Stack Data Structure

A stack is a linear data structure that follows the principle of **Last In First Out (LIFO)**. This means the last element inserted inside the stack is removed first.

You can think of the stack data structure as the pile of plates on top of another.

Stack representation similar to a pile of plate

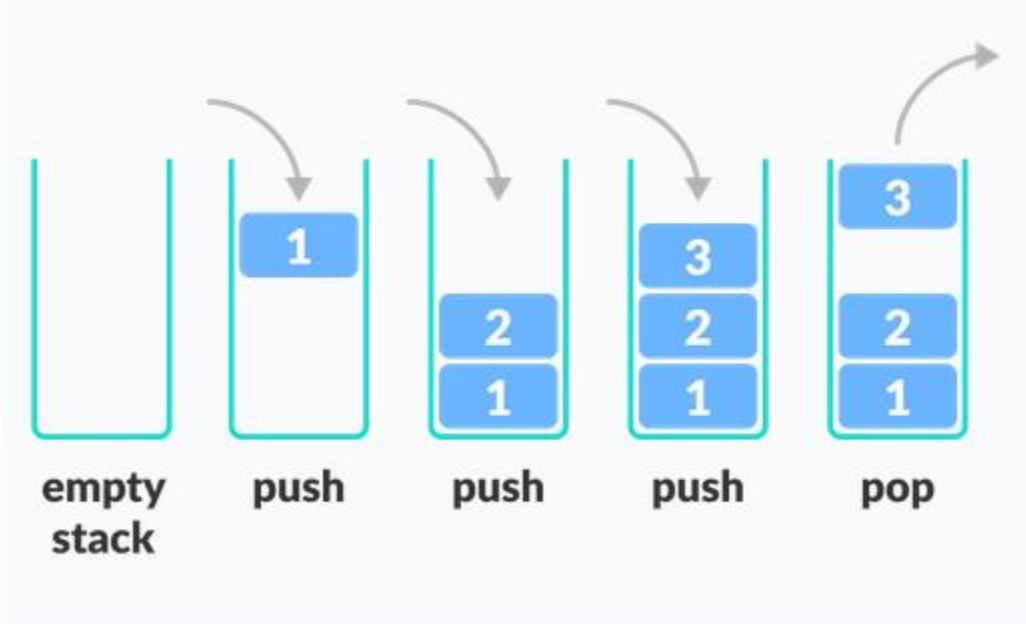
Here, you can:

- Put a new plate on top
- Remove the top plate

And, if you want the plate at the bottom, you must first remove all the plates on top. This is exactly how the stack data structure works.

LIFO Principle of Stack:

In programming terms, putting an item on top of the stack is called **push** and removing an item is called **pop**.



Stack Push and Pop Operations

In the above image, although item **3** was kept last, it was removed first. This is exactly how the **LIFO (Last In First Out) Principle** works.

Basic Operations of Stack:

There are some basic operations that allow us to perform different actions on a stack.

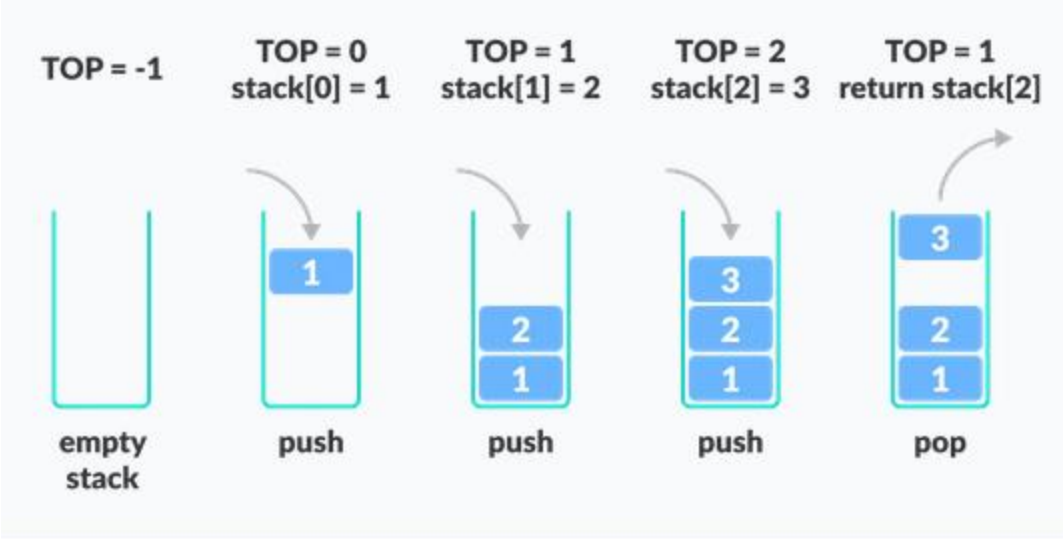
- **Push**: Add an element to the top of a stack
- **Pop**: Remove an element from the top of a stack
- **IsEmpty**: Check if the stack is empty

