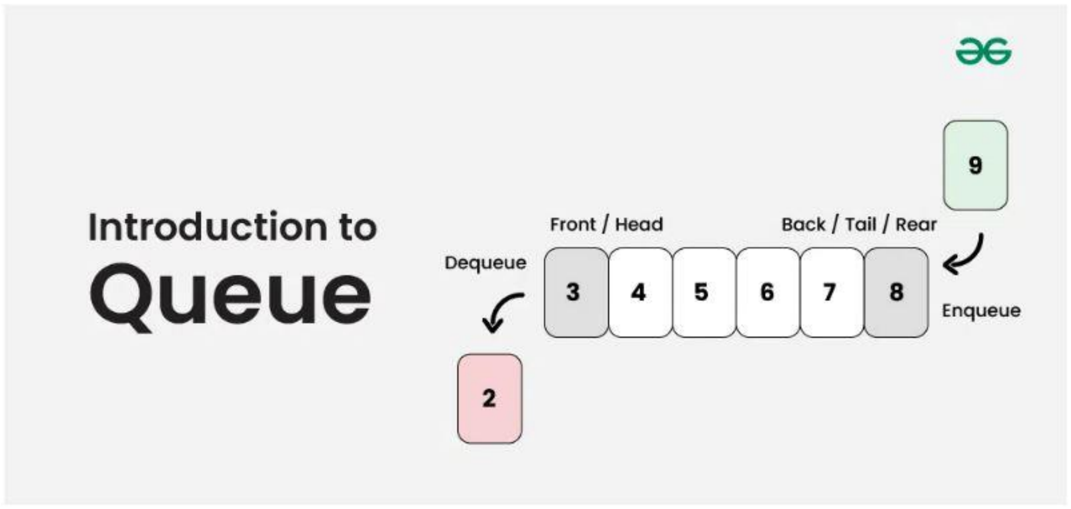
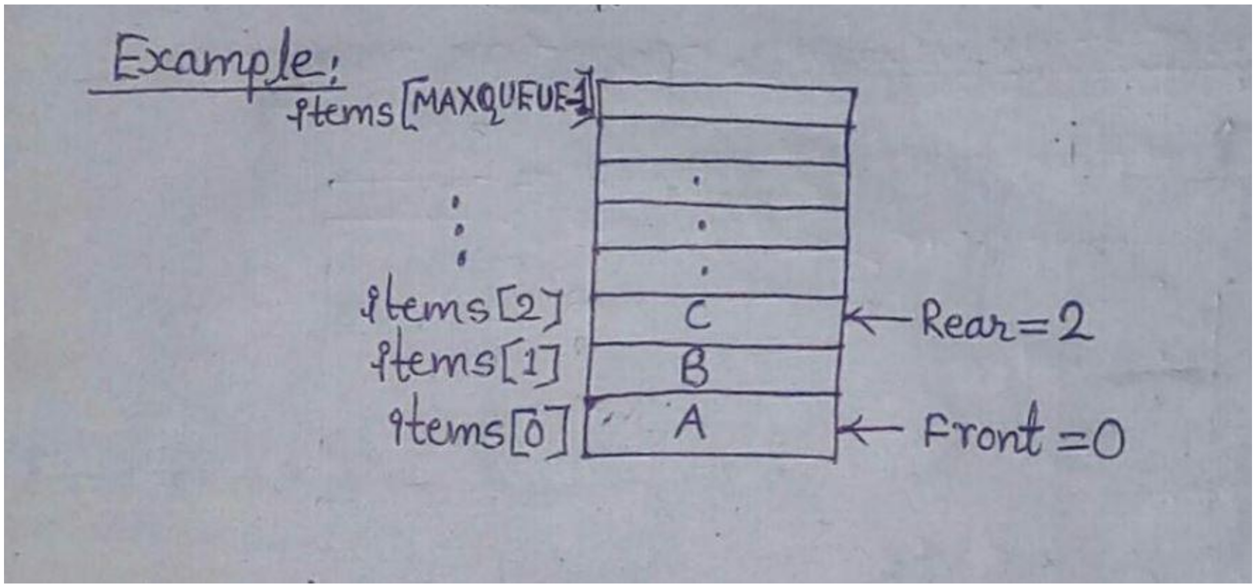
Chapter – 3

Queue and Linked List

Queue:

- **Queue Data Structure** is a linear data structure that follows **FIFO (First In First Out) Principle**, so the first element inserted is the first to be popped out.
- **A queue is an ADT** similar to stack, the thing that makes queue different from stack is the queue is open at both ends.
- The data is inserted into queue through one end **(the rear)** and deleted from it using the other end **(the front).**
- A Queue is like a line waiting to purchase tickets, where the first person in line is the first person served. (i.e. First Come First Serve).



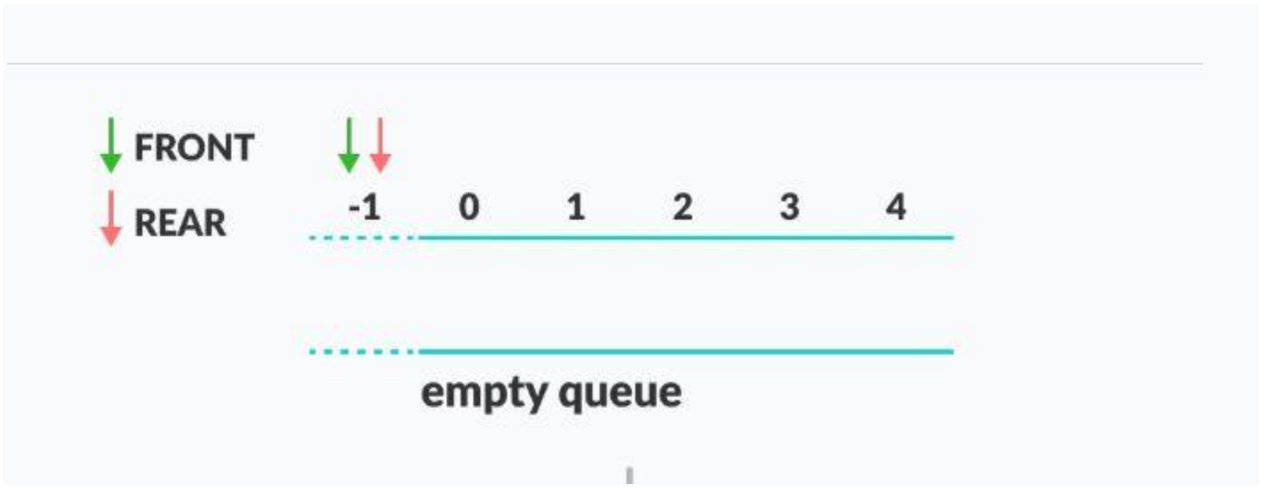


A **Queue** is an **Abstract Data Type (ADT)** that follows the **First-In-First-Out (FIFO)** principle, meaning that the first element added to the queue will be the first to be removed. It is similar to a queue in real life, such as a line at a grocery store, where the first person in line is the first one served.

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



Queue ADT Operations:

Some of the basic operations for Queue in Data Structure are:

- **enqueue() –** Insertion of elements to the queue.
- **dequeue() –** Removal of elements from the queue.
- **peek() or front()-** Acquires the data element available at the front node of the queue without deleting it.
- **rear() –** This operation returns the element at the rear end without removing it.
- **isFull() –** Validates if the queue is full.
- **isEmpty() –** Checks if the queue is empty.
- **size():** This operation returns the size of the queue i.e. the total number of elements it contains.

