

# DATA STRUCTURES

## UNIT-1

### Stack Data Structure

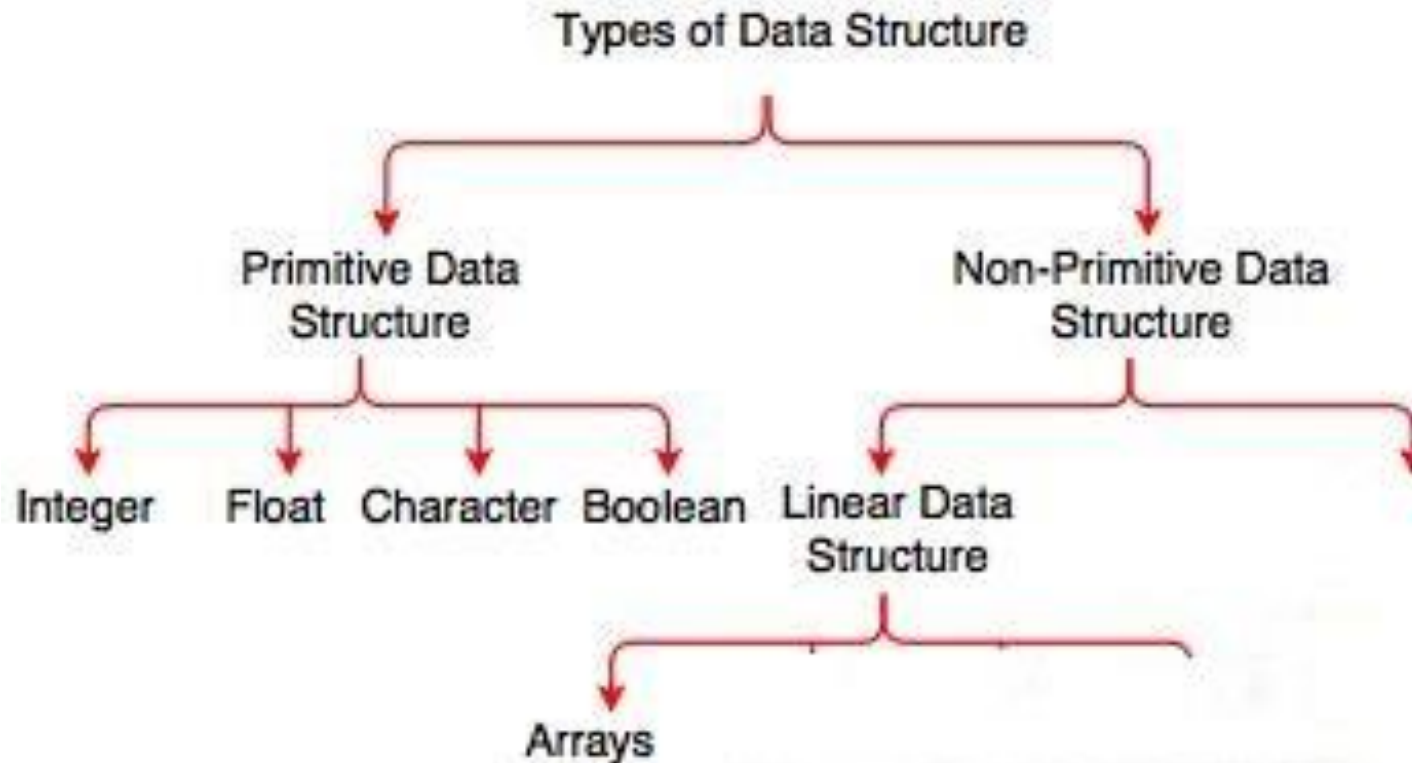
# Topics

- Classification of Data Structures
- Introduction to Stack
- Operations on Stacks
- Stack ADT
- Stack Implementation
- Applications of Stacks

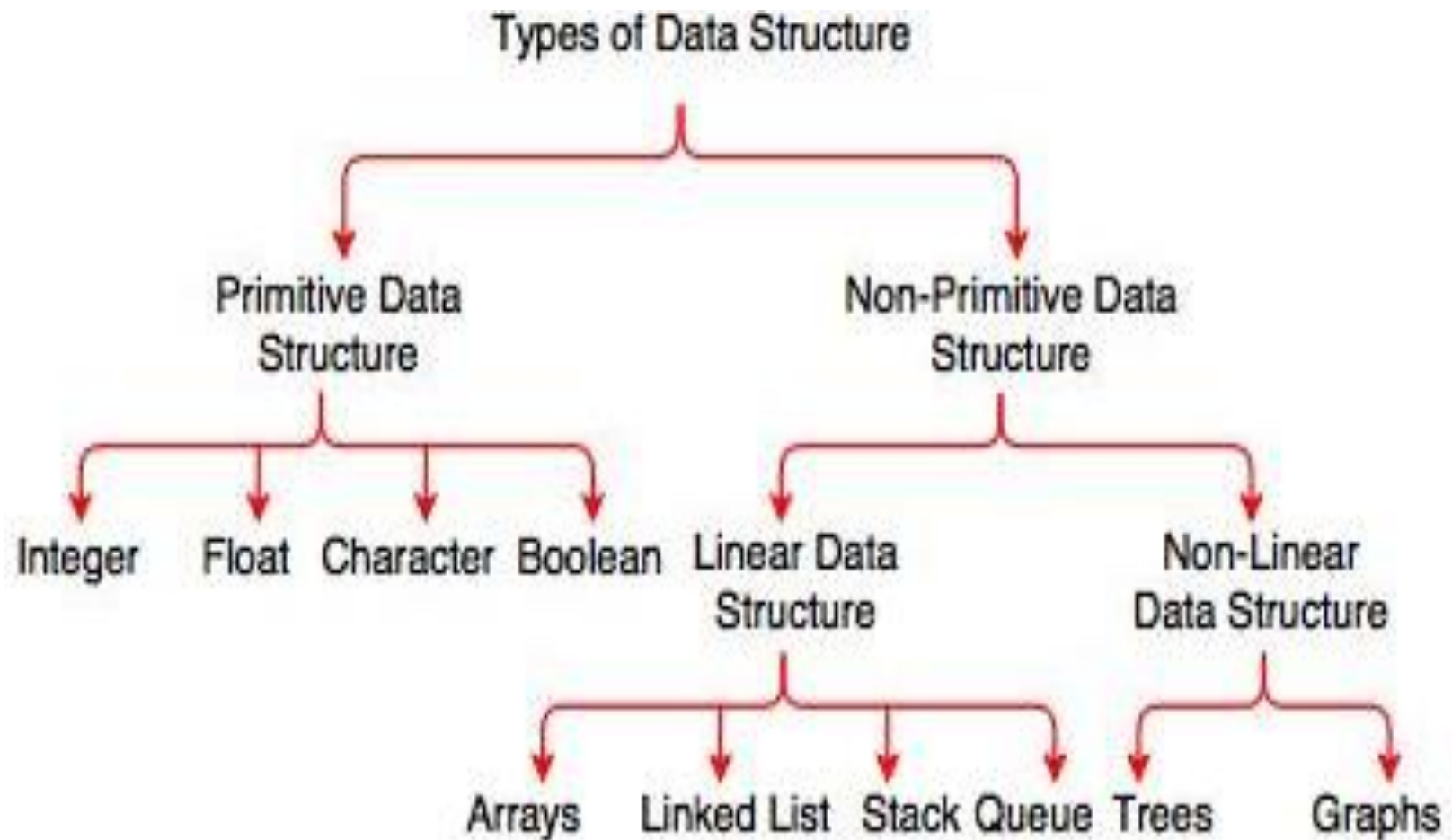
# Topics

- **Classification of Data Structures**
- Introduction to Stack
- Operations on Stacks
- Stack ADT
- Stack Implementation
- Applications of Stacks

# Classification of Data Structures



# Classification of Data Structures



# Primitive Vs Non-primitive Data Structures

- **Primitive Data Structures**

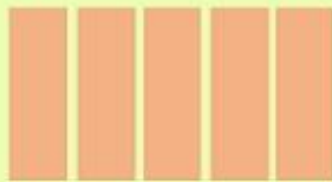
- Primitive Data Structures are the basic data structures that directly operate upon the machine instructions.
- We have different representations on different computers.
- Integers, Floating point numbers, Character constants, String constants and Pointers come under this category.

- **Non-primitive Data Structures**

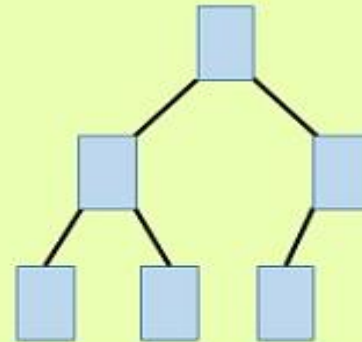
- Non-primitive data structures are derived from primitive data structures.
- They emphasize on grouping same or different data items with relationship between each data item.

# Linear Vs Non-Linear Data Structures

- In a linear data structure, data elements are arranged in a linear order where each and every elements are attached to its previous and next adjacent.
- In a non-linear data structure, data elements are attached in hierarchically manner.



**Linear Data  
Structure**



**Non -Linear Data  
Structure**

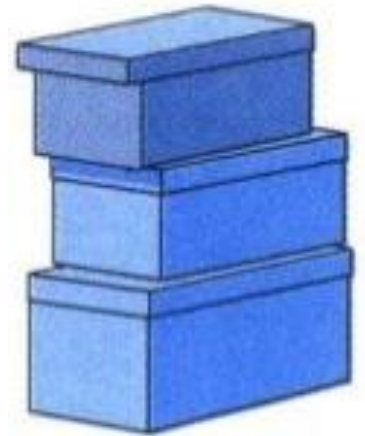
# Linear Vs Non-Linear Data Structures

Sr. No.	Key	Linear Data Structures	Non-linear Data Structures
1	Data Element Arrangement	In linear data structure, data elements are sequentially connected and each element is traversable through a single run.	In non-linear data structure, data elements are hierarchically connected and are present at various levels.
2	Levels	In linear data structure, all data elements are present at a single level.	In non-linear data structure, data elements are present at multiple levels.
3	Implementation complexity	Linear data structures are easier to implement.	Non-linear data structures are difficult to understand and implement as compared to linear data structures.
4	Traversal	Linear data structures can be traversed completely in a single run.	Non-linear data structures are not easy to traverse and needs multiple runs to be traversed completely.
5	Memory utilization	Linear data structures are not very memory friendly and are not utilizing memory efficiently.	Non-linear data structures uses memory very efficiently.
6	Time Complexity	Time complexity of linear data structure often increases with increase in size.	Time complexity of non-linear data structure often remain with increase in size.
7	Examples	Array, List, Queue, Stack.	Graph, Map, Tree.



# Topics

- Classification of Data Structures
- **Introduction to Stacks**
- Operations on Stacks
- Stack ADT
- Stack Implementation
- Applications of Stacks



# Stack Data Structure

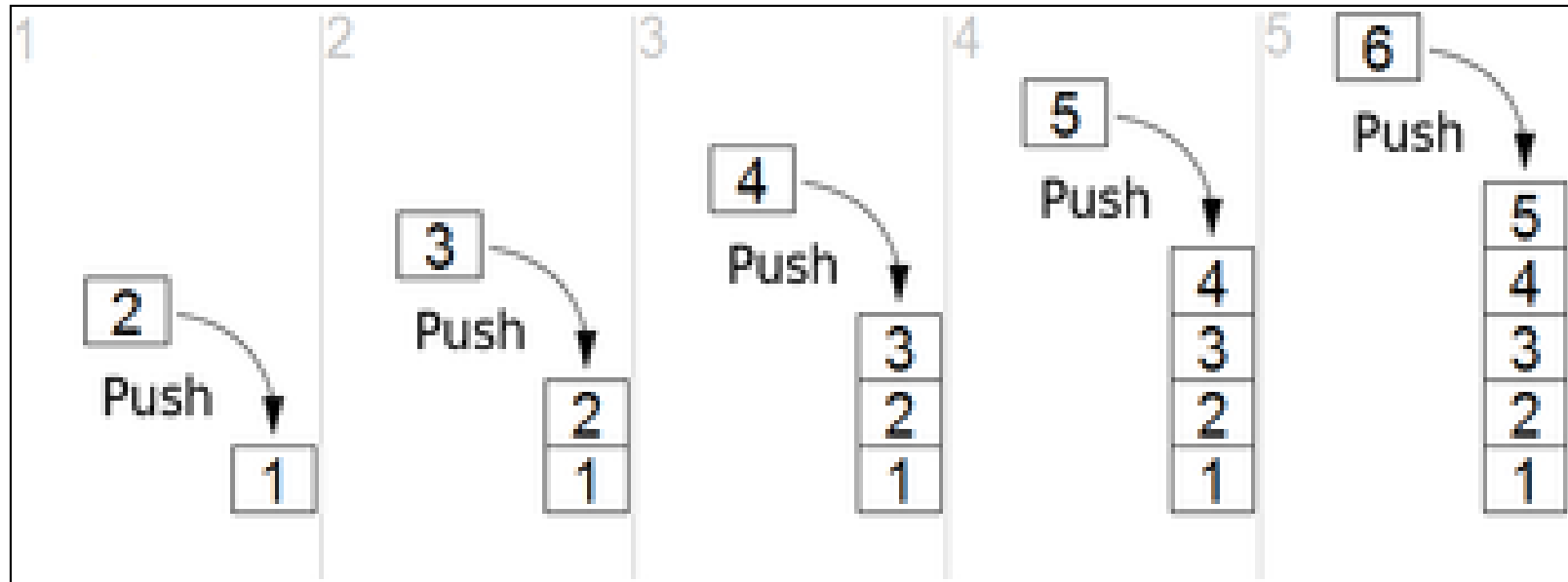
- A **stack** is an ordered collection of items.
- Stack is a linear data structure which follows a particular order in which the operations are performed.
- Addition of new items and the removal of existing items always takes place at the same end.
- The end is commonly referred to as the “**top**.”
- The end opposite the top is known as the “**base**.”
- The order may be **LIFO**(Last In First Out) or **FILO**(First In Last Out).

# Topics

- Classification of Data Structures
- Introduction to Stacks
- **Operations on Stacks**
- Stack ADT
- Stack Implementation
- Applications of Stacks

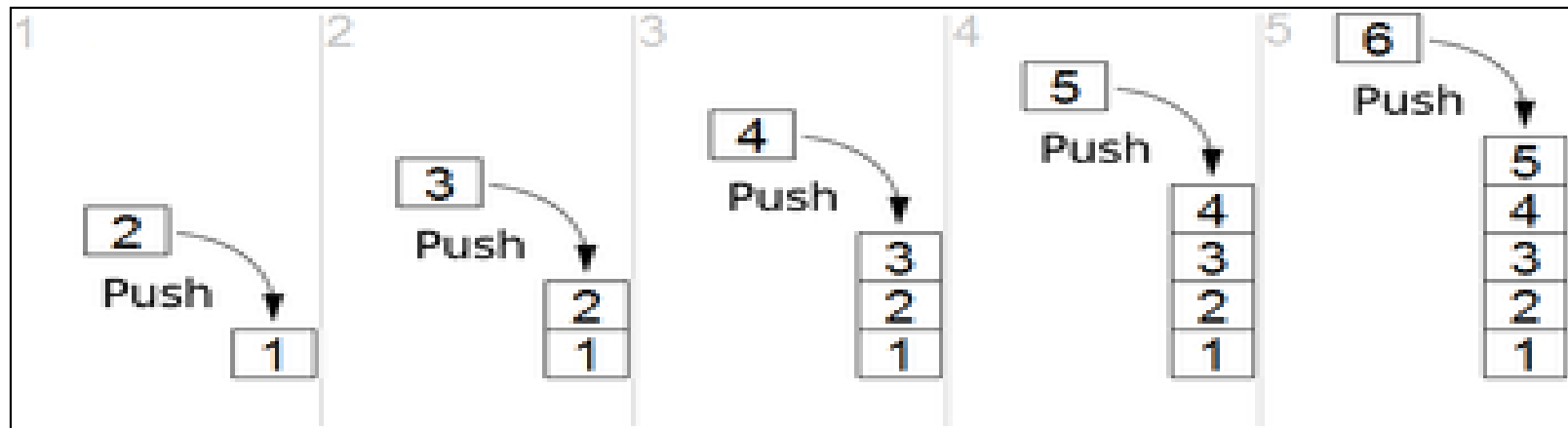
# Operations on Stack

- Push or Insertion or Addition**

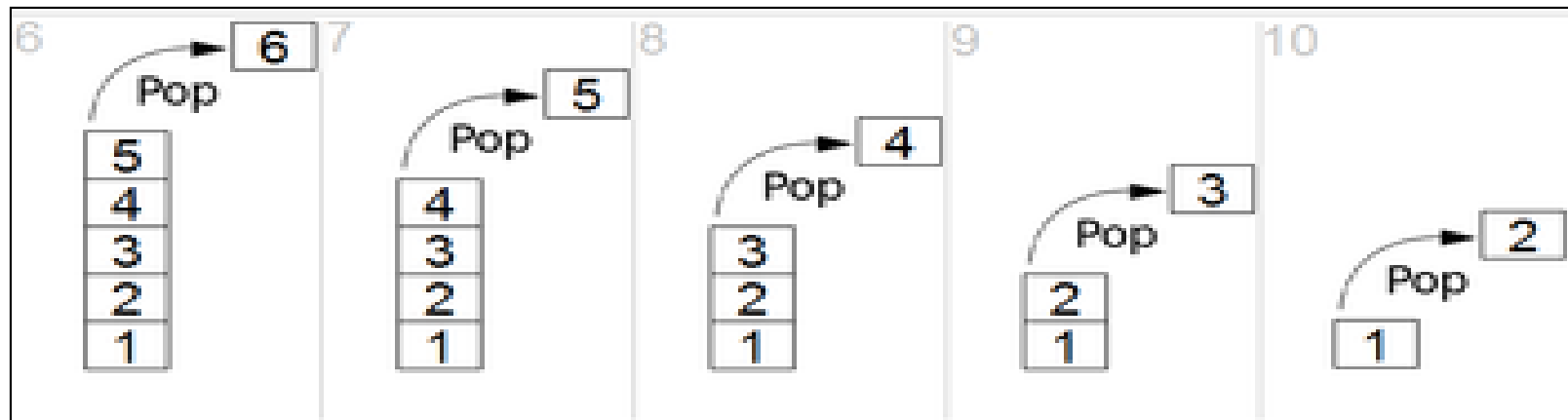


# Operations on Stack

- Push or Insertion or Addition**



- Pop or Deletion or Removal**



# Operations on Stacks

- **Is Empty**

- Check whether stack is empty or not

- **Is Full**

- Check whether stack is full or not

- **Top of the Stack**

- Display element which is in top position

# Topics

- Classification of Data Structures
- Introduction to Stacks
- Operations on Stacks
- **Stack ADT**
- Stack Implementation
- Applications of Stacks



# Stack ADT

ADT name
Objects
Functions

# Stack ADT

- **ADT Stack**

**Objects:** a finite ordered list with zero or more elements

**Functions:**

- **Create(max stack size):** creates a new stack of max stack size that is empty.
- **Push(stack, item):** adds a new item to the top of the stack.
- **Pop(Stack):** removes the top item from the stack.
- **Is Full(Stack):** tests to see whether the stack is full. It returns a Boolean value.
- **Is Empty(Stack):** tests to see whether the stack is empty. It returns a Boolean value.
- **Top of Stack(Stack):** returns the top item from the stack but does not remove it.

