    case 2: for(j=1;j<=M.nu;j++)
            {
              p=M.chead[j];
              while(p)
              {
                cout<<p->i<<"行"<<p->j<<"列值为"<<p->e<<endl;
                p=p->down;
              }
            }
  }
}
void PrintSMatrix1(CrossList M)
{ // 按矩阵形式输出M
  int i,j;
  OLink p;
  for(i=1;i<=M.mu;i++)
  { // 从第1行到最后1行
    p=M.rhead[i]; // p指向该行的第1个非零元素
    for(j=1;j<=M.nu;j++) // 从第1列到最后1列
      if(!p||p->j!=j) // 已到该行表尾或当前结点的列值不等于当前列值
        printf("%-3d",0); // 输出0
      else
      {
        printf("%-3d",p->e);
        p=p->right;
      }
    printf("\n");
  }
}
void CopySMatrix(CrossList M,CrossList &T)
{ // 初始条件：稀疏矩阵M存在。操作结果：由稀疏矩阵M复制得到T
  int i;
  OLink p,q;
  if(T.rhead) // 矩阵T存在
    DestroySMatrix(T);
  T.mu=M.mu;
  T.nu=M.nu;
  InitSMatrixList(T); // 初始化T的表头指针向量
  for(i=1;i<=M.mu;i++) // 按行复制
  {
```

```
    p=M.rhead[i]; // p指向i行链表头
    while(p) // 没到行尾
    {
      if(!(q=(OLNode*)malloc(sizeof(OLNode)))) // 生成结点q
        exit(OVERFLOW);
      *q=*p; // 给结点q赋值
      InsertAscend(T,q); // 将结点q按行列值升序插到矩阵T中
      p=p->right;
    }
  }
}
int comp(int c1,int c2)
{ // AddSMatrix函数要用到，另加
  if(c1<c2)
    return -1;
  if(c1==c2)
    return 0;
  return 1;
}
void AddSMatrix(CrossList M,CrossList N,CrossList &Q)
{ // 初始条件：稀疏矩阵M与N的行数和列数对应相等。操作结果：求稀疏矩阵的和Q=M+N
  int i;
  OLink pq,pm,pn;
  if(M.mu!=N.mu||M.nu!=N.nu)
  {
    printf("两个矩阵不是同类型的,不能相加\n");
    exit(OVERFLOW);
  }
  Q.mu=M.mu; // 初始化Q矩阵
  Q.nu=M.nu;
  Q.tu=0; // Q矩阵元素个数的初值为0
  InitSMatrixList(Q); // 初始化Q的表头指针向量
  for(i=1;i<=M.mu;i++) // 按行的顺序相加
  {
    pm=M.rhead[i]; // pm指向矩阵M的第i行的第1个结点
    pn=N.rhead[i]; // pn指向矩阵N的第i行的第1个结点
    while(pm&&pn) // pm和pn均不空
    {
      pq=(OLink)malloc(sizeof(OLNode)); // 生成矩阵Q的结点
      switch(comp(pm->j,pn->j))
      {
        case -1: *pq=*pm; // M的列<N的列，将矩阵M的当前元素值赋给pq
                 InsertAscend(Q,pq); // 将结点pq按行列值升序插到矩阵Q中
                 pm=pm->right; // 指针向后移
                 break;
        case  0: *pq=*pm; // M、N矩阵的列相等，元素值相加
                 pq->e+=pn->e;
                 if(pq->e!=0) // 和为非零元素
                   InsertAscend(Q,pq); // 将结点pq按行列值升序插到矩阵Q中
                 else
```

```
                free(pq); // 释放结点
                pm=pm->right; // 指针向后移
                pn=pn->right;
                break;
        case  1: *pq=*pn; // M的列>N的列，将矩阵N的当前元素值赋给pq
                InsertAscend(Q,pq); // 将结点pq按行列值升序插到矩阵Q中
                pn=pn->right; // 指针向后移
      }
    }
    while(pm) // pn=NULL
    {
      pq=(OLink)malloc(sizeof(OLNode)); // 生成矩阵Q的结点
      *pq=*pm; // M的列<N的列，将矩阵M的当前元素值赋给pq
      InsertAscend(Q,pq); // 将结点pq按行列值升序插到矩阵Q中
      pm=pm->right; // 指针向后移
    }
    while(pn) // pm=NULL
    {
      pq=(OLink)malloc(sizeof(OLNode)); // 生成矩阵Q的结点
      *pq=*pn; // M的列>N的列，将矩阵N的当前元素值赋给pq
      InsertAscend(Q,pq); // 将结点pq按行列值升序插到矩阵Q中
      pn=pn->right; // 指针向后移
    }
  }
  if(Q.tu==0) // Q矩阵元素个数为0
    DestroySMatrix(Q); // 销毁矩阵Q
}
void SubtSMatrix(CrossList M,CrossList N,CrossList &Q)
{ // 初始条件：稀疏矩阵M与N的行数和列数对应相等。操作结果：求稀疏矩阵的差Q=M-N
  int i;
  OLink pq,pm,pn;
  if(M.mu!=N.mu||M.nu!=N.nu)
  {
    printf("两个矩阵不是同类型的,不能相减\n");
    exit(OVERFLOW);
  }
  Q.mu=M.mu; // 初始化Q矩阵
  Q.nu=M.nu;
  Q.tu=0; // Q矩阵元素个数的初值为0
  InitSMatrixList(Q); // 初始化Q的表头指针向量
  for(i=1;i<=M.mu;i++) // 按行的顺序相减
  {
    pm=M.rhead[i]; // pm指向矩阵M的第i行的第1个结点
    pn=N.rhead[i]; // pn指向矩阵N的第i行的第1个结点
    while(pm&&pn) // pm和pn均不空
    {
      pq=(OLink)malloc(sizeof(OLNode)); // 生成矩阵Q的结点
      switch(comp(pm->j,pn->j))
      {
        case -1: *pq=*pm; // M的列<N的列，将矩阵M的当前元素值赋给pq
```

```
                InsertAscend(Q,pq); // 将结点pq按行列值升序插到矩阵Q中
                pm=pm->right; // 指针向后移
                break;
        case  0: *pq=*pm; // M、N矩阵的列相等，元素值相减
                pq->e-=pn->e;
                if(pq->e!=0) // 差为非零元素
                  InsertAscend(Q,pq); // 将结点pq按行列值升序插到矩阵Q中
                else
                  free(pq); // 释放结点
                pm=pm->right; // 指针向后移
                pn=pn->right;
                break;
        case  1: *pq=*pn; // M的列>N的列，将矩阵N的当前元素值赋给pq
                pq->e*=-1; // 求反
                InsertAscend(Q,pq); // 将结点pq按行列值升序插到矩阵Q中
                pn=pn->right; // 指针向后移
      }
    }
    while(pm) // pn=NULL
    {
      pq=(OLink)malloc(sizeof(OLNode)); // 生成矩阵Q的结点
      *pq=*pm; // M的列<N的列，将矩阵M的当前元素值赋给pq
      InsertAscend(Q,pq); // 将结点pq按行列值升序插到矩阵Q中
      pm=pm->right; // 指针向后移
    }
    while(pn) // pm=NULL
    {
      pq=(OLink)malloc(sizeof(OLNode)); // 生成矩阵Q的结点
      *pq=*pn; // M的列>N的列，将矩阵N的当前元素值赋给pq
      pq->e*=-1; // 求反
      InsertAscend(Q,pq); // 将结点pq按行列值升序插到矩阵Q中
      pn=pn->right; // 指针向后移
    }
  }
  if(Q.tu==0) // Q矩阵元素个数为0
    DestroySMatrix(Q); // 销毁矩阵Q
}
void MultSMatrix(CrossList M,CrossList N,CrossList &Q)
{ // 初始条件：稀疏矩阵M的列数等于N的行数。操作结果：求稀疏矩阵乘积Q=M× N
  int i,j,e;
  OLink pq,pm,pn;
  InitSMatrix(Q);
  Q.mu=M.mu;
  Q.nu=N.nu;
  Q.tu=0;
  InitSMatrixList(Q); // 初始化Q的表头指针向量
  for(i=1;i<=Q.mu;i++)
    for(j=1;j<=Q.nu;j++)
    {
      pm=M.rhead[i];
```

```
      pn=N.chead[j];
      e=0;
      while(pm&&pn)
        switch(comp(pn->i,pm->j))
        {
          case -1: pn=pn->down; // 列指针后移
                   break;
          case  0: e+=pm->e*pn->e; // 乘积累加
                   pn=pn->down; // 行列指针均后移
                   pm=pm->right;
                   break;
          case  1: pm=pm->right; // 行指针后移
        }
      if(e) // 值不为0
      {
        pq=(OLink)malloc(sizeof(OLNode)); // 生成结点
        if(!pq) // 生成结点失败
          exit(OVERFLOW);
        pq->i=i; // 给结点赋值
        pq->j=j;
        pq->e=e;
        InsertAscend(Q,pq); // 将结点pq按行列值升序插到矩阵Q中
      }
    }
  if(Q.tu==0) // Q矩阵元素个数为0
    DestroySMatrix(Q); // 销毁矩阵Q
}
void TransposeSMatrix(CrossList M,CrossList &T)
{ // 初始条件：稀疏矩阵M存在。操作结果：求稀疏矩阵M的转置矩阵T
  int u,i;
  OLink *head,p,q,r;
  CopySMatrix(M,T); // T=M
  u=T.mu; // 交换T.mu和T.nu
  T.mu=T.nu;
  T.nu=u;
  head=T.rhead; // 交换T.rhead和T.chead
  T.rhead=T.chead;
  T.chead=head;
  for(u=1;u<=T.mu;u++) // 对T的每一行
  {
    p=T.rhead[u]; // p为行表头
    while(p) // 没到表尾，对T的每一结点
    {
      q=p->down; // q指向下一个结点
      i=p->i; // 交换.i和.j
      p->i=p->j;
      p->j=i;
      r=p->down; // 交换.down和.right
      p->down=p->right;
      p->right=r;
```

```
      p=q; // p指向下一个结点
    }
  }
}

// main5-4.cpp 检验bo5-4.cpp的主程序
#include"c1.h"
typedef int ElemType;
#include"c5-4.h"
#include"bo5-4.cpp"
void main()
{
  CrossList A,B,C;
  InitSMatrix(A); // CrossList类型的变量在初次使用之前必须初始化
  InitSMatrix(B);
  printf("创建矩阵A: ");
  CreateSMatrix(A);
  PrintSMatrix(A);
  printf("由矩阵A复制矩阵B: ");
  CopySMatrix(A,B);
  PrintSMatrix(B);
  DestroySMatrix(B); // CrossList类型的变量在再次使用之前必须先销毁
  printf("销毁矩阵B后,矩阵B为\n");
  PrintSMatrix1(B);
  printf("创建矩阵B2:(与矩阵A的行、列数相同, 行、列分别为%d,%d)\n",A.mu,A.nu);
  CreateSMatrix(B);
  PrintSMatrix1(B);
  printf("矩阵C1(A+B):\n");
  AddSMatrix(A,B,C);
  PrintSMatrix1(C);
  DestroySMatrix(C);
  printf("矩阵C2(A-B):\n");
  SubtSMatrix(A,B,C);
  PrintSMatrix1(C);
  DestroySMatrix(C);
  printf("矩阵C3(A的转置):\n");
  TransposeSMatrix(A,C);
  PrintSMatrix1(C);
  DestroySMatrix(A);
  DestroySMatrix(B);
  DestroySMatrix(C);
  printf("创建矩阵A2: ");
  CreateSMatrix(A);
  PrintSMatrix1(A);
  printf("创建矩阵B3:(行数应与矩阵A2的列数相同=%d)\n",A.nu);
  CreateSMatrix(B);
  PrintSMatrix1(B);
  printf("矩阵C5(A× B):\n");
  MultSMatrix(A,B,C);
  PrintSMatrix1(C);
```

```
  DestroySMatrix(A);
  DestroySMatrix(B);
  DestroySMatrix(C);
}
```

程序运行结果：

```
创建矩阵A: 请输入稀疏矩阵的行数 列数 非零元个数: 3 3 2↙
请按任意次序输入2个非零元的行 列 元素值:
2 1 2↙
1 2 1↙
3行3列2个非零元素(见图5－15)
请输入选择(1.按行输出 2.按列输出): 1↙
1行2列值为1
2行1列值为2
由矩阵A复制矩阵B: 3行3列2个非零元素
请输入选择(1.按行输出 2.按列输出): 2↙
2行1列值为2
1行2列值为1
销毁矩阵B后,矩阵B为
创建矩阵B2:(与矩阵A的行、列数相同，行、列分别为3,3)
请输入稀疏矩阵的行数 列数 非零元个数: 3 3 2↙
请按任意次序输入2个非零元的行 列 元素值:
2 2 5↙
1 2 -1↙
0  -1 0
0  5  0
0  0  0
矩阵C1(A+B):
0  0  0
2  5  0
0  0  0
矩阵C2(A-B):
0  2  0
2  -5 0
0  0  0
矩阵C3(A的转置):
0  2  0
1  0  0
0  0  0
创建矩阵A2: 请输入稀疏矩阵的行数 列数 非零元个数: 2 3 2↙
请按任意次序输入2个非零元的行 列 元素值:
1 1 1↙
2 3 2↙
1  0  0
0  0  2
创建矩阵B3:(行数应与矩阵A2的列数相同=3)
请输入稀疏矩阵的行数 列数 非零元个数: 3 2 2↙
请按任意次序输入2个非零元的行 列 元素值:
```

$$\begin{bmatrix} 0 & 1 & 0 \\ 2 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

图 5－15　矩阵 A 图示

```
2 2 1↙
3 1 2↙
0  0
0  1
2  0
矩阵C5(A× B):
0  0
4  0
```

5.4　广义表的定义

广义表，顾名思义，不是真正的线性表。它的结构更象第 6 章介绍的树结构，它的许多基本操作都是采用递归算法的。另外，还要定义如何表示一个广义表，也就是广义表的书写形式。本书采用的书写形式是字符串。算法 5.7(在 bo5-5.cpp 和 bo5-6.cpp 中)和算法 5.8(在 func5-1.cpp 中)将广义表的字符串书写形式转换为广义表的存储结构。

5.5　广义表的存储结构

```
// c5-5.h 广义表的头尾链表存储结构(见图5－16)
enum ElemTag{ATOM,LIST}; // ATOM==0: 原子, LIST==1: 子表
typedef struct GLNode
{
  ElemTag tag; // 公共部分，用于区分原子结点和表结点
  union // 原子结点和表结点的联合部分
  {
    AtomType atom; // atom是原子结点的值域，AtomType由用户定义
    struct
    {
      GLNode *hp,*tp;
    }ptr; // ptr是表结点的指针域，prt.hp和ptr.tp分别指向表头和表尾
  };
}*GList,GLNode; // 广义表类型
```

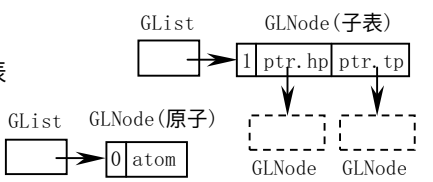


图 5－16　广义表的头尾链表存储结构

图 5－17 是根据 c5-5.h 定义的广义表(a, (b, c, d))的存储结构。它的长度为 2，第 1 个元素为原子 a，第 2 个元素为子表(b, c, d)。

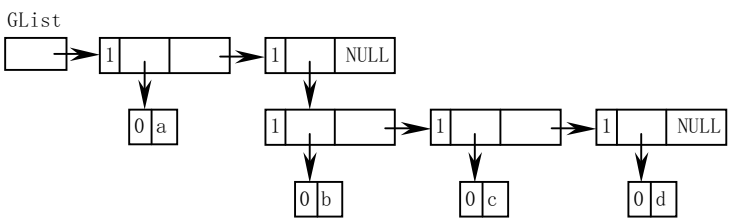


图 5－17　广义表(a, (b, c, d))的头尾链表存储结构

```
// c5-6.h 广义表的扩展线性链表存储表示(见图5－18)
enum ElemTag{ATOM,LIST};
          // ATOM==0: 原子, LIST==1: 子表
typedef struct GLNode1
{
  ElemTag tag; // 公共部分, 用于区分原子结点和表结点
  union // 原子结点和表结点的联合部分
  {
    AtomType atom; // 原子结点的值域
    GLNode1 *hp; // 表结点的表头指针
  };
  GLNode1 *tp; // 相当于线性链表的next, 指向下一个元素结点
}*GList1,GLNode1; // 广义表类型GList1是一种扩展的线性链表
```

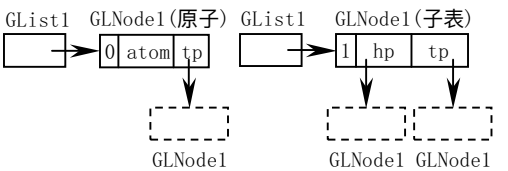


图 5－18　广义表的扩展线性链表存储结构

为了和 c5-5.h 定义的存储结构相区别，令 c5-6.h 定义的结构类型名为 GList1 和 GLNode1。

图 5－19 是根据 c5-6.h 定义的广义表(a,(b,c,d))的扩展线性链表存储结构。在这种结构中，广义表的头指针所指结点的 tag 域值总是 1(表)，其 tp 域总是 NULL。这样看来，广义表的头指针所指结点相当于表的头结点。和图 5－17 相比，图 5－19 这种结构更简洁些。

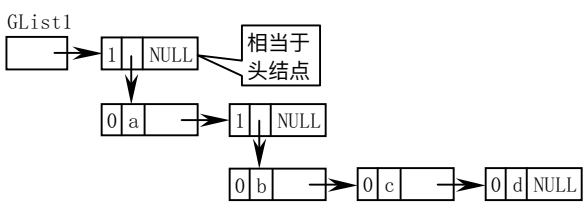


图 5－19　广义表(a,(b,c,d))的扩展线性链表存储结构

5.6　m 元多项式的表示

5.7　广义表的递归算法

5.7.1　求广义表的深度

算法 5.5 在 bo5-5.cpp 中。

5.7.2　复制广义表

算法 5.6 在 bo5-5.cpp 中。

5.7.3　建立广义表的存储结构

```
// func5-1.cpp 广义表的书写形式串为SString类型，包括算法5.8。bo5-5.cpp和bo5-6.cpp调用
#include"c4-1.h" // 定义SString类型
#include"bo4-1.cpp" // SString类型的基本操作
void sever(SString str,SString hstr) // 算法5.8改。SString是数组，不需引用类型
{ // 将非空串str分割成两部分：hstr为第一个',' 之前的子串，str为之后的子串
  int n,k,i; // k记尚未配对的左括号个数
  SString ch,c1,c2,c3;
  n=StrLength(str); // n为串str的长度
  StrAssign(c1,","); // c1=','
  StrAssign(c2,"("); // c2='('
  StrAssign(c3,")"); // c3=')'
  SubString(ch,str,1,1); // ch为串str的第1个字符
  for(i=1,k=0;i<=n&&StrCompare(ch,c1)||k!=0;++i) // i小于串长且ch不是','
  { // 搜索最外层的第一个逗号
    SubString(ch,str,i,1); // ch为串str的第i个字符
    if(!StrCompare(ch,c2)) // ch='('
      ++k; // 左括号个数+1
    else if(!StrCompare(ch,c3)) // ch=')'
      --k; // 左括号个数-1
  }
  if(i<=n) // 串str中存在',' ，它是第i-1个字符
  {
    SubString(hstr,str,1,i-2); // hstr返回串str',' 前的字符
    SubString(str,str,i,n-i+1); // str返回串str',' 后的字符
  }
  else // 串str中不存在','
  {
    StrCopy(hstr,str); // 串hstr就是串str
    ClearString(str); // ',' 后面是空串
  }
}

// bo5-5.cpp 广义表的头尾链表存储(存储结构由c5-5.h定义)的基本操作(11个),包括算法5.5,5.6,5.7
#include"func5-1.cpp" // 算法5.8
void InitGList(GList &L)
{ // 创建空的广义表L
  L=NULL;
}
void CreateGList(GList &L,SString S) // 算法5.7
{ // 采用头尾链表存储结构，由广义表的书写形式串S创建广义表L。设emp="()"
  SString sub,hsub,emp;
  GList p,q;
  StrAssign(emp,"()"); // 空串emp="()"
  if(!StrCompare(S,emp)) // S="()"
    L=NULL; // 创建空表
```

```
  else // S不是空串
  {
    if(!(L=(GList)malloc(sizeof(GLNode)))) // 建表结点
      exit(OVERFLOW);
    if(StrLength(S)==1) // S为单原子，只会出现在递归调用中
    {
      L->tag=ATOM;
      L->atom=S[1]; // 创建单原子广义表
    }
    else // S为表
    {
      L->tag=LIST;
      p=L;
      SubString(sub,S,2,StrLength(S)-2); // 脱外层括号(去掉第1个字符和最后1个字符)给串sub
      do
      { // 重复建n个子表
        sever(sub,hsub); // 从sub中分离出表头串hsub
        CreateGList(p->ptr.hp,hsub);
        q=p;
        if(!StrEmpty(sub)) // 表尾不空
        {
          if(!(p=(GLNode *)malloc(sizeof(GLNode))))
            exit(OVERFLOW);
          p->tag=LIST;
          q->ptr.tp=p;
        }
      }while(!StrEmpty(sub));
      q->ptr.tp=NULL;
    }
  }
}
void DestroyGList(GList &L)
{ // 销毁广义表L
  GList q1,q2;
  if(L)
  {
    if(L->tag==LIST) // 删除表结点
    {
      q1=L->ptr.hp; // q1指向表头
      q2=L->ptr.tp; // q2指向表尾
      DestroyGList(q1); // 销毁表头
      DestroyGList(q2); // 销毁表尾
    }
    free(L);
    L=NULL;
  }
}
void CopyGList(GList &T,GList L)
{ // 采用头尾链表存储结构，由广义表L复制得到广义表T。算法5.6
  if(!L) // 复制空表
    T=NULL;
```

```
  else
  {
    T=(GList)malloc(sizeof(GLNode)); // 建表结点
    if(!T)
      exit(OVERFLOW);
    T->tag=L->tag;
    if(L->tag==ATOM)
      T->atom=L->atom; // 复制单原子
    else
    {
      CopyGList(T->ptr.hp,L->ptr.hp); // 递归复制子表
      CopyGList(T->ptr.tp,L->ptr.tp);
    }
  }
}
int GListLength(GList L)
{ // 返回广义表的长度，即元素个数
  int len=0;
  while(L)
  {
    L=L->ptr.tp;
    len++;
  }
  return len;
}
int GListDepth(GList L)
{ // 采用头尾链表存储结构，求广义表L的深度。算法5.5
  int max,dep;
  GList pp;
  if(!L)
    return 1; // 空表深度为1
  if(L->tag==ATOM)
    return 0; // 原子深度为0，只会出现在递归调用中
  for(max=0,pp=L;pp;pp=pp->ptr.tp)
  {
    dep=GListDepth(pp->ptr.hp); // 递归求以pp->ptr.hp为头指针的子表深度
    if(dep>max)
      max=dep;
  }
  return max+1; // 非空表的深度是各元素的深度的最大值加1
}
Status GListEmpty(GList L)
{ // 判定广义表是否为空
  if(!L)
    return TRUE;
  else
    return FALSE;
}
GList GetHead(GList L)
{ // 生成广义表L的表头元素，返回指向这个元素的指针
  GList h,p;
```

```
  if(!L) // 空表无表头
    return NULL;
  p=L->ptr.hp; // p指向L的表头元素
  CopyGList(h,p); // 将表头元素复制给h
  return h;
}
GList GetTail(GList L)
{ // 将广义表L的表尾生成为广义表，返回指向这个新广义表的指针
  GList t;
  if(!L) // 空表无表尾
    return NULL;
  CopyGList(t,L->ptr.tp); // 将L的表尾拷给t
  return t;
}
void InsertFirst_GL(GList &L,GList e)
{ // 初始条件：广义表存在。操作结果：插入元素e(也可能是子表)作为广义表L的第1元素(表头)
  GList p=(GList)malloc(sizeof(GLNode)); // 生成新结点
  if(!p)
    exit(OVERFLOW);
  p->tag=LIST; // 结点的类型是表
  p->ptr.hp=e; // 表头指向e
  p->ptr.tp=L; // 表尾指向原表L
  L=p; // L指向新结点
}
void DeleteFirst_GL(GList &L,GList &e)
{ // 初始条件：广义表L存在。操作结果：删除广义表L的第一元素，并用e返回其值
  GList p=L; // p指向第1个结点
  e=L->ptr.hp; // e指向L的表头
  L=L->ptr.tp; // L指向原L的表尾
  free(p); // 释放第1个结点
}
void Traverse_GL(GList L,void(*v)(AtomType))
{ // 利用递归算法遍历广义表L
  if(L) // L不空
    if(L->tag==ATOM) // L为单原子
      v(L->atom);
    else // L为广义表
    {
      Traverse_GL(L->ptr.hp,v); // 递归遍历L的表头
      Traverse_GL(L->ptr.tp,v); // 递归遍历L的表尾
    }
}


// main5-5.cpp 检验bo5-5.cpp的主程序
#include"c1.h"
typedef char AtomType; // 定义原子类型为字符型
#include"c5-5.h" // 定义广义表的头尾链表存储
#include"bo5-5.cpp"
void visit(AtomType e)
{
  printf("%c ", e);
```

```
}
void main()
{
  char p[80];
  SString t;
  GList l,m;
  InitGList(l);
  InitGList(m);
  printf("空广义表l的深度=%d l是否空? %d(1:是 0:否)\n",GListDepth(l),GListEmpty(l));
  printf("请输入广义表l(书写形式: 空表:(),单原子:(a),其它:(a,(b),c)):\n");
  gets(p);
  StrAssign(t,p);
  CreateGList(l,t);
  printf("广义表l的长度=%d\n",GListLength(l));
  printf("广义表l的深度=%d l是否空? %d(1:是 0:否)\n",GListDepth(l),GListEmpty(l));
  printf("遍历广义表l: \n");
  Traverse_GL(l,visit);
  printf("\n复制广义表m=l\n");
  CopyGList(m,l);
  printf("广义表m的长度=%d\n",GListLength(m));
  printf("广义表m的深度=%d\n",GListDepth(m));
  printf("遍历广义表m: \n");
  Traverse_GL(m,visit);
  DestroyGList(m);
  m=GetHead(l);
  printf("\nm是l的表头元素, 遍历m: \n");
  Traverse_GL(m,visit);
  DestroyGList(m);
  m=GetTail(l);
  printf("\nm是由l的表尾形成的广义表, 遍历广义表m: \n");
  Traverse_GL(m,visit);
  InsertFirst_GL(m,l);
  printf("\n插入广义表l为m的表头, 遍历广义表m: \n");
  Traverse_GL(m,visit);
  printf("\n删除m的表头, 遍历广义表m: \n");
  DestroyGList(l);
  DeleteFirst_GL(m,l);
  Traverse_GL(m,visit);
  printf("\n");
  DestroyGList(m);
}
```

 　程序运行结果(以教科书图 5.7 为例):

```
空广义表l的深度=1 l是否空? 1(1:是 0:否)
请输入广义表l(书写形式: 空表:(),单原子:(a),其它:(a,(b),c)):
((),(e),(a,(b,c,d)))↙
广义表l的长度=3
广义表l的深度=3 l是否空? 0(1:是 0:否)
```

```
遍历广义表l:
e a b c d
复制广义表m=l
广义表m的长度=3
广义表m的深度=3
遍历广义表m:
e a b c d
m是l的表头元素，遍历m:

m是由l的表尾形成的广义表，遍历广义表m:
e a b c d
插入广义表l为m的表头，遍历广义表m:
e a b c d e a b c d
删除m的表头，遍历广义表m:
e a b c d
```

```
// bo5-6.cpp 广义表的扩展线性链表存储(存储结构由c5-6.h定义)的基本操作(13个)
#include"func5-1.cpp" // 算法5.8
void InitGList(GList1 &L)
{ // 创建空的广义表L
  L=NULL;
}
void CreateGList(GList1 &L,SString S) // 算法5.7改
{ // 采用扩展线性链表存储结构，由广义表的书写形式串S创建广义表L。设emp="()"
  SString emp,sub,hsub;
  GList1 p;
  StrAssign(emp,"()"); // 设emp="()"
  if(!(L=(GList1)malloc(sizeof(GLNode1)))) // 建表结点不成功
    exit(OVERFLOW);
  if(!StrCompare(S,emp)) // 创建空表
  {
    L->tag=LIST;
    L->hp=L->tp=NULL;
  }
  else if(StrLength(S)==1) // 创建单原子广义表
  {
    L->tag=ATOM;
    L->atom=S[1];
    L->tp=NULL;
  }
  else // 创建一般表
  {
    L->tag=LIST;
    L->tp=NULL;
    SubString(sub,S,2,StrLength(S)-2); // 脱外层括号(去掉第1个字符和最后1个字符)给串sub
    sever(sub,hsub); // 从sub中分离出表头串hsub
    CreateGList(L->hp,hsub);
    p=L->hp;
    while(!StrEmpty(sub)) // 表尾不空，则重复建n个子表
    {
      sever(sub,hsub); // 从sub中分离出表头串hsub
```

```
      CreateGList(p->tp,hsub);
      p=p->tp;
    };
  }
}
void DestroyGList(GList1 &L)
{ // 初始条件：广义表L存在。操作结果：销毁广义表L
  GList1 ph,pt;
  if(L) // L不为空表
  { // 由ph和pt接替L的两个指针
    if(L->tag) // 是子表
      ph=L->hp;
    else // 是原子
      ph=NULL;
    pt=L->tp;
    DestroyGList(ph); // 递归销毁表ph
    DestroyGList(pt); // 递归销毁表pt
    free(L); // 释放L所指结点
    L=NULL; // 令L为空
  }
}
void CopyGList(GList1 &T,GList1 L)
{ // 初始条件：广义表L存在。操作结果：由广义表L复制得到广义表T
  T=NULL;
  if(L) // L不空
  {
    T=(GList1)malloc(sizeof(GLNode1));
    if(!T)
      exit(OVERFLOW);
    T->tag=L->tag; // 复制枚举变量
    if(L->tag==ATOM) // 复制共用体部分
      T->atom=L->atom; // 复制单原子
    else
      CopyGList(T->hp,L->hp); // 复制子表
    if(L->tp==NULL) // 到表尾
      T->tp=L->tp;
    else
      CopyGList(T->tp,L->tp); // 复制子表
  }
}
int GListLength(GList1 L)
{ // 初始条件：广义表L存在。操作结果：求广义表L的长度，即元素个数
  int len=0;
  GList1 p=L->hp; // p指向第1个元素
  while(p)
  {
    len++;
    p=p->tp;
  };
  return len;
}
```

```
int GListDepth(GList1 L)
{ // 初始条件：广义表L存在。操作结果：求广义表L的深度
  int max,dep;
  GList1 pp;
  if(L==NULL||L->tag==LIST&&!L->hp)
    return 1; // 空表深度为1
  else if(L->tag==ATOM)
    return 0; // 单原子表深度为0，只会出现在递归调用中
  else // 求一般表的深度
    for(max=0,pp=L->hp;pp;pp=pp->tp)
    {
      dep=GListDepth(pp); // 求以pp为头指针的子表深度
      if(dep>max)
        max=dep;
    }
  return max+1; // 非空表的深度是各元素的深度的最大值加1
}
Status GListEmpty(GList1 L)
{ // 初始条件：广义表L存在。操作结果：判定广义表L是否为空
  if(!L||L->tag==LIST&&!L->hp)
    return OK;
  else
    return ERROR;
}
GList1 GetHead(GList1 L)
{ // 生成广义表L的表头元素，返回指向这个元素的指针
  GList1 h,p;
  if(!L||L->tag==LIST&&!L->hp) // 空表无表头
    return NULL;
  p=L->hp->tp; // p指向L的表尾
  L->hp->tp=NULL; // 截去L的表尾部分
  CopyGList(h,L->hp); // 将表头元素复制给h
  L->hp->tp=p; // 恢复L的表尾(保持原L不变)
  return h;
}
GList1 GetTail(GList1 L)
{ // 将广义表L的表尾生成为广义表，返回指向这个新广义表的指针
  GList1 t,p;
  if(!L||L->tag==LIST&&!L->hp) // 空表无表尾
    return NULL;
  p=L->hp; // p指向表头
  L->hp=p->tp; // 在L中删去表头
  CopyGList(t,L); // 将L的表尾拷给t
  L->hp=p; // 恢复L的表头(保持原L不变)
  return t;
}
void InsertFirst_GL(GList1 &L,GList1 e)
{ // 初始条件：广义表存在。操作结果：插入元素e(也可能是子表)作为广义表L的第1元素(表头)
  GList1 p=L->hp;
  L->hp=e;
  e->tp=p;
```

```
}
void DeleteFirst_GL(GList1 &L,GList1 &e)
{ // 初始条件：广义表L存在。操作结果：删除广义表L的第一元素，并用e返回其值
  if(L&&L->hp)
  {
    e=L->hp;
    L->hp=e->tp;
    e->tp=NULL;
  }
  else
    e=L;
}
void Traverse_GL(GList1 L,void(*v)(AtomType))
{ // 利用递归算法遍历广义表L
  GList1 hp;
  if(L) // L不空
  {
    if(L->tag==ATOM) // L为单原子
    {
      v(L->atom);
      hp=NULL;
    }
    else // L为子表
      hp=L->hp;
    Traverse_GL(hp,v);
    Traverse_GL(L->tp,v);
  }
}

// main5-6.cpp 检验bo5-6.cpp的主程序
#include"c1.h"
typedef char AtomType; // 定义原子类型为字符型
#include"c5-6.h" // 定义广义表的扩展线性链表存储结构
#include"bo5-6.cpp" // 广义表的扩展线性链表存储结构基本操作
void visit(AtomType e)
{
  printf("%c ", e);
}
void main()
{
  char p[80];
  GList1 l,m;
  SString t;
  InitGList(l); // 建立空的广义表l
  printf("空广义表l的深度=%d l是否空? %d(1:是 0:否)\n",GListDepth(l),GListEmpty(l));
  printf("请输入广义表l(书写形式: 空表:(),单原子:(a),其它:(a,(b),c)):\n");
  gets(p);
  StrAssign(t,p);
  CreateGList(l,t);
  printf("广义表l的长度=%d\n",GListLength(l));
  printf("广义表l的深度=%d l是否空? %d(1:是 0:否)\n",GListDepth(l),GListEmpty(l));
```

```
  printf("遍历广义表l: \n");
  Traverse_GL(l,visit);
  printf("\n复制广义表m=l\n");
  CopyGList(m,l);
  printf("广义表m的长度=%d\n",GListLength(m));
  printf("广义表m的深度=%d\n",GListDepth(m));
  printf("遍历广义表m: \n");
  Traverse_GL(m,visit);
  DestroyGList(m);
  m=GetHead(l);
  printf("\nm是l的表头元素，遍历m: \n");
  Traverse_GL(m,visit);
  DestroyGList(m);
  m=GetTail(l);
  printf("\nm是由l的表尾形成的广义表，遍历广义表m: \n");
  Traverse_GL(m,visit);
  InsertFirst_GL(m,l);
  printf("\n插入广义表l为m的表头，遍历广义表m: \n");
  Traverse_GL(m,visit);
  DeleteFirst_GL(m,l);
  printf("\n删除m的表头，遍历广义表m: \n");
  Traverse_GL(m,visit);
  printf("\n");
  DestroyGList(m);
}
```

程序运行结果：

```
空广义表l的深度=1 l是否空? 1(1:是 0:否)
请输入广义表l(书写形式: 空表:(),单原子:(a),其它:(a,(b),c)):
(a,(b),c)↙
广义表l的长度=3
广义表l的深度=2 l是否空? 0(1:是 0:否)
遍历广义表l:
a b c
复制广义表m=l
广义表m的长度=3
广义表m的深度=2
遍历广义表m:
a b c
m是l的表头元素，遍历广义表m:
a
m是由l的表尾形成的广义表，遍历广义表m:
b c
插入广义表l为m的表头，遍历广义表m:
a b c b c
删除m的表头，遍历广义表m:
b c
```

第 6 章　树和二叉树

6.1　树的定义和基本术语

6.2　二叉树

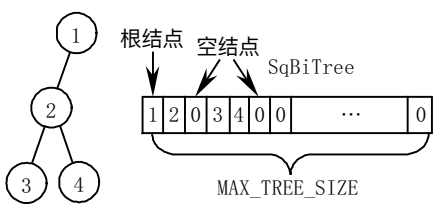
6.2.1　二叉树的定义

6.2.2　二叉树的性质

6.2.3　二叉树的存储结构

```
// c6-1.h 二叉树的顺序存储结构(见图6－1)
#define MAX_TREE_SIZE 100 // 二叉树的最大结点数
typedef TElemType SqBiTree[MAX_TREE_SIZE]; // 0号单元存储根结点
struct position
{
  int level,order; // 结点的层,本层序号(按满二叉树计算)
};
```

在顺序存储结构中，如图 6－2 所示，第 i 层结点的序号从 $2^{i-1}-1$～$2^{i}-2$；序号为 i 的结点，其双亲序号为(i+1)/2−1，其左右孩子序号分别为 2i+1 和 2i+2；除了根结点，序号为奇数的结点是其双亲的左孩子，它的右兄弟的序号是它的序号+1；序号为偶数的结点是其双亲的右孩子，它的左兄弟的序号是它的序号−1；i 层的满二叉树，其结点总数为 $2^{i}-1$。



(a) 二叉树　　(b) 存储表示

图 6－1　二叉树的顺序存储结构

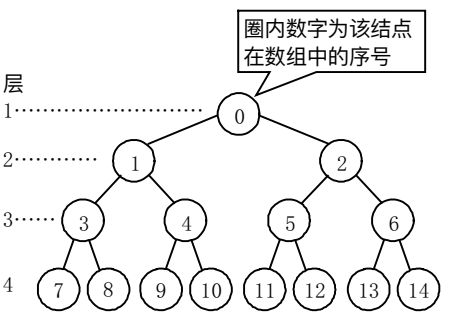


图 6－2　顺序存储结构的二叉树序号图示

显然，在顺序存储结构中，按层序输入二叉树是最方便的。当最后一个结点的值输入后，输入给定符号表示结束。二叉树的顺序存储结构适合存完全二叉树或近似完全二叉树。

bo6-1.cpp 是采用顺序存储结构的基本操作程序，main6-1.cpp 是检验这些基本操作的主程序。为了使这两个程序在结点类型为整型和字符型时都能使用，采用了编译预处理的“ #define”、“ #if”等命令。这样，只要将 main6-1.cpp 的第 2 行或第 3 行改为注释行即可。

```
// bo6-1.cpp 二叉树的顺序存储(存储结构由c6-1.h定义)的基本操作(23个)
#define ClearBiTree InitBiTree // 在顺序存储结构中，两函数完全一样
#define DestroyBiTree InitBiTree // 在顺序存储结构中，两函数完全一样
void InitBiTree(SqBiTree T)
{ // 构造空二叉树T。因为T是数组名，故不需要&
  int i;
  for(i=0;i<MAX_TREE_SIZE;i++)
    T[i]=Nil; // 初值为空(Nil在主程中定义)
}
void CreateBiTree(SqBiTree T)
{ // 按层序次序输入二叉树中结点的值(字符型或整型)，构造顺序存储的二叉树T
  int i=0;
  InitBiTree(T); // 构造空二叉树T
#if CHAR // 结点类型为字符
  int l;
  char s[MAX_TREE_SIZE];
  cout<<"请按层序输入结点的值(字符)，空格表示空结点，结点数≤"<<MAX_TREE_SIZE<<':'<<endl;
  gets(s); // 输入字符串
  l=strlen(s); // 求字符串的长度
  for(;i<l;i++) // 将字符串赋值给T
    T[i]=s[i];
#else // 结点类型为整型
  cout<<"请按层序输入结点的值(整型)，0表示空结点，输999结束。结点数≤"<<MAX_TREE_SIZE<<':'
  <<endl;
  while(1)
  {
    cin>>T[i];
    if(T[i]==999)
    {
      T[i]=Nil;
      break;
    }
    i++;
  }
#endif
  for(i=1;i<MAX_TREE_SIZE;i++)
    if(i!=0&&T[(i+1)/2-1]==Nil&&T[i]!=Nil) // 此结点(不空)无双亲且不是根
    {
      cout<<"出现无双亲的非根结点"<<T[i]<<endl;
      exit(ERROR);
    }
```

```
}
Status BiTreeEmpty(SqBiTree T)
{ // 初始条件：二叉树T存在。操作结果：若T为空二叉树，则返回TRUE；否则FALSE
  if(T[0]==Nil) // 根结点为空，则树空
    return TRUE;
  else
    return FALSE;
}
int BiTreeDepth(SqBiTree T)
{ // 初始条件：二叉树T存在。操作结果：返回T的深度
  int i,j=-1;
  for(i=MAX_TREE_SIZE-1;i>=0;i--) // 找到最后一个结点
    if(T[i]!=Nil)
      break;
  i++; // 为了便于计算
  do
    j++;
  while(i>=pow(2,j));
  return j;
}
Status Root(SqBiTree T,TElemType &e)
{ // 初始条件：二叉树T存在。操作结果：当T不空，用e返回T的根，返回OK；否则返回ERROR，e无定义
  if(BiTreeEmpty(T)) // T空
    return ERROR;
  else
  {
    e=T[0];
    return OK;
  }
}
TElemType Value(SqBiTree T,position e)
{ // 初始条件：二叉树T存在，e是T中某个结点(的位置)
  // 操作结果：返回处于位置e(层,本层序号)的结点的值
  return T[int(pow(2,e.level-1)+e.order-2)];
}
Status Assign(SqBiTree T,position e,TElemType value)
{ // 初始条件：二叉树T存在，e是T中某个结点(的位置)
  // 操作结果：给处于位置e(层,本层序号)的结点赋新值value
  int i=int(pow(2,e.level-1)+e.order-2); // 将层、本层序号转为矩阵的序号
  if(value!=Nil&&T[(i+1)/2-1]==Nil) // 给叶子赋非空值但双亲为空
    return ERROR;
  else if(value==Nil&&(T[i*2+1]!=Nil||T[i*2+2]!=Nil)) //  给双亲赋空值但有叶子(不空)
    return ERROR;
  T[i]=value;
  return OK;
}
TElemType Parent(SqBiTree T,TElemType e)
{ // 初始条件：二叉树T存在，e是T中某个结点
  // 操作结果：若e是T的非根结点，则返回它的双亲；否则返回“ 空”
  int i;
  if(T[0]==Nil) // 空树
    return Nil;
```

```
  for(i=1;i<=MAX_TREE_SIZE-1;i++)
    if(T[i]==e) // 找到e
      return T[(i+1)/2-1];
  return Nil; // 没找到e
}
TElemType LeftChild(SqBiTree T,TElemType e)
{ // 初始条件：二叉树T存在，e是T中某个结点。操作结果：返回e的左孩子。若e无左孩子,则返回“空”
  int i;
  if(T[0]==Nil) // 空树
    return Nil;
  for(i=0;i<=MAX_TREE_SIZE-1;i++)
    if(T[i]==e) // 找到e
      return T[i*2+1];
  return Nil; // 没找到e
}
TElemType RightChild(SqBiTree T,TElemType e)
{ // 初始条件：二叉树T存在，e是T中某个结点。操作结果：返回e的右孩子。若e无右孩子,则返回“空”
  int i;
  if(T[0]==Nil) // 空树
    return Nil;
  for(i=0;i<=MAX_TREE_SIZE-1;i++)
    if(T[i]==e) // 找到e
      return T[i*2+2];
  return Nil; // 没找到e
}
TElemType LeftSibling(SqBiTree T,TElemType e)
{ // 初始条件：二叉树T存在，e是T中某个结点
  // 操作结果：返回e的左兄弟。若e是T的左孩子或无左兄弟，则返回“空”
  int i;
  if(T[0]==Nil) // 空树
    return Nil;
  for(i=1;i<=MAX_TREE_SIZE-1;i++)
    if(T[i]==e&&i%2==0) // 找到e且其序号为偶数(是右孩子)
      return T[i-1];
  return Nil; // 没找到e
}
TElemType RightSibling(SqBiTree T,TElemType e)
{ // 初始条件：二叉树T存在，e是T中某个结点
  // 操作结果：返回e的右兄弟。若e是T的右孩子或无右兄弟，则返回“空”
  int i;
  if(T[0]==Nil) // 空树
    return Nil;
  for(i=1;i<=MAX_TREE_SIZE-1;i++)
    if(T[i]==e&&i%2) // 找到e且其序号为奇数(是左孩子)
      return T[i+1];
  return Nil; // 没找到e
}
void Move(SqBiTree q,int j,SqBiTree T,int i) // InsertChild()用到。加
{ // 把从q的j结点开始的子树移为从T的i结点开始的子树
  if(q[2*j+1]!=Nil) // q的左子树不空
    Move(q,(2*j+1),T,(2*i+1)); // 把q的j结点的左子树移为T的i结点的左子树
  if(q[2*j+2]!=Nil) // q的右子树不空
```

```
    Move(q,(2*j+2),T,(2*i+2)); // 把q的j结点的右子树移为T的i结点的右子树
  T[i]=q[j]; // 把q的j结点移为T的i结点
  q[j]=Nil; // 把q的j结点置空
}
void InsertChild(SqBiTree T,TElemType p,int LR,SqBiTree c)
{ //初始条件: 二叉树T存在, p是T中某个结点的值, LR为0或1, 非空二叉树c与T不相交且右子树为空
  //操作结果:根据LR为0或1,插入c为T中p结点的左或右子树。p结点的原有左或右子树则成为c的右子树
  int j,k,i=0;
  for(j=0;j<int(pow(2,BiTreeDepth(T)))-1;j++) // 查找p的序号
    if(T[j]==p) // j为p的序号
      break;
  k=2*j+1+LR; // k为p的左或右孩子的序号
  if(T[k]!=Nil) // p原来的左或右孩子不空
    Move(T,k,T,2*k+2); // 把从T的k结点开始的子树移为从k结点的右子树开始的子树
  Move(c,i,T,k); // 把从c的i结点开始的子树移为从T的k结点开始的子树
}
typedef int QElemType; // 设队列元素类型为整型(序号)
#include "c3-2.h" // 链队列
#include "bo3-2.cpp" // 链队列的基本操作
Status DeleteChild(SqBiTree T,position p,int LR)
{ // 初始条件: 二叉树T存在, p指向T中某个结点, LR为1或0
  // 操作结果: 根据LR为1或0, 删除T中p所指结点的左或右子树
  int i;
  Status k=OK; // 队列不空的标志
  LinkQueue q;
  InitQueue(q); // 初始化队列, 用于存放待删除的结点
  i=(int)pow(2,p.level-1)+p.order-2; // 将层、本层序号转为矩阵的序号
  if(T[i]==Nil) // 此结点空
    return ERROR;
  i=i*2+1+LR; // 待删除子树的根结点在矩阵中的序号
  while(k)
  {
    if(T[2*i+1]!=Nil) // 左结点不空
      EnQueue(q,2*i+1); // 入队左结点的序号
    if(T[2*i+2]!=Nil) // 右结点不空
      EnQueue(q,2*i+2); // 入队右结点的序号
    T[i]=Nil; // 删除此结点
    k=DeQueue(q,i); // 队列不空
  }
  return OK;
}
void(*VisitFunc)(TElemType); // 函数变量
void PreTraverse(SqBiTree T,int e)
{ // PreOrderTraverse()调用
  VisitFunc(T[e]);
  if(T[2*e+1]!=Nil) // 左子树不空
    PreTraverse(T,2*e+1);
  if(T[2*e+2]!=Nil) // 右子树不空
    PreTraverse(T,2*e+2);
}
void PreOrderTraverse(SqBiTree T,void(*Visit)(TElemType))
{ // 初始条件: 二叉树存在, Visit是对结点操作的应用函数
```

```
  // 操作结果：先序遍历T，对每个结点调用函数Visit一次且仅一次
  VisitFunc=Visit;
  if(!BiTreeEmpty(T)) // 树不空
    PreTraverse(T,0);
  cout<<endl;
}
void InTraverse(SqBiTree T,int e)
{ // InOrderTraverse()调用
  if(T[2*e+1]!=Nil) // 左子树不空
    InTraverse(T,2*e+1);
  VisitFunc(T[e]);
  if(T[2*e+2]!=Nil) // 右子树不空
    InTraverse(T,2*e+2);
}
void InOrderTraverse(SqBiTree T,void(*Visit)(TElemType))
{ // 初始条件：二叉树存在，Visit是对结点操作的应用函数
  // 操作结果：中序遍历T，对每个结点调用函数Visit一次且仅一次
  VisitFunc=Visit;
  if(!BiTreeEmpty(T)) // 树不空
    InTraverse(T,0);
  cout<<endl;
}
void PostTraverse(SqBiTree T,int e)
{ // PostOrderTraverse()调用
  if(T[2*e+1]!=Nil) // 左子树不空
    PostTraverse(T,2*e+1);
  if(T[2*e+2]!=Nil) // 右子树不空
    PostTraverse(T,2*e+2);
  VisitFunc(T[e]);
}
void PostOrderTraverse(SqBiTree T,void(*Visit)(TElemType))
{ // 初始条件：二叉树T存在，Visit是对结点操作的应用函数
  // 操作结果：后序遍历T，对每个结点调用函数Visit一次且仅一次
  VisitFunc=Visit;
  if(!BiTreeEmpty(T)) // 树不空
    PostTraverse(T,0);
  cout<<endl;
}
void LevelOrderTraverse(SqBiTree T,void(*Visit)(TElemType))
{ // 层序遍历二叉树
  int i=MAX_TREE_SIZE-1,j;
  while(T[i]==Nil)
    i--; // 找到最后一个非空结点的序号
  for(j=0;j<=i;j++) // 从根结点起，按层序遍历二叉树
    if(T[j]!=Nil)
      Visit(T[j]); // 只遍历非空的结点
  cout<<endl;
}
void Print(SqBiTree T)
{ // 逐层、按本层序号输出二叉树
  int j,k;
  position p;
```

```
  TElemType e;
  for(j=1;j<=BiTreeDepth(T);j++)
  {
    cout<<"第"<<j<<"层: ";
    for(k=1;k<=pow(2,j-1);k++)
    {
      p.level=j;
      p.order=k;
      e=Value(T,p);
      if(e!=Nil)
        cout<<k<<':'<<e<<' ';
    }
    cout<<endl;
  }
}

// main6-1.cpp 检验bo6-1.cpp的主程序，利用条件编译选择数据类型为char或int
//#define CHAR 1 // 字符型
#define CHAR 0 // 整型(二者选一)
#include"c1.h"
#if CHAR
  typedef char TElemType;
  TElemType Nil=' '; // 设字符型以空格符为空
#else
  typedef int TElemType;
  TElemType Nil=0; // 设整型以0为空
#endif
#include"c6-1.h"
#include"bo6-1.cpp"
void visit(TElemType e)
{
  cout<<e<<' ';
}
void main()
{
  Status i;
  int j;
  position p;
  TElemType e;
  SqBiTree T,s;
  InitBiTree(T);
  CreateBiTree(T);
  cout<<"建立二叉树后,树空否? "<<BiTreeEmpty(T)<<"(1:是 0:否) 树的深度="<<BiTreeDepth(T)
  <<endl;
  i=Root(T,e);
  if(i)
    cout<<"二叉树的根为"<<e<<endl;
  else
    cout<<"树空，无根"<<endl;
  cout<<"层序遍历二叉树:"<<endl;
  LevelOrderTraverse(T,visit);
  cout<<"中序遍历二叉树:"<<endl;
```

```
  InOrderTraverse(T,visit);
  cout<<"后序遍历二叉树:"<<endl;
  PostOrderTraverse(T,visit);
  cout<<"请输入待修改结点的层号 本层序号: ";
  cin>>p.level>>p.order;
  e=Value(T,p);
  cout<<"待修改结点的原值为"<<e<<"请输入新值: ";
  cin>>e;
  Assign(T,p,e);
  cout<<"先序遍历二叉树:"<<endl;
  PreOrderTraverse(T,visit);
  cout<<"结点"<<e<<"的双亲为"<<Parent(T,e)<<",左右孩子分别为";
  cout<<LeftChild(T,e)<<","<<RightChild(T,e)<<",左右兄弟分别为";
  cout<<LeftSibling(T,e)<<","<<RightSibling(T,e)<<endl;
  InitBiTree(s);
  cout<<"建立右子树为空的树s:"<<endl;
  CreateBiTree(s);
  cout<<"树s插到树T中,请输入树T中树s的双亲结点 s为左(0)或右(1)子树: ";
  cin>>e>>j;
  InsertChild(T,e,j,s);
  Print(T);
  cout<<"删除子树,请输入待删除子树根结点的层号 本层序号 左(0)或右(1)子树: ";
  cin>>p.level>>p.order>>j;
  DeleteChild(T,p,j);
  Print(T);
  ClearBiTree(T);
  cout<<"清除二叉树后,树空否? "<<BiTreeEmpty(T)<<"(1:是 0:否) 树的深度="<<BiTreeDepth(T)
  <<endl;
  i=Root(T,e);
  if(i)
    cout<<"二叉树的根为"<<e<<endl;
  else
    cout<<"树空, 无根"<<endl;
}
```

程序运行结果:

```
请按层序输入结点的值(整型), 0表示空结点, 输999结束。结点数≤100:
1 2 3 4 5 0 6 7 999↙(见图6-3)
建立二叉树后,树空否? 0(1:是 0:否) 树的深度=4
二叉树的根为1
层序遍历二叉树:
1 2 3 4 5 6 7
中序遍历二叉树:
7 4 2 5 1 3 6
后序遍历二叉树:
7 4 5 2 6 3 1
请输入待修改结点的层号 本层序号: 2 2↙
待修改结点的原值为3请输入新值: 8↙
先序遍历二叉树:
```

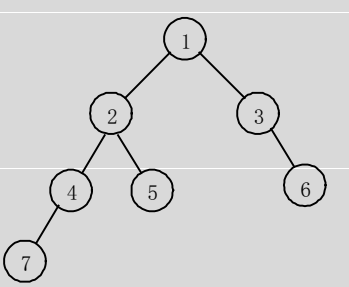


图 6-3　运行 main6-1.cpp 生成的二叉树

```
1 2 4 7 5 8 6
结点8的双亲为1,左右孩子分别为0,6,左右兄弟分别为2,0
建立右子树为空的树s:
请按层序输入结点的值(整型)，0表示空结点，输999结束。结点数≤100:
10 11 0 13 14 0 0 17 999↙(见图6-4)
树s插到树T中,请输入树T中树s的双亲结点 s为左(0)或右(1)子树: 2 1↙(见图6-5)
第1层: 1:1
第2层: 1:2 2:8
第3层: 1:4 2:10 4:6
第4层: 1:7 3:11 4:5
第5层: 5:13 6:14
第6层: 9:17
删除子树,请输入待删除子树根结点的层号 本层序号 左(0)或右(1)子树: 3 2 0↙(见图6-6)
第1层: 1:1
第2层: 1:2 2:8
第3层: 1:4 2:10 4:6
第4层: 1:7 4:5
清除二叉树后,树空否? 1(1:是 0:否) 树的深度=0
树空，无根
```

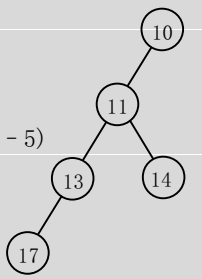


图 6-4　右子树为空的树 s

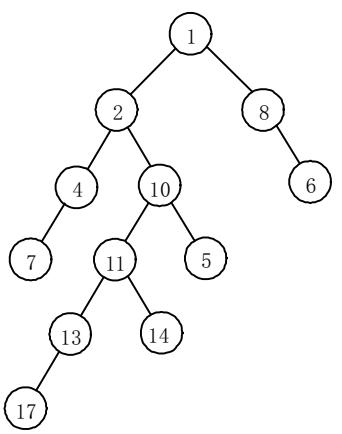


图 6-5　插入树 s 后的二叉树

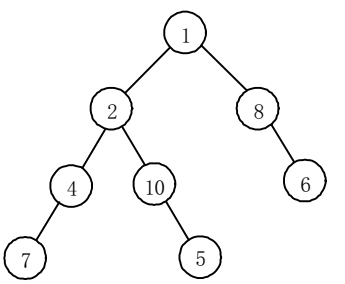


图 6-6　删除子树后的二叉树

```
// c6-2.h 二叉树的二叉链表存储结构(见图6-7)
typedef struct BiTNode
{
  TElemType data;
  BiTNode *lchild,*rchild; // 左右孩子指针
}BiTNode,*BiTree;
```

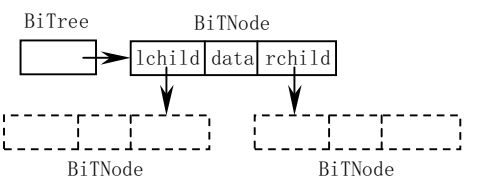


图 6-7　二叉树的二叉链表存储结构

二叉树的二叉链表存储结构删除和插入结点或子树都很灵活。结点动态生成，可充分利用存储空间。图6-8是图6-1(a)所示二叉树的二叉链表存储结构。bo6-2.cpp是二叉链表存储结构的基本操作，其中，调用按先序次序构造二叉链表的函数CreateBiTree()(算法6.4)时，不仅要按先序次序输入结点的值，而且还要把叶子结点的左右孩子指针和度为1的结点的空指针输入。其原因是只根据结点的先序次序还不能惟一确定树的形状。如图6-9所示，三棵树的先序次序都是abc。这样，在调用函数CreateBiTree()时，输入abc就会产生多义性。如果把叶子结点的左右孩子指针和度为1的结点的空指针也按先序输入，则

图6－9(a)输入字符的次序为(以^代替结点的空指针)abc^^^^；图6－9(b)输入字符的次序为ab^^c^^；图6－9(c)输入字符的次序为a^b^c^^。

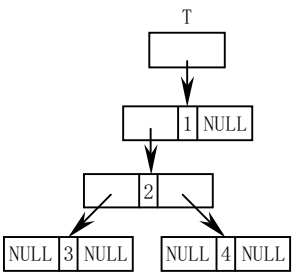


图 6－8　二叉树(见图 6－1(a)所示)的二叉链表存储结构

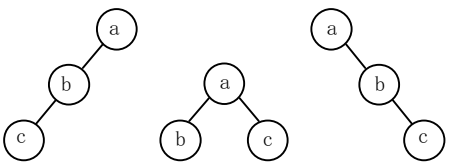


图 6－9　三棵先序遍历都为 abc 的二叉树

bo6-2.cpp 中的许多基本操作都采用了递归函数，因为二叉树的层数是不定的，正确采用递归函数可简化编程。注意到这些递归函数的特点：第 1 是降阶的；第 2 是有出口的。

在 bo6-2.cpp 和 main6-2.cpp 中，采用了编译预处理的“ #define”、“ #ifdef”等命令，通过把 main6-2.cpp 的第 2 行或第 3 行设为注释行，使程序可以在结点类型为整型或字符型的情况下应用。

```
// func6-3.cpp bo6-2.cpp和func9-1.cpp调用
void InitBiTree(BiTree &T)
{ // 操作结果：构造空二叉树T(见图6－10)
  T=NULL;
}
void DestroyBiTree(BiTree &T)
{ // 初始条件：二叉树T存在。操作结果：销毁二叉树T(见图6－10)
  if(T) // 非空树
  {
    if(T->lchild) // 有左孩子
      DestroyBiTree(T->lchild); // 销毁左孩子子树
    if(T->rchild) // 有右孩子
      DestroyBiTree(T->rchild); // 销毁右孩子子树
    free(T); // 释放根结点
    T=NULL; // 空指针赋0
  }
}
void PreOrderTraverse(BiTree T,void(*Visit)(TElemType))
{ // 初始条件：二叉树T存在，Visit是对结点操作的应用函数。算法6.1，有改动
  // 操作结果：先序递归遍历T，对每个结点调用函数Visit一次且仅一次
  if(T) // T不空
  {
    Visit(T->data); // 先访问根结点
    PreOrderTraverse(T->lchild,Visit); // 再先序遍历左子树
    PreOrderTraverse(T->rchild,Visit); // 最后先序遍历右子树
  }
}
```

图 6－10　空的和销毁的二叉树 T

```
void InOrderTraverse(BiTree T,void(*Visit)(TElemType))
{ // 初始条件：二叉树T存在，Visit是对结点操作的应用函数
  // 操作结果：中序递归遍历T，对每个结点调用函数Visit一次且仅一次
  if(T)
  {
    InOrderTraverse(T->lchild,Visit); // 先中序遍历左子树
    Visit(T->data); // 再访问根结点
    InOrderTraverse(T->rchild,Visit); // 最后中序遍历右子树
  }
}


// bo6-2.cpp 二叉树的二叉链表存储(存储结构由c6-2.h定义)的基本操作(22个)，包括算法6.1～6.4
#define ClearBiTree DestroyBiTree // 清空二叉树和销毁二叉树的操作一样
#include"func6-3.cpp"
// 包括InitBiTree()、DestroyBiTree()、PreOrderTraverse()和InOrderTraverse()4函数
void CreateBiTree(BiTree &T)
{ // 算法6.4：按先序次序输入二叉树中结点的值(可为字符型或整型，在主程中定义)，
  // 构造二叉链表表示的二叉树T。变量Nil表示空(子)树。有改动
  TElemType ch;
  scanf(form,&ch);
  if(ch==Nil) // 空
    T=NULL;
  else
  {
    T=(BiTree)malloc(sizeof(BiTNode)); // 生成根结点
    if(!T)
      exit(OVERFLOW);
    T->data=ch;
    CreateBiTree(T->lchild); // 构造左子树
    CreateBiTree(T->rchild); // 构造右子树
  }
}
Status BiTreeEmpty(BiTree T)
{ // 初始条件：二叉树T存在。操作结果：若T为空二叉树，则返回TRUE；否则FALSE
  if(T)
    return FALSE;
  else
    return TRUE;
}
int BiTreeDepth(BiTree T)
{ // 初始条件：二叉树T存在。操作结果：返回T的深度
  int i,j;
  if(!T)
    return 0; // 空树深度为0
  if(T->lchild)
    i=BiTreeDepth(T->lchild); // i为左子树的深度
  else
    i=0;
  if(T->rchild)
    j=BiTreeDepth(T->rchild); // j为右子树的深度
  else
```

```
    j=0;
  return i>j?i+1:j+1; // T的深度为其左右子树的深度中的大者+1
}
TElemType Root(BiTree T)
{ // 初始条件：二叉树T存在。操作结果：返回T的根
  if(BiTreeEmpty(T))
    return Nil;
  else
    return T->data;
}
TElemType Value(BiTree p)
{ // 初始条件：二叉树T存在，p指向T中某个结点。操作结果：返回p所指结点的值
  return p->data;
}
void Assign(BiTree p,TElemType value)
{ // 给p所指结点赋值为value
  p->data=value;
}
typedef BiTree QElemType; // 设队列元素为二叉树的指针类型
#include"c3-2.h" // 链队列
#include"bo3-2.cpp" // 链队列的基本操作
TElemType Parent(BiTree T,TElemType e)
{ // 初始条件：二叉树T存在，e是T中某个结点
  // 操作结果：若e是T的非根结点，则返回它的双亲，否则返回“空”
  LinkQueue q;
  QElemType a;
  if(T) // 非空树
  {
    InitQueue(q); // 初始化队列
    EnQueue(q,T); // 树根指针入队
    while(!QueueEmpty(q)) // 队不空
    {
      DeQueue(q,a); // 出队，队列元素赋给a
      if(a->lchild&&a->lchild->data==e||a->rchild&&a->rchild->data==e)
      // 找到e(是其左或右孩子)
        return a->data; // 返回e的双亲的值
      else // 没找到e，则入队其左右孩子指针(如果非空)
      {
        if(a->lchild)
          EnQueue(q,a->lchild);
        if(a->rchild)
          EnQueue(q,a->rchild);
      }
    }
  }
  return Nil; // 树空或没找到e
}
BiTree Point(BiTree T,TElemType s)
{ // 返回二叉树T中指向元素值为s的结点的指针。另加
  LinkQueue q;
  QElemType a;
```

```
  if(T) // 非空树
  {
    InitQueue(q); // 初始化队列
    EnQueue(q,T); // 根指针入队
    while(!QueueEmpty(q)) // 队不空
    {
      DeQueue(q,a); // 出队，队列元素赋给a
      if(a->data==s)
        return a;
      if(a->lchild) // 有左孩子
        EnQueue(q,a->lchild); // 入队左孩子
      if(a->rchild) // 有右孩子
        EnQueue(q,a->rchild); // 入队右孩子
    }
  }
  return NULL;
}
TElemType LeftChild(BiTree T,TElemType e)
{ // 初始条件：二叉树T存在，e是T中某个结点。操作结果：返回e的左孩子。若e无左孩子,则返回“空”
  BiTree a;
  if(T) // 非空树
  {
    a=Point(T,e); // a是结点e的指针
    if(a&&a->lchild) // T中存在结点e且e存在左孩子
      return a->lchild->data; // 返回e的左孩子的值
  }
  return Nil; // 其余情况返回空
}
TElemType RightChild(BiTree T,TElemType e)
{ // 初始条件：二叉树T存在，e是T中某个结点。操作结果：返回e的右孩子。若e无右孩子,则返回“空”
  BiTree a;
  if(T) // 非空树
  {
    a=Point(T,e); // a是结点e的指针
    if(a&&a->rchild) // T中存在结点e且e存在右孩子
      return a->rchild->data; // 返回e的右孩子的值
  }
  return Nil; // 其余情况返回空
}
TElemType LeftSibling(BiTree T,TElemType e)
{ // 初始条件：二叉树T存在，e是T中某个结点
  // 操作结果：返回e的左兄弟。若e是T的左孩子或无左兄弟，则返回“空”
  TElemType a;
  BiTree p;
  if(T) // 非空树
  {
    a=Parent(T,e); // a为e的双亲
    if(a!=Nil) // 找到e的双亲
    {
      p=Point(T,a); // p为指向结点a的指针
      if(p->lchild&&p->rchild&&p->rchild->data==e) // p存在左右孩子且右孩子是e
```

```
        return p->lchild->data; // 返回p的左孩子(e的左兄弟)
      }
    }
    return Nil; // 其余情况返回空
}
TElemType RightSibling(BiTree T,TElemType e)
{ // 初始条件: 二叉树T存在, e是T中某个结点
  // 操作结果: 返回e的右兄弟。若e是T的右孩子或无右兄弟, 则返回“空”
  TElemType a;
  BiTree p;
  if(T) // 非空树
  {
    a=Parent(T,e); // a为e的双亲
    if(a!=Nil) // 找到e的双亲
    {
      p=Point(T,a); // p为指向结点a的指针
      if(p->lchild&&p->rchild&&p->lchild->data==e) // p存在左右孩子且左孩子是e
        return p->rchild->data; // 返回p的右孩子(e的右兄弟)
    }
  }
  return Nil; // 其余情况返回空
}
Status InsertChild(BiTree p,int LR,BiTree c) // 形参T无用
{ // 初始条件: 二叉树T存在, p指向T中某个结点, LR为0或1, 非空二叉树c与T不相交且右子树为空
  // 操作结果: 根据LR为0或1, 插入c为T中p所指结点的左或右子树。p所指结点的
  //           原有左或右子树则成为c的右子树
  if(p) // p不空
  {
    if(LR==0)
    {
      c->rchild=p->lchild;
      p->lchild=c;
    }
    else // LR==1
    {
      c->rchild=p->rchild;
      p->rchild=c;
    }
    return OK;
  }
  return ERROR; // p空
}
Status DeleteChild(BiTree p,int LR) // 形参T无用
{ // 初始条件: 二叉树T存在, p指向T中某个结点, LR为0或1
  // 操作结果: 根据LR为0或1, 删除T中p所指结点的左或右子树
  if(p) // p不空
  {
    if(LR==0) // 删除左子树
      ClearBiTree(p->lchild);
    else // 删除右子树
      ClearBiTree(p->rchild);
```

```
    return OK;
  }
  return ERROR; // p空
}
typedef BiTree SElemType; // 设栈元素为二叉树的指针类型
#include"c3-1.h" // 顺序栈
#include"bo3-1.cpp" // 顺序栈的基本操作
void InOrderTraverse1(BiTree T,void(*Visit)(TElemType))
{ // 采用二叉链表存储结构，Visit是对数据元素操作的应用函数。算法6.3，有改动
  // 中序遍历二叉树T的非递归算法(利用栈)，对每个数据元素调用函数Visit
  SqStack S;
  InitStack(S);
  while(T||!StackEmpty(S))
  {
    if(T)
    { // 根指针进栈，遍历左子树
      Push(S,T);
      T=T->lchild;
    }
    else
    { // 根指针退栈，访问根结点，遍历右子树
      Pop(S,T);
      Visit(T->data);
      T=T->rchild;
    }
  }
  printf("\n");
}
void InOrderTraverse2(BiTree T,void(*Visit)(TElemType))
{ // 采用二叉链表存储结构，Visit是对数据元素操作的应用函数。算法6.2，有改动
  // 中序遍历二叉树T的非递归算法(利用栈)，对每个数据元素调用函数Visit
  SqStack S;
  BiTree p;
  InitStack(S);
  Push(S,T); // 根指针进栈
  while(!StackEmpty(S))
  {
    while(GetTop(S,p)&&p)
      Push(S,p->lchild); // 向左走到尽头
    Pop(S,p); // 空指针退栈
    if(!StackEmpty(S))
    { // 访问结点，向右一步
      Pop(S,p);
      Visit(p->data);
      Push(S,p->rchild);
    }
  }
  printf("\n");
}
void PostOrderTraverse(BiTree T,void(*Visit)(TElemType))
{ // 初始条件：二叉树T存在，Visit是对结点操作的应用函数
```

```
  // 操作结果：后序递归遍历T，对每个结点调用函数Visit一次且仅一次
  if(T) // T不空
  {
    PostOrderTraverse(T->lchild,Visit); // 先后序遍历左子树
    PostOrderTraverse(T->rchild,Visit); // 再后序遍历右子树
    Visit(T->data); // 最后访问根结点
  }
}
void LevelOrderTraverse(BiTree T,void(*Visit)(TElemType))
{ // 初始条件：二叉树T存在，Visit是对结点操作的应用函数
  // 操作结果：层序递归遍历T(利用队列)，对每个结点调用函数Visit一次且仅一次
  LinkQueue q;
  QElemType a;
  if(T)
  {
    InitQueue(q); // 初始化队列q
    EnQueue(q,T); // 根指针入队
    while(!QueueEmpty(q)) // 队列不空
    {
      DeQueue(q,a); // 出队元素(指针),赋给a
      Visit(a->data); // 访问a所指结点
      if(a->lchild!=NULL) // a有左孩子
        EnQueue(q,a->lchild); // 入队a的左孩子
      if(a->rchild!=NULL) // a有右孩子
        EnQueue(q,a->rchild); // 入队a的右孩子
    }
    printf("\n");
  }
}

// main6-2.cpp 检验bo6-2.cpp的主程序，利用条件编译选择数据类型(另一种方法)
#define CHAR // 字符型
// #define INT // 整型(二者选一)
#include"c1.h"
#ifdef CHAR
  typedef char TElemType;
  TElemType Nil=' '; // 字符型以空格符为空
  #define form "%c" // 输入输出的格式为%c
#endif
#ifdef INT
  typedef int TElemType;
  TElemType Nil=0; // 整型以0为空
  #define form "%d" // 输入输出的格式为%d
#endif
#include"c6-2.h"
#include"bo6-2.cpp"
void visitT(TElemType e)
{
  printf(form" ",e);
}
void main()
```

```
{
  int i;
  BiTree T,p,c;
  TElemType e1,e2;
  InitBiTree(T);
  printf("构造空二叉树后,树空否? %d(1:是 0:否)树的深度=%d\n",BiTreeEmpty(T),BiTreeDepth(T));
  e1=Root(T);
  if(e1!=Nil)
    printf("二叉树的根为 "form"\n",e1);
  else
    printf("树空, 无根\n");
#ifdef CHAR
  printf("请先序输入二叉树(如:ab三个空格表示a为根结点,b为左子树的二叉树)\n");
#endif
#ifdef INT
  printf("请先序输入二叉树(如:1 2 0 0 0表示1为根结点,2为左子树的二叉树)\n");
#endif
  CreateBiTree(T);
  printf("建立二叉树后,树空否? %d(1:是 0:否) 树的深度=%d\n",BiTreeEmpty(T),BiTreeDepth(T));
  e1=Root(T);
  if(e1!=Nil)
    printf("二叉树的根为 "form"\n",e1);
  else
    printf("树空, 无根\n");
  printf("中序递归遍历二叉树:\n");
  InOrderTraverse(T,visitT);
  printf("\n后序递归遍历二叉树:\n");
  PostOrderTraverse(T,visitT);
  printf("\n请输入一个结点的值: ");
  scanf("%*c"form,&e1);
  p=Point(T,e1); // p为e1的指针
  printf("结点的值为"form"\n",Value(p));
  printf("欲改变此结点的值, 请输入新值: ");
  scanf("%*c"form"%*c",&e2); // 后一个%*c吃掉回车符, 为调用CreateBiTree()做准备
  Assign(p,e2);
  printf("层序遍历二叉树:\n");
  LevelOrderTraverse(T,visitT);
  e1=Parent(T,e2);
  if(e1!=Nil)
    printf("%c的双亲是"form"\n",e2,e1);
  else
    printf(form"没有双亲\n",e2);
  e1=LeftChild(T,e2);
  if(e1!=Nil)
    printf(form"的左孩子是"form"\n",e2,e1);
  else
    printf(form"没有左孩子\n",e2);
  e1=RightChild(T,e2);
  if(e1!=Nil)
    printf(form"的右孩子是"form"\n",e2,e1);
  else
```

```
    printf(form"没有右孩子\n",e2);
  e1=LeftSibling(T,e2);
  if(e1!=Nil)
    printf(form"的左兄弟是"form"\n",e2,e1);
  else
    printf(form"没有左兄弟\n",e2);
  e1=RightSibling(T,e2);
  if(e1!=Nil)
    printf(form"的右兄弟是"form"\n",e2,e1);
  else
    printf(form"没有右兄弟\n",e2);
  InitBiTree(c);
  printf("构造一个右子树为空的二叉树c:\n");
#ifdef CHAR
  printf("请先序输入二叉树(如:ab三个空格表示a为根结点,b为左子树的二叉树)\n");
#endif
#ifdef INT
  printf("请先序输入二叉树(如:1 2 0 0 0表示1为根结点,2为左子树的二叉树)\n");
#endif
  CreateBiTree(c);
  printf("先序递归遍历二叉树c:\n");
  PreOrderTraverse(c,visitT);
  printf("\n层序遍历二叉树c:\n");
  LevelOrderTraverse(c,visitT);
  printf("树c插到树T中,请输入树T中树c的双亲结点 c为左(0)或右(1)子树: ");
  scanf("%*c"form"%d",&e1,&i);
  p=Point(T,e1); // p是T中树c的双亲结点指针
  InsertChild(p,i,c);
  printf("先序递归遍历二叉树:\n");
  PreOrderTraverse(T,visitT);
  printf("\n中序非递归遍历二叉树:\n");
  InOrderTraverse1(T,visitT);
  printf("删除子树,请输入待删除子树的双亲结点  左(0)或右(1)子树: ");
  scanf("%*c"form"%d",&e1,&i);
  p=Point(T,e1);
  DeleteChild(p,i);
  printf("先序递归遍历二叉树:\n");
  PreOrderTraverse(T,visitT);
  printf("\n中序非递归遍历二叉树(另一种方法):\n");
  InOrderTraverse2(T,visitT);
  DestroyBiTree(T);
}
```

程序运行结果:

