

Postfix Evaluation

In, Postfix evaluation , the conversion from infix to post fix evaluation will remain same.

Now if we consider a infix expression :

$$\rightarrow (a + b)^{(c - d)}/e$$

can be observed through a stack implementation table:

Stack	Input	Output
Empty	$(a + b)^{(c - d)}/e$	Nothing
($a + b)^{(c - d)}/e$	Nothing
($+b)^{(c - d)}/e$	a
+($b)^{(c - d)}/e$	a
+($)^{(c - d)}/e$	ab
+	$^{(c - d)}/e$	ab
^	$(c - d)/e$	ab +
(^	$c - d)/e$	ab +
(^	$-d)/e$	ab + c
-(^	$d)/e$	ab + c
-(^	$)/e$	ab + cd
^	$/e$	ab + cd -
/	e	ab + cd - ^
/	Empty	ab + cd - ^e
Empty	Empty	ab + cd - ^e/

Similarly, if we have a infix notation:

$$(2 + 3) * (2 + 3)/5$$

Now we get the converted postfix expression as:

$$\rightarrow 23 + 23 + * 5 /$$

Table Generated as Follows:

<i>Stack</i>	<i>Input</i>	<i>Output</i>
<i>Empty</i>	$(2 + 3) * (2 + 3)/5$	<i>Nothing</i>
($2 + 3) * (2 + 3)/5$	<i>Nothing</i>
($+3) * (2 + 3)/5$	2
+($3) * (2 + 3)/5$	2
+($) * (2 + 3)/5$	23
+	$* (2 + 3)/5$	23
*	$(2 + 3)/5$	23 +
(*	$2 + 3)/5$	23 +
(*	$+3)/e$	23 + 2
+(*	$3)/5$	23 + 2
+(*	$)/5$	23 + 23
*	$/5$	23 + 23 +
/	5	23 + 23 + *
/	<i>Empty</i>	23 + 23 + * 5
<i>Empty</i>	<i>Empty</i>	23 + 23 + * 5 /

*Now what will be the result : $5 * 5 / 5 = \frac{25}{5} = 5$*

Lets evaluate it through stack :

Push (2),

Push(3),

Now we get + hence : $Add(2, 3) = 2 + 3 = 5$

Like what it happens in Stack in memory:

2
3
<i>Data Register: +</i>

Then it pop out , 2 and 3 and send $ADD(2, 3)$ to Processor to process.

*Similarly we have the postfix expression: $23 + 23 + * 5 /$*

if it is not operator i. e. operand Push it to the stack.

```

int postfixEvaluation(char *postfix){

    Stack st;
    create(&st, strlen(postfix));

    int i = 0;
    int x1, x2, r;

    while (postfix[i] != '\0')
    {
        if (isOperand(postfix[i]) == 1)
        {
            push(&st, postfix[i] - '0');
        }
        else
        {
            .....
        }
        i++;
    }
    ...
}

```

1. *The above function takes the postfix expression.*
2. *Create another stack again after postfix conversion,*
3. *Till the last character = '\0', the characters will be taken inside the loop.*

Note: Also the character moves from `0` to `9` i.e.:

```
while (postfix[i] != '\0' && postfix[i] >= '0' &&
postfix[i] <= '9')
{
    ... .
}
```

First is 2 and 3 , both are operand and both will get pushed inside Stack.

ASCII value of 0 is 48 . Now we see that we will push integer values not characters, hence 0 is 48 , 1 is 49 , 2 is 50 and so on.

If we do : $50 - 48$ i. e. (ASCII of 2 – ASCII of 0) will be 2. Similarly, $51 - 48$ i. e. (ASCII of 3 – ASCII of 0) will be 3.

