Introduction To Haskell Programming

Prof. S. P. Suresh

Chennai Mathematical Institute

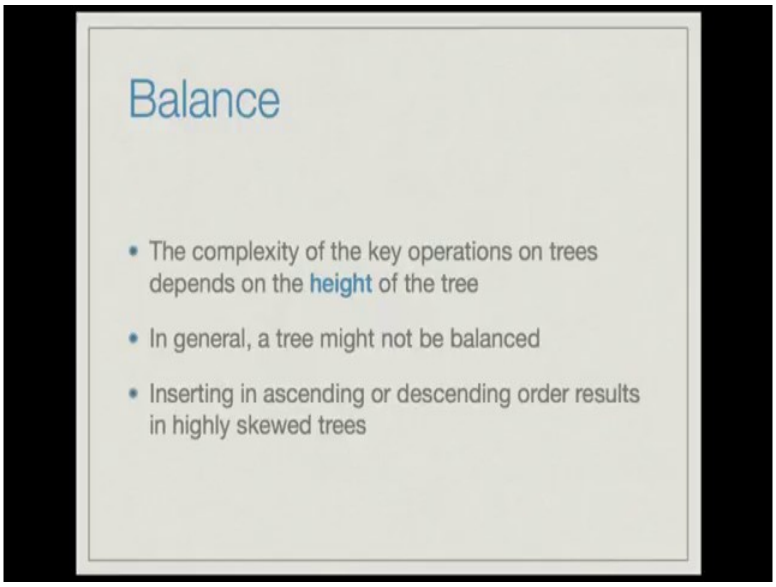
Module # 06

Lecture – 03

Balanced Search Tree

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

(Refer Slide Time: 00:11)



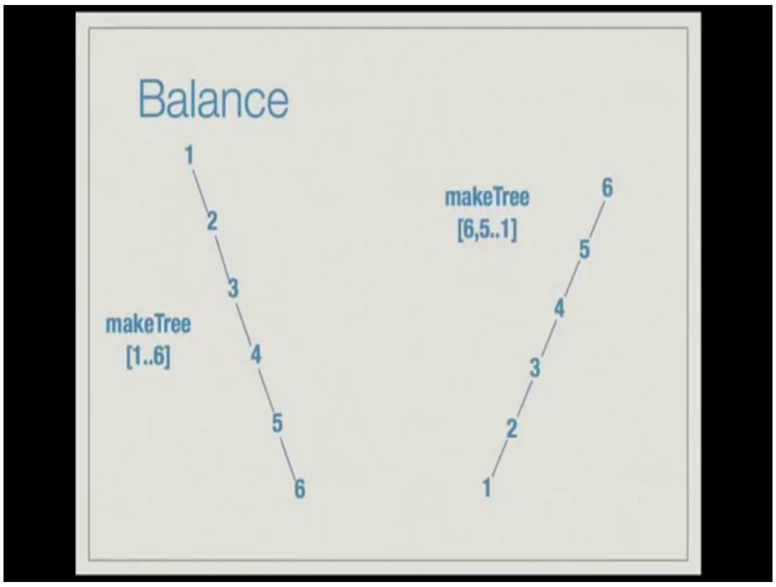
Balance

- The complexity of the key operations on trees depends on the **height** of the tree
- In general, a tree might not be balanced
- Inserting in ascending or descending order results in highly skewed trees

Recall that in the previous lecture, we studied implementations of the set data type, which maintains a 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 the other, which was to store the elements as a binary search tree. The list implementation was considered inefficient, because each search takes time proportional to the length of the list, in other words the number of elements in the set.

So, N operations can take up to O(N2) time. On the other hand, the binary search tree data structure supports insert, search and delete in time proportional 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)

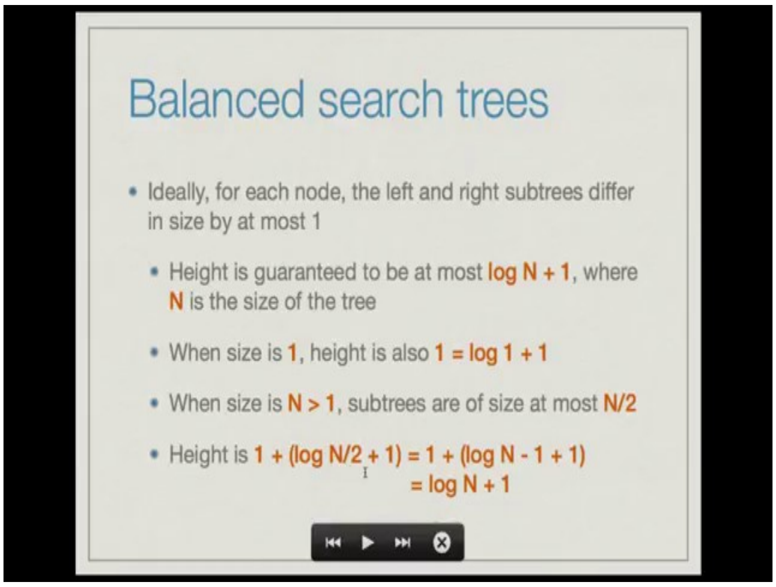

Balance

Here are two examples, if we consider makeTree of 1 to 6, recall that makeTree was a function that starts with the Nil tree and inserts all these elements in the list one after the other. So, makeTree of [1..6] will create a tree that looks like this; it is skewed to the right. makeTree of [6,5..1] inserts elements in descending order into the tree and this tree is skewed to the left. It will start with 6, the next element to be inserted is 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 a binary search tree might take up to O(N) for search, insert and delete. So, a sequence of n operations might still take up to O(N2) time.

