Chapter 7

Stack

Overview

- The stack data structure uses an underlying linear storage organization.
  - The stack is one of the most ubiquitous data structures in computing.

Learning Objectives

- Describe the behavior of a stack.
- Enumerate the primary operations supported by the stack.
- Examine several applications of the stack, including parentheses matching, evaluating postfix expressions, and the conversion of an infix expression to postfix form.
- Understand the public interface of a stack class in Java and the running times of its methods.

Learning Objectives

- Develop a postfix package in Java to implement postfix expression evaluation.
- Study the implementation of a stack class in Java, and the trade-offs involved in choosing between candidate reusable components.

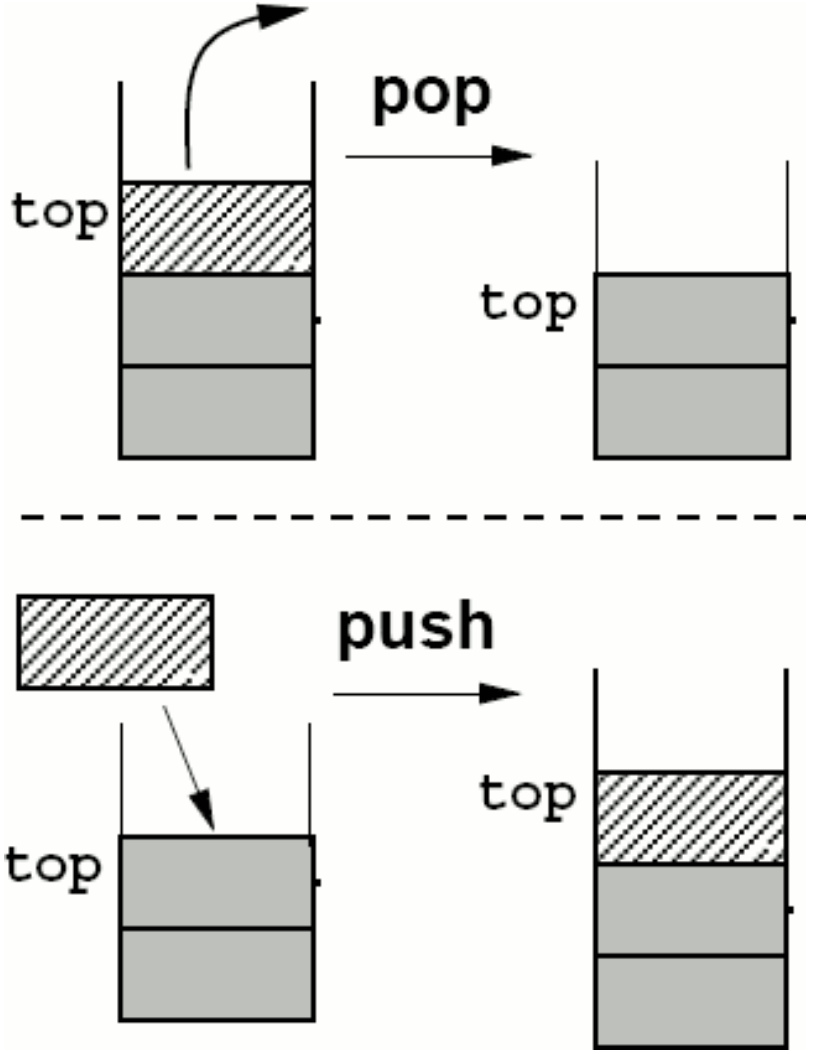
7.1 Stack Properties

- Surfing the Web on a browser:
  - The sequence of back clicks loads the browser with Web pages in reverse order of visit.
  - The last visited page is the first loaded when going back.
- A stack is a collection of entries that demonstrates exactly this last in, first out behavior, called LIFO in short.

7.1 Stack Properties

A stack is a linear collection of entries in which, for every entry y that enters the stack after another entry x, y leaves the stack before x.

7.1 Stack Properties



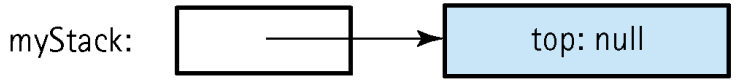
- An entry added or pushes on to the top of a stack.
- An entry is removed, or popped from the top of stack.

