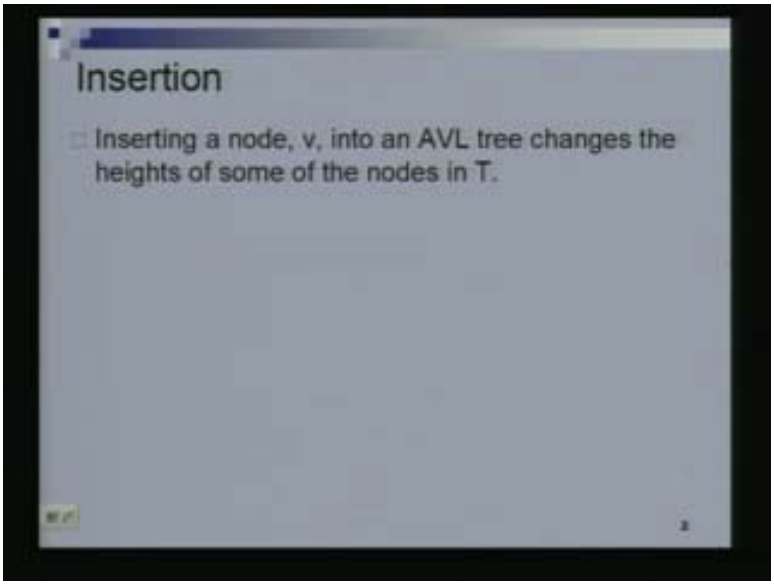
Data Structures and Algorithms
Dr. Naveen Garg
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi
Lecture – 12
Avl trees (Contd.)

Today we are going to continue our discussion on AVL trees. In particular we are going to look at the insertion and deletion procedure in an AVL trees. So we will begin with insertion.

(Refer Slide Time: 01:24)

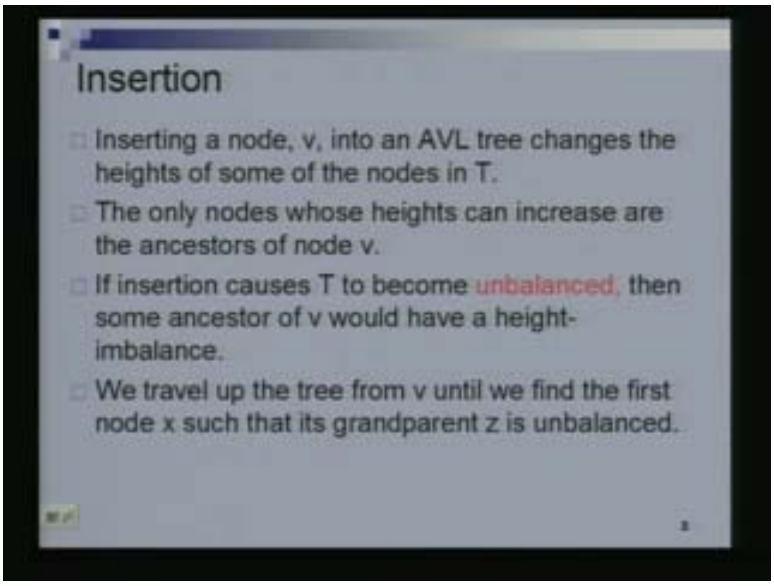


We had started this discussion on insertion in the last class also. Suppose I am trying to insert a node v into an AVL tree. Actually what I am trying to do is insert a key. What is the process of insertion in a binary search tree? First you find where the key is, you go to that place put the key there. Let say the node in which I put the key is v. This is something we had started the discussion, if as a consequence of this insertion it does not remain an AVL tree that is because the height balance property is violated. What are the nodes whose heights could change as a result of this insertion? Which of the nodes whose heights could change? Recall we defined the height of the node as the height of the sub tree rooted at that node. So this is large tree and some where below I insert a particular node. Which are the nodes whose heights could change? It could only be the ancestors of this particular node. Not because it is written here but you should also understand why it is only ancestors. Because it is only in the ancestors of that node whose subtree has changed as result of this insertion process.

For any other node its sub tree has not changed it remains the same as before. So the ancestors of this node, their heights might change. And change means it will only increase. Because we have added a particular node. So it is the ancestor of these nodes whose height might increase as a consequence of this insertion. So if the insertion causes the tree to become imbalance or

unbalanced, then some ancestor of this node v is the culprit and it is the place where one or more ancestors would have a height balance problem.

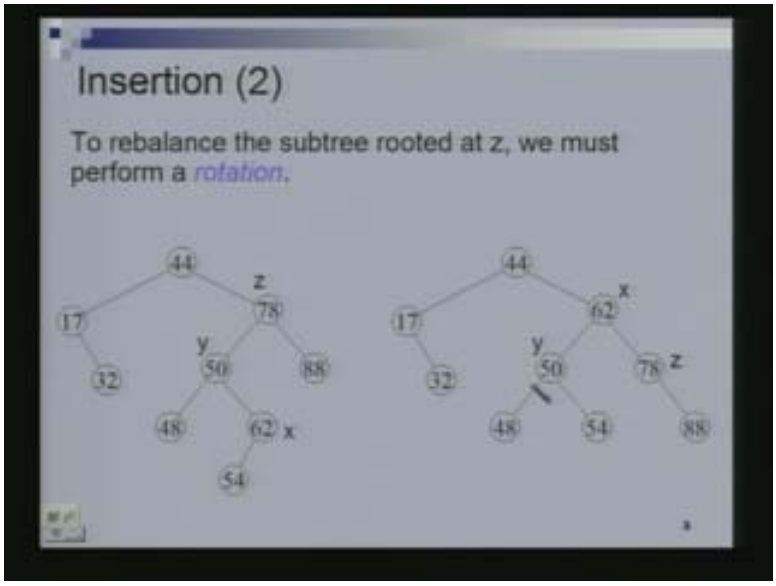
(Refer Slide Time: 03:47)



Height problem means it will become height imbalanced. Height imbalance means one left sub tree and right sub tree, the difference in heights is more than a one. What are we going to do? We are going to essentially travel up the tree from this node v. Travel up the tree which means just keep following the parent pointer till we identify the node z which is unbalanced. So I have said, till we find the first node x whose grandparent is unbalanced, but you can also think of it as, I find the node z which is unbalanced and x is its grandchild. Which grandchild? The grandchild that we traversed or went through the grandchild on the path. Because the node can have many grandchildren. I will show you an example and this will be clear. We will call y as the parent of x. So y is the parent of x and then the child of z.

Suppose this was the situation we had. This was an AVL tree which is given in the below slide and I insert let say 54 into this tree. So 54 would come here. Why, because it is larger than 44, smaller than 78, larger than 50, smaller than 62 and so it would come here (Refer Slide Time: 04:54). Now if there is a problem, actually this tree is not a height balanced any more. This is not an AVL tree any more. If there is a problem we said that the problem would be on one of these nodes. Note that these are the only nodes whose heights are changed. Earlier the node 62 had a height of zero, now it is one. The node 50 had a height of one, now it is 2. The node 78 had a height of 2, now it is 3. The node 44 had a height of 3, now it is 4. Which is the first node on this path which is now imbalanced? Is the node 50 imbalanced? No, the difference of height is 1.Is the node 78 imbalanced? Yes, the difference of height is 2, the node 88 is 1 and the node 50 is of height 3. The height of no other node has changed. The height of the node 17 is not changed.

(Refer Slide Time: 06:56)



Did you understand why the heights of these nodes on this path would change? The node 78 will be z and x would be its grandchild on the path and the parent would be y. Lets call that x is a node, y is a node and z is a node. So x is the node here in 62. So we travel up from 54, we find the first place where the imbalance happens let us call that z. And x is the grandchild of z, grandchild means child's child. So child is y and its child is x. Now we are going to rebalance this tree, so to rebalance this tree in particular we are going to rebalance this sub tree. The sub tree rooted at z and we will do that by performing a rotation. This is what will happen after the rebalance, this is what the tree would look like. What we are going to do? Just understand how we came up with this picture. As you can see I have only changed this sub tree, the one containing these 6 nodes 48, 50, 54, 62, 78, 88. They are here the 6 nodes but organized in a manner so that this node 62 is not height imbalanced any more. And neither this node 44 is height imbalanced. We are going to understand this process today.

