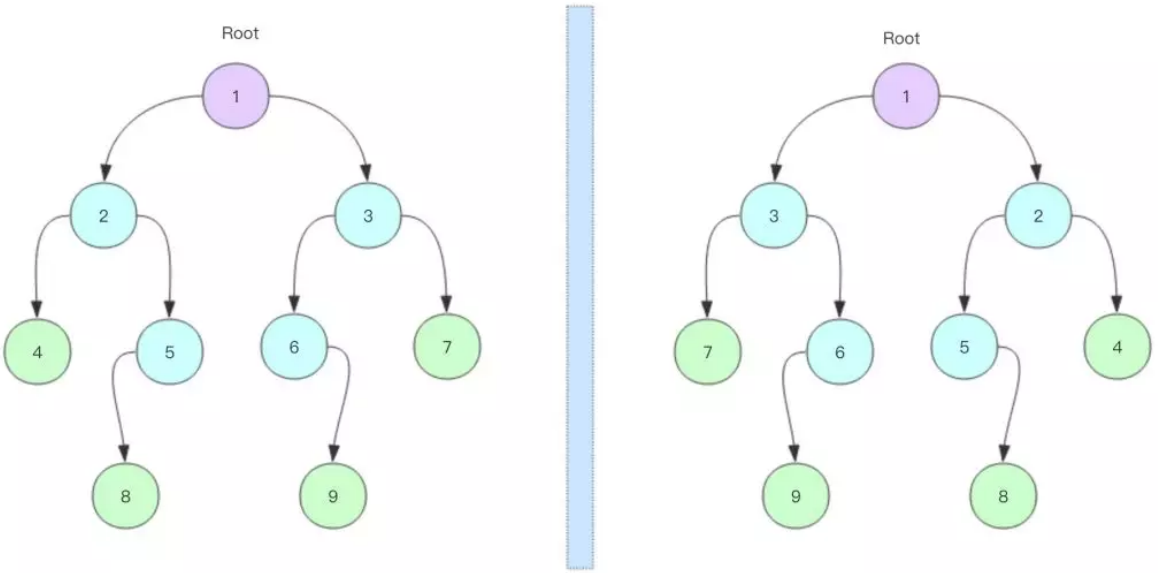
BAT 经典算法笔试题： 镜像二叉树

原创： 佬钱 码洞 今天

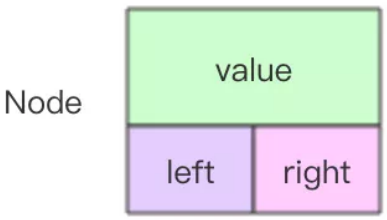
再过不到 2 个月，互联网行业就要再次迎来面试高峰了。为了让大家能顺利通过所有面试环节必经的笔试阶段，我提前给大伙准备了一套常见的算法笔试题。这套算法题来源于 LeetCode，题目都是 BAT、京东头条滴滴美团等大型互联网公司都喜欢考的题目。

算法这个东西很难，纵使你了解了其中的逻辑，用代码写出来仍然不是一件容易的事，内部有太多的细节需要处理。为了便于大伙轻松理解算法逻辑，对于所有的题目，我会使用图文加动画的形式进行讲解，让读者可以轻松理解算法逻辑的同时，还可以留下深刻的影像不容易遗忘。



好，下面我们开始今天的算法题：镜像二叉树，就是将一颗二叉树上的左右节点全部交换，就好比镜子里的二叉树，左右方向是反过来的。

二叉树节点表示

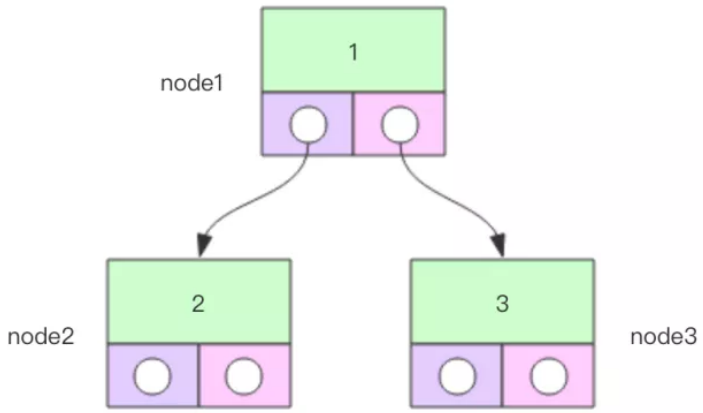



```
class  Node<T>  {
        T  value;
```

```
        Node<T>  left;
        Node<T>  right;

        Node(T  value)  {
                this.value  =  value;
        }

        Node(T  value,  Node<T>  left,  Node<T>  right)  {
                this.value  =  value;
                this.left  =  left;
                this.right  =  right;
        }
}
```