- **IsFull**: Check if the stack is full
- **Peek**: Get the value of the top element without removing it

Working of Stack Data Structure:

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 the stack is already full
6. Before popping, we check if the stack is already empty



Working of Stack Data Structure

Stack Implementations:

```python
# Stack implementation in python


# Creating a stack
def create_stack():
    stack = []
    return stack


# Creating an empty stack
def check_empty(stack):
    return len(stack) == 0


# Adding items into the stack
def push(stack, item):
    stack.append(item)
    print("pushed item: " + item)


# Removing an element from the stack

def pop(stack):
    if (check_empty(stack)):
        return "stack is empty"

    return stack.pop()


stack = create_stack()
push(stack, str(1))
push(stack, str(2))
push(stack, str(3))
push(stack, str(4))
print("popped item: " + pop(stack))
print("stack after popping an element: " + str(stack))
```

Output:

```
pushed item: 1
pushed item: 2
pushed item: 3
pushed item: 4
popped item: 4
stack after popping an element: ['1', '2', '3']
```

Stack Time Complexity:

For the array-based implementation of a stack, the push and pop operations take constant time, i.e. O(1).

Applications of Stack Data Structure:

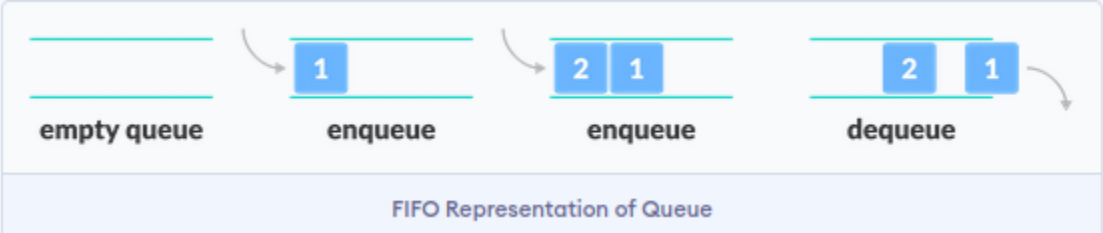
Although stack is a simple data structure to implement, it is very powerful. The most common uses of a stack are:

- **To reverse a word** - Put all the letters in a stack and pop them out. Because of the LIFO order of stack, you will get the letters in reverse order.
- **In compilers** - Compilers use the 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 Data Structure:

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.



FIFO Representation of Queue

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

In programming terms, putting items in the queue is called **enqueue**, and removing items from the queue is called **dequeue**.

Basic Operations of Queue:

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

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

Working of Queue:

Queue operations work as follows:

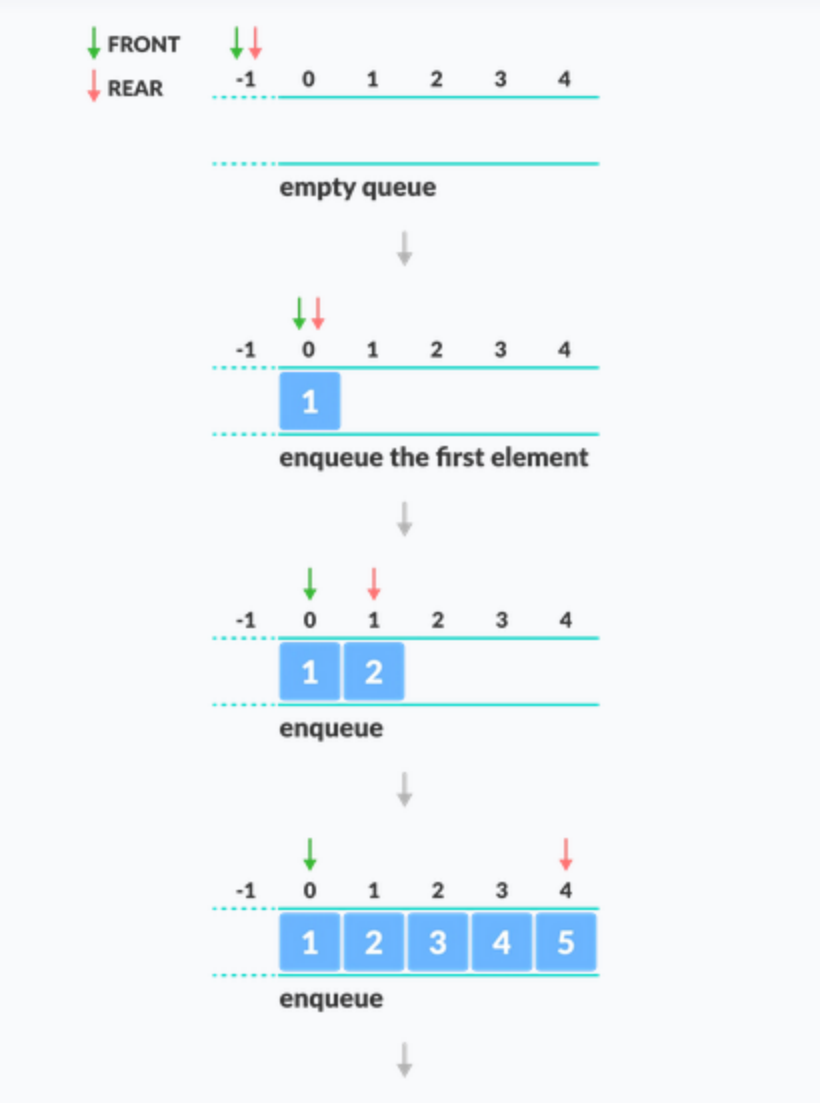
- two pointers *FRONT* and *REAR*
- *FRONT* track the first element of the queue
- *REAR* track the last element of the queue
- initially, set value of *FRONT* and *REAR* to -1

