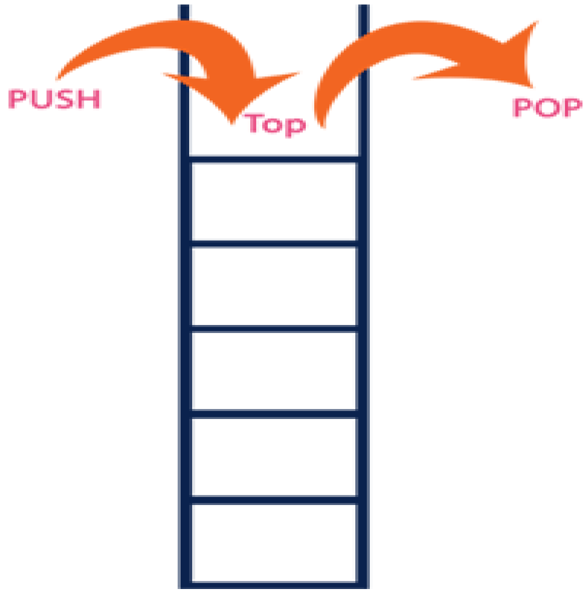
Stacks

- Stack is a linear data structure in which the insertion and deletion operations are performed at only one end.
- In a stack, adding and removing of elements are performed at a single position which is known as "**top**”.
- That means, a new element is added at top of the stack and an element is removed from the top of the stack.
- Inserting an element into stack at top is said to be ***PUSH*** operation.
- Deleting element from top of the stack is said to be ***POP*** operation.
- In stack, the insertion and deletion operations are performed based on **LIFO** (Last In First Out) principle or **FILO** (First in Last Out)



Stack

Standard Stack Operations

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.

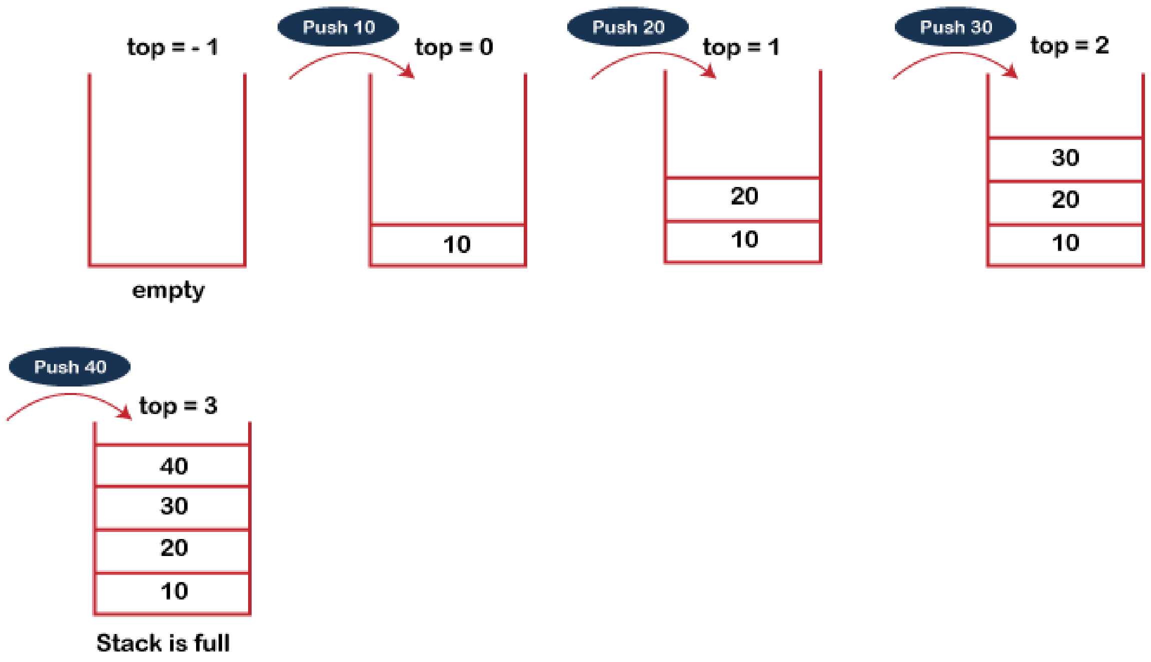
change(): It changes the element at the given position.

display(): It prints all the elements available in the stack.

