QUEUES - DATA STRUCTURES AND ALGORITHMS

---

1. INTRODUCTION TO QUEUES

What is a Queue?

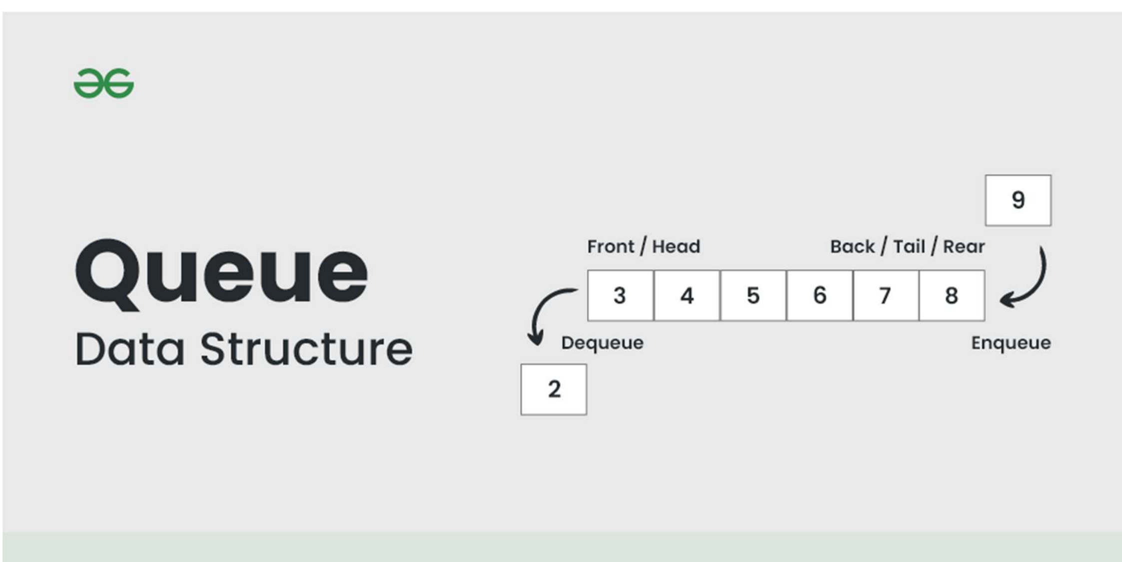
A **queue** is a linear data structure that follows the **First-In-First-Out (FIFO)** principle. It means that the element inserted first will be removed first — just like a real-life queue at a ticket counter.

A **Queue Data Structure** is a fundamental concept in computer science used for storing and managing data in a specific order.

- It follows the principle of "**First in, First out**" **(FIFO)**, where the first element added to the queue is the first one to be removed.
- It is used as a buffer in computer systems where we have speed mismatch between two devices that communicate with each other. For example, CPU and keyboard and two devices in a network
- Queue is also used in Operating System algorithms like CPU Scheduling and Memory Management, and many standard algorithms like Breadth First Search of Graph, Level Order Traversal of a Tree.

Basics

- Introduction
- Basic Operations
- Array Implementations
- Linked List Implementation
- Applications



Implementations in various Programming Languages

- Queue in C++ STL
- Queue In Java
- Queue In Python
- Queue In C#
- Queue in JavaScript
- Queue in Go Language
- Queue in Scala

Implementing Other Data Structures Using Queue

- Implement a stack using single queue
- Implement Queue using Stacks
- LRU Cache Implementation

Easy Problems on Queue

- Implement Stack using Queues
- Minimum Depth of a Binary Tree
- BFS for a Graph
- Detect cycle in an undirected graph using BFS
- Right View of a Binary Tree
- Check whether a graph is Bipartite

Medium Problems on Queue

- Implement k Queues in a single array
- Flatten a multilevel linked list
- Level with maximum number of nodes
- Find if there is a path between two vertices in a directed graph
- Print all nodes between two given levels in Binary Tree
- Find next right node of a given key
- Minimum steps to reach target by a Knight
- Islands in a graph using BFS
- Find the first non-repeating in a stream

