第3章 栈和队列

本章关键词：策略工具和递归思想。

关键词——策略工具。栈和队列是两种特殊的线性表，可以看作是一对双胞胎，它们是操作受限的线性结构。在算法设计模式中有深度优先策略和广度优先策略，支撑它们的工具就是栈和队列这两种数据结构。在本章中给出了采用两种策略来解决迷宫问题。类似的农夫过河问题既可以用队列解决，也可以用栈解决。读者需要理解对于同一个应用可以采用不同的策略和工具来解决，而具体采用哪种策略更合适，和应用问题的场景有关。

关键词——递归思想。递归思想贯穿在整个课程当中，例如链表的定义和建立、树和二叉树的定义和遍历、快速排序和堆排序算法等。栈和递归有着紧密的联系，请结合函数调用理解调用栈并掌握通过栈结构将递归程序转换为非递归的方法。递归算法虽然简洁、清晰，但空间效率低，需要读者明确地意识到。所以并不是所有的问题都适合递归实现，在这里强调的是既要理解递归的过程也要有意识地培养递归的思想。

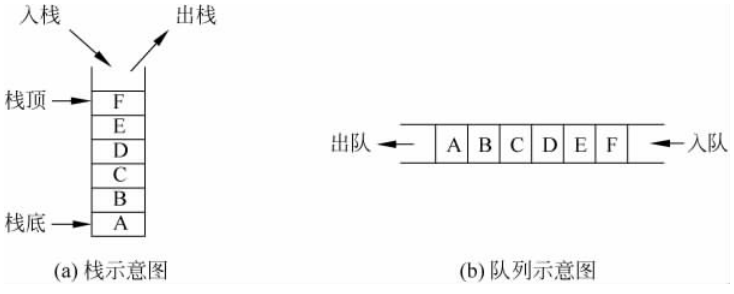
“如切如磋者，道学也；如琢如磨者，自修也。”

——《大学》

3.1 栈和队列的概念

3.1.1 栈和队列的定义

栈和队列是线性表的特例，特殊性在于它们都是操作受限的线性表。栈限定在表尾进行插入或删除。表尾端称**栈顶**，表头端称**栈底**。在图 3-1(a)中，如果依次插入元素 A、B、C、D、E、F，那么 F 就是从栈中删除的第一个元素，所以栈的特点是**后进先出**(LIFO)。在现实



(a) 栈示意图　　(b) 队列示意图

图 3-1　栈和队列示意图

生活中，可以把摞在一起的盘子看作一个栈，取出和放入盘子都是在最顶端进行，这一端为栈顶，另外不能操作的端为栈底。

队列限定在一端进行插入操作、在另一端进行删除操作。插入端称**队尾**，删除端称**队头**。在图 3-1(b)中，如果依次在队尾插入元素 A、B、C、D、E、F，那么 A 就是从队列中删除的第一个元素，可以看出队列的特点是**先进先出**(FIFO)。把移动营业厅或银行对客户的服务看作一个队列，服务的次序是根据客户取号的先后次序办理业务，先来先服务。

3.1.2 栈的抽象数据类型定义

栈的主要操作包括创建一个空栈、判空、入栈(插入栈顶元素)、出栈(删除栈顶元素)和取栈顶元素。栈的抽象数据类型定义如下：

```
ADT Stack is
operations
    Stack SetNullStack(void)
    创建一个空栈
    int IsNullStack(Stack stack)
    判断栈 stack 是否为空
    void Push(Stack stack,Datatype x)
    向栈 stack 中插入元素 x
    void Pop(Stack stack)
    从栈 stack 中删除一个元素
    Datatype Top (Stack stack)
    取栈顶元素
End ADT Stack
```

与栈类似，队列的主要操作包括创建一个空队列、判空、入队(在队尾插入元素)、出队(删除队头元素)和取队头元素。队列的抽象数据类型定义如下：

```
ADT Queue is
operations
    Queue SetNullQueue (void)
    创建一个空队列
    int IsNullQueue (Queue que)
    判断队列 que 是否为空
    void EnQueue (Queue que,Datatype x)
    向队列 que 中插入元素 x
    void DeQueue (Queue que)
    从队列 que 中删除一个元素
    Datatype Front(Queue que)
    取队头元素
End ADT Queue
```

3.1.3 栈混洗

n 个数据(a_1、a_2、…、a_n)依次进栈，并随时可能出栈，按照其出栈次序得到的每一个序列(a_{k1}，a_{k2}，…，a_{kn})称为一个栈混洗。

假设现在有 3 个元素(i、j、k)按照先后次序压入栈中,则可能的栈混洗有(i, j, k)、(k,j, i)、(i, k, j)、(j, i, k)、(j, k, i)。(k, i, j)必然不是栈混洗。表 3-1 给出了每种栈混洗的操作过程。

表 3-1 栈混洗举例

栈混洗	操作 1	操作 2	操作 3	操作 4	操作 5	操作 6
(i, j, k)	push(i)	pop(i)	push(j)	pop(j)	push(k)	pop(k)
(k,j, i)	push(i)	push(j)	push(k)	pop(k)	pop(j)	pop(i)
(i, k, j)	push(i)	pop(i)	push(j)	push(k)	pop(k)	pop(j)
(j, i, k)	push(i)	push(j)	pop(j)	pop(i)	push(k)	pop(k)
(j, k, i)	push(i)	push(j)	pop(j)	push(k)	pop(k)	pop(i)

从表 3-1 可以看出,对于长度为 n 的输入序列,每一个栈混洗一般对应由 n 次 push 和 n 次 pop 构成的合法序列,反之,由 n 次 push 和 n 次 pop 构成的序列,只要满足“任一前缀中的 push 不少于 pop”这个限制,则该序列必然对应一个栈混洗。

对于上述 3 个数据的栈混洗,可以推广到栈混洗甄别的一般情况:任意 3 个元素能否按某相对次序出现于栈混洗中与其他元素无关。对于任何 $1 \leqslant i < j < k \leqslant n$,($k$, i, j)必然不是栈混洗,其余为栈混洗。推广到更为一般的序列(1, 2, 3,…,i,…,j,…,k,…,n),形如(,…,k,…,i,…,j…)的序列必然不是栈混洗。

3.2 顺序栈

采用顺序存储结构的栈称为顺序栈。与顺序表类似,顺序栈用一组连续的存储单元来存放数据元素。栈的操作限定在栈顶进行,因此需要设置一个 top 变量来指示栈顶位置。初始为空栈,设置 top 为−1。在 C 语言中,数组的下标从 0 开始,顺序栈的数据元素也从下标 0 开始存放。入栈时,top 先加 1,再将元素放入 top 指示的数组单元中;出栈时,top 减 1,top 总是指向当前的栈顶元素。图 3-2 描述了栈中内容的变化。创建空栈,依次入栈 A、B、C、D,出栈 D,取栈顶元素,判断栈是否为空。特别要注意 Top()和 IsEmpty()这两个操作并没有影响栈中的内容。

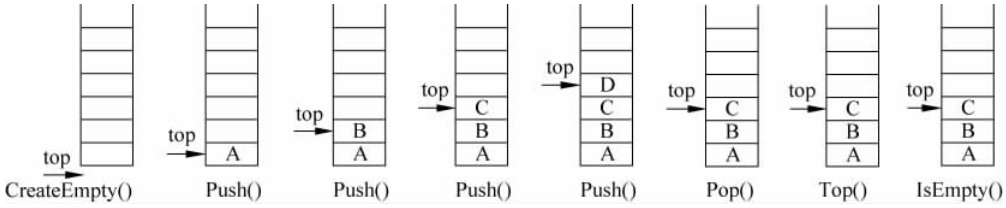


图 3-2 顺序栈的基本操作

顺序栈的类型定义如下:

```
1  typedef int DataType;
2  struct Stack
3  {
```

```
4    int MAX;                              //最大容量
5    int top;                              //栈顶指针
6    DataType * elem;                      //存放元素的起始指针
7  };
8  typedef struct Stack * SeqStack         //定义顺序栈类型
```

3.2.1 创建空栈

创建空的顺序栈就是为顺序栈分配一个预先定义的数组空间，并将顺序栈的栈顶 top 成员变量设置为－1。具体实现见算法 3-1。

算法 3-1 创建空的顺序栈。

```
1  SeqStack SetNullStack_Seq(int m)                  //创建空顺序栈,m 是分配的最大空间
2  {
3      SeqStack sstack = (SeqStack)malloc(sizeof(struct Stack));
4      if(sstack != NULL)
5      {
6          sstack -> elem = (int * )malloc(sizeof(int) * m);
7          if(sstack -> elem != NULL)
8          {
9              sstack -> MAX = m;                    //顺序栈最大容量
10             sstack -> top = -1;                   //设置栈顶初值为 -1
11             return(sstack);
12         }
13         else
14             {
15                 free(sstack);
16                 return NULL;
17             }
18     }
19     else
20     {
21         printf("Alloc failure!");
22         return NULL;
23     }
24 }
```

