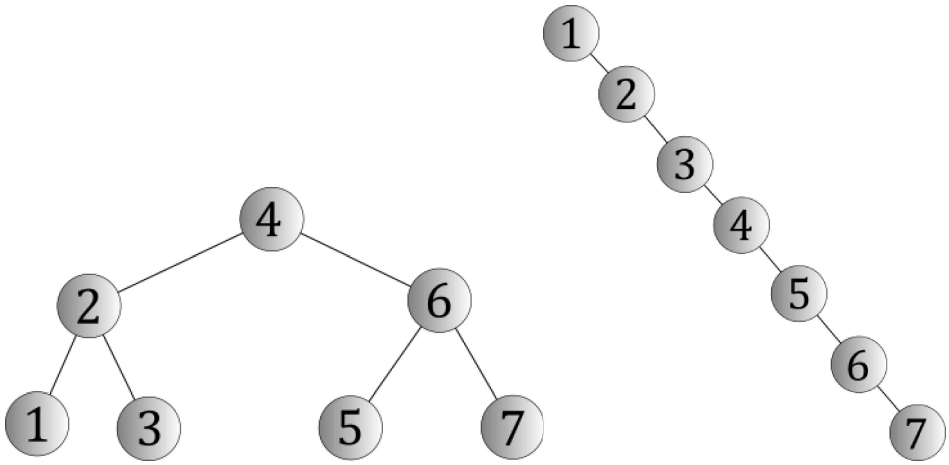
ICS 46 Spring 2022
Notes and Examples: AVL Trees

Why we must care about binary search tree balancing

We've seen previously that the performance characteristics of binary search trees can vary rather wildly, and that they're mainly dependent on the shape of the tree, with the height of the tree being the key determining factor. By definition, binary search trees restrict what keys are allowed to present in which nodes — smaller keys have to be in left subtrees and larger keys in right subtrees — but they specify no restriction on the tree's shape, meaning that both of these are perfectly legal binary search trees containing the keys 1, 2, 3, 4, 5, 6, and 7.



Yet, while both of these are legal, one is better than the other, because the height of the first tree (called a *perfect binary tree*) is smaller than the height of the second (called a *degenerate tree*). These two shapes represent the two extremes — the best and worst possible shapes for a binary search tree containing seven keys.

Of course, when all you have is a very small number of keys like this, any shape will do. But as the number of keys grows, the distinction between these two tree shapes becomes increasingly vital. What's more, the degenerate shape isn't even necessarily a rare edge case: It's what you get when you start with an empty tree and add keys that are already in order, which is a surprisingly common scenario in real-world programs. For example, one very obvious algorithm for generating unique integer keys — when all you care about is that they're unique — is to generate them sequentially.

What's so bad about a degenerate tree, anyway?

Just looking at a picture of a degenerate tree, your intuition should already be telling you that something is amiss. In particular, if you tilt your head 45 degrees to the right, they look just like linked lists; that perception is no accident, as they behave like them, too (except that they're more complicated, to boot!).

From a more analytical perspective, there are three results that should give us pause:

- Every time you perform a lookup in a degenerate binary search tree, it will take $O(n)$ time, because it's possible that you'll have to reach every node in the tree before you're done. As n grows, this is a heavy burden to bear.
- If you implement your lookup recursively, you might also be using $O(n)$ memory, too, as you might end up with as many as n frames on your run-time stack — one for every recursive call. There are ways to mitigate this — for example, some kinds of carefully-written recursion (in some programming languages, including C++) can avoid run-time stack growth as you recurse — but it's still a sign of potential trouble.
- The time it will take you to build the degenerate tree will also be prohibitive. If you start with an empty binary search tree and add keys to it in order, how long does it take to do it?
    - The first key you add will go directly to the root. You could think of this as taking a single step: creating the node.
    - The second key you add will require you to look at the root node, then take one step to the right. You could think of this as taking two steps.
    - Each subsequent key you add will require one more step than the one before it.
    - The total number of steps it would take to add n keys would be determined by the sum $1 + 2 + 3 + ... + n$. This sum, which we'll see several times throughout this course, is equal to $n(n + 1) / 2$.
    - So, the total number of steps to build the entire tree would be $\Theta(n^2)$.

