Tree

deepwaterooo

October 8, 2024

Contents

Chapter 1

Tree 树结构：各种新型数据结构

1.1 最大深度到树的最大直径（不一定经过根节点）

- **【亲爱的表哥的活宝妹，任何时候，亲爱的表哥的活宝妹，就是一定要、一定会嫁给活宝妹的亲爱的表哥!!! 爱表哥，爱生活!!!】**
- 【亲爱的表哥的活宝妹，现在再看、最大直径，它，狠简单】
- 现在的亲爱的表哥的活宝妹，再看这些破烂小题目，太简单了。**文件要删除一些不必要的简单题目去，把文件弄小【TODO】**:

1.1.1 104. Maximum Depth of Binary Tree

- Given the root of a binary tree, return its maximum depth.
- A binary tree's maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

1. 深度优先搜索

```java
// DFS: 深度优先遍历
public int maxDepth(TreeNode r) {
    if (r == null) return 0;
// 下面这个 CASE: 仔细想也是被上面的包括的，所以可以不用
    if (r.left == null && r.right == null) return 1; // 这句可以不用要
    int left = maxDepth(r.left);
    int right = maxDepth(r.right);
// 下面这一行包括上面那一句，所以上面那一句可以不用
    return Math.max(left, right) + 1;
}
```

2. 广度优先搜索

```java
// 广度优先遍历
public int maxDepth(TreeNode r) {
    if (r == null) return 0;
    int cnt = 0;
    Deque<TreeNode> q = new ArrayDeque<>();
    q.offerFirst(r);
    while (!q.isEmpty()) {
        int qsize = q.size();
        for (int size = qsize-1; size >= 0; size--) {
            TreeNode cur = q.pollLast();
            if (cur.left != null) q.offerFirst(cur.left);
            if (cur.right != null) q.offerFirst(cur.right);
        }
        cnt++;
    }
    return cnt;
}
```

1.1.2 543. Diameter of Binary Tree

- Given the root of a binary tree, return the length of the diameter of the tree.
- The diameter of a binary tree is the length of the longest path between any two nodes in a tree. This path may or may not pass through the root.

- The length of a path between two nodes is represented by the number of edges between them.

```
public int diameterOfBinaryTree(TreeNode r) {
    traverse(r);
    return max;
}
int max = 0;
int traverse(TreeNode r) {
    if (r == null) return 0; // 叶子节点会是 1，成了数节点个数
    int left = traverse(r.left);
    int right = traverse(r.right);
    max = Math.max(max, left + right);
    return Math.max(left, right) + 1;
}
```

1.1.3 2385. Amount of Time for Binary Tree to Be Infected

- You are given the root of a binary tree with unique values, and an integer start. At minute 0, an infection starts from the node with value start.
- Each minute, a node becomes infected if:
- The node is currently uninfected.
- The node is adjacent to an infected node.
- Return the number of minutes needed for the entire tree to be infected.

1. 提示的思路如下：
    - Convert the tree to an undirected graph to make it easier to handle.
    - Use BFS starting at the start node to find the distance between each node and the start node. The answer is the maximum distance.
    - 感觉因为树的题目简单，做题的时候只埋头做题了，但是没有很好地总结；思路比较死，可能理解得不是很透彻，所以解题的时候还不太能够灵活应用。
2. BFS 简洁精炼版的

```
public int amountOfTime(TreeNode r, int start) {
    buildGraph(r, null);
// 下面这里注意：两个数据结构的初始化方式
    return maxDistance(new ArrayDeque<>(List.of(start)), new HashSet<>(Set.of(start)));
}
Map<Integer, List<Integer>> m = new HashMap<>();
int maxDistance(Deque<Integer> q, HashSet<Integer> vis) {
    int cnt = 0;
    while (!q.isEmpty()) {
        int size = q.size();
        while (size-- > 0) {
            int u = q.pollLast();
            vis.add(u);
            if (m.get(u) == null) continue;
            for (int v : m.get(u)) {
                if (vis.contains(v)) continue;
                q.offerFirst(v);
            }
        }
        cnt++;
    }
    return cnt-1;
}
void buildGraph(TreeNode r, TreeNode p) { // 我还是比较喜欢自己这种相对简洁的方法
    if (r == null) return ;
// 这里是，从当前节点，加父节点，加左右两个子节点；简洁，但是需要父节点作为参数
    m.computeIfAbsent(r.val, z -> new ArrayList<>());
    if (p != null) m.get(r.val).add(p.val);
    if (r.left != null) m.get(r.val).add(r.left.val);
    if (r.right != null) m.get(r.val).add(r.right.val);
    buildGraph(r.left, r);
    buildGraph(r.right, r);
}
```

3. DFS 简洁精炼版的

```
public int amountOfTime(TreeNode r, int start) {
    buildGraph(r, null);
    maxDistance(new HashSet<>(), start, 0);
    return max;
}
```

```
    Map<Integer, List<Integer>> m = new HashMap<>();
    int max = 0;
    void maxDistance(Set<Integer> vis, int u, int d) {
        if (vis.contains(u)) return ;
        vis.add(u);
        max = Math.max(max, d);
        for (int v : m.get(u))
            maxDistance(vis, v, d+1);
    }
    void buildGraph(TreeNode r, TreeNode p) { // 我还是比较喜欢自己这种相对简洁的方法
        if (r == null) return ;
    // 这里是，从当前节点，加父节点，加左右两个子节点；简洁，但是需要父节点作为参数
        m.computeIfAbsent(r.val, z -> new ArrayList<>());
        if (p != null) m.get(r.val).add(p.val);
        if (r.left != null) m.get(r.val).add(r.left.val);
        if (r.right != null) m.get(r.val).add(r.right.val);
        buildGraph(r.left, r);
        buildGraph(r.right, r);
    }
    void buildGraph(TreeNode r) {
        if (r == null) return ;
        m.computeIfAbsent(r.val, z -> new ArrayList<>());
    // 从左右两个确定存在的子节点，加父节点
        if (r.left != null) {
            m.computeIfAbsent(r.left.val, z -> new ArrayList<>()).add(r.val);
            m.get(r.val).add(r.left.val); // 从当前节点，加左右两个子节点；
            buildGraph(r.left);
        }
        if (r.right != null) {
            m.computeIfAbsent(r.right.val, z -> new ArrayList<>()).add(r.val);
            m.get(r.val).add(r.right.val); // 从当前节点，加左右两个子节点；
            buildGraph(r.right);
        }
    }
```

1.1.4 979. Distribute Coins in Binary Tree

You are given the root of a binary tree with n nodes where each node in the tree has node.val coins. There are n coins in total throughout the whole tree.

In one move, we may choose two adjacent nodes and move one coin from one node to another. A move may be from parent to child, or from child to parent.

Return the minimum number of moves required to make every node have exactly one coin.

```
private int dfs(TreeNode r) { // 统计把自身，左右子树都平衡，需要移动的 coins 个数
    if (r == null) return 0;
    int left = dfs(r.left);      // 左、右子树缺多少
    int right = dfs(r.right);
    res += Math.abs(left) + Math.abs(right); // 左，右子树和自身都平衡需要的移动数
    return left + right + r.val-1;
}
int res;
public int distributeCoins(TreeNode root) {
    res = 0;
    return res;
}
```

1.1.5 1719. Number Of Ways To Reconstruct A Tree - Hard

You are given an array pairs, where pairs[i] = [xi, yi], and:

There are no duplicates. xi < yi Let ways be the number of rooted trees that satisfy the following conditions:

The tree consists of nodes whose values appeared in pairs. A pair [xi, yi] exists in pairs if and only if xi is an ancestor of yi or yi is an ancestor of xi. Note: the tree does not have to be a binary tree. Two ways are considered to be different if there is at least one node that has different parents in both ways.

Return:

0 if ways = 0 1 if ways = 1 2 if ways > 1 A rooted tree is a tree that has a single root node, and all edges are oriented to be outgoing from the root.

An ancestor of a node is any node on the path from the root to that node (excluding the node itself). The root has no ancestors.

