Recitation 09: Rotating Rotation — Thursday March 14th

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.

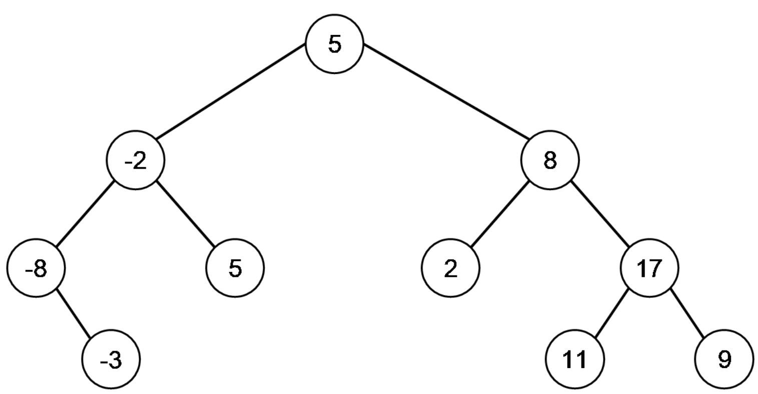
```
typedef struct tree_node tree;
struct tree_node {
  tree* left;
  elem data;
  tree* right;
};
```

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).

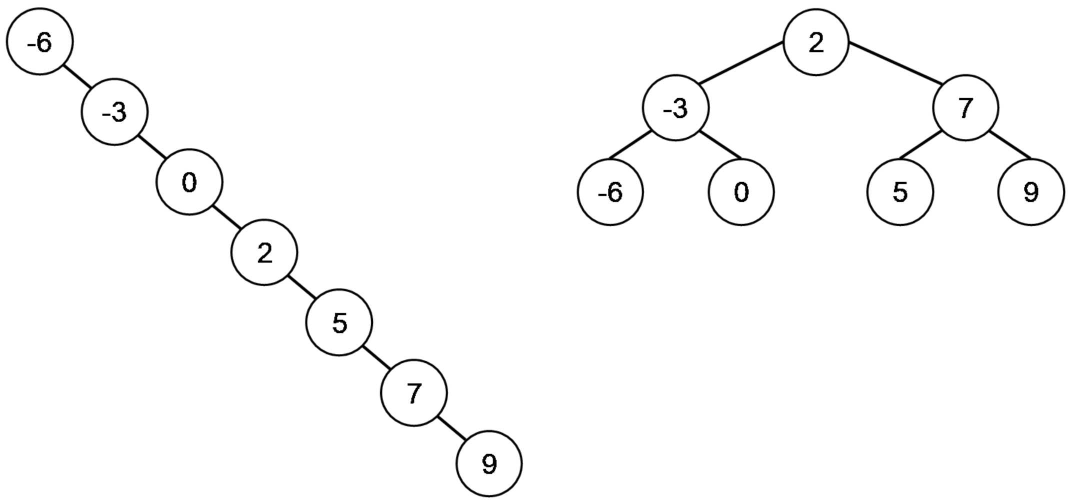
Checkpoint 0

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



Balanced search tree

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



The height of the left tree is 7, while the height of the right tree is only 3 — yet they contain the same elements! 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.

This example illustrates two things:

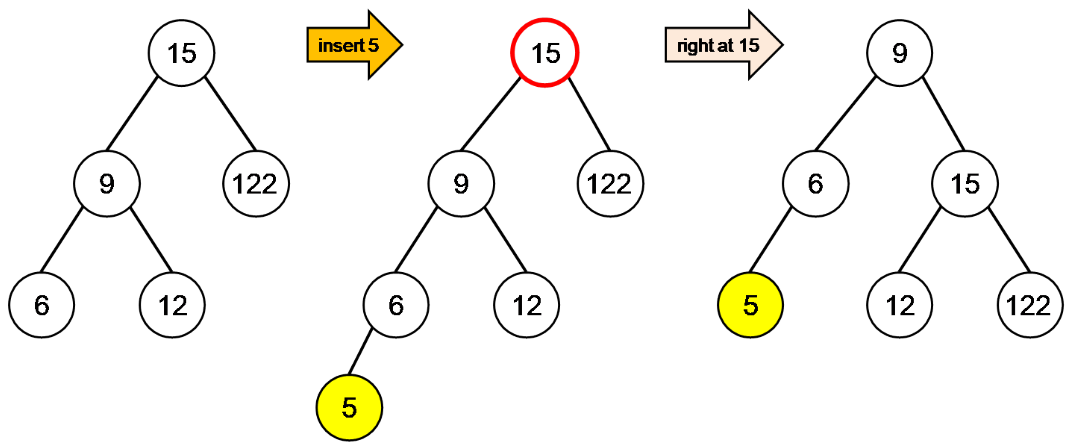
- the cost of looking up a value in a tree is proportional to its height, and
- when looking up a value in an n-element tree, this operation will be faster on shorter trees than on taller trees;

For this reason we are particularly interested in keeping trees short as we insert elements. An n-element tree of height h is *balanced* if $h \in O(\log n)$. We will next explore one type of balanced tree.

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.

