**What is Caching?**

Caching is a technique used to store frequently accessed data temporarily so that it can be quickly retrieved without having to recompute or fetch it from the original source every time. It helps in improving performance by reducing access time and computational overhead.

**Need for Caching:**

->Reduces the time needed to access frequently used data.

-> Helps in lowering the number of database queries or API calls.

-> Saves CPU and network bandwidth by avoiding redundant operations.

-> Improves application responsiveness by delivering content faster.

->Helps handle high traffic by reducing pressure on servers.

**Memory Caching:**

**Redis**

->Redis is an open-source, in-memory data structure store, used as a database, cache, and message broker.

->It supports various data structures such as strings, hashes, lists, sets, and sorted sets.

->Redis offers advanced features like transactions, pub/sub messaging, and Lua scripting.

->It is known for its high performance, scalability, and flexibility.

->Redis can be used for caching, session storage, real-time analytics, and more.

**Memcached**

->Memcached is a high-performance, distributed memory object caching system designed to speed up dynamic web applications by alleviating database load.

->It is simpler than Redis and primarily supports key-value storage.

->Memcached is known for its simplicity, ease of deployment, and efficiency.

->It is commonly used for caching database query results and page rendering.

->Memcached does not support advanced data structures or persistence, unlike Redis.

**Cache memory in computer organization:**

Cache memory is a small, high-speed storage area in a computer. The cache is a smaller and faster memory that stores copies of the data from frequently used main memory locations. There are various independent caches in a CPU, which store instructions and data.

* The most important use of cache memory is that it is used to reduce the average time to access data from the main memory.
* The concept of cache works because there exists locality of reference (the same items or nearby items are more likely to be accessed next) in processes.

**Levels of Memory**

* Level 1 or Register: It is a type of memory in which data is stored and accepted that are immediately stored in the CPU. The most commonly used register is Accumulator, Program counter , Address Register, etc.
* Level 2 or Cache memory: It is the fastest memory that has faster access time where data is temporarily stored for faster access.
* Level 3 or Main Memory: It is the memory on which the computer works currently. It is small in size and once power is off data no longer stays in this memory.
* Level 4 or Secondary Memory: It is external memory that is not as fast as the main memory but data stays permanently in this memory.

**Cache replacement Strategis:**

Cache replacement strategies determine which cache line should be evicted when new data needs to be loaded into the cache. Here are some common cache replacement strategies:

**1. Least Recently Used (LRU)**

* **Concept**: Evicts the cache line that has not been accessed for the longest time.
* **Pros**: Often effective because it leverages temporal locality.
* **Cons**: Requires maintaining a list of usage, which can be complex and resource-intensive.

**2. First In First Out (FIFO)**

* **Concept**: Evicts the cache line that was loaded first.
* **Pros**: Simple to implement.
* **Cons**: Does not consider how frequently or recently data is accessed, which may result in suboptimal performance.

**3. Least Frequently Used (LFU)**

* **Concept**: Evicts the cache line that has been accessed the least number of times.
* **Pros**: Effective when access frequency is a good indicator of future use.
* **Cons**: Requires tracking access counts, which can be resource-intensive.

**4. Random Replacement**

* **Concept**: Evicts a randomly selected cache line.
* **Pros**: Simple to implement and avoids systematic performance degradation.
* **Cons**: May not always make the best eviction choice compared to other algorithms.

**5. Most Recently Used (MRU)**

* **Concept**: Evicts the cache line that was most recently accessed.
* **Pros**: Can be effective in specific scenarios where recently accessed data is less likely to be reused.
* **Cons**: Generally not as effective as LRU in typical access patterns.

**6. Adaptive Replacement Cache (ARC)**

* **Concept**: Balances between recency (LRU) and frequency (LFU) by dynamically adjusting the importance of each factor.
* **Pros**: Adapts to different access patterns, often providing superior performance.
* **Cons**: More complex to implement.

**LRU Cache Using Doubly Linked List**:  
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;

}

}

void put(int k, int v){

if(mp.containsKey(k)){ // update

CDLLNode node = mp.get(k);

node.val = v;

ll.moveToFront(node);

} else { // insert

if(size < capacity){

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

mp.put(k, nn);

size++;

} else { // evict 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;

}

void printLL(){

if(head == null) return;

System.out.print(head.key + " ");

CDLLNode curr = head.next;

while(curr != head){

System.out.print(curr.key + " ");

curr = curr.next;

}

System.out.println();

}

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; // nothing to do

nodeToMove.prev.next = nodeToMove.next;

nodeToMove.next.prev = nodeToMove.prev;

