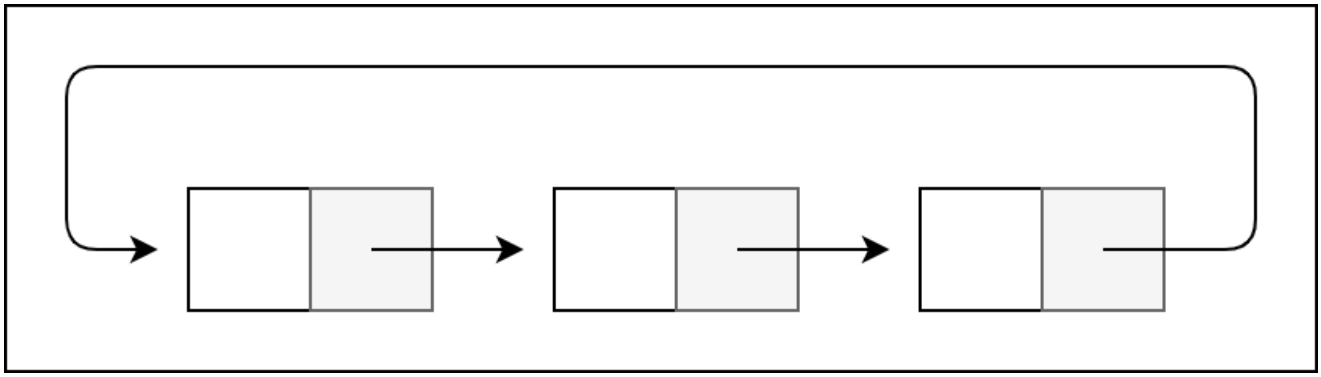
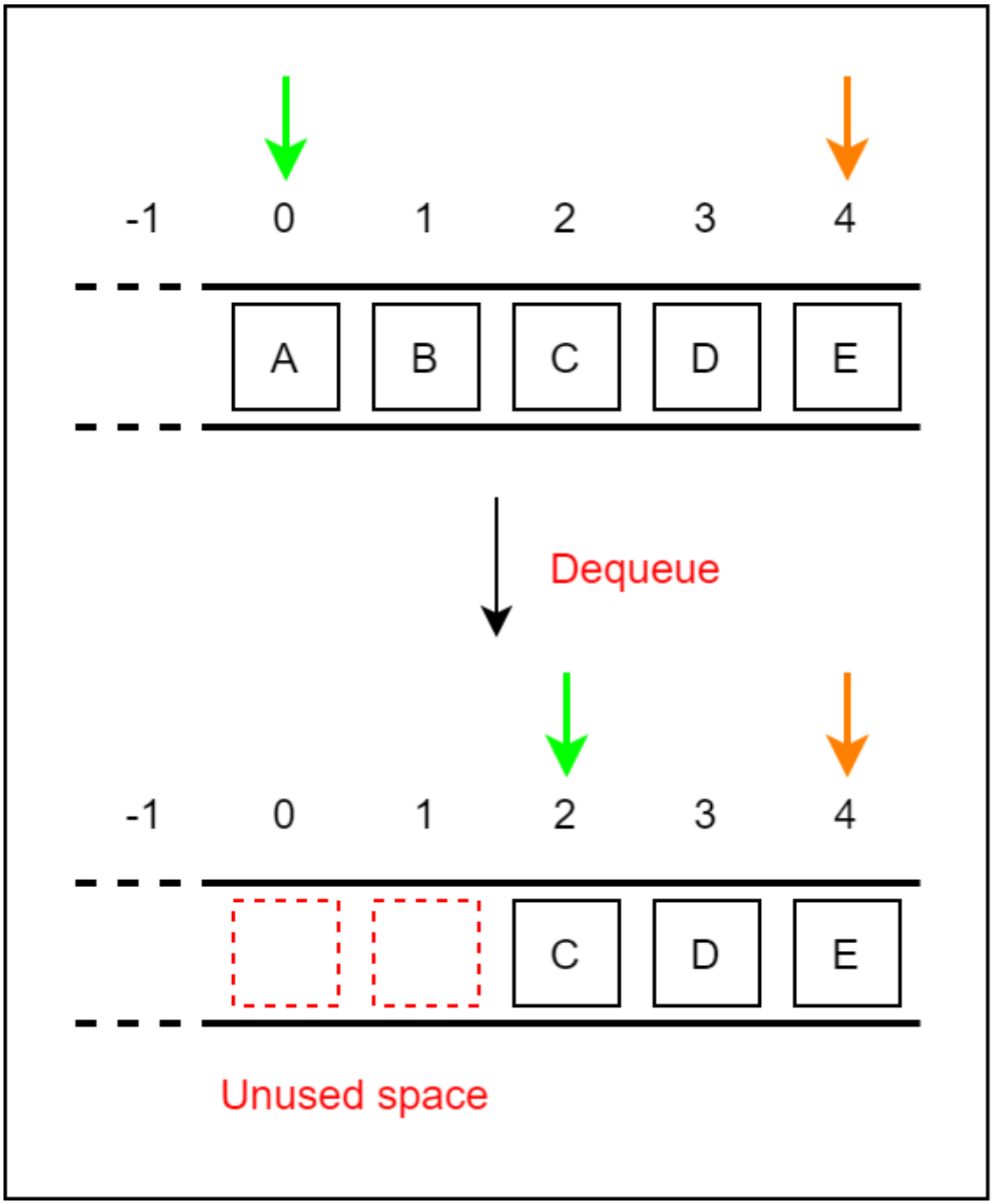
Circular Queue

Alright, let's dive into the concept of the circular queue! It's an extension of the regular queue, but with a twist – the last element is cleverly connected to the first element, creating a circle-like structure.

