Topic Introduction

Today, we're diving into a classic but often underestimated topic in coding interviews: **Queue Variations**.

A **queue** is a linear data structure that follows the First-In-First-Out (FIFO) principle. In other words, the first element added is the first one to be removed. The basic operations are `enqueue` (add to back) and `dequeue` (remove from front). Queues are incredibly useful for scenarios like scheduling, order processing, and breadth-first search in graphs.

However, real-world problems often need more than just simple queues. Sometimes you need to access both ends (think of a line at a bank where people can join or leave from either end), or even the middle! This is where **queue variations** come in:

- **Circular Queue**: Uses a fixed-size array in a circular manner. When you reach the end, you wrap around to the beginning.
- **Circular Deque**: Like a double-ended queue, but with a circular, fixed-size buffer.
- **Front-Middle-Back Queue**: Supports adding and removing from the front, middle, or back.

Queues (and their cousins) are tested in interviews because they reveal your grasp of **pointer/index manipulation, efficiency, and edge-case handling**. They also help interviewers see if you can design data structures under constraints.

Quick Example (not from our problems):
Suppose you have a simple queue supporting `enqueue` and `dequeue`. If you use a Python list and call `pop(0)` for dequeue, that operation takes O(n) time, not O(1), because it has to shift all elements. Using collections.deque or managing indices in a circular buffer solves this efficiently.

Let's see how these ideas show up in three very real interview problems.

Problem 1: Design Circular Queue

LeetCode 622: Design Circular Queue

Problem Restated:
Design a fixed-size circular queue. Support these operations:

- `enQueue(value)`: Insert an element into the circular queue. Return `True` if successful, `False` if the queue is full.
- `deQueue()`: Delete an element from the circular queue. Return `True` if successful, `False` if the queue is empty.
- `Front()`: Get the front item. If the queue is empty, return `-1`.
- `Rear()`: Get the last item. If the queue is empty, return `-1`.
- `isEmpty()`: Check if the queue is empty.
- `isFull()`: Check if the queue is full.

Example:

```
q = MyCircularQueue(3)  # Initialize buffer of size 3
q.enQueue(1)            # Returns True
q.enQueue(2)            # Returns True
q.enQueue(3)            # Returns True
q.enQueue(4)            # Returns False (queue is full)
q.Rear()                # Returns 3
```

```
q.isFull()              # Returns True
q.deQueue()             # Returns True
q.enQueue(4)            # Returns True
q.Rear()                # Returns 4
```

Test yourself:

Suppose you have a queue of size 2. What happens if you call enQueue(1), enQueue(2), isFull(), deQueue(), enQueue(3), Rear()?

How to Think About It:

This is about **efficient array management**. If you just use a Python list and remove from the front, you get O(n) time. The trick is to use a fixed-size array and maintain two pointers: `front` and `rear`. When you reach the end of the array, you "wrap around" to the start—just like the seats in a round table.

Brute-force approach:

Use a Python list, and on every dequeue, shift all elements left by one. This is O(n) per operation—not good for interviews.

Optimal approach:

Use a fixed-size array (list), and two indices:

- `front`: points to the current front element
- `rear`: points to the next insertion index

Advance indices modulo the size (`(index + 1) % size`) to wrap around.

Python Implementation:

```
class MyCircularQueue:
    def __init__(self, k: int):
        self.size = k
        self.queue = [0] * k
        self.front = 0  # points to the front element
        self.count = 0  # number of elements
        self.rear = 0   # points to the next insertion index

    def enQueue(self, value: int) -> bool:
        if self.isFull():
            return False
        self.queue[self.rear] = value
        self.rear = (self.rear + 1) % self.size
        self.count += 1
        return True

    def deQueue(self) -> bool:
        if self.isEmpty():
            return False
        self.front = (self.front + 1) % self.size
        self.count -= 1
        return True
```

```
    def Front(self) -> int:
        if self.isEmpty():
            return -1
        return self.queue[self.front]

    def Rear(self) -> int:
        if self.isEmpty():
            return -1
        # rear points to the next insertion slot, so last element is at rear-1
        return self.queue[(self.rear - 1 + self.size) % self.size]

    def isEmpty(self) -> bool:
        return self.count == 0

    def isFull(self) -> bool:
        return self.count == self.size
```

Time and Space Complexity:

- All operations: O(1) time (constant).
- Space: O(k), for the buffer.

