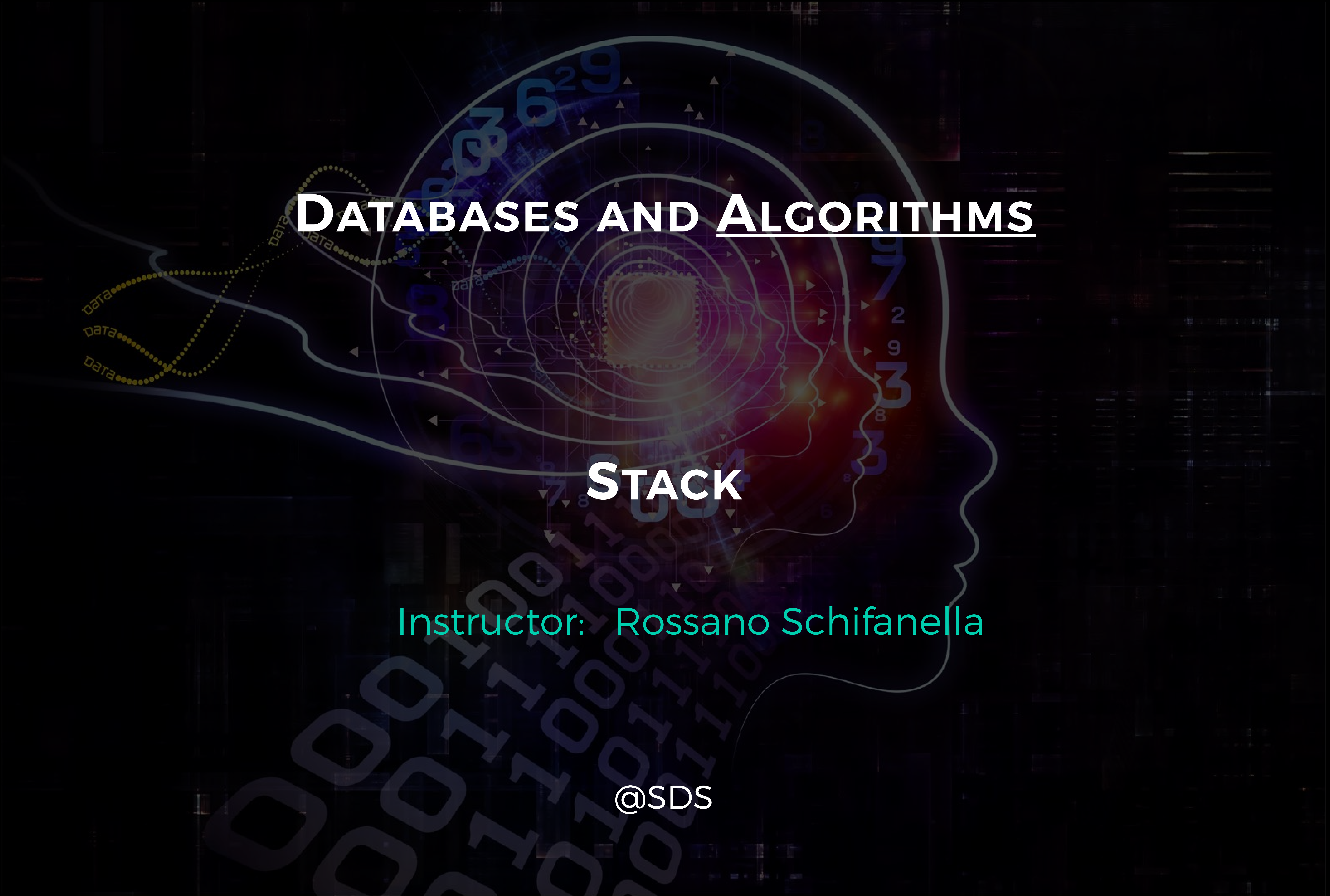


Databases and Algorithms

Stack

Instructor: Rossano Schifanella

@SDS

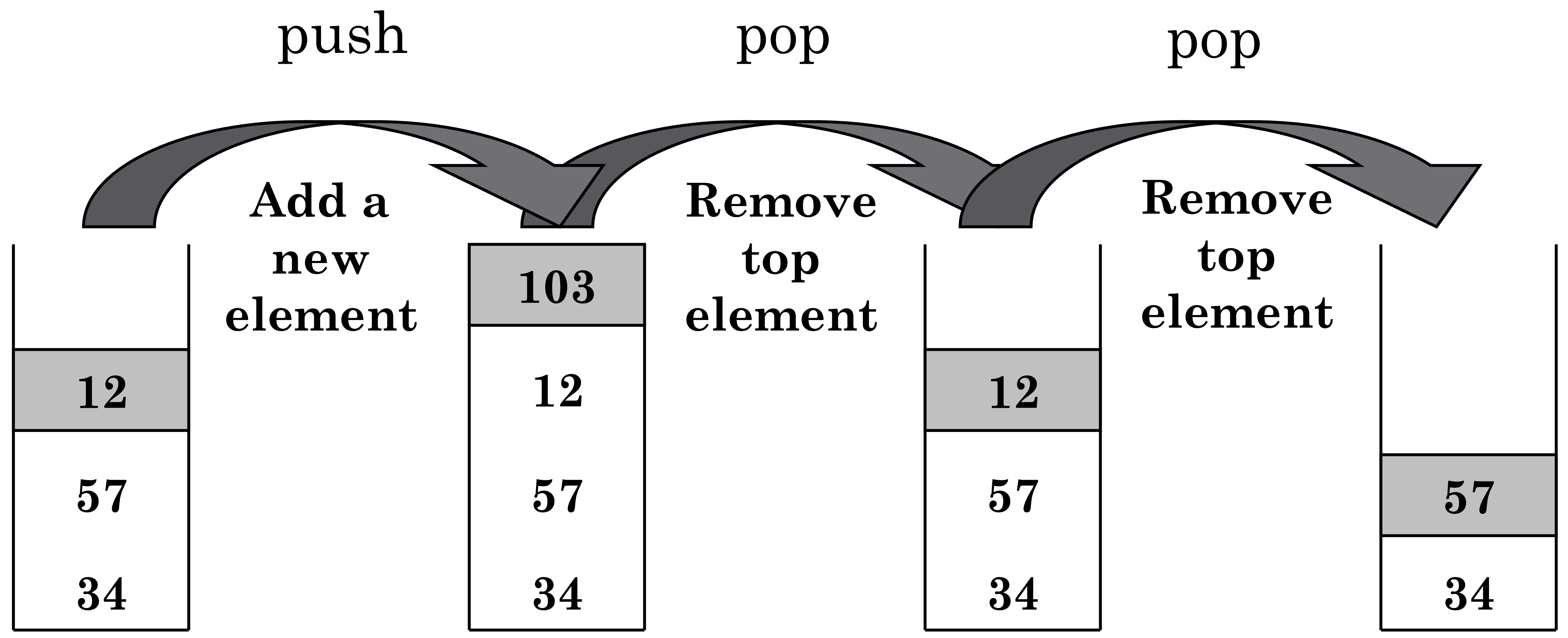
Stack

- A stack is an ordered collection of items where the addition of new items and the removal of existing items always takes place at the same end, referred to as the top of the stack.
  - i.e. add at top, remove from top
- Last-in, first-out (LIFO) property
  - The last item placed on the stack will be the first item removed
- Example:
  - A stack of dishes in a cafeteria

