<https://leetcode.com/problems/lru-cache/>

Design a data structure that follows the constraints of a **[Least Recently Used (LRU) cache](https://en.wikipedia.org/wiki/Cache_replacement_policies" \l "LRU)**.

Implement the LRUCache class:

* LRUCache(int capacity) Initialize the LRU cache with **positive** size capacity.
* int get(int key) Return the value of the key if the key exists, otherwise return -1.
* void put(int key, int value) Update the value of the key if the key exists. Otherwise, add the key-value pair to the cache. If the number of keys exceeds the capacity from this operation, **evict** the least recently used key.

The functions get and put must each run in O(1) average time complexity.

**Example 1:**

Input

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

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

Output

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

Explanation

LRUCache lRUCache = new LRUCache(2);

lRUCache.put(1, 1); // cache is {1=1}

lRUCache.put(2, 2); // cache is {1=1, 2=2}

lRUCache.get(1); // return 1

lRUCache.put(3, 3); // LRU key was 2, evicts key 2, cache is {1=1, 3=3}

lRUCache.get(2); // returns -1 (not found)

lRUCache.put(4, 4); // LRU key was 1, evicts key 1, cache is {4=4, 3=3}

lRUCache.get(1); // return -1 (not found)

lRUCache.get(3); // return 3

lRUCache.get(4); // return 4

**Constraints:**

* 1 <= capacity <= 3000
* 0 <= key <= 104
* 0 <= value <= 105
* At most 2 \* 105 calls will be made to get and put.

**Attempt 1: 2023-06-06**

**Solution 1: Double Linked List and Map (10 min)**

class LRUCache {

class Node {

Node prev;

Node next;

int key;

int val;

public Node(int key, int val) {

this.key = key;

this.val = val;

}

}

int capacity;

Node head;

Node tail;

Map<Integer, Node> map;

public LRUCache(int capacity) {

this.head = new Node(-1, -1);

this.tail = new Node(-1, -1);

// Create double linked list

head.next = tail;

tail.prev = head;

this.map = new HashMap<Integer, Node>();

this.capacity = capacity;

}

public int get(int key) {

if(!map.containsKey(key)) {

return -1;

}

Node cur = map.get(key);

cur.next.prev = cur.prev;

cur.prev.next = cur.next;

moveToTail(cur);

return cur.val;

}

public void put(int key, int value) {

if(get(key) != -1) {

map.get(key).val = value;

return;

}

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

// Remove least recent node from both map and double linked

// list, the node as 'head.next' is the least recent

map.remove(head.next.key);

head.next = head.next.next;

head.next.prev = head;

}

// Add new node to both map and tail of double linked list

Node insert = new Node(key, value);

map.put(key, insert);

moveToTail(insert);

}

// Most recent get/put node move to tail

// least recent get/put node keep at head

private void moveToTail(Node node) {

node.next = tail;

tail.prev.next = node;

node.prev = tail.prev;

tail.prev = node;

}

}

/\*\*

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

\* LRUCache obj = new LRUCache(capacity);

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

\* obj.put(key,value);

\*/

**Refer to**

<https://aaronice.gitbook.io/lintcode/data_structure/lru_cache>

#### HashMap + Doubly LinkedList

LRU，也就是least recently used，最近使用最少的；这样一个数据结构，能够保持一定的顺序，使得最近使用过的时间或者顺序被记录，实际上，具体每一个item最近一次何时被使用的，并不重要，重要的是在这样的一个结构中，item的相对位置代表了最近使用的顺序；满足这样考虑的结构可以是链表list或者数组array，不过前者更有利于insert和delete的操纵，此外，需要记录这个链表的head和tail，方便进行移动到tail或者删除head的操作，即：[head.next作为最近最少使用的item](http://head.next作为最近最少使用的item)，tail.prev为最近使用过的item，在set时，如果超出capacity，则删除[head.next](http://head.next)，同时将要插入的item放入tail.prev, 而get时，如果存在，只需把item更新到tail.prev即可。

这样set与get均为O(1)时间的操作 （HashMap Get/Set + LinkedList Insert/Delete)，空间复杂度为O(n), n为capacity。

#### LInkedHashMap

使用LinkedHashMap能很方便地实现LRU，定义好access-order，自定义removeEldestEntry()。

## **Solution**

#### \*(Preferred Implementation) Use Doubly Linked List

public class LRUCache {

private class Node {

Node prev;

Node next;

int key;

int value;

public Node(int key, int value) {

this.key = key;

this.value = value;

this.prev = null;

this.next = null;

}

}

private int capacity;

private HashMap<Integer, Node> hm = new HashMap<Integer, Node>();

private Node head = new Node(-1, -1);

private Node tail = new Node(-1, -1);

// @param capacity, an integer

public LRUCache(int capacity) {

this.capacity = capacity;

this.head.next = this.tail;

this.tail.prev = this.head;

}

// @return an integer

public int get(int key) {

if (!hm.containsKey(key)) {

return -1;

}

Node current = hm.get(key);

current.prev.next = current.next;

current.next.prev = current.prev;

moveToTail(current);

return hm.get(key).value;

