第 3 章　栈 和 队 列

栈和队列是在软件设计中常用的两种数据结构，它们的逻辑结构和线性表相同。其特点在于运算受到了限制：栈按“后进先出”或“先进后出”的规则进行操作，队列按“先进先出”的规则进行操作，故称它们为操作受限制的线性表。

3.1　栈

3.1.1　栈的定义及其基本运算

栈(Stack)是限制在表的一端进行插入和删除操作的线性表。允许进行插入、删除操作的这一端称为栈顶(Top)，另一个固定端称为栈底。当表中没有元素时称为空栈。如图 3.1 所示的栈中有三个元素，进栈的顺序是 a_1、a_2、a_3，当需要出栈时其顺序为 a_3、a_2、a_1，所以栈又称为“后进先出”(Last-In First-Out，LIFO)或“先进后出”(First-In Last-Out，FILO)的线性表，简称“LIFO 表”或者“FILO 表”。

注意：LIFO 规范的写法有两种，即 Last-In First-Out、“Last In，First Out”。FILO 类似。

在日常生活中，有很多类似栈的例子，读者可以列举。在程序设计中，常常需要将数据按其保存时相反的顺序来使用这些数据，这时就需要用一个栈这样的数据结构来实现。

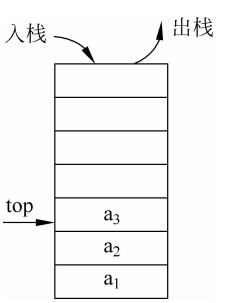

图 3.1　栈的示意图

对于栈，常见的基本运算有：

(1) 栈初始化：Init_Stack(s)

初始条件：栈 s 不存在。

操作结果：构造了一个空栈。

(2) 判栈空：Empty_Stack(s)

初始条件：栈 s 已存在。

操作结果：若 s 为空栈，则返回 1，否则返回 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) 读栈顶元素：Top_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。

3.1.2　栈的存储结构和基本运算的实现

由于栈是运算受限的线性表，因此线性表的存储结构对栈也是适用的，只是操作不同而已。

1. 顺序栈

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，栈中的数据元素用一个预设的足够长度的一维数组来实现：datatype data[MAXSIZE]，栈底位置可以设置在数组的任一个端点，而栈顶是随着插入和删除而变化的，用一个 int top 来作为栈顶的指针，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
```

定义一个指向顺序栈的指针：

```
SeqStack  *s;
```

通常将 0 下标端设为栈底，这样空栈时栈顶指针 top = −1；入栈时，栈顶指针加 1，即 s−>top++；出栈时，栈顶指针减 1，即 s−>top−−。

栈顶指针 top 与栈中数据元素的关系如图 3.2 所示，其中(a)为空栈，(c)是 A、B、C、D、E 5 个元素依次入栈之后，栈的状态以及栈顶指针所指位置，(d)是在(c)之后 E、D 相继出栈，此时，栈中还有 3 个元素，或许最近出栈的元素 D、E 仍然在原先的单元存储中，但由于 top 指针已经指向了新的栈顶，则存储 D、E 的单元已经属于无效数据，意味着元素 D、E 已不在栈中了，通过这个示意图要深刻理解栈顶指针的作用。

在上述存储结构上基本操作的实现如下：

(1) 置空栈

首先建立栈空间，然后初始化栈顶指针。

```
SeqStack  *Init_SeqStack( )
      {   SeqStack  *s;
          s = malloc(sizeof(SeqStack));
          s->top = -1;  return  s;
      }
```

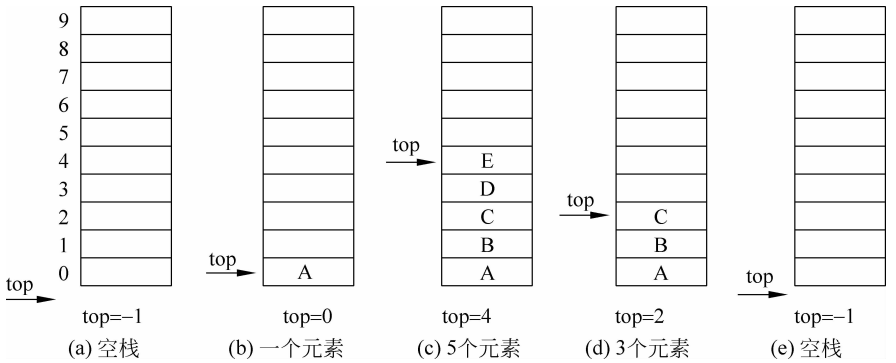

图 3.2　栈顶指针 top 与栈中数据元素的关系

(2) 判栈空

