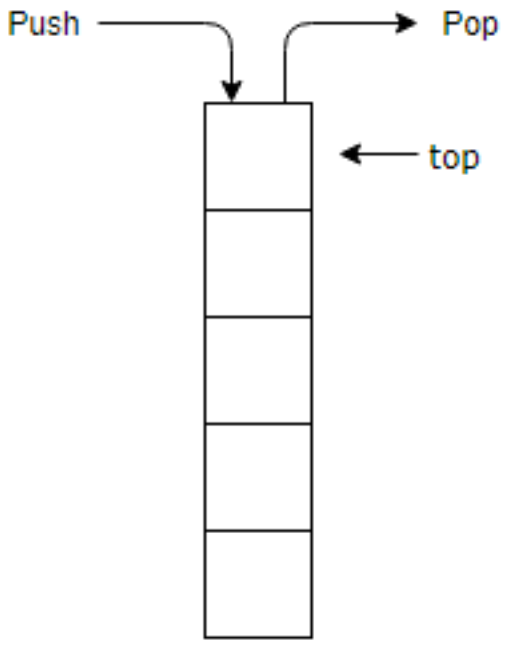
UNIT 4:

Stacks and Queues:

What is Stack?

A Stack is a linear data structure which allows adding and removing of elements in a particular order. New elements are added at the top of Stack. If we want to remove an element from the Stack, we can only remove the top element from Stack. Since it allows insertion and deletion from only one end and the element to be inserted last will be the element to be deleted first, hence it is called Last in First Out data structure (LIFO).



Stack Data Structure

Here we will define three operations on Stack,

- Push - it specifies adding an element to the Stack. If we try to insert an element when the Stack is full, then it is said to be Stack Overflow condition
- Pop - it specifies removing an element from the Stack. Elements are always removed from top of Stack. If we try to perform pop operation on an empty Stack, then it is said to be Stack Underflow condition.
- Peek - it will show the element on the top of Stack(without removing it).

Implementing Stack functionalities using Linked List

Stack can be implemented using both, arrays and linked list. The limitation in case of array is that we need to define the size at the beginning of the implementation. This makes our Stack static. It can also result in *"Stack overflow"* if we try to add elements after the array is full. So, to alleviate this problem, we use linked list to implement the Stack so that it can grow in real time.

First, we will create our Node class which will form our Linked List. We will be using this same Node class to implement the Queue also in the later part of this article.

```
1. internal class Node
2. {
3.     internal int data;
4.     internal Node next;
5.
6.     // Constructor to create a new node.Next is by default in
   itialized as null
7.     public Node(int d)
8.     {
9.         data = d;
10.          next = null;
11.      }
12.  }
```

Now, we will create our Stack Class. We will define a pointer, top, and initialize it to null. So, our LinkedListStack class will be -

```
1. internal class LinkListStack
2. {
3.     Node top;
4.
5.     public LinkListStack()
6.     {
7.         this.top = null;
8.     }
9.  }
```

Push an element into Stack

Now, our Stack and Node class is ready. So, we will proceed to Push operation on Stack. We will add a new element at the top of Stack.

Algorithm

- Create a new node with the value to be inserted.
- If the Stack is empty, set the next of the new node to null.
- If the Stack is not empty, set the next of the new node to top.
- Finally, increment the top to point to the new node.

The time complexity for *Push* operation is O(1). The method for Push will look like this.

```
internal void Push(int value)
{
    Node newNode = new Node(value);
    if (top == null)
    {
        newNode.next = null;
    }
    else
    {
         newNode.next = top;
     }
     top = newNode;
     Console.WriteLine("{0} pushed to stack", value);
 }
```

Pop an element from Stack

We will remove the top element from Stack.

Algorithm

- If the Stack is empty, terminate the method as it is Stack underflow.
- If the Stack is not empty, increment top to point to the next node.
- Hence the element pointed by top earlier is now removed.

The time complexity for Pop operation is O(1). The method for Pop will be like following.

```
internal void Pop()
{
    if (top == null)
    {
        Console.WriteLine("Stack Underflow. Deletion not poss
ible");
        return;
    }

    Console.WriteLine("Item popped is {0}", top.data);
     top = top.next;
 }
```

Peek the element from Stack

The peek operation will always return the top element of Stack without removing it from Stack.

Algorithm

- If the Stack is empty, terminate the method as it is Stack underflow.
- If the Stack is not empty, return the element pointed by the top.

The time complexity for Peek operation is O(1). The Peek method will be like following.

