



In this chapter

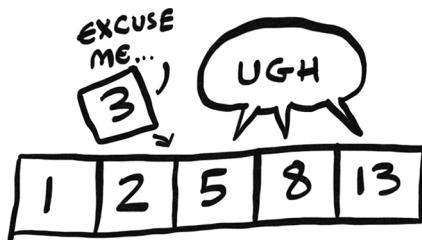
- You learn about a new data structure called binary search trees (BSTs).
- You learn about balanced trees and why they often perform better than arrays or linked lists.
- You also learn about AVL trees, a type of balanced BST. In the worst-case scenario, binary trees can be slow. A balanced tree will help them perform effectively.

In the last chapter, you learned about the new data structure, trees. Now that you and trees are best friends, it's time to see what they are used for. When arrays and linked lists fail to deliver the desired performance, a good next step is to try a tree. In this chapter, we'll discuss the performance that trees can offer. We'll then explore a special type of tree that can offer exceptional performance, called a balanced tree.

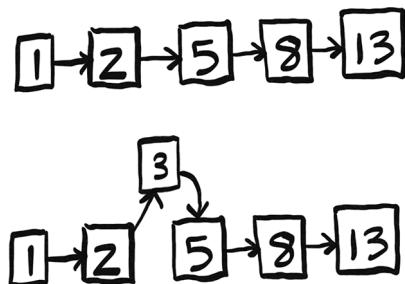


A balancing act

Remember binary search from way back in chapter 1? Using binary search, we are able to find information much more quickly than if we did a simple search using $O(\log n)$ instead of $O(n)$. There is one problem, though: insertion. Sure, searching takes $O(\log n)$ time, but the array needs to be sorted. If you want to insert a new number into your sorted array, it will take $O(n)$ time. The issue is making a spot for the new value. You need to move a bunch of values to make room.



If only we could insert like we do in a linked list, where we just need to change a couple of pointers.



But searching is linear time in linked lists. How can we get the best of both worlds?

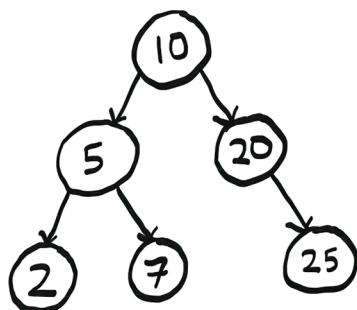
Improving insertion speed with trees

So we really want the search speed of a sorted array with a faster insertion speed. We know that insertions are faster in linked lists. So we want some kind of data structure that combines these ideas.

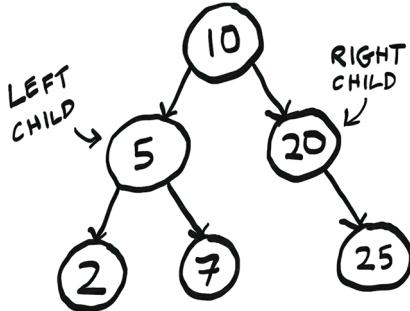
	SEARCH	INSERT
SORTED ARRAY	$O(\log n)$	$O(n)$
LINKED LIST	$O(n)$	$O(1)$
???	$O(\log n)$	FASTER THAN $O(n)$

And that structure is a tree! There are dozens of different types of trees to choose from, so I specifically mean a balanced binary search tree (BST). In this chapter, we will see how a BST works, and then we will learn how to balance it.

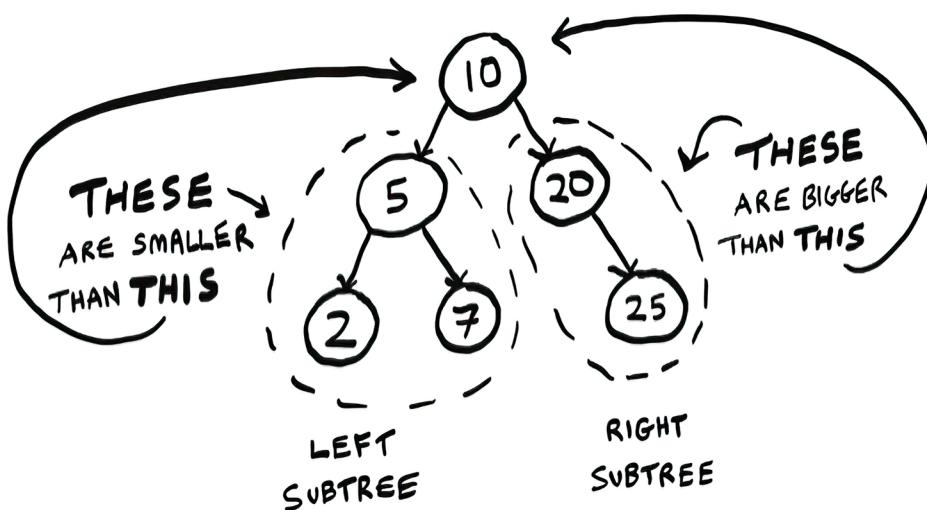
BSTs are a type of binary tree. Here is an example of a BST.



