C语言实现LRU缓存

请你设计并实现一个满足 LRU (最近最少使用) 缓存 约束的数据结构。

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 key 已经存在，则变更其数据值 value ；如果不存在，则向缓存中插入该组 key−value 。如果插入操作导致关键字数量超过 capacity ，则应该 逐出 最久未使用的关键字。

函数 get 和 put 必须以 O(1) 的平均时间复杂度运行。

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

思路：1

1、一开始想的是用一个类似于数组栈来实现，栈中的的数据结构为一个key值和value值：

key	value

在这里插入图片描述

类似于这样，在put的时候，如果栈中元素已满例如存放的数据如下：

在这里插入图片描述

3	3
2	2
1	1
key	value

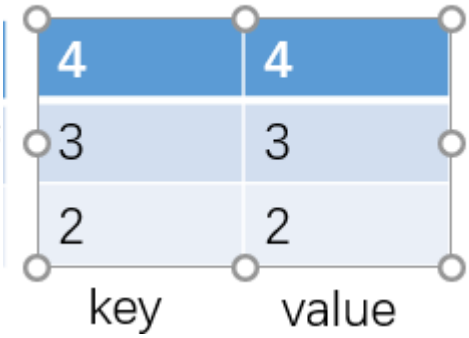
下一步要存放（4，4），数据元素已满，那么依次下移元素：

3	3
2	2
key	value

在这里插入图片描述

在插入元素：（4，4）

在这里插入图片描述



这样就能保证插入的元素在最顶部，如果元素已满，把最久未使用的元素删除。

在Get的时候，如果元素在内部，则把该元素置顶，其他元素依次下移:代码如下：

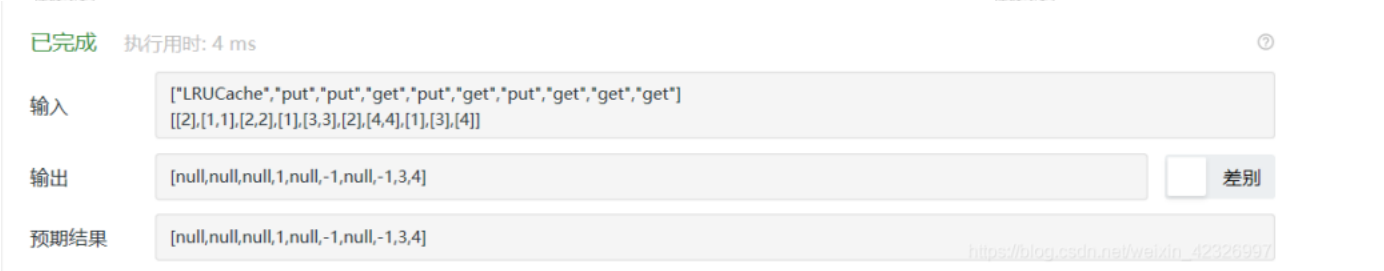
```
typedef struct{
        int key;
         int value;
 }Stack;
 typedef struct {
         Stack *S;
         int top;
         int Max_size;
 } LRUCache;

LRUCache* lRUCacheCreate(int capacity)
{
        LRUCache *obj = (LRUCache *)malloc(sizeof(LRUCache));
        obj->S = (Stack *)malloc(sizeof(Stack)*capacity);
        obj->top = -1;
        obj->Max_size=capacity;
        return obj;
}
int lRUCacheGet(LRUCache* obj, int key) {

        int i = 0;
        Stack temp;
        int j = 0;
        for(i;i<=obj->top;i++)
        {
                if(obj->S[i].key == key)
                {
                        temp = obj->S[i];
                        for( j = i;j<obj->top;j++)
                        {
                                obj->S[j]=obj->S[j+1];
                        }
                        obj->S[j] = temp;
                        return temp.value;
                }
        }
        return -1;
}
void lRUCachePut(LRUCache* obj, int key, int value) {
            int i =0;
            int j = 0;
```

```
        Stack temp;
        for(i;i<=obj->top;i++)
        {
            if(obj->S[i].key == key)
            {
                obj->S[i].value = value;
                temp = obj->S[i];
                for(j=i;j<obj->top;j++)
                {
                    obj->S[j]=obj->S[j+1];
                }
                obj->S[j] = temp;
                return ;
                }
            }
            if(i<obj->Max_size)
            {
                obj->S[i].key = key;
                obj->S[i].value = value;
                obj->top++;
                return ;
            }else{
                int temp_1 = i-1;
                i=0;
                for(i;i<obj->top;i++)
                {
                    obj->S[i] = obj->S[i+1];
                }
                obj->S[temp_1].key = key;
                obj->S[temp_1].value = value;
                return ;
            }
}
void lRUCacheFree(LRUCache* obj) {    free(obj);
}
```

