Chapter 7 The Stack

In this chapter we examine what is arguably the most important abstract data type in computer science, the *stack*. We will see that the stack ADT and its implementation are very simple. The stack's simplicity is misleading, however, for it has a variety of sophisticated applications.

7.1 Our Current Model

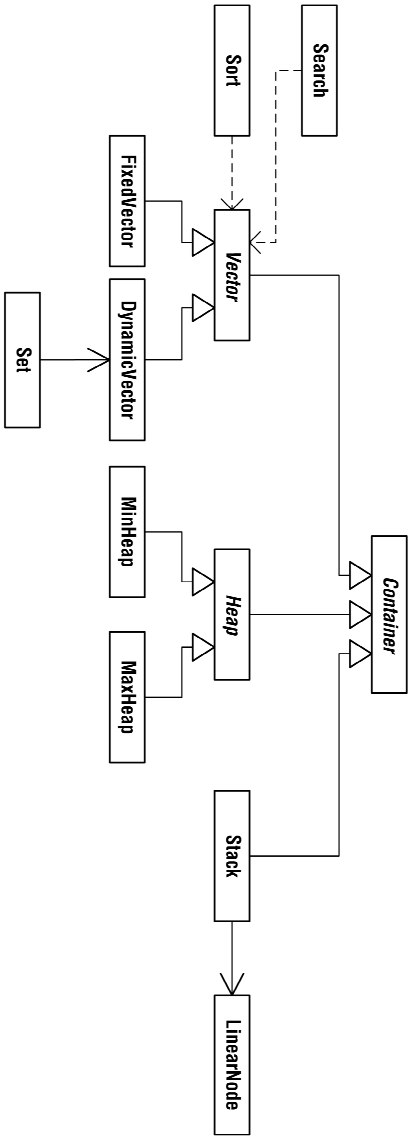


Fig. 7.1. Our current software model, which shows class `Stack` and its association with class `LinearNode`

7.2 The Stack ADT

The stack is so named because it behaves like a deck of playing cards: the current card of interest is the top card, and no other card may be examined or removed. In other words, the stack is a *last-in-first-out* (*LIFO*) collection. The stack ADT appears below.

```
stack: a linear collection of elements that can be accessed at
       only one end; the current element of interest is called the
       top element

operations:

          clear() - Make the collection empty.
        isEmpty() - Is the collection empty?
            pop() - Remove the top element.
    push(element) - Make the given element the top element.
           size() - How many elements are in the collection?
            top() - Get the top element of the collection.
```

7.3 Stack Implementation

The stack ADT can easily be implemented as an array container in which the top element is either stored at index 0 or at index *n* - 1, where *n* is the size of the collection. If the top element is stored at index 0, then the `push` and `pop` operations have linear running times, for the rest of the collection's elements must be shifted to make room for the new top or to remove the old one. If the top element is stored at index *n* - 1, then `push` and `pop` are constant-time operations, and so the array-based stack is customarily implemented in this way.

But the fundamental characteristic of the array is efficient random access. The vector is a random-access collection, and so it makes sense to have implemented the vector as an array container, as we did in Chapters 3 and 4. But the stack is not a random-access collection. The heap is also not a random-access collection, but we saw in the last chapter that the complete binary tree lends itself to an array implementation. Might the same be said of the stack? In other words, do we gain anything by implementing the stack ADT using an array? It is hard to see what a stack implementation might gain from the random access of an array, but it is easy to see what the implementation would lose: an array container often wastes space in the form of unused array elements. Because the stack is not random access, because an array-based stack implementation cannot exploit the array's random access, and because an array-based implementation wastes

space, we will implement the stack ADT using an alternative structure, the *linked list*.

linked list: a linear collection of nodes in which each node has, at a minimum, a data portion and a link portion; a node's link portion is a pointer or reference to the next node in the list, unless there is no next node, in which case the link portion is empty

7.3.1 Linked Lists

The figure below shows a linked list containing the integer values 3, 7, 25, 42, and 12. The absence of an arrow pointing out of the last node indicates an empty link, i.e., the end of the list. The first node of a linked list is kept track of using a reference or pointer called `head`, and sometimes the last node is kept track of using a reference or pointer called `tail`.

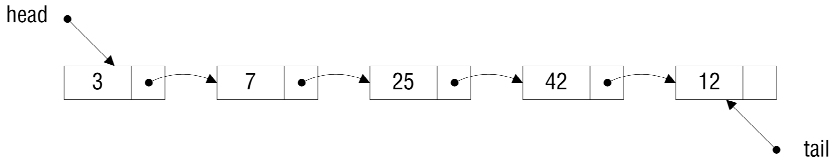


Fig. 7.2. An example linked list

The nodes of a linked list are not stored in contiguous blocks of memory, hence the links. Only the memory addresses of the first and last nodes are stored outside the list; the memory address of any internal node is stored in another node, namely the predecessor. This means that a given internal node can be accessed only by navigating to that node by way of the links, starting with `head`. Thus a linked list is a *sequential-access* structure, like an audio or video cassette.

