Week 10 Module 19 Trees and Binary Search Trees

In this Module 1.2

We've got a lot to cover in this module. We're going to talk about trees and binary search trees. And we're going to start by talking about the definition for trees and some of the definitions of words that we use in conjunction with trees. We're going to talk about tree storage methods and how we can take this model of what a tree is and put it into main memory. We're going to talk about the design of binary search trees and what they are, and how they can help us. We're going to talk about why we need binary search trees and then we're going to talk about tree traversals. And what happens when a binary search tree doesn't meet our requirements that we need and then we're going to talk about balance binary search trees including AVL trees, and a little bit on red-black trees. So we've got a lot to cover in this module; stick with me.

What is a Tree 1.3

The question that we have to ask first is what is a tree, and the answer to that is that it's a data structure designed to hold items in a hierarchical pattern. Now we talk about a hierarchical pattern and what we mean by that is that there's one node that's at the very, very top. And that there's a bunch of nodes that are down further, and some that are at the very very bottom. This is often used for searching; the tree can be used in a lot of for an index for a database for example. Or we might use a binary search tree, hence the name search tree, for searching information that we might store.

Every node has one and only one parent node, and that gives us that hierarchical fashion so that every node has a parent. Now of course there's one node that's a special node which is at the very, very top and it has no parent and that's called the root node. Every node will have at least some storage for pointing to children. So, every node has zero or more children; it may not actually have children but each node is going to have the capability to point to children nodes. The storage for this is actually as a pointer, just to the first node in the tree to the highest node in the tree, and that's the root. So, when we look at the class that we're going to design. It's really only going to have one pointer, very much in the way that we did a linked list is going to have one point there which is the pointer to the root of the tree and that's the highest node in the tree. It's not necessarily the first node in the tree it's not the lowest value, but we may keep it as the center; this is the highest node in the tree. We'll talk about what that looks like.

The great thing about trees is that every node looks like its own sub tree. So every node looks like a tree by itself. Now we know that naturally the root is the natural start of the entire tree, but each node could pretend to be a tree by itself. And that leads us to a discussion of course that going to come up again and again and that is that recursion is going to happen a lot in trees, because each node looks like a tree and there's one node that is the root. But we can treat all the nodes as if they were their own tree.

Some Definitions 1.4

We need to get through some definitions first. The first one that we want to talk about is a binary tree. And a binary tree is a simple tree, remember has a parent, has some children, except the binary tree has a maximum of two children. What we usually refer to them as is the left and right child only for simplicity sake we don't have any reason to call them left or right, but we do have a maximum of two children in the binary tree; doesn't mean that we have always two children, of course some nodes will

have no children and some nodes will have one child and then some nodes will have two children, but the maximum is two children.

We can talk about the size of a tree. The size of a tree is the number of nodes that are in that tree. So if we look at the size of the root node of the tree, we're looking at the size of the entire tree. If we're looking at the size of a node that has no children, that size is just one because it's just that node. And if we're looking at a size of a node with a couple of children well then we have to do some calculations to try and figure out how many nodes are actually on this tree. And we'll see that in a later slide.

We could talk about the height of the tree which is the distance or the number of links from the root to the farthest child. Now we could say that the height of a null, for example the height of a null so no node would be negative one, because it doesn't have any height it's actually a negative height. If we're talking about the height of a node with no children at that height would be zero because the distance from the node from the root to its far this child is actually itself, so that's a distance of zero. And if the root node has only one child and that no doesn't have any children, then the root nodes height is one and the only child's height is zero. So that's a simple way to look at height and give us an idea. We'll use that in a later slide to talk about the balance binary search trees.

We can talk about depth which is the distance from the node to the root and that's actually the inverse of the height. So the root node, for example, has a depth always of zero the distance from the root node to itself will always be zero, of course. And if we're talking about the most sub-child than that's the actual depth that means the depth would be equal to the height of the root node; so the farthest child from the root depth is actually equal to the roots height. So they are inverse to each other.

We could talk about a leaf node, which is a node that has no children. So the left in the right pointers for example in a binary tree would be null. And we can talk about a full node which is the maximum number of children. Obviously, if we're talking about a generic tree, where there's no maximum number of children then it's impossible to have a full node. In a binary tree, any node that has two children would be considered a full note; it can't have any additional children. So we use these definitions throughout the discussion so that we have an idea of what we're talking about. The importance here is that you take away these definitions because they're used in our language for talking about trees and binary search streets.

Tree Storage 1.5

For trees that have an unlimited number of children, so not binary search trees, but trees that have an unlimited number of children. The child storage pointers have to be stored either as an irregular as a linked list. And for that we really come up with two different forms of storage that are common for this. Now this is looking at each individual note as it exists in main memory, so in main memory if we look at the individual node what exactly does that look like.

The first one is a parent-multi-child and the parent node has an arrray of child nodes contained into it. Oftentimes, this is just a vector of child pointers: so, pointers to tree nodes for example. We would store in the node, we would store the data; would probably also store a parent pointer, but I haven't shown that here. We have a data section and then we just have an array or a vector of pointers to children for however many children we have. Obviously, if we have no children we don't want to use any storage any additional storage, so that's where the vector comes in handy because it keeps track of that and it'll adjust itself.

