**Evaluate postfix expressions:**

1. **public** **int** **evaluate()**
2. **{**
3. **Stack** theStack = **new** **Stack(**50**)**;
4. **int** result;
6. **for(int** j = 0; j < input.length**()**; j++**)**
7. **{**
8. **char** ch = input.charAt**(**j**)**;
9. **switch(**ch**)**
10. **{**
11. **case** '0':
12. **case** '1':
13. **case** '2':
14. **case** '3':
15. **case** '4':
16. **case** '5':
17. **case** '6':
18. **case** '7':
19. **case** '8':
20. **case** '9':
21. theStack.push( ch **)**;
22. **case** '+':
23. num2 = theStack.pop**()**;
24. num1 = theStack.pop**()**;
25. result = num1 + num2;
26. **break**;
27. **case** '-':
28. num2 = theStack.pop**()**;
29. num1 = theStack.pop**()**;
30. result = num1 - num2;
31. **break**;
32. **case** '\*':
33. num2 = theStack.pop**()**;
34. num1 = theStack.pop**()**;
35. result = num1 \* num2;
36. **break**;
37. **case** '/':
38. num2 = theStack.pop**()**;
39. num1 = theStack.pop**()**;
40. result = num1 / num2;
41. **break**;
42. **case** '%':
43. num2 = theStack.pop**()**;
44. num1 = theStack.pop**()**;
45. result = num1 % num2;
46. **break**;
47. **default**:
48. result = 0;
49. **}**//end SWITCH
50. theStack.push**(** result**)**;
51. **}**//end FOR
52. result = theStack.pop**()**; // at the end only once
53. **return** result;
55. **}**

**Example :**   
  
Let us see how the above algorithm will be imlemented using an example.   
  
Postfix String : 123\*+4-   
  
Initially the Stack is empty. Now, the first three characters scanned are 1,2 and 3, which are operands. Thus they will be pushed into the stack in that order.

|  |  |
| --- | --- |
| http://scriptasylum.com/tutorials/infix_postfix/algorithms/postfix-evaluation/img1.gif **Stack** | http://scriptasylum.com/tutorials/infix_postfix/algorithms/postfix-evaluation/img2.gif **Expression** |

Next character scanned is "\*", which is an operator. Thus, we pop the top two elements from the stack and perform the "\*" operation with the two operands. The second operand will be the first element that is popped.

|  |  |
| --- | --- |
| http://scriptasylum.com/tutorials/infix_postfix/algorithms/postfix-evaluation/img3.gif **Stack** | http://scriptasylum.com/tutorials/infix_postfix/algorithms/postfix-evaluation/img4.gif **Expression** |

The value of the expression(2\*3) that has been evaluated(6) is pushed into the stack.

|  |  |
| --- | --- |
| http://scriptasylum.com/tutorials/infix_postfix/algorithms/postfix-evaluation/img5.gif **Stack** | http://scriptasylum.com/tutorials/infix_postfix/algorithms/postfix-evaluation/img2.gif **Expression** |

Next character scanned is "+", which is an operator. Thus, we pop the top two elements from the stack and perform the "+" operation with the two operands. The second operand will be the first element that is popped.

|  |  |
| --- | --- |
| http://scriptasylum.com/tutorials/infix_postfix/algorithms/postfix-evaluation/img7.gif **Stack** | http://scriptasylum.com/tutorials/infix_postfix/algorithms/postfix-evaluation/img8.gif **Expression** |

The value of the expression(1+6) that has been evaluated(7) is pushed into the stack.

|  |  |
| --- | --- |
| http://scriptasylum.com/tutorials/infix_postfix/algorithms/postfix-evaluation/img9.gif **Stack** | http://scriptasylum.com/tutorials/infix_postfix/algorithms/postfix-evaluation/img2.gif **Expression** |

Next character scanned is "4", which is added to the stack.

|  |  |
| --- | --- |
| http://scriptasylum.com/tutorials/infix_postfix/algorithms/postfix-evaluation/img11.gif **Stack** | http://scriptasylum.com/tutorials/infix_postfix/algorithms/postfix-evaluation/img2.gif **Expression** |

Next character scanned is "-", which is an operator. Thus, we pop the top two elements from the stack and perform the "-" operation with the two operands. The second operand will be the first element that is popped.

|  |  |
| --- | --- |
| http://scriptasylum.com/tutorials/infix_postfix/algorithms/postfix-evaluation/img7.gif **Stack** | http://scriptasylum.com/tutorials/infix_postfix/algorithms/postfix-evaluation/img14.gif **Expression** |

The value of the expression(7-4) that has been evaluated(3) is pushed into the stack.

|  |  |
| --- | --- |
| http://scriptasylum.com/tutorials/infix_postfix/algorithms/postfix-evaluation/img15.gif **Stack** | http://scriptasylum.com/tutorials/infix_postfix/algorithms/postfix-evaluation/img2.gif **Expression** |

Now, since all the characters are scanned, the remaining element in the stack (there will be only one element in the stack) will be returned.   
  
