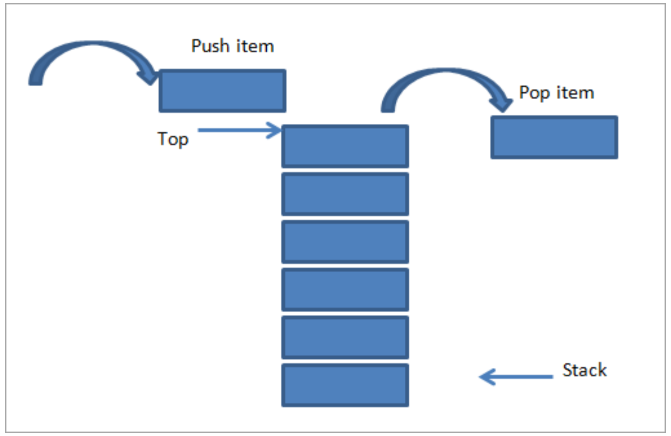
Stacks

CS223: Data Structures

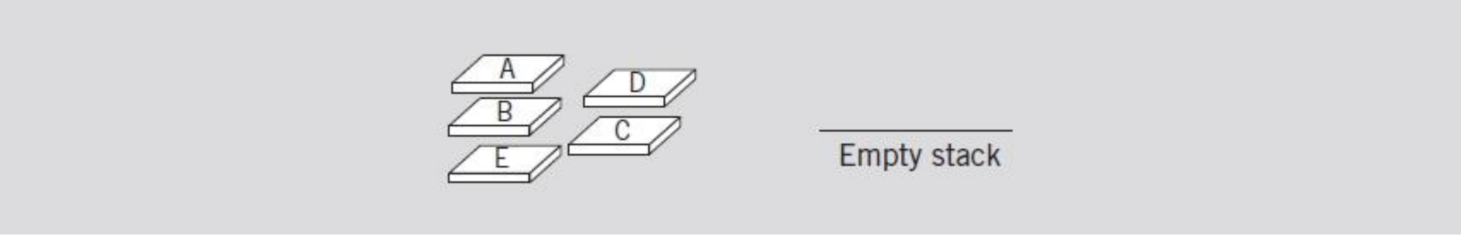
Stack

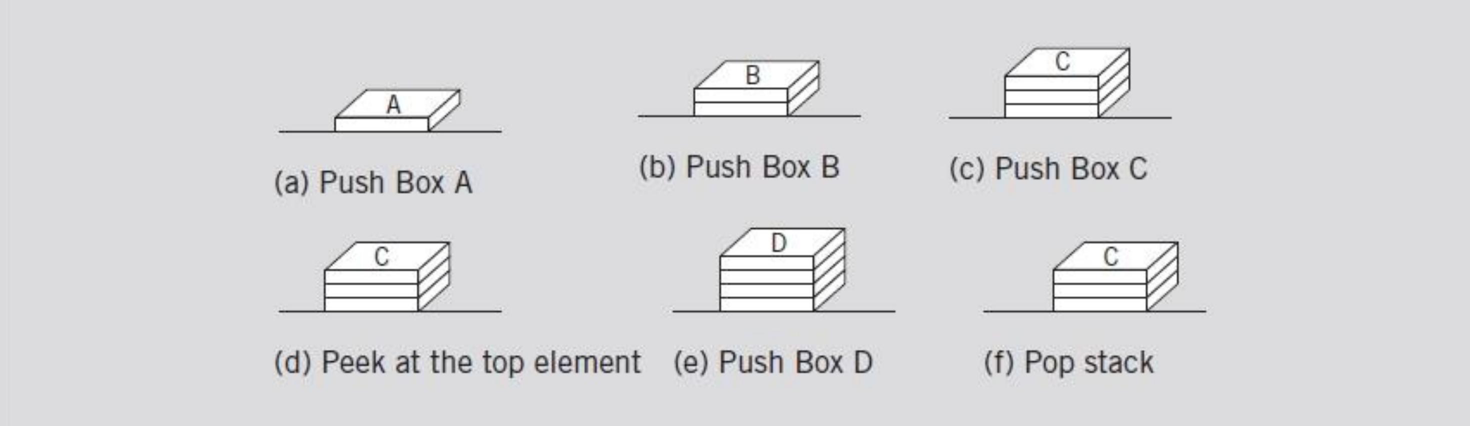
- **Stack**: A list with the restriction that insertions are done at one end and deletions are done at the same end.
  - Last-In, First-Out ("LIFO"): Items are removed from a stack in the reverse order from the way they were inserted
  - Addition and deletion of elements occur only at one end, called the top of the stack.



- Basic stack operations:
  - add (push): Add an element to the top of the stack.
  - remove (pop): Remove an element from the top of the stack.
  - top: retrieve the top element of the stack

Stacks (cont'd.)





(a) Push Box A

(b) Push Box B

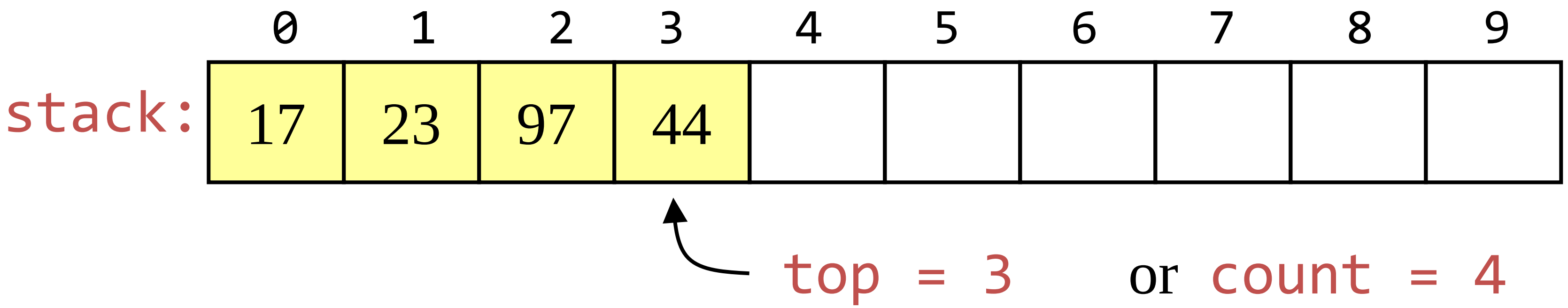
(c) Push Box C

(d) Peek at the top element

(e) Push Box D

(f) Pop stack

Pushing and Popping



	0	1	2	3	4	5	6	7	8	9
stack:	17	23	97	44						

top = 3 or count = 4

- If the bottom of the stack is at location 0, then an empty stack is represented by top = -1 or count = 0
- To add (push) an element, either:
  - Increment top and store the element in stack[top], or
  - Store the element in stack[count] and increment count
- To remove (pop) an element, either:
  - Get the element from stack[top] and decrement top, or
  - Decrement count and get the element in stack[count]

Operations on a Stack

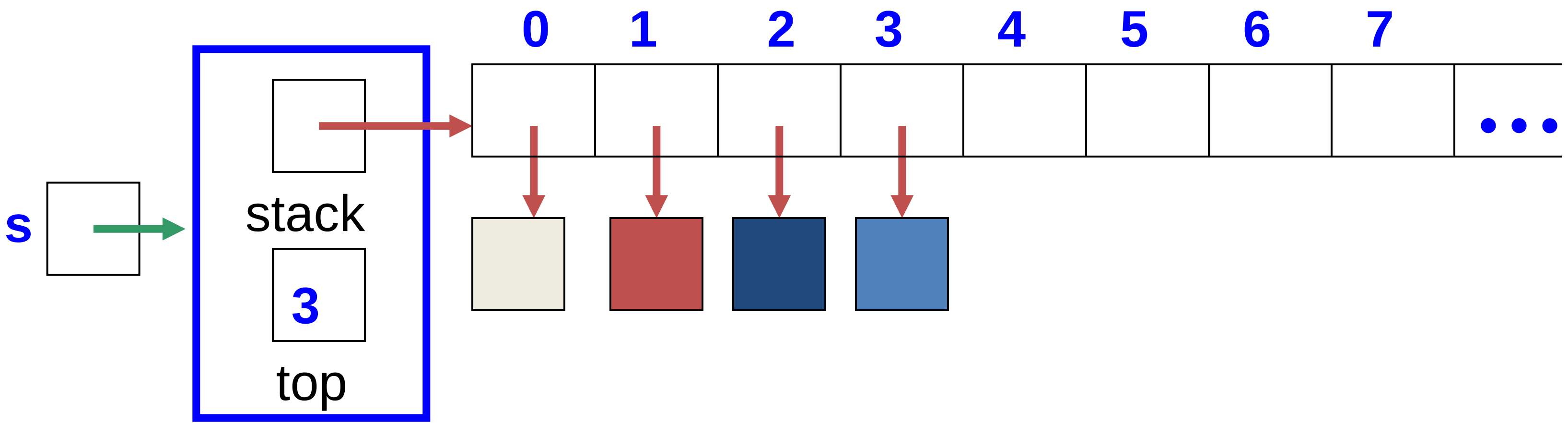
Operation	Description
Push/add	Adds an element to the top of the stack
Pop/remove	Removes an element from the top of the stack
Peek	Examines the element at the top of the stack
isEmpty	Determines whether the stack is empty
size	Determines the number of elements in the stack

