AVL Trees

Step 1

We learned about the **Binary Search Tree**, which we formally proved has an **average-case** time complexity of **O(log *n*)** if we were to make unrealistic assumptions. We then discussed an improved data structure, the **Randomized Search Tree**, which is actually able to obtain this **O(log *n*) average-case** time complexity without the need for unrealistic assumptions. Nevertheless, in the **worst case**, both structures have an **O(*n*)** time complexity. Can we squeeze even more speed out of our trees?

In 1962, two Soviet computer scientists, Georgy Adelson-Velsky and Evgenii Landis, forever changed the landscape of **Binary Search Tree** structures when they created a revolutionary self-balancing tree that achieves a **worst-case** time complexity of **O(log *n*)**. They named this new data structure after themselves, the **A**delson-**V**elsky and **L**andis tree, or as we know it, the **AVL Tree**.

Figure: Georgy Adelson-Velsky (left) playing chess against John McCarthy (right), the creator of the Lisp programming language

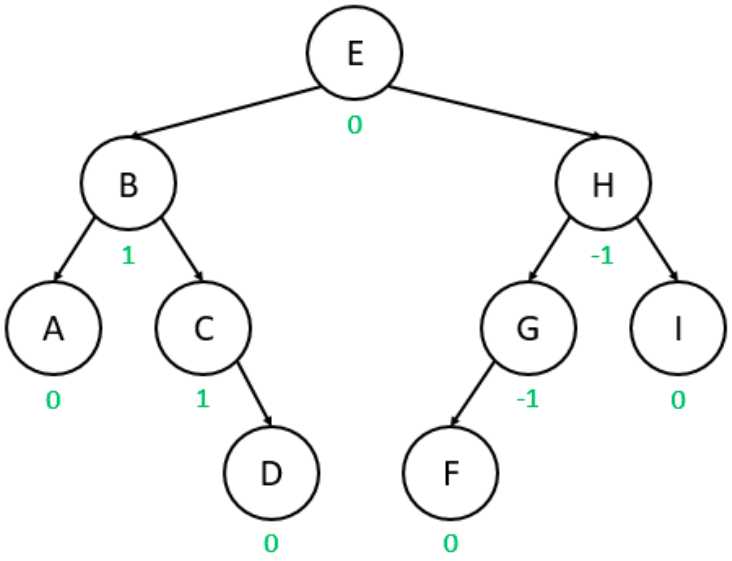
Step 2

An **AVL Tree** is a **Binary Search Tree** in which, for all nodes in the tree, the *heights* of the two child subtrees of the node differ by at most one. If, at any time, the two subtree heights differ by more than one, rebalancing is done to restore this property.

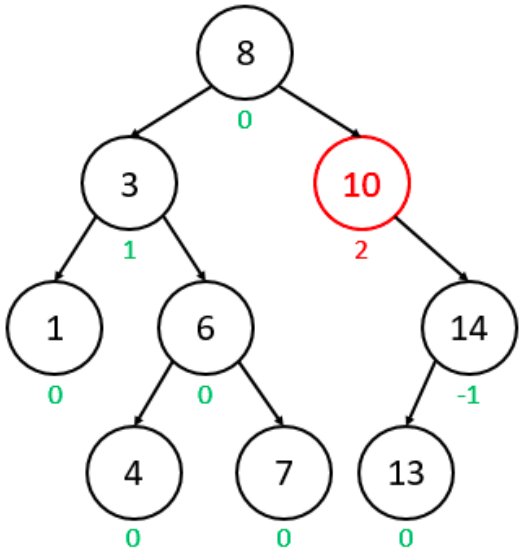
In order to analyze the **AVL Tree**, we will first provide some formal definitions:

- The **balance factor** of a node *u* is equal to the height of *u*'s right subtree minus the height of *u*'s left subtree
- A **Binary Search Tree** is an **AVL Tree** if, for all nodes *u* in the tree, the balance factor of *u* is either -1, 0, or 1

Below is an example of an **AVL Tree**, where each node has been labeled with its balance factor. Note that all nodes have a balance factor of -1, 0, or 1.