Now the other option is that we have a parent-child-sibling structure, which more closely represents the linked list. The parent would point to the first or the favorite child, we won't get into that discussion report to the first child and the first child would point to its sibling and the sibling or point to its sibling and so on and so forth. So what we've done is we've allowed the parent to access all the sibling, all the children, by accessing from the first child and then the first child has access to the sibling and the sibling has access to a sibling and so on and so forth. So it much more closely represents that of a linked list: the child nodes' are on a linked list and the head pointer for that link list is actually the child pointer from the parent. So we can represent this using a either a parent multi child solution or parent child sibling solution.

Now in the case where we have a value, a limited number of children, an N value as we'll call it. If we have an N value, which is the number of children that this node might have, then we can just set up the node with that number of pointers and it's not terribly difficult. So we're going to go forward and consider that we do have a limited number of children because the unlimited number of children problem doesn't often come up. The unlimited number of children problem means that each node has a variable number of children and that complexifies has the issue quite significantly, so we want to kind of avoid this. But I wanted you to see it just so that you know in case you get into trees that might need an unlimited number of children.

Binary Search Tree 1.6

The most common form of tree that we're going to work with is actually a binary search tree. And I'd like to go over that and talk about it, because we are going to use it extensively especially in this module. The binary search tree as I mentioned before, only has… It's a binary tree first off, so that means it only has a left than a right pointer. And it actually has an additional property, and the additional property helps us serve the order of the nodes, helps us decide the order of the nodes. So, in other words which one goes on the left and which one goes on the right.

And the order property in the binary search tree says that: all the values in the left sub tree are going to be values that are less than the value of this node, and all the values in the right subtree are going to be values that are greater than the value of this node. And that means whatever the data is in the data section, and I usually use just integers just to describe it and you can see that here in this in this graphic, the value of the nodes are what the value is in the data section. And if we have this templated then of course that means we need to overload the less than operator for whatever the classes that were storing, but that's outside the scope of this discussion you can look at the previous modules and operator overloading to talk about the less than operator. But if we have that then we can keep these in a regular order.

So here now in this graphic what I have is module, sorry, value twenty is that the tops of the root is value twenty that was the first to know that was inserted into the tree. So, we inserted twenty and then we inserted five, now five is the value less than twenty. So, of course five is going to be on the left hand side of twenty five would be the left child of twenty. We then either inserted three maybe, or ten, or thirty, I'm not sure which the order of insertions wasn't doesn't really matter but if we insert three then three should be on the left of twenty. Unfortunately, five is already on the left to twenty so what we can do is progress down the tree to insert three as the left child of five. So what we're doing is we're looking for a place where, when we do insertion where that value should belong but there's actually a null. And then when we find that we can go ahead and insert a new node at that location. So when we go to insert ten, ten belongs on the left there are twenty but it belongs on the right of five because it is a value

greater than five. If you take a look at that subtree, that three five ten sub tree, that is the left of twenty all the values in that sub tree are less than twenty. And all values in the left subtree of five are less than five and all the values in the right subtree of five are greater than five. So we can evaluate where an insertion should happen based on the comparison of the current value that we're trying to insert, with the value that's already inside the node. and if you take a look at the values on the right of twenty There's a value of thirty on the right of twenty but if it's left child is twenty-five. Twenty-five is a value between twenty and thirty, we can describe it that way. If we were going to insert one into this tree, the only place acceptable in this entire tree that we could insert one would be on the left of three. The only place settable to insert thirty-five, for example or a value any value greater than thirty, would be on the right sub child of thirty.

So the binary search tree holds the properties that all the left's children are less than and all the right children are greater than and that happens recursively over and over again. So we have to maintain that property, not just for the tree as a whole but for each node individually. Now if we also allow equal values then it's very common to say that the right sub child will hold all the equal values but in trees it's very common to prohibit equal values inside of the tree. So you may you may recognize that we don't allow equal values. In other words, insertion of twenty five onto this tree simply wouldn't have any effect on the tree at all.

What we need BSTs For 1.7

Now that we understand what a binary search tree is and how it's created, let's talk about what we need it for. The best thing that we can do with binary search tree, the benefit, the huge benefit that we get is that a binary search tree provides in best case scenario that searching the binary search tree will take Log of N time. So, we can go through a binary search tree in Log N times because what we've done is spread out the accesses, so that each level creates a doubling effect of how many nodes we can have on that level. So for example, if we take a search which would have let's say an array of a million elements. If we were to have a linear search through that on average it would take five hundred thousand accesses through the array to find it; we have to go through half of the array before we find the thing that we're looking for. But if we have it inside of a binary search tree the magic number there, the number of searches that it takes is twenty, not five hundred thousand and twenty because the log of a million is twenty. Each time we look at her node we're saying whether we should go to the left or go to the right or if we're lucky we found it inside the node, so inside the node once we make the decision to go to the left and go to the right or go to the right we've eliminated half of the possibilities. So, every iteration through this searching algorithm eliminates half of the possibilities from the from the set and that means that we have a log N problem. So searching in a right of a million elements would take five hundred thousand accesses; searching a binary search tree of a million elements takes twenty, which is a huge difference of course.