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)



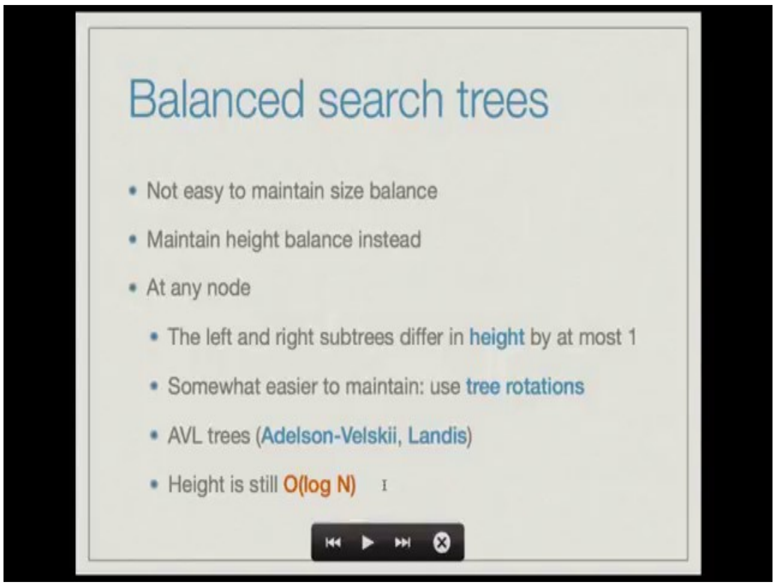
Balanced search trees

- 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

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 + 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 + 1. When the size of the tree is N greater than 1, then observe that both subtrees are of size at most N/2, because if each subtree is of size greater than N/ 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/2 + 1, which is the same as log N - 1 + 1, which is the same as log N. And therefore, the height of the tree itself which is 1 + (maximum of the height of the two subtrees) is 1 + log N.

(Refer Slide Time: 04:05)



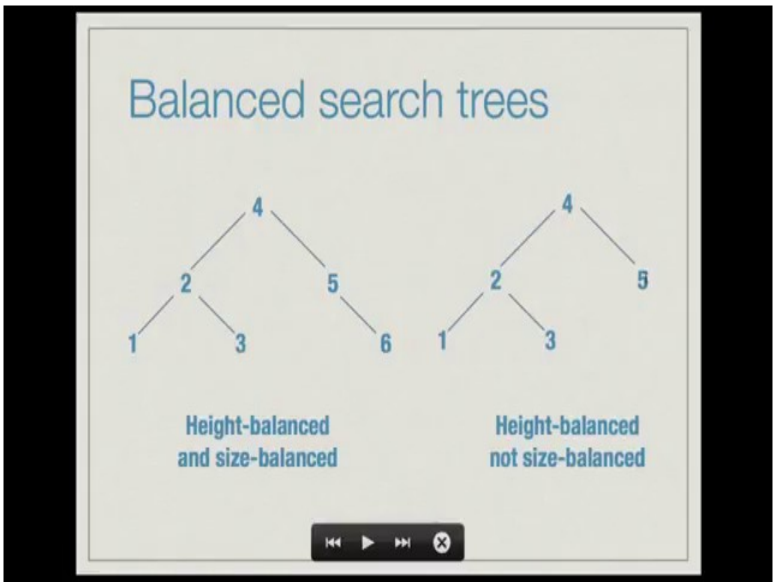
Balanced search trees

- Not easy to maintain size balance
- Maintain height balance instead
- At any node
  - The left and right subtrees differ in height by at most 1
  - Somewhat easier to maintain: use tree rotations
  - AVL trees (Adelson-Velskii, Landis)
  - Height is still O(log N)

But, the trouble is that it is not easy to maintain size balance, what we shall ought 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 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 this 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)



Balanced search trees

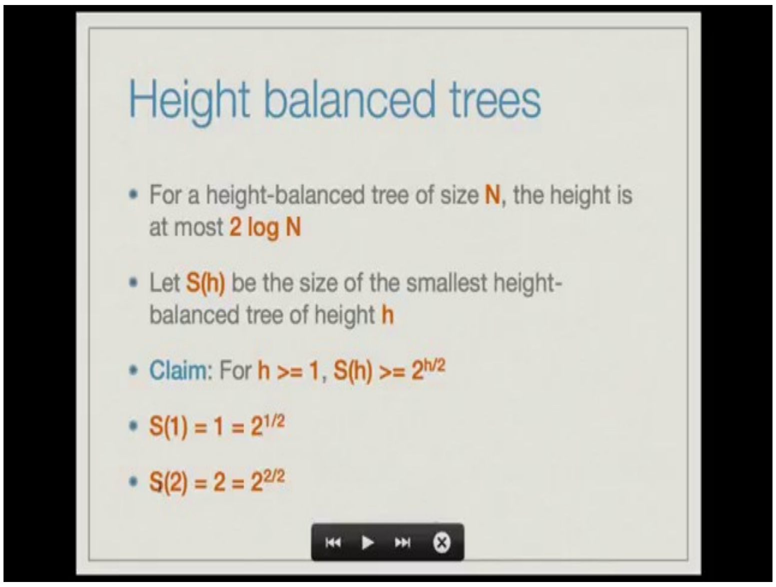
Here are some examples, on the left, you see a tree that is both height balanced and size balanced. We can check it as follows, the tree is of size 6, there are three nodes in the left subtree; that is the root node here and there are two nodes in the right subtree and you see that the difference in 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 they 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, with 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 of height 2, and the right subtree is of height 2. If you 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)



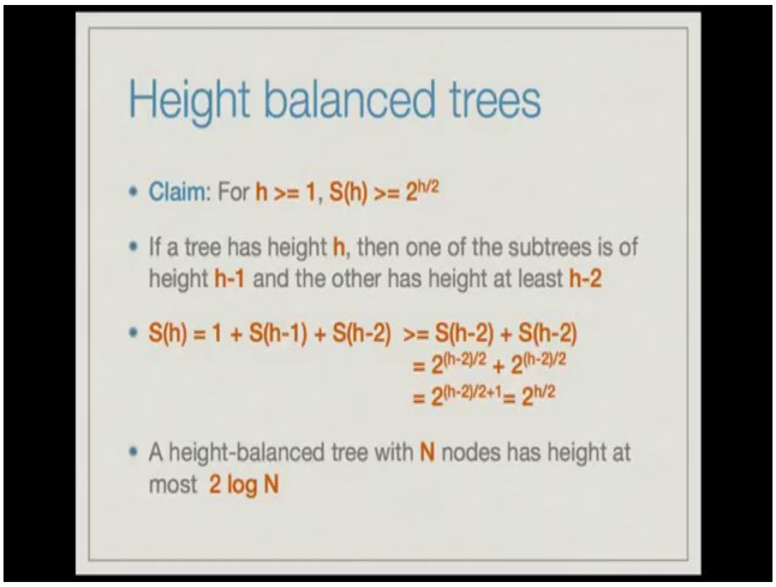
Height balanced trees

- 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 >= 1**, **S(h) >=** $2^{h/2}$
- **S(1) = 1 =** $2^{1/2}$
- **S(2) = 2 =** $2^{2/2}$

Before, we move on to the implementation of a height balanced tree, let us 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 S(h) be the size of the smallest height balanced tree of height h, we claim that for h >=1, S(h) >= 2h/2, S(1) = 1 = 21/2.

