$Infix\ To\ Prefix\ Conversion$

$As\ ususal\ , push\ , pop\ etc. will\ be\ same\ as\ usual\ as\ we\ have\ in\ Postfix\ equation. That\ is\ stack\ operation.$

$More\ or\ less\ approach\ remains\ the\ same\ unless\ we\ come\ to\ prefix\ conversion.$

$Let\ infix\ expression\ be: (a+b)$

$if\ we\ see\ the\ postfix\ expression: ab+$

$But\ for\ prefix\ expression\ we\ have: +ab$

<u>$Now\ look\ at\ the\ approach$</u>:

$(a+b)$

$Now\ if\ we\ scan\ from\ last\ to\ first\ we\ get\ expression$:

$ba+$

$say\ a[0]=b, a[1]=a, and\ a[2]=+$

$Now, lets\ swap\ a[2]\ and\ a[0].$

$i.e.\ a[2]=temp(a\ variable)=+$

$a[2]=a[0]=b$

$a[0]=temp=+.$

$Now\ we\ will\ have +ab.$

Lets take a bigger expression: $(a+b)*(c+d)/e$

```
char *InToPre(char *infix)
{
    Stack st;
    string prefix;
    int len = strlen(infix);
          // Calculate the length of the string
    char *prefixExpr = (char *)malloc((len) *
sizeof(char)); // Allocate the memory for prefix
string
    create(&st, len);
           // Create the stack

    int i = len - 1, j = 0; // Start from the end of
infix expression

          ……..
}
```

len or length of the array of characters $= 13$.

And array proceeds from 0 ***to*** 12 ***and as we have to scan from from last to first , hence*** $i = 12$ ***and*** $j = 0$.

```
char *InToPre(char *infix)
{
    ….

 while (i >= 0)
    {
        if (isOperand(infix[i]) == 1)
        {
            prefixExpr[j] = infix[i];
            j++;
            i--;
        }

    …..
}
```

As the last element is e , hence isOperand(infix[12]) == 1 is true , hence prefix[0] = e. j = 0 + 1 = 1 and i = 12 − 1 = 11.

infix[11] =/ is an operator , hence isOperand()
will return 0.
Hence it will move to else part,

```
char *InToPre(char *infix)
{

   while (i >= 0)
    {
        if (isOperand(infix[i]) == 1)
        {…}
        else
        {
            if (pre(infix[i]) > pre(st.s[st.top]) ||
isEmpty(st) || infix[i] == ')')
            {
                push(&st, infix[i]);
                i--;
            }
         …..
}
```

$$pre\big(infix[i] = infix[11] =/= 2 > pre(top = -1)\big)\ is\ true.$$

$$And\ Stack\ is\ Empty\ , hence, push(/)\ , top = -1 + 1 = 0.$$

/	$Top \rightarrow 0$

$STACK$

$$i = 11 - 1 = 10.$$

```
char *InToPre(char *infix)
{
  …..
   while (i >= 0)
    {
        if (isOperand(infix[i]) == 1)
        {…}
        else
        {
            if (pre(infix[i]) > pre(st.s[st.top]) ||
isEmpty(st) || infix[i] == ')')
            {
                push(&st, infix[i]);
                i--;
            }
         …..
}
```

$$pre(infix[i] = infix[10] = \text{`)`}) = 0$$
$$> pre(s[top] = s[0] = \text{`/`}) = 2) \; is \; false.$$

$$And \, , isEmpty(st) is \; also \; false.$$

$$But, infix[10] = \text{`)`} \, , hence : we \; will \; push \; it \; to \; the \; stack.$$

$$top = 0 + 1 = 1.$$

)	$Top \rightarrow 1$
/	

$STACK$

$$i = 10 - 1 = 9.$$

Now we have $infix[9] =$ d which is an operand , hence it will return 1.

```
char *InToPre(char *infix)
{
  …..
   while (i >= 0)
    {

      if (isOperand(infix[i]) == 1)
        {
            prefixExpr[j] = infix[i];
            j++;
            i--;
        }
     ….
}
```

