树形DP

使用前提：如果题目求解目标是S规则，则求解流程可以定成以每一个节点为头结点的子树在S规则下的每一个答案，并且最终答案一定在其中。

1. 树形dp套路第一步：以某个节点X为头节点的子树中，分析答案有哪些可能性，并且这种分析是以X的左子树、X的右子树和X整棵树的角度来考虑可能性的；
2. 树形dp套路第二步：根据第一步的可能性分析，列出所有需要的信息；
3. 树形dp套路第三步：合并第二步的信息，对左树和右树提出同样的要求，并写出信息结构；
4. 树形dp套路第四步：设计递归函数，递归函数是处理以X为头节点的情况下的答案。包括设计递归的basecase，默认直接得到左树和右树的所有信息，以及把可能性做整合，并且要返回第三步的信息结构这四个小步骤；

二叉树节点间的最大距离问题

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，到达节点b时路径上的节点个数叫作a到b的距离，那么二叉树任何两个节点之间都有距离，求整棵树上的最大距离。

思路：

1. 设以X为头结点的整棵树的最大距离分两种情况讨论：
   - X不参与，要求返回的信息是左子树的最大距离maxdistance和右子树的最大距离maxdistance。maxdistance在左子树的最大距离maxdistance和右子树的最大距离中较大者；
   - X参与，maxdistance为左高度height+1+右高度height；
   - maxdistance为max(X不参与,X参与)
2. 返回值结构：maxdistance、height;

```
function getMaxDistance(head) {
    // 返回值结构
    function Info(maxDistance, height) {
        this.maxDistance = maxDistance;
        this.height = height;
    }
    // base case
    if (head === null) {
        return new Info(0, 0);
    }
    const { maxDistance: leftDistance, height: leftHeight } = getMaxDistance(
        head.left
    );
    const { maxDistance: rightDistance, height: rightHeight } = getMaxDistance(
        head.right
    );

    let height = Math.max(leftHeight, rightHeight) + 1;
    let maxDistance = Math.max(
        leftHeight + rightHeight + 1,
        Math.max(leftDistance, rightDistance)
    );

    return new Info(maxDistance, height);
```

```
}
```

派对的最大快乐值

员工信息的定义如下:

```
interface Employee {
    happy: number; // 这名员工可以带来的快乐值
    subordinates: Employee[];  // 这名员工有哪些直接下级
}
```

公司的每个员工都符合 Employee 类的描述。整个公司的人员结构可以看作是一棵标准的、 没有环的多叉树。树的头节点是公司唯一的老板。除老板之外的每个员工都有唯一的直接上级。 叶节点是没有任何下属的基层员工(subordinates列表为空)，除基层员工外，每个员工都有一个或多个直接下级。

这个公司现在要办party，你可以决定哪些员工来，哪些员工不来。但是要遵循如下规则。

1. 如果某个员工来了，那么这个员工的所有直接下级都不能来
2. 派对的整体快乐值是所有到场员工快乐值的累加
3. 你的目标是让派对的整体快乐值尽量大

给定一棵多叉树的头节点boss，请返回派对的最大快乐值。

思路:

1. 以X为头获得的最大快乐值分为两种情况讨论(X的subordinates为[A,B,C,...]):
   - X参与：X快乐值+A不来的整棵树的最大快乐值+B不来的整棵树的最大快乐值+C不来的整棵树的最大快乐值+...
   - X不参与：0+max{A不来的整棵树的最大快乐值，A来的整棵树的最大快乐值}+max{B来的整棵树的最大快乐值，B来的整棵树的最大快乐值}+...

```
function getMaxHappiness(head) {
    function Info(headNotIn, headIn) {
        // headNotIn: 头节点不参与整个树的最大快乐值
        // headIn: 头节点参与整个树的最大快乐值
        this.headNotIn = headNotIn;
        this.headIn = headIn;
    }
    function process(head) {
        if (head === null) {
            return new Info(0, 0);
        }

        let headNotInMaxHappiness = 0,
            headInMaxHappiness = head.happy;

        for (let i = 0; i < head.subordinates.length; i++) {
            const { headNotIn, headIn } = process(head.subordinates[i]);
            headNotInMaxHappiness += Math.max(headNotIn, headIn);
            headInMaxHappiness += headNotIn;
        }

        return new Info(headNotInMaxHappiness, headInMaxHappiness);
    }
```

```
    const { headNotIn, headIn } = process(head);

    return Math.max(headNotIn, headIn);
}
```

Morris遍历

一种遍历二叉树的方式 时间复杂度O(N) 空间复杂度O(1)

