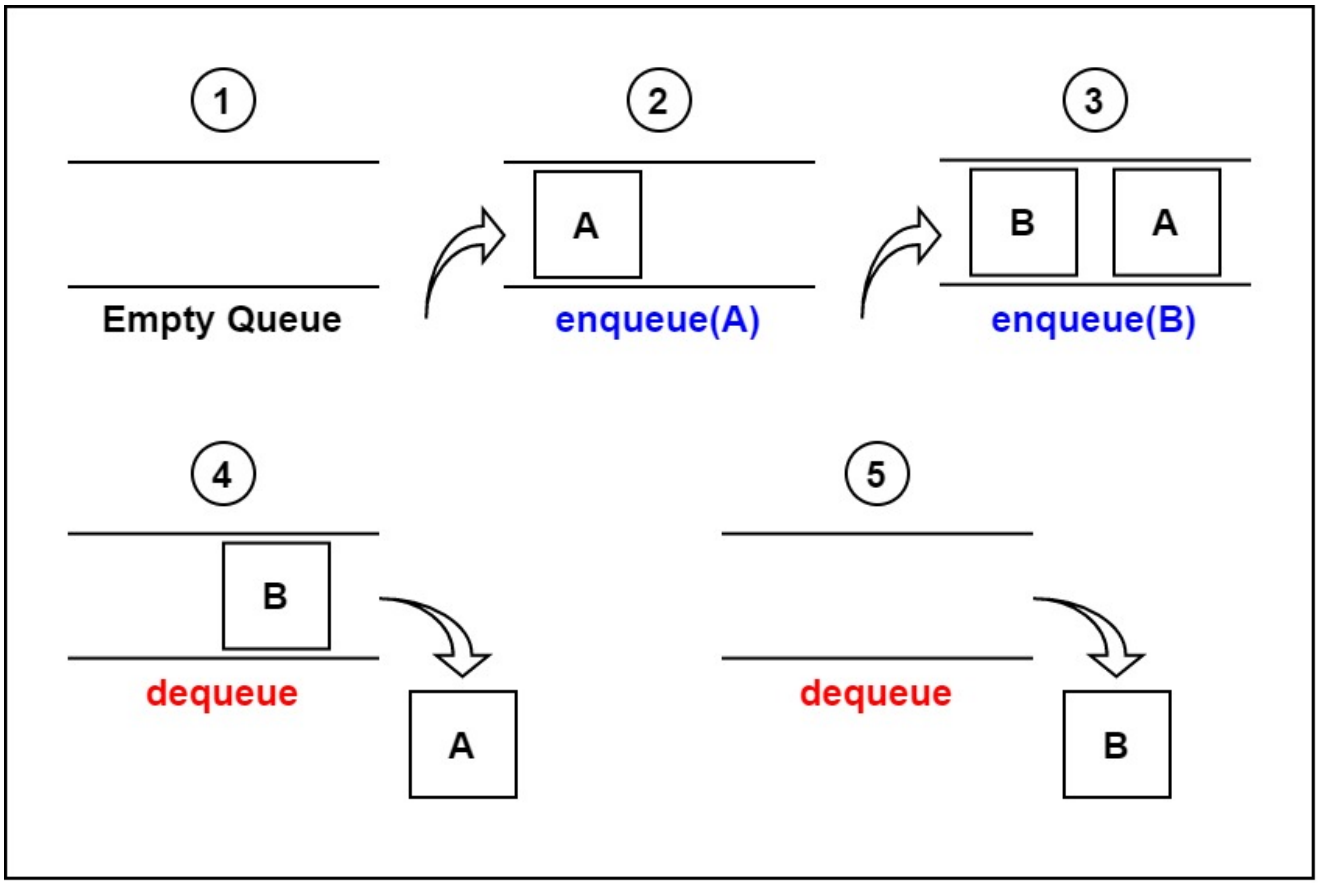
Queue Data Structure

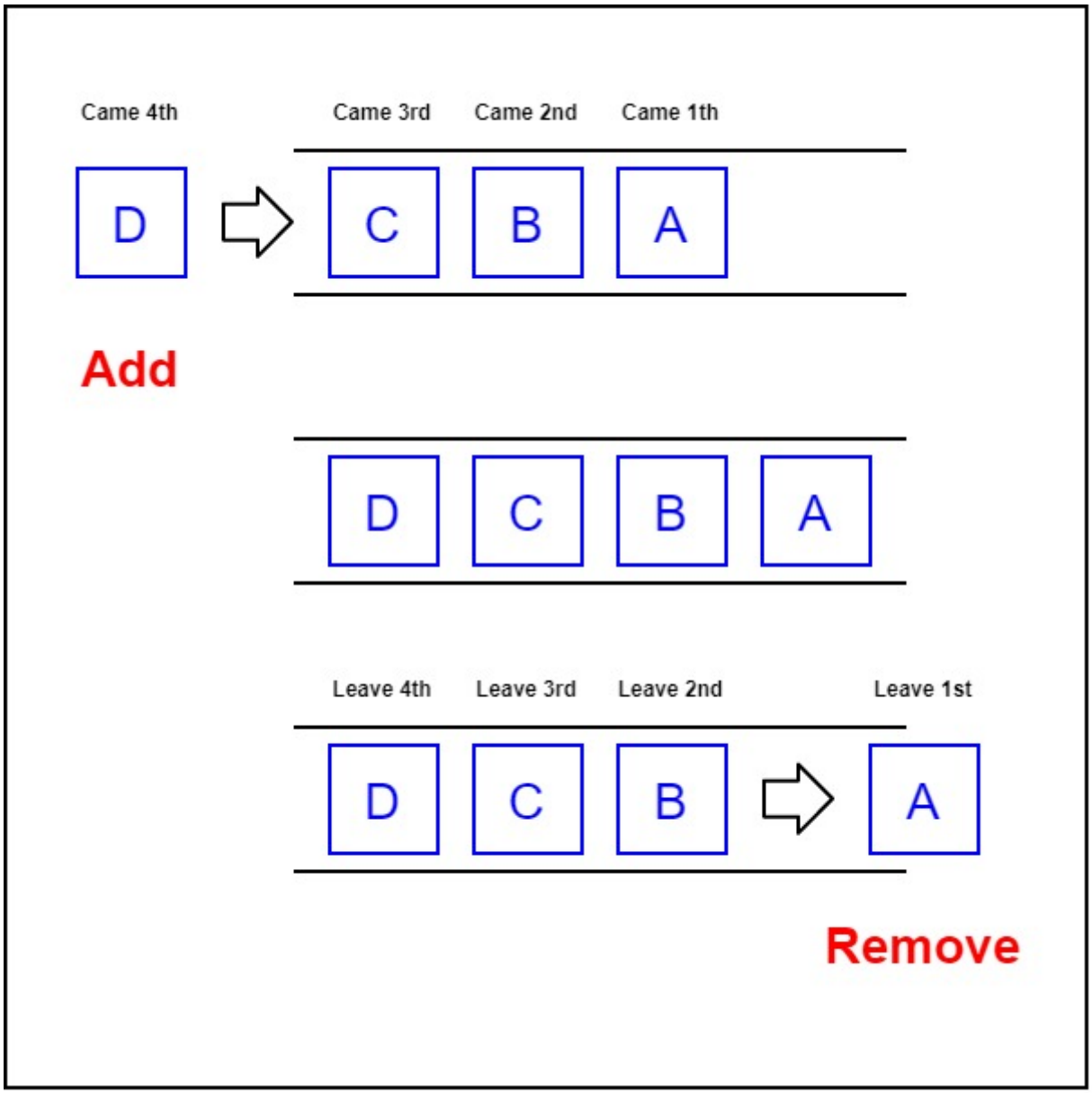
Alright, everyone, today we're going to explore an important data structure in programming called a "queue." You can think of a queue as being similar to the line or queue you see outside a cinema hall. In this queue, the first person who enters is also the first one to get the ticket.