Insertions also should take Log of N time. Deletion should take a log of N time or close to that we're going to see that it's not exactly that in a later slide but it's pretty close. So, the benefit here is that we get for large sets, we get a really significant savings in time when we use a binary search tree to keep this in order. We can use these for in order storage of any items which can be compared using a less than operators, as long as the less than operators overloaded for that class, you can go ahead and use that. for in order storage of elements the binary search tree is a really effective and efficient solution for the storage. If we don't need to keep it in order then it's not so much a concern, we can keep it in an array, but if we do need to keep water on these values than a binary search tree is really beneficial.

BST Node Code 1.8

I wanted you to see the code here for the binary search tree node, and this helps us a little bit to understand what we're looking at when we see the binary search trees no order when we're talking about the binary search tree. What I've got here is just a data element and it looks very much like we had with the linked lists. There's just a data element and I have a templated, so it's a template type T. And then we have a pointer to the parent, pointer to the left sub-child or pointer to the rights sub-child, and that's it. That's all the storage that the entire BST node class has. I had a constructor in the same way that I have a constructor for the linked list; it's just useful for setting things up and it makes creation of nodes a little bit easier later on. I have a friend just like I did in the linked list, so the very common, a very much common with the linked list here except that we've got two pointers, a left and right child. We're going to get into this getSize class, in just a minute of the getSize function in just a minute, what it's going to do is just tell us how many nodes there are as children and this node for the node. So, we're talking about the size really that's by definition if you remember back from the definitions we did a couple of slides ago, this is just the size of this individual node. So this is the format for the BST node class and you're going to see a lot of commonalities with the linked list.

Recursion in Trees 1.9

I mentioned a couple of slides ago that recursion is going to be popular in trees and believe me it is. I wanted to give you an example of that here with that get size function inside the BST node class. So here what we're saying is, we'll start out since we are in a node already we can assume that there is one value the size at least one, so we'll start that counter at one. And then all going to do is just check to see if this node has a left: if it does then add the size of the left, if it has a right then add the size of the right and then return count. And if you take a look at that count, plus equals left arrow get size, that's a recursive function call to a different node. So, what we're doing is checking to see if that node exists first, because we don't want to de-ref null of course, if that no does exist then we'll call its get size function and we'll add its size to our size. So if we have any children, we can assume that our size is the sum of the child sizes plus one.

And that's really all there is to recursion in trees, trying to do this with iteration is really, really hard because you have to keep track of what level inside the tree you're on what depth you're at and then move backwards and forwards. Remember if, for example, there's no… There's no association; there's no pointers between a child node and its cousin node. So, if we have to go back up to an uncle node which would be the parents sibling node, that takes quite an effort and even worse if this these cousins are separated significantly by more and more uncles and aunts in the family tree that looks really crazy. But with recursion it's rather simple because we can just call the recursive function to take a pointer to the child nodes, or in this case we embed it inside the node class and we can use the getSize function for the child nodes inside the parents get size function. So, recursion is going to happen quite often in trees.

Tree Traversals 1.10

When we're working with binary search trees, there are times that we want to process every node in the tree. And the way that we do that, and not by searching I mean we actually want to go into each node in the tree and do some activity… If we're going to do that we have a number of ways that we can do that and the order of it really depends on how we want to go through the tree. Now if we're going to do the same thing to every node really doesn't matter, we can just do it using any of these orders, but if we care about the way that the processing happens. So, for search for example on a file system, which is designed as a tree, on a search on a file system we probably want to search the higher stuff first and search the later stuff later. That's called a level order traversal, but we may want to do different types of

traversals which are: in-order, pre-order, or post order depending on when we want to search this note or when we want to process I'll call it, this node.

So what we're asking here is if we look at it in a recursive fashion, when do we want to search the root node. So, we start the process does that mean that we search the root node first or does that mean we search the leftmost node, the minimum node first. So, in a binary search tree really in any binary tree, we have to decide what the order of traversal is.

So, in-order traversal processes the left nodes recursively first and then it processes the so-called this node and then a processes the right nodes. So, the pre-order traversal would just be that we do this node first and then the left and then the right. And the post-order would be that we do the left and then the right and then the this; so it really decides when we process the this node. Level order traversal processes it based on depth and this might also be known as a breadth first search, it's actually an interesting problem but what we're doing is we're looking at the tree and going out word on the tree. So we're not going down one side and then coming back and doing the other side we're going outward on the levels of the tree. And we'll see the code for this and it will become a lot clearer in the next slides.

Implementation 1.11

So here I want to show you the code for the in-order, the pre-order, and the post-order traversals in the tree. And what you can see is that we've done, we're taking in a pointer to a node. That may not be the best solution to, in fact we might have a driver function which calls this for the root, but we do need to do this recursively so it will take in a node pointer to the node that we care about. And what you can see with these three pieces of code is that in reality they are the same; the only difference is where we process. And in this case it's just an output statement where we process the current node, the node's data section. So, in the in-order traversal it's processed in between the left and the right processing. In the pre-order traversal, it's processes the head of the left and right processing. And the post-order traversal it's done after. And since this is a recursive function that really is going to consequently change the output of the tree based on how we want to go through each individual node.