$prefix[j] = prefix[1] = d.$

$j = j + 1 = 1 + 1 = 2$

$i = i - 1 = 9 - 1 = 8.$

Hence till now converted expression we have $= ed$

Now $infix[8] = +$, hence it will get again pushed into the stack.

```
char *InToPre(char *infix)
{

   while (i >= 0)
    {
        if (isOperand(infix[i]) == 1)
        {…}
        else
        {
            if (pre(infix[i]) > pre(st.s[st.top]) ||
isEmpty(st) || infix[i] == ')')
            {
                push(&st, infix[i]);
                i--;
            }
         …..
}
```

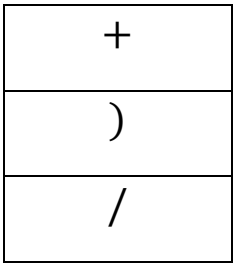
$$pre(infix[i]) = pre(infix[8]) = pre(` + `) = 1$$

$$> pre(s[top]) = pre(s[1]) = pre(`)`) = 0 \; which \; is \; true.$$

$$Though \; isEmpty(st) and \; infix[i] = `)` \; doesnot \; match$$

$$But \; as \; it \; is \; OR \; relation \; , if \; statement \; will \; run.$$

$$push(` + `) \; and \; top = top + 1 = 1 + 1 = 2.$$



STACK

$$i = i - 1 = 8 - 1 = 7$$

Now we have $infix[7] = c$, ***which is an operand , hence***

```
char *InToPre(char *infix)
{
  …..
   while (i >= 0)
    {

      if (isOperand(infix[i]) == 1)
        {
            prefixExpr[j] = infix[i];
            j++;
            i--;
        }
     ….
}
```

$prefix[j] = prefix[2] = infix[7] = c.$

$j = j + 1 = 2 + 1 = 3.$

$i = 7 - 1 = 7 - 1 = 6.$

Hence till now converted expression we have = edc

Now we will have `(` , opening brace, hence:

```
char *InToPre(char *infix)
{
 ….

  while (i >= 0)

    {
        if (isOperand(infix[i]) == 1)
        {…}
        else
          {
           if (pre(infix[i]) > pre(st.s[st.top]) ||
isEmpty(st) || infix[i] == ')')

            {….}

else if (pre(infix[i]) <= pre(st.s[st.top]) &&
infix[i] == '('
            {
                while (st.s[st.top] != ')')
                {
                    prefixExpr[j] = pop(&st);
                    j++;
                }
                pop(&st);
                i--;
            }

…..
}
```

$pre(infix[6] = `(`) = 0 \leq pre(s[2] = `+` = 1)$ and $infix[6] = `(`$, $hence$:

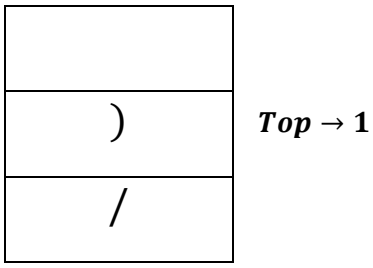
$Hence\ less\ than$ `)`, $it\ will\ pop\ out = `+`$.

$prefix[3] = pop() = +$

$top = top - 1 = 2 - 1 = 1$

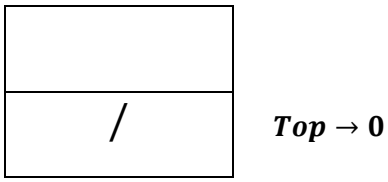
$Hence\ expression\ is\ now$: $ecd +$

$j = j + 1 = 3 + 1 = 4$.



$STACK$

Now, $the\ program\ will\ pop\ out$ `)`. $Top = 1 - 1 = 0$



$STACK$

$i = i - 1 = 6 - 1 = 5$.

$Now\ infix[i] = infix[5] = `*`$.

```
char *InToPre(char *infix)
{
 ….
  while (i >= 0)
    {
        if (isOperand(infix[i]) == 1)
        {…}
        else
          {
           if (pre(infix[i]) > pre(st.s[st.top]) ||
isEmpty(st) || infix[i] == ')')
            {….}

else if (pre(infix[i]) <= pre(st.s[st.top]) &&
infix[i] == '('
            {…}
           else
            {
                while (pre(infix[i]) <=
pre(st.s[st.top]) && !isEmpty(st) && infix[i] != '('
&& infix[i] != ')')
                {
                 prefixExpr[j] = pop(&st);
                 j++;
                }
                push(&st, infix[i]);
                i--;
            }
        }
    }
….
}
```

$Hence: pre(infix[5] = `*`) = 2 \leq pre(s[top] = s[0] = `/` = 2)$ $is\ true$.

