

Introduction To Haskell Programming

Prof. S. P. Suresh

Chennai Mathematical Institute

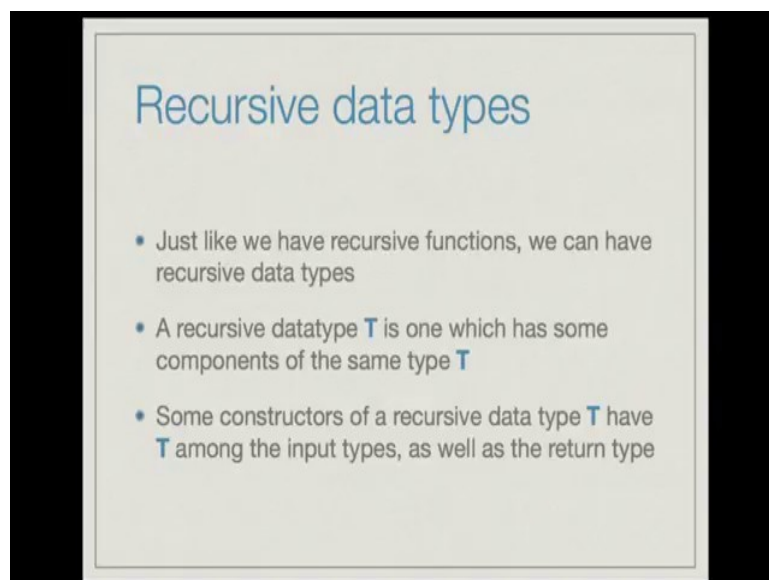
Module # 06

Lecture – 01

Recursive Data Types

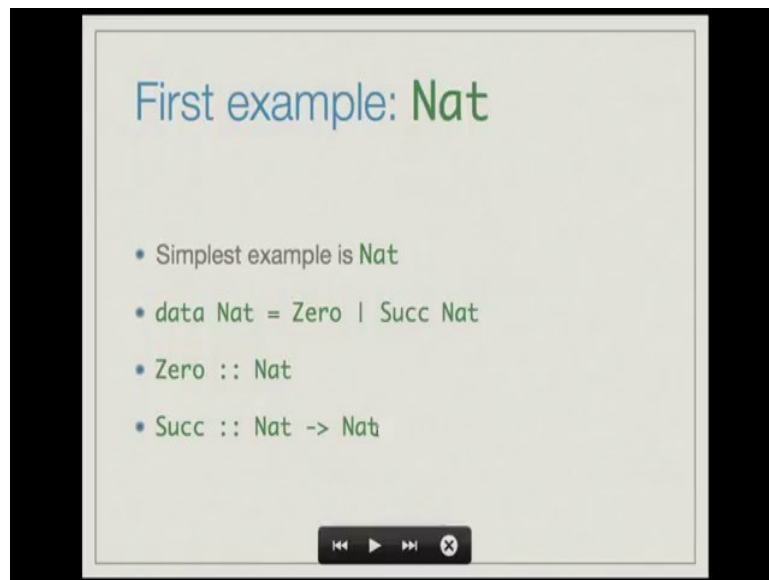
Welcome to week six of the NPTEL course on Functional Programming in Haskell. In this lecture, we shall be introducing Recursive Data Types.

(Refer Slide Time: 00:11)



Just like we have 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.

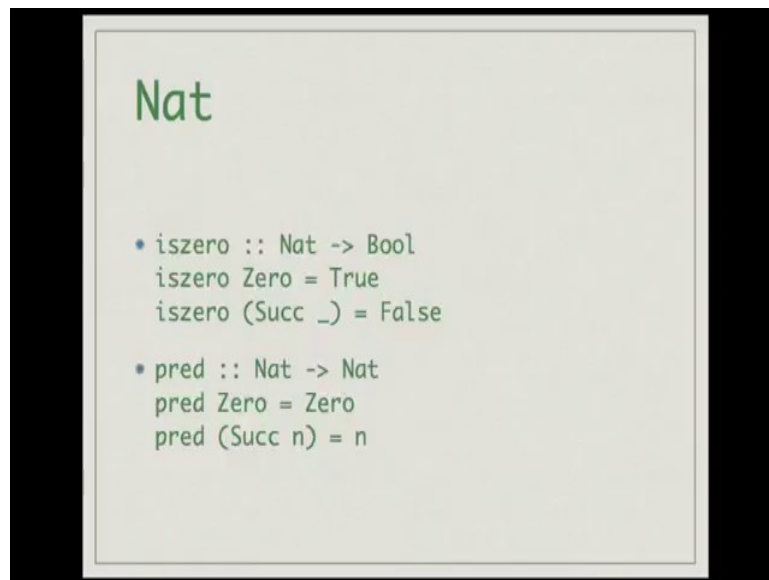
(Refer Slide Time: 00:35)



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` for integers 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 = Zero | Succ Nat`, this is the data declaration. We see that `Nat` is the type constructor or the name of the type and `Zero` and `Succ` are the value constructors.