一个参数的构造器是叶子节点，三个参数的构造器是中间节点，看到这里读者应该知道这是 Java 语言，我使用了范型，可以容纳各种值类型。

构造二叉树

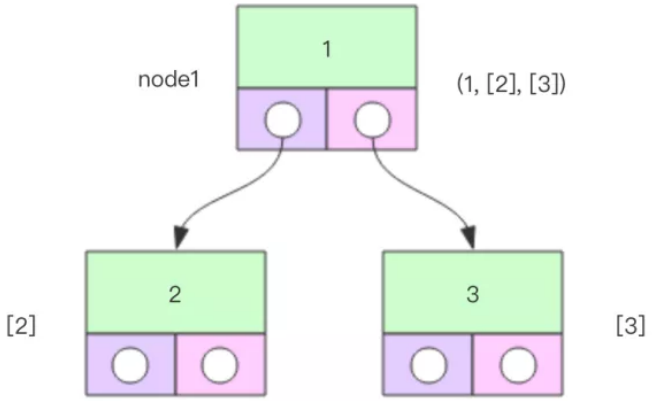



```
Node<Integer>  node2  =  new  Node<>(2);
Node<Integer>  node3  =  new  Node<>(3);
Node<Integer>  node1  =  new  Node<>(1,  node2,  node3);

//  一次性构造
Node<Integer>  node1  =  new  Node<>(1,  new  Node<>(2),  new  Node<>(3));
```

呈现二叉树结构

如果我们正确写出了镜像算法，那如何来直观验证镜像的结构是否正确呢？

LeetCode 使用的是单元测试，它使用一系列的单元测试和压力测试脚本代码来验证用户编写的算法的正确性和性能。但是我们不要这样做，因为不直观。我们选择对二叉树的结构内容进行直观的呈现，如此就可以使用肉眼来进行快速验证。如何直观呈现呢？我们使用最简单的括号表示法，它并不是最直观的，但是它易于实现。

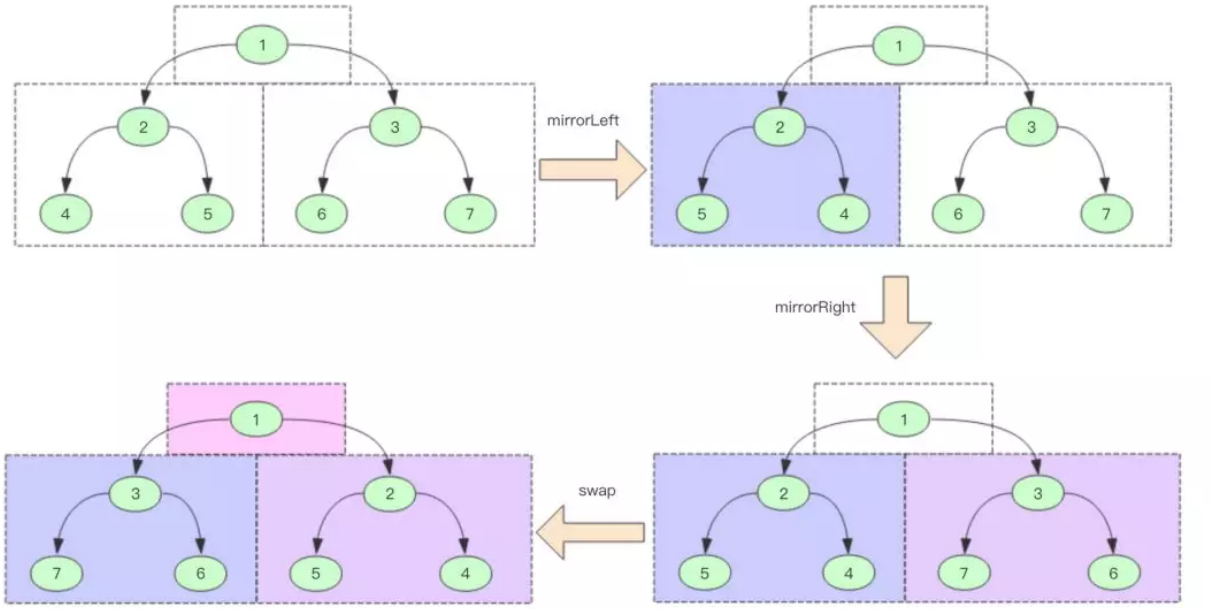



```
class Node<T> {
    T value;
    Node<T> left;
    Node<T> right;

    public String toString() {
        // 叶节点
        if (left == right) {
            return String.format("[%d]", value);
        }
        // 中间节点
        return String.format("(%d, %s, %s)", value, left, right);
    }
}

Node<Integer> node2 = new Node<>(2);
Node<Integer> node3 = new Node<>(3);
Node<Integer> node1 = new Node<>(1, node2, node3);

System.out.println(node1);
System.out.println(node2);
```

```
System.out.println(node3);
--------------------
[2]
[3]
(1,  [2],  [3])
```

