

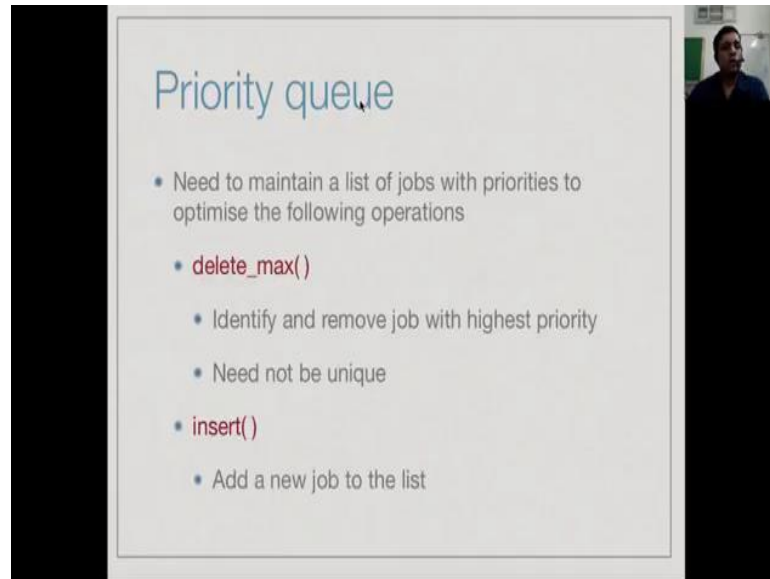
Let us now look at Heaps.

(Refer Slide Time: 00:03)

Module – 05

Lecture - 35

Heaps

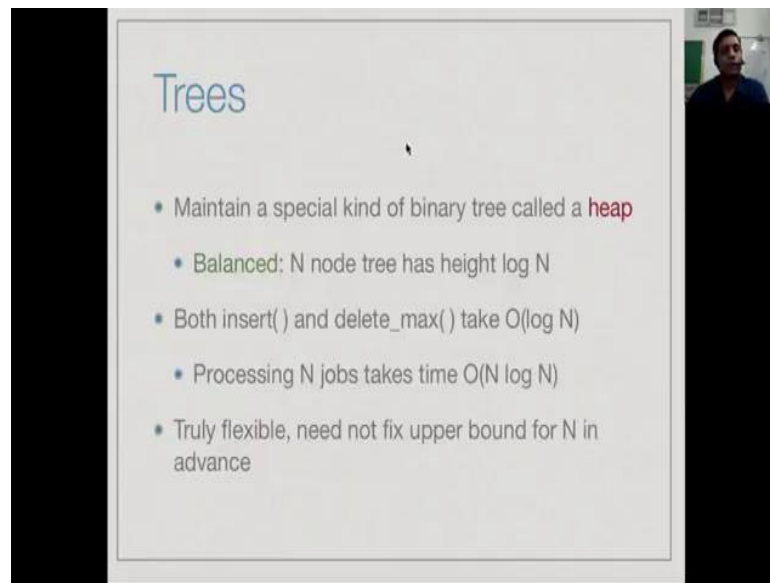


The slide is titled "Priority queue" in a light blue font. It contains a bulleted list of operations and their descriptions. The first bullet point is "Need to maintain a list of jobs with priorities to optimise the following operations". The second bullet point is "delete_max()", which is followed by a sub-bullet "Identify and remove job with highest priority". The third bullet point is "insert()", followed by a sub-bullet "Add a new job to the list". A small video inset of a man is visible in the top right corner of the slide.

- Need to maintain a list of jobs with priorities to optimise the following operations
 - `delete_max()`
 - Identify and remove job with highest priority
 - Need not be unique
- `insert()`
 - Add a new job to the list

So, recall that our goal is to implement a priority queue. In a priority queue, we have a sequence of jobs that keeps entering the system, each job has a priority. Whenever, we are ready to schedule a job to execute, we must pick up not the latest job or the earliest job that we got, but the job which currently has the highest priority among the waiting jobs. Therefore, we need an operation called delete max which will search for the highest priority job among those that are pending and schedule it next. And we obviously, have an insert operation which adds these jobs dynamically as they arrive.

