LRUDemo

```java
public class LRUCacheDemo {
    public static void run() {
        LRUCache<Integer, String> cache = new LRUCache<>(3);

        cache.put(1, "Value 1");
        cache.put(2, "Value 2");
        cache.put(3, "Value 3");

        System.out.println(cache.get(1)); // Output: Value 1
        System.out.println(cache.get(2)); // Output: Value 2

        cache.put(4, "Value 4");

        System.out.println(cache.get(3)); // Output: null
        System.out.println(cache.get(4)); // Output: Value 4

        cache.put(2, "Updated Value 2");

        System.out.println(cache.get(1)); // Output: Value 1
        System.out.println(cache.get(2)); // Output: Updated Value 2
    }
}
```

NODE

```java
class Node<K, V> {
    K key;
    V value;
    Node<K, V> prev;
    Node<K, V> next;

    public Node(K key, V value) {
        this.key = key;
        this.value = value;
    }
}
```

LRUCache

```java
class LRUCache<K, V> {
    private final int capacity;
    private final Map<K, Node<K, V>> cache;
    private final Node<K, V> head;
    private final Node<K, V> tail;

    public LRUCache(int capacity) {
        this.capacity = capacity;
        cache = new HashMap<>(capacity);
        head = new Node<>(null, null);
        tail = new Node<>(null, null);
        head.next = tail;
        tail.prev = head;
    }

    public synchronized V get(K key) {
        Node<K, V> node = cache.get(key);
        if (node == null) {
            return null;
        }
        moveToHead(node);
        return node.value;
    }

    public synchronized void put(K key, V value) {
        Node<K, V> node = cache.get(key);
        if (node != null) {
            node.value = value;
            moveToHead(node);
        } else {
            node = new Node<>(key, value);
            cache.put(key, node);
            addToHead(node);
            if (cache.size() > capacity) {
                Node<K, V> removedNode = removeTail();
                cache.remove(removedNode.key);
            }
        }
    }

    private void addToHead(Node<K, V> node) {
        node.prev = head;
        node.next = head.next;
        head.next.prev = node;
        head.next = node;
    }

    private void removeNode(Node<K, V> node) {
        node.prev.next = node.next;
        node.next.prev = node.prev;
    }

    private void moveToHead(Node<K, V> node) {
        removeNode(node);
        addToHead(node);
    }

    private Node<K, V> removeTail() {
        Node<K, V> node = tail.prev;
        removeNode(node);
        return node;
    }
}
```