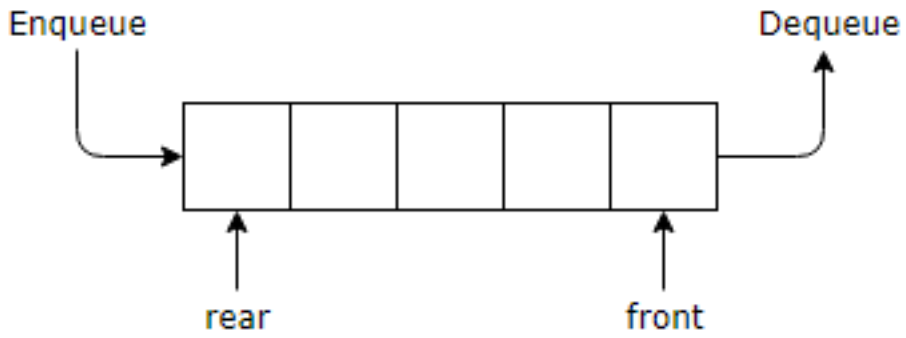
```
internal void Peek()
 {
      if (top == null)
      {
          Console.WriteLine("Stack Underflow.");
          return;
      }

      Console.WriteLine("{0} is on the top of Stack", this.top
.data);
  }
```

Uses of Stack

- Stack can be used to implement back/forward button in the browser.
- Undo feature in the text editors are also implemented using Stack.
- It is also used to implement recursion.
- Call and return mechanism for a method uses Stack.
- It is also used to implement backtracking.

What is Queue?

A Queue is also a linear data structure where insertions and deletions are performed from two different ends. A new element is added from the rear of Queue and deletion of existing element occurs from the front. Since we can access elements from both ends and the element inserted first will be the one to be deleted first, hence Queue is called First in First Out data structure (FIFO).



Queue data structure

Here, we will define two operations on Queue.

- Enqueue - It specifies the insertion of a new element to the Queue. Enqueue will always take place from the rear end of the Queue.
- Dequeue - It specifies the deletion of an existing element from the Queue. Dequeue will always take place from the front end of the Queue.

Implementing Queue functionalities using Linked List

Similar to Stack, the Queue can also be implemented using both, arrays and linked list. But it also has the same drawback of limited size. Hence, we will be using a Linked list to implement the Queue.

The Node class will be the same as defined above in Stack implementation. We will define LinkedListQueue class as below.

```
1. internal class LinkListQueue
2. {
3.     Node front;
4.     Node rear;
5.
6.     public LinkListQueue()
7.     {
8.         this.front = this.rear = null;
9.     }
10.  }
```

Here, we have taken two pointers - rear and front - to refer to the rear and the front end of the Queue respectively and will initialize it to null.

Enqueue of an Element

We will add a new element to our Queue from the rear end.

Algorithm

- Create a new node with the value to be inserted.
- If the Queue is empty, then set both front and rear to point to newNode.
- If the Queue is not empty, then set next of rear to the new node and the rear to point to the new node.

The time complexity for Enqueue operation is O(1). The Method for *Enqueue* will be like the following.

```
1. internal void Enqueue(int item)
2.  {
3.      Node newNode = new Node(item);
4.
5.      // If queue is empty, then new node is front and rear bo
   th
6.      if (this.rear == null)
7.      {
8.          this.front = this.rear = newNode;
9.      }
10.       else
11.       {
```

```
            // Add the new node at the end of queue and change
   rear
            this.rear.next = newNode;
            this.rear = newNode;
        }
        Console.WriteLine("{0} inserted into Queue", item);
    }
```

Dequeue of an Element

We will delete the existing element from the Queue from the front end.

Algorithm

- If the Queue is empty, terminate the method.
- If the Queue is not empty, increment front to point to next node.
- Finally, check if the front is null, then set rear to null also. This signifies empty Queue.

The time complexity for Dequeue operation is O(1). The Method for *Dequeue* will be like following.

```
internal void Dequeue()
 {
      // If queue is empty, return NULL.
      if (this.front == null)
      {
          Console.WriteLine("The Queue is empty");
          return;
      }

       // Store previous front and move front one node ahead
       Node temp = this.front;
       this.front = this.front.next;

       // If front becomes NULL, then change rear also as NULL

       if (this.front == null)
       {
           this.rear = null;
       }

       Console.WriteLine("Item deleted is {0}", temp.data);
  }
```

Uses of Queue

- CPU scheduling in Operating system uses Queue. The processes ready to execute and the requests of CPU resources wait in a queue and the request is served on first come first serve basis.
- Data buffer - a physical memory storage which is used to temporarily store data while it is being moved from one place to another is also implemented using Queue.

