Tarjan 算法与并查集解决二叉树节点间最近公共祖先的批量查询问题

【题目】

如下的 Node 类是标准的二叉树节点结构：

```
public class Node {
        public int value;
        public Node left;
        public Node right;

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

再定义 Query 类如下：

```
public class Query {
        public Node o1;
        public Node o2;

        public Query(Node o1, Node o2) {
                this.o1 = o1;
                this.o2 = o2;
        }
}
```

一个 Query 类的实例表示一条查询语句，表示想要查询 o1 节点和 o2 节点的最近公共祖先节点。

给定一棵二叉树的头节点 head，并给定所有的查询语句，即一个 Query 类型的数组 Query[] ques，请返回 Node 类型的数组 Node[] ans，ans[i]代表 ques[i]这条查询的答案，即 ques[i].o1 和 ques[i].o2 的最近公共祖先。

【要求】

如果二叉树的节点数为 N，查询语句的条数为 M，整个处理过程的时间复杂度要求达到 $O(N+M)$。

【难度】

校 ★★★☆

【解答】

本题的解法利用了 Tarjan 算法与并查集结构的结合。二叉树如图 3-42 所示，假设想要进行的查询为 ques[0]=(节点 4 和节点 7)，ques[1]=(节点 7 和节点 8)，ques[2]=(节点 8 和节点 9)，ques[3]=(节点 9 和节点 3)，ques[4]=(节点 6 和节点 6)，ques[5]=(null 和节点 5)，ques[6]=(null 和 null)。

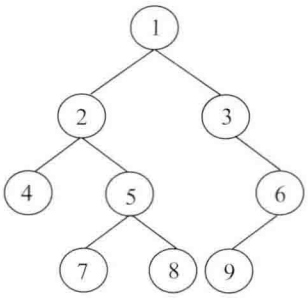

图 3-42

首先生成和 ques 长度一样的 ans 数组，如下三种情况的查询是可以直接得到答案的：

1．如果 o1 等于 o2，答案为 o1。例如，ques[4]，令 ans[4]=节点 6。

2．如果 o1 和 o2 只有一个为 null，答案是不为空的那个。例如，ques[5]，令 ans[5]=节点 5。

3．如果 o1 和 o2 都为 null，答案为 null。例如 ques[6]，令 ans[6]=null。

对不能直接得到答案的查询，我们把查询的格式转换一下，具体过程如下：

1．生成两张哈希表 queryMap 和 indexMap。queryMap 类似于邻接表，key 表示查询涉及的某个节点，value 是一个链表类型，表示 key 与那些节点之间有查询任务。indexMap 的 key 也表示查询涉及的某个节点，value 也是链表类型，表示如果依次解决有关 key 节点的每个问题，该把答案放在 ans 的什么位置。也就是说，如果一个节点为 node，node 与哪些节点之间有查询任务呢？都放在 queryMap 中；获得的答案该放在 ans 的什么位置呢？都放在 indexMap 中。

比如，根据 ques[0～3]，queryMap 和 indexMap 生成记录如下：

q[0]=(4,7)，q[1]=(7,8)，q[2]=(8,9)，q[3]=(9,3)

Key	Value
节点 4	queryMap 中节点 4 的链表：{节点 7} indexMap 中节点 4 的链表：{　0　}
节点 7	queryMap 中节点 7 的链表：{节点 4，节点 8} indexMap 中节点 7 的链表：{　0　，　1　}
节点 8	queryMap 中节点 8 的链表：{节点 7，节点 9} indexMap 中节点 8 的链表：{　1　，　2　}
节点 9	queryMap 中节点 9 的链表：{节点 8，节点 3} indexMap 中节点 9 的链表：{　2　，　3　}
节点 3	queryMap 中节点 3 的链表：{节点 9} indexMap 中节点 3 的链表：{　3　}

读者应该会发现一条(o1,o2)的查询语句在上面的两个表中其实生成了两次。这么做的目的是为了处理时方便找到关于每个节点的查询任务，也方便设置答案，介绍完整个流程之后，会有进一步说明。

接下来是 Tarjan 算法处理 M 条查询的过程，整个过程是二叉树的先左、再根、再右、最后再回到根的遍历。以图 3-42 的二叉树来说明。

1）对每个节点生成各自的集合，{1}，{2}，…，{9}，开始时每个集合的祖先节点设为空。

2）遍历节点 4，发现它属于集合{4}，设置集合{4}的祖先为节点 4，发现有关于节点 4 和节点 7 的查询任务，发现节点 7 属于集合{7}，但集合{7}的祖先节点为空，说明还没遍历到，所以暂时不执行这个查询任务。