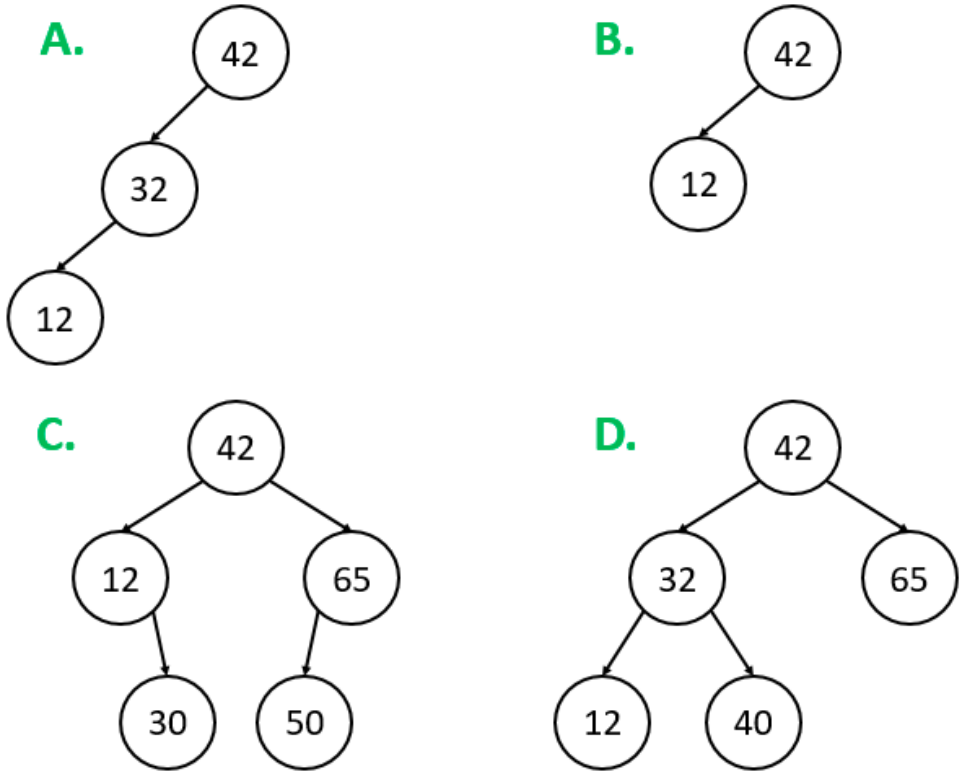


Below, however, is a **Binary Search Tree** that *seems* pretty balanced, but it is in fact **not** an **AVL Tree**: there exists a node (node 10) has an invalid balance factor (2).



Step 3

EXERCISE BREAK: Which of the following trees are valid **AVL Trees**? (Select all that apply)



To solve this problem please visit https://stepik.org/lesson/28865/step/3

Step 4

It should be intuitive that the **AVL Tree** restriction on balance factors keeps the tree *pretty* balanced, but as wary Computer Scientists, we have been trained to trust nobody. Can we formally prove that the height of an **AVL Tree** is O(log *n*) in the worst-case? Luckily for us, this proof is *far* simpler than the **Binary Search Tree** average-case proof we covered earlier in this text.

First, let us define N_h as the minimum number of nodes that can form an **AVL Tree** of height *h*. It turns out that we can come up with a somewhat intuitive recursive definition for N_h. Specifically, because an **AVL Tree** with N_h nodes must have a height of *h* (by definition), the root must have a child that has height *h*-1. To minimize the total number of nodes in the tree, this subtree would have N_{h-1} nodes.

By the property of an **AVL Tree**, because the root can only have a balance factor of -1, 0, or 1, the other child has a minimum height of *h*-2. Thus, by constructing an **AVL Tree** where the root's left subtree contains N_{h-1} nodes and the root's right subtree contains N_{h-2} trees, we will have constructed the **AVL Tree** of height *h* with the least number of nodes possible. In other words, we have defined a recurrence relation: $N_h = N_{h-1} + N_{h-2} + 1$

Now, we can define base cases. When we defined *height* in the **Binary Search Tree** section of this text, we defined the height of an empty tree as -1, the height of a one-node tree as 0, etc. For the purposes of this recurrence, however, let's temporarily change our definition such that an empty tree has a height of 0, a one-node tree has a height of 1, etc. This change in numbering only adds 1 to the height of any given tree (which does not affect Big-O time complexity), but it simplifies the math in our proof. Using this numbering, we can define two trivial base cases. $N_1 = 1$ because a tree of height *h* = 1 is by definition a one-node tree. $N_2 = 2$ because the smallest number of nodes to have a tree with a height of 2 is 2: a root and a single child.

Now that we have defined a recurrence relation ($N_h = N_{h-1} + N_{h-2} + 1$) and two base cases ($N_1 = 1$ and $N_2 = 2$), we can reduce:

- $N_h = N_{h-1} + N_{h-2} + 1$
- $N_{h-1} = N_{h-2} + N_{h-3} + 1$
- $N_h = (N_{h-2} + N_{h-3} + 1) + N_{h-2} + 1$

- $N_h > 2N_{h-2}$
- $N_h > 2^{\frac{h}{2}}$
- $\log N_h > \log 2^{\frac{h}{2}}$
- $2 \log N_h > h$
- $h = O(\log N_h)$

We have now formally proven that an **AVL Tree** containing *n* nodes has a height that is indeed O(log *n*), even in the worst-case. As a result, we have formally proven that an **AVL Tree** has a **worst-case** time complexity of **O(log *n*)**.

