Indian Institute of Information Technology Kottayam

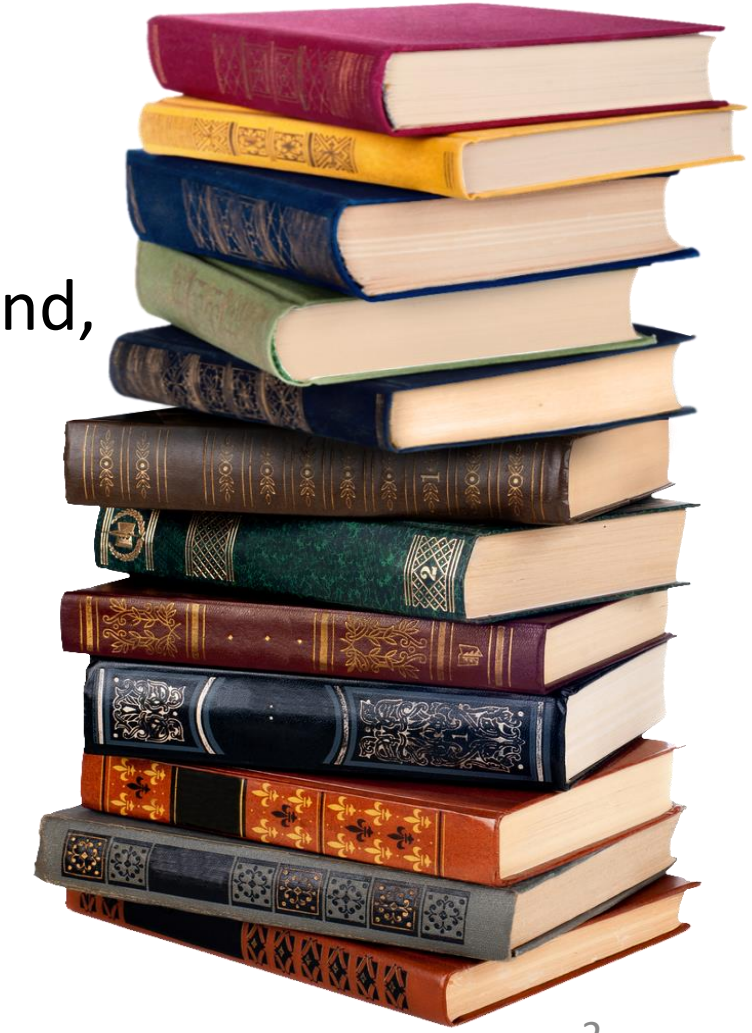
Stacks and Queue

BY

Arun Cyril Jose

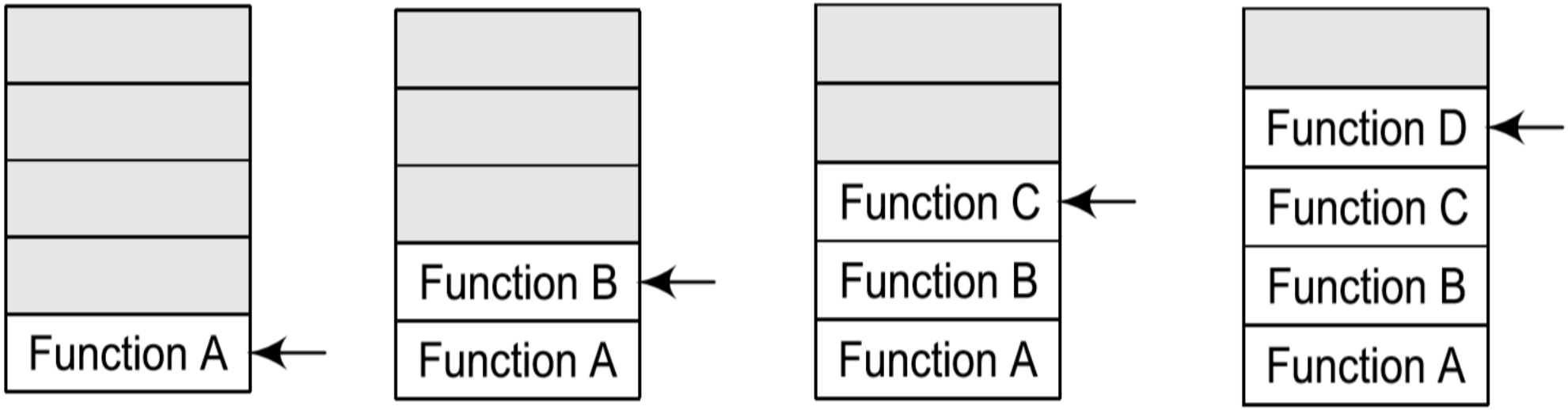
Stacks

- Linear Data Structure.
- Elements are added and removed only from one end, called the TOP.
- LIFO (Last-In-First-Out) data structure.



