6.851 Advanced Data Structures (Spring'12)

Prof. Erik Demaine TAs: Tom Morgan, Justin Zhang

Problem 1 *Sample solution*

Creationist successor data structure. The data structure is a balanced BST, such as an AVL tree or red black tree, augmented with the following fields augmented to each node x:

- t_d : The time at which x is deleted
- t_m : The maximal deletion time among x's subtree

Operations. The three operations are done as following:

- `Insert`$(-\infty,$ "`insert(k)`"): The key k is inserted into the BST, with its deletion time set to ∞. We also set all its ancestors' maximal deletion time to ∞. Rotations are done to balance the tree. During rotation, the property that each note's t_m stores its subtree's maximal deletion time is preserved. This is easy because we only need to update the maximal deletion times of the nodes being rotated. The insertion, the update for ancestors, and the rotations each takes $O(\log n)$ time, where n is the number of elements in tree.
- `Insert`(`t`, "`delete(k)`"): The key k is deleted at time t. Locate k in the BST. Its deletion time t_d should be greater than t. Change its deletion time t_d to t, and walk back to the root, updating the maximal deletion time of all nodes on path. Both the locating and updating takes $O(\log n)$ time.
- `Delete`$(-\infty,$ "`insert(k)`"): Delete the key k from the BST. We then update all of the maximal deletion times of the ancestors of k. Both the deletion and the update take $O(\log n)$ time.
- `Delete`(`t`, "`delete(k)`"): Locate k in the BST. Increase its deletion time to ∞, and set it and all of its ancestors' maximal deletion times to ∞. Both the locating and updating take $O(\log n)$ time.
- `Query`(`t`, "`successor(k)`"): We will essentially perform the standard successor search, except that we will use the t_m's to only walk over the elements of the tree where the keys have not been deleted. Specifically, we will start by finding key k's successor k' in the tree. If k''s deletion time is greater than t, then return k', otherwise do the following. Walk up the tree, until the first right subtree whose maximal deletion time is greater than t. Walk down that subtree, avoiding all subtrees whose maximal deletion time is smaller or equal than t, and find the smallest element (at time t). This is k's successor at t. The walking up and down takes $O(\log n)$ time.

Analysis of operation time. This data structure has $O(\log n)$ for all operations, where n is the number of elements in tree. We have $n \leq m$, where m is the total number of updates. Therefore the data structure has operation time $O(\log m)$ (for all operations.)