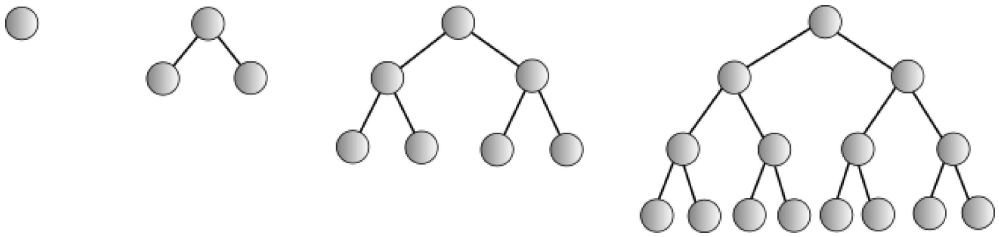
Overall, when n gets large, the tree would be hideously expensive to build, and then every subsequent search would be painful, as well. So this, in general, is a situation we need to be sure to avoid, or else we should probably consider a data structure other than a binary search tree; the worst

case is simply too much of a burden to bear if n might get large. But if we can find a way to control the tree's shape more carefully, to force it to remain more *balanced*, we'll be fine. The question, of course, is how to do it, and, as importantly, whether we can do it while keeping the cost low enough that it doesn't outweigh the benefit.

Aiming for perfection

The best goal for us to shoot for would be to maintain perfection. In other words, every time we insert a key into our binary search tree, it would ideally still be a perfect binary tree, in which case we'd know that the height of the tree would always be $\Theta(\log n)$, with a commensurate effect on performance.

However, when we consider this goal, a problem emerges almost immediately. The following are all perfect binary trees, by definition:



The perfect binary trees pictured above have 1, 3, 7, and 15 nodes respectively, and are the only possible perfect shapes for binary trees with that number of nodes. The problem, though, lies in the fact that there is no valid perfect binary tree with 2 nodes, or with 4, 5, 6, 8, 9, 10, 11, 12, 13, or 14 nodes. So, generally, it's impossible for us to guarantee that a binary search tree will always be "perfect," by our definition, because there's simply no way to represent most numbers of keys.

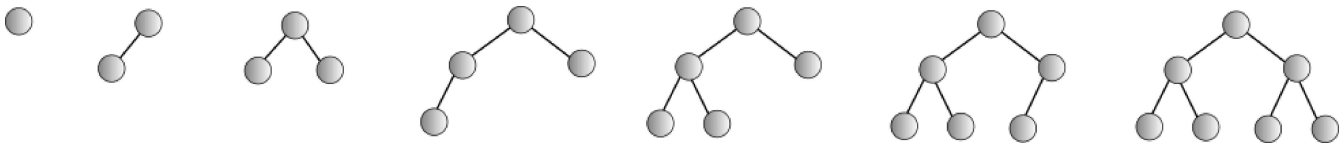
So, first thing's first: We'll need to relax our definition of "perfection" to accommodate every possible number of keys we might want to store.

Complete binary trees

A somewhat more relaxed notion of "perfection" is something called a *complete binary tree*, which is defined as follows.

> A *complete binary tree* of height h is a binary tree where:
>
> - If $h = 0$, its left and right subtrees are empty.
> - If $h > 0$, one of two things is true:
>   - The left subtree is a perfect binary tree of height $h - 1$ and the right subtree is a complete binary tree of height $h - 1$
>   - The left subtree is a complete binary tree of height $h - 1$ and the right subtree is a perfect binary tree of height $h - 2$

That can be a bit of a mind-bending definition, but it actually leads to a conceptually simple result: On every level of a complete binary tree, every node that could possibly be present will be, *except* the last level might be missing nodes, but if it is missing nodes, the nodes that are there will be as far to the left as possible. The following are all complete binary trees:



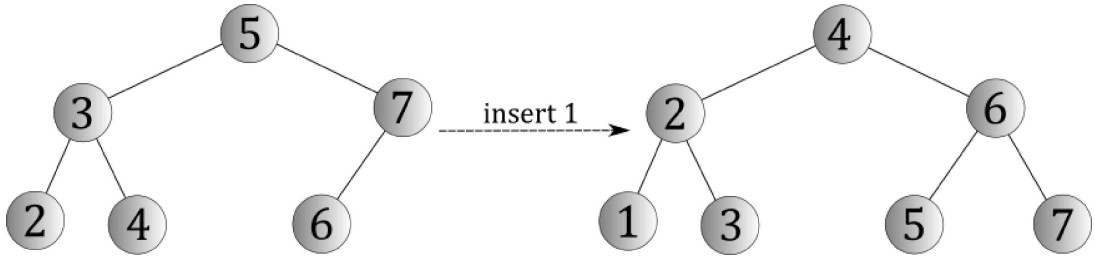
Furthermore, these are the only possible complete binary trees with these numbers of nodes in them; any other arrangement of, say, 6 keys besides the one shown above would violate the definition.