Just like a binary tree, each node has up to two children: the left child and the right child. But it has a special property that makes it a BST: the value of the left child is *always smaller* than the node, and the value of the right child is *always greater*. So for node 10, its left child has a smaller value (5), and its right child has a bigger value (20).

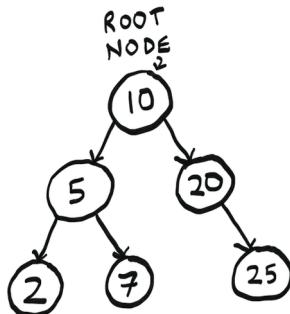


Not only that, all the numbers in the left child subtree are smaller than the node!

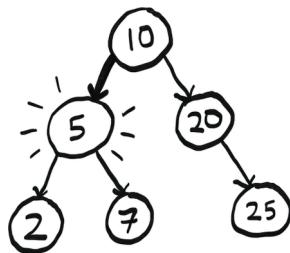


This special property means searches will be very fast.

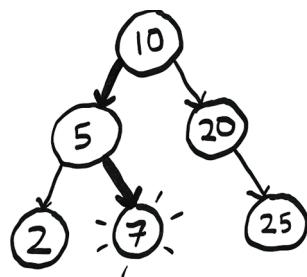
Let's see if the number 7 is in this tree. Here is how we do it. Start at the root node.



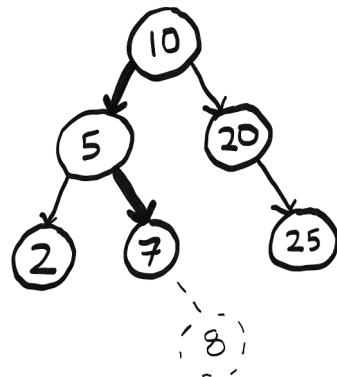
Seven is smaller than 10, so we check the left subtree. Remember, all the nodes with smaller values are on the left, and all the nodes with bigger values are on the right. So we know right away we don't need to check the nodes on the right since 7 won't be there. If we go left from the 10, we get to the 5.



Seven is bigger than 5, so let's go right this time.



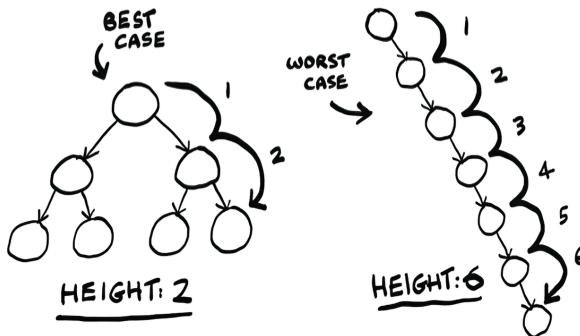
We found it! Now let's look for another number, 8. We follow the exact same path.



Except now it is not there! If it was in the tree, it would be right where the dotted node is. The whole reason we are talking about trees is to see if they are faster than arrays and linked lists. So let's talk about the performance of this tree. To do that, we need to look at the height of the tree.

Shorter trees are faster

Let's look at two trees. They both have seven nodes, but the performance is very different.

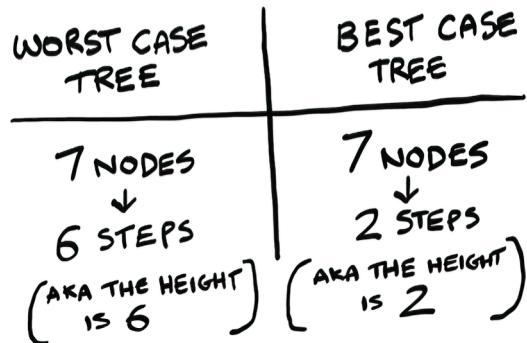


The height of the best-case tree is 2. This means you can get to any node from the root node in, at most, two steps. The height of the worst-case tree is 6. This means you can get to any node from the root node in, at most, six steps. Let's compare this to the performance of binary search versus simple search. Just to remind you, here's the performance of binary search versus simple search:

SIMPLE SEARCH	BINARY SEARCH
100 ITEMS ↓ 100 GUESSES	100 ITEMS ↓ 7 GUESSES
4,000,000,000 ITEMS ↓ 4,000,000,000 GUESSES	4,000,000,000 ITEMS ↓ 32 GUESSES
$O(n)$	$O(\log n)$

Annotations: A large pencil-like shape with the text "O BIG SAVINGS!" is placed over the first comparison row. Another similar annotation is placed over the second comparison row. Arrows point from the text "LINEAR TIME" to the first row and from the text "LOGARITHMIC TIME" to the second row.