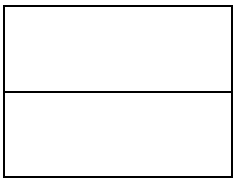
$!isEmpty(st)$ $is\ true\ and\ infix[5]$ $is\ not$ (and) ,

$hence$:

$prefix[j = 4\ ] = pop() = /.$

$hence\ , top = top - 1 = 0 - 1 = -1, i.e. Empty.$

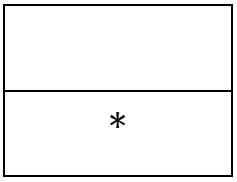
$j = j + 1 = 4 + 1 = 5.$



$Top \rightarrow -1$

$STACK$

$Now, push\ (`*`)$ $to\ the\ stack. Hence\ top\ becomes: -1 + 1 = 0.$



$Top \rightarrow 0$

$STACK$

$Therefore\ , now\ we\ have\ expression: edc +/$

$i = i - 1 = 5 - 1 = 4.$

Now why we have condition like :

```
while (pre(infix[i]) <= pre(st.s[st.top]) &&
!isEmpty(st) && st.s[st.top] != '(' && st.s[st.top] !=
')') {….}
```

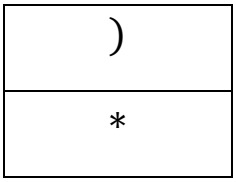
is discussed in Postfix Expression.

Next, we have $(a + b)$ expression.

1. 1*st push*(`)`) *parenthesis. Hence top* $= 0 + 1 = 1$.

```
char *InToPre(char *infix)
{

   while (i >= 0)
    {
        if (isOperand(infix[i]) == 1)
        {…}
        else
        {
            if (pre(infix[i]) > pre(st.s[st.top]) ||
isEmpty(st) || infix[i] == ')')
            {
                push(&st, infix[i]);
                i--;
            }
         …..
}
```



$Top \rightarrow 1$

$STACK$

$i = i - 1 = 4 - 1 = 3.$

2. ***Next we will input the element in the expression.***

```
if (isOperand(infix[i]) == 1)
        {
            prefixExpr[j] = infix[i];
            j++;
            i--;
        }
```

$prefix[j = 5] = infix[i = 3] = b.$

$The\ converted\ expression: edc + /b$

$j = j + 1 = 5 + 1 = 6.$

$i = i - 1 = 3 - 1 = 2.$

3. ***We have*** `` ` + ` `` ***and we will push*** `` (` + `) `` ***into the stack.***

```
if (pre(infix[i]) > pre(st.s[st.top]) || isEmpty(st)
|| infix[i] == ')')
            {

                push(&st, infix[i]);
                i--;
            }
```

$isEmpty(st) is\ false\ and\ infix[2] = `+`\ not\ `)`$

$hence\ false.$

$But,$

$pre(infix[2] = `+`) = 1 > pre(s[top] = s[1] = `)`)$

$= 0\ is\ true.$

$Hence,$

$push(infix[2] = `+`);\ Hence\ top = top + 1 = 1 + 1 = 2.$

+
)
*

$Top \rightarrow 2$

STACK

4. *We have `a` , put element in the expression.*

```
  if (isOperand(infix[i]) == 1)
        {
            prefixExpr[j] = infix[i];
            j++;
            i--;
        }
```

$$prefix[6] = infix[1] = a$$

$$j++ = j + 1 = 6 + 1 = 7$$

$$i-- = i - 1 = 1 - 1 = 0$$

$$Hence\ ,converted\ expression = \ edc+/ba$$

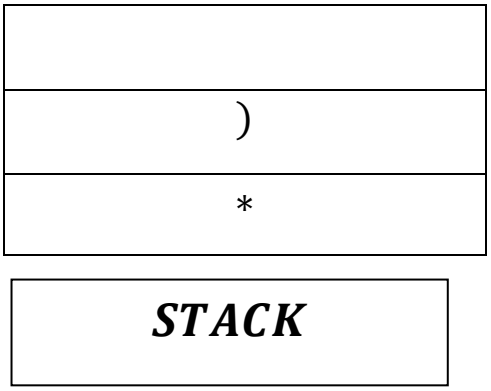
5. *Now as we have `(` , the things will get pop out till `)` .*

```
 else if (pre(infix[i]) <= pre(st.s[st.top]) &&
infix[i] == '(')
            {
                while (st.s[st.top] != ')')
                {
                    prefixExpr[j] = pop(&st);
                    j++;
                }
                pop(&st);
                i--;
            }
```

