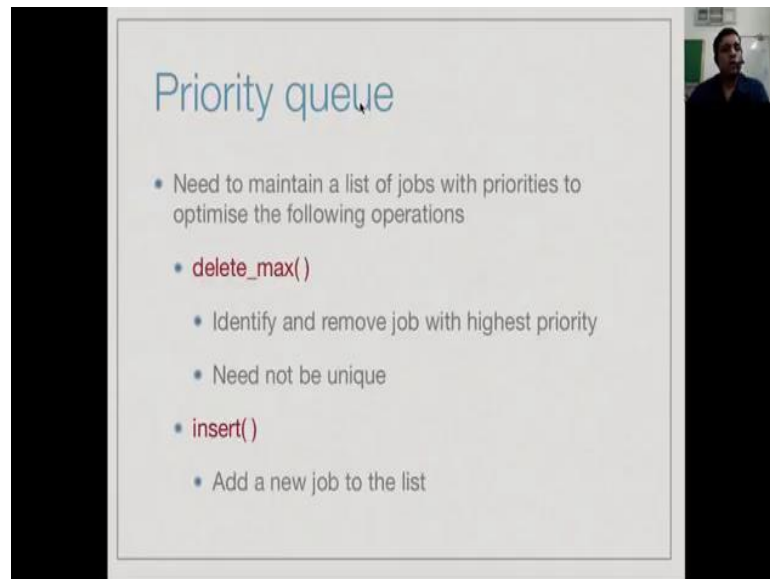


**Design and Analysis of Algorithms**  
**Prof. Madhavan Mukund**  
**Chennai Mathematical Institute**

**Module – 05**  
**Lecture - 35**  
**Heaps**

Let us now look at Heaps.

(Refer Slide Time: 00:03)

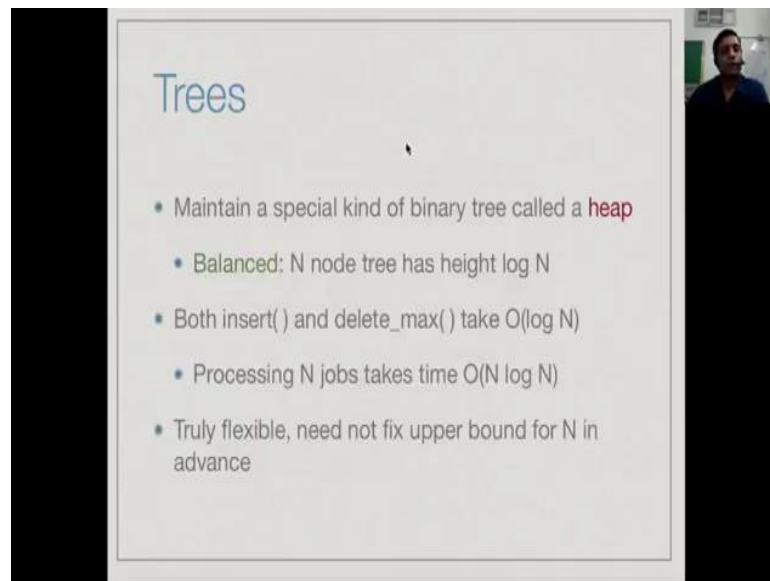


The slide is titled "Priority queue" in a light blue font. It contains a bulleted list of operations and their descriptions. The first bullet point is "Need to maintain a list of jobs with priorities to optimise the following operations". The second bullet point is "delete\_max()", which is followed 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 in the top right corner shows a man speaking.