1. 解题思路与分析

```
    public int checkWays(int[][] pairs) { // 自顶向下
        int max = 0; // [1, 500]
        for (int [] p : pairs) // 求出节点的最大值
            max = Math.max(max, Math.max(p[0], p[1]));
        int [] cnt = new int [max+1]; // 记录每个节点的祖先关系数量
        int [][] adj = new int [max+1][max+1]; // 是否存在祖孙关系的图
        for (int [] p : pairs) {
```

```
        cnt[p[0]]++;
        cnt[p[1]]++;
        adj[p[0]][p[1]] = 1;
        adj[p[1]][p[0]] = 1;
    }
    Integer [] nodes = new Integer [max+1]; // 创建一个新的数组，可以方便后面的按祖先关系数量大小将节点排序，和将零散的节点集中到前面。
    int n = 0; // 使用包装整数类型，方便后面调用 API 排序
    for (int i = 1; i <= max; i++)
        if (cnt[i] > 0) nodes[n++] = i;
    Arrays.sort(nodes, 0, n, (a, b)->cnt[b] - cnt[a]); // 按照祖先关系数量从大到小排序
    if (cnt[nodes[0]] != n-1) return 0; // 当根节点不满足要求
    int [] par = new int [max+1];
    int [][] allPar = new int [max+1][max+1];
    for (int i = 0; i < n; i++)
        for (int j = i-1; j >= 0; j--)
            if (adj[nodes[i]][nodes[j]] == 1) {
                par[nodes[i]] = nodes[j]; // 记录父节点
                for (int f = nodes[j]; f != 0; f = par[f]) // 自底向上：向祖先节点遍历，记录祖先节点，循环遍历直到根节点
                    allPar[nodes[i]][f] = 1;
                break; // 父节点只有一个，已经找到一个合法父节点，并且更新了所有的父节点，就可以不用再遍历了
            }
    int ans = 1;
    for (int i = 1; i <= max; i++)
        for (int j = i+1; j <= max; j++) {
            if (adj[i][j] == 1 && cnt[i] == cnt[j]) ans = 2; // 可以调换位置，有多个解
            if (adj[i][j] != (allPar[i][j] | allPar[j][i]))
                return 0; // 有冲突，无解，出现在已经记录了当前节点和祖先节点的关系，但是 pairs 中没有该关系
        }
    return ans;
}
```

2. 解题思路与分析: dfs: 这个方法好慢

```
public int checkWays(int[][] pairs) { // 这个方法好慢
    for (int [] p : pairs) {
        adj.computeIfAbsent(p[0], z -> new HashSet<>()).add(p[1]);
        adj.computeIfAbsent(p[1], z -> new HashSet<>()).add(p[0]);
    }
    return helper(adj.keySet());
}
Map<Integer, Set<Integer>> adj = new HashMap<>();
int helper(Set<Integer> nodes) {
    Map<Integer, List<Integer>> lenMap = new HashMap<>();
    for (Integer v : nodes)
        lenMap.computeIfAbsent(adj.get(v).size(), z -> new ArrayList<>()).add(v);
    if (!lenMap.containsKey(nodes.size()-1)) return 0; // 不存在合法的根节点
    Integer root = lenMap.get(nodes.size()-1).get(0);  // 这个任命为根的节点是否带有随机性？ lenMap 里 key 为 nodes.size()-1 的值应该只有一个
    for (Integer v : adj.get(root)) // 因为需要 dfs 自顶向下深度遍历，这些东西需要移掉
        adj.get(v).remove(root);
    Set<Integer> vis = new HashSet<>();
    Set<Set<Integer>> group = new HashSet<>(); // 以每个节点作为根节点的子树子节点集合
    for (Integer v : nodes)
        if (!v.equals(root) && !vis.contains(v)) {
            Set<Integer> cur = new HashSet<>();
            dfs(vis, v, cur);
            group.add(cur);
        }
    int ans = lenMap.get(nodes.size()-1).size() > 1 ? 2 : 1; // 如果根节点不止不一个，就可能有并行答案
    for (Set<Integer> g : group) { // 自顶向下：遍历根节点下每个节点的建树是否合法、是否唯一
        int tmp = helper(g);
        if (tmp == 0) return 0; // 不存在合法的根节点
        if (tmp == 2) ans = 2;
    }
    return ans;
}
private void dfs(Set<Integer> vis, int node, Set<Integer> cur) {
    vis.add(node);
    cur.add(node);
    for (int next : adj.get(node))
        if (!vis.contains(next))
            dfs(vis, next, cur);
}
```

3. 解题思路与分析

```
public int checkWays(int [][] pairs) {
    Map<Integer, Integer> cnt = new HashMap<>(); // 统计结点对中各个结点出现的次数
    Map<Integer, List<Integer>> adj = new HashMap<>();
    for (int [] pair : pairs) {
        int from = pair[0], to = pair[1];
        cnt.put(from, cnt.getOrDefault(from, 0) + 1);
        cnt.put(to, cnt.getOrDefault(to, 0) + 1);
        adj.computeIfAbsent(from, x -> new ArrayList<>()).add(to);
        adj.computeIfAbsent(to, x -> new ArrayList<>()).add(from);
    }
    List<Integer> list = new ArrayList<>(cnt.keySet()); // list of ori nodes 将结点对中的结点存储在 List 集合中
    list.sort((a, b) -> cnt.get(b) - cnt.get(a)); // 对 list 集合进行排序
    // pairs 中给出了树中所有具有祖孙关系的结点对，很显然，根节点是其他所有结点的祖先
```

```
        // 所以根结点在 pairs 出现的次数应该为为总结点数-1，找不到符合这个关系的结点，那就不符合题目中构树的要求
        if (cnt.get(list.get(0)) != list.size() - 1) return 0;
        // 判断已排序后的结点集合是否有两个结点具有相同出现次数，如果存在，那么这两个结点可以互换，即为两颗树
        int ans = 1;
        for (int [] p : pairs)
            if (cnt.get(p[0]).equals(cnt.get(p[1]))) {
                ans = 2;
                break;
            }
        // 将所有结点的父结点置为出现结点最多的结点，即根结点
        // 在没有确定除根结点之外的其它结点真正父结点之前，根结点就是它们的祖先
        Map<Integer, Integer> farMap = new HashMap<>();
        Set<Integer> set = new HashSet<>(); // 存储所有父结点
        set.add(list.get(0));
        for (Integer i : list) //
            farMap.put(i, list.get(0));
        // 处理除最大结点数外，按着构树规则处理其它结点
        for (int i = 1; i < list.size(); ++i) {
            for (Integer s : adj.get(list.get(i)))
                // 判断当前结点是否为父结点
                if (!set.contains(s)) {
                    // 如果 s 不是父结点，那么就是当前 list.get(i) 结点的子结点
                    // 在没有更新父结点之前，s 的父结点和 list.get(i) 的父结点是相同的 (父子在一条链上)
                    // 如果父结点不相同，可以理解为 s 的父结点 list.get(i) 有多个父结点，显然是不合理的
                    //  同样也可以把树理解为图，除根结点之外，所有结点的入度都为 1，而上边的情况表示存在一个入度为 2 的结点
                    // 明显与树的构建原理相悖
                    if (farMap.get(s) != farMap.get(list.get(i)))
                        return 0;
                    farMap.put(s, list.get(i));
                }
            set.add(list.get(i));
        }
        return ans;
    }
```

1.1.6 1766. Tree of Coprimes - Hard

There is a tree (i.e., a connected, undirected graph that has no cycles) consisting of n nodes numbered from 0 to n - 1 and exactly n - 1 edges. Each node has a value associated with it, and the root of the tree is node 0.

To represent this tree, you are given an integer array nums and a 2D array edges. Each nums[i] represents the ith node's value, and each edges[j] = [uj, vj] represents an edge between nodes uj and vj in the tree.

Two values x and y are coprime if gcd(x, y) == 1 where gcd(x, y) is the greatest common divisor of x and y.

An ancestor of a node i is any other node on the shortest path from node i to the root. A node is not considered an ancestor of itself.

Return an array ans of size n, where ans[i] is the closest ancestor to node i such that nums[i] and nums[ans[i]] are coprime, or -1 if there is no such ancestor.