We've seen that the height of a perfect binary tree is $\Theta(\log n)$. It's not a stretch to see that the height a complete binary tree will be $\Theta(\log n)$, as well, and we'll accept that via our intuition for now and proceed. All in all, a complete binary tree would be a great goal for us to attain: If we could keep the shape of our binary search trees complete, we would always have binary search trees with height $\Theta(\log n)$.

The cost of maintaining completeness

The trouble, of course, is that we need an algorithm for maintaining completeness. And before we go to the trouble of trying to figure one out, we should consider whether it's even worth our time. What can we deduce about the cost of maintaining completeness, even if we haven't figured out an algorithm yet?

One example demonstrates a very big problem. Suppose we had the binary search tree on the left — which is complete, by our definition — and we wanted to insert the key 1 into it. If so, we would need an algorithm that would transform the tree on the left into the tree on the right.



The tree on the right is certainly complete, so this would be the outcome we'd want. But consider what it would take to do it. *Every key in the tree had to move!* So, no matter what algorithm we used, we would still have to move every key. If there are n keys in the tree, that would take $\Omega(n)$ time — moving n keys takes at least linear time, even if you have the best possible algorithm for moving them; the work still has to get done.

So, in the worst case, maintaining completeness after a single insertion requires $\Omega(n)$ time. Unfortunately, this is more time than we ought to be spending on maintaining balance. This means we'll need to come up with a compromise; as is often the case when we learn or design algorithms, our willingness to tolerate an imperfect result that's still "good enough" for our uses will often lead to an algorithm that is much faster than one that achieves a perfect result. So what would a "good enough" result be?

What is a "good" balance condition

Our overall goal is for lookups, insertions, and removals from a binary search tree to require $O(\log n)$ time in every case, rather than letting them degrade to a worst-case behavior of $O(n)$. To do that, we need to decide on a *balance condition*, which is to say that we need to understand what shape is considered well-enough balanced for our purposes, even if not perfect.

> A *"good" balance condition* has two properties:
>
> - The height of a binary search tree meeting the condition is $\Theta(\log n)$.
> - It takes $O(\log n)$ time to re-balance the tree on insertions and removals.

In other words, it guarantees that the height of the tree is still logarithmic, which will give us logarithmic-time lookups, and the time spent re-balancing won't exceed the logarithmic time we would otherwise spend on an insertion or removal when the tree has logarithmic height. The cost won't outweigh the benefit.

Coming up with a balance condition like this on our own is a tall task, but we can stand on the shoulders of the giants who came before us, with the definition above helping to guide us toward an understanding of whether we've found what we're looking for.

A compromise: AVL trees

There are a few well-known approaches for maintaining binary search trees in a state of near-balance that meets our notion of a "good" balance condition. One of them is called an *AVL tree*, which we'll explore here. Others, which are outside the scope of this course, include red-black trees (which meet our definition of "good") and splay trees (which don't always meet our definition of "good", but do meet it on an amortized basis), but we'll stick with the one solution to the problem for now.

AVL trees

AVL trees are what you might called "nearly balanced" binary search trees. While they certainly aren't as perfectly-balanced as possible, they nonetheless achieve the goals we've decided on: maintaining logarithmic height at no more than logarithmic cost.

So, what makes a binary search tree "nearly balanced" enough to be considered an AVL tree? The core concept is embodied by something called the *AVL property*.

> We say that a node in a binary search tree has the *AVL property* if the heights of its left and right subtrees differ by no more than 1.

