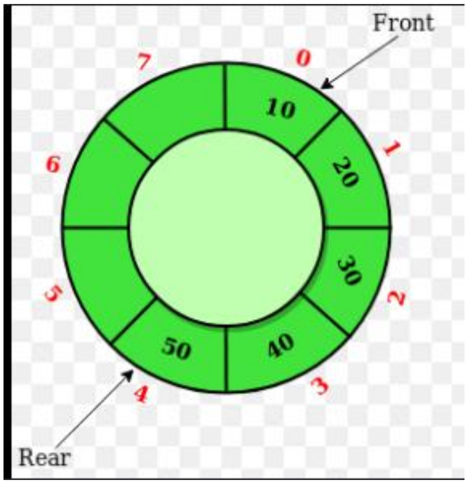
Circular Queue



Key Components

1. **Array**: A fixed-size array to hold the queue elements.
2. **Front Index**: An index to track the front (first) element of the queue.
3. **Rear Index**: An index to track the rear (last) element of the queue.
4. **Size Counter**: A counter to keep track of the number of elements in the queue.
5. **Capacity**: The maximum number of elements the queue can hold.

Operations

1. Initialization

- **Array**: Allocate an array of a fixed size (capacity).
- **Front**: Initialize to 0.
- **Rear**: Initialize to -1.
- **Size**: Initialize to 0.

2. Enqueue (Adding an Element)

- **Check Full**: Before adding, check if the queue is full by comparing the size counter with the capacity.
- **Update Rear**: Increment the rear index by 1. If the rear index exceeds the capacity, wrap it around to the start using modulo arithmetic (rear = (rear + 1) % capacity).
- **Insert Element**: Insert the new element at the updated rear index.
- **Increment Size**: Increase the size counter by 1.

3. Dequeue (Removing an Element)

- **Check Empty**: Before removing, check if the queue is empty by comparing the size counter with 0.
- **Retrieve Element**: Retrieve the element at the front index.
- **Update Front**: Increment the front index by 1. If the front index exceeds the capacity, wrap it around to the start using modulo arithmetic (front = (front + 1) % capacity).
- **Decrement Size**: Decrease the size counter by 1.

4. IsEmpty (Check if the Queue is Empty)

- **Check Size**: Return true if the size counter is 0; otherwise, return false.

5. IsFull (Check if the Queue is Full)

- **Check Size**: Return true if the size counter is equal to the capacity; otherwise, return false.

6. Size (Get the Number of Elements)

- **Return Size**: Simply return the size counter.

Handling Edge Cases

1. **Full Queue**: When the queue is full, attempts to enqueue should raise an error or return a failure status.
2. **Empty Queue**: When the queue is empty, attempts to dequeue should raise an error or return a failure status.
3. **Wrap-around**: Ensure both front and rear indices wrap around correctly using modulo arithmetic to utilize the array space effectively.

Example Scenario

1. **Initialization**:
   - Queue capacity: 5
   - Front index: 0
   - Rear index: -1
   - Size: 0
2. **Enqueue Elements**:
   - Enqueue 1: Rear -> 0, Size -> 1
   - Enqueue 2: Rear -> 1, Size -> 2
   - Enqueue 3: Rear -> 2, Size -> 3
   - Enqueue 4: Rear -> 3, Size -> 4
   - Enqueue 5: Rear -> 4, Size -> 5 (Queue is now full)
3. **Dequeue Elements**:
   - Dequeue: Front -> 1, Size -> 4 (Removed element: 1)
   - Dequeue: Front -> 2, Size -> 3 (Removed element: 2)
4. **Wrap-around**:
   - Enqueue 6: Rear -> 0, Size -> 4 (Wrapped around)
   - Enqueue 7: Rear -> 1, Size -> 5 (Queue is full again, with wrap-around)

Summary

Implementing a circular queue using an array in Java involves managing the front and rear indices and using modulo arithmetic to handle wrap-around behavior. This ensures that the queue utilizes the array space efficiently, maintaining constant time complexity for enqueue and dequeue operations.