二叉树 ☆☆

题目描述

小杨有一棵包含 n 个节点的二叉树，且根节点的编号为 1。这棵二叉树任意一个节点要么是白色，要么是黑色。之后小杨会对这棵二叉树进行 q 次操作，每次小杨会选择一个节点，将以这个节点为根的子树内所有节点的颜色反转，即黑色变成白色，白色变成黑色。

小杨想知道 q 次操作全部完成之后每个节点的颜色。

输入输出格式

输入格式：

第一行：正整数 n，表示节点数量

第二行：$(n-1)$ 个正整数，表示节点 2 到 n 的父亲节点

第三行：长度为 n 的 01 串，表示初始颜色

第四行：正整数 q，表示操作次数

接下来 q 行：每行一个正整数 a_i，表示操作节点

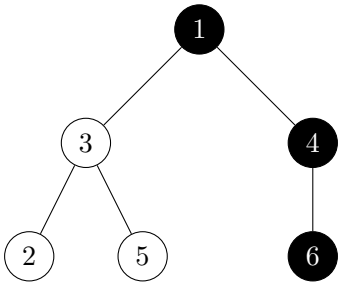
输出格式：

一行：长度为 n 的 01 串，第 i 位为 0 表示节点 i 为白色，第 i 位为 1 表示节点 i 为黑色

输入示例	输出示例
6 3 1 1 3 4 100101 3 1 3 2	010000

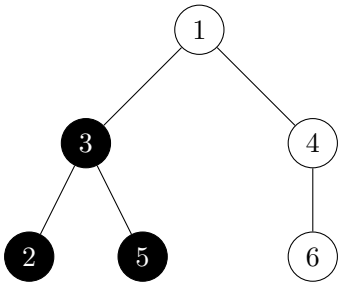
样例解释

1. 初始状态：



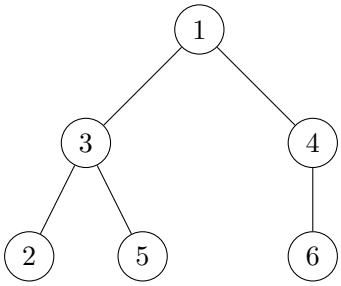
- 节点颜色：1(1), 2(0), 3(0), 4(1), 5(0), 6(1)
- 颜色序列：100101

2. **第一次操作（节点1）：**



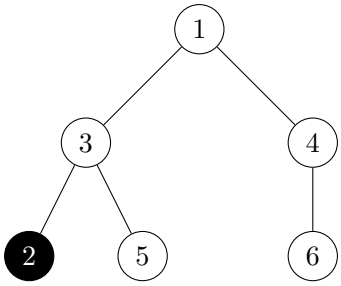
- **反转范围**：整个树（所有节点）
- 反转后颜色：1(0), 2(1), 3(1), 4(0), 5(1), 6(0)
- 颜色序列：011010

3. **第二次操作（节点3）：**



- **反转范围**：节点3及其子树（节点3,2,5）
- 反转后颜色：1(0), 2(0), 3(0), 4(0), 5(0), 6(0)
- 颜色序列：000000

4. **第三次操作（节点2）：**



- **反转范围**：节点2（只包含节点2）
- 反转后颜色：1(0), 2(1), 3(0), 4(0), 5(0), 6(0)
- 颜色序列：010000

背景知识

在解决树结构相关问题时，图的存储方式至关重要。以下是几种常见的图存储方法：

1. **邻接矩阵**

   - 使用二维数组 $graph[i][j]$ 表示节点 i 到节点 j 的边
   - 用 0 表示无边，非 0 表示有边且数值为边的权值

```
vector<vector<int>> graph(n + 1, vector<int>(n + 1, 0));
for (int i = 0; i < m; i++) {
    int u, v, w;
    cin >> u >> v >> w;
    graph[u][v] = w;
    graph[v][u] = w;
}
```

2. **邻接表**

   - 为每个节点维护一个链表，存储该节点的所有相连的节点

```
vector<vector<int>> adj(n + 1);
for (int i = 0; i < m; i++) {
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

3. **优缺点对比**

存储方式	空间复杂度	查询边(u,v)	遍历邻居	优点	缺点
邻接矩阵	$O(n^2)$	$O(1)$	$O(n)$	查询速度快 适合稠密图	空间占用大 不适合稀疏图
邻接表	$O(n+m)$	$O(\deg(u))$	$O(\deg(u))$	空间效率高 适合稀疏图	查询边需遍历 实现稍复杂

算法分析

1. **直接模拟方法**

   对于每次操作，遍历以该节点为根的子树，反转所有节点的颜色

   这种方法的时间复杂度为 $O(n \cdot q)$，在 $n, q = 10^5$ 时会超时。

2. **优化方法：标记传递**

   利用标记数组记录每个节点被反转的次数，最后统一处理。

   （1） **核心思想**

   - 使用一个标记数组 `tag` 记录每个节点被反转的次数

- 对于每次操作，只在操作节点上增加标记
- 通过一次遍历，将父节点的标记传递给子节点
- 如果反转次数是奇数，则反转当前节点, 否则不反转

（2） **算法步骤**

- 构建二叉树结构（使用邻接表）
- 初始化标记数组 tag 为 0
- 对于每个操作，tag[操作节点]++
- 使用 BFS 从根节点开始遍历，将父节点的标记累加到子节点

```
queue<pair<int, int>> qu;  // 当前节点, 父节点
qu.push({1, 0});
while (!qu.empty()) {
    int u = qu.front().first;
    int fa = qu.front().second;
    qu.pop();

    for (int v : adj[u]) {
        if (v == fa) continue;
        tag[v] += tag[u];
        qu.push({v, u});
    }
}
```

- 计算每个节点的最终颜色

```
for (int i = 1; i <= n; i++) {
    if (tag[i] % 2 == 1) {// 反转颜色
    color[i] = (color[i] == '0') ? '1' : '0';
    }
}
```

拓展思考

- 如果颜色的数目修改为k种，且每次操作是将该节点及其子树颜色全部设置为x,那么算法该如何修改？
- 如果树不是二叉树，而是多叉树，算法是否仍然适用？