

Introduction To Haskell Programming

Prof. S. P Suresh

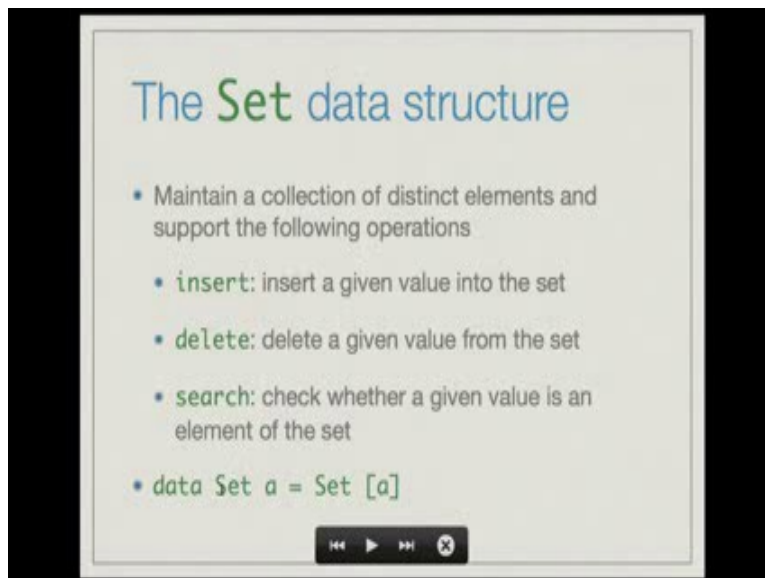
Chennai Mathematical Institute

Module # 06

Lecture – 02

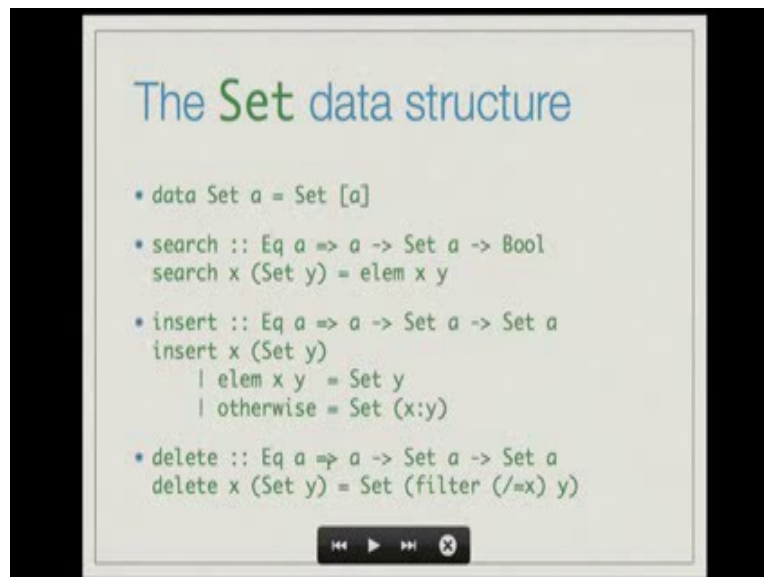
Binary Search Trees

(Refer Slide Time: 00:08)



In this lecture, we shall introduce a very important data structure the binary search trees. Imagine that we want to implement a Set data structure. It is defined as follows. You want to maintain a collection of distinct elements and support the following operations. Insert - it inserts a given value into the set; delete which deletes a given value from the set and search which checks whether a given value is an element of the set or not. Here is one way of defining this. `data Set a = Set [a]`. Recall again that the Set on the left hand side is the type constructor, the Set on the right hand side is the value constructor. The name of the new type that we have defined is Set a and it has a single constructor Set which takes list a, [a] as a argument and outputs an object of type Set a.

(Refer Slide Time: 01:10)



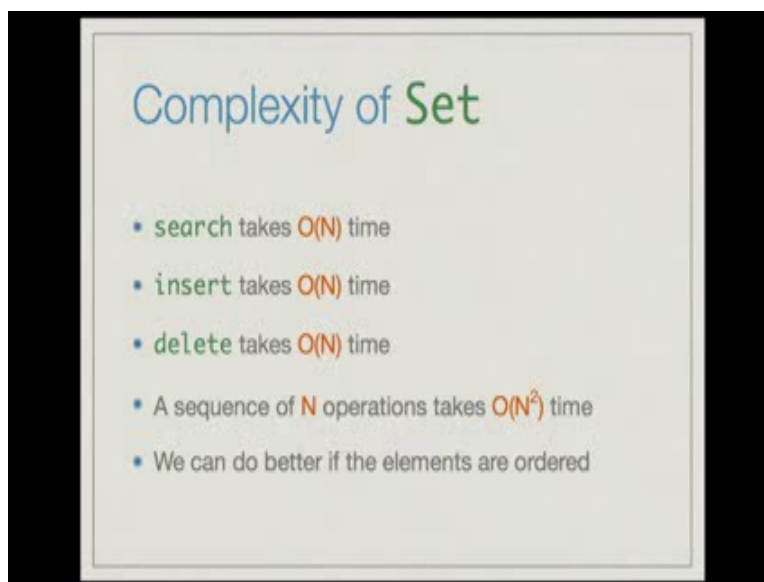
Here is how we might implement the three functions that we described earlier. Search is the function with the following signature. `Eq a => a -> Set a -> Bool`. Search of `x` and `Set y`, searching whether `x` belongs to `Set y`, recall here that `y` is a list consisting of elements of type `a`, so it is the constructor. It is just searching whether `x` is an element of `y`, it is given by the primitive function, the built in function `elem` which checks whether `x` is an element of the list `y`. So, `search x (Set y) = elem x y`, is the implementation of search.

Insert can be implemented as follows. Insert as a function with signature `Eq a => a -> Set a -> Set a`. This is the original set and this is the set that we get after the element has been inserted. Insert of `x` inside `Set y`, there are two cases; if `x` is an element of `y`, you just return `Set y`; this is because we want to maintain a set of distinct elements. If `x` is already present, we don't have to do anything. Otherwise, you insert `x` in `y`, this you do by the simple interface of attaching `x` to the front of the list `y`. So in the otherwise case, you define it to be `Set (x:y)`.

Here is how you might ((Refer Time: 02:49)) define delete. Delete is the function with signature `Eq a -> a -> Set a -> Set a`. Delete of `x` from `Set y` is nothing but the `Set` which is gotten by applying `filter (/=x)` on `y`. Recall that `filter` is a function that applies a predicate to a list and returns all the elements of the list that satisfy the predicate. The predicate that we are giving here is this funny looking object, this is the so called section in Haskell. Recall that `/=` is an operator,

it is a binary operator in Haskell. So whenever you have a binary operator, you can use it as a function by enclosing it in parenthesis; furthermore, you can use it as not just a binary function but a unary function where one of the arguments have been fixed. In this case, we have fixed the argument x , and you want to check whether x is not equal to any of the elements of y . So, this section is gotten by fixing the right argument and making this as unary function. So, this is the predicate that you are applying to each element in the list y . So, this function will return all those elements of y that are not equal to x , which means that it will delete x from y .