# Topics

- Classification of Data Structures
- Introduction to Stacks
- Operations on Stacks
- Stack ADT
- **Stack Implementation**
- Applications of Stacks

# Stacks Implementation using Arrays

- Define Max size with some value.
- Declare **an array Stack** of your preferred type with Max size.
- Declare a **variable TOS** of integer type and **initialize to -1**( because initially stack empty).

# PUSH Operation

- The process of placing a new data element onto stack is known as a Push Operation.
- Push operation involves a series of steps –

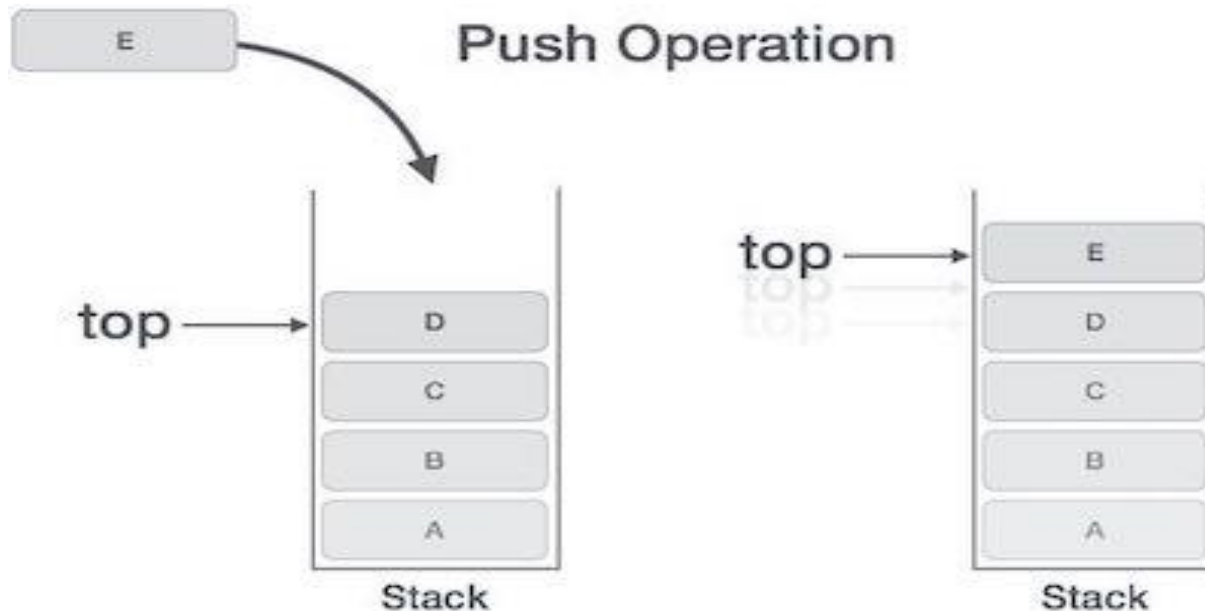
Step 1 – Check is the stack full.

Step 2 – If the stack is full, produces an error and exit.

Step 3 – If the stack is not full, increments top to point next empty space.

Step 4 – Adds data element to the stack location, where top is pointing.

Step 5 – end.



# PUSH Operation

```
Push(int Stack[], int item)
```

```
{
```

```
    if(Is Full(Stack))
```

```
        Printf("stack overflow")
```

```
    else
```

```
    {
```

```
        TOS++;
```

```
        stack[TOS]=item;
```

```
    }
```

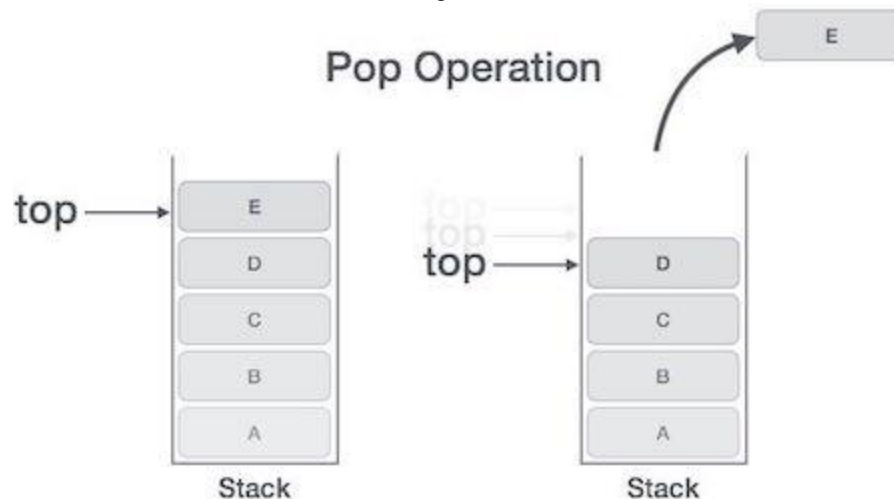
```
}
```

In this state we  
can't perform  
push operation

we can perform  
push operation

# POP Operation

- In an array implementation of pop() operation, the data element is not actually removed, instead top is decremented to a lower position in the stack to point to the next value.
- A Pop operation may involve the following steps:  
Step 1 – Check is the stack empty.  
Step 2 – If the stack is empty, produces an error and exit.  
Step 3 – If the stack is not empty, take the element at TOS  
Step 4 – Decreases the value of TOS by 1.  
Step 5 – end.



# POP Operation

```
Pop()
{
    if(Is Empty(Stack))
        Printf("stack is empty");
    else
    {
        a=stack[TOS];
        TOS--;
        Printf(" Popped out element is a ");
    }
}
```

**In this state we  
can't perform pop  
operation**



**Stack not empty  
we can perform  
pop operation**





# Is Full Operation

- Checks whether the stack is full or not.
- If TOS is equal to the max size of the stack then stack is full.

```
bool is Full(Stack)
{
    if( TOS == Max Size-1)
        return true;
    else
        return false;
}
```

# Is Empty Operation

- Checks whether the stack is empty or not.
- If TOS is equal to the initialized value then stack is empty.

```
bool is Empty(Stack)
{
    if( TOS == -1)
        return true;
    else
        return false;
}
```

# Top of the Stack Operation

- Displays the element which is at top position of the stack

```
Top of the Stack(Stack)
{
    if(Is Empty(Stack))
        Printf("stack is empty");
    else
    {
        Printf(" Top of the stack is TOS ");
    }
}
```

# Topics

- Classification of Data Structures
- Introduction to Stacks
- Operations on Stacks
- Stack ADT
- Stack Implementation
- **Applications of Stacks**

# Applications of Stacks

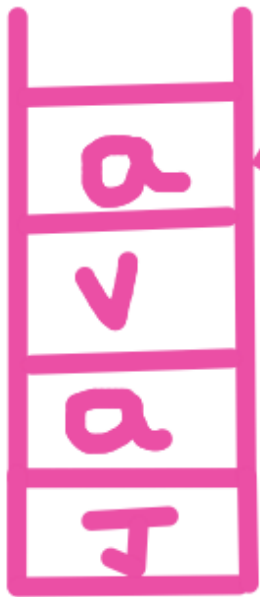
- Conversion of Expressions
- Evaluate Postfix expression
- Checking for Well-formed Parenthesis
- Reverse a string
- Simulate Recursion
- Process Subprogram function calls
- Decimal to binary Conversion
- In Backtracking algorithms (used in optimizations and games)

# Checking for Well-formed Parenthesis

BALANCED EXPRESSION	UNBALANCED EXPRESSION
$(a + b)$	$(a + b$
$[(c - d) * e]$	$[(c - d * e]$
$\{()\}[]$	$\{[(())]\}$

# Reversing a String

**Input String** : Java

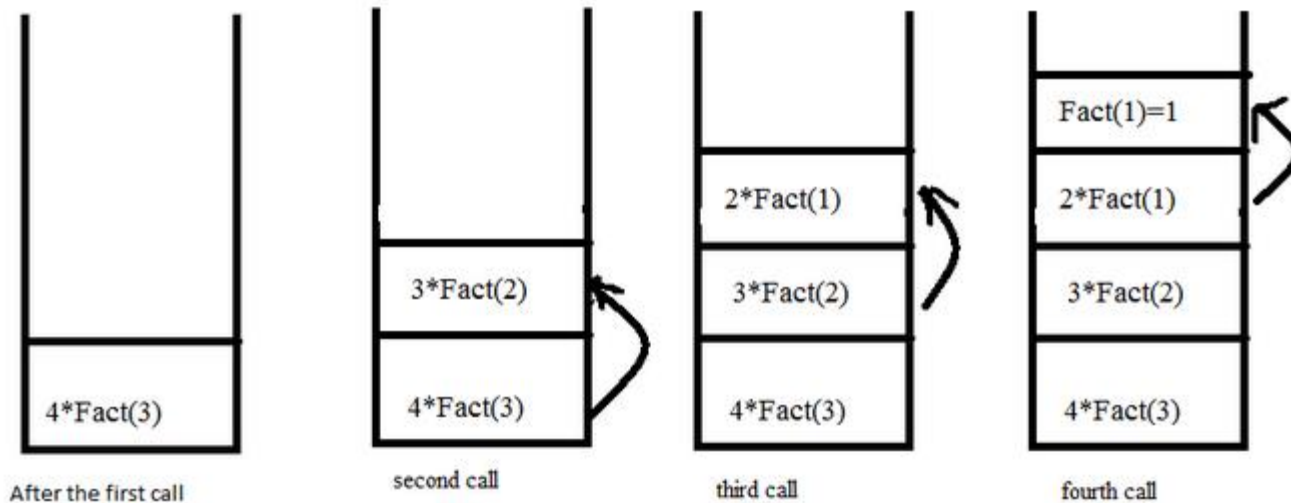


Stack after push  
operation

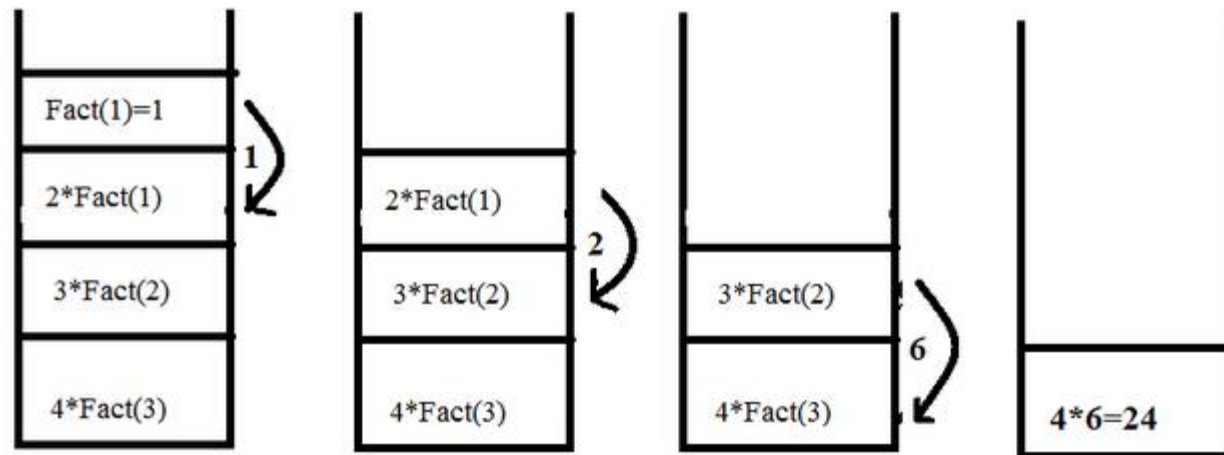
When we do pop operation  
the element push last is the first  
element to be removed

# Recursion using Stack for Factorial(4)

When function call happens previous variables gets stored in stack



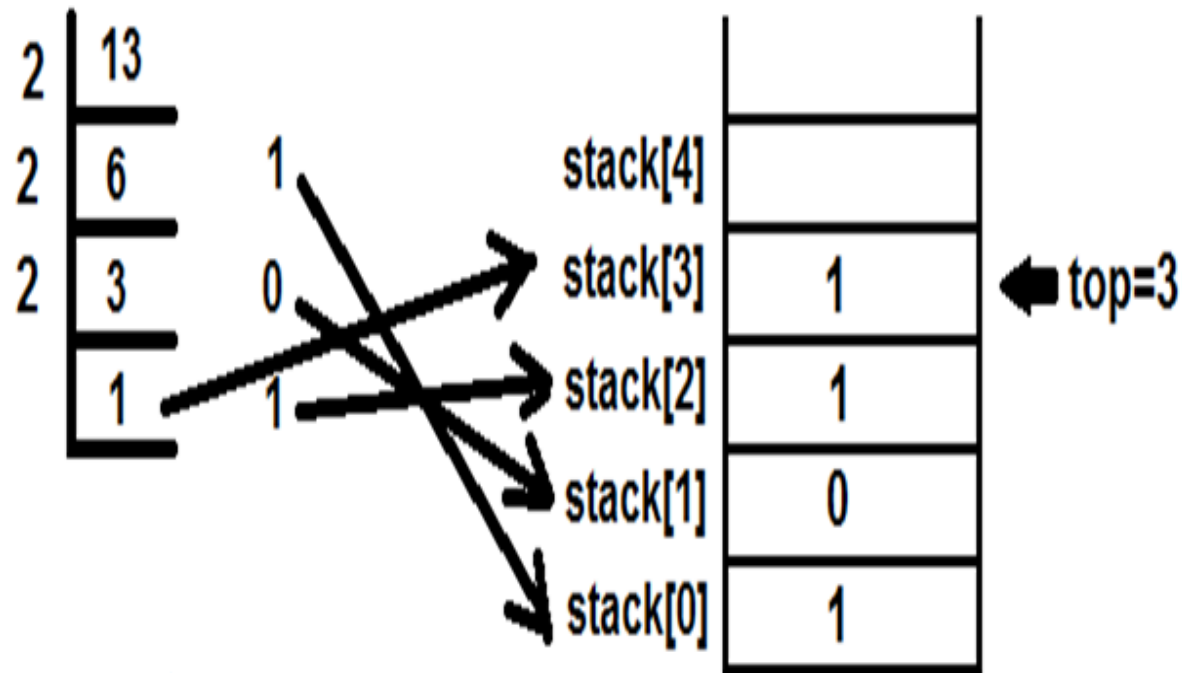
Returning values from base case to caller function





# Decimal to Binary Conversion

- $(13)_{10} = ?$



- $(13)_{10} = (1101)_2$

# Summary

- Primitive Vs Non-Primitive Data Structures
- Linear Vs Non-Linear Data Structures
- What is Stack
- Stack ADT
- Operation on Stack
- Applications of Stacks

# Applications of Stacks

- **Conversion of Expressions**
- Evaluate Postfix expression
- Checking whether the expression is balanced or not

# Expressions

- What is an expression?

An expression is a legal combination of operands and operators.

Ex:  $A+B*C$

$(4^5)*(10+6)$

- Operands means  
variable like x,y,z or constant like 5,4,0,2
- Operator means  
symbol like +,-,\*,/,^... used with operands.

