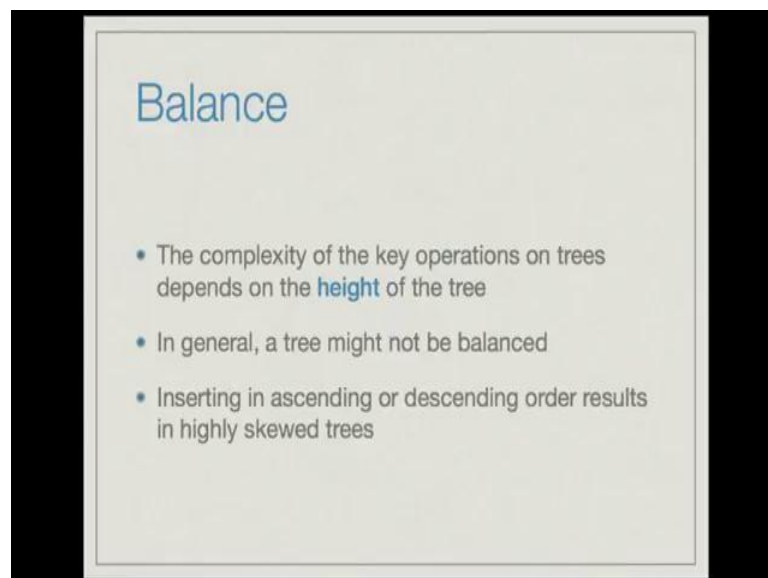


Function Programming in Haskell
Prof. Madhavan Mukund and S. P. Suresh
Chennai Mathematical Institute

Module # 06
Lecture – 03
Balanced Search Tree

In this lecture, we shall study AVL Trees, which is an example of a Balanced Searched Tree.

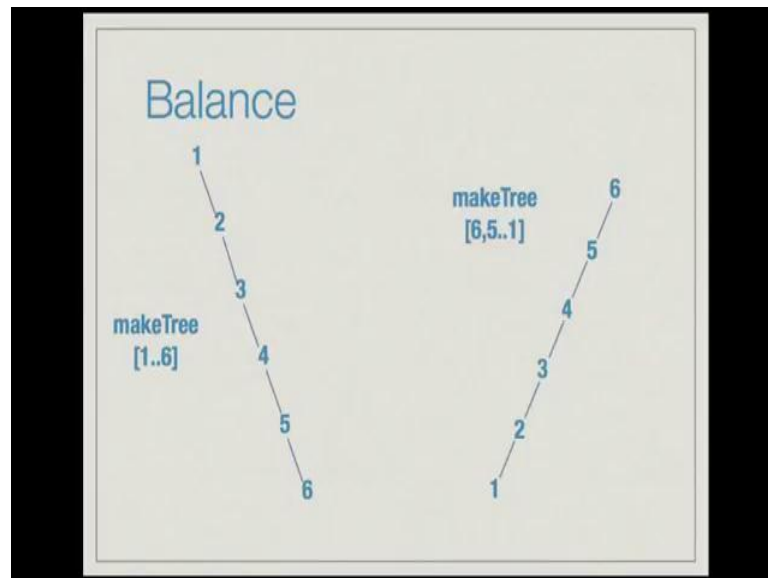
(Refer Slide Time: 00:11)



Recall that in the previous lecture, we studied implementations of the set data type, which maintains the set of elements and supports the operations search, insert and delete. We saw two implementations, one which was to store the elements as a list and on the other, which was to store the elements as a binary search tree. The list implementation was considered inefficient, because each search takes time propositional to the length of the list, in other words the number of elements in the set.

So, N operations can take up to order N squared time. On the other hand, the binary search tree data structure supports insert, search and delete in time propositional to the height of the tree. But, in general, a tree might not be balanced, for instance, inserting elements in ascending or descending order results in highly skewed trees.

(Refer Slide Time: 01:25)

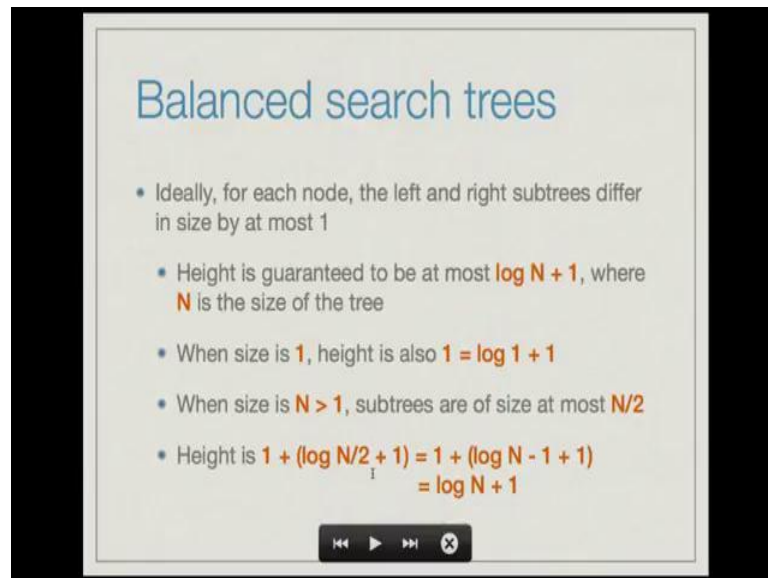


Here are two examples, if we consider make tree of 1 to 6, recall that make tree was a function that starts with the nil tree and inserts all these elements in the list one after the other. So, make tree of 1 to 6 will create a tree that looks like this; it is skewed to the right. Make tree of 6 comma 5 dot, dot 1 inserts elements in descending order into the tree and this tree is skewed to the left. It will start on 6, the next element to be inserted as 5, which will go to the left of 6.

The next element to be inserted is 4, which will go to the left of 6 and to the left of 5 and so on. So, in general, it is possible that a tree has height as much as the number of elements on the tree. This means that the operations on the binary search tree might take up to order N time for search, insert and delete. So, sequence of n operations might still take up to order N squared type.

So, where is the advantage in using a binary search tree? The answer is that we can manage to balance a tree, so that it is of small height. This brings us to the concept of a balanced search tree.

