第3章 chapter 3

栈和队列

栈和队列广泛应用于计算机软硬件系统中。在编译系统、操作系统等系统软件和各类应用软件中经常需要使用栈和队列完成特定的算法设计。它们的逻辑结构和线性表相同，但它们是一种特殊的线性表。其特殊性在于运算操作受到了一定限制，因此栈和队列又被称为操作受限的线性表。栈按“后进先出”的规则进行操作，队列则按“先进先出”的规则进行操作。

【本章学习要求】

掌握：栈的基本概念，存储结构以及入栈、出栈等基本操作。

掌握：在处理实际问题中如何运用栈特点解决问题。

了解：栈在递归实现过程中的作用。

掌握：队列的基本概念，存储结构和入队、出队等基本操作。

了解：如何运用队列解决实际问题。

3.1 案例导引

什么是栈，什么是队列？可以用一句话描述：如果物品（数据结构里是数据）存放的顺序和取用的顺序一致，是队列，反之，物品存放的顺序和取用顺序相反，是栈。

厨房中碗碟的垒放顺序是自下而上，取用的顺序是自上而下，这是生活中的栈式存取结构。车站的购票通道中，旅客窗口购票的顺序是旅客排队加入购票通道的顺序，先来先购票，顾客按照排队顺序依次进行购票，这是生活中队列式存取结构。

再如，枪械是士兵的必备武器，有两种很有代表性的枪械，手枪和重机枪，这两种枪械通过发射子弹进行射击，一般手枪使用弹匣，配备数发子弹的弹匣，如图 3.1 所示。

图 3.1 弹匣

手枪射击的过程中，会从弹匣中提取子弹，提取子弹的顺序和压入子弹的顺序是恰好相反的。

而重机枪一般采用弹链进行供弹，如图 3.2 所示。

弹链中，子弹按照线性方式顺序依次排列，机枪在射击的过程中，子弹按照排列的顺序依次被射出枪管，这种存、取子弹的方式符合队列的模式。

数据结构课程中，不仅可以用栈和队列处理类似的简单问题；还有一些较复杂问题的求解，同样需要栈和队列这两种特殊的数据结构。一般的方法是，找出问题自身隐含的与某种数据结构的内在联系，再设计算法进行求解，下面举例说明。

【案例 3.1】 迷宫问题。

迷宫问题是一个经典的问题，要求游戏者从迷宫入口开始，找出一条路径到达迷宫的出口，游戏爱好者在一些探险类游戏里会经常遇到设置了复杂路径或者通道的迷宫，掌握走迷宫的技巧是迷宫通关游戏的基本要求，图 3.3 是一个游戏迷宫。

图 3.2 弹链

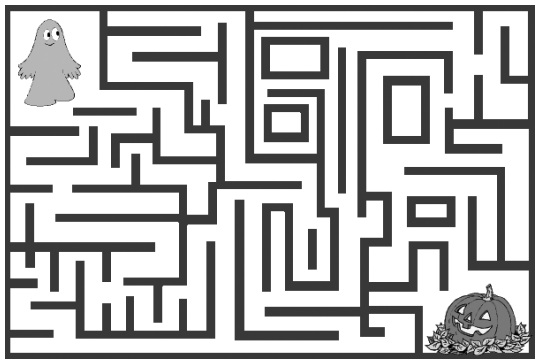

图 3.3 迷宫

如何设计算法求解迷宫问题呢？在迷宫存在路径的前提下，求解迷宫问题的要点有两方面：一是要记住曾经走过的路径或者位置点；二是当遇到走不通的情形时，从当前位置回退到最近一个曾经走过的位置，并重新寻找新的路径去走迷宫。如果从最近的回退位置找不出走出迷宫的路径，从该位置继续回退至上一个位置，继续搜索路径，重复这样的过程直至走出迷宫。这两点综合在一起正好可以利用栈来实现迷宫的路径搜索，因为栈是后进先出的数据结构，最后保存在栈里的位置点，是最新刚走过的位置点，让它最先出栈，正好满足了回退重新搜索路径的需要。

【案例 3.2】 Web 导航。

标准的 Web 浏览器包含前后翻页的功能，使用者在浏览网页的时候，可以根据当时访问的需要，对曾经访问过的页面，进行回退或者前进恢复访问，方便了用户的使用。

浏览器是如何支持这样的功能的实现呢？

一般情况下，栈可以保存曾经走过的路径结点，可以用一个栈 back_Stack 来保存向前浏览网页过程中所访问过的页面，当需要回退时，从栈里取出之前每一步访问过的页面地址，重新让浏览器去解析，就可以实现访问路径的回退功能。

但是，回退过程中，又想再次沿着曾经走过的路径向前浏览，怎么实现呢？其实，道理是一样的，可以把回退浏览的过程看成另一种方式的前进，把回退过程中依次访问的页面逐一保存在另一个栈 forward_Stack 中，这样，当访问者在回退的过程中，如果想再

