Stack Data Structure

- Collection of similar type of elements. Element can be always added on top position and element can be deleted always from top position.

- Follows LIFO. Element which is added on stack at the end can be deleted first

- **Stack can be implemented**
  - Using Static Implementation of Memory
    - Stack size can not be increased or decreased at runtime.
    - Can be used when stack size is predefined
  - Using Dynamic Implementation of Memory
    - Stack size can be increased or decreased at runtime.
    - Optimized usage of memory

Stack Data Structure

- Operations can be performed on stack
  - **push** – adds a element on stack
    - isfull - checks full state of stack top == SIZE-1 ? TRUE : FALSE
  - **pop** – deletes top most element from stack
    - isempty – checks empty state of stack top == -1 ? TRUE : FALSE
  - **peek** – retrieves top most element from stack
    - isempty – checks empty state of stack top == -1 ? TRUE : FALSE

Applications of Stack

- Recursion
- Expression Conversion and Evaluation
- Undo / Redo Operations
- Forward / Backward Web pages
- Depth First search

Expressions

- **Expressions can be represented in 3 forms**
  - **Infix Expression** a + b Left Operand → Operator → Right Operand
  - **Prefix Expression** + a b Operator → Left Operand → Right Operand
  - **Postfix Expression** a b + → Left Operand → Right Operand → Operator

Infix to Postfix

```
If read char == ‘(‘ then
    Push Element on stack
Else if read char is operand then
    join to postfix string
Else if read char is operator then
    peek element from stack
    if peeked element is ‘(‘ then
        push element on stack
    else if peeked element is operator then
        if priority(peeked element) >= priority(read char)
            pop() element from stack
            join peeked element to postfix
        push read char on stack
Else if read char is ‘)’ then
    pop elements from stack till peeked element is ‘(‘
    join each popped element to postfix except ‘(‘
```

(A + (B / C – (D * E ^ F) + G) * H		(Infix to Postfix)
(	(	A
A	(	A
+	(+	A
(	(+(	A
B	(+(	AB
/	(+(/	AB
C	(+(/	ABC
-	(+(-	ABC/
(	(+(-(	ABC/
D	(+(-(	ABC/D
*	(+(-(*	ABC/D
E	(+(-(*	ABC/DE
^	(+(-(*^	ABC/DE
F	(+(-(*^	ABC/DEF

)	(+(-	ABC/DEF^*
+	(+(+	ABC/DEF^*-
G	(+(+	ABC/DEF^*-G
)	(+	ABC/DEF^*-G+
*	(+*	ABC/DEF^*-G+
H	(+*	ABC/DEF^*-G+H
)		ABC/DEF^*-G+H*+

Prefix to Infix

- Start reading string in reverse order
- If read char is operand push on stack
- Else If read char is operator pop() twice
    - Consider first pop() result as left operand
    - Consider second pop() result as right operand
    - Concatenate left operand →Operator → Right Operand
    - Consider converted infix operand hence push on stack

Postfix Evaluation

```
Read is char from postfix string till not null
If read char is operand
    push read char on stack
Else if read char is operator
    pop() element twice
    Consider first pop() result as right operand
    Consider second pop() result as left operand
    Evaluate operands based on read operator
    Consider result as operand and push on stack
If postfix string reaches to null pop() left element from stack is result
```

SUNBEAM

Thank you!