Solution Set

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 a 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.

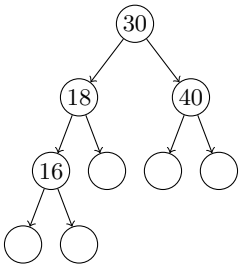
The insertion and deletion methods utilize the tri-node restructuring algorithm upwards through the height of the tree. Since we run insert/delete on a previously balanced binary tree, we know the height of the tree is $\lg n$.

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.

Solution

We'll first explain why the tree is an AVL tree. Recall that a tree T is an AVL tree precisely when T is a binary search tree that satisfies the height-balance property. We'll show our tree is a height-balanced BST.

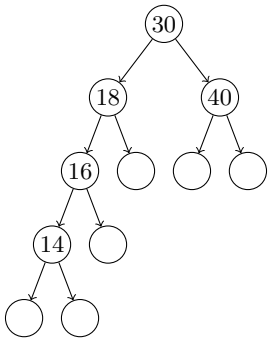
The given tree satisfies the binary search tree property, as for every node, we can see that all its left subtree children are less or equal to it and its right subtree children are greater than or equal to it. It follows the tree is a BST.

We're left to show the tree satisfies the height-balance property i.e. for every internal node, the heights of its children differ by at most 1. Let's consider every internal node. The 16 node has height 1, since its children are both empty (height 0). By similar logic, the 40 node has height 1. The 18 node has height 2, as one child is empty (height 0) and one child is the 16 node (height 1). Lastly, the 30 node has height 3, since one child is node 18 (height 2) and one child is node 40 (height 1). We've inspected all internal nodes and found that the heights of their children differ by at most 1, so our tree satisfies the height-balance property.

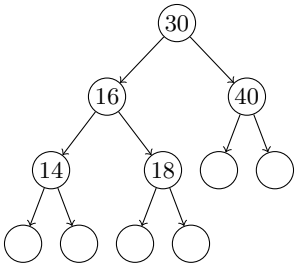
We'll now perform the insertions and deletion on our AVL tree:

Insert 14:

First, insert 14 into the tree as you would in CIS1200 by tracing through the tree, which yields:

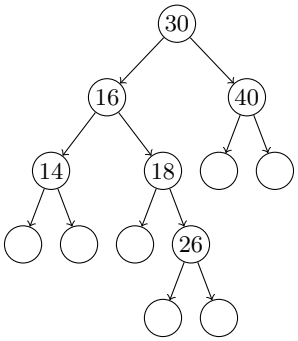


Observe that the tree is now unbalanced, as node 18's left child has height 2 while its right child has height 0, so we need to do trinode restructuring. Using the strategy outlined above, our z is 18, our y is 16, and our x is 14. It follows a is 14, b is 16, and c is 18. Following the strategy still, we'll replace z (18) with b (16) and make this node's left child a (14) its right child c (18). Lastly, note T_0, T_1, T_2, T_3 are all empty trees, so we set 14's and 18's new children to be empty nodes. This restructuring yields an AVL tree:



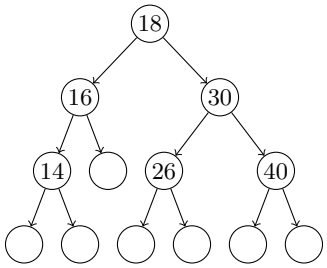
Insert 26:

As with 14's insertion, first insert 26 as if this tree were a normal BST, which yields:



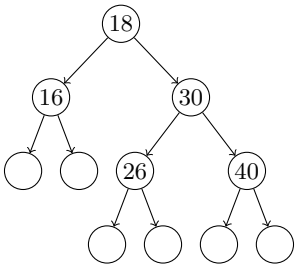
Observe that the tree is now unbalanced, as node 30's right child has height 1 while its left child has height 3, so we need to do trinode restructuring. Using the strategy outlined above, our z is 30, our y is 16, and

our x is 18. It follows a is 16, b is 18, and c is 30. Following the strategy still, we'll replace z (30) with b (18) and make this node's left child a (16) and its right child c (30). Lastly, note T_0 the subtree rooted at 14, T_1 is an empty tree, T_2 is the subtree rooted at 26, and T_3 is the subtree rooted at 40. We lastly fill a and c's children left-to-right with these subtrees. This restructuring yields an AVL tree:



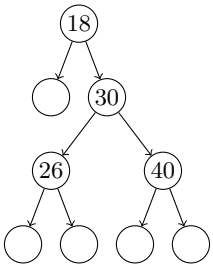
Delete 14:

Remove 14 using standard BST deletion. Below shows the post-deletion tree. Observe it's still balanced, so no restructuring is needed.

