First let's try to solve the problem of a given a tree, fixed modulo and only query operation's. (Assuming each node has some value v).

So, for solving this problem we will root the tree at 1. Now for every Query operation of with a given r and t, there are 3 possible cases:

1. r is an ancestor of t and r.
2. t is an ancestor of t and r.
3. Neither r nor t is an ancestor for t and r.

Now for Case 1 and Case 3, the answer is the same as the answer with root 1. For Case 2, the answer is the same as querying the whole tree with just 1 edge deletion. I would highly recommend the reader to take a pause here and think which edge it could be.

Consider the path from r to t, let x be the node which is closest to t in the path from r to t [alternatively let x be the node which is at a distance of 1 edge from t in the path from t to r], if we delete the edge and consider the remaining tree, we will find that the remaining tree is the one in which we are interested in. But we really do not want to delete an edge and we also want to solve this problem.

So, if we find the sum of the subtree rooted at x and subtract it from the sum of the full tree, we can get the answer which is required.

Now when we have solved this problem, i would further urge my readers to think on how to solve the problem with one more conition added, the UPDATE operation, and keeping the modulo still fixed.

You can easily take care of Update operation with the logic explained above to tranfer the queries from a tree rooted at 1 to a tree rooted at any node. Only pre-requisite i am assuming currently is that you know how to solve the problem with fixed root and fixed modulo. [Hint: You can store the dfs traversal order, then any continous segment of dfs traversal will correspond to a subtree in the tree, now the update and query can be done via segment trees or the binary indexed tree's]

Now, let's come back to the original problem of mantaining it with any modulo. For this problem, the first approach that comes to mind is to store 101 trees (because the constraint on mod was less than 102) for each mod, and query from specific values. But that much memory is not allowed and is a ultra slow solution to pass.

Second approach that comes to mind is to store modulo using every high power of each prime number, there are 26 prime numbers and we need 26 data structures to solve it for each high power of prime number and then combine using Chinese Remainder theorem. High power of 2 is 64 (becuase 128 > 101), high power of 3 is 81, etc,etc.

It has been rude on my side to make that solution TLE because this problem can be solved using 5 data structures only instead of 26! All test cases were made to ensure that this solution does not pass, I apologise and I promise that there will be atleast 1-2 easy testcases from next time in my problem:).

Further let me explain you the solution with 5 binary indexed trees:

Mantain 5 data structures under these 5 modulos:

a. 64*81*25*49*11*13
b. 17*19*23*29*31*37
c. 43*47*53*59*79
d. 61*67*71*73*41
e. 83*89*97*101

Now if you want sum mod p1^e1 * p2^e2 *.....* pn^en, simply see which all branches each part is, and calculate from each branch and later merge them using Chinese remainder theorem.

For example, I want to have answer mod 2*3*17 (which is 102 and which is not in query but is only for demonstration purpose)

- we will find the result modulo (a) and then take mod 6 on the result.
- we will find the result modulo (b) and then take mod 17 on the result.
- we will combine both results using Chinese Remainder theorem.