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

---

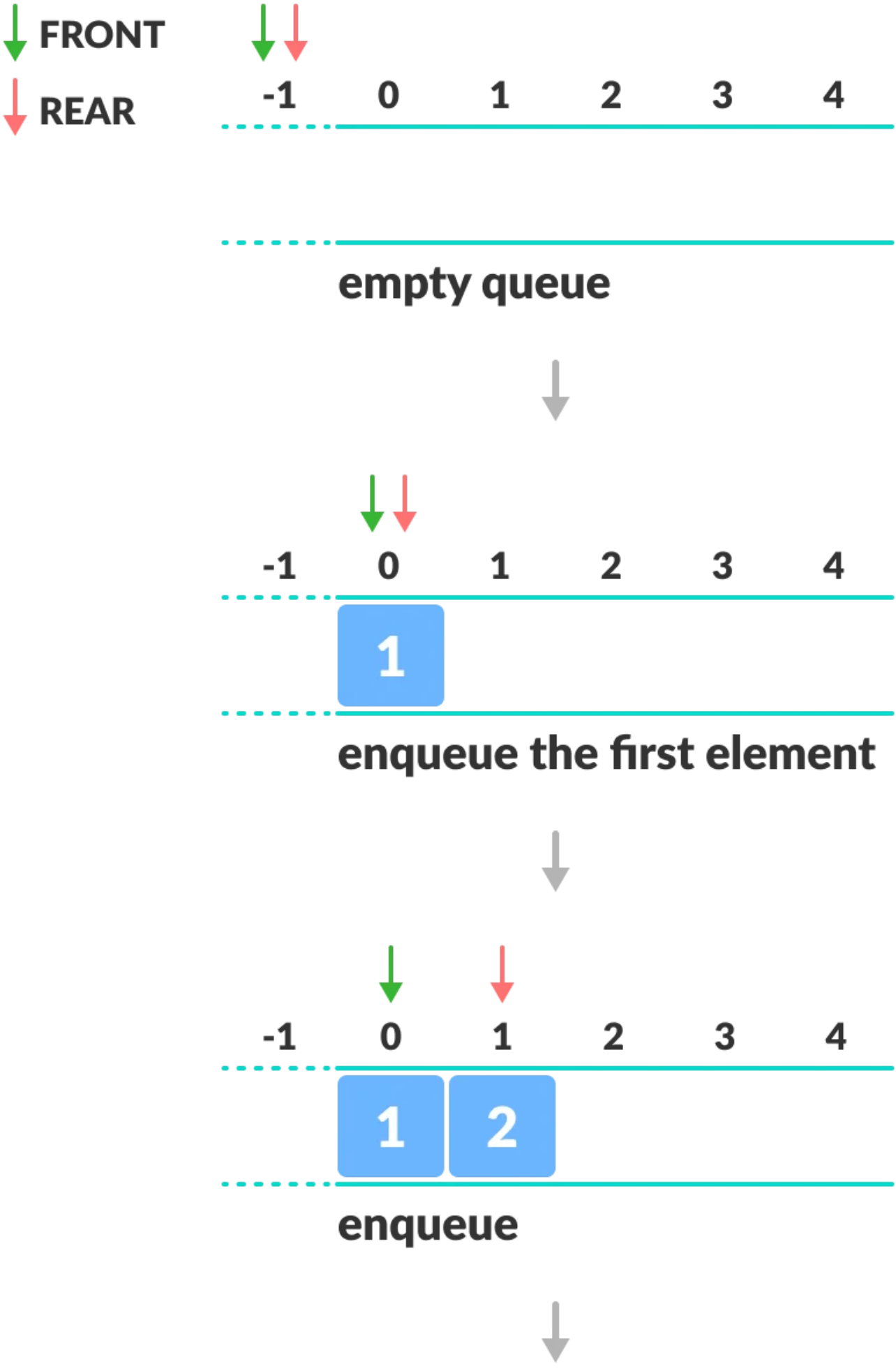
Working of Queue

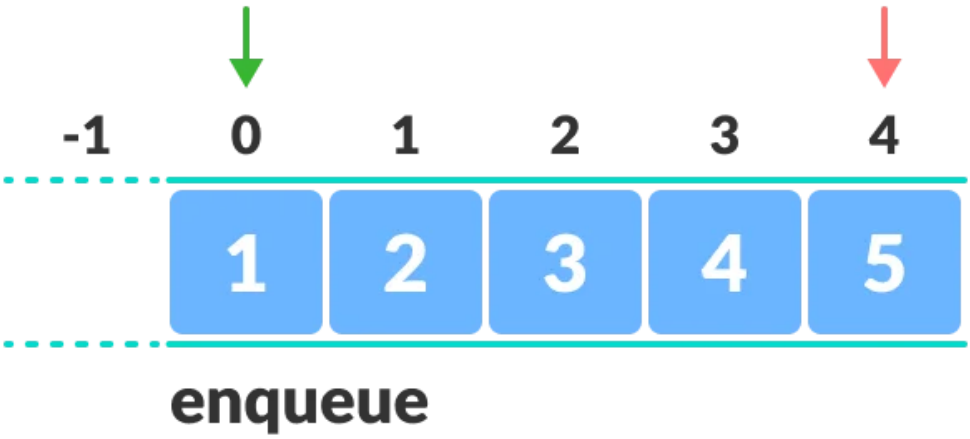
Queue operations work as follows:

