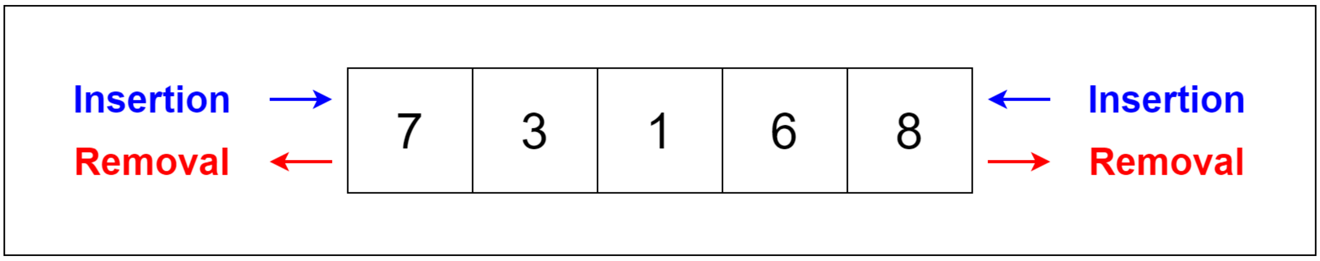
Double Ended Queue

Alright, now let's delve into the world of the Double Ended Queue, also known as the Deque.

A Deque, or Double Ended Queue, is a special type of queue that allows us to insert and remove elements from both the front and the rear. Unlike a regular queue that strictly follows the First In First Out (FIFO) rule, the Deque provides us with more flexibility in managing our elements.

Imagine a line of people waiting to enter a cinema hall. In a regular queue, the first person to arrive is the first one to enter the hall. But in a Deque, people can enter or leave the line from both ends, giving us the freedom to handle the queue in different ways.



Types of Deque

Alright, students, let's now explore the different types of Deques.

First, we have the "Input Restricted Deque." In this type of Deque, we are limited to inserting elements at only one end. However, we can still remove elements from both ends of the Deque. It's like having a line of people waiting to enter a bus, and they can only enter from the front, but they can exit from both the front and the back.

Next, we have the "Output Restricted Deque." This type of Deque is the opposite. Here, we are limited to removing elements from only one end, but we can insert elements at both ends of the Deque. It's like having a line of people again, but this time they can enter from both the front and the back, but they can only leave from the front.

So, you can see that Deques offer us different ways to manage our elements, depending on whether we want more flexibility in inserting or removing them. Understanding these different types of Deques will help you choose the right data structure for specific scenarios.

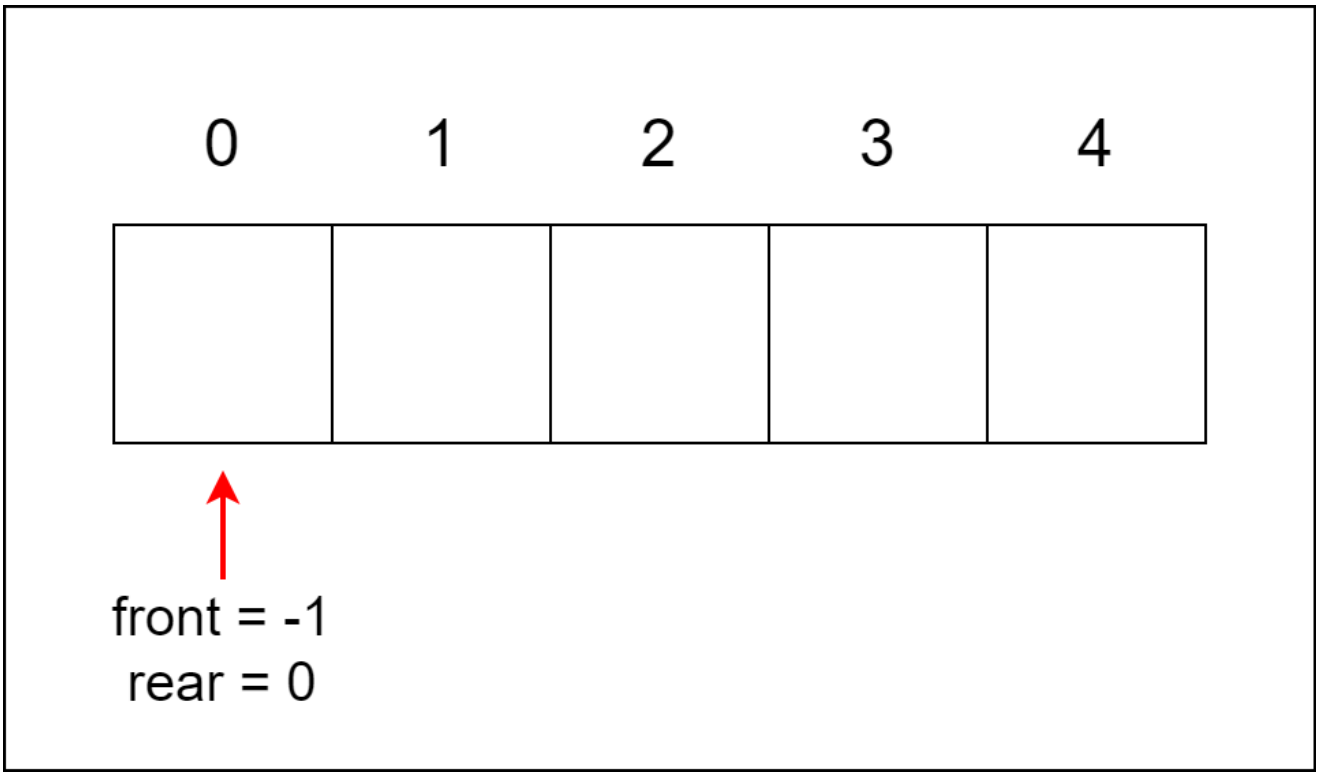
Operations on a Deque

Alright, everyone, now let's explore the circular array implementation of a Deque.