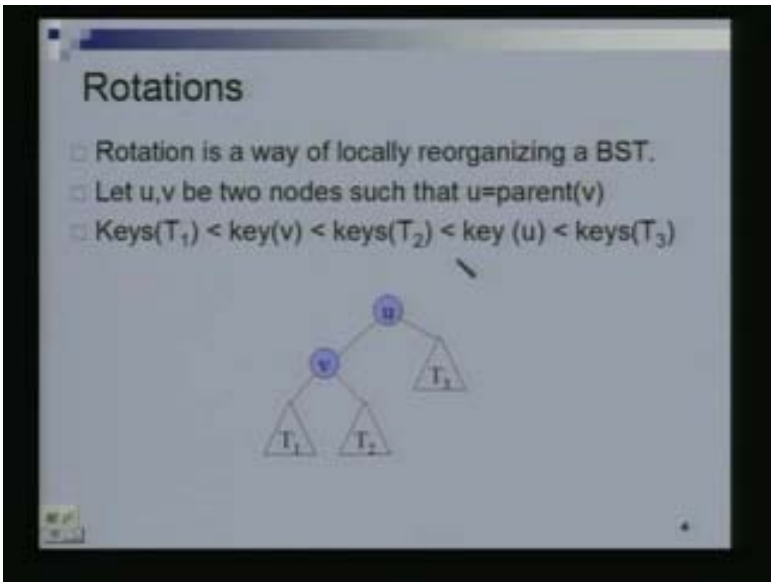
Let us first understand what is a rotation. In the previous slide I used this term rotation. What does rotation mean? So rotation is the way of locally reorganizing a binary search tree. The picture shown in the below slide is a part of my binary search tree. This could be a huge tree but I just consider a part of it. So u is one node, v is its child and these are some sub trees. The T_3 is the sub tree rooted at the right child of u and this T_2 is the sub tree rooted at the right child of v. The T_1 is the sub tree rooted at the left child of u. This T_1 could be a null tree, no node or could be a huge tree, I do not care. What do I know? Because it is a binary search tree and I know all the keys in T_1 are less than the key in v. All the keys in T_2 are more than the key in v.

Keys in T_2 are less than the key in u and keys in T_3 are more than the key in u. This follows from the property of binary search tree. What is the rotation step going to be? First what we are going to do is let us just forget these links and let just look at this (Refer Slide Time: 09:02). And now this is what a rotation does. What has happened? V has become the parent of u. That is what a

rotation is and will put the links back. What happened was v became a parent of u. The binary search tree property still holds by the way. Keys of T_1 are less than v, keys of T_2 are more than v so they come here and they are less than u so they come to the left of u. It is still a binary search tree but we have done some local reorganization and this will be very useful and we will see why.

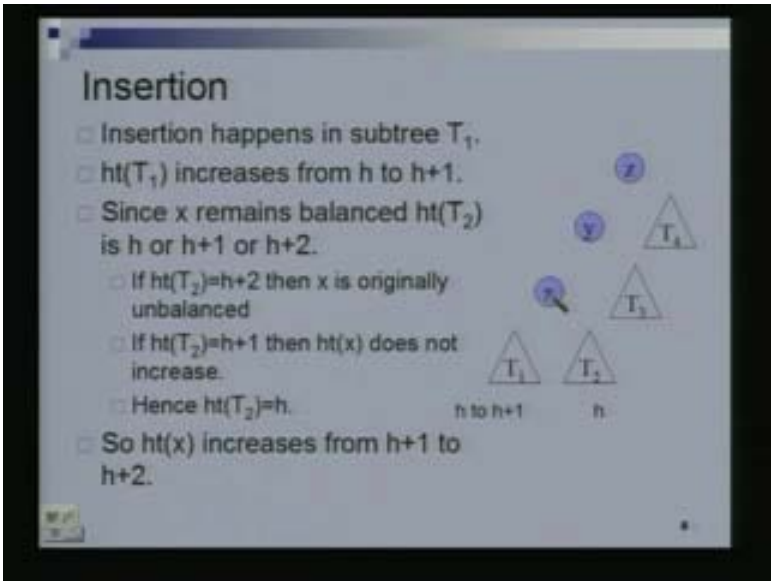
So T_1 remains to the left of v. T_3 was to the right of u, if you remember. T_3 was to the right of u, T_1 was to the left of v, T_2 was to the right of v. So T_3 remains to the right of u, T_1 remains to the left of v but T_2 moves from being to the right of v to the left of u. It is now the left child of u. Everyone follows what a rotation is.

(Refer Slide Time: 07:32)



Now let us see how we will use these rotations to do our insertion. Suppose the insertion happens. This is the tree which is given in the below slide and I have not drawn the links. But it should be clear what the links are. Who can tell me what the links are? The y is the child of z, x is the child of y. These two T_1, T_2 are the children of x, T_3 is the right child of y and T_4 is the right sub tree of z. So in the next few slides you will see pictures with out these links. But that is just to avoid the clutter. It should be completely clear about the relationships. Suppose I did an insertion in T_1. And these are the x y z that we encountered in the procedure. May be insertion happens some where in some leaf, we went up along the path towards the root. The root may be some where here (Refer Slide Time: 11:48). And z was the first place at which we had an imbalance. And y was the child of z and x was the child of y on this path that we took.

(Refer Slide Time: 16:29)

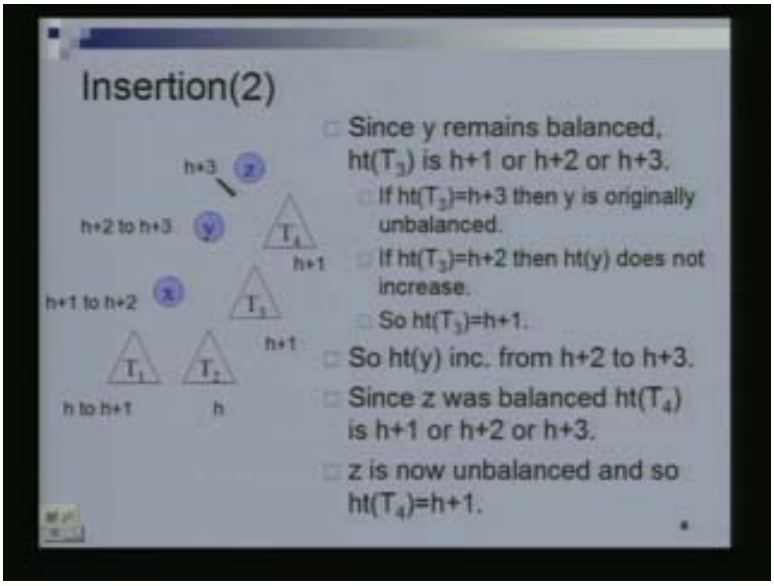


I have taken one picture but it could also be different. The y could have been the right child of z and x could have been the left or the right child of y. How many different cases are possible there? It is 4. The y could be the left of the right child of z and x could be the left of the right child of y. Two times two, 4 different cases. Am looking at one particular case now. That y is the left child of z and x is also a left child of y and the insertion happened in T_1. I am using this ht (T_1) to denote the height of the particular thing. The height of T_1 let us say originally it was h and now because of the insertion it became h+1. It cannot increase by more than a one because after all I am just adding one node. So the increase in height can be at most a one and let say there was an increase in height. So there is an increase in height of this node x also.