(Refer Slide Time: 04:30)

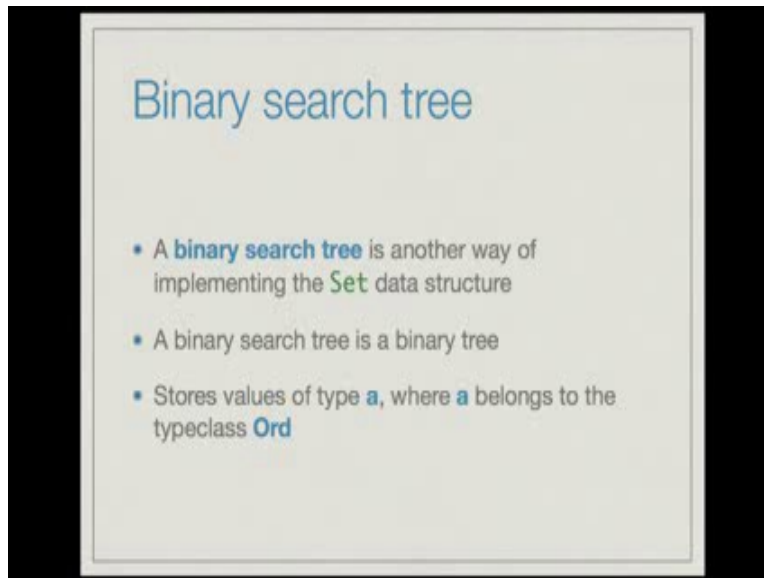


Complexity of Set

- search takes $O(N)$ time
- insert takes $O(N)$ time
- delete takes $O(N)$ time
- A sequence of N operations takes $O(N^2)$ time
- We can do better if the elements are ordered

Let us look at the running time of these operations for the above implementations. Search takes $O(N)$ time, insert takes $O(N)$ time and delete takes $O(N)$ time. Search takes $O(N)$ time, because the built in function `elem` takes $O(N)$ time, where N is the length of the list. Insert also takes $O(N)$ time, because first of all you have to make a search in the list. Delete takes $O(N)$ time, because `filter` takes $O(N)$ time. So a sequence of N operations takes $O(N^2)$ times, and we might want to do better than this. We can do better than this if the elements are ordered, but storing these elements as a sorted list is not the way to go, we need a much better data structure.

(Refer Slide Time: 05:37)

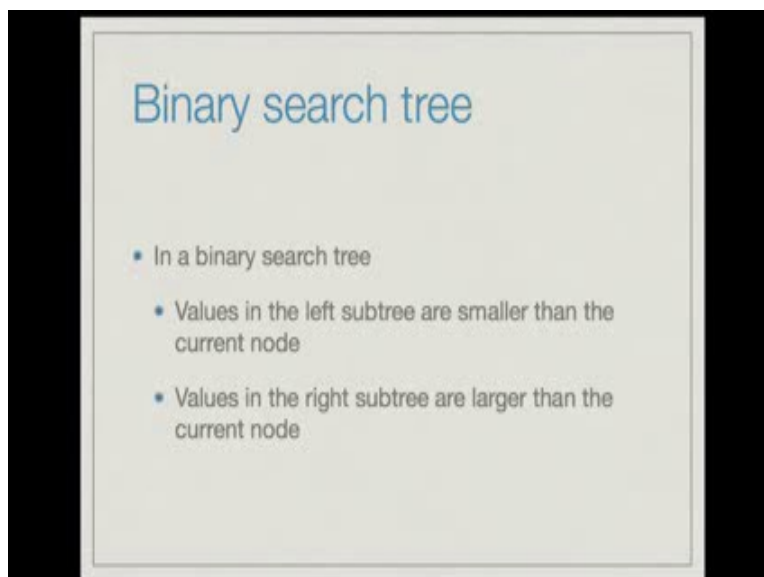


Binary search tree

- A **binary search tree** is another way of implementing the **Set** data structure
- A binary search tree is a binary tree
- Stores values of type **a**, where **a** belongs to the typeclass **Ord**

And that brings us to a binary search tree. A binary search tree is another way of implementing the Set data structure. A binary search tree is just a binary tree that we discussed in the last lecture. It stores values of type `a`, where `a` belongs to the type class `Ord`. So a binary search tree makes sense only for values that can be ordered.

(Refer Slide Time: 06:08)

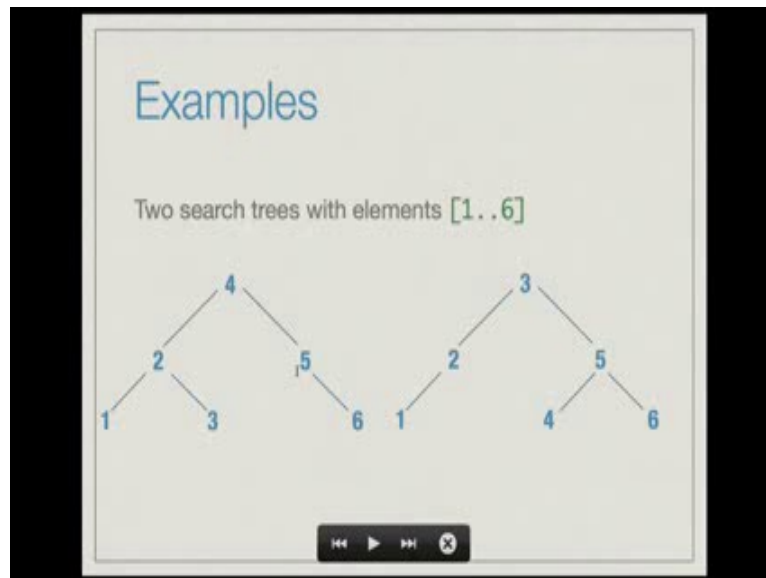


Binary search tree

- In a binary search tree
 - Values in the left subtree are smaller than the current node
 - Values in the right subtree are larger than the current node

The crucial property of the binary search tree is that at any node in a binary search tree, values in the left subtree are smaller than the current node. And values in the right subtree are larger than the current node. And this holds true for every node in the binary search tree.

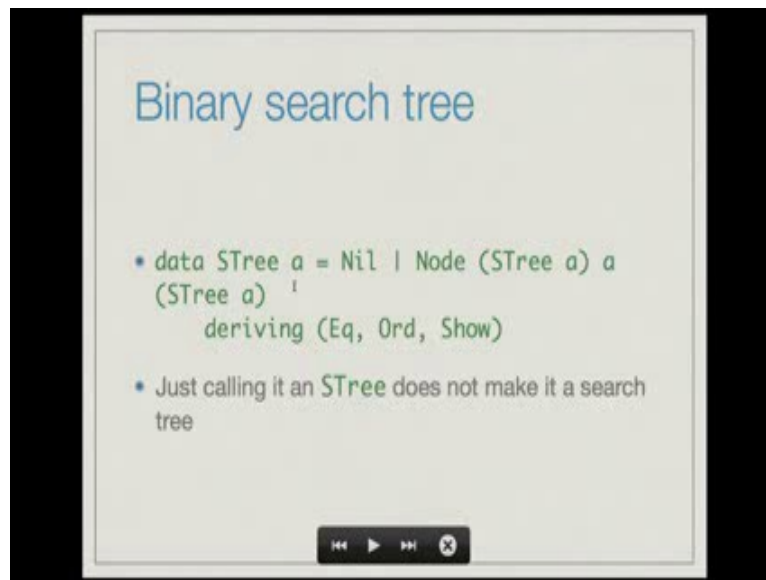
(Refer Slide Time: 06:28)



Here are two examples of binary search trees, which consist of the elements [1..6]. Here is one tree, this tree has root 4 and left subtree consisting of the node 1, 2 and 3, right subtree storing the value of 5 and 6. Notice how the binary search tree property holds true in this tree. Every node in the left subtree has a smaller value than the root, so the left subtree has value as 1, 2 and 3, the root has value 4. Every node in the right subtree namely 5 and 6 has value greater than 4. And recursively in this left subtree, the same property holds; 2 is greater than one and 2 is smaller than 3. Similarly here, 5 is smaller than 6.

