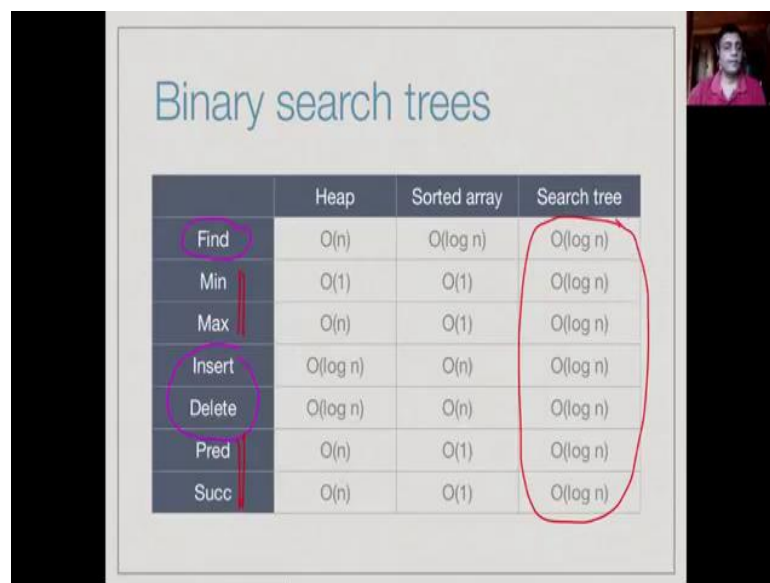


**Design and Analysis of Algorithms**  
**Prof. Madhavan Mukund**  
**Chennai Mathematical Institute**

**Module – 02**  
**Lecture - 40**  
**Balanced Search Trees**

In the previous lecture, we looked at operations on search trees. We claim that these were efficient that we could maintain balance. So, let us see how we can keep search trees balanced.

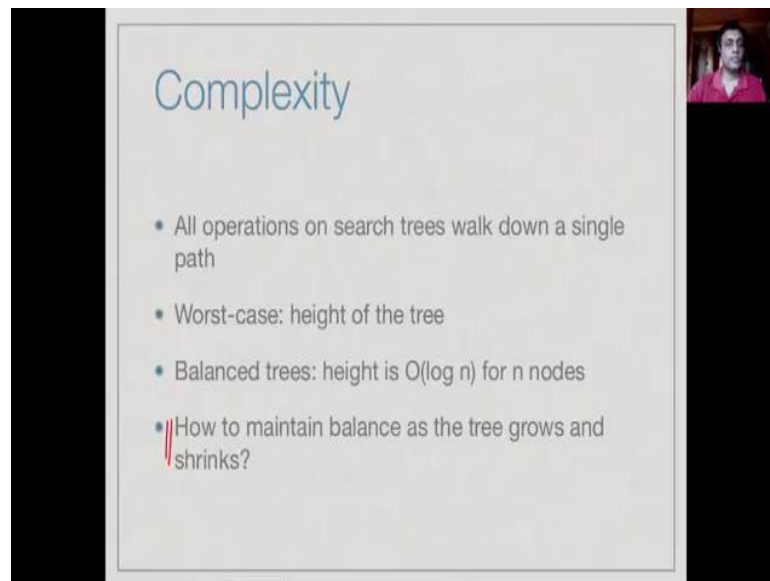
(Refer Slide Time: 00:12)



	Heap	Sorted array	Search tree
Find	$O(n)$	$O(\log n)$	$O(\log n)$
Min	$O(1)$	$O(1)$	$O(\log n)$
Max	$O(n)$	$O(1)$	$O(\log n)$
Insert	$O(\log n)$	$O(n)$	$O(\log n)$
Delete	$O(\log n)$	$O(n)$	$O(\log n)$
Pred	$O(n)$	$O(1)$	$O(\log n)$
Succ	$O(n)$	$O(1)$	$O(\log n)$

So, recall that we are looking at these 7 operations, we want to be able to search for a value, you want to be able to insert and delete values, you also want to be able to compute the minimum and the maximum value in a tree and also find the predecessors and successor of the given value and all of these we claimed would be order  $\log n$  provided the tree is balanced.

(Refer Slide Time: 00:34)

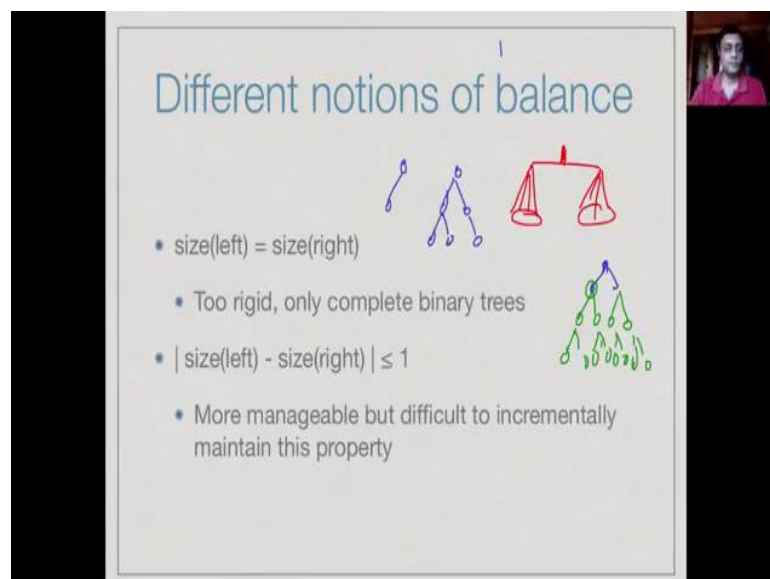


## Complexity

- All operations on search trees walk down a single path
- Worst-case: height of the tree
- Balanced trees: height is  $O(\log n)$  for  $n$  nodes
- How to maintain balance as the tree grows and shrinks?

So, thus because all of the operations as we implemented them, we will walk up and down a single path and so the worst case would be the height of the tree and in our balanced tree the height will always be logarithmic in the size of the tree. So, today's this lecture the goal is to explain, how to maintain the balance as the tree grows and shrinks.

