# 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

思路：

我们定义两个哈希表，第一个freq\_table以频率freq为索引，每个索引存放一个双向链表，这个链表里存放所有使用频率为freq的缓存，缓存里存放三个信息，分别为键key，值value，以及使用频率freq。第二个key\_table以键值key为索引，每个索引存放对应缓存在freq\_table中链表里的内存地址，这样我们就能利用两个哈希表来使得两个操作的时间复杂度均为O(1)。同时需要记录一个当前缓存最少使用的频率minFreq，这是为了删除操作服务的。

对于get(key) 操作，我们能通过索引key在key\_table中找到缓存在freq\_table中的链表的内存地址，如果不存在直接返回-1，否则我们能获取到对应缓存的相关信息，这样我们就能知道缓存的键值还有使用频率，直接返回key对应的值即可。

但是我们注意到get操作后这个缓存的使用频率加一了，所以我们需要更新缓存在哈希表freq\_table中的位置。已知这个缓存的键key，值value，以及使用频率freq，那么该缓存应该存放到freq\_table中freq+1索引下的链表中。所以我们在当前链表中O(1)删除该缓存对应的节点，根据情况更新minFreq值，然后将其O(1)插入到freq + 1索引下的链表头完成更新。这其中的操作复杂度均为O(1)。你可能会疑惑更新的时候为什么是插入到链表头，这其实是为了保证缓存在当前链表中从链表头到链表尾的插入时间是有序的，为下面的删除操作服务。

对于put(key, value)操作，我们先通过索引key在key\_table中查看是否有对应的缓存，如果有的话，其实操作等价于get(key) 操作，唯一的区别就是我们需要将当前的缓存里的值更新为value。如果没有的话，相当于是新加入的缓存，如果缓存已经到达容量，需要先删除最近最少使用的缓存，再进行插入。

先考虑插入，由于是新插入的，所以缓存的使用频率一定是1，所以我们将缓存的信息插入到freq\_table中1索引下的列表头即可，同时更新key\_table[key]的信息，以及更新minFreq = 1。

那么剩下的就是删除操作了，由于我们实时维护了minFreq，所以我们能够知道freq\_table里目前最少使用频率的索引，同时因为我们保证了链表中从链表头到链表尾的插入时间是有序的，所以freq\_table[minFreq]的链表中链表尾的节点即为使用频率最小且插入时间最早的节点，我们删除它同时根据情况更新minFreq，整个时间复杂度均为O(1)。

代码：

// 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);

\*/

## 方法三：哈希表

设计思路：

1、维护以下信息：

key → (value, 使用频率 freq) 映射，快速查找缓存值和频率

freq → 双向链表（维护该频率下的所有 key 按使用时间的顺序）

minFreq记录当前缓存中最小的使用频率（用于快速定位淘汰目标）

2、数据结构

哈希表 keyToValFreq：key 映射到 {value, freq}

哈希表 freqToList：freq 映射到双向链表（用来存储所有 freq 相同的 key，顺序表示最近使用情况）

哈希表 keyToIter：key 映射到它在 freqToList\[freq] 双向链表中的迭代器，方便 O(1) 删除

minFreq：当前缓存中最小的频率，方便快速找到淘汰的频率链表

3、操作流程

1）get(key)

- 如果 key 不存在，返回 -1

- 否则获取该 key 的 value 和 freq

- 在 freqToList\[freq] 中删除该 key，freq 增加 1

- 将 key 插入到 freqToList\[freq+1] 头部

- 更新 keyToFreq 和 keyToIter

- 如果 freqToList\[minFreq] 为空，则 minFreq++

- 返回 value

2）put(key, value)

- 如果容量为0，直接返回

- 如果 key 存在，更新 value，调用 `get(key)` 更新 freq（即调用 `get` 后再更新 value）

- 否则：

如果缓存满了，找到 freqToList\[minFreq] 的尾部（最久未使用），删除该节点对应 key

插入新 key，freq = 1，更新 minFreq = 1

更新相关哈希映射

代码：

#include <unordered\_map>

#include <list>

using namespace std;

class LFUCache {

private:

int capacity;

int minFreq;

// key -> (value, freq)

unordered\_map<int, pair<int,int>> keyToValFreq;

// freq -> keys list (最近使用放前面)

unordered\_map<int, list<int>> freqToList;

// key -> 在 freqToList[freq] 中的迭代器

unordered\_map<int, list<int>::iterator> keyToIter;

public:

LFUCache(int capacity) {

this->capacity = capacity;

minFreq = 0;

}

int get(int key) {

if (keyToValFreq.find(key) == keyToValFreq.end()) return -1;

int val = keyToValFreq[key].first;

int freq = keyToValFreq[key].second;

// 从旧频率链表删除

freqToList[freq].erase(keyToIter[key]);

// 更新 freq

keyToValFreq[key].second = freq + 1;

// 加入新频率链表头部

freqToList[freq+1].push\_front(key);

keyToIter[key] = freqToList[freq+1].begin();

// 如果旧频率链表空了，且是最小频率，更新 minFreq

if (freqToList[freq].empty()) {

freqToList.erase(freq);

if (minFreq == freq)

minFreq++;

}

return val;

}

void put(int key, int value) {

if (capacity == 0) return;

if (keyToValFreq.find(key) != keyToValFreq.end()) {

// key 存在，更新值和频率

keyToValFreq[key].first = value;

get(key); // 调用 get 触发频率更新

return;

}

if (keyToValFreq.size() == capacity) {

// 淘汰 freq 最小链表尾部（最久未使用）

int keyToRemove = freqToList[minFreq].back();

freqToList[minFreq].pop\_back();

if (freqToList[minFreq].empty()) {

freqToList.erase(minFreq);

}

keyToValFreq.erase(keyToRemove);

keyToIter.erase(keyToRemove);

}

// 插入新元素，频率为 1

keyToValFreq[key] = {value, 1};

freqToList[1].push\_front(key);

keyToIter[key] = freqToList[1].begin();

minFreq = 1;

}

};

**复杂度分析：**

时间复杂度：get和 put操作均为 O(1) 平均

空间复杂度：O(capacity)