146. LRU Cache - Copy

Design a data structure that follows the constraints of a **Least Recently Used (LRU) cache**.

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`.

```
class Node:
def __init__(self, k, v):
    self.key = k
    self.val = v
    self.prev = None
    self.next = None

class LRUCache:
def __init__(self, capacity):
    self.capacity = capacity
    self.dic = dict()
    self.head = Node(0, 0)
    self.tail = Node(0, 0)
    self.head.next = self.tail
    self.tail.prev = self.head

def get(self, key):
    if key in self.dic:
        n = self.dic[key]
        self._remove(n)
        self._add(n)
        return n.val
    return -1

def set(self, key, value):
    if key in self.dic:
        self._remove(self.dic[key])
    n = Node(key, value)
    self._add(n)
    self.dic[key] = n
    if len(self.dic) > self.capacity:
        n = self.head.next
        self._remove(n)
        del self.dic[n.key]

def _remove(self, node):
    p = node.prev
    n = node.next
    p.next = n
    n.prev = p

def _add(self, node):
    p = self.tail.prev
    p.next = node
    self.tail.prev = node
    node.prev = p
    node.next = self.tail
```

MY APPROACH

```
from collections import deque
class LRUCache:

    def __init__(self, capacity: int):
        self.cache = deque()
        self.map = {}
        self.size = capacity

    def get(self, key: int) -> int:
        if key not in self.map:
            return -1
        else:
            temp = self.map[key]
            self.cache.remove(key)
            self.cache.appendleft(key)
            return temp


    def put(self, key: int, value: int) -> None:
        if key in self.map:
            self.map[key] = value
            self.cache.remove(key)
            self.cache.appendleft(key)
        else:
            self.cache.appendleft(key)
            self.map[key] = value
            if len(self.cache)>self.size:
                temp = self.cache.pop()
                del self.map[temp]
```