栈和队列的应用

要熟练掌握栈和队列，必须学习栈和队列的应用，把握其中的规律，然后举一反三。接下来简单介绍栈和队列的一些常见应用。

栈在括号匹配中的应用

假设表达式中允许包含两种括号：圆括号和方括号，其嵌套的顺序任意即([]())或[([][])]等均为正确的格式，[()]或([()]或(()]均为不正确的格式。

考虑下列括号序列：

[	(	[	]	[	]	)	]
1	2	3	4	5	6	7	8

分析如下：

1. 计算机接收第1个括号"["后，期待与之匹配的第8个括号"]"出现。
2. 获得了第2个括号"("，此时第1个括号"["暂时放在一边，而急迫期待与之匹配的第7个括号")"出现。
3. 获得了第3个括号"["，此时第2个括号"("暂时放在一边，而急迫期待与之匹配的第4个括号"]"出现。第3个括号的期待得到满足，小姐之后，第2个括号的期待匹配又成为当前最急迫的任务。
4. 以此类推，可见该处理过程与栈的思想吻合。

算法的思想如下：

1. 初始设置一个空栈，顺序读入括号。
2. 若是左括号，则作为一个新的更急迫的期待压入栈中，自然使原有的栈中所有未消解的期待的急迫性降了一级。
3. 若是右括号，则或使置于栈顶的最急迫期待得以消解，或是不合法的情况（括号序列不匹配，退出程序）。算法结束时，栈为空，否则括号序列不匹配。

栈在表达式求值中的应用

表达式求值是程序设计语言编译中一个最基本的问题，它是栈应用的一个典型范例。

算术表达式

中缀表达式（如3+4）是人们常用的算术表达式，操作符以中缀形式处于操作数的中间。与前缀表达式（如+34）或后缀表达式（如34+）相比，中缀表达式不容易被计算机解析，但仍被许多程序语言使用，因为它更符合人们的思维习惯。

与前缀表达式或后缀表达式不同的是，中缀表达式中的括号是必需的。计算过程中必须用括号将操作符和对应的操作数括起来，用于指示运算的次序。后缀表达式的运算符在操作数后面，后缀表达式中考虑了运算符的优先级，没有括号，只有操作数和运算符。

中缀表达式A+B*(C-D)-E/F对应的后缀表达式为ABCD-+EF/-*，将后缀表达式与原表达式对应的表达式树的后序遍历序列进行比较，可发现它们有异曲同工之妙。

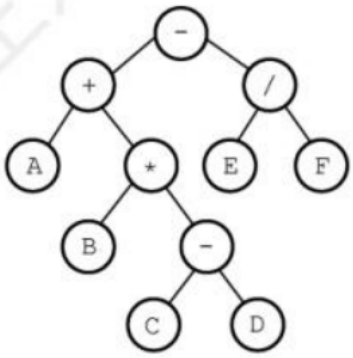

图 3.15　A+B*(C-D)-E/F 对应的表达式树

中缀表达式转后缀表达式

下面先给出一种由中缀表达式转后缀表达式的手算方法。

1. 按照运算符的运算顺序对所有运算单位加括号。
2. 将运算符移至对应括号的后面，相当于按“左操作数 右操作数 运算符”重新组合。
3. 去除所有括号。

例如，中缀表达式A+B*(C-D)-E/F转后缀表达的过程如下（下标表示运算符的运算顺序）：

1. 加括号：((A+(B*(C-D)))-(E/F))。
2. 运算符后移：((A(B(CD)-)*)+(EF)/)-
3. 去除括号后，得到后缀表达式：ABCD-*+EF/-

中缀表达式转后缀表达式的过程(2012,2014)

在计算机中，中缀表达式转后缀表达式时需要借助一个栈，用于保存暂时还不能确定运算顺序的运算符。从左到右依次扫描中缀表达式中的每一项，具体转化过程如下：

1. 遇到操作数。直接加入后缀表达式。
2. 遇到界限符。若为“(”，则直接入栈；若为“)”，则依次弹出栈中的运算符，并加入后缀表达式，直到弹出“(”位置。注意，“(”直接删除，不加入后缀表达式。
3. 遇到运算符。若其优先级高于除“(”外的栈顶运算符，则直接入栈。否则，从栈顶开始，依次弹出栈中优先级高于或等于当前运算符的所有运算符，并加入后缀表达式，直到遇到一个优先级低于它的运算符或遇到“(”时为止，之后将当前运算符入栈。