Conclusion

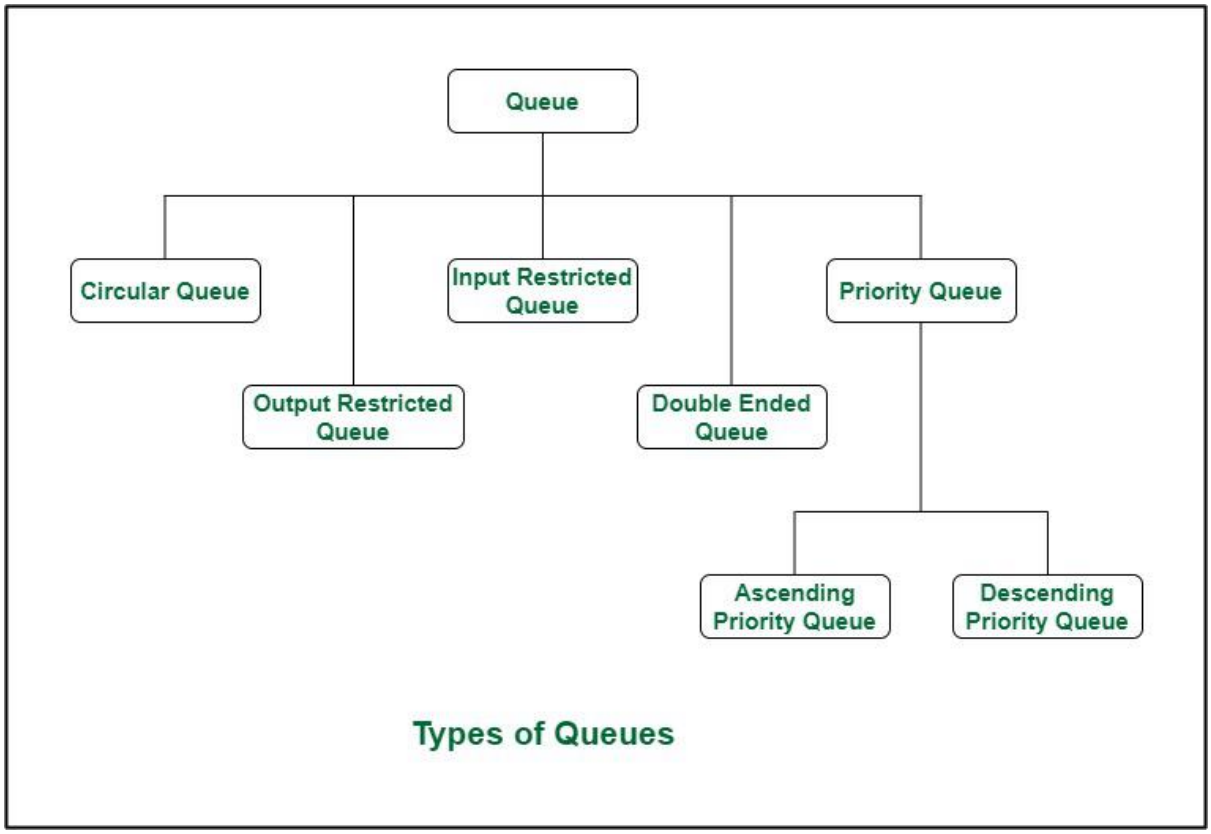
We learned about Stack and Queue data structures and also implemented them using Linked List. Please refer to the attached code for better understanding. You can also find the array implementation of Stack and Queue in the attached code.

A Queue is a linear structure that follows a particular order in which the operations are performed. The order is First In First Out (FIFO). A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first. In this article, the different types of queues are discussed.

Types of Queues:

There are **five different types of queues** that are used in different scenarios. They are:

1. Input Restricted Queue (this is a Simple Queue)
2. Output Restricted Queue (this is also a Simple Queue)
3. Circular Queue
4. Double Ended Queue (Deque)
5. Priority Queue
    - Ascending Priority Queue
    - Descending Priority Queue



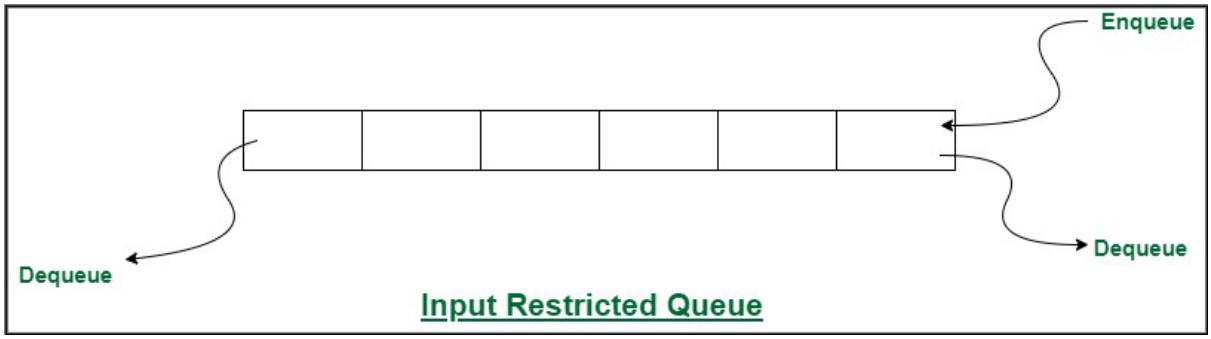
Types of Queues

1. Circular Queue: Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called **'Ring Buffer'**. This queue is primarily used in the following cases:

1. **Memory Management:** The unused memory locations in the case of ordinary queues can be utilized in circular queues.
2. **Traffic system:** In a computer-controlled traffic system, circular queues are used to switch on the traffic lights one by one repeatedly as per the time set.
3. **CPU Scheduling:** Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.

The time complexity for the circular Queue is O(1).

2. Input restricted Queue: In this type of Queue, the input can be taken from one side only(rear) and deletion of elements can be done from both sides(front and rear). This kind of Queue does not follow FIFO(first in first out). This queue is used in cases where the consumption of the data needs to be in FIFO order but if there is a need to remove the recently inserted data for some reason and one such case can be irrelevant data, performance issue, etc.



