# 跳表：Redis中如何实现有序集合

题目来源：Leetcode 1206：https://leetcode-cn.com/problems/design-skiplist/

## 步骤一：实现不含索引的跳表

### Java代码

跳表实现代码：

```
class Skiplist {
    final int HEAD_VALUE = -1;  // 链表头节点的值
    final Node HEAD = new Node(HEAD_VALUE);

    Node head;           // 最左上角的头节点，所有操作的开始位置
    int levels;          // 当前层级，即 head 节点所在的最高层数

    public Skiplist() {
        head = HEAD;
        levels = 1;
    }

    class Node {
        int val;
        Node right, down;

        Node(int val) {
            this(val, null, null);
        }

        Node(int val, Node right, Node down) {
            this.val = val;
            this.right = right;
            this.down = down;
        }
    }

    public void add(int num) {}
    public boolean search(int target) {}
    public boolean erase(int num) {}
}
```

跳表的执行函数实现细节：

```
    /**
     * 插入节点。将节点插入到原链表中正确的排序位置
     *
     * 1.定位插入位置：原链表中 >= num 的最小节点前
     * 2.插入新节点
     * 3.根据扔硬币决定（是否）生成索引
     *
     * @param num
     */
    public void add(int num) {
        // 1.定位插入位置：原链表中 >= num 的最小节点前
        Node node = head;
        int i = 0; // 操作上述数组
        while (node != null) { // node==null，到达原链表
            while (node.right != null && node.right.val < num) {
                node = node.right;
            }
            if (node.down == null) {
                break;
            }
            // 继续查找下一层的位置
            node = node.down;
        }
        // 2.插入新节点
        Node newNode = new Node(num, node.right, null);
        node.right = newNode;

        // 3.TODO 根据扔硬币决定（是否）生成索引

    }

    /**
     * 从 head 开始，从左到右、从上到下依次查找
     * 1.小于，往右
     * 2.相同，则返回
     * 3.链表结尾，或大于，往下
     *
     * @param target
     * @return
     */
    public boolean search(int target) {
        Node n = head;
        while (n != null) {
            // 1.在同一层级上向右查找，直到链表的结尾
            while (n.right != null && n.right.val < target) {
                n = n.right;
            }
            // 2.若找到，返回true
            Node right = n.right; // 要查找的节点
            if (right != null && right.val == target) {
                return true;
            }
            // 3.若右侧数据较大，向下一层
            n = n.down;
        }
        return false;
    }

    /**
     * 遍历跳表，查找与给定值相同的节点，删除每一层
     * 1.获取该指定数据节点的前一个节点
     * 2.与当前层链表断开
     * 3.下移，删除每一层
     *
     * @param num
     * @return
     */
    public boolean erase(int num) {
        boolean exist = false;
        Node n = head;
        while (n != null) {
            // 2.获取该指定数据节点的前一个节点
            while (n.right != null && n.right.val < num) {
                n = n.right;
            }
            // 2.与当前层链表断开
            Node right = n.right; // 要删除的节点
            if (right != null && right.val == num) {
                n.right = right.right;
                right.right = null; // help GC
                exist = true;
            }
            // 删除下一层
            n = n.down;
        }
        return exist;
    }
```