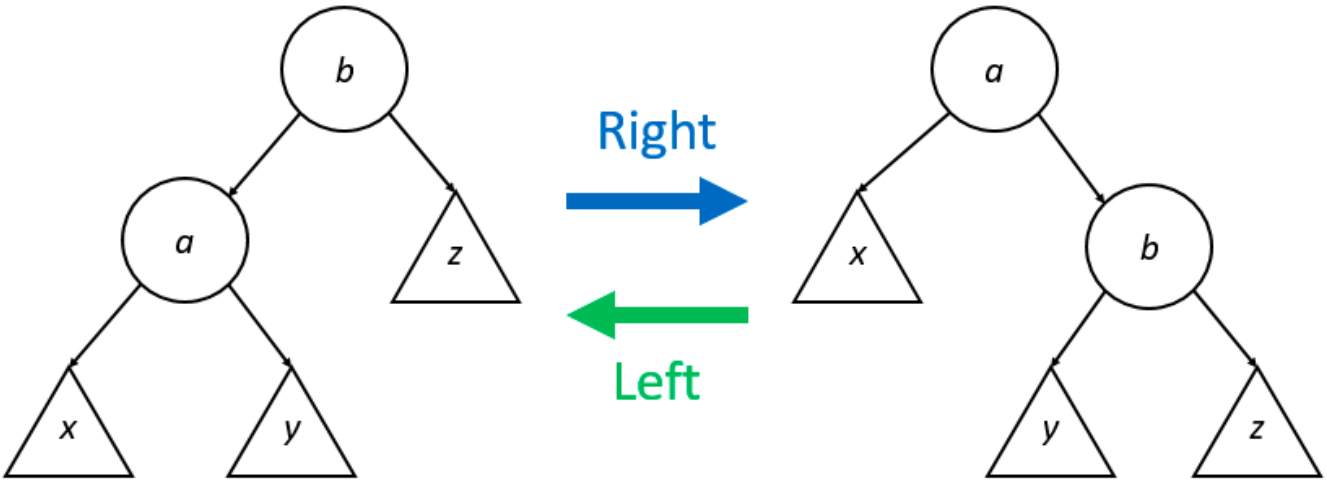
Step 5

We have formally proven that, given an **AVL Tree**, the worst-case time complexity for finding, inserting, or removing an element is O(log *n*) because the height of the tree is at worst O(log *n*). Now, let's discuss the algorithms for these three operations that are able to *maintain* the **AVL Tree** properties without worsening the time complexity.

Finding an element in an **AVL Tree** is trivial: it is simply the **Binary Search Tree** "find" algorithm, which was discussed in detail in the **Binary Search Tree** section of this text.

Before discussing the insertion and removal algorithms of an **AVL Tree**, we want to first refresh your memory with regard to a fundamental operation known as the **AVL Rotation**. AVL rotations were introduced previously in the **Randomized Search Tree** section of this text, but since they are somewhat non-trivial and are so important to **AVL Trees** (hence the name "AVL" rotation), we thought it would be best to review them.

Recall that **AVL Rotations** can be done in two directions: **right** or **left**. Below is a diagram generalizing both right and left AVL Rotations. In the diagram, the triangles represent arbitrary subtrees of any shape: they can be empty, small, large, etc. The circles are the "important" nodes upon which we are performing the rotation.



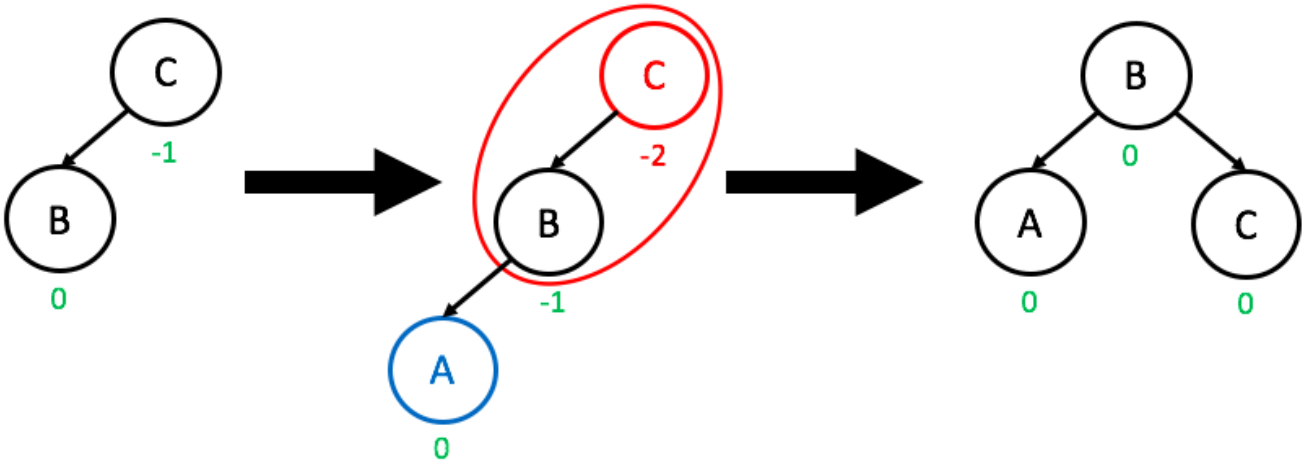
Note that the *y* subtree *was* the right child of node *a* but now becomes the *left* child of node *b*!

Step 6

The **AVL Tree** insertion and removal algorithms rely heavily on the **Binary Search Tree** insertion and removal algorithms, so if you're unsure about either of them, be sure to reread the **BST** section of this text to refresh your memory.

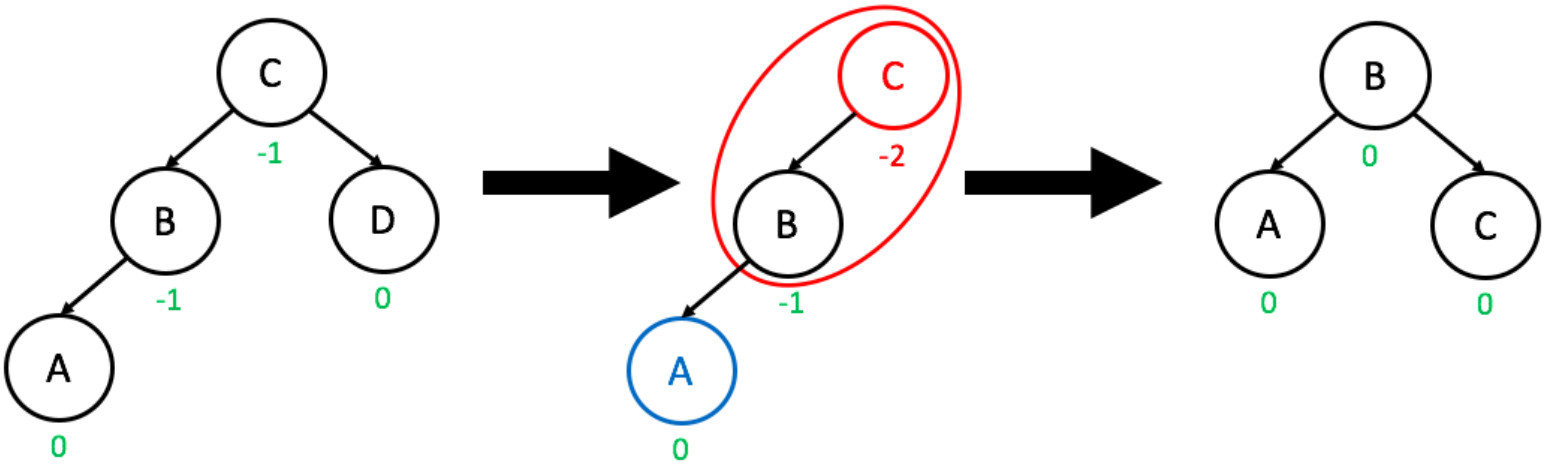
To insert an element into an **AVL Tree**, first perform the regular **BST** insertion algorithm. Then, starting at the newly-inserted node, traverse up the tree and update the balance factor of each of the new node's ancestors. For any nodes that are now "out of balance" (i.e., their balance factor is now less than -1 or greater than 1), perform AVL rotations to fix the balance.