Here, we take 1/ 2 to the 0, S(2) equals 2, which is 22/2, S(2) = 2, because the smallest height balanced 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)



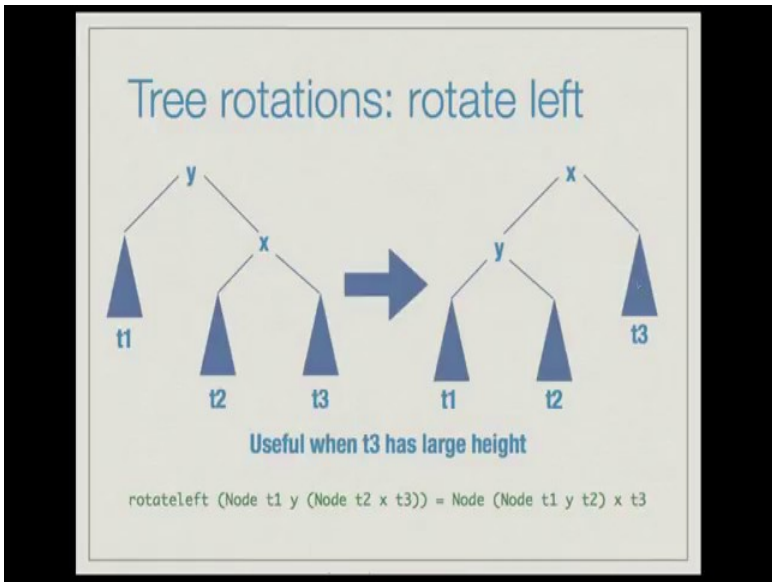
Height balanced trees

- Claim: For h >= 1, S(h) >= $2^{h/2}$
- 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**
- $S(h) = 1 + S(h-1) + S(h-2) >= S(h-2) + S(h-2)$
  $= 2^{(h-2)/2} + 2^{(h-2)/2}$
  $= 2^{(h-2)/2+1} = 2^{h/2}$
- 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-1, and the other has height at least h-2. Now, the height of a tree is defined to be 1 + max of the height of the two subtrees. So, one of the subtrees definitely has to have height exactly h - 1 and the other subtree cannot have height smaller than h-2, because that would violate the height balance property.

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

(Refer Slide Time: 10:06)



Tree rotations: rotate left

Useful when t3 has large height

```
rotateleft (Node t1 y (Node t2 x t3)) = Node (Node t1 y t2) x t3
```

Having established the mathematical bounds, let us look at how to implement a height balanced tree. If you recall, we mentioned 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 its own root y and subtrees t1 and t2, the right subtree of the bigger tree is t3.

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 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 t2 to the left child of x.

In doing this, we still maintain the binary search tree property, notice that, every node in t1 is smaller than y, t1 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, t2 appear.. every node in t2 appears to the right of y, so it is greater than y. But, on the other hand, it appears in the left subtree rooted at x, so it is smaller than x.

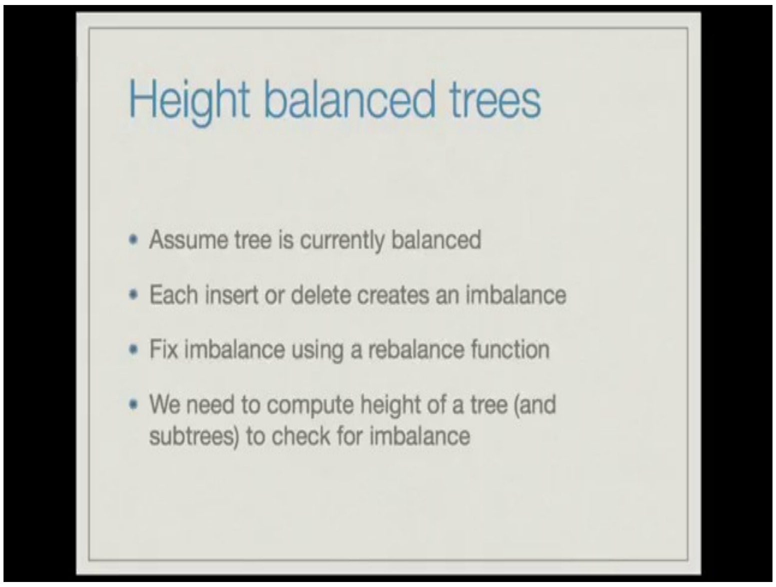
If you look at the new configuration t2 appears to the left of x, so every node in t2 is smaller than x, but t2 is part of the right subtree of y, therefore, every node in t2 is greater than y. So, the search tree property, if it was maintained earlier, it would also be maintained after the rotate. This operation is useful when t1 has large height, as you can see now earlier the subtree to the left of x had height 1 + height of t1.

Now, the left subtree of the root has only height t1, 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 t1 y t2) we just state this pattern here, x which is the root, t3 which is the right subtree equals we just write the pattern for this tree. Node t1, which is the left subtree, y which is the root and Node t2 x t3, 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 t2 and t3, the left child of the original tree is t1. Now, in case t3 has large height, we might want to push t3 up towards 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 t2 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 t2 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 when t3 has a large height and it is exactly the inverse of rotate right and the Haskell description is again very simple. Rotate left of this pattern, which is specified by Node t1 y (Node t2 x t3) equals node of left subtree which is (Node t1 y t2) and x t3, where x is the root and t3 is the right subtree.