Remember our number guessing game? To guess a number out of 100 numbers, it would take 7 guesses with binary search, but 100 guesses with simple search. Well, we're in a similar situation with trees.



The worst-case tree is taller, and it has worse performance. In the worst-case tree, the nodes are all in a line. This tree has height $O(n)$, so searches will take $O(n)$ time. You can think of it this way: this tree is really just a linked list since one node points to another, and so on, in a line. And searching through a linked list takes $O(n)$ time.

The best-case tree has height $O(\log n)$, and searching this tree will take $O(\log n)$ time.



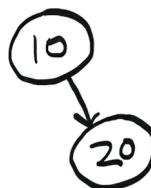
So this situation is very similar to binary search versus simple search! If we can guarantee the height of our tree will be $O(\log n)$, then searching the tree will be $O(\log n)$, just like we wanted.

	SEARCH	INSERT
SORTED ARRAY	$O(\log n)$	$O(n)$
LINKED LIST	$O(n)$	$O(1)$
???	$O(\log n)$	FASTER THAN $O(n)$

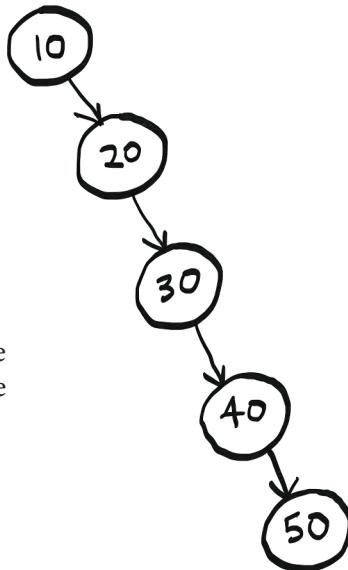
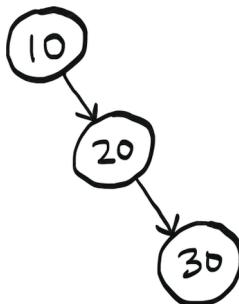
But how can we guarantee the height will be $O(\log n)$? Here's an example where we build a tree that ends up being a worst-case tree (something we want to avoid). We will start with one node.

10

Let's add another one.



So far, so good. Let's add a few more.

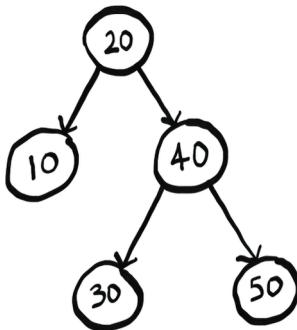


We keep having to add these nodes to the right since they are all bigger than the other nodes.

Yikes, this tree is a worst-case tree—its height is $O(n)$! Shorter trees are faster. The shortest a BST can be is $O(\log n)$. To make a BST shorter, we need to balance it. So let's look at a balanced BST next.

AVL trees: A type of balanced tree

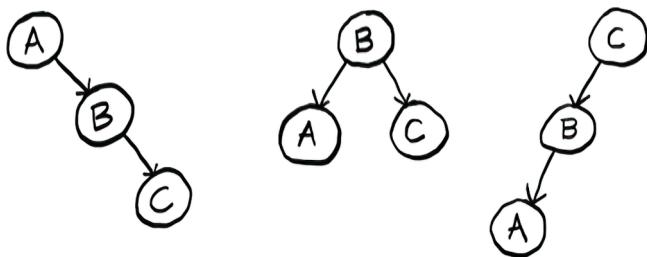
AVL trees are a type of self-balancing BST. This means AVL trees will maintain a height of $O(\log n)$. Whenever the tree is out of balance—that is, the height is not $O(\log n)$ —it will correct itself. For the last example, the tree may balance itself to look like this.



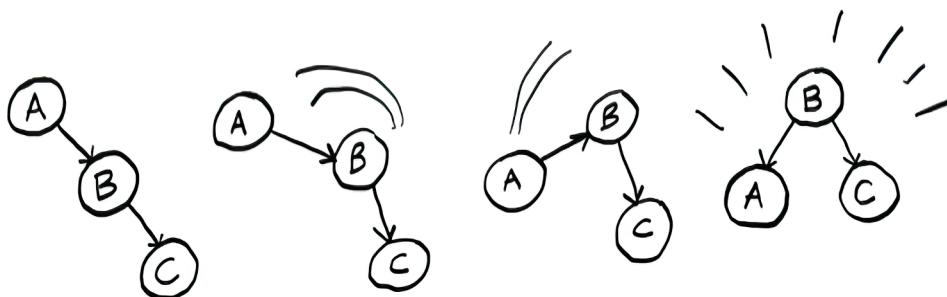
An AVL tree will give us that $O(\log n)$ height we want by balancing itself through rotations.