# 步骤二：实现有索引的插入和删除

## java代码

```
class Skiplist {
    final int HEAD_VALUE = -1;  // 链表头节点的值
    final Node HEAD = new Node(HEAD_VALUE);

    Node head;            // 最左上角的头节点，所有操作的开始位置
    int levels;           // 当前层级，即 head 节点所在的最高层数
    int length;           // 跳表长度，即原链表节点个数

    public Skiplist() {
        head = HEAD;
        levels = 1;
        length = 1;       // 仅包含头节点
    }

    class Node {
        int val;
        Node right, down;

        Node(int val) {
            this(val, null, null);
        }

        Node(int val, Node right, Node down) {
            this.val = val;
            this.right = right;
            this.down = down;
        }
```

```java
    }

    /**
     * 插入节点。将节点插入到原链表中正确的排序位置。
     *
     * 1.定位插入位置：原链表中 >= num 的最小节点前
     * 2.插入新节点
     * 3.根据扔硬币决定（是否）生成索引
     *
     * @param num
     */
    public void add(int num) {
        // 1.定位插入位置：原链表中 > num 的最小节点前
        Node node = head; // 从 head 开始查找
        // 节点向下，可能是生成索引的位置，使用数组记录这些节点
        Node[] nodes = new Node[levels];
        int i = 0; // 操作上述数组
        while (node != null) { // node==null 时，到达原链表
            // 在同一层级上向右查找，直到链表结尾，或者找到
            while (node.right != null && node.right.val < num) {
                node = node.right;
            }
            // 右侧为结尾 or 右侧值大 or 右侧值相同
            nodes[i++] = node;
            // 继续查找下一层的位置
            node = node.down;
        }

        // 2.插入新节点
        node = nodes[--i]; // nodes中最后一个元素
        Node newNode = new Node(num, node.right, null);
        node.right = newNode;
        length++; // 每添加一个节点，长度加 1

        // 3.根据扔硬币决定（是否）生成索引
        addIndicesByCoinFlip(newNode, nodes, i); // i 的值代表索引层数，不包含原链表
    }

    /**
     * 抛硬币的方式决定是否给新节点建立索引。
     * 索引层级可能超出现有跳表的层数，再抛一次决定是否生成索引。
     * 1.抛硬币，在现有跳表层数范围内建立索引
     * 2.抛硬币，决定是否建立一层超出跳表层数的索引层
     *
     * @param target 新节点
     * @param nodes 可能在这些节点后添加新索引节点
     * @param indices 当前索引层数
     */
    private void addIndicesByCoinFlip(Node target, Node[] nodes, int indices) {
        Node downNode = target;
        Random random = new Random();
        int coins = random.nextInt(2); // 0 or 1, 50% 概率
        // 1.抛硬币，在现有跳表层数范围内建立索引
        while (coins == 1 && levels < (length >> 6)) {
            if (indices > 0) {
                Node prev = nodes[--indices]; // 数组的倒数第二个元素，level 2
                Node newIndex = new Node(target.val, prev.right, downNode);
                prev.right = newIndex;
                downNode = newIndex;
                coins = random.nextInt(2);
            } else { // 新建一个索引层级
                // 新建索引节点和 head 节点
                Node newIndex = new Node(target.val, null, downNode);
                Node newHead = new Node(HEAD_VALUE, newIndex, head);
                head = newHead; // head 指针上移
                levels++; // 跳表层数加 1
            }
        }
    }

    /**
     * 从 head 开始，从左到右、从上到下依次查找
     * 1.小于，往右
     * 2.相同，则返回
     * 3.链表结尾，或大于，往下
     *
     * @param target
     * @return
     */
    public boolean search(int target) {
        Node n = get(target, head);
        return n != null;
    }

    /**
     * 遍历跳表，查找与给定值相同的节点，删除每一层
     * 1.获取该指定数据节点的前一个节点
     * 2.与当前层链表断开
     * 3.下移，删除每一层
     *
     * @param num
     * @return
     */
    public boolean erase(int num) {
        boolean exist = false;
        Node node = get(num, head);
        while (node != null) {
            Node right = node.right; // 要删除的节点
            node.right = right.right;
            right.right = null; // help GC
            exist = true;

            node = get(num, node.down);
        }
        if (exist) {
            length--; // 每删除一个节点，长度减 1
        }
        return exist;
    }

    public Node get(int target, Node from) {
        Node n = from;
        while (n != null) {
            // 1.在同一层级上向右查找，直到链表结尾，或者找到
            while (n.right != null && n.right.val < target) {
                n = n.right;
            }
            // 2.若找到，返回true
            Node right = n.right; // 要查找的节点
            if (right != null && right.val == target) {
                return n; // 返回要查找的节点的前一个
            }
            // 3.若右侧数据较大，向下一层
            n = n.down;
        }
        return null;
    }
}
```

