To implement the rotate() method in an ArrayQueue class efficiently, we can directly manipulate the front and rear pointers (or indices) without modifying the size or performing two separate operations (dequeue() followed by enqueue()).

Here's how the rotate() method can be implemented:

1. **Rotation Concept**: The idea is to "rotate" the queue by moving the front element to the rear. This can be achieved by updating the front pointer in a circular manner.
2. **Efficiency**: Instead of calling dequeue() (which removes an item and shifts all elements) followed by enqueue() (which adds an item to the rear), we just update the front pointer, making it point to the next element in the queue, effectively moving it to the rear without altering the size of the queue.

**Code Implementation**

public class ArrayQueue<T> {

private T[] data;

private int front; // Index of the front element

private int rear; // Index of the rear element

private int size; // Current size of the queue

private int capacity;

public ArrayQueue(int capacity) {

this.capacity = capacity;

this.data = (T[]) new Object[capacity];

this.front = 0;

this.rear = 0;

this.size = 0;

}

public boolean isEmpty() {

return size == 0;

}

public boolean isFull() {

return size == capacity;

}

public void enqueue(T item) {

if (isFull()) throw new IllegalStateException("Queue is full");

data[rear] = item;

rear = (rear + 1) % capacity;

size++;

}

public T dequeue() {

if (isEmpty()) throw new IllegalStateException("Queue is empty");

T item = data[front];

front = (front + 1) % capacity;

size--;

return item;

}

// Rotate method: rotate the queue by moving the front element to the rear

public void rotate() {

if (isEmpty()) return; // No need to rotate if the queue is empty

front = (front + 1) % capacity; // Move front pointer to the next element

rear = (rear + 1) % capacity; // Move rear pointer to the next position

}

public T peek() {

if (isEmpty()) throw new IllegalStateException("Queue is empty");

return data[front];

}

public int size() {

return size;

}

}

**Explanation:**

* **rotate() method**: The rotate() method adjusts the front and rear pointers:
  + front is updated to point to the next element, which effectively "removes" the front element from the queue.
  + rear is updated to point to the next available slot, effectively adding the former front element to the rear of the queue.
* **Circular Array**: We use the modulo operation (% capacity) to handle the circular nature of the queue, ensuring that the pointers wrap around to the beginning when they reach the end of the array.

**Efficiency:**

* **Time Complexity**: The rotate() method runs in constant time, O(1), because we are only updating the pointers and not shifting any elements or resizing the array.
* **Space Complexity**: The space complexity remains O(n) for the array that holds the queue elements.

This approach is more efficient than performing two separate operations (dequeue() followed by enqueue()), especially when the queue size is large, as it avoids unnecessary shifting of elements.