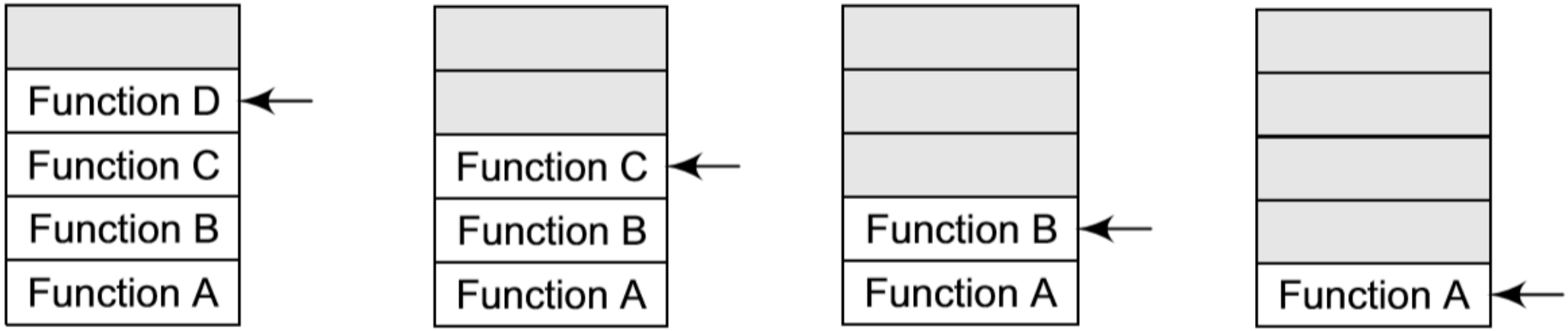
Stacks: Application

- Execution of functions.

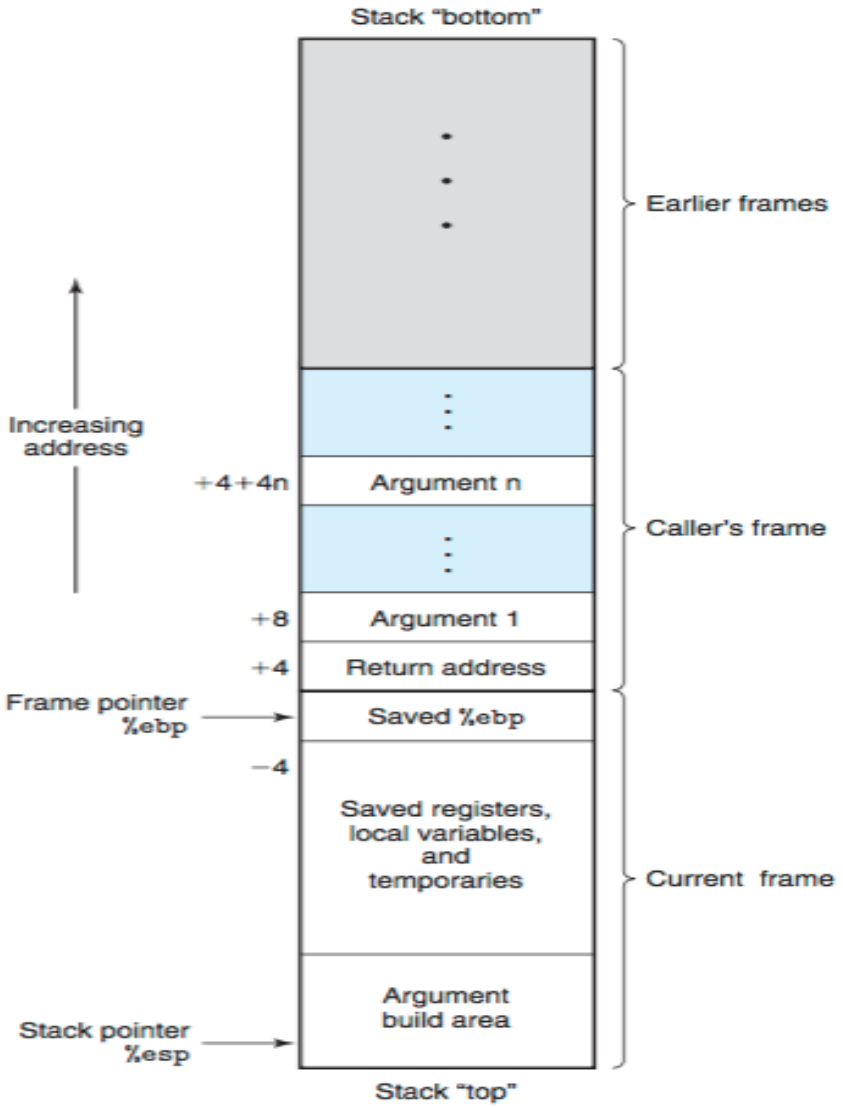


Stacks: Application

- We have already seen an important application.



Stacks: Application



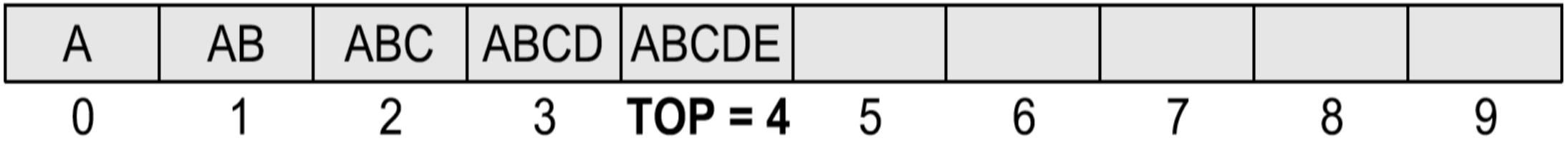
- Additional Reading: https://zhu45.org/posts/2017/Jul/30/understanding-how-function-call-works/