What does each part do?

- The buffer is a fixed-size list.
- `front` and `rear` wrap around using modulo arithmetic.
- `count` tracks how many elements are in the queue.
- `enQueue` writes at `rear`, then advances.
- `deQueue` advances `front`.
- `Front` and `Rear` use the current indices to fetch values.

Trace Example:

Suppose `k = 3`.

- enQueue(1): queue = [1, _, _], front = 0, rear = 1, count = 1
- enQueue(2): queue = [1, 2, _], front = 0, rear = 2, count = 2
- enQueue(3): queue = [1, 2, 3], front = 0, rear = 0, count = 3 (wraps)
- deQueue(): front = 1, rear = 0, count = 2
- enQueue(4): queue = [4, 2, 3], front = 1, rear = 1, count = 3
- Rear(): returns queue[(1-1+3)%3] = queue[0] = 4

Try this test case yourself:

What happens if you initialize with size 2, enQueue(10), enQueue(20), isFull(), deQueue(), enQueue(30), Rear()?

Take a moment to solve this on your own before jumping into the solution!

Problem 2: Design Circular Deque

LeetCode 641: Design Circular Deque

What's Different?

Now you need to support insertions and deletions at **both the front and the rear**. It's a double-ended queue (deque), but with a fixed-size circular buffer.

Operations:

- `insertFront(value)`, `insertLast(value)`
- `deleteFront()`, `deleteLast()`
- `getFront()`, `getRear()`
- `isEmpty()`, `isFull()`

Example:

```
d = MyCircularDeque(3)
d.insertLast(1)    # True
d.insertLast(2)    # True
d.insertFront(3)   # True
d.insertFront(4)   # False, full
d.getRear()        # 2
d.isFull()         # True
d.deleteLast()     # True
d.insertFront(4)   # True
d.getFront()       # 4
```

Your turn:

Suppose you have a deque of size 2. Try: insertFront(1), insertLast(2), isFull(), deleteFront(), insertLast(3), getRear().

Brute-force Approach:

Use a Python list; insertions/deletions at the front require shifting, so O(n) per operation.

Optimal Approach:

Again, use a fixed-size array, but now allow both ends to advance or retreat.

- Maintain `front` and `rear` pointers.
- For `insertFront`, decrement `front` (with wraparound).
- For `insertLast`, increment `rear`.
- For deletions, adjust pointers accordingly.

Step-by-step:

- The buffer is size k.
- `front` points to the first element (for getFront).
- `rear` points to the next insertion slot at the rear (for insertLast).
- For insertFront, move front backward: `front = (front - 1 + k) % k`
- For insertLast, move rear forward: `rear = (rear + 1) % k`
- For deletions, move pointers in the opposite direction.

Pseudocode:

```
class MyCircularDeque:
```

```
    init(size):
        self.size = size
        self.q = [0] * size
        self.front = 0
        self.rear = 0
        self.count = 0

    insertFront(value):
        if isFull(): return False
        self.front = (self.front - 1 + size) % size
        self.q[self.front] = value
        self.count += 1
        return True

    insertLast(value):
        if isFull(): return False
        self.q[self.rear] = value
        self.rear = (self.rear + 1) % size
        self.count += 1
        return True

    deleteFront():
        if isEmpty(): return False
        self.front = (self.front + 1) % size
        self.count -= 1
        return True

    deleteLast():
        if isEmpty(): return False
        self.rear = (self.rear - 1 + size) % size
        self.count -= 1
        return True

    getFront():
        if isEmpty(): return -1
        return self.q[self.front]

    getRear():
        if isEmpty(): return -1
        return self.q[(self.rear - 1 + size) % size]

    isEmpty():
        return self.count == 0
```

```
    isFull():
        return self.count == self.size
```

Trace Example:

With size 3:

- insertLast(1): front = 0, rear = 1, [1, _, _], count = 1
- insertLast(2): front = 0, rear = 2, [1, 2, _], count = 2
- insertFront(3): front = 2, rear = 2, [1, 2, 3], count = 3
- deleteLast(): rear = 1, count = 2
- insertFront(4): front = 1, [1, 4, 3], count = 3

Test yourself:

Try: insertFront(10), insertLast(20), insertFront(30), deleteLast(), insertLast(40), getFront(), getRear() with size 3.

