Functional Programming in Haskell

Prof. Madhavan Mukund and S. P. Suresh

Chennai Mathematical Institute

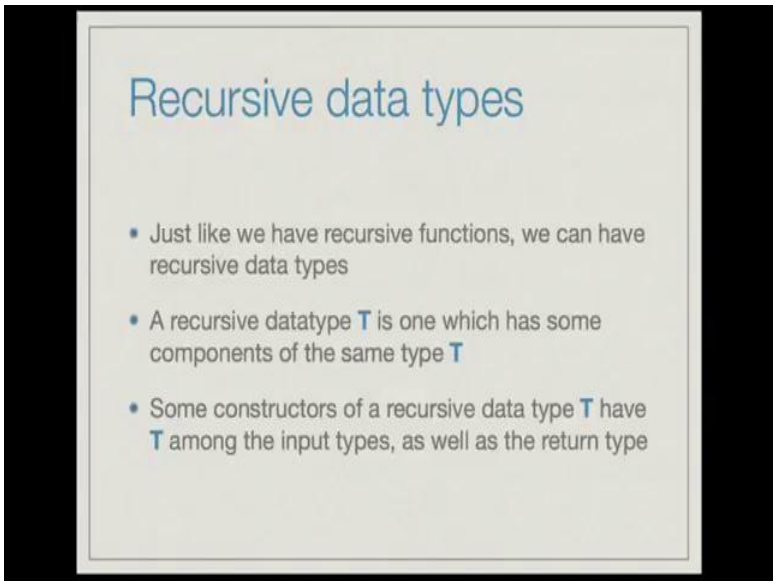
Module # 06

Lecture – 01

Recursive Data Types

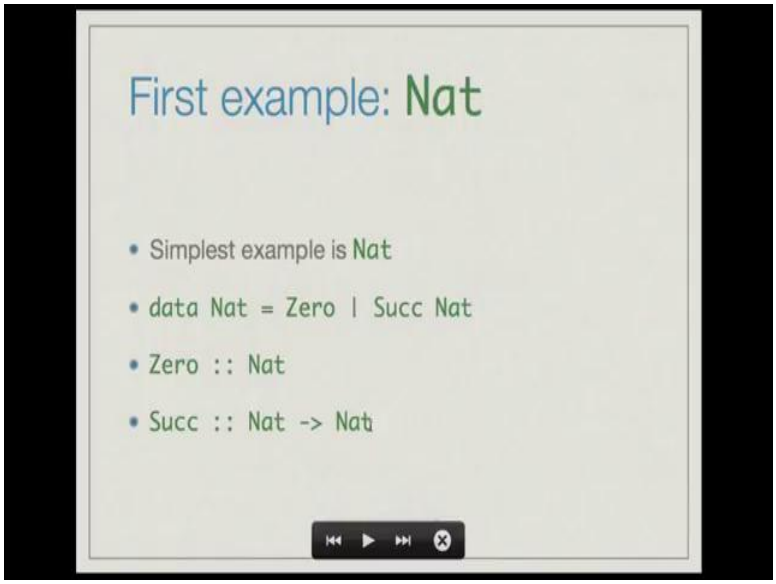
Welcome to week 6 of the NPTEL course on Functional Programming in Haskell. In this lecture, we shall be introducing Recursive Data Types.

(Refer Slide Time: 00:11)



Just like we have a recursive functions in Haskell, we can also have recursive data types. A recursive data type T is one which has some components that are of the same type T, this means that some constructors of a recursive data type T have T among the input types as well as the return type of course.

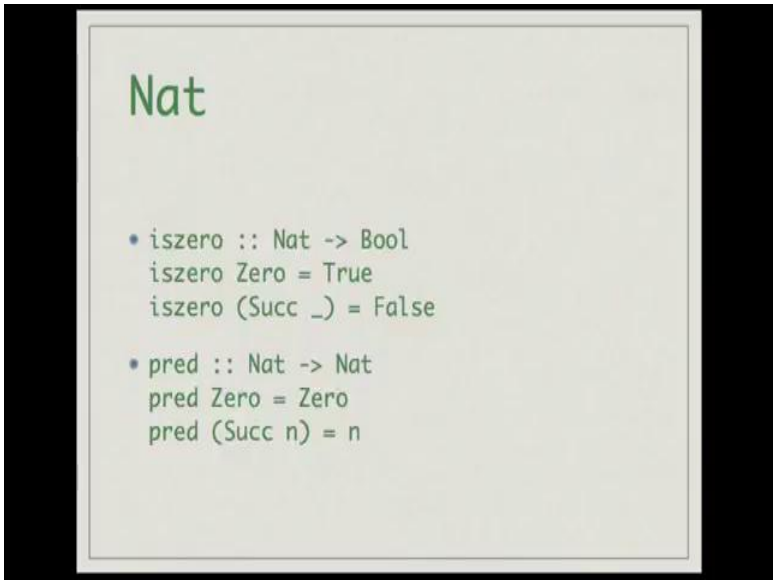
(Refer Slide Time: 00:35)



Here is a simple example of a recursive data type that of natural numbers. We know that Haskell offers a built-in type of int, integers, etcetera, but these represent both positive as well as negative numbers. In this example, we want to represent the non negative numbers that is 0, 1, 2, 3, etcetera; here is how we could do it. Data Nat equals zero OR successor Nat, this is the data declaration. We see that Nat is the type constructor or the name of the type and zero and successor are the value constructors.

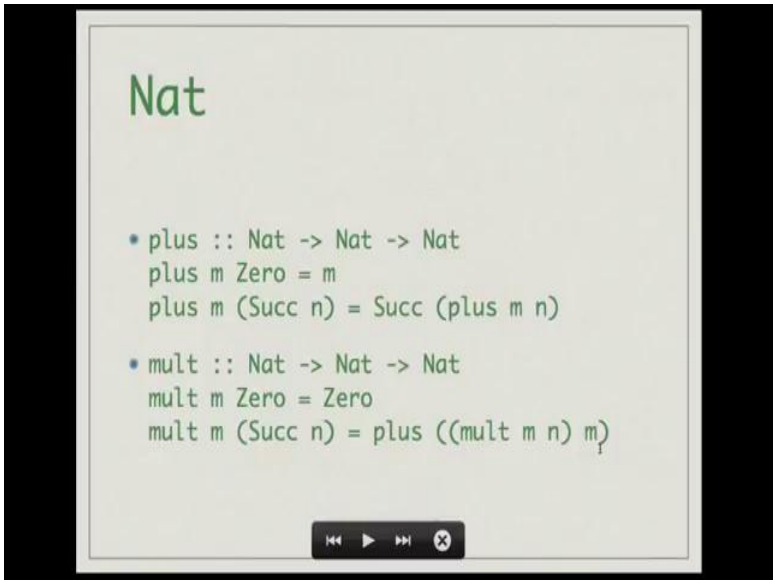
Zero is a value constructor that takes no argument at all whereas, successor is a value constructor that takes a natural number as a argument or an object of type Nat as a argument and returns an object of type Nat. As you can see, zero is a type Nat; successor is a function from Nat to Nat.