(Refer Slide Time: 02:52)



Balanced search trees

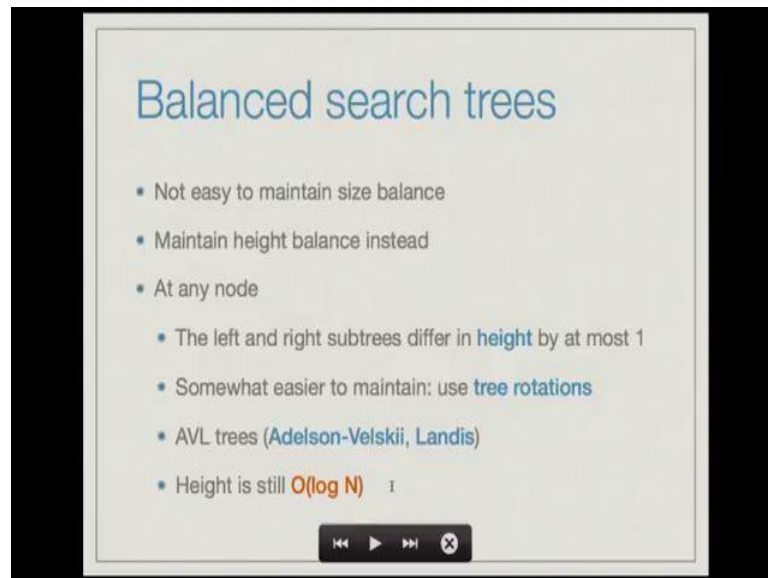
- Ideally, for each node, the left and right subtrees differ in size by at most 1
- Height is guaranteed to be at most $\log N + 1$, where N is the size of the tree
- When size is 1, height is also $1 = \log 1 + 1$
- When size is $N > 1$, subtrees are of size at most $N/2$
- Height is $1 + (\log N/2 + 1) = 1 + (\log N - 1 + 1) = \log N + 1$

Navigation controls: back, play, forward, close

Ideally, we want that for each node, the left and right subtrees differ in size by at most 1. If we maintain this property, then the height of the tree is guaranteed to be at most $\log N$ plus 1, where N is the size of the tree. The proof is as follows, when the size of the tree is 1, the height is also 1, which is the same as $\log 1$ plus 1. When the size of the tree is N greater than 1, then observe that both subtrees are of size at most N by 2, because if each subtree is of size greater than N by 2, then the tree itself would be of size greater than N .

And now, inductively we can see that the height of each subtree is $\log N$ by 2 plus 1, which is the same as $\log N$ minus 1 plus 1, which is the same as $\log N$. And therefore, the height of the tree itself which is 1 plus maximum of the height of the two subtrees is 1 plus $\log N$.

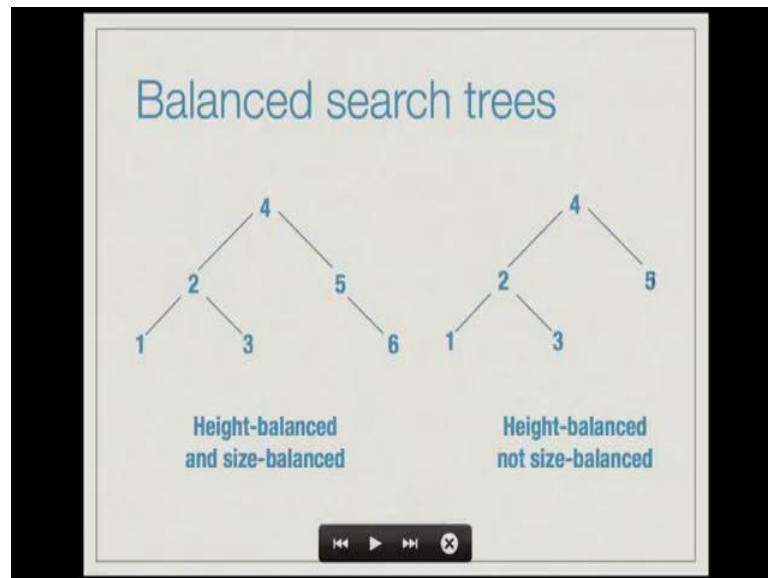
(Refer Slide Time: 04:05)



But, the trouble is that it is not easy to maintain size balance, what we shall ask to do is to maintain height balance instead. This is maintained as follows, at any node we maintain the invariant that the left and right subtrees differ in height by at most 1. In the earlier case, the left and subtrees, the left and right subtrees differed in size by at most 1. But, now we will maintain the somewhat easier to maintain property of the left and right subtrees differing in height by at most 1.

We maintain this property by using tree rotations, which we will describe later. These trees are called AVL trees, named after the inventors of the data structure Adelson Velskii and Landis. One can prove that, if we maintain this property, the height of the tree is order $\log N$, where N is the size of the tree.

(Refer Slide Time: 05:15)



Here is an example, on the left, you see a tree that is both height balanced and size balanced. We can check it as follows, the tree is of size 6, there are three nodes in the left subtree; that is the root node here and there are two nodes in the right subtree and you see that the difference in size of left and right subtrees is 1. There are three nodes in the left subtree; there are two nodes in the right subtree. If we look at the left subtree, the difference in size of the left and right subtree of this tree is 0.

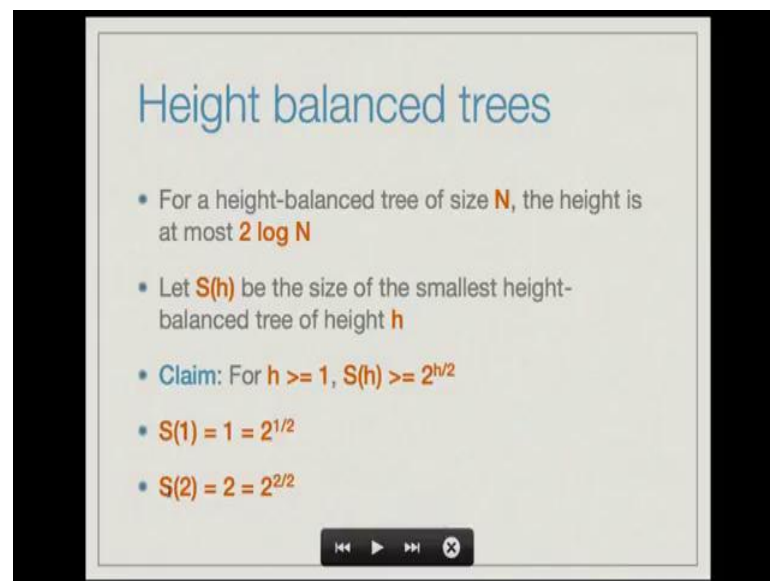
Because, the left subtree here is just the single node 1, the right subtree is just the single node 3 and there are of the same size. So, the left part is size balanced. So, if you consider the right subtree of the original tree, it is this tree, 5, where the right child 6 and with no left child. Here, you see that the left subtree is of size 0 and the right subtree is of size 1.

So, in this tree, we maintain size balance throughout, it is also true the height balance is maintained, because on the left, if you take the overall tree, the left subtree is the height 2, the right subtree is of height 2. So, take this tree, the left subtree is of height 1, the right subtree is also of height 1. So, if you take this subtree the left subtree is of height 0 and the right subtree is of height 1. So, this tree maintains both height balance and size balance.

The tree on the right is height balanced, but it is; however, not size balanced. So, if you take the overall tree, the height of the left subtree is 2, because there is one node here and

it has two children, the height of the right subtree is 1, so the difference is 1. If you consider this subtree, there are no children, so it is trivially height balanced, if you consider this subtree, it is also height balanced, because the height of the left subtree is 1 and height of this subtree is 1. So, both the left and right subtrees of this tree are height balanced, but it is not size balanced, because the left subtree has three nodes and the right subtree has only one node.