(Refer Slide Time: 14:55)



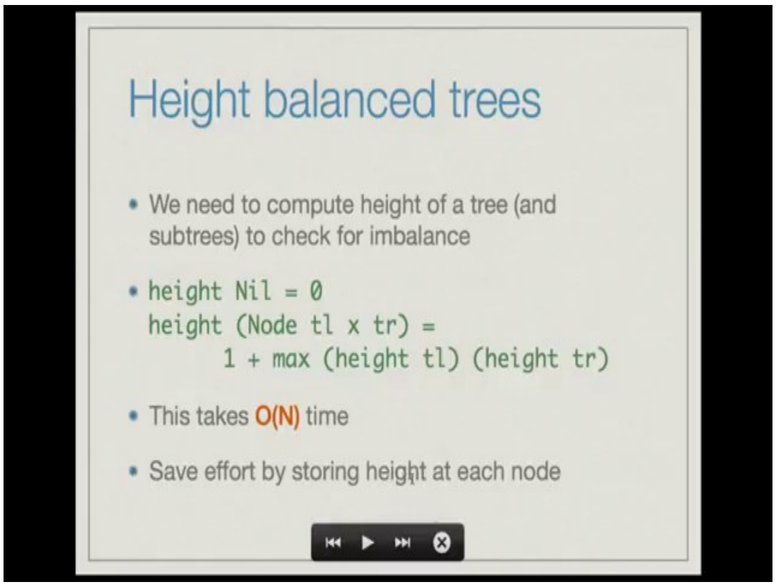
Height balanced trees

- Assume tree is currently balanced
- Each insert or delete creates an imbalance
- Fix imbalance using a rebalance function
- We need to compute height of a tree (and subtrees) to check for imbalance

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 rights 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)



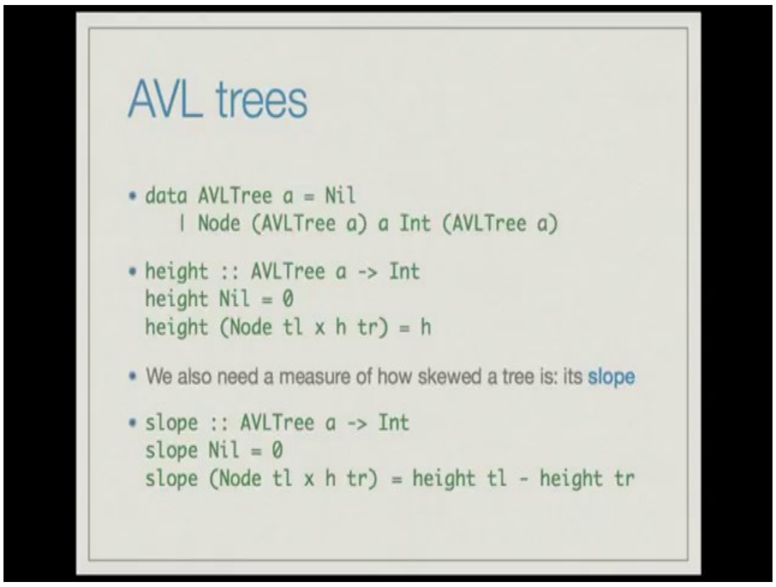
Height balanced trees

- We need to compute height of a tree (and subtrees) to check for imbalance

-
  ```
  height Nil = 0
  height (Node tl x tr) =
        1 + max (height tl) (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 imbalance, usually height would be defined as follows, height of Nil = 0, height (Node tl x tr) = 1 + max (height tl) (height tr). But, unfortunately this definition takes O(N) time to compute height and we would not like to spend O(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

-
  ```
  data AVLTree a = Nil
     | Node (AVLTree a) a Int (AVLTree a)
  ```

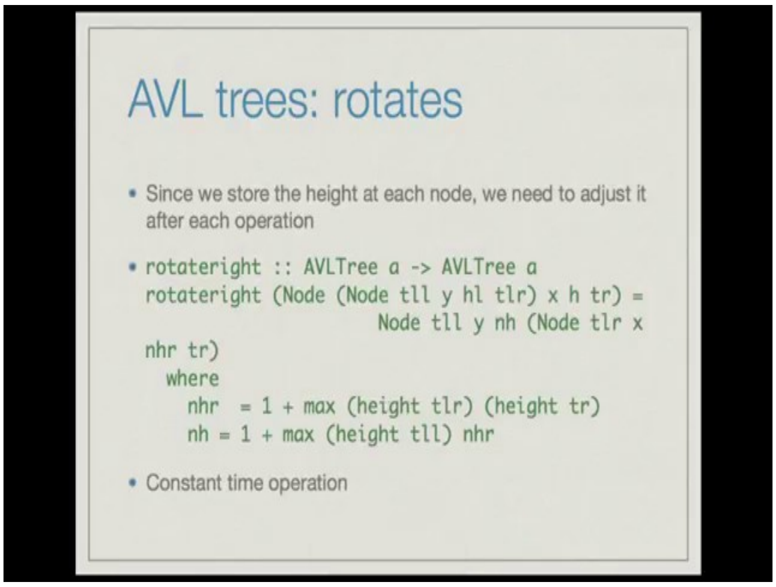
-
  ```
  height :: AVLTree a -> Int
  height Nil = 0
  height (Node tl x h tr) = h
  ```
- We also need a measure of how skewed a tree is: its slope

