Infix to Postfix Conversion

To start with we all have functionaries:

```
typedef struct Stack
{
    int top;
    int size;
    char *s;

} Stack;
```

Creation of Stack

```
void create(Stack *st, int cap)
{
    st->size = cap;
    st->top = -1;
    st->s = (char *)malloc(st->size * sizeof(char));
}
```

Check IsEmpty of Stack

```
int isEmpty(Stack st)
{
    if (st.top == -1)
    {
        return 1;
    }
    return 0;
}
```

Checking Stack is full

```
int isFull(Stack st)
{
    if (st.top == st.size - 1)
    {
        return 1;
    }
    return 0;
}
```

Pushing Element in Stack

```
void push(Stack *st, char x)
{
    if (isFull(*st))
    {
        cout << "Stack Overflow" << endl;
    }
    else
    {
        st->top++;
        st->s[st->top] = x;
    }
}
```

Poping Element in Stack

```
char pop(Stack *st)
{
    if (isEmpty(*st))
    {
        cout << "Stack Underflow" << endl;
        return '\0';
    }
    return st->s[st->top--];
}
```

As we are dealing with characters , hence return will be a character.

$$\boldsymbol{Suppose\ we\ have\ arithmetic\ expression}:$$
$$(\boldsymbol{a} + \boldsymbol{b}) * (\boldsymbol{c}\hat{}\boldsymbol{d})/\boldsymbol{e}$$

1) ***At first it creates the stack.***

```
char *InToPost(char *infix)
{
    Stack st;
    int len = strlen(infix);
       // Calculate the length of the string
    char *postfix = (char *)malloc((len + 1) *
sizeof(char)); // Allocate the memory for postfix
string
    create(&st, len);
}
```

2) ***Now from priority checker it will return* 0 *for* (.**

```
int pre(char x)
{
  …….
    else if (x == '(' || x == ')')
    {
        return 0;
    }
  ….
}
```

3) *Now if its an operator and it must not come at first as we doing infix to postfix conversion.*

That is the expression is:

(a + b) ∗ (c^d)/e

And isOperand() function takes a character in its parameter:

```
int isOperand(char x)
{
    if (x == '+' || x == '-' || x == '*' || x == '/'
|| x == '(' || x == ')' || x == '^' || x == '%')
    {
        return 0;
    }
    return 1;
}
```

if not a character but an operator then return 0 *or false*,

otherwise 1 *or true*.

And as it returns 0 , *it will skip the if part* . *If will run*

when the function returns 1 (*true*):

```
char *InToPost(char *infix)
{
      …..
    while (infix[i] != '\0')
    {
        if (isOperand(infix[i])==1)//Skipped
        {
         …..
        }
     }
}
```

Hence it will now enter the else part:

```
char *InToPost(char *infix)
{
     int i = 0, j = 0;
     …..
    while (infix[i] != '\0')
    {
        if (isOperand(infix[i])==1)//Skipped
        {
         …..
        }
        else
        {
            if (pre(infix[i]) > pre(st.s[st.top]) ||
isEmpty(st) || infix[i] == '(')
            {
                push(&st, infix[i]);
                i++;
            }
```

```
                   ......• •



      }
}
```

$\boldsymbol{pre(infix[infix[0]]) = pre(`(`\,) = 0 > pre(st.s[st.top])}$
$\boldsymbol{= -1\ is\ true\ or\ isEmpty(st) = true\ also\ infix[0] = `(`\ is}$
$\boldsymbol{true.}$

$\boldsymbol{push \rightarrow `(`\,.\,And\ i++= i+1 = 0+1 = 1.}$

$\boldsymbol{And\ also\ now\ top = top+1 = \ -1+1 = 0.}$

<u>Stack</u>

(	$\boldsymbol{\rightarrow Top = 0}$

$\boldsymbol{4)\ Now\ we\ have\ character\ `a`}$

$\boldsymbol{i = 1\ and\ j = 0}$

```
char *InToPost(char *infix)
{
    while (infix[i] != '\0')
    {
        if (isOperand(infix[i])==1)
        {
            postfix[j] = infix[i];
            j++;
            i++;
        }
  …..
}
```

$\boldsymbol{Now\ character\ `a`\ will\ return\ 1\ (true)i.e.we\ have\ an\ operand\ and\ its\ type\ is\ character}:$

```
int isOperand(char x)
{
    if (x == '+' || x == '-' || x == '*' || x == '/'
|| x == '(' || x == ')' || x == '^' || x == '%')
    {
        return 0;
    }
    return 1;
}
```

