The AVL Tree Rotations Tutorial

By John Hargrove

Version 1.0.1, Updated Mar-22-2007

Abstract

I wrote this document in an effort to cover what I consider to be a dark area of the AVL Tree concept. When presented with the task of writing an AVL tree class in Java, I was left scouring the web for useful information on how this all works. There was a lot of useful information on the wikipedia pages for AVL tree and Tree rotation. You can find links to these pages in section 4. The tree rotation page on wikipedia is lacking, I feel. The AVL tree page needs work as well, but this page is hurting badly, and at some point in the future, I will likely integrate most of this document into that page. This document covers both types of rotations, and all 4 applications of them. There is also a small section on deciding which rotations to use in different situations.

1. Rotations: How they work

A tree rotation can be an imtimidating concept at first. You end up in a situation where you're juggling nodes, and these nodes have trees attached to them, and it can all become confusing very fast. I find it helps to block out what's going on with any of the subtrees which are attached to the nodes you're fumbling with, but that can be hard.

Left Rotation (LL)

Imagine we have this situation:

```
Figure 1-1
a
 \
  b
   \
    c
```

To fix this, we must perform a left rotation, rooted at A. This is done in the following steps:

b becomes the new root.
a takes ownership of b's left child as its right child, or in this case, null.
b takes ownership of a as its left child.

The tree now looks like this:

```
Figure 1-2
  b
 / \
a   c
```

Right Rotation (RR)

A right rotation is a mirror of the left rotation operation described above. Imagine we have this situation:

```
Figure 1-3
    c
```

```
   /
  b
 /
a
```

To fix this, we will perform a single right rotation, rooted at C. This is done in the following steps:

b becomes the new root.
c takes ownership of b's right child, as its left child. In this case, that value is null.
b takes ownership of c, as it's right child.

The resulting tree:

```
Figure 1-4
  b
 / \
a   c
```

Left-Right Rotation (LR) or "Double left"

Sometimes a single left rotation is not sufficient to balance an unbalanced tree. Take this situation:

```
Figure 1-5
a
 \
  c
```

Perfect. It's balanced. Let's insert 'b'.

```
Figure 1-6
a
 \
  c
 /
b
```

Our initial reaction here is to do a single left rotation. Let's try that.

```
Figure 1-7
  c
 /
a
 \
  b
```

Our left rotation has completed, and we're stuck in the same situation. If we were to do a single right rotation in this situation, we would be right back where we started. What's causing this? The answer is that this is a result of the right subtree having a negative balance. In other words, because the right subtree was left heavy, our rotation was not sufficient. What can we do? The answer is to perform a right rotation on the right subtree. Read that again. We will perform a right rotation on the *right subtree*. We are not rotating on our current root. We are rotating on our right child. Think of our right subtree, isolated from our main tree, and perform a right rotation on it:

Before:

```
Figure 1-8
  c
 /
b
```

After:

```
Figure 1-9
b
 \
  c
```

After performing a rotation on our right subtree, we have prepared our root to be rotated left. Here is our tree now:

```
Figure 1-10
a
 \
  b
   \
    c
```

Looks like we're ready for a left rotation. Let's do that:

```
Figure 1-11
  b
 / \
a   c
```

Voila. Problem solved.

Right-Left Rotiation (RL) or "Double right"

A double right rotation, or right-left rotation, or simply RL, is a rotation that must be performed when attempting to balance a tree which has a left subtree, that is right heavy. This is a mirror operation of what was illustrated in the section on Left-Right Rotations, or double left rotations. Let's look at an example of a situation where we need to perform a Right-Left rotation.

```
Figure 1-12
  c
 /
a
 \
  b
```

In this situation, we have a tree that is unbalanced. The left subtree has a height of 2, and the right subtree has a height of 0. This makes the balance factor of our root node, c, equal to -2. What do we do? Some kind of right rotation is clearly necessary, but a single right rotation will not solve our problem. Let's try it:

```
Figure 1-13
a
 \
  c
 /
b
```

Looks like that didn't work. Now we have a tree that has a balance of 2. It would appear that we did not accomplish much. That is true. What do we do? Well, let's go back to the original tree, before we did our pointless right rotation:

```
Figure 1-14
  c
 /
a
 \
  b
```

The reason our right rotation did not work, is because the left subtree, or 'a', has a positive balance factor, and is thus right heavy. Performing a right rotation on a tree that has a left subtree that is right heavy will result in the problem we just witnessed. What do we do? The answer is to make our left subtree left-heavy. We do this by performing a left rotation our left subtree. Doing so leaves us with this situation:

```
Figure 1-15
    c
   /
  b
 /
a
```

This is a tree which can now be balanced using a single right rotation. We can now perform our right rotation rooted at C. The result:

```
Figure 1-16
  b
 / \
a   c
```

Balance at last.

2. Rotations, When to Use Them and Why

How to decide when you need a tree rotation is usually easy, but determining which type of rotation you need requires a little thought.

A tree rotation is necessary when you have inserted or deleted a node which leaves the tree in an unbalanced state. An unbalanced state is defined as a state in which any subtree has a balance factor of greater than 1, or less than -1. That is, any tree with a difference between the heights of its two subtrees greater than 1, is considered unbalanced.

This is a balanced tree:

```
Figure 2-1
  1
```

```
 / \
2   3
```

This is an unbalanced tree:

```
Figure 2-2
1
 \
  2
   \
    3
```

This tree is considered unbalanced because the root node has a balance factor of 2. That is, the right subtree of 1 has a height of 2, and the height of 1's left subtree is 0. Remember that balance factor of a tree with a left subtree A and a right subtree B is

B - A

Simple.

In figure 2-2, we see that the tree has a balance of 2. This means that the tree is considered "right heavy". We can correct this by performing what is called a "left rotation". How we determine which rotation to use follows a few basic rules. See psuedo code:

```
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
  }
}
```

As you can see, there is a situation where we need to perform a "double rotation". A single rotation in the situations described in the pseudo code leave the tree in an unbalanced state. Follow these rules, and you should be able to balance an AVL tree following an insert or delete every time.

3. Summary

It’s important to understand that the examples above were on very small trees to keep the concepts clear. In theory, however, if you develop an application which uses AVL trees, programming for the situations shown above while using the rules provided should scale just fine.

If you have comments, questions or criticisms, feel free to e-mail me at castorvx@gmail.com

4. Further Reading

- Tree rotation page on Wikipedia, http://en.wikipedia.org/wiki/Tree_rotation
- AVL tree page on Wikipedia, http://en.wikipedia.org/wiki/AVL_tree
- Animated AVL Tree Java applet, http://webpages.ull.es/users/jriera/Docencia/AVL/AVL%20tree%20applet.htm
- AVL Trees: Tutorial and C++ Implementation, http://www.cmcrossroads.com/bradapp/ftp/src/libs/C++/AvlTrees.html