**Least Recently Used(LRU)**

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

Caching is a technique used in computing to store frequently accessed data in a temporary storage layer to reduce the time needed for future retrievals. By keeping a copy of the most relevant and recently used data closer to the processing unit, caching significantly enhances performance, reduces latency, and decreases the load on the main storage or database.

**Need for Caching:**

* **Improved Performance**: Reduces data access time.
* **Lower Latency**: Speeds up response time for users.
* **Reduced Load on Backend**: Minimizes the number of requests to databases or external services.
* **Efficient Resource Utilization**: Saves computational power and network bandwidth.

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

In-memory caching stores data in RAM, making access significantly faster compared to disk-based storage.

**Redis**

* Redis (Remote Dictionary Server) is an open-source, in-memory key-value store.
* It supports data structures like strings, hashes, lists, sets, and sorted sets.
* Provides persistence options and replication features.
* Commonly used for caching, message brokering, and real-time analytics.

**Memcached**

* Memcached is an open-source, high-performance distributed memory object caching system.
* Designed to speed up dynamic web applications by reducing database load.
* Uses a simple key-value storage model.
* Faster than Redis for simple caching operations but lacks advanced data structures.

**3. Cache Memory in Computer Organization**

Cache memory is a small, high-speed memory unit located close to the CPU. It stores copies of frequently used instructions and data from main memory.

**Levels of Cache Memory:**

* **L1 Cache**: Smallest and fastest, integrated within the CPU.
* **L2 Cache**: Slightly larger, located on or near the CPU.
* **L3 Cache**: Shared among multiple cores in multi-core processors.

**Benefits of Cache Memory:**

* Reduces access time for data.
* Increases overall system performance.
* Minimizes delays caused by slower main memory.

**4. Different Cache Replacement Strategies**

Cache replacement policies decide which cache entry to remove when the cache is full.

**Common Strategies:**

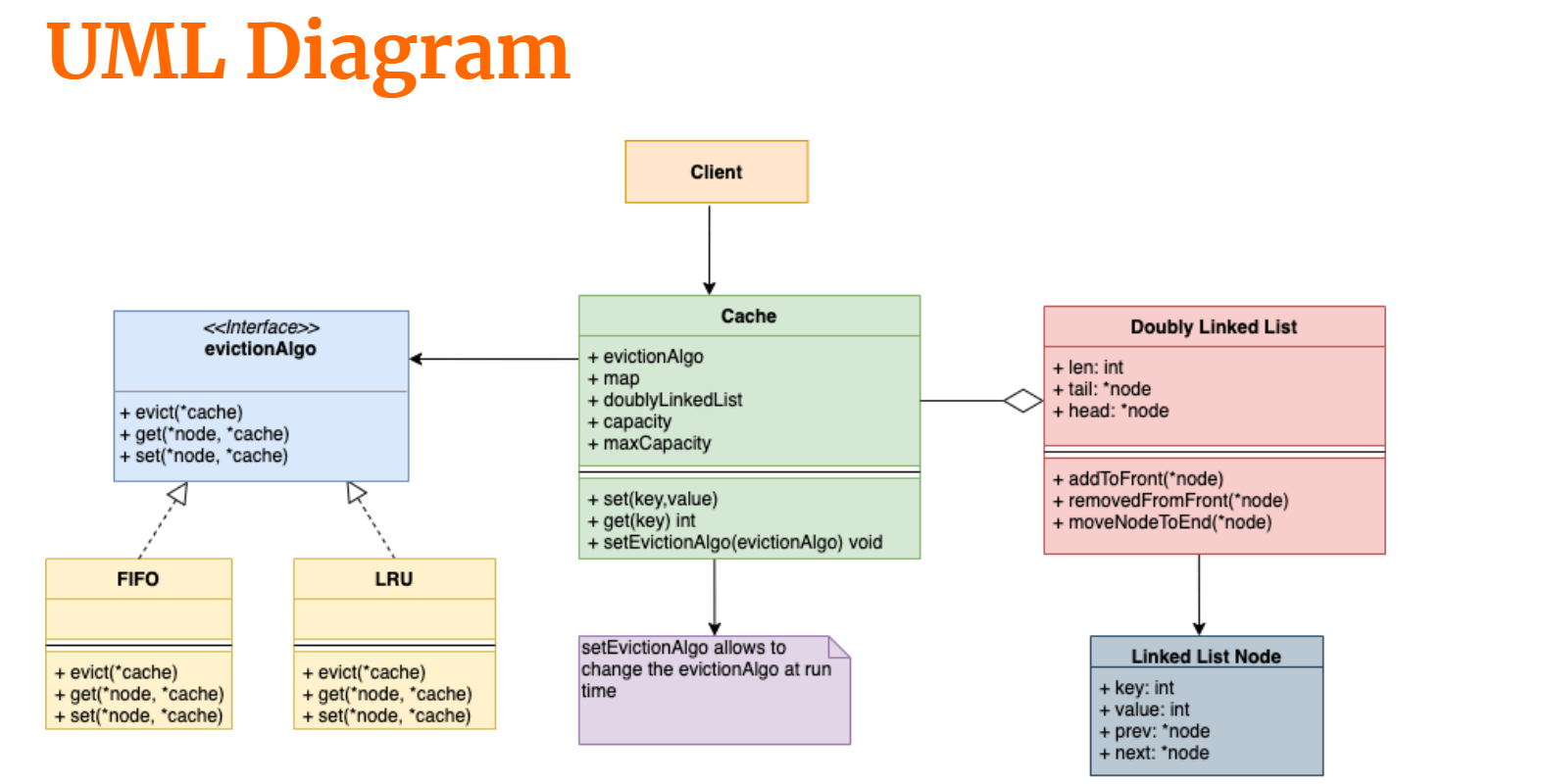
* **Least Recently Used (LRU)**: Removes the least recently accessed item.
* **Least Frequently Used (LFU)**: Removes the least frequently used item.
* **First-In-First-Out (FIFO)**: Removes the oldest item.
* **Random Replacement (RR)**: Removes a random item.