Level Order Traversal 1.12

The level order traversal gets even more complex and I'm going to tie this into the previous module that we did which was talking about stacks and queue's. And here we're using a queue to store the binary search tree pointers, the node pointers. So what we're doing is we start off with the level order traversal by pushing the root node pointer so we push the root node pointer on to the tree, onto the queue excuse me, and then as long as the queue is not empty what we do is look at the front node look at the top of the of the queue. And then pop that off process that node and then push the left and push the right as long as there are not know. What that does is process the root node first of course but then a process is the left child. And if the left child of the root has any children those get pushed onto the queue after the right child of the root. So what we're doing is processing the tree from the top down and outwards. Which is known as a breadth first search, we're going to breadth of the tree first before we go to the next level. So, you can see that this level order traversal tends to operate on the higher nodes the nodes with the less depth first. And it's just another way of going through the same tree and it might help us if we're searching a very, very large tree or for doing some operation on a very, very large tree; it might help us get the nodes that we care about most done first.

Traversal Results 1.13

I wanted to show you what the traversal results look like, so I'll come back to that same tree that we came that we showed you earlier with the twenty at the root node. And what I want you to see is that the pre-order, in-order, post-order and level order traversals really produce very, very different outputs. And so what we're doing is we're saying that the in-order traversal would actually process all the left nodes first and then the node, and then the right node first. The pre-order traversal tends to process the root node first and then it processes the left and processes the right recursively, and that's the important characteristics here.

So, take a look at the pre-order traversal; it means we process twenty first of course because we're doing the pre-processing. So we process twenty first, and then recursively process the left sub-child which means a five comes out next, but that means recursively we have to do the left sub-child of five before we can go back and do anything else. Three comes out and because three doesn't have any left child or right child, we're done and we move back to five in which case we have to process the right sub-child of five. We only process the left; it's now time to process the right. So, if we if you take a look back at the code after we're done processing the left child of five, we're going to have to process the right child of five, which is ten and then we're done because it has no left and no right child. When we come back the five, we're done with five, we go back to twenty and we still have to process the right children. And so that's the way that the pre-order traversal is going to operate. The output would be twenty-five, three, ten, thirty, and twenty-five because we're doing it pre-order.

In-order; you can see that this actually is in order so it's sorted if you will. We're processing all the left children first; when we go to process the left child of three since it has no left child we can simply process the node at three and then process its right child. It has no right child, so we process three first. In other words, the minimum value is the value processed first which makes sense. Likewise, the maximum value is the value that's processed last. So, the output from the in-order traversal would be three, five, ten, twenty, twenty-five and thirty. Just simply because we're going through this in order and since it is a binary search tree it's kept in order.

The level-order is the most interesting one here. Which is that we do twenty first and then if you look at the queue, when we push twenty we're then going to push five and thirty, which are the nodes on the left and right child of twenty. When we're done processing twenty and we start processing five, we'll push three and ten but three and ten are actually after thirty on the queue. So, if you look internally at the queue, what we're going to see is three, thirty, three, ten, and then it's not until we're finished with ten and we process twenty-five that were done with the entire tree. So the level-order traversal would be twenty, five, thirty, three, ten, and then twenty-five and that is exactly in the order of the levels if you look at the picture.

Insertion into Trees 1.14

Insertion into a tree is not terribly difficult. The overview is kind of that we check to see if the tree is empty and if it is of course, this is just the first note on the tree will create it that way. If it's not then we have to go about and finding the insertion point. So, what I've done here is given you the code, the first one is just the first thing is just an if statement to check to see if the tree is empty and if it is just pushing it on to the tree. If the trees down empty it's time to find the insertion point and I do that by using my two pointers temp previous. And again, this should look very similar to what we did with linked lists, in that eventually temp will become null; temp will be the null pointer. And we're going to keep previous as one node back from where we were. when temp falls off into null, we can look at previous and decide

if we're going to push on to the left or the right sub-child. Now inside that while loop what we're doing is comparing the item that we're going to try to insert with the data that's inside the node. And in doing so we can decide if we should go to the left or we should go to the right, based on whether or not this is less than. So if we go to the left and temp becomes null, than the previous node is the node we have to update to point to the new node that we create. And then that last if statement there just does that same check again to decide whether this should be a left child of the node or whether it should be a right child of the node based on whether it's less than an order greater than the node.

There's no real recursion in this in this algorithm; all it is, is a loop to find the null. And then once we found the know all to go to the node immediately before that we saw immediately before that an add the node on as a left child or right child. So as not it's not terribly difficult code; It does involve the little bit of complexity of pointers. But if we keep track of where we were just a moment ago using the previous pointer, then when the temp pointer goes know all we know where the nodes should be connected to.

Removal From a Tree 1.15

So if we look at removing from a tree element; insertion is easy, removal does require a little bit more work. So removing from a tree is easy if the node is empty. If the node is a leaf node, then just remove it and update the parent to point to the null pointer. So there's not much work that has to be done if the node is a leaf node. If the node has just one child then that's not difficult either, because we can promote that child and the child will match the characteristics of the parent. If, for example, it's a left child then the left child is going to be promoted; it's going to have a value less than the parent but the parent relationship to the grandparent will not change and so that should all still work out well.

The problem comes when the node has two children, and we have to choose what's called a candidate replacement. So now we're not going to accept the fact that we can promote; we can't promote a child to the parent position because what if that child has children of its own. We can't just move that child up; what do we do with its children and what do we do with the other child of the grandparents. What we're going to have to do is decide whether we choose the maximum of the right sub-child so in other words go right and then go left, left, left, left, left, left, left, left, and we'll end up with is the maximum value of the… Sorry the minimum value of the right subtree. Or what we can do if we're talking about going to the left is we can go to the left, and then right, right, right, right, right, or we can go to the right and left, left, left, left.

