JavaScript Implement Queue Using Stacks

Challenge

Implement a first in first out (FIFO) queue using only two stacks. The implemented queue should support all the functions of a normal queue (`push`, `peek`, `pop`, and `empty`).

Implement the `MyQueue` class:

- `void push(int x)` Pushes element x to the back of the queue.
- `int pop()` Removes the element from the front of the queue and returns it.
- `int peek()` Returns the element at the front of the queue.
- `boolean empty()` Returns `true` if the queue is empty, `false` otherwise.

> ⓘ Note
>
> Depending on your language, the stack may not be supported natively. You may simulate a stack using a list or deque (double-ended queue) as long as you use only a stack's standard operations.

> Important
>
> You must use only standard operations of a stack, which means only `push to top`, `peek/pop from top`, `size`, and `is empty` operations are valid.

Example

```
Input:
['MyQueue', 'push', 'push', 'peek', 'pop', 'empty']
[[], [1], [2], [], [], []]
Output: [null, null, null, 1, 1, false]
Explanation: MyQueue myQueue = new MyQueue();
myQueue.push(1); // queue is: [1]
myQueue.push(2); /* queue is: [1, 2]
                    (leftmost is front of the queue) */
myQueue.peek(); // return 1
myQueue.pop(); // return 1, queue is [2]
myQueue.empty(); // return false
```

Constraints

- `1 <= x <= 9`
- At most `100` calls will be made to `push`, `pop`, `peek`, and `empty`.
- All the calls to `pop` and `peek` are valid.

Solution

```
class MyQueue {
    data = [];

    constructor() {
        this.data = [];
    }
```

Solution continues on next page...

```
    push(x) {
        this.data.push(x);
    }

    pop() {
        const temp = this.data[0];
        let toReturn;

        if(this.data.length > 0) {
            toReturn = temp;
        }

        this.data.shift();

        return toReturn;
    }

    peek() {
        return this.data[0];
    }

    empty() {
        return this.data.length === 0;
    }
};
```

Explanation

I've built a class called `MyQueue` that represents a queue data structure. The class has several methods: `push`, `pop`, `peek`, and `empty`.

The constructor initializes the `data` array, which will be used to store the elements of the queue.

The `push` method takes a parameter `x` and adds the element to the end of the `data` array using the `push` function.

The `pop` method removes and returns the first element of the queue. It assigns the first element to a temporary variable called `temp`. If the queue is not empty (determined by checking the length of the `data` array), it assigns the value of `temp` to a variable called `toReturn`. It then removes the first element of the queue using the `shift` function of the array. Finally, it returns the value of `toReturn`.

The `peek` method returns the first element of the queue without removing it. It simply accesses the element at index `0` of the `data` array.

The `empty` method checks if the length of the `data` array is equal to zero. If it is, it returns true, indicating that the queue is empty. Otherwise, it returns false.

In summary, the `MyQueue` class provides methods to manipulate a queue data structure. It allows elements to be added to the end of the queue, removed from the front of the queue, accessed without removal, and checked for emptiness.

Author: Trevor Morin