- Need to maintain a list of jobs with priorities to optimise the following operations
  - `delete_max()`
    - Identify and remove job with highest priority
  - `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)



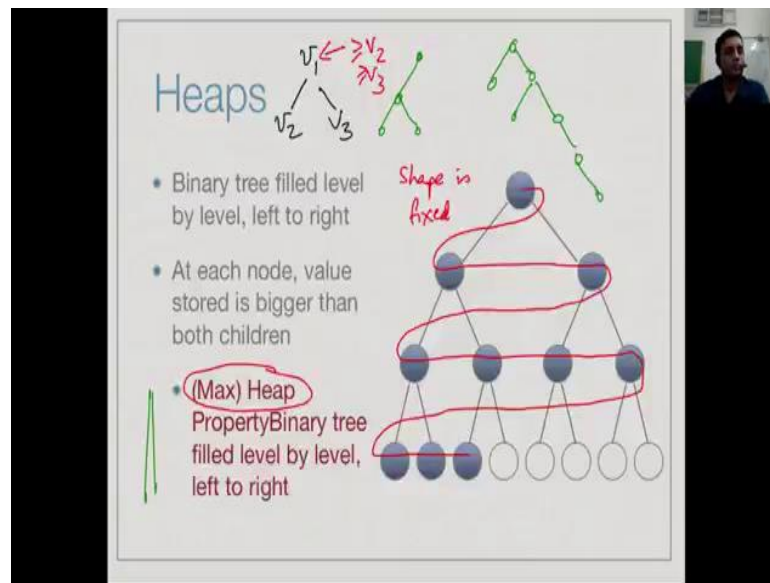
## Trees

- 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)

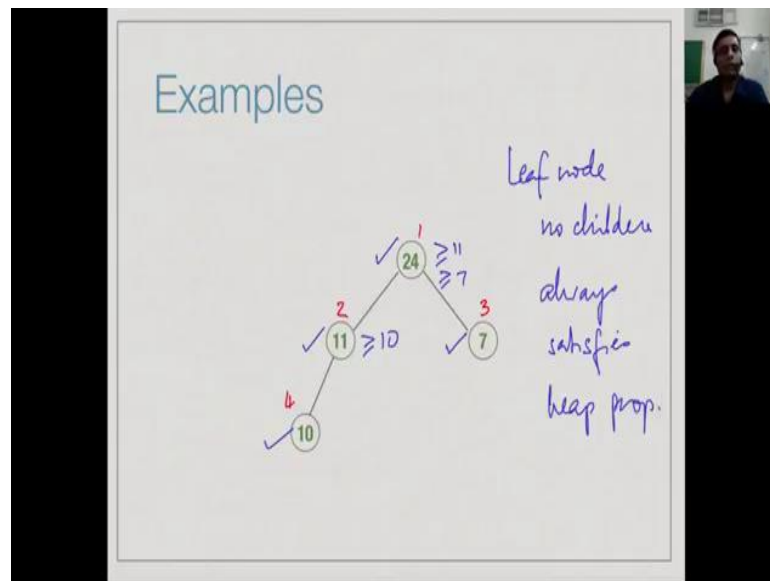


So, let us start looking first at what a heap test. So, a binary tree is a tree where we have a root and every node has 0, 1 or 2 children. So, binary trees in general can have arbitrary shapes. So, we could have binary trees which look like this, where the root has 1 child, this has 2 children or it 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.

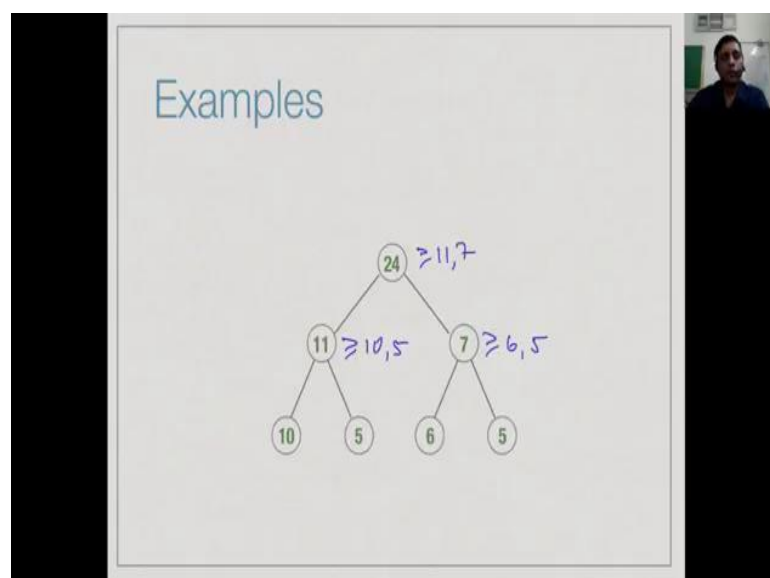
(Refer Slide Time: 03:23)



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.

