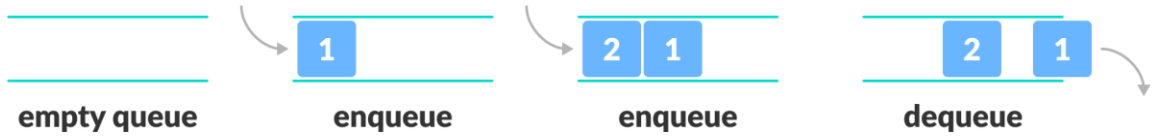
QUEUE DATA STRUCTURE

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.

We can implement the queue in any programming language like C, C++, Java, Python or C#, but the specification is pretty much the same.

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:

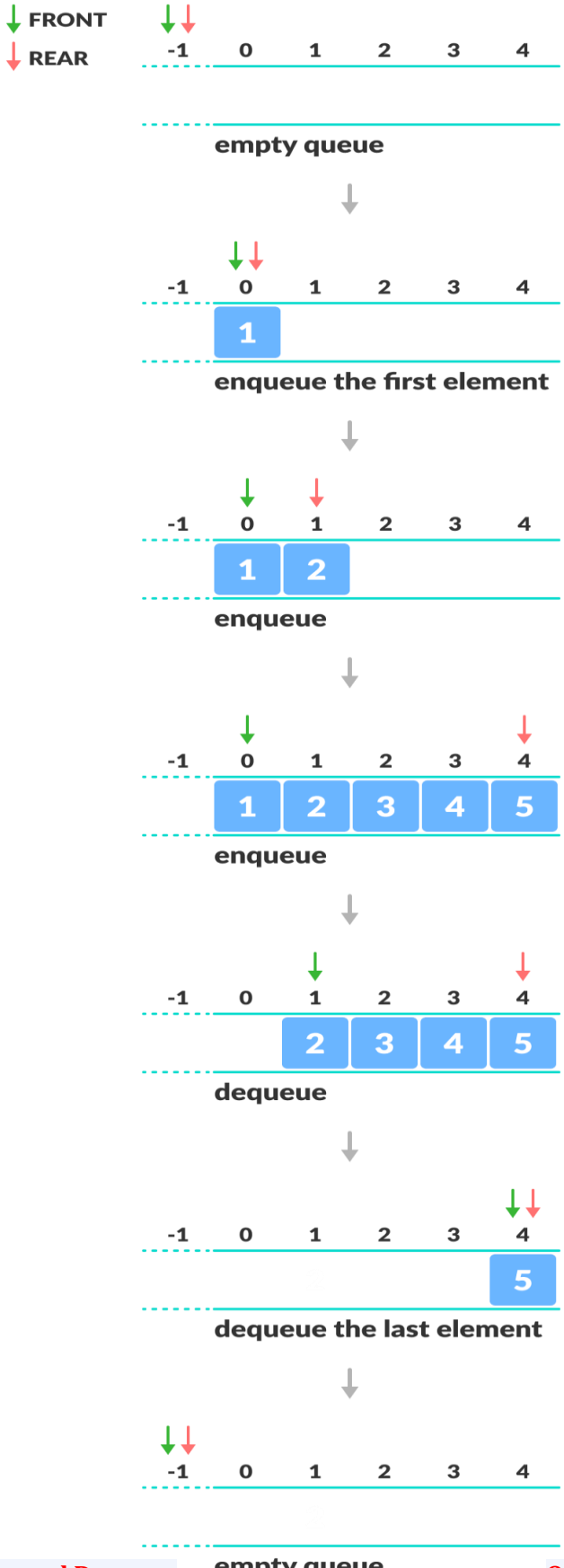
- two pointers FRONT and REAR
- FRONT track the first element of the queue
- REAR track the last element of the queue
- initially, set value of FRONT and REAR to -1

Enqueue Operation

- 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



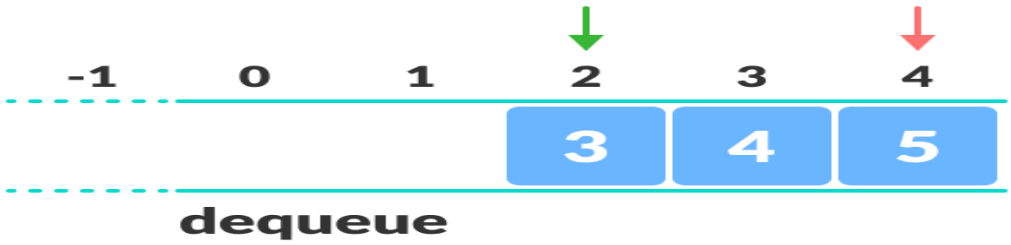
Enqueue and Dequeue Operations

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()
```

LIMITATIONS OF QUEUE

As you can see in the image below, after a bit of enqueuing and dequeuing, the size of the queue has been reduced. the empty spaces at front cannot be used after dequeing from a full queue



Limitation of a queue

And we can only add indexes 0 and 1 only when the queue is reset (when all the elements have been dequeued).

After REAR reaches the last index, if we can store extra elements in the empty spaces (0 and 1), we can make use of the empty spaces. This is implemented by a modified queue called the circular queue.

COMPLEXITY ANALYSIS

The complexity of enqueue and dequeue operations in a queue using an array is O(1). If you use pop(N) in python code, then the complexity might be O(n) depending on the position of the item to be popped.

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.