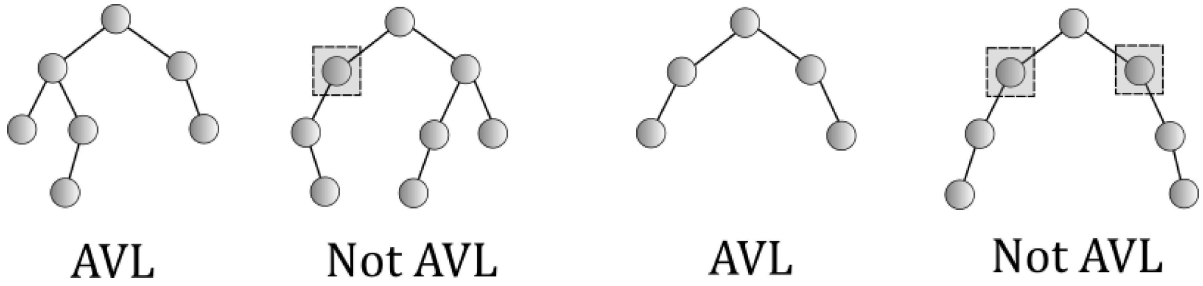
In other words, we tolerate a certain amount of imbalance — heights of subtrees can be slightly different, but no more than that — in hopes that we can more efficiently maintain it.

Since we're going to be comparing heights of subtrees, there's one piece of background we need to consider. Recall that the *height of a tree* is the length of its longest path. By definition, the height of a tree with just a root node (and empty subtrees) would then be zero. But what about a tree that's totally empty? To maintain a clear pattern, relative to other tree heights, we'll say that the *height of an empty tree* is -1. This means that a node with, say, a childless left child and no right child would still be considered balanced.

This leads us, finally, to the definition of an AVL tree:

> An *AVL tree* is a binary search tree in which all nodes have the AVL property.

Below are a few binary trees, two of which are AVL and two of which are not.



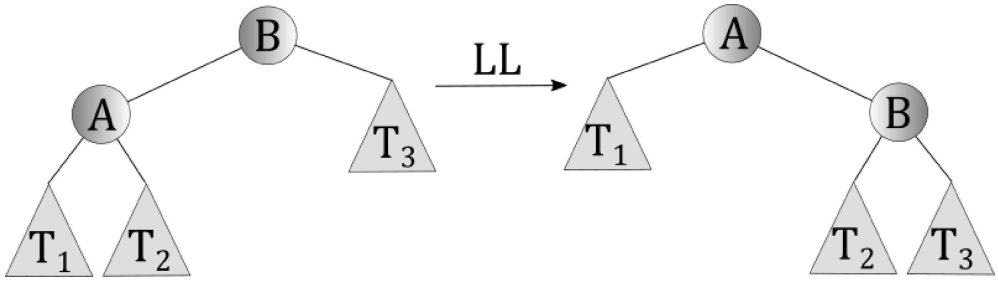
The thing to keep in mind about AVL is that it's not a matter of squinting at a tree and deciding whether it "looks" balanced. There's a precise definition, and the two trees above that don't meet that definition fail to meet it because they each have at least one node (marked in the diagrams by a dashed square) that doesn't have the AVL property.

AVL trees, by definition, are required to meet the balance condition after every operation; every time you insert or remove a key, every node in the tree should have the AVL property. To meet that requirement, we need to restructure the tree periodically, essentially detecting and correcting imbalance whenever and wherever it happens. To do that, we need to rearrange the tree in ways that improve its shape without losing the essential ordering property of a binary search tree: smaller keys toward the left, larger ones toward the right.

Rotations

Re-balancing of AVL trees is achieved using what are called *rotations*, which, when used at the proper times, efficiently improve the shape of the tree by altering a handful of pointers. There are a few kinds of rotations; we should first understand how they work, then focus our attention on when to use them.

The first kind of rotation is called an *LL rotation*, which takes the tree on the left and turns it into the tree on the right. The circle with A and B written in them are each a single node containing a single key; the triangles with T_1, T_2, and T_3 written in them are arbitrary subtrees, which may be empty or may contain any number of nodes (but which are, themselves, binary search trees).



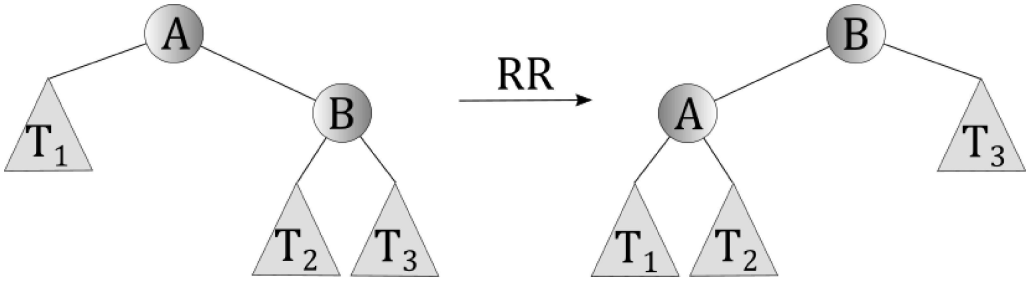
It's important to remember that both of these trees — before and after — are binary search trees; the rotation doesn't harm the ordering of the keys in nodes, because the subtrees T_1, T_2, and T_3 maintain the appropriate positions relative to the keys A and B:

- All keys in T_1 are smaller than A.
- All keys in T_2 are larger than A and smaller than B.
- All keys in T_3 are larger than B.

