CMSC 420: Lecture 7
AVL Trees

Balanced Binary Trees: The binary search trees described in the previous lecture are easy to implement, but they suffer from the fact that if nodes are inserted in a poor order (e.g., increasing or decreasing) then the height of the tree can be much higher than the ideal height of $O(\log n)$. This raises the question of whether we can design a binary search tree that is *guaranteed* to have $O(\log n)$ height, irrespective of the order of insertions and deletions.

Today we will consider the oldest, and perhaps best known example of such a data structure is the famous AVL tree, which was discovered way back in 1962 by G. Adelson-Velskii and E. Landis (and hence the name "AVL").

AVL Trees: AVL tree's are height-balanced binary search trees. For any node v of the tree, let height(v) denote the height of the subtree rooted at v (shown in blue in Fig. 1(a)).

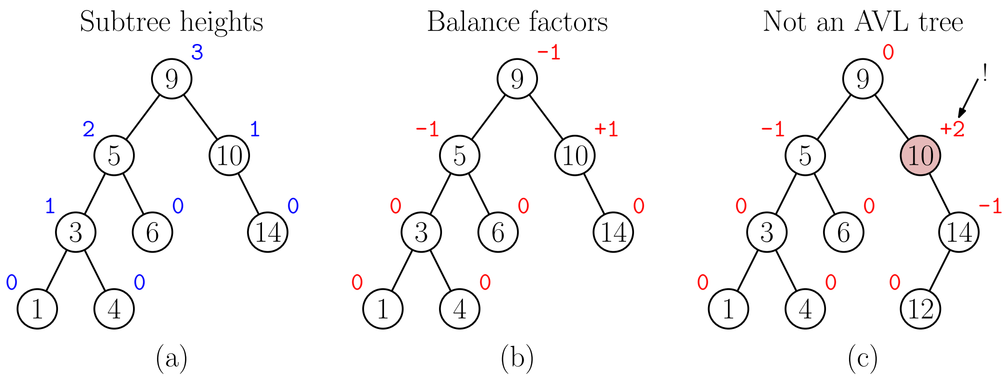

Fig. 1: AVL-tree balance condition.

Formally, we can define *height* recursively as follows

$$\text{height}(v) \;=\; \begin{cases} -1 & \text{if } v = \texttt{null} \\ 1 + \max(\text{height}(v.\text{left}), \text{height}(v.\text{right})) & \text{otherwise} \end{cases}$$

Setting the height of `null` to -1 is a convenient trick to force leaves to have a height of zero.

In an absolutely ideal height-balanced tree, the two children of any internal node would have equal heights, but it is not generally possible to achieve this goal while efficiently processing insertions and deletions. The most natural relaxation of this condition is to allow a height difference of one. Define the *AVL balance condition* to be that for every node in the tree, the absolute difference in the heights of its left and right subtrees is at most 1. An *AVLtree* is a binary search tree that satisfies the AVL balance condition.

To formalize this, define the *balance factor* of a node v (see Fig. 1(b)) to be

$$\text{balance}(v) \;=\; \text{height}(v.\text{right}) - \text{height}(v.\text{left}).$$

The AVL balance condition is equivalent to the requirement that balance(v) $\in \{-1, 0, +1\}$ for all nodes v of the tree. (Thus, Fig. 1(b) is an AVL tree, but the tree of Fig. 1(c) is not because node 10 has a balance factor of +2.) If balance(v) < -1, we say that the subtree is *left heavy* and if balance(v) $> +1$, we say that the subtree is *right heavy*.

Worst-case Height: Before discussing how we maintain this balance condition we should consider the question of whether this condition is strong enough to guarantee that the height of an AVL tree with n nodes is $O(\log n)$. Interestingly, the famous Fibonacci numbers $(0, 1, 1, 2, 3, 5, 8, \ldots)$ will arise in the analysis. Recall that for $h \geq 0$, the hth *Fibonacci number*, denoted F_h is defined by the following recurrence:

$$F_0 \;=\; 0, \quad F_1 \;=\; 1, \quad \text{and} \quad F_h \;=\; F_{h-1} + F_{h-2}, \quad \text{for } h \geq 2.$$

An important and well-known property of the Fibonacci numbers is that they grow exponentially. In particular, $F_h \approx \varphi^h/\sqrt{5}$, where $\varphi = (1 + \sqrt{5})/2 \approx 1.618$ is the famous *Golden Ratio*.[1]

Lemma: An AVL tree of height $h \geq 0$ has $\Omega(\varphi^h)$ nodes, where $\varphi = (1 + \sqrt{5})/2$.