7.1 Stack Properties

Operation	Description
Push	Add an entry to the top of stack
Pop	Delete the entry at the top of stack
Peek	Look at the entry at the top of stack

Results of Stack Operations Using `StackNode` *(Cont'd)*

myStack=new LinkedStack();

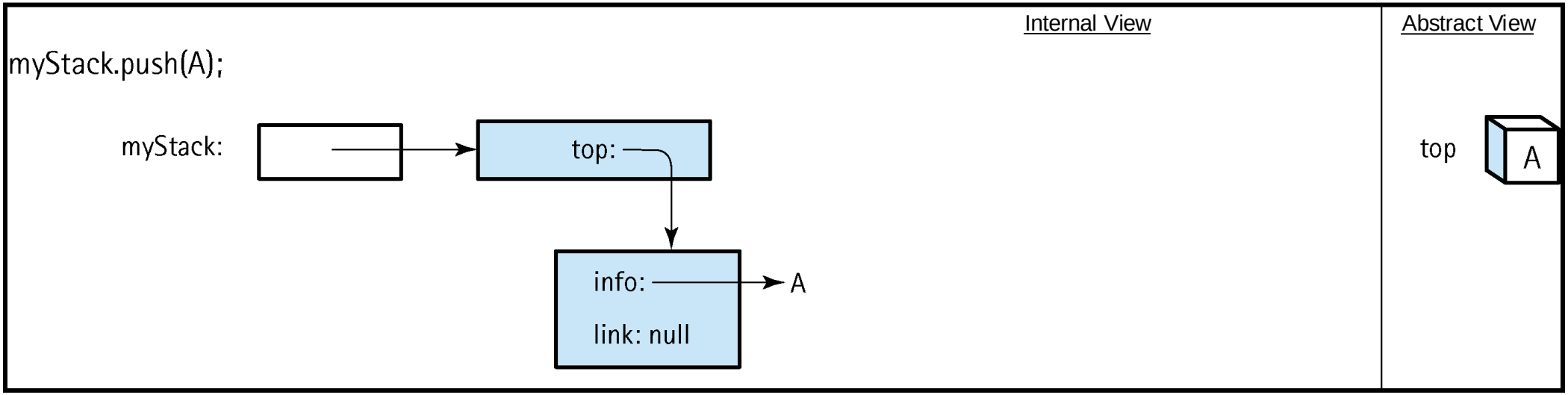


Internal View

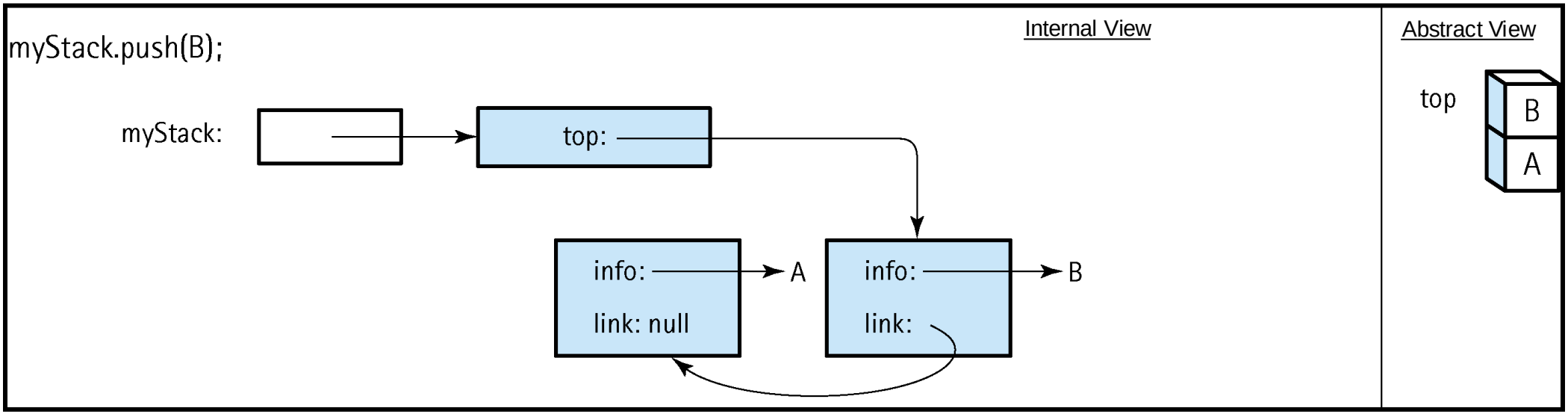
Abstract View

(Empty)

