内容简介

本书和传统同类书籍的区别是除了介绍基本的数据结构容器如栈、队列、链表、树、二叉树、红黑树、AVL 树和图之外，引进了多任务；还介绍了将任意数据结构容器变成支持多任务的方法；另外，还增加了复合数据结构和动态数据结构等新内容的介绍。在复合数据结构中不仅介绍了哈希链表、哈希红黑树、哈希 AVL 树等容器，还介绍了复合数据结构的通用设计方法；在动态数据结构中主要介绍了动态环形队列、动态等尺寸内存管理算法。在内存管理中介绍了在应用程序层实现的内存垃圾回收算法、内存泄漏检查和内存越界检查的方法等。本书选取的内容均侧重于在实际中有广泛应用的数据结构和算法，有很好的商业使用价值。

本书大部分章节中都列举并介绍了应用实例，如用 AVL 树等容器实现的搜索引擎、用数组实现 HOOK 管理、用链表实现的短信息系统中的 CACHE 管理、用哈希表实现 WebServer 中的 CACHE 文件管理和用哈希 AVL 树实现抗 DoS/DDoS 攻击等。

书中重点介绍了软件的各种质量特性如时间效率和空间效率之间的关系，介绍了如何在各种质量特性间取得均衡的原则，并介绍了各种数据结构算法的应用场合和范围。

本书介绍的所有数据结构及算法都以不同复杂程度给出其编码实现。为了便于读者自学，每章末附有小结和思考练习题。

本书可供高校计算机及相关专业作为教学参考书，对从事软件开发与应用的科研人员、工程技术人员以及其他相关人员也具有较高的参考价值。

自序

软件的核心技术是什么？一个软件要做出来后很难模仿才能称之为拥有核心技术。软件上市后，只要使用一下便知道有哪些功能，所以功能性需求是非常容易模仿的。比较难模仿的有两个方面：一是软件设计，二是数据结构与算法。好的算法可以申请专利，用作保护知识产权和限制竞争对手的重要手段，由此可见算法在软件中的重要意义。软件研发人员和测试人员的最大区别在于研发人员在数据结构与算法方面要掌握得更好。

十年前，当我初次来到深圳开始职业软件生涯时，出于对数学的热爱，闲暇时经常看一些数据结构与算法方面的书籍和资料，常从其中受到启发。经过七年的日积月累后，忽然发现CPU的速度已快到达极限，多核CPU已投入实际使用，未来将是多核CPU的天下，对多任务编程提出了更高的要求，而目前数据结构与算法方面的书籍均没有涉足多任务方面，乃下定决心写作本书，在历时三年的写作过程中又有一些新的心得写入了本书中。

数据结构与算法已经成为软件开发工程师的必备的基础知识之一，在学校里，它已经成为计算机学科的重要课程，同时也成为许多其他专业的热门选修课，社会上大多数公司在招聘软件开发人员时必然会考察应聘人员数据结构与算法的掌握程度，并将此作为衡量应聘者水平的重要依据。本书是为那些已从事软件开发或即将从事软件开发的人员而写，也可以供专业人士参考。本书注重实践，注重软件的设计与实现，为软件开发人员向职业化方向发展和进一步提升打好基础。

本书重点讲解了多任务方面的内容，讲解了使数据结构支持多任务的算法。讲解了很多新的数据结构与算法，也讲解了数据结构的设计思想，如复合数据结构和动态数据结构的设计思想，本书中的哈希红黑树和哈希AVL树的设计就是复合数据结构设计的典范，而

动态等尺寸内存管理算法则是动态数据结构设计的代表。传统的垃圾收集算法在进行垃圾内存收集时应用程序会被停住，本书提供的算法在进行垃圾内存收集时不影响应用程序的运行，并且可以在应用程序层使用，效率很高。本书中的实例也都是选用比较热门的商业应用如搜索引擎、短信息系统、抗DoS攻击，WebServer等。

为了使更多的人能看懂此书，本书代码基本都使用C语言来实现，只有少数代码使用C++实现，因此具有C语言基础知识就可以阅读本书，本书的代码也很容易改写成C++，喜欢使用C++的朋友可以按照光盘中的版权申明要求将本书代码改成用C++实现，本书光盘附带的代码是可以免费使用和修改的。

本书最终得以完成有赖于许多人的帮助。首先感谢我的妻子和两岁多的儿子，在忘我投入写作的无数个周末和假日，儿子也渐渐地从襁褓长大到能四处探险。在写作过程中，他虽经常给我捣乱，但也给我带来很多欢乐，他的成长也伴随了这本书的成型。为了使我能完成自己的夙愿，妻子放弃了自己的工作，默默承担起照顾孩子和家庭的重担，有时还要帮我做一些书稿的录入工作。

感谢华中科技大学出版社对此书的重视，感谢编辑王红梅和刘勤老师等人，他们的工作大大改善了本书的质量。

感谢最初引导我进入软件行业的邓耀斌、唐玉天两人。感谢当年在 Santa Cruz 一起工作过的Neil Readshaw，九年前正是在他的指导和帮助下，我对多任务下的数据结构和算法的理解有了较大的飞跃。

本书的写作过程中参考了国内外许多有关数据结构与算法方面的文献，在此，我谨向那些卓有建树的专家、学者致以诚挚的感谢！本书的写作还得到了许多朋友的帮助和鼓励，温野、唐文宁、罗巍等人帮助校对过部分章节的内容，并给出了很好的改进建议。一些朋友多年来持续给予了我鼓励和帮助，包括章俊良、邓耀斌、张莉、洪建明等，章俊良博士还帮我查阅过关于哈密顿圈算法的一些论文资料，还有很多人请恕我在这里不能把他们一一列出，在此对他们一并表示感谢！

周伟明 2006年4月

目　　录

CONTENTS

1 绪　论

本章介绍了一些阅读后续内容所需的预备知识，包括 C 语言编程常见问题分析、任意类型数据处理和多任务的一些基础知识，软件设计简介等。

1.1 引言

本书建议读者：

有 C 语言基础知识，如果读过一本或多本其他数据结构与算法书籍则更佳。

本书多任务那一章要求有多任务编程的基础知识。

当我准备开始写这本书的时候，曾经和朋友谈论过，当时朋友很惊讶地说：“数据结构与算法发展到今天，已有的相关书籍如此之多，并且在 C++中还有一个标准模板库 STL，难道你还能写出新的东西吗？为什么不写其他方面的书呢？”的确，数据结构与算法发展到现在，一般软件工程师经常要用的数据结构与算法不知有多少人写过了。还有 STL，也有很多家公司实现了 Free 的源代码，甚至有些公司的编码实现被一些人奉为优秀代码的典范，在此情况下要写一本数据结构与算法的书籍确实不是一件容易的事情。在经过充分的权衡后，我还是决定要写这本关于数据结构与算法方面的书，主要基于以下考虑。

1) 目前数据结构与算法虽然看上去很成熟了，但随着软件技术越来越多地应用到工程实践中，新的应用会对它提出很多新的需求。特别是多核 CPU 的发展对多任务编程提出了更高的要求，这样数据结构的发展空间还是很大的，如 STL 目前还不能支持多任务。举个简单例子，大家可以考虑一下，如果一个链表有多个任务在操作，有些任务在做插入、删除操作，有些任务在遍历，能不能做到在遍历的过程中进行插入、删除操作？目前在 STL 中肯定是做不到的。本书会有专门的章节来讲述多任务下的数据结构与算法设计。

2) 现在虽然有很多 Free 的数据结构算法源代码，但是这些代码普遍存在的一个问题是可读性较差，没有注释，而且很多代码的命名风格很偏，即使是资深的专业人士看起来也很费力。本书的代码均追求高的可读性，体现良好的程序设计风格。

3) 本书探讨了一些复合数据结构的设计概念，如哈希链表和哈希红黑树。这些数据结构在服务器软件等方面会有很高的实用价值。

4) 本书会对所讲到的绝大部分数据结构与算法给出完整的源代码，即使是复杂的 AVL 树和红黑树也给出了完整代码。

5) 有一句网络名言叫“专业的程序员用 C 语言”，没错，本书代码主要是用 C 语言实现，只有少部分用 C++实现。其主要的设计思想还是采用面向对象的思想，C 语言实现的部分可以很方便地改写成 C++语言，但对大家学习来说可以屏蔽 C++语言复杂的语法，比 C++语言更容易理解和掌握，同时也让初学者明白面向对象与语言无关，并不是只有 C++、Java 这类支持面向对象的语言才可以写面向对象的程序。(其实许多初学者用 C++、Java 写的程序都不是面向对象的。)

6) 本书虽然是专门为专业人士而撰写的，但即使是在校学生也可以很容易看懂，只要有 C 语言基础知识就可以阅读本书。

7) C++标准委员会把模板引入 STL 中，使得 C++中的模板使用非常普及。实际应用中在内存受限的情况下，模板库开销过大使得人们将注意力集中到如何解决类型无关指针的问题。本书是用 void 指针来实现类型无关的，虽然理论上类型转换方面没有模板安全，但数据结构方面的类型转换实际上一般是不会有问题的，因为类型转换错误，执行马上就会异常，只要代码通过测试是不会有问题的。同时模板生成的执行代码庞大，对于内存受限系统，STL 的裁剪是一个非常令人头疼的问题。

对于模板库，或者任何一种技术都得看应用的场合。其实，任何一种思想和方法的滥用都会导致严重的后果，就像设计模式的滥用一样。比如，COM 的滥用导致现在软件质量的严重下降。COM 的设计可以说是软件史上最严重的一次设计失误，COM 的设计违反了软件设计的基本原理，但就这样一个设计居然被推广到整个业界使用，造成的危害实在太大了。我们可以看到，很多常用的软件的 BUG 越来越多，动不动就死掉或要发送错误报告什么的，很多情况估计都是拜 COM 所赐。虽然微软现在再也不敢提 COM 了，但它的副作用短期内仍然难以消除，特别是在中国，谈到设计时很多人现在还在言必称组件。

本书还有一些小的特点请读者在阅读本书时自行体会，限于时间和水平，本书的不足之处和缺陷相信也很多，请大家不吝赐教。

1.2 C 语言编程常见问题分析

C 语言中的一般语法和一些编程技巧在很多书里都讲过了，下面主要讲一些别的

书很少讲到、但是又非常重要的问题。这些都是作者在编程过程中总结出的一些经验教训，可以说职业程序员每天编程时都要遇到这些问题。

1.2.1 参数校验问题

在 C 语言的函数中，一般都要对函数的参数进行校验，但是有些情况下不在函数内进行校验，而由调用者在外部校验，到底什么情况下应该在函数内进行校验，什么情况下不需要在函数内进行校验呢？下列原则可供读者参考。

1) 对于需要在大的循环里调用的函数，不需要在函数内对参数进行校验。

例如链表的逐个遍历函数 void *List_EnumNext(LIST *pList)。

在链表的逐个遍历函数里，要不要对 pList 参数的合法性进行校验呢？答案是否定的。为什么呢？因为链表的逐个遍历函数通常是要在一个循环里使用的，比如一个链表有 10 000 个节点，逐个遍历就要遍历 10 000 次。如果上面的函数对参数 pList 进行了校验，那么对整个链表的逐个遍历过程将校验 10 000 次，不如由调用者在调用函数前校验一次就够了。因此，像这种可能频繁地被调用，且在外面校验只要校验一次就够的函数参数是不需要在函数内部进行校验的。

2) 底层的函数调用频度都比较高，一般不校验。

3) 对于调用频度低的函数，参数要校验。

4) 执行时间开销很大的函数，在参数校验相对整个函数来讲占的比例可以忽略不计的情况下，一般最好对函数参数进行校验。

5) 可以大大提升软件的稳定性时，函数参数要进行校验。

1.2.2 return 语句的问题

在函数里，使用 return 语句可能也是一个值得探讨的问题。有些人认为应该让函数的出口尽量的少，因此要减少 return 语句的使用；特别是在函数中有分配资源操作时，在之后的每个 return 语句里都需要进行对应的释放操作，在编程时很容易出现遗漏而出现资源泄漏。但是我们也知道，如果要减少 return 语句，又会带来另外一些问题。首先，很多情况下要减少 return 语句会增加编程的难度；其次，有时候减少了 return 语句又使得大括号嵌套的层数增加不少，使得代码行长度增加不少，很容易因超过 80 字符而使得代码阅读困难。

那么，到底什么情况下可以减少 return 语句，什么情况下又不能减少 return 语句呢？下列原则供读者参考。

1) 对参数进行校验，校验失败，一般要使用 return 语句，这样做可使程序逻辑清晰，阅读也方便，又减少了大括号嵌套的层数。

2) 对于函数内部的不同出错情况，要有不同的 return 语句；否则对外部调用者来

说，无法区分出错的真正原因。

3) 对于函数内部的同类出错情况，尽量只使用一个 return 语句，即尽量不要让两个 return 语句返回相同的返回值。

1.2.3 while 循环和 for 循环的问题

1. 两种循环在构造死循环时的区别

用 while 构造死循环时，一般会使用 while(TRUE)来构造死循环；而用 for 来构造死循环时，则使用 for(;;)来构造死循环。这两个死循环的区别是：while 循环里的条件被看成表达式，因此，当用 while 构造死循环时，里面的 TRUE 实际上被看成永远为真的表达式，这种情况容易产生混淆，有些工具软件如 PC-Lint 就会认为出错了，因此构造死循环时，最好使用 for(;;)来进行。

2. 两种循环在普通循环时的区别

对一个数组进行循环时，一般来说，如果每轮循环都是在循环处理完后才讲循环变量增加的话，使用 for 循环比较方便；如果循环处理的过程中就要将循环变量增加时，则使用 while 循环比较方便；还有在使用 for 循环语句时，如果里面的循环条件很长，可以考虑用 while 循环进行替代，使代码的排版格式好看一些。

1.2.4 if 语句的多个判断问题

在对参数进行校验时，经常需要校验函数的几个参数，是对每个参数都使用一个单独的 if 语句进行一次校验，还是多个语句都放在一个 if 语句里用逻辑或运算来进行校验呢？还是用实例来说明吧。

例 1-1 参数校验举例。

```
TABLE *CreateTable1( int nRow，int nCol )
{
    if ( nRow > MAX_ROWS )
    {
        return NULL；
    }
    if ( nCol >= MAX_COLS )
    {
        return NULL；
    }
    ……
}
```

```
TABLE *CreateTable2( int nRow，int nCol )
{
    if ( nRow > MAX_ROWS || nCol >= MAX_COLS )
    {
        return NULL；
    }
    ……
}
```

在 CreateTable1()和 CreateTable2()两个函数中，到底哪个写法更好呢？答案是第二种写法更优。为什么呢？因为第二种写法中，少了一个 return 语句，编译后，执行代码会比第一种写法小，执行时耗用的内存自然少了，其他方面的性能都是一样的。有兴趣的读者可以将上面两段代码反汇编出来分析一下。

1.2.5 goto 语句问题

goto 语句在结构化编程技术出来后，被当作破坏结构化程序的典型代表，可以说，在结构化程序设计年代，goto 语句就像洪水猛兽一样，程序员都唯恐避之不及；可后来在微软的一些例子程序中经常把 goto 语句用来处理出错，当出错时，goto 到函数要退出的一个 label 那里进行资源释放等操作。那么，goto 语句是不是只可以用于出错处理，其他地方都不可以用了呢？下列关于使用 goto 语句的原则可以供读者参考。

1) 使用 goto 语句只能 goto 到同一函数内，而不能从一个函数里 goto 到另外一个函数里。

2) 使用 goto 语句在同一函数内进行 goto 时，goto 的起点应是函数内一段小功能的结束处，goto 的目的 label 处应是函数内另外一段小功能的开始处。

3) 不能从一段复杂的执行状态中的位置 goto 到另外一个位置，比如，从多重嵌套的循环判断中跳出去就是不允许的。

1.2.6 switch …case 和 if … else if 的效率区别

许多有关 C 语言的书中都说过，switch … case 的效率比 if … else if 的效率高。其实这个结论并不完全正确，关键看实际代码，少数情况下，if…else if 写法如果得当的话，它的效率是高于 switch…case 的。

例 1-2 if…else if 语句测试举例。

```
void TestIfElse()
{
    unsigned int x ；
    srand( 1 )；        /* 初始化随机数发生器 */
```

```
    x = rand() % 100；
    if ( x == 0 )
    {
        ……
    }
    else if ( x == 1 )
    {
        ……
    }
    else
    {
        ……
    }
}
```

x 为 0～100 范围内的随机数，当 x 不为 0 和 1 时执行 else 分支，因此上面函数有 98%的概率是执行 else 分支的，而执行 if 分支和 else if 分支的概率各为 1%。在执行 else 分支时，要先执行 if 判断语句，再执行 else if 判断语句后才会执行 else 分支里的内容。因此执行 else 分支需要进行两次判断。如果我们按下列方式将 else 分支的执行改成放到 if 里面去执行，则效率将大大提高。

```
void TestIfElse( )
{
    unsigned int x ；
    srand( 1 )；        /* 初始化随机数发生器 */
    x = rand() % 100；
    if ( x > 1 )
    {
        ……
    }
    else if ( x == 0 )
    {
        ……
    }
    else
    {
        ……
    }
}
```

如果上面的代码用 switch...case 来实现，则效率不如上面的函数，读者可以试验一下。

1.3 任意数据类型处理

1.3.1 任意数据类型处理的设计方法

在数据结构与算法中，必然要涉及数据类型的问题，数据类型几乎是无限的集合，我们不可能为每种类型写一套代码，因此，要在一套代码中适应各种数据类型，C++的模板是一种实现方法。其实在 C 语言中，void 指针也是一种很好的方法，它可以指向任意类型的数据，只要使用时做一个简单的类型强制转换就可以了。

当任意类型的数据都用 void 指针来表示时，由于不知道调用者会传什么类型的数据过来，因此，对数据的操作在数据结构与算法代码内部是不知道的，只有外部的调用者才知道数据类型和对数据操作的方法。在数据结构与算法中对数据的操作通常有以下几种方法。

① 数据的比较；
② 数据的资源释放；
③ 数据的访问操作；
④ 数据的拷贝操作；
⑤ 数据的一些其他计算操作。

当外部调用者调用数据结构与算法提供接口时，如果接口要用到对数据的操作，则必须由调用者将数据操作方法告诉接口。在 C 语言中，一般都是通过回调函数来实现。我们可以把上面的数据比较操作定义成如下回调函数。

```
/**   通用类型数据比较函数
   @param   void *pData1——要比较的第 1 个参数
   @param   void *pData2——要比较的第 2 个参数
   @return    INT——小于 0 表示 pData1 小于 pData2；等于 0 表示 pData1 等于 pData2；
                       大于 0 表示 pData1 大于 pData2
*/
typedef INT (*COMPAREFUNC) ( void *pData1，void *pData2 )；
```

比如在排序时，由于不知道数据类型，我们必须知道如何比较数据的大小。因此用户要定义一个比较函数。比较函数的原型与上面 COMPAREFUNC 函数指针的定义一样，调用排序函数时，将比较函数传给排序函数作为参数，这样，在排序函数里面就知道如何比较数据了。

1.3.2 任意数据类型处理的实例

例 1-3 设计一个简单的排序函数，排序的数据放在数组中，且数据类型可以是任意的。

```
void Sort( void **ppData，int nLen，COMPAREFUNC Compare)
{
    int i，j；
    void   *pTemp；
    for ( i = 0； i < nLen； i++ )
    {
        for ( j = i+1； j <nLen； j++ )
        {
            if ( (*Compare) (ppData[i]，ppData[j] ) > 0 )
            {
                pTemp = ppData[j]；
                ppData[j] = ppData[i]；
                ppData[i] = pTemp；
            }
        }
    }
}
```

例 1-4 调用 Sort()函数来排序一个字符串数组。

```
INT StrCmp( void *p1，void *p2)
{
  return strcmp( (char *)p1，(char *)p2)；
}

void main()
{
#define MSG_COUNT   6
    char *pszMsg[MSG_COUNT] = {"ah"，"bee"，"sea"，"degree"，"chineses"，"quit"}；
    Sort( pszMsg，sizeof(pszMsg)，StrCmp)；
    for ( int i = 0； i < MSG_COUNT； i++ )
    {
        printf("%s\n"，pszMsg[i] )；
    }
}
```

1.3.3 任意数据类型处理的回调函数封装

下面给出本书要用到的 7 个任意类型处理的回调函数定义。

```
/** 通用类型数据比较函数
   @param void *pData1——要比较的第 1 个参数
   @param void *pData2——要比较的第 2 个参数
   @return INT—— 小于 0 表示 pData1 小于 pData2；等于 0 表示 pData1 等于 pData2；
                  大于 0 表示 pData1 大于 pData2
*/
typedef INT (*COMPAREFUNC) ( void *pData1，void *pData2 )；

/** 通用类型数据释放函数
   @param void *pData——要释放的数据
   @return——无
*/
typedef void (*DESTROYFUNC) ( void *pData )；

/** 通用类型数据的遍历执行函数
   @param void *pData——要操作的数据指针
   @return void——无
*/
typedef void (*TRAVERSEFUNC)( void *pData)；

/** 通用类型数据的遍历执行函数
   @param void *pData——要操作的数据指针
   @return void——无
*/
typedef INT (*VISITFUNC)(void *pData)；

/** 通用数据拷贝函数
   @param void *pData——要拷贝的数据
   @return void *——成功返回拷贝的数据，失败返回 NULL
*/
typedef void *(*COPYFUNC)(void *pData)；

/** 基数排序的获取关键字转换成序号的函数
   @param void *pData——关键字指针
   @param UINT *uKeyIndex——关键字的位数
   @return UINT——关键字转换后的序号
*/
```

```
typedef UINT (*GETKEYFUNC)( void *pData，UINT uKeyIndex )；

/** 计算哈希值的回调函数
  @param void *pKey——要计算哈希值的关键词
  @param UINT uBucketNum——哈希表中的 bucket 数组大小
  @return UINT——返回计算出的哈希值，一般用作哈希表 bucket 数组下标
*/
typedef UINT (*HASHFUNC)(void *pKey，UINT uBucketNum)。
```

1.4 多任务介绍

1.4.1 多任务简介

多任务编程可以说是未来编程的发展趋势，目前单核 CPU 的速度已接近极限，今后更多的是采用多核 CPU 的技术，在多核 CPU 情况下，软件必须按多任务设计才能充分利用多核 CPU 的优势。

什么是任务？任务是操作系统中的概念，进程、线程都是任务。操作系统中，与任务相关的另外一个概念便是信号量，如互斥信号量便是一种最常用的信号量。

多个任务在同时运行时，必然涉及共享资源的问题，如多个任务都要访问同一全局变量或内存，这时必须要使用互斥信号量来保证在同一时刻只允许一个任务访问共享资源。

1.4.2 锁的概念

对于多个任务访问共享资源的问题，可以举一个形象的例子来说明。

例如，有几个人想要进入同一个房间，已知同一时间内房间内只允许有一个人在里面，为了保证这个前提，一个最简单的方法便是给房门加一道锁；当一个人进去后，由这个进去的人锁上门，这样其他人就进不去了；当这个人出来后，再开锁，这样又可以有一个人进去了。多个任务访问共享资源就类似这个例子，这里说的锁其实等同于互斥信号量。

为了叙述方便，本书后面关于互斥信号量方面的概念统一代之以锁的概念。用上锁、解锁来表示对应的操作。

1.4.3 Windows 下常用多任务操作函数

Windows 下的多任务操作有很多种，常用多任务操作函数见表 1-1、表 1-2。

表 1-1　Windows 下信号量常用操作函数

函　数　名	描　　述
CreateMutex ()	创建一个互斥信号量
WaitForSingleObject ()	等待一个信号量
ReleaseMutex ()	解锁一个互斥信号量
CloseHandle ()	关闭句柄
CreateEvent()	创建一个事件
OpenEvent()	打开一个事件
SetEvent()	设置一个事件
ResetEvent()	重新设置一个事件
InitializeCriticalSection ()	初始化一个临界区对象
EnterCriticalSection()	进入一个临界区
TryEnterCriticalSection()	试着进入一个临界区
LeaveCriticalSection()	离开一个临界区
DeleteCriticalSection()	删除一个临界区对象
CreateSemaphore()	创建一个信号量
ReleaseSemaphore()	给信号量增加计数

表 1-2　Windows 下线程的常用操作函数

函　数　名	描　　述
CreateThread ()	创建线程
SuspendThread ()	挂起线程
ResumeThread ()	继续运行一个挂起的线程
TerminateThread ()	终止一个线程的运行
ExitThread ()	退出线程
_beginThread()	创建线程，和 CreateThread()类似
_endThread()	终止由_beginThread()创建的线程

1.4.4 Linux/Unix 下常用多任务操作函数

Linux/Unix 下的多任务操作有很多种,这里主要列出 POSIX 标准里的常用函数(见表 1-3、表 1-4)。

表 1-3 pthread 常用的线程函数

函 数 名	描 述
pthread_create()	创建一个线程
pthread_exit()	终止线程的运行
pthread_kill()	杀掉一个指定的线程
pthread_join()	等待线程执行完成

表 1-4 pthread 互斥信号量的常用函数

函 数 名	描 述
pthread_mutex_init()	初始化一个互斥信号量
pthread_mutex_destroy()	销毁一个互斥信号量
pthread_mutex_lock()	上锁，相当于 Windows 中的 WaitForSingleObject()函数
pthread_mutex_trylock()	试着上锁，如果不成功不会阻塞
pthread_mutex_unlock()	释放锁，相当于 Windows 中的 ReleaseMutex()函数

1.4.5 VxWorks 下常用多任务操作函数

VxWorks 下的多任务操作有很多种，常用多任务操作函数见表 1-5、表 1-6。

表 1-5 VxWorks 下常用任务操作函数

函 数 名	描 述
taskInit()	初始化一个新任务
taskActivate()	激活一个已初始化的任务
taskSpawn ()	创建并激活一个新任务
taskSuspend ()	挂起一个任务
taskResume ()	恢复一个任务
taskDelay()	任务延时，单位为 tick(1 个 tick 为 10 ms)
nanoSleep()	任务延时，单位为 ns
taskLock()	禁止任务调度器进行任务抢占调度

续表

函 数 名	描 述
taskUnlock()	恢复任务抢占调度
taskSafe()	保护调用任务不被删除
taskUnsafe()	解除任务，删除保护
taskDelete()	终止一个指定的任务，释放任务占用的内存(任务堆栈和任务控制块)
exit()	终止当前任务的执行，释放任务占用的内存(任务堆栈和任务控制块)

注：本表引自《嵌入式实时操作系统 VxWorks 及其开发环境 Tornado》。

表 1-6 VxWorks 常用信号量函数

函 数 名	描 述
semBCreate()	分配并初始化一个二进制信号量
semMCreate()	分配并初始化一个互斥信号量，相当于 Windows 中的 CreateMutex() 函数
semCCreate()	分配并初始化一个计数信号量
semDelete()	终止并释放一个信号量
semTake()	获取一个信号量，相当于 Windows 中的 WaitForSingleObject()函数
semGive()	释放一个信号量，相当于 Windows 中的 ReleaseMutex()函数
semFlush()	解锁所有等待该信号量的任务

1.4.6 多任务函数的封装

本书主要用到一些信号量方面的函数，以 Windows 操作系统为例来封装以下多任务方面的信号量函数。

```
#if defined(_WIN32)

#define LOCK            HANDLE
#define EVENT           HANDLE
#define SEMAPHORE       HANDLE

#define LockCreate()    CreateMutex(NULL，FALSE，NULL)
#define Lock(x)         (void)WaitForSingleObject((x)，INFINITE)
#define Unlock(x)       (void)ReleaseMutex(x)
#define LockClose(x)    (void)CloseHandle(x)

#define EventCreate()   CreateEvent(NULL，TRUE，FALSE，NULL)
#define WaitEvent(x)    (void)WaitForSingleObject((x)，INFINITE)
```

```
#define SendEvent(x)          (void)SetEvent(x)
#define EventClose(x)         (void)CloseHandle(x)

#define SemaCreate(x，y)      CreateSemaphore(NULL，x，y，NULL)
#define SemaWait(x)           WaitForSingleObject(x，INFINITE)
#define SemaRelease(x，y)     ReleaseSemaphore(x，y，NULL)
#define SemaClose(x)          CloseHandle(x)
#endif
```

1.5 软件设计简介

1.5.1 软件设计历史简述

软件设计到目前为止已经历了两个大的发展阶段。第一阶段是以结构化设计技术为代表的时代。结构化技术主要是为了解决软件程序设计而提出的，随着软件规模越来越大，结构化的技术不能适应软件的发展，于是产生了面向对象的方法。第二阶段便是以面向对象设计方法为代表的时代。从 20 世纪 80 年代末到现在，面向对象的方法一直在发展中，特别是 20 世纪 90 年代，面向对象的设计方法风靡世界。虽然面向对象的方法发展了这么多年，但是现在还可以看到，即使是一些著名大公司开发的软件依然达不到很高的质量，经常出现版本升级越高 BUG 愈多的情况。也许有人会问，这不是在危言耸听吧？其实读者只要看看自己手头经常在用的操作系统和办公软件就知道为什么说这句话了。

为什么会出现上面这种情况呢？是设计人员水平不够，没有领会面向对象的基本思想方法？还是编码人员水平不够？还是测试人员不负责任？如果是这样的话，为什么他们在做这些软件的早期版本时，质量问题没有这样多呢？难道那些工程师做的时间越久，水平下降了不成？显然不能简单地用前面的理由来解释这个问题，唯一的解释就是，当软件大到一定规模时，面向对象的方法也无能为力，就像当年结构化技术的局限性一样，面向对象的方法也有它的局限性。首先，作为一种方法，必然有它的使用范围，如果不考虑使用范围而不加限制地使用，必然会带来很多问题。特别是近几年，在面向对象方法基础上又产生了设计模式之类的技术，更应规范它的使用范围。像前面提到的 COM 一样，实际上是一种设计模式，但是被不加限制地推广给用户使用，而大部分用户水平还没有上升到懂得在什么情况下该使用什么设计模式那么高的层次，后果便是设计模式被滥用，于是我们看到大型软件的质量是越来越糟糕。

那么有人会问，既然面向对象的方法不能适应这些大型软件的开发，那么有没有其他方法来设计大型软件，使得软件的质量得以提升呢？要回答这个问题可不是一件容易的事情，这个问题必须等到软件设计发展的第三阶段才能彻底解决。那么，软件

设计发展的第三阶段主要是什么内容呢？能告诉读者的是，软件设计发展的第三阶段现在正处于萌芽期。在这里，如果把第一阶段的结构化技术和第二阶段的面向对象方法纳入“微观设计学”的范畴，那么第三阶段的内容便属于“宏观设计学”的范畴。也许面向对象的忠实支持者看了这个结论会难以接受，但事实上面向对象的设计方法确实只能解决微观层面的问题，不能解决宏观层面的问题。

“宏观设计学”会解决哪些问题呢？第一个要解决的问题便是软件设计原理，可以说软件设计发展这么多年，软件设计原理一直没有被主流研究人员当作一门学问来研究，实在是一件令人遗憾的事情；第二个要解决的问题便是在软件设计原理基础上产生宏观设计学的基本方法。前面已经说过，“宏观设计学”正处于萌芽期，但实际中已经可以看到一些实践者和理论者在向这个方向迈进，如《敏捷软件开发》一书的作者就提到过许多设计原则，不过由于局限性，作者仍然把它称为面向对象的设计原则。实际上，面向对象是作为一种方法出现的，和原则完全是两码事，所以在设计原则前面加上面向对象来修饰实在太牵强了。像设计模式前也加上面向对象的修饰词，有些勉强，事实上，并不是所有的设计模式都是用面向对象方法进行设计的，有少数设计模式可以纳入宏观设计学的范畴。

本书主要介绍数据结构与算法，软件规模很小，属于微观设计学的范畴，因此本书所用的设计方法主要为结构化方法和面向对象方法。

1.5.2 微观设计学原理简介

本书主要运用微观设计学的方法进行软件设计。研究软件设计原理方面的书籍目前很少，还没有专门的书籍来讲解微观设计学原理。其实，很多有经验的软件工程师都多少了解一些微观设计学的原理，只是这方面的研究目前并不是特别成熟，所以，只好在这里尝试着将微观设计学的基本原理给读者简单介绍一下。

与微观设计学的一些基本原理相关的知识读者经常挂在口头上，比如经常会说到的可维护性、可复用性、可测试性等，这些特性通常被称作质量特性，主要包括下列内容。

1) 一致性：指设计的每一个功能都能正确实现对应的需求。

2) 完备性：指软件所有的需求规格都被设计所覆盖。

3) 可维护性：可维护性是一个很广泛也很复杂的概念，在设计和编码中有不同的理解，在设计中的理解主要是指可扩展性。

4) 可复用性：主要是指设计的东西可以被复用，一般在设计中主要是指接口可以被重用。

5) 可测试性：指设计的对象可以很容易被测试验证。

6) 简单性：指设计的对象可以很容易地被实现。

7) **效率：指设计的对象在运行时的执行时间和运行时所消耗空间方面的效率。执行时间效率就是程序执行时的速度，空间效率则包括所消耗内存大小和内存使用的效率。消耗内存的大小包括两部分，一是程序执行代码本身所消耗的内存；另一个就是程序运行时动态占用的内存大小。内存使用效率主要是指内存分配释放的效率，影响它的因素主要有内存碎片和内存管理算法，如垃圾收集算法，在释放时是自动回收的，回收时程序要被挂起直到垃圾内存回收完毕，这些都是空间效率方面要考虑的因素。**

一个好的设计应该能统合综效地体现上述质量特性。在实际中，上述质量特性间可能经常会有冲突，一个统合综效地体现上述质量特性的设计，就是采用最有效的方法去实现上述质量特性要求及解决各种特性间冲突的设计。比如，可维护性有可能和效率产生冲突，到底是优先满足可维护性还是优先满足效率呢？也许很多初学者马上会回答：“当然是优先满足可维护性了。”这种回答显然是不完全正确的，虽然大部分情况下可维护性比效率重要，但是确实在一些情况下效率比可维护性更重要。比如一个服务器软件，可能有成千上万个用户在同一时间内访问它，那么它的响应客户端的某些对全局效率有重大影响的模块的效率就非常重要了，这时为了提高这些模块的效率，你不得不牺牲一些可维护性及其他特性。

2 数　组

本章主要介绍使用数组实现的栈、队列等容器，其中重点介绍了动态环形队列，并将其和普通环形队列、STL 中的动态队列进行了对比分析；进而介绍了时间效率和空间效率的原则；之后讲解了排序表，并在排序表上实现了快速排序算法和二分查找算法；最后介绍了一个使用数组管理 HOOK 功能的实例。

数组是 C 语言设计中常用的元素，本章主要介绍怎样用数组来实现栈和队列的操作，同时利用数组来实现快速排序和二分查找算法。

2.1 栈

2.1.1 栈的基本概念

栈是一种最常用的操作，采用后进先出的方式，当栈为空时执行弹出操作会操作失败；当栈满时压入数据会失败。

本节使用数组来实现一个栈的操作，通常栈的操作主要有以下几个方面。

① 创建栈；

② 将数据压入栈中；

③ 从栈中弹出数据；

④ 释放栈。

栈的操作如图 2-1 所示。

图 2-1 中显示的是栈未满时的操作情况，当数据全部弹出时，栈为空；当栈满时，

无法再压入数据，可以通过将数组大小增加一倍的方式来增加栈中的最大元素个数，以便于插入新的数据，如图 2-2 所示。

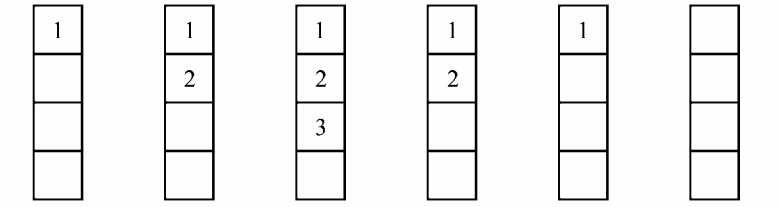


图 2-1　栈的操作

要实现数据类型的任意性，可用 void 指针来实现，所以栈中用一个 void 指针数组来记录栈中的数据。由于用指针来表示数组时，可以动态决定数组的大小，有很好的灵活性，因此应该用一个 void 双指针来记录栈中的数据。另外，要知道栈顶位置，还要知道栈的最大尺寸，栈的最大尺寸也就是数组的大小，因此，可以用以下结构体来描述栈。

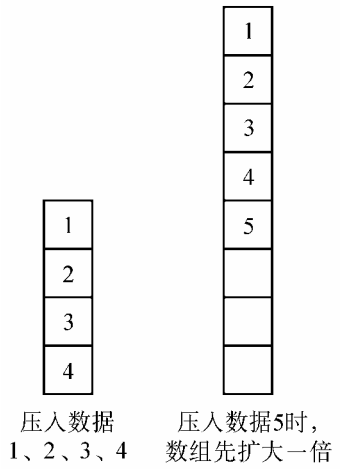


图 2-2　通过扩大数组来调整栈的大小

```
typedef struct STACK_st {
    void **ppBase；
        /* 用来记录任意类型数据的数组 */
    UINT uTop；
        /* 栈顶位置 */
    unsigned uStackSize；
        /* 栈的最大尺寸,也就是数组的大小 */
} STACK；
```

2.1.2　栈的编码实现

在进行栈的操作时，是从一头进并从同一头出的，从数组的尾部进出数据，不会移动前面的数据，所以压入操作和弹出操作都在数组的尾部也就是在数组的 uTop 位置进行，这样实现起来可以充分利用数组操作的高效性，避免移动数据。

栈操作的主要代码如下。

```
/** 创建一个栈
    @param UINT uStackSize——栈的大小
```

```
    @return STACK *——成功返回栈指针；失败返回 NULL
*/
STACK *Stack_Create(UINT uStackSize)
{
    STACK *pStack；
    if ( uStackSize == 0 )
    {
        return NULL；
    }
    pStack = (STACK *)malloc( sizeof(struct STACK_st) )；
    if ( pStack != NULL )
    {
        pStack->ppBase = (void **)malloc( uStackSize *sizeof(void *) )；
        if ( pStack->ppBase == NULL )
        {
            free( pStack )；
            pStack = NULL；
        }
        else
        {
            pStack->ppBase[0] = NULL；
            pStack->uTop = 0；
            pStack->uStackSize = uStackSize；
        }
    }
    return pStack；
}

/** 栈的释放函数，它会将栈中剩余的未弹出数据释放
    @param STACK *pStack——栈指针
    @param DESTROYFUNC DestroyFunc——数据释放回调函数
    @return void——无
*/
void Stack_Destroy(STACK *pStack，DESTROYFUNC DestroyFunc)
{
    if ( pStack != NULL )
    {
        if ( pStack->ppBase != NULL && DestroyFunc != NULL )
        {
```

```
            UINT i；
            for ( i = 0； i < pStack->uTop； i++ )
            {
                if ( pStack->ppBase[i] != NULL )
                {
                    (*DestroyFunc)(pStack->ppBase[i])；
                }
            }
            free( pStack->ppBase )；
        }
        free( pStack )；
        pStack = NULL；
    }
}

/** 栈的弹出函数，弹出栈顶数据，弹出的数据需要调用者自行释放
    @param STACK *pStack——栈指针
    @return void *——成功返回栈顶数据；栈为空则返回 NULL
*/
void *Stack_Pop( STACK *pStack )
{
    void*pData；
    if ( pStack == NULL || pStack->uTop == 0 )
    {
        return NULL；
    }
    pStack->uTop－= 1；
    pData = pStack->ppBase[pStack->uTop]；
    return pData；
}

/** 压栈函数，将数据压入栈顶，数据可以为空
    @param STACK *pStack——栈指针
    @param void *pData——要压入的数据，可以为空
    @return INT——成功返回 CAPI_SUCCESS；失败返回 CAPI_FAILED
*/
INT Stack_Push( STACK *pStack，void *pData )
{
    if ( pStack == NULL )
```

```
    {
        return CAPI_FAILED；
    }

    /* 判断栈是否为满，如果满了则将栈空间增大一倍 */
    if ( pStack->uTop >= pStack->uStackSize－1 )
    {
        pStack->ppBase = (void **)realloc( pStack->ppBase，
            ( pStack->uStackSize *2 ) *sizeof( void *) )；
        if ( pStack->ppBase == NULL )
        {
            return CAPI_FAILED；
        }
        pStack->uStackSize *= 2；
    }
    pStack->ppBase[pStack->uTop] = pData；
    pStack->uTop += 1；
    return CAPI_SUCCESS；
}

/** 判断栈是否为空
    @param STACK *pStack——栈指针
    @return INT——0 表示为空；非零表示非空
*/
INT Stack_IsEmpty( STACK *pStack )
{
    return pStack-> uTop；
}
```

本节用数组实现了栈的操作，当栈中元素超过数组的大小时，通过将数组扩大一倍的方式来消除数组大小固定的限制，当然也可以用后面讲到的链表方式来实现栈的操作。用数组实现栈的操作比较简单，效率也很高，缺点是当栈中元素较多并超过数组大小时，将数组扩大一倍需要消耗比较长的时间，分时性能不如链表。

2.1.3 多任务栈的实现

前面讲的栈的实现是单任务情况下的实现，即只有一个任务对栈进行操作，如果有多个任务要同时进行栈操作，必须要给栈加上锁保护才能避免重入问题。

很多人在编多任务程序时，喜欢在调用的地方先上锁，调用完后再解锁。采用这

种方法编的程序很难维护，正确的方法是在这个函数里直接进行上锁及解锁操作，或者另外写一个单独函数，在这个函数里进行上锁，调用对应函数后再解锁，其他模块只要调用这个具有上锁和解锁操作的函数就可以了。

如对本节的栈操作函数，欲将其变成支持多任务，可以做如下设计。

```
typedef struct MSTACK_st {
    STACK *pStack；
    LOCK pLock；
} MSTACK；
```

对栈的操作只需对要进行读写的数据进行上锁保护就可以实现多任务下的操作，为了方便，可以复用栈的代码，栈的各种操作实现编码如下。

```
/** 多任务栈的创建函数
    @param UINT uStackSize——栈的大小
    @return MSTACK *——成功返回栈指针；失败返回 NULL
*/
MSTACK *MStack_Create(UINT uStackSize)
{
    MSTACK *pMStack = (MSTACK *)malloc(sizeof(MSTACK))；
    if ( pMStack != NULL )
    {
        pMStack->pStack = Stack_Create(uStackSize)；
        if ( pMStack->pStack != NULL )
        {
            /* 创建锁 */
            pMStack->pLock = LockCreate()；
            if ( pMStack->pLock != NULL )
            {
                return pMStack；
            }
            free(pMStack->pStack)；
        }
        free(pMStack)；
    }
    return NULL；
}

/** 多任务栈的释放函数
```

```
    @param MSTACK *pMStack——栈指针
    @param DESTROYFUNC DestroyFunc——数据释放回调函数
    @return void——无
*/
void MStack_Destroy(MSTACK *pMStack，DESTROYFUNC DestroyFunc)
{
    if ( pMStack != NULL )
    {
        Lock(pMStack->pLock)；
        Stack_Destroy(pMStack->pStack，DestroyFunc)；
        LockClose(pMStack->pLock)；
        free(pMStack)；
    }
}

/** 多任务栈的弹出操作函数
    @param MSTACK *pMStack——栈指针
    @return void *——弹出的数据指针
*/
void *MStack_Pop( MSTACK *pMStack )
{
    void *pData；
    Lock(pMStack->pLock)；
    pData = Stack_Pop(pMStack->pStack)；
    Unlock(pMStack->pLock)；
    return pData；
}

/** 栈的压入操作函数
    @param MSTACK *pMStack——栈指针
    @param void *pData——数据指针
    @return INT——返回 CAPI_SUCCESS 表示成功；返回 CAPI_FAILED 表示失败
*/
INT MStack_Push( MSTACK *pMStack，void *pData )
{
    INT nRet；
    Lock(pMStack->pLock)；
    nRet = Stack_Push(pMStack->pStack，pData)；
    Unlock(pMStack->pLock)；
```

```
    return nRet；
}

/** 判断栈是否为空
    @param MSTACK *pMStack——栈指针
    @return INT——0 表示为空；1 表示非空
*/
INT MStack_IsEmpty( MSTACK *pMStack )
{
    INT nRet；
    Lock(pMStack->pLock)；
    nRet = Stack_IsEmpty(pMStack->pStack)；
    Unlock(pMStack->pLock)；
    return nRet；
}
```

2.2 队列

2.2.1 队列的基本概念和接口

队列是一种先进先出(first-in-first-out，FIFO)的数据结构类型，它和栈的区别是：栈是后进先出，而队列的进出方向刚好和栈相反。一般来讲，添加新元素的那一端被称作队尾，而弹出元素的那一端被称作队首。

根据队列的操作特性，一个队列至少应该提供下列接口。

① 创建一个空的队列；
② 判断队列是否为空；
③ 判断队列是否为满；
④ 进队操作(向队列尾部添加元素)；
⑤ 出队操作(从队列中弹出头部元素)；
⑥ 释放整个队列。

其中第 5 个接口多数情况下被分解成两个接口，即获取队首元素接口和删除队首元素接口，分解成两个接口的好处是功能上更加完整，具有更好的扩展性。其缺点是队列的出队操作需要调用获取队首元素和删除队首元素两个接口来完成，效率上会受影响，使用起来也没有直接用一个弹出队首元素的接口方便。本书中对出队操作只实现一个弹出队首元素的接口。

从队列的特性我们知道队列是一个线性表，并且其中的元素是按进队的先后次序排列的。数组是一个连续的空间，操作起来效率非常高，能不能用数组来实现队列的

操作呢？本节将介绍三种用数组来实现队列的方法：第一种是普通的环形队列(Queue)，其中还会讲解如何将环形队列由固定长度队列变成可以动态增长的队列；第二种是 HP 实现的 STL 中的动态队列(STL::deque)；第三种是作者自己设计的动态环形队列(DeQue)。最后会对三种队列的时间效率、空间效率和性能方面进行比较，明确各种队列的适用范围，并结合本节内容讲解时间效率和空间效率的一般性优先原则，供读者设计其他商业性应用模块时作为参考。

2.2.2 环形队列(Queue)

1. 数组实现队列的方法

数组的特点是插入或删除数据不在尾部时效率特别低，需要移动大量的数据，这是首先要解决的问题。对于栈来说，由于进出都在同一端，可以在数组的尾部进行插入和删除来实现栈的 push 和 pop 操作，但队列就不能在数组的尾部同时进行插入和删除操作，因为那样无法满足队列的先进先出要求，因此，队列的插入和删除必须在数组的两端分别进行才能实现先进先出的要求。

如果插入在尾部进行，删除就在头部进行，那么删除后，如何避免移动大量的数据呢？可以与栈的实现一样，用两个指针来指向队列的头部和尾部，在实际删除过程中，只移动头指针和尾指针，不移动整个数组的数据，保证了插入和删除的高效性。

2. 环形队列实现动态增长及满空条件判断

随之而来的问题是，队列的插入和删除操作比较频繁，如果插入的总数超过数组的大小，如何处理呢？由于很多数据被删除，实际上可以把数组看成一个环形队列，到达数组最大边界，尾指针重新指向数组头部。但这又引入了另外一个问题，尾指针可能在头指针的前面，如何去判断队列为空和队列为满的情况呢？

先讨论队列为满的情况，其实不管尾指针在头指针的前面还是后面，当队列为满时，在环形队列里，头指针刚好在尾指针的下一个位置上，当头指针在数组的最大边界时，它的下一个位置为数组的头部，即第 0 个位置，其他情况下，下一个位置是指刚好比它大 1 的位置，因此，判断队列是否为满的依据就是尾指针的下一个位置是否为头指针的位置。

再来讨论队列为空的情况，其实队列为空时头指针和尾指针完全重叠，所以只要判断头指针和尾指针是否相等便知道队列是否为空。

一个用数组实现的环形队列操作如图 2-3 所示。

3. 关于队列空间使用的说明

从图 2-3 可以看出，当队列为满时实际上还有一个空间未使用，这样做的好处是比较便于判断队列是空还是满。如果将所有空间都用上，则队列为满和为空时，头指

针和尾指针都重叠，这时需要通过判断头指针指向的数据是否为空或用其他方法来判断队列为满还是为空，增加了一次判断。由于队列操作本来就很简单，增加这样的判断会使操作效率降低很多，而且增加判断还会增加执行代码的大小，执行代码同样是需要消耗内存空间的，所以留一个元素空间不用并不等于浪费了空间，从全局考虑不仅没有浪费空间还提高了程序效率。另外队列如果空间设得稍微大一点的话，比如 1 024 个元素，未用的一个空间只占 1/1 024，几乎可以忽略不计。

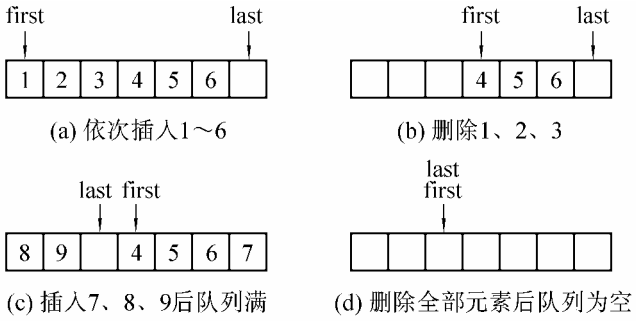


图 2-3　环形队列示意图

4. 环形队列的动态增长

可能读者会问，如果队列已满，想要队列自动增长来插入新的元素，用这种环形队列是否可以实现呢？其实可以采用一种简单的技术来实现环形队列的动态增长，其方法如下。

在插入操作中如果发现队列已满，可以再分配一个是原来队列双倍的空间，将原队列中的数据复制到新分配的空间中，再将新分配的空间作为队列，将原来的队列释放掉。复制时有两种情况，如图 2-4 所示。

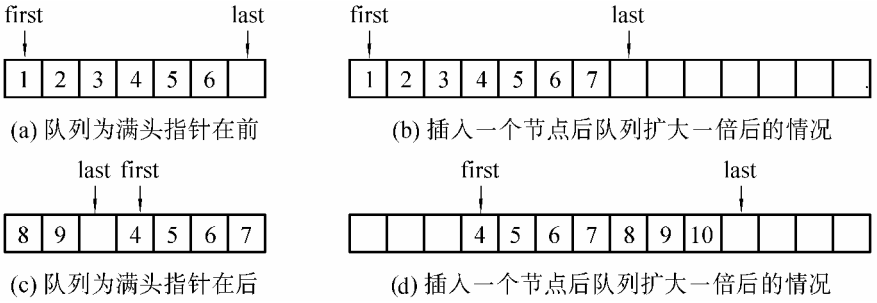


图 2-4　环形队列动态增长示意图

图 2-4(a)所示的是 last 大于 first 的情况，这时只要简单地按顺序将队列中的元素复制到新分配的空间中即可。如图 2-4(b)所示，先将 1～6 复制过去，再插入 7。

图 2-4(c)所示的是 last 小于 first 的情况，这种情况要分两次复制，先将 first 和队列空间最尾部间的元素复制到新空间中，再将队列空间头部和 last 之间的元素复制到刚才复制的元素的后面即可。如图 2-4(d)所示，先将 4～7 复制过去，再将 8，9 复制到 7 的后面，然后再在 9 的后面插入 10。

采用上述方法实现动态环形队列比较简单，且整个队列空间在物理上是连续的，使用了确定的内存空间。但其缺点是需要复制数据，当队列中元素很多时，如果遇上队列满，这时插入数据很费时间。有没有其他办法实现既可以将队列动态扩大，又不需要复制数据呢？

做一个简单的推导。由于队列是 FIFO 的，因此可以将队列看成是一个连续的序列，存储队列元素的空间在物理上或逻辑上也必须是连续的。首先，要想不复制数据，原来的队列空间必须继续使用，否则必然要将原来存在队列中的数据移动到新分配的空间。由于队列空间的前后内存可能已经被使用了，显然无法做到在物理上将队列空间扩大成为连续的空间，因此必须另外分配一块空间，把新分配的空间和原来的队列空间在逻辑上连起来，成为一个在逻辑上是连续的空间。

5. 环形队列的 C 语言结构体描述

```
typedef struct QUEUE_st {
    void **ppData；          /* 存放数据指针的指针数组 */
    UINT uMaxCount；         /* 队列中的最大数据数量 */
    UINT uHead；             /* 队列头部位置 */
    UINT uTail；             /* 队列尾部位置 */
} QUEUE；
```

6. 环形队列的插入尾部和弹出头部编码实现

```
/** 插入数据到队列的尾部的函数，如果队列已满会自动将队列空间扩大一倍再插入
    @param QUEUE *pQueue——队列指针
    @param void *pData——要插入的数据指针
    @return INT—— 返回 CAPI_FAILED 表示队列已满，申请不到内存将队列再扩大，插入失
                  败；返回 CAPI_SUCCESS 表示插入成功
*/
INT Queue_InsertTail( QUEUE *pQueue，void *pData )
{
    UINT uTailNext；
    /* 求出尾部位置的下一个位置 */
    if ( pQueue->uTail == pQueue->uMaxCount－1 ) {
        /* 当到了数组的最尾部时，下一个要从数组头部重新计算 */
```

```
        uTailNext = 0；
    }
    else {
        uTailNext = pQueue->uTail + 1；
    }
    if ( uTailNext != pQueue->uHead ) {    /* 队列未满的情况 */
        pQueue->ppData[pQueue->uTail] = pData；
        pQueue->uTail = uTailNext；
    }
    else {    /* 队列为满的情况 */
        /* 将队列空间扩大一倍 */
        void **ppData = (void **)malloc(pQueue->uMaxCount *2 *sizeof(void *))；
        if ( ppData == NULL ) {
            return CAPI_FAILED；
        }
        if ( pQueue->uHead > pQueue->uTail ) {
            UINT i；
            /* 复制从 uHead 到队列最尾部间的数据 */
            for ( i = pQueue->uHead； i < pQueue->uMaxCount； i++) {
                ppData[i] = pQueue->ppData[i]；
            }
            /* 复制从 0 到 uTail 间的数据 */
            for ( i = 0； i < pQueue->uTail； i++) {
                ppData[i + pQueue->uMaxCount] = pQueue->ppData[i]；
            }
            pQueue->uTail += pQueue->uMaxCount；
        }
        else {
            UINT i；
            /* 复制从 uHead 到 uTail 间的数据 */
            for ( i = pQueue->uHead； i < pQueue->uTail； i++) {
                ppData[i] = pQueue->ppData[i]；
            }
        }
        /* 将数据插入新分配的队列空间中 */
        ppData[pQueue->uTail] = pData；
        pQueue->uTail += 1；
        pQueue->uMaxCount *= 2；
        free(pQueue->ppData)；
```

```
        pQueue->ppData = ppData；
    }
    return CAPI_SUCCESS；
}

/** 队列的弹出头部数据函数
    @param QUEUE *pQueue——队列指针
    @return void *——返回 NULL 表示队列为空，否则返回弹出的头部数据指针
*/
void *Queue_PopHead(QUEUE *pQueue)
{
    UINT uHead；
    uHead = pQueue->uHead；
    if ( uHead != pQueue->uTail ) {
        /* 头部和尾部没有重合表示队列不为空的情况 */
        if ( uHead == pQueue->uMaxCount－1 ) {
            pQueue->uHead = 0；
        }
        else {
            pQueue->uHead += 1；
        }
        return pQueue->ppData[uHead]；
    }
    else {
        return NULL；
    }
}
```

上面只列出了主要的插入尾部和弹出头部的函数，实际上环形队列也可以插入在头部，弹出尾部数据，这样环形队列就可以变成一个双向队列了。这部分内容留给读者作为练习。另外，队列的创建和释放操作代码没有在上面列出来，本书附带的光盘中有完整的代码可供参考。

2.2.3 STL 中的动态队列(STL::deque)

了解 STL 的人都知道 STL 中有一个双向动态队列，HP 是采用多个数组来实现一个动态队列，多个数组用一个 map 来管理。map 其实也是一个数组，通过 map 将多个数组连成一个逻辑上连续的空间，这样便实现了一个在队列满时不需要复制元素的动态队列。下面简单介绍 HP 的 STL 中的动态队列实现，如图 2-5 所示。详情参考 STL

相关的书籍。

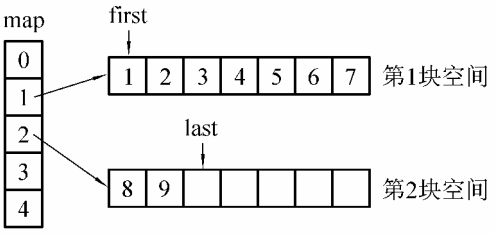


图 2-5　STL 中的动态队列示意图

假设 map 的大小为 5，每块的大小为 7，初始时，map 的第 1 个元素指向第 1 块空间，插入 7 个元素到队列后，由于队列满，需重新分配一块空间，即图中的第 2 块空间，将 map 的第 2 个元素指向这块空间，此时要插入的元素 8，9 都是在第 2 块空间中依次插入。如果要从队列中弹出数据，则从 first 指向的元素弹出，比如要弹出 1～7 这几个元素，当弹出 7 后，第一块空间已经没有了元素，此时需要释放掉第 1 块空间。

HP 的 STL 中的这种实现，随着插入和弹出操作的不断进行会不断地分配新的块和释放空的块。比如块的大小为 10，如果插入了 1 000 个元素又全部弹出，会重新分配 99 个块及释放 99 个块，如果 map 设得很小，还要进行复杂的 map 管理操作。

2.2.4　动态环形队列

1. 动态环形队列概念的产生

通常队列的插入和弹出操作都是交替进行的，插入 1 000 个元素和弹出 1 000 个元素交替进行时，可能队列中的元素最多只有几十个。如有 50 个元素，采用 STL 的动态队列要进行 19 次块的分配和释放操作，但如果采用一个有 50 个元素的环形队列，则不需要重新分配和拷贝数据就可以满足要求。能不能实现一个既可以动态增长又不需要拷贝数据，同时还尽量减少块的分配和释放的操作呢？

要减少块的分配和释放操作，必须提高每个块的利用率，HP 的 STL 中的动态队列，每个块中的元素被弹出后，空间并没有被重复使用，这是导致当数据不断地插入和弹出时，块也不断地分配和释放的原因。在环形队列中，弹出的空间可以重复使用，如果将每个块都设计成环形队列，那么每个块的利用率就大大提高了，就可以减少块的分配和释放操作。

2. 动态环形队列图解

多个环形队列组成的一个动态环形队列如图 2-6 所示。

如图 2-6 所示，初始时将 map 大小设为 6，环形队列大小设为 7，有两个指针 first

和 last 分别指向 map 中的第一个环形队列和最后一个环形队列，先插入 1，2，3，4，5，再删除 1，2，再插入 7，8，9，此时第一个环形队列就满了，因此再插入 10 时就必须新建一个环形队列了，一直插入到 20。由图上看出，共有三个环形队列，每个队列最大可以存储 7 个元素。还可以看出与 STL 的动态队列相比，每个环形队列需要一个 head、一个 tail 和一个空的空间，STL 的动态队列的每个块不需要任何辅助空间。是不是这种设计比 STL 的动态队列多浪费很多辅助空间呢？其实不然，如果环形队列的大小设成比正常情况下队列的元素稍多的话，整个队列中大部分情况下只有一个环形队列，消耗的空间并不比 STL 的动态队列多，当队列元素达到高峰时，需要很多个环形队列，这时消耗的空间会比 STL 的动态队列大，但如果将环形队列大小设成超过一定值时，多出的辅助空间占整个队列空间的比值很小，比如将环形队列大小设成 1 024，则比值为 0.5%左右，几乎可以忽略不计，所以在空间利用率上与 STL 的动态队列差别不是很大，但是动态环形队列的优势在于不会像 STL 的动态队列那样频繁地分配和释放内存。

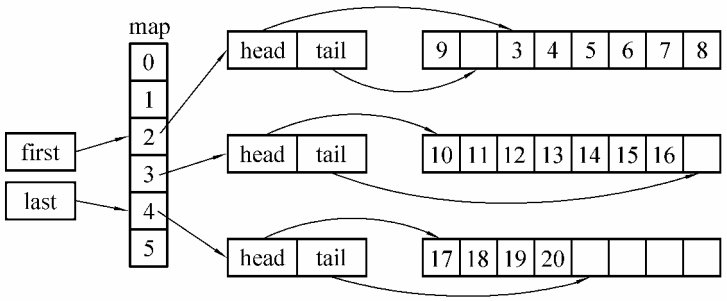


图 2-6　动态环形队列示意图

在多任务环境下，特别是服务器软件，由于是长时间运行，通常对内存的使用有很高的要求，频繁分配、释放内存容易产生内存碎片，必然会影响软件长期运行的稳定性，因此，在这种情况下使用像动态环形队列这种内存性能比较好的数据结构就非常合适。

3. 动态环形队列的 map 管理

动态环形队列中的 map 管理也和 STL 的动态队列一样，当 map 大小不够时，分配一个大一倍的 map，然后将原 map 中的数据复制到新的 map 中，释放原 map，使用新的 map 作为队列的 map 就可以了。复制时，也是将数据复制到新的 map 中，以便在整个队列的前后都可以插入数据。详细的实现可以参阅以下代码。

4. 动态环形队列的 C 语言结构体定义

```
/* DeQue 中的环形队列结构体定义 */
typedef struct DEQUEBLOCK_st {
    UINT uHead；      /* 环形队列的头部 */
    UINT uTail；      /* 环形队列的尾部 */
    UINT uMapPos；    /* 所对应 map 中的位置 */
    void **ppData；   /* 数据指针数组 */
} DEQUEBLOCK；

/*   DeQue 的结构体定义 */
typedef struct DEQUE_st {
    DEQUEBLOCK **ppMap；   /* 管理环形队列的 map 数组指针 */
    DEQUEBLOCK *pFirst；   /* 第一个环形队列指针 */
    DEQUEBLOCK *pLast；    /* 最后一个环形队列指针 */
    UINT uMapSize；        /* map 大小 */
    UINT uBlockSize；      /* 环形队列大小 */
} DEQUE；
```

5. 动态环形队列的插入和弹出头部操作代码

```
/** DeQue 中的环形队列创建函数，其中使用两次 malloc()函数可以优化成调用
    一次 malloc()
    @param UINT uBlockSize——环形队列大小
    @return DEQUEBLOCK *——环形队列指针
*/
DEQUEBLOCK *DeQueBlock_Create(UINT uBlockSize)
{
    DEQUEBLOCK *pBlock；
    pBlock = (DEQUEBLOCK *)malloc(sizeof(DEQUEBLOCK))；
    if ( pBlock != NULL )
    {
        pBlock->ppData = malloc(uBlockSize *sizeof(void *))；
        if ( pBlock->ppData == NULL )
        {
            free(pBlock)；
            return NULL；
        }
        pBlock->uHead = 0；
        pBlock->uTail = 0；
```

```
    }
    return pBlock；
}

/** DeQue 的插入尾部函数
    @param DEQUE *pQue——DeQue 指针
    @param void *pData——要插入的数据指针
    @return INT (by default) —— 返回 CAPI_SUCCESS 表示成功；返回 CAPI_FAILED
                                表示失败
*/
INT DeQue_InsertTail( DEQUE *pQue，void *pData )
{
    DEQUEBLOCK *pBlock；
    UINT uTailNext；
    pBlock= pQue->pLast；
    if ( pBlock->uTail == pQue->uBlockSize－1 ) {
        uTailNext = 0；
    }
    else {
        uTailNext = pBlock->uTail + 1；
    }
    if ( pBlock->uHead != uTailNext ) { /* 队列未满的情况 */
        pBlock->ppData[pBlock->uTail] = pData；
        pBlock->uTail = uTailNext；
    }
    else {
        DEQUEBLOCK *pNewBlock = DeQueBlock_Create(pQue->uBlockSize)；
        if ( pNewBlock == NULL ) {
            return CAPI_FAILED；
        }
        if ( pBlock->uMapPos >= pQue->uMapSize－1 ) {
            /* 重新分配 map */
            DEQUEBLOCK **ppMap =malloc(pQue->uMapSize *2 *
                                    sizeof(DEQUEBLOCK *))；
            if ( ppMap == NULL ) {
                return CAPI_FAILED；
            }
            memcpy(ppMap，pQue->ppMap，
                pQue->uMapSize *sizeof(DEQUEBLOCK *) )；
```

```
                pQue->uMapSize *= 2；
                free(pQue->ppMap)；
                pQue->ppMap = ppMap；
            }
            pNewBlock->uMapPos = pBlock->uMapPos + 1；
            pNewBlock->ppData[0] = pData；
            pNewBlock->uTail += 1；
            pQue->ppMap[pNewBlock->uMapPos] = pNewBlock；
            pQue->pLast = pNewBlock；
        }
        return CAPI_SUCCESS；
    }

    /** DeQue 的弹出头部节点数据函数
        @param DEQUE *pQue——DeQue 指针
        @return void *——弹出的头部数据指针
    */
    void *DeQue_PopHead(DEQUE *pQue)
    {
        DEQUEBLOCK *pBlock；
        UINT uHead；
        pBlock = pQue->pFirst；
        uHead = pBlock->uHead；
        if ( uHead != pBlock->uTail ) {
            /* 头部和尾部没有重合表示队列不为空的情况 */
            if ( uHead == pQue->uBlockSize－1 ) {
                pBlock->uHead = 0；
            }
            else {
                pBlock->uHead += 1；
            }
            if ( pBlock->uHead == pBlock->uTail ) {
                /* 队列中只有一个元素的情况，如果存在下一块则需要指向下一个块 */
                if ( pQue->pLast != pQue->pFirst ) {
                    UINT uMapPos = pBlock->uMapPos；
                    DeQueBlock_Destroy(pQue，pBlock，NULL)；
                    pBlock = pQue->ppMap[uMapPos + 1]；
                    pQue->pFirst = pBlock；
                }
```

```
            }
            return pBlock->ppData[uHead]；
        }
        else {
            return NULL；
        }
    }
```

2.2.5 各种队列的时间效率测试及分析

下面对普通环形队列(Queue)，STL 的动态队列(STL∷deque)和动态环形队列(DeQue)之间的效率差别进行比较。

1. 测试环境

CPU： Celeron 2.4G
Memory： 256M
OS： Windows

2. 测试结果

测试结果如表 2-1 所示。

表 2-1 各种队列的效率测试结果

	初始条件	执行的操作	花费时间(ms)
Queue	队列大小设为 128	插入 2 000 000 个整数再全部弹出	93
Queue	队列大小设为 2 000 000	插入 2 000 000 个整数再全部弹出	47
STL∷deque	STL 中的缺省值(4 096 字节)	重复 20 000 次执行插入 100 个整数再弹出 100 个的过程	110
STL∷deque	STL 中的缺省值(4 096 字节)	插入 2 000 000 个整数再全部弹出	125
DeQue	环形队列大小 128，map 大小 16	插入 2 000 000 个整数再全部弹出	78
DeQue	环形队列大小 1 024，map 大小 16	插入 2 000 000 个整数再全部弹出	78
DeQue	环形队列大小 128，map 大小 16	重复 20 000 次执行插入 100 个整数再弹出 100 个的过程	62

需要注意的是，以上数据是在 Windows 操作系统上测出，由于是分时系统，测试时会受操作系统其他进程的影响，另外由于 Malloc 函数在系统内存碎片不同情况下所消耗的时间是不一样的，每次测出的时间可能都会有差异，因此时间值并不很精确，是测试多次后，取其出现频率最高的一个时间值。

3. 测试结果分析

从上面测试结果可以看出，DeQue 在队列中的元素总数低于其中的环形队列大小时，所花时间和 Queue 没有发生扩大及拷贝时的时间是一样的，DeQue 在不断地分配和释放环形队列时，所花的时间为 78ms，是 DeQue 在没有发生分配和释放环形队列时的 1.26 倍。STL∷deque 效率是最低的了，主要原因是 STL 中使用了模板，如果改写成不使用模板，则应该和 DeQue 在不断分配和释放环形队列时所花的时间差不多。因此可以看出，使用多个环形队列来组成一个动态队列效果是最好的。虽然 Queue 在预先设定成很大值时效率很高，但是队列使用的空间一直是按最大的需求来设定的，实际上大部分情况下队列中的元素总数远小于最大值，会浪费很多空间，另外当空间不够时需要分配内存进行拷贝操作，在队列满时的插入耗时很大。

2.2.6 各种队列的适用范围

Queue、STL∷deque、DeQue 三个队列可以说各有特点，从时间效率上看，虽然有些差别，但它们的效率都是非常高的，实际上它们有不同的适用范围。

Queue 在到达最高峰后，空间不会减少，所以它占用的内存一直是按最大值来分配的，它的优点是作为固定长度的队列使用时，时间效率非常高，一旦队列大小超过预先分配的长度，就会出现重新分配内存并复制的情况，因此 Queue 的平均性能不是很好，它适用于队列最大空间不是很大的情况，具体大小取决于系统对内存使用的限制。

STL∷deque 从算法上看非常符合队列的操作特性，当队列使用的空间用完后，只是简单地重新分配一块内存，但不会发生内存复制现象，因此它的平均性能比 Queue 好多了，空间使用上也比 Queue 好多了。它的空间使用是根据队列的大小动态进行调整的，不会出现到了高峰之后内存却一直被占用不释放的问题。其缺点是队列交替进行插入和弹出操作时，还是一直在分配和释放内存，和 Queue 比起来还需要一个额外的 map，因此这种队列适用于队列元素较多的情况。

可以说 DeQue 是和 STL∷deque 的 HP 实现非常接近的一个队列，它的特点中很多地方和 STL∷deque 差不多，区别是 DeQue 多需要一些额外的空间，在队列交替进行插入和弹出操作时，DeQue 不会出现分配和释放内存的情况，有很好的内存性能，因此 DeQue 适用于内存性能要求较高的场合。从空间的使用和时间效率方面综合考虑，DeQue 是较优的队列，特别是在多任务环境下，许多服务器软件都是长时间运行

的，对内存使用有很高的要求，此时用 DeQue 就有明显的优势。

2.2.7 关于时间效率和空间效率的原则

本节中有两个地方涉及时间效率和空间效率的均衡，一个是环形队列的设计上，有一个元素空间未用；另一个是动态环形队列 DeQue 的设计上，相对 STL::deque 多用了很多辅助的空间。那么到底什么情况下应该让空间效率优先呢？什么情况下应该让时间效率优先呢？在设计过程中总是有许多的软件人员为此烦恼，本书给出以下两条空间效率优先和时间效率优先原则作为参考。

设某模块消耗的全部空间大小为 N，运行固定数量元素的全部操作时间为 T，假设增加 n 个空间能够减少的运行时间为 t，或者说减少 n 个空间会增加的运行时间为 t，则有：

空间效率优先原则　如果 $n / N > K \cdot t / T$，则应该考虑采用空间效率优先，即此时应该降低时间效率来提高空间效率。

时间效率优先原则　如果 $n / N < K \cdot t / T$，则应该考虑采用时间效率优先，即此时应该降低空间效率来增加时间效率。

其中，K 为常数，K 的取值取决于对时间效率和空间效率的需求， 一般情况下 K 可以取值为 1，在内存受限的系统中，对空间效率比较敏感，K 可以取大一些，而对空间没有太大限制的系统中则 K 的取值可小一些。

本节的 DeQue 设计中，当环形队列大小为 1 024 时，$n / N = 0.5\%$，而当队列元素总数小于 DeQue 中的环形队列大小时效率则提高了 26%左右，显然符合时间效率优先原则。

本节主要介绍了用数组来实现队列的设计和实现，这些都是在单任务下的实现，队列的多任务实现会放到第 8 章中进行介绍。另外队列也可以用链表来实现，用链表实现队列将在第 3 章中进行介绍，并会和本节实现的队列进行性能、效率等方面的对比分析。

2.3 排序表

2.3.1 排序算法介绍

在各种数据结构与算法中介绍的排序算法很多，如冒泡排序、插入排序、选择排序、堆排序、归并排序和基数排序等。这些排序有些适用于数组，有些适用于链表，如快速排序用于数组就很快，而插入排序用于数组时由于会有大量数据移动导致很慢。一般的商业应用中，并不会用到所有的排序算法，而是选用几种最有效的算法。本书

主要介绍几种最常用也是最有效的排序算法，本节介绍在数组中比较实用的快速排序算法，在第 3 章中会介绍插入排序、归并排序和基数排序。排序因数据多少、算法的不同，效率会相差很大。插入排序一般用于数据较少的情况；快速排序既可用于数据较少，也可用于数据较多的情况；归并排序则用于数据比较多的情况；而基数排序则用于数据非常多的情况(30 万条以上)。

2.3.2 快速排序算法

上一节讲述了如何用数组来实现栈操作，本节主要讲述用数组来实现二分查找和快速排序的功能。排序的算法在各种数据结构与算法的书中介绍了很多，遗憾的是很多人谈到排序时首先就说到冒泡排序(Bubble sort)，实际上像冒泡排序这类效率非常低的算法在商业软件中很少使用。

快速排序是冒泡排序的一种改进，快速排序算法的基本思想是先从未排序的数据中任意取一个数据，以这个数据为界，将比这个数据小的数据全部移到这个数的左边，比这个数大的数全部移到它的右边，然后对两边的数据重复刚才的过程，最后便得到一个从小到大的有序序列。

在实际中，选取的分界数据的左边数据个数和比分界数据小的数据个数并不是一样多的，在数组中插入数据会引起大量后续数据移动，因此不能简单地把数据移动到分界数据的左边或右边，而要在移动过程中不断调整分界数据的位置，直到分界数据的左边数据全部小于它，右边数据全部大于它为止。

在选定分界数据后，通常是从数组的两端交替地向中间扫描。假设有两个位置变量 uLow 和 uHigh，初始时，uLow 的位置为最左端，也就是 0，uHigh 为数组最右端的位置。先从 uHigh 开始向左依次和分界数据进行比较，直到找到一个比分界数据小的数据，此时 uHigh 的值便是这个比分界数据小的数据的位置，将此数据和 uLow 位置的数据进行交换，然后从 uLow 的下一位置开始向右依次和分界数据进行比较，直到找到比分界数据大的数据，将此数据和 uHigh 位置的数据进行交换，如此下去直到 uHigh 和 uLow 相等为止。

例 2-1　假设每次都用数组的第一个元素作为分界数据，且数组的初始排列如下：

128(uLow)　137　55　42　532　99　12　3　1 000　255(uHigh)

试演示快速排序的过程。

解　第一趟，取 128(左边第一个元素)作为分界数据。

① 将 255 和 128 进行比较，比 128 大；再向左取 1 000 和 128 进行比较，还是大于 128；再取 3 和 128 进行比较，发现比 128 小，将 3 和 128 进行交换得到以下序列：

3(uLow)　137　55　42　532　99　12　128(uHigh)　1 000　255

② 从 3 的下一位置开始取 137 和 128 进行比较，137 比 128 大，因此将 137 和 128 进行交换得到以下序列：

3　128(uLow)　55　42　532　99　12　137(uHigh)　1 000　255

③ 取 12 和 128 进行比较，比 128 小，因此交换 12 和 128，得到以下序列：

3　12(uLow)　55　42　532　99　128(uHigh)　137　1 000　255

④ 取 55 和 128 进行比较，比 128 小；再取 42 和 128 比较，还是比 128 小；再取 532 和 128 进行比较，比 128 大，因此交换 532 和 128，得到以下序列：

3　12　55　42　128(uLow)　99　532(uHigh)　137　1 000　255

⑤ 再取 99 和 128 比较，比 128 小，交换 99 和 128，得到以下序列：

3　12　55　42　99(uLow)　128(uHigh)　532　137　1 000　255

⑥ 最后取 99 和 128 进行比较，比 128 小，再取 128，此时 uLow 和 uHigh 位置相等，第一趟结束。此时以 128 为界，左边数据都比它小，右边数据都比它大。

只要再分别对 128 的左边数据和右边数据重复以上过程，最后就可以得到一个有序的序列，这就是快速排序算法的过程。

分析上面的过程可以发现，第①步时 3 和 128 交换掉了，第②步时 128 又和 137 交换掉了，实际上第③步时我们可以将 128 换成 3，但 3 保持不变，即第④步的序列如下：

3(uLow)　137　55　42　532　99　12　3(uHigh)　1 000　255

同理第②步时 137 保持不变，即第③步的序列如下：

3　137(uLow)　55　42　532　99　12　137(uHigh)　1 000　255

如此下去，可以得到和前面算法一样的效果，但交换减少了一半，效率得到很大的提高。

快速排序算法的效率与选择的分界数据关系非常大，如果每次选取的分界数据都是最小的数据，那么其效率跟冒泡排序一样糟糕。因此，在实际操作中不会像上面一样每次选取最左端的数据作为分界数据，如果每次取的数据刚好是中间数据，也就是说每次取的分界数据刚好比它小的数据和比它大的数据一样多，那么这种情况下快速排序的性能是最好的。但是要将中间数据选择出来可不是一件容易的事情，如果写一个算法来取出中间数据的话，由于这个算法要消耗大量时间，实际上整体效率被降低了。

如果随机取一个数来作分界数据的话，由于概率上的平均性能，排序的性能也是很好的，但取随机数的算法可能要消耗大量时间，因此实际上一般不会采用这种方法。一般可以取左端、右端、中间三个数的中间数据作为分界数据。比如在例 2-1 中，第一趟取 128、255、532 三个数的中间值作为分界数据，中间值为 255，因此第一趟取的分界数据为最右端的 255。

2.3.3 排序表的设计

下面用数组设计一个排序表来实现数组的快速排序功能，排序表用结构体描述如下。

```
typedef struct SORTTABLE_st {
    void **ppData；        /* 存放数据指针的指针数组 */
    UINT uCursorCount；/* 当前排序表中的数据数量 */
    UINT uMaxCount；   /* 排序表中的最大数据数量 */
} SORTTABLE；
```

1. 排序表的创建函数

设计排序表的创建和释放函数。排序表的创建函数如下。

```
/** 排序表的创建函数
    @param UINT uMaxCount——排序表的最大尺寸
    @return SORTTABLE *——成功返回排序表指针；失败返回 NULL
*/
SORTTABLE *SortTable_Create(UINT uMaxCount)
{
    SORTTABLE *pTable；
    if ( uMaxCount == 0 )
    {
        return NULL；
    }
    pTable = (SORTTABLE *)malloc(sizeof(struct SORTTABLE_st))；
    if ( pTable != NULL )
    {
        pTable->ppData = (void **)malloc(uMaxCount *sizeof(void *))；
        if ( pTable->ppData != NULL )
        {
            pTable->ppData[0] = NULL；
            pTable->uMaxCount = uMaxCount；
```

```
            pTable->uCursorCount = 0；
        }
        else
        {
            free( pTable )；
            pTable = NULL；
        }
    }
    return pTable；
}
```

2. **排序表的释放函数**

排序表的释放函数如下。

```
/** 排序表的释放函数
    @param SORTTABLE *pTable——排序表指针
    @param DESTROYFUNC DestroyFunc——数据释放函数
    @return void——无
*/
void SortTable_Destroy( SORTTABLE *pTable，DESTROYFUNC DestroyFunc)
{
    if ( pTable != NULL )
    {
        if ( DestroyFunc != NULL )
        {
            UINT i；
            for ( i = 0； i < pTable->uCursorCount； i++ )
            {
                (*DestroyFunc)( pTable->ppData[i] )；
            }
        }
        free( pTable->ppData )；
        free( pTable )；
    }
}
```

3. **排序表的快速排序函数**

下面设计排序表的快速排序函数。先将排序表的分界操作写成如下劈开函数，这里需要一提的是下面函数中是取数组的第一个数据作为分界数据的，主要是为了方便

读者看代码，省去了取中间数据的代码，读者有兴趣可以自己补充一下取中间数据的代码。

```
/** 排序表的快速排序劈开函数，将给定范围数据按指定数据劈成两部分，一部分全部小
    于指定数据，另一部分全部大于指定数据，指定数据取自给定范围排序表中的第一个
    数据
    @param SORTTABLE *pTable——排序表指针
    @param UINT uStart——表中开始位置
    @param UINT uEnd——表中结束位置
    @param COMPAREFUNC CompareFunc——数据比较函数
    @return UINT——指定数据在表中的位置，即它的数组下标
*/
static UINT SortTable_Split( SORTTABLE *pTable，UINT uStart，UINT uEnd，
                             COMPAREFUNC CompareFunc)
{
    void *pSelData；
    UINT uLow；
    UINT uHigh；
    uLow = uStart；
    uHigh = uEnd；
    pSelData = pTable->ppData[uLow]；
    while ( uLow < uHigh )
    {
        while ( (*CompareFunc)(pTable->ppData[uHigh]，pSelData) > 0
            && uLow != uHigh )
        {
            ——uHigh；
        }
        if ( uHigh != uLow )
        {
            pTable->ppData[uLow] = pTable->ppData[uHigh]；
            ++uLow；
        }
        while ( (*CompareFunc)( pTable->ppData[uLow]，pSelData ) < 0
            && uLow != uHigh )
        {
            ++uLow；
        }
        if ( uLow != uHigh )
```

```
            {
                pTable->ppData[uHigh] = pTable->ppData[uLow]；
                ——uHigh；
            }
        }
        pTable->ppData[uLow] = pSelData；
        return uLow；
    }
```

再写排序表的快速排序函数，先用递归来实现排序表的快速排序，算法如下。

```
/** 排序表的快速排序函数，将指定区间的数据用快速排序算法进行排序
    @param SORTTABLE *pTable——排序表指针
    @param UINT uStart——要排序的区间起点
    @param UINT uEnd——要排序的区间终点
    @param COMPAREFUNC CompareFunc——数据比较函数
    @return void——无
*/
void SortTable_QuickSort( SORTTABLE *pTable，UINT uStart，UINT uEnd，
                          COMPAREFUNC CompareFunc)
{
    UINT uMid = SortTable_Split(pTable，uStart，uEnd，CompareFunc )；
    if ( uMid > uStart )
    {
        (void)SortTable_QuickSort(pTable，uStart，uMid－1，CompareFunc)；
    }
    if ( uEnd > uMid )
    {
        (void)SortTable_QuickSort(pTable，uMid + 1，uEnd，CompareFunc)；
    }
}
```

可以看出上面的递归算法是非常简洁的。在实际商业应用中，通常对效率有很高的要求，经验告诉我们，使用递归算法通常效率较低，因此有必要把递归算法改写成非递归算法，下面就将递归算法改写成非递归算法。