3.2.2 判断栈空

顺序栈的判空是检查栈顶指针是否等于初始化的－1，如果是－1，返回 1，否则返回 0。具体实现见算法 3-2。

算法 3-2 判断顺序栈是否为空。

```
1  int IsNullStack_seq(SeqStack sstack)        //判断一个栈是否为空
2  {
3      return(sstack -> top == -1);            //检查栈顶 top
4  }
```

3.2.3 进栈

顺序栈的进栈算法首先检查栈是否满了，也就是检查栈顶是否已经达到了最大值，如果

是,则不能再进行进栈操作,否则能够进栈。在进栈时需要先修改栈顶,然后将元素压入栈中。具体实现见算法 3-3。需要注意的是,在算法 3-1 中如果将第 8 行修改为 sstack->top=0,那么在算法 3-3 中需要将第 7 行和第 8 行进行交换,即先压入元素再修改栈顶。

算法 3-3 顺序栈的进栈算法。

```
1  void Push_seq(SeqStack sstack,int x)          //入栈
2  {
3      if( sstack->top>=(sstack->MAX-1)) //检查栈是否满
4          printf( "overflow! \n" );
5      else
6      {
7          sstack->top ++;                       //若不满,先修改栈顶变量
8          sstack->elem[sstack->top] = x;    //把元素 x 放到栈顶变量的位置中
9      }
10 }
```

3.2.4 出栈

顺序栈的出栈操作首先检查栈是否为空,如果是空栈,输出提示信息,否则栈顶指针减 1。具体实现见算法 3-4。需要注意的是在这里没有记录原来的栈顶元素,如果需要返回栈顶元素,需要修改算法 3-4。请读者自行完成。

算法 3-4 顺序栈的出栈算法。

```
1  void Pop_seq(SeqStack sstack)                 //出栈
2  {
3      if (IsNullStack_seq(sstack))              //判断栈是否为空,调用算法 3-2
4          printf("Underflow!\n");
5      else
6          sstack->top = sstack->top - 1;        //栈顶减 1
7  }
```

3.2.5 取栈顶元素

在取顺序栈的栈顶元素时,首先检查是否为空栈,如果是空栈,输出提示信息,否则返回栈顶元素。具体实现见算法 3-5。

算法 3-5 取顺序栈的栈顶元素。

```
1  DataType Top_seq(SeqStack sstack)     //取栈顶元素的值
2  {
3      if (IsNullStack_seq(sstack))      //判断 sstack 所指的栈是否为空栈,调用算法 3-2
4          printf("it is empty stack");
5      else
6          return sstack->elem[sstack->top];
7   }
```

3.3 链栈

采用链式存储的栈称为链栈，top 指示栈顶元素，栈底在链表尾部，如图 3-3 所示。

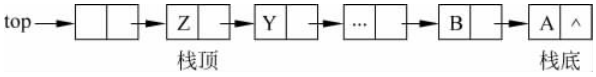


图 3-3 链栈示意图

链栈的类型定义如下：

```
typedef int DataType;
struct Node
{
    DataType       data;
    struct Node *   next;
};
typedef struct Node   * PNode;                    //结点类型
typedef struct Node   * top, * LinkStack;         //栈顶和链栈类型
```

3.3.1 创建空栈

创建带有头结点的空链栈，需要申请 struct Node 结构空间，并设置 top->next 为空。具体实现见算法 3-6。

算法 3-6 创建空的链栈。

```
LinkStack SetNullStack_Link()                     //创建空链栈
{
    LinkStack top = (LinkStack)malloc(sizeof(struct Node));
    if (top != NULL)
        top->next = NULL;
    else
        printf("Alloc failure");
    return top;                                   //返回栈顶指针
}
```

3.3.2 判断栈空

判断链栈是否为空只需要判断栈顶结点的后继指针是否为空，如果为空，返回 1，否则返回 0。具体实现见算法 3-7。

算法 3-7 判断链栈是否为空。

```
int IsNullStack_link(LinkStack top)               //判断一个链栈是否为空
{
    if (top->next == NULL)
        return 1;
```

```
5       else
6           return 0;
7   }
```

3.3.3 进栈

将一个元素压入到链栈中，首先要申请结点空间，然后给数据域和指针域赋值，并修改栈顶 top 的后继指针，将其赋值为新插入的结点。具体实现见算法 3-8。

算法 3-8 链栈的进栈算法。

```
1   void Push_link(LinkStack top, DataType x)          //进栈
2   {
3       PNode p;
4       p = (PNode)malloc(sizeof(struct Node));        //申请结点空间
5       if (p == NULL)
6           printf("Alloc failure");
7       else
8       {
9           p -> data = x;                             //数据域赋值
10          p -> next = top -> next;                   //指针域赋值
11          top -> next = p;                           //修改栈顶
12      }
13  }
```

3.3.4 出栈

在进行出栈操作时首先判断栈是否为空，如果栈不空，修改栈顶指针，释放结点空间。具体实现见算法 3-9。

算法 3-9 链栈的出栈算法。

```
1   void Pop_link(LinkStack top)                       //删除栈顶元素
2   {
3       PNode p;
4       if (IsNullStack_link(LinkStack top))           //判断栈是否为空
5           printf("it is empty stack!");
6       else
7       {
8           p = top -> next;                           //p 指向待删除结点
9           top -> next = p -> next;                   //修改栈顶指针
10          free(p);                                   //释放删除结点空间
11      }
12  }
```

3.3.5 取栈顶元素

先判断栈是否为空，如果栈不空，取栈顶元素的值，并返回，在操作过程中栈结构保持不变。具体实现见算法 3-10。

算法 3-10　取链栈的栈顶元素。

```
1  DataType Pop_seq_return(LinkStack top)        //删除栈顶元素
2  {
3      if (IsNullStack_link(LinkStack top))      //判断栈是否为空
4          printf("it is empty stack!");
5      else
6          return top->next->data;
7  }
```

3.4　栈的应用：进制转换

将十进制数和其他进制 d 进行转换是计算机实现计算的基本问题。下面考查 $d=8$ 时的转换情况，例如$(2017)_{10}=(3741)_8$，具体转换过程如表 3-2 所示。

表 3-2　进制转换

步　骤	*n*	*n* div 8	*n* mod 8
1	2017	252	1
2	252	31	4
3	31	3	7
4	3	0	3

通过表 3-2 可以看出，在转换过程中，最先计算得到的余数是结果的最低位，最后计算得到的余数是结果的最高位。也就是说结果的顺序是计算顺序的逆序输出，这正好符合栈的后进先出的特点，故使用栈来实现进制转换。具体实现见算法 3-11。

算法 3-11　十进制转换为八进制的算法。

```
1  #include "seqstack.h" //包含顺序栈头文件
2  void conversion(SeqStack ps,int n)  //将十进制 n 转换为八进制
3  {
4      while(n)
5      {
6          Push_seq(ps,n%8);
7          n/=8;
8      }
9      printf("转换为八进制后的结果: \n");
10     while(!IsNullStack_seq(ps))
11     {
12         n=Top_seq(ps);
13         printf("%d",n);
14         Pop_seq(ps);
15     }
16 }
```

如果将十进制转换为十六进制，由于在十六进制中涉及的余数是 10～15，需要用 A～F 表示，在输出结果时也需要输出 A～F，在算法中需要做特殊处理。具体过程见算法 3-12。

算法 3-12 十进制转换为十六进制的算法。

```
#include "seqstack.h" //这里使用顺序栈基本操作,故需要包含顺序栈头文件
void Hexconversion(SeqStack ps,int n)  //将十进制的 n 转换为十六进制
{
    while(n)
    {   int tmp = n % 16;
        switch(tmp)
        {
            case 10:tmp = 'A';break;
            case 11:tmp = 'B';break;
            case 12:tmp = 'C';break;
            case 13:tmp = 'D';break;
            case 14:tmp = 'E';break;
            case 15:tmp = 'F';break;
        }
        Push_seq(ps,tmp);
        n = n/16;
    }
    printf("转换为十六进制后的结果: \n");
    while(!IsNullStack_seq(ps))
    {
        n = Top_seq(ps);
        if(n<10) printf("%d",n);
        else printf("%c",n);
        Pop_seq(ps);
    }
}
```

3.5 栈的应用：括号匹配

在编译器对源程序进行语法检查的过程中，检查表达式的括号是否匹配是必需的一个步骤。在 C 语言中括号是可以嵌套使用的，对于一段源程序，假设允许有两种括号——圆括号和方括号，判断其中的括号在嵌套的情况下是否匹配。例如下面的两个例子。