Operations on Stacks

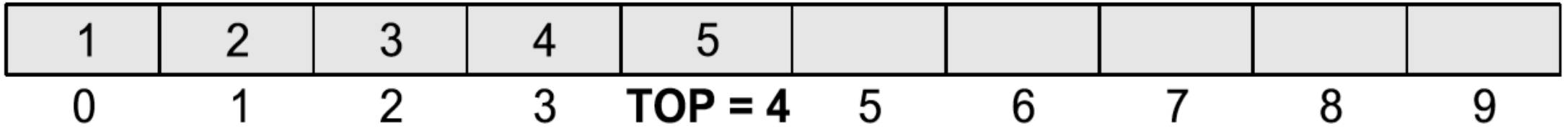
- **Push**: Adds an element to the top of the stack
- **Pop**: Removes the element from the top of the stack
- **Peek**: Returns the value of the topmost element of the stack

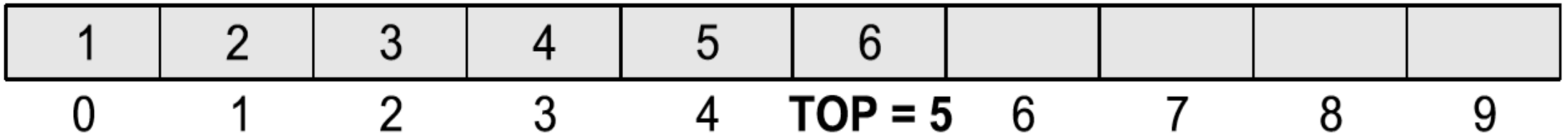
Array Implementation of Stack

- Can be represented as a linear array.
- Every stack has a variable called TOP associated with it.
  - TOP store the address of the topmost element of the stack.
  - Element will be added to or deleted from the TOP.
- MAX stores maximum number of elements a stack can hold.



Stacks: Push Operation

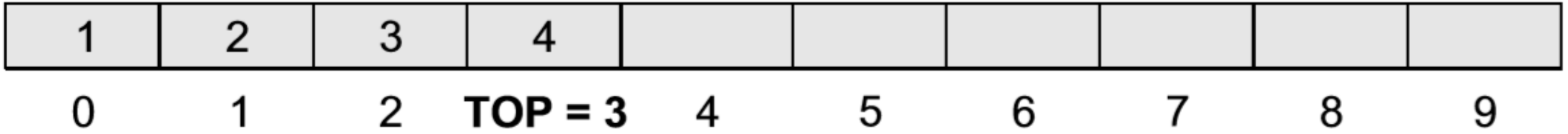




Stacks: Push Operation

```
Step 1: IF TOP = MAX-1
            PRINT "OVERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET TOP = TOP + 1
Step 3: SET STACK[TOP] = VALUE
Step 4: END
```