(Refer Slide Time: 01:47)



We could now define functions on the type Nat by using pattern matching as usual. For instance, here is a function that checks whether an object of type Nat is zero or not, iszero is the function from Nat to Bool, iszero of zero is true, iszero of successor of anything is false. Here is a function that computes the predecessor of a natural number, by convention the predecessor of zero is taken to be zero, since we do not allow non negative numbers. So, pred is a function from Nat to Nat, pred of zero equal zero, pred of successor of n equals n, we could have other similar definitions.

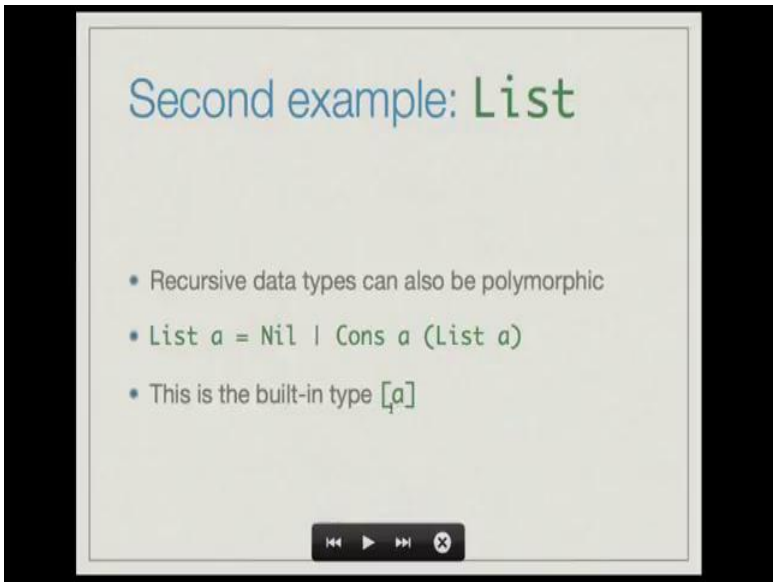
(Refer Slide Time: 02:37)



For instance, here is a definition of the plus function which takes two Nat as inputs and produces a Nat as an output. This function is defined as recursion plus m zero equals m, plus m successor of n equals successor plus m n. Recall that plus m n returns a Nat, as a result and successor precall is the constructor that takes a Nat and produces a Nat. So, successor plus m n produces an object of type Nat.

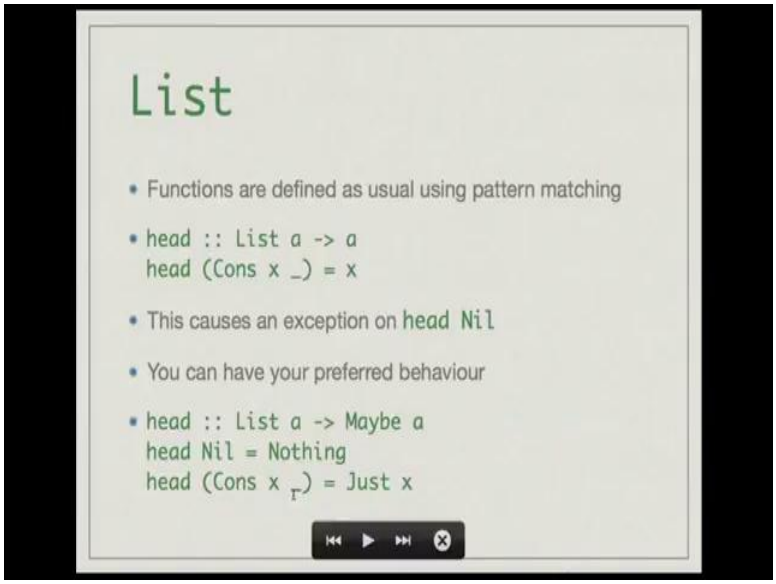
Here is the multiplication function defined recursively, mult m zero equal zero, mult m successor of n equals plus of mult m n and m. This is just saying that m times n plus 1 is the same as m times n plus m.

(Refer Slide Time: 03:31)



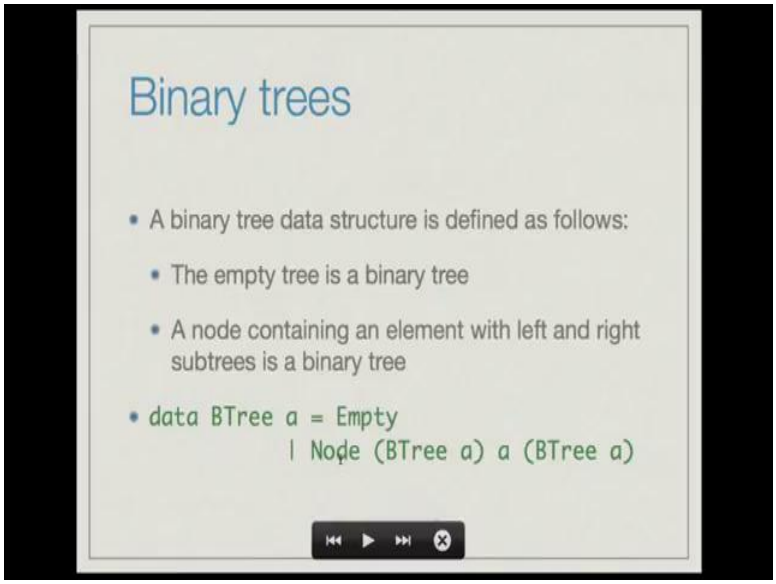
Here is a second example, another simple example of a recursive data type. This is the example of list, this example also shows that recursive data types can be polymorphic. Here is the data declaration, data list a equals Nil OR Cons a list a. Recall again, that Nil and Cons are the value constructors and list is the type constructers. This is just the built-in type list a, that is provided by Haskell, but we are providing this definition just as an example.