1. 解题思路与分析

- 切入点和解题思路
  - 如果用蛮力检查一个节点的所有的祖先节点，那么，一个节点的祖先节点最多能有 n-1n1 个，显然会超时的。
  - 一个重要的切入点是：nums[i] ≤ 50nums[i]50。我们不妨换一种思路：从节点的值 xx 出发，枚举满足 1 ≤ y ≤ 501y50 且 gcd(x,y) = 1gcd(x,y)=1 的 yy，并对每个 yy 找出离着节点 ii 最近的点，最后再在这些点中求出离着当前点最近的点即可。这样只需检查 5050 次即可。
  - 那么，如何对于任一数字 yy，找出离当前节点 ii 最近的祖先节点呢？首先可以想到的是，离着节点 ii 最近的满足条件的祖先节点，也是这些点中最深的。我们不妨对每个数字 1 ～ 50150 维护一个栈，并采用 dfs 的思路。每当我们要遍历下一个节点时，就把当前节点的编号（nodenode）和节点的深度（levellevel）push 到当前节点的值（xx）对应的栈中。这样，栈顶就是数字 xx 的、最深的节点，也是我们之后需要的关于数字 xx 的最近的节点。此外，要记得 dfs 完成后要将之前 push 进去的元素 pop 出来。
- 解题思路
  - 1、邻接表建立，表示每个节点关联的节点
  - 2、准备 50 个栈，以每个节点的数据值为基准，栈内存储的数据为当前数据值对应的层数及节点 i 标识
  - 3、遍历到某个节点时，以当前节点为基准，满足 gcd 条件并且层数最深的为最优解，也就是最近公共祖先节点
  - 4、满足 gcd 条件可能存在多个节点的数据值，遍历可能的数据值里面，离节点 i 最近的，通过 level 来识别；这里需要识别数值和 level 两重条件
  - 5、为啥取栈顶的元素呢，因为我们压栈的时候，level 最大的总是在栈顶的，而这里只需要相同数值里面 level 最大的即可，因为每轮遍历实际是从根节点到当前节点的，所以计算当前节点时，stack 里存储的应该是所有的祖先节点，只需要在所有祖先节点里面取最近的即可

```
    public int[] getCoprimes(int[] a, int[][] edges) {
        cop = new boolean [51][51];
        for (int i = 1; i < 51; i++)
            for (int j = 1; j < 51; j++)
                if (!cop[i][j] && gcd(i, j) == 1) {
                    cop[i][j] = true;
                    cop[j][i] = true;
                }
        int n = a.length;
        li = new ArrayList[n];
        for (int i = 0; i < n; i++) li[i] = new ArrayList<>();
        for (int [] e : edges) {
            li[e[0]].add(e[1]);
            li[e[1]].add(e[0]);
        }
        ans = new int [n];
        for (int i = 0; i < 51; i++)
            st[i] = new ArrayDeque<>();
        dfs(0, -1, 0, a);
        return ans;
    }
    List<Integer>[] li;
    ArrayDeque<int []> [] st = new ArrayDeque[51];
    boolean [][] cop;
    int [] ans;
    void dfs(int node, int pre, int level, int [] a) {
        int re = -1, lev = -1;
        for (int i = 1; i < 51; i++)
            if (st[i].size() > 0 && st[i].peekLast()[0] > lev && cop[i][a[node]]) {
                re = st[i].peekLast()[1];
                lev = st[i].peekLast()[0];
            }
        ans[node] = re;
        for (int next : li[node]) {
            if (next != pre) {
                st[a[node]].offerLast(new int [] {level, node});
                dfs(next, node, level + 1, a);
                st[a[node]].pollLast();
            }
        }
    }
    int gcd(int x, int y) {
        if (y == 0) return x;
        return gcd(y, x % y);
    }
```

1.1.7 1028. Recover a Tree From Preorder Traversal: 栈 + 迭代，递归 - Hard

We run a preorder depth-first search (DFS) on the root of a binary tree.

At each node in this traversal, we output D dashes (where D is the depth of this node), then we output the value of this node. If the depth of a node is D, the depth of its immediate child is D + 1. The depth of the root node is 0.

If a node has only one child, that child is guaranteed to be the left child.

Given the output traversal of this traversal, recover the tree and return its root.

1. 解题思路与分析: 栈 + 迭代

```
public TreeNode recoverFromPreorder(String t) {
    Deque<TreeNode> st = new LinkedList<TreeNode>();
    char [] s = t.toCharArray();
    int n = t.length();
    int idx = 0;
    while (idx < n) {
        int lvl = 0;
        while (s[idx] == '-') {
            ++lvl;
            ++idx;
        }
        int val = 0;
        while (idx < n && Character.isDigit(s[idx])) {
            val = val * 10 + (s[idx] - '0');
            ++idx;
        }
        TreeNode node = new TreeNode(val);
        if (lvl == st.size()) {
            if (!st.isEmpty())
                st.peekLast().left = node;
        } else {
            while (lvl != st.size())
                st.pollLast();
            st.peekLast().right = node;
        }
        st.offerLast(node);
    }
    while (st.size() > 1) st.pollLast();
```

```
        return st.peekLast();
    }
```

2. 解题思路与分析: 递归

   虽然博主最开始想的递归方法不太容易实现，但其实这道题也是可以用递归来做的，这里我们需要一个全局变量 cur，表示当前遍历字符串 S 的位置，递归函数还要传递个当前的深度 level。在递归函数中，首先还是要提取短杠的个数，但是这里有个很 tricky 的地方，我们在统计短杠个数的时候，不能更新 cur，因为 cur 是个全局变量，当统计出来的短杠个数跟当前的深度不相同，就不能再继续处理了，如果此时更新了 cur，而没有正确的复原的话，就会出错。博主成功入坑，检查了好久才找出原因。当短杠个数跟当前深度相同时，我们继续提取出结点值，然后新建出结点，对下一层分别调用递归函数赋给新建结点的左右子结点，最后返回该新建结点即可

```
    private int idx = 0; // 遍历 S 的全局指针
    public TreeNode recoverFromPreorder(String S) {
        if (S.isEmpty()) return null;
        return buildBinaryTree(S.toCharArray(), 0);
    }
    public TreeNode buildBinaryTree(char[] ss, int depth) {
        // 判定当前节点是否是 null
        if (idx + depth >= ss.length || isNullPointer(ss, depth)) return null;
        idx += depth; // idx 指针跳过 depth 个'-', 指向下一个节点的开始位置
        // 左右子树递归
        TreeNode root = new TreeNode(getValue(ss));
        root.left = buildBinaryTree(ss, depth + 1);
        root.right = buildBinaryTree(ss, depth + 1);
        // 返回当前节点
        return root;
    }
    // 获取当前节点的 val 值, 由于可能有多位, 需要遍历一下
    public int getValue(char[] ss) {
        int value = 0;
        while (idx < ss.length && ss[idx] != '-') {
            value = value * 10 + (ss[idx] - '0');
            idx ++;
        }
        return value;
    }
    // 判断当前位置的节点是不是 null
    public boolean isNullPointer(char[] ss, int depth) {
        for (int i = idx; i < idx + depth; i ++)
            if (ss[i] != '-') return true;
        return false;
    }
```

   - 下面是一个简洁版的代码

```
    public TreeNode recoverFromPreorder(String S) {
        if (S.isEmpty()) return null;
        n = S.length();
        return buildBinaryTree(S.toCharArray(), 0);
    }
    private int idx = 0, n; // 遍历 S 的全局指针
    TreeNode buildBinaryTree(char [] s, int level) {
        int cnt = 0, val = 0;
        while (idx + cnt < n && s[idx + cnt] == '-') ++cnt;
        if (cnt != level) return null;
        idx += cnt;
        for (; idx < n && s[idx] != '-'; idx++)
            val = val * 10 + s[idx] - '0';
        TreeNode r =  new TreeNode(val);
        r.left = buildBinaryTree(s, level + 1);
        r.right = buildBinaryTree(s, level + 1);
        return r;
    }
```

1.1.8 1932. Merge BSTs to Create Single BST

You are given n BST (binary search tree) root nodes for n separate BSTs stored in an array trees (0-indexed). Each BST in trees has at most 3 nodes, and no two roots have the same value. In one operation, you can:

Select two distinct indices i and j such that the value stored at one of the leaves of trees[i] is equal to the root value of trees[j]. Replace the leaf node in trees[i] with trees[j]. Remove trees[j] from trees. Return the root of the resulting BST if it is possible to form a valid BST after performing n - 1 operations, or null if it is impossible to create a valid BST.

A BST (binary search tree) is a binary tree where each node satisfies the following property:

Every node in the node's left subtree has a value strictly less than the node's value. Every node in the node's right subtree has a value strictly greater than the node's value. A leaf is a node that has no children.

```
public TreeNode canMerge(List<TreeNode> trees) {
    final int size = trees.size();
    final Map<Integer, TreeNode> roots = new HashMap<>(size);
    for (final TreeNode node : trees)
```

```
        roots.put(node.val, node);
    for (final TreeNode node : trees) {
        if (roots.containsKey(node.val)) { // 这里判断：是因为接下来 buildTree 会将可以合并的子树键值对删除并回收利用建大树了
            final TreeNode root = buildTree(roots, node);
            roots.put(root.val, root);    // update root node
        }
    }
    if (roots.size() != 1) return null;   // 无法合并所有的子树
    final TreeNode root = roots.values().iterator().next(); // 只有这一颗树根
    return isValid(root, Integer.MIN_VALUE, Integer.MAX_VALUE) ? root : null;
}
private TreeNode buildTree(Map<Integer, TreeNode> roots, TreeNode node) { // 用 recursion 把所有需要/可以合并的子树建成一棵完整大树，方法很传神
    final TreeNode next = roots.remove(node.val); // map.remove() 返回值：如果存在 key，则删除并返回 value；如果不存在则返回 null
    if (next != null) {
        if (next.left != null) node.left = buildTree(roots, next.left);
        if (next.right != null) node.right = buildTree(roots, next.right);
    }
    return node;
}
private boolean isValid(TreeNode node, int min, int max) { // 这些个递归写得很传功力，要活学活用到出神入化。。。。。
    if (node == null) return true;
    final int value = node.val;
    if (value <= min || value >= max) return false;
    return isValid(node.left, min, value) && isValid(node.right, value, max);
}
```

1.1.9 687. Longest Univalue Path

Given the root of a binary tree, return the length of the longest path, where each node in the path has the same value. This path may or may not pass through the root.

