第 3 章

栈和队列

从数据结构的角度来说，栈和队列也是线性表，不过是两种特殊的线性表。栈只允许在表的一端进行插入或删除操作，而队列只允许在表的一端进行插入操作，而在另一端进行删除操作。因而，栈和队列也可以被称作操作受限的线性表。通过本章的学习，读者应该掌握栈和队列的逻辑结构和存储结构，以及栈和队列的基本运算和实现算法。

3.1 栈

3.1.1 栈的定义及其运算

1．栈的定义

栈（stack）是一种只允许在一端进行插入和删除操作的线性表，它是一种操作受限的线性表。在表中允许进行插入和删除的一端称为栈顶（top），另一端称为栈底（bottom）。栈的插入操作通常称为入栈或进栈（push），而栈的删除操作则称为出栈或退栈（pop）。当栈中无数据元素时，称为空栈。

根据栈的定义可知，栈顶元素总是最后入栈的，因而是最先出栈；栈底元素总是最先入栈的，因而也是最后出栈。这种表是按照后进先出（LIFO，Last In First Out）的原则组织数据的，因此，栈也被称为后进先出的线性表。

图 3-1 是一个栈的示意图，通常用指针 top 指向栈顶的位置，用指针 bottom 指向栈底。栈顶指针 top 动态反映栈的当前位置。

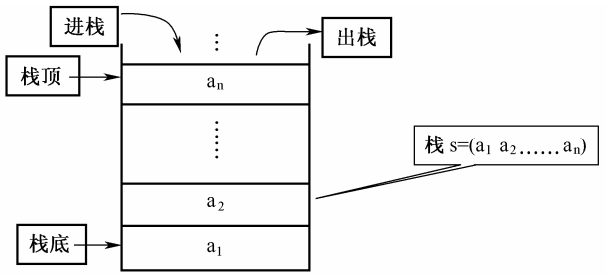


图 3-1 栈的示意图

例 3-1 对于一个栈，给出输入项 A、B、C，如果输入项序列由 ABC 组成，试给出所有可能的输出序列和不可能产生的输出序列。

解：

所有可能的输出序列为：

A 进 A 出 B 进 B 出 C 进 C 出	ABC
A 进 A 出 B 进 C 进 C 出 B 出	ACB
A 进 B 进 B 出 A 出 C 进 C 出	BAC
A 进 B 进 B 出 C 进 C 出 A 出	BCA
A 进 B 进 C 进 C 出 B 出 A 出	CBA

不可能产生的输出序列为 CAB。

2．栈的抽象定义及运算

栈的抽象数据类型定义及各种运算如下：

ADT Stack {

数据对象：D={ a_i | a_i∈ElemSet，i=1，2，...，n，n≥0}

数据关系：R={ $<a_{i-1}$，$a_i>| a_{i-1}$，$a_i \in D$，i=2，...，n }

约定 a_n 端为栈顶，a_1 端为栈底。

基本操作：

InitStack(&S)

操作结果：构造一个空栈 S。

DestroyStack(&S)

初始条件：栈 S 已存在。

操作结果：栈 S 被销毁。

ClearStack(&S)

初始条件：栈 S 已存在。

操作结果：将 S 清为空栈。

StackEmpty(S)

初始条件：栈 S 已存在。

操作结果：若栈 S 为空栈，则返回 true，否则返回 false。

StackLength(S)

初始条件：栈 S 已存在。

操作结果：返回栈 S 中的元素个数，即栈的长度。

GetTop(S,&e)

初始条件：栈 S 已存在且非空。

操作结果：用 e 返回 S 的栈顶元素。

Push(&S,e)

初始条件：栈 S 已存在。

操作结果：压入元素 e 为新的栈顶元素。

Pop(&S,&e)

初始条件：栈 S 已存在且非空。

操作结果：弹出 S 的栈顶元素，并用 e 返回其值。

StackTraverse(S,visit( ))

初始条件：栈 S 已存在且非空，visit( )为元素的访问函数。

操作结果：依次对 S 的每个元素调用函数 visit( )，

一旦 visit( )失败，则操作失败。

} ADT Stack

栈是一种特殊的线性表，可以采用顺序存储结构存储，也可以使用链式存储结构存储。

3.1.2　顺序栈

1. 顺序栈的存储结构

与第 2 章讨论的一般的顺序存储结构的线性表一样，利用一组地址连续的存储单元依次存放自

栈底到栈顶的数据元素，这种形式的栈也称为顺序栈。因此，可以使用一维数组作为栈的顺序存储空间。设指针 top 指向栈顶元素的当前位置，以数组小下标的一端作为栈底，通常以 top=0 时为空栈，在元素进栈时指针 top 不断地加 1，当 top 等于数组的最大下标值时则栈满。

顺序栈类 Stack 中所涉及的数据成员如下：

（1）存放堆栈元素的数组：

T stacklist[MaxStackSize];

/* 使用数组存放栈元素，栈的规模必须小于或等于数组的规模，当栈的规模等于数组的规模时，就不能再向栈中插入元素 */

（2）栈顶所在数组元素的下标：

int top;

/* top 变量是栈顶指针（下标），开始时，栈空，top=EMPTY，其中 EMPTY 是某个表示空值的常数，例如 EMPTY=-1*/

类 Stack 涉及的成员函数如下：

```
Stack (void)                    //栈的初始化
void Push (const T& item)       //入栈操作
T Pop (void)                    //出栈操作
void ClearStack (void)          //栈置空操作
T Peek (void) const             //取栈顶元素操作
int StackEmpty (void) const     //判栈空操作
int StackFull (void) const      //判栈满操作
```

2. 顺序栈类的实现

设数组 stacklist 是一个顺序栈，栈的最大容量 MaxStackSize=4，初始 top=-1，表示栈空，此时出栈则下溢（underflow）。若 top=MaxStackSize-1，表示栈满，此时入栈则上溢（overflow）。上溢是一种出错状态，应该设法避免；下溢常常用来作为控制转移的条件。顺序栈中数据元素与栈顶指针的变化如图 3-2 所示。变量 x 用来存放入栈或出栈的数据。

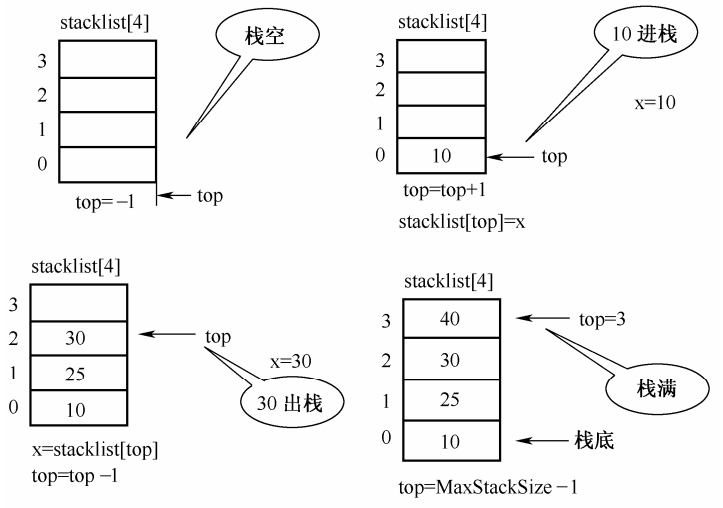


图 3-2 顺序栈中数据元素与栈顶指针的变化

顺序栈类的具体实现描述如下：

```
template < class T >
//栈的初始化
Stack<T>::Stack (void) : top(-1)

//入栈操作
算法 Push(item)
Push1 [堆栈已满?]
IF top=MaxStackSize -1 THEN
( PRINT "STACK OVERFLOW!".    RETURN. ).
Push2 [栈顶指针加 1]
top←top+1.    stacklist[ top ]←item ▌

//出栈操作
算法 Pop(.temp)
Pop1 [堆栈已空?]
IF top=-1
THEN(
PRINT "Attempt to pop an empty stack! ".
RETURN. ).
Pop2 [保存栈顶元素,改变栈顶位置]
temp←stacklist[ top ].
top←top-1 ▌

//取栈顶元素操作
算法 Peek(.temp)
Peek1 [堆栈已空?]
IF top=-1
THEN(
PRINT "Attempt to peek an empty stack!".
RETURN. ).
Peek2 [读取栈顶元素]
temp←stacklist[ top ] ▌

//判栈空操作
算法 StackEmpty(.i )
SE1 [堆栈已空？]
IF top=-1 THEN i←1 ELSE i←0 . ▌

//判栈满操作
算法 StackFull(.i )
SF1 [堆栈已满？]
IF top=MaxStackSize-1 THEN i←1. ELSE i←0 . ▌
```

```
//栈置空操作
算法 ClearStack
CS1 [重置栈顶]
top← -1 .  RETUEN . ▌
```