Below is an example of an insertion into an **AVL Tree**, where the element A is inserted:



To remove an element from an **AVL Tree**, first perform the regular **BST** removal algorithm. Then, starting at the parent of whatever node was actually removed from the tree, traverse up the tree and update each node's balance factor. For any nodes that are now "out of balance," perform AVLs rotation to fix the balance.

Below is an example of a removal from an **AVL Tree**, where the element D is removed:

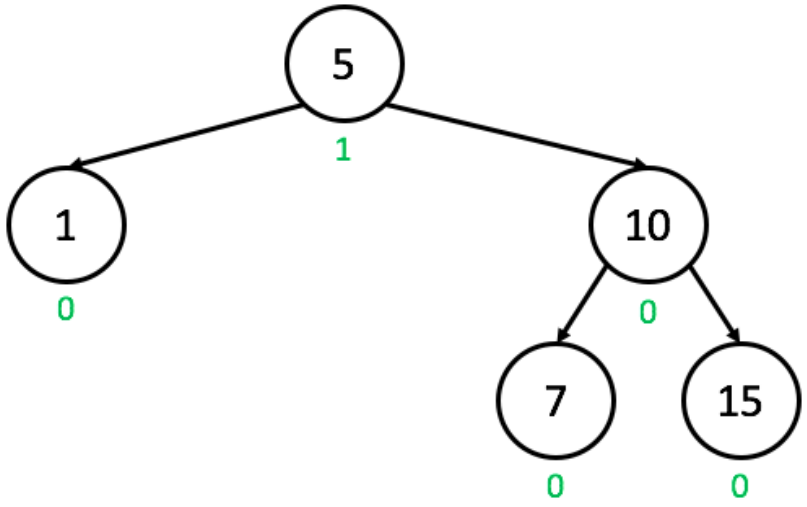


As can be seen, the **AVL Tree** rebalances itself after an insertion or a removal by simply traversing back up the tree and performing AVL rotations along the way to fix any cases where a node has an invalid balance factor. Because of this phenomenon where the tree balances itself without any user guidance, we call **AVL Trees** "self-balancing".

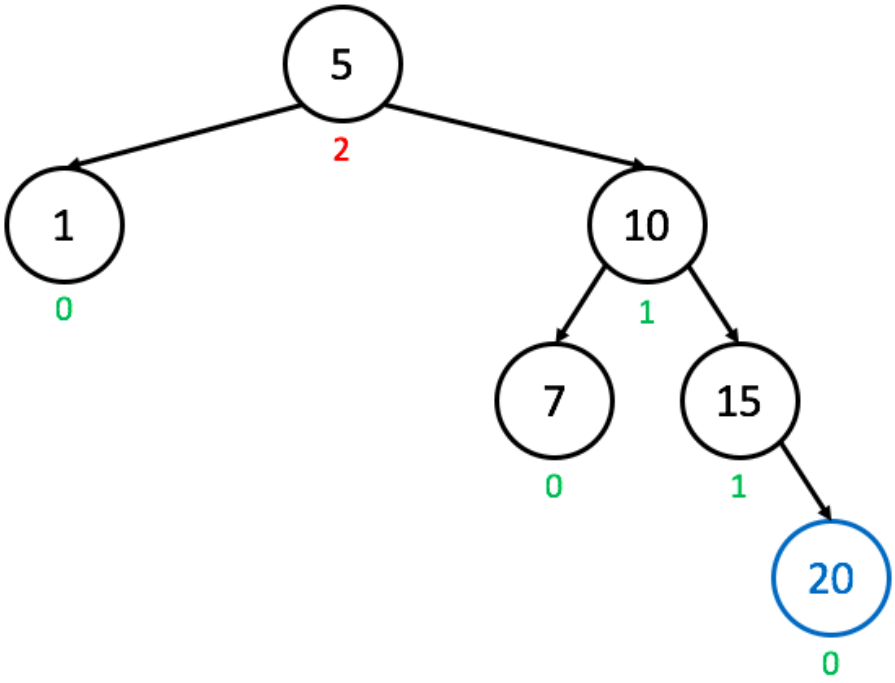
STOP and Think: Can you think of an example where a single AVL rotation will fail to fix the balance of the tree?

