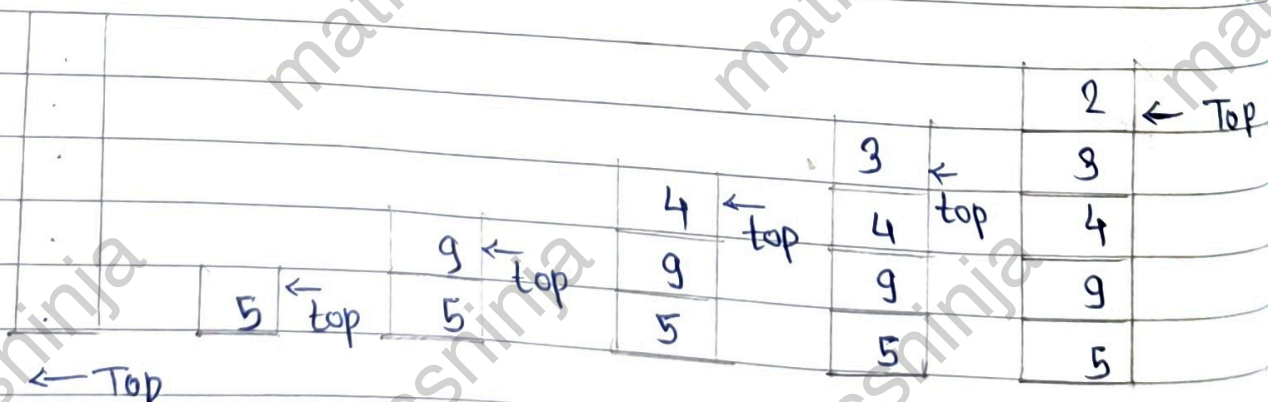
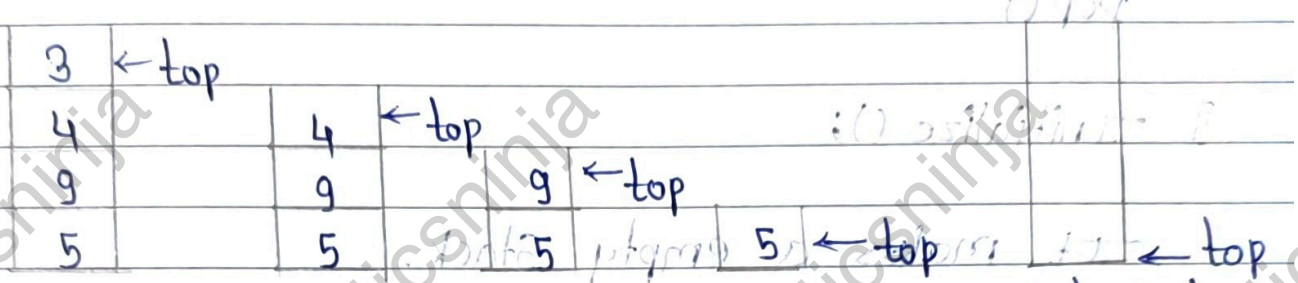
4. Stack

- Stack is a linear data structure as it stores all elements in sequential manner.
- It is a LIFO structure (Last in First out)
- It is an orderlist of the same type of elements.
- It is a linear list where all insertion & deletion are permitted only at one end of the list.
- When elements are added to stack it grows at one end. Similarly when elements are deleted from stack it shrinks at the same end.
- Figure shows expansation & shrinking of stack Initially stack is empty.

1] Insertion of elements : 5, 9, 4, 3, 2.



2] Deletion of element



- A variable top points to the top element of the list.
- Stack can be represented by using array
- It can be defined as follow:

```
typedef struct stack
{
 int data [size];
 int top;
} stack
```

- operations on stack

There are several operations that can be performed on stack.

Initialize()
Empty()

PAGE NO.:

Full()
Push()
Pop()

1] Initialize();

It makes a empty stack.

2] Empty() :

To determine if a stack is empty or not. It returns 1 or 0 depending on whether the stack is empty or not.

3] Full() :

It is used to determine if a stack is full or not It returns 1 or 0 depending on whether the stack is full or not.

4] Push() :
If a stack is not Full then Push a new element at the top of stack.