Stacks: Pop Operation



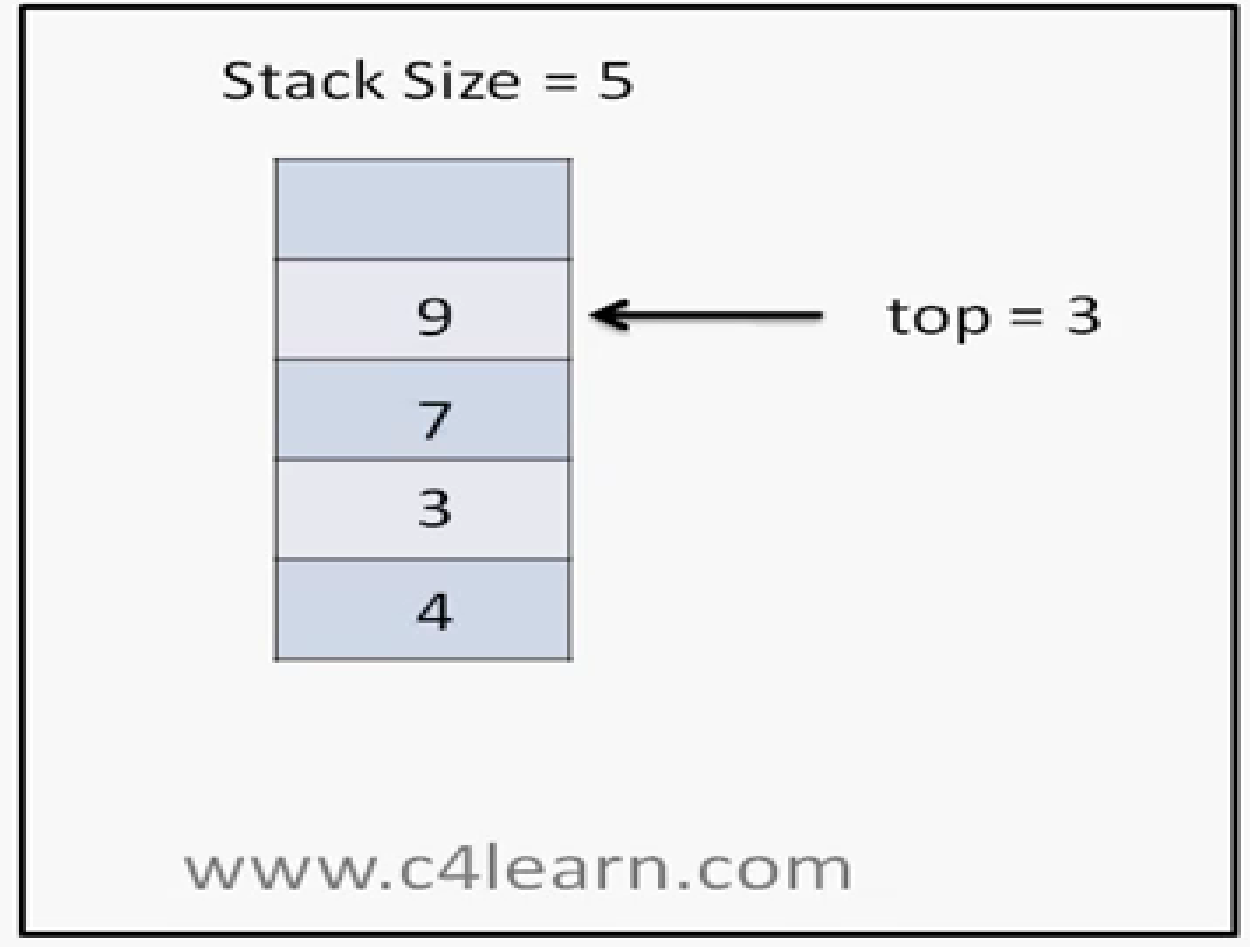
Stacks: Pop Operation

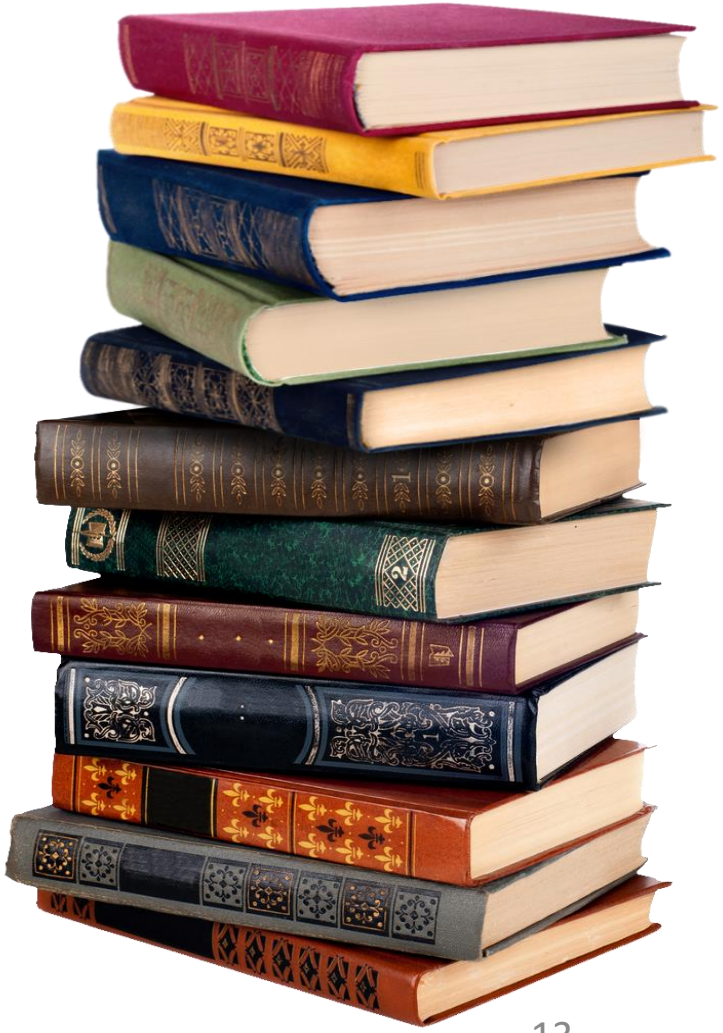
```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET VAL = STACK[TOP]
Step 3: SET TOP = TOP - 1
Step 4: END
```

Stacks: Peek Operation

- Returns the value of the topmost element of the stack without deleting it from the stack.

