225. Implement Stack using Queues

Implement a last in first out (LIFO) stack using only two queues. The implemented stack should support all the functions of a normal queue (`push`, `top`, `pop`, and `empty`).

Implement the `MyStack` class:

- `void push(int x)` Pushes element x to the top of the stack.
- `int pop()` Removes the element on the top of the stack and returns it.
- `int top()` Returns the element on the top of the stack.
- `boolean empty()` Returns `true` if the stack is empty, `false` otherwise.

Notes:

- You must use **only** standard operations of a queue, which means only `push to back`, `peek/pop from front`, `size`, and `is empty` operations are valid.
- Depending on your language, the queue may not be supported natively. You may simulate a queue using a list or deque (double-ended queue), as long as you use only a queue's standard operations.

Example 1:

Input
["MyStack", "push", "push", "top", "pop", "empty"]
[[], [1], [2], [], [], []]
Output
[null, null, null, 2, 2, false]

Explanation
MyStack myStack = new MyStack();
myStack.push(1);
myStack.push(2);
myStack.top(); // return 2
myStack.pop(); // return 2
myStack.empty(); // return False

Constraints:

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

```python
class MyStack:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.q1 = []
        self.q2 = []


    def push(self, x: int) -> None:
        """
        Push element x onto stack.
        """
        self.q1.append(x)


    def pop(self) -> int:
        """
        Removes the element on top of the stack and returns that element.
        """
        while len(self.q1)>1:
            self.q2.append(self.q1.pop(0))
        ret = self.q1[-1]
        self.q1.pop(0)
        self.q1,self.q2 = self.q2,self.q1
        return ret


    def top(self) -> int:
        """
        Get the top element.
        """
        while len(self.q1)>1:
            self.q2.append(self.q1.pop(0))
        ret = self.q1[-1]
        self.q2.append(self.q1.pop(0))
        self.q1,self.q2 = self.q2,self.q1
        return ret



    def empty(self) -> bool:
```

```
        """
        Returns whether the stack is empty.
        """
        return len(self.q1)==0
```

This is pop intensive. That is popping takes time

It takes 2 Queues

1 Queue solution

Approach #3 (One Queue, push - O(n)O(n)O(n), pop O(1)O(1)O(1))

The mentioned above two approaches have one weakness, they use two queues. This could be optimized as we use only one queue, instead of two.

Algorithm

Push

When we push an element into a queue, it will be stored at back of the queue due to queue's properties. But we need to implement a stack, where last inserted element should be in the front of the queue, not at the back. To achieve this we can invert the order of queue elements when pushing a new element.

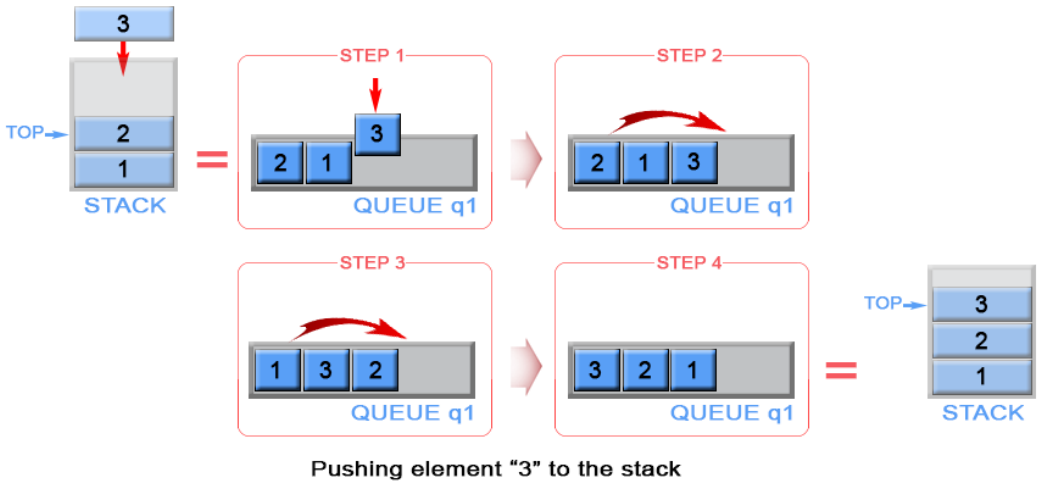

Figure 5. Push an element in stack

Java

```
private LinkedList<Integer> q1 = new LinkedList<>();

// Push element x onto stack.
```

```
public void push(int x) {
    q1.add(x);
    int sz = q1.size();
    while (sz > 1) {
        q1.add(q1.remove());
        sz--;
    }
}
```

Complexity Analysis

- Time complexity : O(n)O(n)O(n). The algorithm removes n elements and inserts n+1n + 1n+1 elements to `q1` , where n is the stack size. This gives 2n+12n + 12n+1 operations. The operations `add` and `remove` in linked lists has O(1)O(1)O(1) complexity.
- Space complexity : O(1)O(1)O(1).

Pop

The last inserted element is always stored at the front of `q1` and we can pop it for constant time.

Java

```
// Removes the element on top of the stack.
public void pop() {
    q1.remove();
}
```

Complexity Analysis

- Time complexity : O(1)O(1)O(1).
- Space complexity : O(1)O(1)O(1).

Empty

Queue `q1` contains all stack elements, so the algorithm checks if `q1` is empty.

```
// Return whether the stack is empty.
public boolean empty() {
    return q1.isEmpty();
}
```

Time complexity : O(1)O(1)O(1).

Space complexity : O(1)O(1)O(1).

Top

The `top` element is always positioned at the front of `q1`. Algorithm return it.

```
// Get the top element.
public int top() {
    return q1.peek();
}
```

Time complexity : O(1)O(1)O(1).

Space complexity : O(1)O(1)O(1).