Infix_postfix Algorithm:

Input : Infix_expression, Stack, Postfix array
Output : Success/Failure

* - 42
(- 40

Psuedo code:

```
1. i <- 0, j <- 0
2. while (Infix_exp[i])
   {
        if (Infix_exp[i] = operand)
        {
            Postfix_exp[j] <- Infix_exp[i]
            j <- j + 1
        }
        else if (Infix_exp[i] = '(')
        {
            Push(s, Infix_exp[i])
        }
        else if(Infix_exp[i] = ')')
        {
            while (s -> stack[s -> top]  != '(')
            {
                Postfix_exp[j] <- Peek(s)
                Pop(s)
                j <- j + 1
            }
            Pop(s)
        }
        else
        {
            while (stk -> top != -1 && s -> stack[s -> top] != '(' &&
                        Priority(Infix_exp[i]) <= Priority(s -> stack[s -> top]))
            {
                Postfix_exp[j] <- Peek(s)
                Pop(s)
                j <- j + 1
            }
            Push(s, Infix_exp[i])
        }
        i <- i + 1
   }
4.  while (s -> top != -1)
    {
        Postfix_exp[j] <- Peek(s)
        Pop(s)
        j <- j + 1
    }
5.  Return SUCCESS
```

1 . char
2. int

pf("%d", '1'); => 49

'*'

Post_fix Evaluation :
Input : Postfix_exp, Stack
Output : result

a b * c - d e / f g + / +

2 3 * 3 - 8 4 / 1 1 + / +

Psuedo code:

```
1. i <- 0
2. while (Postfix_exp[i])
   {
        if (Postfix_exp[i] = operand)
        {
            Push(s, Postfix_exp[i] - 48)
        }
        else
        {
            Operand2 <- Peek(s)
            Pop(s)
            Operand1 <- Peek(s)
            Pop(s)
            Push(s, operand1 Postfix_exp[i] Operand2)
        }

        i <- i + 1
   }
3. result <- Peek(s)
4. Pop(s)
5. Return result
```

i = 13

2 = 50

50 - 48
2

int

stack

12

top = -1

int array

operand2 = 6
operand1 = 3

6 * 3 = 18 + 48 = '6'
3 - 6 = -3
8 / 4 = 2
1 + 1 = 2
2 / 2 = 1
3 + 1 = 4

result = 4

'6' - '0' = 6

'3' - '0' = 3

'6' - '0' = 2

3 * 6 = 18 -> '1''8'

6 * 2 => 12

infix_exp, Postfix_exp => string

implement stack using int array

Infix_prefix Algorithm:

Input : Infix_expression, Stack, Prefix array
Output : Success/Failure

Psuedo code:

```
1. i <- 0, j <- 0
2. while (Infix_exp[i])
   {
        if (Infix_exp[i] = operand)
        {
            Prefix_exp[j] <- Infix_exp[i]
            j <- j + 1
        }
        else if (Infix_exp[i] = ')')
        {
            Push(s, Infix_exp[i])
        }
        else if(Infix_exp[i] = '(')
        {
            while (s -> stack[s -> top] != ')')
            {
                Prefix_exp[j] <- Peek(s)
                Pop(s)
                j <- j + 1
            }
            Pop(s)
        }
        else
        {
            while (stk -> top != -1 && s -> stack[s -> top] != ')' &&
                        Priority(Infix_exp[i]) < Priority(s -> stack[s -> top]))
            {
                Postfix_exp[j] <- Peek(s)
                Pop(s)
                j <- j + 1
            }
            Push(s, Infix_exp[i])
        }
        i <- i + 1
   }
4. while (s -> top != -1)
   {
        Postfix_exp[j] <- Peek(s)
        Pop(s)
        j <- j + 1
   }
5. Return SUCCESS
```

Prefix Evaluation :
Input : Prefix_exp, Stack
Output : result

Psuedo code:

```
1. i <- 0
2. while (Pretfix_exp[i])
   {
        if (Prefix_exp[i] = operand)
        {
            Push(s, Prefix_exp[i])
        }
        else
        {
            Operand1 <- Peek(s)
            Pop(s)
            Operand2 <- Peek(s)
            Pop(2)
        }
        Push(s, operand2 Prefix_exp[i] Operand2)
        i <- i + 1
   }
3. result <- Peek(s)
4. Pop(s)
5. Return result
```

Infix expression : a + b * c

a + {bc*}
abc*+

a + b + c * d

T1=cd*
a + b + T1
T2=ab+

T2 + T1 => T2T1+

final postfix => ab+cd*+

Infix_expression : a * b - c + d / e / (f + g)

T1=fg+
a * b - c + d / e / T1
T2=ab*
T2 - c + d / e / T1
T3=de/
T2 - c + T3 / T1
T4=T3T1/ => de/fg+/

T2 - c + T4

T5=T2c- => ab*c-

T5 + T4 => T5T4+

Final exp = ab*c-de/fg+/+

a b * c - d e / f g + / +

```
   switch (opr)
   {
         case '+' :
         case '-':
               return 1;
         case '*':
         case '/':
               return 2;
         default :
               return 3;
   }
```

post_fix array

a * b - c + d / e / (f + g)

a b * c - d e / f g + / +

1. check operator or operand, if operand store in array
2. if operator check stack is empty, if empty push into stack
3. if operator in stack is having highest / same priority than operator which is in infix array, pop the operator from stack and store it in post fix array
4. check the status of stack and push the operator.
5. If any brakcets then push into stack
6. if u reach ')' then u need to pop all the operators from the stack until u read '('
7. If it is end of the array then pop all the operators from the stack and store it in postfix array

stack