The length of the path between two nodes is represented by the number of edges between them.

- 此题与求二叉树的最长路径边长相似，只是此题要求是节点值相同的路径，也就是说在找最长路径的时候，还需要判断节点值，要是不相同，就重置为 0，在此期间，我们使用一个全局变量来存储最长节点值相同路径的边长。

```
private int topDownTraverse(TreeNode r) {
    if (r == null) return 0;
    int left = topDownTraverse(r.left);
    int right = topDownTraverse(r.right);
    if (r.left == null || r.left.val != r.val) left = 0;
    if (r.right == null || r.right.val != r.val) right = 0;
    max = Math.max(max, left + right);
    return Math.max(left, right) + 1;
}
int max = 0;
public int longestUnivaluePath(TreeNode root) {
    if (root == null) return 0;
    topDownTraverse(root);
    return max;
}
```

1.1.10 652. Find Duplicate Subtrees

Given the root of a binary tree, return all duplicate subtrees.

For each kind of duplicate subtrees, you only need to return the root node of any one of them.

Two trees are duplicate if they have the same structure with the same node values.

```
private String duplicate(TreeNode node) {
    if(node == null) return "X";
    String l = duplicate(node.left);
    String r = duplicate(node.right);
    String s = Integer.toString(node.val) + "-" + l + "-" + r;
    map.put(s, map.getOrDefault(s, 0)+1);
    if (map.get(s) == 2)
        list.add(node);
    return s;
}
HashMap<String,Integer> map = new HashMap<>();
ArrayList list = new ArrayList<>();
public List findDuplicateSubtrees(TreeNode root) {
    duplicate(root);
    return list;
}
```

- 看一下构造的图的效果图

```
      1 -> root
    2, 3,  ->
4, #| 2, 4,  ->
#.#| 4, #| #.#|  ->
#.#|  ->
```

```
map.size(): 4
3-2-4-X-X-X-4-X-X, 1
1-2-4-X-X-X-3-2-4-X-X-X-4-X-X, 1
2-4-X-X-X, 2
4-X-X, 3

res.size(): 2
TREE Level order traversal:
      4 -> root
    #.#|  ->

TREE Level order traversal:
      2 -> root
    4, #|  ->
#.#|  ->
```

- 一种 dfs 的写法

```
HashSet<String> set, added;
List<TreeNode> list;
public List<TreeNode> findDuplicateSubtrees(TreeNode root) {
    set = new HashSet();
    added = new HashSet();
    list = new ArrayList();
    StringBuilder ret = dfs(root);
    return list;
}
private StringBuilder dfs(TreeNode root){
    if (root == null) return null;
    StringBuilder sbL = dfs(root.left), sbR = dfs(root.right);
    if (sbL == null && sbR == null){
        sbL = new StringBuilder();
        sbL.append(root.val);
    } else if (sbL != null){
        sbL.append(" " + root.val);
        if (sbR != null){
            sbL.append(' ');
            sbL.append(sbR);
        } else sbL.append(" n");
    } else if (sbL == null){
        if (sbR != null){
            sbR.insert(0, " n " + root.val);
            sbL = sbR;
        }
    }
    String temp = sbL.toString();
    if (set.contains(temp) && !added.contains(temp)){
        list.add(root);
        added.add(temp);

    }
    set.add(temp);
    return sbL;
}
```

- 这个跑起来很高效，可惜我看不懂。。。。。以后再慢慢消化吧
- https://leetcode.com/problems/find-duplicate-subtrees/discuss/1418487/Java-beats-99.5-in-time

```
Map<Integer, Integer> count;           // frequency of each subtree represented in string
Map<List<Integer>, Integer> numberMap; // ** not hashset since it cannot reserve element order
List<TreeNode> ans;
int globalNumber = 1;
public List<TreeNode> findDuplicateSubtrees(TreeNode root) {
    count = new HashMap();
    numberMap = new HashMap();
    ans = new ArrayList();
    collect(root);
    return ans;
}
public int collect(TreeNode node) {
    if (node == null) return 0;
    int leftNumber = collect(node.left);
    int rightNumber = collect(node.right);
    List<Integer> numberExp = new ArrayList<>(); // construct expression
    numberExp.add(node.val);
    numberExp.add(leftNumber);
    numberExp.add(rightNumber);
    if (!numberMap.containsKey(numberExp)) { // update numberMap
        numberMap.put(numberExp, globalNumber);
        globalNumber++;
    }
    // check number frequency. if == 2, meaning duplication then add to result
    int rootNumber = numberMap.get(numberExp).intValue();
    count.put(rootNumber, count.getOrDefault(rootNumber, 0)+1);
    if (count.get(rootNumber) == 2) // not >=2, otherwise ans will have duplicated nodes
```

```
        ans.add(node);
    return rootNumber;
}

count.size(): 4
1, 3
2, 2
3, 1
4, 1
numberMap.size(): 4
2, 1, 0,
2
3, 2, 1,
3
1, 2, 3,
4
4, 0, 0,
1
```

1.1.11 Create Sorted Array through Instructions

Given an integer array instructions, you are asked to create a sorted array from the elements in instructions. You start with an empty container nums. For each element from left to right in instructions, insert it into nums. The cost of each insertion is the minimum of the following: The number of elements currently in nums that are strictly less than instructions[i]. The number of elements currently in nums that are strictly greater than instructions[i]. For example, if inserting element 3 into nums = [1,2,3,5], the cost of insertion is min(2, 1) (elements 1 and 2 are less than 3, element 5 is greater than 3) and nums will become [1,2,3,3,5]. Return the total cost to insert all elements from instructions into nums. Since the answer may be large, return it modulo 109 + 7

```
// https://blog.csdn.net/qq_28033719/article/details/112506925
private static int N = 100001;
private static int [] tree = new int [N]; // 拿元素值作为 key 对应 tree 的下标值
public int lowbit(int i) {
    return i & -i;
}
public void update(int i, int v) { // 更新父节点
    while (i <= N) {
        tree[i] += v;
        i += lowbit(i);
    }
}
public int getSum(int i) { // 得到以 i 为下标 1-based 的所有子、叶子节点的和，也就是 [1, i] 的和，1-based
    int ans = 0;
    while (i > 0) {
        ans += tree[i];
        i -= lowbit(i);
    }
    return ans;
}
public int createSortedArray(int[] instructions) {
    int n = instructions.length;
    long res = 0;
    Arrays.fill(tree, 0);
    for (int i = 0; i < n; i++) {
        //              严格小于此数的个数 严格大于此数的个数: 为总个数（不含自己） - 小于自己的个数
        res += Math.min(getSum(instructions[i]-1), i-getSum(instructions[i]));
        update(instructions[i], 1);
    }
    return (int)(res % ((int)Math.pow(10, 9) + 7));
}
```

1.1.12 1696. Jump Game VI

You are given a 0-indexed integer array nums and an integer k. You are initially standing at index 0. In one move, you can jump at most k steps forward without going outside the boundaries of the array. That is, you can jump from index i to any index in the range [i + 1, min(n - 1, i + k)] inclusive. You want to reach the last index of the array (index n - 1). Your score is the sum of all nums[j] for each index j you visited in the array. Return the maximum score you can get.

```
public int maxResult(int[] nums, int k) { // O(N) DP with double ended queue
    int n = nums.length;
    int [] dp = new int[n];
    ArrayDeque<Integer> q = new ArrayDeque<>();
    for (int i = 0; i < n; i++) {
        while (!q.isEmpty() && q.peekFirst() < i-k) // 头大尾小
            q.removeFirst();
        dp[i] = nums[i] + (q.isEmpty() ? 0 : dp[q.peekFirst()]);
        while (q.size() > 0 && dp[q.peekLast()] <= dp[i])
            q.removeLast();
        q.addLast(i);
    }
    return dp[n-1];
```

```
}
public int maxResult(int[] nums, int k) { // BigO: O (NlogN)
    int n = nums.length;
    int [] dp = new int[n];
    Queue<int []> q = new PriorityQueue<>(Comparator.comparingInt(e -> -e[0]));
    for (int i = 0; i < n; i++) {
        while (!q.isEmpty() && q.peek()[1] + k < i)
            q.poll();
        dp[i] = nums[i] + (q.isEmpty() ? 0 : q.peek()[0]);
        q.add(new int[] {dp[i], i});
    }
    return dp[n-1];
}
```

1.1.13 1345. Jump Game IV - Hard

Given an array of integers arr, you are initially positioned at the first index of the array.
In one step you can jump from index i to index:

```
i + 1 where: i + 1 < arr.length.
i - 1 where: i - 1 >= 0.
```

j where: arr[i] = arr[j] and i != j.
Return the minimum number of steps to reach the last index of the array.
Notice that you can not jump outside of the array at any time.