7.3.2 Class `LinearNode` and Some Elementary Linked-List Operations

Let us now examine a few elementary linked-list operations, some of which we will use in our stack implementation. All of our examples will make use of the `LinearNode` class shown below. Observe that both the `data` and `next` fields of a `LinearNode` are declared public. This may appear to violate a cardinal rule of object-oriented design and development, but it does not. We will never use a linked list outside a container.

The container will declare `head` and `tail` as protected, and access to the list will be restricted to the methods of the container.

```
public class LinearNode
{
    public Object data;
    public LinearNode next;

    // The following constructors are obviously unnecessary,
    // considering that 'data' and 'next' are public. These
    // constructors are provided for the sake of convenience.

    public LinearNode(Object dat)
    {
        data = dat;
    }

    public LinearNode(Object dat, LinearNode nxt)
    {
        data = dat;
        next = nxt;
    }
}
```

Creating an Empty Linked List

Creating an empty linked list is easy. Simply declare a head reference and a tail reference and set them to null.

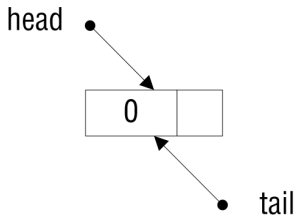
```
LinearNode head = null,
           tail = null;
```

Inserting into an Empty Linked List

To insert an element into an empty linked list, we must instantiate a node, initialize its `data` field, place null in its `next` field, and assign its memory address to `head` and `tail`. The first `LinearNode` constructor will allow us to do all this in one statement.

```
// Create a one-node linked list to store the integer 0.

head = tail = new LinearNode(new Integer(0));
```

The above code creates the linked list shown below.

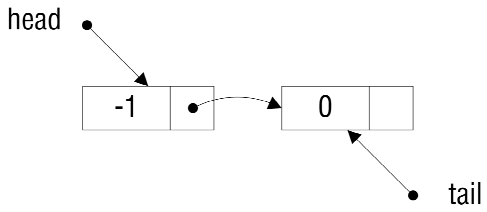


Inserting at the Head or the Tail of a Nonempty Linked List

The code below inserts a new node containing the integer value -1 at the head of the linked list shown above. The second `LinearNode` constructor allows us to do the insertion with a single statement.

```
// Because the assignment to 'head' is carried out after the new
// node has been created, the value of 'head' that is passed to
// the constructor is the memory address of the old head node, the
// node containing 0.

head = new LinearNode(new Integer(-1), head);

// Here is a three-step alternative.

// Instantiate a new node with an empty link.

LinearNode temp = new LinearNode(new Integer(-1));

temp.next = head;   // Link the new node to the head node.
head = temp;        // Make 'head' point at the new node.
```

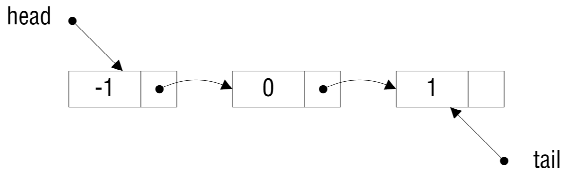
The new list is shown below.



Insertion at the tail can also be accomplished with a single statement. The code below inserts a node containing 1 at the tail of the above list.

```
tail = tail.next = new LinearNode(new Integer(1));

// Here is a two-step alternative.

tail.next = new LinearNode(new Integer(1));

tail = tail.next;  // Move 'tail' ahead to the new node.
```

The resulting list appears below.

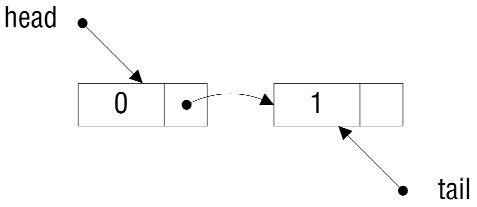


Removing the Head Node of a Linked List

Removing the head node of a linked list amounts to moving `head` forward to the second node, provided that a second node exists. The special case of a one-node list requires that `tail` be updated as well. The code to remove the head node of the above list is shown below.

```
/*
Move 'head' forward. If the list had two or more nodes, then
'head' now points to the second node. If the list had only one
node, then 'head' is now equal to null. In either case, no
reference to the old head node remains, and so that node can be
reclaimed by the garbage collector.
*/

head = head.next;

// If 'head' is now equal to null, then the list had only one
// node. The result of the removal should be an empty list, and so
// 'tail' must be set to null.

if (head == null)
    tail = null;
```

The figure below shows the list that results from removing the head node of the above list.



