What is a Stack?

A Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle. Stack has one end, whereas the Queue has two ends (**front and rear**). It contains only one pointer **top pointer** pointing to the topmost element of the stack. Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack. In other words, a ***stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.***

Some key points related to stack

* It is called as stack because it behaves like a real-world stack, piles of books, etc.
* A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.
* It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.

Working of Stack

Stack works on the LIFO pattern. As we can observe in the below figure there are five memory blocks in the stack; therefore, the size of the stack is 5.

Suppose we want to store the elements in a stack and let's assume that stack is empty. We have taken the stack of size 5 as shown below in which we are pushing the elements one by one until the stack becomes full.



Since our stack is full as the size of the stack is 5. In the above cases, we can observe that it goes from the top to the bottom when we were entering the new element in the stack. The stack gets filled up from the bottom to the top.

When we perform the delete operation on the stack, there is only one way for entry and exit as the other end is closed. It follows the LIFO pattern, which means that the value entered first will be removed last. In the above case, the value 5 is entered first, so it will be removed only after the deletion of all the other elements.

Standard Stack Operations

**The following are some common operations implemented on the stack:**

* **push():** When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
* **pop():** When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
* **isEmpty():** It determines whether the stack is empty or not.
* **isFull():** It determines whether the stack is full or not.'
* **peek():** It returns the element at the given position.
* **count():** It returns the total number of elements available in a stack.
* **change():** It changes the element at the given position.
* **display():** It prints all the elements available in the stack.

PUSH operation

**The steps involved in the PUSH operation is given below:**

* Before inserting an element in a stack, we check whether the stack is full.
* If we try to insert the element in a stack, and the stack is full, then the ***overflow*** condition occurs.
* When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
* When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., **top=top+1,** and the element will be placed at the new position of the **top**.
* The elements will be inserted until we reach the ***max*** size of the stack.



POP operation

**The steps involved in the POP operation is given below:**

* Before deleting the element from the stack, we check whether the stack is empty.
* If we try to delete the element from the empty stack, then the ***underflow*** condition occurs.
* If the stack is not empty, we first access the element which is pointed by the ***top***
* Once the pop operation is performed, the top is decremented by 1, i.e., **top=top-1**.



# Infix, Postfix and Prefix

Infix, Postfix and Prefix notations are three different but equivalent ways of writing expressions. It is easiest to demonstrate the differences by looking at examples of operators that take two operands.

Infix notation: X + Y

Operators are written in-between their operands. This is the usual way we write expressions. An expression such as A \* ( B + C ) / D is usually taken to mean something like: "First add B and C together, then multiply the result by A, then divide by D to give the final answer."

Infix notation needs extra information to make the order of evaluation of the operators clear: rules built into the language about operator precedence and associativity, and brackets ( ) to allow users to override these rules. For example, the usual rules for associativity say that we perform operations from left to right, so the multiplication by A is assumed to come before the division by D. Similarly, the usual rules for precedence say that we perform multiplication and division before we perform addition and subtraction. (see [CS2121 lecture](https://www.cs.man.ac.uk/~pjj/cs212/ho/node3.html#SECTION00031000000000000000)).

Postfix notation (also known as "Reverse Polish notation"): X Y +

Operators are written after their operands. The infix expression given above is equivalent to A B C + \* D /
The order of evaluation of operators is always left-to-right, and brackets cannot be used to change this order. Because the "+" is to the left of the "\*" in the example above, the addition must be performed before the multiplication.
Operators act on values immediately to the left of them. For example, the "+" above uses the "B" and "C". We can add (totally unnecessary) brackets to make this explicit:
( (A (B C +) \*) D /)
Thus, the "\*" uses the two values immediately preceding: "A", and the result of the addition. Similarly, the "/" uses the result of the multiplication and the "D".

Prefix notation (also known as "Polish notation"): + X Y

Operators are written before their operands. The expressions given above are equivalent to / \* A + B C D
As for Postfix, operators are evaluated left-to-right and brackets are superfluous. Operators act on the two nearest values on the right. I have again added (totally unnecessary) brackets to make this clear:
(/ (\* A (+ B C) ) D)

Although Prefix "operators are evaluated left-to-right", they use values to their right, and if these values themselves involve computations then this changes the order that the operators have to be evaluated in. In the example above, although the division is the first operator on the left, it acts on the result of the multiplication, and so the multiplication has to happen before the division (and similarly the addition has to happen before the multiplication).
Because Postfix operators use values to their left, any values involving computations will already have been calculated as we go left-to-right, and so the order of evaluation of the operators is not disrupted in the same way as in Prefix expressions.

In all three versions, the operands occur in the same order, and just the operators have to be moved to keep the meaning correct. (This is particularly important for asymmetric operators like subtraction and division: A - B does not mean the same as B - A; the former is equivalent to A B - or - A B, the latter to B A - or - B A).