2.3.4 非递归的快速排序算法

本节用非递归算法来实现快速排序算法，通常非递归算法用栈来实现，本节先介绍使用栈的非递归算法，然后介绍一个不使用栈的非递归算法。

1. 使用栈的非递归算法

使用栈的非递归算法编码如下。

```
/** 快速排序算法的使用栈的非递归算法函数
    @param SORTTABLE *pTable——排序表指针
    @param UINT uStart——表中要排序数据区间的起点，为数组下标
    @param UINT uEnd——表中要排序数据区间的终点，为数组下标
    @param COMPAREFUNC CompareFunc——数据比较函数
    @return void——无
void   SortTable_QuickSort2( SORTTABLE *pTable，UINT uStart，UINT uEnd，
                                COMPAREFUNC CompareFunc)
{
    STACK *pStack；
    UINT uLow；
    UINT uHigh；
    UINT uMid；
    uLow = uStart；
    uHigh = uEnd；
    pStack = Stack_Create(uHigh - uEnd)；
    (void)Stack_Push(pStack，(void *)uLow)；
    (void)Stack_Push(pStack，(void *)uHigh)；
    while ( !Stack_IsEmpty(pStack) )
    {
        uHigh = (UINT)Stack_Pop(pStack)；
        uLow = (UINT)Stack_Pop(pStack)；
        if ( uLow < uHigh )
        {
            uMid = SortTable_Split(pTable，uLow，uHigh，CompareFunc )；
            if ( uMid > uLow )
            {
                (void)Stack_Push(pStack，(void *)uLow )；
                (void)Stack_Push(pStack，(void *)(uMid－1 ))；
            }
            if ( uHigh > uMid )
            {
```

```
                (void)Stack_Push(pStack，(void *)(uMid + 1) )；
                (void)Stack_Push(pStack，(void *)uHigh)；
            }
        }
        Stack_Destroy(pStack，NULL)；
    }
}
```

在使用栈的非递归算法中，要频繁调用栈的操作函数，函数调用的开销是很大的，如果把栈操作改成不用栈函数调用，则效率会有所提高。

2. 不使用栈的非递归算法

不使用栈函数调用的非递归算法编码如下。

```
/** 排序表的快速排序函数，不使用栈调用的非递归算法，将指定区间的数据用快速排序算
    法排好
    @param SORTTABLE *pTable——排序表指针
    @param UINT uStart——要排序的区间起点
    @param UINT uEnd——要排序的区间终点
    @param COMPAREFUNC CompareFunc——数据比较函数
    @return void——无
*/
void SortTable_QuickSort(SORTTABLE *pTable，UINT uStart，UINT uEnd，
                         COMPAREFUNC CompareFunc)
{
    UINT *puStack；
    UINT uStackTop；
    UINT uLow；
    UINT uHigh；
    UINT uMid；
    if ( uEnd－uStart < 1 )
    {
        return；
    }
    uLow = uStart；
    uHigh = uEnd；
    puStack = (UINT *)malloc((uHigh－uLow + 1) *sizeof(UINT))；
    if ( puStack == NULL )
    {
```

```
        return CAPI_FAILED；
    }
    uStackTop = 0；
    puStack[uStackTop] = uLow；
    ++uStackTop；
    puStack[uStackTop] = uHigh；
    ++uStackTop；
    while ( uStackTop != 0 )
    {
        －－uStackTop；
        uHigh = puStack[uStackTop]；
        －－uStackTop；
        uLow = puStack[uStackTop]；
        if ( uLow < uHigh )
        {
            uMid = SortTable_Split(pTable，uLow，uHigh，CompareFunc )；
            if ( uMid－uLow > uHigh－uMid )
            {
                puStack[uStackTop] = uLow；
                ++uStackTop；
                puStack[uStackTop] = uMid－1；
                ++uStackTop；
                if ( uHigh > uMid )
                {
                    puStack[uStackTop] = uMid + 1；
                    ++uStackTop；
                    puStack[uStackTop] = uHigh；
                    ++uStackTop；
                }
            }
            else
            {
                puStack[uStackTop] = uMid + 1；
                ++uStackTop；
                puStack[uStackTop] = uHigh；
                ++uStackTop；
                if ( uMid > uLow )
                {
                    puStack[uStackTop] = uLow；
```

```
                    ++uStackTop；
                    puStack[uStackTop] = uMid－1；
                    ++uStackTop；
                }
            }
        }
    }
    free( puStack )；
}
```

至此，我们介绍了快速排序算法的三个不同版本：递归实现、使用栈的非递归实现、不使用栈的非递归实现。在这里需要讨论一下递归算法和非递归算法的效率问题。

3. 两种算法的效率比较

递归算法和非递归算法效率差距到底有多大？相信许多人想知道这个答案。如果测试一下上面的递归实现和使用栈的非递归实现，发现效率差距不是很大，但很多有经验的软件工程师发现，有时候递归算法和非递归算法差距非常大，为什么会出现这种现象呢？

要解决上述问题，可以分析一下递归算法使用的栈和非递归算法使用的栈的区别。递归算法使用的栈是由程序自动生成的，程序中定义的局部变量都存放在栈中，还有函数调用时要将参数压入栈，因此如果程序较复杂，里面定义了很多局部变量或调用了很多其他函数，会导致栈很大，在递归调用时，需要先进行栈操作，栈很大后，每次递归调用需要操作很大的栈，效率自然就很低；而非递归算法中，由于递归调用没有使用程序的栈，每轮循环使用自己预先创建的栈，因此不管程序复杂度如何都不会影响到程序的效率，这就是为什么有些程序改写成非递归算法后效率提高很多，而有些程序改写后效率提高却不明显的原因。从快速排序算法中可以看出，在使用递归的算法中，栈中只有函数调用的参数和一个局部变量 uMid，栈中没有其他内容，程序栈是很小的，和自己创建的栈的开销差不多，所以最后的效率也差距不大。在不使用栈的非递归算法中，主要提高的效率就是减少了函数调用，节省了函数调用的栈，所以效率比递归算法和使用栈的非递归算法提高了。

2.3.5 快速排序算法的复杂度分析

假设快速排序的元素个数为 n，所需时间为 T(n)，每趟分界时，左边元素个数用 k(n) 表示，0≤k(n)≤n，每趟分界所需时间和 n 成正比，因此每趟分界操作时间可以用 c·n 来表示，其中 c 为一常数，从上面算法知道有等式

$$T(n) = c \cdot n + T(n-k(n)-1) + T(k(n))$$

当 k(n)取 0 时，便退化成冒泡排序，此时有

$$T(n) = c \cdot n + T(n-1) + T(0)$$

$$T(n) \approx c \cdot n \cdot (n+1) / 2 = O(n^2)$$

当 k(n) 取值为 n/2 时，有

$$T(n) \approx c \cdot n + 2 \cdot T(n / 2) \approx O(n\log n)$$

虽然快速排序算法的最坏时间复杂度为 $O(n^2)$，但快速排序算法的平均时间复杂度仍然为 O(nlogn)，读者可以自己推导一下平均时间复杂度。如果将快速排序算法的时间复杂度记为 C·nlogn，在各种同数量级的时间复杂度排序算法中，快速排序算法的常数 C 是最小的。

再来讨论快速排序算法的空间复杂度，从前面的非递归算法可以看出，需要 O(n)的辅助栈空间，而前面的递归算法好像不需要辅助空间，实际上递归算法仍然是需要辅助栈空间的，只不过递归算法的栈是程序自动生成的。读者也许会问为什么需要 O(n)的辅助栈空间，而不是 O(logn)的辅助栈空间，这是由算法的最坏时间复杂度决定的，因为当达到最坏时间复杂度时，栈的深度为 O(n)。

2.3.6 二分查找算法

二分查找算法是一种最常用的查找算法，用在排好序的表中进行查找。查找的方法是先找中间的数据，如果要查找的数据小于中间数据，则在中间数据左边一半中继续用二分查找法进行查找；如果要查找的数据大于中间数据，则在右边一半中继续查找；如果等于中间数据，则表明找到了要查找的数据。

例 2-2 在数列 1 5 9 13 27 55 89 103 205 中用二分查找法查找 89 这个整数。

解 步骤 1：先计算中间数据位置，总共有 9 个数据，中间数据位置为(1+9)/2 = 5，所以，把第 5 个数据 27 和 89 进行比较，发现大于中间数据，在右边一半中继续查找。

步骤 2：计算右边一半的中间位置为(1 + 4)/2 = 2，所以取 89 和 89 进行比较，发现相等，找到了要找的数据，查找结束。

```
/** 排序表的二分查找函数，调用此函数前必须对表从小到大排好序
    @param SORTTABLE *pTable——排序表指针
    @param void *pData——要查找的匹配数据
    @param COMPAREFUNC CompareFunc——比较函数
    @return void *——成功返回查到的数据；失败返回 NULL
*/
void *SortTable_FindData(SORTTABLE *pTable，void *pData，
                         COMPAREFUNC CompareFunc)
{
```

```
    UINT uLow；
    UINT uHigh；
    UINT uMid；
    if ( pTable == NULL || CompareFunc == NULL || pData == NULL
        || pTable->uCursorCount == 0 )
    {
        return NULL；
    }
    uLow = 0；
    uHigh = pTable->uCursorCount－1；
    uMid = 0；
    while ( (INT)uLow <= (INT)uHigh )
    {
        INT nResult；
        uMid = ( uLow + uHigh ) / 2；
        nResult = (*CompareFunc)( pTable->ppData[uMid]，pData )；
        if ( nResult > 0 )
        {
            uHigh = uMid－1；
        }
        else if ( nResult < 0 )
        {
            uLow = uMid + 1；
        }
        else
        {
            /* 已经发现了匹配数据，返回 */
            return pTable->ppData[uMid]；
        }
    }
    /* 未找到匹配数据，返回空指针 */
    return NULL；
}
```

2.4 实例：HOOK 管理功能的实现

2.4.1 单个函数的 HOOK 实现

很多读者可能用过金山词霸之类的软件，鼠标只要往单词上一指，就可以弹出对应单词的解释，这种屏幕取词便是使用了 HOOK 功能拦截 Windows API 的调用来实现的。本节要用数组来实现对函数 HOOK 的管理功能，首先讨论如何实现对单个函数的 HOOK 功能。

对某个函数的 HOOK 就是通过某种方法使得调用这个函数时调用到指定的钩子函数。实现 HOOK 的方法很多，最简单的一种方法便是将调用函数的开头位置的代码改成一条跳转到钩子函数的指令即可。

譬如我们自己写了一个 MyMalloc()函数，以实现所有对 malloc()函数的调用都调用 MyMalloc()函数，假设 malloc 函数的起始地址是 0x00421540，那么只要将 0x00421540 开始的几个字节改成 jmp DWORD PTR MyMalloc 指令即可， 假设 MyMalloc()函数执行代码的起始地址为 00401830，在 INTEL32 位芯片系列的机器中，jmp DWORD PTR MyMalloc 这条指令的机器码为 0xff2500401830，共 6 个字节，只要将这 6 个字节写入 malloc()函数的开头 6 个字节中，即可实现调用 malloc()函数时跳转到 MyMalloc()函数。

以下代码便能实现将跳转到 MyMalloc()函数的指令写入 malloc 函数起始位置。

```
DWORD lpSrcFunc = (DWORD)malloc；
DWORD dwNewFunc = (DWROD)MyMalloc；
DWORD lppNewFunc = &dwNewFunc；
*(unsigned char *)lpSrcFunc = (unsigned char)0xff；
*(((unsigned char *)lpSrcFunc)+1) = (unsigned char)0x25；
memcpy( (void *)(lpSrcFunc+2)，(const void *)&lppNewFunc，sizeof(DWORD) )；
```

如果是像 Windows NT 系列的操作系统，对执行代码的内存做了保护，是不能直接调用以上代码的，还需要使用 VirtualProtect()函数将 lpSrcFunc 指向的地址处的内存属性改成可以写的，才可以调用以上代码。

还要考虑将 malloc()的调用改掉后再如何恢复的问题，否则程序运行很容易出现问题。恢复的做法便是将 malloc()起始处的 6 个字节先保存起来，当要恢复时，将保存的 6 个字节写回原来位置即可。

还需说明的是，以上介绍的是同一进程内的 HOOK 实现方法，如果要 HOOK 其他进程中的函数调用，还需要将上述代码注入到目标进程中去执行才可以实现 HOOK 功能，将代码注入到其他进程中的方法有很多种，Windows 系统下常用的方法有以下

几种。

① 使用 Windows API 函数 SetWindowsHookEx()将程序注入到其他进程中执行；

② 使用修改 DLL IMPORT 表的方法来实现对 DLL 中函数的 HOOK；

③ 使用 CreateRemoteThread()函数将代码注入到其他进程中执行；

④ 对于 Windows NT 系列操作系统，还可以使用注册表设置来实现将代码注入到其他进程中执行。

以上几种方法在很多书籍中都有详细介绍，这里就不作进一步介绍。

2.4.2 多个函数的 HOOK 实现

采用前面的方法对多个函数实现 HOOK，当设置 HOOK 时，如何保存那些函数的起始 6 个字节？当取消 HOOK 时，如何从保存的数据中找到某个函数的起始 6 个字节？解决这个问题，可以用一个结构体数组来实现，结构体中包含源函数地址、源函数起始位置的 6 个字节数据等内容。HOOK 管理结构体设计如下。

```
typedef struct APIHOOKDATA_st {
    DWORD dwSrcFuncAddr；    /* 源函数的地址 */
    DWORD dwNewFuncAddr；    /* 钩子函数的地址 */
    BYTE byHeaderCode[6]；   /* 源函数起始 6 字节 */
    WORD wFlag；             /* 用来表示是否设置了 HOOK 的标志 */
} APIHOOKDATA；

typedef struct APIHOOK_st {
    APIHOOKDATA *pHookData；
    UINT uMaxFunctions；
} APIHOOK；
```

APIHOOKDATA 结构体用来记录单个函数的 HOOK 数据，APIHOOK 结构体用来管理多个函数的 HOOK，里面包含一个 APIHOOKDATA 结构体数组，uMaxFunctions 是数组的长度，表示最多可以 HOOK 的函数个数。

1. 多个函数的 HOOK 管理

对多个函数的 HOOK 管理可以分为以下四个操作步骤。

① 初始化；

② 设置某个函数的 HOOK；

③ 取消某个函数的 HOOK；

④ 关闭。

下面以 Windows NT 系列操作系统为例来实现以上四个操作的代码。初始化操作

主要是为结构体分配内存，将整个数组清零。编码如下。

```
/** ApiHook 模块初始化函数
    @param INT nMaxHookFuncCounts——最大可设置的钩子数量
    @return INT (by default)——返回 NULL 表示失败
*/
HANDLE ApiHook_Init(UINT uMaxFunctions)
{
    APIHOOK * pApiHook = (APIHOOK *)malloc(sizeof(APIHOOK))；
    if ( pApiHook )
    {
        pApiHook->uMaxFunctions = uMaxFunctions；
        pApiHook->pHookData =
                (APIHOOKDATA *)malloc(sizeof(APIHOOKDATA) *uMaxFunctions)；
        if ( NULL != pApiHook->pHookData)
        {
            memset(pApiHook->pHookData，0，sizeof(APIHOOKDATA) *uMaxFunctions )；
            return (HANDLE)pApiHook；
        }
        free(pApiHook)；
    }
    return NULL；
}
```

2. 设置 HOOK 操作

设置某个函数的 HOOK 操作主要是在数组中查找未使用的节点，将源函数相关信息保存到节点中，修改源函数起始位置的几个字节为一条跳转到钩子函数的指令。

```
/** 通过地址来设置某个函数的钩子函数
    @param HANDLE hApiHook——由 ApiHook_Init()函数生成的句柄
    @param DWORD dwSrcFuncAddr——源函数地址
    @param DWORD dwNewFuncAddr——钩子函数地址
    @return INT (by default)——返回－1 表示失败；返回 0 表示在 HOOK 数组中的序号
*/
INT ApiHook_SetByAddr(HANDLE hApiHook，DWORD dwSrcFuncAddr，
                      DWORD dwNewFuncAddr)
{
    DWORD dwOldProtect；
    DWORD dwNewProtect；
```

```
    DWORD lpSrcFunc；
    DWORD lppNewFunc；
    UINT i；
    INT nAlreadyFlag = 0；
    if ( NULL == hApiHook )
{
    return－1；
}
APIHOOK    *pApiHook = (APIHOOK *)hApiHook；
lpSrcFunc = dwSrcFuncAddr；
/* 查找是否已设置了钩子 */
for ( i = 0； i < pApiHook->uMaxFunctions； i++ )
{
    if ( pApiHook->pHookData[i].dwSrcFuncAddr == lpSrcFunc )
    {
        /* 如果已经设置了钩子，仅仅改变 */
        nAlreadyFlag = 1；
        break；
    }
}
/* 如果没有设置源函数的钩子函数，在表中找出一个可供记录的位置 */
if ( i == pApiHook->uMaxFunctions )
{
    for ( i = 0； i < pApiHook->uMaxFunctions； i++ )
    {
        if (pApiHook->pHookData[i].wFlag == 0 )
        {
            break；
        }
    }
    if ( i == pApiHook->uMaxFunctions )
    {
        return－1；
    }
}
/* 将新的钩子函数地址记录到表中 */
pApiHook->pHookData[i].dwNewFuncAddr = dwNewFuncAddr；
/* 以下这段代码将源函数头部 6 个字节保存到表中 */
lppNewFunc = (DWORD)(&(pApiHook->pHookData[i].dwNewFuncAddr) )；
```

```
if ( !nAlreadyFlag )
{
    /* 将源函数起始处 6 个字节保存到 byHeaderCode.中 */
    memcpy( pApiHook->pHookData[i].byHeaderCode，(const void *)lpSrcFunc，6)；
}
/* 以下这段代码将源函数首部 6 个字节改成为一条跳转到新函数地址的指令 */
if ( VirtualProtect( (LPVOID)lpSrcFunc，6， PAGE_EXECUTE_READWRITE，
                &dwOldProtect ) == 0 )
{
        return－1；
    }
    *(unsigned char *)lpSrcFunc = (unsigned char)0xff；
    *(((unsigned char *)lpSrcFunc)+1) = (unsigned char)0x25；
    memcpy( (void *)(lpSrcFunc+2)，(const void *)&lppNewFunc，4 )；
    if ( VirtualProtect( (LPVOID)lpSrcFunc，6，dwOldProtect， &dwNewProtect) == 0 )
    {
        return－1；
    }；
    pApiHook->pHookData[i].wFlag = 1；
    pApiHook->pHookData[i].dwSrcFuncAddr = lpSrcFunc；
    return (INT)i；
}
```

3. 取消 HOOK 操作

取消某个函数的 HOOK 操作和设置操作相反，主要是在数组中查到要操作的函数的相关记录信息，将源函数起始位置处的 6 个字节恢复，清除数组中的对应记录信息。编码如下。

```
/** 取消某个函数的钩子函数设置
    @param HANDLE hApiHook——由 ApiHook_Init()函数生成的句柄
    @param DWORD dwSrcFuncAddr——源函数地址
    @return INT (by default)——返回－1 表示失败；返回 0 表示在 HOOK 数组中的序号
*/
INT ApiHook_UnSetByAddr(HANDLE hApiHook，DWORD dwSrcFuncAddr)
{
    UINT i；
    DWORD dwOldProtect；
    DWORD dwNewProtect；
    DWORD lpSrcFunc；
```

```
APIHOOK  *pApiHook = (APIHOOK *)hApiHook；
if ( NULL == hApiHook )
{
    return－1；
}
for ( i = 0； i < pApiHook->uMaxFunctions； i++ )
{
    if ( pApiHook->pHookData[i].dwSrcFuncAddr == dwSrcFuncAddr )
    {
        break；
    }
}
if ( i == pApiHook->uMaxFunctions )
{
    return－1；
}
lpSrcFunc = pApiHook->pHookData[i].dwSrcFuncAddr；
/* 将 lpSrcFunc 指向的内存属性修改为可以读写的 */
if ( VirtualProtect( (LPVOID)lpSrcFunc，6，PAGE_EXECUTE_READWRITE，
                    &dwOldProtect ) == 0 )
{
    return－1；
}
memcpy( (void *)lpSrcFunc，pApiHook->pHookData[i].byHeaderCode，6 )；
/* 恢复 lpSrcFunc 指向的内存属性 */
if ( VirtualProtect( (LPVOID)lpSrcFunc，6，dwOldProtect，&dwNewProtect ) == 0 )
{
    return－1；
}
pApiHook->pHookData[i].wFlag = 0；
if ( pApiHook->pHookData[i].pszDllName )
{
    free(pApiHook->pHookData[i].pszDllName)；
    pApiHook->pHookData[i].pszDllName = NULL；
}
if ( pApiHook->pHookData[i].pszFuncName )
{
    free( pApiHook->pHookData[i].pszFuncName )；
```

```
            pApiHook->pHookData[i].pszFuncName = NULL；
        }
        return (INT)i；
    }
```

2.4.3 HOOK 功能的应用简介

HOOK 功能是一项非常重要的功能，是软件工程师必须掌握的技术。HOOK 功能在实际中应用非常广泛，如词典软件、安全软件及各种系统软件中均会用到；还有一些 C 语言语法不能实现的功能也可以用 HOOK 来实现，如 C++标准模板库中就用到了内存配置器的概念，即用户可以自己指定内存分配方法，可以使用自己的内存分配函数来进行内存管理。在 C 语言中，这个功能从语法上要求用函数指针作为回调函数等来实现，但是有很多已经写好的模块是不能随便修改接口的，如果要让这些模块在不修改的基础上使用自定义内存管理算法，就必须使用 HOOK 功能。可以对 malloc()，free()等内存管理函数设置 HOOK，使它们跳转到自定义的内存管理函数即可。

HOOK 功能还有一项非常重要的用途就是在白盒测试方面，像单元测试需要打桩，传统的打桩方法是将源文件中需要打桩的函数用桩函数替代，这个工作虽然可以通过工具来实现，但还是比较麻烦；如果使用 HOOK 来打桩则可以避免修改源文件，只要将桩函数设置成钩子函数，比传统的方法简洁多了。

2.4.4 HOOK 使用的注意事项

使用 HOOK 时要特别小心，使用不当很容易导致程序崩溃。比如要 HOOK malloc()和 free()函数，如果 HOOK 的时机不当，会导致 HOOK 了的 free()函数去释放未被 HOOK 的 malloc()的内存，引起程序崩溃。

本 章 小 结

本章主要用数组来实现栈的设计、队列的设计和排序表的设计，并在排序表中介绍了快速排序算法和二分查找算法的实现。对于查找，有些人觉得二分查找法在数据较大时可能不够快，确实有比二分查找更快的算法，本书第 4 章中讲的哈希表的查找就是比二分查找更快的算法，但二分查找法在数组上实现非常简单，在实际中还是经常会用到。当然有些初学者可能会纳闷，数学中不是讲过黄金分割法比二分法更快吗？要知道，计算机里面处理整数运算比浮点数运算可是要快好多倍，采用黄金分割法在数学理论上比二分法快，但实际中需要用到浮点数运算，自然比不上二分法的速度了。

习题与思考

1. 编程调用栈操作函数压入 100 万个随机整数，再全部弹出来，打印出花费的总时间。
2. 编程将 100 万个随机整数插入到 SortTable 中，再调用快速排序函数排序，打印出排序花费的时间。
3. 编程将 1～100 万的整数一次插入排序表里，调用二分查找函数将 1～100 万这 100 万个数依次查找一遍，打印出花费的时间。
4. 编码实现动态环形队列 DeQue 的弹出尾部节点函数 DeQue_PopTail()和插入头部函数 DeQue_InsertHead()函数。

3 链　　表

本章主要介绍链表，包括单向链表和双向链表以及使用整块内存的链表；介绍了实际应用中适合于链表的排序算法如插入排序、归并排序、基数排序算法；还介绍了多任务下的遍历设计；最后介绍了使用链表管理短信息系统的实例。

链表是数据结构的基础，相信大部分读者已经学习过链表，为什么本书还要论述链表这么简单的问题呢？看完本章后，你会发现本书所介绍的内容与其他同类书籍的区别。本章将所有的链表问题归结为单向链表和双向链表，并将队列、栈等操作都归并到链表中，还实现了链表的各种实用的排序算法。

3.1 单向链表

3.1.1 单向链表的存储表示

数组在内存的存储单元中用的是连续的内存地址，而链表的特点是使用的存储单元可以是连续的也可以是不连续的。要使用不连续的存储单元，在每个存储单元中必须保存后继存储单元的地址，才能实现按顺序地访问存储单元。一般把存有数据或数据地址以及相关的一个或多个同类型存储单元的地址的连续存储单元叫**节点**(Node)。单向链表的节点包括数据或数据指针以及后继节点的地址。

例如，有 4 个字符串“ yellow ”、“ red ”、“ blue ”、“ green ”，如果把它们保存在单向链表中，可以用图 3-1 来表示它们在内存中的储存。

链表节点 1 中存有字符串“ yellow ”和一个指向节点 2 的地址指针，节点 2 中存有字符串“ red ”和一个指向节点 3 的地址指针，节点 3 中存有字符串“ blue ”和一个

指向节点 4 的地址指针，节点 4 中存有字符串“ green ”和一个空指针(NULL)。在这里节点 4 是最后一个节点，因为它没有后继节点，所以将它的后继节点指针赋为 NULL。

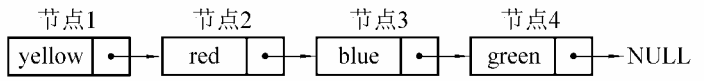


图 3-1　数据用链表储存在内存中的示意图

链表节点在 C 语言中一般用如下结构体来描述。

```
typedef struct SINGLENODE_st {
    void *pData；                          /*  数据指针  */
    struct SINGLENODE_st *pNext；          /*  下一个节点指针  */
} SINGLENODE，*PSINGLENODE；
```

注意：这里使用了 void *pData 来表示数据指针，主要是充分利用 C 语言特性，可以指向任意类型的数据，保证节点上可以存储任意的数据类型。

从图 3-1 可以知道，如果要访问链表，必须知道第一个节点的地址，因此需要一个指针来指向第一个节点；另外在对链表进行增加节点操作时，经常要将节点添加到链表尾部，如果只有第一个节点的指针，在添加到链表尾部时必须要按顺序访问到最后一个节点才能添加进去，当链表节点非常多时效率很低，因此需要用一个指针指向最后一个节点；还需要用一个指针来指向当前节点，这个当前节点的意义在后面讲逐个节点遍历时会进行详细讲解；有时候还要知道链表里面的节点个数，因此需要保存链表里节点的个数；这样就可以设计出链表的数据结构了，用 C 语言结构体描述如下。

```
typedef unsigned int UINT；
typedef struct SINGLELIST_st {
    SINGLENODE *pHead；        /*  第 1 个节点的指针  */
    SINGLENODE *pTail；        /*  最后 1 个节点的指针  */
    SINGLENODE *pCur；         /*  当前节点的指针  */
    UINT uCount；              /*  保存链表节点的个数  */
} SINGLELIST，*PSINGLELIST；
```

改进图 3-1，可画出一个单向链表的完整示意图，如图 3-2 所示。

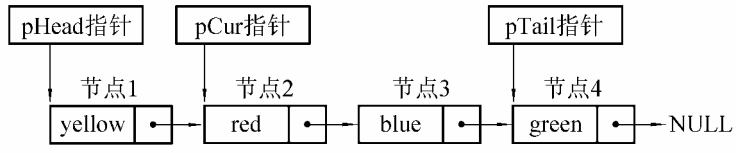


图 3-2　数据用链表储存在内存中的示意图

将链表的存储表示弄清楚后，就要开始设计链表的操作接口了。

3.1.2 单向链表的接口设计

在设计单向链表的操作接口前，我们看一个短消息实例中的情况：有短消息到来时，要保存起来，也就是在链表中增加一个节点，因此链表要提供一个增加节点的操作，并且这个节点增加在链表的尾部；将短消息转发出去后，需要将短消息删除掉，也就是在链表中删除一个节点，因此链表要提供一个删除节点的操作；另外，当其他任务对链表进行转发操作时，是对链表节点一个一个地进行枚举操作，这样链表还需要提供对节点的遍历操作。

按照 C 语言的特点，需要创建操作即初始化分配内存的操作，销毁操作即释放整个链表所有节点及节点数据的内存的操作，因此链表还需具备创建和销毁操作。

这样就得到了链表如下的几个最基本操作。

① 创建链表；
② 销毁链表；
③ 增加节点；
④ 删除节点。

有了这些操作，基本上可以对付短消息方面的存储问题了，但是如果要做一个商业级的有通用性的链表，这些操作是远远不能满足需求的。一般来说我们经常碰到的栈操作、队列操作方面的功能都要放到这里来实现，还有获取某个位置上的节点数据、链表节点数量等操作，还需要排序、查找等操作。

对于链表的栈操作，可以通过将节点添加在链表尾部来实现压栈操作，弹出链表的头节点来实现出栈操作；队列和栈差不多，可以通过从链表尾部添加节点，从头部弹出节点来实现。注意，这里讲栈和队列的操作都是从头部弹出节点，可不可以从尾部弹出节点呢？答案是否定的。对于单向链表，如果从尾部弹出节点，需要先知道尾部节点的上一节点，还要将这个节点的下一节点指针赋为 NULL。但单向链表没有前向指针，所以要得到这个节点只能从链表头部去搜索，这样效率太低了。从前面所讲又可以设计出链表如下的一些栈和队列的操作。

① 从头部添加节点；
② 从尾部添加节点；
③ 弹出头部节点；
④ 查找操作；
⑤ 获取指定位置节点；
⑥ 弹出指定位置节点；
⑦ 获取链表节点数量。

对这些操作接口的设计，由于数据是 void *型，无法进行数据的删除、比较操作，因此，这两个操作要用回调函数来实现，整个接口设计用 C 语言描述如下。

```
typedef INT (*CompareFunc) (void *pData1，void *pData2)；
typedef INT (*DestroyFunc) (void *pData)；
```

这两个回调函数在绪论的任意数据处理里已经介绍。

3.1.3 单向链表的基本功能编码实现

前面已经给出单向链表的结构体设计，下面给出单向链表的基本函数的编码实现。

```
/** 单向链表的创建函数，创建完后链表为空
    @param void——无
    @return SINGLELIST *——失败返回 NULL；成功返回一个单向链表结构体指针
*/
SINGLELIST *SingleList_Create( void )
{
    SINGLELIST    *pSingleList；
    /* 分配内存操作 */
    pSingleList = (SINGLELIST *)malloc(sizeof(SINGLELIST))；
    if ( pSingleList != NULL )
    {
        /* 初始化链表结构体各指针成员为空，链表节点个数为 0 */
        pSingleList->pCur = NULL；
        pSingleList->pHead = NULL；
        pSingleList->pTail = NULL；
        pSingleList->uCount = 0；
    }
    return pSingleList；
}

/** 单向链表的释放函数
    @param SINGLELIST *pSingleList——要释放的单向链表的指针
    @param DESTROYFUNC pDestroyFunc——链表节点数据释放回调函数
    @return void——无
*/
void SingleList_Destroy( SINGLELIST *pSingleList, DESTROYFUNC DestroyFunc )
{
    SINGLENODE *pNode；
    if ( pSingleList )
```

```
    {
        /* 从头节点开始，一个接一个释放链表节点数据及节点 */
        pNode = pSingleList->pHead；
        while ( pNode != NULL )
        {
            SINGLENODE *pDelNode；
            pDelNode = pNode；
            pNode = pNode->pNext；
            if ( DestroyFunc != NULL && pDelNode->pData != NULL )
            {
                /* 释放数据 */
                (*DestroyFunc)( pDelNode->pData )；
            }
            free( pDelNode )； /* 释放节点 */
        }
        /* 释放链表结构体 */
        free( pSingleList )；
    }
}

/** 单向链表的添加节点函数，添加的节点放在单向链表的头部
    @param SINGLELIST *pSingleList——要添加的单向链表指针
    @param void *pData——要添加的节点的数据指针
    @return INT——失败返回 CAPI_FAILED；成功返回 CAPI_SUCCESS
*/
INT SingleList_InsertHead( SINGLELIST *pSingleList，void *pData )
{
    SINGLENODE *pNode；
    /* 参数校验 */
    if ( pSingleList == NULL || pData == NULL )
    {
        return CAPI_FAILED；
    }
    /* 新建一个节点 */
    pNode = (SINGLENODE *)malloc( sizeof(SINGLENODE) )；
    if ( pNode == NULL )
    {
        return CAPI_FAILED；
    }
```

```
    pNode->pData = pData；    /* 将节点数据指针指向传进来的数据 */
    /* 将新建节点的下一节点指针指向头节点，并将头节点改为新建的节点 */
    pNode->pNext = pSingleList->pHead；
    pSingleList->pHead = pNode；
    /* 判断是否尾节点指针为空，如果为空表示原来链表中没有节点，此时应该将
    *  尾节点指向新加入的节点
    */
    if ( pSingleList->pTail == NULL )
    {
        pSingleList->pTail = pNode；
    }
    /* 将链表节点数据加 1 */
    pSingleList->uCount++；
    return CAPI_SUCCESS；
}

/** 单向链表的添加节点函数，添加的节点放在单向链表的尾部
    @param SINGLELIST *pSingleList——要添加的单向链表指针
    @param void *pData——要添加的节点的数据指针
    @return INT——失败返回 CAPI_FAILED；成功返回 CAPI_SUCCESS
*/
INT SingleList_InsertTail( SINGLELIST *pSingleList，void *pData )
{
    SINGLENODE    *pNode；
    /* 参数校验 */
    if ( pSingleList == NULL || pData == NULL )
    {
        return CAPI_FAILED；
    }
    /* 新建一个节点 */
    pNode = (SINGLENODE *)malloc( sizeof(SINGLENODE) )；
    if ( pNode == NULL )
    {
        return CAPI_FAILED；
    }
    pNode->pData = pData；    /* 将节点数据指针指向传进来的数据 */
    pNode->pNext = NULL；    /* 将节点的下一节点赋为空指针 NULL */
```

```
    /* 判断尾节点指针是否为空，如果为空表示原来链表中没有节点，此时应该将
    *  尾节点指向新加入的节点，并且头节点指针也应该指向新节点
    */
    if ( pSingleList->pTail == NULL )
    {
        pSingleList->pTail = pNode；
        pSingleList->pHead = pNode；
    }
    else
    {
        /* 如果尾节点指针不为空，此时应该将尾节点下一节点指针指向新加入的节
        *  点，并且尾节点指针也应该指向新节点
        */
        pSingleList->pTail->pNext = pNode；
        pSingleList->pTail = pNode；
    }
    /* 将链表节点数据加 1 */
    pSingleList->uCount++；
    return CAPI_SUCCESS；
}

/** 单向链表的弹出头节点函数
    @param SINGLELIST *pSingleList——要操作的单向链表指针
    @return void *——失败返回 NULL，成功返回要弹出的头节点的数据指针
*/
void *SingleList_PopHead( SINGLELIST *pSingleList )
{
    SINGLENODE *pPopNode；    /* 用来指向要弹出数据的节点的指针 */
    void *pPopData；    /* 用来指向要弹出的数据的指针 */
    /* 参数校验 */
    if ( pSingleList == NULL || pSingleList->pHead == NULL )
    {
        return NULL；
    }
    /* 将要弹出数据的节点指针指向链表头节点，弹出数据指针指向头节点的数据 */
    pPopNode = pSingleList->pHead；
    pPopData = pPopNode->pData；
```

```
    /* 判断当前节点指针是否指向头节点，若指向头节点则需将其指向头节点的
    *  下一节点
    */
    if ( pSingleList->pCur == pSingleList->pHead )
    {
        pSingleList->pCur = pSingleList->pHead->pNext；
    }
    /* 将头节点指针指向头节点的下一节点 */
    pSingleList->pHead = pSingleList->pHead->pNext；
    /* 将链表节点数量减 1 */
    pSingleList->uCount－－；
    /* 如果链表的节点数量已经为 0 则表示原来只有一个节点，弹出头节点后，此时链
    *  表已经为空，应该将尾节点指针赋空。当前节点指针前面已经处理过了，如果只
    *  有一个节点则肯定为空，所以这里不需要处理当前节点指针
    */
    if ( pSingleList->uCount == 0 ) {
        pSingleList->pTail = NULL；
    }
    /* 释放弹出的节点，注意这里并没有释放节点数据指针 */
    free( pPopNode )；
    return pPopData；       /* 返回头节点的数据指针 */
}

/** 单向链表的弹出尾节点函数，此函数由于要查找尾节点的前一节点，因此效率很低
    @param SINGLELIST *pSingleList——要操作的单向链表指针
    @return void *——失败返回 NULL；成功返回要弹出的尾节点的数据指针
*/
void *SingleList_PopTail( SINGLELIST *pSingleList )
{
    SINGLENODE *pPopNode；          /* 用来指向要弹出数据的节点的指针 */
    SINGLENODE *pTailPrevNode；     /* 用来指向尾节点的上一节点的指针 */
    void *pPopData；                /* 用来指向要弹出的数据的指针 */
    /* 参数校验 */
    if ( pSingleList == NULL || pSingleList->pHead == NULL )
    {
        return NULL；
    }
    /* 将要弹出数据的节点指针指向链表头节点，弹出数据指针指向头节点的数据 */
```

```
pPopNode = pSingleList->pTail；
pPopData = pPopNode->pData；
pTailPrevNode = pSingleList->pHead；
/* 如果链表的头节点和尾节点相同则表示原来只有一个节点，弹出尾节点后，此时链
* 表已经为空，应该将头节点指针赋空。当前节点指针由于前面已经处理过了，如果
* 只有一个节点，则肯定为空，所以这里不需要处理当前节点指针
*/
if ( pSingleList->pTail == pSingleList->pHead )
{
    pTailPrevNode = NULL；
    pSingleList->pHead = NULL；
}
else
{
    while ( pTailPrevNode != NULL )
    {
        if ( pTailPrevNode->pNext == pSingleList->pTail )
        {
            break；
        }
        pTailPrevNode = pTailPrevNode->pNext；
    }
}
/* 判断当前节点指针是否指向尾节点，如果指向头节点则需将其指向尾节点
*  的前一节点
*/
if ( pSingleList->pCur == pSingleList->pTail )
{
    pSingleList->pCur = pTailPrevNode；
}
/* 将尾节点指针指向尾节点的前一节点 */
pSingleList->pTail = pTailPrevNode；
if ( pTailPrevNode != NULL )
{
    pTailPrevNode->pNext = NULL；
}
/* 将链表节点数量减 1 */
pSingleList->uCount－－；
/* 释放弹出的节点，注意这里并没有释放节点数据指针 */
```

```
    free( pPopNode )；
    return pPopData；       /* 返回头节点的数据指针 */
}

/** 链表的删除节点函数，它将删除和 pMatchData 参数有相同数据的节点，如果
    有许多有相同数据的节点它将只删除第一个有相同数据的节点
    @param SINGLELIST *pSingleList——要操作的单向链表指针
    @param void *pMatchData——要删除节点的匹配数据
    @param COMPAREFUNC CompareFunc——数据比较函数用来比较 pMatchData 参数
                                    和链表节点参数是否相等
    @param DESTROYFUNC DestroyFunc——链表节点的数据释放函数
    @return INT (by default) ——返回 CAPI_FAILED 表示失败或链表中没有匹配的数据；返
                              回 CAPI_SUCCESS 表示成功删除
*/
INT SingleList_Delete( SINGLELIST *pSingleList，void *pMatchData，
                       COMPAREFUNC CompareFunc，DESTROYFUNC DestroyFunc )
{
    SINGLENODE *pNode；
    SINGLENODE *pPrevNode；
    /* 参数校验 */
    if ( pSingleList == NULL || CompareFunc == NULL )
    {
        return CAPI_FAILED；
    }
    pNode = pSingleList->pHead；
    pPrevNode = pNode；
    while ( pNode != NULL )
    {
        /* 比较节点数据是否匹配 */
        if ( (*CompareFunc)( pNode->pData，pMatchData ) == 0 )
        {
            if ( pPrevNode == pNode )
            {
                /* 头节点匹配上了，需要删除头节点 */
                pSingleList->pHead = pNode->pNext；
                if ( pSingleList->pTail == pNode )
                {
                    /* 如果尾节点和 pNode 相同，表明链表里只有一个节点，此时
```

```
                    *  需要将链表尾节点指针和链表当前节点指针赋空
                    */
                    pSingleList->pTail = NULL；
                    pSingleList->pCur = NULL；
                }
            }
            else
            {
                pPrevNode->pNext = pNode->pNext；
                if ( pSingleList->pTail == pNode )
                {
                    /* 如果尾节点和 pNode 相同，表明删除的是尾节点，此时需要
                    *  将尾节点指针指向要删除节点的前一个节点
                    */
                    pSingleList->pTail = pPrevNode；
                }
                if ( pSingleList->pCur == pNode )
                {
                    /* 如果链表当前节点和 pNode 相同，表明删除的是当前节点，此时
                    *  需要将当前节点指针指向要删除节点的下一个节点
                    */
                    pSingleList->pCur = pNode->pNext；
                }
            }
            /* 释放节点数据和节点占用的内存 */
            if ( DestroyFunc != NULL && pNode->pData != NULL )
            {
                (*DestroyFunc)( pNode->pData )；
            }
            free( pNode )；
            break；
        }
        pPrevNode = pNode；
        pNode = pNode->pNext；
    }
    return CAPI_SUCCESS；
}
```

```
/** 单向链表获取指定位置数据的函数
    @param SINGLELIST *pSingleList——要操作的单向链表指针
    @param UINT uIndex——要获取的索引位置，索引从 0 开始计算
    @return void *——索引位置节点的数据指针
*/
void *SingleList_GetAt( SINGLELIST *pSingleList，UINT uIndex )
{
    UINT i；
    SINGLENODE *pNode；
    if ( pSingleList == NULL || pSingleList->uCount >= uIndex )
    {
        return NULL；
    }
    pNode = pSingleList->pHead；
    for ( i = 0； i < uIndex； i++ )
    {
        pNode = pNode->pNext；
    }
    return pNode->pData；
}

/** 单向链表的获取链表节点数量函数
    @param SINGLELIST *pSingleList——要操作的单向链表指针
    @return UINT——链表节点数量，为 0 表示链表为空或者参数非法
*/
UINT    SingleList_GetCount(SINGLELIST *pSingleList)
{
    if ( pSingleList == NULL )
    {
        return 0；
    }
    return pSingleList->uCount；
}

/** 单向链表的获取头节点函数
    @param SINGLELIST *pSingleList——要操作的单向链表指针
    @return void *——头节点的数据指针
*/
void *SingleList_GetHead( SINGLELIST *pSingleList )
```

```
{
    if ( pSingleList == NULL )
    {
        return NULL；
    }
    if ( pSingleList->pHead == NULL )
    {
        return NULL；
    }
    return pSingleList->pHead->pData；
}

/** 单向链表的获取尾节点函数
    @param SINGLELIST *pSingleList——要操作的单向链表指针
    @return void *——尾节点的数据指针
*/
void *SingleList_GetTail( SINGLELIST *pSingleList )
{
    if ( pSingleList == NULL )
    {
        return NULL；
    }
    if ( pSingleList->pTail != NULL )
    {
        return pSingleList->pTail->pData；
    }
    else
    {
        return NULL；
    }
}
```


3.2 单向链表的逐个节点遍历

3.2.1 单向链表逐个节点遍历基本概念

遍历是数据结构与算法中经常要遇到的问题，在设计模式中，有一个迭代器设计模式就是用来介绍对一般的数据结构进行逐个遍历的方法。迭代器设计模式的基本思

想是在外部用一个辅助指针来记住当前要操作的节点，每做一次迭代，辅助指针后移到后继节点上。这种模式有一个非常大的缺点，就是在多任务下使用时会遇到问题。比如有一个任务在对链表删除节点，而另一个任务在对链表进行逐个遍历操作，如果在对链表进行逐个遍历操作时，每遍历一个节点就释放锁的话，下一个要被逐个遍历的节点很可能已被另外一个任务删除了，此时再切换到遍历操作的那个任务时，会导致访问已经删除的节点而出现程序异常。要想安全地进行逐个遍历操作，唯一的办法是在整个遍历过程中都锁住，使删除操作不能进行。如果链表有很多节点的话，进行删除操作的任务可能要等上很长的时间，分时效率非常低。

在 C++标准库 STL 中，也是采用普通的迭代器设计模式来设计逐个节点遍历的，而且将这种迭代器作为所谓“泛型”的基础。前面已经讲过了这种模式是不能在多任务下使用的，除非软件永远在单任务下执行，否则采用这种传统的模式设计的软件就存在很大的风险。可现在几乎所有的操作系统都是多任务的，绝大部分的商业软件也是多任务的，多任务已经成为软件发展的趋势。因此，本书不再使用传统的迭代器模式来设计迭代器，而是介绍一个可以满足多任务要求的新的迭代器模式来实现逐个节点遍历的功能。

先分析一下，为什么传统的迭代器模式在多任务时不好使呢？主要的根源在辅助指针上。当有删除操作发生时，如果删除的刚好是辅助指针指向的节点，则辅助指针指向的节点是已经释放的节点，而辅助指针是在外部的，删除操作的函数看不见这个指针，如果能够让删除操作函数看到这个指针，就可以让删除操作函数在删除过程中判断一下当前删除节点是否为辅助指针指向的节点，如果是，则将辅助指针指向删除节点的后一个节点即可。这样，删除操作完后再进行逐个遍历时，不会出现辅助指针指向释放掉的节点的问题。这样就实现了每遍历完一个节点后就可以将锁释放掉，不用等到将整个链表遍历完才将锁释放，实现了真正多任务意义下的逐个遍历。

下面就来设计链表的多任务遍历器。前面链表的数据结构中曾介绍过一个变量 pCur，就是用作逐个遍历的辅助指针的。读者看看链表删除操作中几个函数的代码，可以发现所有的删除操作函数都要判断当前删除的节点是不是 pCur 指向的节点，如果是，则将 pCur 更新了。

3.2.2 单向链表逐个节点遍历编码实现

下面给出单向链表的逐个遍历函数的编码实现。

```
/** 单向链表的枚举初始化函数
    @param SINGLELIST *pSingleList——要操作的单向链表指针
    @return void——无
*/
```

```
void SingleList_EnumBegin( SINGLELIST *pSingleList )
{
    pSingleList->pCur = pSingleList->pHead；
    return；
}

/** 单向链表枚举下一个节点的函数，第一次调用此函数前必须
    先调用 SingleList_EnumBegin()函数
    @param SINGLELIST *pSingleList——要操作的单向链表的指针
    @return void *——枚举到的节点数据指针
*/
void *SingleList_EnumNext( SINGLELIST *pSingleList )
{
    SINGLENODE *pCur；
    pCur = pSingleList->pCur；
    if ( pCur != NULL )
    {
        pSingleList->pCur = pCur->pNext；
        return pCur->pData；
    }
    return NULL；
}
```

可以看出上面两个关于链表逐个节点遍历的函数并没有使用锁保护，还有前面的链表中其他操作函数也没有使用锁保护，要使这个链表支持多任务，必须再加一层封装。本书第 8 章会详细介绍多任务下的问题，这里只是先打个基础。多任务下的遍历还会遇到其他的问题，这些问题也会在第 8 章中进行详细介绍。

3.3 单向链表的排序

第 2 章介绍过快速排序，快速排序非常适合在数组上实现，但不适合在链表上实现，可以在链表上实现的有价值的排序算法主要有插入排序、归并插入排序和基数排序三种。

3.3.1 插入排序

插入排序是这样一种排序算法：假设前面的 k 个数据已按升度排好序，将第 k+1 个数据和已排好序的 k 个数据从头依次比较，直到找到一个比它大的数据为止，将它

插入在这个数据之前。如此下去，便可以将整个数据序列排好序。

例 3-1　试结合由以下七个数据组成的数列演示插入排序的过程。

7　5　4　1　8　3　9

解　步骤 1：比较 5 和 7，发现 5 比 7 小，将 5 插入到 7 的前面，得到以下系列。

5　7　4　1　8　3　9

步骤 2：比较 4 和 7，4 比 7 小，因此从开头查找比 4 大的节点，第 1 个数 5 就比 4 大，因此将 4 插入到 5 的前面，得到以下序列。

4　5　7　1　8　3　9

步骤 3：比较 1 和 7，1 比 7 小，同第 2 步一样，要将 1 插入到 4 的前面，得到以下序列。

1　4　5　7　8　3　9

步骤 4：将 8 和 7 比较，8 比 7 大，继续比较 3 和 7，3 比 7 小，因此从开头按顺序查找比 3 大的数，找到的为 4，将 3 插入到 4 的前面，得到以下序列。

1　3　4　5　7　8　9

步骤 5：将 9 与 8 比较，比 8 大，9 后面没有其他数了，至此得到一个有序的序列，排序结束。

单向链表的插入排序函数如下。

```
/** 单向链表的插入排序函数，排序按照从小到大进行排列，大小由 CompareFunc 来决定，
    因此用户可以通过 CompareFunc 的返回值设置来决定使用升序或降序排序
    @param SINGLELIST *pSingleList——要操作的单向链表指针
    @param COMPAREFUNC CompareFunc——节点数据比较函数
    @return INT——成功返回 CAPI_SUCCESS；失败返回 CAPI_FAILED
*/
INT SingleList_InsertSort( SINGLELIST *pSingleList， COMPAREFUNC CompareFunc )
{
    SINGLENODE *pNode；          /* 用来遍历 pSingleList 的临时指针 */
    SINGLENODE *pPrevNode；      /* pNode 的前一节点指针 */
    /* 参数校验 */
    if ( pSingleList == NULL || CompareFunc == NULL )
    {
        return CAPI_FAILED；
```

```
}
pNode = pSingleList->pHead；
pPrevNode = NULL；
if ( pNode == NULL )
{
    /* 链表中没有节点，我们把它当作已经排好了序 */
    return CAPI_SUCCESS；
}
while ( pNode->pNext != NULL )
{
    SINGLENODE    *pTempNode；
    pTempNode = pSingleList->pHead；
    pPrevNode = NULL；
    while ( pTempNode != pNode->pNext )
    {
        if ( (*CompareFunc)( pNode->pNext->pData，pTempNode->pData ) < 0 )
        {
            SINGLENODE    *pCurNode = pNode->pNext；
            /* 执行插入操作 */
            if ( pPrevNode != NULL ) /* 插入不在头节点前的情况 */
            {
                /* 将 pCurNode 节点弹出来 */
                pNode->pNext = pNode->pNext->pNext；
                /* 将 pNode->pNext 插入 pTempNode 之前 */
                pPrevNode->pNext = pCurNode；
                pCurNode->pNext = pTempNode；
            }
            else /* 插入在头节点前的情况 */
            {
                /* 将 pCurNode 节点弹出来 */
                pNode->pNext = pNode->pNext->pNext；
                /* 将 pNode->pNext 节点变为头节点 */
                pSingleList->pHead = pCurNode；
                pCurNode->pNext = pTempNode；
            }
            /* 修改尾指针指向 */
            if ( pCurNode == pSingleList->pTail )
            {
                pSingleList->pTail = pNode；
```

```
                    }
                    break；
                }
                pPrevNode = pTempNode；
                pTempNode = pTempNode->pNext；
            } /* while ( pTempNode != pNode->pNext ) */
            if ( pTempNode == pNode->pNext ) /* 没有插入的情况 */
            {
                pNode = pNode->pNext；
            }
            else /* 已经插入的情况 */
            {
                /* 已经插入后不需将 pNode 指针后移，因为前面操作过程中已经移动了 */
            }
        }
        return CAPI_SUCCESS；
    }
```

3.3.2 归并插入排序

归并排序和前面讲过的快速排序和插入排序不同，这里“归并”的意思是将两个或多个排好序的表合并成一个新的排好序的表。在这里以二路归并为例来讲述归并排序的基本过程。首先将要排序的表分成两个节点个数基本相等的子表，然后对每个子表进行排序，最后将排好序的两个子表合并成一个子表。在对子表进行排序时可以将子表再分解成两个节点数量基本相同的子表，当子表足够小时，也可以采用其他排序方法对子表进行排序，然后再对排好序的子表进行归并操作，最后将整个表排好序。

例 3-2　试结合由以下八个数据组成的数列演示归并插入排序的过程。

7　5　1　3　2　4　9　8

解　步骤 1：先将序列分解成两个子表如下。

(7　5　1　3)　(2　4　9　8)

步骤 2：将子表继续分解如下。

(7　5)　(1　3)　(2　4)　(9　8)

步骤 3：将每个子表进行排序。

(5　7)　(1　3)　(2　4)　(8　9)

步骤 4：进行一趟归并，将第 1 组和第 2 组归并，第 3 组和第 4 组归并。

(1　　3　　5　　7)　　(2　　4　　8　　9)

步骤 5：再对上面两组数据进行第 2 趟归并操作，得到排好序的表。

(1　　2　　3　　4　　5　　7　　8　　9)

在上面例子中，子表被分解到只有两个元素时才进行排序，也可以分解到每个子表只有一个元素时再进行归并。在实际过程中，一般是将数据分解到一定大小时就用插入排序或其他排序方法进行排序，这样可以减少递归调用的深度，提高效率。下面是在单向链表中实现的归并插入排序，当子表小于预先给定的值时，就采用插入排序算法进行排序，以降低程序递归调用栈的深度。

单向链表归并插入排序，要设计两个辅助的函数，一个函数将单向链表劈分成两个链表；一个函数将两个排好序的链表合并成一个有序的链表。下面分别介绍这些函数。


```
/** 将一个单向链表劈成两个单向链表，只是从原来链表中间断开，劈完后原链表变成劈开后的第一条链表
    @param SINGLELIST *pSingleList——要被劈开的单向链表
    @param UINT nCount——劈开后的第一个链表的节点个数
    @return SINGLELIST *——成功时返回劈开后的第二条链表；失败返回 NULL
*/
SINGLELIST *SingleList_Split(SINGLELIST *pSingleList，UINT uCount)
{
    SINGLENODE *pNode；          /* 用来保存劈开处的节点的指针 */
    SINGLELIST *pSecondList；    /* 用来记录劈开后的第二条链表的指针 */
    UINT uIndex；                /* 临时循环体变量 */
    if ( pSingleList == NULL )
    {
        return NULL；
    }
    /* 参数校验 */
    if ( uCount == 0 || pSingleList->uCount <= uCount)
    {
        return NULL；
    }
    /* 创建一条空链表 */
    pSecondList = SingleList_Create()；
    if ( pSecondList == NULL )
    {
```

```
        return NULL；
    }
    /* 获取要劈开的位置 */
    pNode = pSingleList->pHead；
    for ( uIndex = 1； uIndex < uCount； uIndex++ )
    {
        pNode = pNode->pNext；
    }
    /* 填充第二条链表的内容 */
    pSecondList->pHead = pNode->pNext；
    pSecondList->pTail = pSingleList->pTail；
    pSecondList->uCount = pSingleList->uCount - uCount；
    /* 修改第一条链表的内容 */
    pSingleList->pTail = pNode；
    pSingleList->uCount = uCount；
    /* 将第一条链表尾节点的 pNext 指针指向 NULL */
    pSingleList->pTail->pNext = NULL；
    return pSecondList；
}

/** 将两个已经排好序的链表进行合并，两个链表都必须是从小到大进行排列，即尾节
    点中数据取值最大，合并后的链表也是从小到大进行排列
    @param SINGLELIST *pSingleListA——要合并的链表
    @param SINGLELIST *pSingleListB——要合并的链表
    @param COMPAREFUNC CompareFunc——节点比较函数
    @return INT——返回 CAPI_FAILED 表示失败，失败时参数没有变化；返回
                  CAPI_SUCCESS 表示成功，成功时 pSingleListA 是合并后的
                  结果，pSingleListB 会被释放掉
*/
INT SingleList_Merge (SINGLELIST *pSingleListA，SINGLELIST *pSingleListB，
                      COMPAREFUNC CompareFunc)
{
    SINGLENODE *pNodeA；    /* 用来指向链表 pSingleListA 的节点的临时指针 */
    SINGLENODE *pNodeB；    /* 用来指向链表 pSingleListB 的节点的临时指针 */
    SINGLENODE *pPrevA；    /*  pNodeA 的前一节点指针 */
    /*  参数校验 */
    if ( pSingleListA == NULL || pSingleListB == NULL || CompareFunc == NULL )
    {
        return CAPI_FAILED；
```

```
}
pNodeA = pSingleListA->pHead；
pNodeB = pSingleListB->pHead；
pPrevA = NULL；
while ( pNodeB != NULL )
{
    while ( pNodeA != NULL )
    {
        if ( (*CompareFunc)( pNodeA->pData，pNodeB->pData ) >= 0 )
        {
            SINGLENODE *pNode；
            /* 将 pNodeB 弹出来保存到 pNode 中 */
            pNode = pNodeB；
            pNodeB = pNodeB->pNext；
            /* 将 pNode 插入到 pNodeA 前面 */
            if ( pPrevA == NULL )
            {
                /* 插入在头指针前的情况，需要修改头指针 */
                pSingleListA->pHead = pNode；
                pNode->pNext = pNodeA；
            }
            else
            {
                pPrevA->pNext = pNode；
                pNode->pNext = pNodeA；
            }
            pPrevA = pNode；
            break；
        }
        pPrevA = pNodeA；
        pNodeA = pNodeA->pNext；
    }
    /* 如果 pSingleListB 的所有数据都大于链表 A 的数据，将 pSingleListB 插入到
    *  pSingleListA 尾部
    */
    if ( pNodeA == NULL )
    {
        pSingleListA->pTail->pNext = pNodeB；
```

```
                pSingleListA->pTail = pSingleListB->pTail；
                break；
            }
        }
        /* 修改 pSingleListA 的节点总数量 */
        pSingleListA->uCount += pSingleListB->uCount；
        free( pSingleListB )；
        return CAPI_SUCCESS；
    }

    /** 对链表使用归并插入排序，归并和插入排序结合使用，先使用归并排序，当链表里面
        元素个数小于一定值时便使用插入排序
        @param SINGLELIST *pSingleList——要排序的链表指针
        @param COMPAREFUNC CompareFunc——链表节点比较函数
        @param UINT uInsertSortCount——使用插入排序时的链表节点个数，当链表节点
                                       个数小于这个值时会使用插入排序算法
        @return INT——返回 CAPI_FAILED 表示失败；返回 CAPI_SUCCESS 表示成功
    */
    INT SingleList_MergeSort( SINGLELIST *pSingleList，COMPAREFUNC CompareFunc，
                              UINT uInsertSortCount)
    {
        SINGLELIST *pSecondList；
        /* 参数校验 */
        if ( pSingleList == NULL || CompareFunc == NULL )
        {
            return CAPI_FAILED；
        }
        if ( pSingleList->uCount < 2 )
        {
            /* 如果节点个数少于两个，认为是已经排好序的 */
            return CAPI_SUCCESS；
        }

        /* 如果链表节点个数小于给定的做插入排序的个数，直接使用插入排序 */
        if ( pSingleList->uCount <= uInsertSortCount )
        {
            (void)SingleList_InsertSort( pSingleList，CompareFunc )；
        }
```

```
    else
    {
        /* 将链表劈成两半 */
        pSecondList = SingleList_Split(pSingleList，(pSingleList->uCount) / 2 )；
        /* 对劈完后的第一个链表进行递归调用归并排序 */
        (void)SingleList_MergeSort( pSingleList，CompareFunc，uInsertSortCount )；
        /* 对劈完后的第二个链表进行递归调用归并排序 */
        (void)SingleList_MergeSort( pSecondList，CompareFunc，uInsertSortCount )；
        /* 将排好序的两个链表合成一个 */
        (void)SingleList_Merge( pSingleList，pSecondList，CompareFunc )；
    }
    return CAPI_SUCCESS；
}
```

3.3.3 基数排序

基数排序是一种分类的方法，先看一个简单的例子。

例 3-3 试对由若干两位整数组成的数列按升序进行排序。

18，19，21，22，33，45，47，58，61，67，69

解 观看以上排好序的两位整数可以发现，这些排好序的数有一个规律，数的十位是按从小到大进行排列的，十位相同的情况下，个位也是按从小到大进行排列的。因此启发我们设计一种按位进行排序的方法，可以先按个位的大小排好，然后再按十位的大小排好，这样便可以得到一个排好序的数列。假设上面整数的初始顺序如下。

45，18，69，19，61，21，22，47，58，33，67

按个位排好为

61，21，22，33，45，47，67，18，58，69，19

再按十位排好为

18，19，21，22，33，45，47，58，61，67，69

注意，这里按十位排序时，必须保证十位上相同的数字，原来排在前面的必须继续排在前面。比如 18，19 这两个数字，十位数相同，原来按个位排时 18 排在 19 前面，按十位排时仍然要将 18 排在 19 前面。

有了这种按位排序的方法，计算机处理就很方便了，对每位排序时，可以采用分类的方法，按位将数据放入对应箱子里，然后再将各个箱子的数据按箱子大小顺序首

尾连接起来形成一个新的序列。比如说 32 位整数写成十进制最多有 10 位，只要进行 10 趟按位分类操作，便可以将整数排好序，这样便得到一个线性复杂度的排序算法。

下面给出一个对整数进行基数排序算法的详细过程，比如给定如下几个整数。

3 721，4 485，3 362，7 941，5 843，6 954，9 271，3 457，2 730，2 008，1 996，18 332

以 10 为基数来进行排序的方法如下。

首先要构造 10 个空箱子，每个箱子有一个头指针和一个尾指针。

步骤 1：进行个位数的排序。先按个位数分类，按顺序依次将个位数为 0 的放入第 0 个箱子，个位数为 1 的放入第 1 个箱子，……个位数为 9 的放入第 9 个箱子。先将 3 721 放入第 1 个箱子，箱子 1 的头指针指向 3 721 这个节点，尾指针也指向 3 721 这个节点。以此类推将 4 485 放入第 5 个箱子，3 362 放入第 2 个箱子。再将 7 941 放入第 1 个箱子。这时要注意，因为箱子 1 里面已有了 3 721 这个节点，因此将 3 721 节点的后继指针指向 7 941 这个节点，同时要将箱子 1 的尾指针改成指向 7 941 节点，照此方法将所有数据依次放入对应箱子中，如图 3-3 所示。

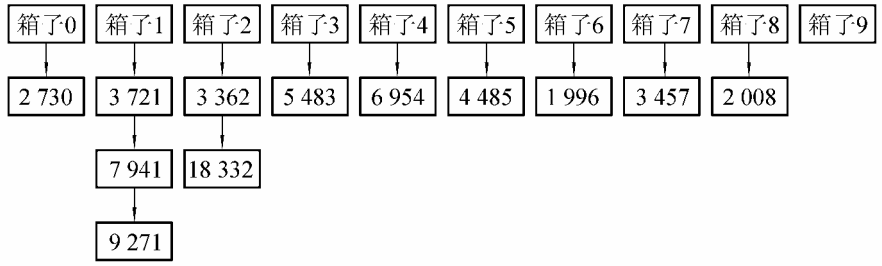


图 3-3　链式基数排序过程示意图 1

步骤 2：当按个位分类完成后，需要将分好类的数据重新排列好，即将箱子 0 的尾节点的后继指针指向箱子 1 的头节点，箱子 1 的尾节点的后继指针指向箱子 2 的头指针，……箱子 8 的尾节点的指针指向箱子 9 的头节点。这样便得到了按以下顺序排列好的整数。

2 730，3 721，7 941，9 271，3 362，18 332，5 843，6 954，4 485，1 996，3 457，2 008

步骤 3：进行十位数的排序。按照第 1 步的办法，对上面的第 2 步中排好的整数再按十位数进行操作，如图 3-4 所示。

再按步骤 2 的办法得到以下排好的整数。

2 008，3 721，2 730，18 332，7 941，5 843，6 954，3 457，3 362，9 271，4 485，1 996

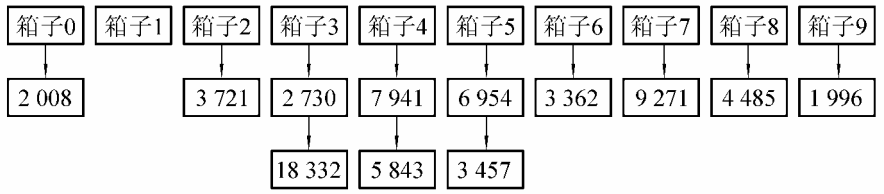


图 3-4　链式基数排序过程示意图 2

步骤 4：进行百位数的排序。对步骤 3 中排好的整数的百位上的数字进行步骤 1、步骤 2 操作，可以得到以下排好的整数。

2 008，9 271，18 332，3 362，3 457，4 485，3 721，2 730，5 843，7 941，6 954，1 996

步骤 5：进行千位数的排序。对步骤 4 中排好的整数的千位上的数字进行步骤 1、步骤 2 操作，可以得到以下排好的整数。

1 996，2 008，2 730，3 362，3 457，3 721，4 485，5 843，6 954，7 941，18 332，9 271

步骤 6：进行万位上的排序。注意这时按步骤 1 方法操作时，除 18 332 有万位上的数字外，其他整数没有万位上的数字，对于没有万位上的数字的整数，可以将它们的万位数字看作 0，因此 18 332 被放入箱子 1 中，其他的被依次放入箱子 0 中，这样便可以得到以下排好序的整数。

1 996，2 008，2 730，3 362，3 457，3 721，4 485，5 843，6 954，7 941，9 271，18 332

至此，完成对上面 12 个整数的排序。

在计算机处理中，对整数排序不会使用 10 作为基数，因为计算机里的整数都是二进制的数，因此可以使用 16 或 256 为基数来处理，因为这样可以使用位操作来取出整数中对应的位，效率比取十进制的位要高。通常，基数排序在数据非常多的时候排序效率才会很高，一般在几十万条记录时，基数排序的效率比归并排序好。

从上面的分析知道，基数排序过程中有两个基本操作，一个是将数据根据基数分类放到对应的箱子中，这个操作被称为分配操作；另一个是依次将各个箱子的数按顺序连成一个链，这个操作被称为收集操作。设计代码时要为这两个操作各设计一个函数。

下面给出单向链表基数排序的编码实现。

```
/** 对链表的数据的第 uKeyIndex 位上的元素进行分类，依照它们的大小
    放入对应的箱子
    @param SINGLELIST *pSingleList——单向链表指针
    @param UINT uRadix——基数排序的基数，与具体数据类型有关，一般来讲整数的基
```

```
                                    数为 16，字符串的基数最大为 255
    @param UINT uKeyIndex——第多少位
    @param SINGLENODE **ppHead——用来记录头指针的箱子
    @param SINGLENODE **ppTail——记录箱子的尾指针
    @param GETKEYFUNC GetKeyFunc——获取数据的第 uKeyIndex 位上的元素值
    @return void——无
*/
static void SingleList_Distribute( SINGLELIST *pSingleList，UINT uRadix，
                                   UINT   uKeyIndex，SINGLENODE **ppHead，
                                   SINGLENODE **ppTail，GETKEYFUNC GetKeyFunc )
{
    SINGLENODE *pNode；
    UINT i；
    /* 初始化子表 */
    for ( i = 0； i < uRadix； i++ )
    {
        ppHead[i] = NULL；
        ppTail[i] = NULL；
    }
    pNode = pSingleList->pHead；
    while ( pNode != NULL )
    {
        UINT uRadixIndex = (*GetKeyFunc)(pNode->pData，uKeyIndex)；
        if ( ppHead[uRadixIndex] == NULL )
        {
            ppHead[uRadixIndex] = pNode；
            ppTail[uRadixIndex] = pNode；
            pNode = pNode->pNext；
            ppTail[uRadixIndex]->pNext = NULL；
        }
        else
        {
            ppTail[uRadixIndex]->pNext = pNode；
            ppTail[uRadixIndex] = ppTail[uRadixIndex]->pNext；
            pNode = pNode->pNext；
            ppTail[uRadixIndex]->pNext = NULL；
        }
    }
}
```

```
/** 对基数排序中分好类的箱子进行收集操作，将箱中数据按序重新连成一条链
    @param SINGLELIST *pSingleList——单向链表指针
    @param UINT uRadix——基数
    @param SINGLENODE **ppHead——用来记录头指针的箱子
    @param SINGLENODE **ppTail——记录箱子的尾指针
    @return void——无
*/
static void SingleList_Collect( SINGLELIST *pSingleList，UINT uRadix，
                                SINGLENODE **ppHead，SINGLENODE **ppTail )
{
    SINGLENODE *pHead；
    SINGLENODE *pTail；
    UINT uRadixIndex；
    /* 查找第 1 个非空子表 */
    uRadixIndex = 0；
    while ( uRadixIndex < uRadix )
    {
        if ( ppHead[uRadixIndex] == NULL )
        {
            uRadixIndex++；
            continue；
        }
        else
        {
            break；
        }
    }
    if ( uRadixIndex == uRadix )
    {
        /* 没有找到非空子表 */
        return ；
    }
    pHead = ppHead[uRadixIndex]；
    pTail = ppTail[uRadixIndex]；
    while ( uRadixIndex < uRadix－1 )
    {
        /* 继续查找下一个非空子表 */
        ++uRadixIndex；
        if ( ppHead[uRadixIndex] == NULL )
```

```
            {
                continue；
            }
            if ( uRadixIndex < uRadix )
            {
                /* 找到了非空子表，需要把它和前一个非空子表链接起来 */
                pTail->pNext = ppHead[uRadixIndex]；
                pTail = ppTail[uRadixIndex]；
            }
        }
        pSingleList->pHead = pHead；
        pSingleList->pTail = pTail；
    }

    /** 单向链表的基数排序函数
        @param SINGLELIST *pSingleList——单向链表指针
        @param UINT uRadix——基数，字符串如果以单字节为基数，则基数为 256；整数以
                             十进制计算基数，则基数为 10
        @param UINT uMaxKeyLen——关键词的长度，字符串以字节为单位则长度为字符串本身
                                 最大可能长度，如果 32 位整数以十六进制为单位，则最
                                 大长度为 8
        @param GETKEYFUNC GetKeyFunc——关键词获取回调函数
        @return INT——返回 CAPI_FAILED 表示失败；返回 CAPI_SUCCESS 表示成功
    */
    INT SingleList_RadixSort(SINGLELIST *pSingleList，UINT uRadix，UINT uMaxKeyLen，
                             GETKEYFUNC GetKeyFunc )
    {
        SINGLENODE **ppHead；      /* 用来记录各个箱子头节点的双指针 */
        SINGLENODE **ppTail；      /* 用来记录各个箱子尾节点的双指针 */
        UINT i；                   /* 临时循环变量 */
        /* 给箱子申请内存 */
        ppHead = (SINGLENODE **)malloc( uRadix *sizeof(SINGLENODE *) )；
        ppTail = (SINGLENODE **)malloc( uRadix *sizeof(SINGLENODE *) )；
        if ( ppHead == NULL || ppTail == NULL )
        {
            return CAPI_FAILED；
        }
        /* 按顺序对关键字的第 i 位进行分配和收集操作 */
        for ( i = 0； i < uMaxKeyLen； i++ )
```

```
    {
        SingleList_Distribute(pSingleList，uRadix，i，ppHead，ppTail，GetKeyFunc)；
        SingleList_Collect(pSingleList，uRadix，ppHead，ppTail)；
    }
    /* 释放分配的箱子 */
    free( ppHead )；
    free( ppTail )；
    return CAPI_SUCCESS；
}
```

3.4 双向链表

3.4.1 双向链表的基本概念

在 3.3 节介绍的单向链表中，要在链表的尾部删除或弹出数据，就要取得尾部节点的前一节点，在单向链表中只能从头部节点开始遍历链表寻找尾部节点的前一节点。当链表中节点较少时是没有问题的，但节点较多时就不适用了。如果由尾部节点指针能够直接得到它的前一节点，那么就不需要从链表头部进行遍历了。要想从尾部节点开始寻找它的前一节点，在尾部节点中必须有一个指针指向它的前一节点，也就是说在链表的节点中要有一个前向指针来指向它的前一节点。这种每个节点既有指向后一节点的指针又有指向前一节点的指针的链表称为双向链表。用双向链表链接各节点的方式如图 3-5 所示。

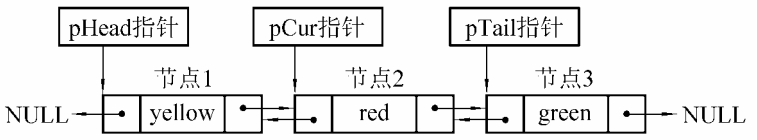


图 3-5 双向链表示意图

3.4.2 双向链表的设计

双向链表和单向链表的区别是双向链表除了有单向链表的后向指针外，还有一个前向指针，因此双向链表的节点可以用 C 语言数据结构描述如下。

```
typedef struct DOUBLENODE_st {
    struct DOUBLENODE_st *pNext；      /* 下一个节点指针 */
    struct DOUBLENODE_st *pPrev；      /* 前一个节点指针 */
    void *pData；                      /* 数据指针 */
```

```
} DOUBLENODE；
```

有了双向链表的节点定义后，再来设计双向链表的数据结构，用结构体描述如下。

```
typedef struct DOUBLELIST_st {
    DOUBLENODE *pHead；     /* 第 1 个节点的指针 */
    DOUBLENODE *pTail；     /* 最后 1 个节点的指针 */
    DOUBLENODE *pCur；      /* 当前节点的指针 */
    UINT uCount；           /* 保存链表节点的个数 */
} DOUBLELIST；
```

这个设计和单向链表一样，有一个链表头节点指针和一个链表尾节点指针。有很多书还介绍环形链表，环形链表的尾节点的下一节点指向头节点，形成了一个环。本书中链表的尾部节点的下一节点为空，因此本书中的链表不是环形链表。但由于本书中的链表既有头节点，又有尾节点，因此也可以实现环形链表的功能。

本书这种链表设计和环形设计的区别是：当要对链表进行遍历时，判断链表是否到了尾部，环形链表中必须要判断下一节点是不是头节点，如果是头节点就表明到了链表的尾部。取头节点可以直接访问 pHead 指针，或者用一个 GetHead()函数来获取。直接访问 pHead 指针，破坏了数据封装特性，如果使用 GetHead()则效率又很低；而在本书设计中，判断链表是否到尾部只要判断下一节点是否为空即可。在多任务环境下，环形链表的 GetHead()函数里还要加锁保护，效率就更低了。在本书的设计中，因为只是和空节点指针进行比较，而空节点指针 NULL 是一个常量，不存在保护问题，因此这种设计比环形设计有更优的性能。

双向链表的接口和单向链表基本相同，只是实现时由于双向链表的节点多了一个 pPrev 前向指针。多一个前向指针的缺点是额外开销比单向链表增加了一倍，但优点是删除尾部节点时不需要从头开始去找尾部节点的上一节点，还有，如果知道某个节点的指针再将其弹出来也非常容易。因此，双向链表和单向链表比起来，空间效率降低了，但在某些操作上的时间效率大大提高了。另外，由于有前向指针，所以在对链表做遍历操作时，可以从头开始依次向后进行遍历，也可以从尾部开始依次向前进行遍历操作。要提一下的是本书的代码只给出了向后遍历操作代码，向前遍历操作代码类似，不再给出，读者有兴趣可以自己仿照向后遍历代码写出来。