Rotations

Suppose you have a tree with three nodes. Any one of them could be the root node.

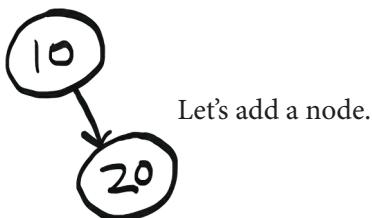


You use rotation to move a set of nodes to end up with a new arrangement. Let's see a rotation in slow motion.



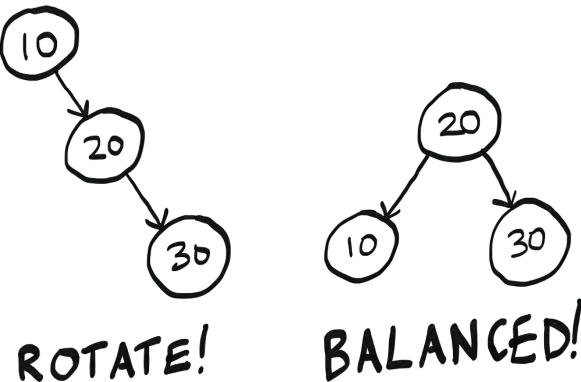
We rotated to the left. We started with an unbalanced tree with A as the root node and ended up with a balanced tree with B as the root node.

Rotations are a popular way to balance trees. AVL trees use rotation to balance. Let's see an example. We'll start with one node again.



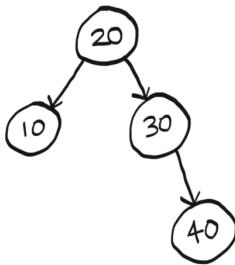
So far, so good. The children aren't exactly the same height; there's a difference of 1. But a difference of 1 is OK for AVL trees. Now we will add one more.

Uh oh! Now the tree is unbalanced. Time to rotate!



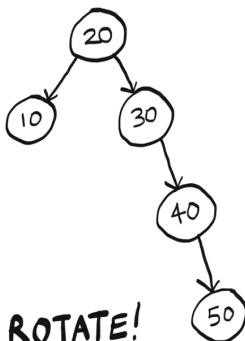
We did a left rotation, and now the tree is balanced again.

Let's add one more node.

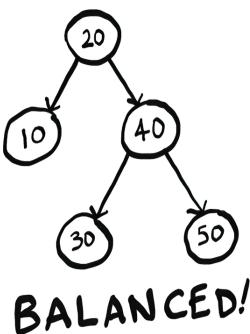


DIFFERENCE
OF 1 IS OK

And add another node.



We need to rotate again!

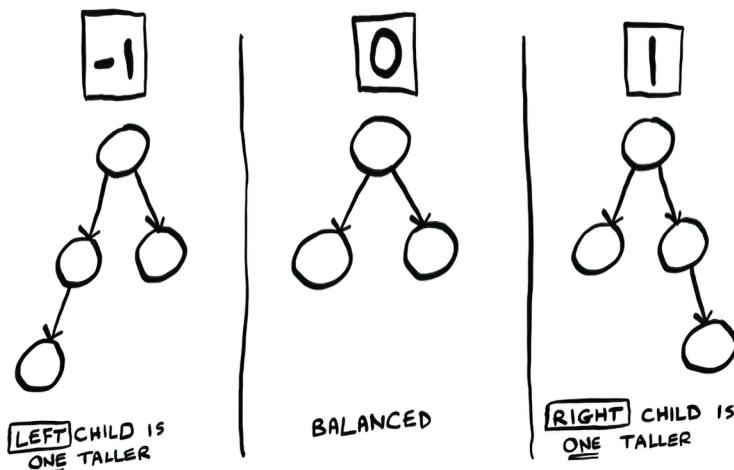


Phew. Using rotation, the AVL tree has balanced itself. Notice in that last example, the node "30" got rotated instead of the node "20". There's an example coming up that explains why.

How does the AVL tree know when it's time to rotate?

We can see visually that the tree is off balance—one side is longer than the other. But how does the tree know that?

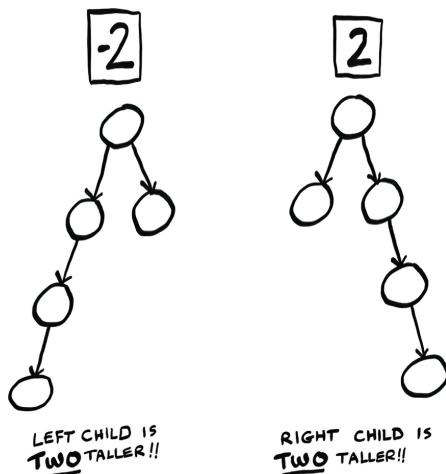
For the tree to know when it's time to balance itself, it needs to store some extra information. Each node stores one of two pieces of information: its height or something called a *balance factor*. The balance factor can be -1 , 0 , or 1 .



