**DSA Project – 2**

**1. What is Caching? Why is it Needed?**

Caching is a technique used to store frequently accessed data in a faster storage medium, reducing latency and improving performance. It helps minimize expensive computations or repeated data fetching from slower storage, such as databases or disk-based systems. Caching is essential in:

* Web applications for faster page loads.
* Database queries to reduce redundant computations.
* Operating systems for quick memory access.
* Content Delivery Networks (CDNs) for efficient data distribution.

**2. In-Memory Cache: Redis and Memcached**

**Redis**

Redis is an open-source, in-memory key-value store that supports advanced data structures like lists, sets, and sorted sets. It is widely used for:

* Real-time analytics
* Session storage
* Message brokering
* Leaderboards in gaming

**Memcached**

Memcached is a high-performance, distributed memory caching system designed for simplicity and speed. It is ideal for caching database query results and web page fragments, reducing the database load and increasing scalability.

**3. Cache Memory in Computer Organization**

Cache memory is a small, high-speed storage layer between the CPU and main memory (RAM). It improves processing speed by storing frequently accessed instructions and data. The cache hierarchy consists of:

* L1 Cache: Closest to the CPU, extremely fast but small.
* L2 Cache: Larger than L1, slightly slower.
* L3 Cache: Shared among CPU cores, improves multi-threaded performance.

**4. Different Cache Replacement Strategies**

Cache replacement policies decide which data should be removed when the cache is full. Common strategies include:

* Least Recently Used (LRU): Removes the least recently accessed item.
* First-In-First-Out (FIFO): Evicts the oldest item in cache.
* Least Frequently Used (LFU): Removes the least accessed item over time.
* Random Replacement: Randomly selects an entry for eviction.

**5. Designing LRU Cache Using Doubly Linked List and HashMap**

The Least Recently Used (LRU) Cache maintains a fixed-size cache that evicts the least recently accessed item when full. It is implemented using:

* Doubly Linked List: Maintains the order of access for fast eviction.
* HashMap (or HashTable): Provides O(1) lookup time for fast retrieval.

Algorithm:

1. Insert (Put operation):
   * If the key exists, update the value and move the node to the front.
   * If the key does not exist:
     + Add a new node to the front.
     + If the cache exceeds capacity, remove the last node (least recently used).
2. Retrieve (Get operation):
   * If the key exists, move it to the front and return its value.
   * If the key does not exist, return -1.

**6. UML Diagram for LRU Cache Design**

A UML diagram helps visualize the structure of an LRU cache, typically including:

* **LRUCache Class**: Stores the cache capacity, HashMap, and Doubly Linked List.
* **Node Class**: Represents the doubly linked list nodes with key-value pairs.
* **Operations**:
  + get(key): Fetches an item and moves it to the front.
  + put(key, value): Inserts or updates an item and maintains the eviction policy.

+------------------+

| LRUCache |

+------------------+

| - capacity: int |

| - cache: HashMap |

| - head: Node |

| - tail: Node |

+------------------+

| + get(key): int |

| + put(key, val) |

+------------------+

|

v

+------------------+

| Node |

+------------------+

| - key: int |

| - value: int |

| - prev: Node |

| - next: Node |

+------------------+

**Code**

import java.util.\*;

// LRU Cache Implementation

class LRUCache {

private CDLL ll; // Circular Doubly Linked List to maintain order

private int capacity; // Maximum cache size

private int size; // Current cache size

private Map<Integer, CDLLNode> mp; // HashMap for quick access to nodes

// Constructor: Initializes the cache with given capacity

public LRUCache(int cap) {

ll = new CDLL(); // Create an empty Circular Doubly Linked List

this.capacity = cap;

this.size = 0;

mp = new HashMap<>(); // Initialize the HashMap

}

// Get method to retrieve a value by key

public int get(int key) {

if (mp.containsKey(key)) { // Check if key exists in cache

CDLLNode node = mp.get(key);

ll.moveToFront(node); // Move accessed node to the front (most recently used)

return node.val; // Return the corresponding value

} else {

return -1; // Key not found

}

}

// Put method to insert or update a key-value pair

public void put(int key, int value) {

if (mp.containsKey(key)) { // If key exists, update its value

CDLLNode node = mp.get(key);

node.val = value; // Update the value

ll.moveToFront(node); // Move updated node to the front

} else { // If key is new, insert it into the cache

if (size < capacity) { // If cache is not full

CDLLNode newNode = ll.insertAtFront(key, value); // Insert at front

mp.put(key, newNode); // Add to HashMap

size++; // Increase cache size

} else { // Cache is full, remove least recently used (LRU) item

int delKey = ll.deleteLast(); // Delete the last (LRU) node

mp.remove(delKey); // Remove from HashMap

CDLLNode newNode = ll.insertAtFront(key, value); // Insert new key-value

mp.put(key, newNode); // Add new node to HashMap

}

}

}

}

// Node class for Circular Doubly Linked List

class CDLLNode {

int key, val; // Key-value pair

CDLLNode prev, next; // Pointers to previous and next nodes

public CDLLNode(int k, int v) {

this.key = k;

this.val = v;

this.prev = this.next = null; // Initialize pointers

}

}

// Circular Doubly Linked List (CDLL) class

class CDLL {

private CDLLNode head; // Head pointer

public CDLL() {

head = null; // Initialize as empty list

}

// Insert a new node at the beginning (Most Recently Used)

public CDLLNode insertAtFront(int key, int value) {

CDLLNode newNode = new CDLLNode(key, value);

newNode.next = newNode;

newNode.prev = newNode;

if (head == null) { // If list is empty, set head to new node

head = newNode;

return newNode;

}

CDLLNode last = head.prev; // Get last node

newNode.next = head;

head.prev = newNode;

last.next = newNode;

newNode.prev = last;

head = newNode; // New node becomes head

return newNode;

}

// Delete the least recently used (last node)

public int deleteLast() {

if (head == null) return -1; // Empty list, return -1

CDLLNode last = head.prev; // Get last node

int removedKey = last.key; // Store key of removed node

if (last == head) { // Only one element in the cache

head = null; // List becomes empty

} else {

last.prev.next = head; // Update previous node's next pointer

head.prev = last.prev; // Update head's previous pointer

}

return removedKey; // Return removed key

}

// Move an accessed node to the front (Most Recently Used)

public void moveToFront(CDLLNode nodeToMove) {

if (nodeToMove == head) return; // Already at front, no action needed

// Detach the node from its current position

nodeToMove.prev.next = nodeToMove.next;

nodeToMove.next.prev = nodeToMove.prev;

// Attach it at the front

CDLLNode last = head.prev; // Get last node

nodeToMove.next = head; // New node's next is current head

head.prev = nodeToMove; // Head's previous becomes new node

last.next = nodeToMove; // Last node's next becomes new node

nodeToMove.prev = last; // New node's previous becomes last node

head = nodeToMove; // New node becomes the head

}

}

// Main Class to Test LRU Cache

public class Main {

public static void main(String[] args) {

LRUCache cache = new LRUCache(3); // Create an LRU cache with capacity 3

// Insert key-value pairs into the cache

cache.put(1, 10);

cache.put(2, 20);

cache.put(3, 30);

System.out.println(cache.get(1));

cache.put(4, 40);

System.out.println(cache.get(2));

System.out.println(cache.get(3));

cache.put(5, 50);

System.out.println(cache.get(1));

System.out.println(cache.get(4));

System.out.println(cache.get(5));

}

}

**Output**

10

-1

30

-1

40

50