Push operation:

The steps involved in the PUSH operation are given below:

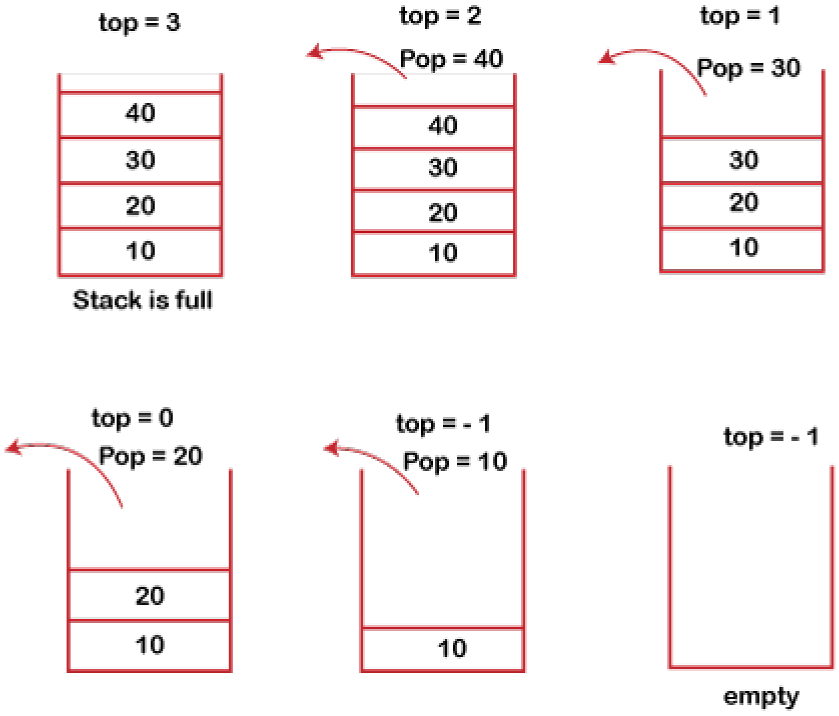
- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the ***overflow*** condition occurs.
- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- When the new element 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 are 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**.



Stack operations:

- Standard operations

Push: To insert an element at the top of stack

Pop: To remove element from the top of stack

Display (user defined): To display the elements of stack

Peak (user defined): Returns the top of the element

Applications of stack:

- Reversing a list
- Parentheses checking

- Conversion of an infix expression into a postfix expression
- Evaluation of a postfix expression
- Conversion of an infix expression into a prefix expression
- Evaluation of a prefix expression
- Recursion
- DFS technique

Implementation of stack:

- Simple Array based Implementation.
- Linked list Implementation.

Simple Array based Implementation:

Push operation:

push() is a function used to insert an element into the stack.

The new element is always inserted at **top** position.

Step 1 - Check whether **stack** is **FULL**. (**top = = SIZE-1** or not)

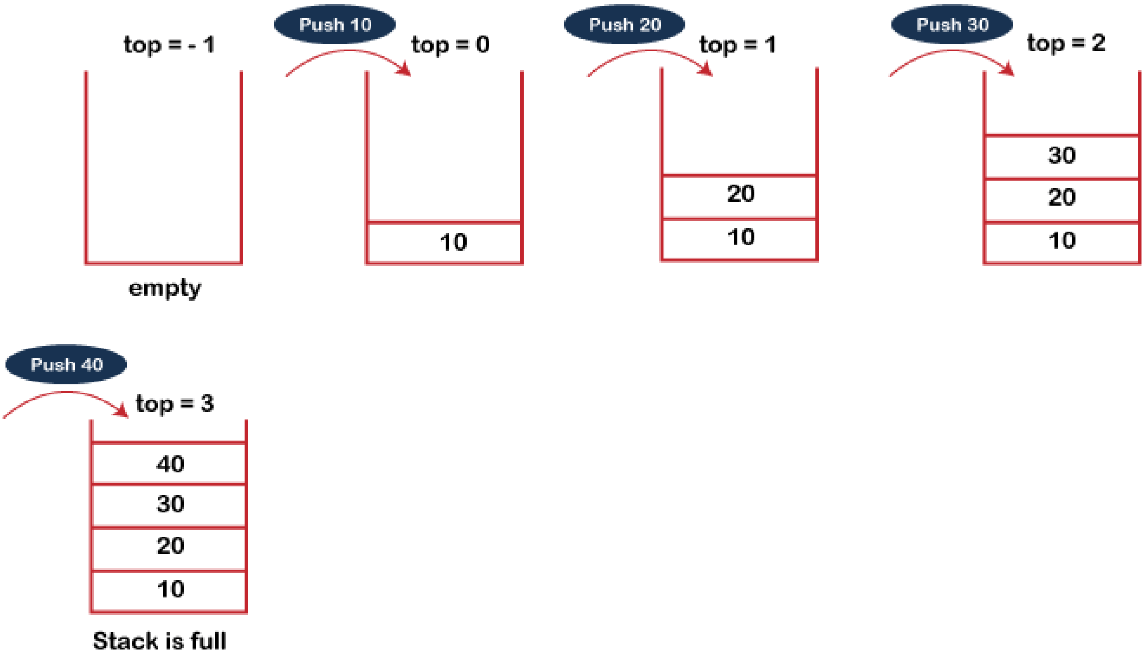
Step 2 - If stack is **FULL**, then display **"Stack OVERFLOW!!! Insertion is not possible!!!"** and terminate the function.