Now, the queue operates on the principle of **First In First Out (FIFO)**. This means that the item that is added to the queue first will be the one that comes out first when we start removing elements from it. Just like how the first person in the cinema ticket queue gets the ticket first.

So, with queues, we always maintain the order of items as they are added, and when we start processing them, we follow the same order in which they were added. This FIFO rule makes queues really handy in various programming scenarios. Let's dive deeper into how queues work and what makes them so useful.



Alright, let's take a closer look at the image provided. As you can see, we have a queue with the letters A and B in that order. Now, since A was added to the queue before B, it will also be the first one to be removed from the queue, following the **First In First Out (FIFO)** rule.



In programming, we have specific terms for these actions. When we add an item to the queue, we call it **enqueue**, and when we remove an item from the queue, we call it **dequeue**.

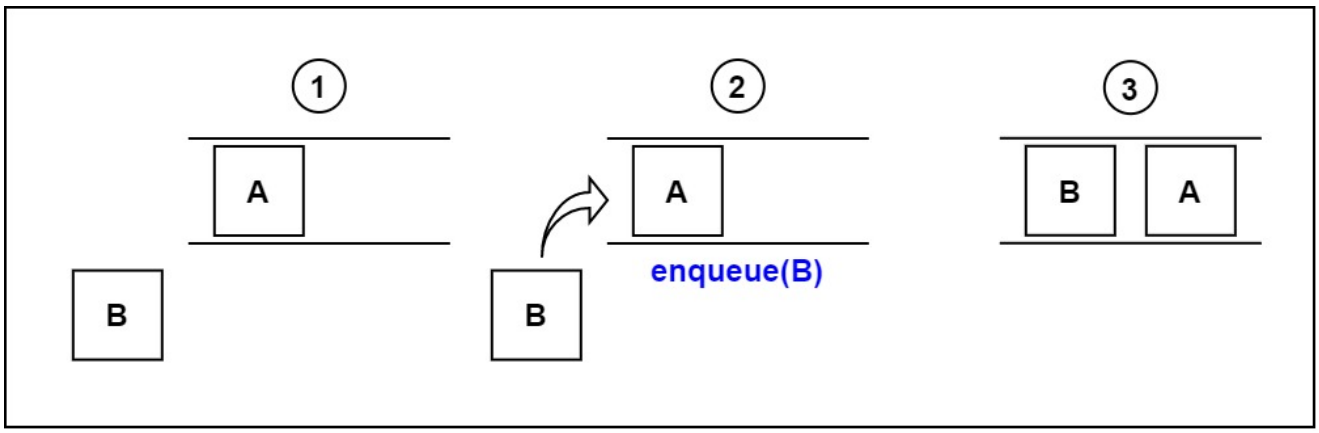
Now, the great thing about queues is that we can implement them in various programming languages like C, C++, Java, Python, or C#. The fundamental specifications for queues remain pretty much the same regardless of the language we use. This makes them versatile and widely applicable in different scenarios. So, let's explore more about how we can use queues and the various operations we can perform on them.

Basic Operations of Queue

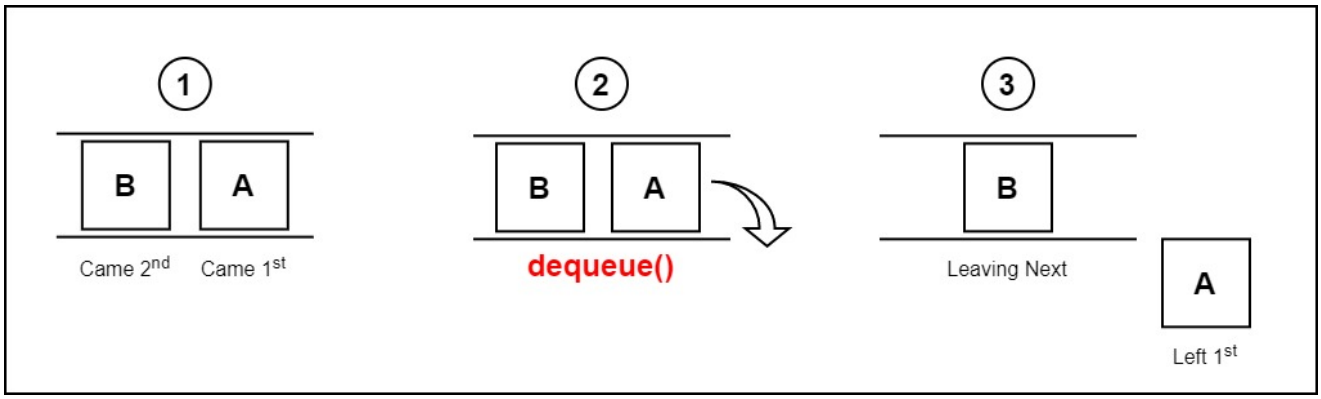
A queue allows us to perform several operations to manage our data effectively.