-
  ```
  slope :: AVLTree a -> Int
  slope Nil = 0
  slope (Node tl x h tr) = height tl - height tr
  ```

So, the new structure is as follows. data AVLTree a = Nil | Node (AVLTree a) a Int (AVLTree a) where the first (AVLTree a) stands for the left subtree, a which stands for the value at the node, Int which stands for the height, AVLTree a, which stands for the right subtree. Now, the height function is a constant time operation, which is defined as follows, height Nil = 0, height (Node tl x h tr) = h..

We also need a measure of how skewed a tree is, namely its slope and we define it as follows, slope of Nil is 0, slope of (Node tl x h tr) is nothing but, height of tl - height of tr. 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

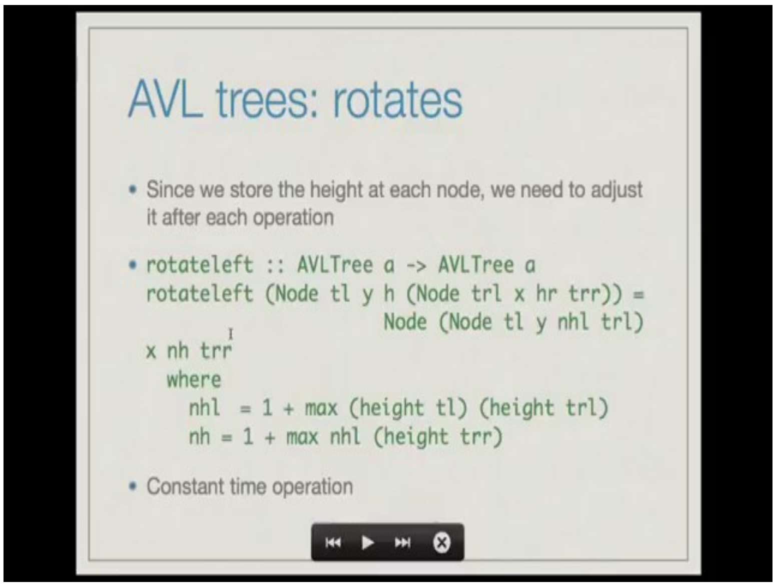
```
• rotateright :: AVLTree a -> AVLTree a
  rotateright (Node (Node tll y hl tlr) x h tr) =
                          Node tll y nh (Node tlr x
  nhr tr)
    where
      nhr  = 1 + max (height tlr) (height tr)
      nh = 1 + max (height tll) nhr
```

- 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, everytime we touch the tree, we need to adjust the height. So, this is the definition of rotate right: rotateright of (Node (Node tll y hl tlr) x h tr) = Node tll y nh (Node tlr x nhr tr) where tll is the left subtree of the left subtree, tlr is the right subtree of the left subtree, hl is the height of the left subtree, x which is the root, h which is the height of the tree itself and tr, which is the right subtree of the original tree, equals Node tll y nh (Node tlr x nhr tr) .

This definition is exactly the same as the definition earlier except that we need to recompute the heights, nh is the new height of the overall tree. And nhr is the new height of the right subtree, nhr clearly is 1+ max of (height tlr) and (height tr) , nh is 1+ max of (height tll) and nhr , because for the overall tree nhr is the height of the right subtree and height of tll is the height of 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 against each other.

(Refer Slide Time: 19:38)



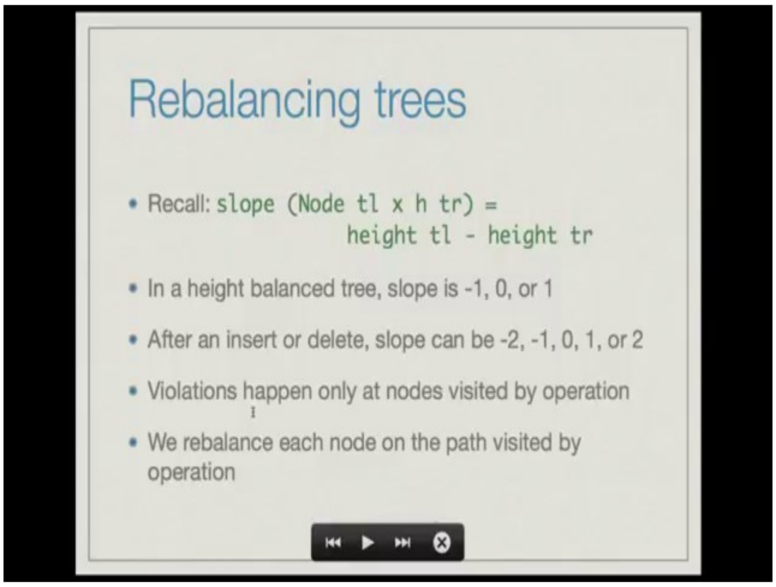
AVL trees: rotates

- 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 nhl and nh, where again nhl is defined as 1+ max of (height of tl) and (height of trl), nh is defined as 1 + max of nhl and (height of trr). This is again a constant time operation.

(Refer Slide Time: 20:12)



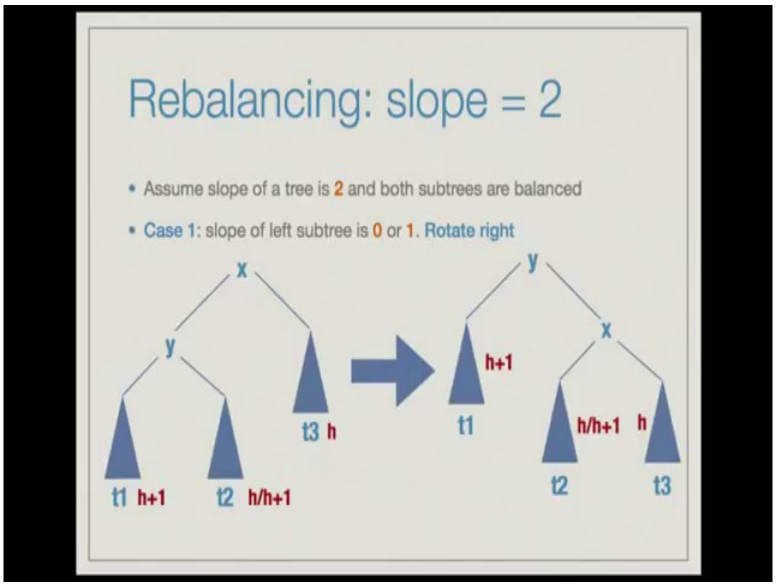
Rebalancing trees

- Recall: `slope (Node tl x h tr) = height tl - 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

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 slope of a tree here is the height of the left subtree minus height of the right subtree. In a height balanced tree slope is -1 or 0 or 1, but after an insert or delete, it can happen that the slope is -2 or +2. A slope of -2 or +2 constitutes a violation, it cannot be less than -2 or greater than +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 be traversing a 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)



Rebalancing: slope = 2

- Assume slope of a tree is 2 and both subtrees are balanced
- Case 1: slope of left subtree is 0 or 1. Rotate right

x
y
t1 h+1
t2 h/h+1
t3 h
y
h+1
t1
x
h/h+1
h
t2
t3

They are many cases to consider, let us first consider the case where the slope is +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 -2 is symmetrical to this and we will not be elaborating that, so let us assume that, the slope of a tree is +2 and both subtrees are balanced.

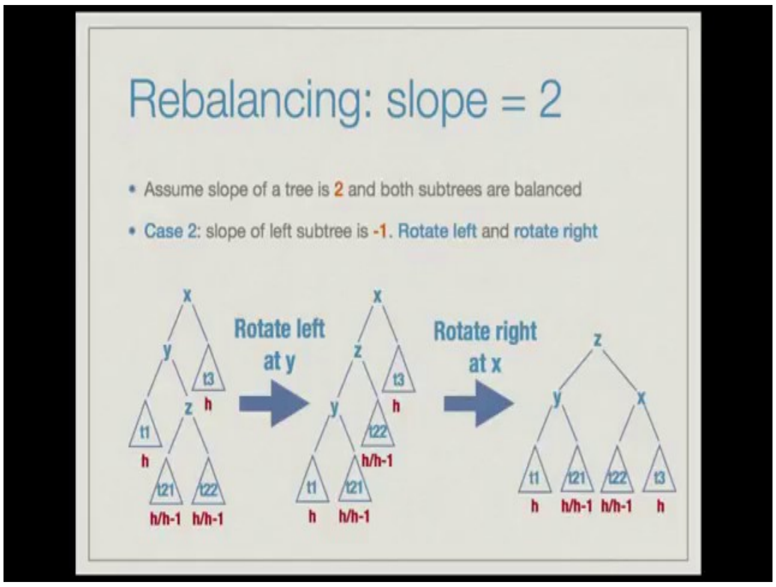
Here, two cases might arise, let us look at the first case, the first case is when slope of left subtree is 0 or 1. Here, is the scenario, we have a root x, whose left child is y, which is the root of a tree consisting of y, t1 and t2. Now, we said that this slope of the overall tree is 2, therefore the height of the tree rooted at y is h+2 and height of t3 is h. If the height of the tree rooted at y is h+2, it means that one of the children is of height h+1 and the other child is of height h or h+1.

Since, we say that the slope of the left subtree is 0 or 1, therefore it has to be the case that the height of t1 is at least as much as the height of t2. If the height of t1 were smaller than the height of t2, then the slope of the tree rooted at y would actually be -1. Since, we say that the slope of the left subtree is 0 or 1, it has to be the case that t1 is of height of h+1 and t2 is of height either h or h+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 t2 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 + 1 - height of the right subtree, which is either 1+ h or 1 + h + 1. So, the height of the right subtree is either h+1 or h+2. So, in doing this rotate right, we have managed to bring this slope from 2 to either 0 or -1. This is for the root node, for the right subtree, the slope is just +1. So, the new tree that we get is height balanced.

