Solution Sketch for Wall

Overview

We'll simplify the notion of the problem as follow:

Initially we have an array of length n where the value at each index is 0, and we are to process k queries in order. We will denote the value at index x as $A[x]$. There are two kinds of operations:

- Minimize(l,r,t): For all indices x between $[l, r]$, the value become $min(A[x], t)$
- Maximize(l,r,t): For all indices x between $[l, r]$, the value become $max(A[x], t)$

The `minimize` operation corresponds to the original `remove` operation;

the `maximize` operation corresponds to the original `add` operation.

Trivial solution - $O(nk)$

A trivial solution to this problem is to use an array of length n to maintain the current height at each index, and for each query perform a linear update in $O(n)$. This solution runs in $O(nk)$.

Segment Tree - $O((n + k) \lg n)$

One crucial observation is that for a specific position, any cascade of operations applied on it could be simplified to one `minimize` operation and one `maximize` operation.

For example, applying (here we omit the paramter `l`, `r` as we consider operations solely on a specific index):

- minimize(9), minimize(8), minimize(7) $\Rightarrow$ minimize(7), maximize($-\inf$)
- minimize(3), maximize(7), minimize(4) $\Rightarrow$ minimize(4), maximize(4)

In general, this observation could be verified by simple induction.

With this observation we could now improve the update time. We instead use a segment tree to maintain *the final operations applied on an interval.*

A segment tree is basically a binary tree, where each node contains the following information:

```
class Node {
    Node *left, *right;  // children nodes
    int from, to;        // corresponding interval [from, to]
    int opmin, opmax;    // min/max operation applied on interval
};
```

An minimize operation would look something like:

```
void Node::minimize(int l, int r, int t) {
    if( from==l && to==r ) { // applied on full segment
        opmin = min(opmin, t);
        opmax = min(opmin, opmax);
    } else {
        // pass down previously applied operation
        left->minimize( left->from, left->to, opmin );
        left->maximize( left->from, left->to, opmax );
        right->minimize( right->from, right->to, opmin );
        right->maximize( right->from, right->to, opmax );
        // recursively apply current minimize operation
        if( r<=left->to ) left->minimize( l, r, t );
        else if( l>=right->from ) right->minimize( l, r, t );
        else {
            left->minimize( l, left->to, t );
            right->minimize( right->from, r, t );
        }
    }
}
```

A single round of minimize operation runs in $O(\lg n)$, likewise for maximize operations. Note that in the snippet above, we use the well-known lazy-update technique to maintain the operations applied at one position. i.e. the message is passed down to children nodes *only when necessary*, that is, when new operations are imposed on only part of the segment, so previous operations should be passed down first.

After all queries are processed, we simply scan each position and check (with $\lg n$ complexity) the operations applied on it, obtaining the final value at the position.

The overall running time is therefore $O((n+k)\lg n)$.

It is possible to optimize further so that the overall running time become $O(n + k\lg k)$, it is however entirely not necessary for this problem.