(1) 正确的匹配情况：

([ ] ( )) 左括号和右括号相等，且匹配。

[([ ] [ ])] 左括号和右括号相等，且匹配。

(2) 错误的匹配情况：

[( ] 缺少右圆括号。

[( )) 最后一个右括号和第一个方括号不匹配。

[( ]) 右括号出现的顺序不正确。

下面考虑下列括号序列的匹配情况，为了更好地分析匹配过程，给括号标记了序号。

括号序列 [( ) [ ]]

括号序号 1 2 3 4 5 6

括号匹配分析：当第一个“[”出现时，它期望其对应的“]”出现，这时出现“(”，此时对于

表达式来说期望出现“)”,即“)”的期待程度比“]”的期待程度要高,也就是说期待程度高的先匹配,因此可以看出后来的左括号要先匹配,符合栈的操作特点。使用栈结构的具体实现见算法 3-13。算法思路如下:

(1) 设置标志位 flag=1。

(2) 顺序扫描表达式。

① 凡出现左括号,进栈。

② 凡出现右括号,首先检查栈是否为空。

a. 若栈空,表明右括号多了。

b. 若栈不空,和栈顶元素比较。

c. 若匹配,则左括号出栈。

d. 否则,设置标志位 flag=0,表示不匹配,退出。

(3) 表达式检验结束时:

① 若栈不空或 flag=0,匹配不成功。

② 若栈空,表明匹配成功。

算法 3-13 括号匹配算法。

```
#include "linkstack.h"                    //这里使用链栈基本操作,故包含链栈头文件
int BracketMatch( LinkStack top)          //括号匹配算法
{
    int flag = 1;
    char ch,temp;
    Push_link(top,'#');                   //栈底放#
    printf("请输入要判断的表达式,用#号结束: ");
    scanf_s("%c",&ch);
    while (ch!='#')
    {
        if (ch=='(')                      //左括号,压栈
            Push_link(top, ch);
        else
        {
            if (ch==')')                  //右括号,出栈
            {
                temp = Top_link(top);
                if (temp=='(')
                    Pop_link(top);
                else
                    flag = 0;break;
            }
        }
        scanf_s("%c",&ch);
    }
    if (!flag || Top_link(top)!='#')
    {
        printf("no\n");
        return 0;
    }
```

```
    else
    {
        printf("yes\n");
        return 1;
    }
}
```

3.6 栈的应用：栈与递归

递归思想贯穿在本书的各个章节中，由于数据结构本身或操作固有的递归特性，它们的操作可递归地描述，例如链表的定义、链表的建立、树和二叉树的定义和遍历、快速排序和堆排序等。有些问题，虽然问题本身没有明显的递归结构，但用递归求解比用迭代求解更简单，例如背包问题、Hanoi 塔问题、八皇后问题。递归算法具有简洁的优点，但是递归算法的空间消耗比较大，这和递归的深度有关。通过分析递归与栈的关系可以理解递归算法的缺点。

递归，简要地说就是函数调用函数自身。阶乘函数就是一个典型的递归例子。阶乘的定义如下：

$$n!=\begin{cases} n(n-1)! & n>1 \\ 1 & n=1 \end{cases}$$

在 n 阶乘的定义中包含了 $n-1$ 阶乘的定义，可以很容易地发现它具有以下两个特点：

(1) n 的阶乘由 n 和 $n-1$ 的阶乘确定，而 $n-1$ 的阶乘由 $n-1$ 和 $n-2$ 的阶乘定义，依此类推，这样就将一个问题转化为规模更小的问题

(2) 在 n 的阶乘定义中存在 $n=1$ 的特殊项，它不再依靠自身定义，这就是递归的出口，即递归终止条件，否则将会陷入无穷递归的死循环中。

根据这两个特点可以很容易地写出 $n!$ 的递归算法，见算法 3-14。

算法 3-14 阶乘算法。

```
int factorial(int n)
{
    int result = n;
    if(n == 1)
    return 1;
    return result = result * factorial(n - 1);
}
void main()
{
    int num;
    num = factorial(3);
}
```

在大部分操作系统中都会为每个运行的二进制程序分配栈空间。通过使用栈可以存储函数调用过程中的相关参数和返回地址等。一般情况下，当在一个函数的运行期间调用另一个函数时，在运行被调用函数之前，系统需要先完成 3 件事：①将所有的实参、返回地址

等信息传递给被调用函数保存；②为被调用函数的局部变量分配存储区；③将控制转移到被调用函数的入口。在从被调用函数返回调用函数之前，系统也应完成 3 件事：①保存被调函数的计算结果；②释放被调函数的数据区；③依照被调用函数保存的返回地址将控制转移到调用函数。当有多个函数构成嵌套调用时，按照“最后调用最先返回”的原则，系统运行期间所需要的数据依次存放在系统栈中。当调用一个子程序时创建一个新的活动记录（或栈帧结构），并通过入栈将其压入栈顶；每当从一个函数退出时，通过出栈删除栈顶活动记录。这里以计算 factorial(3)为例，观察记录递归过程中栈空间的变化，具体如表 3-3 所示。

表 3-3 阶乘算法递归过程

递归深度	实参	栈内容 （实参 n，result，行号地址，返回值） （栈底→栈顶）	说 明
0	main()	((3,11,))	主程序调用 factorial(3)，实参 3，主程序调用地址进栈
1	fact(3)	((3,11,) (3,3,6,))	调用 factorial(3)，实参 3，result=3，返回地址 6 进栈
2	fact(2)	((3,11,) (3,3,6,) (2,2,6,))	调用 factorial(2)，实参 2，result=2，返回地址 6 进栈
3	fact(1)	((3,11,) (3,3,6,) (2,2,6,) (1,1,6,))	调用 factorial(1)，实参 1，result=1，返回地址 6 进栈
	fact(1)	((3,11,) (3,3,6,) (2,2,6,) (1,1,6,**1**))	factorial(1)，返回值为 1
2	fact(2)	((3,11,) (3,3,6,) (2,2,6,**2**))	factorial(2)，返回值为 2
1	fact(1)	((3,11,) (3,3,6,**6**))	factorial(3)，返回值为 6
0	main()	((3,11,**6**))	返回主程序

从表 3-3 可以看出最后调用的最先返回，符合栈后进先出的操作特点，因此可以使用栈写出算法 3-14 的非递归实现，具体见算法 3-15。

算法 3-15 阶乘的非递归算法。

```
1  #include "linkstack.h"      //这里使用链栈,故包含链栈头文件
2  intfactorial_NR(int n)      //阶乘函数
3  {
4      int res;
5      LinkStack st;
6      st = SetNullStack_Link(n);
7      while(n > 0)
8      {
9          Push_link(st, n);
10         n = n - 1;
11     }
12     res = 1;
13     while (!IsNullStack_link(st))
14     {
```

```
15          res = res * Top_link(st);
16          printf("当前栈顶元素是: %d\n", Top_link(st));
17          Pop_link(st);
18      }
19      return(res);
20  }
```

算法扩展："聪明的学生"

一位教逻辑学的教授有 3 名非常善于推理且精于心算的学生 A、B 和 C。有一天，教授给他们 3 人出了一道题：教授在每个人的脑门上贴了一张纸条，并告诉他们每个人的纸条上都写了一个正整数，且某两个数的和等于第三个。于是，每个学生都能看见贴在另外两个同学脑门上的整数，却看不见自己脑门上的整数。

这时，教授先对学生 A 发问："你能猜出自己的数吗?" A 回答："不能。"

教授又转身问学生 B："你能猜出自己的数吗?" B 想了想，也回答："不能。"

教授又问学生 C 同样的问题，学生 C 思考了片刻后，摇了摇头说："不能。"

接着，教授又重新问学生 A 同样的问题，再问学生 B 和学生 C，……，经过若干轮后，当教授再次问某人时，此人露出了得意的笑容，把自己头上的数准确地说了出来。请问，已知 A、B 和 C 脑门上贴的数为 X_1、X_2、X_3，求教授至少需提问多少次，轮到回答问题的那个人才能猜出自己头上的数。

提示：采用递归方法解决，可参考本书配套的实验教材。

3.7 栈的应用：迷宫

这是一个陷入迷宫的老鼠如何找到出口的问题，要求输出老鼠探索出的从入口到出口的路径。老鼠希望尽快地找到出口走出迷宫。如果它到达一个死胡同，将原路返回到上一个位置，尝试新的路径。在每个位置上老鼠可以向 4 个方向运动，即上、下、左、右。无论离出口多远，它总是按照这样的顺序尝试，当到达一个死胡同之后，老鼠将进行"回溯"。这种探索路径的回溯过程是最后走的位置要最先返回，因此可以使用栈来保存走过的路径序列。

迷宫可以用一个二维数组 maze[m+2][n+2]表示，数组中的元素或者为 0，或者为 1。0 表示通路，1 表示墙，迷宫的四周可以设想为全 1，即为墙。设置入口为 maze[1][1]、出口为 maze[6][6]。为了避免检查是否到达了边界，在迷宫四周添加一条取值为 1 的边来表示障碍，如图 3-4 所示。

解决迷宫问题有两种策略，一种是深度优先策略，另外一种是广度优先策略。两种策略分别对应栈和队列做辅助数据结构。在本节采用深度优先策略，在 3.11 节采用广度优先策略解决迷宫问题。

在迷宫问题中要找到路径并输出需要解决 3 个问题。

(1) 从某一个坐标点(x,y)出发如何搜索其相邻位置(g,h)?