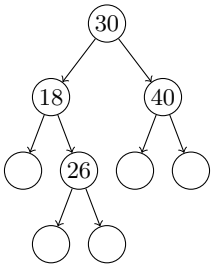


Delete 16:

Remove 16 using standard BST deletion. Below shows the post-deletion tree:



Observe that the tree is now unbalanced, as node 18's right child has height 2 while its left child has height 0, so we need to do trinode restructuring. Using the deletion rebalancing strategy from lecture, our z is 18, our y is 30, and our x is 40. Do a similar process to above to ultimately yield:



Observe it's height-balanced and thus an AVL tree again. We only need one rotation in this case, but some deletions may require more.

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.

Solution

1. **True.** For insertion, the rotation solves the local imbalance (among x, y, z) and the global one by having b take the place of z. This restores the height of the highest ancestor (was z but now b) to be its value before the rotation thus restoring balance. For any ancestor of b, it's height after the insertion increased by at most one, and then after the rotation, since b gets resolved back to z's original height (as b took z's spot), this also resolves the ancestor's height back to its original, which we know by the AVL invariant was balanced. We can repeat this argument up the tree for any ancestor which increased in height because of the insertion.

   In contrast, deletion can reduce the size of the subtree rooted at b by 1 and cause an ancestor of b to become unbalanced while that node's height is the same. After the rotation, b's height can be reduced by 1 compared to z's original height (as b took z's spot), since we reduce the height of the subtree with the deletion. This can cause an imbalance in b's ancestor, requiring another rotation. This can repeat for multiple rotations as we propagate back up the tree at most $O(\lg n)$ times.

   Proof:
   For any node a:

   - a_0- height before insertion
   - a_1- height after insertion but before rotation
   - a_2- height after rotation

   Insertion:
   z is the first node as we go up the tree that is imbalanced. For z to become imbalanced, it must be due to the insertion which happens in y's subtree. Thus, we know that $y_1 = y_0 + 1$. y's sibling s, must now have height 2 less than y_1 to define an imbalance, so $s_1 = y_1 - 2$. Also, $x_1 = x_0 + 1 = y_1 - 1 = y_0$. Useful results:

$$\begin{aligned} z_1 = z_0 + 1 &= y_1 + 1 \\ y_1 = y_0 + 1 \\ x_1 = y_1 - 1 \\ s_1 = y_1 - 2 \end{aligned}$$

   Case 1: Single rotation. In this case, we swap y and z. We can define y's new height: $y_2 = max(x_2, z_2) + 1$. z_2 is the max of its subtrees' heights plus 1. It's one subtree is s which hasn't changed heights: $s_2 = s_1 = y_1 - 2$. The other subtree is x's sibling before the rotation. Because the insertion of w in x's subtree caused y to increase in height, the height of this sibling is at most $x_0 = x_1 - 1 = y_1 - 2$. Thus, $z_2 = (y_1 - 2) + 1 = y_1 - 1$.

   x's subtree is unchanged by rotation, so $x_2 = x_1 = y_1 - 1$. Thus, $y_2 = (y_1 - 1) + 1 = y_1 = z_0$. Therefore, y, which has taken z's spot, has the same exact height after the insertion and rotation that z had before, thus all of y's ancestors must have their original heights as before the insertion, and so no new imbalances have been created.

   Case 2: Double rotation. In this case, we swap x and z. We can define x's new height: $x_2 = max(y_2, z_2) + 1$. z_2 is the max of its subtrees' heights plus 1. It's one subtree is s which hasn't changed heights: $s_2 = s_1 = y_1 - 2$. The other subtree is one of x's children u. $u_0 \leq x_0 - 1 = y_1 - 2$. Thus,

$z_2 = (y_1 - 2) + 1 = y_1 - 1$.

y_2 is the max of its subtrees' heights plus 1. One of them was already y's subtree, so it has at most height of $y_0 - 1 = y_1 - 2$. The other is one of x's children v. $v_0 \leq x_0 - 1 = y_1 - 2$. Thus, $y_2 = (y_1 - 2) + 1 = y_1 - 1$.

Thus, $x_2 = (y_1 - 1) + 1 = z_0$. Therefore, x, which has taken z's spot, has the same exact height after the insertion and rotation that z had before, thus all of x's ancestors must have their original heights as before the insertion, and so no new imbalances have been created.

Conclusion: In both cases, no imbalance could exist after one rotation, so we always only need 1 rotation for each call to insertion. Thus, insertion takes $O(lgn)$ runtime but only $O(1)$ rotations.