Performing this rotation would be a simple matter of adjusting a few pointers — notably, a constant number of pointers, no matter how many nodes are in the tree, which means that this rotation would run in $\Theta(1)$ time:

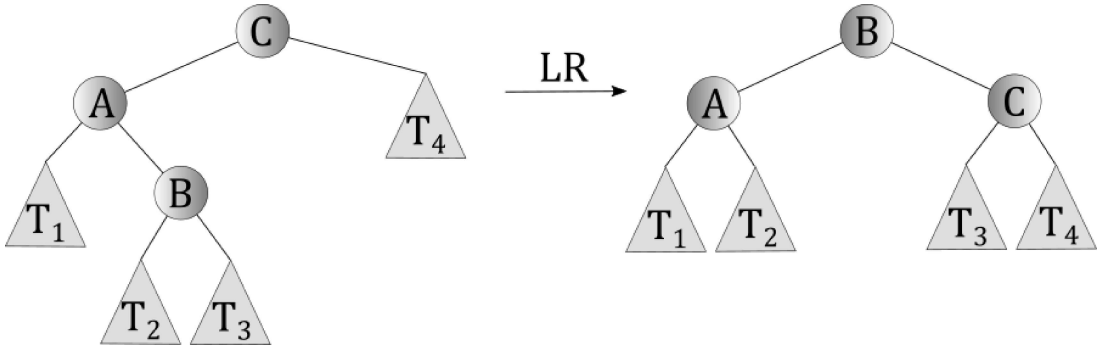
- B's parent would now point to A where it used to point to B
- A's right child would now be B instead of the root of T_2
- B's left child would now be the root of T_2 instead of A

A second kind of rotation is an *RR rotation*, which makes a similar adjustment.



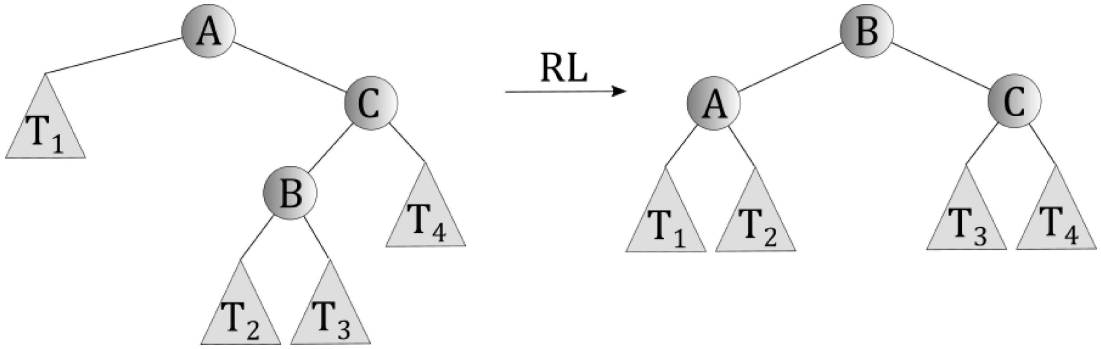
Note that an RR rotation is the mirror image of an LL rotation.

A third kind of rotation is an *LR rotation*, which makes an adjustment that's slightly more complicated.



An LR rotation requires five pointer updates instead of three, but this is still a constant number of changes and runs in Θ(1) time.

Finally, there is an *RL rotation*, which is the mirror image of an LR rotation.



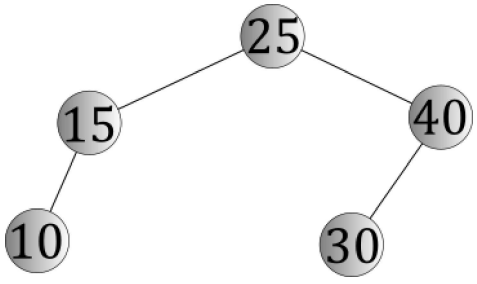
Once we understand the mechanics of how rotations work, we're one step closer to understanding AVL trees. But these rotations aren't arbitrary; they're used specifically to correct imbalances that are detected after insertions or removals.

