COMP3506/7505: Regular Exercise Set 9

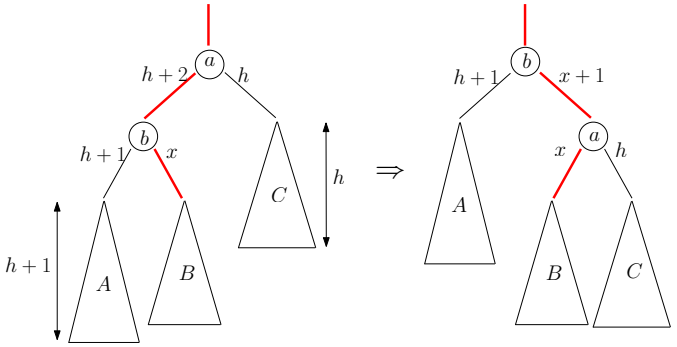
Prepared by Yufei Tao

Problem 1*. Prove that an insertion into the AVL-tree can trigger at most one (single/double) rotation.

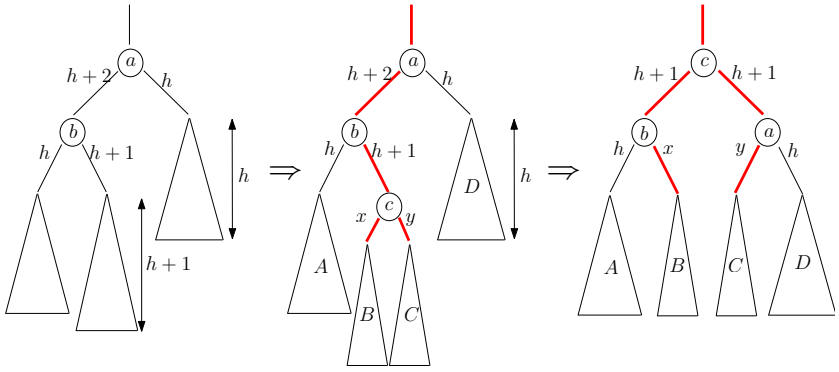
Solution. Recall that each insertion descends a single root-to-leaf path. Let Π be the insertion path (the leaf on Π stores the newly inserted element). Let u be the lowest node on Π that is now imbalanced. Let p be any proper ancestor of u on Π that has become imbalanced. We will show that, fixing the imbalance at u automatically restores the balance at p.

Without loss of generality, suppose that u is in the right subtree of p. Since p is now imbalanced, currently its right subtree height must be 2 plus its left subtree height. We will prove that after fixing the imbalance at u, the right subtree height of u decreases by 1. This, in turn, decreases the right subtree height of p by 1, thus restoring the balance at p.

Recall that single and double rotations are performed as follows:



(a) Single rotation (remedying left-left)



(b) Double rotation (remedying left-right)

Figure 1: Rotations

Let us first look at single rotation. Observe that the value of x *must* be h in an insertion (otherwise, a was imbalanced even before the insertion, which is impossible). This means that the height of the subtree rooted at node a decreases from $h + 2$ to $h + 1$ (note that after the rotation the subtree is rooted at node b), as claimed.

The claim is obvious in double rotation, as shown in the figure.

Problem 2.** Prove that it suffices to handle only 2-level imbalance in the insertion and deletion algorithms of the AVL-tree. In other words, neither algorithm will run into a situation where an imbalanced node sees an absolute difference of 3 or higher in the heights of its left and right subtrees.

Solution. Our solution to Problem 1 essentially serves as the proof here for the insertion algorithm.

Let us now look at the case of deletion. Let Π be the root-to-leaf path followed by the deletion algorithm. The algorithm removes the leaf node of Π, and then fixes the imbalanced nodes of Π in bottom-up order. Next, we will prove the following claims:

1. Before fixing an imbalanced node u, all other nodes on Π are balanced.
2. u has a 2-level imbalance situation.

This will complete our proof for the deletion case as well.

We will prove Claims 1 and 2 by induction, in bottom-up order of the nodes fixed:

Base Case. Consider the moment before fixing the first imbalanced node u of Π. It means that u must have lost a level in one of its subtrees in the deletion. Since u was balanced before the deletion, Claim 2 holds on u. Furthermore, the height of the subtree rooted at u has *not* changed (because it is decided by the subtree of u that did not lose any leaf). This indicates that, for any proper ancestor v of u, both subtrees of v must have the same height as before the deletion; hence, v remains balanced.

Inductive Case. Suppose that, after fixing an imbalanced node u of Π, v is the next imbalanced node to fix. By induction, we know that v was balanced before the fixing of u, which had a 2-level imbalance situation. Remedying the situation was done with a single or double rotation. But either rotation type can reduce the height of one subtree of v by at most 1—let it be the right subtree of v, without loss of generality. Since v is now imbalanced, the left subtree of v must have 2 more levels than its right subtree. This proves Claim 2. Furthermore, the height of the subtree rooted at v has not been affected by the fixing of u (because it is determined by the left subtree). This implies that all the proper ancestors of v must remain balanced. This proves Claim 1.