# Expression Representation

- How to represent expressions
  - three ways.
- **INFIX:** Operator is between operands normal human readable
- **POSTFIX:** Operator is after operands
- **PREFIX:** Operator is before operands
- Both POSTFIX and PREFIX are used by computer systems to evaluate expressions and these does not have parenthesis.

# Examples

Infix	Postfix	Prefix
$A+B$	$AB+$	$+AB$
$(A+B)*C$	$AB+C^*$	$*+ABC$
$(A+B)*(C+D)$	$AB+CD+^*$	$*+AB+CD$

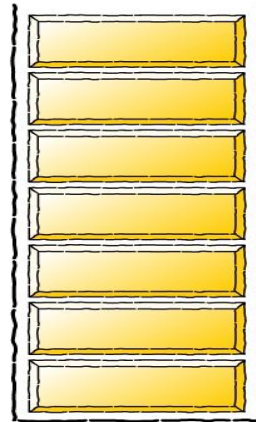
# Conversion of Expressions

- The given expression will be in one format and we need to convert it into another format.
- Users feel convenient to give input in infix notation
- But computer systems process them in postfix notation.
- So conversion required is:

Infix  $\rightarrow$  postfix

# Components needed

- Three components are required
  - Input vector
  - Stack
  - Output vector





# Operator Precedence

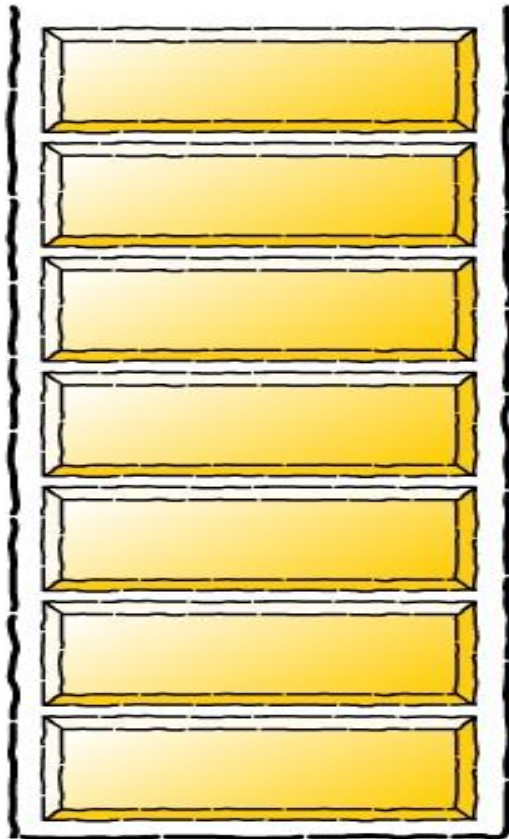
## **Precedence of operators:**

1.        **'^'**        Highest Precedence
2.    **'\*' and '/'**    **'%'**    High Precedence
3.    **'+' and '-'**    Low Precedence
4.        **'='**        Lower Precedence
5.    **'(' and ')'**    Lowest Precedence

# Rules to Convert Infix to Postfix Expression

- Read the given expression
- When the symbol is operand place it at the end of output
- When the token is not operand: it can be
  1. left parenthesis- (,{,[ → push it onto the stack
  2. right parenthesis- ),},] → Pop and display stack element until a left parenthesis is encountered and discard pair of parenthesis
- 3.If Stack is empty or incoming operator has high precedence than TOS :push it on to the Stack.
- 4.If incoming operator has low or equal precedence than TOS :Pop the stack.
- 5. A left parenthesis in the stack is assumed to have lower priority than that of operators.

# Infix to postfix conversion-Example 1



stack

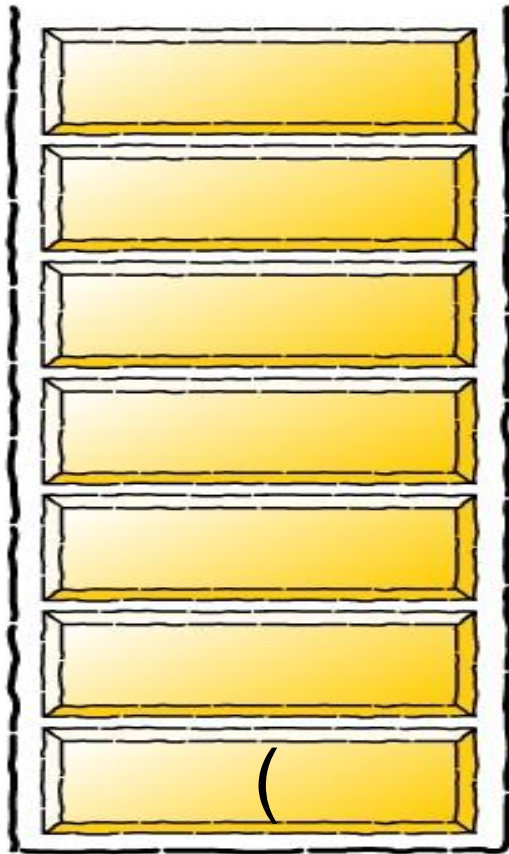
infixVect

$(a + b - c) * d - (e + f)$

postfixVect



# Infix to postfix conversion-Example 1

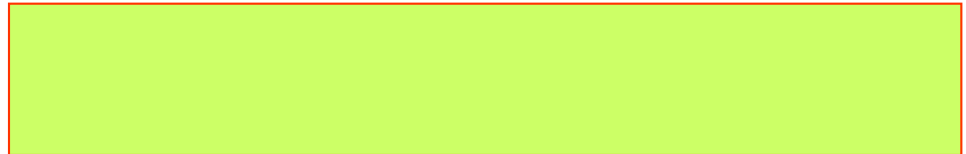


stack

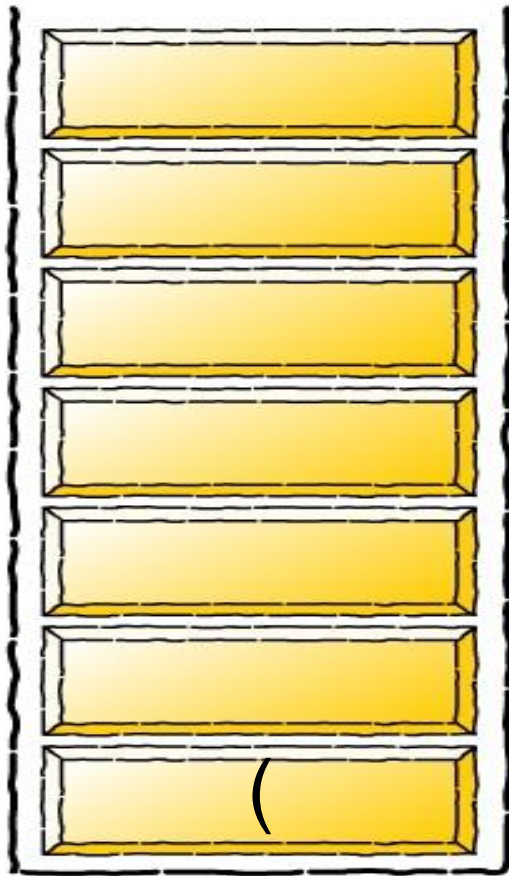
infixVect

$a + b - c ) * d - ( e + f )$

postfixVect



# Infix to postfix conversion-Example 1



stack

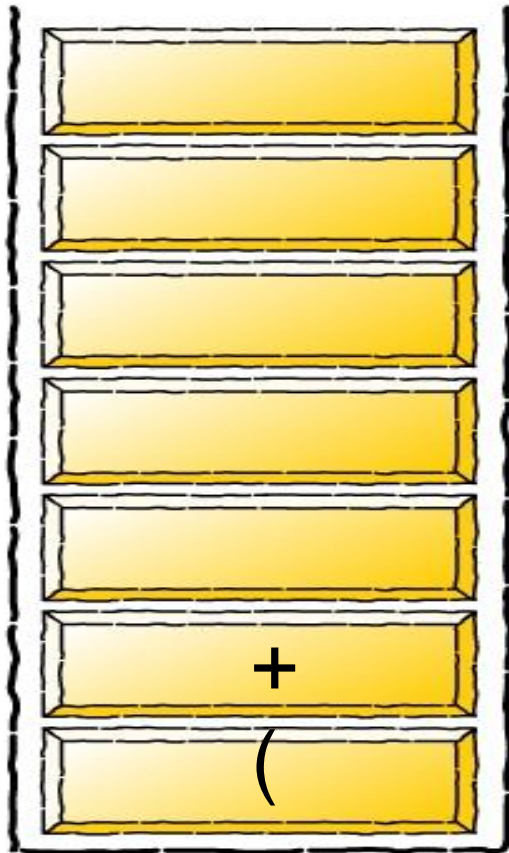
infixVect

$+ b - c ) * d - ( e + f )$

postfixVect

a

# Infix to postfix conversion-Example 1



stack

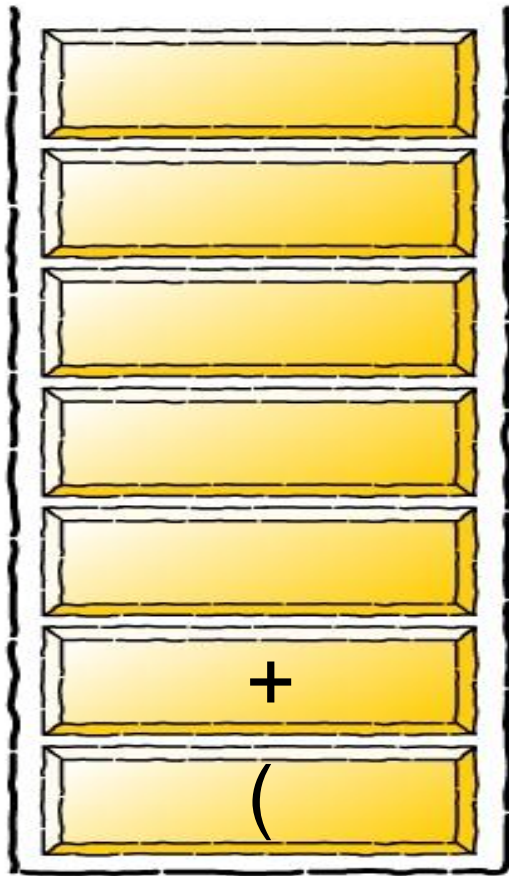
infixVect

$b - c ) * d - ( e + f )$

postfixVect

a

# Infix to postfix conversion-Example 1



stack

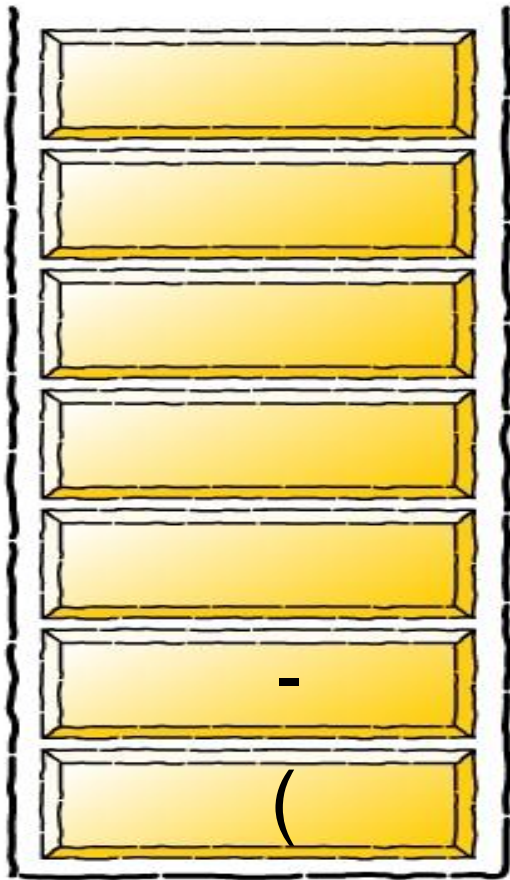
infixVect

- c ) \* d - ( e + f )

postfixVect

a b

# Infix to postfix conversion-Example 1



stack

infixVect

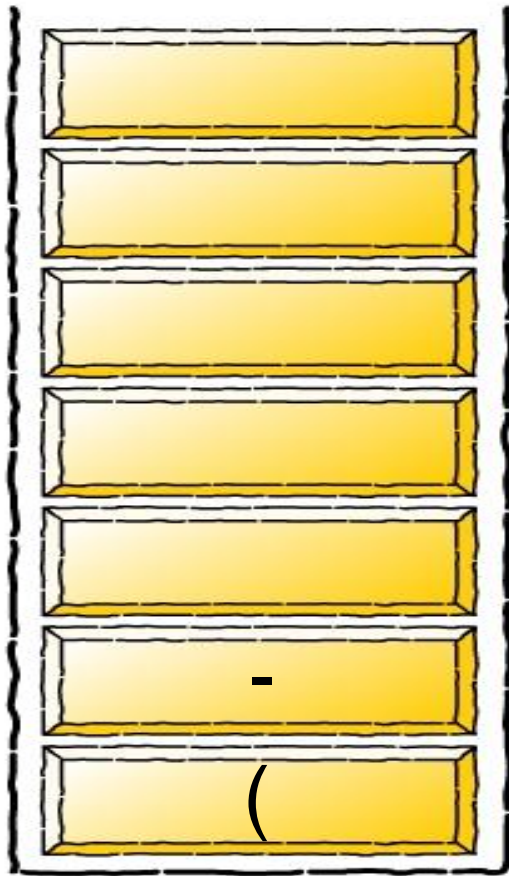
$c) * d - (e + f)$

postfixVect

$a b +$



# Infix to postfix conversion-Example 1



stack

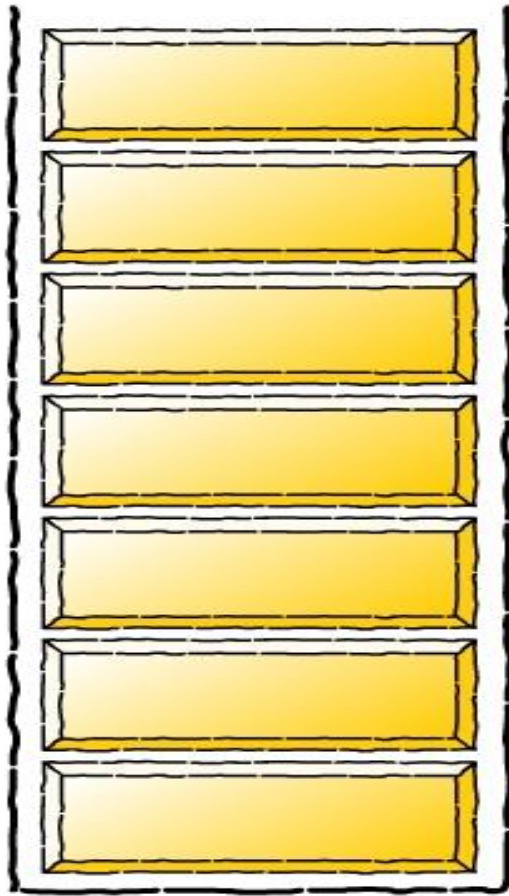
infixVect

) \* d - ( e + f )

