Lecture 9: AVL Trees

Read: Chapt 4 of Weiss, through 4.4.

Balanced Binary Trees: The (unbalanced) binary tree described earlier is fairly easy to implement, but suffers from the fact that if nodes are inserted in increasing or decreasing order (which is not at all an uncommon phenomenon) then the height of the tree can be quite bad. Although worst-case insertion orders are relatively rare (as indicated by the expected case analysis), even inserting keys in increasing order produces a worst-case result. This raises the question of whether we can design a binary search tree which is *guaranteed* to have $O(\log n)$ height, irrespective of the order of insertions and deletions. The simplest example is that of the AVL tree.

AVL Trees: AVL tree's are height balanced trees. The idea is at each node we need to keep track of *balance information*, which indicates the differences in height between the left and right subtrees. In a perfectly balanced (complete) binary tree, the two children of any internal node have equal heights. However, maintaining a complete binary tree is tricky, because even a single insertion can cause a large disruption to the tree's structure. But we do not have to do much to remedy this situation. Rather than requiring that both children have exactly the same height, we only require that their heights differ by at most one. The resulting search trees are called *AVL trees* (named after the inventors, Adelson-Velskii and Landis). These trees maintain the following invariant:

AVL balance condition: For every node in the tree, the heights of its left subtree and right subtree differ by at most 1. (The height of a null subtree is defined to be -1 by convention.)

In order to maintain the balance condition we can add a new field, `balance` to each node, which stores the difference in the height of the right subtree and the height of the left subtree. This number will always be either -1, 0, or $+1$ in an AVL tree. (Thus it can be stored using only 2 bits from each node.) Rather than using the balance field in the code below, we will assume a simpler method in which we store the height of each subtree.

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 will be $O(\log n)$. To prove this, let's let $N(h)$ denote the minimum number of nodes that can be in an AVL tree of height h. We can generate a recurrence for $N(h)$. Clearly $N(0) = 1$. In general $N(h)$ will be 1 (for the root) plus $N(h_L)$ and $N(h_R)$ where h_L and h_R are the heights of the two subtrees. Since the overall tree has height h, one of the subtrees must have height $h-1$, suppose h_L. To make the other subtree as small as possible we minimize its height. It's height can be no smaller than $h-2$ without violating the AVL condition. Thus we have the recurrence

$$\begin{aligned} N(0) &= 1 \\ N(h) &= N(h-1) + N(h-2) + 1. \end{aligned}$$

This recurrence is not well defined since $N(1)$ cannot be computed from these rules, so we add the additional case $N(1) = 2$. This recurrence looks very similar to the Fibonacci recurrence $(F(h) = F(h-1) + F(h-2))$. In fact, it can be argued (by a little approximating, a little cheating, and a little constructive induction) that

$$N(h) \approx \left(\frac{1+\sqrt{5}}{2}\right)^h.$$

[1]Copyright, David M. Mount, 2001

The quantity $(1+\sqrt{5})/2 \approx 1.618$ is the famous *Golden ratio.* Thus, by inverting this we find that the height of the worst case AVL tree with n nodes is roughly $\log_\phi n$, where ϕ is the Golden ratio. This is $O(\log n)$ (because log's of different bases differ only by a constant factor).

All that remains is to show how to perform insertions and deletions in AVL trees, and how to restore the AVL balance condition after each insertion or deletion. We will do this next time.

Insertion: The insertion routine for AVL trees starts exactly the same as the insertion routine for binary search trees, but after the insertion of the node in a subtree, we must ask whether the subtree has become unbalanced. If so, we perform a rebalancing step.

Rebalancing itself is a purely local operation (that is, you only need constant time and actions on nearby nodes), but requires a little careful thought. The basic operation we perform is called a *rotation.* The type of rotation depends on the nature of the imbalance. Let us assume that the source of imbalance is that the left subtree of the left child is too deep. (The right subtree of the right child is handled symmetrically.) See the figure below. The operation performed in this case is a *right single rotation.* Notice that after this rotation has been performed, the balance factors change. The heights of the subtrees of `b` and `d` are now both even with each other.

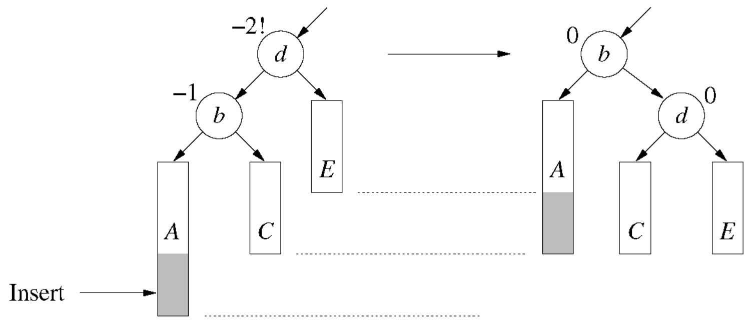

Figure 22: Single rotation.

