```
            }
            return max - Math.min(arr[0], arr[arr.length - 1]);
        }
```

设计可以变更的缓存结构

【题目】

设计一种缓存结构，该结构在构造时确定大小，假设大小为 K，并有两个功能：

- set(key,value)：将记录(key,value)插入该结构。
- get(key)：返回 key 对应的 value 值。

【要求】

1．set 和 get 方法的时间复杂度为 $O(1)$。

2．某个 key 的 set 或 get 操作一旦发生，认为这个 key 的记录成了最经常使用的。

3．当缓存的大小超过 K 时，移除最不经常使用的记录，即 set 或 get 最久远的。

【举例】

假设缓存结构的实例是 cache，大小为 3，并依次发生如下行为：

1．cache.set("A",1)。最经常使用的记录为("A",1)。

2．cache.set("B",2)。最经常使用的记录为("B",2)，("A",1)变为最不经常的。

3．cache.set("C",3)。最经常使用的记录为("C",2)，("A",1)还是最不经常的。

4．cache.get("A")。最经常使用的记录为("A",1)，("B",2)变为最不经常的。

5．cache.set("D",4)。大小超过了 3，所以移除此时最不经常使用的记录("B",2)，加入记录("D",4)，并且为最经常使用的记录，然后("C",2)变为最不经常使用的记录。

【难度】

尉 ★★☆☆

【解答】

这种缓存结构可以由双端队列与哈希表相结合的方式实现。首先实现一个基本的双向链表节点的结构，请参看如下代码中的 Node 类。

```
public class Node<V> {
        public V value;
        public Node<V> last;
        public Node<V> next;

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

根据双向链表节点结构 Node，实现一种双向链表结构 NodeDoubleLinkedList，在该结构中优先级最低的节点是 head（头），优先级最高的节点是 tail（尾）。这个结构有以下三种操作：

- 当加入一个节点时，将新加入的节点放在这个链表的尾部，并将这个节点设置为新的尾部，参见如下代码中的 addNode 方法。
- 对这个结构中的任意节点，都可以分离出来并放到整个链表的尾部，参见如下代码中的 moveNodeToTail 方法。
- 移除 head 节点并返回这个节点，然后将 head 设置成老 head 节点的下一个，参见如下代码中的 removeHead 方法。

NodeDoubleLinkedList 结构全部实现如下。

```
public class NodeDoubleLinkedList<V> {
        private Node<V> head;
        private Node<V> tail;

        public NodeDoubleLinkedList() {
                this.head = null;
                this.tail = null;
        }

        public void addNode(Node<V> newNode) {
                if (newNode == null) {
                        return;
                }
                if (this.head == null) {
                        this.head = newNode;
                        this.tail = newNode;
                } else {
                        this.tail.next = newNode;
                        newNode.last = this.tail;
                        this.tail = newNode;
                }
        }

        public void moveNodeToTail(Node<V> node) {
                if (this.tail == node) {
```

```
                return;
            }
            if (this.head == node) {
                this.head = node.next;
                this.head.last = null;
            } else {
                node.last.next = node.next;
                node.next.last = node.last;
            }
            node.last = this.tail;
            node.next = null;
            this.tail.next = node;
            this.tail = node;
        }

        public Node<V> removeHead() {
            if (this.head == null) {
                return null;
            }
            Node<V> res = this.head;
            if (this.head == this.tail) {
                this.head = null;
                this.tail = null;
            } else {
                this.head = res.next;
                res.next = null;
                this.head.last = null;
            }
            return res;
        }

    }
```

最后实现最终的缓存结构。如何把记录之间按照“访问经常度”来排序，就是上文提到的 NodeDoubleLinkedList 结构。一旦加入新的记录，就把该记录加到 NodeDoubleLinkedList 的尾部（addNode）。一旦获得（get）或设置（set）一个记录的 key，就将这个 key 对应的 node 在 NodeDoubleLinkedList 中调整到尾部（moveNodeToTail）。一旦 cache 满了，就删除“最不经常使用”的记录，也就是移除 NodeDoubleLinkedList 的当前头部(removeHead)。

为了能让每一个 key 都能找到在 NodeDoubleLinkedList 所对应的节点，同时让每一个 node 都能找到各自的 key，我们还需要两个 map 分别记录 key 到 node 的映射，以及 node 到 key 的映射，就是如下 MyCache 结构中的 keyNodeMap 和 nodeKeyMap。具体实现请参看如下代码中的 MyCache 类。

```
    public class MyCache<K, V> {
        private HashMap<K, Node<V>> keyNodeMap;
```

```
    private HashMap<Node<V>, K> nodeKeyMap;
    private NodeDoubleLinkedList<V> nodeList;
    private int capacity;

    public MyCache(int capacity) {
        if (capacity < 1) {
            throw new RuntimeException("should be more than 0.");
        }
        this.keyNodeMap = new HashMap<K, Node<V>>();
        this.nodeKeyMap = new HashMap<Node<V>, K>();
        this.nodeList = new NodeDoubleLinkedList<V>();
        this.capacity = capacity;
    }

    public V get(K key) {
        if (this.keyNodeMap.containsKey(key)) {
            Node<V> res = this.keyNodeMap.get(key);
            this.nodeList.moveNodeToTail(res);
            return res.value;
        }
        return null;
    }

    public void set(K key, V value) {
        if (this.keyNodeMap.containsKey(key)) {
            Node<V> node = this.keyNodeMap.get(key);
            node.value = value;
            this.nodeList.moveNodeToTail(node);
        } else {
            Node<V> newNode = new Node<V>(value);
            this.keyNodeMap.put(key, newNode);
            this.nodeKeyMap.put(newNode, key);
            this.nodeList.addNode(newNode);
            if (this.keyNodeMap.size() == this.capacity + 1) {
                this.removeMostUnusedCache();
            }
        }
    }

    private void removeMostUnusedCache() {
        Node<V> removeNode = this.nodeList.removeHead();
        K removeKey = this.nodeKeyMap.get(removeNode);
        this.nodeKeyMap.remove(removeNode);
        this.keyNodeMap.remove(removeKey);
    }

}
```