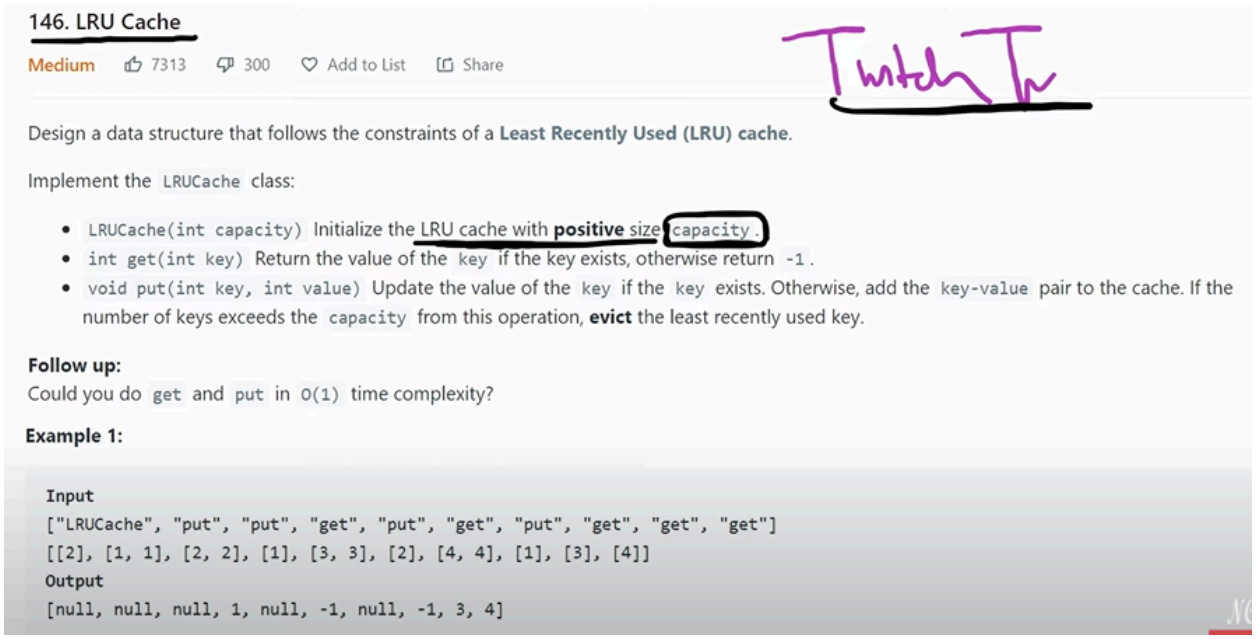
146. LRU Cache

Problem Statement:



146. LRU Cache

Medium 7313 300 Add to List Share

Twitch In

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.

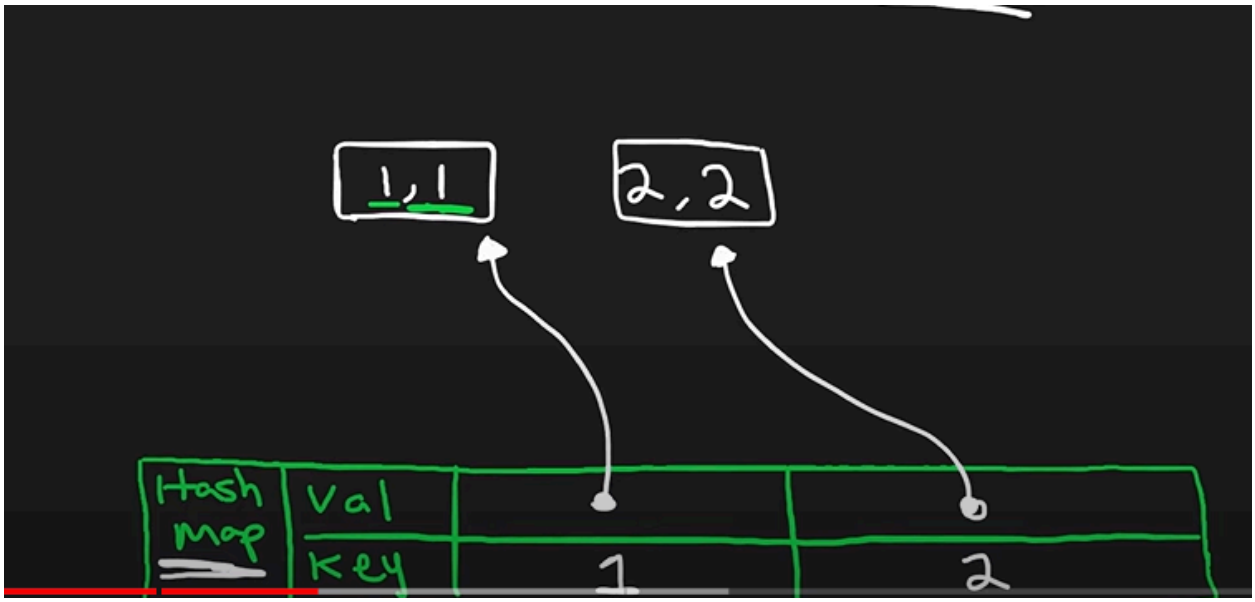
Follow up:
Could you do `get` and `put` in `O(1)` time complexity?