3.4.3 双向链表的编码实现

```
/** 双向链表的创建函数，创建后链表还是空的，没有节点在里面
    @param void——无
    @return DOUBLELIST *——失败返回 NULL；成功时返回一个双向链表结构体指针
```

```
*/
DOUBLELIST *DoubleList_Create( void )
{
    DOUBLELIST *pList；
    /* 分配内存操作 */
    pList = (DOUBLELIST *)malloc(sizeof(DOUBLELIST))；
    if ( pList != NULL )
    {
        /* 初始化链表结构体各指针成员为空，链表节点个数为 0 */
        pList->pHead = NULL；
        pList->pTail = NULL；
        pList->pCur = NULL；
        pList->uCount = 0；
    }
    return pList；
}

/** 双向链表的释放函数
    @param DOUBLELIST *pList——要释放的双向链表的指针
    @param DESTROYFUNC pDestroyFunc——链表节点数据释放回调函数
    @return void——无
*/
void DoubleList_Destroy( DOUBLELIST *pList，DESTROYFUNC DestroyFunc )
{
    DOUBLENODE *pNode；
    if ( pList )
    {
        /* 从头节点开始，一个接一个释放链表节点及节点数据 */
        pNode = pList->pHead；
        while ( pNode != NULL )
        {
            DOUBLENODE *pDelNode；
            pDelNode = pNode；
            pNode = pNode->pNext；
            if ( DestroyFunc != NULL && pDelNode->pData != NULL )
            {
                /* 释放数据 */
                (*DestroyFunc)( pDelNode->pData )；
            }
```

```
            free( pDelNode )；  /* 释放节点 */
        }
        /* 释放链表结构体 */
        free( pList )；
    }
}

/** 双向链表的添加节点函数，添加的节点放在双向链表的头部
    @param DOUBLELIST *pList——要添加的双向链表指针
    @param void *pData——要添加的节点的数据指针
    @return INT——失败返回 CAPI_FAILED；成功返回 CAPI_SUCCESS
*/
INTDoubleList_InsertHead( DOUBLELIST *pList，void *pData )
{
    DOUBLENODE *pNode；
    /* 参数校验 */
    if ( pList == NULL || pData == NULL )
    {
        return CAPI_FAILED；
    }
    /* 新建一个节点 */
    pNode = (DOUBLENODE *)malloc( sizeof(DOUBLENODE) )；
    if ( pNode == NULL )
    {
        return CAPI_FAILED；
    }
    pNode->pData = pData；   /* 将节点数据指针指向传进来的数据 */
    /* 将新建节点的下一节点指针指向头节点，并将头节点改为新建的节点 */
    pNode->pNext = pList->pHead；
    pNode->pPrev = NULL；
    if ( pList->pHead != NULL )
    {
        pList->pHead->pPrev = pNode；
    }
    pList->pHead = pNode；
    /* 判断尾节点指针是否为空，如果为空表示原来链表中没有节点，此时应该将尾节点
    *  指向新加入的节点
    */
```

```
    if ( pList->pTail == NULL )
    {
        pList->pTail = pNode；
    }
    /* 将链表节点数据加 1 */
    pList->uCount++；
    return CAPI_SUCCESS；
}

/** 双向链表的添加节点函数，添加的节点放在双向链表的尾部
    @param DOUBLELIST *pList——要添加的双向链表指针
    @param void *pData——要添加的节点的数据指针
    @return INT——失败返回 CAPI_FAILED；成功返回 CAPI_SUCCESS
*/
INTDoubleList_InsertTail( DOUBLELIST *pList，void *pData )
{
    DOUBLENODE *pNode；
    /* 参数校验 */
    if ( pList == NULL || pData == NULL )
    {
        return CAPI_FAILED；
    }
    /* 新建一个节点 */
    pNode = (DOUBLENODE *)malloc( sizeof(DOUBLENODE) )；
    if ( pNode == NULL )
    {
        return CAPI_FAILED；
    }
    pNode->pData = pData；  /* 将节点数据指针指向传进来的数据 */
    pNode->pNext = NULL；  /* 将节点的下一节点赋为空指针 NULL */
    pNode->pPrev = pList->pTail；
    /* 判断尾节点指针是否为空，如果为空表示原来链表中没有节点，此时应该
    *  将尾节点指向新加入的节点，并且头节点指针也应该指向新节点
    */
    if ( pList->pTail == NULL )
    {
        pList->pHead = pNode；
    }
```

```
    else
    {
        /* 如果尾节点指针不为空，此时应该将尾节点下一节点指针指向新加入的
        *  节点，并且尾节点指针也应该指向新节点
        */
        pList->pTail->pNext = pNode；
    }
    pList->pTail = pNode；
    /* 将链表节点数据加 1 */
    pList->uCount++；
    return CAPI_SUCCESS；
}

/** 双向链表的弹出头节点函数
    @param DOUBLELIST *pList——要操作的双向链表指针
    @return void *——失败返回 NULL；成功返回要弹出的头节点的数据指针
*/
void *DoubleList_PopHead( DOUBLELIST *pList )
{
    DOUBLENODE  *pPopNode；     /* 用来指向要弹出数据的节点的指针 */
    void *pPopData；            /* 用来指向要弹出的数据的指针 */
    /* 参数校验 */
    if ( pList == NULL || pList->pHead == NULL )
    {
        return NULL；
    }
    /* 将要弹出数据的节点指针指向链表头节点，弹出数据指针指向头节点的数据 */
    pPopNode = pList->pHead；
    pPopData = pPopNode->pData；
    /* 判断当前节点指针是否指向头节点，如果指向头节点则需要将其
    *  指向头节点的下一节点
    */
    if ( pList->pCur == pList->pHead )
    {
        pList->pCur = pList->pHead->pNext；
    }
    /* 将头节点指针指向头节点的下一节点 */
    pList->pHead = pList->pHead->pNext；
```

```
    if ( pList->pHead != NULL )
    {
        pList->pHead->pPrev = NULL；
    }
    /* 将链表节点数量减 1 */
    pList->uCount－－；
    /* 如果链表的节点数量已经为 0 则表示原来只有一个节点，弹出头节点后，链表已经
     * 为空，此时应该将尾节点指针赋空。当前节点指针由于前面已经处理过了，如果只
     * 有一个节点，肯定为空，所以这里不需要处理当前节点指针
     */
    if ( pList->uCount == 0 ) {
        pList->pTail = NULL；
    }
    /* 释放弹出的节点，注意这里并没有释放节点数据指针 */
    free( pPopNode )；
    return pPopData；       /* 返回头节点的数据指针 */
}

/** 双向链表的弹出尾节点函数，这个函数由于要查找尾节点的前一节点，因此效率很低
    @param DOUBLELIST *pList——要操作的双向链表指针
    @return void *——失败返回 NULL；成功返回要弹出的尾节点的数据指针
*/
void *DoubleList_PopTail( DOUBLELIST *pList )
{
    DOUBLENODE *pPopNode；   /* 用来指向要弹出数据的节点的指针 */
    void *pPopData；         /* 用来指向要弹出的数据的指针 */
    /* 参数校验 */
    if ( pList == NULL || pList->pHead == NULL )
    {
        return NULL；
}
    /* 将要弹出数据的节点指针指向链表头节点，弹出数据指针指向头节点的数据 */
    pPopNode = pList->pTail；
    pPopData = pPopNode->pData；
    /* 判断当前节点指针是否指向尾节点，如果指向头节点，则需要将其指向空节点
    */
    if ( pList->pCur == pList->pTail )
    {
```

```
        pList->pCur = NULL；
    }
    /* 如果链表的头节点和尾节点相同则表示原来只有一个节点，弹出尾节点后，链表已经
    * 为空，此时应该将头节点指针赋空。当前节点指针由于前面已经处理过了，如果只
    * 有一个节点，则肯定为空，所以这里不需要处理当前节点指针
    */
    if ( pList->pTail == pList->pHead )
    {
        pList->pHead = NULL；
    }
    else
    {
        pList->pTail->pPrev->pNext = NULL；
    }
    pList->pTail = pList->pTail->pPrev；
    /* 将链表节点数量减 1 */
    pList->uCount－－；
    /* 释放弹出的节点，注意这里并没有释放节点数据指针 */
    free( pPopNode )；
    return pPopData；       /* 返回头节点的数据指针 */
}

/** 链表的删除节点函数，它将删除和 pMatchData 参数有相同数据的节点，如果有许
    多数据相同的节点，它将只删除第一个有相同数据的节点
    @param DOUBLELIST *pList——要操作的双向链表指针
    @param void *pMatchData——要删除节点的匹配数据
    @param COMPAREFUNC CompareFunc——数据比较函数用来比较 pMatchData 参数和链
                                    表节点参数是否相等
    @param DESTROYFUNC DestroyFunc——链表节点的数据释放函数
    @return INT (by default) ——返回 CAPI_FAILED 表示失败或链表中没有匹配的数据；
                              返回 CAPI_SUCCESS 表示成功删除
*/
INT  DoubleList_Delete( DOUBLELIST *pList，void *pMatchData，
                        COMPAREFUNC CompareFunc，DESTROYFUNC DestroyFunc )
{
    DOUBLENODE *pNode；
    DOUBLENODE *pPrevNode；
    /* 参数校验 */
    if ( pList == NULL || CompareFunc == NULL )
```

```
{
    return CAPI_FAILED；
}
pNode = pList->pHead；
pPrevNode = pNode；
while ( pNode != NULL )
{
    /* 比较节点数据是否匹配 */
    if ( (*CompareFunc)( pNode->pData，pMatchData ) == 0 )
    {
        if ( pPrevNode == pNode )
        {
            /* 头节点匹配上了，需要删除头节点 */
            pList->pHead = pNode->pNext；
            if ( pList->pHead != NULL )
            {
                pList->pHead->pPrev = NULL；
            }
            else
            {
                /* 链表里只有一个节点，此时需要将链表尾节点指针和当前节点
                * 指针赋空
                */
                pList->pTail = NULL；
                pList->pCur = NULL；
            }
        }
        else
        {
            pPrevNode->pNext = pNode->pNext；
            if ( pNode->pNext != NULL )
            {
                pNode->pNext->pPrev = pPrevNode；
            }
            if ( pList->pTail == pNode )
            {
                /* 如果尾节点和 pNode 相同，表明删除的是尾节点，此时需要将
```

```
                    *  尾节点指针指向要删除节点的前一个节点
                    */
                    pList->pTail = pPrevNode；
                    }
                if ( pList->pCur == pNode )
                {
                    /* 如果链表当前节点和 pNode 相同，表明删除的是当前节点，此时
                    *  需要将当前节点指针指向要删除节点的前一个节点
                    */
                    pList->pCur = pNode->pNext；
                }
            }
            /* 释放节点数据和节点占用的内存 */
            if ( DestroyFunc != NULL && pNode->pData != NULL )
            {
                (*DestroyFunc)( pNode->pData )；
            }
            free( pNode )；
            break；
        }
        pPrevNode = pNode；
        pNode = pNode->pNext；
    }
    return CAPI_SUCCESS；
}

/** 双向链表的获取链表节点数量函数
    @param DOUBLELIST *pList——要操作的双向链表指针
    @return UINT——链表节点数量，为 0 表示链表为空或者参数非法
*/
UINT DoubleList_GetCount(DOUBLELIST *pList)
{
    if ( pList == NULL )
    {
        return 0；
    }
    return pList->uCount；
}
```

```
/** 双向链表的获取头节点函数
    @param DOUBLELIST *pList——要操作的双向链表指针
    @return void *——头节点的数据指针
*/
void *DoubleleList_GetHead( DOUBLELIST *pList )
{
    if ( pList == NULL )
    {
        return NULL；
    }
    if ( pList->pHead == NULL )
    {
        return NULL；
    }
    return pList->pHead->pData；
}

/** 双向链表的获取尾节点函数
    @param DOUBLELIST *pList——要操作的双向链表指针
    @return void *——尾节点的数据指针
*/
void *DoubleList_GetTail( DOUBLELIST *pList )
{
    if ( pList == NULL )
    {
        return NULL；
    }
    if ( pList->pTail != NULL )
    {
        return pList->pTail->pData；
    }
    else
    {
        return NULL；
    }
}

/** 双向链表的查找函数
    @param DOUBLELIST *pList——要操作的双向链表指针
```

```
    @param void *pMatchData——要查找的匹配数据
    @param COMPAREFUNC CompareFunc——数据匹配比较函数
    @return void *——查找到的在双向链表中的匹配数据
*/
void *DoubleList_Find( DOUBLELIST *pList，void *pMatchData，
                       COMPAREFUNC CompareFunc )
{
    DOUBLENODE  *pNode；
    pNode = pList->pHead；
    while ( pNode )
    {
        if ( (*CompareFunc)( pNode->pData，pMatchData ) == 0)
        {
            void *pData；
            pData = pNode->pData；
            return pData；
        }
        pNode = pNode->pNext；
    }
    return NULL；
}

/** 双向链表的枚举初始化函数
    @param SINGLELIST *pSingleList——要操作的双向链表指针
    @return void——无
*/
void DoubleList_EnumBegin( DOUBLELIST *pList )
{
    pList->pCur = pList->pHead；
    return；
}

/** 双向链表枚举下一个节点的函数，第一次调用此函数前必须
    先调用 DoubleList_EnumBegin()函数
    @param DOUBLELIST *pList——要操作的双向链表指针
    @return void *——枚举到的节点数据指针
*/
void *DoubleList_EnumNext( DOUBLELIST *pList )
{
```

```
    DOUBLENODE    *pCur；
    pCur = pList->pCur；
    if ( pCur != NULL )
    {
        pList->pCur = pCur->pNext；
        return pCur->pData；
    }
    return NULL；
}

/** 双向链表的枚举节点函数，第一次调用此函数前要先调用 DoubleList_EnumBegin()函数
    @param DOUBLELIST *pList——要操作的双向链表指针
    @return DOUBLENODE *——从双向链表中枚举到的节点指针
*/
DOUBLENODE *DoubleList_EnumNode( DOUBLELIST *pList )
{
    DOUBLENODE    *pCur；
    pCur = pList->pCur；
    if ( pCur != NULL )
    {
        pList->pCur = pCur->pNext；
        return pCur；
    }
    return NULL；
}

/** 双向节点的指定节点删除函数
    @param DOUBLELIST *pList——要操作的双向链表指针
    @param DOUBLENODE *pNode——要弹出的指定节点指针
    @return DOUBLENODE *——弹出的节点指针，和 pNode 相同
*/
DOUBLENODE *DoubleList_PopNode( DOUBLELIST *pList，DOUBLENODE *pNode )
{
    /* 修改前一个节点的后向指针 */
    if ( pNode->pPrev != NULL )
    {
        pNode->pPrev->pNext = pNode->pNext；
    }
    /* 修改后一个节点的前向指针 */
```

```
    if ( pNode->pNext != NULL )
    {
        pNode->pNext->pPrev = pNode->pPrev；
    }
    /* 判断 pCur 指针是否指向弹出节点，如果是则需更新 pCur */
    if ( pList->pCur == pNode )
    {
        pList->pCur = pNode->pNext；
    }
    /* 返回传入的 pNode 指针 */
    return pNode；
}

/** 双向链表的插入排序函数，排序是按照从小到大进行排列，由 CompareFunc 来决定大
    小，因此用户可以通过 CompareFunc 的返回值设置来决定使用升序或降序排序
    @param DOUBLELIST *pList——要操作的双向链表指针
    @param COMPAREFUNC CompareFunc——节点数据比较函数
    @return INT——成功返回 1；失败返回 0
*/
INT DoubleList_InsertSort(DOUBLELIST *pList，COMPAREFUNC CompareFunc )
{
    DOUBLENODE *pNode；          /* 用来遍历 pList 的临时指针 */
    DOUBLENODE *pPrevNode；    /* pNode 的前一节点指针 */
    pNode = pList->pHead；
    pPrevNode = NULL；
    if ( pNode == NULL )
    {
        /* 链表中没有节点，就把它当作已经排好了序 */
        return CAPI_SUCCESS；
    }
    while ( pNode->pNext != NULL )
    {
        DOUBLENODE   *pTempNode；
        pTempNode = pList->pHead；
        pPrevNode = NULL；
        while ( pTempNode != pNode->pNext )
        {
            if ( (*CompareFunc)( pNode->pNext->pData，pTempNode->pData ) < 0 )
            {
```

```
            DOUBLENODE    *pCurNode = pNode->pNext；
            /* 将 pCurNode 节点弹出来 */
            pNode->pNext = pNode->pNext->pNext；
            if ( pNode->pNext != NULL )
            {
                pNode->pNext->pPrev = pNode；
            }
            /* 执行插入操作 */
            if ( pPrevNode != NULL ) /* 插入不在头节点前的情况 */
            {
                /* 将 pNode->pNext 插入 pTempNode 之前 */
                pPrevNode->pNext = pCurNode；
                pCurNode->pPrev = pPrevNode；
                pCurNode->pNext = pTempNode；
                pTempNode->pPrev = pCurNode；
            }
            else /* 插入在头节点前的情况 */
            {
                /* 将 pNode->pNext 节点变为头节点 */
                pCurNode->pNext = pTempNode；
                pCurNode->pPrev = NULL；
                pList->pHead->pPrev = pCurNode；
                pList->pHead = pCurNode；
            }
            /* 修改尾指针指向 */
            if ( pCurNode == pList->pTail )
            {
                pList->pTail = pNode；
            }
            break；
        }
        pPrevNode = pTempNode；
        pTempNode = pTempNode->pNext；
    } /* while ( pTempNode != pNode->pNext ) */
    if ( pTempNode == pNode->pNext ) /* 没有插入的情况 */
    {
        pNode = pNode->pNext；
    }
    else /* 已经插入的情况 */
```

```
            {
                /* 插入后不需将 pNode 指针后移，因为前面操作过程中已经移动过了 */
            }
        } /* while ( pNode->pNext != NULL ) */
        return 1；
    }

    /** 将两个已经排好序的链表进行合并，两个链表都必须是按从小到大进行排列，即尾节
        点中数据取值最大，合并后的链表也是从小到大进行排列
        @param DOUBLELIST *pListA——要合并的链表 A
        @param DOUBLELIST *pListB——要合并的链表 B
        @param COMPAREFUNC CompareFunc——节点比较函数
        @return INT——返回 0 表示失败，失败时参数没有变化；返回 1 表示成功，成功
                      时 pListA 是合并后的结果，pListB 会被释放
    */
    INT DoubleList_Merge( DOUBLELIST *pListA，DOUBLELIST *pListB，
                          COMPAREFUNC CompareFunc)
    {
        DOUBLENODE *pNodeA； /* 用来指向链表 pListA 的节点的临时指针 */
        DOUBLENODE *pNodeB； /* 用来指向链表 pListB 的节点的临时指针 */
        DOUBLENODE *pPrevA； /* pNodeA 的前一节点指针 */
        pNodeA = pListA->pHead；
        pNodeB = pListB->pHead；
        pPrevA = NULL；
        while ( pNodeB != NULL )
        {
            while ( pNodeA != NULL )
            {
                if ( (*CompareFunc)( pNodeA->pData，pNodeB->pData ) >= 0 )
                {
                    DOUBLENODE *pNode；
                    /* 将 pNodeB 弹出来保存到 pNode 中 */
                    pNode = pNodeB；
                    pNodeB = pNodeB->pNext；
                    /* 将 pNode 插入到 pNodeA 前面 */
                    if ( pPrevA == NULL )
                    {
                        /* 插入在头指针前的情况，需要修改头指针 */
                        pListA->pHead = pNode；
```

```
                pNode->pNext = pNodeA；
                pNodeA->pPrev = pNode；
            }
            else
            {
                pPrevA->pNext = pNode；
                pNode->pPrev = pPrevA；
                pNode->pNext = pNodeA；
                pNodeA->pPrev = pNode；
            }
            pPrevA = pNode；
            break；
        }
        pPrevA = pNodeA；
        pNodeA = pNodeA->pNext；
    }
    /* 如果 pListB 的所有数据都大于 pListA 的数据，将 pListB 插入到 pListA 尾部
    */
    if ( pNodeA == NULL )
    {
        pListA->pTail->pNext = pNodeB；
        pNodeB->pPrev = pListA->pTail；
        pListA->pTail = pListB->pTail；
        break；
    }
}
/* 修改 pListA 的节点总数量 */
pListA->uCount += pListB->uCount；
free( pListB )；
return 1；
}

/** 将一个双向链表劈成两个双向链表，只是从原来链表中间断开，劈完后原链表变成劈
    开后的第一条链表
    @param DOUBLELIST *pList——要被劈开的双向链表
    @param UINT nCount——劈开后第一个链表的节点个数，必须小于原链表节点个数
    @return DOUBLELIST *——成功时返回劈开后的第二条链表；失败返回 NULL
*/
```

```
DOUBLELIST *DoubleList_Split(DOUBLELIST *pList，UINT uCount)
{
    DOUBLENODE *pNode；          /* 用来保存劈开处节点的指针 */
    DOUBLELIST *pSecondList；    /* 用来记录劈开后的第二条链表的指针 */
    UINT uIndex；                /* 临时循环体变量 */
    /* 创建一条空链表 */
    pSecondList = DoubleList_Create()；
    if ( pSecondList == NULL )
    {
        return NULL；
    }
    /* 参数校验 */
    if ( uCount == 0 || pList->uCount <= uCount)
    {
        return NULL；
    }
    /* 获取要劈开的位置 */
    pNode = pList->pHead；
    for ( uIndex = 1； uIndex < uCount； uIndex++ )
    {
        pNode = pNode->pNext；
    }
    /* 填充第二条链表的内容 */
    pSecondList->pHead = pNode->pNext；
    pSecondList->pTail = pList->pTail；
    pSecondList->uCount = pList->uCount - uCount；
    /* 修改第一条链表的内容 */
    pList->pTail = pNode；
    pList->uCount = uCount；
    /* 将第一条链表尾节点的 pNext 指针指向 NULL */
    pList->pTail->pNext = NULL；
    /* 将第二条链表的头节点的前一节点指针赋成空 */
    pSecondList->pHead->pPrev = NULL；
    return pSecondList；
}

/** 对链表使用归并插入排序，归并和插入排序结合使用，先使用归并排序，当链表里面
    元素个数小于一定值时便使用插入排序
    @param DOUBLELIST *pList——要排序的链表指针
```

```
    @param COMPAREFUNC CompareFunc——链表节点比较函数
    @param UINT uInsertSortCount——使用插入排序时的链表节点个数，当链表节点
                                   个数小于这个值时会使用插入排序算法
    @return INT——返回 CAPI_FAILED 表示失败；返回 CAPI_SUCCESS 表示成功
*/
INT DoubleList_MergeSort( DOUBLELIST *pList，COMPAREFUNC CompareFunc，
                          UINT uInsertSortCount)
{
    DOUBLELIST *pSecondList；
    /* 参数校验 */
    if ( pList == NULL || CompareFunc == NULL )
    {
        return CAPI_FAILED；
    }
    if ( pList->uCount < 2 )
    {
        /* 如果节点个数少于两个，认为是已经排好序的 */
        return CAPI_SUCCESS；
    }
    /* 如果链表节点个数小于给定的做插入排序的个数，直接使用插入排序
    */
    if ( pList->uCount <= uInsertSortCount )
    {
        (void)DoubleList_InsertSort( pList，CompareFunc )；
    }
    else
    {
        /* 将链表劈成两半 */
        pSecondList = DoubleList_Split(pList，(pList->uCount) / 2 )；
        /* 对劈完后的第一个链表进行递归调用归并排序 */
        (void)DoubleList_MergeSort( pList，CompareFunc，uInsertSortCount )；
        /* 对劈完后的第二个链表进行递归调用归并排序 */
        (void)DoubleList_MergeSort( pSecondList，CompareFunc，uInsertSortCount )；
        /* 将排好序的两个链表合成一个 */
        (void)DoubleList_Merge( pList，pSecondList，CompareFunc )；
    }
    return CAPI_SUCCESS；
```

```
}

/** 双向链表的遍历函数
    @param DOUBLELIST *pList——要操作的双向链表指针
    @param TRAVERSEFUNC TraverseFunc——节点数据的遍历操作函数
    @return INT——成功返回 1；失败返回 0
*/
void DoubleList_Traverse( DOUBLELIST *pList，TRAVERSEFUNC TraverseFunc )
{
    DOUBLENODE *pNode；
    pNode = pList->pHead；
    /* 开始遍历链表 */
    while ( pNode != NULL )
    {
        (*TraverseFunc)( pNode->pData )； /* 调用遍历回调函数处理数据 */
        pNode = pNode->pNext；
    }
    return；
}

/** 对链表的数据的第 uKeyIndex 位上的元素进行分类，依照它们的大小放入对应的箱子中
    @param DOUBLELIST *pList——要操作的双向链表指针
    @param UINT uRadix——基数排序的基数，与具体数据类型有关，一般来讲整数的
                         基数为 16，字符串的基数最大为 255
    @param UINT uKeyIndex——第多少位
    @param DOUBLENODE **ppHead——用来记录头指针的箱子
    @param DOUBLENODE **ppTail——记录箱子的尾指针
    @param GETKEYFUNC GetKeyFunc——获取数据的第 uKeyIndex 位上的元素值
    @return void——无
*/
void DoubleList_Distribute( DOUBLELIST *pList，UINT uRadix，UINT uKeyIndex，
                            DOUBLENODE **ppHead，DOUBLENODE **ppTail，
                            GETKEYFUNC GetKeyFunc )
{
    DOUBLENODE *pNode；
    UINT i；
    /* 初始化子表 */
    for ( i = 0； i < uRadix； i++ )
    {
```

```
            ppHead[i] = NULL；
            ppTail[i] = NULL；
        }
        pNode = pList->pHead；
        while ( pNode != NULL )
        {
            UINT uRadixIndex = (*GetKeyFunc)(pNode->pData，uKeyIndex)；
            if ( ppHead[uRadixIndex] == NULL )
            {
                ppHead[uRadixIndex] = pNode；
                ppTail[uRadixIndex] = pNode；
                pNode = pNode->pNext；
                ppTail[uRadixIndex]->pNext = NULL；
                ppTail[uRadixIndex]->pPrev = NULL；
            }
            else
            {
                ppTail[uRadixIndex]->pNext = pNode；
                pNode->pPrev = ppTail[uRadixIndex]；
                ppTail[uRadixIndex] = pNode；
                pNode = pNode->pNext；
                ppTail[uRadixIndex]->pNext = NULL；
            }
        }
}

/** 对基数排序中分好类的箱子进行收集操作，将箱中数据按序重新连成一条链
    @param DOUBLELIST *pList——要操作的双向链表指针
    @param UINT uRadix——基数
    @param DOUBLENODE **ppHead——用来记录头指针的箱子
    @param DOUBLENODE **ppTail——记录箱子的尾指针
    @return void——无
*/
void DoubleList_Collect( DOUBLELIST *pList，UINT uRadix，DOUBLENODE **ppHead，
                         DOUBLENODE **ppTail )
{
    DOUBLENODE *pHead；
    DOUBLENODE *pTail；
    UINT uRadixIndex；
```

```
/* 查找第 1 个非空子表 */
uRadixIndex = 0；
while ( uRadixIndex < uRadix )
{
    if ( ppHead[uRadixIndex] == NULL )
    {
        uRadixIndex++；
        continue；
    }
    else
    {
        break；
    }
}
if ( uRadixIndex == uRadix )
{
    /* 没有找到非空子表 */
    return；
}
pHead = ppHead[uRadixIndex]；
pTail = ppTail[uRadixIndex]；
while ( uRadixIndex < uRadix－1 )
{
    /* 继续查找下一个非空子表 */
    ++uRadixIndex；
    if ( ppHead[uRadixIndex] == NULL )
    {
        continue；
    }
    if ( uRadixIndex < uRadix )
    {
        /* 找到了非空子表，需要把它和前一个非空子表链接起来 */
        pTail->pNext = ppHead[uRadixIndex]；
        pTail->pNext->pPrev = pTail；
        pTail = ppTail[uRadixIndex]；
    }
}
pList->pHead = pHead；
```

```
    pList->pTail = pTail；
    return；
}

/** 双向链表的基数排序算法排序函数
    @param DOUBLELIST *pList——要排序的双向链表指针
    @param UINT uRadix——基数，字符串如果以单字节为基数，则基数为 256；整数以
                         十进制计算基数，则基数为 10
    @param UINT uMaxKeyLen——关键词的长度，字符串以字节为单位则长度为字符串
                         本身最大可能长度，如果 32 位整数以十六进制为单
                         位，则最大长度为 8
    @param GETKEYFUNC GetKeyFunc——关键词获取回调函数
    @return INT——返回 CAPI_FAILED 表示失败；返回 CAPI_SUCCESS 表示成功
*/
INT DoubleList_RadixSort( DOUBLELIST *pList，UINT uRadix，UINT uMaxKeyLen，
                          GETKEYFUNC GetKeyFunc )
{
    DOUBLENODE **ppHead；   /* 用来记录各个箱子头节点的双指针 */
    DOUBLENODE **ppTail ；  /* 用来记录各个箱子尾节点的双指针 */
    UINT i；                /* 临时循环变量 */
    /* 给箱子申请内存 */
    ppHead = (DOUBLENODE **)malloc( uRadix *sizeof(DOUBLENODE *) )；
    ppTail = (DOUBLENODE **)malloc( uRadix *sizeof(DOUBLENODE *) )；
    if ( ppHead == NULL || ppTail == NULL )
    {
        return CAPI_FAILED；
    }
    /* 按顺序对关键字的第 i 位进行分配和收集操作 */
    for ( i = 0； i < uMaxKeyLen；i++ )
    {
        DoubleList_Distribute(pList，uRadix，i，ppHead，ppTail，GetKeyFunc)；
        DoubleList_Collect(pList，uRadix，ppHead，ppTail)；
    }
    /* 释放分配的箱子 */
    free( ppHead )；
    free( ppTail )；
    return CAPI_SUCCESS；
}
```

3.5 使用整块内存的链表

3.5.1 整块内存链表的基本概念

前面 3.1 和 3.4 节中讲的单向链表和双向链表在内存组织上都是离散的，很容易产生内存碎片，使内存性能降低，而许多类似服务器之类的软件通常要求要有好的内存性能，本节给出一个将整个链表的节点及数据全部放在一整块内存中的设计。

对于一个在整块内存中的单向链表，为了管理整个空间，必须把自由空间建成一个链表。用一个指针 pEmpty 指向这条自由空间链表的头部，用另外一个指针 pHead 指向有数据的链表的头部，当链表初始化时，pHead 为 NULL。有数据插入到链表时，从自由空间链表中删除一个节点，将数据拷贝到节点中，将节点加入 pHead 指向的链表中。

初始化后的链表情况如图 3-6 所示。

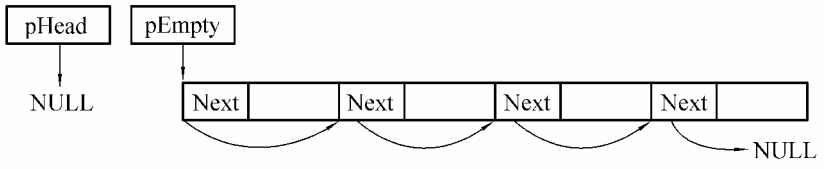


图 3-6 整块内存链表的初始化情况示意图

向链表中插入一个数据时，比如插入一个字符串“ Apple ”，如图 3-7 所示。

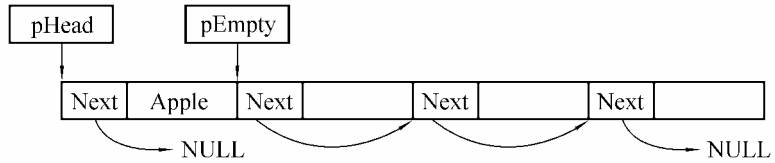


图 3-7 整块内存链表插入 1 个数据后的示意图

继续从链表头部插入数据“ Banana ”,“ Orange ”,“ Tomato ”时，如图 3-8 所示。

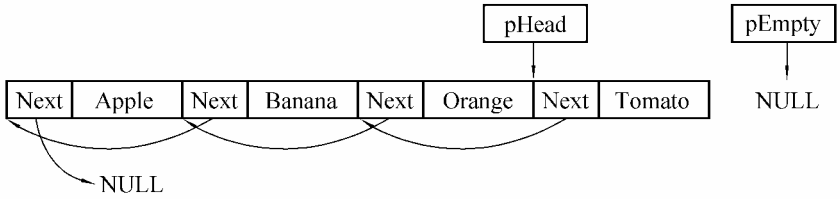


图 3-8 整块内存链表插入多个数据的示意图

如果将“Tomato”删除掉，情形如图 3-9 所示。

整块内存链表通常被用来进行内存的分配和释放，分配内存时，只要将 pEmpty 指向的内存块弹出来即可；释放时，将要释放的内存块重新插入到 pEmpty 指向的链表头部。这种内存释放分配算法非常高效，分配、释放只花费一次链表头部弹出和插入操作的时间，比其他类型的内存管理算法高多了，并且没有内存碎片的问题，因此本书第 9 章会使用这种思想来构造一个功能强大的动态等尺寸块的内存管理算法。

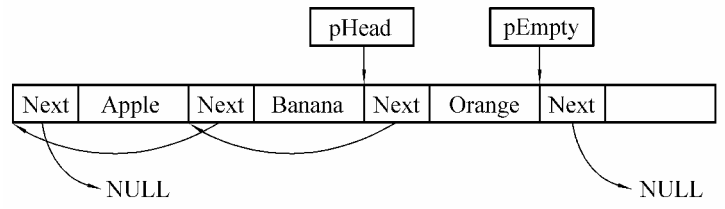


图 3-9　整块内存链表删除 1 个数据的示意图

3.5.2　整块内存链表的编码实现

从图 3-6 中可以看出，要设计一个整块内存中的链表需要一个头指针，一个指向自由空间的链表头指针，可以用 C 语言结构体来描述整块内存中链表的数据结构如下。

```
typedef struct BLOCKLIST_st {
    void *pBlock；                /* 整块内存指针 */
    SINGLENODE *pEmpty；          /* 自由空间表头指针 */
    SINGLENODE *pHead；           /* 链表头指针 */
    UINT uDataSize；              /* 数据大小 */
    UINT uMaxDataCount；          /* 最大的数据节点个数 */
    UINT uFreeCount；             /* 自由空间节点数量 */
} BLOCKLIST；
```

在整块内存链表的接口中，需要实现以下几个函数。

① 创建函数，负责创建和初始化工作；

② 释放函数，负责将整块内存链表释放掉；

③ 头部数据插入函数，负责插入数据到链表头部；

④ 头部数据删除函数，负责删除链表头部数据；

⑤ 分配内存函数，负责从整块内存链表中获取一块未使用内存块；

⑥ 释放内存函数，负责将一个已使用的内存块释放回整块内存链表的未使用空间中。

下面给出整块内存链表的编码实现。

```
#include <stdlib.h>
#include "CapiGlobal.h"
#include "BlockList.h"

/** 整块内存链表的创建函数
    @param UINT uDataSize——数据大小
    @param UINT uMaxDataCount——链表中最大可存放数据的个数
    @return BLOCKLIST *——整块内存链表指针
*/
BLOCKLIST *BlockList_Create(UINT uDataSize，UINT uMaxDataCount)
{
    BLOCKLIST *pList；
    SINGLENODE *pNode；
    UINT i；
    pList = (BLOCKLIST *)malloc( sizeof(BLOCKLIST)
          + uMaxDataCount *(uDataSize + sizeof(SINGLENODE)) )；
    if ( pList != NULL )
    {
        pList->pBlock = (void *)((char *)pList +sizeof(BLOCKLIST) )；
        /* 建立空链表 */
        pList->pEmpty = (SINGLENODE *)pList->pBlock；
        pNode = pList->pEmpty；
        for (i = 0； i < uMaxDataCount－1； i++)
        {
            pNode->pNext = (SINGLENODE *)((char *)pNode + sizeof(SINGLENODE)
                + uDataSize)；
            pNode = pNode->pNext；
        }
        pNode->pNext = NULL；
        pList->uFreeCount = uMaxDataCount；
        pList->uDataSize = uDataSize；
        pList->uMaxDataCount = uMaxDataCount；
        pList->pHead = NULL；
    }
    return pList；
}

/** 整块内存链表的释放函数
    @param BLOCKLIST *pList——整块内存链表指针
```

```
    @return void——无
*/
void BlockList_Destroy(BLOCKLIST *pList)
{
    if ( pList != NULL )
    {
        free( pList )；
    }
}

/**  整块内存链表的插入数据函数
    @param BLOCKLIST *pList——整块内存链表指针
    @param void *pData——要插入的节点数据指针
    @param UINT uDataLen——要插入的数据的长度
    @return INT——失败返回－1；成功返回 0
*/
INT BlockList_InsertHead(BLOCKLIST *pList，void *pData，UINT uDataLen)
{
    SINGLENODE *pNode；
    if ( uDataLen > pList->uDataSize )
    {
        return－1；
    }
    pNode = pList->pEmpty；
    if ( pNode != NULL )
    {
        pList->pEmpty = pNode->pNext；
    }
    else
    {
        return－1；
    }
    pNode->pNext = pList->pHead；
    memcpy(pNode->pData，pData，uDataLen)；
    pList->pHead = pNode；
    pList->uFreeCount－－；
    return 0；
}
```

```
/** 整块内存链表的删除头部节点函数
    @param BLOCKLIST *pList——整块内存链表指针
    @return void——无
*/
void   BlockList_DeleteHead(BLOCKLIST *pList)
{
    SINGLENODE *pPopNode；     /* 用来指向要弹出数据节点的指针 */
    /* 参数校验 */
    if ( pList == NULL || pList->pHead == NULL )
    {
        return；
    }
    /* 弹出链表头节点 */
    pPopNode = pList->pHead；
    pList->pHead = pList->pHead->pNext；
    /* 将弹出的节点加入到空链表的头部 */
    pPopNode->pNext = pList->pEmpty；
    pList->pEmpty = pPopNode；
    /* 将链表节点数量加 1 */
    pList->uFreeCount++；
    return；
}

/** 整块内存链表的内存分配函数，从空链中弹出一个节点，将其作为分配的内存
    @param BLOCKLIST *pList——整块内存链表指针
    @return void *——成功返回分配到的内存指针；失败返回 NULL
*/
void *BlockList_Alloc(BLOCKLIST *pList)
{
    SINGLENODE *pNode；
    pNode = pList->pEmpty；
    if ( pNode != NULL )
    {
        pList->pEmpty = pNode->pNext；
        pList->uFreeCount－－；
        return (void *)pNode；
    }
    return NULL；
}
```

```
/** 整块内存链表的内存释放函数，将内存释放，加入到空链中
    @param BLOCKLIST *pList——整块内存链表指针
    @param void *pData——要释放的数据指针
    @return void——无
*/
void BlockList_Free(BLOCKLIST *pList，void *pData)
{
    SINGLENODE *pNode；
    pNode= (SINGLENODE *)pData；
    pNode->pNext = pList->pEmpty；
    pList->pEmpty = pNode；
    pList->uFreeCount++；
}
```


3.6 实例：使用链表管理短信息系统的 CACHE

3.6.1 短信息系统的 CACHE 管理基本概念

开发一个短信息转发系统时，为了保证响应速度以及不丢失发过来的短信息，需要将收到的短信息请求暂时保存在内存，后台再由其他任务进行转发处理和写数据库操作，操作完成后将信息删除。

上例中，怎样去保存请求信息呢？初学者也许会觉得用一个足够大的数组来保存就可以了，但是，像短信息转发系统高峰时每秒钟要收到成千上万条信息，低谷时每秒钟也许只收到几十条信息，并且不同地方的高峰值也不一样，因此，决定数组的大小就成了很头疼的问题。也许有人会说可以用一个动态数组来保存，但这根本行不通，因为首先数组很大，如果超大后重新分配内存的开销实在太大了；另外还要考虑删除问题，比如开一个 100 万的数组来保存短信息，当数组按顺序用满后，这时候再来的短信息就要保存到那些已经被转发且被删除掉的信息的存储位置上，考虑到实际情况，删除时不可能按顺序删除，比如在 2 位置上保存短信息的目的手机出现通信故障暂时发不出去，但 3 位置上的信息可以发出去，3 位置上的信息被删除掉了而 2 位置上的信息被保存起来了，所以这个时候新来的短信息保存到数组的哪个位置上就成了问题，总不能从数组 0 位置开始一直按顺序搜索到空的位置吧，这样效率将很低。

实际应用中类似的例子比比皆是，如一个服务器软件，服务器每收到一个客户端请求，就要将请求的相关信息保存起来，然后由几个预先创建好的任务轮流去处理保存起来的请求，每处理完一个请求，需要将处理完的请求删除。怎样保存客户端请求呢？

解决上面这些问题的措施便是用本章要介绍的链表，链表可以说是软件工程师最常用的数据结构，也是必须掌握的最基本软件技能，在实际的商业软件中有着非常广泛的应用。下面就介绍如何使用链表来实现短信息系统的 CACHE 管理。

3.6.2 短信息系统的发送和接收分析

前面提到短信息系统中的 CACHE 管理可以使用链表来实现，这节介绍如何使用一个双向链表来实现对短信息系统的 CACHE 管理。在短信息系统中，通常有一个接收任务，负责接收用户发送的短信息；另外有一个发送任务，负责将收到的短信息发送到目标客户去。

接收任务接收到的短信息，可以保存在一条双向链表中，发送任务从这条双向链表中取数据发送出去。一般接收任务将数据保存在链表的尾部，而发送任务从链表头部取数据发出去。如果发送成功，则将头部数据删除掉，如果发送失败，则根据具体原因进行处理。发送失败的原因通常有以下几种。

① 目标手机关机；

② 网络故障；

③ 目标手机号不存在。

对于前两种情况，需要将头部取出的数据重新插入到链表尾部，便于以后重发；对于第三种情况，则直接丢弃该条短信息。

一条短信息通常包含发送时间、源手机号、目标手机号、短信息内容等信息，为了管理方便，还要给它加一个状态信息。

```
#define MAX_PHONE_NUMBER_LEN      16
typedef struct SHORTMSG_st {
    time_t time；
    char pszSrcPhone[MAX_PHONE_NUMBER_LEN]；
    char pszTagPhone[MAX_PHONE_NUMBER_LEN]；
    Char *pszMsg；
    INT nState；
} SHORTMSG；

typedef struct SHORTMSGCACHE_st {
    DOUBLELIST *pList；
} SHORTMSGCACHE；
```

nState 变量可以定义两种状态，一种是等待发送状态，一种是暂时无法发送状态。当然实际情况下暂时无法发送状态比较复杂，可以分解为更多的状态，比如处于关机

状态下如何进行重发，重发失败后如何处理等。根据重发的算法会需要定义更多的状态。本书为方便起见，暂时定义以上两种状态。

接收任务的处理如图 3-10 所示。

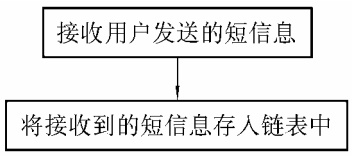


图 3-10　短信息接收任务处理图

发送任务的处理如图 3-11 所示。

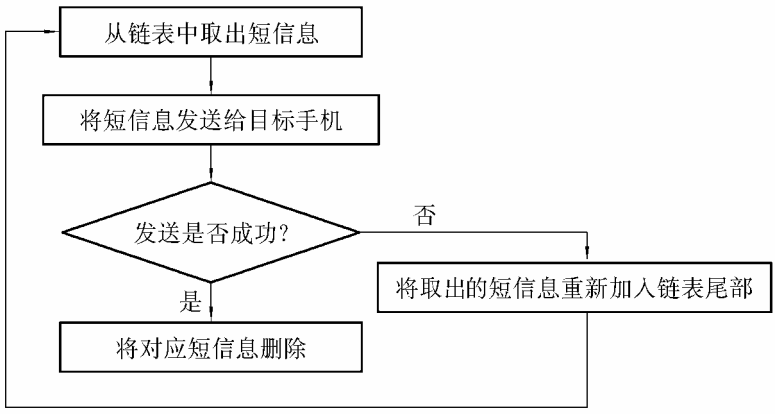


图 3-11　短信息发送任务处理图

3.6.3　短信息系统 CACHE 管理的编码实现

下面就来编码实现短信息系统的 CACHE 管理功能，通过这个功能来了解如何使用链表的操作函数。

```
/** 短信息发送的回调函数指针定义
    @param SHORTMSG——指向一条短信息的指针
    @return INT——返回 CAPI_FAILED 表示发送失败；返回 CAPI_SUCCESS 表示发送成功
*/
typedef INT (*SHORTMSGSENDFUNC)(SHORTMSG *pMsg)；

/** 短信息 CACHE 的创建函数
    @return SHORTMSGCACHE *——成功返回创建的短信息 CACHE 对象指针 失败返回 NULL
*/
```

```
SHORTMSGCACHE *ShortMsgCache_Create()
{
    SHORTMSGCACHE *pCache；
    pCache = (SHORTMSGCACHE *)malloc(sizeof(SHORTMSGCACHE))；
    if ( pCache != NULL )
    {
        pCache->pList = DoubleList_Create()；
        if ( pCache->pList == NULL )
        {
            free(pCache)；
            pCache = NULL；
        }
    }
    return pCache；
}

/** 短信息 CACHE 释放的回调函数，用来释放一条短信息
    @param void *p——指向一条短信息 SHORTMSG 的指针
    @return void——无
*/
void ShortMsgDestroy(void *p)
{
    SHORTMSG *pMsg = (SHORTMSG *)p；
    if ( pMsg != NULL )
    {
        if ( pMsg->pszMsg != NULL )
        {
            free( pMsg->pszMsg )；
        }
        free(pMsg)；
    }
}

/** 短信息 CACHE 的释放函数，将链表中保存的短信息全部释放
    @param SHORTMSGCACHE *pCache——要释放的短信息 CACHE 对象指针
    @return void——无
*/
void ShortMsgCache_Destroy(SHORTMSGCACHE *pCache)
{
```

```
    if ( pCache != NULL )
    {
        DoubleList_Destroy(pCache->pList，ShortMsgDestroy)；
        free(pCache)；
    }
}

/** 短信息 CACHE 接收函数，将接收到的短信息存入链表中
    @param SHORTMSGCACHE *pCache——短信息 CACHE 对象指针
    @param SHORTMSG *pMsg——短信息指针，指向一条短信息
    @return INT——返回 CAPI_FAILED 表示失败；返回 CAPI_SUCCESS 表示成功
*/
INT ShortMsgCache_Recv(SHORTMSGCACHE *pCache，SHORTMSG *pMsg)
{
    return DoubleList_InsertTail(pCache->pList，(void *)pMsg)；
}

/** 短消息发送函数，从 CACHE 里取出一条信息发送出去，发送如果失败则将信息重新
    插入链表尾部
    @param SHORTMSGCACHE *pCache——短信息 CACHE 对象指针
    @param SHORTMSGSENDFUNC SendFunc——发送短信息的回调函数
    @return INT——返回 CAPI_FAILED 表示失败；返回 CAPI_SUCCESS 表示成功
*/
INT ShortMsgCache_Send( SHORTMSGCACHE *pCache，
                        SHORTMSGSENDFUNC SendFunc)
{
    SHORTMSG *pMsg；
    pMsg = DoubleList_PopHead(pCache->pList)；
    if ( pMsg == NULL )
    {
        return CAPI_FAILED；
    }
    if ( (*SendFunc)(pMsg) != CAPI_SUCCESS )
    {
        DoubleList_InsertTail(pCache->pList，(void *)pMsg)；
        return CAPI_FAILED；
    }
    else
    {
```

```
            ShortMsgDestroy(pMsg)；
        }
        return CAPI_SUCCESS；
    }
```

本 章 小 结

本章主要介绍了单向链表、双向链表的实现及适合于链表排序的算法；还介绍了一个使用整块内存的链表，这个链表在后面第 9 章的内存管理算法中会用到；最后讲了一个链表在短信息系统 CACHE 管理上的应用。链表是软件编程中应用最广泛的一个数据结构容器，必须领会透它的设计思想和应用场合，特别是要掌握链表和数组的区别。

习题与思考

1. 在整块内存链表的实现中，如果自由空间的节点已用完，欲再插入数据时，先申请一块大一倍的内存，将原来内存中数据直接拷贝过去，将多出的一半自由空间加入到自由空间链表中，此时再插入数据。试问这种实现方式存在什么问题？
2. 编码实现一个整块内存中的链表插入算法，要求实现任意个数节点的插入。

4 哈 希 表

本章主要介绍哈希表、哈希链表；还介绍了一个使用哈希表管理 WebServer 中动态网页文件的实例，将哈希表、哈希链表和前面讲过的数组、链表等容器从多个不同角度进行了比较，其中的哈希链表是本书介绍的第一个复合数据结构。

哈希表在软件中有着较广泛的应用，最常见的应用是电子词典。如果使用二分法查字典，假设词典有 10 万条记录，需要比较至少 17 次才能查到结果，并且词典还要预先排好序。如果词典在运行过程中有新增条目，还要用二分法查找插入位置，插入时效率非常低。使用哈希表则几乎可以一次就查到对应的条目，且添加操作也非常简单。下面就来介绍哈希表的基本概念和实现。

4.1 哈希表

4.1.1 哈希表的基本概念

哈希这个词是从英文单词 hash 音译过来的，hash 是“杂乱信息”的意思，因此哈希表从字面意义理解也就是一堆杂乱信息组成的表。为了形象地表达哈希表，还是以几个简单的整数为例来说明它。

例 4-1　以下是几个整数和它们的含义，试用哈希表的方法进行查找。

23 177：　黄瓜

54 181：　茄子

875 134：　白菜

334 755： 豆荚

35 789： 西红柿

解　可以先构造一个 5 个指针的数组，然后求每个整数除以 5 的余数，再根据余数值放入数组对应下标的位置上，对于有相同余数的，则以链表的形式存放，如图 4-1 所示。

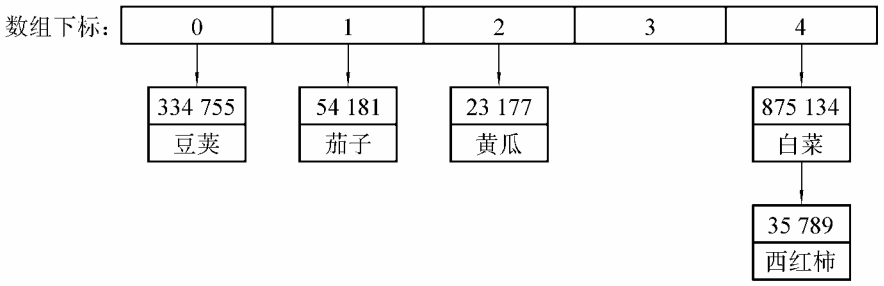


图 4-1　哈希表的示意图

从图 4-1 中可以看出，875 134 和 35 789 都存放在位置 4 上，以链表的形式存放着。在查表时可以根据每个整数除 5 的余数来进行查找，比如查找包含 35 789 的节点，先将 35 789 除以 5，余 4，因此从数组下标为 4 的位置进行查找，得到第 1 个节点；比较第 1 个节点的关键词为 875 134，和 35 789 不一样，继续查找 875 134 节点的下一节点；得到 35 789 所在的节点，至此就查出了对应的解释为西红柿。

从上面过程可以看出，哈希表的查找效率是很高的，一般情况下只要比较一次关键词便可以查到对应数据，如上面的 334 755，54 181，23 177 这几个节点。但是也应该看到，如果同一位置存放多个节点的链表过长，那么查找这些数据时效率也是非常低的，因此使用哈希表时，如何设计一个好的创建索引方法就显得非常重要。下面讨论哈希表的索引方法。

4.1.2　哈希表的索引方法

从上面的分析可以看出，哈希表实际上是将一堆杂乱信息，根据它们的关键词特点，将其映射到一个连续的空间上，本书中将这种映射关系称作索引方法，对应的实现函数叫哈希函数，同时将关键词映射后的值叫做关键词的索引。从例 4-1 可以看出，索引实际上就是数组的下标，数组的每一个位置习惯上也被称作桶(bucket)，在哈希表中桶和索引的叫法是等价的。哈希表中关键词的数据类型可能是任意的。是不是每种数据类型都需要为它设计一个索引方法呢？其实在计算机里，任意类型数据最终都可以转换成整数运算，因此只要搞清楚整数的索引方法，其他类型数据便不难解决。哈希表的索引方法有很多种，下面介绍几种常用的方法。

1. 整除取余法

对于整数来说，整除取余法是一个很常用的方法，例 4-1 中介绍的方法便是整除取余法。整除取余法就是取关键词整除一个给定数的余数作为索引，用 C 语言函数描述如下。

```
/** 哈希表的计算整数的索引函数
    @param UINT uKey——关键词
    @param UINT uBucketCount——哈希表的大小，用来做整除的模
    @return UINT——索引值
*/
UINT HashInt( UINT uKey，UINT uBucketCount )
{
    return uKey % uBucketCount；
}
```

从上面的代码可以看出，整除取余法操作非常简单，只需经过一次取余运算，效率非常高，可以说是最快的索引方法。整除取余法虽然计算关键词的索引很快，但如果 nMod 选取不好的话，可能使很多关键词的索引相同，这样查找起来也会很慢，最坏的情况是所有关键词的索引都相同，这时就退化成顺序查找了。

如果关键词本身就是一个连续的序列或者接近一个连续系列，则整除取余法的效果就非常好。因为连续序列经过整除取余后，得到的索引是不会重复的，索引不会有冲突，那样的话，每次查找都是一次到位。

虽然整除取余法的最坏情况很糟糕，但在实际应用中，不管关键词是静态的集合还是动态的集合，都可以预先使用关键词可能的最大集合计算一下，求出一个使关键词索引尽量不相同的最小 nMod 来。当然 nMod 不能太大，因为最后得出的余数可能为 nMod－1，所以表的大小必须大于 nMod－1，当 nMod 太大时意味着需要表的辅助空间太大。

整除取余法还有一个限制条件是只能对计算机硬件能够处理的整数进行计算，如 32 位 CPU 硬件最大能处理的整数是 2^{32}＝4 294 967 296。在实际中，要处理的关键词可能远远大于这个范围，因此还要用其他方法来将关键词转换成计算机能处理的整数才可以利用整除取余法。

2. 折叠法

当关键词位数很多时，可以将关键词分割成位数相同的几部分，每部分可以转换成一个计算机可以处理的整数，然后将各部分转换后的整数相加，得到一个新的整数，这就是折叠法。折叠法最后得到的那个新的整数不能直接作为关键词的索引，还需要

使用前面介绍的整除取余法来得到关键词的索引。

比如说字符串这种数据类型，便可以用折叠法来进行计算，下面给出一个对字符串用折叠法和整除取余法的 C 语言函数。

```
/** 字符串类型数据的关键词索引计算函数，将字符串折叠成每 5 个字符一段，每段看成
    八进制整数，将各段相加后再取余数
    @param void *pStr——字符串指针
    @param UINT uBucketCount——最大 bucket 的个数，被用来做为整除的模
    @return UINT——字符串关键词的索引
*/
UINT HashString( void *pStr，UINT uBucketCount )
{
    unsigned char *psz；
    UINT uHashValue；
    UINT uRet；
    UINT i；
    psz = (unsigned char *)pStr；
    uHashValue = 0；
    i = 0；
    uRet = 0；
    while ( *psz != '\0' )
    {
        if ( i == 5 )
        {
            i = 0；
            uRet += uHashValue；
            uHashValue = 0；
        }
        uHashValue += uHashValue << 3；
        uHashValue += (UINT)( *psz )；
        psz++；
        i++；
    }
    uRet += uHashValue；
    return uRet % uBucketCount；
}
```

采用折叠法的计算开销比整除取余法大很多，因为它要将关键词中的每个字节都计算一遍，但它的好处是计算出的索引很少会重复。在数据较多的情况下，使用这种

方法效果是很好的。用它计算一个 10 万条词汇的字典，发现相同索引的关键词的最大个数不超过 8 个，只有几个索引有 8 个关键词，一半以上的索引是没有冲突的，超过 3 个关键词有同一索引的所占的比例非常小。同一索引的平均关键词个数大约 1.5 左右，也就是说大部分情况下查找一个关键词只要一次比较，平均只要经过 1.5 次比较，极少数的最坏情况下 8 次比较就可以查到。

3. 平方取中法

顾名思义，平方取中法是将关键词进行平方运算后，再取运算结果的中间几位作为索引。通常一个数经平方运算后，其结果的中间几位和数的每一位都相关，具体取多少位则需要由表的长度来决定。

4. 随机函数法

随机函数法是设计一个产生随机数的函数，以关键词作为随机函数的输入，函数的计算结果作为索引。采用随机函数法，在产生随机数时一般也要用到整除取余，实际上是整除取余法的一个扩展。

不管采用何种方法计算索引，都会对以下几个方面产生影响。

① 计算索引所需要的时间(建议最大不要超过 12 次比较的时间)；

② 每次查找关键词的平均比较次数(建议最大不要超过 5 次)；

③ 最坏情况下的比较次数(建议最大不要超过 50 次)；

④ 哈希表所需要的辅助空间 (建议不要超过实际关键词个数的两倍大小)。

只有综合兼顾以上四种情况的计算索引方法才是好方法。

4.1.3 哈希表的冲突解决方法

在例 4-1 中已经看到，两个不同关键词得到的索引可能是一样的，我们把有相同索引的不同关键词称为冲突关键词。如何解决冲突关键词的存储？前面已经讨论了用单向链表来解决冲突的方法。解决冲突的方法有很多种，每种都有它的优缺点，不管采用何种方法解决冲突，在查找时，同一索引的冲突关键词愈多，则查找的比较次数愈多。下面介绍解决冲突的两种常用方法。

1. 链表存储法

链表储存法就是将同一索引下的关键词放在一个链表中，将哈希表的索引指向链表的表头，实际中一般用单向链表来实现。单向链表插入数据在尾部时要将整个链表遍历一遍，效率很低，因此每次插入数据时，要插入在链表的头部。查找时，先用关键词算出索引，在索引指向的链表中进行顺序查找。

链表存储法的优点是，发生冲突时，只需简单地将数据插入到链表中，只要计算

一次索引，速度较快，并且算法稳定；缺点是使用链表存储数据，每个节点需要消耗附加的链表后向指针的空间，辅助空间增加了。由于速度快，算法简单，稳定性好，实际应用中一般都是采用链表存储法。

链表储存法哈希表的 C 语言描述如下。

```
typedef struct HASHTABLE {
    SINGLENODE **ppBucket；    /* 索引表指针 */
    UINT uBucketCount；        /* 索引表的大小 */
    UINT uNodeCount；          /* 表中的实际节点个数 */
    UINT uCurBucketNo；        /* 当前要执行的 bucket 序号 */
    SINGLENODE *pCurEntry；    /* 当前 bucket 中的下一个要执行的节点条目 */
} HASHTABLE；
```

2. 索引探测法

索引探测法的基本思想是发现索引有冲突后，在索引位置向后查找一个空的索引位置，将数据存放在这个空的索引位置上；查找时，根据插入时查找空的索引位置的方法进行查找；一般探测空的索引位置有线性探测法、二次探测法以及伪随机探测法。索引探测法可以用公式描述如下：

$$H(i) = (Hash(key) + P(i)) \% uBucketCount；$$

$$i = 1，2，…，k\ (k \leq uBucketCount - 1)$$

其中：H(i)是第 i 次探测到的地址；Hash(key) 是前面讲的索引计算函数；P(i)是用来探测的函数，当 P(i) = i 时就是线性探测法，当 $P(i) = i^2$ 时就是二次探测法，当 P(i) = 伪随机数序列时，就是伪随机探测法。

线性探测法的基本方法是：当发现索引处已经存储了数据时，从索引位置向后按顺序查找一个空的索引位置，将数据存储在这个空的位置上。这种方法的优点是，当索引空间足够大时，总能找到一个空的位置去放置数据。在存储上，每个节点不需要链表存储法中的后向节点指针，除了索引需要消耗辅助空间外，不需要其他辅助空间，因此这种方法在空间效率上较好；但是在查找上，当计算出的索引处的关键词和查找的关键词不一样时，需要向后按顺序查找，最坏的可能是遍历整个表，因此这种方法是一种不稳定的方法。

二次探测法、伪随机探测法与线性探测法类似，即空间效率好，时间效率低。因此索引探测法一般只能用在数据较少，内存空间受限的系统中，现在已经很少再用这种方法去构建哈希表了。

索引探测法也可以和链表存储法结合起来使用，这种情况下，索引中要包含两个指针，一个是关键词及数据指针，一个是下一个相同索引的关键词数据指针；当发现冲突时，探测到新的索引位置后，将原来索引中的一个地址指针指向新的索引位置，

这样就可以根据地址指针很快进行查找。

4.1.4 哈希表基本操作的源代码

哈希表的主要操作包含创建、释放、插入、删除、查找、遍历等，本节用链表存储法来实现一个哈希表，下面给出哈希表的基本操作的编码实现。

```
/** 哈希表的创建函数
    @param UINT uBucketCount——索引的大小
    @return HASHTABLE *——成功返回哈希表的指针；失败返回 NULL
*/
HASHTABLE *HashTable_Create(UINT uBucketCount)
{
    HASHTABLE *pTable；
    if ( uBucketCount == 0 )
    {
        return NULL；
    }
    pTable = (HASHTABLE *)malloc( sizeof(HASHTABLE) )；
    if ( pTable == NULL )
    {
        return NULL；
    }
    pTable->uNodeCount = 0；
    pTable->uBucketCount = uBucketCount；
    pTable->ppBucket = (SINGLENODE **)malloc( uBucketCount
        * sizeof(SINGLENODE *))；
    if (pTable->ppBucket == NULL)
    {
        free( pTable )；
        return NULL；
    }
    memset(pTable->ppBucket，0，uBucketCount *sizeof(SINGLENODE *))；
    pTable->pCurEntry = NULL；
    pTable->uCurBucketNo = 0；
    return pTable；
}

/** 哈希表的释放函数
    @param HASHTABLE *pTable——哈希表指针
```

```
    @param DESTROYFUNC DestroyFunc——数据释放函数，为 NULL 时只释放节点辅助
                                    空间，不释放数据
    @return void——无
*/
void HashTable_Destroy(HASHTABLE *pTable，DESTROYFUNC DestroyFunc)
{
    SINGLENODE **ppBucket；
    SINGLENODE *pNode；
    SINGLENODE *pFreeNode；
    UINT i；
    if ( pTable == NULL )
    {
        return；
    }
    ppBucket = pTable->ppBucket；
    for ( i = 0； i < pTable->uBucketCount； i++ )
    {
        pNode = ppBucket[i]；
        while ( pNode != NULL )
        {
            if ( DestroyFunc != NULL )
            {
                (*DestroyFunc)(pNode->pData)；
            }
            pFreeNode = pNode；
            pNode = pNode->pNext；
            free(pFreeNode)；
        }
    }
    free(ppBucket)；
/* 将 ppBucket 设成空指针以避免哈希表被重新使用时造成内存泄漏 */
    pTable->ppBucket = NULL；
    free( pTable )；
}

/** 哈希表的插入函数
    @param HASHTABLE *pTable——哈希表指针
    @param void *pData——数据指针，其中包含关键词信息
    @param ASHFUNC HashFunc——哈希函数，用来计算关键词的索引
```

```
    @return INT (by default) ——返回 CAPI_SUCCESS 表示成功；返回 CAPI_FAILED 表示失败
*/
INT HashTable_Insert( HASHTABLE *pTable，void *pData，HASHFUNC HashFunc )
{
    UINT uIndex；
    SINGLENODE *pNode；
    SINGLENODE *pNewNode；
    if ( pTable == NULL || pData == NULL || HashFunc == NULL )
    {
        return CAPI_FAILED；
    }
    uIndex = (*HashFunc)( pData，pTable->uBucketCount )；
    pNode = (pTable->ppBucket)[uIndex]；
    pNewNode = (SINGLENODE *)malloc( sizeof(SINGLENODE) )；
    if ( pNewNode == NULL )
    {
return CAPI_FAILED；
    }
    /* 将新节点插入到链表的头部 */
    pNewNode->pData = pData；
    pNewNode->pNext = pNode；
    (pTable->ppBucket)[uIndex] = pNewNode；
    pTable->uNodeCount += 1；
    return CAPI_SUCCESS；
}

/** 哈希表的查找函数
    @param HASHTABLE *pTable——哈希表指针
    @param void *pData——包含关键词信息的数据指针
    @param HASHFUNC HashFunc——哈希表的计算索引函数
    @param COMPAREFUNC CompareFunc——关键词比较函数
    @return void *——返回查到的数据指针；未查到返回 NULL
*/
void *HashTable_Find(HASHTABLE *pTable，void *pData，HASHFUNC HashFunc，
                     COMPAREFUNC CompareFunc)
{
    UINT uIndex；
    SINGLENODE * pNode；
    if ( pTable == NULL || pData == NULL
```

```
        || HashFunc == NULL || CompareFunc == NULL )
    {
        return NULL；
    }
    uIndex = (*HashFunc)( pData，pTable->uBucketCount )；
    pNode = (pTable->ppBucket)[uIndex]；
    /* 在 HASHTABLE 中进行查找 */
    while ( pNode != NULL )
    {
        if ( (*CompareFunc)( pNode->pData，pData) == 0 )
        {
            /* 已经找到了关键词，返回 */
            return pNode->pData；
        }
        pNode = pNode->pNext；
    }
        return NULL；
    }

/** 哈希表的删除函数
    @param HASHTABLE *pTable——哈希表指针
    @param void *pData——包含关键词信息的数据指针
    @param HASHFUNC HashFunc——哈希函数，用来计算关键词的索引
    @param COMPAREFUNC CompareFunc——关键词比较函数
    @param DESTROYFUNC DataDestroyFunc——数据释放函数
    @return INT——返回 CAPI_FAILED 表示失败；返回 CAPI_SUCCESS 表示成功
*/
INT HashTable_Delete( HASHTABLE *pTable，void *pData，HASHFUNC HashFunc，
                      COMPAREFUNC CompareFunc，
                      DESTROYFUNC DataDestroyFunc )
{
    UINT uIndex；
    SINGLENODE * pNode；
    SINGLENODE *pPrevNode；
    if ( pTable == NULL || pData == NULL || HashFunc == NULL
        || CompareFunc == NULL )
    {
        return CAPI_FAILED；
```

```
    }
    uIndex = (*HashFunc)(pData，pTable->uBucketCount )；
    pNode = (pTable->ppBucket)[uIndex]；
    pPrevNode = pNode；
    /* 从哈希表中查找 */
    while ( pNode != NULL )
    {
        if ( (*CompareFunc)( pNode->pData，pData ) == 0 )
        {
            if ( pPrevNode == pNode )
            {
                pTable->ppBucket[uIndex] = pNode->pNext；
            }
            else
            {
                pPrevNode->pNext = pNode->pNext；
            }
            /* 删除对应节点 */
            if ( DataDestroyFunc != NULL )
            {
                (*DataDestroyFunc)(pNode->pData)；
            }
            free(pNode)；
            pTable->uNodeCount－= 1；
            return 1；
        }
        pPrevNode = pNode；
        pNode = pNode->pNext；
    }
    return 0；
}

/** 哈希表的逐个节点遍历初始化函数
    @param HASHTABLE *pTable——哈希表指针
    @return void——无
*/
void HashTable_EnumBegin(HASHTABLE *pTable)
{
    pTable->uCurBucketNo = 0；
```

```
    pTable->pCurEntry = pTable->ppBucket[0];
}

/** 哈希表的逐个节点遍历函数
    @param HASHTABLE *pTable——哈希表指针
    @return void *——遍历到的节点的数据指针
*/
void *HashTable_EnumNext(HASHTABLE *pTable)
{
    void *pData;
    while ( pTable->pCurEntry == NULL )
    {
        pTable->uCurBucketNo += 1;
        if ( pTable->uCurBucketNo >= pTable->uBucketCount )
        {
            return NULL;
        }
        pTable->pCurEntry = pTable->ppBucket[pTable->uCurBucketNo];
    }
    pData = pTable->pCurEntry->pData;
    pTable->pCurEntry = pTable->pCurEntry->pNext;
    return pData;
}
```

4.2 哈希链表

4.2.1 哈希表和数组、链表的效率比较

前面已经介绍了哈希表、数组、链表，现从以下几个方面对这几个数据结构与算法中的基本容器将进行比较。

① 插入和删除操作的时间效率；
② 精确查找的时间效率；
③ 模糊查找的支持(模糊查找查找刚好大于或小于某个数据的节点数据)；
④ 空间效率；
⑤ 数据个数变化幅度；
⑥ 有序输出(也就是按大小顺序输出)的支持。

比较结果如表 4-1 所示。

表 4-1　哈希表和数组、链表的效率比较

项目	插入删除	精确查找	模糊查找	空间效率	数据个数变化幅度	有序输出
数组	插入在尾部时效率高；在中间位置时需要移动后面数据，效率很低	排好序后可以使用二分查找之类的查找算法，查找效率高	支持，用二分查找法效率很高	内存空间是连续的，节点只有数据，不需要额外辅助空间	数据个数变化幅度大时，需要频繁重新分配内存，支持性不好	支持
单向链表	插入在头部时效率高；在尾部或中间位置时，需要从头开始查找，效率低	必须按顺序查找，查找效率低	支持，但效率很低	每个节点需要额外的后向指针辅助空间	对数据个数变化幅度大的情况支持很好	支持
双向链表	不管插入在哪个位置，效率均很高	必须按顺序查找，查找效率低	支持，但效率很低	辅助空间为单向链表的两倍	对数据个数变化幅度大的情况支持很好	支持
哈希表	插入效率高	不需要排序就可以进行快速查找，查找效率非常高	不支持	需要一个索引表的辅助空间，另外每个节点和单向链表一样需要后向指针辅助空间，辅助空间和双向链表差不多	需要预先知道最大可能的数据个数来决定索引表的大小，对数据个数变化大的情况支持比数组好，比链表差	不支持

从表 4-1 可以看出，数组、单向链表、双向链表、哈希表每种容器都有它自身的优缺点。数组的缺点是在插入、删除、数据个数变化幅度方面较差；单向链表的缺点是插入、删除尾部或中间节点时效率低，查找效率低；双向链表的缺点是查找效率低，辅助空间比单向链表多一倍；哈希表的缺点是不支持有序输出，辅助空间比单向链表多，且不支持模糊查找。

4.2.2　时间效率和空间效率的关系

通常，在实际中很难做到各方面的性能都很好。在设计时，经常要根据实际情况来权衡时间效率和空间效率。在早期，由于内存很小，不得不牺牲时间效率来换取空

间效率，现在则由于硬件按摩尔定律进行指数级的快速发展，内存越来越大，越来越便宜，因此现在的程序一般都是采用以空间换时间的方法。可能有人会问，既然内存越来越大，但 CPU 的速度也是按摩尔定律进行增长的，速度也非常快，为什么要以空间换时间而不是继续以时间换空间呢？

注意，前面说的是一般情况下以空间换时间，并没有说不能以时间换空间，到底如何选择取决于实际情况。对于那些远远没有达到硬件限制的应用来说，不论以时间换空间还是以空间换时间都无所谓。对于内存受限的系统，当然必须以时间换空间。但在实际情况下，内存受限的系统越来越少了，因为内存便宜，完全可以用少量的资金将内存扩大。另外还应看到的是，一般的算法都是只需要花上一倍的空间就可以换来好几倍的时间效率，反之则一般需要牺牲几倍的时间效率来换取一倍的空间效率。对于有很多客户端访问的服务器软件来说，时间效率通常都非常重要，一般都需要以空间来换取时间。

当然，以空间换时间不能盲目使用，当需要牺牲好几倍的空间来换取一倍的时间时，则需要慎重考虑是否达到或接近系统的限制。在操作系统中，当内存消耗过大时，整个系统的效率会下降，反过来会降低时间效率，这也是一个需要认真考虑的因素。

4.2.3 哈希链表的基本概念

从 4.2.1 节的分析知道，每种容器都有它不同的优点和缺点，像链表可以实现有序输出，而哈希表不能实现有序输出但可以快速精确查找，能不能设计一个数据结构将两者的优点结合起来呢？

要实现一个数据结构既要有哈希表的优点又有链表的优点，那么它必然既需要类似哈希表的结构还需要有链表的结构，这就启发我们将哈希表和链表合并成一个数据结构，这就是本章要介绍的数据结构——哈希链表。哈希链表是本书介绍的第一个复合数据结构，下一章里还会介绍另外一个复合数据结构——哈希红黑树。

要将哈希表和链表复合起来，首先要将哈希表的节点和链表的节点复合起来，在这里用哈希表的节点和双向链表的节点复合起来，得到如下一个哈希链表的节点结构。

```
typedef struct HASHLISTNODE_st {
    struct HASHLISTNODE_st *pListNext；     /* 链表的后向节点指针 */
    struct HASHLISTNODE_st *pListPrev；     /* 链表的前向节点指针 */
    struct HASHLISTNODE_st *pBucketNext； /* 哈希表的链接指针 */
    void *pData；                           /* 数据指针 */
} HASHLISTNODE；
```

从上面的结构体可以看出，这个节点结构既包含了双向链表的前后向指针，又包

含了哈希表的节点数据结构中的内容。有了节点的数据结构后，再来设计哈希链表的数据结构，实际上也就是将哈希表和链表的数据结构复合起来，结构体如下所示。

```
typedef struct HASHLIST_st {
    HASHLISTNODE **ppBuckets；   /* 哈希表的索引表 */
    UINT uBucketCount；          /* 索引表的大小 */
    HASHLISTNODE *pHead；        /* 链表的头指针 */
    HASHLISTNODE *pTail；        /* 链表的尾指针 */
    UINT uNodeCount；            /* 哈希链表中的节点个数 */
} HASHLIST；
```

可以看出上述结构体中既包含了哈希表的信息，又包含了链表的信息，哈希表和链表重复的条目被合并，将哈希表和链表的信息有机地组合起来了。

既然哈希链表数据结构包含哈希表和链表的数据结构，那么它的操作自然也要包含哈希表和链表的操作。首先，它的插入删除操作里既要对哈希表进行插入删除操作，还要对链表进行插入删除操作；它的查找操作则比较简单，直接使用哈希表的查找即可；还可以进行排序，排序是对链表进行的，不会影响到哈希表。

4.2.4 哈希链表的操作

1. 哈希链表的插入操作

例 4-2　下面以三个数据为例来演示数据插入到哈希链表中的过程。

解　步骤 1：在哈希链表中插入 23 177(黄瓜)，如图 4-2 所示，哈希链表中的第 2 个 bucket 指向黄瓜节点，黄瓜节点的链表前、后向指针指向 NULL，链表的 pHead 和 pTail 指针都指向黄瓜。

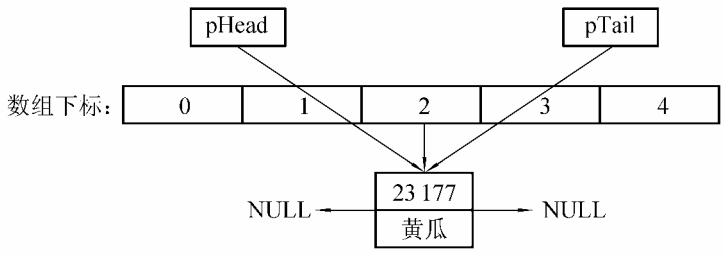


图 4-2　哈希链表插入过程示意图 1

步骤 2：在哈希链表中插入 54 181(茄子)，先将茄子节点插入在哈希链表的第 1 个 bucket 中，然后将黄瓜节点的链表前向指针指向茄子，茄子的链表后向指针指向黄瓜，茄子前向指针和黄瓜后向指针指向 NULL，pHead 指向茄子，pTail 指向黄瓜，如图 4-3

所示。

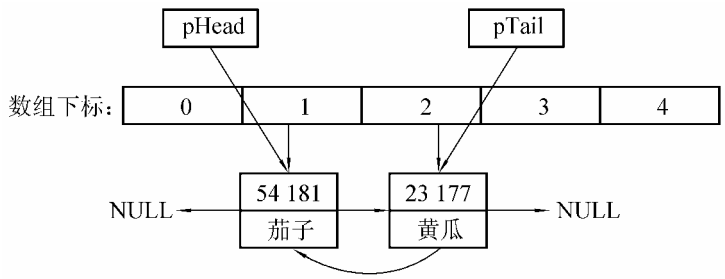


图 4-3　哈希链表插入过程示意图 2

步骤 3：在哈希链表中插入 334 745(豆荚)，首先将豆荚插入到哈希链表的第 0 个 bucket 中，然后将茄子的前向指针指向豆荚，豆荚的后向指针指向茄子，豆荚的前向指针指向 NULL，最后将 pHead 指针指向豆荚，如图 4-4 所示。

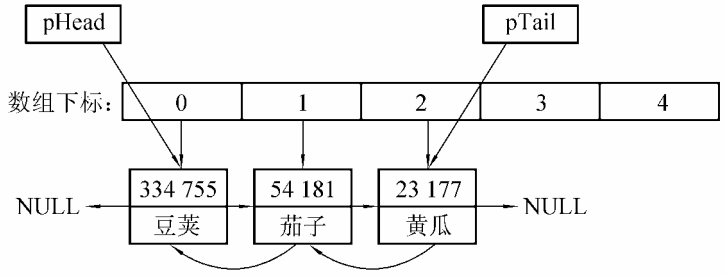


图 4-4　哈希链表插入过程示意图 3

2. 哈希链表的删除操作

哈希链表的删除操作和插入操作刚好相反，假如要在图 4-4 中删除茄子节点，只要将前面豆荚节点的后向指针指向茄子节点的后一节点黄瓜，将黄瓜的前向指针指向豆荚即可将茄子节点删除掉，如图 4-5 所示。

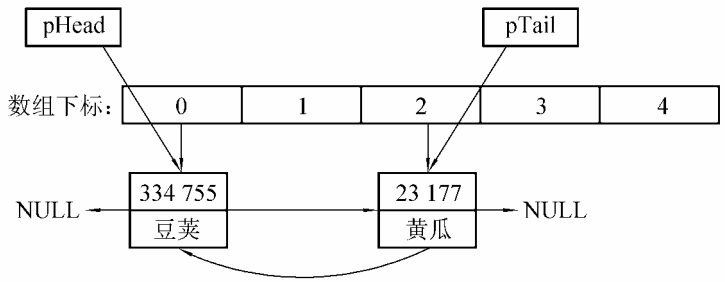


图 4-5　哈希链表的删除节点示意图

4.2.5　哈希链表的编码实现

以下是哈希链表的源代码，这里只给出了创建、释放、插入、删除、查找、插入排序的代码，其他如基数排序、归并排序、逐个遍历等的代码请读者照前面链表的代码自行补充。

```
/** 哈希链表的创建函数
    @param UINT uBucketCount——索引表的大小
    @return HASHLIST *——成功返回哈希表的指针；失败返回 NULL
*/
HASHLIST *HashList_Create(UINT uBucketCount)
{
    HASHLIST *pHashList；
    /* 假设 uBucketCount 最小值不能低于 MINIUM_BUCKET_COUNT */
    if ( uBucketCount < MINIUM_BUCKET_COUNT )
    {
        uBucketCount = MINIUM_BUCKET_COUNT；
    }
    pHashList = (HASHLIST *)malloc(sizeof(HASHLIST))；
    if ( pHashList != NULL)
    {
        /* 创建哈希链表里哈希表的索引表并初始化哈希表 */
        pHashList->ppBuckets = (HASHLISTNODE **)malloc(uBucketCount
            * sizeof(HASHLISTNODE *))；
        if ( pHashList->ppBuckets == NULL )
        {
            free(pHashList)；
            return NULL；
        }
        memset((void *)pHashList->ppBuckets，0，
            uBucketCount *sizeof(HASHLISTNODE *))；
        /* 初始化哈希表里面的双向链表 */
        pHashList->pHead = NULL；
        pHashList->pTail = NULL；
        pHashList->uBucketCount = uBucketCount；
        pHashList->uNodeCount = 0；
    }
```

```
    return pHashList；
}

/** 哈希链表的释放函数
    @param HASHLIST *pHashList——哈希链表指针
    @param DESTROYFUNC DestroyFunc——数据释放回调函数
    @return void——无
*/
void HashList_Destroy(HASHLIST *pHashList，DESTROYFUNC DestroyFunc )
{
    UINT uIndex；
    if ( pHashList == NULL )
    {
        return；
    }
    for ( uIndex = 0； uIndex < pHashList->uBucketCount； uIndex++ )
    {
        HASHLISTNODE    *pNode；
        pNode = pHashList->ppBuckets[uIndex]；
        while ( pNode != NULL )
        {
            HASHLISTNODE *pNodeToFree；
            pNodeToFree = pNode；
            pNode = pNode->pBucketNext；
            if ( DestroyFunc != NULL && pNodeToFree->pData != NULL )
            {
                (*DestroyFunc)(pNodeToFree->pData)；
            }
            free(pNodeToFree)；
        }
    }
    free(pHashList->ppBuckets)；
    free(pHashList)；
    return；
}

/** 哈希链表的数据插入函数，同时插入到哈希表和链表中，插入链表时是插入在链表的
    头部
    @param HASHLIST *pHashList——哈希链表指针
```

```
    @param void *pData——要插入的数据指针
    @param HASHFUNC HashFunc——哈希函数
    @return INT——返回 CAPI_FAILED 表示失败；返回 CAPI_SUCCESS 表示成功
*/
INT HashList_InsertHead(HASHLIST *pHashList，void *pData，HASHFUNC HashFunc)
{
    HASHLISTNODE    *pNode；
    UINT    uBucketIndex；
    if ( pHashList == NULL || pData == NULL )
    {
        return CAPI_FAILED；
    }
    /* 生成哈希链表的节点 */
    pNode = (HASHLISTNODE *)malloc(sizeof(HASHLISTNODE))；
    if ( pNode == NULL )
    {
        return CAPI_FAILED；
    }
    pNode->pData = pData；
    pNode->pBucketNext = NULL；
    pNode->pListPrev = NULL；
    pNode->pListNext = pHashList->pHead；
    /* 插入到哈希表中 */
    uBucketIndex = (*HashFunc)(pData，pHashList->uBucketCount)；
    if ( pHashList->ppBuckets[uBucketIndex] == NULL )
    {
        pHashList->ppBuckets[uBucketIndex] = pNode；
    }
    else
    {
        HASHLISTNODE    *pTempNode；
        pTempNode = pHashList->ppBuckets[uBucketIndex]；
        while ( pTempNode->pBucketNext != NULL )
        {
            pTempNode = pTempNode->pBucketNext；
        }
        pTempNode->pBucketNext = pNode；
    }
    /* 插入到链表中 */
```

```
    if ( pHashList->pHead == NULL )
    {
        pHashList->pHead = pNode；
        pHashList->pTail = pNode；
    }
    else
    {
        pNode->pListNext = pHashList->pHead；
        pHashList->pHead->pListPrev = pNode；
        pHashList->pHead = pNode；
    }
    pHashList->uNodeCount += 1；
    return CAPI_SUCCESS；
}

/** 哈希链表的单个节点删除函数，要同时从哈希表和链表中删除
    @param HASHLIST *pHashList——哈希链表指针
    @param void *pData——数据指针
    @param HASHFUNC HashFunc——哈希函数
    @param COMPAREFUNC CompareFunc——数据比较函数
    @param DESTROYFUNC DestroyFunc——数据释放函数
    @return INT——返回 CAPI_FAILED 表示失败；返回 CAPI_SUCCESS 表示成功
*/
INT HashList_Delete( HASHLIST   *pHashList，void *pData，HASHFUNC HashFunc，
                     COMPAREFUNC   CompareFunc，DESTROYFUNC DestroyFunc)
{
    HASHLISTNODE *pNode；
    HASHLISTNODE *pPrevNode；
    UINT uIndex；
    if ( pHashList == NULL || HashFunc == NULL
        || CompareFunc == NULL )
    {
        return CAPI_FAILED；
    }
    uIndex = (*HashFunc)(pData，pHashList->uBucketCount)；
    pNode = pHashList->ppBuckets[uIndex]；
    pPrevNode = NULL；
    while ( pNode != NULL )
    {
```

```
if ( (*CompareFunc)(pNode->pData，pData ) == 0 )
{
    if (pPrevNode == NULL )
    {
        pHashList->ppBuckets[uIndex] = pNode->pBucketNext；
    }
    else
    {
        pPrevNode->pBucketNext = pNode->pBucketNext；
    }
    /* 从链表中删除节点 */
    if ( pNode->pListPrev != NULL )
    {
        pNode->pListPrev->pListNext = pNode->pListNext；
    }
    else
    {
        /* pNode 是链表头指针 */
        pHashList->pHead = pNode->pListNext；
    }
    if ( pNode->pListNext != NULL )
    {
        pNode->pListNext->pListPrev = pNode->pListPrev；
    }
    else
    {
        /* 现在在链表尾部 */
        pHashList->pTail = pNode；
    }
    if ( pNode->pData != NULL && DestroyFunc != NULL )
    {
        (*DestroyFunc)(pNode->pData)；
    }
    free(pNode)；
    pHashList->uNodeCount一= 1；
    return CAPI_SUCCESS；
}
pPrevNode = pNode；
pNode = pNode->pBucketNext；
```

```
    }
    return CAPI_FAILED；
}

/** 哈希链表的查找节点函数
    @param HASHLIST *pHashList——哈希链表指针
    @param void *pData——要查找的数据指针
    @param HASHFUNC HashFunc——哈希函数
    @param COMPAREFUNC CompareFunc——数据比较函数
    @return HASHLISTNODE *——成功返回查找到的哈希链表节点指针；失败返回 NULL
*/
HASHLISTNODE *HashList_FindNode( HASHLIST *pHashList，void *pData，
                                 HASHFUNC HashFunc，
                                 COMPAREFUNC CompareFunc)
{
    HASHLISTNODE *pNode；
    UINT uIndex；
    if ( pHashList == NULL || HashFunc == NULL
        || CompareFunc == NULL )
    {
        return NULL；
    }
    uIndex = (*HashFunc)(pData，pHashList->uBucketCount)；
    pNode = pHashList->ppBuckets[uIndex]；
    /* 试从哈希表中找匹配的节点 */
    while ( pNode != NULL )
    {
        if ( (*CompareFunc)(pNode->pData，pData) == 0 )
        {
            /* 发现匹配的节点，返回节点指针 */
            return pNode；
        }
        pNode = pNode->pBucketNext；
    }
    /* 没有找到的情况下，返回 NULL */
    return NULL；
}

/** 哈希链表的查找数据函数
```

```
    @param HASHLIST *pHashList——哈希链表指针
    @param void *pData——要查找的数据指针
    @param HASHFUNC HashFunc——哈希函数
    @param COMPAREFUNC CompareFunc——数据比较函数
    @return void *——成功返回查找到的数据指针；失败返回 NULL
*/
void *HashList_FindData( HASHLIST *pHashList，void *pData，HASHFUNC HashFunc，
                         COMPAREFUNC CompareFunc)
{
    HASHLISTNODE *pNode；
    UINT uIndex；
    if ( pHashList == NULL || HashFunc == NULL
        || CompareFunc == NULL )
    {
        return NULL；
    }
    uIndex = (*HashFunc)(pData，pHashList->uBucketCount)；
    pNode = pHashList->ppBuckets[uIndex]；
    /* 试从哈希表中找匹配的节点 */
    while ( pNode != NULL )
    {
        if ( (*CompareFunc)(pNode->pData，pData) == 0 )
        {
            /* 发现匹配的节点，返回节点指针 */
            return pNode->pData；
        }
        pNode = pNode->pBucketNext；
    }
    /* 没有找到的情况下，返回 NULL */
    return NULL；
}

/** 哈希链表的插入排序函数，用插入排序算法对哈希链表里的链表进行排序
    @param HASHLIST *pHashList——哈希链表指针
    @param COMPAREFUNC CompareFunc——数据比较函数
    @return INT——返回 CAPI_FAILED 表示失败；返回 CAPI_SUCCESS 表示成功
*/
INT HashList_InsertSort(HASHLIST *pHashList，COMPAREFUNC CompareFunc)
{
```

```
HASHLISTNODE *pNode；
if ( pHashList == NULL || CompareFunc == NULL )
{
    return 0；
}
pNode = pHashList->pHead；
if ( pNode == NULL )
{
    /* 没有节点在链表中，把它当作已经排好了序 */
    return 1；
}
while ( pNode->pListNext != NULL )
{
    if ( (*CompareFunc)( pNode->pListNext->pData，pNode->pData ) < 0 )
    {
        HASHLISTNODE  *pTempNode；
        pTempNode = pNode->pListPrev；
        while ( pTempNode != NULL )
        {
            if ( (*CompareFunc)( pNode->pListNext->pData，
                    pTempNode->pData ) >= 0 )
        {
                    HASHLISTNODE *pCurNode；
                    /* 将节点弹出来 */
                    pCurNode = pNode->pListNext；
                    pNode->pListNext = pNode->pListNext->pListNext；
                    if ( pCurNode->pListNext != NULL )
                    {
                        pCurNode->pListNext->pListPrev = pNode；
                    }
                    /* 将节点插入到对应位置上 */
                    pCurNode->pListNext = pTempNode->pListNext；
                    pCurNode->pListPrev = pTempNode；
                    pTempNode->pListNext->pListPrev = pCurNode；
                    pTempNode->pListNext = pCurNode；
                    break；
            }
            pTempNode = pTempNode->pListPrev；
        }
```

```
                /* 如果所有数据都大于该节点数据，将该节点插入到链表头部 */
                if ( pTempNode == NULL )
                {
                    HASHLISTNODE *pCurNode；
                    /* 将节点弹出来 */
                    pCurNode = pNode->pListNext；
                    pNode->pListNext = pNode->pListNext->pListNext；
                    if ( pCurNode->pListNext != NULL )
                    {
                        pCurNode->pListNext->pListPrev = pNode；
                    }
                    /* 将节点插入链表头部 */
                    pCurNode->pListPrev = NULL；
                    pCurNode->pListNext = pHashList->pHead；
                    pHashList->pHead = pCurNode；
                }
            }
            else
            {
                pNode = pNode->pListNext；
            }
        }
        return 1；
    }
```

需要强调的是，上面这段代码没有复用哈希表或者双向链表的代码，实际上是可以复用的，由于都比较简单，所以这里没有复用，可以减少函数调用的开销，提高一些效率。

4.3 实例：WebServer 的动态 CACHE 文件管理

4.3.1 WebServer 的动态 CACHE 文件管理基本概念

在 WebServer 中，由于许多网页频繁地被访问，如果每次访问都从硬盘去读显然是很低效的，有些操作系统文件系统本身具有 CACHE 功能，文件读过一次之后就被存放在 CACHE 中，第二次读时就不用再从硬盘读取了，但操作系统的文件系统 CACHE 功能是用户无法控制的，通常 CACHE 的大小是一定的，并不是所有读过的网页文件都能被存放到 CACHE 中，因此有必要单独设计一个网页文件 CACHE 管理模

块。

对于 CACHE 文件管理，主要考虑以下三项需求。

① 当收到客户端的 URL 请求时，要能很快从 CACHE 里找到对应的文件，将文件发送给对方；

② 当频繁访问某个不在 CACHE 中的文件时，需要将文件读入到 CACHE 中；

③ 如果当 CACHE 使用过程中占用的空间超过预定大小时，需要将 CACHE 中满足某些条件比如访问次数较小或文件已隔了一定时间没有被访问等的文件删除掉。

4.3.2 CACHE 文件管理功能的设计

根据上面提到的需求，可以使用哈希表来保存 CACHE 文件，每个 CACHE 文件的具体结构用 C 语言描述如下。

```
typedef struct CACHEFILE_st {
    char *pszFileName；        /* CACHE 节点中的文件名 */
    UINT uAccessTimes；        /* 本节点文件被访问的次数 */
    char *pFileData；          /* 文件中的数据 */
    UINT uFileLength；         /* 文件长度 */
    clock_t LastAccessTime；   /* 文件最后一次被访问的时间 */
} CACHEFILE；

typedef struct CACHE_st {
    UINT uTotalSize；          /* CACHE 占用空间最大尺寸 */
    UINT uUsedSize；           /* CACHE 中使用空间的尺寸 */
    UINT uBucketCount；        /* 管理 CACHE 的哈希表的 bucket 数量 */
    clock_t CollageTime；      /* 定期清除访问不频繁文件的时间 */
    UINT uMinAccessTimes；     /* 最小的访问次数，为清除时使用 */
    HASHTABLE *pTable；        /* 保存 CACHE 文件的哈希表 */
} CACHE；
```