Proof: For $h \geq 0$, let $N(h)$ denote the minimum possible number of nodes in binary tree of height h that satisfies the AVL balance condition. We will prove that $N(h) = F_{h+3} - 1$ (see Fig. 2). The result will then follow from the fact that $F_{h+3} \approx \varphi^{h+3}/\sqrt{5}$, which is equal to φ^h up to constant factors (since φ itself is a constant).

h	0	1	2	3	4	5
F_{h+3}	2	3	5	8	13	21
$N(h)$	1	2	4	7	12	20

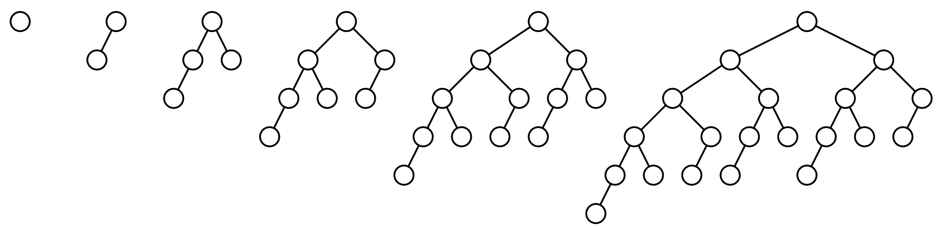

Fig. 2: Minimal AVL trees and Fibonacci numbers.

Our proof is by induction on h. First, observe that a tree of height zero consists of a single root node, so $N(0) = 1$. Also, the smallest possible AVL tree of height one consists of a root and a single child, so $N(1) = 2$. By definition of the Fibonacci numbers, we have $F_{0+3} = 2$ and $F_{1+3} = 3$, and thus $N(i) = F_{i+3} - 1$, for these two basis cases.

For $h \geq 2$, let h_L and h_R denote the heights of the left and right subtrees, respectively. Since the tree has height h, one of the two subtrees must have height $h - 1$, say, h_L. To minimize the overall number of nodes, we should make the other subtree as short as possible. By the AVL balance condition, this implies that $h_R = h - 2$. Adding a $+1$ for the root, we have $N(h) = 1 + N(h - 1) + N(h - 2)$. We may apply our induction hypothesis to conclude that

$$\begin{aligned} N(h) \;&=\; 1 + N(h-1) + N(h-2) \;=\; 1 + (F_{h+2} - 1) + (F_{h+1} - 1) \\ &=\; F_{h+2} + F_{h+1} - 1 \;=\; F_{h+3} - 1, \end{aligned}$$

as desired.

[1] Here is a sketch of a proof. Let us conjecture that $F_h \approx \varphi^h$ for some constant φ. Since the function grows very (exponentially) fast, we may ignore the tiny contribution of the $+1$ in the definition for large h. Substituting our conjectured value for F_h into the above recurrence, we find the φ satisfies $\varphi^h = \varphi^{h-1} + \varphi^{h-2}$. Removing the common factor of φ^{h-2}, we have $\varphi^2 = \varphi + 1$, that is, $\varphi^2 - \varphi - 1 = 0$. This is a quadratic equation, and by applying the quadratic formula, we conclude that $\varphi = (1 + \sqrt{5})/2$.

Corollary: An AVL tree with n nodes has height $O(\log n)$.

Proof: Let lg denote logarithm base 2. From the above lemma, up to constant factors we have $n \geq \varphi^h$, which implies that $h \leq \log_\varphi n = \lg n / \lg \varphi$. Since $\varphi > 1$ is a constant, so is $\log \varphi$. Therefore, h is $O(\log n)$. (If you work through the math, the actual bound on the height is roughly $1.44 \lg n$. In other words, in the worst case, an AVL tree is suboptimal with respect to height by a factor of at most 1.44)

Since the height of the AVL tree is $O(\log n)$, it follows that the `find` operation takes this much time. All that remains is to show how to perform insertions and deletions in AVL trees, and how to restore the AVL balance condition efficiently after each insertion or deletion.

