CIS 1210—Data Structures and Algorithms—Spring 2025

AVL Trees—Tuesday, April 22 / Wednesday, April 23

Readings

- Lecture Notes Chapter 25: AVL Trees

Review: AVL Trees

Binary Search Trees
You may remember from CIS 1200 that Binary Search Trees are constructed with the following invariant:

Let x be a node in a BST. If y is a node in the left subtree of x, then $y.key \leq x.key$. If y is a node in the right subtree of x, then $y.key \geq x.key$.

While this invariant alone makes it simple to search for values by traversing down the tree and making comparisons, it provides no guarantees in tree height. That is, it is possible for a BST to have height $O(n)$. With some clever modifications to our methods, we can do better than this!

We will pursue a smaller bound on tree height—specifically, we want to bound our height to $O(\lg n)$. We will do so by adding the following tree invariant:

Height Balance Property: For every internal node $v \in T$, the heights of the children of v differ by at most 1.

This property provably reduces our height bound to $O(\lg n)$, and you can see the lecture notes for more details on why this is true. Any BST that satisfies the height-balance property is said to be an AVL Tree.

Operations

To maintain these invariants, we can run search, insert, and delete similarly to the simple BST implementation. However, after insertion and deletion, we must check to ensure that our new invariant is maintained and rebalance the tree accordingly. To do this, we traverse up the tree from the site of insertion/deletion and call the trinode restructuring algorithm, detailed below:

Trinode Restructuring Algorithm

1. Let z be the first node that breaks our child height-balance property. Let z's child with larger height be called y. Then, let y's child with larger height be called x (in deletion, if y's children are both the same height, we set x to be y's child on the same side as y is).
2. Let a, b, c be an numerically-ordered listing of x, y, and z. That is, $a \leq b \leq c$.
3. Between x, y, and z, there are 4 child subtrees. Let these trees be T_0, T_1, T_2, T_3 in left-to-right order. Note that all elements in T_0 are less than all elements in T_1, and all of those elements are less than those in T_2, et cetera.

4. Replace the subtree rooted at z with a new subtree rooted at b.
5. Set b's right child to c.
6. Set b's left child to a.
7. Set a's left and right children to be T_0 and T_1, respectively.
8. Set c's left and right children to be T_2 and T_3, respectively.

This algorithm selects an unbalanced node and relevant children of that subtree and re-orders them such that the median value of the three selected nodes becomes the parent. This operation corrects the invalid BST to meet the height-balance invariant. The subtrees of these three nodes are also kept in increasing order, which maintains the BST invariant.

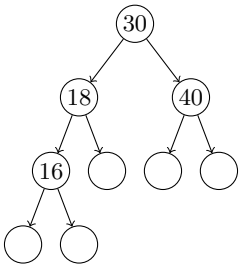
The insertion and deletion methods utilize the tri-node restructuring algorithm upwards through the height of the tree. Since we run insert/delete on a previously balanced binary tree, we know the height of the tree is $\lg n$.

We can see that a single Tri-node restructure involves updating the children of at most 3 nodes. Thus a single restructure will run in constant time. In the worst case we need to restructure the entire path that we traced to insert a node, or delete a node which is at most the height of the tree. Thus we call tri-node restrucure at most $O(\lg n)$ times yielding a $O(\lg n)$ bound on all AVL tree operations.

Problems

Problem 1

Consider the following tree:



Justify why it's an AVL tree, and then perform the following operations:

Insert 14, Insert 26, Delete 14, Delete 16

Be sure to show the resultant tree at each step.

Problem 2: True or False

1. Insertion causes at most one rotation (single or double) while deletion can cause more than one.
2. In an AVL tree, the median of all elements in the tree is always at the root or one of its children.

Problem 3:

You are given a set of n batteries that can each be one of k distinct capacities. While we cannot directly access each battery's capacity, we can take two batteries and determine which one is greater or if they're equal in $O(1)$ time. Given the set of n batteries, design an algorithm in $O(n \lg k)$ time to return the size of the largest subset of equal capacities.