0525

LRU缓存机制

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。

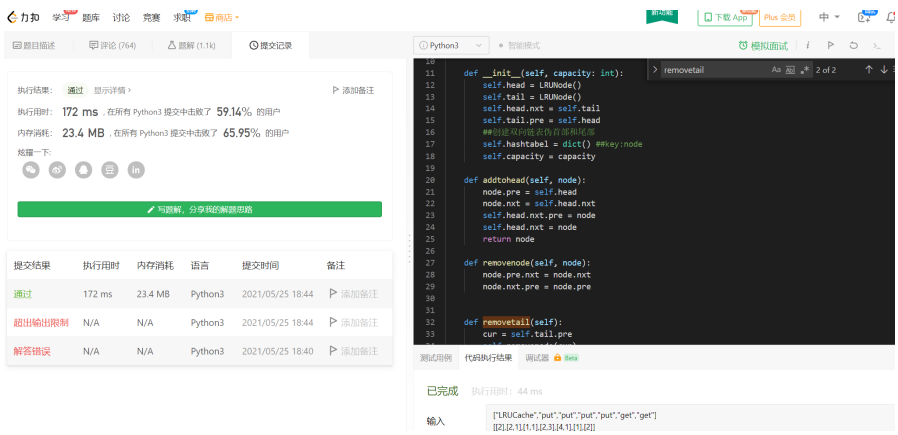
进阶：你是否可以在 O(1) 时间复杂度内完成这两种操作？

demo

```
class LRUNode:
    def __init__(self,key=0,value=0):
        self.key = key
        self.value = value
        self.pre = None
        self.nxt = None


class LRUCache:

    def __init__(self, capacity: int):
        self.head = LRUNode()
        self.tail = LRUNode()
        self.head.nxt = self.tail
        self.tail.pre = self.head
        ##创建双向链表伪首部和尾部
        self.hashtabel = dict() ##key:node
        self.capacity = capacity

    def addtohead(self, node):
        node.pre = self.head
        node.nxt = self.head.nxt
        self.head.nxt.pre = node
        self.head.nxt = node
        return node

    def removenode(self, node):
        node.pre.nxt = node.nxt
        node.nxt.pre = node.pre
```

```
        node.nxt.pre = node.pre


    def removetail(self):
        cur = self.tail.pre
        self.removenode(cur)
        return cur.key

    def movetop(self,cur):
        self.removenode(cur)
        self.addtohead(cur)




    def get(self, key: int) -> int:
        #判断哈希表中是否有值
        if key not in self.hashtabel:
            return -1 ##没有返回-1
        cur = self.hashtabel[key]
        ###移动到双向列表首部
        self.movetop(cur)
        return cur.value



    def put(self, key: int, value: int) -> None:
        ##先判断关键字是否存在
        #存在
        if key in self.hashtabel:
            self.hashtabel[key].value = value
            self.movetop(self.hashtabel[key])
        else:
            ##添加到首部
            cur = LRUNode(key,value)
            cur = self.addtohead(cur)
            self.hashtabel[key] = cur

            ##判断是否超出容量
            if len(self.hashtabel) > self.capacity:
                key = self.removetail()
                # print(self.hashtabel)
                # print(key)
                self.hashtabel.pop(key)
                # print(self.hashtabel)
```

```
#时间复杂度：对于 put 和 get 都是 O(1)O(1)。

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


# Your LRUCache object will be instantiated and called as such:
# obj = LRUCache(capacity)
# param_1 = obj.get(key)
# obj.put(key,value)
```



哈希表常见操作的时间复杂度是多少？遇到哈希冲突是如何解决的

哈希表 哈希表也叫散列表，通过把key映射到表中一个位置来访问记录以加快查找速度，这个映射函数叫做散列函数

- 哈希表 是使用 O(1)时间复杂度 进行数据的插入删除和查找，但是哈希表不保证表中数据的有序性，这样在哈希表中查找最大数据或者最小数据的时间是 O(N)

哈希冲突

由于hash算法被计算的数据是无限的，而计算后的结果范围有限，因此会存在不同数据经过计算得到的值相同，这就是哈希冲突

解决冲突的办法：

（1）线性探查法：冲突后，从发生冲突的单元开始探测，依次查看下一单元是否为空，如果到了最后一个单元还为空，那么再从表首进行判断，直到碰到了空闲的单元或者已经探查完所有单元。

冲突主要取决于：

（1）散列函数，一个好的散列函数的值应尽可能平均分布。

（2）处理冲突方法。

（3）负载因子的大小。太大不一定就好，而且浪费空间严重，负载因子和散列函数是联动的。

解决冲突的办法：

（1）线性探查法：冲突后，线性向前试探，找到最近的一个空位置。缺点是会出现堆积现象。存取时，可能不是同义词的词也位于探查序列，影响效率。

（2）双散列函数法：在位置d冲突后，再次使用另一个散列函数产生一个与散列表桶容量m互质的数c，依次试探（d+n*c）%m，使探查序列跳跃式分布。 （3）链地址法：将哈希值相同的元素构成一个链表，head放在散列表中。一般链表长度超过了8就转为红黑树，长度少于6个。
（4）再哈希法：构造多个哈希函数，当一个发生冲突时使用另一个计算，直到冲突不再产生，这种方法不易产生聚集，但是增加了计算时间