3.1.3 多栈共享邻接空间

在计算机系统软件中，各种高级语言的编译系统都离不开栈的使用。有时，在一个程序设计中需要使用多个同一类型的栈，这时可能会产生一个栈空间过小，容量发生溢出，而另一个栈空间过大，造成大量存储单元浪费的现象。为了充分利用各个栈的存储空间，这时可以使多个栈共享存储单元，即给多个栈分配一个足够大的存储空间，让多个栈实现存储空间优势互补。这就是栈的共享邻接空间。

1. 双向栈在一维数组中的实现

栈的共享中最常见的是两栈的共享。假设两个栈共享一维数组 stack[MAXNUM]，则可以利用栈的“栈底位置不变，栈顶位置动态变化”的特性，两个栈底分别为-1 和 MAXNUM，而它们的栈顶都向中间延伸。因此，只要整个数组 stack[MAXNUM]未被占满，则无论哪个栈的入栈都不会发生上溢。

C++语言定义的这种两栈共享邻接空间的结构如下：

```
typedef struct {
Elemtype stack[MAXNUM];
int lefttop;        //左栈栈顶位置指示器
int righttop;       //右栈栈顶位置指示器
} dupsqstack;
```

两个栈共享邻接空间的示意图如图 3-3 所示。左栈入栈时，栈顶指针加 1，右栈入栈时，栈顶指针减 1。

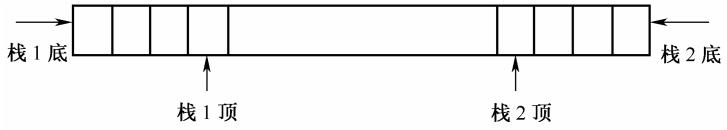


图 3-3　两个栈共享邻接空间

为了识别左右栈，必须另外设定标志：

```
char status;
status='L';  //左栈
status='R';  //右栈
```

在进行栈操作时，需指定栈号，status='L'为左栈，status='R'为右栈；判断栈满的条件为 s->lefttop+1==s->righttop。

2. 共享栈的基本操作

（1）初始化操作。

```
//共享栈的初始化
int initDupStack(dupsqstack *s)
{//创建两个共享邻接空间的空栈由指针 S 指出
```

```
if ((s=(dupsqstack*)malloc(sizeof(dupsqstack)))==NULL) return false;
s->lefttop=-1;
s->righttop=MAXNUM;
return true;
}
```

（2）入栈操作。

```
//共享栈的入栈操作
int pushDupStack(dupsqstack *s,char status,Elemtype x)
{//把数据元素 x 压入左栈（status='L'）或右栈（status='R'）
if(s->lefttop+1==s->righttop) return false;       //栈满
if(status='L')    s->stack[++s->lefttop]=x;       //左栈进栈
else if(status='R')
s->stack[--s->lefttop]=x;                         //右栈进栈
        else
return false;                                     //参数错误
return true;
}
```

（3）出栈操作。

```
//共享栈的出栈操作
Elemtype popDupStack(dupsqstack *s,char status)
{//从左栈（status='L'）或右栈（status='R'）退出栈顶元素
if(status=='L')
{ if (s->lefttop<0)
    return NULL;                          //左栈为空
else return (s->stack[s->lefttop--]);     //左栈出栈
}
else if(status=='R')
    { if (s->righttop>MAXNUM-1)
          return NULL;                    //右栈为空
else return (s->stack[s->righttop++]);    //右栈出栈
}
else return NULL;                         //参数错误
}
```

3.1.4　链栈

1. 链栈

栈也可以采用链式存储结构表示，如图 3-4 所示，这种结构的栈简称为链栈。在一个链栈中，栈底就是链表的最后 1 个结点，而栈顶总是链表的第 1 个结点。因此，新入栈的元素即为链表的新的第 1 个结点，只要系统还有存储空间，就不会有栈满的情况发生。一个链栈可由栈顶指针 top 唯一确定，当 top 为 NULL 时，是一个空栈。

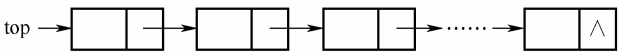


图 3-4　链栈

链栈的运算是受限的单链表，插入和删除操作仅限制在表头位置上进行。由于只能在链表头部进行操作，故链表没有必要像单链表那样附加头结点。栈顶指针就是链表的头指针。

链栈的结点定义如下：

```
typedef struct node
{ ElemType data;
  struct node *next;
} LinkStack;
```

此时只需要对顺序栈的基本操作做两处改动，便可实现链栈的基本操作。其一是，初始化时不需要“MaxStackSize”参数，因为它不需要事先分配空间；其二是，在进行入栈操作时不需要顾忌栈的空间是否已经被填满。

链栈的进栈算法：

```
LinkStack *PUSHLSTACK(LinkStack *top, ElemType x)
{ LinkStack *p;
  p=(LinkStack*)malloc(sizeof(LinkStack));
  p->data=x; p->next=top;
  return p;
}
```

链栈的出栈算法：

```
LinkStack *POPLSTACK(LinkStack *top, ElemType *datap)
{ LinkStack *p;
  if (top==NULL)
     {cout <<"under flow\n";
      return NULL;}
  else
    { *datap=top->data;
      p=top;
      top=top->next;
      free(p);
      return top;
    }
}
```

总之，链栈的特点如下：

（1）链栈无栈满问题，空间可扩充。

（2）插入与删除仅在栈顶处执行。

（3）链栈的栈顶在链头。

（4）适合于多栈操作。

2. 多个链栈的操作

在程序中同时使用两个以上的栈时，使用顺序栈共用邻接空间很不方便，但若使用多个单链栈时，操作极为方便。下面介绍多个链栈的操作。

可将多个单链栈的栈顶指针放在一个一维数组 slStacktype *top[M]中，让 top[0],top[1],…,top[i],…,top[M-1]指向 M 个不同的链栈，操作时只需确定栈号 *i*，然后以 top[i]为栈顶指针进行栈操作，就可实现各种操作。

（1）入栈操作。

```
//多个链栈的入栈操作
int pushDupLs(slStacktype *top[M],int i,Elemtype x)
{//将元素 x 压入链栈 top[i]中
slStacktype *p;
if((p=(slStacktype *)malloc(sizeof(slStacktype)))==NULL)
return false;
//申请到一个结点后入栈
p->data=x; p->next=top[i]; top[i]=p; return true;
}
```

（2）出栈操作。

```
//多个链栈的出栈操作
Elemtype popDupLs(slStacktype *top[M],int i)
{//从链栈 top[i]中删除栈顶元素
slStacktype *p;
Elemtype x;
if (top[i]==NULL) return NULL; //空栈
p=top[i]; top[i]=top[i]->next;
x=p->data;free(p);return x;
}
```

在上面的两个算法中，当指定栈号 i（$0 \leqslant i \leqslant M-1$）时，则只对第 i 个链栈操作，不会影响其他链栈。

3.1.5　栈的应用举例

1. 表达式的计算

表达式求值是程序设计语言编译中的一个最基本问题。它的实现方法是栈的一个典型的应用实例。

在计算机中，任何一个表达式都是由操作数（operand）、运算符（operator）和界限符（delimiter）组成的。其中操作数可以是常数，也可以是变量或常量的标识符；运算符可以是算术运算符、关系运算符和逻辑运算符；界限符为左右括号和标识表达式结束的结束符。在本节中，仅讨论简单算术表达式的求值问题。在这种表达式中只含加、减、乘、除四则运算，所有的运算对象均为单变量。表达式的结束符为“#”。

为了叙述简洁，在此仅讨论只含二元运算符的算术表达式。可将这种表达式定义为：

表达式=操作数 运算符 操作数

操作数=简单变量 | 表达式

简单变量=标识符 | 无符号整数

由于算术运算的规则是先乘除后加减、先左后右和先括号内后括号外，因此，表达式的运算不能只按其中运算符出现的先后次序进行。

在计算机中，为了求解表达式的值，首先要将它转换成另一种形式。对这种二元表达式，有 3 种不同的表示方法。

假设表达式为： Exp=S1 OP S2

其中，S1 为第一操作数，OP 为运算符，S2 为第二操作数。则称

OP S1 S2 为该表达式的前缀表示法（简称前缀式）。

S1 OP S2 为该表达式的中缀表示法（简称中缀式）。

S1 S2 OP 为该表达式的后缀表示法（简称后缀式）。

可见，表达式表示法按照运算符所在不同位置命名。

例 3-2 用三种不同的标识方法表示表达式

Exp=a*b+(c-d/e)*f

解：

前缀式： +*a b*-c/d e f

中缀式： a*b+c-d/e*f

后缀式： a b*c d e /-f *+

综合比较它们之间的关系可得下列结论：

（1）三式中的操作数之间的相对次序相同。

（2）三式中的运算符之间的相对次序不同。

（3）中缀式丢失了括号信息，致使运算的次序不确定。

（4）前缀式的运算规则为：连续出现的两个操作数和在它们之前且紧靠它们的运算符构成一个最小表达式。

（5）后缀式的运算规则为：运算符在式中出现的顺序恰为表达式的运算顺序，每个运算符和在它之前出现且紧靠它的两个操作数构成一个最小表达式。

本书仅就“如何按后缀式进行运算”和“如何将原表达式转换成后缀式”两个问题进行讨论。

（1）如何按后缀式进行运算。

按后缀式进行运算的求值算法如下：“先找运算符，后找操作数。”