$\boldsymbol{As\ we\ return\ 1\ if\ statements\ runs}:$

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

$\boldsymbol{postfix\ [0] = infix[1] = a}$

$\boldsymbol{j++ = j + 1 = 0 + 1 = 1}$

$\boldsymbol{i = i + 1 = 1 + 1 = 2}$

$\boldsymbol{Hence\ postfix\ formation: a}$

5) *we have operator* \` + \`

Hence now again the operand function will return 0.

```
int isOperand(char x)
{
    if (x == '+' || x == '-' || x == '*' || x == '/'
|| x == '(' || x == ')' || x == '^' || x == '%')
    {
        return 0;
    }
    return 1;
}
```

And then it runs the else part:

```
char *InToPost(char *infix)
{
    while (infix[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++;
            }
         ….
}
```

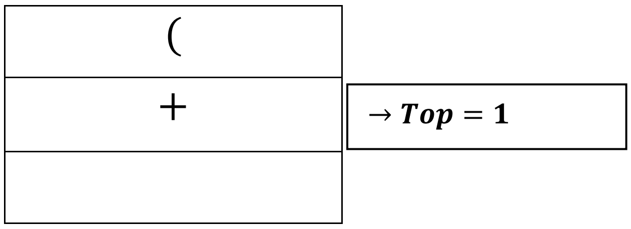
$\boldsymbol{pre(infix[infix[2]]) = pre(`+`) = 1 > pre(st.s[st.top])}$

$\boldsymbol{= pre(st.s[0]) = pre(`(`) = 0 \; is \; true \; .}$

$\boldsymbol{push \rightarrow `+` \; is \; true. \; And \; i++ = i+1 = 2+1 = 3.}$

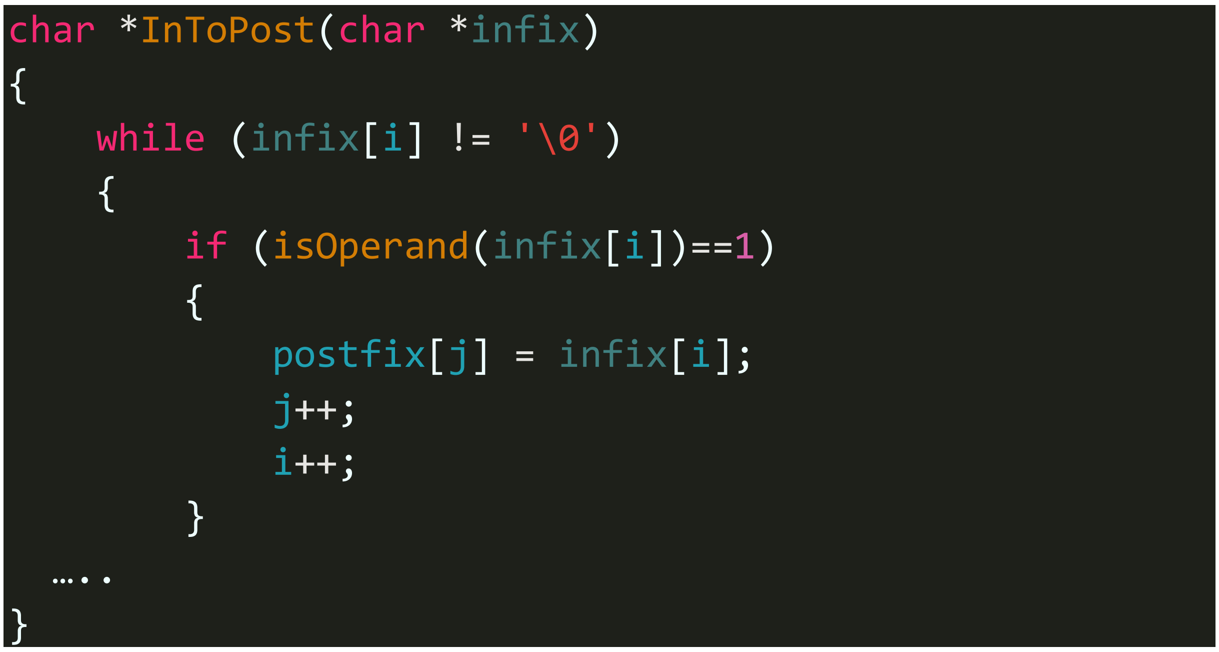
$\boldsymbol{And \; also \; now \; top = top + 1 = \; 0 + 1 = 1.}$

<u>Stack</u>



$\boldsymbol{6) \; we \; have \; operand \; `b`}$

$\boldsymbol{i = 3, j = 1}$



```
char *InToPost(char *infix)
{
    while (infix[i] != '\0')
    {
        if (isOperand(infix[i])==1)
        {
            postfix[j] = infix[i];
            j++;
            i++;
        }
  …. .
}
```