为了简化判断，假设迷宫四周都是墙，即在四周都赋值为 1 的一条边，这样对于任意的位置，与它相邻的位置有 8 个，如图 3-5 所示。

1	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
1	0	0	1	1	0	1	1	**1**
1	1	0	0	0	0	0	0	**1**
1	0	1	0	0	1	1	1	**1**
1	0	1	1	1	0	0	1	**1**
1	1	0	0	1	0	0	0	**1**
1	0	1	1	0	0	0	1	**1**
1	1	1	1	1	1	1	0	**1**
1	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**

图 3-4　迷宫图

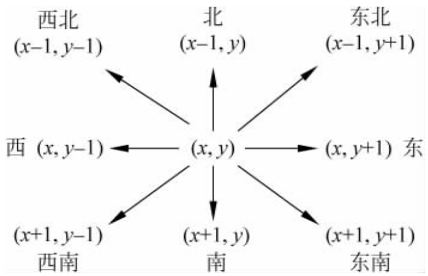


图 3-5　方向示意图

为了让计算机对 8 个位置按照一定的顺序搜索，不妨假设 8 个方向的顺序是从正东按照顺时针，将这 8 个方向的位置的坐标放到一个结构数组 direction[8][2]中，数组内容为：

```
direction[8][2] = { { 0, 1 }, { 1, 1 }, { 1, 0 }, { 1, -1 }, { 0, -1 }, { -1, -1 }, { -1, 0 },
{ -1, 1 } }
```

数组中给出了相邻位置(g,h)相对于当前位置(x,y)的增量，即：

```
g = x + direction[i][j]
h = y + direction[i][j]
```

假设从当前位置(3,5)向南出发，则：

```
g = x + direction[2][0] = 3 + 1 = 4
h = y + direction[2][1] = 5 + 0 = 5
```

(2) 如何记录探索过的路径？

由于采用了回溯方法，因此设计栈来存放探索过的路径，当不能向前继续探索时从栈中弹出元素。为了重复使用前面定义好的栈结构，在这里使用两个栈 linkStackX 和 linkStackY 分别存放行坐标和列坐标。

(3) 如何防止重复探索某位置？

通过设置标志位来识别，初始时各个位置的标志位 mark[i][j]=0，当探索到某位置后设置 mark[i][j]=1。

有了上述基本定义，迷宫算法的思路如下：

(1) 创建两个空栈 StackX 和 StackY。

(2) 将入口 entryX 和 entryY 分别压入栈 StackX 和 StackY 中。

(3) while(栈不空)

① 取栈顶元素，出栈，当前位置为栈顶元素。

② while(mov < 8)，即还存在探索的方向。

a. 按照顺时针依次探索各个位置(posX, posY)

b. 如果(posX, posY)是出口，输出路径，返回 1。

c. 如果(posX, posY)是没有走过的通路：

- 设置标志位 mark[posX][posY]=1。
- 当前位置进栈。

- 将(posX，posY)设置为当前位置。
- 设置 mov＝0。

d. 否则(如果(*X*，*Y*)是没有走过的通路),mov＋＋。

具体实现见算法 3-16,在算法 3-16 中使用了 3.3 节中定义的链栈。

算法 3-16 迷宫算法。

```
#include "linkstack.h"                          //包含链栈头文件
//迷宫深度遍历算法
int MazeDFS(int entryX, int entryY, int exitX, int exitY, Maze * maze)
{
    int direction[8][2] = { { 0, 1 }, { 1, 1 }, { 1, 0 }, { 1, -1 },
                            { 0, -1 }, { -1, -1 }, { -1, 0 }, { -1, 1 } };
    //用于两个栈,分别保存路径中的点
    LinkStack linkStackX = NULL;
    LinkStack linkStackY = NULL;
    int posX,posY;                              //临时变量,存放点坐标(x,y)
    int preposX,preposY;
    int **mark;                                 //标记二维数组,标记哪些点走过,不再重复走
    int i,j;                                    //循环变量
    int mov;                                    //移动的方向
    //给做标记的二维数组分配空间,并赋初值
    mark = (int**)malloc(sizeof(int*) * maze->size);
    for (i = 0;i < maze->size;i++)
        mark[i] = (int*)malloc(sizeof(int) * maze->size);
    //给所有元素设置初值
    for (i = 0;i < maze->size;i++)
    {
        for(j = 0;j < maze->size;j++)
            mark[i][j] = 0;
    }
    linkStackX = SetNullStack_Link();           //初始化栈
    linkStackY = SetNullStack_Link();           //初始化栈
    mark[entryX][entryY] = 1;                   //入口点设置标志位
    Push_link(linkStackX, entryX);              //入口点入栈
    Push_link(linkStackY, entryY);              //入口点入栈
      //如果栈不为空且还没有找到迷宫出口点
    while (!IsNullStack_link(linkStackX))
    {
        preposX = Top_link(linkStackX);
        preposY = Top_link(linkStackY);
        Pop_link(linkStackX);
        Pop_link(linkStackY);
        mov = 0;
        while(mov < 8)
        {
            posX = preposX + direction[mov][0];
            posY = preposY + direction[mov][1];
            if (posX == exitX && posY == exitY)         //到达终点
            {
```

```
44                  Push_link(linkStackX, preposX);       //出口点入栈
45                  Push_link(linkStackY, preposY);       //出口点入栈
46                  printf("深度搜索迷宫路径如下：\n");
47                  printf("%d %d\t", exitX, exitY);     //打印入口点
48                  while (!IsNullStack_link(linkStackX))//将路径逆序输出
49                  {
50                      posX = Top_link(linkStackX);      //取栈顶元素
51                      posY = Top_link(linkStackY);      //取栈顶元素
52                      Pop_link(linkStackX);             //出栈
53                      Pop_link(linkStackY);             //出栈
54                      printf("%d %d\t", posX, posY); //输出栈顶元素
55                  }  //end 48
56                  return 1;
57              }  //end 42
58              //还有路可以走通
59              if (maze->data[posX][posY] == 0 && mark[posX][posY] == 0)
60              {
61                  mark[posX][posY] = 1;
62                  Push_link(linkStackX, preposX);       //入栈
63                  Push_link(linkStackY, preposY);       //入栈
64                  preposX = posX;
65                  preposY = posY;
66                  mov = 0;  //已经往前走了,因此重新从 0 号方向开始搜索
67              }
68              else
69                  mov++;                                //换个方向试试
70          }  //end 38
71      } //end 31
72      return 0;
73  }
```

对于图 3-4 所示的迷宫，输出结果为：

(6,6)　(5,7)　(4,6)　(4,5)　(3,4)　(2,5)　(2,4)　(2,3)　(1,2)　(1,1)

具体搜索路径如图 3-6 所示，其中虚线表示探索过的位置，实线表示路径。

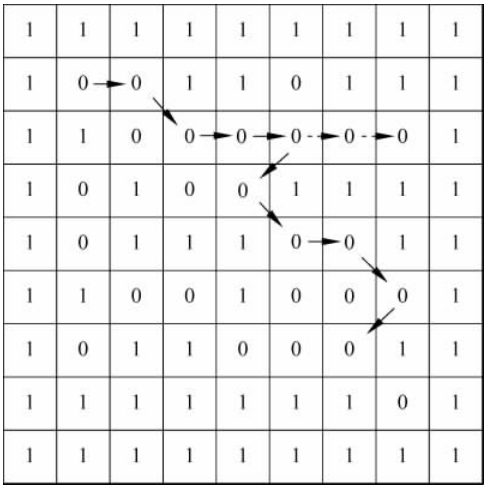


图 3-6　迷宫搜索路径

通过采用回溯算法可以得到一条从入口到出口的路径，但并不能保证这条路径是最短的，那么如何找到从出口到入口的最短路径呢？3.11 节给出了使用队列找到最短路径的方法。

3.8 栈的应用：表达式求值

表达式求值是程序设计语言编译中必须处理的问题，实现方法是栈应用的一个典型实例。中缀表达式就是在中学学习的表达式，操作符号在两个操作数中间。计算表达式的值按照先乘除后加减，对于括号，先算括号里面再算括号外面。括号在这里的作用就是避免产生歧义。例如 31 *（5－22）＋70。那么能不能去掉这些额外的符号呢？逻辑学家波兰发明了不需要括号的记号法，就是前缀表示和后缀表示形式，举例如下。