递归镜像二叉树

镜像二叉树有两种算法，一种是递归，一种是迭代。递归的算法简单易于理解，我们先使用递归算法来求解。递归的思想就是深度遍历，遇到一个节点，先递归镜像它的左子树，再递归镜像它的右子树，然后再交换自己的左右子树。如果遇到的是叶子节点，就不必处理了。为了避免无限递归，一定要及时设置好递归的停止条件，在这里停止条件就是遇到了叶节点。

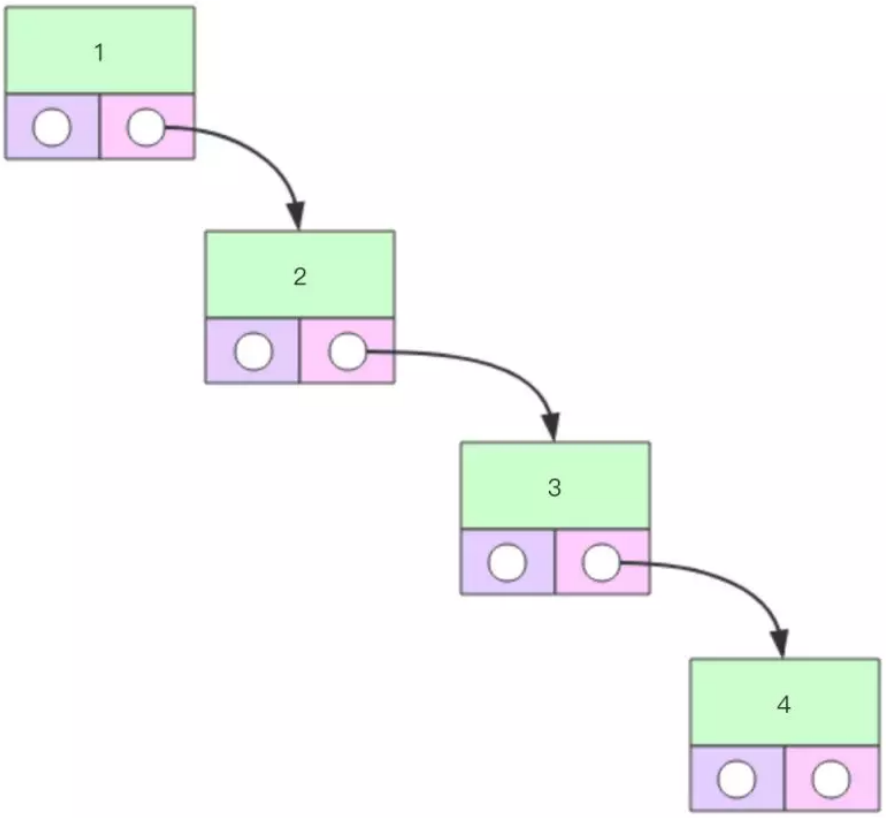



```
public  void  mirrorFrom(Node<T>  node)  {
        //  叶子结点
        if  (node.left  ==  node.right)  {
                return;
        }
        //  递归镜像左子树
        if  (node.left  !=  null)
                mirrorFrom(node.left);
        //  递归镜像右子树
        if  (node.right  !=  null)
```

```
                mirrorFrom(node.right);
        //  交换当前节点的左右子树
        Node<T>  tmp  =  node.left;
        node.left  =  node.right;
        node.right  =  tmp;
}
```