```
构造空二叉树后,树空否? 1(1:是 0:否)树的深度=0
树空, 无根
请先序输入二叉树(如:ab三个空格表示a为根结点,b为左子树的二叉树)
```

```
abdg   e  c f  ↙(见图6－11)
建立二叉树后,树空否? 0(1:是 0:否) 树的深度=4
二叉树的根为 a
中序递归遍历二叉树:
g d b e a c f
后序递归遍历二叉树:
g d e b f c a
请输入一个结点的值: d↙
结点的值为d
欲改变此结点的值, 请输入新值: m↙
层序遍历二叉树:
a b c m e f g
m的双亲是b
m的左孩子是g
m没有右孩子
m没有左兄弟
m的右兄弟是e
构造一个右子树为空的二叉树c:
请先序输入二叉树(如:ab三个空格表示a为根结点,b为左子树的二叉树)
hijl   k   ↙(见图6－12)
先序递归遍历二叉树c:
h i j l k
层序遍历二叉树c:
h i j k l
树c插到树T中,请输入树T中树c的双亲结点 c为左(0)或右(1)子树: b 1↙
先序递归遍历二叉树: (见图6－13)
a b m g h i j l k e c f
中序非递归遍历二叉树:
g m b l j i k h e a c f
删除子树,请输入待删除子树的双亲结点  左(0)或右(1)子树: h 0↙
先序递归遍历二叉树: (见图6－14)
a b m g h e c f
中序非递归遍历二叉树(另一种方法):
g m b h e a c f
```

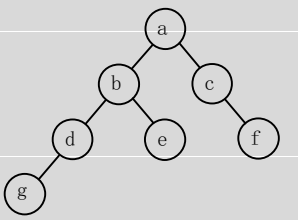


图 6－11　运行 main6-2.cpp 生成的二叉树

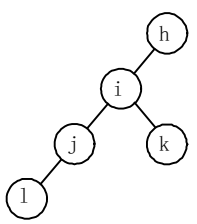


图 6－12　右子树为空的树 c

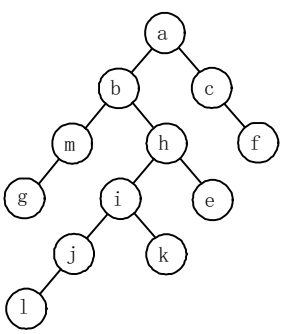


图 6－13　树 c 插到树 T 中作为结点 b 的右子树

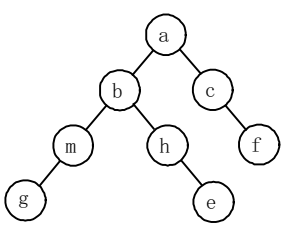


图 6－14　删除 h 的左子树后的二叉树

```
// c6-6.h 二叉树的三叉链表存储结构(见图6－15)
typedef struct BiTPNode
{
```

```
   TElemType data;
   BiTPNode *parent,*lchild,*rchild; // 双亲、左右孩子指针
 }BiTPNode,*BiPTree;
```

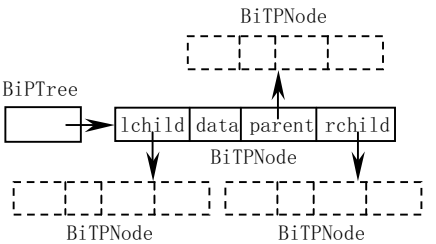


图 6－15　二叉树的三叉链表存储结构

二叉树的三叉链表存储结构比二叉链表多一个指向双亲结点的指针，因此，求双亲和左右兄弟都很容易。但在构造二叉树时要另给双亲指针赋值，从而增加了复杂度。由于三叉链表和二叉链表在结构上的相似性，它们有些相应的基本操作也很类似。图 6－16 是图 6－1(a)所示二叉树的三叉链表存储结构。bo6-6.cpp 是三叉链表存储结构的基本操作。

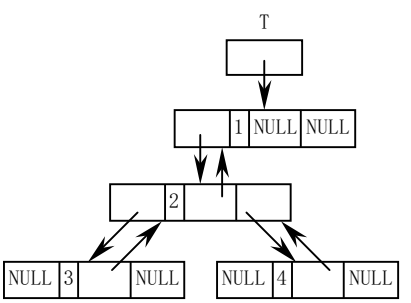


图 6－16　二叉树(见图 6－1(a)所示)的三叉链表存储结构

```
// bo6-6.cpp 二叉树的三叉链表存储(存储结构由c6-6.h定义)的基本操作(21个)
#define ClearBiTree DestroyBiTree // 清空二叉树和销毁二叉树的操作一样
void InitBiTree(BiPTree &T)
{ // 操作结果：构造空二叉树T
  T=NULL;
}
void DestroyBiTree(BiPTree &T)
{ // 初始条件：二叉树T存在。操作结果：销毁二叉树T
  if(T) // 非空树
  {
    if(T->lchild) // 有左孩子
      DestroyBiTree(T->lchild); // 销毁左孩子子树
    if(T->rchild) // 有右孩子
      DestroyBiTree(T->rchild); // 销毁右孩子子树
    free(T); // 释放根结点
    T=NULL; // 空指针赋0
  }
}
void CreateBiTree(BiPTree &T)
{ // 按先序次序输入二叉树中结点的值(可为字符型或整型，在主程中定义)，
```

```
  // 构造三叉链表表示的二叉树T
  TElemType ch;
  scanf(form,&ch);
  if(ch==Nil) // 空
    T=NULL;
  else
  {
    T=(BiPTree)malloc(sizeof(BiTPNode)); // 动态生成根结点
    if(!T)
      exit(OVERFLOW);
    T->data=ch;  // 给根结点赋值
    T->parent=NULL; // 根结点无双亲
    CreateBiTree(T->lchild); // 构造左子树
    if(T->lchild) // 有左孩子
      T->lchild->parent=T; // 给左孩子的双亲域赋值
    CreateBiTree(T->rchild); // 构造右子树
    if(T->rchild) // 有右孩子
      T->rchild->parent=T; // 给右孩子的双亲域赋值
  }
}
Status BiTreeEmpty(BiPTree T)
{ // 初始条件：二叉树T存在。操作结果：若T为空二叉树，则返回TRUE；否则FALSE
  if(T)
    return FALSE;
  else
    return TRUE;
}
int BiTreeDepth(BiPTree T)
{ // 初始条件：二叉树T存在。操作结果：返回T的深度
  int i,j;
  if(!T)
    return 0; // 空树深度为0
  if(T->lchild)
    i=BiTreeDepth(T->lchild); // i为左子树的深度
  else
    i=0;
  if(T->rchild)
    j=BiTreeDepth(T->rchild); // j为右子树的深度
  else
    j=0;
  return i>j?i+1:j+1; // T的深度为其左右子树的深度中的大者+1
}
TElemType Root(BiPTree T)
{ // 初始条件：二叉树T存在。操作结果：返回T的根
  if(T)
    return T->data;
  else
    return Nil;
}
TElemType Value(BiPTree p)
{ // 初始条件：二叉树T存在，p指向T中某个结点。操作结果：返回p所指结点的值
```

```
  return p->data;
}
void Assign(BiPTree p,TElemType value)
{ // 给p所指结点赋值为value
  p->data=value;
}
typedef BiPTree QElemType; // 设队列元素为二叉树的指针类型
#include"c3-2.h" // 链队列
#include"bo3-2.cpp" // 链队列的基本操作
BiPTree Point(BiPTree T,TElemType e)
{ // 返回二叉树T中指向元素值为e的结点的指针。加
  LinkQueue q;
  QElemType a;
  if(T) // 非空树
  {
    InitQueue(q); // 初始化队列
    EnQueue(q,T); // 根结点入队
    while(!QueueEmpty(q)) // 队不空
    {
      DeQueue(q,a); // 出队，队列元素赋给a
      if(a->data==e)
        return a;
      if(a->lchild) // 有左孩子
        EnQueue(q,a->lchild); // 入队左孩子
      if(a->rchild) // 有右孩子
        EnQueue(q,a->rchild); // 入队右孩子
    }
  }
  return NULL;
}
TElemType Parent(BiPTree T,TElemType e)
{ // 初始条件：二叉树T存在，e是T中某个结点
  // 操作结果：若e是T的非根结点，则返回它的双亲；否则返回“空”
  BiPTree a;
  if(T) // 非空树
  {
    a=Point(T,e); // a是结点e的指针
    if(a&&a!=T) // T中存在结点e且e是非根结点
      return a->parent->data; // 返回e的双亲的值
  }
  return Nil; // 其余情况返回空
}
TElemType LeftChild(BiPTree T,TElemType e)
{ // 初始条件：二叉树T存在，e是T中某个结点。操作结果：返回e的左孩子。若e无左孩子,则返回“空”
  BiPTree a;
  if(T) // 非空树
  {
    a=Point(T,e); // a是结点e的指针
    if(a&&a->lchild) // T中存在结点e且e存在左孩子
      return a->lchild->data; // 返回e的左孩子的值
  }
```

```
  return Nil; // 其余情况返回空
}
TElemType RightChild(BiPTree T,TElemType e)
{ // 初始条件：二叉树T存在，e是T中某个结点。操作结果：返回e的右孩子。若e无右孩子,则返回“空”
  BiPTree a;
  if(T) // 非空树
  {
    a=Point(T,e); // a是结点e的指针
    if(a&&a->rchild) // T中存在结点e且e存在右孩子
      return a->rchild->data; // 返回e的右孩子的值
  }
  return Nil; // 其余情况返回空
}
TElemType LeftSibling(BiPTree T,TElemType e)
{ // 初始条件：二叉树T存在，e是T中某个结点
  // 操作结果：返回e的左兄弟。若e是T的左孩子或无左兄弟，则返回“空”
  BiPTree a;
  if(T) // 非空树
  {
    a=Point(T,e); // a是结点e的指针
    if(a&&a!=T&&a->parent->lchild&&a->parent->lchild!=a) // T中存在结点e且e存在左兄弟
      return a->parent->lchild->data; // 返回e的左兄弟的值
  }
  return Nil; // 其余情况返回空
}
TElemType RightSibling(BiPTree T,TElemType e)
{ // 初始条件：二叉树T存在，e是T中某个结点
  // 操作结果：返回e的右兄弟。若e是T的右孩子或无右兄弟，则返回“空”
  BiPTree a;
  if(T) // 非空树
  {
    a=Point(T,e); // a是结点e的指针
    if(a&&a!=T&&a->parent->rchild&&a->parent->rchild!=a) // T中存在结点e且e存在右兄弟
      return a->parent->rchild->data; // 返回e的右兄弟的值
  }
  return Nil; // 其余情况返回空
}
Status InsertChild(BiPTree p,int LR,BiPTree c) // 形参T无用
{ // 初始条件：二叉树T存在，p指向T中某个结点，LR为0或1，非空二叉树c与T不相交且右子树为空
  // 操作结果：根据LR为0或1，插入c为T中p所指结点的左或右子树。p所指结点
  //           的原有左或右子树则成为c的右子树
  if(p) // p不空
  {
    if(LR==0)
    {
      c->rchild=p->lchild;
      if(c->rchild) // c有右孩子(p原有左孩子)
        c->rchild->parent=c;
      p->lchild=c;
      c->parent=p;
    }
```

```
    else // LR==1
    {
      c->rchild=p->rchild;
      if(c->rchild) // c有右孩子(p原有右孩子)
        c->rchild->parent=c;
      p->rchild=c;
      c->parent=p;
    }
    return OK;
  }
  return ERROR; // p空
}
Status DeleteChild(BiPTree p,int LR) // 形参T无用
{ // 初始条件: 二叉树T存在, p指向T中某个结点, LR为0或1
  // 操作结果: 根据LR为0或1, 删除T中p所指结点的左或右子树
  if(p) // p不空
  {
    if(LR==0) // 删除左子树
      ClearBiTree(p->lchild);
    else // 删除右子树
      ClearBiTree(p->rchild);
    return OK;
  }
  return ERROR; // p空
}
void PreOrderTraverse(BiPTree T,void(*Visit)(BiPTree))
{ // 先序递归遍历二叉树T
  if(T)
  {
    Visit(T); // 先访问根结点
    PreOrderTraverse(T->lchild,Visit); // 再先序遍历左子树
    PreOrderTraverse(T->rchild,Visit); // 最后先序遍历右子树
  }
}
void InOrderTraverse(BiPTree T,void(*Visit)(BiPTree))
{ // 中序递归遍历二叉树T
  if(T)
  {
    InOrderTraverse(T->lchild,Visit); // 中序遍历左子树
    Visit(T); // 再访问根结点
    InOrderTraverse(T->rchild,Visit); // 最后中序遍历右子树
  }
}
void PostOrderTraverse(BiPTree T,void(*Visit)(BiPTree))
{ // 后序递归遍历二叉树T
  if(T)
  {
    PostOrderTraverse(T->lchild,Visit); // 后序遍历左子树
    PostOrderTraverse(T->rchild,Visit); // 后序遍历右子树
    Visit(T); // 最后访问根结点
  }
```

```
}
void LevelOrderTraverse(BiPTree T,void(*Visit)(BiPTree))
{ // 层序遍历二叉树T(利用队列)
  LinkQueue q;
  QElemType a;
  if(T)
  {
    InitQueue(q);
    EnQueue(q,T);
    while(!QueueEmpty(q))
    {
      DeQueue(q,a);
      Visit(a);
      if(a->lchild!=NULL)
        EnQueue(q,a->lchild);
      if(a->rchild!=NULL)
        EnQueue(q,a->rchild);
    }
  }
}

// main6-6.cpp 检验bo6-6.cpp的主程序
#define CHAR // 字符型
// #define INT // 整型(二者选一)
#ifdef CHAR
  typedef char TElemType;
  TElemType Nil=' '; // 字符型以空格符为空
  #define form "%c" // 输入输出的格式为%c
#endif
#ifdef INT
  typedef int TElemType;
  TElemType Nil=0; // 整型以0为空
  #define form "%d" // 输入输出的格式为%d
#endif
#include"c1.h"
#include"c6-6.h"
#include"bo6-6.cpp"
void visitT(BiPTree T)
{
  if(T) // T非空
    printf(form" ",T->data);
}
void main()
{
  int i;
  BiPTree T,c,q;
  TElemType e1,e2;
  InitBiTree(T);
  printf("构造空二叉树后,树空否? %d(1:是 0:否)树的深度=%d\n",BiTreeEmpty(T),BiTreeDepth(T));
  e1=Root(T);
  if(e1!=Nil)
```

```
    printf("二叉树的根为 "form"\n",e1);
  else
    printf("树空, 无根\n");
#ifdef CHAR
  printf("请按先序输入二叉树(如:ab三个空格表示a为根结点,b为左子树的二叉树)\n");
#endif
#ifdef INT
  printf("请按先序输入二叉树(如:1 2 0 0 0表示1为根结点,2为左子树的二叉树)\n");
#endif
  CreateBiTree(T);
  printf("建立二叉树后,树空否? %d(1:是 0:否) 树的深度=%d\n",BiTreeEmpty(T),BiTreeDepth(T));
  e1=Root(T);
  if(e1!=Nil)
    printf("二叉树的根为 "form"\n",e1);
  else
    printf("树空, 无根\n");
  printf("中序递归遍历二叉树:\n");
  InOrderTraverse(T,visitT);
  printf("\n后序递归遍历二叉树:\n");
  PostOrderTraverse(T,visitT);
  printf("\n层序遍历二叉树:\n");
  LevelOrderTraverse(T,visitT);
  printf("\n请输入一个结点的值: ");
  scanf("%*c"form,&e1);
  c=Point(T,e1); // c为e1的指针
  printf("结点的值为"form"\n",Value(c));
  printf("欲改变此结点的值, 请输入新值: ");
  scanf("%*c"form"%*c",&e2);
  Assign(c,e2);
  printf("层序遍历二叉树:\n");
  LevelOrderTraverse(T,visitT);
  e1=Parent(T,e2);
  if(e1!=Nil)
    printf("\n"form"的双亲是"form"\n",e2,e1);
  else
    printf(form"没有双亲\n",e2);
  e1=LeftChild(T,e2);
  if(e1!=Nil)
    printf(form"的左孩子是"form"\n",e2,e1);
  else
    printf(form"没有左孩子\n",e2);
  e1=RightChild(T,e2);
  if(e1!=Nil)
    printf(form"的右孩子是"form"\n",e2,e1);
  else
    printf(form"没有右孩子\n",e2);
  e1=LeftSibling(T,e2);
  if(e1!=Nil)
    printf(form"的左兄弟是"form"\n",e2,e1);
  else
    printf(form"没有左兄弟\n",e2);
```

```
  e1=RightSibling(T,e2);
  if(e1!=Nil)
    printf(form"的右兄弟是"form"\n",e2,e1);
  else
    printf(form"没有右兄弟\n",e2);
  InitBiTree(c);
  printf("构造一个右子树为空的二叉树c:\n");
#ifdef CHAR
  printf("请先序输入二叉树(如:ab三个空格表示a为根结点,b为左子树的二叉树)\n");
#endif
#ifdef INT
  printf("请先序输入二叉树(如:1 2 0 0 0表示1为根结点,2为左子树的二叉树)\n");
#endif
  CreateBiTree(c);
  printf("先序递归遍历二叉树c:\n");
  PreOrderTraverse(c,visitT);
  printf("\n树c插到树T中,请输入树T中树c的双亲结点 c为左(0)或右(1)子树: ");
  scanf("%*c"form"%d",&e1,&i);
  q=Point(T,e1);
  InsertChild(q,i,c);
  printf("先序递归遍历二叉树:\n");
  PreOrderTraverse(T,visitT);
  printf("\n删除子树,请输入待删除子树的双亲结点  左(0)或右(1)子树: ");
  scanf("%*c"form"%d",&e1,&i);
  q=Point(T,e1);
  DeleteChild(q,i);
  printf("先序递归遍历二叉树:\n");
  PreOrderTraverse(T,visitT);
  printf("\n");
  DestroyBiTree(T);
}
```

程序运行结果:

```
构造空二叉树后,树空否? 1(1:是 0:否)树的深度=0
树空, 无根
请按先序输入二叉树(如:ab三个空格表示a为根结点,b为左子树的二叉树)
abdg   e  c f  ↙(见图6-11)
建立二叉树后,树空否? 0(1:是 0:否) 树的深度=4
二叉树的根为a
中序递归遍历二叉树:
g d b e a c f
后序递归遍历二叉树:
g d e b f c a
层序遍历二叉树:
a b c d e f g
请输入一个结点的值: d↙
结点的值为d
欲改变此结点的值, 请输入新值: m↙
```

```
层序遍历二叉树:
a b c m e f g
m的双亲是b
m的左孩子是g
m没有右孩子
m没有左兄弟
m的右兄弟是e
构造一个右子树为空的二叉树c:
请先序输入二叉树(如:ab三个空格表示a为根结点,b为左子树的二叉树)
hijl  k  ↙(见图6-12)
先序递归遍历二叉树c:
h i j l k
树c插到树T中,请输入树T中树c的双亲结点 c为左(0)或右(1)子树: b 1↙(见图6-13)
先序递归遍历二叉树:
a b m g h i j l k e c f
删除子树,请输入待删除子树的双亲结点  左(0)或右(1)子树: h 0↙(见图6-14)
先序递归遍历二叉树:
a b m g h e c f
```

6.3　遍历二叉树和线索二叉树

6.3.1　遍历二叉树

遍历二叉树就是按某种规则，对二叉树的每个结点均访问一次，而且仅访问一次。这实际上就是将非线性的二叉树结构线性化。遍历二叉树的方法有先序、中序、后序和层序4种，访问的顺序各不相同。以图6-1(a)所示二叉树为例，先序遍历的顺序为1 2 3 4；中序遍历的顺序为3 2 4 1；后序遍历的顺序为3 4 2 1；层序遍历的顺序为1 2 3 4。对于这棵二叉树，层序遍历和先序遍历的顺序碰巧一致。有关二叉链表存储结构生成二叉树和遍历二叉树的算法6.1～6.4在bo6-2.cpp中。

6.3.2　线索二叉树

为了更方便、快捷地遍历二叉树，最好在二叉树的结点上增加2个指针，它们分别指向遍历二叉树时该结点的前驱和后继结点。这样，从二叉树的任一结点都可以方便地找到其它结点。但这样做大大降低了结构的存储密度。另外，根据二叉树的性质3(教科书124页)，有：$n_0=n_2+1$。空链域$=2n_0+n_1$(叶子结点有2个空链域，度为1的结点有1个空链域)$=n_0+n_1+n_2+1=n+1$。也就是说，在由n个结点组成的二叉树中，有n+1个指针是空指针。如果能利用这n+1个空指针，使它们指向结点的前驱(当左孩子指针空)或后继(当右孩子指针空)，则既可不降低结构的存储密度，又可更方便、快捷地遍历二叉树。不过，这样就无法区别左右孩子指针所指的到底是结点的左右孩子，还是结点的前驱后继了。为了有所区别，另增加2个域LTag和RTag。当所指的是孩子，其值为0(Link)；当所指的是前驱后继，其值为1(Thread)。这样做，结构的存储密度也有所降低，但不大。因为LTag和RTag分别只需要1个比特(二进制位)即可。c6-3.h是二叉树的二叉线索存储结

构。它只是比二叉链表存储结构(在 c6-2.h 中)多了 LTag 和 RTag 个域。

```
// c6-3.h 二叉树的二叉线索存储结构(见图6-17)
enum PointerTag // 枚举
{Link,Thread}; // Link(0): 指针, Thread(1): 线索
struct BiThrNode
{
  TElemType data;
  BiThrNode *lchild,*rchild; // 左右孩子指针
  PointerTag LTag,RTag; // 左右标志
};
typedef BiThrNode *BiThrTree;
```

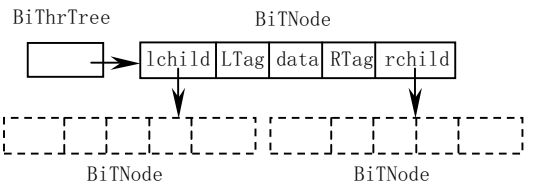


图 6-17　二叉树的二叉线索存储结构

构造线索二叉树的方法和构造以二叉链表存储的二叉树方法相似，都是按先序输入结点的值来构造二叉树的。对比 bo6-2.cpp 中的 CreateBiTree()函数和 bo6-3.cpp 中的 CreateBiThrTree()函数可见，它们的区别有两点：

(1) 二叉树结点的结构不同；

(2) 构造线索二叉树时，若有左右孩子结点，还要给左右标志赋值 0(Link)。

图 6-1(a)所示二叉树调用 CreateBiThrTree()函数产生的二叉树结构如图 6-18 所示。和调用 CreateBiTree()函数产生的二叉树结构(见图 6-8)相比，前者只是多了 LTag 和 RTag 两个域。并且当其相应孩子指针不空时，赋值 0。

调用 CreateBiThrTree()函数，只是构造了一棵可以线索化的二叉树，还没有完成线索化。

因为对于一棵给定的二叉树，其先序、中序、后序和层序遍历的顺序是不同的。显然，其线索化的操作和遍历的操作也是不同的。

图 6-19 是图 6-1(a)所示二叉树的中序线索二叉树存储结构示例。和二叉链表(见图 6-8)存储结构相比，第一，它多了一个头结点。其左孩子指针指向根结点，右孩子指针(线索)指向中序遍历所访问的最后一个结点。第二，它每个结点的左右孩子指针都不是空指针。在没有孩子的情况下，分别指向该结点中序遍历的前驱或后继。第三，中序遍历的第 1 个结点(最左边的结点，它没有左孩子，本例是结点 3)的左孩子指针(线索)和最后 1 个结点(最右边的结点，它没有右孩子，本例是结点 1)的右孩子指针(线索)都指向头结点。其目的是标志遍历的起点和终点。

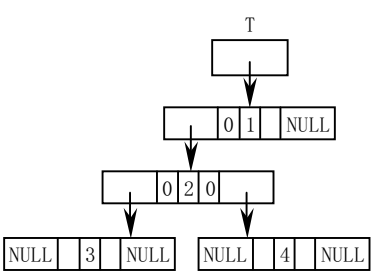


图 6-18　CreateBiThrTree()产生的二叉树(以图 6-1(a)为例)

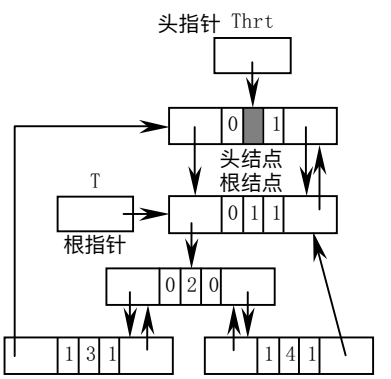


图 6-19　中序线索二叉树(以图 6-1(a)为例)存储结构

图 6 - 20 是空线索二叉树。

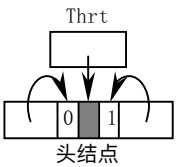


图 6 - 20　空线索二叉树

bo6-3. cpp 中的 InThreading() 函数和 InOrderThreading() 函数共同完成了对二叉树的中序遍历线索化。其算法是：设置全局指针变量 pre(之所以设为全局变量，是因为在递归函数 InThreading() 和 InOrderThreading() 中都要用到，设为全局变量就不必频繁传递变量的值)，令 pre 总是指向遍历的前驱结点，p 指向当前结点；在中序遍历过程中，如果 p 所指结点没有左孩子，则结点的左孩子指针指向 pre 所指结点，结点的 LTag 域的值为 1(Thread)；如果 pre 所指结点没有右孩子，则结点的右孩子指针指向 p，结点的 RTag 域的值为 1(Thread)。

对于图 6 - 19 所示的中序线索二叉树，我们能不能在找到遍历的第 1 个结点后，顺着右孩子指针一直找到遍历的最后 1 个结点呢？这是不一定的。因为结点的右孩子指针并不一定指向后继结点，它可能指向右孩子，而右孩子并不一定恰好是后继结点。

bo6-3. cpp 中的 InOrderTraverse_Thr() 函数完成了对中序线索二叉树的中序遍历操作。其算法是：当树不空时，由树根向左找，一直找到没有左孩子的结点(最左结点)。这就是中序遍历的第 1 个结点。若该结点没有右孩子，则右孩子指针指向其后继结点；否则，以其右孩子为子树的根，向左找，一直找到没有左孩子的结点。这就是后继结点。当结点的右孩子指针指向头结点，遍历结束。

图 6 - 21 是图 6 - 1(a)所示二叉树的先序线索二叉树存储结构示例。bo6-3. cpp 中的先序线索化的递归函数 PreThreading() 与中序线索化的递归函数 InThreading() 很相像，都是利用递归进行线索化，只不过顺序不同。但由于 PreThreading() 是先序线索化，所以判断结点是否有左右孩子就不能由其左右孩子指针是否为空决定，而要由结点的 LTag 和 RTag 域是否为 0(Link)来决定。

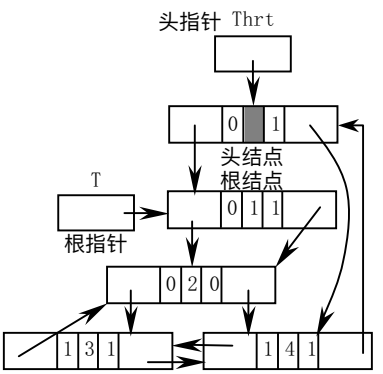


图 6 - 21　先序线索二叉树(以图 6 - 1(a)为例)存储结构

对于先序线索化二叉树的先序遍历算法是这样的：根结点是遍历的第 1 个结点；如果结点有左孩子，则左孩子是其后继；若结点没有左孩子，则右孩子指针所指的结点是其后继(无论该结点有没有右孩子)。相关程序见 bo6-3. cpp 中的 PreOrderTraverse_Thr() 函数。

bo6-3. cpp 中的后序线索化的递归函数 PostThreading() 与中序线索化的递归函数 InThreading() 也很相像，也是利用递归进行线索化，也只是顺序不同。图 6 - 22 是图 6 - 1(a)所示二叉树的后序线索二叉树存储结构示例。对于后序线索化二叉树的后序遍历

算法较复杂。因为根结点是在最后遍历，所以要采用带有双亲指针的三叉链表结构才行。本书没有给出它的算法。

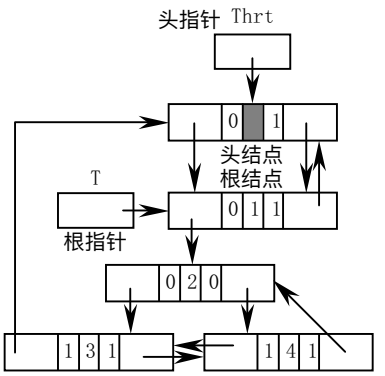


图 6－22　后序线索二叉树(以图 6－1(a)为例)存储结构

```
// bo6-3.cpp 二叉树的二叉线索存储(存储结构由c6-3.h定义)的基本操作，包括算法6.5～6.7
void CreateBiThrTree(BiThrTree &T) // (见图6－18)
{ // 按先序输入线索二叉树中结点的值，构造线索二叉树T。0(整型)/空格(字符型)表示空结点
  TElemType ch;
  scanf(form,&ch);
  if(ch==Nil)
    T=NULL;
  else
  {
    T=(BiThrTree)malloc(sizeof(BiThrNode)); // 生成根结点(先序)
    if(!T)
      exit(OVERFLOW);
    T->data=ch; // 给根结点赋植
    CreateBiThrTree(T->lchild); // 递归构造左子树
    if(T->lchild) // 有左孩子
      T->LTag=Link; // 给左标志赋值(指针)
    CreateBiThrTree(T->rchild); // 递归构造右子树
    if(T->rchild) // 有右孩子
      T->RTag=Link; // 给右标志赋值(指针)
  }
}
BiThrTree pre; // 全局变量，始终指向刚刚访问过的结点
void InThreading(BiThrTree p)
{ // 通过中序遍历进行中序线索化，线索化之后pre指向最后一个结点。算法6.7
  if(p) // 线索二叉树不空
  {
    InThreading(p->lchild); // 递归左子树线索化
    if(!p->lchild) // 没有左孩子
    {
      p->LTag=Thread; // 左标志为线索(前驱)
      p->lchild=pre; // 左孩子指针指向前驱
    }
    if(!pre->rchild) // 前驱没有右孩子
    {
```

```
      pre->RTag=Thread; // 前驱的右标志为线索(后继)
      pre->rchild=p; // 前驱右孩子指针指向其后继(当前结点p)
    }
    pre=p; // 保持pre指向p的前驱
    InThreading(p->rchild); // 递归右子树线索化
  }
}
void InOrderThreading(BiThrTree &Thrt,BiThrTree T)
{ // 中序遍历二叉树T，并将其中序线索化，Thrt指向头结点。算法6.6
  if(!(Thrt=(BiThrTree)malloc(sizeof(BiThrNode)))) // 生成头结点不成功
    exit(OVERFLOW);
  Thrt->LTag=Link; // 建头结点，左标志为指针
  Thrt->RTag=Thread; // 右标志为线索
  Thrt->rchild=Thrt; // 右指针回指
  if(!T) // 若二叉树空，则左指针回指(见图6－20)
    Thrt->lchild=Thrt;
  else // (见图6－19)
  {
    Thrt->lchild=T; // 头结点的左指针指向根结点
    pre=Thrt; // pre(前驱)的初值指向头结点
    InThreading(T); // 中序遍历进行中序线索化，pre指向中序遍历的最后一个结点
    pre->rchild=Thrt; // 最后一个结点的右指针指向头结点
    pre->RTag=Thread; // 最后一个结点的右标志为线索
    Thrt->rchild=pre; // 头结点的右指针指向中序遍历的最后一个结点
  }
}
void InOrderTraverse_Thr(BiThrTree T,void(*Visit)(TElemType))
{ // 中序遍历线索二叉树T(头结点)的非递归算法。算法6.5
  BiThrTree p;
  p=T->lchild; // p指向根结点
  while(p!=T)
  { // 空树或遍历结束时，p==T
    while(p->LTag==Link) // 由根结点一直找到二叉树的最左结点
      p=p->lchild;
    Visit(p->data); // 访问此结点
    while(p->RTag==Thread&&p->rchild!=T) // p->rchild是线索(后继)，且不是遍历的最后一个结点
    {
      p=p->rchild;
      Visit(p->data); // 访问后继结点
    }
    p=p->rchild; // 若p->rchild不是线索(是右孩子)，p指向右孩子，返回循环，
  }              // 找这棵子树中序遍历的第1个结点
}
void PreThreading(BiThrTree p)
{ // PreOrderThreading()调用的递归函数
  if(!pre->rchild) // p的前驱没有右孩子
  {
    pre->rchild=p; // p前驱的后继指向p
    pre->RTag=Thread; // pre的右孩子为线索
  }
  if(!p->lchild) // p没有左孩子
```

```
  {
    p->LTag=Thread; // p的左孩子为线索
    p->lchild=pre; // p的左孩子指向前驱
  }
  pre=p; // 移动前驱
  if(p->LTag==Link) // p有左孩子
    PreThreading(p->lchild); // 对p的左孩子递归调用preThreading()
  if(p->RTag==Link) // p有右孩子
    PreThreading(p->rchild); // 对p的右孩子递归调用preThreading()
}
void PreOrderThreading(BiThrTree &Thrt,BiThrTree T)
{ // 先序线索化二叉树T，头结点的右指针指向先序遍历的最后1个结点
  if(!(Thrt=(BiThrTree)malloc(sizeof(BiThrNode)))) // 生成头结点
    exit(OVERFLOW);
  Thrt->LTag=Link; // 头结点的左指针为孩子
  Thrt->RTag=Thread; // 头结点的右指针为线索
  Thrt->rchild=Thrt; // 头结点的右指针指向自身
  if(!T) // 空树(见图6－20)
    Thrt->lchild=Thrt; // 头结点的左指针也指向自身
  else
  { // 非空树(见图6－21)
    Thrt->lchild=T; // 头结点的左指针指向根结点
    pre=Thrt; // 前驱为头结点
    PreThreading(T); // 从头结点开始先序递归线索化
    pre->rchild=Thrt; // 最后一个结点的后继指向头结点
    pre->RTag=Thread;
    Thrt->rchild=pre; // 头结点的后继指向最后一个结点
  }
}
void PreOrderTraverse_Thr(BiThrTree T,void(*Visit)(TElemType))
{ // 先序遍历线索二叉树T(头结点)的非递归算法
  BiThrTree p=T->lchild; // p指向根结点
  while(p!=T) // p没指向头结点(遍历的最后1个结点的后继指向头结点)
  {
    Visit(p->data); // 访问根结点
    if(p->LTag==Link) // p有左孩子
      p=p->lchild; // p指向左孩子(后继)
    else // p无左孩子
      p=p->rchild; // p指向右孩子或后继
  }
}
void PostThreading(BiThrTree p)
{ // PostOrderThreading()调用的递归函数
  if(p) // p不空
  {
    PostThreading(p->lchild); // 对p的左孩子递归调用PostThreading()
    PostThreading(p->rchild); // 对p的右孩子递归调用PostThreading()
    if(!p->lchild) // p没有左孩子
    {
      p->LTag=Thread; // p的左孩子为线索
      p->lchild=pre; // p的左孩子指向前驱
```

```
    }
    if(!pre->rchild) // p的前驱没有右孩子
    {
      pre->RTag=Thread; // p前驱的右孩子为线索
      pre->rchild=p; // p前驱的后继指向p
    }
    pre=p; // 移动前驱
  }
}
void PostOrderThreading(BiThrTree &Thrt,BiThrTree T)
{  // 后序递归线索化二叉树
  if(!(Thrt=(BiThrTree)malloc(sizeof(BiThrNode)))) // 生成头结点
    exit(OVERFLOW);
  Thrt->LTag=Link; // 头结点的左指针为孩子
  Thrt->RTag=Thread; // 头结点的右指针为线索
  if(!T) // 空树(见图6－20)
    Thrt->lchild=Thrt->rchild=Thrt; // 头结点的左右指针指向自身
  else
  { // 非空树(见图6－22)
    Thrt->lchild=Thrt->rchild=T; // 头结点的左右指针指向根结点(最后一个结点)
    pre=Thrt; // 前驱为头结点
    PostThreading(T); // 从头结点开始后序递归线索化
    if(pre->RTag!=Link) // 最后一个结点没有右孩子
    {
      pre->rchild=Thrt; // 最后一个结点的后继指向头结点
      pre->RTag=Thread;
    }
  }
}
void DestroyBiTree(BiThrTree &T)
{ // DestroyBiThrTree调用的递归函数，T指向根结点
  if(T) // 非空树
  {
    if(T->LTag==0) // 有左孩子
      DestroyBiTree(T->lchild); // 销毁左孩子子树
    if(T->RTag==0) // 有右孩子
      DestroyBiTree(T->rchild); // 销毁右孩子子树
    free(T); // 释放根结点
    T=NULL; // 空指针赋0
  }
}
void DestroyBiThrTree(BiThrTree &Thrt)
{ // 初始条件：线索二叉树Thrt存在。操作结果：销毁线索二叉树Thrt
  if(Thrt) // 头结点存在
  {
    if(Thrt->lchild) // 根结点存在
      DestroyBiTree(Thrt->lchild); // 递归销毁头结点lchild所指二叉树
    free(Thrt); // 释放头结点
    Thrt=NULL; // 线索二叉树Thrt指针赋0
  }
}
```

```
// main6-3.cpp 检验bo6-3.cpp的主程序
#define CHAR 1 // 字符型
// #define CHAR 0 // 整型(二者选一)
#if CHAR
  typedef char TElemType;
  TElemType Nil=' '; // 字符型以空格符为空
  #define form "%c" // 输入输出的格式为%c
#else
  typedef int TElemType;
  TElemType Nil=0; // 整型以0为空
  #define form "%d" // 输入输出的格式为%d
#endif
#include"c1.h"
#include"c6-3.h"
#include"bo6-3.cpp"
void vi(TElemType c)
{
  printf(form" ",c);
}
void main()
{
  BiThrTree H,T;
#if CHAR
  printf("请按先序输入二叉树(如:ab三个空格表示a为根结点,b为左子树的二叉树)\n");
#else
  printf("请按先序输入二叉树(如:1 2 0 0 0表示1为根结点,2为左子树的二叉树)\n");
#endif
  CreateBiThrTree(T); // 按先序产生二叉树
  InOrderThreading(H,T); // 在中序遍历的过程中，中序线索化二叉树
  printf("中序遍历(输出)线索二叉树:\n");
  InOrderTraverse_Thr(H,vi); // 中序遍历(输出)线索二叉树
  printf("\n");
  DestroyBiThrTree(H); // 销毁线索二叉树
#if CHAR
  printf("请按先序输入二叉树(如:ab三个空格表示a为根结点,b为左子树的二叉树)\n");
#else
  printf("请按先序输入二叉树(如:1 2 0 0 0表示1为根结点,2为左子树的二叉树)\n");
#endif
  scanf("%*c"); // 吃掉回车符
  CreateBiThrTree(T); // 按先序产生二叉树T
  PreOrderThreading(H,T); // 在先序遍历的过程中，先序线索化二叉树
  printf("先序遍历(输出)线索二叉树:\n");
  PreOrderTraverse_Thr(H,vi);
  DestroyBiThrTree(H); // 销毁线索二叉树
#if CHAR
  printf("\n请按先序输入二叉树(如:ab三个空格表示a为根结点,b为左子树的二叉树)\n");
#else
  printf("\n请按先序输入二叉树(如:1 2 0 0 0表示1为根结点,2为左子树的二叉树)\n");
#endif
  scanf("%*c"); // 吃掉回车符
  CreateBiThrTree(T); // 按先序产生二叉树T
```

```
  PostOrderThreading(H,T); // 在后序遍历的过程中，后序线索化二叉树
  DestroyBiThrTree(H); // 销毁线索二叉树
}
```

程序运行结果：

```
请按先序输入二叉树(如:ab三个空格表示a为根结点,b为左子树的二叉树)
abdg   e  c f  ↙(见图6－23)
中序遍历(输出)线索二叉树: (见图6－24)
g d b e a c f
请按先序输入二叉树(如:ab三个空格表示a为根结点,b为左子树的二叉树)
abdg   e  c f  ↙(见图6－23)
先序遍历(输出)线索二叉树: (见图6－25)
a b d g e c f
请按先序输入二叉树(如:ab三个空格表示a为根结点,b为左子树的二叉树)
abdg   e  c f  ↙(见图6－23)
```

图 6－26 是后序线索化的示意图。

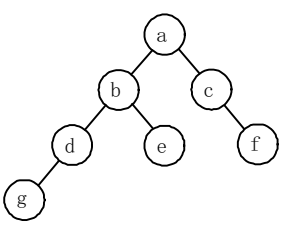


图 6－23　生成的二叉树

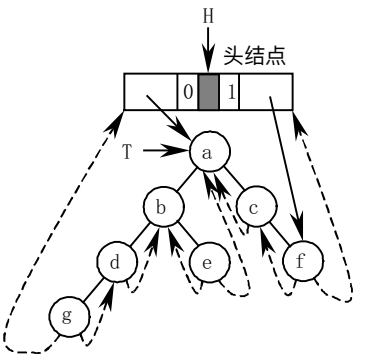


图 6－24　中序线索化二叉树

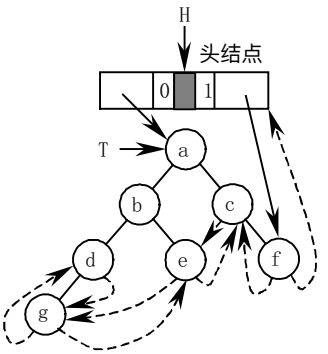


图 6－25　先序线索化二叉树

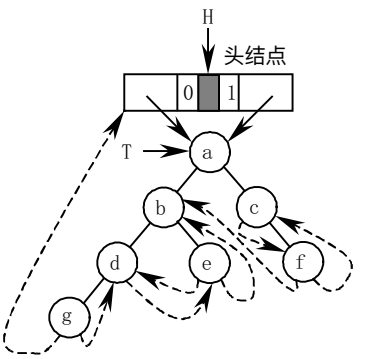


图 6－26　后序线索化二叉树

6.4　树和森林

二叉树是最简单的树，还有多叉树。森林是由 2 棵以上的树组成的。本节介绍多叉树和森林的存储方式。

6.4.1　树的存储结构

c6-4.h(见图 6－27 所示)是用顺序结构存储树的。它是定长的(100 个结点)，由 n 来确定有效结点数。parent 域的值为-1 的是根结点。图 6－28 是教科书中图 6.13 所示之树及其双亲表存储结构。

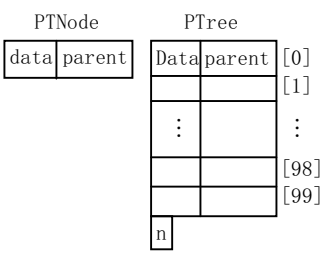
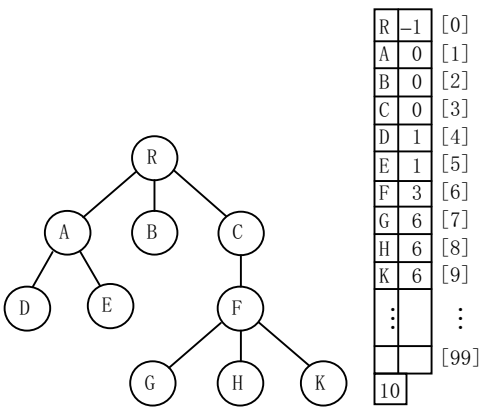


图 6－27　树的双亲表存储结构



(a) 教科书图 6.13 所示之树　(b) 存储表示

图 6－28　采用双亲表存储树的示例

```
// c6-4.h 树的双亲表存储结构(见图6－27)
#define MAX_TREE_SIZE 100
struct PTNode
{
  TElemType data;
  int parent; // 双亲位置域
};
struct PTree
{
  PTNode nodes[MAX_TREE_SIZE];
  int n; // 结点数
};
```

```
// bo6-4.cpp 树的双亲表存储(存储结构由 c6-4.h 定义)的基本操作(14 个)
#define ClearTree InitTree // 二者操作相同
#define DestroyTree InitTree // 二者操作相同
void InitTree(PTree &T)
{ // 操作结果：构造空树T
  T.n=0;
}
```

```
typedef struct
{
  int num;
  TElemType name;
}QElemType; // 定义队列元素类型
#include"c3-2.h" // 定义LinkQueue类型(链队列)
#include"bo3-2.cpp" // LinkQueue类型的基本操作
void CreateTree(PTree &T)
{ // 操作结果：构造树T
  LinkQueue q;
  QElemType p,qq;
  int i=1,j,l;
  char c[MAX_TREE_SIZE]; // 临时存放孩子结点数组
  InitQueue(q); // 初始化队列
  printf("请输入根结点(字符型,空格为空)：");
  scanf("%c%*c",&T.nodes[0].data); // 根结点序号为0，%*c吃掉回车符
  if(T.nodes[0].data!=Nil) // 非空树
  {
    T.nodes[0].parent=-1; // 根结点无双亲
    qq.name=T.nodes[0].data;
    qq.num=0;
    EnQueue(q,qq); // 入队此结点
    while(i<MAX_TREE_SIZE&&!QueueEmpty(q)) // 数组未满且队不空
    {
      DeQueue(q,qq); // 出队一个结点
      printf("请按长幼顺序输入结点%c的所有孩子：",qq.name);
      gets(c);
      l=strlen(c);
      for(j=0;j<l;j++)
      {
        T.nodes[i].data=c[j];
        T.nodes[i].parent=qq.num;
        p.name=c[j];
        p.num=i;
        EnQueue(q,p); // 入队此结点
        i++;
      }
    }
    if(i>MAX_TREE_SIZE)
    {
      printf("结点数超过数组容量\n");
      exit(OVERFLOW);
    }
    T.n=i;
  }
  else
    T.n=0;
}
Status TreeEmpty(PTree T)
{ // 初始条件：树T存在。操作结果：若T为空树，则返回TRUE；否则返回FALSE
  if(T.n)
```

```
    return FALSE;
  else
    return TRUE;
}
int TreeDepth(PTree T)
{ // 初始条件：树T存在。操作结果：返回T的深度
  int k,m,def,max=0;
  for(k=0;k<T.n;++k)
  {
    def=1; // 初始化本结点的深度
    m=T.nodes[k].parent;
    while(m!=-1)
    {
      m=T.nodes[m].parent;
      def++;
    }
    if(max<def)
      max=def;
  }
  return max; // 最大深度
}
TElemType Root(PTree T)
{ // 初始条件：树T存在。操作结果：返回T的根
  int i;
  for(i=0;i<T.n;i++)
    if(T.nodes[i].parent<0)
      return T.nodes[i].data;
  return Nil;
}
TElemType Value(PTree T,int i)
{ // 初始条件：树T存在，i是树T中结点的序号。操作结果：返回第i个结点的值
  if(i<T.n)
    return T.nodes[i].data;
  else
    return Nil;
}
Status Assign(PTree &T,TElemType cur_e,TElemType value)
{ // 初始条件：树T存在，cur_e是树T中结点的值。操作结果：改cur_e为value
  int j;
  for(j=0;j<T.n;j++)
  {
    if(T.nodes[j].data==cur_e)
    {
      T.nodes[j].data=value;
      return OK;
    }
  }
  return ERROR;
}
TElemType Parent(PTree T,TElemType cur_e)
{ // 初始条件：树T存在，cur_e是T中某个结点
```

```
  // 操作结果：若cur_e是T的非根结点，则返回它的双亲；否则函数值为“空”
  int j;
  for(j=1;j<T.n;j++) // 根结点序号为0
    if(T.nodes[j].data==cur_e)
      return T.nodes[T.nodes[j].parent].data;
  return Nil;
}
TElemType LeftChild(PTree T,TElemType cur_e)
{ // 初始条件：树T存在，cur_e是T中某个结点
  // 操作结果：若cur_e是T的非叶子结点，则返回它的最左孩子；否则返回“空”
  int i,j;
  for(i=0;i<T.n;i++)
    if(T.nodes[i].data==cur_e) // 找到cur_e，其序号为i
      break;
  for(j=i+1;j<T.n;j++) // 根据树的构造函数，孩子的序号＞其双亲的序号
    if(T.nodes[j].parent==i) // 根据树的构造函数，最左孩子(长子)的序号＜其它孩子的序号
      return T.nodes[j].data;
  return Nil;
}
TElemType RightSibling(PTree T,TElemType cur_e)
{ // 初始条件：树T存在，cur_e是T中某个结点
  // 操作结果：若cur_e有右(下一个)兄弟，则返回它的右兄弟；否则返回“空”
  int i;
  for(i=0;i<T.n;i++)
    if(T.nodes[i].data==cur_e) // 找到cur_e，其序号为i
      break;
  if(T.nodes[i+1].parent==T.nodes[i].parent)
  // 根据树的构造函数，若cur_e有右兄弟的话则右兄弟紧接其后
    return T.nodes[i+1].data;
  return Nil;
}
void Print(PTree T)
{ // 输出树T。加
  int i;
  printf("结点个数=%d\n",T.n);
  printf(" 结点 双亲\n");
  for(i=0;i<T.n;i++)
  {
    printf("    %c",Value(T,i)); // 结点
    if(T.nodes[i].parent>=0) // 有双亲
      printf("    %c",Value(T,T.nodes[i].parent)); // 双亲
    printf("\n");
  }
}
Status InsertChild(PTree &T,TElemType p,int i,PTree c)
{ // 初始条件：树T存在，p是T中某个结点，1≤i≤p所指结点的度+1，非空树c与T不相交
  // 操作结果：插入c为T中p结点的第i棵子树
  int j,k,l,f=1,n=0; // 设交换标志f的初值为1，p的孩子数n的初值为0
  PTNode t;
  if(!TreeEmpty(T)) // T不空
  {
```

```
    for(j=0;j<T.n;j++) // 在T中找p的序号
      if(T.nodes[j].data==p) // p的序号为j
        break;
    l=j+1; // 如果c是p的第1棵子树，则插在j+1处
    if(i>1) // c不是p的第1棵子树
    {
      for(k=j+1;k<T.n;k++) // 从j+1开始找p的前i-1个孩子
        if(T.nodes[k].parent==j) // 当前结点是p的孩子
        {
          n++; // 孩子数加1
          if(n==i-1) // 找到p的第i-1个孩子，其序号为k1
            break;
        }
      l=k+1; // c插在k+1处
    } // p的序号为j，c插在l处
    if(l<T.n) // 插入点l不在最后
      for(k=T.n-1;k>=l;k--) // 依次将序号l以后的结点向后移c.n个位置
      {
        T.nodes[k+c.n]=T.nodes[k];
        if(T.nodes[k].parent>=l)
          T.nodes[k+c.n].parent+=c.n;
      }
    for(k=0;k<c.n;k++)
    {
      T.nodes[l+k].data=c.nodes[k].data; // 依次将树c的所有结点插于此处
      T.nodes[l+k].parent=c.nodes[k].parent+l;
    }
    T.nodes[l].parent=j; // 树c的根结点的双亲为p
    T.n+=c.n; // 树T的结点数加c.n个
    while(f)
    { // 从插入点之后，将结点仍按层序排列
      f=0; // 交换标志置0
      for(j=l;j<T.n-1;j++)
        if(T.nodes[j].parent>T.nodes[j+1].parent)
        { // 如果结点j的双亲排在结点j+1的双亲之后(树没有按层序排列)，交换两结点
          t=T.nodes[j];
          T.nodes[j]=T.nodes[j+1];
          T.nodes[j+1]=t;
          f=1; // 交换标志置1
          for(k=j;k<T.n;k++) // 改变双亲序号
            if(T.nodes[k].parent==j)
              T.nodes[k].parent++; // 双亲序号改为j+1
            else if(T.nodes[k].parent==j+1)
              T.nodes[k].parent--; // 双亲序号改为j
        }
    }
    return OK;
  }
  else // 树T不存在
    return ERROR;
}
```

```
Status deleted[MAX_TREE_SIZE+1]; // 删除标志数组(全局量)
void DeleteChild(PTree &T,TElemType p,int i)
{ // 初始条件：树T存在，p是T中某个结点，1≤i≤p所指结点的度
  // 操作结果：删除T中结点p的第i棵子树
  int j,k,n=0;
  LinkQueue q;
  QElemType pq,qq;
  for(j=0;j<=T.n;j++)
    deleted[j]=0; // 置初值为0(不删除标记)
  pq.name='a'; // 此成员不用
  InitQueue(q); // 初始化队列
  for(j=0;j<T.n;j++)
    if(T.nodes[j].data==p)
      break; // j为结点p的序号
  for(k=j+1;k<T.n;k++)
  {
    if(T.nodes[k].parent==j)
      n++;
    if(n==i)
      break; // k为p的第i棵子树结点的序号
  }
  if(k<T.n) // p的第i棵子树结点存在
  {
    n=0;
    pq.num=k;
    deleted[k]=1; // 置删除标记
    n++;
    EnQueue(q,pq);
    while(!QueueEmpty(q))
    {
      DeQueue(q,qq);
      for(j=qq.num+1;j<T.n;j++)
        if(T.nodes[j].parent==qq.num)
        {
          pq.num=j;
          deleted[j]=1; // 置删除标记
          n++;
          EnQueue(q,pq);
        }
    }
    for(j=0;j<T.n;j++)
      if(deleted[j]==1)
      {
        for(k=j+1;k<=T.n;k++)
        {
          deleted[k-1]=deleted[k];
          T.nodes[k-1]=T.nodes[k];
          if(T.nodes[k].parent>j)
            T.nodes[k-1].parent--;
        }
        j--;
```

```
      }
    T.n--n; // n为待删除结点数
  }
}
void TraverseTree(PTree T,void(*Visit)(TElemType))
{ // 初始条件：二叉树T存在，Visit是对结点操作的应用函数
  // 操作结果：层序遍历树T，对每个结点调用函数Visit一次且仅一次
  int i;
  for(i=0;i<T.n;i++)
    Visit(T.nodes[i].data);
  printf("\n");
}

// main6-4.cpp 检验bo6-4.cpp的主程序
#include"c1.h"
typedef char TElemType;
TElemType Nil=' '; // 以空格符为空
#include"c6-4.h"
#include"bo6-4.cpp"
void vi(TElemType c)
{
  printf("%c ",c);
}
void main()
{
  int i;
  PTree T,p;
  TElemType e,e1;
  InitTree(T);
  printf("构造空树后,树空否? %d(1:是 0:否) 树根为%c 树的深度为d\n",TreeEmpty(T),Root(T),
  TreeDepth(T));
  CreateTree(T);
  printf("构造树T后,树空否? %d(1:是 0:否) 树根为%c 树的深度为d\n",TreeEmpty(T),Root(T),
  TreeDepth(T));
  printf("层序遍历树T:\n");
  TraverseTree(T,vi);
  printf("请输入待修改的结点的值 新值: ");
  scanf("%c%*c%c%*c",&e,&e1);
  Assign(T,e,e1);
  printf("层序遍历修改后的树T:\n");
  TraverseTree(T,vi);
  printf("%c的双亲是%c,长子是%c,下一个兄弟是c\n",e1,Parent(T,e1),LeftChild(T,e1),
  RightSibling(T,e1));
  printf("建立树p:\n");
  InitTree(p);
  CreateTree(p);
  printf("层序遍历树p:\n");
  TraverseTree(p,vi);
  printf("将树p插到树T中，请输入T中p的双亲结点 子树序号: ");
  scanf("%c%d%*c",&e,&i);
  InsertChild(T,e,i,p);
```

```
  Print(T);
  printf("删除树T中结点e的第i棵子树, 请输入e i: ");
  scanf("%c%d",&e,&i);
  DeleteChild(T,e,i);
  Print(T);
}
```

程序运行结果:

```
构造空树后,树空否? 1(1:是 0:否) 树根为  树的深度为0
请输入根结点(字符型,空格为空): R↙
请按长幼顺序输入结点R的所有孩子: ABC↙
请按长幼顺序输入结点A的所有孩子: DE↙
请按长幼顺序输入结点B的所有孩子: ↙
请按长幼顺序输入结点C的所有孩子: F↙
请按长幼顺序输入结点D的所有孩子: ↙
请按长幼顺序输入结点E的所有孩子: ↙
请按长幼顺序输入结点F的所有孩子: GHK↙
请按长幼顺序输入结点G的所有孩子: ↙
请按长幼顺序输入结点H的所有孩子: ↙
请按长幼顺序输入结点K的所有孩子: ↙
构造树T后,树空否? 0(1:是 0:否) 树根为R 树的深度为4
层序遍历树T:(见图6-28(a))
R A B C D E F G H K
请输入待修改的结点的值 新值: D d↙
层序遍历修改后的树T:
R A B C d E F G H K
d的双亲是A,长子是 ,下一个兄弟是E
建立树p:
请输入根结点(字符型,空格为空): f↙
请按长幼顺序输入结点f的所有孩子: ghk↙
请按长幼顺序输入结点g的所有孩子: ↙
请按长幼顺序输入结点h的所有孩子: ↙
请按长幼顺序输入结点k的所有孩子: ↙
层序遍历树p:(见图6-29)
f g h k
将树p插到树T中, 请输入T中p的双亲结点 子树序号: R 3↙(见图6-30)
结点个数=14
 结点    双亲
    R
    A    R
    B    R
    f    R
    C    R
    d    A
    E    A
    g    f
    h    f
    k    f
```

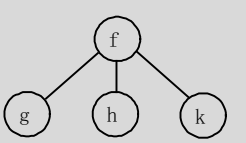


图 6-29　树p图示

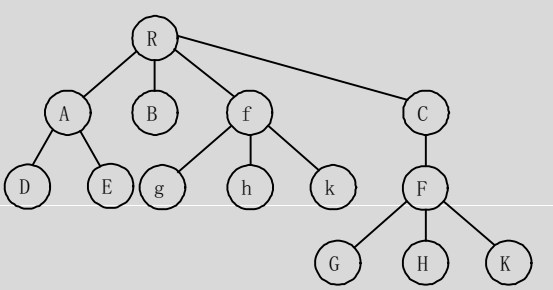


图 6-30　树 p 插到树 T 后的状况

```
    F     C
    G     F
    H     F
    K     F
删除树T中结点e的第i棵子树，请输入e i: C 1↙(见图6－31)
结点个数=10
 结点    双亲
    R
    A     R
    B     R
    f     R
    C     R
    d     A
    E     A
    g     f
    h     f
    k     f
```

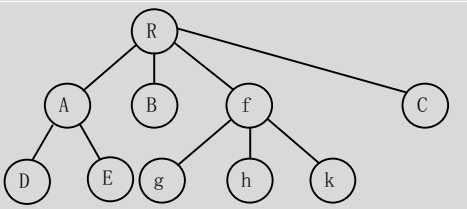


图 6－31　删除树 T 中结点 C 的第 1 棵子树后的状况

```
// c6-5.h 树的二叉链表(孩子—兄弟)存储结构(见图6－32)
typedef struct CSNode
{
  TElemType data;
  CSNode *firstchild,*nextsibling;
}CSNode,*CSTree;
```

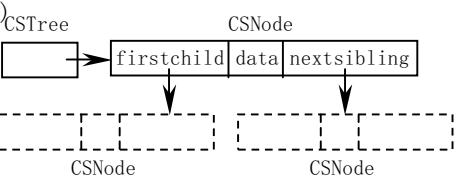


图 6－32　树的(孩子—兄弟)二叉链表存储结构

一棵树无论有多少叉，它最多有一个长子和一个排序恰在其下的兄弟。根据这样的定义，则每个结点的结构就都统一到了二叉链表结构上。这样有利于对结点进行操作。图 6－33 是图 6－28(a)所示之树的二叉链表(孩子—兄弟)存储结构。

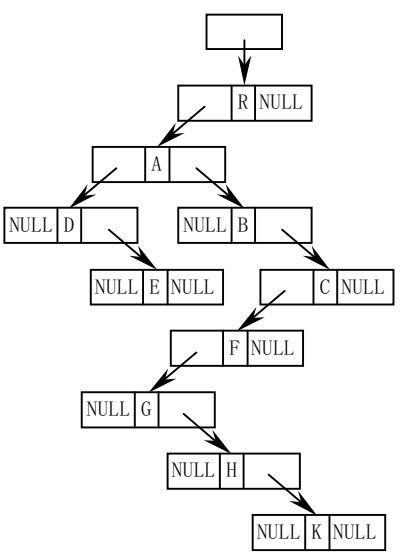


图 6－33　图 6－28(a)所示树的(孩子—兄弟)二叉链表存储结构

```
// func6-2.cpp bo6-5.cpp和algo7-1.cpp调用
void PreOrderTraverse(CSTree T,void(*Visit)(TElemType))
```

```
{ // 先根遍历孩子—兄弟二叉链表结构的树T
  if(T)
  {
    Visit(T->data); // 先访问根结点
    PreOrderTraverse(T->firstchild,Visit); // 再先根遍历长子子树
    PreOrderTraverse(T->nextsibling,Visit); // 最后先根遍历下一个兄弟子树
  }
}

// bo6-5.cpp 树的二叉链表(孩子—兄弟)存储(存储结构由c6-5.h定义)的基本操作(17个)
#define ClearTree DestroyTree // 二者操作相同
#include"func6-2.cpp" // 包括PreOrderTraverse()
void InitTree(CSTree &T)
{ // 操作结果：构造空树T
  T=NULL;
}
void DestroyTree(CSTree &T)
{ // 初始条件：树T存在。操作结果：销毁树T
  if(T)
  {
    if(T->firstchild) // T有长子
      DestroyTree(T->firstchild); // 销毁T的长子为根结点的子树
    if(T->nextsibling) // T有下一个兄弟
      DestroyTree(T->nextsibling); // 销毁T的下一个兄弟为根结点的子树
    free(T); // 释放根结点
    T=NULL;
  }
}
typedef CSTree QElemType; // 定义队列元素类型
#include"c3-2.h" // 定义LinkQueue类型(链队列)
#include"bo3-2.cpp" // LinkQueue类型的基本操作
void CreateTree(CSTree &T)
{ // 构造树T
  char c[20]; // 临时存放孩子结点(设不超过20个)的值
  CSTree p,p1;
  LinkQueue q;
  int i,l;
  InitQueue(q);
  printf("请输入根结点(字符型,空格为空)：");
  scanf("%c%*c",&c[0]);
  if(c[0]!=Nil) // 非空树
  {
    T=(CSTree)malloc(sizeof(CSNode)); // 建立根结点
    T->data=c[0];
    T->nextsibling=NULL;
    EnQueue(q,T); // 入队根结点的指针
    while(!QueueEmpty(q)) // 队不空
    {
      DeQueue(q,p); // 出队一个结点的指针
      printf("请按长幼顺序输入结点%c的所有孩子：",p->data);
      gets(c);
```

```
      l=strlen(c);
      if(l>0) // 有孩子
      {
        p1=p->firstchild=(CSTree)malloc(sizeof(CSNode)); // 建立长子结点
        p1->data=c[0];
        for(i=1;i<l;i++)
        {
          p1->nextsibling=(CSTree)malloc(sizeof(CSNode)); // 建立下一个兄弟结点
          EnQueue(q,p1); // 入队上一个结点
          p1=p1->nextsibling;
          p1->data=c[i];
        }
        p1->nextsibling=NULL;
        EnQueue(q,p1); // 入队最后一个结点
      }
      else
        p->firstchild=NULL; // 长子指针为空
    }
  }
  else
    T=NULL; // 空树
}
Status TreeEmpty(CSTree T)
{ // 初始条件：树T存在。操作结果：若T为空树，则返回TURE；否则返回FALSE
  if(T) // T不空
    return FALSE;
  else
    return TRUE;
}
int TreeDepth(CSTree T)
{ // 初始条件：树T存在。操作结果：返回T的深度
  CSTree p;
  int depth,max=0;
  if(!T) // 树空
    return 0;
  if(!T->firstchild) // 树无长子
    return 1;
  for(p=T->firstchild;p;p=p->nextsibling)
  { // 求子树深度的最大值
    depth=TreeDepth(p);
    if(depth>max)
      max=depth;
  }
  return max+1; // 树的深度=子树深度最大值+1
}
TElemType Value(CSTree p)
{ // 返回p所指结点的值
  return p->data;
}
TElemType Root(CSTree T)
{ // 初始条件：树T存在。操作结果：返回T的根
```

```
  if(T)
    return Value(T);
  else
    return Nil;
}
CSTree Point(CSTree T,TElemType s)
{ // 返回二叉链表(孩子—兄弟)树T中指向元素值为s的结点的指针。另加
  LinkQueue q;
  QElemType a;
  if(T) // 非空树
  {
    InitQueue(q); // 初始化队列
    EnQueue(q,T); // 根结点入队
    while(!QueueEmpty(q)) // 队不空
    {
      DeQueue(q,a); // 出队,队列元素赋给a
      if(a->data==s)
        return a;
      if(a->firstchild) // 有长子
        EnQueue(q,a->firstchild); // 入队长子
      if(a->nextsibling) // 有下一个兄弟
        EnQueue(q,a->nextsibling); // 入队下一个兄弟
    }
  }
  return NULL;
}
Status Assign(CSTree &T,TElemType cur_e,TElemType value)
{ // 初始条件: 树T存在, cur_e是树T中结点的值。操作结果: 改cur_e为value
  CSTree p;
  if(T) // 非空树
  {
    p=Point(T,cur_e); // p为cur_e的指针
    if(p) // 找到cur_e
    {
      p->data=value; // 赋新值
      return OK;
    }
  }
  return ERROR  ; // 树空或没找到
}
TElemType Parent(CSTree T,TElemType cur_e)
{ // 初始条件: 树T存在, cur_e是T中某个结点
  // 操作结果: 若cur_e是T的非根结点, 则返回它的双亲; 否则函数值为“ 空”
  CSTree p,t;
  LinkQueue q;
  InitQueue(q);
  if(T) // 树非空
  {
    if(Value(T)==cur_e) // 根结点值为cur_e
      return Nil;
    EnQueue(q,T); // 根结点入队
```

```
    while(!QueueEmpty(q))
    {
      DeQueue(q,p);
      if(p->firstchild) // p有长子
      {
        if(p->firstchild->data==cur_e) // 长子为cur_e
          return Value(p); // 返回双亲
        t=p; // 双亲指针赋给t
        p=p->firstchild; // p指向长子
        EnQueue(q,p); // 入队长子
        while(p->nextsibling) // 有下一个兄弟
        {
          p=p->nextsibling; // p指向下一个兄弟
          if(Value(p)==cur_e) // 下一个兄弟为cur_e
            return Value(t); // 返回双亲
          EnQueue(q,p); // 入队下一个兄弟
        }
      }
    }
  }
  return Nil; // 树空或没找到cur_e
}
TElemType LeftChild(CSTree T,TElemType cur_e)
{ // 初始条件: 树T存在, cur_e是T中某个结点
  // 操作结果: 若cur_e是T的非叶子结点, 则返回它的最左孩子; 否则返回" 空"
  CSTree f;
  f=Point(T,cur_e); // f指向结点cur_e
  if(f&&f->firstchild) // 找到结点cur_e且结点cur_e有长子
    return f->firstchild->data;
  else
    return Nil;
}
TElemType RightSibling(CSTree T,TElemType cur_e)
{ // 初始条件: 树T存在, cur_e是T中某个结点
  // 操作结果: 若cur_e有右兄弟, 则返回它的右兄弟; 否则返回" 空"
  CSTree f;
  f=Point(T,cur_e); // f指向结点cur_e
  if(f&&f->nextsibling) // 找到结点cur_e且结点cur_e有右兄弟
    return f->nextsibling->data;
  else
    return Nil; // 树空
}
Status InsertChild(CSTree &T,CSTree p,int i,CSTree c)
{ // 初始条件: 树T存在, p指向T中某个结点, 1≤i≤p所指结点的度+1, 非空树c与T不相交
  // 操作结果: 插入c为T中p结点的第i棵子树
  // 因为p所指结点的地址不会改变, 故p不需是引用类型
  int j;
  if(T) // T不空
  {
    if(i==1) // 插入c为p的长子
    {
```

```
      c->nextsibling=p->firstchild; // p的原长子现是c的下一个兄弟(c本无兄弟)
      p->firstchild=c;
    }
    else // 找插入点
    {
      p=p->firstchild; // 指向p的长子
      j=2;
      while(p&&j<i)
      {
        p=p->nextsibling;
        j++;
      }
      if(j==i) // 找到插入位置
      {
        c->nextsibling=p->nextsibling;
        p->nextsibling=c;
      }
      else // p原有孩子数小于i-1
        return ERROR;
    }
    return OK;
  }
  else // T空
    return ERROR;
}
Status DeleteChild(CSTree &T,CSTree p,int i)
{ // 初始条件: 树T存在, p指向T中某个结点, 1≤i≤p所指结点的度
  // 操作结果: 删除T中p所指结点的第i棵子树
  // 因为p所指结点的地址不会改变, 故p不需是引用类型
  CSTree b;
  int j;
  if(T) // T不空
  {
    if(i==1) // 删除长子
    {
      b=p->firstchild;
      p->firstchild=b->nextsibling; // p的原次子现是长子
      b->nextsibling=NULL;
      DestroyTree(b);
    }
    else // 删除非长子
    {
      p=p->firstchild; // p指向长子
      j=2;
      while(p&&j<i)
      {
        p=p->nextsibling;
        j++;
      }
      if(j==i) // 找到第i棵子树
      {
```

```
        b=p->nextsibling;
        p->nextsibling=b->nextsibling;
        b->nextsibling=NULL;
        DestroyTree(b);
      }
      else // p原有孩子数小于i
        return ERROR;
    }
    return OK;
  }
  else
    return ERROR;
}
void PostOrderTraverse(CSTree T,void(*Visit)(TElemType))
{ // 后根遍历孩子—兄弟二叉链表结构的树T
  CSTree p;
  if(T)
  {
    if(T->firstchild) // 有长子
    {
      PostOrderTraverse(T->firstchild,Visit); // 后根遍历长子子树
      p=T->firstchild->nextsibling; // p指向长子的下一个兄弟
      while(p)
      {
        PostOrderTraverse(p,Visit); // 后根遍历下一个兄弟子树
        p=p->nextsibling; // p指向再下一个兄弟
      }
    }
    Visit(Value(T)); // 最后访问根结点
  }
}
void LevelOrderTraverse(CSTree T,void(*Visit)(TElemType))
{ // 层序遍历孩子—兄弟二叉链表结构的树T
  CSTree p;
  LinkQueue q;
  InitQueue(q);
  if(T)
  {
    Visit(Value(T)); // 先访问根结点
    EnQueue(q,T); // 入队根结点的指针
    while(!QueueEmpty(q)) // 队不空
    {
      DeQueue(q,p); // 出队一个结点的指针
      if(p->firstchild) // 有长子
      {
        p=p->firstchild;
        Visit(Value(p)); // 访问长子结点
        EnQueue(q,p); // 入队长子结点的指针
        while(p->nextsibling) // 有下一个兄弟
        {
          p=p->nextsibling;
```

```
          Visit(Value(p)); // 访问下一个兄弟
          EnQueue(q,p); // 入队兄弟结点的指针
        }
      }
    }
  }
}

// main6-5.cpp 检验bo6-5.cpp的主程序
#include"c1.h"
typedef char TElemType;
TElemType Nil=' '; // 以空格符为空
#include"c6-5.h"
#include"bo6-5.cpp"
void vi(TElemType c)
{
  printf("%c ",c);
}
void main()
{
  int i;
  CSTree T,p,q;
  TElemType e,e1;
  InitTree(T);
  printf("构造空树后,树空否? %d(1:是 0:否) 树根为%c 树的深度为%d\n",TreeEmpty(T),Root(T),
  TreeDepth(T));
  CreateTree(T);
  printf("构造树T后,树空否? %d(1:是 0:否) 树根为%c 树的深度为%d\n",TreeEmpty(T),Root(T),
  TreeDepth(T));
  printf("先根遍历树T:\n");
  PreOrderTraverse(T,vi);
  printf("\n请输入待修改的结点的值 新值: ");
  scanf("%c%*c%c%*c",&e,&e1);
  Assign(T,e,e1);
  printf("后根遍历修改后的树T:\n");
  PostOrderTraverse(T,vi);
  printf("\n%c的双亲是%c,长子是%c,下一个兄弟是%c\n",e1,Parent(T,e1),LeftChild(T,e1),
  RightSibling(T,e1));
  printf("建立树p:\n");
  InitTree(p);
  CreateTree(p);
  printf("层序遍历树p:\n");
  LevelOrderTraverse(p,vi);
  printf("\n将树p插到树T中, 请输入T中p的双亲结点 子树序号: ");
  scanf("%c%d%*c",&e,&i);
  q=Point(T,e);
  InsertChild(T,q,i,p);
  printf("层序遍历树T:\n");
  LevelOrderTraverse(T,vi);
  printf("\n删除树T中结点e的第i棵子树, 请输入e i: ");
  scanf("%c%d",&e,&i);
```

```
  q=Point(T,e);
  DeleteChild(T,q,i);
  printf("层序遍历树T:\n",e,i);
  LevelOrderTraverse(T,vi);
  printf("\n");
  DestroyTree(T);
}
```

程序运行结果:

```
构造空树后,树空否? 1(1:是 0:否) 树根为   树的深度为0
请输入根结点(字符型,空格为空): R↙
请按长幼顺序输入结点R的所有孩子: ABC↙
请按长幼顺序输入结点A的所有孩子: DE↙
请按长幼顺序输入结点B的所有孩子: ↙
请按长幼顺序输入结点C的所有孩子: F↙
请按长幼顺序输入结点D的所有孩子: ↙
请按长幼顺序输入结点E的所有孩子: ↙
请按长幼顺序输入结点F的所有孩子: GHK↙
请按长幼顺序输入结点G的所有孩子: ↙
请按长幼顺序输入结点H的所有孩子: ↙
请按长幼顺序输入结点K的所有孩子: ↙
构造树T后,树空否? 0(1:是 0:否) 树根为R 树的深度为4
先根遍历树T:(见图6-28(a))
R A D E B C F G H K
请输入待修改的结点的值 新值: D d↙
后根遍历修改后的树T:
d E A B G H K F C R
d的双亲是A,长子是 ,下一个兄弟是E
建立树p:
请输入根结点(字符型,空格为空): f↙
请按长幼顺序输入结点f的所有孩子: ghk↙
请按长幼顺序输入结点g的所有孩子: ↙
请按长幼顺序输入结点h的所有孩子: ↙
请按长幼顺序输入结点k的所有孩子: ↙
层序遍历树p:(见图6-29)
f g h k
将树p插到树T中, 请输入T中p的双亲结点 子树序号: R 3↙
层序遍历树T:(见图6-30)
R A B f C d E g h k F G H K
删除树T中结点e的第i棵子树, 请输入e i: C 1↙
层序遍历树T:(见图6-31)
R A B f C d E g h k
```

6.4.2　森林与二叉树的转换

6.4.3　树和森林的遍历

6.5　树与等价问题

6.6　赫夫曼树及其应用

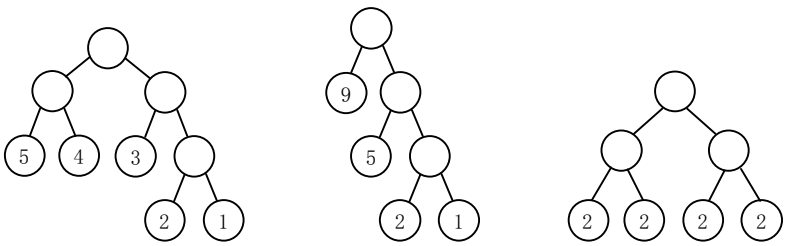
6.6.1　最优二叉树(赫夫曼树)

最优二叉树是带权路径长度最短的二叉树。根据结点的个数、权值的不同，最优二叉树的形状也各不相同。图 6－34 是 3 棵最优二叉树的例子。它们的共同特点是：带权值的结点都是叶子结点。权值越小的结点，其到根结点的路径越长。构造最优二叉树的方法如下：

(1) 将每个带有权值的结点作为一棵仅有根结点的二叉树，树的权值为结点的权值；

(2) 将其中两棵权值最小的树组成一棵新二叉树，新树的权值为两棵树的权值之和；

(3) 重复(2)，直到所有结点都在一棵二叉树上。这棵二叉树就是最优二叉树。



(a)　最优二叉树　　(b)　权值差别大的结点构成的最优二叉树　　(c)　权值相同的结点构成的最优二叉树

图 6－34　3 棵形状不同的最优二叉树

最优二叉树的左右子树是可以互换的，因为这不影响树的带权路径长度。当结点的权值差别大到一定程度，最优二叉树就形成了如图 6－34(b)所示的“一边倒”的形状。有些书称最优二叉树都是这种“一边倒”的形状是不对的。这通过计算二叉树的带权路径长度是否最短就可看出。当所有结点的权值一样，或其权值差别很小，最优二叉树就形成了如图 6－34(c)所示的完全二叉树的形状。叶子结点的路径长度近似相等。

最优二叉树除了叶子结点就是度为 2 的结点，没有度为 1 的结点。这样才使得树的带权路径长度最短。根据二叉树的性质 3，最优二叉树的结点数为叶子数的 2 倍减 1。

6.6.2　赫夫曼编码