- 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

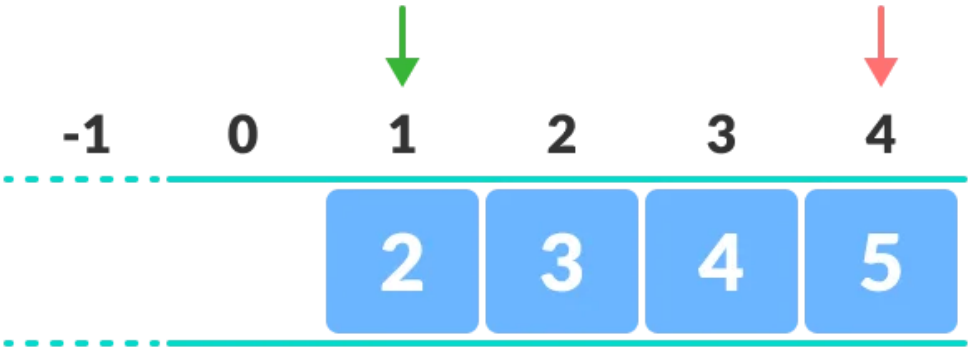
Enqueue Operation

- 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`

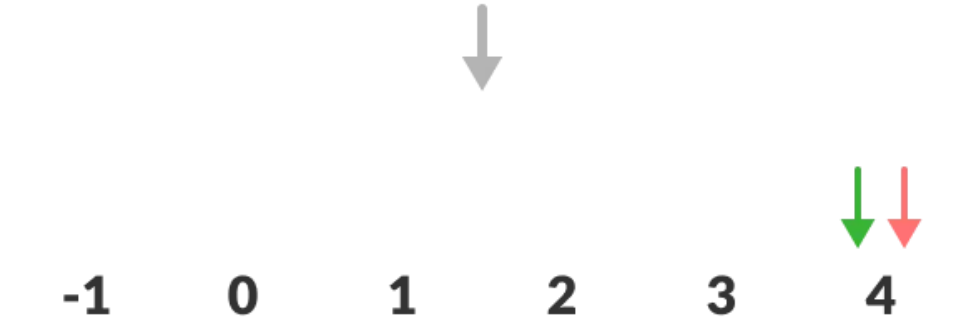

FRONT
REAR
-1
0
1
2
3
4
empty queue
-1
0
1
2
3
4
1
enqueue the first element
-1
0
1
2
3
4
1
2
enqueue