Input Restricted Queue

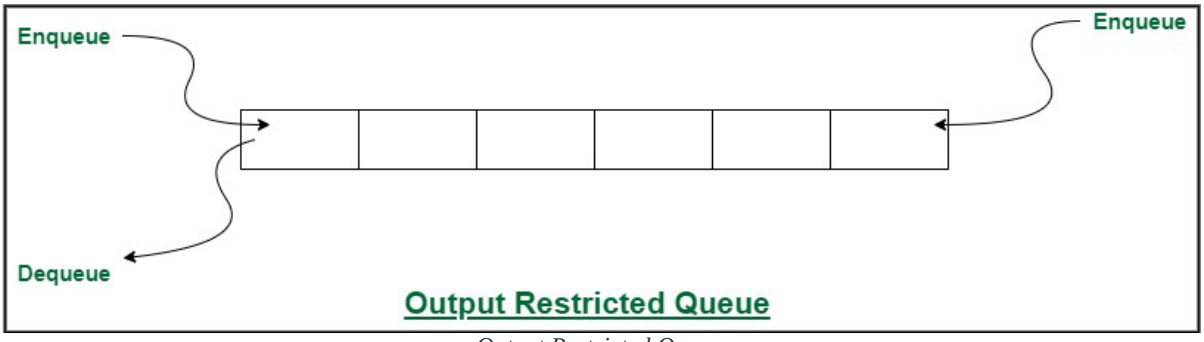
Advantages of Input restricted Queue:

- Prevents overflow and overloading of the queue by limiting the number of items added
- Helps maintain stability and predictable performance of the system

Disadvantages of Input restricted Queue:

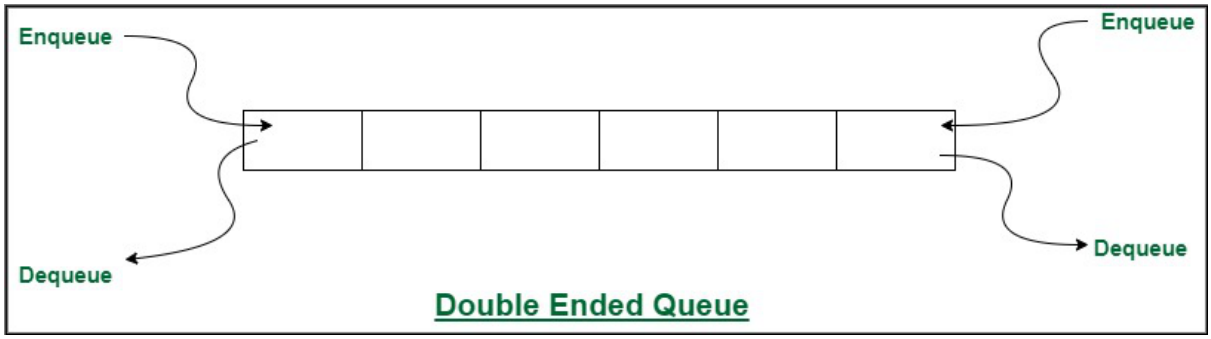
- May lead to resource wastage if the restriction is set too low and items are frequently discarded
- May lead to waiting or blocking if the restriction is set too high and the queue is full, preventing new items from being added.

3. Output restricted Queue: In this type of Queue, the input can be taken from both sides(rear and front) and the deletion of the element can be done from only one side(front). This queue is used in the case where the inputs have some priority order to be executed and the input can be placed even in the first place so that it is executed first.



Output Restricted Queue

4. Double ended Queue: Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (front and rear). That means, we can insert at both front and rear positions and can delete from both front and rear positions. Since Deque supports both stack and queue operations, it can be used as both. The Deque data structure supports clockwise and anticlockwise rotations in O(1) time which can be useful in certain applications. Also, the problems where elements need to be removed and or added both ends can be efficiently solved using Deque.



Double Ended Queue

5. Priority Queue: A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority. There are two types of Priority Queues. They are:

1. **Ascending Priority Queue:** Element can be inserted arbitrarily but only smallest element can be removed. For example, suppose there is an array having elements 4, 2, 8 in the same order. So, while inserting the elements, the insertion will be in the same sequence but while deleting, the order will be 2, 4, 8.
2. **Descending priority Queue:** Element can be inserted arbitrarily but only the largest element can be removed first from the given Queue. For example, suppose there is an array having elements 4, 2, 8 in the same order. So, while inserting the elements, the insertion will be in the same sequence but while deleting, the order will be 8, 4, 2.

The time complexity of the Priority Queue is O(logn).

Applications of a Queue:

The queue is used when things don't have to be processed immediately, but have to be processed in First In First Out order like Breadth First Search. This property of Queue makes it also useful in the following kind of scenarios.

1. When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
2. When data is transferred asynchronously (data not necessarily received at the same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.
3. Linear Queue: A linear queue is a type of queue where data elements are added to the end of the queue and removed from the front of the queue. Linear queues are used in applications where data elements need to be processed in the order in which they are received. Examples include printer queues and message queues.
4. Circular Queue: A circular queue is similar to a linear queue, but the end of the queue is connected to the front of the queue. This allows for efficient use of space in memory and can improve performance. Circular queues are used

in applications where the data elements need to be processed in a circular fashion. Examples include CPU scheduling and memory management.