Stack Implementation

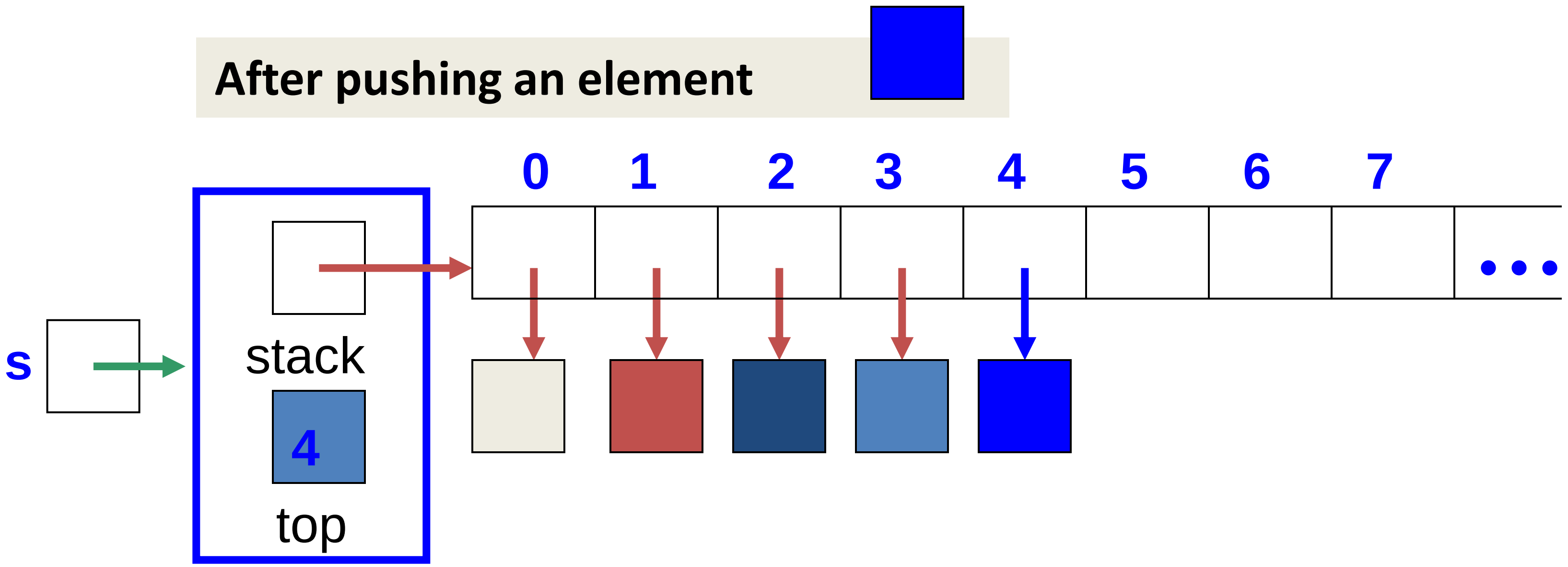
1. Using Array
2. Using Linked List

Array Implementation of a Stack

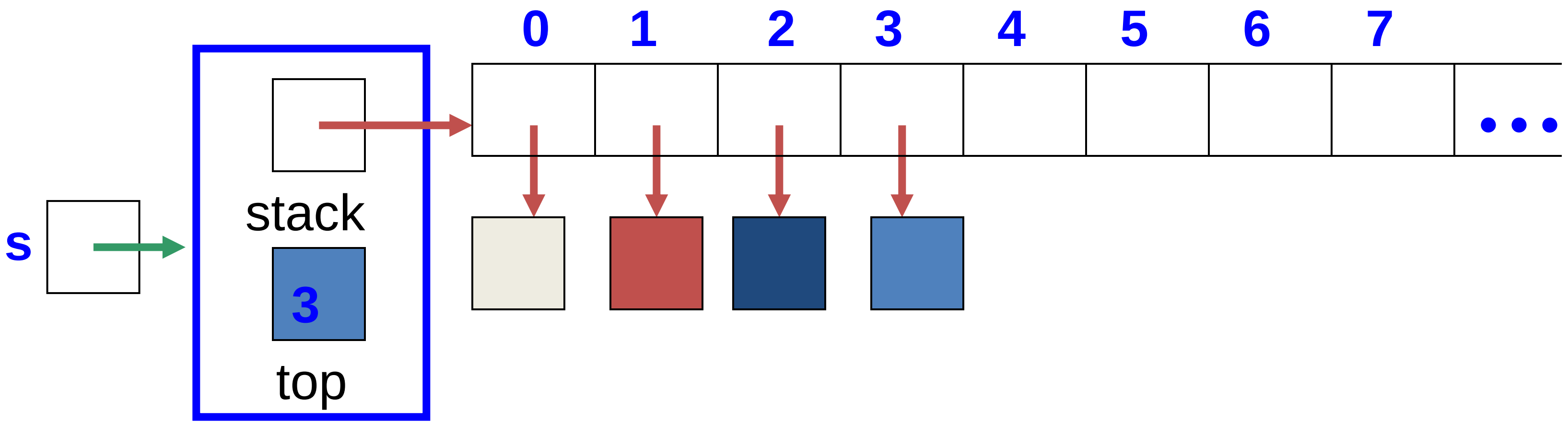
A Stack s with 4 elements



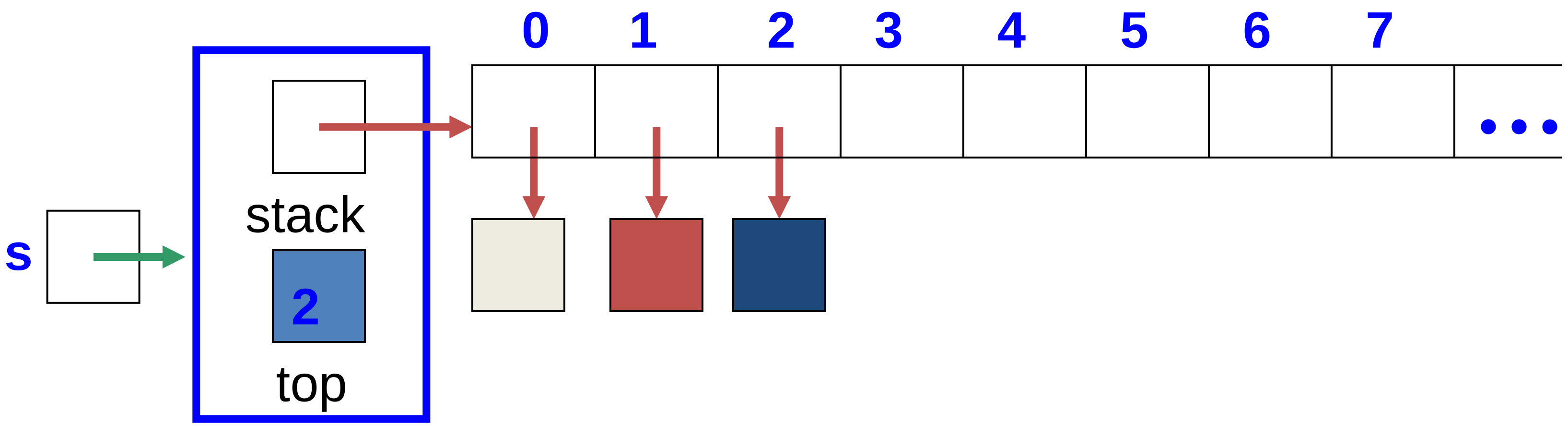
After pushing an element



After popping one element



After popping a second element



Stack – Array: Push Operation

```
push(value)

  if top equal to MAXSIZE - 1
     write “Stack is Full”
  else
     top = top + 1
     stack[top] = value
```