Step 7

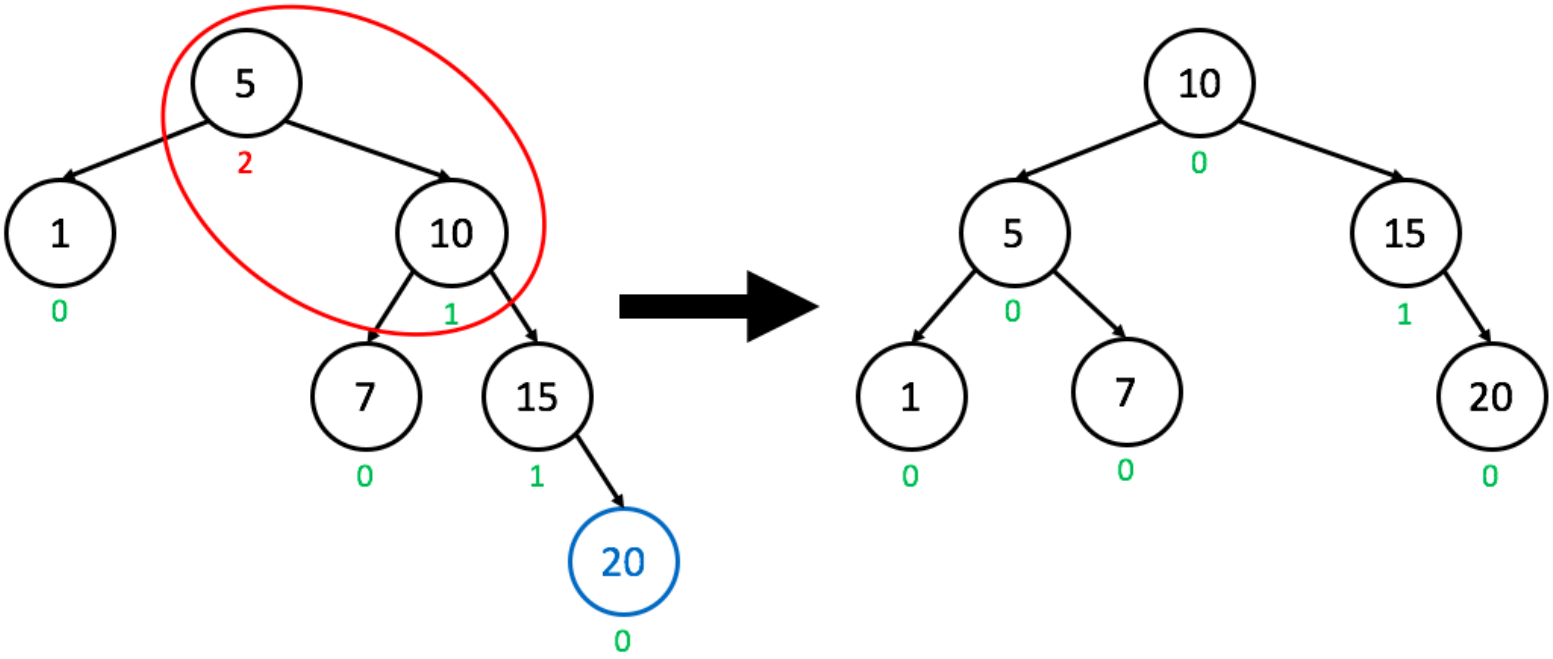
Below is a more complex example in which we insert 20 into the following **AVL Tree**:



Following the traditional **Binary Search Tree** insertion algorithm, we would get the following tree (note that we have updated balance factors as well):

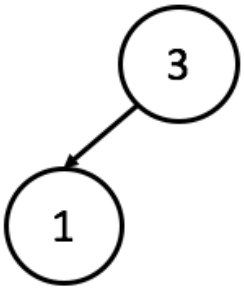


As you can see, the root node is now out of balance, and as a result, we need to fix its balance using an AVL rotation. Specifically, we need to rotate left at the root, meaning 10 will become the new root, 7 will become the right child of 5, and 5 will become the left child of 10:



Step 8

EXERCISE BREAK: If we were to insert 2 into the following **AVL Tree**, which node would go out of balance? (Please enter the number that labels the node, and nothing else)

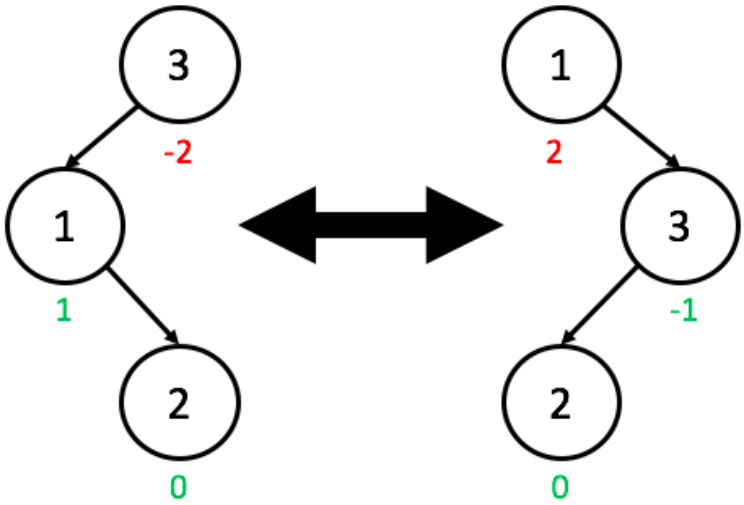


STOP and Think: Will we be able to fix this tree the same way we fixed the trees in the previous step?