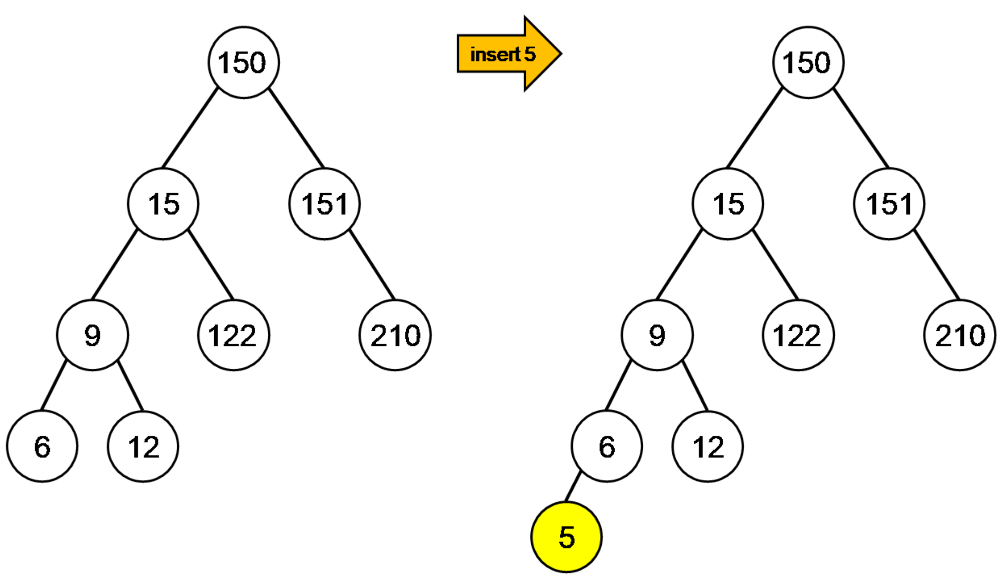
We insert nodes just as we would with a plain BST, but then check to see if (and where) the height invariant is violated. Say we insert 5 into the following tree:



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.

Checkpoint 1

Verify that the (bigger) tree on the left below is an AVL tree. Notice that the subtree rooted at node 15 is exactly the tree we started with in the previous example. Let's now insert 5 into it. How many violations are there on the tree on the right? What happens if you apply the exact same rotation on node 15 as in the previous example?



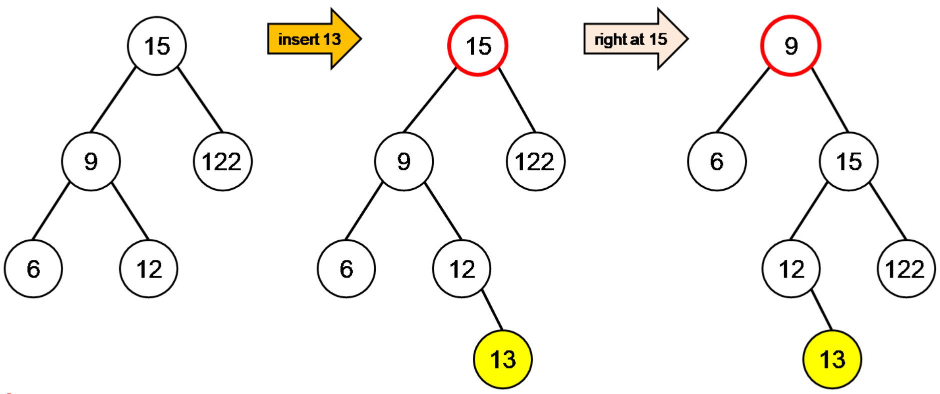
Checkpoint 2

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

```
tree* rotate_right(tree* T)
//@requires is_tree(T);
//@requires T != NULL && T->left !=NULL;
//@ensures is_tree(\result);
{

  tree* L = ______________________________

  ________________________________________

  ________________________________________

  ________________________________________

}
```

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

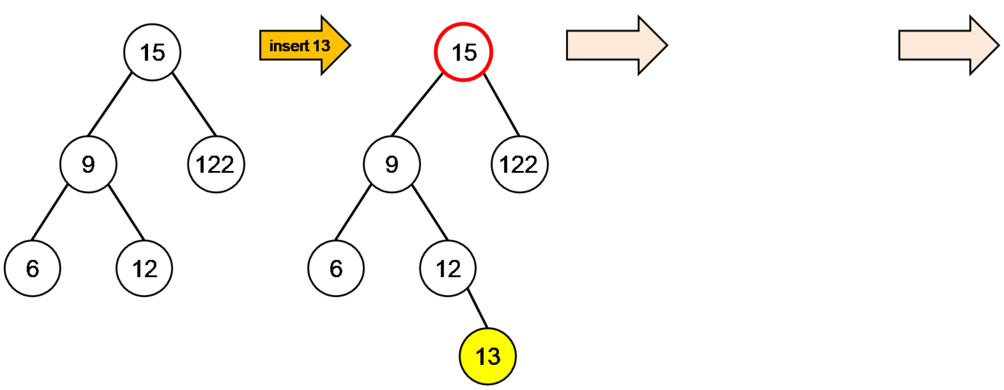
However, sometimes a single rotation is insufficient to rebalance an AVL tree and we need to perform two rotations. Consider inserting 13 into the following tree. Once again our tree is unbalanced at node 15. However, if we rotate right as in the previous example, the tree is still unbalanced!



