LRU Cache - Complete Tutorial - GeeksforGeeks

Source: https://www.geeksforgeeks.org/lru-cache-implementation/

Courses Tutorials Practice Jobs System Design Tutorial HLD LLD Functional and Non Functional Life Cycle Design Patterns UML Diagrams System Design Interview Guide Scalability Databases DSA Software Engineering Technical Scripter 2026 Explore What is System Design System Design Introduction - LLD & HLD System Design Life Cycle (SDLC) What are the components of System Design? Goals and Objectives of System Design Why is it Important to Learn System Design? Important Key Concepts and Terminologies – Learn System Design Advantages of System Design System Design Fundamentals Analysis of Monolithic and Distributed Systems - Learn System Design Requirements Gathering in System Design Differences between System Analysis and System Design Horizontal and Vertical Scaling | System Design Capacity Estimation in Systems Design How to Answer a System Design Interview Problem/Question? Functional and Non Functional Requirements Web Server, Proxies and their role in Designing Systems Scalability in System Design Scalability in System Design Choosing the Right Scalability Approach Primary Scalability Bottlenecks in System Design Databases in Designing Systems Complete Guide to Database Design - System Design SQL vs. NoSQL - Which Database to Choose in System Design? File and Database Storage Systems in System Design Block, Object, and File Storage in System Design Database Sharding - System Design Database Replication in System Design High Level Design(HLD) Introduction to High level Design Availability in System Design Consistency in System Design Reliability in System Design CAP Theorem in System Design What is API Gateway? What is Content Delivery Network(CDN) in System Design Introduction to Load Balancer Caching - System Design Concept Communication Protocols in System Design Activity Diagrams - Unified Modeling Language (UML) Message Queues - System Design Low Level Design(LLD) What is Low Level Design or LLD? Authentication vs Authorization in LLD - System Design Performance Optimization Techniques for System Design Object-Oriented Analysis and Design(OOAD) Data Structures and Algorithms for System Design Containerization Architecture in System Design Modularity and Interfaces In System Design Unified Modeling Language (UML) Diagrams Data Partitioning Techniques in System Design How to Prepare for Low-Level Design Interviews? Essential Security Measures in System Design Design Patterns Design Patterns Tutorial Creational Design Patterns Structural Design Patterns Behavioral Design Patterns Design Patterns Cheat Sheet - When to Use Which Design Pattern? Interview Guide for System Design Cracking the System Design Interview Round System Design Interview Questions and Answers 5 Common System Design Concepts for Interview Preparation 5 Tips to Crack Low-Level System Design Interviews System Design Interview Questions & Answers Most Commonly Asked System Design Interview Questions Design Dropbox - A System Design Interview Question Designing Twitter - A System Design Interview Question System Design Netflix | A Complete Architecture System Design of Uber App | Uber System Architecture Design BookMyShow - A System Design Interview Question Designing Facebook Messenger | System Design Interview Complete Roadmap to Learn System Design for Beginners Guide to System Design for Freshers How Disney+ Hotstar Managed (5 Cr)+ Live Viewers During India's T20 World Cup Win[2024] System Design Live 90% Refund LRU Cache - Complete Tutorial Last Updated : 23 Jul, 2025 What is LRU Cache? Cache replacement algorithms are efficiently designed to replace the cache when the space is full. The Least Recently Used (LRU) is one of those algorithms. As the name suggests when the cache memory is full, LRU picks the data that is least recently used and removes it in order to make space for the new data. The priority of the data in the cache changes according to the need of that data i.e. if some data is fetched or updated recently then the priority of that data would be changed and assigned to the highest priority , and the priority of the data decreases if it remains unused operations after operations. Operations on LRU Cache: LRUCache (Capacity c): Initialize LRU cache with positive size capacity c. get (key) : Returns the value of key ' k' if it is present in the cache otherwise it returns -1. Also updates the priority of data in the LRU cache. put (key, value): Update the value of the key if that key exists, Otherwise, add a key-value pair to the cache. If the number of keys exceeds the capacity of the LRU cache then dismiss the least recently used key. Working of LRU Cache: Let's suppose we have an LRU cache of capacity 3, and we would like to perform the following operations: put (key=1, value=A) into the cache put (key=2, value=B) into the cache put (key=3, value=C) into the cache get (key=2) from the cache get (key=4) from the cache