In this implementation, we use an array to create our Deque, and if the array is full, we start inserting elements from the beginning again. This circular approach allows us to make the most of our available space.

Before we start performing the various operations, we need to take the following steps:

1. First, we create an array (Deque) of a given size, let's call it "n."
2. Then, we set two pointers. One is called "front," and the other is called "rear." We initialize "front" to -1 and "rear" to 0.

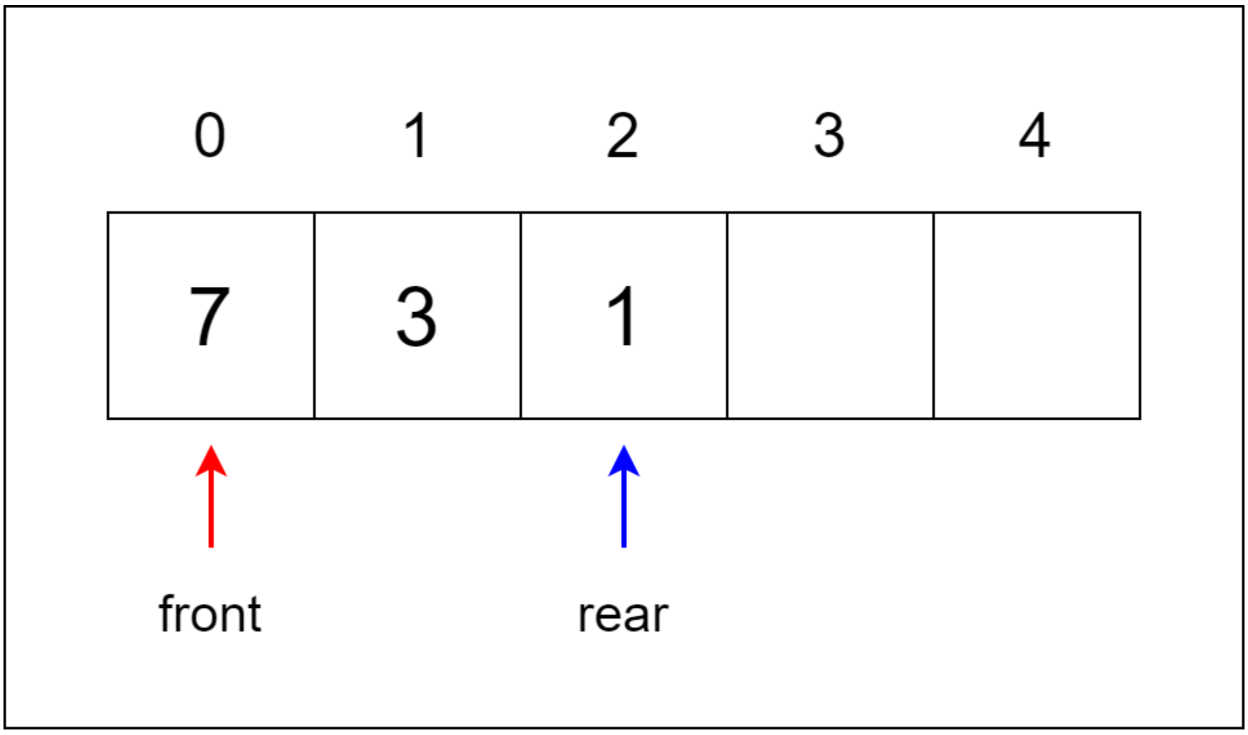


1. Insert at the Front

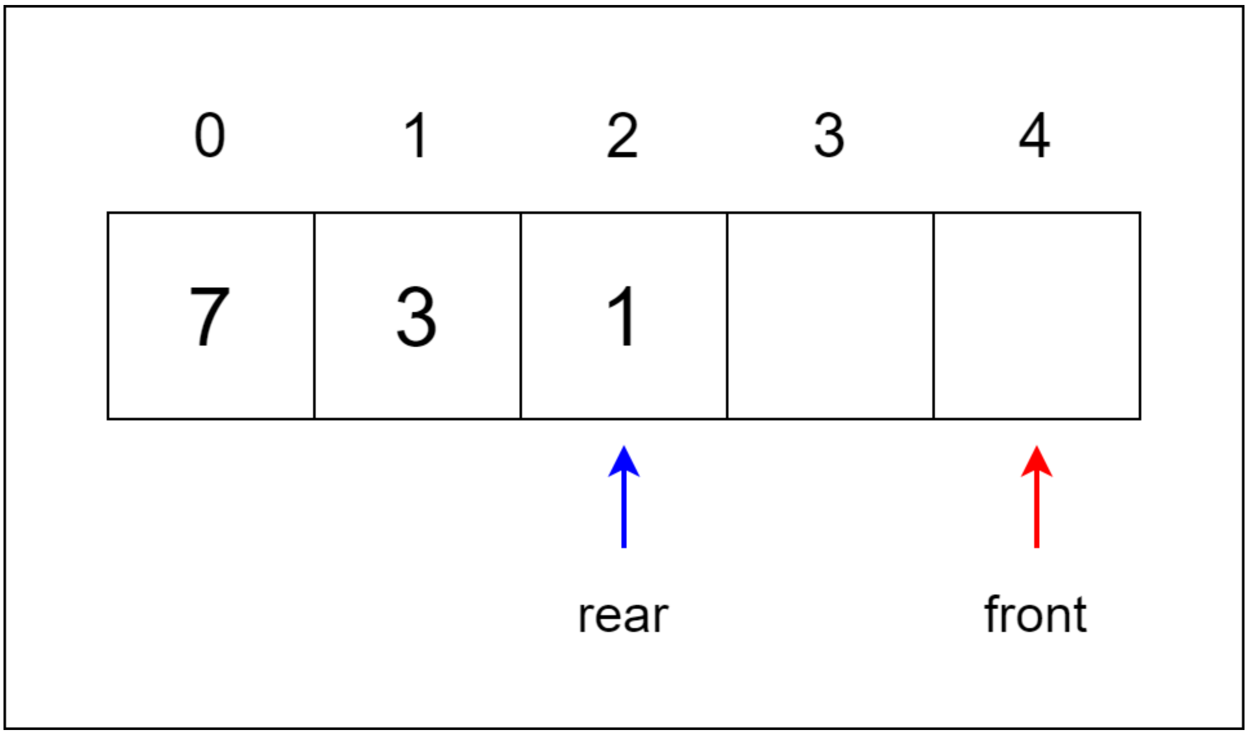
Alright, class, let's now learn about the "Insert at the Front" operation in our circular array-based Deque.