Either way that we decide to go, what we're choosing is either the previous node from the node that we're going to delete or the next node from the node that we're going to delete in order traversal. So, if we look at this traversal in order, it would either be the node immediately before which would be the minimum of the left-most, sorry the minimum of the right subtree or sorry the maximum of the left subtree or would be the minimum of the right subtree. In other words, we're going to delete a node and let's choose to promote, the one of the nodes that are either the on the left or are on the right.
So, one of the nodes adjacent to the node that we're going to be deleting in value, is going to be promoted to the current node. So, that's the way that we're going to handle this; we're either going to go to the left and then go right, right, right, right, right, to find the find the maximum node. Or we're going to go to the right and then go left, left, left, left, left, to find the minimum node. So, the point is that we can actually process this by promoting one of the children and then deleting that child instead of deleting the actual node that we're going to trying to leave. We're going to see that in just a just a minute.

Removal, Given the Node, No Children 1.16

Here's the code for deleting a node if it has no children. And what we're given is a pointer to this node, actually it's a reference to a pointer to this node. The reason for that is that we're going to be changing the pointer; we actually make a change so that temp pointer. So, what we do is to say if temps left is equal to the null pointer and temps right is equal to the null pointer: it just it has no children. So, there's no there's no work that really has to be done here. We'll find out if the parent is the null pointer, if it is then this is the root node of the tree; it has no children if the last note on the tree were done. So, we can just set root equal to null pointer. So, if we're removing the last node in the tree just remove the node. And then of course will have to delete it later.

And then we'll check to see if the parents left is equal to this node. So, if the parents left is equal to the node that we're looking at, so we go up a level and then we go back to the left then we're going to have to set the parents left pointer equal to the null pointer. Remember this node has no children at all; so if the parents left is this node that we're looking at then let's delete the parents left. If the parent's right, obviously if the parent if it's not the parents left in the parent exists then it's this must be the parent's right pointer. So, if the parent's right pointer is this pointer, then we update that parent's right pointer to get rid of this node and then we actually get rid of the node. So, in the no children case this is actually pretty simple. We basically just update the parent's pointer so that it points to null and then delete the node.

Removal, Given the Node, One Child 1.17

If we have one child, then what we want to do is promote that one child's data value and delete the child node. So, this assumes that we've already done the check that we did in the previous slide to see if both children don't exist. If we don't have any children then we're going to take care of it with the previous slide's code but now we recognize that there is one child in either; we have a left child or we have a right child. And what we're going to do is we're going to take a look and say if temps left is equal to a null pointer, so we don't have a left child that means by default we must have a right child. And the reason for that is that if we didn't we would have gotten caught in the if statement up above from the previous slide.

So, we're assuming now that we have this right child and we're going to want to do is we're going to want to get it to delete pointer. This is just another pointer to say that this is the node that we're going to delete. And what we can do is promote the data value up to the to the parent so now we copy to delete data up to temps' data, so we're copying the value from that right node up into the parent node. And then we'll copy the pointers up into the parent node, and then what we can do is make the pointer from our parent point to the new node that we're going to delete. And then we can delete the two delete values, so in other words we're going to copy the value up and then we're going to delete the child node that does exist. So that makes it a little bit easier that we don't necessarily have to update the parent pointers; all we have to do is make the data values bring those pointers up to the to the object that we're working in right now. So the pointer for temp is actually going to remain exactly where it is; the node temp is remain exactly where it is. And what we're going to do is move all the pointers and the data values from the child nodes up to the new temp. We do have to take into account if the temps left pointer exists, that its parent now is going to change. So, if we have a node on the left of the to delete, we're going to have to change that to deletes parent pointer to deletes left parent pointer to the appropriate new value. So, keeping in mind that we're not actually going to delete the node that we

are trying to remove, we're actually just going to promote the value and delete the node that's to the right, and this case is to the right.

We're going to delete the node to the right and promoted it's data value. If we're deciding that we're going to go to the left then it's the exact same thing just dealing with the left pointers instead of the right pointers and keeping all that in mind. So it doesn't really require a lot of different code it's really just flipping of the left versus the right. So it's just almost a copy of the same thing. So that's if we have one child as opposed to no children. The problem with two children is well, bigger and we're going to see that in just a minute.

Removal, Given the Node, Two Children 1.18

So here we are in the else condition. And in the else condition we've already checked to see if we have no children or if the left child is equal to null, or if the right child is equal to null. So, we've already decided that what we ended up with is that we have two children of the Node that we're trying to delete. And that's not going to be the easiest solution, again we're going to either have to decide to find the minimum of the right subtree or the maximum of the left subtree. So, we have to make a decision and this is just a programmer's decision; it's entirely personal preference. Now for me it's easiest to find the minimum value of the right subtree; so, to go right and then go left, left, left, left. And ultimately what I'm going to decide to do is do that, go right and then go left, left, left, left, left, and take that last value that I find, when the left's left pointer, when the the items left pointer is equal to null that's the minimum value of that right subtree. And what I'm gonna do is copy the data value up and then delete that node.