1. 解题思路与分析

    - 首先题目给出了起点和终点，分别是数组的头部和尾部，另外，每次跳跃我们可以跳向相邻的左右 2 点以及与当前数值相同的所有点。描述到这里，题目的图形结构已经非常清晰，这实际上是一道，在已知起点和终点的情况下，求图中最短路径的问题。如果你经常看我的博客，你会马上想到，求最短路径的首选应该是 bfs，某些情况下 dfs 也是可行的。
    - 接下来看解题步骤，既然是图型题，我们需要先将图构建出来，比较重要的部分应该是数组中值相同的部分，我们定义一个 Map，key 是数值，value 是具有该数值的数组下标集合。另外这里有一处可以优化的地方，比如数组中有一连串的相同数字：

    ```
    arr = [11,22,7,7,7,7,7,7,7,22,13]
    ```

    - 对于数组中连续的数字 7，实际上起作用的只有首尾两个，其他 7 无论如何跳都不会优于两边的两个 7 的。因此，当遇上连续相同数字时，我们只在 map 中保存首尾 2 个即可。图形结构构建好之后，就是标准的 bfs 解题逻辑
    - 这就是个 BFS 的题,唯一注意的是:如果 left, current, right 都是同一个数,那么 HashMap<Integer, List<Integer»又要重新访问一遍，那么解决办法就是访问过当前 node 的所有 index 之后，立刻清零；这样每个 index 只访问一遍；O(N)
    - 自已写的臭长的代码

    ```
    public int minJumps(int [] a) {
        int n = a.length;
        if (n == 1) return 0;
        boolean [] vis = new boolean [n];
        Map<Integer, List<Integer>> m = new HashMap<>();
        for (int i = 0; i < n; i++) {
            if (i-1 >= 0 && a[i-1] == a[i] && i+1 < n && a[i+1] == a[i]) { // 任何一端的相等元素都可以 cover 当前元素，直接跳过
                vis[i] = true;
                continue;
            }
            m.computeIfAbsent(a[i], z -> new ArrayList<>()).add(i);
        }
        Deque<Integer> q = new ArrayDeque<>();
        Set<Integer> sc = new HashSet<>(); // set of current
        Set<Integer> sn = new HashSet<>(); // set of next
        sc.add(0);
        int cnt = 0;
        while (sc.size() > 0) {
            for (int v : sc) q.offerLast(v);
            while (!q.isEmpty()) {
                int cur = q.pollFirst();
                if (cur == n-1) return cnt;
                vis[cur] = true;
                if (cur < n-1 && !vis[cur+1]) sn.add(cur+1);
                if (cur > 0 && !vis[cur-1]) sn.add(cur-1);
                for (int idx : m.get(a[cur])) {
                    if (vis[idx] || idx == cur) continue;
                    if (idx == n-1) return cnt + 1;
                    sn.add(idx);
                }
                m.put(a[cur], new ArrayList<>()); // 每个相同数值只处理一次进队列操作
            }
    ```

```
                sc.clear();
                sc.addAll(sn);
                sn.clear();
                cnt++;
            }
            return -1;
        }
```

- 再看一下别人逻辑清晰的代码

```
    public int minJumps(int [] a) { // 思路简洁：比上面的方法快了很多
        int n = a.length;
        Map<Integer, List<Integer>> m = new HashMap<>();
        for (int i = 0; i < n; i++)
            m.computeIfAbsent(a[i], z -> new ArrayList<>()).add(i);
        int cnt = 0;
        boolean [] vis = new boolean [n];
        Deque<Integer> q = new ArrayDeque<>();
        q.offerLast(0);
        vis[0] = true;
        while (!q.isEmpty()) {
            for (int z = q.size()-1; z >= 0; z--) {
                int cur = q.pollFirst();
                if (cur == n-1) return cnt;
                for (int idx : m.get(a[cur]))
                    if (idx != cur && !vis[idx]) {
                        q.offerLast(idx);
                        vis[idx] = true;
                    }
                if (cur-1 >= 0 && !vis[cur-1]) {
                    q.offerLast(cur-1);
                    vis[cur-1] = true;
                }
                if (cur+1 < n && !vis[cur+1]) {
                    q.offerLast(cur+1);
                    vis[cur+1] = true;
                }
                m.put(a[cur], new ArrayList<>()); // 清零操作：每个相同数值只做入队列操作一次
            }
            cnt++;
        }
        return -1;
    }
```

1.1.14 968. Binary Tree Cameras

You are given the root of a binary tree. We install cameras on the tree nodes where each camera at a node can monitor its parent, itself, and its immediate children. Return the minimum number of cameras needed to monitor all nodes of the tree.

```
// 对于每个节点，有一下三种 case:
// case (1): 如果它有一个孩子，且这个孩子是叶子 (状态 0)，则它需要摄像头，res ++，然后返回 1，表示已经给它装上了摄像头。
// case (2): 如果它有一个孩子，且这个孩子是叶子的父节点 (状态 1)，那么它已经被覆盖，返回 2。
// case (0): 否则，这个节点无孩子，或者说，孩子都是状态 2，那么我们将这个节点视为叶子来处理。
// 由于 dfs 最终返回后，整棵树的根节点的状态还未处理，因此需要判断，若根节点被视为叶子，需要在其上加一个摄像头。
private int dfs(TreeNode r) {
    // 空节点不需要被覆盖，归入情况 2
    if (r == null) return 2; // do not need cover
    int left = dfs(r.left);  // 递归求左右孩子的状态
    int right = dfs(r.right);
    // 获取左右孩子状态之后的处理
    // 有叶子孩子，加摄像头，归入情况 1
    if (left == 0 || right == 0) {
        res ++;
        return 1;
    }
    // 孩子上有摄像头，说明此节点已被覆盖，情况 2;
    if (left == 1 || right == 1) return 2;
    return 0;
}
int res = 0;
public int minCameraCover(TreeNode root) {
    // 若根节点被视为叶子，需要在其上加一个摄像头
    return (dfs(root) == 0 ? 1 : 0) + res;
}
```

1.1.15 112. Path Sum

Given the root of a binary tree and an integer targetSum, return true if the tree has a root-to-leaf path such that adding up all the values along the path equals targetSum.

A leaf is a node with no children.

```java
int cnt = 0; // 用一个全局变量
public boolean hasPathSum(TreeNode r, int t) {
    if (r == null) return false;
    cnt += r.val;
    if (r.left == null && r.right == null) // {
        if (cnt == t) return true;
        // return false; // 这一句可以不要，是因为还可以再往下遍历一层，会自动返回 false
    // }
    boolean left = hasPathSum(r.left, t);
    boolean right = hasPathSum(r.right, t);
    cnt -= r.val;
    return left || right;
}
// 下面是太习惯用两个方法来写，以至于用一个方法写，不太适应
public boolean hasPathSum(TreeNode r, int t) {
    traversal(r, 0, t);
    return vis;
}
boolean vis = false;
void traversal(TreeNode r, int v, int t) {
    if (r == null) return ;
    if (r.left == null && r.right == null) {
        if (v + r.val == t) vis = true;
        return ;
    }
    traversal(r.left, v + r.val, t);
    traversal(r.right, v + r.val, t);
}
```

- **【亲爱的表哥的活宝妹，任何时候，亲爱的表哥的活宝妹，就是一定要、一定会嫁给活宝妹的亲爱的表哥!!! 爱表哥，爱生活!!!】**

- **【TODO】：参照、按照、【刷题机器】的总结，把【高难度的、各种树】与【图】总结懂、总结透彻!!**

1.2 【树基础】：二叉树 Morris 遍历：

- **【亲爱的表哥的活宝妹，任何时候，亲爱的表哥的活宝妹，就是一定要、一定会嫁给活宝妹的亲爱的表哥!!! 爱表哥，爱生活!!!】**

- 亲爱的表哥的活宝妹，现在【树题型的基础】还狠不错。不想再写什么 r.left 和 r.right 了。统一简化为 r.l 和 r.r 等.l|.r.