2．遍历节点 2，发现它属于集合{2}，设置集合{2}的祖先为节点 2，此时左孩子节点 4 属于集合{4}，将集合{4}与集合{2}合并，两个集合一旦合并，小的不再存在，而是生成更大的集合{4,2}，并设置集合{4,2}的祖先为当前节点 2。

3．遍历节点 7，发现它属于集合{7}，设置集合{7}的祖先为节点 7，发现有关节点 7 和节点 4 的查询任务，发现节点 4 属于集合{4,2}，集合{4,2}的祖先节点为节点 2，说明节点 4 和节点 7 都已经遍历到，根据 indexMap 知道答案应放在 0 位置，所以设置 ans[0]=节点 2；又发现有节点 7 和节点 8 的查询任务，发现节点 8 属于集合{8}，但集合{8}的祖先节点为空，说明还没遍历到，忽略。

4．遍历节点 5，发现它属于集合{5}，设置集合{5}的祖先为节点 5，此时左孩子节点 7 属于集合{7}，两集合合并为{7,5}，并设置集合{7,5}的祖先为当前节点 5。

5．遍历节点 8，发现它属于集合{8}，设置集合{8}的祖先为节点 8，发现有节点 8 和节点 7 的查询任务，发现节点 7 属于集合{7,5}，集合{7,5}的祖先节点为节点 5，设置 ans[1]=节点 5；发现有节点 8 和节点 9 的查询任务，忽略。

6．从节点 5 的右子树重新回到节点 5，节点 5 属于{7,5}，节点 5 的右孩子节点 8 属于{8}，两个集合合并为{7,5,8}，并设置{7,5,8}的祖先节点为当前的节点 5。

7．从节点 2 的右子树重新回到节点 2，节点 2 属于集合{2,4}，节点 2 的右孩子节点 5 属于集合{7,5,8}，合并为{2,4,7,5,8}，并设置这个集合的祖先节点为当前的节点 2。

8．遍历节点 1，{2,4,7,5,8}与{1}合并为{2,4,7,5,8,1}，这个集合祖先节点为当前的节点 1；

9．遍历节点 3，发现属于集合{3}，集合{3}祖先节点设为节点 3，发现有节点 3 和节点 9 的查询任务，但节点 9 没遍历到，忽略。

10．遍历节点 6，发现属于集合{6}，集合{6}祖先节点设为节点 6。 左根右，再回到根

11．遍历节点 9，发现属于集合{9}，集合{9}祖先节点设为节点 9；发现有节点 9 和节点 8 的查询任务，节点 8 属于{2,4,7,5,8,1}，这个集合的祖先节点为节点 1，根据 indexMap 知道答案应放在 2 位置，所以设置 ans[2]=节点 1；发现有节点 9 和节点 3 的查询任务，节点 3 属于{3}，这个集合的祖先节点为节点 3，根据 indexMap，答案应放在 3 位置，所以设置 ans[3]=节点3。

12．回到节点 6，合并{6}和{9}为{6,9}，{6,9}的祖先节点设为节点 6。

13．回到节点 3，合并{3}和{6,9}为{3,6,9}，{3,6,9}的祖先节点设为节点 3。

14．回到节点 1，合并{2,4,7,5,8,1}和{3,6,9}为{1,2,3,4,5,6,7,8,9}，祖先节点设为节点 1。

15．过程结束，所有的答案都已得到。

现在我们可以解释生成 queryMap 和 indexMap 的意义了，遍历到一个节点时记为 a，queryMap 可以让我们迅速查到有哪些节点和 a 之间有查询任务，如果能够得到答案，indexMap 还能告诉我们把答案放在 ans 的什么位置。假设 a 和节点 b 之间有查询任务，如果此时 b 已经遍历过，自然可以取得答案，然后在有关 a 的链表中，删除这个查询任务；如果此时 b 没有遍历过，依然在属于 a 的链表中删除这个查询任务，这个任务会在遍历到 b 的时候重新被发现，因为同样的任务 b 也存了一份。所以遍历到一个节点，有关这个节点的任务列表会被完全清空，可能有些任务已被解决，有些则没有也不要紧，一定会在后序的过程中被发现并得以解决。这就是 queryMap 和 indexMap 生成两遍查询任务信息的意义。

上述流程很好理解，但大量出现生成集合、合并集合和根据节点找到所在集合的操作，如果二叉树的节点数为 N，那么生成集合操作 $O(N)$次，合并集合操作 $O(N)$次，根据节点找

到所在集合 $O(N+M)$次。所以，如果上述整个过程想达到 $O(N+M)$的时间复杂度，那就要求有关集合的单次操作，平均时间复杂度要求为 $O(1)$，请注意这里说的是平均。存在这么好的集合结构吗？存在。这种集合结构就是接下来要介绍的并查集结构。

并查集结构由 Bernard A. Galler 和 Michael J. Fischer 在 1964 年发明，但证明时间复杂度的工作却持续了数年之久，直到 1989 才彻底证明完毕。有兴趣的读者请阅读《算法导论》一书来了解整个证明过程，本书由于篇幅所限，不再详述证明过程，这里只重点介绍并查集的结构和各种操作的细节，并实现针对二叉树结构的并查集，这是一种经常使用的高级数据结构。

请读者注意，上述流程中提到一个集合祖先节点的概念与接下来介绍并查集时提到的一个集合代表节点（父节点）的概念不是一回事。本题的流程中有关设置一个集合祖先节点的操作也不属于并查集自身的操作，关于这个操作，我们在介绍完并查集结构之后再详细说明。

