Stacks and Queues

Stacks

Ref: "Data Structures and Algorithm Analysis", C. Shaffer, pp. 125–133.

- A stack is one of the most commonly used data structures in computer science
- A stack can be compared to a Pez dispenser
  - Only the top item can be accessed
  - You can extract only one item at a time
- The top element in the stack is the one added to the stack most recently
- The stack's storage policy is *Last-In, First-Out*, or *LIFO*

Specification of the Stack Abstract Data Type

- Only the top element of a stack is visible; therefore the number of operations performed by a stack are few
- We need the ability to
  - test for an empty stack (empty)
  - inspect the top element (peek)
  - retrieve the top element (pop)
  - put a new element on the stack (push)

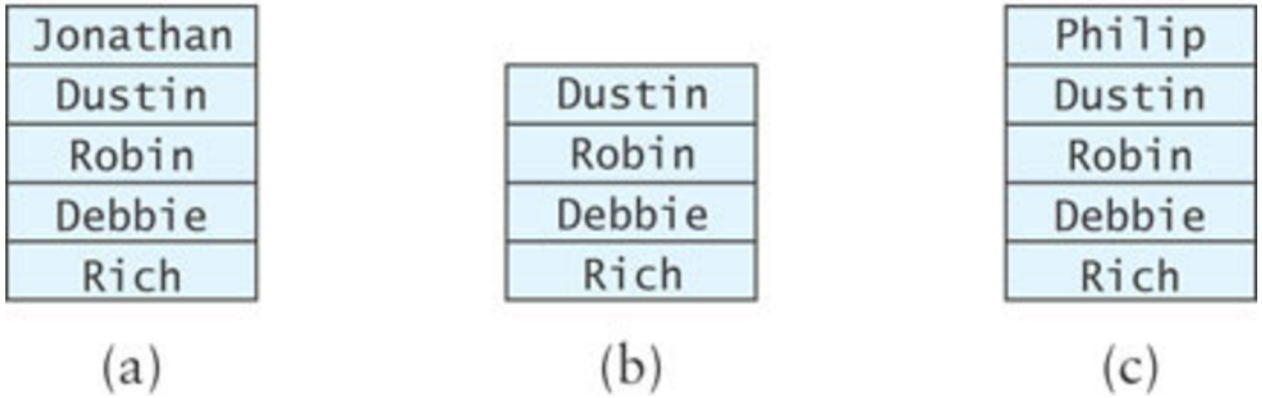
Methods	Behavior
`boolean empty()`	Returns **`true`** if the stack is empty; otherwise, returns **`false`**.
`E peek()`	Returns the object at the top of the stack without removing it.
`E pop()`	Returns the object at the top of the stack and removes it.
`E push(E obj)`	Pushes an item onto the top of the stack and returns the item pushed.

Stack exceptions

Exceptions

- In the Stack ADT, operations pop and top cannot be performed if the stack is empty
- Attempting the execution of pop or top on an empty stack throws an EmptyStackException

A Stack of Strings



- "Rich" is the oldest element on the stack and "Jonathan" is the youngest (Figure a)
- `String last = names.peek();` stores a reference to "Jonathan" in `last`
- `String temp = names.pop();` removes "Jonathan" and stores a reference to it in `temp` (Figure b)
- `names.push("Philip");` pushes "Philip" onto the stack (Figure c)

Source : Dr.Tsekourakis + text book

Stack implementations

- Stack should be an ADT interface with methods push(), pop(), top() etc.
- There should be concrete implementations like ArrayStack, LinkedListStack etc
- But in Java, Stack<E> is implemented as Vector<E> (which is like an ArrayList)
- All operations can be done in O(1) time

Source : Dr. Tsekourakis + text book

Stack applications

- Function calls require a stack where a stack frame is pushed when a function call is made and popped when function returns from the call. Stack frame keeps track of parameters and results and return addresses.
- Many recursive problems require a stack

(a) Reverse a string (push each character onto stack and they will be popped in reverse order)

