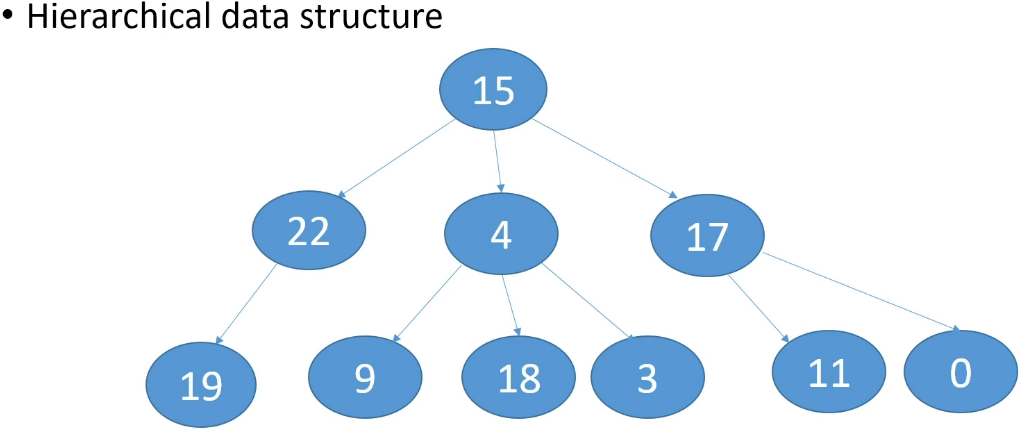
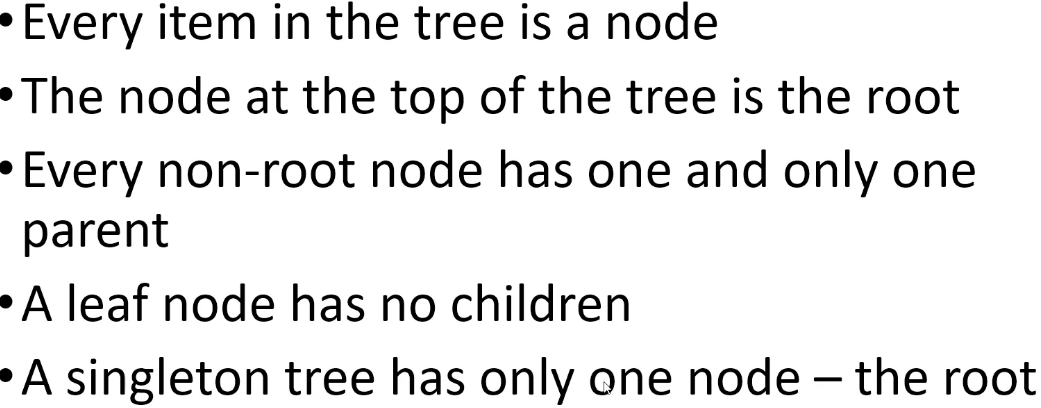
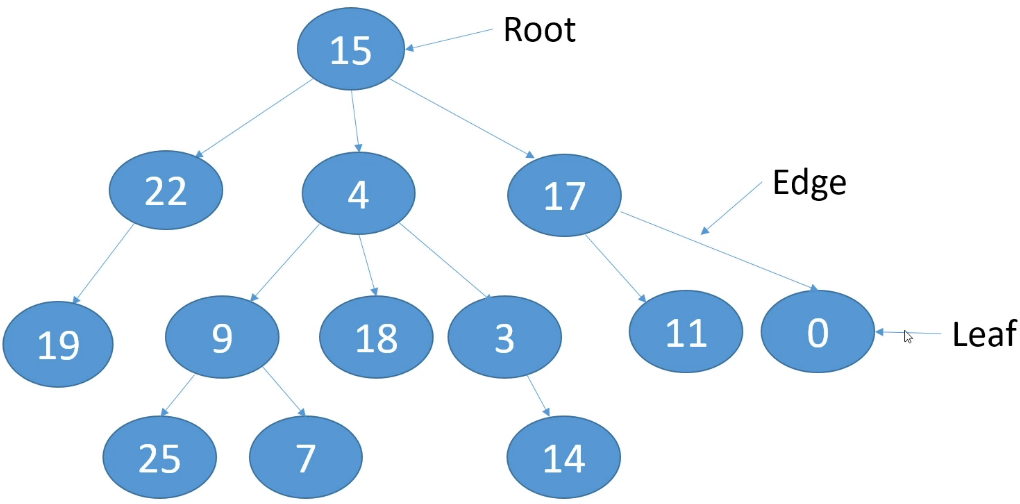
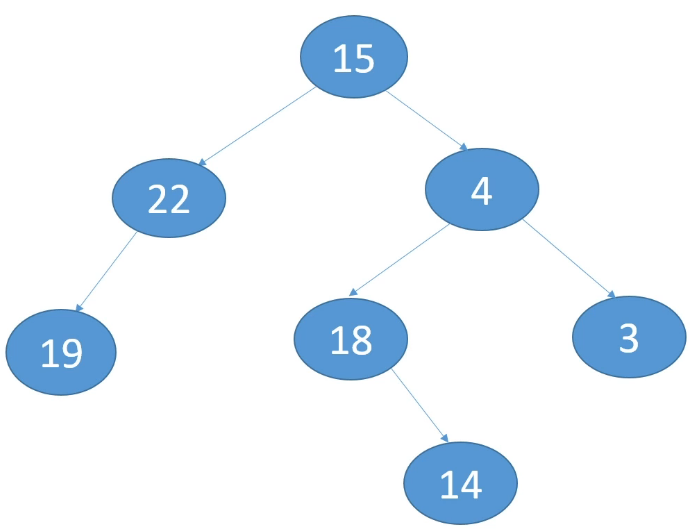
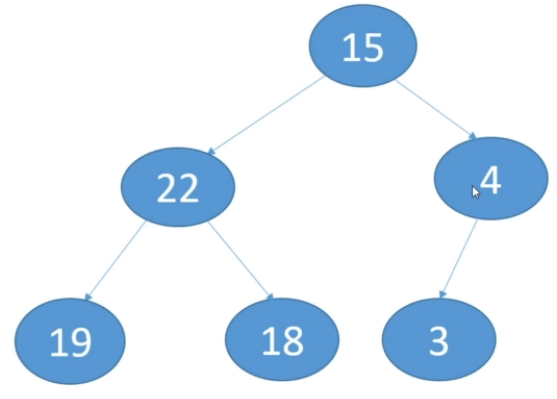
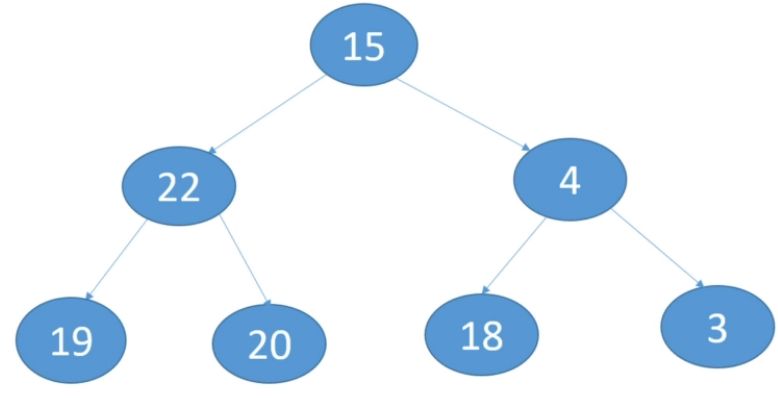
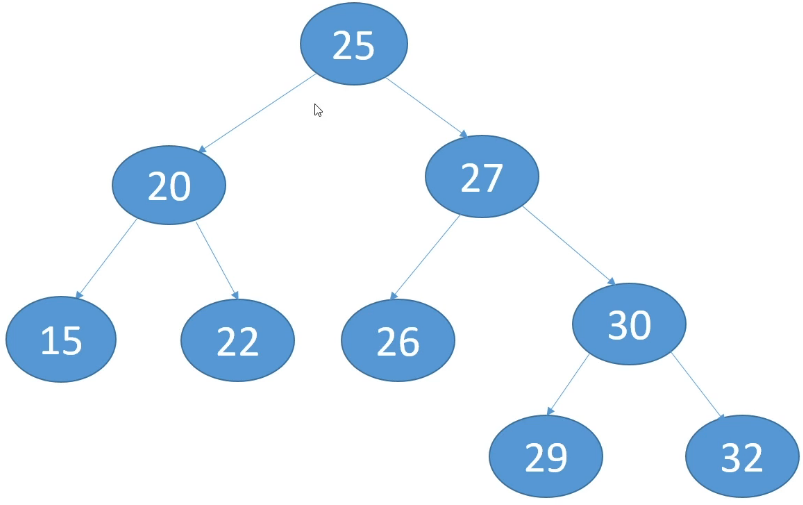
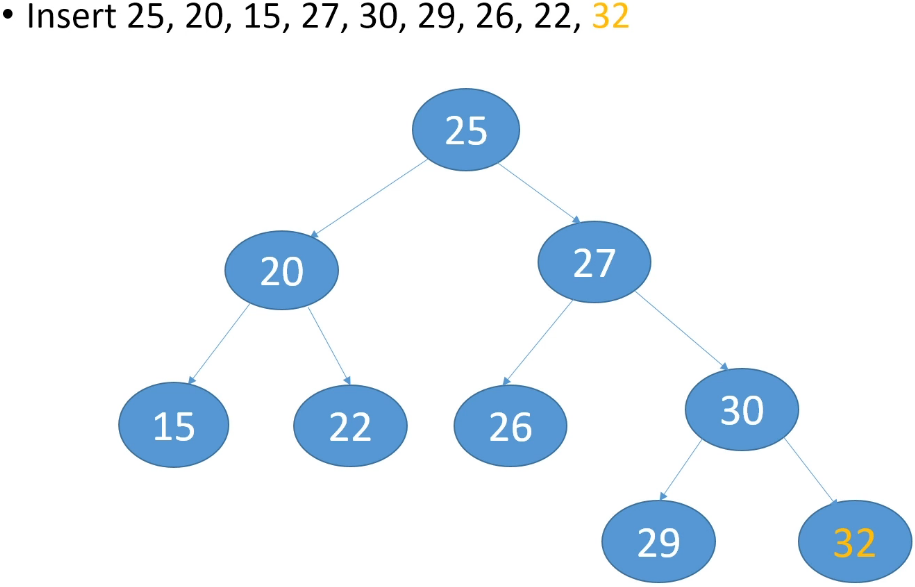
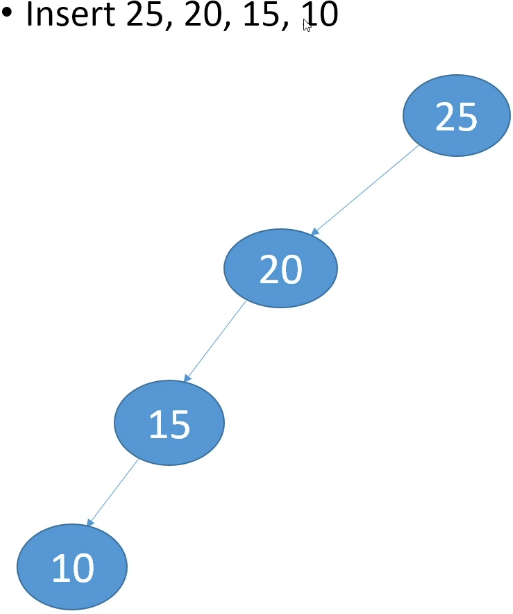
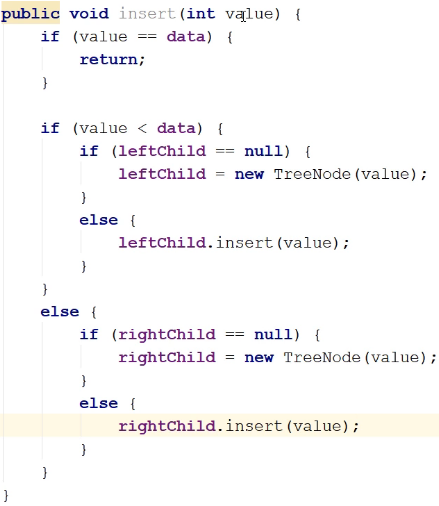
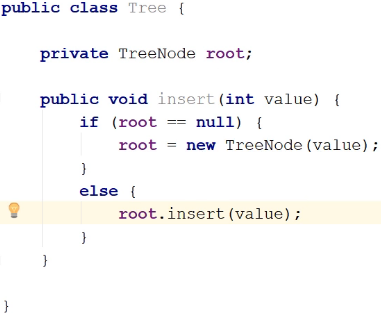
**Introduction to Trees**  
\* **Some people say Trees are** **Data Structures**.  
\* **Some people say Trees are** **Abstract Data Types**.  
\* It’s a bit of a grey area, because trees do dictate how to organize the data.  
\* And you can write a tree using Tree and TreeNode classes and that’s commonly done.  
\* But, you can also back certain types of trees with arrays.  
\* We’re going to start by looking at the characteristics of trees in general and then we’re going to move on to a specific type of tree called a **Binary Search Tree** - **that’s a tree that you’ll probably use in practice**.

