Stack

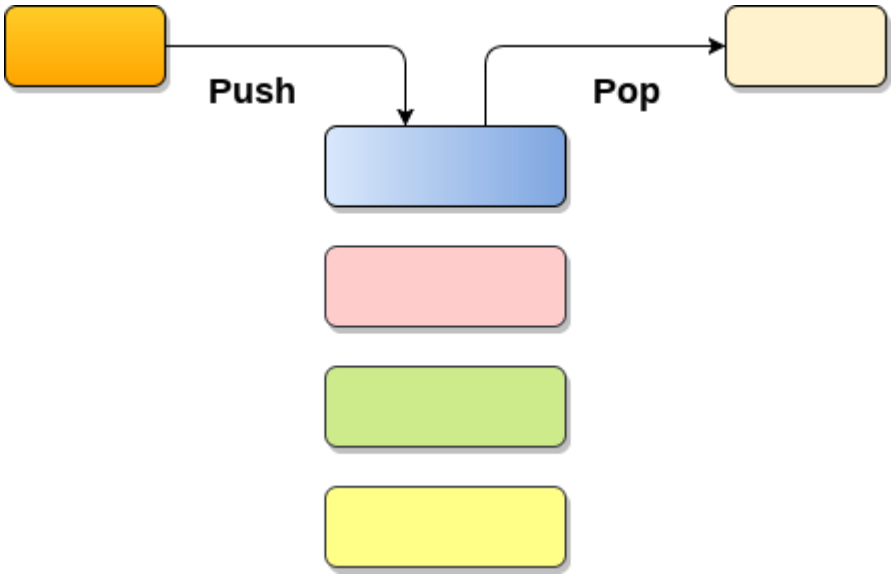
1. Stack is an ordered list in which, insertion and deletion can be performed only at one end that is called **top**.
2. Stack is a recursive data structure having pointer to its top element.
3. Stacks are sometimes called as Last-In-First-Out (LIFO) lists i.e. the element which is inserted first in the stack, will be deleted last from the stack.

Stack Specification

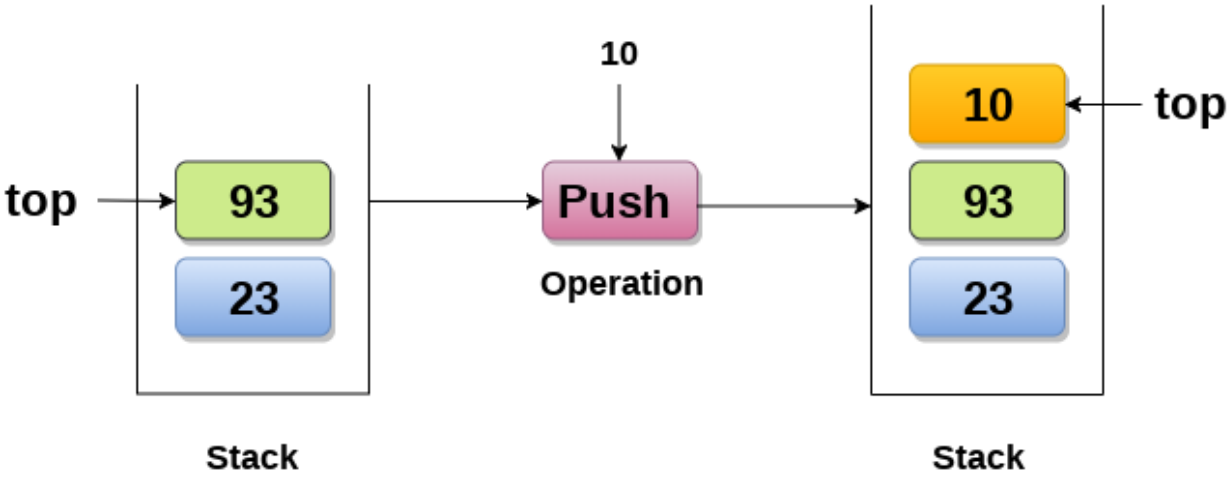
A stack is an object or more specifically an abstract data structure(ADT) that allows the following operations:

- `Push`: Add element to top of stack
- `Pop`: Remove element from top of stack
- `IsEmpty`: Check if stack is empty
- `IsFull`: Check if stack is full
- `Peek`: Get the value of the top element without removing it

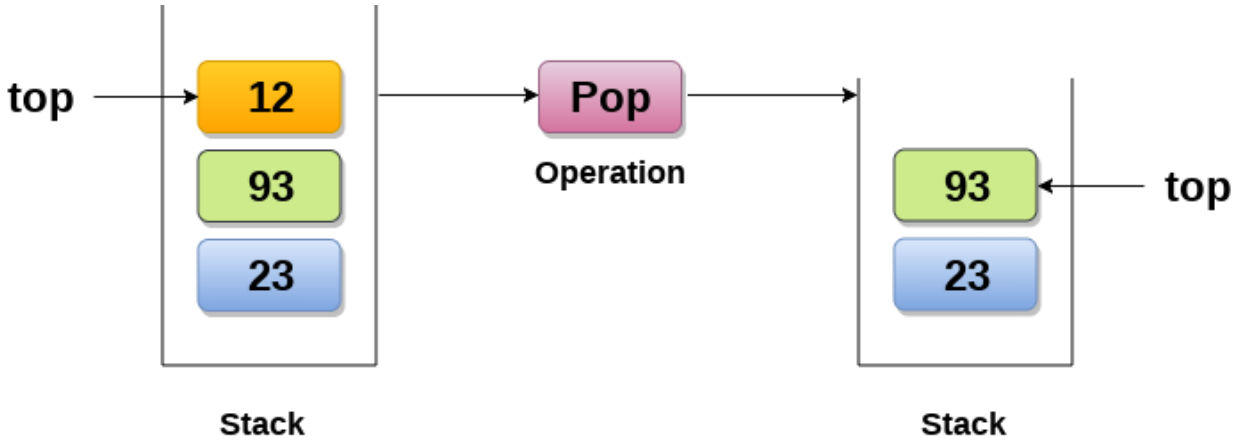
1. Push : Adding an element onto the stack



2. Pop : Removing an element from the stack



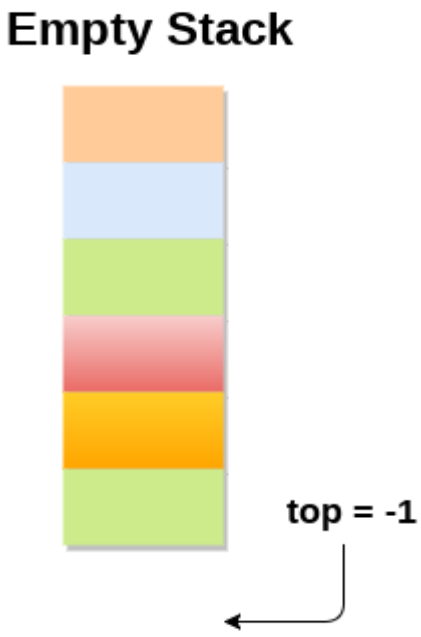
3. Peek : Look all the elements of stack without removing them.



How the stack grows?

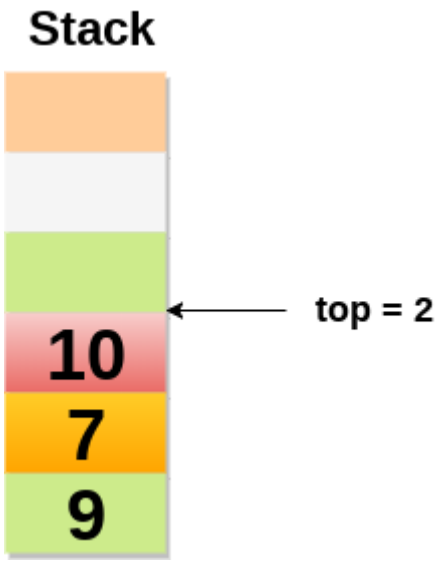
Scenario 1 : Stack is empty

The stack is called empty if it doesn't contain any element inside it. At this stage, the value of variable top is -1.



