Cache Simulation - Project Documentation

Introduction

This project simulates the core behavior of a cache system, focusing on how limited memory is managed efficiently by retaining only the most relevant data. It supports three widely used cache replacement policies: FIFO (First-In-First-Out), LRU (Least Recently Used), and LFU (Least Frequently Used). To achieve O(1) time complexity for get, put, and eviction operations, the system combines key data structures such as queues, doubly linked lists, and hash maps.

The Problem

Caches are critical in speeding up data access by storing frequently or recently used data. However, cache memory is limited in size. This simulation allows users to explore how different replacement policies affect cache behavior and performance under memory constraints.

Users can:

- Set cache capacity (between 1 and 10).
- Perform Get operations (retrieve data using a memory address).
- Perform Put operations (update data at a memory address; when the item is eventually evicted, the change is written back to main memory).

Performance

Time Complexity:

- Get: O(1)
- Put: O(1)

(This applies to all replacement policies: FIFO, LRU, and LFU.)

Space Complexity:

- FIFO: O(capacity) - A queue of size capacity is used.
- LRU: O(capacity) - A doubly linked list and hash map of size capacity.

✧ LFU: O(capacity) - A doubly linked list for each frequency and hash maps for fast lookups.

Constraints

✧ Cache capacity: 1-10

✧ Memory addresses must be within a valid range (0-15)

✧ All operations must be optimized for O(1)

✧ Total space usage should remain practical for small-scale simulations

Choice of Data Structures

✧ Queue: Used in FIFO for maintaining insertion order.

✧ Doubly Linked List: Enables fast insertion/deletion from any position.

✧ Hash Map: Enables constant-time lookup and access.

Algorithm Design

✧ FIFO: Remove the oldest inserted item when cache is full.

✧ LRU: Move recently accessed items to the end; remove least recently accessed on eviction.

✧ LFU: Track usage frequency and evict items with the lowest frequency. Ties are broken using recency.

Class Overview

MainMemory Class

The MainMemory class emulates a simple block of main memory with 17 memory slots (0–16). Each address stores a default value (initially 'A' to 'Q').

Structure: A 2D array is used where each row contains the memory address and its corresponding value.

Static Initialization: It uses a static block to populate the memory with ASCII characters for default values.

Functions:

- ✧ isValid(address): Ensures an operation is performed only on valid memory locations.
- ✧ getValueAt(address): Returns the value stored at the given address.
- ✧ setValueAt(address, value): Updates the value at the specified address.

This class acts as a stand-in for actual memory, which is modified only when an item is evicted from the cache.

FIFO Class

The FIFO cache uses a queue-based eviction strategy—items are removed in the order they were added when the cache is full.

Data Structures:

- ✧ Queue<Node>: Tracks insertion order.
- ✧ Map<Integer, Node>: Enables fast access to cache entries by key.

Behavior:

- ✧ On a **cache miss**, data is fetched from MainMemory, added to the cache, and the oldest entry is removed if full.
- ✧ On a **cache hit**, the value is directly returned.
- ✧ On **insertion**, if the key already exists, the value is updated.

Trade-offs:

- ✧ Simple and fast eviction.
- ✧ No intelligence in eviction—it may remove frequently used data.

Use case: Suitable where temporal ordering is more relevant than usage frequency.

LRU Class:

The LRU cache maintains a doubly linked list to track access order, with the least recently accessed item at the tail.

Data Structures:

- ✧ HashMap<Integer, Node>: Allows O(1) lookup.
- ✧ Doubly Linked List: Tracks the usage order.

Behavior:

- ✧ On **get**, the accessed node is moved to the front.
- ✧ On **put**, if the cache is full, the node at the tail (least recently used) is removed.
- ✧ Removed items are persisted back to MainMemory.

Design:

- ✧ Uses a dummy head and tail node to simplify insertions and deletions.
- ✧ Moves recently accessed items to the front (most recently used).

Ideal for scenarios where data locality is based on recent access history.

LFU Class:

The LFU cache tracks access frequency and evicts the **least frequently accessed** item. Among those with the same frequency, the least recently used is removed.

Data Structures:

- ✧ HashMap<Integer, Node>: Maps keys to nodes.
- ✧ Map<Integer, DoubleLinkedList>: Groups nodes by frequency count.
- ✧ Node: Stores key, value, frequency, and list pointers.

Behavior:

- ✧ **Get** increases a node's frequency and moves it to the corresponding frequency list.
- ✧ **Put** evicts the least frequently used item if the cache is full.
- ✧ Tracks minFrequency to quickly identify which frequency bucket to evict from.

Design Highlights:

- ✧ Nodes are organized into multiple frequency-based doubly linked lists.
- ✧ Cache operations are kept at O(1) by using frequency maps and direct pointers.

Best used when items that are accessed frequently should remain in the cache, even if they were accessed a long time ago.

Summary of Class Design Choices

Class	Key Structure	Eviction Policy	Time Complexity	Notes
MainMemory	2D Array	N/A	O(1)	Acts as a banking memory.
FIFOCache	Queue + HashMap	Oldest item removed	O(1)	Simple, order-based
LRUCache	DLL + HashMap	Least recently used	O(1)	Access pattern-aware.
LFUCache	Multi-DLL + HashMap	Least frequently used	O(1)	Frequency-aware, most complex.