Hard Problems on Queue

- Sliding Window Maximum (Maximum of all subarrays of size K)
- Flood Fill Algorithm
- Minimum time required to rot all oranges
- Shortest path in a Binary Maze
- An Interesting Method to Generate Binary Numbers from 1 to n
- Maximum cost path from source node to destination
- Shortest distance between two cells in a matrix or grid
- Snake and Ladder Problem
- Find shortest safe route in a path with landmines
- Count all possible walks from a source to a destination with exactly K edges
- Minimum Cost of Simple Path between two nodes in a directed and weighted graph
- Minimum Cost Path in a directed graph via given set of intermediate nodes
- Find the first circular tour that visits all petrol pumps

Quick Links:

- 'Practice Problems' on Queue
- 'Quizzes' on Queue
- **Learn Data Structure and Algorithms | DSA Tutorial**

Why use Queues?

- Efficient **task scheduling**
- Managing resources shared among multiple consumers (e.g., **CPU scheduling**)
- Buffer handling (e.g., **IO buffers**, **print queues**, **network data packets**)
- Useful in **BFS**, **caching**, **streaming algorithms**

Basic Operations

Operation	Description
Enqueue	Add element at the rear (tail)
Dequeue	Remove element from the front

Operation	Description
Peek	View the front element
isEmpty	Check if the queue is empty
isFull	(for array-based queue)

---

2. <u>REAL-LIFE ANALOGY</u>

Queues are everywhere:

- **Ticket counters** – People join at the end, served from the front.
- **Print queue** – Print jobs are handled in the order they arrive.
- **Call center support** – Calls are answered based on arrival time.

This analogy helps you understand the **FIFO nature** of queues.

---

3. <u>QUEUE USING ARRAYS</u>

Structure

- Use a **fixed-size array**
- Maintain two indices: front and rear
- front: Points to the first (oldest) element
- rear: Points to the last (newest) element inserted

Java Code – Queue using Array

```
class ArrayQueue {
    private int[] queue;
    private int front, rear, capacity;

    public ArrayQueue(int size) {
        capacity = size;
        queue = new int[capacity];
        front = 0;
        rear = -1;
    }
```

```java
    public void enqueue(int x) {
        if (rear == capacity - 1) {
            System.out.println("Queue Overflow");
            return;
        }
        queue[++rear] = x;
    }

    public int dequeue() {
        if (isEmpty()) {
            System.out.println("Queue Underflow");
            return -1;
        }
        return queue[front++];
    }

    public int peek() {
        return isEmpty() ? -1 : queue[front];
    }

    public boolean isEmpty() {
        return front > rear;
    }
}
```

Time Complexities

Operation	Time
Enqueue	O(1)
Dequeue	O(1)

Operation	Time
Peek	O(1)

---

4. QUEUE USING LINKED LIST

Queue - Linked List Implementation

Linked List implementation of the queue data structure is discussed and implemented. Print '-1' if the queue is empty.

Approach: To solve the problem follow the below idea:

*we maintain two pointers, **front** and **rear**. The front points to the first item of the queue and rear points to the last item.*

- ***enQueue():** This operation adds a new node after the rear and moves the rear to the next node.*

***deQueue():** This operation removes the front node and moves the front to the next node.*

Queue - Linked List Implementation

Linked List implementation of the queue data structure is discussed and implemented. Print '-1' if the queue is empty.
Approach: To solve the problem follow the below idea:
*we maintain two pointers, **front** and **rear**. The front points to the first item of the queue and rear points to the last item.*

- ***enQueue():** This operation adds a new node after the rear and moves the rear to the next node.*
- ***deQueue():** This operation removes the front node and moves the front to the next node.*

Follow the below steps to solve the problem:

- Create a class Node with data members integer data and Node* next
  - A parameterized constructor that takes an integer x value as a parameter and sets data equal to x and next as NULL
- Create a class Queue with data members Node front and rear
- Enqueue Operation with parameter x:
  - Initialize Node* temp with data = x
  - If the rear is set to NULL then set the front and rear to temp and return(Base Case)
  - Else set rear next to temp and then move rear to temp
- Dequeue Operation:
  - If the front is set to NULL return(Base Case)
  - Initialize Node temp with front and set front to its next
  - If the front is equal to NULL then set the rear to NULL
  - Delete temp from the memory