(Refer Slide Time: 07:52)



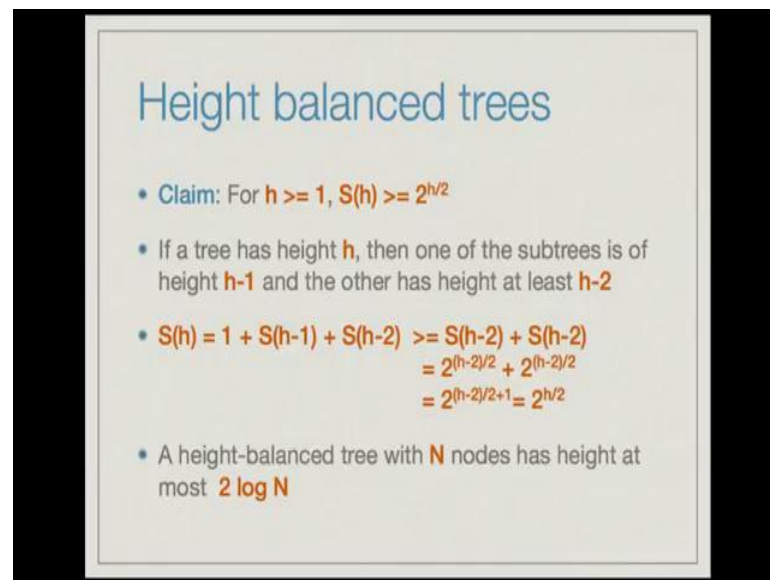
Height balanced trees

- For a height-balanced tree of size N , the height is at most $2 \log N$
- Let $S(h)$ be the size of the smallest height-balanced tree of height h
- Claim: For $h \geq 1$, $S(h) \geq 2^{h/2}$
- $S(1) = 1 = 2^{1/2}$
- $S(2) = 2 = 2^{2/2}$

Before, we move on to the implementation of a height balance tree, let us look at the bounds. We claim that for a height balanced tree of size N , the height is at most twice $\log N$, the proof is as follows. Let us h be the size of the smallest height balance tree of height h , we claim that for h greater than or equal to 1, S of h is greater than or equal to 2 to the h by 2, S of 1 equals 1, which is 2 to the 1 by 2.

Here, we take 1 by 2 to the 0, S of 2 equals 2, which is 2 to the 2 by 2, S of 2 equals 2, because this smallest height balance tree of height 2 that we can create is a node and one child. So, it is the two node tree.

(Refer Slide Time: 08:45)



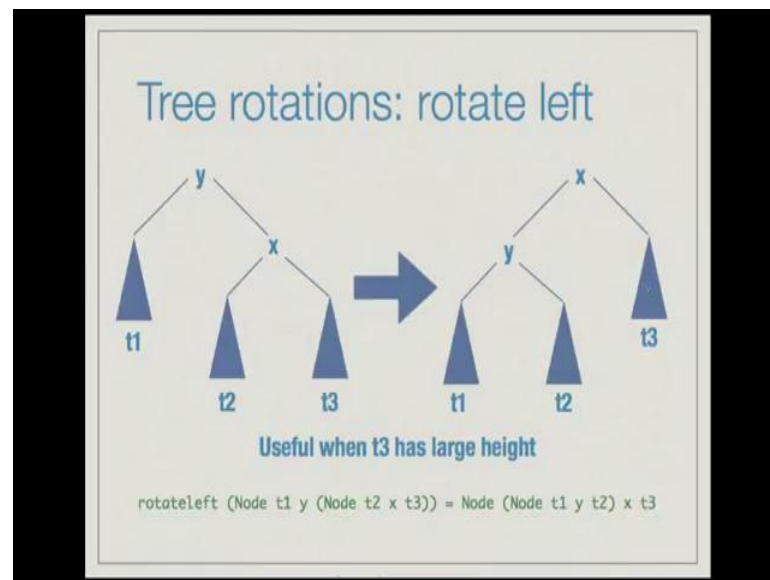
Height balanced trees

- Claim: For $h \geq 1$, $S(h) \geq 2^{h/2}$
- If a tree has height h , then one of the subtrees is of height $h-1$ and the other has height at least $h-2$
- $S(h) = 1 + S(h-1) + S(h-2) \geq S(h-2) + S(h-2)$
 $\quad \quad \quad = 2^{(h-2)/2} + 2^{(h-2)/2}$
 $\quad \quad \quad = 2^{(h-2)/2+1} = 2^{h/2}$
- A height-balanced tree with N nodes has height at most $2 \log N$

In general, if a tree has height h , then one of its subtrees is of height exactly h minus 1, and the other has height at least h minus 2. Now, the height of a tree is defined to be 1 plus the maximum of the height of the two subtrees. So, one of the subtrees definitely has to have height exactly h minus 1 and the other subtree cannot have height smaller than h minus 2, because that would violate the height balance property.

Now, $S(h)$ in general is 1 plus $S(h-1)$ plus $S(h-2)$, which is greater than or equal to this is the loose bound $S(h-2)$ plus $S(h-2)$, which by induction hypothesis is at least $2^{(h-2)/2}$ plus $2^{(h-2)/2}$. This is nothing but, twice $2^{(h-2)/2}$, which is $2^{(h-2)/2+1}$, this simplifies to $2^{h/2}$. So, overall we have $S(h)$ is greater than or equal to $2^{h/2}$, therefore, a height balanced tree with N nodes has height at most $2 \log N$.

(Refer Slide Time: 10:06)



Having established the mathematical bounds, let us look at how to implement a height balanced tree. If you recall, we mention that, we maintain height balance by using tree rotations, there are two types of tree notations, rotate right and rotate left, rotate right is defined as follows. We have a tree with root x and a left subtree with it is own root y and subtree is t 1 and t 2, the right subtree of the bigger tree is t 3.

Now, we rotate this to the right by which we mean that will push this y up to the root and pull the x down, by pulling the x down, we mean that we will make x the right child of y. But, if so, what do we do with the original right child of y, we make that the left child of x, recall that y was the earlier the left child of x, but now once we pull x down, x does not have a left sub child, left child, so we can make t 2 to the left child of x.

In doing this, we still maintain the binary search tree property, notice that, every node in t 1 is smaller than y, t 1 appears to the left of y, y appear to the left of x, y is smaller than x. But, now x appears to the right of y and clearly y is smaller than x or in other words, x is greater than y, every node in t 2 appears to the right of y, so it is greater than y. But, on the other hand, it appears in the left subtree root of vertex, so it is smaller than x.

If you look at the new configuration t 2 appears to the left of x, so every node in t 2 is smaller than x, but t 2 is part of the right subtree of y, therefore, every node in t 2 is greater than y. So, the search tree property, if it was maintain earlier, it would also we

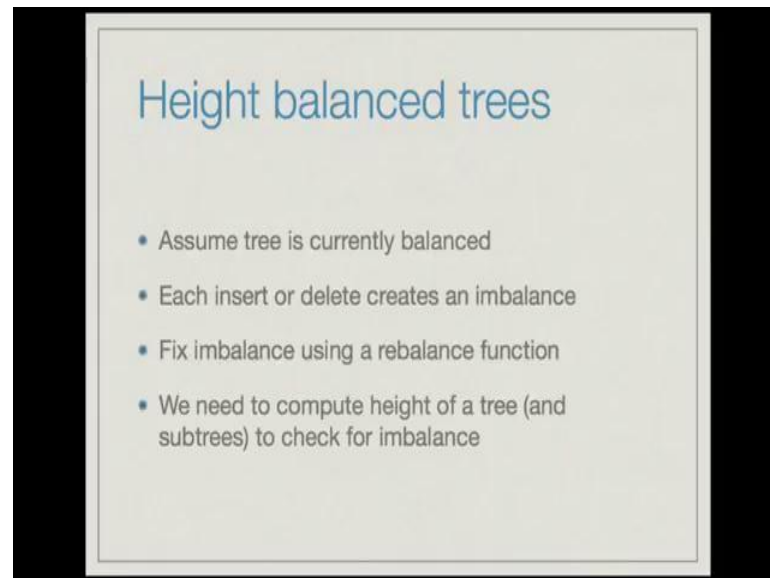
maintain after the rotate. This operation is useful when t_1 has large height, as you can see now earlier the subtree to the left of x had height $1 + \text{height of } t_1$.