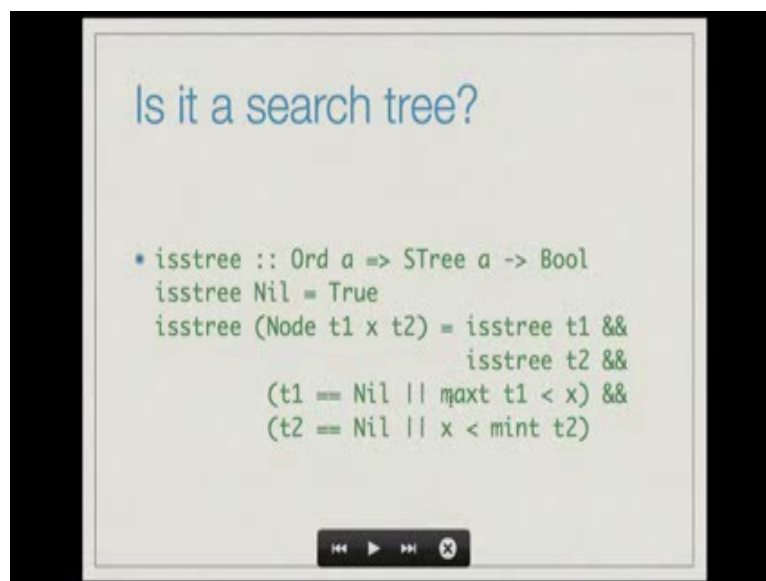
Here is the other tree which represents the same elements. This tree has root 3, and the left subtree has the nodes 2 and 1. In the left subtree 1 is to the left of 2 as it ought to be; and the right subtree has the nodes 4, 5 and 6 and this itself satisfies the search tree property because 5 is the root whose left child is smaller than 5 and whose right child has the value that is greater than 5.

(Refer Slide Time: 07:53)



We define binary search tree as follows. `data STree a = Nil | Node (STree a) a (STree a)`. This is exactly similar to the definition of binary trees that we had earlier, and we derive `Eq`, `Ord`, `Show` etcetera. We might create our own instance of the `show` method or we can just use the default `show` function. Just calling this `STree` does not make it a search tree. We could have objects of type `STree` which completely violate the search tree property.

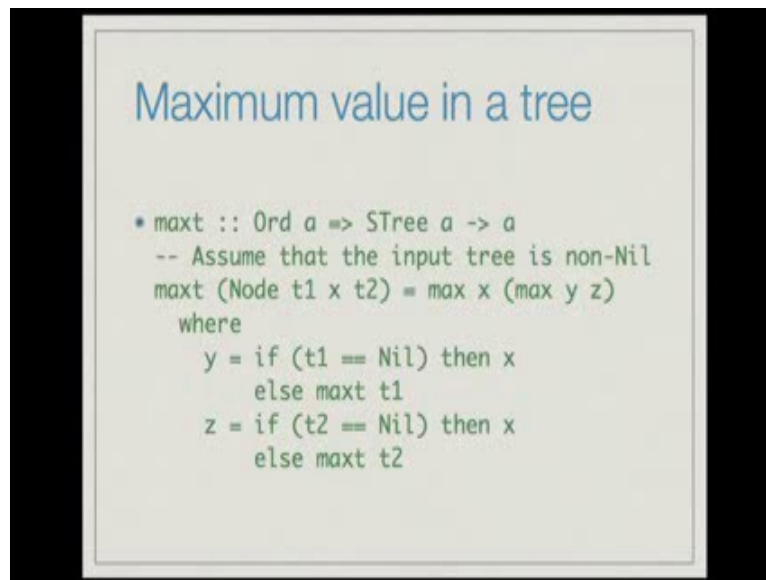
(Refer Slide Time: 08:40)



So here is the function that take a search tree as input and tells whether it is indeed a search tree or not. It takes a binary tree as input and tells whether it is indeed a search tree or not. So this is the function `isstree` which signature is `Ord a => STree a -> Bool`. `isstree` of `Nil` is just `True`, because the empty tree obviously satisfies the binary search tree property. `Isstree` of `(Node t1 x t2)` is nothing but `isstree` of `t1`, the left subtree should recursively satisfy the search tree property, the right subtree also should recursively satisfy the search tree property. So `isstree t1 && isstree t2` and you have to check whether all the values in the left subtree are smaller than `x`. One way to check it, is to check whether the max value in the left subtree `maxt` of `t1` is less than `x`, and we also need to check whether all the values on `t2` are greater than `x`, and one way of checking it, is to check whether `x` is less than the minimum value on `t2` – `x` is less than `mint` of `t2`.

But we had an extra clause here, because the `maxt` function that we will define later is defined only for non-empty trees. So you say either `t1 == Nil` or `maxt t1 < x`, and either `t2 == Nil` but in case it is not null, check if `x < mint t2`, `x` is less than the minimum value in the tree `t2`. This is the definition of is search tree.

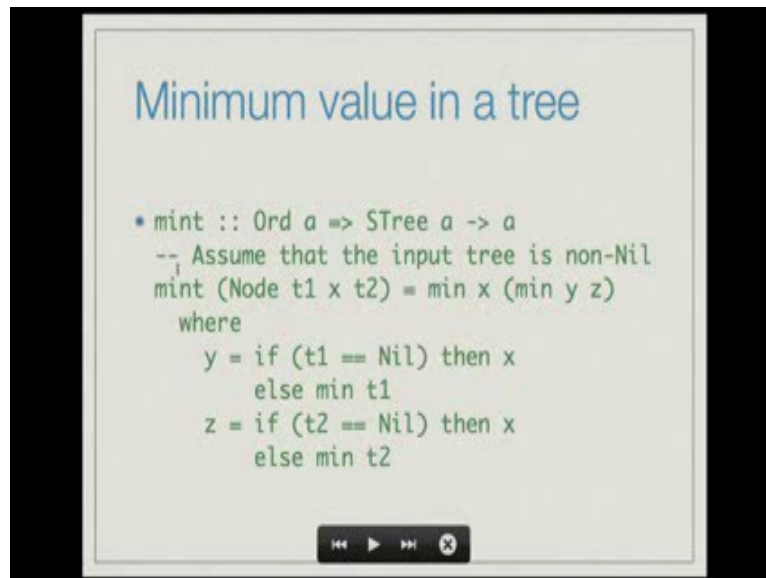
(Refer Slide Time: 10:25)



The maximum value in a tree is computed as follows. `maxt` is a function with signature `Ord a => STree a -> a`. In this function, we assume that the input tree is non-nil. `maxt` of `Node t1 x t2` is nothing but `max` of `x` and `(max of y z)`. Here `y` is supposedly the maximum of `t1`, and `z` is

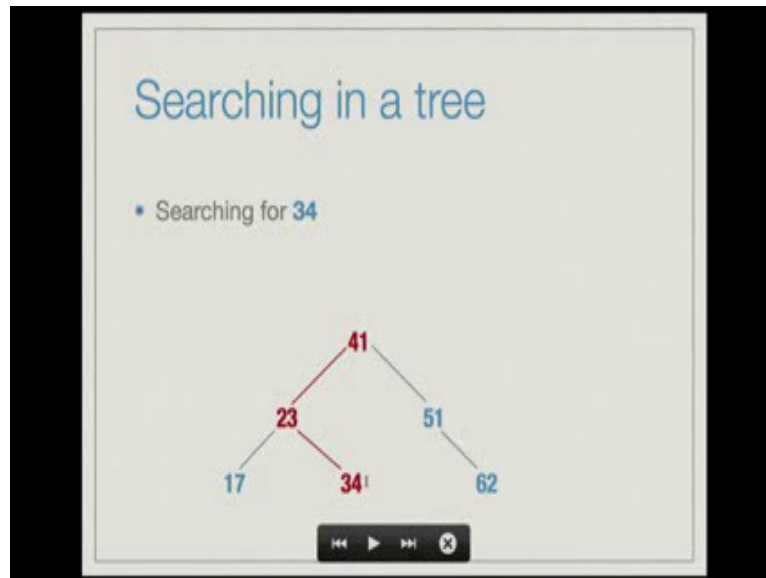
supposed to be the maximum of t_2 , but t_1 could be empty. Therefore, the definition of y is slightly more complicated. The y is defined to be just x incase t_1 is nil; otherwise, it's defined to be maxt of t_1 recursively. And z is defined to be just x incase t_2 is nil; otherwise, it's defined to be maxt of t_2 .

(Refer Slide Time: 11:24)



Symmetrically, you can define the minimum value in a tree also. Mint which is the function with signature $\text{Ord } a \Rightarrow \text{STree } a \rightarrow a$. Again we assume that the input tree is non-nil. Mint of $(\text{Node } t_1 \text{ x } t_2)$ equals min of $(x, (\text{min of } y \text{ z}))$, the minimum of these three values, where y is x incase t_1 is Nil otherwise it is mint of t_1 . And z is x incase t_2 is Nil (refer time: 12:00) otherwise it's mint of t_2 .