When we insert an element at the front of our Deque, we follow these steps:

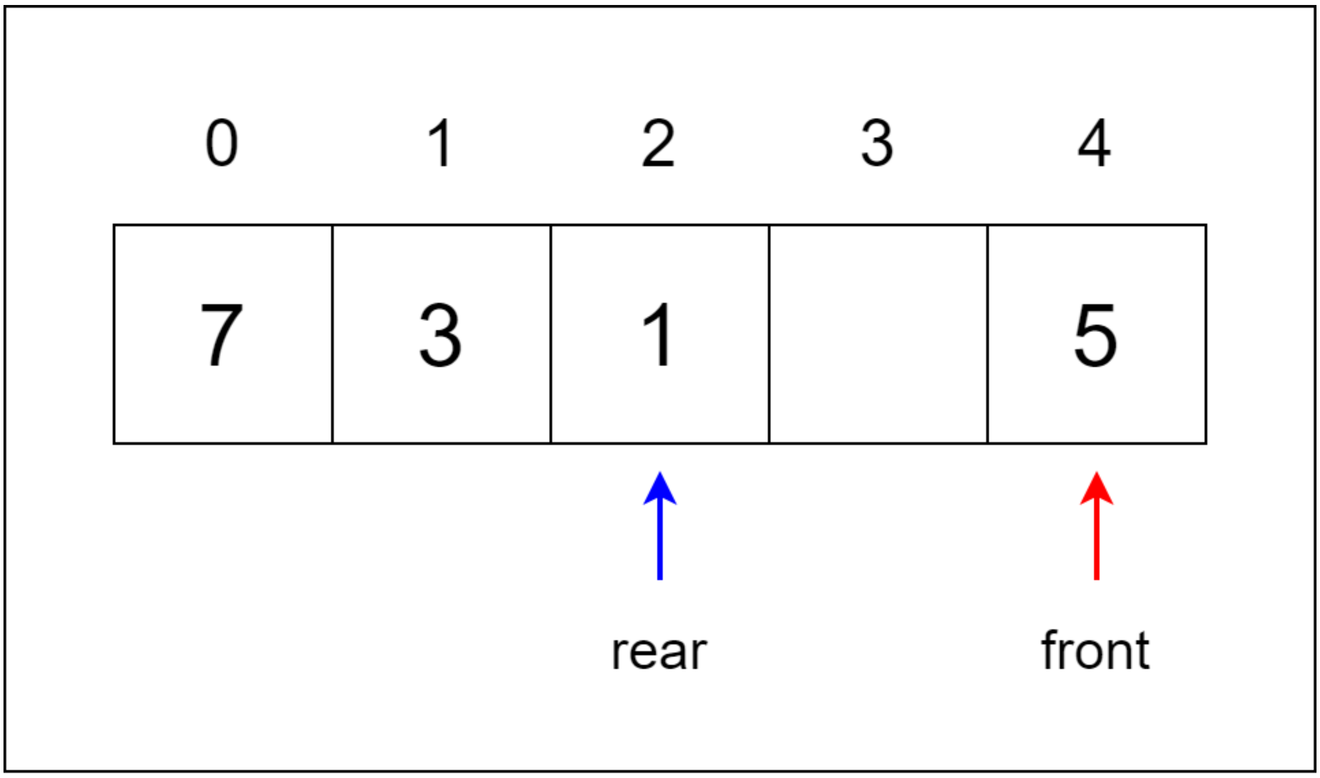
1. First, we check the position of the "front" pointer.



2. If the value of "front" is less than 1, it means we have reached the beginning of the array. In this case, we reinitialize "front" to the last index of the array, which is "n-1."



3. If the value of "front" is not less than 1, we simply decrease the "front" pointer by 1, moving it one step towards the front of the array.
4. Finally, we add the new element (let's say it's "5") into the position pointed by the "front" pointer in the array.

