**Criminal Record & FIR Management System**

****

**By**

**Haseeb Ahmad(55197)**

**Faisal Sharif(53307)**

**Supervised**

**by**

**Muhammad Usman**

**DEPARTMENT OF FACULTY OF COMPUTING**

**RIPHAH INTERNATIONAL UNIVERSITY,**

**ISLAMABAD, PAKISTAN**

**FIR and Criminal Record Management System**

**Introduction and Objectives**

This project, titled the **FIR and Criminal Record Management System** is a console-based application implemented in **Java**. Its core purpose is to demonstrate the practical, hands-on application of fundamental **Abstract Data Types** specifically the **Stack** and the **Queue**—through custom implementation. The system simulates essential administrative workflow processes by managing **Criminal Records** and **First Information Reports (FIRs)**.

The primary objective is to build functional data structures from scratch (Linked List for Stack, Circular Array for Queue) and apply them to a real-world scenario where specific data access policies (LIFO and FIFO) are mandatory for procedural integrity.

**Enforcing Procedural Workflow**

Administrative tasks in record management often require adherence to strict rules for processing order. For example, justice systems frequently prioritize the oldest pending cases (FIRs) to prevent backlogs (FIFO), while criminal processing might focus on the most recently detained individuals (LIFO).

Relying on generic Java collections like ArrayList or HashMap would either fail to enforce these policies or require complex, inefficient code to simulate them. The need is for **custom data structures** that intrinsically enforce the required **Last-In, First-Out** and **First-In, First-Out** behavior.

**Proposed Solution: ADT-Driven Data Flow**

The JRMS provides a modular solution by encapsulating the specific access logic within dedicated classes (CriminalStack and FIRQueue).

**2. Detailed Design Specification**

The system employs a clean **Object-Oriented Design (OOD)** structure, separating data (Criminal, FIR) from behavior (CriminalStack, FIRQueue, AdminSystem).

**2.1. Criminal Record Management: The Linked List Stack**

The CriminalStack implements the **Stack ADT** (LIFO) using a **Singly Linked List** structure, defined by the CriminalNode class. This provides dynamic memory allocation, preventing arbitrary size limits.

**A. Classes**

* **Criminal:** Data object storing name, age, and address.
* **CriminalNode:** The container for the data, holding a Criminal object and a pointer (next) to the subsequent node.

**2.2. FIR Case Management: The Circular Array Queue**

The FIRQueue implements the **Queue ADT** (FIFO) using a fixed-size **Circular Array** structure. This design efficiently utilizes memory by allowing the front and rear pointers to wrap around the array, eliminating the need to shift elements upon dequeueing.

**A. Classes**

* **FIR:** Data object storing caseNumber, description, and the operational status (defaulting to "Pending").

**B. Operations and Complexity**

The circular array uses three integer indices (front, rear, count) to manage state and detect boundary conditions.

**Advanced Analysis and Future Enhancements**

**3.1. Code Critique and Best Practice Recommendations**

The current code is robust and correct for an academic demonstration. To enhance its quality to professional standards, particularly in the context of advanced programming, the following refinements are suggested:

1. **Standardized Exception Handling:**
   * The pop() and dequeue() methods should be modified to **return the removed object** rather than void.
   * On an underflow condition (empty stack/queue), the methods should **throw a standard Java Exception** (e.g., java.util.EmptyStackException or java.util.NoSuchElementException) instead of just printing a console error. This forces the calling AdminSystem to handle potential failure.
2. **Memory Management (Garbage Collection):**
   * In the FIRQueue's dequeue() method, explicitly set the array slot being freed to null before updating the front pointer: queue[front] = null;. This immediately releases the object reference, helping the Java Garbage Collector reclaim memory sooner and preventing memory leaks in long-running applications.
3. **Encapsulation:**
   * Consider making the CriminalNode a **private static nested class** within CriminalStack. Since CriminalNode is purely an implementation detail, hiding it improves the modularity and public interface of the system.

**3.2. Future Expansion**

The system can be significantly enhanced by introducing more advanced data structures and features:

1. **Priority Queue for FIR Processing:**
   * Replace the FIRQueue (Circular Array) with a **Priority Queue** implemented using a **Min-Heap** or **Max-Heap**. This would allow FIRs to be processed based on an assigned priority level (e.g., severity of the crime), overriding the strict chronological FIFO order to better model real-world case management.
2. **Optimized Searching (Criminals):**
   * For fast criminal record retrieval, integrate a data structure that provides O(logn) search time, such as a **Binary Search Tree** (or a self-balancing AVL Tree for guaranteed performance), or a **Hash Table** for average O(1) search.
3. **Data Persistence:**
   * Implement **File I/O** (using serialization or text files) to save the current state of the CriminalStack and FIRQueue to the disk, ensuring that data persists between application sessions.