这一种思路的时间复杂度很大；



思路：二

题目要求时间复杂度为O（1）；

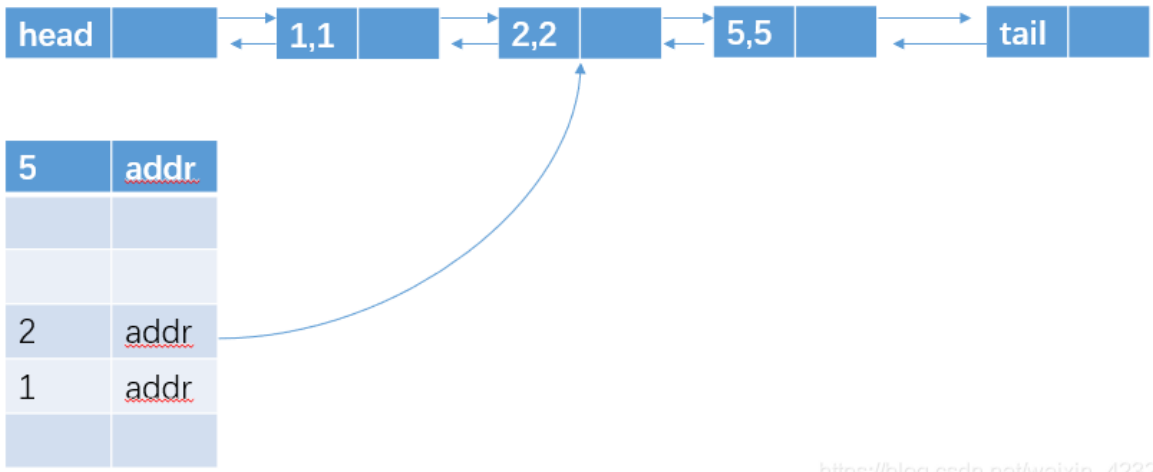
（1）数组的查找时间复杂度为1，但是增加数据和删除数据时时间复杂度很高

（2）链表无法做到更新，查找为1.

（3）哈希表数据是没有顺序的，没有办法找到最久未使用。

可以把链表和哈希表两中数据结构混合使用，哈希表中保存链表中数据的映射，可以快速找到链表中的数据，进而实现时间复杂度近似为1。

例如：要查找2这个数据，在哈希表中可以快速查找到2这个数，之后在链表中实现删除和增加元素。



具体思路：

（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，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。

