

15-122: Principles of Imperative Computation

Recitation Week 8

Nivedita Chopra, Josh Zimmerman

Visualizing AVL trees

Use the visualization at <http://www.cs.usfca.edu/~galles/visualization/AVLtree.html> to insert these keys into the tree in the following order:

1, 2, 5, 3, 4

Then delete the keys 2 and 4.

Two ways of doing rotations

Remember to draw pictures! The way we did rotations in class used the common trick of returning a new root from the function. This made the function easier to write.

```
1 tree* rotate_left(tree* T)
2 //@requires is_tree(T) && T != NULL && T->right != NULL;
3 //@ensures is_tree(\result);
4 {
5     tree* R = _____
6     _____
7     _____
8     _____
9     _____
10    _____
11 }
```

With a bit more work, we can write a rotate function that keeps the root the same as it was before; this means we don't have to return a new root. Are any of the lines below unnecessary?

```
1 void rotate_left(tree* T)
2 //@requires is_tree(T) && T != NULL && T->right != NULL;
3 //@ensures is_tree();
4 {
5     elem x = T->data;
6     elem y = T->right->data;
7     tree* A = T->left;
8     tree* B = T->right->left;
9     tree* C = T->right->right;
10    T->data = _____;
11    T->left = _____;
12    T->left->data = _____;
13    T->left->left = _____;
14    T->left->right = _____;
15    T->right = _____;
16    fix_height(_____);
17    fix_height(_____);
18 }
```

AVL rotations

In each of the blank spots below, *draw* the shape of the tree, including what you know about the height(s) of the subtrees, and show how the tree needs to be modified to rebalance the tree.

```
1 tree* rebalance_right(tree* T)
2 //@requires T != NULL && T->right != NULL;
3 //@requires is_tree(T->left) && is_tree(T->right);
4 // Not specified: T was balanced before an AVL insertion into T->right
5 //@ensures is_tree(\result);
6 {
7   if (height(T->right) - height(T->left) == 2) {
8     if (height(T->right->left) > height(T->right->right)) {
9
10
11
12
13
14
15
16
17
18
19   } else {
20     //@assert height(T->right->left) < height(T->right->right);
21
22
23
24
25
26
27
28
29
30
31   }
32 } else {
33   fix_height(T);
34 }
35 return T;
36 }
```

Checkpoint 0

The correctness of the rebalance function above depends on the invariant that we didn't write as a checked precondition: if the tree is unbalanced, then the right subtree results from a single BST insertion (no rebalancing) into a previously-balanced AVL tree.

What are some inputs that satisfy the preconditions on lines 2 and 3 but violate this (unchecked) precondition and cause a contract to fail as a result? How could we add a simple precondition that would exclude this counterexample?

Checkpoint 1

Write a recursive function that finds the maximum element in a BST.

Write a non-recursive function that finds the maximum element in a BST.