postfixVect

a b + c

# Infix to postfix conversion-Example 1



stack

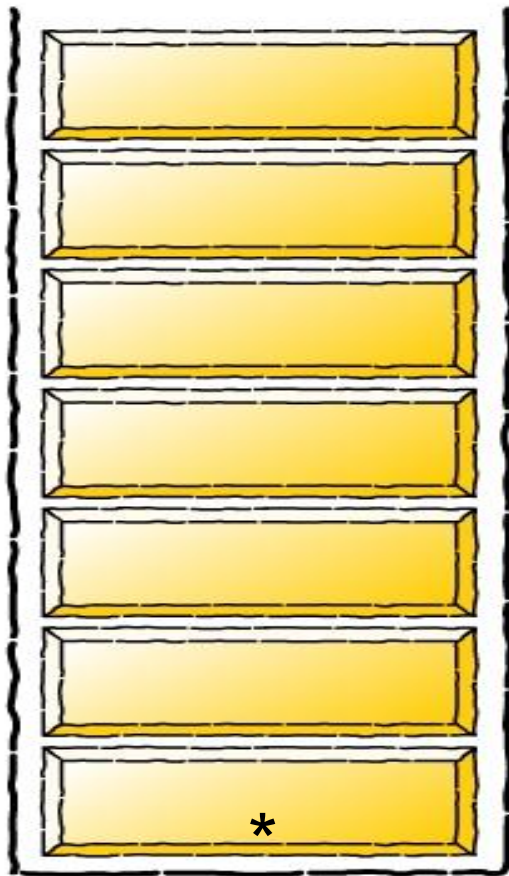
infixVect

$* d - ( e + f )$

postfixVect

$a b + c -$

# Infix to postfix conversion-Example 1



stack

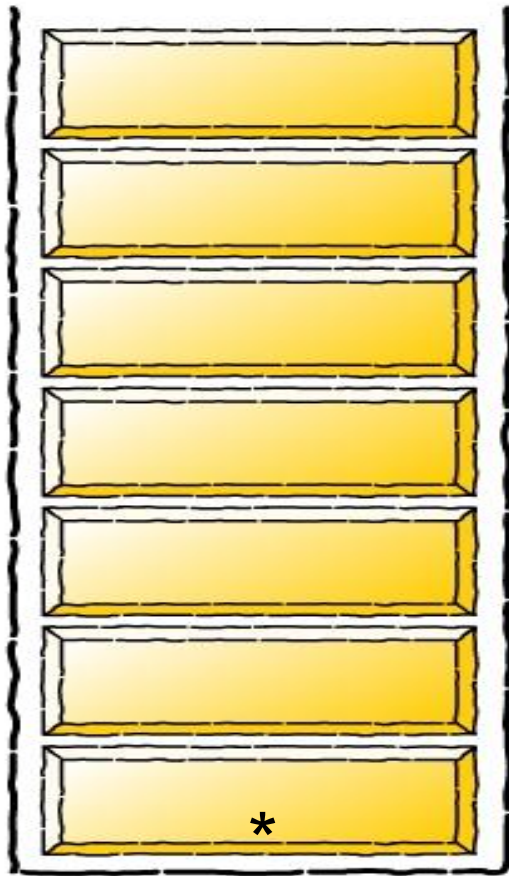
infixVect

$d - (e + f)$

postfixVect

$a b + c -$

# Infix to postfix conversion-Example 1



stack

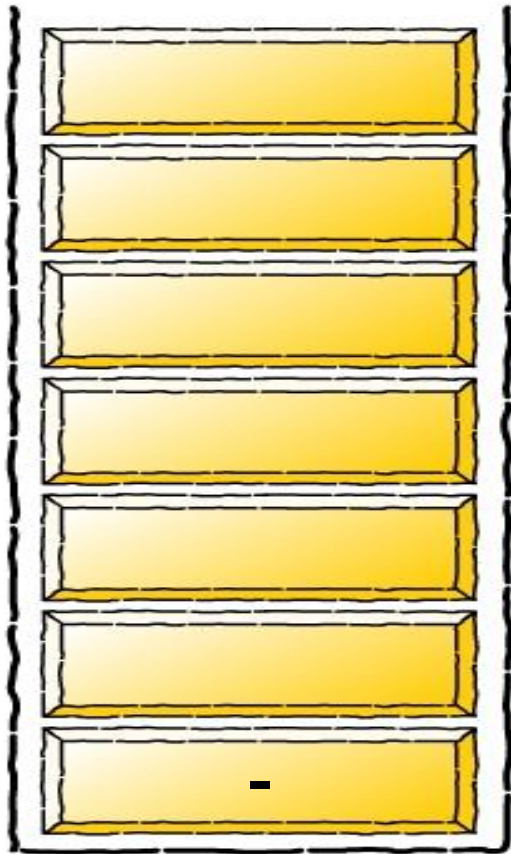
infixVect

$- ( e + f )$

postfixVect

$a b + c - d$

# Infix to postfix conversion-Example 1



stack

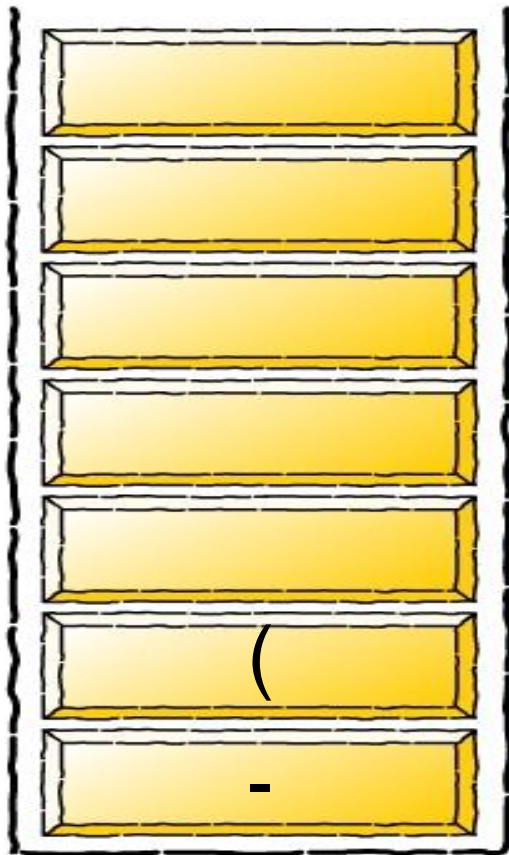
infixVect

( e + f )

postfixVect

a b + c - d \*

# Infix to postfix conversion-Example 1



stack

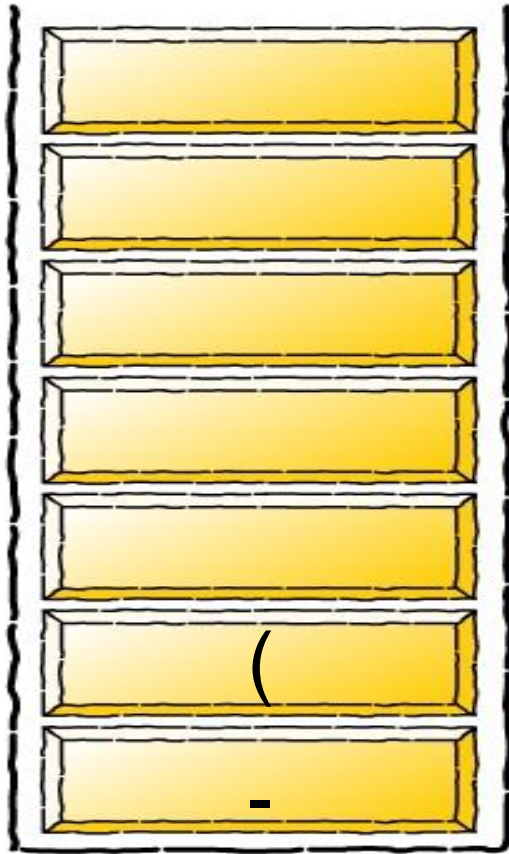
infixVect

e + f )

postfixVect

a b + c - d \*

# Infix to postfix conversion-Example 1



stack

infixVect

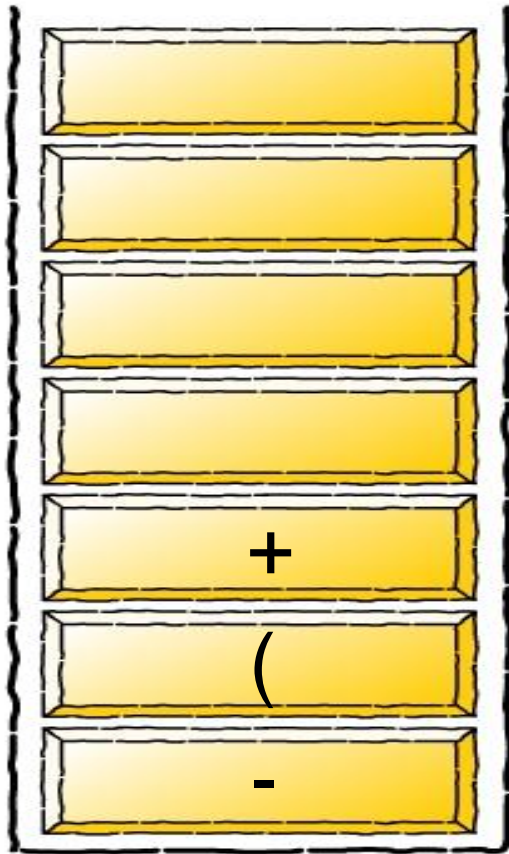
+ f )

postfixVect

a b + c - d \* e



# Infix to postfix conversion-Example 1



stack

infixVect

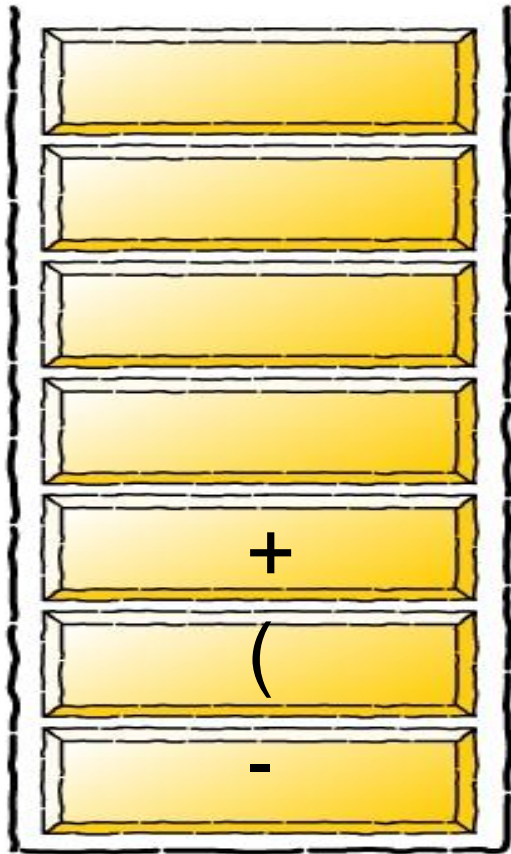
f )

postfixVect

a b + c - d \* e



# Infix to postfix conversion-Example 1



stack

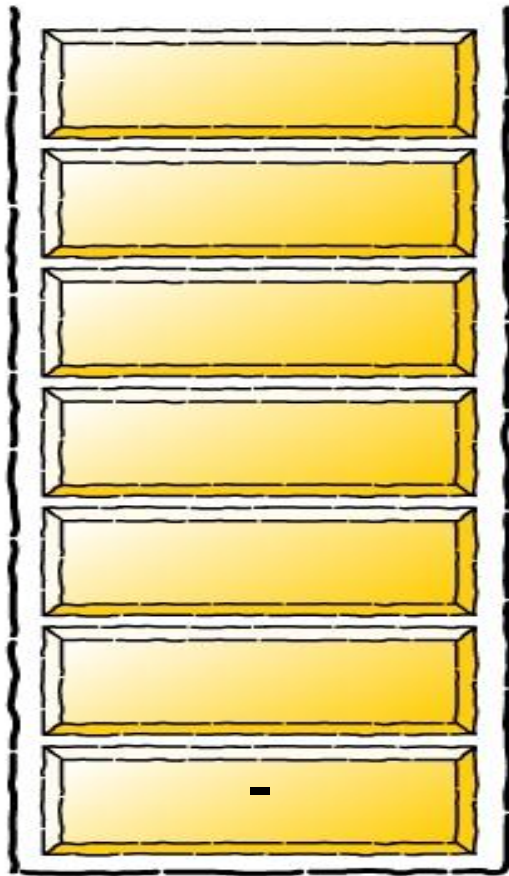
infixVect

)