$\boldsymbol{Now \; character \; `b` \; will \; return \; 1 \; (true) i.e. we \; have \; an \; operand}$

and its type is character:

```c
int isOperand(char x)
{
    if (x == '+' || x == '-' || x == '*' || x == '/'
|| x == '(' || x == ')' || x == '^' || x == '%')
    {
        return 0;
    }
    return 1;
}
```

$$\boldsymbol{postfix\,[1] = infix[3] = b}$$

$$\boldsymbol{j++ = j+1 = 1+1 = 2}$$

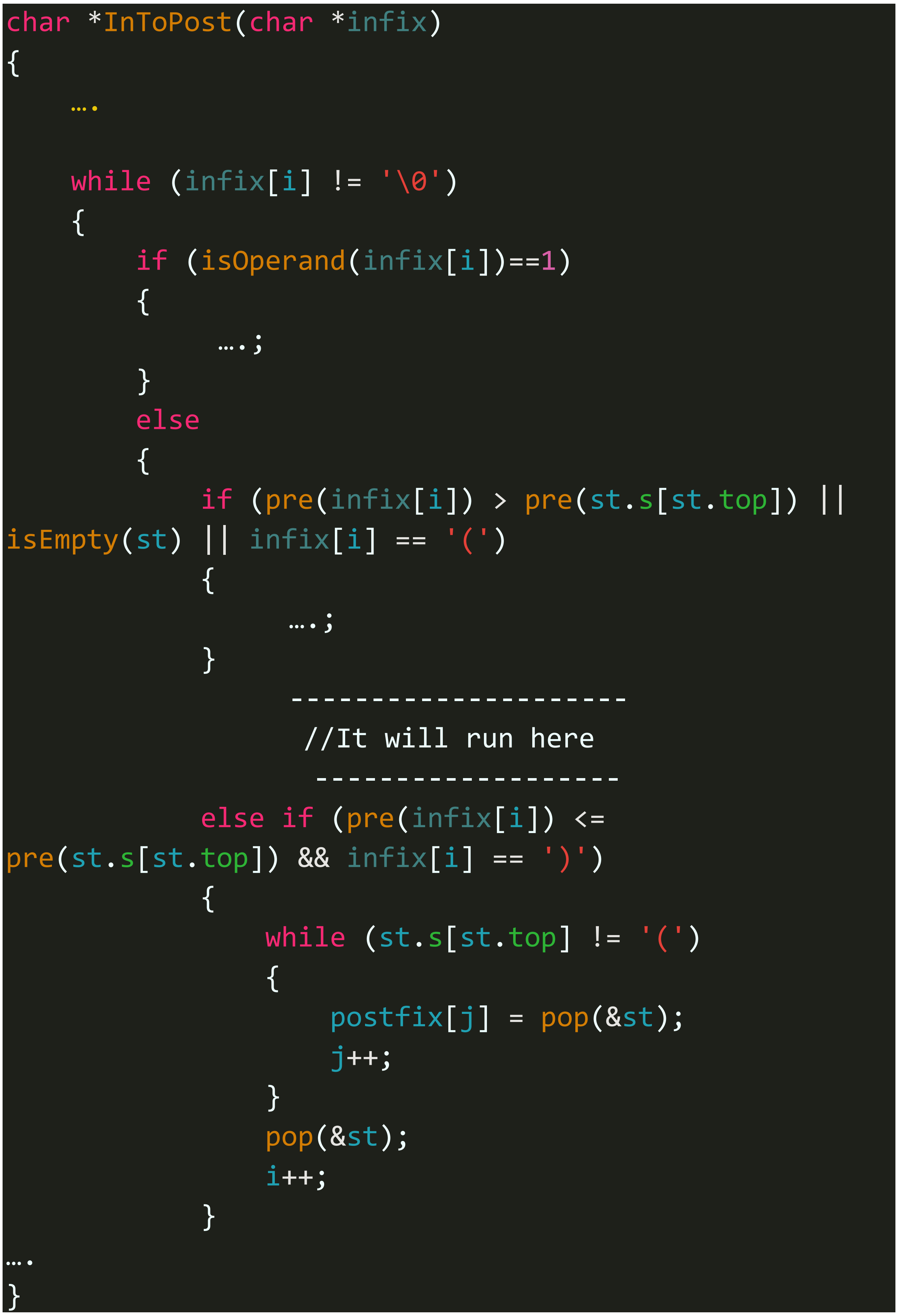
$$\boldsymbol{i = i+1 = 3+1 = 4}$$

Hence postfix formation: ab

7) *Now we have operator i.e.`)`*

Now logic is we will not push) rather we will pop out the element `+` in stack and again pop out the opening brace `)`.

Now 1st pop will continue less than `(` opening brace.