On the other hand, suppose that the heavy grandchild is the right subtree of the left subtree. (Again the the left subtree of the right child is symmetrical.) In this case note that a single rotation will not fix the imbalance. However, two rotations do suffice. See the figure below. In particular, do a left rotation on the left-right grandchild (the right child of the left child), and then a right rotation on the left child, you will restore balance. This operation is called a *double rotation*, and is shown in the figure below. (We list multiple balanced factors to indicate the possible values. We leave it as an exercise to determine how to update the balance factors.)

Before presenting the code for insertion, we present the utilities routines used. The rotations are called `rotateL()` and `rotateR()` for the left and right single rotations, respectively, and `rotateLR()` and `rotateRL()` for the left-right and right-left double rotations, respectively. We also use a utility function `height(t)`, which returns `t.height` if t is nonnull and -1 otherwise (the height of a null tree).

AVL Tree Utilities

```
int height(AvlNode t) { return t == null ? -1 : t.height}

AvlNode rotateR(AvlNode t)              // right single rotation
{
```

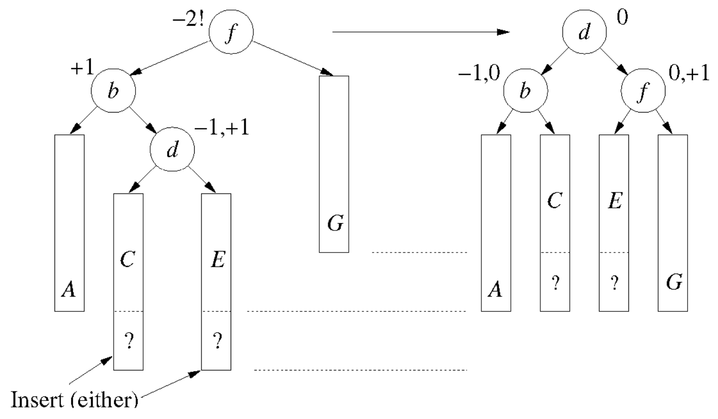

Figure 23: Left-right double rotation.

```
        AvlNode s = t.left;                 // (t.bal = -2; s.bal = 0 or 1)
        t.left = s.right;                   // swap inner child
        s.right = t;                        // bring s above t
        t.height = ...exercise...           // update subtree heights
        s.height = ...exercise...
        return s;                           // s replaces t
    }

    AvlNode rotateLR(AvlNode t)             // left-right double rotation
    {
        t.left = rotateL(t.left);
        return rotateR(t);
    }
```

The recursive insertion code is given below. We use the same structure given in Weiss. We use the usual trick of passing pointer information up the tree by returning a pointer to the resulting subtree after insertion.

AVL Tree Insertion

```
    AvlNode insert(Element x, AvlNode t) {  // recursive insertion method
        if (t == null) {                    // bottom or tree: create new node
            t = new AvlNode(x);             // create node and initialize
        }
        else if (x < t.element) {
            t.left = insert(x, t.left);     // insert recursively on left
                                            // check height condition
            if (height(t.left) - height(t.right) == 2) {
                                            // rotate on the left side
                if (x < t.left.element)     // left-left insertion
                    t = rotateR(t);
                else                        // left-right insertion
                    t = rotateLR(t);
            }
                                            // update height
            t.height = max(height(t.left), height(t.right)) + 1;
        }
        else if (x > t.element) {
            ...symmetric with left insertion...
        }
        else {
            ...Error: duplicate insertion ignored...
        }
        return t;
    }
```

As mentioned earlier, and interesting feature of this algorithm (which is not at all obvious) is that after the first rotation, the height of the affected subtree is the same as it was before the insertion, and hence no further rotations are required.

Deletion: Deletion is similar to insertion in that we start by applying the deletion algorithm for unbalanced binary trees. Recall that this breaks into three cases, leaf, single child, and two children. In the two children case we need to find a replacement key. Once the deletion is finished, we walk back up the tree (by returning from the recursive calls), updating the balance factors (or heights) as we go. Whenever we comes to an unbalanced node, we apply an appropriate rotation to remedy the situation.

The deletion code is messier than the insertion code. (We will not present it.) But the idea is the same. Suppose that we have deleted a key from the left subtree and that as a result this subtree's height has decreased by one, and this has caused some ancestor to violate the height balance condition. There are two cases. First, if balance factor for the right child is either 0 or +1 (that is, it is not left heavy), we can perform a single rotation as shown in the figure below. We list multiple balance factors, because there are a number of possibilities. (We leave it as an exercise to figure out which balance factors apply to which cases.)

