LRU Cache (LeetCode 146 - Hard)

Problem Description

Design and implement a data structure for LRU cache. It should support the following operations: **get** and **put**.

get(key) – Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.

put(key, value) – Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.

Solution

```java
/**
 * LeetCode 144
 *
 * get(key) - Get the value of the key if the key exists in the cache,
 * otherwise return -1.
 * put(key, value) - put or insert the value if the key is not already present.
 * When the cache reached its capacity,
 * it should invalidate the least recently used item before inserting a new item.
 *
 * Use a Double Linked List.
 * log(1) for both get and put
 *
 */

public class LRUCache {

	//Test Case
	public static void main(String[] args) {
		LRUCache myCache = new LRUCache(2);
		myCache.put(2, 1);
		myCache.put(1, 1);
		myCache.get(2);
		myCache.put(4, 1);
	}

}
```

```java
public class LRUCache {

	private class Node {
		int key;
		int value;
		Node pre;
		Node next;

		public int getKey() {
			return key;
		}

		public void setKey(int key) {
			this.key = key;
		}

		public int getValue() {
			return value;
		}

		public void setValue(int value) {
			this.value = value;
		}

		public Node getPre() {
			return pre;
		}

		public void setPre(Node pre) {
			this.pre = pre;
		}

		public Node getNext() {
			return next;
		}

		public void setNext(Node next) {
			this.next = next;
		}
	}

}
```

```
public class LRUCache {

	public HashMap<Integer, Node> map;
	public Node head;
	public Node tail;
	int capacity;
	int count;

	public LRUCache(int capacity) {
		this.count = 0;
		this.capacity = capacity;
		this.map = new HashMap<Integer, Node>();
		this.head = new Node();
		this.tail = new Node();
		this.head.setNext(this.tail);
		this.head.setPre(null);
		this.tail.setPre(this.head);
		this.tail.setNext(null);
	}

	public int get(int key) {
		if (map.containsKey(key)) {
			update(key);
			return map.get(key).getValue();
		} else {
			return -1;
		}
	}

	public void put(int key, int value) {
		if (map.containsKey(key)) {
			update(key, value);
		} else {
			add(key, value);
		}
	}

}
```

```java
public class LRUCache {

	private void update(int key) {
		update(key, map.get(key).getValue());
	}

	private void update(int key, int value) {
		Node cur = map.get(key);
		cur.setValue(value);
		map.put(key, cur);

		if(cur.getPre() == head) {
			return;
		}

		cur.getPre().setNext(cur.getNext());
		cur.getNext().setPre(cur.getPre());
		cur.setPre(head);
		cur.setNext(head.getNext());
		head.getNext().setPre(cur);
		head.setNext(cur);
	}

	private void add(int key, int value) {
		if (capacity <= 0) {
			return;
		}

		Node newNode = new Node();
		newNode.setKey(key);
		newNode.setValue(value);
		map.put(key, newNode);

		if(count >= capacity) {
			tail = tail.getPre();
			tail.setNext(null);
			map.remove(tail.getKey());
			count--;
		}

		newNode.setNext(head.getNext());
		newNode.setPre(head);
		head.getNext().setPre(newNode);
		head.setNext(newNode);
		count++;
	}

}
```