- The first operation we'll look at is called **Enqueue**. When we enqueue an element, it means we are adding it to the end of the queue. Think of it like standing in a line, and

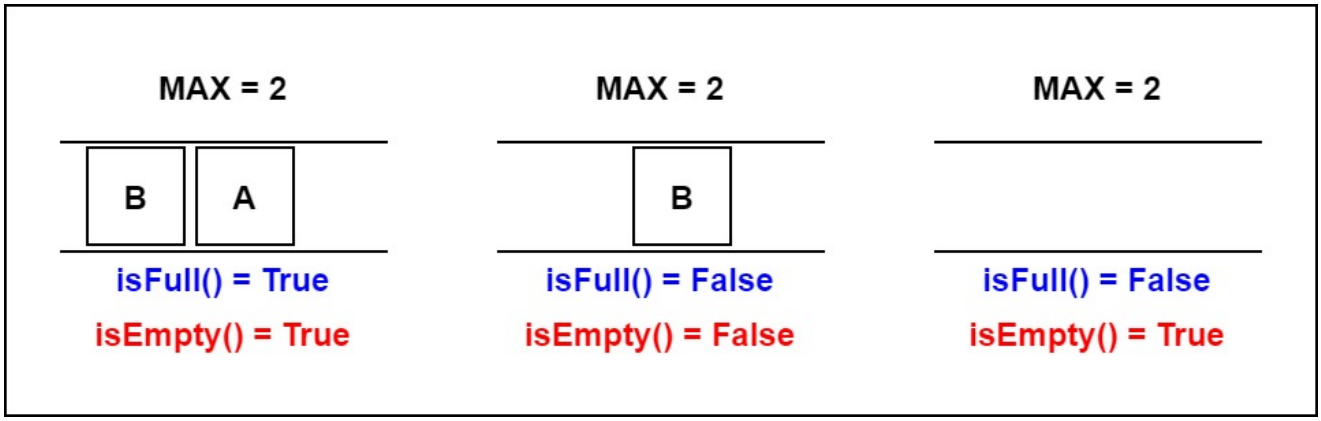
you join the back of the line.



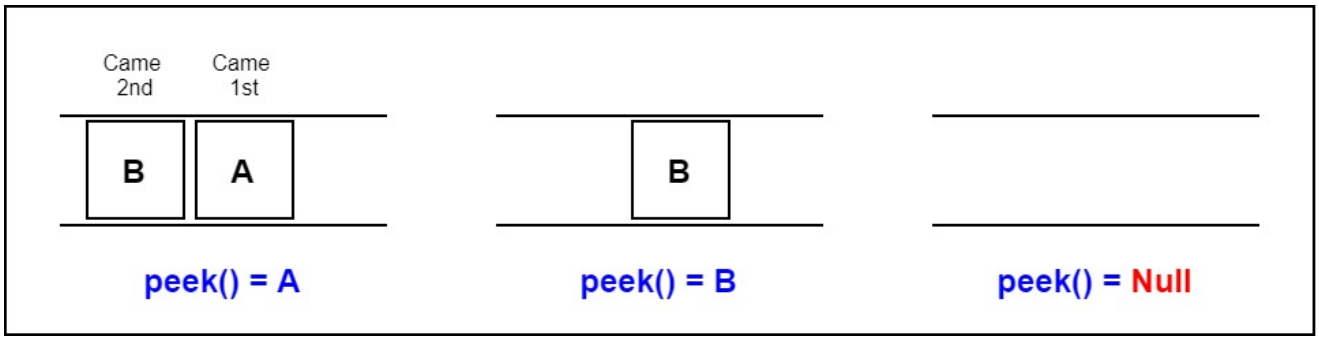
- Next, we have the **Dequeue** operation. When we dequeue an element, we are removing it from the front of the queue. Picture yourself at the front of the line, and it's your turn to move forward and exit the queue.



- To make our lives easier, we can check if the queue is empty or full using the **IsEmpty** and **IsFull** operations, respectively. It's like looking at the queue to see if there are people waiting or if it's already at maximum capacity.



- Lastly, we have the **Peek** operation. This operation allows us to take a sneak peek at the front of the queue without actually removing the element. It's like quickly checking who is at the front of the line without asking them to step out.

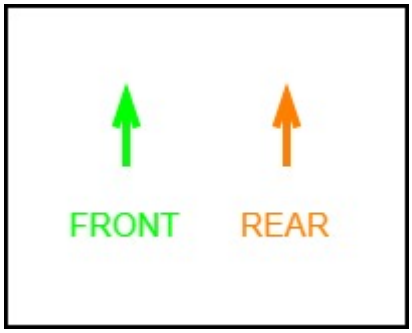


With these operations at our disposal, we can efficiently manage data using a queue in various programming languages. Whether it's C, C++, Java, Python, or C#, the basic functionality of a queue remains quite similar. So, let's continue exploring more about queues and how they can be used to solve different problems.

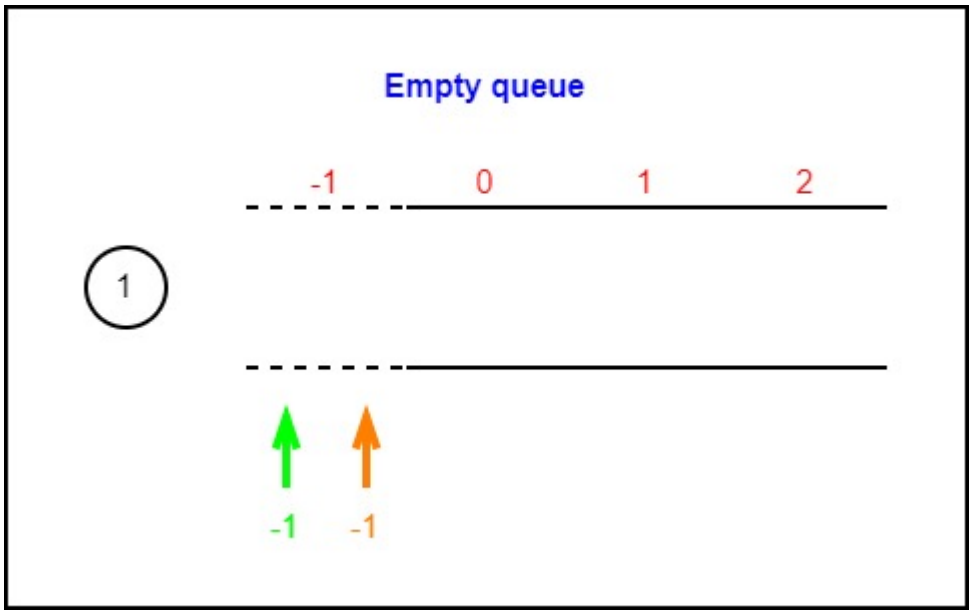
Working of Queue

Alright, students, let's dive deeper into how the queue operations work.

