A **Queue** is a linear data structure that follows the **First-In-First-Out (FIFO)** principle. This means that the first element added to the queue will be the first one to be removed, similar to a line of people waiting for a service. In a queue, elements are added at the **rear** (back) and removed from the **front**.

Key Operations

1. **Enqueue:** Adding an element to the rear of the queue.
2. **Dequeue:** Removing an element from the front of the queue.
3. **Peek/Front:** Getting the front element of the queue without removing it.
4. **isEmpty:** Checking if the queue is empty.

Types of Queues

1. **Simple Queue:** The basic type where elements are added to the rear and removed from the front.
2. **Circular Queue:** The rear of the queue wraps around to the front when it reaches the end, making efficient use of memory.
3. **Priority Queue:** Elements are removed based on priority rather than the order of insertion.
4. **Double-Ended Queue (Deque):** Elements can be added or removed from both the front and rear.

Applications of Queue

- **CPU scheduling** in operating systems.
- **Breadth-First Search (BFS)** in graph traversal.
- **Print queue** for managing print jobs in printers.
- **Handling requests** in web servers.

Simple Queue

The **Simple Queue**, also known as a **Linear Queue**, is the basic form of a queue data structure. In this type, elements are inserted at one end, called the **rear**, and removed from the other end, called the **front**. It strictly follows the **First-In-First-Out (FIFO)** principle, meaning the first element added is the first one to be removed.

Key Characteristics

1. **Insertion at Rear:** New elements are always added at the rear end of the queue.
2. **Deletion from Front:** Elements are removed from the front end of the queue.
3. **FIFO Behavior:** The order in which elements enter the queue is the same order in which they are removed.

Operations in a Simple Queue

1. **Enqueue (Insertion):** Adds an element to the rear of the queue. If the queue is full, this operation cannot be performed (known as "queue overflow").
2. **Dequeue (Deletion):** Removes an element from the front of the queue. If the queue is empty, this operation cannot be performed (known as "queue underflow").
3. **Peek/Front:** Returns the front element without removing it.
4. **isEmpty:** Checks whether the queue is empty.

Limitations

- **Wasted Space:** In a simple queue implemented using an array, after several dequeue operations, the front portion of the array may become unusable, leading to wasted memory.

Example

Imagine a line of customers waiting to be served at a bank:

- The first customer in line (front) is served first (dequeued).
- New customers join the line at the end (rear).

Array Representation

- An array can be used to implement a simple queue, where the front and rear pointers indicate where elements are added and removed.
- When the queue is empty, both front and rear point to -1 or null.

Linked List Representation

- A linked list can also be used, where each node represents an element in the queue, and pointers link each node in the correct order.

The simple queue's straightforward nature makes it suitable for tasks where order needs to be maintained, such as scheduling tasks, managing resources, or simulating real-life scenarios like queues in a supermarket.

Algorithm to insert in Array representation of Linear Queue

```
Step 1:  IF REAR = MAX-1
             Write OVERFLOW
             Goto step 4
             [END OF IF]
Step 2:  IF FRONT = -1 and REAR = -1
             SET FRONT = REAR =
             ELSE
             SET REAR = REAR + 1
             [END OF IF]
Step 3:  SET QUEUE[REAR] = NUM
Step 4:  EXIT
```

Algorithm to delete in Array representation of Linear Queue

```
Step 1:  IF FRONT = -1 OR FRONT > REAR
             Write UNDERFLOW
             ELSE
             SET FRONT = FRONT + 1
             [END OF IF]
Step 2:  EXIT
```

Algorithm to insert in Linked List representation of Linear Queue

```
Step 1:  Allocate memory for the new node and name it as PTR
Step 2:  SET PTR→DATA = VAL
Step 3:  IF FRONT = NULL
             SET FRONT = REAR = PTR
             SET FRONT→ NEXT = REAR →NEXT = NULL
             ELSE
             SET REAR→ NEXT = PTR
             SET REAR = PTR
             SET REAR →NEXT = NULL
             [END OF IF]
Step 4:  END
```

Algorithm to delete in Linked List representation of Linear Queue

```
Step 1:  IF FRONT = NULL
             Write Underflow
             Go to Step 5
             [END OF IF]
Step 2:  SET PTR = FRONT
Step 3: SET FRONT = FRONT → NEXT
Step 4: FREE PTR
Step 5: END
```

CIRCULAR QUEUE

The **Circular Queue** is a more efficient version of the linear queue that addresses the problem of wasted space in the array implementation of a simple queue. In a circular queue, the last position is connected back to the first position to form a circle, which allows for efficient utilization of the storage space.