运算过程如下：对后缀式从左向右“扫描”，遇见操作数则暂时保存，遇见运算符即可进行运算；此时参加运算的两个操作数应该是在遇见运算符之前刚刚遇到的两个操作数，并且先出现的是第一操作数，后出现的是第二操作数。

例 3-3 对于表达式 Exp=a*b+(c-d/e)*f，设计利用堆栈实现按后缀式“ab*cde/-f*+”进行求值运算的过程。

解：对后缀式 ab*cde/-f*+求值过程如图 3-5 所示。

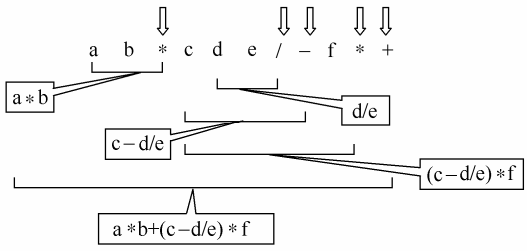


图 3-5 对后缀式 ab*cde/-f*+的求值过程

由此可见，在运算过程中保存操作数的数据结构应该是一个栈。

利用堆栈实现按后缀式进行求值运算的过程如下：

1）从左到右读入后缀表达式。

2）若读入的是一个操作数，就将它压入堆栈。

3）若读入的是一个运算符（op），就从堆栈中弹出两个操作数，设为 x 和 y，计算表达式 x op y 的值，并将计算结果压入堆栈。

4）对整个后缀表达式的读入结束时，栈顶元素就是计算结果。若表达式未输入完，转 1）。

例 3-4　利用堆栈实现按后缀式 35*684/-7*+#进行运算。

解：利用堆栈实现按后缀式 35*684/-7*+#进行运算的过程如图 3-6 所示。

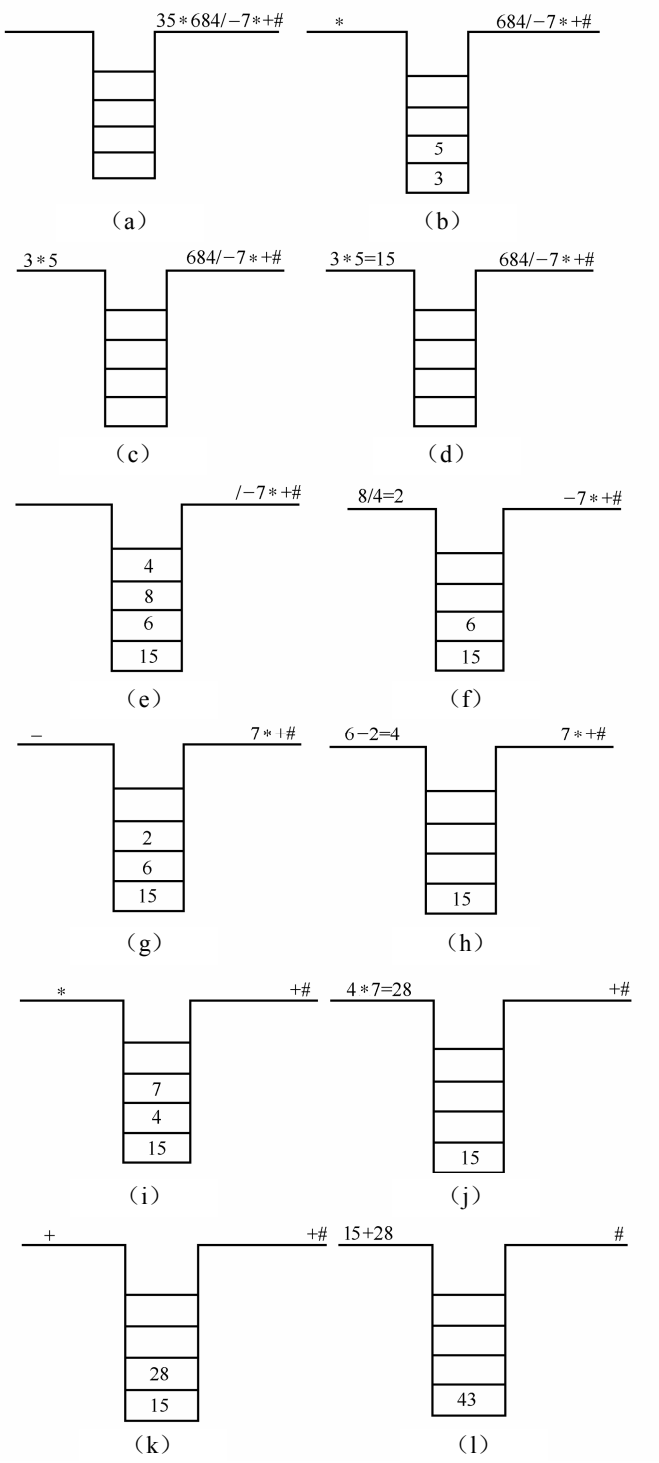


图 3-6　利用堆栈实现按后缀式 35*684/-7*+#进行运算过程

利用堆栈实现对后缀式进行运算的算法实现如下：

```
#include<iostream.h>
#include<stdlib.h>

#define StackSize 15

typedef struct {
int *base;   // 存储空间基址
int top;   // 栈顶指针
int stacksize;   // 允许的最大存储空间以元素为单位
} SeqStack;

void initstack (SeqStack *S,int maxsize)
{
// 构造一个最大存储容量为 maxsize 的空栈 S
if (maxsize==0)
maxsize=StackSize;
S->base=new int[maxsize];
if (!S->base) exit(1);   // 存储分配失败
S->stacksize=maxsize;
S->top=0;   // 空栈中元素个数为 0
}

bool stacktop (SeqStack *S, int e)
{
// 若栈不空，则用 e 返回 S 的栈顶元素，并返回 true；否则返回 false
if (S->top==0) return false;
e=*(S->base + S->top-1);    // 返回非空栈中的栈顶元素
return true;
}

bool push (SeqStack *S, int e)
{
// 若栈的存储空间不满，则插入元素 e 为新的栈顶元素
// 并返回 true；否则返回 false
if (S->top==S->stacksize)    // 栈已满，无法进行插入
return false;
*(S->base + S->top)=e;    // 插入新的元素
++S->top;    // 栈顶指针后移
return true;
}

bool pop (SeqStack *S, int e)
{
```

```
// 若栈不空，则删除 S 的栈顶元素，用 e 返回其值
// 并返回 true；否则返回 false
if (S->top==0) return false;
e=*(S->base + S->top-1);    // 返回非空栈中栈顶元素
--S->top;    // 栈顶指针前移
return true;
}

//GetValue_NiBoLan
int GetValue_NiBoLan(char *str)    //对逆波兰式求值
{ char *p;
   int a,b,r;
   SeqStack *s;
   s=new SeqStack;
   p=str;
   initstack(s,StackSize); //s 为操作数栈
   while(*p!='#')
   { if('0'<=(int)*p && (int)*p<='9')
          push(s,*p);
      else
      {
         pop(s,a);pop(s,b);
         a=a-48;
         b=b-48;
         switch(*p)
               {
                     case '+':     r=a + b ;
                                   break;
                     case '-':     r=a - b;
                                   break;
                     case '*':     r=a * b;
                                   break;
                     case '/':     r=a / b;
                                   break;
               }
               push(s,r);
      }   //else
      p++;
   }   //while

   pop(s,r);
   return r;
}   //GetValue_NiBoLan
```

（2）由原表达式转换为后缀式。

先分析在不同表达式的后缀式中运算符出现的次序有什么不同，例如：

表达式一：a*b/c*d-e+f

后缀式：ab*c/d*e-f+

表达式二：a+b*c-d/e*f

后缀式：abc*+de/f*-

表达式一中运算符出现的先后次序恰为运算的顺序，在其后缀式中运算符出现的次序和原表达式相同。

表达式二中运算符出现的先后次序不是它的运算顺序。按照算术运算规则，表达式二中先出现的“加法”应在它之后出现的“乘法”完成之后进行，且应该在后面出现的“减法”之前进行；同理，后面一个“乘法”应后于在它之前出现的“除法”进行，而先于在它之前的“减法”进行。

可见，每个运算符的运算次序要由它之后的一个运算符来定，优先数高的运算符领先于优先数低的运算符。将每个运算符的优先级用“优先数”表示如下：

运算符：# （ + - × / **

优先数：-1 0 1 1 2 2 3

其中“**”为乘幂运算，“#”为结束符。容易看出，优先数反映了算术运算中的优先关系，即优先数高的运算符应优先于优先数低的运算符进行运算。也就是说，对原表达式中出现的每一个运算符是否即刻进行运算取决于在它后面出现的运算符的优先数，如果它的优先数高于或等于后面运算符的优先数，则它的运算先进行，否则就得等待在它之后出现的所有优先数高于它的运算都完成之后再进行。