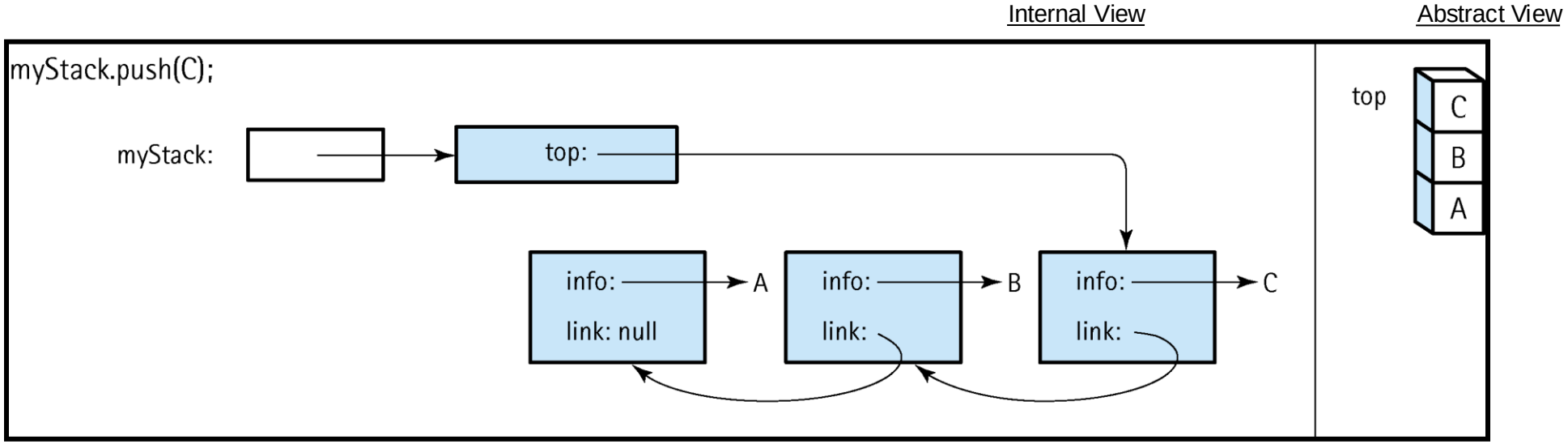
Results of Stack Operations Using `StackNode` *(Cont'd)*



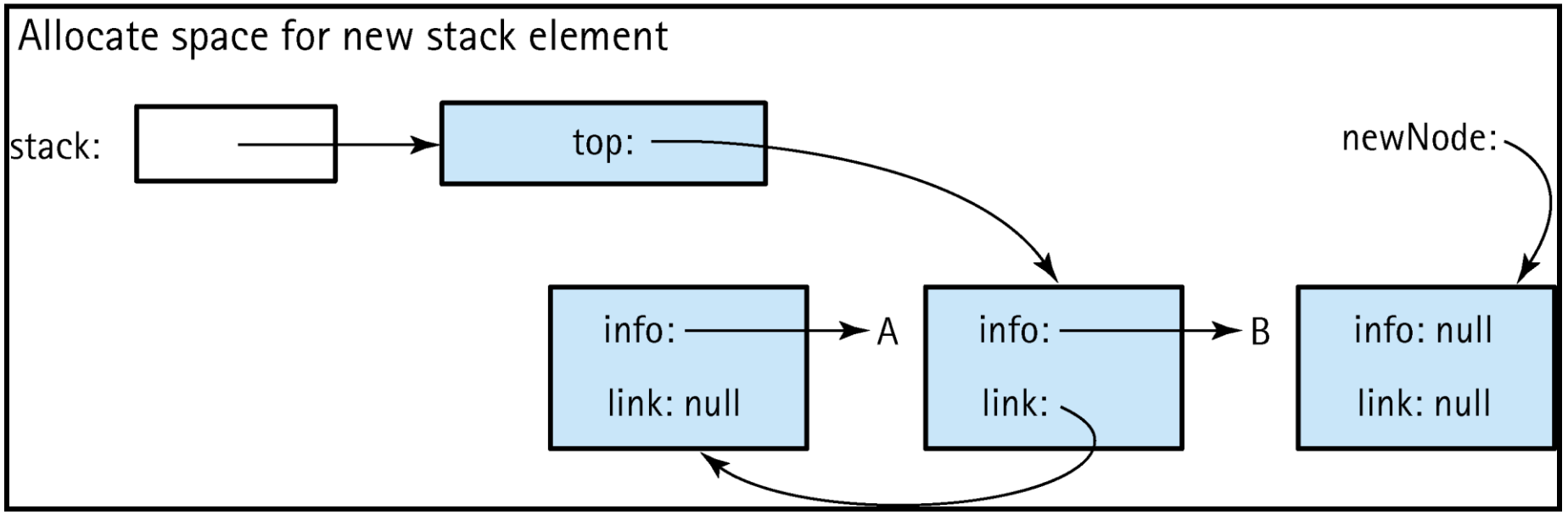
Results of Stack Operations Using `StackNode` *(Cont'd)*



