LeetCode 232 — Implement Queue using Stacks

1. Problem Title & Link

- **Title:** LeetCode 232 — Implement Queue using Stacks
- **Link:** https://leetcode.com/problems/implement-queue-using-stacks/

2. Problem Statement (Short Summary)

Implement a **queue** (FIFO) using two **stacks** (LIFO).
Support the operations:

- push(x) — push element x to the back of the queue.
- pop() — remove and return the element from the front of the queue.
- peek() — return the front element without removing it.
- empty() — return whether the queue is empty.

You must implement these using stack operations only.

3. Examples (Input → Output)

MyQueue q = new MyQueue();

q.push(1);

q.push(2);

q.peek(); // returns 1

q.pop(); // returns 1

q.empty(); // returns false

4. Constraints

- Calls to methods ≤ reasonably large (implementation must be efficient).
- push, pop, peek, empty must behave like a standard queue.
- Allowed operations on stacks: push, pop, peek/top, empty/size.

5. Core Concept (Pattern / Topic)

Simulation using Two Stacks (Amortized O(1) Queue)

Use two stacks:

- inStack — for pushing new elements (enqueue)
- outStack — for popping/peeking (dequeue)

When dequeuing (or peeking) and outStack is empty, **move all elements** from inStack to outStack (this reverses order and brings oldest element to top). This gives **amortized O(1)** per operation.

This is a classic **stack → queue transformation** using reversal.

6. Thought Process (Step-by-Step Explanation)

1. push(x): push onto inStack. Simple.
2. pop():
    - If outStack not empty → pop from outStack.
    - Else move all elements from inStack to outStack, then pop.
3. peek():
    - If outStack not empty → return top of outStack.
    - Else move all elements from inStack → outStack, then return top.
4. empty():
    - True if both stacks empty.

Why it works: moving elements reverses the LIFO order of inStack into FIFO order on outStack, making the oldest element available at top. Each element moves at most once from inStack → outStack, so operations are amortized constant time.

7. Visual / Intuition Diagram

Push sequence: push(1), push(2), push(3)

inStack (top on right): [1,2,3]

outStack: []

On pop():

- outStack empty → move all:
    - pop 3 → push to out → out=[3]
    - pop 2 → out=[3,2]
    - pop 1 → out=[3,2,1] (top is 1)
      Now pop from out → returns 1

Subsequent pops return 2, 3 from out until out empty.

8. Pseudocode

```
initialize inStack = empty stack
initialize outStack = empty stack

function push(x):
    inStack.push(x)

function moveIfNeeded():
    if outStack is empty:
        while inStack not empty:
            outStack.push(inStack.pop())

function pop():
    moveIfNeeded()
```

```
    return outStack.pop()

function peek():
    moveIfNeeded()
    return outStack.top()

function empty():
    return inStack.empty() and outStack.empty()
```

9. Code Implementation

✅ **Python**

```python
class MyQueue:
    def __init__(self):
        # inStack: stack for push operations
        # outStack: stack for pop/peek operations
        self.inStack = []
        self.outStack = []

    def push(self, x: int) -> None:
        self.inStack.append(x)

    def _move_if_needed(self) -> None:
        # Move elements only when outStack is empty
        if not self.outStack:
            while self.inStack:
                self.outStack.append(self.inStack.pop())

    def pop(self) -> int:
        self._move_if_needed()
        return self.outStack.pop()

    def peek(self) -> int:
        self._move_if_needed()
        return self.outStack[-1]

    def empty(self) -> bool:
        return not self.inStack and not self.outStack
```

✅ **Java**

```java
class MyQueue {
```

```
    private Deque<Integer> inStack;
    private Deque<Integer> outStack;

    public MyQueue() {
        inStack = new ArrayDeque<>();
        outStack = new ArrayDeque<>();
    }

    public void push(int x) {
        inStack.push(x); // push onto inStack
    }

    private void moveIfNeeded() {
        if (outStack.isEmpty()) {
            while (!inStack.isEmpty()) {
                outStack.push(inStack.pop());
            }
        }
    }

    public int pop() {
        moveIfNeeded();
        return outStack.pop();
    }

    public int peek() {
        moveIfNeeded();
        return outStack.peek();
    }

    public boolean empty() {
        return inStack.isEmpty() && outStack.isEmpty();
    }
}
```

Note: Using Deque (ArrayDeque) as stack (push/pop/peek) is efficient in Java.