显然，用以保存运算符的存储结构应该是一个栈，从栈底到栈顶的运算符的优先数是从低到高的，因此它们运算的先后应是从栈顶到栈底的。

因此，从原表达式求得后缀式的规则为：

1）设立运算符栈。

2）设表达式的结束符为“#”，预设运算符栈的栈底为“#”。

3）若当前字符是操作数，则直接发送给后缀式。

4）若当前字符为运算符且优先数大于栈顶运算符则进栈，否则退出，栈顶运算符发送给后缀式。

5）若当前字符是结束符，则自栈顶至栈底依次将栈中所有运算符发送给后缀式。

6）“(”对它前后的运算符起隔离作用，则若当前运算符为“(”时进栈。

7）“)”可视为自相应左括号开始的表达式的结束符，则从栈顶起依次退出，栈顶运算符发送给后缀式，直至栈顶字符为“(”止。

例 3-5 运用堆栈从原表达式 a*(b*(c+d/e)-f)#求得后缀式。

解：从原表达式 a*(b*(c+d/e)-f)#求得后缀式的过程如图 3-7 所示。

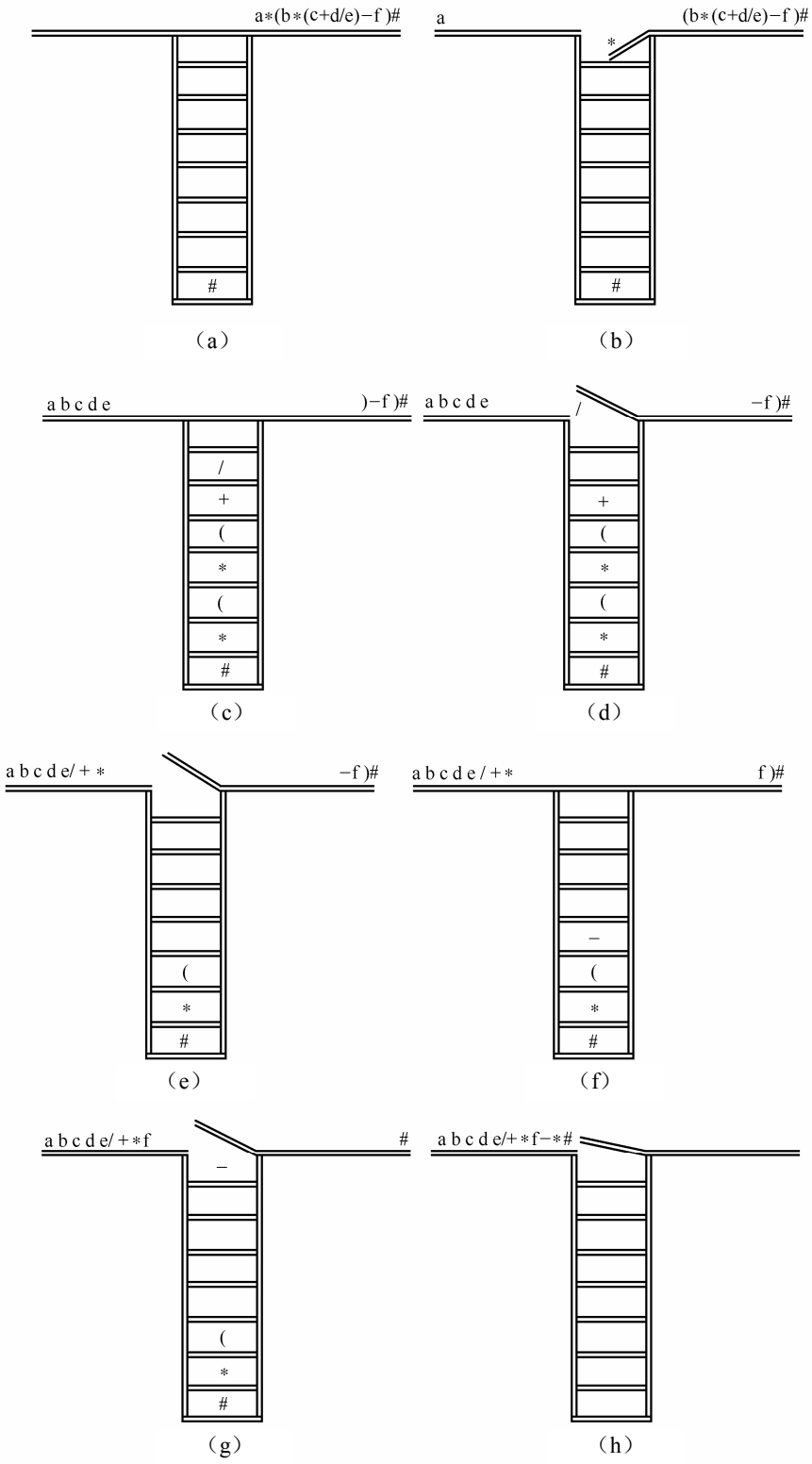


图 3-7　从原表达式 a*(b*(c+d/e)-f)#求得后缀式的过程

从表达式字符串 exp 求得其相应的后缀式的算法如下：

```
void transform(char suffix[ ], char exp[ ] )
{
    // 从表达式字符串 exp 求得其相应的后缀式 suffix
  char *p,ch,c;
  bool bl;
  SeqStack *s;
  initstack(s,StackSize); push(s,'#');
  p=exp; ch=*p;
  bl=stacktop(s,c);
  while (!bl) {
  if (!IN(ch, OP)) pass( suffix, ch);   // 直接发送给后缀式
  else {
      switch (ch)
 {
         case '(' : push(s, ch); break;
         case ')' : {     pop(s, c);
                          while (c!='(' )
                         { pass( suffix, c); pop(s, c) ;}
                          break; }
defult : {            bl=stacktop(s,c);
                        while(!c && ( precede(c,ch)))
                       { pass( suffix, c); pop(s, c); }
                        if ( ch!='#' ) push( s, ch);
                        break;
                    } // defult
      } // switch
    } // else
  if ( ch!='#' ) { p++; ch=*p; }
  } // while
} // transform
/*OP 为运算符的集合，若 ch 是运算符，则函数 IN(ch,OP) 的值为"TRUE"。
  函数 pass(suffix,ch) 的功能是将字符 ch 复制到数组 suffix 中。
  若 c(栈顶运算符)的优先数大于或等于 ch(当前运算符)的优先数，则函数 precede(c,ch)值为"TRUE"。*/
```

2. 函数的嵌套调用

在程序设计中，经常会碰到多个函数的嵌套调用。在程序的执行过程中，调用函数和被调用函数之间的链接和信息交换是由编译程序通过栈来实施的。

当一个函数在运行期间调用另一个函数时，在运行该被调用函数之前，需先完成三件事：

（1）将所有的实参数、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。

而从被调用函数返回调用函数之前，应该完成如下三件事：

（1）保存被调用函数的计算结果。

（2）释放被调用函数的数据区。

（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。

当多个函数嵌套调用时，由于函数的运行规则是后被调用者先返回，因此各函数占有的存储管理应实行“栈式管理”。

以函数的嵌套调用为例，假设主函数 M 调用函数 A，函数 A 又调用函数 B，函数 B 又调用函数 C。在执行时，每当遇到调用语句时，就要暂停原来正在执行的函数，转而执行被调用函数，等被调用函数执行完后，再回过头来执行原来的函数。

为了确保程序安全无误地执行，每当执行到调用语句时，就把原来函数的“现场”保存起来，等调用返回时再恢复现场继续执行。例如，如图 3-8 所示，在函数 B 运行期间主函数 M 和函数 A 占用的存储区都不能被覆盖，反之，当函数 B 运行结束，它所占用的存储区便可释放，同时因为程序控制转移到函数 A，当前程序访问的数据自然就是函数 A 占用的存储区了。

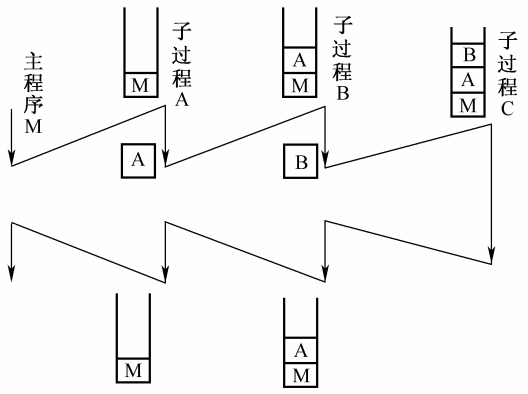


图 3-8　函数的嵌套调用

如果一个函数的函数体中出现调用自身的语句，这个函数就是一个递归（recursion）函数。递归调用与嵌套调用的唯一差别是递归调用的调用者和被调用者是同一个函数。递归函数的执行过程与函数嵌套调用一样，也要使用栈。用于递归调用的栈常称为递归调用栈，简称递归栈。它的作用是：