Now, the left subtree of the root has only height t_1 , so in this manner, we can reduce the height of the left subtree of the overall tree. The rotate right operation can be implemented in Haskell as follows rotate right of node, node t_1 , y t_2 ; we just state this pattern here, x which is the root, t_3 , which is the right subtree equals we just right the pattern for this tree. Node t_1 , which is the left subtree, y which is the root and node t_2 x t_3 , which is the right subtree.

The symmetric operation is rotate left, where the original tree has root y and right child x with its own subtrees t_2 on t_3 , the left child of the original tree is t_1 . Now, in case t_3 has large height, we might want to push t_3 upwards the root of the tree, which is what we are doing. We are achieving this by pushing x towards the root of the tree and pulling y down. So, when we push x to the root of tree and pull y down, y becomes the left child of x , the previous left child of x namely the subtree t_2 now becomes the right child of y , because now earlier the right child of y was x .

Now, the right child of y has been freed up, so we can let t_2 occupy that position. Again, we can check that, if the original tree satisfies the search tree property after the rotate left also, the tree will continue to satisfy the search tree property. This operation is useful and t_3 has a larger height and it exactly the inverse of rotate right and the Haskell description is again very simple. Rotate left of this pattern, which is specified by node t_1 y node t_2 x t_3 equals node of left subtree which is node t_1 by t_2 x , which is the root and t_3 , which is the right subtree.

(Refer Slide Time: 14:55)



Having seen the operations rotate right and rotate left, we will now see, how to use those two operations, assume that the tree is currently balanced, each insert or delete operation performed on the tree might create an imbalance. What we do is to fix the imbalance using a rebalance function, which involves rotate right and rotate lefts. Before, we actually describe the rebalance function; we will take care of some preliminaries.

We need to compute height of a tree and its subtrees to check for imbalance. Recall that, we are implementing a height balanced tree and imbalance means that, we have a node, such that, one subtree and other subtree differ in height by at least 2.

(Refer Slide Time: 15:54)

Height balanced trees

- We need to compute height of a tree (and subtrees) to check for imbalance
- $\text{height Nil} = 0$
 $\text{height (Node tl x tr)} = 1 + \max (\text{height tl}) (\text{height tr})$
- This takes $O(N)$ time
- Save effort by storing height at each node

So, we need to compute the height of a tree and its subtrees to check for an imbalance, usually height would be defined as follows, height of nil equal 0, height of node t l x t r equals 1 plus max height t l and height t r. But, unfortunately this definition takes order N time to compute height and we would not like to spend order N time for performing an operation, which would be performed multiple times during each insert and delete. We can save this effort by storing the height at each node.

(Refer Slide Time: 16:41)

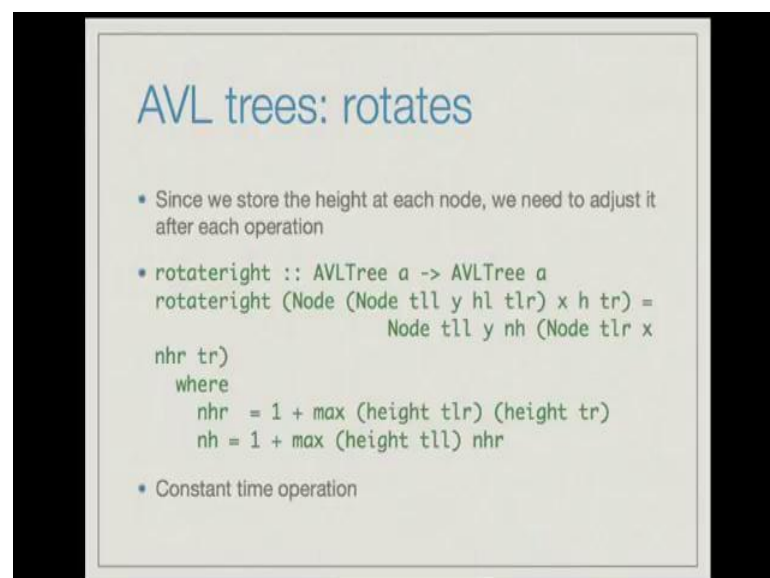
AVL trees

- $\text{data AVLTree a} = \text{Nil} \mid \text{Node (AVLTree a) a Int (AVLTree a)}$
- $\text{height} :: \text{AVLTree a} \rightarrow \text{Int}$
 $\text{height Nil} = 0$
 $\text{height (Node tl x h tr)} = h$
- We also need a measure of how skewed a tree is: its **slope**
- $\text{slope} :: \text{AVLTree a} \rightarrow \text{Int}$
 $\text{slope Nil} = 0$
 $\text{slope (Node tl x h tr)} = \text{height tl} - \text{height tr}$

So, the new structure is as follows data AVL tree a equals nil or node AVL tree a with stands for the left subtree, a with stands for the value at the node, Int with stands for the height, AVL tree a, which stands for the right subtree. Now, the height function is a constant time operation, which is define as follows, height nil equals 0, height node t l x h t r equals h.

We also need a measure of how is skewed a tree is, namely it is slope and it define it as follows, slope of nil is 0, slope of node t l x h t r is nothing but, height of t l minus height of t r. We use this to determine whether a tree is still balanced or not, if slope is greater than or equal to 2, then r greater than or less than or equal to minus 2, then we know that the tree has an imbalance.