要实现对 CACHE 的管理，可以设计以下几个主要操作。

1) 创建操作 **Cache_Create( )**：创建一个 CACHE 文件管理表，通过调用哈希表的创建操作来实现。

2) 释放操作 **Cache_Destroy( )**：负责将 CACHE 文件管理表释放掉，通过调用哈希表的释放操作来实现。

3) 添加操作 **Cache_Add( )**：向 CACHE 文件管理表中添加一个文件，通过调用哈希表的插入操作来实现。

4) 删除操作 **Cache_Delete( )**：从 CACHE 文件管理表中删除一个文件，通过调用

哈希表的删除操作来实现。

5) 查找操作 Cache_Find( )：从 CACHE 文件管理表中查找一个文件，调用哈希表的查找操作来实现。如果 CACHE 中未找到则需要从外存文件中读取，添加到哈希表中。

6) 清除操作 Cache_Clean( )：从 CACHE 管理表中清除满足一定条件的 CACHE 文件，主要清除访问次数较少以及最后一次访问时间和当前时间间隔大于设定时间的所有哈希表中的 CACHE 文件。

此外，还需要设计设置最小访问次数的操作以及设置最后一次访问间隔时间的操作，由于这两个操作较简单，请直接读后面的代码。

4.3.3 CACHE 文件管理功能的编码实现

下面就来编码实现 CACHE 文件管理的功能，通过 CACHE 文件管理功能的实现来掌握哈希表的使用方法。

```
#define     DEFAULT_COLLAGE_TIME          (1 000*30)
#define     DEFAULT_MIN_ACCESS_TIMES      5

/** 创建一个 CACHE 文件管理表
    @param UINT uTotalSize——CACHE 的大小
    @param UINT uCollageTime——CACHE 清理时间间隔
    @param UINT uBucketCount——管理 CACHE 的哈希表的 bucket 大小
    @return CACHE *——成功返回创建的 CACHE 指针；失败返回 NULL
*/
CACHE *Cache_Create( UINT uTotalSize，UINT uCollageTime，UINT uBucketCount，
                     UINT uMinAccessTimes)
{
    CACHE *pCache；
    pCache = (CACHE *)malloc(sizeof(CACHE))；
    if ( pCache != NULL )
    {
        pCache->uBucketCount = uBucketCount；
        pCache->uTotalSize = uTotalSize；
        pCache->uUsedSize = 0；
        pCache->CollageTime = DEFAULT_COLLAGE_TIME；
        pCache->uMinAccessTimes = DEFAULT_MIN_ACCESS_TIMES；
        pCache->pTable = HashTable_Create(uBucketCount)；
        if ( pCache->pTable == NULL )
        {
```

```
                free( pCache )；
                pCache = NULL；
            }
        }
        return pCache；
}

/** 哈希表操作中求哈希值的回调函数，仅供 CACHE 管理模块使用
    @param CACHEFILE *pCacheFile——CACHEFILE 指针
    @param INT uBucketCount——哈希表的 bucket 大小
    @return INT——计算出来的哈希值
*/
INT HashCacheFile(void *p，UINT uBucketCount)
{
    return HashString(((CACHEFILE *)p)->pszFileName，uBucketCount)；
}

/** 哈希表的比较回调函数，将 CACHEFILE 结构和一个文件名进行比较
    @param void *p1——指向一个 CACHEFILE 结构
    @param void *p2——指向一个文件名的字符串
    @return INT——p1 = p2 返回 0；p1<p2 返回－1；p1>p2 返回 1
*/
INT CacheFileCompare(void *p1，void *p2)
{
    CACHEFILE *pCacheFile；
    char *pszFileName；
    pCacheFile = (CACHEFILE *)p1；
    pszFileName = (char *)p2；
    return stricmp(pCacheFile->pszFileName，pszFileName)；
}

/** 哈希表的释放文件回调函数，用来释放一个 CACHEFILE 结构体
    @param void *p——指向要释放的 CACHEFILE 结构体
    @return void——无
*/
void Cache File Destroy(void *p)
{
    CACHEFILE *pCacheFile = (CACHEFILE *)p；
    if ( pCacheFile != NULL )
    {
```

```
        if ( pCacheFile->pszFileName != NULL )
        {
            free(pCacheFile->pszFileName)；
        }
        if ( pCacheFile->pFileData != NULL )
        {
            free(pCacheFile->pFileData)；
        }
    }
}

/** CACHE 管理的添加文件函数
    @param CACHE *pCache——要操作的 CACHE 指针
    @param char *pszFileName——要添加的文件名
    @param char *pFileData——存放在内存中的文件数据指针
    @param UINT uFileLength——文件长度
    @return CACHEFILE *——成功返回一个 CACHEFILE 指针；失败返回 NULL
*/
CACHEFILE *Cache_Add File( CACHE *pCache，char *pszFileName，char *pFileData，
                           UINT uFileLength)
{
    CACHEFILE *pCacheFile；
    pCacheFile = (CACHEFILE *)malloc(sizeof(CACHEFILE))；
    if ( pCacheFile != NULL )
    {
        pCacheFile->pFileData = pFileData；
        pCacheFile->pszFileName = strdup(pszFileName)；
        pCacheFile->uAccessTimes = 0；
        pCacheFile->uFileLength = uFileLength；
        pCache->uUsedSize += uFileLength；
        if ( pCache->uUsedSize > pCache->uTotalSize )
        {
            Cache_Clean(pCache)；
        }
        HashTable_Insert(pCache->pTable，(void *)pCacheFile，HashCacheFile)；
    }
    return pCacheFile；
}

/** 从 CACHE 管理中删除一个文件
```

```
    @param CACHE *pCache——要操作的 CACHE 指针
    @param char *pszFileName——要删除的文件名
    @return void——无
*/
void Cache_Delete(CACHE *pCache，char *pszFileName)
{
    CACHEFILE *pCacheFile；
    pCacheFile = (CACHEFILE *)HashTable_Find(pCache->pTable，
        (void *)pszFileName，HashCacheFile，CacheFileCompare)；
    if ( pCacheFile != NULL )
    {
        pCache->uUsedSize -= pCacheFile->uFileLength；
        HashTable_Delete(pCache->pTable，(void *)pszFileName，
            HashCacheFile，CacheFileCompare，CacheFileDestroy)；
    }
}

/** 从 CACHE 中查找一个文件
    @param CACHE *pCache——要操作的 CACHE 指针
    @param char *pszFileName——要查找的文件名
    @return CACHEFILE *——成功返回一个 CACHEFILE 指针；失败返回 NULL
*/
CACHEFILE *Cache_Find(CACHE *pCache，char *pszFileName)
{
    CACHEFILE *pCacheFile；
    pCacheFile = (CACHEFILE *)HashTable_Find(pCache->pTable，
        (void *)pszFileName，HashCacheFile，CacheFileCompare)；
    if ( pCacheFile == NULL)
    {
        FILE *fp；
        UINT uFileLen；
        char *pFileData；
        pCacheFile = NULL；
        fp = fopen(pszFileName，"rb")；
        if (fp != NULL)
        {
            fseek(fp，0，SEEK_END)；
            uFileLen = ftell(fp)；
            fseek(fp，0，SEEK_SET)；
```

```
            pFileData = (char *)malloc(uFileLen)；
            if ( pFileData != NULL )
            {
                fread(pFileData，uFileLen，sizeof(char)，fp)；
                pCacheFile = Cache_AddFile(pCache，pszFileName，
                    pFileData，uFileLen)；
            }
            fclose(fp)；
        }
        else
        {
            return NULL；
        }
    }
    pCacheFile->uAccessTimes += 1；
    pCacheFile->LastAccessTime = clock()；
    return pCacheFile；
}

/** 将 CACHE 释放
    @param CACHE *pCache——要释放的 CACHE 指针
    @return void——无
*/
void Cache_Destroy(CACHE *pCache)
{
    if ( pCache != NULL )
    {
        HashTable_Destroy(pCache->pTable，CacheFileDestroy)；
        free(pCache)；
    }
}

/** CACHE 的清除函数，清除掉 CACHE 中所有的访问次数低于某个次数或者
    访问间隔时间大于设定时间的文件
    @param CACHE *pCache——要清理的 CACHE 指针
    @return void——无
*/
void Cache_Clean(CACHE *pCache)
{
```

```
    CACHEFILE *pCacheFile；
    clock_t CurTime；
    HashTable_EnumBegin(pCache->pTable)；
    CurTime = clock()；
    while ((pCacheFile = HashTable_EnumNext(pCache->pTable)) != NULL )
    {
        if ( pCacheFile->uAccessTimes < pCache->uMinAccessTimes
            || (pCacheFile->LastAccessTime－CurTime)
                < pCache->CollageTime )
        {
            pCache->uUsedSize－= pCacheFile->uFileLength；

            HashTable_Delete(pCache->pTable，(void *)(pCacheFile->pszFileName)，
                HashCacheFile，CacheFileCompare，CacheFileDestroy)；
        }
    }
}

/** 设置最小访问次数函数
    @param CACHE *pCache——CACHE 指针
    @param UINT uMinAccessTimes——最小访问次数
    @return void——无
*/
void Cache_SetMinAccessTimes(CACHE *pCache，UINT uMinAccessTimes)
{
    if ( pCache != NULL )
    {
        pCache->uMinAccessTimes = uMinAccessTimes；
    }
}

/** 设置 CACHE 清理时间间隔
    @param CACHE *pCache——CACHE 指针
    @param clock_t CollageTime——CACHE 清理时间间隔
    @return void——无
*/
void Cache_SetCollageTime(CACHE *pCache，clock_t CollageTime)
{
    if ( pCache != NULL )
```

```
        {
            pCache->CollageTime = CollageTime；
        }
    }
```

本 章 小 结

本章主要介绍了哈希表的基本概念和实现，在哈希表的基础上还介绍了复合数据结构哈希链表的实现，最后介绍了一个 Web 服务器中使用哈希表来管理网页 CACHE 文件的动态 CACHE 管理实例。

在介绍哈希链表的时候，将数组、链表、哈希表等从多方面进行了比较，分析了这几种数据结构的优缺点和适用范围，进而讨论了时间效率和空间效率的关系。通过本章的学习，读者应该掌握哈希表的使用和设计方法，并掌握如何在时间效率和空间效率之间取得均衡。

习题与思考

1. 编码将 100 万个随机整数插入到哈希表中，再将这 100 万个整数全部查找一遍，统计所花费的时间。
2. 编程比较一下哈希表查找和二分查找算法在 10 个整数，1 000 个整数，10 万个整数和 100 万个整数中将所有的数查找一遍所耗时间的区别。

5 树

本章主要介绍树，包括普通树、二叉树，在普通树中还介绍了一个 XCOPY 算法，二叉树的介绍则侧重于一些应用比较多的红黑树、AVL 树。这些内容均有完整的源代码实现，最后介绍了一个使用二叉树实现的搜索引擎的实例。

5.1 普通树

5.1.1 普通树的描述方法

在实际工作中，我们经常和文件系统打交道。文件系统的目录结构便是一颗树，它有一个根目录，根目录下可以有许多子目录，也可以有许多文件，同样，子目录下也可以有许多更低级的子目录和文件。

抽象地讲，树是这样一种结构：它有一个根，根下面有许多子树(相当于子目录)和叶子(相当于文件)，子树下面又可以有子树和叶子。

比如文件系统，一个子目录下可以有零个或一个文件，也可以有几万个文件；可以有零个或一个子目录，也可以有几万个子目录。由于子树和叶子的数量都是不确定的，并且变化范围很大，因此可以使用链表来描述一个子树下的叶子和子树，下面用 C 语言结构体来描述一颗树。

```
typedef struct TREE_st {
    DOUBLELIST *pLeafList；
    DOUBLELIST *pSubTreeList；
    void *pProperties；
} TREE *LPTREE；
```

普通树的结构如图 5-1 所示。

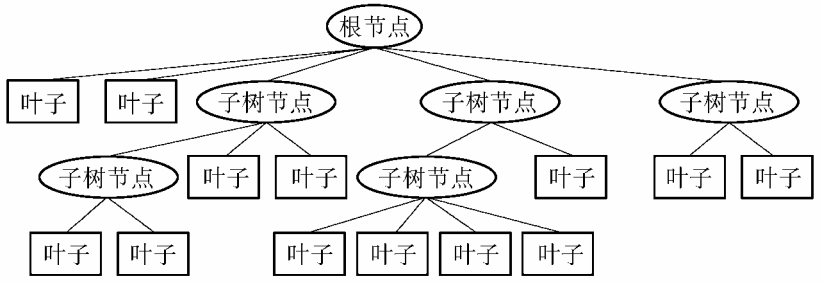


图 5-1　普通树的结构示意图

5.1.2　树的操作接口设计

树的接口操作主要有以下几种。

1) 创建操作 Tree_Create( )：负责创建一颗树。

2) 释放操作 Tree_Destroy( )：负责释放一颗树。

3) 增加叶子节点 Tree_AddLeaf( )：将一个叶子节点加入到树中。

4) 删除叶子节点 Tree_RemoveLead( )：从书中删除一个叶子节点。

5) 增加一个子树 Tree_AddSubTree( )：添加一颗子树到一个存在的树中。

6) 删除一颗子树 Tree_RemoveSubTree( )：从一颗存在的树中删除一颗子树。

7) 树的拷贝 Tree_Copy( )：复制一颗已存在的树。

8) 通过属性查找子树 Tree_FindSubTreeByProp( )：从存在的树中通过子树属性来查找子树。

5.1.3　树的遍历算法

1. 使用递归的遍历算法

对普通树的遍历只有深度优先、宽度优先两种遍历方法，深度优先又可以分为先序和后序两种，但没有中序遍历之说，中序遍历只有二叉树才有。

下面分别给出先序遍历、后序遍历和宽度优先遍历的流程如图 5-2、图 5-3、图 5-4 所示。

2. 使用栈的非递归遍历算法

所有的递归算法都可以通过使用栈操作来变成非递归的算法，本质上讲，递归算法是利用系统本身提供的栈，不过系统本身的栈深度很大时效率非常低，因此可以使用自己的栈来取代系统的栈将递归算法变成非递归算法。

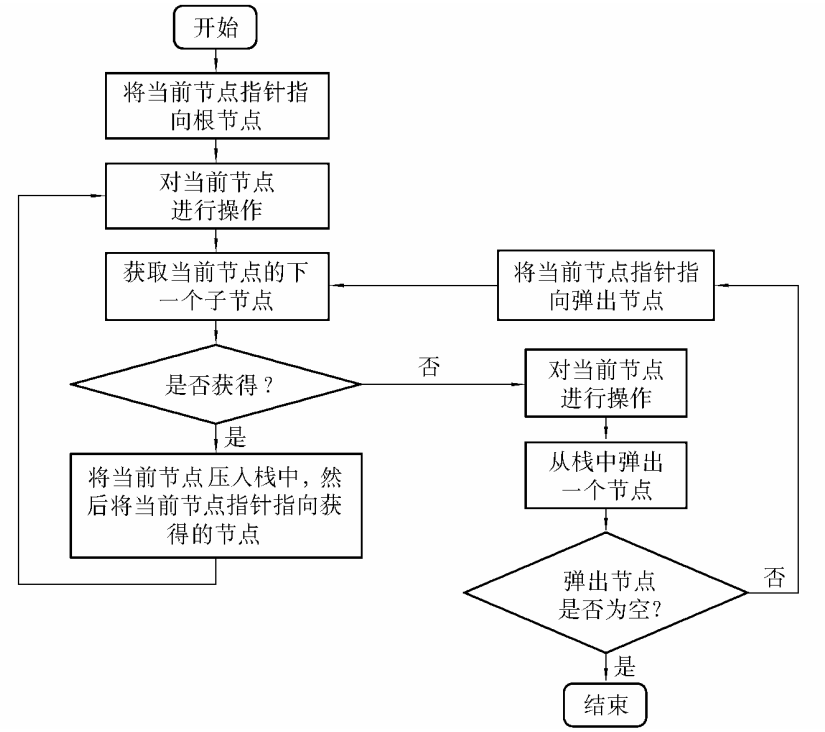


图 5-2　普通树的先序遍历流程图

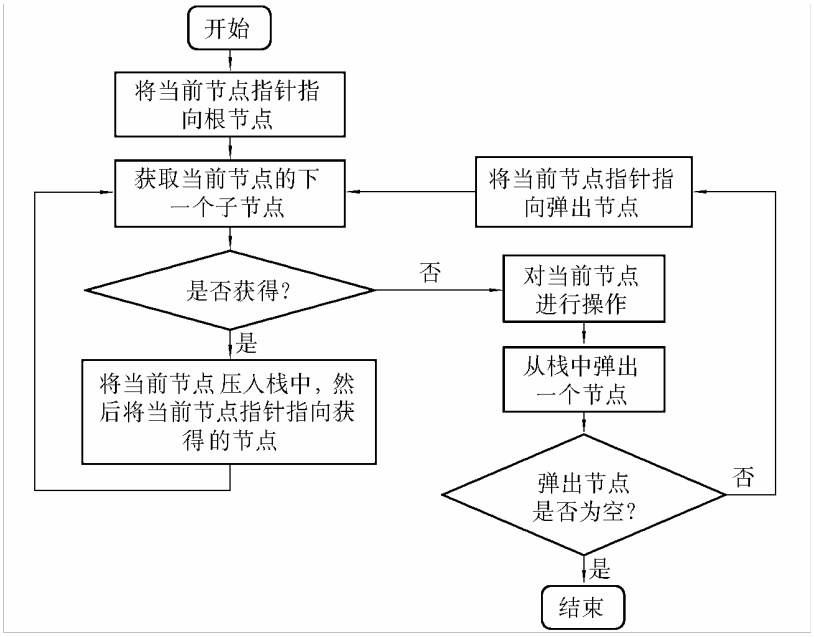


图 5-3　普通树的后序遍历流程图

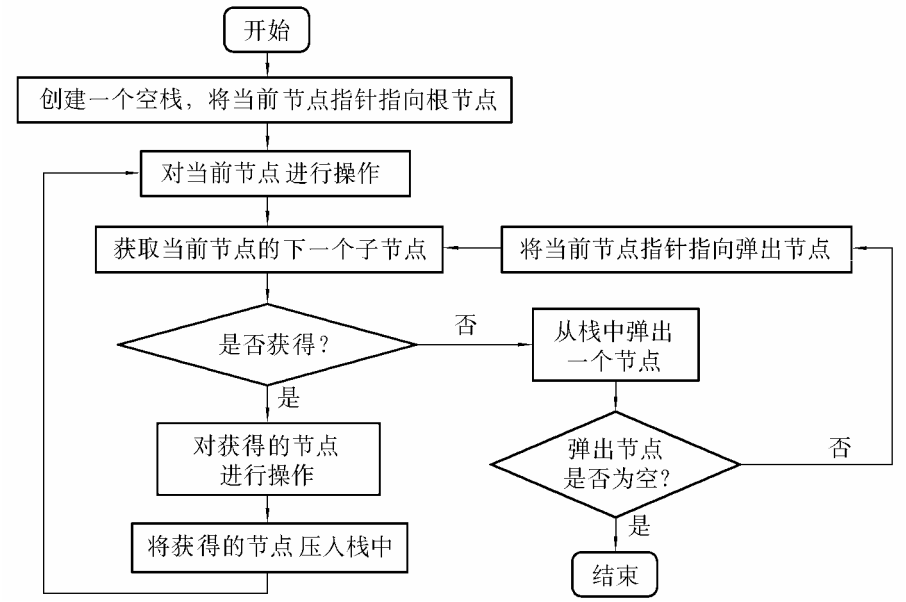


图 5-4　树的宽度方向遍历流程图

3. **不使用栈的非递归遍历算法**

对于普通树来说，也可以设计出不使用栈的非递归遍历算法，不过这要求每个节点都保存它的父节点指针，不使用栈的非递归遍历算法流程如图 5-5 所示。

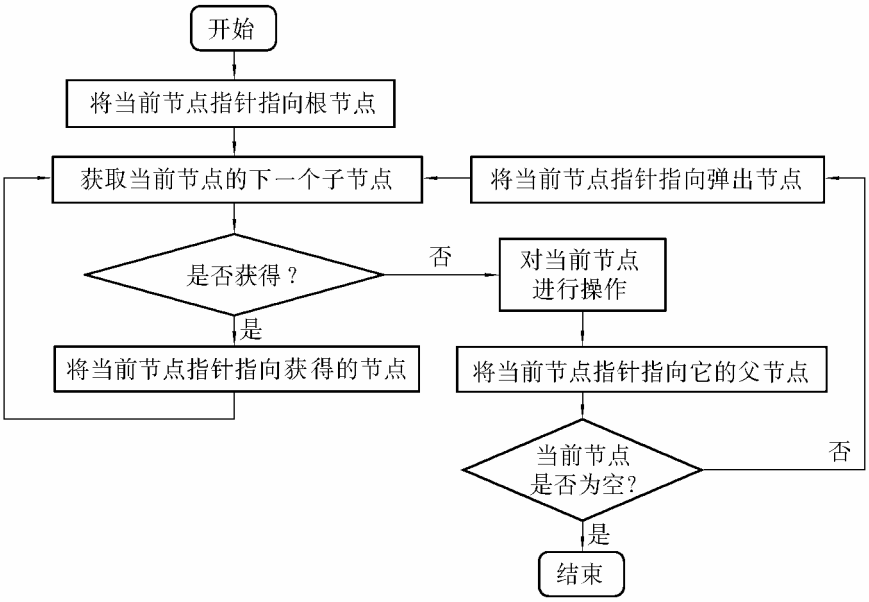


图 5-5　不使用栈的普通树的非递归遍历流程图

5.1.4 树的编码实现

下面给出树的操作的编码实现。

```
/** 树的创建函数
    @return TREE *——新创建的树的指针
*/
TREE *Tree_Create()
{
    TREE *pNewTree；
    pNewTree = (TREE *)malloc(sizeof(struct TREE_st))；
    if ( pNewTree == NULL )
    {
        return NULL；
    }
    /* 创建叶子列表 */
    pNewTree->pLeafList = DoubleList_Create()；
    if ( pNewTree->pLeafList == NULL )
    {
        free( pNewTree )；
        return NULL；
    }
    /* 创建子树列表 */
    pNewTree->pSubTreeList = DoubleList_Create()；
    if ( pNewTree->pSubTreeList == NULL )
    {
        DoubleList_Destroy( pNewTree->pLeafList, NULL )；
        free( pNewTree )；
        return NULL；
    }
    pNewTree->pProperties = NULL；
    return pNewTree；
}

/** 树的释放函数，采用后序遍历算法进行释放
    @param TREE *pTree——要释放的树的指针
    @param DESTROYFUNC LeafDestroyFunc——叶子节点的数据释放函数
    @param DESTROYFUNC PropDestroyFunc——属性释放函数
    @return void——无
```

```
*/
void Tree_Destroy(TREE *pTree, DESTROYFUNC LeafDestroyFunc,
                  DESTROYFUNC PropDestroyFunc )
{
    DOUBLELIST *pList；
    void *pData；
    if ( pTree == NULL || LeafDestroyFunc == NULL
        || PropDestroyFunc == NULL )
    {
        return；
    }
    pList = pTree->pSubTreeList；
    if ( pList == NULL )
    {
        return；
    }
    DoubleList_EnumBegin(pList)；
    /* 逐个遍历子树，递归调用 Tree_Destroy 对子树进行释放 */
    while ( (pData = DoubleList_EnumNext(pList)) != NULL )
    {
        Tree_Destroy((TREE *)pData, LeafDestroyFunc, PropDestroyFunc)；
    }
    /* 释放属性 */
    if ( pTree->pProperties != NULL )
    {
        PropDestroyFunc( pTree->pProperties )；
    }
    /* 释放叶子列表 */
    DoubleList_Destroy( pTree->pLeafList, LeafDestroyFunc )；
    /* 释放树的结构体 */
    free( pTree )；
}

/** 树的增加叶子节点函数
    @param TREE *pTree——要增加叶子的树的指针
    @param void *pLeafData——要增加的叶子的数据
    @return INT——返回 CAPI_FAILE 表示失败；返回 CAPI_SUCCESS 表示成功
*/
INT Tree_AddLeaf(TREE *pTree, void *pLeafData)
```

```
{
    if ( pTree == NULL )
    {
        return CAPI_FAILED；
    }
    return DoubleList_InsertTail( pTree->pLeafList, pLeafData )；
}

/** 树的删除叶子节点函数
    @param TREE *pTree——要删除叶子的树的指针
    @param void *pLeafData——叶子匹配数据的指针
    @param DESTROYFUNC LeafDestroyFunc——叶子数据释放函数
    @return INT——返回 CAPI_FAILE 表示失败；返回 CAPI_SUCCESS 表示成功
*/
INT Tree_RemoveLeaf(TREE *pTree, void *pLeafData,
                    DESTROYFUNC LeafDestroyFunc)
{
    if ( pTree == NULL )
    {
        return CAPI_FAILED；
    }
    return DoubleList_Remove(pTree->pLeafList, pLeafData, LeafDestroyFunc)；
}

/** 树的设置属性函数
    @param TREE *pTree——树指针
    @param void *pProperties——属性指针
    @param DESTROYFUNC PropDestroyFunc——属性释放函数
    @return INT——返回 CAPI_FAILE 表示失败；返回 CAPI_SUCCESS 表示成功
*/
INT Tree_SetProperties( TREE *pTree, void *pProperties,
                        DESTROYFUNC PropDestroyFunc)
{
    if ( pTree == NULL )
    {
        return CAPI_FAILED；
    }
    /* 如果原来已经设置了属性则需要将其释放以防止内存泄漏 */
    if ( pTree->pProperties != NULL )
    {
```

```
            PropDestroyFunc( pTree->pProperties )；
        }
        pTree->pProperties = pProperties；
        return CAPI_SUCCESS；
    }

    /** 树的获取属性函数
        @param TREE *pTree——树指针
        @return void *——成功返回属性指针；返回 NULL 表示失败或属性为 NULL
    */
    void *Tree_GetProperties(TREE *pTree)
    {
        if ( pTree == NULL )
        {
            return NULL；
        }
        return pTree->pProperties；
    }

    /** 树的增加子树函数
        @param TREE *pTree——树指针
        @param TREE *pSubTree——子树指针
        @return INT——返回 CAPI_FAILE 表示失败；返回 CAPI_SUCCESS 表示成功
    */
    INT Tree_AddSubTree(TREE *pTree, TREE *pSubTree)
    {
        if ( pTree == NULL || pSubTree == NULL )
        {
            return CAPI_FAILED；
        }
        return DoubleList_InsertTail( pTree->pSubTreeList, (void *)pSubTree )；
    }

    /** 树的删除子树函数
        @param TREE *pTree——树指针
        @param TREE *pSubTree——子树指针
        @param DESTROYFUNC LeafDestroyFunc——叶子释放函数
        @param DESTROYFUNC PropDestroyFunc——属性释放函数
        @return INT——返回 CAPI_FAILE 表示失败；返回 CAPI_SUCCESS 表示成功
    */
```

```
INT Tree_RemoveSubTree( TREE *pTree, TREE *pSubTree,
                        DESTROYFUNC LeafDestroyFunc,
                        DESTROYFUNC PropDestroyFunc )
{
    DOUBLELIST *pList；
    DOUBLENODE  *pNode；
    if ( pTree == NULL || pSubTree == NULL || LeafDestroyFunc == NULL
        || PropDestroyFunc == NULL)
    {
        return CAPI_FAILED；
    }
    pList = pTree->pSubTreeList；
    if ( pList == NULL )
    {
        return CAPI_FAILED；
    }
    /* 遍历子树链表查找对应的子树进行删除 */
    DoubleList_EnumBegin(pList)；
    while ( (pNode = DoubleList_EnumNode(pList) ) != NULL )
    {
        if ( (TREE *)(pNode->pData) == pSubTree )
        {
            pNode = DoubleList_PopNode(pList, pNode)；
            Tree_Destroy( (TREE *)(pNode->pData), LeafDestroyFunc,
                          PropDestroyFunc )；
            free(pNode)；
            break；
        }
    }
    return CAPI_SUCCESS；
}

/** 树的拷贝函数，采用先序遍历算法进行拷贝
    @param TREE *pTree——树指针
    @param COPYFUNC LeafCopyFunc——叶子拷贝函数
    @param COPYFUNC PropCopyFunc——属性拷贝函数
    @return TREE *——拷贝后的新的树的指针
*/
```

```
TREE *Tree_Copy(  TREE *pTree, COPYFUNC LeafCopyFunc,
                  COPYFUNC PropCopyFunc)
{
    TREE *pNewTree；
    DOUBLELIST *pNewList；
    DOUBLELIST *pList；
    void *pData；
    if ( pTree == NULL || LeafCopyFunc == NULL || PropCopyFunc == NULL )
    {
        return NULL；
    }
    pNewList = DoubleList_Create()；
    if ( pNewList == NULL )
    {
        return NULL；
    }
    pNewTree = Tree_Create()；
    if ( pNewTree == NULL )
    {
        DoubleList_Destroy(pNewList, NULL)；
        return NULL；
    }
    /* 拷贝叶子列表 */
    pNewTree->pLeafList = DoubleList_Copy( pTree->pLeafList, LeafCopyFunc )；
    /* 拷贝属性 */
    pNewTree->pProperties = (*PropCopyFunc)( pTree->pProperties )；
    pList = pTree->pSubTreeList；
    /* 逐个遍历子树列表调用 Tree_Copy 递归拷贝子树列表 */
    DoubleList_EnumBegin(pList)；
    while( (pData = DoubleList_EnumNext(pList)) != NULL )
    {
        TREE *pSubTree = Tree_Copy((TREE*)pData, LeafCopyFunc, PropCopyFunc)；
        DoubleList_InsertTail(pNewList, (void *)pSubTree)；
    }
    return pNewTree；
}

/** 树的按属性查找子树函数
    @param TREE *pTree——树指针
```

```
    @param void *pProperties——要查找的属性指针
    @param COMPAREFUNC CompareFunc——属性比较函数
    @return TREE *——成功返回查找到的子树指针；失败返回 NULL
*/
TREE *Tree_FindSubTreeByProp( TREE *pTree, void *pProperties,
                              COMPAREFUNC CompareFunc )
{
    TREE *pSubTree；
    DOUBLELIST *pList；
    void *pData；
    if (pTree!=NULL)
    {
        return NULL；
    }
    /* 如果已经和树的属性相同，返回当前的树指针 */
    if ( (*CompareFunc)( pProperties, pTree->pProperties )   == 0 )
    {
        return pTree；
    }
    pList = pTree->pSubTreeList；
    DoubleList_EnumBegin(pList)；
    /* 逐个遍历子树并递归调用本函数在子树中进行查找 */
    while ( (pData = DoubleList_EnumNext(pList)) != NULL )
    {
        pSubTree = Tree_FindSubTreeByProp((TREE *)pData, pProperties,
                                          CompareFunc)；
        if ( pSubTree != NULL )
        {
            return pSubTree；
        }
        return NULL；
}
```

5.1.5 使用树的遍历算法来实现 Xcopy 功能

在文件系统中，经常会复制某个目录下的所有文件和子目录及其包含的文件到另外一个目录中，在 DOS 操作系统中，实现这个功能的命令名叫 Xcopy，下面就使用树的递归遍历算法来实现这个 Xcopy 功能。

下面以流行的 Windows 操作系统来实现这个功能，实现编码如下。

```
#include <windows.h>
#include <stdio.h>

void CopyCurrentDir( char *pszSrcDir, char *pszTargetDir, BOOL bOverWrite )；

/** 将一个目录及其子目录下的所有文件复制到另外一个目录下
    @param char *pszSrcDir——要复制的源目录
    @param char *pszTargetDir——目标目录
    @param BOOL bOverWrite——覆盖标志，为 FALSE 表示覆盖原有文件
    @return void——无
*/
void Xcopy( char *pszSrcDir, char *pszTargetDir, BOOL bOverWrite )
{
    char szBaseSearch[MAX_PATH]；
    char szCurrentPath[MAX_PATH]；
    HANDLE hFindFile；
    WIN32_FIND_DATA FindData；
    memset( szCurrentPath, 0, MAX_PATH )；
    GetCurrentDirectory( MAX_PATH, szCurrentPath )；
    sprintf( szBaseSearch, "%s\\*.*", szCurrentPath )；
    hFindFile = FindFirstFile( szBaseSearch, &FindData )；
    if ( hFindFile == INVALID_HANDLE_VALUE ) {
        return；
    }
    do {
        CreateDirectory( pszTargetDir, NULL )；
        if (  !strcmp( FindData.cFileName, "." ) ||
              !strcmp( FindData.cFileName, ".." )) {
              // skip . and ..
              continue；
        }
        if ( FindData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY ) {
              // a directory
              char szBaseDir[MAX_PATH]；
              char szTargeDir[MAX_PATH]；
              sprintf( szBaseDir, "%s\\%s", szCurrentPath, FindData.cFileName )；
              SetCurrentDirectory( szBaseDir )；
              _fullpath( szTargeDir, pszTargetDir, MAX_PATH )；
              sprintf( szTargeDir, "%s\\%s", FindData.cFileName )；
```

```
            Xcopy( szBaseDir, szTargeDir, bOverWrite )；
            SetCurrentDirectory( szCurrentPath )；
        }
    } while ( FindNextFile( hFindFile, &FindData ))；
    FindClose( hFindFile )；
    CopyCurrentDir( pszSrcDir, pszTargetDir, bOverWrite )；
}

/** 将一个目录下的所有文件拷贝到另外一个目录下
    @param char *pszSrcDir——要拷贝的目录
    @param char *pszTargeDir——目标目录
    @param BOOL bOverWrite——覆盖标志，为 FALSE 表示覆盖原有文件
    @return void——无
*/
void CopyCurrentDir( char *pszSrcDir, char *pszTargetDir, BOOL bOverWrite )
{
    char szBaseSearch[MAX_PATH]；
    HANDLE hFindFile；
    WIN32_FIND_DATA FindData；
    sprintf( szBaseSearch, "%s", pszSrcDir )；
    hFindFile = FindFirstFile( szBaseSearch, &FindData )；
    if ( hFindFile == INVALID_HANDLE_VALUE ) {
        return；
    }
    do {
        CreateDirectory( pszTargetDir, NULL )；
        if (  !strcmp( FindData.cFileName, "." ) ||
            !strcmp( FindData.cFileName, ".." )) {
            // skip and ..
            continue；
        }
        if ( FindData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY ) {
            // a directory
            continue；
        }
        else {
            // plain file
            char szSrcDir[MAX_PATH]；
            char szTargeDir[MAX_PATH]；
```

```
                sprintf( szSrcDir, "%s", FindData.cFileName )；
                sprintf( szTargeDir, "%s\\%s", pszTargetDir, FindData.cFileName )；
                CopyFile( szSrcDir, szTargeDir, bOverWrite )；
            }
        } while ( FindNextFile( hFindFile, &FindData ))；
        FindClose( hFindFile )；
    }
```

5.2 二叉树

5.2.1 二叉树的基本概念

在实际情况中，除了前面介绍的普通树外，另外一种最常用的树便是二叉树。二叉树可以看作一种特殊的普通树，它的每个节点刚好有两个子树(其中一个或两个可能为空)，两个子树分别被称作左子树与右子树。

一个典型的二叉树如图 5-6 所示。

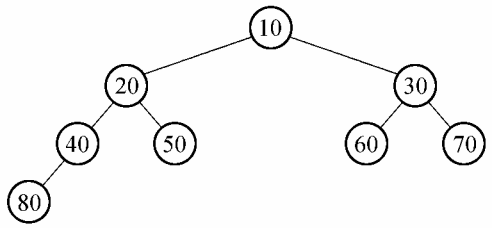


图 5-6　二叉树示意图

5.2.2 二叉树的树梢及二叉树的高度

为便于描述，把二叉树中所有满足左右子树中至少有一个为空的节点称为树梢节点。图 5-6 中的 80，40，50，60，70 均是树梢节点。将离根节点最近的树梢节点称为最小树梢节点，最小树梢节点到根节点的距离称为最小树梢高度。图中 40，50，60，70 四个节点都是最小树梢节点，最小树梢高度为 3。

将离根节点距离最远的树梢节点称为最大树梢节点，最大树梢节点到根节点的距离称为最大树梢高度，最大树梢高度便是二叉树的高度。

将二叉树中离根节点距离为 i 的所有节点称为第 i 层节点。

如果一颗二叉树的高度为 h，节点个数为 2^h-1，那么这颗树被称为完全二叉树。

定理 1　一颗二叉树为完全二叉树的充分必要条件是任一树梢节点到根节点的距

离都相等。

证明　假设二叉树的高度为 h，由完全二叉树的定义可以计算出完全二叉树第 h 层节点个数为 2^{h-1} 个，第 h－1 层的任一节点的左右子节点都不为空，因此完全二叉树第 h－1 层上不存在树梢节点，完全二叉树中所有树梢节点到根节点的距离都为 h，必要性得证。

如果任一树梢节点到根节点的距离都相等的话，那假设距离为 p，显然二叉树的高度也为 p，二叉树中的第 i(i<p)层中的任一节点的左右子节点都不为空，因此可以得出第 1 层节点至少有 2^0 个，第 2 层节点至少有 2^1 个，用归纳法可以得出第 p 层节点有 2^{p-1} 个，因此总节点个数 N 为

$$N = \sum_{i=0}^{p-1} 2^i = 2^p - 1$$

这就满足了完全二叉树的条件，因此充分性也得证。

定理 2　假设二叉树的最小树梢高度为 p，那么二叉树的节点个数大于等于 2^p-1 个。

证明　由于二叉树的最小树梢高度为 p，离根节点距离小于 p 的任一节点的左右子节点都不为空，这是因为如果有离根节点距离小于 p 的节点的左右子节点有为空，那么这个节点就是树梢节点，与最小树梢高度为 p 矛盾。

因此可以知道离根节点距离小于 p 的任一节点的两个子节点均不为空，用归纳法得出二叉树中的任意第 i(i<p)层中的节点个数为 2^{i-1}，所以二叉树上的总节点个数为

$$N \geq \sum_{i=0}^{p-1} 2^i = 2^p - 1$$

定理得证。

5.2.3　二叉树的描述方法

二叉树一般采用链式描述法，即二叉树的每个节点有两个指针分别指向它的左子树和右子树，然后节点中还需要有一个数据指针，为了编程方便，还需要一个父节点指针和一个整型的魔法变量。

父节点指针主要是用来消除对二叉树遍历时递归使用的，魔法变量主要是用来处理一些特殊用途，后面将在 AVL 树和红黑树中用到。

二叉树的节点可以用 C 语言结构体描述如下。

```
typedef struct BINTREEBASENODE_st {
    struct BINTREEBASENODE_st *pLeft；
    struct BINTREEBASENODE_st *pRight；
    struct BINTREEBASENODE_st *pParent；
```

```
    INT nMagic；
    void *pData；
} BINTREEBASENODE；
```

在二叉树中，只要保存一个根节点，通过根节点就可以访问到二叉树中的其他所有节点，另外为统计方便，再保存一个节点总个数。

```
typedef struct BINTREE_st {
    BINTREEBASENODE *pRoot；
    UINT uNodeCount；
} BINTREE；
```

5.3 二叉排序树

5.3.1 二叉排序树的基本概念

如果将二叉树中的节点元素按照一定大小顺序放置的话，就变成了一颗二叉排序树，又称为二叉搜索树。

定义　二叉排序树中的任一节点的左节点小于它，右节点大于它。

根据定义，可以得出这个结论：二叉排序树中的任一节点的左子树中的所有节点都小于它，右子树中所有节点都大于它。

这个定义包含了一个隐性前提，就是二叉排序树中没有大小相等的节点，如果要在二叉排序树中放入关键词相等的节点，只需要将前面定义中的小于改成小于等于，大于改成大于等于就可以了。有了大小顺序后，二叉排序树可以很方便地进行查找，还可以按从小到大的顺序进行遍历。

图 5-7 是一颗二叉排序树的示意图。

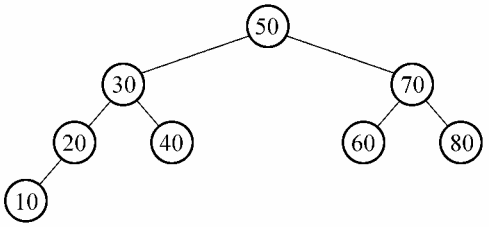


图 5-7　二叉排序树示意图

5.3.2 二叉排序树的查找

二叉排序树的查找算法非常简单，从根节点开始，将要查找的节点和二叉树中的

节点进行比较，如果大于则向右查找，小于则向左进行查找，等于则找到需要查找的节点。查找的比较次数最大不会超过二叉排序树的高度。

如要在图 5-8 中查找节点 40，只要按图中粗线所示方向进行查找，进行 3 次比较就能找到对应的节点，图 5-8 中粗线表示查找路径。

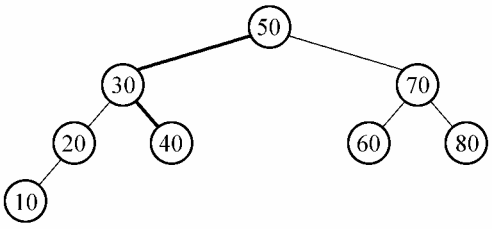


图 5-8　二叉排序树查找示意图

下面给出二叉排序树的查找的编码实现。

```
/**  二叉搜索树的查找函数
     @param BINTREEBASENODE *pRoot——二叉搜索树的根节点指针
     @param void *pData——要查找的数据指针
     @param COMPAREFUNC CompareFunc——数据比较回调函数
     @return void *——成功返回查找到的数据；失败返回 NULL
*/
void *BinTree_Find(BINTREEBASENODE *pRoot, void *pData,
                   COMPAREFUNC CompareFunc)
{
    BINTREEBASENODE *pNode；
    pNode = pRoot；
    while ( pNode != NULL )
    {
        INT nRet = (*CompareFunc)(pNode->pData, pData)；
        if ( nRet < 0 )
        {
            pNode = pNode->pRight；
        }
        else if ( nRet > 0 )
        {
            pNode = pNode->pLeft；
        }
        else
```

```
            {
                return pNode->pData；
            }
        }
        return NULL；
    }
```

5.3.3 二叉排序树的插入

二叉排序树的插入操作也非常简单，只要从根节点开始查找，如果比插入的节点大则往左查找，比插入节点小则往右查找，直到找到的节点的左节点或右节点为空时搜索终止，搜索到的节点就是要插入的位置，将新节点从这个位置插入。

如图 5-9 所示，插入节点 45 时，先从根节点 50 开始查找，50 比 45 大，从节点 50 往左查找到节点 30，30 比 45 小，从节点 30 往右查找到节点 40，40 比 45 小，继续往节点 40 的右边查找，但是节点 40 的右节点为空，因此将节点 45 插入在节点 40 的右边。

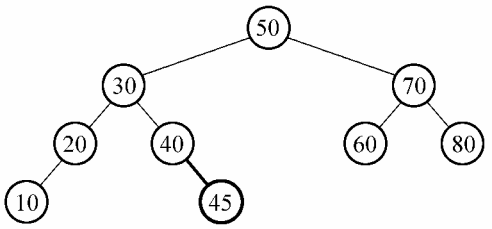


图 5-9 二叉排序树插入示意图

从二叉排序树的插入过程中可以看出，插入操作一定是插入在某个左右节点至少有一个为空的节点上，即插入在树梢节点上，所以新节点插入后，新节点的左右节点都是为空，新插入节点一定是树梢节点。

下面给出二叉排序树的插入编码实现。

```
/** 二叉排序树的插入操作函数
    @param BINTREEBASENODE **pRoot——指向根节点指针的指针
    @param void *pData——要插入的数据指针
    @param COMPAREFUNC CompareFunc——数据比较函数
    @param INT nMagic——魔法值，由调用者决定
    @return INT——返回 CAPI_FAILED 表示插入失败；返回 CAPI_SUCCESS 表示成功
*/
INT BinTree_Insert(BINTREEBASENODE **pRoot, void *pData,
```

```
COMPAREFUNC CompareFunc, INT nMagic)
{
    BINTREEBASENODE *pNode；
    BINTREEBASENODE *pNewNode；
    INT nRet = 0；
    if ( pData == NULL || CompareFunc == NULL )
    {
        return CAPI_FAILED；
    }
    pNode = *pRoot；
    while ( pNode != NULL )
    {
        nRet = (*CompareFunc)(pNode->pData, pData)；
        if ( nRet < 0 )
        {
            if ( pNode->pRight != NULL )
            {
                pNode = pNode->pRight；
                continue；
            }
        }
        else
        {
            if ( pNode->pLeft != NULL )
            {
                pNode = pNode->pLeft；
                continue；
            }
        }
        break；
    }
    pNewNode = (BINTREEBASENODE *)malloc(sizeof(BINTREEBASENODE))；
    if ( pNewNode == NULL )
    {
        return CAPI_FAILED；
    }
    pNewNode->pLeft = NULL；
    pNewNode->pRight = NULL；
    pNewNode->pData = pData；
```

```
        pNewNode->nMagic = nMagic；
        if ( pNode == NULL )
        {
            *pRoot = pNewNode；
            pNewNode->pParent = NULL；
        }
        else
        {
            if ( nRet < 0 )
            {
                pNode->pRight = pNewNode；
            }
            else
            {
                pNode->pLeft = pNewNode；
            }
            pNewNode->pParent = pNode；
        }
        return CAPI_SUCCESS；
    }
```

5.3.4 二叉排序树的删除

二叉排序树的删除操作较插入操作要复杂，根据被删节点的不同有如下三种情况。

1. 被删除节点的左右节点均不存在

如图 5-10 中，欲删除节点 45，只要直接将节点 45 删除掉，将节点 45 的父节点 40 的右节点指针指向空即可。如图 5-11 所示，虚线部分表示被删除掉的节点和链接。

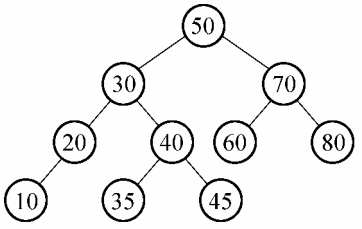


图 5-10 删除前的二叉排序树

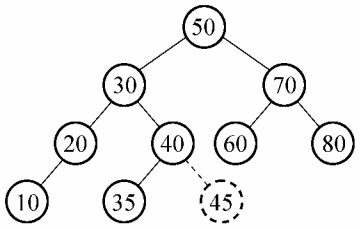


图 5-11 删除节点 45 后的二叉排序树

2. 被删除节点的左右节点有一个不存在

如图 5-10 中，欲删除节点 20，只要将节点 20 删除掉，将节点 20 的父节点的左节

点指针指向节点 10 即可，如图 5-12 所示。

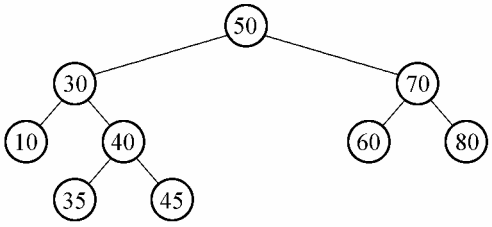


图 5-12　删除节点 20 后的二叉排序树

3. 被删除节点的左右节点都存在

如图 5-10 中，欲删除节点 50，将节点 50 删除后，如果将节点 30 或 70 移到 50 的位置，由于节点 30 和 70 的左右节点都存在，简单地移上去会导致移上去的节点的左右节点难以处理，比如将节点 70 移上去，节点 70 的左节点 60 如何处理便是问题。

因此，在删除节点时必须要找出一个合适的节点来替换被删除的节点。根据二叉排序树的定义，替换的节点必须满足替换后它的左子树的所有节点比它小，右子树所有节点比它大的原则。因此删除节点 50 时，只要在节点 50 的左子树中找一个最大的节点，或者右子树中找一个最小的节点来替换节点 50，就能保证二叉树仍然是一颗二叉排序树。如图 5-13 所示，若要删除节点 50，节点 45 和 60 都可以用来替换节点 50。

当删除节点 50 后，可以从节点 50 的左子树中找一个最大节点 45 来替换它，实际删除过程只要将 50 节点的数据指针指向 45 节点的数据指针，将节点 50 的数据和节点 45(不包括数据)删除掉，将节点 45 的左子树变成父节点 40 的右子树，删除后的情况如图 5-14 所示。

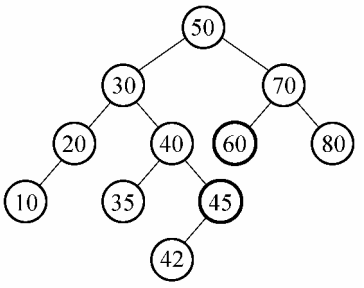


图 5-13　删除前的二叉排序树

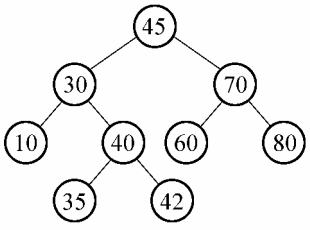


图 5-14　删除节点 50 后的二叉排序树

需要注意的是，以上所有情况下都需要考虑删除节点刚好是根节点的情况，此时

需要考虑修改根节点指针的指向，特别是如果二叉排序树中只有一个节点时，需要将根节点指向空。

下面给出二叉排序树的删除操作的编码实现。

```
/** 二叉排序树的删除函数
    @param BINTREE *pBinTree——二叉树指针
    @param void *pData——要删除的数据指针
    @param COMPAREFUNC CompareFunc——数据比较回调函数
    @param DESTROYFUNC DestroyFunc——数据释放回调函数
    @return INT——CAPI_FAILED 表示失败；CAPI_SUCCESS 表示成功
*/
INT BinTree_Delete( BINTREE *pBinTree, void *pData, COMPAREFUNC    CompareFunc,
                    DESTROYFUNC    DestroyFunc)
{
    BINTREEBASENODE *pNode；
    BINTREEBASENODE *pParentNode；
    BINTREEBASENODE *pMaxNode；
    BINTREEBASENODE *pParentMaxNode；
    INT nRet；
    if ( pBinTree == NULL ||pBinTree->pRoot == NULL
        || pData == NULL || CompareFunc == NULL )
    {
        return CAPI_FAILED；
    }
    pNode = pBinTree->pRoot；
    pParentNode = NULL；
    while ( pNode != NULL )
    {
        pParentNode = pNode；
        nRet = (*CompareFunc)(pNode->pData, pData)；
        if ( nRet < 0 )
        {
            pNode = pNode->pRight；
        }
        else if ( nRet > 0 )
        {
            pNode = pNode->pLeft；
        }
```

```
        else
        {
            break；
        }
    }
    if ( pNode == NULL )
    {
        /* 未找到指定节点 */
        return CAPI_FAILED；
    }
    /* 处理查找到的 pNode 有两个子节点的情况 */
    if ( pNode->pLeft != NULL && pNode->pRight != NULL )
    {
        pMaxNode = pNode->pLeft；
        pParentMaxNode = pNode；
        while ( pMaxNode->pRight != NULL )
        {
            pParentMaxNode = pMaxNode；
            pMaxNode = pMaxNode->pRight；
        }
        if ( DestroyFunc != NULL && pNode->pData != NULL )
        {
            (*DestroyFunc)(pNode->pData)；
        }
        pNode->pData = pMaxNode->pData；
        if ( pMaxNode == pNode->pLeft )
        {
            pNode->pLeft = pMaxNode->pLeft；
        }
        else
        {
            pParentMaxNode->pRight = pMaxNode->pLeft；
        }
        free(pMaxNode)；
        return CAPI_SUCCESS；
    }
    /* 处理最多只有一个子节点的情况 */
    if ( pNode->pLeft != NULL )
    {
```

```
            pMaxNode = pNode->pLeft；
        }
        else
        {
            pMaxNode = pNode->pRight；
        }
        if ( pNode == pBinTree->pRoot )
        {
            pBinTree->pRoot = pMaxNode；
        }
        else
        {
            if ( pParentNode->pLeft == pNode )
            {
                pParentNode->pLeft = pMaxNode；
            }
            else
            {
                pParentNode->pRight = pMaxNode；
            }
        }
        if ( DestroyFunc != NULL && pNode->pData != NULL )
        {
            (*DestroyFunc)(pNode->pData)；
        }
        free( pNode )；
        return CAPI_SUCCESS；
    }
```

5.3.5 二叉排序树的遍历

二叉树的遍历操作可以分为前序遍历、中序遍历、后序遍历和宽度遍历四种方式。二叉排序树的遍历也是这四种方式，唯一的区别是二叉排序树的中序遍历是按照从小到大的顺序进行的。

在设计二叉树节点时，节点里设计了父节点指针，因此可通过父节点指针虚拟栈来实现非递归的遍历操作。

下面给出以中序遍历为例实现二叉排序树的遍历操作的编码实现。

```
/** 二叉排序树的中序遍历函数
```

```
    @param BINTREE *pTree——要操作的二叉排序树
    @param VISITFUNC VisitFunc——节点访问回调函数
    @return void——无
*/
void BinTree_InOrderTraverse(BINTREE *pTree, VISITFUNC VisitFunc)
{
    BINTREEBASENODE *pCursor；
    void *pData；
    pCursor = pTree->pRoot；
    if ( pCursor == NULL )
    {
        return；
    }
    while ( pCursor->pLeft != NULL )
    {
        pCursor = pCursor->pLeft；
    }
    while ( pCursor != NULL )
    {
        pData = pCursor->pData；
        (*VisitFunc)(pData)；
        if ( pCursor->pRight != NULL )
        {
            pCursor = pCursor->pRight；
            while ( pCursor->pLeft != NULL )
            {
                pCursor = pCursor->pLeft；
            }
        }
        else
        {
            /* 回溯到父节点 */
            BINTREEBASENODE *pParent = pCursor->pParent；
            while ( pParent != NULL && pParent->pRight == pCursor )
            {
                pCursor = pParent；
                pParent = pParent->pParent；
            }
            pCursor = pParent；
        }
```

```
        }
    }
```

5.3.6 二叉排序树的旋转操作

在二叉排序树的插入操作中，如果插入数据顺序不当，有可能造成树的高度很高，导致查找效率低下。最坏的情况下，二叉排序树的所有节点都在一条链上，比如依次插入 90，80，70，60，50，40 这 6 个节点，将形成如图 5-15 所示的情况。

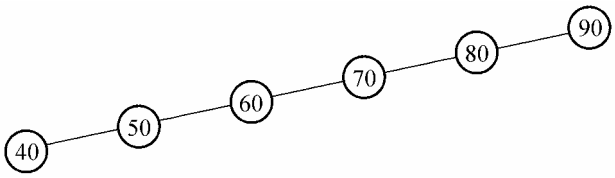


图 5-15 二叉排序树的最坏情况示意图

在图 5-15 中，对二叉排序树的查找退化成了链表的查找，因此二叉排序树中的查找最坏复杂度为 O(n)。

为了使二叉排序树中插入或删除节点后保证树维持一个合适的高度，需要对树做一些调整来改变树的高度，通常进行的两个操作叫做左旋和右旋。后面讲的 AVL 搜索树和红黑树便会用这两个操作来调整树的高度。

下面来讨论如何通过左旋和右旋操作来调整二叉排序树的高度。

1. 右旋操作

右旋操作如图 5-16 所示。

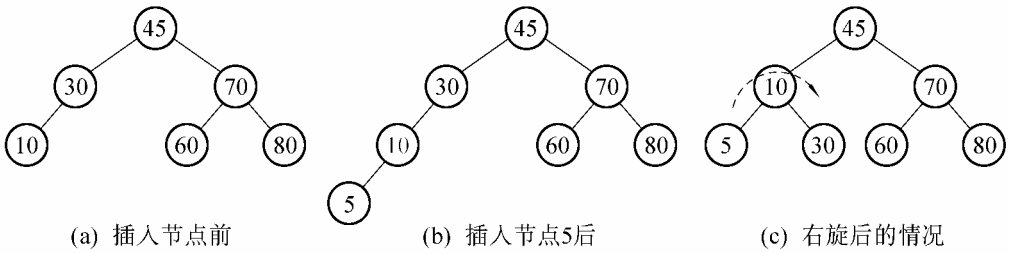


图 5-16 二叉排序树的右旋示意图

图 5-16 中可以看出，当插入了节点 5 后，二叉排序树的高度增加了，此时将节点做如图 5-16(c)中的调整，高度马上减少了，这种调整看起来像是绕着图 5-16(b)中的节点 30 向右旋转 90 °而得到的结果，因此把这种旋转称为右旋。

2. 左旋操作

左旋操作如图 5-17 所示。删除节点 60 后变成了图 5-17(b)所示的情况，此时做如图 5-17(c)所示的调整，树的高度减少了，这种调整看起来像是绕图 5-17(b)中的节点 70 向左旋转 90°的操作，因此把这种旋转称为左旋。

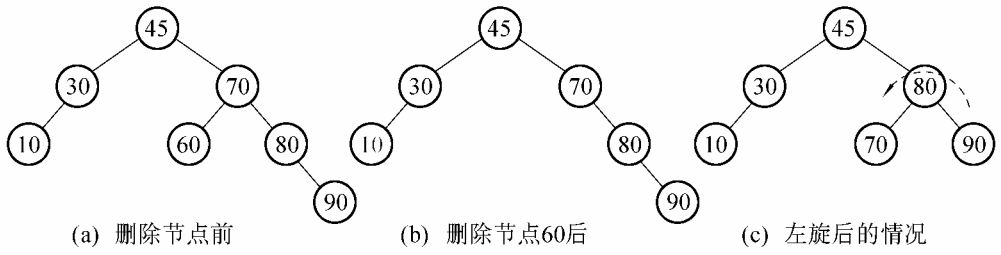


图 5-17 二叉排序树的左旋示意图

以上只是给出了左旋和右旋操作的最简单情况的实例，后面在介绍 AVL 树时还会继续讨论左旋和右旋操作的普遍情况。

下面给出左旋操作和右旋操作的编码实现。

```
/** 二叉排序树的左旋操作函数
    @param BINTREEBASENODE *pANode——要旋转的基节点
    @param BINTREEBASENODE **ppRoot——二叉排序树的根节点
    @return void——无
*/
void BinTree_RotateLeft(BINTREEBASENODE *pANode, BINTREEBASENODE **ppRoot )
{
    BINTREEBASENODE *pBNode；/* B 节点指针 */
    pBNode = pANode->pRight；
    /* 将 B 节点的左节点变成 A 节点的右节点 */
    pANode->pRight = pBNode->pLeft；
    if ( pBNode->pLeft != NULL )
    {
        pBNode->pLeft->pParent = pANode；
    }
    /* 修改 A 节点的父指针和 B 节点的关系 */
    pBNode->pParent = pANode->pParent；
    if ( pANode == *ppRoot )
    {
        *ppRoot = pBNode；
```

```
    }
    else if ( pANode == pANode->pParent->pLeft )
    {
        pANode->pParent->pLeft = pBNode；
    }
    else
    {
        pANode->pParent->pRight = pBNode；
    }
    /* 将 A 节点变成 B 节点的左节点 */
    pBNode->pLeft = pANode；
    pANode->pParent = pBNode；
}

/** 二叉排序树的右旋操作函数
    @param BINTREEBASENODE *pANode——要旋转的基节点
    @param BINTREEBASENODE **ppRoot——二叉排序树的根节点
    @return void——无
*/
void BinTree_RotateRight(BINTREEBASENODE *pANode, BINTREEBASENODE **ppRoot)
{
    BINTREEBASENODE   *pBNode；/* B 节点指针 */
    pBNode = pANode->pLeft；
    /* 将 B 节点的右节点变成 A 节点的左节点 */
    pANode->pLeft = pBNode->pRight；
    if ( pBNode->pRight != NULL )
    {
        pBNode->pRight->pParent = pANode；
    }
    /* 修改 A 节点的父指针和 B 节点的关系 */
    pBNode->pParent = pANode->pParent；
    if ( pANode == *ppRoot )
    {
        *ppRoot = pBNode；
    }
    else if ( pANode == pANode->pParent->pRight )
    {
        pANode->pParent->pRight = pBNode；
```

```
        }
        else
        {
            pANode->pParent->pLeft = pBNode；
        }
        /* 将 A 节点变成 B 节点的左节点 */
        pBNode->pRight = pANode；
        pANode->pParent = pBNode；
    }
```

5.4 AVL 搜索树

5.4.1 AVL 搜索树的基本概念

为了提高二叉排序树的搜索性能，必须将二叉排序树的高度降低。要将高度达到最低，高度必须是 O(logn)，则必须使尽可能多的节点都有左右子树，二叉均衡树便是这样一种树。它又称为 AVL 树，是 1962 年 Adelson-Velskii 和 Landis 提出的一种非常流行的二叉搜索树，满足如下两个条件的树便是 AVL 树。

① 如果树 T 是 AVL 树，那么它的左右子树都是 AVL 树；

② T 的左右子树高度差小于等于 1。

可以根据 AVL 树的定义推算出 AVL 树的高度。假设 N_h 表示一颗高度为 h 的 AVL 树的最小节点数，在最坏情况下，它有一颗高为 h－1 的子树和一颗高为 h－2 的子树，因此有

$$N_h = N_{h-1} + N_{h-2} + 1，N_0 = 0，N_1 = 1$$

令

$$C = (1+\sqrt{5})/2，B_h = N_h + (-1+\sqrt{5})/2 \cdot N_{h-1} + C$$

有

$$B_h = C \cdot B_{h-1}$$

可以推算出 $N^h \approx C^{h+2}/(-1+\sqrt{5})$，因此 $h \approx 1.44\log_2(n+2) = O(\log n)$

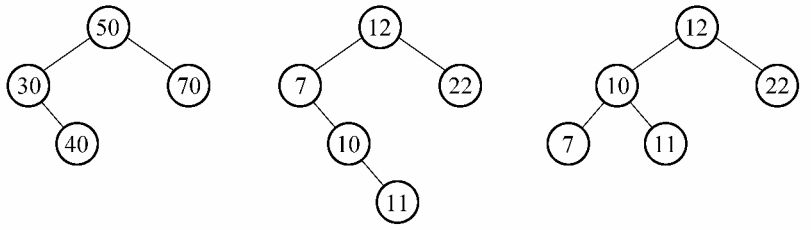
把满足二叉排序树的 AVL 树称为 AVL 搜索树。对 AVL 搜索树来说，为了简化插入和删除操作，要为每个节点引入一个平衡值，节点 X 的平衡值定义为节点 X 的左子树高度－右子树高度，由 AVL 树的定义知道，平衡因子取值可能为－1, 0, 1 中的一个。

显然，AVL 搜索树的搜索和二叉排序树搜索的算法一样，但插入操作和删除操作就不能完全按照二叉排序树的插入和搜索来进行了，因为那样得到的可能不是 AVL 搜索树。

5.4.2 AVL 搜索树的插入

AVL 搜索树的插入操作和二叉搜索树的插入过程是一样的，但是完成插入之后，得到的树可能不是 AVL 树，因此要通过一定的方法重新调整使之达到平衡。如图 5-18 所示。

在图 5-18 中，图 5-18(a)是一颗 AVL 搜索树，当插入节点 11 后，变成了一颗非 AVL 搜索树如图 5-18(b)所示，图 5-18(c)是重新平衡后的 AVL 搜索树。



(a) 插入前的AVL树　　(b) 插入节点11后　　(c) 重新平衡后

图 5-18　AVL 搜索树中插入节点

1. AVL 搜索树的平衡分类

一般 AVL 搜索树插入节点后是否平衡取决于在插入路径上是否存在不平衡点，如果插入前不存在不平衡点，那么在插入路径上各节点的平衡因子最多变为－1 或 1,仍然是 AVL 搜索树。当在插入路径上存在不平衡点时，把离插入点最近的不平衡点称为 A 节点，这时要分以下四种情况考虑。

1) 插入前 A 节点平衡因子为－1,插入节点在 A 节点的左子树中,不影响树的平衡。

2) 插入前 A 节点平衡因子为 1,插入节点在 A 节点的右子树中,不影响树的平衡。

3) 插入前 A 节点平衡因子为 1，插入节点在 A 节点的左子树中，影响树的平衡。

4) 插入前 A 节点平衡因子为－1,插入节点在 A 节点的右子树中,影响树的平衡。

前面两种情况只要调整从 A 节点到插入节点路径上的平衡因子即可。

通常把第三种情况称为 L 型，如果插入在 A 节点的左子树的左子树中称为 LL 型，插入在 A 节点的左子树的右子树中称为 LR 型。

同理，可以将第四种情况分为 RL 型和 RR 型两种。因此 AVL 搜索树需要调整不平衡节点的情况共四种：LL 型、LR 型、RL 型和 RR 型，其他情况只要调整插入路径上的平衡因子值即可。由于 RL 型和 RR 型是对称的，因此只要分析清楚 LL 型和 LR 型就可以了，下面用图解的方法详细描述对 LL 型和 LR 型的调整。

2. 平衡的调整

1) AVL 搜索树的 LL 型平衡调整如图 5-19 所示。

图 5-19 中，虚线圈的节点表示节点可能为空节点；h 和 h+1 表示对应节点的高度；节点旁的数字 0、1、2 等表示对应节点的平衡因子值。由于 A 节点的平衡因子为 1，因此插入前 A 节点的左节点 B 一定存在，插入发生在 B 节点的左子树中，插入后 BL' 节点一定存在。

从图 5-19 可以看出，LL 型的调整方法是将 A 节点按顺时针方向向右转动了一下，B 节点也是以 A 节点为中心向右转动，把这种调整称为右旋，与此对称的 RR 型的调整方法称为左旋。

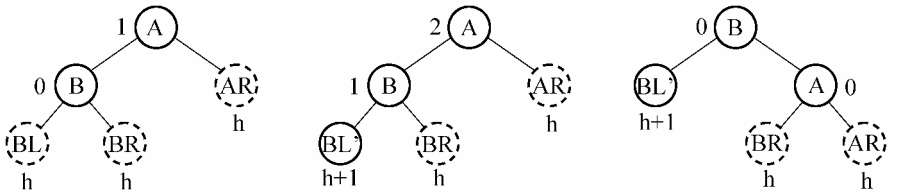


图 5-19　AVL 搜索树的 LL 型平衡调整图

经过旋转之后，B 节点和 A 节点的平衡因子值都变成了 0，从 B 节点到插入节点路径上的平衡因子需要重新调整，由于 B 节点的高度和插入前的 A 节点高度是一样的，因此从根节点到 B 节点的路径上的节点平衡因子都未改变，不需要调整。

2) AVL 搜索树的 LR 型调整如图 5-20 所示。

由于插入在 B 节点的右子树中，因此插入后 B 节点的右节点 C 节点一定存在。

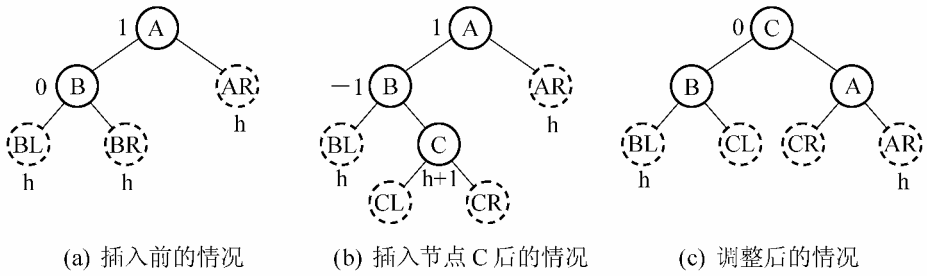


图 5-20　AVL 搜索树的 LR 型平衡调整图

从图 5-20 可以看出，LR 型的调整和 LL 型的有点区别，可以把它看成是先将 B 节点左旋，然后再将 A 节点右旋得到。有些书将这种旋转称为双旋转。

3. 平衡因子的调整

经过旋转之后，从图上可以看出 C 节点的高度和插入前 A 节点的高度是一样的，因此从根节点到 C 节点路径上的平衡因子不需要调整。调整后 C 节点的平衡因子变成了 0，但 B 节点和 A 节点的平衡因子要根据插入后，重新调整前 C 节点的平衡因子来决定(即根据图 5-20(b)中 C 节点的平衡因子值来决定)，可分为以下三种情况。

1) 如果 C 节点平衡因子为 0，则 CL 和 CR 节点的高度均为 h，因此调整后 A 节点和 B 节点平衡因子都为 0。稍微分析一下就可以知道 C 节点就是插入节点，因为 A 节点是离插入节点最近的节点，A 节点和插入节点路径上所有的节点的平衡因子值都为 0，因此插入节点后，在图 5-20(b)中，如果 C 节点不是插入节点，C 节点的平衡因子一定会改变。但 C 节点原来的平衡因子为 0，改变后成为－1 或 1，因此图 5-20(b)中 C 节点平衡因子为 0 的唯一可能是 C 节点就是插入节点。此时从图 5-20(c)中可以看出，除 A 和 B 节点外，其他节点的平衡因子都不需要调整。

2) 如果 C 节点平衡因子为－1，则 CL 节点的高度为 h－1，CR 节点的高度为 h，插入操作发生在 CR 中。调整后 A 节点平衡因子为 0，B 节点平衡因子为 1，从 A 节点到插入节点的路径上的节点平衡因子需要重新调整。

3) 如果 C 节点平衡因子为 1，则 CL 节点的高度为 h，CR 节点的高度为 h－1，插入操作发生在 CL 中。调整后 A 节点平衡因子为－1，B 节点平衡因子为 0，从 B 节点到插入节点路径上的节点的平衡因子需要重新调整。

5.4.3 AVL 搜索树的删除

1. 平衡因子的调整

AVL 搜索树的删除操作比插入操作要稍微复杂一些，首先要确定删除后，哪个节点的平衡需要重新调整。

1) 当删除节点的左右子节点都不存在时，删除的就是这个节点，记删除节点的父节点为 A 节点。

2) 当删除节点的左节点不存在时，用右节点取代删除节点；当删除节点的右节点不存在时用左节点取代删除节点，记删除节点为 A 节点。

3) 当删除节点的左右节点都存在时，使用删除节点左子树的最大节点来取代删除节点，记左子树最大节点的父节点为 A 节点；如果左子树最大节点就是左节点，则记删除节点为 A 节点。

可以看出，以上三种情况下 A 节点的平衡因子都会改变，如果操作发生在 A 节点的左子树，那么 A 节点的平衡因子减 1，如果删除操作发生 A 节点的右子树，则 A 节

点的平衡因子加 1。

2. 几种不平衡类型

可能有以下三种调整情况。

1) 如果 A 节点的新的平衡因子是 0，则表明它原来的平衡因子为－1 或 1，A 节点原来的左右子树高度是不相等的，此时变成了相等，有一个子树的高度减少了 1，A 节点的高度也减少了 1。因此需要改变 A 节点的父节点的平衡因子，并且要根据修改后的 A 节点的父节点的平衡因子值来划分不平衡类型，不平衡类型可以是 1)、2)、3)种的任意一种。

2) 如果 A 节点的新的平衡因子是－1 或 1，表明它原来的平衡因子为 0，即它的左右子树的高度是相等的。删除操作发生后，A 节点的高度不会改变，因此其他节点的平衡因子不需要改变。

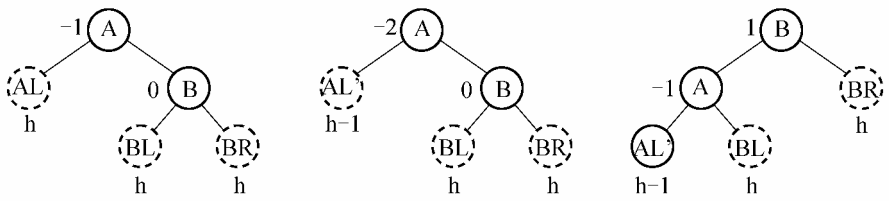
3) 如果 A 节点的新的平衡因子是－2 或 2，那么 AVL 搜索树的平衡需要重新进行调整。

对于第 3)种情况的调整，可以根据删除操作发生在 A 节点的左子树还是右子树来划分不平衡类型。如果删除操作发生在左子树，即 A 节点的平衡因子为－2 的情况，可以把不平衡类型称为 L 型；如果删除操作发生在右子树，即 A 节点的平衡因子为 2 的情况，把这种不平衡类型称为 R 型。

3. 不同类型的调整

由于 L 型和 R 型是对称的，因此只要分析清楚 L 型，R 型也自然就清楚了。由于 L 型情况下 A 节点的平衡因子为－2，因此删除前它的平衡因子必定为－1，因此 A 节点的右节点 B 一定存在，并且删除操作一定发生在 A 节点的左子树。可以根据 B 节点的平衡因子值将 L 型再细分为 L0 型(B 节点的平衡因子为 0)、L1 型(B 节点的平衡因子为 1)、L－1 型(B 节点的平衡因子为－1)三种类型。

1) L0 型的调整如图 5-21 所示。



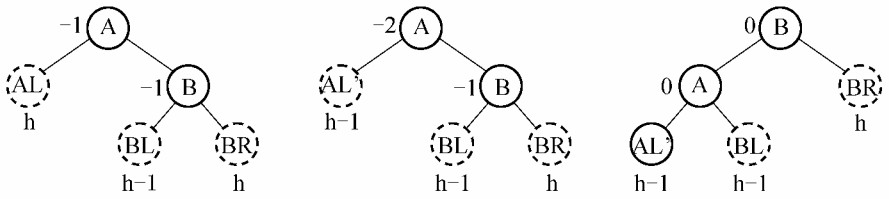
(a) 删除前的情况　(b) 删除后的情况　(c) 调整后的情况

图 5-21　AVL 搜索树删除节点后的 L0 型调整图

由图 5-21 可以看出，L0 型的调整实际上是对 A 节点作一次左旋得到。旋转后 A 节点平衡因子变成－1，B 节点平衡因子变成 1，这时转变成了前面不平衡的第二种情况。此时 B 节点的父节点及其他节点都不需要重新进行调整，因此 L0 型不平衡只要一次旋转便达到了平衡。

R0 和 L0 是对称的，R0 型需要对 A 节点作一次右旋，旋转后，A 节点的平衡因子变成 1，B 节点的平衡因子变成－1。

2) L－1 型的调整如图 5-22 所示。



(a) 删除前的情况　　(b) 删除后的情况　　(c) 调整后的情况

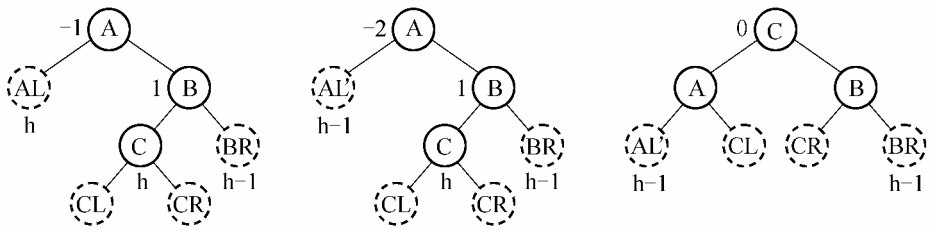
图 5-22　AVL 搜索树删除节点后的 L－1 型调整图

L－1 型也是作一次左旋操作，旋转后 A 和 B 节点的平衡因子都变成了 0，转变成了前面不平衡的第一种情况，按前面第一种情况介绍的方法重新对调整后 B 节点的父节点进行调整即可。

与 L－1 型对称的是 R1 型，R1 型需要作一次右旋，旋转后 A 和 B 节点的平衡因子也都变成了 0。

3) L1 型的调整如图 5-23 所示。

由于 B 节点的平衡因子为 1，因此 B 节点的左节点 C 一定存在。



(a) 删除前的情况　　(b) 删除后的情况　　(c) 调整后的情况

图 5-23　AVL 搜索树删除节点后的 L1 型调整图

L1 型是通过两次旋转进行调整的，先对 B 节点进行右旋，再对 A 节点进行左旋得到。旋转完后，C 节点的平衡因子变成 0，A、B 节点的平衡因子与前面插入时的

LR 旋转类似，需要根据图 5-23(b)中 C 节点的平衡因子来决定。

① 如果图 5-23(b)中 C 节点的平衡因子为 0，表明调整后 A、B 节点的平衡因子为 0；

② 如果图 5-23(b)中 C 节点的平衡因子为－1，表明 CL 高度为 h－2，CR 高度为 h－1，则调整后 A 节点的平衡因子为 1，B 节点的平衡因子为 0；

③ 如果图 5-23(b)中 C 节点的平衡因子为 1，表明 CL 高度为 h－1，CR 高度为 h－2，则调整后 A 节点的平衡因子为 0，B 节点的平衡因子为－1。

L1 型调整完后，C 节点平衡因子为 0，转化成了前面不平衡类型的第一种情况。R－1 型和 L1 型是对称的，只要将平衡因子 1 和－1 左右调换即可。

把删除操作的 L0，L1，L－1 和前面插入操作的 LL，LR 型进行比较可以发现，L0，L－1 和 LL 型是对称的，因此 L0，L－1 和 RR 型是一样的；并且 L－1 和 RR 型调整后的 A、B 节点的平衡因子也一样。L0 型调整后的区别是 A、B 节点的平衡因子不同，L1 型和 RL 型是完全一样的。