An insertion algorithm

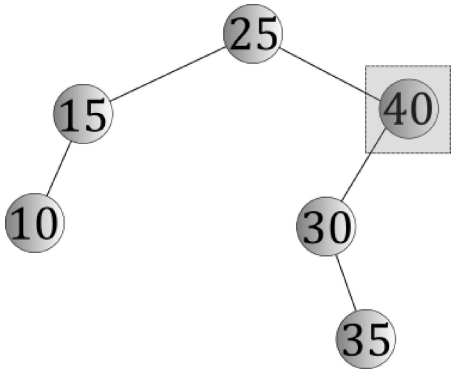
Inserting a key into an AVL tree starts out the same way as insertion into a binary search tree:

- Perform a lookup. If you find the key already in the tree, you're done, because keys in a binary search tree must be unique.
- When the lookup terminates without the key being found, add a new node in the appropriate leaf position where the lookup ended.

The problem is that adding the new node introduced the possibility of an imbalance. For example, suppose we started with this AVL tree:



and then we inserted the key 35 into it. A binary search tree insertion would give us this as a result:



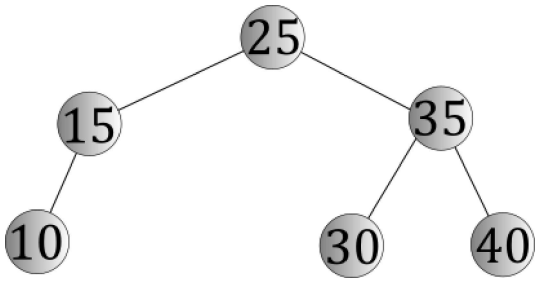
But this resulting tree is not an AVL tree, because the node containing the key 40 does not have the AVL property, because the difference in the heights of its subtrees is 2. (Its left subtree has height 1, its right subtree — which is empty — has height -1.) What can we do about it?

The answer lies in the following algorithm, which we perform after the normal insertion process:

- Work your way back up the tree from the position where you just added a node. (This could be quite simple if the insertion was done recursively.) Compare the heights of the left and right subtrees of each node. When they differ by more than 1, choose a rotation that will fix the imbalance.
  - Note that comparing the heights of the left and right subtrees would be quite expensive if you didn't already know what they were. The solution to this problem is for each node to store its height (i.e., the height of the subtree rooted there). This can be cheaply updated after every insertion or removal as you unwind the recursion.
- The rotation is chosen considering the two links along the path *below* the node where the imbalance is, heading back down toward where you inserted a node. (If you were wondering where the names LL, RR, LR, and RL come from, this is the answer to that mystery.)
  - If the two links are both to the left, perform an LL rotation rooted where the imbalance is.
  - If the two links are both to the right, perform an RR rotation rooted where the imbalance is.
  - If the first link is to the left and the second is to the right, perform an LR rotation rooted where the imbalance is.
  - If the first link is to the right and the second is to the left, perform an RL rotation rooted where the imbalance is.

It can be shown that any one of these rotations — LL, RR, LR, or RL — will correct any imbalance brought on by inserting a key.

In this case, we'd perform an LR rotation — the first two links leading from 40 down toward 35 are a **L**eft and a **R**ight — rooted at 40, which would correct the imbalance, and the tree would be rearranged to look like this:



Compare this to the diagram describing an LR rotation:

- The node containing 40 is C
- The node containing 30 is A
- The node containing 35 is B
- The (empty) left subtree of the node containing 30 is T_1
- The (empty) left subtree of the node containing 35 is T_2
- The (empty) right subtree of the node containing 35 is T_3
- The (empty) right subtree of the node containing 40 is T_4

After the rotation, we see what we'd expect:

- The node B, which in our example contained 35, is now the root of the newly-rotated subtree
- The node A, which in our example contained 30, is now the left child of the root of the newly-rotated subtree
- The node C, which in our example contained 40, is now the right child of the root of the newly-rotated subtree
- The four subtrees T_1, T_2, T_3, and T_4 were all empty, so they are still empty.

Note, too, that the tree is more balanced after the rotation than it was before. This is no accident; a single rotation (LL, RR, LR, or RL) is all that's necessary to correct an imbalance introduced by the insertion algorithm.