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

    while (infix[i] != '\0')
    {
        if (isOperand(infix[i])==1)
        {
             ….;
        }
        else
        {
            if (pre(infix[i]) > pre(st.s[st.top]) ||
isEmpty(st) || infix[i] == '(')
            {
                 ….;
            }
                 ---------------------
                  //It will run here
                  -------------------
            else if (pre(infix[i]) <=
pre(st.s[st.top]) && infix[i] == ')')
            {
                while (st.s[st.top] != '(')
                {
                    postfix[j] = pop(&st);
                    j++;
                }
                pop(&st);
                i++;
            }
….
}
```

$\boldsymbol{isOperand(infix[4] = `)`) = 0[False], hence\ else\ part\ runs.}$

$\boldsymbol{pre(infix[4] = `)`\ ) = 0 \leq pre(s[top] = s[1] = `+`)}$
$\boldsymbol{= 1\ is\ true\ and\ infix[4] = `)`}$

$\boldsymbol{is\ true.}$

$\boldsymbol{Then:}$

```
$\boldsymbol{While(top\ != [not\ equal\ to]\ '(')\{}$
    $\boldsymbol{postfix[2] = pop()\ = `+`;}$
    $\boldsymbol{j++;}$
$\}$
```

$\boldsymbol{j = j + 1 = 2 + 1 = 3\ and\ top = top - 1 = 1 - 1 = 0}$

Therefore, after pop function stack we have:

$\underline{\boldsymbol{Stack}}$

(	$\rightarrow \boldsymbol{Top = 0}$

$\boldsymbol{Now, again\ pop() \rightarrow `(`}$

$\boldsymbol{This\ pop\ function\ will\ make\ stake\ empty:}$

$\boldsymbol{top = top - 1 = 0 - 1 = -1}$

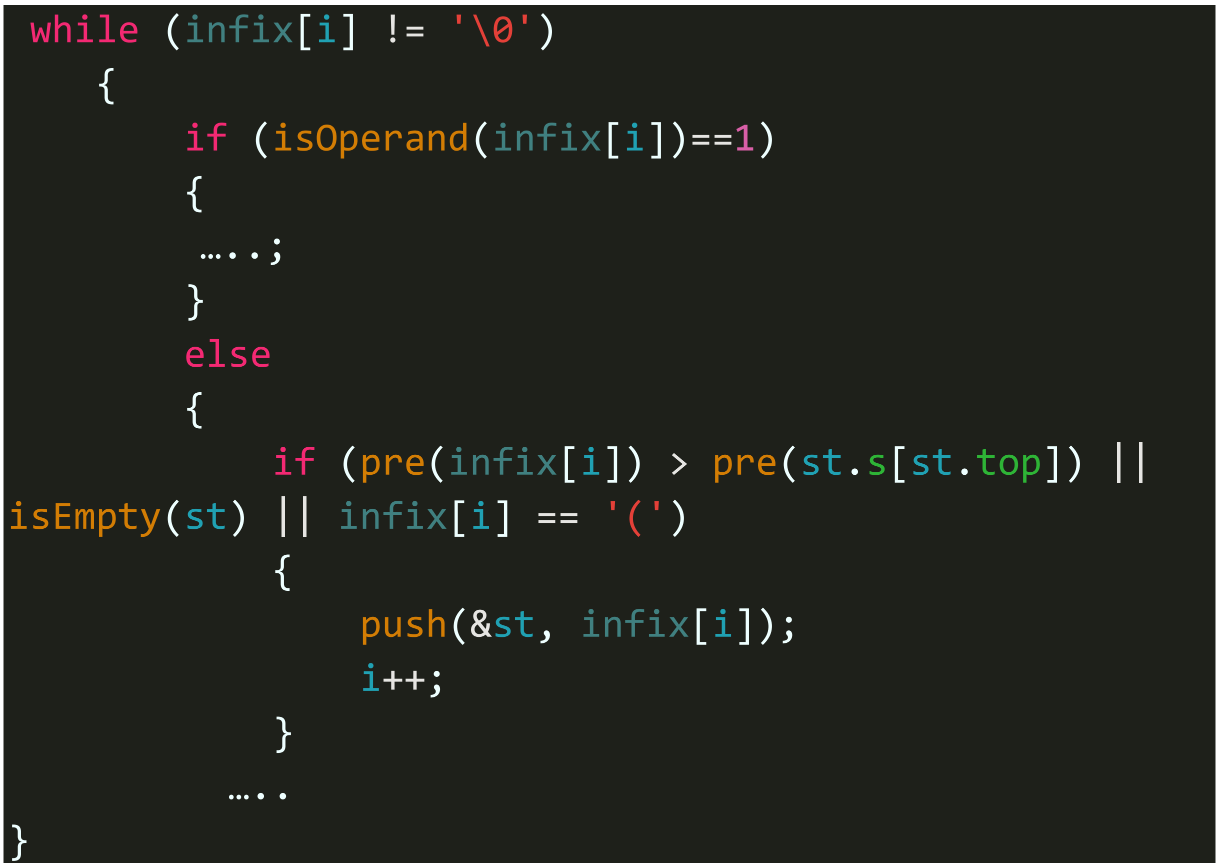
$\underline{\boldsymbol{Stack}}$

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

$\boldsymbol{And\ then\ i++ = \ i+1 \ = \ 4+1 \ = 5}$

$\boldsymbol{Hence\ postfix\ formation: ab +}$

$\boldsymbol{8)\ we\ have\ operator\ `*`}$



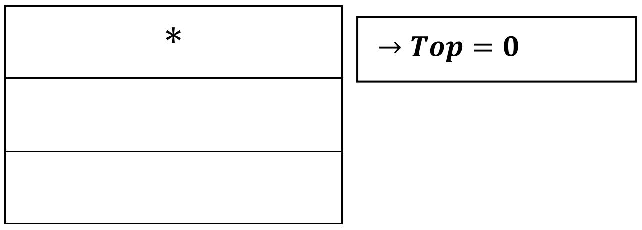
```
 while (infix[i] != '\0')
    {
        if (isOperand(infix[i])==1)
        {
         …..;
        }
        else
        {
            if (pre(infix[i]) > pre(st.s[st.top]) ||
isEmpty(st) || infix[i] == '(')
            {
                push(&st, infix[i]);
                i++;
            }
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