5.4.4 AVL 树的源代码

下面给出 AVL 树的编码实现。

```
typedef BINTREEBASENODE AVLTREENODE；
typedef struct AVLTREE_st {
    AVLTREENODE *pRoot；
    UINT uNodeCount；
} AVLTREE；

/**  AVL 树的创建函数
    @param void——无
    @return AVLTREE *——成功返回创建的 AVL 树指针；失败返回 NULL
*/
AVLTREE *AVLTree_Create(void)
{
    AVLTREE *pTree；
    pTree = (AVLTREE *)malloc(sizeof(AVLTREE))；
    if ( pTree != NULL )
    {
        pTree->pRoot = NULL；
        pTree->uNodeCount = 0；
    }
    return pTree；
}
```

```
/** AVL 树的释放函数，将以某个指定节点为根节点的 AVL 树及树中的全部节点都释放，
    采用后序遍历方式进行释放
    @param AVLTREENODE *pTreeNode——要释放的根节点
    @param DESTROYFUNC DestroyFunc——数据释放回调函数
    @return void——无
*/
void AVLTreeNode_Destroy(AVLTREENODE *pTreeNode, DESTROYFUNC DestroyFunc)
{
    if ( pTreeNode != NULL )
    {
        if ( pTreeNode->pLeft != NULL )
        {
            AVLTreeNode_Destroy(pTreeNode->pLeft, DestroyFunc)；
        }
        if ( pTreeNode->pRight != NULL )
        {
            AVLTreeNode_Destroy(pTreeNode->pRight, DestroyFunc)；
        }
        if ( DestroyFunc != NULL && pTreeNode->pData != NULL )
        {
            (*DestroyFunc)(pTreeNode->pData)；
        }
        free(pTreeNode)；
    }
}

/** AVL 树的释放函数，将一颗 AVL 树释放
    @param AVLTREE *pTree——要释放的 AVL 树指针
    @param DESTROYFUNC DestroyFunc——数据释放回调函数
    @return void——无
*/
void AVLTree_Destroy(AVLTREE *pTree, DESTROYFUNC DestroyFunc)
{
    if ( pTree == NULL )
    {
        return；
    }
    AVLTreeNode_Destroy( pTree->pRoot, DestroyFunc )；
    free( pTree )；
```

```
}

/** AVL 树的查找函数，调用了二叉搜索树的查找函数进行查找
    @param AVLTREE *pTree——要查找的 AVL 树指针
    @param void *pData——要查找的数据指针
    @param COMPAREFUNC CompareFunc——数据比较函数
    @return void *——成功返回查找到的目标数据；失败返回 NULL
*/
void *AVLTree_Find(AVLTREE *pTree, void *pData, COMPAREFUNC CompareFunc)
{
    return BinTree_Find(pTree->pRoot, pData, CompareFunc)；
}

/** AVL 树的调整平衡函数，主要是调整插入操作时的平衡
    @param AVLTREENODE *pStartNode——要调整平衡的起始节点
    @param void *pData——插入的数据指针
    @param COMPAREFUNC CompareFunc——数据比较回调函数
    @return void——无
*/
void AVLTree_FixBalance( AVLTREENODE *pStartNode, void *pData,
                         COMPAREFUNC CompareFunc)
{
    AVLTREENODE *pNode；
    AVLTREENODE *pSearchNode；
    INT nResult；
    pNode = pStartNode；
    while ( pNode != NULL )
    {
        pSearchNode = pNode；
        nResult = (*CompareFunc)(pNode->pData, pData)；
        if ( nResult < 0 )
        {
            pNode = pNode->pRight；
            pSearchNode->nMagic－= 1；
        }
        else if ( nResult > 0 )
        {
            pNode = pNode->pLeft；
            pSearchNode->nMagic += 1；
```

```
            }
            else
            {
                /* 找到相同关键词的节点，中止 */
                break；
            }
        }
        return；
}

/** AVL 树的左旋函数
    @param AVLTREE *pTree——AVL 树指针
    @param AVLTREENODE *pStartNode——要进行左旋操作的节点指针
    @return void——无
*/
void AVLTree_RotateLeft( AVLTREE *pTree, AVLTREENODE *pStartNode )
{
    BinTree_RotateLeft((BINTREEBASENODE *)pStartNode, &(pTree->pRoot))；
}

/** AVL 树的右旋函数
    @param AVLTREE *pTree——AVL 树指针
    @param AVLTREENODE *pStartNode——要进行右旋操作的节点指针
    @return void——无
*/
void AVLTree_RotateRight(AVLTREE *pTree, AVLTREENODE *pStartNode )
{
    BinTree_RotateRight((BINTREEBASENODE *)pStartNode, &(pTree->pRoot))；
}

/** AVL 树的双旋转操作函数，先左旋再右旋
    @param AVLTREENODE **ppRoot——AVL 树的根节点指针
    @param AVLTREENODE *pStartNode——要旋转的节点
    @return void——无
*/
void AVLTree_RotateLeftRight( AVLTREENODE **ppRoot,
                              AVLTREENODE *pStartNode, void *pData,
                              COMPAREFUNC CompareFunc)
{
    AVLTREENODE *pANode；
```

```
    AVLTREENODE *pBNode；
    AVLTREENODE *pCNode；
    INT nRet；
    nRet = 0；
    pANode = pStartNode；
    pBNode = pANode->pLeft；
    pCNode = pBNode->pRight；
    BinTree_RotateLeft(pBNode, ppRoot)；
    BinTree_RotateRight(pANode, ppRoot)；
    switch ( pCNode->nMagic )
    {
    case 1：
        /* 插入在 C 节点左子树的情况 */
        pANode->nMagic =－1；
        pBNode->nMagic = 0；
        break；
    case－1：
        /* 插入在 C 节点右子树的情况 */
        pANode->nMagic = 0；
        pBNode->nMagic = 1；
        break；
    default：
        /* C 节点就是插入点的情况 */
        pANode->nMagic = 0；
        pBNode->nMagic = 0；
        break；
    }
    pCNode->nMagic = 0；
}

/** AVL 树的双旋转函数，先右旋再左旋
    @param AVLTREENODE **ppRoot——AVL 树根节点指针
    @param AVLTREENODE *pStartNode——要旋转的节点指针
    @return void——无
*/
void AVLTree_RotateRightLeft( AVLTREENODE **ppRoot,
                              AVLTREENODE *pStartNode, void *pData,
                              COMPAREFUNC CompareFunc)
{
```

```
    AVLTREENODE *pANode；
    AVLTREENODE *pBNode；
    AVLTREENODE *pCNode；
    INT nRet；
    nRet = 0；
    pANode = pStartNode；
    pBNode = pANode->pRight；
    pCNode = pBNode->pLeft；
    BinTree_RotateRight(pBNode, ppRoot)；
    BinTree_RotateLeft(pANode, ppRoot)；
    switch ( pCNode->nMagic )
    {
    case 1：
        /* 插入在 C 节点左子树的情况 */
        pANode->nMagic = 0；
        pBNode->nMagic =－1；
        break；
    case－1：
        /* 插入在 C 节点右子树的情况 */
        pANode->nMagic = 1；
        pBNode->nMagic = 0；
        break；
    default：
        /* C 节点就是插入点的情况 */
        pANode->nMagic = 0；
        pBNode->nMagic = 0；
        break；
    }
    pCNode->nMagic = 0；
}

/** AVL 树的插入操作函数
    @param AVLTREE *pTree——要执行插入操作的 AVL 树指针
    @param void *pData——要插入的数据指针
    @param COMPAREFUNC CompareFunc——数据比较函数
    @return INT——返回 CAPI_SUCCESS 表示成功；失败返回 CAPI_FAILED
*/
INT AVLTree_Insert(AVLTREE *pTree, void *pData, COMPAREFUNC CompareFunc)
{
```

```
    INT nRet；
    if ( pTree == NULL || pData == NULL || CompareFunc == NULL )
    {
        return CAPI_FAILED；
    }
    nRet = AVLTreeNode_Insert(&(pTree->pRoot), pData, CompareFunc)；
    if ( nRet != CAPI_FAILED )
    {
        pTree->uNodeCount += 1；
    }
    return nRet；
}

/** AVL 树的插入数据函数，将一个数据插入到 AVL 树中
    @param AVLTREENODE **ppRootNode——要插入的 AVL 树根节点指针
    @param void *pData——要插入的数据
    @param COMPAREFUNC CompareFunc——数据比较回调函数
    @return INT——返回 CAPI_SUCCESS 表示插入成功；返回 CAPI_FAILED 表示失
                  败，失败的原因有内存分配失败和已存在相同关键词数据两种情况
*/
INT AVLTreeNode_Insert( AVLTREENODE **ppRootNode, void *pData,
                        COMPAREFUNC CompareFunc)
{
    AVLTREENODE *pNode；
    AVLTREENODE *pANode；
    AVLTREENODE *pNewNode；
    AVLTREENODE *pSearchNode；
    AVLTREENODE *pBNode；
    AVLTREENODE *pCNode；
    INT nRet；
    nRet = 0；
    pANode = NULL；
    pSearchNode = NULL；
    /* 查找要插入节点的位置，并且在查找过程中要记录最后一个不平衡的节点 */
    pNode = *ppRootNode；
    while ( pNode != NULL )
    {
        pSearchNode = pNode；
        if ( pSearchNode->nMagic ==－1 || pSearchNode->nMagic == 1 )
```

```
        {
            pANode = pSearchNode；
        }
        nRet = (*CompareFunc)(pNode->pData, pData)；
        if ( nRet < 0 )
        {
            pNode = pNode->pRight；
        }
        else if ( nRet > 0 )
        {
            pNode = pNode->pLeft；
        }
        else
        {
            /* 找到相同关键词的节点，插入失败 */
            return CAPI_FAILED；
        }
    }
    pNewNode = (AVLTREENODE *)malloc( sizeof(AVLTREENODE) )；
    if ( pNewNode == NULL )
    {
        return CAPI_FAILED；
    }
    pNewNode->pData = pData；
    pNewNode->pLeft = NULL；
    pNewNode->pRight = NULL；
    pNewNode->pParent = pSearchNode；
    pNewNode->nMagic = 0；
    if ( pSearchNode == NULL )
    {
        *ppRootNode = pNewNode；
        return CAPI_SUCCESS；
    }
    if ( nRet < 0 )
    {
        pSearchNode->pRight = pNewNode；
    }
    else
    {
```

```
        pSearchNode->pLeft = pNewNode；
    }
    /* 不存在不平衡的节点，直接插入后再修改平衡因子即可 */
    if ( pANode == NULL )
    {
        /* 修改从根节点到插入节点的平衡因子 */
        AVLTree_FixBalance(*ppRootNode, pData, CompareFunc)；
        return CAPI_SUCCESS；
    }
    nRet = (*CompareFunc)(pANode->pData, pData)；
    /* 以下处理存在不平衡节点的情况 */
    if ( ( pANode->nMagic == 1 && nRet < 0 )
        || (pANode->nMagic ==－1 && nRet > 0 ) )
    {
        /* A 节点平衡因子为 1 且插入节点插入在右子树的情况，以及 A 节点平衡
        * 因子为－1 且插入在左子树的情况，这两种情况插入后还是平衡树，只需
        * 修改相关节点平衡因子
        */
        AVLTree_FixBalance(pANode, pData, CompareFunc)；
    }
    else if ( pANode->nMagic == 1 && nRet > 0 )
    {
        if ( (*CompareFunc)(pANode->pLeft->pData, pData) > 0 )
        {
            /* LL 型不平衡，插入在 A 节点左子树的左子树中 */
            pBNode = pANode->pLeft；
            BinTree_RotateRight(pANode, ppRootNode)；
            AVLTree_FixBalance(pBNode, pData, CompareFunc)；
            pANode->nMagic = 0；
            pBNode->nMagic = 0；
        }
        else
        {
            INT nRetVal；
            /* LR 型不平衡，插入在 A 节点左子树的右子树中 */
            pBNode = pANode->pLeft；
            pCNode = pBNode->pRight；
            nRetVal = (*CompareFunc)(pCNode->pData, pData) ；
            if ( nRetVal > 0 )
```

```
            {
                pCNode->nMagic += 1；
            }
            else if ( nRetVal < 0 )
            {
                pCNode->nMagic－= 1；
            }
            AVLTree_RotateLeftRight(ppRootNode, pANode, pData, CompareFunc)；
            if ( nRetVal > 0 )
            {
                AVLTree_FixBalance(pBNode, pData, CompareFunc)；
            }
            else if ( nRetVal < 0 )
            {
                AVLTree_FixBalance(pANode, pData, CompareFunc)；
            }
            else
            {
            /* 这个分支不需要做任何处理 */
            }
        }
    }
    else /* pANode->nMagic ==－1 && nRet < 0 的情况*/
    {
        if ( (*CompareFunc)(pANode->pRight->pData, pData) > 0 )
        {
            INT nRetVal；
            /* RL 型不平衡，插入在 A 节点右子树的左子树中 */
            pBNode = pANode->pRight；
            pCNode = pBNode->pLeft；
            nRetVal = (*CompareFunc)(pCNode->pData, pData)；
            if ( nRetVal > 0 )
            {
                pCNode->nMagic += 1；
            }
            else if ( nRetVal < 0 )
            {
                pCNode->nMagic－= 1；
            }
```

```
                AVLTree_RotateRightLeft(ppRootNode, pANode, pData, CompareFunc)；
                if ( nRetVal > 0 )
                {
                    AVLTree_FixBalance(pANode, pData, CompareFunc)；
                }
                else if ( nRetVal < 0 )
                {
                    AVLTree_FixBalance(pBNode, pData, CompareFunc)；
                }
                else
                {
                /* 这个分支不需要做任何处理 */
                }
            }
            else
            {
                /* RR 型不平衡，插入在 A 节点右子树的右子树中 */
                pBNode = pANode->pRight；
                BinTree_RotateLeft(pANode, ppRootNode)；
                AVLTree_FixBalance(pBNode, pData, CompareFunc)；
                pANode->nMagic = 0；
                pBNode->nMagic = 0；
            }
        }
        return CAPI_SUCCESS；
}

/** AVL 树的删除操作调整平衡函数，由删除函数调用
    @param AVLTREENODE **ppRoot——指向 AVL 树的根节点指针的指针
    @param AVLTREENODE *pNode——要调整平衡的节点
    @param void *pData——删除的数据指针
    @param COMPAREFUNC CompareFunc——数据比较回调函数
    @return void——无
*/
void AVLTree_AdjustBalanceForDelete( AVLTREENODE **ppRoot,
                                     AVLTREENODE *pNode, void *pData,
                                     COMPAREFUNC CompareFunc )
{
    AVLTREENODE *pANode；
```

```
AVLTREENODE *pParentNode；
AVLTREENODE *pBNode；
pANode = pNode；
while ( pANode != NULL )
{
    switch( pANode->nMagic )
    {
    case 0：
        if ( pANode == *ppRoot )
        {
            /* pANode 为根节点，高度减少 1，但左右子树高度相等，无需调整 */
            break；
        }
        else
        {
            pParentNode = pANode->pParent；
            if ( pANode == pParentNode->pLeft )
            {
                pParentNode->nMagic－= 1；
            }
            else
            {
                pParentNode->nMagic += 1；
            }
            /* 将 pANode 指向它的父节点，继续调整它的父节点的不平衡情况 */
            pANode = pParentNode；
            continue；
        }
    case－1：
    case 1：
        /* pANode 原来的平衡因子为 0，删除操作发生后，高度未改变，因此不需要
        * 再调整，退出即可
        */
        break；
    case－2：
        /* L 型不平衡情况 */
        pBNode = pANode->pRight；
        if ( pBNode->nMagic == 0 )
        {
```

```
        /* L0 型不平衡情况 */
        BinTree_RotateLeft(pANode, ppRoot)；
        pANode->nMagic =－1；
        pBNode->nMagic = 1；
        break；
    }
    else if ( pBNode->nMagic ==－1 )
    {
        /* L－1 型不平衡情况 */
        pParentNode = pANode->pParent；
        if ( pParentNode != NULL )
        {
            if ( pANode == pParentNode->pLeft )
            {
                pParentNode->nMagic－= 1；
            }
            else
            {
                pParentNode->nMagic += 1；
            }
        }
        BinTree_RotateLeft(pANode, ppRoot)；
        pANode->nMagic = 0；
        pBNode->nMagic = 0；
        /* 将 pANode 指向它的父节点，继续调整它的父节点的不平衡情况 */
        pANode = pParentNode；
    }
    else *pBNode->nMagic == 1 的情况 */
    {
        /* L1 型的情况 */
        pParentNode = pANode->pParent；
        if ( pParentNode != NULL )
        {
            if ( pANode == pParentNode->pLeft )
            {
                pParentNode->nMagic－= 1；
            }
            else
            {
```

```
                    pParentNode->nMagic += 1；
                }
            }
            AVLTree_RotateRightLeft(ppRoot, pANode, pData, CompareFunc)；
            /* 将 pANode 指向它的父节点，继续调整它的父节点的不平衡情况 */
            pANode = pParentNode；
        }
        continue；/* 继续 while() 循环 */
    case 2：
        /* R 型不平衡情况 */
        pBNode = pANode->pLeft；
        if ( pBNode->nMagic == 0 )
        {
            /* R0 型不平衡情况 */
            BinTree_RotateRight(pANode, ppRoot)；
            pANode->nMagic = 1；
            pBNode->nMagic =－1；
            break；
        }
        else if ( pBNode->nMagic ==－1 )
        {
            /* R－1 型不平衡情况 */
            pParentNode = pANode->pParent；
            if ( pParentNode != NULL )
            {
                if ( pANode == pParentNode->pLeft )
                {
                    pParentNode->nMagic－= 1；
                }
                else
                {
                    pParentNode->nMagic += 1；
                }
            }
            AVLTree_RotateLeftRight(ppRoot, pANode, pData, CompareFunc)；
            /* 将 pANode 指向它的父节点，继续调整它的父节点的不平衡情况 */
            pANode = pParentNode；
        }
        else /* pBNode->nMagic == 1 的情况 */
```

```
            {
                /* R1 型的情况 */
                pParentNode = pANode->pParent；
                if ( pParentNode != NULL )
                {
                    if ( pANode == pParentNode->pLeft )
                    {
                        pParentNode->nMagic－= 1；
                    }
                    else
                    {
                        pParentNode->nMagic += 1；
                    }
                }
                BinTree_RotateRight(pANode, ppRoot)；
                pANode->nMagic = 0；
                pBNode->nMagic = 0；
                /* 将 pANode 指向它的父节点，继续调整它的父节点的不平衡情况 */
                pANode = pParentNode；
            }
            continue；/* 继续 while() 循环 */
        default：
            break；
        }
        /* switch ( pANode->nMagic ) */
        break；
    }
}

/** AVL 树的删除操作函数
    @param AVLTREE *pTree——AVL 树指针
    @param void *pData——要删除的数据指针
    @param COMPAREFUNC CompareFunc——数据比较回调函数
    @param DESTROYFUNC DestroyFunc——数据释放回调函数
    @return INT——返回 CAPI_FAILED 表示失败；返回 CAPI_SUCCESS 表示成功
*/
INT AVLTree_Delete( AVLTREE *pTree, void *pData, COMPAREFUNC CompareFunc,
                    DESTROYFUNC DestroyFunc)
{
    if ( pTree->pRoot == NULL || pData == NULL
```

```
        || CompareFunc == NULL )
    {
        return CAPI_FAILED；
    }
    if ( AVLTreeNode_Delete(&(pTree->pRoot), pData, CompareFunc, DestroyFunc)
        != CAPI_NOT_FOUND )
    {
        pTree->uNodeCount－= 1；
    }
    return CAPI_SUCCESS；
}

/** AVL 树的删除节点函数
    @param AVLTREENODE **ppRoot——指向 AVL 树根节点指针的指针
    @param void *pData——要删除的数据
    @param COMPAREFUNC CompareFunc——数据比较回调函数
    @param DESTROYFUNC DestroyFunc——数据释放回调函数
    @return INT——返回 CAPI_FAILED 表示失败；返回 CAPI_SUCCESS 表示成功
*/
INT AVLTreeNode_Delete( AVLTREENODE **ppRoot, void *pData,
                        COMPAREFUNC CompareFunc,
                        DESTROYFUNC DestroyFunc)
{
    AVLTREENODE *pNode；
    AVLTREENODE *pANode；
    AVLTREENODE *pDelNode；
    void *pDelData；
    INT nRet = 0；
    pNode = *ppRoot；
    while ( pNode != NULL )
    {
        nRet = (*CompareFunc)(pNode->pData, pData)；
        if ( nRet < 0 )
        {
            pNode = pNode->pRight；
        }
        else if ( nRet > 0 )
        {
            pNode = pNode->pLeft；
```

```
        }
        else
        {
            break；
        }
    }
    if ( pNode == NULL )
    {
        return CAPI_NOT_FOUND；
    }
    pDelData = pNode->pData；
    if ( pNode->pLeft != NULL && pNode->pRight != NULL )
    {
        /* 处理查找到的 pNode 有两个子节点的情况 */
        pDelNode = pNode->pLeft；
        while (pDelNode->pRight != 0)
        {
            pDelNode = pDelNode->pRight；
        }
        pANode = pDelNode->pParent；
        pNode->pData = pDelNode->pData；
        if (pDelNode != pNode->pLeft)
        {
            pANode->pRight = pDelNode->pLeft；
        }
        else
        {
            pANode->pLeft = pDelNode->pLeft；
        }
        if ( pDelNode->pLeft != NULL )
        {
            pDelNode->pLeft->pParent = pANode；
        }
        if ( pDelNode == pANode->pLeft )
        {
            pANode->nMagic一= 1；
        }
        else
        {
```

```
                pANode->nMagic += 1；
            }
        }
        else
        {
            pANode = pNode；
            /* 处理最多只有一个子节点的情况 */
            if ( pNode->pLeft != NULL )
            {
                /* 只有左节点的情况 */
                pDelNode = pNode->pLeft；
                pNode->pData = pDelNode->pData；
                pNode->pLeft = NULL；
                pANode->nMagic－= 1；
            }
            else if ( pNode->pRight != NULL )
            {
                /* 只有右节点的情况 */
                pDelNode = pNode->pRight；
                pNode->pData = pDelNode->pData；
                pNode->pRight = NULL；
                pANode->nMagic += 1；
            }
            else
            {
                /* 处理删除节点的左右子节点都不存在的情况 */
                pANode = pNode->pParent；
                pDelNode = pNode；
                if ( pANode == NULL )
                {
                    *ppRoot = pANode；
                }
                else if ( pANode->pLeft == pNode )
                {
                    pANode->pLeft = NULL；
                    pANode->nMagic－= 1；
                }
                else
                {
```

```
                pANode->pRight = NULL；
                pANode->nMagic += 1；
            }
        }
    }
    /* 删除对应节点 */
    if ( pDelNode != NULL )
    {
        if ( DestroyFunc != NULL )
        {
            (*DestroyFunc)(pDelData)；
        }
        free(pDelNode)；
    }
    /* 调整平衡 */
    if ( pANode != NULL )
    {
        AVLTree_AdjustBalanceForDelete(ppRoot, pANode, pData, CompareFunc)；
    }
    return CAPI_SUCCESS；
}
```

为了提高效率，可以将 AVL **搜索树中需要进行的两次旋转操作合并成一个函数，如** AVLTree_RotateLeftRight()**和** AVLTree_RotateRightLeft ()**函数。**

5.5 红黑树

5.5.1 红黑树的基本概念

5.4 **节介绍的** AVL **搜索树，可以满足快速搜索的要求，但是** AVL **搜索树的插入和删除操作效率较低。从** AVL **搜索树删除过程来看，最坏的情况是不平衡节点到根节点路径上的每一个节点都可能要做旋转操作。软件设计中经常会碰到需要在树中频繁进行插入和删除的情况，当树中节点很多时，树的高度比较大，需要做的旋转操作次数将很可观，为了进一步提高效率，需要减少在插入删除过程中的旋转次数，提高插入和删除节点的效率。**

本节介绍的红黑树便是一种插入和删除效率比 AVL **搜索树高的特殊二叉排序树，红黑树是具有以下特点。**

① 红黑树中，每一个节点的颜色是黑色或红色；

② 红黑树中，从根节点到所有树梢节点的路径上不存在连续两个都是红色的节点；

③ 红黑树中，从根节点到所有树梢节点的路径上都具有相同数目的黑色节点。

满足上述 3 个条件的树便是红黑树，下面根据红黑树的定义来推导红黑树的高度。

假设从根节点到树梢节点路径上的黑色节点数量为 b，h 为红黑树的高度，N_h 表示一颗高度为 h 的红黑树的最小节点数。

首先，由于红黑树从根节点到树梢节点的路径上不存在连续两个都是红色的节点，因此从根节点到任一树梢节点的路径上，红色节点数量最多会比黑色节点数量多 1 个，因此有不等式 $h \leq 2b+1$。

由于红黑树的根节点到任意一个树梢节点的路径上的黑色节点个数为 b，则最小树梢高度为 b，由前面二叉树的树梢定理有以下不等式

$$n \geq 2^b - 1 \quad (n \text{ 为二叉树中的节点个数})$$

所以

$$b < \log_2(n+1)$$

$$h \leq 2b+1 \leq 2\log_2(n+1) + 1$$

所以红黑树查找的复杂度为 O(logn)。红黑树的最大高度 $2\log_2(n+1)$比 AVL 搜索树的最大高度(近似为 $1.44\ \log_2(n+2)$)要大一些，因此一般情况下红黑树的查找效率会比 AVL 搜索树的查找效率低。

图 5-24 便是一颗红黑树，粗圆圈表示节点颜色为黑色，细圆圈表示节点颜色为红色(下同)。从根节点到任一树梢节点的路径上都有两个黑色节点，整棵树中不存在连续的两个红色节点。

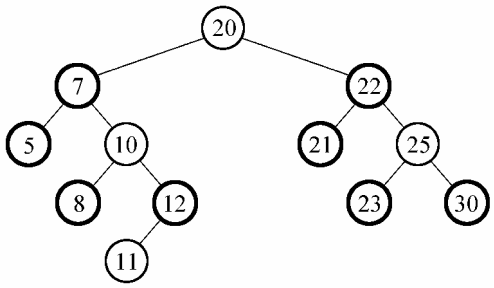


图 5-24　一个实际中的红黑树示意图

5.5.2　红黑树的插入操作

1. 插入操作基本思路

红黑树的插入操作可以利用普通二叉搜索树的插入算法将节点插入到对应位置上，然后判断平衡条件是否被破坏，若已破坏则通过一定操作，将树的平衡恢复。

将节点插入时，如果将插入节点设成黑色，则显然从根节点到插入节点这条路径上的黑色节点数量比其他路径上的黑色节点数量多出一个，违反了红黑树的平衡；如果将插入节点设成红色，则有可能出现连续两个节点都是红色的情况，如果没有出现连续两个节点是红色的情况则树仍然是平衡的，不需要调整。因此从插入的全局效率考虑，将插入节点设成黑色一定要重新调整平衡，而将插入节点设成红色可能不需要调整，因此红黑树插入节点时应将节点颜色初始化成红色，以提高效率。

当插入节点设为红色导致出现连续两个红色节点时，需要重新调整树的平衡，可以通过检查插入节点及其父节点和祖父节点(祖父节点为父节点的父节点)来确定不平衡类型。在图 5-25(a)中，设插入节点为 A 节点，它的父节点为 PA 节点，祖父节点为 GA 节点，由于 A 节点和 PA 节点都是红色，因此这两个节点一定存在，由于根节点是黑色，因此 PA 节点不会是根节点，PA 节点的父节点一定存在，即 GA 节点一定存在，且颜色是黑色的。

2. 几种不平衡类型

根据 A 节点、PA 节点、GA 节点的左右关系及 GA 节点的另外一个子节点的颜色情况可以将红黑树的不平衡类型分为以下八种。

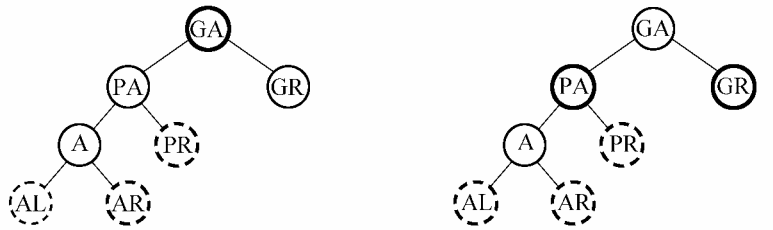
① LLr 型： PA 是 GA 的左节点，A 是 PA 的左节点，GA 的另一子节点为红色；
② LLb 型：PA 是 GA 的左节点，A 是 PA 的左节点，GA 的另一子节点为黑色；
③ LRr 型： PA 是 GA 的左节点，A 是 PA 的右节点，GA 的另一子节点为红色；
④ LRb 型：PA 是 GA 的左节点，A 是 PA 的右节点，GA 的另一子节点为黑色；
⑤ RLr 型： PA 是 GA 的右节点，A 是 PA 的左节点，GA 的另一子节点为红色；
⑥ RLb 型：PA 是 GA 的右节点，A 是 PA 的左节点，GA 的另一子节点为黑色；
⑦ RRr 型： PA 是 GA 的右节点，A 是 PA 的右节点，GA 的另一子节点为红色；
⑧ RRb 型：PA 是 GA 的右节点，A 是 PA 的右节点，GA 的另一子节点为黑色。

实际上 LLb 和 RRb、LLr 和 RRr、LRb 和 RLb、LRr 和 RLr 分别是对称的。因此下面只分析 LLr，LRr，LLb，LRb 四种不平衡情况，另四种情况可以很轻易地根据对称关系得出。

3. 不平衡情况的调整

1) LLr 型的不平衡情况调整如图 5-25 所示。

LLr 型调整需要将 PA 节点改成黑色，GA 节点改成红色，GA 节点的另一子节点 GR 节点改成黑色，则 GA 为根的子树中的各条路径上仍然满足红黑树的平衡条件，但由于 GA 节点改成了红色，有可能导致 GA 节点和 GA 节点的父节点都是红色的情况，因此需要对 GA 节点重新划分不平衡类型进行调整。

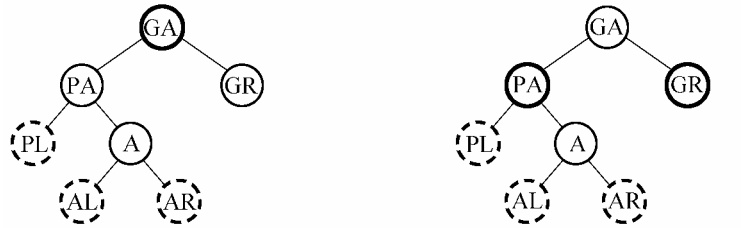


(a) LLr型不平衡图　　(b) 调整后的LLr型不平衡图

图 5-25　红黑树插入节点时的 LLr 型调整示意图

2) LRr **型不平衡情况调整如图** 5-26 **所示。**

LRr **型调整和** LLr **型调整基本一样，也是改变** PA、GA、GR **节点的颜色，再对** GA **节点重新划分不平衡类型进行调整。**



(a) LRr型不平衡图　　(b) 调整后的LRr型不平衡图

图 5-26　红黑树插入节点时的 LRr 型调整示意图

3) LLb **型的不平衡情况调整如图** 5-27 **所示。**

(a) LLb型不平衡图　　(b) 调整后的LLb型不平衡图

图 5-27　红黑树插入节点时的 LLb 型调整示意图

LLb **型调整需要做一次右旋操作，旋转后，将** PA **节点改成黑色，**GA **节点改成红色即可达到平衡。与** LLr **型不同，由于** PA **节点改成了黑色，因此不可能再出现违反红黑树平衡条件的情况，不需要对** PA **节点重新划分不平衡类型，一次旋转就完全达到平衡。**

4) LRb **型的不平衡情况调整如图** 5-28 **所示。**

(a) LRb型不平衡图　　(b) 调整后的LRb型不平衡图

图 5-28　红黑树插入节点时的 LRb 型调整示意图

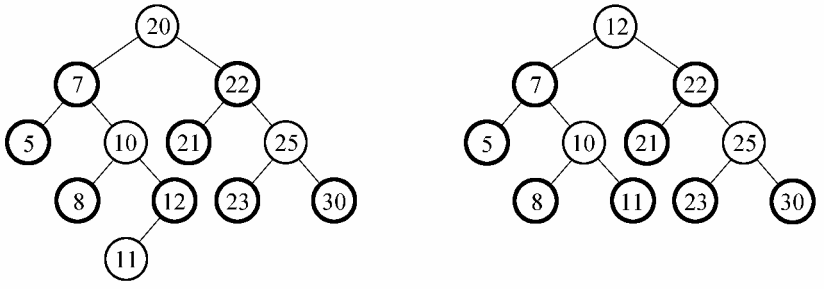
LRb 型的调整需要先对 PA 节点做左旋操作，再对 GA 节点做右旋操作，旋转完后，需要将 A 节点改成黑色，GA 节点改成红色即可达到平衡，和 LLb 型一样，由于 A 节点被改成了黑色，因此不需要再调整，一个双旋转就完成了树的平衡。

5.5.3　红黑树的删除操作

1. 删除操作基本思路

红黑树的删除操作和二叉排序树删除算法是一样的，可以用删除节点的左子树中的最大节点或右子树中的最小节点来替换删除节点，然后再进行颜色的调整使树重新达到平衡。假设用左子树中的最大节点来替换删除的节点，替换后，将替换节点颜色改成和删除节点颜色一样，那么除替换节点的替换前左子树有可能不平衡外，其他路径上仍然是平衡的。因此如果替换节点颜色为红色，则替换前的左子树仍然平衡；如果替换节点为黑色，则其替换前的左子树在替换后少了一个黑色节点，这时需要进行颜色调整使其达到新的平衡。因此，如果删除节点的左子树最大节点为红色就不需要调整颜色，为黑色时才需要调整颜色。

红黑树删除节点操作如图 5-29 所示(节点 20 的父节点省略掉没有画出来)。



(a) 删除节点20前的情况　　(b) 删除节点20后的情况

图 5-29　红黑树删除节点示意图

如果将节点 20 删除，使用节点 12 来替换删除的节点 20，将节点 12 的颜色改成和节点 20 一样的红色。显然在节点 30 和节点 8 的路径上黑色节点数目并没有改变，只有节点 11 这条路径上少了一个黑色节点，需要调整节点 11 的颜色，不妨把需要调整颜色的子树的根节点称为 A 节点。上图中显然 A 节点为节点 11，如果上图中没有节点 11，则 A 节点为空节点。

可以用以下方法来确定 A 节点。

1) 如果删除节点的左右节点都存在，删除时是用左子树的最大节点来替换删除节点的，此时删除节点的左子树最大节点的左节点为 A 节点(A 节点可能为空)；如果 A 节点刚好是删除节点的左节点，则 A 节点的父节点为删除节点的左节点本身，否则为删除节点的左子树最大节点的父节点。

2) 如果删除节点的左节点或右节点不存在，则 A 节点为删除节点的左节点或右节点，A 节点父节点为删除节点的父节点。

如果 A 节点为红色，则只需要将 A 节点改成黑色即可达到平衡，上图中只要将节点 11 改成黑色即可达到平衡。

如果 A 节点为黑色，需要分析进行删除操作后，A 节点和其父节点及父节点的另外一个子节点 B 之间的关系，来决定是否调整颜色使得树仍然满足红黑树的条件，可以分为以下四种基本情况。

① Lr 型，A 节点为其父节点的左节点，父节点另外一个子节点 B 的颜色是红色；

② Lb 型，A 节点为其父节点的左节点，父节点另外一个子节点 B 的颜色是黑色；

③ Rr 型，A 节点为其父节点的右节点，父节点另外一个子节点 B 的颜色是红色；

④ Rb 型，A 节点为其父节点的右节点，父节点另外一个子节点 B 的颜色是黑色。

实际上，Lr 型和 Rr 型是对称的，Lb 型和 Rb 型也是对称的，因此只要分析清楚 Lr 型和 Lb 型，则 Rr 型和 Rb 型可以根据对称关系很容易地分析出来，下面对 Lr 型和 Lb 型进行分析。

2. 红黑树删除的 Lr 型不平衡分析

Lr 型中，B 节点颜色为红色，因此 B 节点的两个子节点都为黑色，父节点也为黑色，否则会出现两个连续红色节点与红黑树定义相矛盾。由于 A 节点为黑色，且所有路径中黑色节点数量相等，因此 B 节点的两个子节点都存在，假设 B 节点的左子节点为 C 节点，由于 C 节点一定存在，考虑 C 节点的左右两个子节点的颜色有四种情况，分为以下四种类型。

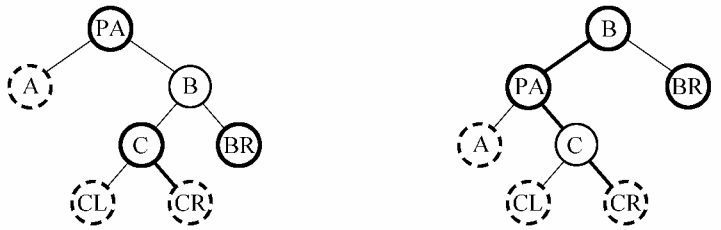
① Lr-bb 型，C 节点的左右子节点都是黑色；

② Lr-br 型，C 节点的左子节点为黑色，右子节点为红色；

③ Lr-rb 型，C 节点的左子节点为红色，右子节点为黑色；

④ Lr-rr 型，C 节点的两个子节点都为红色。

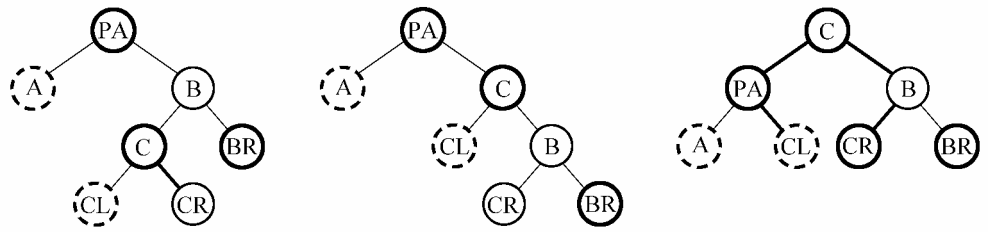
Lr-bb 型如图 5-30 所示(虚线圆圈表示节点可能为空)，只要执行一次左旋操作，然后将 B 节点改成黑色，C 节点改成红色即可达到平衡。图 5-30(b)中的粗直线表示两个节点间的父子关系相对旋转前改变了(后面的图也是用粗直线来表示父子关系改变)。



(a) Lr-bb型不平衡图　　(b) 进行左旋操作后的图

图 5-30　红黑树删除节点时的 Lr-bb 型不平衡调整示意图

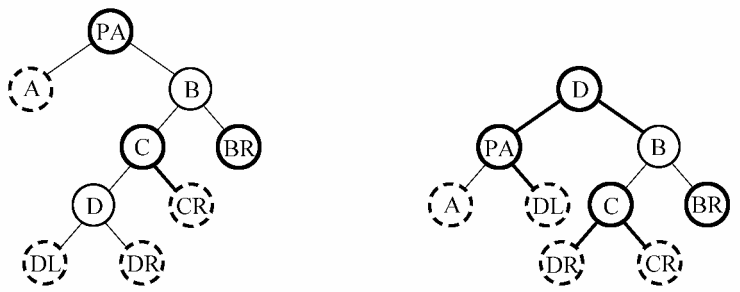
Lr-br 型如图 5-31 所示，需要对 B 节点进行右旋操作，再对 PA 节点进行左旋操作，再把 CR 节点由红色改成黑色即可达到平衡，如图 5-31(c)所示。



(a) Lr-br型不平衡图　　(b) 对B节点执行右旋操作　　(c) 对PA节点执行左旋操作

图 5-31　红黑树删除节点时的 Lr-br 型不平衡调整示意图

Lr-rb 型如图 5-32 所示，需要经过两次右旋和一次左旋操作，然后将节点 D 改成黑色即可达到平衡。三次旋转依次是先对 C 节点右旋，再对 B 节点右旋，然后对 PA 节点进行左旋操作。从图 5-32(b)中可以看出只有四对父子节点间的关系改变了，实际的改变量相当于两次旋转操作的改变量，因此在编码时可以将这三次旋转合并写成一个函数，效率比直接调用三次旋转操作函数高 50%。如果考虑函数调用的开销，合并成一个函数的实际效率可能要比调用三次旋转操作高出一倍。

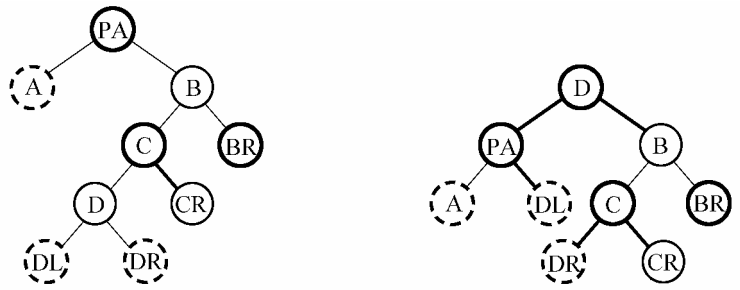


(a) Lr-rb型不平衡图　　(b) 两次右旋一次左旋后的图

图 5-32　红黑树删除节点时的 Lr-rb 型不平衡调整示意图

Lr-rr 型不平衡的处理方法和 Lr-rb 型完全一样，如图 5-33 所示，也是旋转后将 D 节点改成黑色即可。唯一不同的是 Lr-rr 型中 CR 节点本来是红色，而 Lr-rb 型中 CR 节点的颜色本来是黑色。

分析完 Lr 型之后，由于 Rr 型和 Lr 型对称，可以将 Rr 型也分为 Rr-bb、Rr-br、Rr-rb 和 Rr-rr 四种类型进行分析，其中 Rr-bb 和 Lr-bb、Rr-br 和 Lr-rb、Rr-rb 和 Lr-br、Rr-rr 和 Lr-rr 分别对称。



(a) Lr-rr型不平衡图　　(b) 调整后的Lr-rr型图

图 5-33　红黑树删除节点时的 Lr-rr 型不平衡调整示意图

3. 红黑树删除的 Lb 型不平衡分析

在 Lb 型中，仍然将 A 节点父节点的另外一个子节点称为 B 节点，由于 B 节点和 A 节点都是黑色，所以父节点 PA 有可能是红色也有可能是黑色，另外知道 A 节点这条路径上少了一个黑色节点，所以 B 节点一定存在，根据 B 节点的子节点颜色情况将 Lb 型分为以下四种类型。

① Lb-bb 型，B 节点的左右子节点都是黑色；

② Lb-br 型，B 节点的左子节点为黑色，右子节点为红色；

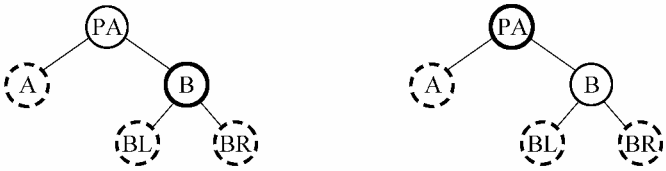
③ Lb-rb 型，B 节点的左子节点为红色，右子节点为黑色；

④ Lb-rr 型，B 节点的两个子节点都为红色。

同理，Rb 型也可以分为 Rb-bb、Rb-br、Rb-rb、Rb-rr 四种类型，其中 Rb-bb 和 Lb-bb、Rb-br 和 Lb-rb、Rb-rb 和 Lb-br、Rb-rr 和 Lb-rr 分别对称。

1) Lb-bb 型如图 5-34 所示，有 PA 节点为红色和黑色两种情况。

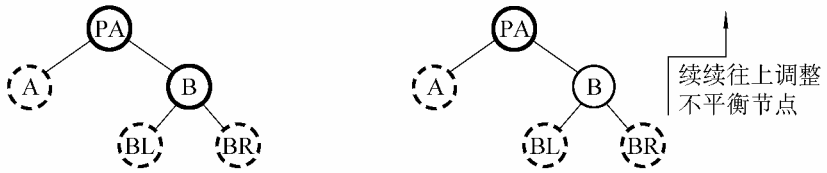
① 当 PA 节点为红色时，只要将 PA 节点改为黑色，再将 B 节点改成红色即可达到平衡，如图 5-34 所示。



(a) 调整前的Lb-bb型图　　(b) 调整后的Lb-bb型图

图 5-34　红黑树删除节点时的 Lb-bb 型不平衡调整示意图 1

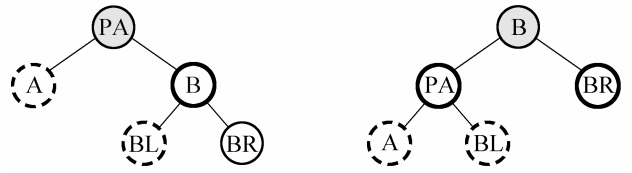
② 当 PA 节点为黑色时，如图 5-35 所示，只要将 B 节点的改成红色，这样 B 节点路径上也缺少了一个黑色节点，因此整个 PA 节点为根的子树上都缺少一个黑色节点，PA 节点变成了新的 A 节点。对新的 A 节点重新按前面所述的方法划分不平衡类型进行调整即可。此种情况下，最坏的调整次数可能等于根节点到达不平衡节点的路径长度。



(a) 调整前的Lb-bb型图　　(b) 调整后的Lb-bb型图

图 5-35　红黑树删除节点时的 Lb-bb 型不平衡调整示意图 2

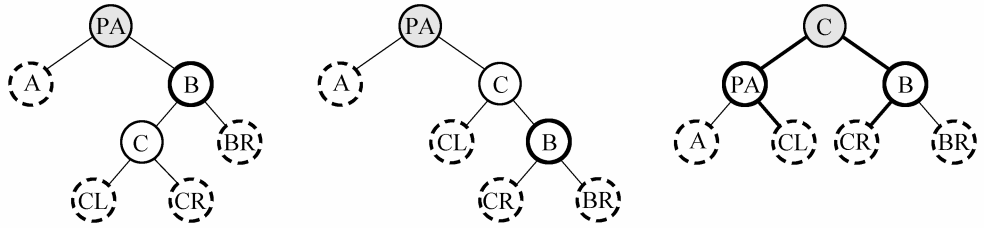
2) Lb-br 型如图 5-36 所示。PA 节点可能是红色，也可能是黑色(在图中用带阴影的图形表示)。对 PA 节点作一次左旋转后 B 节点颜色改成和 PA 节点一样，PA 节点改成黑色，BR 节点改成黑色。则从图中可以看出 A 节点路径上多了一个黑色节点，而 BL 和 BR 节点路径上的黑色节点数量保持不变，因此一次旋转即达到了平衡。



(a) 调整前的Lb-br型图　　(b) 调整后的Lb-br型图

图 5-36　红黑树删除节点时的 Lb-br 型不平衡调整示意图

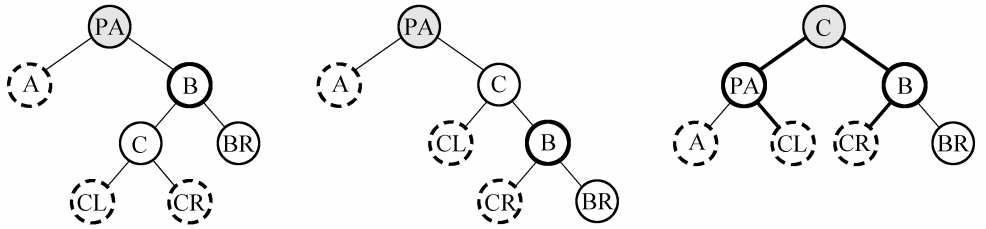
3) Lb-rb 型如图 5-37 所示。PA 节点可能是红色也可能是黑色，先对 B 节点作右旋操作，再对 PA 节点作左旋操作，旋转完后将 C 节点颜色改成和 PA 节点一样，PA 节点改成黑色。



(a) Lb-rb型不平衡图　　(b) 对B节点右旋后的图　　(c) 对PA节点左旋后的图

图 5-37 红黑树删除节点时的 Lb-rb 型不平衡调整示意图

4) Lb-rr 型同 Lb-rb 型的情况基本相同，不同之处仅在于 BR 节点的颜色，如图 5-38 所示。



(a) Lb-rr型不平衡图　　(b) 对B节点右旋后的图　　(c) 对PA节点左旋后的图

图 5-38　红黑树删除节点时的 Lb-rr 型不平衡调整示意图

5.5.4　红黑树的编码实现

红黑树的代码像 AVL 树一样也比较复杂，以下便是红黑树的编码实现，主要是实现红黑树的创建、释放、查找、插入、删除、逐个节点遍历等操作。为了增加可读性，

代码未作优化。

```
#define        RBTREE_COLOR_RED        0
#define        RBTREE_COLOR_BLACK      1

typedef    BINTREEBASENODE    RBTREENODE；

typedef struct RBTREE_st {
    RBTREENODE *pRoot；
    RBTREENODE *pCursor；
    UINT uNodeCount；
} RBTREE；

/** 红黑树的创建函数
    @param void——无
    @return RBTREE *——成功返回创建的红黑树指针；失败返回 NULL
*/
RBTREE *RBTree_Create(void)
{
    RBTREE *pTree；
    pTree = (RBTREE *)malloc(sizeof(RBTREE))；
    if ( pTree != NULL )
    {
        pTree->pRoot = NULL；
        pTree->uNodeCount = 0；
        pTree->pCursor = NULL；
    }
    return pTree；
}

/** 红黑树节点创建函数
    @param void *pData——节点数据指针
    @return static void *——返回创建的节点指针
*/
static void *RBTreeNode_Create(void *pData)
{
    RBTREENODE *pNewNode；
    pNewNode = (RBTREENODE *)malloc(sizeof(RBTREENODE))；
    if ( pNewNode != NULL )
    {
```

```
            pNewNode->pLeft= NULL；
            pNewNode->pRight = NULL；
            pNewNode->nMagic = RBTREE_COLOR_RED；
            pNewNode->pData = pData；
        }
        return (void *)pNewNode；
    }

    /** 红黑树的释放函数，释放以指定节点为根节点的子树
        @param RBTREENODE *pNode——要释放的根节点
        @param DESTROYFUNC DestroyFunc——数据释放回调函数
        @return static void——无
    */
    static void RBTreeNode_Destroy(RBTREENODE *pNode, DESTROYFUNC DestroyFunc)
    {
        if ( pNode != NULL )
        {
            if ( pNode->pLeft != NULL )
            {
                RBTreeNode_Destroy(pNode->pLeft, DestroyFunc)；
            }
            if ( pNode->pRight != NULL )
            {
                RBTreeNode_Destroy(pNode->pRight, DestroyFunc)；
            }
            if ( DestroyFunc != NULL && pNode->pData != NULL )
            {
                (*DestroyFunc)(pNode->pData)；
            }
            free(pNode)；
        }
    }

    /** 红黑树的释放函数
        @param RBTREE *pTree——红黑树指针
        @param DESTROYFUNC DestroyFunc——数据释放函数
        @return void——无
    */
    void RBTree_Destroy(RBTREE *pTree, DESTROYFUNC DestroyFunc)
```

```
{
    if ( pTree != NULL )
    {
        RBTreeNode_Destroy(pTree->pRoot, DestroyFunc)；
        free( pTree )；
    }
}

/** 红黑树的左旋操作，旋转前 A 节点和它的右节点 B 节点均存在，旋转后 A 节点的右
    指针指向 B 节点的左节点，B 节点的左指针指向 A 节点
    @param RBTREE *pTree——红黑树指针
    @param RBTREENODE *pANode——要旋转的节点
    @return static void——无
*/
static void RBTree_RotateLeft(RBTREE *pTree, RBTREENODE *pANode)
{
    RBTREENODE    *pBNode；/* B 节点指针 */
    pBNode = pANode->pRight；
    /* 将 B 节点的左节点变成 A 节点的右节点 */
    pANode->pRight = pBNode->pLeft；
    if ( pBNode->pLeft != NULL )
    {
        pBNode->pLeft->pParent = pANode；
    }
    /* 修改 A 节点的父节点指针和 B 节点的关系 */
    pBNode->pParent = pANode->pParent；
    if ( pANode == pTree->pRoot )
    {
        pTree->pRoot = pBNode；
    }
    else if ( pANode == pANode->pParent->pLeft )
    {
        pANode->pParent->pLeft = pBNode；
    }
    else
    {
        pANode->pParent->pRight = pBNode；
    }
    /* 将 A 节点变成 B 节点的左节点 */
```

```
    pBNode->pLeft = pANode；
    pANode->pParent = pBNode；
}

/** 红黑树的右旋操作，旋转前 A 节点和它的左节点 B 节点均存在，旋转后 A 节点
    的左指针指向 B 节点的右节点，B 节点的右指针指向 A 节点
    @param RBTREE *pTree——红黑树指针
    @param RBTREENODE *pANode——要旋转的节点
    @return static void——无
*/
static void RBTree_RotateRight(RBTREE *pTree, RBTREENODE *pANode)
{
    RBTREENODE *pBNode；/* B 节点指针 */
    pBNode = pANode->pLeft；
    /* 将 B 节点的右节点变成 A 节点的左节点 */
    pANode->pLeft = pBNode->pRight；
    if ( pBNode->pRight != NULL )
    {
        pBNode->pRight->pParent = pANode；
    }
    /* 修改 A 节点的父节点指针和 B 节点的关系 */
    pBNode->pParent = pANode->pParent；
    if ( pANode == pTree->pRoot )
    {
        pTree->pRoot = pBNode；
    }
    else if ( pANode == pANode->pParent->pRight )
    {
        pANode->pParent->pRight = pBNode；
    }
    else
    {
        pANode->pParent->pLeft = pBNode；
    }
    /* 将 A 节点变成 B 节点的左节点 */
    pBNode->pRight = pANode；
    pANode->pParent = pBNode；
}
```

```
/** 红黑树的为插入操作调整颜色函数
    @param RBTREE *pTree——红黑树指针
    @param RBTREENODE *pANode——插入的节点指针
    @return static void——无
*/
static void RBTree_AdjustColorForInsert(RBTREE *pTree, RBTREENODE *pANode)
{
    /* 根据红黑树的算法，需要先将插入的 A 节点颜色预设成红色 */
    pANode->nMagic = RBTREE_COLOR_RED；
    while ( pANode != pTree->pRoot
        && pANode->pParent->nMagic == RBTREE_COLOR_RED )
    {
        if ( pANode->pParent == pANode->pParent->pParent->pLeft )
        {
            /* 父节点为祖父节点左节点的情况 */
            RBTREENODE *pNode = pANode->pParent->pParent->pRight；
            if ( pNode != NULL && pNode->nMagic == RBTREE_COLOR_RED )
            {
                /* LLr 和 LRr 类型，直接修改颜色然后再继续循环调整祖父节点颜色 */
                pANode->pParent->nMagic = RBTREE_COLOR_BLACK；
                pNode->nMagic = RBTREE_COLOR_BLACK；
                pANode->pParent->pParent->nMagic = RBTREE_COLOR_RED；
                /* 由于祖父节点被改变成红色，因此祖父节点和它的父节点有可能
                * 都是红色，需要继续循环来调整祖父节点的平衡情况
                */
                pANode = pANode->pParent->pParent；
            }
            else
            {
                /* LLb 和 LRb 类型，直接通过旋转操作即可调整颜色 */
                if ( pANode == pANode->pParent->pRight )
                {
                    pANode = pANode->pParent；
                    RBTree_RotateLeft(pTree, pANode)；
                }
                pANode->pParent->nMagic = RBTREE_COLOR_BLACK；
                pANode->pParent->pParent->nMagic = RBTREE_COLOR_RED；
                RBTree_RotateRight(pTree, pANode->pParent->pParent)；
                /* 这里由于 A 节点的父节点是黑色的，实际上会退出循环，因此如果
```

```
                    *  要优化代码大小，break 语句可以省略
                    */
                    break；
                }
            }
            else
            {
                /* 父节点为祖父节点右节点的情况 */
                RBTREENODE *pNode = pANode->pParent->pParent->pLeft；
                if ( pNode != NULL && pNode->nMagic == RBTREE_COLOR_RED )
                {
                    /* RLr 和 RRr 类型，直接修改颜色后再继续循环调整祖父节点颜色 */
                    pANode->pParent->nMagic = RBTREE_COLOR_BLACK；
                    pNode->nMagic = RBTREE_COLOR_BLACK；
                    pANode->pParent->pParent->nMagic = RBTREE_COLOR_RED；
                    /* 由于祖父节点被改成红色，因此祖父节点和它的父节点有可能
                    *  都是红色，需要继续循环来调整祖父节点的平衡情况
                    */
                    pANode = pANode->pParent->pParent；
                }
                else
                {
                    /* RLb 和 RRb 类型，直接通过旋转操作即可调整颜色 */
                    if ( pANode == pANode->pParent->pLeft )
                    {
                        pANode = pANode->pParent；
                        RBTree_RotateRight(pTree, pANode)；
                    }
                    pANode->pParent->nMagic = RBTREE_COLOR_BLACK；
                    pANode->pParent->pParent->nMagic = RBTREE_COLOR_RED；
                    RBTree_RotateLeft(pTree, pANode->pParent->pParent)；
                    /* 这里由于 A 节点的父节点是黑色的，实际上会退出循环，因此如果
                    *  要优化代码大小, break 语句可以省略
                    */
                    break；
                }
            }
        }
```

```
    /* 在上面的调整过程中，根节点可能会被改成红色，因此需要将根节点重新赋为黑色
     */
    pTree->pRoot->nMagic = RBTREE_COLOR_BLACK；
}

/** 红黑树的插入函数
    @param RBTREE *pTree——红黑树指针
    @param void *pData——要插入的数据指针
    @param COMPAREFUNC CompareFunc——数据比较函数
    @return INT——返回 CAPI_FAILED 表示失败；返回 CAPI_SUCCESS 表示成功
*/
INT RBTree_Insert(RBTREE *pTree, void *pData, COMPAREFUNC CompareFunc)
{
    INT nRet；
    RBTREENODE *pNode；
    nRet = CAPI_FAILED；
    pNode = RBTreeNode_Create(pData)；
    if ( pNode != NULL )
    {
        nRet = RBTree_Inter_Insert(pTree, pNode, CompareFunc)；
    }
    return nRet；
}

/** 红黑树的插入函数，主要供需要继承的模块使用，一般的红黑树应用模块请不要调用
    这个函数
    @param RBTREE *pTree——红黑树指针
    @param void *pData——数据指针
    @param COMPAREFUNC CompareFunc——比较函数
    @param void *pTreeNode——插入的节点
    @return INT——返回 CAPI_SUCCESS 表示成功
*/
INT RBTree_Inter_Insert(RBTREE *pTree,  RBTREENODE *pNewNode,
                        COMPAREFUNC CompareFunc)
{
    RBTREENODE   *pNode；
    RBTREENODE   *pParentNode；
    INT nRet = 0；
    pParentNode = NULL；
```

```
        pNode = pTree->pRoot；
        while ( pNode != NULL )
        {
            pParentNode = pNode；
            nRet = (*CompareFunc)(pNode->pData, pNewNode->pData)；
            if ( nRet < 0 )
            {
                pNode = pNode->pRight；
            }
            else
            {
                pNode = pNode->pLeft；
            }
        }
        if ( pParentNode == NULL )
        {
            /* 树为空的情况 */
            pTree->pRoot = pNewNode；
            pNewNode->pParent = NULL；
        }
        else
        {
            if ( nRet < 0 )
            {
                pParentNode->pRight = pNewNode；
            }
            else
            {
                pParentNode->pLeft = pNewNode；
            }
            pNewNode->pParent = pParentNode；
        }
        pTree->uNodeCount += 1；
        RBTree_AdjustColorForInsert(pTree, pNewNode)；
        return CAPI_SUCCESS；
    }

    /** 红黑树的查找函数
        @param RBTREE *pTree——红黑树指针
```

```
    @param void *pData——要查找的数据指针
    @param COMPAREFUNC CompareFunc——数据比较回调函数
    @return void *——成功返回查找到的数据指针；失败返回 NULL
*/
void *RBTree_Find(RBTREE *pTree, void *pData, COMPAREFUNC CompareFunc)
{
    RBTREENODE *pNode；
    if ( pTree == NULL || pData == NULL || CompareFunc == NULL )
    {
        return NULL；
    }
    pNode = pTree->pRoot；
    while ( pNode != NULL )
    {
        INT nRet = (*CompareFunc)(pNode->pData, pData)；
        if ( nRet < 0 )
        {
            pNode = pNode->pRight；
        }
        else if ( nRet > 0 )
        {
            pNode = pNode->pLeft；
        }
        else
        {
            return pNode->pData；
        }
    }
    return NULL；
}

/** 红黑树删除操作的 Rr-rr 和 Rr-br 型旋转，这个函数的操作相当于将两次右旋和一次左旋
    操作合并到一个函数里，合并后的计算量相当于两次旋转的计算量。如果 A 节点为基
    准节点，则 B 节点为 A 节点的左节点，C 节点为 B 节点的右节点，先对 C 节点作左旋，
    再对 B 节点左旋，再对 A 节点进行右旋而成
    @param RBTREE *pTree——红黑树指针
    @param RBTREENODE *pStartNode——要旋转的基准节点
    @return void——无
*/
```

```
void RBTree_RotateLeftLeftRight(RBTREE *pTree, RBTREENODE *pStartNode)
{
    RBTREENODE *pParentNode；
    RBTREENODE *pBNode；
    RBTREENODE *pCNode；
    RBTREENODE *pDNode；
    pParentNode = pStartNode；
    pBNode = pParentNode->pLeft；
    pCNode = pBNode->pRight；
    pDNode = pCNode->pRight；
    if ( pParentNode->pParent == NULL )
    {
        pTree->pRoot = pDNode；
    }
    else if ( pParentNode->pParent->pLeft == pParentNode )
    {
        pParentNode->pParent->pLeft = pDNode；
    }
    else
    {
        pParentNode->pParent->pRight = pDNode；
    }
    pDNode->pParent = pParentNode->pParent；
    pCNode->pRight = pDNode->pLeft；
    if ( pDNode->pLeft != NULL )
    {
        pDNode->pLeft->pParent = pCNode；
    }
    pParentNode->pLeft = pDNode->pRight；
    if ( pDNode->pRight != NULL )
    {
        pDNode->pRight->pParent = pParentNode；
    }
    pDNode->pLeft = pBNode；
    pBNode->pParent = pDNode；
    pDNode->pRight = pParentNode；
    pParentNode->pParent = pDNode；
}
```

```
/** 红黑树删除操作的 Lr-rr 和 Lr-rb 型旋转，两次左旋和一次右旋操作合并到一个函数里
    如果 A 节点为基准节点，则 B 节点为 A 节点的右节点，C 节点为 B 节点的左节点先
    对 C 节点作右旋，再对 B 节点右旋，再对 A 节点进行左旋而成
    @param RBTREE *pTree——红黑树指针
    @param RBTREENODE *pStartNode——要旋转的基准节点
    @return void——无
*/
static void RBTree_RotateRightRightLeft(RBTREE *pTree, RBTREENODE *pStartNode)
{
    RBTREENODE *pParentNode；
    RBTREENODE *pBNode；
    RBTREENODE *pCNode；
    RBTREENODE *pDNode；
    pParentNode = pStartNode；
    pBNode = pParentNode->pRight；
    pCNode = pBNode->pLeft；
    pDNode = pCNode->pLeft；
    /* 修改使 D 节点成为 A 节点祖父节点的子节点，即 D 节点替代父节点位置 */
    if ( pParentNode->pParent == NULL )
    {
        pTree->pRoot = pDNode；
    }
    else if ( pParentNode->pParent->pLeft == pParentNode )
    {
        pParentNode->pParent->pLeft = pDNode；
    }
    else
    {
        pParentNode->pParent->pRight = pDNode；
    }
    pDNode->pParent = pParentNode->pParent；
    /* D 节点的右节点变成 C 节点的左节点 */
    pCNode->pLeft = pDNode->pRight；
    if ( pDNode->pRight != NULL )
    {
        pDNode->pRight->pParent = pCNode；
    }
    /* D 节点的左节点变成父节点的右节点 */
    pParentNode->pRight = pDNode->pLeft；
```

```
    if ( pDNode->pLeft != NULL )
    {
        pDNode->pLeft->pParent = pParentNode；
    }
    /* B 节点变成 D 节点的右节点 */
    pDNode->pRight = pBNode；
    pBNode->pParent = pDNode；
    /* 父节点变成 D 节点的左节点 */
    pDNode->pLeft = pParentNode；
    pParentNode->pParent = pDNode；
}

/** 红黑树的删除操作中对不平衡节点的调整操作
    @param RBTREE *pTree——红黑树指针
    @param RBTREENODE *pParentNode——要调整平衡节点的父节点
    @param RBTREENODE *pReplaceNode——要平衡的节点
    @return static void——无
*/
static void RBTree_AdjustColorForDelete( RBTREE *pTree, RBTREENODE *pParentNode,
                                         RBTREENODE *pReplaceNode)
{
    RBTREENODE *pANode；          /* 需要调整平衡的节点 */
    RBTREENODE *pAParentNode；    /* A 节点的父节点 */
    RBTREENODE *pBNode；          /* pAParentNode 的另一子节点 */
    RBTREENODE *pCNode；          /* 临时变量用来记录 B 节点的一个子节点 */
    pANode = pReplaceNode；
    pAParentNode = pParentNode；
    while ( pANode != pTree->pRoot
            && ( pANode == NULL || pANode->nMagic == RBTREE_COLOR_BLACK) )
    {
        if ( pANode == pAParentNode->pLeft )
        {
            pBNode = pAParentNode->pRight；
            if ( pBNode->nMagic == RBTREE_COLOR_RED )
            {
                /* Lr 型 */
                pCNode = pBNode->pLeft；
                if ( (pCNode->pLeft == NULL
                    || pCNode->pLeft->nMagic == RBTREE_COLOR_BLACK)
```

```
            && (pCNode->pRight == NULL
            || pCNode->pRight->nMagic == RBTREE_COLOR_BLACK) )
        {
            /* Lr-bb 型，即 C 节点左右节点都是黑色的情况 */
            pCNode->nMagic = RBTREE_COLOR_RED；
            pBNode->nMagic = RBTREE_COLOR_BLACK；
            RBTree_RotateLeft(pTree, pAParentNode)；
        }
        else if ( pCNode->pRight != NULL
            && pCNode->pRight->nMagic == RBTREE_COLOR_RED
            && ( pCNode->pLeft == NULL
            || pCNode->pLeft->nMagic == RBTREE_COLOR_BLACK) )
        {
            /* Lr-br 型，即 C 节点左节点为黑色，右节点为红色的情况 */
            pCNode->pRight->nMagic = RBTREE_COLOR_BLACK；
            RBTree_RotateRight(pTree, pBNode)；
            RBTree_RotateLeft(pTree, pAParentNode)；
        }
        else
        {
            /* Lr-rb 型和 Lr-rr 型的情况 */
            pCNode->pLeft->nMagic = RBTREE_COLOR_BLACK；
            RBTree_RotateRightRightLeft(pTree, pAParentNode)；
        }
    }
    else
    {
        /* Lb 型, Lb 型分为 Lb-bb 型, Lb-br 型, Lb-rb 型, Lb-rr 型四种情况 */
        if ( ( pBNode->pLeft == NULL
            || pBNode->pLeft->nMagic == RBTREE_COLOR_BLACK )
            && ( pBNode->pRight == NULL
            || pBNode->pRight->nMagic == RBTREE_COLOR_BLACK ) )
        {
            /* Lb-bb 型, B 节点的左右子节点都是黑色的情况 */
            if ( pAParentNode->nMagic == RBTREE_COLOR_BLACK )
            {
                /* 如果 A 节点的父节点为黑色，需要将 B 节点改成红色，然后
                *  将 A 节点的父节点变成 A 节点重新调整平衡
                */
```

```
                    pBNode->nMagic = RBTREE_COLOR_RED；
                    pANode = pAParentNode；
                    pAParentNode = pANode->pParent；
                    continue；
                }
                else
                {
                    /* A 节点的父节点为红色的情况，只需要将 A 节点改成黑
                    * 色，将 B 节点改成红色即可达到平衡
                    */
                    pAParentNode->nMagic = RBTREE_COLOR_BLACK；
                    pBNode->nMagic = RBTREE_COLOR_RED；
                }
            }
            else if ( pBNode->pRight != NULL
                && pBNode->pRight->nMagic == RBTREE_COLOR_RED
                && (pBNode->pLeft == NULL
                || pBNode->pLeft->nMagic == RBTREE_COLOR_BLACK ) )
            {
                /* Lb-br 型，即 B 节点的左子节点为黑色，右子节点为红色的情况*/
                pBNode->nMagic = pAParentNode->nMagic；
                pAParentNode->nMagic = RBTREE_COLOR_BLACK；
                pBNode->pRight->nMagic = RBTREE_COLOR_BLACK；
                RBTree_RotateLeft(pTree, pAParentNode)；
            }
            else
            {
                /* Lb-rb 型和 Lb-rr 型的情况, B 节点的左子节点为红色的情况，需要
                * 先对 B 节点进行右旋再对 A 节点的父节点进行左旋即可达到平衡
                */
                pBNode->pLeft->nMagic = pAParentNode->nMagic；
                pAParentNode->nMagic = RBTREE_COLOR_BLACK；
                RBTree_RotateRight(pTree, pBNode)；
                RBTree_RotateLeft(pTree, pAParentNode)；
            }
        }
    }
    else
    {
```

```
pBNode = pAParentNode->pLeft；
if ( pBNode->nMagic == RBTREE_COLOR_RED )
{
    /* 以下处理 Rr 型的不平衡情况，Rr 型不平衡处理分为四种情况
    * Rr-bb 型，Rr-br 型，Rr-rb 型，Rr-rr 型
    */
    pCNode = pBNode->pRight；
    if ( (pCNode->pLeft == NULL
        || pCNode->pLeft->nMagic == RBTREE_COLOR_BLACK)
        && (pCNode->pRight == NULL
        || pCNode->pRight->nMagic == RBTREE_COLOR_BLACK))
    {
        /* Rr-bb 型，C 节点的左右节点均为黑色，和 Lr-bb 对称 */
        pBNode->nMagic = RBTREE_COLOR_BLACK；
        pCNode->nMagic = RBTREE_COLOR_RED；
        RBTree_RotateRight(pTree, pAParentNode)；
    }
    else if ( pCNode->pRight == NULL
        || pCNode->pRight->nMagic == RBTREE_COLOR_BLACK )
    {
        /* Rr-rb 型，C 节点的左节点为红色，右节点为黑色，和 Lr-br 对称*/
        pCNode->pLeft->nMagic = RBTREE_COLOR_BLACK；
        RBTree_RotateLeft(pTree, pAParentNode->pLeft)；
        RBTree_RotateRight(pTree, pAParentNode)；
    }
    else
    {
        /* 有两种情况：
        * 1) Rr-br 型：C 节点的右节点为红色，左节点为黑色，和 Lr-rb 对称
        * 2) Rr-rr 型：C 节点的左右节点均为红色，和 Lr-rr 对称
        * 这两种情况的旋转操作是一样的，因此合并到一起
        */
        pCNode->pRight->nMagic = RBTREE_COLOR_BLACK；
        RBTree_RotateLeftLeftRight(pTree, pAParentNode)；
    }
}
else
{
    /* Rb 型, Rb 型分为 Rb-bb 型, Rb-br 型, Rb-rb 型, Rb-rr 型四种情况 */
```

```
if ( ( pBNode->pLeft == NULL
    || pBNode->pLeft->nMagic == RBTREE_COLOR_BLACK )
    && ( pBNode->pRight == NULL
    || pBNode->pRight->nMagic == RBTREE_COLOR_BLACK ) )
{
    /* Rb-bb 型, B 节点的左右子节点都是黑色的情况，和 Lb-bb 对称 */
    if ( pAParentNode->nMagic == RBTREE_COLOR_BLACK )
    {
        /* 如果 A 节点的父节点为黑色，需要将 B 节点改成红色，然后
        *  将 A 节点的父节点变成 A 节点重新调整平衡
        */
        pBNode->nMagic = RBTREE_COLOR_RED；
        pANode = pAParentNode；
        pAParentNode = pANode->pParent；
        continue；
    }
    else
    {
        /* A 节点的父节点为红色的情况，只需要将 A 节点改成黑
        * 色，将 B 节点改成红色即可达到平衡
        */
        pAParentNode->nMagic = RBTREE_COLOR_BLACK；
        pBNode->nMagic = RBTREE_COLOR_RED；
    }
}
else if ( pBNode->pLeft != NULL
    && pBNode->pLeft->nMagic == RBTREE_COLOR_RED
    && (pBNode->pRight == NULL
    || pBNode->pRight->nMagic == RBTREE_COLOR_BLACK ) )
{
    /* Rb-rb 型，即 B 节点的左子节点为红色，右子节点为黑色的
    * 情况，和 Lb-br 对称
    */
    pBNode->nMagic = pAParentNode->nMagic；
    pAParentNode->nMagic = RBTREE_COLOR_BLACK；
    pBNode->pLeft->nMagic = RBTREE_COLOR_BLACK；
    RBTree_RotateRight(pTree, pAParentNode)；
}
else
```

```
                {
                    /* Rb-br 型和 Rb-rr 型的情况，B 节点的左子节点为红色时，需要先
                    *  对 B 节点进行右旋再对 A 节点的父节点进行左旋即可达到平衡
                    */
                    pBNode->pRight->nMagic = pAParentNode->nMagic；
                    pAParentNode->nMagic = RBTREE_COLOR_BLACK；
                    RBTree_RotateLeft(pTree, pBNode)；
                    RBTree_RotateRight(pTree, pAParentNode)；
                }
            }
        }
        break；
    } /* while(...) */
    if ( pANode != NULL )
    {
        pANode->nMagic = RBTREE_COLOR_BLACK；
    }
}

/** 红黑树的删除函数
    @param RBTREE *pTree——红黑树指针
    @param void *pData——数据指针
    @param COMPAREFUNC CompareFunc——数据比较函数
    @param DESTROYFUNC DestroyFunc——数据释放函数
    @return INT——返回 CAPI_FAILED 表示失败；返回 CAPI_SUCCESS 表示成功
*/
INT RBTree_Delete( RBTREE *pTree, void *pData, COMPAREFUNC CompareFunc,
                   DESTROYFUNC DestroyFunc)
{
    RBTREENODE *pNode；
    RBTREENODE *pParentNode；
    RBTREENODE *pANode；
    RBTREENODE *pDelNode；
    RBTREENODE *pAParentNode；
    INT nRet = 0；
    if ( pTree == NULL || pData == NULL || CompareFunc == NULL )
    {
        return CAPI_FAILED；
    }
```

```
    pParentNode = NULL；
    pNode = pTree->pRoot；
    while ( pNode != NULL )
    {
        pParentNode = pNode；
        nRet = (*CompareFunc)(pNode->pData, pData)；
        if ( nRet < 0 )
        {
            pNode = pNode->pRight；
        }
        else if ( nRet > 0 )
        {
            pNode = pNode->pLeft；
        }
        else
        {
            break；
        }
    }
    if ( pNode == NULL )
    {
        /* 未找到指定节点，认为删除失败 */
        return CAPI_FAILED；
    }
    pDelNode = pNode；
    if ( pNode->pLeft != NULL && pNode->pRight != NULL )
    {
        INT nMagic；
        /* 处理查找到的 pNode 有两个子节点的情况 */
        pDelNode = pDelNode->pLeft；
        while (pDelNode->pRight != 0)
        {
            pDelNode = pDelNode->pRight；
        }
        pANode = pDelNode->pLeft；    /* pANode 可以是空节点 */
        pNode->pRight->pParent = pDelNode；
        pDelNode->pRight = pNode->pRight；
        if (pDelNode != pNode->pLeft)
        {
```

```
            pAParentNode = pDelNode->pParent；
            if (pANode != NULL)
            {
                pANode->pParent = pDelNode->pParent；
            }
            pDelNode->pParent->pRight = pANode；
            pDelNode->pLeft = pNode->pLeft；
            pNode->pLeft->pParent = pDelNode；
        }
        else
        {
            pAParentNode = pDelNode；
        }
        if (pTree->pRoot == pNode)
        {
            pTree->pRoot = pDelNode；
        }
        else if (pNode->pParent->pLeft == pNode)
        {
            pNode->pParent->pLeft = pDelNode；
        }
        else
        {
            pNode->pParent->pRight = pDelNode；
        }
        pDelNode->pParent = pNode->pParent；
        nMagic = pDelNode->nMagic；
        pDelNode->nMagic = pNode->nMagic；
        pNode->nMagic = nMagic；
        pDelNode = pNode；
    }
    else
    {
        /* 处理最多只有一个子节点的情况 */
        if ( pNode->pLeft != NULL )
        {
            pANode = pNode->pLeft；
        }
        else
```

```
        {
            pANode = pNode->pRight；
        }
        pAParentNode = pDelNode->pParent；
        if ( pANode != NULL )
        {
            pANode->pParent = pDelNode->pParent；
        }
        if ( pTree->pRoot == pNode )
        {
            pTree->pRoot = pANode；
        }
        else
        {
            if ( pNode->pParent->pLeft == pNode )
            {
                pNode->pParent->pLeft = pANode；
            }
            else
            {
                pNode->pParent->pRight = pANode；
            }
        }
    }
    if ( pDelNode->nMagic != RBTREE_COLOR_RED )
    {
        RBTree_AdjustColorForDelete(pTree, pAParentNode, pANode)；
        if ( pTree->pCursor == pDelNode )
        {
            RBTree_EnumNext(pTree)；
        }
        if ( DestroyFunc != NULL )
        {
            (*DestroyFunc)(pDelNode->pData)；
        }
        free(pDelNode)；
    }
    /* 将节点数量减 1 */
    pTree->uNodeCount－= 1；
```

```
    return CAPI_SUCCESS；
}

/** 红黑树的枚举开始函数
    @param RBTREE *pTree——红黑树指针
    @return void——无
*/
void RBTree_EnumBegin(RBTREE *pTree)
{
    RBTREENODE *pCursor；
    pCursor = pTree->pRoot；
    if ( pCursor != NULL )
    {
        while ( pCursor->pLeft != NULL )
        {
            pCursor = pCursor->pLeft；
        }
    }
    pTree->pCursor = pCursor；
}

/** 红黑树的枚举下一个元素函数，按照中序遍历顺序依次获取下一个元素
    @param RBTREE *pTree——红黑树指针
    @return void *——返回获取的下一个元素；如果下一个元素不存在则返回空
*/
void *RBTree_EnumNext(RBTREE *pTree)
{
    void *pData；
    RBTREENODE *pCursor；
    if ( pTree->pCursor == NULL )
    {
        return NULL；
    }
    pCursor = pTree->pCursor；
    pData = pCursor->pData；
    if ( pCursor->pRight != NULL )
    {
        pCursor = pCursor->pRight；
        while ( pCursor->pLeft != NULL )
        {
```

```
                pCursor = pCursor->pLeft；
            }
        }
        else
        {
            RBTREENODE *pParent = pCursor->pParent；
            while ( pParent != NULL && pParent->pRight == pCursor )
            {
                pCursor = pParent；
                pParent = pParent->pParent；
            }
            pCursor = pParent；
        }
        pTree->pCursor = pCursor；
        return pData；
    }

    /** 红黑树的获取指定节点为根节点子树的最小元素函数
        @param RBTREE *pTree——红黑树指针
        @param RBTREENODE *pNode——子树的根节点指针
        @return void *——返回获取的最小节点数据指针
    */
    void *RBTree_GetMinium(RBTREE *pTree, RBTREENODE *pNode)
    {
        RBTREENODE *pTempNode；
        pTempNode = pNode；
        while ( pTempNode->pLeft != NULL )
        {
            pTempNode = pTempNode->pLeft；
        }
        return pTempNode->pData；
    }

    /** 红黑树的获取指定节点为根节点子树的最大元素函数
        @param RBTREE *pTree——红黑树指针
        @param RBTREENODE *pNode——子树的根节点指针
        @return void *——返回获取的最大节点数据指针
    */
    void *RBTree_GetMaxium(RBTREE *pTree, RBTREENODE *pNode)
```

```
{
    RBTREENODE *pTempNode；
    pTempNode = pNode；
    while ( pTempNode->pRight != NULL )
    {
        pTempNode = pTempNode->pRight；
    }
    return pTempNode->pData；
}
```

5.6 实例：搜索引擎的实现

5.6.1 搜索引擎的实现思路和方法

搜索方面最常见的应用莫过于搜索引擎了，许多人几乎每天都要使用搜索引擎到 Internet 上去查找所需的信息，那么搜索引擎是如何实现快速搜索的呢？相信许多读者对这个问题会感兴趣。当然商业级的搜索引擎牵涉到的技术知识非常多，不是本书所能一一介绍的，因此本书只起一个抛砖引玉的作用，在这里介绍一个简易的搜索引擎的实现。

输入关键词时，搜索引擎会搜索出包含此关键词的所有网页，实现这个功能的最笨的方法便是将所有网页全部查找一遍，这种方法在实际中是不可能用的，因为耗时实在太多了。

如果能建立一个关键词词典，在这个词典中每个关键词对应一个含有这个关键词的所有网页列表，这样当用户输入要搜索的关键词时，可以通过查找关键词词典快速搜索出所需要的结果。

要建立上面所说的关键词词典，首先要解决关键词的来源问题，通常的做法是搜索所有的网页，自动生成关键词。下面还是用一个简单实例来讲解搜索词典建立的过程，比如有三个网页 File1.htm、File2.htm、File3.htm，各自包含的内容如下。

File1.htm：How are you？

File2.htm：Where are you？

File3.htm：Where is it？

首先搜索第一个文件 File1.htm，得到三个关键词：How、are、you，将关键词都放在 AVL 树中，每个关键词都对应着 File1.htm，如图 5-39 所示。

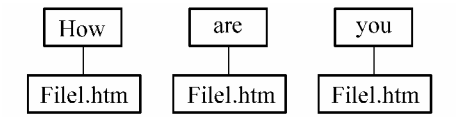


图 5-39　分词过程图 1

再搜索 File2.htm，也有三个关键词：Where、are、you，这样关键词和文件的对应关系如图 5-40 所示。

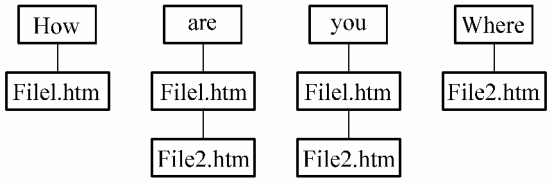


图 5-40　分词过程图 2

继续搜索 File3.htm，有三个关键词：Where、is、it，得到关键词和文件的对应关系如图 5-41 所示。

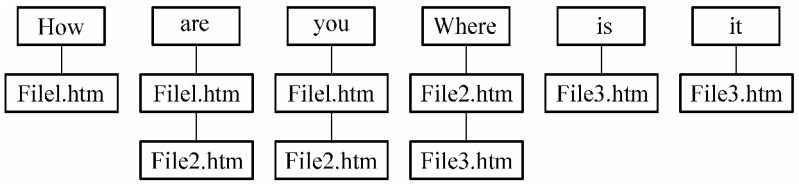


图 5-41　分词过程图 3

在搜索时，输入关键词 you，可以在 AVL 树中查到包含 you 的节点，发现对应的文件有 File1.htm 和 File2.htm。也可以实现模糊搜索，即输入的关键词在 AVL 树中不存在时，可以找一个最接近的关键词，如用户输入“ wher ”时自动搜索“ Where ”。

由上面的分析可以看出，搜索引擎的实现分成两个步骤。

步骤 1：将文件的内容分割成单个的单词。

步骤 2：将单词和对应的文件配对加入到一个搜索词典中，搜索词典使用 AVL 树来实现，也可以使用红黑树等其他类似的数据结构容器来实现。

步骤 2 的实现使用 AVL 树，AVL 树的实现本书前面已经有过介绍，在实际情况中，数据量非常大，需要考虑存放在硬盘中。

步骤 1 的实现与自然语言有关，不同语言分词的方法有所区别，如英文，单个单词的分割只需要通过空格、TAB 符、标点符号等便可以实现，而中文由于所有单词间没有任何分隔符号，分词时便复杂很多。在英文中如果要划分词组，和中文的分词是

一模一样的，分词的整个过程如图 5-42 所示。

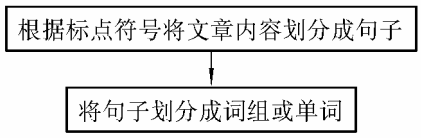


图 5-42　分词过程示意图

将文章内容划分成句子比较容易，将句子划分成词组或单词时相对来说复杂一些，下面给出一种简易的基于词典搜索的分词算法。

若要将句子“将文章内容划分成句子”分割成单词，可以先在词典中查询整个句子，如不存在，去掉最右边一个字，继续查询“将文章内容划分成句”；如果词典中不存在，再去掉最右边一个字，继续查询；到最后只剩一个“将”字时，不再查询了，得到第一个词“将”。

再将句子中已经得到的词“将”去掉，对剩下的“文章内容划分成句子”重复前面的算法，便可以将句子中的单词或词组一个个找出来。

按照这种方法进行单词或词组分割，如果一个句子中有 N 个单词或词组，最坏情况需要进行 N×(N－1)/2 次词典查询。

当搜索词典建好后，搜索就非常容易了，只需在搜索词典中查询输入的关键词即可。

5.6.2　搜索引擎的时间效率和空间效率分析

上面介绍了搜索引擎的实现思路和方法，读者可能会关心按这个思路和方法实现的搜索引擎在实践中是否有实用价值、运行速度是否会太慢或消耗过多的内存空间而使得它在实际中无法使用，下面就对搜索引擎的时间效率和空间效率进行分析。

1. 时间效率分析

从上述搜索引擎的实现可以看出，需要运行时间主要有两个地方：一个是分词；一个是搜索。假设有 100 万个关键词，用 AVL 树进行搜索，最多需要做不超过 29 次关键词的比较就可以找到，所以效率非常高。这个时间相对于网络数据传送所消耗的时间来讲，可以忽略不计。

由于分词可以在后台进行，不需要和搜索服务器放在一台机器上，所以分词的效率不会影响到搜索引擎的搜索效率。但是，还是需要考虑一台服务器一天可以对多少网页进行分词，分词所消耗的时间很大一部分都是在词典查询上。以一个 4 K 左右大小的网页文件为例，假设其中句子平均长度为 10 个单词，共有大约 70 个句子，那么

这个网页文件的分词需要进行的词典查询次数大约为 3 150 次。找一台每秒可以处理 10 万次词典查询的机器(普通 PC 就可以)，那么每秒可以处理 30 个文件；一小时就可以处理 10 万多个文件；一天就可以处理 240 万个文件；一年可以处理 8.76 亿网页文件。放上几十台这样的机器，基本上就可以将世界上大部分网站的网页文件都处理完。所以分词方面的效率也是很高的，在实际中完全可行。

2. 空间效率分析

分析完时间效率后，再来分析一下搜索引擎的空间效率。主要消耗内存空间的是搜索关键词词典。以 10^3 万个关键词，10^6 万个网页文件为例来分析一下搜索关键词词典所需要消耗的空间。

假设平均每个网页有 500 个不重复的单词，那么在关键词词典中，每个网页出现的平均次数为 500 次；所有网页在关键词词典中出现总次数为：500×10^6 万次。每个关键词所对应的网页文件平均个数为：500×10^6 万/10^3 万＝50 万个。

如果每个网页文件名平均需要消耗 32B，那么一个关键词对应的网页文件列表就需要消耗：32 B×50 万＝16 MB。

10^3 万个关键词的词典总共需要消耗：10^3 万×16 MB＝16 万 GB。

16 万 GB，这显然是一个天文数字，现实中的机器不可能有这么大的内存，必须想办法压缩这个词典。可以将网页文件名进行压缩，可以对每个网页文件名按顺序进行编号，从 1 开始编号，编号依次加 1，这样在关键词词典中只需要保存编号就可以了。然后再建一张网页文件名和编号的表，这个表用数组就可以了，编号就是数组的下标。由于 10^6 万超过了 32 位整数的范围，因此需要使用 64 位整数来表示编号，编号需要消耗 8 B 的空间。通过编号就可以直接从数组中取到网页文件名，这样整个消耗的空间由以下两部分组成。