There is an increase in height of node y and there is also an increase in height of node z. That is why z became an imbalanced. If there was no increase in height of y then z would not become imbalanced. (Hindi Conversation) Everything is the same as before that means height of y is also increased. And height of y has increased because height of x has increased. Height of x has increased because height of T_1 has increased and height of T_1 let say increased from h to h+1. What can we say about the height of T_2 now? What is the height of T_2 ? So x is balanced even after the insertion, because z was the first node which was imbalanced. So x was balanced after insertion. If x was balanced after insertion (Hindi Conversation) (Refer Slide Time: 14:20-14:52). So height of T_2 is one of these 3. Which one? Can it be h+2? If it is h+2 then originally x is imbalanced, because originally height of T_1 was h. If this (T_1) is h and this (T_2) is h+2 and then this (x) was imbalanced even to begin with, but that is not the case. Originally it was an AVL tree so the height of T_2 cannot be h+2.

Can it be h+1? If it is h+1 then height of x does not increase. Because this (T_2) is h+1. Then that means what was the height x to begin with? It is h+2. If the height of this increased from h to h+1, even then its height remains h+2. The fact that the height of x has increased implies that this (T_2) cannot be h+1. If this (T_2) was h+1 then the height of x did not increase. It remained what it was before that is h+2. This implies height of T_2 cannot be h+1. It has to be h. So height of T_2 is h. If height of T_2 is h and height of this (T_1) has increased from h to h+1, then what about height of x? What was the original height of x? The original height was h+1 and now it has become h+2. The height of x has increased from h+1 to h +2.

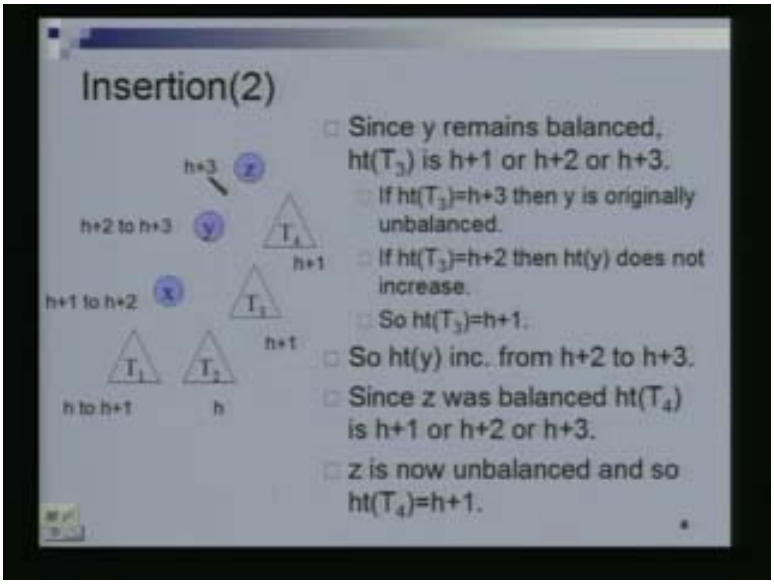
(Refer Slide Time: 20:05)



Let us continue this argument. So this is the picture so far which is given in the above slide. We have argued that the height of T_1 has increased from h to h+1. Then height of T_2 is h then the height of x is increased from h+1 to h+2. What about the height of T_3? Since y remains balanced, the new height of x is h+2. And this y is height balanced. The height of T_3 is h+3 or h+2 or h+1. One of these 3, because the difference in heights can only be one. So its one of these T_3. If it is h+3, we are repeating the argument roughly. If it is h+3 then that means y was originally imbalanced because original height of x was h+1. So y is originally imbalanced. Height of T_3 cannot be h+3. If it is h+2 then that means that the height of y has not increased. Because if this (T_3) was h+2 then the height of y that means originally was h+3 (Hindi Conversation). So height of T_3 cannot be h+2. Height of T_3 has to be h+1. So height of T_3 is h+1. If the height of T_3 is h+1, what is the height of y? Originally it was h+2 because both of these guys were h+1 (Refer Slide Time: 18:21). So this (y) was h+2 originally. And now it has become h+3 so it increased from h+2 to h+3. What about height of T_4? Note that z is imbalanced.

The new height of y is h+3. So what should the height of T_4 be? Or it is h+5, initially it was balanced. Since z was balanced, height of T_4 is h+1 or h+2 or h+3. Since this was originally h+2, this (T_4) could only be h+1, h+2 or h+3 and since it is now unbalanced it cannot be h+2 or h+3, it has to be h+1. The height of T_4 is h+1. What is the height of z initially? This was originally h+2, this (T_4) was h+1, so this (y) was h+3 originally (Refer Slide Time: 20:02). And now of course its heights become h+4, but now we will do some rotation and stuff like that. So that will reduce its size, so its original height was h+3.

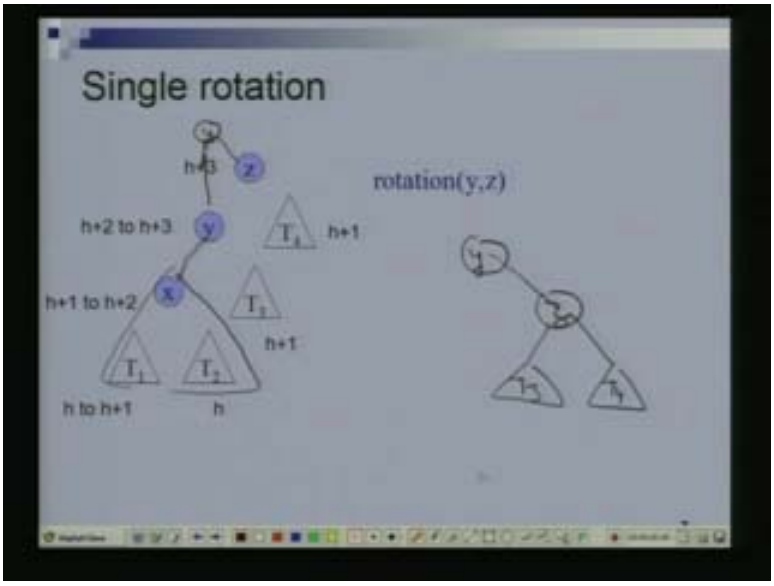
(Refer Slide Time: 20:05)



So we will keep this picture. This is the final thing we argued. These are the heights of the various things. So when I said from here to here that is from h+1 to h+2, the first thing is, what it was originally and what it is now. We just need to look at the new values. We are going to do a rotation around this pair (y, z). What does rotation do? Rotation is going to rotate this, so that y is now going to become the parent of z. What do we want to do? We want to move this y up so that it will come here and y will become the parent of z. This is what the rotation is.

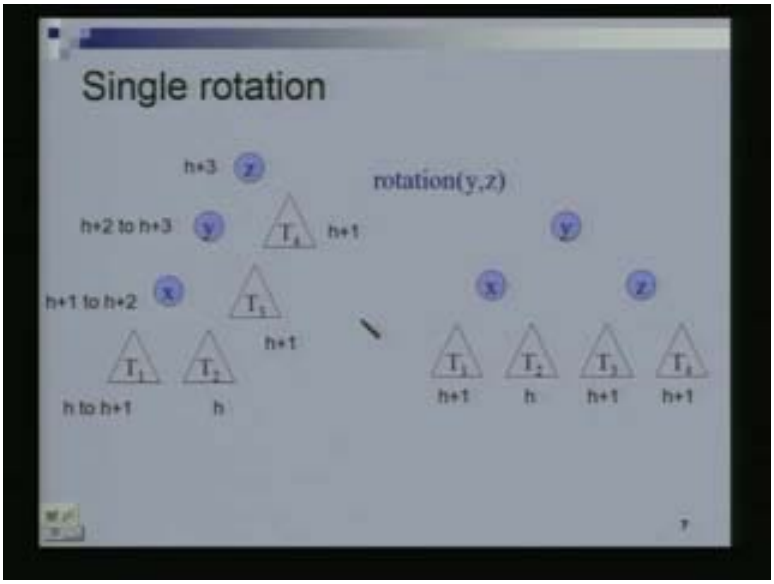
Let me just draw it here. So y will now become the parent of z which is shown in the below slide. Where will I put these two? The x is one sub tree, this remains as it is. This will not be changed. So T_4 will remain at the right and T_3 will come to the left and this big piece under x will remain as it is. That is what the rotation was. This is what will happen.