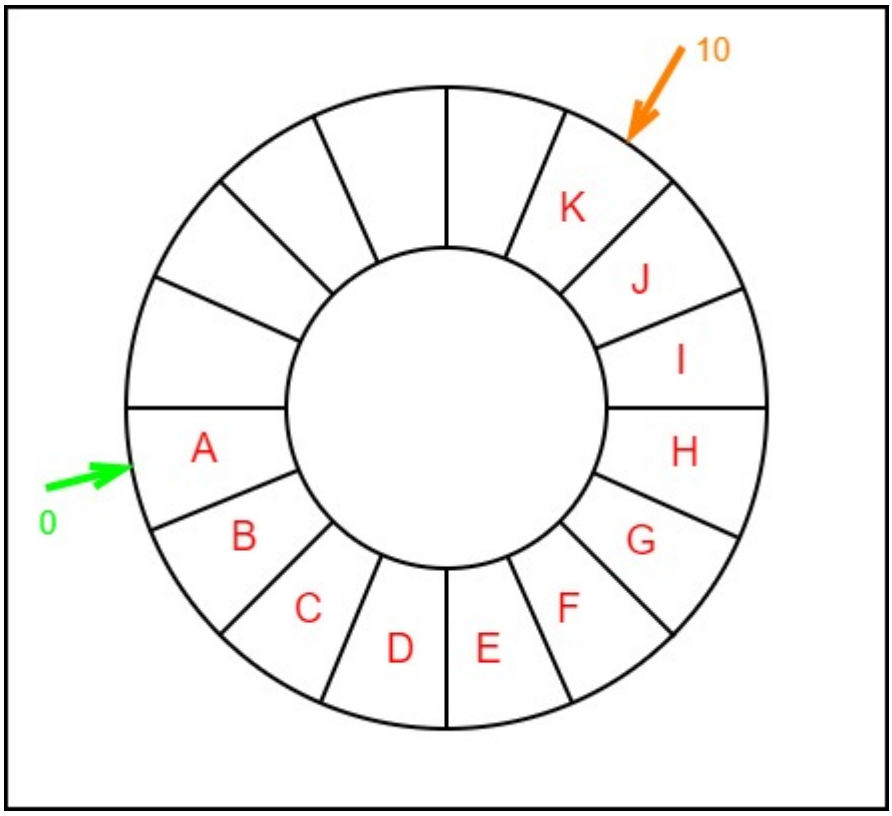


Now, you might wonder, why do we need this circular queue? Well, it solves a significant limitation of the normal queue. In a regular queue, after some insertions and deletions, we end up with non-usable empty spaces, and that's not very efficient.



But, fear not! The circular queue comes to the rescue! By connecting the last element back to the first one, we eliminate those pesky non-usable empty spaces. And guess what? This means that we can use indexes **0** and **1** again, without having to reset the entire queue by deleting all elements. Isn't that neat?

Thanks to this clever circular structure, we can make the most out of our queue, and it becomes a much more versatile and memory-efficient data structure. So, whether you're managing data in a computer program or designing cool games, the circular queue has got your back!



How Circular Queue Works

Great question! Let's understand how the Circular Queue works and what this "circular increment" means.

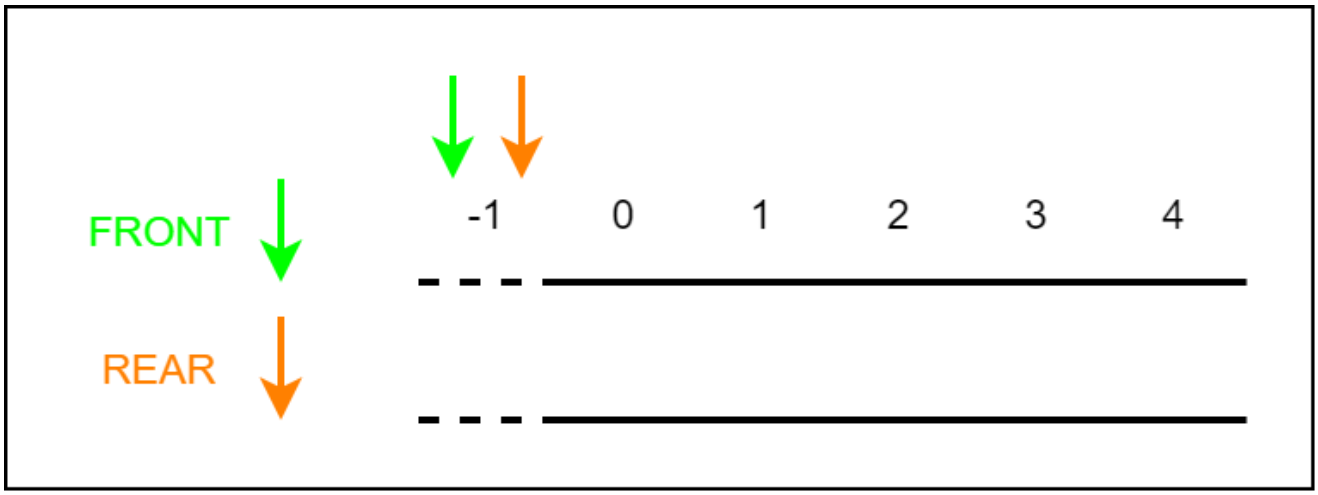
Imagine we have a Circular Queue with five positions, numbered from 0 to 4. When we insert elements into the queue, we move the **REAR** pointer to the next position to accommodate the new element. However, there's a special trick when the **REAR** pointer reaches the end of the queue (position 4 in our case).

Instead of causing an overflow error and stopping there, the Circular Queue cleverly "wraps around" to the beginning of the queue, position 0. It's like completing a circle! This process of "wrapping around" is called "circular increment."