代码实现

```
typedef struct node{
        int val;
        int key;
        struct node *pre;
        struct node *next;
}Node,*LinkList;//双向链表节点结构
typedef struct {
        LinkList store;//用来存放数据
        LinkList *next;//使用拉链发处理冲突
}Hash;//哈希表数据结构
typedef struct {
        int size;//当前缓存大小
        int capacity;//缓存容量
        Hash* table;//哈希表
        LinkList head;// 指向最近使用的数据
        LinkList tail;// 指向最久未使用的数据
} LRUCache;
Hash * HashMap(Hash *table,int key,int capacity){
        int addr = key % capacity;
        return &table[addr];
}
void HeadInsertion(LinkList head, LinkList cur)
```

```
{//双链表头插法
        if (cur->pre == NULL && cur->next == NULL)
        {// cur 不在链表中
                cur->pre = head;
                cur->next = head->next;
                head->next->pre = cur;
                head->next = cur;
        } else
        {// cur 在链表中
                LinkList first = head->next;//链表的第一个数据结点
                if ( first != cur)
                {//cur 是否已在第一个
                cur->pre->next = cur->next;//改变前驱结点指向
                cur->next->pre = cur->pre;//改变后继结点指向
                cur->next = first;//插入到第一个结点位置
                cur->pre = head;
                head->next = cur;
                first->pre = cur;
                }
        }
}
LRUCache* lRUCacheCreate(int capacity) {
    LRUCache* obj = (LRUCache*)malloc(sizeof(LRUCache));
    obj->table = (Hash*)malloc(capacity * sizeof(Hash));
    memset(obj->table, 0, capacity * sizeof(Hash));
    obj->head = (LinkList)malloc(sizeof(Node));
    obj->tail = (LinkList)malloc(sizeof(Node));    //创建头、尾结点并初始化
     obj->head->pre = NULL;
     obj->head->next = obj->tail;
     obj->tail->pre = obj->head;
     obj->tail->next = NULL;
     //初始化缓存 大小 和 容量
     obj->size = 0;
     obj->capacity = capacity;
     return obj;
     }
int lRUCacheGet(LRUCache* obj, int key) {
        Hash* addr = HashMap(obj->table, key, obj->capacity);//取得哈希地址
        addr = addr->next;//跳过头结点
        if (addr == NULL){
         return -1;
         }
         while ( addr->next != NULL && addr->store->key != key)
         {//寻找密钥是否存在
                addr = addr->next;
         }   if (addr->store->key == key)
          {//查找成功
                HeadInsertion(obj->head, addr->store);//更新至表头
                return addr->store->val;
          }
                return -1;
}
void lRUCachePut(LRUCache* obj, int key, int value) {
    Hash* addr = HashMap(obj->table, key, obj->capacity);//取得哈希地址
    if (lRUCacheGet(obj, key) == -1)
    {
          if (obj->size>= obj->capacity)
        {
```

```
                LinkList last = obj->tail->pre->pre;
                LinkList del = last->next;
                last->next = obj->tail;
                obj->tail->pre = last;
                Hash *delt = HashMap(obj->table,del->key,obj->capacity);//找到要删除的地址
                Hash *help_delt = delt;
                delt = delt->next;
                while(delt->store->key != del->key)
                {
                        help_delt = delt;//删除的前一个节点
                        delt = delt->next;
                }
                help_delt->next = delt->next;
                 delt->store = NULL;
                 delt->next=NULL;
                 free(delt);

            Hash * new_insert = (Hash*)malloc(sizeof(Hash));
            new_insert->next = addr->next;
            addr->next = new_insert;
            new_insert->store = del;
            del->key = key;
             del->val=value;
             HeadInsertion(obj->head,del);
        }
        else
        {//LPU未满
        Hash* new_node = (Hash *)malloc(sizeof(Hash));
        new_node->store = (LinkList)malloc(sizeof(Node));
        new_node->next = addr->next;
        addr->next = new_node;
        new_node->store->pre  = NULL;
        new_node->store->next = NULL;
        new_node->store->val  = value;
        new_node->store->key  = key;
        HeadInsertion(obj->head,new_node->store);
        (obj->size)++;
         }
    }
    else
     {
     obj->head->next->val = value;//替换数据值
     }
}
void lRUCacheFree(LRUCache* obj) {
         free(obj->table);
         free(obj->head);
         free(obj->tail);
         free(obj);
}
```

执行结果： 通过 显示详情 >

执行用时： 104 ms ，在所有 C 提交中击败了 89.29% 的用户

内存消耗： 24.5 MB ，在所有 C 提交中击败了 100.00% 的用户

炫耀一下：

写题解，分享我的解题思路

进行下一个挑战：