（1）将递归调用时的实参数和函数返回地址传递给下一层执行的递归函数。

（2）保存本层的参数和局部变量，以便从下一层返回时重新使用它们。

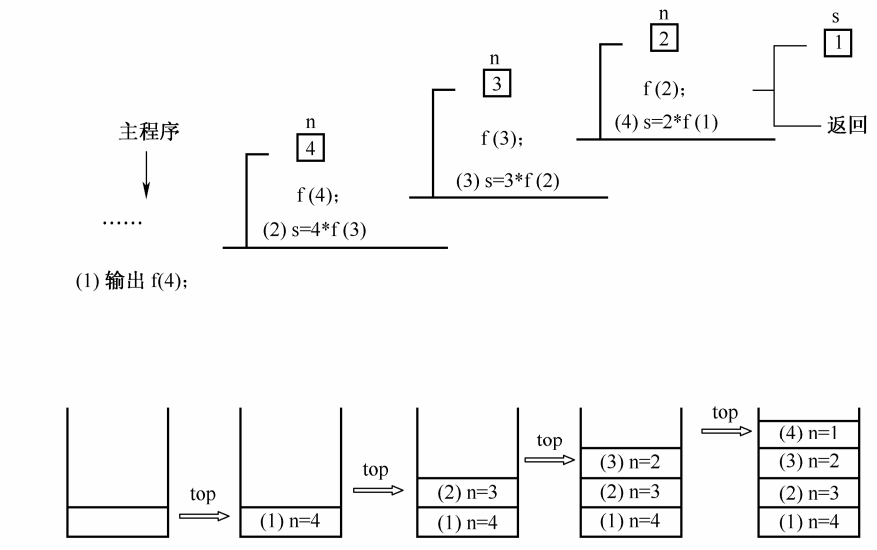
调用递归函数的主函数称为第 0 层，则从主函数调用递归函数被称为进入递归函数的第 1 层，从递归函数的第 i 层递归调用本函数被称为进入递归函数的第 i+1 层。显然，当递归函数执行到第 i 层时，从第 1 层到第 i−1 层的数据都必须被保存下来，以便一层一层退回时继续使用。递归函数执行过程中每一层所占用的内存数据区合起来就是一个递归栈。

例 3-6　写一递归函数求 n!，并以 n=4 为例描述递归调用过程。

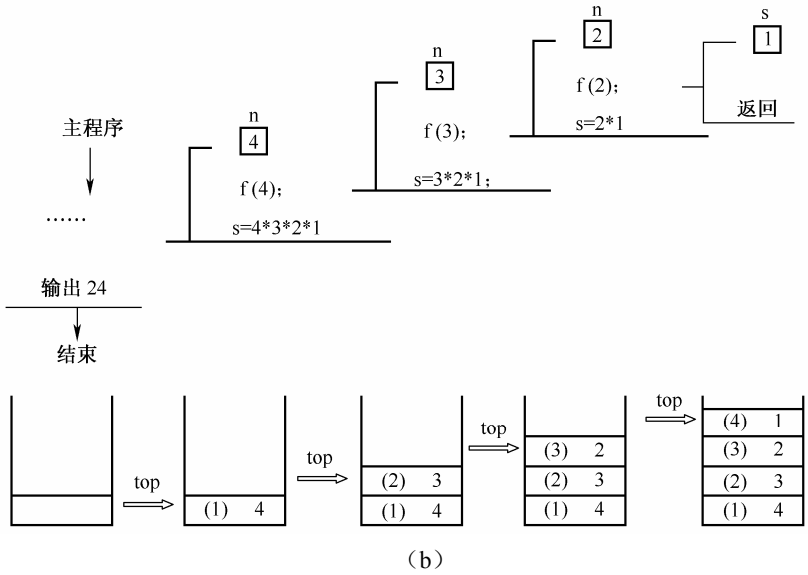
解：

```
long fac(int n)
{
    long f;
    if (n==0)
       f=1;
    else
       f=n*fac(n-1);
    return (f);
}
```

递归调用过程如图 3-9 所示。



（a）



（b）

图 3-9　递归调用过程

3.2　队列（queue）

在日常生活中队列很常见，如排队购物。排队体现了“先来先服务”（即“先进先出”）的原则。

队列在计算机系统中的应用也非常广泛，例如操作系统中的作业排队。在多道程序运行的计算机系统中，可以同时有多个作业运行，它们的运算结果都需要通过通道输出，若通道尚未完成输出，则后来的作业应排队等待，每当通道完成输出时，则从队列的队头退出作业输出操作，而凡是申请

该通道输出的作业都从队尾进入该队列。

计算机系统中输入输出缓冲区的结构也是队列的应用。在计算机系统中经常会遇到两个设备之间的数据传输，不同的设备通常处理数据的速度是不同的，当需要在它们之间连续处理一批数据时，高速设备总是要等待低速设备，这就造成了计算机处理效率的大大降低。为了解决这一速度不匹配的矛盾，通常就是在这两个设备之间设置一个缓冲区。这样，高速设备就不必每次等待低速设备处理完一个数据，而是把要处理的数据依次从一端加入缓冲区，而低速设备从另一端取走要处理的数据。

3.2.1　队列的定义及其运算

1. 队列的定义

队列（queue）是一种只允许在一端进行插入，而在另一端进行删除的线性表，它是一种操作受限的线性表。在表中只允许进行插入的一端称为队尾（rear），只允许进行删除的一端称为队头（front）。队列的插入操作通常称为入队列或进队列，而队列的删除操作则称为出队列或退队列。当队列中无数据元素时，称为空队列。

根据队列的定义可知，队头元素总是最先进入队列的，也总是最先出队列；队尾元素总是最后进入队列，因而也是最后出队列。队列是按照先进先出（FIFO，First In First Out）的原则组织数据的，因此，队列也被称为“先进先出”表。

假如队列 q={a_1,a_2,…,a_n}，进队列的顺序为 a_1，a_2，…，a_n，则队头元素为 a_1，队尾元素为 a_n，如图 3-10 所示。

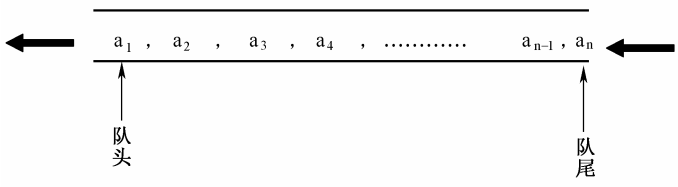


图 3-10　队列的示意图

队列的抽象数据类型定义如下：

```
ADT Queue {
    数据对象：D={ ai| ai∈ElemSet,i=1,2,...,n,n≥0}

    数据关系：R={ < ai-1,ai > | ai-1,ai ∈D,i=2,...,n}
    约定其中 a1 端为队列头，an 端为队列尾。
    基本操作：
    InitQueue(&Q)
    操作结果：构造一个空队列 Q。

    DestroyQueue(&Q)
    初始条件：队列 Q 已存在。
    操作结果：队列 Q 被销毁。

    ClearQueue(&Q)
    初始条件：队列 Q 已存在。
```

操作结果：将 Q 清为空队列。

QueueEmpty(Q)
初始条件：队列 Q 已存在。
操作结果：若 Q 为空队列，则返回 true，否则返回 false。

QueueLength(Q)
初始条件：队列 Q 已存在。
操作结果：返回 Q 的元素个数，即队列的长度。

GetHead(Q,&e)
初始条件：Q 为非空队列。
操作结果：用 e 返回 Q 的队头元素。

EnQueue(&Q,e)
初始条件：队列 Q 已存在。
操作结果：插入元素 e 为 Q 的新的队尾元素。

DeQueue(&Q,&e)
初始条件：Q 为非空队列。
操作结果：删除 Q 的队头元素，并用 e 返回其值。

QueueTraverse(Q,visit( ))
初始条件：队列 Q 已存在且非空，visit( )为元素的访问函数。
操作结果：依次对 Q 的每个元素调用函数 visit( )，一旦 visit( )失败则操作失败。

} ADT Queue

上述队列类型中的 9 个基本操作恰好和栈的 9 个操作是一一对应的。

3.2.2 队列的顺序存储结构

1. 队列的顺序存储结构

队列的顺序存储结构可以简称为顺序队列，也就是利用一组地址连续的存储单元依次存放队列中的数据元素。一般情况下，我们使用一维数组来作为队列的顺序存储空间，另外再设立两个指示器：一个为指向队头元素位置的指示器 front，另一个为指向队尾元素位置的指示器 rear。