To solve this problem please visit https://stepik.org/lesson/28865/step/8

Step 9

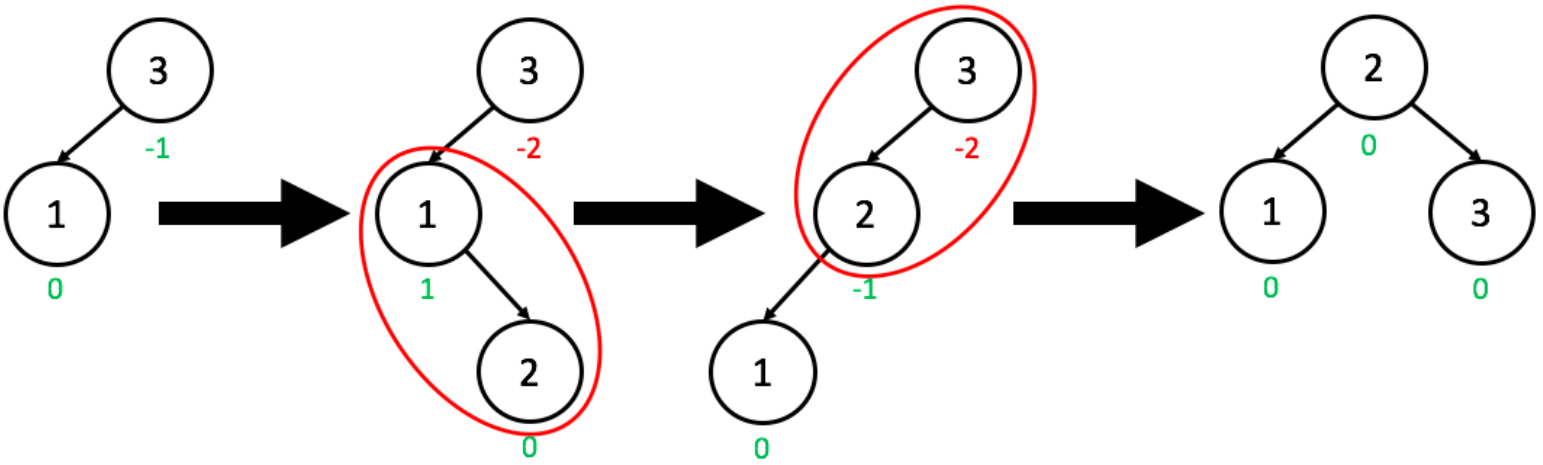
As you may have noticed in the previous step, the simple AVL rotation we learned fails to fix the balance of the tree:



To combat this problem, we will introduce the **double rotation**, which is simply a series of two AVL rotations that will be able to fix the tree's balance. The issue comes about when we have a "kink" in the shape with respect to what exactly went out of balance. Contrast the above tree with the trees on the previous step, which were also out of balance, but which looked like a "straight line" as opposed to a "kink".

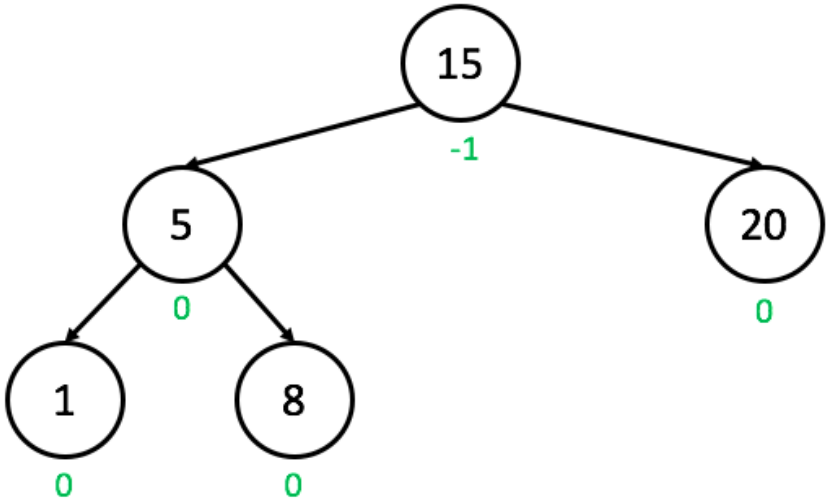
The solution to our problem (a double rotation) is actually quite intuitive: we have a method that works on nodes that are in a straight line (a single rotation), so our solution to kinks is to first perform one rotation to *transform* our problematic kink *into* the straight line case, and then we can simply perform another rotation to solve the straight line case.

Below is the correct way to rebalance this tree (where 2 is the node we inserted into the tree):