So this is recursion; this is a recursive algorithm that simply says once we found the data value that we want to delete which is not the data value that somebody told us to delete. We are going to find the minimum value of the right subtree, we're to go right, left, left, left, left find value and promote that value but then we're going to have duplicates. And so what we do is remove that value that's down further in the tree. Now here, I've guaranteed that it's not infinite recursion. And the way that I'm guaranteed that is by making sure that the minimum of the right subtree is the one that I'm using. And if I'm using the minimum of the right subtree it can't have a left child. It cannot have a left child because if it did that would be the minimum of the the right subtree.

So here we've proven that we can use recursion, but we're only going to go recursion on one level further. Because what we're actually going to be doing is removing the minimum of the right subtree, which means that by definition it can only have it at most one child, hopefully it has zero. But even if it does have one we solve that in a previous example and we don't have to worry about infinite recursion from that sense. So, in fact what seemed to be a very complex problem of removal of the node given that it has two children, actually turns out pretty easy because we're just going to deal with deleting some other node and promoting its data value.

When BSTs Fail 1.19

We know that in a binary search tree, best case scenario, the binary search tree is going to result in big O of Log of N time for everything. So insertion, searches, removals, all that's going to take a Log N time. Unfortunately, where BST's fail is if the insertions that we're doing are already in order. And if they are, then all the insertions that we do are going to result in right hand side operations only. So we're going to insert five and then we're going to insert ten as the right sub child of five, and then we're going to insert fifteen as the right sub child of ten, and then going to insert twenty is the right sub child of fifteen and

then we are insert twenty five as the rights of child of twenty. And what we end up with is all right children and nobody on the left; the left is all equal to null. And so what we end up with really is a linked list and we know that searching a linked list is linear time, not Log N, linear. So going back to our previous example of one million elements now we've got to search five hundred thousand elements because there's no left elements at all; they're all right elements. And so this doesn't really work out very well in the, in the easy binary search tree solution because the operations that we're doing are a little bit too simplistic. What we really need to make sure is that we're not constantly using the right subtree. That we actually end up using those left pointers to some extent and that we have a wider breadth than we do depth, or at least equal breadth and depth.

Balanced Binary Search Trees 1.20

The solution of course is a balanced binary search tree. And the balanced binary search tree guarantees that we use Log N or that we have Log N search time. To do that we need to do some additional work and insertions and removals. The d binary search tree does protect the big O Log N insertion time. Unfortunately as a result, insertions and removals do take a little bit longer, but they're still going to be close to Log N. So, we're not changing this to a linear insertion like we would have with a linked list or within array; god help us. But it does take longer than big theta Log N so it's still going to be big O of Log of N, but it's actually only something closer to like two Log of N so the theta to does change but the big O pretty much stays the same.

There's two types of very popular balance binary search trees. There's one that's really easy to understand but the performance isn't all that great, called an AVL tree, and then there's one that's really hard to understand the but the performance is fabulous and that's called a red-black tree. If you were create a tree, if you were to use a tree inside STL. So, the STL actually does have two trees: one is called a set and the other is called a map. And of course, there are multi-multi-mapped trees in case we're interested in that. But the STL set and the STL map are actually implemented as red-black trees. So here we actually have something that exists in real life or in our real code life at least that we're that we're studying at this point.

AVL Trees 1.21

AVL trees are named for their creators and it's two guys named Adelson Belsky and Landis. What they do is they record a height for each node of the tree and it's a balanced binary search tree because the AVL tree puts in a stipulation that the height of the left and right subtrees can differ by no more than one. So the heights, by definition, can differ by no more than one and that means that we never get to the point where we end up with a linked list where everything is always going to the right or even worse everything's going to the left, but the point is that the heights are restricted. Now one thing that the AVL tree does, or actually the point of the AVL tree is that when we determine that the heights differ by more than one then it's time for a rotation to rebalance the subtree. And we're going to talk about rotations in just a minute, but the heights of the trees should cause a rotation to happen. If they differ by more than one, we're going to have to do a rotation to reproach to choose a better route of that subtree so that we can balance out the tree into.

Good AVL Trees 1.22

Here's an example of a good AVL tree. So what we did was we did some insertions, like we inserted the value of ten and we inserted five and twenty and three, and those all got inserted onto the tree. What we have here is that ten has a height of two. Now how we calculated that was by looking at the height

of the left sub child and the height of the rights sub child, and determining that the maximum of those two heights is one and then adding one to that. So that's why the height of the node at three, for example, is a zero because the height of the left sub child is a negative one and the height of the right sub child is a negative one and we can add one to that, and add one to the maximum of that which is the same value, and we get zero.

So what we can do is say that this is a good AVL tree because the height of all of the nodes are, the height of all the left sub child for every node in the height of all the right sub child for every node, is equal to a value that differing by only no more than one. So that the largest difference here is actually if we get to the root node, if we take a look at the root node, the height of the left sub child five is one and the height of the right sub child is zero, the value twenty. We can also look at that node five which has a height of the left sub child of zero and the height of the right sub child as negative one. So, that's a difference of only, of only one. So, in no case do we differ from the left sub child to the right sub child by more than one and so this is an acceptable AVL tree.

