146. LRU Cache

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.

```
class LRUCache {
public:
    class Node{
        public:
            int key;
            int val;
            Node* prev;
            Node* next;

            Node(int key, int val){
                this->key = key;
                this->val = val;
            }
    };

    Node* head = new Node(-1, -1);
    Node* tail = new Node(-1, -1);

    int cap;
    unordered_map<int, Node*> m;

    LRUCache(int capacity) {
        cap = capacity;
        head -> next = tail;
        tail -> prev = head;
    }

    void addNode(Node* newnode){
        Node* temp = head -> next;
        newnode -> next = temp;
        newnode -> prev = head;

        head -> next = newnode;
        temp -> prev = newnode;
    }

    void deleteNode(Node* delnode){
        Node* prevv = delnode -> prev;
        Node* nextt = delnode -> next;

        prevv -> next = nextt;
        nextt -> prev = prevv;
    }

    int get(int key) {
        if(m.find(key) != m.end()){
            Node* resNode = m[key];
            int ans = resNode -> val;

            m.erase(key);
            deleteNode(resNode);
            addNode(resNode);

            m[key] = head -> next;
            return ans;
        }
        return -1;
    }

    void put(int key, int value) {
        if(m.find(key) != m.end()){
            Node* curr = m[key];
            m.erase(key);
            deleteNode(curr);
        }

        if(m.size() == cap){
            m.erase(tail -> prev -> key);
            deleteNode(tail -> prev);
        }

        addNode(new Node(key, value));
        m[key] = head -> next;
    }
};
```