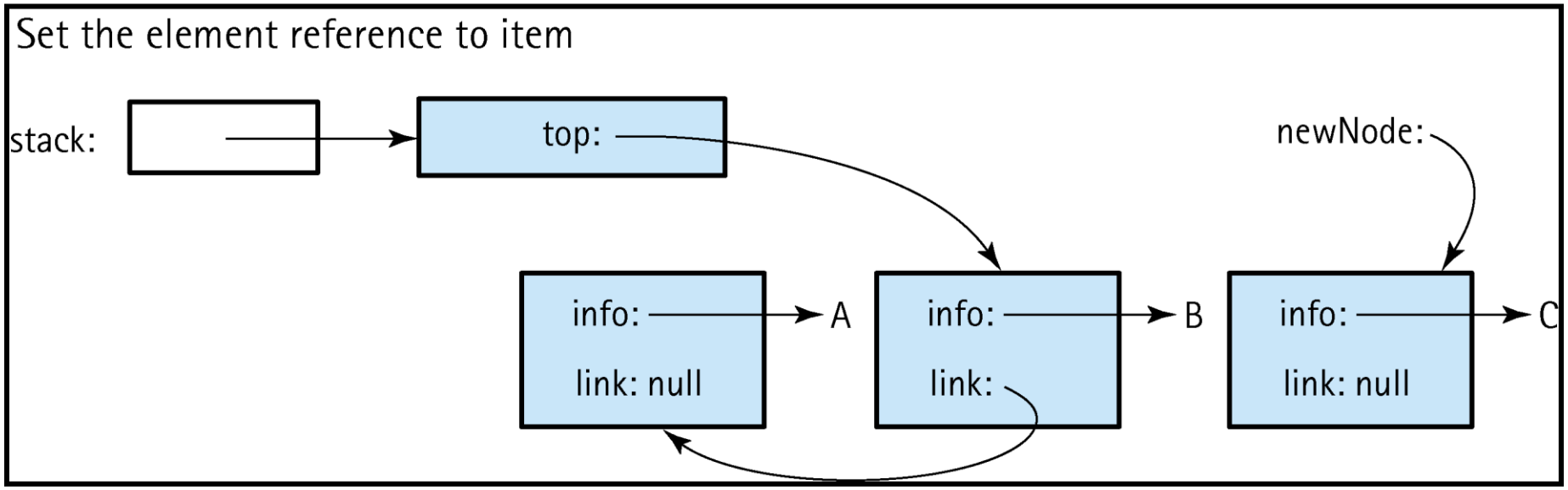
Results of Stack Operations Using `StackNode` *(Cont'd)*