(Refer Slide Time: 00:54)



## Different notions of balance

- $\text{size}(\text{left}) = \text{size}(\text{right})$
- Too rigid, only complete binary trees
- $|\text{size}(\text{left}) - \text{size}(\text{right})| \leq 1$
- More manageable but difficult to incrementally maintain this property

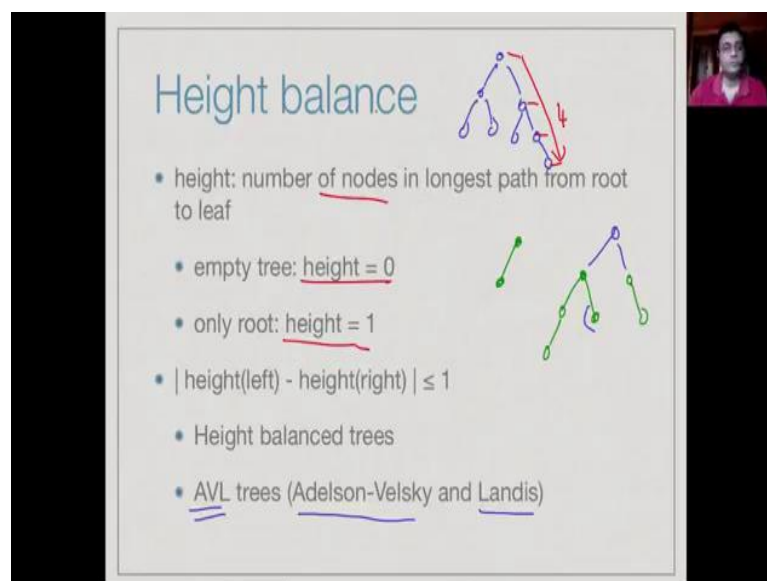
So, trees there are many different notions of balance in a tree, so essentially a balance we should think of it is like a physical balance. So, if you go to an old style vegetable seller, when they will have this kind of a balance and then you want at any point if you hold up a tree by its root, the two sides should be balanced, they should be equal. So, the most direct notion of balance is that the two are exactly that is the number of nodes at the left

is equal to the number of nodes in a right for every node.

Then, it is easy to see that you when you get a complete binary trees, so example you could have a tree which has this structure or if you extend it one more level, then for every node in the left you must extended on the right, but then these must also be balanced, so you must definitely complete this. So, you can have a tree up to 3 levels which is completely filled or up to 4 levels and so on. So, if you want this exact size balance, then it is very restricted.

So, you might have a little bit more flexibility, you might say there we do not want to be exactly equal, we may be wanted it to be at most one off, then you could have structures for instants like this, where you have only a left or you have at this point a left and right put you have only array. These standards structures, now which are not complete binary trees become balanced in this notion. So, this allows us more flexibility and we can get more trees this way, but it is difficult to maintain this property incrementally as we do inserts and deletes. So, we will go for different notion of balance.

(Refer Slide Time: 02:25)



The slide is titled "Height balance" in blue text. It contains a bulleted list of definitions and properties, along with several diagrams of binary trees. The first diagram shows a tree with a root node and a right child, which has a right child of its own, with a red arrow indicating a path of 4 nodes. The second diagram shows a tree with a root node and a left child, which has a right child, with a green arrow indicating a path of 3 nodes. The third diagram shows a tree with a root node and a left child, which has a left child, with a green arrow indicating a path of 2 nodes. The fourth diagram shows a tree with a root node and a left child, which has a right child, with a green arrow indicating a path of 2 nodes.

- height: number of nodes in longest path from root to leaf
- empty tree: height = 0
- only root: height = 1
- $| \text{height}(\text{left}) - \text{height}(\text{right}) | \leq 1$
- Height balanced trees
- AVL trees (Adelson-Velsky and Landis)

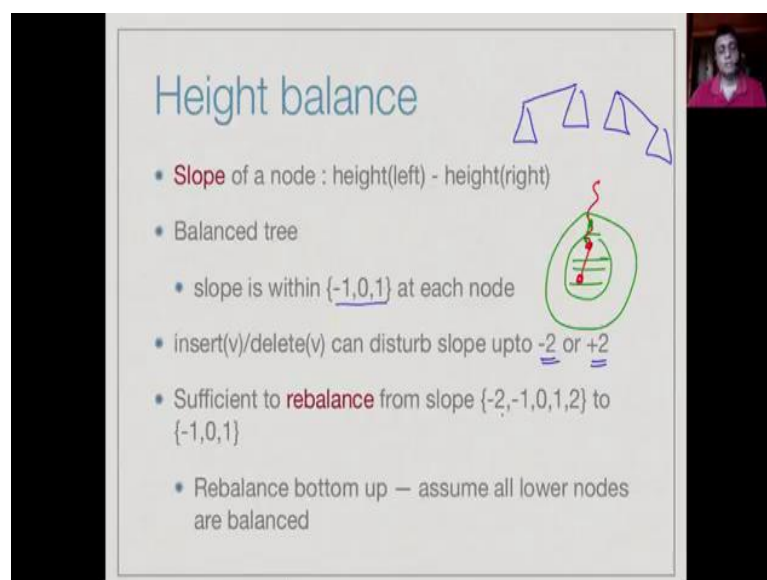
Notion of balance that we will use is not with respect to size, but with respect to height. So, we will say that the height is a number of nodes from the root to a leaf. For example, if I have a tree which looks like this, then here the height is 1, 2, 3, 4 because on this path we have 4 nodes. So, the heights become 4 it is a length of the longest path measured in terms of nodes, the reason we measured in terms of nodes is, then we can distinguish easily, the empty tree from the tree with only are root.

If you measured in terms of edges, the tree with only a root will have height 0, because there are no edges and so would be the empty tree. Whereas, if we measure it in terms of nodes, then the empty tree has height 0 and the tree with only the root has height 1. So, we can distinguish these two. And now in keeping with our earlier relaxation of the size condition, the height balance tree will be one where the height of the left and the height of the right differ the at most 1.

Now, this is more relaxed in the previous thing. For example, now of course, I could start with a height balance tree like this. And then, I could now connect this to form a height balance tree like this and now this which is height 3 tree I can connect with a height 2 tree and form a height balance tree which looks like this. So, the height of the left sub tree is 3, height of the right sub tree is 2 in this recursively the height of left sub tree is 2, the height of the right is sub tree is 1 and so on.

So, we could have things which look quit difference, so size here for instant size is 4 and size is 2. But, nevertheless you can kind of compute that the size even in this case will be exponential in the height or rather the height will be logarithmic in the size. So, these trees are called AVL trees the named after the two people who independently invented them one person called Adelson-Velsky and independently Landis. So, an AVL tree is a height balanced tree which says that at every node the height of the left and height of the right sub trees differ by at most 1.