其一，网页文件名和编号表消耗的空间为：10^6 万×(32＋8) = 400 GB。

其二，关键词词典消耗的空间为：16 万 GB×8/32 = 4 万 GB。

这样需要消耗的空间变成 4.04 万 GB，显然这个空间还是太大了，在目前的机器中是无法实现的，所以还需要想办法继续进行优化。

10^3 万个单词显然并不都是常用的关键词，常用的关键词也许不到 1 万个。因此可以将大部分内容保存到硬盘上，只将常用的关键词放到内存中。另外每个关键词关联的网页 50 万个，用户是不可能将 50 万个网页文件都看一遍的，只需要将前面 1 000 个或更少的网页文件名放到内存中，这样需要放到内存中的关键词词典就大大减小了。比如说放 10 万个关键词，每个关键词放 1 000 个网页文件名，需要消耗的内存空间为：10 万×1 000×8＝800 MB。

800 MB，现在普通的 PC 机都有这么大的内存，对于商业服务器更是不成问题了。

当然 10^3 万个关键词和 10^6 万个网页文件名，只有顶级的搜索引擎商才会有这么大的规模。对普通的企业，如果要构造一个内部的搜索引擎，10 万个关键词，10 万个网页已经绰绰有余。这种情况下关键词词典需要消耗的空间就非常少了，将关键词词典全部放到内存中，普通的 PC 机就可以负担得起。

5.6.3 高级搜索的实现

1. 高级搜索的概念

在很多情况下并不能简单地通过一个关键词就搜索到结果，最经常使用的是在搜索到的结果中继续搜索。前面介绍过搜索关键词时只需要在关键词词典中查找，但找到结果后，如何在结果中继续搜索出含另外一个关键词的新的结果呢？当然不可能逐个查找第一次搜索到的所有文件，看哪些文件含有第二个要搜索的关键词，因为这样做效率太低了。

要在第一个关键词的结果中搜索含有第二个关键词的文件，实际上就是搜索同时含有两个关键词的文件，即含有第一、二个关键词的文件的交集，如图 5-43 所示。

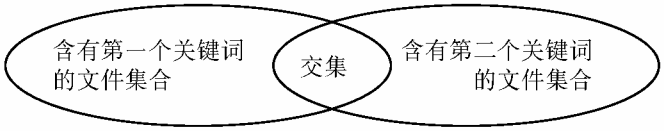


图 5-43 搜索含有两个关键词的文件

2. 高级搜索的算法实现

求包含两个关键词的文件的交集也就是求第一、二个关键词在关键词词典中的所对应的文件的相同部分。只要将关键词词典中的每个关键词所对应的文件编号按大小顺序排好，比较起来速度就非常快。可以用一个简单的比较算法来实现这个过程。

例 5-1 假设与两个关键词 apple 和 banana 关联的文件编号如图 5-44 所示。

其中：apple 关联的文件编号为 1，2，4，7，9，10，11，13；banana 关联的文件编号为 2，3，4，6，9，12，13，14，试找出其中编号相同的文件。

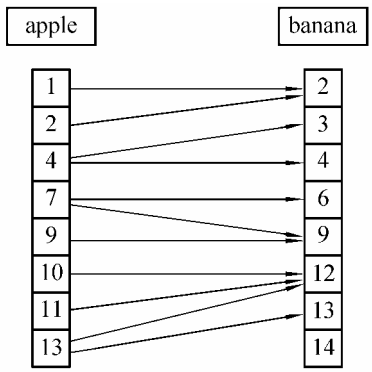


图 5-44　求包含两个关键词的文件之交集举例

解　比较时先将 apple 中 1 和 banana 中的 2 比较，1 比 2 小，取 apple 中 1 的下一个编号 2 继续和 banana 中的 2 比较，发现相等，然后取 apple 中 2 的下一个编号 4 和 banana 中 2 的下一个编号 3 进行比较 ;如此一直比较下去,如图 5-43 中的箭头线所示。最后发现有 2，4，9，13 四个编号相同，因此，2，4，9，13 是关键词 apple 和 banana 关联文件的交集。

3. 高级搜索比较算法的复杂度分析

从例 5-1 中可以看出，每比较一次，如果不相等，较小的那个编号不再进行比较；如果两个相等，则两个都不用再进行比较。每比较一次，至少有一个编号不用再进行比较，所以最多需要比较 N1+N2 次，N1、N2 为第一、二个关键词关联文件的个数。

有了这个在结果中再搜索的比较算法后，其他的高级搜索实现起来也都类似，这里就不再介绍了。

本 章 小 结

本章主要讲解了普通树和二叉树的基本概念和操作。在二叉树的基本概念中介绍了树梢的概念，另外重点介绍了实际应用最广泛的二叉排序树中的 AVL 树和红黑树。通过这些内容的学习，读者应该掌握普通树的设计方法，二叉树的设计和分析方法。最后本章还介绍了一个实现搜索引擎的实例，虽然没有给出实现编码，但基本思路和方法已全部给出，读者可以利用这些思路和方法自行去实现一个搜索引擎。

习题与思考

1. 证明 AVL 树中从根节点到任一树梢节点的路径上不存在连续三个树梢节点。

2. 证明红黑树中从根节点到任一树梢节点的路径上不存在连续三个树梢节点。
3. 使用非递归算法编码实现二叉树的前序遍历函数。
4. 编码实现二叉树的宽度遍历函数。
5. 使用红黑树替代哈希表去实现前面介绍的 WebServer CACHE 文件管理功能。
6. 比较一下哈希表、AVL 树、红黑树在各种操作效率方面的区别。
7. 估算一下对一个 5 000 单词的文件进行分词大约需要多少次词典查询。
8. 某企业计划构建一个内部搜索引擎，假设总共有 10 万个关键词、10 万个网页文件，估算一下搜索关键词词典大约要占多少内存空间。
9. 如何在搜索引擎关键词词典中搜索包含某个关键词但不包含另外一个关键词的结果？
10. 编码实现一个简易的搜索引擎功能。要求对给定目录下的网页文件输出关键词词典，对给定的关键词要求能输出包含这个关键词的所有文件。

6 复合二叉树

本章介绍了复合数据结构哈希红黑树及哈希 AVL 树，这两种复合数据结构采用了不同的设计方法；进而介绍了复合数据结构的分类；最后介绍了一个使用哈希 AVL 树实现的抗 DoS/DDoS 攻击的实例。

6.1 哈希红黑树

红黑树的查找比 AVL 树要慢，而当数据量比较大时，AVL 树的查找比哈希表要慢，但哈希表无法实现按顺序输出。在本书第 4 章哈希表中已经介绍了一个复合数据结构的哈希链表，可以按顺序输出，但缺点是必须要对数据排序。对静态的数据排序是没有问题的，如果有数据动态插入则不好用了。因此，用前面介绍的数据结构都不能快速地实现查找、插入、删除等操作。这里再介绍一个复合数据结构哈希红黑树。顾名思义，哈希红黑树既具有哈希表的特性又具有红黑树的特性，它可以充分利用哈希表的快速查找特性，同时又可以利用红黑树的按顺序输出，还可以利用红黑树满足模糊查找的功能。

6.1.1 哈希红黑树的基本概念

哈希红黑树要满足红黑树的特性，因此它的节点里必须包含红黑树节点的各种属性，同时又要满足哈希表的特性，所以它的节点里还要具备哈希表节点的特性。可以简单地用一个 C 语言结构体来描述哈希红黑树的节点结构。

```
typedef struct HASHRBTREENODE_st {
    RBTREENODE TreeNode；              /* 红黑树节点属性 */
```

```
    struct HASHRBTREENODE_st *pNext；  /*  哈希表中的链接指针  */
} HASHRBTREENODE；
```

从上面结构体可以看出，哈希红黑树的节点中包含 Tree Node 和 pNext 两个成员，TreeNode 用来记录红黑树的左右节点、颜色等属性，pNext 用来记录哈希表中的链接。

图 6-1 便是一个哈希红黑树的示意图。

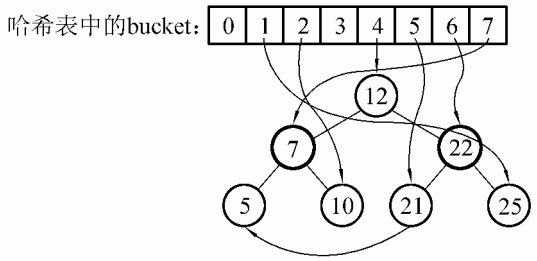


图 6-1　哈希红黑树示意图

在图 6-1 中如果删除了红黑树的左右指针的链接，便成了一个哈希表，如图 6-2 所示。

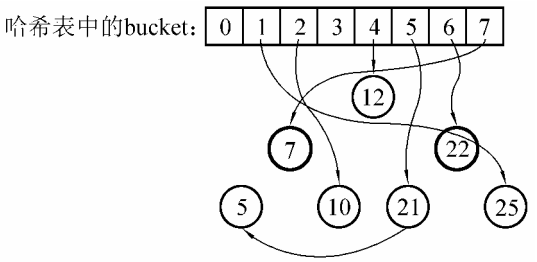


图 6-2　删除红黑树链接的哈希红黑树示意图

如果删除了哈希表的链接，哈希红黑树便变成了一棵红黑树，如图 6-3 所示。

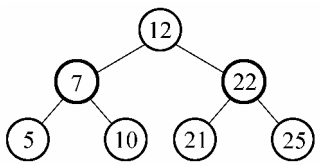


图 6-3　删除哈希表链接的哈希红黑树示意图

可见哈希红黑树既有哈希表的特性又有红黑树的特性，是两者的复合体。

6.1.2 哈希红黑树的查找

哈希红黑树的查找既可以采用哈希表的查找方法，也可以采用红黑树的查找方法。

1. 按哈希表的查找方法进行查找

比如要查找节点 5，按哈希表查找，先算出 bucket 位置为 5，从位置 5 查找下去，指向的第 1 个位置为节点 21，不是节点 5；继续查节点 21 的下一个，为节点 5，经过两次查找得到对应的查找结果，查找路径如图 6-4 中粗线所示。

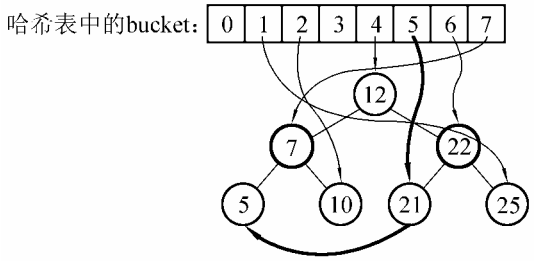


图 6-4 哈希红黑树按哈希表查找方法查找示意图

2. 按红黑树的查找方法进行查找

比如要查找节点 5，先从根节点 12 开始查找，12 比 5 大，往节点 12 的左边找，找到的节点为 7，7 比 5 大；继续往节点 7 的左边查找，找到了节点 5，查找路径如图 6-5 中的粗线所示。

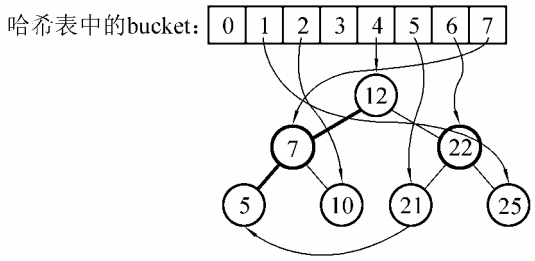


图 6-5 哈希红黑树按红黑树查找方法查找示意图

哈希红黑树的查找比较灵活，一般按哈希表查找的速度比按红黑树查找的速度快。如果要查找到某个节点后按顺序输出一定数量的节点，那么就可以先按哈希表查找方法找到对应的节点，再按红黑树的中序遍历便可实现从这个节点开始按顺序输出。

此外，当需要进行模糊查找时，如需要找到一个和 9 最接近的节点，便可以按红黑树的查找方法进行查找，找出最接近的节点为 10。

6.1.3 哈希红黑树的插入

例 6-1　假设哈希表的 bucket 个数为 8，试将 12，7，22，5，10，21，25 这几个数值构造成哈希红黑树。

解　步骤 1：在哈希红黑树中插入节点 12，这时红黑树中只有一个根节点 12，且为红色，由于 12 除以 8 余 4，因此哈希表 bucket 中的第 4 个位置指向节点 12，如图 6-6 所示。

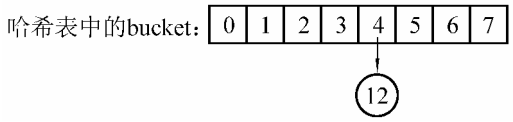


图 6-6　哈希红黑树插入示意图 1

步骤 2：在哈希红黑树中插入节点 7，节点 7 插在节点 12 的左边，此时节点 12 调整为黑色，哈希表 bucket 中的第 7 个位置指向节点 7，如图 6-7 所示。

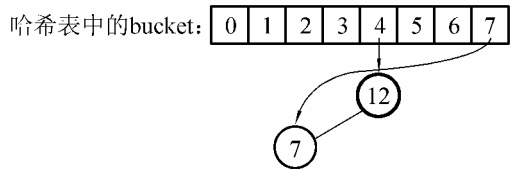


图 6-7　哈希红黑树插入示意图 2

步骤 3：在哈希红黑树中插入节点 22，节点 22 插在节点 12 的右边，节点 22 颜色为红色，哈希表 bucket 中的第 6 个位置指向节点 22，如图 6-8 所示。

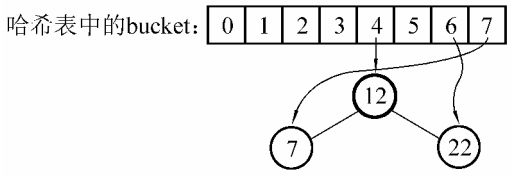


图 6-8　哈希红黑树插入示意图 3

步骤 4：在哈希红黑树中插入节点 5，节点 5 插在节点 7 的左边，此时需要调整红黑树中节点的颜色，哈希表 bucket 中的第 5 个位置指向节点 5，如图 6-9 所示。

步骤 5：在哈希红黑树中插入节点 10，节点 10 插在节点 7 的右边，哈希表 bucket 中的第 2 个位置指向节点 10，如图 6-10 所示。

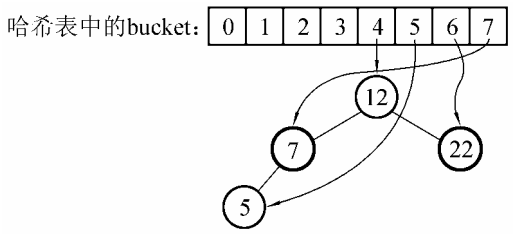


图 6-9　哈希红黑树插入示意图 4

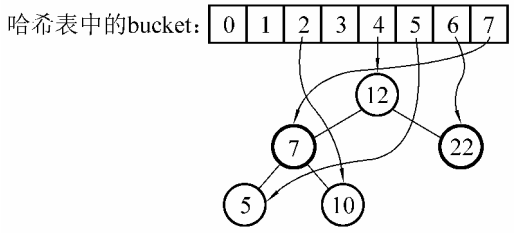


图 6-10　哈希红黑树插入示意图 5

步骤 6 在哈希红黑树中插入节点 21，节点 21 插在 22 的左边，此时需要将哈希表 bucket 中的第 5 个位置指向节点 21，然后将节点 21 的 pNext 指针指向节点 5，如图 6-11 所示。

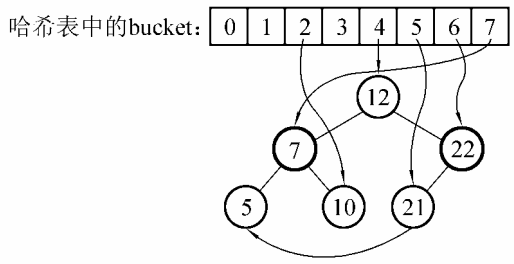


图 6-11　哈希红黑树插入示意图 6

步骤 7 在哈希红黑树中插入节点 25，节点 25 插在节点 22 的右边，将哈希表 bucket 中的第 1 个位置指向节点 25，如图 6-12 所示。

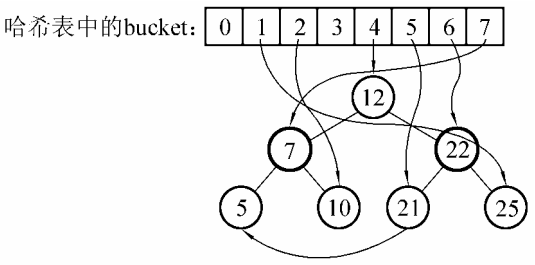


图 6-12　哈希红黑树插入示意图 7

至此，就完成了哈希红黑树的插入和构造过程。

6.1.4 哈希红黑树的删除

下面通过删除图 6-12 中哈希红黑树的节点 21，来介绍哈希红黑树的删除过程。哈希红黑树的删除操作分两个步骤进行。

1. 在哈希表中删除节点

在哈希表中删除节点 21，需要将 bucket 中的指向调整过来。原来 bucket 中指向节点 21 的指针要改成指向节点 5，如图 6-13 中粗线所示。删除节点 21 后，节点 21 指向节点 5 的指针也不存在了，图 6-13 中以虚线表示删除掉的哈希表链接。

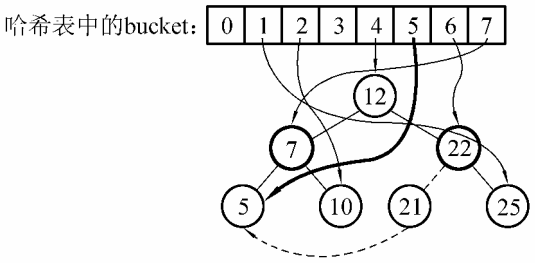


图 6-13 哈希红黑树删除示意图

2. 在红黑树中删除节点

在红黑树中删除节点 21，由于 21 节点为红色，删除后红黑树不需要调整，删除掉的节点及对应链接如图 6-13 中虚线所示。

删除节点 21 后的情况如图 6-14 所示。

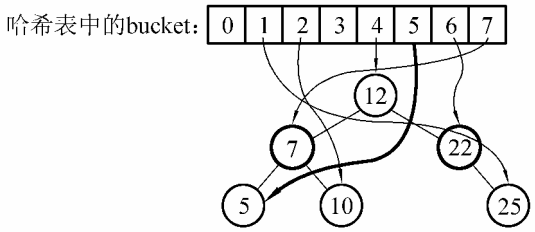


图 6-14 哈希红黑树删除节点 21 后的示意图

6.1.5 哈希红黑树的释放

从哈希红黑树的结构中可以看出，哈希红黑树的释放操作可以先沿红黑树的后序遍历顺序将节点全部释放掉，再将哈希表的 bucket 空间释放掉就完成了整个哈希红黑

树的释放操作。

6.1.6 哈希红黑树的遍历

哈希红黑树的遍历操作既可以按红黑树的遍历方法进行遍历，也可以按哈希表的遍历方法进行遍历，但是由于哈希表遍历是无序的，而红黑树按中序遍历是有序的，因此通常是按红黑树的遍历方法进行遍历。

6.1.7 哈希红黑树的编码实现

在 6.1.1 中介绍了用 C 语言结构体描述的哈希红黑树的节点类型，下面再将整个哈希红黑树用 C 语言结构体描述如下。

```
typedef struct HASHRBTREE_st {
    RBTREE *pTree；                    /* 红黑树的指针 */
    HASHRBTREENODE **ppBucket；  /* 哈希表的索引 */
    UINT uBucketCount；                /* 哈希表索引中 bucket 的数量 */
} HASHRBTREE；
```

由于哈希红黑树具有哈希表和红黑树双重特性，为了便于实现，必须复用已有的哈希表或红黑树的代码，由于红黑树实现复杂，因此还是选择复用红黑树的代码。

下面给出哈希红黑树的编码实现。

```
/** 哈希红黑树的节点创建函数
    @param void *pData——数据指针
    @return static HASHRBTREENODE *——成功返回创建的节点指针；失败返回 NULL
*/
static HASHRBTREENODE *HashRBTreeNode_Create( void *pData )
{
    HASHRBTREENODE *pNewNode；
    pNewNode = (HASHRBTREENODE *)malloc(sizeof(HASHRBTREENODE))；
    if ( pNewNode != NULL )
    {
        RBTREENODE *pTreeNode；
        pTreeNode = &(pNewNode->TreeNode)；
        pTreeNode->pLeft = NULL；
        pTreeNode->pRight = NULL；
        pTreeNode->nMagic = RBTREE_COLOR_RED；
        pTreeNode->pData = pData；
        pTreeNode->pParent = NULL；
```

```
            pNewNode->pNext = NULL；
        }
        return pNewNode；
    }

    /** 哈希红黑树的创建函数
        @param UINT uBucketNum——哈希表的 bucket 数量
        @return HASHRBTREE *——成功返回哈希红黑树指针；失败返回 NULL
    */
    HASHRBTREE *HashRBTree_Create(UINT uBucketCount)
    {
        HASHRBTREE *pHashRBTree；
        if ( uBucketCount == 0 )
        {
            return NULL；
        }
        pHashRBTree = (HASHRBTREE *)malloc( sizeof(HASHRBTREE) )；
        if ( pHashRBTree != NULL )
        {
            pHashRBTree->ppBucket = (HASHRBTREENODE **)malloc( uBucketCount
                * sizeof(HASHRBTREENODE *) )；
            if ( pHashRBTree->ppBucket != NULL )
            {
                pHashRBTree->pTree = RBTree_Create()；
                if ( pHashRBTree->pTree == NULL )
                {
                    free( pHashRBTree->ppBucket )；
                    free( pHashRBTree )；
                    pHashRBTree = NULL；
                }
                else
                {
                    memset(pHashRBTree->ppBucket, 0,
    uBucketCount *sizeof(HASHRBTREENODE *))；
                    pHashRBTree->uBucketCount = uBucketCount；
                }
            }
            else
            {
```

```
            free( pHashRBTree )；
            pHashRBTree = NULL；
        }
    }
    return pHashRBTree；
}

/** 哈希红黑树的释放函数
    @param HASHRBTREE *pHashRBTree——哈希红黑树指针
    @param DESTROYFUNC DestroyFunc——数据释放函数
    @return void——无
*/
void HashRBTree_Destroy(HASHRBTREE *pHashRBTree,
                        DESTROYFUNC DestroyFunc)
{
    /* 哈希红黑树中，只要按红黑树的释放方法将所有节点释放即可 */
    if ( pHashRBTree != NULL && pHashRBTree->pTree != NULL )
    {
        RBTree_Destroy( pHashRBTree->pTree, DestroyFunc )；
        /* 释放哈希表时，由于节点已经被释放了，因此不需要释放节点 */
        free(pHashRBTree->ppBucket)；
        free(pHashRBTree)；
    }
}

/** 哈希红黑树的插入函数
    @param HASHRBTREE *pHashRBTree——哈希红黑树指针
    @param void *pData——数据指针
    @param HASHFUNC HashFunc——哈希函数
    @param COMPAREFUNC CompareFunc——数据比较函数
    @param DESTROYFUNC DestroyFunc——数据释放函数
    @return INT——返回 CAPI_FAILED 表示失败；返回 CAPI_SUCCESS 表示成功
*/
INT HashRBTree_Insert( HASHRBTREE *pHashRBTree, void *pData,
                       HASHFUNC HashFunc,  COMPAREFUNC CompareFunc,
                       DESTROYFUNC DestroyFunc)
{
    UINT uIndex；
    INT nRet = CAPI_FAILED；
    if ( pHashRBTree != NULL )
```

```
    {
        HASHRBTREENODE    *pNewNode；
        pNewNode = (HASHRBTREENODE *)HashRBTreeNode_Create( pData )；
        if ( pNewNode == NULL )
        {
            return CAPI_FAILED；
        }
        /* 先将节点插入到红黑树中 */
        nRet = RBTree_Inter_Insert(pHashRBTree->pTree,
            (RBTREENODE *)pNewNode, CompareFunc)；
        if ( nRet == CAPI_SUCCESS )
        {
            HASHRBTREENODE    *pNode；
            /* 在红黑树中插入成功后，再将其插入到哈希表中 */
            uIndex = (*HashFunc)( pData, pHashRBTree->uBucketCount )；
            pNode = (HASHRBTREENODE *)(pHashRBTree->ppBucket[uIndex])；
        /* 在哈希表中查找对应节点 */
            while ( pNode != NULL )
            {
                if ( (*CompareFunc)( (pNode->TreeNode).pData, pData) == 0 )
                {
                /* 哈希表中存在相同数据的节点，用新数据覆盖节点数据
                    (*DestroyFunc)( (pNode->TreeNode).pData )；
                    (pNode->TreeNode).pData = pData；
                    return nRet；
            }
                pNode = pNode->pNext；
        }
            /* 将新生成的节点插入到 bucket 的头部 */
            pNode = (HASHRBTREENODE *)pHashRBTree->ppBucket[uIndex]；
            pNewNode->pNext = pNode；
            pHashRBTree->ppBucket[uIndex] = (HASHRBTREENODE *)pNewNode；
        }
    }
    return nRet；
}

/** 哈希红黑树的删除函数
    @param HASHRBTREE *pHashRBTree——哈希红黑树指针
```

```
    @param void *pData——数据指针
    @param HASHFUNC HashFunc——哈希函数
    @param COMPAREFUNC CompareFunc——数据比较函数
    @param DESTROYFUNC DestroyFunc——数据释放函数
    @return INT——成功返回 CAPI_SUCCESS；失败返回 CAPI_FAILED
*/
INT HashRBTree_Delete( HASHRBTREE *pHashRBTree, void *pData,
                       HASHFUNC HashFunc, COMPAREFUNC CompareFunc,
                       DESTROYFUNC DestroyFunc)
{
    UINT uIndex；
    HASHRBTREENODE *pNode；
    HASHRBTREENODE *pPrevNode；
    uIndex = (*HashFunc)( pData, pHashRBTree->uBucketCount )；
pNode = (HASHRBTREENODE *)(pHashRBTree->ppBucket[uIndex])；
    pPrevNode = pNode；
    /* 在哈希表中删除对应的节点，注意因为还要在红黑树中删除对应数据的节点
     * 因此节点内存必须在删除红黑树时才能释放
     */
    while ( pNode != NULL )
    {
        if ( (*CompareFunc)( (pNode->TreeNode).pData, pData ) == 0 )
        {
            if ( pPrevNode == pNode )
            {
                pHashRBTree->ppBucket[uIndex] = pNode->pNext；
            }
            else
            {
                pPrevNode->pNext = pNode->pNext；
            }
            break；
        }
        pPrevNode = pNode；
        pNode = pNode->pNext；
    }
    /* 在红黑树中将对应节点删除 */
    return RBTree_Delete(pHashRBTree->pTree, pData, CompareFunc, DestroyFunc)；
}
```

```
/** 哈希红黑树的按哈希表查找方法进行查找的函数
    @param HASHRBTREE *pHashRBTree——哈希红黑树指针
    @param void *pData——要查找的数据
    @param HASHFUNC HashFunc——哈希函数
    @param COMPAREFUNC CompareFunc——数据比较函数
    @return void *——成功返回查找到的数据指针；失败返回 NULL
*/
void *HashRBTree_HashFind( HASHRBTREE *pHashRBTree, void *pData,
                           HASHFUNC HashFunc, COMPAREFUNC CompareFunc)
{
UINT uIndex；
HASHRBTREENODE *pNode；
    /* 参数校验 */
if ( pHashRBTree == NULL || pData == NULL
        || HashFunc == NULL || CompareFunc == NULL )
{
    return NULL；
}
uIndex = (*HashFunc)( pData, pHashRBTree->uBucketCount )；
pNode = (HASHRBTREENODE *)(pHashRBTree->ppBucket[uIndex])；
/* 在哈希表冲突链中进行查找 */
while ( pNode != NULL )
{
    if ( (*CompareFunc)( (pNode->TreeNode).pData, pData) == 0 )
    {
        /* 找到了对应的数据，返回找到的数据指针 */
        return (pNode->TreeNode).pData；
    }
    pNode = pNode->pNext；
}
    return NULL；
}

/** 哈希红黑树的按红黑树查找方法进行查找的函数
    @param HASHRBTREE *pHashRBTree——哈希红黑树指针
    @param void *pData——要查找的数据
    @param COMPAREFUNC CompareFunc——数据比较函数
    @return void *——成功返回查找到的数据指针；失败返回 NULL
*/
```

```
void *HashRBTree_TreeFind( HASHRBTREE *pHashRBTree, void *pData,
                           COMPAREFUNC CompareFunc )
{
    return RBTree_Find(pHashRBTree->pTree, pData, CompareFunc)；
}

/** 哈希红黑树的逐个节点遍历初始化函数
    @param HASHRBTREE *pHashRBTree——哈希红黑树指针
    @return void——无
*/
void HashRBTree_EnumBegin(HASHRBTREE *pHashRBTree)
{
    RBTree_EnumBegin(pHashRBTree->pTree)；
}

/** 哈希红黑树的逐个节点遍历函数，按照红黑树的遍历方法进行遍历
    @param HASHRBTREE *pHashRBTree——哈希红黑树指针
    @return void *——数据指针
*/
void *HashRBTree_EnumNext(HASHRBTREE *pHashRBTree)
{
    return RBTree_EnumNext(pHashRBTree->pTree)；
}
```

哈希红黑树实现代码中，读者可能发现许多地方调用了红黑树的函数，红黑树的节点类型是 RBTREENODE 类型，而哈希红黑树的节点类型是 HASHRBTREENODE 类型，调用红黑树插入操作时是将 HASHRBTREENODE 类型转换成 RBTREENODE 类型后进行插入，HASHRBTREENODE 类型比 RBTREENODE 类型多出了一个 pNext 指针，读者可能会问，用红黑树的函数来操作 HASHRBTREENODE 类型节点时会不会出问题呢？

实际上，在调用红黑树的函数来操作 HASHRBTREENODE 类型节点时，它会自动地先将 HASHRBTREENODE 类型转换成 RBTREENODE 类型进行操作，而 HASHRBTREENODE 类型的前面刚好是 RBTREENODE 类型，所以转换后不会有问题，使用 HASHRBTREENODE 类型节点的方法很好地实现了红黑树代码的复用。

6.1.8 哈希红黑树的效率分析

哈希红黑树的效率方面主要考虑插入数据时的效率、删除数据时的效率、遍历的

效率、查找的效率和空间效率五个方面。下面就从这五个方面将哈希红黑树、哈希表及红黑树的效率进行比较。

1) 插入数据时的效率：哈希红黑树插入操作需要同时做哈希表的插入和红黑树的插入，消耗的时间相当于哈希表的插入操作时间加上红黑树的插入操作时间。

2) 删除数据时的效率：哈希红黑树的删除同样需要做哈希表的删除及红黑树的删除，消耗时间为哈希表删除加上红黑树删除的时间。

3) 遍历的效率：哈希红黑树遍历操作的效率和红黑树一样。

4) 查找的效率：哈希红黑树按红黑树查找方法进行查找时，效率和红黑树一样；按哈希表查找方法进行查找时效率和哈希表一样。

5) 空间效率：哈希红黑树消耗的辅助空间相当于红黑树消耗的辅助空间加上哈希表消耗的辅助空间，如果哈希表的 bucket 数量和节点数量一样多，哈希红黑树消耗的辅助空间为红黑树的 1.4 倍。

从效率分析可以看出，哈希红黑树在插入、删除操作上比哈希表或红黑树慢，查找和哈希表及红黑树相当，使用的辅助空间也比红黑树多，可见哈希红黑树并无优势。使用哈希红黑树的好处就是可以按哈希表查找方法查找，找到后可以有序输出，查找速度比红黑树快，它避免了哈希表里数据无序以及红黑树查找速度不足的两个缺点。

在实际应用中，如以太网二层就需要用查找到某个数据后再按顺序输出一定数量的数据，如果使用哈希红黑树便可以很好地满足和适应这种应用需求。

6.2 哈希 AVL 树

6.2.1 哈希 AVL 树的基本概念

前面介绍了复合数据结构哈希红黑树，它避免了哈希表里数据无序及红黑树查找速度不足的缺点。在软件设计中，经常会追求设计更快的查找算法。像哈希表查找，一次就可以定位到数据，已经够快的了，但是哈希表对哈希函数的设计有很高要求，若哈希函数设计不好，理论上查找的最坏复杂度为 O(n)；在二叉树中，查找速度较快的为 AVL 树，能不能将哈希表和 AVL 树结合起来，使得查找更快呢？

本节就讨论另外一个复合数据结构——哈希 AVL 树。顾名思义，哈希 AVL 树是哈希表和 AVL 树的复合数据结构，那么哈希 AVL 树是不是也采用和哈希红黑树同样的复合方法呢？哈希 AVL 树的每个节点是不是既有哈希表的特性，又有 AVL 树的特性呢？

如果将哈希 AVL 树设计成类似哈希红黑树，通常 AVL 树比红黑树查找要快，但插入删除效率较低，哈希 AVL 树可以采用哈希表的查找方法进行查找，AVL 树的查

找方法在这里不起作用，因此，查找速度也就和哈希表一样，还不如直接使用哈希红黑树。要想查找更快，有必要换个思路来设计哈希 AVL 树。

设计哈希 AVL 树的目的首先是要解决哈希表的缺点。哈希表的使用对哈希函数有很高要求，如果哈希函数设计不好可能有很多个数据具有相同的哈希值，而对这些相同哈希值的数据查找是采用顺序查找方式，效率非常低。如果对哈希表具有相同哈希值的数据不使用链式方法解决冲突，而采用 AVL 树来解决冲突，那么对具有相同哈希值的数据的查找将变成 AVL 树的查找，比链式的顺序查找效率将提高很多。

哈希 AVL 树与哈希表的区别就是 bucket 中的指针是 AVLTREE 的节点类型，因此用 C 语言结构体来描述哈希 AVL 树如下。

```
typedef BINTREEBASENODE AVLTREENODE；

typedef struct HASHAVLTREE_st {
    AVLTREENODE **ppBucket；  /* 索引表指针 */
    UINT uBucketCount；       /* 索引表的大小 */
    UINT uNodeCount；         /* 表中实际节点的个数 */
    UINT uCurBucketNo；       /* 当前要执行的 bucket 序号 */
    AVLTREENODE *pCurEntry；  /* 当前 bucket 中下一个要执行的节点条目 */
} HASHAVLTREE；
```

可见，哈希 AVL 树和哈希表的唯一区别就是将 SINGLENODE 改成了 AVLTREENODE，即哈希 AVL 树中的每个 bucket 指向的是一颗 AVL 树。

哈希 AVL 树的结构如图 6-15 所示。

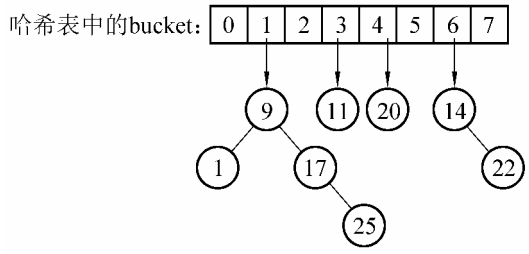


图 6-15　哈希 AVL 树的示意图

6.2.2　哈希 AVL 树的查找

哈希 AVL 树的查找和哈希表类似，唯一的区别就是哈希表是找到 bucket 序号后按链表的顺序进行查找，而哈希 AVL 树则是找到 bucket 序号后进行 AVL 树的查找。下面通过在图 6-15 所示中的哈希 AVL 树中查找数值 17 来介绍哈希 AVL 树的查找过程。

首先将 17 除以 8，余数为 1，得到 bucket 序号为 1，从序号为 1 的 bucket 中开始查找，第 1 个位置为节点 9，9 小于 17，因此再查找节点 9 的右节点，为节点 17，就找到了要找的节点。查找过程如图 6-16 中的粗线所示。

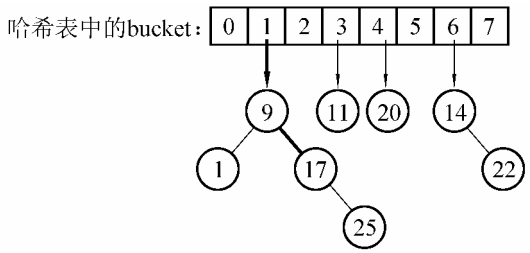


图 6-16　哈希 AVL 树的查找过程图

从查找过程可以看出，当某个 bucket 中存放的数据非常多时，也就是哈希表中的某个 bucket 冲突数据非常多时，采用哈希 AVL 树方式存放这些冲突数据对查找效率会有明显的提高。

6.2.3　哈希 AVL 树的插入

哈希 AVL 树的插入需要先找到 bucket 位置，然后在 bucket 指向的 AVL 树中进行插入，下面还是以图 6-15 中的数据为例来介绍哈希 AVL 树的插入过程。

假设要按 17，11，20，9，14，1， 22， 25 的顺序在哈希 AVL 树中进行插入，插入过程如下。

步骤 1：插入节点 17，17 除以 8 余 1，因此插入在位置 1 上，如图 6-17 所示。

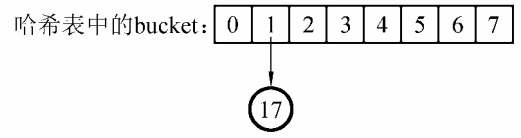


图 6-17　插入节点 17 之后的情况

步骤 2：插入节点 11 和节点 20，11 除以 8 余 3，节点 11 插入在位置 3 上，20 除以 8 余 4，节点 20 插入在位置 4 上，如图 6-18 所示。

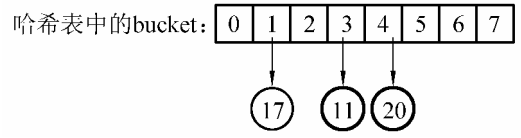


图 6-18　插入节点 11、节点 20 之后的情况

步骤 3：再插入节点 9，9 除 8 余 1，因此也需要插入在位置 1 上，由于位置 1 上已有节点 17，而 9 比 17 小，因此节点 9 插入在节点 17 的左边，如图 6-19 所示。

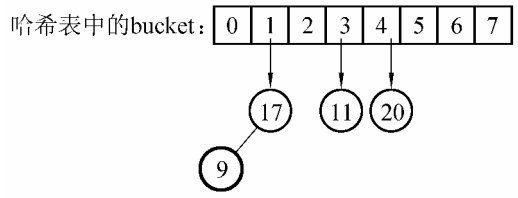


图 6-19　插入节点 9 之后的情况

步骤 4：插入节点 14，14 除以 8 余 6，因此将节点 14 插入在位置 6 上，如图 6-20 所示。

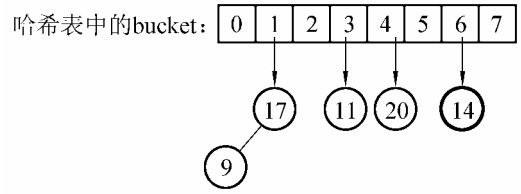


图 6-20　插入节点 14 之后的情况

步骤 5：插入节点 1，1 除以 8 余 1，因此将节点 1 插入在位置 1 上，但由于位置 1 原来有节点 17 和 9，按 AVL 树的插入操作，需要做一次右旋操作来进行平衡，如图 6-21 所示。

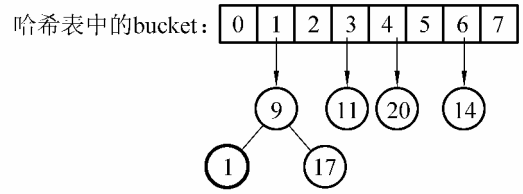


图 6-21　插入节点 1 之后的情况

步骤 6：插入节点 22，22 除以 8 余 6，因此插入在位置 6 上，但由于位置 6 上原来已有节点 14，所以将节点 22 插入在节点 14 的右边，如图 6-22 所示。

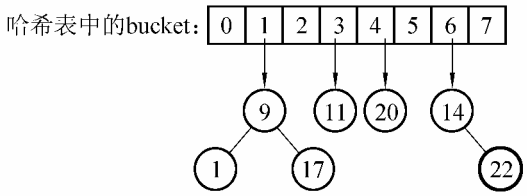


图 6-22　插入节点 22 之后的情况

步骤 7：插入节点 25，25 除 8 余 1，因此插入在位置 1 上，由于原来已有数据，将节点 25 插入在 AVL 树的最右边，如图 6-23 所示。

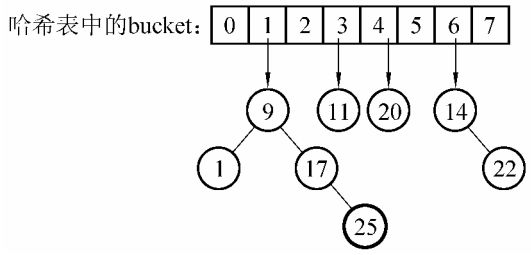


图 6-23　插入节点 25 之后的情况

6.2.4　哈希 AVL 树的删除

下面通过从图 6-23 所示的哈希 AVL 树中删除节点 1 来介绍哈希 AVL 树的删除过程。

步骤 1：先用 1 除以 8 余 1，因此从 bucket 数组中的第 1 个位置开始查找，找到后将节点 1 删除掉，删掉的部分如图 6-24 中的虚线所示。

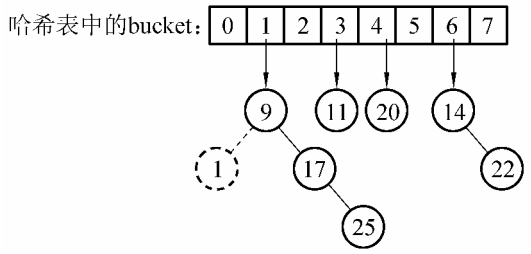


图 6-24　删除节点 1 之后的情况

步骤 2：删除掉节点 1 后，由于节点 9 的左右高度差为 2，要调整 AVL 树的平衡，需要做一次左旋操作，删除后的结果如图 6-25 所示。

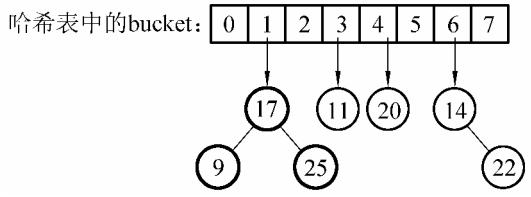


图 6-25　删除节点 1 之后进行左旋调整平衡后的情况

6.2.5 哈希 AVL 树的释放

从哈希 AVL 树的结构中可以看出，哈希 AVL 树的释放操作需要沿哈希表中的 bucket 数组遍历将 bucket 中的 AVL 树全部释放掉，再将哈希表的 bucket 空间释放掉，才能完成整个哈希 AVL 树的释放。

6.2.6 哈希 AVL 树的遍历

哈希 AVL 树的遍历操作需要按哈希表的 bucket 顺序进行，在每个 bucket 中，再按 AVL 树的遍历方法进行遍历。需要注意的是，虽然 AVL 树可以按大小顺序遍历，但整个哈希 AVL 树是没有顺序的，不同 bucket 中的元素大小是无法比较的。

6.2.7 哈希 AVL 树的编码实现

前面介绍了用 C 语言结构体描述的哈希 AVL 树的节点类型，现将整个哈希 AVL 树用 C 语言结构体描述如下。

```
typedef BINTREEBASENODE AVLTREENODE；

typedef struct HASHAVLTREE_st {
    AVLTREENODE **ppBucket；  /* 索引表指针 */
    UINT uBucketCount；       /* 索引表的大小 */
    UINT uNodeCount；         /* 表中实际节点的个数 */
    UINT uCurBucketNo；       /* 当前要执行的 bucket 序号 */
    AVLTREENODE *pCurEntry；  /* 当前 bucket 中下一个要执行的节点条目 */
} HASHAVLTREE；
```

由于哈希 AVL 树是哈希表和 AVL 树的复合体，为了便于实现，必须复用已有的 AVL 树的代码。

下面给出哈希 AVL 树的编码实现。

```
/** 哈希 AVL 树的创建函数
    @param UINT uBucketCount——哈希 AVL 树中 bucket 数量
    @return HASHAVLTREE *——成功返回创建的哈希 AVL 树指针；失败返回 NULL
*/
HASHAVLTREE *HashAVLTree_Create(UINT uBucketCount)
{
    HASHAVLTREE *pTree；
    if ( uBucketCount == 0 )
    {
```

```
        return NULL；
    }
    pTree = (HASHAVLTREE *)malloc( sizeof(HASHAVLTREE) )；
    if ( pTree == NULL )
    {
        return NULL；
    }
    pTree->uNodeCount = 0；
    pTree->uBucketCount = uBucketCount；
    pTree->ppBucket = (AVLTREENODE **)malloc( uBucketCount
        * sizeof(AVLTREENODE *))；
    if (pTree->ppBucket == NULL)
    {
        free( pTree )；
        return NULL；
    }
    memset(pTree->ppBucket, 0, uBucketCount *sizeof(AVLTREENODE *))；
    pTree->pCurEntry = NULL；
    pTree->uCurBucketNo = 0；
    return pTree；
}

/** 哈希 AVL 树的释放函数，将哈希 AVL 树中所有数据和节点及整个哈希 AVL 树整体
    释放
    @param HASHAVLTREE *pHashAVLTree——要释放的哈希 AVL 树指针
    @param DESTROYFUNC DestroyFunc——数据释放回调函数
    @return void——无
*/
void HashAVLTree_Destroy(HASHAVLTREE *pHashAVLTree,
                         DESTROYFUNC DestroyFunc)
{
    AVLTREENODE **ppBucket；
    AVLTREENODE *pNode；
    UINT i；
    if ( pHashAVLTree == NULL )
    {
        return；
    }
    ppBucket = pHashAVLTree->ppBucket；
```

```
    for ( i = 0；i < pHashAVLTree->uBucketCount；i++ )
    {
        pNode = ppBucket[i]；
        BinTreeBaseNode_Destroy(pNode, DestroyFunc)；
    }
    free(ppBucket)；
    /* 将 ppBucket 设成空指针以避免哈希表被重新使用时造成内存泄漏 */
    pHashAVLTree->ppBucket = NULL；
    free( pHashAVLTree )；
}

/** 哈希 AVL 树的插入函数
    @param HASHAVLTREE *pHashAVLTree——要插入的哈希 AVL 树指针
    @param void *pData——要插入的数据指针
    @param HASHFUNC HashFunc——哈希函数
    @param COMPAREFUNC CompareFunc——数据比较回调函数
    @return INT——成功返回 CAPI_SUCCESS；失败返回 CAPI_FAILED
*/
INT HashAVLTree_Insert(HASHAVLTREE *pHashAVLTree, void *pData,
                       HASHFUNC HashFunc, COMPAREFUNC CompareFunc)
{
    UINT uIndex；
    if ( pHashAVLTree == NULL || pData == NULL || HashFunc == NULL )
    {
        return CAPI_FAILED；
    }
    uIndex = (*HashFunc)( pData, pHashAVLTree->uBucketCount )；
    /* 将新节点插入到链表的头部 */
    AVLTreeNode_Insert(&((pHashAVLTree->ppBucket)[uIndex]), pData, CompareFunc)；
    pHashAVLTree->uNodeCount += 1；
    return CAPI_SUCCESS；
}

/** 哈希 AVL 树的节点删除函数
    @param HASHAVLTREE *pHashAVLTree——要删除的哈希 AVL 树指针
    @param void *pData——要删除的数据指针
    @param HASHFUNC HashFunc——哈希函数
```

```
    @param COMPAREFUNC CompareFunc——数据比较回调函数
    @param DESTROYFUNC DestroyFunc——数据释放回调函数
    @return INT——成功返回 CAPI_SUCCESS；失败返回 CAPI_FAILED
*/
INT HashAVLTree_Delete( HASHAVLTREE *pHashAVLTree, void *pData,
                        HASHFUNC HashFunc, COMPAREFUNC CompareFunc,
                        DESTROYFUNC DestroyFunc)
{
    UINT uIndex；
    if ( pHashAVLTree == NULL || pData == NULL || HashFunc == NULL
        || CompareFunc == NULL )
    {
        return CAPI_FAILED；
    }
    uIndex = (*HashFunc)( pData, pHashAVLTree->uBucketCount )；
    if ( AVLTreeNode_Delete(&((pHashAVLTree->ppBucket)[uIndex]), pData,
        CompareFunc, DestroyFunc) != CAPI_NOT_FOUND )
    {
        pHashAVLTree->uNodeCount－= 1；
    }
    return CAPI_SUCCESS；
}

/** 哈希 AVL 树的查找函数
    @param HASHAVLTREE *pHashAVLTree——要查找的哈希 AVL 树指针
    @param void *pData——要查找的数据指针
    @param HASHFUNC HashFunc——哈希函数
    @param COMPAREFUNC CompareFunc——数据比较回调函数
    @return void *——成功返回找到的节点中的数据指针；失败返回 NULL
*/
void *HashAVLTree_Find( HASHAVLTREE *pHashAVLTree, void *pData,
                        HASHFUNC HashFunc, COMPAREFUNC CompareFunc )
{
    UINT uIndex；
    AVLTREENODE *pNode；
    if ( pHashAVLTree == NULL || HashFunc == NULL || CompareFunc == NULL )
    {
        return NULL；
    }
```

```
    uIndex = (*HashFunc)( pData, pHashAVLTree->uBucketCount )；
    pNode = (pHashAVLTree->ppBucket)[uIndex]；
    return BinTree_Find(pNode, pData, CompareFunc)；
}

/** 哈希 AVL 树的逐个节点遍历开始函数
    @param HASHAVLTREE *pHashAVLTree——哈希 AVL 树指针
    @return void——无
*/
void HashAVLTree_EnumBegin(HASHAVLTREE *pHashAVLTree)
{
    AVLTREENODE *pCursor；
    pHashAVLTree->uCurBucketNo = 0；
    pCursor = pHashAVLTree->ppBucket[0]；
    if ( pCursor != NULL )
    {
        while ( pCursor->pLeft != NULL )
        {
            pCursor = pCursor->pLeft；
        }
    }
    pHashAVLTree->pCurEntry = pCursor；
}

/** 哈希 AVL 树的逐个节点遍历函数
    @param HASHAVLTREE *pHashAVLTree——哈希 AVL 树指针
    @return void *——返回遍历的节点数据指针，如果遍历完则返回 NULL
*/
void *HashAVLTree_EnumNext(HASHAVLTREE *pHashAVLTree)
{
    void *pData；
    AVLTREENODE *pCursor；
    pCursor = pHashAVLTree->pCurEntry；
    while ( pCursor == NULL )
    {
        pHashAVLTree->uCurBucketNo += 1；
        if ( pHashAVLTree->uCurBucketNo >= pHashAVLTree->uBucketCount )
        {
            return NULL；
```

```
            }
            pCursor = pHashAVLTree->ppBucket[pHashAVLTree->uCurBucketNo];
        }
        if ( pCursor == NULL )
        {
            return NULL;
        }
        while ( pCursor->pLeft != NULL )
        {
            pCursor = pCursor->pLeft;
        }
        pData = pCursor->pData;
        if ( pCursor->pRight != NULL )
        {
            pCursor = pCursor->pRight;
            while ( pCursor->pLeft != NULL )
            {
                pCursor = pCursor->pLeft;
            }
        }
        else
        {
            AVLTREENODE *pParent = pCursor->pParent;
            while ( pParent != NULL && pParent->pRight == pCursor )
            {
                pCursor = pParent;
                pParent = pParent->pParent;
            }
            pCursor = pParent;
        }
        pHashAVLTree->pCurEntry = pCursor;
        return pData;
    }
```

6.2.8 复合数据结构的分类

本章介绍了哈希红黑树和哈希 AVL 树两种复合数据结构，它们的设计思路各不相

同，可以将设计分成以下两类。

1. 节点叠加型复合

节点叠加型复合是将两种不同数据结构类型的节点内容叠加到一起的设计方法，本书里使用这种方法实现的复合数据结构主要有以下三种。

① 哈希红黑树，采用哈希表和红黑树的节点叠加方法进行复合实现；

② 哈希链表，采用哈希表和链表的节点叠加方法进行复合实现；

③ 本书第 9 章内存管理中的空间链表，采用数组和链表的叠加，即节点中包含了数组下标又包含了链表的后向指针，实现了高效的内存管理算法。

2. 节点链接式复合

节点链接式复合在实际工作中使用得非常多，它是采用一种数据结构的每个节点指向另外一种数据结构的方法来实现的。本书里使用这种方法实现的复合数据结构主要有以下三种。

① 哈希 AVL 树，使用哈希表的 bucket 数组元素指向 AVL 树的方式来进行复合实现，即由数组中的节点链向 AVL 树；

② 哈希表，实际上和哈希 AVL 树类似，采用数组中的元素指向链表的方式来进行复合实现；

③ 整块内存链表，也是采用数组中的元素指向链表的方式来进行复合实现的。

6.3 抗 DoS/DDoS 攻击的实例

6.3.1 DoS/DDoS 攻击的概念

某商店里正在促销一种商品，价钱比平时便宜了一半，于是吸引了很多人排队购买。这个商店的竞争对手为了破坏这次促销行动，找来一批人排队，但排到时只是看一看，并且还和服务员争论一番，并不购买，排完队后马上又接着排队。这样，这个队伍中大部分都是只排队不购买的人员，队伍排得很长，很多真正想购买的人却由于队伍较长，等得不耐烦就离去了，于是这次促销行动就被竞争对手成功地破坏掉了。

上面讲的故事在现实中可能不会发生，但是在 Internet 中，类似的事情却经常发生。攻击者就是采用这种手段，发送大量小报文给被攻击的服务器，使得服务器疲于处理攻击报文，无法再处理其他正常请求，就像促销商店只能处理竞争对手派出的人员，无法再处理其他真正想要购买的人员一样，这种攻击方式被称为 DoS(Denial of Service)攻击，DoS 翻译成中文就是“拒绝服务”的意思。

DoS 攻击是目前 Internet 网络中一种常见的攻击手段。这种攻击一般是采用发送合理的请求来占用过多的服务器资源，使服务器资源消耗殆尽。这种攻击目前还没有

很有效的方法阻止它，因为攻击者发送的是合法报文，服务器很难判断哪些是正常报文，哪些是攻击报文，就像前面故事中的商店很难区分排队的人哪些是想购买的，哪些是不想购买的。只有等攻击者发送的报文足够多时才能判断出它是攻击报文，但此时攻击已经发生了一段时间。

6.3.2 常见 DoS/DDoS 攻击手段及防范策略

DoS 攻击从 1996 年开始到现在已经发展了很长的时间，攻击的方法也多种多样，如控制多台机器来联合发动攻击则被称为 DDoS(Distributed Denial of Service)，意为“分布式拒绝服务”，目前使用得较多的都是 DDoS 攻击。

在 TCP/IP 网络中常见的 DoS/DDoS 攻击手段主要有以下三种。

1) TCP 半连接攻击(又称为 SYN Flood 攻击)。这是一种经典的攻击方法，它是利用 TCP 连接的三次握手机制进行攻击的。第一次握手时，攻击机器向服务器发送虚假的源 IP 和源端口的 SYN 报文；第二次握手时，服务器回送 SYN/ACK 给虚假的源 IP 和源端口，此时，由于源 IP 和源端口不存在，攻击机器自然不会同服务器进行第三次握手，因而服务器处于等待超时状态，存在大量的 SYN_RCVD 状态。由于服务器的缓存资源有限，攻击者只要发送足够数量的小报文便可以将服务器上的缓存资源全部占用，服务器由于等不到对方发送回 ACK 包，处于超时等待，而这个超时缺省是很长的，通常为 75 秒。由于服务器缓存资源被占满，其他合法的用户发向服务器的连接请求都遭拒绝，攻击者的目的达到了。这种攻击方法是一种非常有效的攻击方法，很难防御。

2) TCP 全连接攻击。这种攻击是通过和服务器建立正常连接来占用服务器的连接资源，达到攻击目的的。采用这种方法，攻击者需要使用很多“僵尸”主机来向服务器建立大量的 TCP 连接，直到将服务器的连接资源耗尽，造成服务器的拒绝服务。这种攻击由于建立了连接，因此可以通过追踪源 IP 的方法来进行防御。

3) 资源耗费型攻击。攻击机器向服务器提交某种特定类型的请求如服务器需要查询数据库等，而服务器响应这种请求需要耗费大量的时间，大量的这类请求令服务器不堪重负，最终导致资源耗尽而拒绝其他正常的请求。

当然，实际情况中的攻击方法非常多，如 UDP Flood 攻击、同时用多种攻击方法进行攻击等，这里就不一一列出了。下面简单介绍一下对付上述三种攻击的方法。

对第一种攻击，通常的做法是在防火墙层拦截 TCP 连接，先由防火墙和客户端建立连接，连接成功后再由防火墙代客户端和服务器建立连接，最后防火墙将两个连接透明地合并起来。

对第二种攻击，由于“僵尸”主机的数量是有限的，可以很容易判断出哪些机器占用了大量连接请求，直接将这些机器记入黑名单，拒绝它们的请求即可；同时还可

以根据“僵尸”主机的 IP 去追踪控制“僵尸”主机的攻击者。

对第三种攻击，可以采取多种手段进行规避。首先服务器上尽量不要有资源消耗很大的操作，如果确实有这种操作，可以对这些操作进行特别处理，采取一定的策略限制客户端大量请求这种消耗资源的操作。如可以严格规定这种操作同时进行的最大数量，以及每个客户端同时可以提交这种操作的最大数量和频率等。

6.3.3 抗 DoS/DDoS 攻击的实现

在上述几种攻击中，比较难防范的还是第一种攻击，因此下面就重点介绍第一种攻击的防范方法。

要实现对第一种攻击的防范，需要设计一张表来保存客户端的信息。由于客户端是伪造的，数量非常大，必须采用查找速度非常快的表，这里可以使用哈希 AVL 树来保存客户端信息。对于每个客户，需要保存客户标识(IP 地址)，上次连接的开始时间、连接总时间、连接次数、客户端发送的报文等信息。

```
typedef struct CLIENTDATA_st {
    DWORD dwIPAddr；
    clock_t StartClock；
    clock_t TotalClock；
    DWORD dwConnectTimes；
    INT nTcpData[TCPHEAD_LENGTH]；
} CLIENTDATA；
```

在防火墙端，可以将所有收到的客户端请求信息都保存到表里，通常采用以下策略来进行保护。

① 当表中的数量超过某一规定值时，删除一些老的请求；
② 当某个连接完全建立时，从表中删除对应项；
③ 定时将超时的请求从表中删除掉；
④ 将连接的超时时间设短。

6.3.4 抗 DoS/DDoS 攻击的编码实现

下面给出抗 DoS 攻击的编码，读者可以通过这个编码来学习如何使用哈希 AVL 树。

```
#define   TCPHEAD_LENGTH          20
#define   DEFAULT_CONNECT_TIME    5000
#define   MAX_CONNECT_TIMES       5
```

```
/** 判断是否为 DoS 攻击的回调函数
    @param CLIENTDATA *pData——CLIENTDATA 指针
    @return BOOL——TRUE 表示是 DoS 攻击；FALSE 表示不是
*/
typedef   BOOL (*ISDOSATTACKFUNC)(CLIENTDATA *pData)；

/** DoSUnattack 创建函数，创建一个哈希 AVL 树
    @param UINT uBucketCount——哈希 AVL 树的 bucket 数组大小
    @return HASHAVLTREE *——哈希 AVL 树指针
*/
HASHAVLTREE *DoSUnattack_Create(UINT uBucketCount)
{
    HASHAVLTREE *pTree；
    pTree = HashAVLTree_Create(uBucketCount)；
    return pTree；
}

/** 哈希 AVL 树的比较回调函数，比较两个 IP 地址是否相等
    @param void *p1——CLIENTDATA 指针类型
    @param void *p2——DWORD 类型
    @return INT——返回 0 表示 p1 =p2；返回－1 表示 p1＜p2；返回 1 表示 p1＞p2
*/
INT IPCompare(void *p1, void *p2)
{
    CLIENTDATA    *pData；
    DWORD dwIPAddr；
    pData = (CLIENTDATA *)p1；
    dwIPAddr = (DWORD)p2；
    if ( pData->dwIPAddr == dwIPAddr )
    {
        return 0；
    }
    else if ( pData->dwIPAddr < dwIPAddr )
    {
        return－1；
    }
    else
    {
        return 1；
    }
```

```
}

/** 哈希 AVL 树的计算哈希值的回调函数
    @param void *p——要计算哈希值的对应关键词指针
    @param UINT uBucketCount——哈希 AVL 树中的 bucket 数组大小
    @return INT——bucket 数组下标
*/
INT HashClientData(void *p, UINT uBucketCount)
{
    CLIENTDATA    *pData = (CLIENTDATA *)p；
    return pData->dwIPAddr % uBucketCount；
}

/** 接收客户端的连接
    @param HASHAVLTREE *pTree——哈希 AVL 树指针
    @param DWORD dwIPAddr——对方 IP 地址
    @param INT *pTcpIpData——连接包数据
    @param ISDOSATTACKFUNC IsDosAttack——判断是否为 DoS 攻击的回调函数
    @return void——无
*/
void DoSUnattack_RecvConnection( HASHAVLTREE *pTree, DWORD dwIPAddr,
                                 INT *pTcpIpData, ISDOSATTACKFUNC IsDosAttack)
{
    CLIENTDATA *pData；
    clock_t ConnectClock；
    pData = (CLIENTDATA *)HashAVLTree_Find(pTree, (void *)dwIPAddr,
                                           HashInt, IPCompare)；
    ConnectClock = clock()；
    if ( pData != NULL )
    {
        pData->dwConnectTimes += 1；
        if ( (*IsDosAttack)(pData) )
        {
            /* 发现攻击，拒绝连接，将对方 IP 加入黑名单 */
        }
    }
    else
    {
        pData = (CLIENTDATA *)malloc(sizeof(CLIENTDATA))；
```

```
            if ( pData != NULL )
            {
                pData->dwConnectTimes = 1；
                pData->dwIPAddr = dwIPAddr；
                pData->StartClock = clock()；
                pData->TotalClock = 0；
                memcpy(pData->nTcpData, pTcpIpData, TCPHEAD_LENGTH *sizeof(INT))；
                HashAVLTree_Insert(pTree, pData, HashClientData, IPCompare)；
            }
        }
        return CAPI_SUCCESS；
    }

    /** DosUnattack 的连接任务，负责和客户端的连接
        @param HASHAVLTREE *pTree——哈希 AVL 树指针
        @param ISDOSATTACKFUNC IsDosAttack——判断是否为 DoS 攻击的回调函数
        @return void——无
    */
    void DosUattack_ConnectTask( HASHAVLTREE *pTree,
                                 ISDOSATTACKFUNC IsDosAttack)
    {
        CLIENTDATA    *pData；
        HashAVLTree_EnumBegin(pTree)；
        while( (pData = HashAVLTree_EnumNext(pTree) ) != NULL )
        {
            if ( (*IsDosAttack)(pData) )
            {
                /* 发现攻击，拒绝连接，将对方 IP 加入黑名单*/
            }
            else
            {
                /* 和客户端建立连接 */
            }
        }
    }
```

本 章 小 结

本章主要介绍了复合数据结构哈希红黑树和哈希 AVL 树；并介绍了复合数据结构的分类；最后还介绍了一个抗 DoS 攻击的实例。通过这些内容的学习，读者可以掌握一般的复合数据结构设计和分析方法，学会根据各种数据结构的优缺点灵活应用各种数据结构。

习题与思考

1. 使用哈希 AVL 树编码替代哈希表实现第 4 章的 WebServer CACHE 文件管理。
2. 编码调用哈希红黑树去实现一个英文词典的管理功能。
3. 从插入、删除、查找、有序性、时间效率、空间效率等方面将哈希红黑树、哈希 AVL 树和前面讲过的各种数据结构进行综合比较。

7 图

本章介绍图的操作与算法，包括常用的最短路径算法、最小生成树算法、深度优先搜索算法、宽度优先搜索算法等；还介绍了无环有向图(软件设计中的依赖关系图便是一个无环有向图)的分层算法及哈希顿圈的算法，在哈密顿圈算法的设计中使用了陌生优先的设计思想。

在软件开发中，经常要用到图的操作及算法，如在电信网络中，计算路由时经常要用到最短路径算法；在以太网二层中要用到最小生成树算法，图的遍历更是经常需要用到。本章主要介绍在软件设计中有着非常广泛应用并极具实用价值的图的算法，并给出编码实现。

7.1 图的基本概念和描述方法

7.1.1 图的基本概念

自从 1736 年大数学家欧拉解决哥尼斯堡七桥问题而发表了图论的首篇论文以来，图论这门学科发展到现在已经有 270 年历史了。

图是由顶点和边组成的集合，当图中的边是无方向的时候称为无向图，如果边有方向则称为有向图。当顶点和边的数量是有限个数时称为有限图，顶点或边的数量无限时叫无限图。本书中讨论的都是有限图。

如果图中任一顶点没有包含连至自身的边，并且任意两个顶点间最多只有一条边，这种图被称为简单图。本章中讨论的也都是简单图。

由图的顶点出发的边数称为顶点的度数，由于每条边有两个顶点，因此每条边被

计算了两次度数，可见所有顶点度数之和为总边数的两倍。

7.1.2 图的描述方法

图在计算机中描述通常有矩阵法和邻接表法。对于稠密图(图中的边数比较多)一般使用矩阵法，而稀疏图则一般使用邻接表法。在实际情况中，大部分图都是稀疏图，所以本章采用邻接表法来描述图。

首先给出图的节点的 C 语言结构体描述如下。

```
typedef long DISTANCE；

typedef struct GRAPHNODE_st {
    void *pProperties；
    SINGLELIST *pEdgeList；
    void *pMagic；          /* 内部算法使用，外部不可以改变 */
    INT nMagic；            /* 内部算法使用，外部不可以改变 */
} GRAPHNODE；
```

GRAPHNODE 结构体中的 pProperties 表示节点属性，节点属性可以根据实际情况由调用者自行定义；pMagic 和 nMagic 两个变量是内部使用的，主要用于实现图的算法；pEdgelist 用来保存从该顶点出发的所有边，链表的节点类型是边的结构；边的结构定义如下。

```
typedef struct EDGE_st {
    DISTANCE distance；
    INT nMagic；
    GRAPHNODE *pNode1；
    GRAPHNODE *pNode2；
    void *pProperties；
} EDGE；
```

EDGE 结构中同样也有一个 nMagic 变量，这个变量也是为图的各种算法而预留的；pNode1 和 pNode2 是边的两个顶点；pProperties 是边的属性；distance 表示边的长度。

有了图的节点和边的定义，再给出整个图的定义，用 C 语言结构体描述如下。

```
typedef struct GRAPH_st {
    GRAPHNODE **ppNodeArray；
    SINGLELIST *pEdgeList；
    INT nNodeCount；
```

```
    void *pProperties；
} GRAPH；
```

GRAPH 结构体中，有一个指向 GRAPHNODE 指针的指针数组 ppNodeArray，它用来保存图的所有顶点；pEdgeList 用来保存图中所有边的一条链表；nNodeCount 记录图的顶点总数；pProperties 用来记录图的属性。

7.2 Dijkstra 最短路径算法

7.2.1 Dijkstra 最短路径算法的描述

Dijkstra 最短路径算法，是找出一个顶点到其他顶点间的最短路径的算法，可计算出某个顶点到所有其他顶点间的最短路径。这种算法在实际中有着广泛的应用，算法过程描述如下。

假设 Len(v)表示一个顶点到给定顶点 U_0 的最短距离，w(u, v)表示两个顶点间的距离，如果 U，V 不相邻，w(u, v) = 无穷大。记 md(u,v) = w(u,v) + Len(u)。

步骤 1：对给定的顶点 V，标记这个顶点为用过，Len(v) = 0。同时初始化所有顶点的最短距离为 0。将顶点 V 放入集合 S 中。

步骤 2：计算任一和集合 S 中的顶点相邻且不在集合 S 中的顶点 U 到集合 S 中和它相邻顶点 V 的 md(u,v)值，找出一个最小的 md(u,v)值，得到了满足最小 md(u,v)的顶点 U，将顶点 U 放入集合 S 中，同时记录顶点 U 的父节点为 V，将边 e = uv 标志为用过。

步骤 3：重复步骤 2，直到将所有的顶点都放入集合 S 中。

7.2.2 Dijkstra 最短路径算法的过程图解

下面用图解的方法来展示一个最短路径算法的全过程。

步骤 1：找出与顶点 A 相邻的 4 个顶点 B、C、D、E，计算它们到顶点 A 的距离分别为 15，25，19，12，其中距离最近的顶点为顶点 E，距离值为 12，这样便找到第 1 个到顶点 A 最短距离的顶点为顶点 E，如图 7-1 所示。

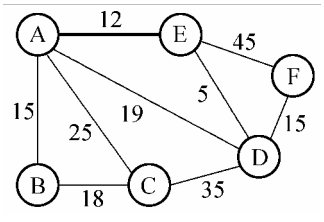


图 7-1　顶点 A 的第 1 个最近顶点 E

步骤 2：找出与顶点 A 相邻的顶点 B、C、D，并找出与 E 相邻的顶点 D、F，计算它们的距离分别为 A->B=15，A->C=25，A->D=19，E->D=AE + ED = 12+5=17，E->F=AE + EF = 12 + 45 = 57，因此得到最小距离值为 A->B = 15，这

样便找到第 2 个到顶点 A 最短距离的顶点为顶点 B，距离值为 15，如图 7-2 所示。

步骤 3：找出与顶点 A 相邻的顶点 C、D，计算它们到顶点 A 的距离分别为 A->C = 25，A->D = 19；找出与顶点 B 相邻的顶点 C，距离为 B->C = AB + BC = 15 + 18 = 33；找出与顶点 E 相邻的顶点 D、F，它们的距离为 E->D = 17，E->F = 57，因此得到最短距离值为 E->D = 17，这样便找到第 3 个到顶点 A 距离最短的顶点 D，如图 7-3 所示。

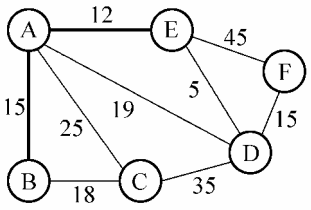


图 7-2　顶点 A 的第 2 个最近顶点 B

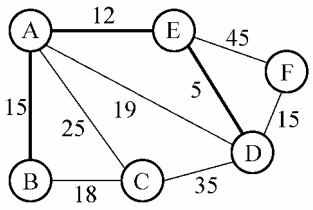


图 7-3　顶点 A 的第 3 个最近顶点 D

以此类推，找到第 4 个到顶点 A 最短距离的顶点为顶点 C，如图 7-4 所示；第 5 个到顶点 A 最短距离的顶点为顶点 F，如图 7-5 所示。

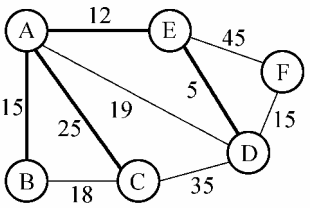


图 7-4　顶点 A 的第 4 个最近顶点 C

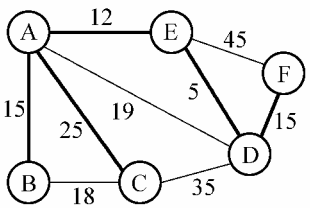


图 7-5　顶点 A 的第 5 个最近顶点 F

7.2.3　Dijkstra 最短路径算法的编码实现

下面给出 Dijkstra 最短路径算法的编码实现。

```
/** 计算图中两个节点间的最短路径函数，计算完后，每个节点的 pMagic 是用来存放到
    源节点最短路径的下一个节点指针，nMagic 用来存放到源节点的最短路径的总距离
    @param LPGRAPH pGraph——要计算的图
    @param LPGRAPHNODE pSrcNode——源节点
    @param LPGRAPHNODE pTagNode——目标节点
    @return DISTANCE——返回 GRAPH_ERROR 表示参数错误；返回 GRAPH_NOT_PATH
                表示没有路径；成功时返回两点间最短路径的总距离
*/
```

```
DISTANCE Graph_GetShortDistance(LPGRAPH pGraph,
                                LPGRAPHNODE pSrcNode,
                                LPGRAPHNODE pTagNode)
{
    INT i, k； /* 循环变量 */
    INT nCalCount；/* 用来记录已算好到 pSrcNode 最短路径的节点数量*/
    GRAPHNODE **ppNode；/* 用来存放已算好到 pSrcNode 最短路径的节点数组*/
    DISTANCE nTagDis； /* 用来保存 pTagNode 到 pSrcNode 最短路径的距离值 */
    nTagDis =－1； /* 初始化为－1, 即初始时 pTagNode 和 pSrcNode 还没算出路径 */
    /* 参数校验 */
    if ( NULL == pGraph || NULL == pSrcNode || NULL == pTagNode )
    {
        return GRAPH_ERROR；
    }
    ppNode = (GRAPHNODE **)malloc(pGraph->nNodeCount *sizeof(GRAPHNODE *))；
    if ( NULL == ppNode )
    {
        return GRAPH_ERROR；
    }
    memset(ppNode, 0, pGraph->nNodeCount *sizeof(GRAPHNODE *) )；
    for ( i = 0；i < pGraph->nNodeCount；i++ )
    {
        pGraph->ppNodeArray[i]->nMagic =－1；/* 初始化为－1，表示还没有计算过 */
        pGraph->ppNodeArray[i]->pMagic = NULL；
    }
    ppNode[0] = pSrcNode；
    ppNode[0]->nMagic = 0；/*初始化为 0，表示 pSrcNode 到 pSrcNode 的距离为 0*/
    ppNode[0]->pMagic = NULL；
    nCalCount = 1； /* 初始化为 1，表示 pSrcNode 这 1 个节点已计算完 */
    /* k 从 1 开始循环是因为 ppNode[0]已经不需计算 */
    for ( k = 1；k < pGraph->nNodeCount；k++ )
    {
        /* 用来保存所有未计算节点中到 pSrcNode 节点最短距离值 */
        DISTANCE nTotalDis；
        /* 用来表示目标节点到 pSrcNode 最短路径上和目标节点相邻的节点 */
```

```
GRAPHNODE *pSNode；
/* 用来表示所有未计算节点中到 pSrcNode 节点最短距离的节点 */
GRAPHNODE *pTNode；
pSNode = NULL；
pTNode = NULL；
nTotalDis = 0；
for ( i = 0；i < nCalCount；i++ )
{
    /* 用来表示所有未计算节点中到某个已算好的节点的最短距离的节点 */
    GRAPHNODE *pShortNode；
    SINGLENODE *pNode；
    SINGLELIST  *pList；
    DISTANCE nMinDis；
    nMinDis = GRAPH_MAX_DISTANCE；
    pShortNode = NULL；
    pList = ppNode[i]->pEdgeList；
    pNode = pList->pHead；
    while ( pNode != NULL )
    {
        EDGE *pEdge；
        GRAPHNODE *pGraphNode；
        pEdge = (EDGE *)pNode->pData；
        if ( pEdge->pNode1 == ppNode[i] )
        {
            pGraphNode = pEdge->pNode2；
        }
        else
        {
            pGraphNode = pEdge->pNode1；
        }
        if ( pGraphNode->nMagic !=－1 )
        {
            pNode = pNode->pNext；
            continue；
        }
```

```
            if ( nMinDis > ppNode[i]->nMagic + pEdge->distance )
            {
                nMinDis = ppNode[i]->nMagic + pEdge->distance；
                pShortNode = pGraphNode；
            }
            pNode = pNode->pNext；
        }
        if ( nTotalDis == 0 )
        {
            nTotalDis = nMinDis；/* 重新初始化 nTotaldis */
            pTNode = pShortNode；
            pSNode = ppNode[i]；
        }
        else
        {
            if ( nTotalDis > nMinDis )
            {
                nTotalDis = nMinDis；
                pTNode = pShortNode；
                pSNode = ppNode[i]；
            }
        }
    } /* for ( i = 0 */
    if ( NULL != pTNode )
    {
        pTNode->nMagic = nTotalDis；
        pTNode->pMagic = pSNode；/* 在 pMagic 中存放最短路径上的相邻节点 */
        if ( pTNode == pTagNode )
        {
            /* 已经算出 pTagNode 和 pSrcNode 的最短路径 */
            nTagDis = nTotalDis；
            break；
        }
    }
    /* 已经找出一个到 pSrcNode 的最短路径 pTNode，下一轮循环需要
     * 扩大 ppNode 中的数量
```

```
                */
                ppNode[nCalCount] = pTNode；
                nCalCount++；
            } /* for ( k = 0 */
            return nTagDis；/* 返回目标节点到源节点的最短路径 */
        }
```

7.3 最小生成树算法

7.3.1 最小生成树算法的基本概念

图的生成树不是唯一的，从不同的顶点出发进行遍历，可以得到不同的生成树。有时，图中每条边都对应一个数值，这个数值称为边的权。对于连通网络 G＝(V，E)，边是带权的，因而 G 的生成树的各边也是带权的。这里把生成树各边的权值总和称为生成树的权，并把权最小的生成树称为 G 的最小生成树(Minimum Spanning Tree)。

生成树和最小生成树有许多重要的应用。令图 G 的顶点表示城市，边表示连接两个城市之间的通信线路。n 个城市之间最多可设立的线路有 n(n－1)/2 条，把 n 个城市连接起来至少要有 n－1 条线路，则图 G 的生成树表示了建立通信网络的可行方案。如果给图中的边都赋予权，而这些权可表示两个城市之间通信线路的长度或建造代价，那么，如何选择 n－1 条线路，使得通信网络线路的总长度最短或总代价最小呢？这就需要构造该图的一棵最小生成树。

下面介绍 Kruskal 最小生成树算法。

假设图 G 顶点个数为 N 个，算法描述如下。

步骤 1：选取一条权最小的边 e1，将其加入生成树中。

步骤 2：假设已经选取 k 条边加入到生成树中，在未被选取的边中选取一条边，先满足这条边和选好的 k 条边组成的子图中无环，再满足这条边的权最小，将这条边加入生成树中。

步骤 3：继续步骤 2，直到选好 N－1 条边为止。

7.3.2 最小生成树算法的过程图解

从算法的描述可以看出这是一个很简单的算法，下面通过一个简单的图来描述这个算法的详细计算过程，如图 7-6 所示。

步骤 1：选取一条权最小的边 DE，将 DE 加入生成树中，如图 7-7 中粗线所示。

步骤 2：在剩下的边中选一条权最小的为 AE，AE 和选好的 DE 没有成环，将 AE

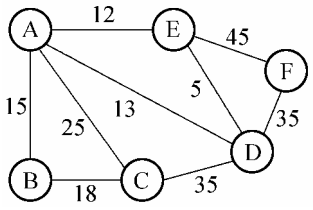


图 7-6　用作描述生成树算法的图

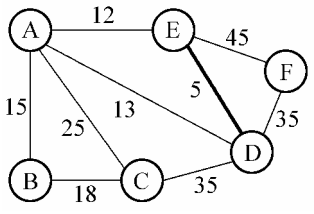


图 7-7　找出第 1 条最小边 DE

加入生成树中，如图 7-8 所示。

步骤 3：在剩下的边中选一条最小边为 AD，发现 AD 和 AE、DE 成环，因此继续选另外一条权最小的边为 AB，AB 和 AE，DE 没有成环，将 AB 加入生成树中，如图 7-9 所示。

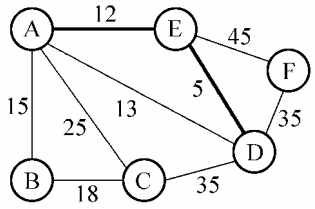


图 7-8　找出第 2 条最小边 AE

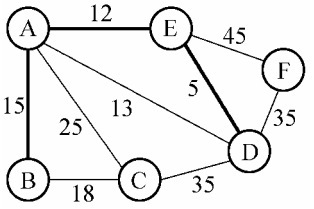


图 7-9　找出第 3 条最小边 AB

步骤 4：继续选取剩下边中和选好的生成树中的边没有成环的最小边，发现 AC 是满足和选好生成树没有成环的边，将 AC 加入生成树中，如图 7-10 所示。

步骤 5：类似前面的步骤，发现 DF 边是和已经选好的生成树没有成环的权最小的边(因为 BC，CD，AD 边都会成环)，将 DF 加入到生成树中，如图 7-11 所示。这样就得到这个图的最小生成树。

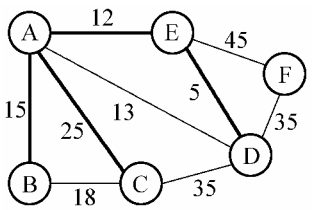


图 7-10　找出第 4 条最小边 AC

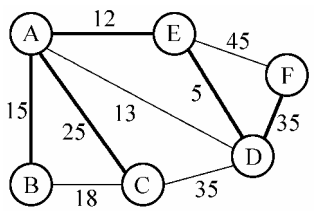


图 7-11　找出第 5 条最小边 DF

7.3.3　最小生成树的算法流程图

最小生成树算法的流程如图 7-12 所示。

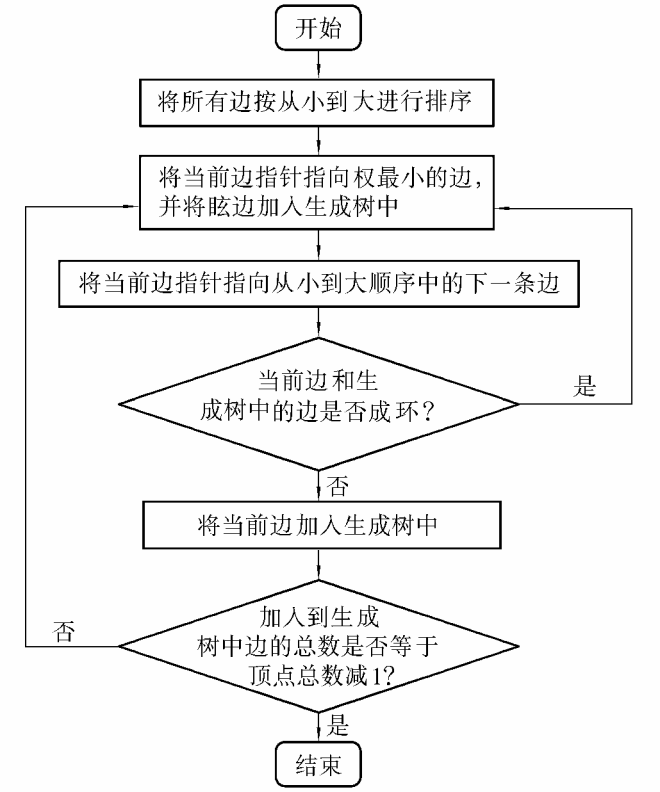


图 7-12　最小生成树算法流程图

7.3.4　最小生成树算法的编码实现

最小生成树的算法在编程时只需要先将所有边从小到大排好序，然后按从小到大顺序依次判断每条边和前面选好的边是否成环，如果没有成环则加入到生成树中，当选好 N－1 条边后，便得到最小生成树。