一次依照之前前进的路径进行恢复时，可以从 forward_Stack 栈中去取最近回退的页面，实现了回退过程中再前进浏览的功能。

也就是说，Web 导航的功能可以通过设置两个辅助栈的方式来实现。

【案例 3.3】 稳定婚姻问题。

稳定婚姻问题是生活中的一个典型问题，可通俗地叙述为有 N 位男生和 N 位女生最后要组成稳定的婚姻家庭；择偶过程开始之前，男生和女生在各自的心目中都按照喜爱程度对 N 位异性有了各自的排序，然后开始选择自己的对象，要求组成 N 对伴侣是感情稳定的。稳定的标准就是在最大程度上尊重男生和女生愿望的前提下，任何一位男生和女生能找到各自满意的对象。

1962 年，美国数学家 David Gale 和 Lloyd Shapley 发明了一种寻找稳定婚姻的策略，称为延迟认可算法(Gale-Shapley 算法)。

其核心思想如下。

先对所有男生进行落选标记，称其为自由男。当存在自由男时，进行以下操作。

(1) 每一位自由男在所有尚未拒绝他的女士中选择一位被他排名最优先的女生。

(2) 每一位女生将正在追求她的自由男与其当前男友进行比较，选择其中排名优先的男生作为其男友，即若自由男优于当前男友，则抛弃前男友；否则保留其男友，拒绝自由男。

(3) 若某男生被其女友抛弃，重新变成自由男。

该问题可以选择队列作为核心数据结构来进行求解。

以上是在实际应用中，可能会遇到的典型的栈与队列问题，其求解需要选择相关的数据结构并设计相关的算法来实现。对这些问题的具体求解过程，在介绍完栈与队列的基本知识后，再进行详细描述。

3.2 栈

3.2.1 栈的定义及基本操作

栈是限制在表的一端进行插入和删除的线性表。在线性表中允许插入、删除的这一端称为栈顶，栈顶的当前位置是动态变化的；不允许插入和删除的另一端称为栈底，栈底是固定不变的。当表中没有元素时称为空栈。栈的插入操作称为进栈、压栈或入栈，栈的删除操作称为退栈或出栈。图 3.4 所示为栈的进栈和出栈过程，进栈的顺序是 e_1、e_2、e_3，出栈的顺序为 e_3、e_2、e_1，所以栈又称为后进先出线性表(last in first out)，简称 LIFO 表或称先进后出线性表。

在日常生活中，有很多后进先出的例子，如食堂里碟子在叠放时是从下到上，从大到小，在取碟子时，则是从上到下，从小到大。在程序设计中，常常需要栈这样的数据结构，使得取数据与保存数据时呈相反的顺序。对于栈，常用的基本操作如下。

(1) 栈初始化：Init_Stack()。

初始条件：栈 S 不存在。

操作结果：构造了一个空栈 S。

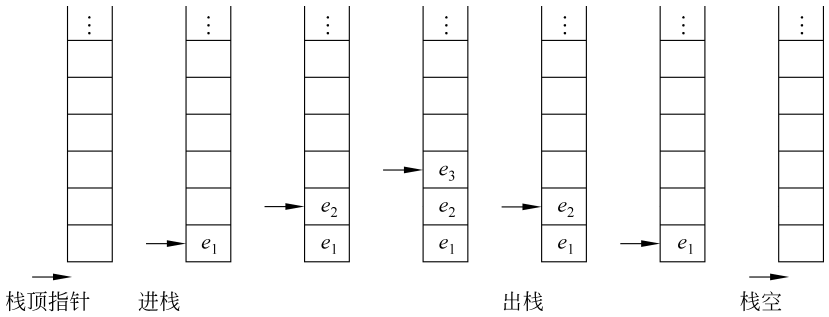

图 3.4　进出栈示意图

（2）判栈空：Empty_Stack(S)。

操作结果：若 S 为空栈返回为 1，否则返回为 0。

（3）入栈：Push_Stack(S,x)。

初始条件：栈 S 已存在。

操作结果：在栈 S 的顶部插入一个新元素 x，x 成为新的栈顶元素。栈发生变化。

（4）出栈：Pop_Stack(S)。

初始条件：栈 S 存在且非空。

操作结果：栈 S 的顶部元素从栈中删除，栈中少了一个元素。栈发生变化。

（5）取栈顶元素：GetTop_Stack(S)。

初始条件：栈 S 存在且非空。

操作结果：栈顶元素作为结果返回，栈不变化。

（6）销毁栈：Destroy_Stack(S)。

初始条件：栈 S 已存在。

操作结果：销毁一个已存在的栈。

3.2.2　栈的顺序存储及操作实现

栈的存储与一般线性表的实现类似，也有两种存储方式：顺序存储和链式存储。

利用顺序存储方式实现的栈称为顺序栈。类似于顺序表的定义，要分配一块连续的存储空间存放栈中的元素，用一个一维数组来实现：DataType data[MAXSIZE]，栈底位置可以固定设置在数组的任一端，如固定在下标为－1 的位置，而栈顶指示当前实际的栈顶元素位置，它是随着插入和删除而变化的，用一个 int top 变量指明当前栈顶的位置，同样将 data 和 top 封装在一个结构中，顺序栈的类型描述如下：

```
#define MAXSIZE  100
typedef  struct {
    DataType  data[MAXSIZE];
    int  top;
}SeqStack, * PSeqStack;
```