(Refer Slide Time: 00:37)



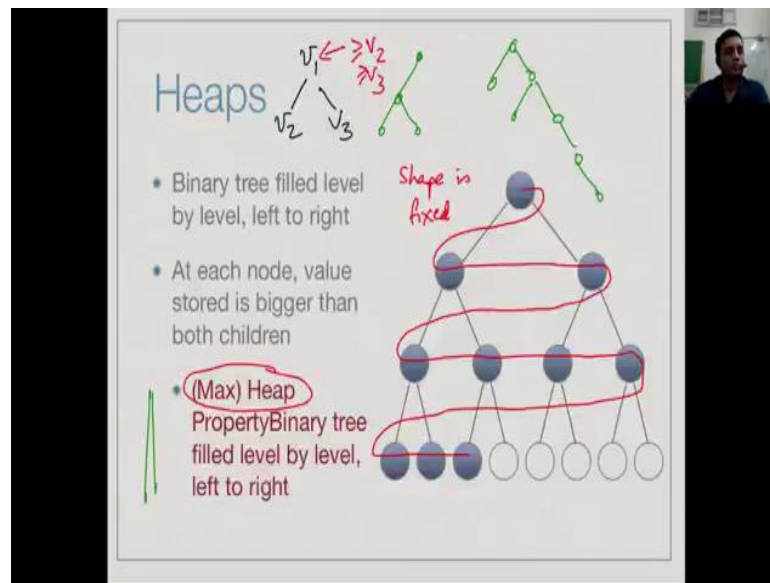
Trees

- Maintain a special kind of binary tree called a **heap**
- **Balanced**: N node tree has height $\log N$
- Both `insert()` and `delete_max()` take $O(\log N)$
- Processing N jobs takes time $O(N \log N)$
- Truly flexible, need not fix upper bound for N in advance

So, we saw last time that a linear structure will not allow us to simultaneously optimize these two. We end up with an order N operation for delete max or an order N operation for insert. Then, we saw trivial two dimensional array which gives us an N root N solution that is the root N operation for each of these, so our N operations is order N to N . But, we said that we will find a much better data structure using a tree of a special type called a heap.

So, the heap is going to be a balance tree whose height is logarithmic in the size that is if N nodes in the tree, the height that is the number of edges from the root to any leaf will be $\log N$. And with this, it will turn out the both insert and delete max are prepositional to $\log N$ and therefore, processing N jobs will take time $N \log N$ as supposed to N root N for the array or N square for the linear representation. We also said that this heap in principle is flexible and can grow as large as we want. So, we do not have to fix in advance the size of the heap that we need to keep.

(Refer Slide Time: 01:39)

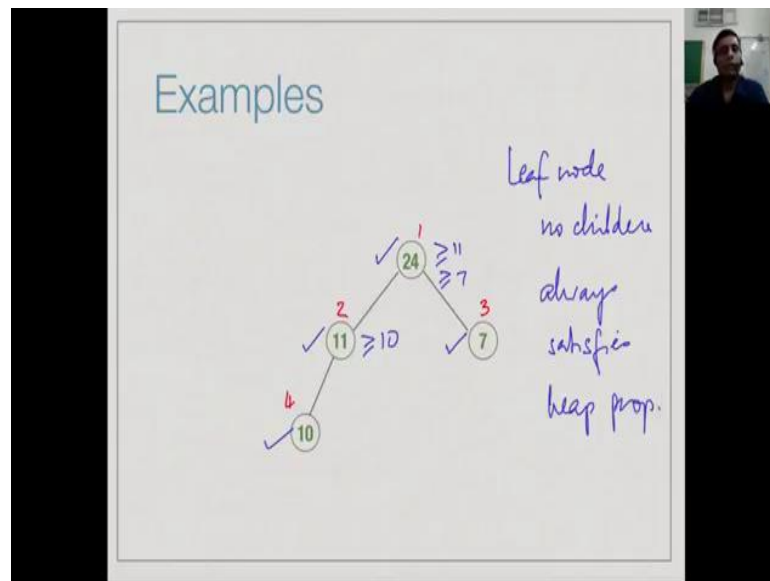


So, let us start looking first at what a heap test. So, a binary tree is a tree where we have a root and every node has 0, 1 or 2 children. So, binary trees in general can have arbitrary shapes. So, we could have binary trees which look like this, where the root has 1 child, this has 2 children or it could look even more skewed in one direction. So, binary trees can have very strange shapes, but a heap is a binary tree which has a very specific shape, where we fill up the tree nodes or we add the tree nodes in a specific order.

