QUEUE :

Features of Queue Data Structure

A **queue** is a **linear data structure** that follows the **FIFO (First In, First Out) principle**, meaning the **element added first is removed first**. It is widely used in real-world applications like **task scheduling, process management, and breadth-first search (BFS) algorithms**.

🔷 Key Features of Queue

1️⃣ Follows FIFO (First In, First Out) Order

- The element that is **inserted first** is the **first one** to be removed.
- **Example**: A queue of people waiting in line for a ticket.
- **Operations:**
  - **Enqueue (Insertion)**: Adds an element at the **rear**.
  - **Dequeue (Removal)**: Removes an element from the **front**.

2️⃣ Two Main Pointers

- **Front**: Points to the first element (element to be dequeued).
- **Rear**: Points to the last element (where a new element is enqueued).

3️⃣ Basic Operations

- `enqueue(x)`: Adds an element `x` at the rear.
- `dequeue()`: Removes an element from the front.
- `front()`: Returns the front element without removing it.
- `rear()`: Returns the last element without removing it.
- `is_empty()`: Checks if the queue is empty.
- `size()`: Returns the number of elements in the queue.

4️⃣ Different Types of Queues

- **Simple Queue** (Basic FIFO queue).
- **Circular Queue** (Efficient memory usage, wraps around when the end is reached).
- **Priority Queue** (Elements are dequeued based on priority, not FIFO).
- **Double-Ended Queue (Deque)** (Insertion and deletion from both ends).

5️⃣ Time Complexity of Queue Operations

- **Enqueue (** `O(1)` **)** – Adding an element at the rear is constant time.
- **Dequeue (** `O(1)` **)** – Removing an element from the front is constant time.
- **Front (** `O(1)` **)** – Accessing the front element is constant time.

- **IsEmpty (`O(1)`)** – Checking if the queue is empty is constant time.
- **List-Based Queue (Python `list.pop(0)`)** – `O(n)` (slow due to shifting elements).
- **Deque-Based Queue (Python `collections.deque`)** – `O(1)` (fast and efficient).

6️⃣ **Used in Various Applications**

- **CPU Scheduling & Process Management** (Round Robin Scheduling).
- **Breadth-First Search (BFS)** (Graph and Tree Traversal).
- **Printer Queue** (Printing jobs are processed in order).
- **Task Scheduling** (Job processing in order of arrival).
- **Call Center System** (Customer service calls are answered in the order they arrive).

Features of Queue Data Structure

A **queue** is a **linear data structure** that follows the **FIFO (First In, First Out) principle**, meaning the **element added first is removed first**. It is widely used in real-world applications like **task scheduling, process management, and breadth-first search (BFS) algorithms**.

🔷 Key Features of Queue

1️⃣ **Follows FIFO (First In, First Out) Order**

- The element that is **inserted first** is the **first one** to be removed.
- **Example**: A queue of people waiting in line for a ticket.
- **Operations:**
  - **Enqueue (Insertion)**: Adds an element at the **rear**.
  - **Dequeue (Removal)**: Removes an element from the **front**.

2️⃣ **Two Main Pointers**

- **Front**: Points to the first element (element to be dequeued).
- **Rear**: Points to the last element (where a new element is enqueued).

3️⃣ **Basic Operations**

- `enqueue(x)`: Adds an element `x` at the rear.
- `dequeue()`: Removes an element from the front.
- `front()`: Returns the front element without removing it.
- `rear()`: Returns the last element without removing it.
- `is_empty()`: Checks if the queue is empty.
- `size()`: Returns the number of elements in the queue.

4️⃣ Different Types of Queues

- **Simple Queue** (Basic FIFO queue).
- **Circular Queue** (Efficient memory usage, wraps around when the end is reached).
- **Priority Queue** (Elements are dequeued based on priority, not FIFO).
- **Double-Ended Queue (Deque)** (Insertion and deletion from both ends).

5️⃣ Time Complexity of Queue Operations

- **Enqueue (** `O(1)` **)** – Adding an element at the rear is constant time.
- **Dequeue (** `O(1)` **)** – Removing an element from the front is constant time.
- **Front (** `O(1)` **)** – Accessing the front element is constant time.
- **IsEmpty (** `O(1)` **)** – Checking if the queue is empty is constant time.
- **List-Based Queue (Python** `list.pop(0)` **)** – `O(n)` (slow due to shifting elements).
- **Deque-Based Queue (Python** `collections.deque` **)** – `O(1)` (fast and efficient).