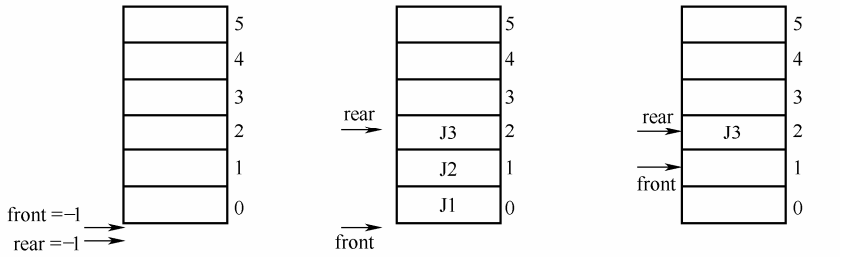
在 C++语言中，数组的下标是从 0 开始的，因此为了算法设计的方便，在此我们约定：初始化队列时，空队列时令 front=rear=-1，当插入新的数据元素时，尾指示器 rear 加 1，而当队头元素出队列时，队头指示器 front 加 1。另外还约定，在非空队列中，队头指示器 front 总是指向队列中实际队头元素的前面一个位置，而尾指示器 rear 总是指向队尾元素。

图 3-11 给出了队列中头尾指针的变化状态。

2. 队列的数据结构定义

```
#define MAXQSIZE 100
typedef   struct
{
    ElemType queue[MAXQSIZE];
```

```
    int front;
    int rear;
}Sqqueue;
```



（a）空队列　（b）J1、J2、J3 入队列后　（c）元素 J1、J2 出队列后

图 3-11　队列的存储结构

3．顺序队列的基本算法

（1）初始化队列。

```
//顺序队列的初始化
int InitQueue(Sqqueue *q)
{//创建一个空队列，由指针 q 指出
   if ((q=(Sqqueue*)malloc(sizeof(Sqqueue)))==NULL)
return false;
   q->front=-1;
   q->rear=-1;
return true;
}
```

（2）入队列操作。

```
//顺序队列的入队列操作
int EnQueue (Sqqueue *q,Elemtype x)
{//将元素 x 插入到队列 q 中，作为 q 的新队尾
   if(q->rear>=MAXNUM-1)
return false; //队列满
   q->rear++;
q->queue[q->rear]=x;
return true;
}
```

（3）出队列操作。

```
//顺序队列的出队列操作
Elemtype DeQueue (Sqqueue *q)
{//若队列 q 不为空，则返回队头元素
Elemtype x;
if(q->rear==q->front)
return NULL; //队列空
x=q->queue[++q->front];
return x;
}
```

（4）取队头元素操作。

```
//顺序队列的取队头元素操作。
Elemtype GetHead (Sqqueue *q)
{//若队列 q 不为空，则返回队头元素
 Sqqueue *s;
   if(q->rear==q->front)
return NULL; //队列空
return (q->queue[++s->front]);
}
```

（5）队列的非空判断操作。

```
//顺序队列的非空判断操作
int QueueEmpty(Sqqueue *q)
{//队列 q 为空时，返回 true；否则返回 false
if (q->rear==q->front)
return true;
   return false;
}
```

（6）队列的求长度操作。

```
//顺序队列的求长度操作
int QueueLength(Sqqueue *q)
{//返回队列 q 的元素个数
   return(q->rear-q->front);
}
```

3.2.3 队列的链式存储结构

1. 链队列的存储结构

用链表表示的队列简称为链队列，在一个链队列中需设定两个指针（头指针和尾指针）分别指向队列的头和尾。为了操作方便，和线性链表一样，我们也给链队列添加一个头结点，并设定头指针指向头结点。因此，空队列的判定条件就成为头指针和尾指针都指向头结点。

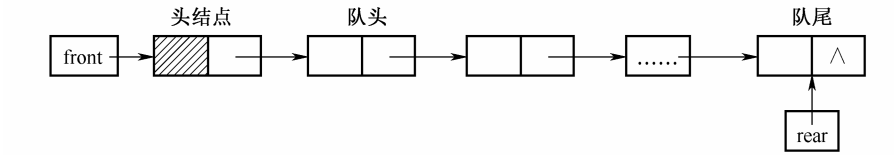
如图 3-12（a）所示为一个链队列；图 3-12（b）为空队列、入队和出队的示意图。

2. 链队列的数据结构定义

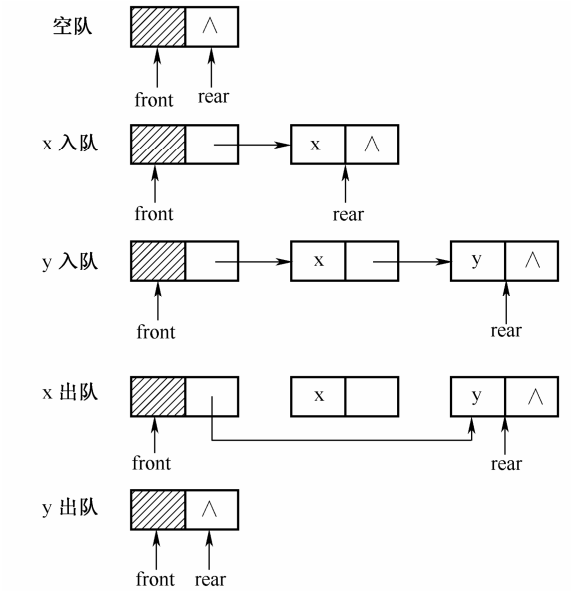
用 C++语言定义链队列结构如下：

```
typedef struct Qnode
{ Elemtype data;
struct Qnode *next;
} Qnodetype; //定义队列的结点

typedef struct
{   Qnodetype *front;  //头指针
Qnodetype *rear;       //尾指针
}Lqueue;
```



（a）



（b）

图 3-12　链队列示意图

3．链队列的基本算法

（1）初始化队列。

```
//链队列的初始化
int InitQueue (Lqueue *q)
{//创建一个空链队列 q
   if ((q->front=(Qnodetype*)malloc(sizeof(Qnodetype)))==NULL)
return false;
   q->rear=q->front;
q->front->next=NULL;
   return true;
}
```

（2）入队列操作。

```
//链队列的入队列操作
int EnQueue (Lqueue *q,Elemtype x)
{//将元素 x 插入到链队列 q 中，作为 q 的新队尾
Qnodetype *p;
if ((p=(Qnodetype*)malloc(sizeof(Qnodetype)))==NULL)
return false;
p->data=x;
```

```
p->next=NULL; //置新结点的指针为空
q->rear->next=p; //将链队列中最后一个结点的指针指向新结点
q->rear=p; //将队尾指向新结点
return true;
}
```

（3）出队列操作。

```
//链队列的出队列操作
Elemtype DeQueue (Lqueue *q)
{//若链队列 q 不为空，则删除队头元素，返回其元素值
Elemtype x;
Qnodetype *p;
if(q->front->next==NULL)
return NULL; //空队列
p=q->front->next; //取队头
q->front->next=p->next; //删除队头结点
x=p->data;
free(p);
return x;
}
```

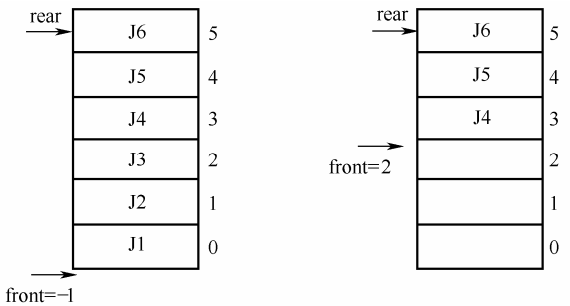
3.2.4 循环队列

1. 循环队列的概念

在顺序队列中，当队尾指针已经指向了队列的最后一个位置时，此时若有元素入列，就会发生溢出。设有数组 queue[M]，则：

（1）当 front=-1，rear=M-1 时，再有元素入队就会发生溢出——真溢出，如图 3-13（a）所示。

（2）当 front≠-1，rear=M-1 时，再有元素入队就会发生溢出——假溢出，如图 3-13（b）所示，虽然队尾指针已经指向最后一个位置，但事实上队列中还有 3 个空位置。也就是说，队列的存储空间并没有满，但队列却发生了溢出。



（a）真溢出　　（b）假溢出

图 3-13　队列溢出

解决这个问题有两种可行的方法：

（1）采用平移元素的方法，当发生假溢出时，就把整个队列的元素平移到存储区的首部，然后再插入新元素。这种方法需移动大量的元素，因而效率是很低的。

（2）将顺序队列的存储区假想为一个环状的空间，如图 3-14 所示。我们可假想 queue[0]接在 queue[M-1]的后面。当发生假溢出时，将新元素插入到第 1 个位置上（queue[0]），即若 rear+1==M，则令 rear=0。这样做虽然物理上队尾在队首之前，但逻辑上队首仍然在前。入列和出列仍按“先进先出”的原则进行，这就是循环队列。

很显然，方法（2）不需要移动元素，操作效率高，空间的利用率也很高。

在循环队列中，每插入一个新元素时，就把队尾指针沿顺时针方向移动一个位置。即：

```
rear=rear+1;
if（rear==m）
 rear=0;
```