(Refer Slide Time: 04:12)



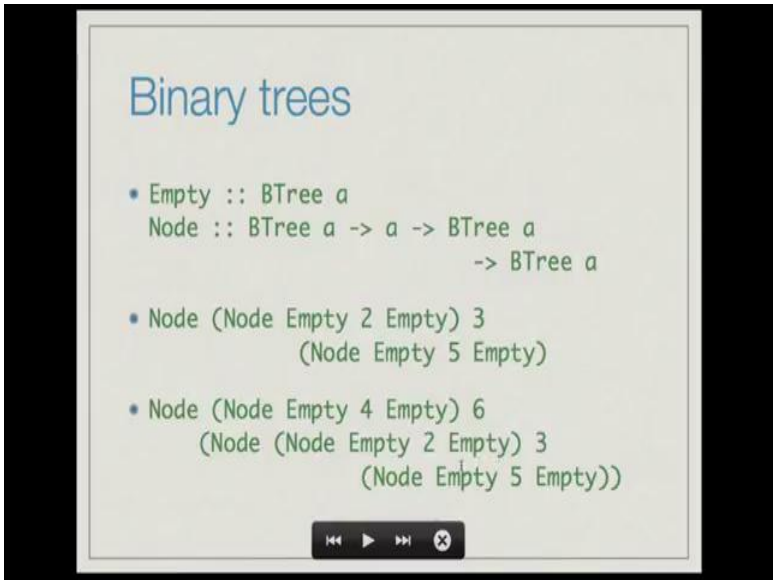
The functions are defined as usual using pattern matching, head is a function that takes a list of type a as input and produces an object of type a as output, head of Cons x anything is x. Notice that this function fails when you invoke head on the list Nil, this causes an exception on head Nil, you can now your own preferred behavior. For an instance, head is a function from list a to maybe a, head Nil equals nothing instead of not providing any definition for head Nil, you define head Nil to be nothing and you define head of Cons x anything to be just x.

(Refer Slide Time: 05:02)



Here, is a third and may be more challenging example that of binary trees. A binary tree data structure is defined as follows, the empty tree is a binary tree and a node containing an element with left and right sub tree is also a binary tree. So, the definition is as follows, data BTree a equals Empty OR Node BTree a, a an another argument which is a BTree a. So, again recall that Empty is a function which does not take any inputs and whose output is an object of type BTree a. Node is a function with type signature BTree a arrow a arrow b tree a arrow BTree a, which is the tree that is returned.

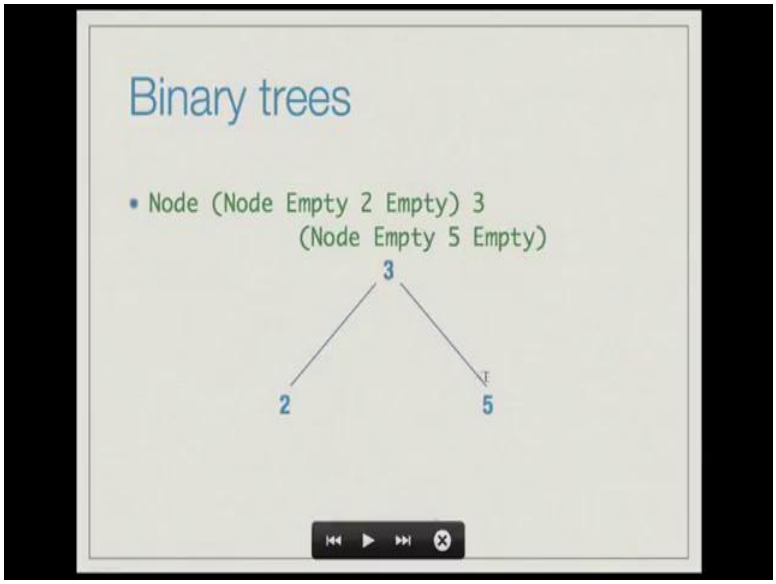
(Refer Slide Time: 05:54)



Now, that is explained here, Empty is a function with type BTree a, Node is a function of type BTree a arrow a arrow BTree a arrow BTree a. Here is an example of a binary tree, node of a left sub tree and the node, the value of the node is 3 and there is a right sub tree. The left sub tree is recursively given by Node and a left sub tree of it and here is the value at the root of the left sub tree, which is 2 and here is the right sub tree of the left sub tree.

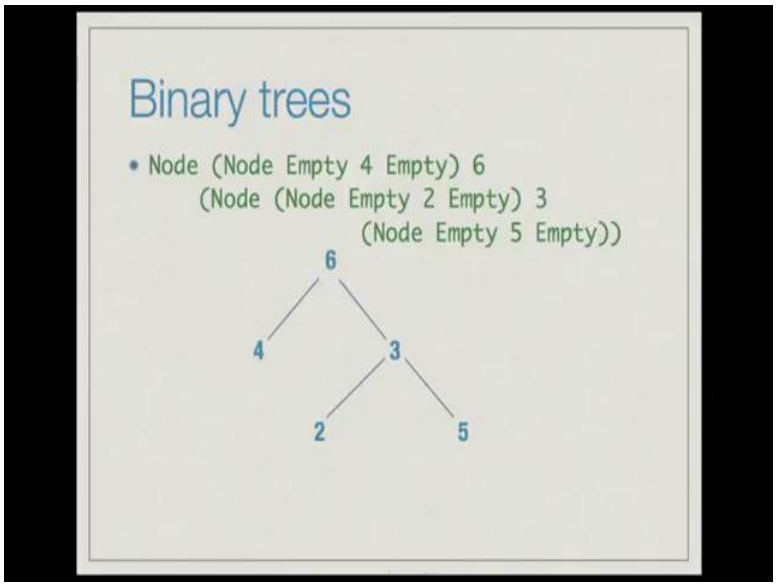
In this case, the left sub tree here and the right sub tree here are empty; similarly the right sub tree of the bigger tree with the root 3 is node empty 5 empty. Here is another example, node of some left sub tree which is given by node empty 4 empty and the root is 6 and the right sub tree itself is a non trivial tree. Node of some left sub tree and the root value being 3 and it is right sub tree being node empty 5 empty.

(Refer Slide Time: 07:09)



This tree would be defined as this, node of node empty 2 empty which defines this left sub tree, 3 which defines the root; node empty 5 empty, which defines this right sub tree which consists only of a single node.

