```
146. LRU Cache
                                                    Example 1:
Medium
                                                    Input
Design a data structure that follows the
                                                    ["LRUCache", "put", "put", "get", "put", "get", "get", "get", "get", "get"]
constraints of a Least Recently Used (LRU)
                                                    [[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]
cache.
                                                    [null, null, null, 1, null, -1, null, -1, 3, 4]
Implement the LRUCache class:
                                                    Explanation
- LRUCache(int capacity) Initialize the LRU
                                                    LRUCache | RUCache = new LRUCache(2);
  cache with positive size capacity.
                                                    lRUCache.put(1, 1); // cache is {1=1}
- int get(int key) Return the value of the key
                                                    lRUCache.put(2, 2); // cache is {1=1, 2=2}
  if the key exists, otherwise return -1.
                                                    lRUCache.get(1); // return 1
                                                    lRUCache.put(3, 3); // LRU key was 2, evicts key 2, cache is {1=1, 3=3}
- void put(int key, int value) Update the
  value of the key if the key exists.
                                                    lRUCache.get(2); // returns -1 (not found)
  Otherwise, add the key-value pair to the
                                                    lRUCache.put(4, 4); // LRU key was 1, evicts key 1, cache is {4=4, 3=3}
  cache. If the number of keys exceeds the
                                                    lRUCache.get(1);
                                                                       // return -1 (not found)
                                                                        // return 3
  capacity from this operation, evict the
                                                    lRUCache.get(3);
  least recently used key.
                                                                        // return 4
                                                    lRUCache.get(4);
The functions get and put must each run in
O(1) average time complexity.
                                                                                  class Node:
 Your LRUCache object will be instantiated and called as such:
                                                                                      def __init__(self, key, val):
     obj = LRUCache(capacity)
                                                                                          self.key = key
     param 1 = obj.get(key)
                                                                                          self.value = value
     obj.put(key,value)
                                                                                          self.prev = None
                                                                                          self.next = None
 class LRUCache:
     def __init__(self, capacity: int):
     def get(self, key: int) -> int:
                                                      self.left = Node(0,0) self.right = Node(0,0)
self.left.prev = None self.right.prev = self.left
self.left.next = self.right self.right.next = None
     def put(self, key: int, value: int) -> None:
    Approach
    1. Define a class Node to define
                                                                          node = Node (1,1)
       and use a doubly linked list
      because a linked list will allow
                                                      Key
                                                             Value
      changing the pointing of LRU
       (least recently used) and MRU
       (most recently used) quickly,
                                                                     use node value in put request
                                                                                                node = Node (2,2)
      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
    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.
                                               node 1
                                                                                       node 2
                                                                                                                                node 3
       The following class method,
                                               \overline{\text{node } 1.\text{key}} = 1
                                                                                       \overline{\text{node 2.key}} = 1
                                                                                                                                \overline{\text{node } 3.\text{key}} = 1
       remove(), removes the current node
       from the doubly linked list.
                                               node 1.value = 1
                                                                                       node 2.value = 1
                                                                                                                                node 3.value = 1
                                               node 1.prev = node 2 _____ node 2.prev = node 1 ____
                                                                                                                                node 3.prev = node 2
           def remove(self, node):
                                               node 1.next = node 2 _____ node 2.next = node 3 _____
                                                                                                                                node 3.next =node 2
               prv = node.prev
               nxt = node.next
               prv.next = nxt
               nxt.prev = prv
       The following class method,
                                                                                                                                                 → self.right
                                               self.left ←
       insert(), inserts the current node
       into the doubly linked list via the
                                               self.left.key = 0
       right (MRU - most recently used)
                                               self.left.value = 0
                                                                                                                                                    self.left.value = 0
       pointer.
                                                                                                                                                    self.left.prev = self.left
                                               self.left.prev = None
                                               self.left.next = self.right
                                                                                                                                                    self.left.next = None
            def insert(self, node):
              prv = self.right.prev
              nxt = self.right
              prv.next = nxt.prev = node
                                               self.left ←
                                                                                                       node
                                                                                                                                                    self.right
              node.next = nxt
                                                                                                       \overline{\text{self.ke}}y = ?
                                                                                                                                                    self.right.key = 0
                                               \overline{\text{self.left.key}} = 0
              node.prev = prv
                                               self.left.value = 0
                                                                                                       self.value = ?
                                                                                                                                                    self.right.value = 0
                                                                                                       self.prev = self.left
                                                                                                                                                    self.right.prev = <del>self.left</del> ne
                                               self.left.prev = None
```

self.next = self.right

self.right.next = None

self.left.next = self.right node