AVL Trees

AVL trees are binary search trees with a height invariant requiring that at any node, the left and right subtrees differ in height by at most 1.

The height of a tree $T$ can be defined as follows:

- The empty tree has height 0.
- If $T$’s left and right subtrees have heights $h_l$ and $h_r$, respectively, then $T$ has height $1 + \max(h_l, h_r)$

When we talked about BSTs last week, we saw that we were able to maintain the ordering invariant just by where in the tree we inserted elements. How do we guarantee that we can preserve the height invariant when we know neither what elements we’re going to insert nor what order we’re going to insert them? We write our insertion function so that after every insert, we dynamically restore balance if it gets thrown out of whack.

This visualizer can show us some AVL trees in action: [http://www.cs.usfca.edu/ galles/visualization/AVLtree.html](http://www.cs.usfca.edu/ galles/visualization/AVLtree.html)

Rotations

Having played a little bit with the visualizer, you’ve probably seen the AVL tree rebalancing itself. That motion is referred to as a rotation. There are two kinds of rotations - left and right - and they work in the following way:

Notice that the left-to-right order stays the same before and after a rotation. Only the heights of the trees change.

When we insert an element into an AVL tree, we do so in such a way that the BST invariant is maintained. Doing so may break the height invariant. We fix this by rebalancing the tree after each insert, using rotations. Wikipedia shows the four ways to rebalance an AVL tree. Notice that two of them are intermediate steps in balancing:
To handle the Left-Left and Right-Right cases, we only need a single rotation. To handle the Left-Right and Right-Left cases, we can change them to the Left-Left and Right-Right cases in just a single rotation. So you need at most 2 rotations, a constant time operation to rebalance any AVL tree.

It is important that we rebalance an AVL tree at the lowest node of violation. Why is that?

**AVL Tree Implementation**

In C, we represent AVL trees as structs with a `data` field, a `height` field, and pointers to the left and right subtrees:

```c
struct tree {
    elem data;  int height;
    struct tree *left; struct tree *right;
};
```

We carry out a right rotation in the following function (knowing that a left rotation is handled symmetrically):

```c
tree* rotate_right(tree* T)
//@requires is_ordtree(T);
//@requires T != NULL && T->left != NULL;
//@ensures is_ordtree(result);
//@ensures result != NULL && result->right != NULL;
{
    tree* root = T->left;
    T->left = root->right;
    root->right = T;
    fix_height(root->right); /* must be first */
    fix_height(root);
    return root;
}
```

We do a right rebalance in the following way (in the case that we inserted into `T->right`):

```c
tree* rebalance_right(tree* T)
//@requires T != NULL;
//@requires is_avl(T->left) && is_avl(T->right);
//@ensures is_avl(result);
{
    tree *l = T->left;
    tree *r = T->right;
    int hl = height(l);
    int hr = height(r);
    if (hr > hl + 1) { //@assert hr == hl + 2;
        if (height(r->right) > height(r->left)) { // "Right right" case in the Wikipedia diagram
            //@assert height(r->right) == hl + 1;
            T = rotate_left(T);
            //@assert height(T) == hl + 2;
            return T;
        } else { // "Right left" case in the Wikipedia diagram
            //@assert height(r->left) == hl + 1;
            T->right = rotate_right(T->right);
            T = rotate_left(T);
            //@assert height(T) == hl + 2;
            return T;
        }
    } else { //@assert !(hr > hl + 1);
        fix_height(T);
        return T;
    }
}
```