```
// c6-7.h 赫夫曼树和赫夫曼编码的存储结构(见图6－35)
typedef struct
{
  unsigned int weight;
  unsigned int parent,lchild,rchild;
}HTNode,*HuffmanTree; // 动态分配数组存储赫夫曼树
typedef char **HuffmanCode; // 动态分配数组存储赫夫曼编码表
```

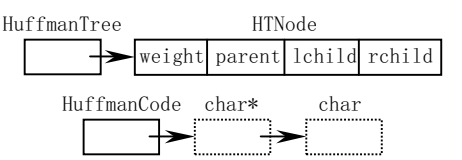


图 6－35　赫夫曼树和赫夫曼编码的存储结构

c6-7.h 定义的二叉树结构是我们在前边没有讨论过的，但它特别适合建立赫夫曼树。赫夫曼树是由多棵二叉树(森林)组合成而的一棵树。这种二叉树结构既适合表示树，也适合表示森林。赫夫曼树结点的结构包括权值、双亲及左右孩子，双亲值为 0 的是根结点，左右孩子值均为 0 的是叶子结点。这种二叉树结构是动态生成的顺序结构。当叶子结点数确定，赫夫曼树的结点数也确定。由图 6－36(d)可见，建成的赫夫曼树除 0 号结点空间不用外，每个结点空间都没空置。

```
// func6-1.cpp  程序  algo6-1.cpp和algo6-2.cpp要调用
int min(HuffmanTree t,int i)
{ // 返回i个结点中权值最小的树的根结点序号，函数select()调用
  int j,flag;
  unsigned int k=UINT_MAX; // 取k为不小于可能的值(无符号整型最大值)
  for(j=1;j<=i;j++)
    if(t[j].weight<k&&t[j].parent==0) // t[j]是树的根结点
      k=t[j].weight,flag=j;
  t[flag].parent=1; // 给选中的根结点的双亲赋1，避免第2次查找该结点
  return flag;
}
void select(HuffmanTree t,int i,int &s1,int &s2)
{ // 在i个结点中选择2个权值最小的树的根结点序号，s1为其中序号小的那个
  int j;
  s1=min(t,i);
  s2=min(t,i);
  if(s1>s2)
  {
    j=s1;
    s1=s2;
    s2=j;
  }
}

// algo6-1.cpp 求赫夫曼编码。实现算法6.12的程序
#include"c1.h"
#include"c6-7.h"
#include"func6-1.cpp"
void HuffmanCoding(HuffmanTree &HT,HuffmanCode &HC,int *w,int n) // 算法6.12
{ // w存放n个字符的权值(均>0)，构造赫夫曼树HT，并求出n个字符的赫夫曼编码HC
  int m,i,s1,s2,start;
  unsigned c,f;
  HuffmanTree p;
  char *cd;
  if(n<=1)
    return;
  m=2*n-1;
  HT=(HuffmanTree)malloc((m+1)*sizeof(HTNode)); // 0号单元未用
  for(p=HT+1,i=1;i<=n;++i,++p,++w)
  {
    (*p).weight=*w;
```

```
    (*p).parent=0;
    (*p).lchild=0;
    (*p).rchild=0;
  }
  for(;i<=m;++i,++p)
    (*p).parent=0;
  for(i=n+1;i<=m;++i) // 建赫夫曼树
  { // 在HT[1～i-1]中选择parent为0且weight最小的两个结点，其序号分别为s1和s2
    select(HT,i-1,s1,s2);
    HT[s1].parent=HT[s2].parent=i;
    HT[i].lchild=s1;
    HT[i].rchild=s2;
    HT[i].weight=HT[s1].weight+HT[s2].weight;
  }
  // 从叶子到根逆向求每个字符的赫夫曼编码
  HC=(HuffmanCode)malloc((n+1)*sizeof(char*));
  // 分配n个字符编码的头指针向量([0]不用)
  cd=(char*)malloc(n*sizeof(char)); // 分配求编码的工作空间
  cd[n-1]='\0'; // 编码结束符
  for(i=1;i<=n;i++)
  { // 逐个字符求赫夫曼编码
    start=n-1; // 编码结束符位置
    for(c=i,f=HT[i].parent;f!=0;c=f,f=HT[f].parent)
    // 从叶子到根逆向求编码
      if(HT[f].lchild==c)
        cd[--start]='0';
      else
        cd[--start]='1';
    HC[i]=(char*)malloc((n-start)*sizeof(char));
    // 为第i个字符编码分配空间
    strcpy(HC[i],&cd[start]); // 从cd复制编码(串)到HC
  }
  free(cd); // 释放工作空间
}
void main()
{
  HuffmanTree HT;
  HuffmanCode HC;
  int *w,n,i;
  printf("请输入权值的个数(>1): ");
  scanf("%d",&n);
  w=(int*)malloc(n*sizeof(int));
  printf("请依次输入%d个权值(整型):\n",n);
  for(i=0;i<=n-1;i++)
    scanf("%d",w+i);
  HuffmanCoding(HT,HC,w,n);
  for(i=1;i<=n;i++)
    puts(HC[i]);
}
```

程序运行结果(以教科书图 6.24 为例，如图 6－36 所示)：

```
请输入权值的个数(>1)：4↙
请依次输入4个权值(整型)：
7 5 2 4↙
0
10
110
111
```

图 6－36 是运行过程的图解。初始状态下(见图 6－36(b))，权值分别为 7、5、2、4 的 4 个结点是 4 棵独立的树(根结点)。它们没有双亲，也没有左右孩子。反复查找权值最小的两棵树，并把它们合并成一棵树，其权值为两树的权值之和。最后，所有结点合并成一棵赫夫曼树(见图 6－36(d))。

算法 6.12(在 algo6-1.cpp 中)在找到两个无双亲且权值最小的结点后，将序号小的结点作为左子树，序号大的结点作为右子树，如果不按这个规则，赫夫曼编码的形式会改变。但码长不会改变，仍然是赫夫曼编码。

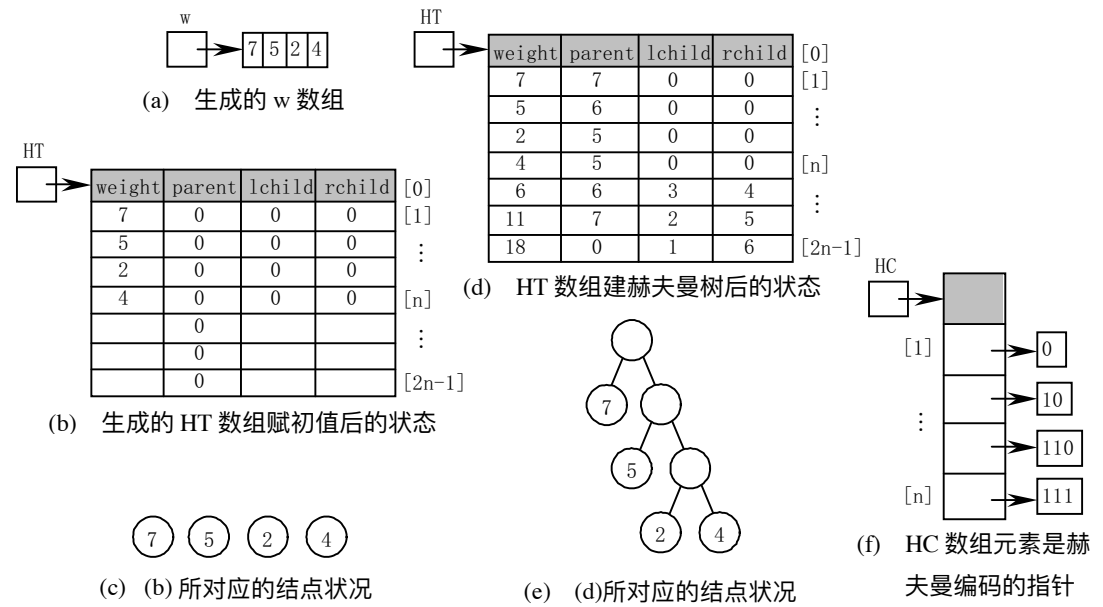


图 6－36　algo6-1.cpp 运行过程的图解

algo6-2.cpp 与 algo6-1.cpp 仅是函数 HuffmanCoding()的后半部分不同，也就是在建立好赫夫曼树后求赫夫曼编码的方法不同。algo6-1.cpp 是先找到叶子结点(左右孩子均为 0 的结点)，再根据其双亲结点逐步找到根结点来求赫夫曼编码的；而 algo6-2.cpp 是由根结点起，依次查找其左右孩子，直到找到叶子结点来求赫夫曼编码的。

```
// algo6-2.cpp 实现算法6.13的程序
#include"c1.h"
```

```
#include"c6-7.h"
#include"func6-1.cpp"
void HuffmanCoding(HuffmanTree &HT,HuffmanCode &HC,int *w,int n) // 前半部分为算法6.12
{ // w存放n个字符的权值(均>0)，构造赫夫曼树HT，并求出n个字符的赫夫曼编码HC
  int m,i,s1,s2; // 此句与algo6-1.cpp不同
  unsigned c,cdlen; // 此句与algo6-1.cpp不同
  HuffmanTree p;
  char *cd;
  if(n<=1)
    return;
  m=2*n-1;
  HT=(HuffmanTree)malloc((m+1)*sizeof(HTNode)); // 0号单元未用
  for(p=HT+1,i=1;i<=n;++i,++p,++w)
  {
    (*p).weight=*w;
    (*p).parent=0;
    (*p).lchild=0;
    (*p).rchild=0;
  }
  for(;i<=m;++i,++p)
    (*p).parent=0;
  for(i=n+1;i<=m;++i) // 建赫夫曼树
  { // 在HT[1～i-1]中选择parent为0且weight最小的两个结点，其序号分别为s1和s2
    select(HT,i-1,s1,s2);
    HT[s1].parent=HT[s2].parent=i;
    HT[i].lchild=s1;
    HT[i].rchild=s2;
    HT[i].weight=HT[s1].weight+HT[s2].weight;
  }
  // 以下为算法6.13，无栈非递归遍历赫夫曼树，求赫夫曼编码，以上同算法6.12
  HC=(HuffmanCode)malloc((n+1)*sizeof(char*));
  // 分配n个字符编码的头指针向量([0]不用)
  cd=(char*)malloc(n*sizeof(char)); // 分配求编码的工作空间
  c=m;
  cdlen=0;
  for(i=1;i<=m;++i)
    HT[i].weight=0; // 遍历赫夫曼树时用作结点状态标志
  while(c)
  {
    if(HT[c].weight==0)
    { // 向左
      HT[c].weight=1;
      if(HT[c].lchild!=0)
      {
        c=HT[c].lchild;
        cd[cdlen++]='0';
      }
      else if(HT[c].rchild==0)
      { // 登记叶子结点的字符的编码
        HC[c]=(char *)malloc((cdlen+1)*sizeof(char));
        cd[cdlen]='\0';
        strcpy(HC[c],cd); // 复制编码(串)
```

```
      }
    }
    else if(HT[c].weight==1)
    { // 向右
      HT[c].weight=2;
      if(HT[c].rchild!=0)
      {
        c=HT[c].rchild;
        cd[cdlen++]='1';
      }
    }
    else
    { // HT[c].weight==2，退回
      HT[c].weight=0;
      c=HT[c].parent;
      --cdlen; // 退到父结点，编码长度减1
    }
  }
  free(cd);
}
void main()
{ // 主程序同algo6-1.cpp
  HuffmanTree HT;
  HuffmanCode HC;
  int *w,n,i;
  printf("请输入权值的个数(>1): ");
  scanf("%d",&n);
  w=(int *)malloc(n*sizeof(int));
  printf("请依次输入%d个权值(整型):\n",n);
  for(i=0;i<=n-1;i++)
    scanf("%d",w+i);
  HuffmanCoding(HT,HC,w,n);
  for(i=1;i<=n;i++)
    puts(HC[i]);
}
```

程序运行结果(以教科书例 6－2 为例):

```
请输入权值的个数(>1): 8↙
请依次输入8个权值(整型):
5 29 7 8 14 23 3 11↙
0110
10
1110
1111
110
00
0111
010
```

第 7 章　图

7.1　图的定义和术语

7.2　图的存储结构

图是比较复杂的数据结构，它由顶点和顶点之间的弧或边组成。任何两个顶点之间都可能存在弧或边。在计算机存储图时，只要能表示出顶点的个数及每个顶点的特征、每对顶点之间是否存在弧(边)及弧(边)的特征，就能表示出图的所有信息，并作为图的一种存储结构。本章介绍了 4 种图的存储结构，它们各有特点。

7.2.1　数组表示法

```
// c7-1.h 图的数组(邻接矩阵)存储结构(见图7－1)
#define INFINITY INT_MAX // 用整型最大值代替∞
#define MAX_VERTEX_NUM 26 // 最大顶点个数
enum GraphKind{DG,DN,UDG,UDN};
                    // {有向图,有向网,无向图,无向网}
typedef struct
{
  VRType adj; // 顶点关系类型。对无权图,
              // 用1(是)或0(否)表示相邻否;
              // 对带权图，则为权值
  InfoType *info; // 该弧相关信息的指针(可无)
}ArcCell,AdjMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];
                          // 二维数组
struct MGraph
{
  VertexType vexs[MAX_VERTEX_NUM]; // 顶点向量
  AdjMatrix arcs; // 邻接矩阵
  int vexnum,arcnum; // 图的当前顶点数和弧数
  GraphKind kind; // 图的种类标志
};
```

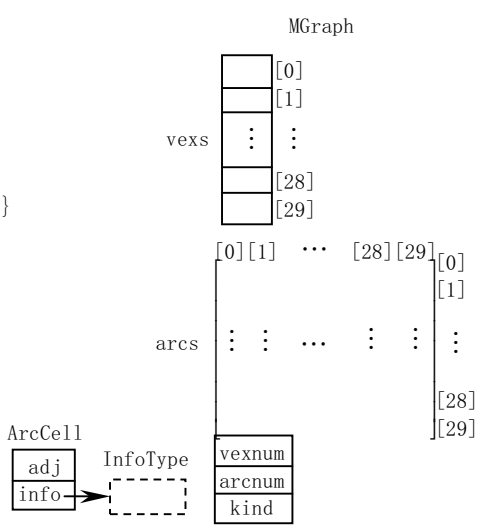
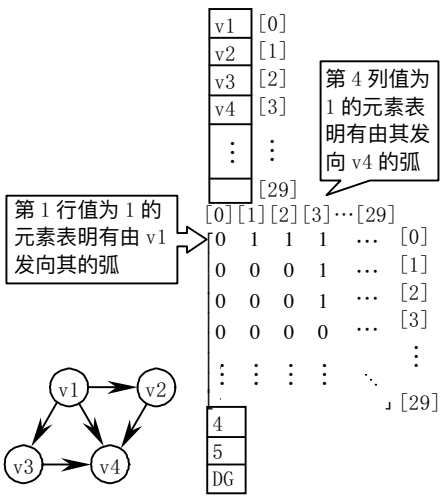


图 7－1　图的数组(邻接矩阵)存储结构

c7-1.h 中，结构体 MGraph 中的顶点向量是 VertexType 类型。一般地，我们都在主程序中定义 VertexType 类型为字符串类型，表示顶点名称(见 main7-1.cpp)。VertexType 类型也可以是结构体。对于 AOV-网(见教科书 7.5.1 节)，顶点不仅包括名称，还包括活

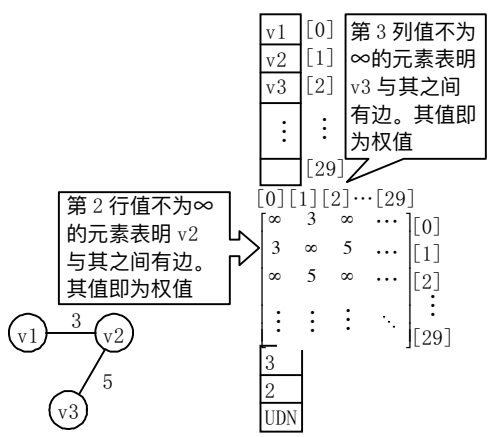
动，所以要用结构体表示。

图 7－2 是根据 c7-1.h 定义的有向图的存储结构。vexs[]数组存放各顶点的信息，arcs[][]数组存放各顶点邻接关系信息(是否互为邻接点)，如果 1 条弧从第 i 个顶点发出，终止于第 j 个顶点，则 arcs[i][j]=1。如图 7－2(b)所示，arcs[0][1]=1，说明从 v1 到 v2 有 1 条弧。设对角元素(arcs[i][i])的邻接关系为 0，则 arcs[][]数组中值为 1 的元素的个数等于有向图的弧数。图 7－3 是根据 c7-1.h 定义的无向网(网也称为带权图)的存储结构。同图 7－2 一样，图 7－3 中的 vexs[]数组仍存放各顶点的信息，arcs[][]数组存放各顶点邻接关系信息。对于网，顶点互为邻接点，则其值为权值；否则其值为∞。设对角元素(arcs[i][i])的邻接关系为∞，如果在第 i 个顶点和第 j 个顶点之间有边(无向)，则 arcs[i][j]= arcs[j][i]=权值。如图 7－3(b)所示，arcs[0][1]= arcs[1][0]=3，说明在 v1、v2 之间有 1 条边，其权值为 3。无向图或网的二维数组是以主对角线为轴对称的，对称的两个单元表示同一条边。arcs[][]数组中值不为∞的元素的个数等于无向网边数的 2 倍。



(a) 有向图(无相关信息)　(b) 存储结构

图 7－2　有向图的数组(邻接矩阵)存储结构



(a) 无向网(无相关信息)　(b) 存储结构(对称矩阵)

图 7－3　无向网的数组(邻接矩阵)存储结构

在这种数组(邻接矩阵)存储结构中，数组所占用的存储空间与图的弧或边数无关，故适用于边数较多的稠密图。

```
// bo7-1.cpp 图的数组(邻接矩阵)存储(存储结构由c7-1.h定义)的基本操作(21个)，包括算法7.1、
// 7.2和7.4～7.6
int LocateVex(MGraph G,VertexType u)
{ // 初始条件：图G存在，u和G中顶点有相同特征
  // 操作结果：若G中存在顶点u，则返回该顶点在图中位置；否则返回-1
  int i;
  for(i=0;i<G.vexnum;++i)
    if(strcmp(u,G.vexs[i])==0)
      return i;
  return -1;
}
```

```
void CreateFUDG(MGraph &G)
{ // 采用数组(邻接矩阵)表示法，由文件构造没有相关信息的无向图G
  int i,j,k;
  char filename[13];
  VertexType va,vb;
  FILE *graphlist;
  printf("请输入数据文件名(f7-1.txt)：");
  scanf("%s",filename);
  graphlist=fopen(filename,"r"); // 打开数据文件，并以graphlist表示
  fscanf(graphlist,"%d",&G.vexnum);
  fscanf(graphlist,"%d",&G.arcnum);
  for(i=0;i<G.vexnum;++i) // 构造顶点向量
    fscanf(graphlist,"%s",G.vexs[i]);
  for(i=0;i<G.vexnum;++i) // 初始化邻接矩阵
    for(j=0;j<G.vexnum;++j)
    {
      G.arcs[i][j].adj=0; // 图
      G.arcs[i][j].info=NULL; // 没有相关信息
    }
  for(k=0;k<G.arcnum;++k)
  {
    fscanf(graphlist,"%s%s",va,vb);
    i=LocateVex(G,va);
    j=LocateVex(G,vb);
    G.arcs[i][j].adj=G.arcs[j][i].adj=1; // 无向图
  }
  fclose(graphlist); // 关闭数据文件
  G.kind=UDG;
}
void CreateFUDN(MGraph &G)
{ // 采用数组(邻接矩阵)表示法，由文件构造没有相关信息的无向网G
  int i,j,k,w;
  char filename[13];
  VertexType va,vb;
  FILE *graphlist;
  printf("请输入数据文件名：");
  scanf("%s",filename);
  graphlist=fopen(filename,"r"); // 打开数据文件，并以graphlist表示
  fscanf(graphlist,"%d",&G.vexnum);
  fscanf(graphlist,"%d",&G.arcnum);
  for(i=0;i<G.vexnum;++i) // 构造顶点向量
    fscanf(graphlist,"%s",G.vexs[i]);
  for(i=0;i<G.vexnum;++i) // 初始化邻接矩阵
    for(j=0;j<G.vexnum;++j)
    {
      G.arcs[i][j].adj=INFINITY; // 网
      G.arcs[i][j].info=NULL; // 没有相关信息
    }
  for(k=0;k<G.arcnum;++k)
  {
    fscanf(graphlist,"%s%s%d",va,vb,&w);
```

```
    i=LocateVex(G,va);
    j=LocateVex(G,vb);
    G.arcs[i][j].adj=G.arcs[j][i].adj=w; // 无向网
  }
  fclose(graphlist); // 关闭数据文件
  G.kind=UDN;
}
void CreateDG(MGraph &G)
{ // 采用数组(邻接矩阵)表示法，构造有向图G
  int i,j,k,l,IncInfo;
  char s[MAX_INFO];
  VertexType va,vb;
  printf("请输入有向图G的顶点数,弧数,弧是否含其它信息(是:1,否:0): ");
  scanf("%d,%d,%d",&G.vexnum,&G.arcnum,&IncInfo);
  printf("请输入%d个顶点的值(<%d个字符):\n",G.vexnum,MAX_NAME);
  for(i=0;i<G.vexnum;++i) // 构造顶点向量
    scanf("%s",G.vexs[i]);
  for(i=0;i<G.vexnum;++i) // 初始化邻接矩阵
    for(j=0;j<G.vexnum;++j)
    {
      G.arcs[i][j].adj=0; // 图
      G.arcs[i][j].info=NULL;
    }
  printf("请输入%d条弧的弧尾 弧头(以空格作为间隔): \n",G.arcnum);
  for(k=0;k<G.arcnum;++k)
  {
    scanf("%s%s%*c",va,vb);  // %*c吃掉回车符
    i=LocateVex(G,va);
    j=LocateVex(G,vb);
    G.arcs[i][j].adj=1; // 有向图
    if(IncInfo)
    {
      printf("请输入该弧的相关信息(<%d个字符): ",MAX_INFO);
      gets(s);
      l=strlen(s);
      if(l)
      {
        G.arcs[i][j].info=(char*)malloc((l+1)*sizeof(char)); // 有向
        strcpy(G.arcs[i][j].info,s);
      }
    }
  }
  G.kind=DG;
}
void CreateDN(MGraph &G)
{ // 采用数组(邻接矩阵)表示法，构造有向网G
  int i,j,k,w,IncInfo;
  char s[MAX_INFO];
  VertexType va,vb;
  printf("请输入有向网G的顶点数,弧数,弧是否含其它信息(是:1,否:0): ");
  scanf("%d,%d,%d",&G.vexnum,&G.arcnum,&IncInfo);
```

```
  printf("请输入%d个顶点的值(<%d个字符):\n",G.vexnum,MAX_NAME);
  for(i=0;i<G.vexnum;++i) // 构造顶点向量
    scanf("%s",G.vexs[i]);
  for(i=0;i<G.vexnum;++i) // 初始化邻接矩阵
    for(j=0;j<G.vexnum;++j)
    {
      G.arcs[i][j].adj=INFINITY; // 网
      G.arcs[i][j].info=NULL;
    }
  printf("请输入%d条弧的弧尾 弧头 权值(以空格作为间隔): \n",G.arcnum);
  for(k=0;k<G.arcnum;++k)
  {
    scanf("%s%s%d%*c",va,vb,&w);  // %*c吃掉回车符
    i=LocateVex(G,va);
    j=LocateVex(G,vb);
    G.arcs[i][j].adj=w; // 有向网
    if(IncInfo)
    {
      printf("请输入该弧的相关信息(<%d个字符): ",MAX_INFO);
      gets(s);
      w=strlen(s);
      if(w)
      {
        G.arcs[i][j].info=(char*)malloc((w+1)*sizeof(char)); // 有向
        strcpy(G.arcs[i][j].info,s);
      }
    }
  }
  G.kind=DN;
}
void CreateUDG(MGraph &G)
{ // 采用数组(邻接矩阵)表示法，构造无向图G
  int i,j,k,l,IncInfo;
  char s[MAX_INFO];
  VertexType va,vb;
  printf("请输入无向图G的顶点数,边数,边是否含其它信息(是:1,否:0): ");
  scanf("%d,%d,%d",&G.vexnum,&G.arcnum,&IncInfo);
  printf("请输入%d个顶点的值(<%d个字符):\n",G.vexnum,MAX_NAME);
  for(i=0;i<G.vexnum;++i) // 构造顶点向量
    scanf("%s",G.vexs[i]);
  for(i=0;i<G.vexnum;++i) // 初始化邻接矩阵
    for(j=0;j<G.vexnum;++j)
    {
      G.arcs[i][j].adj=0; // 图
      G.arcs[i][j].info=NULL;
    }
  printf("请输入%d条边的顶点1 顶点2(以空格作为间隔): \n",G.arcnum);
  for(k=0;k<G.arcnum;++k)
  {
    scanf("%s%s%*c",va,vb); // %*c吃掉回车符
    i=LocateVex(G,va);
```

```
    j=LocateVex(G,vb);
    G.arcs[i][j].adj=G.arcs[j][i].adj=1; // 无向图
    if(IncInfo)
    {
      printf("请输入该边的相关信息(<%d个字符): ",MAX_INFO);
      gets(s);
      l=strlen(s);
      if(l)
      {
        G.arcs[i][j].info=G.arcs[j][i].info=(char*)malloc((l+1)*sizeof(char));
        // 无向，两个指针指向同一个信息
        strcpy(G.arcs[i][j].info,s);
      }
    }
  }
  G.kind=UDG;
}
void CreateUDN(MGraph &G)
{ // 采用数组(邻接矩阵)表示法，构造无向网G。算法7.2
  int i,j,k,w,IncInfo;
  char s[MAX_INFO];
  VertexType va,vb;
  printf("请输入无向网G的顶点数,边数,边是否含其它信息(是:1,否:0): ");
  scanf("%d,%d,%d",&G.vexnum,&G.arcnum,&IncInfo);
  printf("请输入%d个顶点的值(<%d个字符):\n",G.vexnum,MAX_NAME);
  for(i=0;i<G.vexnum;++i) // 构造顶点向量
    scanf("%s",G.vexs[i]);
  for(i=0;i<G.vexnum;++i) // 初始化邻接矩阵
    for(j=0;j<G.vexnum;++j)
    {
      G.arcs[i][j].adj=INFINITY; // 网
      G.arcs[i][j].info=NULL;
    }
  printf("请输入%d条边的顶点1 顶点2 权值(以空格作为间隔): \n",G.arcnum);
  for(k=0;k<G.arcnum;++k)
  {
    scanf("%s%s%d%*c",va,vb,&w); // %*c吃掉回车符
    i=LocateVex(G,va);
    j=LocateVex(G,vb);
    G.arcs[i][j].adj=G.arcs[j][i].adj=w; // 无向
    if(IncInfo)
    {
      printf("请输入该边的相关信息(<%d个字符): ",MAX_INFO);
      gets(s);
      w=strlen(s);
      if(w)
      {
        G.arcs[i][j].info=G.arcs[j][i].info=(char*)malloc((w+1)*sizeof(char));
        // 无向，两个指针指向同一个信息
        strcpy(G.arcs[i][j].info,s);
      }
```

```
    }
  }
  G.kind=UDN;
}
void CreateGraph(MGraph &G)
{ // 采用数组(邻接矩阵)表示法，构造图G。算法7.1改
  printf("请输入图G的类型(有向图:0,有向网:1,无向图:2,无向网:3): ");
  scanf("%d",&G.kind);
  switch(G.kind)
  {
    case DG: CreateDG(G); // 构造有向图
             break;
    case DN: CreateDN(G); // 构造有向网
             break;
    case UDG:CreateUDG(G); // 构造无向图
             break;
    case UDN:CreateUDN(G); // 构造无向网
  }
}
void DestroyGraph(MGraph &G)
{ // 初始条件：图G存在。操作结果：销毁图G
  int i,j,k=0;
  if(G.kind%2) // 网
    k=INFINITY; // k为两顶点之间无边或弧时邻接矩阵元素的值
  for(i=0;i<G.vexnum;i++) // 释放弧或边的相关信息(如果有的话)
    if(G.kind<2) // 有向
    {
      for(j=0;j<G.vexnum;j++)
        if(G.arcs[i][j].adj!=k) // 有弧
          if(G.arcs[i][j].info) // 有相关信息
          {
            free(G.arcs[i][j].info);
            G.arcs[i][j].info=NULL;
          }
    } // 加括号为避免if-else对配错
    else // 无向
      for(j=i+1;j<G.vexnum;j++) // 只查上三角
        if(G.arcs[i][j].adj!=k) // 有边
          if(G.arcs[i][j].info) // 有相关信息
          {
            free(G.arcs[i][j].info);
            G.arcs[i][j].info=G.arcs[j][i].info=NULL;
          }
  G.vexnum=0; // 顶点数为0
  G.arcnum=0; // 边数为0
}
VertexType& GetVex(MGraph G,int v)
{ // 初始条件：图G存在，v是G中某个顶点的序号。操作结果：返回v的值
  if(v>=G.vexnum||v<0)
    exit(ERROR);
  return G.vexs[v];
```

```
}
Status PutVex(MGraph &G,VertexType v,VertexType value)
{ // 初始条件：图G存在，v是G中某个顶点。操作结果：对v赋新值value
  int k;
  k=LocateVex(G,v); // k为顶点v在图G中的序号
  if(k<0)
    return ERROR;
  strcpy(G.vexs[k],value);
  return OK;
}
int FirstAdjVex(MGraph G,VertexType v)
{ // 初始条件：图G存在，v是G中某个顶点
  // 操作结果：返回v的第一个邻接顶点的序号。若顶点在G中没有邻接顶点，则返回-1
  int i,j=0,k;
  k=LocateVex(G,v); // k为顶点v在图G中的序号
  if(G.kind%2) // 网
    j=INFINITY;
  for(i=0;i<G.vexnum;i++)
    if(G.arcs[k][i].adj!=j)
      return i;
  return -1;
}
int NextAdjVex(MGraph G,VertexType v,VertexType w)
{ // 初始条件：图G存在，v是G中某个顶点，w是v的邻接顶点
  // 操作结果：返回v的(相对于w的)下一个邻接顶点的序号，若w是v的最后一个邻接顶点，则返回-1
  int i,j=0,k1,k2;
  k1=LocateVex(G,v); // k1为顶点v在图G中的序号
  k2=LocateVex(G,w); // k2为顶点w在图G中的序号
  if(G.kind%2) // 网
    j=INFINITY;
  for(i=k2+1;i<G.vexnum;i++)
    if(G.arcs[k1][i].adj!=j)
      return i;
  return -1;
}
void InsertVex(MGraph &G,VertexType v)
{ // 初始条件：图G存在，v和图G中顶点有相同特征
  // 操作结果：在图G中增添新顶点v(不增添与顶点相关的弧，留待InsertArc()去做)
  int i,j=0;
  if(G.kind%2) // 网
    j=INFINITY;
  strcpy(G.vexs[G.vexnum],v); // 构造新顶点向量
  for(i=0;i<=G.vexnum;i++)
  {
    G.arcs[G.vexnum][i].adj=G.arcs[i][G.vexnum].adj=j;
    // 初始化新增行、新增列邻接矩阵的值无边或弧)
    G.arcs[G.vexnum][i].info=G.arcs[i][G.vexnum].info=NULL; // 初始化相关信息指针
  }
  G.vexnum++; // 图G的顶点数加1
}
Status DeleteVex(MGraph &G,VertexType v)
```

```
{ // 初始条件：图G存在，v是G中某个顶点。操作结果：删除G中顶点v及其相关的弧
  int i,j,k;
  VRType m=0;
  if(G.kind%2) // 网
    m=INFINITY;
  k=LocateVex(G,v); // k为待删除顶点v的序号
  if(k<0) // v不是图G的顶点
    return ERROR;
  for(j=0;j<G.vexnum;j++)
    if(G.arcs[j][k].adj!=m) // 有入弧或边
    {
      if(G.arcs[j][k].info) // 有相关信息
        free(G.arcs[j][k].info); // 释放相关信息
      G.arcnum--; // 修改弧数
    }
  if(G.kind<2) // 有向
    for(j=0;j<G.vexnum;j++)
      if(G.arcs[k][j].adj!=m) // 有出弧
      {
        if(G.arcs[k][j].info) // 有相关信息
          free(G.arcs[k][j].info); // 释放相关信息
        G.arcnum--; // 修改弧数
      }
  for(j=k+1;j<G.vexnum;j++) // 序号k后面的顶点向量依次前移
    strcpy(G.vexs[j-1],G.vexs[j]);
  for(i=0;i<G.vexnum;i++)
    for(j=k+1;j<G.vexnum;j++)
      G.arcs[i][j-1]=G.arcs[i][j]; // 移动待删除顶点之右的矩阵元素
  for(i=0;i<G.vexnum;i++)
    for(j=k+1;j<G.vexnum;j++)
      G.arcs[j-1][i]=G.arcs[j][i]; // 移动待删除顶点之下的矩阵元素
  G.vexnum--; // 更新图的顶点数
  return OK;
}
Status InsertArc(MGraph &G,VertexType v,VertexType w)
{ // 初始条件：图G存在，v和w是G中两个顶点
  // 操作结果：在G中增添弧<v,w>，若G是无向的，则还增添对称弧<w,v>
  int i,l,v1,w1;
  char s[MAX_INFO];
  v1=LocateVex(G,v); // 尾
  w1=LocateVex(G,w); // 头
  if(v1<0||w1<0)
    return ERROR;
  G.arcnum++; // 弧或边数加1
  if(G.kind%2) // 网
  {
    printf("请输入此弧或边的权值：");
    scanf("%d",&G.arcs[v1][w1].adj);
  }
  else // 图
    G.arcs[v1][w1].adj=1;
```

```
  printf("是否有该弧或边的相关信息(0:无 1:有): ");
  scanf("%d%*c",&i);
  if(i)
  {
    printf("请输入该弧或边的相关信息(<%d个字符): ",MAX_INFO);
    gets(s);
    l=strlen(s);
    if(l)
    {
      G.arcs[v1][w1].info=(char*)malloc((l+1)*sizeof(char));
      strcpy(G.arcs[v1][w1].info,s);
    }
  }
  if(G.kind>1) // 无向
  {
    G.arcs[w1][v1].adj=G.arcs[v1][w1].adj;
    G.arcs[w1][v1].info=G.arcs[v1][w1].info; // 指向同一个相关信息
  }
  return OK;
}
Status DeleteArc(MGraph &G,VertexType v,VertexType w)
{ // 初始条件: 图G存在, v和w是G中两个顶点
  // 操作结果: 在G中删除弧<v,w>, 若G是无向的, 则还删除对称弧<w,v>
  int v1,w1,j=0;
  if(G.kind%2) // 网
    j=INFINITY;
  v1=LocateVex(G,v); // 尾
  w1=LocateVex(G,w); // 头
  if(v1<0||w1<0) // v1、w1的值不合法
    return ERROR;
  G.arcs[v1][w1].adj=j;
  if(G.arcs[v1][w1].info) // 有其它信息
  {
    free(G.arcs[v1][w1].info);
    G.arcs[v1][w1].info=NULL;
  }
  if(G.kind>=2) // 无向, 删除对称弧<w,v>
  {
    G.arcs[w1][v1].adj=j;
    G.arcs[w1][v1].info=NULL;
  }
  G.arcnum--; // 弧数-1
  return OK;
}
Boolean visited[MAX_VERTEX_NUM]; // 访问标志数组(全局量)
void(*VisitFunc)(VertexType); // 函数变量
void DFS(MGraph G,int v)
{ // 从第v个顶点出发递归地深度优先遍历图G。算法7.5
  int w;
  visited[v]=TRUE; // 设置访问标志为TRUE(已访问)
  VisitFunc(G.vexs[v]); // 访问第v个顶点
```

```
  for(w=FirstAdjVex(G,G.vexs[v]);w>=0;w=NextAdjVex(G,G.vexs[v],G.vexs[w]))
    if(!visited[w])
      DFS(G,w); // 对v的尚未访问的序号为w的邻接顶点递归调用DFS
}
void DFSTraverse(MGraph G,void(*Visit)(VertexType))
{ // 初始条件：图G存在，Visit是顶点的应用函数。算法7.4
  // 操作结果：从第1个顶点起，深度优先遍历图G，并对每个顶点调用函数Visit一次且仅一次
  int v;
  VisitFunc=Visit; // 使用全局变量VisitFunc，使DFS不必设函数指针参数
  for(v=0;v<G.vexnum;v++)
    visited[v]=FALSE; // 访问标志数组初始化(未被访问)
  for(v=0;v<G.vexnum;v++)
    if(!visited[v])
      DFS(G,v); // 对尚未访问的顶点v调用DFS
  printf("\n");
}
typedef VRType QElemType; // 队列元素类型
#include"c3-2.h" // 链队列的结构，BFSTraverse()用
#include"bo3-2.cpp" // 链队列的基本操作，BFSTraverse()用
void BFSTraverse(MGraph G,void(*Visit)(VertexType))
{ // 初始条件：图G存在，Visit是顶点的应用函数。算法7.6
  // 操作结果：从第1个顶点起，按广度优先非递归遍历图G,并对每个顶点调用函数Visit一次且仅一次
  int v,u,w;
  LinkQueue Q; // 使用辅助队列Q和访问标志数组visited
  for(v=0;v<G.vexnum;v++)
    visited[v]=FALSE; // 置初值
  InitQueue(Q); // 置空的辅助队列Q
  for(v=0;v<G.vexnum;v++)
    if(!visited[v]) // v尚未访问
    {
      visited[v]=TRUE; // 设置访问标志为TRUE(已访问)
      Visit(G.vexs[v]);
      EnQueue(Q,v); // v入队列
      while(!QueueEmpty(Q)) // 队列不空
      {
        DeQueue(Q,u); // 队头元素出队并置为u
        for(w=FirstAdjVex(G,G.vexs[u]);w>=0;w=NextAdjVex(G,G.vexs[u],G.vexs[w]))
          if(!visited[w]) // w为u的尚未访问的邻接顶点的序号
          {
            visited[w]=TRUE;
            Visit(G.vexs[w]);
            EnQueue(Q,w);
          }
      }
    }
  printf("\n");
}
void Display(MGraph G)
{ // 输出邻接矩阵存储结构的图G
  int i,j;
  char s[7];
```

```
  switch(G.kind)
  {
    case DG: strcpy(s,"有向图");
             break;
    case DN: strcpy(s,"有向网");
             break;
    case UDG:strcpy(s,"无向图");
             break;
    case UDN:strcpy(s,"无向网");
  }
  printf("%d个顶点%d条边或弧的%s。顶点依次是: ",G.vexnum,G.arcnum,s);
  for(i=0;i<G.vexnum;++i) // 输出G.vexs
    printf("%s ",G.vexs[i]);
  printf("\nG.arcs.adj:\n"); // 输出G.arcs.adj
  for(i=0;i<G.vexnum;i++)
  {
    for(j=0;j<G.vexnum;j++)
      printf("%11d",G.arcs[i][j].adj);
    printf("\n");
  }
  printf("G.arcs.info:\n"); // 输出G.arcs.info
  printf("顶点1(弧尾) 顶点2(弧头) 该边或弧的信息: \n");
  for(i=0;i<G.vexnum;i++)
    if(G.kind<2) // 有向
    {
      for(j=0;j<G.vexnum;j++)
        if(G.arcs[i][j].info)
          printf("%5s %11s     %s\n",G.vexs[i],G.vexs[j],G.arcs[i][j].info);
    } // 加括号为避免if-else对配错
    else // 无向,输出上三角
      for(j=i+1;j<G.vexnum;j++)
        if(G.arcs[i][j].info)
          printf("%5s %11s     %s\n",G.vexs[i],G.vexs[j],G.arcs[i][j].info);
}

// main7-1.cpp 检验bo7-1.cpp的主程序
#include"c1.h"
#define MAX_NAME 5 // 顶点字符串的最大长度+1
#define MAX_INFO 20 // 相关信息字符串的最大长度+1
typedef int VRType; // 顶点关系类型
typedef char InfoType; // 相关信息类型
typedef char VertexType[MAX_NAME]; // 顶点类型
#include"c7-1.h"
#include"bo7-1.cpp"
void visit(VertexType i)
{
  printf("%s ",i);
}
void main()
{
  int i,j,k,n;
```

```
  MGraph g;
  VertexType v1,v2;
  printf("请顺序选择有向图,有向网,无向图,无向网\n");
  for(i=0;i<4;i++) // 验证4种情况
  {
    CreateGraph(g); // 构造图g
    Display(g); // 输出图g
    printf("插入新顶点，请输入顶点的值: ");
    scanf("%s",v1);
    InsertVex(g,v1);
    printf("插入与新顶点有关的弧或边，请输入弧或边数: ");
    scanf("%d",&n);
    for(k=0;k<n;k++)
    {
      printf("请输入另一顶点的值: ");
      scanf("%s",v2);
      if(g.kind<=1) // 有向
      {
        printf("对于有向图或网，请输入另一顶点的方向(0:弧头 1:弧尾): ");
        scanf("%d",&j);
        if(j) // v2是弧尾
          InsertArc(g,v2,v1);
        else // v2是弧头
          InsertArc(g,v1,v2);
      }
      else // 无向
        InsertArc(g,v1,v2);
    }
    Display(g); // 输出图g
    printf("删除顶点及相关的弧或边，请输入顶点的值: ");
    scanf("%s",v1);
    DeleteVex(g,v1);
    Display(g); // 输出图g
  }
  DestroyGraph(g); // 销毁图g
}
```

程序运行结果：

```
请顺序选择有向图,有向网,无向图,无向网
请输入图G的类型(有向图:0,有向网:1,无向图:2,无向网:3): 0↙
请输入有向图G的顶点数,弧数,弧是否含其它信息(是:1,否:0): 2,1,0↙
请输入2个顶点的值(<5个字符):
a1 a2↙
请输入1条弧的弧尾 弧头(以空格作为间隔):
a2 a1↙
2个顶点1条边或弧的有向图。顶点依次是: a1 a2 (见图7-4)
G.arcs.adj:
          0          0
```

图 7-4　不含其它信息的有向图

```
          1          0
G.arcs.info:
顶点1(弧尾) 顶点2(弧头) 该边或弧的信息:
插入新顶点，请输入顶点的值: a3↙
插入与新顶点有关的弧或边，请输入弧或边数: 2↙
请输入另一顶点的值: a1↙
对于有向图或网，请输入另一顶点的方向(0:弧头 1:弧尾): 0↙
是否有该弧或边的相关信息(0:无 1:有): 0↙
请输入另一顶点的值: a2↙
对于有向图或网，请输入另一顶点的方向(0:弧头 1:弧尾): 1↙
是否有该弧或边的相关信息(0:无 1:有): 0↙
3个顶点3条边或弧的有向图。顶点依次是: a1 a2 a3 (见图7-5)
G.arcs.adj:
          0          0          0
          1          0          1
          1          0          0
G.arcs.info:
```

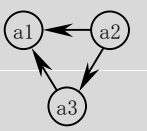


图 7-5　插入顶点 a3 后图示

```
顶点1(弧尾) 顶点2(弧头) 该边或弧的信息:
删除顶点及相关的弧或边，请输入顶点的值: a1↙
2个顶点1条边或弧的有向图。顶点依次是: a2 a3 (见图7-6)
G.arcs.adj:
          0          1
          0          0
G.arcs.info:
```

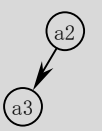


图 7-6　删除顶点 a1 后图示

```
顶点1(弧尾) 顶点2(弧头) 该边或弧的信息:
请输入图G的类型(有向图:0,有向网:1,无向图:2,无向网:3): 1↙
请输入有向网G的顶点数,弧数,弧是否含其它信息(是:1,否:0): 2,1,1↙
请输入2个顶点的值(<5个字符):
b1 b2↙
请输入1条弧的弧尾 弧头 权值(以空格作为间隔):
b1 b2 3↙
请输入该弧的相关信息(<20个字符): Good morning!↙
2个顶点1条边或弧的有向网。顶点依次是: b1 b2 (见图7-7)
G.arcs.adj:
      32767          3
      32767      32767
G.arcs.info:
```

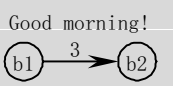


图 7-7　含其它信息的有向网

```
顶点1(弧尾) 顶点2(弧头) 该边或弧的信息:
   b1          b2     Good morning!
插入新顶点，请输入顶点的值: b3↙
插入与新顶点有关的弧或边，请输入弧或边数: 2↙
请输入另一顶点的值: b1↙
对于有向图或网，请输入另一顶点的方向(0:弧头 1:弧尾): 0↙
请输入此弧或边的权值: 5↙
是否有该弧或边的相关信息(0:无 1:有): 1↙
请输入该弧或边的相关信息(<20个字符): Good day!↙
请输入另一顶点的值: b2↙
对于有向图或网，请输入另一顶点的方向(0:弧头 1:弧尾): 1↙
请输入此弧或边的权值: 6↙
是否有该弧或边的相关信息(0:无 1:有): 1↙
请输入该弧或边的相关信息(<20个字符): Good bye!↙
```

```
3个顶点3条边或弧的有向网。顶点依次是: b1 b2 b3 (见图7-8)
G.arcs.adj:
      32767          3       32767
      32767      32767           6
          5      32767       32767
G.arcs.info:
顶点1(弧尾) 顶点2(弧头) 该边或弧的信息:
   b1          b2      Good morning!
   b2          b3      Good bye!
   b3          b1      Good day!
删除顶点及相关的弧或边, 请输入顶点的值: b2↙
2个顶点1条边或弧的有向网。顶点依次是: b1 b3 (见图7-9)
G.arcs.adj:
      32767      32767
          5      32767
G.arcs.info:
顶点1(弧尾) 顶点2(弧头) 该边或弧的信息:
   b3          b1      Good day!
请输入图G的类型(有向图:0,有向网:1,无向图:2,无向网:3): 2↙
请输入无向图G的顶点数,边数,边是否含其它信息(是:1,否:0): 2,1,1↙
请输入2个顶点的值(<5个字符):
c1 c2↙
请输入1条边的顶点1 顶点2(以空格作为间隔):
c1 c2↙
请输入该边的相关信息(<20个字符): good↙
2个顶点1条边或弧的无向图。顶点依次是: c1 c2 (见图7-10)
G.arcs.adj:
          0          1
          1          0
G.arcs.info:
顶点1(弧尾) 顶点2(弧头) 该边或弧的信息:
   c1          c2      good
插入新顶点, 请输入顶点的值: c3↙
插入与新顶点有关的弧或边, 请输入弧或边数: 2↙
请输入另一顶点的值: c1↙
是否有该弧或边的相关信息(0:无 1:有): 1↙
请输入该弧或边的相关信息(<20个字符): better↙
请输入另一顶点的值: c2↙
是否有该弧或边的相关信息(0:无 1:有): 1↙
请输入该弧或边的相关信息(<20个字符): best↙
3个顶点3条边或弧的无向图。顶点依次是: c1 c2 c3 (见图7-11)
G.arcs.adj:
          0          1          1
          1          0          1
          1          1          0
G.arcs.info:
顶点1(弧尾) 顶点2(弧头) 该边或弧的信息:
   c1          c2      good
   c1          c3      better
   c2          c3      best
删除顶点及相关的弧或边, 请输入顶点的值: c3↙
```

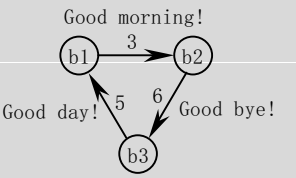


图 7-8　插入顶点 b3 后图示

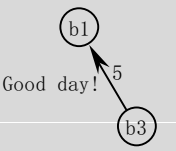


图 7-9　删除顶点 b2 后图示

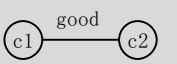


图 7-10　含其它信息的无向图

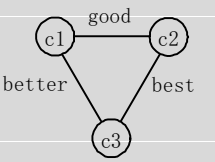


图 7-11　插入顶点 c3 后图示

```
2个顶点1条边或弧的无向图。顶点依次是: c1 c2 (见图7-12)
G.arcs.adj:
          0          1
          1          0
G.arcs.info:
顶点1(弧尾) 顶点2(弧头) 该边或弧的信息:
   c1          c2      good
请输入图G的类型(有向图:0,有向网:1,无向图:2,无向网:3): 3↙
请输入无向网G的顶点数,边数,边是否含其它信息(是:1,否:0): 2,1,0↙
请输入2个顶点的值(<5个字符):
d1 d2↙
请输入1条边的顶点1 顶点2 权值(以空格作为间隔):
d1 d2 5↙
2个顶点1条边或弧的无向网。顶点依次是: d1 d2 (见图7-13)
G.arcs.adj:
      32767          5
          5      32767
G.arcs.info:
顶点1(弧尾) 顶点2(弧头) 该边或弧的信息:
插入新顶点,请输入顶点的值: d3↙
插入与新顶点有关的弧或边,请输入弧或边数: 2↙
请输入另一顶点的值: d1↙
请输入此弧或边的权值: 4↙
是否有该弧或边的相关信息(0:无 1:有): 0↙
请输入另一顶点的值: d2↙
请输入此弧或边的权值: 6↙
是否有该弧或边的相关信息(0:无 1:有): 0↙
3个顶点3条边或弧的无向网。顶点依次是: d1 d2 d3 (见图7-14)
G.arcs.adj:
      32767          5          4
          5      32767          6
          4          6      32767
G.arcs.info:
顶点1(弧尾) 顶点2(弧头) 该边或弧的信息:
删除顶点及相关的弧或边,请输入顶点的值: d1↙
2个顶点1条边或弧的无向网。顶点依次是: d2 d3 (见图7-15)
G.arcs.adj:
      32767          6
          6      32767
G.arcs.info:
顶点1(弧尾) 顶点2(弧头) 该边或弧的信息:
```

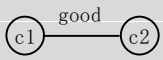


图 7-12　删除顶点 c3 后图示

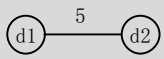


图 7-13　不含其它信息的无向网

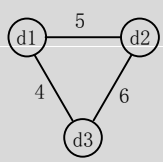


图 7-14　插入顶点 d3 后图示

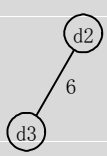


图 7-15　删除顶点 d1 后图示

7.2.2　邻接表

```
// c7-2.h 图的邻接表存储结构(见图7-16)
#define MAX_VERTEX_NUM 20
enum GraphKind{DG,DN,UDG,UDN}; // {有向图,有向网,无向图,无向网}
struct ArcNode
{
  int adjvex; // 该弧所指向的顶点的位置
  ArcNode *nextarc; // 指向下一条弧的指针
```

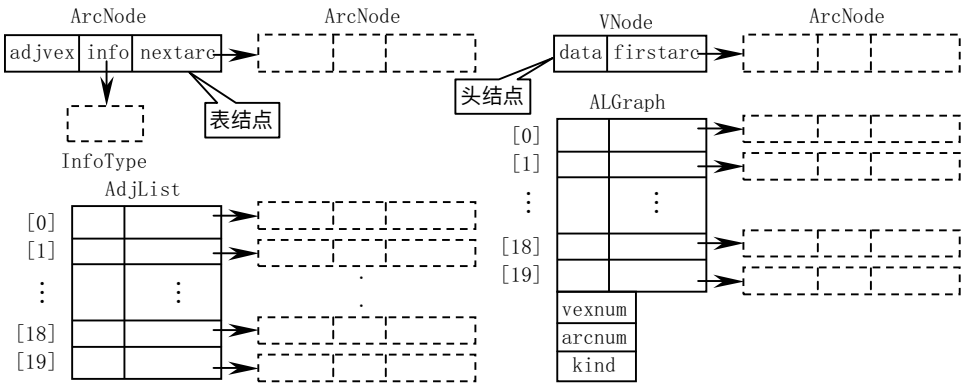
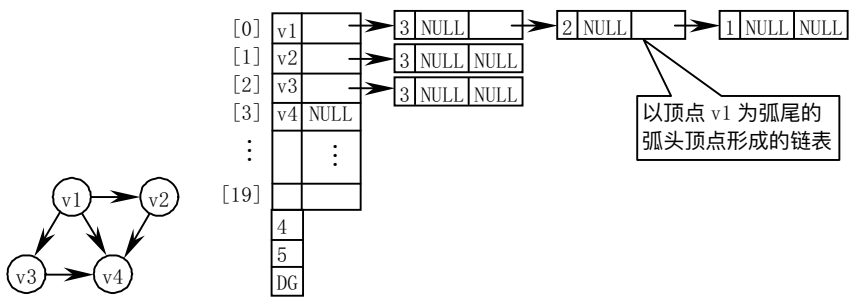


图 7－16　图的邻接表存储结构

```
  InfoType *info; // 网的权值指针
}; // 表结点
typedef struct
{
  VertexType data; // 顶点信息
  ArcNode *firstarc; // 第一个表结点的地址，指向第一条依附该顶点的弧的指针
}VNode,AdjList[MAX_VERTEX_NUM]; // 头结点
struct ALGraph
{
  AdjList vertices;
  int vexnum,arcnum; // 图的当前顶点数和弧数
  int kind; // 图的种类标志
};
```

图 7－17 和图 7－18 分别是有向图和无向网的的邻接表存储结构。要注意的是，为了提高效率，bo7-2.cpp 中的基本操作函数 CreateGraph()产生链表时总是在表头插入结点。所以，对于给定的图，即使它的顶点输入顺序相同，邻接表的存储结构也不惟一。邻接表的存储结构还与弧或边的输入顺序有关。

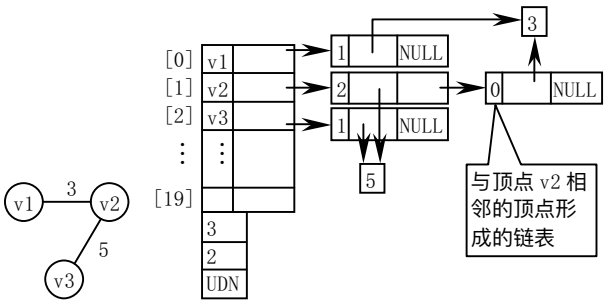


(a) 有向图(无相关信息)　　(b) 存储结构

图 7－17　有向图的邻接表存储结构

对于无向的图或网，每一条边产生 2 个表结点，分别在该边的 2 个顶点的链表上。由图 7－18 可见，2 条边的无向网有 4 个表结点。为简化，无向网的每条边只动态生成 1 个存放权值的存储空间，由两个结点共同指向。由于邻接表存储结构中的链表的长度与该顶

点的邻接出弧或边数相等，显然，图的邻接表存储结构适合存储弧或边相对较少的稀疏图。



(a) 无向网　　(b) 存储结构

图 7－18　无向网的邻接表存储结构

由邻接表的存储结构可见，每个顶点的信息由表示顶点名称的字符串和不带头结点的单链表组合而成。这样，对于单链表的处理，我们就可以利用不带头结点的单链表的基本操作(在 bo2-8.cpp 中)和扩展操作(在 func2-1.cpp 中)来简化编程。为了能够利用这些操作，需要将在 bo2-8.cpp 和 func2-1.cpp 中用到的类型 ElemType、LNode、LinkList 和邻接表的类型 ArcNode 联系起来。c7-21.h 就是根据 c7-2.h 建立了这种联系。

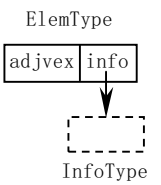


图 7－19　ElemType 类型

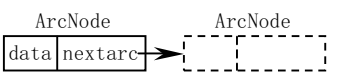


图 7－20　ArcNode 类型

```
// c7-21.h 图的邻接表存储结构(与单链表的变量类型建立联系)
#define MAX_VERTEX_NUM 20
enum GraphKind{DG,DN,UDG,UDN}; // {有向图,有向网,无向图,无向网}
struct ElemType // 加(见图7－19)
{
  int adjvex; // 该弧所指向的顶点的位置
  InfoType *info; // 网的权值指针
};
struct ArcNode // 改(见图7－20)
{
  ElemType data; // 除指针以外的部分都属于ElemType
  ArcNode *nextarc; // 指向下一条弧的指针
}; // 表结点
typedef struct
{
  VertexType data; // 顶点信息
  ArcNode *firstarc; // 第一个表结点的地址,指向第一条依附该顶点的弧的指针
}VNode,AdjList[MAX_VERTEX_NUM]; // 头结点
struct ALGraph
{
  AdjList vertices;
  int vexnum,arcnum; // 图的当前顶点数和弧数
  int kind; // 图的种类标志
};
#define LNode ArcNode // 加，定义单链表的结点类型是图的表结点的类型
#define next nextarc // 加，定义单链表结点的指针域是表结点指向下一条弧的指针域
typedef ArcNode *LinkList; // 加，定义指向单链表结点的指针是指向图的表结点的指针
```

```
// bo7-2.cpp 图的邻接表存储(存储结构由c7-21.h定义)的基本操作(15个)，包括算法7.4～7.6
#include"bo2-8.cpp" // 不带头结点的单链表基本操作
#include"func2-1.cpp" // 不带头结点的单链表扩展操作
int LocateVex(ALGraph G,VertexType u)
{ // 初始条件：图G存在，u和G中顶点有相同特征
  // 操作结果：若G中存在顶点u，则返回该顶点在图中位置；否则返回-1
  int i;
  for(i=0;i<G.vexnum;++i)
    if(strcmp(u,G.vertices[i].data)==0)
      return i;
  return -1;
}
void CreateGraph(ALGraph &G)
{ // 采用邻接表存储结构，构造没有相关信息图或网G(用一个函数构造4种图)
  int i,j,k,w; // w是权值
  VertexType va,vb; // 连接边或弧的2顶点
  ElemType e;
  printf("请输入图的类型(有向图:0,有向网:1,无向图:2,无向网:3)：");
  scanf("%d",&G.kind);
  printf("请输入图的顶点数,边数：");
  scanf("%d,%d",&G.vexnum,&G.arcnum);
  printf("请输入%d个顶点的值(<%d个字符):\n",G.vexnum,MAX_NAME);
  for(i=0;i<G.vexnum;++i) // 构造顶点向量
  {
    scanf("%s",G.vertices[i].data);
    G.vertices[i].firstarc=NULL; // 初始化与该顶点有关的出弧链表
  }
  if(G.kind%2) // 网
    printf("请输入每条弧(边)的权值、弧尾和弧头(以空格作为间隔):\n");
  else // 图
    printf("请输入每条弧(边)的弧尾和弧头(以空格作为间隔):\n");
  for(k=0;k<G.arcnum;++k) // 构造相关弧链表
  {
    if(G.kind%2) // 网
      scanf("%d%s%s",&w,va,vb);
    else // 图
      scanf("%s%s",va,vb);
    i=LocateVex(G,va); // 弧尾
    j=LocateVex(G,vb); // 弧头
    e.info=NULL; // 给待插表结点e赋值，图无权
    e.adjvex=j; // 弧头
    if(G.kind%2) // 网
    {
      e.info=(int *)malloc(sizeof(int)); // 动态生成存放权值的空间
      *(e.info)=w;
    }
    ListInsert(G.vertices[i].firstarc,1,e); // 插在第i个元素(出弧)的表头，在bo2-8.cpp中
    if(G.kind>=2) // 无向图或网，产生第2个表结点，并插在第j个元素(入弧)的表头
    {
      e.adjvex=i; // e.info不变，不必再赋值
```

```
      ListInsert(G.vertices[j].firstarc,1,e); // 插在第j个元素的表头，在bo2-8.cpp中
    }
  }
}
void CreateGraphF(ALGraph &G)
{ // 采用邻接表存储结构，由文件构造没有相关信息图或网G(用一个函数构造4种图)
  int i,j,k,w; // w是权值
  VertexType va,vb; // 连接边或弧的2顶点
  ElemType e;
  char filename[13];
  FILE *graphlist;
  printf("请输入数据文件名(f7-1.txt或f7-2.txt): ");
  scanf("%s",filename);
  printf("请输入图的类型(有向图:0,有向网:1,无向图:2,无向网:3): ");
  scanf("%d",&G.kind);
  graphlist=fopen(filename,"r"); // 以读的方式打开数据文件，并以graphlist表示
  fscanf(graphlist,"%d",&G.vexnum);
  fscanf(graphlist,"%d",&G.arcnum);
  for(i=0;i<G.vexnum;++i) // 构造顶点向量
  {
    fscanf(graphlist,"%s",G.vertices[i].data);
    G.vertices[i].firstarc=NULL; // 初始化与该顶点有关的出弧链表
  }
  for(k=0;k<G.arcnum;++k) // 构造相关弧链表
  {
    if(G.kind%2) // 网
      fscanf(graphlist,"%d%s%s",&w,va,vb);
    else // 图
      fscanf(graphlist,"%s%s",va,vb);
    i=LocateVex(G,va); // 弧尾
    j=LocateVex(G,vb); // 弧头
    e.info=NULL; // 给待插表结点e赋值，图无权
    e.adjvex=j; // 弧头
    if(G.kind%2) // 网
    {
      e.info=(int *)malloc(sizeof(int)); // 动态生成存放权值的空间
      *(e.info)=w;
    }
    ListInsert(G.vertices[i].firstarc,1,e); // 插在第i个元素(出弧)的表头，在bo2-8.cpp中
    if(G.kind>=2) // 无向图或网，产生第2个表结点，并插在第j个元素(入弧)的表头
    {
      e.adjvex=i; // e.info不变，不必再赋值
      ListInsert(G.vertices[j].firstarc,1,e); // 插在第j个元素的表头，在bo2-8.cpp中
    }
  }
  fclose(graphlist); // 关闭数据文件
}
void DestroyGraph(ALGraph &G)
{ // 初始条件: 图G存在。操作结果: 销毁图G
  int i;
  ElemType e;
```

```
  for(i=0;i<G.vexnum;++i) // 对于所有顶点
    if(G.kind%2) // 网
      while(G.vertices[i].firstarc) // 对应的弧或边链表不空
      {
        ListDelete(G.vertices[i].firstarc,1,e); // 删除链表的第1个结点，并将值赋给e
        if(e.adjvex>i) // 顶点序号>i(保证动态生成的权值空间只释放1次)
          free(e.info);
      }
    else // 图
      DestroyList(G.vertices[i].firstarc); // 销毁弧或边链表，在bo2-8.cpp中
  G.vexnum=0; // 顶点数为0
  G.arcnum=0; // 边或弧数为0
}
VertexType& GetVex(ALGraph G,int v)
{ // 初始条件：图G存在，v是G中某个顶点的序号。操作结果：返回v的值
  if(v>=G.vexnum||v<0)
    exit(ERROR);
  return G.vertices[v].data;
}
Status PutVex(ALGraph &G,VertexType v,VertexType value)
{ // 初始条件：图G存在，v是G中某个顶点。操作结果：对v赋新值value
  int i;
  i=LocateVex(G,v);
  if(i>-1) // v是G的顶点
  {
    strcpy(G.vertices[i].data,value);
    return OK;
  }
  return ERROR;
}
int FirstAdjVex(ALGraph G,VertexType v)
{ // 初始条件：图G存在，v是G中某个顶点
  // 操作结果：返回v的第一个邻接顶点的序号。若顶点在G中没有邻接顶点，则返回-1
  ArcNode *p;
  int v1;
  v1=LocateVex(G,v); // v1为顶点v在图G中的序号
  p=G.vertices[v1].firstarc;
  if(p)
    return p->data.adjvex;
  else
    return -1;
}
Status equalvex(ElemType a,ElemType b)
{ // DeleteArc()、DeleteVex()和NextAdjVex()要调用的函数
  if(a.adjvex==b.adjvex)
    return OK;
  else
    return ERROR;
}
int NextAdjVex(ALGraph G,VertexType v,VertexType w)
{ // 初始条件：图G存在，v是G中某个顶点，w是v的邻接顶点
```

```
  // 操作结果：返回v的(相对于w的)下一个邻接顶点的序号。若w是v的最后一个邻接点，则返回-1
  LinkList p,p1; // p1在Point()中用作辅助指针，Point()在func2-1.cpp中
  ElemType e;
  int v1;
  v1=LocateVex(G,v); // v1为顶点v在图G中的序号
  e.adjvex=LocateVex(G,w); // e.adjvex为顶点w在图G中的序号
  p=Point(G.vertices[v1].firstarc,e,equalvex,p1); // p指向顶点v的链表中邻接顶点为w的结点
  if(!p||!p->next) // 没找到w或w是最后一个邻接点
    return -1;
  else // p->data.adjvex==w
    return p->next->data.adjvex; // 返回v的(相对于w的)下一个邻接顶点的序号
}
void InsertVex(ALGraph &G,VertexType v)
{ // 初始条件：图G存在，v和图中顶点有相同特征
  // 操作结果：在图G中增添新顶点v(不增添与顶点相关的弧，留待InsertArc()去做)
  strcpy(G.vertices[G.vexnum].data,v); // 构造新顶点向量
  G.vertices[G.vexnum].firstarc=NULL;
  G.vexnum++; // 图G的顶点数加1
}
Status DeleteVex(ALGraph &G,VertexType v)
{ // 初始条件：图G存在，v是G中某个顶点。操作结果：删除G中顶点v及其相关的弧
  int i,j,k;
  ElemType e;
  LinkList p,p1;
  j=LocateVex(G,v); // j是顶点v的序号
  if(j<0) // v不是图G的顶点
    return ERROR;
  i=ListLength(G.vertices[j].firstarc); // 以v为出度的弧或边数，在bo2-8.cpp中
  G.arcnum-=i; // 边或弧数-i
  if(G.kind%2) // 网
    while(G.vertices[j].firstarc) // 对应的弧或边链表不空
    {
      ListDelete(G.vertices[j].firstarc,1,e); // 删除链表的第1个结点，并将值赋给e
      free(e.info); // 释放动态生成的权值空间
    }
  else // 图
    DestroyList(G.vertices[j].firstarc); // 销毁弧或边链表，在bo2-8.cpp中
  G.vexnum--; // 顶点数减1
  for(i=j;i<G.vexnum;i++) // 顶点v后面的顶点前移
    G.vertices[i]=G.vertices[i+1];
  for(i=0;i<G.vexnum;i++) // 删除以v为入度的弧或边且必要时修改表结点的顶点位置值
  {
    e.adjvex=j;
    p=Point(G.vertices[i].firstarc,e,equalvex,p1); // Point()在func2-1.cpp中
    if(p) // 顶点i的邻接表上有v为入度的结点
    {
      if(p1) // p1指向p所指结点的前驱
        p1->next=p->next; // 从链表中删除p所指结点
      else // p指向首元结点
        G.vertices[i].firstarc=p->next; // 头指针指向下一结点
      if(G.kind<2) // 有向
```

```
      {
        G.arcnum--; // 边或弧数-1
        if(G.kind==1) // 有向网
          free(p->data.info); // 释放动态生成的权值空间
      }
      free(p); // 释放v为入度的结点
    }
    for(k=j+1;k<=G.vexnum;k++) // 对于adjvex域>j的结点，其序号-1
    {
      e.adjvex=k;
      p=Point(G.vertices[i].firstarc,e,equalvex,p1); // Point()在func2-1.cpp中
      if(p)
        p->data.adjvex--; // 序号-1(因为前移)
    }
  }
  return OK;
}
Status InsertArc(ALGraph &G,VertexType v,VertexType w)
{ // 初始条件：图G存在，v和w是G中两个顶点
  // 操作结果：在G中增添弧<v,w>，若G是无向的，则还增添对称弧<w,v>
  ElemType e;
  int i,j;
  i=LocateVex(G,v); // 弧尾或边的序号
  j=LocateVex(G,w); // 弧头或边的序号
  if(i<0||j<0)
    return ERROR;
  G.arcnum++; // 图G的弧或边的数目加1
  e.adjvex=j;
  e.info=NULL; // 初值
  if(G.kind%2) // 网
  {
    e.info=(int *)malloc(sizeof(int)); // 动态生成存放权值的空间
    printf("请输入弧(边)%s→%s的权值：",v,w);
    scanf("%d",e.info);
  }
  ListInsert(G.vertices[i].firstarc,1,e); // 将e插在弧尾的表头，在bo2-8.cpp中
  if(G.kind>=2) // 无向，生成另一个表结点
  {
    e.adjvex=i; // e.info不变
    ListInsert(G.vertices[j].firstarc,1,e); // 将e插在弧头的表头
  }
  return OK;
}
Status DeleteArc(ALGraph &G,VertexType v,VertexType w)
{ // 初始条件：图G存在，v和w是G中两个顶点
  // 操作结果：在G中删除弧<v,w>，若G是无向的，则还删除对称弧<w,v>
  int i,j;
  Status k;
  ElemType e;
  i=LocateVex(G,v); // i是顶点v(弧尾)的序号
  j=LocateVex(G,w); // j是顶点w(弧头)的序号
```

```
  if(i<0||j<0||i==j)
    return ERROR;
  e.adjvex=j;
  k=DeleteElem(G.vertices[i].firstarc,e,equalvex); // 在func2-1.cpp中
  if(k) // 删除成功
  {
    G.arcnum--; // 弧或边数减1
    if(G.kind%2) // 网
      free(e.info);
    if(G.kind>=2) // 无向，删除对称弧<w,v>
    {
      e.adjvex=i;
      DeleteElem(G.vertices[j].firstarc,e,equalvex);
    }
    return OK;
  }
  else // 没找到待删除的弧
    return ERROR;
}
Boolean visited[MAX_VERTEX_NUM]; // 访问标志数组(全局量)
void(*VisitFunc)(char* v); // 函数变量(全局量)
void DFS(ALGraph G,int v)
{ // 从第v个顶点出发递归地深度优先遍历图G。算法7.5
  int w;
  visited[v]=TRUE; // 设置访问标志为TRUE(已访问)
  VisitFunc(G.vertices[v].data); // 访问第v个顶点
  for(w=FirstAdjVex(G,G.vertices[v].data);w>=0;w=NextAdjVex(G,G.vertices[v].data,
  G.vertices[w].data))
    if(!visited[w])
      DFS(G,w); // 对v的尚未访问的邻接点w递归调用DFS
}
void DFSTraverse(ALGraph G,void(*Visit)(char*))
{ // 对图G作深度优先遍历。算法7.4
  int v;
  VisitFunc=Visit; // 使用全局变量VisitFunc，使DFS不必设函数指针参数
  for(v=0;v<G.vexnum;v++)
    visited[v]=FALSE; // 访问标志数组初始化
  for(v=0;v<G.vexnum;v++)
    if(!visited[v])
      DFS(G,v); // 对尚未访问的顶点调用DFS
  printf("\n");
}
typedef int QElemType; // 队列元素类型
#include"c3-2.h" // 链队列的存储结构
#include"bo3-2.cpp" // 链队列的基本操作
void BFSTraverse(ALGraph G,void(*Visit)(char*))
{//按广度优先非递归遍历图G。使用辅助队列Q和访问标志数组visited。算法7.6
  int v,u,w;
  LinkQueue Q;
  for(v=0;v<G.vexnum;++v)
    visited[v]=FALSE; // 置初值
```

```
  InitQueue(Q); // 置空的辅助队列Q
  for(v=0;v<G.vexnum;v++) // 如果是连通图，只v=0就遍历全图
    if(!visited[v]) // v尚未访问
    {
      visited[v]=TRUE;
      Visit(G.vertices[v].data);
      EnQueue(Q,v); // v入队列
      while(!QueueEmpty(Q)) // 队列不空
      {
        DeQueue(Q,u); // 队头元素出队并置为u
        for(w=FirstAdjVex(G,G.vertices[u].data);w>=0;w=NextAdjVex(G,G.vertices[u].data,
        G.vertices[w].data))
          if(!visited[w]) // w为u的尚未访问的邻接顶点
          {
            visited[w]=TRUE;
            Visit(G.vertices[w].data);
            EnQueue(Q,w); // w入队
          }
      }
    }
  printf("\n");
}
void DFS1(ALGraph G,int v,void(*Visit)(char*))
{ // 从第v个顶点出发递归地深度优先遍历图G。仅适用于邻接表存储结构
  ArcNode *p; // p指向表结点
  visited[v]=TRUE; // 设置访问标志为TRUE(已访问)
  Visit(G.vertices[v].data); // 访问该顶点
  for(p=G.vertices[v].firstarc;p;p=p->next) // p依次指向v的邻接顶点
    if(!visited[p->data.adjvex])
      DFS1(G,p->data.adjvex,Visit); // 对v的尚未访问的邻接点递归调用DFS1
}
void DFSTraverse1(ALGraph G,void(*Visit)(char*))
{ // 对图G作深度优先遍历。DFS1设函数指针参数
  int v;
  for(v=0;v<G.vexnum;v++)
    visited[v]=FALSE; // 访问标志数组初始化，置初值为未被访问
  for(v=0;v<G.vexnum;v++) // 如果是连通图，只v=0就遍历全图
    if(!visited[v]) // v尚未被访问
      DFS1(G,v,Visit); // 对v调用DFS1
  printf("\n");
}
void BFSTraverse1(ALGraph G,void(*Visit)(char*))
{ // 按广度优先非递归遍历图G。使用辅助队列Q和访问标志数组visited。仅适用于邻接表结构
  int v,u;
  ArcNode *p; // p指向表结点
  LinkQueue Q; // 链队列类型
  for(v=0;v<G.vexnum;++v)
    visited[v]=FALSE; // 置初值为未被访问
  InitQueue(Q); // 初始化辅助队列Q
  for(v=0;v<G.vexnum;v++) // 如果是连通图，只v=0就遍历全图
    if(!visited[v]) // v尚未被访问
```

```
    {
      visited[v]=TRUE; // 设v为已被访问
      Visit(G.vertices[v].data); // 访问v
      EnQueue(Q,v); // v入队列
      while(!QueueEmpty(Q)) // 队列不空
      {
        DeQueue(Q,u); // 队头元素出队并置为u
        for(p=G.vertices[u].firstarc;p;p=p->next) // p依次指向u的邻接顶点
          if(!visited[p->data.adjvex]) // u的邻接顶点尚未被访问
          {
            visited[p->data.adjvex]=TRUE; // 该邻接顶点设为已被访问
            Visit(G.vertices[p->data.adjvex].data); // 访问该邻接顶点
            EnQueue(Q,p->data.adjvex); // 入队该邻接顶点序号
          }
      }
    }
  printf("\n");
}
void Display(ALGraph G)
{ // 输出图的邻接矩阵G
  int i;
  ArcNode *p;
  switch(G.kind)
  {
    case DG: printf("有向图\n");
             break;
    case DN: printf("有向网\n");
             break;
    case UDG:printf("无向图\n");
             break;
    case UDN:printf("无向网\n");
  }
  printf("%d个顶点: \n",G.vexnum);
  for(i=0;i<G.vexnum;++i)
    printf("%s ",G.vertices[i].data);
  printf("\n%d条弧(边):\n",G.arcnum);
  for(i=0;i<G.vexnum;i++)
  {
    p=G.vertices[i].firstarc;
    while(p)
    {
      if(G.kind<=1||i<p->data.adjvex) // 有向或无向两次中的一次
      {
        printf("%s→%s ",G.vertices[i].data,G.vertices[p->data.adjvex].data);
        if(G.kind%2) // 网
          printf(":%d ",*(p->data.info));
      }
      p=p->nextarc;
    }
    printf("\n");
  }
```