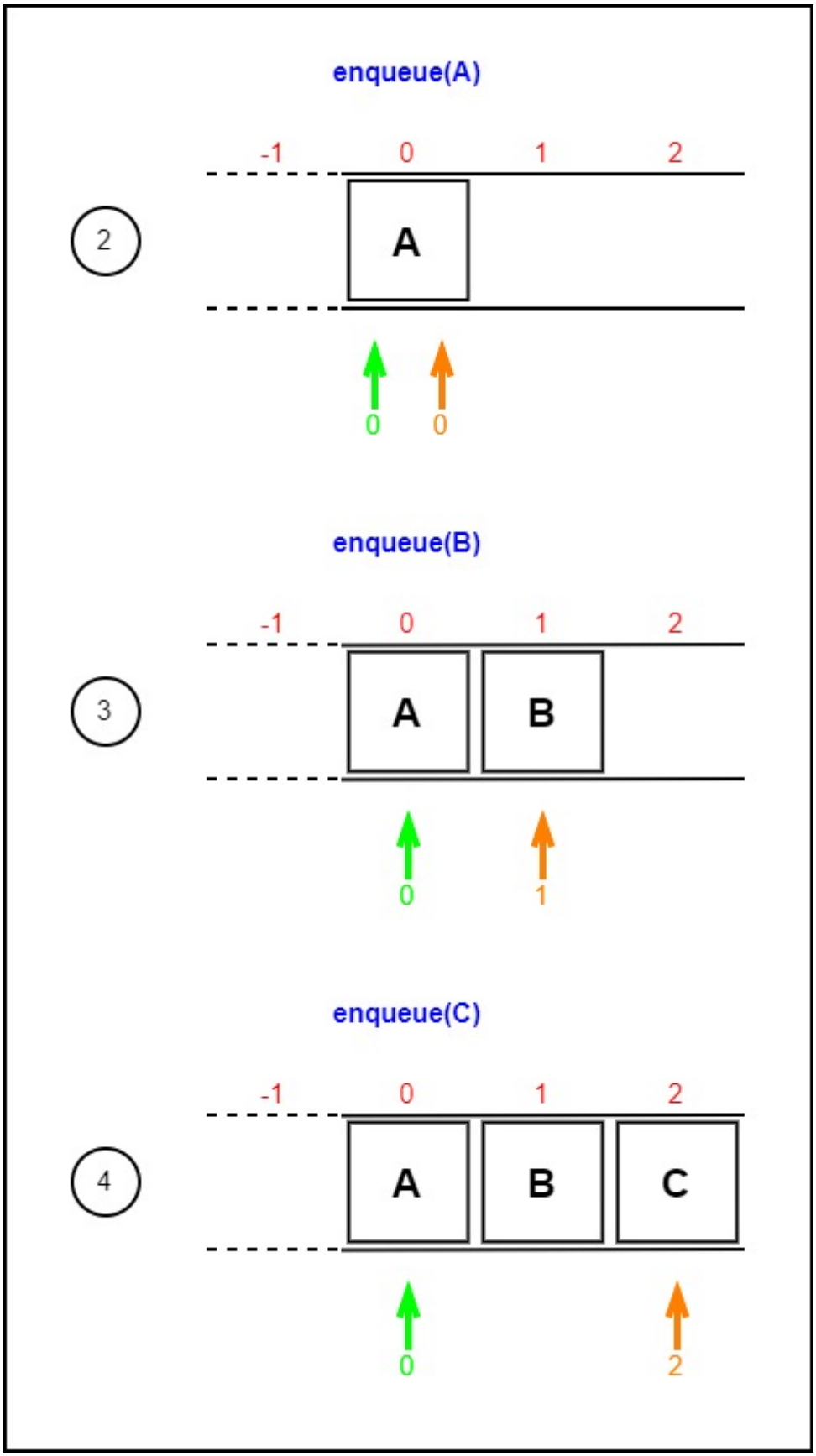
We use two special pointers in a queue called **FRONT** and **REAR**. These pointers help us keep track of the first and last elements of the queue.



When we start with an empty queue, both FRONT and REAR are set to -1. It's like saying we haven't added any elements to the queue yet.

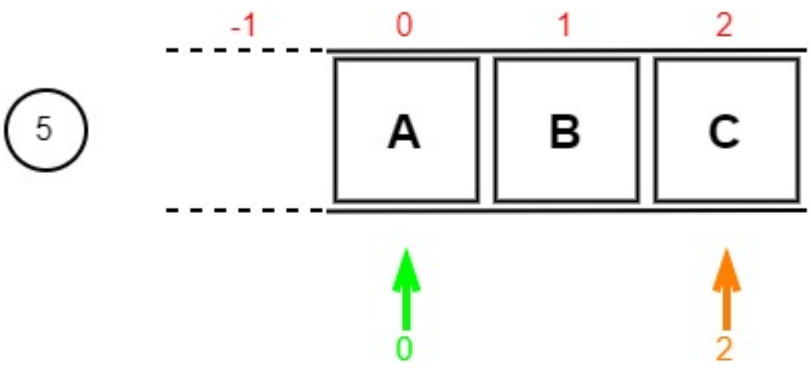


As we enqueue elements into the queue, REAR gets updated to point to the new element we just added to the end. Think of REAR as a moving pointer that always points to the last element.

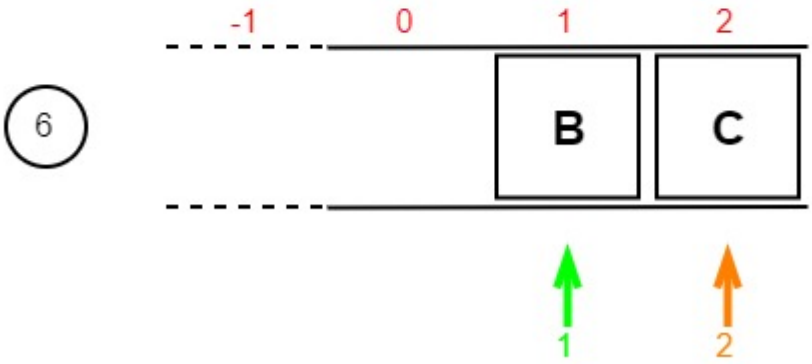


And as we dequeue elements from the queue, FRONT gets updated to point to the next element in line. FRONT acts like a moving pointer that always points to the first element.

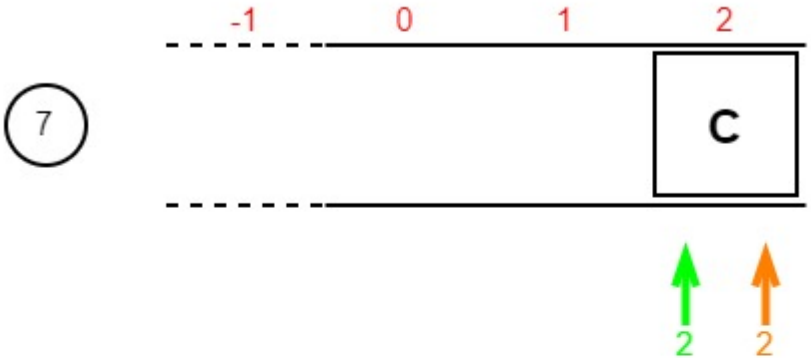
Full queue



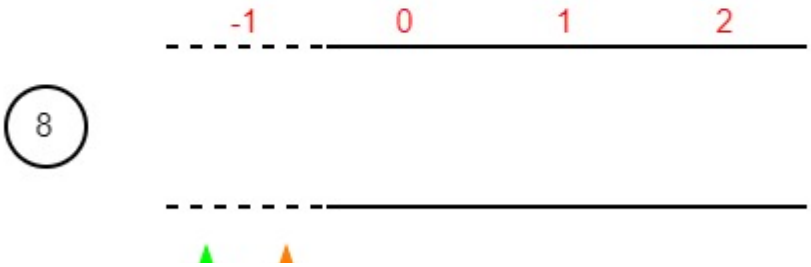
dequeue()



dequeue()



dequeue(C)



So, using these FRONT and REAR pointers, we can efficiently manage the elements in our queue. As we add new elements, REAR moves forward, and as we remove elements, FRONT moves forward. This way, we can maintain the order in which elements were added, following the FIFO rule.

