Exercise 2. Show that if the tree has height h, then this algorithm takes $O((n+q)h)$ time, and that this is tight; it is possible for it to run in $\Theta((n+q)h)$ time in the worst case (if you're unlucky with the queries). (Hint: There are $\Theta(h)$ "recursion layers". Hint 2: For each "recursion layer", the corresponding trees and queries are pairwise disjoint.)

We prove both upper and lower bounds.

[Upper Bound] The algorithm recurses on the root's subtrees. If the tree has height h, then in the worst case, the recursion depth is Θ(h), a path rooted at one end. At each recursion layer:

- Preprocessing for that layer, computing minima to root takes $O(n_{layer})$ time across all subtrees in that layer.
- Queries that do not pass through the current root are delegated downward, but each query is processed in at most one node per layer. Thus, total work over all layers, is O((n + q)h).

[Lower Bound]

We construct a path of length h (so n = h + 1), rooted at one end. The first root has one child containing all other nodes. Each recursion layer reduces the tree by only one node (the current root), giving Θ(h) layers.

Choose queries such that each is of the second kind (i.e., it does not go through current root) until the bottom. Then each query is processed in every layer until it reaches its deepest relevant subtree, i.e., Θ(h) times.

Thus, total work = Θ(nh + qh) = Θ((n + q)h), matching the upper bound.

Therefore, the algorithm runs in Θ((n + q)h) worst-case time.

Exercise 3. Show that any (nonempty) tree has at least one centroid. (Hint: Root the tree arbitrarily, and then show that the node with the least weight $\geq(n/2)$ is a centroid.)

A centroid is a node such that when rooted there, every subtree has size <= n/2.

We root the tree arbitrarily at some node r, then compute subtree sizes for all nodes. Consider the N of nodes whose subtree size in this rooted view is >= n/2.

- r itself is in N because its subtree size is n.
- N is nonempty.

Let c be the node in N with the smallest subtree size.
I claim that c is a centroid.

- For any child v of c, size(v) <= n/2, otherwise, v would be in N with a smaller size than c (since size(v) < size(c)), contradicting minimality.
- The "rest" of the tree (outside c's subtree in the rooted view) has size (n - size(c)) <= n/2 because size(c) >= n/2.

Thus, when rooted at c, every component after removing c has size <= n/2. Hence, c is a centroid.

Exercise 4. Show that any two centroids of a tree are adjacent.

Let u and v be two distinct centroids of a tree T with n nodes. suppose for contradiction, that they are not adjacent.

Consider the unique path $u = x_0, x_1, \ldots, x_k = v$ between them, with $k >= 2$, since they are not adjacent.

Look at the component containing x_1 when u is removed. This component includes $x_2, \ldots$, v and all their subtrees. since u is a centroid, this component's size <= n/2.

Now consider removing v. The component containing x_{k-1} includes all of x_{k-1}'s subtree, which includes u and the other side. Its size = n - size(subtree of v towards u). But the subtree of v towards u is exactly the component from step 4, so its size <= n/2. Hence, the component containing x_{k-1} has size >= n/2.

For v to be a centroid, all components after removing v must have size <= n/2, but the component containing x_{k-1} is >= n/2, so it must be exactly n/2.

If it's exactly n/2, then removing v splits the tree into exactly two components of size n/2 each:

- one containing x_{k-1} (and u)
- the rest

But then x_{k-1} is adjacent to v and lies in the path, so x_{k-1} would also be a centroid, since it's larger side is n/2. Repeating this argument along the path, x_1 becomes a centroid.

Now we check u, the component containing x_1 when removing u has size n/2. The other side, the rest of the tree also has size n/2. But x_1 is a centroid adjacent to u, and both components after removing u have size n/2, so u remains a centroid.

However, this implies that u and x_1 are adjacent centroids, contradicting our assumption that u and v are the only two centroids and are not adjacent since $k >= 2$ implies $x_1 != v$.

Thus, any two centroids must be adjacent.

Exercise 5. Show that any tree has at most two centroids.

From Exercise 4, any two centroids in a tree must be adjacent. Suppose for contradiction, that a tree has three distinct centroids c_1, c_2, c_3. By Exercise 4, each pair must be adjacent, so they form a triangle in the tree. But a tree has no cycles, a triangle is a cycle of three which is impossible.

Thus, there are at most two centroids in a tree.

Exercise 6. Find an O(n) algorithm for finding a centroid in the tree.

1. Root the tree arbitrarily at node r
2. Run a DFS from r to compute:
    a. subtree_size[u] for every node u (size of subtree rooted at u in this rooted view)
3. Initialize centroid = r, n = subtree_size[r].
4. Traverse from the root downward:
    a. While current node c has a child v with subtree_size[v] > n/2:
        i. Move c to v (the heavy child).
5. Return c as the centroid.

This works because at each step, the child with subtree_size > n/2 is unique (if it existed, it would contain more than half the nodes, so no other child can also exceed n/2). Moving to that child reduces the "heavy side" until no child exceeds n/2. Then all components after removing c have size <= n/2, so c is a centroid.

Its time complexity comes from the one DFS used to compute subtree sizes: O(n), at most O(n) steps moving down the tree, with each node visited once in the traversal. Thus, O(n).

Exercise 7. Show that the following returns the centroid: (Hint: Note that there is at most one child with > n/2 nodes.)

```
def find_centroid(r) :  // r is the root
    compute the weights of all nodes in the tree under r
    n := r.weight
    while r has a child x with > n/2 nodes :
        r := x
    return r
```

We claim that the given algorithm returns a centroid.

[Invariant] At the start of each iteration of the while loop, the current node r satisfies:

- r.weight >= ceil(n/2)
- if r.weight = n (i.e., r is the original root), it's fine, otherwise, it was reached by moving from a parent with weight > n/2, so the "outside" part has size < n/2.
- the algorithm maintains the property that the unique heavy child (> n/2) is followed if it exists.

[Initialization] Before the root, r is the original root. Its weight = n, so r.weight >= n/2 holds, and the invariant maintains.

[Maintenance] suppose before an iteration, invariant holds at current r. If r has a child x with weight > n/2, such a child is unique (two would sum > n). We set r = x. Now x.weight > n/2, so x.weight >= ceil(n/2).

The "outside" part for x is everything else, size = n - x.weight < n/2. so the invariant is restored for the next iteration.

[Termination] The loop ends when the current r has no child with weight > n/2. Then all children of r have weight <= n/2. since r.weight >= n/2 by the invariant, the rest of the tree, outside r's subtree has size <= n/2. Therefore, all components after removing r have size <= n/2,

Hence, r is a centroid.

Exercise 8 skipped. Too confusing.

Exercise 9. Use the idea of the centroid decomposition tree to answer path minimum queries online in $\langle n \log n, \log n \rangle$.

First, we find a centroid of the original tree and make it the root of the CDT. We recursively build the CDT for each subtree (connected component after removing the centroid), attaching them as children of the current centroid node. Height of CDT = O(log n).

For each level of the CDT, store the minimum value on the path from that node to that centroid in the original tree. since each node appears in O(log n) levels, total storage = O(n log n).

Given query min_path(u,v):

- In the CDT, find the LCA of u and v in the CDT, call it c.
- By construction of the CDT, the centroid c lies on the path between u and v in the original tree.
- The path u->v in the original tree = u -> c -> v.
- ans = min(min_to_centroid[u][c], min_to_centroid[v][c]), where these values are precomputed minima from node to centroid c.
- CDT height is O(log n), so we can preprocess for LCA in O(n log n) and answer in O(1), or just climb up from both nodes in O(log n).