# 题目

请你为 最不经常使用（LFU）缓存算法设计并实现数据结构。

实现 LFUCache 类：

LFUCache(int capacity) - 用数据结构的容量 capacity 初始化对象

int get(int key) - 如果键 key 存在于缓存中，则获取键的值，否则返回-1。

void put(int key, int value) - 如果键 key 已存在，则变更其值；如果键不存在，请插入键值对。当缓存达到其容量 capacity 时，则应该在插入新项之前，移除最不经常使用的项。在此问题中，当存在平局（即两个或更多个键具有相同使用频率）时，应该去除 最久未使用 的键。

为了确定最不常使用的键，可以为缓存中的每个键维护一个 使用计数器 。使用计数最小的键是最久未使用的键。

当一个键首次插入到缓存中时，它的使用计数器被设置为 1 (由于 put 操作)。对缓存中的键执行 get 或 put 操作，使用计数器的值将会递增。

函数 get 和 put 必须以 O(1) 的平均时间复杂度运行。

示例：

输入：

["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]]

输出：

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

解释：

// cnt(x) = 键 x 的使用计数

// cache=[] 将显示最后一次使用的顺序（最左边的元素是最近的）

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); // 返回 1

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

lfu.put(3, 3); // 去除键 2 ，因为 cnt(2)=1 ，使用计数最小

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

lfu.get(2); // 返回 -1（未找到）

lfu.get(3); // 返回 3

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

lfu.put(4, 4); // 去除键 1 ，1 和 3 的 cnt 相同，但 1 最久未使用

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

lfu.get(1); // 返回 -1（未找到）

lfu.get(3); // 返回 3

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

lfu.get(4); // 返回 4

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

提示：

1 <= capacity <= 104

0 <= key <= 105

0 <= value <= 109

最多调用 2 \* 105 次 get 和 put 方法

# 分析

## 方法一：哈希表+平衡二叉树

## 方法二：双哈希表

// 缓存的节点信息

struct Node {

int key, val, freq;

Node(int \_key,int \_val,int \_freq): key(\_key), val(\_val), freq(\_freq){}

};

class LFUCache {

int minfreq, capacity;

unordered\_map<int, list<Node>::iterator> key\_table;

unordered\_map<int, list<Node>> freq\_table;

public:

LFUCache(int \_capacity) {

minfreq = 0;

capacity = \_capacity;

key\_table.clear();

freq\_table.clear();

}

int get(int key) {

if (capacity == 0) return -1;

auto it = key\_table.find(key);

if (it == key\_table.end()) return -1;

list<Node>::iterator node = it -> second;

int val = node -> val, freq = node -> freq;

freq\_table[freq].erase(node);

// 如果当前链表为空，我们需要在哈希表中删除，且更新minFreq

if (freq\_table[freq].size() == 0) {

freq\_table.erase(freq);

if (minfreq == freq) minfreq += 1;

}

// 插入到 freq + 1 中

freq\_table[freq + 1].push\_front(Node(key, val, freq + 1));

key\_table[key] = freq\_table[freq + 1].begin();

return val;

}

void put(int key, int value) {

if (capacity == 0) return;

auto it = key\_table.find(key);

if (it == key\_table.end()) {

// 缓存已满，需要进行删除操作

if (key\_table.size() == capacity) {

// 通过 minFreq 拿到 freq\_table[minFreq] 链表的末尾节点

auto it2 = freq\_table[minfreq].back();

key\_table.erase(it2.key);

freq\_table[minfreq].pop\_back();

if (freq\_table[minfreq].size() == 0) {

freq\_table.erase(minfreq);

}

}

freq\_table[1].push\_front(Node(key, value, 1));

key\_table[key] = freq\_table[1].begin();

minfreq = 1;

} else {

// 与 get 操作基本一致，除了需要更新缓存的值

list<Node>::iterator node = it -> second;

int freq = node -> freq;

freq\_table[freq].erase(node);

if (freq\_table[freq].size() == 0) {

freq\_table.erase(freq);

if (minfreq == freq) minfreq += 1;

}

freq\_table[freq + 1].push\_front(Node(key, value, freq + 1));

key\_table[key] = freq\_table[freq + 1].begin();

}

}

};

/\*\*

\* Your LFUCache object will be instantiated and called as such:

\* LFUCache\* obj = new LFUCache(capacity);

\* int param\_1 = obj->get(key);

\* obj->put(key,value);

\*/