In this problem you are asked to design a data structure.

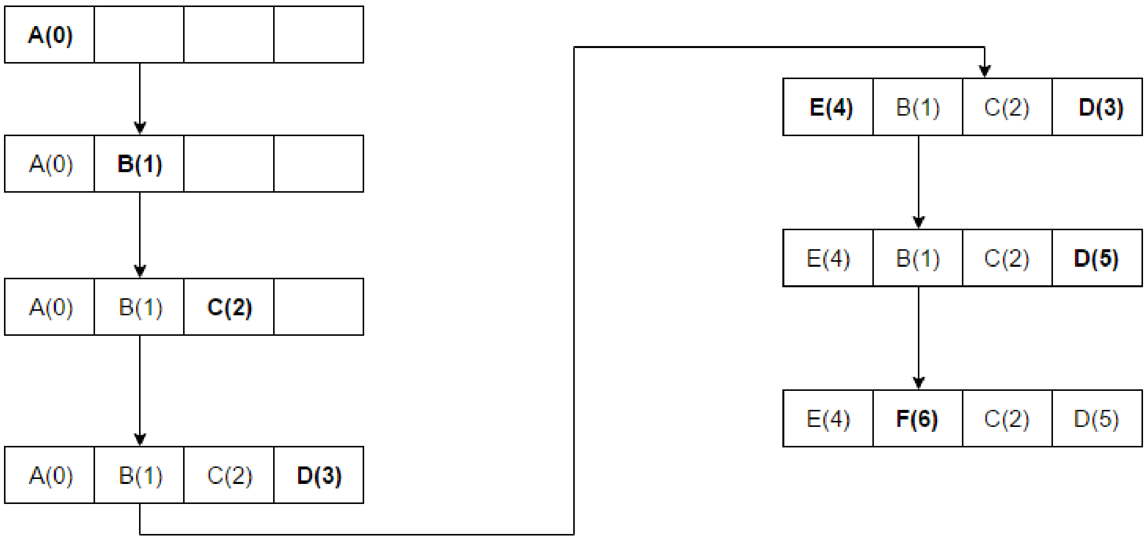
The following lines are taken from Wikipedia.

In computing, **cache algorithms** (also frequently called **cache replacement algorithms** or **cache replacement policies**) are optimizing instructions, or algorithms, that a computer program or a hardware-maintained structure can utilize in order to manage a cache of information stored on the computer. Caching improves performance by keeping recent or often-used data items in memory locations that are faster or computationally cheaper to access than normal memory stores. When the cache is full, the algorithm must choose which items to discard to make room for the new ones.

Least recently used (LRU)[edit]

Discards the least recently used items first. This algorithm requires keeping track of what was used when, which is expensive if one wants to make sure the algorithm always discards the least recently used item. General implementations of this technique require keeping "age bits" for cache-lines and track the "Least Recently Used" cache-line based on age-bits. In such an implementation, every time a cache-line is used, the age of all other cache-lines changes. LRU is actually a family of caching algorithms with members including 2Q by Theodore Johnson and Dennis Shasha,[5] and LRU/K by Pat O'Neil, Betty O'Neil and Gerhard Weikum.[6]

The access sequence for the below example is A B C D E D F.



In the above example once A B C D gets installed in the blocks with sequence numbers (Increment 1 for each new Access) and when E is accessed, it is a miss and it needs to be installed in one of the blocks. According to the LRU Algorithm, since A has the lowest Rank(A(0)), E will replace A.

In the second to last step, D is accessed and therefore the sequence number is updated.

Finally, F is accessed taking the place of B which had the lowest Rank(B(1)) at the moment.

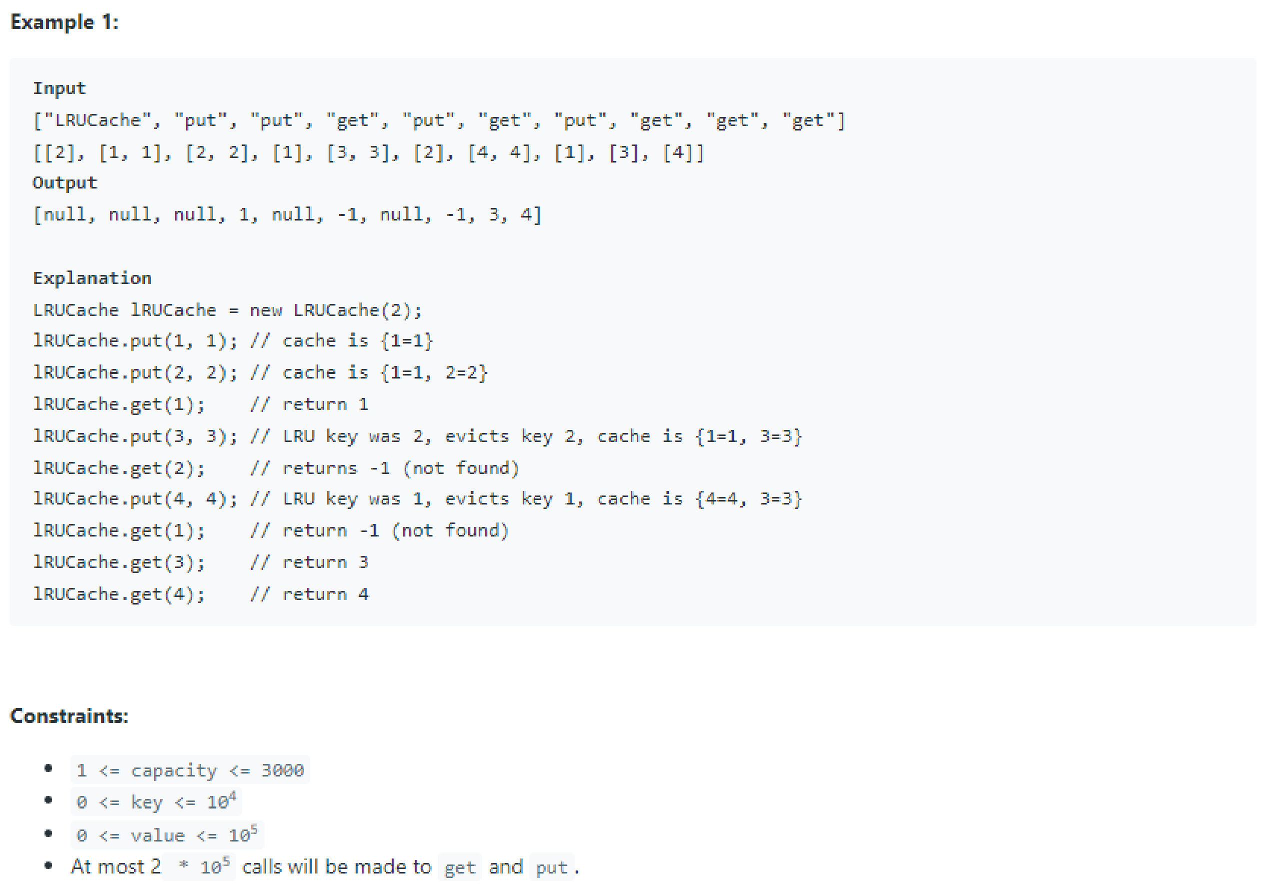
The Problem.

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`
- $0 <= key <= 10^4$
- $0 <= value <= 10^5$
- At most $2 * 10^5$ calls will be made to `get` and `put`.

More examples you can find in the Homework Code

I’ve added 6 test functions. If Everything implemented correctly all the tests should pass the assertions.