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]
```

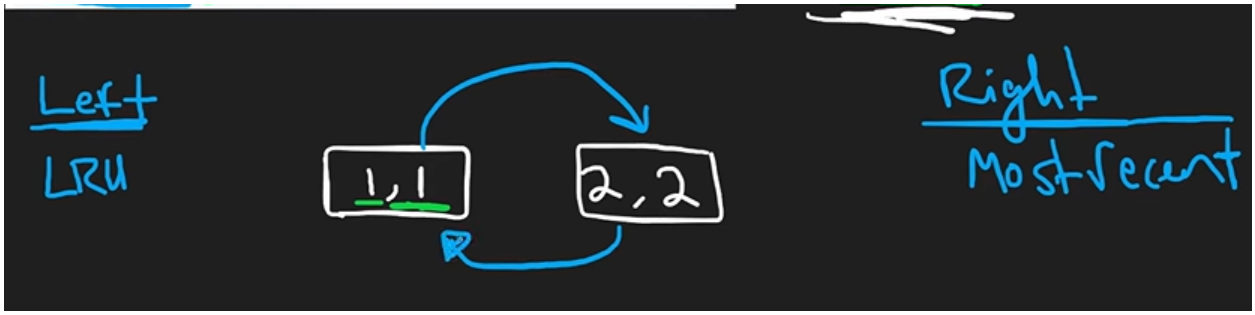
Drawing solution:

1. From the example..the Cache size is 2
2. So first we have 2 PUT operations [1,1] [2,2]
3. Next we have get operation and we need to complete this in O(1) time…so we'll use hashmap for this

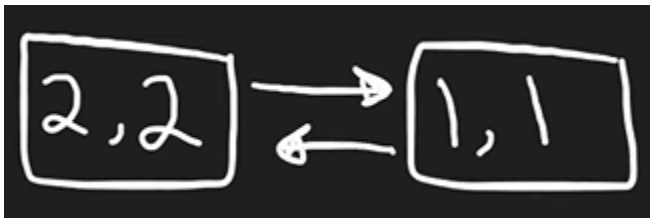


for the key value 1 we'll point node 1,1 and for key value 2 we'll point node 2,2

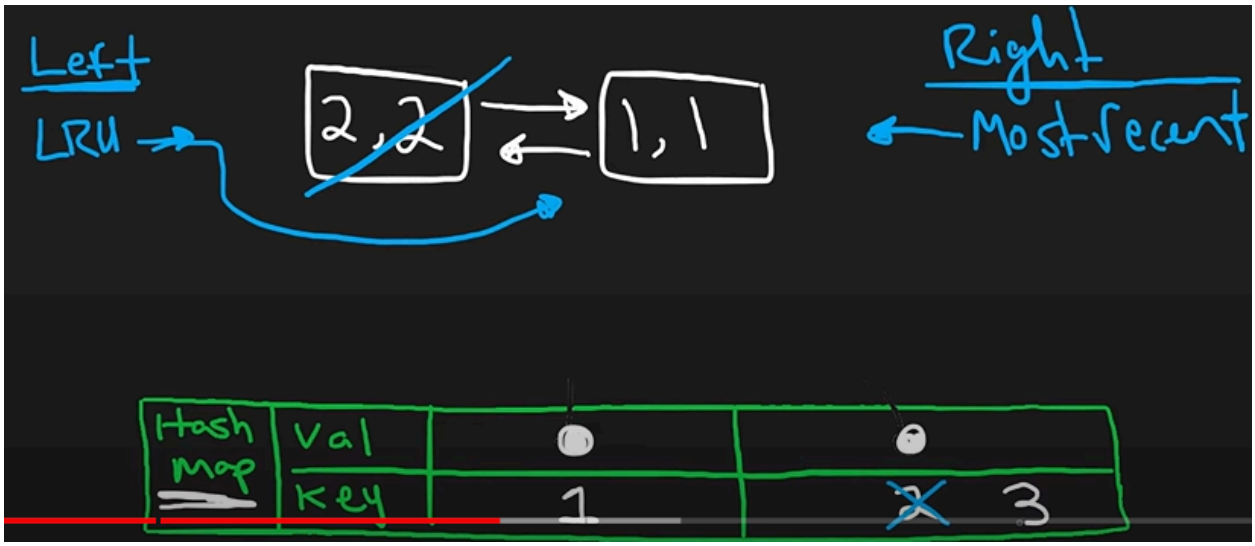
4. Get(1) we'll point to node (1,1) ..since we have retrieved this..it becomes the most recently used and 2,2 becomes LRU
5. We'll arrange nodes such a way that..Left contains LRU and right contains MRU



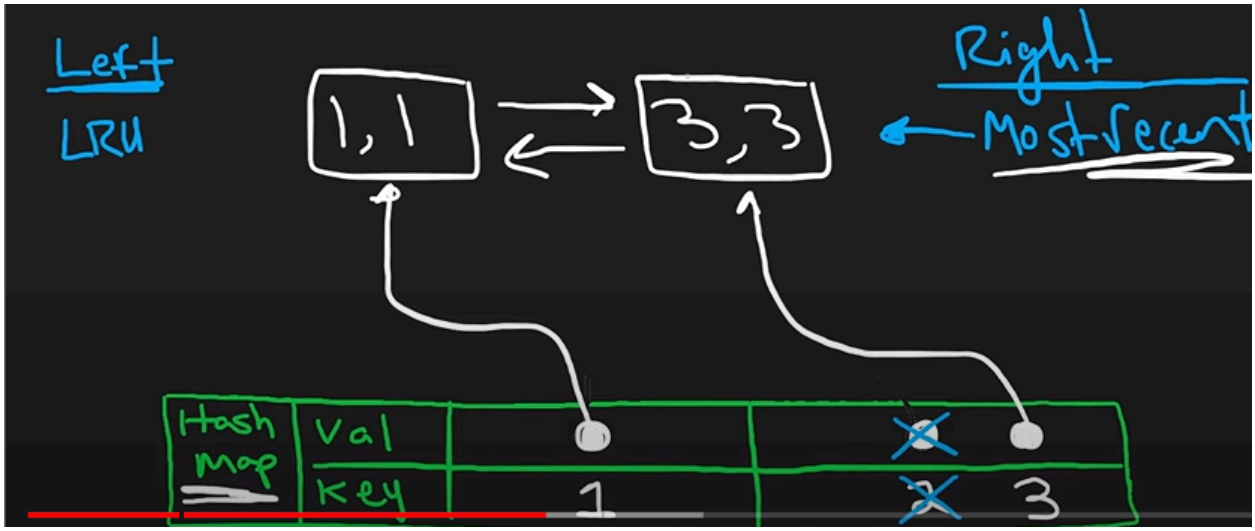
6. We use doubly linked list and swap the arrays and again we will update their pointers



7. Now the next operation is put[3,3] ..as the max size is 2..we'll update LRused and delete it



8. And will update the pointers and nodes



9. And also here Left and right are pointer here

Python code:

1.

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

class LRUCache:

    def __init__(self, capacity: int):
        self.cap = capacity
        self.cache = {} #hashmap which maps keys to node

        #left gives us LRU and right gives us MRU(most recently used)
        self.left,self.right = Node(0,0),Node(0,0)
        self.left.next,self.right.prev = self.right,self.left # we need to keep these nodes
                                                               #connected and we put nodes in bw them
    #remove node from the list
    def remove(self,node):
        prev,nxt = node.prev,node.next
        prev.next,nxt.prev = nxt,prev


    #Add nodes at right
    def insert(self,node):
        prev,nxt = self.right.prev,self.right
        prev.next,nxt.prev = node,node
        node.next,node.prev = nxt,prev
```

```python
def get(self, key: int) -> int:
    if key in self.cache:
        self.remove(self.cache[key])
        self.insert(self.cache[key])
        return self.cache[key].val

    return -1


def put(self, key: int, value: int) -> None:
    if key in self.cache:
        self.remove(self.cache[key])
    self.cache[key] = Node(key,value)
    self.insert(self.cache[key])

    if len(self.cache) > self.cap:
        #remove from the list and delete th LRU from the hashmap
        lru = self.left.next
        self.remove(lru)
        del self.cache[lru.key]
```