**Problem Name:**LFU Cache

**Topics:** Hash Table, Linked list, Design, Doubly linked list

**Companies:** Amazon, Microsoft, Google, Linkedin, Salesforce, Snapchat, Facebook, Bloomberg, Walmart, Arcesium.

**Level: Hard**

**Language:** C++

**Problem Statement:** Design and implement a data structure for a [Least Frequently Used (LFU)](https://en.wikipedia.org/wiki/Least_frequently_used) cache.

Implement the LFUCache class:

* LFUCache(int capacity) Initializes the object with the capacity of the data structure.
* int get(int key) Gets the value of the key if the key exists in the cache. Otherwise, returns -1.
* void put(int key, int value) Update the value of the key if present, or inserts the key if not already present. When the cache reaches its capacity, it should invalidate and remove the **least frequently used** key before inserting a new item. For this problem, when there is a **tie** (i.e., two or more keys with the same frequency), the **least recently used** key would be invalidated.

To determine the least frequently used key, a **use counter** is maintained for each key in the cache. The key with the smallest **use counter** is the least frequently used key.

When a key is first inserted into the cache, its **use counter** is set to 1 (due to the put operation). The **use counter** for a key in the cache is incremented either a get or put operation is called on it.

The functions get and put must each run in O(1) average time complexity.

**Input Format:**

**Output Format:** Print

**Constraints:**

* 0 <= capacity <= 104
* 0 <= key <= 105
* 0 <= value <= 109
* At most 2 \* 105 calls will be made to get and put.

**Examples:**

**Input**

["LFUCache", "put", "put", "get", "put", "get", "get", "put", "get", "get", "get"]

[[2], [1, 1], [2, 2], [1], [3, 3], [2], [3], [4, 4], [1], [3], [4]]

**Output**

[null, null, null, 1, null, -1, 3, null, -1, 3, 4]

**Explanation**

// cnt(x) = the use counter for key x

// cache=[] will show the last used order for tiebreakers (leftmost element is most recent)

LFUCache lfu = new LFUCache(2);

lfu.put(1, 1); // cache=[1,\_], cnt(1)=1

lfu.put(2, 2); // cache=[2,1], cnt(2)=1, cnt(1)=1

lfu.get(1); // return 1

// cache=[1,2], cnt(2)=1, cnt(1)=2

lfu.put(3, 3); // 2 is the LFU key because cnt(2)=1 is the smallest, invalidate 2.

  // cache=[3,1], cnt(3)=1, cnt(1)=2

lfu.get(2); // return -1 (not found)

lfu.get(3); // return 3

// cache=[3,1], cnt(3)=2, cnt(1)=2

lfu.put(4, 4); // Both 1 and 3 have the same cnt, but 1 is LRU, invalidate 1.

// cache=[4,3], cnt(4)=1, cnt(3)=2

lfu.get(1); // return -1 (not found)

lfu.get(3); // return 3

// cache=[3,4], cnt(4)=1, cnt(3)=3

lfu.get(4); // return 4

// cache=[4,3], cnt(4)=2, cnt(3)=3

**Brute force Solution:**

**Explanation:**

**Code:**

**Time Complexity**: O(n)

**Space Complexity:** O(height)

**Optimized Solution:**

**Explanation:**

* The idea is straightforward. We maintain a min-PriorityQueue with the least frequently used element at the top. However, when we access any existing element using get() or set(), its usage frequency should be increased by one, which forces us to change its position in the priority queue (re-heapify). Unfortunately, native STL PriorityQueue does not support this operation. Therefore, we need to create our own priority queue and modify it. I found the so-called Index Priority Queue is suitable for this. It maintains not only a priority queue but also a hash map, "indexMap", that maps the key of an element to its position (index) in the priority queue. Now, we can quickly access any element in the priority queue and re-heapify the PriorityQueue, when the usage frequency of an element changes.
* Another tricky point is that when we need to evict an element, but multiple elements have the same (minimum) usage frequency, we need to evict the least recently used (the oldest) element. To handle this, I maintain a time-stamp variable for each element in the LFU Cache, which indicates the latest time stamp when we access it. Therefore, we maintain the following invariant in the priority queue: **When two elements have the same usage frequency, the least recently used one will always be closer to root node**. When there are multiple least frequently used elements, we always retrieve the one closer to the root.

**Code:**

**Time Complexity**: O(n)

**Space Complexity:** O(height)