Step 3 - If stack is **NOT FULL**, then increment **top** value by one (**top++**) and set stack[top] to value (**stack[top] = value**).

```
Step - 1 :   IF top = Max-1
                 Display "OVERFLOW"
                  Go to step-4
Step – 2:    top++
Step – 3:    stack[top] = value
Step - 4:     End
```

Source code:

```
#define MAX 4   //size of the stack

int stack[MAX];

int top=-1;      //initially top=-1 ,on inserting one element top is incremented by 1

push(int *stack, int top, int ele)   //inserting an element into the top of the stack

   {

 if(top==MAX-1)

{ printf("\n Stack Overflow");

        }

 else

      {  top=top+1;

  stack[top]=ele;

      }

    }
```



Pop operation:

pop() is a function used to delete an element from the stack.

In a stack, the element is always deleted from **top** position.

Visiting element from the stack is possible by calling pop().

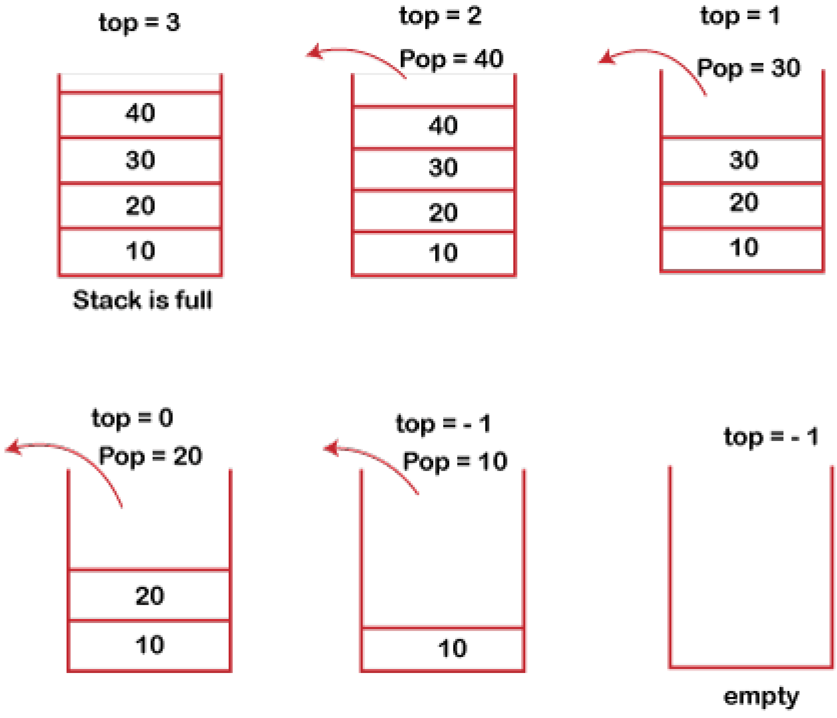
We can use the following steps to pop an element from the stack...

- **Step 1 -** Check whether **stack** is **EMPTY**. (**top = = -1**)
- **Step 2 -** If it is **EMPTY**, then display **“UNDERFLOW! Deletion is not possible!"** and terminate the function.
- **Step 3 -** If it is **NOT EMPTY**, then delete **stack[top]** and decrement **top** value by one (**top--**).

```
Step - 1: IF top = -1
        Display “UNDERFLOW”
        Go to step-5
Step – 2: x = stack[top]
Step – 3: top--
Step – 4: return x
```

Source code:

```
 int pop(int *stack, int top)
{ int val;
   if(top = = -1)
     printf(“\n underflow\n”);
   else
   {  val=stack[top];
       top=top-1;
return val;
    }
}
```



Display operation:

```
void display()

{

 if(top == -1)                    // if top==-1 then stack is empty

       printf("\n Stack Underflow");

   else

   { int  i;

     printf("\n Stack elements are:\n");

      for(i=top;i>=0;i--)                                 //stack    elements    are    displayed
         printf("%d  ", stack[i]); }

 }
```

Finding peek element or top element:

```
int peak()
{    int val;
     if (top==-1)  printf("\n Stack Underflow");
     else   //displaying top element
       {    val=pop();
              push(val);
              return val;
       }}
```

Operations	Complexities
Push	O(1)
Pop	O(1)
Displaying all elements	O(n)
Finding peek element	O(1)

Implementation of STACK using Linked Lists:

- The push operation is the insertion of node into the linked list at the beginning.
- The pop operation is the deletion of node from the beginning (the header/ top node)