6️⃣ Used in Various Applications

- **CPU Scheduling & Process Management** (Round Robin Scheduling).
- **Breadth-First Search (BFS)** (Graph and Tree Traversal).
- **Printer Queue** (Printing jobs are processed in order).
- **Task Scheduling** (Job processing in order of arrival).
- **Call Center System** (Customer service calls are answered in the order they arrive).

Operations on Queue

A queue supports the following operations:

1. **Enqueue (Insertion)** – Adds an element to the rear (end) of the queue.
2. **Dequeue (Deletion)** – Removes an element from the front (beginning) of the queue.
3. **Front (Peek)** – Returns the front element of the queue without removing it.
4. **isEmpty** – Checks if the queue is empty.
5. **Size** – Returns the number of elements in the queue.

1️⃣ Queue Implementations in Python

Implementation	Data Structure Used	enqueue (O(1))	dequeue (O(1))	front (O(1))	is_empty (O(1))	size (O(1))
List (Using `append()` and `pop(0)`)	List (Dynamic Array)	✅ `O(1)`	❌ `O(n)` (Shifting elements)	✅ `O(1)`	✅ `O(1)`	✅ `O(1)`
`collections.deque` (Double-ended queue)	Doubly Linked List	✅ `O(1)`	✅ `O(1)`	✅ `O(1)`	✅ `O(1)`	✅ `O(1)`

Queue from `queue.Queue`	**Thread-Safe Queue**	✅ O(1)	✅ O(1)	✅ O(1)	✅ O(1)	✅ O(1)
Circular Queue (Array-based)	**Fixed-size Array**	✅ O(1)	✅ O(1)	✅ O(1)	✅ O(1)	✅ O(1)

Node Class for Queue:

```
python
CopyEdit
class Node:
    def __init__(self, data):
        self.data = data  # Store the data
        self.next = None  # The next node, initially set to None
```

Queue Class with All Operations:

```
class Queue:
    def __init__(self):
        self.front = None  # Pointer to the front of the queue
        self.rear = None   # Pointer to the rear of the queue

    def is_empty(self):
        return self.front is None  # If front is None, the queue is empty

    def enqueue(self, value):
        new_node = Node(value)  # Create a new node with the value
        if self.rear is None:  # If the queue is empty
            self.front = self.rear = new_node  # Both front and rear point to the new node
            return
        self.rear.next = new_node  # Add the new node after the rear
        self.rear = new_node  # Update the rear to the new node

    def dequeue(self):
        if self.is_empty():
            print("Queue is empty, underflow condition!")
            return None  # Return None if the queue is empty

        dequeued_value = self.front.data  # Get the data of the front node
        self.front = self.front.next  # Move the front pointer to the next node

        # If the queue becomes empty after the dequeue, update the rear to None
        if self.front is None:
            self.rear = None
```

```
        return dequeued_value  # Return the dequeued value

    def peek(self):
        if self.is_empty():
            print("Queue is empty!")
            return None  # If the queue is empty, return None
        return self.front.data  # Return the data of the front node

    def size(self):
        size = 0
        current = self.front
        while current:
            size += 1  # Count the number of nodes in the queue
            current = current.next
        return size  # Return the size of the queue

    def display(self):
        result = []  # To store queue elements for display
        current = self.front  # Start from the front
        while current:
            result.append(str(current.data))  # Append data of current node
            current = current.next  # Move to the next node
        print("→".join(result))  # Join the elements with "→" for visual clarit
```

Code for Enqueuing (Adding an Element to the Queue):

```
python
CopyEdit
# Initialize queue
queue = Queue()

# Enqueue elements
queue.enqueue(10)
queue.enqueue(20)
queue.enqueue(30)

# Display the queue
queue.display()  # Expected Output: 10→20→30
```

Code for Dequeuing (Removing an Element from the Queue):

```
python
CopyEdit
# Initialize queue
queue = Queue()
```

```
# Enqueue elements
queue.enqueue(10)
queue.enqueue(20)
queue.enqueue(30)

# Dequeue an element
print(queue.dequeue())  # Expected Output: 10
queue.display()  # Expected Output: 20→30
```