(Refer Slide Time: 22:07)



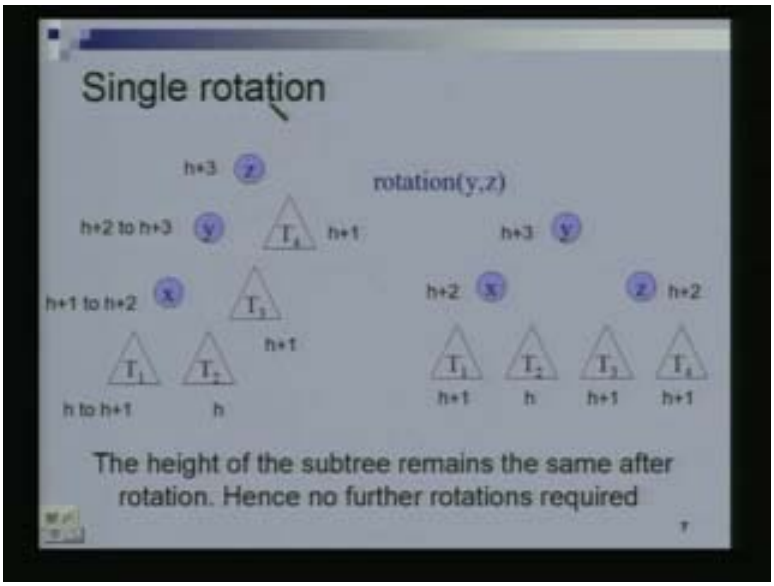
The y has become the parent of z, these two T_4 and T_3 are the children of z. And this entire thing under x was to the left. Again I have not shown the links but you should be clear what the links are. I have written down the heights. The height of T_1 was h+1, T_2 was h, T_3 was h+1 and T_4 was h+1. After a rotation we already saw that the binary search tree properties are maintained. So this is still a binary search tree, but now we want to argue that the height balance property is also restored. (Hindi Conversation)

(Refer Slide Time: 22:31)



Where was the imbalance happening? At z, as you can see this has height h+3, T_4 has height h+1 so this is height imbalanced (Refer Slide Time: 23:20). What is the height of node x? It is h+2 because this (T_1) is h+1 and this (T_2) is h. What is the height of node z? It is h+2. What is the height of node y? It is h+3. Is everything balanced now? This x is balanced because there is only a difference in height of one. This z is balanced because there is no difference. This y is balanced because there is no difference. We have done a rotation. This is called a single rotation. You will soon see why it is called a single rotation. How much time does this operation take for just one rotation? We just have to do a constant number of operations. May be z will have to become a child of y. So there will be 3 or 4 different reference changes that you have to do. May be 5, may be 6 or some constant number, independent of the number of nodes in the tree. Now one interesting thing is happened. The original height of z was h+3. That is why we had written this h+3 here. After this rotation the height of this sub tree is also h+3. Whatever was the original height of this (z) is the new height of this sub tree (y) also.

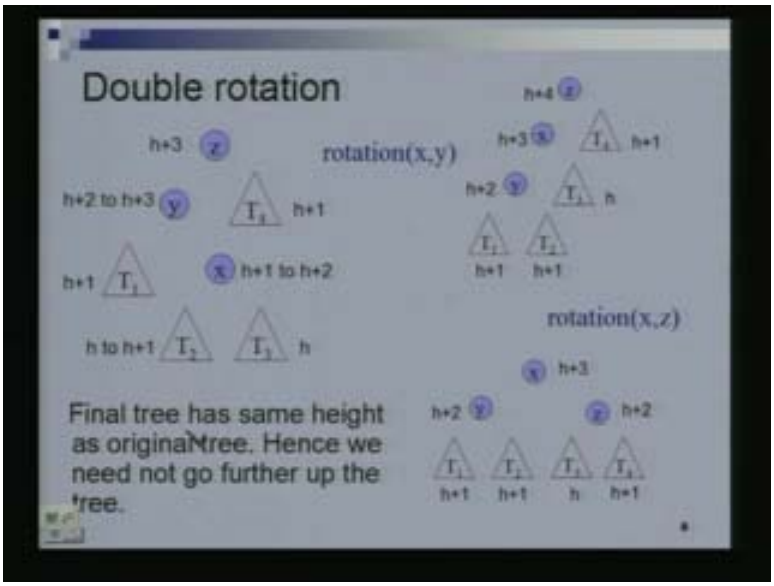
(Refer Slide Time: 25:01)



So the height of the sub tree remains the same after the rotation. The h+3 was the height before insertion, after we inserted and did the rotation the new height also becomes h+3. Why is this important? Because now I do not have to go up further. Because now any ancestor of this, its height would not change any more because whatever was the original height of this h+3 of this node is the new height of this sub tree also. So any of the ancestors of z, their heights would become the same as before. And so there will be no imbalance on them. After I did my insertion, I started moving up I find the first place where there was an imbalance. I did the rotation and I am done, I do not have to go up any further. We have actually considered only one case. One out of 4 different cases. Why one case? Because we said y is the left child of z and x is the left child of y. Now there is one symmetric case, which is the symmetric? Both are right, right. That I am not going to handle because I trust all of you can believe me that it is the symmetric case completely.

The other case is this one. Where x is let say the right child of y which is the left child of z. This has the symmetric case that is, y is the right child of z and x is the left child of y. Again that is completely symmetric and will not handle that. So let us repeat the argument that we had.

(Refer Slide Time: 31:18)



So once again I am assuming that the insertion happens in T_2. It could happen in any one of these that is T_2 and T_3 but again it is symmetric. Let us assume it happens in T_2. So this height went from h to h+1. What about the height of T_3? Since x is balanced, the height of T_3 is either h+2 or h+1 or h. If it is h+2 then that means x was originally imbalanced. If it is h+1 then that means the height of x is not changed. So it has to be h. I am repeating the argument, it is the same as before. If T_3 is h and the new height is h+1. What is the new height of x? It is h+2. What was the original height? It is h+1. So its height moved from h+1 to h+2.

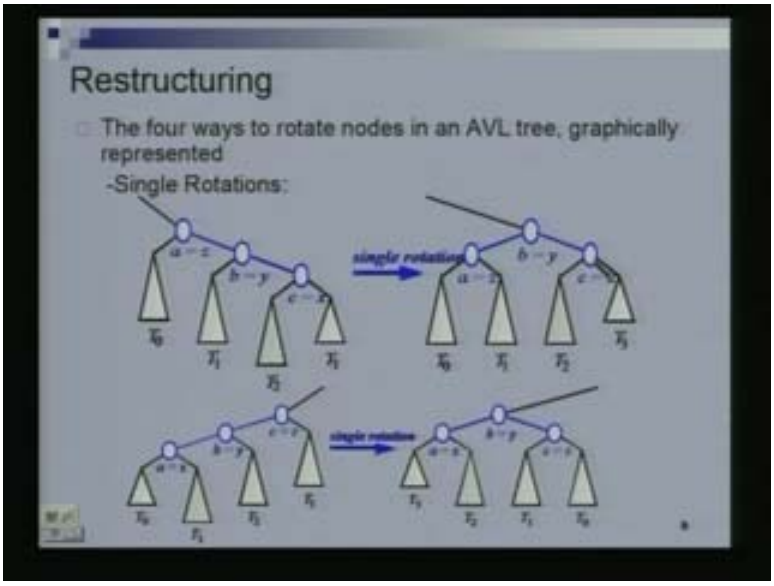
Now let us look at the height of T_1. Since y is still balanced then that means the height of T_1 is either h+3 or h+2 or h+1. If it is h+3 then that means y was originally not balanced. If it was h+2 then the height of y has not increased. It has to be h+1, which means the height of y has increased from h+2 to h+3 which means that now since z is imbalanced, the height of T_4 has to be h+1. Which implies that the original height of z was h+3. Exactly the same as before, we do not make any difference. But now the rotations will have to be a bit different. In x, y and z, which of these 3 keys is the middle key? x, y or z which of these is middle. It is x. z is the largest, y is the smallest. If you recall in the previous rotation, we had x, y and z again. In the previous rotation which was the middle key? y, because they were all in a line. z was the top, y was its left child so it means y is less than z. And x was its left child so x is less than y is less than z. So after the rotation we ended up making y as the root. The middle key we ended up making the root. Here also we wanted to do something similar but except that the middle key is now x. So we are going to do a 2 step rotation. That is why it is called a double rotation.