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

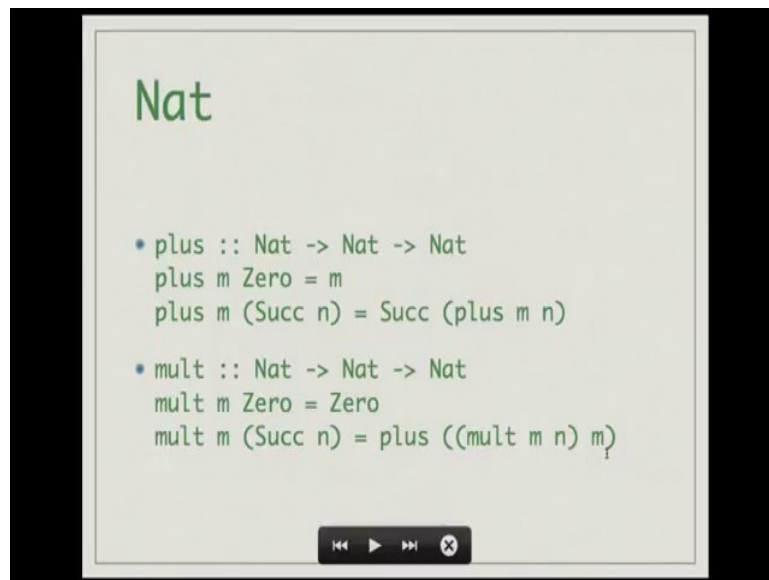
(Refer Slide Time: 01:47)



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 -> Bool`, `isZero Zero = True`, `isZero of Zero is True`. `isZero (Succ _) = False`, `isZero of Succ 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 -> Nat`, `pred Zero = Zero`, `pred (Succ n) = n`, we could have other similar definitions.

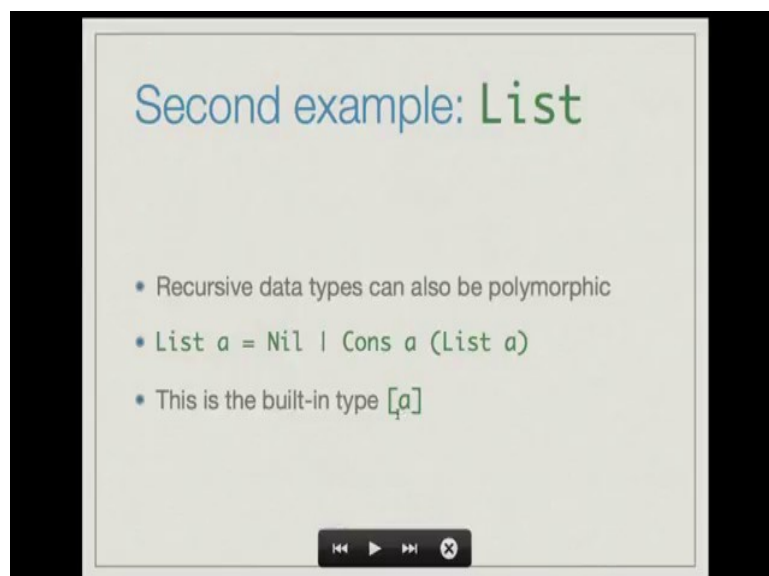
(Refer Slide Time: 02:37)



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 using recursion. `plus m Zero = m`, `plus m (Succ n) = Succ (plus m n)`. Recall that `plus m n` returns a Nat as a result and `Succ`, recall is the constructor that takes a Nat and produces a Nat. So, `Succ (plus m n)` produces an object of type Nat.

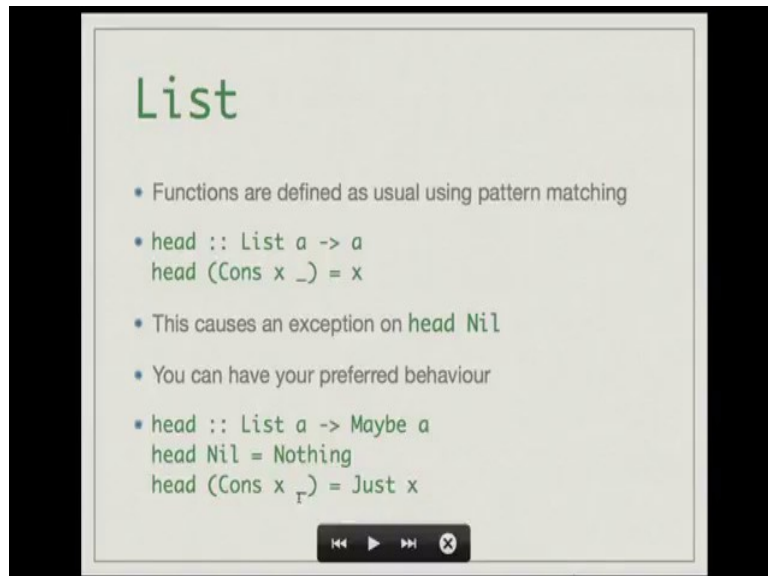
Here is the multiplication function defined recursively, `mult m Zero = Zero`, `mult m (Succ n) = plus ((mult m n) m)`. This is just saying that $m \cdot (n+1)$ is the same as $m \cdot n + m$.