定义一个指向顺序栈的指针：

```
PSeqStack  S;
S=(PSeqStack)malloc(sizeof(SeqStack));
```

栈的顺序存储如图 3.5 所示。

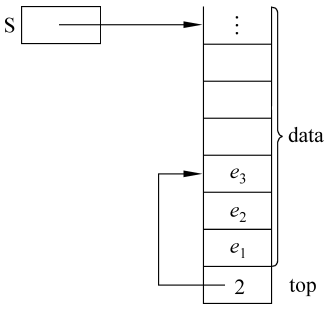

图 3.5 栈的存储示意图

由于顺序栈是静态分配存储，而栈的操作是一个动态过程：随着入栈的进行，有可能栈中元素的个数超过给栈分配的最大空间的大小，这时产生栈的溢出现象——上溢；随着出栈的进行，也有可能栈中元素全部出栈，这时栈中再也没有元素出栈了，也会造成栈的溢出现象——下溢。所以，在下面的操作实现中，请读者注意在出栈和入栈时要首先进行栈空和栈满的检测。

顺序栈的基本操作实现如下。

1. 初始化空栈

顺序栈的初始化即构造一个空栈，要返回一个指向顺序栈的指针。首先动态分配存储空间，然后，将栈中 top 置为－1，表示空栈。具体算法如下。

【算法 3.1】

```
PSeqStack  Init_SeqStack(void)
{  /*创建一个顺序栈,入口参数无,返回一个指向顺序栈的指针,为零表示分配空间失败*/
    PSeqStack  S;
    S=(PSeqStack)malloc(sizeof(SeqStack));
    if (S)
      S->top=-1;
    return S;
}
```

2. 判栈空

判断栈中是否有元素，算法思想：只需判断 top 是否等于－1 即可。具体算法如下。

【算法 3.2】

```
int Empty_SeqStack(PSeqStack  S)
{  /*判断栈是否为空,入口参数:顺序栈,返回值:1表示为空,0表示非空*/
   if (S->top==-1)
      return 1;
   else
      return 0;
}
```

3. 入栈

入栈操作是在栈的顶部进行插入操作，也即相当于在线性表的表尾进行插入，因而无

须移动元素。算法思想：首先判断栈是否已满，若满则退出，否则，由于栈的 top 指向栈顶，只要将入栈元素赋到 top+1 的位置，同时执行 top++即可。具体算法如下。

【算法 3.3】

```
int Push_SeqStack(PSeqStack  S, DataType  x)
{  /*在栈顶插入一新元素 x,入口参数：顺序栈,返回值：1 表示入栈成功,0 表示失败*/
  if (S->top==MAXSIZE-1)
    return 0;                    /*栈满不能入栈*/
  else
  {   S->top++;
      S->data[S->top]=x;
      return 1;
  }
}
```

4. 出栈

出栈操作是在栈的顶部进行删除操作，也即相当于在线性表的表尾进行删除，因而无须移动元素。算法思想：首先判断栈是否为空，若空则退出，否则，由于栈的 top 指向栈顶，只要修改 top 为 top－1 即可。具体算法如下。

【算法 3.4】

```
int  Pop_SeqStack  S, DataType*x)
{  /*删除栈顶元素并保存在*x,入口参数：顺序栈,返回值：1 表示出栈成功,0 表示失败*/
    if (Empty_SeqStack(S))
        return 0;                /*栈空不能出栈*/
    else
    {   *x=S->data[S->top];
       S->top--;
       return 1;
    }
}
```

5. 取栈顶元素

取栈顶元素操作是取出栈顶指针 top 所指的元素值。算法思想：首先判断栈是否为空，若空则退出，否则，由于栈的 top 指向栈顶，返回 top 所指单元的值，栈不发生变化。具体算法如下。

【算法 3.5】

```
int  GetTop_SeqStack  S, DataType  *X)
{  /*取出栈顶元素,入口参数：顺序栈,被取出的元素指针,这里用指针带出栈顶值*/
   /*返回值：1 表示成功,0 表示失败*/
   if (Empty_SeqStack(S))
      return 0;                          /*给出栈空信息*/
```

```
    else
       *x=S->data[S->top];                    /*栈顶元素存入*x中*/
       return(1);

}
```

6. 销毁栈

顺序栈被构造,使用完后,必须要销毁,否则可能会造成申请的内存不能释放,顺序栈的销毁操作是初始化操作的逆运算。由于要修改栈的指针变量,所以要将指针地址传给该函数。首先判断要销毁的栈是否存在,然后在顺序表存在的情况下释放该顺序表所占用的空间,将顺序栈指针赋0。具体算法如下。

【算法 3.6】

```
void Destroy_SeqStack(PSeqStack  *S)
{  /*销毁顺序栈,入口参数:为要销毁的顺序栈指针地址,无返回值*/
   if (*S)
      free(*S);
   *S=NULL;
   return;
}
```

设调用函数为主函数,主函数对初始化函数和销毁函数的调用如下:

```
main()
{  PSeqStack  S;
   S=Init_SeqStack();
   ⋮
   Destroy_SeqStack(&S);
}
```

3.2.3 栈的链式存储及操作实现

栈也可以用链式存储方式实现,一般链栈用单链表表示,其结点结构与单链表的结构相同,即结点结构为

```
typedef  struct node {
    DataType data;
    struct node  *next;
}StackNode, *PStackNode;
```