```
int  Empty_SeqStack(SeqStack  * s)
  {  if (s->top= = -1)  return  1;
     else  return  0;
  }
```

(3) 入栈

```
int  Push_SeqStack (SeqStack  * s,  datatype  x)
  { 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

```
int  Pop_SeqStack(SeqStack  * s,  datatype  * x)
  { if(Empty_SeqStack ( s ) )  return  0;                  /* 栈空不能出栈 */
      else { * x=s->data[s->top];
             s->top--;  return  1;
           }                                               /* 栈顶元素存入 * x,返回 */
  }
```

(5) 取栈顶元素

```
datatype  Top_SeqStack(SeqStack  * s)
      {   if ( Empty_SeqStack ( s ) )  return  0;          /* 栈空 */
            else  return (s->data[s->top] );
      }
```

说明：

(1) 对于顺序栈，入栈时，首先判断栈是否满了，栈满的条件为：s->top= =MAXSIZE-1。栈满时不能入栈，否则会出现空间溢出，引起错误，这种现象称为上溢。

(2) 出栈和读栈顶元素操作时,先判断栈是否为空,为空时不能操作,否则会产生错误。通常栈空时常作为一种控制转移的条件。

2. 链栈

用链式存储结构实现的栈称为链栈。通常链栈用单链表表示,因此其结点结构与单链表的结点结构相同,在此用 LinkStack 表示,即有:

```
typedef  struct node
      {  datatype  data;
         struct node  * next;
      }  StackNode  * LinkStack;
```

说明: top 为栈顶指针,即"LinkStack top ;"。

因为栈中的主要运算是在栈顶进行插入、删除操作,显然在链表的头部做栈顶是最方便的,而且没必要像单链表那样为了运算方便而附加一个头结点。通常将链栈表示为图 3.3的形式。链栈基本操作的实现如下:

(1) 置空栈

```
LinkStack  Init_LinkStack( )
        {  return  NULL;
        }
```

(2) 判栈空

```
int  Empty_LinkStack(LinkStack  top)
    {  if (top = = NULL)  return 1;
          else  return  0;
    }
```

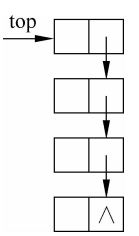

图 3.3　链栈示意图

(3) 入栈

```
LinkStack  Push_LinkStack(LinkStack  top,  datatype x)
        { StackNode  * s;
          s = malloc(sizeof(StackNode));
          s -> data = x;   s -> next = top;    top = s;
          return  top;
        }
```

(4) 出栈

```
LinkStack  Pop_LinkStack (LinkStack  top, datatype  * x)
        { StackNode  * p;
          if (top = = NULL)  return  NULL;
              else  { * x = top -> data;   p = top;    top = top -> next;   free (p);
                    return  top;
              }
        }
```

3.1.3 栈的应用举例

由于栈有“后进先出”的特点,因此在很多实际问题中都利用栈做一个辅助的数据结构来进行求解,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,其转换方法是利用辗转相除法,下面以 N=3467、r=8 为例,介绍转换方法如下:

N	N/8(整除)	N%8(求余)	
3467	433	3	↑ 低
433	54	1	
54	6	6	
6	0	6	高

按逆序取余数,即得转换结果:

$$(3467)_{10} = (6613)_8$$

可以看出,转换得到的八进制数是按低位到高位的顺序产生的,而转换结果的输出通常是从高位到低位依次输出,恰好与产生过程相反,因此,在转换过程中,每得到一位八进制数就将其进栈保存,转换完毕后再依次出栈,则正好是转换结果。