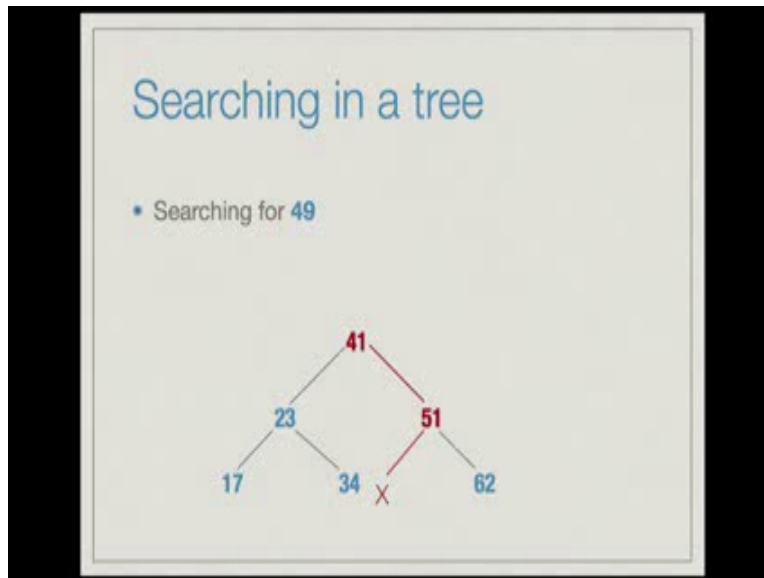
(Refer Slide Time: 12:05)



Now let us look at an important function on tree, namely searching for an element in a tree. Let say we want to search for the element 34 in this tree; this tree consist of the nodes 41, 23, 17, 34, 51, 62. As you can see this is the search tree, because it satisfies the search tree property. Every value in the left subtree is smaller than the root; and every value in the right subtree is greater than the value of the root. And recursively this property holds for both the left subtree and the right subtree. So searching starts at the root of the tree. We check whether 34 is same as the value 41; in this case, we see that 34 is not equal to 41, but we also see that 34 is smaller than 41. And therefore, if the value 34 is represented in this tree, it has to lie in the left sub tree, because the value we have searching for is smaller than 41 and it cannot lie in the right subtree, therefore we go left.

And we check whether 34 is equal to the value in this node; value in this node is 23 which is smaller than the 34. And now we know that if 34 were to be present in this tree, it has to be in the right subtree of this node, because to the left of this node, all node, all values are smaller than 23, and therefore and then smaller than 34. So from here, we go right. And here we see that the value is indeed 34 which is what we were seeking for, so we stop at this point.

(Refer Slide Time: 13:55)



Let us say we are searching for 49 in the same tree. Now 49 again we start the search at the root. Now, 49 is greater than 41, so it has to lie in the right subtree, so we go right. The value that we have to examine now is 51 and 51 is greater than 49, so 49 if it exists at all has to lie to the left of 51, but there is no element to the left of 51. So this search ends in a failure and we say that 49 is not present in the tree.

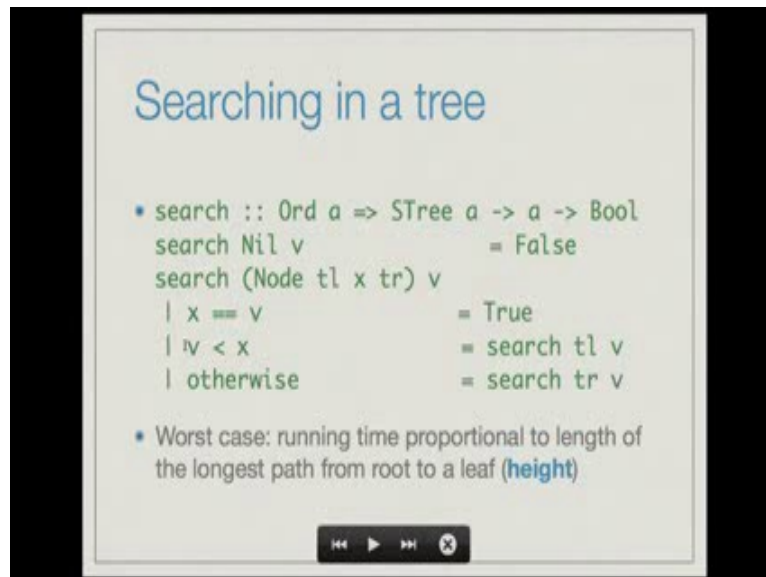
(Refer Slide Time: 14:32)

Searching in a tree

- Searching for value v in a search tree
- If the tree is empty, report **No**
- If the tree is nonempty
 - If v is the value at the root, report **Yes**
 - If v is smaller than the value at the root, search in left subtree (which could be empty)
 - If v is larger than the value at the root, search in right subtree (which could be empty)

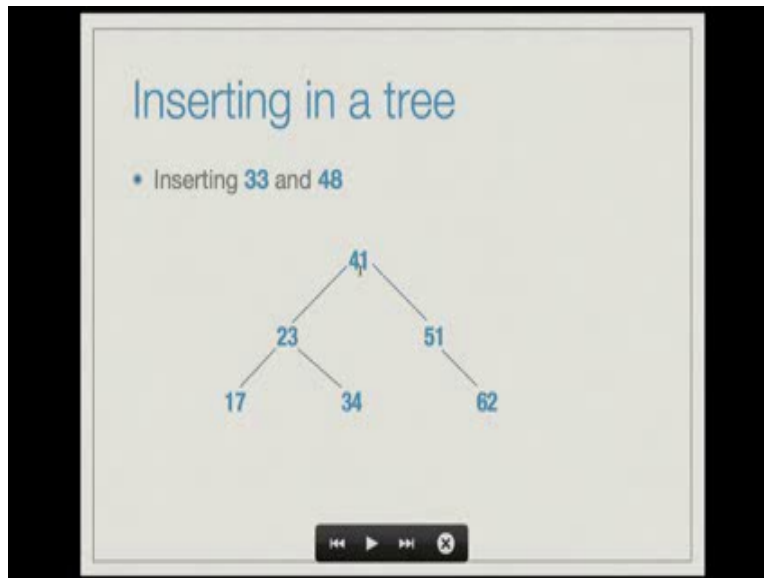
So here is the strategy for searching for an element in a tree. We are searching for a value v in a search tree. If the tree is empty, we report No. If the tree is nonempty then we do a case analysis. If the v is equal to the value at the root, we report yes. If v is smaller than the value at the root, we search in the left subtree, which could be empty, in which case we would report No. If v is larger than the value at the root, we search in the right subtree; again this could be empty, in which case we would report No.

(Refer Slide Time: 15:09)



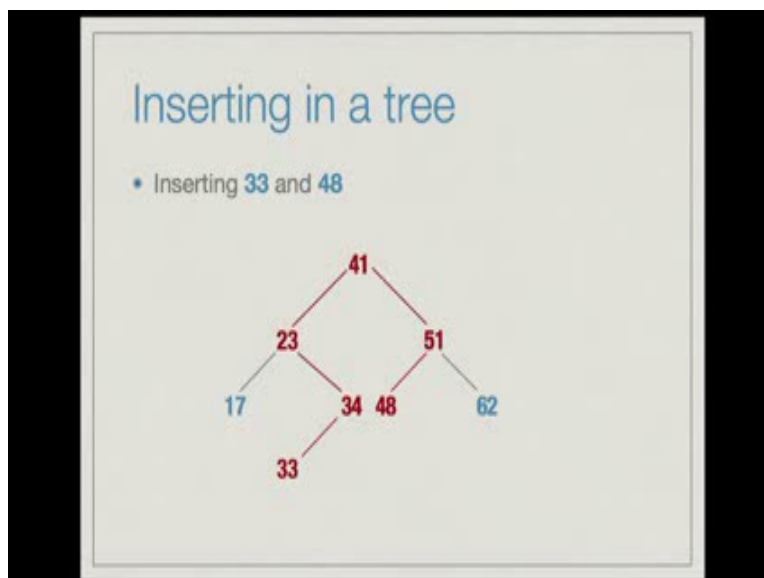
Here is the code that realizes the above strategy. Search which is the function with signature `Ord a => STree a -> a -> Bool`. `search Nil v`, if we are searching for the value v in the empty tree, then the output is false. Search of v in `(Node tl x tr)`, this is the tree with left subtree tl , right subtree tr and the value at the root equals x . Here there are three cases, incase x is equal to v then we say `True`; incase v is smaller than x then we search in the left subtree – `search tl v`. Otherwise, we search in the right subtree – `search tr v`. The worst case running time of this procedure is proportional to the length of the longest path from root to a leaf, because we start at root and keep descending the tree till we reach a leaf or we reach the value that we are seeking for, but we are talking about the worst case. And the length of the longest path from root to a leaf is nothing but the height of the tree. So in the worst case, the running time of search is proportional to the height of the tree.