First I will rotate x y. Let see what will happen after I rotate x y. This is what it will look like. x has moved up y has moved down, T_1 remains the left child of y, T_3 remains the right sub tree of x and T_2 switches loyalties from x to y. So earlier it was the left sub tree of x, now it is the right of sub tree of y and T_4 remains as it is. And I have just copied the same height so T_1 has height h+1, T_2 has height h+1, T_3 has height h and T_4 has height h+1. No difference. Is this balanced? Is this height balanced? Height of y is h+2 because both of these T_1 and T_2 have height h+1. Height of x will be h+3, actually now there is an imbalance at x itself. Because y has h+2 and T_2 has h. There is an imbalanced in x and height of z would be h+4 because height of x is h+3 that is one more than that.

So this rotation has not done the job for us yet. We need to do one more rotation. What are the other rotation I need to do? Rotation (x, z). What will happen now? x will go up and z will come down. x will become the parent of z. T_4 was the right sub tree of z, so it will remain the right subtree. y had T_1 and T_2 as its left, so they will remain as they are. And T_3 which was the right sub tree of x now becomes the left sub tree of z, the same thing. Now let us compute heights. Height of y, h+2. Height of z, h+2. Height of x, h+3. Height balance happens, this is balanced in y, its balanced in x and its balanced in z. Further the height of this sub tree is the same as the original sub tree, h+3. So the final tree has the same height as the original tree. Hence we need not go further up the tree.

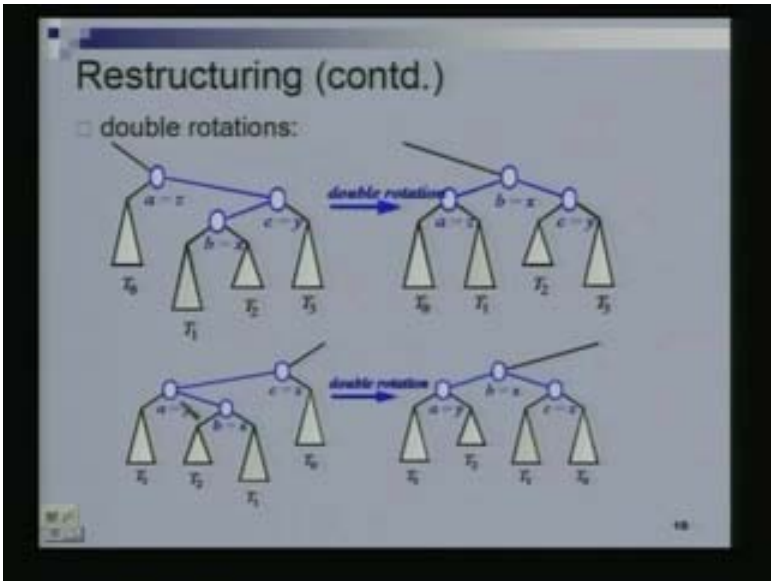
Did you understand the need for the double rotation? We ended up doing the same thing, as I said the middle key ended up being at the top. Because we want to be able to split the thing uniformly. Why was the height imbalance happening? Because x was the middle key, it was coming way down. When I kind of split uniformly the heights reduced and there is a height balanced. It is roughly what is happening here. How much time does the double rotation take? Constant time. So just as a quick recap, we have 4 different ways to rotate nodes in an AVL tree. The single rotation was something like this given in the below slide. There were all in a line x y and z or they were like this x y and z (Refer Slide Time: 33:33). And after rotation this is the picture you get and here after rotation this is the picture you get.

(Refer Slide Time: 33:30)



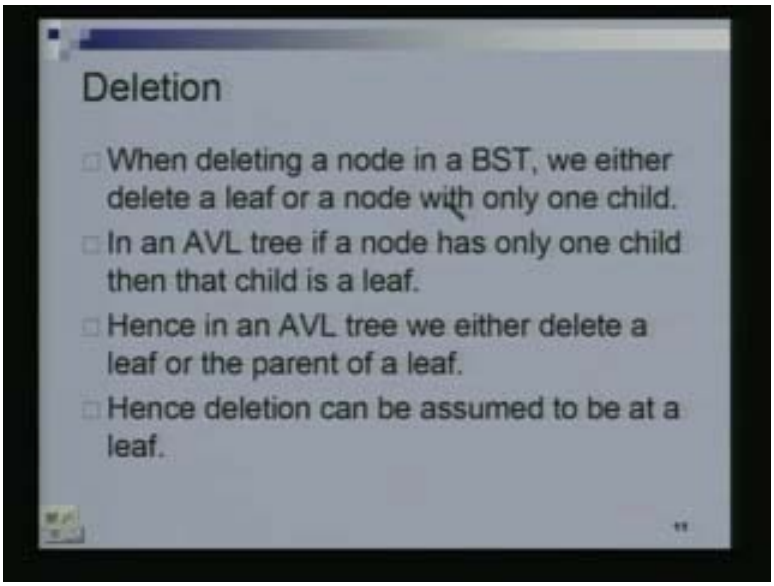
This is just a recap, you understand why we are doing this thing and why this picture is a height balanced picture and we also saw double rotations. So either like this, in which case after rotation you got something like this or it could be like this left and right and in which case again after rotation you got something like that. This is just to show you the picture, you do not have to understand much here. You are hopefully understood why the single and double rotations are done in the way they are done.

(Refer Slide Time: 33:56)



Let us come to deletion because exactly the same principles are going to be used for deletion also. It is a binary tree. The difference between the height, does it become zero? We saw that. Did we see that?

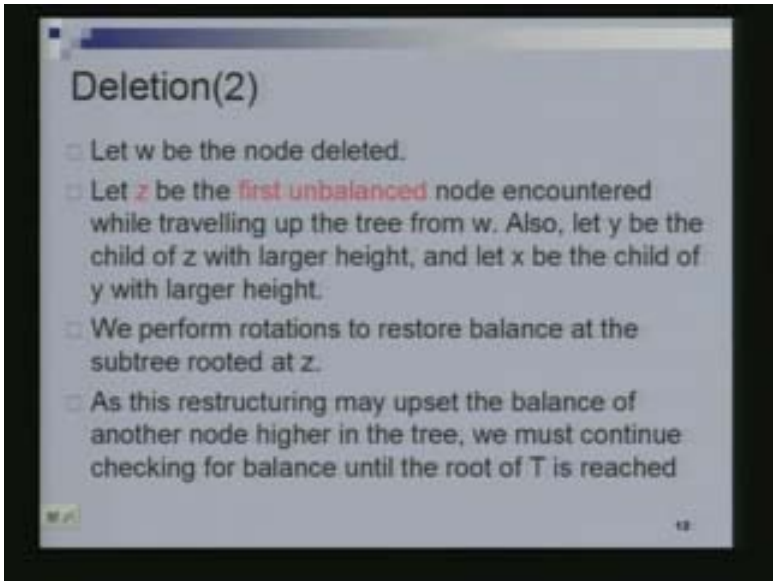
(Refer Slide Time: 37: 50)



Let us see here. Here the difference is zero but in this node z the difference is a one (Refer Slide Time: 34:50). Some nodes there would be a difference of a one, some other nodes there would be a difference of zero. Let us look at deletion. In a binary search tree when I delete a node, we will have 3 cases if you remember. When I am deleting the node which is a leaf or I am deleting a node which is only one child or I am deleting a node which has 2 children. When I delete a node which has 2 children, what did I do? I went to the successor of that node. I copied the content of their successor in to that node and I deleted the successor. The actual node I deleted was the successor node and the successor node has only one child or no children. Why does the successor has only one child? Because it does not have the left child, because if it had the left child then it would not be the successor. It has only one child or it has no children. So the actual node that you end up deleting is either a leaf node or a node with only one child. This is the actual node that you ended up deleting.