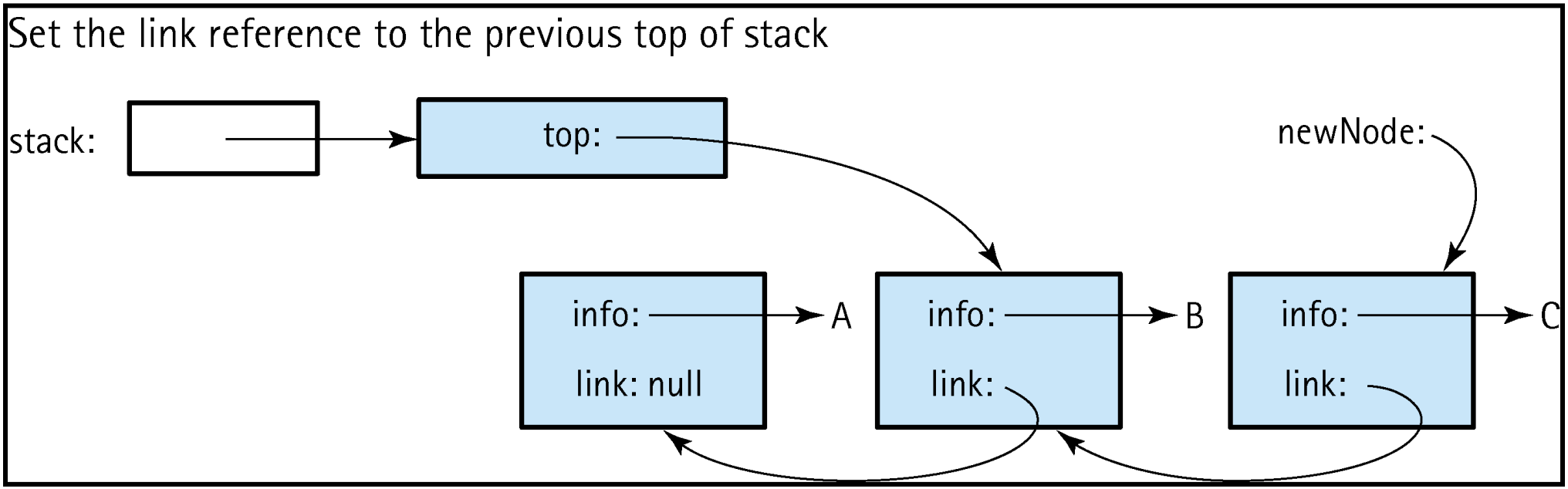
Results of `push` Operation



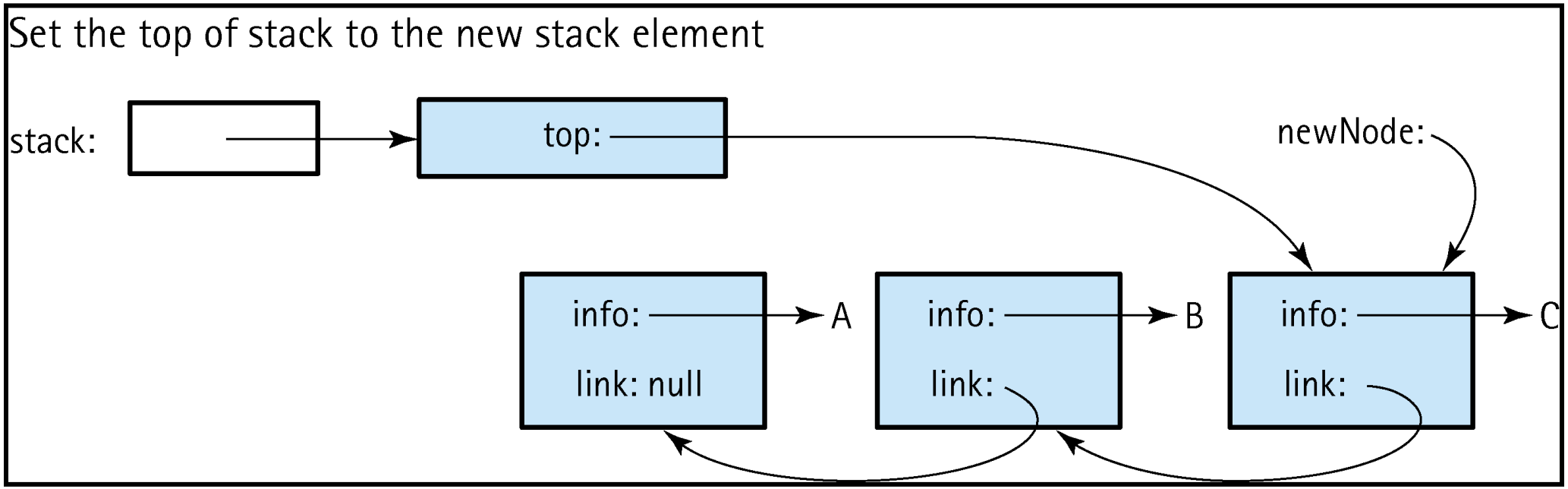
Results of `push` Operation *(Cont'd)*



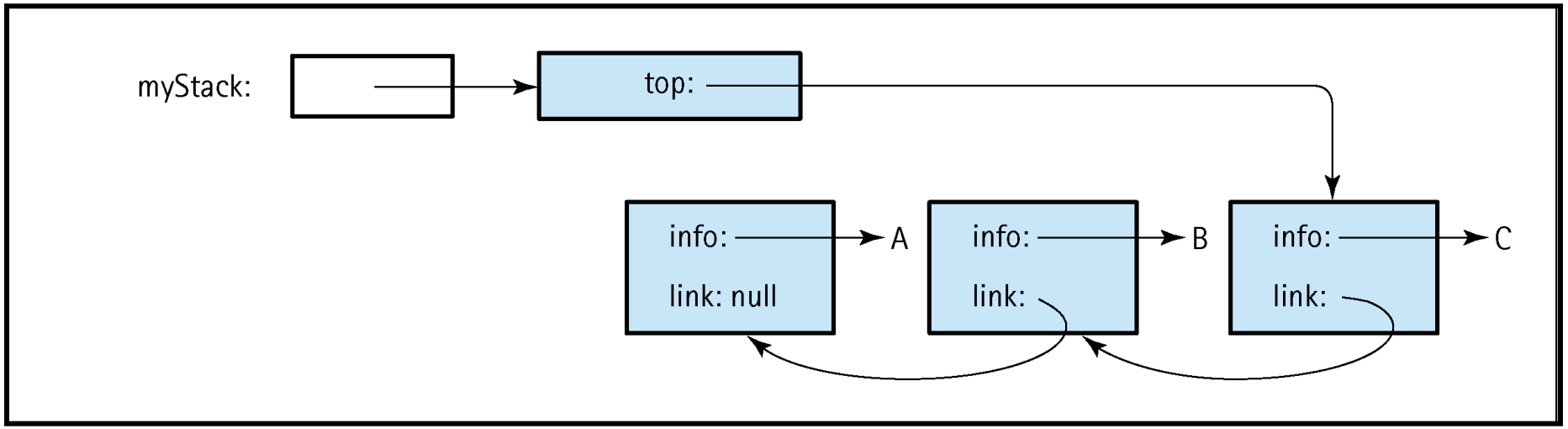
Results of `push` Operation *(Cont'd)*



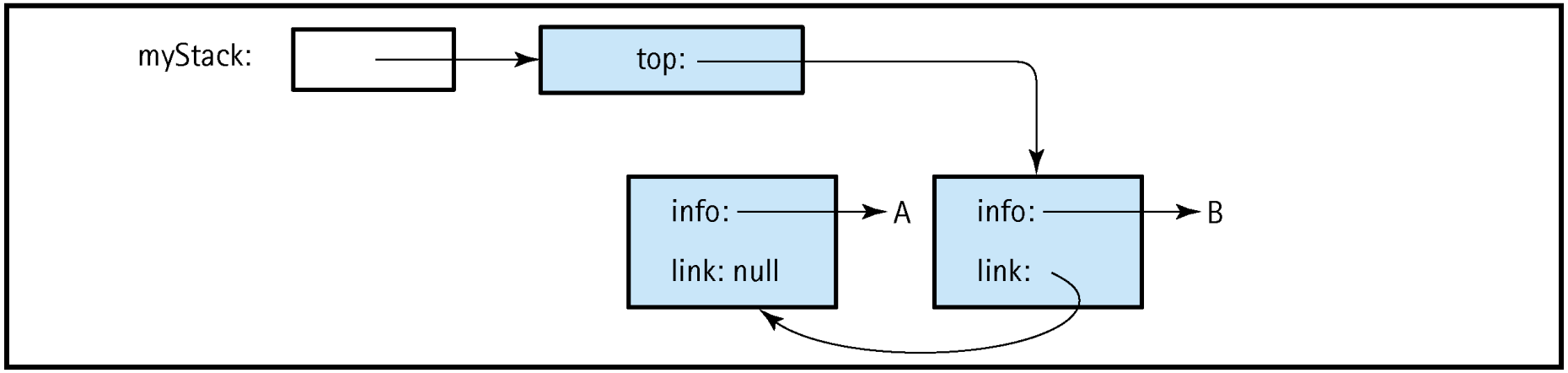
Results of `push` Operation *(Cont'd)*



Results of `pop` Operation



top = top.next



7.2.2 Postfix Expression Evaluation

- We write arithmetic expressions like so:
- Consider the expression:

```
-a / (b + c) * d - e
```

```
2 - 3 + 5 * 2 * 3 + 4
```

- It cannot simply be scan left to right.