- 二叉树遍历的核心问题是，当遍历当前节点的子节点后，如何返回当前节点并继续遍历。遍历二叉树的递归方法和非递归方法都使用了栈结构，记录返回路径，来实现从下层到上层的移动。其空间复杂度最好时为 O(log n)，最坏时为 O(n)（二叉树呈线性）。

- Morris 遍历的实质是避免使用栈，利用底层节点空闲的 right 指针指回上层的某个节点，从而完成下层到上层的移动。

- Morris 遍历的过程

  - 假设来到当前节点 cur，开始时来到根节点位置。
  - 如果 cur 为空时遍历停止，否则进行以下过程。
  - 如果 cur 没有左子树，cur 向右移动（cur = cur->right）。
  - 如果 cur 有左子树，找到左子树上最右的节点，记为 mostRight。
  - 如果 mostRight 的 right 指针指向空，让其指向 cur，然后 cur 向左移动（cur = cur->left）。
  - 如果 mostRight 的 right 指针指向 cur，将其修改为 null，然后 cur 向右移动（cur = cur->right）。

- 这里先举个例子：

```
[1, 2, 3, 4, 5, 6, 7]

TREE Level order traversal:

       1 -> root
     2, 3,  ->
4, 5, 6, 7,  ->
#.#| #.#| #.#| #.#|  ->

cur.val: 1
rightMost.val: 5
cur.val: 2
rightMost.val: 4
cur.val: 4
cur.val: 2
cur.val: 5
cur.val: 1
cur.val: 3
rightMost.val: 6
cur.val: 6
cur.val: 3
cur.val: 7
```

```java
void morris(TreeNode root) {
    TreeNode cur = root;
    while (cur != null) {
        System.out.println("cur.val: " + cur.val);
// 下面这个分支：曾经被搜索到、作为过 rightMost 【叶子节点、其右节点不再为空】的，会被链接到、其先前 cur 的节点!!!
        if (cur.left == null) {
            cur = cur.right;
            continue;
        }
        // 找到当前节点的左子树的最右节点
        TreeNode rightMost = cur.left;
        // 寻找 rightMost.l 子树里，最右的节点，将其 right 指针，指向【当前节点】
//  【写错了】：遍历到：【左子树的：最右、非空、右节点】如此，才能保证：遍历完【左子树】去遍历【右子树】
        // while (rightMost.right != null)
        while (rightMost.right != null && rightMost.right != cur)  // 遍历到：【左子树的：最右、非空、右节点】如此，才能保证：遍历完【左子树】去遍历【右子树】
            rightMost = rightMost.right;
        // 有【前后】 2 次：遍历当前节点 rightMost
        if (rightMost.right == null) { // 第一次遍历到：赋值，将其 right 指针，指向【当前节点 cur】
            // 遍历中搭建：【临时指针】，指向，当前父节点
            System.out.println("rightMost.val: " + rightMost.val);
            rightMost.right = cur;
            cur = cur.left; // 当前节点：向左移动
        } else { //  【TODO】：什么情况下，会走到这个分支的 ???
            // 如果最右节点的 right 指针指向当前节点，说明左子树已经遍历完毕，进入右子树
            rightMost.right = null;
            cur = cur.right; // 当前节点：向右移动.
        }
    }
```

- **【亲爱的表哥的活宝妹，任何时候，亲爱的表哥的活宝妹，就是一定要、一定会嫁给活宝妹的亲爱的表哥!!! 爱表哥，爱生活!!!】**

- **【亲爱的表哥的活宝妹，任何时候，亲爱的表哥的活宝妹，就是一定要、一定会嫁给活宝妹的亲爱的表哥！！！爱表哥，爱生活！！！】**

- **【亲爱的表哥的活宝妹，任何时候，亲爱的表哥的活宝妹，就是一定要、一定会嫁给活宝妹的亲爱的表哥！！！爱表哥，爱生活！！！】**

- **【亲爱的表哥的活宝妹，任何时候，亲爱的表哥的活宝妹，就是一定要、一定会嫁给活宝妹的亲爱的表哥！！！爱表哥，爱生活！！！】**

- **【亲爱的表哥的活宝妹，任何时候，亲爱的表哥的活宝妹，就是一定要、一定会嫁给活宝妹的亲爱的表哥！！！爱表哥，爱生活！！！】**

- **【亲爱的表哥的活宝妹，任何时候，亲爱的表哥的活宝妹，就是一定要、一定会嫁给活宝妹的亲爱的表哥！！！爱表哥，爱生活！！！】**

- **【亲爱的表哥的活宝妹，任何时候，亲爱的表哥的活宝妹，就是一定要、一定会嫁给活宝妹的亲爱的表哥！！！爱表哥，爱生活！！！】**

1.3 【树基础】：二叉搜索树 & 平衡树

- 小题目：查找排名为 k 的元素

- 在一棵子树中，根节点的排名取决于其左子树的大小。

  - 若其左子树的大小大于等于 k，则该元素在左子树中；
  - 若其左子树的大小在区间 [k-count,k-1]（count 为当前结点的值的出现次数）中，则该元素为子树的根节点；
  - 若其左子树的大小小于 k-count，则该元素在右子树中。

- 时间复杂度 O(h)。

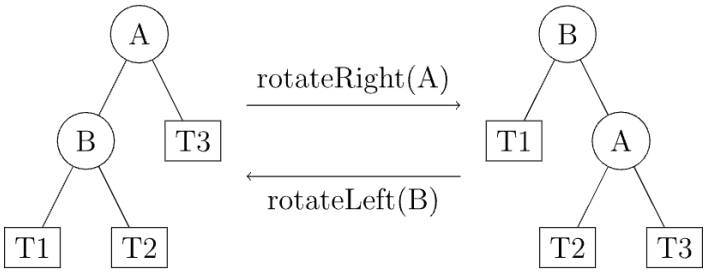
```
int querykth(TreeNode f, int k) {
    if (f == null) return -1;
    if (f.l != null) {
        if (f.l.size >= k)
            return querykth(f.l, k);
        if (f.l.size + f.count >= k)
            return f.val;
    } else {
        if (k == 1)
            return f.val;
    }
    return querykth(f.r, k - (f.l == null ? 0 : f.l.size) - f.count);
}
```

平衡的调整过程

对不满足平衡条件的搜索树进行调整操作，可以使不平衡的搜索树重新具有平衡性。

关于二叉平衡树，平衡的调整操作分为包括 **左旋（Left Rotate 或者 zag）** 和 **右旋（Right Rotate 或者 zig）** 两种。由于二叉平衡树在调整时需要保证中序遍历序列不变。这两种操作均不改变中序遍历序列。

在这里先介绍右旋，右旋也称为「右单旋转」或「LL 平衡旋转」。对于结点 A 的右旋操作是指：将 A 的左孩子 B 向右上旋转，代替 A 成为根节点，将 A 结点向右下旋转成为 B 的右子树的根结点，B 的原来的右子树变为 A 的左子树。



右旋操作只改变了三组结点关联，相当于对三组边进行循环置换一下，因此需要暂存一个结点再进行轮换更新。

对于右旋操作一般的更新顺序是：暂存 B 结点（新的根节点），让 A 的左孩子指向 B 的右子树 $T2$，再让 B 的右孩子指针指向 A，最后让 A 的父结点指向暂存的 B。

完全同理，有对应的左旋操作，也称为「左单旋转」或「RR 平衡旋转」。左旋操作与右旋操作互为镜像。

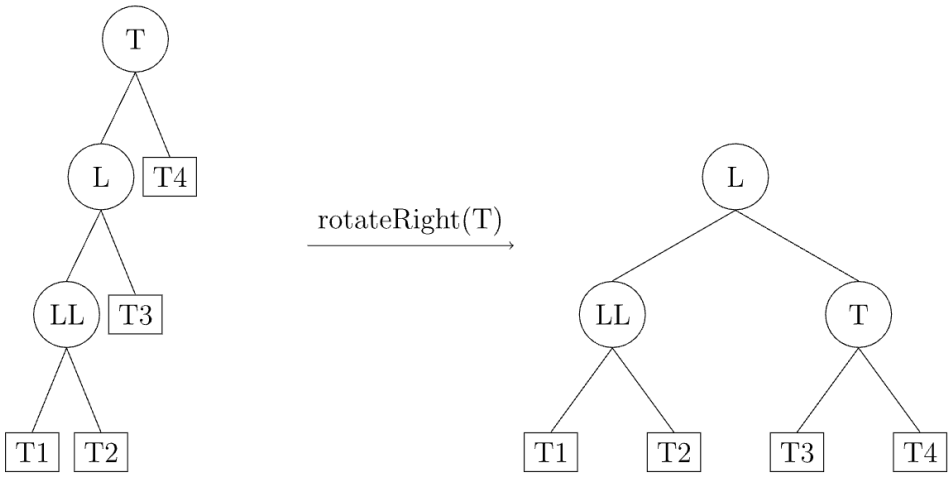
下面给出左旋和右旋的代码。

```
TreeNode rotateLeft(TreeNode root) {
    TreeNode newRoot = root.r; // 新: 根节点
    root.r = newRoot.l; // 原根节点、右子节点:
    newRoot.l = root;
    // 更新相关节点的信息
    updateHeight(root);
    updateHeight(newRoot);
    return newRoot;  // 返回新的根节点
}
TreeNode rotateRight(TreeNode root) {
    TreeNode newRoot = root.l; // 新: 根节点
    root.l = newRoot.r;
    newRoot.r = root;
    // 更新相关节点的信息
    updateHeight(root);
    updateHeight(newRoot);
    return newRoot;  // 返回新的根节点
}
```