Array Implementation of queue:

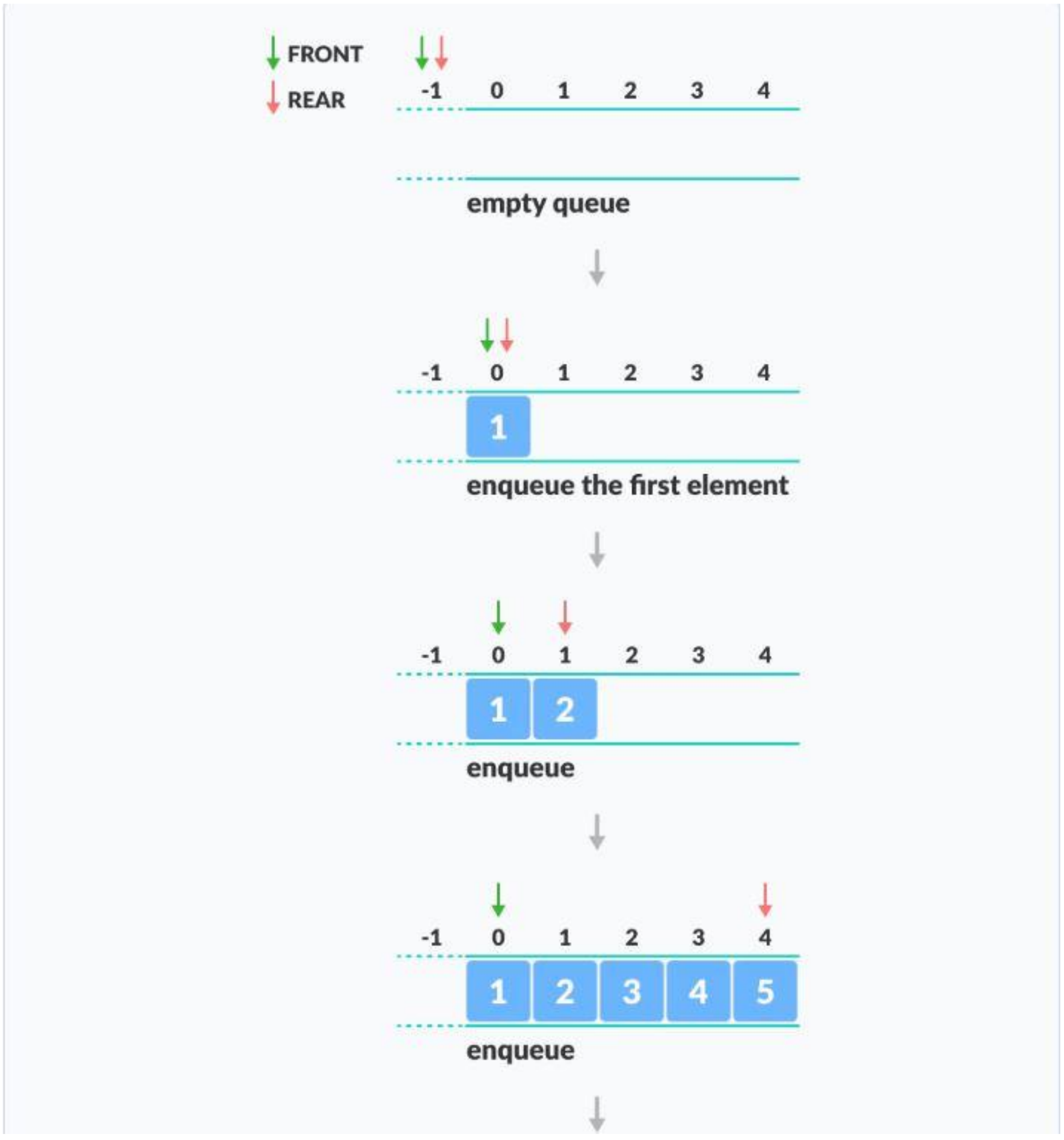
Operation 1: enqueue()

Inserts an element at the end of the queue i.e. at the rear end.

The following steps should be taken to enqueue (insert) data into a queue:

- Check if the queue is full.
- If the queue is full, return overflow error and exit.
- If the queue is not full, increment the rear pointer to point to the next empty space.
- Add the data element to the queue location, where the rear is pointing.
- return success.

```
void enqueue(int value) {

    if (rear == MAX - 1) {

        printf("Queue is full\n");

    } else if (front == -1 && rear == -1) {

        front = rear = 0;

        queue[rear] = value;

        printf("Enqueued: %d\n", value);

    } else {

        rear++;

        queue[rear] = value;

        printf("Enqueued: %d\n", value);

    }

}
```



Complexity Analysis:
Time Complexity: O(1)
Space Complexity: O(N)

Operation 2: dequeue()

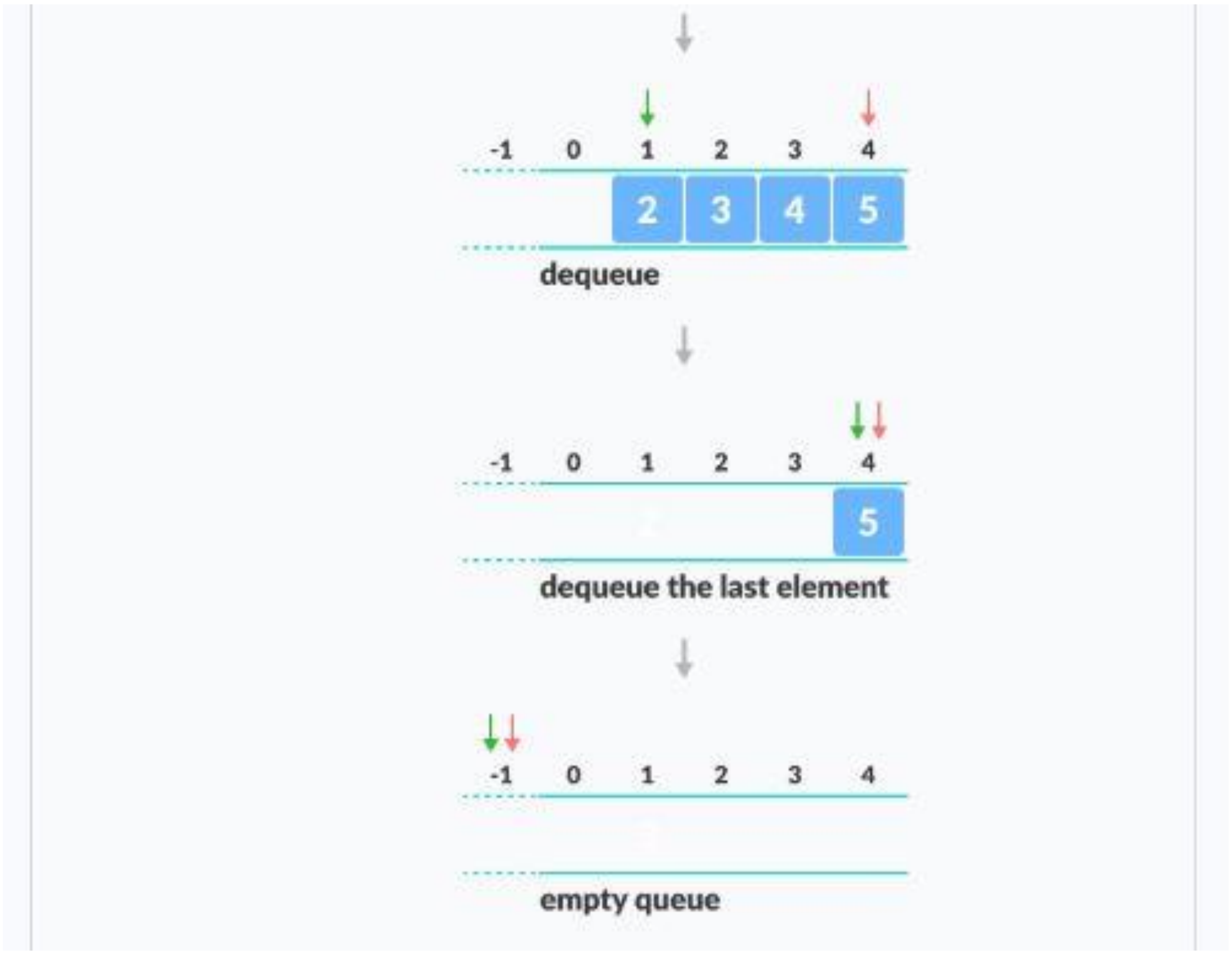
This operation removes and returns an element that is at the front end of the queue.