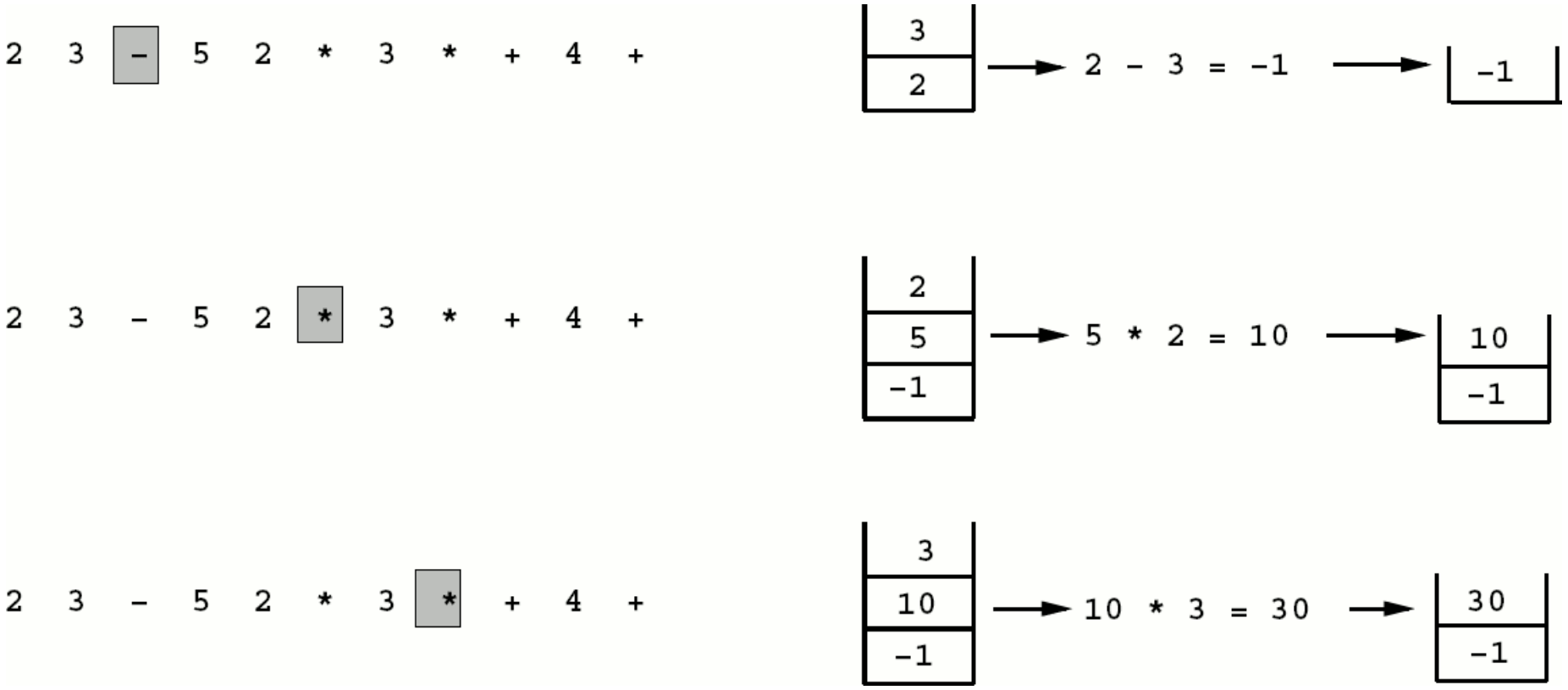
```
(((2 - 3) + ((5 * 2) * 3)) + 4)
```

7.2.2 Postfix Expression Evaluation

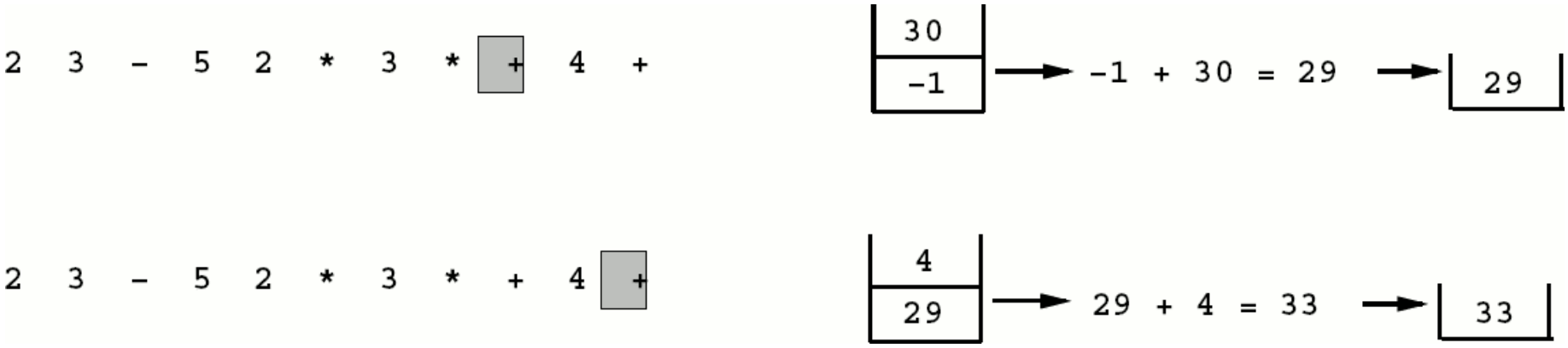
Infix	Postfix
a + b * c – d	a b c * + d –
(a + b) * c – d	a b + c * d –
a + b * c – f / d	a b c * + f d / –
a + b * (c – f)/d	a b c f – * d /+
2 – 3 + 5 * 2 * 3 + 4	2 3 – 5 2 * 4 * + 4 +

- Postfix, does away with the need for parentheses.
- An operator always follows the operands or sub-expressions on which it operates.

7.2.2 Postfix Expression Evaluation



7.2.2 Postfix Expression Evaluation



7.2.2 Postfix Expression Evaluation

- Two conditions that must be met for the evaluation process to terminate successfully:
  - When an operator is encountered, there must exist a most recent pair of operands or temporary results for application.
  - When the scanning of the expression is complete, there must be exactly one value on the stack.

7.2.2 Postfix Expression Evaluation

algorithm postfixEvaluation

```
expr ← input postfix expression string
tok ← first token of expr
while (tok is not empty) do
    if (tok is an operator) then
        topval ← pop stack
        nextval ← pop stack
        result ← apply tok on
                    topval and nextval
        push result on stack
    else
        value ← tok string converted to
                    integer
        push value on stack
    endif
    tok ← next token of expr
endwhile

    result ← pop stack
```

7.2.2 Postfix Expression Evaluation

- Two possible errors that may be encountered by the algorithm:
  - One is that of insufficient operands.
  - The other is that of too many operands.
    - Insufficient operands case is detected when the token is an operator, but the stack has less than the two operands on which the operator must be applied.
    - Too many operands case is detected after the while loop, when the stack has more than one entry in it.