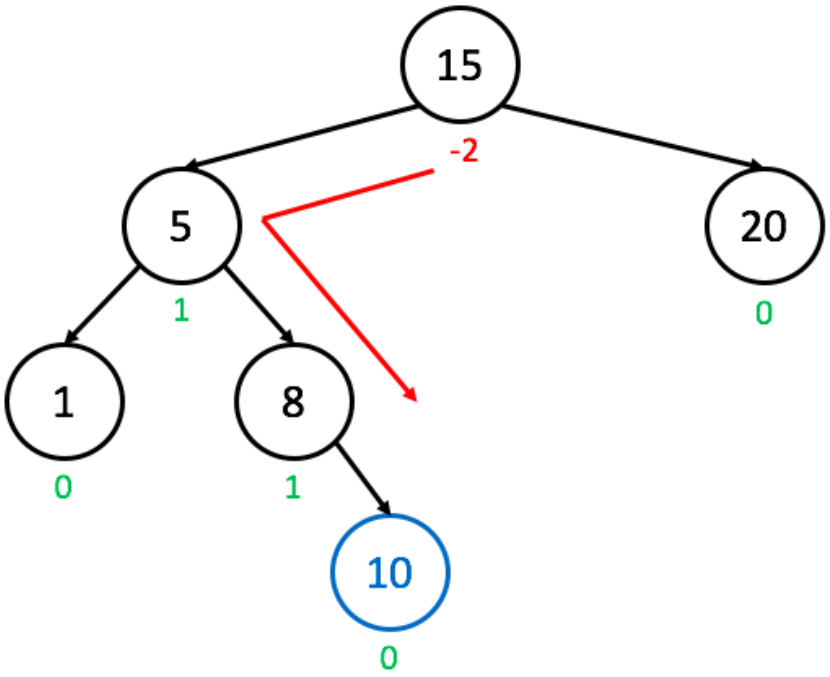


Step 10

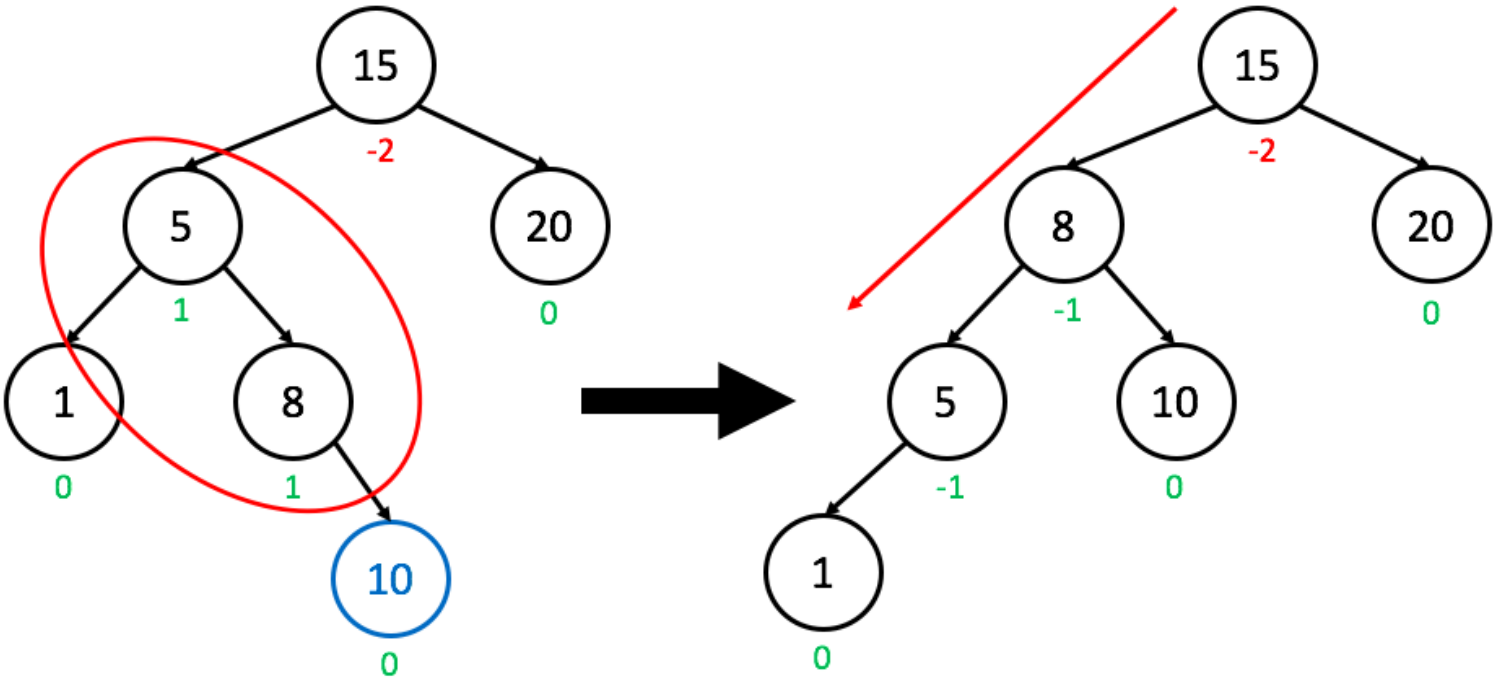
Below is a more complex example in which we insert 10 into the following **AVL Tree**:



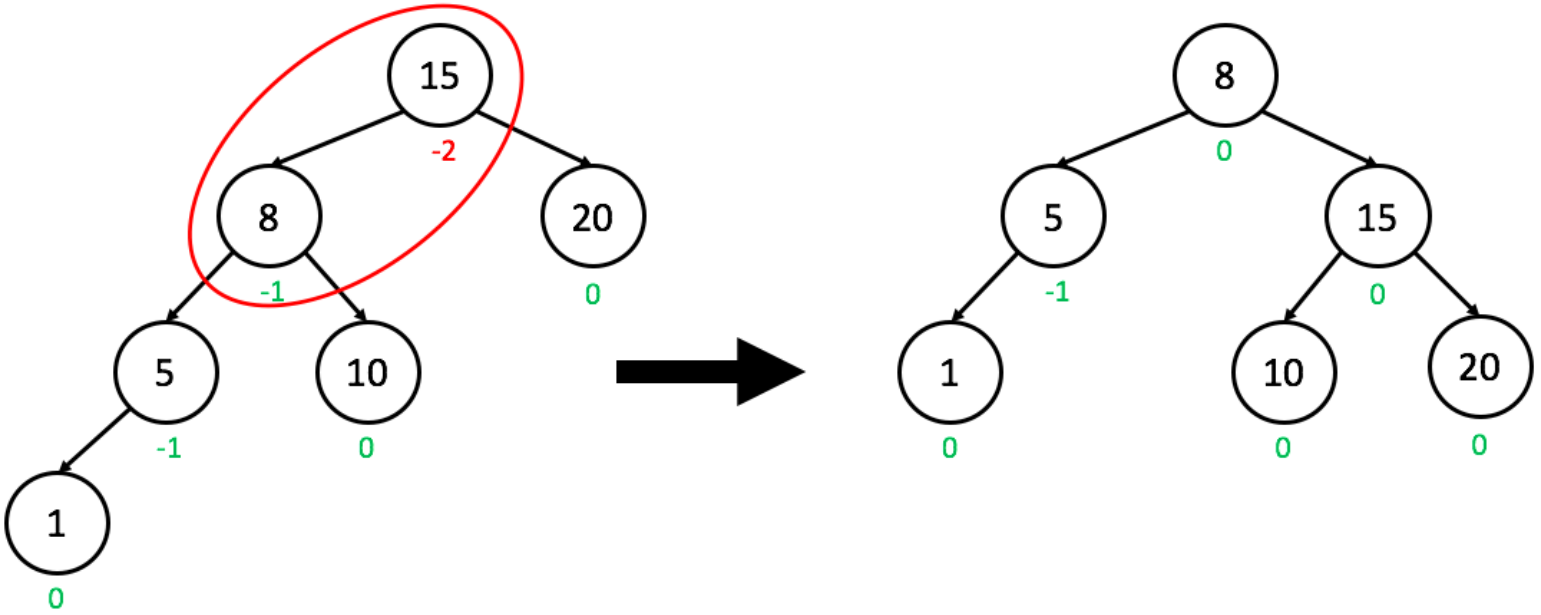
Following the traditional **Binary Search Tree** insertion algorithm, we would get the following tree (note that we have updated balance factors as well):



As you can see, the root node is now out of balance, and as a result, we need to fix its balance. However, as we highlighted via the bent red arrow above, this time, our issue is in a "kink" shape (not a "straight line" shape, as it was in the previous complex example). As a result, a single AVL rotation will no longer suffice, and we are forced to perform a double rotation. The first rotation will be a left rotation on node 5 in order to transform this "kink" shape into a "straight line" shape:

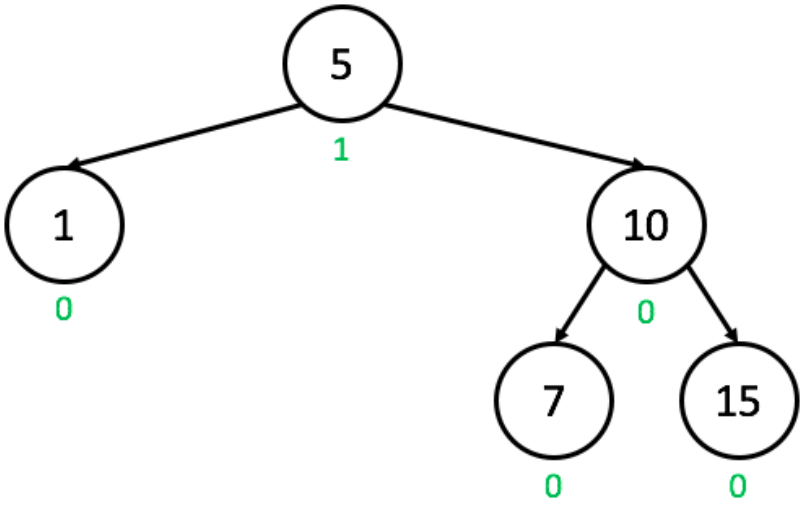


Now, we have successfully transformed our problematic "kink" shape into the "straight line" shape we already know how to fix. Thus, we can perform a right rotation on the root to fix our tree:



Step 11

EXERCISE BREAK: If we were to insert 6 into the following **AVL Tree**, after performing the required rotation(s) to fix the tree's balance, which node would be the new root? (Please enter the number that labels the node, and nothing else)



To solve this problem please visit https://stepik.org/lesson/28865/step/11

Step 12

Below is a visualization of an **AVL Tree**, created by David Galles at the University of San Francisco. Note that the visualization tool performs double rotations in a single step instead of breaking it down into two individual single rotations as we did, so it may seem a tad confusing initially.

Step 13

We have now discussed the **AVL Tree**, which is the first ever **Binary Search Tree** structure able to rebalance itself automatically in order to guarantee a **worst-case** time complexity of **O(log *n*)** for finding, inserting, and removing elements. Given that a perfectly balanced **Binary Search Tree** has a worst-case O(log *n*) time complexity, you might be thinking "Wow, our new data structure is pretty darn fast! We can just call it a day and move on with our lives." However, it is known fact that a Computer Scientist's hunger for speed is insatiable, and in our hearts, we want to go *even faster*!

Because of the realization above about perfectly balanced trees, it is clear that, given that we are dealing with **Binary Search Trees**, we cannot improve the worst-case *time complexity* further than O(log *n*). However, note that, in the worst case, an **AVL Tree** must perform roughly 2*log(*n*) operations when inserting or removing an element: it must perform roughly log(*n*) operations to traverse *down* the tree, and it must then perform roughly log(*n*) operations to traverse back *up* the tree in order to restructure the tree to maintain balance. In the next section, we will discuss the **Red-Black Tree**, which guarantees the same O(log *n*) worst-case time complexity, but which only needs to perform a *single* pass through the tree (as opposed to the two passes of an **AVL Tree**), resulting in an even *faster* in-practice run-time for insertion and removal.