Morris遍历的实质: 通过来回标记遍历整棵树

*Morris*遍历细节

假设来到当前节点cur，开始时cur来到头节点位置

1. 如果cur没有左孩子，cur向右移动(cur = cur.right)
2. 如果cur有左孩子，找到左子树上最右的节点mostRight：
   - 如果mostRight的右指针指向空，让其指向cur，然后cur向左移动(cur = cur.left)
   - 如果mostRight的右指针指向cur，让其指向null，然后cur向右移动(cur = cur.right)
3. cur为空时遍历停止

无左树，只有一次到达自己

有左树，会经过自己两次，根据左树的最右节点指向判断第几次达到自己

```
function morris(head) {
    if (head === null) {
        return;
    }
    let cur = head;
    let mostRight = null;

    while (cur !== null) {
        mostRight = cur.left;
        if (mostRight !== null) {
            // cur有左孩子

            // 找到左孩子的最右节点
            while (mostRight.right !== null && mostRight.right !== cur) {
                mostRight = mostRight.right;
            }

            if (mostRight.right === null) {
                // 第一次经过cur
                mostRight.right = cur;
                cur = cur.left;
            } else {
                // 第二次经过cur
                mostRight.right = null;
                cur = cur.right;
            }
        } else {
```

```
            // cur没有左孩子
            cur = cur.right;
        }
    }
}
```

前序遍历

- 只到达自己一次直接打印；
- 能够到达自己两次，在第一次打印；

```
// 前序
function morrisPre(head) {
    if (head === null) {
        return;
    }
    let cur = head;
    let mostRight = null;

    while (cur !== null) {
        mostRight = cur.left;
        if (mostRight !== null) {
            // cur有左孩子

            // 找到左孩子的最右节点
            while (mostRight.right !== null && mostRight.right !== cur) {
                mostRight = mostRight.right;
            }

            if (mostRight.right === null) {
                // 第一次经过cur
                console.log(cur.value);
                mostRight.right = cur;
                cur = cur.left;
            } else {
                // 第二次经过cur
                mostRight.right = null;
                cur = cur.right;
            }
        } else {
            // cur没有左孩子
            console.log(cur.value)
            cur = cur.right;
        }
    }
}
```

中序遍历

- 只到达自己一次直接打印；
- 能够到达自己两次，在第二次打印；

```
// 中序
function morrisIn(head) {
    if (head === null) {
        return;
    }
    let cur = head;
    let mostRight = null;

    while (cur !== null) {
        mostRight = cur.left;
        if (mostRight !== null) {
            // cur有左孩子

            // 找到左孩子的最右节点
            while (mostRight.right !== null && mostRight.right !== cur) {
                mostRight = mostRight.right;
            }

            if (mostRight.right === null) {
                // 第一次经过cur
                mostRight.right = cur;
                cur = cur.left;
            } else {
                // 第二次经过cur
                console.log(cur.value);
                mostRight.right = null;
                cur = cur.right;
            }
        } else {
            // cur没有左孩子
            console.log(cur.value)
            cur = cur.right;
        }
    }
}
```

后序遍历

- 能够到达两次的节点，在第二次到达时逆序打印其左子树的右边界（将以cur.left为头的右边界看作成链表反转打印，打印完成在逆序回去）；
- 遍历完成后，打印整个树的右边界；

```
// 后序
function reverseNode(head) {
    if (head === null) {
        return null;
    }
    let prev = null;
    let cur = head;
    let right = null;
    while (cur !== null) {
```

```
        right = cur.right;
        cur.right = prev;
        prev = cur;
        cur = right;
    }
    return prev;
}
function printEdge(head) {
    let cur = reverseNode(head);
    while (cur !== null) {
        console.log(cur.value);
        cur = cur.right;
    }
    reverseNode(head);
}
function morrisPos(head) {
    if (head === null) {
        return;
    }
    let cur = head;
    let mostRight = null;

    while (cur !== null) {
        mostRight = cur.left;
        if (mostRight !== null) {
            // cur有左孩子

            // 找到左孩子的最右节点
            while (mostRight.right !== null && mostRight.right !== cur) {
                mostRight = mostRight.right;
            }

            if (mostRight.right === null) {
                // 第一次经过cur
                mostRight.right = cur;
                cur = cur.left;
            } else {
                // 第二次经过cur
                mostRight.right = null;
                printEdge(cur.left)
                cur = cur.right;
            }
        } else {
            // cur没有左孩子
            cur = cur.right;
        }
    }
    printEdge(head)
}
```