$pre(infix[i] = infix[0] = `(`) = 0 \leq pre(s[top] = s[2] = `+`) = 1$ *is true and* $infix[0] = `(`$ *,hence*:

$prefix[8] = pop() = +$ *and* $top = top - 1 = 2 - 1 = 1$

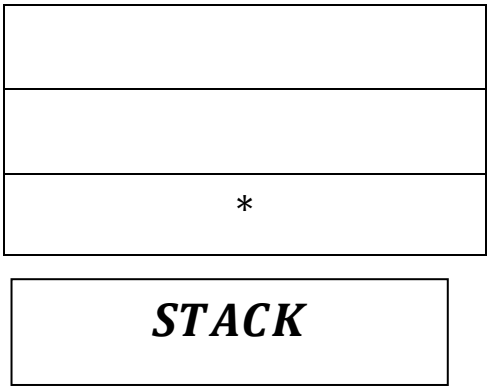
$j = j + 1 = 7 + 1 = 8$



$Top \rightarrow 1$

$Now, pop() =), top = top - 1 = 1 - 1 = 0$

$i = i - 1 = 0 - 1 = -1$



$Top \rightarrow 0$

Hence converted expression $= edc + /ba +$

Now what ever in the stack just pop() and input element in converted expression.

```
char *InToPre(char *infix)
{
 ….
  while (i >= 0)
    {
        if (isOperand(infix[i]) == 1)
        {…}
        else
          {
           if (pre(infix[i]) > pre(st.s[st.top]) ||
isEmpty(st) || infix[i] == ')')
            {….}

else if (pre(infix[i]) <= pre(st.s[st.top]) &&
infix[i] == '(')
            {…}
           else
            {
                while (pre(infix[i]) <=
pre(st.s[st.top]) && !isEmpty(st) && infix[i] != '('
&& infix[i] != ')')
                {…}
        }
    }
```

```
while (!isEmpty(st))
    {
        prefixExpr[j] = pop(&st);
        j++;
    }

    prefixExpr[j] = '\0';

…..

}
```

$$\boldsymbol{prefix[8] = pop() = *, top = top - 1 = 0 - 1 = -1(empty).}$$

$$\boldsymbol{Top \rightarrow -1}$$

STACK

$$\boldsymbol{Hence\ converted\ expression = edc + /ba + *}$$

$$\boldsymbol{now\ j = j + 1 = 8 + 1 = 9.}$$

$$\boldsymbol{And\ prefix[9] = '\backslash 0'\ i.e.\ end\ of\ character.}$$

Now the most important part:

Reverse or Swap the characters from the last to first to get the converted infix expression.

```
char *InToPre(char *infix)
{
 ….
  while (i >= 0)
    {
        if (isOperand(infix[i]) == 1)
        {…}
        else
          {
           if (pre(infix[i]) > pre(st.s[st.top]) ||
isEmpty(st) || infix[i] == ')')
            {….}

else if (pre(infix[i]) <= pre(st.s[st.top]) &&
infix[i] == '('
            {…}
           else
            {
                while (pre(infix[i]) <=
pre(st.s[st.top]) && !isEmpty(st) && infix[i] != '('
&& infix[i] != ')')
                {…}
        }
    }
```

```
while (!isEmpty(st))
    {
        prefixExpr[j] = pop(&st);
        j++;
    }

    prefixExpr[j] = '\0';

// Reverse the prefix expression


    int start = 0;
    int end = j - 1;
    while (start < end)
    {
        char temp = prefixExpr[start];
        prefixExpr[start] = prefixExpr[end];
        prefixExpr[end] = temp;
        start++;
        end--;
    }

    free(st.s);

    return prefixExpr;


}
```

$\boldsymbol{Start = 0\,, End = 9 - 1 = 8.}$

$\boldsymbol{From\ start\ to\ end, it\ will\ traverse\ for\ swapping.}$

$\boldsymbol{temp = prefix[0] = e.}$

$\boldsymbol{prefix[0] = prefix[8] = *.}$

$\boldsymbol{prefix[8] = temp = e.}$

$\boldsymbol{And\ this\ process\ continues\ to\ get\ the\ infix\ expression:}$

$$\boldsymbol{prefix\ expression : *+ab/+cde}$$

****

Algorithmic Analysis

After analysing the workings of the program we can generate the algorithmic structure :

Step 1: Repeat until each character in the infix notation is scanned from last to first.

