CS 2112 : Object-Oriented Design and Data Structures – Honors
Fall 2014

Balanced binary trees

We've already seen that by imposing the **binary search tree invariant** (BST invariant), we can search for keys in a tree of height h in O(h) time, assuming that the keys are part of a total order that permits pairwise ordering tests. However, nothing thus far ensured that h is not linear in the number of nodes n in the tree, whereas we would like to know that trees are **balanced**: that their height h, and therefore their worst-case search time, is logarithmic in the number of nodes in the tree.

AVL Trees

AVL trees, named after its inventors, Adelson-Velsky and Landis, were the original balanced binary search trees. They strengthen the usual BST invariant with an additional invariant regarding the heights of subtrees. The **AVL invariant** is that at each node, the heights of the left and right subtrees differ by at most one.

The **balance factor** of a tree node is defined as the difference between the height of the left and right subtrees. Letting h(t) be the height of the subtree rooted at node t, where an empty tree is considered to have height −1, the balance factor BF(t) is:

$$BF(t) = h(t.left) - h(t.right)$$

The AVL invariant at node t is, then, that $|BF(t)| \leq 1$ and also that the invariant holds on both the left and right subtrees if they are nonempty.

Is an AVL tree balanced?

A balanced tree has the property that the height h is O(lg n); that is, that $h \leq k \lg n$ for some k. To show this, we need to show that a tree of height h contains a number of nodes that grows geometrically as h increases.

Let us determine the minimum number of nodes that can exist in an AVL tree of height h. Call it N(h). The root node at height h has two subtrees, one of which must be height h-1. By the AVL invariant, the other subtree must be at least height h-1. Therefore, the least number of nodes that can be in the tree is N(h-1)+N(h-2)+1: that is, the number of nodes in the two subtrees, plus the root node itself. Given this **recurrence**,

$$N(h) = 1 + N(h-1) + N(h-2)$$

and the fact that N(-1) = 0 and N(0) = 1, we can derive the minimum number of nodes for the first few heights:

```
N(-1) = 0
N(0)  = 1
N(1)  = 2
N(2)  = 4
N(3)  = 7
```

```
N(4)  = 12
N(5)  = 20
     ...
```

It's not obvious that this function is growing exponentially. However, you may already have noticed that the recurrence above is very similar to the Fibonacci recurrence:

$$F_n = F_{n-1} + F_{n-2}$$

In fact, if we add 1 to each term in the sequence of values for N(h), the familiar Fibonacci sequence emerges: 1, 2, 3, 5, 8, 13, ... In general, $N(h) = F_{h+3} - 1$. Asympotically, the –1 term doesn't matter, so N(h) grows asymptotically in the same way as the Fibonacci sequence.

If we can show that the Fibonacci sequence grows exponentially, we'll know N(h) does too. In fact, it does. The exact formula for the Fibonacci numbers is:

$$F_n = (\varphi^n - \bar{\varphi}^n)/\sqrt{5}$$

Here, φ is the **golden ratio** $(1+\sqrt{5})/2$ (≈1.618) and $\bar{\varphi}$ is its negative reciprocal $(1-\sqrt{5})/2$ (≈−0.618). The golden ratio and its negative reciprocal both have the interesting property that $\varphi^2 = \varphi+1$ (and $\bar{\varphi}^2 = \bar{\varphi}+1$). Multiplying both sides of the equation by φ^{n-2}, we can conclude that for *any* exponent n, we have $\varphi^n = \varphi^{n-1} + \varphi^{n-2}$, and similarly for $\bar{\varphi}$.

To show that the exact formula for the Fibonacci numbers is correct, first observe that the formula for F_n works for n=0 if we consider $F_0 = 0$, and it also works for n=1 since $\varphi - \bar{\varphi} = \sqrt{5}$.

We can show using induction that the formula above works for all greater values of n too. Assume that the formula above works for the entire sequence up to but not including F_n. Then,

$$\begin{aligned} F_n = F_{n-1} + F_{n-2} &= (\varphi^{n-1} - \bar{\varphi}^{n-1})/\sqrt{5} + (\varphi^{n-2} - \bar{\varphi}^{n-2})/\sqrt{5} \\ &= (\varphi^{n-1} + \varphi^{n-2} - \bar{\varphi}^{n-1} - \bar{\varphi}^{n-2})/\sqrt{5} \\ &= (\varphi^n - \bar{\varphi}^n)/\sqrt{5} \end{aligned}$$

Since we know the formula works for 0 and 1, it must work for n=2, and by induction, all greater values of n as well.