Examples:

|  |  |  |  |
| --- | --- | --- | --- |
| **Infix** | **Postfix** | **Prefix** | **Notes** |
| A \* B + C / D | A B \* C D / + | + \* A B / C D | multiply A and B,divide C by D,add the results |
| A \* (B + C) / D | A B C + \* D / | / \* A + B C D | add B and C,multiply by A,divide by D |
| A \* (B + C / D) | A B C D / + \* | \* A + B / C D | divide C by D,add B,multiply by A |

Evaluation of Expressions: (A + B) \* C - (D - E) \* (F + G) convert into Infix, Prefix and Postfix Expressions

(10 mark Questions)

Evaluate the Expression using Stack:

In expression evaluation problem, we have given a[string](https://www.tutorialcup.com/interview/string) s of length n representing an expression that may consist of integers, balanced parentheses, and binary operations ( +, -, \*, / ). Evaluate the expression. An expression can be in any one of prefix, infix, or postfix [notation](https://www.tutorialcup.com/interview/array/infix-to-postfix.htm).

## Example

See few examples for expression evaluation:

**Input :**s = “100 \* ( 2 + 12 )”

**Output :**1400

## Algorithm for Expression Evaluation

Now we know the problem statement for expression evaluation. So, without wasting our time we move towards [algorithm](https://www.tutorialcup.com/interview/algorithm) uses for the solution of expression evaluation.

1. Initialize a string s of length n consisting of expression.
2. Create one [stack](https://www.tutorialcup.com/interview/stack) to store values and other to store operators.
3. Traverse through the string and check if the current character is a white space continue the loop. Else if it is an opening parenthesis push it in a stack of operators.
4. Else if the current character is a digit. Initialize a integer val as 0. Traverse from the current position + 1 till the end of the string while the current character is a digit and update the val as val \* 10 + current digit. Push it in the stack of values.
5. Else if it is a closing parenthesis, traverse while the stack of operators is not empty and current character in it is not an opening parenthesis.
6. Pop the top 2 digits from the stack of values and an operator from operator stack. Perform the arithmetic operation and push the result in a stack of values.
7. While the operator’s stack is not empty, pop the top 2 digits from the stack of values and an operator from operator stack. Perform the arithmetic operation and push the result in a stack of values.
8. Return the top of the stack of values.

## C++ Program for Expression Evaluation

#include <bits/stdc++.h>

using namespace std;

int precedence**(**char op**){**

if**(**op == '+'||op == '-'**)**

return 1;

if**(**op == '\*'||op == '/'**)**

return 2;

return 0;

**}**

int applyOp**(**int a, int b, char op**){**

switch**(**op**){**

case '+':

return a + b;

case '-':

return a - b;

case '\*':

return a \* b;

case '/':

return a / b;

**}**

**}**

int evaluate**(**string tokens**){**

int i;

stack **<**int**>** values;

stack **<**char**>** ops;

for**(**i = 0; i **<** tokens.length**()**; i++**){**

if**(**tokens**[**i**]** == ' '**)**

continue;

else if**(**tokens**[**i**]** == '('**){**

ops.push**(**tokens**[**i**])**;

**}**

else if**(**isdigit**(**tokens**[**i**])){**

int val = 0;

while**(**i **<** tokens.length**()** && isdigit**(**tokens**[**i**])){**

val = **(**val\*10**)** + **(**tokens**[**i**]**-'0'**)**;

i++;

**}**

values.push**(**val**)**;

**}**

else if**(**tokens**[**i**]** == ')'**){**

while**(**!ops.empty**()** && ops.top**()** != '('**){**

int val2 = values.top**()**;

values.pop**()**;

int val1 = values.top**()**;

values.pop**()**;

char op = ops.top**()**;

ops.pop**()**;

values.push**(**applyOp**(**val1, val2, op**))**;

**}**

if**(**!ops.empty**())**

ops.pop**()**;

**}**

else**{**

while**(**!ops.empty**()** && precedence**(**ops.top**())** **>**= precedence**(**tokens**[**i**])){**

int val2 = values.top**()**;

values.pop**()**;

int val1 = values.top**()**;

values.pop**()**;

char op = ops.top**()**;

ops.pop**()**;

values.push**(**applyOp**(**val1, val2, op**))**;

**}**

ops.push**(**tokens**[**i**])**;

**}**

**}**

while**(**!ops.empty**()){**

int val2 = values.top**()**;

values.pop**()**;

int val1 = values.top**()**;

values.pop**()**;

char op = ops.top**()**;

ops.pop**()**;

values.push**(**applyOp**(**val1, val2, op**))**;

**}**

return values.top**()**;

**}**

int main**(){**

cout **<<** evaluate**(**"100 \* ( 2 + 12 )"**)** **<<** endl;

return 0;

**}**

1400

Applications of Stack

**The following are the applications of the stack:**

* **Balancing of symbols:** Stack is used for balancing a symbol. For example, we have the following program:
1. **int** main()
2. {
3. cout<<"Hello";
4. cout<<"javaTpoint";
5. }

As we know, each program has *an opening* and *closing* braces; when the opening braces come, we push the braces in a stack, and when the closing braces appear, we pop the opening braces from the stack. Therefore, the net value comes out to be zero. If any symbol is left in the stack, it means that some syntax occurs in a program.