$\boldsymbol{isOperand(infix[5] = `*`) = 0[False], hence\ else\ part\ runs.}$

$\boldsymbol{prefix(infix[5] = `*`) = 2 > pre(st.s[st.top]) = \ -1}$

$\boldsymbol{push \rightarrow *}$

<u>Stack</u>



$top = top + 1 = -1 + 1 = 0$ ***and*** $i = i++ = 5 + 1 = 6$

9) ***Now we can relate how*** $(c\wedge d)$***will be processed.***

1***st it will push*** (

<u>Stack</u>

*	
(	$\rightarrow$ ***Top*** $= 1$

$top = top + 1 = 0 + 1 = 1$ ***and*** $i = i + 1 = 6 + 1 = 7.$

2***nd for c it will be put inside posfix array***:

$postfix[3] = infix[7] = c$

$i = i + 1 = 7 + 1 = 8$

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

Hence postfix formation: $ab + c$

3***rd for ^ , it will again will be pushed inside the stack***

<u>Stack</u>

*	
(	
^	$\rightarrow$ ***Top*** $= 2$

$top = top + 1 = 1 + 1 = 2$ ***and*** $i = i + 1 = 8 + 1 = 9.$

$4rth\ for\ d\ it\ will\ be\ put\ inside\ posfix\ array$:

$postfix[4] = infix[9] = d$

$i = i + 1 = 9 + 1 = 10$

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

$Hence\ postfix\ formation$: $ab + cd$

$5th\ for$ \`(\` :

$Everything\ will\ get\ popped\ out\ from\ stack\ less\ than$

\`(\` :

$postfix[j = 5] = pop()$ = \`^\` ;