An Example

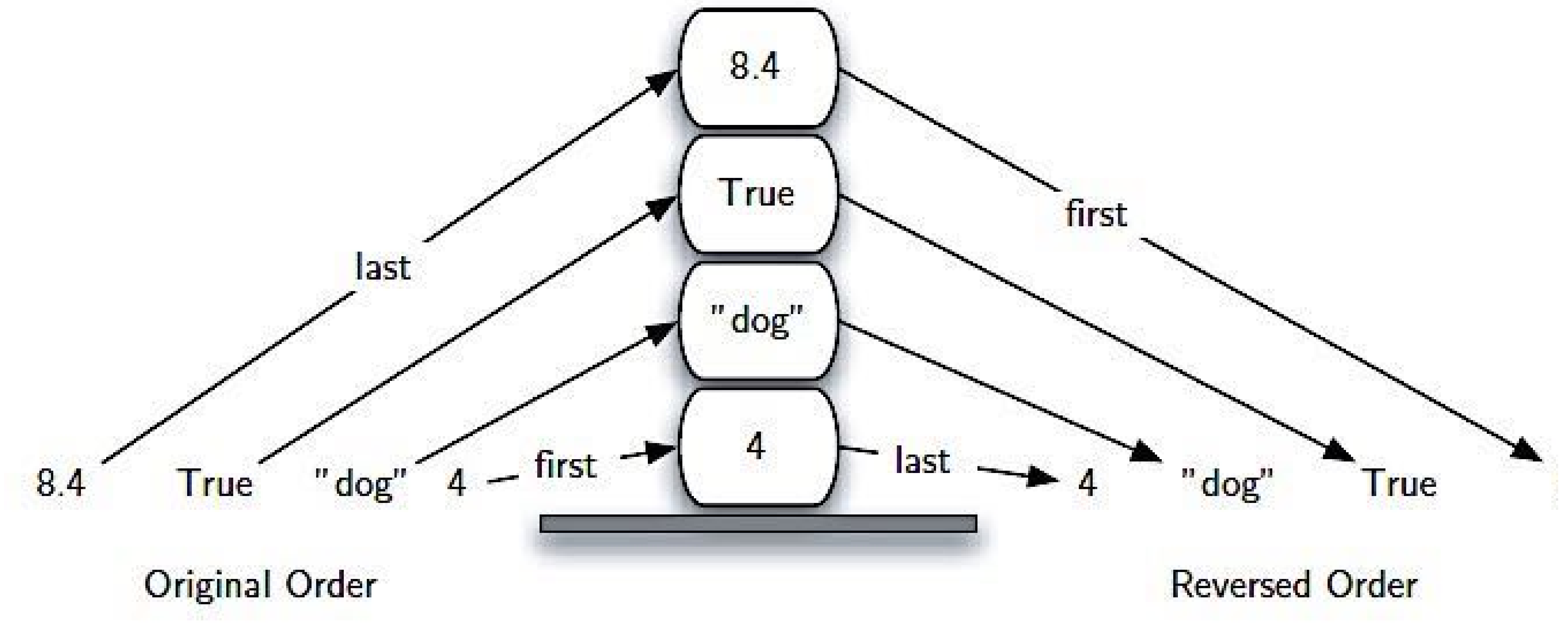
Last In - First Out (LIFO)

- Add only to the top of a Stack
- Remove only from the top of the Stack
  - Note: The last item placed on the stack will be the first item removed



Orders

- The base of the stack contains the oldest item, the one which has been there the longest.
- For a stack the order in which items are removed is exactly the reverse of the order that they were placed.



ADT Stack Operations

- What are the operations which can be used with a Stack Abstract Data?
  - Create an empty stack
  - Determine whether a stack is empty
  - Add a new item to the stack
    - push
  - Remove from the stack the item that was added most recently
    - pop
  - Retrieve from the stack the item that was added most recently
    - peek

The Stack Abstract Data Type

- Stack() creates a new stack that is empty.
  - It needs no parameters and returns an empty stack.
- push(item) adds a new item to the top of the stack.
  - It needs the item and returns nothing.
- pop() removes the top item from the stack.
  - It needs no parameters and returns the item. The stack is modified.
- peek() returns the top item from the stack but does not remove it.
  - It needs no parameters. The stack is not modified.
- is_empty() tests to see whether the stack is empty.
  - It needs no parameters and returns a boolean value.
- size() returns the number of items on the stack.
  - It needs no parameters and returns an integer.

Code Example

Stack operations, state of the stack, values returned

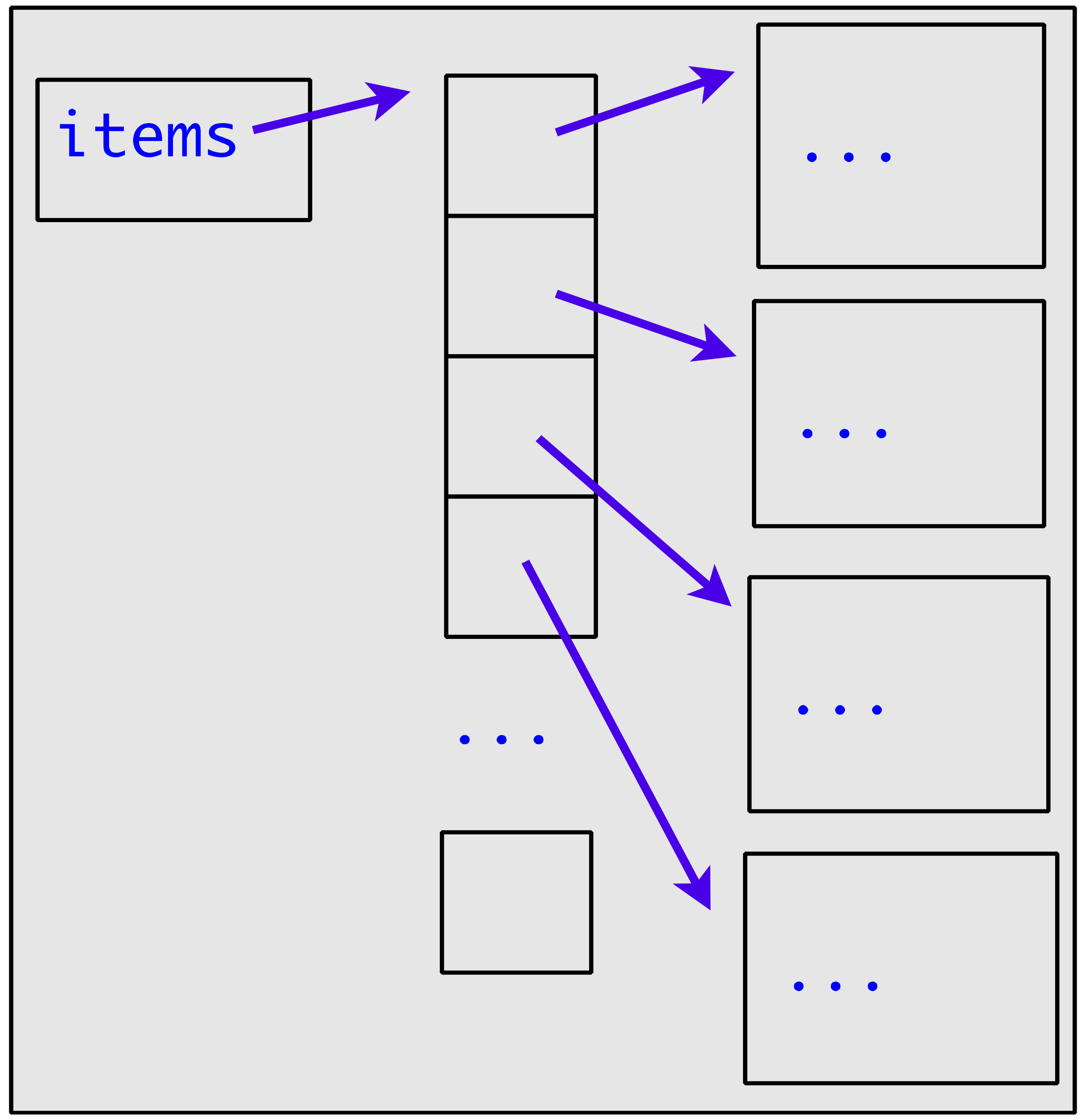
`s.is_empty()`	`[]`	`True`
`s.push(4)`	`[4]`	
`s.push('dog')`	`[4,'dog']`	
`s.peek()`	`[4,'dog']`	`'dog'`
`s.push(True)`	`[4,'dog',True]`	
`s.size()`	`[4,'dog',True]`	`3`
`s.is_empty()`	`[4,'dog',True]`	`False`
`s.push(8.4)`	`[4,'dog',True,8.4]`	
`s.pop()`	`[4,'dog',True]`	`8.4`
`s.pop()`	`[4,'dog']`	`True`
`s.size()`	`[4,'dog']`	`2`