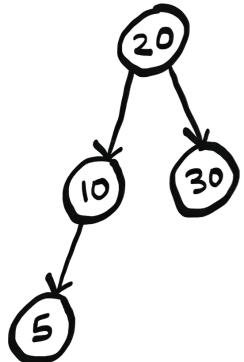
Note that this image shows you the balance factors for the root nodes only, but you would need to store the balance factor for each node (I'll show an example of this shortly).

The balance factor tells you which child is taller and by how much. The balance factor lets the tree know when to rebalance. Zero means the tree is balanced. A -1 or 1 is OK, too, because remember, AVL trees don't have to be perfectly balanced: a difference of one is OK.

But if the balance factor drops below -1 or moves above 1 , it's time to rebalance. To the right are two trees that need rebalancing.



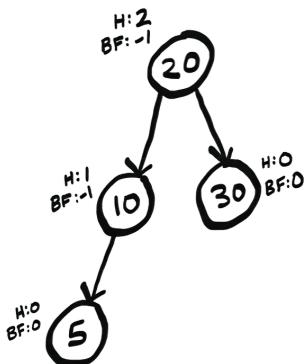
As I said, each node needs to store either the height or the balance factor. In my example, I am going to store both so you can see how they both change. But if you have the heights of each subtree, it is easy to compute the balance factor. Let's see an example. Take this tree.



We are going to add this node to it.

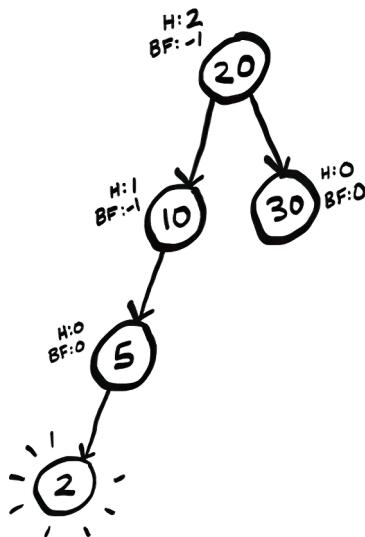


First, let's write out the height and balance factors for each node. In this image, H is height, and BF is balance factor.



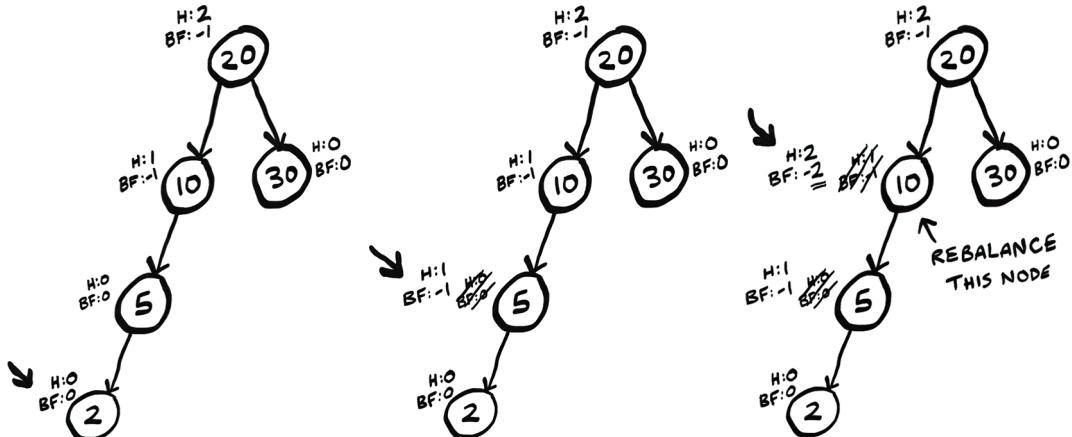
Remember, I am storing both values to show how they change, but you would only need to store one. Make sure these numbers make sense to you. Note that all the leaf nodes have a balance factor of 0: they have no child nodes, so there's nothing to keep in balance.

Now let's add the new node.



After adding this node, we need to set its height and balance factor.

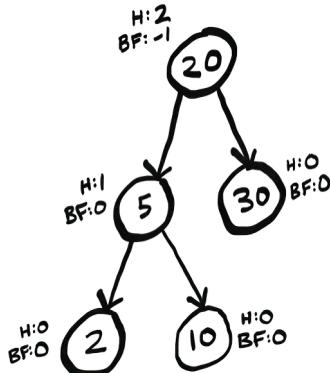
Then we can go up the tree, updating the heights and balance factors for all its ancestors.