End result :

* Postfix String : 123\*+4-
* Result : 3
* **Evaluate expressions:**
* **Below are evaluations of the same expression in three different notations. Time "flows" from top to bottom.   
          infix notation          prefix notation          postfix notation   
  0:      (1 + (2 \* 3)            +  1  \*  2  3            1  2  3  \*  +   
  1:      (1 +    6  )            +  1  6                  1        6  +   
  2:         7                    7                                    7   
  The basic properties of rewriting are well-known. At each step the algorithm works by (1) inspecting parts of an expression, and then (2) replacing that part by a value. So the method actually consumes the original expression. Note also that all rewriting is in-place.   
  For larger expressions there may be several places where rewriting could take place, and the rewriting could then even proceed concurrently. For infix and prefix expressions it may be necessary to search to the left or to the right to find a suitable expressions to evaluate next. For postfix it is always possible to restrict the search to the right, this is used in the stack machine below.   
  Evaluating expressions by a queue machine   
  Here is a method which uses the prefix expression as a queue. Time again flows "flows" from top to bottom.   
  0:            +  1  \*  2  3   
  1:                 1  \*  2  3  +                   (2b)   
  2:                     \*  2  3  +  1               (2c)   
  3:                                  +  1  6          (2a)   
  4:                                           7         (2a)   
    
  The algorithm is quite different from in-place rewriting in the first section and also different from the stack-rewriting in the second section.   
    
  WHILE the queue has more than one element DO   
    
      (1)  Inspect the front element.   
      (2a) If it is an operator followed by all its arguments,   
           append the value to the rear of the queue AND THEN   
           remove operator and arguments from the front of the queue   
      (2b) If it is an operator not followed by all its arguments,   
           remove it from the front, copy it to the rear.   
      (2c)  (it is an operand)   
           remove it from the front, copy it to the rear.   
  The algorithm resembles rewriting by starting with a prefix expression and ending with a value, but the rewriting is not in-place. The algorithm resembles the stack machine by successively examining the next symbol of the expression, but it does not use an auxiliary structure. The significance of the "AND THEN" emphasis in step (2a) will become apparent shortly.   
  In the same way, it is just as simple to evaluate a postfix expression as a queue - merely by reading the queue from right to left.   
  0:                        1  2  3  \*  +   
  1:                    +  1  2  3  \*             (2b)   
  2:                6  +  1                         (2a)   
  3:            1  6  +                             (2c)   
  4:          7                                        (2a)   
  Of course postfix and prefix are not just reverses of each other, as is seen by examples with non-commutative operators such as subtraction or division. But as the two examples demonstrate, there is no inherent connection between stack evaluation and postfix, or between queue evaluation and prefix. Any such connection is just by convention. This being said, I now follow this convention and speak of stack machines being driven by postfix expressions. In the same way I shall now take queue machines to be certain rewritings of prefix expressions.   
  For any particular prefix evaluation the efficiency will depend on the ratio of the (2a) steps that really do the work, and the (2b) and (2c) bookkeeping steps that are at best a nuisance. The ratio depends on the patters of (binary) operators b and values v. In the example, of the form   
                  b  v  b  v  v   
  the ratio was 2:1:1. For expressions of the form   
                  b  b  v  v  b  v  v   
  the ratio is 3:1:0 which is better. For expressions of the form   
                  b  b  v  b  v  v  v   
  the ratio is 3:3:3 which is worse. In general, the algorithm is best for expressions representing a balanced binary tree. Since most expressions depart from this ideal, the entire method would seem to be barely more than a cute but silly curiosity.   
  But let us recapitulate: the other methods use in-place rewriting of either an original infix or prefix or postfix expression, or of a separate stack. For word-size values such as integers this hardly matters because (1) integer operations are indivisible single machine operations, and (2) the result will always fit into the space that has been vacated by the operands. The queue machine is also a rewriting method, but it is not in-place. This has a number of repercussions, to be discussed below.**