(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, $\text{List } a = \text{Nil} \mid \text{Cons } a (\text{List } a)$. Recall again, that Nil and Cons are the value constructors and List is the type constructor. 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)

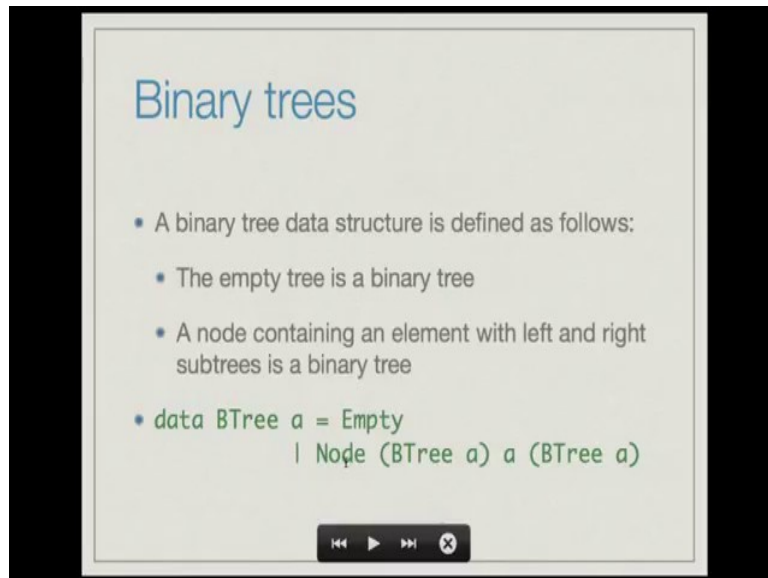


The slide is titled "List" in green. It contains a bulleted list of points and two code snippets. The first code snippet shows the standard implementation of head, and the second shows an implementation that returns a Maybe value to handle the Nil case gracefully.

- Functions are defined as usual using pattern matching
- `head :: List a -> a`
`head (Cons x _) = x`
- This causes an exception on `head Nil`
- You can have your preferred behaviour
- `head :: List a -> Maybe a`
`head Nil = Nothing`
`head (Cons x _) = Just x`

The functions are defined as usual using pattern matching, `head :: List a -> a`, head is a function that takes a list of type a as input and produces an object of type a as output, `head (Cons x _) = x`. Notice that this function fails when you invoke head on the list Nil, this causes an exception on head Nil, you can have your own preferred behavior. For instance, head is a function from list a to Maybe a written as `head :: List a -> Maybe a`, `head Nil = Nothing`. instead of not providing any definition for head Nil, you define head Nil to be nothing and you define `head (Cons x _)` to be Just x.

(Refer Slide Time: 05:02)

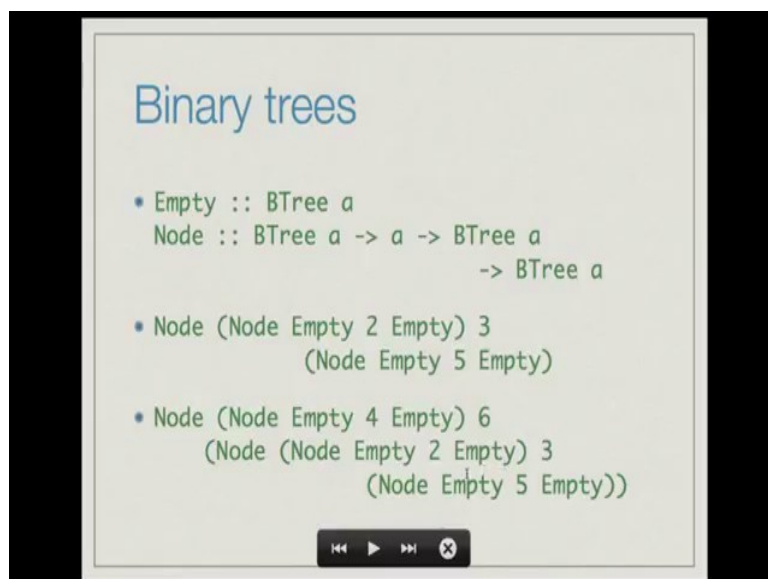


Binary trees

- A binary tree data structure is defined as follows:
 - The empty tree is a binary tree
 - A node containing an element with left and right subtrees is a binary tree
- ```
data BTree a = Empty
 | Node (BTree a) a (BTree a)
```

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 subtree is also a binary tree. So, the definition is as follows, `data BTree a = Empty | Node (BTree a) 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 -> a -> BTree a -> BTree a`, which is the tree that is returned.

(Refer Slide Time: 05:54)