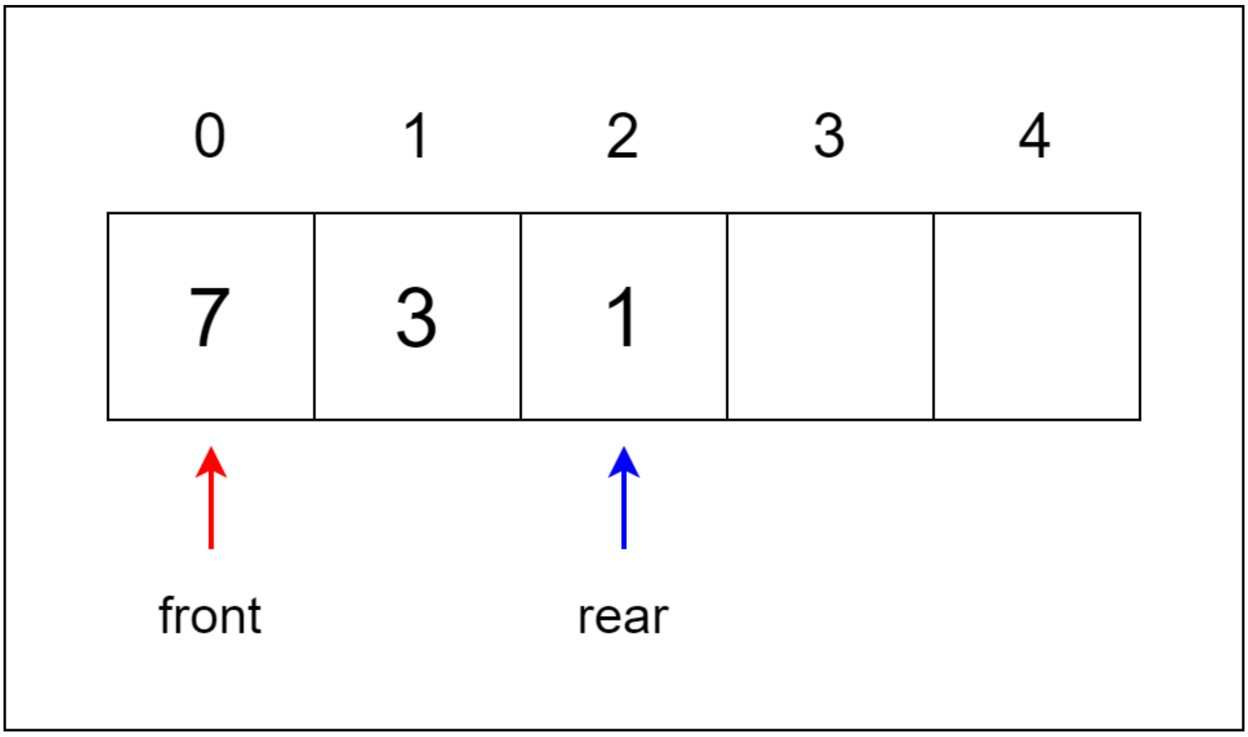


2. Insert at the Rear

Great! Let's now explore the "Insert at the Rear" operation for our circular array-based Deque.

When we insert an element at the rear of our Deque, we follow these steps:

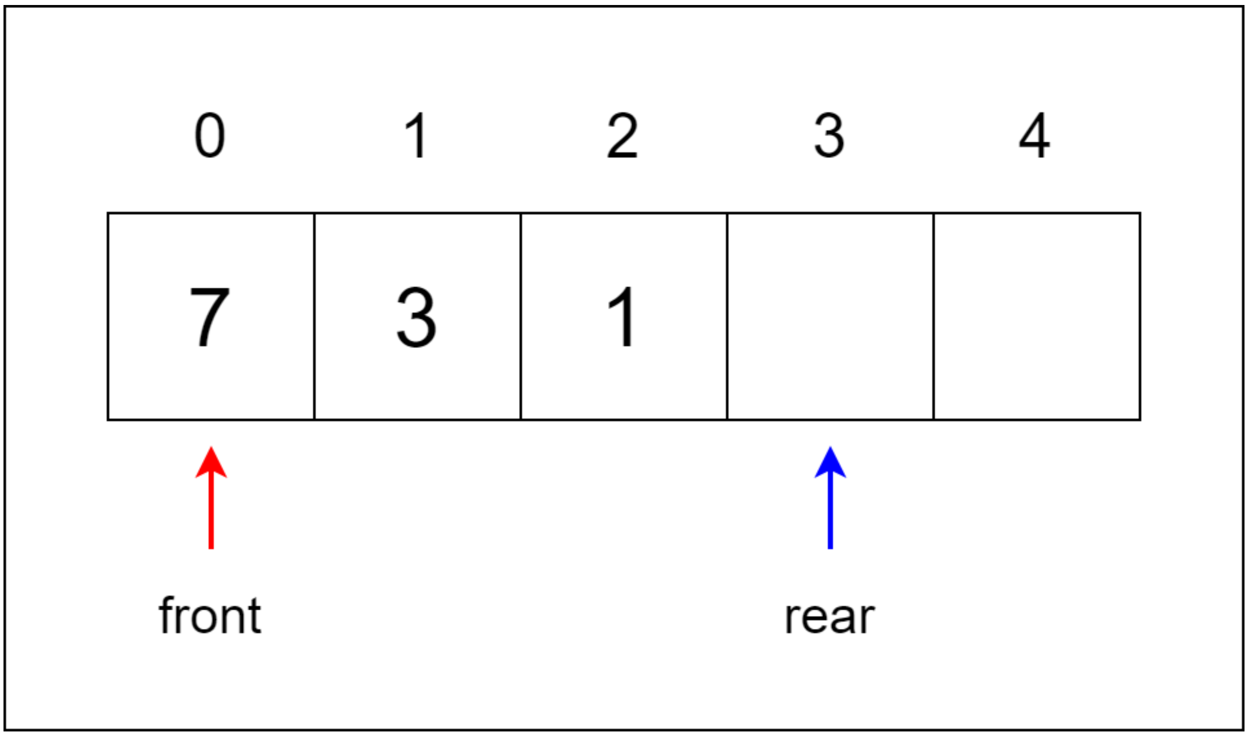
1. First, we check if the array is full, which means there is no more space to add new elements.