Below is the Implementation of the above approach:

```
// Node class definition
class Node {
```

```
    int data;
    Node next;
    Node(int new_data) {
        data = new_data;
        next = null;
    }
}

// Queue class definition
class Queue {
    private Node front;
    private Node rear;
    public Queue() {
        front = rear = null;
    }

    // Function to check if the queue is empty
    public boolean isEmpty() {
        return front == null;
    }

    // Function to add an element to the queue
    public void enqueue(int new_data) {
        Node new_node = new Node(new_data);
        if (isEmpty()) {
            front = rear = new_node;
            printQueue();
            return;
        }
        rear.next = new_node;
        rear = new_node;
        printQueue();
    }

    // Function to remove an element from the queue
    public void dequeue() {
        if (isEmpty()) {
            return;
        }
        Node temp = front;
        front = front.next;
        if (front == null) rear = null;
        temp = null;
        printQueue();
    }

    // Function to print the current state of the queue
    public void printQueue() {
```

```java
        if (isEmpty()) {
            System.out.println("Queue is empty");
            return;
        }
        Node temp = front;
        System.out.print("Current Queue: ");
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
        System.out.println();
    }
}

// Driver code to test the queue implementation
public class Main {
    public static void main(String[] args) {
        Queue q = new Queue();

        // Enqueue elements into the queue
        q.enqueue(10);
        q.enqueue(20);

        // Dequeue elements from the queue
        q.dequeue();
        q.dequeue();

        // Enqueue more elements into the queue
        q.enqueue(30);
        q.enqueue(40);
        q.enqueue(50);

        // Dequeue an element from the queue (this should print 30)
        q.dequeue();
    }
}
```

Output

```
Current Queue: 10

Current Queue: 10 20

Current Queue: 20

Queue is empty

Current Queue: 30

Current Queue: 30 40

Current Queue: 30 40 50
```

```
Current Queue: 40 50
```

Time Complexity: O(1), The time complexity of both operations enqueue() and dequeue() is O(1) as it only changes a few pointers in both operations
Auxiliary Space: O(1), The auxiliary Space of both operations enqueue() and dequeue() is O(1) as constant extra space is required

- Queue using Linked List

Follow the below steps to solve the problem:

- Create a class Node with data members integer data and Node* next
  - A parameterized constructor that takes an integer x value as a parameter and sets data equal to x and next as NULL
- Create a class Queue with data members Node front and rear
- Enqueue Operation with parameter x:
  - Initialize Node* temp with data = x
  - If the rear is set to NULL then set the front and rear to temp and return(Base Case)
  - Else set rear next to temp and then move rear to temp
- Dequeue Operation:
  - If the front is set to NULL return(Base Case)
  - Initialize Node temp with front and set front to its next
  - If the front is equal to NULL then set the rear to NULL
  - Delete temp from the memory

Below is the Implementation of the above approach:

```
// Node class definition

class Node {

   int data;

   Node next;

   Node(int new_data) {

      data = new_data;

      next = null;

   }

}
```

```
// Queue class definition
class Queue {
    private Node front;
    private Node rear;
    public Queue() {
        front = rear = null;
    }

    // Function to check if the queue is empty
    public boolean isEmpty() {
        return front == null;
    }

    // Function to add an element to the queue
    public void enqueue(int new_data) {
        Node new_node = new Node(new_data);
        if (isEmpty()) {
            front = rear = new_node;
            printQueue();
            return;
        }
        rear.next = new_node;
        rear = new_node;
        printQueue();
    }

    // Function to remove an element from the queue
    public void dequeue() {
        if (isEmpty()) {
            return;
```

```java
        }
        Node temp = front;
        front = front.next;
        if (front == null) rear = null;
        temp = null;
        printQueue();
    }

    // Function to print the current state of the queue
    public void printQueue() {
        if (isEmpty()) {
            System.out.println("Queue is empty");
            return;
        }
        Node temp = front;
        System.out.print("Current Queue: ");
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
        System.out.println();
    }
}

// Driver code to test the queue implementation
public class Main {
    public static void main(String[] args) {
        Queue q = new Queue();

        // Enqueue elements into the queue
```

```
        q.enqueue(10);
        q.enqueue(20);

        // Dequeue elements from the queue
        q.dequeue();
        q.dequeue();

        // Enqueue more elements into the queue
        q.enqueue(30);
        q.enqueue(40);
        q.enqueue(50);

        // Dequeue an element from the queue (this should print 30)
        q.dequeue();
    }
}
```