The following steps are taken to perform the dequeue operation:

- Check if the queue is empty.

- If the queue is empty, return the underflow error and exit.
- If the queue is not empty, access the data where the front is pointing.
- Increment the front pointer to point to the next available data element.
- The Return success.

```
int dequeue() {

    if (front == -1 && rear == -1) {

        printf("Queue is empty\n");

        return -1;  // Return a default value

    } else if (front == rear) {

        int value = queue[front];

        front = rear = -1;

        return value;

    } else {

        int value = queue[front];

        front++;

        return value;

    }

}
```



Complexity Analysis:
Time Complexity: O(1)
Space Complexity: O(N)

Types of Queue:

There are four different types of queues:

- Simple Queue
- Circular Queue
- Priority Queue

- Double Ended Queue

1. **Simple Queue (Linear Queue)**

Definition:

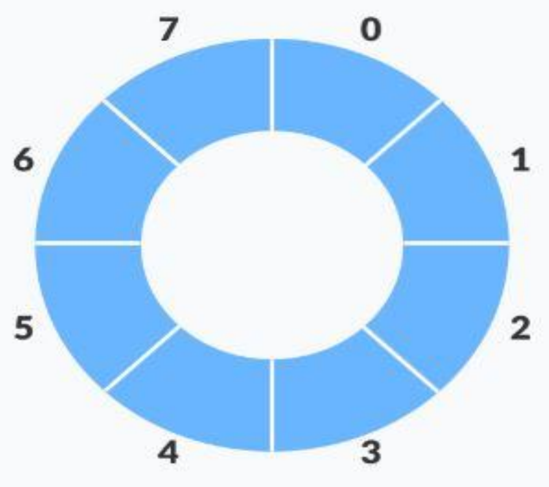
- A basic queue where elements are inserted at the rear and removed from the front.
- Operates on the **FIFO (First In, First Out)** principle.
- Once an element is deleted, we cannot insert another element in its position. This disadvantage of a linear queue is overcome by a circular queue, thus saving memory.

```
#include <stdio.h>

#define MAX 5



int queue[MAX];

int front = -1, rear = -1;



void enqueue(int value) {

    if (rear == MAX - 1) {

        printf("Queue is full\n");

    } else if (front == -1 && rear == -1) {

        front = rear = 0;

        queue[rear] = value;

        printf("Enqueued: %d\n", value);

    } else {
        rear++;

        queue[rear] = value;

        printf("Enqueued: %d\n", value);

    }

}


int dequeue() {

    if (front == -1 && rear == -1) {

        printf("Queue is empty\n");

        return -1;  // Return a default value

    } else if (front == rear) {

        int value = queue[front];

        front = rear = -1;

        return value;

    } else {

        int value = queue[front];

        front++;

        return value;

    }

}
```

```
int main() {

    enqueue(10);

    enqueue(20);

    enqueue(30);

    printf("Dequeued: %d\n", dequeue());

    return 0;

}
```

2. **Circular Queue:**

- In Circular Queue, all the nodes are represented as circular. It is similar to the linear Queue except that the last element of the queue is connected to the first element. It is also known as Ring Buffer, as all the ends are connected to another end.
-