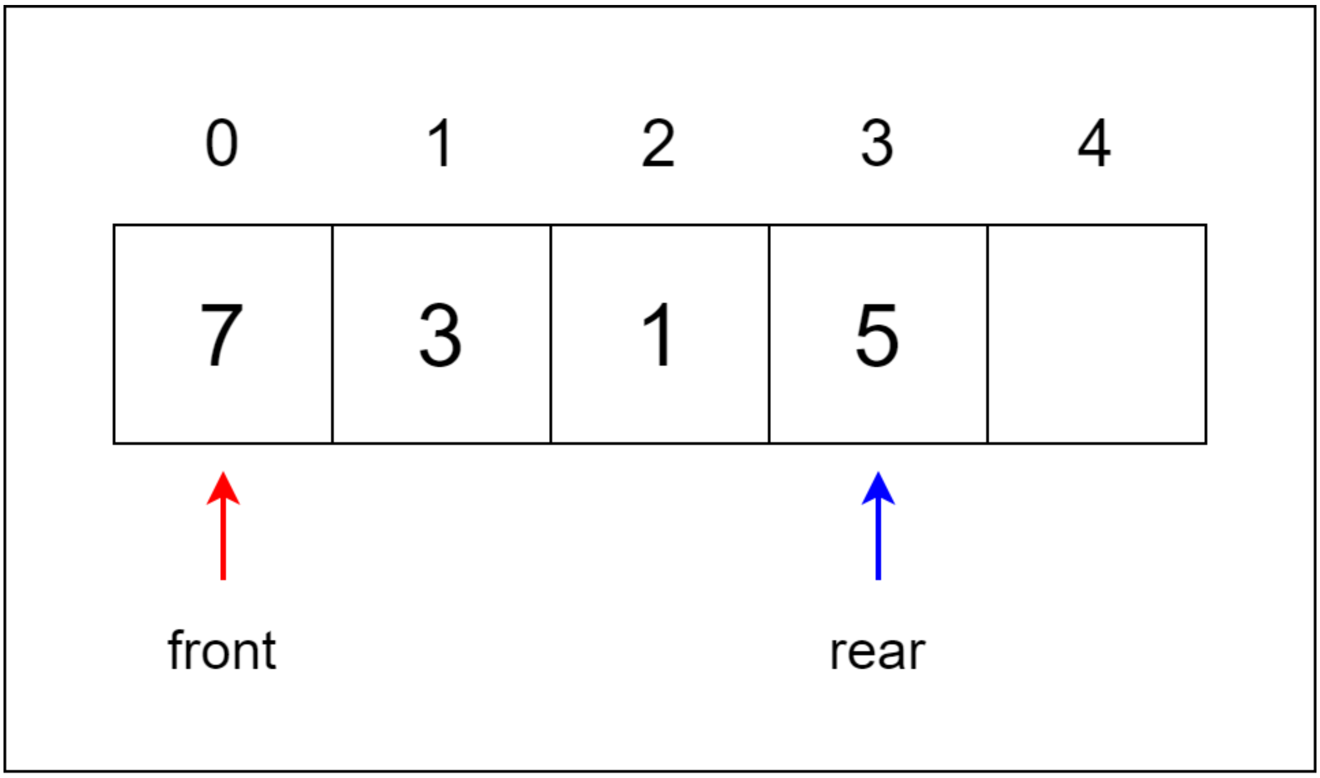
2. If the Deque is full, we reinitialize the "rear" pointer to 0, starting from the beginning of the array. This is because we are implementing a circular array, so if the rear has

reached the end of the array, we need to wrap it back to the beginning to make use of any available space.

3. If the Deque is not full and there is space for more elements, we simply increase the "rear" pointer by 1, moving it one step towards the rear of the array.