Code for Peeking at the Front Element:

```
python
CopyEdit
# Initialize queue
queue = Queue()

# Enqueue elements
queue.enqueue(10)
queue.enqueue(20)
queue.enqueue(30)

# Peek at the front element
print(queue.peek())  # Expected Output: 10
```

Code for Checking if the Queue is Empty:

```
python
CopyEdit
# Initialize queue
queue = Queue()

# Check if queue is empty
print(queue.is_empty())  # Expected Output: True

# Enqueue an element
queue.enqueue(10)

# Check if queue is empty again
print(queue.is_empty())  # Expected Output: False
```

Code for Getting the Size of the Queue:

```
python
CopyEdit
# Initialize queue
queue = Queue()

# Enqueue elements
queue.enqueue(10)
queue.enqueue(20)
queue.enqueue(30)

# Get the size of the queue
print(queue.size())  # Expected Output: 3
```

Code for Displaying the Queue:

```
python
CopyEdit
# Initialize queue
queue = Queue()

# Enqueue elements
queue.enqueue(10)
queue.enqueue(20)
queue.enqueue(30)

# Display the queue
queue.display()  # Expected Output: 10→20→30
```

Code for Dequeuing All Elements from the Queue:

```
python
CopyEdit
# Initialize queue
queue = Queue()

# Enqueue elements
queue.enqueue(10)
queue.enqueue(20)
queue.enqueue(30)

# Dequeue all elements
print(queue.dequeue())  # Expected Output: 10
print(queue.dequeue())  # Expected Output: 20
print(queue.dequeue())  # Expected Output: 30
```

```
print(queue.dequeue())  # Expected Output: Queue is empty, underflow condition!
```

Code for Checking the Queue After Dequeuing All Elements:

```
python
CopyEdit
# Initialize queue
queue = Queue()

# Enqueue elements
queue.enqueue(10)
queue.enqueue(20)
queue.enqueue(30)

# Dequeue all elements
queue.dequeue()
queue.dequeue()
queue.dequeue()

# Display the queue after dequeuing all elements
queue.display()  # Expected Output: (empty queue, no output)
```

Explanation of the Flow:

- **Enqueue Operation**: Adds a new node to the rear of the queue. If the queue is empty, both `front` and `rear` will point to the new node.
- **Dequeue Operation**: Removes a node from the front of the queue and returns its value. The front pointer is updated to the next node. If the queue becomes empty after this operation, the rear pointer is also set to `None`.
- **Peek Operation**: Returns the data of the front node without removing it from the queue.
- **Is Empty Operation**: Checks if the queue is empty by checking if the `front` pointer is `None`.
- **Size Operation**: Counts and returns the number of nodes in the queue.
- **Display Operation**: Prints the elements of the queue from front to rear.

Queue:

Queue - A Detailed Explanation

A **queue** is a fundamental data structure in computer science that works based on the **First In First Out (FIFO)** principle. This means that the element which is inserted first will be the first one to be removed.

Key Concepts of Queue

1. **FIFO Principle**:
   - **First In First Out**. The first element that is added to the queue is the first one to be removed.
   - Similar to a queue in real life, like a line at a ticket counter, where the first person to stand in line is the first one to get served.
2. **Queue Operations**:
   - **Enqueue**: Add an element to the back (rear) of the queue.
   - **Dequeue**: Remove an element from the front of the queue.
   - **Peek/Front**: View the front element without removing it.
   - **Is Empty**: Check if the queue has any elements or is empty.
   - **Size**: Get the number of elements in the queue.
   - **Display**: Show all elements in the queue in the order they are arranged.

Queue Terminology

- **Front**: The front of the queue is the position where elements are removed (dequeued). This is where the first element resides.
- **Rear**: The rear of the queue is where elements are inserted (enqueued). The most recently added element resides here.
- **Empty Queue**: A queue with no elements. Both `front` and `rear` will be `None`.
- **Full Queue**: In some implementations, especially in bounded queues, when the queue is full and cannot hold any more elements. However, in dynamic queues (like the one we implemented), this is not a concern as the queue can grow indefinitely.