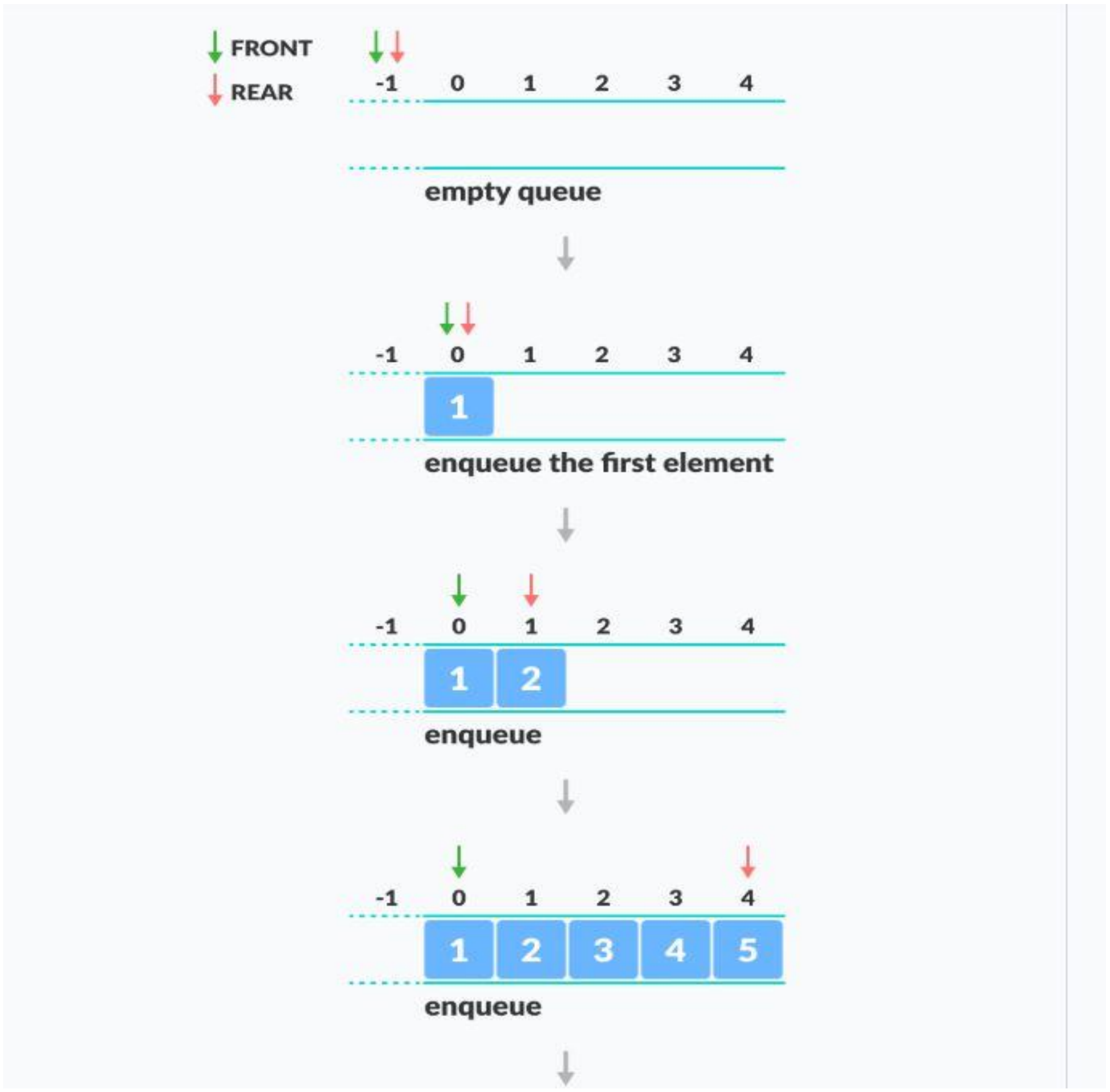
Circular queue representation

Implement Circular Queue using Array:

1. Initialize an array queue of size **n**, where n is the maximum number of elements that the queue can hold.
2. Initialize two variables front and rear to -1.

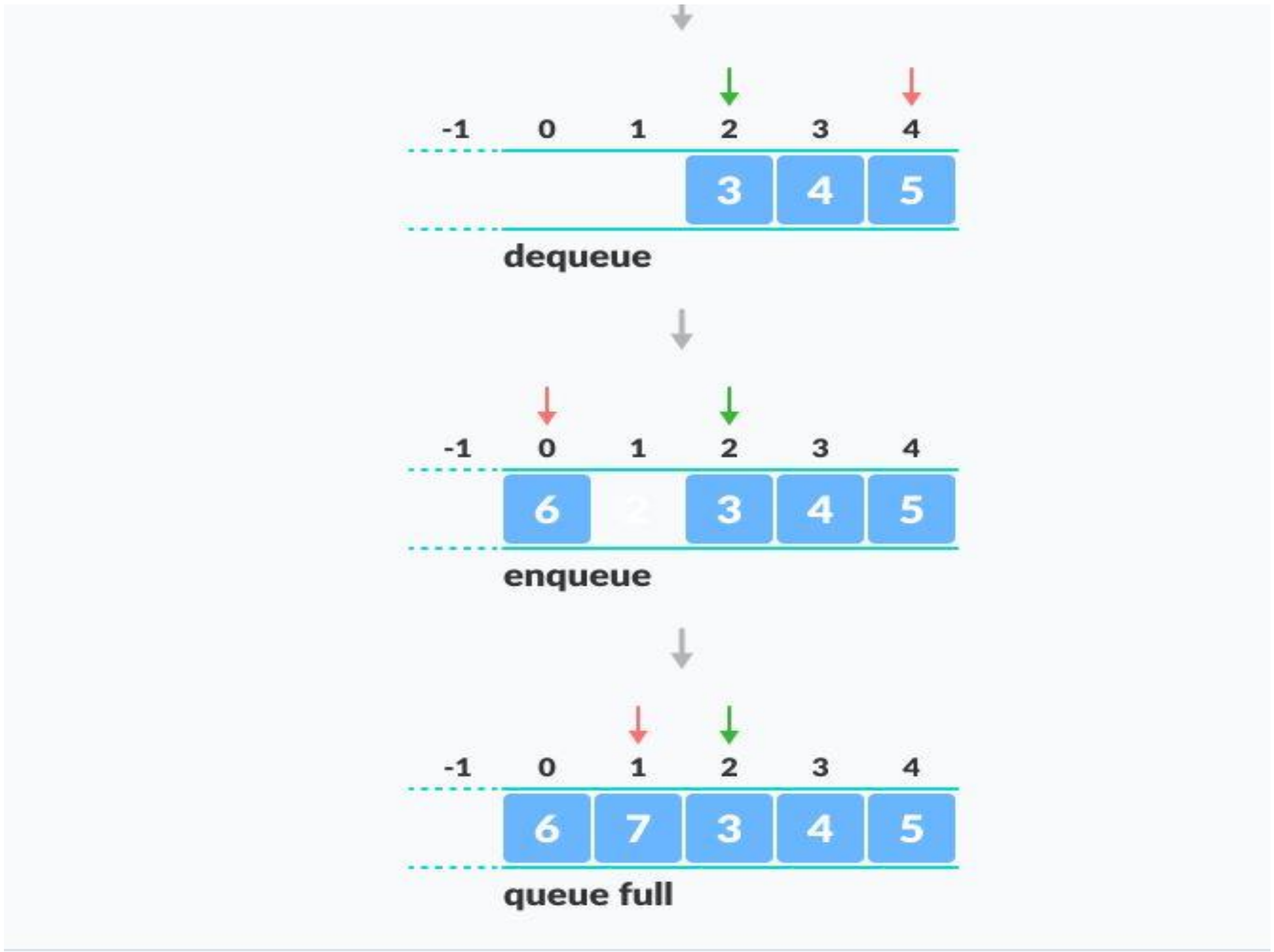
3. **Enqueue:** To enqueue an element **x** into the queue, do the following:
   - Increment rear by 1.
     - If **rear** is equal to n, set **rear** to 0.
   - If **front** is -1, set **front** to 0.
   - Set queue[rear] to x.

```
void enqueue(int element) {
    if ((rear + 1) % MAX_SIZE == front) {
        printf("Queue is full\n");
        return;
    }
    if (front == -1) {
        front = 0;  // Initialize front if queue is empty
    }
    rear = (rear + 1) % MAX_SIZE;  // Circular increment for rear
    queue[rear] = element;
    printf("Enqueued: %d\n", element);
}
```



4. **Deqeue:**
5. To dequeue an element from the queue, do the following:
   - Check if the queue is empty by checking if **front** is -1.
     - If it is, return an error message indicating that the queue is empty.
   - Set **x** to queue[front].
   - If **front** is equal to **rear**, set **front** and **rear** to -1.
   - Otherwise, increment **front** by 1 and if **front** is equal to n, set **front** to 0.
   - Return x.