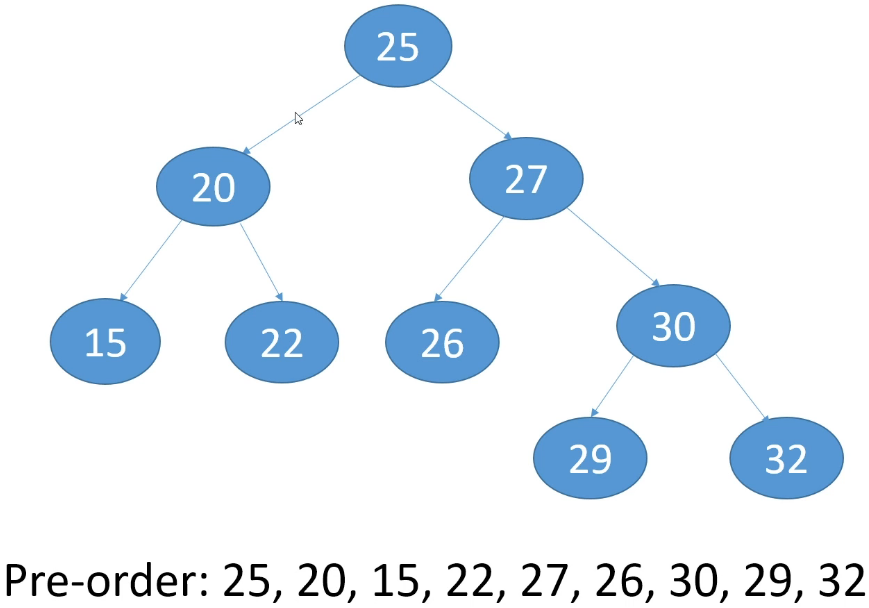
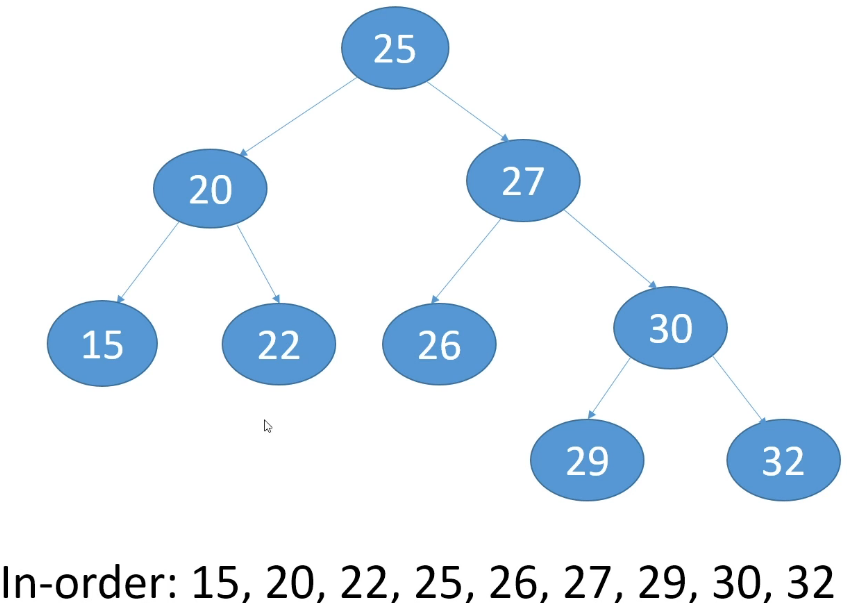
**Trees (Theory)**  
\* **Tree** is a **hierarchical data structure**.  
  
**Node** => every circle in this picture.  
=> **Nodes can have children**.  
=> Each Node can have **only 1 parent**.  
**Leaf** => **no children**.  
**Root** => a special Node in every tree, **doesn’t have a parent**.  
=> Eech Tree can have **only 1 Root Node**.  
\* This picture is how trees are usually visually represented.  
\* **Trees - ideal when things can contain other things or when things can descend from other things**.  
=> For example here 22 is a descendant of 15 because 15 is a parent of 22.  
=> 19 descends from 22 because 22 is a parent of 19.  
=> **So when you have a situation where items can contain other items or there’s a hierarchical relationship such that items can descend from other items, trees are a good data structure to use**.  
\* A couple of examples would be the **Java class hierarchy**.  
=> **That’s a tree** because a class can extend only 1 other class, meaning that each class can only have 1 parent. But you can have multiple classes extending the same class, so a class can have many children. The Java class hierarchy only has **1 Root - Object class, because every class in the JDK ultimately descends from Object**.  
\* File System on your hard drive => every folder can have children and those children would be folders or files, a folder/file can only belong to one folder so it can only have 1 parent. The File System could have multiple Roots in the sense that it can have multiple drives, but each drive is a Tree and so the Root of the C drive would be the C directory, the topmost directory.  
**Singleton** => a singleton tree has **only 1 node - the Root**.  
  