(Refer Slide Time: 17:53)



AVL trees: rotates

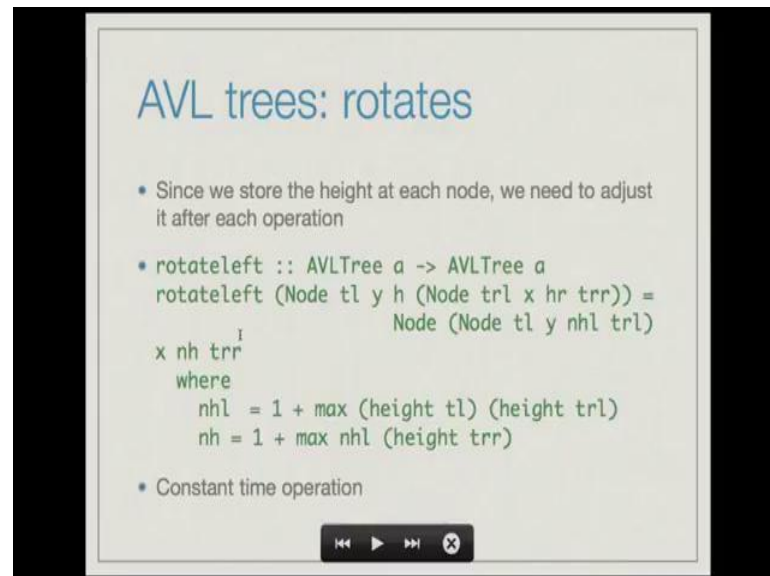
- Since we store the height at each node, we need to adjust it after each operation
- `rotateright :: AVLTree a -> AVLTree a`
`rotateright (Node (Node tll y hl tlr) x h tr) =`
`Node tll y nh (Node tlr x`
`nhtr tr)`
`where`
`nhtr = 1 + max (height tlr) (height tr)`
`nh = 1 + max (height tll) nhtr`
- Constant time operation

Let us now give implementations of rotate right and rotate left with the new data declaration. The wrinkle is that, since we store the height at each node, every time we touch the tree, we need to adjust the height. So, that definition of rotate right, rotate right of node t l l, y, h l, t l r, t l l is the subtree of the left subtree, t l r is the right subtree of the left subtree, h l is the height of the left subtree, x which is the root, h which is the height of the tree itself and t r, which is the right subtree of the original tree, equals node t l l y n h node t l r, x, n h r, t r.

This definition is exactly the same as the definition earlier except that we need to recomputed the heights, n h is the new height of the overall tree. And n h r is the new

height of the right subtree, nh_r clearly is 1 plus max of height tl_r and height tr_r , nh is 1 plus max of height tl_l and nh_r , because if the overall tree nh_r is the height of the right subtree and n and height of tl_l is that the left subtree. Clearly, this is the constant time operation, because we modify one pattern to another pattern and in computing the new heights, we are just comparing a constant number of heights to each other.

(Refer Slide Time: 19:38)

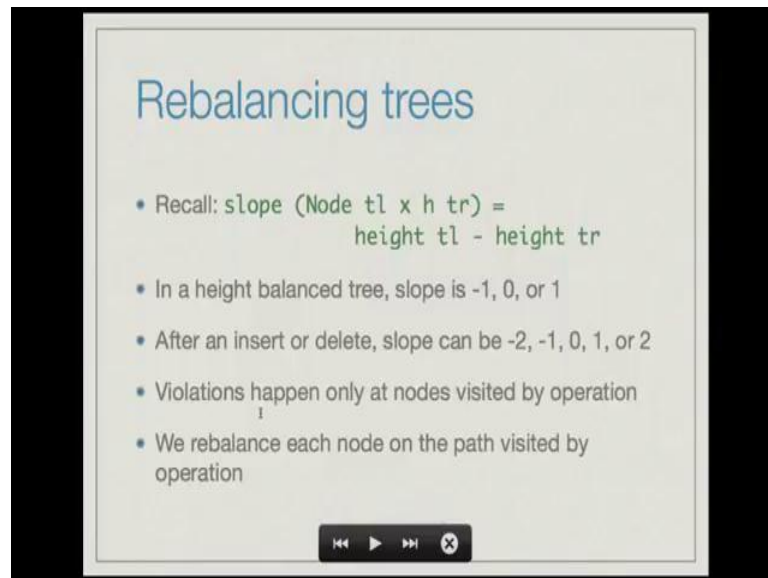


AVL trees: rotates

- Since we store the height at each node, we need to adjust it after each operation
- `rotateleft :: AVLTree a -> AVLTree a`
`rotateleft (Node tl y h (Node trl x hr trr)) =`
`Node (Node tl y nhl trl)`
`x nh trr`
`where`
`nhl = 1 + max (height tl) (height trl)`
`nh = 1 + max nhl (height trr)`
- Constant time operation

Here is the symmetric definition for rotate left; we just repeat the earlier definition with the appropriate heights, inserted. And in the result, we have the new heights nh_l and nh , we are again nh_l is defined as 1 plus max height of tl height of tr_l , nh is defined as 1 plus max of nh_l and height of tr_r . This is again the constant time operation.

(Refer Slide Time: 20:12)



Rebalancing trees

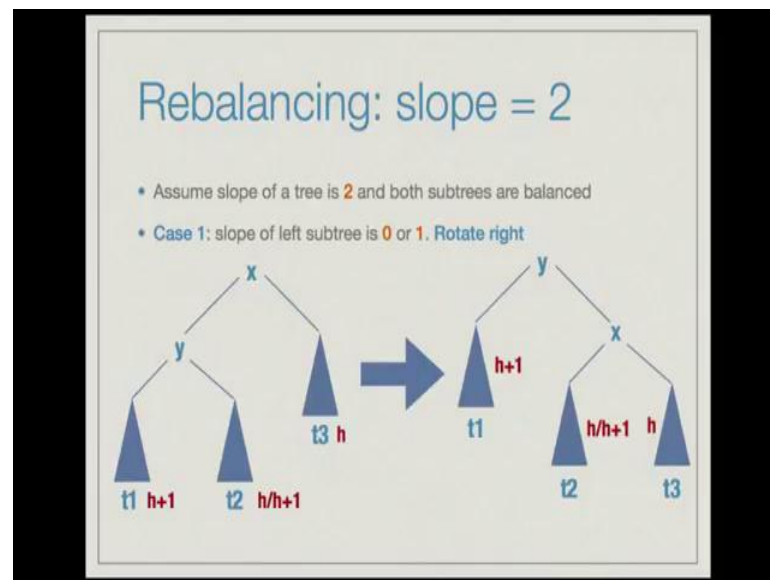
- Recall: $\text{slope}(\text{Node } tl \times h \ tr) = \text{height } tl - \text{height } tr$
- In a height balanced tree, slope is -1, 0, or 1
- After an insert or delete, slope can be -2, -1, 0, 1, or 2
- Violations happen only at nodes visited by operation
- We rebalance each node on the path visited by operation

Navigation controls: back, play, forward, close

Having looked at the rotates, let us now consider how to rebalance trees, this is the most crucial function in the implementation of AVL trees. Recall that the slope of a tree here is the height of the left subtree minus height of the right subtree. In a height balanced tree, the slope is minus 1 or 0 or 1, but after an insert or delete, it can happen that the slope is minus 2 or plus 2. A slope of minus 2 or plus 2 constitutes a violation, it cannot be less than minus 2 or greater than plus 2, because we are just inserting one node or deleting one node.

Now, the violations can happen only at nodes that are visited by an operation, recall that to insert or delete, we need to start at the root and search for the node that we need to insert or delete and in doing so, we will traverse in the path from root to that node. Only those nodes along the path will be affected. So, what we need to do is to rebalance each node on the path visited by the operation.

(Refer Slide Time: 21:31)



There are many cases to consider, let us first consider the case where the slope is plus 2, which means, that the height of the left subtree is exactly two more than the height of the right subtree. The case where slope is minus 2, this is symmetrical to this and we will not be elaborating that, so let us assume that, the slope of a tree is plus 2 and both subtrees are balanced.

