第5章 堆 栈

堆栈和队列可能是使用频率最高的数据结构，二者都来自于第 3章中的线性表数据结构（经过某种限制以后）。本章将研究堆栈，下一章研究队列。堆栈数据结构是通过对线性表的插入和删除操作进行限制而得到的（插入和删除操作都必须在表的同一端完成），因此，堆栈是一个后进先出（last-in-first-out, LIFO）的数据结构。

由于堆栈是一种特殊的线性表，因此可以很自然地从相应的线性表类中派生出堆栈类。我们可以从程序3-1的LinearList类派生出基于公式描述的堆栈类，也可以从程序 3-8的Chain类派生出基于链表结构的堆栈类。通过类的派生，可以大大简化程序设计的任务，但同时代码的执行效率有明显损失。由于堆栈是一个很基本的数据结构，许多程序都要用到堆栈，本章中也直接给出了基于公式描述和基于链表结构的堆栈类（而不是从其他类派生而来）。这两种类与对应的派生类相比，在执行效率上将有很大提高。

本章还给出六个使用堆栈的应用程序。第一个应用是一个用来匹配表达式中左、右括号的简单程序；第二个应用是一个经典的汉诺塔问题求解程序。汉诺塔问题要求把一个塔上的所有碟子按照一定的规则搬到另一个塔上，每次只能搬动一个碟子，其间可以借助于第 3个塔的帮助。在汉诺塔问题的求解过程中，每个塔都被视为一个堆栈；第三个应用使用堆栈来解决火车车厢重排问题，其目标是把火车车厢按所希望的次序重新排列；第四个应用是关于电子布线的问题，在这个应用中，用堆栈来确定一个电路是否可以成功布线；第五个应用用于解决 3.8.3节所介绍的离线等价类问题。采用堆栈可以帮助我们在线性时间内确定等价类；最后一个应用用来解决经典的迷宫问题，即寻找一条从入口到出口的迷宫路径。应该仔细地研究这个应用，因为它体现了许多软件工程的原理和思想。

本章中所用到的C++语言的新的特性是派生类和继承。

5.1 抽象数据类型

定义 **[堆栈]** 堆栈（stack）是一个线性表，其插入（也称为添加）和删除操作都在表的同一端进行。其中一端被称为栈顶（top），另一端被称为栈底（bottom）。

图5-1a 给出了一个四元素的堆栈。假定希望在 5-1a 的堆栈中添加一个元素E，这个元素将被放到元素D的顶部，所得到的结果如图5-1b 所示。如果想从5-1b 的堆栈中删除一个元素，那么元素E将被删除，删除E之后又将得到5-1a 的结果。如果对5-1b 的堆栈连续执行三次删除操作，则将得到图5-1c 所示的堆栈。

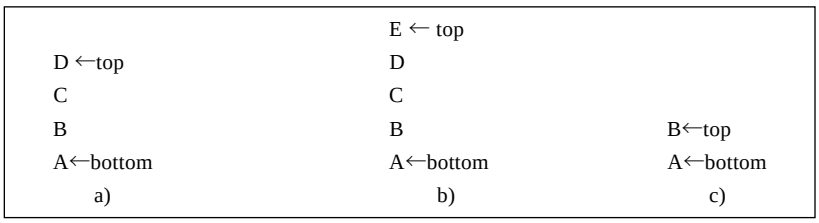


图5-1 堆栈结构

从上面的讨论中可以看出，堆栈是一个后进先出表。这种类型的表在计算过程中将频繁使用。ADT堆栈的描述见ADT 5-1。

ADT5-1 **堆栈的抽象数据类型描述**

抽象数据类型*Stack*{

实例

元素线性表，栈底，栈顶

操作

Create()：创建一个空的堆栈

IsEmpty()：如果堆栈为空，则返回true，否则返回false

IsFull()：如果堆栈满，则返回true，否则返回false

Top()：返回栈顶元素

Add(*x*)：向堆栈中添加元素*x*

Delete(*x*)：删除栈顶元素，并将它传递给*x*

}

5.2 派生类和继承

我们经常会处理这样一类新的数据对象，它是某种更通用的数据对象的特殊版本或限制版本。例如，本章中的堆栈数据对象就是更通用的线性表对象的限制版本。堆栈数据对象的实例也是线性表数据对象的实例，而且，所有的堆栈操作都可以利用线性表操作来实现。例如，如果把表的左端定义为栈底，右端定义为栈顶，那么堆栈的添加操作等价于在表的右端进行插入操作，删除操作等价于在表的右端进行删除操作。

根据上述观察，我们期望如果能够明确地把堆栈表示成特殊的线性表，那么可以简化 C++堆栈类的实现，而且，可以参照线性表的操作来定义堆栈操作。

若类B是另一个类A的限制版本，那么可以从A派生出B。我们称A为基类，B为派生类。从类A派生出的类B继承了基类A的所有成员——共享成员、保护成员和私有成员。类型为 B的每个对象都与A所有的数据成员和函数相关联。类B可以采用如下三种基本方式之一来继承类A的成员：共享成员、保护成员和私有成员。比如，对于共享成员方式，可以采用如下语法形式：

```
class B: public A
```

一个类可以从多个类派生而来。若类 B从A和C派生而来，并且以共享成员方式继承 A的属性，以私有成员方式继承C的属性，相应的语法形式如下：

```
class B: public A, private C
```

在所有继承方式中，基类A的私有成员仍是A的私有成员，类B的成员不能够访问它们。不同的继承方式仅影响对基类的保护成员和共享成员的访问。

当B按共享成员方式从A派生而来时，A的保护成员成为B的保护成员，A的共享成员成为B的共享成员。所以在编写 B的成员代码时，可以直接访问 A的共享成员和保护成员，而不能够访问A的私有成员。如果X的类型为B，那么，仅当F是B或A的共享成员时，用户才可以使用语句X.F（）来执行X中的F函数。

如果继承方式为保护成员，那么A中的共享成员和保护成员均成为B的保护成员。如果X和F如上所述，那么仅当函数F是B的共享成员时，用户才可以执行X..F()操作。

如果继承方式为私有成员，那么A中的共享成员和保护成员均成为B的私有成员。

所有的继承方式都可以加上一个前缀virtual，第12章将介绍这个前缀的含义。

5.3 公式化描述

由于堆栈是一个受限的线性表（插入和删除操作仅能在表的同一端进行），因此可以参考3.3节的线性表描述，令栈顶元素存储在element[length-1]中，栈底元素存储在element[0]中。程序5-1中定义的Stack类是从程序3-1的LinearList类派生而来。由于继承方式为私有成员，因此LinearList的共享成员和保护成员都可以被Stack的类成员访问，而LinearList的私有成员不可以被Stack的类成员访问。

程序5-1 公式化描述的堆栈类

```
template<class T>
class Stack :: private LinearList <T>{
// LIFO 对象
  public:
    Stack(int MaxStackSize = 10)
       ：LinearList<T>  (MaxStackSize)  {}
    bool IsEmpty() const
       {return LinearList<T>::IsEmpty();}
    bool IsFull() const
       {return (Length() == GetMaxSize());}
    T Top()  const
      {if (IsEmpty()) throw OutOfBounds();
      T x; Find(Length(), x); return x;}
    Stack<T>& Add(const T& x)
      {Insert(Length(), x); return *this;}
    Stack<T>& Delete(T& x)
      {LinearList<T>::Delete(Length(), x);
      return *this;}
};
```

Stack的构造函数简单地调用线性表的构造函数，提供的参数为堆栈的大小MaxStackSize。没有为Stack类定义析构函数，因此当Stack类型的对象被删除时，将自动调用LinearList的析构函数。IsEmpty()函数是对线性表相应函数的简单调用。使用操作符 :: 来区分基类和派生类中的同名成员。

在实现函数IsFull时，由于Stack的成员不能访问LinearList的私有成员，因此有一定的困难。为了实现这个函数，必须知道 LinearList<T>::MaxSize的值。可以通过把 MaxSize定义成LinearList的保护成员来克服这个问题。然而，假如我们在稍后又修改了 LinearList的定义并删除了成员 MaxSize，那么就不得不同时修改 Stack的定义。一个比较好的解决方法是为类LinearList增加一个保护成员GetMaxSize()，语法如下：

```
protected:
  int GetMaxSize()  const {return MaxSize;}
```

这个函数可以被 Stack的成员所访问。如果 LinearList的实现发生变化，那么只需修改GetMaxSize的实现，而不必修改它的任何派生类。另一种解决办法是为类 LinearList定义一个IsFull成员函数。

在程序5-1的Stack<T>::IsFull代码中，没有使用语法LinearList<T>::Length()，因为在Stack

中没有定义Length()函数。

函数Top首先判断堆栈是否为空，如果为空，则不存在栈顶元素，引发一个 OutOfBounds异常。当堆栈不为空时，调用线性表的Find函数来查找最右端的元素。

函数Add在栈顶添加一个元素x，做法是在线性表的最右端插入元素 x。Delete函数删除栈顶元素，并把该元素赋予x。堆栈的删除操作是通过在线性表的最右端执行删除操作而完成的。Add函数和Delete函数均返回变化后的堆栈。 Add函数和Delete函数都不能捕获可能由 Insert或LinearList::Delete引发的异常。它们把异常留给调用它们的函数来处理。

假定X是一个类型为Stack的对象，当F 是Stack的任一个共享函数时， X 的使用者可以执行语句X.F。然而，对于LinearList中的函数（如Length），不可以使用语句X.F，因为LinearList中的函数都不是Stack的共享成员（因继承方式为私有成员）。

