146. LRU Cache
Medium

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:

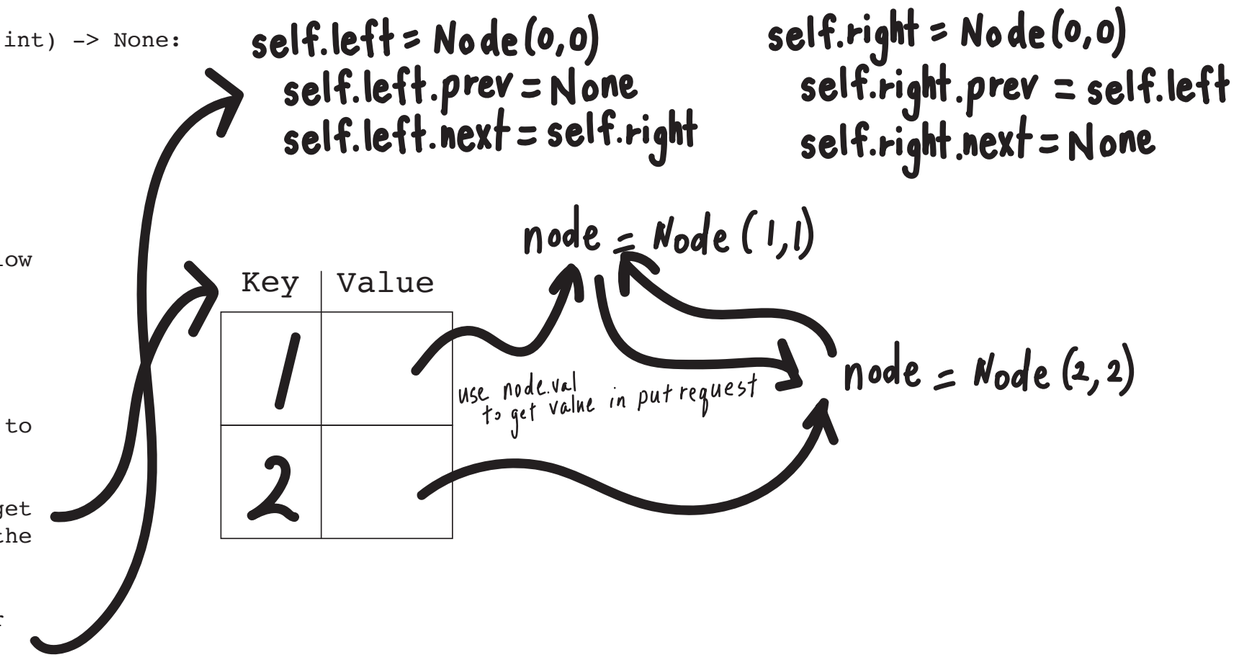
```
Input
["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]
Output
[null, null, null, 1, null, -1, null, -1, 3, 4]

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
```

```
'''
Your LRUCache object will be instantiated and called as such:
    obj = LRUCache(capacity)
    param_1 = obj.get(key)
    obj.put(key,value)
'''
class LRUCache:
    def __init__(self, capacity: int):

    def get(self, key: int) -> int:

    def put(self, key: int, value: int) -> None:
```

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

Approach

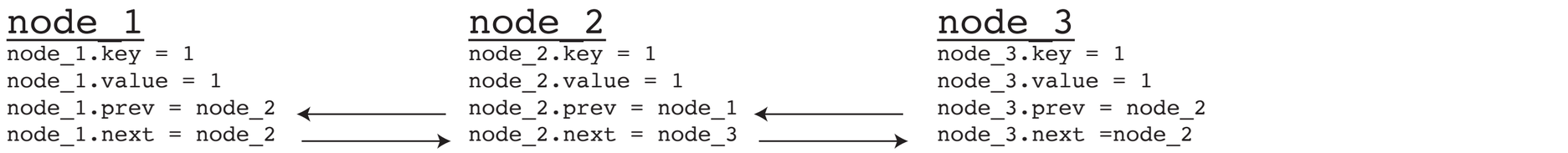
1. Define a class Node to define and use a doubly linked list because a linked list will allow changing the pointing of LRU (least recently used) and MRU (most recently used) quickly, and also add and remove other nodes in O(1) time.
2. Define a variable, 'capacity' to store the capacity of linked list
3. Create a hashmap in order to get the value in O(1) time. Make the hashmap's value point to the Node
4. Left pointer and Right pointer as nodes to keep track of the LRU and MRU



The nodes that the hashmap link to is a part of a doubly linked list.

The following class method, remove(), removes the current node from the doubly linked list.

```
    def remove(self, node):
        prv = node.prev
        nxt = node.next
        prv.next = nxt
        nxt.prev = prv
```



The following class method, insert(), inserts the current node into the doubly linked list via the right (MRU - most recently used) pointer.

```
    def insert(self, node):
        prv = self.right.prev
        nxt = self.right
        prv.next = nxt.prev = node
        node.next = nxt
        node.prev = prv
```