* **String reversal:** Stack is also used for reversing a string. For example, we want to reverse a "**javaTpoint**" string, so we can achieve this with the help of a stack.
First, we push all the characters of the string in a stack until we reach the null character.
After pushing all the characters, we start taking out the character one by one until we reach the bottom of the stack.
* **UNDO/REDO:** It can also be used for performing UNDO/REDO operations. For example, we have an editor in which we write 'a', then 'b', and then 'c'; therefore, the text written in an editor is abc. So, there are three states, a, ab, and abc, which are stored in a stack. There would be two stacks in which one stack shows UNDO state, and the other shows REDO state.
If we want to perform UNDO operation, and want to achieve 'ab' state, then we implement pop operation.
* **Recursion:** The recursion means that the function is calling itself again. To maintain the previous states, the compiler creates a system stack in which all the previous records of the function are maintained.
* **DFS(Depth First Search):** This search is implemented on a Graph, and Graph uses the stack data structure.
* **Backtracking:** Suppose we have to create a path to solve a maze problem. If we are moving in a particular path, and we realize that we come on the wrong way. In order to come at the beginning of the path to create a new path, we have to use the stack data structure.
* **Expression conversion:** Stack can also be used for expression conversion. This is one of the most important applications of stack. The list of the expression conversion is given below:
* Infix to prefix
* Infix to postfix
* Prefix to infix
* Prefix to postfix

Postfix to infix

* **Memory management:** The stack manages the memory. The memory is assigned in the contiguous memory blocks. The memory is known as stack memory as all the variables are assigned in a function call stack memory. The memory size assigned to the program is known to the compiler. When the function is created, all its variables are assigned in the stack memory. When the function completed its execution, all the variables assigned in the stack are released.

We can easily represent queue by using linear arrays. There are two variables i.e. front and rear, that are implemented in the case of every queue. Front and rear variables point to the position from where insertions and deletions are performed in a queue. Initially, the value of front and queue is -1 which represents an empty queue. Array representation of a queue containing 5 elements along with the respective values of front and rear, is shown in the following figure.



The above figure shows the queue of characters forming the English word **"HELLO"**. Since, No deletion is performed in the queue till now, therefore the value of front remains -1 . However, the value of rear increases by one every time an insertion is performed in the queue. After inserting an element into the queue shown in the above figure, the queue will look something like following. The value of rear will become 5 while the value of front remains same.



After deleting an element, the value of front will increase from -1 to 0. however, the queue will look something like following.



## Algorithm to insert any element in a queue

Check if the queue is already full by comparing rear to max - 1. if so, then return an overflow error.

If the item is to be inserted as the first element in the list, in that case set the value of front and rear to 0 and insert the element at the rear end.

Otherwise keep increasing the value of rear and insert each element one by one having rear as the index.

## Algorithm

* **Step 1:** IF REAR = MAX - 1
Write OVERFLOW
Go to step
[END OF IF]
* **Step 2:** IF FRONT = -1 and REAR = -1
SET FRONT = REAR = 0
ELSE
SET REAR = REAR + 1
[END OF IF]
* **Step 3:** Set QUEUE[REAR] = NUM
* **Step 4:** EXIT

## C Function

1. **void** insert (**int** queue[], **int** max, **int** front, **int** rear, **int** item)
2. {
3. **if** (rear + 1 == max)
4. {
5. printf("overflow");
6. }
7. **else**
8. {
9. **if**(front == -1 && rear == -1)
10. {
11. front = 0;
12. rear = 0;
13. }
14. **else**
15. {
16. rear = rear + 1;
17. }
18. queue[rear]=item;
19. }
20. }

## Algorithm to delete an element from the queue

If, the value of front is -1 or value of front is greater than rear , write an underflow message and exit.

Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time.

## Algorithm

* **Step 1:** IF FRONT = -1 or FRONT > REAR
Write UNDERFLOW
ELSE
SET VAL = QUEUE[FRONT]
SET FRONT = FRONT + 1
[END OF IF]
* **Step 2:** EXIT

## C Function

1. **int** delete (**int** queue[], **int** max, **int** front, **int** rear)
2. {
3. **int** y;
4. **if** (front == -1 || front > rear)
5.
6. {
7. printf("underflow");
8. }
9. **else**
10. {
11. y = queue[front];
12. **if**(front == rear)
13. {
14. front = rear = -1;
15. **else**
16. front = front + 1;
17.
18. }
19. **return** y;
20. }
21. }

**Application of Queue in Data Structure**

* Managing requests on a single shared resource such as CPU scheduling and disk scheduling.
* Handling hardware or real-time systems interrupts.
* Handling website traffic.
* Routers and switches in networking.
* Maintaining the playlist in media players
* **Difference between stack and queue**