7.3.3 A Linked-List Stack Implementation

We now implement the stack ADT as a linked-list container. Class `Stack` inherits from class `Container` and uses the `LinearNode` class defined above. A `Stack`'s top element is stored in the head node.

```
public class Stack extends Container
{
    protected LinearNode head;  // We need only a head reference.
                                // There is no use for a tail
                                // reference because the top
                                // element is at the head of the
                                // list.

    /*
    Method clear() is inherited from Container, but no linked list
    is present at that level of abstraction. Thus class Stack
    overrides the superclass method so as to make the linked list
    ready for garbage collection.

    The list is prepared for garbage collection by setting 'head'
    to null. After 'head' has been set to null, the linked list
    can no longer be accessed. The unreachable list is recognized
    as such by the garbage collector and is reclaimed.
    */

    public void clear()
    {
        super.clear();  // Call Container.clear.
        head = null;    // Prepare the list for garbage
    }                   // collection.

    /*
     precondition: If the collection is empty this method has
                   nothing to do, in which case it returns null.
    postcondition: The top element has been removed from the
                   collection and has been returned.
    */

    public Object pop()
    {
        if (isEmpty())
            return null;
        Object element = head.data;  // Save the top element.
        head = head.next;            // Move 'head' forward.
        numItems--;
        return element;              // Return the top element.
    }

    /*
     precondition: N/A
    postcondition: The given element is the new top element.
    */

    public void push(Object element)
    {
        if (isEmpty())
            head = new LinearNode(element);
        else
            head = new LinearNode(element, head);
        numItems++;
    }
```

```
    /*
     precondition: If the collection is empty it has no top
                   element, in which case this method returns
                   null.
    postcondition: The top element has been returned.
    */

    public Object top()
    {
        if (isEmpty())
            return null;
        return head.data;
    }
}
```

7.4 Stack Applications

The stack ADT has many applications, including argument passing, pushdown automata and Turing machines, and expression evaluation. Argument passing is the subject of the next chapter, and the theory of computation is beyond the scope of this book. Thus we turn to a discussion of expression evaluation in the context of *constant folding*, a type of optimization that is done by programming-language compilers.

7.4.1 Constant Folding

Consider the following Java statement.

```
// Compute the volume of a sphere of radius 2.

double volume = 4.0 / 3 * Math.PI * 2 * 2 * 2;
```

A non-optimizing Java compiler might generate byte code for this statement that is roughly equivalent to the following pseudocode.

```
fld 3.0         // Load 3.0.
fld 4.0         // Load 4.0.
fdiv            // Compute 4.0 / 3.0.
fld Math.PI     // Load Math.PI.
fmul            // Compute 4.0 / 3.0 * Math.PI.
fld 2.0         // Load 2.0.
fmul            // Compute 4.0 / 3.0 * Math.PI * 2.0.
fld 2.0         // Load 2.0.
fmul            // Compute 4.0 / 3.0 * Math.PI * 2.0 * 2.0.
fld 2.0         // Load 2.0.
fmul            // Compute 4.0 / 3.0 * Math.PI * 2.0 * 2.0 * 2.0.
fstp volume     // Store the result in 'volume'.
```

The preceding byte code is far from optimal. The expression to be evaluated consists only of constants, the values of which are obviously

available to the Java compiler. If the Java compiler were to evaluate the expression, rather than generating byte code to compute the value at run time, the resulting program would see a considerable decrease in its execution time, for the two instructions shown below will surely execute much faster than the previous 12 instructions.

```
fld 33.5
fstp volume
```

The compile-time evaluation of constant expressions is called constant folding. Constant folding can be accomplished using the stack ADT.

7.4.2 Using a Stack to Evaluate an Arithmetic Expression

Our approach to expression evaluation will have two steps:

1. Use a stack to convert an *infix* expression to a *postfix* expression.
2. Use a stack to evaluate the postfix expression that resulted from step 1.

In an infix expression, a binary operator appears between its two operands. In a postfix expression, a binary operator appears after its two operands. Below are some examples of infix expressions along with their corresponding postfix expressions. The ^ symbol denotes exponentiation.

