【第二十二周】手撕红黑树(上)

981. 基于时间的键值存储

主要是选择用Map数据结构存储键值对。

```
/**
 * Initialize your data structure here.
 */
var TimeMap = function () {
    this.m = new Map();
};

/**
 * @param {string} key
 * @param {string} value
 * @param {number} timestamp
 * @return {void}
 */
TimeMap.prototype.set = function (key, value, timestamp) {
    this.m.has(key) ? this.m.get(key).push([value, timestamp]) : this.m.set(key,
[[value, timestamp]]);
};

/**
 * @param {string} key
 * @param {number} timestamp
 * @return {string}
 */
TimeMap.prototype.get = function (key, timestamp) {
    if (!this.m.has(key)) return '';
    let h = this.m.get(key), l = 0, r = h.length - 1, m;
    while (l <= r) {
        m = l + r >> 1;
        if (h[m][1] <= timestamp) l = m + 1;
        else r = m - 1;
    }
    return r < 0 ? '' : h[r][0];
};

/**
 * Your TimeMap object will be instantiated and called as such:
 * var obj = new TimeMap()
 * obj.set(key,value,timestamp)
 * var param_2 = obj.get(key,timestamp)
 */
```

971. 翻转二叉树以匹配先序遍历

1、题目给了一个待操作的树 A 的根节点 root， 和树 voyage 先序遍历的数组，然后求的是 A 能否翻转最少的 n 个节点，使得 A 和 voyage 一致。

2、首先开始前序遍历树 voyage，然后用 voyage 的值去匹配 A， 看看能否进行树的匹配。

3、对于一次遍历，我们首先判断根节点的值是一致，才会进入这一次的遍历中，然后主要是看左右树，对于树voyage 来说，用 pos 不断按照先序遍历给出值，而对于voyage 来说，你可以用左树匹配，如果左树不匹配，用右树先行，然后再走左树，这种情况就需要翻转一下当前的节点，保证树 A 要在前序遍历的情况和voyage 匹配。

4、如果在匹配过程中 A 的左右树都没有匹配成功，则会提取走出 A 的遍历，这个时候 pos 就没有迭代完，这个时候就是异常，返回 [-1]

```
var flipMatchVoyage = function(root, voyage) {
    if(root.val !== voyage[0]) return [-1];
    const ret = [];
    let pos = 0;
    const dfs = root => {
         pos++ ;
        if(root.left && root.left.val === voyage[pos] ){
            dfs(root.left);
        }
        if(root.right && root.right.val === voyage[pos] ){
            dfs(root.right) ;
            if(root.left && root.left.val === voyage[pos] ){
                ret.push(root.val);
                dfs(root.left);
            }
        }
    }

    dfs(root);
    if(pos<voyage.length){
        return [-1];
    }

    return ret;
};
```

1339. 分裂二叉树的最大乘积

1、首先计算所有节点和值，得出中位数

2、递归去遍历二叉树每一颗子树，去求每一棵子树的和值

3、枚举最大的乘积：当前拆分的子树越接近平均值，乘积越大

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
```

```
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number}
 */
let avg,ans = 0;
var maxProduct = function(root) {
    let total = getTotal(root);
    avg = total / 2;
    ans = total;
    getTotal(root);
    return ans * (total - ans) % (10 ** 9 + 7)
};

var getTotal = function(root){
    if(root == null) return 0;
    let val = root.val + getTotal(root.left) + getTotal(root.right);
    if(Math.abs(val - avg) < Math.abs(ans - avg)) ans = val;
    return val;
}
```

449. 序列化和反序列化二叉搜索树

1、利用前序遍历将数值存在一个数组中，然后序列化的时候加上空格分割数值，这样就完成了序列化操作

2、根据二叉搜索树的性值：中序遍历的结果是从小到大有序的，所以可以根据前序遍历的结果，获取到前序遍历的数组，然后对数组进行 sort() 排序后，就得到了中序遍历的结果数组，这样利用前序和中序遍历的结果数组，就可以根据数据构建一颗二叉树。

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */

/**
 * Encodes a tree to a single string.
 *
 * @param {TreeNode} root
 * @return {string}
 */
var serialize = function(root) {
    if (root === null) {
        return '';
    }
    let stringArray = [];
```

```
    var postDFS = function (node) {
        if (node === null) {
            return;
        }
        stringArray.push(node.val)
        postDFS(node.left);
        postDFS(node.right);
    }
    postDFS(root)
    return stringArray.join(' ');
};

/**
 * Decodes your encoded data to tree.
 *
 * @param {string} data
 * @return {TreeNode}
 */
var deserialize = function(data) {
    if (data.length === 0) {
        return null;
    }
    let preorder = data.split(' ').map(item => {
        return Number.parseInt(item);
    })
    let inorder = [...preorder];
    inorder.sort((a, b) => {
        return a - b;
    })
    let preLen = preorder.length ;
    let inLen = inorder.length;
    let map = new Map();
    for (let i = 0; i < inLen; i++) {
        map.set(inorder[i], i);
    }

    const build = function (preorder, preLeft, preRight, map, inLeft, inRight) {
        if (preLeft > preRight || inLeft > inRight) {
            return null;
        }
        let root = new TreeNode(preorder[preLeft]);
        let pIndex = map.get(root.val);
        root.left = build(preorder, preLeft + 1, pIndex - inLeft + preLeft, map,
inLeft, pIndex - 1);
        root.right = build(preorder, pIndex - inLeft + preLeft + 1, preRight,
map, pIndex + 1, inRight);
        return root;
    }
    return build(preorder, 0, preLen - 1, map, 0, inLen - 1);
};

/**
 * Your functions will be called as such:
 * deserialize(serialize(root));
 */
```

220. 存在重复元素 III

1、我们按照元素的大小进行分桶，维护一个滑动窗口内的元素对应的元素。

2、对于元素 x，其影响的区间为 [x - t, x + t]。于是我们可以设定桶的大小为 t+1。如果两个元素同属一个桶，那么这两个元素必然符合条件。

3、如果两个元素属于相邻桶，那么我们需要校验这两个元素是否差值不超过 *t*。如果两个元素既不属于同一个桶，也不属于相邻桶，那么这两个元素必然不符合条件。

4、我们遍历该序列，假设当前遍历到元素 x，那么我们首先检查 x 所属于的桶是否已经存在元素，如果存在，那么我们就找到了一对符合条件的元素，否则我们继续检查两个相邻的桶内是否存在符合条件的元素。

```
/**
 * @param {number[]} nums
 * @param {number} k
 * @param {number} t
 * @return {boolean}
 */
var containsNearbyAlmostDuplicate = function (nums, k, t) {
    function getId(x) {
        return Math.floor(x / (t + 1));
    }
    if (t < 0) return false;

    const map = new Map();

    for (let i = 0; i < nums.length; i++) {
        const m = getId(nums[i]);
        if (map.has(m)) {
            return true;
        } else if (map.has(m + 1) && Math.abs(map.get(m + 1) - nums[i]) <= t) {
            return true;
        } else if (map.has(m - 1) && Math.abs(map.get(m - 1) - nums[i]) <= t) {
            return true;
        }
        map.set(m, nums[i]);
        if (i >= k) {
            map.delete(getId(nums[i - k]));
        }
    }
    return false;
};
```

开课吧