- ***If*** `)` ***is encountered push it on the stack.***
- ***If an operand(whether a digit or a character)is encountered, add it to converted expression.***
- ***If a*** `(` ***is encoutered , then***:
  - ***Repeatedly pop from the stack and add it to the converted expression.***
  - ***Discard the***). ***That is, remove the*** `)` ***from the stack and donot add it to the converted expression.***
- ***If an operator is encountered and stack is not empty and top of the stack doesnot contain*** `)`:
  - ***a. Repeatedly pop from stack and add each operator (popped from the stack)to the converted expression which has the same precedence or a higher precedence*** .

- $b.$ ***Push the operator to the Stack.***

  - $[$***END OF IF***$]$

- ***Step*** $\mathbf{2}$: ***Repeatedly pop from the stack and add it to the Converted expression until the stack is empty.***

- ***Step*** $\mathbf{3}$: ***Add*** `\0` ***at the end of the expression.***

- ***Step*** $\mathbf{4}$: ***Reverse the Converted Expression to get prefix expression***

$[$***Reverse the Converted Expression***$]$

prefix$[\ ]$: ***is the array that contain converted expression.***
Len: ***Length of the converted expression*** .
Start: $\mathbf{0}$.
End: $\boldsymbol{Len - 1}$.

$\boldsymbol{While(Start < End)}$:
$\boldsymbol{temp := prefix[Start]}$
$\boldsymbol{prefix[Start] := prefix[End]}$
$\boldsymbol{prefix[End] := temp}$
$\boldsymbol{Start := Start + 1}$
$\boldsymbol{End := End - 1}$

- **Step 5: Return the prefix expression.**

Time Complexity Analysis

```
char *InToPre(char *infix)
{
    Stack st;→ O(1)
    string prefix;→ O(1)
    int len = strlen(infix); → O(N)
    char *prefixExpr = (char *)malloc((len) *
sizeof(char)); → O(N)
    create(&st, len);  → O(N)

    int i = len - 1, j = 0; → O(1)

    while (i >= 0)→ O(N)
    {
        if (isOperand(infix[i]) == 1)→ O(1)
        {
            prefixExpr[j] = infix[i];
            j++;
            i--;
        }
        else
        {
            if (pre(infix[i]) > pre(st.s[st.top]) ||
isEmpty(st) || infix[i] == ')')→ O(1)
            {
                cout << infix[i] << endl;

                push(&st, infix[i]);
                i--;
            }
```

```
            else if (pre(infix[i]) <=
pre(st.s[st.top]) && infix[i] == '(')
            {
                while (st.s[st.top] != ')')→ O(N)
                {
                    prefixExpr[j] = pop(&st);
                    j++;
                }
                pop(&st);
                i--;
            }
            else
            {
                while (pre(infix[i]) <=
pre(st.s[st.top]) && !isEmpty(st) && infix[i] != '('
&& infix[i] != ')')→ O(N)
                {

                    prefixExpr[j] = pop(&st);

                    j++;
                }
                push(&st, infix[i]);
                i--;
            }
        }
    }

    while (!isEmpty(st))→ O(N)
    {
        prefixExpr[j] = pop(&st);
        j++;
    }
```

```
    prefixExpr[j] = '\0';

    // Reverse the prefix expression to get the
correct order
    int start = 0;
    int end = j - 1;
    while (start < end)→ O(N)
    {
        char temp = prefixExpr[start];
        prefixExpr[start] = prefixExpr[end];
        prefixExpr[end] = temp;
        start++;
        end--;
    }

    free(st.s);

    return prefixExpr;
}
```

The time complexity analysis of the provided program can be done by considering the operations performed in each loop and conditional statement.

