DSA CHAPTER 3

Q1. Define Stack. Write and explain Push and Pop operation algorithm of a stack.

=> A **stack** is a linear data structure that follows the **Last In, First Out (LIFO)** principle. This means that the last element added to the stack will be the first one to be removed. This Operations are performed by only one place known as Top of the stack.

=> Push Operation Algorithm and Explanation.

=> **Push algorithm**

```
Algorithm Push(stack, top, val)
{
        Step 1:         [Check Overflow]
                        If ( top >= size-1 )  then write
                        “Stack is Overflow”
                        Return
        Step 2:         [increment top value]
                        top  <= top+1
        Step 3:         [insert value in stack]
                        stack[top]  =  val
        Step 4:   exit
}
```

=> **Push Explanation**

- The **Push** operation first checks if the stack has space to add a new element (not full).
- If there's space, the **top** pointer is incremented to point to the next available slot.
- The new element is then placed at this position.

=> **Pop Algorithm**

Algorithm Pop (stack, top)

{

Step 1: [check stack underflow]

If (top == -1) then write

“Stack is underflow”

Return

Step 2: [remove top most element]

Temp <=stack[top]

Step 3: [Decrement top pointer]

Top <= top-1

Step 4: [Return Value]

Return temp

Step 5: exit

}

=> **Pop Explanation**

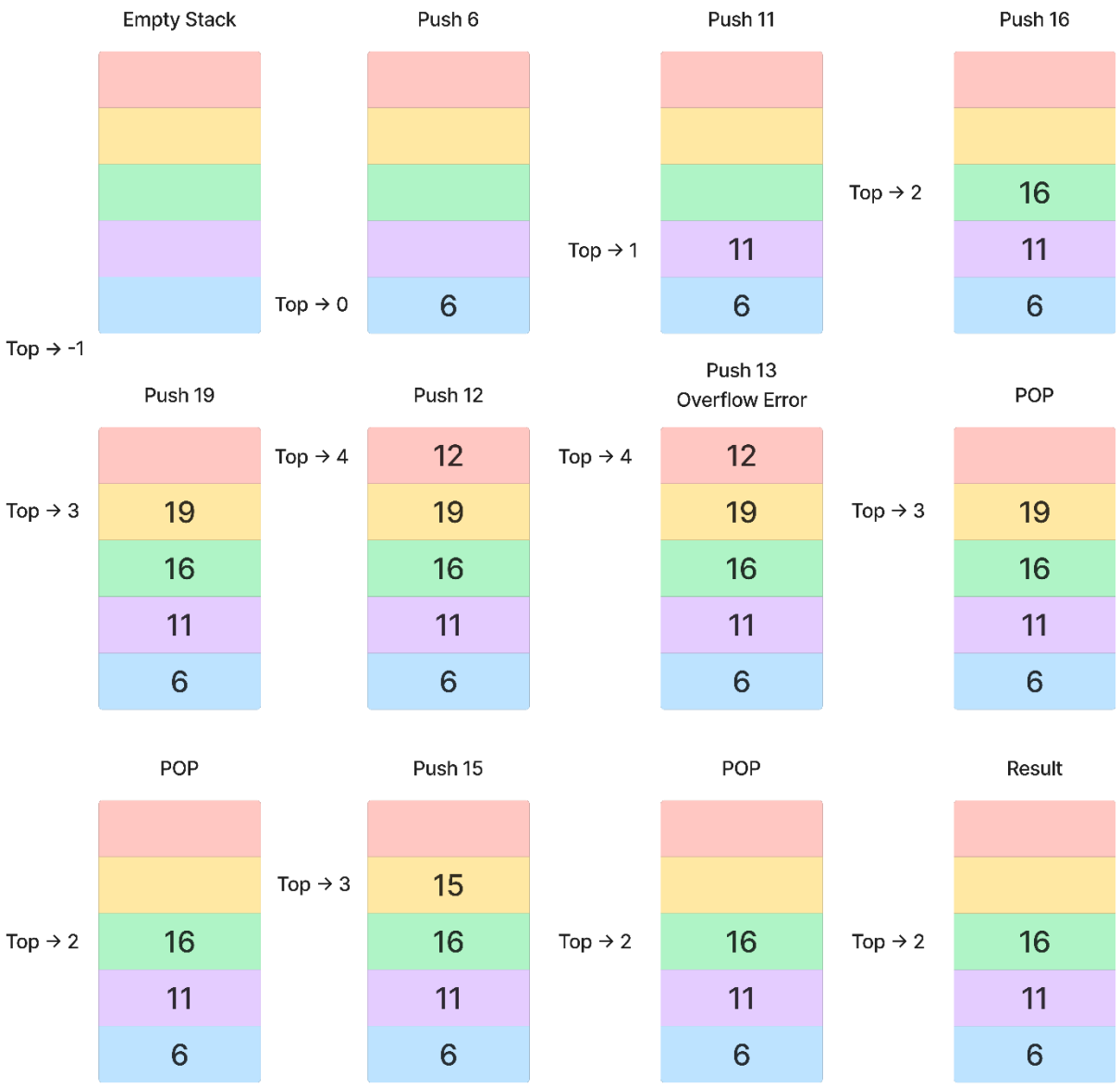
- The **Pop** operation first checks if the stack is empty.
- If it's not empty, the element at the top of the stack (pointed by **top**) is accessed.
- The **top** pointer is then decremented to effectively remove the element from the stack.