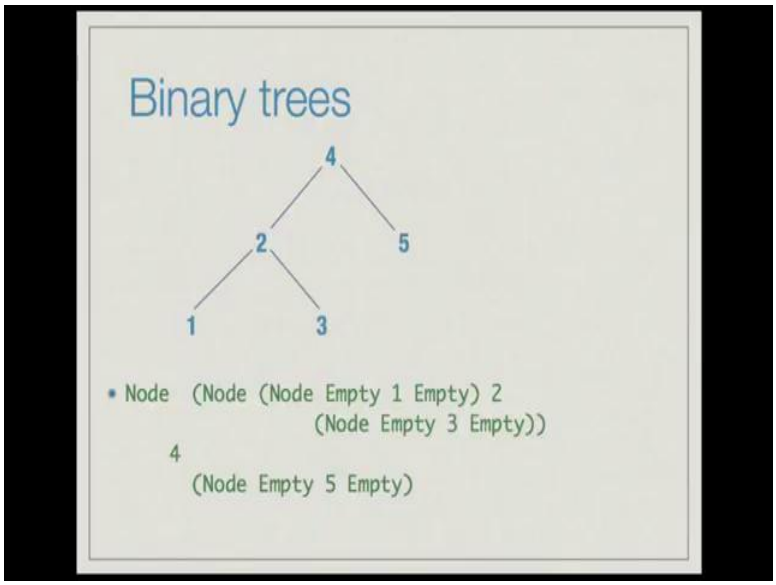
(Refer Slide Time: 07:33)



Here is an another example, you have a binary tree with 6 as root, 4 as the only node in the left sub tree and the right sub tree consisting of 3 as the root, 2 as the only node it is left sub tree and 5 as the only node in it is right sub tree. This would be defined as

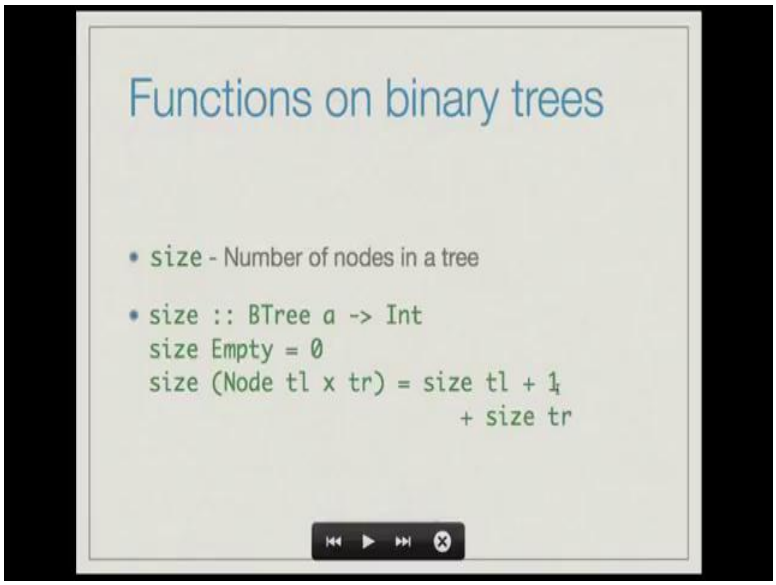
follows, node of node empty 4 empty, 6 node of node empty 2 empty, 3 node empty 5 empty, the structure should be fairly simple now.

(Refer Slide Time: 08:07)



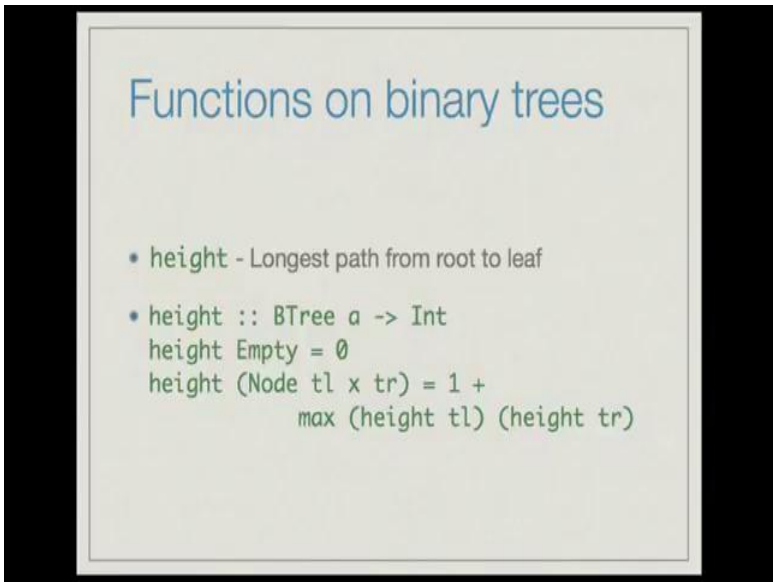
Here is an another example, this tree is node of a fairly large left sub tree, a non trivial left sub tree and the root value being 4, which is declared here and a right sub tree which has only a single node. The right sub tree is node empty 5 empty, the left sub tree itself is a tree which is of the form node 2 and a left sub tree which says node empty 1 empty and a right sub tree which says node empty 3 empty. This is how you represent binary trees.

(Refer Slide Time: 08:48)



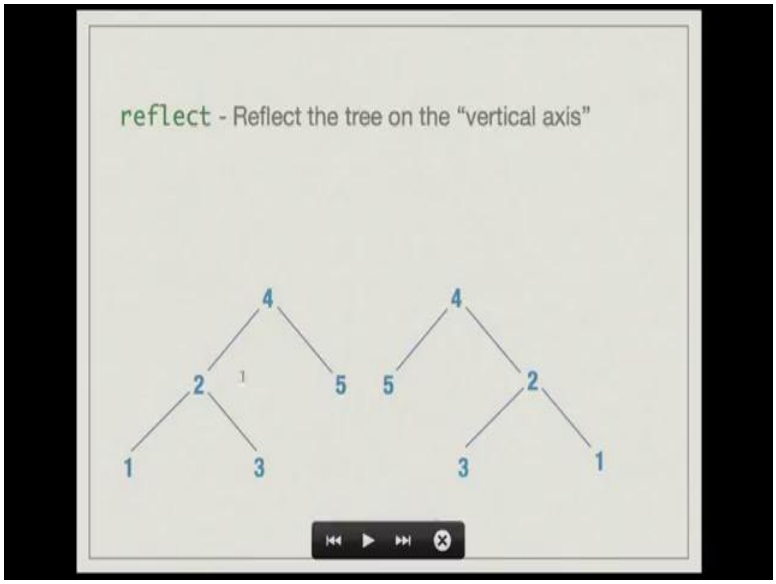
Now, you can define functions on binary trees as usual by using pattern matching. Here is the simple function on binary trees, the function size which returns the number of nodes in a tree. Size is a function with signature BTree a arrow Int, size of Empty equals 0, size of Node t l x t r, t l represents the left sub tree, t r represents the right sub tree, x denotes the value at the node equals just the size of the left sub tree plus 1, because the root is also a node plus the size of the right sub tree.