因为栈的主要操作是在栈顶插入、删除,显然以链表的头部做栈顶是最方便的,而且没有必要像单链表那样为了操作方便附加一个头结点,同时为了方便操作和强调栈顶是栈的一个属性,定义一个栈:

```
typedef struct {
    PStackNode  top;
}LinkStack, * PLinkStack;
PLinkStack  S;
S=(PLinkStack)malloc(sizeof(LinkStack));
```

栈的链式存储如图 3.6 所示。

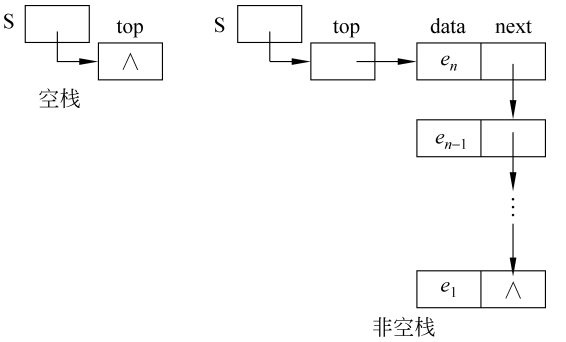

图 3.6　栈的链式存储示意图

链栈基本操作的实现如下。

1. 初始化空栈

【算法 3.7】

```
PLinkStack  Init_LinkStack(void)
{  /*初始化链栈,入口参数:空,返回值:链栈指针,null 表示初始化失败*/
     PLinkStack S;
     S=(PLinkStack)malloc(sizeof(LinkStack));
     if (S)   S->top=NULL;
     return(S);
}
```

2. 判栈空

【算法 3.8】

```
int  Empty_LinkStack(PLinkStack  S)
{  /*判断链栈是否为空,入口参数:链栈指针,返回值:1 表示栈空,0 表示栈非空*/
   return(S->top==NULL);
}
```

3. 入栈

【算法 3.9】

```
int  Push_LinkStack(PLinkStack  S, DataType x)
```

```
{  /*进栈,入口参数:链栈指针,进栈元素,返回值:1表示入栈成功,0表示失败*/
   PStackNode p;
   p=(PStackNode) malloc(sizeof(StackNode));
   if (!p)
   {
      printf("内存溢出");
      return(0);
   }
   p->data=x;
   p->next=S->top;
   S->top=p;
   return(1);
}
```

4. 出栈

【算法 3.10】

```
int Pop_LinkStack(PLinkStack  S, DataType *x)
{  /*出栈,返回值:1表示出栈成功,0表示失败,*x保存被删除的元素值*/
   PStackNode p;
   if (Empty_LinkStack(S))
   {
      printf("栈空,不能出栈");
      return(0);
   }
   *x=S->top->data;
   p=S->top;
   S->top=S->top->next;
   free(p);
   return(1);
}
```

5. 取栈顶元素

【算法 3.11】

```
int GetTop_LinkStack(PLinkStack  S, DataType  *x)
{  /*得到栈顶元素,入口参数:链栈指针,出栈元素存放空间地址*/
   /*返回值:1表示出栈成功,0表示失败*/
   if (Empty_LinkStack(S))
   {
        printf("栈空");
        return(0);                        /*栈空*/
   }
```

```
    *x=S->top->data;                    /*栈顶元素存入*x中*/
    return(1);
}
```

6. 销毁栈

链栈被构造，使用完后，必须要销毁，否则可能会造成申请的内存不能释放。具体算法如下：

【算法 3.12】

```
void Destroy_LinkStack(PLinkStack  *LS)
{  /*销毁链栈，入口参数：要销毁的链栈指针地址，无返回值*/
   PStackNode  p, q
   if (*LS)
   {
        p=(*LS)->top;
        while(p)
        {
            q=p;
            p=p->next;
            free(q);
        }
        free(*LS);
   }
   *LS=NULL;
   return;
}
```

主函数对初始化函数和销毁函数的调用方法类似算法 3.6。

3.3 栈的应用举例

由于栈的“后进先出”特点，在很多实际问题中都利用栈做一个辅助的数据结构来实现逆向操作的求解，下面通过几个例子进行说明。

【例 3.1】 数制转换问题。

将十进制数 N 转换为 r 进制的数，其转换方法利用辗转相除法：以 $N=1234$，$r=8$ 为例转换方法如下：

N	$N/8$（整除）	$N\%8$（求余）	
1234	154	2	低
154	19	2	
19	2	3	
2	0	2	高

所以：$(1234)_{10}=(2322)_8$

我们看到所转换的八进制数按从低位到高位的顺序产生，而通常的输出应该从高位到低位，与计算过程正好相反，因此转换过程中每得到一位八进制数则进栈保存，转换完毕后依次出栈则正好是转换结果。