按上述方法扫描所有字符后，将栈中剩余运算符依次弹出，并加入后缀表达式。

例如，中缀表达式A+B*(C-D)-E/F转后缀表达式的过程如表所示。

表 3.1　中缀表达式 A+B*（C-D）-E/F 转后缀表达式的过程

步	待处理序列	栈内	后缀表达式	扫描项	说　明
1	A+B*(C-D)-E/F			A	A 加入后缀表达式
2	+B*(C-D)-E/F		A	+	+入栈
3	B*(C-D)-E/F	+	A	B	B 加入后缀表达式
4	*(C-D)-E/F	+	AB	*	*优先级高于栈顶，*入栈
5	(C-D)-E/F	+*	AB	(	(直接入栈
6	C-D)-E/F	+*(	AB	C	C 加入后缀表达式
7	-D)-E/F	+*(	ABC	-	栈顶为(，-直接入栈
8	D)-E/F	+*(-	ABC	D	D 加入后缀表达式
9	)-E/F	+*(-	ABCD	)	遇到)，弹出-，删除(
10	-E/F	+*	ABCD-	-	-优先级低于栈顶，依次弹出*、+，-入栈
11	E/F	-	ABCD-*+	E	E 加入后缀表达式
12	/F	-	ABCD-*+E	/	/优先级高于栈顶，/入栈
13	F	-/	ABCD-*+E	F	F 加入后缀表达式
14		-/	ABCD-*+EF		字符扫描完毕，弹出剩余运算符
			ABCD-*+EF/-		结束

栈的深度分析（2009、2012）

所谓栈的深度，是指栈中的元素个数，通常是给出进栈和出栈顺序，求最大深度（栈的容量应大于或等于最大深度）。有时会间接给出进栈和出栈序列，求最大深度（栈的容量应大于或等于最大深度）。有时会间接给出进栈和出栈序列，例如以中缀表达式和后缀表达式的形式给出进栈和出栈序列。掌握栈的先进后出的特点进行手工模拟是解决这类问题的有效方法。

后缀表达式求值

通过后缀表示计算机表达式值的过程：从左往右依次扫描表达式的每一项，若该项是操作数，则将其压入栈中；若该项是操作符<op>，则从栈中推出两个操作数Y和X，形成运算指令X<op>Y，并将计算结果压入栈中。当所有项都扫描并处理完后，栈顶存放的就是最后的计算结果。

例如，后缀表达式ABCD-*+EF/-求值的过程需要12步，如下表所示。

表 3.2　后缀表达式 ABCD-*+EF/-求值的过程

步	扫 描 项	项 类 型	动　作	栈中内容
1			置空栈	空
2	A	操作数	进栈	A
3	B	操作数	进栈	A B
4	C	操作数	进栈	A B C
5	D	操作数	进栈	A B C D
6	-	操作符	D、C 退栈，计算 C-D，结果 R_1 进栈	A B R_1
7	*	操作符	R_1、B 退栈，计算 B×R_1，结果 R_2 进栈	A R_2
8	+	操作符	R_2、A 退栈，计算 A+R_2，结果 R_3 进栈	R_3
9	E	操作数	进栈	R_3 E
10	F	操作数	进栈	R_3 E F
11	/	操作符	F、E 退栈，计算 E/F，结果 R_4 进栈	R_3 R_4
12	-	操作符	R_4、R_3 退栈，计算 R_3-R_4，结果 R_5 进栈	R_5

栈在递归中的应用

递归是一种重要的程序设计方法。简单地说，若在一个函数、过程或数据结构的定义中又应用了它自身，则这个函数、过程或数据结构称为是递归定义的，简称递归。

它通常把一个大型的复杂问题层层转化为一个与原问题相似的规模较小的问题来求解，递归策略只需少量的代码就可以描述出解题过程所需要的多次重复计算，大大减少了程序的代码量。但在通常情况下，它的效率并不是太高。

以斐波那契数列为例，其定义为

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

这就是递归的一个典型例子，用程序实现时如下：

```
int F(int n){//斐波那契数列的实现
    int(n==0)
        return 0;//边界条件
    else if(n==1)
        return 1;//边界条件
    else
        return F(n-1)+F(n-2);//递归表达式
}
```

必须注意递归模型不能是循环定义的，其必须满足下面的两个条件：

- 递归表达式（递归体）。
- 边界条件（递归出口）。

递归的精髓在于能否将原始问题转换为属性相同但规模较小的问题。

栈在函数调用中的作用和工作原理

在递归调用的过程中，系统为每一层的返回点、局部变量、传入实参等开辟了递归工作栈来进行数据存储，递归次数过多容易造成栈溢出等。而其效率不高的原因是递归调用过程中包含很多重复的计算。下面以n=5为例，列出递归调用执行过程，如下图所示。