Scenario 2 : Stack is not empty

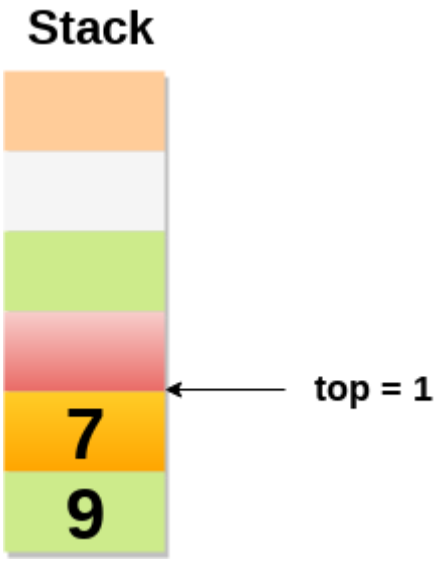
Value of top will get increased by 1 every time when we add any element to the stack. In the following stack, After adding first element, top = 2.



Scenario 3 : Deletion of an element

Value of top will get decreased by 1 whenever an element is deleted from the stack.

In the following stack, after deleting 10 from the stack, top = 1.



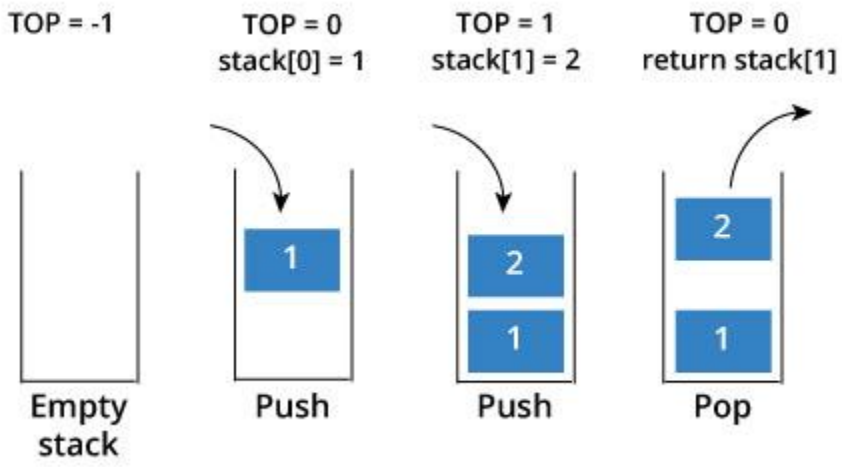
Top and its value :

Top position	Status of stack
-1	Empty
0	Only one element in the stack
N-1	Stack is full
N	Overflow

How stack works

The operations work as follows:

1. A pointer called *TOP* is used to keep track of the top element in the stack.
2. When initializing the stack, we set its value to -1 so that we can check if the stack is empty by comparing `TOP == -1`.
3. On pushing an element, we increase the value of *TOP* and place the new element in the position pointed to by *TOP*.
4. On popping an element, we return the element pointed to by *TOP* and reduce its value.
5. Before pushing, we check if stack is already full
6. Before popping, we check if stack is already empty



Stack.c/stacksample.c

Use of stack

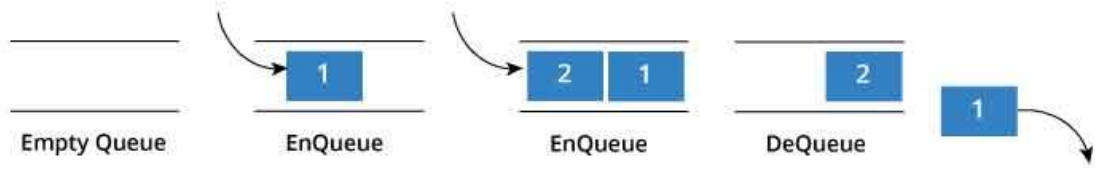
Although stack is a simple data structure to implement, it is very powerful. The most common uses of a stack are:

- **To reverse a word** - Put all the letters in a stack and pop them out. Because of LIFO order of stack, you will get the letters in reverse order.
- **In compilers** - Compilers use stack to calculate the value of expressions like 2+4/5*(7-9) by converting the expression to prefix or postfix form.
- **In browsers** - The back button in a browser saves all the urls you have visited previously in a stack. Each time you visit a new page, it is added on top of the stack. When you press the back button, the current URL is removed from the stack and the previous url is accessed.

Queue

A queue is a useful data structure in programming. It is similar to the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets the ticket.

Queue follows the **First In First Out(FIFO)** rule - the item that goes in first is the item that comes out first too.



In the above image, since 1 was kept in the queue before 2, it was the first to be removed from the queue as well. It follows the FIFO rule.

In programming terms, putting an item in the queue is called an "enqueue" and removing an item from the queue is called "dequeue".

Queue Specifications

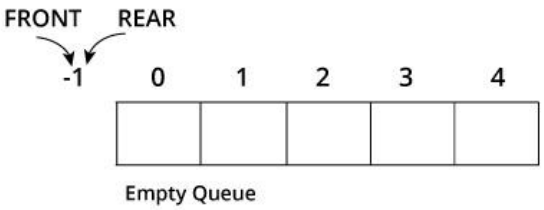
A queue is an object or more specifically an abstract data structure(ADT) that allows the following operations:

- *Enqueue:* Add element to end of queue
- *Dequeue:* Remove element from front of queue
- *IsEmpty:* Check if queue is empty
- *IsFull:* Check if queue is full
- *Peek:* Get the value of the front of queue without removing it

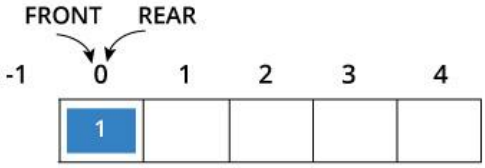
How Queue Works

Queue operations work as follows:

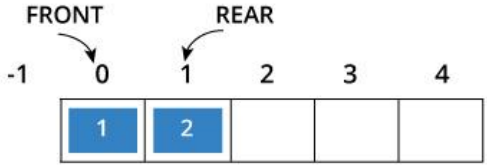
1. Two pointers called *FRONT* and *REAR* are used to keep track of the first and last elements in the queue.
2. When initializing the queue, we set the value of *FRONT* and *REAR* to -1.
3. On enqueing an element, we increase the value of *REAR* index and place the new element in the position pointed to by *REAR*.
4. On dequeueing an element, we return the value pointed to by *FRONT* and increase the *FRONT* index.
5. Before enqueing, we check if queue is already full.
6. Before dequeuing, we check if queue is already empty.
7. When enqueing the first element, we set the value of *FRONT* to 0.
8. When dequeing the last element, we reset the values of *FRONT* and *REAR* to -1.



