Project Title: Fixed-Capacity Queue Implementation Using Two Stacks

1. Introduction

In this project, you will build a queue—a First-In-First-Out (FIFO) data structure—using two stacks, which naturally follow a Last-In-First-Out (LIFO) order. This exercise challenges you to combine these two contrasting data structures to simulate the behavior of a queue. Additionally, you will implement capacity constraints and a full suite of queue operations that mirror those found in standard libraries.

2. Objectives

- **Data Structures Mastery:**
  Gain a deeper understanding of stacks and queues by implementing a queue using two stacks.
- **Algorithm Design:**
  Develop an efficient algorithm that leverages two stacks to ensure that enqueue and dequeue operations maintain FIFO order.
- **Capacity Management:**
  Incorporate a fixed capacity for the queue, ensuring that no more than the specified number of elements can be stored.
- **Method Semantics:**
  Implement the following methods with behavior consistent with Java's and Python's queue interfaces:
  - `add(element)` : Inserts an element. Returns true if successful; throws an exception if the queue is full.
  - `offer(element)` : Inserts an element. Returns true if the element was added, or false if the queue is full.
  - `remove()` : Removes and returns the head of the queue. Throws an exception if the queue is empty.
  - `poll()` : Removes and returns the head of the queue. Returns null (or `None` in Python) if the queue is empty.
  - `element()` : Retrieves, but does not remove, the head of the queue. Throws an exception if the queue is empty.

    - `peek()` : Retrieves, but does not remove, the head of the queue. Returns null (or `None` in Python) if the queue is empty.

- **Testing:**
  Ensure your solution passes a series of 25 test cases (provided in a Java testing file) that cover normal operations, edge cases, and error conditions.

3. Requirements

Functional Requirements

1. **Fixed Capacity:**
   The queue must have a fixed capacity defined at initialization (for example, 3 elements).

2. **Queue Operations:**

   - **Enqueue Operations:**
     - `add(element)` : Should add an element if there is available space; if not, throw an exception.
     - `offer(element)` : Should add an element if there is available space; if not, return false.
   - **Dequeue Operations:**
     - `remove()` : Should remove and return the head element; if the queue is empty, throw an exception.
     - `poll()` : Should remove and return the head element; if the queue is empty, return null (or `None`).
   - **Peek Operations:**
     - `element()` : Should return the head element without removing it; throw an exception if the queue is empty.
     - `peek()` : Should return the head element without removing it; return null (or `None`) if the queue is empty.

3. **Data Structure Implementation:**
   Use two stacks:

   - **Stack In:** Used for enqueue (insertion) operations.
   - **Stack Out:** Used for dequeue (removal) and peek operations. When `stackOut` is empty, transfer all elements from `stackIn` to `stackOut` to reverse the order, ensuring FIFO behavior.

4. **Error Handling:**

    - The methods `remove()` and `element()` must throw exceptions (e.g., `NoSuchElementException` in Java or `IndexError` in Python) if the queue is empty.
    - The `add()` method must throw an exception if the queue is full.
    - The methods `offer()`, `poll()`, and `peek()` should return a failure value (`false` or `null` / `None`) when the operation cannot be performed.

Non-Functional Requirements

- **Efficiency:**
  Aim for O(1) amortized time for queue operations by minimizing the number of transfers between stacks.

- **Code Quality:**
  Ensure your code is well-organized, with clear documentation and modular design.

- **Portability:**
  The solution should be implemented in a way that it can be developed in either Java or Python with equivalent functionality.

4. Design Overview

Two-Stack Concept

1. **Data Structures:**

    - **stackIn:** Stores incoming elements. All enqueue operations push items onto this stack.
    - **stackOut:** Used for dequeue and peek operations. When this stack is empty, move all elements from `stackIn` to `stackOut` to reverse the order, making the oldest element accessible.

2. **Capacity Check:**
   Before adding a new element using `add()` or `offer()`, check that the current size (the sum of elements in both stacks) is less than the fixed capacity.