  
**Edge** => the connection between Nodes.  
**Subtree** - **the node and all of its descendants**.  
\* **Every tree consists of 1 or more substrees**.  
\* For substrees you can start at any given Node.  
=> Node 17’s sub-tree consists of 17, 11 and 0.  
=> For 15 it would be the entire tree.  
**Path** => a sequence of nodes required to go from 1 Node to another (includes the start/end too).  
=> The path between Nodes 4 and 25 is: 4 - 9 - 25.  
\* **You can’t have cyclic paths in a Tree**.  
=> **We can’t have a path that crosses the same Node more than once.**  
\* That’s a defining characteristic of a Tree.  
**Root** **Path** => the path going in the other direction - it’s **how you get from a Node to the Root**.  
=> The root path for 3 would be 3 - 4 - 15.  
**Depth** of a Node => **the number of edges from the Node to the Root**.  
=> **The root always has the depth of 0**.  
**Height** of a Node => **the number of edges on the longest path from the Node to a Leaf**.  
=> The height of 4 is 2 because the longest path from 4 to a Leaf is 2 edges.  
=> **Leaf nodes will have a height of 0**.  
**Height of the Tree** => **the height of its root node = the longest path from the Root to a Leaf**.  
=> The height of our tree is 3.  
\* When we’re figuring out depth, we start at the Node and we work up to the Root.  
\* When we’re figuring out height, we start at the Node and we work down the longest path to a Leaf.  
**Siblings** => the nodes that are on the same level.  
=> Siblings don’t necessarily have the same height.  
**Level** of a tree => contains all the nodes that are at the same depth.  
=> The root is at level 0, it has a depth 0.  
=> 22, 4, 17 are at level 1, they have a depth 1.  
**Ancestor** => **if it’s in the Node’s path from the Root to that Node**.  
=> 3 has 2 Ancestors - 4 and 15, because we have to pass through 15 and 4 to get to 3.

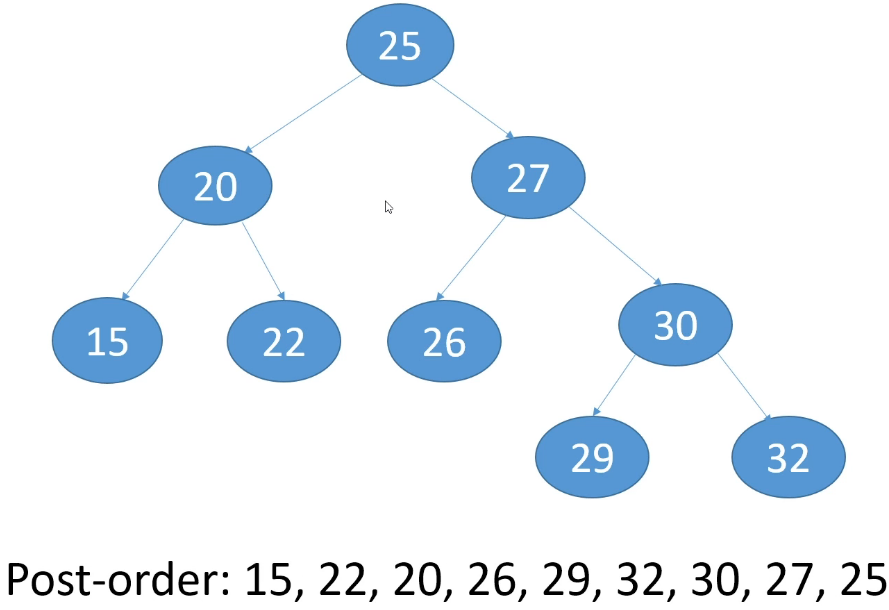
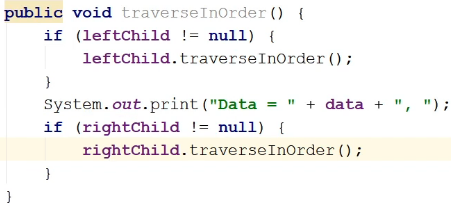
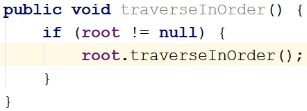
**Binary Search Trees (Theory)**  
=> **Every node has 0, 1, or 2 children**.  
=> Children are referred to as **left** **child** and **right** **child**.  
\* In practice, we don’t use a regular Binary Tree, we use a **Binary Search Tree**.  
  
**Complete Binary Tree** => **a Binary Tree is complete if**:  
=> **every level except the last level has 2 children**.  
=> **on the last level, all of the nodes are as left as possible**.  
  
=> **Every level except the last level is completely filled**.  
=> All of the iterior Nodes have to have 2 children.  
=> On the last level, all of the Nodes have to be to the left as much as possible.  
**Full** **Binary** **Tree**   
=> **a Complete tree**=> **every node other than the Leaves, has to have 2 children**.  
  
\* It’s fine to have incomplete binary trees.  

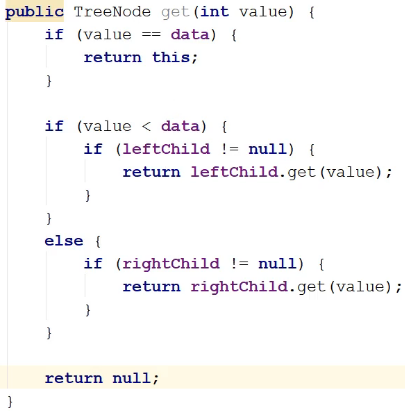
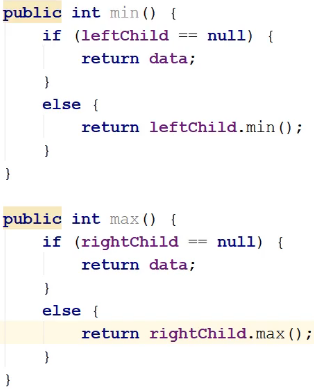
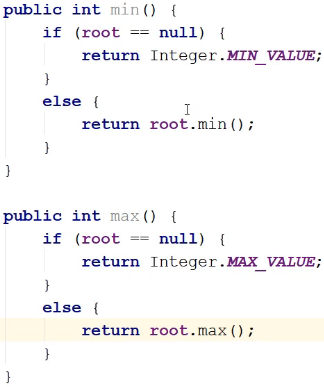

\* In practice, we generally don’t use just any binary tree, we use what’s called:  
**Binary Search Tree (BST)**  
=> The reason that binary search trees are popular is that we can perform:  
**insertions**, **deletions** and **retrievals** in **O(logn)** time.  
=> They also have **faster searching than unsorted arrays** do, but **equivalent time complexity to sorted arrays** - when we search a sorted array using a binary search algorithm, we can do it in O(logn).  
=> **The left child always has a smaller value than its parent**.  
=> **The right child always has a larger value than its parent**.  
=> This means **everything to the left of the Root or to the left of a parent, is < less than the value** of the Root or the parent.  
=> And **everything right of the Root or right of the parent, is > greater than the value** of the Root or the parent.  
=> **Because of that, we can do a binary search**. Because we look at the value of the root, if it’s equal to the value we want, we’re done, if it’s < value at the root, then we look at the left subtree, if the value is > vale, we look at the right subtree.  
\* **So right away, just by checking the root, we cut the number of values we have to search in half**.  
\* **Binary Search Trees are ideal for doing binary searches** and that’s why they’re called Binary Search Trees.  
=> And because of that as well, we can do insertions, deletions and retrievals in O(logn). Because it only ever takes us O(logn) steps to find the insertion point or to find the value we want to delete or retrieve.  
  
\* **What about duplicate values?**=> There are a couple of approaches to this.  
1) => Some implementations just say that **duplicates are not allowed**. And so if you try to insert a duplicate value, it’s not allowed.  
2) => **If you want to allow them, one way to handle them is to always store duplicates either in the left sub-tree or the right sub-tree, you have to choose one and stick with it**.  
3) => A third approach is to have a counter with each Node, and so rather than adding a separate Node for a duplicate value, you would just increment the counter.

  
\* How would be build this tree?  
\* What insertions did we do to get this tree?  
=> **The order in which you insert the Nodes is going to influence how the tree ultimately looks**.  
\* **We always insert the Node into the first empty spot we find**.  
  
\* If we mixed up the insertion order of these values, we would end up building a different tree because different values are going to go into the Root and our comparisons will be different.  
\* **One really important characteristic to note for Binary Trees is**:  
=> **you can get the minimum value in the tree just by following the left edges**.  
=> **you can get the maximum value in the tree just by following the right edges**.  
\* And so **you can get min and max values very quickly**.  
\* This maya sound counter intuitive but if you insert sorted data into a BST, you’re going to end up with this situation:  
  
=> **This is not a good situation, this is essentially a Linked List**.  
=> **If you search in this, you’re going to get O(n).**  
\* Ideally when you’re building a BST, you try to keep the tree as balances as possible and that means that the heights of the left subtree and the right subtree don’t differ by much.  
\* There are BSTs that are:   
**Self Balancing BST => after every insertion or deletion, they look at the tree and if the tree is starting to get unbalanced, they rebalance the tree by shifting Nodes around**.   
\* **We’re not going to look at them in this course** because **they’re more advanced** but I just wanted to mention that they exist.  
\* Two common ones are:  
**AVL Trees  
Red-Black Trees**  
\* AVL - Adelson-Velskii and Landis names by the inventors.