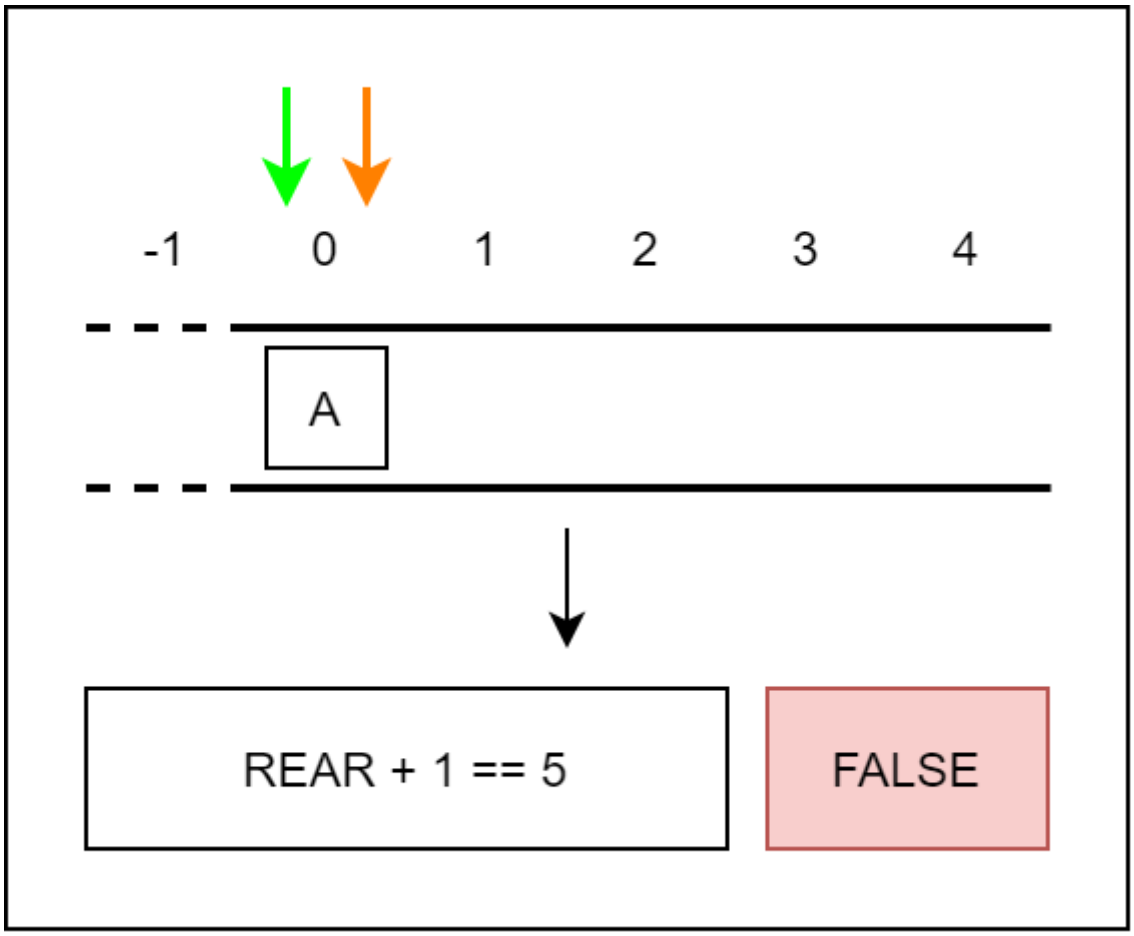
```
if REAR + 1 == 5 (overflow!), REAR = (REAR + 1) % 5 = 0 (start of queue)
```

Let's see it in action with an example:

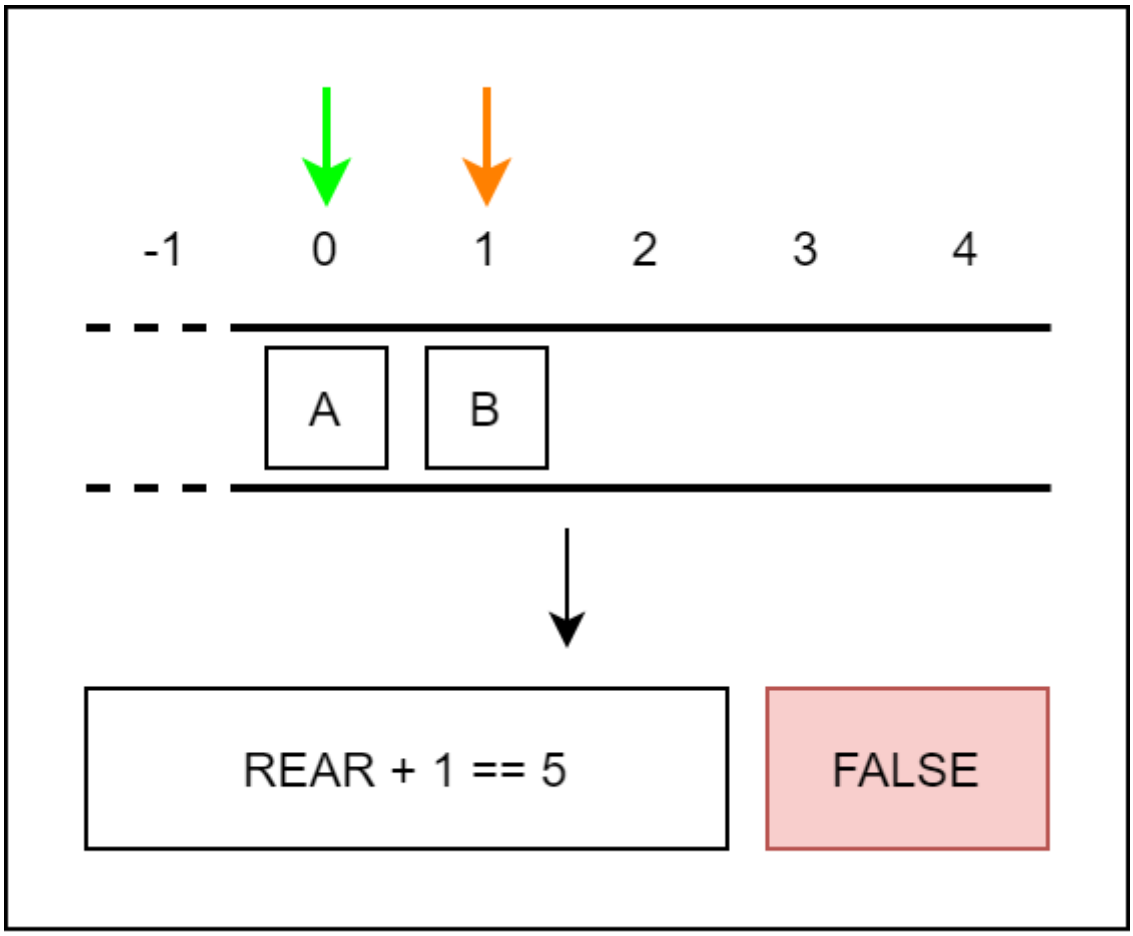
Suppose our Circular Queue is empty, and we want to insert an element at the **REAR** position, which is initially at index 0.