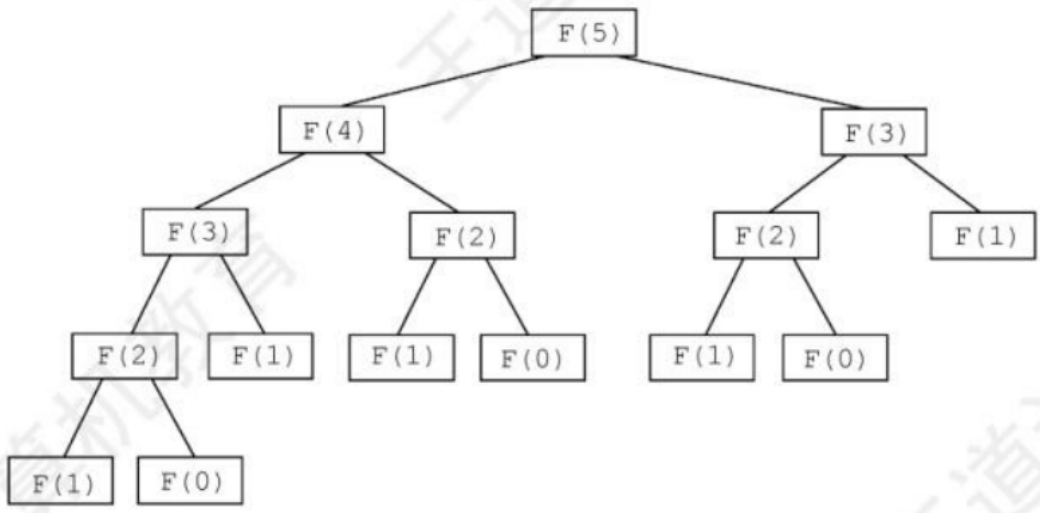

图 3.16　F(5) 的递归执行过程

显然，在递归调用的过程中，F(3)被计算2次，F(2)被计算3次。F(1)被调用5次，F(0)被调用3次。所以，递归的效率地下，但优点是代码简单，容易理解。在第5章的树中利用了递归的思想，代码变得十分简单。通常情况下，初学者很难理解递归的调用过程，若读者想具体了解递归是如何实现的，可以参阅编译原理教材中的相关内容。

可以将递归算法转换为非递归算法，通常需要借助栈来实现这种转换。

队列在层次遍历中的应用

在信息处理中有一大类问题需要逐层或逐行处理。这类问题的解决方法往往是在处理当前层或当前行就对下一层或下一行做预处理，把处理顺序安排好，等到当前层或当前行处理完毕，就可以处理下一层或下一行。使用队列是为了保存下一步的处理顺序。下面用二叉树层次遍历的例子，说明队列的应用。下表显示了层次遍历二叉树的过程。

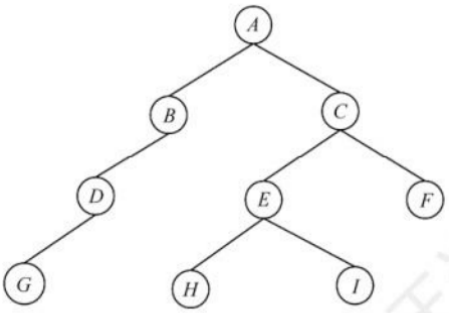

图 3.17　二叉树

表 3.3　层次遍历二叉树的过程

序	说　明	队内	队　外
1	*A* 入	*A*	
2	*A* 出，*BC* 入	*BC*	*A*
3	*B* 出，*D* 入	*CD*	*AB*
4	*C* 出，*EF* 入	*DEF*	*ABC*

（续表）

序	说　明	队内	队　外
5	*D* 出，*G* 入	*EFG*	*ABCD*
6	*E* 出，*HI* 入	*FGHI*	*ABCDE*
7	*F* 出	*GHI*	*ABCDEF*
8	*GHI* 出		*ABCDEFGHI*

该过程的简单描述如下：

1. 根结点入队。
2. 若队空（所有结点都已处理完毕），则结束遍历；否则重复3操作。
3. 队列中第一个结点出队，并访问之。若其有左孩子，则将左孩子入队；若其有右孩子，则将右孩子入队，返回2。

队列在计算机系统中的应用

队列在计算机系统中的应用非常广泛，以下仅从两个方面来阐述：第一个方面是解决主机与外部设备之间速度不匹配的问题，第二个方面是解决由多用户引起的资源竞争问题。

缓冲区的逻辑结构

对于第一个方面，仅以主机和打印机之间速度不匹配的问题为例做简要说明。主机输出数据给打印机打印，输出数据的速度比打印数据的速度要快得多，因为速度不匹配，若直接把输出的数据送给打印机打印，则显然是不行的。解决的方法是设置一个打印数据缓冲区，主机把要打印输出的数据依次写入这个缓冲区，写满后就暂停输出，转去做其他的事情。打印机就从缓冲区中按照先进先出的原则依次取出数据并打印，打印完后再向主机发出请求。主机接到请求后再向缓冲区写入打印数据。这样做既保证了打印数据的正确，又使主机提高了效率。由此可见，打印数据缓冲区中所存储的数据就是一个队列。

多队列出队/入队操作的应用

对于第二个方面，CPU（即中央处理器，它包括运算器和控制器）资源的竞争就是一个典型的例子。在一个带有多终端的计算机系统上，有多个用户需要CPU各自运行自己的程序，它们分别通过各自的终端向操作系统提出占用CPU的请求。操作系统通常按照每个请求在时间上的先后顺序，把它们排成一个队列，每次把CPU分配给队首请求的用户使用。当相应的程序运行结束或用完规定的时间间隔后，令其出队，再把CPU分配给新的队首请求的用户使用。这样既能满足每个用户的请求，又使CPU能够正常运行。