算法思想如下。

(1) 初始化栈，初始化 N 为要转换的数，r 为进制数。

(2) 判断 N 的值，为 0 转步骤(4)，否则 $N\%r$ 压入栈 s 中。

(3) 用 N/r 代替 N 转步骤(2)。

(4) 出栈，出栈序列即为结果。

具体算法如下。

【算法 3.13】

```
typedef  int  DataType;
int conversion(int n,int  r)
{  PSeqStack  S;                          /*定义一个顺序栈*/
   DataType   x;
   if (!r)
   {
      printf("基数不能为 0");
      return(0);
   }
   S=Init_SeqStack();                     /*初始化栈*/
   if (!S)
   {
      printf("栈初始化失败");
      return(0);
   }
   while (n)
   {
      Push_SeqStack(S,n%r);               /*余数入栈*/
      n=n/ r;                             /*商作为被除数继续*/
   }
   while (!Empty_SeqStack(S))             /*直到栈空退出循环*/
   {    Pop_SeqStack(S,&x);               /*弹出栈顶元素*/
        printf("%d ",x);                  /*输出栈顶元素*/
   }
   Destroy_ SeqStack(&S);                 /*销毁栈*/
}
```

当应用程序中需要一个与数据保存时顺序相反的数据时，通常使用栈。用顺序栈的情况较多。

【例 3.2】 利用栈实现迷宫的求解。

问题：这是实验心理学中的一个经典问题，心理学家把一只老鼠从一个无顶盖的大

盒子的入口处赶进迷宫。迷宫中设置很多隔壁，对前进方向形成了多处障碍，心理学家在迷宫的唯一出口处放置了一块奶酪，吸引老鼠在迷宫中寻找通路以到达出口。

求解思想：回溯法是一种不断试探且及时纠正错误的搜索方法。下面的求解过程即回溯法。从入口出发，按某一方向向前探索，若能走通并且未走过，即某处可以到达，则到达新点，否则试探下一个方向；若所有的方向均没有通路，则沿原路返回前一点，换下一个方向再继续试探，直到找到一条通路，或无路可走又返回入口点。

在求解过程中，为了保证在到达某一点后不能向前继续行走（无路）时，能正确返回前一点以便继续从下一个方向向前试探，则需要用一个栈保存所能够到达的每一点的下标及从该点前进的方向。

实现该算法需要解决的4个问题如下。

1）表示迷宫的数据结构

设迷宫为 m 行 n 列，利用 maze[m][n]来表示一个迷宫，maze[i][j]=0 或 1。其中，0 表示通路，1 表示不通，当从某点向下试探时，中间点有 4 个方向可以试探，而 4 个角点有两个方向，其他边缘点有三个方向，用 maze[m+2][n+2]来表示迷宫，而迷宫的四周的值全部为 1，这样做使问题简单了，每个点的试探方向全部为 4（实际上是 8 个方向，本书为将问题简单化只考虑 4 个方向），不用再判断当前点的试探方向有几个。

如图 3.7 所示的迷宫是一个 6×8 的迷宫。

入口(1,1)

	0	1	2	3	4	5	6	7	8	9
0	1	1	1	1	1	1	1	1	1	1
1	1	0	1	1	1	0	1	1	1	1
2	1	0	0	0	0	1	1	1	1	1
3	1	0	1	0	0	0	0	0	1	1
4	1	0	1	1	1	0	0	1	1	1
5	1	1	0	0	1	1	0	0	0	1
6	1	0	1	1	0	0	1	1	0	1
7	1	1	1	1	1	1	1	1	1	1

出口(6,8)

图 3.7　用 maze[m+2][n+2]表示的迷宫

入口坐标为(1,1)，出口坐标为(6,8)。

迷宫的定义如下：

```
#define  m   6                    /*迷宫的实际行*/
#define  n   8                    /*迷宫的实际列*/
int maze[m+2][n+2];
```

2）试探方向

在上述表示迷宫的情况下，每个点有 4 个方向去试探，如当前点的坐标(x,y)，与其相邻的 4 个点的坐标都可以根据与该点的相邻方位而得到，如图 3.8 所示。因为出口在(m,n)，因此，试探顺序规定为：从当前位置向前试探的方向为从正东沿顺时针方向进

行。为了简化问题，方便求出新点的坐标，将从正东开始沿顺时针进行的这 4 个方向的坐标增量放在一个结构数组 move[4]中，在 move 数组中，每个元素有两个域组成，x 为横坐标增量，y 为纵坐标增量。move 数组如图 3.9 所示。

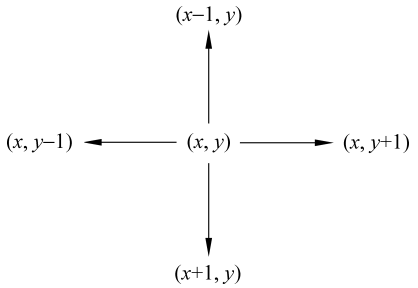

图 3.8 与点(x,y)相邻的 4 个点及坐标

	x	y
0	0	1
1	1	0
2	0	−1
3	−1	0

图 3.9 增量数组 move

move 数组定义如下：

```
typedef  struct {
  int x,y;
} item;
item move[4];
```

这样对 move 的设计会很方便地求出从某点(x,y)按某一方向 v($0 \leqslant v \leqslant 3$) 到达的新点($i$,$j$)的坐标：$i = x +$ move[v].x；$j = y +$ move[v].y；。