Types of Queues

1. **Simple Queue**:
   - The regular FIFO queue where elements are inserted at the rear and removed from the front.
   - It works just like a line at a bank or supermarket.
2. **Circular Queue**:
   - A variation of the queue in which the last position is connected back to the first position.
   - This helps in utilizing the available space efficiently (no wasted space between the front and rear).
3. **Priority Queue**:
   - In a priority queue, elements are dequeued in order of their priority rather than the order in which they were enqueued.
   - Higher priority elements are dequeued before lower priority ones, regardless of the order of arrival.

4. **Double-Ended Queue (Deque)**:
    - A deque allows insertion and deletion of elements from both ends — front and rear.
    - It can function as both a queue and a stack.

Operations on a Queue

1. **Enqueue Operation**:
    - **Definition**: Adding an element to the queue.
    - **Procedure**:
        - Create a new node with the value to be added.
        - If the queue is empty, both `front` and `rear` pointers point to the new node.
        - If the queue is not empty, add the new node to the rear and update the `rear` pointer.
    - **Time Complexity**: O(1), because inserting an element at the rear is a constant-time operation.
2. **Dequeue Operation**:
    - **Definition**: Removing an element from the front of the queue.
    - **Procedure**:
        - If the queue is empty, return an underflow condition.
        - Retrieve the data from the front node.
        - Update the `front` pointer to the next node in the queue.
        - If the queue becomes empty after the dequeue, set the `rear` pointer to `None`.
    - **Time Complexity**: O(1), because removing an element from the front is a constant-time operation.
3. **Peek/Front Operation**:
    - **Definition**: Viewing the front element of the queue without removing it.
    - **Procedure**:
        - If the queue is empty, return an empty condition.
        - Return the data of the node at the front.
    - **Time Complexity**: O(1), since accessing the front node takes constant time.
4. **Is Empty Operation**:
    - **Definition**: Checking whether the queue is empty or not.
    - **Procedure**:
        - If the `front` pointer is `None`, the queue is empty.
        - Return `True` if empty, otherwise `False`.
    - **Time Complexity**: O(1), as it only involves checking whether the `front` pointer is `None`.
5. **Size Operation**:
    - **Definition**: Get the total number of elements in the queue.
    - **Procedure**:

    - Traverse the queue from the front node to the rear node.
    - Count the number of nodes visited.
  - **Time Complexity**: O(n), where n is the number of elements in the queue. In a linked list-based queue, this operation requires traversing the entire queue.
6. **Display Operation**:
  - **Definition**: Display all elements of the queue in the order they were enqueued.
  - **Procedure**:
    - Traverse the queue starting from the `front` node.
    - Print each node's data until reaching the end of the queue.
  - **Time Complexity**: O(n), since it involves traversing through all the elements in the queue.

Queue Implementations

A queue can be implemented using:

1. **Array-based Implementation**:
  - Use an array (or list in Python) to store the elements.
  - The **rear** pointer tracks where the new elements will be added, and the **front** pointer tracks where elements are removed.

  Drawbacks:
  - Fixed size (if using arrays).
  - The front of the queue needs to be adjusted after each dequeue operation, which might cause inefficiency (shifting elements).
2. **Linked List-based Implementation**:
  - Use a linked list to implement the queue.
  - Each node points to the next node, and the **front** and **rear** pointers track the beginning and end of the queue.

  Advantages:
  - Dynamic size (grows and shrinks as needed).
  - No need to shift elements after dequeueing (efficient).

  Example of Linked List-based Queue:
  The code implementation I provided above demonstrates a **queue using linked lists**. In this approach, both the `front` and `rear` pointers track the queue efficiently.

Applications of Queues

Queues are used in various real-life and computer science applications. Some common uses include:

1. **Job Scheduling**:
  - In operating systems, processes or tasks are scheduled in a queue, with the CPU processing one task at a time in a FIFO manner.
2. **Handling Requests in Servers**:

- Requests from clients are placed in a queue and processed in the order they are received.

3. **Data Buffers**:
   - In networking or streaming, data is buffered in a queue before being transmitted or processed.
4. **Breadth-First Search (BFS)**:
   - In graph theory, BFS uses a queue to explore nodes level by level, starting from the source node.
5. **Simulations**:
   - In discrete event simulations, entities such as customers in a bank or cars at a toll booth are modeled as elements in a queue.