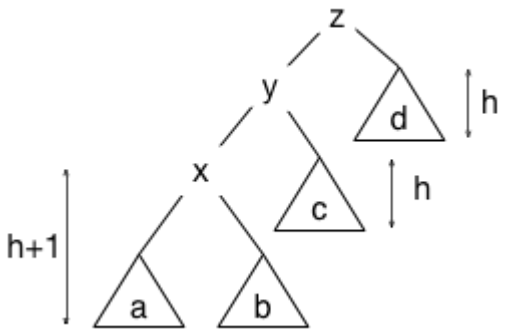
Now, we can apply this formula to show that AVL trees are balanced. Observe that since $|\bar{\varphi}| < 1$, the term $\bar{\varphi}^n$ becomes vanishingly small for large n. Asymptotically, the Fibonacci numbers grow as φ^n (technically, they are $\Theta(\varphi^n)$, meaning that φ^n is both an asymptotic upper *and lower* bound on F_n).

In fact, for all h≥0, $N(h) \geq \varphi^h$. Given an arbitrary AVL tree of height h containing n nodes, we know $n \geq N(h) \geq \varphi^h$, so $h \leq \log_\varphi n$. Since all logarithms are related by constant factors, h is therefore O(lg n). AVL trees are balanced.

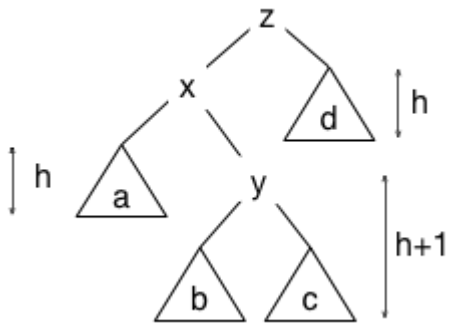
Inserting into an AVL tree

To insert a new element (key/value pair) into an AVL tree, we start by using the key in the usual way to find where the key can be inserted as a leaf while preserving the BST invariant. Adding a new leaf makes the path to that leaf one longer than previously, so the AVL invariant may now be broken. To fix the invariant, we find the lowest node along that path to the leaf where the invariant is broken, and apply one or two **tree rotations**. Assuming that the insertion is done recursively, it is easy to identify where along the path the invariant is broken as the recursive calls return, assuming that each node keeps track of its height in the tree. Of course, that also means that nodes' heights must also be updated as the recursion unwinds.

If inserting a new leaf breaks the AVL invariant at some node t, the invariant can only be broken by 1; that is, either BF(t) = 2 or BF(t) = –2, depending on whether the insertion happened under the left child of t or the right child. Without loss of generality, let us consider the left-child case, where BF(t) = 2. Suppose the height of the right subtree is some h and the left subtree has height h+2. Since insertion only affected one path, one of the two subtrees of the left subtree must have height h+1 and the other, h. Depending on which subtree has height h+1, there two cases to consider:



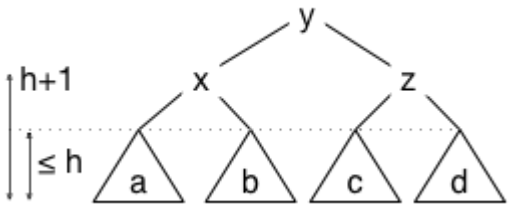
LL case



LR case

In this figure, z is the lowest node that is unbalanced. Therefore, the shorter subtree (c on the left side and a on the right side) must have height h; if their heights were lower, z would not be the lowest unbalanced node.

Now, how to fix the AVL invariant? If we are in the LL case shown on the left, we perform a single tree rotation to make y the parent of z, and update z so that its left child is now the old left child of y (that is, c). The resulting tree looks as follows:



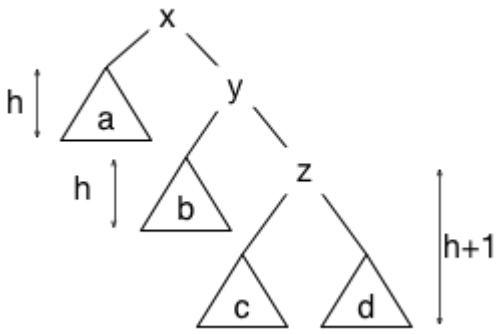
After tree rotation(s)

Does this rotation preserve the BST invariant? Because the BST invariant held before the rotation, we know that a < x < b < y < c < z < d (where a, b, c, d stand for all nodes in subtrees a, b, c, d). This ordering of the keys is preserved in the rotated tree.

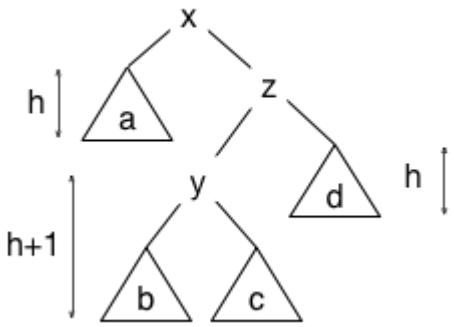
Does this rotation establish the AVL invariant? Since subtrees a, b, c, and d all have at most height h, the nodes x and z are now at height h+1, and node y is at height h+2. The longest path within this part of the tree is now one shorter than before–it's back to h+2—so this change also fixes the AVL invariant for all nodes above y.