What is a node which has only one child? In an AVL tree if I tell you, here is a node which has only one child. What can you say about that node? This is a node with only one child. Can it have another child? Can this node have the child? No, if it had a child like this, then what would be the problem. There would be a height imbalanced. So it cannot have this child or it cannot have the other child, which means this node is a leaf exactly. If in an AVL tree, a node has only one child, then that child is a leaf. What are we saying? When I am deleting in an AVL tree I am either deleting a leaf or I am deleting the parent of a leaf. If I am deleting a node with only one child, then it is a parent of a leaf. Which means that I am essentially deleting a leaf. If I am deleting the parent of a leaf them essentially what am I doing? I can just think of it as if I was deleting the leaf and copying the content of the leaf in to the parent. So I can always think of it as if I am deleting a leaf. So either deleting a leaf or parent, let us just keep that in mind.

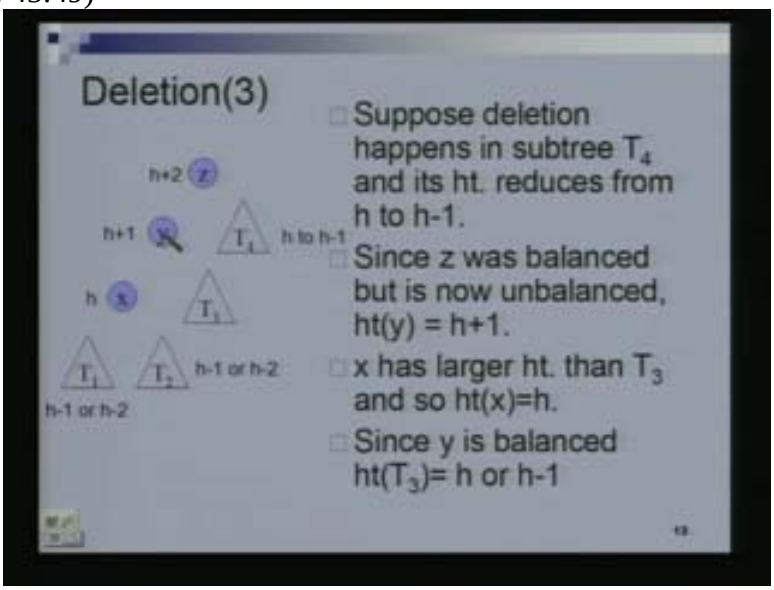
(Refer Slide Time: 39:46)



Let us say w is the node that we are deleting. We are going to define our x y and z slightly in a different way. So z is once again the first unbalanced node that we encountered as we go up from w towards the root. When I deleted w, once again what is going to happen? The ancestors of w, their height could reduce. So one of these ancestors will be unbalanced, if any are unbalanced then one of this will be unbalanced. Let us say z is the first unbalanced node encountered while we are traveling up the tree from w. Now y is not the child of z on the path but we are defining y as child of z with larger height and x is the child of y with larger height. z has 2 children, one of them has a larger height than the other one. So we take that one. Once again we will perform rotations to restore the height balance of the sub tree rooted at z. In the case of insertion what was happening is that once you did this rotation, we did not have to worry any more on the ancestor nodes. Everything was taken care of, we could stop after doing the rotation. In delete what we are going to see is that we might have to continue up and we will see what the reason for that. So you might have to continue up the tree, go to the ancestor of z and once again find the first node which is unbalanced and repeat the rotation there and after that go even further up. Find the first node which is unbalanced, repeat the rotation there and so on till you reach the root.

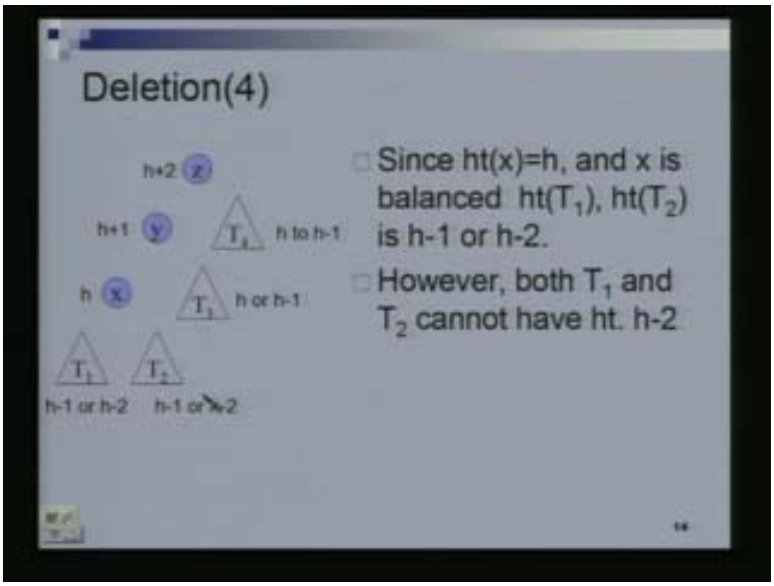
Let us understand what is happening? The x is the child of y with larger height. If both of them have the same height, then we will say which of them should be x. We will say that in a minute. It could happen so this is a valid question. Both the children of y might have the same height. Then which is x? We will see which is x. The two children of z cannot have the same height because that is the imbalanced node. Ignore these h-1 or h-2 for a minute. They should have come at the end. This is the picture which is given below, I have z which is the first unbalanced node. y is the child of z which has larger height and x is the child of y which has larger height. I did a deletion in T_4, I started going up the tree I found a z. Can y be this node here (Refer Slide Time: 42:18)?

(Refer Slide Time: 45:49)



If y was here then its height would have actually decreased. What is a problem? Let see. Why did I draw y to be this? So w is some where here, the node I deleted is some where here. I started walking up and I came to z. And this was a first node I identified which had an imbalance. And then what did I say? This node z has 2 children, this is one child of z and the other child of z is y. Let us take the child of z which has larger height. Why could it not have been this node which has larger height? This is very simple actually.

