DS

Data structure

What is data structure?
- Organising data into memory
- Processing the data efficiently

Need of DS?
To achieve, 1. Efficiency
2. Reusability

Two types of data structure
Linear data structures (Basic)
Array
Structure
Linked List
Stack
Queue

Non - Linear data structures (Advanced)
Tree
Graph

Stack
Stack is Last in first out [LIFO] or
first in Last out [FILO] structure

Stack operations
- Push () - Add / insert / push
- Pop () - delete / remove / pop
- Peek () - return value of top index.
- is_empty ()
- is_full ()

size = 5

4	70	Top
3	60	
2	50	
1	40	
0	10	
-1		

- Insertion & deletion is allowed from only one end called as TOP
- Top will always point to last inserted data
- LIFO / FILO

```
int arr [5] ;
int top ;
```

1. Empty	Top == -1	Top == -1
2. Push	increment top Add data	top ++ arr[top] = data
3. Pop	decrement top	top --
4. full		top == n-1
5. Peek	Return value of top index	return arr [top]

Expression & types.

Expression : Collection of operator & oprands
+, -, * 2, 3,

Types : 1. Infix - a+b human

2. Prefix - +ab comp

3. Postfix - ab+ comp

Priorities :-

()	$	* / %	+ -
bracket	power	m d r S R	p m

Infix to post fix Conversion.

Infix Expression : a * b / c * d + e - f * h + i

Process it from left to right & set Priorities.

1 4 2 3 5 6 5 4 7
a * b / c * d + e - f * h + i

Post fix Expression.

1 2 3 5 6 4 7
a * b / c * d + e - f * h + i

ab* / c * d + e - f * h + i
ab*c/ * d + e - f * h + i
ab*c/d * + e - f * h + i
ab*c/d * + e - f h* + i
ab* c/d * e + - f h* + i
ab*c/d * e + fh* - + i
ab*c/d * e+ fh* - i +

Infix to pre fix Conversion

Infix Expression : a * b / c * d + e - f * h + i

set priorities : 1 2 3 5 6 4 7
a * b / c * d + e - f * h + i

*ab / c * d + e - f * h + i

/ * abc * d + e - f * h + i

* / * abcd + e - f * h + i
* / * abcd + e - * fh + i
+ * / * abcd e - * fh + i
- + * / * abcd e * fh + i
+ - + * / * abcde * fhi

Post fix evaluation

- Process Expression from left to right
- If element is oprand - Push it on stack

- If element is operator -
  Pop 2 elements (oprands) from stack
  First poped ele - Op-2
  secnd poped ele - Op-1
  Perform operation

- Push back result on stack

- single value remain on stack ⇒ Result.

Expression : 4 5 6 * 3 / + 9 + 7 -

	operator		op		op		op		op	
6	*	3	/		+		+		-	
5	↓	30	↓	10		9		7		
4	5 * 6	4	30 / 3	4	4 + 10	14	14 + 9	23	23 - 7	16

Ans :- 16.

Prefix Evaluation

- ~~Left~~ ~~to~~ right to left
- Element is oprand - Push it on stack
- Element is operator
  Pop 2 elements (oprands) from stack
  First poped ele - Op-1
  2nd poped ele - Op-2
  perform operation
- Push back result on stack
- single value remain on stack ⇒ stack

Expression : - + + 4 / * 5 6 3 9 7

	op		op		op		op		op
5	*		/		+		+		-
6	5 * 6	30	30/3	4					23-7
3	= 30	3		10	4+10	14	14+9		
9		9		9		9		23	
7		7		7		7		7	16

Ans :- 16.

Prefix to Postfix

- Right to left
- Element is operand - Push it on stack

- Element is operator
  Pop two elements from stack
  1st ele - OP-1
  2nd ele - OP-2

- Form a string OP-1 + OP2 + Op^r
  Push back on stack

- Repeat until end of prefix expression
- Last remaining on stack is post fix expression

Expression : *+ab-cd

	op		op		op
	-	a	+		*
dc	cd-	b		ab+	
d		cd-		cd-	
					ab+ cd- *

Postfix to infix.

- Left to right
- If element is operand - Push it on stack

- If element is operator
  Pop two elements from stack
  1st ele - OP-2
  2nd ele - OP-1

- form a string OP-1 + Opr + Op-2
  Push back on stack

- Repeat until end of postfix expression
- Last remaining on stack is infix expression

Expression : abc-+de-fg-h+/*

	op -		op +		op -		op -		op +
c				e		g		h	
b	b-c	b-c	a+b-c	d		f	f-g	f-g	
a		a		a+b-c		d-e		d-e	
						a+b-c		a+b-c	

	op /	op *
→ f-g+h	d-e/f-g+h	
d-e	d	a+b-c * d-e/f-g+h
a+b-c	a+b-c	

Ans:- a+b-c * d-e/f-g+h