(b) Check if a parenthesized expression has balanced parentheses

e.g. (() ((()()))) – balanced

()((()()) – not balanced

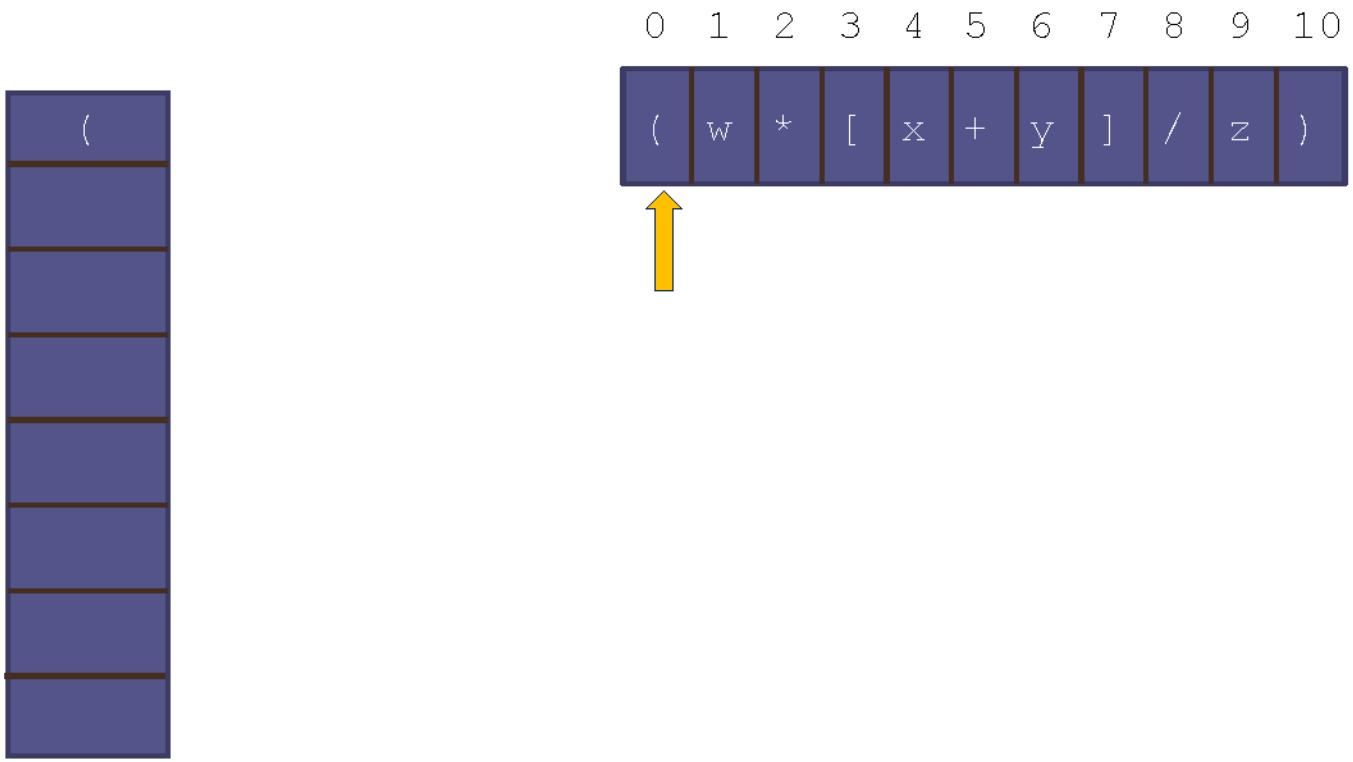
- Undo sequences in an editor
- Browser history of visited web pages

Algorithm for method `isBalanced`

1. Create an empty stack of characters.
2. Assume that the expression is balanced (`balanced` is `true`).
3. Set index to 0.
4. `while` `balanced` is `true` and index < the expression's length
5. Get the next character in the data string.
6. `if` the next character is an opening parenthesis
7. Push it onto the stack.
8. `else if` the next character is a closing parenthesis
9. Pop the top of the stack.
10. `if` stack was empty or its top does not match the closing parenthesis
11. Set balanced to `false`.
12. Increment index.
13. Return `true` if `balanced` is `true` and the stack is empty.

Balanced Parentheses (cont.)

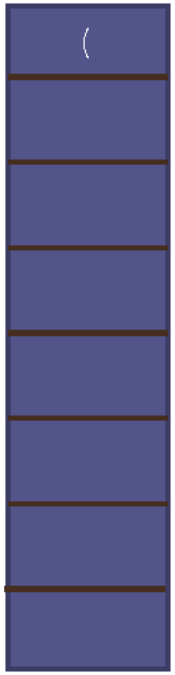
Expression: `(w * [x + y] / z)`

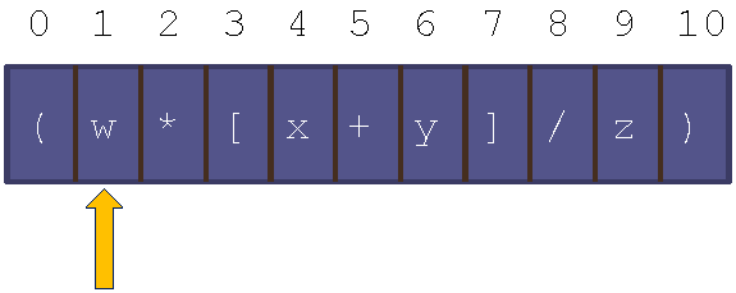


Source : Dr. Tsekourakis + text book

Balanced Parentheses (cont.)