## C++代码

```
```

## Python代码

```
'''
执行结果：通过。显示详情
执行用时：300 ms, 在所有 Python3 提交中击败了91.53%的用户
内存消耗：20.9 MB, 在所有 Python3 提交中击败了87.93%的用户
'''
import math
from random import random

class Node:
    def __init__(self, val, right, down):
        self.val = val
        self.right = right
        self.down = down

class Skiplist:
    _HEAD_VALUE = -1  # 链表头节点的值
    head = None  # 最左上角的头节点，所有操作的开始位置

    def __init__(self):
        self.head = Node(self._HEAD_VALUE, None, None)
        self.levels = 1  # 当前层级，即 head 节点所在的最高层数
        self.length = 1  # 节点所在的最高层数

    '''
    从 head 开始，从左到右、从上到下依次查找
    '''
    def search(self, target: int) -> bool:
        return self.get(target, self.head) is not None

    '''
    插入节点。将节点插入到原链表中正确的排序位置。
    1.定位插入位置：原链表中 >= num 的最小节点前
    2.插入新节点
    3.根据扔硬币决定（是否）生成索引
    '''
    def add(self, num: int) -> None:
        # 1.定位插入位置：原链表中 > num 的最小节点前
        node = self.head
        nodes = []  # 定义列表存储在其后可能生成索引的节点
        while node is not None:
            # 在同一层级上向右查找，直到链表结尾，或者找到
            while node.right is not None and node.right.val < num:
                node = node.right
            # 右侧为结尾 or 右侧值大 or 右侧值相同
            nodes.append(node)
            # 继续查找下一层的位置
            node = node.down

        # 2.插入新节点
        node = nodes[-1]  # nodes中最后一个元素，在底层的原链表中
        newNode = Node(num, node.right, None)
        node.right = newNode
        self.length = self.length + 1

        # 3.根据扔硬币决定（是否）生成索引
        self.addIndicesByCoinFlip(newNode, nodes, len(nodes) - 1)

    '''
    抛硬币决定是否生成索引
    索引层级可能超出现有跳表的层数，再抛一次决定是否生成索引。
        1.抛硬币，在现有跳表层数范围内建立索引
        2.抛硬币，决定是否建立一层超出跳表层数的索引层
    索引层数不超过原链表数据量的对数（以2为底）
    '''
    def addIndicesByCoinFlip(self, target: Node, nodes: [], indices: int):
        if self.length < 8: # 8个元素以内，无须创建索引
            return
        downNode = target
        # 抛硬币，在现有跳表层数范围内建立索引：0 or 1, 50% 概率
        while (coins := random.randint(0, 1)) == 1 \
                and (self.levels - 1) < int(math.log(self.length, 2)):  # 索引层数 < O(logN)
            if indices > 0:
                indices = indices - 1
                prev = nodes[indices]  # 列表的倒数第二个元素，level 2
                newIndex = Node(target.val, prev.right, downNode)
                prev.right = newIndex
                # 下一个索引将建立在当前生成的新索引上
                downNode = newIndex
            else:
                newIndex = Node(target.val, None, downNode)
                newHead = Node(self._HEAD_VALUE, newIndex, self.head)
                self.head = newHead  # head 指针上移
                self.levels = self.levels + 1  # 跳表层数加 1

    '''
    遍历跳表，查找与给定值相同的节点，删除每一层
    1.获取该指定数据节点的前一个节点
    2.与当前层链表断开
    3.下移，删除每一层
    '''
    def erase(self, num: int) -> bool:
        exist = False
        # 1.获取该指定数据节点的前一个节点
        node = self.get(num, self.head)
        while node is not None:
            r = node.right  # 要删除的节点
            # 2.与当前层链表断开
            node.right = r.right
            r.right = None
            exist = True
            # 3.下移，删除每一层
            node = self.get(num, node.down)
        return exist

    '''
    获取指定target的前一个节点，
    从fromm节点开始查找
    '''
    def get(self, target: int, fromm: Node) -> Node:
        n = fromm
        while n is not None:
            # 1.在同一层级上向右查找，直到链表结尾，或者找到
            while n.right is not None and n.right.val < target:
                n = n.right
            # 2.若找到，返回true
            right = n.right  # 要查找的节点
            if right is not None and right.val == target:
                return n  # 返回要查找的节点的前一个
            # 3.若右侧数据较大，向下一层
            n = n.down
        return None
```

## 测试用例

```
操作：
["Skiplist","add","add","add","search","add","search","erase","erase","search"]
输入：[[],[1],[2],[3],[0],[4],[1],[0],[1],[1]]
输出：[null,null,null,null,false,null,true,false,true,false]
```

```
操作：
["Skiplist","add","add","add","add","add","erase","erase","add","search","search
","add","erase","search","add","add","add","erase","search","erase","search","se
arch","search","erase","erase","search","erase","add","add","erase","add","searc
h","search","search","search","search"]
输入：[[],[9],[4],[5],[6],[9],[2],[1],[2],[7],[4],[5],[6],[5],[6],[7],[4],[3],[6],
[3],[4],[3],[8],[7],[6],[7],[4],[1],[6],[3],[4],[7],[6],[1],[0],[3]]
输出：
[null,null,null,null,null,null,false,false,null,false,true,null,true,true,null,n
ull,null,false,true,false,true,false,false,true,true,false,true,null,null,false,
null,false,true,true,false,false]
```

```
操作：
["Skiplist","add","add","add","add","add","add","add","add","add","erase","searc
h","add","erase","erase","erase","add","search","search","search","erase","searc
h","add","add","add","erase","search","add","search","erase","search","search","
erase","erase","add","erase","search","erase","erase","search","add","add","eras
e","erase","erase","add","erase","add","erase","erase","add","add","add","search
","search","add","erase","search","add","add","search","add","search","erase","e
rase","search","search","erase","search","add","erase","search","erase","search"
,"erase","erase","search","search","add","add","add","add","search","search","se
arch","search","search","search","search","search","search"]
输入：[[],[16],[5],[14],[13],[0],[3],[12],[9],[12],[3],[6],[7],[0],[1],[10],[5],
[12],[7],[16],[7],[0],[9],[16],[3],[2],[17],[2],[17],[0],[9],[14],[1],[6],[1],
[16],[9],[10],[9],[2],[3],[16],[15],[12],[7],[4],[3],[2],[1],[14],[13],[12],[3],
[6],[17],[2],[3],[14],[11],[0],[13],[2],[1],[10],[17],[0],[5],[8],[9],[8],[11],
[10],[11],[10],[9],[8],[15],[14],[1],[6],[17],[16],[13],[4],[5],[4],[17],[16],
[7],[14],[1]]
输出：
[null,null,null,null,null,null,null,null,null,null,true,false,null,true,false,fa
lse,null,true,true,true,true,false,null,null,null,false,false,null,false,false,t
rue,true,false,false,null,true,true,false,true,true,null,null,false,true,false,n
ull,true,null,true,true,null,null,null,false,false,null,true,false,null,null,tru
e,null,false,false,false,true,true,false,true,null,true,false,false,false,true,t
rue,false,false,null,null,null,null,true,true,true,true,true,true,false,false,tr
ue]
```