Table 7.1. Some example infix arithmetic expressions and their postfix equivalents

Infix Expression	Postfix Expression
3 + 6	3 6 +
(3 + 6) % 7	3 6 + 7 %
((3 + 6) % 7) ^ 5	3 6 + 7 % 5 ^
((3 + 6) % 7) ^ 5 - 1	3 6 + 7 % 5 ^ 1 -
3 + 6 % 7 ^ 5 - 1	3 6 7 5 ^ % + 1 -

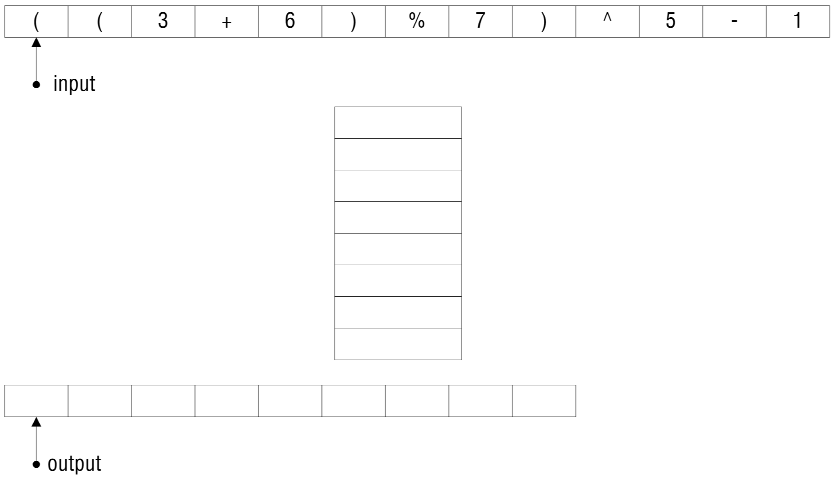
Notice that the order of an infix expression's operands is preserved in its postfix expression. The order of the operators may change, however, depending on their precedence. Postfix expressions do not require parentheses because the precedence of an operator in a postfix expression is implied by the ordering of the expression's operands and operators.

Infix-to-Postfix Conversion

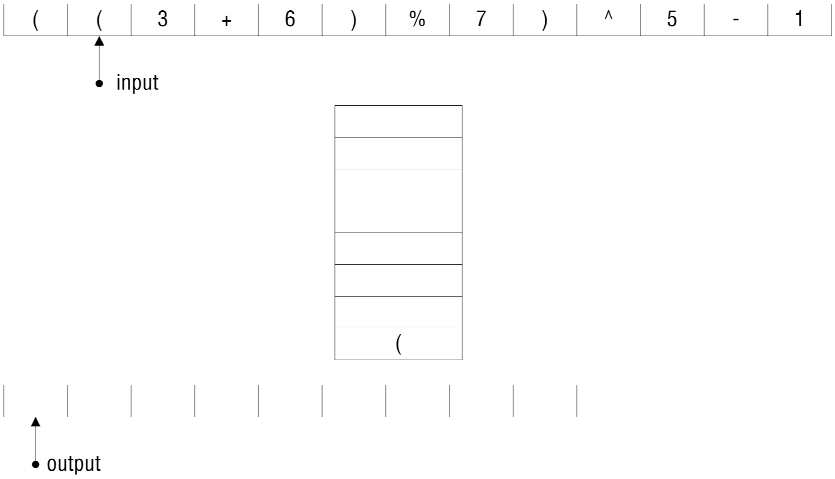
The infix-to-postfix conversion algorithm is specified below.

```
CONVERTING AN INFIX EXPRESSION TO A POSTFIX EXPRESSION

1. Initialize a stack of characters to hold the operation symbols
   and parentheses.

2. do
       if (the next input is a left parenthesis)
           Read the parenthesis and push it onto the stack.
       else if (the next input is a number or other operand)
           Read the operand and write it to the output.
       else if (the next input is an operation symbol)
       {
           Print the top operation and pop it; keep doing this
           until one of the following occurs.

           (1) The stack becomes empty.
           (2) The stack's top item is a left parenthesis.
           (3) The stack's top item is an operation with lower
               precedence than the next input.

           When one of these situations occurs, stop popping, read
           the next input, and push the input onto the stack.
       }
       else
       {
           Read and discard the next input, a right parenthesis.
           Print the top operation and pop it; keep printing and
           popping until the next symbol on the stack is a left
           parenthesis. (If no left parenthesis is found, print an
           error message announcing unbalanced parentheses and
           halt.) Finally, pop the left parenthesis.
       }
   while (there is more input to read);

3. Print and pop any operations that remain on the stack. (There
   should be no left parentheses on the stack; otherwise, the
   input expression did not have balanced parentheses.)
```