Deletion:
z is the first node as we go up the tree that is imbalanced. For z to become imbalanced, it must be due to the deletion which happens in y's sibling's subtree. This is because y is by definition the subtree of z after deletion with greater height. Thus, we know $s_1 = s_0 - 1$ and $s_1 = y_1 - 2$ since before the deletion, s_0 was within 1 of y_0. $y_1 = y_0$ and $x_1 = x_0$ since the deletion occurred in s's subtree. Lastly, $z_1 = z_0 = y_0 + 1 = y_1 + 1$ since $y_0 = s_0 + 1$ and $z_0 = max(y_0, s_0) + 1$.

Case 1: Single rotation. In this case, we swap y and z. We can define y's new height: $y_2 = max(x_2, z_2) + 1$. z_2 is the max of its subtrees' heights plus 1. It's one subtree is s which hasn't changed heights: $s_2 = s_1 = y_1 - 2$. The other subtree is x's sibling before the rotation. The height of this sibling is at most $y_1 - 1$ since x has greater or equal height by definition and has height at most $y_1 - 1$. In the worst case, x has strictly greater height, so this sibling has height $y_2 - 2$. Thus, $z_2 \leq (y_1 - 2) + 1 = y_1 - 1$

x's subtree is unchanged by rotation, so $x_2 = x_1 = y_1 - 1$. Thus, $y_2 = (y_1 - 1) = y_1 = z_0 - 1$. Therefore, y, which has taken z's spot, has 1 minus the height after the deletion and rotation that z had before, thus y might now have height which is 2 less than its sibling, which means that another imbalance can exist.

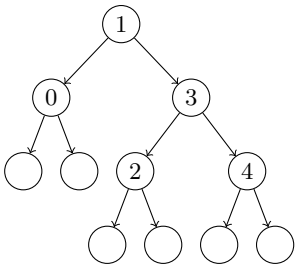
Case 2: Double rotation. In this case, we swap x and z. We can define x's new height: $x_2 = max(y_2, z_2) + 1$. z_2 is the max of its subtrees' heights plus 1. It's one subtree is s which hasn't changed heights: $s_2 = s_1 = y_1 - 2$. The other subtree is one of x's children u. $u_0 \leq x_0 - 1 = y_1 - 2$. Thus, $z_2 = (y_1 - 2) + 1 = y_1 - 1$.

y_2 is the max of its subtrees' heights plus 1. One of them was x's sibling which we showed has height of $y_1 - 2$ or $y_1 - 1$, but in the worst case it's $y_1 - 2$. The other is one of x's children v. $v_0 \leq x_0 - 1 = y_1 - 2$. Thus, $y_2 = (y_1 - 2) + 1 = y_1 - 1$.

Thus, $x_2 = (y_1 - 1) + 1 = z_0 - 1$. Therefore, x, which has taken z's spot, has 1 minus the height after the deletion and rotation that z had before, thus x might now have height which is 2 less than its sibling, which means that another imbalance can exist.

Conclusion: In both cases, an imbalance could exist after one rotation, so in the worst case, an imbalance exists after every rotation, which continues up the tree until the root is balanced. Thus, deletion takes $O(lgn)$ rotations worst case.

2. **False.** This isn't always the case. If the left and right subtree have equal size, then the median would be at the root. If the right subtree is larger than the left subtree, then the median will be in the right subtree and vice versa. In the following counterexample, the right subtree has exactly one more node, and so the median is at the leftmost node in the right subtree.



Observe that this is a valid AVL by the height balance property. The median of this set of numbers is 2, yet it is not the root nor a child of the root.

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.

Solution

Algorithm: Create an empty AVL tree. Each node in the AVL tree will point to a battery with a distinct value. Each node will also have a field *count*.

For a given battery b, we can insert it by comparing it to the battery at the root node, which we can call r. If b has more/less capacity than r, recursively search the right/left subtree; otherwise b and r have equal capacity, so increment the *count* field in r. If the root node is not defined (that is a battery with capacity k_b has not been seen before), insert a new node there which points to the battery b and with $count = 1$. Re-balance the tree as necessary.

Repeatedly insert all n batteries into the AVL tree. After this, run BFS or another graph traversal algorithm to find the max count value in the tree, and return it as the size of the largest subset of equal capacities.

Proof of Correctness: Our insertion maintains the binary search tree property at each iteration, where batteries with greater capacity than the root node are on the right subtree, while capacities less than the root not are in the left subtree. We maintain the *count* field in each node at every insertion by incrementing the *count* when we see a capacity again or by inserting a new node is made with $count = 1$, whenever we see a capacity for the first time. Thus, our final search correctly finds the max *count* as the size of the largest subset of equal capacities.

Runtime Analysis: We only insert a node when it's capacity doesn't already exist in the tree, so at any point, it has at most k unique nodes. We maintain the AVL tree properties which makes our insertion take $O(\lg k)$ time. We perform n insertions for $O(n \lg k)$ runtime. Our final traversal takes $O(k)$ time while performing constant time operations to keep track of the max.