【算法思路】 当 N>0 时,重复步骤(1)、(2)。

(1) 若 N≠0,则将 N%r 压入栈 s 中,执行步骤(2);若 N = 0,将栈 s 的内容依次出栈,算法结束。

(2) 用 N/r 代替 N,返回步骤(1)。

算法实现如下,分别用两种不同的方法进行描述。

算法 3.1(a)

```
typedef  int  datatype;
void  conversion( int  N, int  r)
  { SeqStack  s;
    datetype  x;
    Init_SeqStack(&s);
    while( N )
      { Push_SeqStack ( &s,N % r );
        N = N / r ;
      }
    while(! Empty_SeqStack(& s ) )
      { Pop_SeqStack (&s,&x ) ;
        printf (" % d",x ) ;
      }
  }
```

算法 3.1(b)

```
#define  L  10
void  conversion( int  N,int  r )
  { int  s[L], top;          /* 定义一个顺序栈 */
    int  x;
    top = -1;                /* 初始化栈 */
    while ( N )
    { s [++top] = N % r;     /* 将余数入栈 */
      N = N / r ;            /* 商作为被除数继续 */
    }
    while ( top != -1)
    {  x = s[top -- ];
       printf( " % d",  x );
    }
}
```

算法 3.1(a)是将对栈的操作抽象为模块调用，使问题的层次更加清楚；而算法 3.1(b)中是直接用 int 数组 s 和 int 变量 top 作为一个栈来使用。两种描述方法的实现过程其实是相同，只是前者的可读性要更清晰。初学者往往将栈视为一个很复杂的东西，不知道如何使用，通过这个例子可以消除栈的“神秘”。当应用程序中需要按数据保存时相反的顺序使用数据时，就要想到栈。通常用顺序栈较多，因为很便利。

在后面的例子中，为了在算法中表现出问题的层次，有关栈的操作调用了相关函数，如算法 3.1(a)那样，对余数的入栈操作为“Push_SeqStack(&s,N %r); ”，因为是用 C 语言描述，第一个参数是栈的地址才能对栈进行加工(在后面的例子中，为了算法的清楚易读，在不至于混淆的情况下，不再加地址运算符，请读者注意)。

例 3.2　表达式求值问题。

表达式求值是程序设计语言编译中一个最基本的问题，它的实现也是需要运用栈。下面介绍的算法是由运算符优先级确定运算顺序后对表达式进行求值的算法。

一个表达式是由运算数(operand)、运算符(operator)和界限符(delimiter)组成的有意义的式子。一般的，运算数既可以是常量，也可以是被说明的变量或常量标识符。运算符从运算数的个数上分，有单目运算符和双目运算符；从运算类型上分，有算术运算符、关系运算符和逻辑运算符。而界限符主要是指左右括号和表达式结束符。在此仅限于讨论只含常量的双目运算的算术表达式。

假设所讨论的算术运算符包括：+ 、- 、* 、/ 、%、^(乘方)和括号()。

设运算规则如下：

- 运算符的优先级为：()⟶^⟶* 、/ 、% ⟶+ 、- 。
- 有括号出现时先算括号内的，后算括号外的。多层括号，由内向外进行。
- 乘方连续出现时先算最右面的。

可以将表达式作为一个满足表达式语法规则的串来存储，如 3*2^(4+2*2-1*3)-5。

为实现表达式求值，需要设置两个栈：一个称为运算符栈 OPTR，用于寄存运算符；另一个称为运算数栈 OPND，用于寄存运算数和运算结果。求值的处理过程是：自左至右扫描表达式的每一个字符，当扫描到的是运算数，就将其压入栈 OPND；当扫描到的是运算符时，若这个运算符比 OPTR 栈顶运算符的优先级高，则入栈 OPTR，并继续向后处理；若这个运算符比 OPTR 栈顶运算符的优先级低，则从 OPND 栈中弹出两个运算数，从 OPTR 栈中弹出栈顶运算符进行运算，并将运算结果压入栈 OPND，继续处理当前字符，直到遇到结束符为止。

