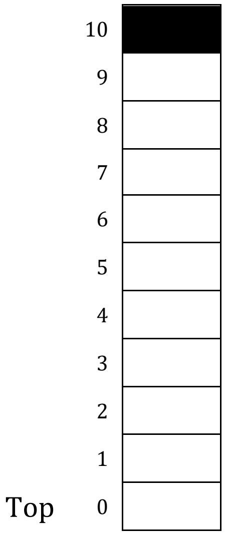
Stack

LIFO: Last In First Out.

Stack Operations:

Size = 10

Empty: if(top == 0)
Full: if(top == size)
push(x): insert x at top position and increase the value of top by 1.
pop( ): decrease the value of top by 1.
getTopElement( ): returns the element in top-1 position.
print( ): print the stack from top-1 to 0.



Stack Applications:

Mathematical Expression

Prefix: Operator-Operand-Operand // + A B

Infix: Operand-Operator-Operand // A + B

Postfix: Operand-Operand-Operator // A B +

Left to Right	BODMAS	PEMDAS
2 + 4 * 5 – 6 / 2 + 9 = 6 * 5 – 6 / 2 + 9 = 30 – 6 / 2 + 9 = 24 / 2 + 9 = 12 + 9 = 21	2 + 4 * 5 – 6 / 2 + 9 = 2 + 4 * 5 – 3 + 9 = 2 + 20 – 3 + 9 = 31 – 3 = 28	2 + 4 * 5 – 6 / 2 + 9 = 2 + 20 – 6/2 + 9 = 2 + 20 – 3 + 9 = 31 – 3 = 28

Input: Infix Expression

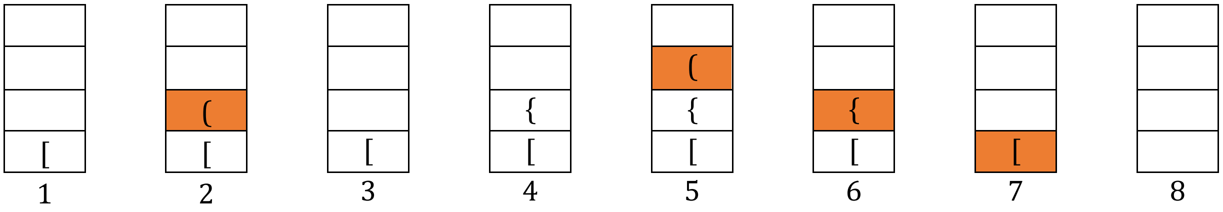
Process:

1. **Validate** the Infix Expression.
2. **Convert** the Infix Expression to Postfix Expression.
3. **Evaluate** the Postfix Expression.

Output: Result.

Part 1: Validate the Infix Expression

9	*	[	7	-	1	+	(	5	+	1	/	2	)	*	3	-	{	(	6	+	3	)	-	8	}	+	4	]



Input and Initialization: char infixExpr[ ], char validationStack[ ]

Process:

1. Read the Infix expression from left to right.
2. Check, each symbol.
    a. If it is an opening parenthesis, push it in the stack.
    b. Else if it is a closing parenthesis, check the top element of the stack.
        i. If it pairs up with the current symbol, pop from stack and the expression might be valid.
        ii. Else, it does not pair up with the current symbol, the expression is invalid. Exit.
    c. Else, it is an operator or operand, ignore.
3. Repeat 1 and 2 till the end of the expression.
4. If the stack is empty, the expression is valid. Else, the expression is invalid.

Output: Valid or Invalid

Part 2: Infix to Postfix Conversion

9	*	[	7	-	1	+	(	5	+	1	/	2	)	*	3	-	{	(	6	+	3	)	-	8	}	+	4	]
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29

9	7	1	-	5	1	2	/	+	3	*	+	6	3	+	8	-	-	4	+	*

											/	/								+	+							
									+	+	+	+						(	(	(	(		-	-				
							(	(	(	(	(	(		*	*		{	{	{	{	{	{	{	{				
				-	-	+	+	+	+	+	+	+	+	+	+	-	-	-	-	-	-	-	-	-	-	+	+	
		[	[	[	[	[	[	[	[	[	[	[	[	[	[	[	[	[	[	[	[	[	[	[	[	[	[	
	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29

Current	+	/	*	-	-	+	
Top	-	+	+	*	+	-	

Input and Initializations: char infixExpr[ ], char convertionStack[ ], char postfixExpr[ ]

Process:

1. Read the Infix expression from left to right.
2. Check, each symbol
    a. If, it is an operand, insert it in postfix expression.
    b. Else if, it is an opening parenthesis, push it in stack.
    c. Else if, it is an operator, check the top element of the stack.
        i. If it is an operator, compare the precedence of the operator with the current symbol.
            - If the precedence of top element is higher or equal to the current symbol, repeatedly pop it from the stack and insert it in the postfix expression.
            - Else, the precedence of top element is less than the current symbol, push the current symbol into the stack.
        ii. Else, (if it is an opening parenthesis or nothing), push the current symbol into stack.
    d. Else if it is a closing parenthesis, repeatedly pop from stack and insert in the postfix expression until the corresponding opening parenthesis is at the top. Now, remove the opening parenthesis from the stack.
3. Repeat 1 and 2 till the end of the expression.
4. At the end of the expression, pop all the elements from stack and insert in postfix expression.

Output: char postfixExpr[ ]

Part 3: Evaluate the postfix expression

9	7	1	-	5	1	2	/	+	3	*	+	6	3	+	8	-	-	4	+	*

						2														
					1	1	0		3				3		8					
		1		5	5	5	5	5	5	15		6	6	9	9	1		4		
	7	7	6	6	6	6	6	6	6	6	21	21	21	21	21	21	20	20	24	
9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	216
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

B	Op	A	Result
7	-	1	6
1	/	2	0
5	+	0	5
5	*	3	15
6	+	15	21
6	+	3	9
9	-	8	1
21	-	1	20
20	+	4	24
9	*	24	216

Input and Initializations: char postfixExpr[ ], int evaluationStack[ ];

Process:

1. Read the Postfix Expression from left to right.
2. Check each symbol,
    a. If, it is an operand, push it in stack.
    b. Else, it is an operator. Do the followings:
        i. Initialize the top element of the stack in a ***variable A***.
        ii. Pop from stack.
        iii. Initialize the top element of the stack in a ***variable B***.
        iv. Pop from stack
        v. Evaluate the operation ***B operator A***.
        vi. Push the ***result*** of the operation into the stack.
3. At the end of the expression, the top element of the stack is the result of the postfix expression.

Output: result

9	7	1	-	5	1	2	/	+	3	*	+	6	3	+	8	-	-	4	+	*

9	*	[	7	-	1	+	(	5	+	1	/	2	)	*	3	-	{	(	6	+	3	)	-	8	}	+	4	]
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29

= 9 * [7 – 1 + (5 + 0) * 3 – {9 – 8} + 4]
= 9 * [7 – 1 + 5 * 3 – 1 + 4]
= 9 * [7 – 1 + 15 – 1 +4]
= 9 * [26 – 2]
= 9 * 24
= 216