LRU Cache and DoublyLinkedList Implementation

Introduction

As a graduate student exploring efficient data structures, implementing an **LRU (Least Recently Used) Cache** using a **Doubly Linked List** and **HashMap** is a valuable exercise. This implementation balances **performance, concurrency, and memory management**, making it ideal for real-world applications like web caching, database optimizations, and memory management systems.

DoublyLinkedList Implementation

The **DoublyLinkedList** supports the following operations efficiently:

- `addFirst()` : Adds a node to the front (most recently used)
- `addLast()` : Adds a node to the end (least recently used)
- `remove()` : Removes a specific node from anywhere in the list
- `removeLast()` : Removes the least recently used node from the end

Edge Cases Handled:

- Managing an **empty list** without errors
- Handling **single node conditions**
- Properly maintaining **head and tail pointers**
- Connecting and disconnecting **neighboring nodes correctly**

> **Why This Matters?** In an LRU cache, maintaining order is crucial to ensuring fast retrieval and eviction of elements.

LRUCache Implementation

The **LRU Cache** is built using:

- **A HashMap** for **O(1) lookup time**
- **A Doubly Linked List** to efficiently maintain **usage order**

Thread Safety Mechanism

Graduate students working with **multithreading** need to ensure that shared resources are accessed safely. To achieve this, we use **ReentrantReadWriteLock**, which provides:

- **Read lock** for operations that do not modify the cache (like `get()`).
- **Write lock** for operations that modify the cache (`put()`, eviction, reordering nodes).

Key Operations

`put(K key, V value)`

- If the key **exists**, update its value and move it to the front.
- If the key **does not exist**, create a new node.
- If the cache is **at capacity**, remove the least recently used node before adding a new one.
- Ensure **cache size consistency** by maintaining an accurate count.

`get(K key)`

- First, **attempt with a read lock** to check for the key.
- If found, **upgrade to a write lock** to move the node to the front.
- Use **try-finally blocks** to ensure proper lock handling.

Concurrency Considerations

Avoiding Race Conditions

- **Capacity Enforcement**: Prevents cache overflow using a write lock.
- **Cache Size Consistency**: Ensures cache count and structure remain in sync.
- **Access Ordering**: Maintains the correct order of usage by protecting list modifications.
- **Eviction Race Condition Prevention**: Ensures only one thread can evict a node at a time.
- **Node Manipulation Safety**: Prevents disconnected or incorrectly linked nodes.

Performance Considerations

- **Read-Write Locking**: Improves throughput for read-heavy workloads.
- **Lock Granularity**: Minimizes the duration of lock holding for efficiency.
- **Lock Upgrading**: `get()` method starts with a read lock and upgrades only when necessary.

Time Complexity Analysis

Operation	Time Complexity
`get()`	**O(1)**
`put()`	**O(1)**

> This ensures high efficiency, even with concurrent access, making it **ideal for large-scale systems**.

Conclusion

This implementation balances **efficiency, scalability, and concurrency**. As a graduate student, understanding these concepts will help you in **systems design interviews**, **real-world software engineering**, and **distributed computing applications**.