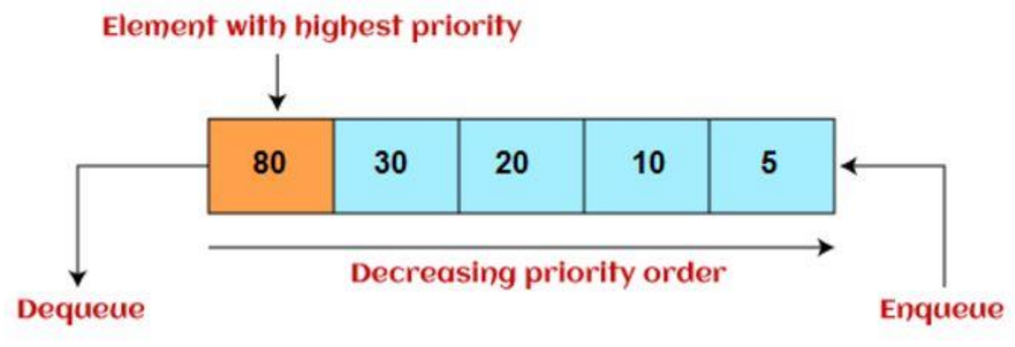
```
int dequeue() {
    if (front == -1 && rear == -1) {
        printf("Queue is empty\n");
        return -1;
    }
    int element = queue[front];  // Get the front element
    if (front == rear) {
        // If the queue has only one element, reset to empty
        front = -1;
        rear = -1;
    } else {
        // Move front to the next position in a circular manner
        front = (front + 1) % MAX_SIZE;
    }
    printf("Dequeued: %d\n", element);
    return element;
}
```



Enque and Deque Operations

Priority Queue

- A priority queue is a **special type of queue** in which each element is associated with a **priority value**. And, elements are served on the basis of their priority. That is, higher priority elements are served first.
- However, if elements with the same priority occur, they are served according to their order in the queue.

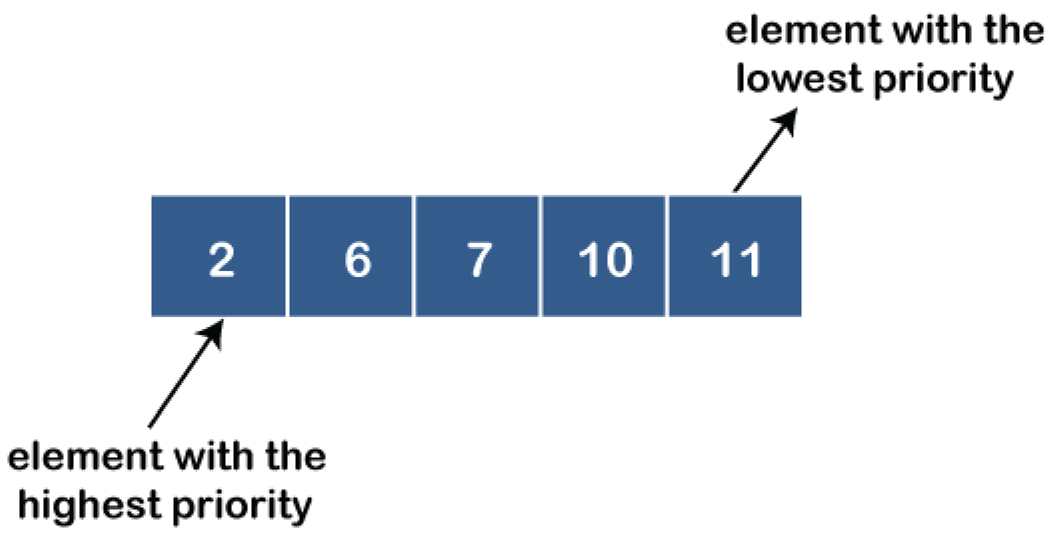


Insertion in priority queue takes place based on the arrival, while deletion in the priority queue occurs based on the priority. Priority queue is mainly used to implement the CPU scheduling algorithms.

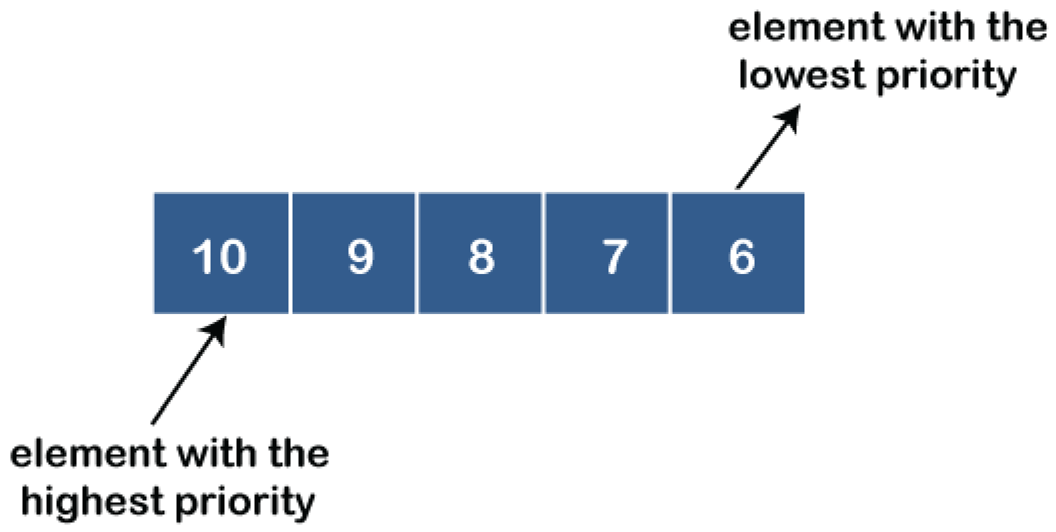
Types of Priority Queue

There are two types of priority queue:

- **Ascending order priority queue:** In ascending order priority queue, a lower priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e., 1 is given as the highest priority in a priority queue.


element with the lowest priority
2
6
7
10
11
element with the highest priority

- **Descending order priority queue:** In descending order priority queue, a higher priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number, i.e., 5 is given as the highest priority in a priority queue.


element with the lowest priority
10
9
8
7
6
element with the highest priority

1**. Enqueue (Insert) Algorithm:**

Algorithm for Enqueue in Priority Queue:

1. **Enqueue**: To enqueue an element x into the priority queue, do the following:
   - **Check if the queue is full** (if necessary).
     - If the queue is full, print "Queue is full" and exit.
   - **Insert the element in the queue** based on its priority.
     - Find the correct position for the element in the queue so that the queue maintains a **sorted order** based on priority.
     - Elements with higher priority (numerically lower values) will be at the front of the queue.
   - **Shift the elements** (if necessary) to make space for the new element.
   - **Insert the element at the correct position**.

1**. Dequeue Algorithm:**

Dequeue Algorithm for Priority Queue:

1. **Dequeue**: To dequeue an element from the priority queue, do the following:
   - **Check if the queue is empty**:
     - If the queue is empty, print "Queue is empty" and exit.
   - **Remove the element at the front of the queue** (the element with the highest priority).
   - **Shift all elements** in the queue to fill the gap left by the removed element.

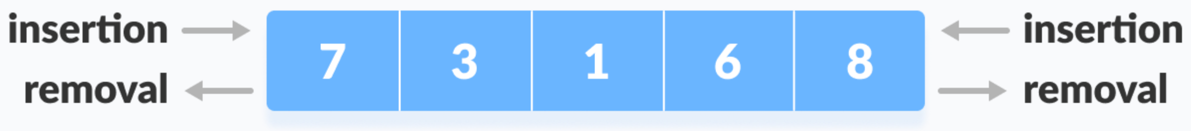
- **Decrement the size** of the queue.

Applications of Priority Queue:

- **Task Scheduling**: Prioritizing tasks based on their importance or urgency.
- **Dijkstra's Algorithm**: For finding the shortest path in a graph, where the priority queue is used to select the node with the smallest tentative distance.
- **Huffman Encoding**: For building an optimal prefix tree where the lowest frequency symbols are merged first.
- **Event Simulation**: Scheduling events in a simulator where events are handled in order of priority.