Here, two cases might arise, let us look at the first case, the first case is when slope of left subtree is 0 or 1. Here, is the scenario, we have a root x , whose left child is y , which is the root of a tree consisting of y , t_1 and t_2 . Now, we said that this slope of the overall tree is 2, therefore the height of the tree root at y is h plus 2 and height of t_3 is h . If the height of the tree rooted at y is h plus 2, it means that one of the children is of height h plus 1 or the other child is of height h or h plus 1.

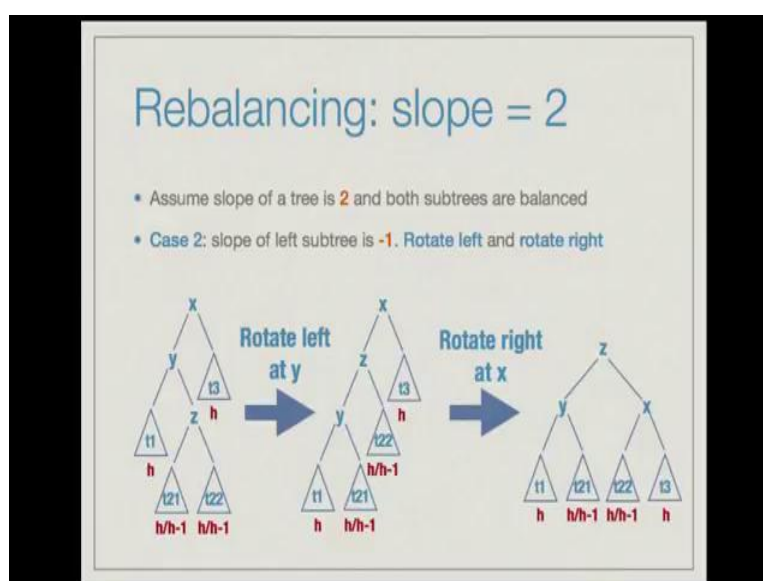
Since, we said that the slope of the left subtree is 0 or 1, therefore it has to be the case that the height of t_1 is at least as much as the height of t_2 . If the height of t_1 is smaller than the height of t_2 , then the slope of the tree rooted at y could actually be minus 1. Since, we said the slope of the left tree is 0 or 1, it has to be the case that t_1 is of height h plus 1 and t_2 is of height either h or h plus 1, this is the scenario.

In which case, all we need to do is rotate right, this brings y up to the root, it brings h down to be the right child of y and it shifts t_2 from being the right child of y to the left child of x , everything else is unchanged. Let us look at the slopes of the various subtrees

before and after the operation, earlier the slope of x was 2 and slope of the left subtree was 0 or 1.

Now, the slope of the new tree is height of left subtree, which is h plus 1 minus the height of the right subtree, which is either 1 plus h or 1 plus h plus 1. So, the height of the right subtree is either h plus 1 or h plus 2. So, in doing this rotate right, we have manage to bring this slope from 2 either 0 or minus 1. This is for the root node, for the right subtree, the slope is just plus 1. So, the new tree that we get is height balanced.

(Refer Slide Time: 24:49)



Here is the next case, this is the case when the slope of the tree is 2 and both subtrees are balanced, but the slope of the left subtree is minus 1. So, the tree looks as follows, we have the root x , whose left child is y , which has subtrees t_1 and t_2 , but expanded t_2 into it is root z and the two subtrees, t_{21} and t_{22} , the right subtree of x is t_3 . Since, this slope of the tree is 2, the height of the left subtree rooted at y is h plus 2 and the height of t_3 is h .

But, the slope of the left subtree is minus 1, which means that the height of t_1 is h and the height of the subtree rooted at z is h plus 1. For the height of the subtree rooted at z to be h plus 1, it has to be the case that either t_{21} or t_{22} is of right h and the other is of height h minus 1. So, this is the scenario, now we achieve a height balance tree in two steps, in the first step, we rotate left at y , so what happens is that, we just concentrate on the left subtree of the original tree.

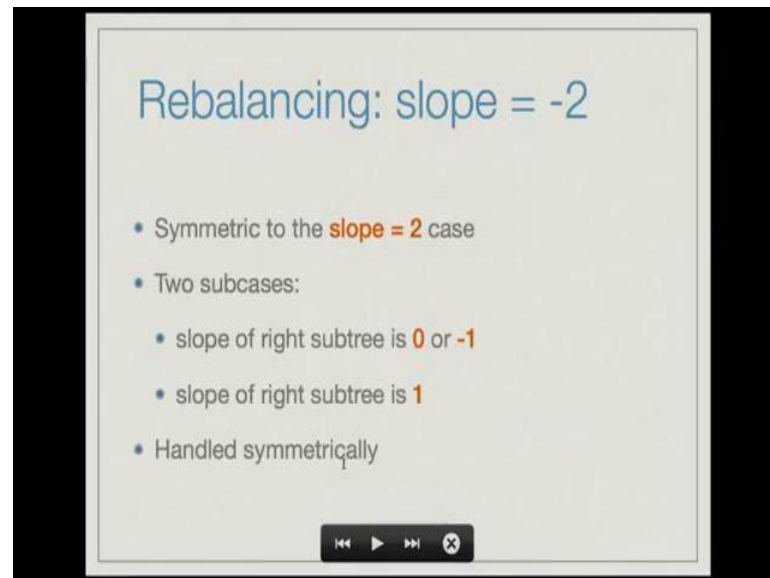
This portion and we rotate left at y, which means that, we pull z up, we push y down that is what we have done. But, when you pull z up and push y down, you make y the left child of z, which means that t_{21} means that place to go and it goes as the right subtree of y, which is what happens here. You have t_1 choose the original left subtree of y, still being the left subtree of y, but you have t_{21} , which will originally to the left of z, now to the right of y, z has moved up and become the parent of y and t_{22} is as before the right subtree of the tree rooted at z, we only change to this portion.

Now, what we have what we have achieved is to balance the subtree rooted at y. So, consider this subtree rooted at y, the slope is either 1 or 0, but the subtree rooted at z might have an imbalance and we in balance will occur in case t_{22} is of height h minus 1. Because, the left subtree rooted at y, now has height h plus 1, because t_1 is of height h, but the right subtree has height only if the right subtree has height only h minus 1, then there will be an in balance of z.

And this in balance would be caused by it is slope being 2 to fix this situation would was simple trick which is to do a rotate right at x. If you do a rotate right at x, what happens is that x goes down as the right child of z, z becomes the new root, t_{22} becomes the left child of x, this is the situation. We have z as the root y as the left child of z and t_1 and t_{21} being the subtrees of the tree rooted at y has before as here. But, we would done a rotate right at x, so z is the root, x is the right child and t_{22} and t_3 are children or subtrees of the tree rooted at x.

Now, let us compute the various heights height of t_1 has not changed it height of t_{21} is either h or h minus 1 height of t_{22} is either h or h minus 1 t_3 is of height h. now, height of the subtree rooted at y is h plus 1 and height of the subtree rooted at x is also h plus 1. So, we have got at tree whose slope is 0 and balance has been restored.