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

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

<u>Stack</u>

$*$	
(	$\rightarrow Top = 1$

$And\ then$ \`(\` $will\ be\ popped\ out$

$top = top - 1 = 1 - 1 = 0$

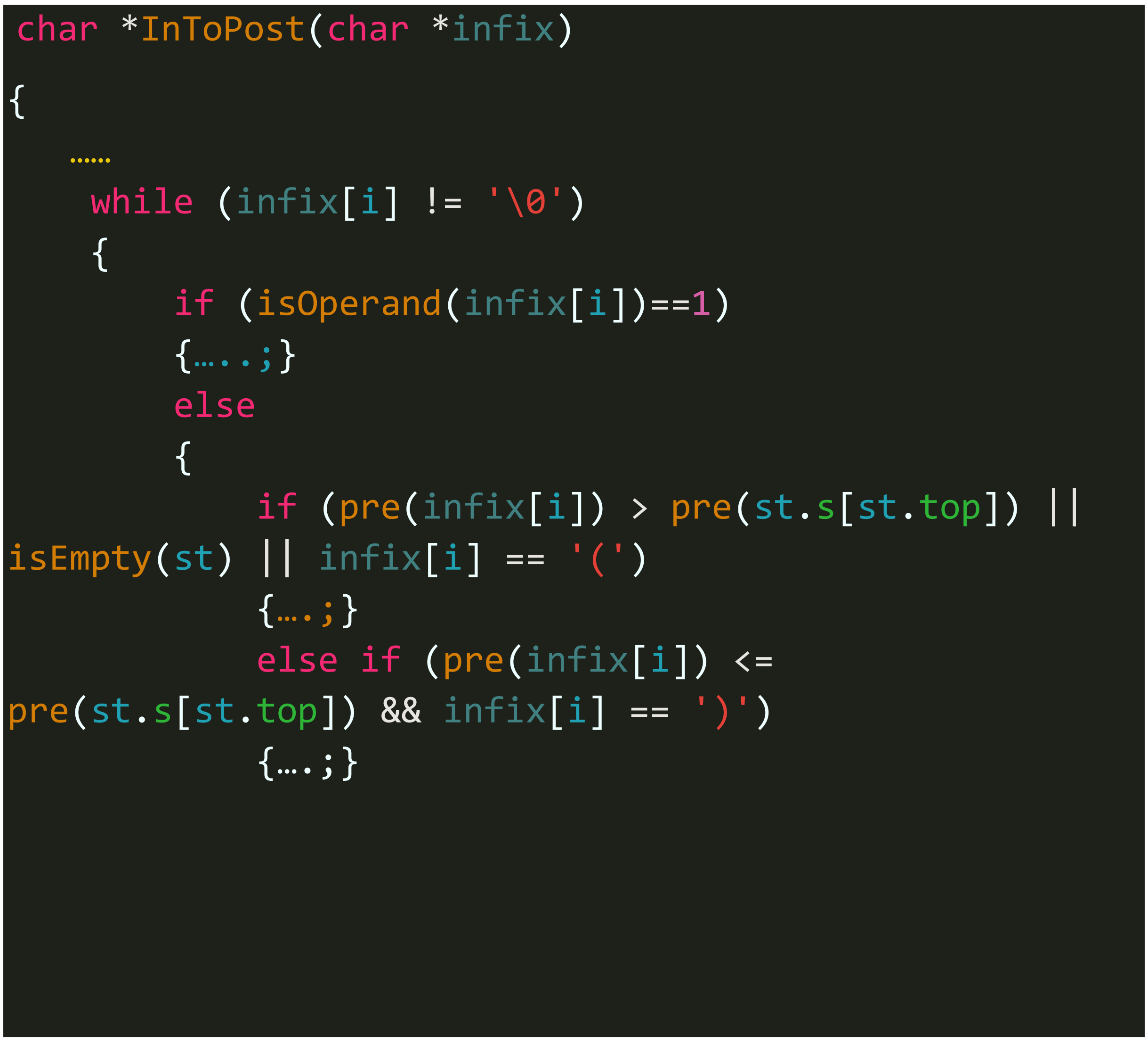
<u>Stack</u>

$*$	$\rightarrow Top = 0$

$i = i + 1 = 10 + 1 = 11$

Hence postfix formation: $ab + cd^\wedge$

10)*Now if we see the Stack* \` $*$ \` *is left to be popped out. Now there is no parenthesis left in the arithmetic expression. Hence, there must be a condition which will pop out the left over of stack and help in pushing the operator in stack when there is no parenthesis in the expression :*



```
char *InToPost(char *infix)
{
   ......
    while (infix[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] != ')')
                {

                    postfix[j] = pop(&st);
                    j++;
                }
                push(&st, infix[i]);
                i++;
            }
           ….;

}
```

$\boldsymbol{If\ you\ see\ the\ whole\ arithmetic\ expression:}$

$$\boldsymbol{(a+b)*(c\hat{\ }d)/e}$$

$\boldsymbol{Hence\ not\ only\ we\ have\ to\ pop * but\ also\ must\ push\ /\ .}$

$\boldsymbol{isOperand(infix[i] = infix[11] = /)}$
$\boldsymbol{\qquad\qquad = 0\ hence\ move\ to\ else\ part.}$

$\boldsymbol{pre(infix[i] = infix[11] = /) = 2\ \le\ pre(st.s[st.top]}$

$\boldsymbol{= st.s[0] = * = 2)\ = true}$

$\boldsymbol{!\,isEmpty(st) = true}$

$\boldsymbol{And\ infix\ expression\ doesnot\ contain\ (\ and\ )\ parenthesis.}$

$\boldsymbol{postfix[j = 6] = pop() \ =*}$

$\boldsymbol{j++= j+1 = 6+1 = 7}$

$\boldsymbol{Hence\ postfix\ formation: ab + cd\hat{}\ *}$

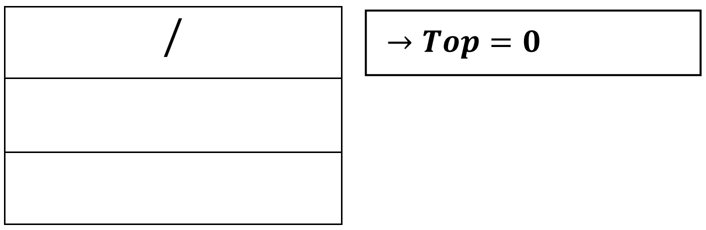
$\boldsymbol{top = top - 1 = 0 - 1 = -1\ i.e.empty}$