Enqueue Operation

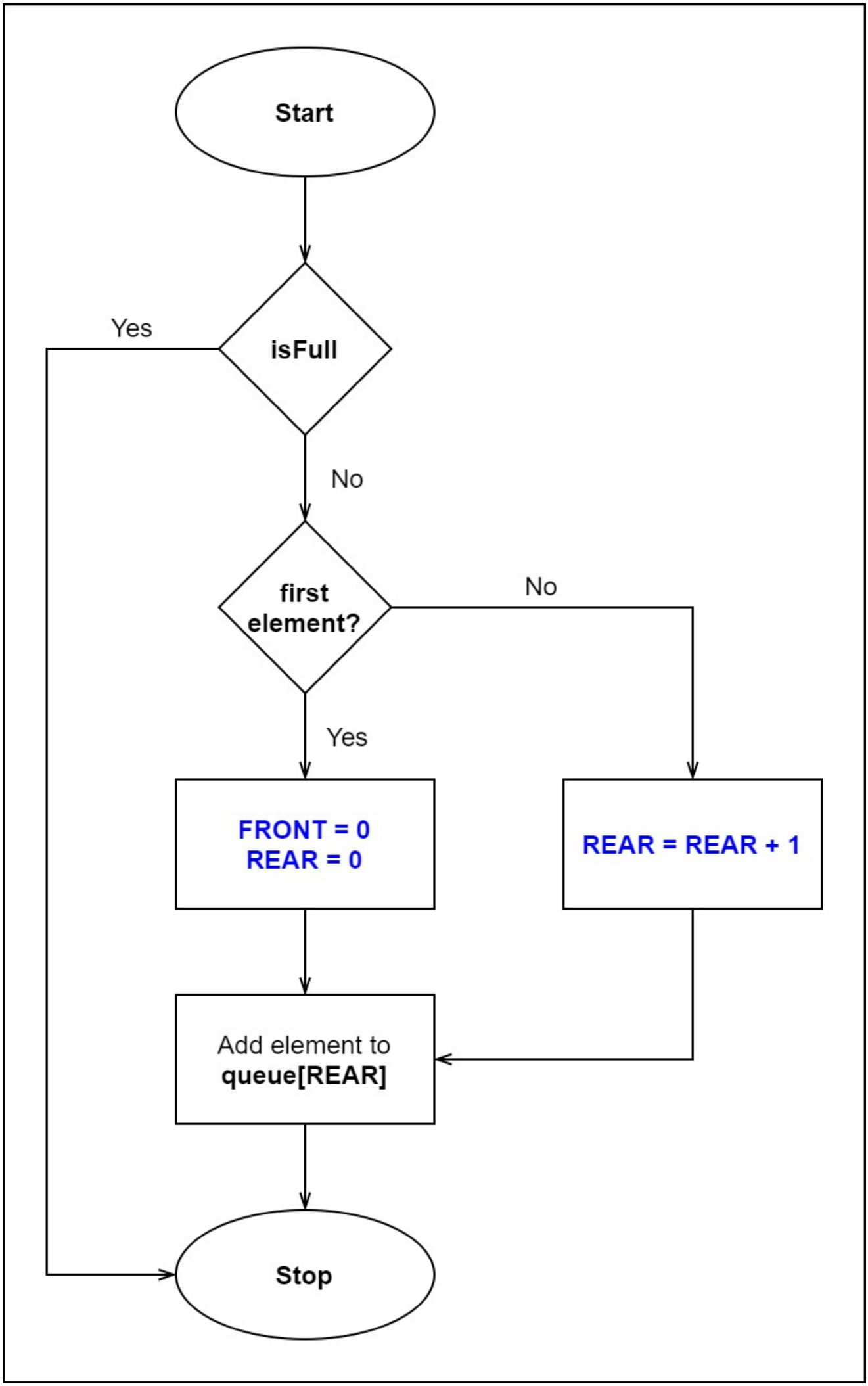
Alright, let's now understand how the **Enqueue** operation works in a queue.

Before we add an element to the queue, we need to check if the queue is already full. We don't want to add more elements than the queue can handle!

Now, when we add the very first element to the queue, we set the value of **FRONT** to 0. This is because we have just one element, and it's both the first and last element in the queue.

Next, we increase the **REAR** index by 1. It's like saying, "Hey, we have a new element, and it's going to be at the end of the queue!"

Finally, we add the new element to the position pointed to by **REAR**. So now, we have successfully added an element to the end of the queue, and our queue has grown by one element. That's how the **Enqueue** operation works!


Start
isFull
Yes
No
first element?
No
Yes
FRONT = 0
REAR = 0
REAR = REAR + 1
Add element to queue[REAR]
Stop

Dequeue Operation

Now, let's learn about the **Dequeue** operation in a queue.