Output

Current Queue: 10

Current Queue: 10 20

Current Queue: 20

Queue is empty

Current Queue: 30

Current Queue: 30 40

Current Queue: 30 40 50

Current Queue: 40 50

Time Complexity: O(1), The time complexity of both operations enqueue() and dequeue() is O(1) as it only changes a few pointers in both operations
Auxiliary Space: O(1), The auxiliary Space of both operations enqueue() and dequeue() is O(1) as constant extra space is required

Advantages over Array Queue:

- **Dynamic size**
- No overflow unless system memory is exhausted

Java Code – Queue using Linked List

```
class Node {
    int data;
    Node next;
    Node(int d) { data = d; next = null; }
}

class LinkedListQueue {
    private Node front, rear;

    public void enqueue(int x) {
        Node newNode = new Node(x);
        if (rear == null) {
            front = rear = newNode;
            return;
        }
        rear.next = newNode;
        rear = newNode;
    }

    public int dequeue() {
        if (front == null) {
            System.out.println("Queue Underflow");
            return -1;
        }
        int val = front.data;
        front = front.next;
```

```
        if (front == null) rear = null;
        return val;
    }

    public int peek() {
        return front == null ? -1 : front.data;
    }

    public boolean isEmpty() {
        return front == null;
    }
}
```

---

5. CIRCULAR QUEUE

Why Circular?

- In a regular array queue, after multiple enqueues and dequeues, free space at the start of the array cannot be reused.
- Circular Queue **wraps around** using modulo arithmetic, solving this inefficiency.

Java Code – Circular Queue

```
class CircularQueue {
    private int[] queue;
    private int front, rear, size;

    public CircularQueue(int k) {
        queue = new int[k];
        front = -1;
        rear = -1;
        size = k;
    }
```

```
    public boolean enqueue(int value) {
        if ((rear + 1) % size == front) return false; // Full
        if (front == -1) front = 0;
        rear = (rear + 1) % size;
        queue[rear] = value;
        return true;
    }

    public boolean dequeue() {
        if (front == -1) return false; // Empty
        if (front == rear) {
            front = rear = -1;
        } else {
            front = (front + 1) % size;
        }
        return true;
    }

    public int Front() {
        return front == -1 ? -1 : queue[front];
    }

    public int Rear() {
        return rear == -1 ? -1 : queue[rear];
    }
}
```

Key Points:

- Efficient space usage
- Circular movement using: (rear + 1) % size

- Detect full queue condition using: (rear + 1) % size == front

---

6. DEQUE (DOUBLE ENDED QUEUE)

What is a Deque?

A **Deque** (pronounced "deck") is a **double-ended queue**, where insertion and deletion can happen from both ends — **front and rear**.

Use Cases:

- **Palindromic checks**
- **Sliding window maximums**
- **Undo operations**
- **Browser history navigation**

Java Code – Deque using Linked List

```
class Deque {
  LinkedList<Integer> deque;

  public Deque() {
    deque = new LinkedList<>();
  }

  public void insertFront(int x) {
    deque.addFirst(x);
  }

  public void insertRear(int x) {
    deque.addLast(x);
  }

  public void deleteFront() {
    if (!deque.isEmpty()) deque.removeFirst();
```

```
    }

    public void deleteRear() {
        if (!deque.isEmpty()) deque.removeLast();
    }

    public int getFront() {
        return deque.isEmpty() ? -1 : deque.getFirst();
    }

    public int getRear() {
        return deque.isEmpty() ? -1 : deque.getLast();
    }
}
```

---

7. SLIDING WINDOW MAXIMUM

Problem Statement

Given an array and an integer k, find the maximum for each sliding window of size k.