postfixVect

a b + c - d \* e f

# Infix to postfix conversion-Example 1



stack

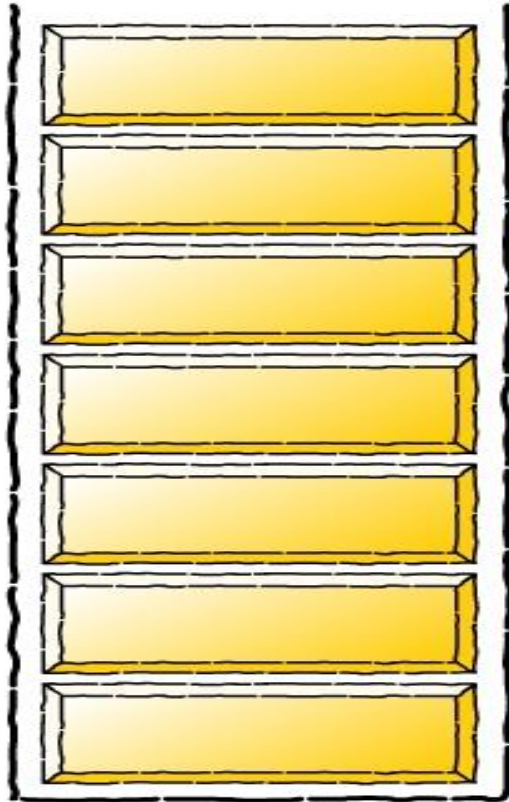
infixVect



postfixVect

a b + c - d \* e f +

# Infix to postfix conversion-Example 1



stack

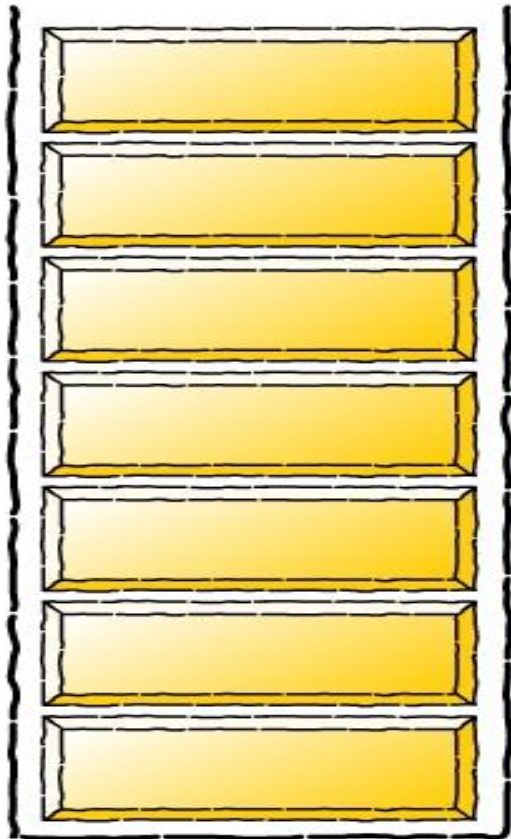
infixVect



postfixVect

a b + c - d \* e f + -

# Infix to postfix conversion-Example 2

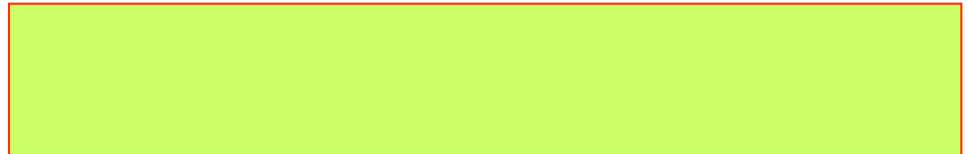


stack

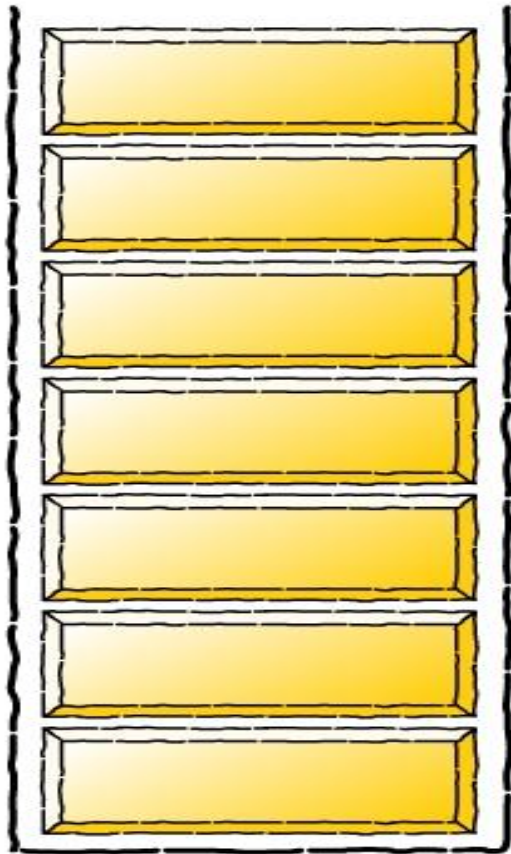
infixVect

$a + b * c + (d * e + f) * g$

postfixVect



# Infix to postfix conversion-Example 2



stack

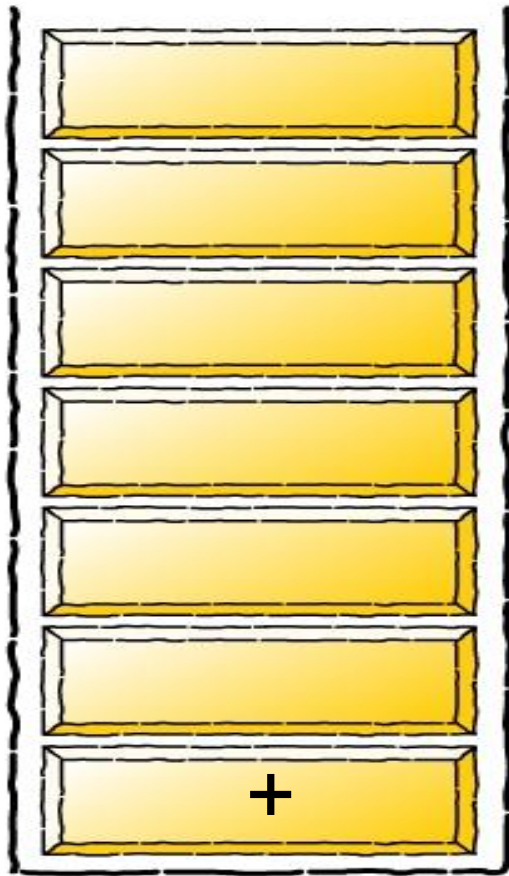
infixVect

$+ b * c + (d * e + f) * g$

postfixVect

a

# Infix to postfix conversion-Example 2



stack

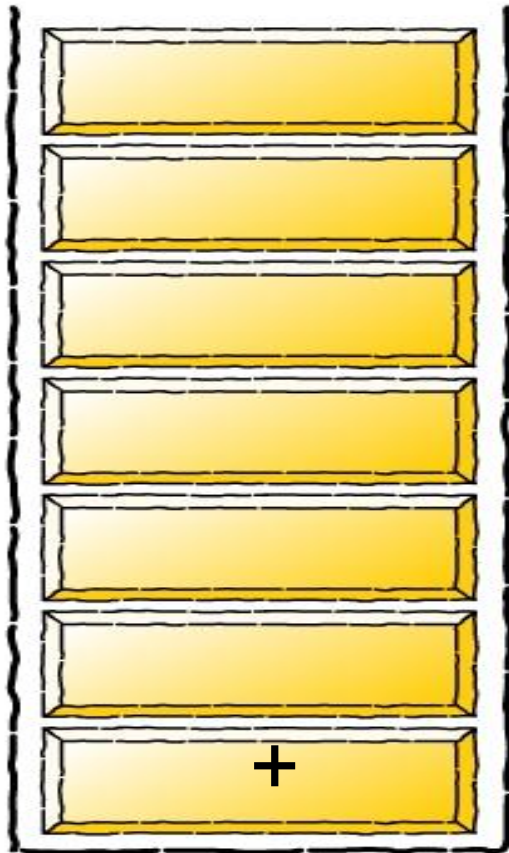
infixVect

$b * c + (d * e + f) * g$

postfixVect

a

# Infix to postfix conversion-Example 2



stack

infixVect

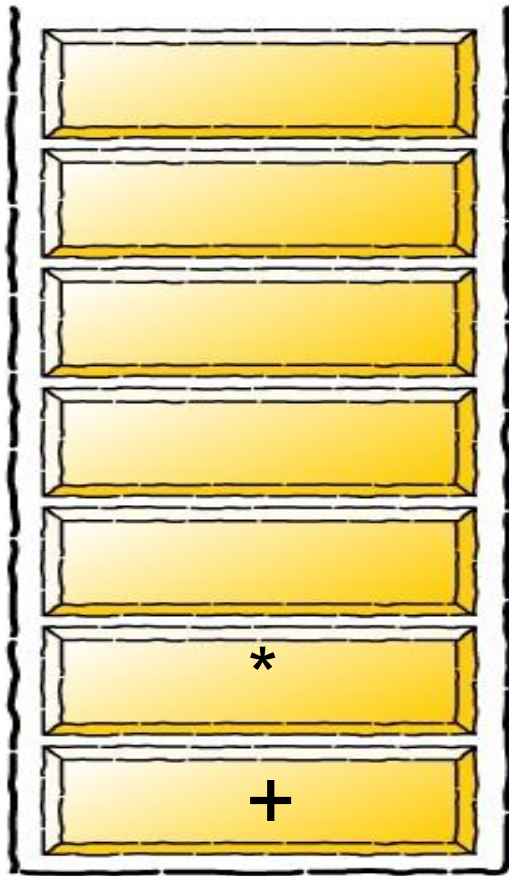
\* c +(d \* e + f ) \* g

postfixVect

a b



# Infix to postfix conversion-Example 2



stack

infixVect

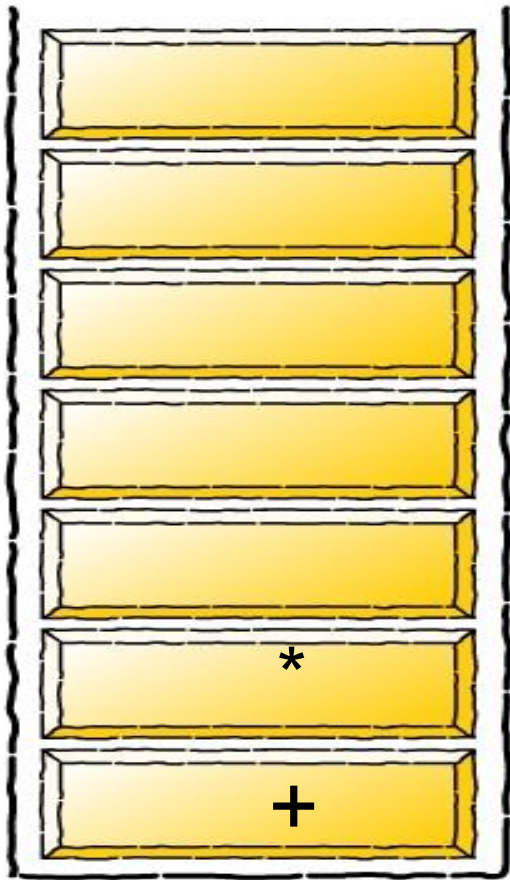
$c + (d * e + f) * g$

postfixVect

a b



# Infix to postfix conversion-Example 2



stack

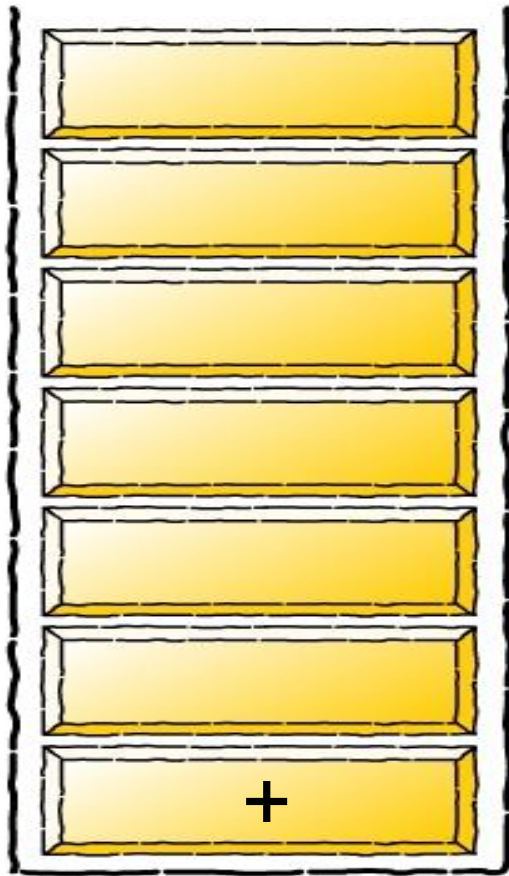
infixVect

$+(d * e + f) * g$

postfixVect

a b c

# Infix to postfix conversion-Example 2



stack

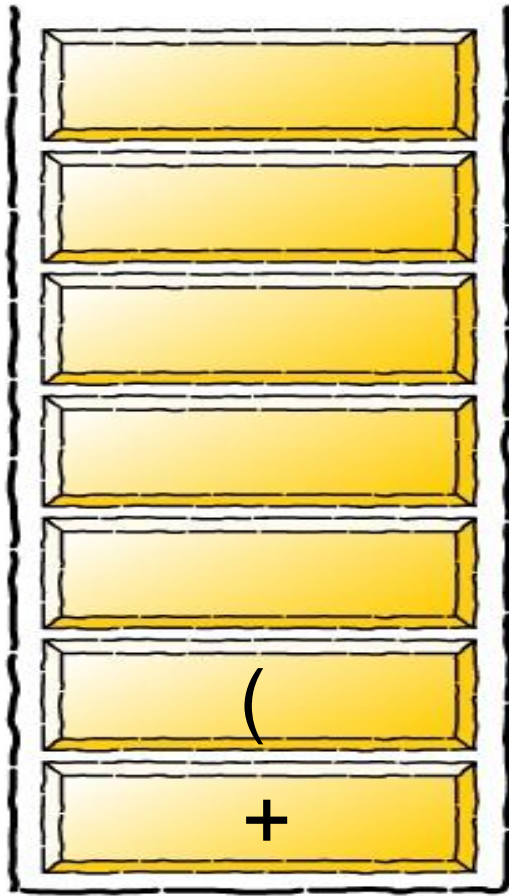
infixVect

$(d * e + f) * g$

postfixVect

$a b c * +$

# Infix to postfix conversion-Example 2



stack

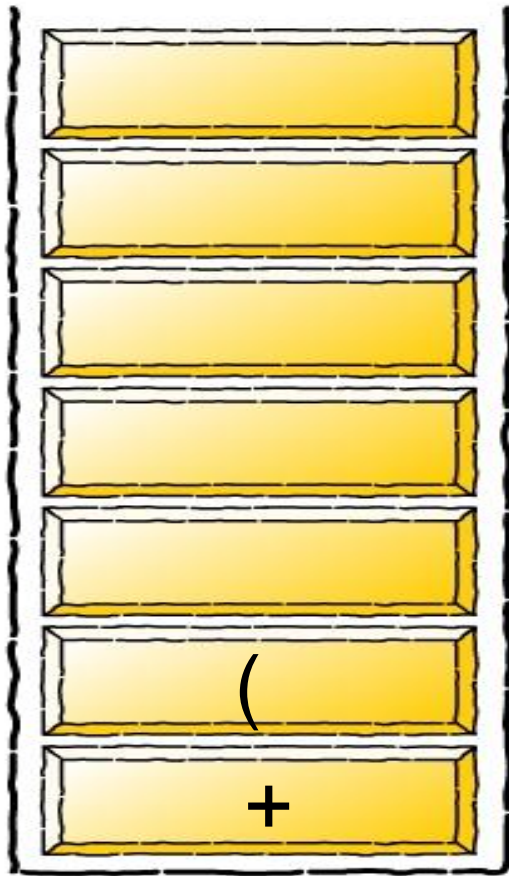
infixVect

$d * e + f ) * g$

postfixVect

$a b c * +$

# Infix to postfix conversion-Example 2



stack

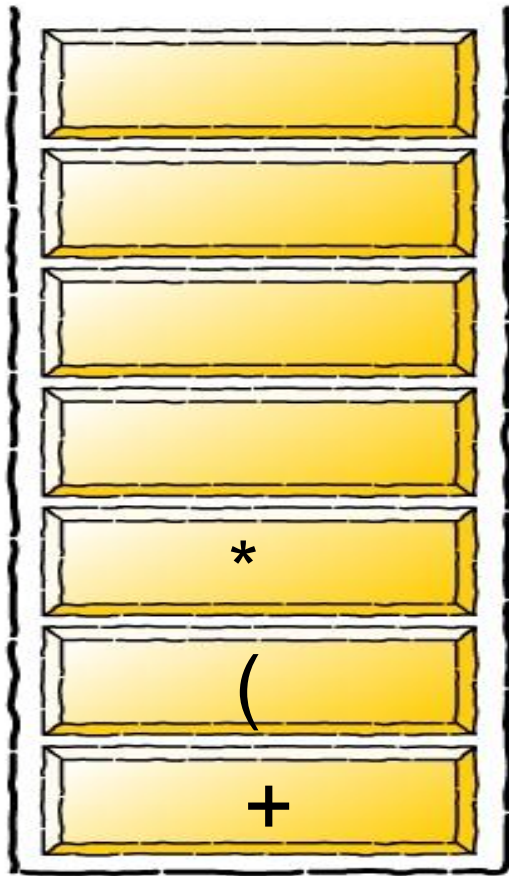
infixVect

\* e + f ) \* g

postfixVect

a b c \* + d

# Infix to postfix conversion-Example 2



stack

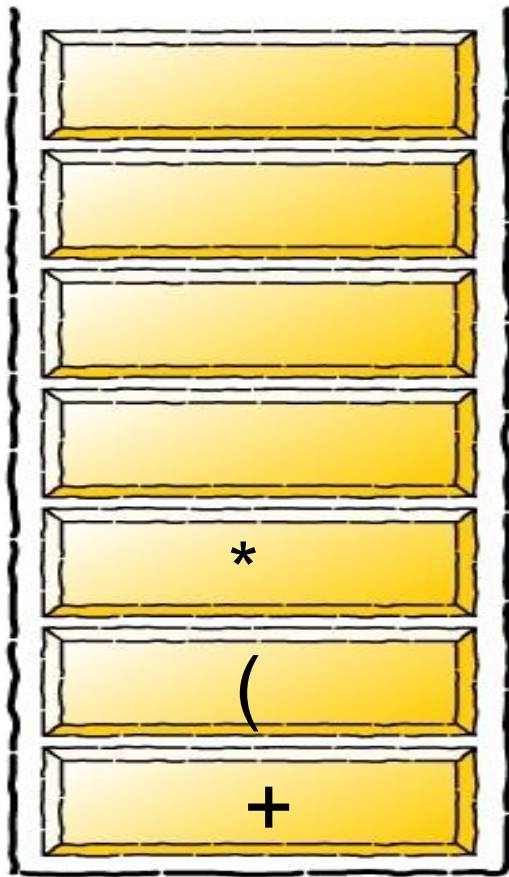
infixVect

$e + f ) * g$

postfixVect

$a b c * + d$

# Infix to postfix conversion-Example 2



stack

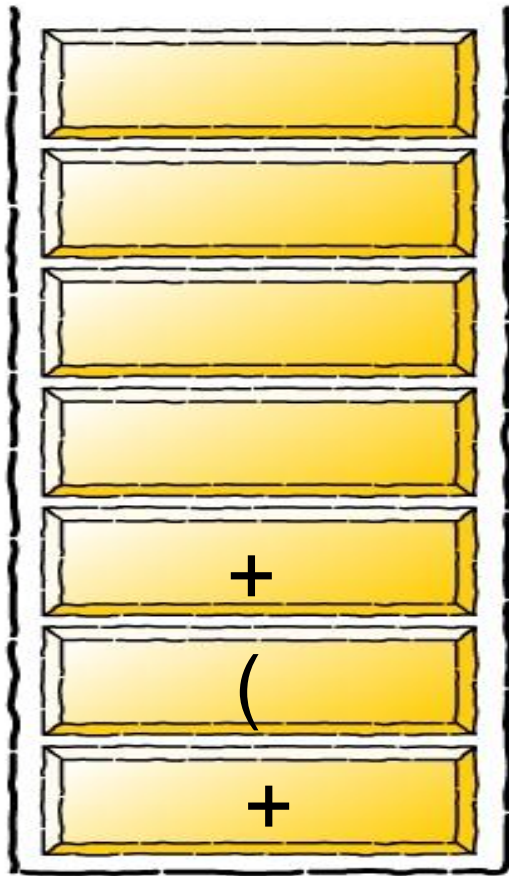
infixVect

+ f ) \* g

postfixVect

a b c \* + d e

# Infix to postfix conversion-Example 2



stack

infixVect

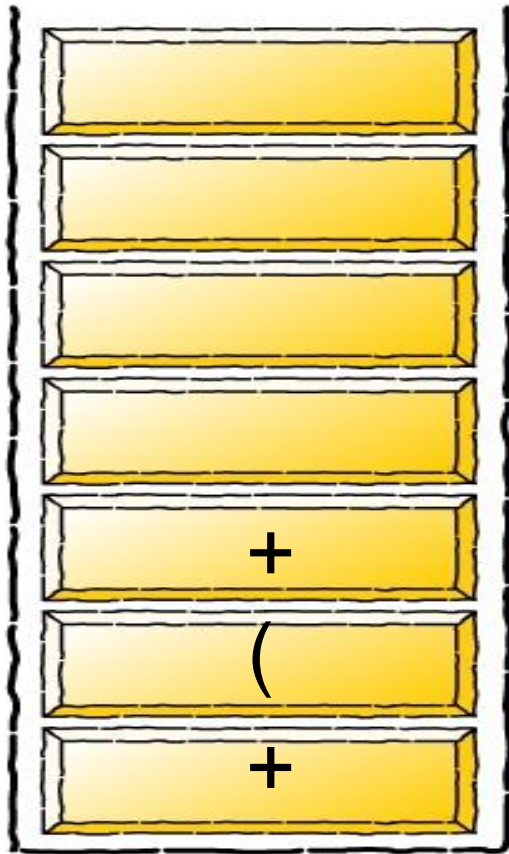
f ) \* g

postfixVect

a b c \* + d e \*



# Infix to postfix conversion-Example 2



stack

infixVect

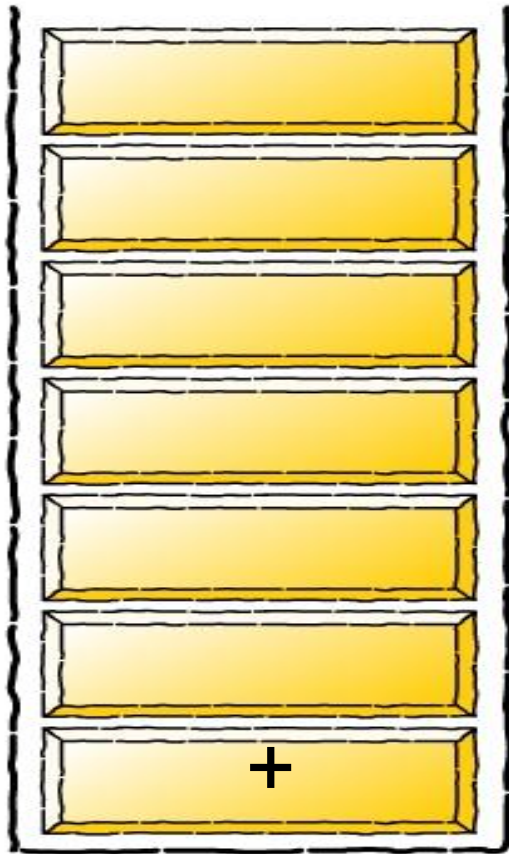
) \* g

postfixVect

a b c \* + d e \* f



# Infix to postfix conversion-Example 2



stack

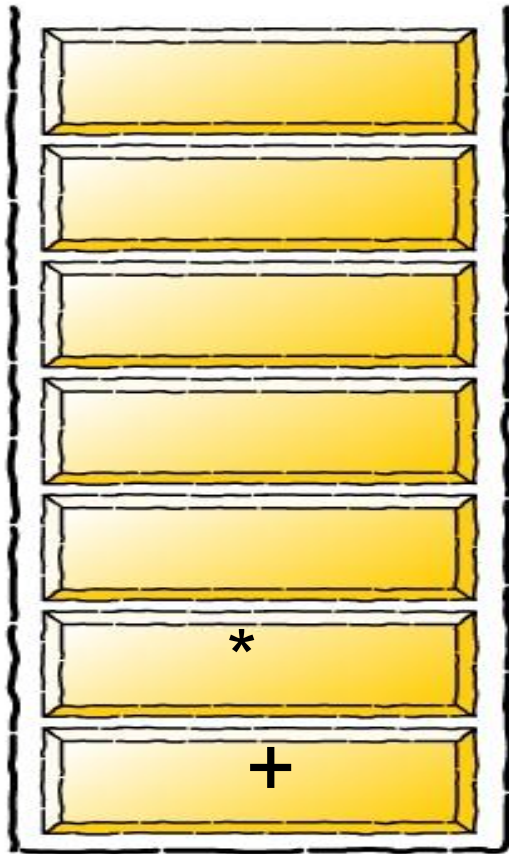
infixVect

\*g

postfixVect

a b c \* + d e \* f +

# Infix to postfix conversion-Example 2



stack

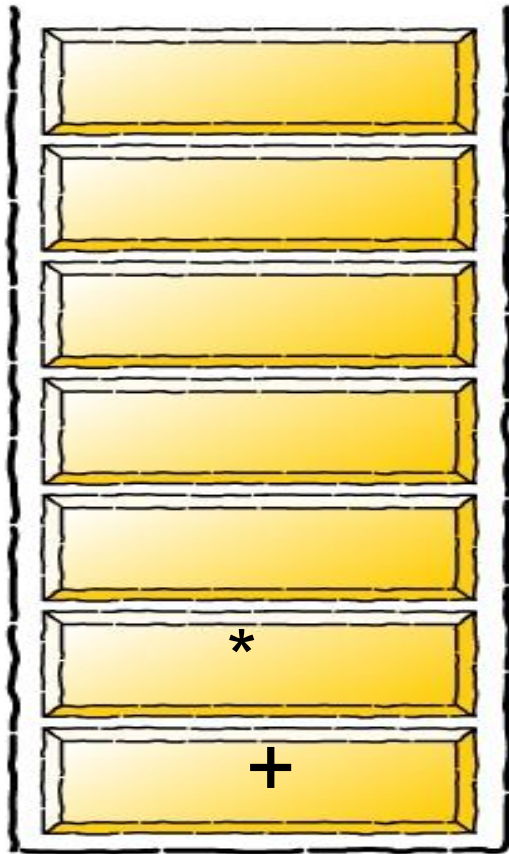
infixVect

g

postfixVect

a b c \* + d e \* f +

# Infix to postfix conversion-Example 2



stack

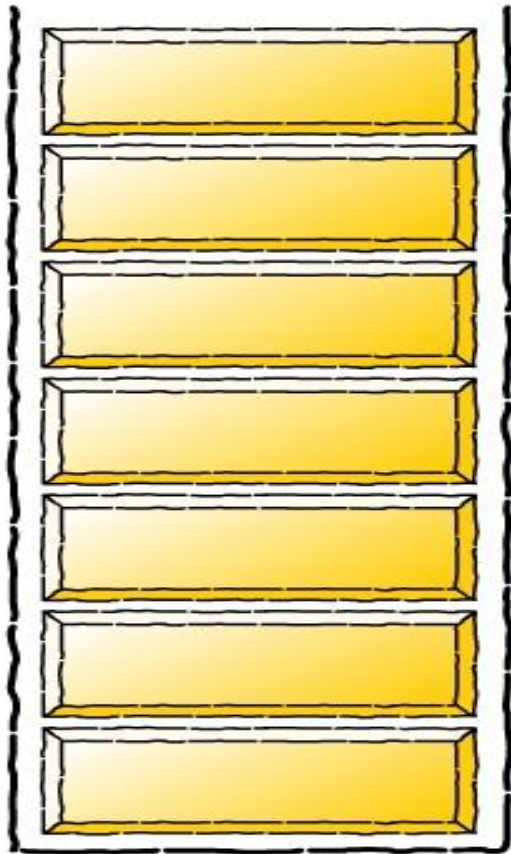
infixVect



postfixVect

a b c \* + d e \* f + g

# Infix to postfix conversion-Example 2



stack

infixVect



postfixVect

a b c \* + d e \* f + g \* +

# Infix to postfix conversion-Example 3

Infix Expression:  $(A + (B * C - (D / E ^ F) * G) * H)$  where  $^$  is an exponential operator.

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

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

# Process of Infix to postfix

**Step 1:** Repeat until each character in the infix notation is scanned

**IF** a "(" is encountered, push it on the stack

**IF** an operand (either a digit or a character) is encountered, add it postfix expression.

**IF** a ")" is encountered, then

a. Repeatedly pop from stack and add it to the postfix expression until "(" is encountered.

b. Discard the "(" . That is, remove the "(" from stack and do not add it to the postfix expression

**IF** an operator is encountered, then

a. if the current operator has equal or low precedence than TOS, then Repeatedly pop from stack and add each operator (popped from the stack) to the postfix expression

b. otherwise Push the operator to the stack

**Step 2:** Repeatedly pop from the stack and add it to the postfix expression until the stack is empty

# Algorithm for Infix to postfix conversion

## Algorithm infix-to-postfix(infix)

```
{
    i=0,k=0;
    while( (ch=infix[i]) != '\0')
    {
        if( ch == '(' )    push(ch);
        else if(isalnum(ch))    postfix[k++]=ch;    /* isalnum() function from ctype.h library */