In Python

- We use a python List (array) data structure to implement the stack.
  - Remember: the addition of new items and the removal of existing items always takes place at the same end, referred to as the top of the stack.
    - But which "end" is better?

```
class Stack:
   def __init__(self):
        self.items = []
   def is_empty(self):
        return self.items == []
    def size(self):
        return len(self.items)

    def push(self, item) ...
    def pop(self) ...
    def peek(self) ...
```

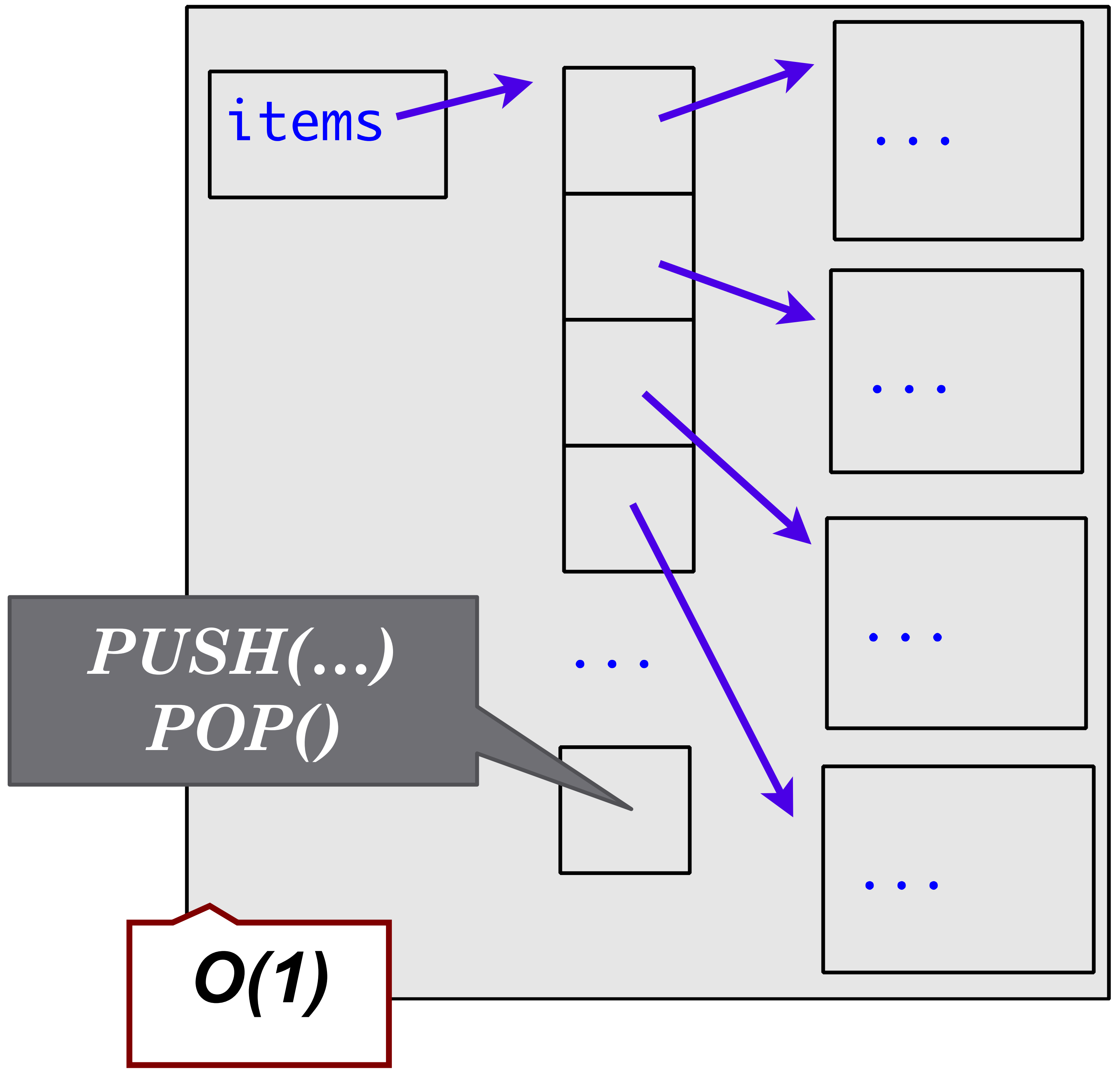


Version 1

- This implementation assumes that the end of the list will hold the top element of the stack.
  - As the stack grows, new items will be added on the END of the list. Pop operations will manipulate the same end

```
class Stack:
  ...
  def push(self, item):
    self.items.append(item)

  def pop(self):
    return self.items.pop()
 ...
```