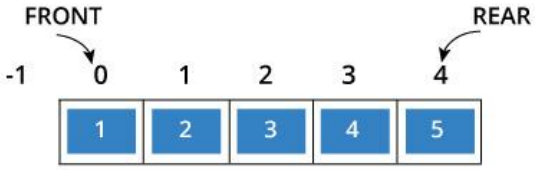
Empty Queue



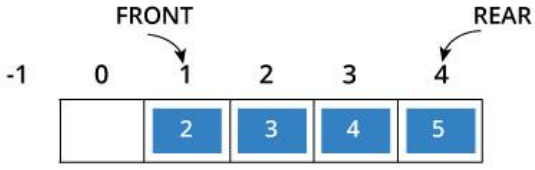
EnQueue first element



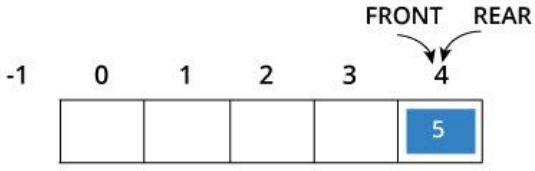
EnQueue



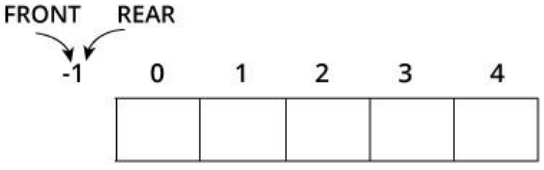
EnQueue



DeQueue



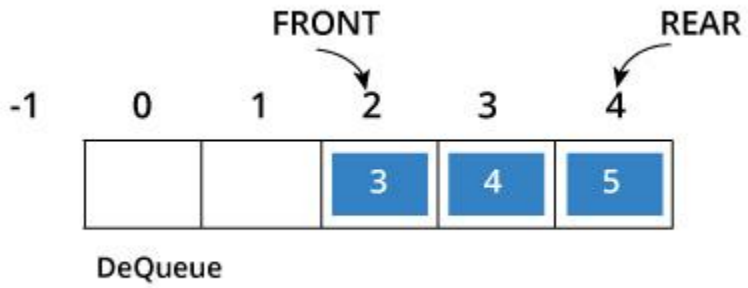
DeQueue last element



Empty Queue

Limitation of this implementation

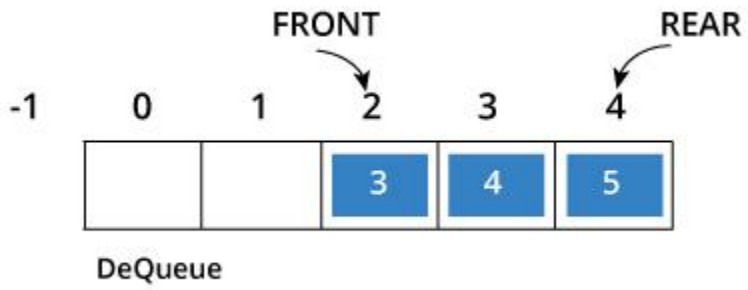
As you can see in the image below, after a bit of enqueueing and dequeueing, the size of the queue has been reduced.



The indexes 0 and 1 can only be used after the queue is reset when all the elements have been dequeued.

Circular Queue

Circular queue avoids the wastage of space in a regular queue implementation using arrays.



As you can see in the above image, after a bit of enqueueing and dequeueing, the size of the queue has been reduced.

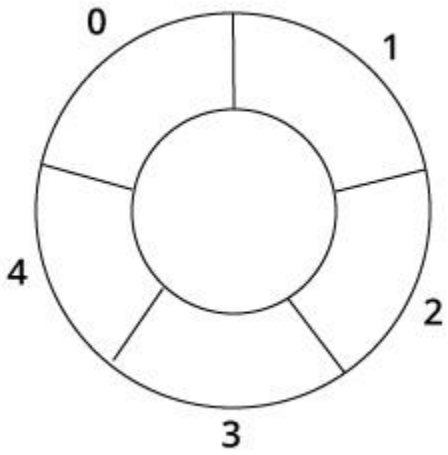
The indexes 0 and 1 can only be used after the queue is reset when all the elements have been dequeued.

How Circular Queue Works

Circular Queue works by the process of circular increment i.e. when we try to increment any variable and we reach the end of queue, we start from the beginning of queue by modulo division with the queue size.

i.e.

```
if REAR + 1 == 5 (overflow!), REAR = (REAR + 1)%5 = 0 (start of queue)
```