enqueue



dequeue



dequeue the last element

empty queue

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 Implementations

```
// 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

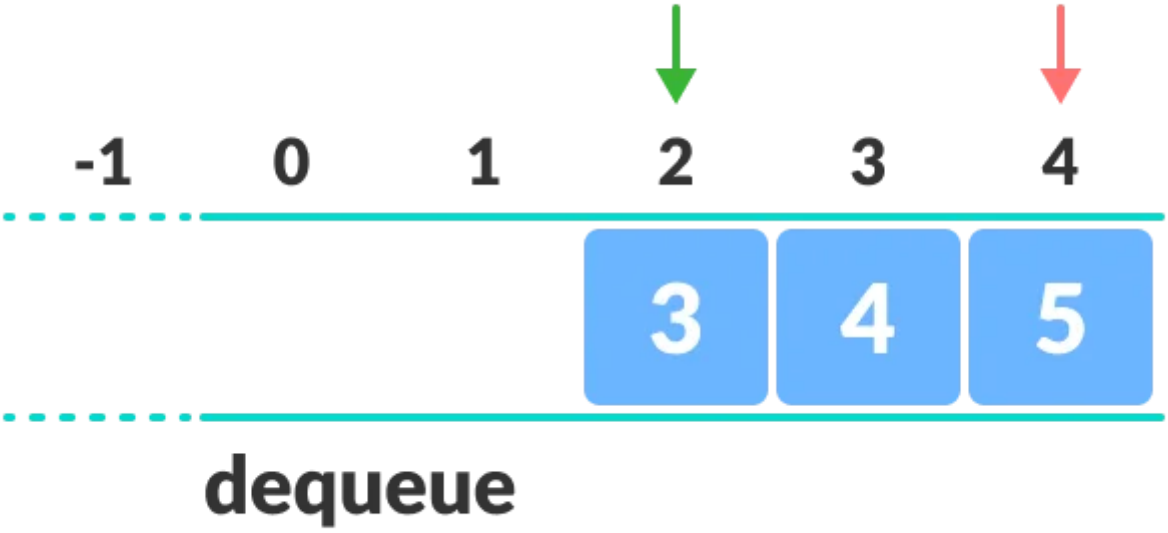
As you can see in the image below, after a bit of enqueuing and dequeuing, the size of the queue has been reduced.

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.

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.


-1
0
1
2
3
4
3
4
5
dequeue

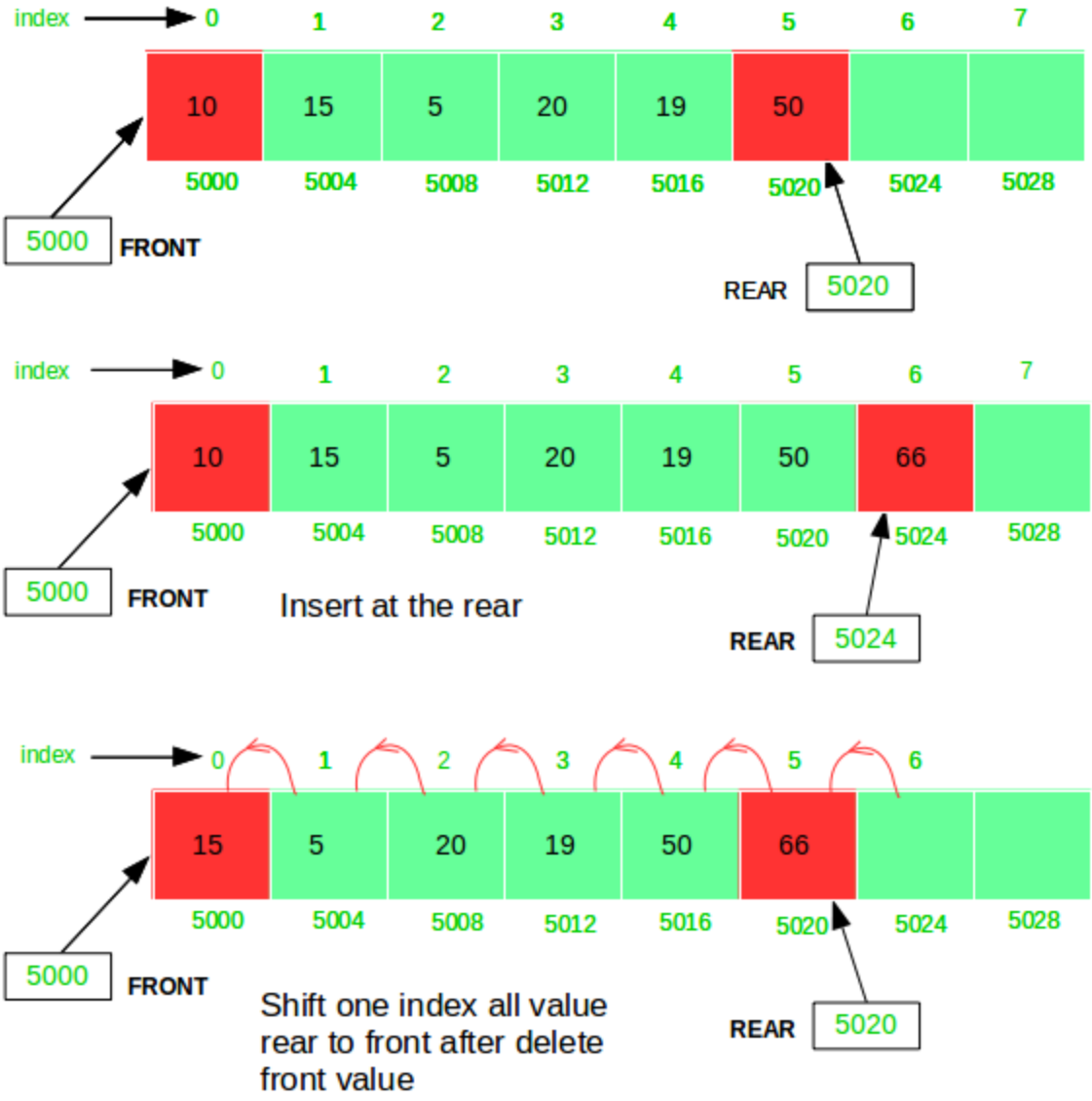
How to implement Queue using Array?

To implement a queue using an array,

- create an array arr of size **n** and
- take two variables **front** and **rear** both of which will be initialized to 0 which means the queue is currently empty.
- Element
    - rear is the index up to which the elements are stored in the array and
    - front is the index of the first element of the array.

Now, some of the implementations of queue operations are as follows:

- ***Enqueue:*** *Addition of an element to the queue. Adding an element will be performed after checking whether the queue is full or not. If rear < n which indicates that the array is not full then store the element at arr[rear] and increment rear by 1 but if rear == n then it is said to be an Overflow condition as the array is full.*
- ***Dequeue:*** *Removal of an element from the queue. An element can only be deleted when there is at least an element to delete i.e. rear > 0. Now, the element at arr[front] can be deleted but all the remaining elements have to shift to the left by one position in order for the dequeue operation to delete the second element from the left on another dequeue operation.*
- ***Front:*** *Get the front element from the queue i.e. arr[front] if the queue is not empty.*
- ***Display:*** *Print all elements of the queue. If the queue is non-empty, traverse and print all the elements from the index front to rear.*



Below is the implementation of a queue using an array:

In the below code , we are initializing front and rear as 0, but in general we have to initialize it with -1.
If we assign rear as 0, rear will always point to next block of the end element, in fact , rear should point the index of last element,
eg. When we insert element in queue , it will add in the end i.e. after the current rear and then point the rear to the new element ,
According to the following code:
IN the first dry run, front=rear = 0;

in void queueEnqueue(int data)

else part will be executed,

so arr[rear] =data;// rear =0, rear pointing to the latest element

rear++; //now rear = 1, rear pointing to the next block after end element not the end element

//that's against the original definition of rear