A removal algorithm

Removals are somewhat similar to insertions, in the sense that you would start with the usual binary search tree removal algorithm, then find and correct imbalances while the recursion unwinds. The key difference is that removals can require more than one rotation to correct imbalances, but will still only require rotations on the path back up to the root from where the removal occurred — so, generally, $O(\log n)$ rotations.

Asymptotic analysis

The key question here is *What is the height of an AVL tree with n nodes?* If the answer is $\Theta(\log n)$, then we can be certain that lookups, insertions, and removals will take $O(\log n)$ time. How can we be so sure?

Lookups would be $O(\log n)$ because they're the same as they are in a binary search tree that doesn't have the AVL property. If the height of the tree is $\Theta(\log n)$, lookups will run in $O(\log n)$ time. Insertions and removals, despite being slightly more complicated in an AVL tree, do their work by traversing a single path in the tree — potentially all the way down to a leaf position, then all the way back up. If the length of the longest path —

that's what the height of a tree is! — is $\Theta(\log n)$, then we know that none of these paths is longer than that, so insertions and removals will take $O(\log n)$ time.

So we're left with that key question. What is the height of an AVL tree with n nodes? (If you're not curious, you can feel free to just assume this; if you want to know more, keep reading.)

What is the height of an AVL tree with n nodes? (Optional)

The answer revolves around noting how many nodes, at minimum, could be in a binary search tree of height n and still have it be an AVL tree. It turns out AVL trees of height $n \geq 2$ that have the minimum number of nodes in them all share a similar property:

> The AVL tree with height $h \geq 2$ with the minimum number of nodes consists of a root node with two subtrees, one of which is an AVL tree with height $h - 1$ with the minimum number of nodes, the other of which is an AVL tree with height $h - 2$ with the minimum number of nodes.

Given that observation, we can write a recurrence that describes the number of nodes, at minimum, in an AVL tree of height h.

```
M(0) = 1       When height is 0, minimum number of nodes is 1 (a root node with no children)
M(1) = 2       When height is 1, minimum number of nodes is 2 (a root node with one child and not the other)
M(h) = 1 + M(h - 1) + M(h - 2)
```

While the repeated substitution technique we learned previously isn't a good way to try to solve this particular recurrence, we can prove something interesting quite easily. We know for sure that AVL trees with larger heights have a bigger minimum number of nodes than AVL trees with smaller heights — that's fairly self-explanatory — which means that we can be sure that $1 + M(h - 1) \geq M(h - 2)$. Given that, we can conclude the following:

$$M(h) \geq 2M(h - 2)$$

We can then use the repeated substitution technique to determine a lower bound for this recurrence:

$$\begin{aligned}
M(h) &\geq 2M(h - 2) \\
&\geq 2(2M(h - 4)) \\
&\geq 4M(h - 4) \\
&\geq 4(2M(h - 6)) \\
&\geq 8M(h - 6) \\
&\cdots \\
&\geq 2^j M(h - 2j) \qquad \text{We could prove this by induction on } j\text{, but we'll accept it on faith} \\
&\text{let } j = h/2 \\
&\geq 2^{h/2} M(h - h) \\
&\geq 2^{h/2} M(0) \\
M(h) &\geq 2^{h/2}
\end{aligned}$$

So, we've shown that the minimum number of nodes that can be present in an AVL tree of height h is at least $2^{h/2}$. In reality, it's actually more than that, but this gives us something useful to work with; we can use this result to figure out what we're really interested in, which is the opposite: what is the height of an AVL tree with n nodes?

$$\begin{aligned}
M(h) &\geq 2^{h/2} \\
\log_2 M(h) &\geq h/2 \\
2 \log_2 M(h) &\geq h
\end{aligned}$$

Finally, we see that, for AVL trees of height h with the minimum number of nodes, the height is no more than $2 \log_2 n$, where n is the number of nodes in the tree. For AVL trees with more than the minimum number of nodes, the relationship between the number of nodes and the height is even better, though, for reasons we've seen previously, we know that the relationship between the number of nodes and the height of a binary tree can never be better than logarithmic. So, ultimately, we see that the height of an AVL tree with n nodes is $\Theta(\log n)$.

(In reality, it turns out that the bound is lower than $2 \log_2 n$; it's something more akin to about $1.44 \log_2 n$, even for AVL trees with the minimum number of nodes, though the proof of that is more involved and doesn't change the asymptotic result.)