1.*The program starts by calculating the length of the input infix expression, which takes $O(N)$ time, where N is the length of the expression.*

2.*The program then allocates memory for the prefix expression using malloc, which takes $O(N)$ time.*

3. The program creates a stack and initializes it, which takes $O(N)$ time.

4. The program enters a while loop that iterates N times, where N is the length of the infix expression.

5. Inside the loop, there are several conditional statements and operations:

a. The program checks if the current character is an operand. This operation takes constant time.

b. If the current character is an operand, it is added to the prefix expression. This operation takes constant time.

c. If the current character is an operator, there are three possible scenarios:

i. If the precedence of the current operator is higher than the top operator in the stack, or if the stack is empty, or if the current operator is $'('$, the current operator is pushed onto the stack. These operations take constant time i.e. $O(1)$.

ii. If the precedence of the current operator is less than or equal to the top operator in the stack, and the current operator is $')'$, the program pops operators from the stack and adds them to the prefix expression until it encounters $'('$. These operations take at most $O(N)$ time in the worst case, as it may need to pop all the operators in the stack.

iii. For all other cases, the program pops operators from the stack and adds them to the prefix expression until the precedence of the current operator is higher than the top operator in the stack, or until the stack is empty, or

$\boldsymbol{until\ the\ current\ operator\ is\ '('\ or\ ')'.\ These\ operations\ take\ at\ most\ O(N)\ time\ in\ the\ worst\ case.}$

$\boldsymbol{6.\ After\ the\ while\ loop,\ the\ program\ pops\ the\ remaining\ operators\ from\ the\ stack\ and\ adds\ them\ to\ the\ prefix\ expression.}$

$\boldsymbol{This\ operation\ takes\ at\ most\ O(N)\ time\ in\ the\ worst\ case.}$

$\boldsymbol{7.\ The\ program\ then\ reverses\ the\ prefix\ expression\ using\ a\ while\ loop,\ which\ takes\ O(N)\ time.}$

$\boldsymbol{8.\ Finally,\ the\ program\ frees\ the\ memory\ allocated\ for\ the\ stack,\ which\ takes\ constant\ time.}$

$\boldsymbol{Overall,\ the\ time\ complexity\ of\ the\ program\ is\ O(N),\ where\ N\ is\ the\ length\ of\ the\ infix\ expression.}$

<u>In most simpliest words :</u>

$\boldsymbol{Time\ complexity\ of\ push\ and\ pop\ is : O(1)\ .}$

$\boldsymbol{We\ traverse\ infix\ characters\ from\ last\ to\ first\ takes: O(N).}$
$\boldsymbol{Poping\ out\ everything\ from\ stack\ and\ adding\ it\ to\ expression,\ takes\ O(N)time,}$
$\boldsymbol{and\ reversing\ of\ the\ expression\ takes\ O(N)time,\ hence:}$

$\boldsymbol{In\ a\ summary : O(1) + O(N) + O(N) + O(N) = O(N)\ time\ complexity.\ Hence\ Time\ Complexity\ is\ O(N).}$

$\boldsymbol{More\ specifically\ O(N),\ where\ N\ is\ the\ length\ of\ the\ infix\ expression.}$

$Operations$	$Time\ Complexity$
$isEmpty()$	$O(1)$
$isFull()$	$O(1)$
$isOperand()$	$O(1)$
$isPre()/isPriority()$	$O(1)$
$push()$	$O(1)$
$pop()$	$O(1)$
$infix - prefix$	$O(N)$

$Space\ Complexity$

$The\ program\ allocates\ memory\ for\ the\ prefix\ expression\ using\ malloc.The\ amount\ of\ memory\ allocated\ is\ (len) * sizeof(char), where\ len\ is\ the\ length\ of\ the\ infix\ expression.Therefore, the\ space\ complexity\ for\ the\ prefix\ expression\ is\ O(N), where\ N\ is\ the\ length\ of\ the\ infix\ expression.$

$The\ program\ creates\ a\ stack\ of\ size\ len\ using\ the\ create()function.The\ stack\ requires\ space\ to\ store\ the\ characters\ in\ the\ infix\ expression.$

$Therefore, the\ space\ complexity\ for\ the\ stack\ is\ also\ O(N).$

$The\ program\ uses\ a\ few\ additional\ variables\ such\ as\ len, i, j, start, end, and\ temp, which\ require\ constant\ space.$

And the push operation happens `N` times in stack taking $O(N)$ space complexity.

Where as the adding element to prefix expression and swapping of elements and reversing the expression takes a constant amount of space i.e. $O(1)$.

If we focus only on the space taken during the operation , then we only have the Push and Array operation i.e.

$$O(N) + O(1) = O(N) .$$

Hence space complexity is $O(N)$.