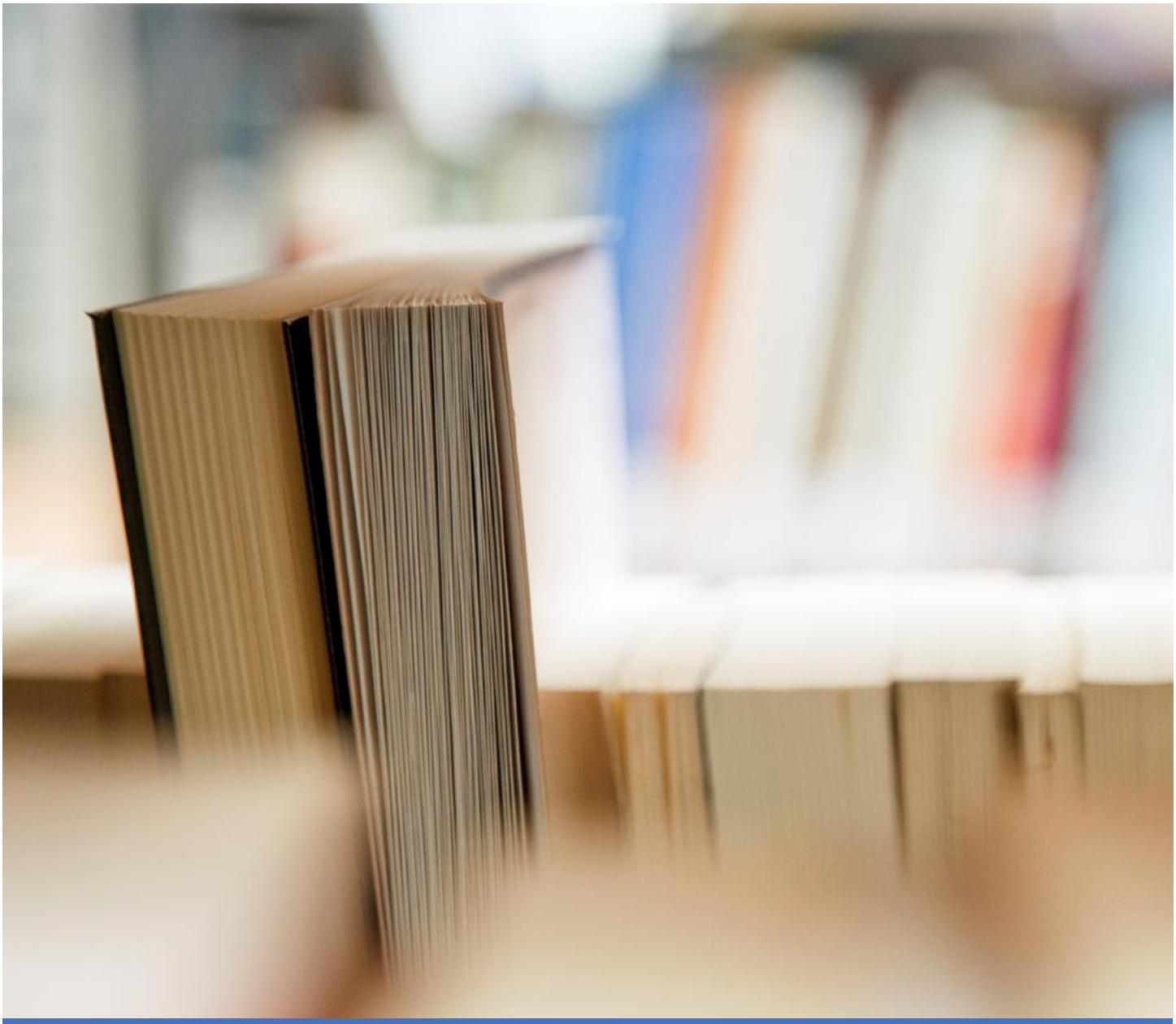

CS3515: Advanced Programming with Java

A03: Test Driven Development (TDD)

Advait Mathur Student Number: 123104113 05/12/2025

In this assignment we have been tasked to develop a LRUCache class that implements the LRU algorithm using TDD. The implementation details of the program are as follows:

1. The LRU uses a combination of a HashMap and a doubly-linked list:
   - A HashMap<Integer, Node> stores cache entries and allows O(1) lookup by the object's hash code.
   - A private inner Node class stores:
     - Object value
     - Node prev
     - Node next
   - All nodes are linked in a doubly linked list, where:
     - The head is the most recently used item
     - The tail is the least recently used item

This structure allows fast updates of the usage order and fast eviction of the least recently used entry.

2. Recentness is maintained by rearranged nodes in the doubly linked list:
   - When put(value) is called:
     - If the value already exists: Its node is moved to the head, making it most recent.
     - If it does not exist:
       - A new node is added to the head.
       - If capacity is exceeded – The tail node is removed (LRU eviction).
   - When get(hash) is called:
     - If the item is present – Its node is moved to the head.
     - This reflects that it has been recently accessed.

Recentness is therefore maintained by efficient O(1) linked-list operations (moveToHead() and removeTail()).

3. State testing focuses on observable inputs and outputs of the LRUCache class. The following behaviours can be tested:
   - Constructor behaviour
     - Creating a cache with invalid capacity throws an exception.
   - put(value) behaviour
     - New values are stored.
     - The method returns the hash code of the value.
     - Inserting an existing value updates its recentness.
     - Inserting a new value when cache is full evicts the LRU element.
   - get(hash) behaviour
     - Returns the correct object when present.
     - Throws NoSuchElementException when the hash is not present.
     - Accessing an object updates its recentness.

All of these are directly testable without inspecting internal data structures, satisfying state-based testing requirements.

4. To support testability and simplify logic, the implementation includes:
    - A private Node class
        - Encapsulates value, prev and next pointers.
        - This keeps list structure clean, predictable, and easy to reason about during testing.
    - Helper methods
        - Small internal methods allow behaviour to be implemented and tested indirectly via state:
            - moveToHead(Node node): Abstracts priority updates.
            - addToHead(None node): Ensures consistent insertion logic.
            - removeTail(): Ensures eviction logic is consistent.

These methods are not tested directly (white-box), but they ensure stable behaviour that is verifiable through state testing.

The approach was beneficial because TDD ensured every feature of the LRU emerged only in response to a failing test, improving correctness and preventing lengthy and unnecessary code.

However, the issues I faced were:

1. Strict TDD slowed early development because functionality had to evolve in very small steps.
2. The logic of the program had to thought through such as using HashMap and doubly linked list required me to implement certain helper methods.

Overall, the approach supports correctness and testability alongside careful and disciplined development.