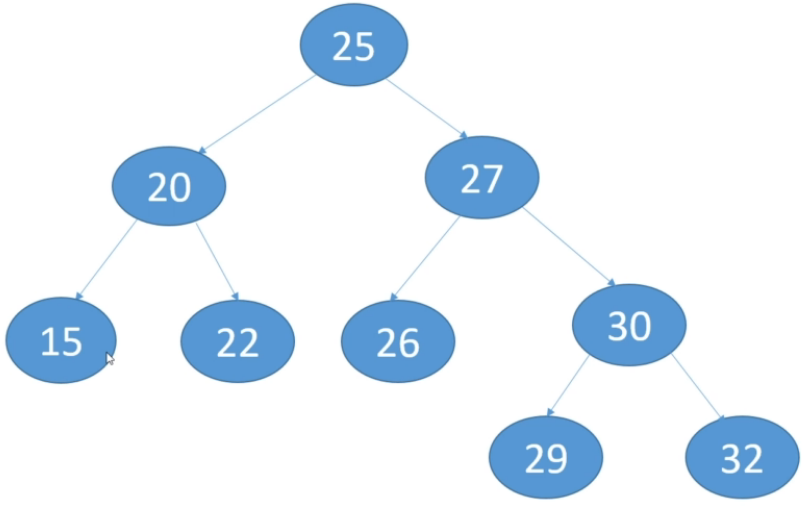
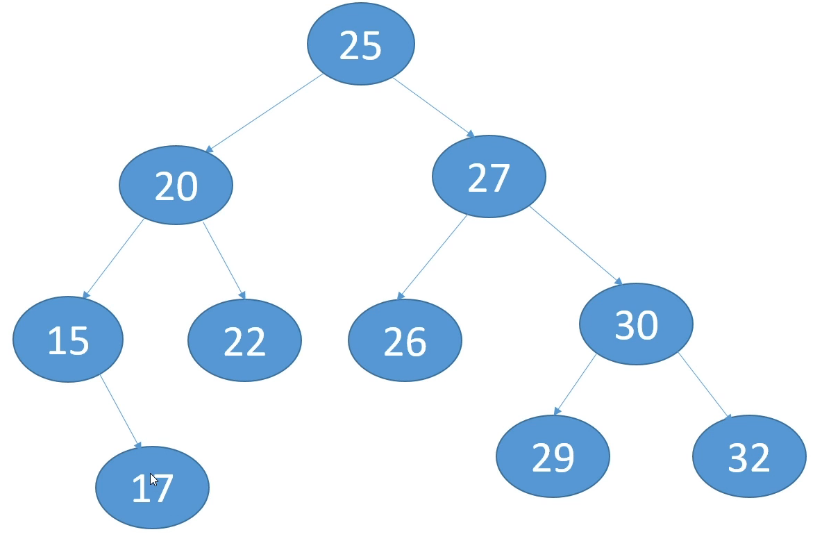
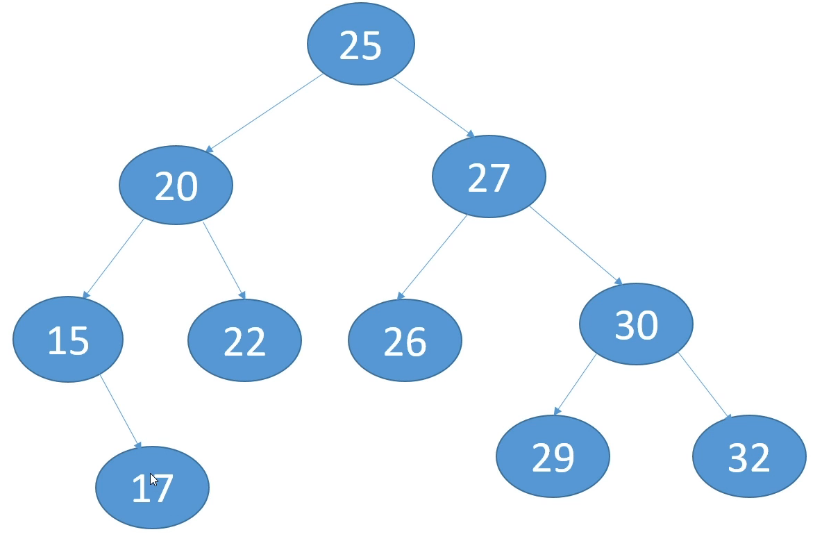
**Binary Search Trees (Insertion)**  
\* You can store any type of data in a Binary Search tree.  
\* We’re going to store ints.  
\* We’re not going to allow duplicate values.  
\* **We will have an insert() method in the Tree but also in the TreeNode**.  
\* **insert() in TreeNode**:  
  
\* If the value is < this node, we’re going to explore its left sub-tree.  
\* If the value is > this node, we’re going to explore its right sub-tree.  
\* **insert() in Tree**:  


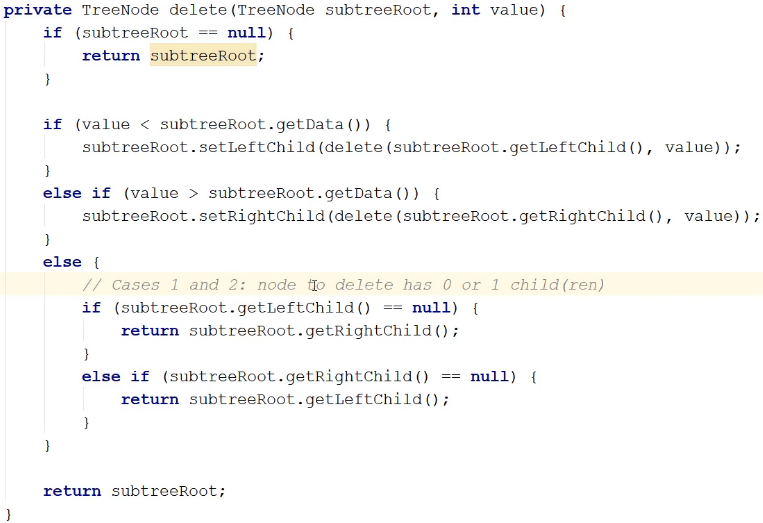
**Binary Search Trees (Traversal)**  
\* We say **visit a Node** when talking about Trees.  
\* There are 4 ways you can traverse a tree, 1 isn’t used very often and the other 3 are.  
1) **Level** => visit nodes on each level, starting from the top  
=> so we would visit the root (level 0), then we visit the nodes at level 1 going from left to right and then we would visit the nodes at level 2 from left to right, etc.  
2) **Pre-order** => visit the root of every subtree first  
=> visit the root, then we’re going to visit the roof the left subtree and the root of its left subtree, etc. until we get down to the first Leaf and then we visit the Leaves of each subtree working back up the tree.  
3) **Post-order** => visit the root of every subtree last  
=> visit the root last, instead of starting at the root, we travel all the way down to the first Leaf and that’s where we start our traversal.  
4) **In-order** => visit left child, then root, then right child  
=> visit left child, then the root, then the right child.  
   
  
=> **Root first, left subtree, right subtree**  
=> **The data is in an order in which you can create this tree by inserting**.  
=> Each time we look at a node, we visit the node before we look at its children.  
  
=> **The data is sorted** for **In-order**, that’s why it’s called In-order.  
=> Completely visit the left side and then visit the root and then completely visit the right side.  
=> When we visit the right side, we do the same - visit the left side first then root then right side.

\* And so it’s really easy to get the sorted data once you’ve inserted them into a BST, you just have to do an In-order traversal of the tree.  
\* If you want to sort an array for example, you could insert the values into a BST and then traverse the tree In-order.  
=> **That would actually be faster than some of the sort algorithms we looked at**.  
  
=> Visit the root last.  
=> **Visit left subtree, right subtree, root.**  
=> Visit the entire left subtree for a node and then the entire right subtree for the node and then finally you visit the node.  
\* Level => levels  
\* Pre-order => root, left, right  
\* In-order => left, root, right  
\* Post-order => left, right, root  
\* **In TreeNode class**: \* **In Tree class**:  
 

**Binary Search Trees (Get, Min, Max)**  
\* **In TreeNode class**: \* **In Tree class**:  
   
\* **Most of the work is done in the TreeNode version of the method and as usual we call it from the Tree class on the root**.  
\* **It’s really easy to get the minimum and maximum values from a BST**.  
\* **In TreeNode class**: \* **In Tree class**:  
 