5.3.1 Stack的效率

当T是一个内部数据类型时，堆栈的构造函数和析构函数的复杂性均为 $\Theta(1)$，当T是用户自定义的类时，构造函数和析构函数的复杂性均为 O(MaxStackSize)。其余每个堆栈操作的复杂性均为$\Theta(1)$。注意通过从LinearList派生Stack，一方面大大减少了编码量，另一方面也使程序的可靠性得到很大提高，因为 LinearList经过测试并被认为是正确的。然而，不幸的是，代码编写的简化带来了运行效率的损失。例如，为了向堆栈中添加一个元素，首先要确定堆栈的长度length()，然后调用函数Insert()。Insert函数首先必须判断插入操作是否会越界，然后需要付出一个for循环的开销来执行0个元素的移动。为了消除额外的开销，可以把 Stack定义成一个基类，而不是派生类。

另一种潜在的问题是派生类 Stack也会受到LinearList本身所受限制的影响。例如，必须为数据类型为T的成员定义操作符 <<和==，因为前者用于对线性表操作 <<的重载，后者用于对LinearList::Search的重载。

5.3.2 自定义Stack

程序5-2把Stack类定义为一个基类，所采用的描述形式与线性表相同。堆栈元素存储在数组stack之中，top用于指向栈顶元素。堆栈的容量为MaxTop+1。

程序5-2 自定义Stack

```
template<class T>
class Stack{
// LIFO 对象
  public:
     Stack(int MaxStackSize = 10);
     ~Stack () {delete [] stack;}
     bool IsEmpty() const {return top == -1;}
     bool IsFull() const {return top == MaxTop;}
     T Top() const;
     Stack<T>& Add(const T& x);
     Stack<T>& Delete(T& x);
  private:
     int top;     // 栈顶
     int MaxTop;  // 最大的栈顶值
```

```
    T *stack;    // 堆栈元素数组
};
template<class T>
Stack<T>::Stack(int MaxStackSize)
{// Stack 类构造函数
  MaxTop = MaxStackSize - 1;
  stack = new T[MaxStackSize];
  top = -1;
}
template<class T>
T Stack<T>::Top() const
{// 返回栈顶元素
  if (IsEmpty()) throw OutOfBounds();
  else return stack[top];
}
template<class T>
Stack<T>& Stack<T>::Add(const T& x)
{//添加元素x
if (IsFull())  throw NoMem();
   stack[++top] = x;
   return *this;
}
template<class T>
Stack<T>& Stack<T>::Delete(T& x)
{// 删除栈顶元素，并将其送入x
  if (IsEmpty())  throw OutOfBounds();
  x = stack[top--];
  return *this;
}
```

采用一个for循环执行100 000次堆栈添加操作和删除操作来进行实际的运行测试，结果发现程序5-1比程序5-2多用了50%多的时间。

练习

1. 对ADT堆栈进行扩充，增加以下函数：

1) 确定堆栈的大小（即堆栈中元素的数目）。

2) 输入一个堆栈。

3) 输出一个堆栈。

扩充基于公式化描述的堆栈定义，增加上述成员函数。编写相应的代码并进行测试。

2. 对ADT堆栈进行扩充，增加以下函数：

1) 把堆栈拆分为两个部分，第一部分包含从栈底开始的一半元素，第二部分包含其余的元素。

2) 合并两个堆栈，把第二个堆栈的所有元素放到第一个堆栈的顶部，并且第二个堆栈中元素的相对次序不发生变化。合并完成后，第二个堆栈为空。

扩充基于公式化描述的堆栈定义，增加上述成员函数。编写相应的代码并进行测试。

3. 基于公式化描述的堆栈定义所存在的缺陷是在创建堆栈时，必须指定MaxStackSize的值。

克服这种缺陷的一种方法是在创建堆栈时取 MaxTop=0。如果在Add操作期间没有可用的空间来容纳新的元素，可将MaxTop变成2*MaxTop+1，然后分配一个新的大小为MaxTop+1的数组，并将原数组中的元素复制到新数组中，最后删除原数组。类似地，在删除操作期间，如果表的大小变成数组容量的1/4，则可以分配一个容量为原数组一半的新数组，并将原数组中的元素复制到新数组中，最后删除原数组。

1) 采用上述思想重新实现自定义Stack。构造函数不带参数，其任务是置MaxTop=0，分配一个容量为1的数组，并置top为－1。

2) 考察从一个空堆栈开始，连续执行 n 个添加和删除操作。假定采用原来的方法时总的执行步数为 $f(n)$。证明对于新的方法，执行步数最多为 $cf(n)$，其中 c 为常量。

5.4 链表描述

虽然上一节给出的用数组实现堆栈的方法既优雅又高效，但若同时使用多个堆栈，这种方法将浪费大量的空间。其原因与3.3.4节所分析的线性表（用单个数组实现）空间利用率不高的原因相同。不过，若仅同时使用两个堆栈，则是一种例外。可以让一个堆栈从数组的起始位置向后延伸，而让另一个堆栈从数组的结束位置（ MaxSize-1）向前延伸，这样可以保持空间的利用率和运行效率。两个堆栈都向数组的中部延伸（如图5-2所示）。在描述两个以上的堆栈时，可以借鉴在一个数组中描述多个线性表的方法。不过，这样做在提高空间利用率的同时，却使Add操作在最坏情况下的时间复杂性从 $\Theta(1)$变成了O(ArraySize)。Delete操作的时间复杂性仍然保持为 $\Theta(1)$。

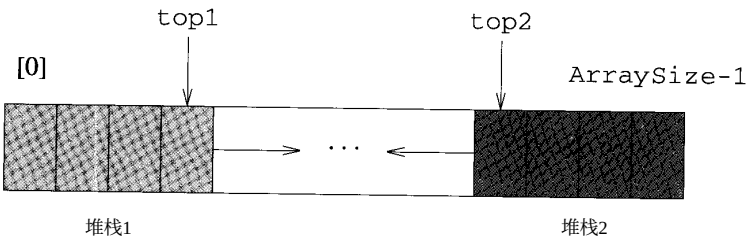


图5-2 一个数组中的两个堆栈

可以采用一个堆栈对应一个链表的方法来高效地描述多个堆栈。这种描述方法使每个堆栈元素多用了一个额外的指针空间。不过，每个堆栈操作的时间复杂性均变为 $\Theta(1)$。

在使用链表来表示堆栈时，必须确定链表的哪一端对应于栈顶。如果把链表的右端作为栈顶，那么可以利用链表操作Insert(n,x)和Delete(n,x)来实现堆栈的插入和删除操作，其中n为链表中的节点数目。这两种链表操作的时间复杂性均为 $\Theta(n)$。另一方面，如果把链表的左端作为栈顶，则可以利用链表操作Insert(0,x)和Delete(1,x)来实现堆栈的插入和删除操作。这两种链表操作的时间复杂性均为 $\Theta(1)$。以上分析表明，应该把链表的左端作为栈顶。

程序5-3给出了从Chain类派生而来的链表形式的堆栈。链表的左端为栈顶，右端为栈底。

程序5-3 从Chain派生的链表形式的堆栈

```
template<class T>
class LinkedStack : private Chain<T> {
  public:
```

```
    bool IsEmpty() const
      {return Chain<T>::IsEmpty();}
    bool IsFull() const;
    T Top() const
      {if (IsEmpty()) throw OutOfBounds();
       T x; Find(1, x); return x;}
    LinkedStack<T>& Add(const T& x)
      {Insert(0, x); return *this;}
    LinkedStack<T>& Delete(T& x)
      {Chain<T>::Delete(1, x); return *this;}
};
template<class T>
bool LinkedStack<T>::IsFu11() const
{// 堆栈是否满?
    try {ChainNode<T> *p = new ChainNode<T>;
      delete p; return false;}
    catch (NoMem) {return true;}
}
```

IsFull的实现不够优雅，因为它是通过实际创建一个类型为 Node的节点来判断能否添加一个新元素。判断之后必须删除所创建的节点，因为并不打算使用所创建的节点。

与程序5-1的情形一样，为了提高运行效率，也可以自定义一个用链表形式的堆栈。相应的代码见程序5-4。

对于执行10 000次添加和删除操作的for循环来说，程序5-3的代码将比程序5-4的代码多耗时25%。

程序5-4 自定义链表形式的堆栈

```
template <class T>
class Node{
  friend LinkedStack<T>;
  private:
    T data;
    Node<T> *link;
};
template<class T>
class LinkedStack {
  public:
    LinkedStack () {top = 0;}
    ~LinkedStack();
    bool IsEmpty() const {return top==0;}
    bool IsFull() const;
    T Top() const;
    LinkedStack<T>& Add(const T& x);
    LinkedStack<T>& Delete(T& x);
  private:
    Node<T> *top; // 指向栈顶节点
};
template<class T>
LinkedStack<T>::~LinkedStack()
```

```
{// 析构函数
  Node<T> *next;
  while (top) {
    next = top->link;
    delete top;
    top = next;
    }
}
template<class T>
bool LinkedStack<T>::IsFu11() const
{// 堆栈是否满?
  try {Node<T> *p = new Node<T>;
     delete p;
     return false;}
  catch (NoMem) {return true;}
}
template<class T>
T LinkedStack<T>::Top() const
{// 返回栈顶元素
  if (IsEmpty())  throw OutOfBounds();
  return top->data;
}
template<class T>
LinkedStack<T>& LinkedStack<T>::Add(const T& x)
{// 添加元素 x
  Node<T> *p = new Node<T>;
  p->data = x;
  p->link = top;
  top = p;
  return *this;
}
template<class T>
LinkedStack<T>& LinkedStack<T>::Delete(T& x)
{// 删除栈顶元素，并将其送入x
    if (IsEmpty()) throw OutOfBounds();
    x = top->data;
    Node<T> *p = top;
    top = top->link;
    delete p;
    return *this;
}
```