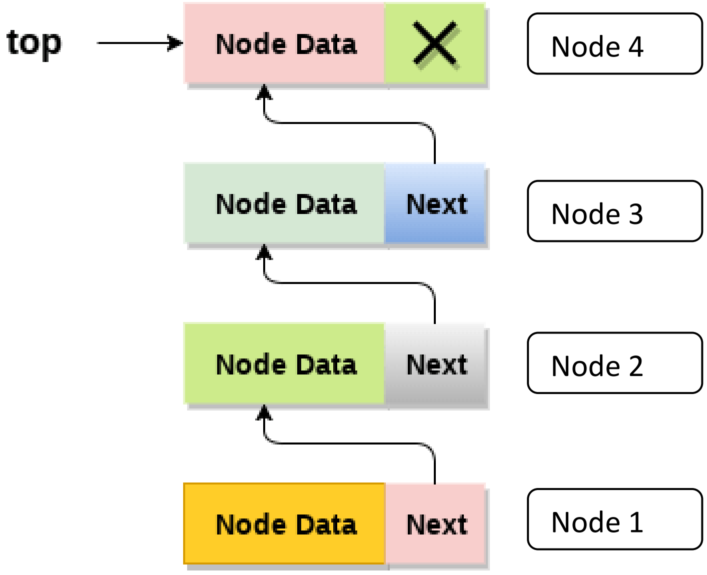
Push operation:

Insertion of new element to stack is known as *push operation* in stack. We can push elements at top of stack.

Step by step descriptive logic to push elements in stack:

1. Check stack overflow, i.e. if (size >= CAPACITY), then print "Stack overflow" error message. Otherwise move to below step.
2. Create a new stack node using dynamic memory allocation i.e. struct stack * newNode = (struct stack *) malloc(sizeof(struct stack);
3. Assign data to the newly created node using newNode->data = data;.
4. Link new node with the current stack top most element.

    Say newNode->next = top; and increment size count by 1.
5. Finally make sure the top of stack should always be the new node i.e. top = newNode;.



Stack

Node 1: It is the first inserted node. It will be popped last

Node 4: It is the last inserted node. So will be popped first

Source code:

```
void push(STACK *top,  int value)
{     STACK *new_node;
       new_node = (STACK *) malloc (sizeof(STACK));
       if(new_node == NULL)
          printf("\n Stack Overflow … “);
       else
       { new_node → data = value;  new_node→next = NULL;
          if(top == NULL) // This is the first node
          top = new_node;
          else     // There exist some nodes
           {   new_node →next = top;
                top = new_node;
                printf("\n Insertion success!!!");
          }
 }
```

Pop operation:

Removal of top most element from stack is known as *pop operation* in stack.

Step by step descriptive logic to pop elements from stack.

1. If size <= 0 then throw "Stack is Empty" error, otherwise move to below step.
2. Assign the top most element reference to some temporary variable, say struct stack *topNode = top;. Similarly copy data of stack top element to some variable say int data = top->data;
3. Make second element of stack as top element i.e. top = top->next;.
4. Delete the top most element from memory using free(topNode);.
5. Decrement stack size by one and return data.

Source code:

```
int pop(STACK *top)

{

    int element;

    STACK *t;

    if(top  == NULL)

          printf("\n Stack Underflow”);

    else

     {

         element = top→data;

         t = top;

         top = top →next;

         free(t);
         return element;

     }

 }
```

//top most element is removed first

(Time complexities are same as that of array implementation)

Parenthesis checking

- {((((([])))))} // Check the parenthesis are balanced or not
- if char is ‘(‘ or ‘{‘ or ‘[‘ then we push it onto the top of stack.

- If char is ‘)’ or ‘}’ or ‘]’ then we perform pop operation and if the popped character is the matches with the starting bracket then fine otherwise parenthesis are not balanced and terminate the task.
- If char is the end of string and if stack is empty, then it is balanced.
- In other cases, it is said to imbalanced.

Notations – infix, prefix and postfix

- Any arithematic expression consists of operators and operands.
- The way we write the arithmetic expression is called notation;
- There are three different notations used to write the athematic expression.
    1. Infix Expression
    2. Prefix Expression
    3. Postfix expression
- We can convert the expression in one notation to another notation.

Infix notation:

- An expression is said to be in infix notation if the operators in the expression are placed in between the operands on which the operator works.
- For example - a + b * c

In the above example the operator is * is placed between the operands on which this operator works, here ***b*** and ***c.***

- Infix expressions are easy for humans to read, write and understand, but it is not the case for computing devices.
- It is costly (in terms of space and time) to process the infix expressions in algorithms.

Postfix notation:

- An expression is said to be in prefix notation if the operators in the expression are placed before the operands on which the operator works.
- For example - +a*bc
- In the above example the operator is * is placed before the operands on which this operator works, here ***b*** and ***c,*** similarly the + is placed before ***a*** and the result of ***(b*c)***.
- Prefix notation is also called as Polish notation.

Prefix notation:

- An expression is said to be in postfix notation if the operators in the expression are placed after the operands on which the operator works.
- For example - abc*+
- In the above example the operator is * is placed after the operands on which this operator works, here ***b*** and ***c,*** similarly the + is placed after ***a*** and the result of ***(b*c)***.
- Postfix notation is also called as Reverse Polish notation.
- Widely used notation for evaluating the expressions.

Evaluation of expressions:

When evaluating the arithmetic expression two things need be considered:

The precedence of the operator - If any operand is present in-between two different operators, the precedence (priority) of the one operator over the other decides which operator should use the operand first or which operator to be evaluated first. For example in the arithmetic expression a + b * c the operand b is surrounded by the operators * and +. In computer languages the * enjoying the higher precedence than the +, so * takes b first.