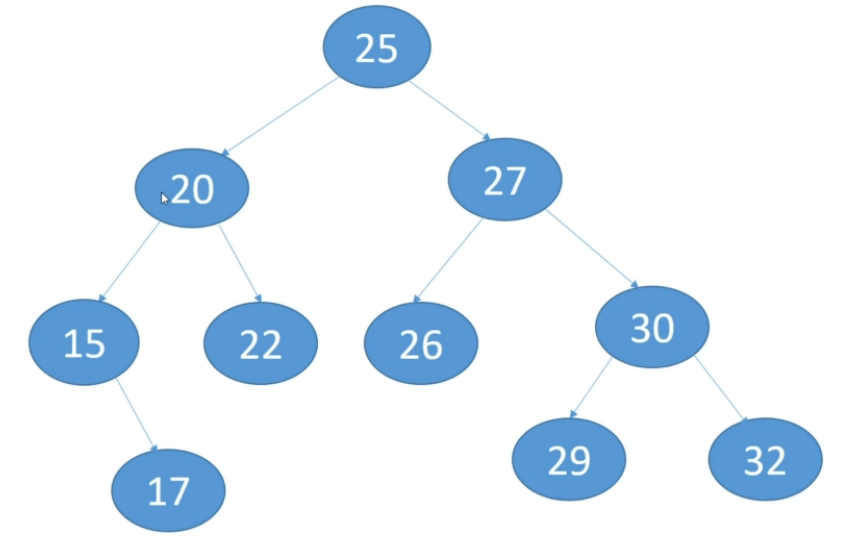
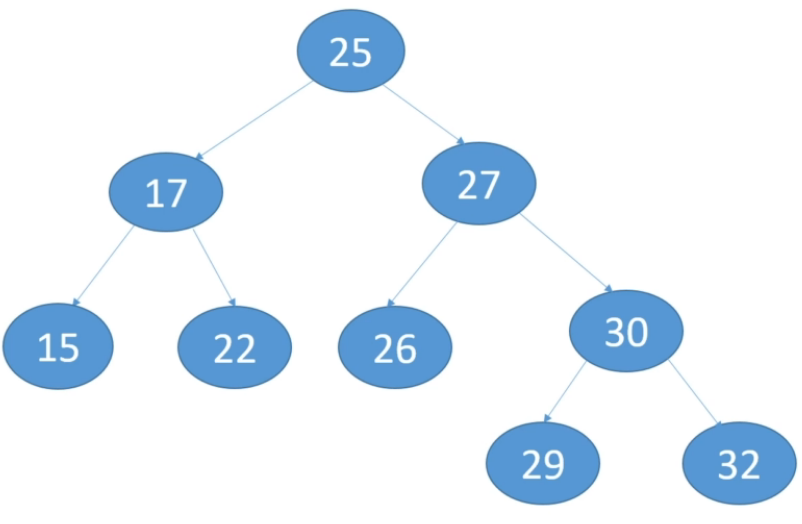
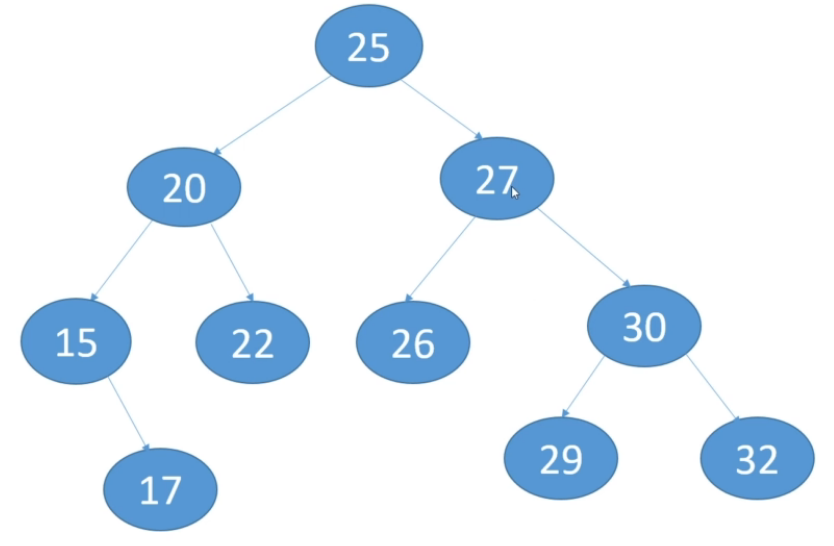
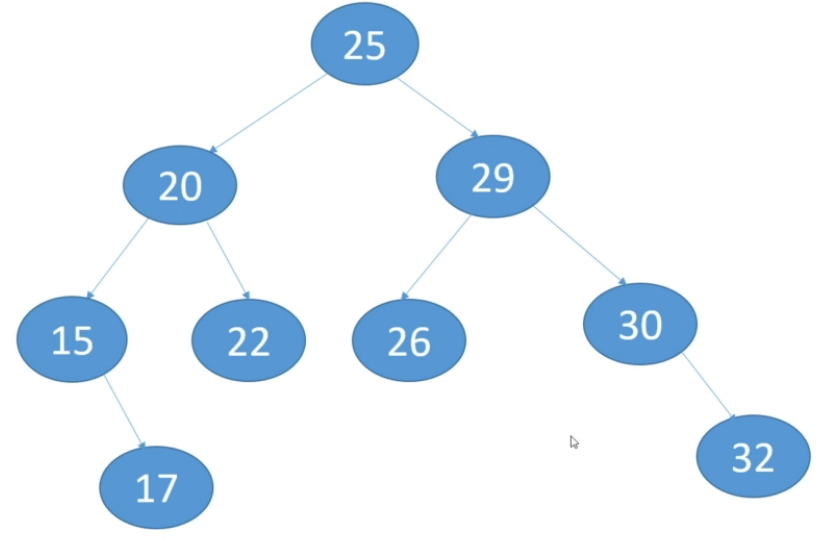
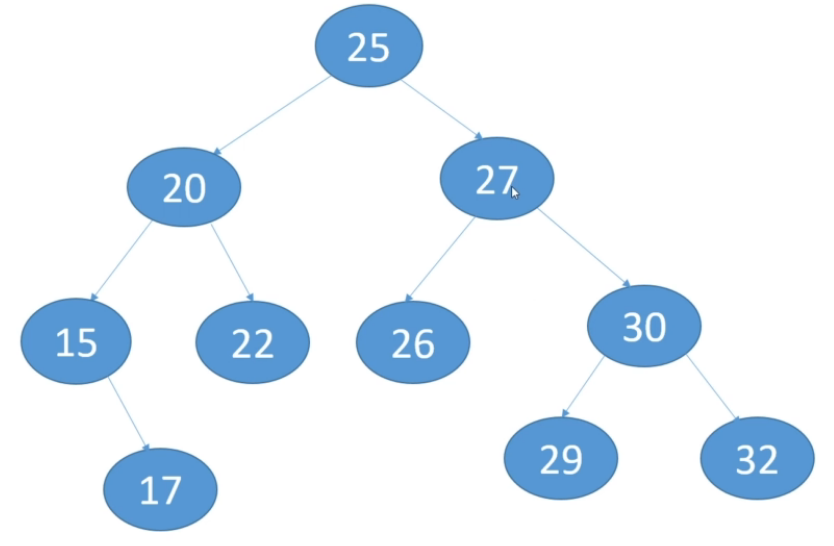
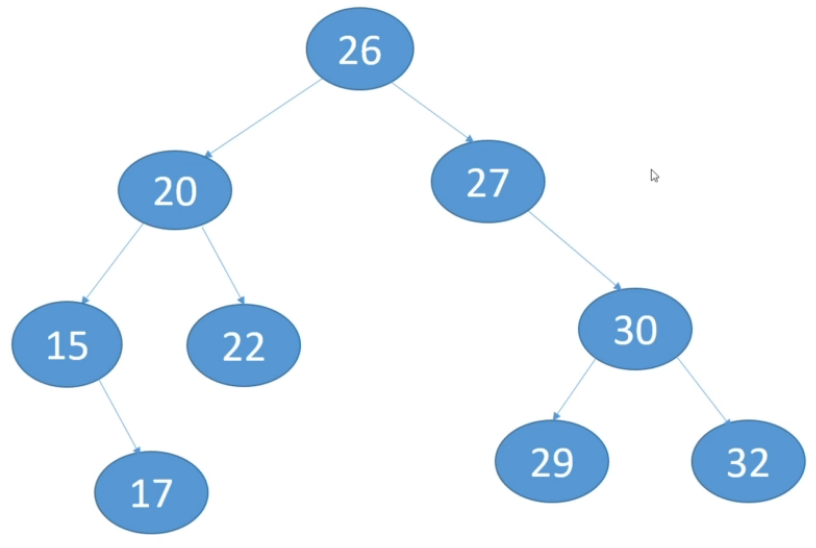
\* We could return a TreeNode here but when we implement delete(), we’re going to call the min() method and we’re going to want an integer returned, not a TreeNode - or return a TreeNode and then call the getData() on it.

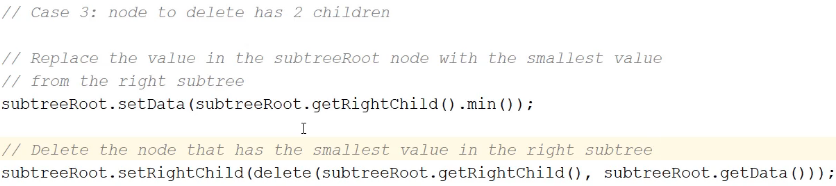
**Binary Search Trees (Delete Cases 1 and 2)**  
\* **Delete can be a little more complicated because we have children to worry about**.  
\* **Delete can be quite complex**.  
\* Because we’re dealing with a BST, meaning that Nodes can have 0 - 1 - 2 children, there are 3 possibilities:  
**1) Node is a Leaf => remove the Node  
2) Node has 1 child => the child replaces the Node  
3) Node has 2 children**  
\* The 1st case will be really easy, if a Node doesn’t have any children, we can just remove it from the Tree, we have no children to worry about.  
\* The 2nd case will be pretty easy as well, if a Node only has 1 child, the child basically replaces the Node we’re deleting.  
\* The 3rd case is complex, we’re going to look at the 3rd case in another video.  
**#1 Delete a Node that’s a Leaf**  
  
\* We added Node 17 as the Leaf that we want to delete.  
=> All we have to do is remove the Node.  
=> **Basically just null out Node 15’s right child**.  
=> **Null out the parent’s left or right child depending on which side is the Leaf that you’re deleting**.  
  
**#2 Delete a Node with 1 child**  
  
\* Let’s say we want to delete 15.  
=> **Replace the Node with the child**.  
=> It’s safe to do that because we know that everything in 20’s left subtree is < 20.