练习

4. 1) 编写一个测试程序，用它来测量100 000个交替的堆栈添加和删除操作所需要的运行时间。分别对程序5-1、5-2、5-3、5-4以及练习3进行测试。

2) 交替的添加和删除操作对于练习3的代码来说是最理想的情形。请给出能使练习3得到最坏的时间复杂性的测试数据，同时利用该数据测量100 000次堆栈操作所需要的时间。

5. 扩充LinkedStack类的定义，增加以下堆栈操作：

1) 确定堆栈的大小（即堆栈中元素的数目）。

2) 输入一个堆栈。

3) 输出一个堆栈。

6. 扩充LinkedStack类的定义，增加以下操作：

1) 把堆栈拆分为两个部分，第一部分包含从栈底开始的一半元素，第二部分包含其余的元素。

2) 合并两个堆栈，把第二个堆栈的所有元素放到第一个堆栈的顶部，并且第二个堆栈中元素的相对次序不发生变化。合并完成后，第二个堆栈为空。

5.5 应用

5.5.1 括号匹配

在这个问题中将要匹配一个字符串中的左、右括号。例如，字符串 (a*(b+c)+d)在位置1和4有左括号，在位置8和11有右括号。位置1的左括号匹配位置11的右括号，位置4的左括号匹配位置8的右括号。对于字符串 (a+b))(，位置6的右括号没有可匹配的左括号，位置 7的左括号没有可匹配的右括号。我们的目标是编写一个 C++程序，其输入为一个字符串，输出为相互匹配的括号以及未能匹配的括号。注意，括号匹配问题可用来解决 C++程序中的{和}的匹配问题。

可以观察到，如果从左至右扫描一个字符串，那么每个右括号将与最近遇到的那个未匹配的左括号相匹配。这种观察结果使我们联想到可以在从左至右的扫描过程中把所遇到的左括号存放到堆栈内。每当遇到一个右括号时，就将它与栈顶的左括号（如果存在）相匹配，同时从栈顶删除该左括号。程序5-5给出了完整的C++程序。图5-3给出了某次运行所得到的输入/输出结果。程序5-5的时间复杂性为$\Theta(n)$，其中n 为输入串的长度。

程序5-5 产生匹配括号的程序

```
#include <iostream.h>
#include <string.h>
#include <stdio.h>
#include "stack.h"
const int MaxLength = 100; // 最大的字符串长度
void PrintMatchedPairs(char *expr)
{// 括号匹配
  Stack<int> s(MaxLength);
  int j, length = strlen(expr);
  // 从表达式 expr 中搜索 ( 和 )
for (int i = 1; I<=length; i++) {
  if (expr[i - 1] ==' ( ' )  s.Add(I);
  else if (expr[i - 1] ==' ) ' )
    try{s.Delete(j);
       cout << j <<'  ' << i << endl;}
  catch (OutOfBounds)
    { cout << "No match for right parenthesis" << " at " << i << endl;}
  }
// 堆栈中所剩下的(都是未匹配的
while(!s.IsEmpty()) {
```

```
    s.Delete(j);
    cout << "No match for left parenthesis at " <<j< endl;}
}
void main(void)
{
    char expr[MaxLength];
    cout << "Type an expression of length at most " << MaxLength << endl;
    cin.getline(expr, MaxLength);
    cout <<"The pairs of matching parentheses in" << endl;
    puts (expr);
    cout <<"are" << endl;
    PrintMatcnedPairs(expr);
}
```

```
Type an expression of length at most 100
(d+(a+b)*c*(d+e)-f))(()
The pairs of matching parentheses in the expression
(d+(a+b)*c*(d+e)-f))(()
are
4 8
12 16
1 19
No match for right parenthesis at 20
22 23
No match for left parenthesis at 21
```

图5-3 括号匹配程序运行示例

5.5.2 汉诺塔

汉诺塔（Towers of Hanoi）问题来自一个古老的传说：在世界刚被创建的时候有一座钻石宝塔（塔1），其上有64个金碟（如图5-4所示）。所有碟子按从大到小的次序从塔底堆放至塔顶。紧挨着这座塔有另外两个钻石宝塔（塔 2和塔3）。从世界创始之日起，婆罗门的牧师们就一直在试图把塔1上的碟子移动到塔2上去，其间借助于塔3的帮助。由于碟子非常重，因此，每次只能移动一个碟子。另外，任何时候都不能把一个碟子放在比它小的碟子上面。按照这个传说，当牧师们完成他们的任务之后，世界末日也就到了。

在汉诺塔问题中，已知n个碟子和3座塔。初始时所有的碟子按从大到小次序从塔 1的底部堆放至顶部，我们需要把碟子都移动到塔 2，每次移动一个碟子，而且任何时候都不能把大碟子放到小碟子的上面。在继续往下阅读之前，可以先尝试对 n=2,3和4来解决这个问题。

一个非常优雅的解决办法是使用递归。为了把最大的碟子移动到塔 2，必须把其余n-1个碟子移动到塔3，然后把最大的碟子移动到塔 2。接下来是把塔3上的n-1个碟子移动到塔2，为此可以利用塔2和塔1。可以完全忽视塔2上已经有一个碟子的事实，因为这个碟子比塔 3上将要移过来的任一个碟子都大，因此，可以在它上面堆放任何碟子。程序 5-6给出了按递归方式实现的C++代码。初始调用的语句是TowersOfHanoi(n,1,2,3)。程序5-6的正确性很容易证明。

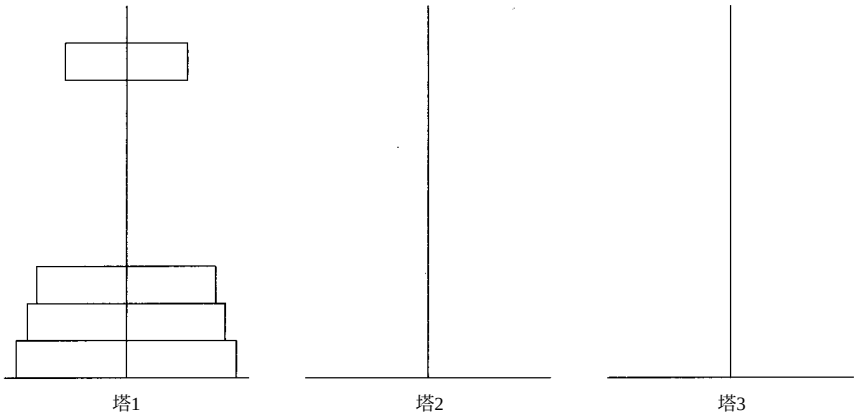


图5-4 汉诺塔

程序5-6 求解汉诺塔问题的递归程序

```
void TowersOfHanoi(int n, int x, int y, int z)
{//把 n 个碟子从塔x 移动到塔 y，可借助于塔z
  if (n > 0) {
    TowersOfHanoi(n-1, x,z,y);
    cout << "Move top disk from tower " << x <<" to top of tower " << y << endl;
    TowersOfHanoi(n-l, z, y, x);}
}
```

程序5-6所花费的时间正比于所输出的信息行数目，而信息行的数目则等价于碟子移动的次数。考察程序5-6，可以得到碟子的移动次数*moves* (*n*)如下：

$$moves(n) = \begin{cases} 0 & n=0 \\ 2moves(n-1)+1 & n>0 \end{cases}$$

可以使用第 2章介绍的叠代方法（见程序 2-20）来计算这个公式。得到的结果应为 $moves(n)=2^n-1$。可以证明，2^n-1实际上是最少的移动次数。在婆罗门宝塔中 $n=64$，因此婆罗门牧师们用不了多少年就可以完成任务。根据上面的公式，可以断定函数 TowersOfHanoi的复杂性为$\Theta(2^n)$。

程序5-6的输出给出了把碟子从塔 1移动到塔2所需要的碟子移动次序。假定希望给出每次移动之后三座塔的状态（即塔上的碟子及其次序），那么必须在内存中保留塔的状态，并在每次移动碟子之后，对塔的状态进行修改。这样每移动一个碟子时，就可以在一个输出设备（如计算机屏幕、打印机等）上输出塔的信息。

由于从每个塔上移走碟子时是按照 LIFO的方式进行的，因此可以把每个塔表示成一个堆栈。三座塔在任何时候都总共拥有 n 个碟子，因此，如果使用链表形式的堆栈，只需申请 n个元素所需要的空间。如果使用的是基于公式化描述的堆栈，塔 1和塔2的容量都必须是n，而塔3的容量必须为n-1，因而所需要的空间总数为 3n-1。前面的分析已经指出，汉诺塔问题的复杂性是以n 为指数的函数，因此在可以接受的时间范围内，只能解决 n 值比较小（如$n \leq 30$）的汉诺塔问题。对于这些较小的n 值，基于公式描述和基于链表描述的堆栈在空间需求上的差别相当小，因此可以随意使用。

程序5-7的代码使用了基于公式描述的堆栈。 TowersOfHanoi(n)是递归函数Hanoi::TowersOfHanoi的预处理程序，它是根据程序5-6的模式来设计的。预处理程序创建三个堆栈S[1:3]用来存储3座塔的状态。所有的碟子从1（最小碟子）到n（最大碟子）编号，因此每个堆栈的类型均为int。如果没有足够的空间来创建三个堆栈，堆栈构造函数将引发一个类型为NoMem的异常，预处理程序终止执行。如果有足够的空间，预处理程序将调用Hanoi::TowersOfHanoi。在该程序中没有给出由Hanoi::TowersOfHanoi所调用的函数ShowState，原因是该函数的实现取决于输出设备的性质（如计算机屏幕、打印机等）。