Hence,

1. push($2 - 0 = 2$)

2. push ($3 - 0 = 3$)

3
2

STACK

4. Now we have `+` operator, hence we pop out 3 first and then 2 and it adds the two operands and after the result we get ,push it into the stack.

```
int postfixEvaluation(char *postfix){
    ..... •
    while (postfix[i] != '\0')
    {
        ..... •
        else
        {
            x2 = pop(&st);
            x1 = pop(&st);

            switch (postfix[i])
            {
                case '+':
                    r = x1 + x2;
                    break;
                case '-':
                    r = x1 - x2;
                    break;
                case '*':
                    r = x1 * x2;
                    break;
                case '/':
                    r = x1 / x2;
                    break;
                case '^':
                    r = x1 ^ x2;
                    break;
                case '%':
```

```

        r = x1 % x2;
        break;
    }
    push(&st, r);
}
i++;
}
.....;
}

```

2 + 3 = 5 and we push(5) into the stack.

5

STACK

5.The process continues i. e.push 3 and 2 into the stack,

3
2
5

STACK

Then pop(2, 3) from stack and Add(2, 3) = 5 and

push(5) into the stack.

5
5

STACK

6. Now we get `` operator , hence both operands i. e.
5 and 5 will get pop out from the stack and get multiplied
with output = 25.*

The output 25 will get pushed into the stack.

25

STACK

7. Now we get operand 5 and push 5.

5
25

STACK

Now, we get operator `/` and therefore pop out 25 and 5.

Divide : $\frac{25}{5} = 5$ and push (5) into the stack.

5

STACK

8. Now, pop out 5 as output:

```
int postfixEvaluation(char *postfix){
    ..... •
    while (postfix[i] != '\0')
    {
        ..... •
        else
        {
            ..... •
            {
                ..... •
            }
            push(&st, r);
        }
        i++;
    }
    return pop(&st);
}
```

This process is known as Postfix Evaluation.

Time Complexity of PostFix Evaluation

```
int postfixEvaluation(char *postfix){  
  
    Stack st;  $\rightarrow O(1)$   
    create(&st, strlen(postfix));  $\rightarrow O(n)$   
  
    int i = 0;  $\rightarrow O(1)$   
    int x1, x2, r;  $\rightarrow O(1)$   
  
    while (postfix[i] != '\0')  $\rightarrow O(n)$   
    {  
        if (isOperand(postfix[i]) == 1)  $\rightarrow O(1)$   
        {  
            push(&st, postfix[i] - '\0');  
        }  
        else  
        {  
            x2 = pop(&st);  $\rightarrow O(1)$   
            x1 = pop(&st);  $\rightarrow O(1)$   
  
            switch (postfix[i])  $\rightarrow O(1)$   
            {  
                case '+':  
                    r = x1 + x2;  
                    break;  
                case '-':  
                    r = x1 - x2;  
                    break;  
                case '*':  
                    r = x1 * x2;
```

```

        break;
    case '/':
        r = x1 / x2;
        break;
    case '^':
        r = x1 ^ x2;
        break;
    case '%':
        r = x1 % x2;
        break;
    }
    push(&st, r);  $\rightarrow O(1)$ 
}
i++;
}
return pop(&st);  $\rightarrow O(1)$ 
}

```

Analysing the above code , we can say that :

1. Creating object of stack `Stack st` takes $O(1)$ time.

2. Creation of Stack according to the Postfix expression : `create(&st, strlen(Postfix))` takes $O(n)$ time, Where n is the length of the expression..

3. `int i = 0;` takes $O(1)$ time.

4. Declaration of variable `int x1, x2, r` also takes $O(1)$ time.

5. While loop help to traverse the postfix expression till $n - 1$ times as last character is `\0` and traversal is less than `\0` .

6. Push and Pop occurs $O(1)$ time at each operation. Total push pop occurs $O(n)$ as we scan the postfix expression from first to last.

7. Performing arithmetic operations: The switch statement performs arithmetic operations such as addition, subtraction, multiplication, division, exponentiation, and modulo. These operations also take constant time, $O(1)$ time at each operation. As at each specific switch , each case runs only one time. When it runs upto the length of the expression it takes $O(n)$ time. That is cases operates $O(n)$ times in switch-case.

8. And returning the pop operation also takes $O(1)$ time.

Hence, $O(1) + O(n) + O(1) + O(1) + O(n - 1) + O(1) = O(n)$ time complexity.

$O(n)$, where 'n' is the length of the postfix string.

Space Complexity

Push operation in stack takes $O(n - 1) = O(n)$ complexity. Hence Space Complexity = $O(n)$ complexity.