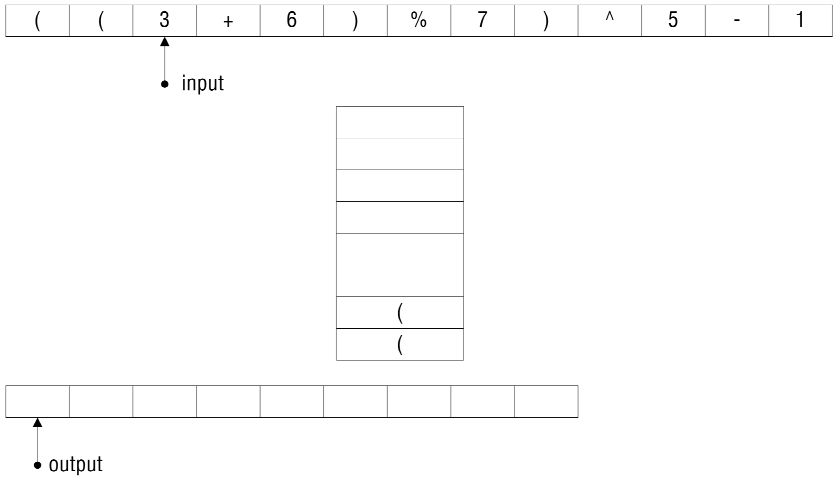
Let us apply the algorithm to the infix expression ((3 + 6) % 7) ^ 5 - 1. The initial states of our input, our stack, and our output are shown below.



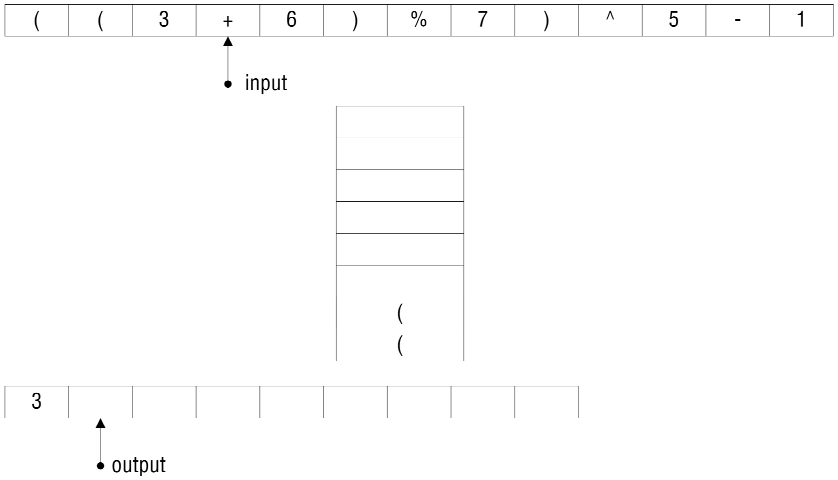
The first character of input is a left parenthesis. According to the algorithm, the left parenthesis should be pushed onto the stack. The pointer into the input is then moved forward, leaving the system in the state shown below.



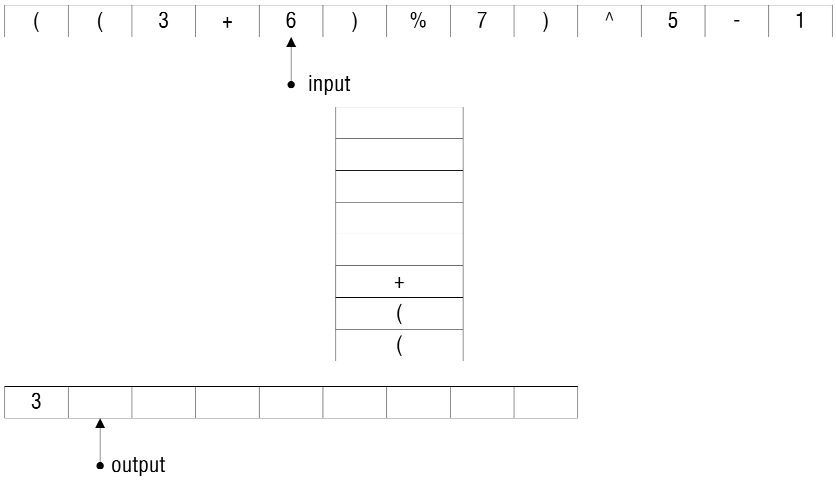
The next character of input is another left parenthesis. The parenthesis is pushed onto the stack and the input pointer is again moved forward. The following diagram shows the system's new state.