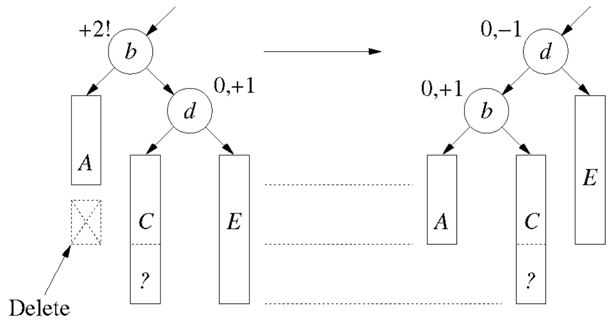

Figure 24: Single rotation for deletion.

On the other hand, if the right child has a balance factor of −1 (it is left heavy) then we need to perform a double rotation, as shown in the following figure. A more complete example of a deletion is shown in the figure below. We delete element 1. The causes node 2 to be unbalanced. We perform a single left rotation at 2. However, now the root is unbalanced. We perform a right-left double rotation to the root.

Lazy Deletion: The deletion code for AVL trees is almost twice as long as the insertion code. It is not all that more complicated conceptually, but the number of cases is significantly larger. Our text suggests an interesting alternative for avoiding the pain of coding up the deletion algorithm, called *lazy deletion*. The idea is to not go through the entire deletion process. For each node we maintain a boolean value indicating whether this element is *alive* or *dead*. When

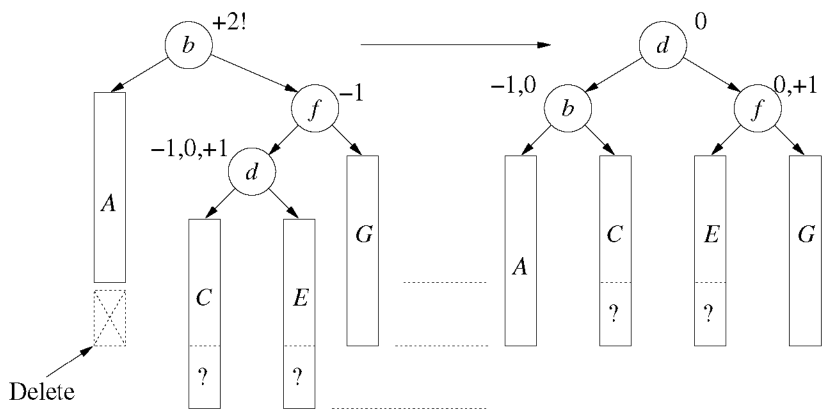

Figure 25: Double rotation for deletion.

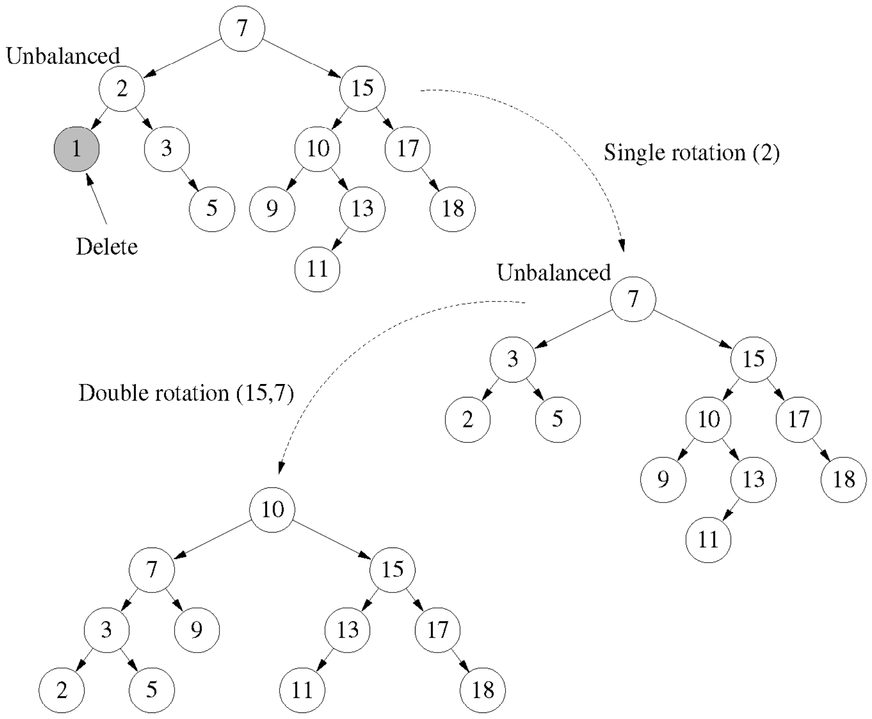

Figure 26: AVL Deletion example.

a key is deleted, we simply declare it to be dead, but leave it in the tree. If an attempt is made to insert the same key value again, we make the element alive again. Of course, after a long sequence of deletions and insertions, it is possible that the tree has many dead nodes. To fix this we periodically perform a garbage collection phase, which traverses the tree, selecting only the live elements, and then building a new AVL tree with these elements.