5. Priority Queue: A priority queue is a type of queue where each element is assigned a priority level. Elements with higher priority levels are processed before elements with lower priority levels. Priority queues are used in applications where certain tasks or data elements need to be processed with higher priority. Examples include operating system task scheduling and network packet scheduling.
6. Double-ended Queue: A double-ended queue, also known as a deque, is a type of queue where elements can be added or removed from either end of the queue. This allows for more flexibility in data processing and can be used in applications where elements need to be processed in multiple directions. Examples include job scheduling and searching algorithms.

7. Concurrent Queue: A concurrent queue is a type of queue that is designed to handle multiple threads accessing the queue simultaneously. Concurrent queues are used in multi-threaded applications where data needs to be shared between threads in a thread-safe manner. Examples include database transactions and web server requests.

Issues of Queue :

Some common issues that can arise when using queues:

1. Queue overflow: Queue overflow occurs when the queue reaches its maximum capacity and is unable to accept any more elements. This can cause data loss and can lead to application crashes.
2. Queue underflow: Queue underflow occurs when an attempt is made to remove an element from an empty queue. This can cause errors and application crashes.
3. Priority inversion: Priority inversion occurs in priority queues when a low-priority task holds a resource that a high-priority task needs. This can cause delays in processing and can impact system performance.
4. Deadlocks: Deadlocks occur when multiple threads or processes are waiting for each other to release resources, resulting in a situation where none of the threads can proceed. This can happen when using concurrent queues and can lead to system crashes.
5. Performance issues: Queue performance can be impacted by various factors, such as the size of the queue, the frequency of access, and the type of operations performed on the queue. Poor queue performance can lead to slower system performance and reduced user experience.
6. Synchronization issues: Synchronization issues can arise when multiple threads are accessing the same queue simultaneously. This can result in data corruption, race conditions, and other errors.

7. Memory management issues: Queues can use up significant amounts of memory, especially when processing large data sets. Memory leaks and other memory management issues can occur, leading to system crashes and other errors.

Applications of stacks:

Expression Evaluation:

While reading the expression from left to right, push the element in the stack if it is an operand. Pop the two operands from the stack, if the element is an operator and then evaluate it. Push back the result of the evaluation. Repeat it till the end of the expression.

Expression Conversion:

One of the applications of Stack is in the conversion of arithmetic expressions in high-level programming languages into machine readable form.An expression can be represented in infix,postfix and prefix and stack proves to be useful while converting one form to another.

Syntax Parsing:

Conversion from one form of the expression to another form needs a stack. Many compilers use a stack for parsing the syntax of expressions, program blocks etc before translating into low level code.

Parenthesis Checking:

One of the most important applications of stacks is to check if the parentheses are balanced in a given expression. The compiler generates an error if the parentheses are not matched.

String Reversal:

Reversing string is an operation of Stack by using it we can reverse any string.

Function Call:

The function call stack (often referred to just as the call stack or the stack) is responsible for maintaining the local variables and parameters during function execution

Conversion of infix to postfix

Here, we will use the stack data structure for the conversion of infix expression to prefix expression. Whenever an operator will encounter, we push operator into the stack. If we encounter an operand, then we append the operand to the expression.

Rules for the conversion from infix to postfix expression

1. Print the operand as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator on to the stack.
3. If the incoming symbol is '(', push it on to the stack.
4. If the incoming symbol is ')', pop the stack and print the operators until the left parenthesis is found.
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has lower precedence than the top of the stack, pop and print the top of the stack. Then test the incoming operator against the new top of the stack.
7. If the incoming operator has the same precedence with the top of the stack then use the associativity rules. If the associativity is from left to right then pop and print the top of the stack then push the incoming operator. If the associativity is from right to left then push the incoming operator.
8. At the end of the expression, pop and print all the operators of the stack.

Let's understand through an example.

Infix expression: K + L - M*N + (O^P) * W/U/V * T + Q

Input Expression	Stack	Postfix Expression
K		K
+	+	
L	+	K L
-	-	K L+
M	-	K L+ M
*	- *	K L+ M
N	- *	K L + M N
+	+	K L + M N* K L + M N* -
(	+ (	K L + M N *-
O	+ (	K L + M N * - O
^	+ (^	K L + M N* - O
P	+ (^	K L + M N* - O P
)	+	K L + M N* - O P ^
*	+ *	K L + M N* - O P ^
W	+ *	K L + M N* - O P ^ W
/	+ /	K L + M N* - O P ^ W *
U	+ /	K L + M N* - O P ^W*U
/	+ /	K L + M N* - O P ^W*U/
V	+ /	KL + MN*-OP^W*U/V
*	+ *	KL+MN*-OP^W*U/V/

T	+ *	KL+MN*-OP^W*U/V/T
+	+	KL+MN*-OP^W*U/V/T* KL+MN*-OP^W*U/V/T*+
Q	+	KL+MN*-OP^W*U/V/T*Q
		KL+MN*-OP^W*U/V/T*+Q+

The final postfix expression of infix expression(K + L - M*N + (O^P) * W/U/V * T + Q) is KL+MN*-OP^W*U/V/T*+Q+.