7.3 A Stack Class

Class Stack<E>

`public void` **`clear`**`()`
Removes all of the elements

`Public boolean` **`empty`**`()`
Tests if this stack is empty.

`public E` **`peek`**`()`
Looks at the object at the top of this stack without removing it from the stack.
Throws: `EmptyStackException` - if this stack is empty.

`public E` **`pop`**`()`
Removes the object at the top of this stack and returns that object as the value of this function. **Throws:** `EmptyStackException` - if this stack is empty.

`public E` **`push`**`(E item)`
Pushes an item onto the top of this stack.

`public int` **`size`**`()`
Returns the number of components

`Public boolean` **`contains`**`(Object o)`
Returns `true` if this vector contains the specified element.

7.4 A Postfix Expression Evaluation Package

```
Enter the postfix expression below
   ==> 12.5 15.6 10.8 + -
```

- Every step of the evaluation processes one token of the expression.

7.4.1 Class PostfixEvaluator

```
package apps.linear.postfix;
import java.util.StringTokenizer;
import java.util.NoSuchElementException;
import java.io.PrintWriter;

public class PostfixEvaluator {
    StringTokenizer exprTok;

    StackKeeper postStack;

    public PostfixEvaluator() {
        postStack = new StackKeeper();

    }

   public void init(String expr) { ··· }
   public float runAll(){ ··· }

}
```

7.4.1 Class *PostfixEvaluator*

- *java.util.StringTokenizer* parses the postfix expression into tokens and deals them out one at a time.

- *StackKeeper* maintains the evaluation stack.

7.4.1 Class *PostfixEvaluator*

Message	Action
init	Initialize all objects to evaluate a specified postfix expression
runAll	Run the evaluator to completion and return the result

7.4.1 Class *PostfixEvaluator*

- *RunAll* message evaluates and produces the results in one shot.
- Restart the evaluator by the *init* message.

7.4.3 Class *StackKeeper*

Class File 7.2 *StackKeeper.java*

```
package apps.linear.postfix;
 import java.util.*;
import java.util.NoSuchElementException;
import java.io.PrintWriter;
import structures.linear.Stack;

class StackKeeper {
    static final char[] operators = {'+', '-', '*', '/'};
    Stack<Float> evalStack;
```

7.4.3 Class *StackKeeper*

```
    StackKeeper() { evalStack = new Stack<Float>(); }
    void init() { evalStack.clear(); }
    int size() {
        return evalStack.size();
    }
    void update(String token) {
        if (isOperator(token)) {
            evaluate(token.charAt(0));
        } else {
            evalStack.push(Float.valueOf(token));
        }
    }
    float getTop() {
        Float top = evalStack. peek( );
        if (top == null) {
            throw new NoSuchElementException();
        }
        return top;
    }
    boolean isOperator(String instr) { ... }
    void evaluate(char op) { ... }
}
```

7.4.3 Class *StackKeeper*

```
boolean isOperator(String instr) {
    if (instr.length() > 1) {
        return false;
    }
    char c = instr.charAt(0);
    for (int i=0;
         i < operators.length;
         i++) {
        if (c == operators[i]) {
            return true;
        }
    }
    return false;
}
```

7.4.3 Class *StackKeeper*

```
void evaluate(char op) {
    Float topval = evalStack.pop();
    Float nextval = evalStack.pop();
    float tempval=0;
    switch (op) {
        case '+':  tempval = nextval + topval; break;
        case '-':  tempval = nextval - topval; break;
        case '*':  tempval = nextval * topval; break;
        case '/':  tempval = nextval / topval; break;
    }
    evalStack.push(tempval);
}
```

7.4.4 Class *PostfixEvaluator* Implementation

```
public void init(String expr) {
    postStack.init();
    exprTok = new StringTokenizer(expr);
}
public float runAll(){
    while (exprTok.hasMoreTokens()) {
    String nextTok = exprTok.nextToken();

        postStack.update(nextTok);
    }
   return postStack.getTop(); // result
}
```

7.4.4 Class *PostfixEvaluator* Implementation

- The S*tringTokenizer* method count *Tokens* returns the number of tokens that remain to be enumerated.
  - At the end of the run, the stack must contain exactly one element.

7.4.4 Class *PostfixEvaluator* Implementation

- The *NoSuchElementException* does two things:
  - Prints the current evaluation status so the calling application gets as much information as possible about the source of the exception.
  - Throws an *IllegalExpressionException*, in order to deliver the most precise and complete information about the cause of the exception.
    - This is much better than just passing through the *NoSuchElementException*, which, in this context, is not informative enough.