15-122: Principles of Imperative Computation

Recitation 19 — Josh Zimmerman

AVL Trees

When talking about binary search trees, we saw that operations like insert and lookup ran in $O(\log(n))$ time on a balanced binary tree with n items. But if the tree isn't balanced, we could get far worse times—even up to $O(n)$!

AVL trees solve this problem. AVL trees are binary search trees where we also maintain a *height invariant*.

First, we need to say what the height of a binary tree is. There are two equivalent definitions we can use. We'll mention both so that you can use whichever makes more sense to you.

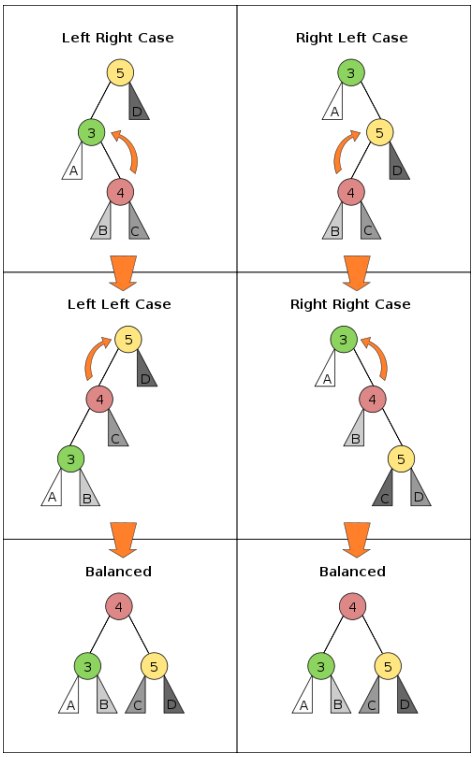
The first is that the height of a tree is the maximum number of nodes from the root to a leaf, inclusive.

The second is recursive: the empty tree has height 0 and any other tree has height $1 + \max(hl, hr)$ where hl and hr are the heights of the left and right children of the tree, respectively.

The height invariant for AVL trees states that for every node in the tree, the height of the left and right subtrees differ by at most 1, or in other words, if the left and right subtrees have height hl and hr, respectively, $|hl - hr| \leq 1$. This lets us maintain balance of the tree, and so ensures that we'll have at worst $O(\log n)$ lookup time.

In an AVL tree, we insert an element much like we would in a BST, but we then check to see if the height invariant is violated and rebalance the tree if necessary. This approach means that the difference between the height of any two children is at most 2 when we're rebalancing, which lets us split up the possibilities into four cases.

Here's a depiction of the cases and the rotations we do, from Wikipedia.



Note that two of the cases (the ones the diagram calls "Left Right" and "Right Left") can be transformed into the other two with a single rotation, and those two can be made balanced with a single rotation. Importantly, this means that we *never* need more than 2 rotations to restore balance an AVL tree after inserting an element. Since rotation is a constant time operation, this means that insertion into an AVL tree is only at worst a constant amount slower than insertion into a BST!

It's also important to note that we do these rotations at the *lowest* violation of the height invariant in the AVL tree.

Let's now go through some examples of these cases, to familiarize ourselves with the rotations. (We'll use `http://www.cs.usfca.edu/~galles/visualization/AVLtree.html`)

Implementation

Now that we have a conceptual understanding of AVL trees, let's talk about the implementation details.

To keep track of the height in an efficient manner, we add a `height` field to the data structure:

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

Note that we now need to do more work in the `is_balanced` function, since we need to check that `height` accurately describes the height of each node of the tree (if it didn't, we wouldn't be able to guarantee that the tree was balanced, since we use the `height` variable when checking balance in our `rebalance_right` and `rebalance_left` functions).

First, let's look at `rotate_right`. (`rotate_left` is exactly symmetric)

```
1  tree *rotate_right(tree *T) {
2    REQUIRES(is_ordtree(T));
3    REQUIRES(T != NULL && T->left != NULL);
4    tree *root = T->left;
5    T->left = root->right;
6    root->right = T;
7    fix_height(root->right); /* must be first */
8    fix_height(root);
9    ENSURES(is_ordtree(root));
10   ENSURES(root != NULL && root->right != NULL);
11   return root;
12 }
```

Let's go through some sketches to help illustrate what this function is doing. If you're reading through this after recitation, I encourage you to sketch some diagrams on a piece of paper or a whiteboard to help you understand the code. (Note that `rotate_left` is symmetric to `rotate_right`, so if you understand one you should be able to understand the other.)

Now, let's look at our `rebalance_right` code, on an intuitive level. (The `rebalance_left` code is similar, and for the sake of a shorter handout I'll leave it out. I highly encourage you to work with it and make sure you understand it, as the code for AVL trees is tricky.)

```
1 tree *rebalance_right(tree *T) {
2  REQUIRES(T != NULL);
3  REQUIRES(is_avl(T->left) && is_avl(T->right));
4  /* also requires that T->right is result of insert into T */
5
6  tree *l = T->left;
7  tree *r = T->right;
8  int hl = height(l);
9  int hr = height(r);
10 if (hr > hl + 1) {
11   ASSERT(hr == hl + 2);
12   if (height(r->right) > height(r->left)) {
13     // We're in the "right right" case in the diagram from Wikipedia
14     ASSERT(height(r->right) == hl + 1);
15     T = rotate_left(T);
16     // Note that hl + 2 == hr here
17     ASSERT(height(T) == hl+2);
18   }
19   else {
20     // We're in the "right left" case in the diagram from Wikipedia
21     ASSERT(height(r->left) == hl + 1);
22     /* double rotate left */
23     T->right = rotate_right(T->right);
24     T = rotate_left(T);
25     // Note that hl + 2 == hr here
26     ASSERT(height(T) == hl+2);
27   }
28 }
29 else {
30   // the tree is already balanced, so just update the height
31   ASSERT(!(hr > hl+1));
32   fix_height(T);
33 }
34 ENSURES(is_avl(T));
35 return T;
36 }
```

Note: We're missing a precondition we need here, so we won't do a formal proof.

We will, however, discuss the intuition behind the function.

Give an informal explanation of why this function works. This shouldn't be a proof—just explain it, as though you're trying to teach someone.

It may help you to explain it to the person sitting next to you, since explaining things is a very good way to learn them.