topOfStack 减 1。

注意，这些操作不仅以常数时间运行，而且是以非常快的常数时间运行。在某些机器上，若在带有自增和自减寻址功能的寄存器上操作，则(整数的)push 和 pop 都可以写成一条机器指令。最现代化的计算机将栈操作作为它的指令系统的一部分，这个事实强化了这样一种观念，即栈很可能是在计算机科学中在数组之后的最基本的数据结构。

3.6.3 应用

毫不奇怪，如果我们把操作限制在对一个表上进行，那么这些操作会执行得很快。然而，令人惊奇的是，这些少量的操作非常强大和重要。在栈的许多应用中，我们给出三个例子，第三个实例深刻说明程序是如何组织的。

平衡符号

编译器检查程序的语法错误，但是常常由于缺少一个符号(如遗漏一个花括号或是注释起始符)引起编译器列出上百行的诊断，而真正的错误并没有找出。(幸运的是，大部分 Java 编译器在这一点上是相当好的。但不是所有的语言和编译器都这么可靠。)

在这种情况下，一个有用的工具就是检验是否每件事情都能成对的程序。于是，每一个右花括号、右方括号及右圆括号必然对应其相应的左括号。序列[()]是合法的，但[( ])是错误的。显然，不值得为此编写一个大型程序，事实上检验这些事情是很容易的。为简单起见，我们仅就圆括号、方括号和花括号进行检验并忽略出现的任何其他字符。

这个简单的算法用到一个栈，叙述如下：

做一个空栈。读入字符直到文件结尾。如果字符是一个开放符号，则将其推入栈中。如果字符是一个封闭符号，则当栈空时报错。否则，将栈元素弹出。如果弹出的符号不是对应的开放符号，则报错。在文件结尾，如果栈非空则报错。

我们应该能够确信这个算法是会正确运行的。很清楚，它是线性的，事实上它只需对输入进行一趟检验。因此，它是联机(on-line)的，是相当快的。当报错时决定如何处理需要做一些附加的工作——例如判断可能的原因。

后缀表达式

假设我们有一个便携式计算器并想要计算一趟外出购物的花费。为此，我们将一列数据相加并将结果乘以 1.06；它是所购物品的价格以及附加的地方销售税。如果购物各项花销为 4.99、5.99 和 6.99，那么输入这些数据的自然的方式将是

$$4.99 + 5.99 + 6.99 * 1.06 =$$

随着计算器的不同，这个结果或者是所要的答案 19.05，或者是科学答案 18.39。最简单的四功能计算器都将给出第一个答案，但是许多先进的计算器是知道乘法的优先级高于加法的。

另一方面，有些项是需要上税的而有些项则不是，因此，如果只有第一项和最后一项是要上税的，那么计算的顺序

$$4.99 * 1.06 + 5.99 + 6.99 * 1.06 =$$

将在科学计算器上给出正确的答案(18.69)而在简单计算器上给出错误的答案(19.37)。科学计算器一般包含括号，因此我们总可以通过加括号的方法得到正确的答案，但是使用简单计算器我们需要记住中间结果。

该例的典型计算顺序可以是将 4.99 和 1.06 相乘并存为 A_1，然后将 5.99 和 A_1 相加，再将结果存入 A_1；我们再将 6.99 和 1.06 相乘并将答案存为 A_2，最后将 A_1 和 A_2 相加并将最后结果放入 A_1。我们可以将这种操作顺序书写如下：

$$4.99\ 1.06 * 5.99 + 6.99\ 1.06 * +$$