Stacks

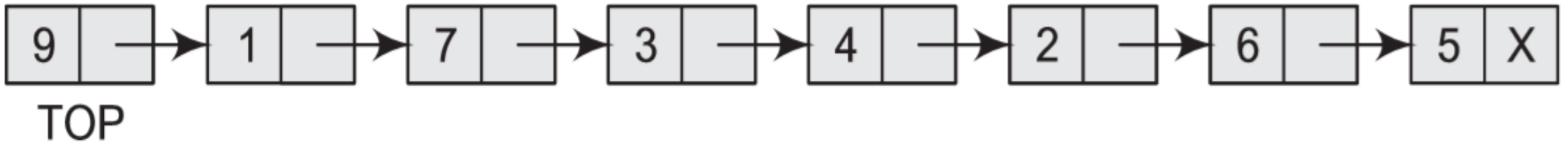




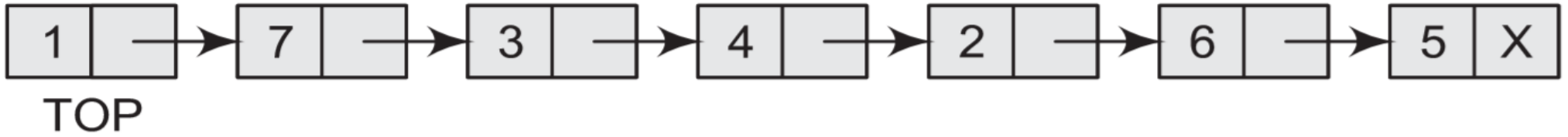
Linked List Implementation of Stack

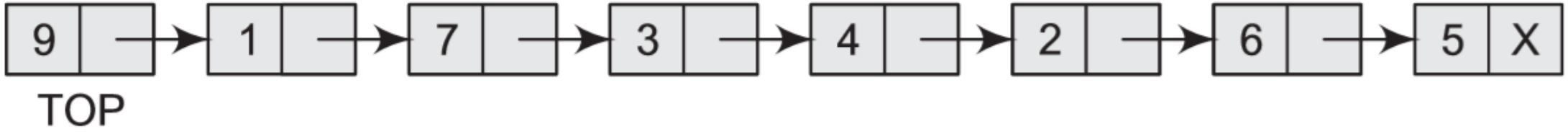
- Array must be declared to have some fixed size.
  - Efficient when the stack is very small or the maximum size is known in advance.
- Every node has two parts—data part and link to the next node.
- START pointer of the linked list is used as TOP.
- All insertions and deletions are done at the node pointed by TOP.

Linked List Implementation of Stack



Linked List Implementation of Stack: Push

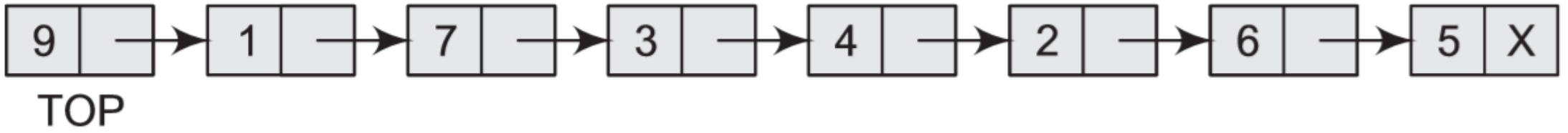


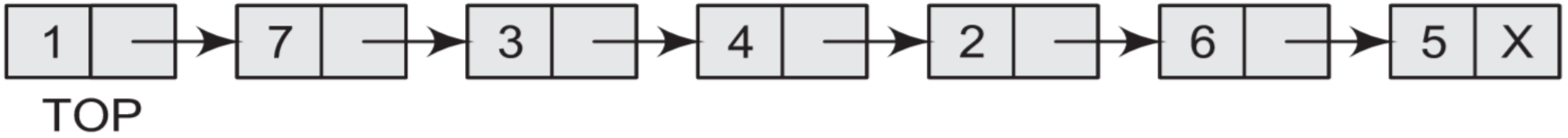


Linked List Implementation of Stack: Push

```
Step 1: Allocate memory for the new
        node and name it as NEW_NODE
Step 2: SET NEW_NODE -> DATA = VAL
Step 3: IF TOP = NULL
            SET NEW_NODE -> NEXT = NULL
            SET TOP = NEW_NODE
        ELSE
            SET NEW_NODE -> NEXT = TOP
            SET TOP = NEW_NODE
        [END OF IF]
Step 4: END
```

Linked List Implementation of Stack: Pop





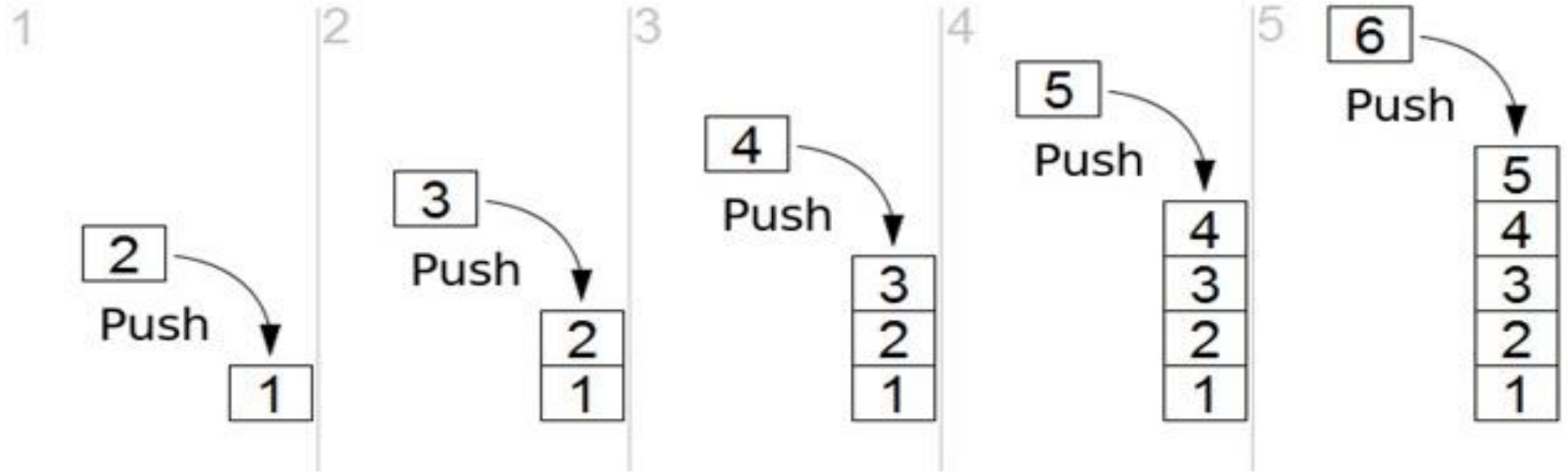
Linked List Implementation of Stack: Pop

```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 5
        [END OF IF]
Step 2: SET PTR = TOP
Step 3: SET TOP = TOP -> NEXT
Step 4: FREE PTR
Step 5: END
```