Deque Data Structure

Deque or Double Ended Queue is a type of queue in which insertion and removal of elements can either be performed from the front or the rear. Thus, it does not follow FIFO rule (First In First Out).



Representation of Dequeue

Characteristics: of Deque:

- **Dynamic size:** The size of a deque can change dynamically during the execution of a program.
- **Linear:** Elements in a deque are stored linearly and can be accessed in a sequential manner.
- **Double Ended:** Elements can be added and removed from both the front and the rear end

Application of deque:

- **Browser History:** A deque can be used to store the history of recently visited web pages.
- **Text Editor:** A deque can be used to undo and redo operations in a text editor.
- **Disk Scheduling:** A deque can be used to schedule disk requests to improve disk performance.

Types of Deque

Input Restricted Deque

An Input-restricted deque is a type of queue where we can perform deletion from both ends but, Insertion can only be done from one end. Even though it ruins the basic definition of the queue we are considering this due to the requirements and also by using this method the queue can be very customized.

Output Restricted Deque

An Output-restricted deque is a type of queue where we can insert the required elements on both ends precisely from both rear and front. But, Deletion can be done from only one end.

Operations on Deque

1. **InsertFront()** : The function InsertFront can be used to add an element to the queue at the beginning of the queue.
2. **InsertRear()** : The function InsertFront can be used to add an element to the queue at the end of the queue.
3. **DeleteFront()** : The function DeleteFront() is used to delete the starting element in the queue, it is the conventional method of deque in a normal queue.
4. **DeleteRear()** : The function DeleteRear() is used to delete the last element in the queue, it is the conventional method of popping in a stack.
5. **GetFront()** : We can find the data stored in the front element using this function. Since the front pointer always points to the first element, we can simply access its data.
6. **GetRear()** : We can find the last entered element into the queue using the function GetRear(). As the rear pointer always points to the last element of the queue, we can simply get the last element by accessing its data.
7. **IsEmpty()** : This condition is usually prompted when we try to delete an element from an empty queue. We can check this condition by checking whether the rear and the front pointer point to the same node except the first node, then, we can print that the queue is empty.

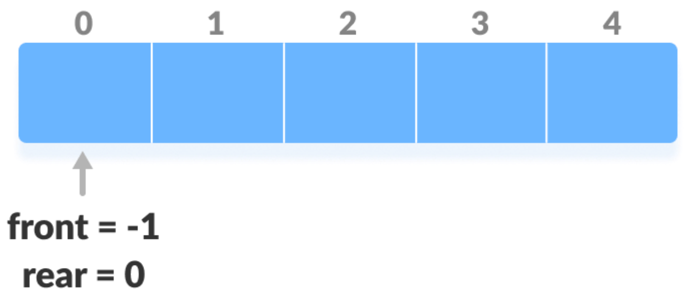
8. **IsFull()** : This is a special function as it can only be checked if the queue is implemented using arrays only, As linked lists use dynamic allocation, there is no condition that the queue is full.

Operations on a Deque

Below is the circular array implementation of deque. In a circular array, if the array is full, we start from the beginning.
But in a linear array implementation, if the array is full, no more elements can be inserted. In each of the operations below, if the array is full, "overflow message" is thrown.

Before performing the following operations, these steps are followed.

1. Take an array (deque) of size n.
2. Set two pointers front = -1 and rear = 0.

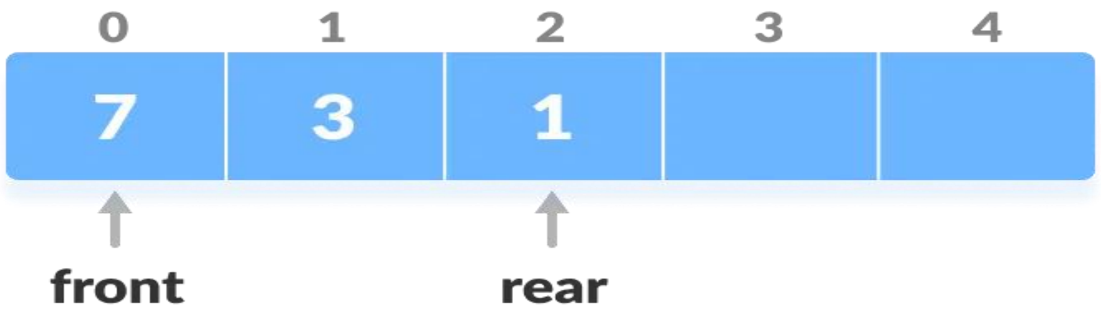


1. Insert at the Front

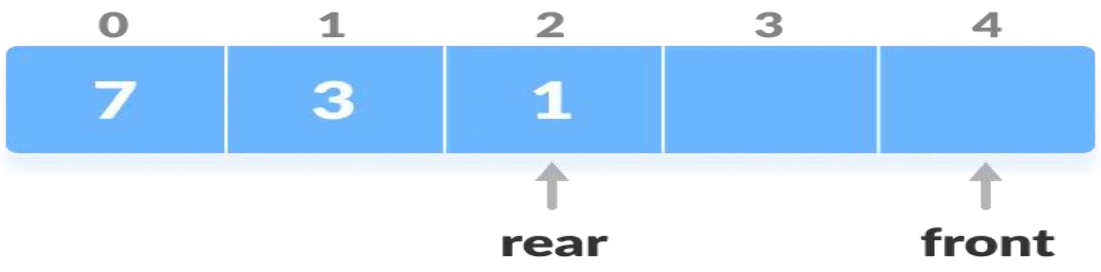
This operation adds an element at the front.

1. Check if the deque is full. Check the position of front

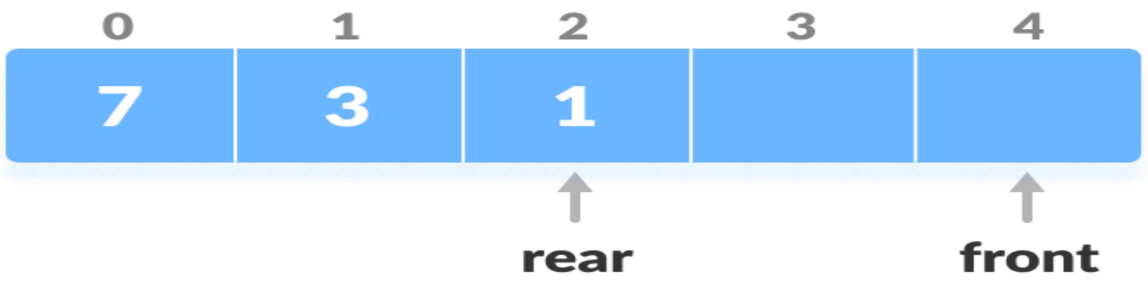
2. If the deque is full (i.e. (front == 0 && rear == n - 1) || (front == rear + 1)), insertion operation cannot be performed (**overflow condition**).