So, first we start at the root, then we must add the left child of the root, then the right child and this way keep going level by level left to right. So, we add this node, then we add this node, then we add this node. So, once I know how many nodes are there in the tree, I know precisely what the shape is, so the shape is fixed. So, that is the first feature of the heap that if I have a heap with n nodes, then the shape of the tree is deterministically fixed by this rule that the n nodes must be inserted top to bottom, left to right, then we have a value property.

So, the value property says that... So, what does happening in the tree is that we have nodes and each nodes is the value, so whenever I see a node with value v_1 which has children v_2 and v_3 , then what we want is, this is bigger than or equal to v_2 and bigger than or equal to v_3 . So, among these three nodes the largest one must be v_1 , so this is what is called the max heap property. So, this is the local property, it just tells us at every node look at that node, look at the 2 children, the node must be bigger than it is 2 children.

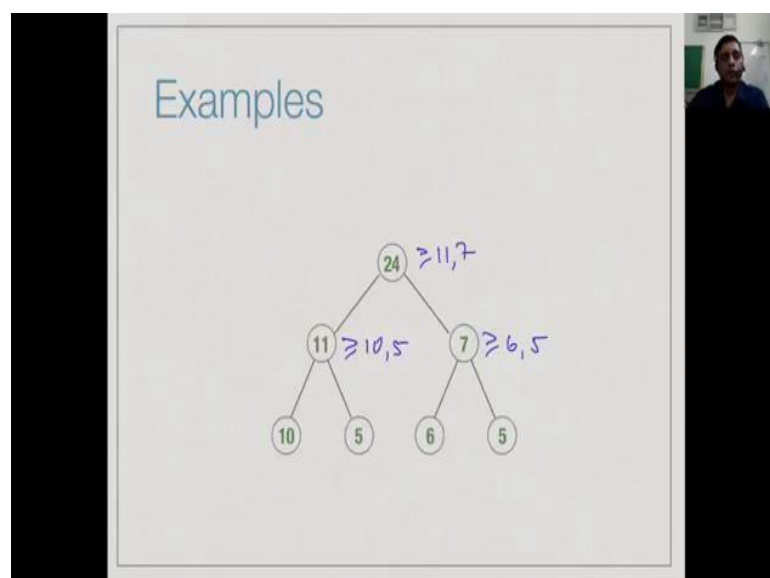
(Refer Slide Time: 03:23)



So, here is an example of the heap with 4 nodes, so first because it is 4 nodes, every 4 node heap will have the shape. Because, the first node will be the root, the second will be the roots left child, third node will be the right child and the fourth node will start a new line, then more over we can check the heap property. So, we see the 24 is bigger than 11, 24 is bigger than 7. So, this is a valid node for a heap property, 11 is bigger than 10 there is no right child, so this is a valid heap, there is no child of 7 at all.

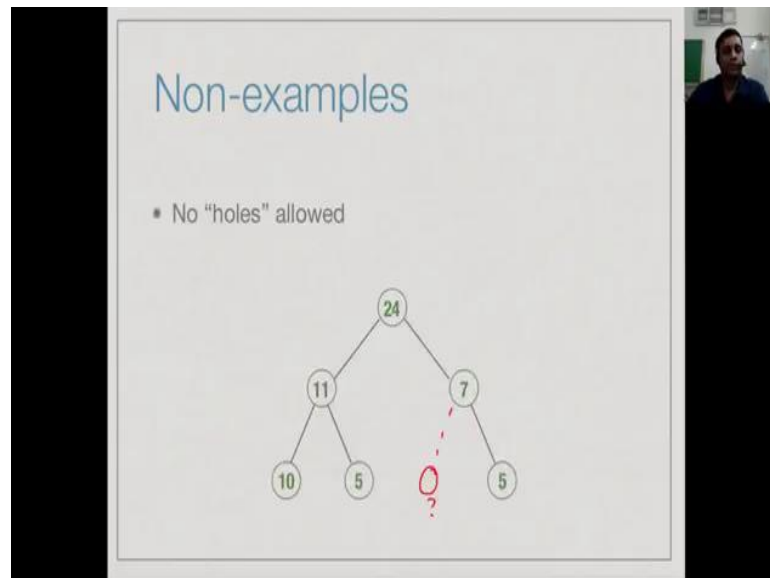
So, by trivially this is a valid heap node and 10 is the valid heap node for the same reason. So, every leaf node which has no children always satisfies the heap property. So, once you have a leaf node, then nothing to check.