```
// C++ program to implement a queue using an array
#include <bits/stdc++.h>
using namespace std;

struct Queue {
    int front, rear, capacity;
    int* queue;
    Queue(int c)
    {
        front = rear = 0;
        capacity = c;
        queue = new int;
    }

    ~Queue() { delete[] queue; }

    // function to insert an element
    // at the rear of the queue
    void queueEnqueue(int data)
    {
        // check queue is full or not
        if (capacity == rear) {
            printf("\nQueue is full\n");
            return;
        }

        // insert element at the rear
        else {
            queue[rear] = data;
            rear++;
        }
        return;
    }

    // function to delete an element
    // from the front of the queue
    void queueDequeue()
    {
        // if queue is empty
        if (front == rear) {
            printf("\nQueue is  empty\n");
            return;
        }

        // shift all the elements from index 2 till rear
        // to the left by one
```

```
        else {
            for (int i = 0; i < rear - 1; i++) {
                queue[i] = queue[i + 1];
            }

            // decrement rear
            rear--;
        }
        return;
    }

    // print queue elements
    void queueDisplay()
    {
        int i;
        if (front == rear) {
            printf("\nQueue is Empty\n");
            return;
        }

        // traverse front to rear and print elements
        for (i = front; i < rear; i++) {
            printf(" %d <-- ", queue[i]);
        }
        return;
    }

    // print front of queue
    void queueFront()
    {
        if (front == rear) {
            printf("\nQueue is Empty\n");
            return;
        }
        printf("\nFront Element is: %d", queue[front]);
        return;
    }
};

// Driver code
int main(void)
{
    // Create a queue of capacity 4
    Queue q(4);

    // print Queue elements
    q.queueDisplay();

    // inserting elements in the queue
    q.queueEnqueue(20);
    q.queueEnqueue(30);
    q.queueEnqueue(40);
    q.queueEnqueue(50);

    // print Queue elements
```

```
    q.queueDisplay();

    // insert element in the queue
    q.queueEnqueue(60);

    // print Queue elements
    q.queueDisplay();

    q.queueDequeue();
    q.queueDequeue();

    printf("\n\nafter two node deletion\n\n");

    // print Queue elements
    q.queueDisplay();

    // print front of the queue
    q.queueFront();

    return 0;
}
```

Output

```
Queue is Empty

 20 <--  30 <--  40 <--  50 <--

Queue is full

 20 <--  30 <--  40 <--  50 <--


after two node deletion


 40 <--  50 <--

Front Element is: 40
```

Time Complexity: O(1) for Enqueue (element insertion in the queue) as we simply increment pointer and put value in array and O(N) for Dequeue (element removing from the queue) as we use for loop to implement that.
Auxiliary Space: O(N), as here we are using an N size array for implementing Queue
Optimizations :
We can implement both enqueue and dequeue operations in O(1) time. To achieve this, we can either use Linked List Implementation of Queue or circular array implementation of queue.