Hikers

The problem was not intended to be easy, so some level of familiarity with advanced data structures and techniques is expected.

Subtask 1 $n, q, \sum k \leq 10^3$ — 12 points

Since the constraints are small, you can process events directly. Use a Disjoint Set Union (DSU) to keep groups. Each node is either occupied by some group or not occupied at all. When processing an event v, you can go through all nodes in the subtree of v and "lift" them. When a node is already occupied by some other group, you can merge them in DSU and keep the existing group.

You can use path compression technique in DSU to keep the merges efficient. Note that there will be at most $m - 1$ merges.

Time complexity: $O(nq + n \log n)$.

Subtask 2 $p_i = \lfloor \frac{i}{2} \rfloor$ and $n = 2^k - 1$ for some k — 13 points

The key observation: after processing an event v, all leaf nodes in the subtree can be safely deleted.

Since we are given a perfect binary tree, the number of leaf nodes is always more than the number of non-leaf nodes. So the whole complexity can be calculated in terms of leaf nodes in the subtree. But since we are deleting them immediately after a query, we will do $O(n)$ amortized operations.

To prove the statement above: imagine any node v. If it is a leaf, it will be visited once and immediately get deleted. If it is not a leaf, there is always a leaf node beneath it that "pays" its visit cost.

Time complexity: $O(n \log n + q)$.

Some important observations

- If there are multiple hikers on the same node, we can assume there is only one. Obviously, this does not change anything.
- We can forget about hikers completely and focus only on nodes. This is not too obvious. Imagine $m = n$ and each node is a group on its own. If we get to know which nodes get merged after each event, we can easily solve the original problem with groups as well. When two nodes u and v are getting merged, you can just check which groups u and v belong to.

Thus, from now on we assume $m = n$ and all groups are of size 1. I.e. group = hiker = node.

Subtask 3 $p_i = i - 1$ — 18 points

We are dealing with a chain $1 - \cdots - n$. Each event becomes a "move the suffix" event.

Suppose we are processing an event v. Find the next vertex on the chain. Let's call it u. Nodes v and u get merged and v gets deleted. After that, all nodes in the suffix move one step to the left.

We can just simulate the process above by storing a binary sequence $b_1, \ldots, b_n$. b_i will denote if node i still exists or not.

When given an event v, if the number of existing nodes $\sum b_i$ is less than v, we can just ignore the event. Otherwise, we find the v-th and $v + 1$ nodes from the left and do exactly the procedure described above. This can be done with a segment tree that stores count of existing nodes in its subtree.

Time complexity: $O((n + q) \log n)$.

Subtask 4 $v_i = 1$ in all events — 14 points

Here the process of "lifting" nodes is very trivial. We just need to make sure we are merging all nodes properly.

Calculate heights h_v for all nodes. Height is simply a distance to the root node 1. In particular, $h_1 = 0$. Which node will be at vertex v after k events? Any node u such that $h_u = h_v + k$ should work.

Let's say we are processing k-th event.

First, find all nodes with at least 2 children. Let's say one of them is u. We will go through every child of u. In each child's subtree, find an arbitrary node at height $h_u + k$. There will be at least one (otherwise that child would be already deleted). After you find those nodes, you can merge them together.

Then, just delete all leaf nodes from the tree. They will not be needed anymore.

We can store the leaf nodes and nodes with at least 2 children with sets. When a leaf gets deleted, make sure to update those sets properly.

Each operation is set insertion/removal and merging two nodes. It can be seen that we will $O(n)$ operations in total. We will remove each node exactly one when it's a leaf. And every merge we do is a valid one, so there are at most $n - 1$ of them.

Time complexity: $O((n + q) \log n)$.

Subtask 5 $n, q, \sum k \leq 10^5$ — 24 points

This subtask is designed for sub-optimal solutions, which have time complexities $O((n + q) \log^2 n)$ or $O((n + q)\sqrt{n})$. We will not be describing those ideas, but note that they exist.

Subtask 6 No additional constraints — 19 points

The solution we are aiming for has time complexity $O((n + q) \log n)$ and involves several more ideas. Similar to subtask 5, we would ideally like to efficiently handle all merges in the subtree and delete all leaf nodes in the subtree.

Let's do a DFS traversal of the tree (Euler tour) and record in and out times for each node. We will refer to them as L_i (in) and R_i (out). Subtree of node v forms a segment $[L_v, R_v]$. Thus, maintaining a set of leaf nodes and a set of nodes with at least two children is still possible. You just have to keep these nodes ordered by their L_v value. Retrieving all leaf nodes in the subtree will just be equivalent to visiting all set values in segment $[L_v + 1, R_v]$. Note that we use $L_v + 1$ to skip visiting node v itself. Of course, this works for the other set as well.

The remaining part is the hardest: suppose we are at a node u which has children $c_1, \ldots, c_k$. For each child c_i, how to retrieve a node that is currently located there? If we can efficiently answer this question, the whole problem is solved.

For each node v, let's keep a value a_v — its current height. We will keep nodes ordered by their L_v value. Thus, "lifting" all nodes in the subtree will be equivalent to the following:

- For each node x in $[L_v + 1, R_v]$, if $a_x > h_v$ then decrease a_x by 1.

We can technically support this operation with a segment tree. In each segment tree node, store the minimum and maximum on the range. Also, some nodes can get deleted if they get merged. Make sure not to include them in your min/max calculations. When traversing the segment tree, you have the following cases to handle:

- If the segment has no active nodes, just stop.

- If maximum value on the segment is not more than h_v, there is nothing to update, just stop.
- If minimum value on the segment is more than h_v, we can update the whole segment (with pushes).
- Otherwise, just keep descending further.

This will not work at first. There can be too many segments to update. How can we prevent that? Let's introduce one rule: When merging two nodes u and v, keep the one with a smaller L_u value and delete the other one.

We claim that if we follow this rule, all nodes that "escape" the subtree of v will form a prefix of its segment (if we ignore deleted nodes). We can prove it with an induction. Obviously it holds for leaf nodes. Assume it is true for all children of v. Does it hold for v as well?

Before leaving the subtree of v, a node has to pass through v. If there is a different child of v that has been visited earlier by the DFS (Euler tour), then the current node will definitely be deleted. Otherwise, the whole prefix before it is already deleted or already "above" v. Letting this node escape does not break any conditions.

So our segment tree updates are actually $O(\log n)$ in complexity. There is only one part that remains unanswered: how to find which node is currently located at vertex v?

Well, since we proved that all nodes which escape the subtree form a prefix of its segment, we can just binary search the node we need. If the candidate is some position m in segment $[L_v + 1, R_v]$, you can check the maximum value on segment $[L_v + 1, m]$ and verify that it is more than h_v. This is $O(\log^2 n)$ in complexity.

We can do better. You can traverse the segment tree and split $[L_v + 1, R_v]$ into $O(\log n)$ segments. For each segment, we already know its maximum. Thus, you can find the exact segment where the desired node is. Since that segment will just be a single node in the segment tree, you can just keep descending left or right from there instead of binary searching.

Final time complexity is $O((n + q) \log n)$.

Extra comments

There exists a completely different solution which involves compressing vertical chains. You need to learn to merge two vertical chains and delete arbitrary numbers inside chains. When using an efficient data structure like a treap, this solution is also $O((n + q) \log n)$ in theory. Although, it has a lot of cases and is definitely harder implementation-wise.