Understand binary search trees

Selecting transcript lines in this section will navigate to timestamp in the video

- A binary tree is a specialized type of tree. It adds the constraint that each node has two immediate child nodes. More terminology here. We call the child nodes left and right nodes respectively. The left child node and the right child node could both be null, both have values, or one of them could be null. What happens with a binary search tree, also called a BST? A binary search tree is also a specialized type where we add an additional constraint, order. In a BST, we keep track of order and keep a sorted data structure by being particular about what values are in the left child, right child, and parent nodes. Why do this? It makes the data structure more than just a collection of stuff strung together. It's a data structure that naturally stays sorted without immense amounts of reshuffling that would be needed in a basic array. What's the rule? A left child node must be less than its parent and a right child node must be more than its parent. That rule follows all the way down. As we insert new nodes with values, the tree must always stay sorted. A binary search tree is often used to sort key value pairs. What we'd be representing here is the key to an object. We can start a binary search tree by adding one node, let's say its key is 30, and since it's the first node, it's the root node. Then, let's add another node. This has the key 20 and since this value is less than the value in the root, 30, we may get the left child. Let's add another node. It will have the value 40 and since it's greater than the value in the root, it becomes a right child. Next, we'll add 25. This is less than the root, so we go to the left and then compare it to what's in the left child node. 25 is greater than what's in the left child node of 20, so it becomes a right child of node 20. What if we add a 35? This is greater than the root node and then we compare it to the right child which is 40. 35 is less than 40, so our new node becomes a left child of 40. Now, what if we add 50, then 60, then 70? These would build out the right side of the tree with 50 becoming a right child of 40, 60 becoming a right child of 50, and 70 becoming a right child of 60. Great, now we have some nodes in our binary search tree, so how do we retrieve these nodes? We would need to find out if the tree contains an object with a certain key, let's say 35. First, we'll compare 35 to the root node and since it's more than that, then we compare it to the right child. The right child is greater than 35, so we look at the left child and there's our match. Why do it this way? Each time we're heading down a path, we are discarding entire sections of the tree, so it's very quick to find a specific element. To keep this optimized, though, the tree must remain sorted. You probably noticed that we have more levels of nodes on the right-hand side than the left. When this happens, we say our tree is unbalanced. The downside to having an unbalanced tree is we would have to perform more checks to find, insert, and delete values on the right-hand side than we would the left. How are these supported in various languages? Typically, they are used behind-the-scenes to implement some data structure. For example, the C++ implementation of a set data structure is done with a binary search tree. It's obstructed from you, the programmer, so you don't even have to worry about it. Although dictionaries are often implemented with hash tables, if you want to keep your keys in a sorted order, hash tables aren't really good at that while binary search trees do a great job.