**Binary Search Trees (Implement Cases 1 and 2)**  
\* Normally, we’ve been putting a method in the TreeNode class and one in the Tree class, but for delete(), we’re going to do everything in the Tree class.  
\* What we’re ultimately going to return is the root of the Tree.  
\* If it turns out that the root is the value we want to delete, then the root will change.  
\* The delete() method returns the replacement node, if a node doesn’t have to be replaced, then it returns the same node back.  
\* The subtreeRoot is the root of the subtree that we want to search, when we start out, we want to search the entire Tree from its root because we don’t know which direction to go in yet.  
\* After checking for null, there are 3 possibilities:  
1) **search the left subtree**=> If the value is < root of the subtree, then we want to move to the root’s left child and at the end, we’re going to replace the subtree’s left child with whatever the result of the delete is. It’s possible that the left child will be replaced because the left child might be the value that we delete. If it doesn’t change, we’ll just get the same node back that’s already the left child.  
2) **search the right subtree**=> If the value is > root of the subtree, we’re going to look down the right subtree and if it turns out that the right child needs to be replaced, what will be returned is the new node and so we’ll set the right child to that new node. If it doesn’t need to be replaced, we’ll just get back the same node.  
3) **The root of the subtree is the value we’re looking for**.  
=> That means we have found the node that we want to delete and it is the subtreeRoot.  
=> We’re now going to handle the first 2 cases.  
**0 children**  
=> We just want to remove the node and so the **replacement node is null**.  
**1 child**=> The **replacement node is the child (left or right)**  
=> In our code we check if left is null and if it is, we return the right child which is null if there are 0 children so it would return null as the replacement node, or the right node as the replacement node.  
=> What that wil do ultimately because of the recursion, is the appropriate child of the parent will be nulled out because we’re in the < > comparison we’re setting the left/right children with the result of the delete(), and so if we’re returning null, that would mean the left or right child is being set to null.  
=> And so in the case of a Leaf, we’d return null, meaning we’re telling the parents to set the left/right child to null.  
=> If it does have a right child, then we want that right child to replace subtreeRoot and so we want the right child to replace the left or right child of this node’s parent. And we do the same for left.  
\* If we make it all the way down to return subtreeRoot, that means this node is not the node we want to delete and so we just return it - the subtreeRoot.  
\* That’s why this works - because we’re always returning the replacement node and if this subtreeRoot isn’t the node we want to delete, then we just want the same node. Essentially, the node will be a replacement node for itself if it’s not the node we’re deleting. And so when we return to the parent and we’re setting the left/right child, we’d essentially be setting the left/right child to the existing value.  
  
  
\* The 3rd case is when the node has 2 children - when it’s not a Leaf and it doesn’t have just 1 child.

**Binary Search Trees (Delete Case 3)**  

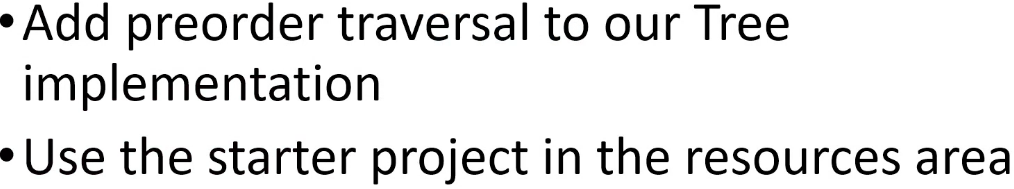

\* **We have 2 options, we can replace the node with:**:  
=> **maximum from left subtree**  
=> **minimum from right subtree**  
\* If we take the largest value in the LEFT subtree, everything else in the LEFT subtree is smaller.  
\* If we take the smallest value in the RIGHT subtree, everything else in the right subtree is larger.

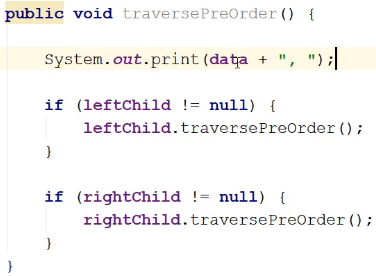
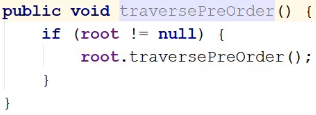
   
\* Let’s say we want to delete 20.  
=> **LEFT subtree** => largest value is 17.  
=> RIGHT subtree => largest value is 22.  
\* There are different ways you can implement deletes for no matter what situation you’re in.  
=> You can either physically move the node and then you have a bunch of rewiring to do, or we can just take the value (17 in our case) and replace the value in the existing node. And then of course we have to clean up any references to the original 17 node.   
=> If 17 had a child, because it’s the largest value, it can’t have a right child so it can only have a left child.  
=> So the replacement node can either be a Leaf or have a LEFT child.  
=> If it has a child, that child will replaces it.  
   
\* Let’s delete 27.  
\* Let’s find the smallest value in the right subtree this time.  
=> That’s 29. We null out its parent’s child.  
=> If 29 had a child, it would have to be a right child.  
=> The child would replace the 29.  
   
\* Let’s delete 25 - the root and look at the right subtree’s minimum value.  
=> We find 26. We null out its parent’s child.

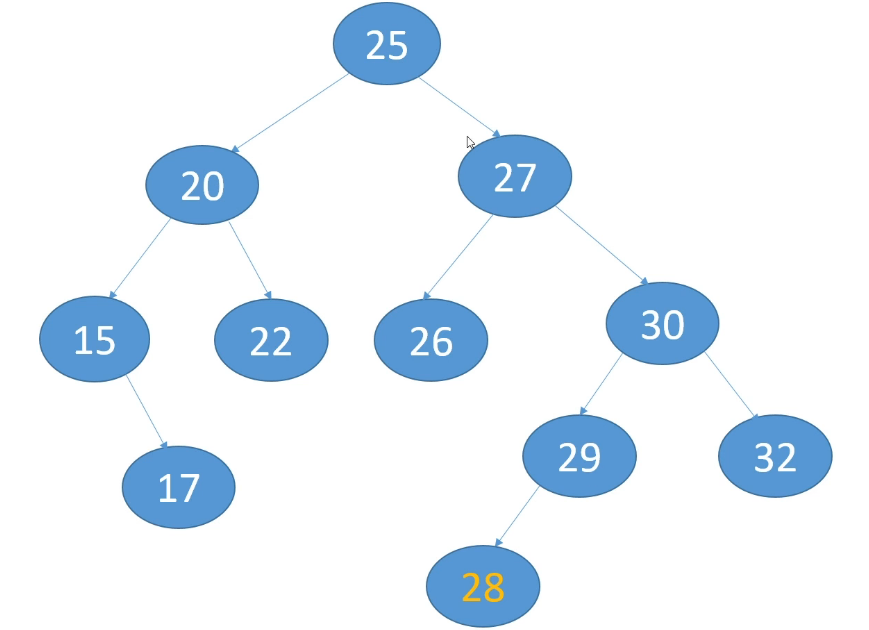
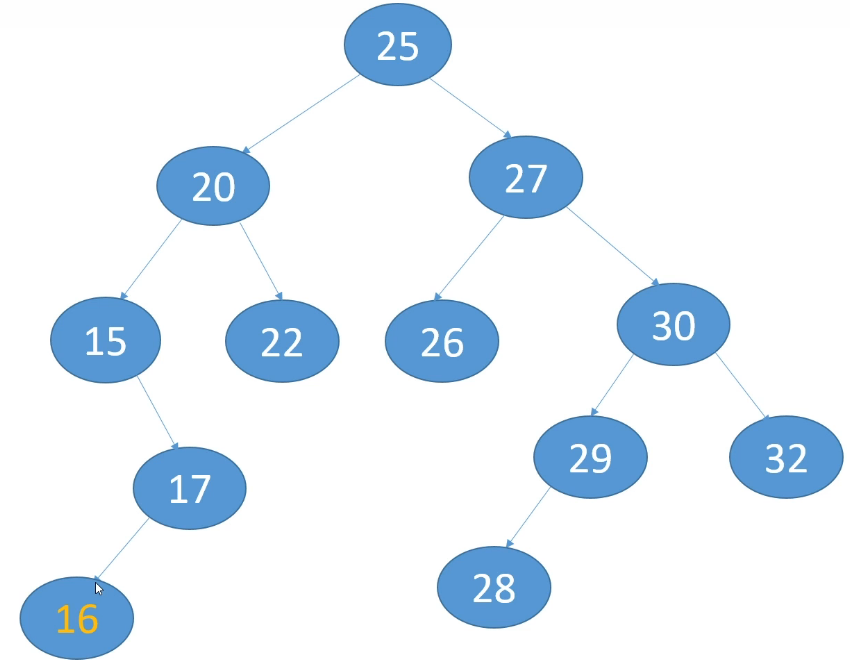
**Binary Search Trees (Implement Case 3)**  
**(implemented my own version)**\* Because we have a recursive method, it’s easy to do.  
\* We’re not actually going to rewire any nodes, we’re not going to physically take the replacement node and move it in the tree, instead we’re just going to set the value of the existing node to the value of the replacement node and then we’re going to delete the replacement node.  
\* We’re just going to delete it by calling our delete method and so we’re going to replace the right child of the node that was the deleted node.  
**(if its right subtree has a lot of levels to it, then this will end up being a redundant replacement but if we have a situation where the node that is the smallest value is the immediate child, then this will result in the right child being set to something new).**  
  
\* If we were to use an iterative implementation, the code for deleting a node with 2 children actually goes from 2 lines of code to let’s say 30 lines of code.