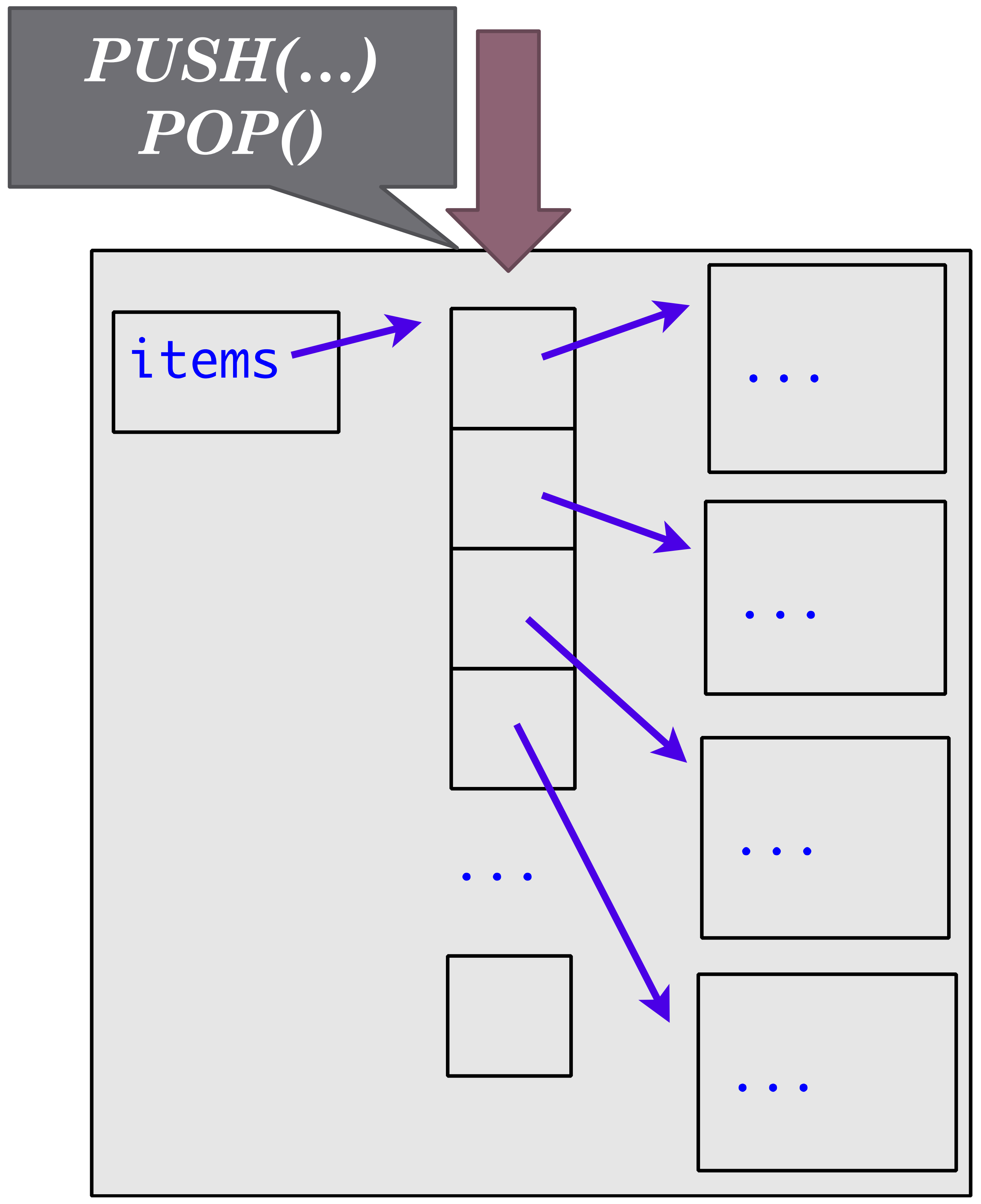
What is the Big-O of the push()/pop()?

Version 2

- This implementation assumes that the beginning of the list will hold the top element of the stack.

```
class Stack:
  ...
  def push(self, item):
    self.items.insert(0,item)

  def pop(self):
    return self.items.pop(0)
 ...
```



O(n)

What is the Big-O of the push()/pop()?

Example Applications

- Checking for Balanced Braces
- Bracket matching
- Postfix calculation
- Conversion from Infix to Postfix

1. Checking for Balanced Braces

- Using a stack to verify whether braces are balanced
  - An example of balanced braces
    - {{}{{}}}
  - An example of unbalanced braces
    - {}}{{}
- Requirements for balanced braces
  - Each time you encounter a “}”, it matches an already encountered “{”
  - When you reach the end of the string, you have matched each “{”

Balanced Braces - Algorithm

- Steps:

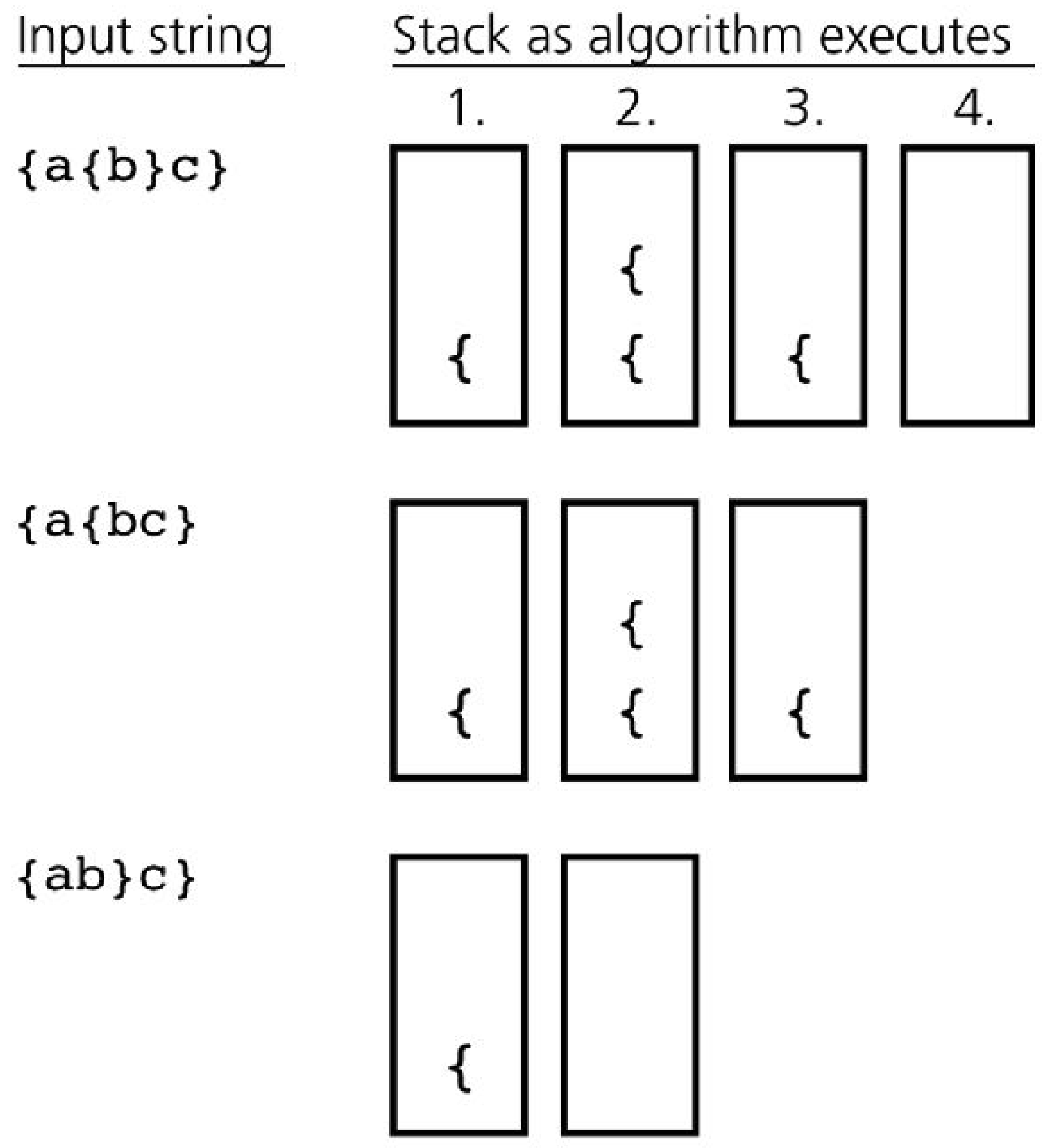
Initialise the stack to empty

For every char read

- *If it is an open bracket then push onto stack*
- *If it is a close bracket,*
  - *If the stack is empty, return ERROR*
  - *Else, pop an element out from the stack*
- *if it is a non-bracket character, skip it*