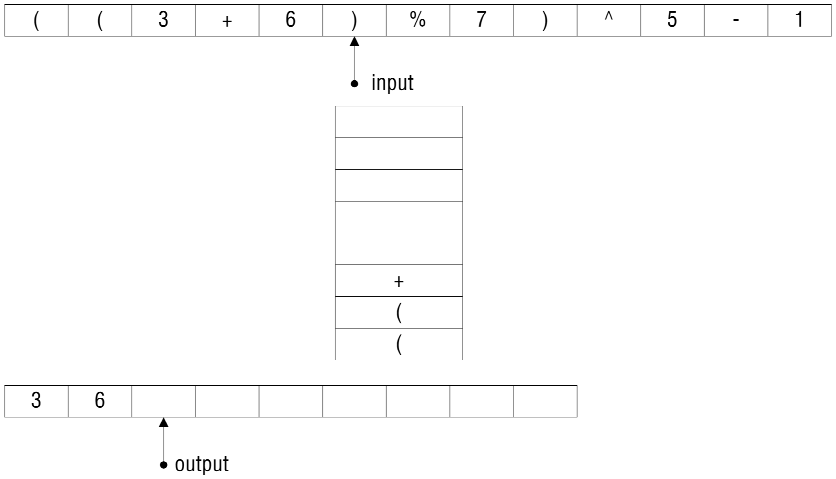
The next input character is an operand. Any operand is immediately written to the output. Both the input and output pointers are then moved forward.



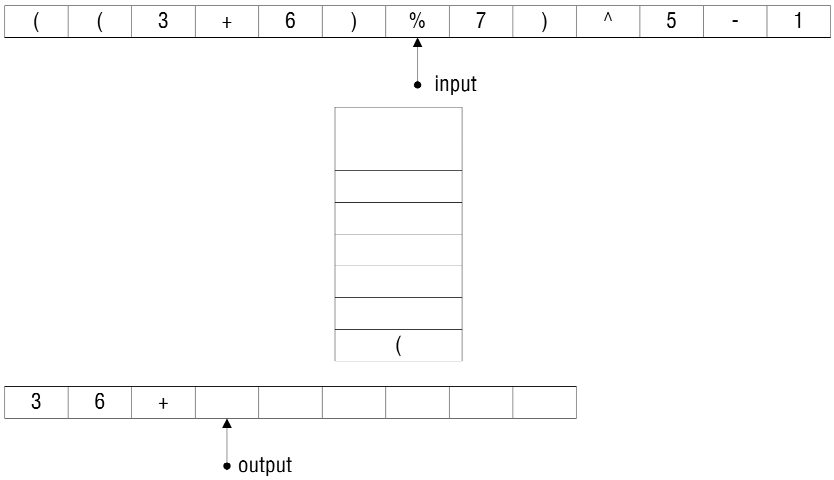
The algorithm now encounters an operator. There is no operator on top of the stack, and so the input operator, +, is immediately pushed onto the stack and the input pointer is moved forward.



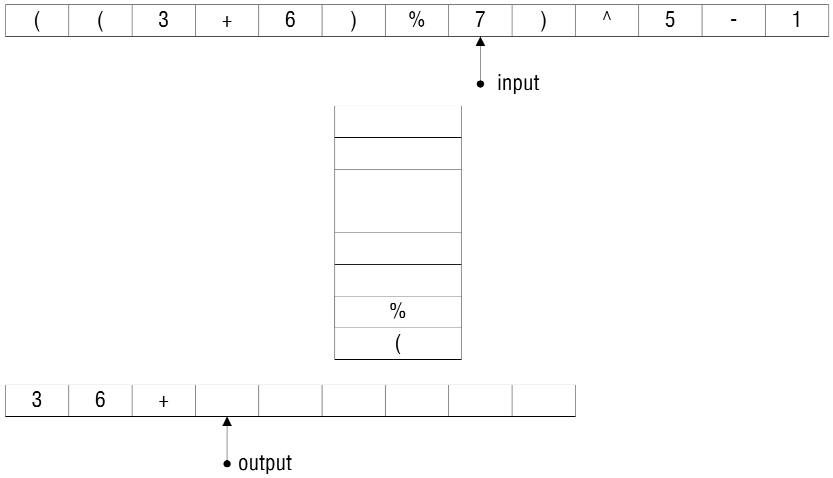
The next input, an operand, is written to the output string.



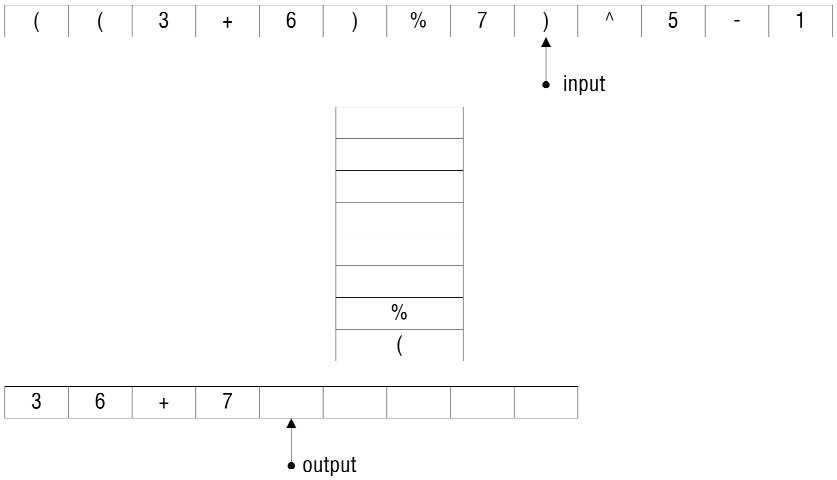
The right parenthesis is now read and discarded. Then the stack's top operator is popped and printed, after which the top left parenthesis is popped and discarded.