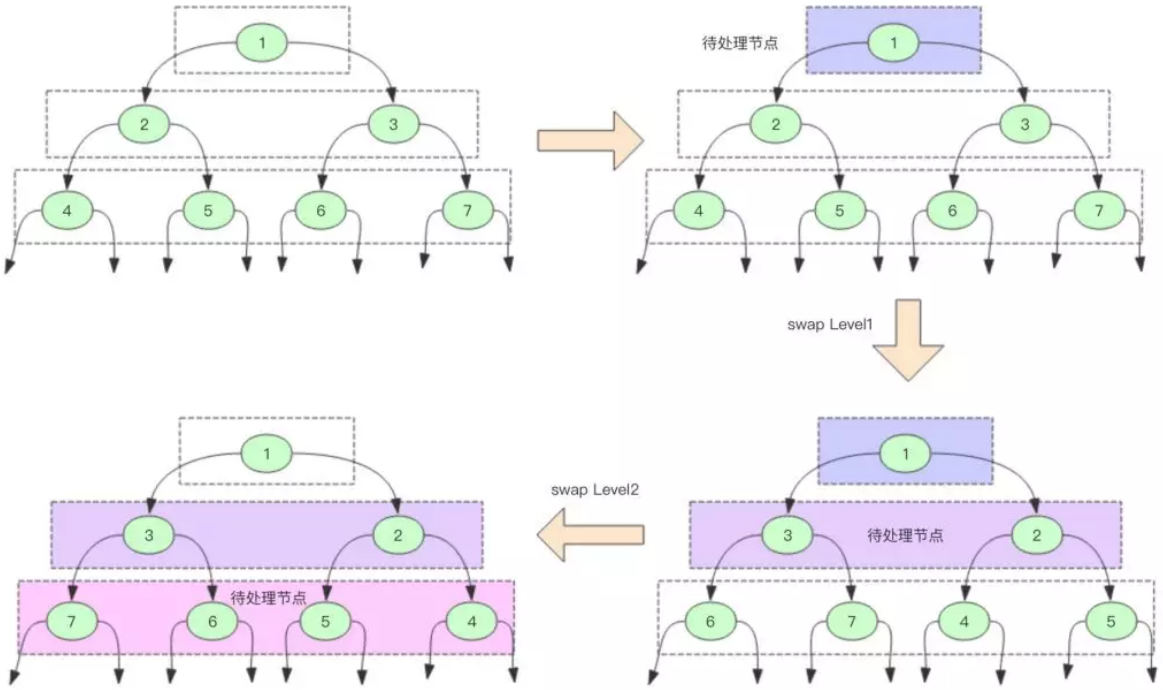
迭代镜像二叉树

递归算法的优势在于逻辑简单，缺点在于每一次递归调用函数都会增加一个新的函数堆栈，如果树的深度太深，函数的堆栈内存就会持续走高，一不小心就会触发臭名昭著的异常 StackOverflowException。如果二叉树分布比较均匀，那么树就不会太深，但是遇到偏向的二叉树，比如所有的子节点都挂在了右节点上，二叉树就退化成了线性链表，链表的长度就是树的深度，那这颗树的深度就比较可怕了。



所以下面我来介绍第二种算法 —— 迭代算法。迭代的基本思想就是将递归算法转换成循环算法，用一个 for 循环来交换所有节点的左右子树。我们需要再重新理解一下算法的目标，这个目标非常简单，就是遍历整颗二叉树，将遍历途中遇到的所有中间节点的左右指针交换一下。