Key Characteristics

1. **Wrap-around Movement:** When the queue's rear reaches the last position of the array, it wraps around to the first position if there is space available, forming a circular structure.
2. **Front and Rear Pointers:** The front pointer indicates where elements are removed, and the rear pointer indicates where elements are added.
3. **Full and Empty Conditions:**
    - The queue is **full** if (rear + 1) % size == front.
    - The queue is **empty** if front == -1.

Advantages

- **Efficient Space Utilization:** Circular queues make full use of the array space by reusing positions that become free after dequeue operations.
- **Avoids Wasted Space:** Unlike simple queues, where unused space may accumulate at the beginning of the array, circular queues wrap around to use all available memory.

Operations in Circular Queue

1. Insertion (Enqueue) Algorithm

1. **Check if the queue is full**: If (rear + 1) % size == front, the queue is full (overflow condition).
2. **Increment the rear pointer**: rear = (rear + 1) % size.
3. **Add the new element at the rear**: queue[rear] = element.
4. **If the queue was initially empty**, set the front pointer: If front == -1, set front = 0.

2. Deletion (Dequeue) Algorithm

1. **Check if the queue is empty**: If front == -1, the queue is empty (underflow condition).
2. **Remove the element from the front**: element = queue[front].
3. **Increment the front pointer**: front = (front + 1) % size.
4. **If the queue becomes empty after the operation**, reset both pointers: If front == rear + 1, set front = -1 and rear = -1.

Applications of Circular Queue

- **Buffer Management:** Used in circular buffers for streaming data (audio, video).
- **CPU Scheduling:** Round-robin scheduling in operating systems.
- **Traffic Management:** Circular queues can model processes that loop through a fixed set of states.

Algorithm to insert in Circular Queue

```
Step 1:  IF FRONT = and Rear = MAX - 1
         Write OVERFLOW
         Goto Step 4
         [End of IF]
Step 2:  IF FRONT = -1 and REAR = -1
         SET FRONT = REAR = 0
         ELSE IF REAR = MAX - 1 and FRONT != 0
         SET REAR = 0
         ELSE
         SET REAR = REAR + 1
         [END OF IF]
Step 3:  SET QUEUE[REAR] = VAL
Step 4:  EXIT
```

Algorithm to delete in Circular Queue

```
Step 1:  IF FRONT = -1
         Write UNDERFLOW
         Goto Step 4
         [END of IF]
Step 2:  SET VAL = QUEUE[FRONT]
Step 3:   IF FRONT = REAR
                 SET FRONT = REAR = -1
         ELSE
                 IF FRONT = MAX -1
                         SET FRONT = 0
                 ELSE
                         SET FRONT = FRONT + 1
                 [END of IF]
         [END OF IF]
Step 4:  EXIT
```

PRIORITY QUEUES

The **Priority Queue** is a type of queue where each element is associated with a **priority**, and elements are dequeued based on their priority rather than the order in which they were enqueued. In a priority queue, elements with **higher priority** are dequeued before elements with **lower priority**. If two elements have the same priority, they follow the **First-In-First-Out (FIFO)** order.

Key Characteristics

1. **Priority-Based Removal:** Elements are dequeued based on their priority. Higher-priority elements are removed before lower-priority ones.
2. **Dynamic Priorities:** Priorities can be assigned dynamically when elements are added, and the queue adjusts based on these priorities.
3. **FIFO for Equal Priority:** If two elements have the same priority, they are processed in the order they were enqueued.

Types of Priority Queues

1. **Ascending Priority Queue:** The element with the **lowest priority** is dequeued first.
2. **Descending Priority Queue:** The element with the **highest priority** is dequeued first (more common).

Operations in a Priority Queue

1. **Insertion (Enqueue):** Adding an element with a specified priority.
2. **Deletion (Dequeue):** Removing the element with the highest (or lowest, depending on the type) priority.
3. **Peek:** Accessing the element with the highest priority without removing it.
4. **isEmpty:** Checking if the queue is empty.

Implementations of Priority Queue

1. **Array-Based Implementation:**
    - Elements are stored in an array along with their priorities.
    - When an element is dequeued, a linear search is performed to find the element with the highest priority.
2. **Linked List Implementation:**
    - Nodes are stored in a linked list in a sorted order based on priority.
    - Insertion involves finding the correct position based on priority.
3. **Heap-Based Implementation:**
    - The **binary heap** (min-heap or max-heap) is commonly used because it allows for efficient insertion and deletion operations.
    - A **min-heap** allows the smallest element (highest priority) to be at the root, while a **max-heap** allows the largest element to be at the root.