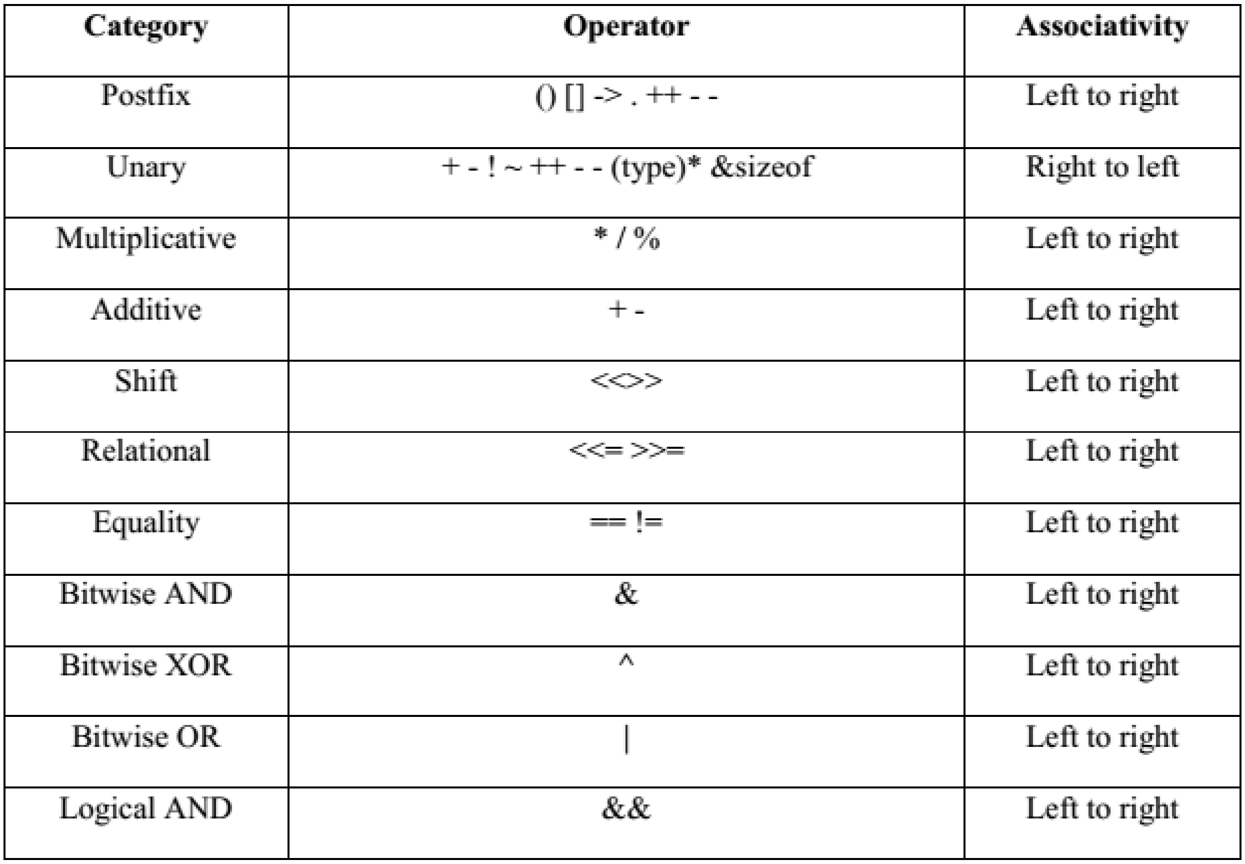
The Associativity of the operator - It resolves the ties between the same precedence of operators in the arithmetic expression, by considering the whether the operators are evaluated from left to right or right to left. For example in the expression a* b * c, here b is surrounded by two * operators, So which * should take b first or which multiplication must be done first? It is decided by the associativity of *. So in computer languages * is a left associative, so the first * will be performed first (multiply and b first then the result will be multiplied by c).

- It is not advantageous to evaluate the infix expressions, so first convert them to postfix or prefix and then compute the expressions.

Precedence of operators:

(+ and –) -> Same precedence but both have less precedence than *.

(* and /) have the same precedence.



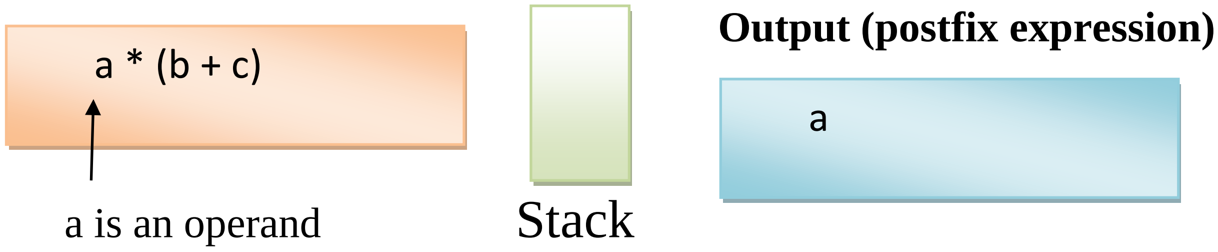
Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* &sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<<>>	Left to right
Relational	<<= >>=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right