Expression: `(w * [x + y] / z)`



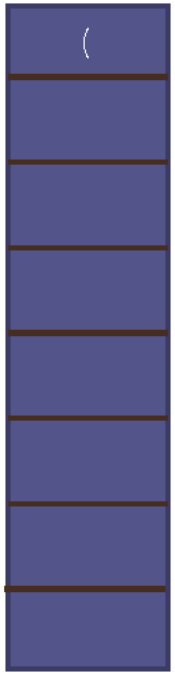


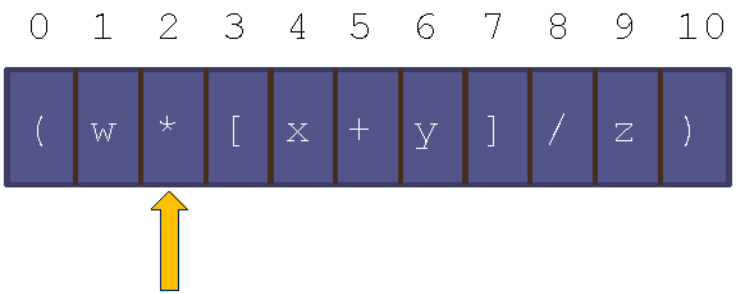

```
balanced : true
index    : 1
```

Source : Dr. Tsekourakis + text book

Balanced Parentheses (cont.)

Expression: `(w * [x + y] / z)`



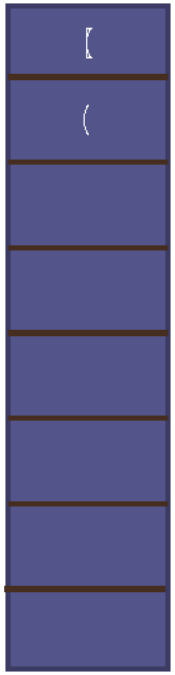


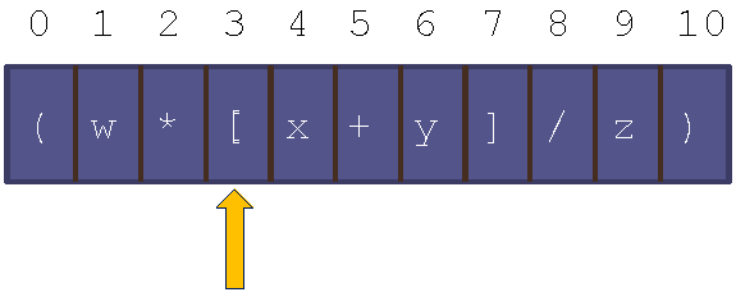

```
balanced : true
index    : 2
```

Source : Dr. Tsekourakis + text book

Balanced Parentheses (cont.)

Expression: `(w * [x + y] / z)`



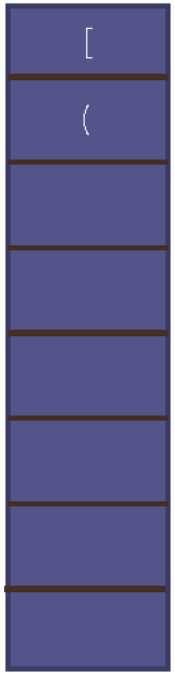


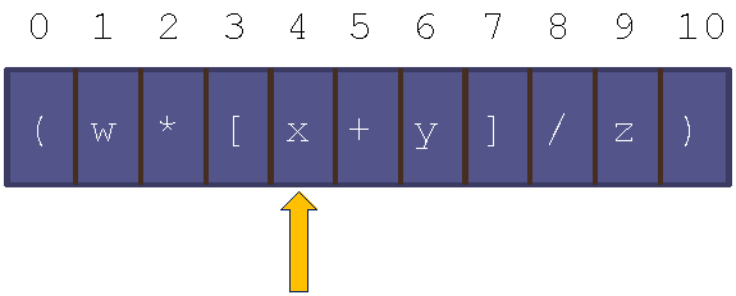

```
balanced : true
index    : 3
```

Source : Dr. Tsekourakis + text book

Balanced Parentheses (cont.)

Expression: `(w * [x + y] / z)`



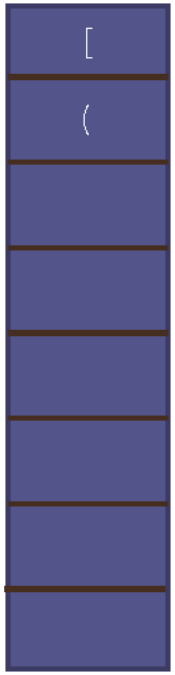


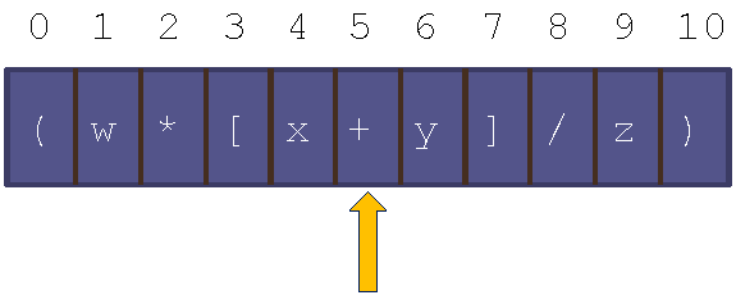

```
balanced : true
index    : 4
```

Source : Dr. Tsekourakis + text book

Balanced Parentheses (cont.)