**5. Designing an LRU Cache using a Doubly Linked List and HashMap**

LRU (Least Recently Used) Cache keeps track of the recently accessed items using a combination of:

1. **HashMap**: Provides O(1) access to cached data.
2. **Doubly Linked List**: Maintains the order of usage.

**6. UML Diagram for LRU Cache**

****

**7. Code Implementation with Explanation**

import java.util.\*;

class LRUCache {

    CDLL ll;

    int capacity;

    int size;

    Map<Integer, CDLLNode> mp;

    public LRUCache(int cap) {

        ll = new CDLL();

        this.capacity = cap;

        this.size = 0;

        mp = new HashMap<>();

    }

    int get(int k) {

        if (mp.containsKey(k)) {

            CDLLNode node = mp.get(k);

            ll.moveToFront(node);

            return node.val;

        } else {

            return -1; // Key not found

        }

    }

    void put(int k, int v) {

        if (mp.containsKey(k)) { // Update existing key

            CDLLNode node = mp.get(k);

            node.val = v;

            ll.moveToFront(node);

        } else { // Insert new key-value pair

            if (size < capacity) {

                CDLLNode nn = ll.insAtBegin(k, v);

                mp.put(k, nn);

                size++;

            } else { // Evict least recently used (LRU)

                int delKey = ll.delLast();

                mp.remove(delKey);

                CDLLNode nn = ll.insAtBegin(k, v);

                mp.put(k, nn);

            }

        }

    }

}

class CDLLNode {

    int key, val;

    CDLLNode prev, next;

    public CDLLNode(int k, int v) {

        this.key = k;

        this.val = v;

        this.prev = this.next = null;

    }

}

class CDLL {

    CDLLNode head;

    public CDLL() {

        head = null;

    }

    CDLLNode insAtBegin(int k, int v) {

        CDLLNode nn = new CDLLNode(k, v);

        nn.next = nn;

        nn.prev = nn;

        if (head == null) {

            head = nn;

            return nn;

        }

        CDLLNode last = head.prev;

        nn.next = head;

        head.prev = nn;

        last.next = nn;

        nn.prev = last;

        head = nn;

        return nn;

