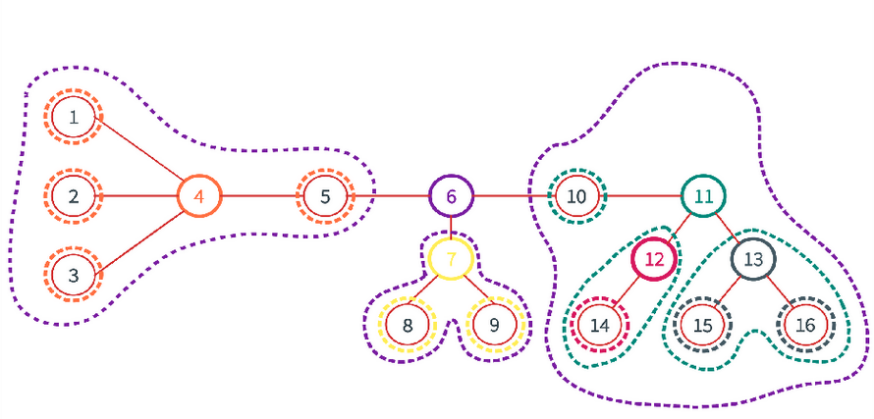
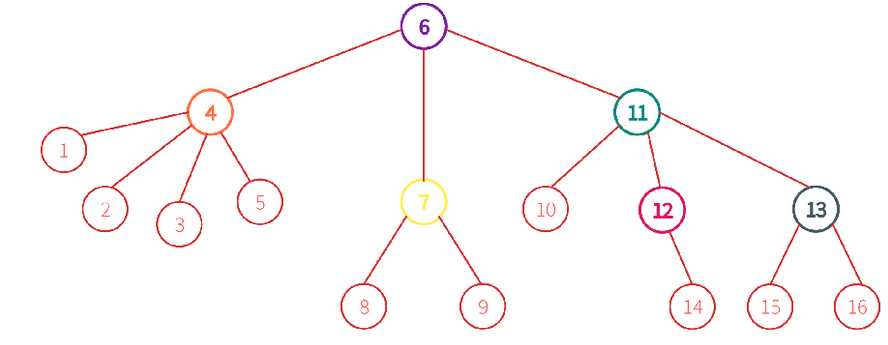
Centroid

A centroid is a node whose removal splits the given tree into a forest of trees, where each of the resulting tree contains no more than N/2 nodes.

Procedure to find centroid and proof of its existence:

Start with any node u. If u meets the requirements, it is a centroid. Else, find the subtree with the largest size, and go to the root of the subtree. Repeat this process until a centroid is found. We never go back to old vertices because our current one is larger than N/2, meaning the old ones are less than N/2. Since we don’t repeat vertices, and the number of vertices is finite, the process must terminate, which means a centroid must exist.

Centroid tree

After removing the centroid, the tree decomposes into 2 or more trees, each with size < N/2. We can find the centroid of the smaller trees, and connect it to the centroid of the original tree.

Properties:

* The centroid tree contains all N nodes.
* The height of the centroid tree is at most .
* Consider any 2 nodes A and B, and their LCA C. The path between them in the original tree can be broken down to A to C and B to C in the centroid tree.
  + If we assign labels to the centroids in the order in which they are removed from the graph, C would be the node with the smallest label in the path from A to B in the original tree.
* Hence, we decompose the tree in to different paths, such that any path in the original tree is a concatenation of 2 different path from that set.
* We can find LCA in centroid tree in time just by moving up because the height is .

Problem 1

Given a tree with vertices. How many paths are there in the tree with length ?

We need to solve the problem: How many paths going through the centroid have length ?

We can solve this by dfs from the centroid, counting the number of paths with different lengths, then iterating the length and adding . The path must go through the centroid!! (cannot just dfs once)

After that, delete the centroid, find the centroid of the subtrees, and repeat the process.

Since the height is at most , any node is at most under subtrees. Since any node is at most under subtrees, even if we do every time, the total time complexity is still . (similar to IOI race)

[Problem 2](https://codeforces.com/contest/342/problem/E)

Given a tree with vertices. Initially, node 1 is painted red, and the rest is painted blue. There will be 2 types of operations: update and query. In an update, paint a specified blue node red. In a query, find the shortest distance from a specified node to a red node.

Let denote the minimum distance to a red node from a centroid in its corresponding subtree (in the centroid tree).

To paint node red, we move up all ancestors (including itself) of in the centroid tree and update .

To query for node , again we move up all ancestors (including itself) of in the centroid tree and update .

Remember, if is the parent of in the centroid tree, that doesn’t mean the same in the original tree, it just means ’s subtree is part of ’s subtree.

Time complexity is because distance can be calculated in time using LCA.

Since the height of the centroid tree is at most , we can also precompute a distance array from all nodes to their ancestors in time.

Total time complexity: similar to this [problem](https://codeforces.com/gym/100570/problem/F)

Note: cannot be lazy and don’t use LCA to calculate distances between centroids

[Problem 3](https://www.spoj.com/problems/QTREE5/)

Very similar to the previous problem, but a red node can be painted blue.

Instead of maintaining the minimum for each centroid, just maintain a multiset.

Total time complexity:

Problem 4

Given a weighted tree with vertices, and updates and queries. Each update tells you to colour all vertices that are at most distance from vertex with colour . Each query requires you to output colour of vertex .

We can solve this problem similarly to problem 2.

For each centroid, maintain a stack, where the distance of the queries are decreasing and time is increasing. (in the order of the queries)

When we want to update vertex , push into the stack of if is smaller than the last element in the stack. If its larger, pop until it is not. Why does this work? because if its larger, that means all vertices changed in the previous update is included in the current update!

Also, update the centroid-ancestors of . Remember to subtract the distance between and the centroid-ancestor from before updating them. The vertices included in the centroid-ancestors may overlap