Expression: `(w * [x + y] / z)`



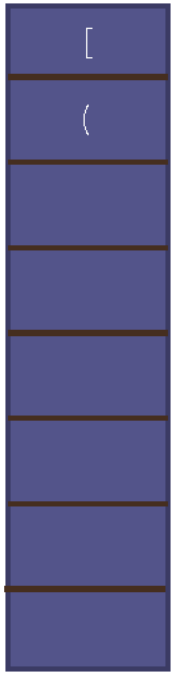


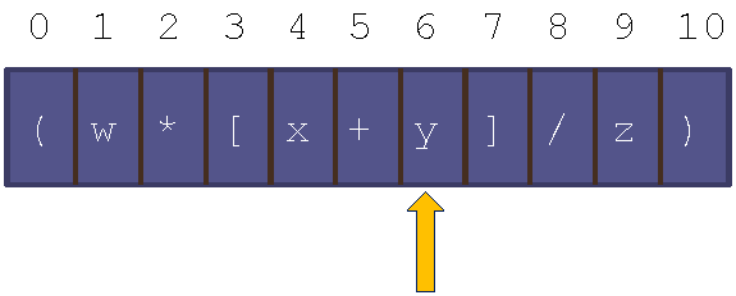

```
balanced : true
index    : 5
```

Source : Dr. Tsekourakis + text book

Balanced Parentheses (cont.)

Expression: `(w * [x + y] / z)`



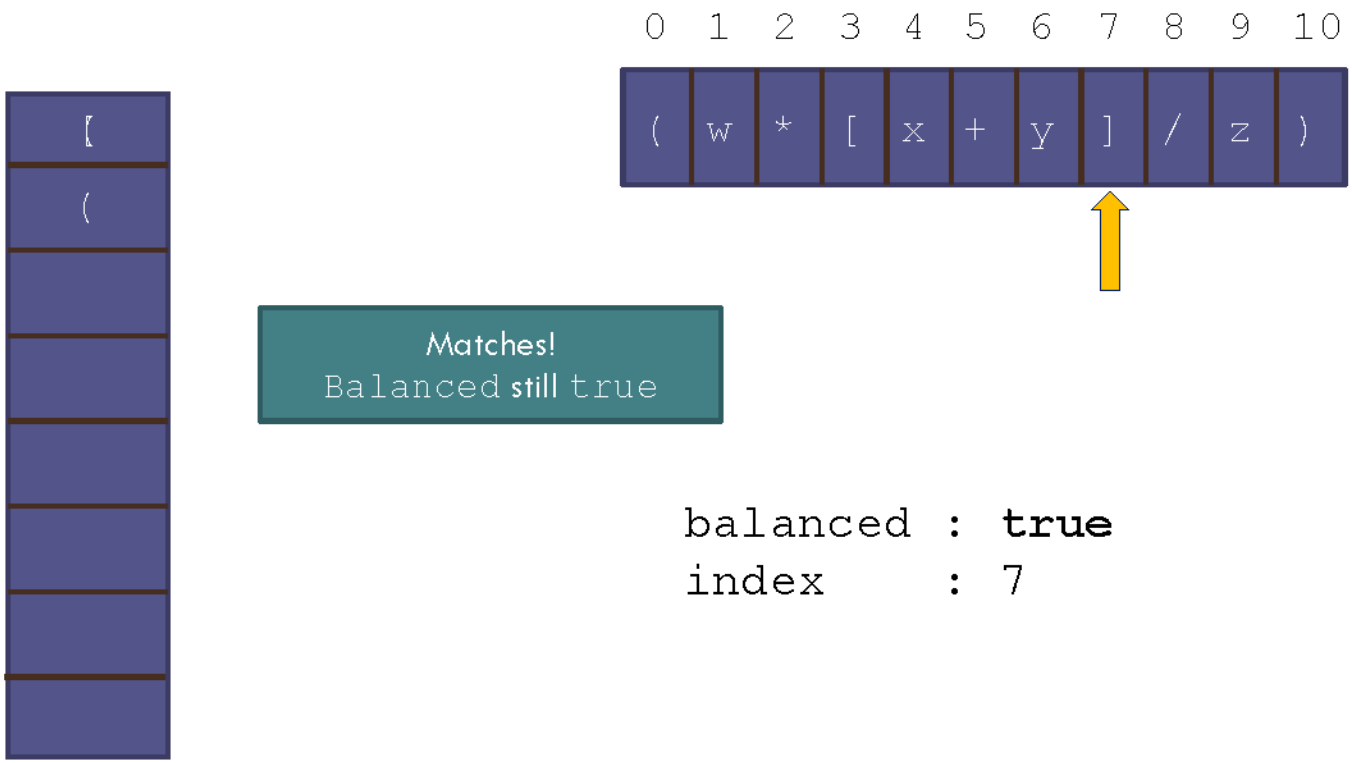



```
balanced : true
index    : 6
```

Source : Dr. Tsekourakis + text book

Balanced Parentheses (cont.)