(Refer Slide Time: 09:36)



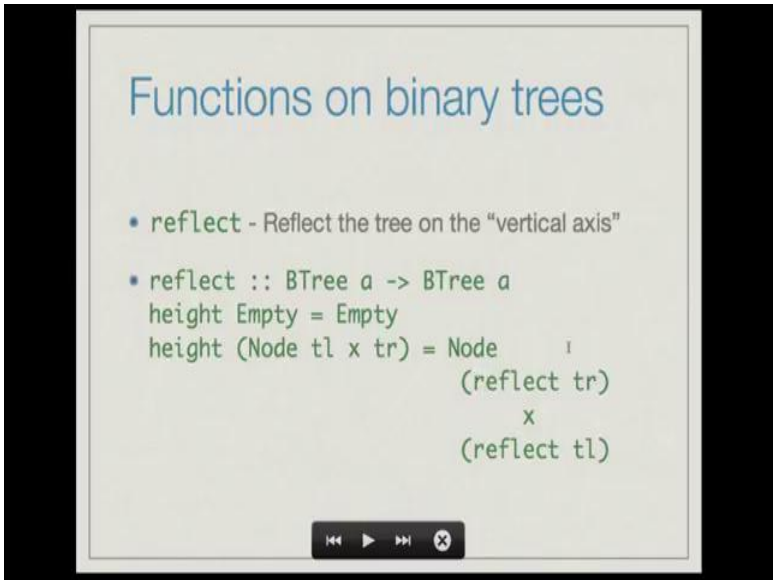
Here is another simple function on binary trees, height which gives the longest path from root to leaf. Height is a function with signature Btree a arrow Int, height of Empty equals 0, because there are no nodes at all, the root is the same as the leaf. Height of Node t l x t r equals 1 plus max of height t l and height t r. If the left sub tree had a height 4 and the right sub tree had height 3, then the tree itself would have height 5, because 5 is 1 plus max of 4, 3.

(Refer Slide Time: 10:20)



Here is a function which reflects the tree on the vertical axis, for instance you start out with 4, left sub tree having 2, 1, 3; right sub tree having just the node 5. If you reflect it, the left sub tree will no have a single node 5 and the right sub tree will have the reflection of the left sub tree here. So, you will get 2, 3, 1 instead of 2, 1, 3.

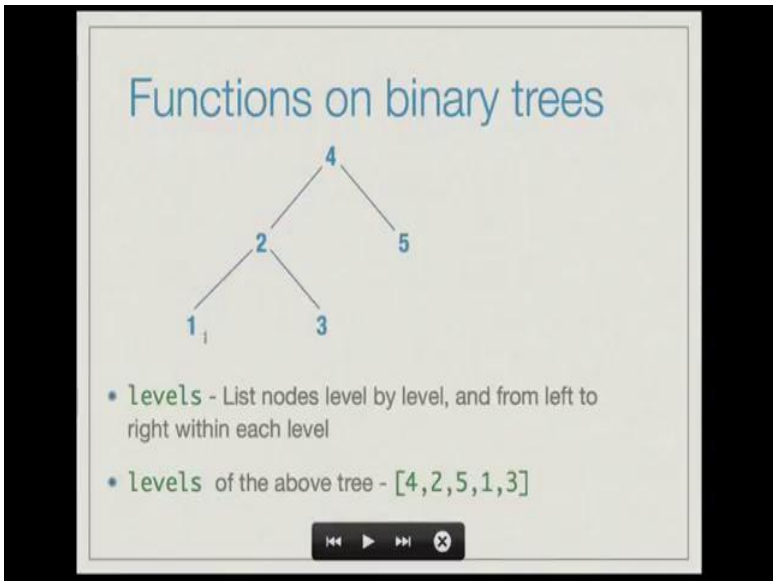
(Refer Slide Time: 10:51)



How would you define this? Reflect a function of signature BRree a arrow BTree a, reflect of Empty equals Empty, reflect of Node t l x t r is Node reflect t r x reflect t l.

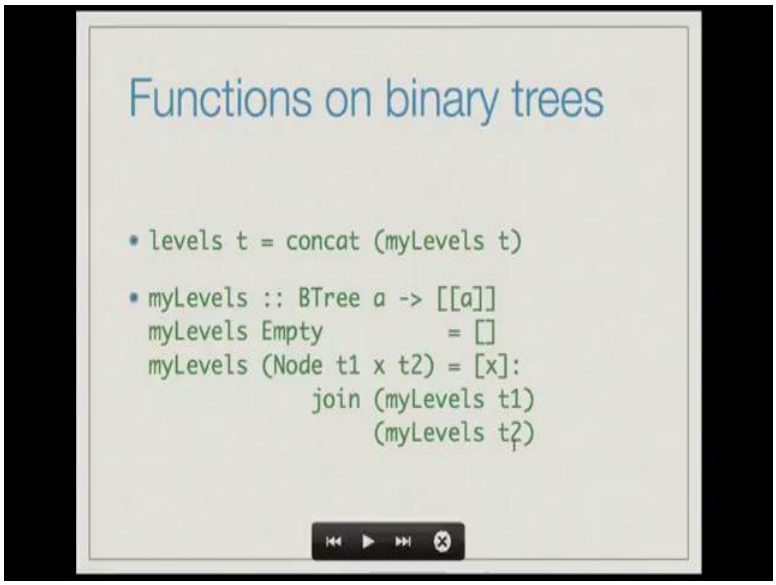
Notice that you form the node with the reflection of the right sub tree on the left and the reflection of the left sub tree on the right.