        else if( ch == ')')
        {
            while( (c=pop()) != '(')
                postfix[k++]=c;
        }
        else
        {
            /* in case of Operator */
            while( precedence(stack[top]) >= precedence(ch) )
                postfix[k++]=pop();
            push(ch);
        }
        i++;
    }
    while( top != -1)    /* Pop from stack till empty */
        postfix[k++]=pop();
    Write Postfix[];
}
```

# Applications of Stacks

- Conversion of Expressions
- **Evaluate Postfix expression**
- Checking for Balanced Expression



# How to evaluate postfix (manually)

Going from left to right, if you see an operator, apply it to the previous two operands (numbers)

Example:

$$\begin{array}{ccccccccccc} A & B & C & * & D & / & + & E & F & - & - \\ & \underbrace{\hspace{1.5cm}} & & & & & & \underbrace{\hspace{1.5cm}} & & & \\ & (B * C) & & & & & & (E - F) & & & \\ & \underbrace{\hspace{2.5cm}} & & & & & & & & & \\ & ((B * C) / D) & & & & & & & & & \\ & \underbrace{\hspace{3.5cm}} & & & & & & & & & \\ & (A + (B * C) / D) & & & & & & & & & \\ & \underbrace{\hspace{4.5cm}} & & & & & & & & & \\ & ((A + (B * C) / D) - (E - F)) & & & & & & & & & \end{array}$$

Equivalent infix:  $A + B * C / D - (E - F)$


# Evaluating Postfix Expressions using Stack

- Easy to do with a stack
- given a proper postfix expression:
  - get the next token
  - if it is an operand push it onto the stack
  - else if it is an operator
    - pop the stack for the right hand operand
    - pop the stack for the left hand operand
    - apply the operator to the two operands
    - push the result onto the stack
  - when the expression has been exhausted the result is the top (and only element) of the stack

# Evaluating Postfix Expressions-Example 1

Postfix Expression    **5 3 + 8 2 - \***

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

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

# Evaluating Postfix Expressions-Example 1

Postfix Expression

**5 3 + 8 2 - \***

5

push(5)



Nothing

# Evaluating Postfix Expressions-Example 1

Postfix Expression

**5 3 + 8 2 - \***

3

push(3)



Nothing

# Evaluating Postfix Expressions-Example 1

Postfix Expression

**5 3 + 8 2 - \***

+

```
value1 = pop()
value2 = pop()
result = value2 + value1
push(result)
```



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

**(5 + 3)**

# Evaluating Postfix Expressions-Example 1

Postfix Expression

**5 3 + 8 2 - \***

8

push(8)



(5 + 3)

# Evaluating Postfix Expressions-Example 1

Postfix Expression

**5 3 + 8 2 - \***

2

push(2)



(5 + 3)



# Evaluating Postfix Expressions-Example 1

Postfix Expression

**5 3 + 8 2 - \***

-

```
value1 = pop()
value2 = pop()
result = value2 - value1
push(result)
```



```
value1 = pop(); // 2
value2 = pop(); // 8
result = 8 - 2; // 6
Push( 6 )
```

**(8 - 2)**

**(5 + 3) , (8 - 2)**

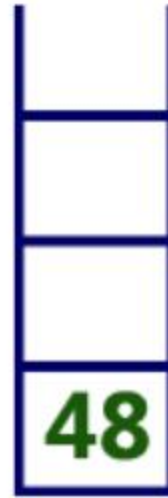
# Evaluating Postfix Expressions-Example 1

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)**

# Evaluating Postfix Expressions-Example 1

Postfix Expression

**5 3 + 8 2 - \***

\$

End of Expression

result = pop()



Display (result)

**48**

As final result

# Evaluating Postfix Expressions-Example 2

A digital display interface for evaluating postfix expressions. The top section features a horizontal row of ten light blue boxes, each containing a character of the postfix expression: 4, 3, \*, 6, 7, +, 5, -, +, and ). Below this row is a large, empty rectangular area with a dark blue background, intended for showing the step-by-step evaluation process.

4	3	*	6	7	+	5	-	+	)
---	---	---	---	---	---	---	---	---	---

# Evaluating Postfix Expressions-Example 2

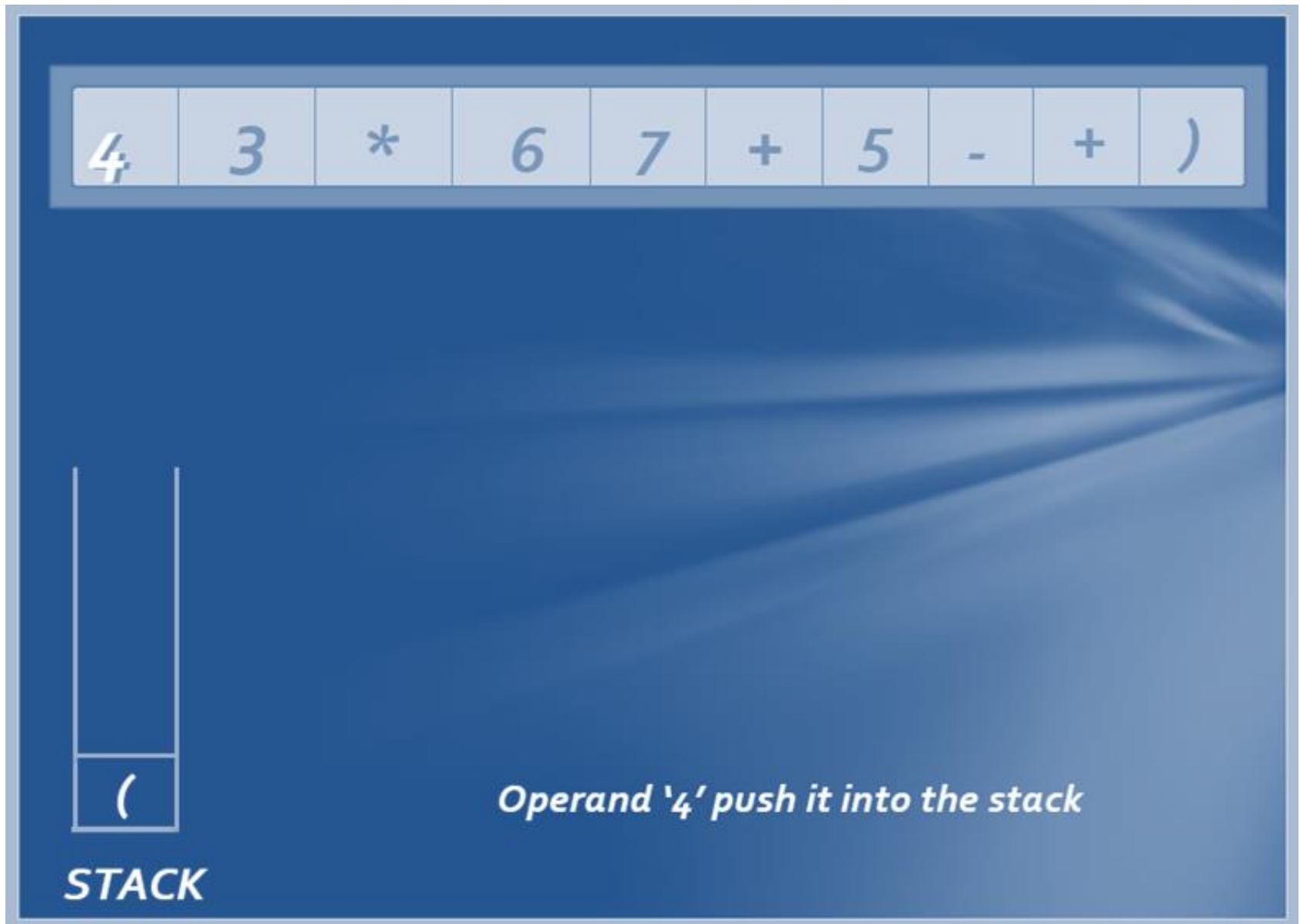
The diagram illustrates the initial state of a postfix expression evaluator. At the top, a horizontal sequence of tokens is shown: 4, 3, \*, 6, 7, +, 5, -, +, ). Below this, on the left, is a vertical rectangle representing a stack, which is currently empty. The word "STACK" is written below the rectangle. To the right of the stack, the instruction "Push the opening bracket '(' into the stack ." is displayed.