(Refer Slide Time: 16:27)



Here is the other important function that we wanted to look, that we wanted to implement inserting an element in a tree. Let us consider the example of inserting 33 and 48 into this tree; the same tree that we considered earlier.

(Refer Slide Time: 16:59)

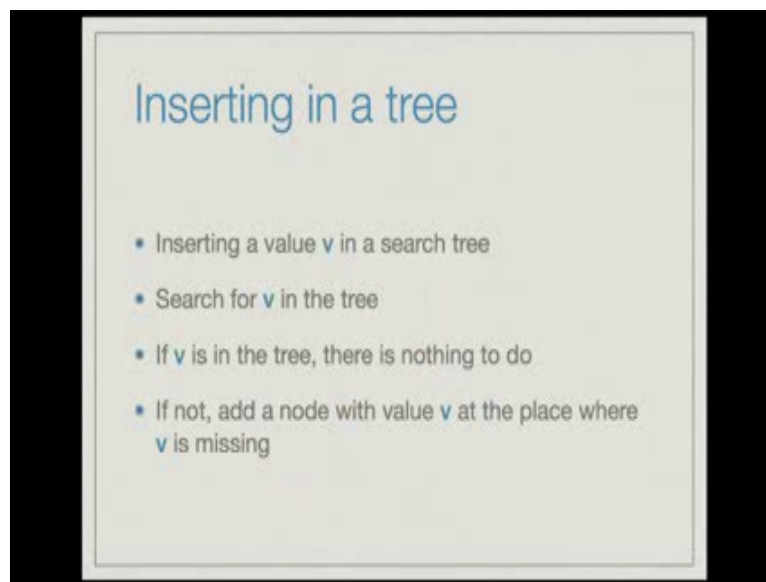


To insert 33 and 48, we first need to determine the location where 33 should go. So we start to search for thirty-three in the list, in the tree. We start at the root and compare the value thirty-

three with the value at the node which is 41, $33 < 41$, so we descend along the left subtree. Now 33 is greater than 23, so it should if it were present in the tree, it should lie in the right subtree, so we descend right. Now we see a node 34; $33 < 34$, but there is nothing to the left of 34. If 33 were present in the tree, in the subtree starting from 34, it should be present at the left of 34, but there is nothing to the left of 34. So this tells us that the element 33 is not present in the tree, and we can add it. And the most natural place to add it is, in fact the only correct place to add it is as the left child of 34, so we add it there.

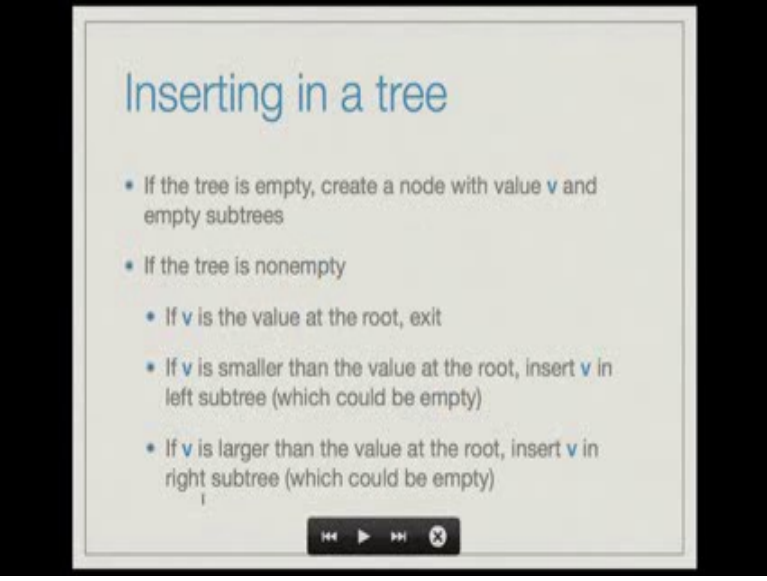
Now let us consider 48. We again check with 41; 48 is greater than 41, so we descend on the right subtree. 51 is greater than 48, so we have to descend down the left subtree, but there is nothing on the left subtree. So this again tells us that 48 is not present in the tree, so we add it as the left child of 51.

(Refer Slide Time: 18:18)



So this is the overall strategy for insert. To insert a value v in a search tree, you search for v in the tree. If v is in the tree, there is nothing to do. If not, add a node with value v at the place where v is missing. And the place where v is missing is found by the search routine.

(Refer Slide Time: 18:41)

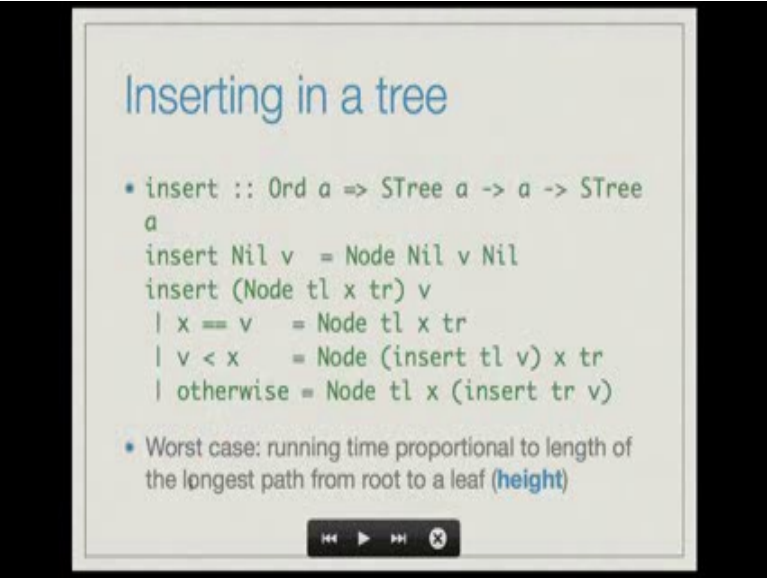


Inserting in a tree

- If the tree is empty, create a node with value v and empty subtrees
- If the tree is nonempty
 - If v is the value at the root, exit
 - If v is smaller than the value at the root, insert v in left subtree (which could be empty)
 - If v is larger than the value at the root, insert v in right subtree (which could be empty)

If the tree is empty, you create a node with value v and empty subtrees. If the tree is nonempty, if the v is the value at the root, you exit. If v is smaller than the value at the root, you insert v in the left subtree; if v is larger than the value at the root, you insert v in the right subtree.