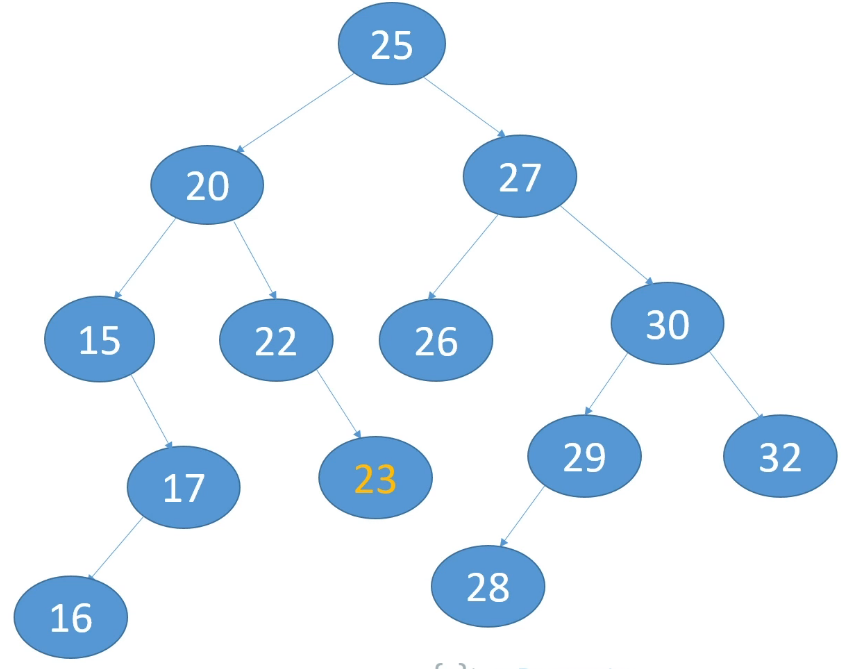
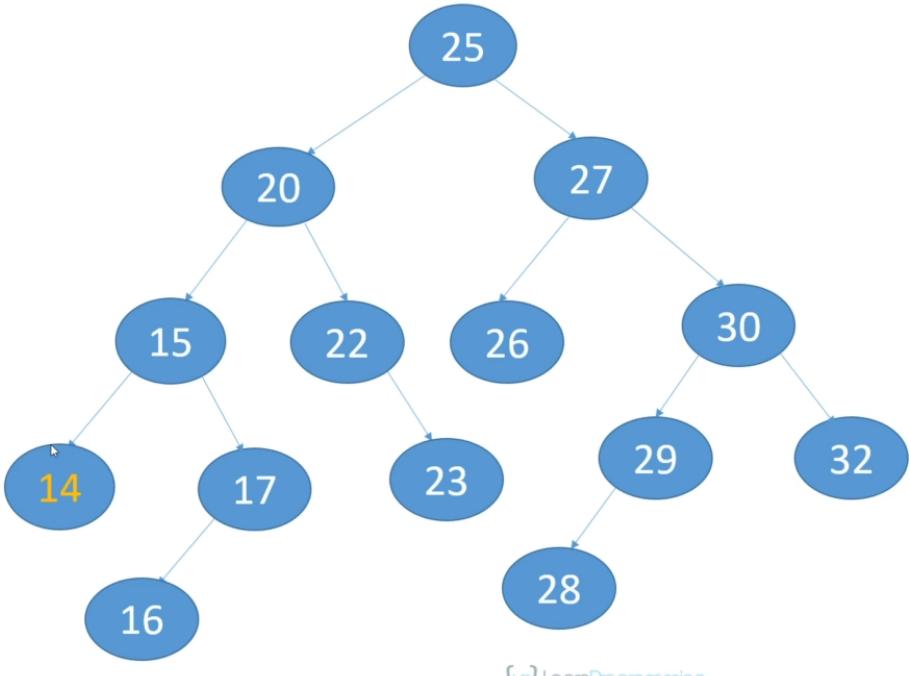
**Trees and the JDK**  
\* There aren’t a lot of classes for Trees in the JDK.  
**TreeMap** => **the one that you will probably use**  
<https://docs.oracle.com/javase/9/docs/api/java/util/TreeMap.html>  
\* The TreeMap class takes key/value pairs.  
\* Let’s say we wanted to store our employees in a Tree.  
=> We could provide a key like an integer key and then the employee object as the value.  
\* It’s a **Red-Black Tree** based **NavigableMap** **implementation**.  
=> **Red-Black trees are self balancing trees**.  
=> **After every insertion or deletion**, they check the tree to see how balanced it is.  
=> **Red-Black tree doesn’t perfectly balance the tree, but it’s good enough**.  
=> **Red-Black tree is the preferred self balancing tree these days because it has a good trade off between balancing a tree to a good enough degree and performance**.  
\* It says it guarantees:  
**O(logn)** **for**:   
**containsKey()  
get()  
put()  
remove()**  
\* **That’s because** **Red-Black tree is a Binary Search Tree**.  
\* And so as we know with BST, as long as they’re not too out of balance, you can do inserts, deletes and retrievals in O(logn).  
\* TreeMap is **not synchronized**.  
\* If we need synchronization, you can wrap it:  

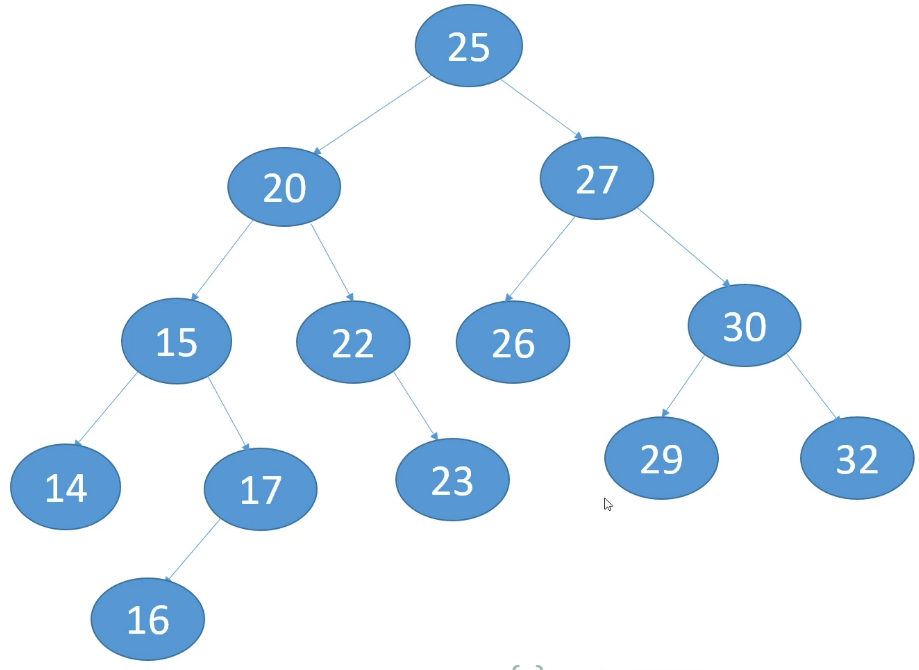
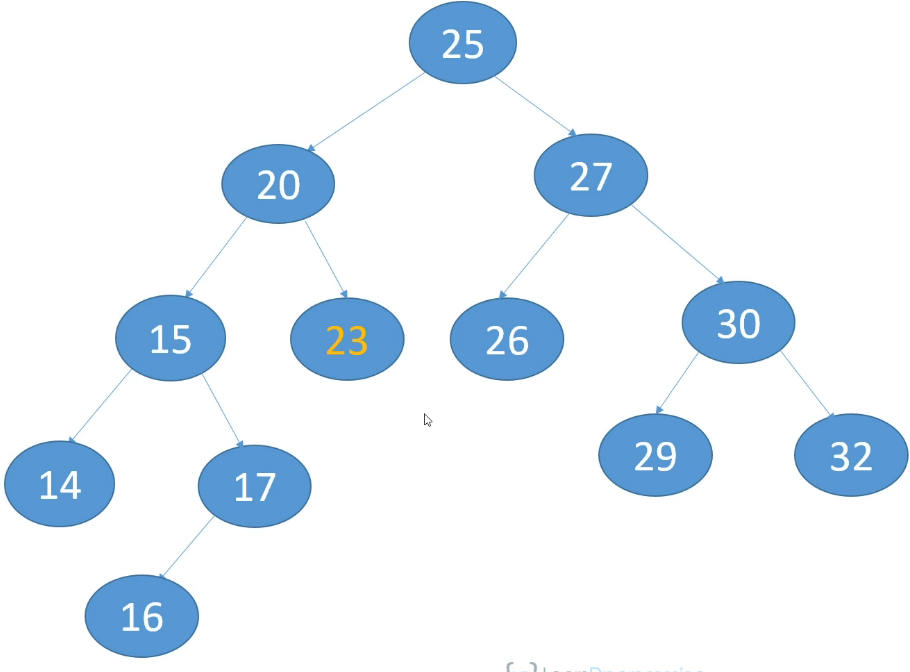
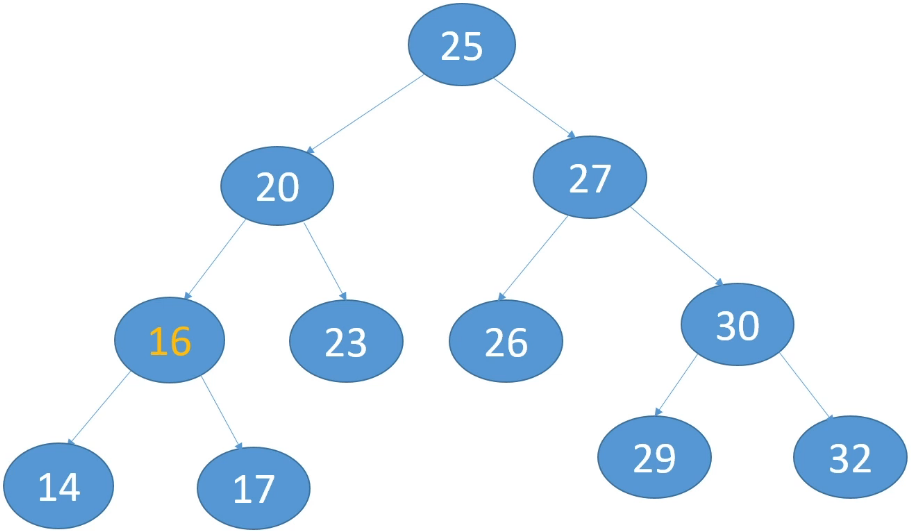
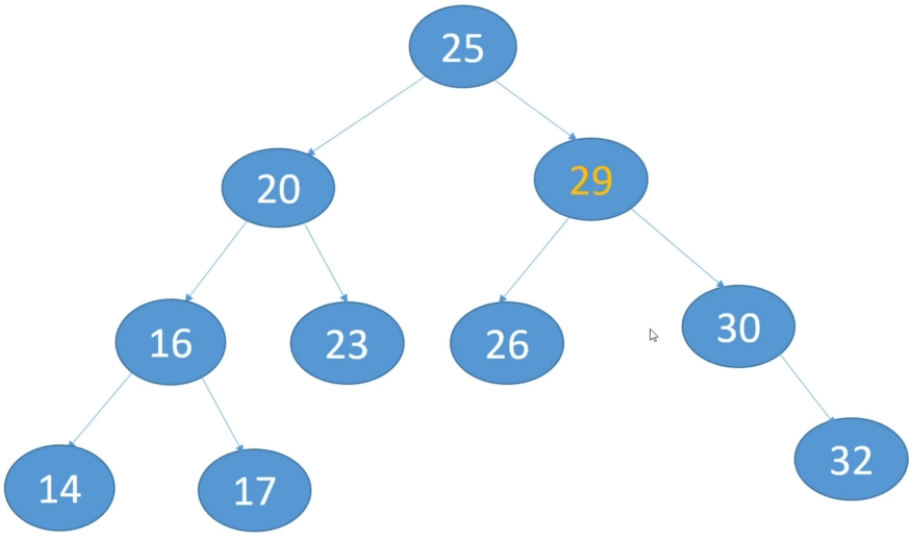