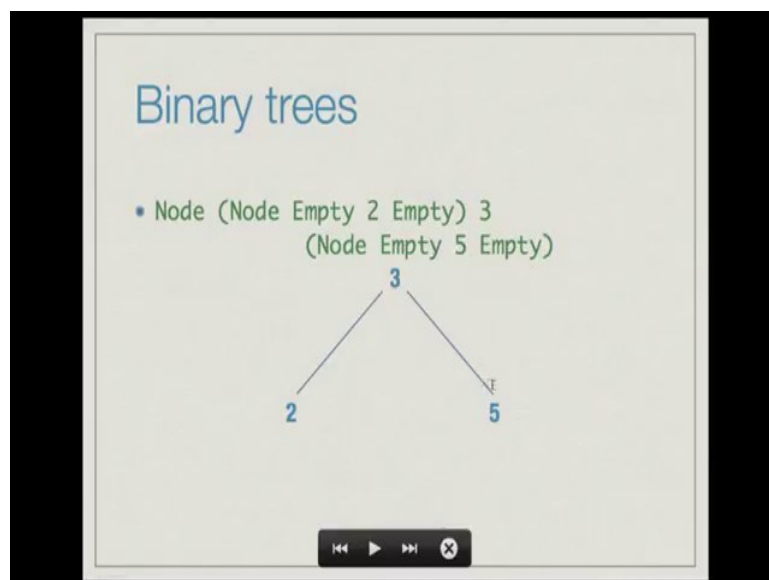
### Binary trees

- ```
Empty :: BTree a
Node  :: BTree a -> a -> BTree a
                                -> BTree a
```
- ```
Node (Node Empty 2 Empty) 3
 (Node Empty 5 Empty)
```
- ```
Node (Node Empty 4 Empty) 6
      (Node (Node Empty 2 Empty) 3
            (Node Empty 5 Empty))
```

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

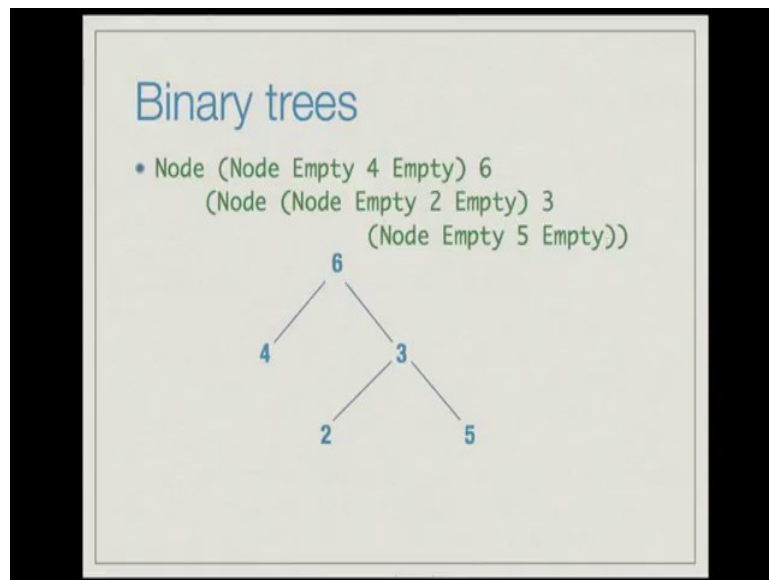
In this case, the left sub tree and the right sub tree are empty; similarly the right sub tree of the bigger tree with the root 3 is (Node Empty 5 Empty). Here is another example, Node of some left sub tree which is given by (Node Empty 4 Empty) and the root is 6 and the right sub tree itself is a non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

(Refer Slide Time: 07:09)



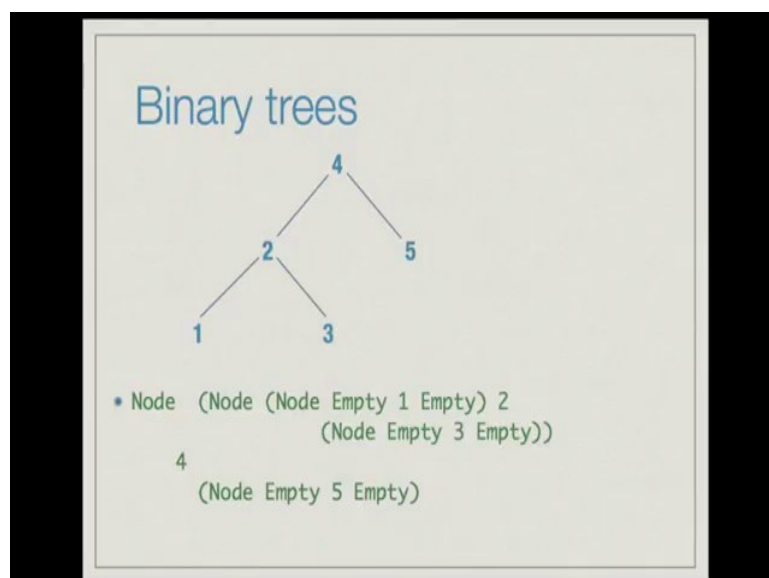
This tree would be defined as this, Node (Node Empty 2 Empty) which defines this left sub tree, 3 which defines the root; (Node Empty 5 Empty), which defines this right subtree which consists only of a single Node.

(Refer Slide Time: 07:33)



Here is another example, you have a binary tree with 6 as root, 4 as the only Node in the left subtree and the right subtree consisting of 3 as the root, 2 as the only Node in its left subtree and 5 as the only Node in its right subtree. This would be defined as follows, Node (Node Empty 4 Empty) 6 (Node (Node Empty 2 Empty) 3 (Node Empty 5 Empty)), the structure should be fairly simple now.