3) 栈的设计

当到达了某点而无路可走时需返回前一点，再从前一点开始向下一个方向继续试探。因此，压入栈中的不仅是顺序到达的各点的坐标，而且还要有从前一点到达本点的方向。栈中元素是一个由行、列、方向组成，栈元素的设计如下：

```
typedef struct {
  int x, y, d;                    /* 横纵坐标及方向 */
}DataType;
```

栈的定义仍然为：PSeqStack S；。

4) 如何防止重复到达某点，以避免发生死循环

一种方法是另外设置一个标志数组 mark[m][n]，它的所有元素都初始化为 0，一旦到达了某一点 (i,j)之后，使 mark[i][j]置 1，下次再试探这个位置时就不能再走了。另一种方法是当到达某点(i,j)后使 maze[i][j]置 −1，以便区别未到达过的点，同样也能起到防止走重复点的目的，本算法采用后一种方法，算法结束前可恢复原迷宫。

迷宫求解算法思想如下。

(1) 栈初始化。

(2) 将入口点坐标及到达该点的方向(设为 −1)入栈。

(3) while (栈不空)

```
{    栈顶元素=>(x, y, d)
```

```
    出栈;
    求出下一个要试探的方向 d++;
    while  (还有剩余试探方向时)
    {    if(d 方向可走)
         则{(x, y, d)入栈;
            求新点坐标  (i, j);
            将新点(i, j)切换为当前点(x, y);
            if((x,y)==(m,n))结束;
            else 重置 d=0;
           }
         else  d++;
    }
}
```

找不到通路,结束!

具体算法如下。

【算法 3.14】

```
#define  m   6                                  /*迷宫的实际行*/
#define  n   8                                  /*迷宫的实际列*/
int  mazepath(int maze[][n+2],item move[],int x0,int y0)
{  /*求迷宫路径,入口参数:指向迷宫数组的指针,下标移动的增量数组,开始点(x0,y0),到
     达点(m,n),返回值:1表示求出路径,0表示无路径*/
   PSeqStack  S;
   DataType  temp;
   int x, y, d, i, j;
   temp.x=x0;temp.y=y0;  temp.d=-1;
   S=Init_SeqStack();                         /*初始化栈*/
   if(!S)
   {  printf("栈初始化失败");
      return(0);
   }
   Push_SeqStack(S,temp);                     /*迷宫入口点入栈*/
   while (!Empty_SeqStack(S))
   {  Pop_SeqStack(S,&temp);
      x=temp.x;  y=temp.y;  d=temp.d+1;
      while(d<4)                              /*存在剩余方向可以搜索*/
      {   i=x+move[d].x;   j=y+move[d].y;
          if(maze[i][j]==0)                   /*此方向可走*/
          {temp.x=x;
          temp.y=y;
          temp.d=d;
          Push_SeqStack(S, temp);
                                              /*点{x,y}可以走,用栈保存可以走的路径*/
              x=i;  y=j;  maze[x][y]=-1;
```

```
            if (x==m&&y==n)                    /*迷宫有路*/
            {  while (!Empty_SeqStack(S))
               { Pop_SeqStack (S,&temp);
                 printf("(%d,%d)<-",temp.x,temp.y);
                                               /*打印可走的路径*/
               }
               Destroy_SeqStack(&S);           /*销毁栈*/
               return 1;
            }
            else  d=0;                         /*方向复位,从第一个方向开始试探*/
         }
         else  d++;                            /*试探下一个方向*/
      }                                        /*while (d<4)*/
   }                                           /*while*/
   Destroy_SeqStack(&S);                       /*销毁栈*/
   return  0;                                  /*迷宫无路*/
}
```

【例 3.3】 表达式求值。

表达式求值是程序设计语言编译中一个最基本的问题。它的实现也是栈的应用中的典型例子之一。

任何一个表达式都是由操作数、运算符和界限符组成的有意义的式子。一般地，操作数既可以是常数，也可以是变量或常量。运算符从运算对象的个数上分，有单目运算符、双目运算符和三目运算符；从运算类型上分，有算术运算、关系运算、逻辑运算。界限符有左右括号和表达式结束符等。运算符、界限符统称为算符。为简单化，在这里，仅限于讨论只含二目运算符的加、减、乘、除算术表达式，并且操作数为一位字符表示的整数。

在表达式求值时，一般表达式有以下 3 种表示形式。

(1) 后缀表示：＜操作数＞＜操作数＞＜运算符＞。

(2) 中缀表示：＜操作数＞＜运算符＞＜操作数＞。

(3) 前缀表示：＜运算符＞＜操作数＞＜操作数＞。

平常所用的表达式都是中缀表达。如：1＋2＊(8－5)－4/2。

由于中缀表示中有算符的优先级问题，有时还采用括号改变运算顺序，因此一般在表达式求值中，较少采用中缀表示，在编译系统中更常见的是采用后缀表示。上述式子的计算顺序和用后缀表示的计算顺序如图 3.10 所示。

1) 后缀表达式(也称逆波兰式)求值

由于后缀表达式的操作数总在运算符之前，并且表达式中即无括号又无优先级的约束，算法比较简单。具体做法是，只使用一个操作数栈，当从左向右扫描表达式时，每遇到一个操作数就送入栈中保存，每遇到一个运算符就从栈中取出两个操作数进行当前的计算，然后把结果再入栈，直到整个表达式结束，这时送入栈顶的值就是结果。