put (key=4, value=D) into the cache put (key=3, value=E) into the cache get (key=4) from the cache put (key=1, value=A) into the cache The above operations are performed one after another as shown in the image below: Detailed Explanation of each operation: put (key 1, value A) : Since LRU cache has empty capacity=3, there is no need for any replacement and we put {1 : A} at the top i.e. {1 : A} has highest priority. put (key 2, value B) : Since LRU cache has empty capacity=2, again there is no need for any replacement, but now the {2 : B} has the highest priority and priority of {1 : A} decrease. put (key 3, value C) : Still there is 1 empty space vacant in the cache, therefore put {3 : C} without any replacement, notice now the cache is full and the current order of priority from highest to lowest is {3:C}, {2:B}, {1:A}. get (key 2) : Now, return value of key=2 during this operation, also since key=2 is used, now the new priority order is {2:B}, {3:C}, {1:A} get (key 4): Observe that key 4 is not present in the cache, we return '-1' for this operation. put (key 4, value D) : Observe that cache is FULL, now use LRU algorithm to determine which key is least recently used. Since {1:A} had the least priority, remove {1:A} from our cache and put {4:D} into the cache. Notice that the new priority order is {4:D}, {2:B}, {3:C} put (key 3, value E) : Since key=3 was already present in the cache having value=C, so this operation won't result in removal of any key, rather it will update the value of key=3 to ' E' . Now, the new order of priority will become {3:E}, {4:D}, {2:B} get (key 4) : Return the value of key=4. Now, new priority will become {4:D}, {3:E}, {2:B} put (key 1, value A) : Since our cache is FULL, so use our LRU algorithm to determine which key was least recently used, and since {2:B} had the least priority, remove {2:B} from our cache and put {1:A} into the cache. Now, the new priority order is {1:A}, {4:D}, {3:E}. How to design your own LRU Cache? Input: [LRUCache cache = new LRUCache (2) , put(1 ,1) , put(2 ,2) , get(1) , put(3 ,3) , get(2) , put(4 ,4) , get(1) , get(3) , get(4)] Output: [1 ,-1, -1, 3, 4] Explanation: The values mentioned in the output are the values returned by get operations . Initialize LRUCache class with capacity = 2. cache.put(1, 1) : (key, pair) = (1,1) inserted and has the highest priority. cache.put(2, 2) : (key , pair) = (2,2) inserted and has the highest priority. cache.get(1) : For key 1, value is 1, so 1 returned and (1,1) moved to the highest priority. cache.put(3, 3) : Since cache is full, remove least recently used that is (2,2), (3,3) inserted with the highest priority. cache.get(2): returns -1 (key 2 not found) cache.put(4, 4) : Since the cache is full, remove least recently used that is (1,1). (4,5) inserted with the highest priority. cache.get(1): return -1 (not found) cache.get(3): return 3 , (3,3) will moved to the highest priority. cache.get(4) : return 4 , (4,4) moved to the highest priority. Thoughts about Implementation Using Arrays, Hashing and/or Heap The idea is to implement using an array to store nodes, where each node holds a key-value pair. The primary operations, get and put, are performed with O(n) time complexity due to the need to search through the array. The size of the array will be equal to the given capacity of the cache. put(int key, int value) If the cache is full, find the node with the oldest timestamp (least recently used) and replace this node with the new key and value. else, simply add the new node to the end of the array with the timestamp of insertion. Time Complexity: O(n) (because you might have to search for the oldest node) get(int key) Search through the array for the node with the matching key. If found, update its timestamp and return its value , else return -1. Time Complexity: O(n) (because you might have to check every node) Can we make both operations in O(1) time? we can think of hashing. With hashing, we can insert, get and delete in O(1) time, but changing priorities would take linear time. We can think of using heap along with hashing for priorities. We can find and remove the least recently used (lowest priority) in O(Log n) time which is more than O(1) and changing priority in the heap would also be required. Please refer Design LRU Cache for details of all approaches. Efficient Solution - Using Doubly Linked List and Hashing The basic idea behind implementing an LRU (Least Recently Used) cache using a key-value pair approach is to manage element access and removal efficiently through a combination of a doubly linked list and a hash map. When adding a new key-value pair, insert it as a new node at the head of the doubly linked list. This ensures that the newly added key-value pair is marked as the most recently used. If the key is already present in the cache , get the corresponding node in the doubly linked list using hashmap , update its value and move it to the head of the list, and update its position in the hashmap also. This operation ensures that the accessed key-value pair is considered the most recently used. The priority of nodes in the doubly linked list is based on their distance from the head . Key-value pairs closer to the head are more recently used and thus have higher priority . Also, key-value pairs closer to the tail are considered less recently used and have lower priority. When the cache reaches its maximum capacity and a new key-value pair needs to be added, remove the node from hashmap as well as from the tail in the doubly linked list . Tail node represents the least recently used key-value pair and is removed to make space for the new entry. Please refer LRU cache implementation using Doubly Linked List and Hashing for details Complexity Analysis of the Efficient Solution Time Complexity: put() operation: O(1) i.e. time

required to insert or update new key-value pair is constant get() operation: O(1) i.e. time required to get the value of a key is constant Auxiliary Space: O(c) where c is the capacity of the Cache. Advantages of LRU cache: Accessing data is done in O(1) time Updating a key-value pair also takes O(1) time. Removing the least recently used item is performed in O(1) time. LRU plays a crucial role in page replacement strategies to minimize page faults . Disadvantages of LRU cache: It requires large cache size to increase efficiency. It requires additional Data Structure to be implemented. In LRU, error detection is difficult as compared to other algorithms. It has limited acceptability. Real-World Application of LRU Cache: In Database Systems for fast query results. In Operating Systems to minimize page faults. Text Editors and IDEs to store frequently used files or code snippets. Network routers and switches use LRU to increase the efficiency of computer network. In compiler optimizations. Text Prediction and autocompletion tools. Comment Article Tags: Article Tags: System Design Amazon Morgan Stanley Snapdeal MakeMyTrip STL cpp-unordered_map DSA Tutorials + 4 More