Stack – Array: Pop Operation

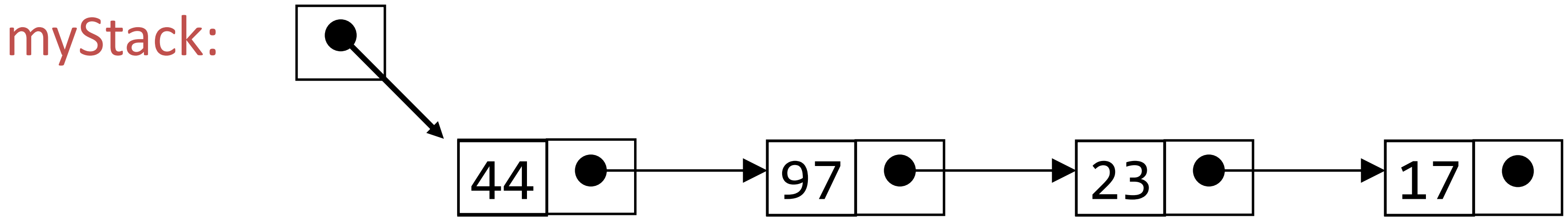
```
 **pop()**
if top equals to -1
       write “Stack is Empty”
   else
      value = stack[top]
       top  = top - 1
       return value
```

Stack Implementation

1. Using Array
2. Using Linked List

Stack Implementation using Linked-list

- Since all the actions happen at the top of a stack, a **singly-linked list** is a fine way to implement it
- The header of the list points to the top of the stack



- **Pushing** is the action of inserting an element at the beginning of the list
- **Popping** is the action of removing an element from the beginning of the list

Stack Implementation using Linked List: Push Operation

```
push (value)

    Allocate the space for the new node PTR
    Set PTR -> DATA = value
    if TOP equal to NULL
       Set TOP = PTR
       Set TOP -> NEXT = NULL
    else
       Set PTR -> NEXT = TOP
       Set TOP = PTR
```

Stack Implementation using Linked List: Pop Operation

```
pop ()

        if TOP equal to NULL
           write “Stack is Empty”
        Else
           Set PTR = TOP
           Set TOP = PTR -> NEXT
           Delete PTR
```

Motivation - Stacks Applications

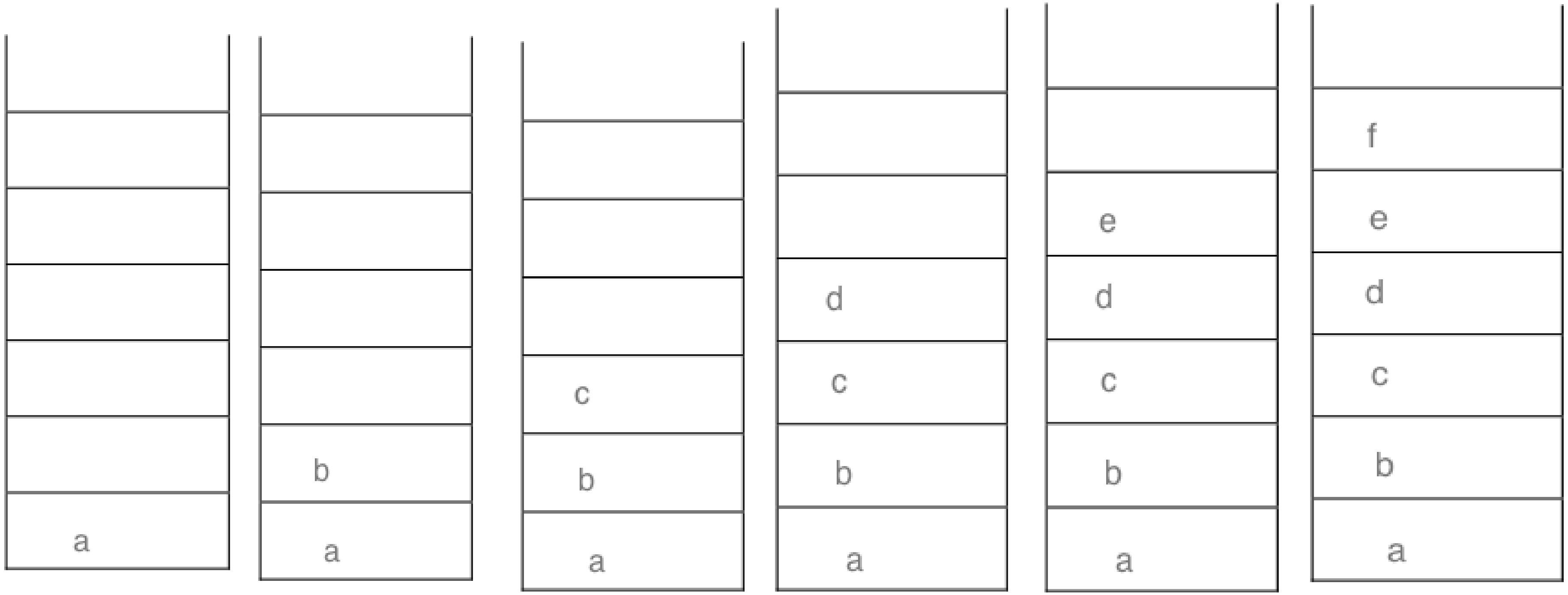
- Line editing
- Reverse a string
- Bracket matching
- Postfix calculation
- Function call stack
- Browsers: to keep track of pages visited in a browser tab

Reversing Strings:

A simple application of stack is reversing strings.

- To reverse a string, the characters of string are pushed onto the stack one by one as the string is read from left to right.
- Once all the characters of string are pushed onto stack, they are popped one by one.
- Since the character last pushed in comes out first, subsequent pop operation results in the reversal of the string.

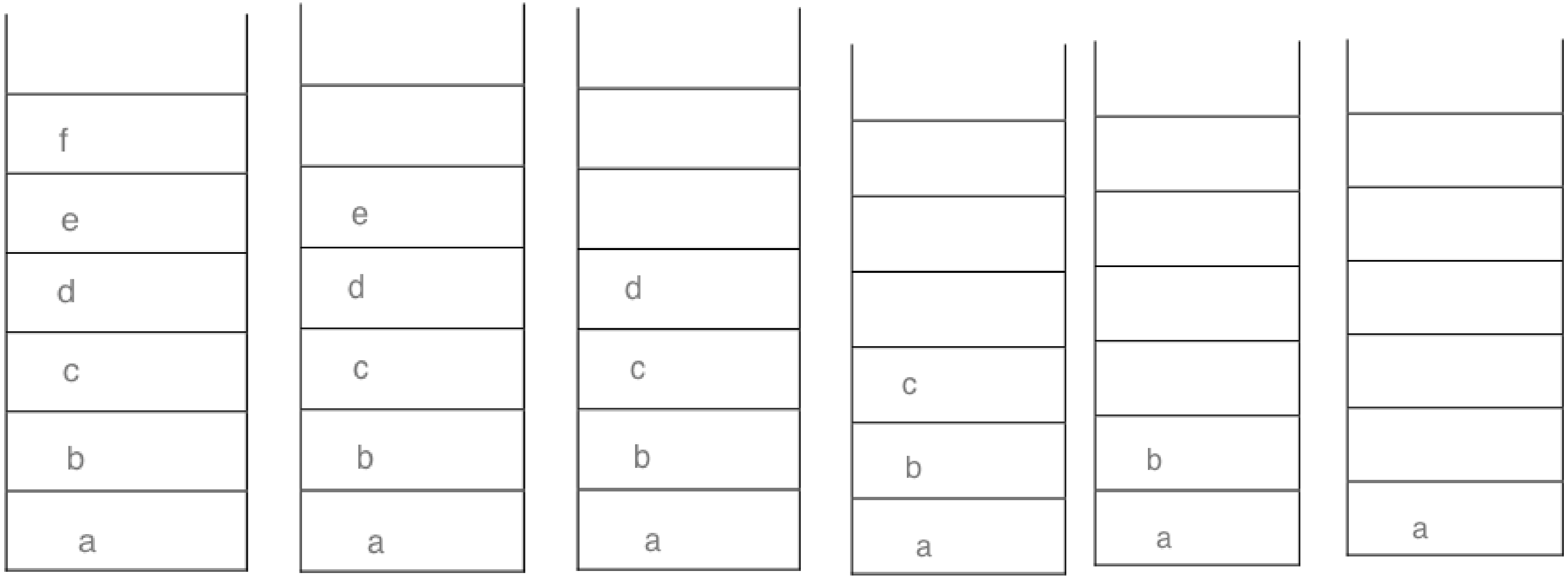
Reverse String...



String is "a b c d e f"

PUSH to STACK

Reverse String...