(Refer Slide Time: 19:01)



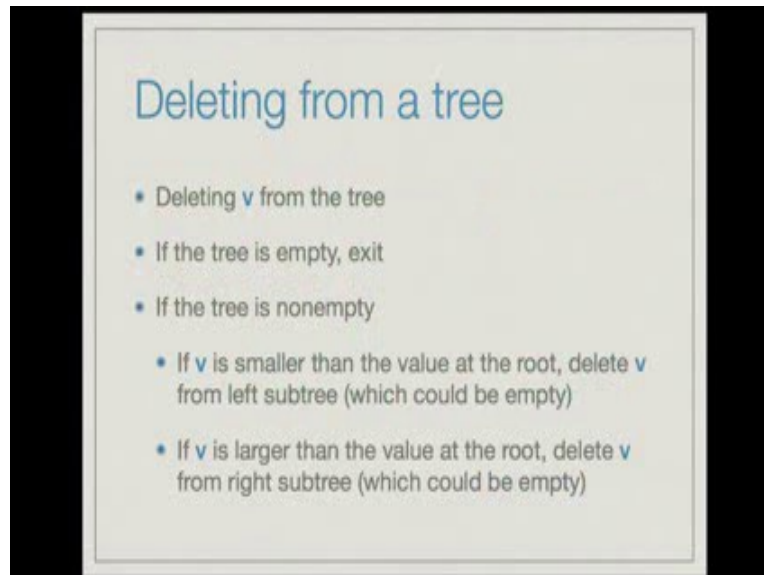
Inserting in a tree

- ```
insert :: Ord a => STree a -> a -> STree a
insert Nil v = Node Nil v Nil
insert (Node tl x tr) v
 | x == v = Node tl x tr
 | v < x = Node (insert tl v) x tr
 | otherwise = Node tl x (insert tr v)
```
- Worst case: running time proportional to length of the longest path from root to a leaf (**height**)

Here is code that realizes the above strategy. Insert which is a function with signature `Ord a => STree a -> a -> STree a`. `insert Nil v = Node Nil v Nil`. This is the node with root  $v$  and two

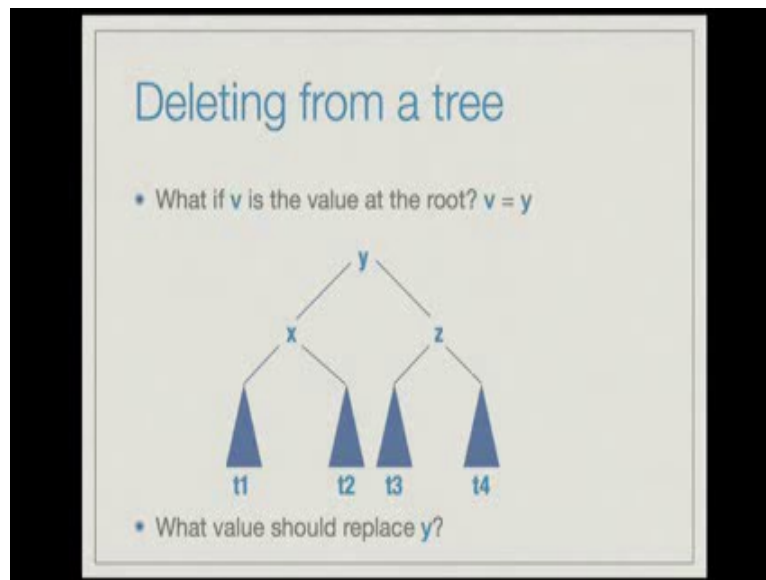
empty subtrees. This is what happens when we insert  $v$  into the empty tree. Insert (Node  $tl$   $x$   $tr$ )  $v$ , we are inserting  $v$  into a tree with root  $x$  at left subtree  $tl$  and right subtree  $tr$ . If  $x$  is the same as  $v$  then we just return that tree (Node  $tl$   $x$   $tr$ ), because there is nothing to do. If  $v$  is smaller than  $x$  then we go down the left subtree, Node (insert  $tl$   $v$ )  $x$   $tr$ . Notice that we modify the left subtree by inserting  $v$  into  $tl$  and then we make that new tree as the left subtree of the overall tree. So you first compute insert  $tl$   $v$  and then make that the left tree of this tree which is given by Node (insert  $tl$   $v$ )  $x$   $tr$ . Otherwise, which means that  $v$  is greater than  $x$ , you insert  $v$  in the right subtree; Node  $tl$   $x$  (insert  $tr$   $v$ ). Again in the worst case, the running time of this routine is proportional to the length of the longest path from root to leaf or the height of the tree.

(Refer Slide Time: 20:29)



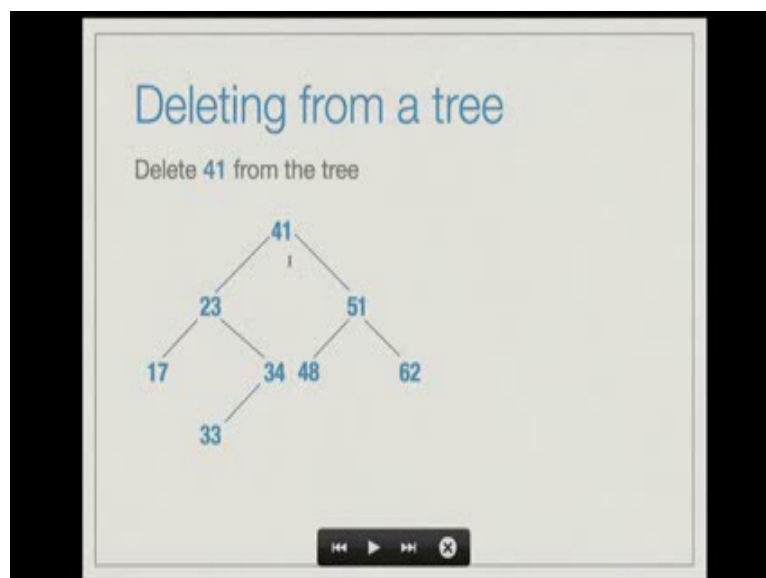
Let us now consider deletion from a tree. If you want to delete  $v$  from the tree, if the tree is empty, we can just exit, because there is nothing to delete. If the tree is nonempty, then if  $v$  is smaller than the value at the root then you delete  $v$  from the left subtree as usual, because  $v$  is evidently not the value at the root, because  $v$  is smaller than value at the root. And  $v$  cannot lie on the right subtree, therefore  $v$  has to lie in the left subtree, and we descend down the left subtree and delete  $v$ . If  $v$  is larger than the value at the root, we similarly delete  $v$  from the right subtree.

(Refer Slide Time: 21:21)



The important case.. The important case is when  $v$  is equal to the value at the root. For example, let's consider the tree with root  $y$ , left subtree consisting of the node  $x$  and its subtree is  $t1$  and  $t2$ . Right subtree being a node  $z$  with left subtree  $t3$  and right subtree  $t4$ . Now if we want to delete  $v$ , we have to remove  $y$  from this tree. But if you remove  $y$ , what node should be should replace  $y$ ,  $zx$  or  $z$  or something else? One can easily see that it is not possible to blindly push  $x$  or  $z$  up the tree.

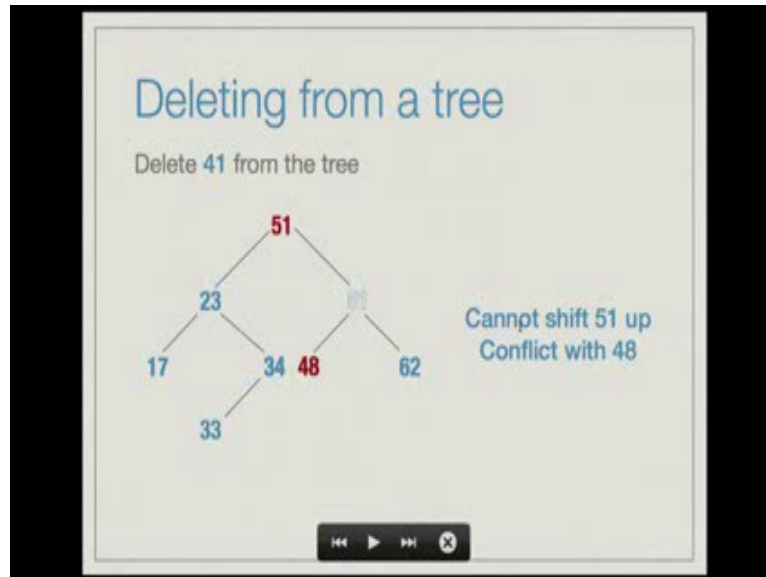
(Refer Slide Time: 22:17)





This is illustrated in the following example. Suppose we want to delete 41 from this tree; this is the tree that we considered earlier after we have inserted 33 into that tree.