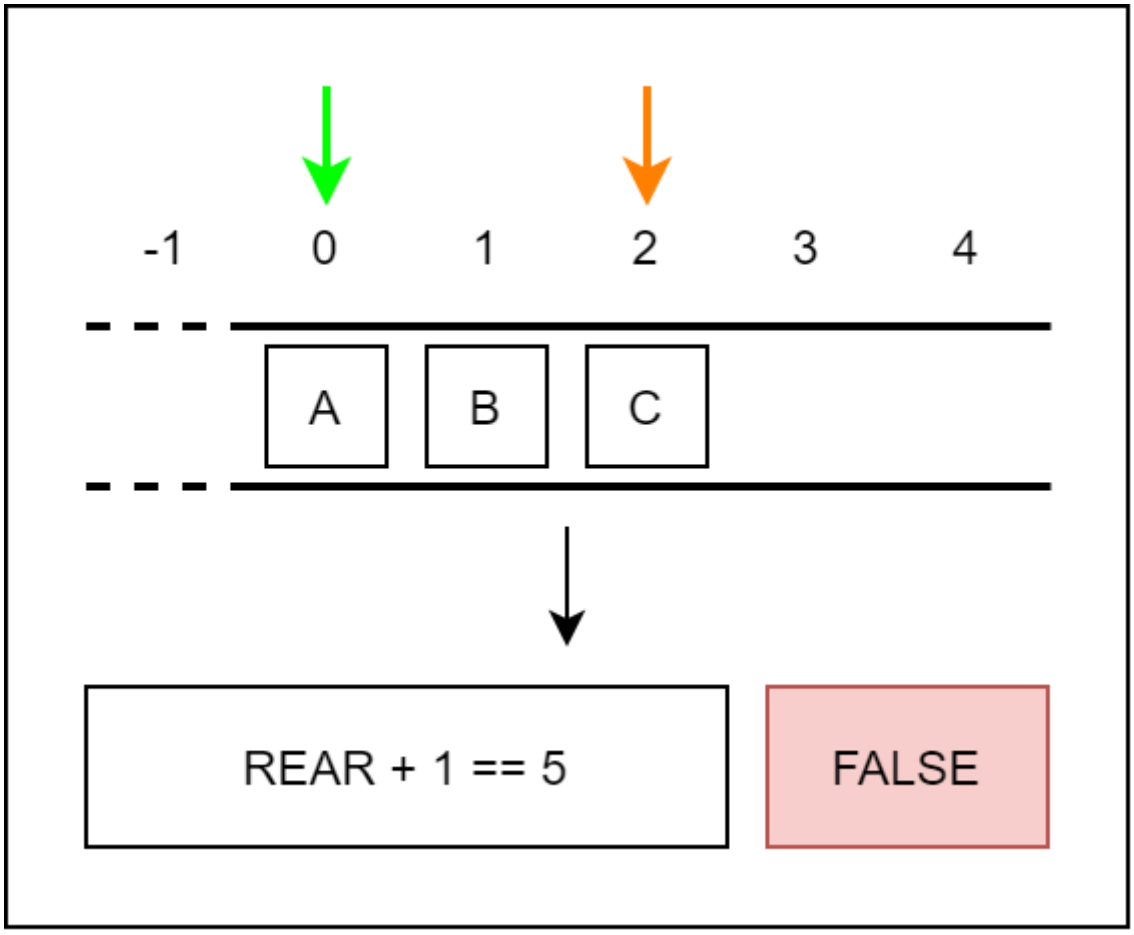
1. We insert an element at index 0, and **REAR** becomes 1. (Queue: [A, , , ,])