根据运算规则，左括号“(”在栈外时级别最高，而进栈后级别最低。乘方运算的结合性是自右向左，所以，它的栈外级别高于栈内，也就是说有的运算符栈内栈外的级别是不同的。当遇到右括号“)”时，一直需要对 OPTR 栈进行出栈操作，并弹出相应的操作数，做对应的运算，直到遇到栈顶为左括号“(”时，将其出栈为止。

OPND 栈初始化为空，为了使表达式中的第一个运算符入栈，OPTR 栈中预设一个最低级的运算符“(”。根据以上分析，每个运算符的栈内、栈外优先级别如表 3.1 所示。

表 3.1　运算符栈内、栈外的级别比较

运　算　符	栈内级别	栈外级别
^	3	4
*、/、%	2	2
+、-	1	1
(	0	4
)	-1	-1

以上算法的基本思想是：

(1) 首先置 OPND 栈为空栈，表达式起始符“(”为 OPTR 栈的栈底元素。

(2) 依次读入表达式中每个字符，若是运算数则进 OPND 栈；若是运算符，则和 OPTR 栈的栈顶运算符比较优先级后，再作相应操作，直至整个表达式求值完毕(即 OPTR 栈的栈顶元素为“(”且当前读入的字符为“#”)。

算法 3.2　表达式求值算法。

```
OperandType  EvaluateExpression( )
{ /* 算术表达式求值的算符优先算法,设 OPTR 和 OPND 分别为运算符栈和运算数栈 */
  /* OP 为运算符集合 */
  Init_Stack (OPTR);
  Push_Stack (OPTR, '(');
  Init_Stack (OPND);
  c = getchar( );
  while ( c! = '#')
    { if (!In (c, OP))                          /* 不是运算符则进栈 */
          { Push_Stack (OPND,c);  c = getchar(); }
      else
        switch (Precede(GetTop(OPTR),c))
            {
            case '<':                           /* 栈顶元素优先级低 */
                    Push_Stack (OPTR,c);  c = getchar();
                    break;
            case '=':                           /* 去括号并接受下一字符 */
                    Pop_Stack (OPTR,x);  c = getchar();
                    break;
            case '>':                           /* 退栈并将运算结果入栈 */
                    Pop_Stack (OPTR,  theta);
                    Pop_Stack (OPND, b);
                    Pop_Stack (OPND, a);
                    Push_Stack (OPND, Operate(a, theta, b));
                    break;
            }                                   /* Switch */
    }                                           /* while */
      return  GetTop(OPND);
}                                               /* EvaluateExpression */
```

算法 3.2 中还调用了两个函数。其中，Precede 是判定 OPTR 栈的栈顶运算符与读

入的运算符之间优先关系的函数；Operate 为进行二元运算 aθb 的函数。如果是编译表达式，则产生这个运算的一组相应指令并返回存放结果的中间变量名；如果是解释执行表达式，则直接进行该运算，并返回运算的结果。

在实际编译程序中，为了处理方便，常常首先把表达式转换成等价的后缀表达式。所谓后缀表达式是指将运算符放在运算数之后的表达式。在后缀表达式中，不再需要括号，所有的计算按运算符出现的顺序严格地从左向右进行，而不用再考虑运算符的级别和运算规则。如表达式 3＊2^(4＋2＊2－1＊3)－5 的后缀表达式为 3 2 4 2 2 ＊ ＋ 1 3 ＊ － ^ ＊ 5 －，其求值过程中两个栈的情况如表 3.2 所示。