4	3	*	6	7	+	5	-	+	)
---	---	---	---	---	---	---	---	---	---

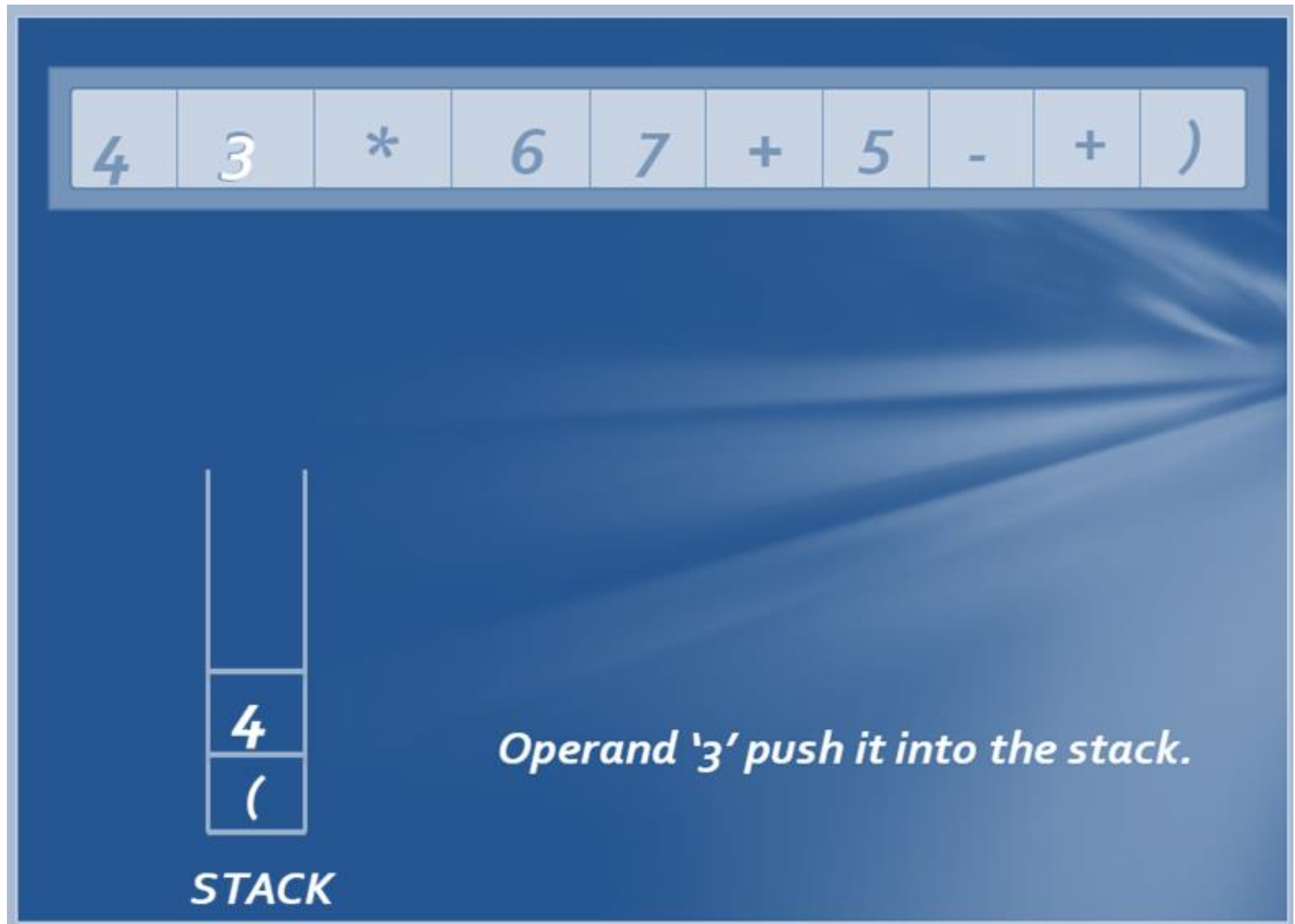
STACK

*Push the opening bracket '(' into the stack .*

# Evaluating Postfix Expressions-Example 2



# Evaluating Postfix Expressions-Example 2

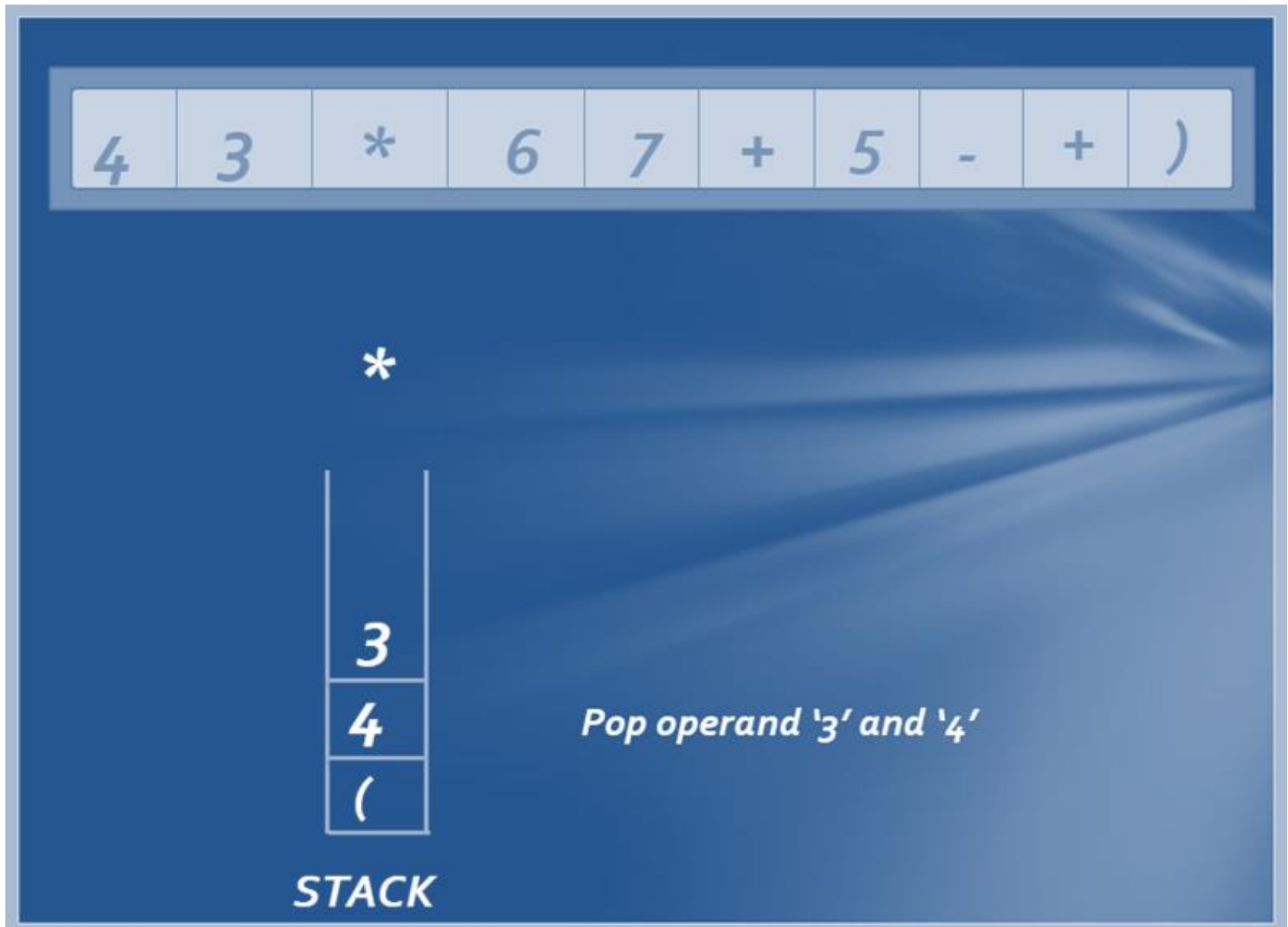


# Evaluating Postfix Expressions-Example 2

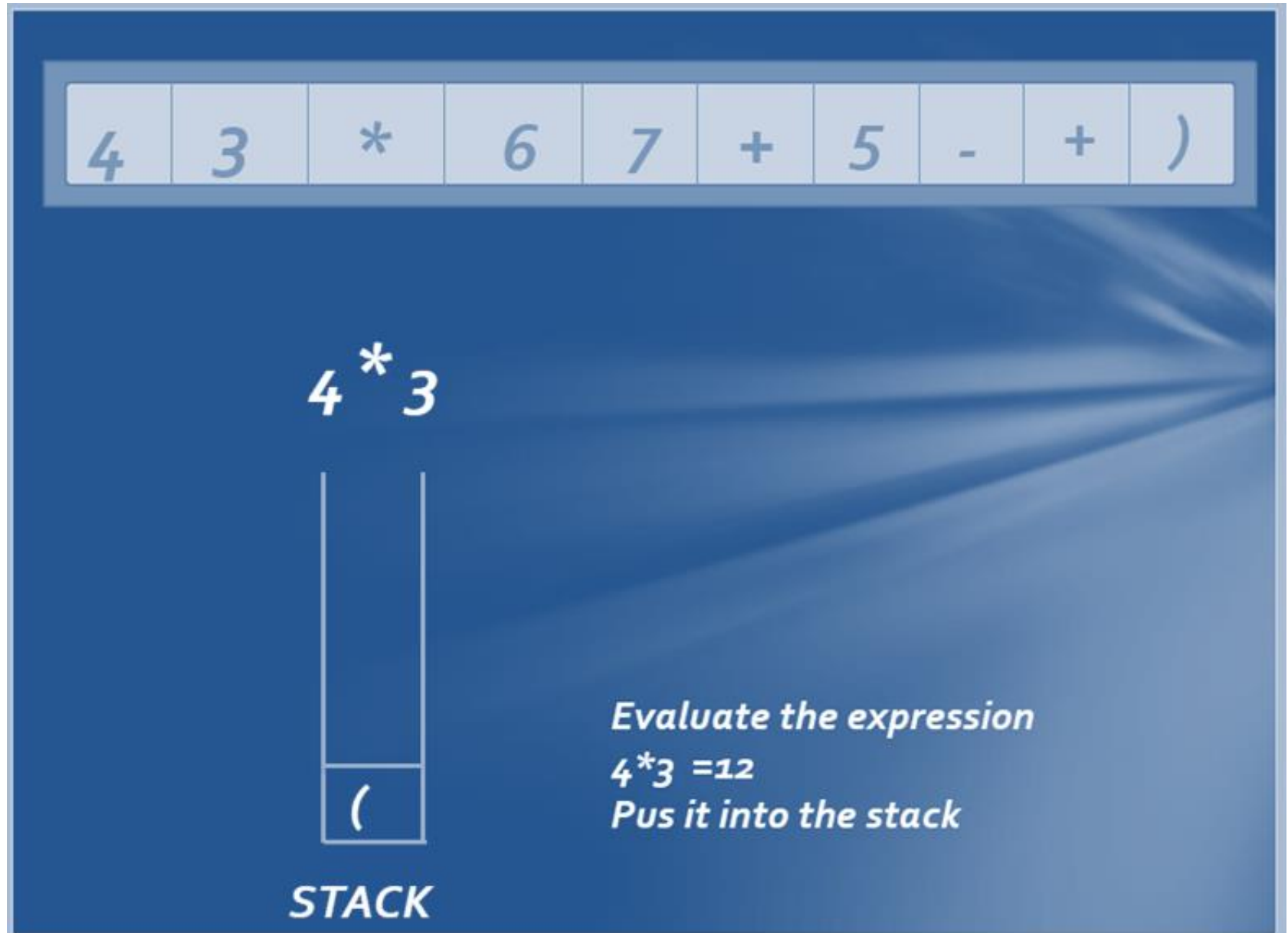




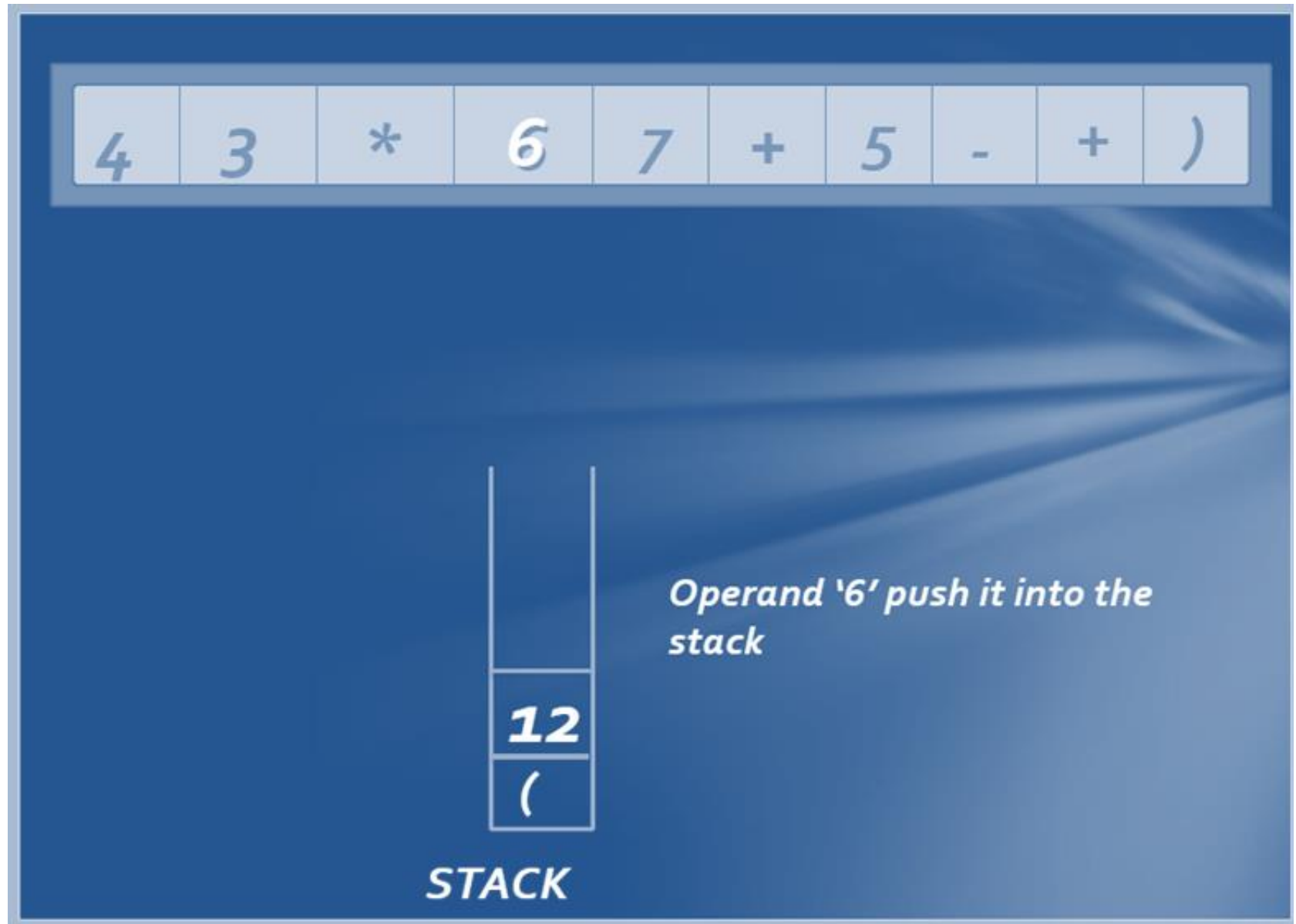
# Evaluating Postfix Expressions-Example 2



## Evaluating Postfix Expressions-Example 2



# Evaluating Postfix Expressions-Example 2



# Evaluating Postfix Expressions-Example 2

4	3	*	6	7	+	5	-	+	)
---	---	---	---	---	---	---	---	---	---

*Operand '7' push it into the stack*

6
12
(

**STACK**

# Evaluating Postfix Expressions-Example 2

4	3	*	6	7	+	5	-	+	)
---	---	---	---	---	---	---	---	---	---

**6 + 7**

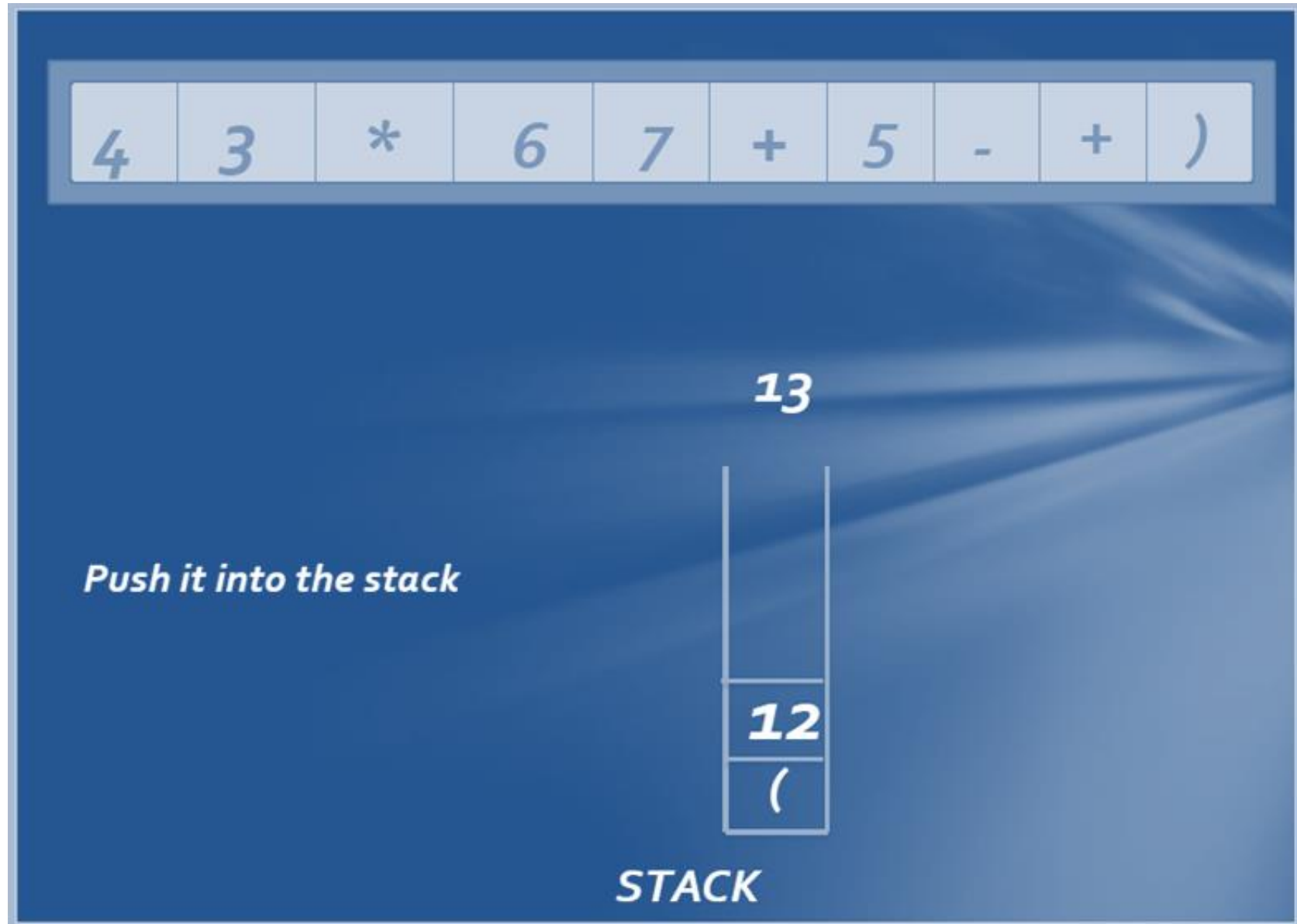
*Evaluate '6+7'*

**=13**



**STACK**

# Evaluating Postfix Expressions-Example 2



# Evaluating Postfix Expressions-Example 2

4 3 \* 6 7 + 5 - + )