Expression: `(w * [x + y] / z)`

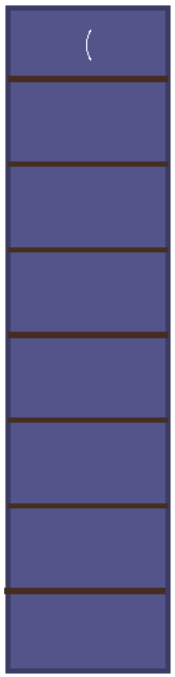


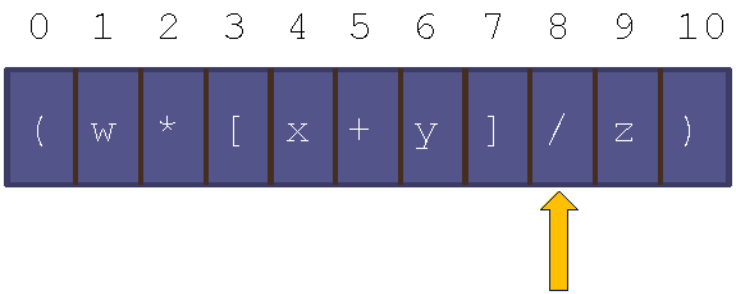

```
balanced : true
index    : 7
```

Source : Dr. Tsekourakis + text book

Balanced Parentheses (cont.)

Expression: `(w * [x + y] / z)`



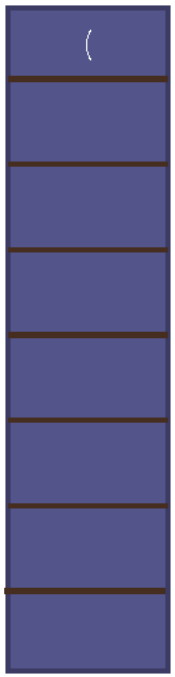


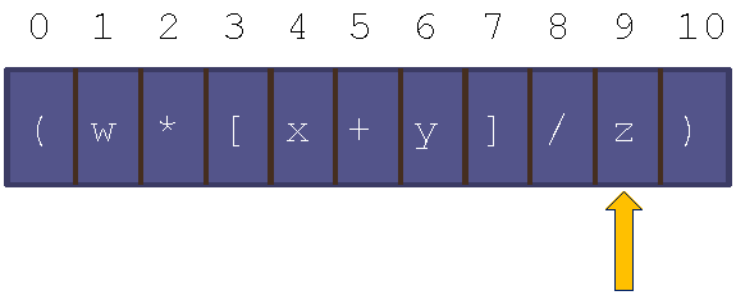

```
balanced : true
index    : 8
```

Source : Dr. Tsekourakis + text book

Balanced Parentheses (cont.)

Expression: `(w * [x + y] / z)`



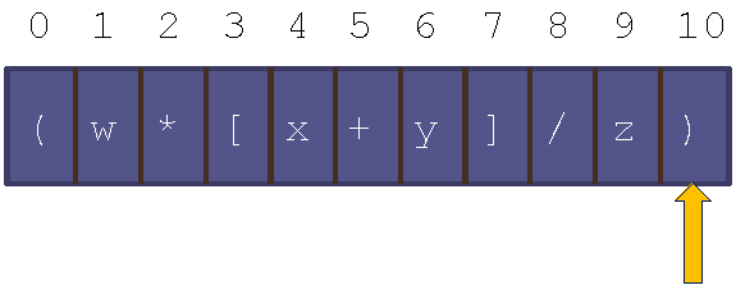


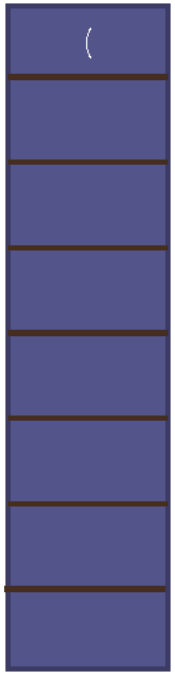

```
balanced : true
index    : 9
```

Source : Dr.Tsekourakis + text book

Balanced Parentheses (cont.)