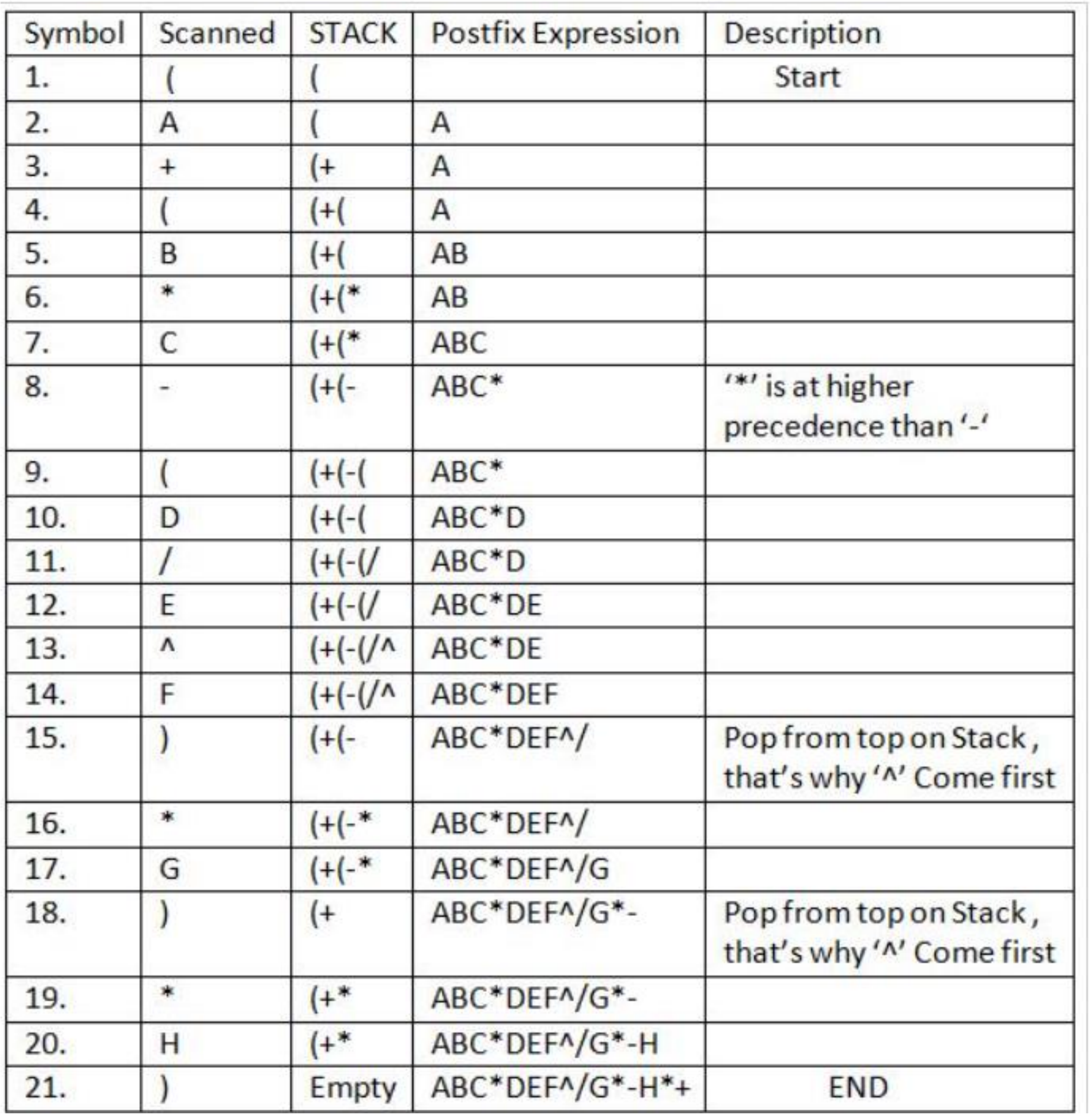
Reversed String: "f e d c b a“

POP from STACK

Infix to Postfix Conversion

- Read all the symbols one by one from left to right in the given Infix Expression.
- If the reading symbol is **operand**, then directly print it to the result (Output).
- If the reading symbol is left parenthesis '**(**', then Push it on to the Stack.
- If the reading symbol is right parenthesis '**)**', then Pop all the contents of stack until respective left parenthesis is popped and print each popped symbol to the result.
- If the reading symbol is **operator** (+ , - , * , / etc.,), then Push it on to the Stack. However, first pop the operators which are already on the stack that have higher or equal precedence than current operator and print them to the result.

Infix to Postfix Conversion Example 1

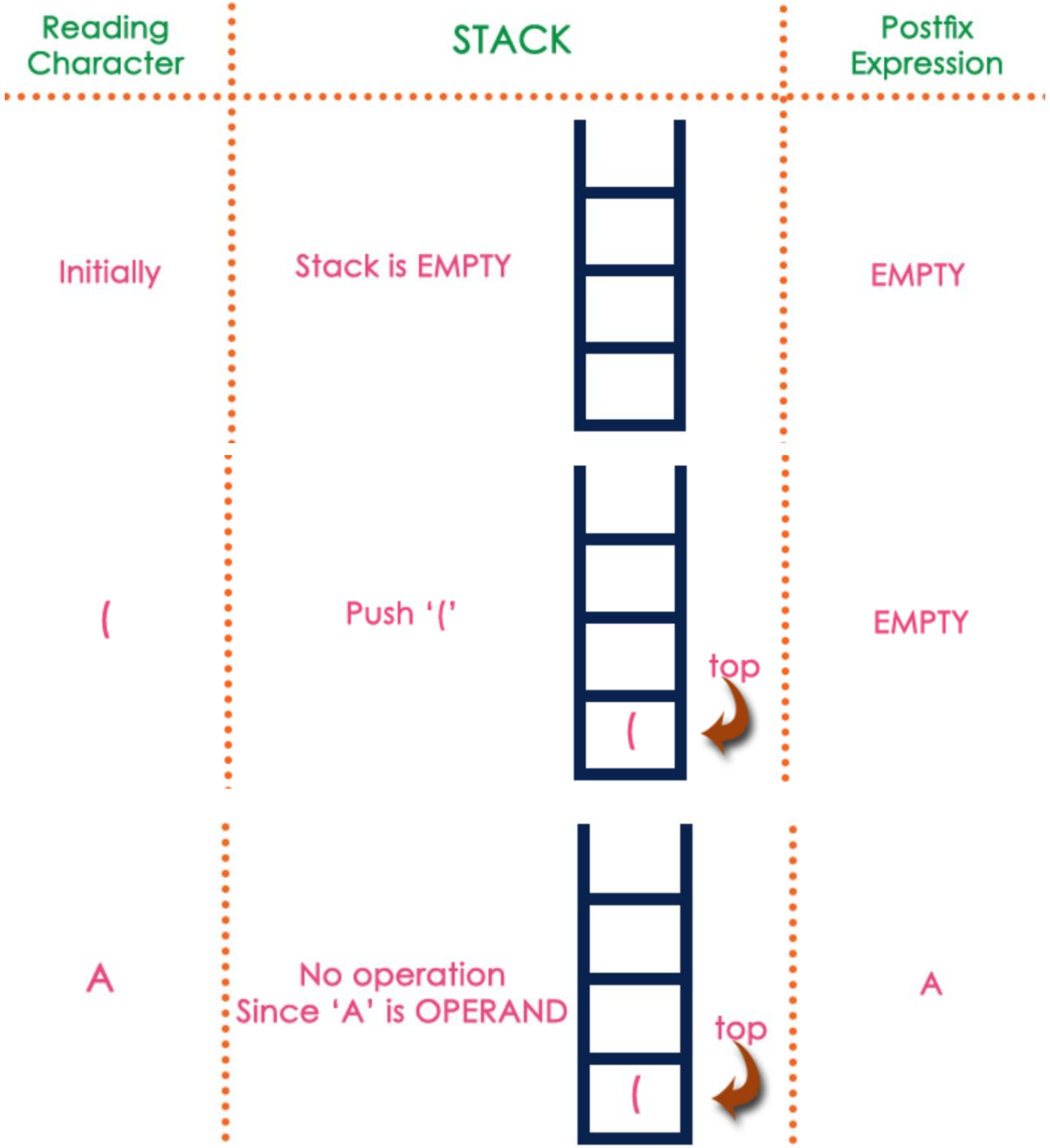


Symbol	Scanned	STACK	Postfix Expression	Description
1.	(	(		Start
2.	A	(	A	
3.	+	(+	A	
4.	(	(+(	A	
5.	B	(+(	AB	
6.	*	(+(*	AB	
7.	C	(+(*	ABC	
8.	-	(+(-	ABC*	‘*’ is at higher precedence than ‘-‘
9.	(	(+(-(	ABC*	
10.	D	(+(-(	ABC*D	
11.	/	(+(-(/	ABC*D	
12.	E	(+(-(/	ABC*DE	
13.	^	(+(-(/^	ABC*DE	
14.	F	(+(-(/^	ABC*DEF	
15.	)	(+(-	ABC*DEF^/	Pop from top on Stack , that’s why ‘^’ Come first
16.	*	(+(-*	ABC*DEF^/	
17.	G	(+(-*	ABC*DEF^/G	
18.	)	(+	ABC*DEF^/G*-	Pop from top on Stack , that’s why ‘^’ Come first
19.	*	(+*	ABC*DEF^/G*-	
20.	H	(+*	ABC*DEF^/G*-H	
21.	)	Empty	ABC*DEF^/G*-H*+	END

