232. Implement Queue using Stacks

Easy | Stack | Design | Queue

Problem Description

The task is to implement a queue with the FIFO (first in, first out) principle using two stacks. In a typical queue, elements are added, or 'pushed', to the back and removed, or 'popped', from the front. Additionally, you should be able to 'peek' at the element at the front without removing it and test if the queue is 'empty'. This should be done using only the standard operations of a stack: 'push to top', 'peek/pop from top', 'size', and 'is empty'.

Intuition

The key to solving this problem is to use two stacks, `stk1` and `stk2`, to invert the order of elements twice so that they come out in the same order that they went in. Initially, all new elements are pushed onto `stk1`. However, we can't directly pop from `stk1` for the queue's `pop` operation because stacks follow LIFO order. So, to get the FIFO order of a queue, we reverse `stk1` by popping all its elements and pushing them onto `stk2`. The first element pushed into `stk1` (and therefore the first in queue order) is now at the top of `stk2`, allowing us to perform the FIFO `pop` correctly.

The `move` method is our helper that handles this transfer if `stk2` is empty. It is lazily called only when necessary (when popping or peeking). This efficiency is important as it minimizes the number of operations. Once all elements are transferred to `stk2`, they can be popped or peeked in the correct FIFO order.

The `empty` method simply checks both stacks. If both are empty, the queue is empty.

The elegance of this solution arises from the delayed transfer of elements until necessary (amortized analysis), which minimizes the number of total operations needed.

Solution Approach

The implementation consists of the following steps, utilizing two stacks, `stk1` and `stk2`, which are simply represented as Python lists:

1. **Constructor (`__init__`)**: Two empty stacks are initialized. `stk1` is for adding new elements (push operation), and `stk2` is used for FIFO retrieval (pop and peek operations).

   ```
   self.stk1 = []
   self.stk2 = []
   ```

2. **Push Operation (`push`)**: Elements are added to `stk1`. Each new element is simply appended to the end of `stk1`, which is the top of the stack.

   ```
   def push(self, x: int) -> None:
       self.stk1.append(x)
   ```

3. **Pop Operation (`pop`)**: To remove an element from the front of the queue, we need to get it from the bottom of `stk1`. The `move` method is called to transfer elements from `stk1` to `stk2`, if `stk2` is empty, effectively reversing the stack order. The element at the top of `stk2` (the front of the queue) is then popped.

   ```
   def pop(self) -> int:
       self.move()
       return self.stk2.pop()
   ```

4. **Peek Operation (`peek`)**: Similar to pop, but instead of removing the element at the front of the queue, we only retrieve it. `move` ensures the element is moved to `stk2` so that it can be peeked at.

   ```
   def peek(self) -> int:
       self.move()
       return self.stk2[-1]
   ```

5. **Empty Operation (`empty`)**: This operation checks if both `stk1` and `stk2` are empty. The queue is empty if and only if both stacks are empty.

   ```
   def empty(self) -> bool:
       return not self.stk1 and not self.stk2
   ```

6. **Move Helper Method (`move`)**: This is an essential method that transfers elements from `stk1` to `stk2` when `stk2` is empty. It's called only before a pop or peek operation and only when necessary (i.e., when `stk2` is empty and the next front of the queue needs to be accessed).

   ```
   def move(self):
       if not self.stk2:
           while self.stk1:
               self.stk2.append(self.stk1.pop())
   ```

At its core, this approach leverages the fact that the stack data structure (using `append` and `pop` in Python lists) can be reversed by transferring elements from one stack to another. By having two stacks, we can ensure elements are in the correct FIFO order for queue operations by handling elements in the 'lazy' manner - that is, by only moving elements when absolutely necessary.

Example Walkthrough

Let's walk through a small example to illustrate how the queue implementation using two stacks—`stk1` and `stk2`—works.

Consider the following sequence of operations:

1. `push(1)` - Add the element '1' to the queue.
2. `push(2)` - Add the element '2' to the queue after '1'.
3. `peek()` - Get the element at the front of the queue without removing it.
4. `pop()` - Remove the element from the front of the queue.
5. `empty()` - Check if the queue is empty.