```
}

// main7-2.cpp 检验bo7-2.cpp的主程序
#include"c1.h"
#define MAX_NAME 3 // 顶点字符串的最大长度+1
typedef int InfoType; // 网的权值类型
typedef char VertexType[MAX_NAME]; // 顶点类型为字符串
#include"c7-21.h"
#include"bo7-2.cpp"
void print(char *i)
{
  printf("%s ",i);
}
void main()
{
  int i,j,k,n;
  ALGraph g;
  VertexType v1,v2;
  printf("请顺序选择有向图,有向网,无向图,无向网\n");
  for(i=0;i<4;i++) // 验证4种情况
  {
    CreateGraph(g);
    Display(g);
    printf("插入新顶点，请输入顶点的值: ");
    scanf("%s",v1);
    InsertVex(g,v1);
    printf("插入与新顶点有关的弧或边，请输入弧或边数: ");
    scanf("%d",&n);
    for(k=0;k<n;k++)
    {
      printf("请输入另一顶点的值: ");
      scanf("%s",v2);
      if(g.kind<=1) // 有向
      {
        printf("对于有向图或网，请输入另一顶点的方向(0:弧头 1:弧尾): ");
        scanf("%d",&j);
        if(j)
          InsertArc(g,v2,v1);
        else
          InsertArc(g,v1,v2);
      }
      else // 无向
        InsertArc(g,v1,v2);
    }
    Display(g);
    if(i==3)
    {
      printf("删除一条边或弧，请输入待删除边或弧的弧尾 弧头: ");
      scanf("%s%s",v1,v2);
      DeleteArc(g,v1,v2);
      printf("修改顶点的值，请输入原值 新值: ");
```

```
        scanf("%s%s",v1,v2);
        PutVex(g,v1,v2);
      }
      printf("删除顶点及相关的弧或边，请输入顶点的值：");
      scanf("%s",v1);
      DeleteVex(g,v1);
      Display(g);
      DestroyGraph(g);
    }
}
```

程序运行结果：

```
请顺序选择有向图,有向网,无向图,无向网
请输入图的类型(有向图:0,有向网:1,无向图:2,无向网:3)：0↙
请输入图的顶点数,边数：2,1↙
请输入2个顶点的值(<3个字符)：
a1 a2↙
请输入每条弧(边)的弧尾和弧头(以空格作为间隔)：
a2 a1↙
有向图(见图7－21)
2个顶点：
a1 a2
1条弧(边)：
a2→a1
插入新顶点，请输入顶点的值：a3↙
插入与新顶点有关的弧或边，请输入弧或边数：2↙
请输入另一顶点的值：a1↙
对于有向图或网，请输入另一顶点的方向(0:弧头 1:弧尾)：0↙
请输入另一顶点的值：a2↙
对于有向图或网，请输入另一顶点的方向(0:弧头 1:弧尾)：1↙
有向图(见图7－22)
3个顶点：
a1 a2 a3
3条弧(边)：
a2→a3 a2→a1
a3→a1
删除顶点及相关的弧或边，请输入顶点的值：a1↙
有向图(见图7－23)
2个顶点：
a2 a3
1条弧(边)：
a2→a3

请输入图的类型(有向图:0,有向网:1,无向图:2,无向网:3)：1↙
请输入图的顶点数,边数：2,1↙
请输入2个顶点的值(<3个字符)：
```

图 7－21　有向图

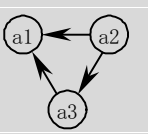


图 7－22　插入顶点 a3 后的有向图

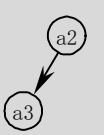


图 7－23　删除顶点 a1 后的有向图

b1 b2↙
请输入每条弧(边)的权值、弧尾和弧头(以空格作为间隔):
3 b1 b2↙
有向网(见图7－24)
2个顶点:
b1 b2
1条弧(边):
b1→b2 :3

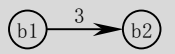


图 7－24　有向网

插入新顶点，请输入顶点的值: b3↙
插入与新顶点有关的弧或边，请输入弧或边数: 2↙
请输入另一顶点的值: b1↙
对于有向图或网，请输入另一顶点的方向(0:弧头 1:弧尾): 0↙
请输入弧(边)b3→b1的权值: 5↙
请输入另一顶点的值: b2↙
对于有向图或网，请输入另一顶点的方向(0:弧头 1:弧尾): 1↙
请输入弧(边)b2→b3的权值: 6↙
有向网(见图7－25)
3个顶点:
b1 b2 b3
3条弧(边):
b1→b2 :3
b2→b3 :6
b3→b1 :5

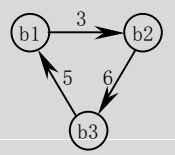


图 7－25　插入顶点 b3 后的有向网

删除顶点及相关的弧或边，请输入顶点的值: b2↙
有向网(见图7－26)
2个顶点:
b1 b3
1条弧(边):
b3→b1 :5

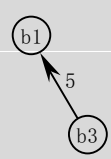


图 7－26　删除顶点 b2 后的有向网

请输入图的类型(有向图:0,有向网:1,无向图:2,无向网:3): 2↙
请输入图的顶点数,边数: 2,1↙
请输入2个顶点的值(<3个字符):
c1 c2↙
请输入每条弧(边)的弧尾和弧头(以空格作为间隔):
c1 c2↙
无向图(见图7－27)
2个顶点:
c1 c2
1条弧(边):
c1→c2

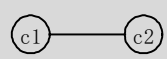


图 7－27　无向图

插入新顶点，请输入顶点的值: c3↙
插入与新顶点有关的弧或边，请输入弧或边数: 2↙
请输入另一顶点的值: c1↙
请输入另一顶点的值: c2↙
无向图(见图7－28)
3个顶点:
c1 c2 c3
3条弧(边):

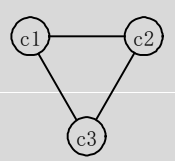


图 7－28　插入顶点 c3 后的无向图

```
c1→c3 c1→c2
c2→c3

删除顶点及相关的弧或边，请输入顶点的值：c3↙
无向图(见图7－29)
2个顶点：
c1 c2
1条弧(边)：
c1→c2

请输入图的类型(有向图:0,有向网:1,无向图:2,无向网:3)：3↙
请输入图的顶点数,边数：2,1↙
请输入2个顶点的值(<3个字符)：
d1 d2↙
请输入每条弧(边)的权值、弧尾和弧头(以空格作为间隔)：
5 d1 d2↙
无向网(见图7－30)
2个顶点：
d1 d2
1条弧(边)：
d1→d2 :5

插入新顶点，请输入顶点的值：d3↙
插入与新顶点有关的弧或边，请输入弧或边数：2↙
请输入另一顶点的值：d1↙
请输入弧(边)d3→d1的权值：4↙
请输入另一顶点的值：d2↙
请输入弧(边)d3→d2的权值：6↙
无向网(见图7－31)
3个顶点：
d1 d2 d3
3条弧(边)：
d1→d3 :4 d1→d2 :5
d2→d3 :6

删除一条边或弧，请输入待删除边或弧的弧尾 弧头：d1 d2↙
修改顶点的值，请输入原值 新值：d1 D1↙
删除顶点及相关的弧或边，请输入顶点的值：d2↙
无向网(见图7－32)
2个顶点：
D1 d3
1条弧(边)：
D1→d3 :4
```

图 7－29　删除顶点 c3 后的无向图

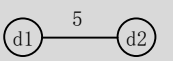


图 7－30　无向网

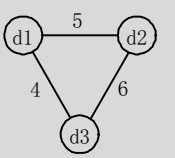


图 7－31　插入顶点 d3 后的无向网

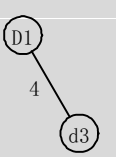


图 7－32　删除顶点 d2 后的无向网

7.2.3　十字链表

```
// c7-3.h 有向图的十字链表存储结构(见图7－33)
#define MAX_VERTEX_NUM 20
struct ArcBox // 弧结点
{
```

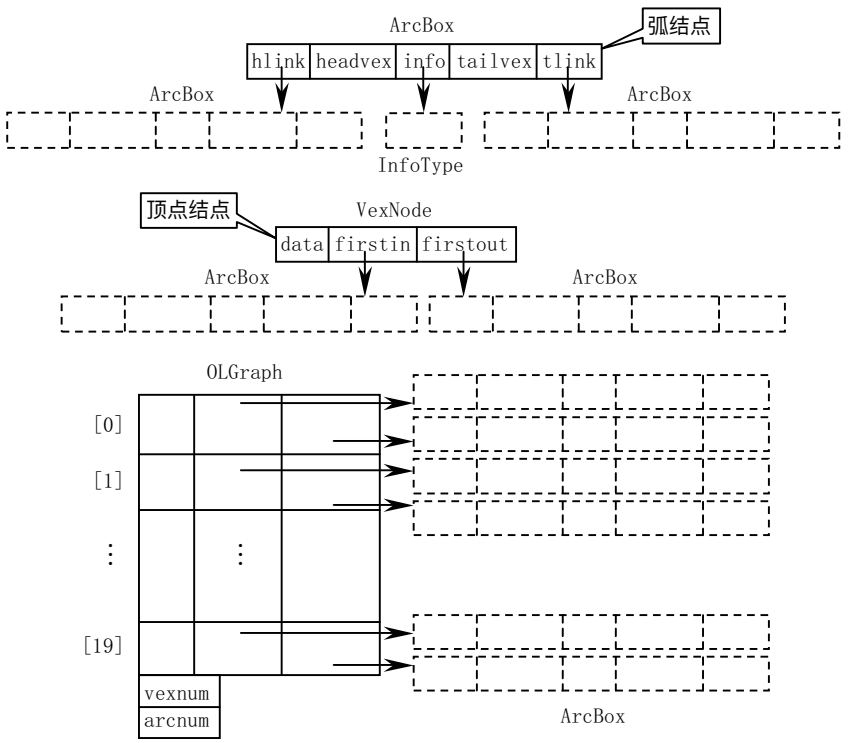


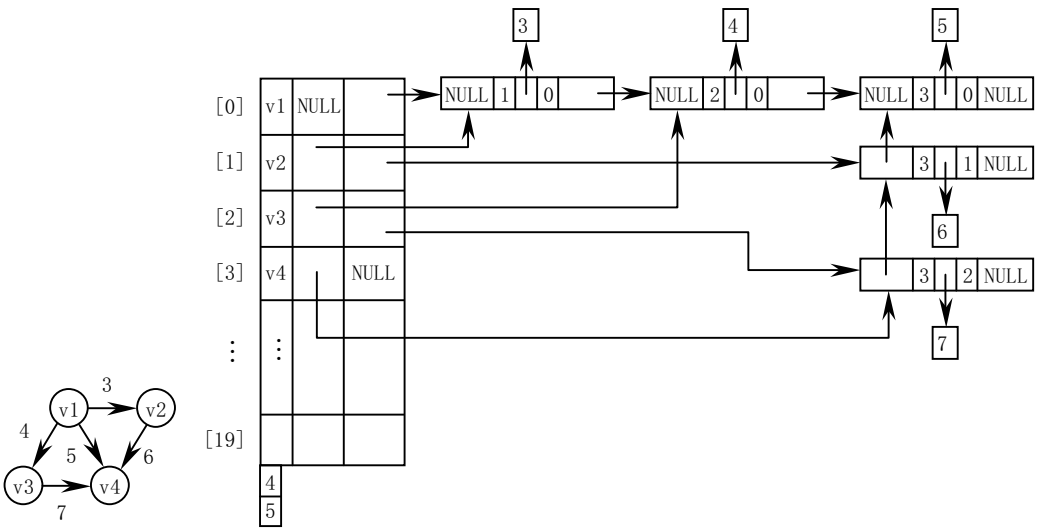
图 7－33　有向图的十字链表存储结构

```
  int tailvex,headvex; // 该弧的尾和头顶点的位置
  ArcBox *hlink,*tlink; // 分别为弧头相同和弧尾相同的弧的链域
  InfoType *info; // 该弧相关信息的指针，可指向权值或其他信息
};
struct VexNode // 顶点结点
{
  VertexType data;
  ArcBox *firstin,*firstout; // 分别指向该顶点第一条入弧和出弧
};
struct OLGraph
{
  VexNode xlist[MAX_VERTEX_NUM]; // 表头向量(数组)
  int vexnum,arcnum; // 有向图的当前顶点数和弧数
};
```

图 7－34 是根据 c7-3.h 定义的带权有向图(有向网)的存储结构。和邻接表相比，十字链表不仅有出弧链表，还有入弧链表，而且不增加链表结点的数量。所以，十字链表在既需要用到出弧，也需要用到入弧的情况下是很方便的。和邻接表结构同样的原因，由于 bo7-3.cpp 中基本操作函数 CreateDG()总是在入弧和出弧的表头插入弧结点，所以，对于给定的图，它的弧结点的链表结构也不惟一，而是与弧的输入顺序有关。

从存储结构上看，十字链表的出弧链表和入弧链表也是不带头结点的单链表结构。为了能够利用不带头结点的单链表的基本操作(在 bo2-8.cpp 中)，需要将单链表的类型 ElemType、LNode、LinkList 和十字链表的类型 ArcBox 联系起来。c7-31.h 就是根据

c7-3.h 建立了这种联系。



(a) 带权有向图(网)　　(b) 存储结构

图 7－34　有向图的十字链表存储结构

```
// c7-31.h 有向图的十字链表存储结构(与单链表的变量类型建立联系)
#define MAX_VERTEX_NUM 20
struct ArcBox1 // 用来定义hlink的类型(见图7－35)
{
  int tailvex,headvex; // 该弧的尾和头顶点的位置
  InfoType *info; // 该弧相关信息的指针，可指向权值或其它信息
  ArcBox1 *hlink,*tlink; // 分别为弧头相同弧尾相同的弧的链域
};
```

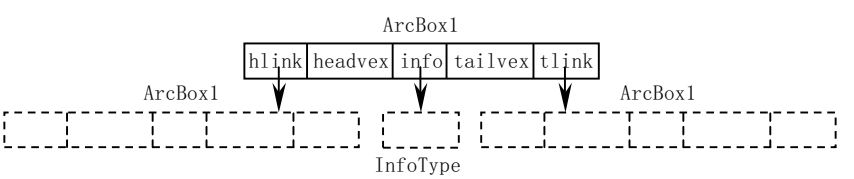


图 7－35　ArcBox1 的存储结构

```
struct ElemType(见图7－36)
{
  int tailvex,headvex; // 该弧的尾和头顶点的位置
  InfoType *info; // 该弧相关信息的指针，可指向权值或其它信息
  ArcBox1 *hlink;
};
struct ArcBox(见图7－37)
{
  ElemType data;
  ArcBox *tlink;
};
struct VexNode // 顶点结点(见图7－33)
{
  VertexType data;
```

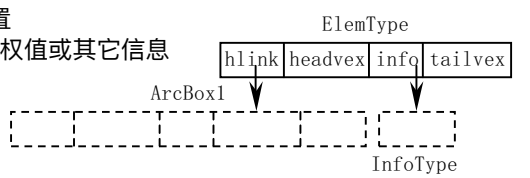


图 7－36　ElemType 的存储结构

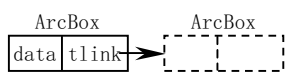


图 7－37　ArcBox 类型

```
  ArcBox1 *firstin; // 指向该顶点第一条入弧
  ArcBox *firstout; // 指向该顶点第一条出弧
};
struct OLGraph(见图7－33)
{
  VexNode xlist[MAX_VERTEX_NUM]; // 表头向量(数组)
  int vexnum,arcnum; // 有向图的当前顶点数和弧数
};
#define LNode ArcBox // 加，定义单链表的结点类型是图的表结点的类型
#define next tlink // 加，定义单链表结点的指针域是表结点指向下一条出弧的指针域
typedef ArcBox *LinkList; // 加，定义指向单链表结点的指针是指向图的表结点的指针
```

由于十字链表有 2 个指向结点的指针，因此它与单链表建立联系就比邻接表要复杂一些。c7-31.h 中的结构体 ArcBox1 和 ArcBox 的内部存储结构是一样的，可以通过强制类型转换，使一种类型的指针指向另一种类型，但它们必须是不同的类型。bo2-8.cpp 中的操作只能用于 ArcBox 类型和 VexNode 的出弧链表。

```
// bo7-3.cpp 有向图或网的十字链表存储(存储结构由c7-31.h定义)的基本函数(16个)，包括算法7.3、
// 7.4、7.6
#include"bo2-8.cpp" // 不带头结点的单链表基本操作
int LocateVex(OLGraph G,VertexType u)
{ // 返回顶点u在有向图G中的位置(序号)，如不存在则返回-1
  int i;
  for(i=0;i<G.vexnum;++i) // 用循环查找该结点
    if(!strcmp(G.xlist[i].data,u))
      return i;
  return -1;
}
void CreateDG(OLGraph &G)
{ // 采用十字链表存储结构，构造有向图G。算法7.3
  int i,j,k;
  int IncInfo;
  ArcBox *p;
  VertexType v1,v2;
  printf("请输入有向图的顶点数,弧数,是否为带权图(是:1，否:0)：");
  scanf("%d,%d,%d",&G.vexnum,&G.arcnum,&IncInfo);
  printf("请输入%d个顶点的值(<%d个字符):\n",G.vexnum,MAX_VERTEX_NAME);
  for(i=0;i<G.vexnum;++i)
  { // 构造表头向量
    scanf("%s",&G.xlist[i].data); // 输入顶点值
    G.xlist[i].firstin=NULL; // 初始化入弧的链表头指针
    G.xlist[i].firstout=NULL; // 初始化出弧的链表头指针
  }
  printf("请输入%d条弧的弧尾和弧头(空格为间隔):\n",G.arcnum);
  for(k=0;k<G.arcnum;++k)
  { // 输入各弧并构造十字链表
    scanf("%s%s",&v1,&v2);
    i=LocateVex(G,v1); // 确定v1和v2在G中的位置
```

```
    j=LocateVex(G,v2);
    p=(ArcBox *)malloc(sizeof(ArcBox)); // 产生弧结点(假定有足够空间)
    p->data.tailvex=i; // 对弧结点赋值
    p->data.headvex=j;
    p->data.hlink=G.xlist[j].firstin; // 完成在入弧和出弧链表表头的插入
    p->tlink=G.xlist[i].firstout;
    G.xlist[j].firstin=(ArcBox1 *)p; // 强制类型转换
    G.xlist[i].firstout=p;
    if(IncInfo)
    { // 是网
      p->data.info=(InfoType *)malloc(sizeof(InfoType));
      printf("请输入该弧的权值: ");
      scanf("%d",p->data.info);
    }
    else // 弧不含有相关信息
      p->data.info=NULL;
  }
}
void DestroyGraph(OLGraph &G)
{ // 初始条件: 有向图G存在。操作结果: 销毁有向图G
  int i;
  ElemType e;
  for(i=0;i<G.vexnum;i++) // 对所有顶点
    while(G.xlist[i].firstout) // 出弧链表不空
    {
      ListDelete(G.xlist[i].firstout,1,e); // 删除其第1个结点, 其值赋给e, 在bo2-8.cpp中
      if(e.info) // 带权
        free(e.info); // 释放动态生成的权值空间
    }
  G.arcnum=0;
  G.vexnum=0;
}
VertexType& GetVex(OLGraph G,int v)
{ // 初始条件: 有向图G存在, v是G中某个顶点的序号。操作结果: 返回v的值
  if(v>=G.vexnum||v<0)
    exit(ERROR);
  return G.xlist[v].data;
}
Status PutVex(OLGraph &G,VertexType v,VertexType value)
{ // 初始条件: 有向图G存在, v是G中某个顶点。操作结果: 对v赋新值value
  int i;
  i=LocateVex(G,v);
  if(i<0) // v不是G的顶点
    return ERROR;
  strcpy(G.xlist[i].data,value);
  return OK;
}
int FirstAdjVex(OLGraph G,VertexType v)
{ // 初始条件: 有向图G存在, v是G中某个顶点
  // 操作结果: 返回v的第一个邻接顶点的序号。若顶点在G中没有邻接顶点, 则返回-1
```

```
  int i;
  ArcBox *p;
  i=LocateVex(G,v);
  p=G.xlist[i].firstout; // p指向顶点v的第1个出弧
  if(p)
    return p->data.headvex;
  else
    return -1;
}
int NextAdjVex(OLGraph G,VertexType v,VertexType w)
{ // 初始条件：有向图G存在,v是G中某个顶点，w是v的邻接顶点
  // 操作结果：返回v的(相对于w的)下一个邻接顶点的序号，若w是v的最后一个邻接顶点，则返回-1
  int i,j;
  ArcBox *p;
  i=LocateVex(G,v); // i是顶点v的序号
  j=LocateVex(G,w); // j是顶点w的序号
  p=G.xlist[i].firstout; // p指向顶点v的第1个出弧
  while(p&&p->data.headvex!=j)
    p=p->tlink;
  if(p) // w不是v的最后一个邻接顶点
    p=p->tlink; // p指向相对于w的下一个邻接顶点
  if(p) // 有下一个邻接顶点
    return p->data.headvex;
  else
    return -1;
}
void InsertVex(OLGraph &G,VertexType v)
{ // 初始条件：有向图G存在，v和有向图G中顶点有相同特征
  // 操作结果：在有向图G中增添新顶点v(不增添与顶点相关的弧，留待InsertArc()去做)
  strcpy(G.xlist[G.vexnum].data,v); // 拷贝顶点名称
  G.xlist[G.vexnum].firstin=NULL; // 初始化入弧链表
  G.xlist[G.vexnum].firstout=NULL; // 初始化出弧链表
  G.vexnum++; // 顶点数+1
}
Status equal(ElemType c1,ElemType c2)
{
  if(c1.headvex==c2.headvex)
    return TRUE;
  else
    return FALSE;
}
Status DeleteVex(OLGraph &G,VertexType v)
{ // 初始条件：有向图G存在，v是G中某个顶点。操作结果：删除G中顶点v及其相关的弧
  int i,j,k;
  ElemType e1,e2;
  ArcBox *p;
  ArcBox1 *p1,*p2;
  k=LocateVex(G,v); // k是顶点v的序号
  if(k<0) // v不是图G的顶点
    return ERROR; // 以下删除顶点v的入弧
```

```
  e1.headvex=k; // e1作为LocateElem()的比较元素
  for(j=0;j<G.vexnum;j++) // 顶点v的入弧是其它顶点的出弧
  {
    i=LocateElem(G.xlist[j].firstout,e1,equal);
    if(i) // 顶点v是顶点j的出弧
    {
      ListDelete(G.xlist[j].firstout,i,e2); // 删除该弧结点，其值赋给e2
      G.arcnum--; // 弧数-1
      if(e2.info) // 带权
        free(e2.info); // 释放动态生成的权值空间
    }
  } // 以下删除顶点v的出弧
  for(j=0;j<G.vexnum;j++) // 顶点v的出弧是其它顶点的入弧
  {
    p1=G.xlist[j].firstin;
    while(p1&&p1->tailvex!=k)
    {
      p2=p1;
      p1=p1->hlink;
    }
    if(p1) // 找到顶点v的出弧
    {
      if(p1==G.xlist[j].firstin) // 是首结点
        G.xlist[j].firstin=p1->hlink; // 入弧指针指向下一个结点
      else // 不是首结点
        p2->hlink=p1->hlink; // 在链表中移去p1所指结点
      if(p1->info) // 带权
        free(p1->info); // 释放动态生成的权值空间
      free(p1); // 释放p1所指结点
      G.arcnum--; // 弧数-1
    }
  }
  for(j=k+1;j<G.vexnum;j++) // 序号>k的顶点依次向前移
    G.xlist[j-1]=G.xlist[j];
  G.vexnum--; // 顶点数减1
  for(j=0;j<G.vexnum;j++) // 结点序号>k的要减1
  {
    p=G.xlist[j].firstout; // 处理出弧
    while(p)
    {
      if(p->data.tailvex>k)
        p->data.tailvex--; // 序号-1
      if(p->data.headvex>k)
        p->data.headvex--; // 序号-1
      p=p->tlink;
    }
  }
  return OK;
}
Status InsertArc(OLGraph &G,VertexType v,VertexType w)
```

```
{ // 初始条件：有向图G存在，v和w是G中两个顶点。操作结果：在G中增添弧<v,w>
  int i,j;
  int IncInfo;
  ArcBox *p;
  i=LocateVex(G,v); // 弧尾的序号
  j=LocateVex(G,w); // 弧头的序号
  if(i<0||j<0)
    return ERROR;
  p=(ArcBox *)malloc(sizeof(ArcBox)); // 生成新结点
  p->data.tailvex=i; // 给新结点赋值
  p->data.headvex=j;
  p->data.hlink=G.xlist[j].firstin; // 插在入弧和出弧的链头
  p->tlink=G.xlist[i].firstout;
  G.xlist[j].firstin=(ArcBox1*)p;
  G.xlist[i].firstout=p;
  G.arcnum++; // 弧数加1
  printf("要插入的弧是否带权(是：1,否：0)：");
  scanf("%d",&IncInfo);
  if(IncInfo) // 带权
  {
    p->data.info=(InfoType *)malloc(sizeof(InfoType)); // 动态生成权值空间
    printf("请输入该弧的权值：");
    scanf("%d",p->data.info);
  }
  else
    p->data.info=NULL;
  return OK;
}
Status DeleteArc(OLGraph &G,VertexType v,VertexType w)
{ // 初始条件：有向图G存在，v和w是G中两个顶点。操作结果：在G中删除弧<v,w>
  int i,j,k;
  ElemType e;
  ArcBox1 *p1,*p2;
  i=LocateVex(G,v); // 弧尾的序号
  j=LocateVex(G,w); // 弧头的序号
  if(i<0||j<0||i==j)
    return ERROR;
  p1=G.xlist[j].firstin; // p1指向w的入弧链表
  while(p1&&p1->tailvex!=i) // 使p1指向待删结点
  {
    p2=p1;
    p1=p1->hlink;
  }
  if(p1==G.xlist[j].firstin) // 首结点是待删结点
    G.xlist[j].firstin=p1->hlink; // 入弧指针指向下一个结点
  else // 首结点不是待删结点
    p2->hlink=p1->hlink; // 在链表中移去p1所指结点(该结点仍在出弧链表中)
  e.headvex=j; // 待删弧结点的弧头顶点序号为j，e作为LocateElem()的比较元素
  k=LocateElem(G.xlist[i].firstout,e,equal); // 在出弧链表中的位序
  ListDelete(G.xlist[i].firstout,k,e); // 在出弧链表中删除该结点，其值赋给e
```

```
  if(e.info) // 带权
   free(e.info); // 释放动态生成的权值空间
  G.arcnum--; // 弧数-1
  return OK;
}
Boolean visited[MAX_VERTEX_NUM]; // 访问标志数组
void(*VisitFunc)(VertexType); // 函数变量
void DFS(OLGraph G,int i) // DFSTraverse()调用
{
  ArcBox *p;
  visited[i]=TRUE; // 访问标志数组置1(已被访问)
  VisitFunc(G.xlist[i].data); // 遍历第i个顶点
  p=G.xlist[i].firstout; // p指向第i个顶点的出度
  while(p&&visited[p->data.headvex]) // p没到表尾且该弧的头顶点已被访问
    p=p->tlink; // 查找下一个结点
  if(p&&!visited[p->data.headvex]) // 该弧的头顶点未被访问
    DFS(G,p->data.headvex); // 递归调用DFS()
}
void DFSTraverse(OLGraph G,void(*Visit)(VertexType))
{ // 初始条件: 有向图G存在, v是G中某个顶点, Visit是顶点的应用函数(算法7.4)
  // 操作结果:从第1个顶点起,按深度优先递归遍历有向图G,并对每个顶点调用函数Visit一次且仅一次
  int v;
  VisitFunc=Visit;
  for(v=0;v<G.vexnum;v++)
    visited[v]=FALSE; // 访问标志数组置初值(未被访问)
  for(v=0;v<G.vexnum;v++) // 由序号0开始, 继续查找未被访问过的顶点
    if(!visited[v])
      DFS(G,v);
  printf("\n");
}
typedef int QElemType; // 队列元素类型
#include"c3-2.h" // 链队列的存储结构
#include"bo3-2.cpp" // 链队列的基本操作
void BFSTraverse(OLGraph G,void(*Visit)(VertexType))
{ // 初始条件: 有向图G存在, Visit是顶点的应用函数。算法7.6
  // 操作结果: 从第1个顶点起, 按广度优先非递归遍历有向图G, 并对每个顶点调用函数Visit
  //           一次且仅一次。使用辅助队列Q和访问标志数组visited
  int v,u,w;
  LinkQueue Q;
  for(v=0;v<G.vexnum;v++)
    visited[v]=FALSE;
  InitQueue(Q);
  for(v=0;v<G.vexnum;v++)
    if(!visited[v])
    {
      visited[v]=TRUE;
      Visit(G.xlist[v].data);
      EnQueue(Q,v);
      while(!QueueEmpty(Q))
      {
```

```
        DeQueue(Q,u);
        for(w=FirstAdjVex(G,G.xlist[u].data);w>=0;w=NextAdjVex(G,G.xlist[u].data,
        G.xlist[w].data))
          if(!visited[w]) // w为u的尚未访问的邻接顶点的序号
          {
            visited[w]=TRUE;
            Visit(G.xlist[w].data);
            EnQueue(Q,w);
          }
      }
    }
  printf("\n");
}
void Display(OLGraph G)
{ // 输出有向图G
  int i;
  ArcBox *p;
  printf("共%d个顶点: ",G.vexnum);
  for(i=0;i<G.vexnum;i++) // 输出顶点
    printf("%s ",G.xlist[i].data);
  printf("\n%d条弧:\n",G.arcnum);
  for(i=0;i<G.vexnum;i++) // 顺出弧链表输出
  {
    p=G.xlist[i].firstout;
    while(p)
    {
      printf("%s→%s ",G.xlist[i].data,G.xlist[p->data.headvex].data);
      if(p->data.info) // 该弧有相关信息(权值)
        printf("权值: %d ",*p->data.info);
      p=p->tlink;
    }
    printf("\n");
  }
}

// main7-3.cpp 检验bo7-3.cpp的主程序
#include"c1.h"
typedef int InfoType; // 权值类型
#define MAX_VERTEX_NAME 3  // 顶点字符串最大长度+1
typedef char  VertexType[MAX_VERTEX_NAME];
#include"c7-31.h"
#include"bo7-3.cpp"
void visit(VertexType v)
{
  printf("%s ",v);
}
void main()
{
  int j,k,n;
  OLGraph g;
```

```
  VertexType v1,v2;
  CreateDG(g);
  Display(g);
  printf("修改顶点的值，请输入原值 新值: ");
  scanf("%s%s",v1,v2);
  PutVex(g,v1,v2);
  printf("插入新顶点，请输入顶点的值: ");
  scanf("%s",v1);
  InsertVex(g,v1);
  printf("插入与新顶点有关的弧，请输入弧数: ");
  scanf("%d",&n);
  for(k=0;k<n;k++)
  {
    printf("请输入另一顶点的值 另一顶点的方向(0:弧头 1:弧尾): ");
    scanf("%s%d",v2,&j);
    if(j)
      InsertArc(g,v2,v1);
    else
      InsertArc(g,v1,v2);
  }
  Display(g);
  printf("删除一条弧，请输入待删除弧的弧尾 弧头: ");
  scanf("%s%s",v1,v2);
  DeleteArc(g,v1,v2);
  Display(g);
  printf("删除顶点及相关的弧，请输入顶点的值: ");
  scanf("%s",v1);
  DeleteVex(g,v1);
  Display(g);
  printf("深度优先搜索的结果:\n");
  DFSTraverse(g,visit);
  printf("广度优先搜索的结果:\n");
  BFSTraverse(g,visit);
  DestroyGraph(g);
}
```

程序运行结果:

```
请输入有向图的顶点数,弧数,是否为带权图(是:1, 否:0): 2,1,1↙
请输入2个顶点的值(<3个字符):
a b↙
请输入1条弧的弧尾和弧头(空格为间隔):
a b↙
请输入该弧的权值: 3↙
共2个顶点: a b (见图7－38)
1条弧:
a→b 权值: 3
```

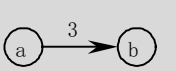


图 7－38　带权有向图

```
修改顶点的值，请输入原值 新值：a A↙
插入新顶点，请输入顶点的值：c↙
插入与新顶点有关的弧，请输入弧数：2↙
请输入另一顶点的值 另一顶点的方向(0:弧头 1:弧尾)：A 1↙
要插入的弧是否带权(是：1,否：0)：1↙
请输入该弧的权值：4↙
请输入另一顶点的值 另一顶点的方向(0:弧头 1:弧尾)：b 0↙
要插入的弧是否带权(是：1,否：0)：1↙
请输入该弧的权值：5↙
共3个顶点：A b c (见图7－39)
3条弧:
A→c 权值：4 A→b 权值：3
```

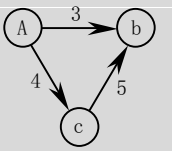


图 7－39　插入顶点 c 后图示

```
c→b 权值：5
删除一条弧，请输入待删除弧的弧尾 弧头：A b↙
共3个顶点：A b c (见图7－40)
2条弧:
A→c 权值：4

c→b 权值：5
删除顶点及相关的弧，请输入顶点的值：A↙
共2个顶点：b c (见图7－41)
1条弧:

c→b 权值：5
深度优先搜索的结果:
b c
广度优先搜索的结果:
b c
```

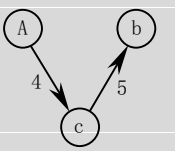


图 7－40　删除弧 A→b 后图示

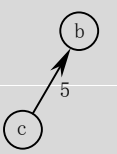


图 7－41　删除顶点 A 后图示

7.2.4　邻接多重表

```
// c7-4.h 无向图的邻接多重表存储结构(见图7－42)
#define MAX_VERTEX_NUM 20
enum VisitIf{unvisited,visited};
struct EBox
{
  VisitIf mark; // 访问标记
  int ivex,jvex; // 该边依附的两个顶点的位置
  EBox *ilink,*jlink; // 分别指向依附这两个顶点的下一条边
  InfoType *info; // 该边信息指针，可指向权值或其它信息
};
struct VexBox
{
  VertexType data;
  EBox *firstedge; // 指向第一条依附该顶点的边
};
struct AMLGraph
{
  VexBox adjmulist[MAX_VERTEX_NUM];
  int vexnum,edgenum; // 无向图的当前顶点数和边数
```

};

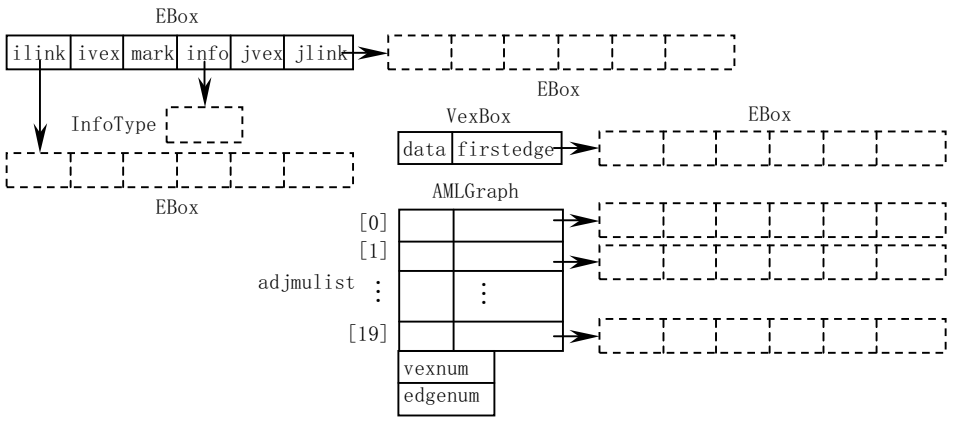
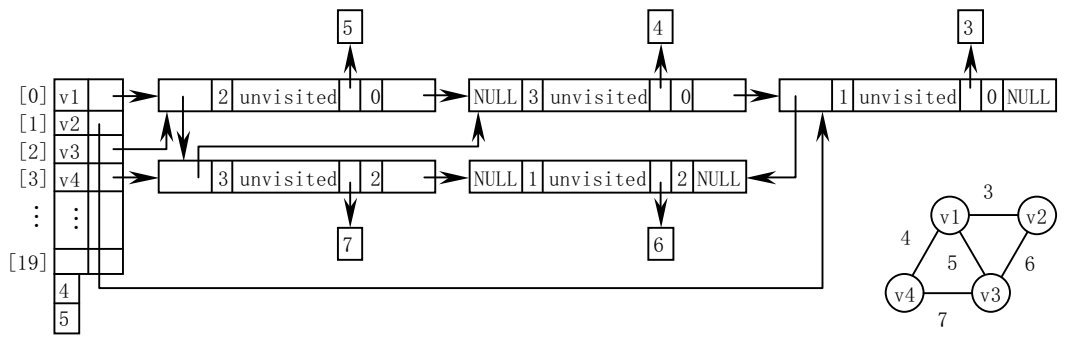


图 7－42　无向图的邻接多重表存储结构

图 7－43 是根据 c7-4. h 定义的无向图的存储结构。与 bo7-2. cpp、bo7-3. cpp 一样，bo7-4. cpp 中基本操作函数 CreateGraph()也是在表头插入边结点的。所以，对于给定的图，它的边结点的链表结构也不惟一，与边的输入顺序有关。采用邻接多重表存储结构，每条边只生成一个结点。而用邻接表存储结构表示无向图，图的每条边生成两个结点。



(a) 存储结构　　(b)无向网

图 7－43　无向图及其邻接多重表存储结构

在无向图中边的两个顶点是没有顺序的，顶点是根据与其顶点号对应的指针域形成邻接顶点链表的。如图 7－43(a)所示，上排权值分别是 5、4、3 的 3 个结点形成与 v1 相连的 3 条边。它们的 jvex=0，从头指针 adjmulist[0]. firstedge 指向上排左边结点(这个结点表示连接顶点 v1 和 v3 的边，因为它的 jvex 和 ivex 分别为 0 和 2)开始，通过 jlink 指针链接在一起。与某一顶点相连的边不一定都由 ilink 或 jlink 指向，它取决于结点的 ivex 和 jvex 中的哪一个与该顶点的序号相同。以与顶点 v3 相连的边为例，从头指针 adjmulist[2]. firstedge 指向上排左边结点，因为该结点的 ivex=2，所以，该结点的 ilink 指向与顶点 v3 相连的下一条边(下排左边结点)。而这个结点的 jvex=2，则该结点的 jlink 指向与顶点 v3 相连的下一条边(下排右边结点)。这个结点的 jvex=2，jlink=NULL，表明链表结束，不再有与顶点 v3 相连的边。这样，与顶点 v3 邻接的顶点有 3 个，依次是 v1、v4 和 v2。与图示相符。

虽然邻接多重表存储结构也是不带头结点的单链表结构，但由于指向下一结点的指针

域是变化的，可能是 ilink 指向下一个结点，也可能是 jlink 指向下一个结点，这取决于 ivex 和 jvex 的值。所以不带头结点的单链表基本操作(在 bo2-8.cpp 中)应用于邻接多重表存储结构的基本操作中很不方便，这使得邻接多重表存储结构的基本操作(在 bo7-4.cpp 中)比较冗长。

```
// bo7-4.cpp 无向图的邻接多重表(存储结构由c7-4.h定义)的基本函数类型(16个)，包括算法7.4，7.6
int LocateVex(AMLGraph G,VertexType u)
{ // 初始条件：无向图G存在，u和G中顶点有相同特征
  // 操作结果：若G中存在顶点u，则返回该顶点在无向图中位置；否则返回-1
  int i;
  for(i=0;i<G.vexnum;++i)
    if(strcmp(u,G.adjmulist[i].data)==0)
      return i;
  return -1;
}
void CreateGraph(AMLGraph &G)
{ // 采用邻接多重表存储结构，构造无向图G
  int i,j,k,IncInfo;
  VertexType va,vb;
  EBox *p;
  printf("请输入无向图的顶点数,边数,是否为带权图(是:1，否:0)：");
  scanf("%d,%d,%d",&G.vexnum,&G.edgenum,&IncInfo);
  printf("请输入%d个顶点的值(<%d个字符):\n",G.vexnum,MAX_NAME);
  for(i=0;i<G.vexnum;++i) // 构造顶点向量
  {
    scanf("%s",G.adjmulist[i].data);
    G.adjmulist[i].firstedge=NULL;
  }
  printf("请顺序输入每条边的两个端点(以空格作为间隔):\n");
  for(k=0;k<G.edgenum;++k) // 构造表结点链表
  {
    scanf("%s%s%*c",va,vb); // %*c吃掉回车符
    i=LocateVex(G,va); // 一端
    j=LocateVex(G,vb); // 另一端
    p=(EBox*)malloc(sizeof(EBox));
    p->mark=unvisited; // 设初值
    p->ivex=i;
    p->ilink=G.adjmulist[i].firstedge; // 插在一端的表头
    G.adjmulist[i].firstedge=p;
    p->jvex=j;
    p->jlink=G.adjmulist[j].firstedge; // 插在另一端的表头
    G.adjmulist[j].firstedge=p;
    if(IncInfo) // 网
    {
      p->info=(InfoType*)malloc(sizeof(InfoType));
      printf("请输入该边的权值：");
      scanf("%d",p->info);
    }
    else
      p->info=NULL;
```

```
  }
}
VertexType& GetVex(AMLGraph G,int v)
{ // 初始条件：无向图G存在，v是G中某个顶点的序号。操作结果：返回v的值
  if(v>=G.vexnum||v<0)
    exit(ERROR);
  return G.adjmulist[v].data;
}
Status PutVex(AMLGraph &G,VertexType v,VertexType value)
{ // 初始条件：无向图G存在，v是G中某个顶点。操作结果：对v赋新值value
  int i;
  i=LocateVex(G,v);
  if(i<0) // v不是G的顶点
    return ERROR;
  strcpy(G.adjmulist[i].data,value);
  return OK;
}
int FirstAdjVex(AMLGraph G,VertexType v)
{ // 初始条件：无向图G存在，v是G中某个顶点
  // 操作结果：返回v的第一个邻接顶点的序号。若顶点在G中没有邻接顶点，则返回-1
  int i;
  i=LocateVex(G,v);
  if(i<0) // G中不存在顶点v
    return -1;
  if(G.adjmulist[i].firstedge) // v有邻接顶点
    if(G.adjmulist[i].firstedge->ivex==i)
      return G.adjmulist[i].firstedge->jvex;
    else
      return G.adjmulist[i].firstedge->ivex;
  else
    return -1;
}
int NextAdjVex(AMLGraph G,VertexType v,VertexType w)
{ // 初始条件：无向图G存在，v是G中某个顶点，w是v的邻接顶点
  // 操作结果：返回v的(相对于w的)下一个邻接顶点的序号。若w是v的最后一个邻接点，则返回-1
  int i,j;
  EBox *p;
  i=LocateVex(G,v); // i是顶点v的序号
  j=LocateVex(G,w); // j是顶点w的序号
  if(i<0||j<0) // v或w不是G的顶点
    return -1;
  p=G.adjmulist[i].firstedge; // p指向顶点v的第1条边
  while(p)
    if(p->ivex==i&&p->jvex!=j) // 不是邻接顶点w(情况1)
      p=p->ilink; // 找下一个邻接顶点
    else if(p->jvex==i&&p->ivex!=j) // 不是邻接顶点w(情况2)
      p=p->jlink; // 找下一个邻接顶点
    else // 是邻接顶点w
      break;
  if(p&&p->ivex==i&&p->jvex==j) // 找到邻接顶点w(情况1)
  {
```

```
    p=p->ilink;
    if(p&&p->ivex==i)
      return p->jvex;
    else if(p&&p->jvex==i)
      return p->ivex;
  }
  if(p&&p->ivex==j&&p->jvex==i) // 找到邻接顶点w(情况2)
  {
    p=p->jlink;
    if(p&&p->ivex==i)
      return p->jvex;
    else if(p&&p->jvex==i)
      return p->ivex;
  }
  return -1;
}
Status InsertVex(AMLGraph &G,VertexType v)
{ // 初始条件：无向图G存在，v和G中顶点有相同特征
  // 操作结果：在G中增添新顶点v(不增添与顶点相关的弧，留待InsertArc()去做)
  if(G.vexnum==MAX_VERTEX_NUM) // 结点已满，不能插入
    return ERROR;
  if(LocateVex(G,v)>=0) // 结点已存在，不能插入
    return ERROR;
  strcpy(G.adjmulist[G.vexnum].data,v);
  G.adjmulist[G.vexnum++].firstedge=NULL;
  return OK;
}
Status DeleteArc(AMLGraph &G,VertexType v,VertexType w)
{ // 初始条件：无向图G存在，v和w是G中两个顶点。操作结果：在G中删除弧<v,w>
  int i,j;
  EBox *p,*q;
  i=LocateVex(G,v);
  j=LocateVex(G,w);
  if(i<0||j<0||i==j)
    return ERROR;  // 图中没有该点或弧。以下使指向待删除边的第1个指针绕过这条边
  p=G.adjmulist[i].firstedge; // p指向顶点v的第1条边
  if(p&&p->jvex==j) // 第1条边即为待删除边(情况1)
    G.adjmulist[i].firstedge=p->ilink;
  else if(p&&p->ivex==j) // 第1条边即为待删除边(情况2)
    G.adjmulist[i].firstedge=p->jlink;
  else // 第1条边不是待删除边
  {
    while(p) // 向后查找弧<v,w>
      if(p->ivex==i&&p->jvex!=j) // 不是待删除边
      {
        q=p;
        p=p->ilink; // 找下一个邻接顶点
      }
      else if(p->jvex==i&&p->ivex!=j) // 不是待删除边
      {
        q=p;
```

```
      p=p->jlink; // 找下一个邻接顶点
    }
    else // 是邻接顶点w
      break;
  if(!p) // 没找到该边
    return ERROR;
  if(p->ivex==i&&p->jvex==j) // 找到弧<v,w>(情况1)
    if(q->ivex==i)
      q->ilink=p->ilink;
    else
      q->jlink=p->ilink;
  else if(p->ivex==j&&p->jvex==i) // 找到弧<v,w>(情况2)
    if(q->ivex==i)
      q->ilink=p->jlink;
    else
      q->jlink=p->jlink;
} // 以下由另一顶点起找待删除边且删除之
p=G.adjmulist[j].firstedge; // p指向顶点w的第1条边
if(p->jvex==i) // 第1条边即为待删除边(情况1)
  G.adjmulist[j].firstedge=p->ilink;
else if(p->ivex==i) // 第1条边即为待删除边(情况2)
  G.adjmulist[j].firstedge=p->jlink;
else // 第1条边不是待删除边
{
  while(p) // 向后查找弧<v,w>
    if(p->ivex==j&&p->jvex!=i) // 不是待删除边
    {
      q=p;
      p=p->ilink; // 找下一个邻接顶点
    }
    else if(p->jvex==j&&p->ivex!=i) // 不是待删除边
    {
      q=p;
      p=p->jlink; // 找下一个邻接顶点
    }
    else // 是邻接顶点v
      break;
  if(p->ivex==i&&p->jvex==j) // 找到弧<v,w>(情况1)
    if(q->ivex==j)
      q->ilink=p->jlink;
    else
      q->jlink=p->jlink;
  else if(p->ivex==j&&p->jvex==i) // 找到弧<v,w>(情况2)
    if(q->ivex==j)
      q->ilink=p->ilink;
    else
      q->jlink=p->ilink;
}
if(p->info) // 有相关信息(或权值)
  free(p->info); // 释放相关信息(或权值)
free(p); // 释放结点
```

```
  G.edgenum--; // 边数-1
  return OK;
}
Status DeleteVex(AMLGraph &G,VertexType v)
{ // 初始条件：无向图G存在，v是G中某个顶点。操作结果：删除G中顶点v及其相关的边
  int i,j;
  EBox *p;
  i=LocateVex(G,v); // i为待删除顶点的序号
  if(i<0)
    return ERROR;
  for(j=0;j<G.vexnum;j++) // 删除与顶点v相连的边(如果有的话)
    DeleteArc(G,v,G.adjmulist[j].data); // 如果存在此弧，则删除
  for(j=i+1;j<G.vexnum;j++) // 排在顶点v后面的顶点的序号减1
    G.adjmulist[j-1]=G.adjmulist[j];
  G.vexnum--; // 顶点数减1
  for(j=i;j<G.vexnum;j++) // 修改序号大于i的顶点在表结点中的序号
  {
    p=G.adjmulist[j].firstedge;
    if(p)
      if(p->ivex==j+1)
      {
        p->ivex--;
        p=p->ilink;
      }
      else
      {
        p->jvex--;
        p=p->jlink;
      }
  }
  return OK;
}
void DestroyGraph(AMLGraph &G)
{ // 初始条件：有向图G存在。操作结果：销毁有向图G
  int i;
  for(i=G.vexnum-1;i>=0;i--) // 由大到小依次删除顶点
    DeleteVex(G,G.adjmulist[i].data);
}
Status InsertArc(AMLGraph &G,VertexType v,VertexType w)
{ // 初始条件：无向图G存在，v和W是G中两个顶点。操作结果：在G中增添弧<v,w>
  int i,j,IncInfo;
  EBox *p;
  i=LocateVex(G,v); // 一端
  j=LocateVex(G,w); // 另一端
  if(i<0||j<0||i==j)
    return ERROR;
  p=(EBox*)malloc(sizeof(EBox));
  p->mark=unvisited;
  p->ivex=i;
  p->ilink=G.adjmulist[i].firstedge; // 插在表头
  G.adjmulist[i].firstedge=p;
```

```
  p->jvex=j;
  p->jlink=G.adjmulist[j].firstedge; // 插在表头
  G.adjmulist[j].firstedge=p;
  printf("该边是否有权值(1:有 0:无): ");
  scanf("%d",&IncInfo);
  if(IncInfo) // 有权值
  {
    p->info=(InfoType*)malloc(sizeof(InfoType));
    printf("请输入该边的权值: ");
    scanf("%d",p->info);
  }
  else
    p->info=NULL;
  G.edgenum++;
  return OK;
}
Boolean visite[MAX_VERTEX_NUM]; // 访问标志数组(全局量)
void(*VisitFunc)(VertexType v);
void DFS(AMLGraph G,int v)
{
  int j;
  EBox *p;
  VisitFunc(G.adjmulist[v].data);
  visite[v]=TRUE;
  p=G.adjmulist[v].firstedge;
  while(p)
  {
    j=p->ivex==v?p->jvex:p->ivex;
    if(!visite[j])
      DFS(G,j);
    p=p->ivex==v?p->ilink:p->jlink;
  }
}
void DFSTraverse(AMLGraph G,void(*visit)(VertexType))
{ // 初始条件: 图G存在, Visit是顶点的应用函数。算法7.4
  // 操作结果: 从第1个顶点起, 深度优先遍历图G, 并对每个顶点调用函数Visit一次且仅一次
  int v;
  VisitFunc=visit;
  for(v=0;v<G.vexnum;v++)
    visite[v]=FALSE;
  for(v=0;v<G.vexnum;v++)
    if(!visite[v])
      DFS(G,v);
  printf("\n");
}
typedef int QElemType; // 队列元素类型
#include"c3-2.h" // 链队列的存储结构, BFSTraverse()用
#include"bo3-2.cpp" // 链队列的基本操作, BFSTraverse()用
void BFSTraverse(AMLGraph G,void(*Visit)(VertexType))
{ // 初始条件: 图G存在, Visit是顶点的应用函数。算法7.6
  // 操作结果: 从第1个顶点起, 按广度优先非递归遍历图G, 并对每个顶点调用函数
```

```
  //          Visit一次且仅一次。使用辅助队列Q和访问标志数组visite
  int v,u,w;
  LinkQueue Q;
  for(v=0;v<G.vexnum;v++)
    visite[v]=FALSE; // 置初值
  InitQueue(Q); // 置空的辅助队列Q
  for(v=0;v<G.vexnum;v++)
    if(!visite[v]) // v尚未访问
    {
      visite[v]=TRUE; // 设置访问标志为TRUE(已访问)
      Visit(G.adjmulist[v].data);
      EnQueue(Q,v); // v入队列
      while(!QueueEmpty(Q)) // 队列不空
      {
        DeQueue(Q,u); // 队头元素出队并置为u
        for(w=FirstAdjVex(G,G.adjmulist[u].data);w>=0;w=NextAdjVex(G,G.adjmulist[u].data,
        G.adjmulist[w].data))
          if(!visite[w]) // w为u的尚未访问的邻接顶点的序号
          {
            visite[w]=TRUE;
            Visit(G.adjmulist[w].data);
            EnQueue(Q,w);
          }
      }
    }
  printf("\n");
}
void MarkUnvizited(AMLGraph G)
{ // 置边的访问标记为未被访问
  int i;
  EBox *p;
  for(i=0;i<G.vexnum;i++)
  {
    p=G.adjmulist[i].firstedge;
    while(p)
    {
      p->mark=unvisited;
      if(p->ivex==i)
        p=p->ilink;
      else
        p=p->jlink;
    }
  }
}
void Display(AMLGraph G)
{ // 输出无向图的邻接多重表G
  int i;
  EBox *p;
  MarkUnvizited(G); // 置边的访问标记为未被访问
  printf("%d个顶点: \n",G.vexnum);
  for(i=0;i<G.vexnum;++i)
```

```
    printf("%s ",G.adjmulist[i].data);
  printf("\n%d条边:\n",G.edgenum);
  for(i=0;i<G.vexnum;i++)
  {
    p=G.adjmulist[i].firstedge;
    while(p)
      if(p->ivex==i) // 边的i端与该顶点有关
      {
        if(!p->mark) // 只输出一次
        {
          printf("%s－%s ",G.adjmulist[i].data,G.adjmulist[p->jvex].data);
          p->mark=visited;
          if(p->info) // 输出附带信息
            printf("权值: %d ",*p->info);
        }
        p=p->ilink;
      }
      else // 边的j端与该顶点有关
      {
        if(!p->mark) // 只输出一次
        {
          printf("%s－%s ",G.adjmulist[p->ivex].data,G.adjmulist[i].data);
          p->mark=visited;
          if(p->info) // 输出附带信息
            printf("权值: %d ",*p->info);
        }
        p=p->jlink;
      }
    printf("\n");
  }
}

// main7-4.cpp 检验bo7-4.cpp的主程序
#include"c1.h"
#define MAX_NAME 3 // 顶点字符串的最大长度+1
typedef int InfoType; // 权值类型
typedef char VertexType[MAX_NAME]; // 字符串类型
#include"c7-4.h"
#include"bo7-4.cpp"
void visit(VertexType v)
{
  printf("%s ",v);
}
void main()
{
  int k,n;
  AMLGraph g;
  VertexType v1,v2;
  CreateGraph(g);
  Display(g);
  printf("修改顶点的值, 请输入原值 新值: ");
```

```
  scanf("%s%s",v1,v2);
  PutVex(g,v1,v2);
  printf("插入新顶点，请输入顶点的值: ");
  scanf("%s",v1);
  InsertVex(g,v1);
  printf("插入与新顶点有关的边，请输入边数: ");
  scanf("%d",&n);
  for(k=0;k<n;k++)
  {
    printf("请输入另一顶点的值: ");
    scanf("%s",v2);
    InsertArc(g,v1,v2);
  }
  Display(g);
  printf("删除一条边，请输入待删除边的两顶点(以空格作为间隔): ");
  scanf("%s%s",v1,v2);
  DeleteArc(g,v1,v2);
  Display(g);
  printf("删除顶点及相关的边，请输入顶点的值: ");
  scanf("%s",v1);
  DeleteVex(g,v1);
  Display(g);
  printf("深度优先搜索的结果:\n");
  DFSTraverse(g,visit);
  printf("广度优先搜索的结果:\n");
  BFSTraverse(g,visit);
  DestroyGraph(g);
}
```

程序运行结果：

```
请输入无向图的顶点数,边数,是否为带权图(是:1，否:0): 2,1,1↙
请输入2个顶点的值(<3个字符):
a b↙
请顺序输入每条边的两个端点(以空格作为间隔):
a b↙
请输入该边的权值: 3↙
2个顶点: (见图7－44)
a b
1条边:
a－b 权值: 3

修改顶点的值，请输入原值 新值: a A↙
插入新顶点，请输入顶点的值: c↙
插入与新顶点有关的边，请输入边数: 2↙
请输入另一顶点的值: A↙
该边是否有权值(1:有 0:无): 1↙
请输入该边的权值: 4↙
请输入另一顶点的值: b↙
```

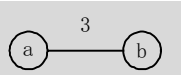


图 7－44　无向网

```
该边是否有权值(1:有 0:无): 1↙
请输入该边的权值: 5↙
3个顶点:(见图7-45)
A b c
3条边:
c-A 权值: 4 A-b 权值: 3
c-b 权值: 5
```

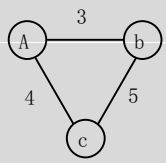


图 7-45　插入顶点 c 后图示

```
删除一条边,请输入待删除边的两顶点(以空格为间隔): A b↙
3个顶点:(见图7-46)
A b c
2条边:
c-A 权值: 4
c-b 权值: 5
```

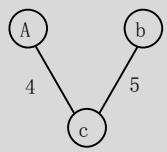


图 7-46　删除 A-b 边后图示

```
删除顶点及相关的边,请输入顶点的值: c↙
2个顶点:(见图7-47)
A b
0条边:
```

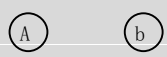


图 7-47　删除顶点 c 后图示

```
深度优先搜索的结果:
A b
广度优先搜索的结果:
A b
```

7.3　图的遍历

对图的搜索就是对图中顶点的遍历。图中各顶点的关系比较复杂、一个顶点可能有多个邻接顶点，也可能是独立顶点(非连通图)。为了不重复地访问所有顶点，需设立一个访问标志数组 visited[]，并置其初值为 FALSE(未被访问)。遍历时只访问那些未被访问过的顶点，且在访问后，将其访问标志的值改为 TRUE。当所有顶点访问标志的值都为 TRUE，则图已遍历。遍历一般从图的第 1 个顶点开始。确定遍历顶点有两个搜索原则：深度优先搜索和广度优先搜索。

7.3.1　深度优先搜索

算法 7.4、7.5 是利用递归对图进行深度优先搜索的算法，它的主要思想是：先访问图的第 1 个顶点，然后访问这个顶点的第 1 个邻接顶点，再访问第 1 个邻接顶点的第 1 个邻接顶点。如果这个顶点被访问过了，就访问第 2 个邻接顶点，…… 所谓第 1 个邻接顶点、第 2 个邻接顶点不是由图的拓扑关系决定的，它取决于图的存储结构。即使是同一个图，如果它的存储结构不同，那么它的某个顶点的第 1 个邻接顶点、第 2 个邻接顶点也可能不同。关于这一点，将在后面 algo7-10.cpp、algo7-11.cpp 中根据实例做进一步的说明。算法 7.4、7.5 是基于基本操作的，与图的具体存储结构无关，所以很容易移植到各种存储结构中，只要那种存储结构的有关基本操作函数存在即可。在 bo7-1.cpp～bo7-4.cpp 中都有实现算法

7.4 和算法 7.5 的函数。

7.3.2　广度优先搜索

算法 7.6 是对图进行广度优先搜索的算法，它的主要思想是：先访问图的第 1 个顶点，然后依次访问这个顶点的所有邻接顶点，再依次访问这些邻接顶点的所有邻接顶点。这需要建立 1 个先进先出的队列，依次将访问过的顶点入队。当前 1 个顶点的所有邻接顶点都被访问了，就出队 1 个顶点，再访问这个顶点的所有邻接顶点且将它们入队。直至所有顶点都被访问过。算法 7.6 也是基于基本操作的，在 bo7-1.cpp～bo7-4.cpp 中也都有实现算法 7.6 的函数。algo7-10.cpp 是在邻接矩阵的存储结构下，调用算法 7.4、7.5 和 7.6，对图进行深度优先搜索和广度优先搜索的程序。

```
// algo7-10.cpp 检验深度优先和广度优先的程序(邻接矩阵存储结构)
#include"c1.h"
#define MAX_NAME 5 // 顶点字符串的最大长度+1
#define MAX_INFO 20 // 相关信息字符串的最大长度+1
typedef int VRType; // 顶点关系类型
typedef char InfoType; // 相关信息类型
typedef char VertexType[MAX_NAME]; // 顶点类型
#include"c7-1.h" // 邻接矩阵存储结构
#include"bo7-1.cpp" // 邻接矩阵存储结构的基本操作
void visit(VertexType i)
{
  printf("%s ",i);
}
void main()
{
  MGraph g;
  VertexType v1,v2;
  CreateFUDG(g); // 利用数据文件创建无向图，在bo7-1.cpp中
  Display(g); // 输出无向图,在bo7-1.cpp中
  printf("深度优先搜索的结果:\n");
  DFSTraverse(g,visit); // 在bo7-1.cpp中
  printf("修改顶点的值，请输入原值 新值: ");
  scanf("%s%s",v1,v2);
  PutVex(g,v1,v2); // 在bo7-1.cpp中
  printf("删除一条边或弧，请输入待删除边或弧的弧尾 弧头: ");
  scanf("%s%s",v1,v2);
  DeleteArc(g,v1,v2); // 在bo7-1.cpp中
  printf("广度优先搜索的结果:\n");
  BFSTraverse(g,visit); // 在bo7-1.cpp中
}
```

数据文件 f7-1.txt 的内容(图 7－48 是其所表示的无向图)：

```
8
14
a
```

```
b
c
d
e
f
g
h
a b
a c
a e
a f
a g
a h
b d
b e
b h
c g
c h
d h
e f
f g
```

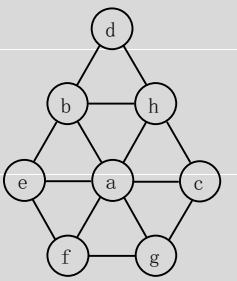


图 7－48　数据文件 f7-1.txt 所表示的无向图

程序运行结果：

```
请输入数据文件名(f7-1.txt或f7-2.txt)：f7-1.txt↙
8个顶点14条边或弧的无向图。顶点依次是：a b c d e f g h
G.arcs.adj:
        0       1       1       0       1       1       1       1
        1       0       0       1       1       0       0       1
        1       0       0       0       0       0       1       1
        0       1       0       0       0       0       0       1
        1       1       0       0       0       1       0       0
        1       0       0       0       1       0       1       0
        1       0       1       0       0       1       0       0
        1       1       1       1       0       0       0       0
G.arcs.info:
顶点1(弧尾) 顶点2(弧头) 该边或弧的信息：
深度优先搜索的结果：
a b d h c g f e
修改顶点的值，请输入原值 新值：e E↙
删除一条边或弧，请输入待删除边或弧的弧尾 弧头：a b↙
广度优先搜索的结果：
a c E f g h b d
```

图有 2 个基本操作：FirstAdjVex(G, v)和 NextAdjVex(G, v, w)。FirstAdjVex(G, v)返回图 G 中顶点 v 的第 1 个邻接顶点(在图中的位置)。在邻接矩阵存储结构中，返回邻接矩阵 G. arcs. adj 中 v 所对应的行的第 1 个值为 1(图)或权值(网)的顶点(在图中的位置)。以 f7-

1.txt 所表示的图 G 为例，FirstAdjVex(G,a)(在 bo7-1.cpp 中)返回 b 在图中的位置 1；FirstAdjVex(G,e)返回 a 在图中的位置 0。NextAdjVex(G,v,w)比 FirstAdjVex(G,v)多了 1 个形参 w，它返回图 G 中顶点 v 的所有邻接顶点中排在邻接顶点 w 后面的那个邻接顶点(在图中的位置)。在邻接矩阵存储结构中，返回邻接矩阵 G.arcs.adj 中 v 所对应的行的 w 那列后面的第 1 个值为 1(图)或权值(网)的顶点(在图中的位置)。以 f7-1.txt 所表示的图 G 为例，NextAdjVex(G,a,c)(在 bo7-1.cpp 中)返回第 1 行(a 行)排在第 3 列(c 列)后面的第 1 个值为 1 的顶点 e 在图中的位置 4。由这样的定义，我们可以推知，algo7-10.cpp 调用算法 7.4 和 7.5 对图 G 深度优先搜索的过程：首先访问 G 的第 1 个顶点 a；接下来访问 a 的第 1 个邻接顶点 b；再准备访问 b 的第 1 个邻接顶点 a，但 a 已被访问过，则不再访问 a，转而访问 b 排在 a 后的邻接顶点 d；再准备访问 d 的第 1 个邻接顶点 b，由于同样的原因，转而访问 b 排在 d 后的邻接顶点 h；再访问 h 的第 1 个未被访问的邻接顶点 c、c 的第 1 个未被访问的邻接顶点 g、g 的第 1 个未被访问的邻接顶点 f、f 的第 1 个未被访问的邻接顶点 e。遍历结束，其顺序与程序运行结果相同。

algo7-10.cpp 调用算法 7.6 对图 G 广度优先搜索的过程(这时 e 已被改为 E，a—b 边已被删除，b 不再是 a 的邻接顶点)：首先访问 G 的第 1 个顶点 a，将 a 入队，在队不空的情况下，出队元素 a，依次访问 a 的所有邻接顶点 c、E、f、g、h，并将它们入队；依次出队 c、E，访问 E 的邻接顶点 b，将 b 入队；依次出队 f、g、h，访问 h 的邻接顶点 d。遍历结束，其顺序与程序运行结果相同。

algo7-11.cpp 是在邻接表的存储结构下，对图 G 深度优先搜索和广度优先搜索的程序。其中，不仅调用基于基本操作的算法 7.4、7.5 和 7.6，对图 G 深度优先搜索和广度优先搜索，而且，调用了基于邻接表的存储结构的对图 G 深度优先搜索和广度优先搜索的函数 DFSTraverse1()和 BFSTraverse1()。

```
// algo7-11.cpp 检验深度优先和广度优先的程序(邻接表存储结构)
#include"c1.h"
#define MAX_NAME 5 // 顶点字符串的最大长度+1
typedef int InfoType; // 网的权值类型
typedef char VertexType[MAX_NAME]; // 顶点类型为字符串
#include"c7-21.h" // 邻接表存储结构
#include"bo7-2.cpp" // 邻接表存储结构的基本操作
void visit(char *i)
{
  printf("%s ",i);
}
void main()
{
  ALGraph g;
  CreateGraphF(g); // 利用数据文件创建无向图，在bo7-2.cpp中
  Display(g); // 输出无向图，在bo7-2.cpp中
  printf("深度优先搜索的结果:\n");
  DFSTraverse(g,visit); // 调用算法7.4，在bo7-2.cpp中
  DFSTraverse1(g,visit); // 另一种方法，在bo7-2.cpp中
  printf("广度优先搜索的结果:\n");
  BFSTraverse(g,visit); // 调用算法7.6，在bo7-2.cpp中
```

```
  BFSTraverse1(g,visit); // 另一种方法，在bo7-2.cpp中
  DestroyGraph(g); // 销毁图g
}
```

程序运行结果：

```
请输入数据文件名(f7-1.txt或f7-2.txt): f7-1.txt↙
请输入图的类型(有向图:0,有向网:1,无向图:2,无向网:3): 2↙
无向图
8个顶点:
a b c d e f g h
14条弧(边):
a→h a→g a→f a→e a→c a→b
b→h b→e b→d
c→h c→g
d→h
e→f
f→g


深度优先搜索的结果:
a h d b e f g c
a h d b e f g c
广度优先搜索的结果:
a h g f e c b d
a h g f e c b d
```

algo7-11.cpp 和 algo7-10.cpp 一样，都是利用数据文件 f7-1.txt 构造图的。它们构造的图的拓扑结构完全相同，如图 7－48 所示，但它们的深度优先搜索却并不同。这是因为虽然在它们的存储结构中顶点序号是相同的，但由于邻接表结构在构造边时，总是将邻接顶点插在表头，这样，边的输入顺序不同，图的存储结构也就不同。相对于某一顶点，它的“第 1 个邻接顶点”、“下 1 个邻接顶点”也不同，这导致了搜索顺序不同。

algo7-11.cpp 利用数据文件 f7-1.txt 构造的无向图 G 的存储结构如图 7－49 所示(略去网的权值指针域，且为直观，用顶点名称代替顶点位置)。根据这样的存储结构，FirstAdjVex(G, a)(在 bo7-2.cpp 中)返回 h 在图中的位置 7；NextAdjVex(G, a, c)(在 bo7-2.cpp 中)返回 b 在图中的位置 1。这样，我们可以推知，algo7-11.cpp 调用算法 7.4 和 7.5 对图 G 深度优先搜索的过程：首先访问 G 的第 1 个顶点 a；接下来访问 a 的第 1 个邻接顶点 h；再访问 h 的第 1 个邻接顶点 d；再准备访问 d 的第 1 个邻接顶点 h，由于 h 已被访问，转而访问 d 排在 h 后的邻接顶点 b；再访问 b 的第 1 个未被访问的邻接顶点 e、e 的第 1 个未被访问的邻接顶点 f、f 的第 1 个未被访问的邻接顶点 g、g 的第 1 个未被访问的邻接顶点 c。遍历结束，其顺序与程序运行结果相同。algo7-11.cpp 调用算法 7.6 对图 G 广度优先搜索的过程：首先访问 G 的第 1 个顶点 a；接下来依次访问 a 的所有邻接顶点 h、g、f、e、c 和 b；再访问 h 的邻接顶点 d。遍历结束，其顺序与程序运行结果相同。DFSTraverse1()和 BFSTraverse1()没有调用图的基本操作函数 FirstAdjVex()和

NextAdjVex()，它们直接用表结点的指针 p，用 p=G. vertices[v]. firstarc 和 p=p->next 代替 FirstAdjVex()和 NextAdjVex()的作用。这样做效率高、直观，但仅适用于邻接表存储结构。它们得到的结果是一样的。

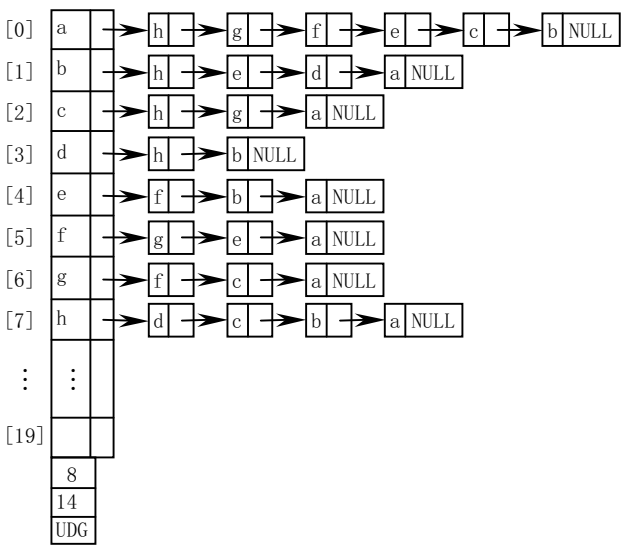


图 7 - 49　根据数据文件 f7-1. txt 所产生的邻接表

数据文件 f7-2. txt 所表示的无向图与 f7-1. txt 的一样，如图 7 - 48 所示。只是边的输入顺序不同。数据文件 f7-2. txt 的内容如下：

```
8
14
a
b
c
d
e
f
g
h
f g
e f
d h
c h
c g
b h
b e
b d
a h
a g
a f
a e
a c
a b
```

利用数据文件 f7-2.txt 运行 algo7-11.cpp，产生的邻接表如图 7-50 所示。

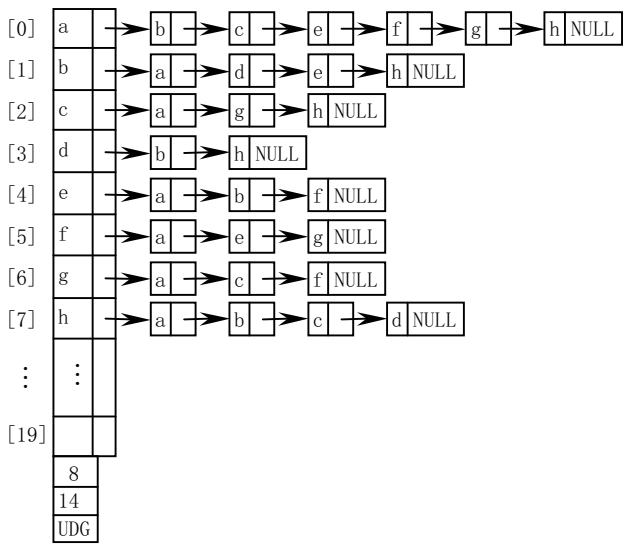


图 7-50　根据数据文件 f7-2.txt 所产生的邻接表

程序运行结果如下：

```
请输入数据文件名(f7-1.txt或f7-2.txt)： f7-2.txt↙
请输入图的类型(有向图:0,有向网:1,无向图:2,无向网:3)： 2↙
无向图
8个顶点:
a b c d e f g h
14条弧(边):
a→b a→c a→e a→f a→g a→h
b→d b→e b→h
c→g c→h
d→h
e→f
f→g

深度优先搜索的结果:
a b d h c g f e
a b d h c g f e
广度优先搜索的结果:
a b c e f g h d
a b c e f g h d
```

其中，深度优先搜索的顺序与 algo7-10.cpp 的一样。algo7-10.cpp 利用数据文件 f7-2.txt 的运行结果与利用 f7-1.txt 的一样，读者可自行验证。

除了 c7-1.h 存储结构，c7-2.h～c7-4.h 中边或弧都是以链表形式存储的，且边或弧总是插在表头。当边或弧的输入顺序不同时，其存储结构就不同，故搜索的顺序不同。

7.4　图的连通性问题

7.4.1　无向图的连通分量和生成树

具有 n 个顶点的无向连通图至少有 n-1 条边，如果只有 n-1 条边，则不会形成环，这样的图称为“ 生成树”。连通图可通过遍历构造生成树，非连通图的每个连通分量可构造一棵生成树，整个非连通图构造为生成森林。algo7-1.cpp 调用算法 7.7、7.8，将无向图构造为生成森林，并以孩子—兄弟二叉链表存储之。

```
// algo7-1.cpp 调用算法7.7、7.8
#include"c1.h"
#define MAX_NAME 2 // 顶点字符串的最大长度+1
typedef char VertexType[MAX_NAME];
typedef VertexType TElemType; // 定义树的元素类型为图的顶点类型
#include"c6-5.h" // 孩子—兄弟二叉链表存储结构
#include"func6-2.cpp" // 孩子—兄弟二叉链表存储结构的先根遍历操作
typedef int InfoType; // 权值类型
#include"c7-21.h" // bo7-2.cpp采用的存储类型
#include"bo7-2.cpp" // 邻接表的基本操作
void DFSTree(ALGraph G,int v,CSTree &T)
{ // 从第v个顶点出发深度优先遍历图G，建立以T为根的生成树。算法7.8
  Boolean first=TRUE;
  int w;
  CSTree p,q;
  visited[v]=TRUE;
  for(w=FirstAdjVex(G,G.vertices[v].data);w>=0;w=NextAdjVex(G,G.vertices[v].data,
  G.vertices[w].data)) // w依次为v的邻接顶点
    if(!visited[w]) // w顶点不曾被访问
    {
      p=(CSTree)malloc(sizeof(CSNode)); // 分配孩子结点
      strcpy(p->data,G.vertices[w].data);
      p->firstchild=NULL;
      p->nextsibling=NULL;
      if(first)
      { // w是v的第一个未被访问的邻接顶点
        T->firstchild=p;
        first=FALSE; // 是根的第一个孩子结点
      }
      else // w是v的其它未被访问的邻接顶点
        q->nextsibling=p; // 是上一邻接顶点的兄弟姐妹结点(第1次不通过此处，以后q已赋值)
      q=p;
      DFSTree(G,w,q); // 从第w个顶点出发深度优先遍历图G，建立子生成树q
    }
}
void DFSForest(ALGraph G,CSTree &T)
{ // 建立无向图G的深度优先生成森林的(最左)孩子(右)兄弟链表T。算法7.7
  CSTree p,q;
```