下面是后缀表达式求值的算法，在下面的算法中假设，每个表达式是合乎语法的，并且假设后缀表达式已被存入一个足够大的字符数组 A 中，且以＃为结束字符。

中缀表达式计算顺序　　后缀表达式计算顺序

图 3.10　中缀、后缀表达式计算顺序

【算法 3.15】

```
typedef  double DataType;
int IsNum(char c)
{  /*判断字符是否为操作数。若是返回 1,否则返回 0*/
  if(c>='0' && c<='9')  return(1);
  else  return(0);
}
double  postfix_exp(char *A)
{  /*本函数返回由后缀表达式 A 表示的表达式运算结果*/
  PSeqStack  S;
  double  Result,a,b,c; char ch;
  ch=*A++;
  S=Init_SeqStack();                    /*初始化栈*/
  while (ch!='#')
  {  if(IsNum(ch))  Push_SeqStack (S, ch-'0');
     else
     {    Pop_SeqStack(S,&b);
          Pop_SeqStack(S,&a);           /*取出两个运算量*/
          switch (ch)
            { case  '+':  c=a+b; break;
              case  '-':  c=a-b; break;
              case  '*': c=a*b;break;
              case  '/': c=a/b; break;
              case  '%':  c=(int)a%(int)b; break;
            }
          Push_SeqStack(S, c);
     }
     ch=*A++;
  }
  GetTop _SeqStack(S,&Result);
  Destroy_ SeqStack(&S);                /*销毁栈*/
  return  Result;
}
```

2）中缀表达式转换为后缀表达式

根据中缀表达式中算术运算规则，式子：＜操作数＞θ_1＜操作数＞θ_2＜操作数＞中，θ_1、θ_2 运算优先级如图 3.11 所示。为简单起见，算符仅限图 3.11 所示的几种，操作数仅限个位数。

算符	#	)	+	–	*	/	(
优先级	1	2	3	3	4	4	5

图 3.11 算符优先级定义

表达式作为一个满足表达式语法规则的串存储，转换过程：初始化一个算符栈，并将结束符'＃'放入栈中，然后自左向右扫描表达式，直到扫描到'＃'并且当前栈顶也是'＃'时结束。当扫描到的是操作数时直接输出，扫描到算符时不能马上输出，因为后面可能还有更高优先级的运算，要对下列几种情况分别处理。

（1）算符栈栈顶算符是'('，如果当前扫描到的算符是')'，则算法出栈不作任何处理，同时扫描下个字符，此过程称为脱括号；如果当前扫描到的算符不是')'，则当前算符进栈。

（2）算符栈栈顶算符不是'('，并且算符栈栈顶算符优先级比当前扫描到的算符优先级低，则入栈；若算符栈栈顶算符优先级比当前扫描到的算符优先级高（或相等），则从算符栈出栈并输出，当前算符继续与新的栈顶算符比较。如表达式“1＋2＊（8－5）－ 4/2＃”的转换过程如表 3.1 所示。为简单起见，先在栈中放入一个结束符＃。

表 3.1 中缀表达式 1＋2＊（8－5）－4/2 的转换过程

读字符	符栈S	说　明	输出的后缀表达式
1	#	1为操作数直接输出	1
+	#+	#＜+，+入栈S	1
2	#+	2为操作数直接输出	12
*	#+*	+＜*，*入栈S	12
(	#+*(	*＜(，(入栈S	12
8	#+*(	8为操作数直接输出	128
–	#+*(–	栈顶为(，–直接入栈S	128
5	#+*(–	5为操作数直接输出	1285
)	#+*(	–＞)，–出栈输出	1285–
	#+*	(＝)，(出栈，脱括号	1285–
–	#+*	*＞–，*出栈输出	1285–*
	#	+＞=–，+出栈输出	1285–*+
	#–	#＜–，–入栈S	1285–*+
4	#–	4为操作数直接输出	1285–*+4
/	#–/	–＜/，/入栈S	1285–*+4
2	#–/	2为操作数直接输出	1285–*+42
#	#–	/＞#，/出栈输出	1285–*+42/
	#	–＞#，–出栈输出	1285–*+42/–
		#=#，#出栈，栈空结束	1285–*+42/–

上述操作的算法步骤如下。

（1）初始化算符栈 s，将结束符'#'加入算符栈 s 中。

（2）读表达式字符＝＞w。

（3）当栈顶为'#'并且 w 也是'#'时结束；否则循环做下列步骤。

（3-1）如果 w 是操作数，直接输出，读下一个字符＝＞w；转步骤(3)。

（3-2）w 若是算符，则：

（3-2-1）如果栈顶为'('并且 w 为')' 则'('出栈不输出，读下一个字符＝＞w；转步骤(3)。

（3-2-2）如果栈顶为'('或者栈顶优先级小于 w 优先级，则 w 入栈，读下一个字符＝＞w；转步骤(3)。否则：从算符栈中出栈并输出，转步骤(3)。

为实现上述算法，要能够判断表达式字符是否是操作数（只考虑一位操作数），要能够比较出算符的优先级，因此用下面的两个函数实现。

（1）判断字符是否位操作数。若是返回 1，否则返回 0，实现方法如下：

```
int  IsNum(char c)
{   if(c>='0' && c<='9')  return(1);
    else  return(0);
}
```