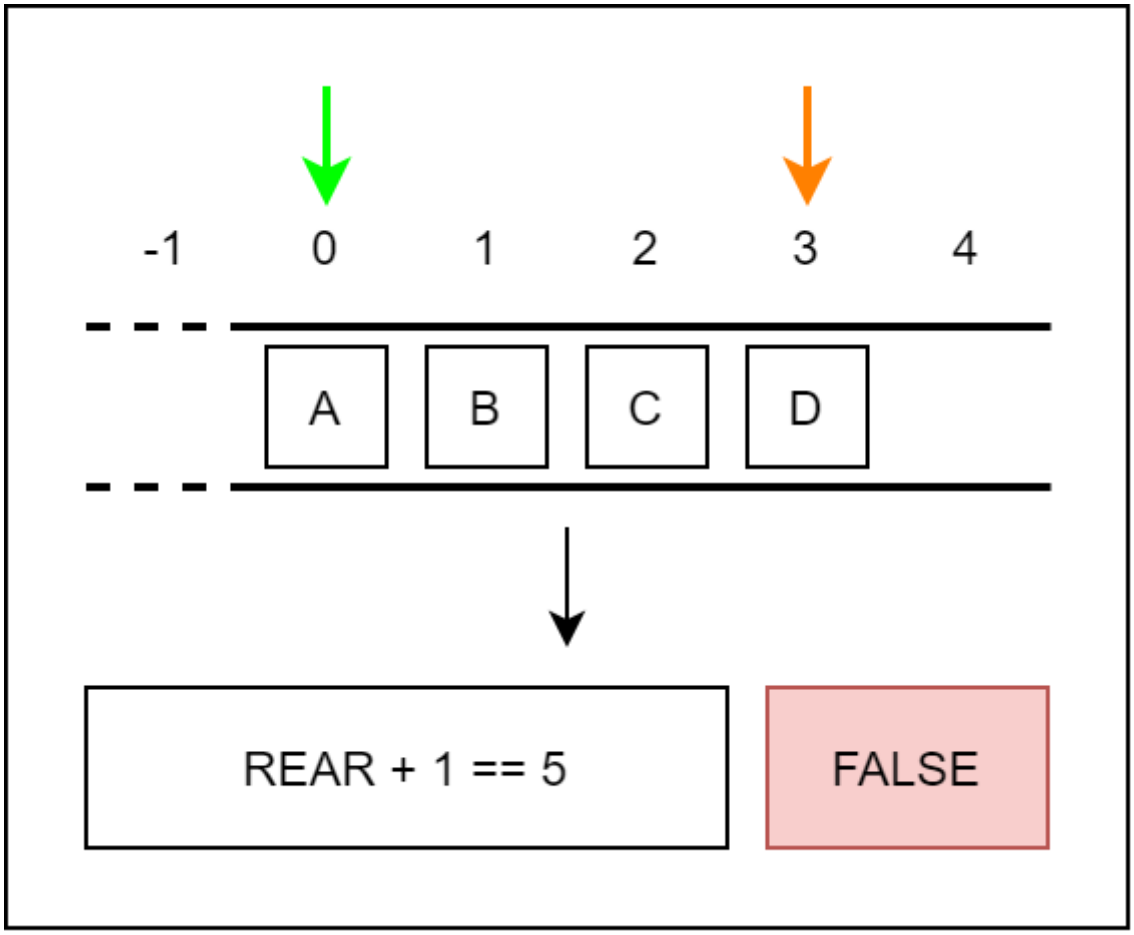
2. We insert another element at index 1, and **REAR** becomes 2. (Queue: [A, B, , , _])



3. We insert one more element at index 2, and **REAR** becomes 3. (Queue: [A, B, C, ,])

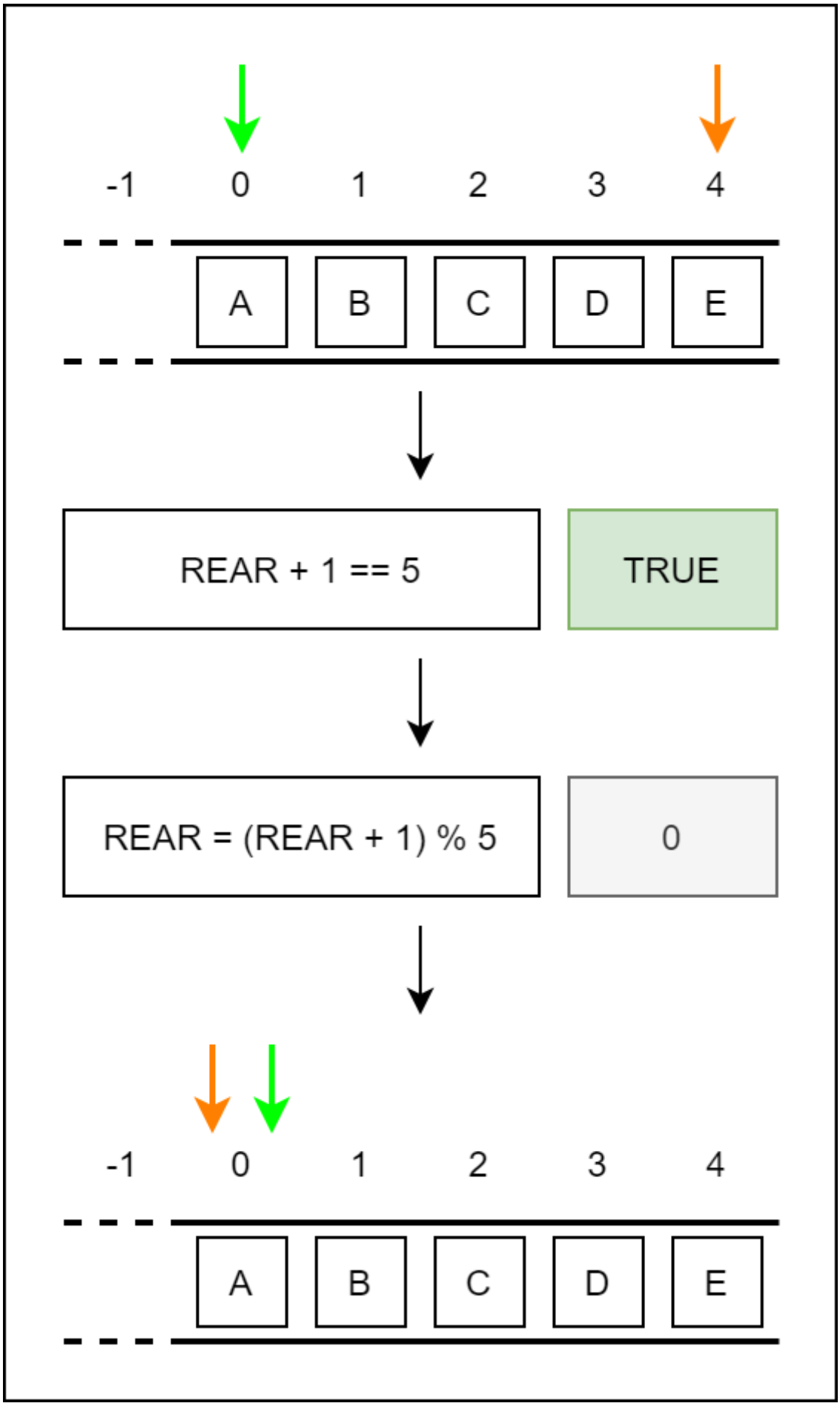


4. We insert yet another element at index 3, and **REAR** becomes 4. (Queue: [A, B, C, D, _])



5. Now, if we try to insert an element at index 4, the **REAR** pointer would normally become 5, but remember, this is a Circular Queue!