（1）前缀表达式——波兰式(Polish Notation，PN)：

```
+ * 31 -5 22 70
```

（2）中缀表达式：

```
31 * (5 - 22) + 70
```

（3）后缀表达式——逆波兰式(Reverse Polish Notation，RPN)：

```
31 5 22 - * 70 +
```

其中，后缀表达式是计算机最方便处理的形式。下面先分析后缀表达式的计算过程，再分析如何将熟悉的中缀表达式转换为后缀表达式。

后缀表达式用栈很容易计算结果，算法思路如下：

（1）顺序扫描后缀表达式

① 如果是操作数，压入栈中。

② 如果是操作符，从栈中弹出两个操作数进行计算，把结果再压入栈。

（2）扫描结束时，栈顶元素就是表达式的值。

后缀表达式求值算法的具体实现见算法 3-17，在算法 3-17 中使用了链栈。

算法 3-17 后缀表达式求值算法。

```
1  #include "linkstack.h"                    //包含链栈头文件
2  //后缀表达式求值,返回计算结果
3  int evaluatePostfix(char * expression)
4  {
5      int i,len;
6      len = strlen(expression);
7      LinkStack stack = SetNullStack_Link();
8      for (i = 0; i < len; i++)             //从左到右逐个判断后缀表达式中的字符
9      {
10         if (isdigit(expression[i]))       //如果是数字,将它入栈
11         {
12             Push_link(stack, expression[i] - '0');
13         }
```

```
14          else //如果是操作符,取两个栈顶元素作为操作数进行运算,将结果入栈
15          {
16              int val1 = Top_link(stack);
17              Pop_link(stack);
18              int val2 = Top_link(stack);
19              Pop_link(stack);
20              switch (expression[i])
21              {
22              case '+': Push_link(stack, val2 + val1); break;
23              case '-': Push_link(stack, val2 - val1); break;
24              case '*': Push_link(stack, val2 * val1); break;
25              case '/': Push_link(stack, val2/val1); break;
26              }
27          }
28      }
29      return Top_link(stack);
30  }
```

例如,后缀表达式“6　2　/　3－4　2　*　＋”的计算过程如表 3-4 所示。

表 3-4　后缀表达式计算过程

步骤	待处理的后缀表达式（头→尾）	栈（顶→底）	进行的计算
1	6　2　/　3－4　2　*　＋		
2	2　/　3－4　2　*　＋	6	
3	/　3－4　2　*　＋	2　6	
4	3－4　2　*　＋	3	6/2
5	－4　2　*　＋	3　3	
6	4　2　*　＋	0	3－3
7	2　*　＋	4　0	
8	*　＋	2　4　0	
9	＋	8　0	2*4
10		8	0＋8

下面介绍如何将中缀表达式转换为后缀表达式,从前面的例子可以看出,在后缀表达式中所有整数出现的次序与它们在中缀表达式中出现的相对次序是一致的,差别仅在于运算符按照实际执行的次序放到对应的运算分量后面。算法思路如下:

(1) 从左至右依次扫描中缀表达式。

① 如果是操作数,直接输出。

② 如果是“(”,入栈中。

③ 如果是“)”弹出栈顶元素并放入后缀表达式中,反复执行直到栈顶元素为“(”时为止,表明这一层括号内的操作处理完毕。

④ 如果是操作符,将该操作符和操作符栈顶元素比较。

- 如果**高于**栈顶元素的优先级,将它压入栈中。
- 如果**低于**栈顶元素的优先级,取出栈顶元素放入后缀表达式,并弹出该栈顶元素,反复执行直到栈顶元素的优先级低于当前操作符的优先级。

（2）重复上述步骤直到遇到中缀表达式的结束“#”，弹出栈中的所有元素并放入后缀表达式中，算法结束。

从上述算法可以看出，在转换过程中需要对操作符进行优先级的比较，操作符的优先级关系如表 3-5 所示。中缀表达式转换为后缀表达式的具体过程见算法 3-18，在算法 3-18 中使用了链栈。

表 3-5　操作符优先级关系表

优先级关系	+	−	*	/	(	)	#
+	>	>	<	<	<	>	>
−	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	>
)	>	>	>	>	=	>	>
#	<	<	<	<	<	<	=

算法 3-18　中缀表达式转换为后缀表达式。

```
#include "linkstack.h" //包含链栈头文件
//获得某个运算符的优先级，加减返回 1，乘除返回 2，其他返回 - 1
int precedenceOf(char ch)
{
    switch (ch)
    {
    case '+':
    case '-':
        return 1;
    case '*':
    case '/':
        return 2;
    }
    return -1;
}

//中缀表达式转后缀表达式，返回值为存有后缀表达式的字符串
char *infixToPostfix(char* infixExp)
{
 int i, len, k = 0;
 char *postfixExp;
 len = strlen(infixExp);
 postfixExp = (char *)malloc(sizeof(char) * len);
 LinkStack stack = SetNullStack_Link();
 for (i = 0; i < len; i++)                    //从左到右依次判断中缀表达式中的各个字符
 {
     if (isdigit(infixExp[i]))                //如果是数字，直接添加到后缀表达式中
     {
         postfixExp[k] = infixExp[i];
         k++;
```

```
        }
        else if (infixExp[i] == '(')          //如果是左括号,将它入栈
        {
            Push_link(stack, infixExp[i]);
        }
        else if (infixExp[i] == ')')
          //如果是右括号,将栈中的操作符出栈,直到遇到左括号
        {
            while (!IsNullStack_link(stack) && Top_link(stack)!='(')
            {
                postfixExp[k] = Top_link(stack);
                k++;
                Pop_link(stack);
            }
            Pop_link(stack);
        }
        else
        {
            //出栈,直到栈为空或该操作符的优先级高于栈顶操作符的优先级
            while(!IsNullStack_link(stack) && precedenceOf(infixExp[i])
                                        <= precedenceOf(Top_link(stack)))
            {
                postfixExp[k] = Top_link(stack);
                Pop_link(stack);
                k++;
            }
            Push_link(stack, infixExp[i]);  //将该操作符入栈
        }
    }
    //将栈中元素全部出栈并添加到后缀表达式中
    while (!IsNullStack_link(stack))
    {
        postfixExp[k] = Top_link(stack);
        Pop_link(stack);
        k++;
    }
    postfixExp[k] = '\0';
    return postfixExp;
}
```

例如,将中缀表达式“31＊(5－22)＋70”转换为后缀表达式,具体过程如表 3-6 所示。

表 3-6　中缀表达式转换为后缀表达式的过程

步骤	待处理的中缀表达式（头→尾）	栈（顶→底）	已经输出的部分后缀表达式（头→尾）
1	31＊(5－22)＋70		
2	＊(5－22)＋70		31
3	(5－22)＋70	＊	31
4	5－22)＋70	(＊	31

续表

步骤	待处理的中缀表达式（头→尾）	栈（顶→底）	已经输出的部分后缀表达式（头→尾）
5	－22)＋70	(＊	31 5
6	22)＋70	－(＊	31 5
7	)＋70	－(＊	31 5 22
8	＋70	＊	31 5 22 －
9	70	＋	31 5 22 － ＊
10		＋	31 5 22 － ＊ 70
11			31 5 22 － ＊ 70 ＋

3.9 循环队列

队列采用顺序存储时会存在假溢出的问题。表 3-7 给出了一个实例。假设分配的顺序队列空间为 8 个。在步骤 1～10 中依次进行了相关的操作，在第 10 步入队后，元素 40 占据顺序队列的第 8 个空间，这样在第 11 步不能进行入队的操作。但是在第 4 步和第 5 步进行出队操作，顺序表的第 1 个和第 2 个空间已经腾空，由于队列操作的受限性，这时不能完成入队的操作，这种现象称为**“假溢出”**。

表 3-7 循环队列举例

步骤	操作	队列（队头→队尾）							
1	enQueue(qu,5)	05							
2	enQueue(qu,10)	05	10						
3	enQueue(qu,15)	05	10	15					
4	deQueue(qu)		10	15					
5	deQueue(qu)			15					
6	enQueue(qu,20)			15	20				
7	enQueue(qu,25)			15	20	25			
8	enQueue(qu,30)			15	20	25	30		
9	enQueue(qu,35)			15	20	25	30	35	
10	enQueue(qu,40)			15	20	25	30	35	40
11	enQueue(qu,45)								

这里给出采用循环队列解决假溢出的一种方法，如图 3-7 所示。该方法是将顺序队列设想为一个环形空间。在队列处于如图 3-7(d)所示的状态时，如果再进队元素 45，则从 0 的位置开始，因为 0 的位置空，所以 45 入队 0 的位置，如图 3-7(e)所示。这时如果再进队 50，50 入队 1 的位置，此时队头指针 f 和队尾指针 r 重合，如图 3-7(f)所示。注意这种情况队列已经满了，但是由于队列空的状态也是队头指针 f 和队尾指针 r 重合，因此需要采取措施区分队列空和队列满。采取的方式就是浪费一个空间，如图 3-7(e)所示表明队列满，即队尾 r 还差一个位置和队头重合。

