LRU（最近最少使用）淘汰算法

1. 简介

LRU是**Least Recently Used**的缩写，即**最近最少使用**，是一种常用的页面置换算法，选择最近最久未使用的页面予以淘汰。
该算法赋予每个页面一个访问字段，用来记录一个页面自上次被访问以来所经历的时间 t，当须淘汰一个页面时，选择现有页面中其 t 值最大的，即最近最少使用的页面予以淘汰。

2. 原理分析

LRU算法主要利用的数据结构是双向链表+哈希表。
可以利用哈希表的快速查找的优势和利用双向链表的快速插入和删除的特点，可以让LRU算法时间复杂度为O(1), 但副作用时空间复杂度会稍微增加。

需要注意的几点：

1. 当被cache中已有的数据被操作时（获取或者添加），需要把操作的数据移动到链表的头部节点。
2. 当cache容量已满，此时数据被添加时，需要把双向链表的最后一个节点删除，然后才能进行添加操作。

3. 源码

```
/**
 * LRU算法 Created by MOON
 *
 * @param <K>
 * @param <V>
 */
public class LruCache<K, V> {
    // LRU算法全称叫Least recently used（最近最少使用）
    // 算法根据数据的历史访问记录来进行淘汰数据，其核心思想是“如果数据最近被访问
```

```
过，那么将来被访问的几率也更高”。

    // 基本数据类型为双向链表的节点，有前驱节点和后续节点。
    private class Node {
        private K k;
        private V v;
        private Node pre;
        private Node next;

        public Node() {
        }

        public Node(K k, V v) {
            this.k = k;
            this.v = v;
        }
    }

    // 双向LinkedList
    private class DoubleLinkedList {
        private Node head, tail;
        private int size;

        public DoubleLinkedList() {
            // 这里的head和tail的作用是哨兵，它们并不是真正数据的头和尾
            head = new Node();
            tail = new Node();
            head.next = tail;
            tail.pre = head;
            size = 0;
        }

        // 向链表的头加入节点
        public void addFirst(Node node) {
            node.next = head.next;
            head.next.pre = node;
            head.next = node;
            node.pre = head;
            size++;
```

```
    }

    // 删除链表中的节点
    public void remove(Node node) {
        node.pre.next = node.next;
        node.next.pre = node.pre;
        size--;
    }

    // 删除链表中的最后一个节点
    public Node removeLast() {
        if (size > 0) {
            Node last = tail.pre;
            remove(last);
            return last;
        } else {
            return null;
        }
    }

    public int getSize() {
        return size;
    }

}

 // 哈希表，用来以O(1)的速度查找数据在cache中位置
private HashMap<K, Node> map;
// 双向链表，用来存储cache数据，因为是双向可以以O(1)速度来进行添加删除操作
private DoubleLinkedList cache;
// cache的容量
private int capacity;

public LruCache(int capacity) {
    this.capacity = capacity;
    map = new HashMap<>();
    cache = new DoubleLinkedList();
}
```

```
    // 获取
    public V get(K key) {
        if (!map.containsKey(key)) {
            return null;
        }
        V value = map.get(key).v;
        // 当cache被操作时，需要把它移动到链表的头部
        put(key, value);
        return value;
    }

    // 插入
    public void put(K key, V value) {
        Node node = new Node(key, value);
        if (map.containsKey(key)) {
            cache.remove(map.get(key));
            cache.addFirst(map.get(key));
        } else {
            if (cache.size >= capacity) {
                Node lastNode = cache.removeLast();
                map.remove(lastNode.k);
            }
            cache.addFirst(node);
        }
        map.put(key, node);
    }

    // 打印key和value
    public void print() {
        Set keys = map.keySet();
        Iterator iterator = keys.iterator();

        while (iterator.hasNext()) {
            Object obj = iterator.next();
            System.out.print("key: " + obj);
            System.out.println("  value: " + map.get(obj).v);
        }
    }
```

```
    // 打印双向链表中最后一个节点的key和value
    public void printLastNode() {
        System.out.print("key: " + cache.tail.pre.k);
        System.out.println("  value: " + cache.tail.pre.v);
    }
}
```