表 3.2　表达式 3＊2^(4＋2＊2 － 1＊3) －5 求值过程中两个栈的情况

读字符	栈 OPND(s1)	栈 OPTR(s2)	说　　明
3	3	(	3 入栈 s1
＊	3	(＊	＊入栈 s2
2	3,2	(＊	2 入栈 s1
^	3,2	(＊^	^入栈 s2
(	3,2	(＊^(	(入栈 s2
4	3,2,4	(＊^(	4 入栈 s1
＋	3,2,4	(＊^(＋	＋入栈 s2
2	3,2,4,2	(＊^(＋	2 入栈 s1
＊	3,2,4,2	(＊^(＋＊	＊入栈 s2
2	3,2,4,2,2	(＊^(＋＊	2 入栈 s1
－	3,2,4,4	(＊^(＋	做 2＊2＝4，结果入栈 s1
	3,2,8	(＊^(	做 4＋4＝8，结果入栈 s1
	3,2,8	(＊^(－	－入栈 s2
1	3,2,8,1	(＊^(－	1 入栈 s1
＊	3,2,8,1	(＊^(－＊	＊入栈 s2
3	3,2,8,1,3	(＊^(－＊	3 入栈 s1
)	3,2,8,3	(＊^(－	做 1＊3，结果 3 入栈 s1
	3,2,5	(＊^(	做 8－3，结果 5 入栈 s1
	3,2,5	(＊^	(出栈
－	3,32	(＊	做 2^5，结果 32 入栈 s1
	96	(	做 3＊32，结果 96 入栈 s1
	96	(－	－入栈 s2
5	96,5	(－	5 入栈 s1
结束符	91	(	做 96－5，结果 91 入栈 s1

3.1.4　栈与递归的实现

栈的另一个非常重要的应用就是在程序设计语言中实现递归。递归是指在定义自身的同时又出现对自身的引用。如果一个函数在其定义体内直接调用自己，则称为直接递

归函数；如果一个函数经过一系列中间调用语句，通过其他函数间接调用自己，则称为间接递归函数。

1. 具有递归特性的问题

现实中，许多问题具有固有的递归特性。

(1) 递归定义的数学函数

很多数学函数是递归定义的，例如大家熟悉的阶乘函数可写成递归定义：

$$\mathrm{Fact}(n)=\begin{cases}1 & n=0\\ n\cdot \mathrm{Fact}(n-1) & n>0\end{cases}$$

以上函数如果用 C 语言实现，代码如下：

```
long  Fact ( int  n )
  {  long  f;                                  /* 长整数类型可以使数据不容易溢出 */
     if  ( n == 0 )  f = 1;
       else  f = n * Fact( n - 1 );
     return  f;
  }
```

(2) 递归数据结构的处理

在本书的后续章节中将要学习的一些数据结构，如二叉树、树等，由于结构本身固有的递归特性，因此自然地采用递归方法进行处理。

(3) 递归求解方法

许多问题的求解可以通过递归分解的方法实现，虽然有些问题本身没有明显的递归结构，但用递归方法求解比迭代求解更简单，也更直观，如八皇后问题、Hanoi 塔问题等。

n 阶 Hanoi 塔问题：假设有三个分别命名为 X、Y、Z 的塔座，在塔座 X 上叠放着 n 个直径大小各不相同、小盘压在大盘之上圆盘堆。现要求将塔座 X 上的 n 个圆盘移至塔座 Z 上，并仍按同样顺序叠放。移动圆盘时必须遵循下列规则。

① 每一次只能够移动一个圆盘；

② 圆盘可以插放在 X、Y 和 Z 中任何一个塔座上；

③ 任何时刻都不能将一个较大的圆盘压在较小的圆盘之上。

解决以上问题的基本思想如下：

当 n=1 时，问题简单，只要将该圆盘从塔座 X 上移动到塔座 Z 上即可。

当 n>1 时，需要利用塔座 Y 作辅助塔座。若能设法将除底下最大的圆盘之外的 n−1 个圆盘从塔座 X 移动到塔座 Y 上，就可以将最大的圆盘从塔座 X 移至塔座 Z 上，然后再将塔座 Y 上的其余 n−1 个圆盘移动到塔座 Z 上。而如何将 n−1 个圆盘从一个塔座移至另一个塔座，则跟原问题具有相同的特征属性，只是问题的规模小 1，因此可以用同样的方法求解。