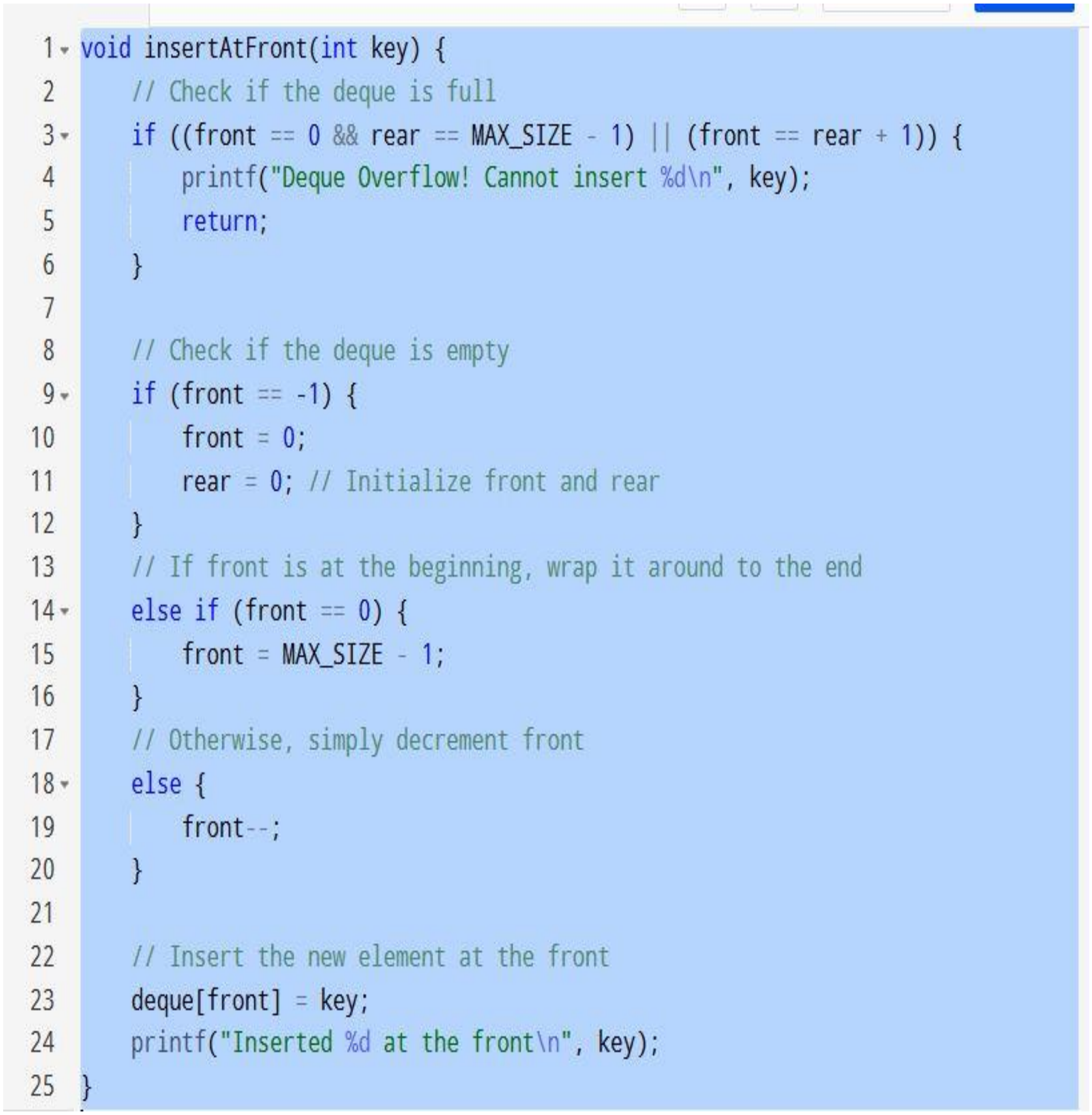


3. If the deque is empty, reinitialize front = 0. And, add the new key into array[front].
4. If front = 0, reinitialize front = n-1 (last index).



5. Else, decrease front by 1.
   Add the new key 5 into array[front].





```c
void insertAtFront(int key) {
    // Check if the deque is full
    if ((front == 0 && rear == MAX_SIZE - 1) || (front == rear + 1)) {
        printf("Deque Overflow! Cannot insert %d\n", key);
        return;
    }

    // Check if the deque is empty
    if (front == -1) {
        front = 0;
        rear = 0; // Initialize front and rear
    }
    // If front is at the beginning, wrap it around to the end
    else if (front == 0) {
        front = MAX_SIZE - 1;
    }
    // Otherwise, simply decrement front
    else {
        front--;
    }

    // Insert the new element at the front
    deque[front] = key;
    printf("Inserted %d at the front\n", key);
}
```

#include <stdio.h>

#define MAX_SIZE 5 // Maximum size of the deque

```
int deque[MAX_SIZE];
int front = -1, rear = -1; // Global variables for front and rear


void insertAtFront(int key) {
    // Check if the deque is full
    if ((front == 0 && rear == MAX_SIZE - 1) || (front == rear + 1)) {
        printf("Deque Overflow! Cannot insert %d\n", key);
        return;
    }


    // Check if the deque is empty
    if (front == -1) {
        front = 0;
        rear = 0; // Initialize front and rear
    }
    // If front is at the beginning, wrap it around to the end
    else if (front == 0) {
        front = MAX_SIZE - 1;
    }
    // Otherwise, simply decrement front
    else {
        front--;
    }
```

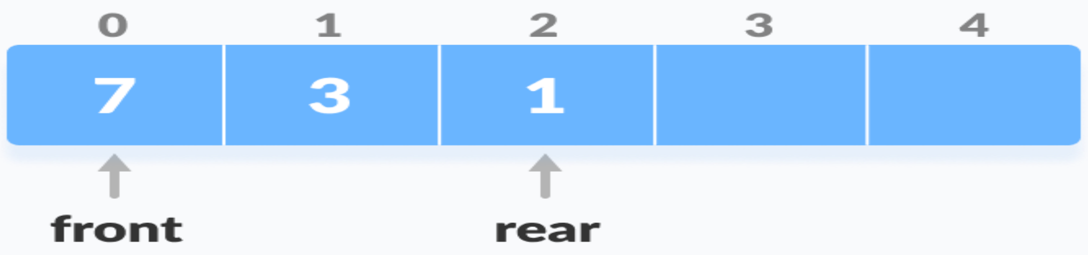
```c
    // Insert the new element at the front
    deque[front] = key;
    printf("Inserted %d at the front\n", key);
}

// Function to display the deque
void display() {
    if (front == -1) {
        printf("Deque is empty\n");
        return;
    }

    printf("Deque elements: ");
    int i = front;
    while (1) {
        printf("%d ", deque[i]);
        if (i == rear) break;
        i = (i + 1) % MAX_SIZE; // Circular traversal
    }
    printf("\n");
}
```

```
int main() {
    // Example usage of the deque
    insertAtFront(10);
    insertAtFront(20);
    insertAtFront(30);
    insertAtFront(40);
    insertAtFront(50); // This will fill the deque

    display();

    // Attempt to insert when deque is full
    insertAtFront(60);
    return 0;
}
```

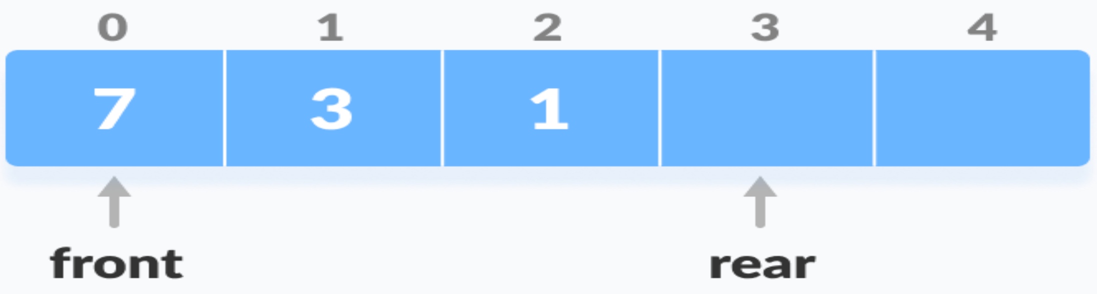
2. Insert at the Rear

This operation adds an element to the rear.