程序5-7 使用堆栈求解汉诺塔问题

```
class Hanoi{
   friend void TowersOfHanoi(int);
   public:
      void TowersOfHanoi(int n, int x, int y, int z);
  private:
      Stack<int> *S[4];
};
void Hanoi::TowersOfHanoi(int n, int x, int y, int z)
{// 把 n 个碟子从塔x 移动到塔 y，可借助于塔z
  int d; // 碟子编号
  if (n > 0) {
     TowersOfHanoi(n-l, x, z, y);
     S[x]->Delete(d); //从x中移走一个碟子
     S[y]->Add(d); //把这个碟子放到y 上
     ShowState();
     TowersOfHanoi(n-l, z, y, x);}
}
void TowersOfHanoi(int n)
{// Hanoi::towersOfHanoi的预处理程序
  Hanoi X；
  X.S[1] = new Stack<int> (n);
  X.S[2] = new Stack<int> (n);
  X.S[3] = new Stack<int> (n);

  for (int d = n; d > 0; d--) // 初始化
     X.S[1]->Add(d); // 把碟子 d 放到塔 1上

     //把塔 1上的n个碟子移动到塔 3上，借助于塔2 的帮助
     X.TowersOfHanoi(n, 1, 2, 3);
}
```

5.5.3 火车车厢重排

一列货运列车共有n节车厢，每节车厢将停放在不同的车站。假定n个车站的编号分别为1~n，货运列车按照第n站至第1站的次序经过这些车站。车厢的编号与它们的目的地相同。为了便于从列车上卸掉相应的车厢，必须重新排列车厢，使各车厢从前至后按编号1到n的次序排列。当所有的车厢都按照这种次序排列时，在每个车站只需卸掉最后一节车厢即可。我们在一

个转轨站里完成车厢的重排工作，在转轨站中有一个入轨、一个出轨和 k个缓冲铁轨（位于入轨和出轨之间）。图5-5a 给出了一个转轨站，其中有 k=3个缓冲铁轨$H1$，$H2$和$H3$。开始时，n节车厢的货车从入轨处进入转轨站，转轨结束时各车厢从右到左按照编号 1至编号n的次序离开转轨站（通过出轨处）。在图5-5a 中，n=9，车厢从后至前的初始次序为5，8，1，7，4，2，9，6，3。图5-5b 给出了按所要求的次序重新排列后的结果。

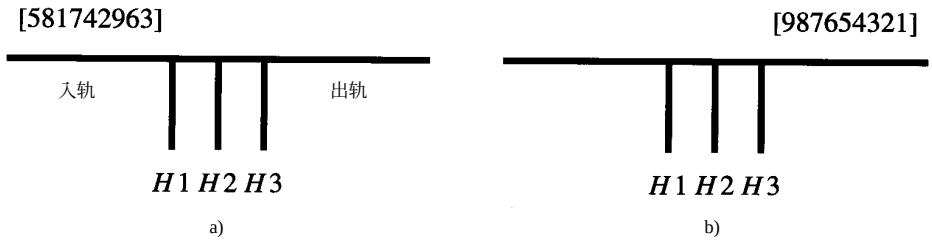


图5-5 具有三个缓冲铁轨的转轨站

a) 开始 b) 结束

为了重排车厢，需从前至后依次检查入轨上的所有车厢。如果正在检查的车厢就是下一个满足排列要求的车厢，可以直接把它放到出轨上去。如果不是，则把它移动到缓冲铁轨上，直到按输出次序要求轮到它时才将它放到出轨上。缓冲铁轨是按照 LIFO的方式使用的，因为车厢的进和出都是在缓冲铁轨的顶部进行的。在重排车厢过程中，仅允许以下移动：

- 车厢可以从入轨的前部（即右端）移动到一个缓冲铁轨的顶部或出轨的左端。
- 车厢可以从一个缓冲铁轨的顶部移动到出轨的左端。

考察图5-5a。3号车厢在入轨的前部，但由于它必须位于 1号和2号车厢的后面，因此不能立即输出3号车厢，可把3号车厢送入缓冲铁轨$H1$。下一节车厢是6号车厢，也必须送入缓冲铁轨。如果把6号铁轨送入$H1$，那么重排过程将无法完成，因为 3号车厢位于6号车厢的后面，而按照重排的要求，3号车厢必须在6号车厢之前输出。因此可把 6号车厢送入$H2$。下一节车厢是9号车厢，被送入$H3$，因为如果把它送入 $H1$或$H2$，重排过程也将无法完成。*请注意：当缓冲铁轨上的车厢编号不是按照从顶到底的递增次序排列时，重排任务将无法完成。*至此，缓冲铁轨的当前状态如图5-6a 所示。

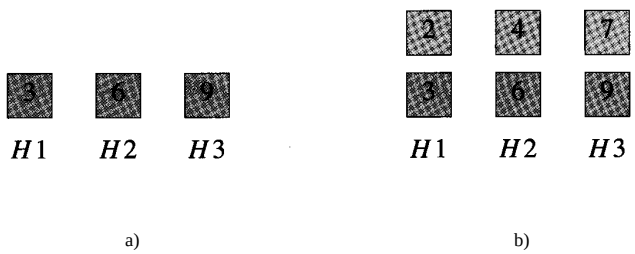


图5-6 缓冲铁轨中间状态

接下来处理2号车厢，它可以被送入任一个缓冲铁轨，因为它能满足缓冲铁轨上车厢编号必须递增排列的要求，不过，应优先把2号车厢送入$H1$，因为如果把它送入$H3$，将没有空间来移动7号车厢和8号车厢。如果把2号车厢送入$H2$，那么接下来的4号车厢必须被送入$H3$，这样将无法移动后面的5号、7号和8号车厢。新的车厢u应送入这样的缓冲铁轨：其顶部的车厢编号

v 满足v>u，且v 是所有满足这种条件的缓冲铁轨顶部车厢编号中最小的一个编号。只有这样才能使后续的车厢重排所受到的限制最小。我们将使用这条分配规则（ assignment rule）来选择缓冲铁轨。

接下来处理4号车厢时，三个缓冲铁轨顶部的车厢分别是2号、6号和9号车厢。根据分配规则，4号车厢应送入*H*2。这之后，7号车厢被送入*H*3。图5-6b 给出了当前的状态。接下来，1号车厢被送至出轨，这时，可以把*H*1中的2号车厢送至出轨。之后，从*H*1输出3号车厢，从*H*2输出4号车厢。至此，没有可以立即输出的车厢了。

接下来的8号车厢被送入*H*1，然后5号车厢从入轨处直接输出到出轨处。这之后，从 *H*2输出6号车厢，从*H*3输出7号车厢，从*H*1输出8号车厢，最后从*H*3输出9号车厢。

对于图5-5a 的初始排列次序，在进行车厢重排时，只需三个缓冲铁轨就够了，而对于其他的初始次序，可能需要更多的缓冲铁轨。例如，若初始排列次序为 1, *n*, *n*-1, ..., 2，则需要*n*-1个缓冲铁轨。

为了实现上述思想，用k 个链表形式的堆栈来描述k 个缓冲铁轨。之所以采用链表形式的堆栈而不是公式化形式的堆栈，原因在于前者仅需要 n-1个元素。函数Railroad（见程序5-8）用于确定重排n 个车厢，它最多可使用k 个缓冲铁轨并假定车厢的初始次序为p[1:n]。如果不能成功地重排， Railroad返回false，否则返回true。如果由于内存不足而使函数失败，则引发一个异常NoMem。

函数Railroad在开始时创建一个指向堆栈的数组H，H[i]代表缓冲铁轨i，1≤i≤k。NowOut是下一个欲输出至出轨的车厢号。 minH是各缓冲铁轨中最小的车厢号， minS是minH号车厢所在的缓冲铁轨。

在for循环的第i 次循环中，首先从入轨处取车厢p[i]，若p[i]=NowOut，则将其直接送至出轨，并将NowOut的值增1，这时，有可能会从缓冲铁轨中输出若干节车厢（通过 while循环把它们送至出轨处）。如果p[i]不能直接输出，则没有车厢可以被输出，按照前述的铁轨分配规则把p[i]送入相应的缓冲铁轨之中。

程序5-8 火车车厢重排程序

```
bool Railroad(int p[], int n, int k)
{// k个缓冲铁轨，车厢初始排序为p[1:n]
 // 如果重排成功，返回true，否则返回false
 // 如果内存不足，则引发异常 NoMem 。
  //创建与缓冲铁轨对应的堆栈
  LinkedStack<int> *H;
  H = new LinkedStack<int> [k + 1];
  int NowOut = 1; //下一次要输出的车厢
  int minH = n+l;  //缓冲铁轨中编号最小的车厢
  int minS;   //minH号车厢对应的缓冲铁轨
  //车厢重排
  for (int i = 1; i <= n; i++)
    if (p[i] == NowOut) {// 直接输出t
      cout << "Move car " << p[i] << " from input to output" << endl;
      NowOut++;
    //从缓冲铁轨中输出
    while (minH == NowOut) {
      Output(minH, minS, H, k, n);
```

```
        NowOut++;
        }
      }
    else {// 将 p[i] 送入某个缓冲铁轨
      if (!Hold(p[i], minH, minS, H, k, n))
      return false;}
    return true;
  }
```

程序5-9和5-10分别给出了Railroad中所使用的函数Output和Hold。Output用于把一节车厢从缓冲铁轨送至出轨处，它同时将修改minS和minH。函数Hold根据车厢分配规则把车厢c送入某个缓冲铁轨，必要时，它也需要修改minS和minH。