List Implementation:

Queue Using List (Array-based Implementation)

```
python
CopyEdit
class Queue:
    def __init__(self):
        self.queue = []  # Initialize an empty list to represent the queue

    # Enqueue operation: Adds an element to the rear of the queue
    def enqueue(self, item):
        self.queue.append(item)

    # Dequeue operation: Removes an element from the front of the queue
    def dequeue(self):
        if self.is_empty():
            print("Queue is empty. Underflow condition.")
            return None
        return self.queue.pop(0)

    # Peek operation: Views the front element of the queue without removing it
    def peek(self):
        if self.is_empty():
            print("Queue is empty.")
            return None
        return self.queue[0]

    # Is Empty operation: Checks if the queue is empty
    def is_empty(self):
        return len(self.queue) == 0

    # Size operation: Returns the number of elements in the queue
    def size(self):
        return len(self.queue)
```

```
    # Display operation: Displays all elements in the queue
    def display(self):
        if self.is_empty():
            print("Queue is empty.")
            return
        print("Queue elements:", " -> ".join(map(str, self.queue)))


# Create a Queue object
q = Queue()

# Enqueue some elements
q.enqueue(10)
q.enqueue(20)
q.enqueue(30)
q.enqueue(40)

# Display the queue
q.display()

# Peek at the front element
print("Front element:", q.peek())

# Dequeue an element and display the queue
print("Dequeued element:", q.dequeue())
q.display()

# Display the size of the queue
print("Queue size:", q.size())

# Check if the queue is empty
print("Is the queue empty?", q.is_empty())

# Dequeue all elements and display the queue
print("Dequeued element:", q.dequeue())
print("Dequeued element:", q.dequeue())
print("Dequeued element:", q.dequeue())

q.display()  # After dequeuing all elements
```

Explanation of Code:

1. **Queue Initialization (`__init__` method)**:
    - Initializes an empty list (`self.queue`) to store the elements of the queue.
2. **Enqueue Operation (`enqueue` method)**:

- Adds an element to the end (rear) of the queue using `append()`. This operation takes constant time, O(1).

3. **Dequeue Operation (`dequeue` method)**:
   - Removes an element from the front of the queue using `pop(0)`. Since removing the first element requires shifting all other elements, the time complexity for this operation is O(n).
4. **Peek Operation (`peek` method)**:
   - Views the front element of the queue without removing it. If the queue is empty, it returns `None`.
5. **Is Empty Operation (`is_empty` method)**:
   - Checks if the queue is empty by evaluating the length of the list (`len(self.queue)`). If the list is empty, it returns `True`; otherwise, `False`.
6. **Size Operation (`size` method)**:
   - Returns the number of elements currently in the queue by getting the length of the list using `len(self.queue)`.
7. **Display Operation (`display` method)**:
   - Displays all the elements currently in the queue. If the queue is not empty, it prints each element, separated by an arrow (`>`), in the order they were added.

Output:

```
plaintext
CopyEdit
Queue elements: 10 → 20 → 30 → 40
Front element: 10
Dequeued element: 10
Queue elements: 20 → 30 → 40
Queue size: 3
Is the queue empty? False
Dequeued element: 20
Dequeued element: 30
Dequeued element: 40
Queue is empty.
```

Explanation of Output:

1. After enqueueing the elements 10, 20, 30, and 40, the queue contains those elements in that order.
2. The `peek()` method shows that the front element is 10.
3. After dequeueing 10, the queue contains 20, 30, and 40.
4. The size of the queue is 3 after one dequeue operation.
5. After dequeuing all elements, the queue is empty.

Advantages of List-based Queue:

- **Simple implementation**: The queue is implemented using Python's built-in list, so no additional data structures are required.

- **Dynamic sizing**: The list can grow or shrink as needed.

Disadvantages of List-based Queue:

- **Inefficient dequeue operation**: The `pop(0)` operation, which removes the front element, has a time complexity of O(n) because all other elements have to be shifted to the left.
- **Memory overhead**: Lists in Python are dynamically resized, which could result in memory inefficiency when the queue size fluctuates.