Binary Search Trees

A binary tree, like a linked list, is a recursive data structure. The only difference is that a node can have 0, 1, or 2 children, which can be other nodes or leaves. A leaf is a type of node that has no children.

```c
typedef struct tree_node tree;
struct tree_node {
  elem data;
  tree* left;
  tree* right;
};
```

We call left and right subtrees.

A binary search tree (BST) has an additional invariant, the ordering invariant. For a node with key \( k \), all elements in the left subtree must have keys that are strictly less than than \( k \), and all elements in the right subtree must have keys that are strictly greater than \( k \) (By this definition, we do not allow duplicate keys).

**Checkpoint 0**

Circle all of the nodes that contain keys that violate the ordering invariant.

![Diagram of a binary search tree with keys -8, -2, 5, 0, 2, 7, 9, 11, 17, and 8, with nodes 5, 2, and 9 violating the ordering invariant.]

**Balanced Search Tree**

Let’s take a look at two binary trees that contain the same elements.

![Two binary trees with the same elements: one with height 7 and the other with height 3. The height of the left tree is 7, while the height of the right tree is only 3. Say we want to access the element 9. In the left tree, we need to travel down 7 levels, while on the right we only need to go down 3. Remember that we have to do a comparison at each level, so we’d like our trees to be as short as possible.]

The height of the left tree is 7, while the height of the right tree is only 3. Say we want to access the element 9. In the left tree, we need to travel down 7 levels, while on the right we only need to go down 3. Remember that we have to do a comparison at each level, so we’d like our trees to be as short as possible.
AVL Trees

AVL trees add an additional invariant in order to ensure the tree is balanced. The **height invariant** requires the height of the left and right subtrees only differ by at most 1. How do we preserve this invariant? Rotations.

We insert nodes just as we would with a plain BST, but then check to see if (and where) the height invariant is violated. Say we insert 5 into the following tree:

```
  15
 /   \
9     122
 /  \
6   12
 /  \  \
5   12 122
```

Our tree is looking pretty unbalanced. But where is the violation? 6 and 9’s subtrees only differ by 1, but the left subtree of 15 has height 3, while the right has height 1. To fix this, we rotate right at 15. Notice that 12 is now the left child of 15, rather than the right child of 9.

Use the visualization at [http://www.cs.usfca.edu/~galles/visualization/AVLtree.html](http://www.cs.usfca.edu/~galles/visualization/AVLtree.html) to insert 1, 2, 5, 3, 4 into the tree in the following order, then delete the keys 2 and 4.

**Checkpoint 1**

Now that you’ve seen rotations, let’s write code.

```c
1  tree* rotate_right(tree* T)
2    //@requires is_tree(T) && T != NULL && T->left !=NULL;
3    //@ensures is_tree(result);
4    {
5       tree* L = ____________________________
6       ____________________________
7       ____________________________
8       ____________________________
9    }
```

The code for `rotate_left` is simply the mirror of this function.