(Refer Slide Time: 46:32)

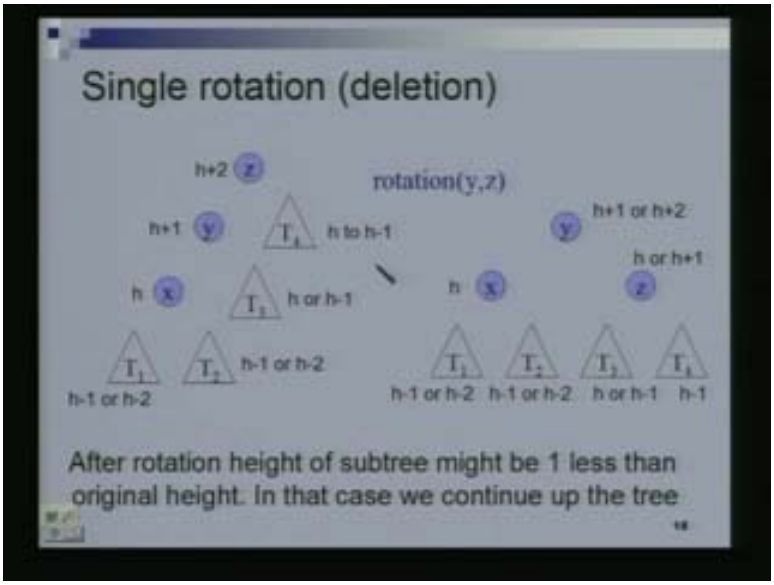


There is an imbalance that happened here. Earlier there was no imbalance. Imbalance happen because the height of this guy decreased (Refer Slide Time: 43:55). If there are 2 things which was balanced and one of them decreased then what can we say about the relationship between

these two things initially? Could this have been smaller than this? No, if this was smaller than this then actually it will become more balanced (Refer Slide Time: 44:20). So this y must have been larger than this for imbalance to have happened. So this is y therefore. The height of T_4 has reduced from h to h-1. What can I say about the height of y? It is h+1. It means that y must be h+1 or h+2. It cannot be h+2 because then originally also it was unbalanced. It cannot be h-3, because then initially it was unbalanced. It has to be h+1. So height of y is h+1. Of those 2, x is the one which has the larger height. The height of x is h. What can I say about of height of T_3? It can be h. Since this y is balanced this can be h or h-1 because y is balanced. So this is what we have argued so far. This goes from h to h-1, z is h+2, y is h+1, x is h, T_3 is h or h-1. So the height of x is h and this is also balanced. So the heights of these two (T_1 , T_2) are h-1 or h-2. One has to have a height of h-1, both cannot have a height of h-2. That is the only thing we can say. Both can have a height of h-1. You cannot say that only one can have a height of h-1. That is a wrong statement, at least one has a height of h-1.

Let us do a rotation now to see what needs to be done. So these are the various heights that have seen which is given in the slide below. This is what we argued in the last 2 slides. What kind of a rotation should I do? I will do a rotation (y, z) once again. Similar to what I did in my insertion. So as a consequence you will have this kind of a picture now (Refer Slide Time: 47:12). y went up, z went down, T_4 and T_3 became the 2 children of z, T_1 and T_2 are the children of x. I have written down the heights h-1 or h-2, h-1 or h-2, h or h-1 and h-1. Because T_4 went from h to h-1, so this is h-1. What is the height of x? This is h-1 or h-2, this is h-1 or h-2 but one of them is at least h-1, so x is h.

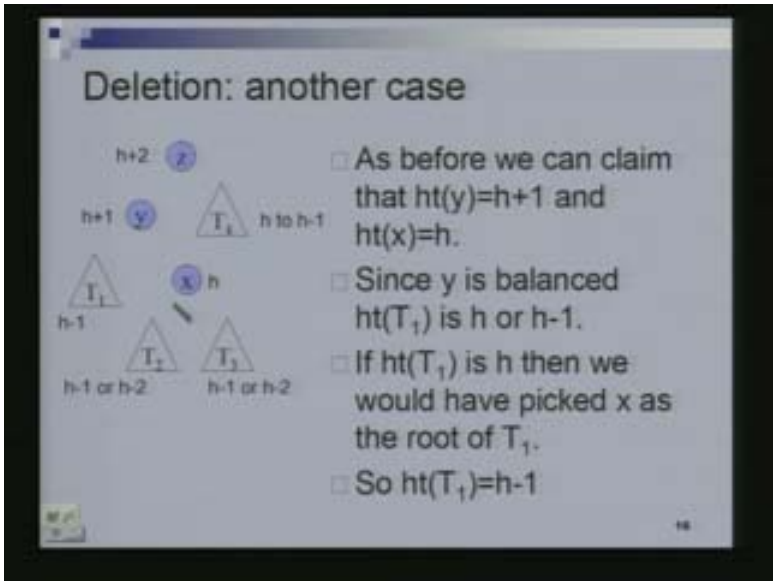
(Refer Slide Time: 48:59)



What is height of z? h or h+1. What is the height of y? h+1 or h+2. What was the original height of this tree? h+2. So if this is h+2 then we are okay, we do not have to continue. But if this is h+1 then we may have to continue because this now becomes the bigger tree (Refer Slide Time:

48:21) (Hindi Conversation). We will have to continue the argument as we go up. The way we said (Hindi Conversation) height has reduced from h to h -1, now we might have to say that this bigger thing height has reduced from h+2 to h+1 and we will have to repeat the argument at the next higher level and so on.

(Refer Slide Time: 51:26)



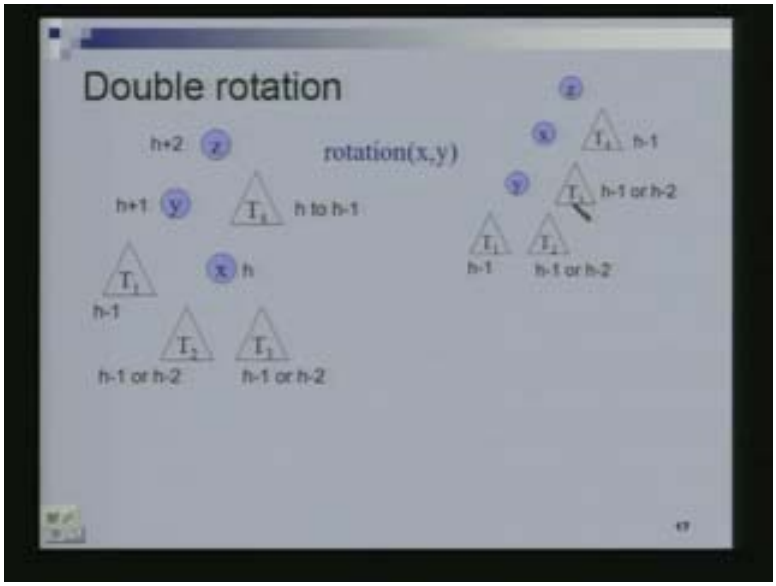
After rotation the height of sub tree might be one less than the original height. In that case we have to continue up the tree. It is might be, you understand? Because it could not have reduced, in which case we can just stop. So this is single rotation, in the case when this was the picture which is given above. y was the left child of z and x was the left child of y.

But we could have this kind of a picture which is given above, that x is the right child of y. So the first part of the argument is the same. This T_4 has gone from h to h-1, so we argued that the height of y is h+1. So height of y is h+1 and height of x is h because x is the one which has larger height and height of z is h+2 because height of y is h+1. This T_4 is of height h or h-1. How about the height of T_1 ? So y is balanced so height of T_1 is either h or h-1. Now if height of T_1 was h then what I would do is, I would pick the root of T_1 as x. So in that case I will be able to do that single rotation of mine, the previous case.

If the height of T_1 is h and the height of x is also h, the same question which he had asked earlier. Which do we pick? Which one will I pick? I will pick the one which will give me the single rotation case. I cannot say that I will pick the left child or the right child. I will pick the left child, if y is the left child of z. If y were a right child of z then I will pick the right child. Since such was not the case, if it was h then I would have picked x as the root of T_1 . So height of T_1 is h-1. Since the height of x was h and these T_1 , T_2 are the same as before h-1 or h-2 for both of them, with one of them at least being h-1. This is our new picture (Refer Slide Time: 51:35). Let me just copy it here. These are the heights of the various nodes and trees. Let us do the double

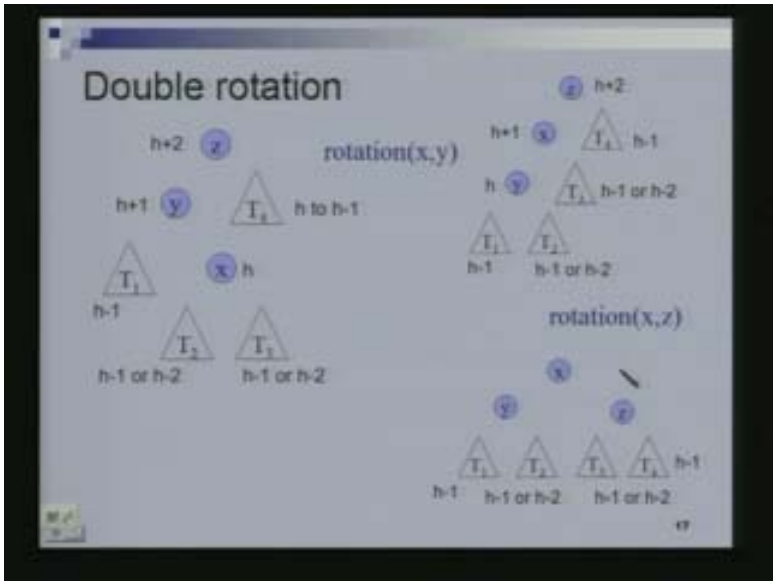
rotation step. So first I am going to rotate as before (x, y), the same process as an insertion essentially.