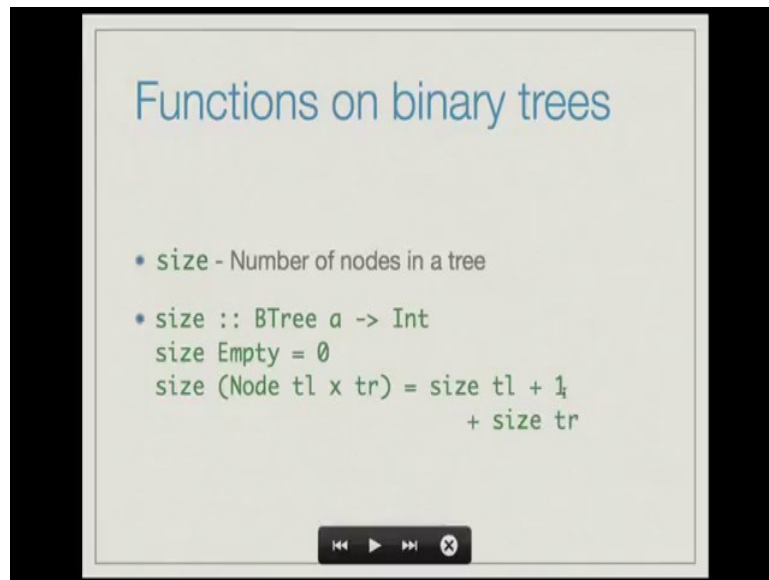
(Refer Slide Time: 08:07)



Here is another example, this tree is Node of a fairly large left sub tree, a non trivial left subtree and the root value being 4, which is declared here and a right subtree which has only

a single Node. The right subtree 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 subtree 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 -> Int, size of Empty equals 0, size of (Node tl x tr) where tl represents the left subtree, tr represents the right subtree, x denotes the value at the Node, equals size tl + 1 +size tr, the size of the left subtree plus 1, because the root is also a Node plus the size of the right subtree.

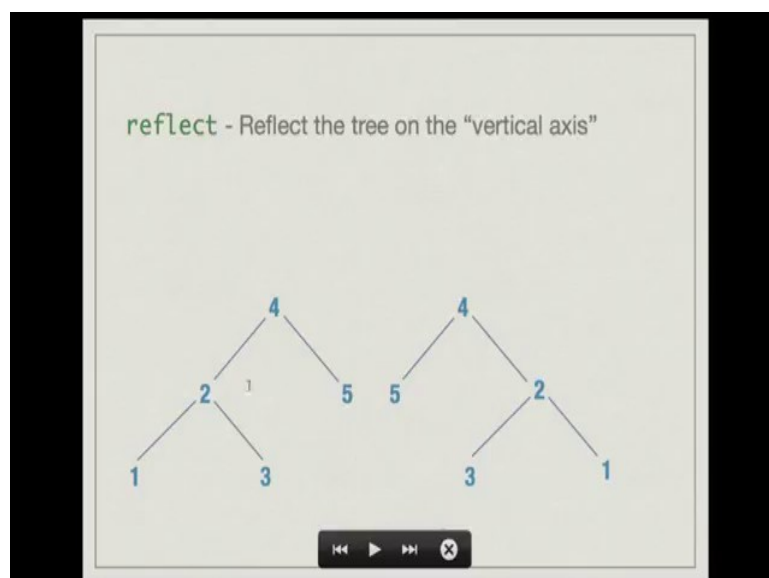
(Refer Slide Time: 09:36)

Functions on binary trees

- **height** - Longest path from root to leaf
- **height** :: BTree a -> Int
height Empty = 0
height (Node tl x tr) = 1 +
 max (height tl) (height tr)

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 -> Int, height of Empty equals 0, because there are no Nodes at all, the root is the same as the leaf. Height of (Node tl x tr) equals 1 plus max of height tl and height tr. If the left sub tree had height 4 and the right sub tree had height 3, then the tree itself would have height 5, because 5 is 1 + max (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 subtree having just the Node 5. If you reflect it, the left sub

tree will now 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)

Functions on binary trees

- **reflect** - Reflect the tree on the "vertical axis"
- **reflect** :: BTree a -> BTree a

```
height Empty = Empty
height (Node tl x tr) = Node (reflect tr) x (reflect tl)
```

Navigation controls: back, forward, search, close

How would you define this? Reflect a function of signature BTree a -> BTree a, reflect of Empty equals Empty, reflect of (Node tl x tr) is Node (reflect tr) x (reflect tl). 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)

Functions on binary trees

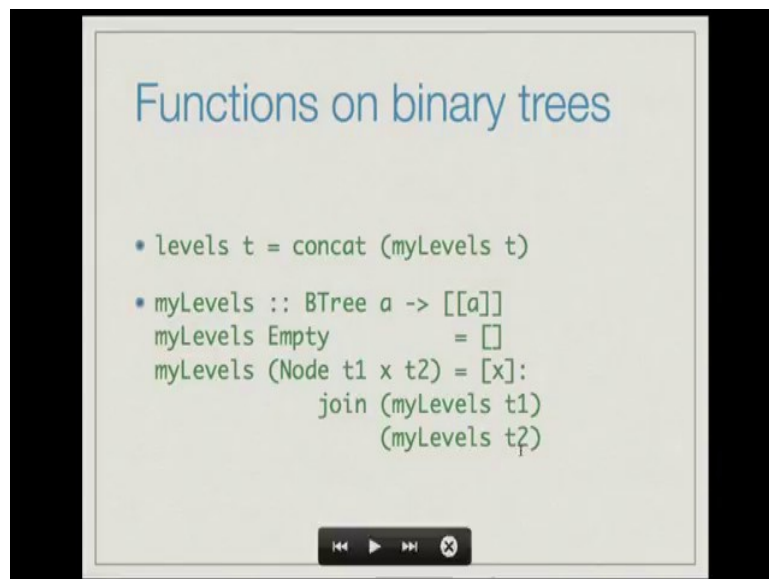
```
graph TD
    4 --- 2
    4 --- 5
    2 --- 1
    2 --- 3
```

- **levels** - List nodes level by level, and from left to right within each level
- **levels** of the above tree - [4,2,5,1,3]

Navigation controls: back, forward, search, close

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].