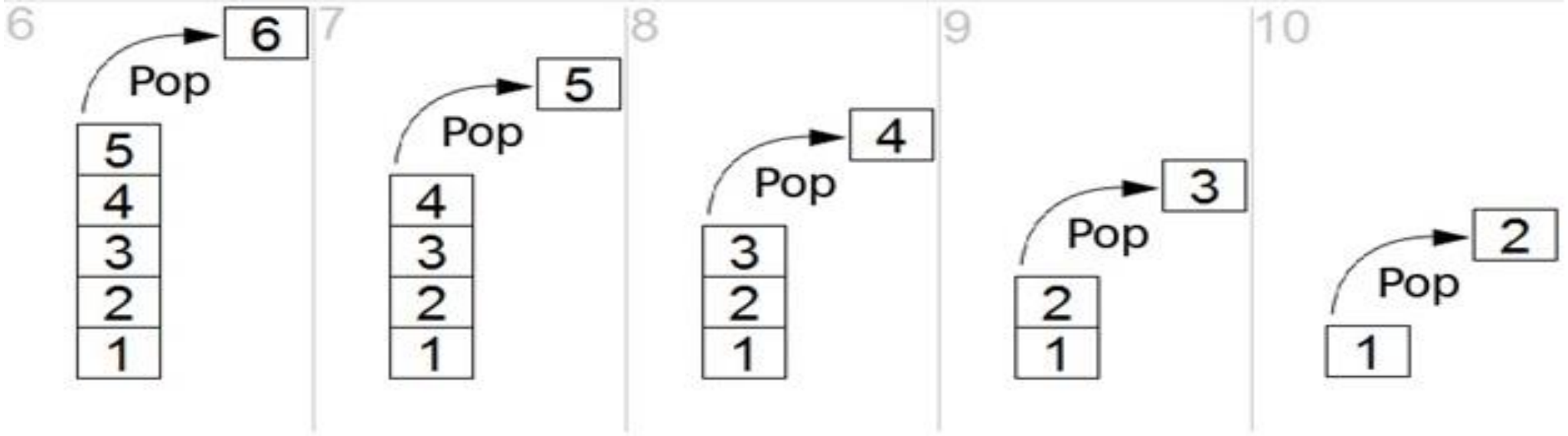
Reversing a List using Stacks

- Read each number from the array starting from first index and PUSH it on a stack.
- Continue until all numbers are read.
- POP each of them out and store it in the array starting from the first index.