(Refer Slide Time: 04:16)



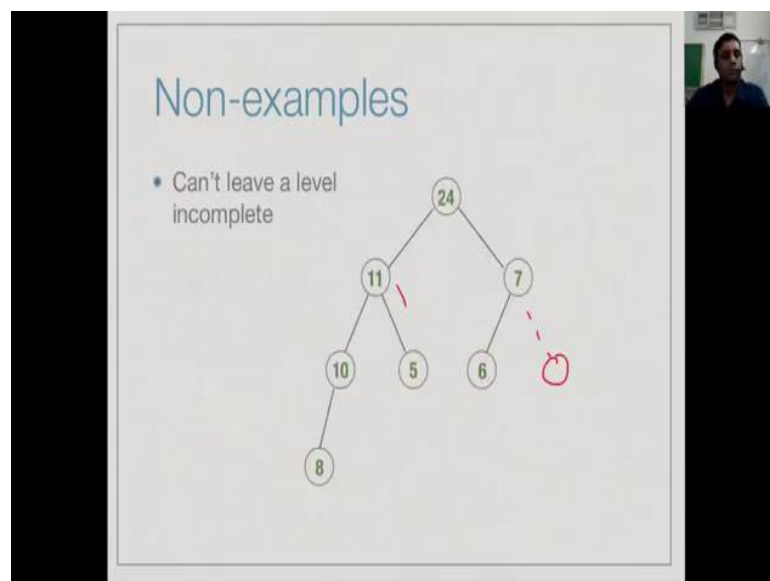
So, here is another heap, this one has 7 nodes, so again the shape is fixed and again you can check that this is bigger than 11 and 7, 11 is bigger than 10 and 5 and 7 is bigger than 6 and 5 and the rest are all leaf node, so there is no problem. So, these are two examples of heaps. So, what is an example of something that is not a heap?

(Refer Slide Time: 04:35)



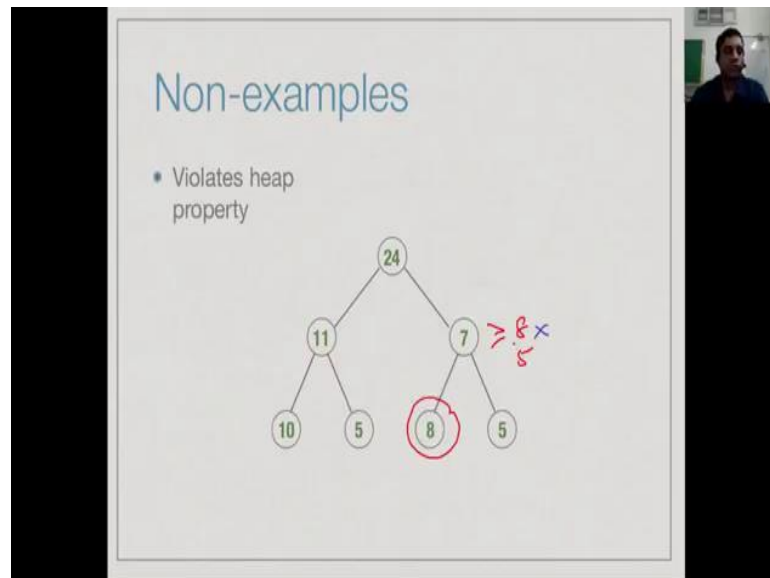
So, here we have something which is not a heap, because the structure is wrong. So, we said that you cannot leave holes, you must go top to bottom left to right, so there should be some node here, before you add the node in the right. So, where is this node? This node is missing, so this structure is not right.

