15-122: Principles of Imperative Computation

R8: Leaf the rotations to us

Andrew Benson, Josh Zimmerman

Binary Search Trees

We define a binary search tree (BST) recursively as a node with data and pointers to left and right subtrees:

```
1 typedef struct tree_node tree;
2 struct tree_node {
3   elem data;
4   tree* left;
5   tree* right;
6 };
```

BSTs must abide by the following **ordering invariant**:

For any node with key k in a BST, all keys of elements in the left subtree are strictly less than k and all keys of elements in the right subtree are strictly greater than k.

As a corollary, no element can appear more than once in our definition of a BST.

Recall that we also defined the **height** of a BST as the maximum number of nodes that can be reached by a sequence of left and right subtree pointers. Formally, we define the height of a `NULL` BST as 0 and the height of any other tree as `1 + max(T->left, T->right)`.

Checkpoint 0

Suppose you have the following keys:

$$-6, 5, 0, 2, 7, -3, 9$$

Draw two binary search trees that hold these 7 keys: one that has the minimum possible height, and one that has the maximum possible height.

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.

AVL Trees

AVL Trees are binary search trees that have the additional invariant that the tree is balanced. We write this formally as the **height invariant**: at any node in the tree, the heights of the left and right subtrees may differ by at most 1. We use rotations to maintain this invariant across insertions and deletions.

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. Note that you should maintain AVL tree invariants; in particular, you may find the function `void fix_height(tree* T)` useful.

```
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 }
```