\* TreeMap descends from **AbstractMap** which **implements the Map interface**.  
\* We’re not going to go through this and we’re not going to code it because it’s just a matter of reading the method description and coding it. We’ve implemented a simple tree and you know how they work and this is just 1 implementation.  
**put()  
get()  
remove()  
replace()  
size()**  
\* And more.  
**TreeSet** => We haven’t covered Sets yet - basically a Set just means that the data structure cannot contain duplicate elements. **Set = Abstract Data Type**  
<https://docs.oracle.com/javase/9/docs/api/java/util/TreeSet.html>  
\* You can have a Tree Set, List as a Set, etc.

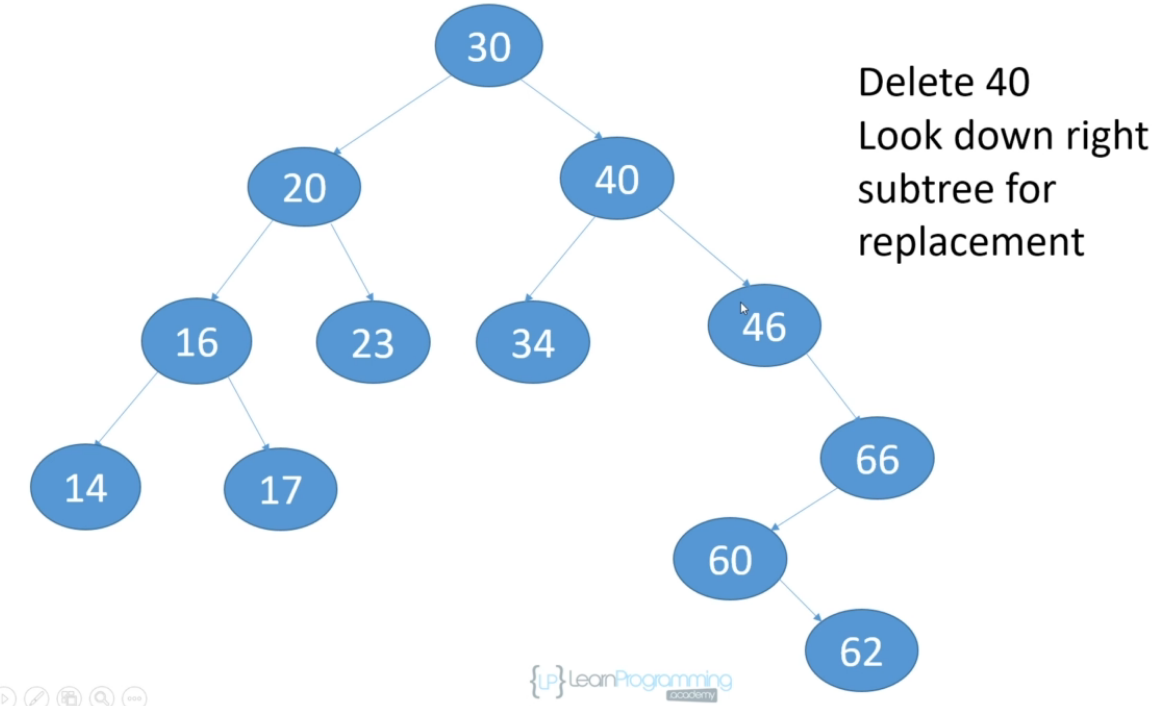
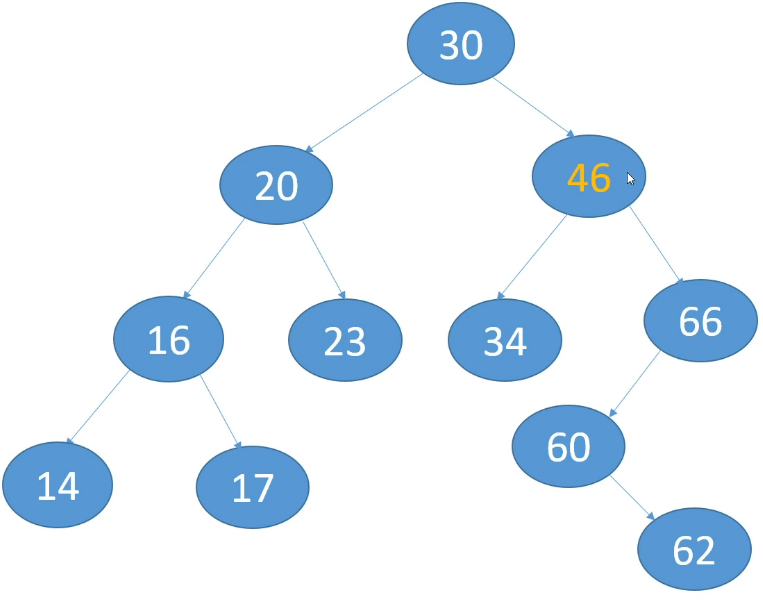
**Binary Search Trees Challenge #1**  
 **\* Pre-order traversal means that we visit the ROOT first.**

**Binary Search Trees Challenge #1 Solution**  
**(Implemented the challenge before watching + also Post-order)**  
\* **In TreeNode class:**  \* **In Tree class:**  
 

**Binary Search Trees Challenge #2**  
\* To end off this section, I’m just going to have you practice inserting and deleting Nodes from a Binary Search Tree and we’re just going to go through this using slides because that’s what you really have to understand.  
\* We’ve already coded insert, delete, get, min, max, traversal.  
\* So let’s now just practice inserting and deleting values.  
\* Insert 28 Insert 16  
 

\* Insert 23 Insert 14  
 

\* Delete 28 Delete 22  
   
\* Delete 15 (using right subtree) Delete 27 (using right subtree)  
 

\* Let’s add some new nodes before we continue: Delete 40 (using right subtree)  
 

\* That’s it for practicing insertions and deletions.  
\* If you understand how to do those, then you understand Binary Search Trees.

**Resources**  
TreeMap class javadoc  
<https://docs.oracle.com/javase/9/docs/api/java/util/TreeMap.html>  
TreeSet class javadoc  
<https://docs.oracle.com/javase/9/docs/api/java/util/TreeSet.html>