Infix to postfix Conversion

1. Read all the symbols one by one from left to right in the given Infix Expression.
2. If the reading symbol is operand, then immediately send it to the output.
3. If the reading symbol is left parenthesis '(', then Push it on to the Stack.
4. If the reading symbol is right parenthesis ')', then Pop all the contents of stack until respective left parenthesis is popped and print each popped symbol to the output.
5. If the reading symbol is operator (+ , - , * , / etc.,), then Push it on to the Stack. However, first pop the operators which are already on the stack that have higher or equal precedence than the current operator and output them. If open parenthesis is there on top of the stack then push the operator into stack, even though the precedence of (is more than any other operator and it is the exceptional case.
6. If the input is over, so pop all the remaining symbols from the stack and output them.

Infix to postfix example:

1) Infix expression: a * (b + c)



Output (postfix expression)

a * (b + c)

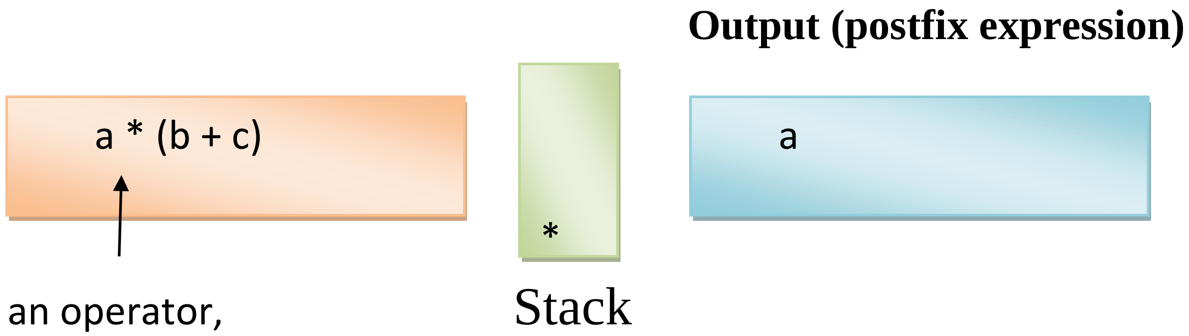
Stack

a

a is an operand

Action - Output a

2)



Output (postfix expression)

a * (b + c)

*

Stack

a

* is an operator,

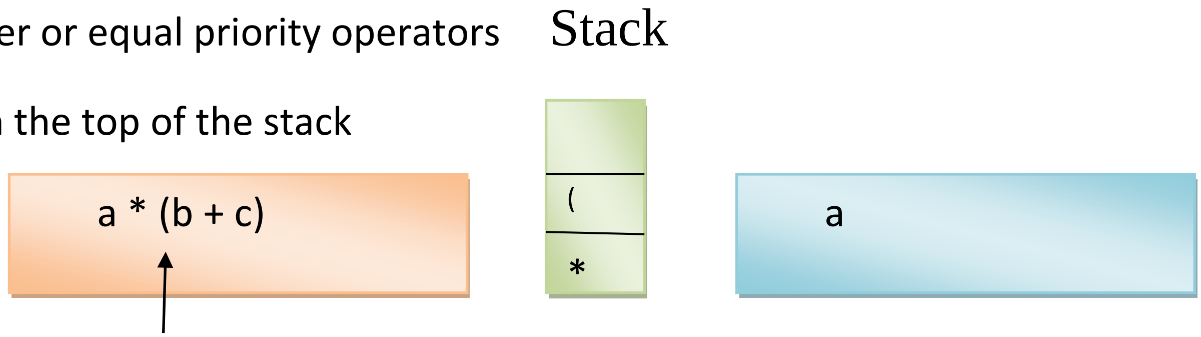
Action - push it into stack

after removing the

higher or equal priority operators

from the top of the stack

3)



Stack

a * (b + c)