(A+ (B*C-(D/E^F)*G)*H)

Resultant Postfix Expression: ABC*DEF^/G*-H*+

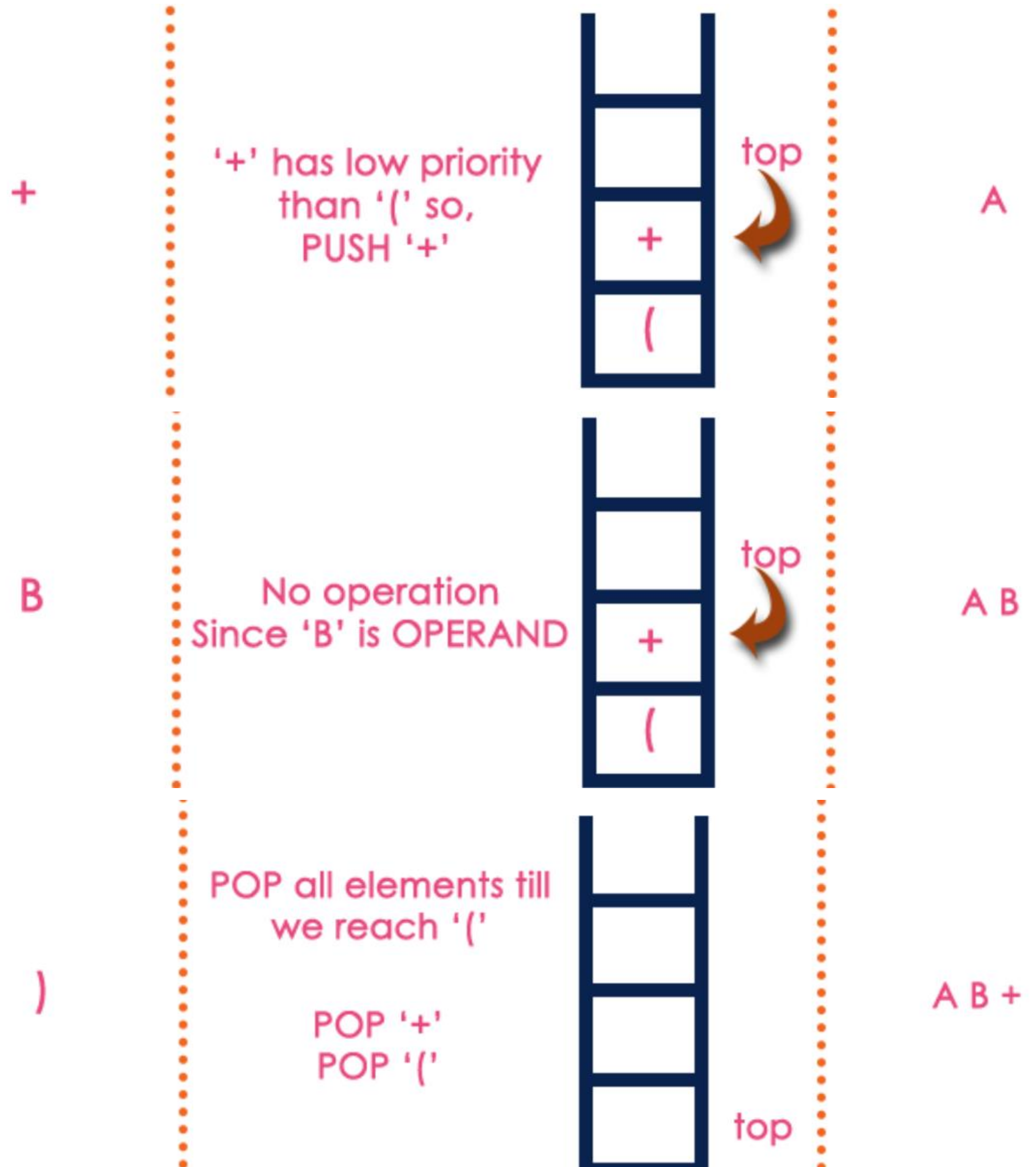
Infix to Postfix Conversion Example 2



Reading Character	STACK	Postfix Expression
Initially	Stack is EMPTY	EMPTY
(	Push ‘(’	EMPTY
A	No operation Since ‘A’ is OPERAND	A

(A + B) * (C - D)

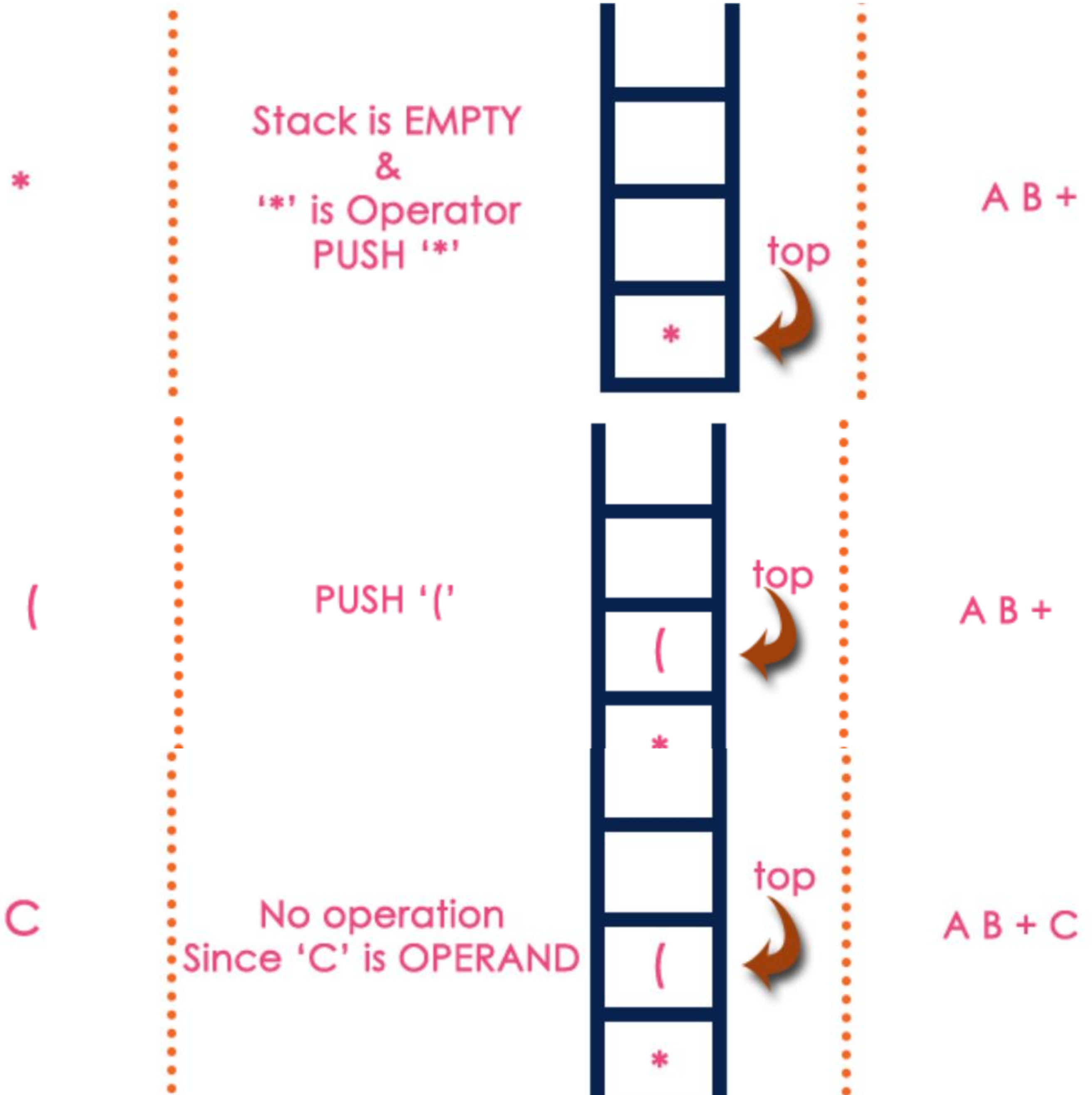
Infix to Postfix Conversion Example 2



+	'+' has low priority than '(' so, PUSH '+'	+ (	A
B	No operation Since 'B' is OPERAND	+ (	A B
)	POP all elements till we reach '(' POP '+' POP '('		A B +

(A + B) * (C - D)

