Note: In this problem you **must NOT** generate any output on your own. Any such solution will be considered as being against the rules and its author will be disqualified. The output of your solution must be generated by the uneditable code provided for you in the solution template.

An important concept in Object-Oriented Programming is the open/closed principle, which means writing code that is open to *extension* but closed to *modification*. In other words, new functionality should be added by writing an extension for the existing code rather than modifying it and potentially breaking other code that uses it. This challenge simulates a real-life problem where the open/closed principle can and should be applied.

A *Tree* class implementing a rooted tree is provided in the editor. It has the following publicly available methods:

- `getValue()`: Returns the *value* stored in the node.
- `getColor()`: Returns the *color* of the node.
- `getDepth()`: Returns the depth of the node. Recall that the depth of a node is the number of edges between the node and the tree's root, so the tree's root has depth 0 and each descendant node's depth is equal to the depth of its parent node $+1$.

In this challenge, we treat the internal implementation of the tree as being closed to modification, so we cannot directly modify it; however, as with real-world situations, the implementation is written in such a way that it allows external classes to extend and build upon its functionality. More specifically, it allows objects of the *TreeVis* class (a Visitor Design Pattern) to visit the tree and traverse the tree structure via the `accept` method.

There are two parts to this challenge.

Part I: Implement Three Different Visitors

Each class has three methods you must write implementations for:

1. `getResult()`: Return an integer denoting the $\boldsymbol{result}$, which is different for each class:
    - The *SumInLeavesVisitor* implementation must return the sum of the values in the tree's *leaves* only.
    - The *ProductRedNodesVisitor* implementation must return the *product* of values stored in all *red* nodes, including leaves, computed modulo $10^9 + 7$. Note that the product of zero values is equal to 1.
    - The *FancyVisitor* implementation must return the absolute difference between the sum of values stored in the tree's *non-leaf nodes at even depth* and the sum of values stored in the tree's *green leaf nodes*. Recall that zero is an even number.
2. `visitNode(TreeNode node)`: Implement the logic responsible for visiting the tree's *non-leaf* nodes such that the *getResult* method returns the correct $\boldsymbol{result}$ for the implementing class' visitor.
3. `visitLeaf(TreeLeaf leaf)`: Implement the logic responsible for visiting the tree's *leaf* nodes such that the *getResult* method returns the correct $\boldsymbol{result}$ for the implementing class' visitor.

Part II: Read and Build the Tree

Read the n-node tree, where each node is numbered from 1 to n. The tree is given as a list of node values $(x_1, x_2, \ldots, x_n)$, a list of node colors $(c_1, c_2, \ldots, c_n)$, and a list of edges. Construct this tree as an instance of the *Tree* class. The tree is always rooted at node number 1.

Your implementations of the three visitor classes will be tested on the tree you built from the given input.

Input Format

The first line contains a single integer, n, denoting the number of nodes in the tree. The second line contains n space-separated integers describing the respective values of $x_1, x_2, \ldots, x_n$.
The third line contains n space-separated binary integers describing the respective values of $c_1, c_2, \ldots, c_n$. Each c_i denotes the color of the i^{th} node, where 0 denotes *red* and 1 denotes *green*.
Each of the $n - 1$ subsequent lines contains two space-separated integers, u_i and v_i, describing an

edge between nodes u_i and v_i.

Constraints

- $2 \leq n \leq 10^5$
- $1 \leq x_i \leq 10^3$
- $c_i \in \{0, 1\}$
- $1 \leq v_i, u_i \leq n$
- It is guaranteed that the tree is rooted at node 1.

Output Format

Do not print anything to stdout, as this is handled by locked stub code in the editor. The three `getResult()` methods provided for you must return an integer denoting the $\boldsymbol{result}$ for that class' visitor (defined above). Note that the value returned by *ProductRedNodesVisitor*'s *getResult* method must be computed modulo $10^9 + 7$.

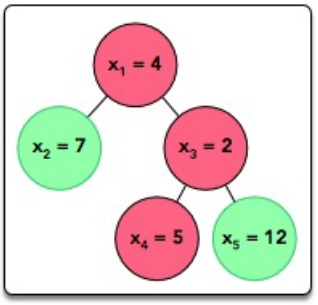
Sample Input

```
5
4 7 2 5 12
0 1 0 0 1
1 2
1 3
3 4
3 5
```

Sample Output

```
24
40
15
```

Explanation



Locked stub code in the editor tests your three class implementations as follows:

1. Creates a *SumInLeavesVisitor* object whose *getResult* method returns the *sum* of the *leaves* in the tree, which is $7 + 5 + 12 = 24$. The locked stub code prints the returned value on a new line.
2. Creates a *ProductOfRedNodesVisitor* object whose *getResult* method returns the *product* of the *red* nodes, which is $4 \cdot 2 \cdot 5 = 40$. The locked stub code prints the returned value on a new line.
3. Creates a *FancyVisitor* object whose *getResult* method returns the *absolute difference* between the *sum* of the values of *non-leaf* nodes at *even depth* and the *sum* of the values of *green leaf nodes*, which is $|4 - (7 + 12)| = 15$. The locked stub code prints the returned value on a new line.