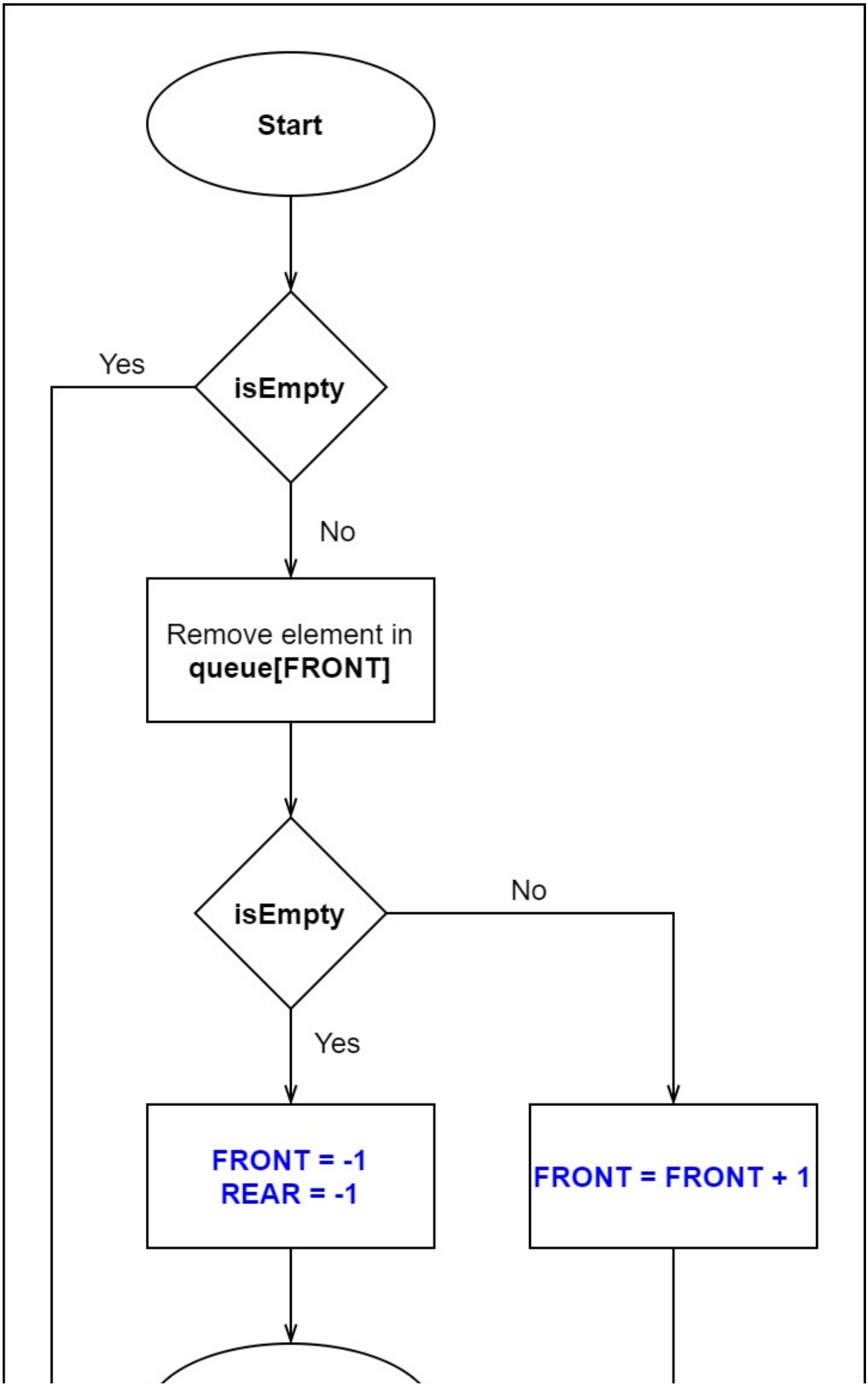
Before we remove an element from the queue, we need to check if the queue is empty. We don't want to remove elements when there's nothing in the queue!

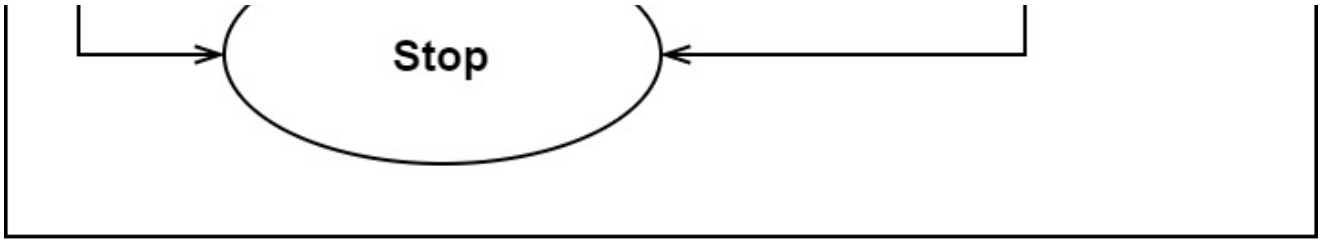
If the queue is not empty, we proceed with the removal process. The first element in the queue is the one pointed by **FRONT**. So, we return the value that **FRONT** is pointing to, and that's the element we're removing from the queue.

After removing the element, we increase the **FRONT** index by 1. It's like saying, "The first element is gone, so now the next element will be the first."

Now, there's one special case we need to handle. If the last element is removed from the queue, we need to reset the values of both **FRONT** and **REAR** to -1. This is because if both pointers are set to -1, it indicates that the queue is empty.

And there you have it! That's how the **Dequeue** operation works in a queue. It allows us to remove elements from the front of the queue, just like how people in a line come forward to get their turn!


Start
isEmpty
Yes
No
Remove element in
queue[FRONT]
isEmpty
No
Yes
FRONT = -1
REAR = -1
FRONT = FRONT + 1



Queue Implementation in C++

In C++, when we want to implement queues, we often use arrays. Similarly, in Java and C++, arrays are a popular choice for queue implementations. However, if we're working with Python, we use lists for creating queues.

The good news is that the basic concept of a queue remains the same across all these programming languages. Whether we use arrays or lists, the functionality and operations of a queue, like **Enqueue** and **Dequeue**, stay consistent.

So, regardless of which language you choose, you can implement queues to efficiently manage data and follow the **First In First Out (FIFO)** principle. This will help you build robust programs and solve various real-world problems effectively!