Reversing a List using Stacks



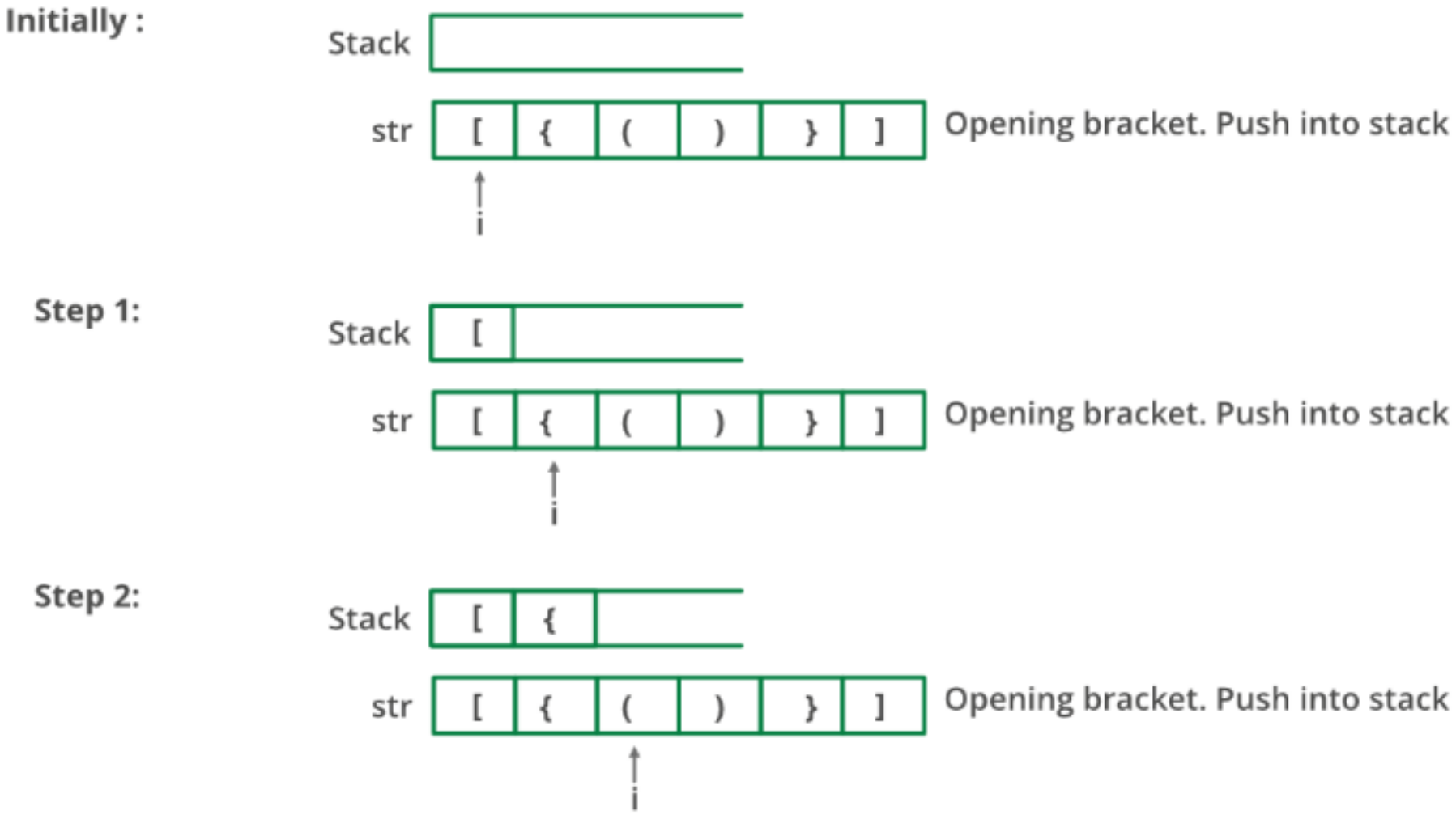
Reversing a List using Stacks



Parentheses Checker using Stacks

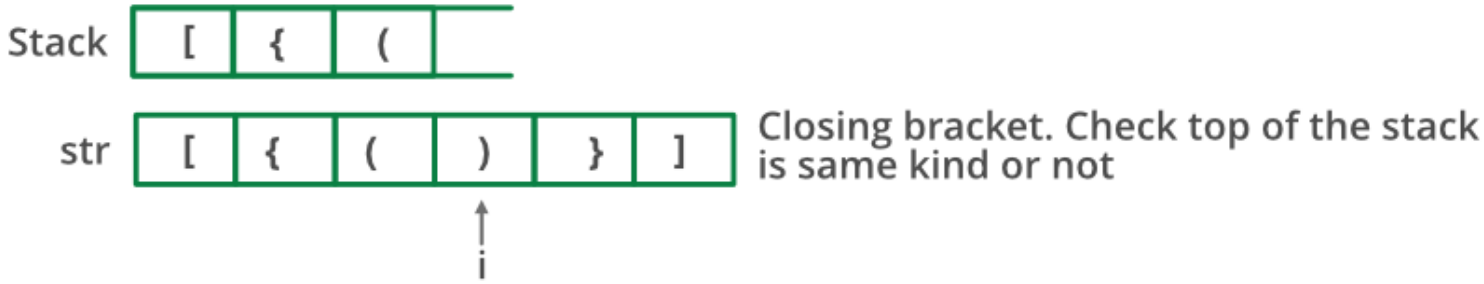
- Read the expression from left to right.
- If the current character is a starting bracket **‘(‘ or ‘{‘ or ‘[‘** then PUSH it to stack.
- If the current character is a closing bracket **‘)’ or ‘}’ or ‘]’** then POP from stack and the popped element must match with the corresponding opening bracket.
- If the parenthesis are not matched then the parenthesis are not balanced.

