Problem Statement: Implement a First-In-First-Out (FIFO) queue using only two stacks. The implemented queue should support the following functions:

1. push(int x): Pushes element x to the back of the queue.
2. pop(): Removes the element from the front of the queue and returns it.
3. peek(): Returns the element at the front of the queue.
4. empty(): Returns true if the queue is empty, otherwise returns false.

Constraints:

- Only standard stack operations (push to top, pop from top, peek, and checking if empty) are allowed.
- A list or deque can be used to simulate a stack if necessary.

Understanding the Approach

Thought Process

- Since a stack follows LIFO (Last-In-First-Out) and a queue follows FIFO (First-In-First-Out), directly implementing a queue using a stack is not possible.
- To achieve FIFO behavior, we use **two stacks**:
    - st: The main stack where elements are pushed.
    - helper: A temporary stack used for reversing the order when performing pop and peek operations.
- The idea is to transfer elements from st to helper whenever we need to access the front element.
- After retrieving the required element, we move elements back from helper to st.

Code Explanation

1. Constructor (MyQueue())

```
MyQueue() {}
```

- Initializes the two stacks (st and helper).
- Time Complexity: **O(1)** (constant-time initialization)

2. Push Operation (push(int x))

```
void push(int x) {
   st.push(x);
}
```

- Simply pushes the element onto the st stack.

- Time Complexity: **O(1)** (push operation on a stack is constant time)

3. Pop Operation (pop())

```
int pop() {
   while (st.size() > 0) {
      helper.push(st.top());
      st.pop();
   }
   int x = helper.top();
   helper.pop();
   while (helper.size() > 0) {
      st.push(helper.top());
      helper.pop();
   }
   return x;
}
```

- Transfers all elements from st to helper, reversing their order.
- Retrieves the front element from helper (which corresponds to the queue's front element).
- Moves the remaining elements back to st to restore the order.
- Time Complexity: **O(n)** (since all elements are moved twice in the worst case)

4. Peek Operation (peek())

```
int peek() {
   while (st.size() > 0) {
      helper.push(st.top());
      st.pop();
   }
   int x = helper.top();
   while (helper.size() > 0) {
      st.push(helper.top());
      helper.pop();
   }
   return x;
```

}

- Similar to pop(), but instead of removing the front element, it just retrieves it.
- Time Complexity: **O(n)** (same reasoning as pop())

5. Empty Check (empty())

```
bool empty() {
   if (st.size() > 0) return false;
   else return true;
}
```

- Simply checks if st is empty.
- Time Complexity: **O(1)** (constant-time operation)

Time Complexity Summary

Function	Time Complexity
push(x)	O(1)
pop()	O(n)
peek()	O(n)
empty()	O(1)

Alternative Optimization

- Instead of transferring elements back to st in every pop() and peek(), we can keep them in helper until helper becomes empty. This improves efficiency and makes pop() amortized **O(1)**.

Conclusion

- The code correctly implements a queue using two stacks.
- The current approach ensures correctness but can be optimized for better performance.
- While push() and empty() are O(1), pop() and peek() have an O(n) worst-case time complexity.