(Refer Slide Time: 24:49)



Rebalancing: slope = 2

- Assume slope of a tree is 2 and both subtrees are balanced
- Case 2: slope of left subtree is -1. Rotate left and rotate right

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 -1. So, the tree looks as follows, we have the root x, whose left child is y, which has subtrees t1 and t2, but expanded t2 into it is root z and the two subtrees, t21 and t22, the right subtree of x is t3. Since, this slope of the tree is 2, the height of the left subtree rooted at y is h + 2 and the height of t3 is h.

But, the slope of the left subtree is -1, which means that the height of t1 is h and the height of the subtree rooted at z is h+1. For the height of the subtree rooted at z to be h + 1, it has to be the case that either t21 or t22 is of height h and the other is of height h-1. So, this is the scenario, now we achieve a height balanced 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.

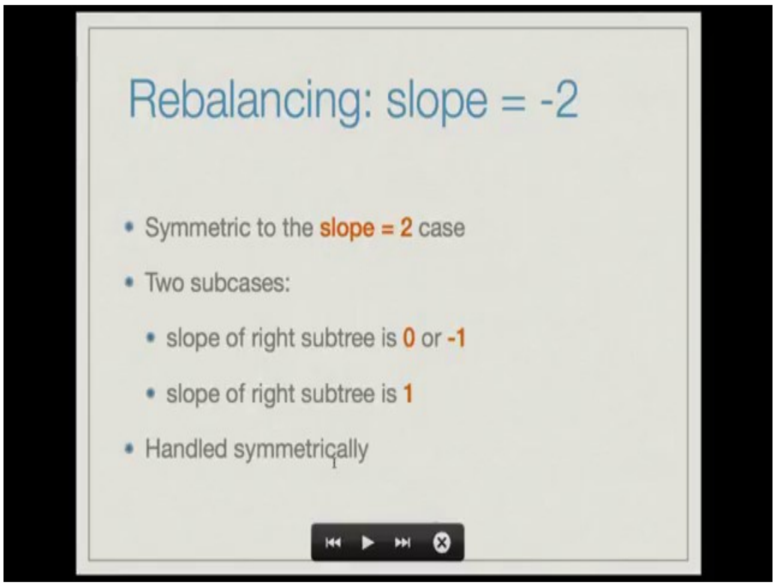
This portion and we rotate left at y, which means that, we pull z up, we push y down that is what we have done. But, when you pull z up and push y down, you make y the left child of z, which means that t21 needs a place to go and it goes as the right subtree of y, which is what happens here. You have t1 choose the original left subtree of y, still being the left subtree of y, but you have t21, 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 t22 is as before the right subtree of the tree rooted at z, we only changed this portion.

Now, what we have achieved is to balance the subtree rooted at y. So, if you consider this subtree rooted at y, the slope is either 1 or 0, but the subtree rooted at z might have an imbalance and the imbalance will occur in case t22 is of height h-1. Because, the left subtree rooted at y, now has height h+1, because t1 is of height h, but the right subtree has height only if the right subtree has height only h-1, then there will be an imbalance of z.

And this imbalance would be caused by its slope being 2. To fix this situation we do a 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, t22 becomes the left child of x, this is the situation. We have z as the root y, as the left child of z and t1 and t21 being the subtrees of the tree rooted at y as before as here. But, we have done a rotate right at x, so z is the root, x is the right child and t22 and t3 are children or subtrees of the tree rooted at x.

Now, let us compute the various heights. height of t1 has not changed yet. height of t21 is either h or h-1. height of t22 is either h or h-1, t3 is of height h. now, height of the subtree rooted at y is h+1 and height of the subtree rooted at x is also h+1. So, we have got a tree whose slope is 0 and balance has been restored.

(Refer Slide Time: 29:07)

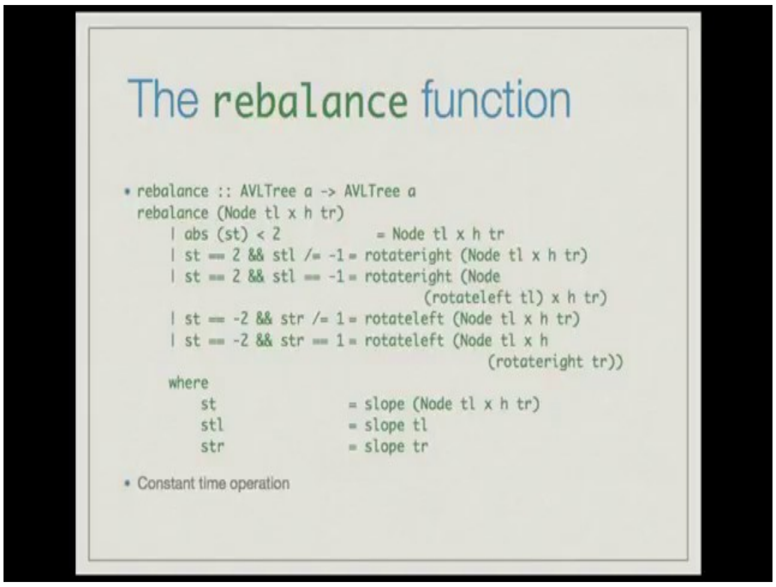


Rebalancing: slope = -2

- 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 -2 is symmetric to the slope equals 2 case. We have two sub cases, one where the slope of the right subtree of the root is either 0 or -1 and one where the slope of the right subtree is 1 and the cases are handled 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

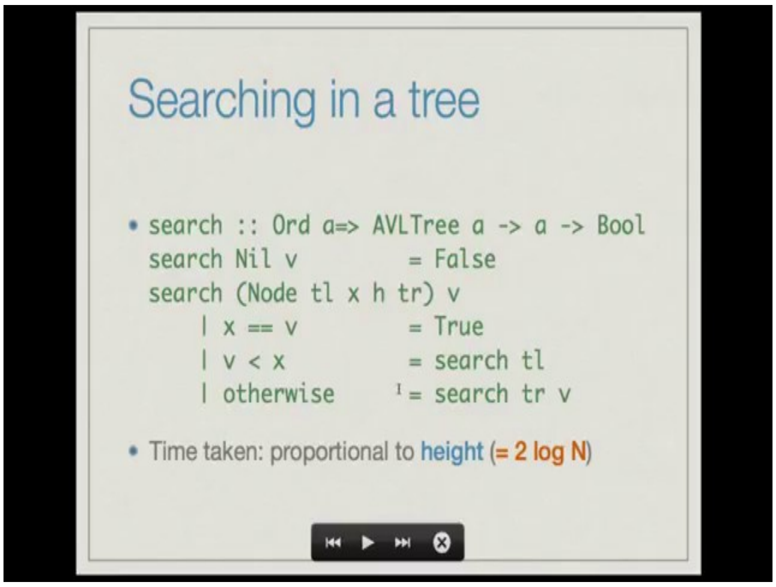
So, here is a Haskell implementation of the rebalance function, rebalance is a function from which takes an input AVLTree a and produces an output of type AVLTree a. Rebalance of (Node tl x h tr) is in the case, where the absolute value of st, obs(st), is less than 2, st is the slope of the tree, slope of Node tl x h tr. Just for references tl is the slope of the left subtree, tl, str is the slope of the right subtree tr.

So, in the case where the absolute value of st is less than 2 which means that st is either -1 or 0 or 1, then you just return the tree as it is. In case when st is equal to 2 and stl is not equal to -1, which means that stl 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 -1, then you first do a rotate left of the left subtree and then, you do a rotate right of the overall tree.

And the cases for the slope being -2 are symmetric, if the slope of the tree is -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 -2 and the slope of the right subtree is +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 as an exercise for the reader to work out the symmetric cases and ensure themselves of the correctness.

Notice that, rebalance is a 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 are themselves constant time operations and apart from this, we just compute the slope, which is again a constant time operation.

(Refer Slide Time: 31:56)



Searching in a tree

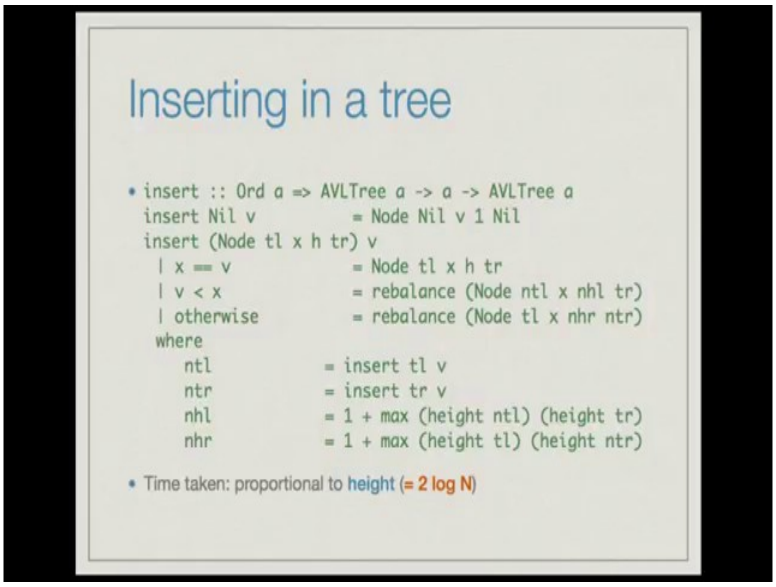
- search :: Ord a=> AVLTree a -> a -> Bool