Rotation: In order to maintain the tree's balance, we will employ a simple operation that locally modifies subtree heights, while preserving the tree's inorder properties. This operation is called *rotation*. It comes in two symmetrical forms, called a *right rotation* and a *left rotation* (see Fig. 3(a) and (b)).

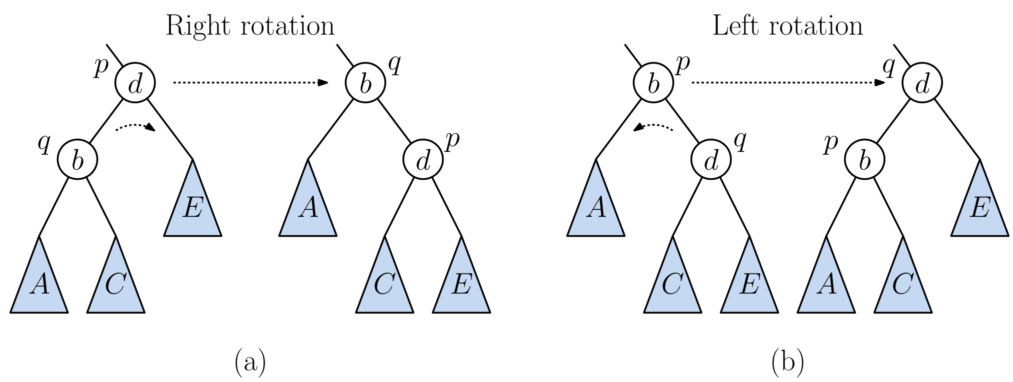

Fig. 3: (Single) Rotations. (Triangles denote subtrees, which may be null.)

We have intentionally labeled the elements of Fig. 3 to emphasize the fact that the inorder properties of the tree are preserved. That is, $A < b < C < d < E$.

Unfortunately, a single rotation is not always be sufficient to rectify a node that is out of balance. To see why, observe that the single rotation does not alter the height of subtree C. If it is too heavy, we need to do something else to fix matters. This is done by combining two rotations, called a *double rotation*. They come in two forms, *left-right rotation* and *right-left rotation* (Fig. 4). To help remember the name, note that the left-right rotation, called `rotateLeftRight(p)`, is equivalent to performing a left rotation to the `p.left` (labeled b in Fig. 4(a)) followed by a right rotation to `p` (labeled d in Fig. 4(a)). The right-left rotation is symmetrical (see Fig. 4(b)).

Insertion: The insertion routine for AVL trees starts exactly the same as the insertion routine for standard (unbalanced) binary search trees. In particular, we search for the key and insert a new node at the point that we fall out of the tree. After the insertion of the node, we must update the subtree heights, and if the AVL balance condition is violated at any node, we then apply rotations as needed to restore the balance.

The manner in which rotations are applied depends on the nature of the imbalance. An insertion results in the addition of a new leaf node, and so the balance factors of the ancestors can be altered by at most ± 1. Suppose that after the insertion, we find that some node has a

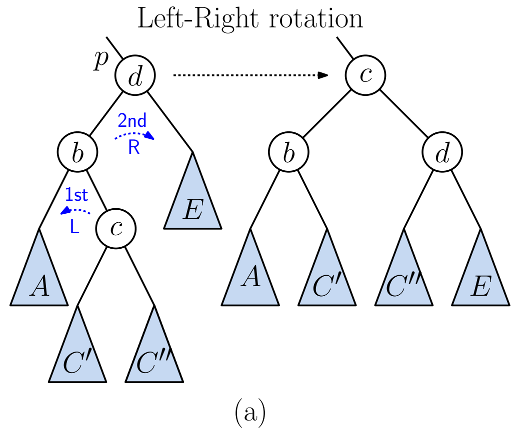

Fig. 4: Double rotations (`rotateLeftRight(p)` and `RotateRightLeft(p)`).

balance factor of -2. For concreteness, let us consider the naming of the nodes and subtrees shown in Fig. 5, and let the node in equation be d. Note that this node must be along the search path for the inserted node, since these are the only nodes whose subtree heights may have changed. Clearly, d's left subtree, is too deep relative to d's right subtree E. Let b denote the root of d's left subtree.