Expression: `(w * [x + y] / z)`





Matches!
Balanced still true

```
balanced : true
index    : 10
```

Source : Dr. Tsekourakis + text book

Additional Stack Applications

- Postfix and infix notation
  - Expressions normally are written in infix form, but
  - it easier to evaluate an expression in postfix form since there is no need to group sub-expressions in parentheses or worry about operator precedence

Postfix Expression	Infix Expression	Value
4 7 *	4 * 7	28
4 7 2 + *	4 * (7 + 2)	36
4 7 * 20 -	(4 * 7) - 20	8
3 4 7 * 2 / +	3 + ((4 * 7) / 2)	17

Source : Dr.Tsekourakis + text book

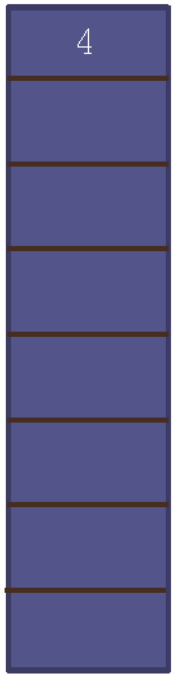
Evaluating Postfix Expressions

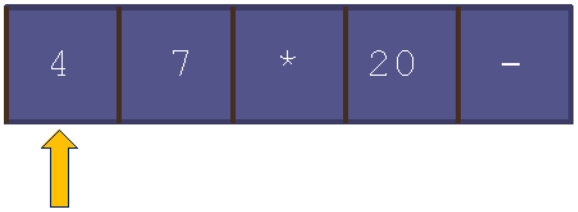
- Write a class that evaluates a postfix expression
- Use the space character as a delimiter between tokens

Data Field	Attribute
`Stack<Integer> operandStack`	The stack of operands (`Integer` objects).
Method	**Behavior**
`public int eval(String expression)`	Returns the value of `expression`.
`private int evalOp(char op)`	Pops two operands and applies operator `op` to its operands, returning the result.
`private boolean isOperator(char ch)`	Returns **`true`** if `ch` is an operator symbol.

Source : Dr.Tsekourakis + text book

Evaluating Postfix Expressions (cont.)

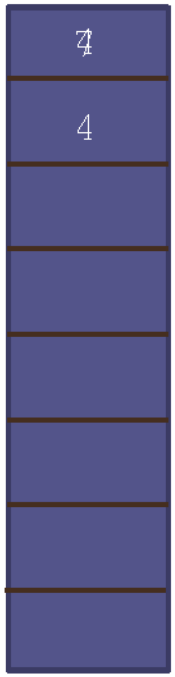


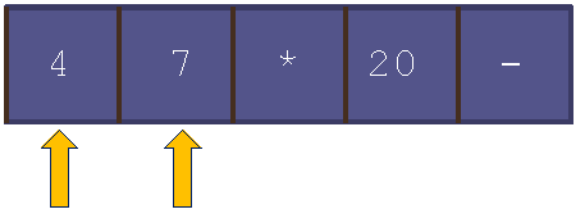



```
1. create an empty stack of integers
2. while there are more tokens
3.    get the next token
4.    if the first character of the token is a digit
5.       push the character on the stack
6.    else if the token is an operator
7.       pop the right operand off the stack
8.       pop the left operand off the stack
9.       evaluate the operation
10.      push the result onto the stack
11. pop the stack and return the result
```

Source : Dr.Tsekourakis + text book

Evaluating Postfix Expressions (cont.)

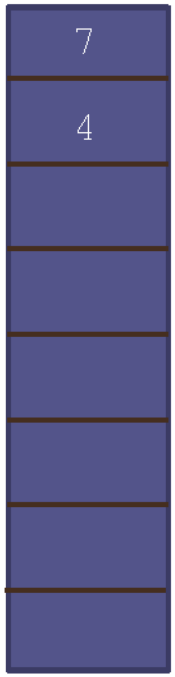




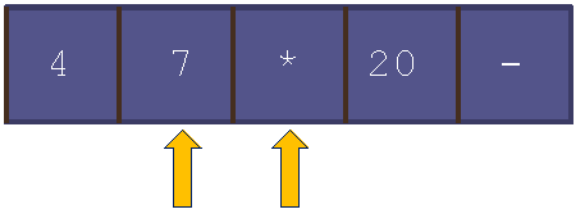

```
1.  create an empty stack of integers
2.  while there are more tokens
3.    get the next token
4.    if the first character of the token is a digit
5.       push the character on the stack
6.    else if the token is an operator
7.       pop the right operand off the stack
8.       pop the left operand off the stack
9.       evaluate the operation
10.      push the result onto the stack
11. pop the stack and return the result
```

Source : Dr.Tsekourakis + text book

Evaluating Postfix Expressions (cont.)