(Refer Slide Time: 29:07)

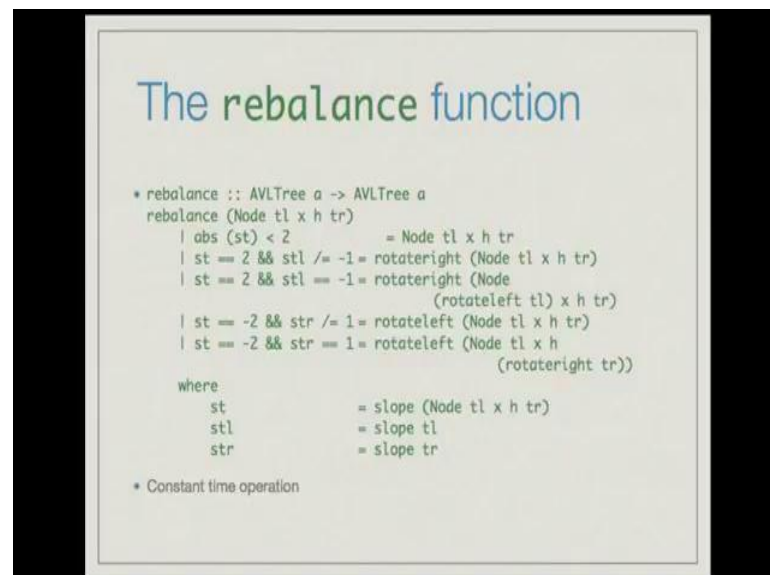


Rebalancing: slope = -2

- Symmetric to the slope = 2 case
- Two subcases:
 - slope of right subtree is 0 or -1
 - slope of right subtree is 1
- Handled symmetrically

Rebalancing for the case when slope equals minus 2 is symmetric to the slope equals two cases. We have two sub cases, one where the slope of the right subtree of the root is either 0 or minus 1 and 1, where the slope of the right subtree is 1 and the cases are handle symmetrically.

(Refer Slide Time: 29:35)



The rebalance function

```
* rebalance :: AVLTree a -> AVLTree a
rebalance (Node tl x h tr)
  | abs (st) < 2 = Node tl x h tr
  | st == 2 && stl /= -1 = rotateright (Node tl x h tr)
  | st == 2 && stl == -1 = rotateright (Node
                                     (rotateleft tl) x h tr)
  | st == -2 && str /= 1 = rotateleft (Node tl x h tr)
  | st == -2 && str == 1 = rotateleft (Node tl x h
                                     (rotateright tr))
  where
    st = slope (Node tl x h tr)
    stl = slope tl
    str = slope tr
```

• Constant time operation

So, here is a Haskell implementation of the rebalance function, rebalance is a function from which takes an input AVL tree a and produces an output of type AVL tree a. Rebalance of node t l, x, h, t r is in the case, where the absolute value of x t is less than 2,

s_t is the slope of the tree, slope of node t_l , x , h t_r . Just for references t_l is the slope of the left subtree, t_l , s_{t_r} is the slope of the right subtree t_r .

So, in the case where the absolute value of s_t is less than 2 which means that s_t is either minus 1 or 0 or 1, then you just written that tree as it is. In case when s_t is equal to 2 and s_{t_l} is not equal to minus 1, which means that s_{t_l} is either 0 or 1, this is the first subcase we looked at, then you just do a rotate right at the root x . In case the slope is 2 and the slope of the left subtree is minus 1, then you first do a rotate of the left subtree and then, you do a rotate right of the overall tree.

And the cases for the slope being minus 2 are symmetric, if the slope of the tree is minus 2 and slope of the right subtree is not equal to 1, then you do a rotate left at the root. If the slope is minus 2 and the slope of the right subtree is plus 1, then you first do a rotate right of the right subtree and then, you do a rotate left at the root. It is left is the exercise for the reader to work out the symmetric cases and ensure themselves of the correctness.

Notice that, rebalance is the constant time operation, because we just do either a rotate right or a rotate left or a rotate right and a rotate left. Recall that, rotate left and rotate right at themselves constant time operations and apart from this, we just compute the slope, which is again constant time operation.

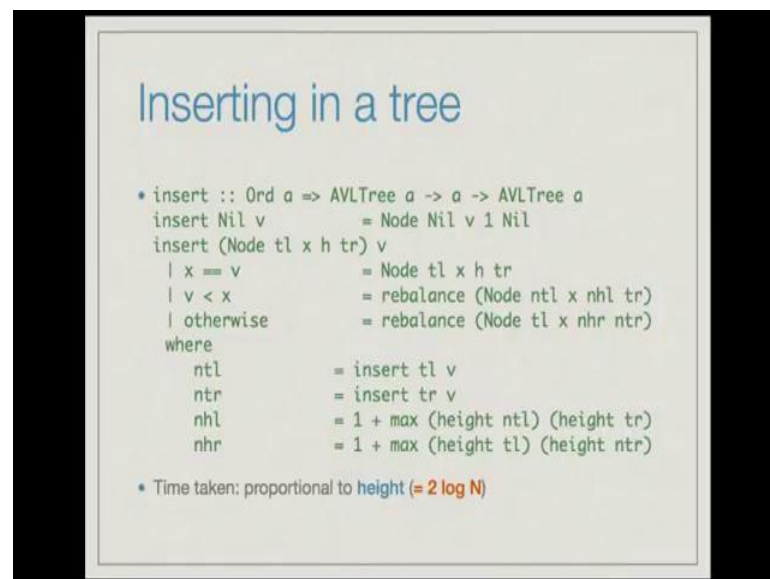
(Refer Slide Time: 31:56)

Searching in a tree

- `search :: Ord a => AVLTree a -> a -> Bool`
`search Nil v = False`
`search (Node t_l x h t_r) v`
 `| x == v = True`
 `| v < x = search t_l`
 `| otherwise = search t_r v`
- Time taken: proportional to height (= $2 \log N$)

Now, let us consider the key functions on a AVL tree, namely search, insert and delete. Search is as usual, search of nil v is false, search of node t l, x, h t r; we are searching for a value v in a tree given by this pattern. If x is the same as v, then you written true, if v is smaller than x, then you have to decent down the left subtree. So, you do a search t l, search t l v, otherwise you do a search t r v and the time taken this proportional to the height of the tree. But, now we have proved that in the height balanced tree, the height is twice $\log N$, where N is the size of the tree. So, therefore search case order $\log N$ time.

(Refer Slide Time: 32:46)



Inserting in a tree

```

• insert :: Ord a => AVLTree a -> a -> AVLTree a
insert Nil v      = Node Nil v 1 Nil
insert (Node tl x h tr) v
  | x == v        = Node tl x h tr
  | v < x         = rebalance (Node ntl x nhl tr)
  | otherwise     = rebalance (Node tl x nhr ntr)
  where
    ntl           = insert tl v
    ntr           = insert tr v
    nhl           = 1 + max (height ntl) (height tr)
    nhr           = 1 + max (height tl) (height ntr)

```

• Time taken: proportional to height (= $2 \log N$)

More important is insert and delete, here we see how to use rebalance, insert is a function which are the signature order a implies AVL tree a, arrow a, arrow AVL tree a. This is the input tree; this is the value to be inserted; that is the output tree. Insert v in the nil tree is nothing but, node nil v 1 nil, notice that, we enter the height of the tree, when we create the single note tree. Insert node t l x h t r v is as usual, if x is equal to v, then you do not need to do anything. So, you return the tree itself. If v is smaller than x, then we have to insert v in the left subtree.