Bad AVL Trees 1.23

Here's an example of a really bad AVL tree, or rather I should say one that doesn't meet the AVL properties, right. We call it a bad AVL tree but it just doesn't meet the AVL property. So, what happened was we inserted ten, we inserted five, we inserted three, which sort of looks like we had it before except the ten's left child is five and five's left child is three. Now the height of three is zero, which means the height of its left child is negative one, the head of its right child is negative one; that's fine. They differ by no more than one. The height of the node at five is one; the height of its left sub child is zero, the height of its right sub child is a negative one. So that differs by no more than one.

Where the problem comes as at that root node ten there, which is a height of two. And the height of two; the left sub child, five, has a height of one but the right sub child has a height of negative one. So the difference there is two and now we've got a problem. And we can see this starting to form the same classic problem where we had the right, right, right, right, right problem from the previous slide, where all the nodes are being inserted as greater values. Here the nodes are all being inserted lower values, lesser values, so it's all going to the left. Regardless, if we let this propagate this is going to end up as a linked list, we're going to have the really poor insertion and searching times that we had with linked lists.

Another example of a bad AVL tree here. We can see that we did, it looks like the insertion of ten, five, three, and twenty and that all was fine; everything was okay from the same as the previous example. Except now we take the added component of adding four, so here we added four onto the right sub child of three. And what happened was the node at four is balanced, left sub child is negative one, right sub child is negative one. The height at three is balanced; left sub child is negative one, right sub child zero. The height at five, well that one we've got a bit of an issue here because the height of the node five, the left sub child is one and the right sub child is a negative one. So, that causes us a little bit of heartache we're going to have to deal with that. Unfortunately, we have to deal with these as two different problems, because the solutions that we use are different based on what whether or not, well we'll see them in a later slide, but they're based on a little bit of difference in how we did the last insertion. Either way these trees do not meet the AVL property. So, we do have to do some work to get them to balance again.

Rotation Solutions 1.24

Let's look at a problem where we have a grandparent, parent, child kind of situation that's unbalanced. So, we have a grandparent node which has a child of a parent node, and the parent node has a child of the child node. So, we have that sort of situation and it doesn't really matter whether we're doing insertions on the left or insertions on the right, but the grandparent parent child situation is set up here so that we have an unbalanced situation. If the left parent's left sub child is greater than the left parent's right sub child, or if we have the heights what we're talking about is the height, or we have the right parent's right sub child is greater than the right parents' left sub child. Or what I like to call it is either an outside-outside insertion versus an outside-inside insertion; outside-inside is a different problem, outside-outside.

So if we go right right, that's one condition; if we go left left, that's one condition and that's solved by a single rotation. And what we're talking about with the single rotation is taking the parents and making it the new grandparent position. So, in the case that I have described here, we've got a node twenty and it's left child is ten and the left child of ten is three and what we've got is we're going to promote ten. Now when we promote ten, it's like picking up the node ten and the rest of the node sort of drop like Christmas balls. So,if we pick up ten, the twenty node converts (turns into) the right node of ten; the twenty node becomes the right note of ten.

So the reason that that works, picking it up and then having ten's right node become twenty is because in this condition, ten doesn't have a right child. What would happen if ten did have a right child? Well if ten didn't have a right child, then the left child of twenty would become empty; it would become null pointer. Because we're picking up then the left child of twenty is no longer important, because that was the pointer that pointed to ten, but we don't need that anymore. So, the perfect solution is to make the left child of twenty, actually point to the right child of ten.

And if we do that remembering the fact that all the nodes to that are greater than ten would be on the right of ten. Then it's important to recognize that this still holds the binary search tree property, because when we pick up ten and its right pointer now points to twenty; that's a value greater than. And remembering that all the values that were greater than ten were on the right side of ten. So, if we make the left pointer of twenty point to the values that are greater than ten (were greater than ten before) then we still have that binary search tree property maintained and everything still works out the way that we wanted to.

So, this is what's known as a single rotation, in this case actually it's called a single clockwise rotation because we're kind of going clockwise. and what we can do is recognize that this works. Now it only works in the case of an outside-outside, so if the heavier node that the node with the higher height, the bigger height, the greater height is right right or left the left this work. But if we have a problem, we do have a problem, if we're going left rights or if we're going right left and that's the higher node. So, we're going to see that as another solution in the next slide.

Double Rotation 1.25

What happens when we do have that outside-inside insertion? And in either way whether we're talking about left right or whether we're talking about right left, what we're saying is that the heavier node is a different child than what we decided on the parent. So when we're looking at the grandparent node, in this case we have twenty: the left child of twenty is ten, the right child of ten is fifteen. We've got is a problem in which a single rotation would not solve this. What we really need to do is reorganize these as

a double rotation. And a double rotation is actually pretty simple because all we need to do is first do a counter clockwise rotation around ten, so that we end up in the same situation.

So first we'll do a counter clockwise rotation about ten. Which means the twenty's left sub child will be fifteen, and then fifteen's left sub child will be ten. So, initially we sort of make the problem not go away but we put the problem in a solution that we can solve because we're going to generate an outside-outside problem. So initially the single rotation would not solve this at all. If we only used a single rotation we would still have the problem of, even if we promoted ten to be the root its right sub tree would be twenty and the left sub child of twenty would be fifteen. So that's doesn't solve the problem but if we do a single rotation about the parent, not about the grandparent but about the parent, if do the single rotation about the parent first. Then what we end up with is a problem that we can solve. So we end up with twenty; its left child being fifteen, fifteen's left child being ten and that's just a single rotation that we can solve. The end point is that we have fifteen as the root; its left some child is ten and its rights of child is twenty.