(Refer Slide Time: 04:51)



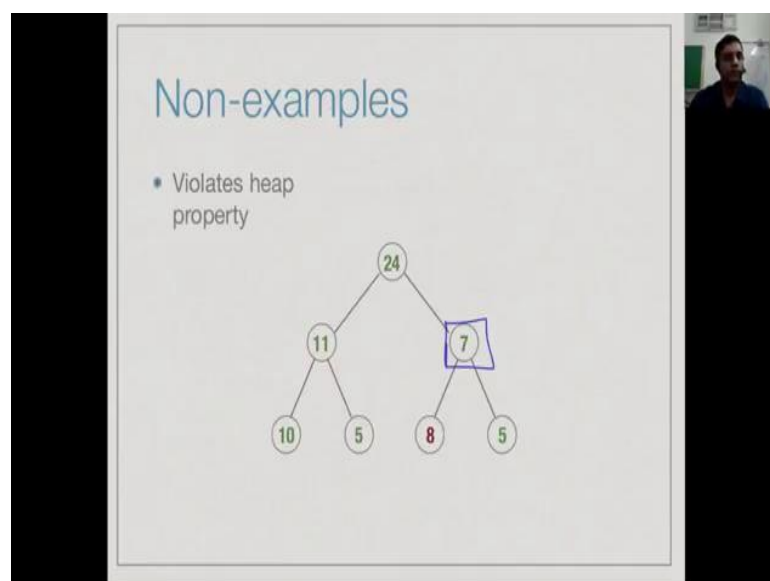
For the same reason, this structure is also not correct, because we have here something which is missing, a node at this level and we started a new line. So, both of these are not a leaf for structural reasons.

(Refer Slide Time: 05:04)



Here on the other hand, we saw something which is a valid structure, in fact we saw heap before which has the structure, the problem is with this node. So, we want 7 to be bigger than 8 and 5, but this is of course, not case. 7 is not bigger than 8, 7 is smaller than 8.

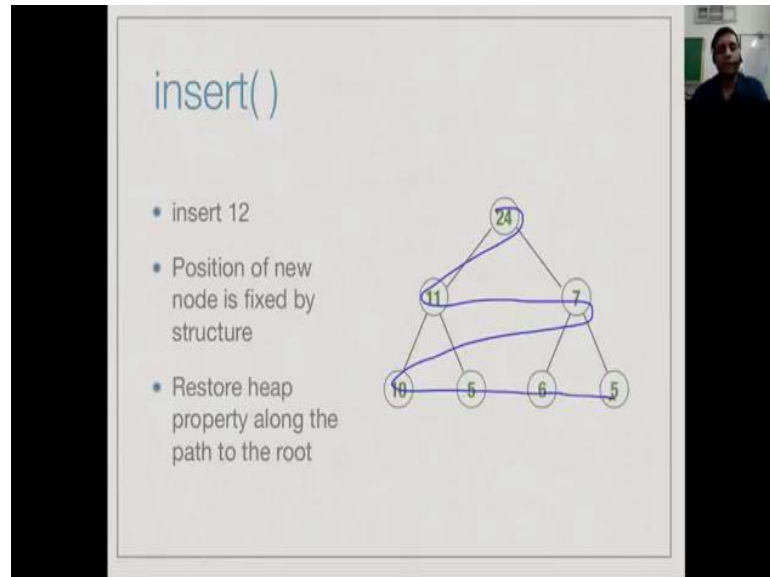
(Refer Slide Time: 05:20)



So, this node 8 actually violates the heap property, so something can fail to be a heap, either because the tree structure is wrong or because at some node the heap property is

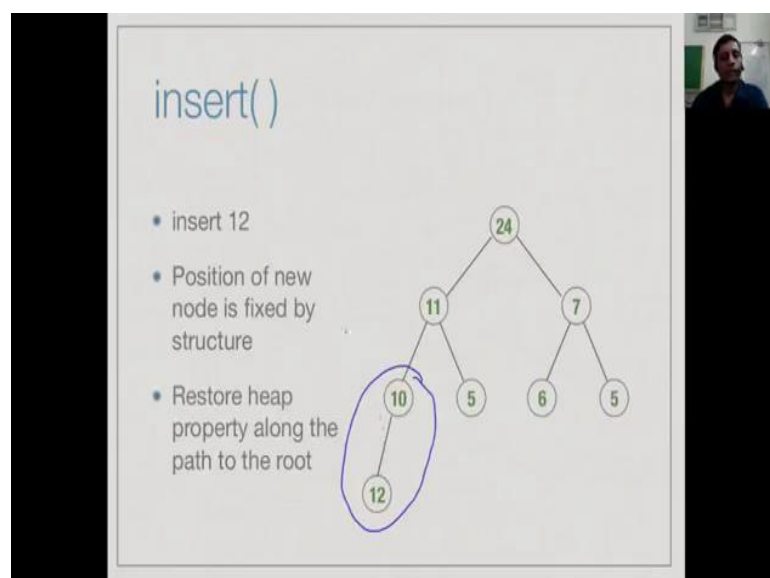
violated. In this case, node that violates the heap property is 7, because one of its children are actually bigger than the node itself.