```
search :: Ord a=> AVLTree a -> a -> Bool
search Nil v          = False
search (Node tl x h tr) v
    | x == v          = True
    | v < x           = search tl
    | otherwise       = search tr v
```

- Time taken: proportional to height (= 2 log N)

Now, let us consider the key functions on an AVL tree, namely search, insert and delete. Search is as usual, search of Nil v is false, search of (Node tl x h tr); we are searching for a value v in a tree given by this pattern. If x is the same as v, then you return true, if v is smaller than x, then you have to descend down the left subtree. So, you do a search tl, search tlv, otherwise you do a search trv and the time taken is proportional to the height of the tree. But, now we have proved that in a height balanced tree, the height is 2 log N, where N is the size of the tree. So, therefore search takes O (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)
```

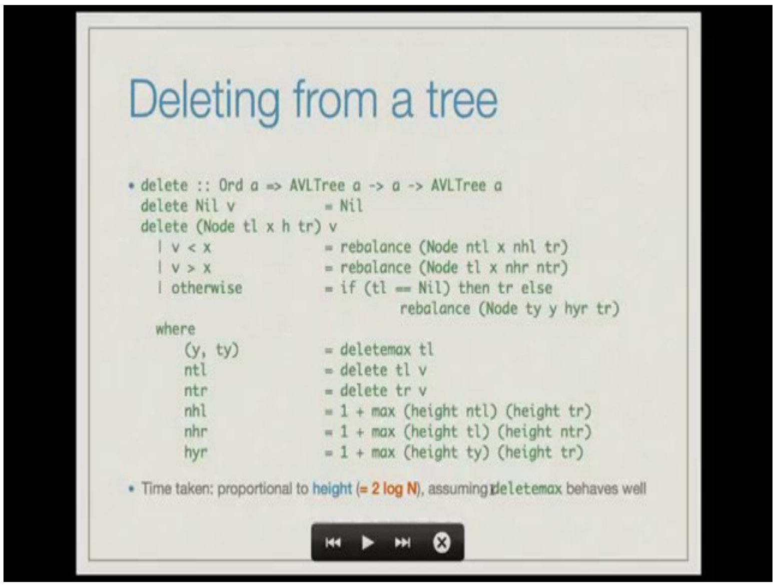
- Time taken: proportional to height (= 2 log N)