Checkpoint 3

What two rotations can we perform that will rebalance the above tree? Draw the resulting tree.

____



Checkpoint 4

Let's implement double rotations! Using `rotate_right` and `rotate_left`, implement the following function which carries out the transformation you just used.

```
tree* rotate_left_right(tree* T)
//@requires is_tree(T);
//@requires ______________________________________________________________;
//@ensures is_tree(\result);
{
  ______________________________________________________________________

  ______________________________________________________________________
  return T;
}
```

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

Checkpoint 5

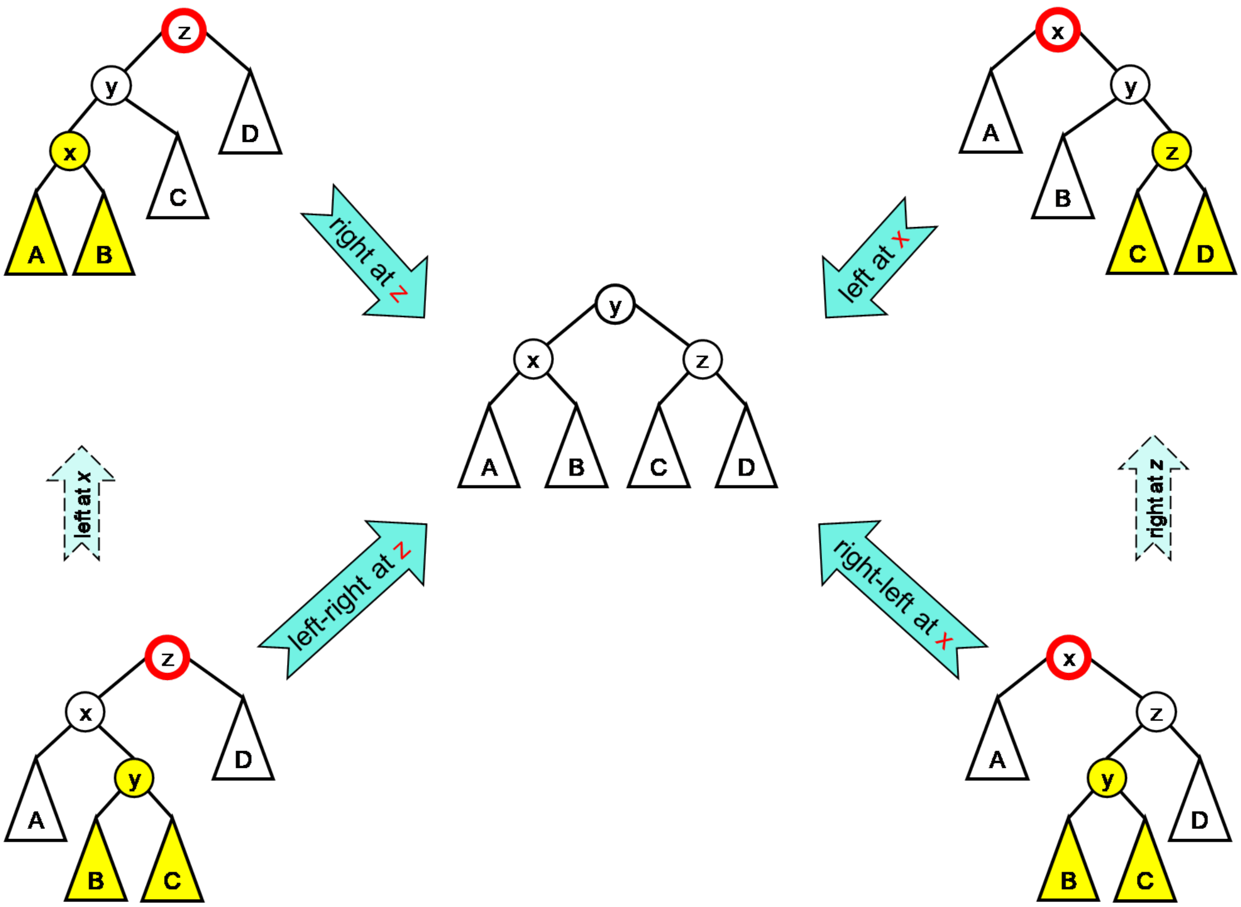
In general, in what situations is only one rotation necessary? In what situation do we need two rotations?

____

Use the visualization at http://www.cs.usfca.edu/~galles/visualization/AVLtree.html to insert $1, 2, 5, 3, 4$ into an initially empty tree in the given order.

Appendix: AVL Rotations

The diagram below summarizes the four AVL rotations. Each of them is performed when a new node is inserted in the subtree highlighted in yellow and this causes a height violation in the node highlighted in red — this must be the lowest violation.



A double rotation can be understood (and implemented!) as two single rotations performed in sequence, as indicated by the vertical arrows.