(Refer Slide Time: 11:18)



Here is another function, we want to list all the nodes in a tree level by level and from left to right within each level. So, in this tree there are three levels, this is one level, this is another level and this is the third level and you want to list elements from the first level and then, from the second level and then from the third level and within each level from left to right. So, levels on the above tree would yield 4, 2, 5, 1, 3.

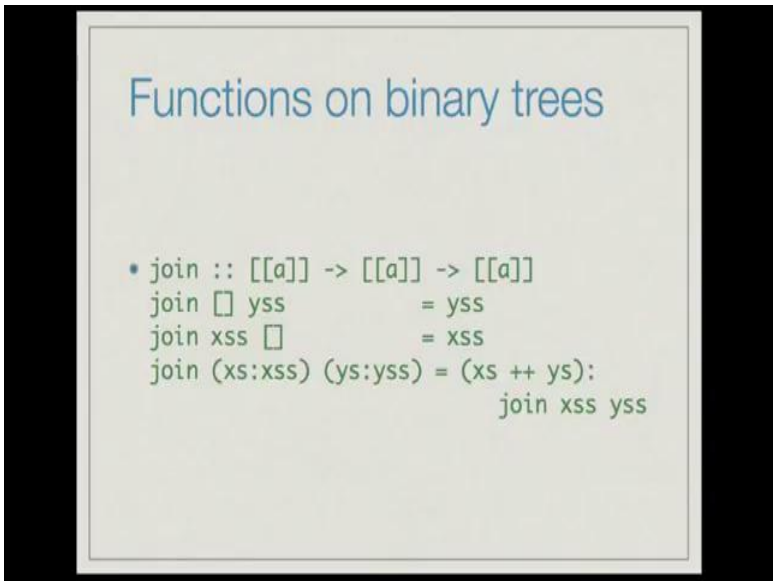
(Refer Slide Time: 11:56)



So, levels is a function which has signature BTree a arrow list a and it is defined as follows, levels t equals concat of myLevels t. Recall that the concat function takes a list of lists of type a and produces a list of type a, so myLevels t have to be a function that returns a list of list of type a, myLevels is a function with signature BTree a arrow list of list of a. This list is suppose to represent, the first list in this list represents all the nodes in the first level, the second list inside this list represents all the nodes in the second level, etcetera.

So, myLevels of Empty is nothing but, the empty list because there are no levels in an empty tree, myLevels of Node t 1 x t 2 is the list that you get by depending the list containing x, because x is the only node in the top level. So, the list consisting of x should be the first list in this list of lists and following that, you have bunch of lists that you get from myLevels t 1 and you have a bunch of lists that you get from myLevels t 2, you join the lists at the appropriate levels, so join. So, the definition of myLevels node t 1 x t 2 is the list consisting of x coming in front of join of myLevels t 1 myLevels t 2.

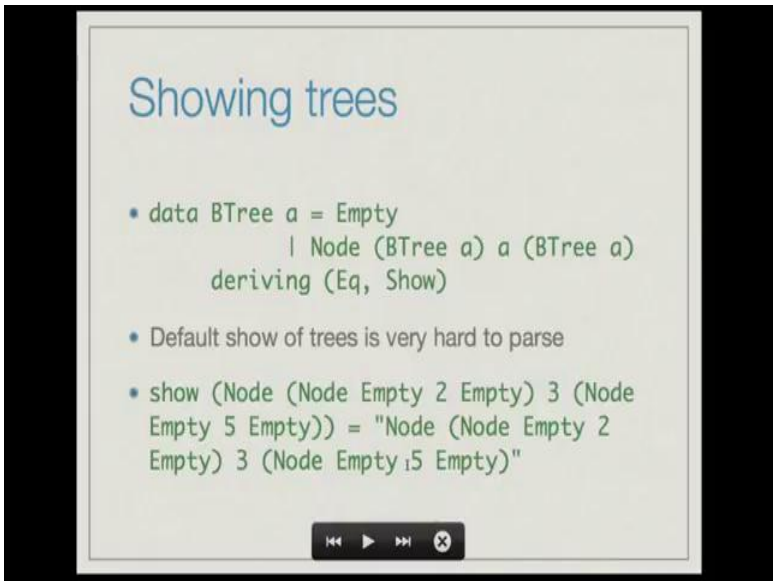
(Refer Slide Time: 13:56)



The code for join is given here, join is a function which has signature list of list of a arrow list of list of a arrow list of list of a. Now, the function join is very similar to zip with of plus plus applied to xss and yss, but there is a slide difference in this case. So, let us look at the definition in a bit more detail, join of empty list yss equals yss; in the case of zip with this would have been empty list; join of xss empty list equals xss.

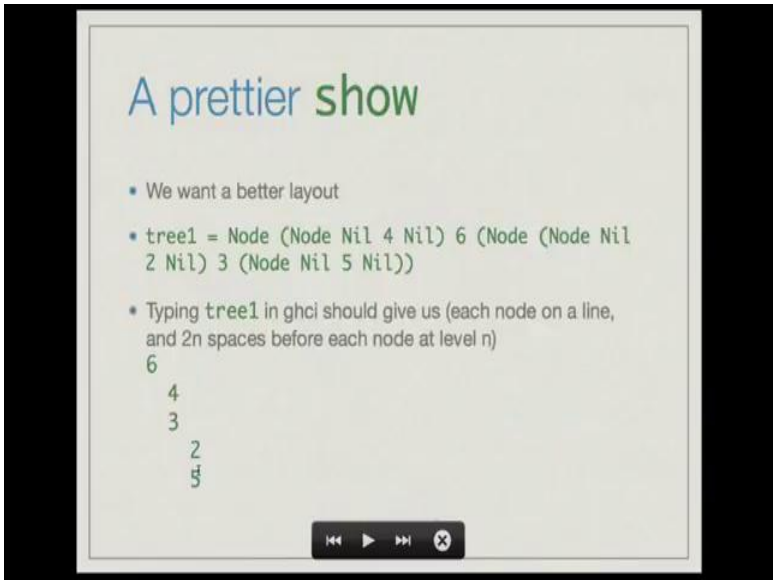
These two cases represent the situations, where the left sub tree has fewer levels than the right sub tree and this represents the case where the left sub tree has more levels than the right sub tree. Now, join of xs followed by xss and ys followed by yss is just xs plus plus ys coming in front of join of xss and yss, so this completes the definition of levels.