10. Time & Space Complexity

Operation	Amortized Time
push	O(1)
pop	O(1) amortized
peek	O(1) amortized
empty	O(1)

- **Why amortized?** Each element is moved from inStack → outStack at most once. So over a sequence of n operations, total cost of moves is O(n).

Space Complexity: O(n) extra space for the two stacks (stores all elements).

11. Common Mistakes / Edge Cases

❌ **Moving elements on every pop/peek** — unnecessary and costly. Only move when outStack is empty.

❌ **Using pop from inStack for dequeue** — that reverses FIFO.

❌ **Forgetting to check both stacks in empty()** — queue is empty only if both stacks empty.

❌ **Not preserving order when moving** — must pop from inStack and push to outStack (reversal step).

Edge cases:

- Multiple pushes then multiple pops (verify move logic).
- Interleaved push/pop (push after some pops) — ensure subsequent pops work correctly.
- Peek on empty queue — LeetCode assumes valid calls; otherwise handle exceptions.

12. Detailed Dry Run (Step-by-Step Table)

Use this example sequence:

push(1)

push(2)

peek() -> should be 1

pop() -> should return 1

empty() -> should be false

push(3)

pop() -> should return 2

pop() -> should return 3

empty() -> should be true

We start: inStack = [], outStack = []

Step	Operation	inStack (top→right)	outStack (top→right)	Action / Explanation	Return
1	push(1)	[1]	[]	push into inStack	—
2	push(2)	[1,2]	[]	push into inStack	—
3	peek()	[1,2]	[]	outStack empty → move all: pop 2→push out, pop1→push out → out=[2,1]; peek out top=1	1
4	pop()	[ ]	[2]	pop from out → returns 1 (out now [2])	1
5	empty()	[ ]	[2]	both not empty? inStack empty but outStack not → false	FALSE
6	push(3)	[3]	[2]	push into inStack	—
7	pop()	[3]	[2]	outStack not empty → pop out → returns 2	2

8	pop()	[ ]	[ ]	out empty → move in->out: pop 3→out [3] → pop out → returns 3	3
9	empty()	[ ]	[ ]	both empty → true	TRUE

Final sequence of returns: peek()=1, pop()=1, empty()=false, pop()=2, pop()=3, empty()=true — behaves like a queue.

13. Common Use Cases (Real-Life / Interview)

- When only stack primitive is available but queue behavior is needed.
- Implementing FIFO buffers on top of LIFO-only systems.
- Teaching data structure transformations and amortized analysis.
- Interview pattern: “simulate one DS using another”.

14. Common Traps (Important!)

- Moving elements eagerly (every pop/peek) costs O(n) per operation worst-case — avoid.
- Using built-in queue where the problem explicitly asks to implement via stacks (assignment requirement).
- Not handling concurrency (this is single-threaded problem).
- Mistakenly using inStack as the pop source.

15. Builds To (Related Problems)

- **LC 225** — Implement Stack using Queues (reverse problem)
- **LC 155/155** — Min stack variations (stack design patterns)
- Queue/stack interplay problems and amortized analysis exercises.

16. Alternate Approaches + Comparison

Approach	Complexity	Notes
Two stacks (lazy move)	Amortized O(1)	Best and expected solution
Two stacks (move on every op)	O(n) per op worst-case	Simpler but inefficient
Single stack + recursion on pop	O(n) per pop	Not recommended for heavy usage

17. Why This Solution Works (Short Intuition)

inStack collects new items in arrival order. Moving them to outStack reverses the order so the oldest element becomes the top of outStack. Each element is moved at most once, giving amortized constant time per operation — so the two-stack system faithfully reproduces FIFO behavior with good performance.

18. Variations / Follow-Up Questions

- Implement queue with **one** stack and recursion (explain costs).
- Make this thread-safe (synchronization) — discuss locks.
- Implement circular buffer queue for O(1) worst-case.
- Extend to support size() or clear() efficiently.