(

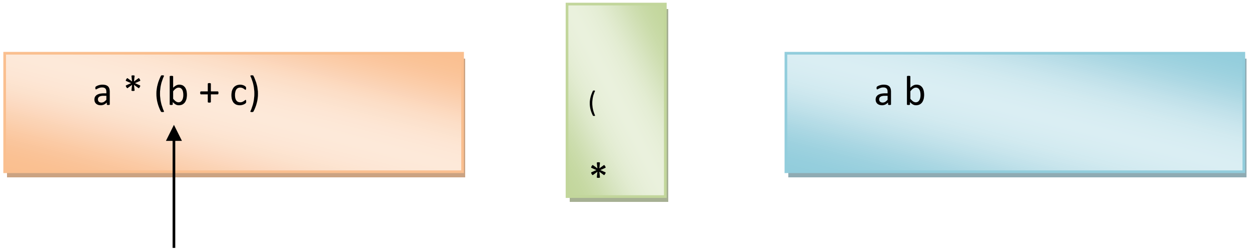
*

a

(is an open parenthesis

Action - push it into stack

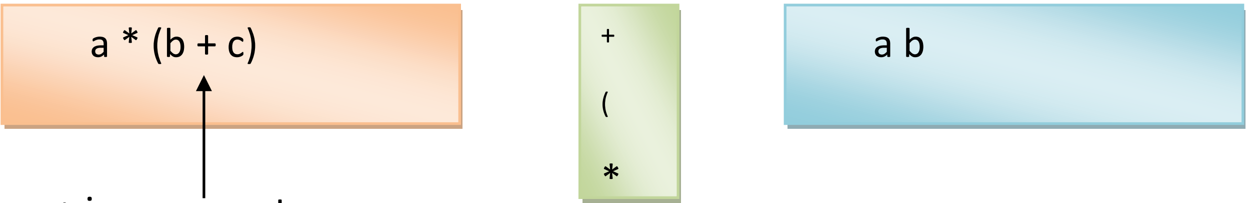
4)



b is an operand, so output b

Stack

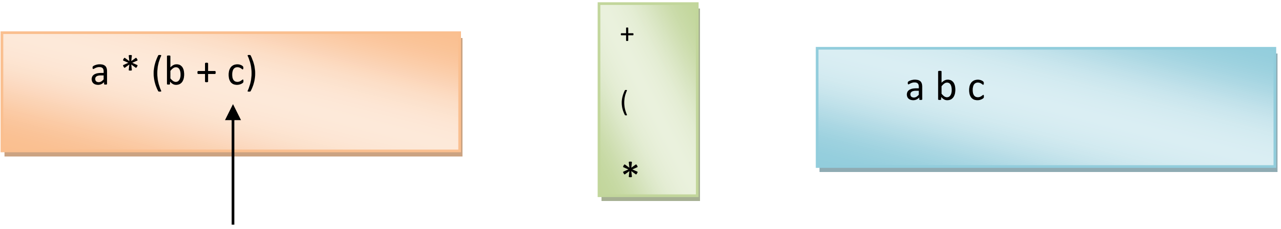
5)



+ is an operator

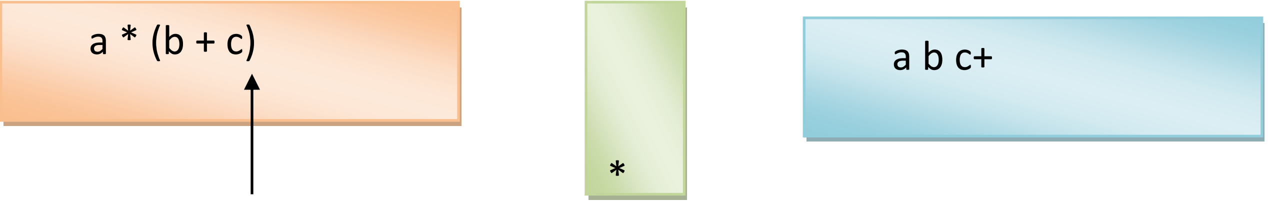
Action - push it into stack after removing the higher or equal priority operators from the top of the stack. Present operator on stack is (and it has higher precedence than +, but for (we have exception, so push + into stack.

6)



c is an operand, so output c

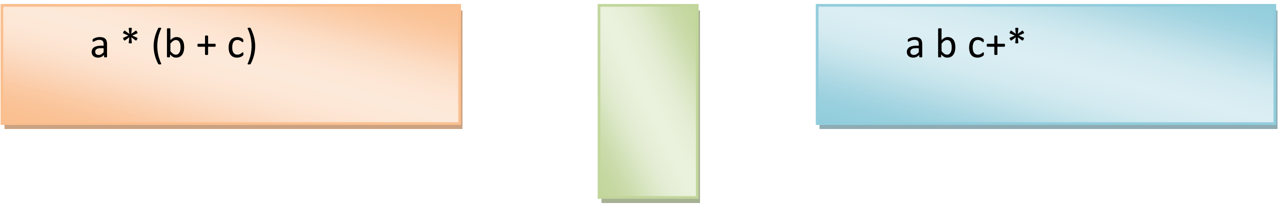
7)



) is an operator,

Action – Pop all the contents of stack until respective left parenthesis is popped and send each popped symbol to the output.

8)



Action: Input is over so pop all the remaining symbols from the stack and output them.