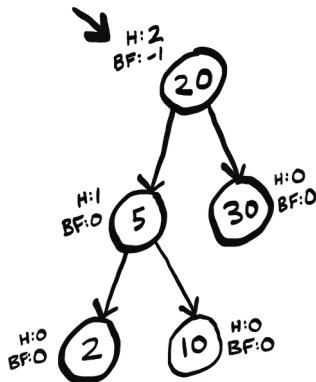
Set the height and balance factor.

Go up the tree and update the height and balance factor for the ancestors.

Aha! We just set the balance factor to -2 , which means it's time to rotate! I will show the rest of the example next, but this is the main takeaway: after an insert, you update the balance factors for that node's ancestors. The AVL tree looks at the balance factor to know when it needs to rebalance. Finishing the example, let's rotate the 10 .



Now that subtree is balanced. Let's keep moving up the tree.



Nothing to update there. We actually didn't need to keep moving up the tree because AVL trees require, at most, one rebalancing.

AVL trees are a good option if you want a balanced BST. Let's recap our journey:

- Binary trees are a type of tree.
- In binary trees, each node has, at most, two children.

- BSTs are a type of binary tree where all the values in the left subtree are smaller than the node, and all the values in the right are greater than the node.
- BSTs can give great performance if we can guarantee their height will be $O(\log n)$.
- AVL trees are self-balancing BSTs, guaranteeing their height will be $O(\log n)$.
- AVL trees balance themselves through rotations.

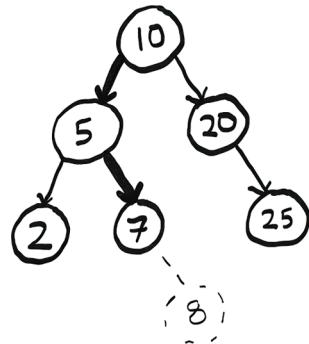
We haven't covered everything. We have covered one case for rotations, and there are other cases. We won't spend time digging into them, as you will rarely need to implement an AVL tree yourself.

Now we know AVL trees offer $O(\log n)$ search performance. What about insertions? Well, insertions are just a matter of searching for a place to insert the node and adding a pointer, just like a linked list. For example, if we want to insert an 8 in this tree, we just need to find where to add the pointer.

So insertions are $O(\log n)$ as well.

At the start of this chapter, we were looking for a data structure that offered both fast searches and fast inserts.

We have found our magic data structure: it's a balanced BST!

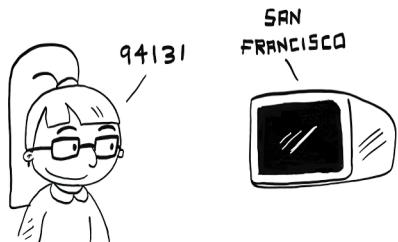


	SEARCH	INSERT
SORTED ARRAY	$O(\log n)$	$O(n)$
LINKED LIST	$O(n)$	$O(1)$
BST	$O(\log n)$	$O(\log n)$

Splay trees

AVL trees are a good basic balanced BST that guarantees $O(\log n)$ time for a bunch of operations.

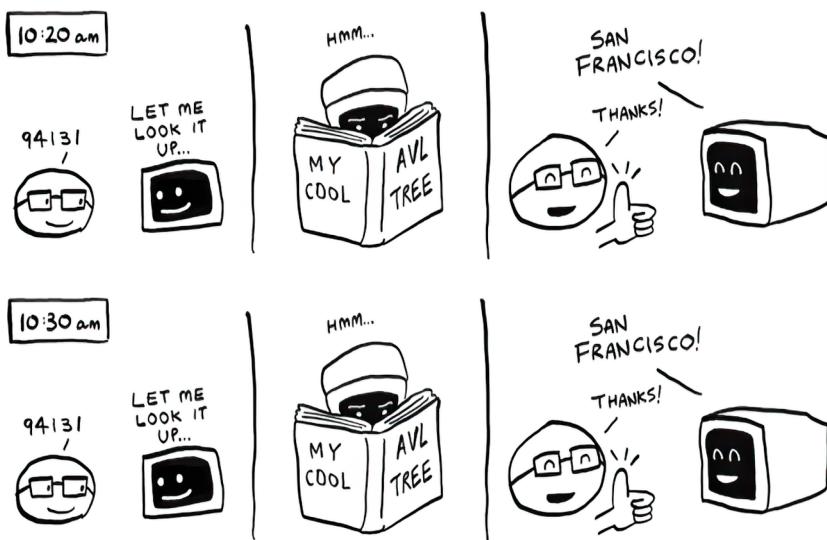
Splay trees are a different take on balanced BSTs. The cool thing about splay trees is if you have recently looked up an item, the next time you look it up, the lookup will be faster. There is something intuitively pleasing about this. For example, suppose you have some software where you give it a zip code, and it will look up the city for you.