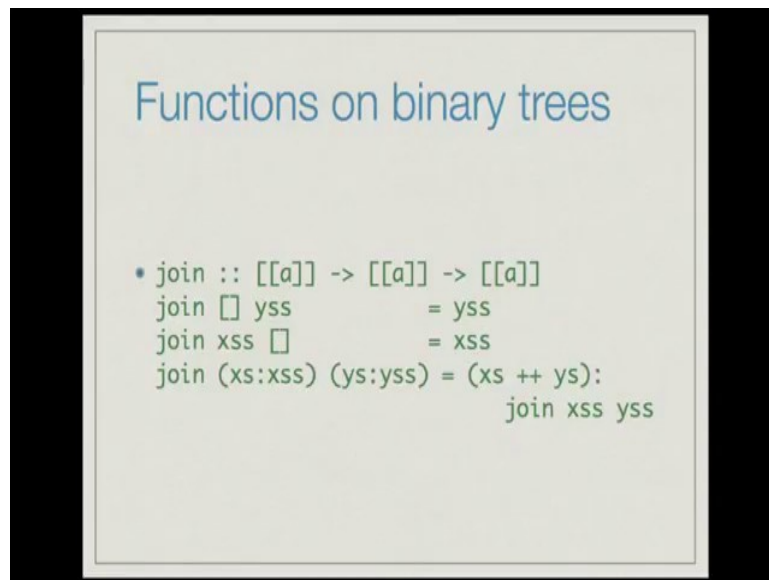
(Refer Slide Time: 11:56)



So, levels is a function which has signature `BTree a -> list a` and it is defined as follows, `levels t = concat (myLevels t)`. Recall that the `concat` function takes a list of lists of type `a`, `[[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 -> [[a]]`. This list is supposed 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 t1 x t2)` is the list that you get by prepending 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 a bunch of lists that you get from `myLevels t1` and you have a bunch of lists that you get from `myLevels t2`, you join the lists at the appropriate levels. So, the definition of `myLevels (Node t1 x t2)` is the list consisting of `x` coming in front of `join (myLevels t1) (myLevels t2)`.

(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, $[[a]] \rightarrow [[a]] \rightarrow [[a]]$. Now, the function join is very similar to zip with of ++ applied to xss and yss, but there is a slight difference in this case. So, let us look at the definition in a bit more detail, join [] yss equals yss; in the case of zip with this would have been empty list; join xss [] 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:xss) (ys:yss) is just (xs ++ ys) coming in front of join xss yss. so this completes the definition of levels.

(Refer Slide Time: 15:12)

Showing trees

- ```
data BTree a = Empty
 | Node (BTree a) a (BTree a)
 deriving (Eq, Show)
```
- Default show of trees is very hard to parse
- ```
show (Node (Node Empty 2 Empty) 3 (Node
Empty 5 Empty)) = "Node (Node Empty 2
Empty) 3 (Node Empty 5 Empty)"
```

Let us look at how we can display trees, so a simple way of displaying tree is to say deriving Eq 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)

A prettier show

- We want a better layout
- ```
tree1 = Node (Node Nil 4 Nil) 6 (Node (Node Nil
2 Nil) 3 (Node Nil 5 Nil))
```
- Typing tree1 in ghci should give us (each node on a line, and 2n spaces before each node at level n)

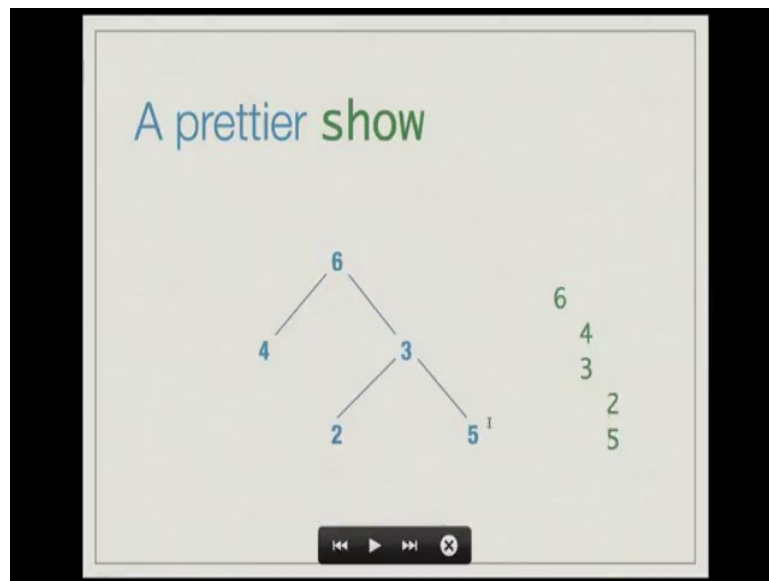
```
6
 4
 3
 2
 5
```

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 desired behavior that we want. In this, we see that each Node is printed on a line, is printed on a line and there are  $2n$  spaces before each Node at level  $n$ .