(Refer Slide Time: 15:12)



Let us look at how we can display trees, so a simple way of displaying tree is to say deriving equality and show, is to say deriving show in the data type declaration. But, the default show method on trees is very hard to parse, for instance show of this complicated tree is just the same representation given inside codes. As you can see, this is not very easy to read.

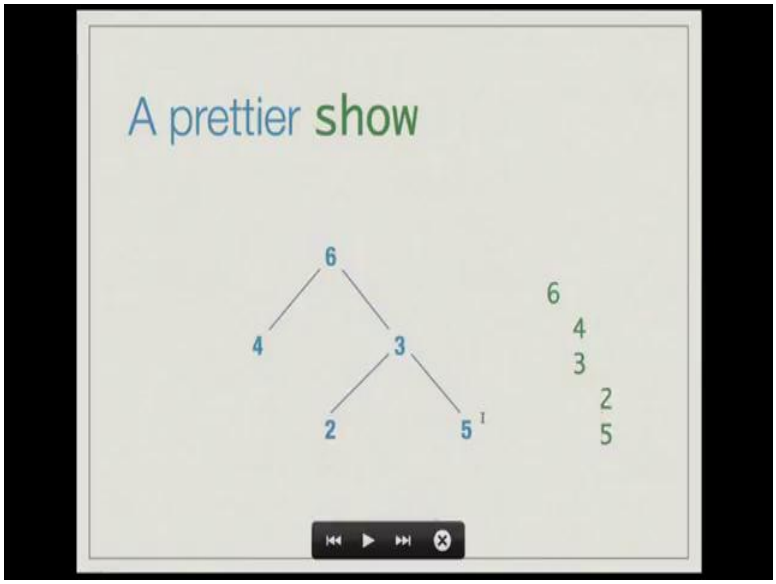
(Refer Slide Time: 15:52)



We may want a prettier show with a better layout, for instance suppose we define tree1 to be this tree, node with a root 6 and left sub tree consisting of a single node 4 and right sub tree consisting of a root 3 with left child 2 and right child 5. Then, typing tree1 in ghci should give us this, this is the decided behavior that we want. In this, we see that each node is printed on a line, is printed on a line and there are 2 n spaces before each node at level n.

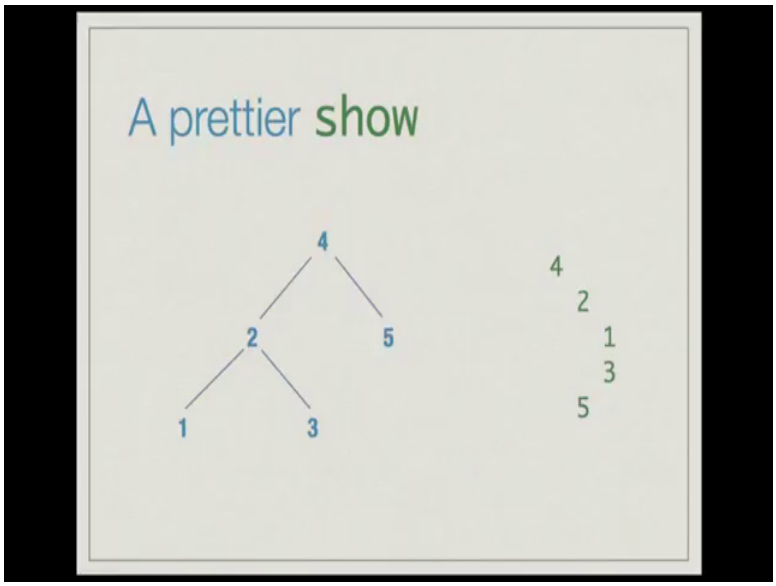
Here are assuming that the root is in level 0. So, 4 and 3 which are nodes at level 2 have two spaces before them and 2 and 5 which are nodes at level 3 have four spaces before them. And this node makes the structural of the tree clear, 6 is the root with two children 4 and 3, 4 does not have any children, whereas 3 has children 2 and 5.

(Refer Slide Time: 17:07)



You can see that the printed layout mirrors the structure of the tree and makes it pretty clear, 6 is the root, 4 and 3 are the children, 4 does not have any children, 3 has two children 2 and 5.

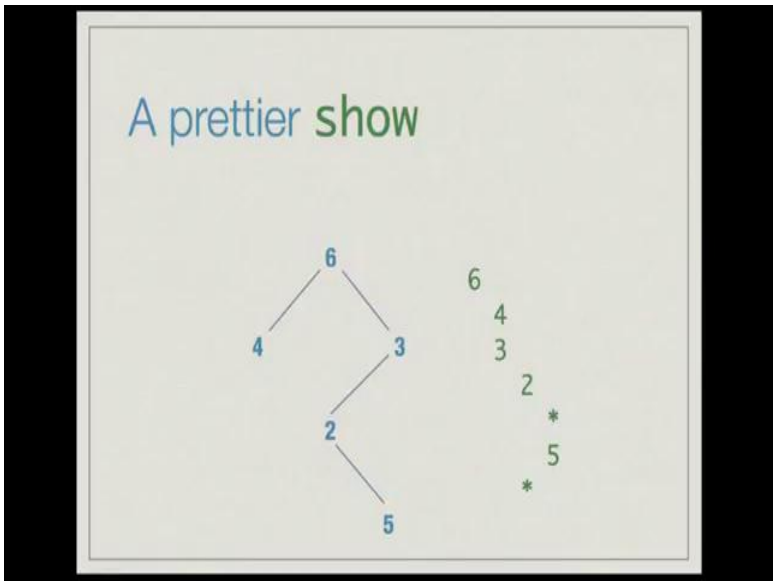
(Refer Slide Time: 17:27)



Here is another tree, 4 with left sub tree consisting of 2, 1 and 3 and the right sub tree being a single node 5. We want this to be layout in this fashion, 4 with two children 2 and 5 and the children of 2 immediately displayed on the lines following 2, 1 and 3 and 5 would displayed after that. But, you can look at which column 5 appears in and

determine whose child it is, 5 is the child of 4 because 5 appears in column 3, whereas 4 appears in column 4.

