Mock Interviews @ ACM

Solution Card

Course Queue

Problem Statement

The OIT admins are finally updating TigerHub with a new front-end... as well as a new back-end! Princeton has acquired a new high-speed computing cluster that runs optimally when most operations are performed in small, localized sections of memory. As the head of technology at OIT, you are assigned the task of implementing a very specific data structure for the course queue—a circular queue.

Design your implementation of the circular queue. The circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.

Implementation the `MyCircularQueue` class:

- `MyCircularQueue(k)` Initializes the object with the size of the queue to be `k`.
- `int Front()` Gets the front item from the queue. If the queue is empty, return `-1`.
- `int Rear()` Gets the last item from the queue. If the queue is empty, return `-1`.
- `boolean enQueue(int value)` Inserts an element into the circular queue. Return `true` if the operation is successful.
- `boolean deQueue()` Deletes an element from the circular queue. Return `true` if the operation is successful.
- `boolean isEmpty()` Checks whether the circular queue is empty or not.
- `boolean isFull()` Checks whether the circular queue is full or not.

Source: https://leetcode.com/problems/design-circular-queue/

Hints

Hint 1: What data structure might we use? (array) What are the requirements?

Solution

We can use an array or array-like data structure (in C++, vector) and keep track of the current index of the head of the queue. When we enqueue and dequeue, we can update the index as well as write data to that position, e.g. at i+1 (mod len).

```
class MyCircularQueue {
public:
    /** Initialize your data structure here. Set the size of the queue to be k.
*/
    MyCircularQueue(int k) {
        data.resize(k);
        head = 0;
        tail = 0;
        reset = true;
    }

    /** Insert an element into the circular queue. Return true if the operation
is successful. */
    bool enQueue(int value) {
        if (isFull()) return false;
        // update the reset value when first enqueue happens
        if (head == tail && reset) reset = false;
        data[tail] = value;
        tail = (tail + 1) % data.size();
        return true;
    }

    /** Delete an element from the circular queue. Return true if the operation
is successful. */
    bool deQueue() {
        if (isEmpty()) return false;
        head = (head + 1) % data.size();
        // update the reset value when last dequeue happens
        if (head == tail && !reset) reset = true;
        return true;
    }

    /** Get the front item from the queue. */
    int Front() {
        if (isEmpty()) return -1;
        return data[head];
    }
```

```
    /** Get the last item from the queue. */
    int Rear() {
        if (isEmpty()) return -1;
        return data[(tail + data.size() - 1) % data.size()];
    }

    /** Checks whether the circular queue is empty or not. */
    bool isEmpty() {
        if (tail == head && reset) return true;
        return false;
    }

    /** Checks whether the circular queue is full or not. */
    bool isFull() {
        if (tail == head && !reset) return true;
        return false;
    }
private:
    vector<int> data;
    int head;
    int tail;
    // reset is the mark when the queue is empty
    // to differentiate from queue is full
    // because in both conditions (tail == head) stands
    bool reset;
};

/**
 * Your MyCircularQueue object will be instantiated and called as such:
 * MyCircularQueue obj = new MyCircularQueue(k);
 * bool param_1 = obj.enQueue(value);
 * bool param_2 = obj.deQueue();
 * int param_3 = obj.Front();
 * int param_4 = obj.Rear();
 * bool param_5 = obj.isEmpty();
 * bool param_6 = obj.isFull();
 */
```

Analysis

Time: O(1) for each method, O(n) for constructor

Space: O(1) for each method, O(n) for constructor

Tips and tricks

- For many constant-time operations, we can call our own methods as much as possible for information hiding and cohesion, e.g. Front(), Rear(), isEmpty()
- One of the benefits of the circular queue is that we can make use of the spaces in front of the queue. In a normal queue, once the queue becomes full, we cannot insert the next element even if there is a space in front of the queue. But using the circular queue, we can use the space to store new values.

Follow-ups

- What kinds of data structures might benefit from this circular design? Stack/heap/array/etc? Why or why not?
- What are the benefits of using a circular queue rather than a normal one?