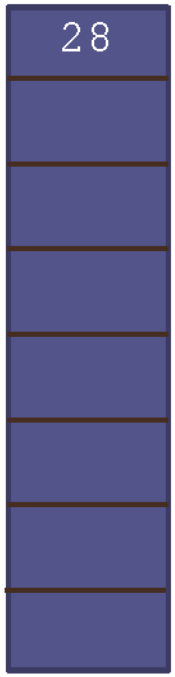
4 * 7



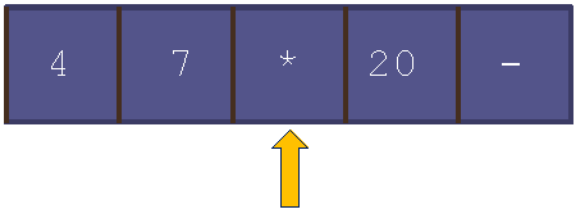

```
1.  create an empty stack of integers
2.  while there are more tokens
3.    get the next token
4.    if the first character of the token is a digit
5.       push the character on the stack
6.    else if the token is an operator
7.       pop the right operand off the stack
8.       pop the left operand off the stack
9.       evaluate the operation
10.      push the result onto the stack
11. pop the stack and return the result
```

Source : Dr.Tsekourakis + text book

Evaluating Postfix Expressions (cont.)



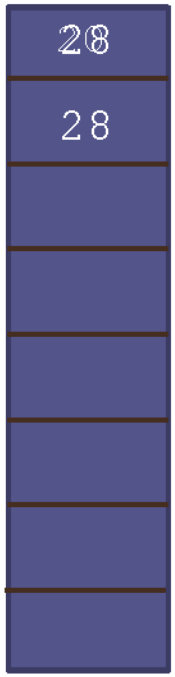
28

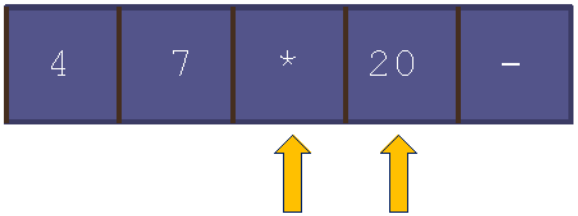



```
1.  create an empty stack of integers
2.  while there are more tokens
3.    get the next token
4.    if the first character of the token is a digit
5.       push the character on the stack
6.    else if the token is an operator
7.       pop the right operand off the stack
8.       pop the left operand off the stack
9.       evaluate the operation
10.      push the result onto the stack
11. pop the stack and return the result
```

Source : Dr.Tsekourakis + text book

Evaluating Postfix Expressions (cont.)

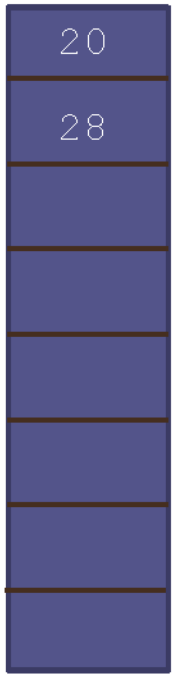




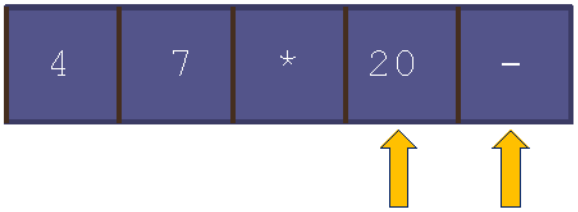

```
1.  create an empty stack of integers
2.  while there are more tokens
3.    get the next token
4.    if the first character of the token is a digit
5.       push the character on the stack
6.    else if the token is an operator
7.       pop the right operand off the stack
8.       pop the left operand off the stack
9.       evaluate the operation
10.      push the result onto the stack
11. pop the stack and return the result
```

Source : Dr.Tsekourakis + text book

Evaluating Postfix Expressions (cont.)



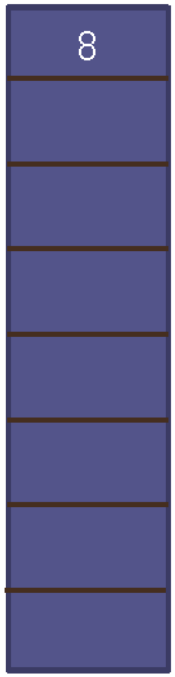
28 - 20



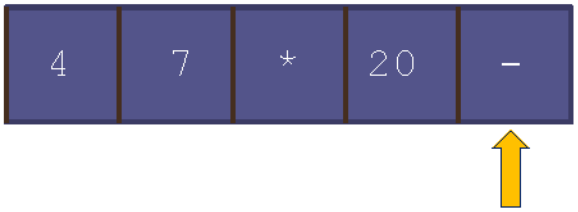

```
1.  create an empty stack of integers
2.  while there are more tokens
3.    get the next token
4.    if the first character of the token is a digit
5.       push the character on the stack
6.    else if the token is an operator
7.       pop the right operand off the stack
8.       pop the left operand off the stack
9.       evaluate the operation
10.      push the result onto the stack
11. pop the stack and return the result
```

Source : Dr.Tsekourakis + text book

Evaluating Postfix Expressions (cont.)



