Queue Data Structure

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 to Queue Data Structure

Queue 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.

FIFO Principle in Queue:

FIFO Principle states that the first element added to the Queue will be the first one to be removed or processed. So, Queue is like a line of people waiting to purchase tickets, where the first person in line is the first person served. (i.e. First Come First Serve).

Basic Terminologies of Queue

- Front: Position of the entry in a queue ready to be served, that is, the first entry that will be removed from the queue, is called the front of the queue. It is also referred as the head of the queue.
- Rear: Position of the last entry in the queue, that is, the one most recently added, is called the rear of the queue. It is also referred as the tail of the queue.
- Size: Size refers to the current number of elements in the queue.
- Capacity: Capacity refers to the maximum number of elements the queue can hold.

Representation of Queue

Queue Operations

1. Enqueue: Adds an element to the end (rear) of the queue. If the queue is full, an overflow error occurs.

2. Dequeue: Removes the element from the front of the queue. If the queue is empty, an underflow error occurs.
3. Peek/Front: Returns the element at the front without removing it.
4. **Size: Returns the number of elements in the queue.**
5. **isEmpty: Returns true if the queue is empty, otherwise false.**
6. **isFull: Returns true if the queue is full, otherwise false.**

Types of Queues

Queue data structure can be classified into 4 types:

1. Simple Queue: Simple Queue simply follows FIFO Structure. We can only insert the element at the back and remove the element from the front of the queue. A simple queue is efficiently implemented either using a linked list or a circular array.
2. Double-Ended Queue (Deque): In a double-ended queue the insertion and deletion operations, both can be performed from both ends. They are of two types:
    - Input Restricted Queue: This is a simple queue. In this type of queue, the input can be taken from only one end but deletion can be done from any of the ends.
    - Output Restricted Queue: This is also a simple queue. In this type of queue, the input can be taken from both ends but deletion can be done from only one end.
3. Priority Queue: A priority queue is a special queue where the elements are accessed based on the priority assigned to them. They are of two types:

- Ascending Priority Queue: In Ascending Priority Queue, the elements are arranged in increasing order of their priority values. Element with smallest priority value is popped first.
- Descending Priority Queue: In Descending Priority Queue, the elements are arranged in decreasing order of their priority values. Element with largest priority is popped first.

Basic Operations for Queue in Data Structure

Queue 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.

Basic Operations on Queue

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.
- **getFront()-** Acquires the data element available at the front node of the queue without deleting it.
- **getRear() -** 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.

Queue Data Structure

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.

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.

Operation 3: getFront()

This operation returns the element at the front end of the queue without removing it.

The following steps are taken to perform the getFront() operation:

- If the queue is empty, return the most minimum value (e.g., -1).
- Otherwise, return the front value of the queue.

getFront

Operation 4: getRear()

This operation returns the element at the rear end without removing it.

The following steps are taken to perform the rear operation:

- If the queue is empty return the most minimum value.
- otherwise, return the rear value.

getRear

Operation 5: isEmpty()

This operation returns a boolean value that indicates whether the queue is empty or not.

The following steps are taken to perform the Empty operation:

- check if front value is equal to -1 or not, if yes then return true means queue is empty.
- Otherwise return false, means queue is not empty.

isEmpty

Operation 6: size()

This operation returns the size of the queue i.e. the total number of elements it contains.

size()

Code Implementation of all the operations:

```
import java.util.LinkedList;

import java.util.Queue;


public class GfG {

    static Queue<Integer> q = new LinkedList<>();
```

```java
static boolean isEmpty() {
    return q.isEmpty();
}

static void qEnqueue(int data) {
    q.add(data);
}

static void qDequeue() {
    if (isEmpty()) {
        return;
    }
    q.poll();
}

static int getFront() {
    if (isEmpty()) return -1;
    return q.peek();
}

static int getRear() {
    if (isEmpty()) return -1;
    return ((LinkedList<Integer>) q).getLast();
}

public static void main(String[] args) {
    qEnqueue(1);
    qEnqueue(8);
    qEnqueue(3);
    qEnqueue(6);
```

```
        qEnqueue(2);

        if (!isEmpty()) {
            System.out.print("Queue after enqueue operation: ");
            for (int num : q) {
                System.out.print(num + " ");
            }
            System.out.println();
        }

        System.out.println("Front: " + getFront());
        System.out.println("Rear: " + getRear());

        System.out.println("Queue size: " + q.size());

        qDequeue();

        System.out.println("Is queue empty? " + (isEmpty() ? "Yes" : "No"));
    }
}
```

Output

```
Queue after enqueue operation: 1 8 3 6 2
Front: 1
Rear: 2
Queue size: 5
Is queue empty? No
```

Introduction and Array Implementation of Queue

Similar to stack,queue is a linear data structure that follows a particular order in which the operations are performed for storing data. The order is First In First Out **(FIFO)**. One can imagine a queue as a line of people waiting to receive something in sequential order which starts from the beginning of the line.

- It is an ordered list in which insertions are done at one end which is known as the rear and deletions are done from the other end known as the front.
- A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first.
- The difference between stack and queue is in removing an element. In a stack we remove the item that is most recently added while in a queue, we remove the item that is least recently added.

Queue Data structure

Simple Array implementation Of Queue:

For implementing the queue, we only need to keep track of two variables: **front and size**. We can find the rear as front + size - 1.

- The Enqueue operation is simple, we simply insert at the end of the array. This operation takes O(1) time
- The Dequeue operation is costly as we need remove from the beginning of the array. To remove an item, we need to move all items one position back. Hence this operations takes O(n) time.
- If we do otherwise that insert at the begin and delete from the end, then insert would become costly.

Operations	Complexity
Enqueue (insertion)	O(1)
Deque (deletion)	**O(n)**
Front (Get front)	O(1)
Rear (Get Rear)	O(1)
IsFull (Check queue is full or not)	O(1)
IsEmpty (Check queue is empty or not)	O(1)

Circular Array implementation Of Queue:

We can make all operations in O(1) time using circular array implementation.

- The idea is to treat the array as a circular buffer. We move front and rear using modular arithmetic
- When we insert an item, we increment front using modular arithmetic (which might leave some free space at the beginning of the array).
- When we delete an item, we decrement rear using modular arithmetic.

Queue - Linked List Implementation

In this article, the 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 nod*
- 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;
        }
        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