    }

    int delLast() {

        if (head == null) return -1;

        CDLLNode last = head.prev;

        int ret = last.key;

        if (last == head) {

            head = null;

            return ret;

        }

        last.prev.next = head;

        head.prev = last.prev;

        return ret;

    }

    void moveToFront(CDLLNode nodeToMove) {

        if (nodeToMove == head) return;

        nodeToMove.prev.next = nodeToMove.next;

        nodeToMove.next.prev = nodeToMove.prev;

        CDLLNode last = head.prev;

        nodeToMove.next = head;

        head.prev = nodeToMove;

        last.next = nodeToMove;

        nodeToMove.prev = last;

        head = nodeToMove;

    }

}

// Main class to take user input and interact with LRU Cache

public class Main {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.print("Enter cache capacity: ");

        int capacity = sc.nextInt();

        LRUCache cache = new LRUCache(capacity);

        while (true) {

            System.out.println("\nChoose operation: 1. Get  2. Put  3. Exit");

            int choice = sc.nextInt();

            if (choice == 1) { // Get operation

                System.out.print("Enter key to fetch: ");

                int key = sc.nextInt();

                int result = cache.get(key);

                System.out.println("Value: " + (result == -1 ? "Key not found" : result));

            }

            else if (choice == 2) { // Put operation

                System.out.print("Enter key: ");

                int key = sc.nextInt();

                System.out.print("Enter value: ");

                int value = sc.nextInt();

                cache.put(key, value);

                System.out.println("Inserted: (" + key + ", " + value + ")");

            }

            else if (choice == 3) { // Exit

                System.out.println("Exiting...");

                break;

            }

            else {

                System.out.println("Invalid choice! Try again.");

            }

        }

        sc.close();

    }

}

**Explanation:**

The **Least Recently Used (LRU) Cache** is a data structure that stores a fixed number of key-value pairs and removes the least recently used item when the cache reaches its capacity. This implementation achieves **O(1) time complexity** for both get() and put() operations using:

1. **A HashMap (mp)** – To quickly access key-value pairs.
2. **A Circular Doubly Linked List (CDLL)** – To efficiently track and update the usage order.

**Components and Their Roles**

**1. LRUCache (Main Cache Class)**

* Maintains the cache's **capacity** and **current size**.
* Uses a **HashMap** to store key-to-node mappings.
* Uses a **Circular Doubly Linked List** to track the order of access.
* Implements two main operations:
  + **get(k)**: Retrieves the value of the key if present, moves it to the front (most recently used).
  + **put(k, v)**: Inserts a new key-value pair or updates an existing one. If full, removes the least recently used item before inserting a new one.

**2. CDLL (Circular Doubly Linked List)**

* Stores **nodes** containing a key-value pair.
* Maintains links between nodes using **previous (prev) and next (next) pointers**.
* Supports operations for:
  + **Insertion at the front** (newly used items move here).
  + **Deletion of the least recently used item** (removes the last node when the cache is full).
  + **Moving an accessed node to the front** (for get() and put() updates).

**Working of the LRU Cache**

**1. Adding a New Key (put(k, v))**

* If the key already exists, update its value and move it to the front.
* If the key doesn’t exist:
  + If space is available, insert it at the front.
  + If the cache is full, remove the least recently used (last) node and insert the new key.

**2. Fetching a Key (get(k))**

* If the key exists, return its value and move it to the front (most recently used).
* If the key doesn’t exist, return -1.

**Example Execution**

**Case 1: Inserting Items**

Cache capacity = 3  
**Operations:**

1. put(1, 10) → (Cache: {1=10})
2. put(2, 20) → (Cache: {1=10, 2=20})
3. put(3, 30) → (Cache: {1=10, 2=20, 3=30})

**Case 2: Accessing an Item**

* get(2) → Moves 2=20 to the front (most recently used).  
  (New order: {2=20, 1=10, 3=30})

**Case 3: Cache Full – Adding a New Item**

* put(4, 40) → Cache full, remove least recently used (key 3).  
  (New Cache: {2=20, 1=10, 4=40})