So, we do a rebalance after we insert v into the left subtree. So, we do a rebalance of node n t l, x, n h l, t r; where n t l is insert t l v, n h l is the new height of the left subtree, which we calculate by 1 plus max of height n t l height t r. otherwise, this is the case when v is greater than x, in this case, we would a insert v in the right subtree. So, we

insert v in the right subtree and get $n\ h\ r$, $n\ t\ r$, $n\ t\ r$ is insert of $t\ r\ v$, $n\ h\ r$ is 1 plus max height $t\ l$ height $n\ t\ r$ and then, we do a rebalance.

Again, the time taken is proportional to height and we do a rebalance at each node; that is touched on the way to insert, but rebalance is the constant time operation and height is proportional to $\log N$. So, therefore, insert again is an operation that takes time proportional to $\log N$.

(Refer Slide Time: 35:07)

Deleting from a tree

```

* delete :: Ord a => AVLTree a -> a -> AVLTree a
delete Nil v      = Nil
delete (Node tl x h tr) v
  | v < x         = rebalance (Node ntl x nhl tr)
  | v > x         = rebalance (Node tl x nhr ntr)
  | otherwise     = if (tl == Nil) then tr else
                    rebalance (Node ty y hyr tr)

where
  (y, ty)         = deletemax tl
  ntl              = delete tl v
  ntr              = delete tr v
  nhl              = 1 + max (height ntl) (height tr)
  nhr              = 1 + max (height tl) (height ntr)
  hyr              = 1 + max (height ty) (height tr)

* Time taken: proportional to height (= 2 log N), assuming deletemax behaves well

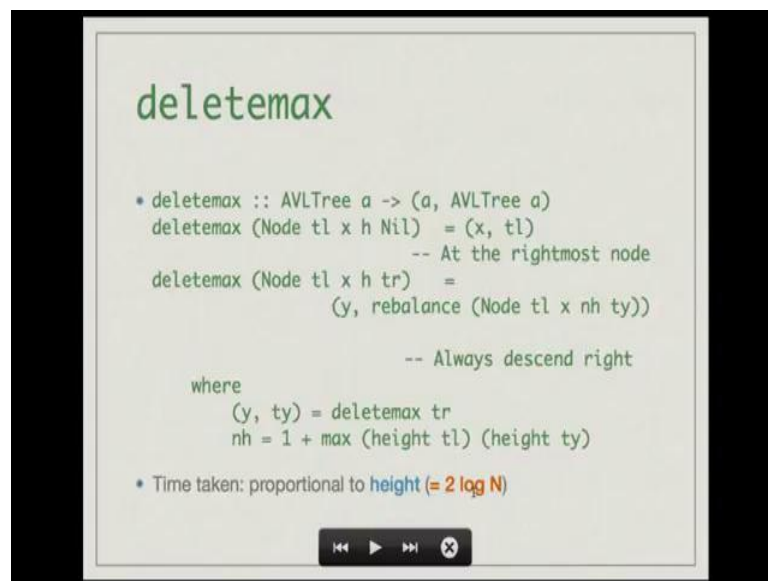
```

Here is delete, delete nil v is just nil, delete v from a tree of the form node $t\ l, x, h, t\ r$, again we consider three cases, if v is smaller than x , then we delete v from the left subtree and get $n\ t\ l$, $n\ t\ l$ is delete $t\ l\ v$, then new height of the left subtree is $n\ h\ l$. This is computed as usual 1 plus max height $n\ t\ l$ height $t\ r$ and we take this tree under rebalance it. And the case where v is greater than x , we delete v from the right subtree and we rebalance the overall tree.

To crucial case is, when v is equal to x in which case, we do deletion as in the binary search tree case, we first check if the left subtree is nil, if that is the case, then we just replace the tree node $t\ l\ x\ h\ t\ r$ by just $t\ r$. Because, we are deleting x and there is no left subtree of x , so $t\ l$ is nil and $t\ r$ is the right subtree of x , $t\ r$ can take the place of x , so this is what we written.

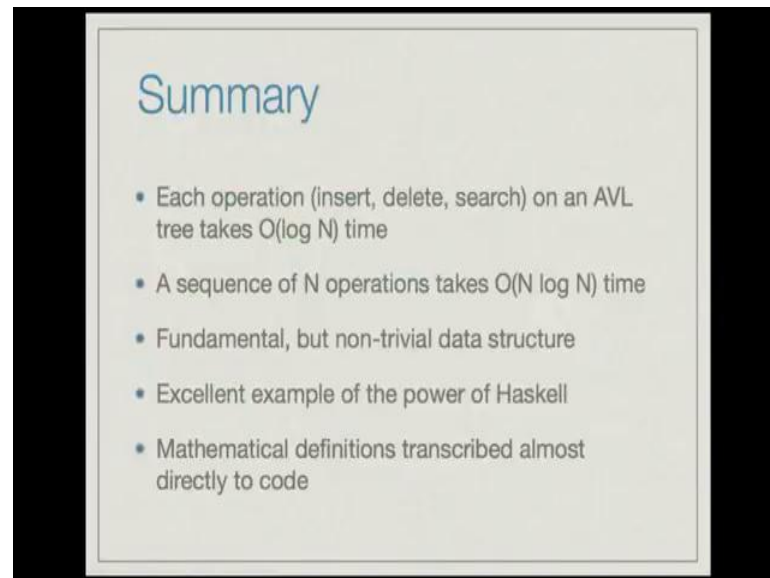
Otherwise, we consider the maximum element of the left subtree, which is y , which is gotten by delete max of t_l , this delete max is exactly as we saw earlier and t_y is the left subtree after the maximum element has being deleted. And we just replace the node x by y here and t_l is replaced by t_y and h_y is the new height and then rebalance this tree. Again, the time taken by this operation is proportional to height which is $2 \log N$ assuming delete max behaves well.

(Refer Slide Time: 37:14)



But, delete max does not behave well and the implementation is just like earlier. Modulo adjusting the heights, again time taken is proportional to twice $\log N$.

(Refer Slide Time: 37:38)



The slide is titled "Summary" in a blue font. It contains five bullet points, each preceded by a blue square marker. The background of the slide is light gray, and it is framed by a dark border.

- Each operation (insert, delete, search) on an AVL tree takes $O(\log N)$ time
- A sequence of N operations takes $O(N \log N)$ time
- Fundamental, but non-trivial data structure
- Excellent example of the power of Haskell
- Mathematical definitions transcribed almost directly to code

In summary, we have defined AVL trees with supports a set data structure, supporting the operations insert, delete and search, each of which take order $\log N$ time, where N is the size of the set. Therefore, a sequence of n operations takes order $N \log N$ time, this is the fundamental, but non-trivial data structure and is an excellent example of the power of Haskell. As you may have seen during the course of this extended example, the mathematical definitions pertaining to height balance trees could be transcribed almost directly to Haskell code.