If the stack is NON-EMPTY, ERROR

Balanced Braces - Examples



1. push "{"
2. push "{"
3. pop
4. pop
Stack empty $\Longrightarrow$ balanced

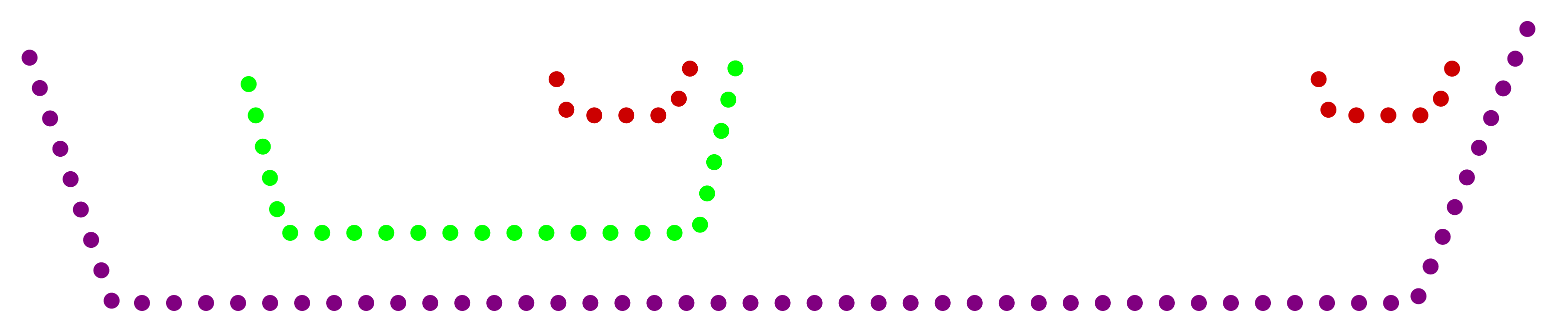
1. push "{"
2. push "{"
3. pop
Stack not empty $\Longrightarrow$ not balanced

1. push "{"
2. pop
Stack empty when last "}" encountered $\Longrightarrow$ not balanced

2. Bracket Matching

- Ensure that pairs of brackets are properly matched: e.g.

{a,(b+f[4])*3,d+f[5]}



- Other Examples:

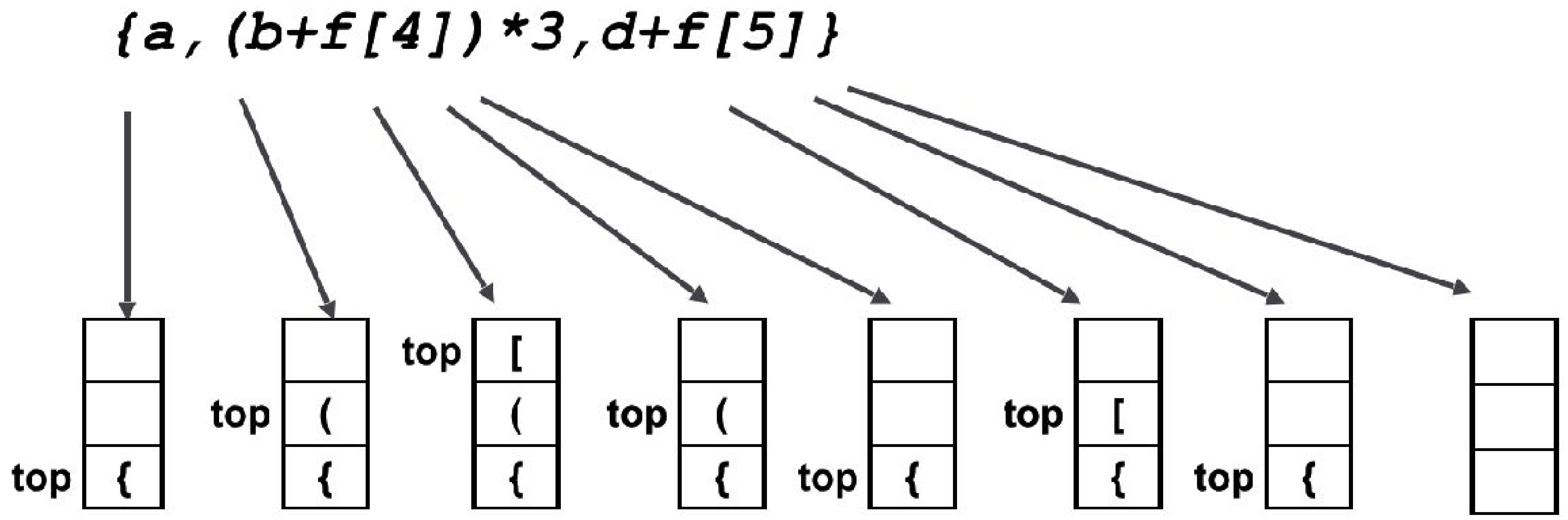
`(..)..)`	// too many closing brackets
`(..(..)`	// too many open brackets
`[..(..]..)`	// mismatched brackets

2. Bracket Matching - Algorithm

```
Initialise the stack to empty
For every char read
  • if it is an open bracket then push onto stack
  • if it is a close bracket, then
    • If the stack is empty, return ERROR
    • pop from the stack
    • if they don't match then return ERROR
  • if it is a non-bracket, skip the character
If the stack is NON-EMPTY, ERROR
```

