Chapter 5

AVL Trees

5.1 Introduction

Remember those few minutes weeks ago where we talked about degenerate binary trees and their bad qualities? Well, what if we had a way to avoid that... We do! With **AVL trees**! They do fancy things to prevent this. They're called **AVL trees** because they're named after someone but that's not important for knowing how they work. Sorry inventors!

Do note that **AVL trees** are only one of several version of self-balancing binary trees. Another example would be a red-black tree which may or may not be covered depending on how much time there is left at the end of the semester.

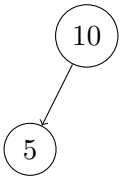
5.2 Description

You've heard of binary search trees, well the **AVL tree** is a binary search tree but a lot fancier. The important thing here is **balance**. When a tree is balanced, this means the heights of the two child subtrees of a node can only differ by up to 1. Let's take a look at multiple examples to understand this.

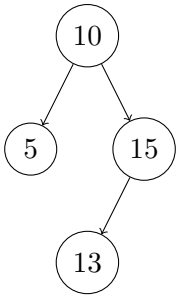
Consider this tree.

What tree? The empty tree of course! Naturally it falls under the category of an **AVL tree**. Jokes aside, let's look at a tree with 1 node.

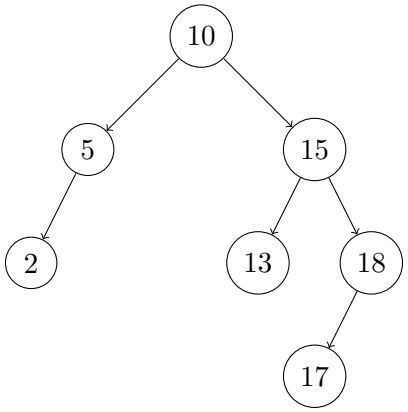
Consider the one node. Do its two child subtrees' heights differ by more than 1? No, both of those subtrees are empty and have the same height. For this class empty trees are considered to have a height of -1. Alright let's add another node.



Is this an **AVL tree**? Alright, heights for child subtrees for 10 are 0 and -1 which works, then heights for child subtrees for 5 are -1 and -1 which also works, so it's good. Let's get a little more complicated:



10 has heights 0 and 1, 5 has heights -1 and -1, 15 has heights 0 and -1, 13 has heights -1 and -1, so yes again this is an **AVL tree**. Now for a large example:



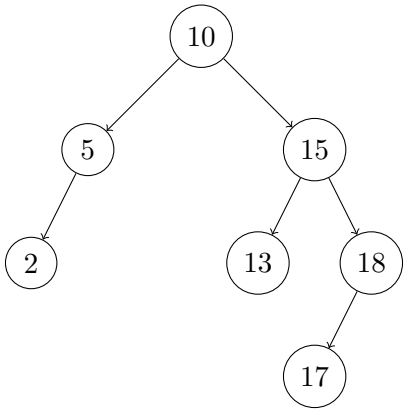
Is this an **AVL tree**? Well I'll tell you it is, but you will have to confirm it. Actually there's something interesting going on here that I'll let you in on.

5.3 Searching

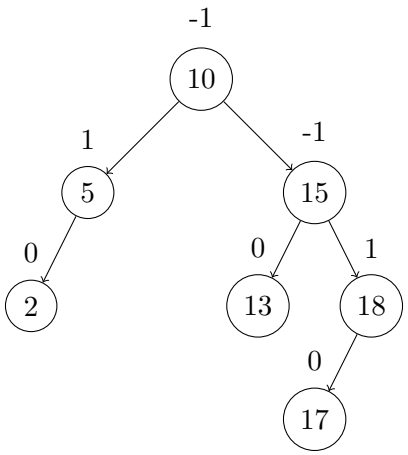
This is a binary search tree, searching works no differently.

5.4 Self Balancing

Whenever the contents of an **AVL tree** are modified, there is a potential for rebalancing the tree. One way to keep track of the balance is to keep a number known as the **balance factor** at each node. This number keeps track of the difference of the heights of the left and right subtrees. Let's look at one of the previous trees:



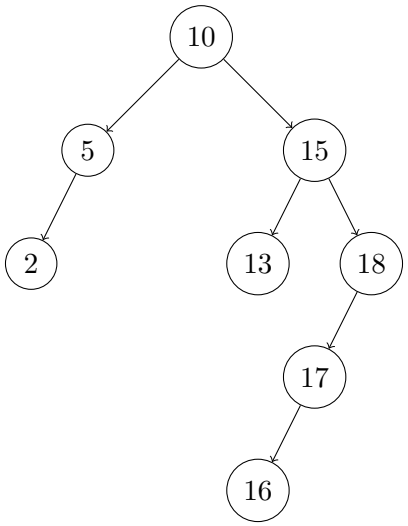
Right away, we can go ahead and assign the leaves a **balance factor** of 0 because their subtrees' heights differ by 0. What about for the node 15? The heights of its left and right subtrees are 0 and 1 respectively. We can subtract the left tree's height from the right tree's to get a **balance factor** of -1. We do this for the rest of the nodes and end up with something like this:



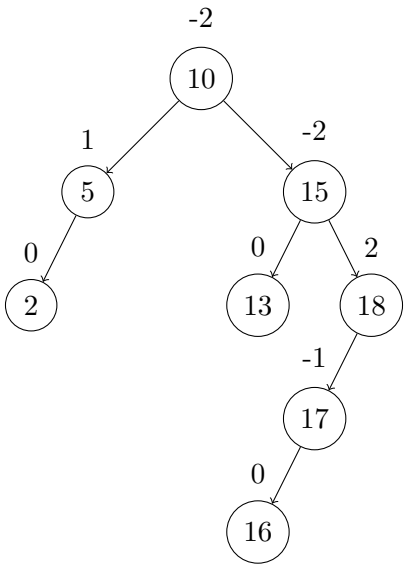
Whats important to note is that all of these balance factors are between -1 and 1 (inclusive). If they were any higher or lower, then that would mean the heights differ by more than 1 and a **rotation** would be needed.

5.5 Insertion

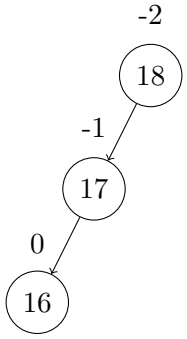
Insertion is like a normal binary search tree. Let's take out previous tree and insert 16.



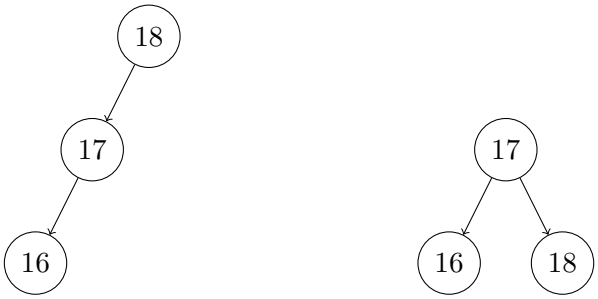
Hopefully you have the hang of inserting into a normal binary search tree. But wait, let's also update the balance factors.



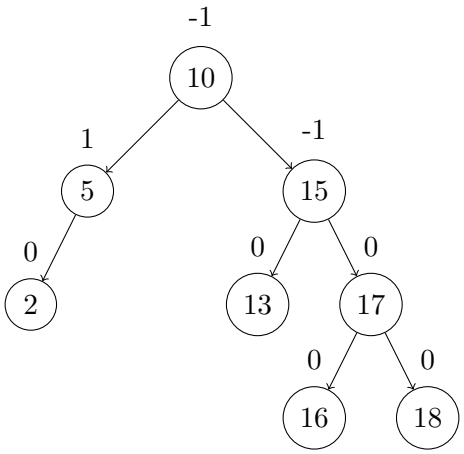
Uh oh, we now have (-)2s as some balance factors. This means we need to **rotate**. We start from the node we inserted and work our way towards the root until we find a balance factor that is too high or too low. This is node 18. We then pick its children whose balance factors are consecutive to it so we will pick these 3 nodes:



, now we want to balance this so what we will rotate these nodes to lower the overall height yet keep them as a valid binary search tree arrangement. Can you see how this is done? Just like this:



And all that's left is to put it back where it was! As you can see, the new balance factors are good.



There are more ways to rotate, but I have a simpler way to remember them all which will be detailed later.