1.3.1 四种平衡性破坏的情况：LL RR LR RL

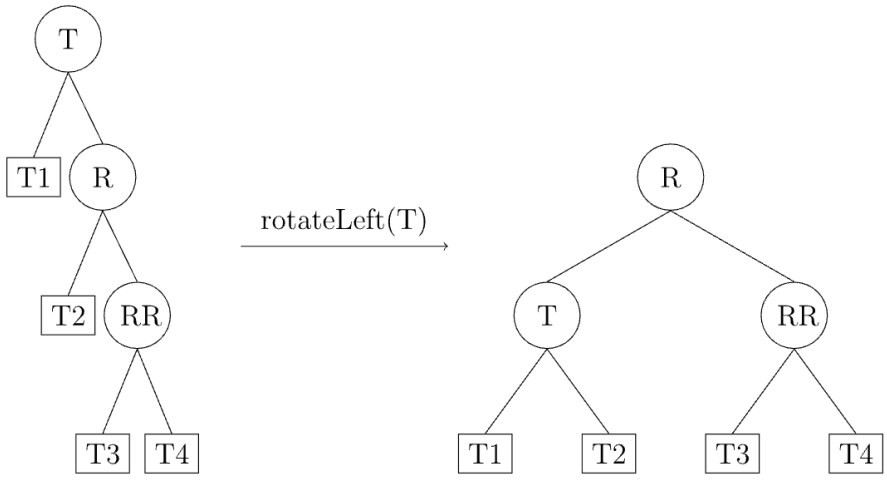
LL 型：T 的左孩子的左子树过长导致平衡性破坏。

调整方式：右旋节点 T。



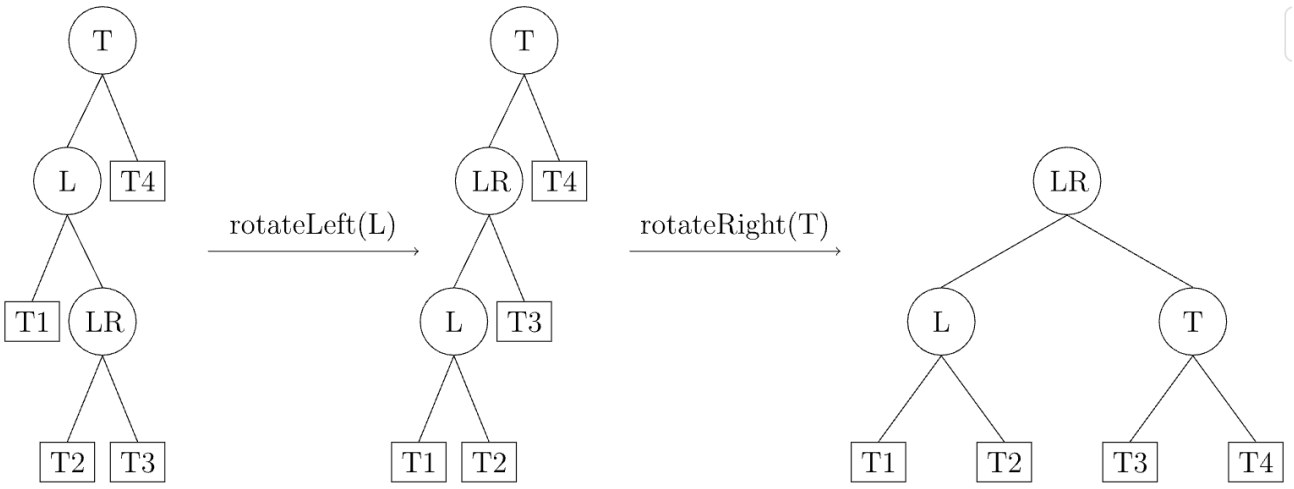
RR 型：与 LL 型类似，T 的右孩子的右子树过长导致平衡性破坏。

调整方式：左旋节点 T。



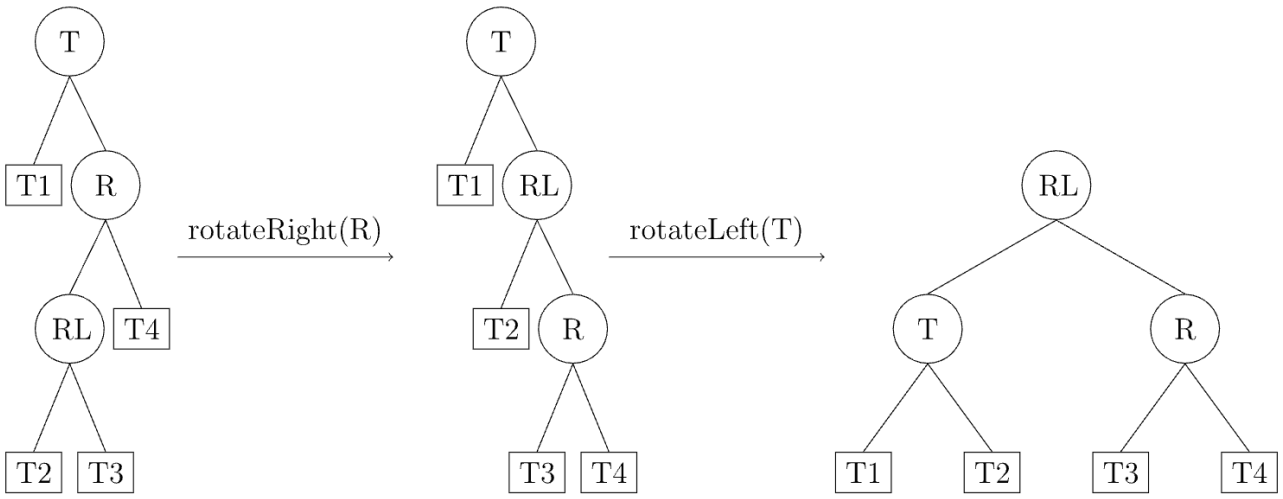
LR 型：T 的左孩子的右子树过长导致平衡性破坏。

调整方式：先左旋节点 L，成为 LL 型，再右旋节点 T。



RL 型：与 LR 型类似，T 的右孩子的左子树过长导致平衡性破坏。

调整方式：先右旋节点 R，成为 RR 型，再左旋节点 T。



1.3.2 Treap 基础：旋转 Treap 的基本操作

- **【亲爱的表哥的活宝妹，任何时候，亲爱的表哥的活宝妹，就是一定要、一定会嫁给活宝妹的亲爱的表哥!!! 爱表哥，爱生活!!!】**

```
Random rand = new Random();
class Node {
    // Node l, r; // 左右子节点
    Node [] n;  // 左右子节点
    int v, rn, r, s; // 值 val; rank; repeat_cnts; size 以当前节点为根的子树大小
    public Node(int v) {
        n[0] = n[1] = null;
        this.v = v;
        r = 1; s = 1; // 1 个重复、1 个节点
        rn = rand.nextInt(1000);
    }
    void updateSize() {
        // 用于旋转和删除过后，重新计算 siz 的值
        s = r;
        if (n[0] != null) s += n[0].s;
        if (n[1] != null) s += n[1].s;
    }
}
// enum rttype { // 反着写：是有原因的!!
//     RT, // 0
```

```
//     LF  // 1
// }
int RT = 0, LF = 1;
void _rotate(Node r, int dir) {
    // dir 参数代表旋转的方向 0 为右旋, 1 为左旋
    // 注意传进来的 cur 是指针的引用, 也就是改了这个
    // cur, 变量是跟着一起改的, 如果这个 cur 是别的 树的子节点, 根据 ch
    // 找过来的时候, 也是会找到这里的

    // 以下的代码解释的均是【左旋】时的情况
    Node t = r.n[dir]; // t: 临时、节点指针, 指向成为【新的根节点】
    /* 【左旋】: 也就是让【右子节点】变成【根节点】
     *          A <-- r           C
     *         / \               / \
     *        B  C <-- t  ==>   A   E
     *          / \            / \
     *         D   E          B   D
     */
    r.n[dir] = t.n[1-dir]; // 让 A 的右子节点变成 D
    t.n[1-dir] = r;        // 让 C 的左子节点变成 A
    r.updateSize(); // 更新大小信息
    t.updateSize(); // 最后把临时储存 C 树的变量赋值到当前根节点上 (注意 r 是引用)
    r = t;
}
//  【插入】: 同【普通二叉搜索树】, 多维护【优先级、最小堆】的步骤
void _insert(Node r, int v) {
    if (r == null) // 没这个节点直接新建
        r = new Node(v);
    else if (r.v == v) {
        // 如果有这个值相同的节点, 就把重复数量加一
        r.r ++; // 重复数＋＋
        r.s ++; // 树大小＋＋
    } else if (v < r.v) {
        // 维护搜索树性质, val 比当前节点小就插到左边, 反之亦然
        _insert(r.n[0], v); // 普通【二叉搜索树】插入
        if (r.n[0].rn < r.rn)
            //  【小根堆】中, 上面节点的优先级一定更小
            // 因为新插的左子节点比父节点小, 现在需要让左子节点变成父节点
            _rotate(r, RT); // 注意前面的旋转性质, 要把左子节点转上来, 需要右旋
        r.updateSize(); // 插入之后大小会变化, 需要更新
    } else { // r.v < v
        _insert(r.n[1], v);
        if (r.n[1].rn < r.rn)
            _rotate(r, LF);
        r.updateSize(); // 插入之后大小会变化, 需要更新
    }
}
//  【删除】:
void _del(Node r, int v) {
    if (r == null) return ; // 根: 空
    if (r.n[0] != null && v <= r.n[0].v) { //  【左子节点】
        _del(r.n[0], v);
        r.updateSize();
    } else if (r.n[1] != null && r.n[1].v <= v) {  //  【右子节点】
        _del(r.n[1], v);
        r.updateSize();
    } else if (r.r > 1) { // 根节点: 有重复, 只降个数
        r.r--;
        r.s--;
    } else { // 根节点: 无重复. 【最复杂】: 分情况讨论
        int state = ((r.n[0] != null ? 1 : 0) | (r.n[1] == null ? 0 : (1 << 1)));
        switch (state) {
        case 0:
            r = null;
            break;
        case 1: // 有左无右
            r = r.n[0];
            break ;
        case 2: // 有右无左: 删除根节点 r, 变【右子节点】为根节点
            r = r.n[1];
            break ;
        case 3: // 有左有右: 选择【优先级小的、作根节点】、删除【原根节点、现在 r.n[d] 子节点】???
            Node t = r;
            // r = r.n[0];
            int d = (r.n[0].rn < t.n[1].rn ? RT : LF); // dir 是 rank 更小的那个儿子
            _rotate(r, d); // 这里的旋转可以把优先级更小的儿子转上去, rt 是 0,
                           // 而 lf 是 1, 刚好跟实际的子树下标【反过来】
            _del(r.n[1-d], v); // 旋转完成后原来的根节点就在旋方向那边, 所以需要
            // 继续把这个原来的根节点删掉