(Refer Slide Time: 05:37)



So, now we have to implement these two operations on heaps, insert and delete max. So, let us see how it works? So, first let us insert 12, so insert 12 means I have to add a value to the heap. So, the first thing when I add a value to the heap is I must expand a tree. So, where do I put this node? So, this now fixed because we know that heaps can only grow and shrink in a particular way, so I must add the new node left to right, top to bottom. So, in this case if I go left to right, top to bottom I come to this point, now I cannot add anything more, so it must come at the next level.

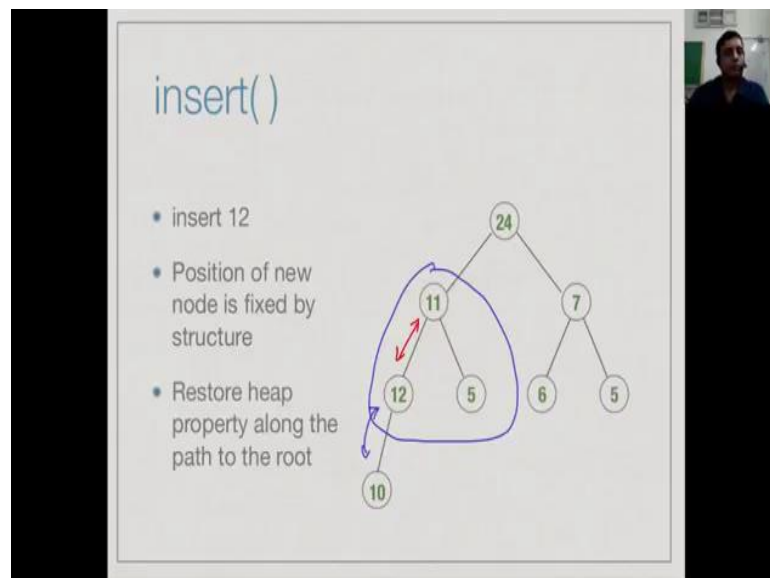
(Refer Slide Time: 06:16)



So, the first thing is that this is where the new node must come to contain 12. Now, if I put 12 into this position, the problem is that I might not have the heap properties satisfy. In this case, you can see that the heap property actually face right here, because 12 is bigger than it is parent. So, 10 violates the heap property, but notice that this can only happen above, so what I mean is that when you insert something it is a leaf.

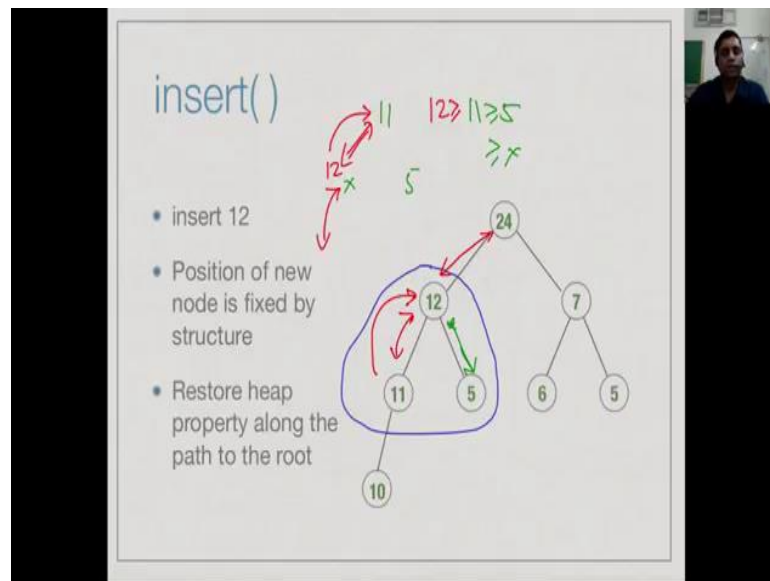
So, when you since 12 is a leaf, it cannot fail the heap property below 12, because there is no child below 12. If at all the property fails, it is because the parent of the new node has a value which is too small. So, there is a simple way of fix this locally at least and that is to exchange these two.

