**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#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.

/\*\*

\* @param {number} capacity

\*/

var LRUCache = function(capacity) {

};

/\*\*

\* @param {number} key

\* @return {number}

\*/

LRUCache.prototype.get = function(key) {

};

/\*\*

\* @param {number} key

\* @param {number} value

\* @return {void}

\*/

LRUCache.prototype.put = function(key, value) {

};

/\*\*

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

\* var obj = new LRUCache(capacity)

\* var param\_1 = obj.get(key)

\* obj.put(key,value)

\*/

Approach 1: Ordered dictionary

**Intuition**

We're asked to implement [the structure](https://en.wikipedia.org/wiki/Cache_replacement_policies#LRU) which provides the following operations in \mathcal{O}(1)O(1) time :

* Get the key / Check if the key exists
* Put the key
* Delete the first added key

The first two operations in \mathcal{O}(1)O(1) time are provided by the standard hashmap, and the last one - by linked list.

There is a structure called *ordered dictionary*, it combines behind both hashmap and linked list. In Python this structure is called *[OrderedDict](https://docs.python.org/3/library/collections.html" \l "collections.OrderedDict)* and in Java *[LinkedHashMap](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html)*.

Let's use this structure here.

**Implementation**

class LRUCache extends LinkedHashMap<Integer, Integer>{

private int capacity;

public LRUCache(int capacity) {

super(capacity, 0.75F, true);

this.capacity = capacity;

}

public int get(int key) {

return super.getOrDefault(key, -1);

}

public void put(int key, int value) {

super.put(key, value);

}

@Override

protected boolean removeEldestEntry(Map.Entry<Integer, Integer> eldest) {

return size() > capacity;

}

}

/\*\*

\* 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);

\*/

**Complexity Analysis**

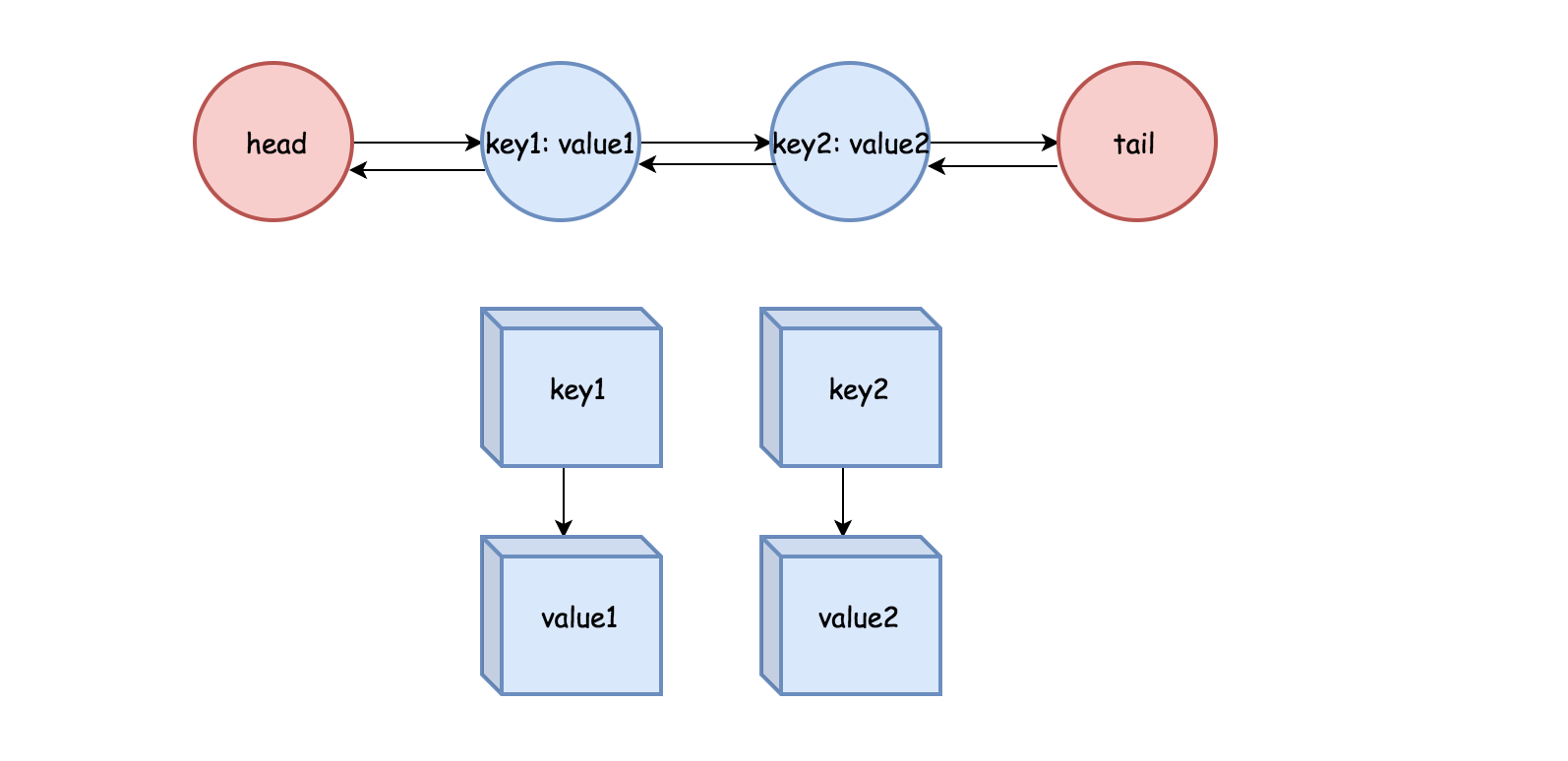
* Time complexity : \mathcal{O}(1)O(1) both for put and get since all operations with ordered dictionary : get/in/set/move\_to\_end/popitem (get/containsKey/put/remove) are done in a constant time.
* Space complexity : \mathcal{O}(capacity)O(*capacity*) since the space is used only for an ordered dictionary with at most capacity + 1 elements.