Instead of causing an overflow error, the **REAR** pointer wraps around and becomes 0. (Queue: [A, B, C, D, E])



So, the Circular Queue creates this circular motion, ensuring we can keep adding elements without wasting any space, making it much more efficient. And all of this is done using simple modulo division with the size of the queue.

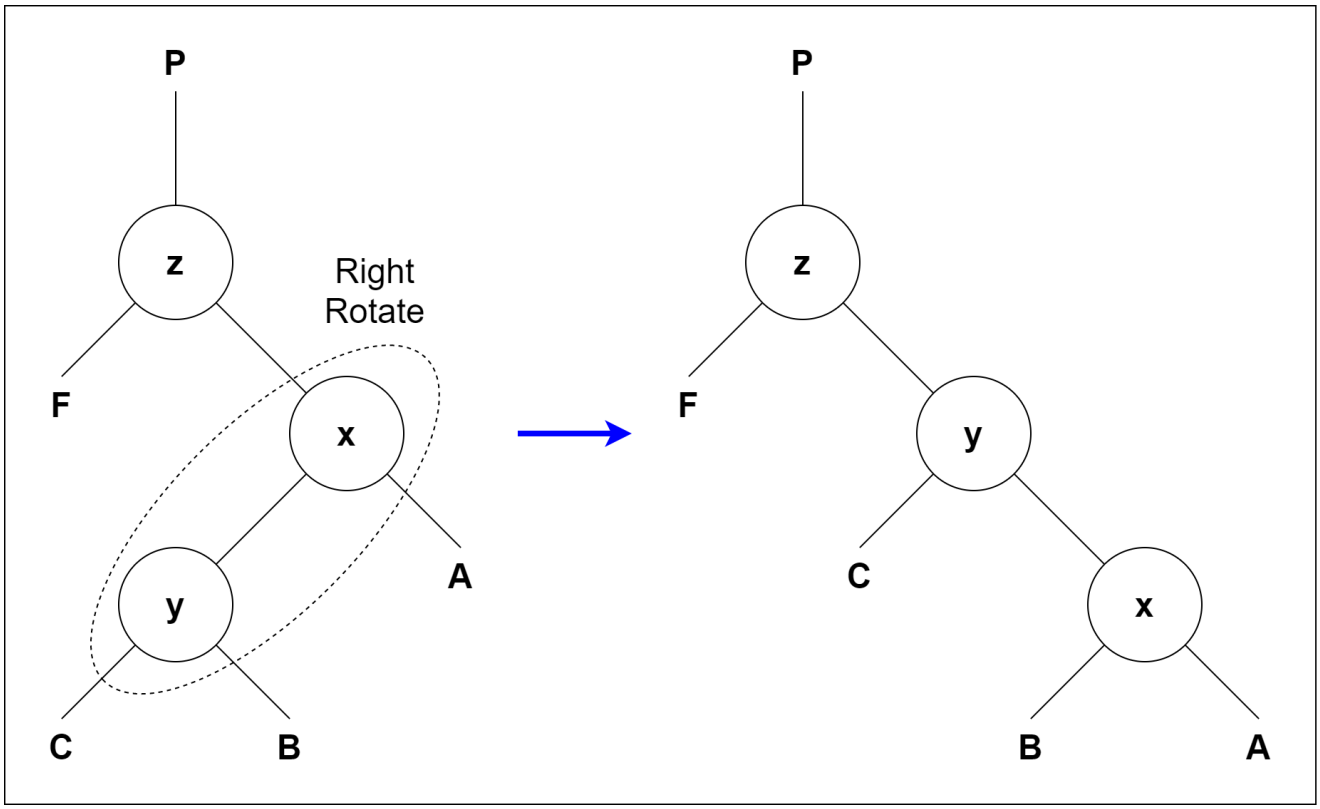
Isn't that a clever way to make the most out of our data structure? Circular Queues are pretty smart, and they come in handy when we need to optimize memory usage.

Circular Queue Operations

Alright, students, let's explore how the Circular Queue works. Think of it as a special type of queue with some clever features!

In a Circular Queue, we use two pointers to keep track of the elements:

1. The **FRONT** pointer points to the first element of the queue, which is the element we'll remove when we dequeue.
2. The **REAR** pointer points to the last element of the queue, which is where we'll insert new elements when we enqueue.