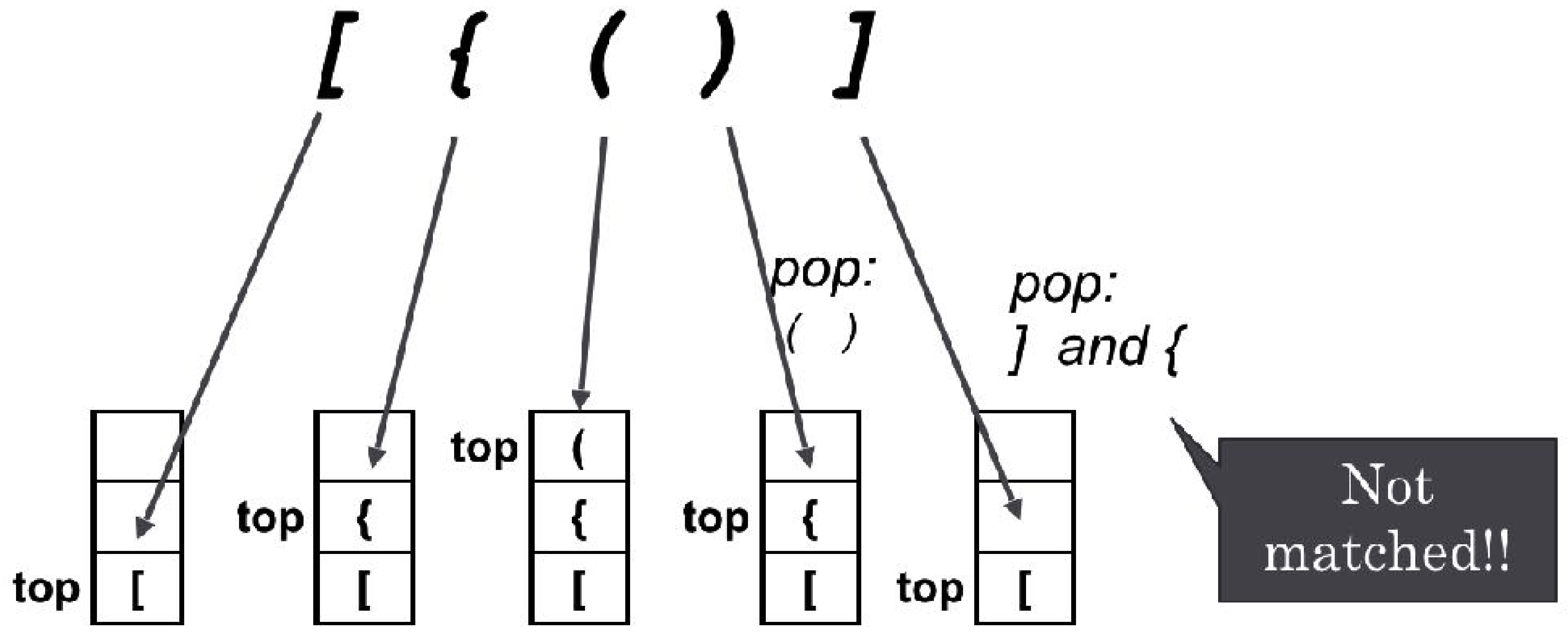
2. Bracket Matching - Examples

Example 1:



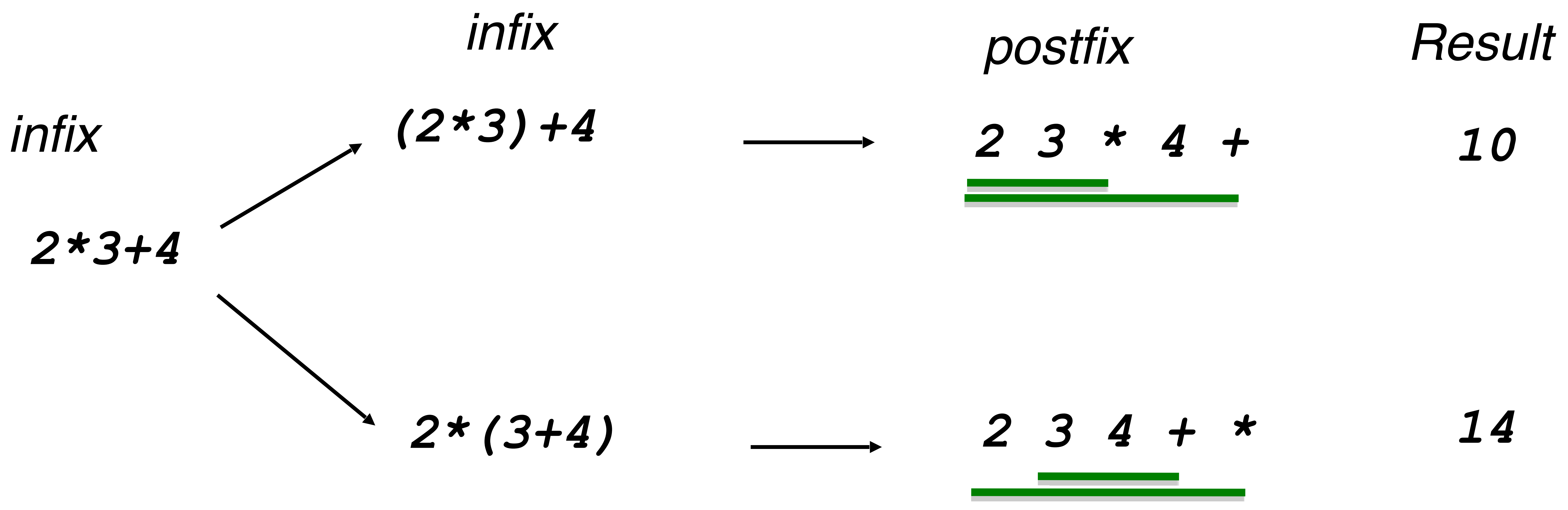
2. Bracket Matching - Examples

Example 2:



3. Postfix Calculator

- Computation of arithmetic expressions can be efficiently carried out in Postfix notation with the help of stack
  - Infix - arg1 op arg2
  - Prefix - op arg1 arg2
  - Postfix - arg1 arg2 op



3. Postfix Calculator

- Requires you to enter postfix expressions
  - Example: 2 3 4 + *
- Steps:

When an operand is entered,

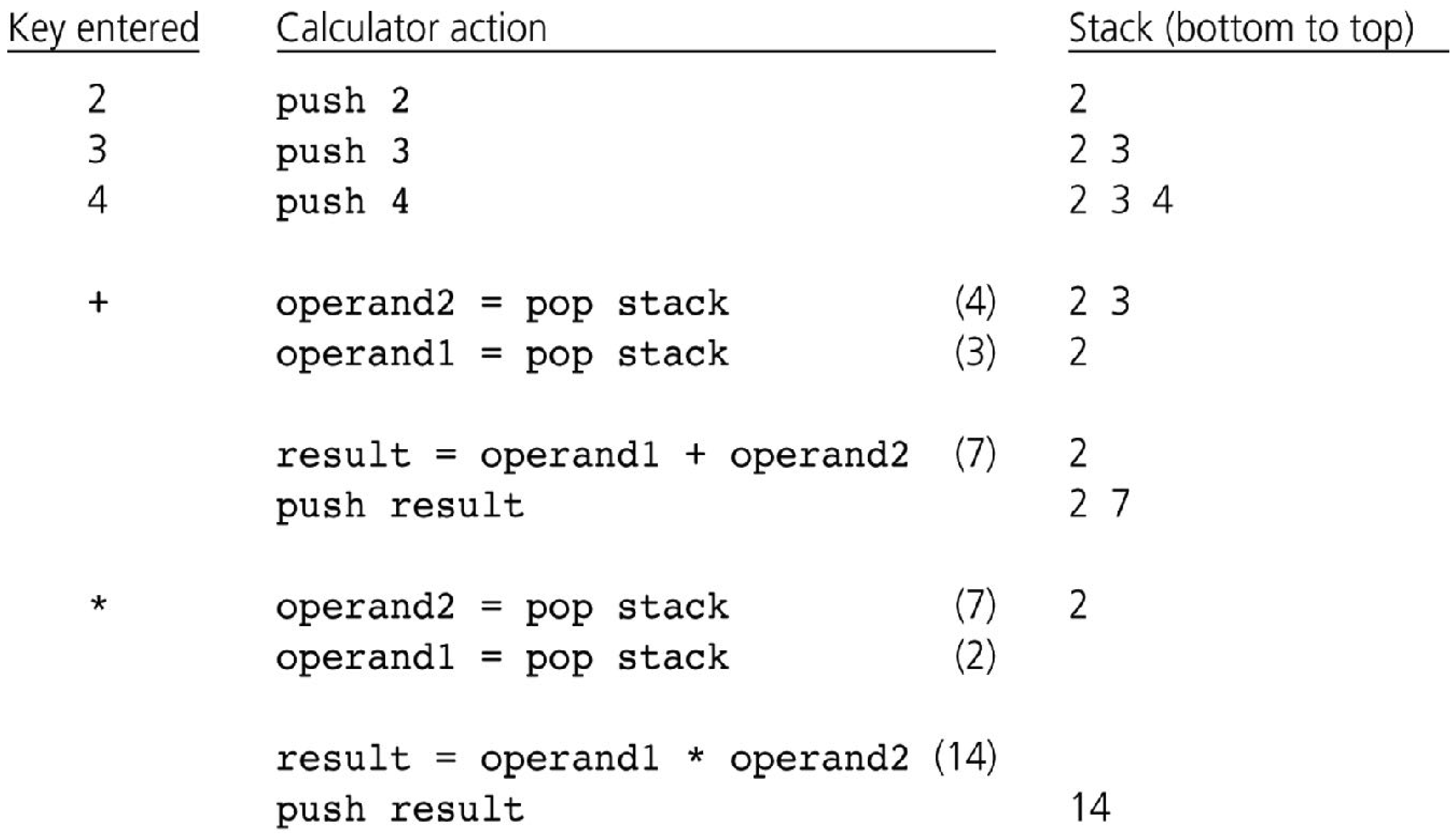
- *the calculator pushes it onto a stack*