Convert infix to prefix notation

What is infix notation?

An infix notation is a notation in which an expression is written in a usual or normal format. It is a notation in which the operators lie between the operands. The examples of infix notation are A+B, A*B, A/B, etc.

As we can see in the above examples, all the operators exist between the operands, so they are infix notations. Therefore, the syntax of infix notation can be written as:

<operand> <operator> <operand>

Parsing Infix expressions

In order to parse any expression, we need to take care of two things, i.e., **Operator precedence** and **Associativity**. Operator precedence means the precedence of any operator over another operator. For example:

A + B * C → A + (B * C)

As the multiplication operator has a higher precedence over the addition operator so B * C expression will be evaluated first. The result of the multiplication of B * C is added to the A.

Precedence order

Operators	Symbols
Parenthesis	{ }, ( ), [ ]
Exponential notation	^
Multiplication and Division	*, /
Addition and Subtraction	+, -

Associativity means when the operators with the same precedence exist in the expression. For example, in the expression, i.e., A + B - C, '+' and '-' operators are having the same precedence, so they are evaluated with the help of associativity. Since both '+' and '-' are left-associative, they would be evaluated as (A + B) - C.

Associativity order

Operators	Associativity
^	Right to Left
*, /	Left to Right
+, -	Left to Right

Let's understand the associativity through an example.

1 + 2*3 + 30/5

Since in the above expression, * and / have the same precedence, so we will apply the associativity rule. As we can observe in the above table that * and / operators have the left to right associativity, so we will scan from the leftmost operator. The operator that comes first will be evaluated first. The operator * appears before the / operator, and multiplication would be done first.

1+ (2*3) + (30/5)

1+6+6 = 13

What is Prefix notation?

A prefix notation is another form of expression but it does not require other information such as precedence and associativity, whereas an infix notation requires information of precedence and associativity. It is also known as **polish notation**. In prefix notation, an operator comes before the operands. The syntax of prefix notation is given below:

<operator> <operand> <operand>

For example, if the infix expression is 5+1, then the prefix expression corresponding to this infix expression is +51.

If the infix expression is:

a * b + c

↓

***ab+c**

↓

+*abc (Prefix expression)

Consider another example:

A + B * C

First scan: In the above expression, multiplication operator has a higher precedence than the addition operator; the prefix notation of B*C would be (*BC).

A + *BC

Second scan: In the second scan, the prefix would be:

+A *BC

In the above expression, we use two scans to convert infix to prefix expression. If the expression is complex, then we require a greater number of scans. We need to use that method that requires only one scan, and provides the desired result. If we achieve the desired output through one scan, then the algorithm would be efficient. This is possible only by using a stack.

Conversion of Infix to Prefix using Stack

K + L - M * N + (O^P) * W/U/V * T + Q

If we are converting the expression from infix to prefix, we need first to reverse the expression.

The Reverse expression would be:

Q + T * V/U/W *) P^O(+ N*M - L + K

To obtain the prefix expression, we have created a table that consists of three columns, i.e., input expression, stack, and prefix expression. When we encounter any symbol, we simply add it into the prefix expression. If we encounter the operator, we will push it into the stack.

Input expression	Stack	Prefix expression
Q		Q
+	+	Q
T	+	QT
*	+*	QT
V	+*	QTV
/	+*/	QTV
U	+*/	QTVU
/	+*//	QTVU

W	+*//	QTVUW
*	+*//*	QTVUW
)	+*//*)	QTVUW
P	+*//*)	QTVUWP
^	+*//*)^	QTVUWP
O	+*//*)^	QTVUWPO
(	+*//*	QTVUWPO^
+	++	QTVUWPO^*//*
N	++	QTVUWPO^*//*N
*	++*	QTVUWPO^*//*N
M	++*	QTVUWPO^*//*NM
-	++-	QTVUWPO^*//*NM*
L	++-	QTVUWPO^*//*NM*L
+	++-+	QTVUWPO^*//*NM*L
K	++-+	QTVUWPO^*//*NM*LK
		QTVUWPO^*//*NM*LK+-++

The above expression, i.e., QTVUWPO^*//*NM*LK+-++, is not a final expression. We need to reverse this expression to obtain the prefix expression.

Rules for the conversion of infix to prefix expression:

- First, reverse the infix expression given in the problem.
- Scan the expression from left to right.
- Whenever the operands arrive, print them.

- If the operator arrives and the stack is found to be empty, then simply push the operator into the stack.
- If the incoming operator has higher precedence than the TOP of the stack, push the incoming operator into the stack.
- If the incoming operator has the same precedence with a TOP of the stack, push the incoming operator into the stack.
- If the incoming operator has lower precedence than the TOP of the stack, pop, and print the top of the stack. Test the incoming operator against the top of the stack again and pop the operator from the stack till it finds the operator of a lower precedence or same precedence.
- If the incoming operator has the same precedence with the top of the stack and the incoming operator is ^, then pop the top of the stack till the condition is true. If the condition is not true, push the ^ operator.
- When we reach the end of the expression, pop, and print all the operators from the top of the stack.
- If the operator is ')', then push it into the stack.
- If the operator is '(', then pop all the operators from the stack till it finds) opening bracket in the stack.
- If the top of the stack is ')', push the operator on the stack.
- At the end, reverse the output.