程序5-9 程序5-8中所使用的Output 函数

```
void Output(int& minH, int& minS, LinkedStack<int> H[ ], int k, int n)
{//把车厢从缓冲铁轨送至出轨处，同时修改 minS和minH
  int c; // 车厢索引
  //  从堆栈minS中删除编号最小的车厢minH
  H[minS].Delete(c);
  cout << "Move car " << minH << " from holding track " << minS << " to output" << endl;
  // 通过检查所有的栈顶，搜索新的 minH和minS
  minH = n + 2;
  for (int i = 1; i <= k; i++)
    if (!H[i].IsEmpty() && (c = H[i].Top()) < minH) {
      minH = c;
      minS = i;}
}
```

程序5-10 程序5-8中所使用的Hold函数

```
bool Hold(int c, int& minH, int &minS, LinkedStack<int> H[], int k, int n)
(// 在一个缓冲铁轨中放入车厢 c
 // 如果没有可用的缓冲铁轨，则返回 false
 // 如果空间不足，则引发异常 NoMem
 // 否则返回true
  // 为车厢c寻找最优的缓冲铁轨
  // 初始化
  int BestTrack = 0, // 目前最优的铁轨
  BestTop = n + 1, // 最优铁轨上的头辆车厢
  x;// 车厢索引
  //扫描缓冲铁轨
  for (int i = 1; i <= k; i++)
    if (!H[i].IsEmpty()) {// 铁轨 i 不空
      x = H[i].Top();
      if (c < x && x < BestTop) {
        //铁轨 i 顶部的车厢编号最小
        BestTop = x;
        BestTrack = i;}
      }
```

```
    else // 铁轨 i 为空
      if (!BestTrack) BestTrack = i;
  if (!BestTrack) return false; //没有可用的铁轨
  //把车厢c 送入缓冲铁轨
  H[BestTrack].Add(c);
  cout << "Move car " << c << " from input " "to holding track " << BestTrack << endl;
  //必要时修改 minH 和 minS
  if (c < minH) {minH = c; minS = BestTrack;}
  return true;
}
```

为了计算程序5-8的时间复杂性，我们首先注意到Output和Hold的复杂性均为Θ(k)。由于在Railroad的while循环中最多可以输出n-1节车厢，且在else语句中最多有n-1节车厢被送入缓冲铁轨，因此，由函数Output和Hold所消耗的总时间为O(kn)。Railroad中for循环的其余部分需耗时Θ(n)。所以，程序5-8总的时间复杂性为O(kn)。若使用一个平衡折半搜索树（如AVL树）来存储缓冲铁轨顶部的车厢编号（见第 11章），则程序的复杂性可以降至O(nlogk)。在使用平衡折半搜索树时，可以重写函数Output和Hold，以使其具有复杂性O(logk)。对于本应用，仅当k很大时，才推荐使用平衡折半搜索树。

5.5.4 开关盒布线

开关盒布线问题是这样的：给定一个矩形布线区域，其外围有若干针脚。两个针脚之间通过布设一条金属线路而实现互连。这条线路被称为电线，被限制在矩形区域内。如果两条电线发生交叉，则会发生电流短路。所以，不允许电线间的交叉。每对互连的针脚被称为网组。我们的目标是要确定对于给定的网组，能否合理地布设电线以使其不发生交叉。图 5-7a 给出了一个布线的例子，其中有八个针脚和四个网组。四个网组分别是 (1,4,)，(2,3)，(5,6)和(7,8)。图5-7b 给出的布线方案有交叉现象发生（(1,4) 和(2,3) 之间），而图5-7c 则没有交叉现象发生。由于四个网组可以通过合理安排而不发生交叉，因此可称其为可布线开关盒（routable switch box）。（在具体实现时，还需要在两个相邻的电线间留出一定的间隔，为使问题简化，本应用中忽略这个额外的要求）。我们要解决的问题是，给定一个开关盒布线实例，确定它是不是一个可布线的。

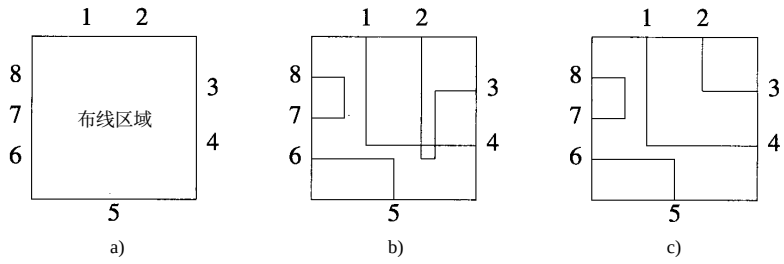


图5-7 开关盒布线示例

图5-7b 和5-7c 中的电线都是由平行于x 轴和y 轴的垂直线段构成的，当然也可以使用不与x 轴和y 轴平行的线段。

为了解决开关盒布线问题，我们注意到，当两个针脚互连时，其电线把布线区分成两个分

区。例如，当(1,4)互连时，就得到了两个分区，一个分区包含针脚2和3，另一个分区包含针脚5~8。现在如果有一个网组，其两个针脚分别位于这两个不同的分区，那么这个网组是不可以布线的，因而整个电路也是不可布线的。如果没有这样的网组，则可以继续判断每个独立的分区是不是可布线的。为此，可以从一个分区中取出一个网组，利用该网组把这个分区又分成两个子分区，如果任一个网组的两个针脚都分布在同一个子分区之中（即不会出现两个针脚分别位于两个子分区的情形），那么这个分区就是可布线的。

为了实现上述策略，可以按顺时针或反时针方向沿着开关盒的外围进行遍历，可从任意一个针脚开始。例如，如果按顺时针方向从针脚 1开始遍历图5-7a 中的针脚，那么将依次检查针脚1, 2, ..., 8。针脚1和4属于同一个网组，那么在针脚1至针脚4之间出现的所有针脚构成了第一个分区，而在针脚4至针脚1之间出现的所有针脚构成了第二个分区。把针脚 1放入堆栈，然后继续处理，直至遇到针脚 4。这个过程使我们仅在处理完一个分区之后才能进入下一个分区。下一个针脚是针脚2，它与针脚3同属一个网组，它们又把当前分区分成两个子分区。与前面的做法一样，把针脚2放入堆栈，然后继续处理直至遇到针脚3。由于针脚3与针脚2属同一个网组，而针脚2正处在栈顶，这表明已经处理完一个子分区，因此可将针脚 2从栈顶删除。接下来将遇到针脚4，由于与之互连的针脚1正处在栈顶，因此当前的分区已经处理完毕，可从栈顶删除针脚1。按照这种方法继续进行下去，直至检查完八个针脚，堆栈变空，所创建的分区都已处理完毕为止。

那么，对于不可布线的开关盒将会出现什么样的情况呢？假定图 5-7a 中的网组是：(1,5)，(2,3)，(4,7)和(6,8)。初始时，针脚1和2被放入堆栈。在检查针脚3时，将针脚2从栈顶删除。接下来针脚4被放入堆栈，因为针脚 4与栈顶的针脚不能构成一个网组。在检查针脚 5时，它也被放入堆栈。尽管已经遇到了针脚1和针脚5，但还不能结束由这两个针脚所定义的第一个分区的处理过程，因为针脚 4的布线将不得不跨越这个分区的边界。因此，当完成对所有针脚的检查时，堆栈不会变空。

程序5-11给出了按上述策略实现的C++程序。它要求对每个网组进行编号，并且每个针脚也得有一个对应的网组编号。所以，对于图 5-7c 中的例子，输入数组net=[1,2,2,1,3,3,4,4]。程序5-11的复杂性为$\Theta(n)$，其中n 为针脚的数目。

程序5-11 开关盒布线

```
bool CheckBox(int net[ ], int n)
{// 确定开关盒是否可布线
  Stack<int> *s = new Stack<int> (n);
  //顺时针扫描各网组
  for (int i = 0; i < n; i++) {
    //检查net[i]
    if (!s->IsEmpty() ) {
      if (net[i] == net[s->Top()] ) {
        // net[i] 可布线，从堆栈中删除
        int x;
        s->Delete(x);) }
      else s->Add(i); }
    else s->Add(i);
    }
  // 是否有不可布线的网组？
  if (s->IsEmpty()) {
```

```
    delete s;
    cout << "Switch box is routable" << endl;
    return true;}
  delete s;
  cout << "Switch box is not routable" << endl;
  return false;
}
```

5.5.5 离线等价类问题

离线等价类问题的定义见3.8.3节。这个问题的输入是元素数目n、关系数目r 以及r 对关系，问题的目标是把n个元素分配至相应的等价类。程序 5-12给出了解决离线等价类问题的 C++程序。在程序5-12a 中，输入为n, r 和r 对关系，对于每个元素都建立了一个相应的链表。与元素 i 对应的链表chain[i]中包含所有这样的元素j：(i, j)或(j, i)是所输入的关系。程序5-12b 用于输出等价类，其中使用了一个数组out，当且仅当i已经被作为某个等价类的成员输出时，out[i]=true。堆栈stack用于定位一个等价类中所有的元素。在这个等价类中含有当前等价类中所有的元素。

为了找到下一个等价类中的第一个成员，可以扫描数组 out以寻找尚未被输出的元素。如果没有这样的元素，则表明不再有新的等价类。如果找到一个这样的元素，则开始搜索下一个等价类。把这个元素放入堆栈，然后依次对堆栈中的元素进行检查，看看这些元素是否与该元素等价。具体的检查方法是：从堆栈中删除一个元素 m，然后检查chain[m]中的所有元素。当堆栈为空时，在当前等价类中将不会存在这样的成员 m：(m,p)是所输入的关系，而p尚未被输出（因为p一定位于chain[m]之中，所以从堆栈中删除m时，p肯定已经被输出）。因此，在每次do循环过程中所输出的元素都构成了一个等价类。