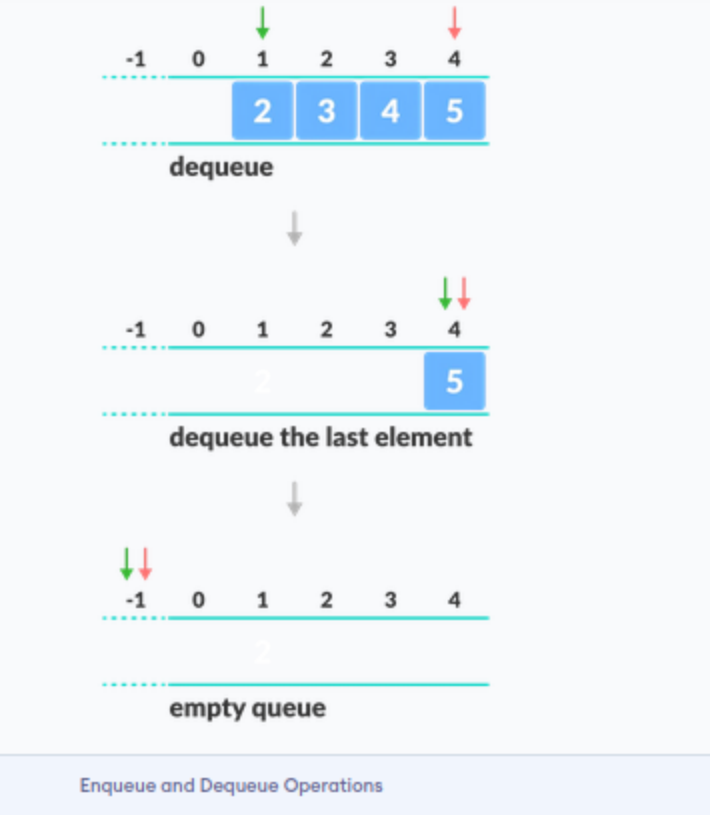
Enqueue Operations:

- check if the queue is full
- for the first element, set the value of *FRONT* to 0
- increase the *REAR* index by 1
- add the new element in the position pointed to by *REAR*

Dequeue Operation:

- check if the queue is empty
- return the value pointed by *FRONT*
- increase the *FRONT* index by 1
- for the last element, reset the values of *FRONT* and *REAR* to -1


FRONT
REAR
-1 0 1 2 3 4
empty queue
-1 0 1 2 3 4
1
enqueue the first element
-1 0 1 2 3 4
1 2
enqueue
-1 0 1 2 3 4
1 2 3 4 5
enqueue



Enqueue and Dequeue Operations

Queue Implementations:

```python
# Queue implementation in Python

class Queue:

    def __init__(self):
        self.queue = []

    # Add an element
    def enqueue(self, item):
        self.queue.append(item)

    # Remove an element
    def dequeue(self):
        if len(self.queue) < 1:
            return None
        return self.queue.pop(0)

    # Display  the queue
    def display(self):
        print(self.queue)

    def size(self):
        return len(self.queue)


q = Queue()
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
q.enqueue(4)
q.enqueue(5)

q.display()

q.dequeue()

print("After removing an element")
q.display()
```

Output:

```
[1, 2, 3, 4, 5]
After removing an element
[2, 3, 4, 5]
```

The complexity of enqueue and dequeue operations in a queue using an array is O(1).

Applications of Queue:

- CPU scheduling, Disk Scheduling
- When data is transferred asynchronously between two processes. The queue is used for synchronization. For example: IO Buffers, pipes, file IO, etc
- Handling of interrupts in real-time systems.
- Call Center phone systems use Queues to hold people calling them in order.