(Refer Slide Time: 04:53)



**Height balance**

- **Slope** of a node :  $\text{height}(\text{left}) - \text{height}(\text{right})$
- Balanced tree
  - slope is within  $\{-1, 0, 1\}$  at each node
- $\text{insert}(v)/\text{delete}(v)$  can disturb slope upto  $\underline{-2}$  or  $\underline{+2}$
- Sufficient to **rebalance** from slope  $\{-2, -1, 0, 1, 2\}$  to  $\{-1, 0, 1\}$
- Rebalance bottom up — assume all lower nodes are balanced

So, let us refer to the difference between the height as just the slope, so we have a

intuitively in our pictures. So, if it is unbalance then thing is treated, so we could have till this way or till this way, so we call this the slope. So, we let us say this slope is height of the left minus height, so the height of the left is less than the height of the right, then you have a positive slope. If the height of the left is bigger than the height of the right, then you have right, left is smaller than right you have negative slope, left is bigger than right you are positive slope.

So, in a balanced tree since the height difference absolute value must be 1, you can only have three possible slopes throughout the tree, either there is no slope they are exactly the same or it is minus 1 or plus 1. Now, if you can argue very easily that if the current value is of the slope some minus 1 plus 1, when you delete a node, you can reduce one of the heights by 1. So, the height difference can go from 1 to 2 or when you increase you can make the height difference, again go from 1 to 2.

So, the new slope after a single insert or a single delete can be at most minus 1 or plus 2, minus 2 or plus 2. So, what we will end up to do what we will try to do is that whenever we do an insert or a delete we will immediately try to rebalance the tree. So, we would have a single disturbance from minus 2 or plus 2 it will never become very badly unbalance and we will immediately restore the balance to within minus 1 to plus 1.

So, you will do this rebalancing we will also do this rebalancing bottom up, so what happens we will be do an insert, if you remember is that we go down and we find a place to insert. So, this point we add a new node, so therefore now at this point that could be some imbalance, so we fix it, then we will go back to the up this path and we will go there and we will fix the path here, but at this point you will assume that the tree below has been balanced. So, whenever we rebalance the slope which is outside the range, you will assume that the sub trees below that are already balanced, because this balancing as we will see is going to be done bottom up.

(Refer Slide Time: 06:59)

### Unbalanced, slope +2

- Node x has slope +2
- Assume left and right subtrees are balanced
- All slopes in  $\{-1, 0, +1\}$

The diagram shows a node  $x$  with a slope of  $+2$ . It has a left subtree  $TL$  of height  $h+2$  and a right subtree  $TR$  of height  $h$ . The subtrees are assumed to be balanced, meaning all slopes within them are in the set  $\{-1, 0, +1\}$ .

So, here is a typical situation that we would reach after a single operation which removes the balance. So, we might have a node which has slope plus 2 or minus 2, so let us look at plus 2 minus 2 turn out be symmetric. So, we have a node which we call  $x$  which has slope plus 2 and what it means is, it has a left tree and right tree. Such that, the height of the left tree is 2 more than the height of the right tree, remember this slope is right or left minus right or left, so  $h$  plus 2 minus  $h$  will be 2.

Now, recursively we are going to assume that all the slopes here and here are at most plus 1 or minus 1. So, we are assuming that everything below this has been fixed and the only in balance in this sub tree at  $x$  is  $x$  itself.

(Refer Slide Time: 07:50)

### Unbalanced, slope +2

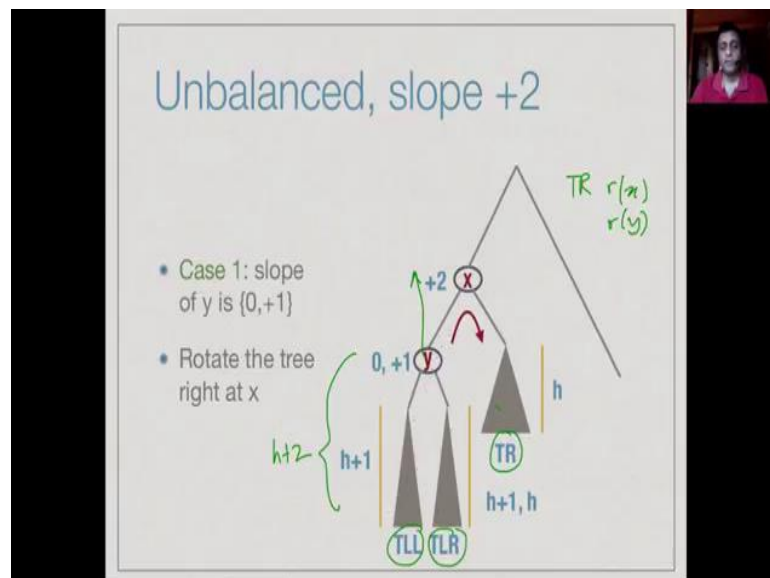
- $TL$  is not empty: expand
- Slope of  $y$  is in  $\{-1, 0, +1\}$
- Bottom up rebalancing
- Case analysis

The diagram shows a node  $x$  with a slope of  $+2$ . Its left child is a node  $y$  with a slope in  $\{-1, 0, +1\}$ . Node  $y$  has two children,  $TLL$  and  $TLR$ , both of height  $h+2$ . The right child of  $x$  is a subtree  $TR$  of height  $h$ . The subtrees  $TLL$  and  $TLR$  are assumed to be balanced, meaning all slopes within them are in the set  $\{-1, 0, +1\}$ .

So, now since the left has height  $h$  plus 2, it has height at least 2,  $h$  can be at most at least smallest  $h$  can be 0, so it has height at least 2, so there at least 1 node here. I mean 2 means that there are at least 2 nodes here, so we have at least 1 node in particular. So, we will expand this by exposing it is root and the root will have in general 2 sub trees, so now this whole thing as height  $h$  plus 2.

So, we will now look at this new node that we have expected. So, this slope is minus 1 0 or plus 1 and we are going to do some bottom up rebalancing, we are assuming everything below it is case. So, I have going to do some case analysis based on what is the slope of  $y$ .