Naive Approach

Use a loop to find max in each window → Time: O(n*k)

Efficient Approach: Using Deque

Maintain a deque where:

- Elements are stored in decreasing order
- Front is always max of the window

Java Code – Sliding Window Max

```
public int[] maxSlidingWindow(int[] nums, int k) {
    Deque<Integer> dq = new ArrayDeque<>();
```

```
    int n = nums.length;
    int[] result = new int[n - k + 1];
    int ri = 0;

    for (int i = 0; i < nums.length; i++) {
        while (!dq.isEmpty() && dq.peek() < i - k + 1) dq.poll();
        while (!dq.isEmpty() && nums[dq.peekLast()] < nums[i]) dq.pollLast();
        dq.offer(i);
        if (i >= k - 1) result[ri++] = nums[dq.peek()];
    }

    return result;
}
```

Time Complexity: O(n)

---

8. MONOTONIC QUEUE

A **Monotonic Queue** maintains elements in a **sorted order** (increasing or decreasing). It's often used for **range queries** or **window problems**.

Properties

- Maintains **monotonicity** (increasing/decreasing)
- Supports **sliding window max/min** in O(n) time

Monotonic Queue for Min Element

```
class MonotonicQueue {
    Deque<Integer> dq = new ArrayDeque<>();

    public void push(int n) {
        while (!dq.isEmpty() && dq.getLast() > n)
            dq.removeLast();
```

```java
        dq.addLast(n);
    }

    public void pop(int n) {
        if (!dq.isEmpty() && dq.getFirst() == n)
            dq.removeFirst();
    }

    public int min() {
        return dq.getFirst();
    }
}
```

Use this in sliding windows to get **window minimum** in O(n) time.

Practice Problems

Problem	**Description**
Implement Queue using Stacks	Use 2 stacks to simulate queue behavior
Circular Tour (Petrol Pump)	Find start point to complete circle
Rotten Oranges	BFS using queue to track infected oranges
LRU Cache	Use doubly linked list + hashmap
First Non-Repeating Char in Stream	Use queue + hashmap
Maximum of All Subarrays of Size K	Sliding window maximum problem
Online Stock Span	Similar to NGE using stack/queue
Sum of Min/Max in all subarrays of size k	Monotonic queue trick

---

Further Reading and Tools

- **LeetCode**: Problems 239 (Sliding Window Max), 641 (Design Circular Deque)
- **GeeksforGeeks**: Queue Data Structures and Applications

- **Books**:
    - "Cracking the Coding Interview"
    - "Introduction to Algorithms" by Cormen
- **YouTube Channels**:
    - Take U Forward
    - Abdul Bari
    - TechDose

RECOMMENDED YOUTUBE LINKS

1. https://youtu.be/zp6pBNbUB2U?si=M4MGqUC5k7v6cono
2. https://youtu.be/va_6RmSrKCg?si=2tq8aiR0hwkrrTos
3. https://youtu.be/PvDoT79oHTs?si=Xwcsok9mepgJYFGH
4. https://youtu.be/rHQI4mrJ3cg?si=q1l5x3u8Hvge17c1
5. https://youtu.be/lXIcS3qXGMY?si=MAbpuveYLxCOd8uG
6. https://youtu.be/okr-XE8yTO8?si=crNrzPxWqSGkBCus
7. https://youtu.be/Khf9v67Ya30?si=3f-5a_Z_pBLd547h
8. https://www.youtube.com/live/4_xdaIsY2nk?si=3Tc4YLFRtj4X8BZG
9. https://youtu.be/4mKKolshFD0?si=flBZh-dUCpZW9_YE
10. https://youtu.be/Uu6Y8aDSDww?si=09CMCIiLUo_abzBG