Evaluation of Postfix Expression

Given a postfix expression, the task is to evaluate the postfix expression.

Postfix expression: *The expression of the form "a b operator" (ab+) i.e., when a pair of operands is followed by an operator.*

Examples:

Input*: str = "2 3 1 * + 9 -"*

Output*: -4*

Explanation: *If the expression is converted into an infix expression, it will be 2 + (3 * 1) – 9 = 5 – 9 = -4.*
Input*: str = "100 200 + 2 / 5 * 7 +"*
Output*: 757*

Evaluation of Postfix Expression using Stack:

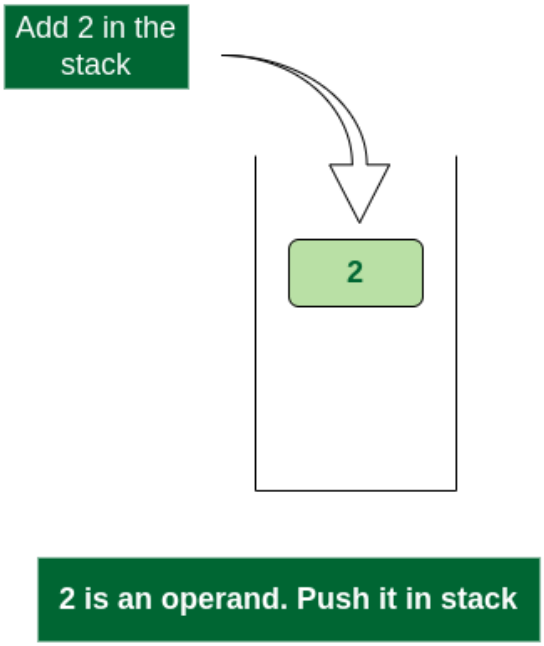
To evaluate a postfix expression we can use a stack.
Iterate the expression from left to right and keep on storing the operands into a stack. Once an operator is received, pop the two topmost elements and evaluate them and push the result in the stack again.

Illustration:

Follow the below illustration for a better understanding:

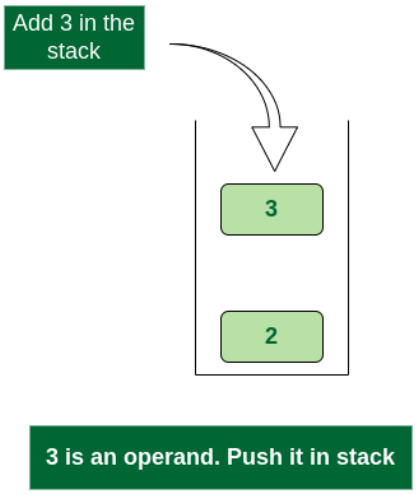
Consider the expression: exp = ***"2 3 1 * + 9 -"***

- *Scan* ***2****, it's a number, So push it into stack. Stack contains '2'.*



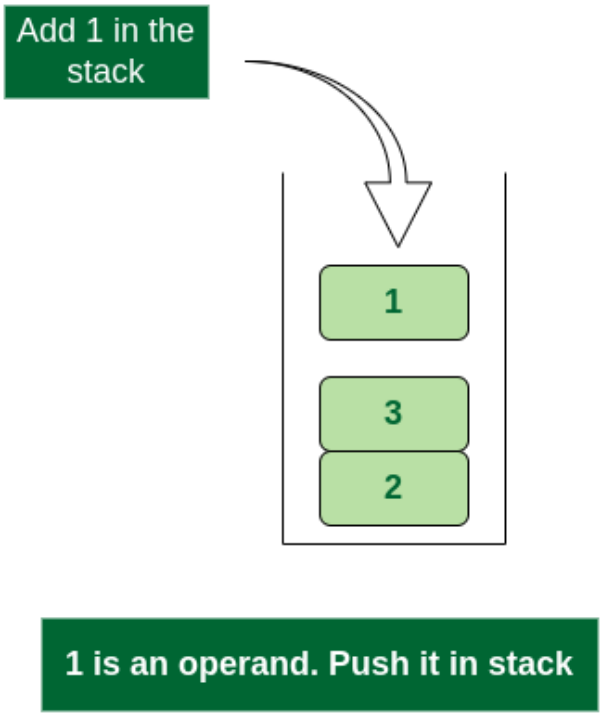
Push 2 into stack

- *Scan 3, again a number, push it to stack, stack now contains '2 3' (from bottom to top)*