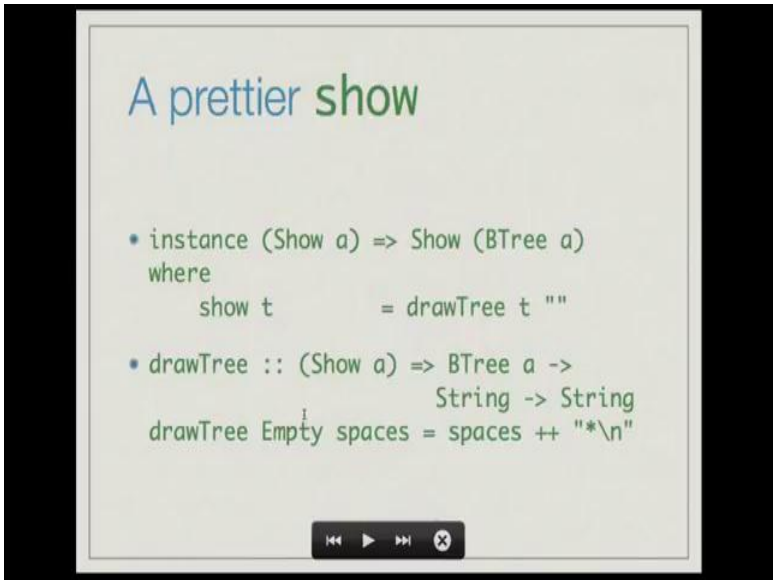
(Refer Slide Time: 18:07)



Here is another tree, it is a tree with root 6, left sub tree consisting of a single node 4, right sub having a node 3 whose left child is 2 and whose right child is 5. This is the layout as follows, 6 followed by it is two children 4 and 3, followed by it is left child which is 2 whose children are displayed here as star and 5, this is because this node has only one child and the other child is empty and we use star to denote which child is empty, in case there is only one child.

In case both child's are empty as in the example of this node 4, we do not display anything. But, if one sub tree is empty and the other sub tree is non empty, then we display a star corresponding to the sub tree that is empty. So, 2 has left sub tree which is the empty and right sub which consists of a single node 5 and 3 itself has this is the left sub tree, 2 star 5 and it is right sub tree is empty which is denoted by star. Now, how do we define this show function?

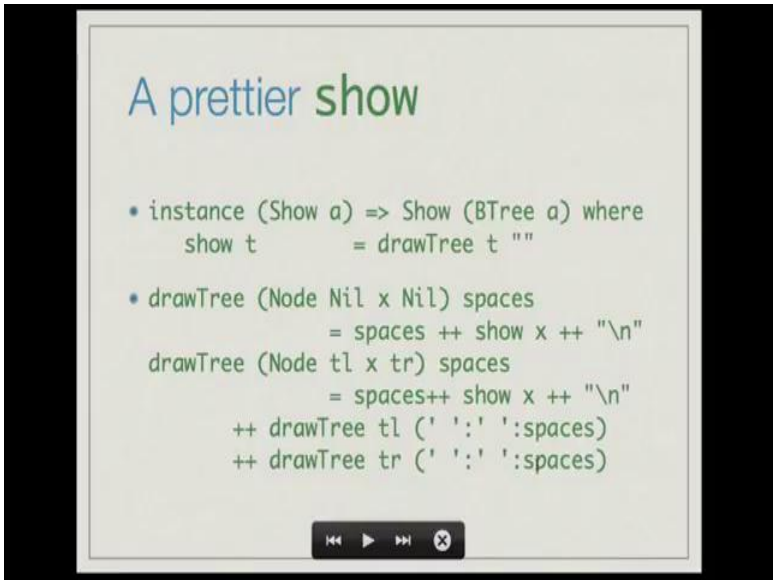
(Refer Slide Time: 19:23)



We defined it as follows using the instant declaration, instance Show a implies Show BTree of a, where Show t, t is the tree here, it just drawTree t empty string. DrawTree will draw the layout of the tree, where the second parameter which is the string tells how may spaces to insert before drawing the current sub tree. So, drawTree is a function with signature Show a implies BTree a arrow string arrow string, this string is the number of spaces that have to be return displayed before the current sub tree is displayed and this string is actually the output, the layout of the tree.

DrawTree empty, this is the empty tree spaces equals spaces plus plus star back slash and which denotes the new line character. You see that you will encounter this case only when this is the sole empty sub tree of a node.

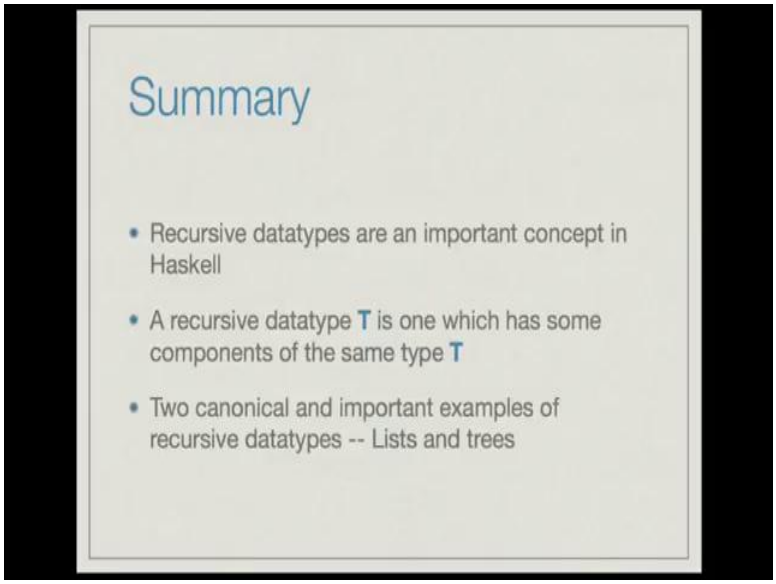
(Refer Slide Time: 20:47)



The other cases when you have a node x both of whose sub trees are empty, Node Nil x Nil, drawTree of Node Nil x Nil spaces is nothing but, spaces plus plus show x plus plus new line. DrawTree of Node t l x t r, where t l is the left sub tree and t r is the right sub tree spaces is nothing but, spaces plus plus show x plus plus new line. Remember that we always add this much, this many spaces before displaying the current sub tree, plus you append that with drawTree of t l.

But, when you draw the left sub tree you have to add two more spaces, so which you do by adding two spaces from the beginning of this list. And then, you concat that with drawTree of t r which is the right sub tree, again giving two spaces adding two more spaces in the beginning of laying out the right sub tree.

(Refer Slide Time: 21:58)



Since summary, recursive data types have been introduced in this lecture, they are a very important concept in Haskell. A recursive data type T is one which has some of it is components of the same type T and we have seen two canonical and important examples of recursive data types, list and trees. In the next lecture, we will see more on trees.