4. Finally, we add the new element (let's say it's "5") into the position pointed by the "rear" pointer in the array.

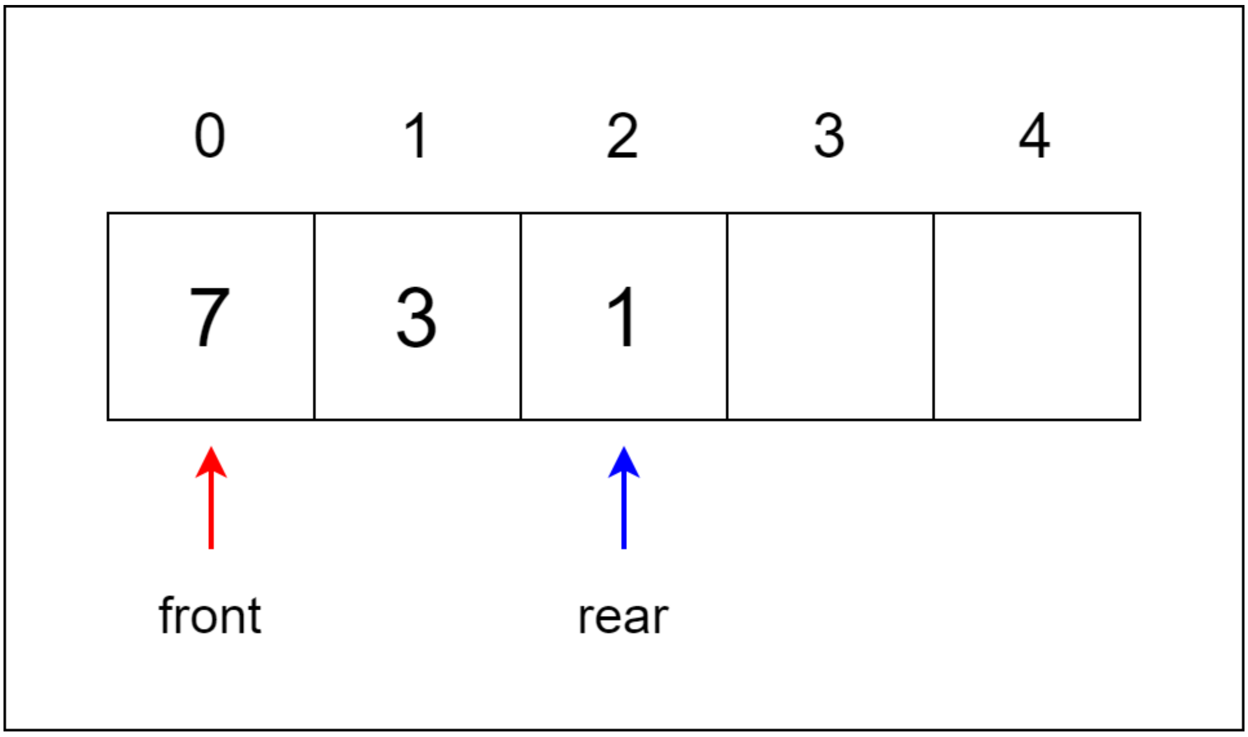


3. Delete from the Front

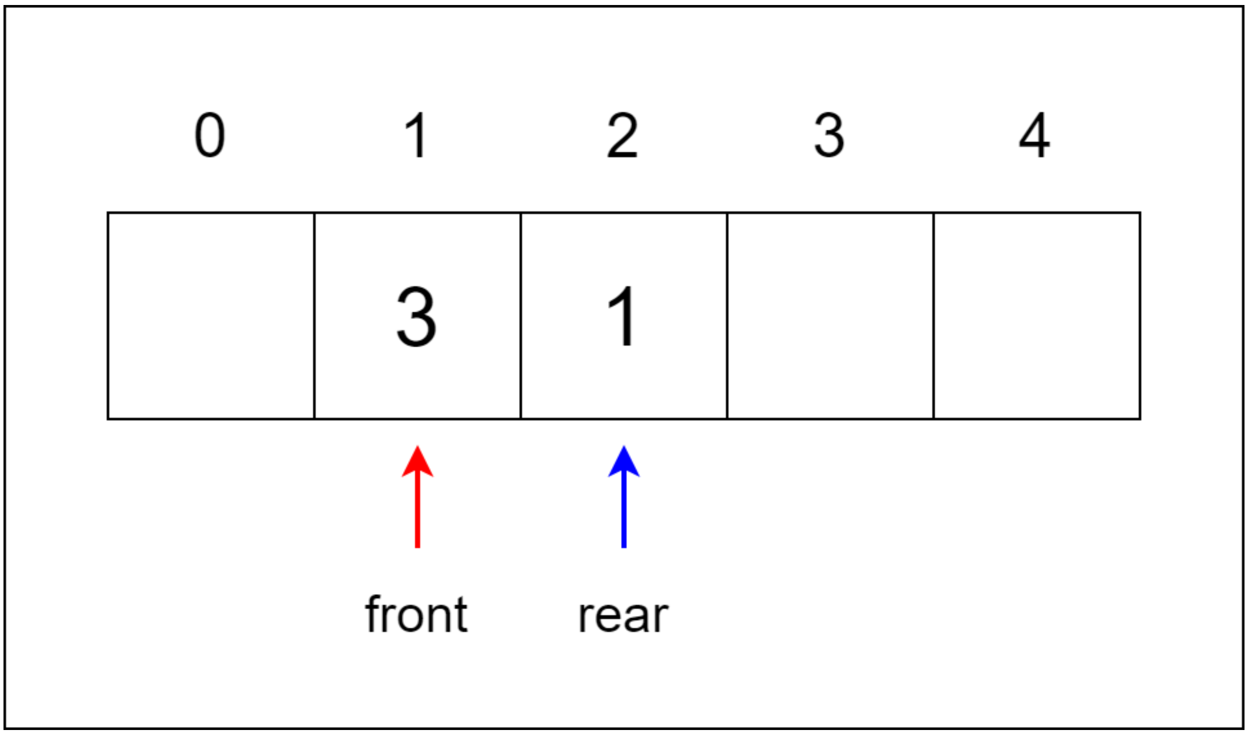
Let's explore the "Delete from the Front" operation for our circular array-based Deque.

When we want to delete an element from the front of our Deque, we need to follow these steps:

1. First, we check if the Deque is empty, which means there are no elements in the Deque to delete.



2. If the Deque is empty (i.e., the "front" pointer is -1), we cannot perform the deletion operation, and this condition is known as the **underflow condition**.
3. If the Deque has only one element (i.e., "front" and "rear" both point to the same element), we remove that element from the Deque. After deletion, we set both "front" and "rear" to -1, indicating that the Deque is now empty.
4. If "front" is pointing to the last element in the array (i.e., "front = n - 1" where "n" is the size of the array), we cannot delete an element from the front, as it will wrap around to the beginning of the circular array. To handle this situation, we set "front" to 0, going back to the front of the array.
5. For all other cases, we simply move the "front" pointer one step ahead by increasing its value by 1. This action effectively deletes the element that was previously at the front of the Deque.