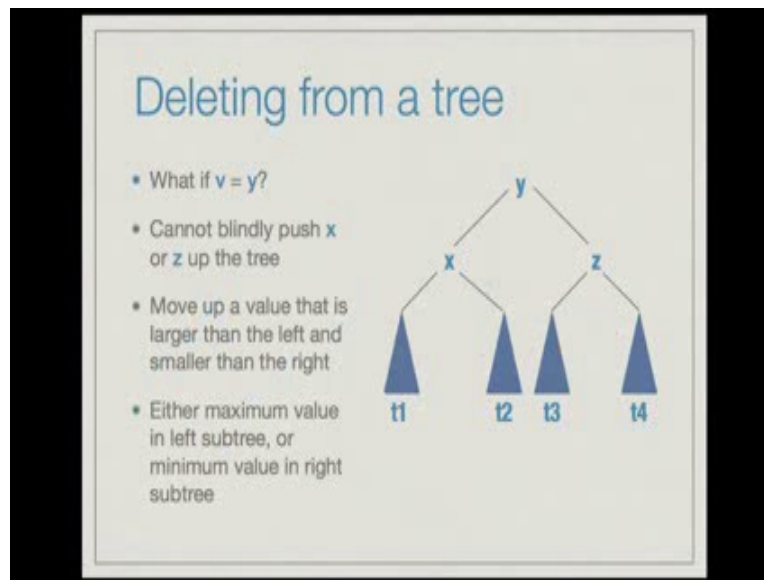
(Refer Slide Time: 22:34)



Suppose we want to delete 41 from this tree, we cannot shift 23 up, because there is a conflict with the value 34. 34 earlier, it was in the right subtree of a node having the value 23, but now it is part of left subtree of the root and the root has value 23. And any node in the left subtree has to be smaller than the value at the root. And if we push 23 up the tree to the root then there is a conflict between 23 and 34.

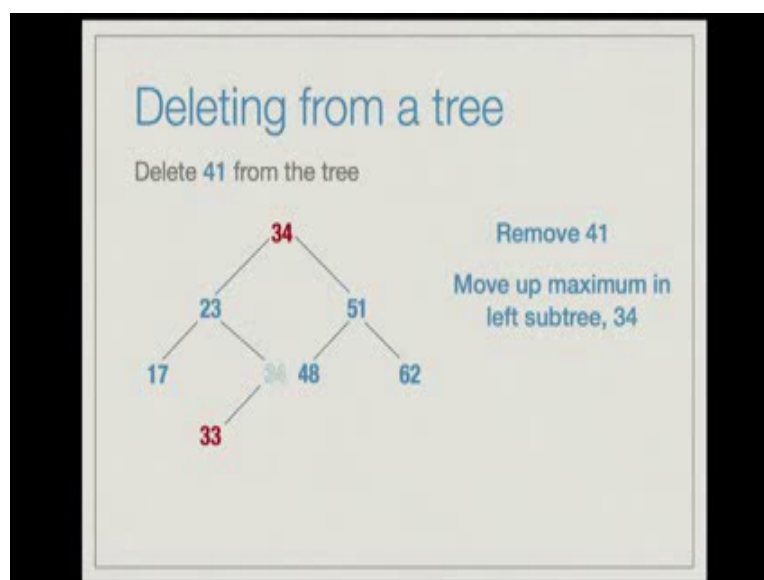
Can we therefore push 51 to the root? We cannot shift 51 up to the root, because there is a conflict between 51 and 48. Again 48 lies in the right subtree of the node which contains the value 51. And according to the search tree property, 48 has to be greater than 51, but 48 is smaller than 51. So there is a conflict again.

(Refer Slide Time: 23:36)



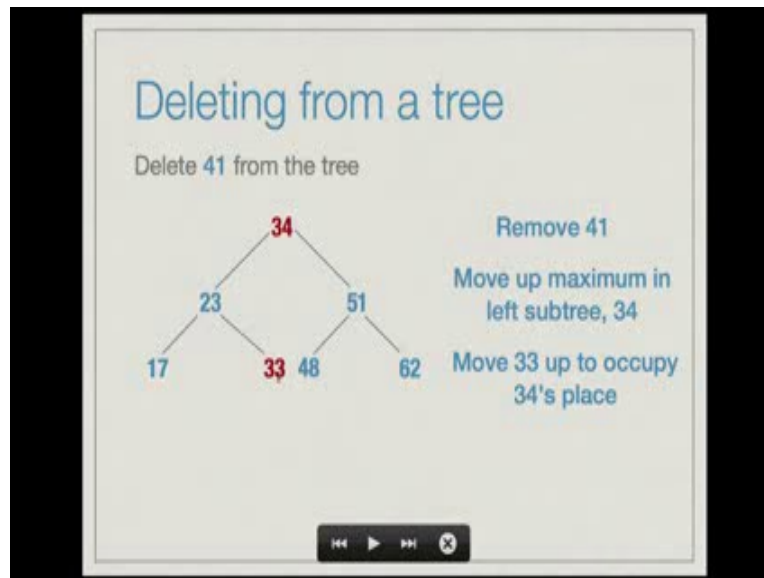
How do we resolve this conflict. As we saw earlier we cannot blindly push  $x$  or  $z$  up this tree. The solution is to move up a value to the root that is larger than the left subtree and smaller than the right subtree, which means either the maximum value in the left subtree, which will lie somewhere here, or the minimum value in the right subtree, which will lie somewhere here. So you should pick a value which lies somewhere inside  $t2$  and replace  $y$  with that or we can pick a value which lies somewhere within  $t3$  the smallest value in  $t3$  and replace  $y$  with that.

(Refer Slide Time: 24:44)



To consider the same example again, suppose we want to delete 41 from this tree. We remove 41, and we find the maximum value in the left subtree which is 34, and we move 34 up to the root, but now this leaves a hole at the position where 34 was.

(Refer Slide Time: 25:05)



One can argue that 34 cannot have a right subtree. If 34 had a right subtree then there would be a value that is larger than 34 in the left subtree of the original tree, therefore 34 will not have a right subtree, can only have a left subtree. We can push the left subtree up to occupy the place of 34. In this case, the left subtree consists of a single node 33, so we move 33 up to occupy 34's position. (Refer Slide Time: 25:32)

## Deleting the maximum value

- Keep going right till you reach a node whose right subtree is empty
- Remove the node
- Replace the node by its left subtree

The diagram illustrates a binary tree structure. The root node is labeled 'x'. It has a left child labeled 'tx' and a right child labeled 'y'. Node 'y' has a left child labeled 'ty' and a right child labeled 'z'. Node 'z' has a left child labeled 'tz'. A dashed line connects 'x' to 'y', showing the path taken to find the maximum value. The nodes are represented by blue triangles.

So here is how we delete the maximum value. Keep going right till you reach a node whose right subtree is empty. From x, we keep going right till we see y here and then we see a z here, whose right subtree is empty. At this point, we know that we have found the maximum value in the tree, whose root is x. Now we remove this node. Since it does not have a right subtree, we do not need to worry about that. We just push tz in the place of z, this is how we delete the maximum value of a tree.