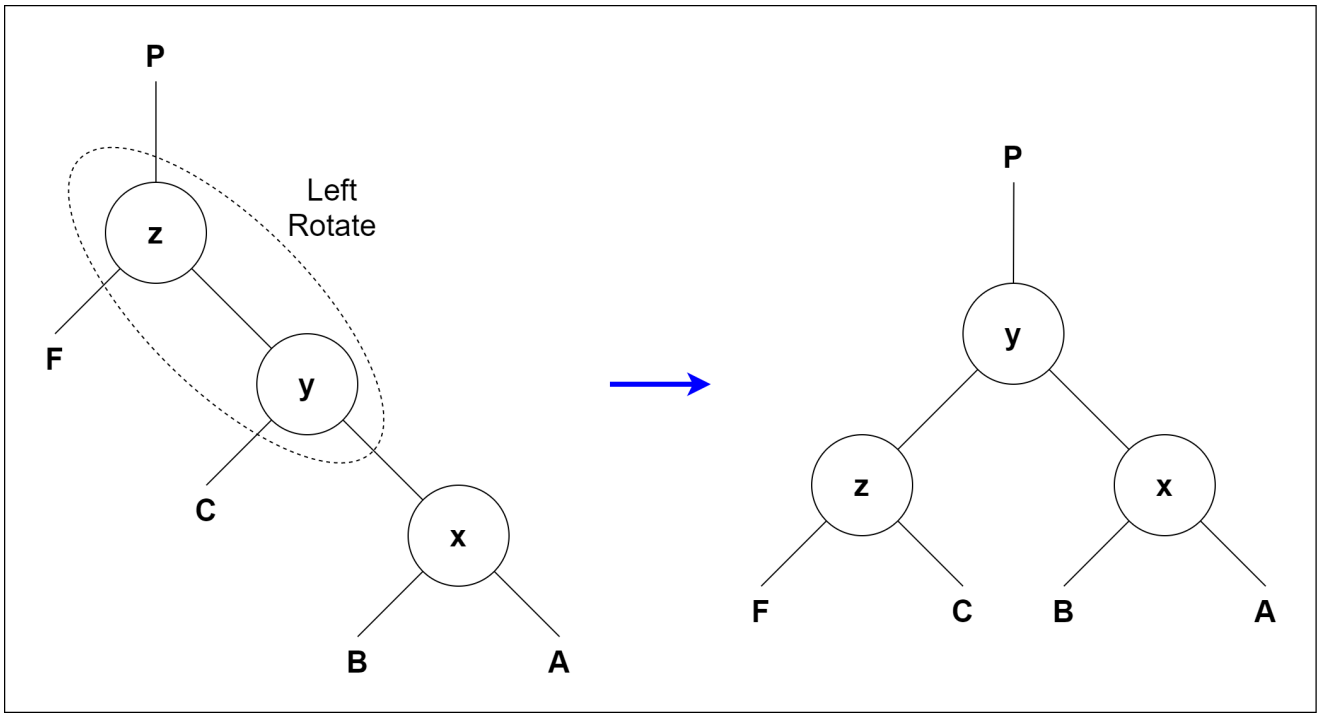
Now, here's the fascinating part: When the **REAR** pointer reaches the end of the queue, instead of causing an error, it wraps around and starts again from the beginning! It's like going in a circle, hence the name "Circular Queue."

To make sure our pointers are ready to go, we initially set both **FRONT** and **REAR** to -1. This signals that our Circular Queue is empty and has no elements.

With these pointers in place, we can efficiently insert and remove elements from our Circular Queue, making it a powerful tool for various applications.

1. Enqueue Operation

1. First, we need to check if the queue is full before adding a new element. We don't want to overflow it, right?
2. When we insert the first element into the Circular Queue, we set the value of **FRONT** to 0. It's like marking the starting point.
3. Next, when we add a new element, we do something really cool! We circularly increase the **REAR** index by 1. What does that mean? Well, imagine the **REAR** reaches the end of the queue. Instead of stopping there, it magically appears at the start of the queue! It's like a loop that keeps the queue going in a circle.
4. Now, we know where to add the new element because **REAR** points to the correct position. So, we simply place the new element in the position pointed to by **REAR**.



2. Dequeue Operation

1. First, we need to check if the queue is empty before removing an element. We don't want to remove something that's not there, right?
2. When we remove an element, we return the value pointed to by **FRONT**. That's the element we want to take out of the queue.
3. Now comes the exciting part! We circularly increase the **FRONT** index by 1. Just like we did with **REAR**, when the **FRONT** reaches the end of the queue, it magically appears at the start. Remember, our queue is like a circle!
4. But, there's a special case we need to watch out for. When **FRONT** reaches 0 and **REAR** is equal to **SIZE - 1**, or when **FRONT** is just one less than **REAR**, that means our queue is full! So we need to handle this case carefully.
5. Lastly, if we remove the last element from the queue, we reset the values of **FRONT** and **REAR** to -1. This indicates that the queue is now empty and ready to be used

again.

And there you have it, students! That's how the Dequeue operation works in a Circular Queue. It's pretty cool, isn't it? The circular incrementing makes it so efficient and allows us to make the most out of our queue's memory space.

Circular Queue Implementations in C++

Let's talk about the different ways we can implement a queue in programming. The most common and widely used method is by using arrays. Arrays provide a simple and efficient way to manage our queue.

However, that's not the only option we have! We can also implement a queue using lists. Lists offer some unique advantages, and they can be a great choice depending on our specific needs.

So, whether we go with arrays or lists, the basic functionality of the queue remains the same. We can perform operations like Enqueue and Dequeue, and check if the queue is empty or full.

By understanding these different implementations, you'll become more versatile programmers, capable of solving a wide range of problems. So keep exploring and practicing, and you'll master queues in no time!

```
// Circular Queue implementation in C++

#include <iostream>
#define SIZE 5 /* Size of Circular Queue */

using namespace std;

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

   public:
  Queue() {
    front = -1;
    rear = -1;
  }
  // Check if the queue is full
  bool isFull() {
    if (front == 0 && rear == SIZE - 1) {
      return true;
    }
```