Approach 2: Hashmap + DoubleLinkedList

**Intuition**

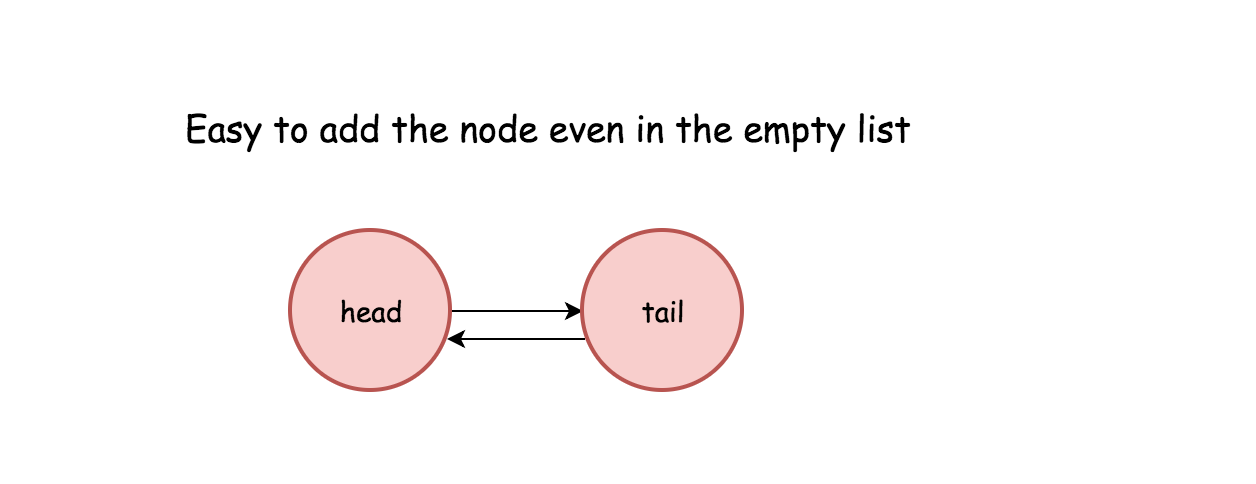
This Java solution is an extended version of the [the article published on the Discuss forum](https://leetcode.com/problems/lru-cache/discuss/45911/Java-Hashtable-%2B-Double-linked-list-(with-a-touch-of-pseudo-nodes)).

The problem can be solved with a hashmap that keeps track of the keys and its values in the double linked list. That results in \mathcal{O}(1)O(1) time for put and get operations and allows to remove the first added node in \mathcal{O}(1)O(1) time as well.



One advantage of *double* linked list is that the node can remove itself without other reference. In addition, it takes constant time to add and remove nodes from the head or tail.

One particularity about the double linked list implemented here is that there are *pseudo head* and *pseudo tail* to mark the boundary, so that we don't need to check the null node during the update.



**Implementation**

public class LRUCache {

class DLinkedNode {

int key;

int value;

DLinkedNode prev;

DLinkedNode next;

}

private void addNode(DLinkedNode node) {

/\*\*

\* Always add the new node right after head.

\*/

node.prev = head;

node.next = head.next;

head.next.prev = node;

head.next = node;

}

private void removeNode(DLinkedNode node){

/\*\*

\* Remove an existing node from the linked list.

\*/

DLinkedNode prev = node.prev;

DLinkedNode next = node.next;

prev.next = next;

next.prev = prev;

}

private void moveToHead(DLinkedNode node){

/\*\*

\* Move certain node in between to the head.

\*/

removeNode(node);

addNode(node);

}

private DLinkedNode popTail() {

/\*\*

\* Pop the current tail.

\*/

DLinkedNode res = tail.prev;

removeNode(res);

return res;

}

private Map<Integer, DLinkedNode> cache = new HashMap<>();

private int size;

private int capacity;

private DLinkedNode head, tail;

public LRUCache(int capacity) {

this.size = 0;

this.capacity = capacity;

head = new DLinkedNode();

// head.prev = null;

tail = new DLinkedNode();

// tail.next = null;

head.next = tail;

tail.prev = head;

}

public int get(int key) {

DLinkedNode node = cache.get(key);

if (node == null) return -1;

// move the accessed node to the head;

moveToHead(node);

return node.value;

}

public void put(int key, int value) {

DLinkedNode node = cache.get(key);

if(node == null) {

DLinkedNode newNode = new DLinkedNode();

newNode.key = key;

newNode.value = value;

cache.put(key, newNode);

addNode(newNode);

++size;

if(size > capacity) {

// pop the tail

DLinkedNode tail = popTail();

cache.remove(tail.key);

--size;

}

} else {

// update the value.

node.value = value;

moveToHead(node);

}

}

}

**Complexity Analysis**

* Time complexity : \mathcal{O}(1)O(1) both for put and get.
* Space complexity : \mathcal{O}(capacity)O(*capacity*) since the space is used only for a hashmap and double linked list with at most capacity + 1 elements.