MST Qn Marked as RED 😊

Queue:

A **queue** is a linear data structure that follows the **First In, First Out (FIFO)** principle. In a queue, the first element added is the first one to be removed. It is similar to a real-life queue, like people standing in line for a service.

Queue as an ADT (Abstract Data Type):

An Abstract Data Type (ADT) is a data structure defined by a set of operations and the behavior of those operations, but not by its implementation. The details of how the operations are carried out are hidden from the user, and only the interface is provided.

A **queue** can be described as an ADT with the following operations:

1. **Enqueue**: Adds an element to the rear of the queue.
2. **Dequeue**: Removes the front element from the queue.
3. **Front**: Returns the element at the front of the queue without removing it.
4. **IsEmpty**: Checks if the queue is empty.
5. **IsFull**: Checks if the queue is full (in case of fixed-size queues).
6. **Size**: Returns the number of elements in the queue.

Queue Operations:

1. **Enqueue**:
   - Add an element to the rear of the queue.
   - If the queue is full, it may result in overflow.
2. **Dequeue**:
   - Remove an element from the front of the queue.
   - If the queue is empty, it may result in underflow.
3. **Front**:
   - Peek at the front element without removing it.
4. **IsEmpty**:
   - Checks whether the queue has no elements.
5. **IsFull**:
   - Checks whether the queue has reached its maximum size (in case of array-based implementation).

Array Representation of Queue:

In an **array-based queue**, the queue is represented by an array with two pointers (or indices):

- **front**: Points to the front element of the queue.
- **rear**: Points to the last element of the queue.

The array is of a fixed size, and elements are added at the rear and removed from the front. The queue wraps around when the rear reaches the end of the array (for circular implementation).

Operations:

1. **Enqueue**: Add elements at the rear.
2. **Dequeue**: Remove elements from the front.
3. **IsFull**: If (rear + 1) % size == front, the queue is full.
4. **IsEmpty**: If front == rear, the queue is empty.

Linked Representation of Queue:

In a **linked list-based queue**, the queue is represented by a linked list where:

- The **front** points to the first node (front of the queue).
- The **rear** points to the last node (rear of the queue).

This representation allows dynamic resizing, and we can enqueue or dequeue elements without worrying about array size limitations.

Operations:

1. **Enqueue**: Add elements to the rear by creating a new node and linking it.
2. **Dequeue**: Remove the node from the front.

Types of Queues:

1. **Circular Queue**:
   - In a **circular queue**, the rear and front are connected in a circular manner, meaning when the rear reaches the end of the array, it wraps around to the beginning if there is space.
   - It helps in efficiently utilizing the space in a fixed-size array.

Operations:

- **Enqueue**: Add elements at the rear, considering the circular nature.

   - **Dequeue**: Remove elements from the front, considering wrap-around behavior.

2. **Deque (Double-Ended Queue)**:
   - A **deque** allows insertion and deletion of elements from both ends (front and rear).
   - Operations:
     - **Insert Front**: Insert an element at the front.
     - **Insert Rear**: Insert an element at the rear.
     - **Delete Front**: Remove an element from the front.
     - **Delete Rear**: Remove an element from the rear.

Here's the additional information for **Linear Queue** and **Priority Queue**:

3. Linear Queue:

- A **linear queue** is a basic queue structure where elements are added at the **rear** and removed from the **front** in a linear manner.
- Once the queue is full, no new elements can be added until some elements are dequeued, even if there is space in the array due to the front elements being dequeued.

Operations:

- **Enqueue**: Add elements to the rear of the queue.
- **Dequeue**: Remove elements from the front of the queue.
- **IsEmpty**: Checks if the queue is empty (front == rear).
- **IsFull**: Checks if the queue is full (for array-based implementation).

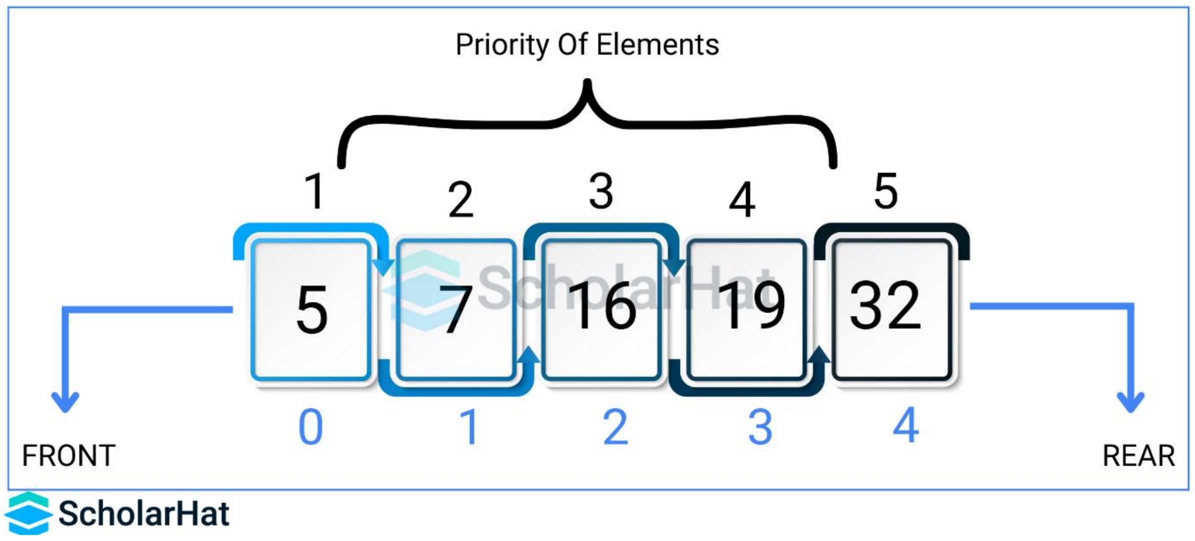
Priority Queue:

A **priority queue** is a type of queue where each element is associated with a priority. Elements are dequeued based on their priority rather than their order of insertion. The highest priority elements are dequeued first.

- **Max-priority queue**: The element with the highest priority is dequeued first.
- **Min-priority queue**: The element with the lowest priority is dequeued first.

Applications:

- **Job scheduling** in operating systems.
- **Dijkstra's shortest path algorithm**.
- **Simulation systems**.



Applications of Queues:

1. **CPU Scheduling**: Managing the order of processes in the CPU.
2. **Job Scheduling**: Managing tasks in a queue for execution in order of priority.
3. **Breadth-First Search (BFS)**: Queue is used for exploring nodes in BFS of a graph.
4. **Handling Requests in Web Servers**: Requests are handled in a FIFO manner.
5. **Asynchronous Data Transfer**: Queues are used in data buffers for transferring data asynchronously (e.g., in printers, I/O devices).
6. **Simulation Systems**: Queues are used to simulate waiting lines or processes in various systems.

Array Representation of Queue (LINEAR QUEUE):

Enqueue Algorithm:

1. **Start**.
2. Check if the queue is **full**: If rear == size - 1 (no space to insert).
   - If yes, print "Overflow" and **exit**.
3. Increment rear using rear = rear + 1.
4. Insert the element at position rear.

5. **Exit**.

2. Dequeue Algorithm:

1. **Start**.
2. Check if the queue is **empty**: If front == rear (no elements).
    - If yes, print "Underflow" and **exit**.
3. Remove the element from position front.
4. Increment front using front = front + 1.
5. **Exit**.

3. IsEmpty Algorithm:

1. **Start**.
2. Check if the queue is **empty**: If front == rear.
    - If yes, print "Queue is Empty".
3. **Exit**.

4. IsFull Algorithm:

1. **Start**.
2. Check if the queue is **full**: If rear == size - 1.
    - If yes, print "Queue is Full".
3. **Exit**.

Array Representation of Queue (CIRCULAR QUEUE):

Enqueue Algorithm:

1. **Start**.
2. Check if the queue is **full**: If (rear + 1) % size == front.
    - If yes, print "Overflow" and **exit**.
3. Increment rear using rear = (rear + 1) % size.
4. Insert the element at position rear.
5. **Exit**.

Dequeue Algorithm:

1. **Start**.
2. Check if the queue is **empty**: If front == rear.
    - If yes, print "Underflow" and **exit**.
3. Remove the element from position front.

4. Increment front using front = (front + 1) % size.
5. **Exit**.

IsFull Algorithm:

1. **Start**.
2. Check if the queue is **full**:
    - If (rear + 1) % size == front, print "Queue is Full".
3. **Exit**.

IsEmpty Algorithm:

1. **Start**.
2. Check if the queue is **empty**:
    - If front == rear, print "Queue is Empty".
3. **Exit**.

Linked List Representation of Queue:

Enqueue Algorithm:

1. **Start**.
2. Create a new node newNode.
3. If the queue is **empty** (front == NULL):
    - Set both front and rear to newNode.
4. Otherwise, link the rear node to newNode and update the rear pointer.
5. **Exit**.

Dequeue Algorithm:

1. **Start**.
2. Check if the queue is **empty**: If front == NULL.
    - If yes, print "Underflow" and **exit**.
3. Remove the element from the front node.
4. If the queue becomes empty (i.e., front == NULL), set rear = NULL.
5. Update the front pointer to the next node.
6. Exit.

IsFull Algorithm:

1. **Start**.
2. In the case of a linked list, the queue is **never full** unless memory is exhausted. Hence, no specific check is required.
3. **Exit**.

IsEmpty Algorithm:

1. **Start**.
2. Check if the queue is **empty**:
    - If front == NULL, print "Queue is Empty".
3. **Exit**.

Qn. What are the Drawback of Linear Queue adn How it can be OverCome?

Drawbacks of Linear Queue:

1. **WSastage of Space:**
    - After dequeuing, the front pointer doesn't move, causing space at the front to remain unused.
    - Even if the queue has space at the front, it cannot be utilized.
2. **Overflow despite Available Space:**
    - When the rear pointer reaches the last position, the queue may appear full, even though there might be empty spaces at the front.
    - This leads to inefficient usage of the array.

How to Overcome:

1. **Circular Queue:**
    - Both front and rear pointers move circularly, utilizing all available space.
    - The queue can wrap around when it reaches the end, using space at the front.
2. **Linked List-based Queue:**
    - No fixed size like an array, so space wastage and overflow due to limited size are avoided.
    - The queue dynamically grows, efficiently using memory as needed.