为了分析等价类程序的复杂性，可以注意到，由于每次链表的插入操作都是在链表的首部进行的，因此每次插入操作需耗时 $\Theta(1)$。程序5-12a 需耗时 $\Theta(n+r)$（用于输入和链表初始化）。对于程序5-12b，可以发现，每个元素都仅被输出一次，每个元素都只进堆栈一次，并被从堆栈中删除一次，因此，执行堆栈添加和删除操作所需要的总时间为 $\Theta(n)$。当从堆栈中删除一个元素时，该元素对应链表中的所有元素也将被删除。由于每次删除都是从链表首部进行的，因此每次删除需耗时 $\Theta(1)$。在程序5-12a 中输入结束时，所有n 个链表中的元素总数为2r，而在程序5-12b 中所有2r个链表元素都将被删除，因此删除链表元素共需耗时 $\Theta(r)$。这样，程序5-12总的时间复杂性为 $\Theta(n+r)$。

程序5-12a 离线等价类程序 (一)

```
void main(void)
{// 离线等价类问题
  int n, r;
  //输入n 和r
  cout << "Enter number of elements" << endl;
  cin >> n;
  if (n < 2) {cerr << "Too few elements" << endl; exit(l);}
  cout << "Enter number of relations" << endl;
  cin >> r;
  if (r < 1) {cerr << "Too few relations" << endl; exit(l);}
  //创建一个指向n个链表的数组
```

```
Chain<int> *chain;
try {chain = new Chain<int> [n+1];}
catch (NoMem) {cerr << "Out of memory" << endl; exit(1);}
//输入 r个关系，并存入链表
for (int i = 1; i <= r; i++) {
  cout << "Enter next relation/pair" << endl;
  int a, b;
  cin >> a >> b;
  chain[a].Insert(0,b);
  chain[b].Insert(0,a);
}
```

程序5-12b 离线等价类程序（二）

```
//对欲输出的等价类进行初始化
LinkedStack<int> stack;
bool *out;
try {out = new bool [n+1];}
catch (NoMem) {cerr << "Out of memory" << endl; exit(l);}
for (int i = 1; i <= n; i++)
  out[i] = false;
//输出等价类
for (int i = 1; i <= n; i++)
  if (!out[i]) {//开始一个新的等价类
    cout << "Next class is: " << i << ' ' ;
    out[i] = true;
    stack.Add(i);
    //从堆栈中取其余的元素
    while (!stack.IsEmpty()) {
      int *q, j;
      stack.Delete(j);
      //链表chain[j]中的元素都在同一个等价类中，使用遍历器 c来取这些元素
      ChainIterator<int> c;
      q = c.Initialize(chain[j1);
      while (q){
        if (!out[*q]) {
          cout << "q << '  ';
          out[*q] = true;
          stack.Add(*q);}
        q = c.Next();
      }
    }
  cout << endl;
  }
cout << endl << "End of class list" << endl;
}
```

注意，在程序5-12中并未删除为chain和out所分配的空间。由于在程序结束时，这些空间会被自动释放，因此，没有必要特意去删除它们。如果把程序 5-12改编成一个函数，那么必须

增加相应的语句来删除这两个数组所占用的空间，以便这些空间能为程序的其他部分所用。

5.5.6 迷宫老鼠

1. 问题描述

迷宫（maze）是一个矩形区域，它有一个入口和一个出口。在迷宫的内部包含不能穿越的墙或障碍。在图5-8所示的迷宫中，障碍物沿着行和列放置，它们与迷宫的矩形边界平行。迷宫的入口在左上角，出口在右下角。

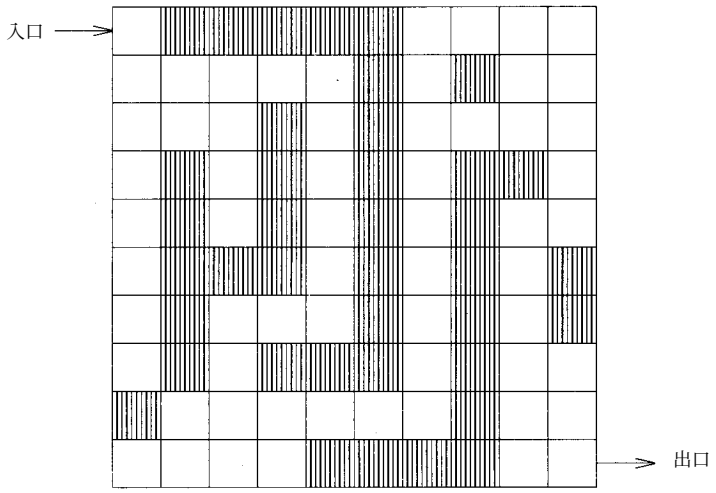


图5-8 迷宫

假定用$n \times m$的矩阵来描述迷宫，位置(1,1)表示入口，(n,m) 表示出口，n和m分别代表迷宫的行数和列数。迷宫中的每个位置都可用其行号和列号来指定。在矩阵中，当且仅当在位置(i,j)处有一个障碍时其值为1，否则其值为0。图5-9给出了图5-8中迷宫对应的矩阵描述。迷宫老鼠（rat in a maze）问题要求寻找一条从入口到出口的路径。路径是由一组位置构成的，每个位置上都没有障碍，且每个位置（第一个除外）都是前一个位置的东、南、西或北的邻居（如图5-10所示）。

```
0 1 1 1 1 1 0 0 0 0
0 0 0 0 0 1 0 1 0 0
0 0 0 1 0 1 0 0 0 0
0 1 0 1 0 1 0 1 1 0
0 1 0 1 0 1 0 1 0 0
0 1 1 1 0 1 0 1 0 1
0 1 0 0 0 1 0 1 0 1
0 1 0 1 1 1 0 1 0 0
1 0 0 0 0 0 0 0 1 0 0
0 0 0 0 1 1 1 1 1 0 0
```

图5-9 图5-8中迷宫的矩阵描述

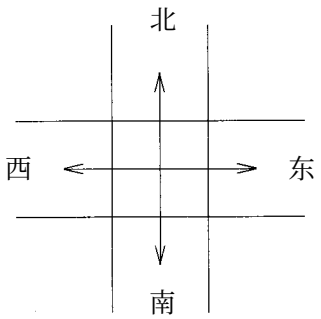


图5-10 迷宫任意位置处的4种移动方向

下面将要编写程序来解决迷宫老鼠问题。假定程序中所使用的迷宫是正方形的（即行数等

于列数），且迷宫足够小，以便能在目标计算机的内存中描述整个迷宫。所编写的程序应是一个独立的产品，想要在迷宫中寻找路径的人可以直接使用它。

2. 设计

可以采用自顶向下的模块化方法来设计这个程序。不难看出，这个程序可以划分为三个部分：输入迷宫、寻找路径和路径输出。每个部分可以专门用一个程序模块来实现。第四个模块用来显示欢迎信息、软件名称及作者信息，这个模块与其他三个模块之间没有任何直接关系，主要用于增强程序界面的友好性。

寻找路径的模块不会直接与用户打交道，因此不必提供帮助机制，也不需要由菜单驱动。其他三个模块都会与用户进行交互，因此应多花一些精力来设计它们的用户接口。好的用户接口能使用户喜欢使用你的程序。

首先来设计欢迎模块。我们希望显示如下信息：

Welcome To
RAT IN A MAZE
© Joe Bloe, 1998

如果觉得按这种形式显示信息显得有些单调，可以引入其他手段以得到满意的效果。比如，可以用多种颜色来显示这些信息，可以使每行文字的大小不一样（甚至每个字符的大小也不一样），也可以让每个字符按一定的时间间隔依次出现。除此以外，还可以考虑使用声音效果。信息显示的时间也需要确定，显示的时间应足够长，以便用户能够读完它，但也不能太长（以至于用户打哈欠）。因此，欢迎信息（乃至整个用户接口）的设计需要一定的艺术技巧。

输入模块应告诉用户需输入一个由0和1组成的矩阵。为此，需要确定是按逐行方式还是逐列的方式来输入矩阵。由于逐行方式是比较自然的方式，因此可要求用户按逐行方式输入矩阵。在输入矩阵数据之前，用户必须首先给出矩阵的行数（它同时也是矩阵的列数）。可按图5-11的形式来提示用户输入所需要的数据。

```
Please enter the number of rows in the maze.

I work on square mazes only.

Enter -1 to terminate the program.

Press <Enter> when done.
```

图5-11 获取迷宫大小的屏幕显示信息

接下来要求用户逐行输入矩阵数据：

Please Enter Row 1

Press <Enter> when done

每输入完一行数据，都应把它们显示在屏幕上以便于用户验证。为此，有必要提供编辑功能来帮助纠正错误。重新输入整行数据是一种可取的编辑方式（从用户的观点来看）。输入模块需要验证在迷宫的入口或出口处是否有障碍物，如果有障碍物，则不可能存在路径。此外，输入模块应对用户所有可能的输入错误进行校验。在后面的讨论中假定输入模块已经做过这样的验证，并且在入口和出口处没有障碍物。

在设计输入模块的用户接口时还涉及其他一些问题，如颜色和声音的使用；询问信息的显示尺寸；询问信息在屏幕上的显示位置；每输入完一行数据时屏幕是否需要滚动；或者是否需要清除上一行数据并在同一位置显示下一行数据等。