Infix to Prefix Conversion

1. Reverse the given infix expression, for example the reverse of the infix expression a + b *c is c * b + a. When reversing the parenthesis ')' will become '(' and '(' will become ')' (applicable to all types of brackets)
2. Apply the infix to postfix conversion algorithm on the reversed infix expression. For the above example the resultant postfix expression is: cb*a+
3. Reverse the obtained postfix expression, and is the required prefix expression for the given infix expression. For the above example, the infix expression is +a*bc

Prefix to Infix Conversion

1. Reverse the given prefix expression, for example the reverse of the prefix expression *cd is dc*
2. Read the character by character of the reversed infix expression and repeat the step 3 and 4 till there are no characters in the reversed prefix expression.
3. If the character read is an operand then push the operand into the stack. (push d and c)
4. If the character read is an operator, *op*, then pop the top two symbols from the stack, the first one is *p1*, and the second one is *p2*. Push the concatenated string ***p1 op p2*** to stack. (push c *d)
5. Now the value in the stack is the required infix expression. (c*d)

Prefix to Postfix Conversion

1. Reverse the given prefix expression, for example the reverse of the prefix expression *cd is cd*.

2. Read the character by character of the reversed infix expression and repeat the step 3 and 4 till there are no characters in the reversed prefix expression .
3. If the character read is an operand then push the operand into the stack. (push d and c)
4. If the character read is an operator, *op*, then pop the top two symbols from the stack, the first one is *p1*, and the second one is *p2*. Push the concatenated string ***p1 p2 op*** to stack. (push c d *)
5. Now the value in the stack is the required postfix expression. (c d *)

Postfix to Infix Conversion

1. Read the character by character of the given postfix expression and repeat the step 3 and 4 till there are no characters in the postfix expression. For example assume the postfix expression ab+;
2. If the character read is an operand then push the operand into the stack. (push a and b)
3. If the character read is an operator, *op*, then pop the top two symbols from the stack, the first one is *p1*, and the second one is *p2*. Push the concatenated string ***p2 op p1*** to stack. (push a * b)
4. Now the value in the stack is the required infix expression. (a * b)

Postfix to Prefix Conversion

1. Read the character by character of the given postfix expression and repeat the step 2 and 3 till there are no characters in the postfix expression. For example assume the postfix expression ab+;
2. If the character read is an operand then push the operand into the stack. (push a and b)

3. If the character read is an operator, *op*, then pop the top two symbols from the stack, the first one is *p1*, and the second one is *p2*. Push the concatenated string **op *p2 p1*** to stack. (push a * b)
4. Now the value in the stack is the required prefix expression. (*a b)

Examples

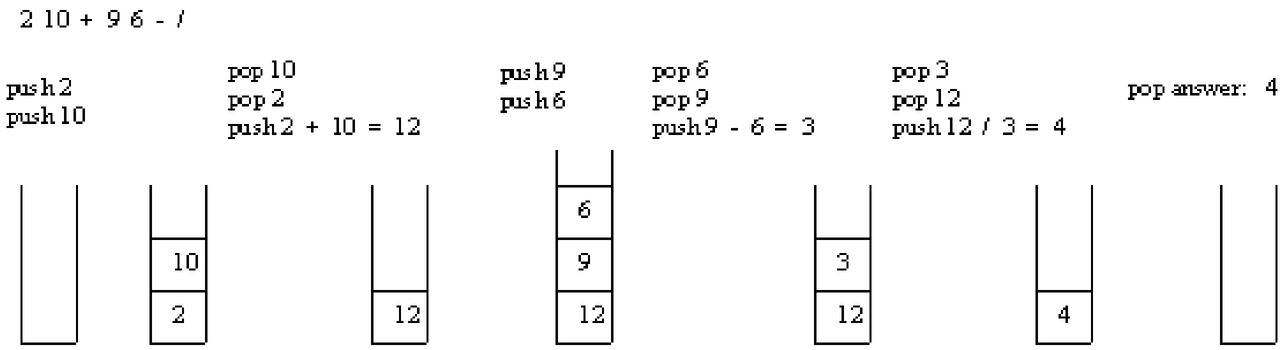
S.No	Infix Expression	Prefix Expression	Postfix Expression
1	a + b * c	+a*bc	a b c*+
2	(a + b) * (c + d)	* + a b + c d	a b + c d + *
3	a * b + c * d	+ * a b * c d	a b * c d * +
4	a + b + c + d	+ + + a b c d	a b + c + d +
5	(a + b) * c – d	-*+a b c d	ab + c * d-

Evaluation of Postfix Expression

1. Scan the symbols one by one from left to right in the given Postfix Expression
2. If the reading symbol is operand, then push it on to the Stack.
3. If the reading symbol is binary operator (+ , - , * , / etc.,), then perform two pop operations and do the operation specified by

the reading symbol (if it is * do multiplication, etc) using the two popped operands and push the result back on to the stack.

4. Repeat steps 2 & 3 till the postfix expression completes.
5. Finally! perform a pop operation and display the popped value as final result.



Queues

- Queue is also a linear data structure in which the insertion and deletion operations are performed at two different ends as it is opposed to stacks.
- The insertion is performed at one end (Called as **enqueue**) and deletion is performed at the other end (called as **dequeue**).
- To perform insertion and deletion we used two pointers **rear** and **front**.
- Insertion operation is performed at a position pointed by '**rear**' and the deletion operation is performed at a position pointed by '**front**'.