And that's the way that double rotation is done. So in fact if you look at the code, a double rotation is actually done by two calls to the single rotation function: once clockwise and the others counter-clockwise so there's not really much work that has to be done there. But the double rotation problem does exist and we do have to recognize it.

Red-Black Trees 1.26

Red-black trees are designed to try and solve the problem of the balanced binary search tree. One of the issues that comes up with AVL trees that we kind of glossed over Is the need to calculate those heights. And unfortunately, whenever we do an insertion, the heights are going to change. Whenever we do a removal, the heights are going to change. Whenever we rebalance, the heights are really going to change, so if we do a rotation, really going to change. And unfortunately, that means that we're going to have to calculate those heights which takes Log of N time, so really we're adding Log N time on top of Log N time and that's two Log N for big Theta. It's not terrible, by no means, I mean it would be much worse to put this into a linked list but we can do a little bit better.

So the red-black tree aims to avoid the problem of having to go back and recalculate heights every time. And the way that it does that is by four laws and the four laws are kind of cryptic and really doesn't make sense until you see it. So, we're going to see it in a later slide, but let's go over the laws and just get a good idea.

So, the basis of the laws is that all nodes… First law, one, right. All nodes are colored either red or black. Well that's not hard because all we have to do is put a Boolean in there, in each node, to say it's either red or black. It's just a Boolean value so that's very simple. The root is always black, and for that we just say if we ever find that the root is red we'll just recolor it to black; it has no impact on anything. So law two is very easy to comply with; if the root is ever turned red, we just make it black again. Here's where life gets a little bit tougher, but it's good to keep in mind that the purpose of these laws is to determine when a rotation is going to be necessary. So, what we'd like to do is try and determine when the rotation is necessary. Keep that in mind as we go through laws three and four. Law three says that a red node cannot under any circumstances have a red child. Now, don't read any more into it than just that, because it doesn't say that a black node can't have a red child and it does not say that a black node can't have a black child. All it says is that if we have a red nose, we cannot have a red child. So that's the only restriction there. What happens if we try to insert a red node onto a red child well then we're going to

recognize that it needs to do a rotation or we need to do a rebalance or something is going to have to change. Perhaps the color of the parent node is simple enough that we can just recolor it to black; maybe it's the root. But we'll take care of that as we go through more and we look at the actual code for this. The core of this is that law three: a red node can never have a red child.

The really hard one to maintain is that all paths from the root to all children have to pass through the same number of black nodes. And this is where life gets really, really confusing because if we're going through a node we need to count it. Now, how do we go about recognizing this without going through all the paths. It's important to keep in mind that the purpose of the laws is only to recognize when a rotation is necessary. So, what we do is we'll say that we start off with one node, the root, and the root is always black; there's two nulls, one on the left than one on the right of the root node. So if we go through and find the left null, we've gone through one node, if we go through and find the right null that we've gone through one black node and that's no problem.

Insertions into a red black tree are going to be red nodes. So whenever we do an insertion of a new node. We'll color the new node red. That's not a law, that's just an implementation factor. So when we implement a new node we're going to insert it as a red note. So, if we had a value, for example of twenty and we inserted ten is the left sub child Then the left sub child will have two nulls and that changes the number of paths but it doesn't change the number of black nodes that we go through. We're still only going through the one black node at the root. So that's why this makes this makes some semblance of sense.

Now you're going to ask yourself what happens if we need to insert another node onto the left some child of ten. And the answer that is well it would be a red node of a red child and all that means that we're violating a law three, and that we need to recognize that a rotation is necessary. So, we've got these sort of solutions that we can use to not violate the laws. And we'll talk about that later in the next slide or in a later slide. But what we're going to say is that these are the laws of red black trees and we need to recognize and maintain them; how we do that is a much bigger problem.

In this Module, We Learned 1.27

We covered a lot of stuff in this module. We talked about the definitions for trees, we've talk about the heights and sizes and the depths and that sort of stuff. We talked about how we can store trees in main memory; remember, we talked about the parent-child relationship, parent-multi-child relationship and then we talk about the parent-child-sibling relationship. We talked about the design of binary search trees and what we need them for. We actually went through the binary search tree code. We talked about tree traversals: in-order, pre-order, post-order and level-order. We talked about when binary search trees fail and fall back on a linked list, and even if that doesn't happen as an actual linked list, even a portion of that could cause us to lose our big O of Log N. And then we talked about balanced binary search trees: the ATL and the red-black trees.

So we certainly covered a lot in this module and I want you to take away the idea that trees are very popular. We come across them quite often in computer science. In fact, if you just look at the files on your on your hard drive, what you're looking at is a tree. You're looking at a regular tree; it's not a binary search tree but it is a regular tree because you do have folders inside of folders in this higher structure. But they're not difficult and they're not complex. And they are incredibly useful, certainly in the binary search tree for being able to store a large quantity of data and being able to search it in big O of Log N time. So, I hope you took away a lot of good information from this module.