那如何设计这个循环呢？一个很明显的方法是分层循环，第一次循环处理第 1 层二叉树节点，也就是唯一的根节点。下一个循环处理第 2 层二叉树节点，也就是根节点的两个儿子。如此一直处理到最底层，循环的终止条件就是后代节点没有了。所以我们需要使用一个容器来容纳下一次循环需要处理的后代节点。




```
public MirrorBinaryTree<T> mirrorByLoop() {
        // 空树不必处理
        if (root == null) {
                return this;
        }
        // 当前循环需要处理的节点
        LinkedList<Node<T>> expandings = new LinkedList<>();
        expandings.add(root);
        // 没有后台节点就可以终止循环
        while (!expandings.isEmpty()) {
                // 下一次循环需要处理的节点
                // 也就是当前节点的所有儿子节点
                LinkedList<Node<T>> nextExpandings = new LinkedList<>();
                // 遍历处理当前层的所有节点
                for (Node<T> node : expandings) {
                        // 将后代节点收集起来，留着下一次循环
```

```
                    if (node.left != null) {
                        nextExpandings.add(node.left);
                    }
                    if (node.right != null) {
                        nextExpandings.add(node.right);
                    }
                    // 交换当前节点的左右指针
                    Node<T> tmp = node.left;
                    node.left = node.right;
                    node.right = tmp;
                }
                // 将后代节点设置为下一轮循环的目标节点
                expandings = nextExpandings;
            }
        return this;
    }
```

完整代码

下面的完整代码可以拷贝过去直接运行，如果读者还是不够明白欢迎在留言区及时提问。

```
package leetcode;

import java.util.LinkedList;

public class MirrorBinaryTree<T> {

    static class Node<T> {
        T value;
        Node<T> left;
        Node<T> right;

        Node(T value) {
            this.value = value;
        }
```

```
        Node(T value, Node<T> left, Node<T> right) {
            this(value);
            this.left = left;
            this.right = right;
        }

        public String toString() {
            if (left == right) {
                return String.format("[%d]", value);
            }
            return String.format("
(%d, %s, %s)", value, left, right);
        }

    }

    private Node<T> root;

    public MirrorBinaryTree(Node<T> root) {
        this.root = root;
    }

    public MirrorBinaryTree<T> mirrorByLoop() {
        if (root == null) {
            return this;
        }
        LinkedList<Node<T>> expandings = new LinkedList<>();
        expandings.add(root);
        while (!expandings.isEmpty()) {
            LinkedList<Node<T>> nextExpandings = new LinkedList<>
();
            for (Node<T> node : expandings) {
                if (node.left != null) {
                    nextExpandings.add(node.left);
                }
```

```
                    if (node.right != null) {
                        nextExpandings.add(node.right);
                    }
                    Node<T> tmp = node.left;
                    node.left = node.right;
                    node.right = tmp;
                }
                expandings = nextExpandings;
            }
            return this;
        }

        public MirrorBinaryTree<T> mirrorByRecursive() {
            mirrorFrom(root);
            return this;
        }

        public void mirrorFrom(Node<T> node) {
            if (node.left == node.right) {
                return;
            }

            if (node.left != null)
                mirrorFrom(node.left);
            if (node.right != null)
                mirrorFrom(node.right);

            Node<T> tmp = node.left;
            node.left = node.right;
            node.right = tmp;
        }

        public String toString() {
            if (root == null) {
                return "()";
```

```
            }
            return root.toString();
        }

        public static void main(String[] args) {
            Node<Integer> root = new Node<>(
                    1,
                    new Node<>(
                            2,
                            new Node<>(4),
                            new Node<>(
                                    5,
                                    new Node<>(8),
                                    null)),
                    new Node<>(
                            3,
                            new Node<>(
                                    6,
                                    null,
                                    new Node<>
(9)),
                            new Node<>(7)));
            MirrorBinaryTree<Integer> tree = new MirrorBinaryTree<>
(root);
            System.out.println(tree);
            tree.mirrorByRecursive();
            System.out.println(tree);
            tree.mirrorByLoop();
            System.out.println(tree);
        }

}
---------------
(1, (2, [4], (5, [8], null)), (3, (6, null, [9]), [7]))
(1, (3, [7], (6, [9], null)), (2, (5, null, [8]), [4]))
```

```
(1, (2, [4], (5, [8], null)), (3, (6, null, [9]), [7]))
```

扩展思考：为什么镜子里面左右是反过来的，但是上下不是？这不是一道编程题，但是确实不容易回答。