(Refer Slide Time: 52:03)



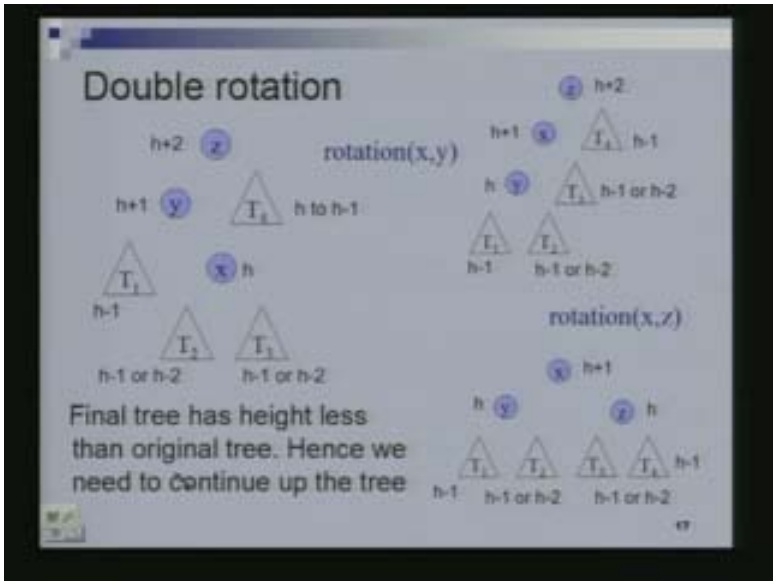
So with the rotation of (x, y) y will come down and x will move up I would get such a picture which is given in the right side of the above slide. T_3 is the right child of x, T_1 and T_2 are the two children of y that is the left and right sub trees of y. The T_1 is h-1, T_2 is h-1 or h-2, T_3 is h-1 or h-2 and T_4 is h-1. What are the heights of these nodes? Height of y is h. What about the height of x? h+1 and there could be an imbalance at x if this where h-2 and height of z is h+2 because x is h+1. And there is an imbalance at this node z, so there could be here at x and there is here at z. There is none here at y. But we are not done so we will now do a rotation around (x, z).

(Refer Slide Time: 52:55)



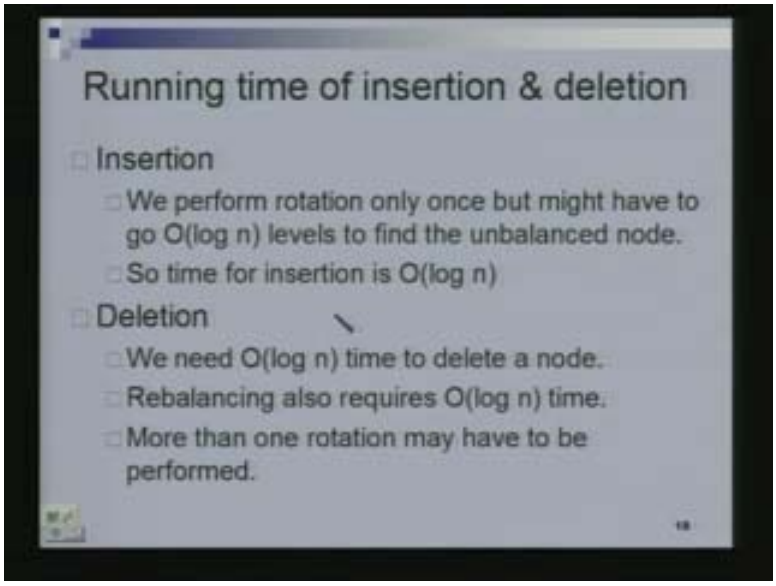
What would happen? x moves up, z moves down , T_3 and T_4 becomes the two children of the z. T_3 has height of h-1 or h-2, T_4 has height of h-1 and T_2 has the height of h-1 or h-2 and T_1 has a height of h-1. What about the height of y? It is h. Is it balanced? Yes, T_1 is h-1 and T_2 is h-1 or h-2. What is the height of z? It is h. Is it balanced? Yes, h-1 and h-1 or h-2. What is the height of x? It is h+1, it is also balanced. But this height is now strictly one less than this (z). (Hindi Conversation) There is no might this time. Why did I make that argument (Hindi Conversation) height h-1 (Hindi Conversation) h we could have done single rotation (Hindi Conversation) and all that thing (Hindi Conversation) You understand y I had to make this kind of an argument? (Hindi Conversation) (Refer Slide Time: 54:00). So hopefully you all understand this.

(Refer Slide Time: 55:12)



What has happened is, final tree has height less than original tree, we need to continue up the tree. You understand the need for continuing up the tree. Because height has reduced by one, as the consequence they could still be imbalanced at the ancestors. Ancestors of this node, that is what ever is this node. This will require a proof even if it is correct, so think about it. It is a good question. He is asking me whether we can just be satisfied by checking the parent of this node. So think about this and we will answer it, may be in the next class or may be after the class.

(Refer Slide Time: 56:38)



Let us quickly look at the running time of insert and delete. So for insertion we spent log n time in finding way to insert. Why log n? Height of the tree, we actually spent time proportional to the height of the tree which is the height we argued in the last class that is log n. So we spent log n time coming down then we spent log n time may be moving up. At most log n because that is the height and then we spent constant time in doing a rotation and one rotation and we are done. The entire thing is only log n.

Deletion, recall that in insertion you will first find the node in the binary search tree whether the insertion has to be done. You will insert the node then you will start moving up the tree to find the place where the imbalance occurs. The first place and then in that place we said we will just do a rotation and with one rotation you will be able to satisfy the height balance property once again. Insertion basically requires order log n time to insert the node and you might have to spend order log n time to move up and a constant time to do the rotation. So in all it just takes a log n order time. Deletion on the other hand also requires only order log n time but we need to do a little bit more work. The reason for that is to delete a node, recall that you have to identify which of those 3 cases the node is does in. Whether the node you are deleting is a leaf node or if it has only one child or if it has 2 children then we need to find the successor of the node. We need to go right and keep going left, find the successor, swap contents and then delete the successor node.

Once you have deleted, now you have to move up the tree to find the first place where the imbalance occurs. Having found that you do a rotation, that rotation may or may not solve your problem. If it does not solve the problem of height balanced, it does not restore height balance then you might have to continue up from that from that node. And may be once again perform a rotation, if that is also the problem then you stop otherwise you will have to continue up. So in all the number of rotations you might require is as large as the height of the tree. Because with every rotation you are moving one level up. You might require as many as order log n rotations, but each of those rotations only taking a constant time. The total time required for all these rotations put together is only order log n. And we took order log n time to delete the node. All the rotations also took order log n time. The total time required for the entire delete operation is still order log n. So with that we will end today's class. We saw how to do an insertion and deletion in AVL trees. We argued eventually that the total time taken for both insertion and deletion is only order log n. In the last class we had seen that the time taken for search is also only order log n in the case of an AVL tree. So all the 3 operations of insert, search and delete can be done in log n time in an AVL tree.