```
  int v;
  T=NULL;
  for(v=0;v<G.vexnum;++v)
    visited[v]=FALSE; // 赋初值, visited[]在bo7-2.cpp中定义
  for(v=0;v<G.vexnum;++v) // 从第0个顶点找起
    if(!visited[v]) // 第v个顶点不曾被访问
    { // 第v顶点为新的生成树的根结点
      p=(CSTree)malloc(sizeof(CSNode)); // 分配根结点
      strcpy(p->data,G.vertices[v].data);
      p->firstchild=NULL;
      p->nextsibling=NULL;
      if(!T) // 是第一棵生成树的根(T的根)
        T=p;
      else // 是其它生成树的根(前一棵的根的"兄弟")
        q->nextsibling=p; // 第1次不通过此处, 以后q已赋值
      q=p; // q指示当前生成树的根
      DFSTree(G,v,p); // 建立以p为根的生成树
    }
}
void print(char *i)
{
  printf("%s ",i);
}
void main()
{
  ALGraph g;
  CSTree t;
  printf("请选择无向图\n");
  CreateGraph(g); // 构造无向图g
  Display(g); // 输出无向图g
  DFSForest(g,t); // 建立无向图g的深度优先生成森林的孩子—兄弟链表t
  printf("先序遍历生成森林: \n");
  PreOrderTraverse(t,print); // 先序遍历生成森林的孩子—兄弟链表t
  printf("\n");
}
```

程序运行结果(以教科书中图 7.3(a)的 G3 为例):

```
请选择无向图
请输入G的类型(有向图:0,有向网:1,无向图:2,无向网:3): 2↙
请输入G的顶点数,边数: 13,13↙(见图7-51)
请输入13个顶点的值(<2个字符):
A B C D E F G H I J K L M↙
请顺序输入每条弧(边)的弧尾和弧头(以空格作为间隔):
A B↙
A C↙
A F↙
A L↙
B M↙
D E↙
```

G H↙
G I↙
G K↙
H K↙
J L↙
J M↙
L M↙
无向图(见图7－52)
13个顶点:
A B C D E F G H I J K L M
13条弧(边):
A－L A－F A－C A－B
B－M

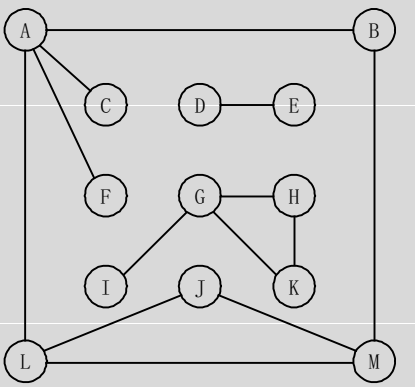


图 7－51　非连通无向图

D－E
G－K G－I G－H
H－K
J－M J－L
L－M

先序遍历生成森林:(见图7－53)
A L M J B F C D E G K H I

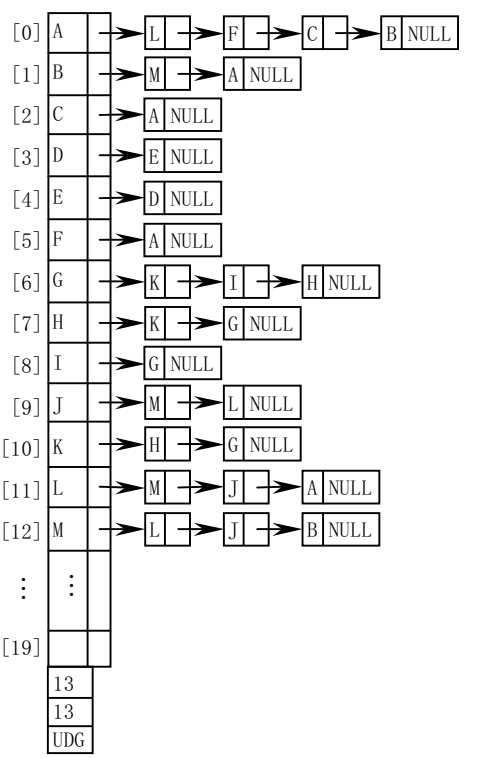


图 7－52　根据输入产生的邻接表

以上对图的输入产生的邻接表如图 7－52 所示，仍略去网的权值指针域，并用顶点名称代替顶点位置。调用算法 7.7 产生的生成森林如图 7－53 所示，此生成森林以孩子—兄弟二叉链表存储的结构如图 7－54 所示。

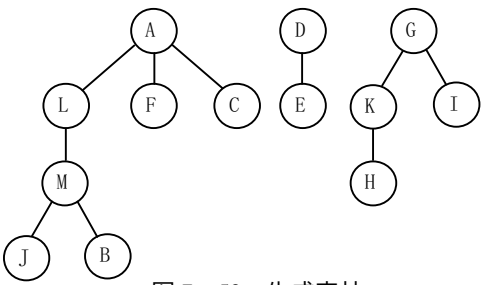


图 7－53　生成森林

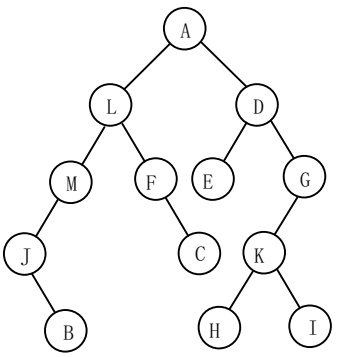


图 7－54　生成森林以孩子—兄弟二叉链表存储的结构

7.4.2　有向图的强连通分量

7.4.3　最小生成树

```
// algo7-2.cpp 实现算法7.9的程序
#include"c1.h"
typedef int VRType;
typedef char InfoType;
#define MAX_NAME 3 // 顶点字符串的最大长度+1
#define MAX_INFO 20 // 相关信息字符串的最大长度+1
typedef char VertexType[MAX_NAME];
#include"c7-1.h"
#include"bo7-1.cpp"
typedef struct
{ // 记录从顶点集U到V-U的代价最小的边的辅助数组定义(见图7－55)
  VertexType adjvex;
  VRType lowcost;
}minside[MAX_VERTEX_NUM];
int minimum(minside SZ,MGraph G)
{ // 求SZ.lowcost的最小正值，并返回其在SZ中的序号
  int i=0,j,k,min;
  while(!SZ[i].lowcost)
    i++;
  min=SZ[i].lowcost; // 第一个不为0的值
  k=i;
  for(j=i+1;j<G.vexnum;j++)
    if(SZ[j].lowcost>0&&min>SZ[j].lowcost) // 找到新的大于0的最小值
    {
      min=SZ[j].lowcost;
      k=j;
    }
  return k;
}
```

minside

	adjvex	lowcost
[0]		
[1]		
[2]		
⋮	⋮	⋮
[25]		

图 7－55　minside 类型

```
void MiniSpanTree_PRIM(MGraph G,VertexType u)
{ // 用普里姆算法从第u个顶点出发构造网G的最小生成树T，输出T的各条边。算法7.9
  int i,j,k;
  minside closedge;
  k=LocateVex(G,u);
  for(j=0;j<G.vexnum;++j) // 辅助数组初始化
  {
    strcpy(closedge[j].adjvex,u);
    closedge[j].lowcost=G.arcs[k][j].adj;
  }
  closedge[k].lowcost=0; // 初始,U={u}
  printf("最小代价生成树的各条边为\n");
  for(i=1;i<G.vexnum;++i)
  { // 选择其余G.vexnum-1个顶点
    k=minimum(closedge,G); // 求出T的下一个结点：第k顶点
    printf("(%s-%s)\n",closedge[k].adjvex,G.vexs[k]); // 输出生成树的边
    closedge[k].lowcost=0; // 第k顶点并入U集
    for(j=0;j<G.vexnum;++j)
      if(G.arcs[k][j].adj<closedge[j].lowcost)
      { // 新顶点并入U集后重新选择最小边
        strcpy(closedge[j].adjvex,G.vexs[k]);
        closedge[j].lowcost=G.arcs[k][j].adj;
      }
  }
}
void main()
{
  MGraph g;
  CreateUDN(g); // 构造无向网g
  Display(g); // 输出无向网g
  MiniSpanTree_PRIM(g,g.vexs[0]); // 用普里姆算法从第1个顶点出发输出g的最小生成树的各条边
}
```

程序运行结果(以教科书图 7.16 为例)：

```
请输入无向网G的顶点数,边数,边是否含其它信息(是:1,否:0)： 6,10,0↙
请输入6个顶点的值(<3个字符)：
V1 V2 V3 V4 V5 V6↙
请输入10条边的顶点1 顶点2 权值(以空格作为间隔)：
V1 V2 6↙
V1 V3 1↙
V1 V4 5↙
V2 V3 5↙
V2 V5 3↙
V3 V4 5↙
V3 V5 6↙
V3 V6 4↙
V4 V6 2↙
V5 V6 6↙
6个顶点10条边或弧的无向网。顶点依次是： V1 V2 V3 V4 V5 V6 (见图7－56)
```

```
G.arcs.adj:
    32767      6      1      5  32767  32767
        6  32767      5  32767      3  32767
        1      5  32767      5      6      4
        5  32767      5  32767  32767      2
    32767      3      6  32767  32767      6
    32767  32767      4      2      6  32767
G.arcs.info:
顶点1(弧尾) 顶点2(弧头) 该边或弧的信息:
最小代价生成树的各条边为
(V1-V3)
(V3-V6)
(V6-V4)
(V3-V2)
(V2-V5)
```

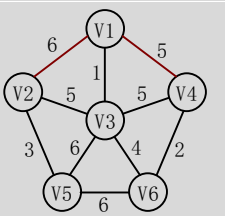


图 7－56　无向网

图 7－57 是根据以上程序运行的例子，显示了算法 7.9(普里姆算法)求最小生成树的过程。首先，主程序构造了图 7－56 所示的无向网。然后，调用 MiniSpanTree_PRIM()，由顶点 V1 开始，求该网的最小生成树。这样，最小生成树顶点集最初只有 V1，其中用到了辅助数组 closedge[]。closedge[i].lowcost 是最小生成树顶点集中的顶点到 i 点的最小权值。若 i 点属于最小生成树，则 closedge[i].lowcost=0。closedge[i].adjvex 是最小生成树顶点集中到 i 点为最小权值的那个顶点。图 7－57(a)显示了 closedge[]的初态。这时最小生成树顶点集中只有 V1，所以 closedge[i].adjvex 都是 V1，closedge[i].lowcost 是 V1 到 i 的权值。closedge[0].lowcost=0，说明 V1 已属于最小生成树顶点集了。在 closedge[].lowcost 中找最小正数，closedge[2].lowcost=1，是最小正数。令 k=2，将 V3 并入最小生成树的顶点集(令 closedge[2].lowcost=0)，输出边(V1—V3)。因为 V3 到 V2、V5 和 V6 的权值小于 V1 到它们的权值，故将它们的 closedge[].lowcost 替换为 V3 到它们的权值；将它们的 closedge[].adjvex 替换为 V3，如图 7－57(b)所示。重复这个过程，依次如图 7－57(c)、(d)和(e)所示。最后，closedge[]包含了最小生成树中每一条边的信息。

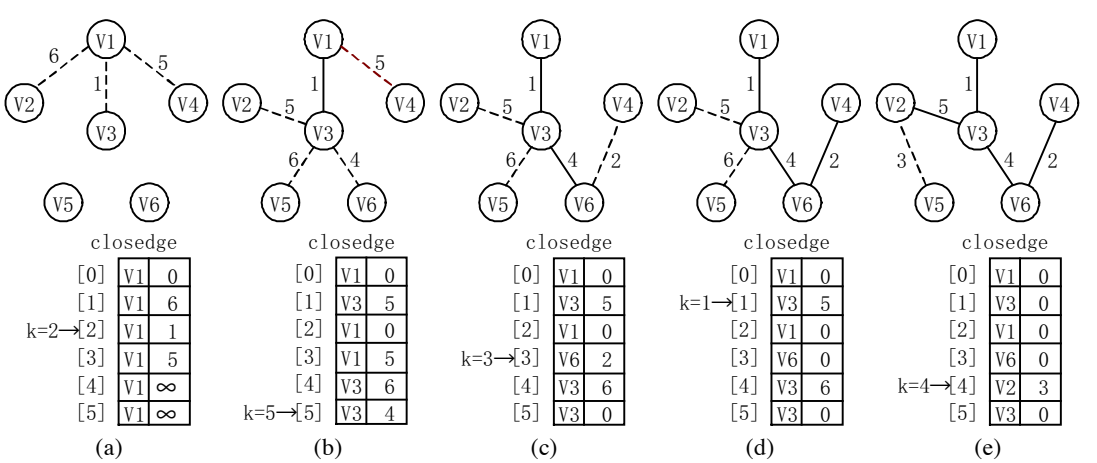


图 7－57　运行 algo7-2.cpp 过程图示

```
// algo7-8.cpp 克鲁斯卡尔算法求无向连通网的最小生成树的程序
#include"c1.h"
typedef int VRType;
typedef char InfoType;
#define MAX_NAME 3 // 顶点字符串的最大长度+1
#define MAX_INFO 20 // 相关信息字符串的最大长度+1
typedef char VertexType[MAX_NAME];
#include"c7-1.h"
#include"bo7-1.cpp"
void kruskal(MGraph G)
{
  int set[MAX_VERTEX_NUM],i,j;
  int k=0,a=0,b=0,min=G.arcs[a][b].adj;
  for(i=0;i<G.vexnum;i++)
    set[i]=i; // 初态，各顶点分别属于各个集合
  printf("最小代价生成树的各条边为\n");
  while(k<G.vexnum-1) // 最小生成树的边数小于顶点数-1
  { // 寻找最小权值的边
    for(i=0;i<G.vexnum;++i)
      for(j=i+1;j<G.vexnum;++j) // 无向网，只在上三角查找
        if(G.arcs[i][j].adj<min)
        {
          min=G.arcs[i][j].adj; // 最小权值
          a=i; // 边的一个顶点
          b=j; // 边的另一个顶点
        }
    min=G.arcs[a][b].adj=INFINITY; // 删除上三角中该边，下次不再查找
    if(set[a]!=set[b]) // 边的两顶点不属于同一集合
    {
      printf("%s-%s\n",G.vexs[a],G.vexs[b]); // 输出该边
      k++; // 边数+1
      for(i=0;i<G.vexnum;i++)
        if(set[i]==set[b]) // 将顶点b所在集合并入顶点a集合中
          set[i]=set[a];
    }
  }
}
void main()
{
  MGraph g;
  CreateUDN(g); // 构造无向网g
  Display(g); // 输出无向网g
  kruskal(g); // 用克鲁斯卡尔算法输出g的最小生成树的各条边
}
```

程序运行结果(以教科书图 7.16 为例)：

```
请输入无向网G的顶点数,边数,边是否含其它信息(是:1,否:0): 6,10,0↙
请输入6个顶点的值(<3个字符):
V1 V2 V3 V4 V5 V6↙
请输入10条边的顶点1 顶点2 权值(以空格作为间隔):
V1 V2 6↙
V1 V3 1↙
V1 V4 5↙
V2 V3 5↙
V2 V5 3↙
V3 V4 5↙
V3 V5 6↙
V3 V6 4↙
V4 V6 2↙
V5 V6 6↙
6个顶点10条边或弧的无向网。顶点依次是: V1 V2 V3 V4 V5 V6 (见图7-56)
G.arcs.adj:
      32767          6          1          5      32767      32767
          6      32767          5      32767          3      32767
          1          5      32767          5          6          4
          5      32767          5      32767      32767          2
      32767          3          6      32767      32767          6
      32767      32767          4          2          6      32767
G.arcs.info:
顶点1(弧尾) 顶点2(弧头) 该边或弧的信息:
最小代价生成树的各条边为
(V1-V3)
(V4-V6)
(V2-V5)
(V3-V6)
(V2-V3)
```

图 7-58 是根据以上程序运行的例子，显示了克鲁斯卡尔算法求最小生成树的过程。首先，主程序构造了图 7-56 所示的无向网。然后，调用 kruskal()，求该网的最小生成树。其中用到了辅助数组 set[]。set[i]表示第 i 个顶点所在的集合。设初态 set[i]=i，6 个顶点分属于 6 个集合，如图 7-58(a)所示。在邻接矩阵的上三角中找权值最小的边(因为是无向网)，边(V1—V3)的权值最小，将 V1 和 V3 并到 1 个集合中。方法是将 V3 的集合 set[2]赋值为 set[0](V1 的集合)，同时将该边删除(令其上在三角的值为无穷)并输出该边，如图 7-58(b)所示。用此方法依次将 V4 和 V6、V2 和 V5 分别并到 1 个集合中，如图 7-58(c)、图 7-58(d)所示。这时，邻接矩阵上三角中权值最小的边是(V3—V6)，这两顶点分属于两个集合 0 和 3。将集合 3 合并到集合 0 中。方法是把集合 3 中的 V4、V6 都并到集合 0 中，如图 7-58(e)所示。这时在邻接矩阵的上三角中首先找到的权值最小边是(V1—V4)，但它们属于同一个集合(set[0]=set[3]=0)，删除该边，继续查找。找到(V2—V3)是权值最小的边，且它们分属于不同的集合。把 V3 所在集合中的顶点都并到 V2 所在集合中，使所有顶点都在集合 1 中，如图 7-58(f)所示，最后构成了最小生成树。程序运行结果和普里姆算法的一样。

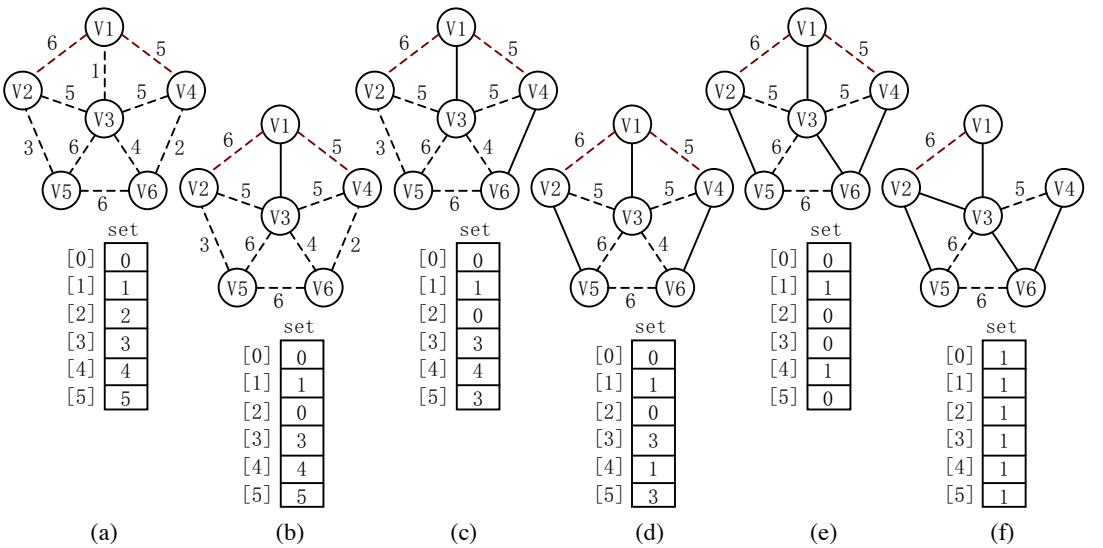


图 7 - 58　运行 algo7-8.cpp 过程图示

需要指出的是，虽然在 kruskal()中修改了无向网的邻接矩阵，由于 kruskal()的形参不是引用类型，故在主调函数中并没有改变图的结构。

7.4.4　关节点和重连通分量

```
// algo7-3.cpp 实现算法7.10、7.11的程序
#include"c1.h"
#define MAX_NAME 2 // 顶点字符串的最大长度+1
typedef int InfoType;
typedef char VertexType[MAX_NAME]; // 字符串类型
#include"c7-21.h" // 邻接表存储结构
#include"bo7-2.cpp" // 邻接表的基本操作
int count,lowcount=1; // 全局量count对访问顺序计数,lowcount对求得low值的顺序计数
int low[MAX_VERTEX_NUM],lowOrder[MAX_VERTEX_NUM];
// 全局数组，low[]存顶点的low值，lowOrder存顶点求得low值的顺序
void DFSArticul(ALGraph G,int v0)
{ // 从第v0个顶点出发深度优先遍历图G，查找并输出关节点
  int min,w;
  ArcNode *p;
  visited[v0]=min=++count;
  // v0是第count个访问的顶点，visited[]是全局变量,在bo7-2.cpp中定义,min的初值为v0的访问顺序
  for(p=G.vertices[v0].firstarc;p;p=p->nextarc) // 依次对v0的每个邻接顶点检查
  {
    w=p->data.adjvex; // w为v0的邻接顶点位置
    if(visited[w]==0) // w未曾访问，是v0的孩子
    {
      DFSArticul(G,w);
      // 从第w个顶点出发深度优先遍历图G，查找并输出关节点。返回前求得low[w]
      if(low[w]<min) // 如果v0的孩子结点w的low[]小，这说明孩子结点还与其它结点(祖先)相邻
        min=low[w]; // 取min值为孩子结点的low[]，则v0不是关节点
      else if(low[w]>=visited[v0]) // v0的孩子结点w只与v0相连，则v0是关节点
        printf("%d %s\n",v0,G.vertices[v0].data); // 输出关节点v0
```

```
    }
    else if(visited[w]<min) // w已访问，则w是v0在生成树上的祖先，它的访问顺序必小于min
      min=visited[w]; // 故取min为visited[w]
  }
  low[v0]=min; // vo的low[]值为三者中的最小值
  lowOrder[v0]=lowcount++; // 记录v0求得low[]值的顺序(附加),总是在返回主调函数之前求得low[]
}
void FindArticul(ALGraph G)
{ // 连通图G以邻接表作存储结构，查找并输出G上全部关节点。全局量count对访问计数。算法7.10
  int i,v;
  ArcNode *p;
  count=1; // 访问顺序
  visited[0]=count; // 设定邻接表上0号顶点为生成树的根，第1个被访问
  for(i=1;i<G.vexnum;++i)
    visited[i]=0; // 其余顶点尚未访问，设初值为0
  p=G.vertices[0].firstarc; // p指向根结点的第1个邻接顶点
  v=p->data.adjvex; // v是根结点的第1个邻接顶点的序号
  DFSArticul(G,v); // 从第v顶点出发深度优先查找关节点
  if(count<G.vexnum) // 由根结点的第1个邻接顶点深度优先遍历G，访问的顶点数少于G的顶点数
  { // 说明生成树的根有至少两棵子树，则根是关节点
    printf("%d %s\n",0,G.vertices[0].data); // 根是关节点，输出根
    while(p->nextarc) // 根有下一个邻接点
    {
      p=p->nextarc; // p指向根的下一个邻接点
      v=p->data.adjvex;
      if(visited[v]==0) // 此邻接点未被访问
        DFSArticul(G,v); // 从此顶点出发深度优先查找关节点
    }
  }
}
void main()
{
  int i;
  ALGraph g;
  printf("请选择无向图\n");
  CreateGraph(g); // 构造无向图g
  Display(g); // 输出无向图g
  printf("输出关节点: \n");
  FindArticul(g); // 求连通图g的关节点
  printf(" i G.vertices[i].data visited[i] low[i] lowOrder[i]\n"); // 输出辅助变量
  for(i=0;i<g.vexnum;++i)
    printf("%2d %9s %14d %8d %8d\n",i,g.vertices[i].data,visited[i],low[i],lowOrder[i]);
}
```

程序运行结果(以教科书中图 7.19、7.20 为例):

```
请选择无向图
请输入图的类型(有向图:0,有向网:1,无向图:2,无向网:3): 2↙(见图7－59)
请输入图的顶点数,边数: 13,17↙
请输入13个顶点的值(<2个字符):
```

```
A B C D E F G H I J K L M↙
请输入每条弧(边)的弧尾和弧头(以空格作为间隔):
A B↙
A C↙
A F↙
A L↙
B C↙
B D↙
B G↙
B H↙
B M↙
D E↙
G H↙
G I↙
G K↙
H K↙
J L↙
J M↙
L M↙
无向图(见图7－60)
13个顶点:
A B C D E F G H I J K L M
17条弧(边):
A→L A→F A→C A→B
B→M B→H B→G B→D B→C
D→E
G→K G→I G→H
H→K
J→M J→L
L→M
输出关节点:
6 G
1 B
3 D
1 B
0 A
```

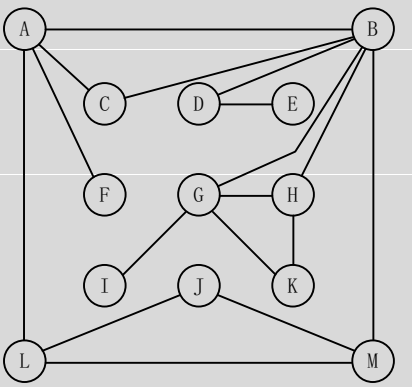


图 7－59　连通无向图

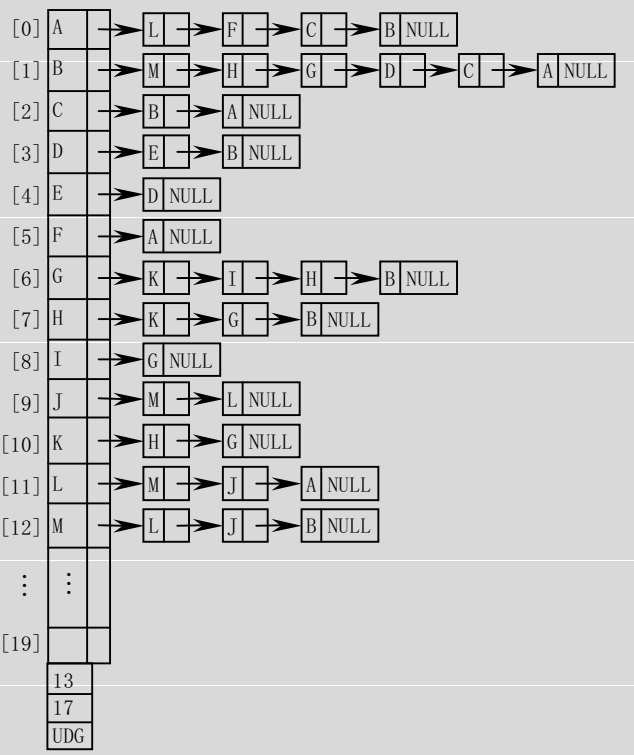


图 7－60　根据输入产生的邻接表

i	G.vertices[i].data	visited[i]	low[i]	lowOrder[i]
0	A	1	0	0 (没求A的low[])
1	B	5	1	9
2	C	12	1	8
3	D	10	5	7
4	E	11	10	6
5	F	13	1	12
6	G	8	5	3
7	H	6	5	5

8	I	9	8	2
9	J	4	2	1
10	K	7	5	4
11	L	2	1	11
12	M	3	1	10

运行 algo7-3.cpp 构造的深度优先生成树如图 7－61 所示。图 7－61 和图 7－59 的拓扑结构是一样的。5 条用虚线表示的边是构造深度优先生成树多余的边，它们是连接祖先的回边。如果一个结点不仅有连向双亲的边，还有连向祖先的回边。则对这个结点来说，它的双亲结点不是关节点。如图 7－61 中的结点 B，它有连向双亲结点 M 的边，还有连向祖先结点 A 的边。这样，如果删除结点 M，B 仍然与图的其它部分连通(重连通)。而图 7－61 中的结点 I，它只有连向双亲结点 G 的边。一旦结点 G 被删除，结点 I 就与图的其它部分不连通，也就是一个连通分量被分割成了多个连通分量。结点 G 被称为关节点。

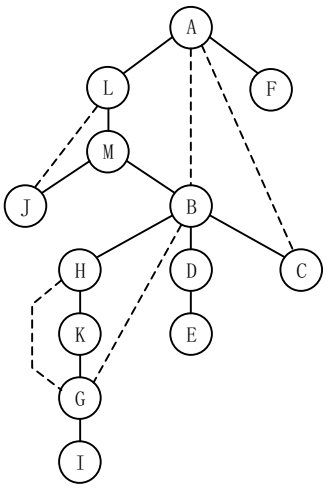


图 7－61　深度优先生成树

如何确定关节点？算法 7.10、7.11 的思路是这样的：首先在深度优先遍历图时，不仅标注某顶点是否被访问，还标注它的访问顺序。visited[]不再只是 FALSE 和 TRUE，而是 1～顶点数。由于采用深度优先遍历，某结点的祖先被访问的顺序必先于该结点被访问的顺序。仍以图 7－61 为例，由第 1 个结点 A 深度优先遍历的顺序是：A、L、M、J、B、…… 增加 1 个辅助数组 low[]，对顶点 v，定义 low[v]=min(visited[v], low[w], visited[k])。其中 w 和 k 分别是 v 的孩子和由回边相连的祖先。由算法 7.11 可知，low[]是在递归调用返回之前求得的。所以，求得 low[]的顺序是：…… B、M、L、…… 也就是说，孩子的 low[]是先于双亲的 low[]而获得的。这可由程序运行结果中的 lowOrder[]看出(增加辅助数组 lowOrder[]的目的就是帮助分析求得 low[]的顺序)。

如果顶点 v 有孩子 w，且有 low[w]≥visited[v]，则顶点 v 必为关节点。下面分析几种可能存在的情况：

(1) 如果顶点 v 有通过回边相连的祖先 k，则 low[v]=visited[k](祖先顶点 k 被访问的顺序)。同时 k 也是 v 的双亲 u 的祖先或双亲，故有 low[v]=visited[k]＜visited[u](结点祖先或双亲必先于该结点被访问)。不满足判定关节点的公式，故 u 不是 v 的关节点。这种情况如图 7－61 中顶点 B 的 low[]=顶点 A 的 visited[]=1，其双亲 M 的 visited[]=3，故 M 不是 B 的关节点。

(2) 如果顶点 v 没有通过回边相连的祖先，但有孩子 w，而孩子顶点 w 有通过回边相连的祖先 k，则 low[w]=visited[k]，而 k 也是 v 的双亲 u 的祖先，仍有 visited[k]≤visited[u]。如顶点 K 没有通过回边相连的祖先，但有孩子 G，而 G 有通过回边相连的祖先 B。顶点 G 的 low[]等于顶点 B 的 visited[]=5，也等于顶点 K 的 low[]。而 K 的双亲 H 的 visited[]=6，故 H 不是 K 的关节点。

(3) 如果顶点 v 既无孩子又无通过回边相连的祖先，则其双亲结点 u 是关节点。在这

种情况下，low[v]=visited[v](顶点 v 被访问的顺序)。而 u 被访问的顺序必定小于 v 的，故有 low[v]=visited[v]＞visited[u]。所以 u 是 v 的关节点。如顶点 E 就是既无孩子又无通过回边相连的祖先，则其双亲结点 D 是 E 的关节点。

(4) 如果顶点 v 没有通过回边相连的祖先，虽有孩子顶点 w，但 w 也没有通过回边相连的祖先，则 v 的双亲结点 u 是关节点。在这种情况下，low[w]= visited[w](顶点 w 被访问的顺序)＞low[v]=visited[v](顶点 v 被访问的顺序)＞visited[u]，故 u 是 v 的关节点。如顶点 D 虽有孩子顶点 E，但 E 没有通过回边相连的祖先，则 low[D]=5=visited[B]。故 B 是 D 的关节点。

通过 low[w]≥visited[v]来判断连通图关节点的方法不能用于根结点。因为根结点的 visited[]=1，是最小值。判断根结点是否为关节点要看它有几棵子树，如果超过 1 棵，则根结点就是关节点。原因是，它的每棵子树上的结点都和其它子树的不相连。否则在深度优先遍历其它子树时，就会遍历到，也就不成为根结点的子树了。所以算法 7.10 在深度优先遍历时，不是直接从根结点遍历，而是从根结点的第 1 个邻接顶点开始遍历。当遍历完这个邻接顶点的生成子树，若还有顶点没被访问，则说明根结点是关节点。如图 7－61 所示，对根结点 A 的第 1 棵子树 L 遍历结束后，A 还有邻接点 F 没被访问到。说明除根结点 A 之外，L 子树上的任何一个结点都不和 F 邻接。这样，若根结点 A 被删除，原图就会被分割成 L 子树和 F 两部分。故根结点 A 是关节点。

运行 algo7-3.cpp 在输出关节点时，B 被输出了 2 次。其原因是删除 B 使连通图分割成 3 个连通分量。

7.5　有向无环图及其应用

7.5.1　拓扑排序

```
// func7-1.cpp algo7-4.cpp和algo7-5.cpp要调用
void FindInDegree(ALGraph G,int indegree[])
{ // 求顶点的入度，算法7.12、7.13调用
  int i;
  ArcNode *p;
  for(i=0;i<G.vexnum;i++)
    indegree[i]=0; // 赋初值
  for(i=0;i<G.vexnum;i++)
  {
    p=G.vertices[i].firstarc;
    while(p)
    {
      indegree[p->data.adjvex]++;
      p=p->nextarc;
    }
  }
}

// algo7-4.cpp 输出有向图的一个拓扑序列。实现算法7.12的程序
#include"c1.h"
```

```
#define MAX_NAME 5 // 顶点字符串的最大长度
typedef int InfoType;
typedef char VertexType[MAX_NAME]; // 字符串类型
#include"c7-21.h" // 邻接表存储结构
#include"bo7-2.cpp" // 邻接表存储结构的基本操作
#include"func7-1.cpp"
typedef int SElemType; // 栈元素类型
#include"c3-1.h" // 顺序栈的存储结构
#include"bo3-1.cpp" // 顺序栈的基本操作
Status TopologicalSort(ALGraph G)
{ // 有向图G采用邻接表存储结构。若G无回路，则输出G的顶点的一个拓扑序列并返回OK,
  // 否则返回ERROR。算法7.12
  int i,k,count=0; // 已输出顶点数，初值为0
  int indegree[MAX_VERTEX_NUM]; // 入度数组，存放各顶点当前入度数
  SqStack S;
  ArcNode *p;
  FindInDegree(G,indegree); // 对各顶点求入度indegree[]，在func7-1.cpp中
  InitStack(S); // 初始化零入度顶点栈S
  for(i=0;i<G.vexnum;++i) // 对所有顶点i
    if(!indegree[i]) // 若其入度为0
      Push(S,i); // 将i入零入度顶点栈S
  while(!StackEmpty(S)) // 当零入度顶点栈S不空
  {
    Pop(S,i); // 出栈1个零入度顶点的序号，并将其赋给i
    printf("%s ",G.vertices[i].data); // 输出i号顶点
    ++count; // 已输出顶点数+1
    for(p=G.vertices[i].firstarc;p;p=p->nextarc)
    { // 对i号顶点的每个邻接顶点
      k=p->data.adjvex; // 其序号为k
      if(!(--indegree[k])) // k的入度减1，若减为0，则将k入栈S
        Push(S,k);
    }
  }
  if(count<G.vexnum) // 零入度顶点栈S已空，图G还有顶点未输出
  {
    printf("此有向图有回路\n");
    return ERROR;
  }
  else
  {
    printf("为一个拓扑序列。\n");
    return OK;
  }
}
void main()
{
  ALGraph f;
  printf("请选择有向图\n");
  CreateGraph(f); // 构造有向图f，在bo7-2.cpp中
  Display(f); // 输出有向图f，在bo7-2.cpp中
  TopologicalSort(f); // 输出有向图f的1个拓扑序列
}
```

程序运行结果(以教科书图 7.28 为例):

```
请选择有向图
请输入G的类型(有向图:0,有向网:1,无向图:2,无向网:3): 0↙(见图7-62)
请输入G的顶点数,边数: 6,8↙
请输入6个顶点的值(<5个字符):
V1 V2 V3 V4 V5 V6↙
请顺序输入每条弧(边)的弧尾和弧头(以空格作为间隔):
V1 V2↙
V1 V3↙
V1 V4↙
V3 V2↙
V3 V5↙
V4 V5↙
V6 V4↙
V6 V5↙
有向图(见图7-63)
6个顶点:
V1 V2 V3 V4 V5 V6
8条弧(边):
V1→V4 V1→V3 V1→V2
V3→V5 V3→V2
V4→V5
V6→V5 V6→V4
V6 V1 V3 V2 V4 V5 为一个拓扑序列。
```

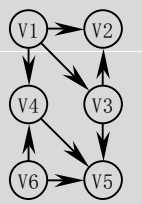


图 7-62　有向图

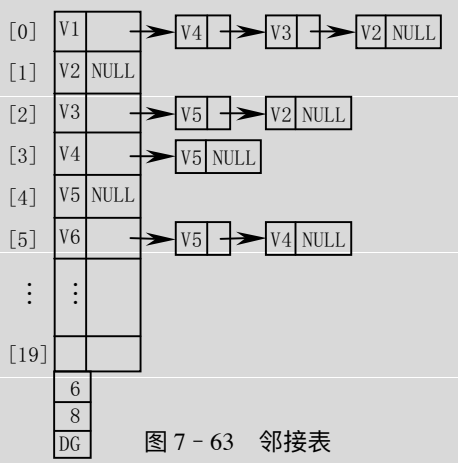
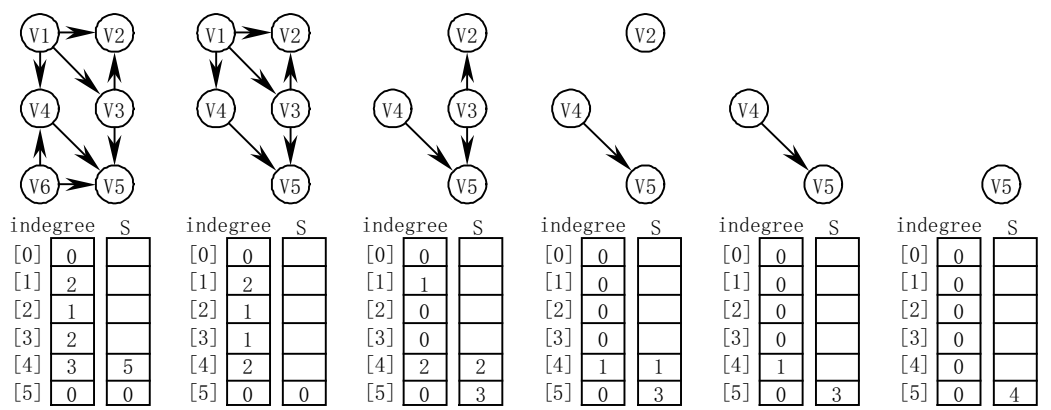


图 7-63　邻接表

图 7-64 显示了运行 algo7-4.cpp 的过程。其中的 S 栈也可用队列代替，这样将输出一个不同的拓扑序列。



(a) while 循环前　(b) 输出 V6　(c) 输出 V1　(d) 输出 V3　(e) 输出 V2　(f) 输出 V4

图 7-64　运行 algo7-4.cpp 过程图示

7.5.2　关键路径

```
// algo7-5.cpp 求关键路径。实现算法7.13、7.14的程序
#include"c1.h"
#define MAX_NAME 5 // 顶点字符串的最大长度+1
typedef int InfoType;
typedef char VertexType[MAX_NAME]; // 字符串类型
#include"c7-21.h"
#include"bo7-2.cpp"
#include"func7-1.cpp"
int ve[MAX_VERTEX_NUM]; // 事件最早发生时间，全局变量(用于算法7.13和算法7.14)
typedef int SElemType; // 栈元素类型
#include"c3-1.h" // 顺序栈的存储结构
#include"bo3-1.cpp" // 顺序栈的基本操作
Status TopologicalOrder(ALGraph G,SqStack &T)
{ // 算法7.13 有向网G采用邻接表存储结构,求各顶点事件的最早发生时间ve(全局变量)。T为拓扑序列
  // 顶点栈,S为零入度顶点栈。若G无回路,则用栈T返回G的一个拓扑序列,且函数值为OK；否则为ERROR
  int i,k,count=0; // 已入栈顶点数，初值为0
  int indegree[MAX_VERTEX_NUM]; // 入度数组，存放各顶点当前入度数
  SqStack S;
  ArcNode *p;
  FindInDegree(G,indegree); // 对各顶点求入度indegree[]，在func7-1.cpp中
  InitStack(S); // 初始化零入度顶点栈S
  printf("拓扑序列：");
  for(i=0;i<G.vexnum;++i) // 对所有顶点i
    if(!indegree[i]) // 若其入度为0
      Push(S,i); // 将i入零入度顶点栈S
  InitStack(T); // 初始化拓扑序列顶点栈
  for(i=0;i<G.vexnum;++i) // 初始化ve[]=0(最小值,先假定每个事件都不受其它事件约束)
    ve[i]=0;
  while(!StackEmpty(S)) // 当零入度顶点栈S不空
  {
    Pop(S,i); // 从栈S将已拓扑排序的顶点j弹出
    printf("%s ",G.vertices[i].data);
    Push(T,i); // j号顶点入逆拓扑排序栈T(栈底元素为拓扑排序的第1个元素)
    ++count; // 对入栈T的顶点计数
    for(p=G.vertices[i].firstarc;p;p=p->nextarc)
    { // 对i号顶点的每个邻接点
      k=p->data.adjvex; // 其序号为k
      if(--indegree[k]==0) // k的入度减1，若减为0，则将k入栈S
        Push(S,k);
      if(ve[i]+*(p->data.info)>ve[k]) // *(p->data.info)是<i,k>的权值
        ve[k]=ve[i]+*(p->data.info); // 顶点k事件的最早发生时间要受其直接前驱顶点i事件的
    }                 // 最早发生时间和<i,k>的权值约束。由于i已拓扑有序，故ve[i]不再改变
  }
  if(count<G.vexnum)
  {
    printf("此有向网有回路\n");
    return ERROR;
  }
```

```
  else
    return OK;
}
Status CriticalPath(ALGraph G)
{ // 算法7.14 G为有向网，输出G的各项关键活动
  int vl[MAX_VERTEX_NUM]; // 事件最迟发生时间
  SqStack T;
  int i,j,k,ee,el,dut;
  ArcNode *p;
  if(!TopologicalOrder(G,T)) // 产生有向环
    return ERROR;
  j=ve[0]; // j的初值
  for(i=1;i<G.vexnum;i++)
    if(ve[i]>j)
      j=ve[i]; // j=Max(ve[]) 完成点的最早发生时间
  for(i=0;i<G.vexnum;i++) // 初始化顶点事件的最迟发生时间
    vl[i]=j; // 为完成点的最早发生时间(最大值)
  while(!StackEmpty(T)) // 按拓扑逆序求各顶点的vl值
    for(Pop(T,j),p=G.vertices[j].firstarc;p;p=p->nextarc)
    { // 弹出栈T的元素,赋给j,p指向j的后继事件k,事件k的最迟发生时间已确定(因为是逆拓扑排序)
      k=p->data.adjvex;
      dut=*(p->data.info); // dut=<j,k>的权值
      if(vl[k]-dut<vl[j])
        vl[j]=vl[k]-dut; // 事件j的最迟发生时间要受其直接后继事件k的最迟发生时间
    }                    // 和<j,k>的权值约束。由于k已逆拓扑有序，故vl[k]不再改变
  printf("\ni ve[i] vl[i]\n");
  for(i=0;i<G.vexnum;i++) // 初始化顶点事件的最迟发生时间
  {
    printf("%d   %d     %d",i,ve[i],vl[i]);
    if(ve[i]==vl[i])
      printf(" 关键路径经过的顶点");
    printf("\n");
  }
  printf("j   k  权值  ee  el\n");
  for(j=0;j<G.vexnum;++j) // 求ee，el和关键活动
    for(p=G.vertices[j].firstarc;p;p=p->nextarc)
    {
      k=p->data.adjvex;
      dut=*(p->data.info); // dut=<j,k>的权值
      ee=ve[j]; // ee=活动<j,k>的最早开始时间(在j点)
      el=vl[k]-dut; // el=活动<j,k>的最迟开始时间(在j点)
      printf("%s→%s %3d %3d %3d ",G.vertices[j].data,G.vertices[k].data,dut,ee,el);
      // 输出各边的参数
      if(ee==el) // 是关键活动
        printf("关键活动");
      printf("\n");
    }
  return OK;
}
void main()
{
```

```
  ALGraph h;
  printf("请选择有向网\n");
  CreateGraph(h); // 构造有向网h，在bo7-2.cpp中
  Display(h); // 输出有向网h，在bo7-2.cpp中
  CriticalPath(h); // 求h的关键路径
}
```

程序运行结果(以教科书中图 7.30 为例)：

```
请选择有向网(见图7－65)
请输入图的类型(有向图:0,有向网:1,无向图:2,无向网:3): 1↙
请输入图的顶点数,边数: 6,8↙
请输入6个顶点的值(<5个字符):
V1 V2 V3 V4 V5 V6↙
请输入每条弧(边)的权值、弧尾和弧头(以空格作为间隔):
3 V1 V2↙
2 V1 V3↙
2 V2 V4↙
3 V2 V5↙
4 V3 V4↙
3 V3 V6↙
2 V4 V6↙
1 V5 V6↙
有向网(见图7－66)
6个顶点:
V1 V2 V3 V4 V5 V6
8条弧(边):
V1→V3 :2 V1→V2 :3
V2→V5 :3 V2→V4 :2
V3→V6 :3 V3→V4 :4
V4→V6 :2
V5→V6 :1
```

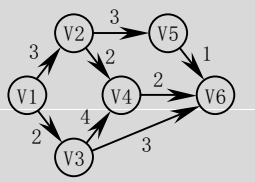


图 7－65　有向网

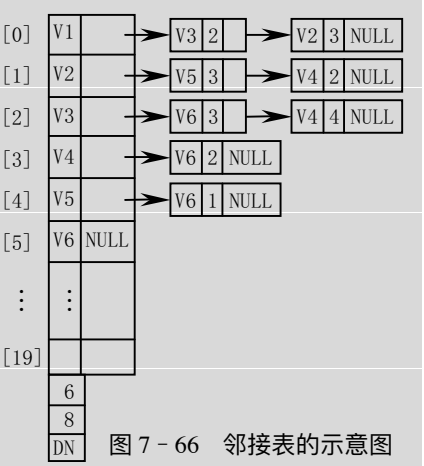


图 7－66　邻接表的示意图

```
拓扑序列: V1 V2 V5 V3 V4 V6
i ve[i] vl[i] (见图7－67)
0   0    0 关键路径经过的顶点
1   3    4
2   2    2 关键路径经过的顶点
3   6    6 关键路径经过的顶点
4   6    7
5   8    8 关键路径经过的顶点
j   k  权值  ee  el
V1→V3   2   0   0 关键活动
V1→V2   3   0   1
V2→V5   3   3   4
V2→V4   2   3   4
V3→V6   3   2   5
V3→V4   4   2   2 关键活动
V4→V6   2   6   6 关键活动
V5→V6   1   6   7
```

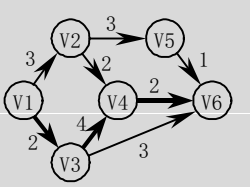


图 7－67　关键路径

图 7－66 是邻接表的示意图。为直观和方便起见，表结点中邻接顶点的序号直接用其名称代替，动态生成的权值也直接写在权值的指针域中。

运行 algo7-5.cpp 用到 2 个辅助数组：事件(顶点)最早发生时间 ve[]和事件最迟发生时间 vl[]。

顶点 i 的事件(顶点)最早发生时间 ve[i]取决于其直接前驱事件的发生时间和二者之间活动的持续时间(弧的权值)。如图 7－65 中，V2 事件的最早发生时间取决于 V1 的 ve[]和<V1, V2>的权值 3。即 V2 的 ve[]等于 V1 的 ve[]+3。如果顶点 i 有多个直接前驱事件则 ve[i]取最大值，如 V4 的 ve[]取决于 V2 的 ve[]+<V2, V4>的权值 2 与 V3 的 ve[]+<V3, V4>的权值 4 这 2 者中的大值。没有直接前驱事件的顶点，其 ve[]=0(是最小值)，如顶点 V1。由于求顶点的 ve[]要求其直接前驱的 ve[]已知，故应先对有向网进行拓扑排序。排序前设所有顶点的 ve[]初值=0(最小值)，当出现较大的值，则用这个大值更新 ve[]。调用 TopologicalOrder()后，ve[]如以上程序运行结果所示。

所谓事件最迟发生时间是指在不影响工期的情况下，某事件可以最迟发生的时间。顶点 i 的事件(顶点)最迟发生时间 vl[i]取决于其直接后继事件的最迟发生时间和二者之间活动的持续时间(弧的权值)。如图 7－65 中，V4 事件的最迟发生时间取决于 V6 的 vl[]和<V4, V6>的权值 2。即 V4 的 vl[]等于 V6 的 vl[]-2。如果顶点 i 有多个直接后继事件则 vl[i]取最小值。如 V3 的 vl[]取决于 V4 的 vl[]-<V3, V4>的权值 4 与 V6 的 vl[]-<V3, V6>的权值 3 这 2 者中的小值。没有直接后继事件的顶点，其 vl[]=ve[](是最大值，已先期求出)，如顶点 V6。由于求顶点的 vl[]时，要求其直接后继的 vl[]已知，故应先形成有向网的逆拓扑序列。TopologicalOrder()将已拓扑排序的顶点入栈 T，形成逆拓扑序列。排序前设所有顶点的 vl[]初值等于没有后继的那个顶点的 vl[](最大值)，本例中这个顶点是 V6。当出现较小的值，则用这个小值更新 vl[]。调用 CriticalPath()，vl[]如以上程序运行结果所示。

若对于顶点 i，有 ve[i]=vl[i]，即事件最早发生时间等于事件最迟发生时间。说明为保证工期，事件(顶点)i 的发生时间不可变更。如果变小，则前面的活动(入弧)还没完成；如果变大，则影响后继事件按时完成。因此，顶点 i 是关键路径要经过的点。如以上程序运行结果所示，V1、V3、V4 和 V6 是关键路径要经过的顶点。但光根据这些顶点，还不能确定关键路径。如图 7－65 中，虽然 V3、V4 和 V6 是关键路径要经过的顶点，但弧<V3, V4>、<V3, V6>和<V4, V6>中哪个是关键路径还不清楚。

如果一个活动(弧)<j, k>，它的前端事件的最早发生时间 ve[j]+<j, k>的权值等于 vl[k](后端事件的最迟发生时间)，那么这个活动的发生时间就没有变更的余地，它就是整个关键路径的一部分。求得 ve[]和 vl[]后，对于每一个弧<j, k>，判断它的 ve[j]是否等于它的 vl[k]+dut(弧的权值)，可求出所有关键路径。图 7－67 中粗箭头弧是关键路径。

7.6　最短路径

7.6.1　从某个源点到其余各顶点的最短路径

```
// algo7-6.cpp 实现算法7.15的程序。迪杰斯特拉算法的实现
```

```
#include"c1.h"
#define MAX_NAME 5 // 顶点字符串的最大长度+1
#define MAX_INFO 20 // 相关信息字符串的最大长度+1
typedef int VRType;
typedef char InfoType;
typedef char VertexType[MAX_NAME];
#include"c7-1.h" // 邻接矩阵存储结构
#include"bo7-1.cpp" // 邻接矩阵存储结构的基本操作
typedef int PathMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM]; // 路径矩阵，二维数组
typedef int ShortPathTable[MAX_VERTEX_NUM]; // 最短距离表，一维数组
void ShortestPath_DIJ(MGraph G,int v0,PathMatrix P,ShortPathTable D)
{ // 用Dijkstra算法求有向网G的v0顶点到其余顶点v的最短路径P[v]及带权长度
  // D[v]。若P[v][w]为TRUE，则w是从v0到v当前求得最短路径上的顶点。
  // final[v]为TRUE当且仅当v∈S，即已经求得从v0到v的最短路径 算法7.15
  int v,w,i,j,min;
  Status final[MAX_VERTEX_NUM]; // 辅助矩阵，为真表示该顶点到v0的最短距离已求出，初值为假
  for(v=0;v<G.vexnum;++v)
  {
    final[v]=FALSE; // 设初值
    D[v]=G.arcs[v0][v].adj; // D[]存放v0到v的最短距离，初值为v0到v的直接距离
    for(w=0;w<G.vexnum;++w)
      P[v][w]=FALSE; // 设P[][]初值为FALSE，没有路径
    if(D[v]<INFINITY) // v0到v有直接路径
      P[v][v0]=P[v][v]=TRUE; // 一维数组p[v][]表示源点v0到v最短路径通过的顶点
  }
  D[v0]=0; // v0到v0距离为0
  final[v0]=TRUE; // v0顶点并入S集
  for(i=1;i<G.vexnum;++i) // 其余G.vexnum-1个顶点
  { // 开始主循环，每次求得v0到某个顶点v的最短路径，并将v并入S集
    min=INFINITY; // 当前所知离v0顶点的最近距离，设初值为∞
    for(w=0;w<G.vexnum;++w) // 对所有顶点检查
      if(!final[w]&&D[w]<min) //在S集之外的顶点中找离v0最近的顶点，并将其赋给v，距离赋给min
      {
        v=w;
        min=D[w];
      }
    final[v]=TRUE; // 将v并入S集
    for(w=0;w<G.vexnum;++w) // 根据新并入的顶点，更新不在S集的顶点到v0的距离和路径数组
      if(!final[w]&&min<INFINITY&&G.arcs[v][w].adj<INFINITY&&(min+G.arcs[v][w].adj<D[w]))
      { // w不属于S集且v0→v→w的距离＜目前v0→w的距离
        D[w]=min+G.arcs[v][w].adj; // 更新D[w]
        for(j=0;j<G.vexnum;++j) // 修改P[w]，v0到w经过的顶点包括v0到v经过的顶点再加上顶点w
          P[w][j]=P[v][j];
        P[w][w]=TRUE;
      }
  }
}
void main()
{
  int i,j;
  MGraph g;
```

```
  PathMatrix p; // 二维数组，路径矩阵
  ShortPathTable d; // 一维数组，最短距离表
  CreateDN(g); // 构造有向网g
  Display(g); // 输出有向网g
  ShortestPath_DIJ(g,0,p,d);//以g中位置为0的顶点为源点，球其到其余各顶点的最短距离。存于d中
  printf("最短路径数组p[i][j]如下:\n");
  for(i=0;i<g.vexnum;++i)
  {
    for(j=0;j<g.vexnum;++j)
      printf("%2d",p[i][j]);
    printf("\n");
  }
  printf("%s到各顶点的最短路径长度为\n",g.vexs[0]);
  for(i=0;i<g.vexnum;++i)
    if(i!=0)
      printf("%s-%s:%d\n",g.vexs[0],g.vexs[i],d[i]);
}
```

程序运行结果(以教科书图 7.34 的 G6 为例)：

```
请输入有向网G的顶点数,弧数,弧是否含其它信息(是:1,否:0): 6,8,0↙(见图7-68)
请输入6个顶点的值(<5个字符):
V0 V1 V2 V3 V4 V5↙
请输入8条弧的弧尾 弧头 权值(以空格作为间隔):
V0 V5 100↙
V0 V4 30↙
V0 V2 10↙
V1 V2 5↙
V2 V3 50↙
V3 V5 10↙
V4 V3 20↙
V4 V5 60↙
6个顶点8条边或弧的有向网。顶点依次是: V0 V1 V2 V3 V4 V5
G.arcs.adj:
      32767      32767         10      32767         30        100
      32767      32767          5      32767      32767      32767
      32767      32767      32767         50      32767      32767
      32767      32767      32767      32767      32767         10
      32767      32767      32767         20      32767         60
      32767      32767      32767      32767      32767      32767
G.arcs.info:
顶点1(弧尾) 顶点2(弧头) 该边或弧的信息:
最短路径数组p[i][j]如下:
 0 0 0 0 0 0
 0 0 0 0 0 0
 1 0 1 0 0 0
 1 0 0 1 1 0
 1 0 0 0 1 0
 1 0 0 1 1 1
```

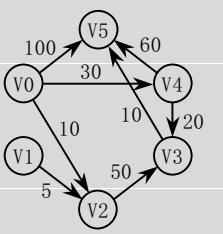
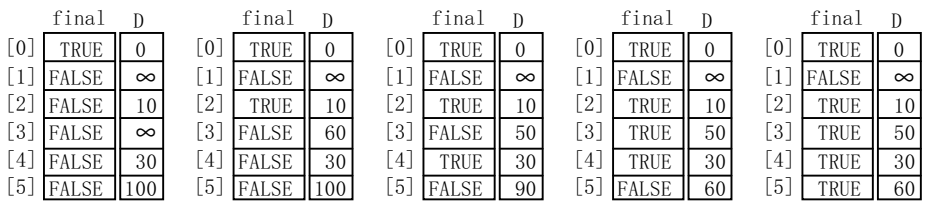


图 7-68　有向网

```
V0到各顶点的最短路径长度为
V0-V1:32767
V0-V2:10
V0-V3:50
V0-V4:30
V0-V5:60
```

函数 ShortestPath_DIJ()利用 2 个辅助数组 final[]和 D[]求得给定点 v0 到图 G 中其余各顶点的最短距离。D[]存放当前 v0 到其余各顶点的最短距离，final[]的初值为 FALSE。final[]的值为 TRUE，表示 v0 到该顶点的最短距离已求出。图 7－69 通过 final[]和 D[]的变化演示了求解过程。



(a) 初值　(b) V2 并入 S　(c) V4 并入 S　(d) V3 并入 S　(e) V5 并入 S

图 7－69　运行 algo7-6.cpp 过程图示

首先，final[]的初值中只有 final[v0]为真，最短距离顶点集 S 中只有顶点 v0(源点，此例中实参为 V0)。D[]的初值是邻接矩阵中 v0 行所对应的值。另令 D[v0]=0(v0 到自己的距离当然为 0)。v0 到某顶点 i 的最短距离可能是二者的直接距离 G.arcs[v0][i].adj，也可能是由 v0 出发，经过其它顶点，最后到达顶点 i 的距离。如图 7－68 中 V0 到 V5 的最短距离不是它们的直接距离 100，而是由 V0 经过 V4、V3，最后到达 V5 的距离 60。

根据图 7－69(a)，在不属于 S 集的顶点中，V0 到 V2 的距离最短。可以断定，V0 到 V2 的距离 10 是最短距离。V0 通过其它顶点绕道到达 V2 的距离一定会比 10 大，故将 V2 并入 S 集中(final[2]=TRUE)。并考察 S 集外的顶点中，有没有哪个顶点 i，使得 V0 先到 V2(距离为 10)，再由 V2 到达顶点 i 比直接从 V0 到达顶点 i 的距离要小? 也就是满足 10+G.arcs[2][i].adj＜D[i]。如有，则改写 D[i]。V0 本无直接到达 V3 的路径，但有 V0→V2→V3 的路径，为 10+G.arcs[2][3].adj=10+50=60，故改写 D[3]=60，如图 7－69(b)所示。用这样的方法，依次将 V4、V3 和 V5 并入 S。详见图 7－69(c)、(d)和(e)。

通过 final[]和 D[]可求得给定点 v0 到图 G 中其余各顶点的最短距离是多少。但却不知道其间通过哪些顶点。矩阵 P[][]有这些顶点的信息。以程序运行结果为例，1 维数组 p[2][]中的 1 是 V0 到 V2 经过的顶点(只有 V0 和 V2 两个顶点)；p[3][]中的 1 是 V0 到 V3 经过的顶点(V0、V3 和 V4)。

7.6.2　每一对顶点之间的最短路径

```
// func7-2.cpp 算法7.16,algo7-7.cpp和algo7-9.cpp用到
void ShortestPath_FLOYD(MGraph G,PathMatrix P,DistancMatrix D)
{ // 用Floyd算法求有向网G中各对顶点v和w之间的最短路径P[v][w]及其带权长度D[v][w]。
  // 若P[v][w][u]为TRUE，则u是从v到w当前求得最短路径上的顶点。算法7.16
  int u,v,w,i;
```

```
  for(v=0;v<G.vexnum;v++) // 各对结点之间初始已知路径及距离
    for(w=0;w<G.vexnum;w++)
    {
      D[v][w]=G.arcs[v][w].adj; // 顶点v到顶点w的直接距离
      for(u=0;u<G.vexnum;u++)
        P[v][w][u]=FALSE; // 路径矩阵初值
      if(D[v][w]<INFINITY) // 从v到w有直接路径
        P[v][w][v]=P[v][w][w]=TRUE; // 由v到w的路径经过v和w两点
    }
  for(u=0;u<G.vexnum;u++)
    for(v=0;v<G.vexnum;v++)
      for(w=0;w<G.vexnum;w++)
        if(D[v][u]<INFINITY&&D[u][w]<INFINITY&&D[v][u]+D[u][w]<D[v][w])
        { // 从v经u到w的一条路径更短
          D[v][w]=D[v][u]+D[u][w]; // 更新最短距离
          for(i=0;i<G.vexnum;i++)
            P[v][w][i]=P[v][u][i]||P[u][w][i]; // 从v到w的路径经过从v到u和从u到w的所有路径
        }
}

// algo7-7.cpp 实现算法7.16的程序
#define MAX_NAME 5 // 顶点字符串的最大长度+1
#define MAX_INFO 20 // 相关信息字符串的最大长度+1
typedef int VRType;
typedef char VertexType[MAX_NAME];
typedef char InfoType;
#include"c1.h"
#include"c7-1.h" // 邻接矩阵存储结构
#include"bo7-1.cpp" // 邻接矩阵存储结构的基本操作
typedef int PathMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM][MAX_VERTEX_NUM]; // 3维数组
typedef int DistancMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM]; // 2维数组
#include"func7-2.cpp" // 求有向网中各对顶点之间最短距离的Floyd算法
void main()
{
  MGraph g;
  int i,j,k,l,m,n;
  PathMatrix p; // 3维数组
  DistancMatrix d; // 2维数组
  CreateDN(g); // 构造有向网g
  for(i=0;i<g.vexnum;i++)
    g.arcs[i][i].adj=0; // ShortestPath_FLOYD()要求对角元素值为0，因为两点相同，其距离为0
  Display(g); // 输出有向网g
  ShortestPath_FLOYD(g,p,d); // 求每对顶点间的最短路径，在func7-2.cpp中
  printf("d矩阵:\n");
  for(i=0;i<g.vexnum;i++)
  {
    for(j=0;j<g.vexnum;j++)
      printf("%6d",d[i][j]);
    printf("\n");
  }
  for(i=0;i<g.vexnum;i++)
```

```
    for(j=0;j<g.vexnum;j++)
      if(i!=j)
        printf("%s到%s的最短距离为%d\n",g.vexs[i],g.vexs[j],d[i][j]);
  printf("p矩阵:\n");
  for(i=0;i<g.vexnum;i++)
    for(j=0;j<g.vexnum;j++)
      if(i!=j)
      {
        printf("由%s到%s经过: ",g.vexs[i],g.vexs[j]);
        for(k=0;k<g.vexnum;k++)
          if(p[i][j][k]==1)
            printf("%s ",g.vexs[k]);
        printf("\n");
      }
}
```

程序运行结果(以教科书中图 7.36 的 G7 为例):

```
请输入有向网G的顶点数,弧数,弧是否含其它信息(是:1,否:0): 3,5,0↙(见图7-70)
请输入3个顶点的值(<5个字符):
A B C↙
请输入5条弧的弧尾 弧头 权值(以空格作为间隔):
A B 4↙
A C 11↙
B A 6↙
B C 2↙
C A 3↙
3个顶点5条边或弧的有向网。顶点依次是: A B C
G.arcs.adj:
          0          4         11
          6          0          2
          3      32767          0
G.arcs.info:
顶点1(弧尾) 顶点2(弧头) 该边或弧的信息:
d矩阵:
     0     4     6
     5     0     2
     3     7     0
A到B的最短距离为4
A到C的最短距离为6
B到A的最短距离为5
B到C的最短距离为2
C到A的最短距离为3
C到B的最短距离为7
p矩阵:
由A到B经过: A B
由A到C经过: A B C
由B到A经过: A B C
由B到C经过: B C
```

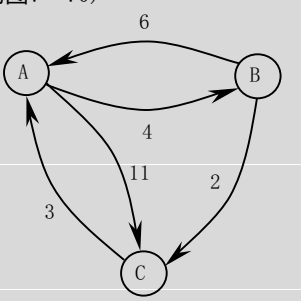


图 7-70　有向网

```
由C到A经过: A C
由C到B经过: A B C
```

求有向网中各对顶点之间最短距离的 Floyd 算法 ShortestPath_FLOYD()要求其网的邻接矩阵中对角元素的权值为 0。因为用邻接矩阵表示各顶点之间的距离，显然，同一点之间的距离为 0。

ShortestPath_FLOYD()算法的思路很简单，首先以 2 顶点之间的直接路径为最短路径，如果能找到第 3 点，使 2 顶点通过第 3 点的路径比直接路径要短，则以这 3 点形成的路径为 2 顶点之间的最短路径。依次再找第 4 点、第 5 点、…… 如以上程序运行结果所示，根据图 7－70 及邻接矩阵，A→C 的直接距离是 11，但 A→B→C 的距离是 6，则以 6 取代 11 作为 A→C 的最短距离。

ShortestPath_FLOYD()不仅可用于有向网，也可用于无向网。因为无向网的 1 条边相当于有向网的 2 条弧。

教科书中图 7.33 是描述中国大陆地区铁路交通的无向网。程序 algo7-9.cpp 利用 ShortestPath_FLOYD()可求得该网中每 2 个站点之间的最短距离。该无向网的数据是利用文件 map.txt 输入的。map.txt 的内容如下(在教科书中图 7.33 的基础上另加孤立顶点台北):

```
26
30
乌鲁木齐
呼和浩特
哈尔滨
西宁
兰州
成都
昆明
贵阳
南宁
柳州
株州
广州
深圳
南昌
福州
上海
武汉
西安
郑州
徐州
北京
天津
沈阳
大连
长春
台北
```

```
乌鲁木齐 兰州 1892
呼和浩特 兰州 1145
呼和浩特 北京 668
哈尔滨 长春 242
西宁 兰州 216
兰州 西安 676
西安 成都 842
西安 郑州 511
成都 昆明 1100
成都 贵阳 967
昆明 贵阳 639
贵阳 柳州 607
柳州 株州 672
柳州 南宁 255
贵阳 株州 902
株州 武汉 409
株州 广州 675
株州 南昌 367
广州 深圳 140
南昌 福州 622
南昌 上海 825
武汉 郑州 534
郑州 北京 695
郑州 徐州 349
徐州 天津 674
徐州 上海 651
北京 天津 137
天津 沈阳 704
沈阳 大连 397
沈阳 长春 305
```

```
// algo7-9.cpp 实现教科书图7.33的程序(另加孤立顶点台北)
#define MAX_NAME 9 // 顶点字符串的最大长度+1
#define MAX_INFO 20 // 相关信息字符串的最大长度+1
typedef int VRType;
typedef char VertexType[MAX_NAME];
typedef char InfoType;
#include"c1.h"
#include"c7-1.h" // 邻接矩阵存储结构
#include"bo7-1.cpp" // 邻接矩阵存储结构的基本操作
typedef int PathMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM][MAX_VERTEX_NUM]; // 3维数组
typedef int DistancMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM]; // 2维数组
#include"func7-2.cpp" // 求有向网中各对顶点之间最短距离的Floyd算法
void path(MGraph G,PathMatrix P,int i,int j)
{ // 求由序号为i的起点城市到序号为j的终点城市最短路径沿途所经过的城市
  int k;
  int m=i; // 起点城市序号赋给m
  printf("依次经过的城市: \n");
  while(m!=j) // 没到终点城市
  {
    G.arcs[m][m].adj=INFINITY; // 对角元素赋值无穷大
```

```
    for(k=0;k<G.vexnum;k++)
      if(G.arcs[m][k].adj<INFINITY&&P[m][j][k]) // m到k有直接通路，且k在m到j的最短路径上
      {
        printf("%s ",G.vexs[m]);
        G.arcs[m][k].adj=G.arcs[k][m].adj=INFINITY; // 将直接通路设为不通
        m=k; // 经过的城市序号赋给m，继续查找
        break;
      }
  }
  printf("%s\n",G.vexs[j]); // 输出终点城市
}
void main()
{
  MGraph g;
  int i,j,k,q=1;
  PathMatrix p; // 3维数组
  DistancMatrix d; // 2维数组
  printf("数据文件名为map.txt\n");
  CreateFUDN(g); // 通过文件构造无向网g
  for(i=0;i<g.vexnum;i++)
    g.arcs[i][i].adj=0; // ShortestPath_FLOYD()要求对角元素值为0，因为两点相同，其距离为0
  ShortestPath_FLOYD(g,p,d); // 求每对顶点间的最短路径，在func7-2.cpp中
  while(q)
  {
    printf("请选择: 1 查询  0 结束\n");
    scanf("%d",&q);
    if(q)
    {
      for(i=0;i<g.vexnum;i++)
      {
        printf("%2d %-9s",i+1,g.vexs[i]);
        if(i%6==5) // 输出6个数据就换行
          printf("\n");
      }
      printf("\n请输入要查询的起点城市代码 终点城市代码: ");
      scanf("%d%d",&i,&j);
      if(d[i-1][j-1]<INFINITY) // 有通路
      {
        printf("%s到%s的最短距离为%d\n",g.vexs[i-1],g.vexs[j-1],d[i-1][j-1]);
        path(g,p,i-1,j-1); // 求最短路径上由起点城市到终点城市沿途所经过的城市
      }
      else
        printf("%s到%s没有路径可通\n",g.vexs[i-1],g.vexs[j-1]);
      printf("与%s到%s有关的p矩阵:\n",g.vexs[i-1],g.vexs[j-1]);
      for(k=0;k<g.vexnum;k++)
        printf("%2d",p[i-1][j-1][k]);
      printf("\n");
    }
  }
}
```

程序运行结果：

```
数据文件名为map.txt
请输入数据文件名：map.txt↙
请选择：1 查询  0 结束
1↙
 1 乌鲁木齐  2 呼和浩特  3 哈尔滨    4 西宁      5 兰州      6 成都
 7 昆明      8 贵阳      9 南宁     10 柳州     11 株州     12 广州
13 深圳     14 南昌     15 福州     16 上海     17 武汉     18 西安
19 郑州     20 徐州     21 北京     22 天津     23 沈阳     24 大连
25 长春     26 台北
请输入要查询的起点城市代码 终点城市代码：1 10↙
乌鲁木齐到柳州的最短距离为4694
依次经过的城市：
乌鲁木齐 兰州 西安 郑州 武汉 株州 柳州
与乌鲁木齐到柳州有关的p矩阵：
 1 0 0 0 1 0 0 0 0 1 1 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0
请选择：1 查询  0 结束
1↙
 1 乌鲁木齐  2 呼和浩特  3 哈尔滨    4 西宁      5 兰州      6 成都
 7 昆明      8 贵阳      9 南宁     10 柳州     11 株州     12 广州
13 深圳     14 南昌     15 福州     16 上海     17 武汉     18 西安
19 郑州     20 徐州     21 北京     22 天津     23 沈阳     24 大连
25 长春     26 台北
请输入要查询的起点城市代码 终点城市代码：21 26↙
北京到台北没有路径可通
与北京到台北有关的p矩阵：
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
请选择：1 查询  0 结束
0↙
```

和 algo7-6.cpp 中的 ShortestPath_DIJ()类似，algo7-9.cpp 中的 ShortestPath_FLOYD()也有存放最短路径通过的顶点的数组 P。在这里，数组 P 是三维的。一维数组 P[v][w][]中的信息是从顶点 v 到顶点 w 最短距离所通过的顶点。如 P[v][w][u]=1，说明从顶点 v 到顶点 w 最短距离通过顶点 u。而 P[v][w][t]=0，说明从顶点 v 到顶点 w 最短距离不通过顶点 t。以上面的程序运行结果为例，D[0][9]是乌鲁木齐到柳州的最短距离，P[0][9][]={1 0 0 0 1 0 0 0 0 1 1 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0}，对照序号可知，乌鲁木齐到柳州的最短距离经过乌鲁木齐、兰州、柳州、株州、武汉、西安和郑州 7 个城市。

为了求得依次经过的城市，调用 path()。path()的算法是：对于从顶点 v 到顶点 w 最短距离路径的起点 v，它的下一点 u 是满足 G.arcs[v][u].adj 不是无穷(v 到 u 有直接通路)同时 P[v][w][u]=1(u 在 v 到 w 的最短路径上)条件的惟一顶点。将 G.arcs[v][u].adj 改为无穷(避免又回头找 v)，再从 u 找满足 G.arcs[u][x].adj 不是无穷(u 到 x 有直接通路)同时 P[v][w][x]=1(x 在 v 到 w 的最短路径上)条件的惟一顶点 x。循环这个过程，直至到达终点 w。以上程序运行结果验证了 path()的作用。

孤立顶点台北和北京不在同一个连通分量中，故它们之间没有路径可通。对应的一维数组 P[20][25][]是全 0。

algo7-9. cpp 虽然能够求得任意两城市间的最短路径，但它的 DOS 界面总让我们感到不方便。我们也可以在可视化的界面下实现 algo7-9. cpp 的功能。

光盘 VC\shortest 子目录中的软件实现了 algo7-9. cpp 的可视化。在 Visual C++6. 0 环境下打开文件 VC\shortest\shortest. dsw，按 F7 编译后，按 Ctrl+F5 运行，就会出现图 7－71 所示界面。

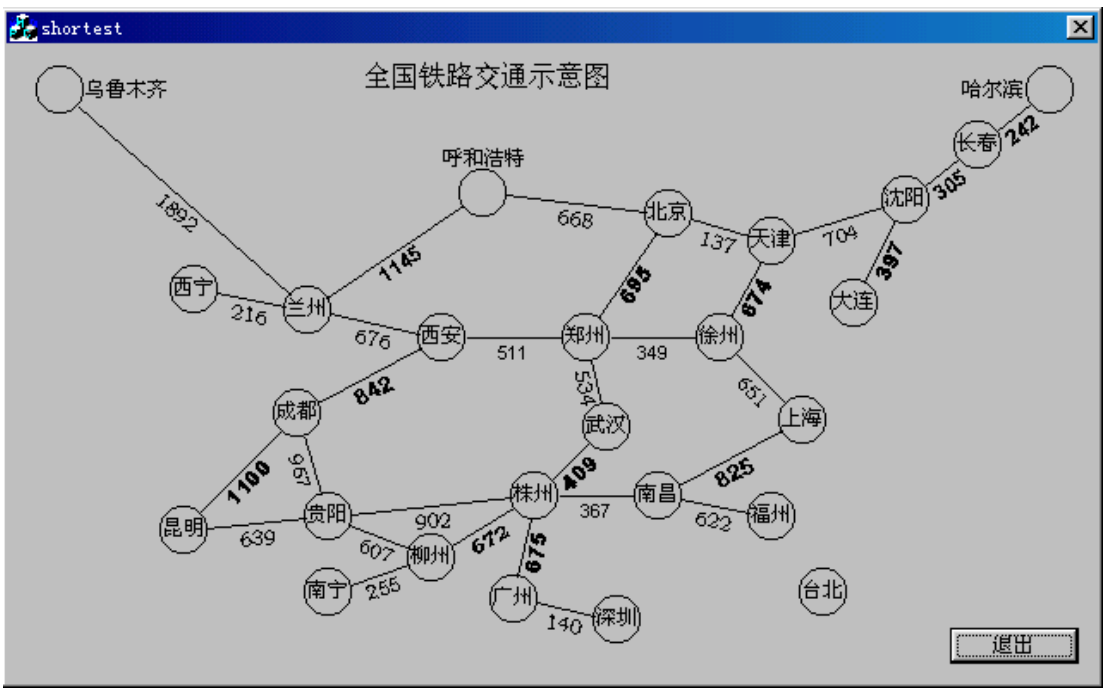


图 7－71　运行 VC\shortest 出现的初始界面

移动鼠标光标到待查询的起点城市圆圈中并单击鼠标左键，即选定了起点城市。该城市的圆圈变成虚线。再移动鼠标光标到待查询的终点城市圆圈中并再次单击鼠标左键，即选定了终点城市。这时，从起点城市到终点城市最短距离的沿途城市圆圈及沿线均变成虚线，显示出求得的最短路径。同时，在图的右部中间还用文字说明两城市间的最短距离及依次经过的城市。

图 7－72 是运行 VC\shortest 子目录中的程序并依次用鼠标左键单击南昌和天津的结果。这个过程可以反复进行，直到按下“退出”按钮。

如果没有 Visual C++6. 0 软件，可将文件 VC\shortest\Debug\shortest. exe 和文件 VC\shortest\mapvc. txt 拷到硬盘的同一个子目录下，直接运行 shortest. exe 即可。

algo7-9. cpp 的许多函数都嵌到 VC\shortest 软件中了。有一些根据具体情况做了修改。如 mapvc. txt 的内容如下：

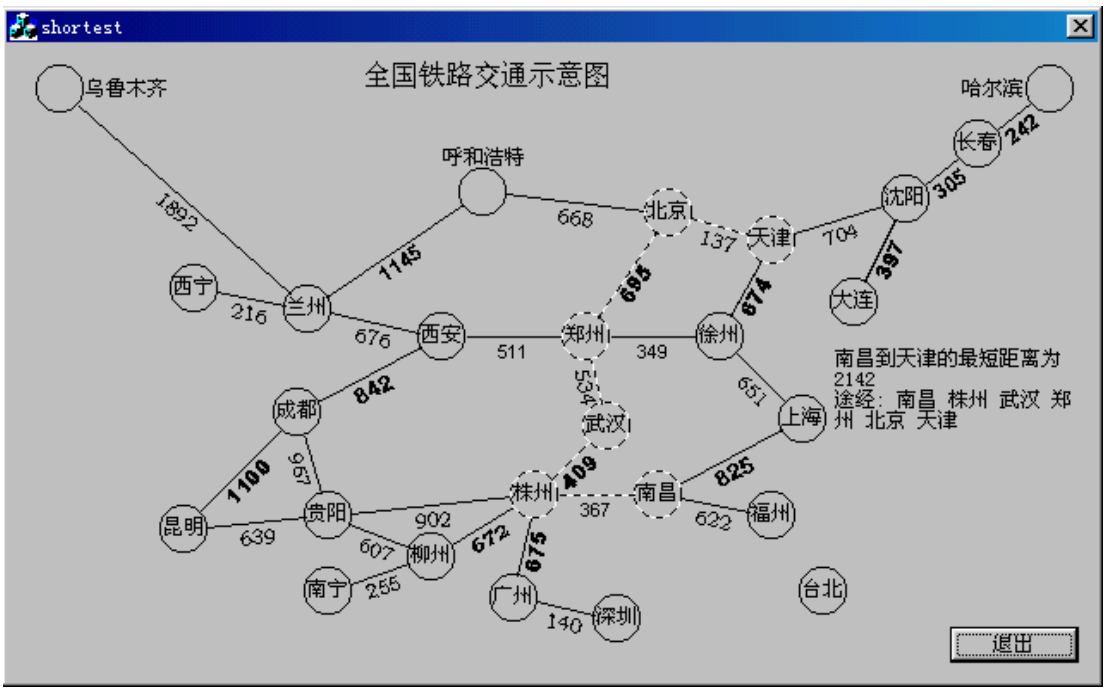


图 7－72　运行 VC\shortest 求得南昌到天津的最短距离