Problem 3. Let T be a balanced binary tree of n nodes. For each node u of T, define its *count* as the number of nodes in its subtree (remember that the subtree includes the node itself). Modify the insertion and deletion algorithms to maintain the counts of all the nodes. Your algorithms must still perform an insertion and deletion in $O(\log n)$ time.

Solution. Notice that the count of a node u can be obtained from those of its child nodes in constant time. We can utilize this fact to update the counts in a bottom-up manner along the insertion/deletion path. Next, we elaborate the details for insertion, because the same ideas apply to deletion as well.

First, insert a new leaf in T as described in the lecture. Set the count of the leaf to 1. Let Π be the insertion path. Set b to this leaf. Next, repeat the following steps.

1. If b is the root of T, finish.
2. Let a be the parent of u. Update the count of a from its child nodes in constant time.
3. If a is still balanced, set b to a, and repeat from Step 1.

4. Otherwise, perform a single or double rotation, and update the counts of at most 3 nodes accordingly:

   - Single rotation: See Figure 1a. Update first the count of a (from its children), and then the count of b. Now repeat from Step 1.
   - Double rotation: See Figure 1b. Update first the counts of a, b (from their children), and then the count of c. Now set b to c, and repeat from Step 1.

Clearly, we spend constant time per level. The total cost of an insertion is therefore $O(\log n)$.

Problem 4. In this exercise, we will design an algorithm to detect whether network packets have been sent out in a wrong order. A network packet here is defined as a pair (t, k) where t is the timestamp when the packet was sent out, and k is an integer representing the packet's contents. These packets arrive in an arbitrary order (the order is not necessarily in ascending t, because the packets may have been transmitted at different speeds). Design an algorithm to detect whether you have received any two pairs (t_1, k_1) and (t_2, k_2) such that $t_1 < t_2$ but $k_1 > k_2$. You may assume that all the packets have distinct t-values and distinct k-values. Your algorithm must process every incoming packet in $O(\log n)$ time, where n is the number of packets received.

Solution. Observe that the answer is *no* if and only if the following is true: when the packets are sorted in ascending order of t, they are also sorted in ascending order of k. Utilizing this observation, we simply maintain an AVL-tree on the t-values of the packets, and store the k-value of each packet in the same node where the packet's t-value is the key. Then, we process an incoming packet (t, k) as follows:

1. Find the packet (t_1, k_1) where t_1 is the predecessor of t, among the t-values of the received packets. Report "yes" if $k < k_1$.
2. Find the packet (t_2, k_2) where t_2 is the successor of t, among the t-values of the received packets. Report "yes" if $k > k_2$.
3. Insert (t, k) into the AVL-tree according to t.

The processing time is clearly $O(\log n)$ per packet.

Problem 5.** In two-dimensional space, a point (x, y) *dominates* another point (x', y') if $x > x'$ and $y > y'$. Let S be a set of n points in two-dimensional space, such that no two points share the same x- or y-coordinate. A point $p \in S$ is a *maximal point* of S if no point in S dominates p. For example, suppose that $S = \{(1, 1), (5, 2), (3, 5)\}$; then S has two maximal points: $(5, 2)$ and $(3, 5)$.

Describe a data structure to support the following operations on a dynamic set S:

- INSERT(p): Adds a new point p to S.
- QUERY: Reports all the maximal points of S.

If n is the current size of S, your structure must support an insertion in $O(\log n)$ amortized time, and a query in $O(1 + k)$ time, where k is the number of maximal points.

Solution. Let P be the set of maximal points of S. Observe that if we sort the points of P in ascending order of x-coordinate, then they are also sorted in *descending* order of y-coordinate. Based on this observation, we maintain an AVL-tree T on the x-coordinates of the points in P.

Given an incoming point $p = (x, y)$, we process it as follows.

1. Find $p' = (x', y')$, where x' is the successor of x, among the x-coordinates of all the points in P.

2. If p' exists and dominates p, discard p, and finish.

3. Find $p'' = (x'', y')$, where x'' is the predecessor of x, among the x-coordinates of all the points in P. If p'' does not exist, set p'' to nil.

4. Do one of the following:

   4.1 If p'' is nil, finish.

   4.2 If p does not dominate p'', finish.

   4.3 Otherwise, delete p'' from T, and repeat from Step 3.

Every step can be performed in $O(\log n)$ time except Step 4.3. However, since each point can be deleted only once, the total amount of time spent on Step 4.3 is bounded by $O(n \log n)$. Therefore, our algorithm supports each insertion in $O(\log n)$ amortized time.

To answer a query, simply output all the k points in T using $O(1 + k)$ time.