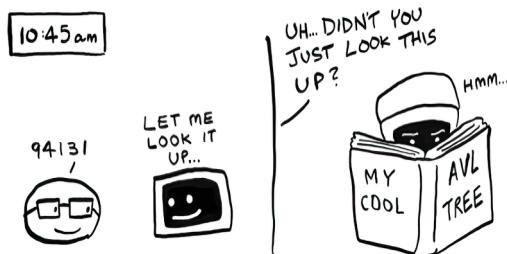


You can picture the interaction going like this.



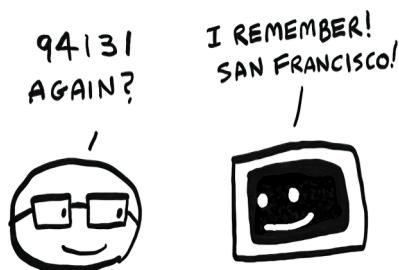
Now suppose you are repeatedly looking up the same zip code.





That feels sort of silly.

The software just looked up the zip code; why can't it remember that? It should really go something like this.



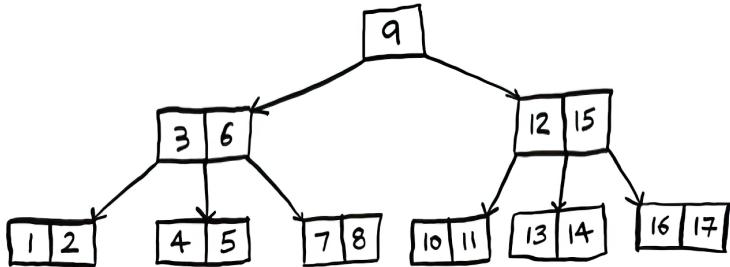
This is what splay trees allow you to do. When you look up a node in a splay tree, it will make that node the new root, so if you look it up again, the lookup will be instant. In general, the nodes you have looked up recently get clustered to the top and become faster to look up.

The tradeoff is the tree is not guaranteed to be balanced. So *some* searches might take longer than $O(\log n)$ time. Some searches may take as long as linear time! Also, while performing the search, you may have to rotate the node up to the root if it is not already the root, and that will take time.

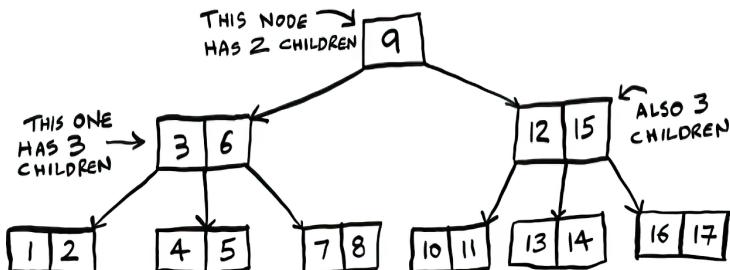
But we are OK with the tradeoff of not having a balanced tree all the time. Because the cool thing is that if you do n searches, the total time is $O(n \log n)$ *guaranteed*—that is, $O(\log n)$ per search. So even though a single search may take longer than $O(\log n)$ time, overall, they will average out to $O(\log n)$ time, and the faster search time is our goal.

B-trees

B-trees are a generalized form of binary tree. They are often used for building databases. Here is a B-tree.

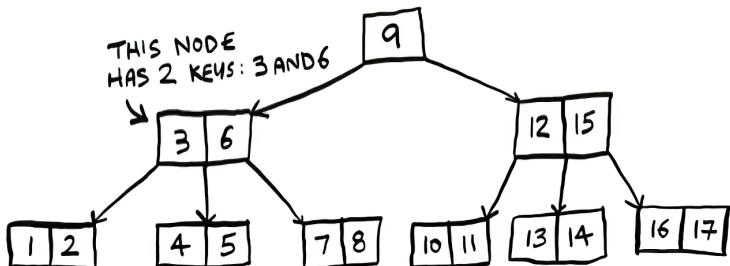


Looks pretty wild, huh? You may notice that some of these nodes have more than two children.



Unlike binary trees, B-trees can have many more children.

You've probably also noticed that, unlike our previous trees, most nodes have two keys.



So not only can nodes in B-trees have more than two children, they can have more than one key! This is why I said B-trees are a generalized form of BSTs.

What is the advantage of B-trees?

B-trees have a very interesting optimization because it's a physical optimization. Computers are physical objects. So when we are looking things up in a tree, a physical object has to move to retrieve that data. This is called *seek time*. Seek time can be a big factor in how fast or slow your algorithm is.