At this point there are two cases to consider. Either b's left child is deeper or its right child is deeper. (The subtree that is deeper will be the one into which the insertion took place.)

Left-left heavy: Let's first consider the case where the insertion took place in the the subtree A (see Fig. 5(b)). In this case, we can restore balance by performing a right rotation at node d. This operation pulls the deep subtree A up by one level, and it pushes the shallow subtree E down by one level (see Fig. 5(c)). Observe that the depth of subtree C is unaffected by the operation. It follows that the balance factors of the resulting subtrees rooted at b and d are now both zero. The AVL balance condition is satisfies by all nodes, and we are in good shape.

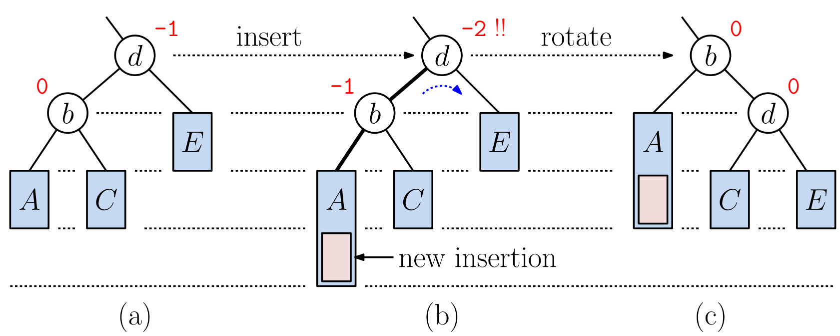

Fig. 5: Restoring balance after insertion through a single rotation.

Left-right heavy: Next, us consider the case where the insertion occurs within subtree C (see Fig. 6(b)). As observed earlier, the rotation at d does not alter C's depth, so we will need to do something else to fix this case. Let c be the root of the subtree C, and let C' and C'' be its two subtrees (either of these might be `null`). The insertion took place into either C' or C''. (We don't care which, but the "?" in the figure indicate our uncertainty.) We restore balance by performing two rotations, first a left rotation at b and then a right rotation at d (see Fig. 6(c)). This *double rotation* has the effect of moving the subtree E down one level, leaving A's level unchanged, and pulling both C' and C'' up by one level.

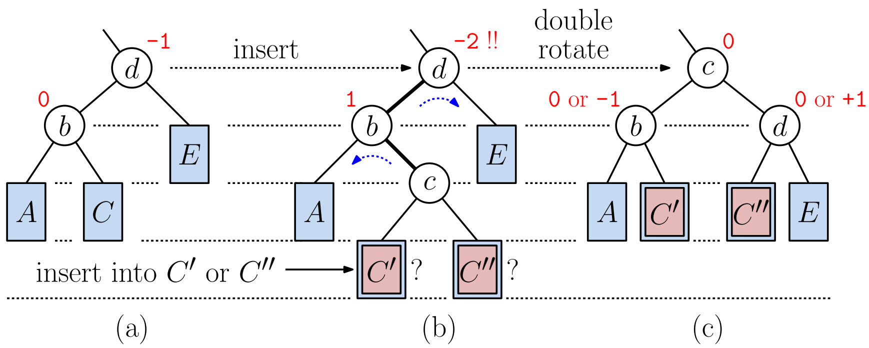

Fig. 6: Restoring balance after insertion through a double rotation.

The balance factors at nodes b and d will depend on whether the insertion took place into C' or C'', but irrespective of which, they will be in the range from -1 to $+1$. The balance factor at the new root node c is now 0. So, again we are all good with respect to the AVL balance condition.

Utilities: Before giving the code for insertion and deletion, let's introduce a few utilities. We assume that we store the height of each node in a field p.height. We first define utility functions `height` and `updateHeight`, which access and update height information. Next, `balanceFactor` computes the balance factor from the child heights. Finally, we give code for single and double rotations.