5] Pop() :

If a stack is not empty then Pop the element from it's top

Q. The stack contain 10, 20, 22, 26, 28, 30 with 30 being at the top of stack show diagramatically the affect of :

- PUSH 46
- PUSH 48
- Pop
- Pop
- Pop
- PUSH 82

30	← Top
28	
26	
22	
20	
10	

1] Initial

46	← Top
30	
28	
26	
22	
20	
10	

2] PUSH 46

48	← Top
46	
30	
28	
26	
22	
20	
10	

3] PUSH 48

46	← Top
30	
28	
26	
22	
20	
10	

4] Pop

PAGE NO.:

30 ← Top
28
26
22
20
10

5] pop

28 ← Top
26
22
20
10

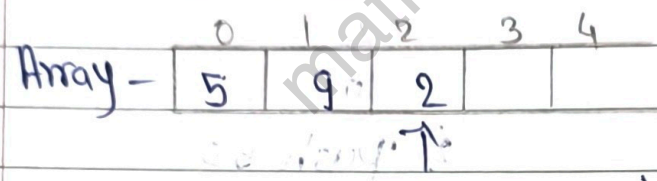
6] pop

82 ← Top
28
26
22
20
10

7] PUSH 82

⟹ Array Representation

one dimentional array used to hold elements of stack. Another variable top is used to keep track of the index of the topmost element.



Stack grows from left to right.

Initially top = -1 when the value of top becomes max-1 after a series of insertion it is full. A stack with top -1 is an empty stack. After push operation top = top+1 [Stack growing]
After pop operation top = top -1 [Stack shrinking]

⟹ C function for premitive operations on Stack.

1] Initialize

```
Void initialize (Stack *s)
{
    s->top = -1;
}
```

iii) Empty

```
int empty (stack *s)
{
  if (s->top == -1)
  return(1);
  return(0);
}
```

iii) Full

```
int full (stack *s)
{
 if (s->top == max-1)
 return (1);
  return (0);
}
```

iv) Push

```
Void push (stack *s, int x)
{
 s->top = s->top+1;
 s->data [s->top] = x;
}
```

v) Pop

```
int pop (stack * s, int x)
{
  x = s->data [s->top];
   s->top = s->top-1;
} return(x);
```

⟹ Overflow and underflow condition

Stack overflow is caused by insertion in stack that is already full. ~~Stack underflow~~

Stack underflow Deletion from an empty stack will cause a stack underflow.

~~LIK~~

- LINK Representation of stack.

Stack can be represented efficiently through linked list

Stack can be represented by using singly connected linked list

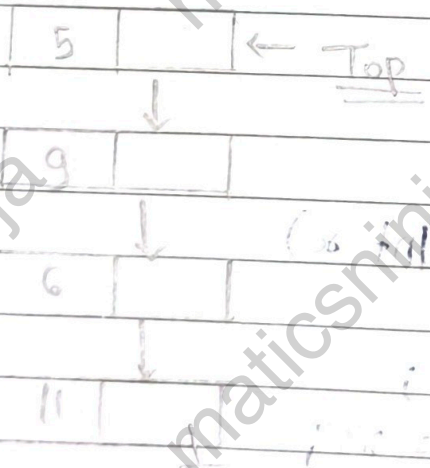
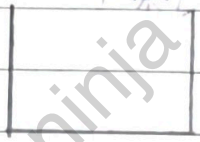


fig - Linked list representation of stack.

- Expression evaluate the postfix using stack.

1] Evaluation of 653+9*+

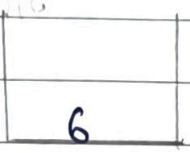
653+9*+



Initially stack is empty

empty

53+9*+

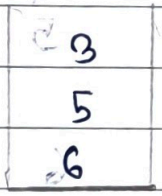


6 is an operan
So push on sta

3+9*+

5 is an operand
So push on stac

+9*+

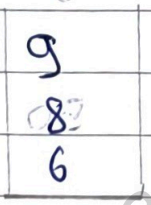


3 is an operand
Push on stack

9*+

Pop 2 topmost operanc
Perform operation &
Push result on sta

*+



9 is an operand
Push on stack.

+

Pop top2 most eleme
Perform addition
operation & Push
result on stack.

78 ← Final result

Evaluation of 653 + 9 * + = 78

① 546 + * 498 / + *

Expression	Stack	Remark
546 + * 498 / + *		Initially stack is empty
46 + * 498 / + *	5	5 is an operand so push on stack
6 + * 498 / + *	4 5	4 is an operand so push on stack
+ * 498 / + *	6 4 5	6 is an operand so push on stack
* 498 / + *	10 5	Pop topmost 2 element & add put result.
498 / + *	50	Pop topmost 2 element & multiply put result
98 / + *	4 50	4 is an operand then push on stack.

3/+*	9 4 50	9 is an operand then push on stack.
/+*	3 9 4 50	3 is an operand Push on stack
+*	3 4 50	Pop 2 element & Divide it
*	7 50	Pop 2 element & add it
end	350	Pop 2 element & multiply and put into result.
	350	← Final result.