            // 如果说要删的这个节点是在整个树的「上层的」, 那我们会一直通过这【TODO】: 这些破烂, 没读懂。。。
            // 这里的旋转操作, 把它转到没有子树了 (或者只有一个), 再删掉它。
            r.updateSize(); // 删除会造成大小改变
        }
    }
}
// 根据值查询排名
// 操作含义: 查询以 cur 为根节点的子树中, val 这个值的大小的排名 (该子树中小于 val 的节点的个数 + 1)
int _query_rank(Node r, int v) {
    int lesCnts = (r.n[0] == null ? 0 : r.n[0].s);
```

```
    if (r.v == v) // 根节点
        return lesCnts + 1;
    if (v < r.v) { // 左子节点
        if (r.n[0] == null)
            return 1; //  【左子树、空】: 比当前最小节点【根节点】小， rn=1
        return _query_rank(r.n[0], v);
    }
    // 右子节点
    if (r.n[1] == null)
        // 没有右子树的话直接整个树 + 1 相当于 less_siz + cur->rep_cnt + 1
        return r.s + 1;
    // 如果要查的值比这个节点大，那这个节点的【左子树】以及这个【节点自身】肯定都比要查的值小
    // 所以要加上这两个值，再加上往右边找的结果
    //  (以右子树为根的子树中，val 这个值的大小的排名)
    return lesCnts + r.r + _query_rank(r.n[1], v);
}
// 根据排名查询值
// 要根据排名查询值，我们首先要知道如何判断要查的节点在树的哪个部分:
int _query_val(Node r, int rn) {
    int lesCnts = (r.n[0] == null ? 0 : r.n[0].s);
    if (rn <= lesCnts)
        return _query_rank(r.n[0], rn);
    if (rn <= lesCnts + r.r)
        return r.v;
    return _query_rank(r.n[1], rn - lesCnts - r.r);
}
// 查询第一个比 val 小的节点
// 注意这里使用了一个类中的全局变量，q_prev_tmp。
// 这个值是只有在 val 比当前节点值大的时候才会被更改的，所以返回这个变量就是返回 val 最后一次比当前节点的值大，之后就是更小了。
int q_prev_tmp;
int _query_prev(Node r, int v) {
    if (v <= r.v) {
        if (r.n[0] != null) // 左子节点: 非空，才存在解；否则无解
            return _query_prev(r.n[0], v);
    } else { // r.v < v
        q_prev_tmp = r.v;
        if (r.n[1] != null)
            _query_prev(r.n[1], v); // 递归调用时，可能再次、多次更新 q_prev_tmp 全局变量
        return q_prev_tmp;
    }
    return -1; // 无解
}
// 查询第一个比 val 大的节点
// 跟前一个很相似，只是大于小于号换了一下。
int q_nex_tmp;
int _query_nex(Node r, int v) {
    if (r.v <= v) {
        if (r.n[1] != null)
            return _query_nex(r.n[1], v);
    } else { // v < r.v
        q_nex_tmp = r.v;
        if (r.n[0] != null)
            _query_nex(r.n[0], v);
        return q_nex_tmp;
    }
    return -1;
}
// 按值分裂
Node [] split(Node r, int v) {
    if (r == null) return new Node [] {null, null};
;           if (r.v <= v) { // 分裂: 右子节点
        Node [] rr = split(r.n[1], v);
        r.n[1] = rr[0];
        r.updateSize(); // 不要忘记、这点。。。
        return new Node [] {r, rr[1]};
    } else {  // 分裂: 左子节点
        Node [] ll = split(r.n[0], v);
        r.n[0] = ll[1];
        r.updateSize();
        return new Node [] {ll[0], r};
    }
}
 Node [] split_by_rk(Node r, int rn) {
     if (r == null) return new Node [] {null, null, null};
     int lesCnts = (r.n[0] == null ? 0 : r.n[0].s);
     if (rn <= lesCnts) {
         Node [] ll = split_by_rk(r.n[0], rn);
         r.n[0] = ll[2];
         r.updateSize();
         return new Node [] {ll[0], ll[1], r};
     }
     if (rn <= lesCnts + r.r) {
         Node ll = r.n[0], rr = r.n[1];
         r.n[0] = r.n[1] = null;
         // r.updateSize(); // 把这一步、极度简化
         r.s = r.r;
         return new Node [] {ll, r, rr};
     }
     Node [] rr = split_by_rk(r.n[1], rn);
```

```
        r.n[1] = rr[0];
        r.updateSize();
        return new Node [] {r, rr[1], rr[2]};
    }
    //  【亲爱的表哥的活宝妹，任何时候，亲爱的表哥的活宝妹，就是一定要、一定会嫁给活宝妹的亲爱的表哥!!! 爱表哥，爱生活!!!】
    Node merge(Node f, Node g) {
        if (f == null && g == null) return null;
        if (f == null && g != null) return g;
        if (f != null && g == null) return f;
        if (f.rn < g.rn) { // f 根结点
            f.n[1] = merge(f.n[1], g);
            f.updateSize();
            return f;
        } else { // g 根节点
            g.n[0] = merge(f, g.n[0]);
            g.updateSize();
            return g;
        }
    }
```