Infix to Postfix Conversion Example 2

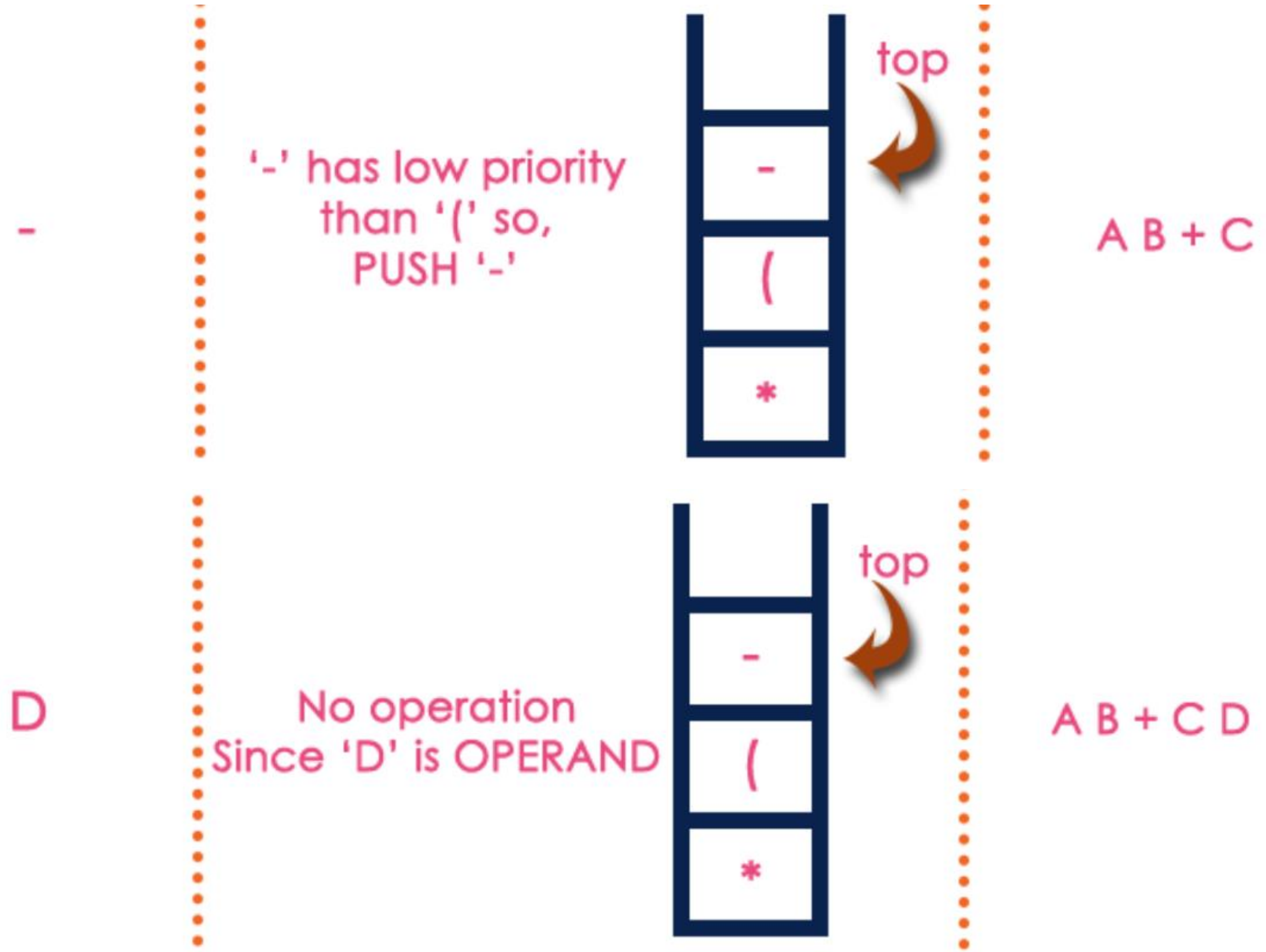


*	Stack is EMPTY & '*' is Operator PUSH '*'	top *	A B +
(	PUSH '('	top (*	A B +
C	No operation Since 'C' is OPERAND	top (*	A B + C

(A + B) * (C - D)

Infix to Postfix Conversion Example 2

(A + B) * (C - D)



-	'-' has low priority than '(' so, PUSH '-'	A B + C
D	No operation Since 'D' is OPERAND	A B + C D

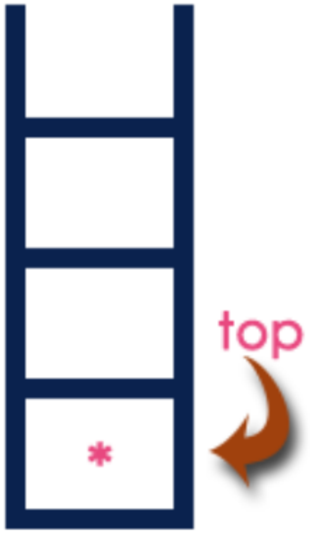
Infix to Postfix Conversion Example 2

(A + B) * (C - D)

)

POP all elements till we reach ‘(’

POP ‘-’
POP ‘(’



A B + C D -

$

POP all elements till Stack becomes Empty

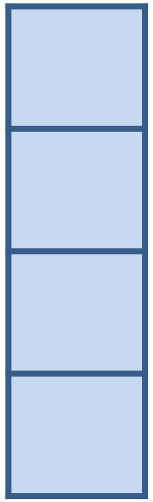
A B + C D - *

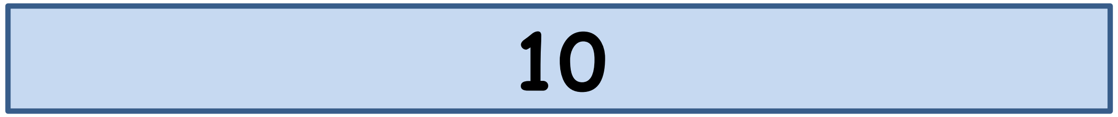
Result Postfix Expression: **A B + C D - ***

Infix to Postfix Conversion Example 3

Ex: 10 + 2 * 8 - 3

- We see the first number 10, output it



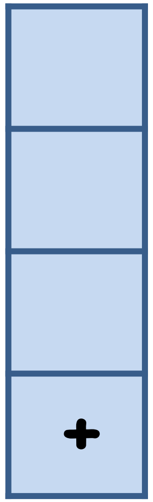


10

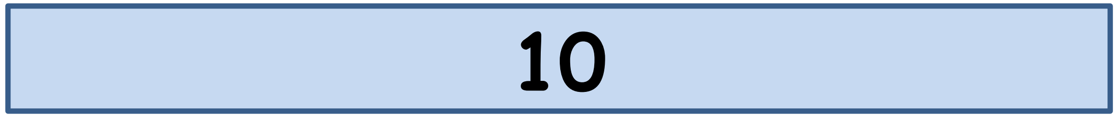
Infix to Postfix Conversion Example 3

Ex: 10 + 2 * 8 - 3

- We see the first operator +, push it into the stack



+

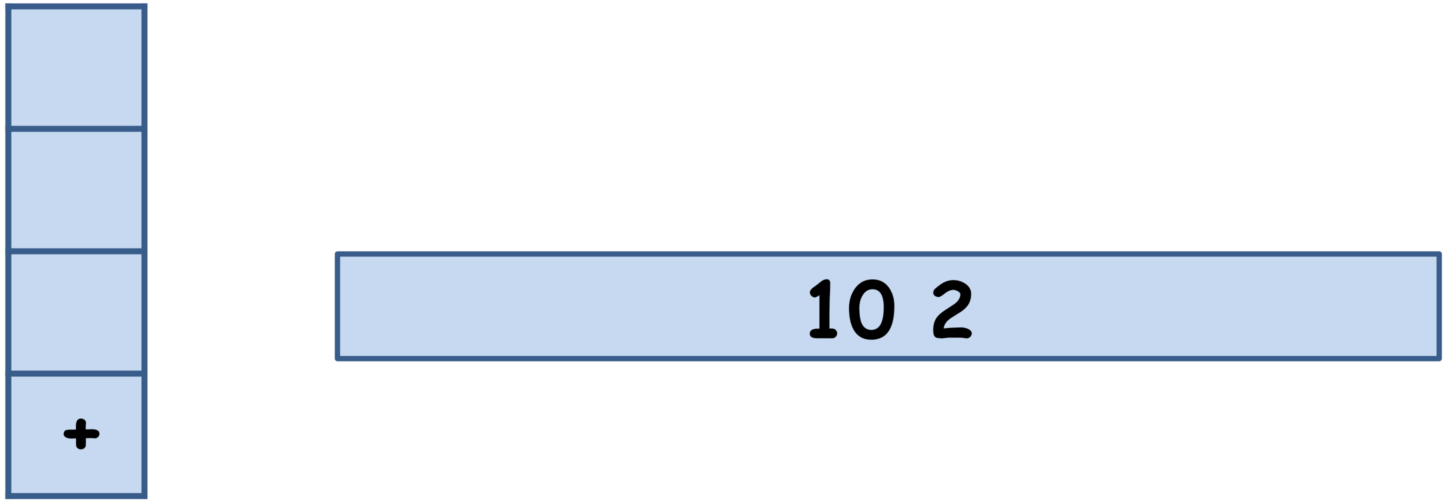


10

Infix to Postfix Conversion Example 3

Ex: 10 + 2 * 8 - 3

- We see the number 2, output it



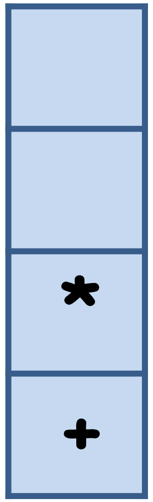
+

10 2

Infix to Postfix Conversion Example 3

Ex: 10 + 2 * 8 - 3

- We see the operator *, since the top operator in the stack, +, has lower priority then *, push(*)

