HW11 Problem Set

CS-3020

Assignment

Deitel & Deitel Exercises 19.6, 19.7, 20.12

HW11-1: (Deitel & Deitel Exercise 19.6)

19.6 ***(Evaluating Expressions with a Stack)*** Stacks are used by compilers to evaluate expressions and generate machine-language code. In this and the next exercise, we investigate how compilers evaluate arithmetic expressions consisting only of constants, operators and parentheses.

Humans generally write expressions like `3 + 4` and `7 / 9`, in which the operator (`+` or `/` here) is written between its operands—this is called *infix notation*. Computers "prefer" *postfix notation*, in which the operator is written to the right of its two operands. The preceding infix expressions would appear in postfix notation as `3 4 +` and `7 9 /`, respectively.

To evaluate a complex infix expression, a compiler would first convert the expression to postfix notation, then evaluate the postfix version of the expression. Each of these algorithms requires only a single left-to-right pass of the expression. Each algorithm uses a stack object in support of its operation, and in each algorithm the stack is used for a different purpose. Here, you'll implement the infix-to-postfix conversion algorithm. In the next exercise, you'll implement the postfix-expression evaluation algorithm.

Write class `InfixToPostfixConverter` to convert an ordinary infix arithmetic expression (assume a valid expression is entered), with single-digit integers, such as

```
(6 + 2) * 5 - 8 / 4
```

to a postfix expression. The postfix version of the preceding infix expression is

```
6 2 + 5 * 8 4 / -
```

The program should read the expression into `StringBuilder infix`, then use class `StackInheritance` (implemented in Fig. 19.13) to help create the postfix expression in `StringBuilder postfix`. The algorithm for creating a postfix expression is as follows:

a) Push a left parenthesis `'('` on the stack.
b) Append a right parenthesis `')'` to the end of `infix`.
c) While the stack is not empty, read `infix` from left to right and do the following:
    If the current character in `infix` is a digit, append it to `postfix`.
    If the current character in `infix` is a left parenthesis, push it onto the stack.
    If the current character in `infix` is an operator:
        Pop operators (if there are any) at the top of the stack while they have equal or higher precedence than the current operator, and append the popped operators to `postfix`.
        Push the current character in `infix` onto the stack.
    If the current character in `infix` is a right parenthesis:
        Pop operators from the top of the stack and append them to `postfix` until a left parenthesis is at the top of the stack.
        Pop (and discard) the left parenthesis from the stack.

The following arithmetic operations are allowed in an expression:

`+` addition
`-` subtraction
`*` multiplication
`/` division
`^` exponentiation
`%` modulus

Some of the methods you may want to provide in your program follow:

a) Method `ConvertToPostfix`, which converts the infix expression to postfix notation.
b) Method `IsOperator`, which determines whether `c` is an operator.
c) Method `Precedence`, which determines whether the precedence of `operator1` (from the infix expression) is less than, equal to or greater than the precedence of `operator2` (from the stack). The method returns `true` if `operator1` has lower precedence than or equal precedence to `operator2`. Otherwise, `false` is returned.

```
1  // Fig. 19.13: StackInheritanceLibrary.cs
2  // Implementing a stack by inheriting from class List.
3  using LinkedListLibrary;
4
5  namespace StackInheritanceLibrary
6  {
7     // class StackInheritance inherits class List's capabilities
8     public class StackInheritance : List
9     {
10       // pass name "stack" to List constructor
11       public StackInheritance()
12          : base( "stack" )
13       {
14       } // end constructor
15
```

Fig. 19.13 | Implementing a stack by inheriting from class `List`. (Part 1 of 2.)

```
16       // place dataValue at top of stack by inserting
17       // dataValue at front of linked list
18       public void Push( object dataValue )
19       {
20          InsertAtFront( dataValue );
21       } // end method Push
22
23       // remove item from top of stack by removing
24       // item at front of linked list
25       public object Pop()
26       {
27          return RemoveFromFront();
28       } // end method Pop
29    } // end class StackInheritance
30 } // end namespace StackInheritanceLibrary
```

Fig. 19.13 | Implementing a stack by inheriting from class `List`. (Part 2 of 2.)

HW11-2: (Deitel & Deitel Exercise 19.7)

19.7 ***(Evaluating a Postfix Expression with a Stack)*** Write class `PostfixEvaluator`, which evaluates a postfix expression (assume it is valid) such as

```
6 2 + 5 * 8 4 / -
```

The program should read a postfix expression consisting of digits and operators into a `StringBuilder`. Using the stack class from Exercise 19.6, the program should scan the expression and evaluate it. The algorithm (for single-digit numbers) is as follows:

a) Append a right parenthesis `')'` to the end of the postfix expression. When the right-parenthesis character is encountered, no further processing is necessary.

b) When the right-parenthesis character has not been encountered, read the expression from left to right.

If the current character is a digit, do the following:

Push its integer value on the stack (the integer value of a digit character is its value in the computer's character set minus the value of `'0'` in Unicode).

Otherwise, if the current character is an *operator*:

Pop the two top elements of the stack into variables `x` and `y`.

Calculate `y` *operator* `x`.

Push the result of the calculation onto the stack.

c) When the right parenthesis is encountered in the expression, pop the top value of the stack. This is the result of the postfix expression.

[*Note:* In *Part b* above (based on the sample expression at the beginning of this exercise), if the operator is `'/'`, the top of the stack is `4` and the next element in the stack is `8`, then pop `4` into `x`, pop `8` into `y`, evaluate `8 / 4` and push the result, `2`, back on the stack. This note also applies to operator `'-'`.] The arithmetic operations allowed in an expression are:

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`^`	exponentiation
`%`	modulus

You may want to provide the following methods:

a) Method `EvaluatePostfixExpression`, which evaluates the postfix expression.

b) Method `Calculate`, which evaluates the expression `op1` *operator* `op2`.

HW11 Problem Set

CS-3020

HW11-3: (Deitel & Deitel Exercise 20.12)

20.12 *(Generic Classes* ***TreeNode*** *and* ***Tree****)* Convert classes `TreeNode` and `Tree` from Fig. 19.20 into generic classes. To insert an object in a `Tree`, the object must be compared to the objects in existing `TreeNodes`. For this reason, classes `TreeNode` and `Tree` should specify `IComparable<T>` as the interface constraint of each class's type parameter. After modifying classes `TreeNode` and `Tree`, write a test app that creates three `Tree` objects—one that stores `ints`, one that stores `doubles` and one that stores `strings`. Insert 10 values into each tree. Then output the preorder, inorder and postorder traversals for each `Tree`.

Grading Rubric

Each problem is worth 10 pts (score will be recorded as a percentage of that amount)

10% Properly submitted
10% Properly named
20% Adequate comments
10% Runs
20% Produces correct output
30% Effort evidenced by the submitted work