Algorithms for Priority Queue Operations

1. Array-Based Implementation

Insertion (Enqueue) Algorithm:

1. Add the new element at the end of the array.
2. Assign the priority to the element.

Deletion (Dequeue) Algorithm:

1. Find the element with the highest priority by scanning the array.
2. Remove the element and shift the remaining elements.

2. Linked List Implementation

Insertion (Enqueue) Algorithm:

1. Create a new node with the given priority.
2. Traverse the list to find the correct position based on priority.
3. Insert the node at the appropriate position.

Deletion (Dequeue) Algorithm:

1. Remove the node at the front (highest priority).

Applications of Priority Queue

- **CPU Scheduling:** Managing processes with different priorities.
- **Dijkstra's Algorithm:** For finding the shortest path in graphs.
- **Huffman Coding:** Building optimal prefix codes.
- **Simulation Systems:** Handling events that occur with varying levels of importance.

DOUBLE ENDED QUEUE

The next type of queue is the **Double-Ended Queue**, commonly known as a **Deque** (pronounced "deck"). In a deque, elements can be added or removed from both the front and the rear, giving it more flexibility than a simple queue.

Key Characteristics

1. **Bidirectional Operations:** Elements can be enqueued (added) or dequeued (removed) from both the front and the rear.
2. **Generalization of Queue and Stack:** A deque can act as both a queue (FIFO behavior) and a stack (LIFO behavior) depending on how elements are added or removed.
3. **Not Restricted to FIFO:** Unlike a simple queue, it does not strictly follow the FIFO order. Instead, elements can be inserted and deleted at either end.

Types of Deques

1. **Input-Restricted Deque:** Insertion is allowed only at one end (either front or rear), but deletion can occur from both ends.
2. **Output-Restricted Deque:** Deletion is allowed only at one end, but insertion can occur from both ends.

Operations in a Deque

1. **Insertion at Front:** Adding an element to the front of the deque.
2. **Insertion at Rear:** Adding an element to the rear of the deque.
3. **Deletion from Front:** Removing an element from the front of the deque.
4. **Deletion from Rear:** Removing an element from the rear of the deque.
5. **Peek/Front:** Accessing the front element without removing it.
6. **Peek/Rear:** Accessing the rear element without removing it.
7. **isEmpty:** Checking if the deque is empty.
8. **isFull:** Checking if the deque is full (only for fixed-size implementations).

Array Representation of a Deque

In an array-based deque, the front and rear pointers are used to keep track of where elements should be added or removed. The circular nature of the array helps in efficiently managing the wrap-around behavior.

1. Insertion at Front Algorithm

1. **Check if the deque is full**: If (front == 0 && rear == size - 1) || (front == rear + 1), then the deque is full.
2. **If the deque is empty**, set both front and rear to 0.
3. **If the front is at the beginning of the array**, set front = size - 1.
4. **Otherwise**, decrement the front pointer: front = front - 1.
5. **Insert the element at deque[front]**.

2. Insertion at Rear Algorithm

1. **Check if the deque is full**: If (front == 0 && rear == size - 1) || (front == rear + 1), then the deque is full.
2. **If the deque is empty**, set both front and rear to 0.
3. **If the rear is at the end of the array**, set rear = 0.
4. **Otherwise**, increment the rear pointer: rear = rear + 1.
5. **Insert the element at deque[rear]**.

3. Deletion from Front Algorithm

1. **Check if the deque is empty**: If front == -1, the deque is empty (underflow condition).
2. **If there is only one element**, set both front and rear to -1.
3. **If the front is at the end of the array**, set front = 0.
4. **Otherwise**, increment the front pointer: front = front + 1.

4. Deletion from Rear Algorithm

1. **Check if the deque is empty**: If front == -1, the deque is empty (underflow condition).
2. **If there is only one element**, set both front and rear to -1.
3. **If the rear is at the beginning of the array**, set rear = size - 1.
4. **Otherwise**, decrement the rear pointer: rear = rear - 1.

Linked List Representation of a Deque

A deque can also be implemented using a doubly linked list, where each node has pointers to both the previous and the next node. This allows for efficient insertion and deletion at both ends.

Applications of Deques

- **Palindromic Checking:** Can be used to check whether a string is a palindrome.
- **Undo Operations in Software:** Maintaining a history of operations.
- **Scheduling Algorithms:** Deques can be used in scheduling algorithms that require both FIFO and LIFO order.