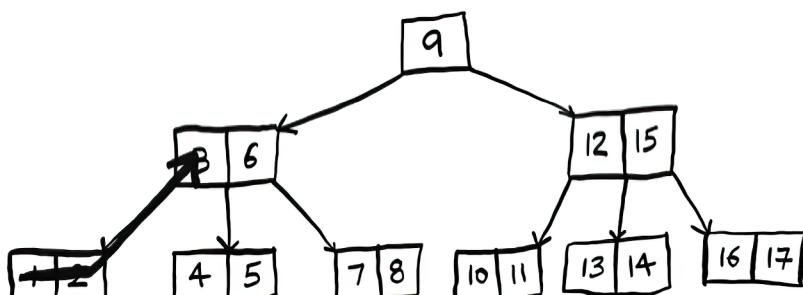
Think of it like going to the grocery store. You could go shopping for one item at a time. Suppose you decide to buy milk. After coming home, you realize you should get some bread, too, so you go back to the store. After coming home again, you realize you're out of coffee. So you go back to the store again. What an inefficient way to shop! It would be much better to shop once and buy a bunch of stuff while you are there. In this example, driving to and from the store is the seek time.

The fundamental idea with B-Trees is that *once you've done the seek, you might as well read a bunch of stuff into memory*. That is, once you're at the store, you might as well buy everything you need instead of going back repeatedly.

B-trees have bigger nodes: each node can have many more keys and children than a binary tree. So we spend more time reading each node. *But we seek less because we read more data in one go*. This is what makes B-trees faster.

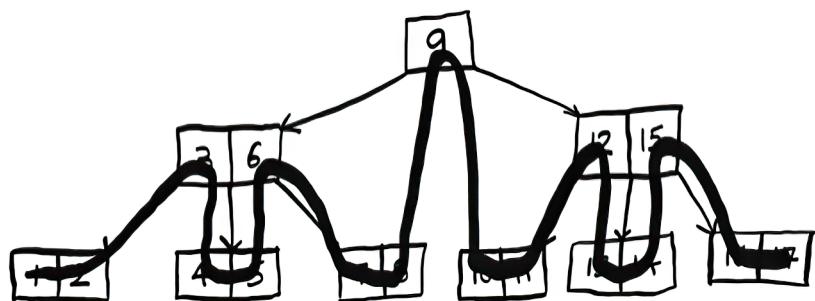
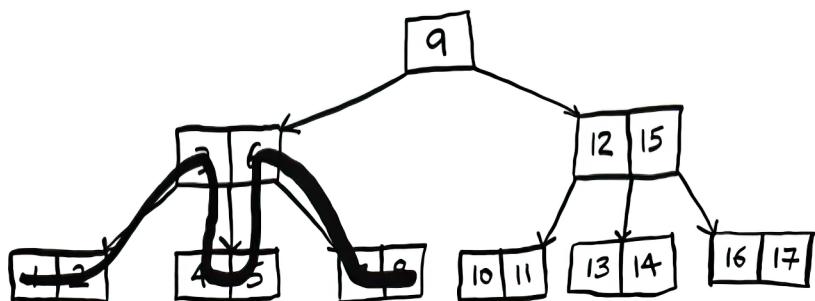
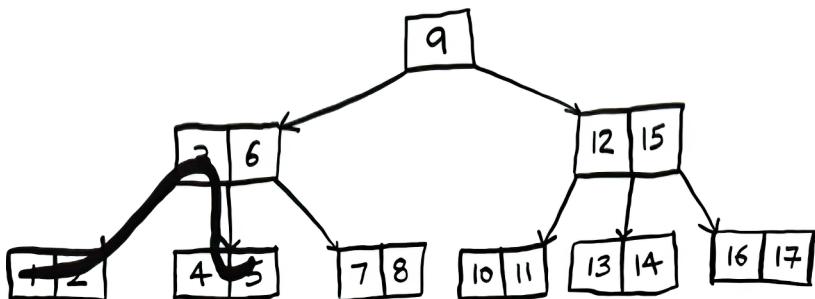
B-trees are a popular data structure for databases, which is no surprise as databases spend a lot of time retrieving data from disk.

Notice the ordering in a B-tree; it is pretty interesting, too. You start at the lower left.



Where do you go from here?

You'll snake across the whole tree.



Notice that it is still following the property of the BST, where for each key, the keys to the left are smaller, and the keys to the right are bigger. For example, for key 3, the keys to the left are 1 and 2, and the keys to the right are 4 and 5.

Also notice that the number of children is one greater than the number of keys. So the root node has one key and two children. Each of those children has two keys and three children.

That concludes our two chapters on trees. It's unlikely you will need to implement a tree yourself, but it's important to know that trees are a type of graph, and they offer great performance. In the next chapter, we will return to graphs and talk about a new type of graph: a weighted graph.

Recap

- Balanced binary search trees (BSTs) offer the same Big O search performance as arrays with better insertion performance.
- The height of a tree affects its performance.
- AVL trees are a popular type of balanced BST. Like most balanced trees, AVL trees balance themselves through rotation.
- B-trees are generalized BSTs, where each node can have multiple keys and multiple child nodes.
- Seek time is like travel time to a grocery store. B-trees try to minimize seek time by reading more data in one go.