(Refer Slide Time: 08:41)

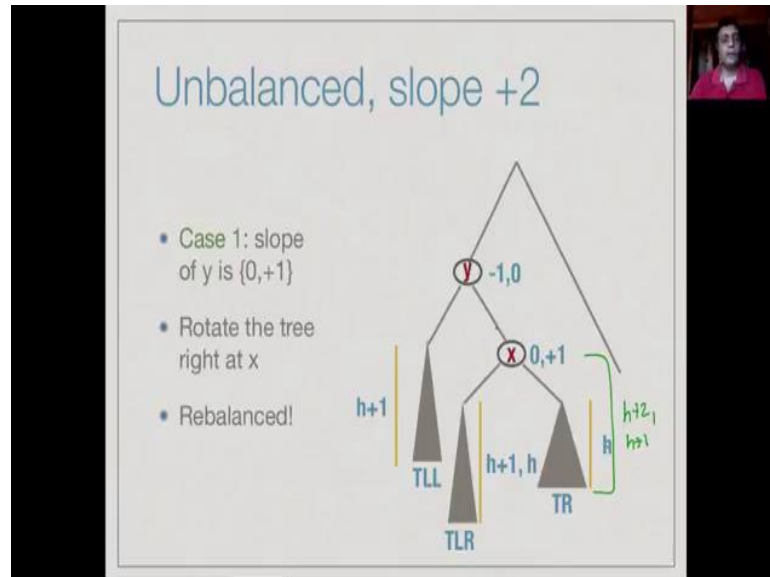


So, let us first look at the situation where the slope of  $y$  is either 0 or plus 1, so if it is 0 or plus 1 it means that so remember this whole thing was  $h$  plus 2 of which 1 node is here, so it is left child must be at least  $h$  plus 1 and because it is slope is 0 or plus 1, the right child is either  $h$  plus 1 in case slope is 0 or it is  $h$  in case the slope is plus 1. So, now this is the current situation as we have it with an unbalanced node  $x$  everything below is balanced. But we have just come to a situation where we try to analyze what is the situation behind.

So,  $x$  is got a balanced unbalance of plus 2 and below it we have why which whereas assuming is either 0 or plus 1. So, now we do this rotation, so we take this tree and we kind of hang it out by  $y$  and we reattach things. So, in this rotation when you hang it out by  $y$ ,  $x$  comes down and now we look at this sub trees, so we have the 3 sub trees, we

have TLL, TLR and TR. So, TR is to the right of x and it is also to the right of y, so it is the right of both, TLR is to the left of x to the right of y, TLL is to left of y, left of x.

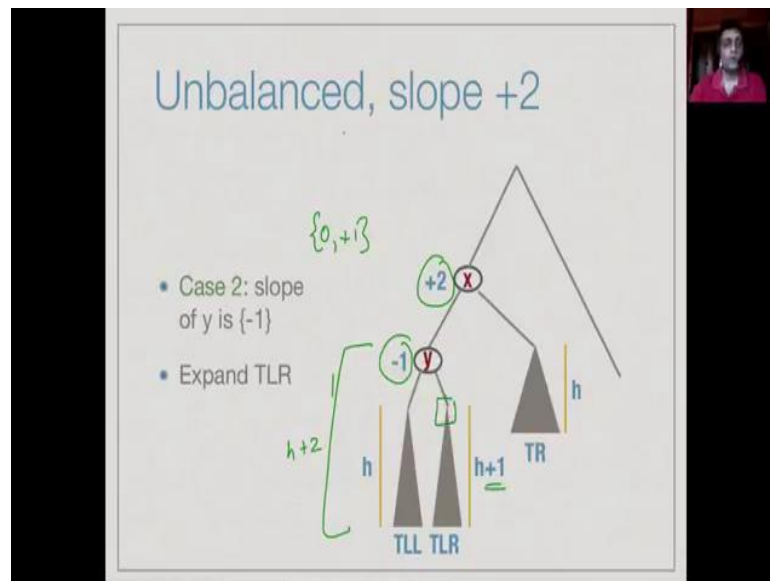
(Refer Slide Time: 09:57)



So, if you go there we find that TR is to the right of both, TLR is to the left of x right of y and TLL is to the left of both. So, we hang up these trees, so now all the values we can verify will be currently organized as a search tree issue. But, now if you look at the slopes, we have just inherited this slope some what we knew that the slope of TLL of h plus 1, TLR is h plus 1 or h and TR is h.

So, this means that if I look at this over all height at this point, it is either h plus 1 or at most h plus 2, so this is h plus 2 or h plus 1. If it is h plus 2, then the height slope at y is minus 1, if it h plus 1 then both sides at h plus 1 slope at y is 0 and if you look at x, here we have h plus 1 and here we have h, so the difference is either 0 or plus 1. So, x is now balanced, y is balanced and by assumption inductively all the grey sub trees are balanced. So, by one right rotation, we have rebalanced the tree.

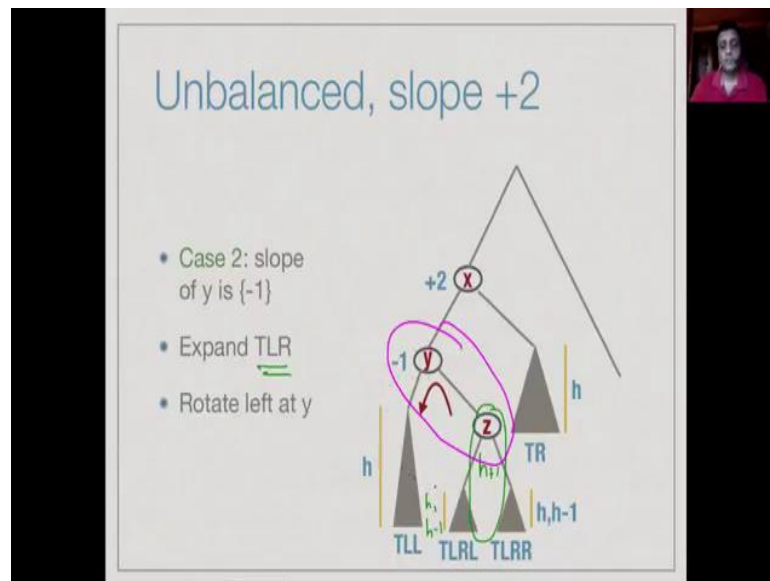
(Refer Slide Time: 10:55)



So, the third situation is that it is not 0 or plus 1, but the slope at while minus 1. So, we have already dealt with the case where 0 or plus 1 is the possibility. So, if it is minus 1 this means that the right sub tree must be strictly taller than the next left sub tree. So, with the whole thing remember again is  $h$  plus 2 the high assumption, because the whole thing has plus 2, so this whole thing is  $h$  plus 2 and this thing is  $h$ .