在这里再一次看到，本来看上去很简单的任务（读入一个矩阵）实际上很复杂，因为我们想使它的界面非常友好。

在设计输出模块时所涉及的问题基本上与设计输入模块所考虑的问题相同。

3. 开发计划

在前面设计阶段已经指出需要设计四个程序模块。实际上还需要一个根模块来调用这四个模块，调用的次序是：欢迎模块、输入模块、寻找路径模块和输出模块。

程序的模块结构如图5-12所示。每个模块都可以独立地编写。根模块将被编写成一个 main 函数，欢迎模块、输入模块、寻找路径模块和输出路径模块将分别对应于一个函数。

至此，程序如图5-13所示。

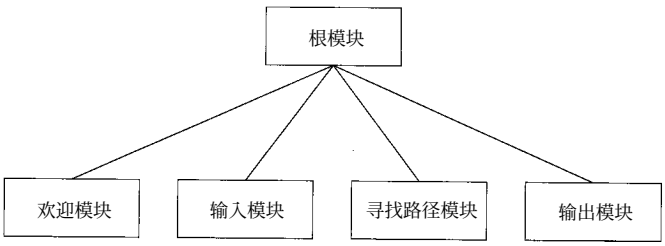


图5-12 迷宫程序的模块结构

```
// 函数模块
// 欢迎模块 Welcome
// 输入模块 InputMaze
// 寻找路径模块 FindPath
// 输出模块 OutputPath
void main(void)
{
  Welcome();
  InputMaze();
  if (FindPath())  OutputPath;
       else cout << "No path" << endl;
     }
```

图5-13 迷宫程序的形式

4. 程序开发

数据结构和算法的问题仅出现在寻找路径模块的开发过程中。因此，这里仅设计寻找路径模块。练习16要求读者设计出其他的模块。在详细设计寻找路径模块的编码之前，可以先给出图5-14所示的C++伪代码。可以很容易判断这段代码的正确性。遗憾的是，不能直接在计算机中使用这种形式的代码，必须把这种伪代码细化成真正的C++代码。

在试图对图5-14的伪代码进行细化之前，首先探讨如何寻找迷宫路径。首先把迷宫的入口作为当前位置。如果当前位置是迷宫出口，那么已经找到了一条路径，搜索工作结束。如果当

前位置不是迷宫出口，则在当前位置上放置障碍物，以便阻止搜索过程又绕回到这个位置。然后检查相邻的位置中是否有空闲的（即没有障碍物），如果有，就移动到这个新的相邻位置上，然后从这个位置开始搜索通往出口的路径。如果不成功，选择另一个相邻的空闲位置，并从它开始搜索通往出口的路径。为了方便移动，在进入新的相邻位置之前，把当前位置保存在一个堆栈中。如果所有相邻的空闲位置都已经被探索过，并且未能找到路径，则表明在迷宫中不存在从入口到出口的路径。

```
bool FindPath()
{
 在迷宫中寻找一条通往出口的路径;
 if (找到一条路径) return true;
 else return false;
   }
```

图5-14 FindPath的第一个版本

使用上述策略来考察图5-8的迷宫。首先把位置(1,1)放入堆栈，并从它开始进行搜索。由于位置(1,1)只有一个空闲的邻居(2,1)，所以接下来将移动到位置(2,1)，并在位置(1,1)上放置障碍物，以阻止稍后的搜索再次经过这个位置。从位置(2,1)可以移动到(3,1)或(2,2)。假定移动到位置(3,1)。在移动之前，先在位置(2,1)上放置障碍物并将其放入堆栈。从位置(3,1)可以移动到(4,1)或(3,2)。假定移动到位置(4,1)，则在位置(3,1)上放置障碍物并将其放入堆栈。从位置(4,1)开始可以依次移动到(5,1)、(6,1)、(7,1)和(8,1)。到了位置(8,1)以后将无路可走。此时堆栈中包含的路径从(1,1)至(8,1)。为了探索其他的路径，从堆栈中删除位置(8,1)，然后回退至位置(7,1)，由于位置(7,1)也没有新的、空闲的相邻位置，因此从堆栈中删除位置(7,1)并回退至位置(6,1)。按照这种方式，一直要回退到位置(3,1)，然后才可以继续移动（即移动到位置（3,2））。注意在堆栈中始终包含从入口到当前位置的路径。如果最终到达了出口，那么堆栈中的路径就是所需要的路径。

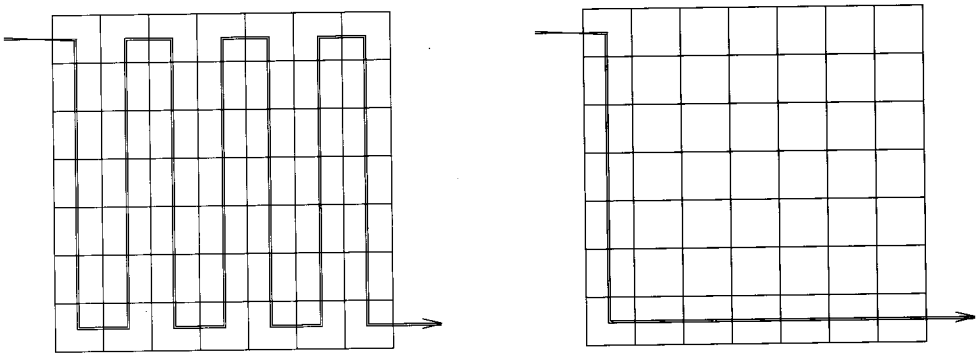
为了细化图5-14，需要对迷宫（一个0和1的矩阵）、迷宫的每个位置以及堆栈进行描述。首先考虑迷宫的描述。迷宫一般被描述成一个int类型的二维数组maze。（由于每个迷宫位置仅有0或1两种取值，因此可以把16个迷宫位置压缩成一个int类型的变量，假定每个变量的长度是两个字节。这种压缩可以使迷宫所需的空间减小16倍。不过，由于访问每个迷宫位置的难度增加，因此程序的运行时间将随之增加）。迷宫矩阵中的位置（i, j）对应于数组maze的位置[i, j]。

对于迷宫内部的位置（非边界位置），有四种可能的移动方向：右、下、左和上。对于迷宫的边界位置，只有两种或三种可能的移动。为了避免在处理内部位置和边界位置时存在差别，可以在迷宫的周围增加一圈障碍物。对于一个m×n的迷宫，这一圈障碍物将占据数组maze的第0行，第m+1行，第0列和第m+1列（如图5-15所示）。

现在迷宫中所有的位置都处在所添加的障碍物的边界以内，因此，对于原迷宫中的每个位置，都可以有四种可能的移动方向。通过在迷宫周围添加边界，可以使程序不必去处理边界条件，这可以大大简化代码的设计。这种简化是以稍稍增加数组maze的空间

1	1	1	1	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	0	0	0	0	1
1	0	0	0	0	0	1	0	1	0	0	1
1	0	0	0	1	0	1	0	0	0	0	1
1	0	1	0	1	0	1	0	1	1	0	1
1	0	1	0	1	0	1	0	1	0	0	1
1	0	1	1	1	0	1	0	1	0	1	1
1	0	1	0	0	0	1	0	1	0	1	1
1	0	1	0	1	1	1	0	1	0	0	1
1	1	0	0	0	0	0	0	1	0	0	1
1	0	0	0	0	1	1	1	1	0	0	1
1	1	1	1	1	1	1	1	1	1	1	1

图5-15 为图5-8的迷宫增加一圈障碍物



a) b)

图5-16 没有障碍物的迷宫中的路径

a) 一条长路径 b) 一条短路径

需求为代价的。

可以用行号和列号来指定每个迷宫位置，行号和列号被分别称之为迷宫位置的行坐标和列坐标。可以定义一个相应的类Position来表示迷宫位置，它有两个私有成员row和col。可以使用类型为Position的对象来跟踪迷宫的位置。为了保存从入口到当前位置的路径，可以采用以下基于公式化描述的堆栈：

```
Stack<Position> path(MaxPathLength);
```

其中MaxPathLength是指最大可能的路径长度（从入口到迷宫中任一位置）。由于在一条路径中所出现的位置各不相同，因此对于一个没有障碍物的m×m迷宫，最长的路径可包含m^2个位置（如图5-16a所示）。由于路径中的最后一个位置不必存储到堆栈中，因此有MaxPathLength= m^2-1。注意，在一个没有障碍物的迷宫中，任意两点之间总是存在一条少于2m个位置的路径。不过，这里我们并没有保证搜索程序能找到最短路径。

现在可以来细化图5-14，细化的结果见图5-17，它已经接近了C++程序。接下来需要着手解决这样的问题：对于位置here，下一步将移动到它的哪一个相邻位置。如果按一种固定的方式来

```
bool FindPath()
{// 寻找从位置(1,1)到出口(m,m)的路径
 增加一圈障碍物;

// 对跟踪当前位置的变量进行初始化
Position here;
here.row = 1;
here.col = 1;

maze [1] [1] = 1; // 阻止返回入口

//寻找通往出口的路径
while (不是出口) do {
 选择一个相邻位置;
 if (存在这样一个相邻位置) {
   把当前位置 here 放入堆栈path;
   // 移动到相邻位置，并在当前位置放上障碍物
   here = neighbor;
   maze[here.row] [here.col] = 1;}
 else {
   // 不能继续移动，需回溯
   if (堆栈path为空) return false;
   回溯到path栈顶中的位置 here; }
}
return true;
}
```

图5-17 图5-14的细化版本

选择可行的位置，将可以使问题得到简化。例如，可以首先尝试向右移动，然后是向下，向左，最后是向上。一旦做出了选择，就需要知道下一个位置的坐标。可以通过保留一个如图 5-18所示的偏移量表来计算这些坐标。可以分别把向右、向下、向左和向上移动编号为 0，1，2和3。在图5-18中, offset[i].row和offset[i].col分别给出了从当前位置沿方向i移动到下一个相邻位置时，row和col坐标的增量。例如，如果当前处于位置 (3,4)，则其右边相邻位置的行坐标为3+offset[0].row=3，列坐标为4+offset[0].col=5。