When an operator is entered,

- *the calculator applies it to the top two operands of the stack*
  - *Pops the top two operands from the stack*
  - *Pushes the result of the operation on the stack*

3. Postfix Calculator - Example

Evaluating the expression: 2 3 4 + *



Key entered	Calculator action		Stack (bottom to top)
2	push 2		2
3	push 3		2 3
4	push 4		2 3 4
+	operand2 = pop stack	(4)	2 3
	operand1 = pop stack	(3)	2
	result = operand1 + operand2	(7)	2
	push result		2 7
*	operand2 = pop stack	(7)	2
	operand1 = pop stack	(2)	
	result = operand1 * operand2	(14)	
	push result		14

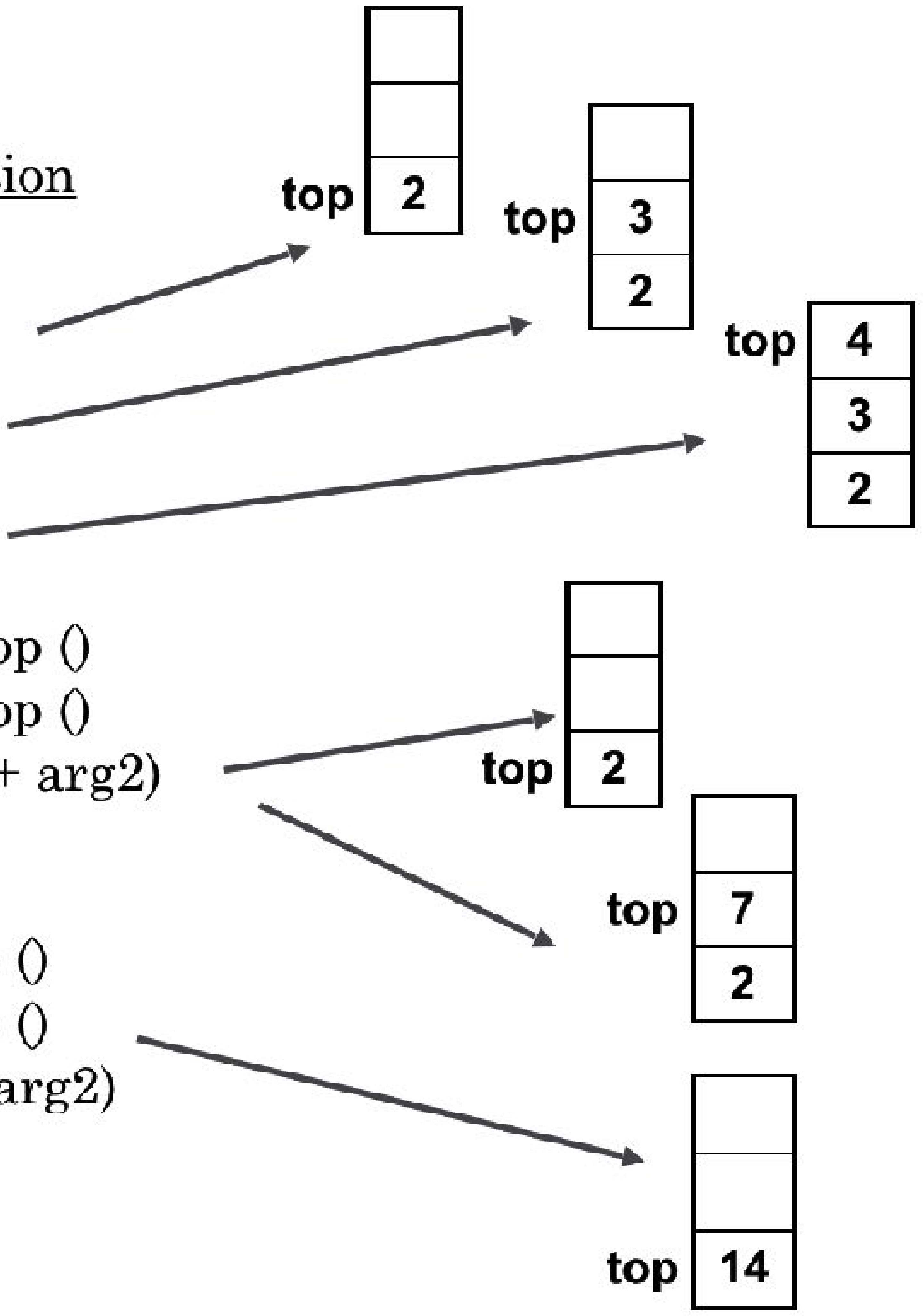
Evaluating the expression: 2 3 4 + *

Example

key entered	Calculator action
2	s.push (2)
3	s.push (3)
4	s.push (4)

+	arg2 = s.pop ()
	arg1 = s.pop ()
	s.push (arg1 + arg2)

*	arg2 = s.pop ()
	arg1 = s.pop ()
	s.push (arg1 * arg2)



