Infix to Postfix Condition

① If the current character is a operand (a, b, A, 0) etc, Add it to output

② If the current character is a opening bracket (( ) - Add it to stack

③ If the current character is closing bracket ( )) - the pop item from the stack till opening bracket (( ) is Encountered

④ If the current character is a operator then

(i) Check if current character has greater precedence than the operator on top of stack

```
if precedence[cur_char] > precedence[top of stack]:
    # push cur_char to stack
    stack.append(cur_char)
```

(ii) If cur char has less or equal precedence than top of stack, then keep pop from stack Until the cur char has greater precedence from top of stack

```
while stack and precedence[cur_ch] <= precedence
                                      [top of stack]
    val = stack.pop()
    # append this poped value to op
    op.append(val)
```

Infix to Prefix Notation

Steps:

1) Reverse the Expression
2) Swap the paranthesis : while reversing the exp, Even the paranthesis gets reversed → '(' to ')' & ')' to '(' → so swap it back
3) Perform Infix to Postfix [In a Controlled Environment)
4) Reverse the output Expression

Infix to Postfix (Controlled Environment)

If the current character is a operator, we need to perform a check

① If cr_char is '^':

we will only pop if precedence of cr_char ('^') is less than or equal to top of stack.

i.e while stack and precedence('^') ≤ precedence(top of stack):
pop()

② For Any other operators (Excluding '^')

We will pop only if precedence of cr_char is less than precedence of top of stack.

i.e while stack and precedence(cr_char) < precedence(top of stack)

pop()

Postfix To Infix Operation

for an Infix Operation to be valid - We need to have a Operator in b/n 2 Operands → a+b

Logic

1) If the cur-char is a operand — Push to stack

2) If the cur-char is operator

(i) Pop top 2 operand from stack

```
ele-2 = stack.pop()
ele-1 = stack.pop()
```

(ii) Add operator b/n the 2 operand — wrap it in paranthe-sis

```
new-string = '(' + ele-1 + operator + ele-2 + ')'
```

(iii) Push back the new-string to stack

```
stack.append(new_string)
```

Output

After the iteration is over, the stack will have the final Infix Expression

Prefix to Infix

Similar to postfix to Infix, But we traverse in reverse direction.

Careful with poped element

① In Postfix to Infix:

Eg ab+, stack will have $\begin{bmatrix} b \\ a \end{bmatrix}$, b as first element and a as second element.

We want a+b, we want operator to be in b/n them so we pop

```
ele_2 = stack.pop()  # b
ele_1 = stack.pop()  # a
```

② In Prefix to Infix:

Eg +ab, stack will have $\begin{bmatrix} a \\ b \end{bmatrix}$, a as first element and b as second Element

We want a+b, so

```
ele_1 = stack.pop()  # a
ele_2 = stack.pop()  # b
```

Postfix To Prefix

Look close

* Prefix Operation = +ab

* Postfix Operation = ab+

Logic:

1) If the ur-char is operand, push to stack

2) If the ur-char is operator:

   ① Pop top 2 operand from stack

   ele_2 = stack.pop()

   ele_1 = stack.pop()

   ② Add the operator in the front

   new_string = ch + ele_1 + ele_2

   ③ Push the new-string back to the stack

   stack.append(new_string)

Output

After the iteration is Over, the stack will have the final Prefix Expression.

Prefix to Postfix

Similar to postfix to prefix, but we traverse in reverse direction

Careful with popped element

① In Postfix to Prefix:

Eg ab+, stack will have $\begin{bmatrix} b \\ a \end{bmatrix}$, b as first element & a as second element

We want ab+, so

```
ele_2 = stack.pop()  # b
ele_1 = stack.pop()  # a
```

② In Prefix to Postfix

Eg: +ab, stack will have $\begin{bmatrix} a \\ b \end{bmatrix}$, a as first element & b as second element

We want +ab, so

```
ele_1 = stack.pop()
ele_2 = stack.pop()
```