~Created By Dhruv Prajapati

Q2. Consider maximum size of stack is 5. Perform following operation on stack and show the status of stack and top pointer after each operation.

PUSH 6, PUSH 11, PUSH 16, PUSH 19, PUSH 12, PUSH 13, POP, POP, PUSH 15, POP

=> Here is Above stack operation Performed.



Q3. Define: infix, prefix and postfix expression. Convert following infix expressions into postfix expressions and prefix expressions:

a. (a + b) * c – (d – e)
b. A+B/C*D-E/F-G
c. A – (B / C + (D % E * F) / G)* H

=> Expressions in programming and mathematics are ways to write and evaluate arithmetic operations.

=> **Infix Expression**

- In an infix expression, the operator is placed between the operands.
- This is the most common notation used in arithmetic expressions.
- **Infix notation** is easy to read and understand for humans, but it can be difficult for computers to evaluate efficiently.

Example

A+B , 3 * (4 + 5)

=> **Prefix Expression (Polish Notation)**

- In a **prefix expression**, the operator is placed before the operands.
- No parentheses are needed, as the position of the operator and operands provides the expression's meaning.

Example

+AB , *+534

=> **Postfix Expression (Reverse Polish Notation**

- In a **postfix expression**, the operator is placed after the operands.
- Like prefix notation, no parentheses are required, as the order of operations is clear from the sequence.

Example

AB+ , 354+*

=> Infix: Operators are between operands (e.g., A + B).

=> Prefix: Operators are before operands (e.g., + A B).

=> Postfix: Operators are after operands (e.g., A B +).

~Created By Dhruv Prajapati

=> **(A) (a + b) * c – (d – e)**

-> Converting into Postfix

= ab+ * c – (d – e)

= ab+ * c – de-

= ab+ c* – de-

= ab+ c* de- -

-> Converting into Prefix

= +ab * c – (d – e)

= +ab * c – -de

= *+ab c – -de

= - * +ab c -de

=> **(B) A + B/C * D - E/F - G**

-> Converting into Postfix

= A + BC/ * D - E/F - G

= A + BC/D* – E/F - G

= A + BC/D* – EF/ - G

= ABC/D*+ – EF/ - G

= ABC/D*+EF/- - G

= ABC/D*+EF/-G-

-> Converting into Prefix

= A + /BC * D - E/F - G

= A + */BCD – E/F - G

= A + */BCD – /EF - G

= +A*/BCD – /EF - G

= -+A*/BCD/EF - G

= --+A*/BCD/EF G

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

-> Converting into Postfix

= A – (B/C+(DE% * F)/G) *H

= A – (B/C + DE%F* / G) *H

= A – (BC/ + DE%F* / G) *H

= A – (BC/ + DE%F*G/) *H

= A – BC/DE%F*G/+ * H

= A – BC/DE%F*G/+H*

= ABC/DE%F*G/+H*-

-> Converting into Postfix

= A – (B / C + (%DE * F) / G)* H

= A – (B / C + *%DEF / G) * H

= A – (/BC + *%DEF / G) * H

= A – (/BC + /*%DEFG) * H

= A – +/BC/*%DEFG * H

= A – *+/BC/*%DEFGH

= -A*+/BC/*%DEFGH

Q4. Convert following expression into reverse polish notation using stack:

a. a + b * c / d * e – f + g * h / i

b. (A-B↑C↑D) * (E – F/D)

c. (m+n)/(x-y)+z

=> **(A). a + b * c / d * e – f + g * h / I = a + b * c / d * e – f + g * h / I)**

Sr no	Scanned symbol	stack	Postfix expression
	(	(	
1	A	(	A
2	+	(+	A
3	B	(+	AB
4	*	(+*	AB
5	C	(+*	ABC
6	/	(+/	ABC*
7	D	(+/	ABC*D
8	*	(+*	ABC*D/
9	E	(+*	ABC*D/E
10	-	(-	ABC*D/E*+
11	F	(-	ABC*D/E*+F
12	+	(+	ABC*D/E*+F-
13	G	(+	ABC*D/E*+F-G
14	*	(+*	ABC*D/E*+F-G
15	H	(+*	ABC*D/E*+F-GH
16	/	(+/	ABC*D/E*+F-GH*
17	I	(+/	ABC*D/E*+F-GH*I
18	)	Stack Empty	ABC*D/E*+F-GH*I/+