```
26
30
乌鲁木齐 31 26
呼和浩特 277 86
哈尔滨 607 26
西宁 109 142
兰州 175 154
成都 169 214
昆明 103 282
贵阳 187 274
南宁 187 318
柳州 247 298
株州 307 262
广州 295 322
深圳 355 334
南昌 379 262
福州 445 274
上海 463 218
武汉 349 222
西安 253 170
郑州 337 170
徐州 415 170
北京 385 98
天津 445 114
沈阳 523 90
大连 493 150
长春 565 58
```

```
台北 475 318
乌鲁木齐 兰州 1892
呼和浩特 兰州 1145
……(以下同数据文件map.txt，故略)
```

和 algo7-9.cpp 调用的数据文件 map.txt 相比，mapvc.txt 中的顶点信息不仅有城市名称，还有城市在图中的 x、y 坐标。因此，顶点信息用结构体 bb 来表示：

```
struct bb
{
  VertexType a;
  int x;
  int y;
};
```

其中，VertexType 仍然是字符串类型，存放城市名称。同时，由文件构造无向网的函数 CreateFUDN()也做了相应的修改。求有向网中各对顶点之间最短路径的函数 ShortestPath_FLOYD()直接用在了程序中。求由起点城市到终点城市最短路径沿途所经过城市的函数 path()做了修改。

这个软件主要目的在于说明：在视窗时代，算法和数据结构并没有过时，仍然是软件的灵魂。视窗手段能使界面变漂亮，但要想实现众多的复杂功能，还必须依靠算法和数据结构。

第 8 章　动态存储管理

8.1　概述

8.2　可利用空间表

8.3　边界标识法

8.3.1　可利用空间表的结构

```
// c8-1.h 边界标识法可利用空间表的结点结构(见图8－1)
// head和foot分别是可利用空间表中结点的第一个字和最后一个字(WORD)
typedef struct WORD // 字类型
{
  union
  {
    WORD *llink; // 头部域，指向前驱结点
    WORD *uplink; // 底部域，指向本结点头部
  };
  int tag; // 块标志，0：空闲，1：占用，头部和尾部均有
  int size; // 头部域，块大小
  WORD *rlink; // 头部域，指向后继结点
}WORD,head,foot,*Space; // *Space：可利用空间指针类型
#define FootLoc(p) p+p->size-1 // 带参数的宏定义，指向p所指结点的底部(最后一个字)
```

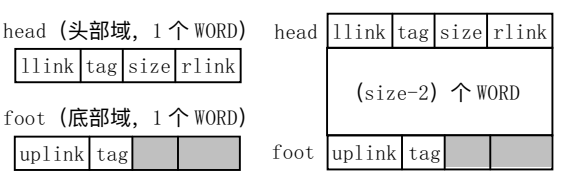
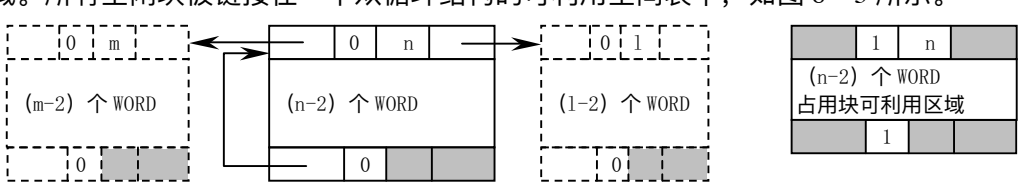


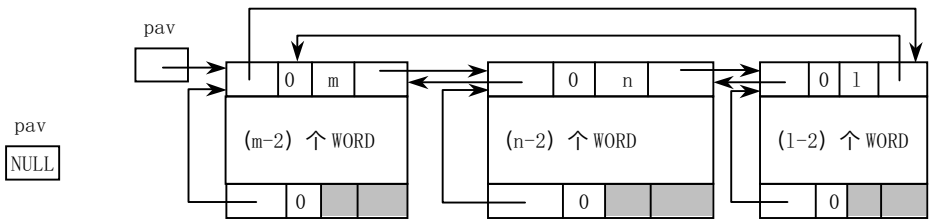
图 8－1　边界标识法可利用空间表的结点结构

图 8－2 是空闲块和占用块的表示。占用块的可利用区域是除头部域和尾部域之外的区域。所有空闲块被链接在一个双循环结构的可利用空间表中，如图 8－3 所示。



(a) 空闲块　　(b) 占用块

图 8－2　空闲块和占用块的表示



(a) 空的可利用空间表　　(b) 有 3 个空闲块的可利用空间表

图 8－3　可利用空间表图示

8.3.2　分配算法

```
// algo8-1.cpp 边界标识法。实现算法8.1的程序
#include"c1.h"
#include"c8-1.h"
#define MAX 1000 // 可利用空间的大小(以WORD的字节数为单位)
#define e 10 // 块的最小尺寸-1(以WORD的字节数为单位)
Space AllocBoundTag(Space &pav,int n) // 算法8.1(首次拟合法)
{ // 若可利用空间表pav中有不小于n的空闲块，则分配相应的存储块，并返回其首地址；否则返回NULL
  // 若分配后可利用空间表不空，则pav指向表中刚分配过的结点的后继结点
  Space p,f;
  for(p=pav;p&&p->size<n&&p->rlink!=pav;p=p->rlink); // 在pav中查找不小于n的空闲块
  if(!p||p->size<n) // 找不到
    return NULL; // 返回空指针
  else // p指向找到的空闲块的头部域
  {
    f=FootLoc(p); // f指向p所指空闲块的底部域
    pav=p->rlink; // 移动pav，使其指向p所指结点的后继结点
    if(p->size-n<=e) // 整块分配，不保留<=e的剩余量，删除该块
    {
      if(pav==p) // 可利用空间表只有1个空闲块
        pav=NULL; // 可利用空间表变为空表
      else // 在表中删除该块
      {
        pav->llink=p->llink;
        p->llink->rlink=pav;
      }
      p->tag=f->tag=1; // 修改分配结点的头部和底部标志为占用
    }
    else // 分配该块的后n个字(高地址部分)，不删除该块(见图8－4)
    {
      f->tag=1; // 修改分配块的底部标志
      p->size-=n; // 置剩余块大小
      f=FootLoc(p); // 指向剩余块底部
      f->tag=0; // 设置剩余块底部
      f->uplink=p;
      p=f+1; // 指向分配块头部
      p->tag=1; // 设置分配块头部
      p->size=n;
    }
```

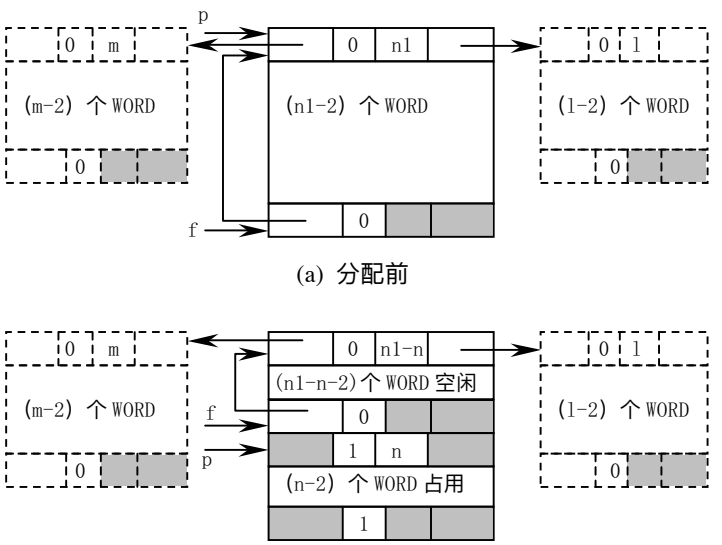
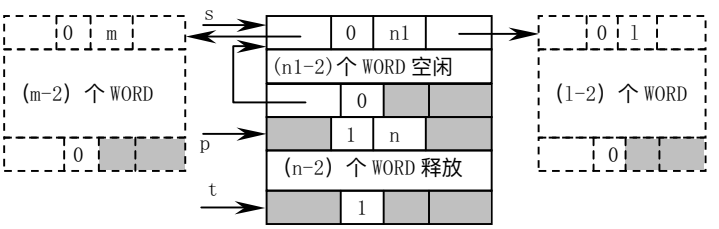
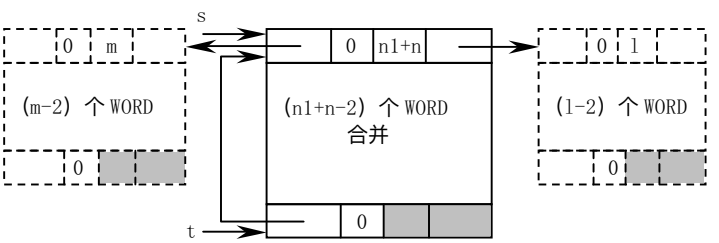


图 8－4　将空闲块的后 n 个 WORD 分配为占用块

```
    return p; // 返回分配块首地址
  }
}
void Reclaim(Space &pav,Space &p)
{ // 边界标识法的回收算法，将p所指的释放块回收到可利用空间表pav中
  Space s,t=p+p->size; // t指向释放块右邻块的首地址
  int l=(p-1)->tag,r=(p+p->size)->tag; // l、r分别指示释放块的左、右邻块是否空闲
  if(!pav) // 可利用空间表空
  { // 将释放块加入到可利用空间表pav中
    pav=p->llink=p->rlink=p; // 头部域的两个指针及pav均指向释放块
    p->tag=0; // 修改头部域块标志为空闲
    (FootLoc(p))->uplink=p; // 修改底部域指针，使其指向释放块的头部域
    (FootLoc(p))->tag=0; // 修改底部域块标志为空闲
  }
  else // 可利用空间表不空
    if(l==1&&r==1) // 左右邻区均为占用块
    { // 将释放块插入到可利用空间表pav中
      p->tag=0; // 修改释放块头部域块标志为空闲
      (FootLoc(p))->uplink=p; // 修改底部域指针，使其指向释放块的头部域
      (FootLoc(p))->tag=0; // 修改底部域块标志为空闲
      pav->llink->rlink=p; // 将p所指结点(刚释放的结点)插在pav所指结点之前
      p->llink=pav->llink;
      p->rlink=pav;
      pav->llink=p;
      pav=p; // 修改pav，令刚释放的结点为下次分配时的最先查询的结点
    }
    else if(l==0&&r==1) // 左邻区为空闲块，右邻区为占用块(见图8－5)
    { // 合并左邻块和释放块(将释放块“粘到”左邻块的下边，不改变可利用空间表pav)
      s=(p-1)->uplink; // s为左邻空闲块的头部域地址
      s->size+=p->size; // 设置合并的空闲块大小
```



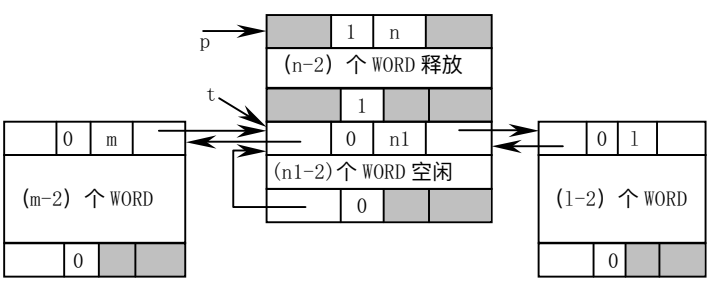
(a) 合并前



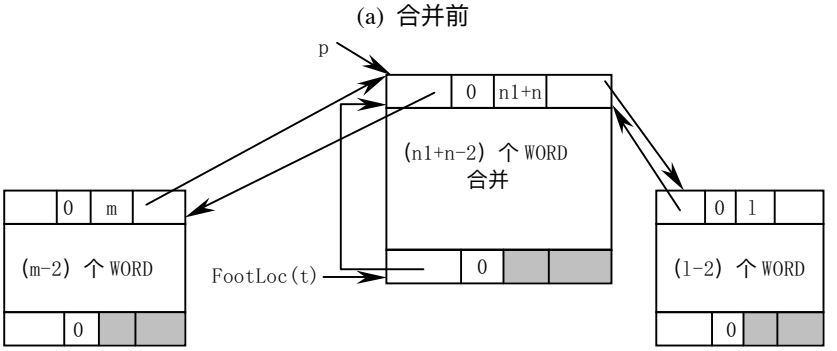
(b) 合并后

图 8－5　将释放块和左邻空闲块合并的图示

```
  t=FootLoc(p); // t指向合并的空闲块底部域(释放块的底部域)
  t->uplink=s; // 设置合并的空闲块底部域指针，使其指向合并的空闲块的头部域
  t->tag=0; // 设置合并的空闲块底部域块标志为空闲
}
else if(l==1&&r==0) // 右邻区为空闲块，左邻区为占用块(见图8－6)
{ // 合并右邻块和释放块，t为右邻空闲块的头部域地址，用合并块取代右邻空闲块在pav中的位置
  p->tag=0; // P为合并后的结点头部域地址，设置其块标志为空闲
```



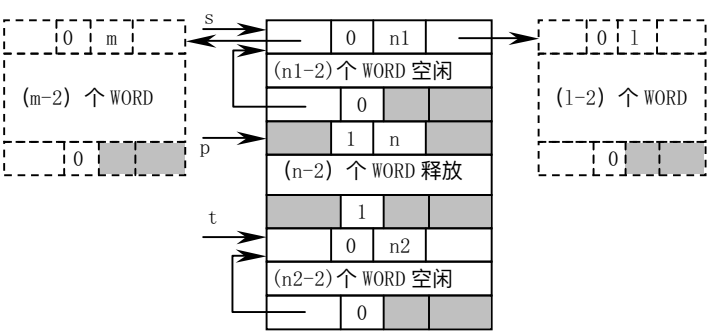
(a) 合并前



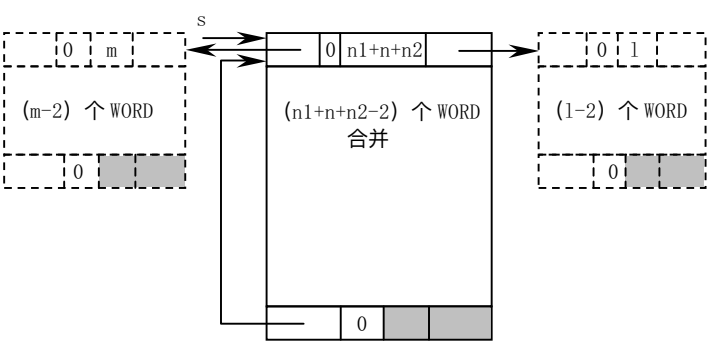
(b) 合并后

图 8－6　将释放块和右邻空闲块合并的图示

```
      p->llink=t->llink; // p的前驱为原t的前驱
      p->llink->rlink=p; // p的前驱的后继指向p
      p->rlink=t->rlink; // p的后继为原t的后继
      p->rlink->llink=p; // p的后继的前驱指向p
      p->size+=t->size; // 新的合并块的大小
      (FootLoc(t))->uplink=p; // 底部域(原t的底部域)指针指向新的合并块的头部域
      if(pav==t) // 可利用空间表的头指针指向t(t已不是空闲结点首地址了)
        pav=p; // 修改pav，令刚释放的结点为下次分配时的最先查询的结点
    }
    else // 左右邻区均为空闲块(见图8-7)
    { // 合并左右邻块和释放块，t为右邻空闲块的头部域地址
      t->llink->rlink=t->rlink; // 在pav中删去右邻空闲块结点
      t->rlink->llink=t->llink;
      s=(p-1)->uplink; // s为左邻空闲块的头部域地址，也是新的合并块的头部域地址
      s->size+=p->size+t->size; // 设置新结点的大小(3块之和)
      (FootLoc(t))->uplink=s; // 新结点底部(原t的底部)指针指向其头部
      if(pav==t) // 可利用空间表的头指针指向t(t已不是空闲结点首地址了)
        pav=s; // 修改pav，令刚释放的结点为下次分配时的最先查询的结点
    }
  p=NULL; // 令刚释放的结点的指针为空
}
```



(a) 合并前(先将右邻块 t 从 pav 中删去)



(b) 合并后

图 8-7　将释放块和左右邻空闲块合并的图示

```
void Print(Space p)
{ // 输出p所指的可利用空间表
  Space h,f;
  if(p) // 可利用空间表不空
```

```
  {
    h=p; // h指向第一个结点的头部域(首地址)
    f=FootLoc(h); // f指向第一个结点的底部域
    do
    {
      printf("块的大小=%d 块的首地址=%u ",h->size,f->uplink); // 输出结点信息
      printf("块标志=%d(0:空闲 1:占用) 邻块首地址=%u\n",h->tag,f+1);
      h=h->rlink; // 指向下一个结点的头部域(首地址)
      f=FootLoc(h); // f指向下一个结点的底部域
    }while(h!=p); // 没到循环链表的表尾
  }
}
void PrintUser(Space p[])
{ // 输出p数组所指的已分配空间
  for(int i=0;i<MAX/e;i++)
    if(p[i]) // 指针不为0(指向一个占用块)
    {
      printf("块%d的首地址=%u ",i,p[i]); // 输出结点信息
      printf("块的大小=%d 块头标志=%d(0:空闲 1:占用)",p[i]->size,p[i]->tag);
      printf(" 块尾标志=%d\n",(FootLoc(p[i]))->tag);
    }
}
void main()
{
  Space pav,p; // 空闲块指针
  Space v[MAX/e]={NULL}; // 占用块指针数组(初始化为空)
  int n;
  printf("结构体WORD为%d个字节\n",sizeof(WORD));
  p=new WORD[MAX+2]; // 申请大小为MAX*sizeof(WORD)个字节的空间(见图8-8)
  p->tag=1; // 设置低址边界，以防查找左邻块时出错
  pav=p+1; // 可利用空间表的表头
  pav->rlink=pav->llink=pav; // 初始化可利用空间(一个整块)
  pav->tag=0;
  pav->size=MAX;
  p=FootLoc(pav); // p指向底部域
  p->uplink=pav;
  p->tag=0;
  (p+1)->tag=1; // 设置高址边界，以防查找右邻块时出错
  printf("初始化后，可利用空间表为\n");
  Print(pav);
  n=300;
  v[0]=AllocBoundTag(pav,n);
  printf("分配%u个存储空间后，可利用空间表为\n",n);
  Print(pav);
  PrintUser(v);
  n=450;
  v[1]=AllocBoundTag(pav,n);
  printf("分配%u个存储空间后，pav为\n",n);
  Print(pav);
  PrintUser(v);
  n=300; // 分配不成功
```

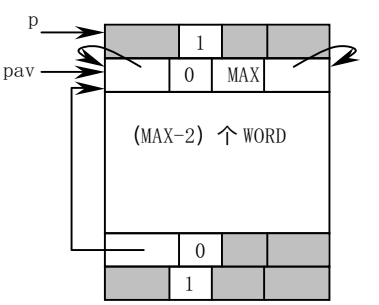


图 8-8　初始化可利用空间

```
  v[2]=AllocBoundTag(pav,n);
  printf("分配%u个存储空间后(不成功), pav为\n",n);
  Print(pav);
  PrintUser(v);
  n=242; // 分配整个块(250)
  v[2]=AllocBoundTag(pav,n);
  printf("分配%u个存储空间后(整块分配), pav为\n",n);
  Print(pav);
  PrintUser(v);
  printf("回收v[0](%d)后(当pav空时回收), pav为\n",v[0]->size);
  Reclaim(pav,v[0]); // pav为空
  Print(pav);
  PrintUser(v);
  printf("1按回车键继续");
  getchar();
  printf("回收v[2](%d)后(左右邻区均为占用块),pav为\n",v[2]->size);
  Reclaim(pav,v[2]); // 左右邻区均为占用块
  Print(pav);
  PrintUser(v);
  n=270; // 查找空间足够大的块
  v[0]=AllocBoundTag(pav,n);
  printf("分配%u个存储空间后(查找空间足够大的块), pav为\n",n);
  Print(pav);
  PrintUser(v);
  n=30; // 在当前块上分配
  v[2]=AllocBoundTag(pav,n);
  printf("分配%u个存储空间后(在当前块上分配), pav为\n",n);
  Print(pav);
  PrintUser(v);
  printf("回收v[1](%d)后(右邻区为空闲块,左邻区为占用块), pav为\n",v[1]->size);
  Reclaim(pav,v[1]); // 右邻区为空闲块, 左邻区为占用块
  Print(pav);
  PrintUser(v);
  printf("2按回车键继续");
  getchar();
  printf("回收v[0](%d)后(左邻区为空闲块,右邻区为占用块), pav为\n",v[0]->size);
  Reclaim(pav,v[0]); // 左邻区为空闲块, 右邻区为占用块
  Print(pav);
  PrintUser(v);
  printf("回收v[2](%d)后(左右邻区均为空闲块),pav为\n",v[2]->size);
  Reclaim(pav,v[2]); // 左右邻区均为空闲块
  Print(pav);
  PrintUser(v);
}
```

程序运行结果(见图 8－9)：

```
结构体WORD为8个字节
初始化后, 可利用空间表为(见图8－9(a))
```

```
块的大小=1000 块的首地址=2812 块标志=0(0:空闲 1:占用) 邻块首地址=10812
分配300个存储空间后，可利用空间表为(见图8－9(b))
块的大小=700 块的首地址=2812 块标志=0(0:空闲 1:占用) 邻块首地址=8412
块0的首地址=8412 块的大小=300 块头标志=1(0:空闲 1:占用) 块尾标志=1
分配450个存储空间后，pav为(见图8－9(c))
块的大小=250 块的首地址=2812 块标志=0(0:空闲 1:占用) 邻块首地址=4812
块0的首地址=8412 块的大小=300 块头标志=1(0:空闲 1:占用) 块尾标志=1
块1的首地址=4812 块的大小=450 块头标志=1(0:空闲 1:占用) 块尾标志=1
分配300个存储空间后(不成功)，pav为(见图8－9(c))
块的大小=250 块的首地址=2812 块标志=0(0:空闲 1:占用) 邻块首地址=4812
块0的首地址=8412 块的大小=300 块头标志=1(0:空闲 1:占用) 块尾标志=1
块1的首地址=4812 块的大小=450 块头标志=1(0:空闲 1:占用) 块尾标志=1
分配242个存储空间后(整块分配)，pav为(见图8－9(d))
块0的首地址=8412 块的大小=300 块头标志=1(0:空闲 1:占用) 块尾标志=1
块1的首地址=4812 块的大小=450 块头标志=1(0:空闲 1:占用) 块尾标志=1
块2的首地址=2812 块的大小=250 块头标志=1(0:空闲 1:占用) 块尾标志=1
回收v[0](300)后(当pav空时回收)，pav为(见图8－9(e))
块的大小=300 块的首地址=8412 块标志=0(0:空闲 1:占用) 邻块首地址=10812
块1的首地址=4812 块的大小=450 块头标志=1(0:空闲 1:占用) 块尾标志=1
块2的首地址=2812 块的大小=250 块头标志=1(0:空闲 1:占用) 块尾标志=1
1按回车键继续↙
回收v[2](250)后(左右邻区均为占用块)，pav为(见图8－9(f))
块的大小=250 块的首地址=2812 块标志=0(0:空闲 1:占用) 邻块首地址=4812
块的大小=300 块的首地址=8412 块标志=0(0:空闲 1:占用) 邻块首地址=10812
块1的首地址=4812 块的大小=450 块头标志=1(0:空闲 1:占用) 块尾标志=1
分配270个存储空间后(查找空间足够大的块)，pav为(见图8－9(g))
块的大小=250 块的首地址=2812 块标志=0(0:空闲 1:占用) 邻块首地址=4812
块的大小=30 块的首地址=8412 块标志=0(0:空闲 1:占用) 邻块首地址=8652
块0的首地址=8652 块的大小=270 块头标志=1(0:空闲 1:占用) 块尾标志=1
块1的首地址=4812 块的大小=450 块头标志=1(0:空闲 1:占用) 块尾标志=1
分配30个存储空间后(在当前块上分配)，pav为(见图8－9(h))
块的大小=30 块的首地址=8412 块标志=0(0:空闲 1:占用) 邻块首地址=8652
块的大小=220 块的首地址=2812 块标志=0(0:空闲 1:占用) 邻块首地址=4572
块0的首地址=8652 块的大小=270 块头标志=1(0:空闲 1:占用) 块尾标志=1
块1的首地址=4812 块的大小=450 块头标志=1(0:空闲 1:占用) 块尾标志=1
块2的首地址=4572 块的大小=30 块头标志=1(0:空闲 1:占用) 块尾标志=1
回收v[1](450)后(右邻区为空闲块,左邻区为占用块)，pav为(见图8－9(i))
块的大小=480 块的首地址=4812 块标志=0(0:空闲 1:占用) 邻块首地址=8652
块的大小=220 块的首地址=2812 块标志=0(0:空闲 1:占用) 邻块首地址=4572
块0的首地址=8652 块的大小=270 块头标志=1(0:空闲 1:占用) 块尾标志=1
块2的首地址=4572 块的大小=30 块头标志=1(0:空闲 1:占用) 块尾标志=1
2按回车键继续↙
回收v[0](270)后(左邻区为空闲块,右邻区为占用块)，pav为(见图8－9(j))
块的大小=750 块的首地址=4812 块标志=0(0:空闲 1:占用) 邻块首地址=10812
块的大小=220 块的首地址=2812 块标志=0(0:空闲 1:占用) 邻块首地址=4572
块2的首地址=4572 块的大小=30 块头标志=1(0:空闲 1:占用) 块尾标志=1
回收v[2](30)后(左右邻区均为空闲块)，pav为(见图8－9(a))
块的大小=1000 块的首地址=2812 块标志=0(0:空闲 1:占用) 邻块首地址=10812
```

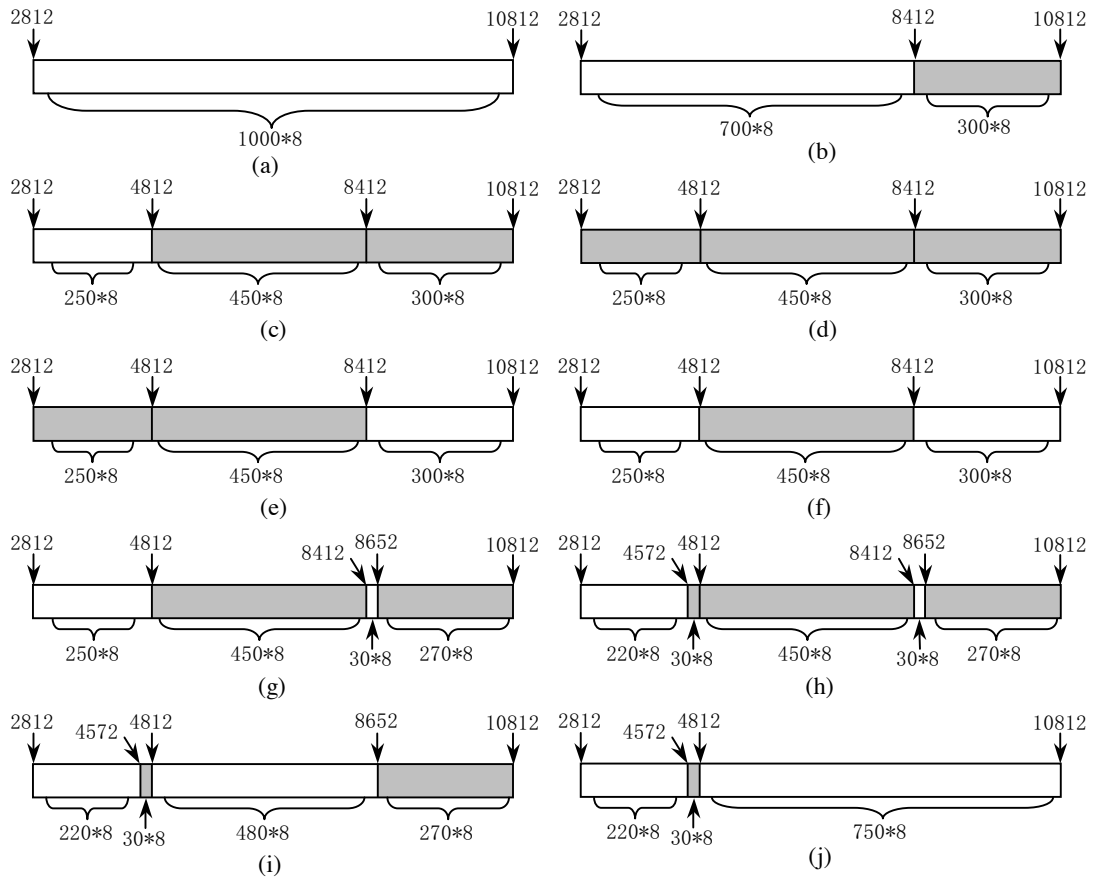


图 8－9　运行 algo8-1. cpp 动态存储区的变化图示(阴影为占用块)

边界标识法对动态分配块的大小不做限制，在块的两头各设一个头部域和底部域，用以标识该块是否被占用。设指针 p 指向动态分配块的头部域，则 p-1 指向其左邻块(低地址紧邻区)的底部域，其底部域的 tag 成员指示该左邻块是否被占用。如果没被占用，其底部域的 uplink 成员还指向该左邻块的头部域；p+p->size 指向其右邻块(高地址紧邻区)的头部域，其头部域的 tag 成员指示该右邻块是否被占用。根据左右邻块被占用的情况，决定该动态分配块是否与其邻块合并。

8.3.3　回收算法

回收算法在 algo8-1. cpp 中。

8.4　伙伴系统

8.4.1　可利用空间表的结构

```
// c8-2.h 伙伴系统可利用空间表的结构(见图8－10)
#define m 10 // 可利用空间总容量1024字的2的幂次，子表的个数为m+1
typedef struct WORD_b
```

```
{
  WORD_b *llink; // 指向前驱结点
  int tag; // 块标志，0：空闲，1：占用
  int kval; // 块大小，值为2的幂次k
  WORD_b *rlink; // 头部域，指向后继结点
}WORD_b,head,*Space; // WORD_b:内存字类型，结点的第一个字也称为head
typedef struct HeadNode
{
  int nodesize; // 该链表的空闲块的大小
  WORD_b *first; // 该链表的表头指针
}FreeList[m+1]; // 表头向量类型(见图8－11)
```

图 8－10　伙伴系统可利用空间表的结点结构

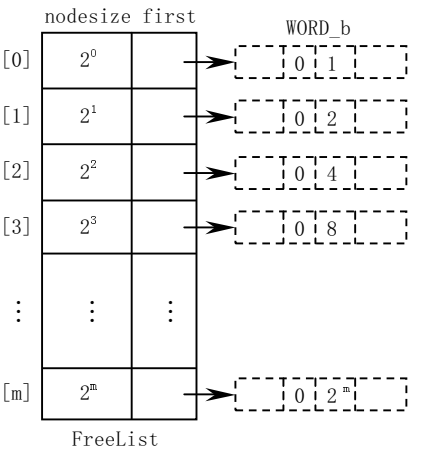


图 8－11　伙伴系统可利用空间表的表头结构

8.4.2　分配算法

```
// algo8-2.cpp 伙伴系统。实现算法8.2的程序
#include"c1.h"
#include"c8-2.h"
#define N 100 // 占用块个数的最大值
Space r; // r为生成空间的首地址，全局量
Space AllocBuddy(FreeList avail,int n)
{ // avail[0..m]为可利用空间表，n为申请分配量，
  // 若有不小于n的空闲块，则分配相应的存储块，
  // 并返回其首地址；否则返回NULL。算法8.2
  int i,k;
  Space pa,pi,pre,suc;
  for(k=0;k<=m&&(avail[k].nodesize<n+1||!avail[k].first);++k);
  // 从小到大查找满足分配要求的子表序号k
  if(k>m) // 分配失败，返回NULL
    return NULL;
  else // 进行分配
  {
    pa=avail[k].first; // pa指向可分配子表的第一个结点
    pre=pa->llink; // pre和suc分别指向pa所指结点的前驱和后继
    suc=pa->rlink;
    if(pa==suc) // 可分配子表只有1个结点
```

```
      avail[k].first=NULL; // 分配后该子表变成空表
    else // 从子表中删去pa所指结点(链表的第1个结点)
    {
      pre->rlink=suc;
      suc->llink=pre;
      avail[k].first=suc; // 该子表的头指针指向pa所指结点的后继
    }
    for(i=1;avail[k-i].nodesize>=n+1;++i)
    { // 从大到小将剩余块插入相应子表，约定将低地址(最前面)的块作为分配块
      pi=pa+int(pow(2,k-i)); // pi指向再分割的后半块(剩余块)
      pi->rlink=pi; // pi是该链表的第1个结点，故左右指针都指向自身
      pi->llink=pi;
      pi->tag=0; // 块标志为空闲
      pi->kval=k-i; // 块大小
      avail[k-i].first=pi; // 插入链表
    }
    pa->tag=1; // 最后剩给pa的是分配块，令其块标志为占用
    pa->kval=k-(--i); // 块大小
  }
  return pa; // 返回分配块的地址
}
Space buddy(Space p)
{ // 返回起始地址为p，块大小为pow(2,p->kval)的块的伙伴地址
  if((p-r)%int(pow(2,p->kval+1))==0) // p为前块
    return p+int(pow(2,p->kval)); // 返回后块地址
  else // p为后块
    return p-int(pow(2,p->kval)); // 返回前块地址
}
void Reclaim(FreeList pav,Space &p)
{ // 伙伴系统的回收算法。将p所指的释放块回收到可利用空间表pav中
  Space s;
  s=buddy(p); // 伙伴块的起始地址
  while(s>=r&&s<r+pav[m].nodesize&&s->tag==0&&s->kval==p->kval) // 归并伙伴块
  { // 伙伴块起始地址在有效范围内且伙伴块空闲并与p块等大，从链表上删除该伙伴块结点
    if(s->rlink==s) // 链表上仅此一个结点
      pav[s->kval].first=NULL; // 置此链表为空
    else // 链表上不止一个结点
    {
      s->llink->rlink=s->rlink; // 前驱的后继为该结点的后继
      s->rlink->llink=s->llink; // 后继的前驱为该结点的前驱
      if(pav[s->kval].first==s) // s是链表的第一个结点
        pav[s->kval].first=s->rlink; // 修改表头指向下一个结点
    } // 以下修改结点头部
    if((p-r)%int(pow(2,p->kval+1))==0) // p为前块
      p->kval++; // 块大小加倍
    else // p为后块(s为前块)
    {
      s->kval=p->kval+1; // 块大小加倍
      p=s; // p指向新块首地址
    }
```

```
    s=buddy(p); // 下一个伙伴块的起始地址
  } // 以下将p插到可利用空间表中
  p->tag=0; // 设块标志为空闲
  if(pav[p->kval].first==NULL) // 该链表空
    pav[p->kval].first=p->llink=p->rlink=p; // 左右指针及表头都指向自身
  else // 该链表不空，插在表头
  {
    p->rlink=pav[p->kval].first;
    p->llink=p->rlink->llink;
    p->rlink->llink=p;
    p->llink->rlink=p;
    pav[p->kval].first=p;
  }
  p=NULL;
}
void Print(FreeList p)
{ // 输出p中所有可利用空间表
  int i;
  Space h;
  for(i=0;i<=m;i++)
    if(p[i].first) // 第i个可利用空间表不空
    {
      h=p[i].first; // h指向链表的第一个结点的头部域(首地址)
      do
      {
        printf("块的大小=%d 块的首地址=%u ",int(pow(2,h->kval)),h); // 输出结点信息
        printf("块标志=%d(0:空闲 1:占用)\n",h->tag);
        h=h->rlink; // 指向下一个结点的头部域(首地址)
      }while(h!=p[i].first); // 没到循环链表的表尾
    }
}
void PrintUser(Space p[])
{ // 输出p数组所指的已分配空间
  for(int i=0;i<N;i++)
    if(p[i]) // 指针不为0(指向一个占用块)
    {
      printf("占用块%d的首地址=%u ",i,p[i]); // 输出结点信息
      printf("块的大小=%d",int(pow(2,p[i]->kval)));
      printf(" 块标志=%d(0:空闲 1:占用)\n",p[i]->tag);
    }
}
void main()
{
  int i,n;
  FreeList a;
  Space q[N]={NULL}; // q数组为占用块的首地址
  printf("sizeof(WORD_b)=%u m=%u int(pow(2,m))=%u\n",sizeof(WORD_b),m,int(pow(2,m)));
  for(i=0;i<=m;i++) // 初始化a(见图8－12)
  {
```

```
    a[i].nodesize=int(pow(2,i));
    a[i].first=NULL;
  }
  r=a[m].first=new WORD_b[a[m].nodesize];
  // 在最大链表中生成一个结点
  if(r) // 生成结点成功
  {
    r->llink=r->rlink=r; // 初始化该结点
    r->tag=0;
    r->kval=m;
    Print(a);
    n=100;
    q[0]=AllocBuddy(a,n);
    // 向a申请100个WORD_b的内存(实际获得128个WORD_b)
    printf("申请%d个字后，可利用空间为\n",n);
    Print(a);
    PrintUser(q);
    n=200;
    q[1]=AllocBuddy(a,n); // 向a申请200个WORD_b的内存(实际获得256个WORD_b)
    printf("申请%d个字后，可利用空间为\n",n);
    Print(a);
    PrintUser(q);
    n=220;
    q[2]=AllocBuddy(a,n); // 向a申请220个WORD_b的内存(实际获得256个WORD_b)
    printf("申请%d个字后，可利用空间为\n",n);
    Print(a);
    PrintUser(q);
    Reclaim(a,q[1]); // 回收q[1]，伙伴不空闲
    printf("回收q[1]后，可利用空间为\n");
    Print(a);
    PrintUser(q);
    Reclaim(a,q[0]); // 回收q[0]，伙伴空闲
    printf("回收q[0]后，可利用空间为\n");
    Print(a);
    PrintUser(q);
    Reclaim(a,q[2]); // 回收q[2]，伙伴空闲，生成一个大结点
    printf("回收q[2]后，可利用空间为\n");
    Print(a);
    PrintUser(q);
  }
  else
    printf("ERROR\n");
}
```

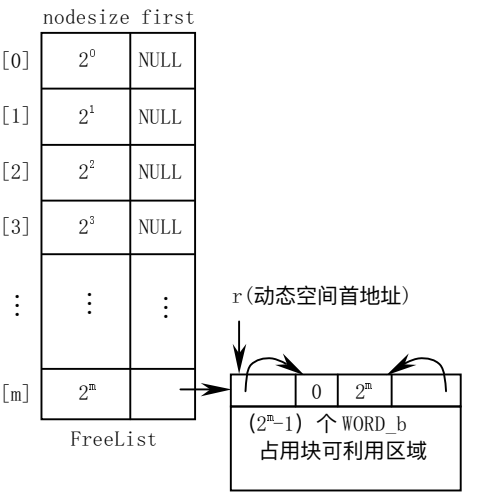
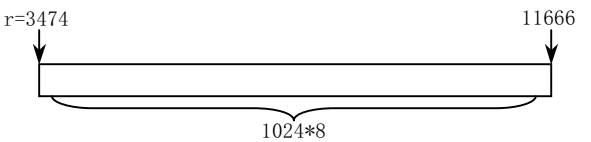


图 8 - 12　可利用空间表的初态

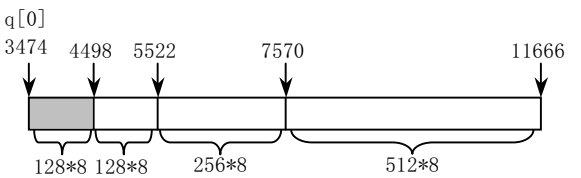
程序运行结果(见图 8 - 13)：

```
sizeof(WORD_b)=8 m=10 int(pow(2,m))=1024
块的大小=1024 块的首地址=3474 块标志=0(0:空闲 1:占用)(见图8 - 13(a))
申请100个字后，可利用空间为(见图8 - 13(b))
```

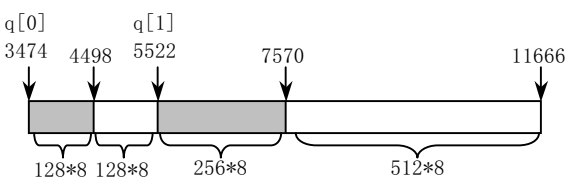
```
块的大小=128 块的首地址=4498 块标志=0(0:空闲 1:占用)
块的大小=256 块的首地址=5522 块标志=0(0:空闲 1:占用)
块的大小=512 块的首地址=7570 块标志=0(0:空闲 1:占用)
占用块0的首地址=3474 块的大小=128 块标志=1(0:空闲 1:占用)
申请200个字后，可利用空间为(见图8－13(c))
块的大小=128 块的首地址=4498 块标志=0(0:空闲 1:占用)
块的大小=512 块的首地址=7570 块标志=0(0:空闲 1:占用)
占用块0的首地址=3474 块的大小=128 块标志=1(0:空闲 1:占用)
占用块1的首地址=5522 块的大小=256 块标志=1(0:空闲 1:占用)
```



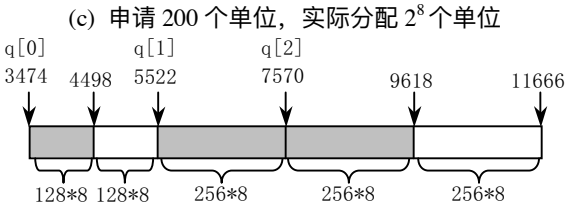
(a) 动态生成内存管理空间，r 为首地址，大小为 2^m 个 WORD_b



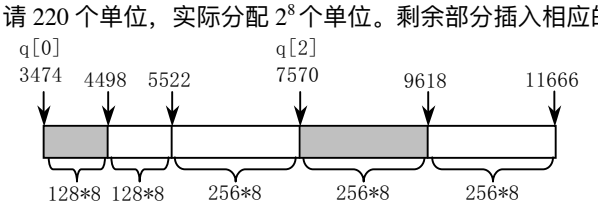
(b) 申请 100 个单位，实际分配低地址的 2^7 个单位。剩余部分分为几块插入各自的表中



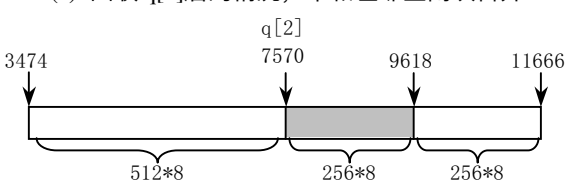
(c) 申请 200 个单位，实际分配 2^8 个单位



(d) 申请 220 个单位，实际分配 2^8 个单位。剩余部分插入相应的表中



(e) 回收 q[1]后的情况，不和左邻空闲块合并



(f) 回收 q[0]后的情况，与空闲的右邻块合并之后再合并

图 8－13　运行 algo8-2. cpp 内存区的变化图示(阴影为占用块)

```
申请220个字后，可利用空间为(见图8－13(d))
块的大小=128 块的首地址=4498 块标志=0(0:空闲 1:占用)
块的大小=256 块的首地址=9618 块标志=0(0:空闲 1:占用)
占用块0的首地址=3474 块的大小=128 块标志=1(0:空闲 1:占用)
占用块1的首地址=5522 块的大小=256 块标志=1(0:空闲 1:占用)
占用块2的首地址=7570 块的大小=256 块标志=1(0:空闲 1:占用)
回收q[1]后，可利用空间为(见图8－13(e))
块的大小=128 块的首地址=4498 块标志=0(0:空闲 1:占用)
块的大小=256 块的首地址=5522 块标志=0(0:空闲 1:占用)
块的大小=256 块的首地址=9618 块标志=0(0:空闲 1:占用)
占用块0的首地址=3474 块的大小=128 块标志=1(0:空闲 1:占用)
占用块2的首地址=7570 块的大小=256 块标志=1(0:空闲 1:占用)
回收q[0]后，可利用空间为(见图8－13(f))
块的大小=256 块的首地址=9618 块标志=0(0:空闲 1:占用)
块的大小=512 块的首地址=3474 块标志=0(0:空闲 1:占用)
占用块2的首地址=7570 块的大小=256 块标志=1(0:空闲 1:占用)
回收q[2]后，可利用空间为(见图8－13(a))
块的大小=1024 块的首地址=3474 块标志=0(0:空闲 1:占用)
```

8.4.3　回收算法

回收算法在 algo8-2.cpp 中。伙伴系统的动态分配块只设头部域，因为它的大小不是任意的。通过它的头部域信息，就可以找到其左右邻块，并可知道其左右邻块是否空闲，所以不需要设底部域。伙伴系统在回收释放块时，也要合并它的左右邻块。和边界标识法不同的是，在回收释放块时，其左右邻块即使是空闲块，也不一定合并，还要看它们是否为伙伴。如图 8－14 所示，E 块和 D 块是伙伴，E 块和 F 块就不是伙伴。两相邻块是伙伴

图 8－14　伙伴关系图示

的条件有两个：(1) 它们的块大小相等；(2) 由整个可利用内存区的起点 r 到合并块之前的空间大小是这个合并块的整数倍。如 r 到 q 的空间和 D、E 合并后的空间大小相等，且 D、E 两块的大小相等，所以 D、E 是伙伴块，可以合并。而 r 到 p 的空间是 E、F 合并后的空间大小的 1.5 倍，虽然 E、F 两块的大小相等且相邻，但不是伙伴块，不能合并。伙伴块分前块和后块。从 r 到前块首地址的空间大小是前块大小的偶数倍；从 r 到后块首地址的空间大小是后块大小的奇数倍。如 r 到 q 的空间是 D 的空间大小的 2 倍，故 D 是前块；而 r 到 p 的空间是 E 的空间大小的 3 倍，故 E 是后块。知道某块是前块还是后块，又知道它的大小，就能求得它的伙伴块的地址。如果它是前块，则它的伙伴块的地址是它的地址加上它的大小，如 D 是前块，它的伙伴块 E 的地址 p 是 q+q->size；如果它是后块，则它的伙伴块的地址是它的地址减去它的大小，如 E 是后块，它的伙伴块 D 的地址 q 是 p-p->size。algo8-2.cpp 中的函数 buddy()就是根据块地址判断其是前块还是后块，并返回其伙伴块地址的函数。有了伙伴块地址，还要看其伙伴块的大小是否与其大小相等。如根据函数 buddy()判断，F 是前块，其伙伴块地址是 t，但 G 并不是它的伙伴块，因为 G 的

大小与 F 不同。只有当 G 与 H 合并之后，其合并块才是 F 的伙伴块。

8.5　无用单元收集

```
// c8-3.h 加标志域的广义表的头尾链表存储结构(由c5-5.h改)(见图8－15)
enum ElemTag{ATOM,LIST}; // ATOM==0: 原子, LIST==1: 子表
typedef struct GLNode
{
  int mark; // 加此域, 其余同c5-5.h
  ElemTag tag;
        // 公共部分, 用于区分原子结点和表结点
  union // 原子结点和表结点的联合部分
  {
    AtomType atom;
          // atom是原子结点的值域, AtomType由用户定义
    struct
    {
      GLNode *hp,*tp;
    }ptr; // ptr是表结点的指针域, prt.hp和ptr.tp分别指向表头和表尾
  };
}*GList,GLNode; // 广义表类型
```

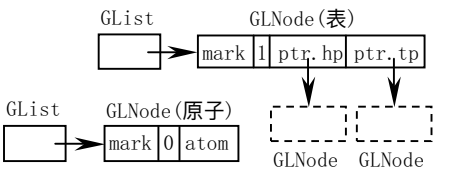


图 8－15　加标志域的广义表的头尾链表存储结构

```
// algo8-3.cpp 实现算法8.3的程序
#include"c1.h"
typedef char AtomType; // 定义原子类型为字符型
#include"c8-3.h"
#include"bo5-5.cpp"
void MarkList(GList GL) // 算法8.3改
{ // 遍历非空广义表GL(GL!=NULL且GL->mark不定), 对表中所有未加标志的结点加标志
  GList q,p=GL,t=NULL; // t指示p的母表
  Status finished=FALSE;
  if(GL==NULL)
    return;
  while(!finished)
  {
    while(p->mark!=1)
    {
      p->mark=1; // MarkHead(p)的细化
      q=p->ptr.hp; // q指向*p的表头
      if(q&&q->mark!=1)
        if(q->tag==0)
          q->mark=1; // ATOM, 表头为原子结点
        else
        { // 继续遍历子表
          p->ptr.hp=t;
          p->tag=ATOM;
          t=p;
          p=q;
        }
    } // 完成对表头的标志
```

```
    q=p->ptr.tp; // q指向*p的表尾
    if(q&&q->mark!=1)
    { // 继续遍历表尾
      p->ptr.tp=t;
      t=p;
      p=q;
    }
    else // BackTrack(finished)的细化
    {
      while(t&&t->tag==1) // LIST, 表结点, 从表尾回溯
      {
        q=t;
        t=q->ptr.tp;
        q->ptr.tp=p;
        p=q;
      }
      if(!t)
        finished=TRUE; // 结束
      else
      { // 从表头回溯
        q=t;
        t=q->ptr.hp;
        q->ptr.hp=p;
        p=q;
        p->tag=LIST;
      } // 继续遍历表尾
    }
  }
}
void Traverse_GL(GList L,void(*v)(GList))
{ // 利用递归算法遍历广义表L, 由bo5-5.cpp改
  if(L) // L不空
    if(L->tag==ATOM) // L为单原子
      v(L);
    else // L为广义表
    {
      v(L);
      Traverse_GL(L->ptr.hp,v);
      Traverse_GL(L->ptr.tp,v);
    }
}
void visit(GList p)
{
  if(p->tag==ATOM)
    printf("mark=%d %c\n",p->mark,p->atom);
  else
    printf("mark=%d list\n",p->mark);
}
void main()
{
  char p[80];
```

```
  SString t;
  GList l;
  printf("请输入广义表l(书写形式: 空表:(),单原子:a,其它:(a,(b),c)):\n");
  gets(p);
  StrAssign(t,p);
  CreateGList(l,t); // 在bo5-5.cpp中
  MarkList(l); // 加标志
  Traverse_GL(l,visit);
}
```

程序运行结果:

```
请输入广义表l(书写形式: 空表:(),单原子:a,其它:(a,(b),c)):
(a,(b,c,d))↙(见图8-16)
mark=1 list
mark=1 a
mark=1 list
mark=1 list
mark=1 b
mark=1 list
mark=1 c
mark=1 list
mark=1 d
```

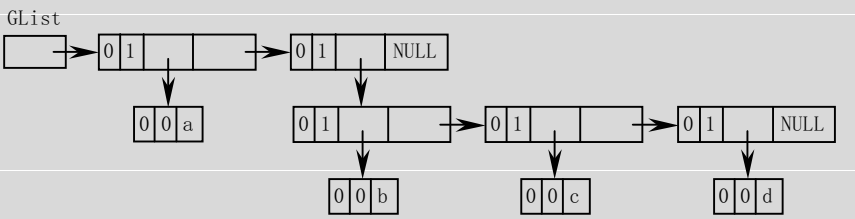


图 8-16　创建加标志域的广义表(a,(b,c,d))的头尾链表存储结构

第9章　查　　找

在实际工作中，经常会遇到需要查找某个数据或某类数据的情况。如在学籍管理的信息中查找某人的各科成绩；在高考信息中查找总分高于分数线的考生的姓名、住址和考号等。一般要按关键字来查找，姓名、考号和总分都可以作为关键字。惟一地标识一个记录的关键字称为主关键字，如考生的考号；标识若干个记录的关键字称为次关键字，如高考分数。

一般来说，数据的值不止包括关键字，还包括其它信息。关键字只是查找的依据。为了描述方便，有时仅讨论关键字。

9.1　静态查找表

静态查找表在查找过程中不改变表的状态——不插不删。它适合用于不变动或不常变动的表的查找。如高考成绩查询表、本单位职工信息表等。

9.1.1　顺序表的查找

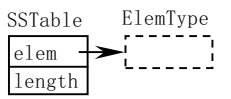


图 9－1　静态查找表的顺序存储结构

```
// c9-1.h 静态查找表的顺序存储结构(见图9－1)
struct SSTable
{
  ElemType *elem; // 数据元素存储空间基址，建表时按实际长度分配，0号单元留空
  int length; // 表长度
};
```

待查找的数据元素类型 ElemType 为结构体类型，其中必含有关键字 key 成员。为了适应各种情况，ElemType 中的 key 的类型及结构体的其它成员不定，由调用基本操作的主程序根据具体情况定义。

```
// c9-7.h 对两个数值型关键字的比较约定为如下的宏定义
#define EQ(a,b) ((a)==(b))
#define LT(a,b) ((a)<(b))
#define LQ(a,b) ((a)<=(b))
```

```
// bo9-1.cpp 静态查找表(顺序表和有序表)的基本操作(7个)，包括算法9.1，9.2
void Creat_Seq(SSTable &ST,ElemType r[],int n)
{ // 操作结果：由含n个数据元素的数组r构造静态顺序查找表ST(见图9－2)
  int i;
  ST.elem=(ElemType*)calloc(n+1,sizeof(ElemType));// 动态生成n+1个数据元素空间(0号单元不用)
```

```
  if(!ST.elem)
    exit(ERROR);
  for(i=1;i<=n;i++)
    ST.elem[i]=r[i-1]; // 将数组r的值依次赋给ST
  ST.length=n;
}
void Ascend(SSTable &ST)
{ // 重建静态查找表为按关键字非降序排序
  int i,j,k;
  for(i=1;i<ST.length;i++)
  {
    k=i;
    ST.elem[0]=ST.elem[i]; // 待比较值存[0]单元
    for(j=i+1;j<=ST.length;j++)
      if LT(ST.elem[j].key,ST.elem[0].key)
      {
        k=j;
        ST.elem[0]=ST.elem[j];
      }
    if(k!=i) // 有更小的值则交换
    {
      ST.elem[k]=ST.elem[i];
      ST.elem[i]=ST.elem[0];
    }
  }
}
void Creat_Ord(SSTable &ST,ElemType r[],int n)
{ // 操作结果：由含n个数据元素的数组r构造按关键字非降序查找表ST
  Creat_Seq(ST,r,n); // 建立无序的查找表ST
  Ascend(ST); // 将无序的查找表ST重建为按关键字非降序查找表
}
Status Destroy(SSTable &ST)
{ // 初始条件：静态查找表ST存在。操作结果：销毁表ST(见图9-3)
  free(ST.elem);
  ST.elem=NULL;
  ST.length=0;
  return OK;
}
int Search_Seq(SSTable ST,KeyType key)
{ // 在顺序表ST中顺序查找其关键字等于key的数据元素。若找到，则返回
  // 该元素在表中的位置；否则返回0。算法9.1
  int i;
  ST.elem[0].key=key; // 哨兵
  for(i=ST.length;!EQ(ST.elem[i].key,key);--i); // 从后往前找
  return i; // 找不到时，i为0
}
int Search_Bin(SSTable ST,KeyType key)
{ // 在有序表ST中折半查找其关键字等于key的数据元素。若找到，则返回
  // 该元素在表中的位置；否则返回0。算法9.2
  int low,high,mid;
  low=1; // 置区间初值
```

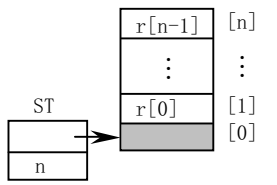


图9-2 静态顺序查找表

图9-3 销毁表ST

```
  high=ST.length;
  while(low<=high)
  {
    mid=(low+high)/2;
    if EQ(key,ST.elem[mid].key)  // 找到待查元素
      return mid;
    else if LT(key,ST.elem[mid].key)
      high=mid-1; // 继续在前半区间进行查找
    else
      low=mid+1; // 继续在后半区间进行查找
  }
  return 0; // 顺序表中不存在待查元素
}
void Traverse(SSTable ST,void(*Visit)(ElemType))
{ // 初始条件：静态查找表ST存在，Visit()是对元素操作的应用函数
  // 操作结果：按顺序对ST的每个元素调用函数Visit()一次且仅一次
  ElemType *p;
  int i;
  p=++ST.elem; // p 指向第 1 个元素
  for(i=1;i<=ST.length;i++)
    Visit(*p++);
}
```

顺序表可以是有序的，也可以是无序的。如果是有序表，可采用折半法查找，每查找一次，就把查找范围缩小一半。在数据量大的情况下，这种方法效率很高。如果是无序表，则只能从表的一端起，逐一查找了。

```
// algo9-1.cpp 检验bo9-1.cpp(顺序表部分)的程序
#include"c1.h"
#define N 5 // 数据元素个数
typedef long KeyType; // 设关键字域为长整型
#define key number // 定义关键字为准考证号
struct ElemType // 数据元素类型(以教科书图9.1高考成绩为例)
{
  long number; // 准考证号，与关键字类型同
  char name[9]; // 姓名(4个汉字加1个串结束标志)
  int politics; // 政治
  int Chinese; // 语文
  int English; // 英语
  int math; // 数学
  int physics; // 物理
  int chemistry; // 化学
  int biology; // 生物
  int total; // 总分
};
#include"c9-7.h"
#include"c9-1.h"
#include"bo9-1.cpp"
void print(ElemType c) // Traverse()调用的函数
{
```

```
  printf("%-8ld%-8s%4d%5d%5d%5d%5d%5d%5d%5d\n",c.number,c.name,c.politics,
  c.Chinese,c.English,c.math,c.physics,c.chemistry,c.biology,c.total);
}
void main()
{
  ElemType r[N]={{179328,"何芳芳",85,89,98,100,93,80,47},
                 {179325,"陈红",85,86,88,100,92,90,45},
                 {179326,"陆华",78,75,90,80,95,88,37},
                 {179327,"张平",82,80,78,98,84,96,40},
                 {179324,"赵小怡",76,85,94,57,77,69,44}}; // 数组不按关键字有序
  SSTable st;
  int i;
  long s;
  for(i=0;i<N;i++) // 计算总分
    r[i].total=r[i].politics+r[i].Chinese+r[i].English+r[i].math+r[i].physics+
    r[i].chemistry+r[i].biology;
  Creat_Seq(st,r,N); // 由数组r产生顺序静态查找表st
  printf("准考证号　姓名　政治 语文 外语 数学 物理 化学 生物 总分\n");
  Traverse(st,print); // 按顺序输出静态查找表st
  printf("请输入待查找人的考号: ");
  scanf("%ld",&s);
  i=Search_Seq(st,s); // 顺序查找
  if(i)
    print(st.elem[i]);
  else
    printf("没找到\n");
  Destroy(st);
}
```

 　程序运行结果：

```
准考证号　姓名　政治 语文 外语 数学 物理 化学 生物 总分
179328　何芳芳　85　89　98　100　93　80　47　592
179325　陈红　85　86　88　100　92　90　45　586
179326　陆华　78　75　90　80　95　88　37　543
179327　张平　82　80　78　98　84　96　40　558
179324　赵小怡　76　85　94　57　77　69　44　502
请输入待查找人的考号: 179326↙
179326　陆华　78　75　90　80　95　88　37　543
```

在 algo9-1.cpp 中，定义 ElemType 的类型为结构体，包括准考证号、姓名、各科成绩及总分。关键字 key 是结构体 ElemType 的一个成员，在此定义为准考证号。由这个例子可见，由于 ElemType 和 key 的类型是在主程序中定义，使得 bo9-1.cpp 的应用范围不受具体元素类型限制。

9.1.2　有序表的查找

```
// algo9-2.cpp 检验bo9-1.cpp(有序表部分)的程序
```

```
#include"c1.h"
#define N 11 // 数据元素个数
typedef int KeyType; // 设关键字域为整型
struct ElemType // 数据元素类型
{
  KeyType key; // 仅有关键字域
};
#include"c9-7.h"
#include"c9-1.h"
#include"bo9-1.cpp"
void print(ElemType c) // Traverse()调用的函数
{
  printf("%d ",c.key);
}
void main()
{
  SSTable st;
  int i;
  KeyType s;
  ElemType r[N]={5,13,19,21,37,56,64,75,80,88,92}; // 数据元素(以教科书第219页的数据为例)
  Creat_Ord(st,r,N); // 由全局数组产生非降序静态查找表st
  Traverse(st,print); // 顺序输出非降序静态查找表st
  printf("\n");
  printf("请输入待查找值的关键字: ");
  scanf("%d",&s);
  i=Search_Bin(st,s); // 折半查找有序表
  if(i)
    printf("%d 是第%d个记录的关键字\n",st.elem[i].key,i);
  else
    printf("没找到\n");
  Destroy(st);
}
```

 　程序运行结果:

```
5 13 19 21 37 56 64 75 80 88 92
请输入待查找值的关键字: 56↙
56 是第6个记录的关键字
```

在 algo9-2.cpp 中，待查找的数据本身就是关键字。但为利用 bo9-1.cpp 中的函数，仍定义 ElemType 为结构体，其中只有一个成员，就是关键字 key。

9.1.3　静态树表的查找

静态树表是把有序的静态查找表根据数据被查找的概率生成一棵二叉树(在 c6-2.h 中定义)，使得从二叉树的根结点起，查找左右子树的概率大致相等，使平均查找长度较短。若每个数据的查找概率相等，直接使用折半法即可，不必生成二叉树。

```
// algo9-3.cpp 静态查找表(静态树表)的操作，包括算法9.3，9.4
#include"c1.h"
#define N 9 // 数据元素个数
typedef char KeyType; // 设关键字域为字符型
struct ElemType // 数据元素类型
{
  KeyType key; // 关键字
  int weight; // 权值
};
#include"c9-7.h"
#include"c9-1.h"
#include"bo9-1.cpp"
typedef ElemType TElemType;
#include"c6-2.h"
Status SecondOptimal(BiTree &T, ElemType R[],int sw[],int low,int high)
{ // 由有序表R[low..high]及其累计权值表sw(其中sw[0]==0)递归构造次优查找树T。算法9.3
  int i,j;
  double min,dw;
  i=low;
  min=fabs(sw[high]-sw[low]);
  dw=sw[high]+sw[low-1];
  for(j=low+1;j<=high;++j) // 选择最小的△Pi值
    if(fabs(dw-sw[j]-sw[j-1])<min)
    {
      i=j;
      min=fabs(dw-sw[j]-sw[j-1]);
    }
  if(!(T=(BiTree)malloc(sizeof(BiTNode))))
    return ERROR;
  T->data=R[i]; // 生成结点
  if(i==low)
    T->lchild=NULL; // 左子树空
  else
    SecondOptimal(T->lchild,R,sw,low,i-1); // 构造左子树
  if(i==high)
    T->rchild=NULL; // 右子树空
  else
    SecondOptimal(T->rchild,R,sw,i+1,high); // 构造右子树
  return OK;
}
void FindSW(int sw[],SSTable ST)
{ // 按照有序表ST中各数据元素的Weight域求累计权值表sw
  int i;
  sw[0]=0;
  for(i=1;i<=ST.length;i++)
    sw[i]=sw[i-1]+ST.elem[i].weight;
}
typedef BiTree SOSTree; // 次优查找树采用二叉链表的存储结构
void CreateSOSTree(SOSTree &T,SSTable ST)
{ // 由有序表ST构造一棵次优查找树T。ST的数据元素含有权域weight。算法9.4
  int sw[N+1]; // 累计权值
```

```
  if(ST.length==0)
    T=NULL;
  else
  {
    FindSW(sw,ST); // 按照有序表ST中各数据元素的weight域求累计权值表sw
    SecondOptimal(T,ST.elem,sw,1,ST.length);
  }
}
Status Search_SOSTree(SOSTree &T,KeyType key)
{ // 在次优查找树T中查找关键字等于key的元素。找到则返回OK；否则返回FALSE
  while(T) // T非空
    if(T->data.key==key)
      return OK;
    else if(T->data.key>key)
      T=T->lchild;
    else
      T=T->rchild;
  return FALSE; // 顺序表中不存在待查元素
}
void print(ElemType c) // Traverse()调用的函数
{
  printf("(%c %d) ",c.key,c.weight);
}
void main()
{
  SSTable st;
  SOSTree t;
  Status i;
  KeyType s; // 以教科书例9－1的数据为例
  ElemType r[N]={{'A',1},{'B',1},{'C',2},{'D',5},{'E',3},{'F',4},{'G',4},{'H',3},{'I',5}};
  Creat_Ord(st,r,N); // 由全局数组产生非降序静态查找表st
  Traverse(st,print);
  CreateSOSTree(t,st); // 由有序表构造一棵次优查找树
  printf("\n请输入待查找的字符: ");
  scanf("%c",&s);
  i=Search_SOSTree(t,s);
  if(i)
    printf("%c的权值是%d\n",s,t->data.weight);
  else
    printf("表中不存在此字符\n");
}
```

程序运行结果：

```
(A 1) (B 1) (C 2) (D 5) (E 3) (F 4) (G 4) (H 3) (I 5)
请输入待查找的字符: G↙
G的权值是4
```

9.1.4　索引顺序表的查找

9.2 动态查找表

动态查找表在查找过程中可改变表的状态，即可插入或删除数据，它适合用在表的内容要经常变化的情况下，如飞机航班的旅客信息表。

9.2.1 二叉排序树和平衡二叉树

```
// func9-1.cpp 包括算法9.5(a)和func6-3.cpp,bo9-2.cpp和bo9-3.cpp调用
#include"func6-3.cpp"
// 包括InitBiTree()、DestroyBiTree()、PreOrderTraverse()和InOrderTraverse()4函数
#define InitDSTable InitBiTree // 与初始化二叉树的操作同
#define DestroyDSTable DestroyBiTree // 与销毁二叉树的操作同
#define TraverseDSTable InOrderTraverse // 与中序遍历二叉树的操作同
BiTree SearchBST(BiTree T,KeyType key)
{ // 在根指针T所指二叉排序树中递归地查找某关键字等于key的数据元素，
  // 若查找成功，则返回指向该数据元素结点的指针；否则返回空指针。算法9.5(a)
  if(!T||EQ(key,T->data.key))
    return T; // 查找结束
  else if LT(key,T->data.key) // 在左子树中继续查找
    return SearchBST(T->lchild,key);
  else
    return SearchBST(T->rchild,key); // 在右子树中继续查找
}


// bo9-2.cpp 动态查找表(二叉排序树)的基本操作(8个)，包括算法9.5(b)，9.6～9.8
typedef ElemType TElemType;
#include"c6-2.h" // 二叉树的存储结构
#include"func9-1.cpp"
Status SearchBST(BiTree &T,KeyType key,BiTree f,BiTree &p) // 算法9.5(b)
{ // 在根指针T所指二叉排序树中递归地查找其关键字等于key的数据元素，若查找
  // 成功，则指针p指向该数据元素结点，并返回TRUE；否则指针p指向查找路径上
  // 访问的最后一个结点并返回FALSE，指针f指向T的双亲，其初始调用值为NULL
  if(!T) // 查找不成功
  {
    p=f;
    return FALSE;
  }
  else if EQ(key,T->data.key) //  查找成功
  {
    p=T;
    return TRUE;
  }
  else if LT(key,T->data.key)
    return SearchBST(T->lchild,key,T,p); // 在左子树中继续查找
  else
    return SearchBST(T->rchild,key,T,p); //  在右子树中继续查找
}
Status InsertBST(BiTree &T, ElemType e)
{ // 当二叉排序树T中不存在关键字等于e.key的元素时，插入e并返回TRUE；否则返回FALSE。算法9.6
```

```
  BiTree p,s;
  if(!SearchBST(T,e.key,NULL,p)) // 查找不成功
  {
    s=(BiTree)malloc(sizeof(BiTNode));
    s->data=e;
    s->lchild=s->rchild=NULL;
    if(!p)
      T=s; // 被插结点*s为新的根结点
    else if LT(e.key,p->data.key)
      p->lchild=s; // 被插结点*s为左孩子
    else
      p->rchild=s; // 被插结点*s为右孩子
    return TRUE;
  }
  else
    return FALSE; // 树中已有关键字相同的结点，不再插入
}
void Delete(BiTree &p)
{ // 从二叉排序树中删除结点p，并重接它的左或右子树。算法9.8
  BiTree q,s;
  if(!p->rchild) // p的右子树空则只需重接它的左子树(待删结点是叶子也走此分支)
  {
    q=p;
    p=p->lchild;
    free(q);
  }
  else if(!p->lchild) // p的左子树空，只需重接它的右子树
  {
    q=p;
    p=p->rchild;
    free(q);
  }
  else // p的左右子树均不空
  {
    q=p;
    s=p->lchild;
    while(s->rchild) // 转左，然后向右到尽头(找待删结点的前驱)
    {
      q=s;
      s=s->rchild;
    }
    p->data=s->data; // s指向被删结点的“前驱”(将被删结点前驱的值取代被删结点的值)
    if(q!=p) // 情况1
      q->rchild=s->lchild; // 重接*q的右子树
    else // 情况2
      q->lchild=s->lchild; // 重接*q的左子树
    free(s);
  }
}
Status DeleteBST(BiTree &T,KeyType key)
{ // 若二叉排序树T中存在关键字等于key的数据元素时，则删除该数据元素结点，
```

```
  // 并返回TRUE；否则返回FALSE。算法9.7
  if(!T) // 不存在关键字等于key的数据元素
    return FALSE;
  else
  {
    if EQ(key,T->data.key) // 找到关键字等于key的数据元素
      Delete(T);
    else if LT(key,T->data.key)
      DeleteBST(T->lchild,key);
    else
      DeleteBST(T->rchild,key);
    return TRUE;
  }
}

// algo9-4.cpp 检验bo9-2.cpp的程序
#include"c1.h"
#define N 10 // 数据元素个数
typedef int KeyType; // 设关键字域为整型
struct ElemType // 数据元素类型
{
  KeyType key;
  int others;
};
#include"c9-7.h"
#include"bo9-2.cpp"
void print(ElemType c)
{
  printf("(%d,%d) ",c.key,c.others);
}
void main()
{
  BiTree dt,p;
  int i;
  KeyType j;
  ElemType r[N]={{45,1},{12,2},{53,3},{3,4},{37,5},{24,6},{100,7},{61,8},{90,9},{78,10}};
  // 以教科书图9.7(a)为例，另加除关键字之外的其他信息
  InitDSTable(dt); // 构造空表
  for(i=0;i<N;i++)
    InsertBST(dt,r[i]); // 依次插入数据元素
  TraverseDSTable(dt,print);
  printf("\n请输入待查找的值: ");
  scanf("%d",&j);
  p=SearchBST(dt,j);
  if(p)
  {
    printf("表中存在此值。");
    DeleteBST(dt,j);
    printf("删除此值后:\n");
    TraverseDSTable(dt,print);
    printf("\n");
```

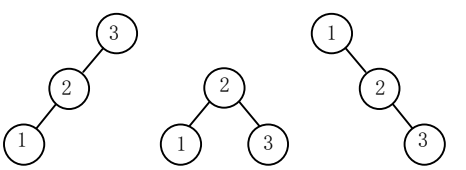
```
  }
  else
    printf("表中不存在此值\n");
  DestroyDSTable(dt);
}
```

程序运行结果：

```
(3,4) (12,2) (24,6) (37,5) (45,1) (53,3) (61,8) (78,10) (90,9) (100,7)
请输入待查找的值：53↙
表中存在此值。删除此值后：
(3,4) (12,2) (24,6) (37,5) (45,1) (61,8) (78,10) (90,9) (100,7)
```

二叉排序树中任何一个结点，其左子树上所有结点的关键字值均小于该结点的关键字值；其右子树上所有结点的关键字值均大于该结点的关键字值。中序遍历二叉排序树可得到按关键字有序的序列。

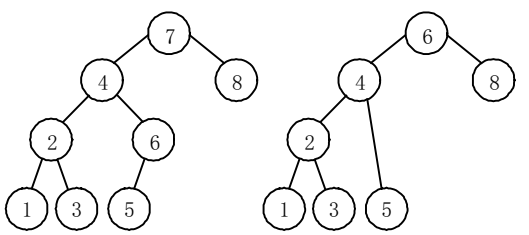
二叉排序树的形态与数据元素插入的顺序有关。图 9－4 显示了三种由 1、2、3 三个数据元素组成的二叉排序树。当 3 个数据元素的输入顺序不同，生成了 3 棵不同的二叉排序树。



(a) 输入 3,2,1 (b) 输入 2,3,1 (c) 输入 1,2,3

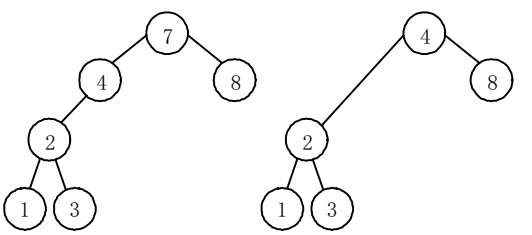
图 9－4　由 1, 2, 3 组成的三棵二叉排序树

在二叉排序树中插入一个结点，这个结点总是叶子结点。删除一个结点分 3 种情况：删除叶子结点；删除只有一个子树的结点；删除有两个子树的结点。前两种情况很好处理，最后一种情况算法 9.8 又分为两种情况来处理：① 待删结点的左孩子有右子树。以图 9－5 为例来说明：找到待删结点 7 的左子树的最右下结点 6(这个结点没有右子树且其值是左子树中最大的)，用 6 取代 7；原来指向 6 的指针指向 6 的左孩子；最后释放 6 这个结点。② 待删结点的左孩子没右子树。以图 9－6 为例来说明：用待删结点 7 的左孩子的值 4(这个值是左子树中最大的)取代 7；待删结点的左孩子指针指向 4 的左孩子；删除结点 4。这样仍然构成排序二叉树。



(a) 删除结点 7 之前　(b)　删除结点 7 之后

图 9－5　算法 9.8 图示(情况 1)



(a) 删除结点 7 之前　(b)　删除结点 7 之后

图 9－6　算法 9.8 图示(情况 2)

如果数据输入的顺序不当，排序二叉树的深度有可能很深，导致平均查找长度很大，失去了排序二叉树存在的意义。平衡二叉树能克服排序二叉树的这个缺点，它通过调换树或子树的根结点，保持树的深度尽量浅。