Parentheses Checker using Stacks

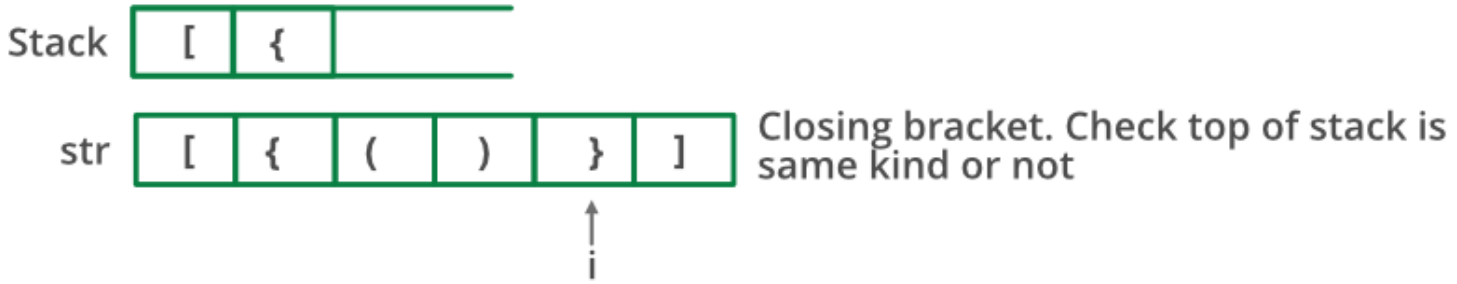


Parentheses Checker using Stacks

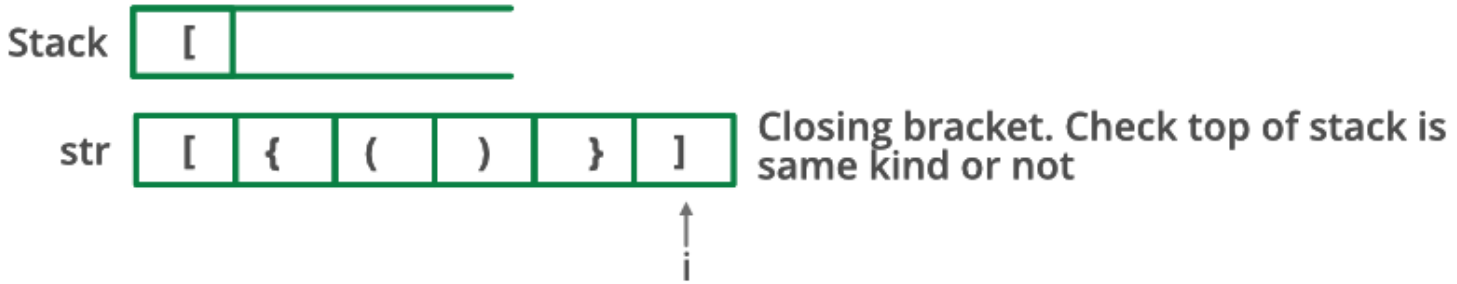
Step 3:



Step 4:



Step 5:



Infix, Postfix, and Prefix Notations

- Infix, postfix, and prefix notations.

- Infix Notation: Operator is placed in between the operands, E.g. A+B
- Postfix Notation: Operator is placed after the operands.

- Example: A + B in infix will become AB+ in postfix notation.

- Order of evaluation is always from left to right.

Infix, Postfix, and Prefix Notations

- **Infix:** (A + B) * C
- Postfix ??
- Infix: (A + B) * C
- [AB+]*C
- **Postfix: AB+C***

- **Infix:** (A–B) * (C+D)
- Postfix ??
- Infix: (A–B) * (C+D)
- [AB–] * [CD+]
- **Postfix: AB–CD+***

- **Infix:** (A + B) / (C + D) – (D * E)
- Postfix ??
- Infix: (A + B) / (C + D) – (D * E)
- [AB+] / [CD+] – [DE*]
- [AB+CD+/] – [DE*]
- **Postfix: AB+CD+/DE*–**

Infix, Postfix, and Prefix Notations

- Prefix Notation: The operator is placed before the operands.
- Example: A+B in infix becomes +AB in prefix notation.
- Evaluated from left to right.
- Like postfix, prefix expressions also do not follow the rules of operator precedence and associativity

Infix, Postfix, and Prefix Notations

- **Infix:** (A + B) * C
- Prefix ??
- Infix: (A + B) * C
- [+AB]*C
- **Prefix: *+ABC**

- **Infix:** (A–B) * (C+D)
- Prefix ??
- Infix: (A–B) * (C+D)
- [-AB] * [+CD]
- **Prefix: * –AB+CD**

- **Infix:** (A + B) / (C + D) – (D * E)
- Prefix ??
- Infix: (A + B) / (C + D) – (D * E)
- [+AB] / [+CD] – [*DE]
- [/+AB+CD] – [*DE]
- **Prefix: –/+AB+CD*DE**

Infix to Postfix Using Stacks

```
Step 1: Add ")" to the end of the infix expression
Step 2: Push "(" on to the stack
Step 3: Repeat until each character in the infix notation is scanned
       IF a "(" is encountered, push it on the stack
       IF an operand (whether a digit or a character) is encountered, add it to the
       postfix expression.
       IF a ")" is encountered, then
         a. Repeatedly pop from stack and add it to the postfix expression until a
            "(" is encountered.
         b. Discard the "(". That is, remove the "(" from stack and do not
            add it to the postfix expression
       IF an operator O is encountered, then
         a. Repeatedly pop from stack and add each operator (popped from the stack) to the
            postfix expression which has the same precedence or a higher precedence than O
         b. Push the operator O to the stack
       [END OF IF]
Step 4: Repeatedly pop from the stack and add it to the postfix expression until the stack is empty
Step 5: EXIT
```

Infix to Postfix Using Stacks

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

Infix Character Scanned	Stack	Postfix Expression
	(	
A	(	A
–	(–	A
(	(– (	A
B	(– (	A B
/	(– (/	A B
C	(– (/	A B C
+	(– (+	A B C /
(	(– (+ (	A B C /
D	(– (+ (	A B C / D
%	(– (+ (%	A B C / D
E	(– (+ (%	A B C / D E
*	(– (+ (% *	A B C / D E
F	(– (+ (% *	A B C / D E F
)	(– (+	A B C / D E F * %
/	(– (+ /	A B C / D E F * %
G	(– (+ /	A B C / D E F * % G
)	(–	A B C / D E F * % G / +
*	(– *	A B C / D E F * % G / +
H	(– *	A B C / D E F * % G / + H
)		A B C / D E F * % G / + H * –

Operator Precedence

Highest	( ) [ ]
	! ~ ++ -- (type) * & sizeof
	* / %
	+ —
	<< >>
	< <= > >=
	== !=
	&
	^
	|
	&&
	||
	?:
	= += -= *= /= etc.
Lowest	,

Infix to Prefix Using Stacks

```
Step 1: Reverse the infix string. Note that
        while reversing the string you must
        interchange left and right parentheses.
Step 2: Obtain the postfix expression of the
        infix expression obtained in Step 1.
Step 3: Reverse the postfix expression to get
        the prefix expression
```

- Infix expression: (A – B / C) * (A / K – L)
- Reverse the infix string: (L – K / A) * (C / B – A)
- Obtain the corresponding postfix expression: L K A / – C B / A – *
- Reverse the postfix expression: * – A / B C – /A K L