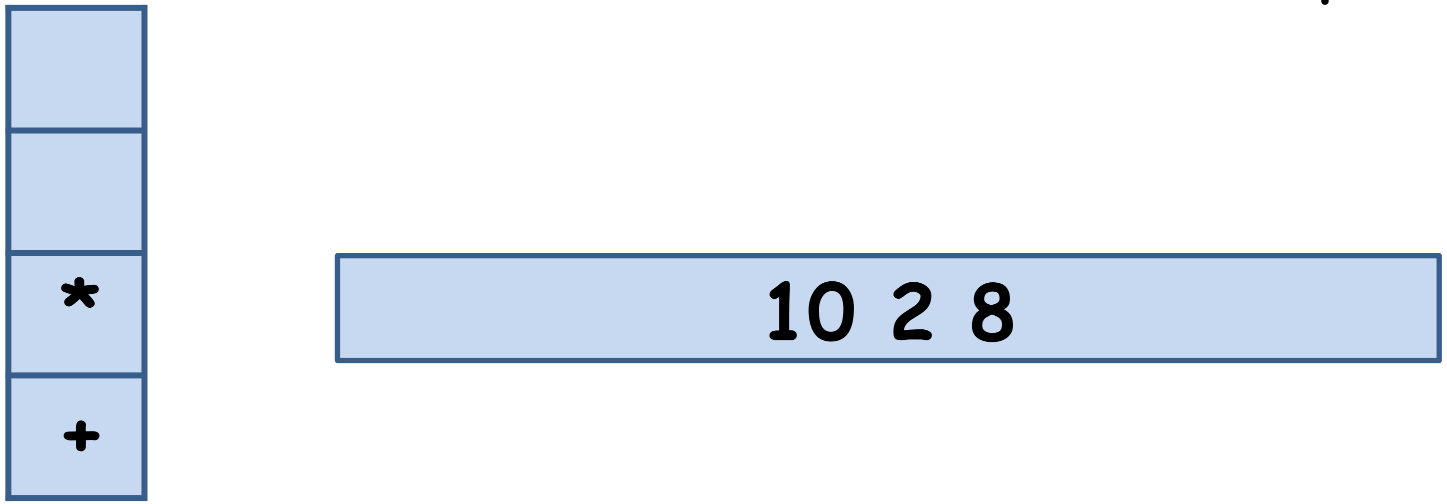


10 2

Infix to Postfix Conversion Example 3

Ex: 10 + 2 * 8 - 3

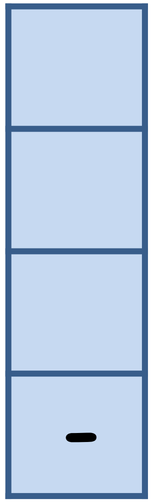
- We see the number 8, output it



Infix to Postfix Conversion Example 3

Ex: 10 + 2 * 8 - 3

- We see the operator -, because its priority is lower then *, we pop. Also, because + is on the left of it, we pop +, too. Then we push(-)