AVL Tree Utilities

```
int height(Node p) { return p == null ? -1 : p.height }
void updateHeight(Node p) { p.height = 1 + max(height(p.left), height(p.right)) }
int balanceFactor(Node p) { return height(p.right) - height(p.left) }

Node rotateRight(Node p) {              // right single rotation
    Node q = p.left
    p.left = q.right                    // swap inner child
    q.right = p                         // bring q above p
    updateHeight(p)                     // update subtree heights
    updateHeight(q)
    return q                            // q replaces p
}

Node rotateLeft(Node p) { ... symmetrical to rotateRight ... }

Node rotateLeftRight(Node p) {          // left-right double rotation
    p.left = rotateLeft(p.left)
    return rotateRight(p)
}
Node rotateRightLeft(Node p) { ... symmetrical to rotateLeftRight ... }
```

AVL Insertion: The insert function is virtually identical to the insert function for standard binary search trees, but with one change. After the insertion, we invoke a function `rebalance`, which updates heights, checks balance factors, and applies the appropriate rotations to restore the AVL balance properties. To determine which rotation to apply, we first determine whether we are left or right heavy. If we are left heavy, then we determine whether we are left-left heavy or left-right heavy. In the former case, we do a single rotation, and in the latter we do a double rotation. The right-heavy cases are symmetrical.

AVL Tree Insertion

```
Node insert(Key x, Value v, Node p) {   // insert (x,v) into p's subtree
    if (p == null) {                    // fell out of tree -- create new node
        p = new Node(x, v, null, null)
    }
    else if (x < p.key) {               // x is smaller - insert left
        p.left = insert(x, p.left)      // ... insert left
    else if (x > p.key) {               // x is larger - insert right
        p.right = insert(x, p.right)    // ... insert right
    }
    else throw DuplicateKeyException    // key already in the tree?
    return rebalance(p)                 // ONLY CHANGE FROM STANDARD BST INSERT
}

Node rebalance(Node p) {                // rebalance subtree at p
    if (p == null) return p             // null - nothing to do
    if (balanceFactor(p) < -1) {        // left heavy?
        if (height(p.left.left) >= height(p.left.right)) { // left-left heavy?
             p = rotateRight(p)         // fix with single rotation
        else                            // left-right heavy?
             p = rotateLeftRight(p)     // fix with double rotation
    } else if (balanceFactor(p) > +1) { // right heavy?
        if (height(p.right.right) >= height(p.right.left)) { // right-right heavy?
             p = rotateLeft(p)          // fix with single rotation
        else                            // right-left heavy?
             p = rotateRightLeft(p)     // fix with double rotation
    }
    updateHeight(p)                     // update p's height
    return p                            // return link to updated subtree
}
```

An interesting feature of the insertion algorithm (which is not at all obvious) is that whenever rebalancing is required, the height of the modified subtree is the same as it was before the insertion. This implies that no further rotations are required. (This is not the case for deletion, however.)

Deletion: After having put all the infrastructure together for rebalancing trees, deletion is actually relatively easy to implement. As with insertion, deletion process is the same as for standard unbalanced binary search trees. The only difference is that as we are backing out of the recursion following the deletion, we need to update heights, check balance factors, and perform the necessary rotations.

As with insertion, there are multiple cases. If a node is out of balance, it is either left-heavy or right-heavy (but here, the heaviness is due to a deletion from the opposite side of the tree).

We need only consider left-heaviness, since right-heaviness is symmetrical.

We say that the node `p` is left-left heavy if its left-left grandchild's height is greater than or equal to its left-right grandchild (see Fig. 7). If so, we remedy this with a single right rotation at `p`.