- Algorithm

```
x: tokentype
operand 1, operand 2 : operandtype;
operater : operatertype;
 S: Stack
initialize (S);
x = nexttoken();
while (x)
 {
(if x is an operater)
 {
operand 2 = Pop (S);
}
operand 1 = Pop (S);
push (S, Evaluate (x, operand 1, operand 2));
}
else
 {
Push (S,x);
 x = nexttoken();
}
```

Evalution of Prefix.

A prefix expression is evaluated from right to left

An operand is Push on top of stack in case of operator 2 operands are Pop from the the stack evaluated in case of an operator

Two operands are from the stack evaluated on stack

Final result is found on top of stack.

Q: A=16, B=2, C=3, D=10, E=4 Prefix expression
-+/A^BC*DE*AC

Input	Stack	Result
-+/A^BC*DE*AC		Initially Stack is empty
-+/A^BC*DE*A	3	C=3 is an operand Push into stack
-+/A^BC*DE*	16 3	A=16 is an operand Push into stack
-+/A^BC*DE	48	Pop two topmost elemen & multiply put result into stack
-+/A^BC*D	4 48	E=4 is an operand Push into stack
-+/A^BC*	10 4 48	D=10 is an operand Push into stack
-+/A^BC	40 48	Pop two topmost element & multiply push result into stack
-+/A^B	3 40 48	C=3 is an operand Push into stack

-+/A^	2 3 40 48	B=2 is an operand Push into stack
-+/A	8 40 48	Pop topmost element & return exponent & result into stack $2^3 = 8$
-+/	16 8 40 48	A=16 is an operand Push into stack.
-+	2 40 48	Pop 2 topmost element & Divide push result into stack
-	42 48	Pop 2 topmost element & add it push result stack.
	-6	Pop 2 topmost element & Subtract push result Stack
End	-6	← Final result.

Algorithm for evaluation of prefix expression.

```
S: Stack ;
Initialize S ;
Prefix ← Read the prefix expression;
for (i ← index of the last character of prefix ,
decrease to zero)
{
  x ← Prefix [i];
  if ( x is an operand)
  {
  push S, char_to_int (x);
  };
  else
  {
  operand 1 = pop (S);
  operand 2 = pop (S);
  Val ← evaluate (x, operand1, operand2)
  push (S, val);
  }
}
  Val = pop(S);
  Print Val;
```

Q1. Explain Conversion of infix to postfix with example.

→ An infix expression can be converted into postfix from with the help of following steps.

Step 1 : Scan the infix expression from left to right.

Step 2 : If the current token is an operand (number or Variable) put into output.

Step3 : If the current token is an operator (+, -, /, * etc) push onto the stack.

Step4 : Pop operators from the stack with higher or equal precedence & output them.

Ex:

A*(B+C)/D-G

Input	Stack	output
A*(B+C)/D-G	Empty	Null
*(B+C)/D-G	Empty	A
(B+C)/D-G	*	A
B+C)/D-G	*(	A
+C)/D-G	*(	AB

C)/D-G	*(+	AB
)/D-G	*(+	ABC
/D-G	*	ABC+
D-G	/	ABC+*
-G	/	ABC+*D
G	-	ABC+*D/
End	-	ABC+*D/G
End	Empty	ABC+*D/G-

Q2. Explain Conversion of infix to prefix with example.

→ An infix expression can be Converted into prefix form with the help of following steps.

Step 1: Reverse the infix expression.

Step 2: make every opening bracket as closing bracket & every closing bracket as a opening bracket.

Step 3: Convert the expression in Postfix.

Step 4: Reverse the postfix expression.

Ex: (A-B/C) * (D* E-F)

→

Step 1 : Reversing the expression we get
)F-E*D(*)C/B+A(

Step 2 : Make closing bracket as openening)as(and make opening bracket as closing bracket (as).

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

Step 3 : Converting to Postfix form.

Input	Stack	output
(F-E*D)*(C/B-A)	empty	NULL
F-E*D)*(C/B-A)	(	Null
-E*D)*(C/B-A)	(	F
E*D)*(C/B-A)	(-	F
D)(C/B-A)	(-	FE
D)*(C/B-A)	(-*	FE
)*(C/B-A)	(-*	FED

*(C/B-A)	empty	FED*-
(C/B-A)	*	FED*-
C/B-A)	*(	FED*-
/B-A)	*(C	FED*-(
B-A)	*(/	FED*-(
-A)	*(/	FED*-(B
A)	*(-	FED*-(B/
)	*(-	FED*-(B/A
End	*	FED*-(B/A-
End	Empty	FED*-(B/A-*

Step 4: Reverse the postfix expression
FED*-(B/A-*

Prefix form = *-A/BC-*DEF