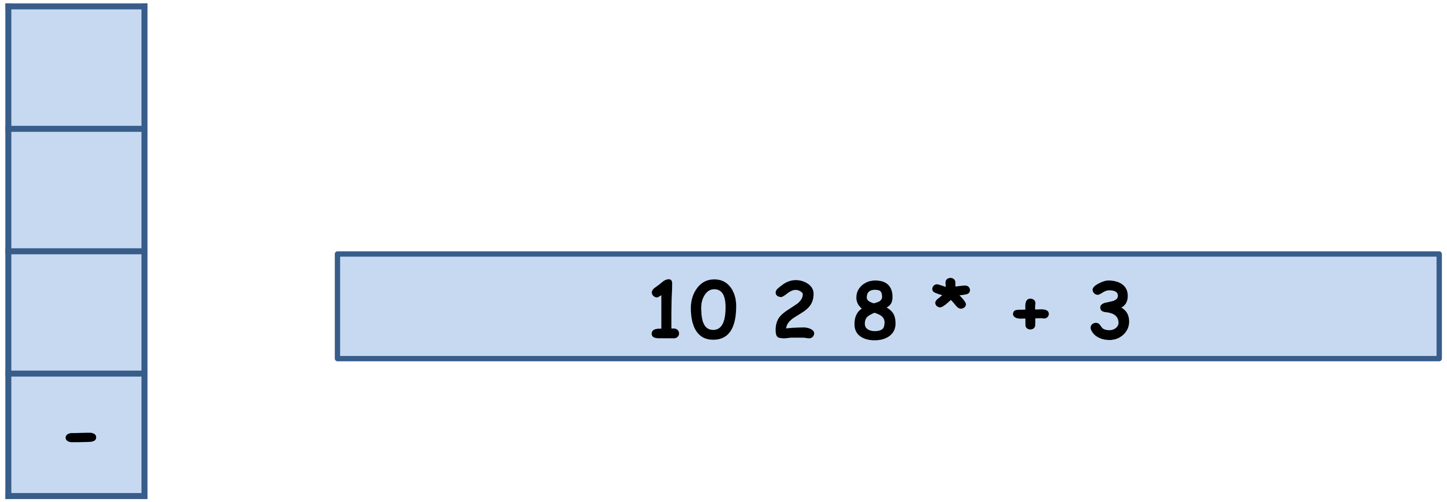


10 2 8 * +

Infix to Postfix Conversion Example 3

Ex: 10 + 2 * 8 - 3

- We see the number 3, output it



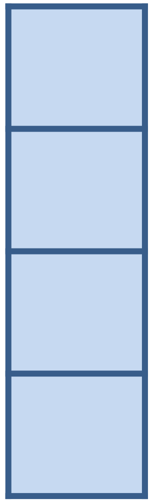
-

10 2 8 * + 3

Infix to Postfix Conversion Example 3

Ex: 10 + 2 * 8 - 3

- Because the expression is ended, we pop all the operators in the stack



10 2 8 * + 3 -

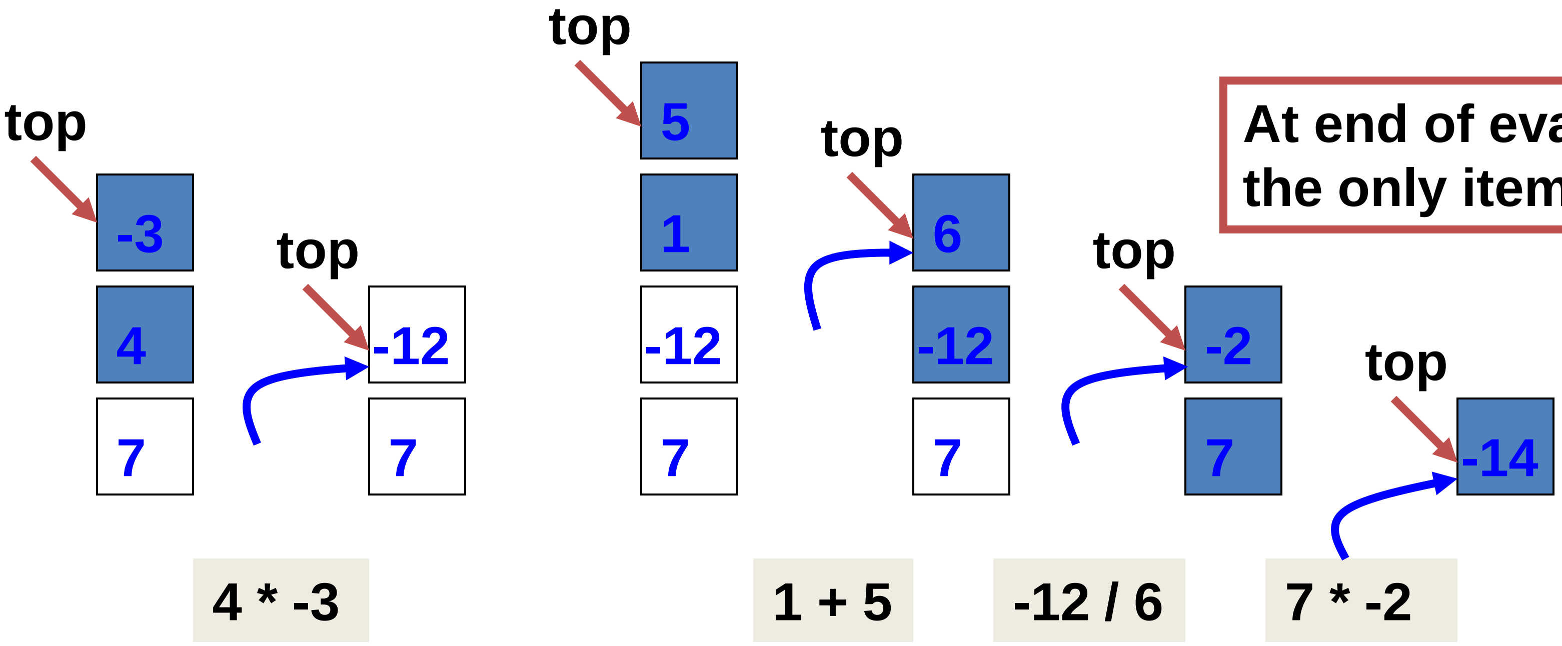
Evaluating Postfix Expressions

1. Create a stack to store operands (or values).
2. Scan the given expression and do following for every scanned element.
    - If the element is a number, push it into the stack
    - If the element is an operator, pop operands for the operator from stack. Evaluate the operator and push the result back to the stack
3. When the expression is ended, the number in the stack is the final answer

Evaluate a Postfix Expression Example 1

Evaluation of

7 4 -3 * 1 5 + / *

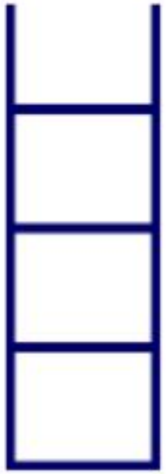


At end of evaluation, the result is the only item on the stack

Evaluate a Postfix Expression Example 2

Infix Expression (5 + 3) * (8 - 2)

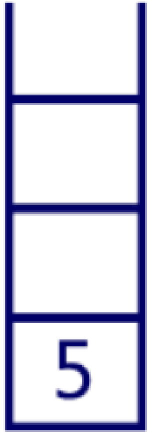
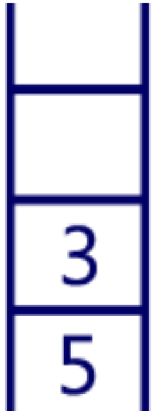
Postfix Expression 5 3 + 8 2 - *

Reading Symbol	Stack Operations	Evaluated Part of Expression
Initially	Stack is Empty 	Nothing

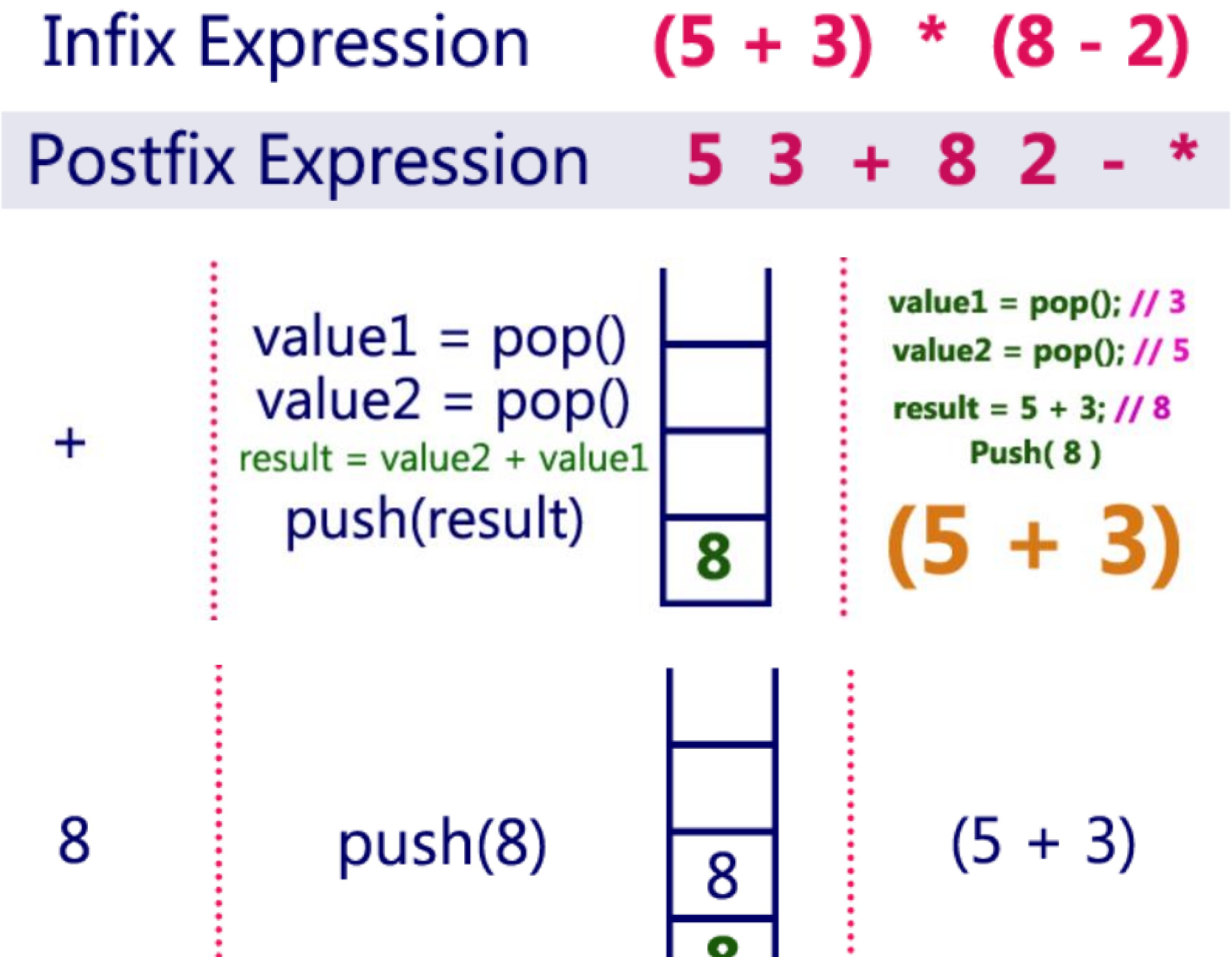
Evaluate a Postfix Expression Example 2

Infix Expression (5 + 3) * (8 - 2)

Postfix Expression 5 3 + 8 2 - *

5	push(5)		Nothing
3	push(3)		Nothing

Evaluate a Postfix Expression Example 2



Infix Expression (5 + 3) * (8 - 2)

Postfix Expression 5 3 + 8 2 - *

+	value1 = pop()
	value2 = pop()
	result = value2 + value1
	push(result)
		8	value1 = pop(); // 3
			value2 = pop(); // 5
			result = 5 + 3; // 8
			Push(8)
			(5 + 3)

8	push(8)	8 8	(5 + 3)

Evaluate a Postfix Expression Example 2

Infix Expression (5 + 3) * (8 - 2)

Postfix Expression 5 3 + 8 2 - *

2	push(2)	2 8 8	(5 + 3)

-	value1 = pop()
	value2 = pop()
	result = value2 - value1
	push(result)
		6
		8
			value1 = pop(); // 2
			value2 = pop(); // 8
			result = 8 - 2; // 6
			Push(6)
			(8 - 2)
			(5 + 3) , (8 - 2)

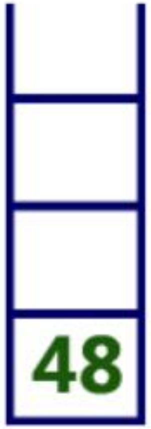
Evaluate a Postfix Expression Example 2

Infix Expression (5 + 3) * (8 - 2)

Postfix Expression 5 3 + 8 2 - *

*

value1 = pop()
value2 = pop()
result = value2 * value1
push(result)



```
value1 = pop(); // 6
value2 = pop(); // 8
result = 8 * 6; // 48
Push( 48 )
```

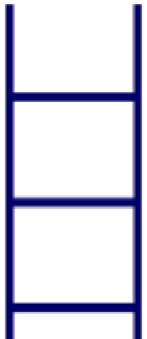
(6 * 8)

(5 + 3) * (8 - 2)

$

End of Expression

result = pop()



Display (result)

48

As final result