=> (B). (A-B↑C↑D) * (E – F/D) = (A-B↑C↑D) * (E – F/D))

Sr No	Scanned Symbol	Stack	Postfix Expression
	(	(	
1	(	((	
2	A	((	A
3	-	((-	A
4	B	((-	AB
5	^	((-^	AB
6	C	((-^	ABC
7	^	((-^	ABC^
8	D	((-^	ABC^D
9	)	(	ABC^D^-
10	*	(*	ABC^D^-
11	(	(*(	ABC^D^-
12	E	(*(	ABC^D^-E
13	-	(*(-	ABC^D^-E
14	F	(*(-	ABC^D^-EF
15	/	(*(-/	ABC^D^-EF
16	D	(*(-/	ABC^D^-EFD
17	)	(*	ABC^D^-EFD/-
18	)	Empty Stack	ABC^D^-EFD/-*

=> (C). (m+n)/(x-y)+z = (m+n)/(x-y)+z)

Sr no	Scanned Symbol	Stack	Postfix Expression
	(	(	
1	(	((	
2	M	((	M
3	+	((+	M
4	N	((+	MN
5	)	(	MN+
6	/	(/	MN+
7	(	(/(	MN+
8	X	(/(	MN+X
9	-	(/(-	MN+X
10	Y	(/(-	MN+XY
11	)	(/	MN+XY-
12	+	(+	MN+XY-/
13	Z	(+	MN+XY-/Z
14	)	Empty Stack	MN+XY-/Z+

Q5. Convert following expression into reverse polish notation if required and evaluate it using stack.

a. 5*8 /2+1
b. 2 3 * 2 1 – / 5 4 1 – * +

=> Ans (a). First Convert infix into postfix to evaluate.

a. 5*8 /2+1 = 5*8 /2+1)

Sr no	Scanned Symbol	Stack	Postfix Expression
		(	
1	5	(	5
2	*	(*	5
3	8	(*	58
4	/	(/	58*
5	2	(/	58*2
6	+	(+	58*2/
7	1	(+	58*2/1
8	)		58*2/1+

=> Evaluate 58*2/1+

Sr no	symbol	Op1	Op2	value	Stack
1	5				5
2	8				5,8
3	*	5	8	5*8=40	40
4	2				40,2
5	/	40	2	40/2=20	20
6	1				20,1
7	+	20	1	20+1=21	21

=> **(B) 2 3 * 2 1 – / 5 4 1 – * +**

Sr no	symbol	Op1	Op2	value	Stack
1	2				2
2	3				2,3
3	*	2	3	2*3=6	6
4	2				6,2
5	1				6,2,1
6	-	2	1	2-1=1	6,1
7	/	6	1	6/1=6	6
8	5				6,5
9	4				6,5,4
10	1				6,5,4,1
11	-	4	1	4-1=3	6,5,3
12	*	5	3	5*3=15	6,15
13	+	6	15	6+15=21	21

Q.6 Explain recursion with the help of GCD example.

=> Recursion is a programming technique where a function calls itself directly or indirectly to solve a problem. Each recursive call typically reduces the problem's size, eventually reaching a base case that stops the recursion.

=> in every recursion function there are two case

1. Base case

2. Recursive case

(1) Base case

=> the **base case** is a condition that stops the recursion by solving the smallest possible subproblem directly, without making any further recursive calls.

(2) Recursive case

=> To Solve problem This are steps In Recursive case.

1. Divide problem into smaller subproblems.
2. Call recursive function using smaller subproblems
3. Combine solution of function subproblems.

Stack Management: Each recursive call is stored on the call stack until the base case is reached, after which the results are returned step-by-step.

=> Example of GCD

The Greatest Common Divisor (GCD) of two integers is the largest number that divides both of them without leaving a remainder.

$$a>b \begin{cases} GCD(a,b) = \begin{cases} b \text{ if } a\%b == 0 \\ \text{else } \{ GCD(b,a\%b) \end{cases} \end{cases}$$

GCD Diagram

=> Recursive Approach to GCD

- **Base Case:** If the second number (b) is 0, the GCD is a (because any number divided by 0 has no remainder, so a is the GCD).
- **Recursive Case:** Otherwise, the GCD of a and b is the same as the GCD of b and a % b (where % is the modulus operator, giving the remainder of the division of a by b).

Example GCD(48,18)

1. gcd(48, 18) → waits for gcd(18, 12)
2. gcd(18, 12) → waits for gcd(12, 6)
3. gcd(12, 6) → waits for gcd(6, 0)
4. gcd(6, 0) → returns 6, which is then returned up the call stack to all previous calls.

Q7. List out applications of stack

=>

1. Function Calls.
2. Recursion.
3. Memory Management.
4. Syntax Parsing.
5. Expression evaluation.
6. Revising a list.
7. Parentheses Checker.