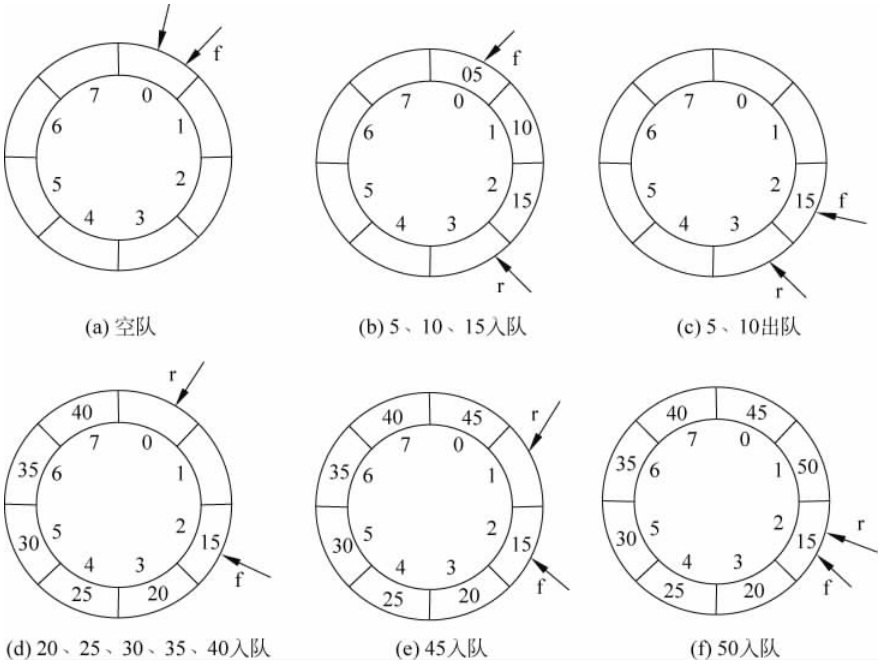


图 3-7　循环队列示意图

循环队列的类型定义如下：

```
typedef int DataType;
 struct Queue
 {
  int Max;                //队列空间最大值
  int f,r;                //队头和队尾指针
  DataType * elem;        //队列元素空间
 };
typedef struct Queue * SeqQueue;
```

3.9.1　创建空队列

创建空队列首先申请 struct Queue 结构体大小的空间，申请成功后再申请一个预先定义的数组空间，并将循环队列的队头和队尾设置为零。具体实现见算法 3-19。

算法 3-19　创建空队列。

```
SeqQueue SetNullQueue_Seq(int m)
{
    SeqQueue squeue;
    squeue = (SeqQueue)malloc(sizeof(struct Queue));
    if (squeue == NULL)
    {
        printf("Alloc failure\n");
        return NULL;
    }
```

```
    squeue->elem=(int*)malloc(sizeof(DataType)*m);
    if (squeue->elem!=NULL)
    {
        squeue->Max=m;          //设置循环队列最大值空间
        squeue->f=0;            //队头初值
        squeue->r=0;            //队尾初值
        return squeue;
    }
}
```

3.9.2 判断队列是否为空

检查队头和队尾指针是否相等，相等为空队列，返回 1，否则返回 0。具体实现见算法 3-20。

算法 3-20 判断队列是否为空。

```
int IsNullQueue_seq(SeqQueue squeue)     //判断队列是否为空
{
    return (squeue->f==squeue->r);
}
```

3.9.3 入队

首先检查队列是否满，如果不满，则在队尾插入元素，并修改队尾指针。具体实现见算法 3-21。

算法 3-21 入队。

```
void EnQueue_seq(SeqQueue squeue, DataType x)
{
    if ((squeue->r+1) % squeue->Max==squeue->f)          //判断队列是否满
        printf("It is FULL Queue!");
    else{
        squeue->elem[squeue->r]=x;                       //将元素 x 插入队尾
        squeue->r=(squeue->r+1) % (squeue->Max);         //队尾指针加 1
    }
}
```

3.9.4 出队

首先检查队列是否为空，若队列非空，则删除队头元素，修改队头指针。具体实现见算法 3-22。

算法 3-22 出队。

```
void DeQueue_seq(SeqQueue squeue)
{
    if (IsNullQueue_seq(squeue))                         //判断队列是否为空
        printf("It is empty queue!\n");
```

```
5       else
6           squeue->f = (squeue->f + 1) % (squeue->Max);          //队头指针加 1
7   }
```

3.9.5 取队头元素

检查队列是否为空，若队列非空，返回队头元素。具体实现见算法 3-23。

算法 3-23 取队头元素。

```
1   DataType FrontQueue_seq(SeqQueue squeue)
2   {
3       if (squeue->f == squeue->r)          //判断队列是否为空
4           printf("It is empty queue!\n");
5       else
6           return(squeue->elem[squeue->f]);
7   }
```

3.10 链队列

采用链式存储的队列称为链队列，队头指针指向链队列的第一个结点，队尾指针指向最后一个结点，如图 3-8 所示。

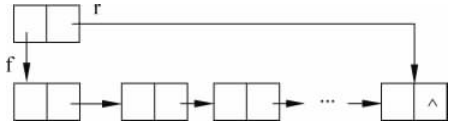


图 3-8 链队列示意图

链队列的类型定义如下：

```
1   typedef int DataType;
2   struct Node
3   {
4       DataType data;
5       struct Node * link;
6   };
7   typedef struct Node  * PNode;
8   struct Queue
9   {
10      PNode f;
11      PNode r;
12  };
13  typedef struct Queue  * LinkQueue;
```

3.10.1 创建空队列

创建空队列，申请 struct Queue 结构体空间，并设置队头和队尾指针为空，然后返回指

向该结点的指针。具体实现见算法 3-24。

算法 3-24 创建空队列。

```
1  LinkQueue  SetNullQueue_Link()  //创建空队列
2  {
3      LinkQueue lqueue;
4      lqueue = (LinkQueue)malloc(sizeof(struct Queue));
5      if (lqueue!= NULL)
6      {
7          lqueue -> f = NULL;
8          lqueue -> r = NULL;
9      }
10     else
11         printf("Alloc failure! \n");
12     return  lqueue;
13 }
```

3.10.2 判断队列是否为空

检查队头指针,队列为空返回 1,否则返回 0。具体实现见算法 3-25。

算法 3-25 判断队列是否为空。

```
1  int IsNullQueue_Link(LinkQueue lqueue)  //判断队列是否为空
2  {
3      return (lqueue -> f == NULL);
4  }
```

3.10.3 入队

申请结点,如果申请成功,为结点的数据域和指针域赋值,如果是空队列,对入队的第一个结点进行特殊处理,即队头和队尾都指向该结点;如果不是空队列,则在队尾插入。具体实现见算法 3-26。

算法 3-26 入队。

```
1  void EnQueue_link(LinkQueue lqueue, DataType x)        //入队操作
2  {
3      PNode  p;
4      p = (PNode)malloc(sizeof(struct Node));            //申请结点空间
5      if (p == NULL)
6          printf("Alloc failure!");
7      else{
8              p -> data = x;                             //数据域赋值
9              p -> link = NULL;                          //指针域赋值
10             if (lqueue -> f == NULL)                   //空队列的特殊处理
11             {
12                 lqueue -> f = p;
13                 lqueue -> r = p;
14             }
```

```
15                else
16              {
17                  lqueue->r->link = p;              //插入队尾
18                  lqueue->r = p;                    //修改队尾指针
19              }
20          }
21  }
```

3.10.4 出队

检查队列是否为空，若队列非空，修改队头指针，并释放队头结点空间。具体实现见算法 3-27。

算法 3-27 出队。

```
1   void DeQueue_link(LinkQueue lqueue)               //出队
2   {
3       struct Node  * p;
4       if (lqueue->f == NULL)                        //判断队列是否为空
5           printf( "It is empty queue!\n ");
6       else
7       {
8           p = lqueue->f;                            //p 指向队头结点，以方便后面的释放
9           lqueue->f = lqueue->f->link;              //修改队头指针
10          free(p);                                  //释放结点空间
11      }
    }
```

3.10.5 取队头元素

检查队列是否为空，如果非空，返回队头元素。具体实现见算法 3-28。

算法 3-28 取队头元素。

```
1   DataType  FrontQueue_link(LinkQueue lqueue)     //取队头元素
2   {
3       if (lqueue->f == NULL)                      //判断队列是否为空
4       {
5           printf("It is empty queue!\n");
6           return 0;
7       }
8       else
9           return (lqueue->f->data);               //返回队头结点数据域
10  }
```

3.11 队列的应用：迷宫

在 3.7 节中使用栈采用深度策略找到了一条从入口到出口的路径，但是怎样找到一条最短路径呢？本节使用队列采用广度策略找到一条从入口到出口的最短路径。

迷宫的表示和搜索 8 个方向的顺序同前面的定义，迷宫算法思路如下：

(1) 创建两个空栈 linkQueueX 和 linkQueueY。

(2) 将入口 entryX 和 entryY 分别压入栈 linkQueueX 和 linkQueueY 中。

(3) while(队列不空)