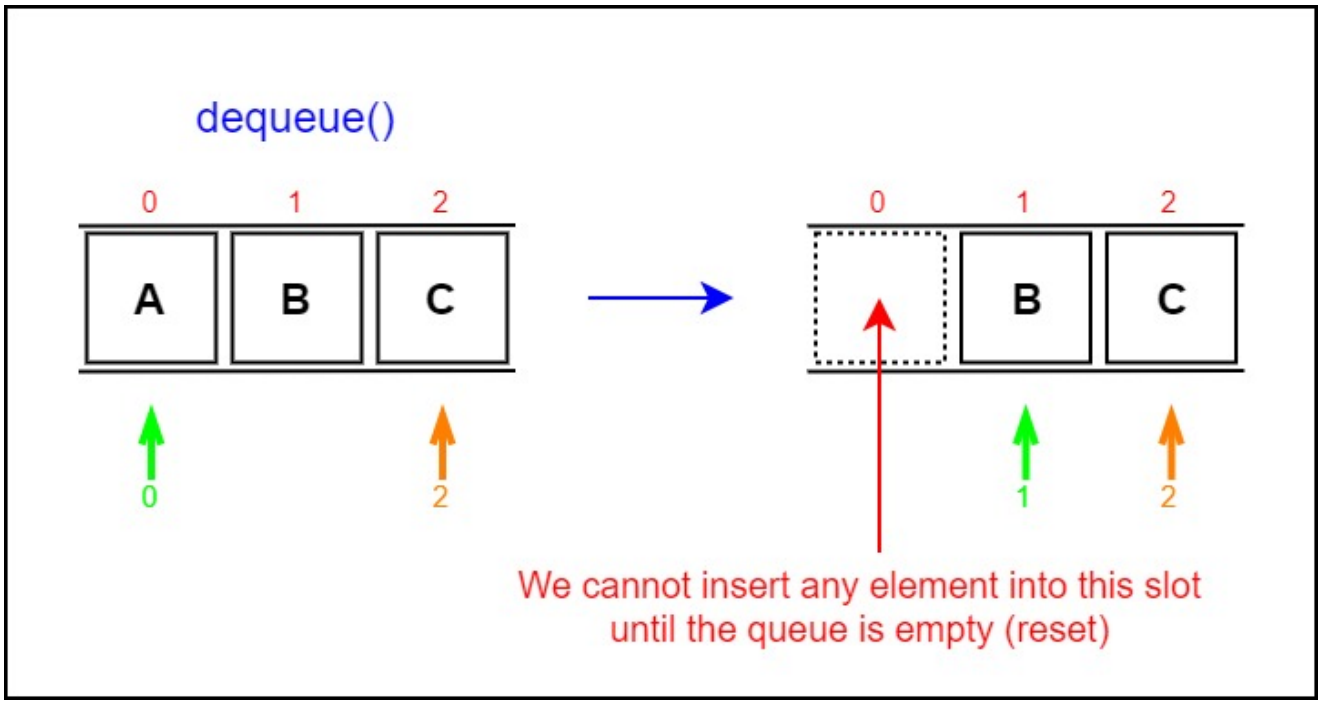
```
// Queue implementation in C++

#include <iostream>
#define SIZE 5

using namespace std;

class Queue {
   private:
  int items[SIZE], front, rear;

   public:
  Queue() {
    front = -1;
    rear = -1;
  }

  bool isFull() {
    if (front == 0 && rear == SIZE - 1) {
      return true;
    }
    return false;
  }

  bool isEmpty() {
    if (front == -1)
      return true;
```

```
  else
    return false;
}

void enQueue(int element) {
  if (isFull()) {
    cout << "Queue is full";
  } else {
    if (front == -1) front = 0;
    rear++;
    items[rear] = element;
    cout << endl
       << "Inserted " << element << endl;
  }
}

int deQueue() {
  int element;
  if (isEmpty()) {
    cout << "Queue is empty" << endl;
    return (-1);
  } else {
    element = items[front];
    if (front >= rear) {
      front = -1;
      rear = -1;
    } /* Q has only one element, so we reset the queue after deleting it. */
    else {
      front++;
    }
    cout << endl
       << "Deleted -> " << element << endl;
    return (element);
  }
}

void display() {
  /* Function to display elements of Queue */
  int i;
  if (isEmpty()) {
    cout << endl
       << "Empty Queue" << endl;
  } else {
    cout << endl
       << "Front index-> " << front;
    cout << endl
       << "Items -> ";
    for (i = front; i <= rear; i++)
      cout << items[i] << "  ";
```

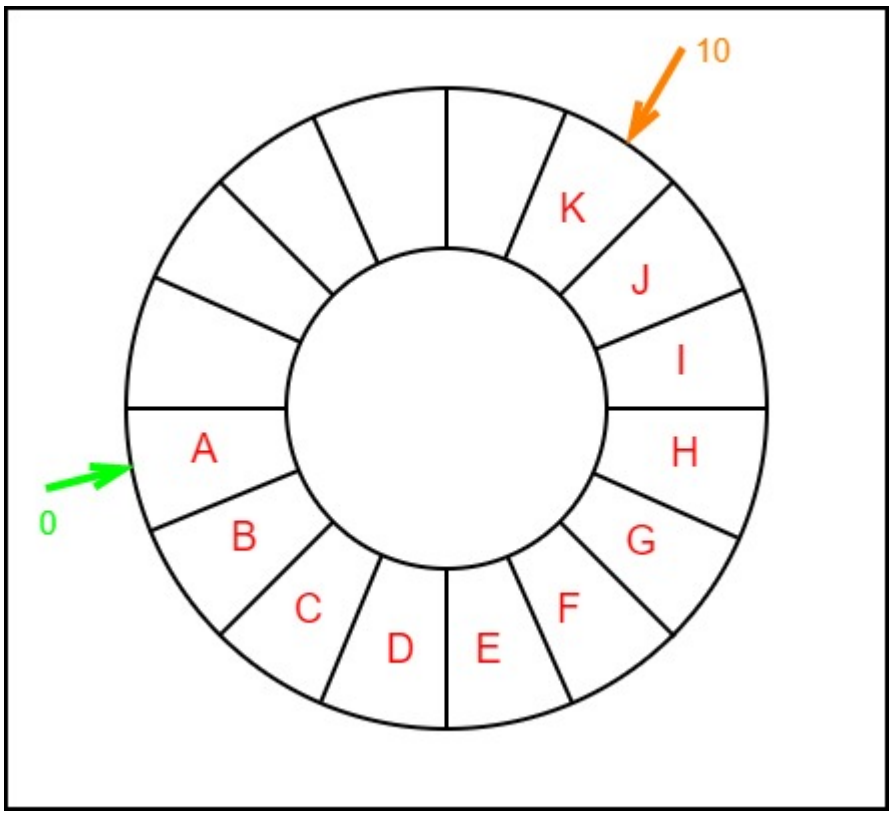
```
      cout << endl
         << "Rear index-> " << rear << endl;
    }
  }
};

int main() {
  Queue q;

  //deQueue is not possible on empty queue
  q.deQueue();

  //enQueue 5 elements
  q.enQueue(1);
  q.enQueue(2);
  q.enQueue(3);
  q.enQueue(4);
  q.enQueue(5);

  // 6th element can't be added to because the queue is full
  q.enQueue(6);

  q.display();

  //deQueue removes element entered first i.e. 1
  q.deQueue();

  //Now we have just 4 elements
  q.display();

  return 0;
}
```

Limitations of Queue

Now, let's discuss some limitations of the standard queue. As you can observe in the image below, after a series of enqueuing and dequeuing operations, the size of the queue has reduced, and we can only add elements at index 0 when the queue is reset (when all the elements have been dequeued).



However, there's a way to make better use of the available space. When the **REAR** reaches the last index of the queue, instead of leaving those empty spaces unused, we can utilize them to store additional elements. This modification is known as the **circular queue**.