把上述细化工作并入图 5-17的代码之中，就得到了程序 5-13中的C++代码。5-13中的代码假定maze、m(迷宫的大小)和path都是按如下方式定义的全局变量：

```
int **maze, m;
Stack<Position> *path;
```

并且FindPath是Position的一个友元。变量maze和m由函数InputMaze负责初始化。

移动	方向	offset[move].row	offset[mov].col
0	向右	0	1
1	向下	1	0
2	向左	0	-1
3	向上	-1	0

图5-18 偏移量表

程序5-13 搜索迷宫路径的代码

```
bool FindPath（）
{// 寻找从位置(1,1)到出口(m,m)的路径
 //如果成功则返回true ， 否则返回false
 // 如果内存不足则引发异常NoMem
   path = new Stack<Position>(m * m - 1);
   //对偏移量进行初始化
   Position offset[4];
   offset[0].row = 0;  offset[0].col = 1; //向右
   offset[l].row = 1;  offset[l].col = 0; // 向下
   offset[2].row = 0;  offset[2].col = -1; //向左
   offset[3].row = -1; offset[3].col = 0; //向上
   //在迷宫周围增加一圈障碍物
   for (int i = 0; i <= m+l; i++) {
      maze [0] [i] = maze[m+l] [i] = 1; //底和顶
      maze [i] [0] = maze[i] [m+l] = 1; // 左和右
      }
   Position here;
   here.row = 1;
   here.col = 1;
   maze [i] [i] = 1; //  阻止返回入口
   int option = 0;
   int LastOption = 3;
   //寻找一条路径
   while (here.row != m || here.col != m) {// 不是出口
     //寻找并移动到一个相邻位置
```

```
    int r, c;
    while (option <= LastOption) {
      r = here.row + offset[option].row;
      c = here.col + offset[option].col;
      if (maze[r][c] == 0) break;
      option++; //下一个选择
    }
    // 找到一个相邻位置了吗?
    if (option <= LastOption) {// 移动到maze[r] [c]
      path->Add(here);
      here.row = r; here.col = c;
      //设置障碍物以阻止再次访问
      maze [r] [c] = 1;
      option = 0;
      }
   else {//没有可用的相邻位置，回溯
     if (path->IsEmpty()) return false;
     Position next;
     path->Delete(next);
     if (next.row == here,row)
       option = 2 + next.col - here.col;
     else option = 3 + next.row - here.row;
     here = next;
     }
   }
   return true;//到达迷宫出口
}
```

FindPath首先创建一个足够大的堆栈*path，然后对偏移量数组进行初始化，并在迷宫周围增加一圈障碍物。在while循环中，从当前位置here出发，按下列次序来选择下一个移动位置：向右、向下、向左和向上。如果能够移动到下一个位置，则将当前位置放入堆栈 path，并移动到下一个位置。如果找不到下一个可以移动的位置，则退回到前一个位置。如果无法回退一个位置（即堆栈为空），则表明不存在通往出口的路径。当回退至堆栈的顶部位置 (next)时，可以重新选择另一个可能的相邻位置，这可以利用 next和here来推算。注意here是next的一个邻居。对下一个移动位置的选择可用以下代码来实现：

```
  if ( next.row == here.row)
Option= 2 + next.col - here.col;
  else option = 3 + next.row - here.row;
```

下面分析程序的时间复杂性。可以注意到，在最坏情况下，可能要遍历每一个空闲的位置，而每个位置进入堆栈的机会最多有四次。（每次从一个位置向下一个位置移动时，该位置都要进入堆栈，而对于每个位置，可以有四种可能的移动选择）。因而每个位置从堆栈中被删除的机会也最多有四次。对于每个位置，需花费$\Theta(1)$的时间来检查它的相邻位置。因此，程序的时间复杂性应为O(*unblocked*)，其中*unblocked*是迷宫中的空闲位置数目。O(unblocked)=O(m^2)。

练习

7. 利用归纳法（对碟子的数目进行归纳）证明程序5-6的正确性。

8. 假定汉诺塔中的碟子按1~n 编号，最小的碟子为1号碟子。修改程序5-6，让它同时输出被移动碟子的编号。这种修改只需简单地改动语句 cout 即可，要求不得对其他地方进行修改。

9. 编写程序5-7中的ShowState 函数，假定输出设备为计算机屏幕。注意考虑时间延迟，以便显示信息不至于变化太快。用不同的颜色显示每个碟子。

10. 当输出设备为计算机屏幕时，能否在每次移动之后只显示三座塔的状态而不必在内存中明确地存储塔的状态？试解释。

11. 假定在重排火车车厢问题中用k个基于公式化描述的堆栈来表示k 个缓冲铁轨。请问每个堆栈应有多大？

*12. 1) 采用k个缓冲铁轨来进行车厢重排，程序5-8总能成功吗？

2) 车厢被移动的总次数为：n+ (车厢被移动到缓冲铁轨的次数)。假定对于给定的初始次序，使用程序5-8和k 个缓冲铁轨能够完成车厢重排。请给出程序 5-8最少的移动次数，并证明该结论。

13. 假定每个缓冲铁轨i 中最多只能容纳s_i 节车厢，其中$1 \le i \le k$，请重新编写一个车厢重排程序。

14. 在开关盒布线问题中，注意到当一个网组的两个针脚都进入堆栈的时候，处理过程将结束。假定开关盒布线问题的输入是一组网组，每个网组对应于两个针脚。编写一个 C++程序来输入每个网组，并判断开关盒是否可以布线。要求使用一个新的函数 CheckBox来进行这种判断。如果向堆栈中添加一个针脚时，堆栈中出现了两个属于同一网组的针脚，则函数 CheckBox可以立即终止。程序的时间复杂性应为$\Theta(n)$，其中n为针脚的数目。需要多大的堆栈？

15. 用类Chain和LinkedStack解决离线等价类问题，在生成等价类时，使用了函数Chain::Delete来删除节点，用函数LinkedStack::Add来创建节点。为了避免这两种操作，可以编写自己的代码来对链表和堆栈进行插入和删除操作，要求仅当节点 chain[j]的data域x满足out[x]=0时，才将该节点加入堆栈。也即，堆栈中的每个节点任何时候都只会出现在一个链表中。在执行完关系的输入以及数组 out的创建以后，不必再调用函数 new。 另外由于在输出所有的等价类以后程序终止运行，因此可以不必调用函数 delete。

1) 编写一个解决离线等价类问题的程序，要求不使用 Chain类和LinkedStack类。程序应使用普通的代码（即自己编写的代码）来对链表和堆栈进行添加和删除操作，而且始终不要使用delete函数，并且在数组out创建完以后也不去调用 new函数。程序的复杂性应与元素及关系的数目成正比，试证明之。

2) 使用适当的测试数据比较本代码及程序5-12代码的运行时间。

16. 完成迷宫程序的设计，编写出以下C++程序：

1) 一个漂亮的Welcome函数

2) 一个稳定性好的InputMaze函数。例如，如果没有足够的内存来创建数组 maze，应能检测出来并输出一个错误信息。另外，在输入过程中应给出足够的用户提示。

3) 一个OutputPath函数，用于输出从入口到出口的路径（不是从出口到入口）。

使用一些迷宫例子来测试代码。

17. 重新设计迷宫程序，要求按图5-8 的形式来显示迷宫，并且用不同的颜色标出原始障碍物位置、由算法添加的障碍物位置、空闲位置以及从入口到当前位置的路径。 FindPath函数应在每一次移动时都相应地修改显示状态。为了能让用户看清显示过程，必须使代码减速，做到大约每秒钟移动一次。为此，可在代码中插入等待语句，每次等待一定的时间（比如 1秒）。使用一些迷宫例子来测试代码。

18. 修改函数FindPath，使得从当前位置开始，可尝试向以下方向的相邻位置移动：北、东北、东、东南、西和西北。使用适当的迷宫来测试修改后代码的正确性。

19. 对于堆栈path的最大尺寸，给出一个比m^2-1更理想的上限。

20. 由于堆栈path是动态定义的，因此其大小可以取值为迷宫中空闲位置的数目减 1。按照这种思想对程序5-13进行修改。

21. 修改程序5-13，用链表形式的堆栈来实现path。对于迷宫应用来说，采用链表形式的堆栈与采用公式化描述的堆栈各有哪些优缺点？

22. 在程序5-13的代码中，堆栈应足够大以便满足所有的添加操作。因此，在类 Stack的添加操作中没有必要测试堆栈是否已满。同时，利用 Stack的成员函数按照从入口到出口的次序输出路径也存在不少困难。用一个普通的堆栈来重写程序 5-13的代码，也即使用一个动态定义的一维数组和一个变量top来实现堆栈。最终的路径可按照 path中位置1到位置top的次序输出。使用迷宫的例子来测试代码。

23. 在迷宫中寻找路径的策略实际上是一个递归的策略。从当前位置寻找并移动到它的一个相邻位置，然后确定从这个相邻位置到出口之间是否存在一条路径，如果存在一条路径，则路径搜索过程结束，否则必须继续寻找并移动到另一个相邻位置。采用递归方法重写函数FindPath（见程序5-13）。采用适当的迷宫例子来测试代码的正确性。

5.6 参考及推荐读物

开关盒布线算法选自如下论文：

1) C.Hsu. General River Routing Algorithm. *ACM/IEEE Design Automation Conference,* 578~583, 1983。

2) R.Pinter. River-routing:Methodology and Analysis. *Third Caltech Conference on VLSI,* 1983.3。