① 取队头元素，出队。

② for(mov=0;mov < 8;mov++)，即还存在可以探索的相邻方向。

a. 按照顺时针依次探索各个位置(X, Y)。

b. 如果(posX, posY)是出口，则输出路径，返回。

c. 如果(posX, posY)是没有走过的通路：

- 设置标志位 mark[posX][posY]=1。
- 当前位置入队。
- 记录前驱位置，方便输出路径。

在实现中设置 preposMarkX 和 preposMarkX 数组，用来存放迷宫行走过程中的前驱结点，以方便在找到出口时能够逆序输出迷宫路径。具体实现见算法 3-29，在算法 3-29 中使用了链队列。

算法 3-29 迷宫算法。

```
1  #include "LinkQueue.h"                          //包含链队列头文件
2  //迷宫广度遍历算法
3  int MazeBSF(int entryX, int entryY, int exitX, int exitY, Maze * maze)
4  {
5      int direction[8][2] = { { 0, 1 }, { 1, 1 }, { 1, 0 }, { 1, -1 },
6                              { 0, -1 }, { -1, -1 }, { -1, 0 }, { -1, 1 } };
7      //用于两个队列,分别保存等待扩展的点
8      LinkQueue linkQueueX = NULL;
9      LinkQueue linkQueueY = NULL;
10     int posX,posY;                              //临时变量,存放点坐标(x,y)
11     int preposX,preposY;
12     int **preposMarkX;                          //记录迷宫行走过程中的前驱 x 值
13     int **preposMarkY;                          //记录迷宫行走过程中的前驱 y 值
14     int**mark;                                  //标记二维数组,标记哪些点走过,不再重复走
15     int i,j,mov;
16     //给存放前驱 x 值的数组分配空间
17     preposMarkX = (int **)malloc(sizeof(int *) * maze->size);
18     for (i = 0; i<maze->size; i++)
19     {
20         preposMarkX[i] = (int *)malloc(sizeof(int) * maze->size);
21     }
22     //给存放前驱 y 值的数组分配空间
23     preposMarkY = (int **)malloc(sizeof(int *) * maze->size);
24     for (i = 0; i<maze->size; i++)
25     {
26         preposMarkY[i] = (int *)malloc(sizeof(int) * maze->size);
27     }
28     //给做标记的二维数组分配空间,并赋初值
29     mark = (int **)malloc(sizeof(int *) * maze->size);
```

```
    for (i = 0; i < maze -> size; i++)
    {
        mark[i] = (int * )malloc(sizeof(int) * maze -> size);
    }
    for (i = 0; i < maze -> size; i++)                    //给所有元素设置初值
    {
        for (j = 0; j < maze -> size; j++)
        {
            preposMarkX[i][j] = -1;
            preposMarkY[i][j] = -1;
            mark[i][j] = 0;
        }
    }
    linkQueueX = SetNullQueue_Link();                     //创建空队列
    linkQueueY = SetNullQueue_Link();                     //创建空队列
    EnQueue_link(linkQueueX, entryX);                     //迷宫入口点入队
    EnQueue_link(linkQueueY, entryY);                     //迷宫入口点入队
    mark[entryX][entryY] = 1;                             //入口点设置标志位
    //如果队列不为空且还没有找到迷宫出口点
    while (!IsNullQueue_Link(linkQueueX))
    {
        preposX = FrontQueue_link(linkQueueX);            //取队头
        DeQueue_link(linkQueueX);                         //出队
        preposY = FrontQueue_link(linkQueueY);            //取队头
        DeQueue_link(linkQueueY);                         //出队
        //将与当前点相邻接且满足一定条件的点放入队列
        for (mov = 0; mov < 8; mov++)
        {
            posX = preposX + direction[mov][0];
            posY = preposY + direction[mov][1];
            if (posX == exitX && posY == exitY)           //到达出口点
            {
                preposMarkX[posX][posY] = preposX;
                preposMarkY[posX][posY] = preposY;
                printf("广度搜索迷宫路径如下: \n%d %d\t", posX, posY);
                //将路径逆序输出
                while (!(posX == entryX && posY == entryY))
                {
                    //继续往前寻找前驱
                    preposX = preposMarkX[posX][posY];
                    preposY = preposMarkY[posX][posY];
                    posX = preposX;
                    posY = preposY;
                    printf("%d %d\t", posX, posY);
                }  //end 66
                return 1;
            }  //end 60
            //如果能走,且没有被扩展过
            if (maze -> data[posX][posY] == 0 && mark[posX][posY] == 0)
            {
```

```
80                      EnQueue_link(linkQueueX, posX);          //入队扩展
81                      EnQueue_link(linkQueueY, posY);
82                      mark[posX][posY] = 1;                    //做标记
83                      preposMarkX[posX][posY] = preposX;       //记录前驱
84                      preposMarkY[posX][posY] = preposY;
85                  }  //end 78
86              }  //end 56
87          }  //end 49
88          return 0;
89      }
```

对于图 3-4 所示的迷宫，输出结果为：

(6,6) (5,6) (4,5) (3,4) (2,3) (1,2) (1,1)

搜索路径过程中的位置及前驱位置的状态变化如表 3-8 所示，加粗表示输出迷宫路径中通过前驱 preposMarkX、preposMarkY 逆序输出路径的过程。搜索路径示意图如图 3-9 所示，其中虚线表示探索过的位置，实线表示路径。

表 3-8 位置以及其前驱位置的状态变化

posX	**1**	2	**2**	3	3	2	**3**	4	2	1	**4**	5	2	4	**5**	5	5	6	2	5	**6**	…
posY	**2**	2	**3**	3	1	4	**4**	1	5	5	**5**	2	6	6	**6**	5	3	1	7	7	**6**	…
preposMarkX	**1**	1	**1**	2	2	2	**2**	3	3	2	2	**3**	4	2	1	4	**4**	4	5	5	2	4	**5**	…
preposMarkY	**1**	1	**2**	2	2	3	**3**	3	1	4	4	**4**	1	5	5	5	**5**	5	2	2	6	6	**6**	…

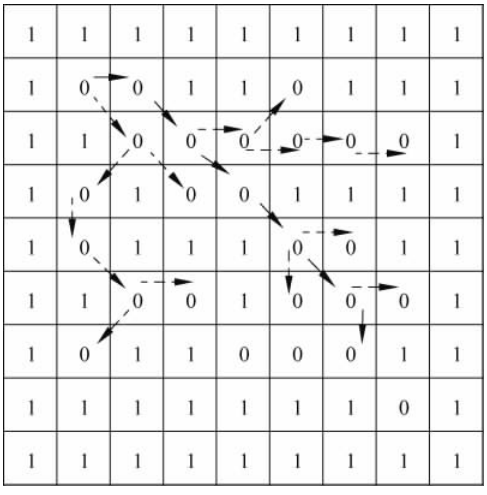


图 3-9 迷宫搜索路径

3.12 队列的应用：农夫过河

一个农夫带了一只狼、一只羊和一棵白菜来到河边，他要把它们从河的南岸带到河的北岸。现在仅有一只很小的船，农夫最多只能带其中之一，请问农夫怎样才能将它们安全地运

到北岸？要求给出一种可行方案。

要找到可行的方案并输出，需要解决以下 3 个问题。

(1) 位置与状态：为了表示它们的位置，采用二进制位来区分南岸和北岸，0 表示在南岸，1 表示在北岸。用 4 个二进制位分别表示农夫、狼、白菜和羊 4 个物品所在的位置。例如 1110 表示农夫、狼和羊在北岸，白菜在南岸。农夫过河问题的初始状态为 0000，结束状态为 1111。

(2) 安全状态判断：在算法求解过程中需要判断状态是否安全。例如 1110 是安全状态，1000 是不安全状态。分析所有的状态，其中不安全状态有两个，一是羊和白菜在同一岸但农夫没有和它们在一起，另外一个是狼和羊在同一岸但农夫没有和它们在一起。其他的都是安全状态。

(3) 中间状态记录：为了方便地记录中间的状态，设置一维数据 path[16]。初始时 path[j]=−1(j=0,…,15)。如果状态 j 由状态 i 得到，则更新 path[j]=i。这样当到达结束状态 1111 时可以通过 path[]数组反向回推到初始状态 0000，中间的状态则是农夫过河问题的一个解。