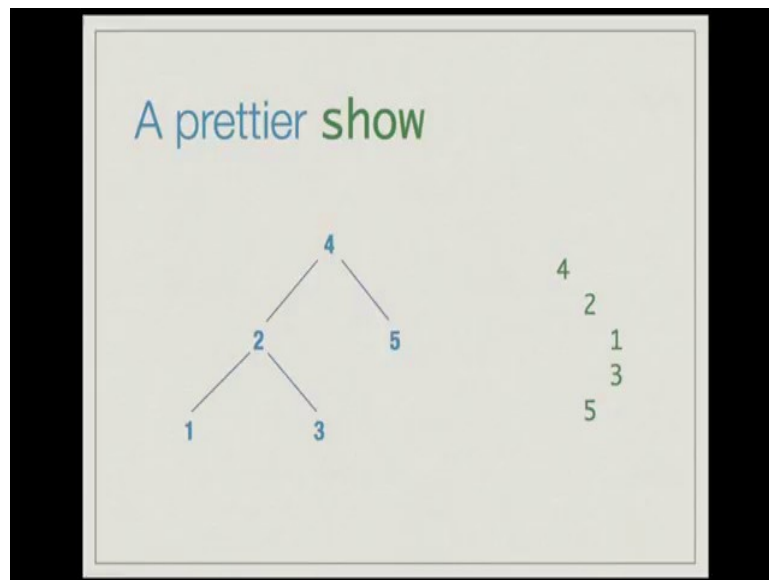
Here are assuming that the root is in level 0. So, 4 and 3 which are Nodes at level 2 have two spaces before them and 2 and 5 which are Nodes at level 3 have four spaces before them. And this layout makes the structure of the tree clear, 6 is the root with two children 4 and 3, 4 does not have any children,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

(Refer Slide Time: 17:07)



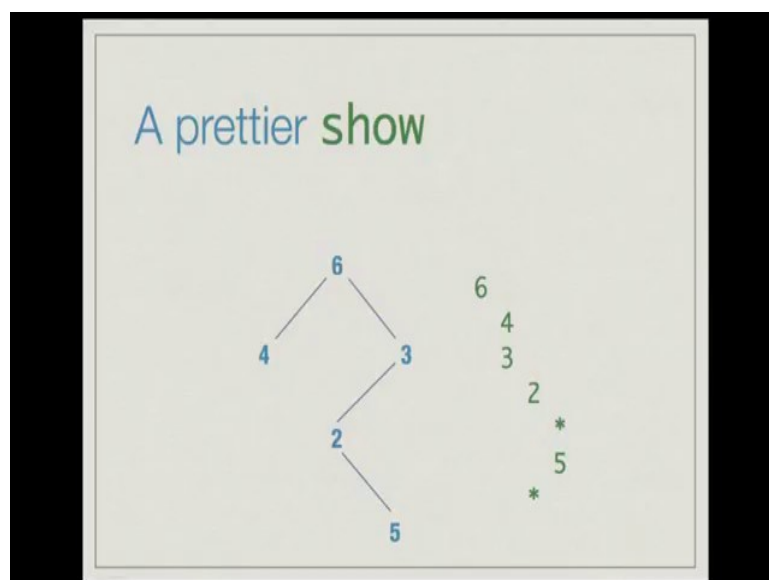
You can see that the printed layout mirrors the structure of the tree and makes it pretty clear, 6 is the root, 4 and 3 are the children, 4 does not have any children, 3 has two children 2 and 5.

(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 laid out 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, 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)



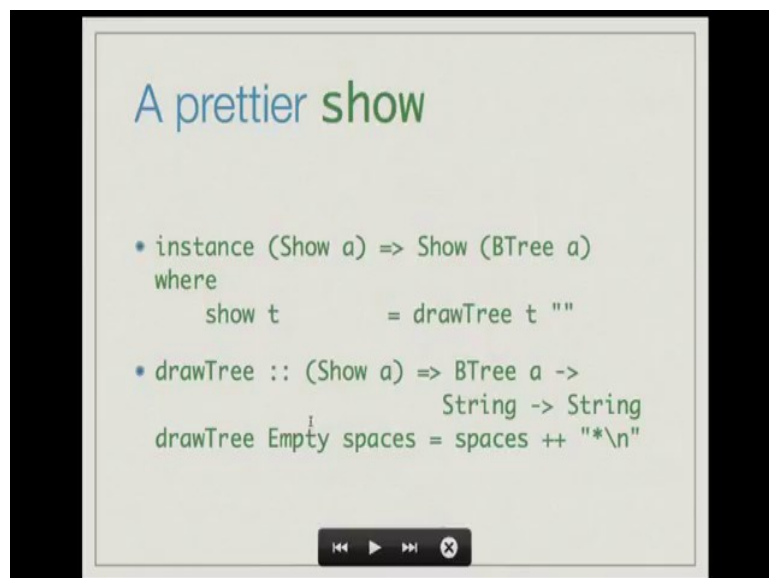
Here is another tree, it is a tree with root 6, left sub tree consisting of a single Node 4, right subtree having a Node 3 whose left child is 2 and whose right child is 5. This is laid out as