Now, this  $h$  plus 2 it now beings split of  $y$  and the rest therefore  $h$  plus 1 must be coming in the right and  $h$  1 the left we got the slope by assumption we are assuming slope is minus 1. So, now we are going to expand out this node now it is a  $h$  plus 1, so there is at least 1 node here even a way which is 0 it has at least a root node. So, we will expose a root node as we exposed  $y$  earlier.

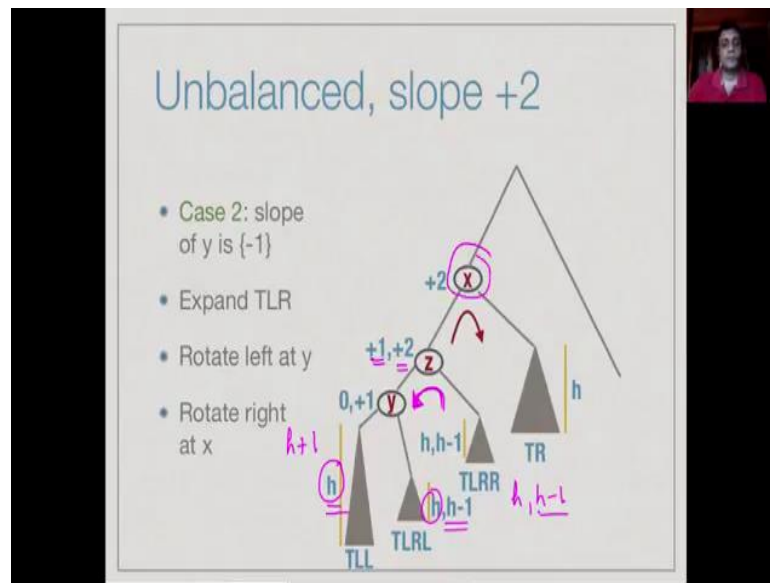
(Refer Slide Time: 11:48)



So, now we are kind of expand it this TLR as z with 2 sub trees TLRL. So, this supposed to indicate let us started their original tree go and left go right and go left, so TLRL go left, go right, go right. So, it is TLRR, so that is notation for the sub trees, so this whole thing was h plus 1. So, therefore, this h plus 1 can come either sides, so either this is h or h minus 1 or this is h or h minus 1 actually is one of then must be h.

Because, we are both are h minus 1 then the whole thing cannot be h plus 1. So, at least 1 is h and it is balanced, so I do not know which way it is, but at least one of these tress as h I will h the other one could have h or h minus 1 and it terms out it will not matter which one. So, now I have a two-step procedure what I will do is, now I will do is similar rotation, but to the left involving these two nodes. So, I will hang up these sub trees here by z.

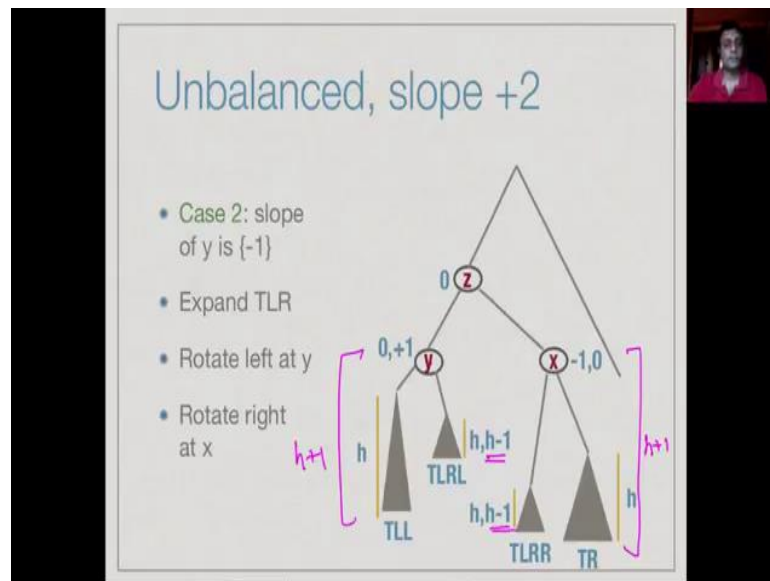
(Refer Slide Time: 12:50)



So, if I hang up these sub trees then what happens is that these 3 trees below I have to be reattached and we reattach some in the correct with TLL is bit to the left of both y and z TLRR is to the right of both y and z and TLRL is to the left of said and to the right of 1. And now, if you go back and check all the heights, then you find that at y I either have height 0 or I have height plus 1. Now, if I look at z then the left hand side is now h plus 1 and the right hand side is h or h minus 1.

So, if is h plus 1 and this is the h then I get plus 1, if it is h plus 1 this h minus 1 I get plus 2. So, in the process now not only did I have x which was unbalanced and temporarily created is z which potentially is unbalance. So, I cannot say for sure it is unbalance, but it could to a unbalance, because is I do not know which is h and which is h minus h 1, but this is only in intermediate steps, so now what I did, so earlier what we did, we will did on left rotation here. So, I am going to follow this a by a right rotation at x.

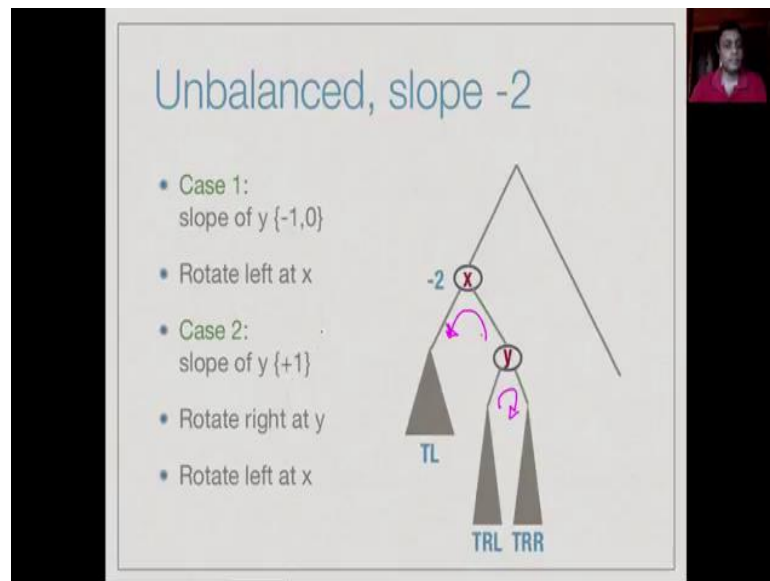
(Refer Slide Time: 14:08)



So, we rotate write a text z goes up x goes down and now again we hang of the 4 trees and there is only one way to hang of you can verify it will go back and check out their hanging. So, TRR must be bigger than z and x, TLL must be smaller than y and z and so on. And now we have a very happy situation which is that if you look at the slope of y, then it is the either 0 or it could be plus 1 because this could be h minus 1.