To fix the AVL invariant in the LR case, we convert the tree into exactly the same structure as for the LL case. However, this requires more work: all three nodes x, y, and z must be changed. We can think of this as a **double rotation** in which we first rotate x and y, and then rotate y and z. Or we can just update x, y, and z directly.

Symmetrically to the LL and LR cases, there are RR and RL cases. They are handled in exactly the same way, yielding the same resulting tree shown above.



RR case



RL case

An optimization

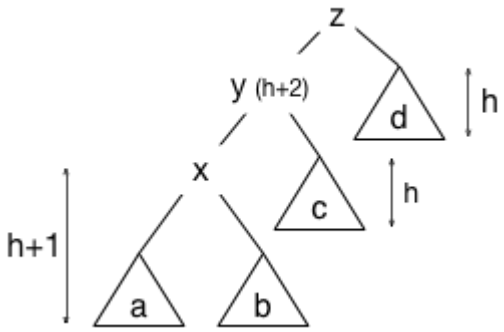
It is not really necessary to store the height of each node at the node. Instead, we can store just the balance factor for the node: –1, 0, or 1. Storing one of three options requires only two bits of space if the programming language is cooperative. If only the balance factor is stored in the node, balance factors only need to be updated when they change. When inserting a node, the only nodes whose balance factors change are those on the path from the leaf up to the first unbalanced node (or the root), and those involved in whatever tree rotations are performed.

Deleting from an AVL tree

Removing a key from the tree can also make it unbalanced. The algorithm works in the usual way for BST deletion, depending on the number of children of node storing the deleted key. Recall that if that node has 0 children, it is pruned; if 1 child, it is spliced out, and if 2 children, its element is replaced with that from the node storing the next (or previous) key in the tree. The node storing that next or previous key is the node that is deleted. In any case, deleting a node (whether the node storing the key or the node storing the next/previous key) may break the AVL invariant along the path to the deleted node. AVL deletion therefore walks back up the tree from the deletion point using tree rotations to restore the AVL invariant.

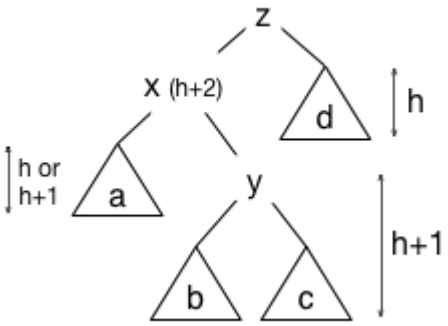
To see how this works, consider the lowest tree node that becomes unbalanced as the result of deleting a node below it. Without loss of generality, let's assume that the deleted node is on the right side of the unbalanced node. Just as for insertion, the cases for deletion on the other side are symmetrical. The right child is at some height h (formerly h+1), and the left side is at height h+2. One of the two grandchildren on the left side must have height h+1. Let us first consider the case

in which the left grandchild has height h+1 but the right grandchild has only height h. The tree then looks as shown in this figure, essentially the same as the LL case above:



To rebalance the tree, we simply use the same single rotation of y and z as in the LL case above, with the y node becoming the new root of the subtree. However, notice that this rotation reduces the height of the subtree from h+3 to h+2. Therefore, it is necessary to continue walking up toward the root, potentially fixing other unbalanced nodes along the way.

We just assumed that the right grandchild had height h. What if the right grandchild has height h+1, instead? The picture then looks like much like the LR case above:



As in the LR case, we use a double rotation of the tree to arrive at the tree rotation shown above. There is one twist, however. Depending on the height of the subtree a, the y node after rotation may be either h+2 or h+3. In other words, the double rotation may or may not change the height of the whole subtree. It is only necessary to check whether nodes above are still balanced if the height of the y node becomes h+2.

Other balanced binary trees

Other balanced binary search trees (and more generally, n-ary search trees) also strengthen the search tree invariant to ensure that the tree remains balanced. There are many balanced search tree data structures; some of the most common are listed:

Red-black trees

2-3-4 trees

B-trees

Splay trees

Each of these except splay trees imposes an additional invariant that ensures the tree remains balanced. For example, red–black trees have a **color invariant**: every node is either red or black, and on every path from the root to a leaf, there are the same number of black nodes but no adjacent red nodes. B-trees and 2-3-4 trees are perfectly balanced n-ary search trees in which the number of children varies between ⌈n/2⌉ and n.