Project Title: Custom Queue Implementation

Overview

This project focuses on designing and implementing a Queue data structure from scratch. The Queue follows the First-In-First-Out (FIFO) principle. The goal is to gain a deep understanding of how queues work, including capacity restrictions, exception handling, and the difference in underlying storage mechanisms (using dynamic arrays versus linked lists). In this project, we provide two distinct implementations:

1. **ArrayList-Based Queue (Student Created):**
   This version uses an underlying dynamic array (or a circular array approach) to manage elements. It is best suited for scenarios where random access might be needed, and it demonstrates how to handle fixed capacity restrictions by throwing exceptions or returning a status indicator.

2. **LinkedList-Based Queue (Student Created):**
   This version uses a custom singly linked list. It demonstrates efficient insertion and removal at the head of the list and provides a clear example of pointer-based data structures.

Both implementations expose the same methods so that they can be interchanged easily in any application.

Queue Methods Description

Each Queue implementation supports the following operations:

1. **`boolean add(E e)`**

   - **Purpose:** Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.
   - **Behavior:** Returns `true` upon successful insertion. If the queue is full (i.e., no space is available), it throws an `IllegalStateException` (or a custom exception in other languages).
   - **Use Case:** Use this method when you expect the element to be added and want an exception to be raised if the queue is full.

2. **`E element()`**

   - **Purpose:** Retrieves—but does not remove—the head of this queue.
   - **Behavior:** Throws a `NoSuchElementException` if the queue is empty.

  - **Use Case:** When you need to examine the head element and you require an exception if the queue is empty.

3. **`boolean offer(E e)`**

   - **Purpose:** Inserts the specified element into this queue if it is possible to do so immediately.
   - **Behavior:** Returns `true` upon successful insertion. If the queue is full, it returns `false` instead of throwing an exception.
   - **Use Case:** Use this method when you prefer a non-exception-based approach to handling full queues.

4. **`E peek()`**

   - **Purpose:** Retrieves—but does not remove—the head of this queue.
   - **Behavior:** Returns the head element if it exists; if the queue is empty, it returns `null` .
   - **Use Case:** This method is used when you want to safely check the head element without risking an exception.

5. **`E poll()`**

   - **Purpose:** Retrieves and removes the head of this queue.
   - **Behavior:** Returns the head element if it exists; if the queue is empty, it returns `null` .
   - **Use Case:** Use this method when you want to remove and return the head element but want a safe return value (null) if the queue is empty.

6. **`E remove()`**

   - **Purpose:** Retrieves and removes the head of this queue.
   - **Behavior:** Throws a `NoSuchElementException` if the queue is empty.
   - **Use Case:** Use this method when you require the removal of the head element and want an exception if the queue is empty.

Implementation 2: LinkedList-Based Queue (Student Created)

Design & Approach

- **Underlying Structure:**
  This implementation uses a custom singly linked list where each node holds an element and a reference to the next node.
- **Key Points:**

  - **Dynamic Memory:** Nodes are created on demand, so there is no fixed array size. (For demonstration, you can simulate a capacity if desired.)
  - **FIFO Order:**
    - **Insertion:** New elements are added at the tail of the list.
    - **Removal:** Elements are removed from the head of the list.
  - **Method Operations:**
    All operations follow the same behavior as described:
    - `add(e)` throws an exception when capacity is reached.
    - `offer(e)` returns false if the queue is full.
    - `element()` , `peek()` , `poll()` , and `remove()` work on the head node.

Conclusion

This project gives a thorough understanding of queue behavior with detailed descriptions of each method:

- **Insertion Methods:**
  `add(e)` (throws an exception if full) and `offer(e)` (returns false if full).
- **Inspection Methods:**
  `element()` (throws exception if empty) and `peek()` (returns null if empty).
- **Removal Methods:**
  `poll()` (returns null if empty) and `remove()` (throws exception if empty).