A8: Binary Search Trees and AVL Trees

Due October 28th

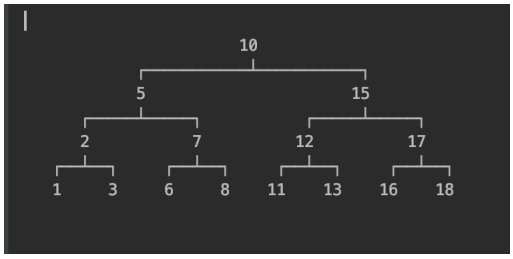
1 Starter

You are given the following starter files:

- `TreePrinter.java` which includes a utility to print trees;
- `Pair.java` which contains a very simple class that can represent any pair of objects;
- `BST.java` which contains the skeleton of an implementation of binary search trees;
- `AVL.java` which contains the skeleton of an implementation of AVL trees;
- `BSTTest.java` and `AVLTest.java` which contains a few test cases to get you started. As usual, you should write your own test cases.

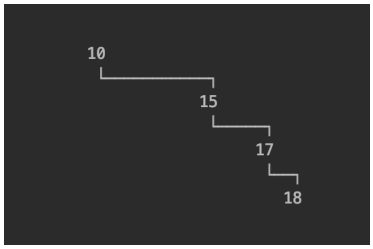
2 Binary Search Trees

These are binary trees that satisfy the following *order property*: all the nodes to the left of a root are smaller than it, and all the nodes to the right of a root are larger than or equal to it. Here is an ideal case:



We can determine if a given number is in the tree or not using a maximum of four comparisons, i.e., using $O(\log n)$ comparisons.

But binary search trees make no guarantees about being balance. The following is also a binary search tree:



In the latter tree it might take $O(n)$ comparisons to search for an element.

We will start by implementing binary search trees with no concerns about balance. Here are some important points for you to consider:

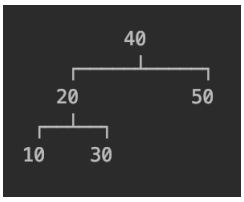
- our implementation will be a *persistent* implementation. Insertions and deletions return *new* trees: they do not modify the existing tree. Thus we will always write: `bst = bst.BSTinsert(5);` etc.
- compute the height as part of the constructor and store it in an instance variable. This will ensure that `BSTHeight()` is an $O(1)$ operation.
- inserting into a BST follows a simple strategy: if the value to be inserted is smaller than the root, recursively insert in the left child; otherwise insert in the right child.
- deleting from a BST is done as follows: if the value to be deleted is smaller than the root, recursively delete from the left child; if it is larger than the root, recursively delete from the right child; otherwise we must delete the root. To do that, we will the leftmost value in the right subtree and move it to the root.
- producing an iterator that returns the nodes using an *in-order traversal*. The main insight needed is to keep quite a bit of information in instance variables that every call to `next()` uses and updates.

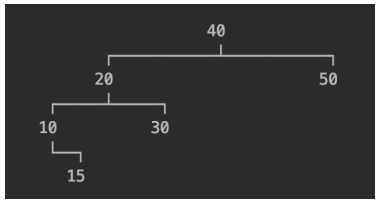
3 AVL Trees

We will call a tree *balanced* when the height of any two siblings never differs by more than 1.

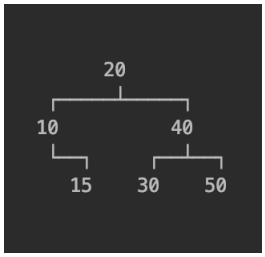
3.1 Insertion

In order to maintain the balanced condition, we reason as follows. Assume the tree is balanced before an insertion, i.e., no two siblings differ in height by more than 1. A single insertion might increase the height of one particular subtree. If we are unlucky, that tree was already 1-higher than its sibling and the new insertion causes to be 2-higher. For example, if we insert 15 in the balanced tree displayed on the left, we get the unbalanced tree displayed on the right:

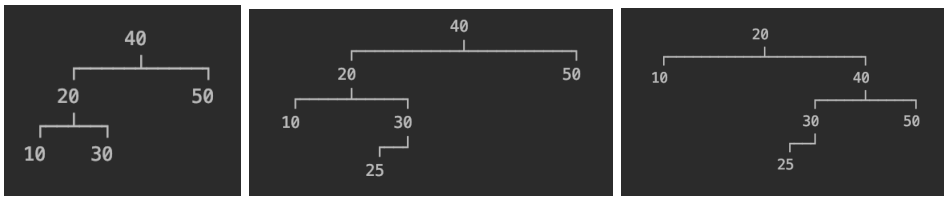




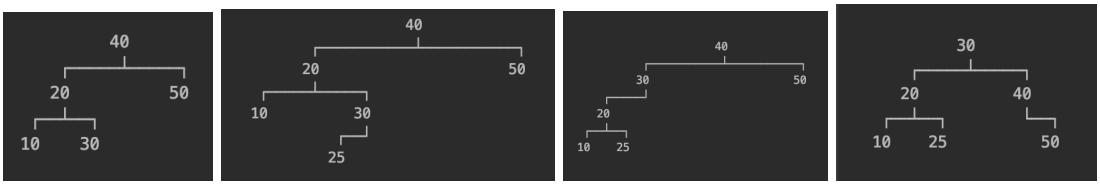
It is quite easy to detect the unbalance by comparing heights. It is also straightforward to restore the balance by doing what we call an `easyRight` rotation which takes the "heavy" tree on the left and slides it to the root:



There is actually an even more unlucky case. Consider the same initial tree as before, and let's insert 25, and follow it with an `easyRight` rotation:



The result is however unbalanced. It turns out that, before doing an `easyRight` rotation of the entire tree, we need to an `easyLeft` rotation of the left subtree, and follows that with an `easyRight` rotation:



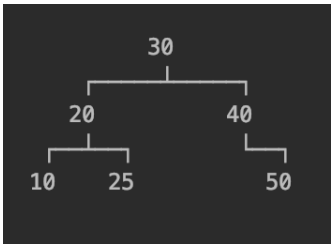
It is remarkable that this is all that is needed and that this is guaranteed to maintain balance while inserting. To summarize, here is what AVL insertion does:

- Start with the template for BST insertion;
- Assume the insertion happens in the left subtree (the other situation is symmetric);
- Compare the height of the new left subtree to the height of the right subtree;
- If the tree is balanced, we are done
- If the tree is unbalanced, perform `rotateRight`
- A `rotateRight` action on a tree focuses on the left subtree first and compares the heights of its children: if the left child is higher, just do an `easyRight` action; otherwise first do an `easyLeft` action on the left child and then an `easyRight` action on the whole tree.

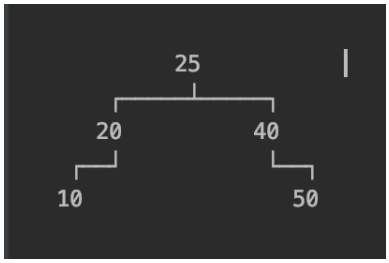
3.2 Deletion

Deleting a node from an AVL has two easy cases and one difficult case. First the easy cases. Assume we want to delete a node that happens to be in the left subtree (again the other situation is symmetric). We simply make a recursive call on the left subtree, check if the new height is too short, and do the appropriate rotation in that case (`rotateLeft`).

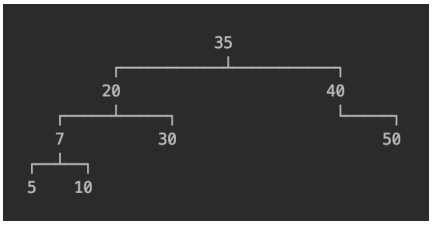
The difficult case is when we need to delete the root of a tree. For example, deleting 30 from the following tree:



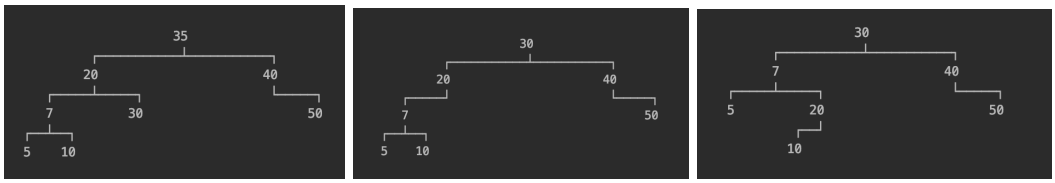
We find the largest node in the left subtree (25 in this case) and move it to the root:



But this may not work so well. Consider deleting 35 from the following tree:



As before, we find the largest node in the left subtree (30), delete it from its position, and put it at the root. However, remove 30 from its position leaves the tree rooted at 20 unbalanced. So we must rotate it.



To summarize, deletion in AVL trees performs the following actions:

- If we are at a node with two subtrees and the deletion happens in one of the subtrees, we simply check if the resulting tree is balanced and if not perform the appropriate rotation;
- If we are at a node with two subtrees and we want to delete the root, then we first invoke a helper called `shrink` on the left subtree. The result of `shrink` will be a pair whose first element is the largest node in the left subtree and whose second component is the (balanced) remainder of the left subtree after deleting that largest node.
- The result is a new tree whose root is the first component returned by `shrink`, whose left subtree is the second component returned by `shrink`, and whose right subtree is the original right subtree. If that result is not balanced, it must rotated, however.
- The strategy for implementing `shrink` is somewhat similar to `deleteLeftMostLeaf` except that we need to check for unbalanced trees and perform appropriate rotations at every recursive call.