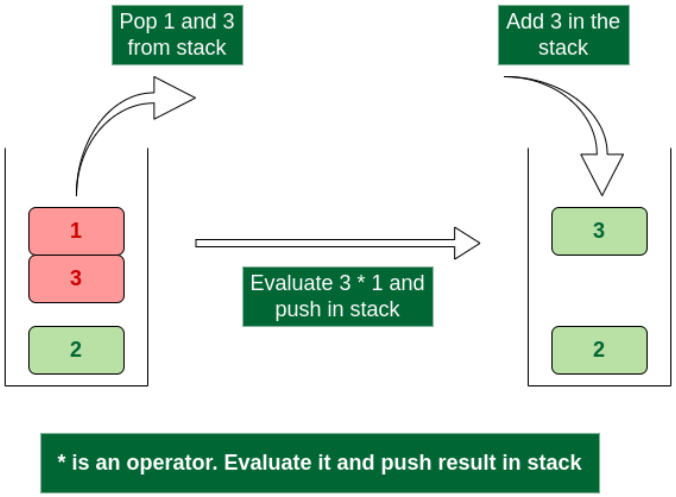
Push 3 into stack

- *Scan 1, again a number, push it to stack, stack now contains '2 3 1'*



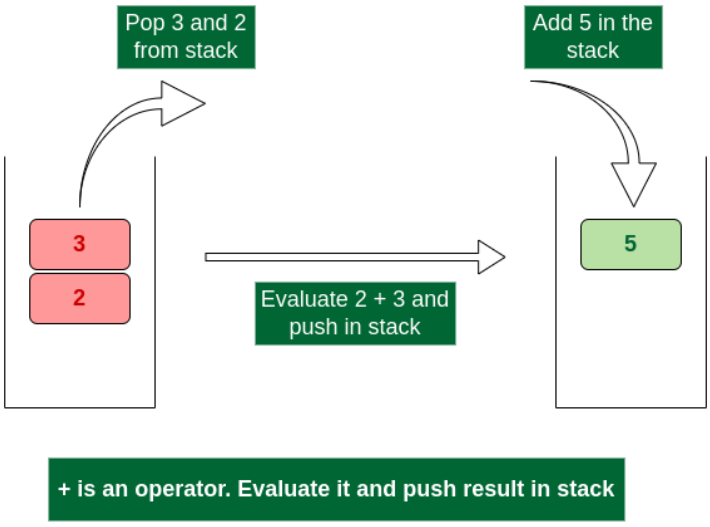
Push 1 into stack

- *Scan *, it's an operator. Pop two operands from stack, apply the * operator on operands. We get 3*1 which results in 3. We push the result 3 to stack. The stack now becomes '2 3'.*



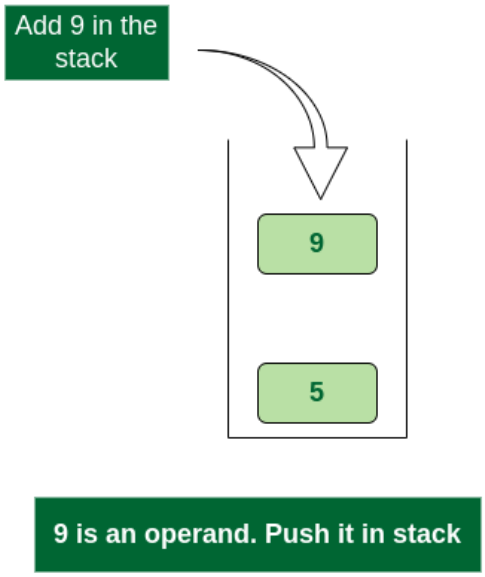
*Evaluate * operator and push result in stack*

- *Scan +, it's an operator. Pop two operands from stack, apply the + operator on operands. We get 3 + 2 which results in 5. We push the result 5 to stack. The stack now becomes '5'.*



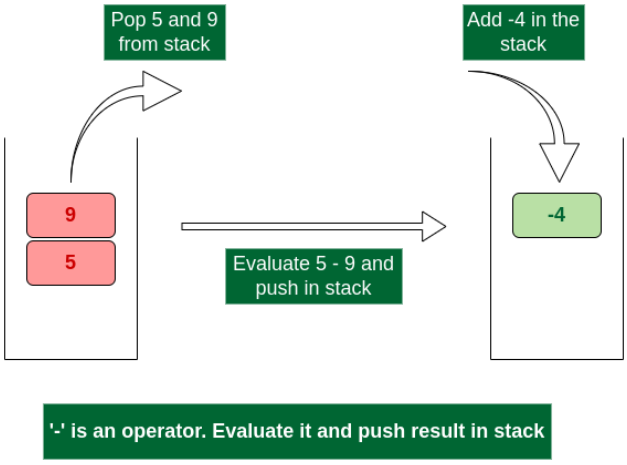
Evaluate + operator and push result in stack

- *Scan 9, it's a number. So we push it to the stack. The stack now becomes '5 9'.*



Push 9 into stack

- *Scan -, it's an operator, pop two operands from stack, apply the – operator on operands, we get 5 – 9 which results in -4. We push the result -4 to the stack. The stack now becomes '-4'.*



Evaluate '-' operator and push result in stack

- *There are no more elements to scan, we return the top element from the stack (which is the only element left in a stack).*

So the result becomes ***-4****.*

Follow the steps mentioned below to evaluate postfix expression using stack:

- Create a stack to store operands (or values).

- Scan the given expression from left to right and do the following for every scanned element.
    - If the element is a number, push it into the stack.
    - If the element is an operator, pop operands for the operator from the stack. Evaluate the operator and push the result back to the stack.
- When the expression is ended, the number in the stack is the final answer.