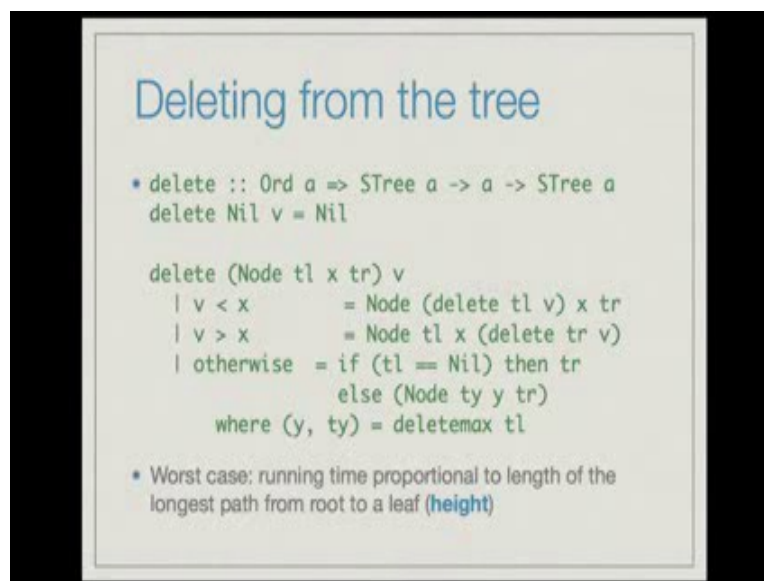
(Refer Slide Time: 26:14)

## Deleting the maximum value

- `deletemax :: Ord a => STree a -> (a, STree a)`
- At the rightmost node
  - `deletemax (Node tl x Nil) = (x, tl)`
  - Always descend right
  - `deletemax (Node tl x tr) = (y, Node tl x ty)`  
 where `(y, ty) = deletemax tr`
- `deletemax` returns the maximum value and the modified tree

Here is the code, `deletemax` is a function whose signature is `Ord a => STree a -> ( a , STree a)`. `Deletemax` as you see returns both the maximum value of the tree as well as the modified tree after having deleted the maximum value. There are two cases to consider. `Deletemax` of `(Node tl x Nil)` is the first case, this is the case where we are already at the right most node then we know that `x` is the maximum value, so we return `x` and we have to return the modified tree after deleting `x`. In this case, `x` is the root, we have deleted `x`, there is nothing to do, so `tl` is the tree that we get after deleting `x`. The other case is when there is a non null right subtree. `Deletemax` of `(Node tl x tr)`, where `tr` is the right subtree which is not equal to `Nil` is `(y ,Node tl x ty)` where `y` is the maximum value in `tr` and `ty` is the tree that you get after deleting `y` from `tr`. So `deletemax` of `(Node tl x tr)` is the pair `(y , node tl x ty)`, where `(y , ty)` equals `deletemax` of `tr`.

(Refer Slide Time: 27:43)

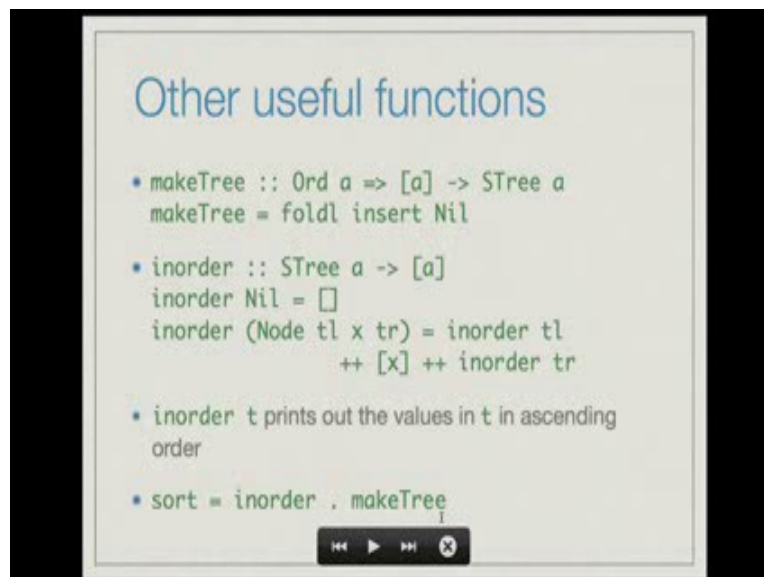


Having defined how to delete the largest value in a subtree, let's return to the original task which was to delete an element from a tree. `Delete` is a function whose signature is `Ord a => STree a -> a -> STree a`. `Delete Nil v = Nil`. If you want to delete `v` from the empty tree, you do not have to do anything. `delete (Node tl x tr) v`, this is a non trivial tree. In case, `v` is smaller than `x`, then you delete `v` from `tl` from the left subtree; so in case, `v` is smaller than `x`, the result is `Node ( delete tl v) x tr`. In case `v` is larger than `x`, the result is `Node tl x (delete tr v)`. Otherwise this is the case when `v` is equal to `x`; if `tl` is the empty tree, then you have a tree whose root is `x`, whose right

subtree is tr and whose left subtree is empty and you are deleting x, there is nothing to do, you just return tr.

Else you have a tree, where x is the root, right subtree is tr and left subtree is tl, you delete the maximum from tl and that will give you (y,ty). The maximum element is y, ty is the tree that you get after deleting y from tl; and the new tree that you form is nothing but (Node ty y tr). Recall that our strategy is to find the maximum element in the left subtree and replace x by that value, so y is the maximum element in the left subtree, we replace x by y. So y becomes the new root, ty is the new left subtree and tr is the right subtree as usual. Again in the worst case, this code runs in time proportional to the height of the tree.

(Refer Slide Time: 29:46)

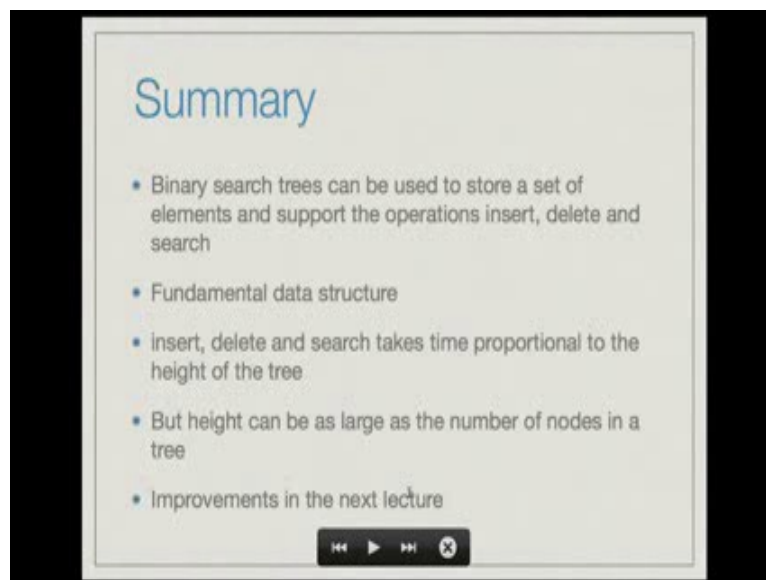


Here are some other useful functions. makeTree which makes a tree from a list of elements; the signature is `Ord a => [a] -> STree a`. makeTree is nothing but `foldl insert Nil`. Recall that `foldl` is the function that takes a function as the first argument, an initial value as the second argument and then applies this function repeatedly on a list that is given as input by using this as an initial value. So the initial value that we give is the empty Tree denoted by `nil`. So what this will do is it will first insert the first element of the list into the empty tree. Then it will insert the second element of the list into the resulting tree, the third element into that tree etcetera. So this will

achieve the effect of inserting all the elements in the given list into the tree, into the empty tree, so it forms a tree with all these elements.

inorder is another useful function. inorder of Nil is nothing but empty list. Inorder of (Node tl x tr) is to recursively print inorder the left subtree and then print the root and then print inorder the right subtree. This is called inorder, because the root is placed in between the inorder of the left subtree and inorder of the right subtree. You can observe that inorder t prints out the values in the tree t in ascending order. And therefore, one way to sort a list of integers or a list of any values of type Ord a is to makeTree with that list and then apply the function inorder on that tree, which we define as inorder.makeTree. The dot syntax in Haskell allows to compose two functions. So sort is the composition of inorder and makeTree.

(Refer Slide Time: 31:58)



So in summary binary search trees can be used to store a set of elements and support the operations insert, delete and search. It is the fundamental data structure that is widely studied. Insert, delete and search takes time proportional to the height of the tree. But the height of the tree can be as large as the number of nodes in a tree. So we will see how to improve this in the next lecture.