下面给出最小生成树的编码实现。

```
/** 利用 Kruskal 算法求最小生成树，计算过程中，节点的 nMagic 用来记录父顶点编号；
    计算完后，nMagic 为 GRAPH_EDGE_SEARCHED 的边组成一颗最小生成树
    @param LPGRAPH pGraph——要求最小生成树的图
    @return INT——返回 CAPI_FAILED 表示失败；返回 1 表示成功
*/
INT Graph_GetMinTree(LPGRAPH pGraph)
{
    SINGLENODE *pEdgeListNode；
    INT i；
```

```
INT *pNodeNo；/* 用来记录所有节点父节点号以便判断是否成环 */
if ( NULL == pGraph )
{
    return CAPI_FAILED；
}
pNodeNo = (INT *)malloc(pGraph->nNodeCount *sizeof(INT))；
if ( NULL == pNodeNo )
{
    return CAPI_FAILED；
}
/*初始化所有节点的 nMagic 为节点编号，父节点号为－1 表示还没有加入到生成树中*/
for ( i = 0；i < pGraph->nNodeCount；i++ )
{
    pGraph->ppNodeArray[i]->nMagic = i；
    pNodeNo[i] =－1；/* 初始化所有节点的父节点号，为－1 表示还没有计算过 */
}
Graph_SortEdgeInOrder(pGraph)；
/* 初始化所有边的 nMagic 为 GRPAH_EDGE_NOT_SEARCH */
pEdgeListNode = pGraph->pEdgeList->pHead；
while ( NULL != pEdgeListNode )
{
    EDGE *pEdge；
    pEdge = (EDGE *)pEdgeListNode->pData；
    pEdge->nMagic = GRAPH_EDGE_NOT_SEARCH；
    pEdgeListNode = pEdgeListNode->pNext；
}
pEdgeListNode = pGraph->pEdgeList->pHead；
while ( NULL != pEdgeListNode )
{
    EDGE *pEdge；
    INT bnf；     /* 一条边开始顶点的父节点编号 */
    INT edf；     /* 一条边结束顶点的父节点编号 */
    pEdge = (EDGE *)pEdgeListNode->pData；
    bnf = pEdge->pNode1->nMagic；/* nMagic 表示节点编号 */
    while ( pNodeNo[bnf] !=－1 )
    {
        bnf = pNodeNo[bnf]；
    }
    edf = pEdge->pNode2->nMagic；/* nMagic 表示节点编号 */
```

```
            while ( pNodeNo[edf] !=－1 )
            {
                edf = pNodeNo[edf]；
            }
            if ( bnf != edf )
            {
                /* 没有成环，将编号为 edf 的节点作为编号为 bnf 的节点的父节点 */
                pNodeNo[bnf] = edf；
                /* 将此边标志为已经搜索过，表示已放入生成树中 */
                pEdge->nMagic = GRAPH_EDGE_SEARCHED；
            }
            else
            {
                /* 已经成环，将此边标志为不在生成树中 */
                pEdge->nMagic = GRAPH_EDGE_NOT_TREE；
            }
            pEdgeListNode = pEdgeListNode->pNext；
        } /* while */
        free(pNodeNo)；
        return 1；
    }
```

7.4 深度优先搜索算法

7.4.1 深度优先搜索算法的描述

正如算法名称那样，深度优先搜索所遵循的搜索策略是尽可能“深”地搜索图。在深度优先搜索中，对于最新发现的顶点，如果它还有以此为起点而未搜索到的边，就沿此边继续搜索下去。当顶点 V 的所有边都已被搜索过，搜索将回溯到发现顶点 V 的那条边的另一个顶点。这一过程一直进行到已发现从源顶点可达的所有顶点为止。对于不连通的图，还存在未被发现的顶点，则选择其中一个作为源顶点并重复以上过程，整个过程反复进行直到所有顶点都被发现为止。为了算法的方便，通常将发现顶点 V 的那条边的另一个顶点称为顶点 V 的父节点，当从一个顶点不能发现未搜索的边时，就回溯到它的父节点继续搜索未发现的边，如果回溯到源顶点后还不能搜索到未被搜索过的边，则与此源顶点连通的所有边都被搜索过了。

简单连通图深度优先搜索算法描述如下。

步骤 1：将所有的边标志为未用过，在图中任意选取一个初始顶点 V，令 f(v)为空。

步骤 2：选取 V 的任一未访问过的相邻顶点 U，如果选取成功转步骤 3，否则转步骤 4。

步骤 3：f(u)= v，将边 e = uv、顶点 U 标志为访问过，令 u=v，重复步骤 2。

步骤 4：如果 f(v)为空则算法中止，否则令 v = f(v)，重复步骤 2。

对于非连通图，就像前面介绍的一样，只要再查找一个未被发现的顶点，不断重复以上算法，直到所有顶点被发现为止。

7.4.2 深度优先搜索算法的过程图解

下面用图解的方法来描述一个简单图的深度优先搜索过程，如图 7-13 所示，从顶点 1 开始进行深度优先搜索的一个可能的搜索过程为

1－4－8－9－7－6－5→6→7→9→8－3－2→3→8→4→1

其中，符号“→”表示回溯。

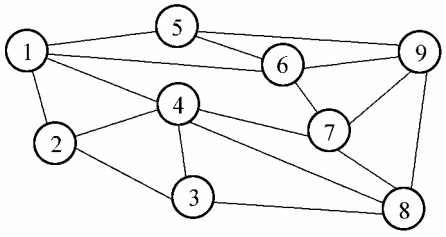


图 7-13 用作描述深度优先搜索的图

步骤 1：从 1－4－8－9－7－6－5 进行深度搜索，带箭头的粗实线表示搜索方向，如图 7-14 所示。

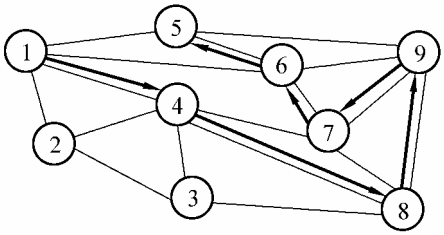


图 7-14 第 1 步搜索，依次搜索 1－4－8－9－7－6－5

步骤 2：从 5→6→7→9→8 进行回溯，带箭头的虚线表示回溯方向，如图 7-15 所示。

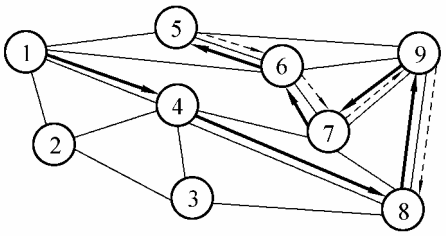


图 7-15　第 2 步搜索，从 5→6→7→9→8 进行回溯

步骤 3：按 8－3－2 方向继续进行深度搜索，如图 7-16 所示。

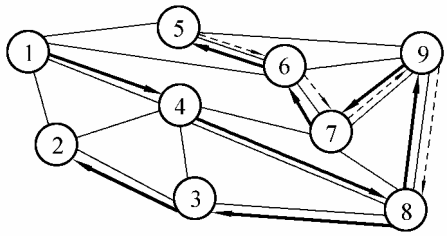


图 7-16　第 3 步搜索，按 8－3－2 方向继续进行深度搜索

步骤 4：从 2→3→8→4→1 进行回溯，回溯到顶点 1 后发现没有其他未搜索过的相邻顶点，搜索完毕，如图 7-17 所示。

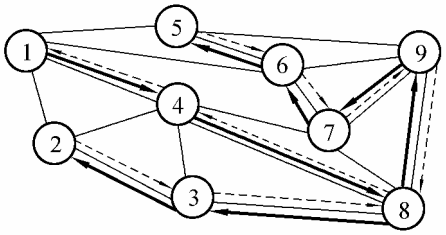


图 7-17　第 4 步搜索，从 2→3→8→4→1 进行回溯

整个深度搜索算法完成后，访问过的边组成一颗生成树，如图 7-17 中，带箭头的粗实线组成一颗生成树。

7.4.3　深度优先搜索算法的流程图

前面只是一个大概的算法描述，如果要把它变成程序，还得再细化。如何标志顶点被访问过，边被访问过，可以通过将顶点编码来表示搜索顺序。编码为 k 表示第 k 个访问的顶点，编码为 0 表示未被访问过，把这种编码称为 DFS 编码，每个顶点有一个 DFS 编码，同时每个顶点需要一个父顶点指针，该算法的详细流程如图 7-18 所示。

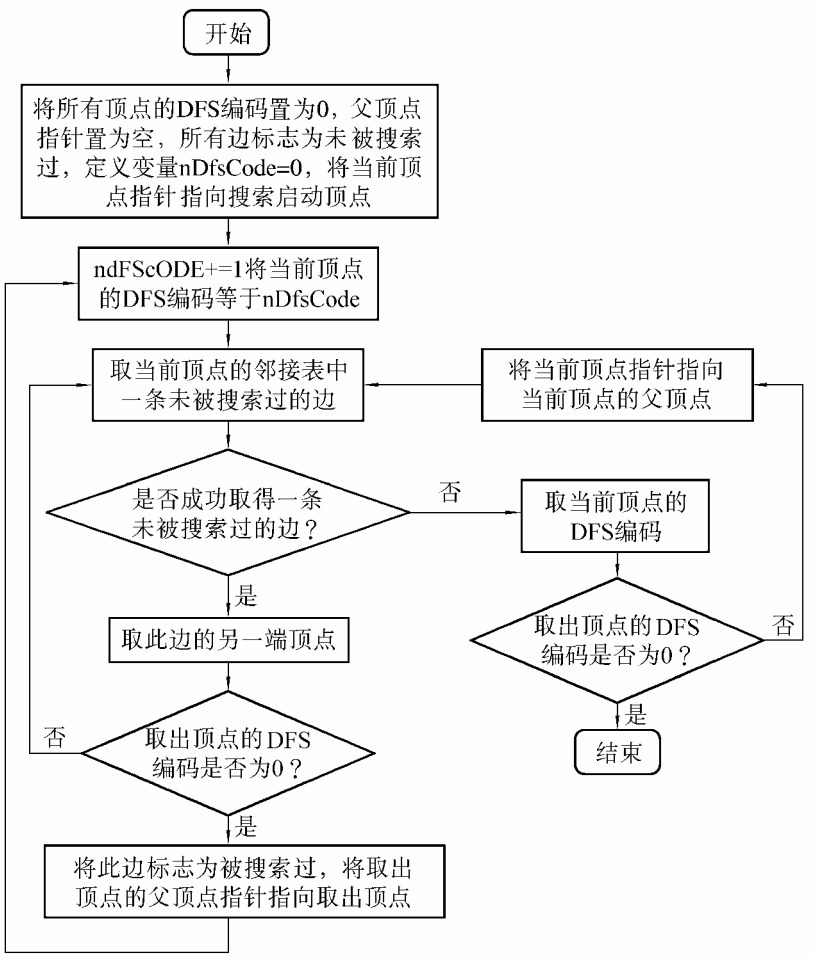


图 7-18　深度优先搜索算法流程图

7.4.4　深度优先搜索算法的编码实现

下面给出深度优先搜索的编码实现。

```
typedef INT    (*EDGEDUMPFUNC) (LPEDGE pEdge)；

/** 对图进行深度优先搜索，计算完后，每个节点的 pMagic 用来存放父节点指针；节点
    的 nMagic 用来存放 DFS 编码；边的 nMagic 用来存放是否使用的标志，搜索完后，所
    有标志为 GRAPH_EDGE_SEARCHED 的边构成一颗生成树，树的根节点为搜索起点
    @param LPGRAPH pGraph——要搜索的图
    @param LPGRAPHNODE pOrgNode——要搜索的起点，为图中的某个顶点
    @param LPEDGE pTagEdge——要搜索的指定的边，当搜索到此边后停止搜索
```

```
                                        如果为 NULL 则一直将整个图完全遍历搜索
    @param EDGEDUMPFUNC EdgeDumpFunc——边的回调函数，为 NULL 则不执行回调
    @return  INT——返回 0 表示失败；回 1 表示搜索完成；回 2 表示搜索到指定边 pTagEdge
*/
INT Graph_DepthFirstSearch( LPGRAPH pGraph, LPGRAPHNODE pOrgNode,
                            LPEDGE pTagEdge,
                            EDGEDUMPFUNC EdgeDumpFunc)
{
    LPGRAPHNODE pCurrentNode；
    INT nDfsCode；
    INT i；
    SINGLENODE *pListNode；
    EDGE *pEdge；
    INT nControlFlag；
    if ( NULL == pGraph || NULL == pOrgNode )
    {
        return CAPI_FAILED；
    }
    /* 初始化变量 */
    nControlFlag = 0；
    nDfsCode = 0；
    pCurrentNode = NULL；
    /* 初始化所有节点的 DFS 编码为 0 以及父节点指针为空 */
    for ( i = 0；i < pGraph->nNodeCount；i++ )
    {
        /* 本算法中，nMagic 被用来存放 DFS 编码 */
        pGraph->ppNodeArray[i]->nMagic = 0；
        /* 本算法中，pMagic 被用来存放父节点 */
    pGraph->ppNodeArray[i]->pMagic = NULL；
    }
    /* 初始化所有的边为 0，表示未被搜索过
     * 注：pEdge->nMagic 为 GRAPH_EDGE_NOT_SEARCH 时表示边未被搜索过
     * 为 GRAPH_EDGE_SEARCHED 表示已被搜索过
     */
    pListNode = pGraph->pEdgeList->pHead；
    while ( pListNode != NULL )
    {
        pEdge = (EDGE *)pListNode->pData；
        /* 初始化 nMagic 为 GRAPH_EDGE_NOT_SEARCH 表示此边未被搜索过 */
```

```
        pEdge->nMagic = GRAPH_EDGE_NOT_SEARCH；
        pListNode = pListNode->pNext；
    }
    pCurrentNode = pOrgNode；
    /* 将所有节点的邻接表当前位置放置在链表头部以便于后面从头搜索 */
    for ( i = 0；i < pGraph->nNodeCount；i++ )
    {
        SingleList_EnumBegin(pGraph->ppNodeArray[i]->pEdgeList)；
    }
    for (；；)
    {
        void    *pData；
        if ( nControlFlag == 0 )
        {
            /* 步骤 2：给顶点标上 DFS 编码 */
            nDfsCode++；
            pCurrentNode->nMagic = nDfsCode；
        }
        else
        {
            nControlFlag = 0；
        }
        pEdge = NULL；
        /* 步骤 3：查找未被搜索过的关联边 */
        while ((pData = SingleList_EnumNext(pCurrentNode->pEdgeList)) != NULL)
        {
            pEdge = (EDGE *)pData；
            if ( pEdge->nMagic == GRAPH_EDGE_NOT_SEARCH )
            {
                break；
            }
        }
        if ( pData == NULL )
        {
            pEdge = NULL；
        }
        if ( NULL != pEdge ) /* 判断是否找到未被搜索过的边 */
        {
            GRAPHNODE     *pNextNode；
```

```
/* 步骤 4：已经找到了一条未被搜索过的关联边 */
/* 取出当前边的相邻节点放到 pNextNode 中 */
if ( pEdge->pNode1 == pCurrentNode )
{
    pNextNode = pEdge->pNode2；
}
else
{
    pNextNode = pEdge->pNode1；
}
/* 判断相邻节点的 DFS 编码是否为 0 */
if ( pNextNode->nMagic == 0 )
{
    pEdge->nMagic = GRAPH_EDGE_SEARCHED；/* 将此边标志为被搜索过 */
    /* 将当前节点作为搜索到的节点的父节点 */
    pNextNode->pMagic = pCurrentNode；
    /* 将当前节点指针指向搜索到的相邻节点 */
    pCurrentNode = pNextNode；
    nControlFlag = 0；
    if ( EdgeDumpFunc )
    {
        (*EdgeDumpFunc)(pEdge)；
    }
    continue；   /* 转步骤 2 */
}
else
{
    /* 此边已经搜索过但不在生成树中 */
    pEdge->nMagic = GRAPH_EDGE_NOT_TREE；
    if ( EdgeDumpFunc )
    {
        (*EdgeDumpFunc)(pEdge)；
    }
    nControlFlag = 1；
    /* 转步骤 3 */
}
if ( pEdge == pTagEdge)
{
    return 2；
```

```
                }
            }
            else
            {
                /* 步骤 5：如果当前节点 DFS 编码为 1，即到了 pOrgNode，则搜索完毕需中止*/
                if ( pCurrentNode->nMagic == 1 )
                {
                    break；
                }
                /* 步骤 6：回溯到父节点转步骤 3 继续查找 */
                pCurrentNode = pCurrentNode->pMagic；
                nControlFlag = 1；
                continue；/* 转步骤 3 */
            }
        }
        return 1；
    }
```

7.5 宽度优先搜索算法

7.5.1 宽度优先搜索算法的描述

宽度优先搜索是另外一种对图的遍历方法，与深度优先搜索不同的是宽度优先搜索是从宽度方向进行搜索而不是尽可能深地搜索。如图 7-19 所示，从顶点 1 开始，搜索的顺序为：先搜索顶点 1 所有相邻的顶点 2，4，6，5；再搜索顶点 2 的相邻顶点 3，顶点 4 的相邻顶点 8，7，顶点 6 的相邻顶点 9；再搜索顶点 3，5，8，7，9 的相邻顶点，发现搜索不到未被搜索的顶点，则搜索完毕，整个搜索顺序为 1－2－4－6－5－3－8－7－9。图 7-19 中粗实线表示整个搜索过程。

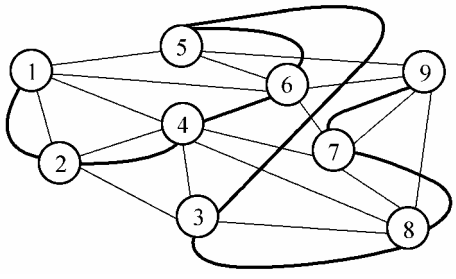


图 7-19 宽度优先搜索过程示意图

宽度优先搜索算法描述如下。

步骤 1：选取一个顶点 V，将其标号为 0，令 k = 0。

步骤 2：如果标号为 k 的顶点的相邻顶点全部都已经标号，转步骤 4，否则转步骤 3。

步骤 3：对任意标号为 k 的顶点 U，查找顶点 U 所有的关联边，如果关联边的另外一个顶点未标号的话，则将此边标志为用过，同时给那个顶点标号为 k+1，当所有的标号为 k 的顶点都被搜索完后，令 k=k+1，重复步骤 2。

步骤 4：算法结束。

算法结束后，各顶点的编码为 BFS 编码。如果图是连通的，则所有用过的边组成一颗生成树，如图 7-20 所示，图中标出了图 7-19 在搜索结束时各顶点的 BFS 编码及用过的边，其中粗实线表示搜索结束时所有用过的边，它们组成了一颗生成树。

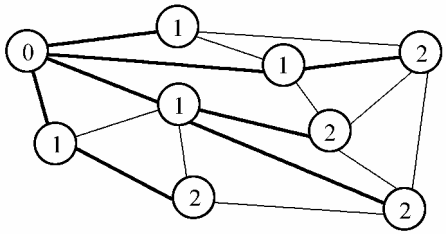


图 7-20　宽度优先搜索的生成树

7.5.2　宽度优先搜索算法的编码实现

下面给出宽度优先搜索的编码实现。

```
typedef INT    (*EDGEDUMPFUNC)(LPEDGE pEdge)；
typedef INT    (*GRAPHNODEDUMPFUNC)(LPGRAPHNODE pGraphNode)；

/** 图的宽度优先搜索函数，搜索完后，节点的 nMagic 用来存放 BFS 编码，节点的 pMagic
    用来存放下一个被搜索到节点的指针，因此由 pMagic 可以得到搜索的节点顺序，边
    的 nMagic 用来存放是否被搜索的标志，所有标志为 GRAPH_EDGE_SEARCHED 的边
    构成一颗生成树，树的根节点为搜索起点
    @param LPGRAPH pGraph——要搜索的图
    @param LPGRAPHNODE pOrgNode——要搜索的起始节点
    @param LPGRAPHNODE pTagNode——要搜索的目标节点
    @param GRAPHNODEDUMPFUNC GraphNodeDumpFunc——节点回调函数
    @param EDGEDUMPFUNC EdgeDumpFunc——边回调函数
    @return INT——返回 0 表示失败；返回 1 表示搜索完成；返回 2 表示搜索到指定目标节
                  点 pTagNode
*/
```

```
INT Graph_BreadFirstSearch( LPGRAPH pGraph, LPGRAPHNODE pOrgNode,
                            LPGRAPHNODE pTagNode,
                            GRAPHNODEDUMPFUNC GraphNodeDumpFunc,
                            EDGEDUMPFUNC EdgeDumpFunc)
{
    INT nBfsCode；                  /* BFS 编码 */
    GRAPHNODE *pCurrentNode；       /* 记录当前要操作的节点 */
    GRAPHNODE *pLastNode；          /* 记录最后一个要操作的节点 */
    INT i；
    SINGLENODE *pTmpNode；
    for ( i = 0；i < pGraph->nNodeCount；i++ )
    {
        /* nMagic 在此算法中用来存放节点的 BFS 编码 */
        pGraph->ppNodeArray[i]->nMagic =－1； /* 初始化为－1 表示还没有编码 */
        /* pMagic 在此算法中用来将已搜索到但未对本节点进行搜索的节点形成链表*/
        pGraph->ppNodeArray[i]->pMagic = NULL；
    }
    pTmpNode = pGraph->pEdgeList->pHead；
    while ( NULL != pTmpNode )
    {
        EDGE *pEdge；
        pEdge = (EDGE *)pTmpNode->pData；
        pEdge->nMagic = GRAPH_EDGE_NOT_SEARCH；
        pTmpNode = pTmpNode->pNext；
    }
    nBfsCode = 0； /* 初始化为 0 */
    pCurrentNode = pOrgNode；
    pCurrentNode->nMagic = nBfsCode； /* nMagic 用来存放 BFS 编码 */
    pLastNode = pOrgNode；
    while ( NULL != pCurrentNode )
    {
        SINGLENODE *pListNode；
        nBfsCode = pCurrentNode->nMagic + 1；
        pListNode = pCurrentNode->pEdgeList->pHead；
        while ( NULL != pListNode )
        {
            EDGE *pEdge；
            GRAPHNODE *pNode；
            pEdge = (EDGE *)pListNode->pData；
```

```
            pListNode = pListNode->pNext；
            if ( pEdge->nMagic != GRAPH_EDGE_NOT_SEARCH )
            {
                /* 边已经被搜索过了 */
                continue；
            }
            if ( pEdge->pNode2 == pCurrentNode )
            {
                pNode = pEdge->pNode1；
            }
            else
            {
                pNode = pEdge->pNode2；
            }
            if ( pNode->nMagic ==－1 )  /* 等于－1 表示节点还未编码 */
            {
                pNode->nMagic = nBfsCode；
                pLastNode->pMagic = pNode；
                pLastNode = pNode；
                pEdge->nMagic = GRAPH_EDGE_SEARCHED；
                if ( GraphNodeDumpFunc )
                {
                    /* 调用节点的回调函数 */
                    (*GraphNodeDumpFunc)(pNode)；
                }
                if ( pNode == pTagNode )
                {
                    return 2；
                }
            }
            else
            {
                pEdge->nMagic = GRAPH_EDGE_NOT_TREE；
            }
            /* 调用边的回调函数 */
            if ( EdgeDumpFunc )
            {
                (*EdgeDumpFunc)(pEdge)；
            }
        }
```

```
            /* 将当前节点指向下一个未搜索节点 */
            pCurrentNode = pCurrentNode->pMagic；
        }
        return 1；
    }
```

7.6 无环有向图的分层算法

7.6.1 无环有向图的分层算法描述

前面几节介绍了图的一些常用基本算法，本节要介绍一个无环有向图的分层算法。读者也许会问：“为什么要讲这个算法呢？在实际中好像从来没用到过这个算法呀”。软件设计同仁一定知道：一个好的软件的模块依赖关系图应该是一种分层的依赖关系图，一个好的依赖关系图应该是无环的。这两者间有没有什么联系呢？本节将告诉你一个有趣的结论：无环有向图是可分层的图。下面就构造一个算法来证明这个有趣的结论。

要证明这个结论，先要简单介绍图论中的一些概念和定义。

定义 1　把有向图各边的方向去掉，所得到的无向图称为该有向图的底图，当有向图 G 的底图是连通图时，称有向图 G 为弱连通有向图。

定义 2　如果有向图 G 无环，也没有端点相同、方向相同的边，那么称有向图 G 为严格有向图。

如果用图论的方法来描述模块依赖关系图，可以把无环的模块依赖关系图转化成一个弱连通的严格有向图。

定义 3　如果有向图 G 可以分成多个层，使得不同层中的顶点满足图中任意一个有向边的起点所在层比终点所在层高，并且对任意一个顶点，存在一条以此顶点为尾点的边，这条边的起点刚好比终点高一层，那么称图 G 是严格分层图。

由图论的知识知道，如果一个有向图中的所有顶点都有进入边，那么在这个图中存在有向环，因此对一个无环的图可以构造以下算法来将图进行分层。

步骤 1：对任一顶点 V，如果满足 d－(v) = 0, 则将 V 放入第 1 层，同时编号为 1。

步骤 2：假设已经将第 k 层顶点选好，在由第 k 层顶点形成的子图中，对任意顶点 V，如果满足 d－(v)＞0，则将顶点 V 放入第 k + 1 层中，同时编号为 k+1；对未分层的剩下的任一顶点 U，如果能找到一条以 U 为终点，以第 k 层的某一顶点为起点的有向边，则将顶点 U 放入第 k + 1 层中，同时编号为 k + 1。

步骤 3：将 k = k + 1，继续步骤 2，直到步骤 2 不能继续执行则完成对所有顶点的分层。

由于知道图中无环，因此步骤 1 中选出的放到第 1 层的顶点数是大于 0 的，且步

骤 1 是可行的；在步骤 2 中，由于图中无环，因此剩下的在第 k 层中的顶点数大于 0；由于图是有限的，因此经过有限的步骤后算法必然会结束。现在需要证明算法结束后所有的顶点都已经分好了层，使用反证法，假设有若干个顶点没有分好层，考察这个顶点组成的子图，由无环性知道这个子图中至少存在一个顶点 V 的进入边数为 0，但由步骤 1 知道所有进入边数为 0 的顶点都已经被选择进行了分层，所以顶点 V 必然有从已经分好层的顶点为起点的进入边，由步骤 2 知道，顶点 V 被选择到了分层中，这与顶点 V 不在分层中矛盾，算法的正确性得到证明。

由以上的算法可以得出以下重要而实用的定理。

定理　有向图 G 是严格分层图的充要条件是有向图 G 是弱连通的严格有向图。

如果把这个定理用在软件的模块依赖关系图上，就可以得出以下推论。

推论　一个好的模块依赖关系图应该是一个严格分层图。

7.6.2　无环有向图的分层算法过程图解

为了对上述分层算法有一个好的认识，下面用图解法来描述它。

假设待分层的例图如图 7-21 所示。

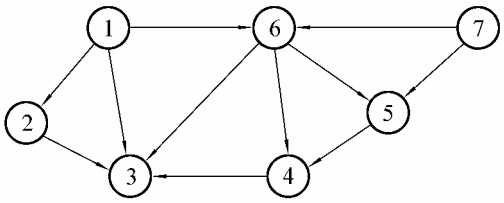


图 7-21　用作分层算法的实例图

步骤 1：将 1 和 7 这两个没有进入边的顶点放入第 1 层，其他顶点暂不放入分层。如图 7-22 所示。

步骤 2：将未分层的顶点构成的子图，找出进入度数为 0 的顶点 2 和 6，放入第 2 层中，其他顶点未分层。如图 7-23 所示。

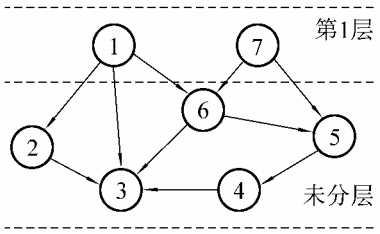


图 7-22　找出第 1 层的顶点

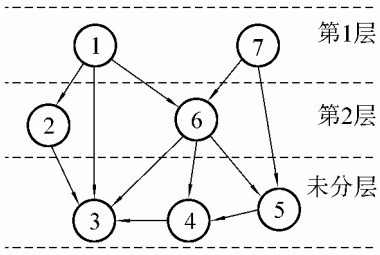


图 7-23　找出第 2 层的顶点

步骤 3：在未分层顶点构成的子图中，找到进入度数为 0 的顶点 5 放入第 3 层，其他顶点未分层。如图 7-24 所示。

步骤 4：在未分层的顶点构成的子图中，找到进入度数为 0 的顶点 4 放入第 4 层中，剩下的顶点 3 依此类推放入第 5 层中，如图 7-25 所示，至此分层完毕。

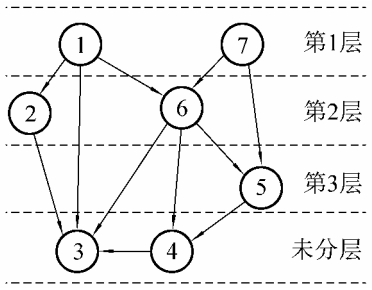


图 7-24　找出第 3 层的顶点

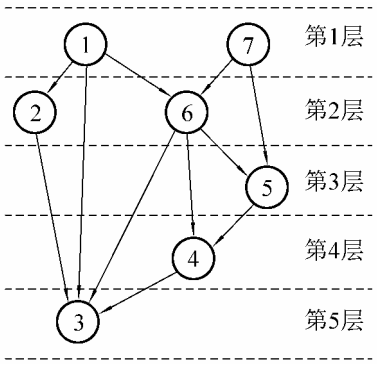


图 7-25　找出第 4 层和第 5 层的顶点

一个好的模块依赖关系图是严格分层图，那么是否满足严格分层图的模块依赖关系图就一定是好的模块依赖关系图呢？答案是否定的。严格分层图仅仅是好的模块依赖关系图的必要条件之一，并非充分条件。一个好的模块依赖关系图除了是严格分层图外，还有很多其他必要条件，这些内容超出本书范畴，应放到软件设计方面的书籍中去阐述。

7.7　哈密顿圈算法

7.7.1　哈密顿圈算法的描述

要给哈密顿圈找一个多项式复杂度的算法是一个难题，到目前为止还没有人找到 NP 问题的解法，也没有人能证明 NP 问题无解，因此给这些 NP 问题设计近似算法也是一件很有意义的事情，在本节中就来给哈密顿圈构造一个带上界的近似算法。

假设图的顶点个数为 N，每个顶点有顶点指针、编号、访问次数。

步骤 1：初始化所有的顶点编号为 0，访问次数为 0。

步骤 2：任意选取一个顶点，将其编号置为 1，访问次数置为 1，然后从这个编号为 1 的顶点任意选取 1 个相邻顶点，将其编号置为 2，访问次数置为 1。

步骤 3：假设已经选取了 n 个顶点(1< n < N)，此时选取编号为 n 的顶点的任意一个编号为 0 的且访问次数最小的相邻顶点，如果选取成功则转步骤 4，否则转步骤 5。

步骤 4：将步骤 3 中取出的顶点编号置为 n+1，访问次数加 1，如果 n+1 等于 N 则转步骤 7，否则令 n = n+1，继续转步骤 3。

步骤 5：取顶点 n 的相邻顶点中除 n－1 外访问次数最小的顶点里编号最大的顶点，如果未取到则转步骤 6，否则，假设取得的那个最大编号顶点的编号为 k，将原来顶点编号置为 k+1,…，n－1 的顶点重新编号为 0，将编号为 n 的顶点重新编号为 k+1，令 n = k+1，将编号为 k 的顶点的访问次数加 1，继续转步骤 3。

步骤 6：表示图中不存在哈密顿圈，退出。

步骤 7：判断编号为 N 的顶点是否和编号为 1 的顶点相邻，如果相邻则转步骤 8，如果不相邻，令 n = N，转步骤 5。

步骤 8：已经成功得到一条哈密顿圈，退出。

7.7.2 哈密顿圈算法的过程图解

图 7-26 是一个用来计算哈密顿圈的实例图，顶点中前一个数字表示编号，后一个数字表示访问次数。

步骤 1：从图中取一个顶点(位于左上角)，将其编号置为 1，访问次数加 1，然后再取一个编号为 0 的相邻顶点，将其编号置为 2，访问次数加 1，如图 7-27 所示。

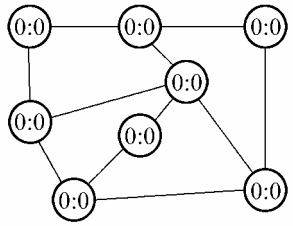


图 7-26 用作哈密顿圈算法的实例图

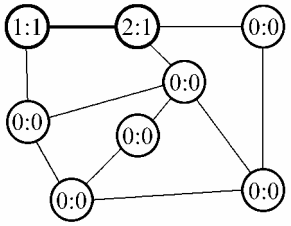


图 7-27 哈密顿圈搜索过程图 1

步骤 2：从编号为 2 的顶点取一个编号为 0 的相邻顶点，将其编号置为 3，访问次数加 1，如图 7-28 所示。

步骤 3：从编号为 3 的顶点取一个编号为 0 的相邻顶点，将其编号置为 4，访问次数加 1；从编号为 4 的顶点取一个编号为 0 的相邻顶点，将其编号置为 5，访问次数加 1；从编号为 5 的顶点取一个编号为 0 的相邻顶点，将编号置为 6，访问次数加 1。如图 7-29 所示。

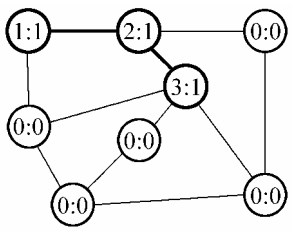


图 7-28 哈密顿圈搜索过程图 2

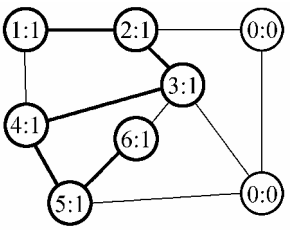


图 7-29 哈密顿圈搜索过程图 3

步骤 4：从编号为 6 的顶点取一个编号为 0 的相邻顶点，无法取得编号为 0 的顶点，此时要执行算法描述中步骤 5 的操作，将编号为 4，5 的顶点重新置为 0，编号为 6 的顶点改为编号 4，如图 7-30 所示。

步骤 5：从编号为 4 的顶点取一个编号为 0 的相邻顶点，将其编号置为 5，访问次数加 1；从编号为 5 的顶点取一编号为 0 的相邻顶点，将编号置为 6，访问次数加 1；从编号为 6 的顶点取一个编号为 0 的相邻顶点，将编号置为 7，访问次数加 1。如图 7-31 所示。

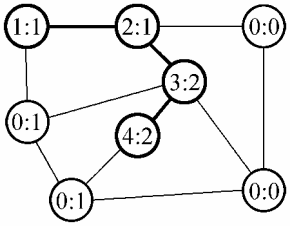


图 7-30　哈密顿圈搜索过程图 4

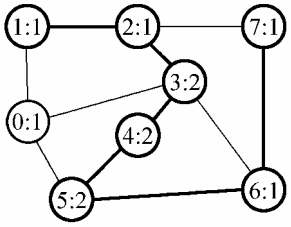


图 7-31　哈密顿圈搜索过程图 5

步骤 6：从编号为 7 的顶点取一个编号为 0 的相邻顶点，无法取得编号为 0 的顶点，此时要执行算法描述中步骤 5 的操作，将编号为 3，4，5，6 的顶点重新置为 0，编号为 7 的顶点改为编号 3，如图 7-32 所示。

步骤 7：从编号为 3 的顶点取一个编号为 0 的相邻顶点，将其编号置为 4，访问次数加 1；从编号为 4 的顶点取一编号为 0 的相邻顶点，将编号置为 5，访问次数加 1；从编号为 5 的顶点取一个编号为 0 的相邻顶点，将编号置为 6，访问次数加 1；从编号为 6 的顶点取一个编号为 0 的相邻顶点，将编号置为 7，访问次数加 1；从编号为 7 的顶点取一个编号为 0 的相邻顶点，将编号置为 8，访问次数加 1。如图 7-33 所示。

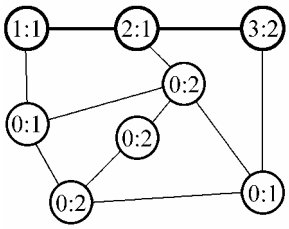


图 7-32　哈密顿圈搜索过程图 6

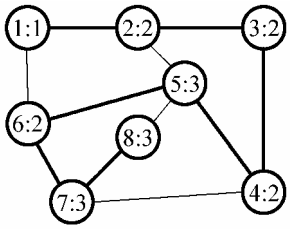


图 7-33　哈密顿圈搜索过程图 7

步骤 8：由于顶点总数为 8，因此这时要执行算法描述中步骤 7 的操作，需要判断编号为 8 的顶点是否和编号为 1 的顶点相邻，由于不相邻，此时要执行算法描述中步骤 5 的操作，将编号为 6、7 的顶点重新置为 0，将编号为 8 的顶点改为编号 6，如图

7-34 所示。

步骤 9：从编号为 6 的顶点取一个编号为 0 的相邻顶点，将其编号置为 7，访问次数加 1；从编号为 7 的顶点取一编号为 0 的相邻顶点，将编号置为 8，访问次数加 1。由于编号已经等于顶点总数，此时需要执行算法描述中步骤 7 的操作，判断顶点 8 是否和顶点 1 相邻，刚好顶点 8 和顶点 1 相邻，表示已取得 1 个哈密顿圈，如图 7-35 所示。

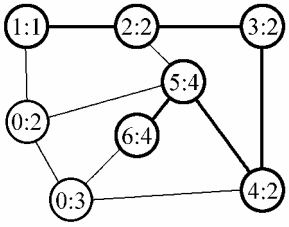


图 7-34　哈密顿圈搜索过程图 8

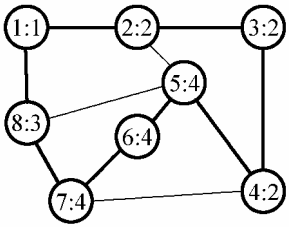


图 7-35　哈密顿圈搜索过程图 9

至此，就完成了一个哈密顿圈的算法过程演示，图中编号为 1～8 的顶点刚好组成了一个哈密顿圈。

本 章 小 结

本章主要介绍了图的几个最常用算法：Dijkstra 最短路径算法、最小生成树算法、深度优先搜索算法和宽度优先算法，这几个算法在实际中有着非常广泛的应用，均给出了完整的编码实现，属于需要重点掌握的内容。之后介绍了无环有向图的分层算法及哈密顿圈的算法，通过学习无环有向图的分层算法，可以对软件设计有所帮助，对哈密顿圈算法的学习可以让读者初步了解如何设计 NP 问题近似算法。

习题与思考

1. 画出宽度优先搜索算法的流程图。
2. 编程实现无环有向图的分层算法。
3. 编程实现哈密顿圈算法。

8

多任务算法

本章主要介绍通用的多任务基本算法，包括多任务下的遍历算法和退出算法，通过这两个算法可以将任意数据结构变成支持多任务结构；介绍了使用这两个算法实现多任务链表和多任务二叉树；还介绍了多任务下常用的读写锁算法和采用介绍多任务链表来实现的消息队列；最后介绍了线程池调度的实现实例。

20 世纪 90 年代后期，随着 Internet 网的兴起，多任务方面的应用如雨后春笋。现在的商业软件中很难找到不需要多任务的。目前单核 CPU 的速度已经接近极限，以后大都采用多核 CPU，对多任务方面的要求就更高了。以往的数据结构和算法书籍对多任务算法方面几乎没有涉及，本章主要介绍一些多任务方面的算法，并探讨诸如链表、树等数据结构在多任务下的各种操作以及多任务下的一些常见问题的解决办法。

8.1 读写锁

8.1.1 读写锁概念的引出

在开发服务器软件时往往会面临这样的困惑：该服务器软件里有一个块内存同时要被多个任务访问，有些任务是读取，有些任务既要读又要写，由于内存块很大，可能超过 1 M，用互斥信号量对内存块进行保护，即当一个任务在访问这块内存时，其他任务将被挂起一个比较长的时间，特别是对于网络任务的处理，如果在对内存操作时被挂起将严重影响网络性能，有没有办法解决这类问题呢？

在多任务中，经常会对一些需要由多个任务进行读写操作的数据进行互斥保护，

可以有效地阻止重入问题。如果要保护的数据很大，当一个任务获取对数据的操作权时，由于互斥保护，其他任务都不能对这些数据进行操作，必须等那个获得操作权的任务完成对数据的操作，数据量大就使得其他任务需要等很长时间，这就影响软件性能。多个任务同时读数据时是不会有问题的，但是如果有任务在执行写操作时不允许其他任务读或写，因此解决这个问题的一个有效办法便是允许多个任务同时读，但是在有任务进行写操作时，其他任务都不能操作，其他任务必须等待写任务操作完，这就是本节读写锁的思想。

8.1.2 读写锁算法的分析和实现

1. 读写锁算法的分析

读写锁算法主要需要实现对共享资源访问时，可以有多个任务同时进行读操作，但在同一时间内只能有一个任务进行写操作，并且在写操作时，其他任务也都不能读。下面就来分析一下如何实现读写锁。

在保护读操作的时候，要实现多个读操作可以同时进行，但是读操作时不能有写操作，因此读操作发生时要有一把锁来锁住写操作，这就可以当有写操作要发生时，由于已锁住了写操作，写操作就会阻塞。当读操作全部结束时，需要将锁住的写操作解锁以便进行写操作。

要判断读操作是否全部结束，需要使用一个计数器来进行计数，当有读操作发生时，计数加 1；读操作结束时，计数减 1；计数为 0 时表示此时没有正在进行的读操作，此时需要对写操作进行解锁。并不需要每次读操作发生时都锁住写操作，只要对第 1 个读操作上锁就可以了。计数器的值如果为 1，则表示需要给写操作上锁，因此整个读写锁中需要一把写操作锁、一个计数器变量，另外为了保护计数器变量的读写，还需要另外再设一把锁，可以将读写锁的数据结构用 C 语言结构体描述如下。

```
typedef struct RWLOCK_st {
    LOCK ReadLock；     /* 读操作锁，主要用来保护计数器变量的读写 */
    LOCK WriteLock；    /* 写操作锁 */
    UINT uReadCount；   /* 正在进行的读操作的数量 */
} RWLOCK；
```

可以设计下列四个基本操作来保护数据的读写。

① 读操作锁 RWLock_LockRead( )：读操作前要调用这个函数；

② 读操作解锁 RWLock_UnlockRead( )：读操作完成后要调用这个函数；

③ 写操作锁 RWLock_LockWrite( )：写操作前要调用这个函数；

④ 写操作解锁 RWLock_UnlockWrite( )：写操作完成后要调用这个函数。

2. 读写锁算法的实现

1) 读操作保护的实现伪代码如下。

```
RWLock_LockRead()
{
    上锁锁住计数器变量的读写
    计数器加 1
    if ( 计数器的值为 1 )
    {
        上锁锁住写操作
    }
    解锁计数器变量的读写
}
RWLock_UnlockRead()
{
    上锁锁住计数器变量的读写
    计数器减 1
    if ( 计数器的值为 0 )
    {
        解锁写操作
    }
    解锁计数器变量的读写
}
```

2) 写操作保护的实现伪代码如下。

```
RWLock_LockWrite()
{
    上锁锁住写操作
}
RWLock_UnlockWrite()
{
    解锁写操作
}
```

写操作保护非常简单，就像普通的互斥量保护一样，只是一个简单的上锁和解锁操作。

8.1.3 读写锁的编码实现

读写锁除前面介绍的四个操作外，还应增加读写锁的创建和释放函数。下面给出

读写锁完整的编码实现。

```
/** 读写锁的创建函数，负责创建一个读写锁对象
    @return RWLOCK *——成功返回 RWLOCK 指针；失败返回 NULL
*/
RWLOCK *RWLock_Create(void)
{
    RWLOCK *pRWLock；
    pRWLock = (RWLOCK *)malloc(sizeof(RWLOCK))；
    if ( pRWLock != NULL )
    {
        pRWLock->ReadLock = LockCreate()；
        if ( pRWLock->ReadLock != NULL )
        {
            pRWLock->WriteLock = LockCreate()；
            pRWLock->uReadCount = 0；
        }
        else
        {
            free(pRWLock)；
            pRWLock = NULL；
        }
    }
    return pRWLock；
}

/** 释放一个 RWLOCK 对象
    @param RWLOCK *pRWLock——读写锁对象指针
    @return void——无
*/
void RWLock_Destroy(RWLOCK *pRWLock)
{
    if ( pRWLock != NULL )
    {
        LockClose(pRWLock->ReadLock)；
        LockClose(pRWLock->WriteLock)；
        free(pRWLock)；
    }
}
```

```
/** 读写锁的读操作锁函数，有读操作时需要调用这个函数进行读操作保护
    @param RWLOCK *pRWLock——读写锁对象指针
    @return void——无
*/
void RWLock_LockRead(RWLOCK *pRWLock)
{
    Lock(pRWLock->ReadLock)；
    pRWLock->uReadCount += 1；
    if ( pRWLock->uReadCount == 1 )
    {
        /* 当有第 1 个读操作的时候，需要将写操作锁上防止写操作 */
        Lock(pRWLock->WriteLock)；
    }
    Unlock(pRWLock->ReadLock)；
}

/** 读写锁的读操作解锁函数，需要和 RWLock_LockRead()函数配对使用
    @param RWLOCK *pRWLock——读写锁对象指针
    @return void——无
*/
void RWLock_UnlockRead(RWLOCK *pRWLock)
{
    Lock(pRWLock->ReadLock)；
    pRWLock->uReadCount－= 1；
    if ( pRWLock->uReadCount == 0 )
    {
        /* 当没有读操作的时候，需要将写操作解锁以便其他任务可以进行写操作 */
        Unlock(pRWLock->WriteLock)；
    }
    Unlock(pRWLock->ReadLock)；
}

/** 读写锁的写操作锁函数，写操作发生前需要先调用这个函数进行写操作保护
    @param RWLOCK *pRWLock——读写锁对象指针
    @return void——无
*/
void RWLock_LockWrite(RWLOCK *pRWLock)
{
    Lock(pRWLock->WriteLock)；
```

```
}

/** 读写锁的写操作解锁函数，需要和 RWLock_LockWrite()函数配对使用
    @param RWLOCK *pRWLock——读写锁对象指针
    @return void——无
*/
void RWLock_UnlockWrite(RWLOCK *pRWLock)
{
    Unlock(pRWLock->WriteLock)；
}
```

8.2 多任务资源释放问题

8.2.1 子任务释放问题

先看一个实例，在多任务下，有一个子任务一直在对链表循环地进行遍历操作，这时候想退出主任务，如何将这条链表释放呢？如果在主任务退出时简单地将链表释放，而程序还没有退出，这时如果发生任务切换，切换到那个进行遍历操作的子任务上，这个子任务还会继续对链表进行遍历操作，但是主任务已经将链表释放了，这就会导致访问已经释放的内存而产生异常。

有必要设计一个算法来避免上述问题，分析上述例子可以发现，要使程序安全地退出，必须在释放链表前让所有对链表有操作的任务退出，这样在释放链表后就不会再有任务访问已经释放的链表了。但怎样能在释放链表前让其他对链表有操作的任务安全地退出呢？有操作系统基础的读者应该知道任务有两种：一种是进程；一种是线程。一般来说，线程如果不是正常退出会导致资源泄漏，进程非正常退出是否会造成泄漏则与具体的操作系统及具体资源有关了。虽然现在大多数操作系统都支持进程间完全隔离，但一些老的操作系统如 Win98 还是进程不隔离的，进程非正常退出会导致资源泄漏。即使进程隔离的操作系统，如果资源是全局的话，非正常退出也会导致资源泄漏。因此如果设计成任务正常退出，则不论在何种操作系统上，只要程序本身没有资源泄漏就不会有问题。

要将对链表进行操作的任务在链表释放前退出，必须让这些任务知道要进行释放操作了，可以使用一个退出标志来实现这一点，也就是在进行释放操作前先设置一个退出标志。子任务中要定期检测这个标志，如果标志表示可以退出时即退出任务。在退出任务时，要让进行释放操作的任务知道进行遍历操作的子任务已经退出了，通常可以让子任务在退出前发送一个事件通知——释放任务可以进行释放操作了。

8.2.2 多个子任务释放

实际上可能有多个子任务在对链表进行操作，如果每个子任务都在退出前发送一个事件通知给释放操作的任务，可能第一个子任务已经发送了事件通知，第二个子任务还在运行，但释放操作的任务已经收到事件通知、进行了释放操作；由于第二个通知任务还在运行，如果这时任务切换到第二个子任务上，则会导致第二个子任务访问已经释放的链表。要解决这个问题，必须保证让进行释放操作的任务在所有子任务都退出后才能释放链表，这样又引入一个问题，如何让释放操作的任务知道所有子任务都已经退出了呢？

可以回顾一下 8.1 节中读写锁的概念，读写锁操作中，要进行写操作时，必须等到所有的读操作结束，写操作的任务是通过一个计数变量来判断是否还有其他读操作任务的，如果计数为 0 表示没有其他读操作，这时就可以进行写操作了。实际上可以把释放操作和写操作对应，对链表的操作和读操作对应，释放操作的任务要知道所有对链表操作的任务是否都已经退出的问题，和读写锁中写操作任务要知道所有读操作是否都结束一样，也可以用一个计数变量来实现，计数变量初始为 0，每创建一个对链表操作的任务，计数加 1；每退出一个操作任务，计数减 1，因此在对链表操作的任务中，只要判断计数是否为 0，若为 0，就发送一个事件通知给释放操作的任务表示可以释放即可。

经过上面的分析，可以设计出多任务下资源释放的数据结构如下。

```
typedef struct MTASK_st {
    LOCK pLock；          /* 操作锁 */
    EVENT pExitEvent；    /* 退出事件 */
    UINT uExitFlag；      /* 退出标志 */
    UINT uTaskCount；     /* 操作的子任务数量 */
} MTASK；
```

8.2.3 多任务释放的实现

下面给出多任务资源释放的编码实现。

```
/** 多任务资源释放的创建函数
    @return MTASK *——多任务释放结构指针
*/
MTASK *MTask_Create()
{
    MTASK   *pMTask；
    pMTask = (MTASK *)malloc( sizeof(MTASK) )；
```

```
    if ( pMTask != NULL )
    {
        pMTask->pLock = LockCreate()；
        if ( pMTask->pLock == NULL )
        {
            free( pMTask )；
            return NULL；
        }
        pMTask->pExitEvent = EventCreate()；
        if ( pMTask->pExitEvent == NULL )
        {
            LockClose( pMTask->pLock )；
            free( pMTask )；
            return NULL；
        }
        pMTask->uTaskCount = 0；
        pMTask->uExitFlag = MTASK_NO_EXIT；
    }
    return pMTask；
}

/** 多任务资源释放的释放函数
    @param MTASK *pMTask——多任务释放结构指针
    @return void——无
*/
void MTask_Destroy(MTASK *pMTask)
{
    Lock( pMTask->pLock )；
    pMTask->uExitFlag = MTASK_EXIT；
    if ( pMTask->uTaskCount != 0 )
    {
        Unlock( pMTask->pLock )；
        WaitEvent( pMTask->pExitEvent )；
    }
    else
    {
        Unlock( pMTask->pLock )；
    }
```

```
    /* 关闭操作的锁和退出事件 */
    LockClose( pMTask->pLock )；
    EventClose( pMTask->pExitEvent )；
    free( pMTask )；
}

/** 多任务资源释放的锁操作函数
    @param MTASK *pMTask——多任务释放结构指针
    @return void——无
*/
void MTask_Lock(MTASK *pMTask)
{
    Lock( pMTask->pLock )；
}

/** 多任务资源释放的解锁操作函数
    @param MTASK *pMTask——多任务释放结构指针
    @return void——无
*/
void MTask_Unlock(MTASK *pMTask)
{
    Unlock( pMTask->pLock )；
}

/** 多任务资源释放的获取退出标志函数
    @param MTASK *pMTask——多任务释放结构指针
    @return UINT——返回 MTASK_NO_EXIT 表示不退出；返回 MTASK_EXIT 表示退出
*/
UINT MTask_GetExitFlag(MTASK *pMTask)
{
    UINT uExitFlag；
    Lock( pMTask->pLock )；
    uExitFlag = pMTask->uExitFlag；
    Unlock( pMTask->pLock )；
    return uExitFlag；
}

/** 设置退出标志函数
    @param MTASK *pMTask——多任务释放结构指针
```

```
    @param UINT uExitFlag——返回 MTASK_NO_EXIT 表示不退出；返回
                                MTASK_EXIT 表示退出
    @return void——无
*/
void MTask_SetExitFlag(MTASK *pMTask, UINT uExitFlag)
{
    Lock( pMTask->pLock )；
    pMTask->uExitFlag = uExitFlag；
    Unlock( pMTask->pLock )；
}

/** 进入操作任务函数，作用是使计数变量加 1
    @param MTASK *pMTask——多任务释放结构指针
    @return void——无
*/
void MTask_EnterTask(MTASK *pMTask)
{
    Lock( pMTask->pLock )；
    pMTask->uTaskCount += 1；
    Unlock( pMTask->pLock )；
}

/** 离开操作任务函数，作用是使计数变量减 1，当计数变量为 0 时
    发送退出事件通知，释放操作任务可以释放了
    @param MTASK *pMTask——多任务释放结构指针
    @return void——无
*/
void MTask_LeaveTask(MTASK *pMTask)
{
    Lock( pMTask->pLock )；
    pMTask->uTaskCount－= 1；
    if ( pMTask->uTaskCount == 0 )
    {
        SendEvent( pMTask->pExitEvent )；
    }
    Unlock( pMTask->pLock )；
}
```

8.3 多任务下的遍历问题

8.3.1 链表在多任务下的遍历问题

前面介绍的短消息系统，实际上是在多任务环境中运行的。如果有一个任务专门做接收动作，接收后添加到链表中，另外一个任务做反复遍历转发动作，转发成功后要删除，也就是说有一个任务在遍历链表，如果遍历到链表尾部时要重新从头继续遍历，并且遍历过程中要做转发删除操作，这种情况下如何实现保护？某工程师是这样设计的，添加操作时锁住不能遍历，并且添加在链表头部，遍历时锁住不能往链表中添加，为了提高效率每次遍历时都从头开始，直到成功转发一个为止再解锁，这确实是一个成功的解决办法，避免了在读写数据时的可重入问题，但是效率却很低，能不能找到一种更好的解决方案呢？比如说每次添加在尾部，并且遍历从头部开始，如果链表中有很多数据，实际上并不存在重入问题，只有遍历操作到尾部节点的上一节点时才存在重入问题。其实在第 3 章中已经介绍了多任务的迭代器，只要在链表的所有具有删除功能的函数中，判断一下当前删除的节点是不是要遍历的节点即可，如果是要进行遍历的节点，删除后，必须将要遍历的当前节点指针指向下一个节点，这样就避免了删除和遍历的冲突。

8.3.2 多任务链表的设计和编码实现

下面就来设计一个带遍历功能的支持多任务遍历的链表，首先要设计多任务链表的结构体，可以采用继承多任务资源释放退出算法和双向链表来实现。

```
typedef struct MTLIST_st {
    DOUBLELIST *pList；      /* 双向链表指针 */
    MTASK *pMTask；          /* 多任务退出结构指针 */
} MTLIST；
```

下面给出多任务链表的编码实现。

```
#define      MTLIST_DELETE_NODE    0
#define      MTLIST_NO_DELETE      1

/** 多任务链表的创建函数
    @return MTLIST *——多任务链表指针
    @param void——无
*/
MTLIST *MTList_Create(void)
{
```

```
    MTLIST    *pList；
    pList = (MTLIST *)malloc(sizeof(MTLIST))；
    if ( pList != NULL )
    {
        pList->pList = DoubleList_Create()；
        if ( pList->pList != NULL )
        {
            pList->pMTask = MTask_Create()；
            if ( pList->pMTask == NULL )
            {
                free(pList->pList)；
                free(pList)；
                pList = NULL；
            }
        }
        else
        {
            free(pList)；
            pList = NULL；
        }
    }
    return pList；
}

/** 多任务链表的释放函数
    @param MTLIST *pList——多任务链表指针
    @param DESTROYFUNC DestroyFunc——数据释放回调函数
    @return void——无
*/
void MTList_Destroy(MTLIST *pList, DESTROYFUNC DestroyFunc)
{
    if ( pList == NULL )
    {
        return；
    }
    MTask_Destroy(pList->pMTask)；
    DoubleList_Destroy(pList->pList, DestroyFunc)；
}
```

```
/** 获取退出标志
    @param MTLIST *pList——多任务链表指针
    @return UINT——返回 CAPI_EXIT_TASK 表示可以退出任务；返回
                    CAPI_NOT_EXIT_TASK 表示不可以退出任务
*/
UINT MTList_GetExitFlag(MTLIST *pList)
{
    return MTask_GetExitFlag(pList->pMTask)；
}

/** 设置退出标志函数
    @param MTLIST *pList——多任务链表指针
    @param UINT uExitFlag——返回 CAPI_EXIT_TASK 表示可以退出任务；返回
                            CAPI_NOT_EXIT_TASK 表示不可以退出任务
    @return void——无
*/
void MTList_SetExitFlag(MTLIST *pList, UINT uExitFlag)
{
    MTask_SetExitFlag(pList->pMTask, uExitFlag)；
}

/** 进入任务函数，在进入任务循环前要先调用这个函数
    @param MTLIST *pList——多任务链表指针
    @return void——无
*/
void MTList_EnterTask(MTLIST *pList)
{
    MTask_EnterTask(pList->pMTask)；
}

/** 离开任务函数
    @param MTLIST *pList——多任务链表指针
    @return void——无
*/
void MTList_LeaveTask(MTLIST *pList)
{
    MTask_LeaveTask(pList->pMTask)；
}

/** 插入数据到多任务链表头部的函数
```

```
    @param MTLIST *pList——多任务链表指针
    @param void *pData——要插入的数据
    @return INT——失败返回 CAPI_FAILED；成功返回 CAPI_SUCCESS
*/
INT MTList_InsertHead(MTLIST *pList, void *pData)
{
    INT nRet = CAPI_FAILED；
    if ( pList == NULL )
    {
        return CAPI_FAILED；
    }
    MTask_Lock( pList->pMTask )；
    nRet = DoubleList_InsertHead(pList->pList, pData)；
    MTask_Unlock( pList->pMTask )；
    return nRet；
}

/** 删除多任务链表头部节点函数
    @param MTLIST *pList——多任务链表指针
    @param DESTROYFUNC DestroyFunc——数据释放回调函数
    @return INT——失败返回 CAPI_FAILED；成功返回 CAPI_SUCCESS
*/
INT MTList_DeleteHead(MTLIST *pList, DESTROYFUNC DestroyFunc)
{
    void *pData；
    if ( pList == NULL )
    {
        return CAPI_FAILED；
    }
    MTask_Lock( pList->pMTask )；
    pData = DoubleList_PopHead(pList->pList)；
    MTask_Unlock( pList->pMTask )；
    if ( DestroyFunc != NULL && pData != NULL )
    {
        (*DestroyFunc)(pData)；
    }
    return CAPI_SUCCESS；
}
```

```
/** 插入数据到多任务链表尾部的函数
    @param MTLIST *pList——多任务链表指针
    @param void *pData——要插入到尾部的数据
    @return INT——失败返回 CAPI_FAILED；成功返回 CAPI_SUCCESS
*/
INT MTList_InsertTail(MTLIST *pList, void *pData)
{
    INT nRet；
    MTask_Lock( pList->pMTask )；
    nRet = DoubleList_InsertTail(pList->pList, pData)；
    MTask_Unlock( pList->pMTask )；
    return nRet；
}

/** 弹出多任务链表头部节点的函数
    @param MTLIST *pList——多任务链表指针
    @return void *——弹出的头部节点数据
*/
void *MTList_PopHead(MTLIST *pList)
{
    void *pData；
    MTask_Lock( pList->pMTask )；
    pData = DoubleList_PopHead(pList->pList)；
    MTask_Unlock( pList->pMTask )；
    return pData；
}

/** 弹出多任务链表尾部节点的函数
    @param MTLIST *pList——多任务链表指针
    @return void *——弹出的尾部节点数据
*/
void *MTList_PopTail(MTLIST *pList)
{
    void *pData；
    MTask_Lock( pList->pMTask )；
    pData = DoubleList_PopTail(pList->pList)；
    MTask_Unlock( pList->pMTask )；
    return pData；
}
```

8.3.3 多任务链表的遍历操作编码实现

多任务链表的逐个节点遍历操作和单任务下的操作完全不同，因为单任务下的逐个节点遍历操作是将数据指针返回给调用者操作，如果多任务下也采用这种方式，有可能调用者在操作数据时遇到其他任务也要操作这个数据，导致重入问题产生。

为了避免重入问题，必须在操作数据时加上锁保护，而数据返回给调用者是很难加锁的，原因是多个调用者需要使用共同的锁，这样还需要提供一个公用的锁接口供调用者使用，增加了使用的复杂度。

在多任务下访问数据最好的方法是由调用者提供一个回调函数，这样锁保护操作就可以在链表的遍历函数内部实现，调用者不必关心上锁解锁的问题。

可提供四种在多任务下遍历的方案，分别适用于不同的应用场合，实现遍历操作的初始化编码如下。

```
/** 多任务链表遍历的初始化函数
    @param MTLIST *pList——多任务链表指针
    @return void——无
*/
void MTList_EnumBegin(MTLIST *pList)
{
    if ( pList == NULL )
    {
        return；
    }
    MTask_Lock( pList->pMTask )；
    DoubleList_EnumBegin(pList->pList)；
    MTask_Unlock( pList->pMTask )；
    return；
}
```

1) 第一种遍历方案：采用传回调函数的方法，由链表遍历操作内部来进行锁保护，实现编码如下。

```
/** 多任务链表遍历的下一个节点函数
    @param MTLIST *pList——多任务链表指针
    @param VISITFUNC VisitFunc——数据访问函数
    @return INT——失败返回 CAPI_FAILED；成功返回 CAPI_SUCCESS
*/
INT MTList_EnumNext(MTLIST *pList, VISITFUNC VisitFunc)
{
```

```
    void *pData；
    INT nRet = CAPI_FAILED；
    MTask_Lock( pList->pMTask )；
    pData = DoubleList_EnumNext(pList->pList)；
    if ( VisitFunc != NULL )
    {
        nRet = (*VisitFunc)(pData)；
    }
    MTask_Unlock( pList->pMTask )；
    return nRet；
}
```

2) **第二种遍历方案：仍然采用传回调函数的方法，但在操作数据时是先将数据拷贝出来进行操作，解锁在数据操作之前完成，这种方案的好处是当操作数据需要花费很长的时间时，可以取得更好的分时效果，实现编码如下。**

```
/** 多任务链表的枚举下一个节点并将节点拷贝出来执行的函数，先访问节点，将节
    点拷贝出来，解锁后再调用 VisitFunc 来访问数据
    @param MTLIST *pList——多任务链表指针
    @param VISITFUNC VisitFunc——节点数据访问回调函数
    @param DESTROYFUNC DestroyFunc——数据释放回调函数
    @param COPYFUNC CopyFunc——数据拷贝函数
    @return INT——取决于 VisitFunc 的返回值
*/
INT MTList_EnumNextCopy( MTLIST *pList, VISITFUNC VisitFunc,
                         DESTROYFUNC DestroyFunc, COPYFUNC CopyFunc)
{
    void *pData；
    INT nRet = CAPI_FAILED；
    MTask_Lock( pList->pMTask )；
    pData = DoubleList_EnumNext(pList->pList)；
    pData = (*CopyFunc)( pData )；
    MTask_Unlock( pList->pMTask )；
    nRet = (*VisitFunc)( pData )；
    if ( DestroyFunc != NULL && pData != NULL )
    {
        (*DestroyFunc)(pData)；
    }
```

```
    return nRet；
}
```

3) **第三种遍历方案：仍然采用传回调函数的方法，这种方案在遍历的节点要被删除的情况下使用。由于遍历的节点数据在操作完后要被删除，因此可以先从链表里将遍历的节点弹出来，弹出后其他任务就访问不到这个节点了，因此弹出后对数据操作就不需要加锁保护了，实现编码如下。**

```
/** 多任务链表的枚举并且删除节点函数，先访问节点，将节点弹出来，再解锁，然
    后再用 VisitFunc 访问节点，最后将弹出的节点释放
    @param MTLIST *pList——多任务链表指针
    @param VISITFUNC VisitFunc——数据访问函数
    @param DESTROYFUNC DestroyFunc——数据释放函数
    @return INT——取决于 VisitFunc 的返回值
*/
INT MTList_EnumNextAndDelete( MTLIST *pList, VISITFUNC VisitFunc,
                              DESTROYFUNC DestroyFunc)
{
    DOUBLENODE *pNode；
    INT nRet = CAPI_FAILED；
    MTask_Lock( pList->pMTask )；
    pNode = DoubleList_EnumNode(pList->pList)；
    DoubleList_PopNode(pList->pList, pNode)；
    MTask_Unlock( pList->pMTask )；
    nRet = (*VisitFunc)( pNode->pData )；
    if ( DestroyFunc != NULL && pNode->pData != NULL )
    {
        (*DestroyFunc)(pNode->pData)；
    }
    free(pNode)；
    return nRet；
}
```

4) **第四种遍历方案：前面三种遍历方案都是每访问一个节点就要进行一次上锁和解锁操作，如果链表中元素很多时，比如有 100 万个节点，要将所有节点遍历一遍将要进行 100 万次的上锁和解锁操作，软件整体效率非常低。因此，还需要设计一种遍历方法，可以在一次上锁和解锁操作中完成对多个节点的访问，实现编码如下。**

```
/** 多任务链表的遍历下几个节点函数
    @param MTLIST *pList——多任务链表指针
    @param VISITFUNC VisitFunc——数据访问函数
    @param INT nCount——要访问的节点个数
    @return INT——失败返回 CAPI_FAILED；成功返回 CAPI_SUCCESS
*/
void MTList_EnumNextSeveral(MTLIST *pList, VISITFUNC VisitFunc, INT nCount)
{
    void *pData；
    int i；
    MTask_Lock( pList->pMTask )；
    for ( i = 0；i < nCount；i++ )
    {
        pData = DoubleList_EnumNext(pList->pList)；
        if ( VisitFunc != NULL )
        {
            (*VisitFunc)(pData)；
        }
    }
    MTask_Unlock( pList->pMTask )；
}
```

8.3.4 多个任务同时遍历的情况

前面探讨了在多任务下对链表的遍历是通过各种删除操作及遍历操作修改 pCur **指针来实现的，当只有一个遍历的任务时，不会有任何问题，但是当有多个任务同时进行遍历时，会产生什么问题呢？**

多个任务同时遍历时，由于每个任务都会去修改 pCur **指针，会导致** pCur **指针混乱，各个任务都可能无法遍历到自己想要的结果。要支持多个任务同时遍历，得修改链表的设计，为每个遍历任务提供一个** pCur **指针，因此要将** pCur **指针改成指针数组，修改后的** DOUBLELIST **结构体定义如下。**

```
typedef struct DOUBLELIST_st {
    DOUBLENODE *pHead；          /* 第 1 个节点的指针 */
    DOUBLENODE *pTail；          /* 最后 1 个节点的指针 */
    DOUBLENODE **ppCur；         /* 当前节点的指针数组 */
    UINT uCurCount；             /* 当前节点指针数组的元素个数 */
    UINT uTaskCount；            /* 正在进行遍历操作的任务个数 */
    UINT uCount；                /* 保存链表节点的个数 */
```

```
} DOUBLELIST；
```

在链表的每个删除操作中，需要判断删除的元素是否在 ppCur 数组中，如果在则需要更新数组中的对应元素。

遍历操作时，遍历开始函数需要修改，要设返回值，返回值为 ppCur 数组的下标，遍历下一个节点的函数要增加一个传 ppCur 数组下标值的参数，这样就可以实现每个任务对应 ppCur 数组中唯一的指针，避开了多个任务同时修改一个 pCur 指针的情况。支持多个任务同时遍历的具体编码实现留给读者作为练习，这里就不列出了。

8.4 多任务二叉树的设计

为了加深对多任务数据结构设计的理解，再来设计一个多任务二叉树。为方便起见，这里使用红黑树做容器，为了使得程序接口具有扩展性，在多任务二叉树的接口设计中将屏蔽掉红黑树的具体信息。多任务二叉树结构体描述如下。

```
typedef struct MTREE_st {
    RBTREE *pRBTree；
    RBTREENODE *pCursor；
    MTASK *pMTask；
} MTREE；
```

需要强调的是，不需要将 MTREE 结构体提供给调用者，这个结构体可以放在.c 源文件中，而不必放到.h 头文件中，这样设计可以充分增加软件的扩展性，比如以后想将红黑树换成 AVL 树，接口不需要做任何修改。

仍然像多任务链表一样复用 MTASK 模块的编码，下面给出多任务二叉树的编码实现。

```
/** 多任务树的创建函数
    @param void——无
    @return HANDLE——成功返回多任务树的句柄；失败返回 NULL
*/
HANDLE MTree_Create(void)
{
    MTREE *pTree；
    pTree = (MTREE *)malloc(sizeof(MTREE))；
    if ( pTree != NULL )
    {
        pTree->pMTask = MTask_Create()；
```

```
        if ( pTree->pMTask != NULL )
        {
            pTree->pRBTree = RBTree_Create()；
            if ( pTree->pRBTree != NULL )
            {
                pTree->pCursor = NULL；
            }
            else
            {
                free( pTree->pMTask)；
                free( pTree )；
                pTree = NULL；
            }
        }
        else
        {
            free( pTree )；
            pTree = NULL；
        }
    }
    return (HANDLE)pTree；
}

/** 多任务树的释放函数
    @param HANDLE hTree——多任务树的句柄
    @param DESTROYFUNC DestroyFunc——数据释放回调函数
    @return void——无
*/
void MTree_Destroy(HANDLE hTree, DESTROYFUNC DestroyFunc)
{
    MTREE *pTree；
    pTree = (MTREE *)hTree；
    if ( pTree == NULL )
    {
        return；
    }
    MTask_Destroy(pTree->pMTask)；
    RBTree_Destroy( pTree->pRBTree, DestroyFunc )；
    free( pTree )；
```

```
}

/** 多任务树的设置退出标志函数
    @param HANDLE hTree——多任务树句柄
    @param UINT uExitFlag——返回 MTASK_NO_EXIT 表示不退出任务；返回
                            MTASK_EXIT 表示退出任务
    @return void——无
*/
void MTree_SetExitFlag(HANDLE hTree, UINT uExitFlag )
{
    MTREE *pTree；
    pTree = (MTREE *)hTree；
    MTask_SetExitFlag(pTree->pMTask, uExitFlag)；
}

/** 多任务树的获取退出标志函数
    @param HANDLE hTree——多任务树句柄
    @return UINT——返回 MTASK_NO_EXIT 表示不退出任务；返回
                  MTASK_EXIT 表示退出任务
*/
UINT MTree_GetExitFlag(HANDLE hTree )
{
    MTREE *pTree；
    pTree = (MTREE *)hTree；
    return MTask_GetExitFlag(pTree->pMTask)；
}

/** 多任务树的进入任务函数
    @param HANDLE hTree——多任务树的句柄
    @return void——无
*/
void MTree_EnterTask(HANDLE hTree )
{
    MTREE *pTree；
    pTree = (MTREE *)hTree；
    MTask_EnterTask(pTree->pMTask)；
}

/** 多任务树的离开任务函数
```

```
    @param HANDLE hTree——多任务树的句柄
    @return void——无
*/
void MTree_LeaveTask(HANDLE hTree )
{
    MTREE *pTree；
    pTree = (MTREE *)hTree；
    MTask_LeaveTask(pTree->pMTask)；
}

/** 多任务树的数据插入函数
    @param HANDLE hTree——多任务树的句柄
    @param void *pData——要插入的数据指针
    @param COMPAREFUNC CompareFunc——数据比较函数
    @return INT——和 RBTree_Insert()函数的返回值相同
*/
INT MTree_Insert(HANDLE hTree, void *pData, COMPAREFUNC CompareFunc)
{
    MTREE *pTree；
    INT nRet；
    pTree = (MTREE *)hTree；
    MTask_Lock(pTree->pMTask)；
    nRet = RBTree_Insert(pTree->pRBTree, pData, CompareFunc)；
    MTask_Unlock(pTree->pMTask)；
    return nRet；
}

/** 多任务树的删除数据函数
    @param HANDLE hTree——多任务树的句柄
    @param void *pData——要删除的数据指针
    @param COMPAREFUNC CompareFunc——数据比较回调函数
    @param DESTROYFUNC DestroyFunc——数据释放回调函数
    @return INT——同 RBTree_Delete()函数
*/
INT MTree_Delete( HANDLE hTree, void *pData, COMPAREFUNC CompareFunc,
                  DESTROYFUNC DestroyFunc)
{
    MTREE *pTree；
```

```
    INT nRet；
    pTree = (MTREE *)hTree；
    MTask_Lock(pTree->pMTask)；
    nRet = RBTree_Delete(pTree->pRBTree, pData, CompareFunc, DestroyFunc)；
    MTask_Unlock(pTree->pMTask)；
    return nRet；
}

/** 多任务树的枚举开始函数
    @param HANDLE hTree——多任务树的句柄
    @return void——无
*/
void MTree_EnumBegin(HANDLE hTree)
{
    MTREE *pTree；
    pTree = (MTREE *)hTree；
    MTask_Lock(pTree->pMTask)；
    RBTree_EnumBegin(pTree->pRBTree)；
    MTask_Unlock(pTree->pMTask)；
}

/** 多任务树的枚举下一个并拷贝节点函数
    @param HANDLE hTree——多任务树的句柄
    @param COPYFUNC CopyFunc——数据拷贝回调函数
    @return void *——返回拷贝的数据
*/
void *MTree_EnumNextCopy(HANDLE hTree, COPYFUNC CopyFunc)
{
    MTREE *pTree；
    void *pData；
    pTree = (MTREE *)hTree；
    MTask_Lock(pTree->pMTask)；
    pData = RBTree_EnumNext(pTree->pRBTree)；
    pData = (*CopyFunc)(pData)；
    MTask_Unlock(pTree->pMTask)；
    return pData；
}
```

多任务二叉树遍历操作的编码这里只给出了一个，其余三个留给读者作为练习。

8.5 消息队列

8.5.1 消息队列的基本概念

消息队列是一种可以有多个任务同时向队列里发送和接收数据的队列。要实现多个任务同时向队列里收发数据，那么必须在收发操作上加上锁保护。另外还要分别分析队列为空和队列为满的情况。

1. 队列为空的情况

此时向队列发送数据是没有问题的；如果此时要从队列接收数据，由于队列中暂无数据，接收任务必须等到数据发送到队列中去，才能从队列中接收到数据。

2. 队列为满的情况

此时要从队列中接收数据是没有问题的，如果此时向队列发送数据，由于队列已满，发送任务必须等待其他任务从队列里将数据接收出去，才能将数据发送到队列中。当然也可以不限制队列的大小，也就不需要考虑队列满的情况了，不过要保证发送到队列里的数据数量的上界值不会超过系统的负荷，特别是网络软件，如果接收网络数据放入消息队列中，而不设置队列大小限制就很容易受到攻击。

8.5.2 消息队列的设计和编码实现

消息队列中，需要一个计数器来记录队列中元素的个数，而且当计数为 0 时，要将接收操作阻塞，由于 Semaphore 信号量刚好有这个特性，因此可以使用计数信号量来实现计数功能。为简化起见，先不考虑队列满的情况，假设可以一直发送数据到队列中去。这样消息队列用 C 语言结构体描述如下。

```
typedef struct MSGQUEUE_st {
    MTLIST *pList；          /*  多任务链表  */
    SEMAPHORE pSem；    /*  处理队列为空时的计数信号量*/
    INT nMaxLen；            /*  队列的最大长度  */
} MSGQUEUE；
```

1. 发送操作

发送操作只需要将数据加入到链表中，然后将计数加 1，发送操作编码如下。

```
/**  消息队列的发送函数，将数据发送到消息队列中
    @param MSGQUEUE *pQueue——消息队列指针
```

```
    @param void *pData——要发送到消息队列里的数据指针
    @return INT——同 MTList_InsertTail()函数
*/
INT MsgQueue_Send(MSGQUEUE *pQueue, void *pData)
{
    INT nRet；
    nRet = MTList_InsertTail(pQueue->pList, pData)；
    SemaRelease(pQueue->pSem, 1)； /* 将计数加 1 */
    return nRet；
}
```

2. 接收操作

接收操作中需要将计数减 1，并且当计数为 0 时要阻塞接收操作直到有数据到来，然后再从链表中将数据取出来，接收操作编码如下。

```
/** 消息队列的接收数据函数，从消息队列中接收数据出来
    @param MSGQUEUE *pQueue——消息队列指针
    @return void *——返回从消息队列中接收到的数据
*/
void *MsgQueue_Recv(MSGQUEUE *pQueue)
{
    SemaWait(pQueue->pSem)；/* 将计数减 1，计数为 0 则会阻塞住 */
    return MTList_PopHead(pQueue->pList)；
}
```

从消息队列发送和接收的实现编码来看，整个实现非常简单。下面再来实现消息队列创建操作和释放操作的编码。

3. 消息队列的创建和释放操作

消息队列创建编码比较简单，只是申请一个 MSGQUEUE 结构体，并初始化结构体成员，然后返回。创建时，计数信号量的初始计数要设为 0，因为这时队列还是空的。

```
#define   DEFAULT_MSGQUEUE_LEN      65 535

/** 消息队列的创建函数
    @param INT nMaxLen——消息队列的最大长度
    @return MSGQUEUE *——成功返回创建的消息队列指针；失败返回 NULL
*/
```

```
MSGQUEUE *MsgQueue_Create(INT nMaxLen)
{
    MSGQUEUE *pQueue；
    pQueue = (MSGQUEUE *)malloc(sizeof(MSGQUEUE))；
    if ( pQueue != NULL )
    {
        pQueue->pSem = SemaCreate(0, nMaxLen)；
        if ( pQueue->pSem != NULL )
        {
            pQueue->nMaxLen = nMaxLen；
            pQueue->pList = MTList_Create()；
            if ( pQueue->pList == NULL )
            {
                SemaClose(pQueue->pSem)；
                free(pQueue)；
                pQueue = NULL；
            }
        }
        else
        {
            free(pQueue)；
            pQueue = NULL；
        }
    }
    return pQueue；
}
```

消息队列的释放操作要比创建操作复杂一点，主要是要考虑退出的资源释放问题。释放整个消息队列前，需要通知所有阻塞在消息队列接收操作上的任务退出，因此要使用 SemRelease()**函数将计数值设到最大。**

```
/**  消息队列的释放函数
    @param MSGQUEUE *pQueue——消息队列指针
    @param DESTROYFUNC DestroyFunc——数据释放回调函数
    @return void——无
*/
void MsgQueue_Destroy(MSGQUEUE *pQueue, DESTROYFUNC DestroyFunc)
{
    if ( pQueue != NULL )
```

```
    {
        INT nMaxLen；
        if ( pQueue->nMaxLen > DEFAULT_MSGQUEUE_LEN)
        {
            nMaxLen = pQueue->nMaxLen；
        }
        else
        {
            nMaxLen = DEFAULT_MSGQUEUE_LEN；
        }
        SemaRelease(pQueue->pSem, nMaxLen )；/* 让所有阻塞的接收操作可以继续*/
        MTList_Destroy(pQueue->pList, DestroyFunc)；
        SemaClose(pQueue->pSem)；
        free(pQueue)；
    }
}
```

4. **消息队列的遍历**

由于使用了多任务链表来实现消息队列，而多任务链表可以支持多任务下的遍历，因此消息队列的遍历操作可以借助于链表的遍历功能来实现，编码如下。

```
/** 消息队列的枚举初始化函数
    @param MSGQUEUE *pQueue——消息队列指针
    @return void——无
*/
void MsgQueue_EnumBegin(MSGQUEUE *pQueue)
{
    MTList_EnumBegin(pQueue->pList)；
}

/** 消息队列的枚举下一个数据函数
    @param MSGQUEUE *pQueue——消息队列指针
    @param VISITFUNC VisitFunc——数据访问回调函数
    @return INT——同 MTList_EnumNext()函数
*/
INT MsgQueue_EnumNext(MSGQUEUE *pQueue, VISITFUNC VisitFunc)
{
    return MTList_EnumNext(pQueue->pList, VisitFunc)；
}
```

8.6 实例：线程池调度的管理

8.6.1 线程池调度管理的基本概念

在写多线程的服务器软件时，线程池调度可以说是无法回避的事情，最简单的一种调度方法就是当收到客户端需要处理的请求时，创建一个线程，在这个线程中处理完收到的这个请求，再退出这个线程。这种调度算法的缺点是每收到一个请求都要创建和退出一次线程，线程创建和退出频繁，开销太大。

在目前的商业软件如 Apache、IIS 等软件中，一般采用的调度算法是预先创建若干个线程，当收到客户端请求时，将其放入一个队列中，然后每个线程从一个队列中读取请求，处理完一个请求后接着继续从队列中读取下一个请求进行处理，这样就避免了线程的创建和退出，线程的创建只要在初始化时创建一次就可以了。

上面讨论的线程池调度算法可以说是服务器软件中的一个最基础算法，目前几乎所有的商业系统中都是采用这个算法，不论你想做一个像 ICQ 一样的即时通讯软件，还是想做一个游戏服务器，或是做一个短信息系统，抑或一个 Web 服务器，上面的线程池调度算法都是非用不可的。

从上面的描述中可以看出，线程池调度算法要用到一个队列，由于有多个任务要从这个队列中取出请求数据，因此这个队列必须要支持多任务操作。采用目前商业中使用的消息队列来实现当然是可以的，不过如果用户要实时查看当前有多少客户端连接信息时，服务器就必须暂停处理客户端请求了，连接数量少时还影响不大，一旦连接数量很大的时候就会对服务器的响应性能造成影响。所幸的是，本书中用多任务链表实现的消息队列已经解决了这个问题，下面就来看一下是如何用本书中的消息队列实现这个线程池调度算法的。

实际上服务器的处理包括接收请求和处理请求两部分。

1) 接收请求的处理过程如图 8-1 所示。

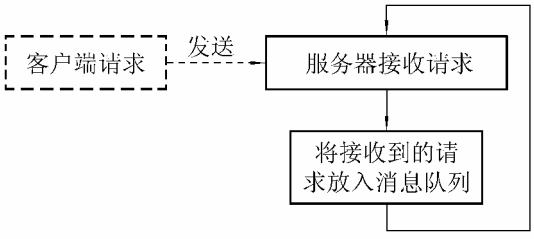


图 8-1 接收请求的处理过程图

2) 处理请求的处理过程如图 8-2 所示。

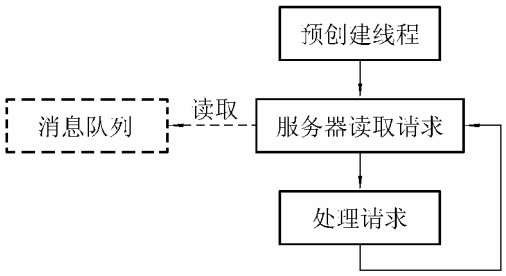


图 8-2　处理请求的处理过程图

8.6.2　线程池调度管理的编码实现

下面以一个简单的服务器为例来实现上述算法。假设客户端向服务器发送一个“Hello!”字符串，服务器收到后判断一下是不是“Hello!”这个字符串，如果是则发回一个“OK”字符串给客户端。