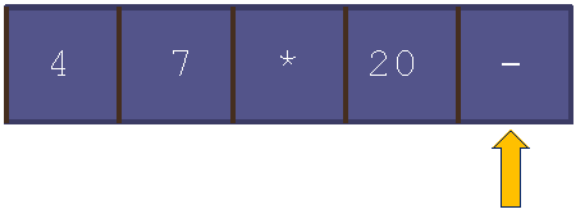
8

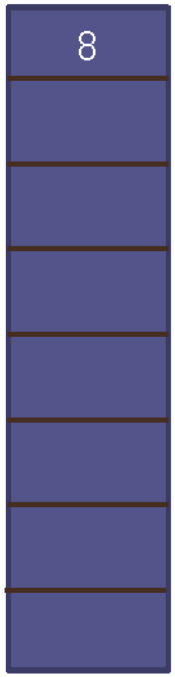



```
1.  create an empty stack of integers
2.  while there are more tokens
3.    get the next token
4.    if the first character of the token is a digit
5.       push the character on the stack
6.    else if the token is an operator
7.       pop the right operand off the stack
8.       pop the left operand off the stack
9.       evaluate the operation
10.      push the result onto the stack
11. pop the stack and return the result
```

Source : Dr.Tsekourakis + text book

Evaluating Postfix Expressions (cont.)






```
1.  create an empty stack of integers
2.  while there are more tokens
3.    get the next token
4.    if the first character of the token is a digit
5.       push the number on the stack
6.    else if the token is an operator
7.       pop the right operand off the stack
8.       pop the left operand off the stack
9.       evaluate the operation
10.      push the result onto the stack
11. pop the stack and return the result
```

Source : Dr.Tsekourakis + text book

Converting from Infix to Postfix (cont.)

- Example: convert

  w – 5.1 / sum * 2

  to its postfix form

  w 5.1 sum / 2 * -

Converting from infix to Postfix

- **Examples :**

(a) 2.3 – 5.1 / 3 ∗ 2 ➔ 2.3 5.1 3 / 2 ∗ –

(b) ((2 + 3)* 5) ** 2 = 2 3 + 5 * 2 **

Converting from Infix to Postfix (cont.)

Next Token	Action	Effect on operatorStack	Effect on postfix
2.3	Append 2.3 to postfix.		2.3
-	The stack is empty Push - onto the stack	-	2.3
5.1	Append 5.1 to postfix	-	2.3 5.1
/	precedence(/) > precedence(-), Push / onto the stack	/ -	2.3 5.1
3	Append 3 to postfix	/ -	2.3 5.1 3
*	precedence(*) equals precedence(/) Pop / off of stack and append to postfix	-	2.3 5.1 3

Source : Dr.Tsekourakis + text book

Converting from Infix to Postfix (cont.)

Next Token	Action	Effect on operatorStack	Effect on postfix
*	precedence(*) > precedence(-), Push * onto the stack	* -	2.3 5.1 3 /
2	Append 2 to postfix	* -	2.3 5.1 3 / 2
End of input	Stack is not empty, Pop * off the stack and append to postfix	-	2.3 5.1 3 / 2 *
End of input	Stack is not empty, Pop - off the stack and append to postfix		2.3 5.1 3 / 2 * -

Source : Dr. Tsekourakis + text book

Converting from Infix to Postfix (cont.)

Algorithm for Method convert

1. Initialize `postfix` to an empty `StringBuilder`.
2. Initialize the operator stack to an empty stack.
3. **`while`** there are more tokens in the infix string
4. Get the next token.
5. **`if`** the next token is an operand
6. Append it to `postfix`.
7. **`else if`** the next token is an operator
8. Call `processOperator` to process the operator.
9. **`else`**
10. Indicate a syntax error.
11. Pop remaining operators off the operator stack and append them to `postfix`.

Source : Dr.Tsekourakis + text book

Converting from Infix to Postfix (cont.)

Algorithm for Method processOperator

```
1.  if the operator stack is empty
2.      Push the current operator onto the stack.
    else
3.      Peek the operator stack and let topOp be the top operator.
4.      if the precedence of the current operator is greater than the
        precedence of topOp
5.          Push the current operator onto the stack.
        else
6.          while the stack is not empty and the precedence of the current
            operator is less than or equal to the precedence of topOp
7.              Pop topOp off the stack and append it to postfix.
8.              if the operator stack is not empty
9.                  Peek the operator stack and let topOp be the top
                    operator.
10.         Push the current operator onto the stack.
```

Source : Dr.Tsekourakis + text book