If you look at x it could be minus 1, because this could be h minus 1 this definitely h or we could be 0. But, this whole thing is definitely h plus 1 because I do have at least 1 sub tree of size h, this whole thing is again h plus 1, because I have at least 1 sub tree of size h and therefore, the slope at 0 at z is 0. So, both all three nodes x, y and z have slopes in the correct range and therefore, I again I will stored the balance.

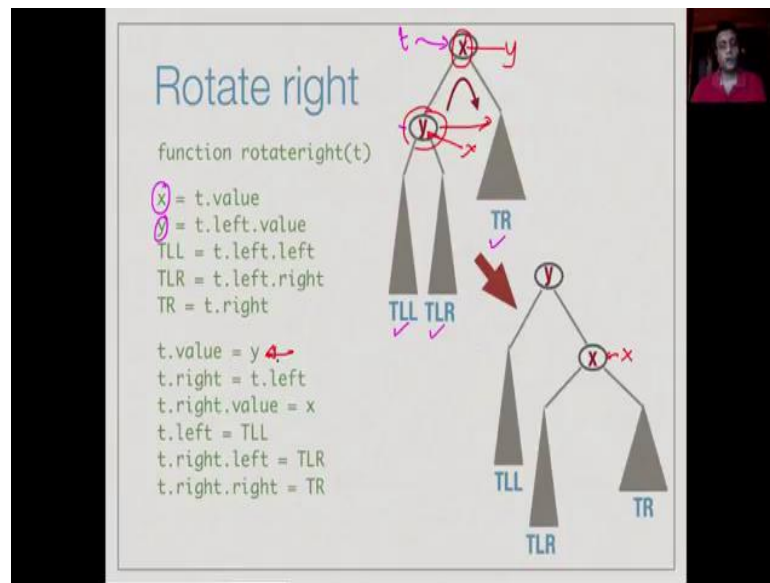
(Refer Slide Time: 14:59)



So, to summarize this what we have said is that if we have plus 2, then we look at the left child of x call it y, if the slope of y is 0 or 1 we rotate at right at x, if the slope at y is minus 1 be first rotate this and then we rotate y. Either case we rotate at x, but first we rotated y in cases show to minus 1. Now, it will turn out that if you have the other set case, the other extreme where the slope at x is minus 2, then will have a symmetric picture, so we will have to expose the right, so I will call it y.

So, then the basic basics operation is that if y has slope minus 1 or 0 it symmetric remember, then we will do a single left rotation x. Otherwise, we will first do a right rotation at y, in case is slope at y is plus 1 and then we will do a left rotation itself. So, we one not look at their minus 2 cases in detail, but it is easy to see that it is symmetric to the plus 2 case.

(Refer Slide Time: 15:58)



So, how do you do these rotations, where will you can just draw this pictures and figure at outs. So, you just give names to everything, so we say that we want to rotate this x down in the y up, now we have are original t pointing here remember. So, it is we cannot change in the node that t is pointing 2, so we remember the value at t we remember the value here. So, we need this names x and y to that contains and then we have this counters pointing to these three trees TLL, TLR and TR and then we reconnect that.

So, what we do is we first replace this value by y, then what we do is we make this node come to the right and we reset it is value to x. So, we put an x here and we have note it there and then we have hang of below that top node on the left we put TLL and below this right new right node on the left if put TLR on the right would TR. So, you just do this kind of reconnection is like a you know un hooking and re hooking trees and just we just keep track of the all the names. So, that everything is un hook and re hook correctly.

So, it is a very simple, so you notice that this the kind of constant set of operation involving of few of these left and right pointers. So, it will be regardless of the size of the tree, it is a very local operation. So, we can treat this as one constant unit operation.

(Refer Slide Time: 17:25)

### Rotate left

```
function rotateleft(t)
y = t.value
z = t.right.value
TLL = t.left
TLRL = t.right.left
TLRR = t.right.right

t.value = z
t.left = t.right
t.left.value = y
t.left.left = TLL
t.left.right = TLRL
t.right = TLRR
```

The diagram shows a binary tree structure before and after a left rotation. Initially, node Y is the root, with a left child (TLL) and a right child Z. Node Z has two children: TLRL and TLRR. A red arrow points to the resulting tree after the rotation. In the new tree, node Z is the root, with node Y as its left child. Node Y now has two children: TLL and TLRL. Node Z's right child is now TLRR.

And a similar thing for the left, we give names and just we do this updating exactly has we had done before. So, again these are constant set of operations which implements rotate left.

(Refer Slide Time: 17:38)

### Rebalance

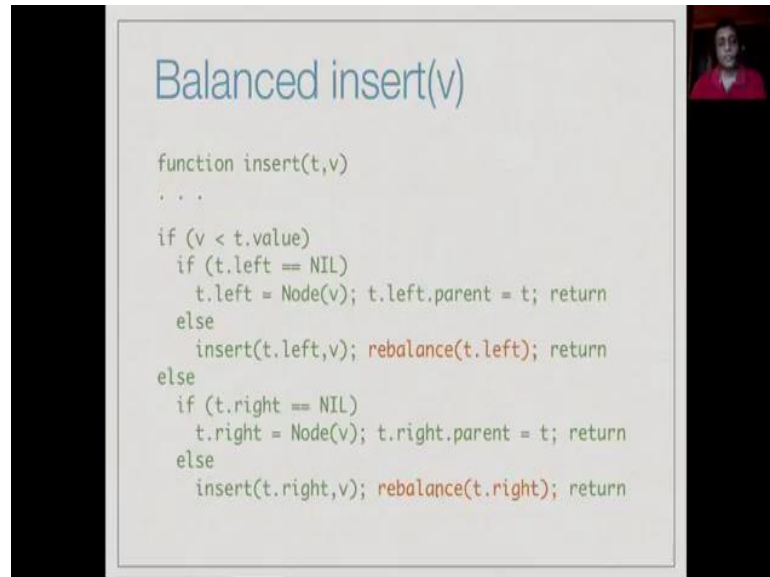
```
function rebalance(t)
if (slope(t) == 2)
  if (slope(t.left) == -1)
    rotateleft(t.left)
    rotateright(t)
  else
    rotateright(t)
if (slope(t) == -2)
  if (slope(t.right) == 1)
    rotateright(t.right)
    rotateleft(t)
  else
    rotateleft(t)
return
```

The diagram illustrates rebalancing operations. The top part shows a node with a balance factor of +2 and a left child with a balance factor of -1. A green arrow indicates a left rotation on the left child, followed by a right rotation on the root. The bottom part shows a node with a balance factor of -2 and a right child with a balance factor of +1. A blue arrow indicates a right rotation on the right child, followed by a left rotation on the root.