并查集由一群集合构成，比如步骤 1 中对每个节点都生成各自的集合，所有集合的全体构成一个并查集={ {1}, {2}, ..., {9} }。这些集合可以合并，如果最终合并成一个大集合(步骤 14)，那么此时并查集中有一个元素，这个元素是这个大集合，即并查集={ {1,2，…，9} }。其实主要是想说明并查集是集合的集合这个概念。

并查集先经历初始化的过程，就向流程中的步骤 1 一样，把每个节点都生成一个只含有自己的集合。那么并查集中的单个集合是什么结构呢？如果集合中只有一个元素，记为节点 a 时，如图 3-43 所示。

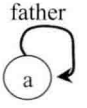

图 3-43

当集合中只有一个元素时，这个元素的 father 为自己，也就意味着这个集合的代表节点就是唯一的元素。实现记录节点 father 信息的方式有很多，本书使用哈希表来保存所有并查集中所有集合的所有元素的 father 信息，记为 fatherMap。比如，对于这个集合，在 fatherMap 中肯定有某一条记录为(节点 a(key) 节点 a(value)) 表示 key 节点的 father 为 value 节点。每个元素除了 father 信息，还有另一个信息叫 rank，rank 为整数代表一个节点的秩，秩的概念可以粗略地理解为一个节点下面还有多少层节点，但是并查集结构对每个节点秩的更新并不严格，所以每个节点的秩只能粗略描述该节点下面的深度，正是由于秩在更新

上的不严格，换来了极好的时间复杂度，而也正是因为这种不严格增加了并查集时间复杂度证明的难度。集合中只有一个元素时，这个元素的 rank 初始化为 0。所有节点的秩信息保存在 rankMap 中。

对二叉树结构并查集初始化的具体过程请参看如下 DisjointSets 类中的 makeSets 方法。

当集合有多个节点时，下层节点的 father 为上层节点，最上层的节点 father 指向自己，最上层的节点又叫集合的代表节点，如图 3-44 所示。

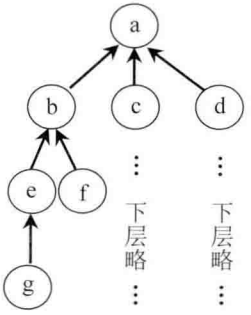

图 3-44

在并查集中，若要查一个节点属于哪个集合，就是在查这个节点所在集合的代表节点是什么，一个节点通过 father 信息逐渐找到最上面的节点，这个节点的 father 是自己，代表整个集合。比如图 3-44 中，任何一个节点最终都找到节点 a，比如节点 g。如果另外一个节点假设为 z，找到的代表节点不是节点 a，那么可以肯定节点 g 和节点 z 不在一个集合中。通过一个节点找到所在集合代表节点的过程叫作 findFather 过程。findFather 最终会返回代表节点，但过程并不仅是单纯的查找过程，还会把整个查找路径压缩。比如，执行 findFather(g)，通过 father 逐渐向上，找到最上层节点 a 之后，会把从 a 到 g 这条路径上所有节点的 father 都设置为 a，则集合变成图 3-45 的样子。

经过路径压缩之后，路径上每个节点下次在找代表节点的时候都只需经过一次移动的过程。这也是整个并查集结构的设计中最重要的优化。

根据一个节点查找所在集合代表节点的过程请参看如下 DisjointSets 类中的 findFather 方法。

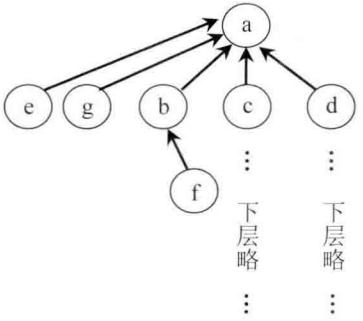

图 3-45

前面已经展示了并查集中的集合如何初始化，如何根据某一个节点查找所在集合的代表元素以及如何做路径压缩的过程，接下来介绍集合如何合并。首先，两个集合进行合并操作时，参数并不是两个集合，而是并查集中任意的两个节点，记为 a 和 b。所以集合的合并更准确的说法是，根据 a 找到 a 所在集合的代表节点是 findFather(a)，记为 aF，根据 b 找到 b 所在集合的代表节点是 findFather(b)，记为 bF，然后用如下策略决定由哪个代表节点作为合并后大集合的代表节点。

1．如果 aF==bF，说明 a 和 b 本身就在一个集合里，不用合并。

2．如果 aF!=bF，那么假设 aF 的 rank 值记为 aFrank，bF 的 rank 值记为 bFrank。根据对 rank 的解释，rank 可以粗描一个节点下面的层数，而 aF 和 bF 本身又是各自集合中最上面的节点，所以 aFrank 粗描 a 所在集合的总层数，bFrank 粗描 b 所在集合的总层数。如果 aFrank<bFrank，那么把 aF 的 father 设为 bF，表示 a 所在集合因为层数较少，所在挂在了 b 所在集合的下面，这样合并之后的大集合 rank 不会有变化。如果 aFrank>bFrank，就把 bF 的 father 设为 aF。如果 aFrank==bFrank，那么 aF 和 bF 谁做大集合的代表都可以，本文的实现是用 aF 作为代表，即把 bF 的 father 设为 aF，此时 aF 的 rank 值增加 1。

合并过程如图 3-46 和图 3-47 所示。

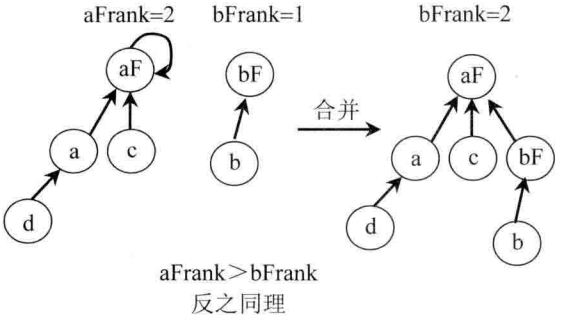

图 3-46

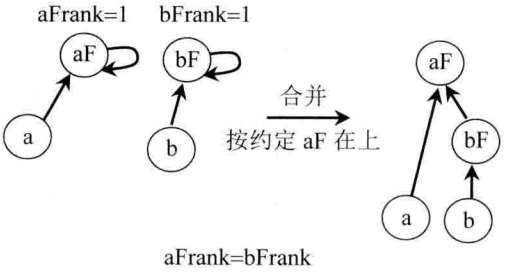

图 3-47

根据两个节点合并两个集合的过程请参看如下 DisjointSets 类中的 union 方法。