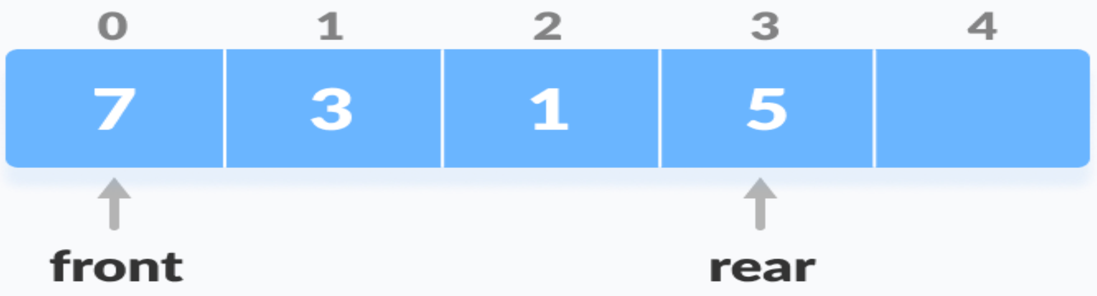
1. Check if the deque is full.



2. If the deque is full, insertion operation cannot be performed (**overflow condition**).
3. If the deque is empty, reinitialize `rear = 0`. And, add the new key into `array[rear]`.
4. If `rear = n - 1`, reinitialize `real = 0` (first index).
5. Else, increase `rear` by 1.



6. Add the new key `5` into `array[rear]`.



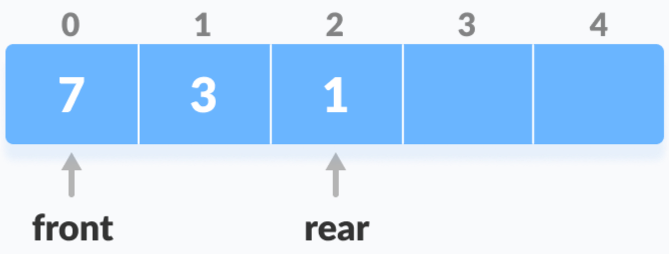
Same as Circular queue enqueue method

```
#include <stdio.h>

#define MAX_SIZE 5  // Maximum size of the deque


// Global variables for the deque

int queue[MAX_SIZE];

int front = -1, rear = -1;
```

```
// Function to insert an element at the rear
void insertRear(int element) {
    // Check if the deque is full
    if ((rear + 1) % MAX_SIZE == front) {
        printf("Deque is full. Cannot insert %d at the rear.\n", element);
        return;
    }

    // Check if the deque is empty
    if (front == -1) {
        front = 0;  // Initialize front if the deque is empty
    }

    // Circular increment for rear
    rear = (rear + 1) % MAX_SIZE;
    queue[rear] = element;

    printf("Inserted %d at the rear.\n", element);
}

// Function to display the deque
void display() {
    if (front == -1) {
        printf("Deque is empty.\n");
        return;
    }

    printf("Deque elements: ");
    int i = front;
    while (1) {
        printf("%d ", queue[i]);
        if (i == rear) break;
        i = (i + 1) % MAX_SIZE;  // Circular increment
    }
    printf("\n");
}

// Main function to test the insertion at the rear
int main() {
    insertRear(10);
    insertRear(20);
    insertRear(30);
    insertRear(40);
    insertRear(50);  // Should show that the deque is full
    display();
    return 0;

}
```

3. Delete from the Front

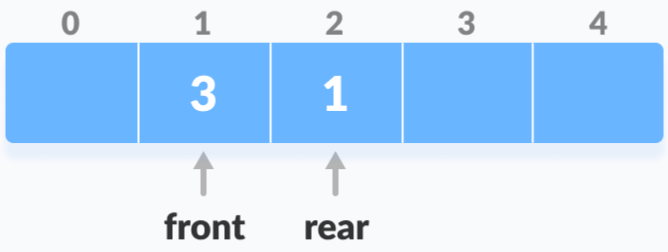
The operation deletes an element from the `front`.



1. Check if the deque is empty.

   Check if deque is empty

2. If the deque is empty (i.e. `front = -1`), deletion cannot be performed (**underflow condition**).
3. If the deque has only one element (i.e. `front = rear`), set `front = -1` and `rear = -1`.
4. Else if `front` is at the last index (i.e. `front = n - 1`), set `front = 0`.



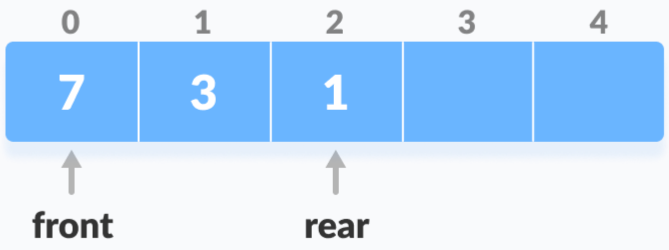
5. Else, `front = front + 1`.

   Increase the front

Same as circular queue dequeue operation program.

4. Delete from the Rear

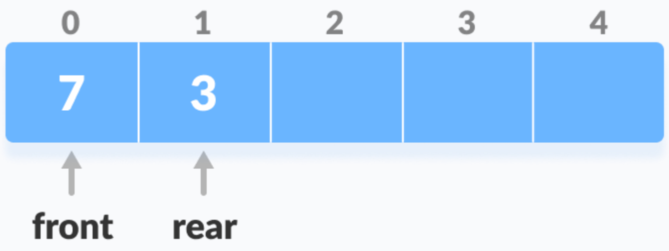
This operation deletes an element from the `rear`.



1. Check if the deque is empty.

   Check if deque is empty

2. If the deque is empty (i.e. `front = -1`), deletion cannot be performed (**underflow condition**).
3. If the deque has only one element (i.e. `front = rear`), set `front = -1` and `rear = -1`, else follow the steps below.
4. If `rear` is at the first index (i.e. `rear = 0`), reinitialize `rear = n - 1`.



5. Else, `rear = rear - 1`. Decrease the rear

Program

void deleteRear() {

    // Inline check if the deque is empty

    if (front == -1) {

```
        printf("Deque is empty. Cannot delete from the rear.\n");
        return;
    }

    // Check if the deque has only one element
    if (front == rear) {
        printf("Deleted %d from the rear.\n", queue[rear]);
        front = -1;
        rear = -1;  // Reset the deque
    } else if (rear == 0) {
        // If rear is at the first index, wrap it around to the last index
        printf("Deleted %d from the rear.\n", queue[rear]);
        rear = MAX_SIZE - 1;
    } else {
        // Otherwise, simply decrease the rear index
        printf("Deleted %d from the rear.\n", queue[rear]);
        rear--;
    }
}
```