Now let's examine how each operation is handled:

1. **push(1)**: `stk1` receives the element as `[1]`. `stk2` remains `[]`.
2. **push(2)**: `stk1` grows to `[1, 2]`. `stk2` is still `[]`.
3. **peek()**: We want to see the front element of the queue, which is '1'. However, since `stk1` is LIFO, we need to move elements to `stk2` to access '1'. `move()` is called, transferring all elements from `stk1` to `stk2`, resulting in `stk1` as `[]` and `stk2` as `[2, 1]`. Now we can peek the top of `stk2` which is '1', the first element.
4. **pop()**: We need to pop the front element, which is '1'. Since `stk2` already has the correct order, we simply pop from `stk2`. `stk2` becomes `[2]` after popping '1', and `stk1` is still `[]`.
5. **empty()**: To determine if the queue is empty, we check if both `stk1` and `stk2` are empty. Since `stk2` has an element '2', the method returns `False`, indicating the queue is not empty.

The sequence of `stk1` and `stk2` after each operation is shown below:

1. After `push(1)`: `stk1: [1], stk2: []`
2. After `push(2)`: `stk1: [1, 2], stk2: []`
3. After `peek()`: `stk1: [], stk2: [2, 1]`
4. After `pop()`: `stk1: [], stk2: [2]`
5. After `empty()`: No change in stacks, `stk1: [], stk2: [2]`

The queue is operational, demonstrating that two stacks used in this manner can effectively implement a FIFO queue.

Python Solution

```python
class MyQueue:
    def __init__(self):
        # Initialize two stacks
        self.in_stack = []
        self.out_stack = []

    def push(self, x: int) -> None:
        # Push an element onto the end of the queue
        self.in_stack.append(x)

    def pop(self) -> int:
        # Pop an element from the start of the queue
        self._shift_stacks()
        return self.out_stack.pop()

    def peek(self) -> int:
        # Get the front element
        self._shift_stacks()
        return self.out_stack[-1]

    def empty(self) -> bool:
        # Return True if the queue is empty, False otherwise
        return not self.in_stack and not self.out_stack

    def _shift_stacks(self):
        # Move elements from in_stack to out_stack if out_stack is empty
        if not self.out_stack:
            while self.in_stack:
                self.out_stack.append(self.in_stack.pop())

# The MyQueue object will be instantiated and called as following:
# obj = MyQueue()
# obj.push(x)
# param_2 = obj.pop()
# param_3 = obj.peek()
# param_4 = obj.empty()

```

Java Solution

```java
class MyQueue {
    // Use two stacks to simulate a queue:
    // stkInput is used for input operations (push)
    // stkOutput is used for output operations (pop and peek)
    private Deque<Integer> stkInput = new ArrayDeque<>();
    private Deque<Integer> stkOutput = new ArrayDeque<>();

    // Constructor for MyQueue. No initialization needed as
    // member variables are already initialized.
    public MyQueue() {
    }

    // Push element x to the back of the queue. Since a stack is LIFO (last-in, first-out),
    // pushing to stkInput will reverse the order when transferred to stkOutput.
    public void push(int x) {
        stkInput.push(x);
    }

    // Pop the element from the front of the queue.
    // If stkOutput is empty, refill it by popping all elements
    // from stkInput and pushing them into stkOutput.
    public int pop() {
        move();
        return stkOutput.pop();
    }

    // Get the front element.
    // Similar to pop, ensure stkOutput contains elements by moving
    // them from stkInput if necessary and then return the top element.
    public int peek() {
        move();
        return stkOutput.peek();
    }

    // Return true if the queue is empty, which is when both stacks are empty.
    public boolean empty() {
        return stkInput.isEmpty() && stkOutput.isEmpty();
    }

    // Helper method to move elements from stkInput to stkOutput. It ensures that
    // stkOutput contains elements in correct queue order for peeking or popping.
    private void move() {
        // Only move elements if stkOutput is empty.
        if (stkOutput.isEmpty()) {
            // Move all elements from stkInput to stkOutput.
            while (!stkInput.isEmpty()) {
                stkOutput.push(stkInput.pop());
            }
        }
    }
}

/**
 * The following operations demonstrate how to instantiate and operate on the MyQueue object:
 *
 * MyQueue obj = new MyQueue();   // Creates an instance of MyQueue
 * obj.push(x);                   // Pushes element x to the back of the queue
 * int param_2 = obj.pop();       // Retrieves and removes the front element of the queue
 * int param_3 = obj.peek();      // Retrieves but does not remove the front element of the queue
 * boolean param_4 = obj.empty(); // Checks whether the queue is empty
 */