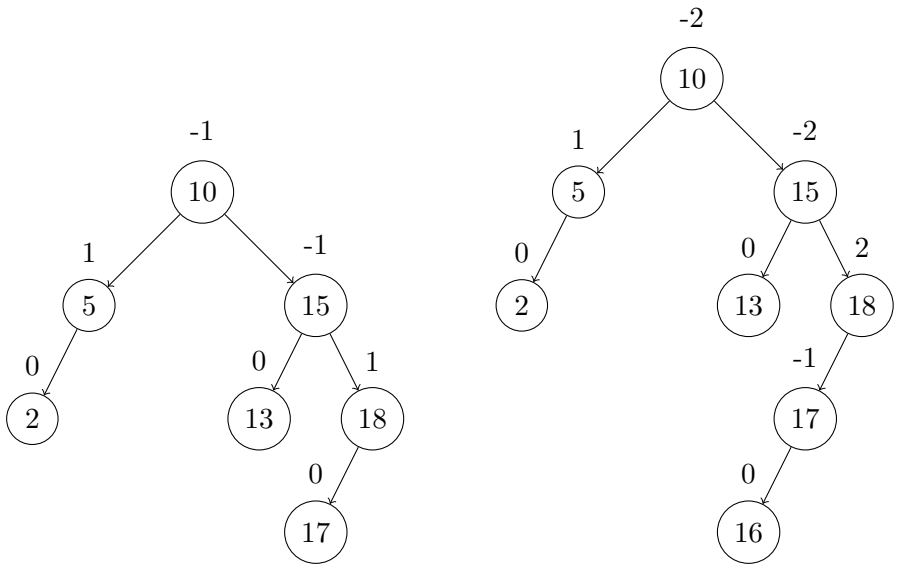
5.6 Deletion

Deletion is the same as a binary search tree, just make sure to rotate if necessary after.

5.7 Easy Rotations

Rotations are the core idea behind **AVL trees** and may be tough. Let's make try to make them simpler.

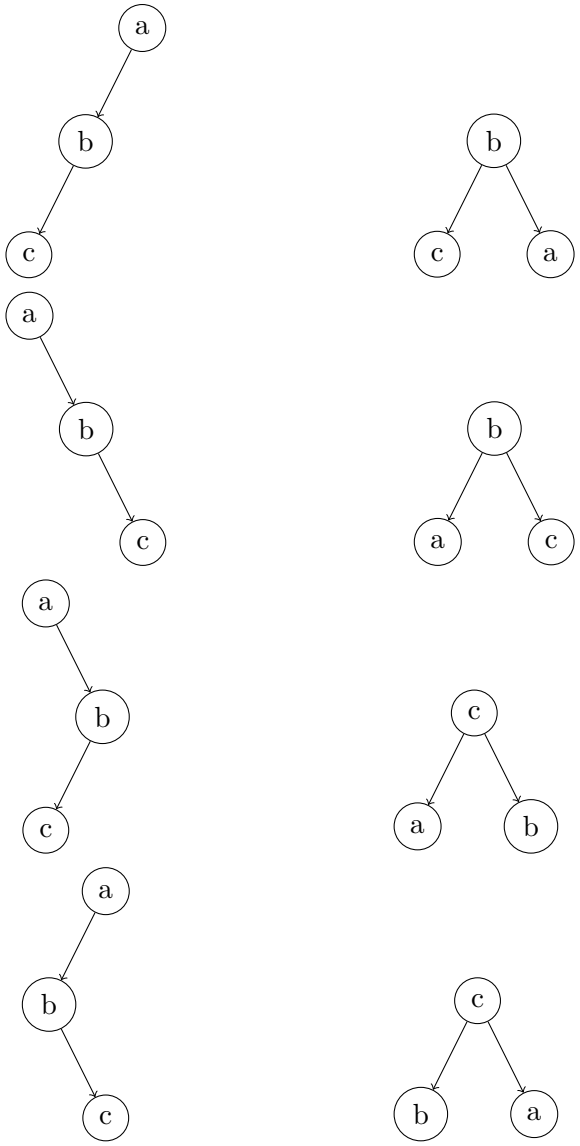
One observation you should be able to make is that when inserting or deleting from a binary search tree, only the heights of the subtrees rooted at nodes on the path from the inserted or deleted node to the root are affected. This also suggests that only balance factors along this path are changed .It's a mouthful but let's look at what it means. By going through our insertion example.



Remember we inserted 16. The balance factors that changed are the ones on the nodes from 16 to the root, 10. This shows us where we may have to rebalance.

Picking out 3 nodes is pretty simple - just work your way up from 16. But once you find them, how should you rotate them? How many ways can you arrange 3 nodes such that they are unbalanced? Exactly 4, and since you know that you need to keep the nodes arranged so they are a valid binary search tree, there's no reason to memorize anything (although I've pictured them below).

One important thing to note is that there may be other subtrees connected to these 3 nodes. Similarly for these, you can also reason out there new positions if necessary.



One final note - you may need to rotate multiple times after an insertion or deletion.

5.8 Examples

Removed for now.