follows, 6 followed by its two children 4 and 3, followed by its left child which is 2 whose children are displayed here as \* 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 children are empty as in the example of this Node 4, we do not display anything. But, if one subtree is empty and the other subtree is non empty, then we display a star corresponding to the sub tree that is empty. So, 2 has left sub tree which is empty and right sub which consists of a single Node 5 and 3 itself has this as the left sub tree, 2 \* 5 and its right sub tree is empty which is denoted by \*. Now, how do we define this show function?

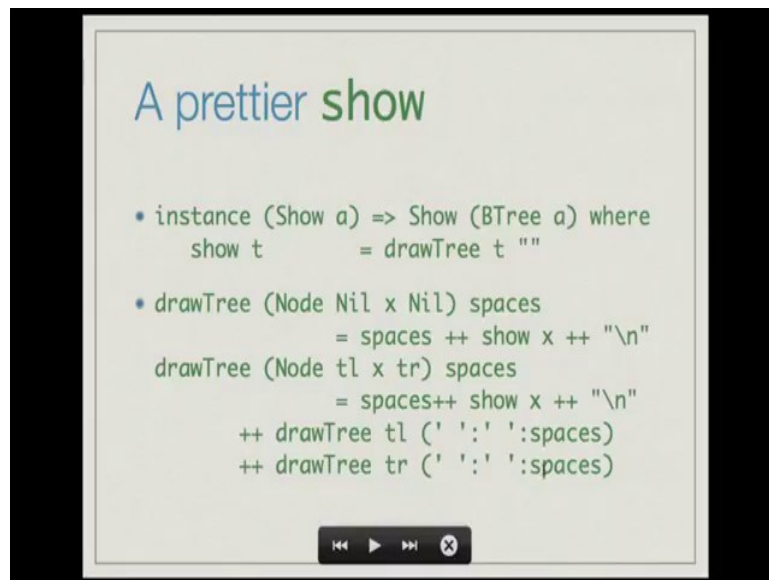
(Refer Slide Time: 19:23)



We defined it as follows using an instance declaration, instance (Show) a implies Show (BTree a), where Show t, t is the tree here, is just drawTree t "", "" is empty string. drawTree will draw the layout of the tree, where the second parameter which is the string tells how many spaces to insert before drawing the current sub tree. So, drawTree is a function with signature (Show a) => BTree a -> String -> String, this string is the number of spaces that have to be displayed before the current sub tree is displayed and this string is actually the output, the layout of the tree.

drawTree Empty spaces = spaces ++ "\*\n" which denotes the new line character. We will see that you will encounter this case only when this is the sole empty subtree of a Node.

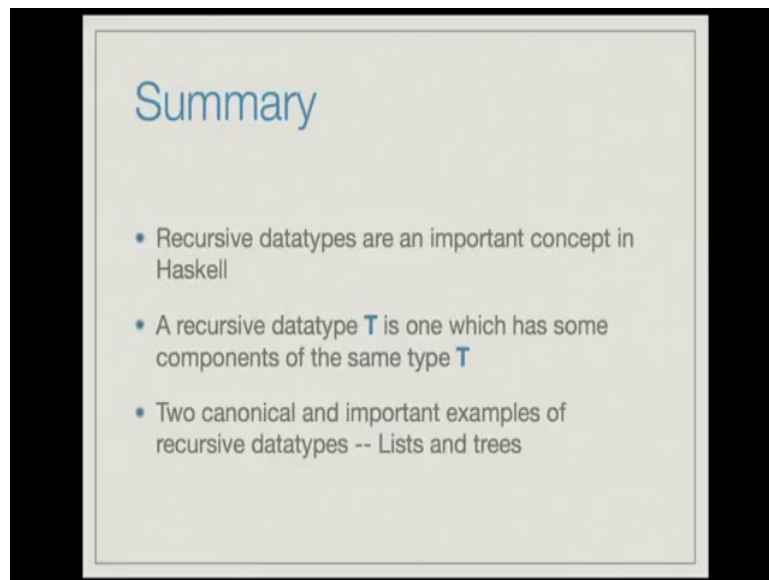
(Refer Slide Time: 20:47)



The other cases when you have a Node x both of whose subtrees are empty, (Node Nil x Nil), drawTree (Node Nil x Nil) spaces is nothing but, spaces ++ show x ++ “\n”. drawTree of (Node tl x tr) spaces, where tl is the left subtree and tr is the right subtree, spaces is nothing but, spaces ++ show x ++ “\n”. Remember that we always add this much, this many spaces before displaying the current subtree, plus you append that with drawTree of tl.

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 concatenate that with drawTree of tr which is the right subtree, again giving two spaces adding two more spaces in the beginning of laying out the right subtree.

(Refer Slide Time: 21:58)



So in 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 its components of the same type **T** and we have seen two canonical and important examples of recursive data types, lists and trees. In the next lecture, we will see more on trees.