More important is insert and delete, here we see how to use rebalance, insert is a function which has the signature Ord a => AVLTree a -> a -> AVLTree a which are in the order of input tree, value to be inserted and 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 node tree. Insert (Node tl x h tr) 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 < 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 ntl x nhl tr) where ntl is insert tl v, nhl is the new height of the left subtree, which we calculate by 1 + max of (height ntl) and (height tr). otherwise, this is the case when v>x, in this case, we have to insert v in the right subtree. So, we insert v in the right subtree and get ntr. ntr is insert of tr v and nhr is 1 + max of (height tl) and (height ntr) 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 a 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

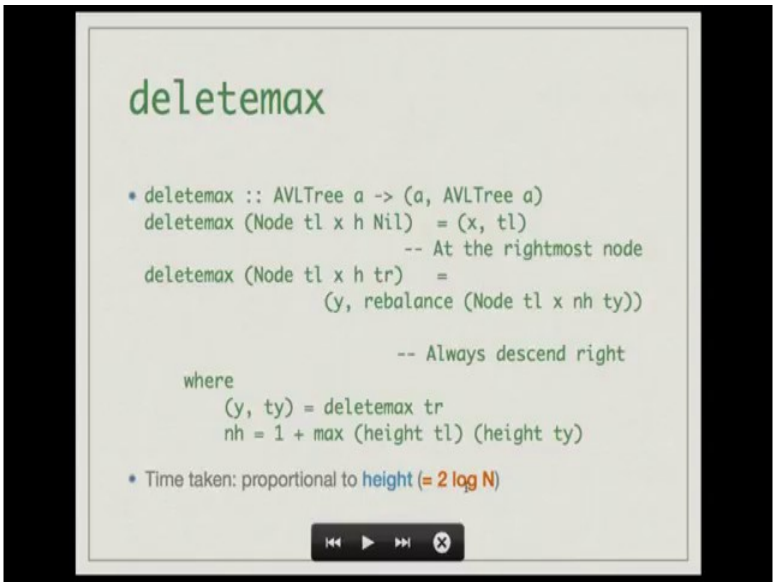
-
```
delete :: Ord a => AVLTree a -> a -> AVLTree a
delete Nil v              = Nil
delete (Node tl x h tr) v
  | v < x                 = rebalance (Node ntl x nhl tr)
  | v > x                 = rebalance (Node tl x nhr ntr)
  | otherwise             = if (tl == Nil) then tr else
                                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 tl x h tr), again we consider three cases, if v is smaller than x, then we delete v from the left subtree and get ntl, ntl is delete tl v, the new height of the left subtree is nhl. This is computed as usual 1 + max (height ntl) (height tr) and we take this tree and rebalance it. And the case where v > x, we delete v from the right subtree and we rebalance the overall tree.

The 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 tl x h tr) by just tr. Because, we are deleting x and there is no left subtree of x, so tl is Nil and tr is the right subtree of x, tr can take the place of x, so this is what we return.

Otherwise, we consider the maximum element of the left subtree, which is y, which is gotten by deletemax of tl, this deletemax is exactly as we saw earlier and ty is the left subtree after the maximum element has been deleted. And we just replace the node x by y here and tl is replaced by ty and hyr is the new height and we 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)



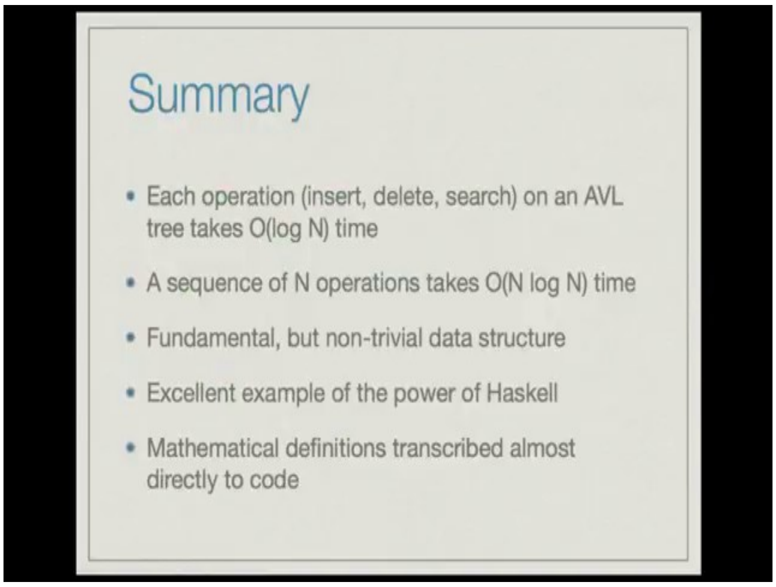
deletemax

-
  ```
  deletemax :: AVLTree a -> (a, AVLTree a)
  deletemax (Node tl x h Nil)  = (x, tl)
                              -- At the rightmost node
  deletemax (Node tl x h tr)   =
                    (y, rebalance (Node tl x nh ty))

                              -- Always descend right
      where
          (y, ty) = deletemax tr
          nh = 1 + max (height tl) (height ty)
  ```
- Time taken: proportional to height (= 2 log N)

But, delete max does indeed behave well and the implementation is just like earlier. Modulo adjusting the heights, again time taken is proportional to 2 log N.

(Refer Slide Time: 37:38)



Summary

- 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 which supports a set data structure, supporting the operations insert, delete and search, each of which take O(log N) time, where N is the size of the set. Therefore, a sequence of N operations takes O(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 balanced trees could be transcribed almost directly to Haskell code.