CDLLNode last = head.prev;

nodeToMove.next = head; head.prev = nodeToMove;

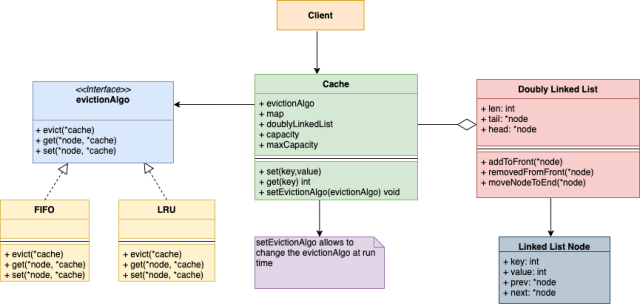
last.next = nodeToMove; nodeToMove.prev = last;

head = nodeToMove;

}

}

**UML DIAGRAM**



**Explaination:**

// Main class for the LRU Cache

class LRUCache {

CDLL ll; // Doubly linked list to store the cache elements

int capacity; // Maximum capacity of the cache

int size; // Current size of the cache

Map<Integer, CDLLNode> mp; // HashMap to store key and node references

// Constructor to initialize the cache with a given capacity

public LRUCache(int cap){

ll = new CDLL();

this.capacity = cap;

this.size = 0;

mp = new HashMap<>();

}

// Method to get the value of a key from the cache

int get(int k){

if(mp.containsKey(k)){ // Check if key is in the cache

CDLLNode node = mp.get(k);

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

return node.val; // Return the value of the node

} else {

return -1; // Return -1 if the key is not found

}

}

// Method to put a key-value pair into the cache

void put(int k, int v){

if(mp.containsKey(k)){ // If the key is already in the cache (update)

CDLLNode node = mp.get(k);

node.val = v; // Update the value of the node

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

} else { // If the key is not in the cache (insert)

if(size < capacity){ // If there is space in the cache

CDLLNode nn = ll.insAtBegin(k, v); // Insert the new node at the beginning

mp.put(k, nn); // Add the new node to the HashMap

size++; // Increase the size of the cache

} else { // If the cache is full (evict the least recently used)

int delKey = ll.delLast(); // Remove the last node (least recently used)

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

CDLLNode nn = ll.insAtBegin(k, v); // Insert the new node at the beginning

mp.put(k, nn); // Add the new node to the HashMap

}

}

}

}

// Class representing a node in the doubly linked list

class CDLLNode {

int key, val; // Key and value of the node

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

// Constructor to initialize a node with a key and value

public CDLLNode(int k, int v){

this.key = k;

this.val = v;

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

}

}

// Class representing the circular doubly linked list

class CDLL {

CDLLNode head; // Pointer to the head of the linked list

// Constructor to initialize the linked list

public CDLL(){

head = null;

}

// Method to insert a node at the beginning of the linked list

CDLLNode insAtBegin(int k, int v){

CDLLNode nn = new CDLLNode(k, v); // Create a new node

nn.next = nn; nn.prev = nn; // Set the new node to point to itself

if(head == null) { head = nn; return nn; } // If the list is empty, set the head to the new node

CDLLNode last = head.prev; // Get the last node in the list

nn.next = head; head.prev = nn; // Insert the new node at the beginning

last.next = nn; nn.prev = last; // Update the pointers of the last node and the new node

head = nn; return nn; // Update the head and return the new node

}

// Method to print the linked list (for debugging)

void printLL(){

if(head == null) return;

System.out.print(head.key + " ");

CDLLNode curr = head.next;

while(curr != head){

System.out.print(curr.key + " ");

curr = curr.next;

}

System.out.println();

}

// Method to delete the last node in the linked list (least recently used)

int delLast(){

if(head == null) return -1; // If the list is empty, return -1

CDLLNode last = head.prev; // Get the last node in the list

int ret = last.key; // Get the key of the last node

if(last == head) { head = null; return ret; } // If there is only one node, set the head to null and return the key

last.prev.next = head; head.prev = last.prev; // Update the pointers to remove the last node

return ret; // Return the key of the removed node

}

// Method to move a node to the front of the linked list

void moveToFront(CDLLNode nodeToMove){

if(nodeToMove == head) return; // If the node is already at the front, do nothing

nodeToMove.prev.next = nodeToMove.next; // Remove the node from its current position

nodeToMove.next.prev = nodeToMove.prev;

CDLLNode last = head.prev; // Get the last node in the list

nodeToMove.next = head; head.prev = nodeToMove; // Insert the node at the front

last.next = nodeToMove; nodeToMove.prev = last; // Update the pointers of the last node and the node to move

head = nodeToMove; // Update the head

}

}