So, now that we have that two rotations, then what does rebalancing set, rebalancing set that if I have node which has plus 2, then I will expose it is left child and I will check it slope, if it is slope is minus 1 then first I will do a left rotation there and then regardless I will do a right rotation at the top. In the symmetric cases, if I have a slope just minus 2 at the top, then it will have a right child and then if this as plus 1 when I will first to a right rotation here and then regardless I will do a left rotation of the top. So, rebalancing is

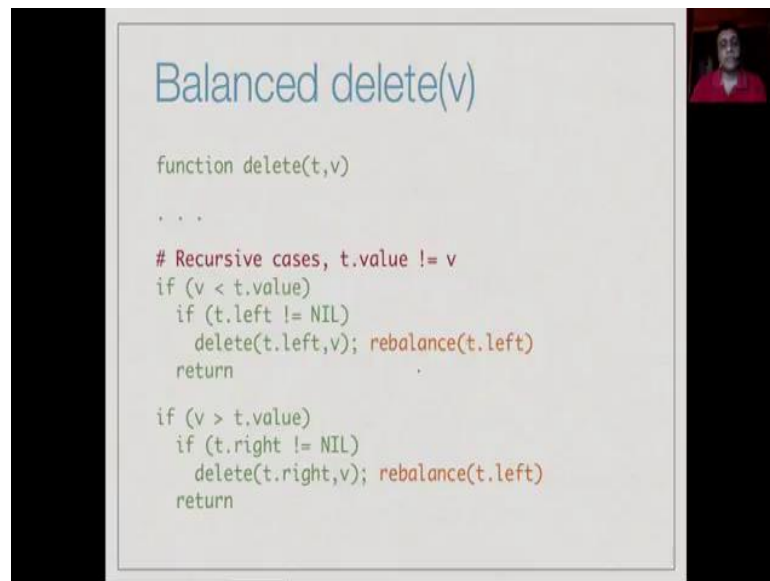
these two steps and rotation this is set up other steps, so basically rebalancing at a given node is a constant number operations at that node.

(Refer Slide Time: 18:24)



So, now what we do is every time we make our change in our tree which could affect the balance of the slope of a node, we do a balance. So, when we in our earlier code for insert, when we did a recursive insert in the left of a tree we rebalanced, similarly we insert in the right of the tree we rebalanced, we just introduce this rebalance code and note is that this a constant operations for this node. So, you will do this all the way along the paths. So, this will do a log n times in constant number operations, so this would not affect anything in terms of the asymptotic complexity of our operation it will be a logarithmic number of constant time operations for each node.

(Refer Slide Time: 19:04)

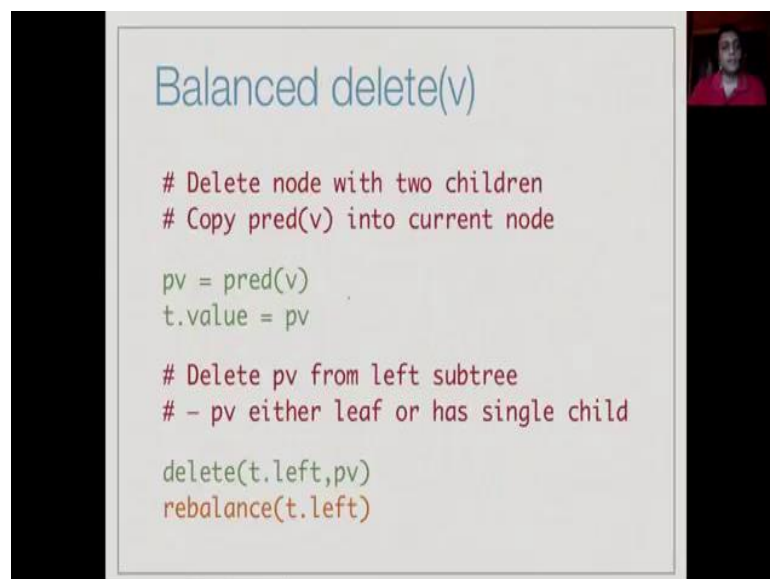


### Balanced delete(v)

```
function delete(t,v)
. . .
# Recursive cases, t.value != v
if (v < t.value)
    if (t.left != NIL)
        delete(t.left,v); rebalance(t.left)
    return
if (v > t.value)
    if (t.right != NIL)
        delete(t.right,v); rebalance(t.right)
    return
```

The same with the delete, where above we do a recursive delete we rebalance.

(Refer Slide Time: 19:10)



### Balanced delete(v)

```
# Delete node with two children
# Copy pred(v) into current node

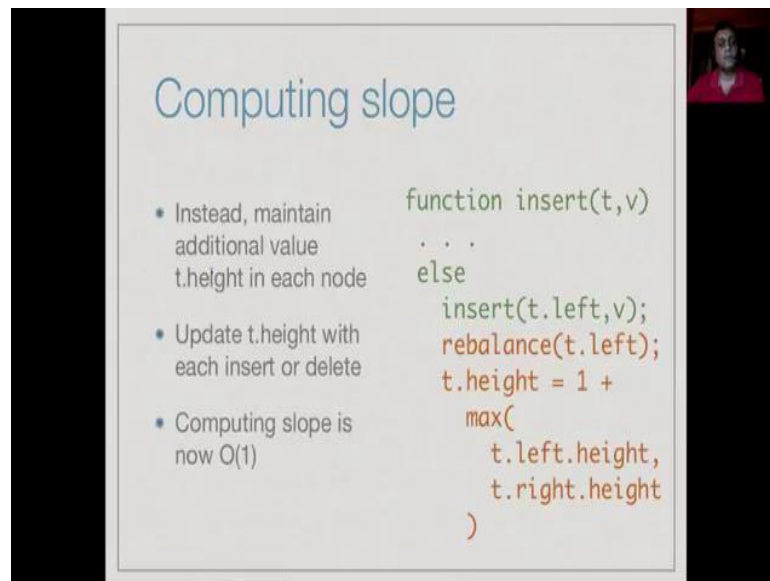
pv = pred(v)
t.value = pv

# Delete pv from left subtree
# - pv either leaf or has single child

delete(t.left,pv)
rebalance(t.left)
```

And then there was a case where we deleted the maximum value or the predecessor, so again we do a rebalance. So, where ever we had are insert and delete affecting the structure of the tree we just make sure that we rebalanced the tree that ((Refer Time: 19:22)).

