# 题目

运用你所掌握的数据结构，设计和实现一个  LRU (最近最少使用) 缓存机制 。

实现 LRUCache 类：

LRUCache(int capacity) 以正整数作为容量 capacity 初始化 LRU 缓存

int get(int key) 如果关键字 key 存在于缓存中，则返回关键字的值，否则返回 -1 。

void put(int key, int value) 如果关键字已经存在，则变更其数据值；如果关键字不存在，则插入该组「关键字-值」。当缓存容量达到上限时，它应该在写入新数据之前删除最久未使用的数据值，从而为新的数据值留出空间。

进阶：你是否可以在 O(1) 时间复杂度内完成这两种操作？

示例：

输入

["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]

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

输出

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

解释

LRUCache lRUCache = new LRUCache(2);

lRUCache.put(1, 1); // 缓存是 {1=1}

lRUCache.put(2, 2); // 缓存是 {1=1, 2=2}

lRUCache.get(1); // 返回 1

lRUCache.put(3, 3); // 该操作会使得关键字 2 作废，缓存是 {1=1, 3=3}

lRUCache.get(2); // 返回 -1 (未找到)

lRUCache.put(4, 4); // 该操作会使得关键字 1 作废，缓存是 {4=4, 3=3}

lRUCache.get(1); // 返回 -1 (未找到)

lRUCache.get(3); // 返回 3

lRUCache.get(4); // 返回 4

提示：

1 <= capacity <= 3000

0 <= key <= 10000

0 <= value <= 105

最多调用2 \*105次get和put

# 分析

## 方法一：哈希表+双向链表

LRU缓存机制可以通过哈希表辅以双向链表实现，我们用一个哈希表和一个双向链表维护所有在缓存中的键值对。

双向链表按照被使用的顺序存储了这些键值对，靠近头部的键值对是最近使用的，而靠近尾部的键值对是最久未使用的。

哈希表即为普通的哈希映射（HashMap），通过缓存数据的键映射到其在双向链表中的位置。

这样以来，我们首先使用哈希表进行定位，找出缓存项在双向链表中的位置，随后将其移动到双向链表的头部，即可在O(1)的时间内完成get或者put操作。具体的方法如下：

1、对于get操作，首先判断key是否存在：

如果key不存在，则返回-1；

如果key存在，则key对应的节点是最近被使用的节点。通过哈希表定位到该节点在双向链表中的位置，并将其移动到双向链表的头部，最后返回该节点的值。

2、对于put操作，首先判断key是否存在：

如果key不存在，使用key和value创建一个新的节点，在双向链表的头部添加该节点，并将key和该节点添加进哈希表中。然后判断双向链表的节点数是否超出容量，如果超出容量，则删除双向链表的尾部节点，并删除哈希表中对应的项；

如果key存在，则与get操作类似，先通过哈希表定位，再将对应的节点的值更新为value，并将该节点移到双向链表的头部。

上述各项操作中，访问哈希表的时间复杂度为O(1)，在双向链表的头部添加节点、在双向链表的尾部删除节点的复杂度也为O(1)。而将一个节点移到双向链表的头部，可以分成「删除该节点」和「在双向链表的头部添加节点」两步操作，都可以在O(1)时间内完成。

小贴士：

在双向链表的实现中，使用一个伪头部（dummy head）和伪尾部（dummy tail）标记界限，这样在添加节点和删除节点的时候就不需要检查相邻的节点是否存在。

**代码：**

struct DLinkedNode {

int key, value;

DLinkedNode\* prev;

DLinkedNode\* next;

DLinkedNode(): key(0), value(0), prev(nullptr), next(nullptr) {}

DLinkedNode(int \_key, int \_value): key(\_key), value(\_value), prev(nullptr), next(nullptr) {}

};

class LRUCache {

private:

unordered\_map<int, DLinkedNode\*> cache;

DLinkedNode\* head;

DLinkedNode\* tail;

int size;

int capacity;

public:

LRUCache(int \_capacity): capacity(\_capacity), size(0) {

// 使用伪头部和伪尾部节点

head = new DLinkedNode();

tail = new DLinkedNode();

head->next = tail;

tail->prev = head;

}

int get(int key) {

if (!cache.count(key)) {

return -1;

}

// 如果 key 存在，先通过哈希表定位，再移到头部

DLinkedNode\* node = cache[key];

moveToHead(node);

return node->value;

}

void put(int key, int value) {

if (!cache.count(key)) {

// 如果 key 不存在，创建一个新的节点

DLinkedNode\* node = new DLinkedNode(key, value);

// 添加进哈希表

cache[key] = node;

// 添加至双向链表的头部

addToHead(node);

++size;

if (size > capacity) {

// 如果超出容量，删除双向链表的尾部节点

DLinkedNode\* removed = removeTail();

// 删除哈希表中对应的项

cache.erase(removed->key);

// 防止内存泄漏

delete removed;

--size;

}

}

else {

// 如果 key 存在，先通过哈希表定位，再修改 value，并移到头部

DLinkedNode\* node = cache[key];

node->value = value;

moveToHead(node);

}

}

void addToHead(DLinkedNode\* node) {

node->prev = head;

node->next = head->next;

head->next->prev = node;

head->next = node;

}

void removeNode(DLinkedNode\* node) {

node->prev->next = node->next;

node->next->prev = node->prev;

}

void moveToHead(DLinkedNode\* node) {

removeNode(node);

addToHead(node);

}

DLinkedNode\* removeTail() {

DLinkedNode\* node = tail->prev;

removeNode(node);

return node;

}

};

**复杂度分析：**

时间复杂度：对于put和get都是O(1)。

空间复杂度：O(capacity)，因为哈希表和双向链表最多存储capacity+1个元素。