}

// @param key, an integer

// @param value, an integer

// @return nothing

public void set(int key, int value) {

if (get(key) != -1) {

hm.get(key).value = value;

return;

}

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

hm.remove(head.next.key);

head.next = head.next.next;

head.next.prev = head;

}

Node insert = new Node(key, value);

hm.put(key, insert);

moveToTail(insert);

}

private void moveToTail(Node current) {

current.next = tail;

tail.prev.next = current;

current.prev = tail.prev;

tail.prev = current;

}

}

**Using LinkedHashMap**

import java.util.LinkedHashMap;

public class LRUCache {

private Map<Integer, Integer> map;

private final int maxEntries;

public LRUCache(int capacity) {

this.maxEntries = capacity;

map = new LinkedHashMap<Integer, Integer>(16, 0.75f, true) {

protected boolean removeEldestEntry(Map.Entry eldest) {

return size() > capacity;

}

};

}

public int get(int key) {

return map.getOrDefault(key, -1);

}

public void set(int key, int value) {

map.put(key,value);

}

}

### Follow-up for LRU Cache

Concurrent LRU cache implementation

See Stackoverflow:

<https://stackoverflow.com/questions/40239485/concurrent-lru-cache-implementation>

If multiple threads access a linked hash map concurrently, and at least one of the threads modifies the map structurally, it must be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the map. If no such object exists, the map should be "wrapped" using the

Collections.synchronizedMap

method. This is best done at creation time, to prevent accidental unsynchronized access to the map:

The best you can do is to make it thread-safe is to wrap it with Collections.synchronizedMap(map) as explained in the [JavaDoc](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html):

Map m = Collections.synchronizedMap(new LinkedHashMap(...));

However it is not enough to make it fully thread-safe you sill need to protect any iteration over the content of the map using the instance of the wrapped map as object's monitor:

Map m = Collections.synchronizedMap(map);

...

Set s = m.keySet(); // Needn't be in synchronized block

...

synchronized (m) { // Synchronizing on m, not s!

Iterator i = s.iterator(); // Must be in synchronized block

while (i.hasNext())

foo(i.next());

}

**Refer to**

<https://segmentfault.com/a/1190000003743083>

### 双向链表加哈希表

#### 复杂度

时间 Get O(1) Set O(1) 空间 O(N)

#### 思路

缓存讲究的就是快，所以我们必须做到O(1)的获取速度，这样看来只有哈希表可以胜任。但是哈希表无序的，我们没办法在缓存满时，将最早更新的元素给删去。那么是否有一种数据结构，可以将先进的元素先出呢？那就是队列。所以我们将元素存在队列中，并用一个哈希表记录下键值和元素的映射，就可以做到O(1)获取速度，和先进先出的效果。然而，当我们获取一个元素时，还需要把这个元素再次放到队列头，这个元素可能在队列的任意位置，可是队列并不支持对任意位置的增删操作。而最适合对任意位置增删操作的数据结构又是什么呢？是链表。我可以用链表来实现一个队列，这样就同时拥有链表和队列的特性了。不过，如果仅用单链表的话，在任意位置删除一个节点还是很麻烦的，要么记录下该节点的上一个节点，要么遍历一遍。所以双向链表是最好的选择。我们用双向链表实现一个队列用来记录每个元素的顺序，用一个哈希表来记录键和值的关系，就行了。

#### 注意

* 这题更多的是考察用数据结构进行设计的能力，再写代码时尽量将子函数拆分出来，先写个整体的框架。
* 移出链表最后一个节点时，要记得在链表和哈希表中都移出该元素，所以节点中也要记录Key的信息，方便在哈希表中移除

#### 代码

public class LRUCache {

int size;

int capacity;

ListNode tail;

ListNode head;

Map<Integer, ListNode> map;

public LRUCache(int capacity) {

this.head = new ListNode(-1,-1);

this.tail = new ListNode(-1,-1);

head.next = tail;

tail.prev = head;

this.size = 0;

this.capacity = capacity;

this.map = new HashMap<Integer, ListNode>();

}

public int get(int key) {

ListNode n = map.get(key);

if(n != null){

moveToHead(n);

return n.val;

} else {

return -1;

}

}

public void set(int key, int value) {

ListNode n = map.get(key);

if(n == null){

n = new ListNode(value, key);

attachToHead(n);

size++;

} else {

n.val = value;

moveToHead(n);

}

// 如果更新节点后超出容量，删除最后一个

if(size > capacity){

removeLast();

size--;

}

map.put(key, n);

}

// 将一个孤立节点放到头部

private void attachToHead(ListNode n){

n.next = head.next;

n.next.prev = n;

head.next = n;

n.prev = head;

}

// 将一个链表中的节点放到头部

private void moveToHead(ListNode n){

n.prev.next = n.next;

n.next.prev = n.prev;

attachToHead(n);

}

// 移出链表最后一个节点

private void removeLast(){

ListNode last = tail.prev;

last.prev.next = tail;

tail.prev = last.prev;

map.remove(last.key);

}

public class ListNode {

ListNode prev;

ListNode next;

int val;

int key;

public ListNode(int v, int k){

this.val = v;

this.prev = null;

this.next = null;

this.key = k;

}

}

}