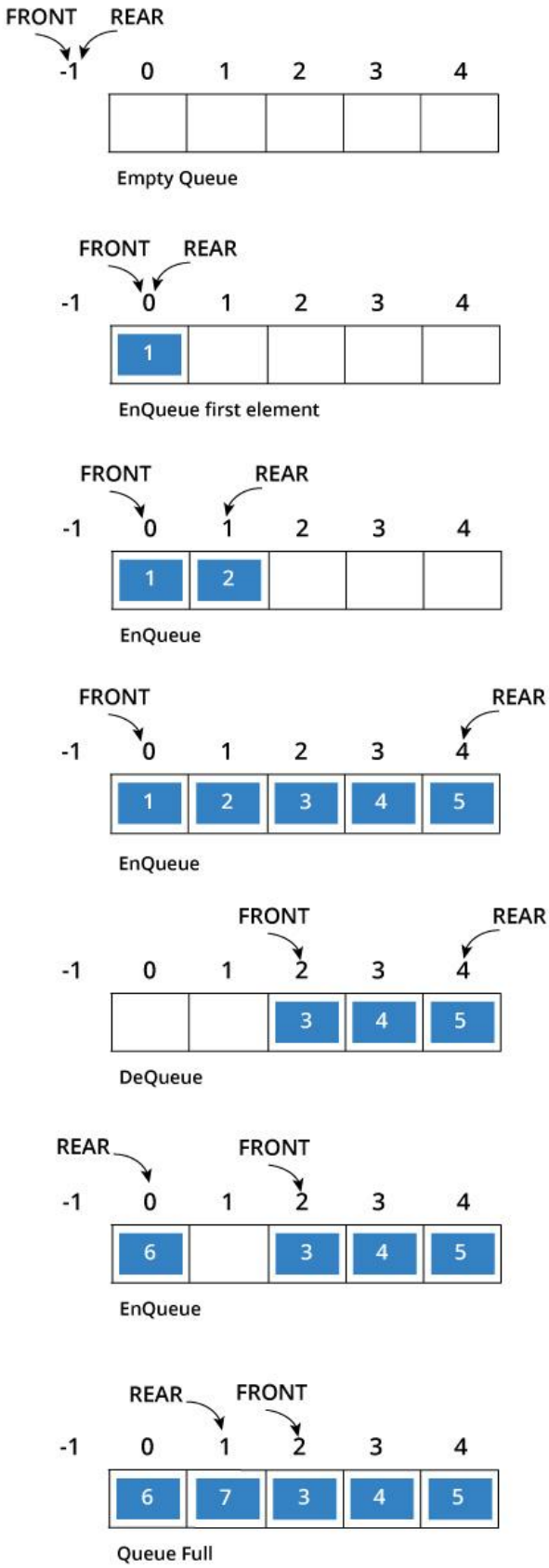
Queue operations work as follows:

- Two pointers called *FRONT* and *REAR* are used to keep track of the first and last elements in the queue.
- When initializing the queue, we set the value of *FRONT* and *REAR* to -1.
- On enqueing an element, we circularly increase the value of *REAR* index and place the new element in the position pointed to by *REAR*.
- On dequeueing an element, we return the value pointed to by *FRONT* and circularly increase the *FRONT* index.
- Before enqueing, we check if queue is already full.
- Before dequeuing, we check if queue is already empty.
- When enqueing the first element, we set the value of *FRONT* to 0.
- When dequeing the last element, we reset the values of *FRONT* and *REAR* to -1.

However, the check for full queue has a new additional case:

- Case 1: *FRONT* = 0 && `REAR == SIZE - 1`
- Case 2: `FRONT = REAR + 1`

The second case happens when *REAR* starts from 0 due to circular increment and when its value is just 1 less than *FRONT*, the queue is full.


FRONT
REAR
-1
0
1
2
3
4
Empty Queue
FRONT
REAR
-1
0
1
2
3
4
1
EnQueue first element
FRONT
REAR
-1
0
1
2
3
4
1
2
EnQueue
FRONT
REAR
-1
0
1
2
3
4
1
2
3
4
5
EnQueue
FRONT
REAR
-1
0
1
2
3
4
3
4
5
DeQueue
REAR
FRONT
-1
0
1
2
3
4
6
3
4
5
EnQueue
REAR
FRONT
-1
0
1
2
3
4
6
7
3
4
5
Queue Full