在循环队列中，每删除一个元素时，就把队头指针沿顺时针方向移动一个位置。即：

```
front=front+1;
if（front==M）
front=0;
```

如图 3-14（b）所示，循环队列为队空时，有 front==rear；如图 3-14（c）所示，循环队列为队满时，也有 front==rear；因此仅凭 front==rear 不能判定队列是空还是满。

区分循环队列是空还是满的方法如下：

（1）另外设一个标志以区别队空、队满。

（2）少用一个元素空间：

队空：front==rear

队满：(rear+1)% M==front

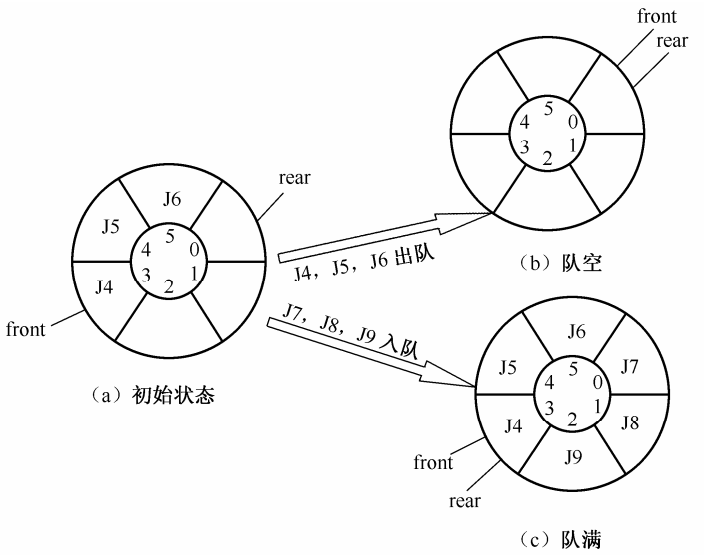


图 3-14　循环队列

2. 循环队列的数据结构定义

若设定一个标志位 s，用以区别队空、队满，当 s=0 时为空队列，当 s=1 时队列非空。那么用 C++语言定义循环队列数据结构如下：

```
#define MAXNUM 100
```

```
typedef struct
{Elemtype queue[MAXNUM];
int front; //队头指示器
int rear; //队尾指示器
int s; //队列标志位
}qqueue;
```

3．循环队列的基本算法

（1）初始化队列。

```
//循环队列的初始化
int initQueue(qqueue *q)
{//创建一个空队列，由指针 q 指出
if ((q=(qqueue*)malloc(sizeof(qqueue)))==NULL) return false;
q->front=MAXNUM-1;
q->rear=MAXNUM-1;
q->s=0;
return true;
}
```

（2）入队列操作。

```
//循环队列的入队列操作
int append(qqueue *q,Elemtype x)
{//将元素 x 插入到队列 q 中，作为 q 的新队尾
if (( q->s==1)&&(q->front==q->rear))
return false; //队列满
q->rear++;
if (q->rear==MAXNUM) q->rear=0;
q->queue[q->rear]=x;
q->s=1; //置队列非空
return true;
}
```

（3）出队列操作。

```
//循环队列的出队列操作
Elemtype delet(qqueue *q)
{//若队列 q 不为空，则返回队头元素
Elemtype   x;
if (q->s==0) return NULL; //队列为空
q->front++;
if (q->front==MAXNUM) q->front=0;
x=q->queue[q->front];
if (q->front==q->rear) q->s=0; //置队列空
return x; }
```

3.2.5 队列的应用举例

编写一个打印二项式系数表（即杨辉三角）的算法，杨辉三角如下所示：

```
      1
    1 2 1
  1 3 3 1
1 4 6 4 1
……
```

系数表中的第 k 行有 k+1 个数，除了第一个和最后一个数为 1 之外，其余的数则为上一行中位于其左上方、右上方的两数之和。

这个问题的程序可以有很多种写法，一种最直接的想法是利用两个数组，其中一个存放已经计算得到的第 k 行的值，然后输出第 k 行的值同时计算第 k+1 行的值。如此得到的程序显然结构清晰，但需要两个辅助数组的空间，并且这两个数组在计算过程中需相互交换。如若引入“循环队列”，则可以省略一个数组的辅助空间，而且可以利用队列的操作使程序结构变得清晰，容易被人理解。

如果要求计算并输出杨辉三角前 n 行的值，则队列的最大空间应为 n+2。假设队列中已存有第 k 行的计算结果，并为了计算方便，在两行之间添加一个 0 作为行界值，则在计算第 k+1 行之前，头指针正指向第 k 行的 0，而尾元素为第 k+1 行的 0。由此从左到右依次输出第 k 行的值，并将计算所得的第 k+1 行的值插入队列的基本操作为：

```
do {
    DeQueue(Q, s);      // s 为二项式系数表第 k 行中“左上方”的值
    GetHead(Q, e);      // e 为二项式系数表第 k 行中“右上方”的值
    cout<<e;            // 输出 e 的值
    EnQueue(Q, s+e);  // 计算所得第 k+1 行的值入队列
  } while (e!=0);
```

例如，假设 n=6，当前队列中存放的是第 4 行的值，计算第 5 行的值同时输出第 4 行的值的过程如图 3-15 所示。

完整的算法需要补充的细节如下：

（1）输出 e 是有条件的，即行末的 0 不需要输出。

（2）一行计算结束之后需要补一个 0，这个 0 值既是刚刚计算完的这行的尾 0，又是即将开始计算的下一行的头 0。

（3）其他初始化和结尾（输出最后一行）等。

完整的算法如下：

```
//计算杨辉三角的算法
#include "stdlib.h"
#include "iostream.h"
#define MAXNUM 100
typedef struct
{int queue[MAXNUM];
int front; //队头指示器
int rear; //队尾指示器
int s; //队列标志位
}Queue;
```