(Refer Slide Time: 06:57)



So, I exchange this 12 and 10 and now I fix the problem here, but now I change the value at this point. So, I have to look at this configuration to see whether what I move the 12 into violates heap property or not and here again you can see that there is a problem because 12 is still bigger than it is parent.

(Refer Slide Time: 07:20)

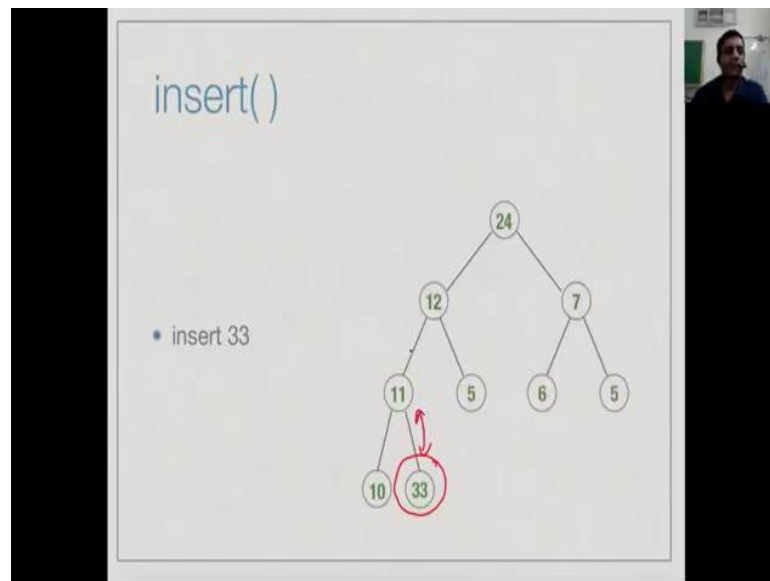


So, then I exchange that, so now I exchange this, now in the process what is happened is that well was gone from here to here. And now, the first thing we have to convince ourselves is that the heap that we local heap property that we just fixed does a need any further fixing. In other words, we have to guaranty that there is no problem and the other side between 12 and 5 and this cannot happen, because we originally had 11, 5 and something here, we know that 11 was bigger than 5 and it to bigger than the something, then we did and exchange and be bought at 12 here.

So, if at all the problems only would be 11 and 12 and since if there is a violation well wish bigger than a 11 that is wide is the violation, but 11 is node to be bigger than the other side. So, if I move 12 to the top of this three node structure, it cannot be smaller than other side. Because, the root currently is already bigger than the other side, the only reason I move 12 of this because it is still bigger than that was ((Refer Time: 08:18)). So, then in an actual when I do and upward swap like this I can be sure that do not have to look on the other side I just keep looking up.

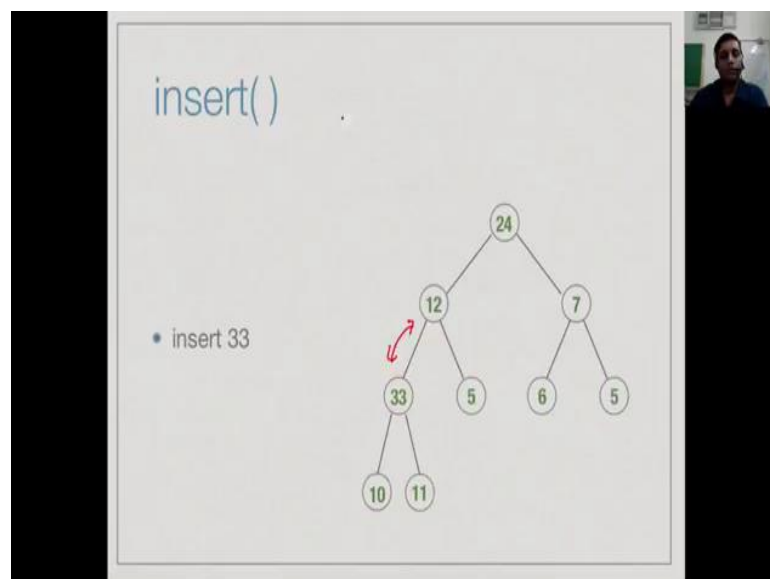
So, now I got 12 up to this point, so now I need to check whether there is any further violation and it turns out the 12 is smaller than 24. So, I can stop, so this was the result of inserting 12 into this heap.