(Refer Slide Time: 19:23)



### Computing slope

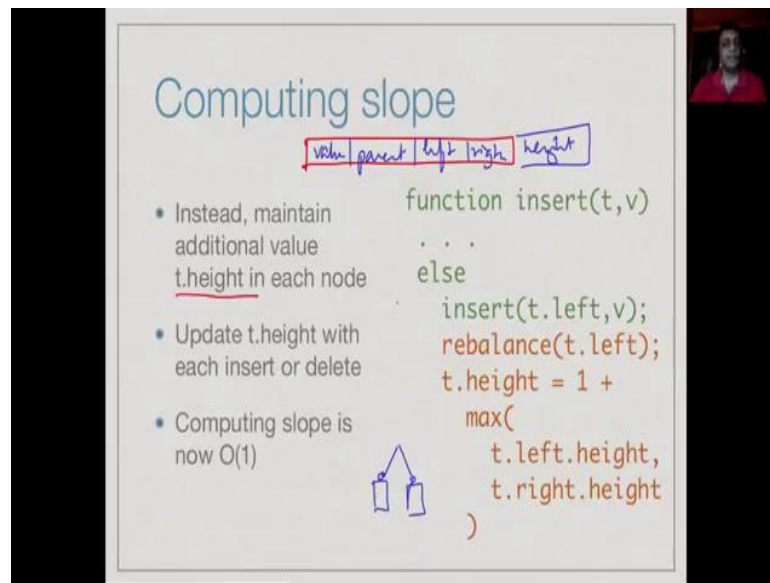
- Instead, maintain additional value `t.height` in each node
- Update `t.height` with each insert or delete
- Computing slope is now  $O(1)$

```
function insert(t,v)
...
else
    insert(t.left,v);
    rebalance(t.left);
    t.height = 1 +
        max(
            t.left.height,
            t.right.height
        )
```

So, there is one point we have to be bit careful about in this thing, so we said that we have to do all these rebalancing. So, if you go back to rebalancing, so rebalancing requires to compute the slope and slope we said is define to the height of the left minus height of the right. Now, it is possible to compute the height on the fly, the height of the tree is recursively computed, if it is nil it has height 0; otherwise, you recursively compute the height of the left in the right and then accounting for this node you add one to that.

So, you take the maximum of the two sub trees and add one, but this unfortunately involves examining every node in the tree. So, this will be propositional to order of the size of the tree. So, this will be exponential operation in some sense depend I mean compare to the path, so we are try to do login operation, this will be a order n operation. So, this will be killing all our attempts to do something efficiently, because in order to compute the height we actually have to see the entire tree which is not we want to do.

(Refer Slide Time: 20:28)



The slide is titled "Computing slope". At the top, a node structure is shown with fields: `val`, `parent`, `left`, `right`, and `height`. The first four fields are grouped in a red box, and `height` is in a blue box. Below this, a bulleted list explains the approach:

- Instead, maintain additional value `t.height` in each node
- Update `t.height` with each insert or delete
- Computing slope is now  $O(1)$

To the right of the list, a code snippet for the `insert` function is shown:

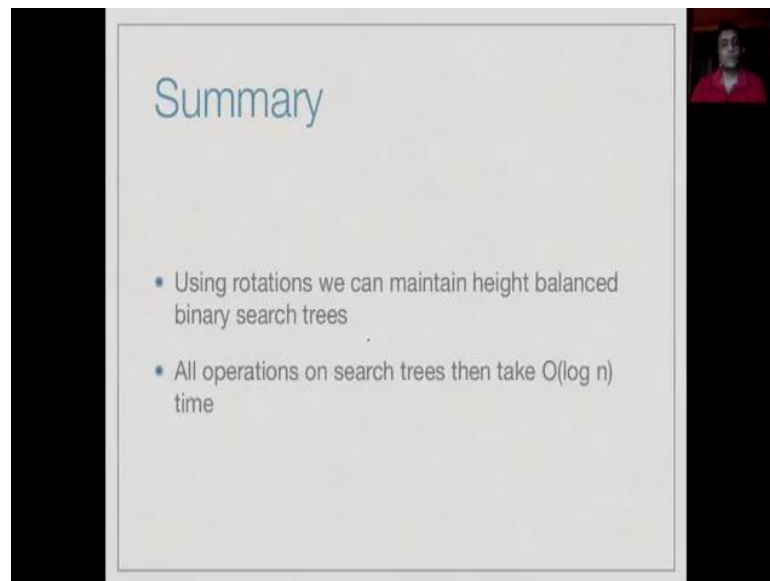
```
function insert(t,v)
...
else
    insert(t.left,v);
    rebalance(t.left);
    t.height = 1 +
        max(
            t.left.height,
            t.right.height
        )
```

A small diagram of a tree node with two children is also present.

But, we can get around this by just keeping the current value of the height and updating it each then. So, we have additional things, so we have in our tree node, we currently had the value, parent, left and right. So, now we just add one more field, so now whenever we do any rebalancing, then the height of this tree may change the current node. So, we just look inductively we assume that below has the height got set correctly. So, we look at the two heights which are locally there add want to the maximum ((Refer Time: 21:07)).

Now, this is just looking up one value and are two neighbors below to two children. So, this now becomes a constant time operations, it does not require is to traverse entire trees. So, as we are rebalancing we readjust the heights and every time you want to check to slope we just have to check the value of the two tree is below us and check that difference of their heights. Because, that will be given locally by this height field in the node. So, we have to be careful not to compute height recursively, but to store the height as part of the tree and also update that with every update.

(Refer Slide Time: 21:39)



## Summary

- Using rotations we can maintain height balanced binary search trees
- All operations on search trees then take  $O(\log n)$  time

So, to summarize you can use rotations to maintain height balanced binary search trees and then height balance search tree we have claimed that the height is going to be logarithmic in the size. And since all our operations are propositional to the height, because they all go along and one path, all these operations namely find, insert, delete, minimum, maximum, predecessor and successor can all be done in logarithmic time.