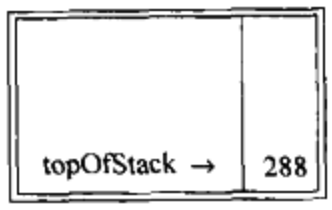
最后，遇到一个‘*’号，从栈中弹出 48 和 6；将结果 6 * 48 = 288 压进栈中



计算一个后缀表达式花费的时间是 $O(N)$，因为对输入中的每个元素的处理都是由一些栈操作组成从而花费常数的时间。该算法的计算是非常简单的。注意，当一个表达式以后缀记号给出时，没有必要知道任何优先的规则，这是一个明显的优点。

中缀到后缀的转换

栈不仅可以用来计算后缀表达式的值，而且还可以用栈将一个标准形式的表达式(或叫做中缀表达式(infix))转换成后缀式。我们通过只允许操作 +，*，(，)，并坚持普通的优先级法则而将一般的问题浓缩成小规模的问题。此外，还要进一步假设表达式是合法的。假设将中缀表达式

```
a+b*c+(d*e+f)*g
```

转换成后缀表达式。正确的答案是 abc*+de*f+g*+。

当读到一个操作数的时候，立即把它放到输出中。操作符不立即输出，从而必须先存在某个地方。正确的做法是将已经见到过但尚未放到输出中的操作符推入栈中。当遇到左圆括号时我们也要将其推入栈中。计算从一个空栈开始。

如果见到一个右括号，那么就将栈元素弹出，将弹出的符号写出直至遇到一个(对应的)左括号，但是这个左括号只被弹出并不输出。

如果我们见到任何其他的符号(+，*，( )，那么我们从栈中弹出栈元素直到发现优先级更低的元素为止。有一个例外：除非是在处理一个)的时候，否则我们决不从栈中移走(。对于这种操作，+ 的优先级最低，而(的优先级最高。当从栈弹出元素的工作完成后，我们再将操作符压入栈中。

最后，如果读到输入的末尾，我们将栈元素弹出直到该栈变成空栈，将符号写到输出中。

这个算法的想法是，当看到一个操作符的时候，把它放到栈中。栈代表挂起的操作符。然而，栈中有些具有高优先级的操作符现在知道当它们不再被挂起时要完成使用，应该被弹出。这样，在把当前操作符放入栈中之前，那些在栈中并在当前操作符之前要完成使用的操作符被弹出。详细的解释见下表：

表达式	在处理第 3 个操作符时的栈	动作
a*b-c+d	-	- 完成，+ 进栈
a/b+c*d	+	没有操作符完成操作，* 进栈
a-b*c/d	- *	* 完成，/ 进栈
a-b*c+d	- *	* 和 - 完成，+ 进栈

圆括号增加了额外的复杂因素。当左括号是一个输入符号时我们可以把它看成是一个高优先级的操作符(使得挂起的操作符仍是挂起的)，而当它在栈中时把它看成是低优先级的操作符(从而不会被操作符意外地删除)。右括号被处理成特殊的情况。

为了理解这种算法的运行机制，我们将把上面长的中缀表达式转换成后缀形式。首先，符号 a 被读入，于是它被传向输出。然后，‘+’被读入并被放入栈中。接下来 b 读入并流向输出。这一时刻的状态如下：