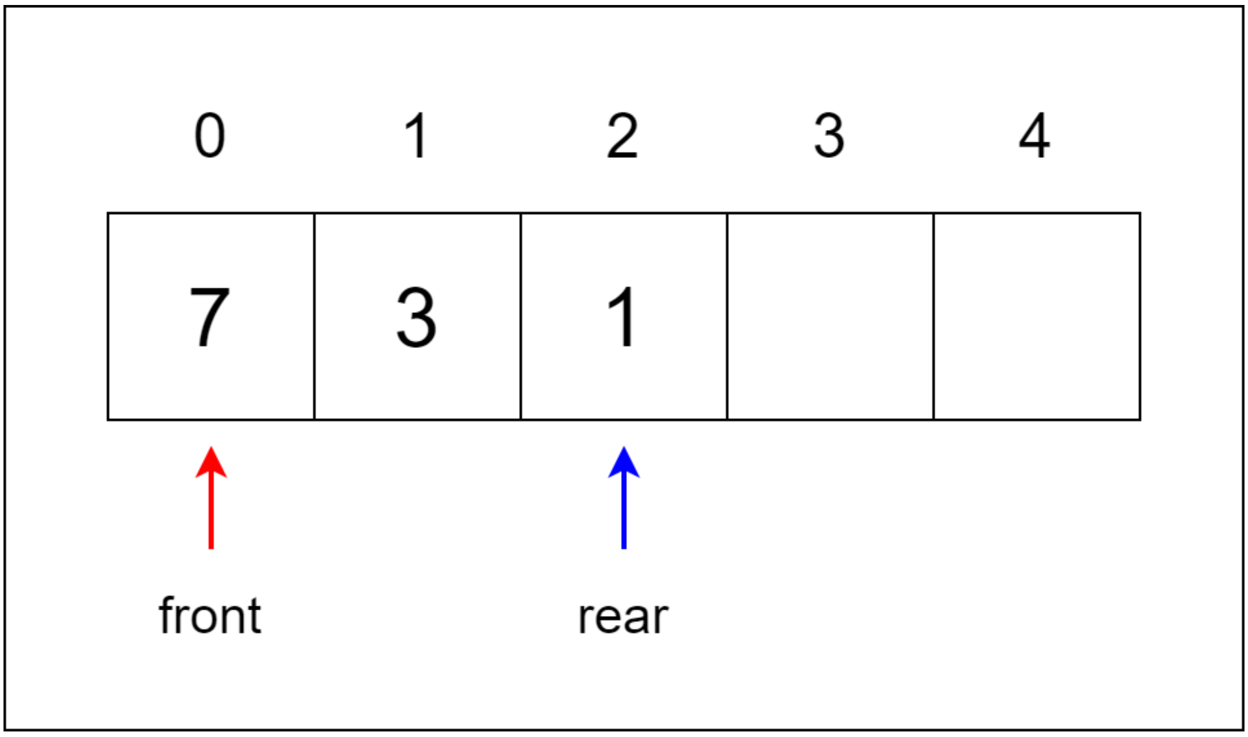


4. Delete from the Rear

Let's delve into the "Delete from the Rear" operation for our circular array-based Deque.

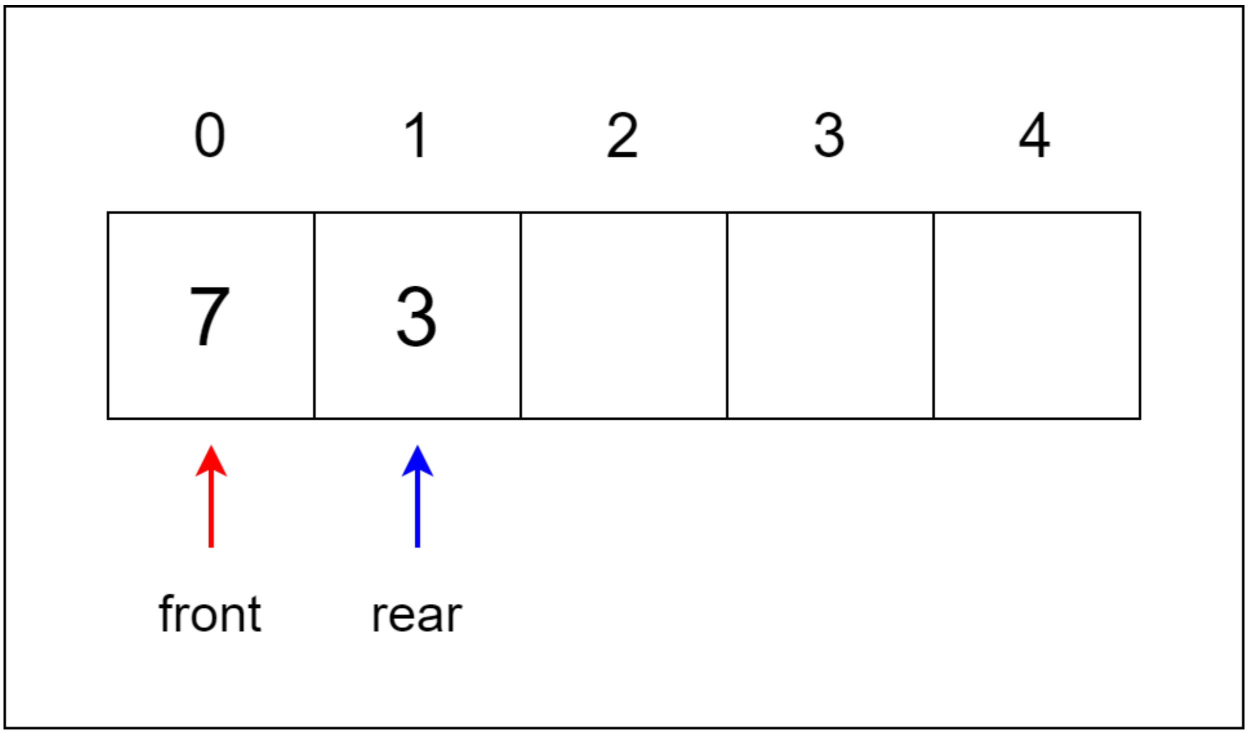
When we want to delete an element from the rear of our Deque, we need to follow these steps:

1. First, we check if the Deque is empty, which means there are no elements in the Deque to delete.



2. If the Deque is empty (i.e., the "front" pointer is -1), we cannot perform the deletion operation, and this condition is known as the **underflow condition**.