（2）求算符优先级。

```
int  priority(char op)                    /* 按照图 3.11 给每个算符定义优先级 */
{  switch (op)
    {  case '#': return(1);
       case ')': return(2);
       case '+':
       case '-': return(3);
       case '*':
       case '/': return(4);
       case '(': return(5);
       default: return(0);
    }
}
```

将中缀表达式转换成后缀表达式的具体算法如下所示。

【算法 3.16】

```
typedef  char  DataType;
int infix_exp_value(char *infixexp,char *postfixexp)
{   PSeqStack  S;
    char  c,w, topelement;
    S=Init_SeqStack();                          /* 初始化栈 */
    if (!S)
```

```
    {   printf("栈初始化失败");
        return(0);
    }
    Push_SeqStack(S,'#');                          /*先在算符栈中放入'#'*/
    w=*infixexp;
    while((GetTop_SeqStack(S,&c),c)!='#'||w!='#')       /*栈顶元素不是'#'或w不
                                                          是'#'*/
    { if(IsNum(w))
      {   *postfixexp=w;
          postfixexp++;
          w=*(++infixexp);
      }
      else
      {
        if ((GetTop_SeqStack(S,&c),c)=='('&&w==')')     /*栈顶是'('并且栈外是
                                                          ')',脱括号*/
        {       Pop_SeqStack(S,&topelement);
                w=*(++infixexp);
        }
        else
          if ((GetTop_SeqStack(S,&c),c)=='('||
               priority((GetTop_SeqStack(S,&c),c))<priority(w))
          {    Push_SeqStack(S,w);
               w=*(++infixexp);
          }
          else
          {    Pop_SeqStack(S,&topelement);
              *postfixexp=topelement;
               postfixexp++;
          }
      }
    }
    *postfixexp='#';
    *(++postfixexp)='\0';                  /*添加字符串结束符号*/
    Destroy_SeqStack(&S);                  /*销毁栈*/
    return(1);
}
```

这个算法可以将中缀表达式“1+2*(8-5)- 4/2#”转化为后缀表达式“1 2 8 5 - * + 4 2 / - #”。

表达式的求值，如果是后缀表达式，可以直接采用后缀表达式求值的算法求解，如果是中缀表达式，可以采用先将中缀表达式转换为后缀表达式，再用后缀表达式算法求解。读者也可以直接用中缀表达式求值，算法和将中缀表达式转换成后缀表达式算法类似，只是要设两个栈，一个保存运算符，一个保存操作数，在算符栈栈顶算符大于当前算符时

不是输出运算符而是直接运算出结果入操作数栈。读者可以自己去实现。另外，对于前缀表达式，由于较少采用，有兴趣的读者可以自行考虑如何实现。

【例 3.4】 Web 导航。

Web 导航问题在 3.1 节有总体上的描述。其主要解决的问题是，使用者在浏览网页的时候，可以根据当时访问的需要，对曾经访问过的页面，进行回退或者再前进。

假如用户已经浏览 5 个网页：a、b、c、d、e，这里用小写的英文字母表示具体的网页名称，假设在浏览网页 e 后，用户回退进行浏览，回退到 d 网页，继续回退到 c 网页，然后又从 c 网页向前推进到 d 网页，再继续推进并停止在 e 网页。

用字符序列来表示网页访问的整个过程就是

```
a, b, c, d, e, d, c, d, e
```

它所对应的访问操作流就是访问 a，访问 b，访问 c，访问 d，访问 e，单击 back（退至 d），单击 back（退至 c），单击 forward（进至 d），单击 forward（进至 e），单击 forward（停止在 e）。

下面用算法来模拟这样的过程。

设置两个辅助字符栈：back_Stack 和 forward_Stack，分别保存前进路径上访问的页面序列和回退路径上访问的页面序列，back_Stack 栈是为了回退后恢复页面使用的，而 forward_Stack 栈是为了回退后再前进时恢复页面使用的。

用读取（或者输入）字符的方式表示访问某个页面，用＞、＜字符分别表示前进和后退。在不断向前访问页面的过程中，连续把刚刚访问的页面保存到 back_Stack 中，如果在访问的某个时刻，需要回退，则从 back_Stack 栈中恢复最近刚访问过的页面，并且把回退看成是一个反方向上的前进，因此，在回退恢复刚访问过页面之前，也把当前页面保存到 forward_Stack 中，以备回退后需要再前进时恢复曾经访问过的页面。

这里有一点要注意，就是回退后，再前进不是通过按＞键，而是重新打开一个新网页 X，那么此时，forward_Stack 栈要清空，因为前进的路径变了，回退后再恢复的页面，已经不是之前 forward_Stack 栈中所保存的页面了，这一点和 Word 中撤销键入和恢复键入的处理模式很相似。

具体的算法实现如下。

【算法 3.17】

```
typedef char DataType;
int ischar(char ch)                          /*定义判断 ch 是否是字符的函数*/
{
    if(ch>='a' && ch <='z')return 1;
    if(ch>='A' && ch <='Z')return 1;
    return 0;
}
void main()
{
    char ay_visiturl[20]={'a','b','c','d','e','<','<','>','>','>','#'};
```