$\boldsymbol{Now\ coming\ out\ from\ the\ loop\ , we\ get:}$

$\boldsymbol{Push \rightarrow /\ \ into\ the\ stack.}$

$\boldsymbol{top = \ -1 + 1 = 0}$

$\underline{\boldsymbol{Stack}}$



$\boldsymbol{i = i++= i+1 = 11+1 = 12}$

$\boldsymbol{11)\ Now\ we\ have\ `e`\ left\ for\ assigning\ it\ to\ array.}$

```
char *InToPost(char *infix)
{
    while (infix[i] != '\0')
    {
        if (isOperand(infix[i])==1)
        {
            postfix[j] = infix[i];
            j++;
            i++;
        }
  .....
}
```

$\boldsymbol{Now\ isOperand(infix[i] = infix[12]) = 1(true) will\ enter}$

$\boldsymbol{if's\ body.}$

$\boldsymbol{postfix[j = 7] = infix\ [i = 12] = e}$

$\boldsymbol{Hence\ postfix\ formation: ab + cd\hat{}*e}$

$\boldsymbol{j = j + 1 = 7 + 1 = 8}$

$\boldsymbol{i = i + 1 = 12 + 1 = 13}$

$\boldsymbol{12)\ Now\ one\ element\ left\ in\ the\ stack\ to\ be\ popped\ out\ i.e./}$

```
 char *InToPost(char *infix)
{
   …..
    while (infix[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] != ')')
                {….}
            }
        }
    }

//What ever left in the stack will be popped out.

while (!isEmpty(st))
    {
        postfix[j] = pop(&st);

        j++;
    }

    postfix[j] = '\0';

    free(st.s);

    return postfix;
}
```

we have already finished elements in i to be compared:

i.e. what ever left in the stack now, will be popped out till its empty.

$Postfix[j = 8] = pop() = /$

$top = top - 1 = 0 - 1 = -1.$

i.e. stack is empty.

$j = j + 1 = 9$

And 9th element will have a null terminator '\0'.

It implies we have finished our post fix conversion or end of string. Null teminator marks the end of the string .

Hence, $postfix[j = 9] = '/0'$.

Hence postfix formation: $ab + cd\wedge * e/$

Now, free the stack memory $= free(st.s)$ *and return the post fix expression.*

Now, question may arise on:

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

                    postfix[j] = pop(&st);
                    j++;
                }
                push(&st, infix[i]);
                i++;
            }
```

***Now consider three arithmetic expression*:**

1. $(a+b)/(c-b)+e$
2. $(a+b)+(c-b)+e$
3. $(a+b)+(c-b)/e$

<u>***Consider 1st expression :***</u>

we know how $(a+b)$ ***and*** $(c-b)$ ***will be executed***

***The stack remains*:**

/	$\rightarrow Top = 0$

And we have $ab+cb-$ ***as half converted postfix expression.***

Now we will have a comparison between $+$ ***and*** $/$.

$$pre(infix[i=11]='+') = 1 \le pre(st[s.top]='/') = 2 \text{ is true.}$$

The above loop runs based on this condition(as this satisfy the requirement).

Consider 2nd expression :

we know how $(a + b)$and $(c - b)$will be executed

The stack remains:

$+$

$\rightarrow \boldsymbol{Top = 0}$

And we have $ab + cb -$ as halfconverted postfix expression.

Now we will have a comparison between $+$ and $/$.

$$pre(infix[i = 11] = '+') = 1 \leq pre(st[s.top] = '+') = 1 \text{ is true.}$$

The above loop runs based on this condition(as this satisfy the requirement)

Consider 3rd expression :

we know how $(a + b)$and $(c - b)$will be executed

The stack remains:

$+$	$\rightarrow Top = 0$

And we have $ab + cb -$ as half converted postfix expression.

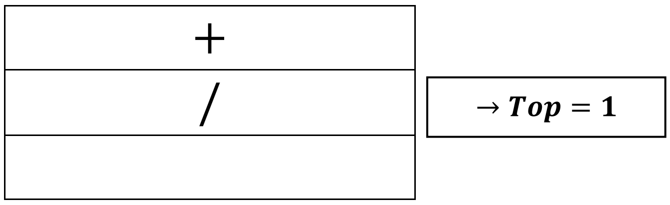
Now we will have a comparison between $+$ and $/$.

$$pre(infix[i = 11] = '/') = 2 \leq pre(st[s.top] = ' +') = 1 \text{ is false.}$$

What it will do is : it will push the next element in the stack :