```
// c9-2.h 平衡二叉树的类型(见图9－7)
typedef struct BSTNode
{
  ElemType data;
  int bf; // 结点的平衡因子
  BSTNode *lchild,*rchild; // 左、右孩子指针
}BSTNode,*BSTree;
```

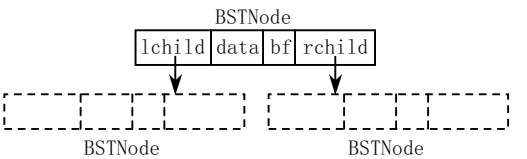


图 9－7 平衡二叉树类型的存储结构

```
// bo9-3.cpp 动态查找表(平衡二叉树)的基本操作，包括算法9.9～9.12
typedef ElemType TElemType; // 定义二叉树基本操作的树元素类型
typedef BSTree BiTree; // 定义二叉树基本操作的指针类型
#include"func9-1.cpp"
void R_Rotate(BSTree &p)
{ // 对以*p为根的二叉排序树作右旋处理，处理之后p指向新的树
  // 根结点，即旋转处理之前的左子树的根结点。算法9.9(见图9－8)
  BSTree lc;
  lc=p->lchild; // lc指向p的左子树根结点
  p->lchild=lc->rchild; // lc的右子树挂接为p的左子树
  lc->rchild=p;
  p=lc; // p指向新的根结点
}
```

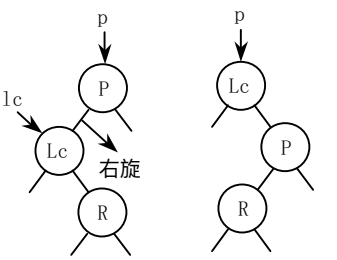


图 9－8 调用 R_Rotate()图示

```
void L_Rotate(BSTree &p)
{ // 对以*p为根的二叉排序树作左旋处理，处理之后p指向新的树
  // 根结点，即旋转处理之前的右子树的根结点。算法9.10(见图9－9)
  BSTree rc;
  rc=p->rchild; // rc指向p的右子树根结点
  p->rchild=rc->lchild; // rc的左子树挂接为p的右子树
  rc->lchild=p;
  p=rc; // p指向新的根结点
}
#define LH +1 // 左高
#define EH 0  // 等高
#define RH -1 // 右高
void LeftBalance(BSTree &T)
{ // 对以指针T所指结点为根的二叉树作左平衡旋转处理，本算法结束时，
  // 指针T指向新的根结点。算法9.12
  BSTree lc,rd;
  lc=T->lchild; // lc指向*T的左子树根结点
  switch(lc->bf)
  { // 检查*T的左子树的平衡度，并作相应平衡处理
    case LH: // 新结点插入在*T的左孩子的左子树上，要作单右旋处理
             T->bf=lc->bf=EH;
             R_Rotate(T);
             break;
    case RH: // 新结点插入在*T的左孩子的右子树上，要作双旋处理(见图9－10(a))
             rd=lc->rchild; // rd指向*T的左孩子的右子树根
             switch(rd->bf)
             { // 修改*T及其左孩子的平衡因子
               case LH: T->bf=RH;
                        lc->bf=EH;
```

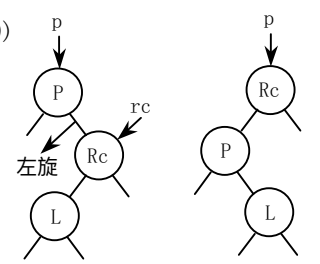
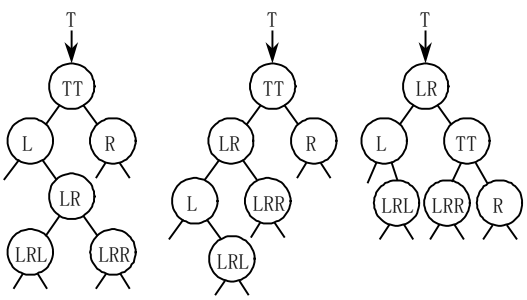


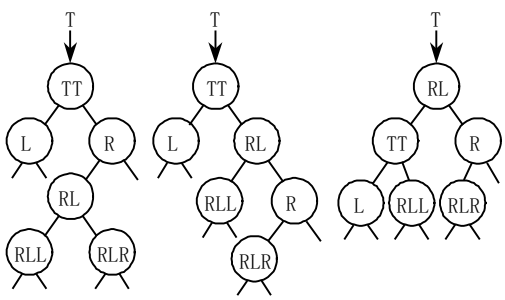
图 9－9 调用 L_Rotate()图示

```
                    break;
              case EH: T->bf=lc->bf=EH;
                    break;
              case RH: T->bf=EH;
                    lc->bf=LH;
            }
            rd->bf=EH;
            L_Rotate(T->lchild); // 对*T的左子树作左旋平衡处理(见图9－10(b))
            R_Rotate(T); // 对*T作右旋平衡处理(见图9－10(c))
  }
}
void RightBalance(BSTree &T)
{ // 对以指针T所指结点为根的二叉树作右平衡旋转处理，本算法结束时，
  // 指针T指向新的根结点
  BSTree rc,rd;
  rc=T->rchild; // rc指向*T的右子树根结点
  switch(rc->bf)
  { // 检查*T的右子树的平衡度，并作相应平衡处理
    case RH: // 新结点插入在*T的右孩子的右子树上，要作单左旋处理
            T->bf=rc->bf=EH;
            L_Rotate(T);
            break;
    case LH: // 新结点插入在*T的右孩子的左子树上，要作双旋处理(见图9－11(a))
            rd=rc->lchild; // rd指向*T的右孩子的左子树根
            switch(rd->bf)
            { // 修改*T及其右孩子的平衡因子
              case RH: T->bf=LH;
                    rc->bf=EH;
                    break;
              case EH: T->bf=rc->bf=EH;
                    break;
              case LH: T->bf=EH;
                    rc->bf=RH;
            }
            rd->bf=EH;
            R_Rotate(T->rchild); // 对*T的右子树作右旋平衡处理(见图9－11(b))
            L_Rotate(T); // 对*T作左旋平衡处理(见图9－11(c))
  }
}
```



(a) 旋转前　(b) 左子树左旋　(c) T 右旋

图 9－10　LR 型平衡旋转图示



(a) 旋转前　(b) 右子树右旋　(c) T 左旋

图 9－11　RL 型平衡旋转图示

```
Status InsertAVL(BSTree &T,ElemType e,Status &taller)
{ // 若在平衡的二叉排序树T中不存在和e有相同关键字的结点，则插入一个
  // 数据元素为e的新结点，并返回1；否则返回0。若因插入而使二叉排序树
  // 失去平衡，则作平衡旋转处理，布尔变量taller反映T长高与否。算法9.11
  if(!T)
  { // 插入新结点，树“长高”，置taller为TRUE
    T=(BSTree)malloc(sizeof(BSTNode));
    T->data=e;
    T->lchild=T->rchild=NULL;
    T->bf=EH;
    taller=TRUE;
  }
  else
  {
    if EQ(e.key,T->data.key)
    { // 树中已存在和e有相同关键字的结点则不再插入
      taller=FALSE;
      return FALSE;
    }
    if LT(e.key,T->data.key)
    { // 应继续在*T的左子树中进行搜索
      if(!InsertAVL(T->lchild,e,taller)) // 未插入
        return FALSE;
      if(taller) //  已插入到*T的左子树中且左子树“长高”
        switch(T->bf) // 检查*T的平衡度
        {
          case LH: // 原本左子树比右子树高，需要作左平衡处理
                   LeftBalance(T);
                   taller=FALSE;
                   break;
          case EH: // 原本左、右子树等高，现因左子树增高而使树增高
                   T->bf=LH;
                   taller=TRUE;
                   break;
          case RH: T->bf=EH; // 原本右子树比左子树高，现左、右子树等高
                   taller=FALSE;
        }
    }
    else
    { // 应继续在*T的右子树中进行搜索
      if(!InsertAVL(T->rchild,e,taller)) // 未插入
        return FALSE;
      if(taller) // 已插入到T的右子树且右子树“长高”
        switch(T->bf) // 检查T的平衡度
        {
          case LH: T->bf=EH; // 原本左子树比右子树高，现左、右子树等高
                   taller=FALSE;
                   break;
          case EH: // 原本左、右子树等高，现因右子树增高而使树增高
                   T->bf=RH;
                   taller=TRUE;
```

```
                    break;
            case RH: // 原本右子树比左子树高，需要作右平衡处理
                    RightBalance(T);
                    taller=FALSE;
          }
      }
    }
  return TRUE;
}
```

在构造平衡二叉树时，每插入一个结点就检查是否仍为平衡二叉树。如果由于插入结点导致二叉树失衡，则要通过旋转、改变根结点等措施保证仍为平衡的二叉排序树。插入结点导致二叉树不平衡有 4 种情况：

(1) 在左子树的左孩子分支上插入结点，导致不平衡，称 LL 型；

(2) 在左子树的右孩子分支上插入结点，导致不平衡，称 LR 型；

(3) 在右子树的右孩子分支上插入结点，导致不平衡，称 RR 型；

(4) 在右子树的左孩子分支上插入结点，导致不平衡，称 RL 型。

对于 LL 型，做右旋，如图 9－8 所示，降低了左子树的左孩子分支的深度，重新构成平衡二叉树。相应地，对于 RR 型，做左旋，如图 9－9 所示。对于 LR 型，先令左子树左旋，再令整个树右旋，如图 9－10 所示，降低了 LR 的两个子树的深度，重新构成平衡二叉树。对于 RL 型，先令右子树右旋，再令整个树左旋，如图 9－11 所示，降低了 RL 的两个子树的深度，重新构成平衡二叉树。

```
// algo9-5.cpp 检验bo9-3.cpp的程序
#include"c1.h"
#define N 5 // 数据元素个数
typedef char KeyType; // 设关键字域为字符型
struct ElemType // 数据元素类型
{
  KeyType key;
  int order;
};
#include"c9-7.h"
#include"c9-2.h"
#include"bo9-3.cpp"
void print(ElemType c)
{
  printf("(%d,%d)",c.key,c.order);
}
void main()
{
  BSTree dt,p;
  Status k;
  int i;
  KeyType j;
  ElemType r[N]={{13,1},{24,2},{37,3},{90,4},{53,5}}; // (以教科书图9.12为例)
  InitDSTable(dt); // 初始化空树
```

```
  for(i=0;i<N;i++)
    InsertAVL(dt,r[i],k); // 建平衡二叉树
  PreOrderTraverse(dt,print); // 先序遍历平衡二叉树
  printf("先序遍历平衡二叉树\n");
  TraverseDSTable(dt,print); // 按关键字顺序遍历二叉树
  printf("\n请输入待查找的关键字: ");
  scanf("%d",&j);
  p=SearchBST(dt,j); // 查找给定关键字的记录
  if(p)
    print(p->data);
  else
    printf("表中不存在此值");
  printf("\n");
  DestroyDSTable(dt);
}
```

程序运行结果(见图 9－12)：

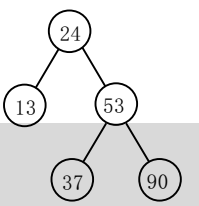


图 9－12　生成的排序二叉树

```
(24,2)(13,1)(53,5)(37,3)(90,4)先序遍历平衡二叉树
(13,1)(24,2)(37,3)(53,5)(90,4)
请输入待查找的关键字: 37↙
(37,3)
```

9.2.2　B_树和 B^+树

```
// c9-3.h B_树的结点类型
struct Record // 记录类型(见图9－13)
{
  KeyType key; // 关键字
  Others others; // 其它部分(由主程定义)
};
typedef struct BTNode
{
  int keynum; // 结点中关键字个数, 即结点的大小
  BTNode *parent; // 指向双亲结点
  struct Node // 结点向量类型(见图9－14)
  {
    KeyType key; // 关键字向量
    BTNode *ptr; // 子树指针向量
    Record *recptr; // 记录指针向量
  }node[m+1]; // key,recptr的0号单元未用
}BTNode,*BTree; // B_树结点和B_树的类型(见图9－15)
struct Result // B_树的查找结果类型(见图9－16)
{
  BTNode *pt; // 指向找到的结点
  int i; // 1..m, 在结点中的关键字序号
  int tag; // 1: 查找成功, 0: 查找失败
}
```

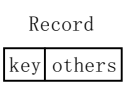


图 9－13　记录类型

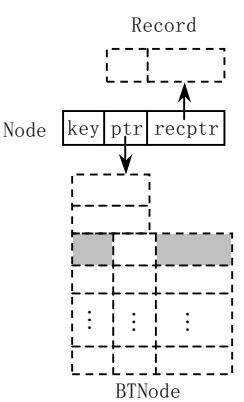


图 9－14　Node 结点向量类型

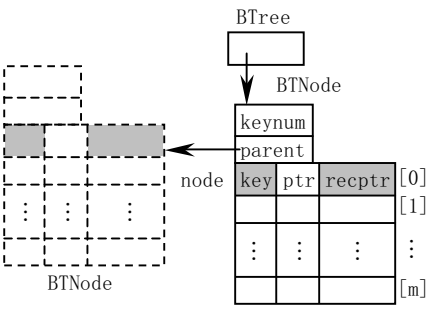


图 9－15　m 阶 B_树结点及指针类型

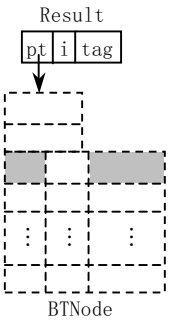


图 9－16　B_树的查找结果类型

```
// bo9-4.cpp 动态查找表(B_树)的基本操作，包括算法9.13，9.14
void InitDSTable(BTree &DT)
{ // 操作结果：构造一个空的动态查找表DT
  DT=NULL;
}
void DestroyDSTable(BTree &DT)
{ // 初始条件：动态查找表DT存在。操作结果：销毁动态查找表DT
  int i;
  if(DT) // 非空树
  {
    for(i=0;i<=DT->keynum;i++)
      DestroyDSTable(DT->node[i].ptr); // 依次销毁第i棵子树
    free(DT); // 释放根结点
    DT=NULL; // 空指针赋0
  }
}
int Search(BTree p, KeyType K)
{ // 在p->node[1..keynum].key中查找i，使得p->node[i].key≤K＜p->node[i+1].key
  int i=0,j;
  for(j=1;j<=p->keynum;j++)
    if(p->node[j].key<=K)
      i=j;
  return i;
}
Result SearchBTree(BTree T, KeyType K)
{ // 在m阶B_树T上查找关键字K，返回结果(pt,i,tag)。若查找成功，则特征值
  // tag=1，指针pt所指结点中第i个关键字等于K；否则特征值tag=0，等于K的
  // 关键字应插入在指针Pt所指结点中第i和第i+1个关键字之间。算法9.13
  BTree p=T,q=NULL; // 初始化，p指向待查结点，q指向p的双亲
  Status found=FALSE;
  int i=0;
  Result r;
  while(p&&!found)
  {
    i=Search(p,K); // p->node[i].key≤K<p->node[i+1].key
    if(i>0&&p->node[i].key==K) // 找到待查关键字
      found=TRUE;
    else
    {
```

```
      q=p;
      p=p->node[i].ptr;
    }
  }
  r.i=i;
  if(found) // 查找成功
  {
    r.pt=p;
    r.tag=1;
  }
  else //  查找不成功，返回K的插入位置信息
  {
    r.pt=q;
    r.tag=0;
  }
  return r;
}
void Insert(BTree &q,int i,Record *r,BTree ap)
{ // 将r->key、r和ap分别插入到q->key[i+1]、q->recptr[i+1]和q->ptr[i+1]中
  int j;
  for(j=q->keynum;j>i;j--) // 空出q->node[i+1]
    q->node[j+1]=q->node[j];
  q->node[i+1].key=r->key;
  q->node[i+1].ptr=ap;
  q->node[i+1].recptr=r;
  q->keynum++;
}
void split(BTree &q,BTree &ap)
{ // 将结点q分裂成两个结点，前一半保留，后一半移入新生结点ap
  int i,s=(m+1)/2;
  ap=(BTree)malloc(sizeof(BTNode)); // 生成新结点ap
  ap->node[0].ptr=q->node[s].ptr; // 后一半移入ap
  for(i=s+1;i<=m;i++)
  {
    ap->node[i-s]=q->node[i];
    if(ap->node[i-s].ptr)
      ap->node[i-s].ptr->parent=ap;
  }
  ap->keynum=m-s;
  ap->parent=q->parent;
  q->keynum=s-1; // q的前一半保留，修改keynum
}
void NewRoot(BTree &T,Record *r,BTree ap)
{ // 生成含信息(T,r,ap)的新的根结点*T，原T和ap为子树指针
  BTree p;
  p=(BTree)malloc(sizeof(BTNode));
  p->node[0].ptr=T;
  T=p;
  if(T->node[0].ptr)
    T->node[0].ptr->parent=T;
  T->parent=NULL;
```

```
  T->keynum=1;
  T->node[1].key=r->key;
  T->node[1].recptr=r;
  T->node[1].ptr=ap;
  if(T->node[1].ptr)
    T->node[1].ptr->parent=T;
}
void InsertBTree(BTree &T,Record *r,BTree q,int i)
{ // 在m阶B_树T上结点*q的key[i]与key[i+1]之间插入关键字K的指针r。若引起
  // 结点过大，则沿双亲链进行必要的结点分裂调整，使T仍是m阶B_树。算法9.14改
  BTree ap=NULL;
  Status finished=FALSE;
  int s;
  Record *rx;
  rx=r;
  while(q&&!finished)
  {
    Insert(q,i,rx,ap);//将r->key、r和ap分别插入到q->key[i+1]、q->recptr[i+1]和q->ptr[i+1]中
    if(q->keynum<m)
      finished=TRUE; // 插入完成
    else
    { // 分裂结点*q
      s=(m+1)/2;
      rx=q->node[s].recptr;
      split(q,ap); // 将q->key[s+1..m],q->ptr[s..m]和q->recptr[s+1..m]移入新结点*ap
      q=q->parent;
      if(q)
        i=Search(q,rx->key); // 在双亲结点*q中查找rx->key的插入位置
    }
  }
  if(!finished) // T是空树(参数q初值为NULL)或根结点已分裂为结点*q和*ap
    NewRoot(T,rx,ap); // 生成含信息(T,rx,ap)的新的根结点*T，原T和ap为子树指针
}
void TraverseDSTable(BTree DT,void(*Visit)(BTNode,int))
{ // 初始条件：动态查找表DT存在，Visit是对结点操作的应用函数
  // 操作结果：按关键字的顺序对DT的每个结点调用函数Visit()一次且至多一次
  int i;
  if(DT) // 非空树
  {
    if(DT->node[0].ptr) // 有第0棵子树
      TraverseDSTable(DT->node[0].ptr,Visit);
    for(i=1;i<=DT->keynum;i++)
    {
      Visit(*DT,i);
      if(DT->node[i].ptr) // 有第i棵子树
        TraverseDSTable(DT->node[i].ptr,Visit);
    }
  }
}

// algo9-6.cpp 检验bo9-4.cpp的程序
```

```
#include"c1.h"
#define m 3 // B_树的阶，暂设为3
#define N 16 // 数据元素个数
#define MAX 5 // 字符串最大长度+1
typedef int KeyType; // 设关键字域为整型
struct Others // 记录的其它部分
{
  char info[MAX];
};
#include"c9-3.h"
#include"bo9-4.cpp"
void print(BTNode c,int i) // TraverseDSTable()调用的函数
{
  printf("(%d,%s)",c.node[i].key,c.node[i].recptr->others.info);
}
void main()
{
  Record r[N]={{24,"1"},{45,"2"},{53,"3"},{12,"4"},{37,"5"},{50,"6"},{61,"7"},{90,"8"},
               {100,"9"},{70,"10"},{3,"11"},{30,"12"},{26,"13"},{85,"14"},{3,"15"},
               {7,"16"}}; // (以教科书中图9.16为例)
  BTree t;
  Result s;
  int i;
  InitDSTable(t);
  for(i=0;i<N;i++)
  {
    s=SearchBTree(t,r[i].key);
    if(!s.tag)
      InsertBTree(t,&r[i],s.pt,s.i);
  }
  printf("按关键字的顺序遍历B_树:\n");
  TraverseDSTable(t,print);
  printf("\n请输入待查找记录的关键字: ");
  scanf("%d",&i);
  s=SearchBTree(t,i);
  if(s.tag)
    print(*(s.pt),s.i);
  else
    printf("没找到");
  printf("\n");
  DestroyDSTable(t);
}
```

程序运行结果：

```
按关键字的顺序遍历B_树:
(3,11)(7,16)(12,4)(24,1)(26,13)(30,12)(37,5)(45,2)(50,6)(53,3)(61,7)(70,10)(85,14)(90,8)
(100,9)
请输入待查找记录的关键字: 12↙
(12,4)
```

9.2.3　键树

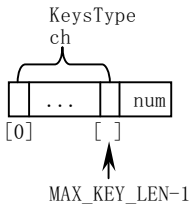


图 9－17　KeysType 关键字类型

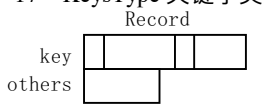


图 9－18　Record 记录类型

```
// c9-4.h 双链树的存储结构
#define MAX_KEY_LEN 16 // 关键字的最大长度
struct KeysType // 关键字类型(见图9－17)
{
  char ch[MAX_KEY_LEN]; // 关键字
  int num; // 关键字长度
};
struct Record // 记录类型(见图9－18)
{
  KeysType key; // 关键字
  Others others; // 其它部分(由主程定义)
};
enum NodeKind{LEAF,BRANCH}; // 结点种类: {叶子,分支}
typedef struct DLTNode
                  // 双链树类型(见图9－19)
{
  char symbol;
  DLTNode *next; // 指向兄弟结点的指针
  NodeKind kind;
  union
  {
    Record *infoptr; // 叶子结点的记录指针
    DLTNode *first; // 分支结点的孩子链指针
  };
}DLTNode,*DLTree;
```

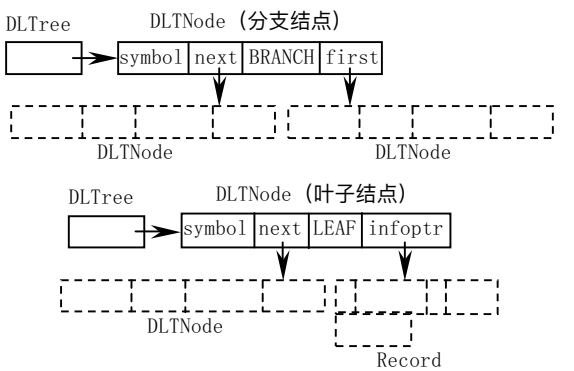


图 9－19　双链树类型

```
// bo9-5.cpp 动态查找表(双链键树)的基本操作，包括算法9.15
void InitDSTable(DLTree &DT)
{ // 操作结果: 构造一个空的双链键树DT
  DT=NULL;
}
void DestroyDSTable(DLTree &DT)
{ // 初始条件: 双链键树DT存在。操作结果: 销毁双链键树DT
  if(DT) // 非空树
  {
    if(DT->kind==BRANCH&&DT->first) // *DT是分支结点且有孩子
      DestroyDSTable(DT->first); // 销毁孩子子树
    if(DT->next) // 有兄弟
      DestroyDSTable(DT->next); // 销毁兄弟子树
    free(DT); // 释放根结点
    DT=NULL; // 空指针赋0
  }
}
Record *SearchDLTree(DLTree T,KeysType K)
{ // 在非空双链键树T中查找关键字等于K的记录，若存在，
  // 则返回指向该记录的指针；否则返回空指针。算法9.15，有改动
  DLTree p;
  int i;
```

```
  if(T)
  {
    p=T; // 初始化
    i=0;
    while(p&&i<K.num)
    {
      while(p&&p->symbol!=K.ch[i]) // 查找关键字的第i位
        p=p->next;
      if(p&&i<K.num) // 准备查找下一位
        p=p->first;
      ++i;
    } // 查找结束
    if(!p) // 查找不成功
      return NULL;
    else // 查找成功
      return p->infoptr;
  }
  else
    return NULL; // 树空
}
void InsertDSTable(DLTree &DT,Record *r)
{ // 初始条件: 双链键树DT存在, r为待插入的数据元素的指针
  // 操作结果: 若DT中不存在其关键字等于(*r).key.ch的数据元素, 则按关键字顺序插r到DT中
  DLTree p=NULL,q,ap;
  int i=0;
  KeysType K=r->key;
  if(!DT&&K.num) // 空树且关键字符串非空
  {
    DT=ap=(DLTree)malloc(sizeof(DLTNode));
    for(;i<K.num;i++) // 插入分支结点
    {
      if(p)
        p->first=ap;
      ap->next=NULL;
      ap->symbol=K.ch[i];
      ap->kind=BRANCH;
      p=ap;
      ap=(DLTree)malloc(sizeof(DLTNode));
    }
    p->first=ap; // 插入叶子结点
    ap->next=NULL;
    ap->symbol=Nil;
    ap->kind=LEAF;
    ap->infoptr=r;
  }
  else // 非空树
  {
    p=DT; // 指向根结点
    while(p&&i<K.num)
    {
      while(p&&p->symbol<K.ch[i]) // 沿兄弟结点查找
```

```
      {
        q=p;
        p=p->next;
      }
      if(p&&p->symbol==K.ch[i]) // 找到与K.ch[i]相符的结点
      {
        q=p;
        p=p->first; // p指向将与K.ch[i+1]比较的结点
        ++i;
      }
      else // 没找到，插入关键字
      {
        ap=(DLTree)malloc(sizeof(DLTNode));
        if(q->first==p)
          q->first=ap; // 在长子的位置插入
        else // q->next==p
          q->next=ap; // 在兄弟的位置插入
        ap->next=p;
        ap->symbol=K.ch[i];
        ap->kind=BRANCH;
        p=ap;
        ap=(DLTree)malloc(sizeof(DLTNode));
        i++;
        for(;i<K.num;i++) // 插入分支结点
        {
          p->first=ap;
          ap->next=NULL;
          ap->symbol=K.ch[i];
          ap->kind=BRANCH;
          p=ap;
          ap=(DLTree)malloc(sizeof(DLTNode));
        }
        p->first=ap; // 插入叶子结点
        ap->next=NULL;
        ap->symbol=Nil;
        ap->kind=LEAF;
        ap->infoptr=r;
      }
    }
  }
}
struct SElemType // 定义栈元素类型
{
  char ch;
  DLTree p;
};
#include"c3-1.h" // 顺序栈
#include"bo3-1.cpp" // 顺序栈的基本操作
void TraverseDSTable(DLTree DT,void(*Vi)(Record))
{ // 初始条件：双链键树DT存在，Vi是对结点操作的应用函数，ViR是对记录操作的应用函数
  // 操作结果：按关键字的顺序输出关键字及其对应的记录
```

```
  SqStack s;
  SElemType e;
  DLTree p;
  int i=0,n=8;
  if(DT)
  {
    InitStack(s);
    e.p=DT;
    e.ch=DT->symbol;
    Push(s,e);
    p=DT->first;
    while(p->kind==BRANCH) // 分支结点
    {
      e.p=p;
      e.ch=p->symbol;
      Push(s,e);
      p=p->first;
    }
    e.p=p;
    e.ch=p->symbol;
    Push(s,e);
    Vi(*(p->infoptr));
    i++;
    while(!StackEmpty(s))
    {
      Pop(s,e);
      p=e.p;
      if(p->next) // 有兄弟结点
      {
        p=p->next;
        while(p->kind==BRANCH) // 分支结点
        {
          e.p=p;
          e.ch=p->symbol;
          Push(s,e);
          p=p->first;
        }
        e.p=p;
        e.ch=p->symbol;
        Push(s,e);
        Vi(*(p->infoptr));
        i++;
        if(i%n==0)
          printf("\n"); // 输出n个元素后换行
      }
    }
  }
}

// algo9-7.cpp 检验bo9-5.cpp的程序
#include"c1.h"
```

```
#define N 16 // 数据元素个数
struct Others // 记录的其它部分
{
  int ord;
};
#define Nil ' ' // 定义结束符为空格(与教科书不同)
#include"c9-4.h"
#include"bo9-5.cpp"
void print(Record e)
{
  int i;
  printf("(");
  for(i=0;i<e.key.num;i++)
    printf("%c",e.key.ch[i]);
  printf(",%d)",e.others.ord);
}
void main()
{
  DLTree t;
  int i;
  char s[MAX_KEY_LEN+1];
  KeysType k;
  Record *p;
  Record r[N]={{{"CAI"},1},{{"CAO"},2},{{"LI"},3},{{"LAN"},4},{{"CHA"},5},{{"CHANG"},6},
               {{"WEN"},7},{{"CHAO"},8},{{"YUN"},9},{{"YANG"},10},{{"LONG"},11},
               {{"WANG"},12},{{"ZHAO"},13},{{"LIU"},14},{{"WU"},15},{{"CHEN"},16}};
  // 数据元素(以教科书式9-24为例)
  InitDSTable(t);
  for(i=0;i<N;i++)
  {
    r[i].key.num=strlen(r[i].key.ch);
    p=SearchDLTree(t,r[i].key);
    if(!p) // t中不存在关键字为r[i].key的项
      InsertDSTable(t,&r[i]);
  }
  printf("按关键字符串的顺序遍历双链键树:\n");
  TraverseDSTable(t,print);
  printf("\n请输入待查找记录的关键字符串: ");
  scanf("%s",s);
  k.num=strlen(s);
  strcpy(k.ch,s);
  p=SearchDLTree(t,k);
  if(p)
    print(*p);
  else
    printf("没找到");
  printf("\n");
  DestroyDSTable(t);
}
```

程序运行结果(见图 9－20)：

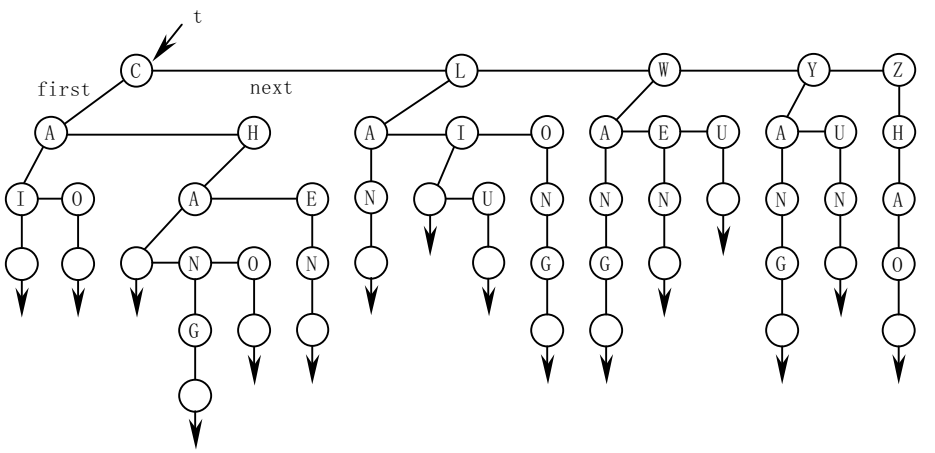


图 9－20　运行 algo9-7.cpp 生成的双链键树的结构

注：叶子结点的符号为空格，箭头指向含有相应关键字符串的记录

```
按关键字符串的顺序遍历双链键树:
(CAI,1) (CAO,2) (CHA,5) (CHANG,6) (CHAO,8) (CHEN,16) (LAN,4) (LI,3)
(LIU,14) (LONG,11) (WANG,12) (WEN,7) (WU,15) (YANG,10) (YUN,9) (ZHAO,13)

请输入待查找记录的关键字符串: WU↙
(WU,15)
```

```
// c9-5.h Trie键树类型
#define MAX_KEY_LEN 16 // 关键字的最大长度，同c9-4.h
struct KeysType // 关键字类型，同c9-4.h
{
  char ch[MAX_KEY_LEN]; // 关键字
  int num; // 关键字长度
};
struct Record // 记录类型，同c9-4.h
{
  KeysType key; // 关键字
  Others others; // 其它部分(由主程定义)
};
enum NodeKind{LEAF,BRANCH}; // 结点种类：{叶子,分支}，同c9-4.h
typedef struct TrieNode // Trie键树类型(见图9－21)
{
  NodeKind kind;
  union
  {
    struct // 叶子结点
    {
      KeysType K;
      Record *infoptr;
    }lf;
```

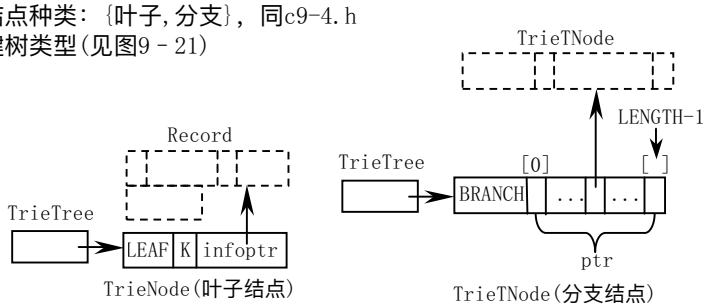


图 9－21　Trie 键树类型

```
    struct // 分支结点
    {
      TrieNode *ptr[LENGTH]; // LENGTH为结点的最大度+1，在主程定义
    //  int num; 改
    }bh;
  };
}TrieNode,*TrieTree;

// c9-8.h 对两个字符串型关键字的比较约定为如下的宏定义
#define EQ(a,b) (!strcmp((a),(b)))
#define LT(a,b) (strcmp((a),(b))<0)
#define LQ(a,b) (strcmp((a),(b))<=0)

// bo9-6.cpp 动态查找表(Trie键树)的基本操作，包括算法9.16
void InitDSTable(TrieTree &T)
{ // 操作结果：构造一个空的Trie键树T
  T=NULL;
}
void DestroyDSTable(TrieTree &T)
{ // 初始条件：Trie树T存在。操作结果：销毁Trie树T
  int i;
  if(T) // 非空树
  {
    for(i=0;i<LENGTH;i++)
      if(T->kind==BRANCH&&T->bh.ptr[i]) // 第i个结点不空
        if(T->bh.ptr[i]->kind==BRANCH) // 是子树
          DestroyDSTable(T->bh.ptr[i]);
        else // 是叶子
        {
          free(T->bh.ptr[i]);
          T->bh.ptr[i]=NULL;
        }
    free(T); // 释放根结点
    T=NULL; // 空指针赋0
  }
}
int ord(char c)
{
  c=toupper(c);
  if(c>='A'&&c<='Z')
    return c-'A'+1; // 英文字母返回其在字母表中的序号
  else
    return 0; // 其余字符返回0
}
Record *SearchTrie(TrieTree T,KeysType K)
{ // 在键树T中查找关键字等于K的记录。算法9.16
  TrieTree p;
  int i;
  for(p=T,i=0;p&&p->kind==BRANCH&&i<K.num;p=p->bh.ptr[ord(K.ch[i])],++i);
  // 对K的每个字符逐个查找，*p为分支结点，ord()求字符在字母表中序号
  if(p&&p->kind==LEAF&&p->lf.K.num==K.num&&EQ(p->lf.K.ch,K.ch)) // 查找成功
```

```
    return p->lf.infoptr;
  else // 查找不成功
    return NULL;
}
void InsertTrie(TrieTree &T,Record *r)
{ // 初始条件: Trie键树T存在, r为待插入的数据元素的指针
  // 操作结果: 若T中不存在其关键字等于(*r).key.ch的数据元素, 则按关键字顺序插r到T中
  TrieTree p,q,ap;
  int i=0,j;
  KeysType K1,K=r->key;
  if(!T) // 空树
  {
    T=(TrieTree)malloc(sizeof(TrieNode));
    T->kind=BRANCH;
    for(i=0;i<LENGTH;i++) // 指针量赋初值NULL
      T->bh.ptr[i]=NULL;
    p=T->bh.ptr[ord(K.ch[0])]=(TrieTree)malloc(sizeof(TrieNode));
    p->kind=LEAF;
    p->lf.K=K;
    p->lf.infoptr=r;
  }
  else // 非空树
  {
    for(p=T,i=0;p&&p->kind==BRANCH&&i<K.num;++i)
    {
      q=p;
      p=p->bh.ptr[ord(K.ch[i])];
    }
    i--;
    if(p&&p->kind==LEAF&&p->lf.K.num==K.num&&EQ(p->lf.K.ch,K.ch)) // T中存在该关键字
      return;
    else // T中不存在该关键字, 插入之
      if(!p) // 分支空
      {
        p=q->bh.ptr[ord(K.ch[i])]=(TrieTree)malloc(sizeof(TrieNode));
        p->kind=LEAF;
        p->lf.K=K;
        p->lf.infoptr=r;
      }
      else if(p->kind==LEAF) // 有不完全相同的叶子
      {
        K1=p->lf.K;
        do
        {
          ap=q->bh.ptr[ord(K.ch[i])]=(TrieTree)malloc(sizeof(TrieNode));
          ap->kind=BRANCH;
          for(j=0;j<LENGTH;j++) // 指针量赋初值NULL
            ap->bh.ptr[j]=NULL;
          q=ap;
          i++;
        }while(ord(K.ch[i])==ord(K1.ch[i]));
```

```
        q->bh.ptr[ord(K1.ch[i])]=p;
        p=q->bh.ptr[ord(K.ch[i])]=(TrieTree)malloc(sizeof(TrieNode));
        p->kind=LEAF;
        p->lf.K=K;
        p->lf.infoptr=r;
      }
  }
}
void TraverseDSTable(TrieTree T,void(*Vi)(Record*))
{ // 初始条件: Trie键树T存在, Vi是对记录指针操作的应用函数
  // 操作结果: 按关键字的顺序输出关键字及其对应的记录
  TrieTree p;
  int i;
  if(T)
    for(i=0;i<LENGTH;i++)
    {
      p=T->bh.ptr[i];
      if(p&&p->kind==LEAF)
        Vi(p->lf.infoptr);
      else if(p&&p->kind==BRANCH)
        TraverseDSTable(p,Vi);
    }
}

// algo9-8.cpp 检验bo9-6.cpp的程序
#include"c1.h"
#define N 16 // 数据元素个数
#define LENGTH 27 // 结点的最大度+1(大写英文字母)
struct Others // 记录的其它部分
{
  int ord;
};
#define Nil ' ' // 定义结束符为空格(与教科书不同)
#include"c9-5.h"
#include"c9-8.h"
#include"bo9-6.cpp"
void pr(Record *r)
{
  printf("(%s,%d)",r->key.ch,r->others.ord);
}
void main()
{
  TrieTree t;
  int i;
  char s[MAX_KEY_LEN+1];
  KeysType k;
  Record *p;
  Record r[N]={{{"CAI"},1},{{"CAO"},2},{{"LI"},3},{{"LAN"},4},{{"CHA"},5},{{"CHANG"},6},
               {{"WEN"},7},{{"CHAO"},8},{{"YUN"},9},{{"YANG"},10},{{"LONG"},11},
               {{"WANG"},12},{{"ZHAO"},13},{{"LIU"},14},{{"WU"},15},{{"CHEN"},16}};
  // 数据元素(以教科书式9-24为例)
```

```
  InitDSTable(t);
  for(i=0;i<N;i++)
  {
    r[i].key.num=strlen(r[i].key.ch)+1;
    r[i].key.ch[r[i].key.num]=Nil; // 在关键字符串最后加结束符
    p=SearchTrie(t,r[i].key);
    if(!p)
      InsertTrie(t,&r[i]);
  }
  printf("按关键字符串的顺序遍历Trie树(键树):\n");
  TraverseDSTable(t,pr);
  printf("\n请输入待查找记录的关键字符串: ");
  scanf("%s",s);
  k.num=strlen(s)+1;
  strcpy(k.ch,s);
  k.ch[k.num]=Nil; // 在关键字符串最后加结束符
  p=SearchTrie(t,k);
  if(p)
    pr(p);
  else
    printf("没找到");
  printf("\n");
  DestroyDSTable(t);
}
```

程序运行结果:

```
按关键字符串的顺序遍历Trie树(键树):
(CAI,1)(CAO,2)(CHA,5)(CHANG,6)(CHAO,8)(CHEN,16)(LAN,4)(LI,3)(LIU,14)(LONG,11)(WANG,12)
(WEN,7)(WU,15)(YANG,10)(YUN,9)(ZHAO,13)
请输入待查找记录的关键字符串: WU↙
(WU,15)
```

9.3 哈希表

9.3.1 什么是哈希表

9.3.2 哈希函数的构造方法

9.3.3 处理冲突的方法

由于哈希函数是杂凑函数，它有很大的随意性，有可能在一个哈希地址上分配多个数据，这种情况称为冲突。处理冲突，就是把原本分配到一个哈希地址上的多个数据，根据某种原则分配到不同的地址上。

9.3.4　哈希表的查找及其分析

```
// c9-6.h 开放定址哈希表的存储结构
int hashsize[]={11,19,29,37}; // 哈希表容量递增表，一个合适的素数序列
int m=0; // 哈希表表长，全局变量
struct HashTable(见图9－22)
{
  ElemType *elem; // 数据元素存储基址，动态分配数组
  int count; // 当前数据元素个数
  int sizeindex; // hashsize[sizeindex]为当前容量
};
#define SUCCESS 1
#define UNSUCCESS 0
#define DUPLICATE -1
```

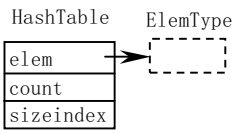


图 9－22　开放定址哈希表存储结构

```
// bo9-7.cpp 哈希函数的基本操作，包括算法9.17、9.18
void InitHashTable(HashTable &H)
{ // 操作结果：构造一个空的哈希表(见图9－23)
  int i;
  H.count=0; // 当前元素个数为0
  H.sizeindex=0; // 初始存储容量为hashsize[0]
  m=hashsize[0];
  H.elem=(ElemType*)malloc(m*sizeof(ElemType));
  if(!H.elem)
    exit(OVERFLOW); // 存储分配失败
  for(i=0;i<m;i++)
    H.elem[i].key=NULL_KEY; // 未填记录的标志
}
void DestroyHashTable(HashTable &H)
{ // 初始条件：哈希表H存在。操作结果：销毁哈希表H
  free(H.elem);
  H.elem=NULL;
  H.count=0;
  H.sizeindex=0;
}
unsigned Hash(KeyType K)
{ // 一个简单的哈希函数(m为表长，全局变量)
  return K%m;
}
void collision(int &p,int d) // 线性探测再散列
{ // 开放定址法处理冲突
  p=(p+d)%m;
}
Status SearchHash(HashTable H,KeyType K,int &p,int &c)
{ // 在开放定址哈希表H中查找关键码为K的元素，若查找成功，以p指示待查数据
  // 元素在表中位置，并返回SUCCESS；否则，以p指示插入位置，并返回UNSUCCESS
  // c用以计冲突次数，其初值置零，供建表插入时参考。算法9.17
  p=Hash(K); // 求得哈希地址
  while(H.elem[p].key!=NULL_KEY&&!EQ(K,H.elem[p].key))
  { // 该位置中填有记录．并且关键字不相等
    c++;
```

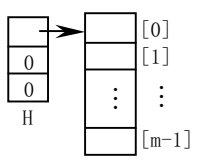


图 9－23　空的哈希表 H

```
    if(c<m)
      collision(p,c); // 求得下一探查地址p
    else
      break;
  }
  if EQ(K,H.elem[p].key)
    return SUCCESS; // 查找成功，p返回待查数据元素位置
  else
    return UNSUCCESS; // 查找不成功(H.elem[p].key==NULL_KEY)，p返回的是插入位置
}
Status InsertHash(HashTable &,ElemType); // 对函数的声明
void RecreateHashTable(HashTable &H)
{ // 重建哈希表
  int i,count=H.count;
  ElemType *p,*elem=(ElemType*)malloc(count*sizeof(ElemType));
  p=elem;
  printf("重建哈希表\n");
  for(i=0;i<m;i++) // 保存原有的数据到elem中
    if((H.elem+i)->key!=NULL_KEY) // 该单元有数据
      *p++=*(H.elem+i);
  H.count=0;
  H.sizeindex++; // 增大存储容量
  m=hashsize[H.sizeindex];
  p=(ElemType*)realloc(H.elem,m*sizeof(ElemType));
  if(!p)
    exit(OVERFLOW); // 存储分配失败
  H.elem=p;
  for(i=0;i<m;i++)
    H.elem[i].key=NULL_KEY; // 未填记录的标志(初始化)
  for(p=elem;p<elem+count;p++) // 将原有的数据按照新的表长插入到重建的哈希表中
    InsertHash(H,*p);
}
Status InsertHash(HashTable &H,ElemType e)
{ // 查找不成功时插入数据元素e到开放定址哈希表H中，并返回OK；
  // 若冲突次数过大，则重建哈希表，算法9.18
  int c,p;
  c=0;
  if(SearchHash(H,e.key,p,c)) // 表中已有与e有相同关键字的元素
    return DUPLICATE;
  else if(c<hashsize[H.sizeindex]/2) // 冲突次数c未达到上限，(c的阀值可调)
  { // 插入e
    H.elem[p]=e;
    ++H.count;
    return OK;
  }
  else
  {
    RecreateHashTable(H); // 重建哈希表
    return UNSUCCESS;
  }
}
```

```
void TraverseHash(HashTable H,void(*Vi)(int,ElemType))
{ // 按哈希地址的顺序遍历哈希表
  printf("哈希地址0～%d\n",m-1);
  for(int i=0;i<m;i++)
    if(H.elem[i].key!=NULL_KEY) // 有数据
      Vi(i,H.elem[i]);
}
Status Find(HashTable H,KeyType K,int &p)
{ // 在开放定址哈希表H中查找关键码为K的元素，若查找成功，以p指示待查数据
  // 元素在表中位置，并返回SUCCESS；否则，返回UNSUCCESS
  int c=0;
  p=Hash(K); // 求得哈希地址
  while(H.elem[p].key!=NULL_KEY&&!EQ(K,H.elem[p].key))
  { // 该位置中填有记录．并且关键字不相等
    c++;
    if(c<m)
      collision(p,c); // 求得下一探查地址p
    else
    return UNSUCCESS; // 查找不成功(H.elem[p].key==NULL_KEY)
  }
  if EQ(K,H.elem[p].key)
    return SUCCESS; // 查找成功，p返回待查数据元素位置
  else
    return UNSUCCESS; // 查找不成功(H.elem[p].key==NULL_KEY)
}


// algo9-9.cpp 检验bo9-7.cpp的程序
#include"c1.h"
#define NULL_KEY 0 // 0为无记录标志
#define N 10 // 数据元素个数
typedef int KeyType; // 设关键字域为整型
struct ElemType // 数据元素类型
{
  KeyType key;
  int ord;
};
#include"c9-7.h"
#include"c9-6.h"
#include"bo9-7.cpp"
void print(int p,ElemType r)
{
  printf("address=%d (%d,%d)\n",p,r.key,r.ord);
}
void main()
{
  ElemType r[N]={{17,1},{60,2},{29,3},{38,4},{1,5},{2,6},{3,7},{4,8},{60,9},{13,10}};
  HashTable h;
  int i,p;
  Status j;
  KeyType k;
  InitHashTable(h);
```

```
  for(i=0;i<N-1;i++)
  { // 插入前N-1个记录
    j=InsertHash(h,r[i]);
    if(j==DUPLICATE)
      printf("表中已有关键字为%d的记录，无法再插入记录(%d,%d)\n",r[i].key,r[i].key,
      r[i].ord);
  }
  printf("按哈希地址的顺序遍历哈希表:\n");
  TraverseHash(h,print);
  printf("请输入待查找记录的关键字: ");
  scanf("%d",&k);
  j=Find(h,k,p);
  if(j==SUCCESS)
    print(p,h.elem[p]);
  else
    printf("没找到\n");
  j=InsertHash(h,r[i]); // 插入第N个记录
  if(j==ERROR) // 重建哈希表
    j=InsertHash(h,r[i]); // 重建哈希表后重新插入
  printf("按哈希地址的顺序遍历重建后的哈希表:\n");
  TraverseHash(h,print);
  printf("请输入待查找记录的关键字: ");
  scanf("%d",&k);
  j=Find(h,k,p);
  if(j==SUCCESS)
    print(p,h.elem[p]);
  else
    printf("没找到\n");
  DestroyHashTable(h);
}
```

程序运行结果(以教科书中图 9.25 为例):

```
表中已有关键字为60的记录，无法再插入记录(60,9)
按哈希地址的顺序遍历哈希表:
哈希地址0～10
address=1 (1,5)
address=2 (2,6)
address=3 (3,7)
address=4 (4,8)
address=5 (60,2)
address=6 (17,1)
address=7 (29,3)
address=8 (38,4)
请输入待查找记录的关键字: 13↙
没找到
重建哈希表
按哈希地址的顺序遍历重建后的哈希表:
哈希地址0～18
```

```
address=0 (38,4)
address=1 (1,5)
address=2 (2,6)
address=3 (3,7)
address=4 (4,8)
address=6 (60,2)
address=10 (29,3)
address=13 (13,10)
address=17 (17,1)
请输入待查找记录的关键字: 13↙
address=13 (13,10)
```

第 10 章　内 部 排 序

10.1　概述

排序就是把一组数据按关键字的大小有规律地排列。经过排序的数据更易于查找。所谓内部排序，就是先把待排序数据都放到内存中，再进行排序。

```
// c10-1.h 待排记录的数据类型
#define MAX_SIZE 20 // 一个用作示例的小顺序表的最大长度
typedef int KeyType; // 定义关键字类型为整型
struct RedType // 记录类型(见图10－1)
{
  KeyType key; // 关键字项
  InfoType otherinfo; // 其它数据项，具体类型在主程中定义
};
struct SqList // 顺序表类型(见图10－2)
{
  RedType r[MAX_SIZE+1]; // r[0]闲置或用作哨兵单元
  int length; // 顺序表长度
};
```

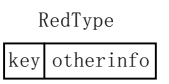


图 10－1　记录类型

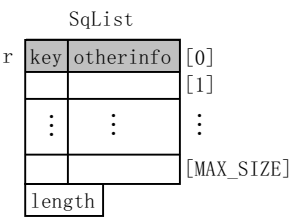


图 10－2　顺序表类型

10.2　插入排序

10.2.1　直接插入排序

```
// bo10-1.cpp 顺序表插入排序的函数(3个)，包括算法10.1，10.2
void InsertSort(SqList &L)
{ // 对顺序表L作直接插入排序。算法10.1
  int i,j;
  for(i=2;i<=L.length;++i)
    if LT(L.r[i].key,L.r[i-1].key) // "<",需将L.r[i]插入有序子表
    {
      L.r[0]=L.r[i]; // 复制为哨兵
      for(j=i-1;LT(L.r[0].key,L.r[j].key);--j)
        L.r[j+1]=L.r[j]; // 记录后移
      L.r[j+1]=L.r[0]; // 插入到正确位置
    }
}
```

```
void BInsertSort(SqList &L)
{ // 对顺序表L作折半插入排序。算法10.2
  int i,j,m,low,high;
  for(i=2;i<=L.length;++i)
  {
    L.r[0]=L.r[i]; // 将L.r[i]暂存到L.r[0]
    low=1;
    high=i-1;
    while(low<=high)
    { // 在r[low..high]中折半查找有序插入的位置
      m=(low+high)/2; // 折半
      if LT(L.r[0].key,L.r[m].key)
        high=m-1; // 插入点在低半区
      else
        low=m+1; // 插入点在高半区
    }
    for(j=i-1;j>=high+1;--j)
      L.r[j+1]=L.r[j]; // 记录后移
    L.r[high+1]=L.r[0]; // 插入
  }
}
void P2_InsertSort(SqList &L)
{ // 2_路插入排序
  int i,j,first,final;
  RedType *d;
  d=(RedType*)malloc(L.length*sizeof(RedType)); // 生成L.length个记录的临时空间
  d[0]=L.r[1]; // 设L的第1个记录为d中排好序的记录(在位置[0])
  first=final=0; // first、final分别指示d中排好序的记录的第1个和最后1个记录的位置
  for(i=2;i<=L.length;++i) // 依次将L的第2个～最后1个记录插入d中
    if(L.r[i].key<d[first].key)
    { // 待插记录小于d中最小值，插到d[first]之前(不需移动d数组的元素)
      first=(first-1+L.length)%L.length; // 设d为循环向量
      d[first]=L.r[i];
    }
    else if(L.r[i].key>d[final].key)
    { // 待插记录大于d中最大值，插到d[final]之后(不需移动d数组的元素)
      final=final+1;
      d[final]=L.r[i];
    }
    else
    { // 待插记录大于d中最小值，小于d中最大值，插到d的中间(需要移动d数组的元素)
      j=final++; // 移动d的尾部元素以便按序插入记录
      while(L.r[i].key<d[j].key)
      {
        d[(j+1)%L.length]=d[j];
        j=(j-1+L.length)%L.length;
      }
      d[j+1]=L.r[i];
    }
  for(i=1;i<=L.length;i++) // 把d赋给L.r
    L.r[i]=d[(i+first-1)%L.length]; // 线性关系
```

```
}

// algo10-1.cpp 检验bo10-1.cpp的程序
#include"c1.h"
typedef int InfoType; // 定义其它数据项的类型
#include"c9-7.h"
#include"c10-1.h"
#include"bo10-1.cpp"
void print(SqList L)
{
  int i;
  for(i=1;i<=L.length;i++)
    printf("(%d,%d)",L.r[i].key,L.r[i].otherinfo);
  printf("\n");
}
#define N 8
void main()
{
  RedType d[N]={{49,1},{38,2},{65,3},{97,4},{76,5},{13,6},{27,7},{49,8}};
  SqList l1,l2,l3;
  int i;
  for(i=0;i<N;i++) // 给l1.r赋值
    l1.r[i+1]=d[i];
  l1.length=N;
  l2=l3=l1; // 复制顺序表l2、l3与l1相同
  printf("排序前:\n");
  print(l1);
  InsertSort(l1);
  printf("直接插入排序后:\n");
  print(l1);
  BInsertSort(l2);
  printf("折半插入排序后:\n");
  print(l2);
  P2_InsertSort(l3);
  printf("2_路插入排序后:\n");
  print(l3);
}
```

程序运行结果(以教科书中式 10－4 的数据为例):

```
排序前:
(49,1)(38,2)(65,3)(97,4)(76,5)(13,6)(27,7)(49,8)
直接插入排序后:
(13,6)(27,7)(38,2)(49,1)(49,8)(65,3)(76,5)(97,4)
折半插入排序后:
(13,6)(27,7)(38,2)(49,1)(49,8)(65,3)(76,5)(97,4)
2_路插入排序后:
(13,6)(27,7)(38,2)(49,1)(49,8)(65,3)(76,5)(97,4)
```

10.2.2　其它插入排序

折半插入排序和 2_路插入排序在 bo10-1.cpp 中。

```
// c10-2.h 静态链表类型
#define SIZE 100 // 静态链表容量
typedef int KeyType; // 定义关键字类型为整型
struct RedType // 记录类型(见图10－1)
{
  KeyType key; // 关键字项
  InfoType otherinfo; // 其它数据项，具体类型在主程中定义
};
struct SLNode // 表结点类型(见图10－3)
{
  RedType rc; // 记录项
  int next; // 指针项
};
struct SLinkListType // 静态链表类型(见图10－4)
{
  SLNode r[SIZE]; // 0号单元为表头结点
  int length; // 链表当前长度
};
```

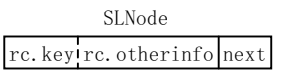


图 10－3　表结点类型

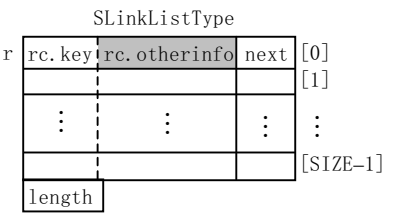


图 10－4　静态链表类型

```
// algo10-2.cpp 静态链表，包括算法10.3，10.18
#include"c1.h"
typedef int InfoType; // 定义其它数据项的类型
#include"c10-2.h"
void TableInsert(SLinkListType &SL,RedType D[],int n)
{ // 由数组D建立n个元素的表插入排序的静态链表SL
  int i,p,q;
  SL.r[0].rc.key=INT_MAX; // 表头结点记录的关键字取最大整数(非降序链表的表尾)
  SL.r[0].next=0; // next域为0表示表尾(现为空表，初始化)
  for(i=0;i<n;i++)
  {
    SL.r[i+1].rc=D[i]; // 将数组D的值赋给静态链表SL
    q=0;
    p=SL.r[0].next;
    while(SL.r[p].rc.key<=SL.r[i+1].rc.key)
    { // 静态链表向后移
      q=p;
      p=SL.r[p].next;
    }
    SL.r[i+1].next=p; // 将当前记录插入静态链表
    SL.r[q].next=i+1;
  }
  SL.length=n;
}
void Arrange(SLinkListType &SL) // 算法10.3
{ // 根据静态链表SL中各结点的指针值调整记录位置，使得SL中记录按关键字非递减有序顺序排列
  int i,p,q;
```

```
  SLNode t;
  p=SL.r[0].next; // p指示第一个记录的当前位置
  for(i=1;i<SL.length;++i)
  { // SL.r[1..i-1]中记录已按关键字有序排列,第i个记录在SL中的当前位置应不小于i
    while(p<i)
      p=SL.r[p].next; // 找到第i个记录，并用p指示其在SL中当前位置
    q=SL.r[p].next; // q指示尚未调整的表尾
    if(p!=i)
    {
      t=SL.r[p]; // 交换记录，使第i个记录到位
      SL.r[p]=SL.r[i];
      SL.r[i]=t;
      SL.r[i].next=p; // 指向被移走的记录，使得以后可由while循环找回
    }
    p=q; // p指示尚未调整的表尾，为找第i+1个记录作准备
  }
}
void Sort(SLinkListType L,int adr[])
{ // 求得adr[1..L.length]，adr[i]为静态链表L的第i个最小记录的序号
  int i=1,p=L.r[0].next;
  while(p)
  {
    adr[i++]=p;
    p=L.r[p].next;
  }
}
void Rearrange(SLinkListType &L,int adr[])
{ // adr给出静态链表L的有序次序，即L.r[adr[i]]是第i小的记录。
  // 本算法按adr重排L.r，使其有序。算法10.18(L的类型有变)
  int i,j,k;
  for(i=1;i<L.length;++i)
    if(adr[i]!=i)
    {
      j=i;
      L.r[0]=L.r[i]; // 暂存记录L.r[i]
      while(adr[j]!=i)
      { // 调整L.r[adr[j]]的记录到位直到adr[j]=i为止
        k=adr[j];
        L.r[j]=L.r[k];
        adr[j]=j;
        j=k; // 记录按序到位
      }
      L.r[j]=L.r[0];
      adr[j]=j;
    }
}
void print(SLinkListType L)
{
  int i;
  for(i=1;i<=L.length;i++)
    printf("key=%d ord=%d next=%d\n",L.r[i].rc.key,L.r[i].rc.otherinfo,L.r[i].next);
```

```
}
#define N 8
void main()
{
  RedType d[N]={{49,1},{38,2},{65,3},{97,4},{76,5},{13,6},{27,7},{49,8}};
  SLinkListType l1,l2;
  int *adr,i;
  TableInsert(l1,d,N);
  l2=l1; // 复制静态链表l2与l1相同
  printf("排序前:\n");
  print(l1);
  Arrange(l1);
  printf("l1排序后:\n");
  print(l1);
  adr=(int*)malloc((l2.length+1)*sizeof(int));
  Sort(l2,adr);
  for(i=1;i<=l2.length;i++)
    printf("adr[%d]=%d ",i,adr[i]);
  printf("\n");
  Rearrange(l2,adr);
  printf("l2排序后:\n");
  print(l2);
}
```

程序运行结果(以教科书中图 10.3 和图 10.4 的数据为例):

```
排序前:
key=49 ord=1 next=8
key=38 ord=2 next=1
key=65 ord=3 next=5
key=97 ord=4 next=0
key=76 ord=5 next=4
key=13 ord=6 next=7
key=27 ord=7 next=2
key=49 ord=8 next=3
l1排序后:
key=13 ord=6 next=6
key=27 ord=7 next=7
key=38 ord=2 next=7
key=49 ord=1 next=6
key=49 ord=8 next=8
key=65 ord=3 next=7
key=76 ord=5 next=8
key=97 ord=4 next=0
adr[1]=6 adr[2]=7 adr[3]=2 adr[4]=1 adr[5]=8 adr[6]=3 adr[7]=5 adr[8]=4
l2排序后:
key=13 ord=6 next=7
key=27 ord=7 next=2
key=38 ord=2 next=1
```

```
key=49 ord=1 next=8
key=49 ord=8 next=3
key=65 ord=3 next=5
key=76 ord=5 next=4
key=97 ord=4 next=0
```

10.2.3　希尔排序

```
// algo10-3.cpp 希尔排序，包括算法10.4，10.5
#include<stdio.h>
typedef int InfoType; // 定义其它数据项的类型
#include"c9-7.h"
#include"c10-1.h"
void ShellInsert(SqList &L,int dk)
{ // 对顺序表L作一趟希尔插入排序。本算法是和一趟直接插入排序相比，作了以下修改:
  // 1.前后记录位置的增量是dk，而不是1;
  // 2.r[0]只是暂存单元，不是哨兵。当j<=0时，插入位置已找到。算法10.4
  int i,j;
  for(i=dk+1;i<=L.length;++i)
    if LT(L.r[i].key,L.r[i-dk].key)
    { // 需将L.r[i]插入有序增量子表
      L.r[0]=L.r[i]; // 暂存在L.r[0]
      for(j=i-dk;j>0&&LT(L.r[0].key,L.r[j].key);j-=dk)
        L.r[j+dk]=L.r[j]; // 记录后移，查找插入位置
      L.r[j+dk]=L.r[0]; // 插入
    }
}
void print(SqList L)
{
  int i;
  for(i=1;i<=L.length;i++)
    printf("%d ",L.r[i].key);
  printf("\n");
}
void print1(SqList L)
{
  int i;
  for(i=1;i<=L.length;i++)
    printf("(%d,%d)",L.r[i].key,L.r[i].otherinfo);
  printf("\n");
}
void ShellSort(SqList &L,int dlta[],int t)
{ // 按增量序列dlta[0..t-1]对顺序表L作希尔排序。算法10.5
  int k;
  for(k=0;k<t;++k)
  {
    ShellInsert(L,dlta[k]); // 一趟增量为dlta[k]的插入排序
    printf("第%d趟排序结果: ",k+1);
    print(L);
  }
}
```

```
#define N 10
#define T 3
void main()
{
  RedType d[N]={{49,1},{38,2},{65,3},{97,4},{76,5},{13,6},{27,7},{49,8},{55,9},{4,10}};
  SqList l;
  int dt[T]={5,3,1}; // 增量序列数组
  for(int i=0;i<N;i++)
    l.r[i+1]=d[i];
  l.length=N;
  printf("排序前: ");
  print(l);
  ShellSort(l,dt,T);
  printf("排序后: ");
  print1(l);
}
```

程序运行结果(以教科书中图 10.5 的数据为例):

```
排序前: 49 38 65 97 76 13 27 49 55 4
第1趟排序结果: 13 27 49 55 4 49 38 65 97 76
第2趟排序结果: 13 4 49 38 27 49 55 65 97 76
第3趟排序结果: 4 13 27 38 49 49 55 65 76 97
排序后: (4,10)(13,6)(27,7)(38,2)(49,8)(49,1)(55,9)(65,3)(76,5)(97,4)
```

10.3　快速排序

```
// algo10-4.cpp 调用起泡排序(在教科书1.4.3节算法效率的度量中)的程序
#include"c1.h"
#define N 8
void bubble_sort(int a[],int n)
{ // 将a中整数序列重新排列成自小至大有序的整数序列(起泡排序)
  int i,j,t;
  Status change;
  for(i=n-1,change=TRUE;i>1&&change;--i)
  {
    change=FALSE;
    for(j=0;j<i;++j)
      if(a[j]>a[j+1])
      {
        t=a[j];
        a[j]=a[j+1];
        a[j+1]=t;
        change=TRUE;
      }
  }
}
void print(int r[],int n)
```

```
{
  int i;
  for(i=0;i<n;i++)
    printf("%d ",r[i]);
  printf("\n");
}
void main()
{
  int d[N]={49,38,65,97,76,13,27,49};
  printf("排序前:\n");
  print(d,N);
  bubble_sort(d,N);
  printf("排序后:\n");
  print(d,N);
}
```

程序运行结果(以教科书中图 10.6 的数据为例):

```
排序前:
49 38 65 97 76 13 27 49
排序后:
13 27 38 49 49 65 76 97
```

```
// bo10-2.cpp 快速排序的函数，包括算法10.7，10.8
void QSort(SqList &L,int low,int high)
{ // 对顺序表L中的子序列L.r[low..high]作快速排序。算法10.7
  int pivotloc;
  if(low<high)
  { // 长度大于1
    pivotloc=Partition(L,low,high); // 将L.r[low..high]一分为二
    QSort(L,low,pivotloc-1); // 对低子表递归排序，pivotloc是枢轴位置
    QSort(L,pivotloc+1,high); // 对高子表递归排序
  }
}
void QuickSort(SqList &L)
{ // 对顺序表L作快速排序。算法10.8
  QSort(L,1,L.length);
}
void print(SqList L)
{
  int i;
  for(i=1;i<=L.length;i++)
    printf("(%d,%d)",L.r[i].key,L.r[i].otherinfo);
  printf("\n");
}

// algo10-5.cpp 调用算法10.6(a)的程序
#include<stdio.h>
typedef int InfoType; // 定义其它数据项的类型
```

```
#include"c10-1.h"
int Partition(SqList &L,int low,int high)
{ // 交换顺序表L中子表L.r[low..high]的记录，使枢轴记录到位，
  // 并返回其所在位置，此时在它之前(后)的记录均不大(小)于它。算法10.6(a)
  RedType t;
  KeyType pivotkey;
  pivotkey=L.r[low].key; // 用子表的第一个记录作枢轴记录
  while(low<high)
  { // 从表的两端交替地向中间扫描
    while(low<high&&L.r[high].key>=pivotkey)
      --high;
    t=L.r[low]; // 将比枢轴记录小的记录交换到低端
    L.r[low]=L.r[high];
    L.r[high]=t;
    while(low<high&&L.r[low].key<=pivotkey)
      ++low;
    t=L.r[low]; // 将比枢轴记录大的记录交换到高端
    L.r[low]=L.r[high];
    L.r[high]=t;
  }
  return low; // 返回枢轴所在位置
}
#include"bo10-2.cpp"
#define N 8
void main()
{
  RedType d[N]={{49,1},{38,2},{65,3},{97,4},{76,5},{13,6},{27,7},{49,8}};
  SqList l;
  int i;
  for(i=0;i<N;i++)
    l.r[i+1]=d[i];
  l.length=N;
  printf("排序前:\n");
  print(l);
  QuickSort(l);
  printf("排序后:\n");
  print(l);
}
```

 　程序运行结果(以教科书中图 10.7 的数据为例)：

```
排序前:
(49,1)(38,2)(65,3)(97,4)(76,5)(13,6)(27,7)(49,8)
排序后:
(13,6)(27,7)(38,2)(49,1)(49,8)(65,3)(76,5)(97,4)
```

```
// algo10-6.cpp 调用算法10.6(b)的程序(算法10.6(a)的改进)
#include<stdio.h>
typedef int InfoType; // 定义其它数据项的类型
#include"c10-1.h"
```

```
int Partition(SqList &L,int low,int high)
{ // 交换顺序表L中子表r[low..high]的记录，枢轴记录到位，并返回其
  // 所在位置，此时在它之前(后)的记录均不大(小)于它。算法10.6(b)
  KeyType pivotkey;
  L.r[0]=L.r[low]; // 用子表的第一个记录作枢轴记录
  pivotkey=L.r[low].key; // 枢轴记录关键字
  while(low< high)
  { // 从表的两端交替地向中间扫描
    while(low<high&&L.r[high].key>=pivotkey)
      --high;
    L.r[low]=L.r[high]; // 将比枢轴记录小的记录移到低端
    while(low<high&&L.r[low].key<=pivotkey)
      ++low;
    L.r[high]=L.r[low]; // 将比枢轴记录大的记录移到高端
  }
  L.r[low]=L.r[0]; // 枢轴记录到位
  return low; // 返回枢轴位置
}
#include"bo10-2.cpp"
#define N 8
void main()
{
  RedType d[N]={{49,1},{38,2},{65,3},{97,4},{76,5},{13,6},{27,7},{49,8}};
  SqList l;
  int i;
  for(i=0;i<N;i++)
    l.r[i+1]=d[i];
  l.length=N;
  printf("排序前:\n");
  print(l);
  QuickSort(l);
  printf("排序后:\n");
  print(l);
}
```

 　程序运行结果（以教科书中图 10.7 的数据为例）：

```
排序前:
(49,1)(38,2)(65,3)(97,4)(76,5)(13,6)(27,7)(49,8)
排序后:
(13,6)(27,7)(38,2)(49,1)(49,8)(65,3)(76,5)(97,4)
```

10.4　选择排序

10.4.1　简单选择排序

```
// algo10-7.cpp 简单选择排序，包括算法10.9
#include<stdio.h>
typedef int InfoType; // 定义其它数据项的类型
```

```
#include"c10-1.h"
int SelectMinKey(SqList L,int i)
{ // 返回在L.r[i..L.length]中key最小的记录的序号
  KeyType min;
  int j,k;
  k=i; // 设第i个为最小
  min=L.r[i].key;
  for(j=i+1;j<=L.length;j++)
    if(L.r[j].key<min) // 找到更小的
    {
      k=j;
      min=L.r[j].key;
    }
  return k;
}
void SelectSort(SqList &L)
{ // 对顺序表L作简单选择排序。算法10.9
  int i,j;
  RedType t;
  for(i=1;i<L.length;++i)
  { //  选择第i小的记录，并交换到位
    j=SelectMinKey(L,i); // 在L.r[i..L.length]中选择key最小的记录
    if(i!=j)
    { // 与第i个记录交换
      t=L.r[i];
      L.r[i]=L.r[j];
      L.r[j]=t;
    }
  }
}
void print(SqList L)
{
  int i;
  for(i=1;i<=L.length;i++)
    printf("(%d,%d)",L.r[i].key,L.r[i].otherinfo);
  printf("\n");
}
#define N 8
void main()
{
  RedType d[N]={{49,1},{38,2},{65,3},{97,4},{76,5},{13,6},{27,7},{49,8}};
  SqList l;
  int i;
  for(i=0;i<N;i++)
    l.r[i+1]=d[i];
  l.length=N;
  printf("排序前:\n");
  print(l);
  SelectSort(l);
  printf("排序后:\n");
```

```
  print(l);
}
```

程序运行结果(以教科书中式 10－4 的数据为例):

```
排序前:
(49,1)(38,2)(65,3)(97,4)(76,5)(13,6)(27,7)(49,8)
排序后:
(13,6)(27,7)(38,2)(49,1)(49,8)(65,3)(76,5)(97,4)
```

10.4.2　树形选择排序

```
// algo10-8.cpp 树形选择排序
#include"c1.h"
typedef int InfoType; // 定义其它数据项的类型
#include"c10-1.h"
void TreeSort(SqList &L)
{ // 树形选择排序
  int i,j,j1,k,k1,l,n=L.length;
  RedType *t;
  l=(int)ceil(log(n)/log(2))+1; // 完全二叉树的层数
  k=(int)pow(2,l)-1; // l层完全二叉树的结点总数
  k1=(int)pow(2,l-1)-1; // l-1层完全二叉树的结点总数
  t=(RedType*)malloc(k*sizeof(RedType)); // 二叉树采用顺序存储结构
  for(i=1;i<=n;i++) // 将L.r赋给叶子结点
    t[k1+i-1]=L.r[i];
  for(i=k1+n;i<k;i++) // 给多余的叶子的关键字赋无穷大
    t[i].key=INT_MAX;
  j1=k1;
  j=k;
  while(j1)
  { // 给非叶子结点赋值
    for(i=j1;i<j;i+=2)
      t[i].key<t[i+1].key?(t[(i+1)/2-1]=t[i]):(t[(i+1)/2-1]=t[i+1]);
    j=j1;
    j1=(j1-1)/2;
  }
  for(i=0;i<n;i++)
  {
    L.r[i+1]=t[0]; // 将当前最小值赋给L.r[i]
    j1=0;
    for(j=1;j<l;j++) // 沿树根找结点t[0]在叶子中的序号j1
      t[2*j1+1].key==t[j1].key?(j1=2*j1+1):(j1=2*j1+2);
    t[j1].key=INT_MAX;
    while(j1)
    {
      j1=(j1+1)/2-1; // 序号为j1的结点的双亲结点序号
      t[2*j1+1].key<=t[2*j1+2].key?(t[j1]=t[2*j1+1]):(t[j1]=t[2*j1+2]);
```

```
    }
  }
  free(t);
}
void print(SqList L)
{
  int i;
  for(i=1;i<=L.length;i++)
    printf("(%d,%d)",L.r[i].key,L.r[i].otherinfo);
  printf("\n");
}
#define N 8
void main()
{
  RedType d[N]={{49,1},{38,2},{65,3},{97,4},{76,5},{13,6},{27,7},{49,8}};
  SqList l;
  int i;
  for(i=0;i<N;i++)
    l.r[i+1]=d[i];
  l.length=N;
  printf("排序前:\n");
  print(l);
  TreeSort(l);
  printf("排序后:\n");
  print(l);
}
```

 　程序运行结果(以教科书中图 10.9 的数据为例):

```
排序前:
(49,1)(38,2)(65,3)(97,4)(76,5)(13,6)(27,7)(49,8)
排序后:
(13,6)(27,7)(38,2)(49,1)(49,8)(65,3)(76,5)(97,4)
```

10.4.3　堆排序

堆排序是采用顺序结构存储待排序元素，并把这些元素看作一棵完全二叉树上的结点。这与 c6-1.h 定义的二叉树的顺序存储结构有相同之处。堆的定义是(以升序为例)：任何一个非叶子结点的关键字值都不大于它左右孩子结点的关键字值。所以，对于一个满足堆定义的顺序存储结构，虽然它的所有元素的排列并不是完全有序的，但堆顶元素(第 1 个元素)必定是完全排序后的第 1 个元素。堆排序的优点是：在调整顺序结构成为堆排序的过程中，由下到上，堆顶元素顶多只需与其左右孩子之一(3 元素中的最小值)作交换。同时如果堆顶元素被交换，则只有交换的影响波及到的小堆可能需要再作交换，且所有叶子结点(具有 n 个结点的完全二叉树只有不大于 n/2 个非叶子结点)没有孩子。这样就可用较少的比较、调整步骤，达到排序目的。