*Operand '5' push it into the stack*

(  
12  
13

STACK

# Evaluating Postfix Expressions-Example 2

4	3	*	6	7	+	5	-	+	)
---	---	---	---	---	---	---	---	---	---

*Operands '5' and '13' are popped*

*Now operator '-' occurs*

5
13
12
(

STACK



# Evaluating Postfix Expressions-Example 2

4	3	*	6	7	+	5	-	+	)
---	---	---	---	---	---	---	---	---	---

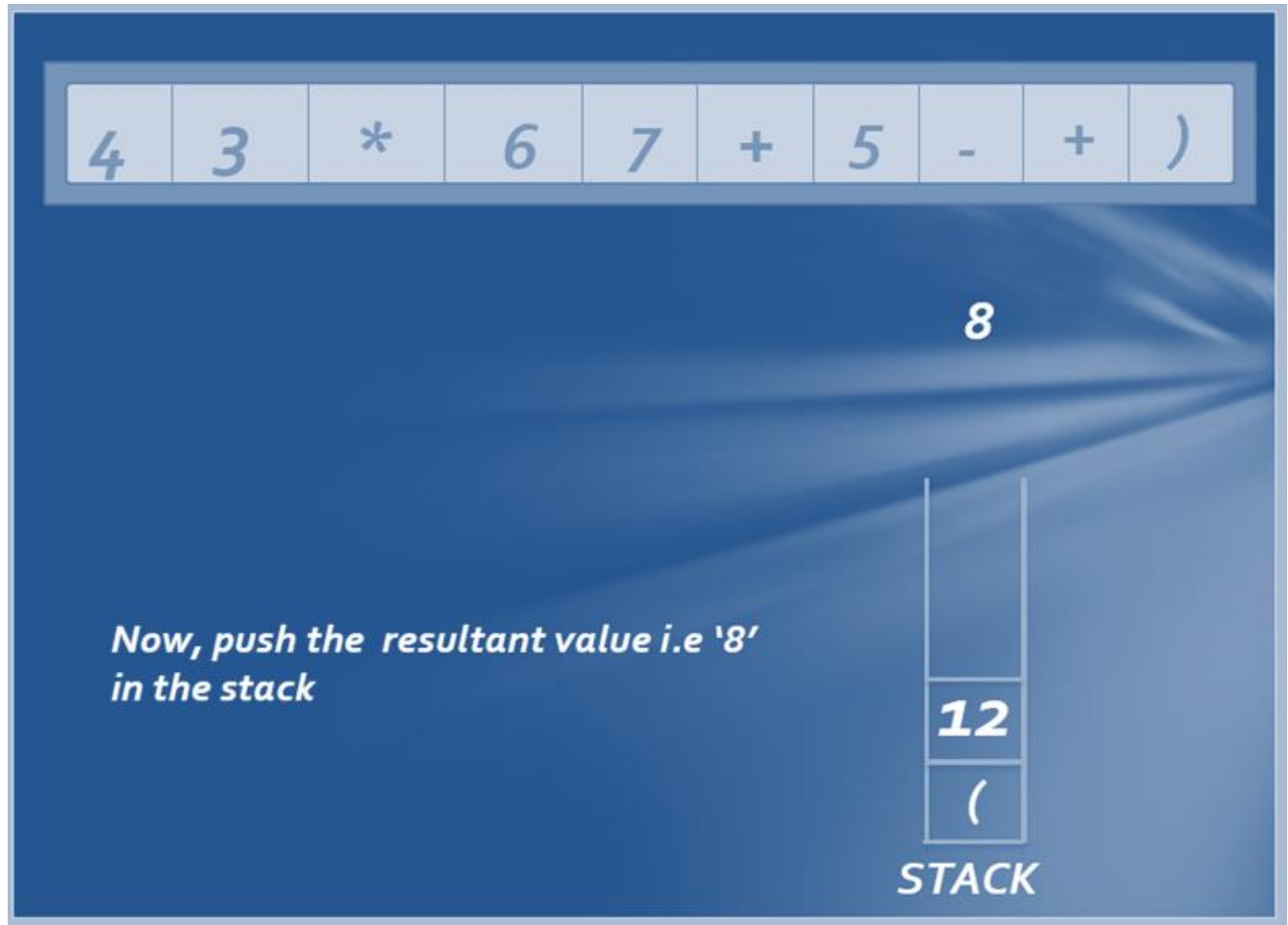
*Evaluating "13-5"*

*We get '8'*

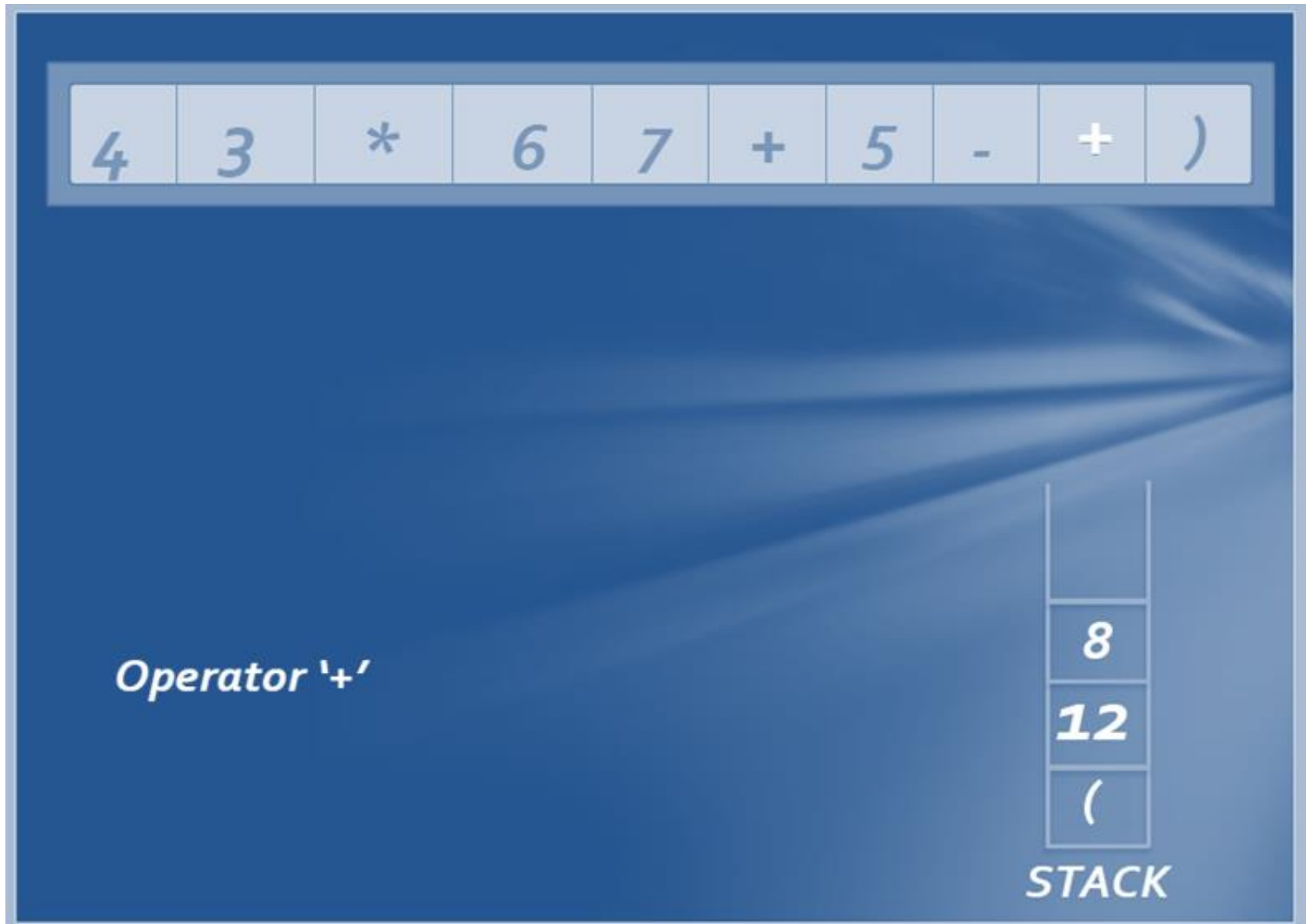
12
(

STACK

# Evaluating Postfix Expressions-Example 2



# Evaluating Postfix Expressions-Example 2



# Evaluating Postfix Expressions-Example 2

4	3	*	6	7	+	5	-	+	)
---	---	---	---	---	---	---	---	---	---

**12 + 8**

*By Evaluating , we get*

**20**

(

**STACK**

# Evaluating Postfix Expressions-Example 2



# Evaluating Postfix Expressions-Example 2

4	3	*	6	7	+	5	-	+	)
---	---	---	---	---	---	---	---	---	---

*Now atleast ')' occurs*

*Now , pop the element till  
the opening bracket*

20

(

STACK

# Evaluating Postfix Expressions using Stack

Start with an empty stack. We scan Postfix Expression from left to right.

While (we have not reached the end of P)

    If an operand is found

        push it onto the stack

    End-If

    If an operator is found

        Pop the stack and store the value as A

        Pop the stack and store the value as B

        Evaluate B op A using the operator just found.

        Push the resulting value onto the stack

    End-If

End-While

Pop the stack

# Evaluating Postfix Expressions using Stack

```
for (i = 0; postfix[i] != '\0'; i++)
{
    ch = postfix[i];
    if (isalnum(ch)) push(ch);
    else if (ch == '+' || ch == '-' || ch == '*' || ch == '/')
    {
        A = pop();
        B = pop();
        switch (ch)
        {
            case '*':
                val = B * A; break;
            case '/':
                val = B / A; break;
            case '+':
                val = B + A; break;
            case '-':
                val = B - A; break;
        }
        push(val);    /* push the value obtained above onto the stack */
    }
}
print("\n Result of expression evaluation : %d \n", pop());
```



# Applications of Stacks

- Conversion of Expressions
- Evaluate Postfix expression
- **Checking for Balanced Expression**

# Check for balanced expression or not

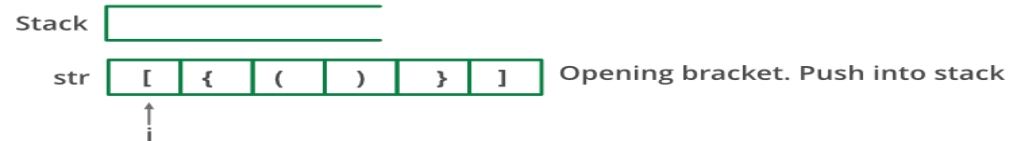
- Given an expression string `exp`, write a program to examine whether the pairs and the orders of “{”, “}”, “(”, “)”, “[”, “]” are correct in `exp`.
- **Example:**
  - **Input:** `exp = “[()]{}{[()()]()}"`  
**Output:** Balanced
  - **Input:** `exp = “[()]”`  
**Output:** Not Balanced

# Check for balanced expression or not

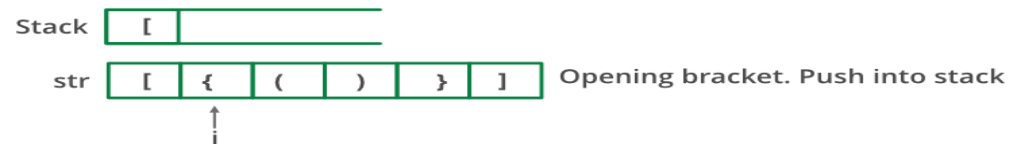
- **Algorithm:**
  - Declare a character stack S.
  - Now traverse the expression string exp.
    - If the current character is a starting bracket ('(' or '{' or '[') then push it to stack.
    - If the current character is a closing bracket (')' or '}' or ']') then pop from stack and if the popped character is the matching starting bracket then fine else brackets are not balanced.
  - After complete traversal, if there is some starting bracket left in stack then “not balanced”

# Check for balanced expression or not: Example 1

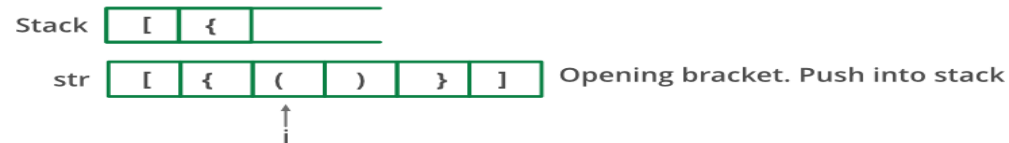
Initially :



Step 1:



Step 2:



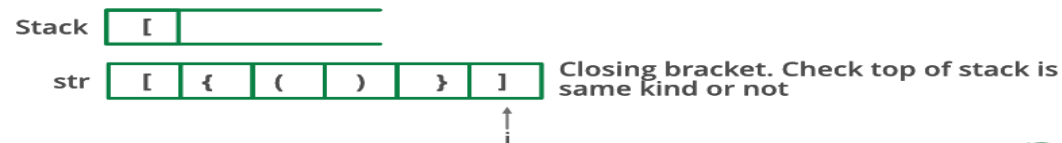
Step 3:



Step 4:



Step 5:



# Check for balanced expression or not: Example 2

Let string  $s = "[ ( ) ] \{ \} \{ [ ( ) ( ) ] ( ) \}"$  and  $i$  pointing the current index -

$i = 0, s = [ ( ) ] \{ \} \{ [ ( ) ( ) ] ( ) \}$	$\Rightarrow$	stack = [	
$i = 1, s = [ ( ] \{ \} \{ [ ( ) ( ) ] ( ) \}$	$\Rightarrow$	stack = ( [	
$i = 2, s = [ ( ) ] \{ \} \{ [ ( ) ( ) ] ( ) \}$	$\Rightarrow$	stack = [	(pop the top i.e. opening parentheses for ')')
$i = 3, s = [ ( ) ] \{ \} \{ [ ( ) ( ) ] ( ) \}$	$\Rightarrow$	stack =	(pop the top i.e. opening parentheses for ']')
$i = 4, s = [ ( ) ] \{ \} \{ [ ( ) ( ) ] ( ) \}$	$\Rightarrow$	stack = {	
$i = 5, s = [ ( ) ] \{ \} \{ [ ( ) ( ) ] ( ) \}$	$\Rightarrow$	stack =	(pop the top i.e. opening parentheses for ')')
$i = 6, s = [ ( ) ] \{ \} \{ [ ( ) ( ) ] ( ) \}$	$\Rightarrow$	stack = {	
$i = 7, s = [ ( ) ] \{ \} \{ [ ( ) ( ) ] ( ) \}$	$\Rightarrow$	stack = [ {	
$i = 8, s = [ ( ) ] \{ \} \{ [ ( ) ( ) ] ( ) \}$	$\Rightarrow$	stack = ( [ {	
$i = 9, s = [ ( ) ] \{ \} \{ [ ( ) ( ) ] ( ) \}$	$\Rightarrow$	stack = [ {	(pop the top i.e. opening parentheses for ')')
$i = 10, s = [ ( ) ] \{ \} \{ [ ( ) ( ) ] ( ) \}$	$\Rightarrow$	stack = ( [ {	
$i = 11, s = [ ( ) ] \{ \} \{ [ ( ) ( ) ] ( ) \}$	$\Rightarrow$	stack = [ {	(pop the top i.e. opening parentheses for ')')
$i = 12, s = [ ( ) ] \{ \} \{ [ ( ) ( ) ] ( ) \}$	$\Rightarrow$	stack = {	(pop the top i.e. opening parentheses for ']')
$i = 13, s = [ ( ) ] \{ \} \{ [ ( ) ( ) ] ( ) \}$	$\Rightarrow$	stack = ( {	
$i = 14, s = [ ( ) ] \{ \} \{ [ ( ) ( ) ] ( ) \}$	$\Rightarrow$	stack = {	(pop the top i.e. opening parentheses for ')')
$i = 15, s = [ ( ) ] \{ \} \{ [ ( ) ( ) ] ( ) \}$	$\Rightarrow$	stack =	(pop the top i.e. opening parentheses for ']')

$\Rightarrow$  As stack is empty at the end, therefore string expression is balanced.

# Check for balanced expression or not

**Algorithm balanced-parenthesis (expr)**

\\ expr is the input expression given by the user

```
{
    length = strlen (expr);
    for i=0 to length-1
    {
        if(expr[i] == '(' or expr[i] == '{' or expr[i] == '[')
            push(expr[i]);
        else if(expr[i] == ')' or expr[i] == '}' or expr[i] == ']')
        {
            if(isEmpty() or (!ArePair(gettop(), expr[i])))
                Write Result - Invalid expression - Not a Balanced one;
            else
                pop();
        }
    }
    if( isEmpty ())
        Write Result - Valid expression - Perfectly Balanced;
    else
        Write Result - Invalid expression - Not a Balanced one;
}
```

# Tasks for Practice

1. Convert the following Infix expressions to Postfix expressions.

a)  $5+3/8-(2^4-5)*2+(5^2*2^2)*1$

b)  $(a*b)/c*(a+b)-a^2+(a+b+a+b)*c$

2. Evaluation the above Postfix Expressions

a) evaluate postfix of 1(a)

b) at  $a=5$ ,  $b=2$ ,  $c=1$

3. Check the following expressions are balanced or not.

a)  $\{( )\}[( )] (\{[]\})[\{ \}]$

b)  $[()]\{\{()\}\}[][\{()\}]$

**IF YOU DON'T  
PRACTICE  
YOU DON'T  
DESERVE TO WIN**

— ANDRE AGASSI