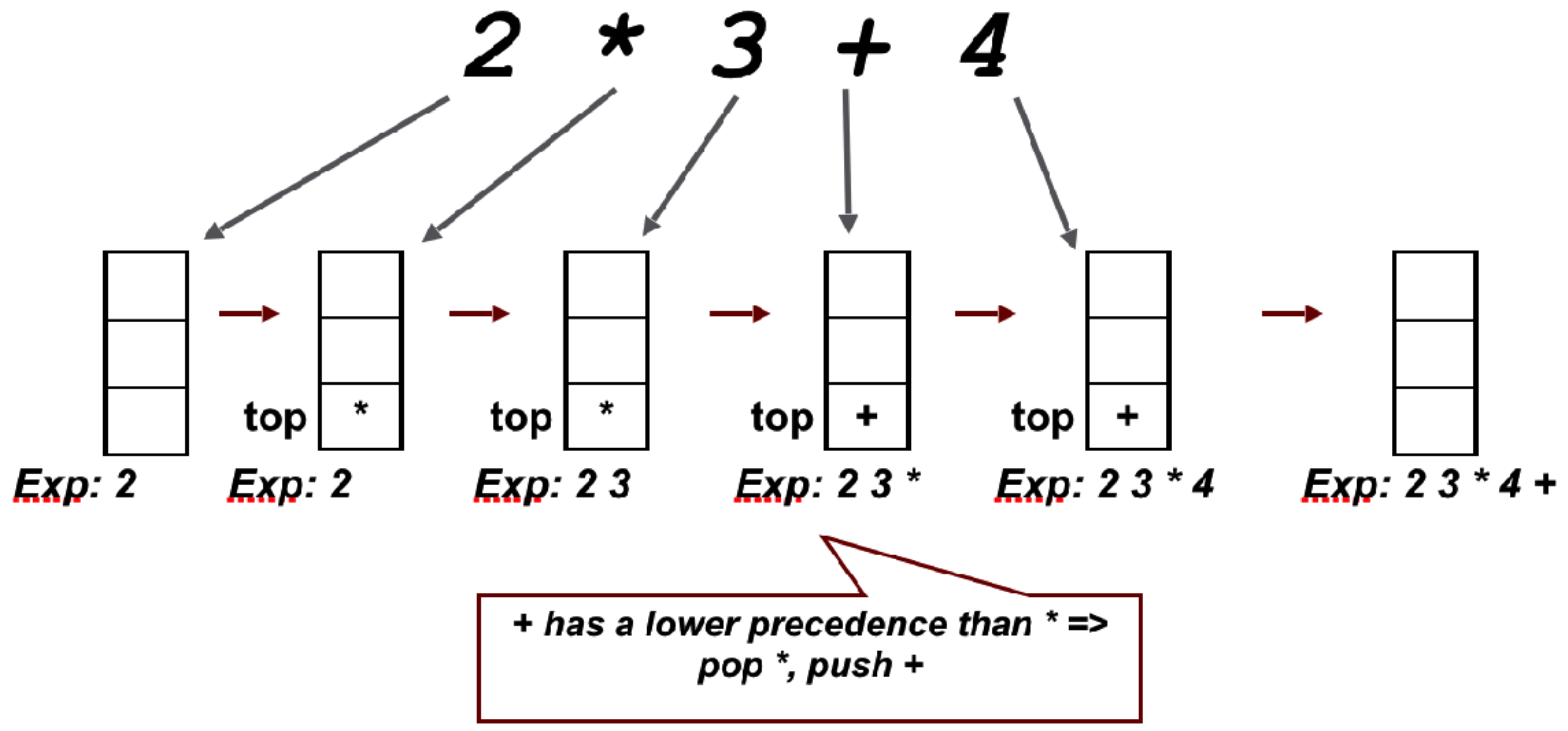
4. Conversion from Infix to Postfix

- Examples:
  - 2 * 3 + 4 => 2 3 * 4 +
  - 2 + 3 * 4 => 2 3 4 * +
  - 1 * 3 + 2 * 4 => 1 3 * 2 4 * +
- Steps:
  - Operands always stay in the same order with respect to one another
  - An operator will move only “to the right” with respect to the operands
  - All parentheses are removed

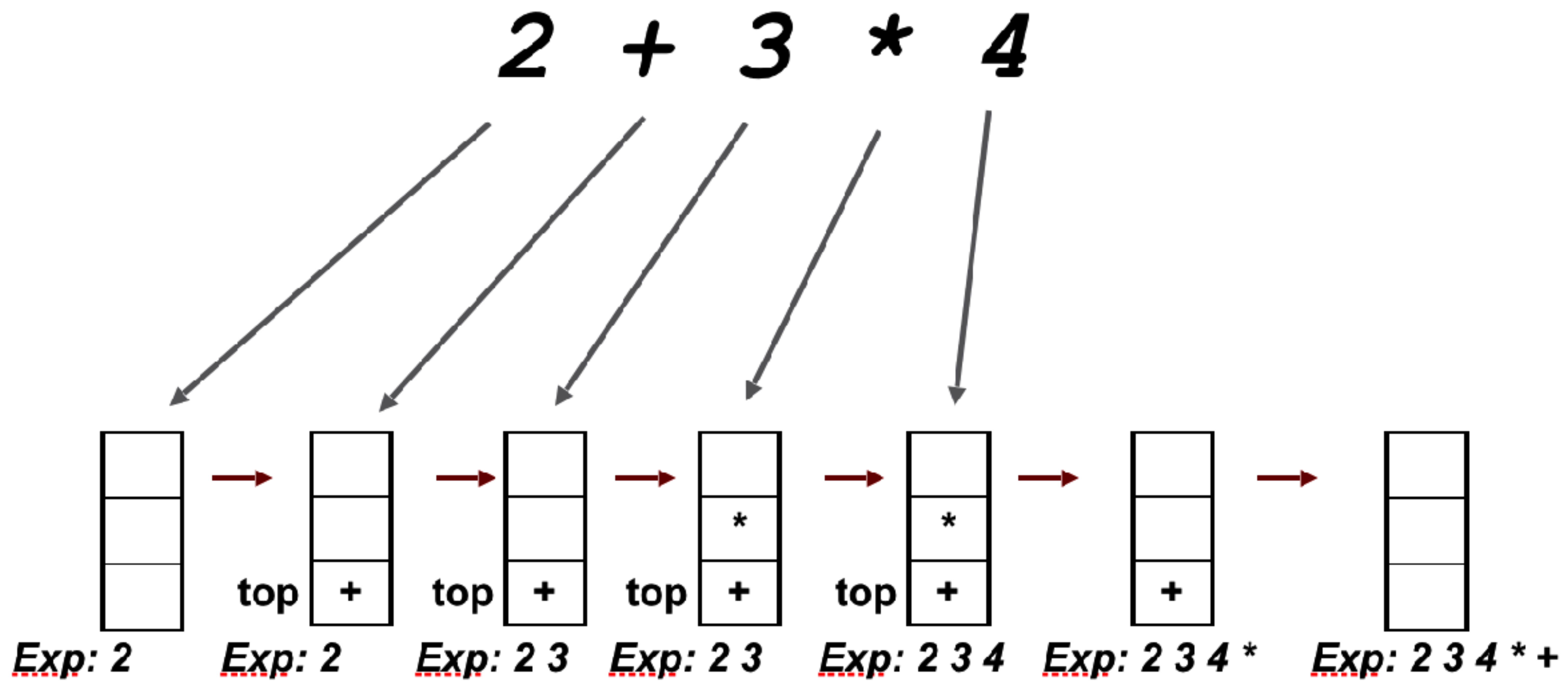
Algorithm

- operand – append it to the output string postfixExp
- “(“ – push onto the stack
- operator
  - If the stack is empty, push the operator onto the stack
  - If the stack is non-empty
    - Pop the operators of greater or equal precedence from the stack and append them to postfixExp
    - Stop when encounter either a “(“ or an operator of lower precedence or when the stack is empty
    - Push the new operator onto the stack
- “)” – pop the operators off the stack and append them to the end of postfixExp until encounter the match “(“
- End of the string – append the remaining contents of the stack to postfixExp

Examples - Converting Infix to Postfix



Examples - Converting Infix to Postfix



Conversion from Infix to Postfix

a – (b + c ∗ d) / e

ch	stack (bottom to top)	postfixExp	
a		a	
–	–	a	
(	– (	a	
b	– (	ab	
+	– (+	ab	
c	– (+	abc	
*	– (+ *	abc	
d	– (+ *	abcd	
)	– (+	abcd*	Move operators
	– (	abcd*+	from stack to
	–	abcd*+	postfixExp until " ("
/	– /	abcd*+	
e	– /	abcd*+e	Copy operators from
		abcd*+e/–	stack to postfixExp

Questions?

@rschifan

schifane@di.unito.it

http://www.di.unito.it/~schifane