下面给出线程池调度管理的编码实现。

```
#define    MAX_QUEUE_SIZE    4 096
#define    MAX_LISTEN_SIZE   16
#define    MAX_THREADS       10
#define    DEFAULT_PORT      8 080

MSGQUEUE      *g_pQue；
void ProcessRequest(void *args)；

/** 初始化函数，初始化时会先创建消息队列，并预创建 MAX_THREADS 个线程
    @return INT——返回 CAPI_FAILED 表示失败；返回 CAPI_SUCCESS 表示成功
*/
INT Init()
{
    INT i；
    g_pQue = MsgQueue_Create(MAX_QUEUE_SIZE)；
    if ( g_pQue == NULL )
    {
        return CAPI_FAILED；
    }
    for ( i = 0；i < MAX_THREADS；i++ )
    {
        _beginthread(ProcessRequest, 0, NULL)；
    }
```

```
    return CAPI_SUCCESS；
}

/** 接收请求的处理函数，实现将接收请求的 SOCKET 标识符放入到消息队列中
    @return INT——返回 CAPI_FAILED 表示失败；返回 CAPI_SUCCESS 表示成功
*/
INT RecvQuest()
{
    struct sockaddr_in local, from；
    SOCKET s, msgsock；
    int fromlen；
    s = socket(AF_INET, SOCK_STREAM, 0)；
    if (s ==    INVALID_SOCKET)
    {
        return CAPI_FAILED；
    }
    local.sin_family = AF_INET；
    local.sin_addr.s_addr = INADDR_ANY；
    local.sin_port = htons(DEFAULT_PORT)；
    if ( bind(s, (struct sockaddr*)&local, sizeof(local)) == SOCKET_ERROR)
    {
        return CAPI_FAILED；
    }
    if ( listen(s, MAX_LISTEN_SIZE ) == SOCKET_ERROR)
    {
        return CAPI_FAILED；
    }
    while( msgsock = accept(s, (struct sockaddr*)&from, &fromlen)
        != INVALID_SOCKET )
    {
        SOCKET *pNewSock = new SOCKET；
        *pNewSock = msgsock；
        /* 事实上可放更多信息到消息队列中，这里放一个 SOCK ID 示意一下 */
        MsgQueue_Send(g_pQue, (void *)pNewSock)；
    }
    return CAPI_SUCCESS；
}

/** 处理请求的任务处理函数，这个函数是处理任务的入口函数，在里面处理接收客户端
```

```
    发来的字符串“ Hello! ”并发送响应“ OK ”到客户端
    @param void *——未使用
    @return void——无
*/
void ProcessRequest(void *)
{
    SOCKET *pSock；
    while (pSock = MsgQueue_Recv(g_pQue))
    {
        int    len；
        char buf[1024]；
        len = recv(*pSock, buf, sizeof(buf)-1, 0)；
        if ( len > 0 )
        {
            buf[len] = '\0'；
            if ( strcmp(buf, "Hello!")== 0 )
            {
                strcpy(buf, "Ok")；
                send(*pSock, buf, strlen(buf), 0)；
            }
        }
    }
}
```

可以看出，使用了消息队列后，整个线程池调度非常简单，由于本书中实现的消息队列在进行遍历时不会影响收发，因此可以实时查看有多少客户在线，使用消息队列的以下调用函数便可以实现对整个消息队列的遍历。

```
INT MsgQueue_EnumNext(MSGQUEUE *pQueue, VISITFUNC VisitFunc)；
```

这里只需要写一个访问函数 VisitFunc 即可，比如要想将当前所有在线的客户端连接 SOCKET ID 打印到图形界面中，可以写一个简单的访问函数来实现，编码如下。

```
INT PrintAllIDToScreen(void *pData)
{
    SOCKET *pSock = pData；
WriteToScreen(*pSock)；      /* WriteToScreen 是将数据写到图形界面的函数  */
}
```

再调用‘ MsgQueue_EnumNext(g_pQue, PrintAllIPToScreen) ’语句就可以实现遍历。

需要注意的是，上面的编码只是在消息队列中放入了一个 SOCKET ID，在程序中可以定义一个结构体，结构体中可以包含 SOCKET ID、客户端 IP 地址、连接建立时间等各种信息，然后将结构体发送到消息队列即可。这样遍历的时候就可以看到客户端 IP 地址、连接建立的时间等更多的信息了。

本 章 小 结

本章主要介绍了多任务算法的思想概念，重点介绍了多任务下的资源释放和遍历问题，通过本章的学习，读者应该掌握将任意数据结构变成支持多任务的方法。

习题与思考

1. 实现多个任务可以同时遍历操作的多任务链表。
2. 将前面第 4 章中的 WebServer CACHE 文件管理改写成支持多任务的。
3. 编码实现：创建多个任务，一些任务不停地向消息队列中发送数据，一些任务不停地从消息队列接收数据，然后还有一个任务不停地遍历消息队列里的数据。
4. 将本章的消息队列编码改写成：当队列满时，发送操作会阻塞在哪里？
5. 编码实现一个支持多任务操作的哈希 AVL 树。

9 内存管理算法

本章介绍了一个应用前景美好的动态等尺寸内存管理算法和一个在应用程序层实现的垃圾回收算法及其在多任务下的实现；并介绍了如何使用垃圾回收算法进行内存泄漏检查；最后介绍了利用动态等尺寸内存管理算法构造一个内存管理系统，同时介绍了内存越界检查的实现方法。

9.1 动态等尺寸内存的分配算法

内存分配算法是计算机中最基础的算法，不论是操作系统还是应用软件中都必须用到。有实际经验的读者想必知道各种内存分配算法都有它的优点和缺点，一个理想的内存算法应该满足以下要求。

① 能分配任意大小的内存；
② 能有效管理内存，即无限地分配下去，直到系统的内存耗光为止；
③ 释放的内存能重新被分配；
④ 内存碎片要尽量少；
⑤ 分配速度要快；
⑥ 内存管理消耗的辅助空间要尽量小。

9.1.1 静态等尺寸内存分配算法的分析

等尺寸块的分配算法在链表那章已经介绍过，使用整块内存的链表实际上就是一个最简单的静态等尺寸内存块分配算法，这种算法在目前商业软件中有着很多的应用，很多服务器软件都使用了这种算法。

1. 静态等尺寸内存分配算法的性能分析

根据前面介绍理想的内存分配算法所应达到的要求，可以分析一下使用静态等尺寸内存分配算法的优缺点。

1) 主要有以下几个方面的优点。

① 时间效率非常高，每次分配所花的时间为常量，相当于一个单向链表弹出头部节点的时间；

② 用于内存管理的辅助开销为 0，已被分配的节点的内存全部可以使用，不需要为内存管理作保留；

③ 内存性能非常好，不存在内存碎片问题，因此得以大规模应用；

④ 算法实现非常简单。

2) 主要有以下几个方面的缺点。

① 能分配的内存大小是固定的，实际使用中存在一定的浪费现象；

② 当需要分配的内存超过等尺寸内存分配中块的大小时，就无法分配了；

③ 最大可分配块数也是预先定义好的，当超过这个块数时，无法动态增加；

④ 当系统不需要这样大小的内存块时，已存在与内存链表中的内存块无法被利用，即使这些内存已经被释放。

2. 静态等尺寸内存分配算法不足之处的解决措施

针对第一条缺点，通常的做法是预先分配多个不同尺寸的整块内存链表，一般按 2 的幂来设定，如 32 Byte，64 Byte，128 Byte，256 Byte，512 Byte 等大小的整块内存链表。需要分配时，找一个最接近的尺寸的链表中进行分配，如分配一个 40 Byte 大小的内存时，就从 64 Byte 大小的整块内存链表中分配，分配 96 Byte 的内存就从 128 Byte 的链表中分配，这样就保证了内存浪费最大只有一倍。

对第二条缺点，也可以采用第一种措施解决。当超过一定大小时，就采用系统提供的分配函数如 malloc()进行分配。(注：malloc 是在未被本链表管理的内存区域中进行指定长度内存的分配，但是它的效率要低些。)

解决第三条和第四条缺点的措施，是本章要探讨的重点内容，下面便来探讨如何解决这两个问题。

9.1.2 动态等尺寸内存分配算法

1. 动态等尺寸内存分配算法概念的引出

要解决 9.1.1 节提到的第三个问题，也就是要解决当预先分配好的内存被用完时，能够重新申请新的内存进行分配可以从第 2 章数组中的动态队列得到启示——用一个数组来管理多个整块内存链表：预先创建一个整块内存链表，当第 1 块整块内存链表分配完后，创建第 2 个整块内存链表，将其加入数组中进行管理，以此类推，其结构

如图 9-1 所示。

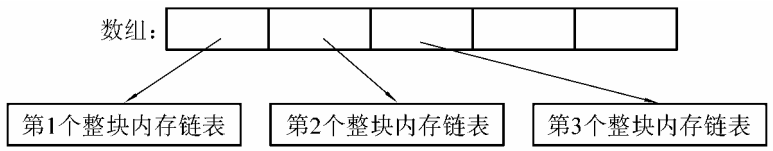


图 9-1　动态等尺寸内存管理示意图

上述措施实现了当预分的内存用完时，仍然可以申请分配内存，这样第三个问题就得到解决。

接下来解决第四个问题，即内存释放后，可以被其他任务使用。一般只要能做到某个整块内存链表中的内存全部被释放后就将这个整块内存链表释放掉，并从数组中移去这个链表即可。比如图 9-1 中第 2 个整块内存链表被释放后的情形如图 9-2 所示。

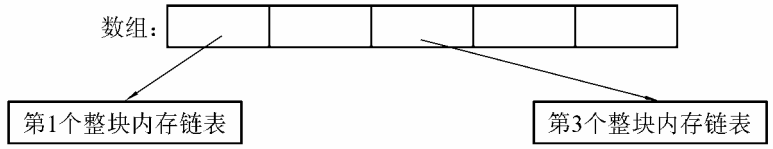


图 9-2　删除一个整块内存链表后的动态等尺寸内存管理示意图

2. 动态等尺寸内存分配算法的设计思路

下面先来设计分配算法，首先要从数组找到一个有自由空间的整块链表，于普通数组必须从数组开始到结尾按顺序一个一个地查找，这样效率就很低，有没有办法一次性地找到有自由空间的整块链表呢？只要分析整块内存链表中的内存分配是如何实现的就得到答案了。必须把管理多个整块内存链表的数组也变成一个整块内存链表，将有自由空间的链表也连成一个链表，这样每次分配内存时就可以从这个链表里直接找到有自由空间的整块内存链表了。

这样就可以画一个如图 9-3 所示的动态等尺寸内存管理示意图。

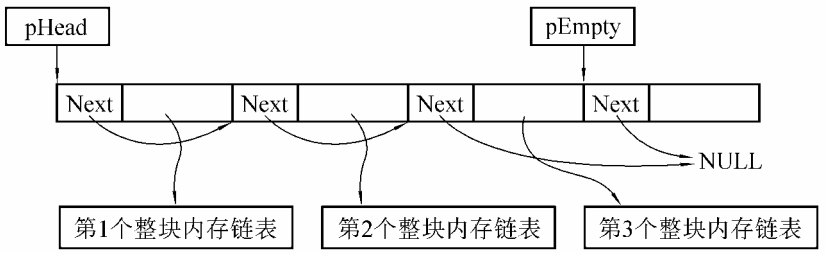


图 9-3　动态等尺寸内存管理示意图

1) **分配算法过程如下。**

步骤 1：从自由空间链中获取头部节点，如果头部节点为空则表明所有内存都已经被分配完，需要重新创建一个整块内存链表并加入自由空间链中，然后再从自由空间链中取出头部节点。

步骤 2：从获取的整块内存链表中分配一块内存。

步骤 3：判断获取的整块内存链表中的自由空间是否全部分配完。如果全部分配完则将其从自由空间链中弹出来，否则结束分配。

2) **分配算法流程如图 9-4 所示。**

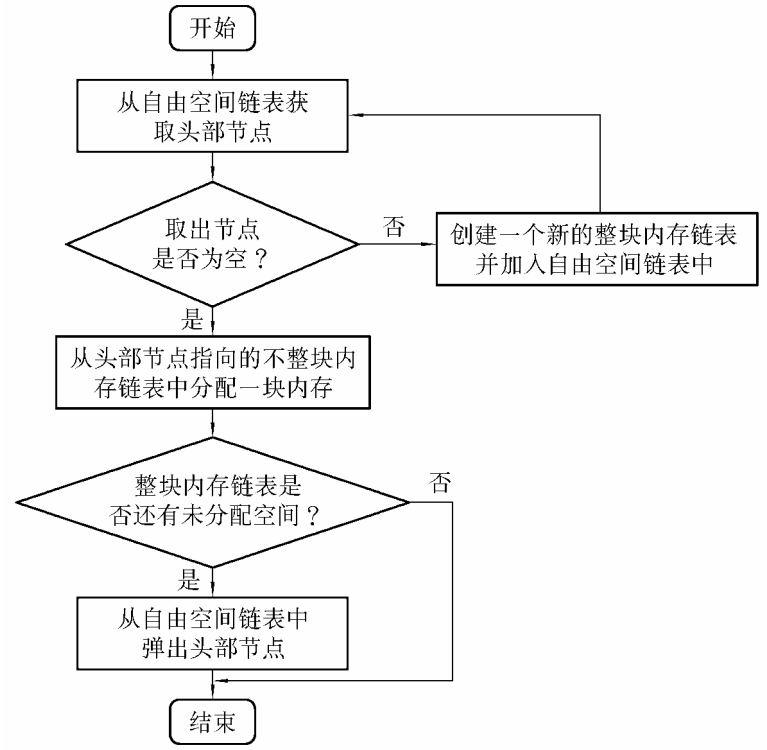


图 9-4　动态等尺寸内存管理分配算法流程图

3. **释放算法的设计思路**

设计完分配算法后，再来考虑释放算法的设计。要释放某块内存须要知道这块内存属于哪个整块内存链表，最简单的方法就是每块分配的内存都必须记住自己所属的整块内存链表在数组中的下标，这样每块被分配的内存要增加一个辅助空间来记录数组下标，因此需要修改第 3 章中介绍的整块内存链表的设计。

下面来设计一个专门为动态等尺寸内存分配使用的整块内存链表，为了有别于前

面的整块内存链表，不妨把它称为空间链表，用 C 语言结构体描述如下。

```
typedef struct SPNODE_st {
    UINT uPos；                 /* 位置，即在数组中的位置，为数组下标 */
    struct SPNODE_st *pNext；   /* 下一个节点指针 */
} SPNODE；

typedef struct SPLIST_st {
    void *pBlock；              /* 一块连续内存 */
    SPNODE *pHead；             /* 自由空间链表头部节点 */
    UINT uFreeCount；           /* 自由空间节点的个数 */
} SPLIST；
```

读者可能注意到上面的设计和第 3 章链表的整块内存链表 BlockList 设计中除了节点中多一个 uPos 外，在链表的设计中少了 uDataSize 和 uMaxDataCount 等变量。这是因为在整个动态等尺寸内存的设计中，所有的空间链表都具有相同的数据尺寸大小和数据个数，所以不需要每个空间链表都保存这些信息，统一放到一个地方保存就可以了。

1) 释放算法过程如下。

步骤 1：先根据要释放内存下标找到对应的空间链表。

步骤 2：释放掉要释放的内存。

步骤 3：判断这个空间链表是否满足整个被释放的条件，如果满足则将整个空间链表释放掉并从数组中移去，否则转下一步。

步骤 4：判断空间链表是否在自由空间链中，如果不在则需要将其加入到自由空间链中。

读者可能会问，步骤 3 中释放整个空间链表的条件是什么？

首先，必须要满足整块内存链表中没有数据，全部是空闲内存。

其次，如果仅仅满足这个条件就释放又会造成另外一个现象，即如果某个整块内存链表全部是自由空间，这时分配了一块内存，马上又释放这块内存，这种情况很常见，这时就看到内存被释放后整块内存链表又变成全部是自由空间了，如果将其整个释放掉显然效率非常低，所以将某个整块内存链表释放时要做一些条件限制，刚才讨论的那种情况必须避免。实际上，判断一下欲释放的链表是否为当前正在使用的链表就可以了。

还有一种情况不能释放，那就是整个管理数组中必须保存一个设定的最小数量的整块内存链表，否则频繁地进行整块内存链表的创建和释放也是一件很困难的事情。

2) **释放算法的流程图如图** 9-5 **所示。**

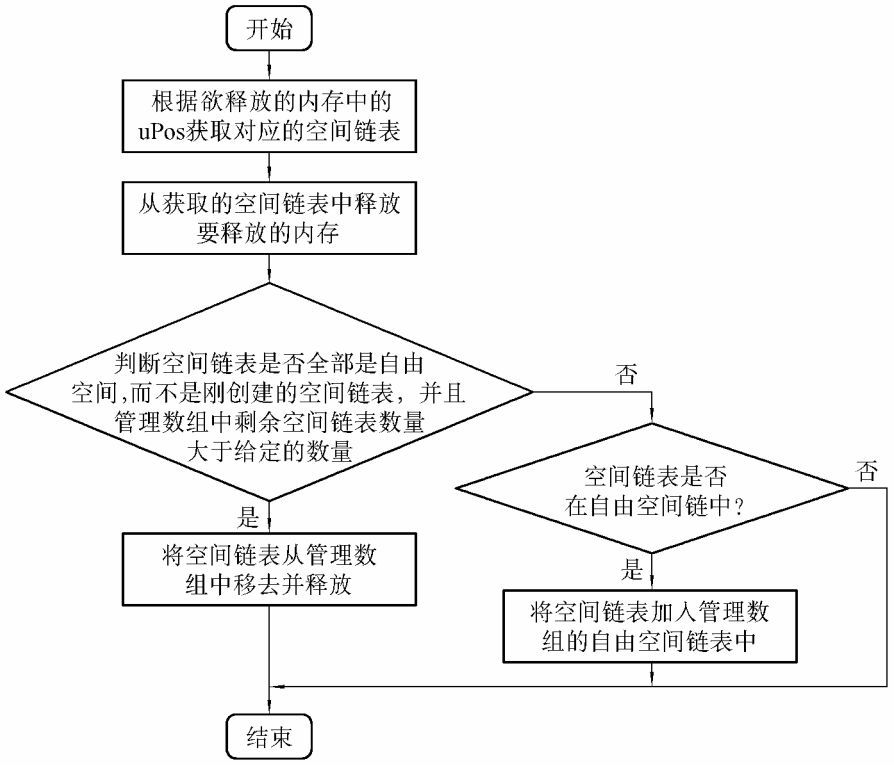


图 9-5　**动态等尺寸内存管理释放算法流程图**

4. **动态等尺寸内存分配算法的编码实现**

1) **动态等尺寸内存分配算法的设计可以用以下结构体来描述。**

```
typedef struct DSPACENODE_st {
    struct DSPACENODE_st *pPrev；  /*  前一个节点指针  */
    struct DSPACENODE_st *pNext；  /*  后一个节点指针  */
    SPLIST *pSpList；              /*可用空间表指针  */
    UINT uInListFlag；             /*  标示本节点是否在链表中的标志  */
} DSPACENODE；

typedef struct DSPACELIST_st {
    DSPACENODE *pDSNode；     /*  用来管理可用空间表的 DSPACENODE 数组  */
    DSPACENODE *pHead；       /*  有自由空间的 DSPACENODE 链的头部  */
    DSPACENODE *pTail；       /*  有自由空间的 DSPACENODE 链的尾部  */
    UINT uDataCount；         /*  每个可用空间表中的数据个数  */
```

```
    UINT uDataSize；              /* 数据大小 */
    UINT uDSNodeCount；           /* 总的 DSPACENODE 节点数量 */
    UINT uCurrentDSNode；         /* 当前正在使用的 DSPACENODE 索引 */
    UINT uMinDSNodeCount；        /* 最小使用的 DSPACENODE 数量 */
    UINT uUsedDSNodeCount；       /* 正在使用的 DSPACENODE 数量 */
} DSPACELIST；
```

2) **下面给出空间链表的编码实现。**

```
/** 空间链表的创建函数
    @param UINT uSpaceCount——空间数量
    @param UINT uDataSize——数据大小
    @param UINT uPos——在管理数组中的位置
    @return SPLIST *——成功返回空间链表指针；失败返回 NULL
*/
SPLIST *SpList_Create(UINT uSpaceCount, UINT uDataSize, UINT uPos)
{
    SPLIST *pSpList；
    SPNODE  *pNode；
    UINT i；
    pSpList = (SPLIST *)malloc( sizeof(SPLIST)
        + uSpaceCount *(uDataSize + sizeof(SPNODE)) )；
    if ( pSpList != NULL )
    {
        pSpList->pBlock = (void *)((char *)pSpList +sizeof(SPLIST) )；
        /* 创建自由空间链表 */
        pSpList->pHead = (SPNODE *)pSpList->pBlock；
        pNode = pSpList->pHead；
        for (i = 0；i < uSpaceCount；i++)
        {
            pNode->uPos = uPos；
            pNode->pNext = (SPNODE *)((char *)pNode + sizeof(SPNODE)
                + uDataSize)；
            pNode = pNode->pNext；
        }
        pNode->pNext = NULL；
        pSpList->uFreeCount = uSpaceCount；
    }
    return pSpList；
}
```

```
/** 空间链表的释放函数
    @param SPLIST *pSpList——空间链表指针
    @return void——无
*/
void SpList_Destroy(SPLIST *pSpList)
{
    if ( pSpList != NULL )
    {
        free( pSpList )；
    }
}

/** 空间链表的分配内存函数
    @param SPLIST *pSpList——空间链表指针
    @return void *——成功返回分配到的内存；若无自由内存供分配则返回 NULL
*/
void *SpList_Alloc(SPLIST *pSpList)
{
    SPNODE *pNode；
    pNode = pSpList->pHead；
    if ( pNode != NULL )
    {
        pSpList->pHead = pNode->pNext；
        pSpList->uFreeCount－－；
        return (void *)((char *)pNode + sizeof(SPNODE))；
    }
    return NULL；
}

/** 空间链表的释放内存函数
    @param SPLIST *pSpList——空间链表指针
    @param void *pData——欲释放的内存数据指针
    @return void——无
*/
void SpList_Free(SPLIST *pSpList, void *pData)
{
    SPNODE   *pNode；
    pNode= (SPNODE *)((char *)pData－sizeof(SPNODE))；
    pNode->pNext = pSpList->pHead；
```

```
    pSpList->pHead = pNode；
    pSpList->uFreeCount++；
}
```

3) **下面给出动态等尺寸内存分配管理算法编码实现。**

```
#define   DSPACENODE_IN_LIST          1
#define   DSPACENODE_NOT_IN_LIST   0

/** 动态等尺寸内存分配管理的创建函数
    @param UINT uDataSize——所管理的空间链表数据大小
    @param UINT uDataCount——所管理的空间链表数据数量
    @param UINT uMinDSNodeCount——最小保存的空间链表个数
    @param UINT uDSNodeCount——最大保存的空间链表个数
    @return DSPACELIST *——成功返回等尺寸内存分配管理指针；失败返回 NULL
*/
DSPACELIST *DSpaceList_Create(UINT uDataSize, UINT uDataCount,
                              UINT uMinDSNodeCount, UINT uDSNodeCount)
{
    DSPACELIST *pList；
    UINT i；
    pList = (DSPACELIST *)malloc(sizeof(DSPACELIST))；
    if ( pList == NULL )
    {
        return NULL；
    }
    pList->pDSNode = (DSPACENODE *)malloc(uDSNodeCount *sizeof(DSPACENODE))；
    if ( pList->pDSNode == NULL )
    {
        free(pList)；
        return NULL；
    }
    pList->pDSNode[0].pSpList = SpList_Create(uDataCount, uDataSize, 0)；
    if ( pList->pDSNode[0].pSpList == NULL )
    {
        free(pList->pDSNode)；
        free(pList)；
        return NULL；
    }
    /* 将第 0 个节点插入到双向链表中 */
```

```
    pList->pHead = &(pList->pDSNode[0])；
    pList->pTail = pList->pHead；
    pList->pHead->pNext = NULL；
    pList->pHead->pPrev = NULL；
    pList->pDSNode[0].uInListFlag = DSPACENODE_IN_LIST；
    if ( uDSNodeCount < uMinDSNodeCount )
    {
        uDSNodeCount = uMinDSNodeCount；
    }
    pList->uCurrentDSNode = 0；
    pList->uDataCount = uDataCount；
    pList->uDataSize = uDataSize；
    pList->uDSNodeCount = uDSNodeCount；
    pList->uMinDSNodeCount = uMinDSNodeCount；
    pList->uUsedDSNodeCount = 1；
    for ( i = 0；i < uDSNodeCount；i++ )
    {
        pList->pDSNode[i].pSpList = NULL；
        pList->pDSNode[i].uInListFlag = DSPACENODE_IN_LIST；
        pList->pDSNode[i].pNext = NULL；
        pList->pDSNode[i].pPrev = NULL；
    }
    return pList；
}

/** 动态等尺寸内存分配管理的释放函数
    @param DSPACELIST *pList——动态等尺寸内存分配管理指针
    @return void——无
*/
void DSpaceList_Destroy(DSPACELIST *pList)
{
    UINT i；
    if ( pList != NULL )
    {
        for ( i = 0；i < pList->uDSNodeCount；i++ )
        {
            if ( pList->pDSNode[i].pSpList != NULL )
            {
                SpList_Destroy(pList->pDSNode[i].pSpList)；
```

```
            }
        }
        free(pList)；
    }
}

/** 动态等尺寸内存分配管理的内存分配函数
    @param DSPACELIST *pList——动态等尺寸内存分配管理指针
    @return void *——成功返回分配到的内存；失败返回 NULL
*/
void *DSpaceList_Alloc(DSPACELIST *pList)
{
    UINT i；
    void *pData；
    DSPACENODE *pDSNode；
    pData = NULL；
    pDSNode = pList->pHead；
    if ( pDSNode != NULL )
    {
        pData = SpList_Alloc(pDSNode->pSpList)；
        if ( pDSNode->pSpList->uFreeCount == 0 )
        {
            /* 头部节点指向的可用空间表已没有自由空间，将头部节点弹出 */
            pList->pHead = pDSNode->pNext；
            if ( pList->pHead != NULL )
            {
                pList->pHead->pPrev = NULL；
            }
            pDSNode->uInListFlag = DSPACENODE_NOT_IN_LIST；
        }
    }
    else
    {
        /* 查找并重新分配新的可用空间表来使用 */
        for ( i = pList->uCurrentDSNode + 1；i < pList->uDSNodeCount；i++ )
        {
            if ( pList->pDSNode[i].pSpList != NULL )
            {
                break；
```

```
        }
    }
    if ( pList->pDSNode[i].pSpList != NULL )
    {
        for ( i = 0；i < pList->uCurrentDSNode；i++ )
        {
            if ( pList->pDSNode[i].pSpList != NULL )
            {
                break；
            }
        }
        if ( pList->pDSNode[i].pSpList != NULL )
        {
            UINT j；
            /* 将 DSPACENODE 表的空间扩大一倍 */
            DSPACENODE *p = (DSPACENODE *)malloc(
                2 *pList->uDSNodeCount *sizeof(DSPACENODE))；
            if ( p == NULL )
            {
                return NULL；
            }
            i = pList->uDSNodeCount；
            memcpy(p, pList->pDSNode,
                pList->uDSNodeCount *sizeof(DSPACENODE))；
            pList->uDSNodeCount *= 2；
            free(pList->pDSNode)；
            pList->pDSNode = p；
            /* 重新初始化扩大后增加部分的数据 */
            for ( j = i；j < pList->uDSNodeCount；j++ )
            {
                pList->pDSNode[j].pSpList = NULL；
                pList->pDSNode[j].uInListFlag = DSPACENODE_NOT_IN_LIST；
                pList->pDSNode[j].pNext = NULL；
                pList->pDSNode[j].pPrev = NULL；
            }
        }
    }
    pDSNode = &(pList->pDSNode[i])；
    pDSNode->pSpList = SpList_Create(pList->uDataCount, pList->uDataSize, i)；
```

```
            if ( pDSNode->pSpList != NULL )
            {
                pList->uUsedDSNodeCount++；
                pList->uCurrentDSNode = i；
                /* 将 pDSNode 插入到双向链表尾部 */
                pDSNode->pNext = NULL；
                pDSNode->pPrev = NULL；
                pList->pHead = pDSNode；
                pList->pTail = pDSNode；
                pDSNode->uInListFlag = DSPACENODE_IN_LIST；
                pData = SpList_Alloc(pDSNode->pSpList)；
            }
        }
        return pData；
}

/** 动态等尺寸内存分配管理的内存释放函数
    @param DSPACELIST *pList——动态等尺寸内存分配管理指针
    @param void *pData——要释放的内存数据
    @return void——无
*/
void DSpaceList_Free(DSPACELIST *pList, void *pData)
{
    SPNODE *pNode；
    DSPACENODE *pDSNode；
    pNode = (SPNODE *)((char *)pData－sizeof(SPNODE))；
    pDSNode = &(pList->pDSNode[pNode->uPos])；
    /* 判断是否符合将整个可用空间链表释放的条件 */
    if ( pDSNode->pSpList->uFreeCount == pList->uDataCount
        && pList->uUsedDSNodeCount > pList->uMinDSNodeCount
        && pNode->uPos != pList->uCurrentDSNode )
    {
        /* 释放此节点指向的可用空间链表 */
        SpList_Destroy(pDSNode->pSpList)；
        pDSNode->pSpList = NULL；
        /* 从双向链表中删除此节点 */
        if ( pDSNode->pPrev != NULL )
        {
            pDSNode->pPrev->pNext = pDSNode->pNext；
```

```
        }
        if ( pDSNode->pNext != NULL )
        {
            pDSNode->pNext->pPrev = pDSNode->pPrev；
        }
        if ( pList->pHead == pDSNode )
        {
            pList->pHead = pDSNode->pNext；
        }
        if ( pList->pTail == pDSNode )
        {
            pList->pTail = pDSNode->pPrev；
        }
        pDSNode->pNext = NULL；
        pDSNode->pPrev = NULL；
        pDSNode->uInListFlag = DSPACENODE_NOT_IN_LIST；
        /* 更新当前使用的 DSPACENODE 总数的计数 */
        pList->uUsedDSNodeCount－－；
    }
    else
    {
        SpList_Free(pDSNode->pSpList, pData)；
        if ( pDSNode->uInListFlag == DSPACENODE_NOT_IN_LIST )
        {
            /* 将其插入双向链表尾部 */
            pDSNode->pPrev = pList->pTail；
            pDSNode->pNext = NULL；
            if ( pList->pTail != NULL )
            {
                pList->pTail->pNext = pDSNode；
            }
            else
            {
                pList->pHead = pDSNode；
            }
            pList->pTail = pDSNode；
            pDSNode->uInListFlag = DSPACENODE_IN_LIST；
        }
    }
```

```
        return；
    }
```

9.2 内存垃圾收集算法

9.2.1 垃圾收集算法简介

目前，C/C++语言的内存管理要求申请的内存必须由用户手工释放，如果用户没有进行手工释放就会造成内存泄漏，这对编程提出了很高的要求。特别是在一些有高可靠性要求的应用和服务器软件中，如果产生内存泄漏将严重影响软件的可用性，同时手工回收内存还大大增加了程序员的工作量。为了让分配的内存在不再使用时自动释放，产生了垃圾收集算法。

垃圾收集算法最早出现于 20 世纪 60 年代，到现在已经发展了几十年。由于 20 世纪 90 年代 Java 语言支持垃圾收集功能，使得垃圾收集算法被广大程序员所认识。现在许多语言如 Smalltalk 等都拥有了垃圾收集的支持，对于 C/C++，也出现了一些支持收集的库。

目前常见的垃圾收集算法有以下几种。

1. 引用计数算法

引用计数算法中，要给每一个动态分配的内存进行引用计数，也就是当每次对内存的引用增加时就加 1，当取消对内存的引用时就减 1，当引用计数变成 0 时，表示内存没有被使用，内存就可以被回收了。

在 C/C++中，每次指针指向一块分配的内存时，就要将这块内存的引用计数加 1，增加了指针操作的开销。回收时相对而言非常简单，只要判断引用计数为 0 就释放。这个算法的缺点是当有循环引用时，循环引用的内存块由于相互都有指针指向对方，因此它的引用计数必然大于等于 1，导致无法将其释放。

引用计数的一个很大优点就是实现起来非常容易，并且在没有编译器支持的情况下，在用户程序层也很容易实现。

2. 标志清扫算法

在实际内存中，所有分配的内存对象组成了一个或多个连通图，可以通过从根节点开始对图进行遍历，能遍历到的内存对象就是被使用的对象，无法遍历到的内存对象便是需要释放的对象。在 C/C++程序中，根节点就是程序中的指针变量等元素。标志清扫算法便是基于以上原理设计的，分为如下两个阶段进行。

1) 先将所有被分配的内存对象设置为未标志状态，然后通过根节点直接或间接地

进行遍历操作，将能遍历到的内存对象标志成正在使用状态。

2) 遍历所有被分配的内存，如果处于未标志状态就将其释放掉。

标志清扫算法的优点在于循环引用对它来说不是问题，对指针的操作也没有增加任何开销，在回收之前并没有增加开销。不过它也有两个缺点：首先，回收时要扫描所有被分配的内存，如果已分配内存数量较多会花费很多时间；其次，标志清扫算法在概念上很简单，实现起来却并不容易。

3. 节点复制算法

节点复制算法将内存空间分成两个区，一个区存放现有数据，称为活动空间，另外一个区存有已废弃的数据，称作空闲空间。当进行垃圾回收时，将活动空间中的正在使用对象复制到空闲空间中，然后将空闲空间变成活动空间，活动空间变成空闲空间。这种算法的好处是回收后，正在使用的内存对象被压缩到一片连续区域中，减少了内存碎片；缺点是只有一半的自由内存可以被使用，另外还有一个缺点是当内存使用超出一定量时，回收时效率会受影响。

9.2.2 用户层垃圾回收算法的实现

由于很多垃圾回收算法都是在编译器层面实现的，而编译器层面实现起来牵涉的内容很复杂，一般软件人员理解起来很困难，实际上他们也没有必要去详细了解编译器层面的垃圾回收算法是如何实现的，为了讲清楚垃圾回收的原理，我们还是以一个用户程序层面的垃圾回收算法的具体实现来进行讲解。

前面提到，引用计数算法简单易懂，且可以在用户程序层面实现，下面就以引用计数算法来实现一个 C++用户程序层面的垃圾回收算法。

1. C++用户程序层的垃圾回收实现原理

首先要了解清楚 C++用户程序层的垃圾回收实现原理，引用计数算法中，如何对一个已分配内存进行计数的增减操作呢？先看一段 C++的程序吧。

```
int *p；
{
    int *q = new int；
    *q = 100；
    p = new int；
    *p = 200；
}
```

在这段程序中，当执行“ *q= 100 ；”语句时，需要对分配的内存引用计数加 1，当退出大括号“ } ”后，由于指针变量 q 已经被编译器释放掉了，所以这时指向的内存的

引用计数要减 1。但在上面这段程序中，是没有实现引用计数功能的。要实现引用计数功能，就必须执行 int *q = new int；这样的语句能自动给分配的内存增加引用计数，而指针变量 q 释放时由于是编译器实现的，用户程序并不知道，所以还需要找到发现变量被编译器释放的方法。

2. 模拟指针的 C++实现

如果可以设计一个类来模拟指针，那么就可以利用类的构造函数和析构函数来操作指针变量的引用计数，幸运的是 C++中的语法提供了这种功能，可以通过重载运算符“ * ”、“ ，”、“ = ”和“ -> ”来实现。

```
template <class T> class GCPtr {
    T *m_pAddr；
public：
    GCPtr(T *t=NULL){m_pAddr = t；}；
    ~GCPtr()；
    T &operator*() { return *m_pAddr；}；
    T *operator->() { return m_pAddr；}；
    T *operator=(T *t) { m_pAddr = t；}；
    GCPtr & operator=(GCPtr &r) { m_pAddr = r.m_pAddr；}；
}；
```

在通过如下方式定义一个对象 p 时：

```
GCPtr<int> p；
```

可以将对象 p 如 int 类型指针一样来使用，实际的操作都是通过重载运算符来操作 GCPtr 类里面的 pAddr 成员实现的。

有了模拟指针，就可以通过类 GCPtr 来定义任意类型的指针。只要在 GCPtr 的构造函数和析构函数里进行引用计数的增减，就可以实现引用计数方法的垃圾回收了。

也许有些读者认为可以直接在析构函数里将指针指向的内存释放，但这个想法是在只有这一个指针指向释放的内存的情况下才可行，如果有其他指针指向同一块内存，并且其他指针的生存期更长，就会出现程序运行异常，下面可以看一段简单的程序。

```
GCPtr<int> p；
p = new int；
 {
    GCPtr q；
    q = p；
  }
```

*p = 100；

以上这段程序中 p 和 q 指向的是同一块内存，如果在析构函数中直接将内存释放，那么“ *p=100 ;”这条语句将出现异常，所以不能简单地将一块内存释放掉，必须确认没有指针指向它时才可以释放。因此必须给内存加上引用计数，有多少个指针指向它，计数就为多少，如果引用计数为 0 就表示没有指针指向它，这时才可以将内存释放掉。

3. 引用计数的实现

上面实现了一个模拟指针的功能，要将模拟指针变成一个具有引用计数功能的模拟指针，首要问题就是要为每块分配的内存找一个保存引用计数的地方。

可不可以在 GCPtr 类里面直接定义一个变量来保存引用计数呢？答案是否定的。因为指针和内存并不是一一对应的关系，而是可能有多个指针指向同一块内存。引用计数必须和分配的内存一一对应，如果 GCPtr 类里面直接定义一个变量来保存引用计数的话，当另外一个 GCPtr 对象指向一个已有 GCPtr 对象指向的内存时，如何去增加引用计数？这显然是无法实现的，所以必须设计一个和分配内存一一对应的保存引用计数的地方。

首先想到的就是设计一张表来保存分配的内存地址及对应的引用计数，重点是考虑在操作引用计数时，如何快速通过分配的内存地址定位到对应的引用计数变量。

这时也许有人会想到用一个哈希表之类的能够快速查找的数据结构来保存内存地址和引用计数，还有没有更快的方法呢？

更快的方法就是通过内存地址直接就可以访问引用计数，这就要求引用计数保存在和分配的内存连续的空间上。但问题又来了，系统提供的内存管理算法并没有留下可以保留引用计数的位置，要实现这样的功能就必须写一个自己的内存管理算法。

由于是在应用程序层，写一个内存管理算法实际上还是需要先向操作系统申请一大块内存，然后使用自己的内存管理算法来管理这块内存，问题是如何知道需要向操作系统申请多少内存呢？也许目前需要 1 兆内存，但下个版本也许又需要 2 兆内存，总之内存的需求是变化无常的，使用自己的内存管理算法在实际应用中必然存在浪费的情况，所以这种方法实现起来不能令人满意。

难道又退回到用哈希表吗？用哈希表在每次指针赋值时，增加引用计数需要进行哈希表的查找，对一个赋值语句来说，开销实在太大了，自己写内存管理算法也行不通，所以还是想想有没有别的方法吧。那么，可不可以继承系统的内存管理算法呢？

由于系统的内存管理没有在管理上给分配的内存预留引用计数位置，但可以在分配给用户使用的内存上自己预留引用计数的空间。比如在 32 位系统上，要想分配一个 128 B 的内存块，那就应该分配一个 132 B 的内存块，将内存块前面 4 个字节留作引用计

数，将第 4 个字节后的空间返回给程序使用。这样当需要通过内存地址来操作引用计数时，直接就可以将指针地址向后移 4 个字节，然后在这 4 个字节上写引用计数就可以了，这样就可以设计一个用户的内存管理函数如下。

```
#define    INT_LEN    4
void *GC_Malloc(size_t size)
{
    void *p = malloc( size + INT_LEN )；
    if ( p != NULL )
{
        return (void *)((char *)p+INT_LEN)；
    }
    return NULL；
}
```

当然还要设计一个对应的释放函数如下。

```
void GC_Free(void *p)
{
void *pFree = (void *)((char *)p – INT_LEN)；
free(pFree)；
}
```

使用了垃圾回收算法后，GC_Free( )一般只由垃圾回收算法来调用，用户就不需要调用 GC_Free( )了，也不需要花费时间去关心内存释放的问题了。

下面给出使用 GCPtr 类管理引用计数的一个初步编码实现。

```
/* 支持垃圾内存回收的 GCPtr 类，使用模板实现 */
template <class T> class GCPtr {
public：
    T *m_pAddr；        /* 用来记住定义的指针地址便于析构函数使用 */
public：
    GCPtr(T *t = NULL)
    {
        m_pAddr = t；
        INT *p = (INT *)m_pAddr－1；
        *p += 1；
    }；

    ～GCPtr()
```

```
    {
        T *p = m_pAddr；
        INT *p = (INT *)m_pAddr－1；
        *p－= 1；
    }；
}；
```

以上编码便是使用 GCPtr 类管理引用计数的一个初步实现，当然这里只是为了演示一下如何管理引用计数，并不是全部功能的实现。具体应用中还有许多操作符需要重载，将在后面进行介绍。

4. 构造函数和析构函数的说明

可以看到在 GCPtr 的构造函数中，直接由内存地址定位到存放引用计数的位置，将引用计数加 1。

析构函数要做的事情也很简单，就是当一个 GCPtr 类型的变量释放时，直接由要释放的内存地址(已经存放到 m_pAddr 中)定位到存放引用计数的位置，将引用计数减 1。

这种方法在修改引用计数时效率非常高，与哈希表相比无需查表等操作。

5. 需要修改引用计数的情况

实际上，不仅仅是构造函数和析构函数要修改引用计数，牵涉指针的操作基本上都需要修改引用计数，主要有以下三种情况。

1) 当将新地址赋给 GCPtr 时，需要将原来地址的引用计数减 1，新的地址的引用计数加 1，这时需要重载运算符“ = ”号。

2) 当将一个 GCPtr 赋给另外一个 GCPtr 时，需要将等号左边的 GCPtr 指向的内存块的引用计数减 1，等号右边的 GCPtr 指向的内存块的引用计数加 1。

3) 当需要对象的副本时，需要调用 GCPtr 的拷贝构造函数，如将对象当作参数传递给函数，作为函数的返回值等情况都会调用拷贝构造函数。由于复制的对象是指向相同的地址，而复制对象在释放时要调用析构函数，析构函数将引用计数减 1，因此需要在拷贝函数里加 1 以维持引用计数的平衡。

下面给出这几个操作的编码，由于构造函数和析构函数等在前面已经给出，所以这里省略。

```
template <class T> class GCPtr {
public：
    T *m_pAddr；        /* 用来记住定义的指针地址便于析构函数使用 */
public：
    /* 拷贝构造函数 */
```

```
    GCPtr(const GCPtr &gcPtr)
    {
        INT *p = (INT *)gcPtr.m_pAddr－1；
        *p += 1；
        m_pAddr = gcPtr.m_pAddr；
    }
    T *operator=(T *t)
    {
        /* 将原来指向的内存引用计数减 1 */
        INT *p = (INT *)m_pAddr－1；
        *p－= 1；
        m_pAddr = t；
        /* 将新指向的内存的引用计数加 1 */
        p = (INT *)t－1；
        *p += 1；
        return t；
    }；
    GCPtr & operator = (GCPtr &r)
    {
        /* 将原来指向的内存引用计数减 1 */
        INT *p = (INT *)m_pAddr－1；
        *p－= 1；
        m_pAddr = r.m_pAddr；
        /* 将新指向的内存的引用计数加 1 */
        p = (INT *)r.m_pAddr－1；
        *p += 1；
        return r；
    }；
}；
```

6. 垃圾收集功能的实现

引用计数的功能实现后，接着要做的就是实现垃圾内存回收，也就是将引用计数为 0 的内存全部释放后，如何去访问已经分配的内存？必须用一张表来保存所有已分配的内存地址，可以在 GC_Malloc()函数分配内存时就将分配的内存地址保存到哈希表里，然后就可以通过对哈希表的遍历来实现对内存垃圾的回收，由于牵涉哈希表的操作，除了修改 GC_Malloc()函数外，还需要增加一个初始化函数，在初始化函数里创建哈希表，以下便是实现编码。

```
HASHTABLE    *g_pTable；/* 哈希表指针 */

/** 垃圾内存收集算法的初始化函数
    @param INT nBucketCount——哈希表的 bucket 的数量
    @return INT——返回 CAPI_SUCCESS 表示成功；返回 CAPI_FAILED 表示失败
*/
INT GC_Init(INT nBucketCount)
{
    g_pTable = HashTable_Create(nBucketCount)；
    if ( g_pTable != NULL )
    {
        return CAPI_SUCCESS；
    }
    return CAPI_FAILED；
}

/** 垃圾内存收集算法的内存分配函数
    @param size——要分派的内存大小，以字节为单位
    @return——成功返回分配到的内存地址；失败返回 NULL
*/
void *GC_Malloc(size_t size)
{
    void *p = malloc( size + INT_LEN )；
    if ( p == NULL )
    {
        GC_Collect()；
        p = malloc( size + INT_LEN )；
        if ( p == NULL )
        {
            return NULL；
        }
    }
    HashTable_Insert( g_pTable, p, HashInt)；
    *((INT *)p) = 0；
    return (void *)((char *)p+INT_LEN)；
}

/** 垃圾收集函数，遍历哈希表，将所有引用计数为 0 的内存释放
    @return void——无
```

```
*/
void GC_Collect()
{
    void *p；
    HashTable_EnumBegin(g_pTable)；
    while ( (p = HashTable_EnumNext(g_pTable)) != NULL )
    {
        INT *pRef = (INT *)p－1；
        if ( *pRef == 0 )
        {
            HashTable_Delete(g_pTable, p, HashInt, IntCompare, NULL)；
            GC_Free(p)；
        }
    }
}
```

注意：上面 GC_Malloc()**函数里当分配失败后会调用** GC_Collect()**函数收集垃圾，然后再继续分配内存。**

7. **手工释放及循环引用的释放**

由于引用计数不能处理循环引用的情况，另外可能有些特殊情况下想自己手工将某块内存释放掉，比如申请了一大块内存，用完后不马上释放会浪费很多内存，调用 GC_Collect()**函数执行的时间开销又比较大，所以有必要设计一个可以让用户手工释放的函数，用户可以通过手工释放函数去释放内存，也可以通过手工释放函数去手工释放循环引用的内存。**

要实现手工释放功能，必须保证在手工释放后 GC_Collect()**函数不会再对这块内存进行重复释放。从** GC_Collect()**函数的实现可以看出它是通过对哈希表的遍历来查找引用计数为** 0 **的内存进行释放的，因此只要手工释放时将对应的内存从哈希表中删除，**GC_Collect()**函数就不会重复释放这块内存了。**

下面给出手工释放函数的编码实现。

```
/** 垃圾内存收集算法的手工释放内存函数
    @param void *p——要释放的内存地址
    @return void——无
*/
void GC_ManualFree(void *p)
{
    void *pFree = (void *)((char *)p－INT_LEN)；
```

```
        HashTable_Delete(g_pTable, pFree, HashInt, IntCompare, NULL)；
        free(pFree)；
    }
```

可以看出，手工释放实现也非常简单，和 GC_Free()的区别就是多了一行哈希表的删除操作而已。使用手工释放函数给用户的使用提供了极大的方便，因为用户可以通过这个函数手工释放循环引用的内存。

9.2.3 多任务下的垃圾收集

前面通过使用引用计数实现了一个用户程序层的垃圾回收算法，并且可以通过手工释放函数来实现对循环引用内存的释放，前面介绍的都是单任务下的情况，那么在多任务下有何区别呢？

1. 多任务下垃圾收集需要解决的问题

在多任务下，主要有以下三个问题要考虑。

1) 对引用计数进行增减操作时，如果有多个任务操作一块内存的引用计数，那么必须给引用计数的操作加上锁保护。

2) 分配内存、手工释放内存、垃圾收集三个函数里都需要操作哈希表，哈希表必须支持多任务。

3) 在多任务应用环境中，大部分分配的内存都只有一个任务访问它，有多个任务访问的内存只是极少数，如何解决单任务下垃圾收集高效率与多任务下垃圾收集效率低下的矛盾？

对第一、二个问题，只要加锁就可以解决；对第三个问题，可以再设计一个支持多任务下的垃圾收集模块。如果有多个任务共享的内存，则使用多任务下的垃圾收集模块，单任务访问的内存仍然使用前面实现的垃圾收集模块，这样避免全部使用同样的算法导致效率低下。

2. 支持多任务垃圾收集的模块设计

下面就来设计支持多任务的垃圾收集模块，首先得将 GCPtr 修改掉，将里面的对引用计数操作全部上锁。但是要上锁，必须创建一把锁，在需要访问引用计数或需要操作哈希表时，均需要先获取锁，操作完后再解锁，这样还得将 GCInit()，GC_Malloc()，GC_Collect()等函数略作修改，为了有别于单任务的命名，可以在支持多任务的函数或类的命名前面加一个字符“M”。

1) 多任务下的 MGCPtr 类的编码实现如下。

```
extern LOCK g_lock；   /* 用来保护引用计数读写及哈希表操作的锁 */
```

```
/* 支持垃圾内存回收的类 GCPtr，使用模板实现 */
template <class T> class MGCPtr {
public：
    T *m_pAddr；      /* 用来记住定义的指针地址便于析构函数使用 */
public：
    MGCPtr(T *t = NULL)
    {
        m_pAddr = t；
        INT *p = (INT *)((char *)m_pAddr－INT_LEN)；

        Lock(g_lock)；
        *p += 1；
        Unlock(g_lock)；
    }；
    ~MGCPtr()
    {
        INT *p = (INT *)((char *)m_pAddr－INT_LEN)；
        Lock(g_lock)；
        *p－= 1；
        Unlock(g_lock)；
    }；
    /* 拷贝构造函数 */
    MGCPtr(const MGCPtr &gcPtr)
    {
        INT *p = (INT *)((char *)gcPtr.m_pAddr－INT_LEN)；
        Lock(g_lock)；
        *p += 1；
        Unlock(g_lock)；
        m_pAddr = gcPtr.m_pAddr；
    }
    T & operator* () { return *m_pAddr；}；
    T *operator-> () { return m_pAddr；}；
    operator T *() { return m_pAddr；}；
    T *operator=(T *t)
    {
        /* 将原来指向的内存引用计数减 1 */
        INT *p = (INT *)((char *)m_pAddr－INT_LEN)；

        Lock(g_lock)；
```

```
            *p－= 1；
            m_pAddr = t；
            /* 将新指向的内存的引用计数加 1 */
            p = (INT *)((char *)t－INT_LEN)；
            *p += 1；
            Unlock(g_lock)；
            return t；
        }；
        MGCPtr & operator=(MGCPtr &r)
        {
            /* 将原来指向的内存引用计数减 1 */
            INT *p = (INT *)((char *)m_pAddr－INT_LEN)；
            Lock(g_lock)；
            *p－= 1；
            m_pAddr = t；
            /* 将新指向的内存的引用计数加 1 */
            p = (INT *)((char *)r.m_pAddr－INT_LEN)；
            *p += 1；
            Unlock(g_lock)；
            return r；
        }；
    }；
```

需要指出的是，为了使编码容易理解，上面代码中省去了对指针是否为空的校验。如果要在实际中能够正常运行，还需要加上对需要修改引用计数的指针是否为空的校验。

2) 垃圾回收内存管理函数的编码实现如下。

```
LOCK g_lock；                    /* 用来保护引用计数读写及哈希表操作的锁 */
HASHTABLE *g_pMTable；           /* 多任务哈希表指针 */

INT MGC_Init(INT nBucketCount)
{
    g_lock = LockCreate()；
    if ( g_lock != NULL )
    {
        g_pMTable = HashTable_Create(nBucketCount)；
        if ( g_pMTable != NULL )
        {
```

```
            return CAPI_SUCCESS；
        }
        else
        {
            HashTable_Destroy(g_pMTable, NULL)；
        }
    }
    return CAPI_FAILED；
}

/** 多任务下的垃圾内存收集算法的内存分配函数
    @param size_t size——要分配的内存大小，以字节为单位
    @return void *——成功返回分配到的内存地址；失败返回 NULL
*/
void *MGC_Malloc(size_t size)
{
    void *p = malloc( size + INT_LEN )；
    if ( p == NULL )
    {
        MGC_Collect()；
        p = malloc( size + INT_LEN )；
        if ( p == NULL )
        {
            return NULL；
        }
    }
    Lock(g_lock)；
    HashTable_Insert( g_pMTable, p, HashInt)；
    *((INT *)p) = 0；
    Unlock(g_lock)；
    return (void *)((char *)p + INT_LEN )；
}

/** 多任务下的垃圾内存收集算法的内存释放函数
    @param void *p——要释放的内存地址
    @return void——无
*/
void MGC_Free(void *p)
{
```

```
    void *pFree = (void *)((char *)p－INT_LEN)；
    free(pFree)；
}

/** 多任务下的垃圾内存收集算法的手工释放内存函数
    @param void *p——要释放的内存地址
    @return void——无
*/
void MGC_ManualFree(void *p)
{
    void *pFree = (void *)((char *)p－INT_LEN)；
    Lock(g_lock)；
    HashTable_Delete(g_pMTable, pFree, HashInt, IntCompare, NULL)；
    Unlock(g_lock)；
    free(pFree)；
}

/** 支持多任务的垃圾收集函数，遍历哈希表，将所有引用计数为 0 的内存释放
    @return void——无
*/
void MGC_Collect()
{
    void *p；
    Lock(g_lock)；
    HashTable_EnumBegin(g_pMTable)；
    while ( (p = HashTable_EnumNext(g_pMTable)) != NULL )
    {
        INT *pRef = (INT *)((char *)p－INT_LEN)；
        if ( *pRef == 0 )
        {
            HashTable_Delete(g_pMTable, p, HashInt, IntCompare, NULL)；
            MGC_Free(p)；
        }
    }
    Unlock(g_lock)；
}
```

注意：上面定义全局哈希表对象时使用了另外一个 g_pMTable **变量，主要是有别**

于支持单任务的哈希表对象，便于采取不同的策略进行管理。

3. 使用单独任务进行垃圾收集

实现多任务支持之后，如何进行垃圾收集呢？可以看出在上面实现的MGC_Collect()函数中，只是简单地加锁，然后收集，再解锁。这样做的缺点是当分配的内存数量比较多时，需要耗费大量的时间进行收集，并且在收集的过程中其他的内存操作全部都会被挂起，直到收集完成解锁后，其他的内存操作才能继续，这就是目前实际应用中使用较多的收集方法。本书前面已经介绍了多任务下如何遍历的问题，所以这里要利用本书的多任务算法来实现更好的垃圾收集功能，使得在进行垃圾收集时不影响其他的内存操作，使得应用程序继续运行，让用户感觉不到垃圾收集在运行。

要实现收集时不影响内存操作，必须使用支持多任务的哈希表。考虑到程序效率，就不再像多任务链表那样单独写一个多任务哈希表模块，若写成单独模块，多任务哈希表自己得有一个锁，加上引用计数使用的锁 g_lock 总共有两个锁，需要进行两次加锁解锁操作。而锁的操作相对于内存读写操作是非常耗费时间的，所以还是让哈希表共用 g_lock 锁变量。另外还得发挥多任务的优势，特别是在使用多核 CPU 时，更应该发挥多任务的优势，尤其需要将垃圾回收放到一个单独的任务里运行。下面我们就来实现用单独的垃圾收集任务收集垃圾。

要支持多任务，首先必须定义一个多任务变量如下。

```
MTASK g_pMTask；
```

还得在 MGC_Init()函数里创建 MTASK 对象，修改后的编码如下。

```
/**  多任务下的垃圾内存收集算法的初始化函数
     @param INT nBucketCount——哈希表的 bucket 的数量
     @return INT——成功返回 CAPI_SUCCESS；失败返回 CAPI_FAILED
*/
INT MGC_Init(INT nBucketCount)
{
    g_lock = LockCreate()；
    if ( g_lock != NULL )
    {
        g_pMTable = HashTable_Create(nBucketCount)；
        if ( g_pMTable != NULL )
        {
            g_pMTask = MTask_Create()；
            if ( g_pMTask != NULL )
```

```
                {
                    return CAPI_SUCCESS；
                }
                else
                {
                    HashTable_Destroy(g_pMTable, NULL)；
                    LockClose(g_lock)；
                }
            }
            else
            {
                LockClose(g_lock)；
            }
        }
        return CAPI_FAILED；
    }
```

然后就可以实现整个垃圾收集的退出函数了，编码如下。

```
    /** 垃圾收集的关闭函数，必须在程序运行的最后面执行，建议调用 atexit()
        函数来执行这个函数
        @return void——无
    */
    void MGC_Shutdown()
    {
        MTask_Destroy(g_pMTask)；
        Lock(g_lock)；
        HashTable_Destroy(g_pTable, free)；
        HashTable_Destroy(g_pMTable, free)；
        LockClose(g_lock)；
    }
```

垃圾回收的退出函数实现很简单，先是调用 MTask_Destroy()**函数来释放** MTASK **对象，它会使收集任务退出，当任务退出后，就可以将哈希表释放，包括哈希表中保存的内存都释放。**

可以使用 C **标准库的** atexit()**函数注册** GC_Shutdown()**函数，就可以实现程序退出时自动调用** GC_Shutdown()**函数。**

4. 垃圾收集在实际应用中的进一步考虑

上面已经实现了一个支持多任务的垃圾收集算法，如果要在实际中使用它，还要考虑以下三个方面的问题。

1) 能不能使用 GCPtr 或 MGCPtr 来定义全局变量？

由于全局变量的析构函数执行比 GC_Shutdown() 函数晚，而在析构函数里要操作内存的引用计数，但 GC_Shutdown()函数已经将所有分配的内存都释放了，因此定义全局变量会导致异常。如果要定义全局变量，只需设置一个标志，缺省值为 false，GC_Shutdown()函数执行完后将标志设为 true，在析构函数里判断一下标志，如果标志为 true 则不执行任何操作，这样就可以支持全局变量的定义。

但如果支持全局变量，又会引发新的问题，那就是效率的问题。可能有读者会觉得奇怪，不就是在析构函数里增加一条判断语句吗，对效率影响应该不大吧？通过仔细分析就会发现对标志的访问有多个任务，除了 GC_Shutdown()会对标志进行写操作外，其他的任务都是对标志进行读操作。问题就出在 GC_ShutDown()函数上，当 GC_Shutdown()正在设置标志时，也许某个任务正好在执行一个析构函数，所以必须对标志的读写加上锁保护。当然，在析构函数里判断标志时也要加上锁保护。对 MGCPtr 类来讲，由于析构函数本来就需要加锁、解锁，不会影响效率，但对 GCPtr 来说，效率损失就很大。

所以建议 GCPtr 尽量不要支持全局变量，MGCPtr 可以支持全局变量，支持全局变量的编码较简单，请读者自行实现。

2) 能不能使用 GCPtr 或 MGCPtr 定义指针数组？

本书实现的都是对普通指针的垃圾内存管理，没有考虑指针数组的情况，主要是避免把问题复杂化，让读者能更好地理解。如果要实现对数组的支持，需要对上面的实现做一些修改，必须在 GCPtr 中增加标志记住定义的是否为指针数组，释放的时候需要将数组中每个指针指向的对应内存都释放。具体的实现留给读者作为练习，这里就不再详细介绍了。

3) 使用中有哪些限制？

当使用 GCPtr 时，由于使用 GCPtr 比使用普通指针效率低，有很多时候可能需要使用普通指针来指向 GCPtr 指向的内存进行操作。这种情况下使用 GCPtr 时应有一些限制，如果 GCPtr 指向的内存有其他普通指针也指向这块内存，并且普通指针是在 GCPtr 对象之前被废弃，那么垃圾回收不会有任何问题。但是当普通指针的生存期比指向它的 GCPtr 指针长时，当这块内存的引用计数为 0 时，还有普通指针指向它，此时进行垃圾回收就会造成程序崩溃，所以不能让 GCPtr 指向一个比它生存期更长的普通指针指向的内存。

9.2.4 使用垃圾回收算法来做内存泄漏检查

使用引用计数的方法实现的内存垃圾回收不能自动回收循环引用的内存，循环引用的内存需要使用手工进行释放，但手工释放很可能由于编程人员的失误导致内存泄漏，所以必须检查是否存在内存泄漏。要完成这项任务，只要在程序退出时检查一下哈希表中还有哪些内存的引用计数不为 0，就可以知道有哪些内存还没有被释放，内存没有被释放的原因有以下两种。

① 内存是供全局使用的，必须等到要退出时才能释放；

② 内存中有循环引用，导致不能释放。

还需要知道哈希表中保存的内存地址指向的内存是在程序的哪一行分配的，才能定位到具体发生泄漏的位置。C 语言中有一个宏可以获取源程序中的行号，下面就用这个宏来实现内存泄漏检查。

内存泄漏只是在软件的 DEBUG 版本才需要用到，因此下面就来设计一个 DEBUG 版本的内存垃圾回收。

首先要修改内存中需要保存的数据，除了保存引用计数外，还需要保存分配这块内存的源程序文件名和行号，还要保存分配内存的大小信息。为方便起见，可以将引用计数和内存大小保存在分配内存的头部，行号和文件名保存在分配内存的尾部，如图 9-6 所示。

图 9-6 检查内存泄漏需要保存的数据

关于获取源程序文件名和行号，可以使用宏_FILE_和_LINE_来获取，要注意的是，_LINE_和_FILE_是调用它们的地方的源文件名和行号，如果简单地在 GC_Malloc()函数中使用这两个宏，那么所有调用 GC_Malloc()函数的地方获得的源文件名和行号都是 GC_Malloc()函数的源文件名和行号，所有获取的源文件名和行号均相同，显然达不到预想的目的。

预想的目的是为了获取调用 GC_Malloc()处的源文件名和行号，因此需要将 GC_Malloc()函数定义成宏，这样在调用它的地方就会将编码展开，便能得到正确的调用 GC_Malloc()处的源文件名和行号，有助于确认发生内存泄漏的位置及原因。

通常只要在程序的调试版本中检查内存泄漏就可以了，修改后对应的 GC_Malloc( ) 和 GC_Free( )函数如下。

```
#ifdef _DEBUG
#define GC_Malloc(size)\
```

```
{\
    void *p；\
    INT *q；\
    char *psz；\
    \
    p = malloc( size + DOUBLE_INT_LEN + INT_LEN + strlen(__FILE__) + 1)；\
    if ( p == NULL )\
    {\
        GC_Collect()；\
        p = malloc( size + DOUBLE_INT_LEN + INT_LEN + strlen(__FILE__) + 1)；\
        if ( p == NULL )\
        {\
                return NULL；\
        }\
    }\
    \
    HashTable_Insert( g_pTable, p, HashInt)；\
    \
    *((INT *)p) = 0；\
    *((INT *)p + 1) = size；\
    \
    q = (INT *)((char *p) + size + DOUBLE_INT_LEN)；\
    *q = __LINE__；\
    psz = (char *)p + size + DOUBLE_INT_LEN + INT_LEN；\
    strcpy(psz, __FILE__)；\
    \
    return (void *)((char *)p + DOUBLE_INT_LEN)；\
}

/**  垃圾内存收集算法的内存释放函数
     @param void *p——要释放的内存地址
     @return void——无
*/
void GC_Free(void *p)
{
     void *pFree = (void *)((char *)p－DOUBLE_INT_LEN)；
     free(pFree)；
}
```

```
#endif
```

下面就来实现内存泄漏的检查。内存泄漏检查只需要在程序退出时检查哪些内存的引用计数不为 0，只要对哈希表做一个遍历操作就可以获取哪些内存的引用计数不为 0，编码如下。

```
/** 垃圾内存收集算法的内存泄漏检查函数
    @return void——无
*/
void GC_CheckMemoryLeak()
{
    void *p；
    HashTable_EnumBegin(g_pTable)；
    while ( (p = HashTable_EnumNext(g_pTable)) != NULL )
    {
        INT *pRef；
        INT *pSize；
        INT *pLine；
        char *pszFile；
        pRef = (INT *)p；
        pSize = pRef + 1；
        if ( *pRef   != 0 )   /* 判断引用计数是否为 0 */
        {
            pLine = (INT *)((char *)p + *pSize + DOUBLE_INT_LEN)；
            pszFile = (char *)pLine + INT_LEN；
            printf("File：%s, Line：%d have memory leak.\n", pszFile, *pLine)；
        }
    }
}
```

注意：程序中使用了 printf()函数将文件名和行号打印出来，这里使用 printf()函数只是做一个示意，实际应用中可能需要改成其他类型的信息输出函数。

可以使用函数调用来实现在程序退出时自动调用 GC_CheckMemory()函数。

```
atexit(GC_CheckMemoryLeak)；
```

这里使用了 C 标准库的 atexit()函数，这个函数是在整个程序退出时调用它的参数指向的函数来执行，但是调用的时间比全局变量的释放要早，所以如果使用了全局变量，全局变量也被当作泄漏报告出来。

9.3 实例：动态等尺寸内存管理算法的应用

9.3.1 Emalloc 内存管理的概念

本节用动态等尺寸内存管理算法来构造一个 Emalloc 内存管理的实例。Emalloc 是一个缩写，其中 E 表示等尺寸，主要是用动态等尺寸算法来实现内存管理。前面等尺寸算法中已经提到，在实际应用中，需要分配各种不同大小的内存，通常的做法是将内存分成 8，16，32，64，128，256，512，1 024，2 048，4 096 等尺寸，当需要分配某个大小的内存时，找一个刚好比它大的块分给它，比如要分配一个 100 B 的内存，可以分配一个 128 B 的内存给它。

可以简单地使用第 1 章介绍的排序表来管理多个动态等尺寸内存，如图 9-7 所示。

可以按照等尺寸块大小将各个动态等尺寸内存在表中排好序，依次为 8，16，32，64，…，4096 等字节的动态等尺寸内存。分配时，采用二分查找算法先找一个和要分配的内存最接近且大于等于它的动态等尺寸内存，在里面分配一块内存即可。由于动态等尺寸内存会随着分配内存的数量进行动态增长和动态释放，所以用户不用关心系统到底需要多少内存，只要预先分配一定数量的初始内存，后续它自然会根据分配内存多少进行动态调整。

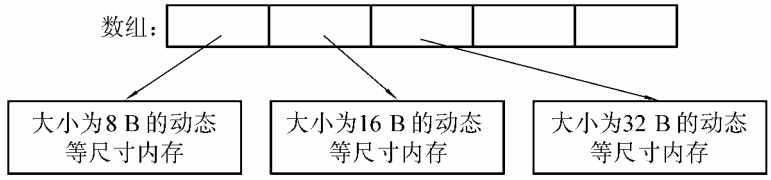


图 9-7 多个动态等尺寸内容分配管理示意图

9.3.2 Emalloc 内存管理的编码实现

下面给出 Emalloc 内存管理的编码实现。

```
#define   SORTTABLE_MAXCOUNT   128
#define   MINDSNODECOUNT       5
#define   DSNODECOUNT          64

SORTTABLE *g_pTable = NULL；

/**  SORTTABLE 的回调函数，用来比较两个 DSPACELIST 对象是否相等
     @param void *p1——第 1 个 DSPACELIST 指针
     @param void *p2——第 2 个 DSPACELIST 指针
```

```
    @return INT——参见 COMPAREFUNC 的说明
*/
INT DSpaceList_Compare(void *p1, void *p2)
{
    DSPACELIST *pList1, *pList2；
    pList1 = (DSPACELIST *)p1；
    pList2 = (DSPACELIST *)p2；
    if ( pList1->uDataSize > pList2->uDataSize )
    {
        return 1；
    }
    else if ( pList1->uDataSize < pList2->uDataSize )
    {
        return－1；
    }
    else
    {
        return 0；
    }
}

/** Emalloc 内存管理的创建函数
    @param void——无
    @return INT——成功返回 CAPI_SUCCESS；失败返回 CAPI_FAILED
*/
INT Emalloc_Create( void )
{
    g_pTable = SortTable_Create(SORTTABLE_MAXCOUNT)；
    if ( g_pTable == NULL )
    {
        return CAPI_FAILED；
    }
    return CAPI_SUCCESS；
}

/** Emalloc 内存管理的整体释放函数
    @param void——无
    @return void——无
*/
```

```
void Emalloc_Destroy(void)
{
    if ( g_pTable != NULL )
    {
        SortTable_Destroy(g_pTable, DSpaceList_Destroy)；
    }
}

/** Emalloc 内存管理的添加函数，添加指定大小动态等尺寸内存块到 Emalloc 内存管理中
    @param UINT uMemSize——动态等尺寸的内存块大小
    @param UINT uMemCount——动态等尺寸的内存块数量
    @return INT——成功返回 CAPI_SUCCESS；失败返回 CAPI_FAILED
*/
INT Emalloc_Add(UINT uMemSize, UINT uMemCount)
{
    DSPACELIST *pList = DSpaceList_Create( uMemSize, uMemCount,
                                           MINDSNODECOUNT, DSNODECOUNT)；
    if ( pList == NULL )
    {
        return CAPI_FAILED；
    }
    return SortTable_Add(g_pTable, pList)；
}

/** Emalloc 内存管理的排序函数，将添加进 Emalloc 内存管理中的动态等尺寸内存
    按内存块大小从小到大排好序
    @return void——无
*/
void Emalloc_Sort()
{
    SortTable_Sort(g_pTable, DSpaceList_Compare)；
}

/** SORTTABLE 的比较回调函数
    @param void *p1——DSPACELIST 指针
    @param void *p2——分配的内存数据大小
    @return INT——参见 COMPAREFUNC 的说明
*/
INT DSpaceList_CmpData(void *p1, void *p2)
{
```

```
    DSPACELIST *pList；
    pList = (DSPACELIST *)p1；
    if ( pList->uDataSize < (UINT)p2 )
    {
        return－1；
    }
    else if ( pList->uDataSize > (UINT)p2 )
    {
        return 1；
    }
    else
    {
        return 0；
    }
}

/** 内存分配函数，从排序表中找一个大小刚好相等或刚好大于的动态等尺寸内存
    进行分配，如果找不到则使用 malloc()进行分配
    @param size_t size——要分配的内存大小，以字节为单位
    @return void *——成功返回分配的内存指针；失败返回 NULL
*/
void *Emalloc(size_t size)
{
    DSPACELIST *pList = SortTable_BlurFind(g_pTable, (void *)size, DSpaceList_CmpData)；
    if ( pList != NULL )
    {
        return DSpaceList_Alloc(pList)；
    }
    else
    {
        return malloc(size)；
    }
}

/** 对应于 Emalloc()内存分配的释放函数
    @param void *p——要释放的内存指针
    @param size_t size——内存大小
    @return void——无
*/
```

```
void Efree(void *p, size_t size)
{
    DSPACELIST *pList = SortTable_BlurFind(g_pTable, (void *)size, DSpaceList_CmpData)；
    if ( pList != NULL )
    {
        DSpaceList_Free(pList, p)；
    }
    else
    {
        free(p)；
    }
}
```

上述编码中，需要使用的函数有 Emalloc_Create(), Emalloc_Destroy(), Emalloc_Add(), Emalloc_Sort(), Emalloc(), Efree()等。如果增加进去的动态等尺寸内存是无序的，就需要调用 Emalloc_Sort()函数进行排序，否则 Emalloc()，Efree()函数将无法按二分查找算法找到对应的内存来进行分配。

在 Efree()函数设计中，使用了两个参数，第二个参数的使用会给用户带来很大的不便。这个参数其实是可以去掉的，这个问题留给读者作为练习。

9.3.3 Emalloc 内存管理的使用方法

例 9-1 Emalloc 内存管理应用简例。试编程实现：将字节数为 8，16，32，…，4 096 的动态等尺寸内存添加到内存管理中，然后从其中分配一个 48 字节的内存再释放，最后调用 Emalloc_Destroy()将内存整体释放掉。

解 为实现上述功能编码如下。

```
#define   MEMORY_BLOCK_SIZE   32 768
main()
{
    int i；
    int uMemSize；
    char *psz；
    uMemSize = 8；
    Emalloc_Create()；
    for ( i = 0；i < 10；i++ )
    {
        Emalloc_Add(uMemSize, MEMORY_BLOCK_SIZE / uMemSize)；
        uMemSize *= 2；
```

```
        }
        psz = Emalloc(48)；
        free(psz, 48)；
        Emalloc_Destroy()；
    }
```

一般情况下，可以像上例编码一样使用 Emalloc 内存管理，Eamlloc_Create()，Emalloc_Add(), Emalloc_Sort()函数只需要在程序初始化时调用，而 Emalloc_Destroy()函数通常在程序退出时使用，Emalloc()和 Efree()函数在任何需要分配内存的地方均可使用。

9.3.4 Emalloc 内存管理的内存越界检查

在实际应用中，经常需要对内存越界进行检查。目前市面上有很多静态或动态的内存越界检查工具可以检查内存越界，但是这些工具只能检查系统分配的内存是否越界，而用户自己实现的内存管理算法如前面介绍的动态等尺寸内存管理算法，其内存分配是在系统分配内存的基础上进行二次分配，如果发生越界，只是发生在动态等尺寸内存管理的内存内部，从系统分配的内存来看并没有发生越界，这样上述工具很难对这种分配的内存进行有效越界检查。

因此有必要实现对用户自己实现的内存管理进行内存越界检查。实现方法是将分配的内存尾部留 4 个字节做校验字节，先填充好某个指定的内容，当程序测试完成后，遍历所有管理的内存，检查各个内存块尾部 4 个字节是否遭到破坏。

当然，由于存在越界时写入的内存有可能刚好和填充的内容相同的情况，所以可以进行多次测试，每次测试时校验字节填充的内容均不相同，如果经过多次测试后校验字节的内容都没有遭到修改，则认为没有越界发生。

经过检查便知道是否有越界发生，但是并不能知道源程序的哪一行操作导致了内存越界发生，因为要拦截内存操作需要在编译器内部才能实现，超出了本书的范畴。所以本书的实现只是检测是否有越界发生，指出内存越界发生的内存地址。

内存越界检查通常都在调试版本中进行，需要用户修改 Emalloc()和 Efree()函数。在分配的内存尾部增加 4 个校验字节，当分配内存时，比要分配的内存多分配 4 个字节，在尾部 4 个字节上填校验字节，在释放时需检验内存尾部 4 个校验字节是否被破坏，如果破坏，打印出发生越界的内存地址及这块内存的大小。以下便是修改后的 Emalloc()函数和 Efree()函数。

```
#ifdef _DEBUG
#define    MEMORY_CHECKSUM        0xeeeeeeee
```

```
/** 内存分配函数，从排序表中找一个等于或大于的动态等尺寸内存
    进行分配，如果找不到则使用 malloc()进行分配
    @param size_t size——要分配的内存大小，以字节为单位
    @return void *——成功返回分配的内存指针；失败返回 NULL
*/
void *Emalloc(size_t size)
{
    void *p；
    DSPACELIST *pList = SortTable_BlurFind(g_pTable, (void *)(size+4),
                                DSpaceList_CmpData)；
    if ( pList != NULL )
    {
        p = DSpaceList_Alloc(pList)；
    }
    else
    {
        p = malloc(size + 4)；
    }

    if ( p != NULL )
    {
        *((INT *)((char *)p + size)) = MEMORY_CHECKSUM；
    }
}

/** 对应于 Emalloc()内存分配的释放函数
    @param void *p——要释放的内存指针
    @param size_t size——内存大小
    @return void——无
*/
void Efree(void *p, size_t size)
{
    DSPACELIST *pList；
    if ( *((INT *)((char *)p + size)) != MEMORY_CHECKSUM )
    {
        printf("Memory address overrun：Address=%ld, size=%d\n", (INT)p, size)；
    }
    pList = SortTable_BlurFind(g_pTable, (void *)size, DSpaceList_CmpData)；
    if ( pList != NULL )
```

```
        {
            DSpaceList_Free(pList, p)；
        }
        else
        {
            free(p)；
        }
    }
    #endif      /* _DEBUG */
```

以上代码用于释放内存时检查是否有越界发生，当然用户也可能会忘记释放内存，因此可以考虑和前面的垃圾内存管理中的内存泄漏检查结合起来使用，这个问题就留给读者作为练习。

本 章 小 结

本章主要介绍了动态等尺寸内存管理算法以及使用引用计数的垃圾内存回收算法，还介绍了动态等尺寸内存管理算法的应用实例——Emalloc 内存管理。C/C++软件设计中，内存管理策略的设计是非常重要的环节。本章所介绍的动态等尺寸内存管理算法，可以有效地消除内存碎片，且效率很高，对系统内存的使用也可以动态增长和释放，是一个应用前景美好的算法。

垃圾内存回收算法则是在用户程序层实现的，采用引用计数方法，并可以采用手工释放方式来消除内存循环引用的情况。对手工释放可能导致的内存泄漏，设计了内存泄漏检查方法，同样是一个可以在实际中投入商业使用的垃圾内存回收管理算法。

习题与思考

1. 将 9.3.2 节中的 Efree(void *p, size_t size) 函数的第 2 个参数去掉，如何修改编码来实现它?
2. 如何将本章介绍的内存泄漏检查和内存越界检查结合起来使用？
3. 如果原来已经写好了一个软件，如何在不修改原来代码的基础上将其中的内存管理算法替换成本章所介绍的动态等尺寸内存分配算法？

附

参考文献

1. Herbert Schildt. C++**编程艺术**. **曹蓉蓉，刘小荷译**. **北京：清华大学出版社**, 2005
2. **卢开澄**. **计算机算法导引——设计与分析**. **北京：清华大学出版社**, 1996
3. Robert Sedgewick. C++**算法——图算法**. **林琪译**. **北京：清华大学出版社**, 2003
4. Sartaj Sahni. **数据结构算法与应用——**C++**语言描述**. **汪诗林，孙晓东等译**. **北京：机械工业出版社**, 2000
5. Clifford A. Shaffer. **数据结构与算法分析(英文版)**. **北京：电子工业出版社**, 2004
6. Robert L. Kruse. **数据结构与程序设计**——C **语言(第二版)**. **敖富江译**. **北京：清华大学出版社**, 2005
7. William J. Collins. **数据结构与** STL. **周翔译**. **北京：机械工业出版社**, 2004
8. Richard Jones, Rafael Lins. **垃圾收集**. **谢之易译**. **北京：人民邮电出版社，**2004
9. Adam Drozdek. **数据结构与算法——**C++**版(第** 2 **版)**. **北京：清华大学出版社**, 2003
10. Hector Garcia-Molina Jeffrey D, Ullman Jennifer Widom. **数据系统实现(英文版)**. **北京：机械工业出版社**, 2001
11. Stephen D. Huston, James CE Johnson, Umar Syyid. ACE **程序员指南——网络与系统编程的实用设计模式**. **马维达译**. **北京：中国电力出版社**, 2004
12. Herb Sutter. Exceptional C++**中文版**. **卓小涛泽**. **北京：中国电力出版社**, 2003
13. Rene. Alexander, Graham. Benstey. C++**高效编程：内存与性能优化**. **王峰**, **史金虎译**. **北京：中国电力出版社**, 2003
14. Stanley B. Lippman, Josee Lajoie. C++ Primer **第三版(中文版)**. **潘爱民，张丽译**. **北京：中国电力出版社**, 2002
15. Scott Meyers. Effective C++**中文版**. **侯捷译**. **武汉：华中科技大学出版社**, 2001
16. Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman. **数据结构与算法(英文版)**. **北京：清华大学出版社**, 2003

17. Chistos. H. Papadimitriou. **计算复杂性(英文版). 北京：清华大学出版社**, 2004
18. James Noble, Charles Weir. **内存受限系统之软件开发. 侯捷等译. 武汉：华中科技大学出版社**, 2003
19. Jon Bentley. **编程珠玑(第** 2 **版). 谢君英, 石朝江译. 北京：中国电力出版社**, 2004
20. Robert Sedgewich. **算法Ⅰ—Ⅳ**(C++**实现)：基础、数据结构、排序和搜索(第** 3 **版). 张铭泽等译. 北京：中国电力出版社**, 2004
21. Kris Kaspersky. **代码优化：有效使用内存. 谭明金译. 北京：电子工业出版社**, 2004
22. **周培德. 算法设计与分析. 北京：机械工业出版社**, 1996
23. Andrew S. Tanenbaum. **现代操作系统. 陈向群等译. 北京：机械工业出版社**,1999
24. Gary R. Wright, W. Richard Stevens. TCP/IP **详解卷** 2**：实现. 陆雪莹等译. 北京：机械工业出版社**, 2000
25. Kay A. Robbins, Steven Robbins. **实用** UNIX **编程. 刘宗田等译. 北京：机械工业出版社**, 1999
26. **孔祥营, 柏桂枝. 嵌入式实时操作系统** VxWorks **及其开发环境** Tornado. **北京：中国电力出版社**, 2002
27. **唐恒永, 赵传立. 排序引论. 北京：科学出版社**, 2002
28. Nancy A. Lynch. **分布式算法. 舒继武等译. 北京：机械工业出版社. 中信出版社** 2004
29. **王树禾. 离散数学原理之二——图论及其算法. 合肥：中国科学技术大学出版社**, 1990
30. Thomas Krantz,Virgile Mogbil. Note Encoding Hamiltonian circuits into multiplicative linear logic. Theoretical Computer Science, 2001, 266：987～996
31. Michael Hoffmann, Csaba D. Tóth. Alternating paths through disjoint line segments. Information Processing Letters, 2003,87：287～294
32. Yuanqiu Huang. Maximum genus of a graph in terms of its embedding properties. Discrete Mathematics, 2003, 262：171～180
33. M. Funk, Bill Jackson, D. Labbate, J. Sheehan. 2-Factor hamiltonian graphs. Journal of Combinatorial Theory,Series B,2003,87：138～144
34. Andrew Thomason. A simple linear expected time algorithm for finding a hamilton path. Discrete Mathematics, 1989, 75：373～379

《多任务下的数据结构与算法》读者信息反馈卡

尊敬的读者：

感谢您对我们的支持与厚爱。为了今后能为您提供更优秀的图书，请您抽出宝贵的时间将您的意见以下表(可另附页)的方式及时告知我们，以利我们工作的改进。谢谢！我们将从中评选出热心读者若干名，免费赠阅我们以后出版的图书。

1. **读者信息**

姓名：_____ **性别：**____ **年龄：**____ **职业：**_____ **学历：**_____

电话：_____ E-mail：_____

通信地址：_____ **邮编：**_____

影响您购买本书的原因：_____

2. **对本书的评价**

技术内容：□很满意 □比较满意 □一般 □较不满意 □不满意

改进意见_____

文字质量：□很满意 □比较满意 □一般 □较不满意 □不满意

改进意见_____

封面设计：□很满意 □比较满意 □一般 □较不满意 □不满意

改进意见_____

版式设计：□很满意 □比较满意 □一般 □较不满意 □不满意

改进意见_____

印装质量：□很满意 □比较满意 □一般 □较不满意 □不满意

改进意见_____

3. **您认为本书有哪些地方需要改进？**

哪些章节需要精简？_____

哪些内容需要更新？_____

4. **您感兴趣或希望看到的图书选题有：**_____

请将读者信息反馈卡寄至：

4	3	0	0	7	4

湖北省武汉市珞喻路 1037 **号，华中科技大学出版社** 501 **室**

王红梅 (收)