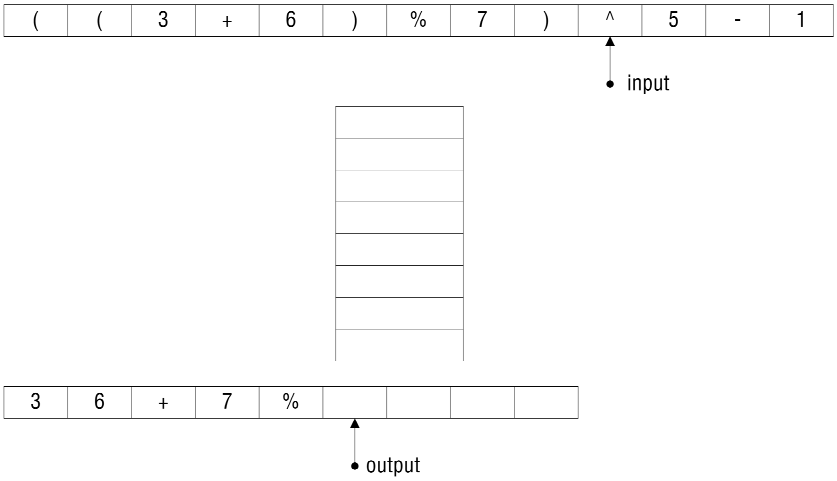
The next character of input is another operator. There are no operators on top of the stack, and so the % operator is immediately pushed.



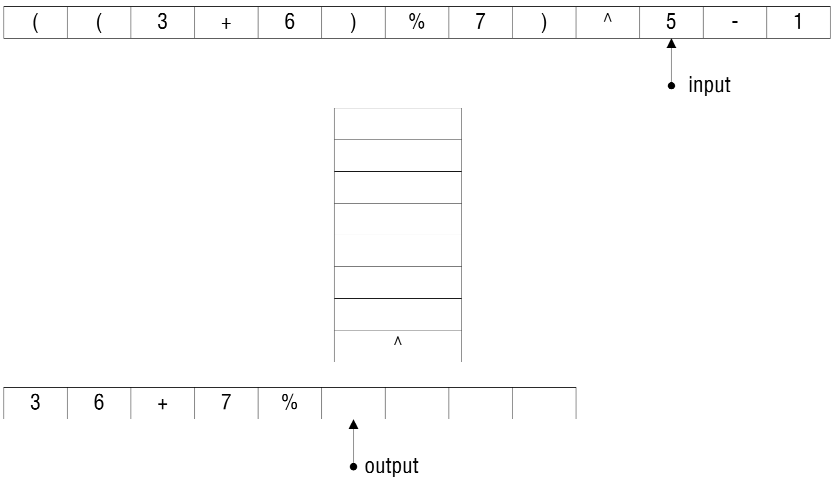
The next input, an operator, gets written to the postfix string.



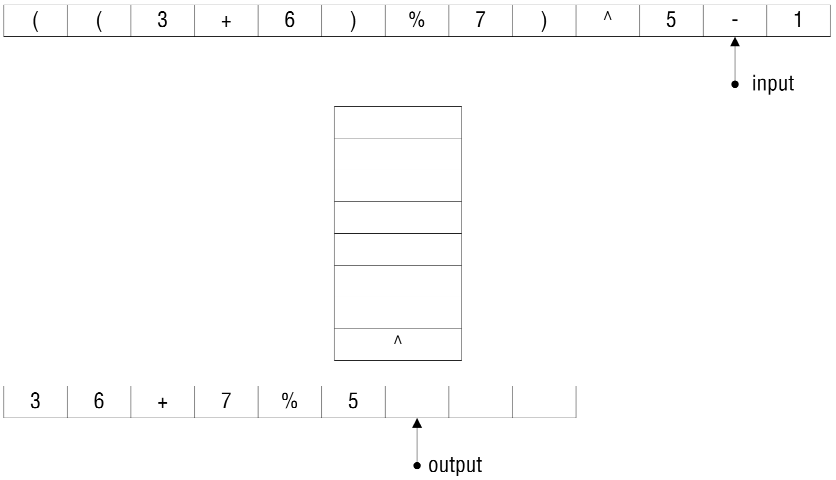
The right parenthesis is now read and discarded, after which the stack's top operator is popped and printed to the output string. The right parenthesis's matching left parenthesis is then popped and discarded.