```
public class DisjointSets {
        public HashMap<Node, Node> fatherMap;
        public HashMap<Node, Integer> rankMap;

        public DisjointSets() {
                fatherMap = new HashMap<Node, Node>();
                rankMap = new HashMap<Node, Integer>();
        }

        public void makeSets(Node head) {
                fatherMap.clear();
                rankMap.clear();
                preOrderMake(head);
        }

        private void preOrderMake(Node head) {
                if (head == null) {
                        return;
                }
                fatherMap.put(head, head);
```

```
            rankMap.put(head, 0);
            preOrderMake(head.left);
            preOrderMake(head.right);
        }

        public Node findFather(Node n) {
            Node father = fatherMap.get(n);
            if (father != n) {
                father = findFather(father);
            }
            fatherMap.put(n, father);
            return father;
        }

        public void union(Node a, Node b) {
            if (a == null || b == null) {
                return;
            }
            Node aFather = findFather(a);
            Node bFather = findFather(b);
            if (aFather != bFather) {
                int aFrank = rankMap.get(aFather);
                int bFrank = rankMap.get(bFather);
                if (aFrank < bFrank) {
                    fatherMap.put(aFather, bFather);
                } else if (aFrank > bFrank) {
                    fatherMap.put(bFather, aFather);
                } else {
                    fatherMap.put(bFather, aFather);
                    rankMap.put(aFather, aFrank + 1);
                }
            }
        }