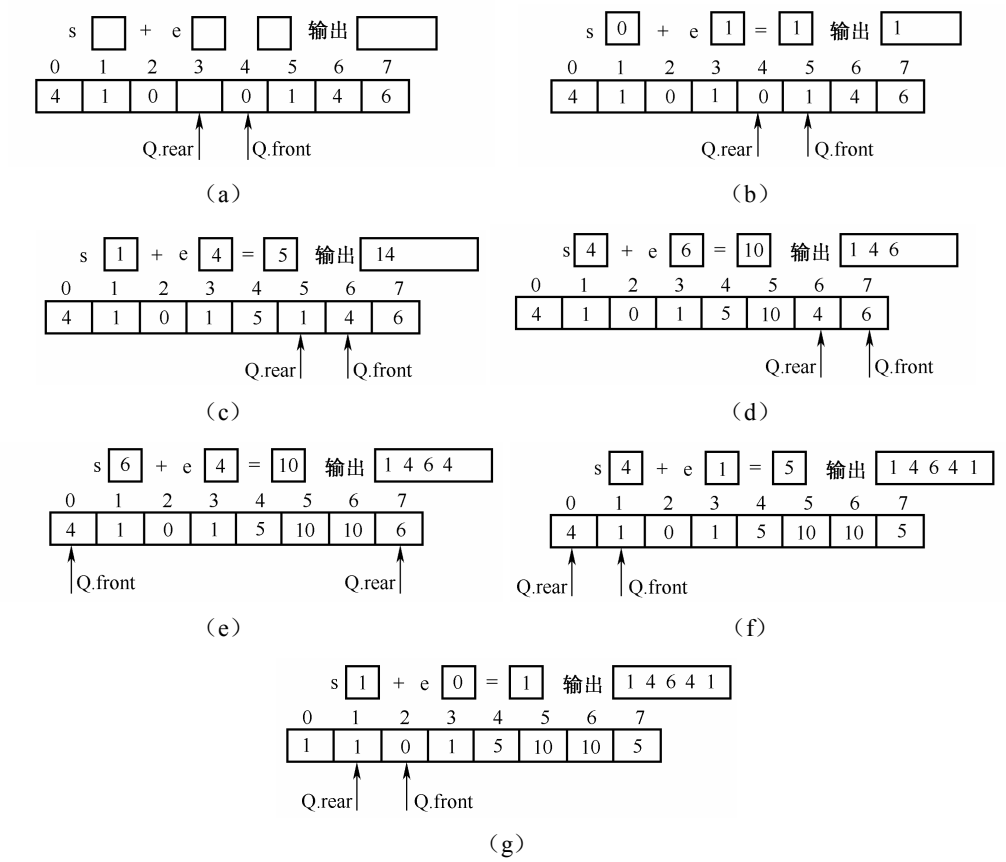


图 3-15　计算杨辉三角的算法示意

```
int N,Realsize;
//循环队列的初始化
int InitQueue(Queue &q,int maxsize)
{//创建一个空队列由指针 q 指出，最大存储空间为 maxsize
   Queue *p;
   if (maxsize==0)
       maxsize=MAXNUM;
if ((p=(Queue*)malloc(sizeof(Queue)))==NULL) return false;
p->front=0;
p->rear=0;
p->s=0;
q=*p;
return true;
}

//循环队列的入队列操作
int EnQueue(Queue &q,int x)
{//将元素 x 插入到队列 q 中作为 q 的新队尾
if (( q.s==1)&&(q.front==q.rear))
```

```
return false; //队列满
if (q.rear==Realsize) q.rear=0;
q.queue[q.rear]=x;
q.rear++;
q.s=1; //置队列为非空
return true;
}

//循环队列的出队列操作
int DeQueue(Queue &q)
{//若队列 q 不为空，则返回队头元素
int x;
if (q.s==0) return NULL; //队列为空
if (q.front==Realsize) q.front=0;
x=q.queue[q.front];
q.front++;
if (q.front==q.rear) q.s=0; //置队列为空
return x; }

int GetHead (Queue &q)
{//若队列 q 不为空，则返回队头元素
   if(q.rear==q.front)
       return NULL; //队列空
   if (q.front==Realsize)
    q.front=0;
   return q.queue[q.front];
}

int QueueEmpty(Queue &q)
{//队列 q 为空时，返回 true；否则返回 false
if (q.rear==q.front)
return true;
   return false;
}

void yanghui ( int n )
{   // 打印输出杨辉三角的前 n（n>0）行
 int e,s;
     Queue q;
     for( int i=1; i<=n; i++)         cout<< ' ';
        cout<< '1'<<endl;   // 在中心位置输出杨辉三角最顶端的 1
 Realsize=n+2;
     InitQueue(q,Realsize);                          // 设置最大容量为 n+2 的空队列
     EnQueue(q,0 );                                  // 添加行界值
     EnQueue( q,1);
```

```
    EnQueue( q,1 );                          // 第 1 行的值入队列
    int k=1;
    while ( k < n )
{ // 通过循环队列输出前 n-1 行的值
      for( i=1; i<=n-k; i++)    cout<< ' ';    // 输出 n-k 个空格以保持三角型
        EnQueue ( q,0 );                     // 行界值 0 入队列
      do {                                   // 输出第 k 行，计算第 k+1 行
          s=DeQueue( q );// s 为二项式系数表第 k 行中“左上方”的值
          e=GetHead( q );// e 为二项式系数表第 k 行中“右上方”的值
          if (e) cout<< e << ' ';
       // 若 e 为非行界值 0，则打印输出 e 的值并加一空格
          else cout << endl;     // 否则回车换行，为下一行输出做准备
          EnQueue(q,s+e);        // 计算所得第 k+1 行的值入队列
      } while (e!=0);
       k++;
    } // while
    e=DeQueue ( q );                         // 行界值 0 出队列
    while (!QueueEmpty (q) )
 {   // 单独处理第 n 行的值的输出
    e=DeQueue ( q );
    cout<<e<< ' ';
 } // while
} // yanghui

void main()
{ N=5;
  yanghui ( N ) ;
}
```

容易看出此算法的时间复杂度为 $O(n^2)$。

思考题与习题

3.1 说明线性表、栈与队的异同点。

3.2 设有编号为 1、2、3、4 的 4 辆列车，顺序进入一个如图 3-16 所示的栈式结构的车站，具体写出这 4 辆列车开出车站的所有可能的顺序。

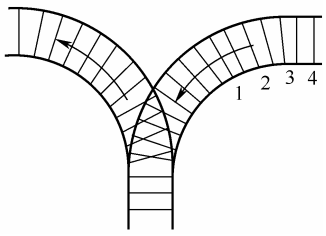


图 3-16 习题 3.2 图 栈式结构的车站

3.3　正读和反读都相同的字符序列称为“回文”，例如 abba 和 abcba 是回文，abcde 和 ababab 则不是回文。假设一字符序列已存入计算机，请分析用线性表、堆栈和队列等方式正确输出其回文的可能性。

3.4　顺序队的“假溢出”是怎样产生的？如何知道循环队列是空还是满？

3.5　设循环队列的容量为 40（序号从 0 到 39），现经过一系列的入队和出队运算后，有① front=11，rear=19；② front=19，rear=11。问在这两种情况下，循环队列中各有多少个元素？

3.6　设表达式的中缀表示为 a*x-b/x↑2，试利用栈将它改为后缀表示 ax*bx2↑/-。写出转换过程中栈的变化。

3.7　写出下列程序段的输出结果（栈的元素类型为 char）。

```
void main( ){
Stack S;
char x,y;
InitStack(S);
X='c';y='k';
Push(S,x); Push(S,'a');Push(S,y);
Pop(S,x); Push(S,'t'); Push(S,x);
Pop(S,x); Push(S,'s');
while(!StackEmpty(S)){ Pop(S,y);cout <<y; };
cout <<x;
}
```

3.8　写出下列程序段的输出结果（队列中的元素类型为 char）。

```
void main( ){
Queue Q; InitQueue (Q);
char x='e'; y='c';
EnQueue (Q,'h'); EnQueue (Q,'r');   EnQueue (Q, y);
DeQueue (Q,x); EnQueue (Q,x);
DeQueue (Q,x); EnQueue (Q,'a');
while(!QueueEmpty(Q)){ DeQueue (Q,y);cout<<y; };
cout <<x;
}
```

3.9　简述以下算法的功能（栈和队列的元素类型均为 int）。

```
void algo3(Queue &Q){
Stack S; int d;
InitStack(S);
while(!QueueEmpty(Q)){
DeQueue (Q,d);   Push(S,d);
};
while(!StackEmpty(S)){
Pop(S,d); EnQueue (Q,d);
}
}
```

3.10　假设一个算术表达式中包含圆括号、方括号和大括号三种类型的括号，编写一个判别表达式中括号是否正确配对的函数 correct(exp,tag)；其中 exp 为字符串类型的变量（可理解为每个字

符占用一个数组元素），表示被判别的表达式，tag 为布尔型变量。

3.11 假设一个数组 squ[m]存放循环队列的元素。若要使这 *m* 个分量都得到利用，则需另一个标志 tag，以 tag 为 0 或 1 来区分尾指针和头指针值相同时队列的状态是“空”还是“满”。试编写相应的入队和出队的算法。

3.12 试写一个算法判别读入的一个以@为结束符的字符序列是否是回文。

3.13 改写顺序栈的进栈成员函数 Push(x)，要求当栈满时执行一个 stackFull()操作进行栈满处理。其功能是：动态创建一个比原来的栈数组大二倍的新数组，代替原来的栈数组，原来栈数组中的元素占据新数组的前 MaxSize 个位置。

3.14 试证明：若借助栈可由输入序列 1，2，3，…，*n* 得到一个输出序列 p1，p2，p3，…，pn （它是输入序列的某一种排列），则在输出序列中不可能存在 $i<j<k$，使得 $p_j<p_k<p_i$。（提示：用反证法）

3.15 写出下列中缀表达式的后缀形式：

（1）A*B*C

（2）-A+B-C+D

（3）A*-B+C

（4）(A+B)*D+E/(F+A*D)+C

（5）A && B|| !(E>F) （按 C++的优先级分析）

（6）!(A && !((B<C)||(C>D)) ||(C<E))

3.16 试利用优先级队列实现栈和队列。

3.17 假设以数组 Q[m]存放循环队列中的元素，同时以 rear 和 length 分别指示环形队列中的队尾位置和队列中所含元素的个数。试给出该循环队列的队空条件和队满条件，并写出相应的插入（enqueue）和删除（dlqueue）元素的操作。

3.18 假设以数组 Q[m]存放循环队列中的元素，同时设置一个标志 tag，以 tag==0 和 tag==1 来区别在队头指针（front）和队尾指针（rear）相等时，队列状态为“空”还是“满”。试编写与此结构相应的插入（enqueue）和删除（dlqueue）算法。

3.19 若使用循环链表来表示队列，p 是链表中的一个指针。试基于此结构给出队列的插入（enqueue）和删除（dlqueue）算法，并给出 p 为何值时队列空。

3.20 若将一个双端队列顺序表示在一维数组 V[m]中，两个端点设为 end1 和 end2，并组织成一个循环队列。试写出双端队列所用指针 end1 和 end2 的初始化条件及队空与队满条件，并编写基于此结构的相应的插入（enqueue）和删除（dlqueue）算法。

3.21 设用链表表示一个双端队列，要求可在表的两端插入，但限制只能在表的一端删除。试编写基于此结构的队列的插入（enqueue）和删除（dlqueue）算法，并给出队列空和队列满的条件。

3.22 试建立一个继承结构，以栈、队列和优先级队列为派生类，建立它们的抽象基类——Bag 类。写出各个类的声明。统一命名各派生类的插入操作为 Add，删除操作为 Remove，存取操作为 Get 和 Put，初始化操作为 MakeEmpty，判空操作为 Empty，判满操作为 Full，计数操作为 Length。