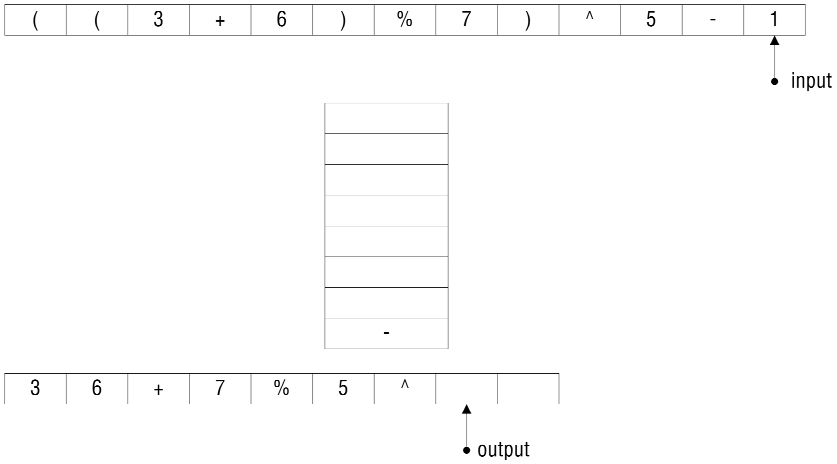
The next input character is an operator and the stack is empty. Consequently, the operator gets pushed.



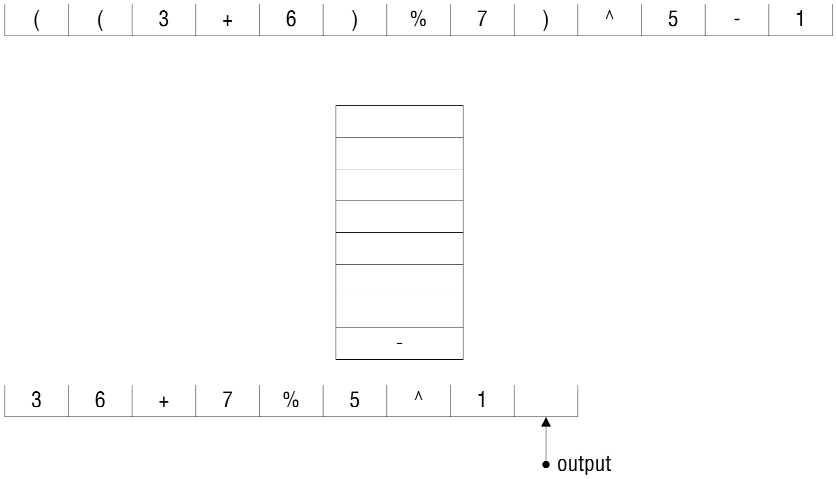
The next input, 5, is an operand. It gets written to the postfix string.



The next input is the subtraction operator. The stack's top symbol is an operator with higher precedence, and so the ^ operator must be popped and written to the postfix string before the - operator gets pushed.



The remaining input, 1, is an operand. The 1 is written to the output.



Having reached the end of the input string, the algorithm's do-while loop terminates. According to step 3, any operators that remain on the stack should be popped and written to the output. Thus the - operator is popped and written. The stack is now empty, which indicates that the infix expression had balanced parentheses.

(	(	3	+	6	)	%	7	)	^	5	-	1

3	6	+	7	%	5	^	1	-

Postfix Evaluation

The postfix evaluation algorithm is specified below. Apply it to the postfix expression above.

```
EVALUATING A POSTFIX EXPRESSION

1. Initialize a stack to hold the operands.

2. do
       if (the next input is a number)
           Read the next input and push it onto the stack.
       else
       {
           Read the next input, an operation symbol. Get the top
           two numbers off of the stack. Apply the operation to
           the two numbers, making sure to use the first number
           off as the right operand and the second number off as
           the left operand. Push the result onto the stack.
       }
   while (there is more input to read);

3. The stack now contains one number, the value of the expression.
```

Exercises

1. Add attributes and operations to the class diagram shown at the beginning of the chapter.
2. Perform a space analysis of the array and the linked list. Hint: Assume that Java is the target programming language and that a Java reference occupies *k* bytes. For a collection of size *n*, how many array elements must go unused for an array's space requirement to exceed that of a linked list?
3. Design and code a program to identify palindromes.
4. Design and code a program that prompts its user for an infix expression, reads the expression, converts the expression to postfix, evaluates the postfix expression, and displays the result. Your program should handle only nonnegative single-digit integer operands and the integer operators +, -, *, /, %, and ^. Be sure not to divide by zero.