(Refer Slide Time: 08:41)



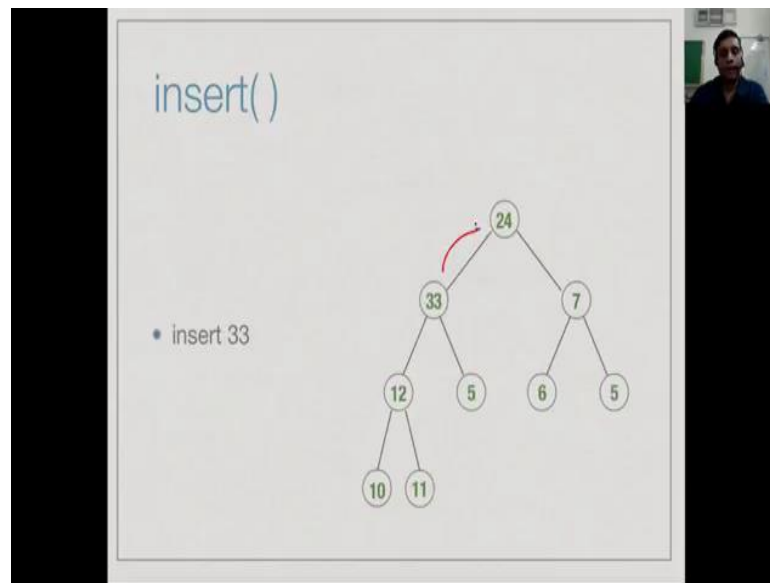
Now, I can insert a new node 33 just to convince as we know how to do, so as before if I insert 33. So, this was the result of the heap after 12 and now I have inserted a 33 here, so it has to be the right because that is a next node in the structure and I put the value, but it violates the heap property would be 11 and 33, so I swap it up.

(Refer Slide Time: 09:00)



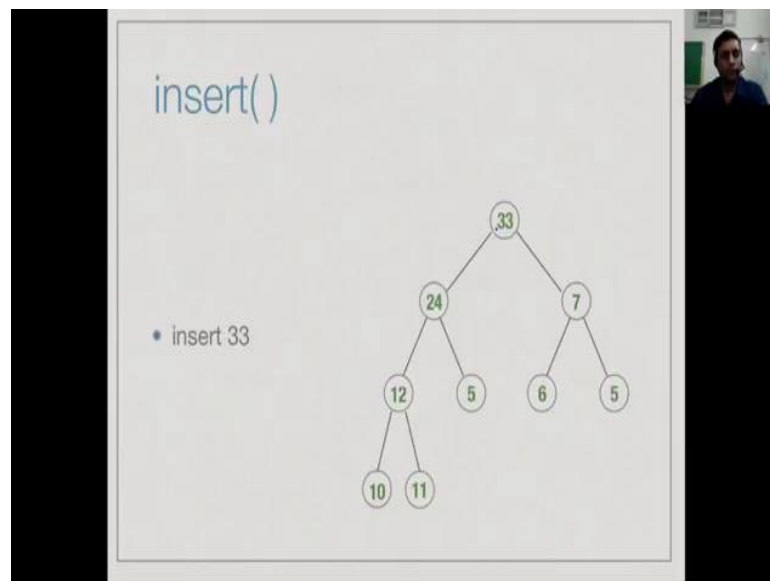
And then again I have a violation here, so I swap it up again.

(Refer Slide Time: 09:06)



Again I have a violation here, so I swap it up again.

(Refer Slide Time: 09:07)



And now I reach the root, so actually it is become the biggest node and now I can stop.