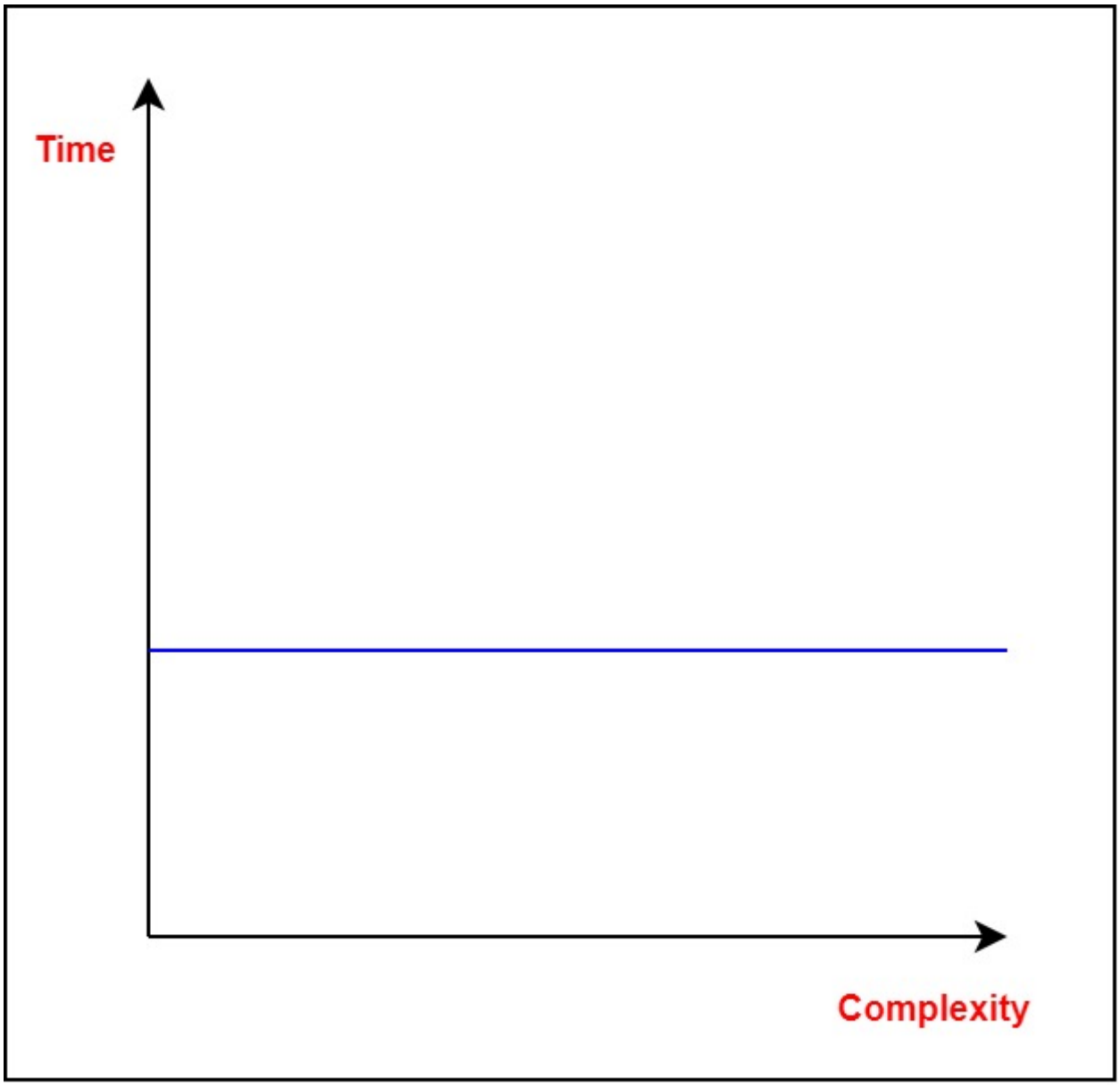
```
  if (front == rear + 1) {
    return true;
  }
  return false;
}
// Check if the queue is empty
bool isEmpty() {
  if (front == -1)
    return true;
  else
    return false;
}
// Adding an element
void enQueue(int element) {
  if (isFull()) {
    cout << "Queue is full";
  } else {
    if (front == -1) front = 0;
    rear = (rear + 1) % SIZE;
    items[rear] = element;
    cout << endl
       << "Inserted " << element << endl;
  }
}
// Removing an element
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 = (front + 1) % SIZE;
    }
    return (element);
  }
}

void display() {
  // Function to display status of Circular Queue
  int i;
  if (isEmpty()) {
```

```
      cout << endl
         << "Empty Queue" << endl;
    } else {
      cout << "Front -> " << front;
      cout << endl
         << "Items -> ";
      for (i = front; i != rear; i = (i + 1) % SIZE)
        cout << items[i];
      cout << items[i];
      cout << endl
         << "Rear -> " << rear;
    }
  }
};

int main() {
  Queue q;

  // Fails because front = -1
  q.deQueue();

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

  // Fails to enqueue because front == 0 && rear == SIZE - 1
  q.enQueue(6);

  q.display();

  int elem = q.deQueue();

  if (elem != -1)
    cout << endl
       << "Deleted Element is " << elem;

  q.display();

  q.enQueue(7);

  q.display();

  // Fails to enqueue because front == rear + 1
  q.enQueue(8);

  return 0;
}
```

Circular Queue Complexity Analysis

Now, let's dive into the complexity of enqueue and dequeue operations for a circular queue when we implement it using arrays.

You'll be happy to know that the complexity of both enqueue and dequeue operations is quite efficient! We can perform these operations in constant time, represented by O(1). This means that the time taken to enqueue or dequeue an element does not depend on the number of elements in the queue. It's always the same, making it a fast and efficient way to manage our data.



Applications of Circular Queue

Now, let's explore some real-world applications of the circular queue. You'll be amazed to see how versatile this data structure is!

One significant application of the circular queue is in **CPU scheduling**. In computer systems, the CPU has to manage multiple processes that need to be executed. The circular queue helps in efficiently scheduling these processes, ensuring that each one gets its fair share of CPU time.

Another important area where the circular queue shines is **memory management**. In computer systems, memory is a precious resource, and the circular queue plays a crucial role in managing it. It helps in organizing and allocating memory to different processes, making the best use of available space.

Lastly, the circular queue is also employed in **traffic management** systems. In scenarios like traffic signals, where vehicles need to wait in a queue before crossing an intersection, the circular queue ensures a smooth and fair flow of vehicles, preventing congestion and ensuring safety.

So, you can see how the circular queue is an essential tool in various domains, making our systems more efficient and organized. Keep exploring and applying your knowledge, and

you'll see the incredible impact of data structures like the circular queue in the world of computing!