

# AVL Tree Rotation

---

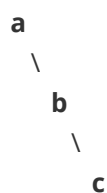
Written and compiled by *Trontor*, with special [thanks to John Hargrove for his tutorial](#), of which this explanation was derived from.

This concept of rotation in AVL's may seem intimidating, and it can get confusing very fast. The following may seem needlessly long to a few, but for my own understanding of the entire process I will delve through the process step by step.

## Identify a Need for Rotation

---

Imagine we have such a tree:



With the following balance factors

$$\mathbf{a} = 2 - 0 = 2$$

$$\mathbf{b} = 1 - 0 = 1$$

$$\mathbf{c} = 0 - 0 = 0$$

As we can see, node **a** has an imbalance out of the acceptable range  $(-1, 0, 1)$ .

We have an **unbalanced state** and therefore need a rotation.

Simple.

## Determining Rotation Type

---

This is the harder part. In the previous example, we see that node **a** has a balance factor of **2**. If we think back to the balance equation:

$$\text{Balance\_Factor} = \text{Height}(\text{Right\_Subtree}) - \text{Height}(\text{Left\_Subtree})$$

Since  $\text{Height}()$  will always be positive, a balance factor of **+2** means that:

$$\text{Height}(\text{Right\_Subtree}) > \text{Height}(\text{Left\_Subtree})$$

This means that the tree is **right heavy**. Logically, to fix a right heavy tree, we can perform a **left rotation**!

Trivially, a **left heavy** tree requires a **right rotation**!

## Single Left Rotation

Back to the example, with example numbers this time:

**a** = 9, *Balance* = 2 - 0 = **+2**

\

**b** = 14, *Balance* = 1 - 0 = **1**

\

**c** = 200, *Balance* = 0

- **b** becomes the new root
- **a** takes **b's** *left* child and makes it its right child (in this case, null)
- **b** makes **a** its *left* child

**b** = 14

/   \

**a** = 9   **c** = 200

A simple sanity check is to verify that order has been preserved!

## Single Right Rotation

Mirror of left rotation:

**c** = 200, *Balance* = 0 - 2 = **-2 !!!**

/

**b** = 14, *Balance* = 0 - 1 = **-1**

/

**a** = 9, *Balance* = 0

Category: **left heavy**

Fix: **right rotation**

- **b** becomes new root
- **c** takes **b's** *right* child, and makes it its left child (in this case, null)
- **b** makes **c** its *right* child

**b** = 14

/   \

**a** = 9   **c** = 200

## How can I identify this need just from observation?

If we look at the case of a single rotation, we can see the balance signs match. That is, for a right heavy tree **(+2, +1)** and a left heavy tree **(-2, -1)**.

When we need a double rotation, the balance factor signs do not match! We will see in the next example, that the balance factor pair is **(+2, -1)**.

However, once we rotate the subtree, the signs start to match **(+2, +1)**! Then, we can perform a final complete tree rotation.

## Double Rotations, huh?

---

Observe the following tree:

```
a = 9, Balance = 2 - 0 = +2
 \
  c = 14, Balance = 0 - 1 = -1
 /
b = 12, Balance = 0
```

This is obviously *right heavy*, so lets do a *left rotation*.

- **c** becomes the new root
- **a** takes **c's** *left* child, and makes it its right child (in this case, **b**)
- **c** makes **a** its *left child*

Result:

```
      c = 14 (-2)
     /
    a = 9 (+1)
     \
      b = 12
```

???

Unbalanced. Doing a right rotation on this result will get us back to where we started.

We need a **double rotation**.

# Performing a Double Rotation

---

## Left-Right Rotation

To fix the aforementioned issue we first apply a **right rotation** on the right subtree then a **left rotation**.

Given this tree:

```
a = 9
 \
  c = 14
 /
b = 12
```

Right Subtree is:

```
  c = 14
 /
b = 12
```

We perform a **right rotation** on this subtree:

```
b = 12
 \
  c = 14
```

Now the whole tree looks like:

```
a = 9, Balance = +2
 \
  b = 12, Balance = +1
   \
    c = 14
```

Notice how the signs match now? Lets now perform a final **left rotation**, as it is **right heavy**

```
  b = 12, Balance = 0
 /  \
a = 9 c = 14
```

Perfect.

Onto the next one (zzz)

## Right-Left Rotation

Mirror of the previous Left-Right rotation. Let's speed through this.

```
      c = 14 (-2)
     /
    a = 9 (+1)
     \
      b = 12
```

Different signs, **double rotate**. Since it is **left heavy** we first do a **left rotation** on the **left** subtree.

*Left subtree:*

```
a
 \
  b
```

*Rotated:*

```
  b
 /
a
```

*Tree updated:*

```
      c = 14 (-2)
     /
    b = 12 (-1)
   /
  a = 9
```

Signs match, still **left heavy**, so **right rotate**:

- **b** becomes new root
- **c** takes **b**'s right child and makes it its left child
- **b** makes **a** its left child

```
  b = 12
 /  \
a    c = 14
```

This concept is hard enough to do conceptually, but implementing a recursive balancing algorithm that handles these 4 cases is harder.

# Overall Procedure

---

Here is some pseudocode to keep in mind:

```
IF tree is right heavy
{
    IF tree's right subtree is left heavy
    {
        Perform Double Left rotation
    }
    ELSE
    {
        Perform Single Left rotation
    }
}
ELSE IF tree is left heavy
{
    IF tree's left subtree is right heavy
    {
        Perform Double Right rotation
    }
    ELSE
    {
        Perform Single Right rotation
    }
}
```