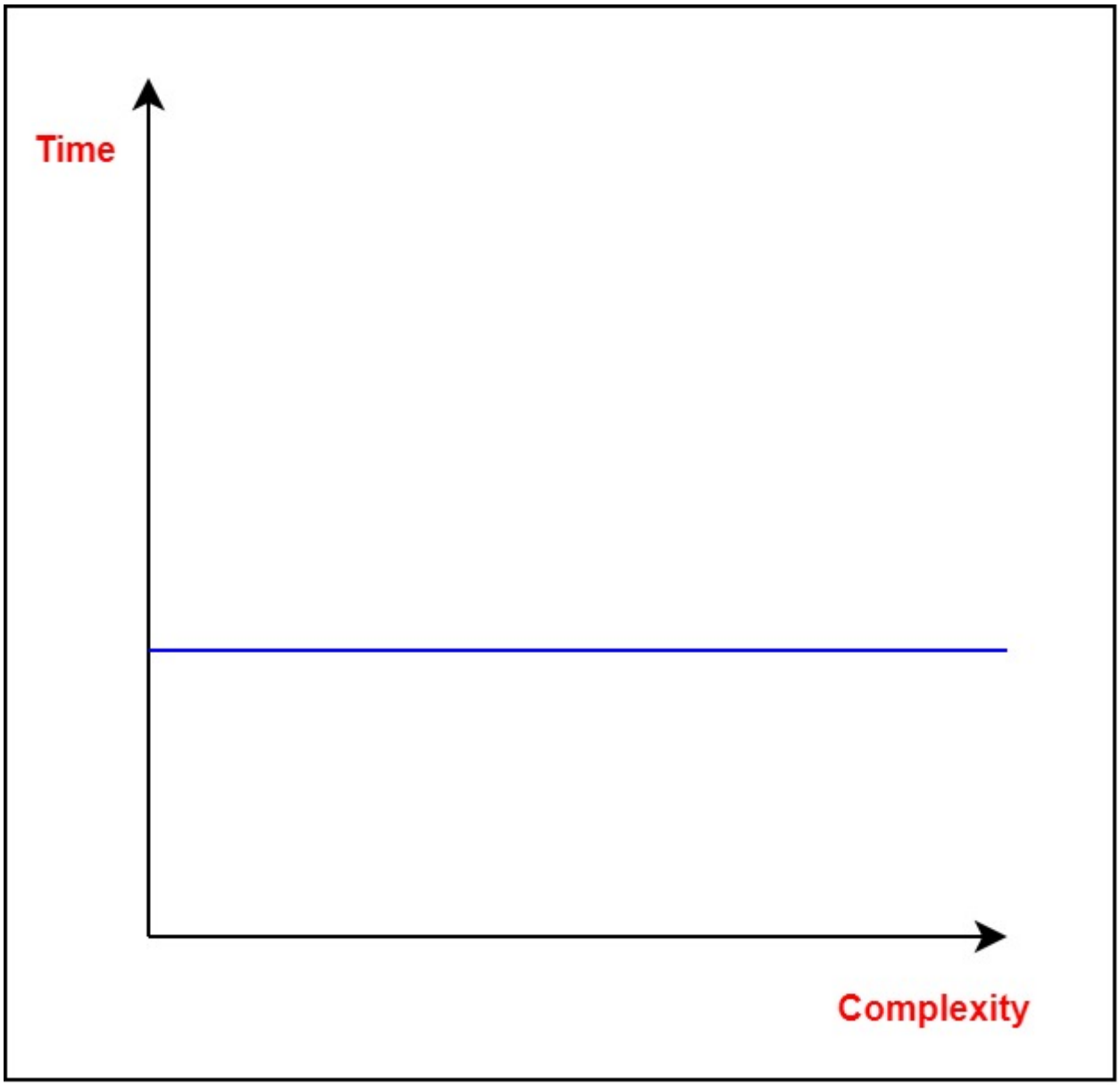
By employing a circular queue, we can optimize the use of space, making it more efficient and practical for certain scenarios. This is a valuable enhancement to the standard queue, and it allows us to manage data more effectively in various applications!

Complexity Analysis

Now, let's discuss the complexity analysis of the enqueue and dequeue operations in a queue that uses an array.

The complexity of the **enqueue operation** is **constant time**, denoted as **O(1)**. It means that the time taken to enqueue an element into the queue doesn't depend on the number of elements already present in the queue. So, adding an element is a fast and efficient operation, regardless of the queue's size.

Similarly, the complexity of the **dequeue operation** is also **constant time**, represented as **O(1)**. This means that removing an element from the front of the queue is a quick process, regardless of the queue's size.

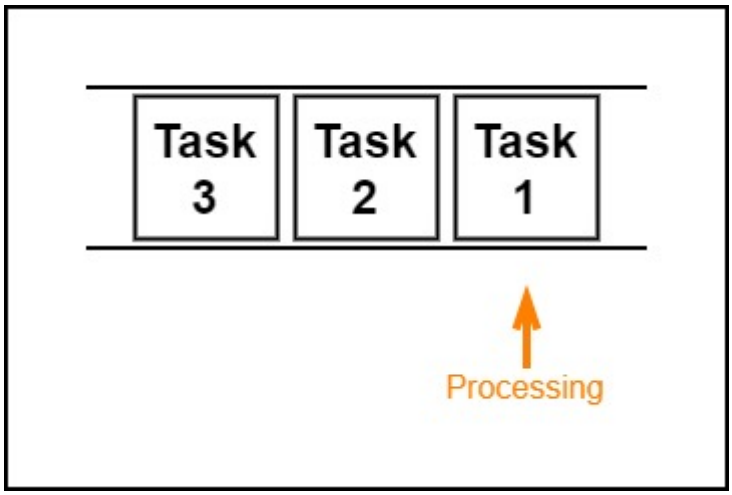


Overall, using an array to implement a queue ensures that both enqueue and dequeue operations are efficient and have constant time complexity, making it a reliable data structure for managing data in various programming scenarios.

Applications of Queue

Let's explore some of the real-world applications where queues come in handy:

1. **CPU Scheduling, Disk Scheduling**: In operating systems, queues are used to manage tasks for the central processing unit (CPU) and to handle disk operations efficiently. Tasks are placed in queues and scheduled for execution based on priority or other algorithms.



2. **Data Transfer and Synchronization**: When data is transferred asynchronously between two processes or devices, queues are used for synchronization. For instance, input/output (IO) buffers, pipes, and file IO often employ queues to manage the flow of data.
3. **Interrupt Handling in Real-Time Systems**: In real-time systems, interrupts need to be managed promptly and efficiently. Queues are used to prioritize and handle interrupts as they occur, ensuring timely responses.
4. **Call Center Phone Systems**: Call centers use queues to manage incoming calls in an organized manner. When people call the center, they are placed in a queue, and calls are answered in the order they were received, following the "first in, first out" principle.

By understanding the various applications of queues, we can appreciate their versatility and importance in solving real-world problems efficiently. Queues enable us to organize and manage data effectively, making them a valuable tool in computer science and various industries.