```
// algo10-9.cpp 堆排序，包括算法10.10，10.11
#include<stdio.h>
typedef int InfoType; // 定义其它数据项的类型
#include"c9-7.h"
#include"c10-1.h"
typedef SqList HeapType; // 堆采用顺序表存储表示
void HeapAdjust(HeapType &H,int s,int m) // 算法10.10
{ // 已知H.r[s..m]中记录的关键字除H.r[s].key之外均满足堆的定义，本函数
  // 调整H.r[s]的关键字，使H.r[s..m]成为一个大顶堆(对其中记录的关键字而言)
  RedType rc;
  int j;
  rc=H.r[s];
  for(j=2*s;j<=m;j*=2)
  { // 沿key较大的孩子结点向下筛选
    if(j<m&&LT(H.r[j].key,H.r[j+1].key))
      ++j; // j为key较大的记录的下标
    if(!LT(rc.key,H.r[j].key))
      break; // rc应插入在位置s上
    H.r[s]=H.r[j];
    s=j;
  }
  H.r[s]=rc; // 插入
}
void HeapSort(HeapType &H)
{ // 对顺序表H进行堆排序。算法10.11
  RedType t;
  int i;
  for(i=H.length/2;i>0;--i) // 把H.r[1..H.length]建成大顶堆
    HeapAdjust(H,i,H.length);
  for(i=H.length;i>1;--i)
  { // 将堆顶记录和当前未经排序子序列H.r[1..i]中最后一个记录相互交换
    t=H.r[1];
    H.r[1]=H.r[i];
    H.r[i]=t;
    HeapAdjust(H,1,i-1); // 将H.r[1..i-1]重新调整为大顶堆
  }
}
void print(HeapType H)
{
  int i;
  for(i=1;i<=H.length;i++)
    printf("(%d,%d)",H.r[i].key,H.r[i].otherinfo);
  printf("\n");
}
#define N 8
void main()
{
  RedType d[N]={{49,1},{38,2},{65,3},{97,4},{76,5},{13,6},{27,7},{49,8}};
  HeapType h;
  int i;
```

```
  for(i=0;i<N;i++)
    h.r[i+1]=d[i];
  h.length=N;
  printf("排序前:\n");
  print(h);
  HeapSort(h);
  printf("排序后:\n");
  print(h);
}
```

 　程序运行结果(以教科书中式 10－4 的数据为例):

```
排序前:
(49,1)(38,2)(65,3)(97,4)(76,5)(13,6)(27,7)(49,8)
排序后:
(13,6)(27,7)(38,2)(49,1)(49,8)(65,3)(76,5)(97,4)
```

10.5　归并排序

```
// alg10-10.cpp 归并排序，包括算法10.12～10.14
#include<stdio.h>
typedef int InfoType; // 定义其它数据项的类型
#include"c9-7.h"
#include"c10-1.h"
void Merge(RedType SR[],RedType TR[],int i,int m,int n)
{ // 将有序的SR[i..m]和SR[m+1..n]归并为有序的TR[i..n]。算法10.12
  int j,k,l;
  for(j=m+1,k=i;i<=m&&j<=n;++k) // 将SR中记录由小到大地并入TR
    if LQ(SR[i].key,SR[j].key)
      TR[k]=SR[i++];
    else
      TR[k]=SR[j++];
  if(i<=m)
    for(l=0;l<=m-i;l++)
      TR[k+l]=SR[i+l]; // 将剩余的SR[i..m]复制到TR
  if(j<=n)
    for(l=0;l<=n-j;l++)
      TR[k+l]=SR[j+l]; // 将剩余的SR[j..n]复制到TR
}
void MSort(RedType SR[],RedType TR1[],int s, int t)
{ // 将SR[s..t]归并排序为TR1[s..t]。算法10.13
  int m;
  RedType TR2[MAX_SIZE+1];
  if(s==t)
    TR1[s]=SR[s];
  else
  {
```

```
    m=(s+t)/2; // 将SR[s..t]平分为SR[s..m]和SR[m+1..t]
    MSort(SR,TR2,s,m); // 递归地将SR[s..m]归并为有序的TR2[s..m]
    MSort(SR,TR2,m+1,t); // 递归地将SR[m+1..t]归并为有序的TR2[m+1..t]
    Merge(TR2,TR1,s,m,t); // 将TR2[s..m]和TR2[m+1..t]归并到TR1[s..t]
  }
}
void MergeSort(SqList &L)
{ // 对顺序表L作归并排序。算法10.14
  MSort(L.r,L.r,1,L.length);
}
void print(SqList L)
{
  int i;
  for(i=1;i<=L.length;i++)
    printf("(%d,%d)",L.r[i].key,L.r[i].otherinfo);
  printf("\n");
}
#define N 7
void main()
{
  RedType d[N]={{49,1},{38,2},{65,3},{97,4},{76,5},{13,6},{27,7}};
  SqList l;
  int i;
  for(i=0;i<N;i++)
    l.r[i+1]=d[i];
  l.length=N;
  printf("排序前:\n");
  print(l);
  MergeSort(l);
  printf("排序后:\n");
  print(l);
}
```

 　程序运行结果(以教科书中图 10.13 的数据为例):

```
排序前:
(49,1)(38,2)(65,3)(97,4)(76,5)(13,6)(27,7)
排序后:
(13,6)(27,7)(38,2)(49,1)(65,3)(76,5)(97,4)
```

10.6　基数排序

10.6.1　多关键字的排序

10.6.2　链式基数排序

```
// c10-3.h 基数排序的数据类型
#define MAX_NUM_OF_KEY 8 // 关键字项数的最大值
```

```
#define RADIX 10 // 关键字基数，此时是十进制整数的基数
#define MAX_SPACE 1000
struct SLCell // 静态链表的结点类型(见图10－5)
{
  KeysType keys[MAX_NUM_OF_KEY]; // 关键字
  InfoType otheritems; // 其它数据项
  int next;
};
```

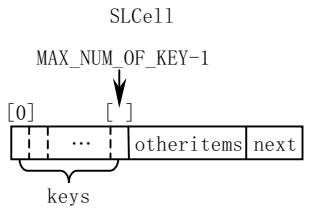


图 10－5　静态链表的结点类型

```
struct SLList // 静态链表类型(见图10－6)
{
  SLCell r[MAX_SPACE]; // 静态链表的可利用空间，r[0]为头结点
  int keynum; // 记录的当前关键字个数
  int recnum; //  静态链表的当前长度
};
typedef int ArrType[RADIX]; // 指针数组类型
```

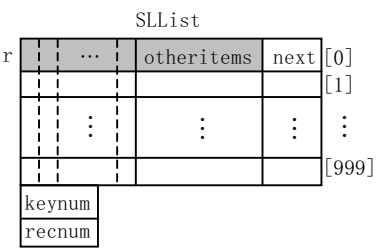


图 10－6　静态链表类型

```
// alg10-11.cpp 链式基数排序，包括算法10.15～10.17
typedef int InfoType; // 定义其它数据项的类型
typedef int KeyType; // 定义RedType类型的关键字为整型
struct RedType // 记录类型(同c10-1.h)
{
  KeyType key; // 关键字项
  InfoType otherinfo; // 其它数据项
};
typedef char KeysType; // 定义关键字类型为字符型
#include"c1.h"
#include"c10-3.h"
void InitList(SLList &L,RedType D[],int n)
{ // 初始化静态链表L(把数组D中的数据存于L中)
  char c[MAX_NUM_OF_KEY],c1[MAX_NUM_OF_KEY];
  int i,j,max=D[0].key; // max为关键字的最大值
  for(i=1;i<n;i++)
    if(max<D[i].key)
      max=D[i].key;
  L.keynum=int(ceil(log10(max)));
  L.recnum=n;
  for(i=1;i<=n;i++)
  {
    L.r[i].otheritems=D[i-1].otherinfo;
    itoa(D[i-1].key,c,10); // 将十进制整型转化为字符型，存入c
    for(j=strlen(c);j<L.keynum;j++) // 若c的长度<max的位数，在c前补'0'
    {
      strcpy(c1,"0");
      strcat(c1,c);
      strcpy(c,c1);
    }
    for(j=0;j<L.keynum;j++)
      L.r[i].keys[j]=c[L.keynum-1-j];
  }
}
int ord(char c)
```

```
{ // 返回k的映射(个位整数)
  return c-'0';
}
void Distribute(SLCell r[],int i,ArrType f,ArrType e) // 算法10.15
{ // 静态键表L的r域中记录已按(keys[0], …, keys[i-1])有序。本算法按
  // 第i个关键字keys[i]建立RADIX个子表，使同一子表中记录的keys[i]相同。
  // f[0..RADIX-1]和e[0..RADIX-1]分别指向各子表中第一个和最后一个记录
  int j,p;
  for(j=0;j<RADIX;++j)
    f[j]=0; // 各子表初始化为空表
  for(p=r[0].next;p;p=r[p].next)
  {
    j=ord(r[p].keys[i]); // ord将记录中第i个关键字映射到[0..RADIX-1]
    if(!f[j])
      f[j]=p;
    else
      r[e[j]].next=p;
    e[j]=p; // 将p所指的结点插入第j个子表中
  }
}
int succ(int i)
{ // 求后继函数
  return ++i;
}
void Collect(SLCell r[],ArrType f,ArrType e)
{ // 本算法按keys[i]自小至大地将f[0..RADIX-1]所指各子表依次链接成一个链表，
  // e[0..RADIX-1]为各子表的尾指针。算法10.16
  int j,t;
  for(j=0;!f[j];j=succ(j)); // 找第一个非空子表，succ为求后继函数
  r[0].next=f[j];
  t=e[j]; // r[0].next指向第一个非空子表中第一个结点
  while(j<RADIX-1)
  {
    for(j=succ(j);j<RADIX-1&&!f[j];j=succ(j)); // 找下一个非空子表
    if(f[j])
    { // 链接两个非空子表
      r[t].next=f[j];
      t=e[j];
    }
  }
  r[t].next=0; // t指向最后一个非空子表中的最后一个结点
}
void printl(SLList L)
{ // 按链表输出静态链表
  int i=L.r[0].next,j;
  while(i)
  {
    for(j=L.keynum-1;j>=0;j--)
      printf("%c",L.r[i].keys[j]);
    printf(" ");
    i=L.r[i].next;
```

```
  }
}
void RadixSort(SLList &L)
{ // L是采用静态链表表示的顺序表。对L作基数排序，使得L成为按关键字自小到大的有序静态链表，
  // L.r[0]为头结点。算法10.17
  int i;
  ArrType f,e;
  for(i=0;i<L.recnum;++i)
    L.r[i].next=i+1;
  L.r[L.recnum].next=0; // 将L改造为静态链表
  for(i=0;i<L.keynum;++i)
  { // 按最低位优先依次对各关键字进行分配和收集
    Distribute(L.r,i,f,e); // 第i趟分配
    Collect(L.r,f,e); // 第i趟收集
    printf("第%d趟收集后:\n",i+1);
    printl(L);
    printf("\n");
  }
}
void print(SLList L)
{ // 按数组序号输出静态链表
  int i,j;
  printf("keynum=%d recnum=%d\n",L.keynum,L.recnum);
  for(i=1;i<=L.recnum;i++)
  {
    printf("keys=");
    for(j=L.keynum-1;j>=0;j--)
      printf("%c",L.r[i].keys[j]);
    printf(" otheritems=%d next=%d\n",L.r[i].otheritems,L.r[i].next);
  }
}
void Sort(SLList L,int adr[]) // 改此句(类型)
{ // 求得adr[1..L.length]，adr[i]为静态链表L的第i个最小记录的序号
  int i=1,p=L.r[0].next;
  while(p)
  {
    adr[i++]=p;
    p=L.r[p].next;
  }
}
void Rearrange(SLList &L,int adr[]) // 改此句(类型)。算法10.18(L的类型有变)
{ // adr给出静态链表L的有序次序，即L.r[adr[i]]是第i小的记录。本算法按adr重排L.r，使其有序
  int i,j,k;
  for(i=1;i<L.recnum;++i) // 改此句(类型)
    if(adr[i]!=i)
    {
      j=i;
      L.r[0]=L.r[i]; // 暂存记录L.r[i]
      while(adr[j]!=i)
      { // 调整L.r[adr[j]]的记录到位直到adr[j]=i为止
        k=adr[j];
```

```
        L.r[j]=L.r[k];
        adr[j]=j;
        j=k; // 记录按序到位
      }
      L.r[j]=L.r[0];
      adr[j]=j;
    }
}
#define N 10
void main()
{
  RedType d[N]={{278,1},{109,2},{63,3},{930,4},{589,5},{184,6},{505,7},{269,8},{8,9},
                {83,10}};
  SLList l;
  int *adr;
  InitList(l,d,N);
  printf("排序前(next域还没赋值):\n");
  print(l);
  RadixSort(l);
  printf("排序后(静态链表):\n");
  print(l);
  adr=(int*)malloc((l.recnum)*sizeof(int));
  Sort(l,adr);
  Rearrange(l,adr);
  printf("排序后(重排记录):\n");
  print(l);
}
```

 　程序运行结果(以教科书中图 10.14 的数据为例):

```
排序前(next域还没赋值):
keynum=3 recnum=10
keys=278 otheritems=1 next=8302
keys=109 otheritems=2 next=28521
keys=063 otheritems=3 next=8291
keys=930 otheritems=4 next=28526
keys=589 otheritems=5 next=25376
keys=184 otheritems=6 next=28792
keys=505 otheritems=7 next=25971
keys=269 otheritems=8 next=26983
keys=008 otheritems=9 next=29793
keys=083 otheritems=10 next=31079
第1趟收集后:
930 063 083 184 505 278 008 109 589 269
第2趟收集后:
505 008 109 930 063 269 278 083 184 589
第3趟收集后:
008 063 083 109 184 269 278 505 589 930
```

```
排序后(静态链表):
keynum=3 recnum=10
keys=278 otheritems=1 next=7
keys=109 otheritems=2 next=6
keys=063 otheritems=3 next=10
keys=930 otheritems=4 next=0
keys=589 otheritems=5 next=4
keys=184 otheritems=6 next=8
keys=505 otheritems=7 next=5
keys=269 otheritems=8 next=1
keys=008 otheritems=9 next=3
keys=083 otheritems=10 next=2
排序后(重排记录):
keynum=3 recnum=10
keys=008 otheritems=9 next=3
keys=063 otheritems=3 next=10
keys=083 otheritems=10 next=2
keys=109 otheritems=2 next=6
keys=184 otheritems=6 next=8
keys=269 otheritems=8 next=1
keys=278 otheritems=1 next=7
keys=505 otheritems=7 next=5
keys=589 otheritems=5 next=4
keys=930 otheritems=4 next=0
```

10.7　各种内部排序方法的比较讨论

算法 10.18 在 algo10-2.cpp 中。

第 11 章　外 部 排 序

第 10 章介绍的内部排序需要把待排序的数据先全部放在内存中，然后再排序，这就限制了待排序数据的规模。当数据量特别大时，软件的数据区有可能放不下。algo11-4.cpp 可说明这个问题。

```
// algo11-4.cpp
#define N 32767
void main()
{
  int a[N];
}
```

在 Borland C++ Version 3.1 或 Turbo C 2.0 软件下，当 N≤32 767 时，编译 algo11-4.cpp 顺利通过。而当 N＞32 767 时，编译 algo11-4.cpp 时出错：Array size too large。这个例子说明，要对多于 32 767 个整数进行内部排序是不可能的。首先就没有足够的内存空间存放这些数据。在 Visual C++6.0 环境下，N 的范围可大些，但也是有限的(在 Visual C++6.0 环境下，增加了语句：for(int i=0;i<N;i++) a[i]=i;)。当 N≤259 946 时，可运行 algo11-4.cpp。而当 N＞259 946 时，运行 algo11-4.cpp 时出现消息框，提示：该程序执行了非法操作。经 Debug 查看，原因是：Stack Overflow(堆栈溢出)。

本章介绍在没有足够的内存空间存放待排序数据的情况下，对数据排序的方法。

11.1　外存信息的存取

11.2　外部排序的方法

当存放待排序记录的文件所占存储空间大于可用的内存空间时(把这样的文件称为“大文件”)，显然不能将文件中的所有记录一次读入内存，采用第 10 章介绍的内部排序的方法进行排序。在这种情况下，一般是根据可用内存的大小，先把大文件分几次读入内存，对每次读入的记录进行内部排序，生成几个有序的子文件，如图 11-1 所示。再将几个有序的子文件归并成一个大文件，存于外存中。

11.3　多路平衡归并的实现

将多个有序的子文件归并成 1 个有序的大文件时，每归并 1 次，就要在外存读 1 次记

录。如果减少归并次数，就可减少读文件的次数。但减少归并次数，就要多路归并(同时将几个文件进行归并)。而多路归并要在多个关键字中比较，找到最小值，显然是较慢的。利用“ 败者树”可减少比较次数。

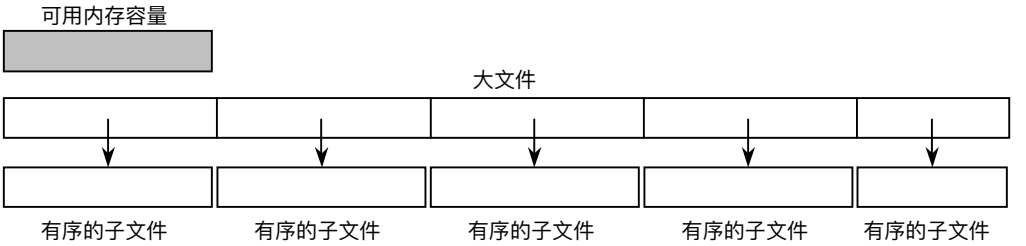


图 11 - 1　将 1 个无序的大文件通过内部排序生成几个有序的子文件

“ 败者树”是完全二叉树，其叶子数恰为待归并的文件数。每个叶子的值为相应有序子文件的当前关键字。“ 败者”的编号存于其双亲结点中,“ 胜者”向上一级去比较。将最终“ 胜者”的记录存于大文件中。“ 胜者”所在的子文件的下一个数据取代它在叶子结点中的位置，继续比较，求出新的“ 胜者”。但它不必和所有叶子的关键字去比较，只需沿着从它的叶子结点到根结点的路径去调整“ 败者树”。这就大大减少了比较次数。

bo11-1.cpp 是 k_路平衡归并要调用的函数。归并路数 k 的值在调用 bo11-1.cpp 的主程序中定义。

algo11-1.cpp 是将一个大文件进行外部排序的完整程序，它是以教科书中图 11.4 的数据为例的。首先由关键字依次为 16，15，10，20，9，18，22，20，40，15，25，6，12，48，37 的结构体数组 a 产生含有 15 个记录的“ 大文件”ori1。假设受内存容量限制，一次只能读入 3 个记录。从“ 大文件”ori1 读记录，经过内部排序，生成 5 个含有 3 个记录的按关键字有序的子文件 f0（10，15，16），f1（9，18，20），f2（20，22，40），f3（6，15，25），f4（12，37，48）。括号中是每个记录的关键字。为了方便起见，在有序子文件 f0，f1，f2，…的尾部附加 1 个关键字为最大值的记录。打开这 5 个有序子文件，利用“ 败者树”归并成 1 个“ 大文件”fout。

```
// bo11-1.cpp k路平衡归并的函数，包括算法11.1～11.3
FILE *fp[k+1]; // k+1个文件指针(fp[k]为大文件指针)，全局变量
typedef int LoserTree[k]; // 败者树是完全二叉树且不含叶子，可采用顺序存储结构
typedef RedType ExNode,External[k+1]; // 外结点，有改变
External b; // 全局变量
#define MIN_KEY INT_MIN
#define MAX_KEY INT_MAX
void input(int i,KeyType &a)
{ // 从第i个文件(第i个归并段)读入该段当前第1个记录的关键字到外结点
  fread(&a,sizeof(KeyType),1,fp[i]);
}
void output(int i)
{ // 将第i个文件(第i个归并段)中当前的记录写至输出归并段
  ExNode a;
  a.key=b[i].key; // 当前记录的关键字已读到b[i].key中
  fread(&a.otherinfo,sizeof(InfoType),1,fp[i]);
```

```
  fwrite(&a,sizeof(ExNode),1,fp[k]);
}
void Adjust(LoserTree ls,int s)
{ // 沿从叶子结点b[s]到根结点ls[0]的路径调整败者树。算法11.2
  int i,t;
  t=(s+k)/2; // ls[t]是b[s]的双亲结点
  while(t>0)
  {
    if(b[s].key>b[ls[t]].key)
    {
      i=s;
      s=ls[t]; // s指示新的胜者
      ls[t]=i;
    }
    t=t/2;
  }
  ls[0]=s;
}
void CreateLoserTree(LoserTree ls)
{ // 已知b[0]到b[k-1]为完全二叉树ls的叶子结点，存有k个关键字，沿从叶子
  // 到根的k条路径将ls调整成为败者树。算法11.3
  int i;
  b[k].key=MIN_KEY;
  for(i=0;i<k;++i)
    ls[i]=k; // 设置ls中“败者”的初值
  for(i=k-1;i>=0;--i) // 依次从b[k-1]，b[k-2]，…，b[0]出发调整败者
    Adjust(ls,i);
}
void K_Merge(LoserTree ls,External b)
{ // 利用败者树ls将编号从0到k-1的k个输入归并段中的记录归并到输出归并段。b[0]至b[k-1]为
  // 败者树上的k个叶子结点，分别存放k个输入归并段中当前记录的关键字。算法11.1
  int i,q;
  for(i=0;i<k;++i) // 分别从k个输入归并段读入该段当前第一个记录的关键字到外结点
    input(i,b[i].key);
  CreateLoserTree(ls); // 建败者树ls，选得最小关键字为b[ls[0]].key
  while(b[ls[0]].key!=MAX_KEY)
  {
    q=ls[0]; // q指示当前最小关键字所在归并段
    output(q); // 将编号为q的归并段中当前(关键字为b[q].key)的记录写至输出归并段
    input(q,b[q].key); // 从编号为q的输入归并段中读入下一个记录的关键字
    Adjust(ls,q); // 调整败者树，选择新的最小关键字
  }
  output(ls[0]); // 将含最大关键字MAX_KEY的记录写至输出归并段
}

// algo11-1.cpp 调用bo11-1.cpp的程序
#include"c1.h"
typedef int InfoType; // 定义其它数据项的类型
#include"c9-7.h"
#include"c10-1.h" // 定义KeyType、RedType及SqList
#include"bo10-1.cpp"
```

```
#define k 5 // k路归并
#include"bo11-1.cpp"
#define N 3 // 设每个小文件有N个数据(可将整个文件一次读入内存的称为小文件)
#define M 10 // 设输出M个数据换行
void print(RedType t)
{
  printf("(%d,%d)",t.key,t.otherinfo);
}
void main()
{
  RedType a[k*N]={{16,1},{15,2},{10,3},{20,4},{9,5},{18,6},{22,7},{20,8},{40,9},{15,10},
                  {25,11},{6,12},{12,13},{48,14},{37,15}}; // 有k*N个记录的数组
  RedType r,t={MAX_KEY}; // 小文件尾部的结束标志
  SqList l;
  int i,j;
  char fname[k][3],fori[4]="ori",fout[4]="out",s[3];
  LoserTree ls;
  // 由数组a创造1个大文件(不能将整个文件一次读入内存的称为大文件)
  fp[k]=fopen(fori,"wb"); // 以写的方式打开大文件fori
  fwrite(a,sizeof(RedType),k*N,fp[k]); // 将数组a中的数据写入文件fori中(表示1个大文件)
  fclose(fp[k]); // 关闭文件fori
  fp[k]=fopen(fori,"rb"); // 以读的方式打开大文件fori
  printf("大文件的记录为\n");
  for(i=1;i<=N*k;i++)
  {
    fread(&r,sizeof(RedType),1,fp[k]); // 依次将大文件fori的数据读入r
    print(r); // 输出r的内容
    if(i%M==0)
      printf("\n");
  }
  printf("\n");
  rewind(fp[k]);
  // 使fp[k]的指针重新返回大文件fori的起始位置，以便重新读入内存，产生有序的子文件
  for(i=0;i<k;i++) // 将大文件fori的数据分成k组，每组N个数据
  { // 排序后分别存到小文件f0，f1，…
    fread(&l.r[1],sizeof(RedType),N,fp[k]); // 将大文件fori的N个数据读入l
    l.length=N;
    InsertSort(l); // 对l进行内部排序
    itoa(i,s,10); // 生成k个文件名f0，f1，f2，…
    strcpy(fname[i],"f");
    strcat(fname[i],s);
    fp[i]=fopen(fname[i],"wb"); // 以写的方式打开文件f0，f1，…
    fwrite(&l.r[1],sizeof(RedType),N,fp[i]); // 将排序后的N个数据分别写入f0，f1，…
    fwrite(&t,sizeof(RedType),1,fp[i]); // 将文件结束标志分别写入f0，f1，…
    fclose(fp[i]); // 关闭文件f0，f1，…
  }
  fclose(fp[k]); // 关闭大文件fori
  for(i=0;i<k;i++)
  { // 依次打开f0，f1，f2，…，k个文件
    itoa(i,s,10); // 生成k个文件名f0，f1，f2，…
    strcpy(fname[i],"f");
```

```
    strcat(fname[i],s);
    fp[i]=fopen(fname[i],"rb"); // 以读的方式打开文件f0, f1, …
    printf("有序子文件f%d的记录为\n",i);
    for(j=0;j<=N;j++)
    {
      fread(&r,sizeof(RedType),1,fp[i]); // 依次将f0, f1, …的数据读入r
      print(r); // 输出r的内容
    }
    printf("\n");
    rewind(fp[i]); // 使fp[i]的指针重新返回f0, f1, …的起始位置, 以便重新读入内存
  }
  fp[k]=fopen(fout,"wb"); // 以写的方式打开大文件fout
  K_Merge(ls,b); // 利用败者树ls将k个输入归并段中的记录归并到输出归并段, 即大文件fout
  for(i=0;i<k;i++)
    fclose(fp[i]); // 关闭文件f0, f1, …
  fclose(fp[k]); // 关闭文件fout
  fp[k]=fopen(fout,"rb"); // 以读的方式打开大文件fout验证排序
  printf("排序后的大文件的记录为\n");
  for(i=1;i<=N*k+1;i++)
  {
    fread(&t,sizeof(RedType),1,fp[k]);
    print(t);
    if(i%M==0)
      printf("\n");
  }
  printf("\n");
  fclose(fp[k]); // 关闭文件fout
}
```

程序运行结果(以教科书中图 11.4 的数据为例):

```
大文件的记录为
(16,1)(15,2)(10,3)(20,4)(9,5)(18,6)(22,7)(20,8)(40,9)(15,10)
(25,11)(6,12)(12,13)(48,14)(37,15)
有序子文件f0的记录为
(10,3)(15,2)(16,1)(32767,0)
有序子文件f1的记录为
(9,5)(18,6)(20,4)(32767,0)
有序子文件f2的记录为
(20,8)(22,7)(40,9)(32767,0)
有序子文件f3的记录为
(6,12)(15,10)(25,11)(32767,0)
有序子文件f4的记录为
(12,13)(37,15)(48,14)(32767,0)
排序后的大文件的记录为
(6,12)(9,5)(10,3)(12,13)(15,10)(15,2)(16,1)(18,6)(20,4)(20,8)
(22,7)(25,11)(37,15)(40,9)(48,14)(32767,0)
```

11.4　置换—选择排序

要提高外部排序的效率，除了选择“败者树”归并方法外，还应该尽量使有序子文件较长，从而减少待归并文件的数量。但像图 11-1 所示那样由外存整体读入数据到内存，经内部排序后，再整体输出到外存形成的有序子文件，则有序子文件的长度不会大于可用内存容量。

置换—选择排序采取增多待排序记录数的方法使生成的有序子文件更长。具体算法是：逐个把已排序的记录送入外存，再由“大文件”读入一个记录到内存中空出的位置，只要这个记录的关键字不小于刚送到外存的记录的关键字，该记录就会排到当前有序子文件中。下面以教科书中图 11.5 为例来说明：

设内存只能存 6 个记录。首先由大文件 FI 中读取前 6 个记录（51，49，39，46，38，29）到内存 WA，找出最小值（29），存于外存文件 FO 中。再由大文件 FI 读取第 7 个记录（14）到内存 WA，占据原 29 的位置。由于 14 小于 29，所以 14 不可能在当前外存文件 FO 中。要在内存 WA 中不小于 29 的关键字中找最小值，找到 38。将关键字为 38 的记录存于外存文件 FO 中。再由大文件 FI 读取第 8 个记录（61）到内存 WA，占据原 38 的位置。由于 61＞38，所以 61 可以排在当前外存文件 FO 中。依此类推，直到内存 WA 中所有的关键字都小于外存文件 FO 的最后一个记录的关键字，第 1 个初始归并段完成。写入关键字为最大整数的记录作为段结束标志，开始建立第 2 个初始归并段。依此类推，直到 FI 的所有记录都存到 FO 的初始归并段中。

algo11-2.cpp 是以教科书中图 11.5 的数据为例采用置换—选择排序的方法产生初始归并段文件的例子。按照一般的方法，要产生 4 个初始归并段文件。而采用置换—选择排序的方法只产生 3 个初始归并段文件。

```
// algo11-2.cpp 通过置换－选择排序产生不等长的初始归并段文件，包括算法11.4～11.7
#include"c1.h"
typedef int InfoType; // 定义其它数据项的类型
#include"c10-1.h" // 定义KeyType、RedType及SqList
#define MAX_KEY INT_MAX
#define RUNEND_SYMBOL INT_MAX
#define w 6 // 内存工作区可容纳的记录个数
#define M 10 // 设输出M个数据换行
#define N 24 // 设大文件有N个数据
typedef int LoserTree[w]; // 败者树是完全二叉树且不含叶子，可采用顺序存储结构
typedef struct
{
  RedType rec; // 记录
  KeyType key; // 从记录中抽取的关键字
  int rnum; // 所属归并段的段号
}RedNode,WorkArea[w]; // 内存工作区，容量为w
void Select_MiniMax(LoserTree ls,WorkArea wa,int q) // 算法11.6
{ // 从wa[q]起到败者树的根比较选择MINIMAX记录，并由q指示它所在的归并段
  int p,s,t;
```

```
  for(t=(w+q)/2,p=ls[t];t>0;t=t/2,p=ls[t])
    if(wa[p].rnum<wa[q].rnum||wa[p].rnum==wa[q].rnum&&wa[p].key<wa[q].key)
    {
      s=q;
      q=ls[t]; // q指示新的胜利者
      ls[t]=s;
    }
  ls[0]=q;
}
void Construct_Loser(LoserTree ls,WorkArea wa,FILE *fi)
{ // 输入w个记录到内存工作区wa，建得败者树ls，选出关键字最小的记录并由s指示
  // 其在wa中的位置。算法11.7
  int i;
  for(i=0;i<w;++i)
    wa[i].rnum=wa[i].key=ls[i]=0; // 工作区初始化
  for(i=w-1;i>=0;--i)
  {
    fread(&wa[i].rec,sizeof(RedType),1,fi); // 输入一个记录
    wa[i].key=wa[i].rec.key; // 提取关键字
    wa[i].rnum=1; // 其段号为“1”
    Select_MiniMax(ls,wa,i); // 调整败者
  }
}
void get_run(LoserTree ls,WorkArea wa,int rc,int &rmax,FILE *fi,FILE *fo)
{ // 求得一个初始归并段，fi为输入文件指针，fo为输出文件指针。算法11.5
  int q;
  KeyType minimax;
  while(wa[ls[0]].rnum==rc) // 选得的MINIMAX记录属当前段时
  {
    q=ls[0]; // q指示MINIMAX记录在wa中的位置
    minimax=wa[q].key;
    fwrite(&wa[q].rec,sizeof(RedType),1,fo); // 将刚选得的MINIMAX记录写入输出文件
    fread(&wa[q].rec,sizeof(RedType),1,fi); // 从输入文件读入下一记录(改)
    if(feof(fi))
    { // 输入文件结束，虚设记录(属“rmax+1”段)
      wa[q].rnum=rmax+1;
      wa[q].key=MAX_KEY;
    }
    else
    { // 输入文件非空时
      wa[q].key=wa[q].rec.key; // 提取关键字
      if(wa[q].key<minimax)
      { // 新读入的记录属下一段
        rmax=rc+1;
        wa[q].rnum=rmax;
      }
      else // 新读入的记录属当前段
        wa[q].rnum=rc;
    }
    Select_MiniMax(ls,wa,q); // 选择新的MINIMAX记录
  }
```

```
}
void Replace_Selection(LoserTree ls,WorkArea wa,FILE *fi,FILE *fo)
{ // 在败者树ls和内存工作区wa上用置换－选择排序求初始归并段，fi为输入文件
  // (只读文件)指针，fo为输出文件(只写文件)指针，两个文件均已打开。算法11.4
  int rc,rmax;
  RedType j;
  j.key=RUNEND_SYMBOL;
  Construct_Loser(ls,wa,fi); // 初建败者树
  rc=rmax=1;// rc指示当前生成的初始归并段的段号，rmax指示wa中关键字所属初始归并段的最大段号
  while(rc<=rmax) // "rc=rmax+1"标志输入文件的置换－选择排序已完成
  {
    get_run(ls,wa,rc,rmax,fi,fo); // 求得一个初始归并段
    j.otherinfo=rc;
    fwrite(&j,sizeof(RedType),1,fo); // 将段结束标志写入输出文件
    rc=wa[ls[0]].rnum; // 设置下一段的段号
  }
}
void print(RedType t)
{
  printf("(%d,%d)",t.key,t.otherinfo);
}
void main()
{
  RedType b,a[N]={{51,1},{49,2},{39,3},{46,4},{38,5},{29,6},{14,7},{61,8},{15,9},{30,10},
                  {1,11},{48,12},{52,13},{3,14},{63,15},{27,16},{4,17},{13,18},{89,19},
                  {24,20},{46,21},{58,22},{33,23},{76,24}};
  FILE *fi,*fo;
  LoserTree ls;
  WorkArea wa;
  int i,k,j=RUNEND_SYMBOL;
  char s[3],fname[4];
  fo=fopen("ori","wb"); // 以写的方式打开大文件ori
  fwrite(a,sizeof(RedType),N,fo); // 将数组a写入大文件ori
  fclose(fo);
  fi=fopen("ori","rb"); // 以读的方式重新打开大文件ori
  printf("大文件的记录为\n");
  for(i=1;i<=N;i++)
  {
    fread(&b,sizeof(RedType),1,fi); // 依次将大文件ori的数据读入b
    print(b); // 输出b的内容
    if(i%M==0)
      printf("\n");
  }
  printf("\n");
  rewind(fi); // 使fi的指针重新返回大文件ori的起始位置，以便重新读入内存，产生有序的子文件
  fo=fopen("out","wb"); // 以写的方式打开初始归并段文件out
  Replace_Selection(ls,wa,fi,fo); // 用置换－选择排序求初始归并段
  fclose(fo);
  fclose(fi);
  fi=fopen("out","rb"); // 以读的方式重新打开初始归并段文件out
  printf("初始归并段文件的记录为\n");
```

```
  i=1;
  do
  {
    k=fread(&b,sizeof(RedType),1,fi); // 依次将大文件out的数据读入b
    if(k==1)
    {
      print(b); // 输出b的内容
      if(i++%M==0)
        printf("\n");
    }
  }while(k==1);
  printf("\n");
  rewind(fi); // 使fi的指针重新返回大文件ori的起始位置，以便重新读入内存，产生有序的子文件
  k=0;
  while(!feof(fi)) // 按段输出初始归并段文件out
  {
    itoa(k,s,10); // 依次生成文件名f0, f1, …
    strcpy(fname,"f");
    strcat(fname,s);
    fo=fopen(fname,"wb"); // 依次以写的方式打开文件f0, f1, …
    do
    {
      i=fread(&b,sizeof(RedType),1,fi);
      if(i==1) // fread()调用成功
      {
        fwrite(&b,sizeof(RedType),1,fo); // 将b写入文件f0, f1, …
        if(b.key==j) // 1个归并段结束
        {
          k++;
          fclose(fo);
          break;
        }
      }
    }while(i==1);
  };
  fclose(fi);
  printf("共产生%d个初始归并段文件\n",k);
}
```

程序运行结果(以教科书中图 11.5 的数据为例)：

```
大文件的记录为
(51,1)(49,2)(39,3)(46,4)(38,5)(29,6)(14,7)(61,8)(15,9)(30,10)
(1,11)(48,12)(52,13)(3,14)(63,15)(27,16)(4,17)(13,18)(89,19)(24,20)
(46,21)(58,22)(33,23)(76,24)
初始归并段文件的记录为
(29,6)(38,5)(39,3)(46,4)(49,2)(51,1)(61,8)(32767,1)(1,11)(3,14)
(14,7)(15,9)(27,16)(30,10)(48,12)(52,13)(63,15)(89,19)(32767,2)(4,17)
(13,18)(24,20)(33,23)(46,21)(58,22)(76,24)(32767,3)
共产生3个初始归并段文件
```

algo11-3.cpp 是将运行 algo11-2.cpp 产生的 3 个初始归并段文件 f0，f1，f2 归并成 1 个有序的大文件的程序，它和 algo11-1.cpp 很相像，只是去掉了程序前面的一些内容。

```
// algo11-3.cpp 将algo11-2.cpp产生的有序子文件f0,f1,f2归并成1个有序的大文件out
#include"c1.h"
typedef int InfoType; // 定义其它数据项的类型
#include"c10-1.h" // 定义KeyType、RedType及SqList
#define k 3 // k路归并
#define M 10 // 设输出M个数据换行
#include"bo11-1.cpp"
void print(RedType t)
{
  printf("(%d,%d)",t.key,t.otherinfo);
}
void main()
{
  RedType r;
  int i,j;
  char fname[k][4],fout[5]="out",s[3];
  LoserTree ls;
  for(i=0;i<k;i++)
  { // 依次打开f0，f1，f2，…，k个文件
    itoa(i,s,10); // 生成k个文件名f0，f1，f2，…
    strcpy(fname[i],"f");
    strcat(fname[i],s);
    fp[i]=fopen(fname[i],"rb"); // 以读的方式打开文件f0，f1，…
    printf("有序子文件f%d的记录为\n",i);
    do
    {
      j=fread(&r,sizeof(RedType),1,fp[i]); // 依次将f0，f1，…的数据读入r
      if(j==1)
        print(r); // 输出r的内容
    }while(j==1);
    printf("\n");
    rewind(fp[i]); // 使fp[i]的指针重新返回f0，f1，…的起始位置，以便重新读入内存
  }
  fp[k]=fopen(fout,"wb"); // 以写的方式打开大文件fout
  K_Merge(ls,b); // 利用败者树ls将k个输入归并段中的记录归并到输出归并段，即大文件fout
  for(i=0;i<=k;i++)
    fclose(fp[i]); // 关闭文件f0，f1，…和文件fout
  fp[k]=fopen(fout,"rb"); // 以读的方式重新打开大文件fout验证排序
  printf("排序后的大文件的记录为\n");
  i=1;
  do
  {
    j=fread(&r,sizeof(RedType),1,fp[k]); // 将fout的数据读入r
    if(j==1)
      print(r); // 输出r的内容
    if(i++%M==0)
      printf("\n"); // 换行
```

```
  }while(j==1);
  printf("\n");
  fclose(fp[k]); // 关闭大文件fout
}
```

程序运行结果(以教科书中图 11.5 的数据为例)：

```
有序子文件f0的记录为
(29,6)(38,5)(39,3)(46,4)(49,2)(51,1)(61,8)(32767,1)
有序子文件f1的记录为
(1,11)(3,14)(14,7)(15,9)(27,16)(30,10)(48,12)(52,13)(63,15)(89,19)(32767,2)
有序子文件f2的记录为
(4,17)(13,18)(24,20)(33,23)(46,21)(58,22)(76,24)(32767,3)
排序后的大文件的记录为
(1,11)(3,14)(4,17)(13,18)(14,7)(15,9)(24,20)(27,16)(29,6)(30,10)
(33,23)(38,5)(39,3)(46,4)(46,21)(48,12)(49,2)(51,1)(52,13)(58,22)
(61,8)(63,15)(76,24)(89,19)(32767,2)
```

第 12 章　文　　件

12.1　有关文件的基本概念

12.2　顺序文件

```
// algo12-1.cpp 根据事务文件成批地更改主文件并得到一个新的主文件，包括算法12.1
#include"c1.h"
struct RedType // 主文件记录类型
{
  int accounts; // 帐号
  int amount; // 余额
};
struct RcdType // 事务文件记录类型(比主文件记录类型多了成员code)
{
  int accounts; // 帐号
  int amount; // 存取的数量(存为+，取为-)
  char code; // 修改要求(I：插入，U：修改，D：删除)
};
#define key accounts
RedType P(RcdType g)
{ // 把g加工为q的结构返回
  RedType q;
  q.accounts=g.accounts;
  q.amount=g.amount;
  return q;
}
void Q(RedType &f,RcdType g)
{ // 将f和g归并成一个f结构的记录
  f.amount+=g.amount;
}
void MergeFile(FILE *f,FILE *g,FILE *h) // 算法12.1
{ // 由按关键字递增有序的非空顺序文件f和g归并得到新文件h，三个文件均已打开，其中，f和g为
  // 只读文件，文件中各附加一个最大关键字记录，且g文件中对该记录的操作为插入。h为只写文件
  RedType fr,fn;
  RcdType gr;
  int i;
  fread(&fr,sizeof(RedType),1,f);
  fread(&gr,sizeof(RcdType),1,g);
```

```
  while(!feof(f)||!feof(g))
  {
    if(fr.key<gr.key)
      i=1;
    else if(gr.code=='D' &&fr.key==gr.key)
      i=2;
    else if(gr.code=='I' &&fr.key>gr.key)
      i=3;
    else if(gr.code=='U' &&fr.key==gr.key)
      i=4;
    else
      i=0;
    switch(i)
    {
      case 1: // 复制“旧”主文件中记录
              fwrite(&fr,sizeof(RedType),1,h);
              if(!feof(f))
                fread(&fr,sizeof(RedType),1,f);
              break;
      case 2: // 删除“旧”主文件中记录，即不复制
              if(!feof(f))
                fread(&fr,sizeof(RedType),1,f);
              if(!feof(g))
                fread(&gr,sizeof(RcdType),1,g);
              break;
      case 3: // 插入
              fn=P(gr); // 函数P把gr加工为h的结构
              fwrite(&fn,sizeof(RedType),1,h);
              if(!feof(g))
                fread(&gr,sizeof(RcdType),1,g);
              break;
      case 4: // 更改“旧”主文件中记录
              Q(fr,gr); // 函数Q将fr和gr归并成一个h结构的记录
              fwrite(&fr,sizeof(RedType),1,h);
              if(!feof(f))
                fread(&fr,sizeof(RedType),1,f);
              if(!feof(g))
                fread(&gr,sizeof(RcdType),1,g);
              break;
      default:exit(ERROR); // 其它均为出错情况
    }
  }
}
void print(RedType t)
{
  printf("%6d%4d\n",t.accounts,t.amount);
}
void printc(RcdType t)
{
  printf("%6d%6d%8c\n",t.accounts,t.amount,t.code);
}
```

```
void main()
{
  RedType c,a[8]={{1,50},{5,78},{12,100},{14,95},{15,360},{18,200},{20,510},{INT_MAX,0}};
  // 主文件数据
  RcdType d,b[6]={{8,100, 'I'},{12,-25, 'U'},{14,38, 'U'},{18,-200, 'D'},{21,60, 'I'},
  {INT_MAX,0, 'U'}}; // 已排序的事务文件数据
  FILE *f,*g,*h;
  int j;
  f=fopen("old","wb"); // 以写的方式打开主文件old
  fwrite(a,sizeof(RedType),8,f); // 将数组a中的数据写入文件old
  fclose(f); // 关闭文件old, 形成主文件
  f=fopen("change","wb"); // 以写的方式打开事务文件change
  fwrite(b,sizeof(RcdType),6,f); // 将数组b中的数据写入文件change
  fclose(f); // 关闭文件change, 形成已排序的事务文件
  f=fopen("old","rb"); // 以读的方式打开主文件old
  printf("主文件内容:\n");
  printf("  帐号 余额\n");
  do
  {
    j=fread(&c,sizeof(RedType),1,f);
    if(j==1)
      print(c); // 输出r的内容
  }while(j==1);
  rewind(f); // 使f的指针重新返回文件的起始位置, 以便重新读入内存
  g=fopen("change","rb"); // 以读的方式打开已排序的事务文件change
  printf("已排序的事务文件内容:\n");
  printf("  帐号 存取数量 修改要求\n");
  do
  {
    j=fread(&d,sizeof(RcdType),1,g);
    if(j==1)
      printc(d); // 输出r的内容
  }while(j==1);
  rewind(g); // 使g的指针重新返回文件的起始位置, 以便重新读入内存
  h=fopen("new","wb"); // 以写的方式打开新主文件new
  MergeFile(f,g,h); // 生成新主文件
  fclose(f); // 关闭文件old
  fclose(g); // 关闭文件change
  fclose(h); // 关闭文件new
  f=fopen("new","rb"); // 以读的方式打开新主文件new
  printf("新主文件内容:\n");
  printf("  帐号 余额\n");
  do
  {
    j=fread(&c,sizeof(RedType),1,f);
    if(j==1)
      print(c); // 输出r的内容
  }while(j==1);
  fclose(f); // 关闭文件new
}
```

程序运行结果(以教科书中图 12.4 为例)：

```
主文件内容:
  帐号  余额
     1   50
     5   78
    12  100
    14   95
    15  360
    18  200
    20  510
 32767    0
已排序的事务文件内容:
  帐号  存取数量  修改要求
     8    100        I
    12    -25        U
    14     38        U
    18   -200        D
    21     60        I
 32767      0        U
新主文件内容:
  帐号  余额
     1   50
     5   78
     8  100
    12   75
    14  133
    15  360
    20  510
    21   60
 32767    0
```

附录 A　关于标准 C 程序

本书中的程序一般需经修改才能在标准 C 的环境下运行，主要有以下几个原因：

(1) 教材的算法中采用了 C++语言的引用调用的参数传递方式。在形参表中，以&打头的参数即为引用参数。

标准 C 不支持引用参数，对此需进行转换。下面以 bo1-1.cpp 和 bo1-1.c 中 DestroyTriplet()函数为例来说明这种转换。bo1-1.cpp 中含有引用参数的函数如下：

```
Status DestroyTriplet(Triplet &T)
{ // 操作结果：三元组T被销毁
  free(T);
  T=NULL;
  return OK;
}
```

转换后在 bo1-1.c 中的标准 C 程序如下：

```
Status DestroyTriplet(Triplet *T) /* 将&T改为*T */
{ /* 操作结果：三元组T被销毁 */
  free(*T); /* 将T改为*T */
  *T=NULL; /* 将T改为*T */
  return OK;
}
```

对照以上 2 个函数可见：将 C++函数形参表中以&打头的参数改成以*打头的参数，再在函数中该参数前加*即可。要注意的是，在这两个函数中，形参的类型是不同的。在 bo1-1.cpp 中，T 的类型是 Triplet；在 bo1-1.c 中，T 的类型是 Triplet 的指针。但为方便起见，没有改写 C 程序中的注释。另外，在标准 C 程序中调用该函数，实参前应加&。如 main1-1.cpp 中调用 DestroyTriplet()的语句为

```
DestroyTriplet(T);
```

相应的标准 C 程序 main1-1.c 中调用 DestroyTriplet()的语句为(注意带下划线部分)

```
DestroyTriplet(&T);
```

其中，在调用 DestroyTriplet()的两程序中，两实参 T 的类型是相同的。另外，在转换过程中，遇到&*或*&可“抵消”，即将*&T 转换为 T。

(2) 标准 C 在指明所定义的结构体或枚举类型时，在类型前要加 strutc 或 enum，而 C++则不必加。如在 C++的 c2-1.h 中定义结构体 SqList 如下：

```
struct SqList
{
  ElemType *elem; // 存储空间基址
  int length; // 当前长度
  int listsize; // 当前分配的存储容量(以 sizeof(ElemType)为单位)
};
```

C++在指明变量或形参的类型时，只需用 SqList 即可。如 bo2-1. cpp 中的一个函数如下：

```
int ListLength(SqList L)
{ // 初始条件：顺序线性表L已存在。操作结果：返回L中数据元素个数
  return L.length;
}
```

如果在标准 C 程序中像 C++的 c2-1. h 那样定义变量 L 的类型，则在 bo2-1. c 中该函数应如下(注意带下划线部分)：

```
int ListLength(struct SqList L)
{ /* 初始条件：顺序线性表L已存在。操作结果：返回L中数据元素个数 */
  return L.length;
}
```

说明变量 L 时要用 struct SqList。当用到某个结构体就要在其类型前加 struct，用到某个枚举类型就要在其类型前加 enum 是很麻烦的。为了方便起见，可用 typedef 定义类型。在标准 C 程序的 c2-1. h 中定义 SqList 如下(注意带下划线部分)：

```
typedef struct
{
  ElemType *elem; /* 存储空间基址 */
  int length; /* 当前长度 */
  int listsize; /* 当前分配的存储容量(以sizeof(ElemType)为单位) */
}SqList;
```

这样，在 bo2-1. c 中相应的函数为

```
int ListLength(SqList L)
{ /* 初始条件：顺序线性表L已存在。操作结果：返回L中数据元素个数 */
  return L.length;
```

```
}
```

这个函数与 bo2-1.cpp 很相似。可见，只要在定义结构体时使用 typedef，则在指明结构体的类型时就不必加 struct。本书中的标准 C 都采用这种方法定义结构体。定义枚举类型时使用 typedef 的方法与此类似。

(3) 标准 C 的共用体必须有变量名，而 C++可以省略。如在 c5-6.h(C++)中定义 GLNode1 如下：

```
// c5-6.h 广义表的扩展线性链表存储结构
enum ElemTag{ATOM,LIST}; // ATOM==0: 原子, LIST==1: 子表
typedef struct GLNode1
{
  ElemTag tag; // 公共部分，用于区分原子结点和表结点
  union // 原子结点和表结点的联合部分
  {
    AtomType atom; // 原子结点的值域
    GLNode1 *hp; // 表结点的表头指针
  };
  GLNode1 *tp; // 相当于线性链表的next，指向下一个元素结点
}*GList1,GLNode1; // 广义表类型GList1是一种扩展的线性链表
```

注意：其中的共用体没有变量名。在 bo5-6.cpp 中的函数 GListEmpty()如下：

```
Status GListEmpty(GList1 L)
{ // 初始条件: 广义表L存在。操作结果: 判定广义表L是否为空
  if(!L||L->tag==LIST&&!L->hp)
    return OK;
  else
    return ERROR;
}
```

而标准 C 的 c5-6.h 如下(注意带下划线部分)：

```
/* c5-6.h 广义表的扩展线性链表存储结构 */
typedef enum{ATOM,LIST}ElemTag; /* ATOM==0: 原子, LIST==1: 子表 */
typedef struct GLNode1
{
  ElemTag tag; /* 公共部分，用于区分原子结点和表结点 */
  union /* 原子结点和表结点的联合部分 */
  {
    AtomType atom; /* 原子结点的值域 */
    struct GLNode1 *hp; /* 表结点的表头指针 */
  }a;
  struct GLNode1 *tp; /* 相当于线性链表的next，指向下一个元素结点 */
}*GList1,GLNode1; /* 广义表类型GList1是一种扩展的线性链表 */
```

它内部的共用体必须有变量名。在 bo5-6.c 的函数 GListEmpty()中也变动如下(注意带下

划线部分)：

```
Status GListEmpty(GList1 L)
{ /* 初始条件：广义表L存在。操作结果：判定广义表L是否为空 */
  if(!L||L->tag==LIST&&!L->a.hp)
    return OK;
  else
    return ERROR;
}
```

(4) C++可重载。即在一个程序中，可有几个同名的函数同时存在。只要它们的形参个数或类型有所不同即可。标准 C 不可重载。在 C++转换成标准 C 时，必须将同名函数改为不同名。如在 bo9-2.cpp 中有 1 个 SearchBST()函数(算法 9.5(b))，在 bo9-2.cpp 所包含的文件 func9-1.cpp 中也有 1 个 SearchBST()函数(算法 9.5(a))，但这两个同名函数的形参个数不同。C++根据函数的形参个数可分辨所调用的是哪个函数。而标准 C 没有这个能力，所以把 bo9-2.c 中的 SearchBST()函数改为 SearchBST1()函数。

(5) C++允许在执行语句中变量使用之前定义变量。如 algo8-1.cpp 中的 PrintUser()函数(注意带下划线部分)：

```
void PrintUser(Space p[])
{ // 输出p数组所指的已分配空间
  for(int i=0;i<MAX/e;i++)
    if(p[i]) // 指针不为0(指向一个占用块)
    {
      printf("块%d的首地址=%u ",i,p[i]); // 输出结点信息
      printf("块的大小=%d 块头标志=%d(0:空闲 1:占用)",p[i]->size,p[i]->tag);
      printf(" 块尾标志=%d\n",(FootLoc(p[i]))->tag);
    }
}
```

这种形式在标准 C 语言中是不允许的。相应的 algo8-1.c 中的 PrintUser()函数为(注意带下划线部分)：

```
void PrintUser(Space p[])
{ /* 输出p数组所指的已分配空间 */
  int i;
  for(i=0;i<MAX/e;i++)
    if(p[i]) /* 指针不为0(指向一个占用块) */
    {
      printf("块%d的首地址=%u ",i,p[i]); /* 输出结点信息 */
      printf("块的大小=%d 块头标志=%d(0:空闲 1:占用)",p[i]->size,p[i]->tag);
      printf(" 块尾标志=%d\n",(FootLoc(p[i]))->tag);
    }
}
```

(6) C++允许在转换类型时采用函数的形式，如 algo8-2.cpp 中 AllocBuddy()函数的

一条语句如下(注意带下划线部分):

```
pi=pa+int(pow(2,k-i));
```

而在标准 C 程序 algo8-2.c 中必须改为以下形式(注意带下划线部分):

```
pi=pa+(int)pow(2,k-i);
```

(7) 在 C++中可用 new 申请空间，如 bo8-1.cpp 中主函数 main()的一条语句如下:

```
p=new WORD[MAX+2]; // 申请大小为MAX*sizeof(WORD)个字节的空间
```

在标准 C 的 bo8-1.c 中要改为

```
p=(WORD*)malloc((MAX+2)*sizeof(WORD)); /* 申请大小为MAX*sizeof(WORD)个字节的空间 */
```

(8) 在 C++中可用 cout 和 cin 做输入输出语句，如 main1-1.cpp 中有

```
cout<<T[0]<<' '<<T[1]<<' '<<T[2]<<endl;
```

它们的好处是不必给出格式符。这样，当变量的类型发生变化时，不必修改语句。但在 main1-1.c 中必须改为

```
printf("%d %d %d\n",T[0],T[1],T[2]);/*当ElemType的类型变化时，要相应改变printf()的格式符 */
```

(9) 在 C++中“//”后到本行末的内容为注释，而标准 C 的注释必须放在“/*”、“*/”之间。如上许多例子所示。

(10) 本书中标准 C 程序的扩展名为.c，C++程序的扩展名为.cpp。定义数据存储结构的文件，其扩展名是.h，在两种语言中不能混用。

附录 B　光盘文件目录

```
Directory of F:\
BC              <DIR>
TC              <DIR>
VC              <DIR>
0 file(s)     0 bytes

Directory of F:\BC
CH1             <DIR>
CH2             <DIR>
CH3             <DIR>
CH4             <DIR>
CH5             <DIR>
CH6             <DIR>
CH7             <DIR>
CH8             <DIR>
CH9             <DIR>
CH10            <DIR>
CH11            <DIR>
CH12            <DIR>
0 file(s)     0 bytes

Directory of F:\BC\CH1
文件名 扩展名   字节   章节
ALG01-1  CPP     515  1.4.3
ALG01-2  CPP     483  1.4.3
ALG01-3  CPP     551  1.3
ALG01-4  CPP     279  1.3
B01-1    CPP   1,503  1.3
MAIN1-1  CPP   1,608  1.3
C1       H       660  1.3
C1-1     H       165  1.3
8 file(s)      5,764 bytes

Directory of F:\BC\CH2
文件名 扩展名   字节   章节
ALG02-1  CPP   1,292 2.2
ALG02-2  CPP   1,576 2.2
ALG02-3  CPP   1,775 2.2
ALG02-4  CPP   1,704 2.2
ALG02-5  CPP   2,107 2.3.1
ALG02-6  CPP   6,154 2.3.1
ALG02-7  CPP     900 2.3.1
ALG02-8  CPP   1,825 2.3.1
ALG02-9  CPP   1,399 2.3.1
ALG02-10 CPP     835 2.3.2
ALG02-11 CPP   2,402 2.3.3
ALG02-12 CPP   1,326 2.3.1
ALG02-13 CPP   1,706 2.3.1
B02-1    CPP   4,114 2.2
B02-2    CPP   4,203 2.3.1
B02-31   CPP   4,014 2.3.1
B02-32   CPP   4,249 2.3.1
B02-4    CPP   4,243 2.3.2
B02-5    CPP   4,835 2.3.3
B02-6    CPP   5,997 2.3.3
B02-7    CPP   4,523 2.4
B02-8    CPP   3,171 2.3.1
B02-9    CPP   1,026 2.3.1
FUNC2-1  CPP   1,695 2.3.1
FUNC2-2  CPP     615 2.3.1
FUNC2-3  CPP     482 2.2
MAIN2-1  CPP   3,259 2.2
MAIN2-2  CPP   2,021 2.3.1
MAIN2-31 CPP   1,958 2.3.1
MAIN2-32 CPP   2,367 2.3.1
MAIN2-4  CPP   1,278 2.3.2
MAIN2-5  CPP   1,519 2.3.3
MAIN2-6  CPP   2,961 2.3.3
MAIN2-7  CPP     988 2.4
MAIN2-8  CPP   2,003 2.3.1
C2-1     H       309 2.2
C2-2     H       150 2.3.1
C2-3     H       167 2.3.1
C2-4     H       135 2.3.3
C2-5     H       280 2.3.3
C2-6     H       233 2.4
TABLE    TXT     108 2.3.1
41 file(s)    87,904 bytes

Directory of F:\BC\CH3
文件名 扩展名   字节   章节
ALG03-1  CPP     804 3.2.1
ALG03-2  CPP     842 3.2.1
ALG03-3  CPP   1,279 3.2.2
ALG03-4  CPP   1,386 3.2.3
ALG03-5  CPP   2,879 3.2.4
ALG03-6  CPP   2,133 3.2.5
ALG03-7  CPP   2,387 3.2.5
ALG03-8  CPP     406 3.3
ALG03-9  CPP   1,255 3.3
ALG03-10 CPP     846 3.3
ALG03-11 CPP   2,619 3.4.3
ALG03-12 CPP   3,324 3.5
ALG03-13 CPP   3,787 3.5
B03-1    CPP   1,683 3.1.2
B03-2    CPP   1,955 3.4.2
B03-3    CPP   1,674 3.4.3
B03-4    CPP     897 3.4.3
B03-5    CPP   1,327 3.1.2
B03-6    CPP   1,609 3.4.2
B03-7    CPP   1,908 3.4.3
B03-8    CPP   1,430 3.4.3
B03-9    CPP     890 3.4.3
FUNC3-1  CPP   1,430 3.2.4
FUNC3-2  CPP   1,793 3.2.5
FUNC3-3  CPP   2,537 3.5
MAIN3-1  CPP     778 3.1.2
MAIN3-2  CPP   1,052 3.4.2
MAIN3-3  CPP   1,362 3.4.3
MAIN3-4  CPP   1,216 3.4.3
MAIN3-5  CPP     734 3.1.2
MAIN3-6  CPP   1,081 3.4.2
MAIN3-7  CPP     812 3.4.3
MAIN3-8  CPP   1,426 3.4.3
C3-1     H       314 3.1.2
C3-2     H       194 3.4.2
C3-3     H       271 3.4.3
C3-4     H       445 3.4.3
C3-5     H       380 3.4.3
A        TXT   1,041 3.5
```

```
B          TXT   1,092 3.5
C          TXT     940 3.5
D          TXT   1,131 3.5
ED         TXT      27 3.2.3
43 file(s)     57,376 bytes

Directory of F:\BC\CH4
文件名 扩展名   字节   章节
ALGO4-1    CPP   2,079 4.3.2
ALGO4-2    CPP   5,971 4.4.1
ALGO4-3    CPP   6,212 4.4.2
ALGO4-4    CPP   3,250 4.4.2
BO4-1      CPP   4,307 4.2.1
BO4-2      CPP   4,815 4.2.2
BO4-3      CPP   6,583 4.2.3
MAIN4-1    CPP   1,918 4.2.1
MAIN4-2    CPP   1,565 4.2.2
MAIN4-3    CPP   2,101 4.2.3
C4-1       H       160 4.2.1
C4-2       H       139 4.2.2
C4-3       H       239 4.2.3
BOOKIDX    TXT     196 4.4.2
BOOKINFO   TXT     220 4.4.2
FILE       TXT      28 4.4.1
NOIDX      TXT      23 4.4.2
17 file(s)     39,806 bytes

Directory of F:\BC\CH5
文件名 扩展名   字节   章节
ALGO5-1    CPP   1,435 5.3.2
ALGO5-2    CPP     959 5.2
BO5-1      CPP   2,097 5.2
BO5-2      CPP   5,245 5.3.2
BO5-3      CPP   6,131 5.3.2
BO5-4      CPP  10,679 5.3.2
BO5-5      CPP   4,144 5.7.3
BO5-6      CPP   4,350 5.7.3
FUNC5-1    CPP   1,205 5.7.3
MAIN5-1    CPP   1,245 5.2
MAIN5-2    CPP   1,000 5.3.2
MAIN5-3    CPP   1,098 5.3.2
MAIN5-4    CPP   1,327 5.3.2
MAIN5-5    CPP   1,449 5.7.3
MAIN5-6    CPP   1,478 5.7.3
C5-1       H       409 5.2
C5-2       H       310 5.3.2
C5-3       H       445 5.3.2
C5-4       H       381 5.3.2
C5-5       H       443 5.5
C5-6       H       436 5.5
21 file(s)     46,266 bytes

Directory of F:\BC\CH6
文件名 扩展名   字节   章节
ALGO6-1    CPP   1,922 6.6.2
ALGO6-2    CPP   2,393 6.6.2
BO6-1      CPP   8,515 6.2.3
BO6-2      CPP   8,050 6.2.3
BO6-3      CPP   6,465 6.3.2
BO6-4      CPP   7,781 6.4.1
BO6-5      CPP   8,298 6.4.1
BO6-6      CPP   7,124 6.2.3
FUNC6-1    CPP     668 6.6.2
FUNC6-2    CPP     352 6.4.1
FUNC6-3    CPP   1,235 6.2.3
MAIN6-1    CPP   1,973 6.2.3
MAIN6-2    CPP   3,479 6.2.3
MAIN6-3    CPP   1,816 6.3.2
MAIN6-4    CPP   1,211 6.4.1
MAIN6-5    CPP   1,407 6.4.1
MAIN6-6    CPP   3,264 6.2.3
C6-1       H       236 6.2.3
C6-2       H       151 6.2.3
C6-3       H       277 6.3.2
C6-4       H       212 6.4.1
C6-5       H       149 6.4.1
C6-6       H       169 6.2.3
C6-7       H       242 6.6.2
24 file(s)     67,389 bytes

Directory of F:\BC\CH7
文件名 扩展名   字节   章节
ALGO7-1    CPP   2,517 7.4.1
ALGO7-2    CPP   1,876 7.4.3
ALGO7-3    CPP   2,962 7.4.4
ALGO7-4    CPP   1,806 7.5.1
ALGO7-5    CPP   4,026 7.5.2
ALGO7-6    CPP   2,768 7.6.1
ALGO7-7    CPP   1,518 7.6.2
ALGO7-8    CPP   1,346 7.4.3
ALGO7-9    CPP   2,470 7.6.2
ALGO7-10   CPP   1,021 7.3.2
ALGO7-11   CPP     841 7.3.2
BO7-1      CPP  15,915 7.2.1
BO7-2      CPP  13,970 7.2.2
BO7-3      CPP  10,295 7.2.3
BO7-4      CPP  11,524 7.2.4
FUNC7-1    CPP     385 7.5.1
FUNC7-2    CPP   1,054 7.6.2
MAIN7-1    CPP   1,414 7.2.1
MAIN7-2    CPP   1,462 7.2.2
MAIN7-3    CPP   1,244 7.2.3
MAIN7-4    CPP   1,168 7.2.4
C7-1       H       610 7.2.1
C7-2       H       588 7.2.2
C7-21      H       949 7.2.2
C7-3       H       531 7.2.3
C7-31      H     1,059 7.2.3
C7-4       H       530 7.2.4
F7-1       TXT     101 7.3.2
F7-2       TXT     101 7.3.2
MAP        TXT     641 7.6.2
30 file(s)     86,692 bytes

Directory of F:\BC\CH8
文件名 扩展名   字节   章节
ALGO8-1    CPP   7,504 8.3.1
ALGO8-2    CPP   5,423 8.4.2
ALGO8-3    CPP   1,963 8.5
C8-1       H       533 8.3.1
C8-2       H       509 8.4.1
C8-3       H       503 8.5
6 file(s)      16,435 bytes

Directory of F:\BC\CH9
文件名 扩展名   字节   章节
ALGO9-1    CPP   1,636 9.1.1
ALGO9-2    CPP     871 9.1.2
ALGO9-3    CPP   2,658 9.1.3
ALGO9-4    CPP     972 9.2.1
ALGO9-5    CPP     967 9.2.1
ALGO9-6    CPP   1,149 9.2.2
ALGO9-7    CPP   1,311 9.2.3
ALGO9-8    CPP   1,382 9.2.3
ALGO9-9    CPP   1,411 9.3.4
BO9-1      CPP   2,397 9.1.1
BO9-2      CPP   2,640 9.2.1
BO9-3      CPP   5,040 9.2.1
BO9-4      CPP   4,054 9.2.2
BO9-5      CPP   4,205 9.2.3
BO9-6      CPP   3,209 9.2.3
BO9-7      CPP   3,580 9.3.4
C9-1       H       168 9.1.1
C9-2       H       174 9.2.1
C9-3       H       644 9.2.2
C9-4       H       599 9.2.3
C9-5       H       718 9.2.3
C9-6       H       380 9.3.4
C9-7       H       141 9.1.1
C9-8       H       167 9.2.3
FUNC9-1    CPP     783 9.2.1
25 file(s)     41,256 bytes
```

```
Directory of F:\BC\CH10
文件名 扩展名   字节   章节
ALG010-1 CPP     818 10.2.1
ALG010-2 CPP   2,887 10.2.2
ALG010-3 CPP   1,608 10.2.3
ALG010-4 CPP     743 10.3
ALG010-5 CPP   1,142 10.3
ALG010-6 CPP   1,137 10.3
ALG010-7 CPP   1,225 10.4.1
ALG010-8 CPP   1,627 10.4.2
ALG010-9 CPP   1,580 10.4.3
ALG10-10 CPP   1,640 10.5
ALG10-11 CPP   4,621 10.6.2
BO10-1   CPP   2,143 10.2.1
BO10-2   CPP     644 10.3
C10-1    H       388 10.1
C10-2    H       449 10.2.2
C10-3    H       541 10.6.2
16 file(s)    23,193 bytes

Directory of F:\BC\CH11
文件名 扩展名   字节   章节
ALG011-1 CPP   3,123 11.3
ALG011-2 CPP   5,205 11.4
ALG011-3 CPP   1,597 11.4
ALG011-4 CPP      74 11
BO11-1   CPP   2,172 11.3
5 file(s)     12,171 bytes

Directory of F:\BC\CH12
文件名 扩展名   字节   章节
ALG012-1 CPP   4,226 12.2
1 file(s)      4,226 bytes

Directory of F:\TC
CH1             <DIR>
CH2             <DIR>
CH3             <DIR>
CH4             <DIR>
CH5             <DIR>
CH6             <DIR>
CH7             <DIR>
CH8             <DIR>
CH9             <DIR>
CH10            <DIR>
CH11            <DIR>
CH12            <DIR>
0 file(s)       0 bytes

Directory of F:\TC\CH1
ALG01-1  C              526
ALG01-2  C              504
ALG01-3  C              583
ALG01-4  C              282
BO1-1    C            1,549
MAIN1-1  C            1,604
C1       H              662
C1-1     H              174
8 file(s)      5,884 bytes

Directory of F:\TC\CH2
ALG02-1  C            1,398
ALG02-2  C            1,631
ALG02-3  C            1,862
ALG02-4  C            1,790
ALG02-5  C            2,220
ALG02-6  C            6,772
ALG02-7  C              941
ALG02-8  C            1,889
ALG02-9  C            1,437
ALG02-10 C              860
ALG02-11 C            2,524
ALG02-12 C            1,365
ALG02-13 C            1,759
BO2-1    C            4,347
BO2-2    C            4,385
BO2-31   C            4,185
BO2-32   C            4,423
BO2-4    C            4,476
BO2-5    C            5,005
BO2-6    C            6,319
BO2-7    C            4,708
BO2-8    C            3,305
BO2-9    C            1,064
FUNC2-1  C            1,764
FUNC2-2  C              640
FUNC2-3  C              491
MAIN2-1  C            3,332
MAIN2-2  C            2,057
MAIN2-31 C            1,989
MAIN2-32 C            2,407
MAIN2-4  C            1,299
MAIN2-5  C            1,551
MAIN2-6  C            3,077
MAIN2-7  C            1,000
MAIN2-8  C            2,040
C2-1     H              335
C2-2     H              179
C2-3     H              173
C2-4     H              145
C2-5     H              324
C2-6     H              249
TABLE    TXT             69
42 file(s)     91,786 bytes

Directory of F:\TC\CH3
ALG03-1  C              850
ALG03-2  C              881
ALG03-3  C            1,319
ALG03-4  C            1,435
ALG03-5  C            3,033
ALG03-6  C            2,244
ALG03-7  C            2,419
ALG03-8  C              416
ALG03-9  C            1,309
ALG03-10 C              877
ALG03-11 C            2,741
ALG03-12 C            3,459
ALG03-13 C            3,944
BO3-1    C            1,811
BO3-2    C            2,092
BO3-3    C            1,788
BO3-4    C              971
BO3-5    C            1,374
BO3-6    C            1,704
BO3-7    C            2,057
BO3-8    C            1,567
BO3-9    C              954
FUNC3-1  C            1,483
FUNC3-2  C            1,821
FUNC3-3  C            2,633
MAIN3-1  C              787
MAIN3-2  C            1,064
MAIN3-3  C            1,370
MAIN3-4  C            1,225
MAIN3-5  C              748
MAIN3-6  C            1,091
MAIN3-7  C              821
MAIN3-8  C            1,435
C3-1     H              351
C3-2     H              220
C3-3     H              294
C3-4     H              474
C3-5     H              405
A        TXT          1,041
B        TXT          1,092
C        TXT            940
D        TXT          1,131
ED       TXT             27
43 file(s)     59,698 bytes

Directory of F:\TC\CH4
ALG04-1  C            2,101
```

```
ALG04-2  C          6,076
ALG04-3  C          6,522
ALG04-4  C          3,407
BO4-1    C          4,418
BO4-2    C          5,112
BO4-3    C          6,884
MAIN4-1  C          1,920
MAIN4-2  C          1,584
MAIN4-3  C          2,127
C4-1     H            170
C4-2     H            156
C4-3     H            278
BOOKIDX  TXT          200
BOOKINFO TXT          223
FILE     TXT           28
NOIDX    TXT           23
17 file(s)    41,229 bytes

Directory of F:\TC\CH5
ALG05-1  C          1,493
ALG05-2  C            990
BO5-1    C          2,213
BO5-2    C          5,328
BO5-3    C          6,565
BO5-4    C         11,420
BO5-5    C          4,454
BO5-6    C          4,675
FUNC5-1  C          1,270
MAIN5-1  C          1,268
MAIN5-2  C          1,007
MAIN5-3  C          1,105
MAIN5-4  C          1,351
MAIN5-5  C          1,463
MAIN5-6  C          1,499
C5-1     H            440
C5-2     H            343
C5-3     H            483
C5-4     H            406
C5-5     H            480
C5-6     H            483
21 file(s)    48,736 bytes

Directory of F:\TC\CH6
ALG06-1  C          2,011
ALG06-2  C          2,514
BO6-1    C          8,872
BO6-2    C          8,499
BO6-3    C          6,948
BO6-4    C          8,238
BO6-5    C          8,698
BO6-6    C          7,500
FUNC6-1  C            692
FUNC6-2  C            361
FUNC6-3  C          1,312
MAIN6-1  C          2,095
MAIN6-2  C          3,509
MAIN6-3  C          1,876
MAIN6-4  C          1,218
MAIN6-5  C          1,415
MAIN6-6  C          3,291
C6-1     H            255
C6-2     H            164
C6-3     H            289
C6-4     H            237
C6-5     H            159
C6-6     H            182
C6-7     H            251
24 file(s)    70,586 bytes

Directory of F:\TC\CH7
ALG07-1  C          2,880
ALG07-2  C          1,913
ALG07-3  C          3,075
ALG07-4  C          1,889
ALG07-5  C          4,165
ALG07-6  C          2,858
ALG07-7  C          1,554
ALG07-8  C          1,434
ALG07-9  C          2,532
ALG07-10 C          1,000
ALG07-11 C            867
BO7-1    C         16,862
BO7-2    C         14,746
BO7-3    C         10,728
BO7-4    C         11,968
FUNC7-1  C            388
FUNC7-2  C          1,082
MAIN7-1  C          1,413
MAIN7-2  C          1,484
MAIN7-3  C          1,256
MAIN7-4  C          1,182
C7-1     H            658
C7-2     H            659
C7-21    H          1,043
C7-3     H            594
C7-31    H          1,169
C7-4     H            592
F7-1     TXT          101
F7-2     TXT          101
MAP      TXT          641
30 file(s)    90,834 bytes

Directory of F:\TC\CH8
ALG08-1  C          7,984
ALG08-2  C          5,724
ALG08-3  C          2,115
C8-1     H            589
C8-2     H            561
C8-3     H            543
6 file(s)     17,516 bytes

Directory of F:\TC\CH9
ALG09-1  C          1,702
ALG09-2  C            905
ALG09-3  C          2,780
ALG09-4  C          1,000
ALG09-5  C          1,004
ALG09-6  C          1,162
ALG09-7  C          1,333
ALG09-8  C          1,410
ALG09-9  C          1,455
BO9-1    C          2,517
BO9-2    C          2,796
BO9-3    C          5,302
BO9-4    C          4,255
BO9-5    C          4,416
BO9-6    C          3,415
BO9-7    C          3,777
FUNC9-1  C            804
C9-1     H            184
C9-2     H            190
C9-3     H            721
C9-4     H            675
C9-5     H            790
C9-6     H            405
C9-7     H            144
C9-8     H            170
25 file(s)    43,312 bytes

Directory of F:\TC\CH10
ALG010-1 C            827
ALG010-2 C          3,047
ALG010-3 C          1,685
ALG010-4 C            747
ALG010-5 C          1,199
ALG010-6 C          1,201
ALG010-7 C          1,267
ALG010-8 C          1,674
ALG010-9 C          1,664
ALG10-10 C          1,684
ALG10-11 C          4,847
BO10-1   C          2,323
BO10-2   C            666
```

```
C10-1    H             429
C10-2    H             508
C10-3    H             588
16 file(s)    24,356 bytes

Directory of F:\TC\CH11
ALG011-1 C           3,231
ALG011-2 C           5,416
ALG011-3 C           1,656
ALG011-4 C              76
B011-1   C           2,247
5 file(s)     12,626 bytes

Directory of F:\TC\CH12
ALG012-1 C           4,398
1 file(s)      4,398 byte

Directory of F:\Vc
SHORTEST       <DIR>
0 file(s)          0 bytes

Directory of F:\Vc\shortest
DEBUG          <DIR>
RES            <DIR>
MAPVC    TXT           840
SHORTEST CPP         2,091
SHORTE~1 CPP        11,452
STDAFX   CPP           210
SHORTEST DSW           539
RESOURCE H             686
SHORTE~1 H           2,355
STDAFX   H           1,054
SHORTEST H           1,346
SHORTEST RC          5,210
SHORTEST DSP         4,195
11 file(s)    29,978 bytes

Directory of
F:\Vc\shortest\Debug
SHORTEST EXE       114,753
1 file(s)    114,753 bytes

Directory of
F:\Vc\shortest\res
SHORTEST RC2           400
SHORTEST ICO         1,078
2 file(s)      1,478 bytes
```

注：光盘中文件的属性为“只读”。如要修改文件，可将文件拷贝到磁盘中，并将文件的“只读”属性去掉。