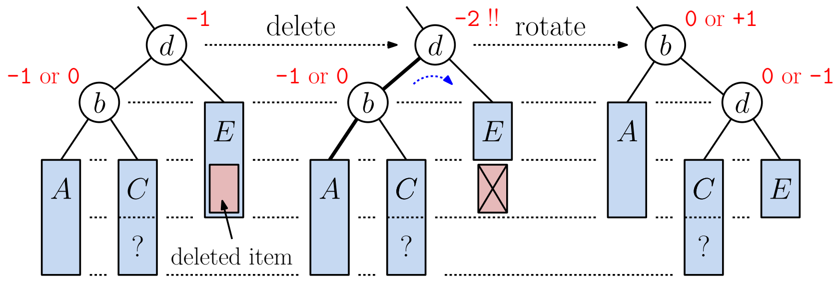

Fig. 7: Restoring balance after deletion in the left-left heavy case. Node d is left heavy following the deletion from its right subtree. The left-left grandchild A is at least as tall as the left-right grandchild C. ()The "?" illustrate places where the subtree's height is not fully determined.) A single right rotation at d restores balance.

The other possibility is that `p` is left-right heavy, meaning that its left-right grandchild is strictly taller than its left-left grandchild (see Fig. 8). In this case, we remedy the balance through a left-right double rotation.

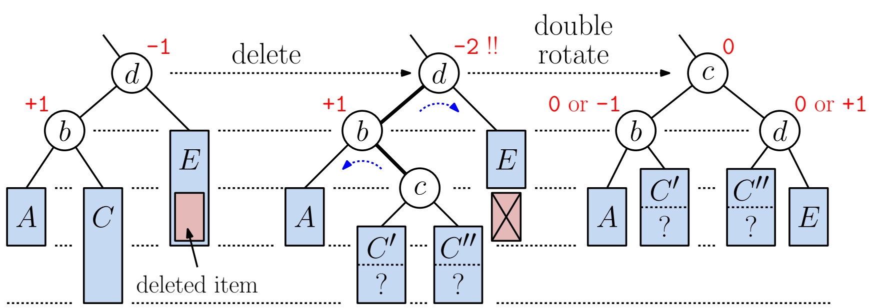

Fig. 8: Restoring balance after deletion in the left-right heavy case. Node d is left heavy following the deletion from its right subtree. The left-right grandchild C is strictly taller than its left-left grandchild A. A left-right rotation at d restores balance.

Note that in the case of the double rotation, the height of the entire tree rooted at d has decreased by 1. This means that further ancestors need to be checked for the balance condition. Unlike insertion, where at most one rebalancing operation is needed, deletion could result in a cascade of $O(\log n)$ rebalancing operations.

AVL Deletion: The pseudocode for deletion is presented below. The code is identical to the deletion code for standard (unbalanced) binary search trees, except for the final call to `rebalance`.

Lazy Deletion: (Optional) The deletion code for standard binary search tree (and, by extension, AVL trees and other balanced search trees) is generally more complicated than the insertion code. An intriguing alternative for avoiding coding up the deletion algorithm is called *lazy*

AVL Tree Deletion

```
Node delete(Key x, Node p) {              // delete x from p' subtree
    if (p == null)                        // fell out of tree?
        throw KeyNotFoundException        // ...error -- no such key
    else {
        if (x < p.data)                   // delete from left subtree
            p.left = delete(x, p.left)
        else if (x > p.data)              // delete from right subtree
            p.right = delete(x, p.right)
                                          // found it!
        else if (p.left == null || p.right == null) { // 0 or 1 child
            if (p.left == null) p = p.right
            else                p = p.left
        }
        else {                            // both children present
            r = findReplacement(p)        // find replacement node
            copy r's contents to p        // copy its contents to p
            p.right = delete(r.key, p.right) // delete the replacement
        }
    }
    return rebalance(p)                   // ONLY CHANGE FROM STANDARD BST DELETE
}
```

deletion. For each node, we maintain a boolean value indicating whether this element is *alive* or *dead*. When a key is deleted, we simply declare it to be dead, but leave it in the tree. If an attempt is made to insert a value that comes in the same relative order as a dead key, we store the new key-value pair in the dead node and declare it to now be alive. Of course, your tree may generally fill up with lots of dead nodes, so lazy deletion is usually applied only in circumstances where the number of deletions is expected to be significantly smaller than the number of insertions. Alternatively, if the number dead nodes gets too high, you can invoke a *garbage collection* process, which builds an entirely new search tree containing just the alive nodes.