```

C++ Solution

```cpp
#include <stack>
using std::stack;

class MyQueue {
public:
    // Constructor for MyQueue doesn't need to do anything since
    // the standard library stack initializes itself
    MyQueue() {}

    // Adds an element to the back of the queue
    void push(int x) {
        inputStack.push(x);
    }

    // Removes the element from the front of the queue and returns it
    int pop() {
        prepareOutputStack();
        int element = outputStack.top(); // Save the top element
        outputStack.pop();               // Remove element from stack
        return element;                  // Return the saved element
    }

    // Returns the element at the front of the queue without removing it
    int peek() {
        prepareOutputStack();
        return outputStack.top(); // Return the top element
    }

    // Checks if the queue is empty
    bool empty() {
        // The queue is empty only if both stacks are empty
        return inputStack.empty() && outputStack.empty();
    }

private:
    stack<int> inputStack;  // Stack for enqueuing elements
    stack<int> outputStack; // Stack for dequeuing elements

    // Helper function to move elements from inputStack to outputStack
    void prepareOutputStack() {
        // Only move elements if outputStack is empty
        if (outputStack.empty()) {
            while (!inputStack.empty()) {
                outputStack.push(inputStack.top()); // Move element to outputStack
                inputStack.pop();                   // Remove it from inputStack
            }
        }
    }
};

/**
 * Example usage:
 * MyQueue* queue = new MyQueue();
 * queue->push(1);
 * queue->push(2);
 * int elem1 = queue->pop(); // returns 1
 * int elem2 = queue->peek(); // returns 2, the new front after popping 1
 * bool empty = queue->empty(); // returns false since there's still 2 in the queue
 * delete queue; // Don't forget to free memory
 */

```

Typescript Solution

```typescript
// These arrays will act as the stack containers for the queue.
let stack1: number[] = [];
let stack2: number[] = [];

// This function simulates the push operation of a queue, where 'x' is the element to be added to the queue.
function push(x: number): void {
    stack1.push(x);
}

// This function simulates the pop operation of a queue, by moving elements from the first stack to the second if necessary.
function pop(): number {
    moveStacks();
    return stack2.pop();
}

// This function simulates the peek operation of a queue, returning the element at the front without removing it.
function peek(): number {
    moveStacks();
    return stack2[stack2.length - 1];
}

// This function checks whether the queue is empty.
function empty(): boolean {
    return stack1.length === 0 && stack2.length === 0;
}

// This helper function moves elements from stack1 to stack2 if stack2 is empty, effectively reversing the order to simulate queue be
function moveStacks(): void {
    if (stack2.length === 0) {
        while (stack1.length !== 0) {
            stack2.push(stack1.pop());
        }
    }
}

// Usage
// Instead of creating an instance of MyQueue, you would directly call the functions:
push(1);
let val1 = pop();
let val2 = peek();
let isEmpty = empty();

```

Time and Space Complexity

Time Complexity:

- `__init__()`: O(1) - Initializing two empty stacks takes constant time.
- `push(x)`: O(1) - Append operation on a list (stack) is an amortized constant time operation.
- `pop()`: Amortized O(1) - In the worst case, this operation can be O(n), where n is the number of elements in `stk1`, because it has to move all elements from `stk1` to `stk2` if `stk2` is empty. However, each element is only moved once due to the two-stack arrangement, and, across a series of `m` operations, this gives an average (or amortized) time complexity of O(1).
- `peek()`: Amortized O(1) - Similar to `pop()`, it may involve moving all elements from `stk1` to `stk2` in the worst case, but due to the amortized analysis, it averages to constant time.
- `empty()`: O(1) - Checking if two lists are empty is a constant time operation.
- `move()`: Amortized O(1) - Although it can be O(n) in the worst case when moving all elements from `stk1` to `stk2`, it is part of the `pop()` and `peek()` operations and contributes to their amortized time complexity.

Space Complexity:

- Overall space complexity for the `MyQueue` class is O(n), where n is the number of elements in the queue at a given time. This is because all elements are stored in two stacks (`stk1` and `stk2`). No additional space is used that is proportional to the number of elements in the queue except for these two stacks.