有了以上基本定义，农夫过河的算法的具体思路如下。

(1) 初始状态 0000 入队。

(2) 当队列不空且没有到达结束状态 1111 时循环以下操作：

① 队头状态出队。

② 按照农夫一个人走、农夫分别带上 3 个走循环以下操作：

- 如果农夫和它们在同一岸，则计算新的状态。
- 如果新状态是安全的并且是没有处理过的，则更新 path[ ]，并将新状态入队。

③ 当状态为 1111 时逆向输出 path[]数组。

需要注意的是状态能否入队，要判断以下 3 条，满足任何一条都不能入队。

(1) 不可能：通过判断是否在同一岸；

(2) 不安全：通过安全函数判断；

(3) 处理过：记录处理过的状态。

农夫过河问题的具体实现见算法 3-30，在算法 3-30 中使用了循环队列。

算法 3-30 农夫过河问题。

```
1  #include "SeqQueue.h"                          //包含顺序队列头文件
2  int FarmerOnRight(int status)                  //判断当前状态下农夫的位置
3  {
4      return (0!=(status & 0x08));
5  }
6  int WolfOnRight(int status)                    //判断当前状态下狼的位置
7  {
8      return (0!=(status & 0x04));
9  }
10 int CabbageOnRight(int status)                 //判断当前状态下白菜的位置
11 {
12     return (0!=(status & 0x02));
13 }
```

```
int GoatOnRight(int status)                    //判断当前状态下羊是否在南岸
{
    return (0!= (status & 0x01));
}
int IsSafe(int status)                         //判断当前状态是否安全
{
    if((GoatOnRight(status) == CabbageOnRight(status))&&
                (GoatOnRight(status)!= FarmerOnRight(status)))
          return (0);                          //羊吃白菜
    if((GoatOnRight(status) == WolfOnRight(status))&&
                 (GoatOnRight(status)!= FarmerOnRight(status)))
          return (0);                          //狼吃羊
    return (1);                                //其他状态是安全的
}
void FarmerRiver()                             //农夫过河算法
{
    int i, movers, nowstatus, newstatus;
        int status[16];                        //用于记录已考虑的状态路径
    SeqQueue moveTo;                           //用于记录可以安全到达的中间状态
    moveTo = SetNullQueue_seq(20);             //创建空队列
    EnQueue_seq(moveTo, 0x00);                 //初始状态时所有物品在南岸,初始状态入队
        for (i = 0; i < 16; i++)               //数组 status 初始化为 - 1
            status[i] = - 1;
        status[0] = 0;
    while (!IsNullQueue_seq(moveTo) && (status[15] == - 1))
      //队列非空且没有到达结束状态
        {
        nowstatus = FrontQueue_seq(moveTo); //取队头状态为当前状态
        DeQueue_seq(moveTo);
          for (movers = 1; movers <= 8; movers << = 1)  //遍历 3 个要移动物品
            //考虑各种物品移动
        if ((0!= (nowstatus & 0x08)) == (0!= (nowstatus &movers)))
          //农夫与移动的物品在同一侧
          {
            newstatus = nowstatus ^ (0x08 | movers);    //计算新状态
            //如果新状态是安全的且之前没有出现过
              if (IsSafe(newstatus) && (status[newstatus] == - 1))
              {
                status[newstatus] = nowstatus;          //记录新状态
                EnQueue_seq(moveTo, newstatus);         //新状态入队
              }
          }
        }
        //输出经过的状态路径
        if (status[15]!= - 1)                           //到达最终状态
        {
          printf("The reverse path is : \n");
        for (nowstatus = 15; nowstatus >= 0; nowstatus = status[nowstatus])
          {
            printf("The nowstatus is : %d\n", nowstatus);
```

```
64                 if (nowstatus == 0)
65                     exit(0);
66             }
67         }
68         else
69             printf("No solution.\n");                    //问题无解
70     }
```

3.13　双端队列

双端队列是同时具有队列和栈的性质的数据结构，即可以在头部和尾部插入和删除元素的数据结构。在 STL 中有双端队列容器，有兴趣的读者也可以自己实现双端队列。下面给出双端队列的一个简单应用——滑动最小值。

问题描述：给定一个长度为 n 的数列 $a_0, a_1, a_2, \cdots, a_{n-1}$ 和一个整数 k，求数列 $b_i = \min\{a_i, a_{i+1}, \cdots, a_{i+k-1}\}$。

限制条件：$1 \leqslant k \leqslant n \leqslant 106$，$0 \leqslant a_i \leqslant 109$

输入：

$n=5$

$k=3$

$a=\{1,3,5,4,2\}$

输出：

$b=\{1,3,2\}$

双端队列开始为空，队列中存放数组 a 的下标，然后通过维护双端队列得到最小值。

算法思路：

（1）把 0 到 $k-1$ 依次加入队列。当加入 i 时，若双端队列的末尾的值 j 满足 $a_j \geqslant a_i$，则不断从末尾取出，直到双端队列为空或者 $a_j < a_i$，之后在末尾加入 i。

（2）在 $k-1$ 都加入双端队列后，查看双端队列头部的值 j，那么 $b_0 = a_j$，如果 $j=0$，由于之后的计算中不再需要，因而从头部删除。

（3）继续计算 b_i，在双端队列的末尾加入 k，进入 a 重复执行。

具体实现见算法 3-31，在算法 3-31 中使用了 STL 中的双端队列。

算法 3-31　滑动最小值问题的算法。

```
1   #include<iostream>
2   #include<deque>  //STL中的双端队列
3   using namespace std;
4   int main()
5   {
6       deque<int> d;
7       int n, k, a[100], b[100];
8       cout<<"请输入 n 和 k: ";
9       cin>>n>>k;
10      cout<<"请输入数组 a 的元素: ";
```

```
    for (int i = 0; i < n; i++)
    {
        cin >> a[i];
    }
    int count = 0;
    d.push_back(0);
    for (int i = 1; i < n; i++)
    {
        while (!d.empty() && a[d.back()] >= a[i])
        {
            d.pop_back();
        }
        d.push_back(i);
        if (i - k + 1 >= 0)
        {
            b[i - k + 1] = a[d.front()];
            count++;
        }
        if (d.front() == i - k + 1)
        {
            d.pop_front();
        }
    }
    for (int i = 0; i < count; ++i)
    {
        printf("%d ", b[i]);
    }
    return 1;
}
```

习题

3-1 有 4 个元素(a、b、c、d)按照先后次序压入栈中,栈混洗有多少种可能?把它们全部列出来。

3-2 设栈 S 和队列 Q 的初始状态为空,元素 1、2、3、4、5 依次通过栈 S,一个元素出栈后即刻进入队列 Q。若这 6 个元素出队列的顺序是 2,4,3,6,5,1,则栈的容量至少应该多大?

3-3 将表达式(1+5)×(2+12/4)+6 转换为后缀表达式,并画出栈中内容变化的过程。

3-4 设两个栈 S_1 和 S_2 共享同一数组 $a[0,1\cdots MAX]$,为了最大限度地利用数组空间,两个栈应该如何共享?给出入栈、出栈、判断栈满和判断栈空的算法。

3-5 已知递归函数 $F(m)$,其中 div 表示整除。

$$F(m)=\begin{cases}1 & \text{当 } m=0 \text{ 时}\\ mF(m \text{ div } 2) & \text{当 } m>0 \text{ 时}\end{cases}$$

(1) 写出 $F(m)$的递归算法。

(2) 写出 $F(m)$的非递归算法。

3-6 设用循环链表来表示队列,只设置一个指针指向队尾结点,不设置头指针,要求实现创建空队列、判断队列是否为空、入队、出队和取队头元素等操作,并在主程序中进行测试。

3-7 假设用一个数组 $a[0,1\cdots MAX]$来表示循环队列,队列只设置队头指针 front,不设置队尾指针 rear,通过设置计数器 count 来表示队列中结点的个数。编写创建空队列、判断空队列、入队和出队等算法。

3-8 八皇后问题是一个古老且著名的问题,该问题由国际西洋棋棋手马克斯·贝瑟尔于 1848 年提出:在 8×8 格的国际象棋棋盘上摆放 8 个皇后,使其不能互相攻击,即任意两个皇后都不能处于同一行、同一列或同一斜线上,这样的一种格局称为一个解。请用回溯法和递归方法解决八皇后问题。

3-9 分别采用栈的深度搜索策略和队列的广度搜索策略来解决迷宫问题,给定不同形态的迷宫进行测试和比较,比较方面包括时间消耗、迷宫大小、迷宫形态、应用需求等,从而说明两种方式的适用性。

3-10 应用题:实现一个简易的表达式求值系统。从键盘输入带有括号的中缀表达式,要求首先判断括号是否匹配,如果括号匹配,表达式合法,再将中缀表达式转换为后缀表达式,并计算和输出表达式的值。对常见输入错误进行判断,例如括号错误、非法字符、运算符号错误、除数为零等情况。可以使用自己实现的栈或使用 STL 中的栈。

3-11 应用题:用队列模拟银行业务。设银行有 s 个不同业务的服务窗口,顾客来到银行通过取票得到其对应业务的服务号,s 个窗口根据顾客到来的顺序提供“先进先出”的服务。请编写算法模拟该过程。