算法 3.3 求解 n 阶 Hanoi 塔的递归算法。

```
void  Hanoi ( int  n,  char  x,  char  y,  char  z )
/* 将塔座 x 上从上到下编号为 1 至 n 且按直径由小到大叠放的 n 个圆盘，按规则移动到塔座 z
```

```
上,塔座 y 用作辅助塔座 * /
{ if( n= =1 )
    move ( x,  n,  z ) ;                  /* 将编号为 n 的圆盘从塔座 x 移动到塔座 z 上 */
  else
    { Hanoi ( n-1,  x,  z,  y );          /* 将塔座 x 上编号为 1 至 n-1 的圆盘移至塔座 y、
                                             z,并作辅助塔座 * /
      move ( x,  n,  z );                 /* 将编号为 n 的圆盘从塔座 x 移动到塔座 z * /
      Hanoi ( n-1,  y,  x,  z );          /* 将塔座 y 上编号为 1 至 n-1 的圆盘移至塔座 z、
                                             x,并作辅助塔座 * /
    }
}
```

掌握递归的具体执行过程,对于理解递归具有重要作用。

下面就以三个圆盘的移动为例来说明以上递归函数 Hanoi(3,A,B,C)的具体递归调用过程,如图 3.4 所示。

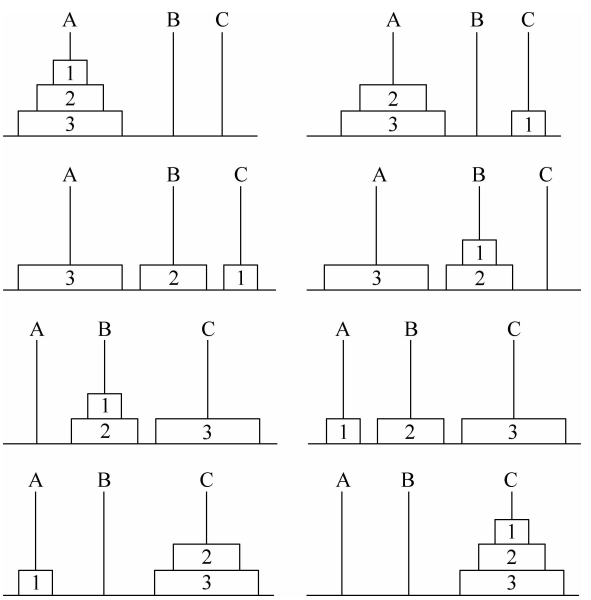

图 3.4　Hanoi 塔的递归函数运行示意图

```
Hanoi ( 3,  A,  B,  C )          /* 调用函数 hanoi 将 A 上的 3 个圆盘移至 B、C 作辅助塔座 * /
    Hanoi ( 2,  A,  C,  B );             /* 将 A 上的 2 个圆盘移至 B、C 作辅助塔 * /
        Hanoi ( 1,  A,  B,  C );         /* 将 A 上的 1 个圆盘移至 C * /
            move ( A,  1,  C );          /* 将 1 号圆盘从 A 搬到 C * /
        move ( A,  2,  C );              /* 将 2 号圆盘从 A 搬到 B * /
        Hanoi ( 1,  C,  A,  B );         /* 将 C 上的 1 个圆盘移至 B * /
            move ( C,  1,  B );          /* 将 1 号圆盘从 C 搬回到 B * /
    move ( A,  3,  C );                  /* 将 3 号圆盘从 A 搬到 C * /
    Hanoi ( 2,  B,  A,  C );             /* 将 B 上的 2 个圆盘移至 C、A 作辅助塔座 * /
        Hanoi ( 1,  B,  C,  A );         /* 将 B 上的 1 个圆盘移至 C * /
            move (B,  1,  A );           /* 将 1 号圆盘从 B 搬到 A * /
```