    }
```

介绍完并查集的结构之后，最后解释一下在总流程中如何设置一个集合的祖先节点，如上流程中的每一步都有把当前点 node 所在集合的祖先节点设置为 node 的操作。在整个流程开始之前，建立一张哈希表，参看如下 Tarjan 类中的 ancestorMap，我们知道在并查集中，每个集合都是用该集合的代表节点来表示的。所以，如果想把 node 所在集合的祖先节点设为 node，只用把记录(findFather(node) , node)放入 ancestorMap 中即可。同理，如果想得到一个节点 a 所在集合的祖先节点，令 key 为 findFather(a)，然后从 ancestorMap 中取出相应的记录即可。ancestorMap 同时还可以表示一个节点是否被访问过。

全部的处理流程请参看如下代码中的 tarJanQuery 方法。

```
// 主方法
public Node[] tarJanQuery(Node head, Query[] quries) {
```

```
        Node[] ans = new Tarjan().query(head, quries);
        return ans;
    }

    // Tarjan 类实现处理流程
    public class Tarjan {
        private HashMap<Node, LinkedList<Node>> queryMap;
        private HashMap<Node, LinkedList<Integer>> indexMap;
        private HashMap<Node, Node> ancestorMap;
        private DisjointSets sets;

        public Tarjan() {
            queryMap = new HashMap<Node, LinkedList<Node>>();
            indexMap = new HashMap<Node, LinkedList<Integer>>();
            ancestorMap = new HashMap<Node, Node>();
            sets = new DisjointSets();
        }

        public Node[] query(Node head, Query[] ques) {
            Node[] ans = new Node[ques.length];
            setQueries(ques, ans);
            sets.makeSets(head);
            setAnswers(head, ans);
            return ans;
        }

        private void setQueries(Query[] ques, Node[] ans) {
            Node o1 = null;
            Node o2 = null;
            for (int i = 0; i != ans.length; i++) {
                o1 = ques[i].o1;
                o2 = ques[i].o2;
                if (o1 == o2 || o1 == null || o2 == null) {
                    ans[i] = o1 != null ? o1 : o2;
                } else {
                    if (!queryMap.containsKey(o1)) {
                        queryMap.put(o1, new LinkedList<Node>());
                        indexMap.put(o1, new LinkedList<Integer>());
                    }
                    if (!queryMap.containsKey(o2)) {
                        queryMap.put(o2, new LinkedList<Node>());
                        indexMap.put(o2, new LinkedList<Integer>());
                    }
                    queryMap.get(o1).add(o2);
                    indexMap.get(o1).add(i);
                    queryMap.get(o2).add(o1);
                    indexMap.get(o2).add(i);
                }
            }
        }

        private void setAnswers(Node head, Node[] ans) {
```

```
                if (head == null) {
                        return;
                }
                setAnswers(head.left, ans);
                sets.union(head.left, head);
                ancestorMap.put(sets.findFather(head), head);
                setAnswers(head.right, ans);
                sets.union(head.right, head);
                ancestorMap.put(sets.findFather(head), head);
                LinkedList<Node> nList = queryMap.get(head);
                LinkedList<Integer> iList = indexMap.get(head);
                Node node = null;
                Node nodeFather = null;
                int index = 0;
                while (nList != null && !nList.isEmpty()) {
                        node = nList.poll();
                        index = iList.poll();
                        nodeFather = sets.findFather(node);
                        if (ancestorMap.containsKey(nodeFather)) {
                                ans[index] = ancestorMap.get(nodeFather);
                        }
                }
        }

}
```

二叉树节点间的最大距离问题

【题目】

从二叉树的节点 A 出发，可以向上或者向下走，但沿途的节点只能经过一次，当到达节点 B 时，路径上的节点数叫作 A 到 B 的距离。

比如，图 3-48 所示的二叉树，节点 4 和节点 2 的距离为 2，节点 5 和节点 6 的距离为 5。给定一棵二叉树的头节点 head，求整棵树上节点间的最大距离。

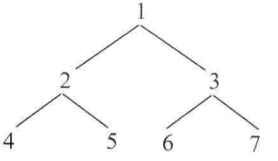

图 3-48