```
if (pre(infix[i]) > pre(st.s[st.top]) || isEmpty(st)
|| infix[i] == '(')
            {
                push(&st, infix[i]);
                i++;
            }
```

Now stack will be :



After the postfix operation we will get:

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

$ab + cb - e$

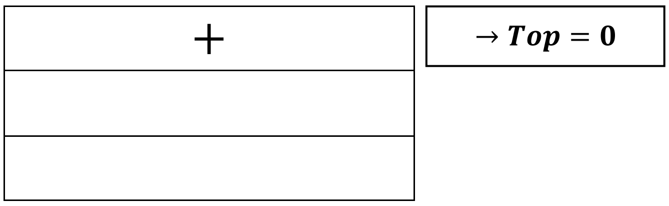
And then we have finished elements in i to be compared:

Hence everything will be popped out one by one:

1. $pop \rightarrow /$

```
while (!isEmpty(st))
    {
        postfix[j] = pop(&st);

        j++;
    }
```

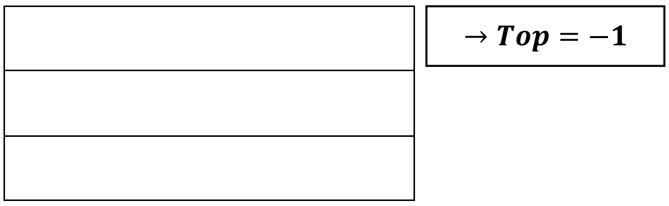


$\boldsymbol{Hence\ post\ fix\ expression\ now: ab + cb - e/}$

$\mathbf{2.}\ \boldsymbol{pop} \rightarrow\ +$

```
while (!isEmpty(st))
    {
        postfix[j] = pop(&st);

        j++;
    }
```



$\boldsymbol{Hence\ post\ fix\ expression\ now: ab + cb - e/+}$

Analysis of Algorithm

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

Infix To Postfix

Algorithm

Step 1: Repeat until each character in the infix notation is scanned

If a (is encountered, push it on the stack.
If an operand (whether a digit or a character) is encountered, add it postfix expression.
If a) is encountered, then:
a. Repeatedly pop from stack and add it to the postfix expression until a (is encountered.
b. Discard the (. That is, remove the (from stack and do not add it to the postfix expression.

If an operator is encountered and stack is not empty and infix expression doesnot contain: `(` and `)` , then:

a. Repeatedly pop from stack and add each operator

```
(popped from the stack) to the
postfix expression which has the same precedence or a higher
precedence than
b. Push the operator to the stack
[END OF IF]

Step 4: Repeatedly pop from the stack and add it to the postfix
expression until the stack is empty
Step 5: EXIT
```

Time Complexity Analysis

For push and pop we know it will be $O(1)$ *complexity* .

But for *postfix character addition in an array*.

We see each character is added 1 *time consuming* 1 *unit of time*:

```
postfix[j] = infix[i];
postfix[j] = pop(&st);
```

if there is no parenthesis it takes n *unit time to complete*.

if 1 *parenthesis then it takes* $n-1$ *unit of time to complete*.

if 2 *parenthesis then it takes* $n-2$ *unit of time to complete*.

if k *no. of parenthesis then it takes* $n-k$ *unit of time to complete, where* k *is constant* .

Also if we compare it with memory = postfifix takes a single unit of memory and runs upto n *times or* $n-k$ *times to generate a postfix expression*.

Hence post fix take $O(n)$*times complexity*.

$Operation$	$Time\ Complexity$
$isEmpty()$	$O(1)$
$isFull()$	$O(1)$
$isOperand()$	$O(1)$
$ispre()/\ ispriority()$	$O(1)$
$push()$	$O(1)$
$pop()$	$O(1)$
$infix - postfix$	$O(n)$

*****In Addition*****

- **1.** ***Push and Pop takes $O(1)$ i.e. each at a constant amount of time . But on a full scan of the infix expression from first to last takes $O(n)$ times.***

- **2.** ***Calculating the length of the infix string, strlen(infix) function takes $O(n)$ times complexity.***

- **3.** ***create(&st, len) , where len has 'n' size i.e. creating a stack of 'n' size will take $O(n)$ time complexity.***

- **4.** ***If we notice Push and Pop and adding of elements in postfix string runs 'n' times helps the loop while loop to scan the length of the infix expression from first to last 'n' times. Hence the time complexity is $O(n)$ where n is the length of the infix expression.***

Space Complexity Analysis

if we have k no. of characters and in which we have n number of operators.

We know when we add anything in array it takes $O(1)$*,*
i.e. same unit of space used to input data in the auxiliary space created for array.

Where as n times the stack is used for push operation consuming n space complexity.
Hence if we sum up then we get: $O(1) + O(n) = O(n)$ *space complexity.*