3. If the Deque has only one element (i.e., "front" and "rear" both point to the same element), we remove that element from the Deque. After deletion, we set both "front" and "rear" to -1, indicating that the Deque is now empty.
4. If "rear" is pointing to the front of the array (i.e., "rear = 0"), we cannot delete an element from the rear, as it will wrap around to the end of the circular array. To handle this situation, we set "rear" to "n - 1", going to the end of the array.
5. For all other cases, we simply move the "rear" pointer one step back by decreasing its value by 1. This action effectively deletes the element that was previously at the rear of the Deque.



5. Check Empty

Let's discuss the "Check Empty" operation for our circular array-based Deque.

The purpose of this operation is to check whether the Deque is empty or not. We can achieve this by examining the value of the "front" pointer.

Here's how the operation works:

1. We check the value of the "front" pointer.
2. If the value of "front" is -1, it indicates that the Deque is empty because there are no elements in the Deque. In other words, when "front" is -1, there is no front element, and hence, there are no elements at all.

6. Check Full

Let's delve into the "Check Full" operation for our circular array-based Deque.

The purpose of this operation is to determine whether the Deque is full or not. To achieve this, we need to examine the values of both the "front" and "rear" pointers.

Here's how the operation works:

1. We check the value of the "front" pointer and see if it is equal to 0.
2. We also check the value of the "rear" pointer and see if it is equal to "n - 1" (where "n" is the size of our circular array) or if "front" is exactly one less than "rear".
3. If either of these conditions is met, it indicates that the Deque is full.

Now, let's understand the two cases in which the Deque can be full: a) The first case: If the "front" is at the beginning of the circular array (i.e., "front = 0") and "rear" is at the last position (i.e., "rear = n - 1"), it means our Deque is full. b) The second case: If "front" is exactly one less than "rear" (i.e., "front = rear + 1"), it also indicates that the Deque is full.

Deque Implementation in C++

```
// Deque implementation in C++

#include <iostream>
using namespace std;

#define MAX 10

class Deque {
  int arr[MAX];
  int front;
  int rear;
  int size;

   public:
  Deque(int size) {
    front = -1;
    rear = 0;
    this->size = size;
  }

  void insertfront(int key);
  void insertrear(int key);
  void deletefront();
  void deleterear();
  bool isFull();
  bool isEmpty();
  int getFront();
  int getRear();
```

```
};

bool Deque::isFull() {
  return ((front == 0 && rear == size - 1) ||
      front == rear + 1);
}

bool Deque::isEmpty() {
  return (front == -1);
}

void Deque::insertfront(int key) {
  if (isFull()) {
    cout << "Overflow\n"
       << endl;
    return;
  }

  if (front == -1) {
    front = 0;
    rear = 0;
  }

  else if (front == 0)
    front = size - 1;

  else
    front = front - 1;

  arr[front] = key;
}

void Deque ::insertrear(int key) {
  if (isFull()) {
    cout << " Overflow\n " << endl;
    return;
  }

  if (front == -1) {
    front = 0;
    rear = 0;
  }

  else if (rear == size - 1)
    rear = 0;

  else
    rear = rear + 1;
  arr[rear] = key;
}

void Deque ::deletefront() {
  if (isEmpty()) {
    cout << "Queue Underflow\n"
       << endl;
    return;
  }

  if (front == rear) {
    front = -1;
    rear = -1;
  } else if (front == size - 1)
    front = 0;

  else
    front = front + 1;
}

void Deque::deleterear() {
  if (isEmpty()) {
    cout << " Underflow\n"
       << endl;
    return;
  }

  if (front == rear) {
    front = -1;
    rear = -1;
  } else if (rear == 0)
    rear = size - 1;
  else
    rear = rear - 1;
}

int Deque::getFront() {
  if (isEmpty()) {
    cout << " Underflow\n"
       << endl;
    return -1;
  }
  return arr[front];
}

int Deque::getRear() {
  if (isEmpty() || rear < 0) {
    cout << " Underflow\n"
       << endl;
```

```
      return -1;
    }
    return arr[rear];
  }

int main() {
    Deque dq(4);

    cout << "insert element at rear end \n";
    dq.insertrear(5);
    dq.insertrear(11);

    cout << "rear element: "
       << dq.getRear() << endl;

    dq.deleterear();
    cout << "after deletion of the rear element, the new rear element: " <<
dq.getRear() << endl;

    cout << "insert element at front end \n";

    dq.insertfront(8);

    cout << "front element: " << dq.getFront() << endl;

    dq.deletefront();

    cout << "after deletion of front element new front element: " << dq.getFront()
<< endl;
}
```

Time Complexity

The great thing about these operations is that they all have a constant time complexity, denoted as "O(1)." What does this mean? It means that the time taken to perform any of these operations does not depend on the size of the Deque. Whether we have a small Deque or a large one, the time it takes to insert, delete, or check whether it's full or empty remains constant.

This is fantastic news because constant time complexity is very efficient and desirable in data structures. It allows us to handle our Deque operations swiftly, making our code run smoothly and quickly, regardless of how many elements are in the Deque.

Applications of Deque Data Structure

Let's now explore some of the practical applications of the Deque data structure:

1. **Undo Operations on Software**: Have you ever made a mistake while typing or editing a document and wished you could simply undo the changes? Well, Deque can be used to implement the "Undo" functionality in software applications. It allows you to keep track of previous states or actions, so if you make a mistake, you can easily go back to a previous version.
2. **Storing History in Browsers**: Ever wondered how browsers keep track of the web pages you've visited, so you can easily go back and forth between them? They use the Deque data structure to store the browsing history. When you click the back or forward button, the browser efficiently retrieves the previous or next URL from the Deque, allowing you to navigate through your browsing history seamlessly.
3. **Implementing Stacks and Queues**: A Deque is quite versatile. It can be used to implement both stacks and queues. By using specific operations, you can transform a Deque into either a Stack (Last In First Out) or a Queue (First In First Out), making it a valuable tool in solving various programming problems.