Time and Space Complexity:

All operations: O(1) time.

Space: O(k).

Problem 3: Design Front Middle Back Queue

LeetCode 1670: Design Front Middle Back Queue

What's New?

Now, you can insert and remove from the **front**, **middle**, or **back**. Think of a queue where someone can jump into the exact middle!

Operations:

- `pushFront(val)`, `pushMiddle(val)`, `pushBack(val)`
- `popFront()`, `popMiddle()`, `popBack()`

Example:

```
q = FrontMiddleBackQueue()
q.pushFront(1)       # [1]
q.pushBack(2)        # [1, 2]
q.pushMiddle(3)      # [1, 3, 2]
q.pushMiddle(4)      # [1, 4, 3, 2]
q.popFront()         # returns 1, queue: [4, 3, 2]
q.popMiddle()        # returns 3, queue: [4, 2]
q.popMiddle()        # returns 4, queue: [2]
q.popBack()          # returns 2, queue: []
```

Try this:

Start with an empty queue and perform: pushBack(5), pushMiddle(10), pushFront(15), popMiddle(), popBack(), popFront().

Brute-force Approach:

Use a list and insert/remove at the appropriate index. For the middle, insertion/removal is O(n).

Optimal Approach:

To get O(1) or O(log n) operations, use two deques to simulate the two halves of the queue:

- `left`: first half
- `right`: second half

Keep them balanced so that the size difference is at most 1. The "middle" is the end of `left` if size is odd, or the beginning of `right` if even.

Pseudocode:

```
class FrontMiddleBackQueue:
    init():
        left = deque()
        right = deque()

    pushFront(val):
        left.appendleft(val)
        balance()

    pushBack(val):
        right.append(val)
        balance()

    pushMiddle(val):
        if len(left) > len(right):
            right.appendleft(left.pop())
        left.append(val)
        balance()

    popFront():
        if not left and not right: return -1
        if left: ret = left.popleft()
        else: ret = right.popleft()
        balance()
        return ret

    popBack():
        if not left and not right: return -1
        if right: ret = right.pop()
        else: ret = left.pop()
        balance()
        return ret

    popMiddle():
        if not left and not right: return -1
```

```
        if len(left) == len(right):
            ret = left.pop()
        else:
            ret = left.pop()
        balance()
        return ret

    balance():
        # Keep len(left) == len(right) or len(left) == len(right) + 1
        while len(left) > len(right) + 1:
            right.appendleft(left.pop())
        while len(left) < len(right):
            left.append(right.popleft())
```

Trace Example:

Suppose you perform pushFront(1), pushBack(2), pushMiddle(3):

- left: [1, 3], right: [2]

Try this test case:

pushMiddle(7), pushBack(8), pushFront(9), popMiddle(), popFront(), popBack()

Time and Space Complexity:

- Each operation: O(1) (since deque operations at both ends are O(1)).
- In the worst case, balancing moves up to O(n) elements, but overall, amortized complexity per operation stays constant.

Summary and Next Steps

These problems are all about **queue variations** and how to efficiently manage data with fixed-size or flexible constraints. Here's what you should remember:

- **Circular buffer**: Use modulo arithmetic to wrap around. Keep track of how many elements you have.
- **Deque**: Allow insertions and deletions at both ends with careful pointer/index management.
- **Front-Middle-Back Queue**: Split data into two parts and keep them balanced to support efficient middle operations.

Common mistakes to watch for:

- Off-by-one errors in index arithmetic.
- Forgetting to wrap pointers using modulo.
- Not maintaining size invariants (like balancing the two halves in Problem 3).
- Over-complicating: Sometimes, two deques or a simple array with indices is enough!

Action List

- Try implementing all three problems yourself—even the one with code provided.
- For Problem 2 and 3, experiment with different data structures (like linked lists or two deques).

- Explore LeetCode discussions or similar problems using circular buffers or multi-ended queues.
- Dry-run your code on paper for edge cases (empty/full, wraparound, etc.).
- If you get stuck, review the patterns above and try again—practice is the best way to master these!

Happy coding!