15-122: Principles of Imperative Computation

Recitation 19 Solutions — Josh Zimmerman

Implementation and explanation

Here's the code for rebalancing the right subtree.

```
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.

Solution:

NOTE: This is *NOT* a formal proof and we would not accept it as a proof. It is solely to help you get the intuition for how AVL rotations work.

If we enter the branch of the conditional that starts on line 10, we know the tree is unbalanced, since the height of the right subtree is more than 1 larger than the height of the left subtree.

Now, we have two cases: either the right sub-subtree has a larger height than the left sub-subtree, or not.

If the right sub-subtree is longer, then we simply need to rotate left to even things out, as we can see intuitively in the wikipedia picture, since the left sub-subtree doesn't cause any problems by being where it is but the right sub-subtree does. Note that this means the height of the whole tree is now equal to the height of the right subtree, since the height of the left subtree was less than the height of the right subtree.

Otherwise, the left sub-subtree is longer, and so it's causing the problems. If we rotate the right subtree right, then we're in the "right right" case, and the left sub-subtree is no longer causing a problem, so the same reasoning as above applies.