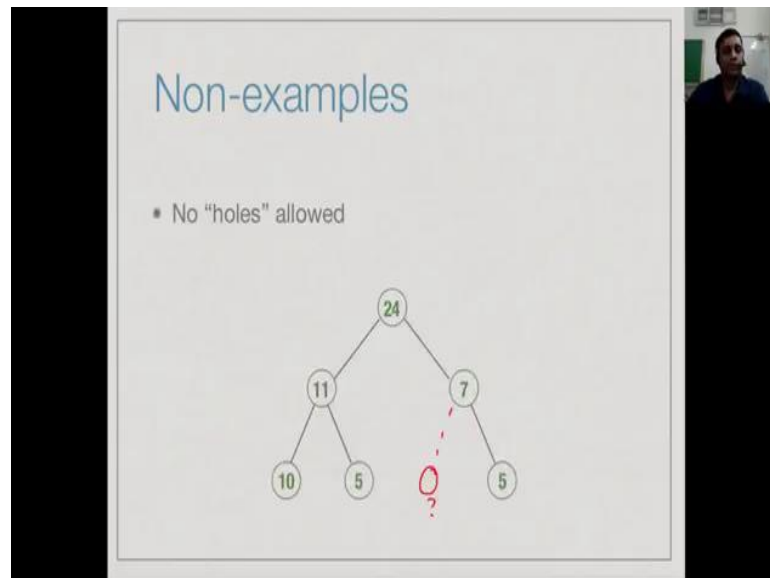
So, by trivially this is a valid heap node and 10 is the valid heap node for the same reason. So, every leaf node which has no children always satisfies the heap property. So, once you have a leaf node, then nothing to check.

(Refer Slide Time: 04:16)



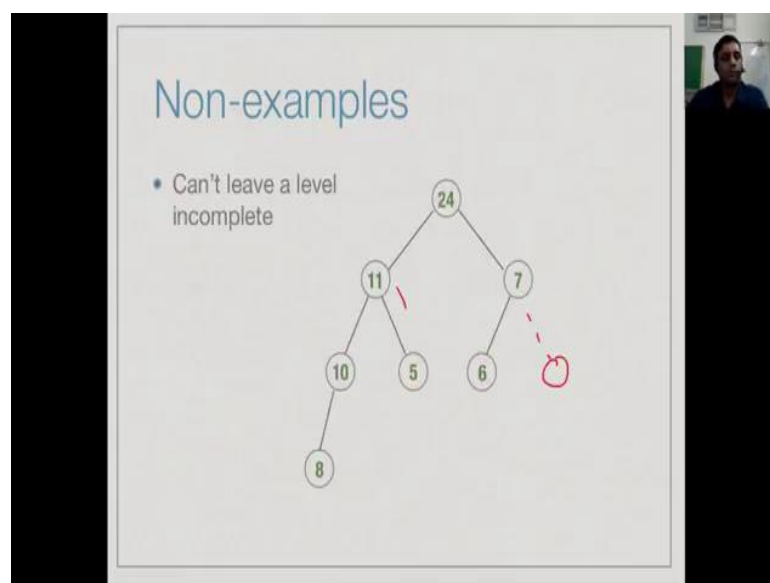
So, here is another heap, this one has 7 nodes, so again the shape is fixed and again you can check that this is bigger than 11 and 7, 11 is bigger than 10 and 5 and 7 is bigger than 6 and 5 and the rest are all leaf node, so there is no problem. So, these are two examples of heaps. So, what is an example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

(Refer Slide Time: 04:35)



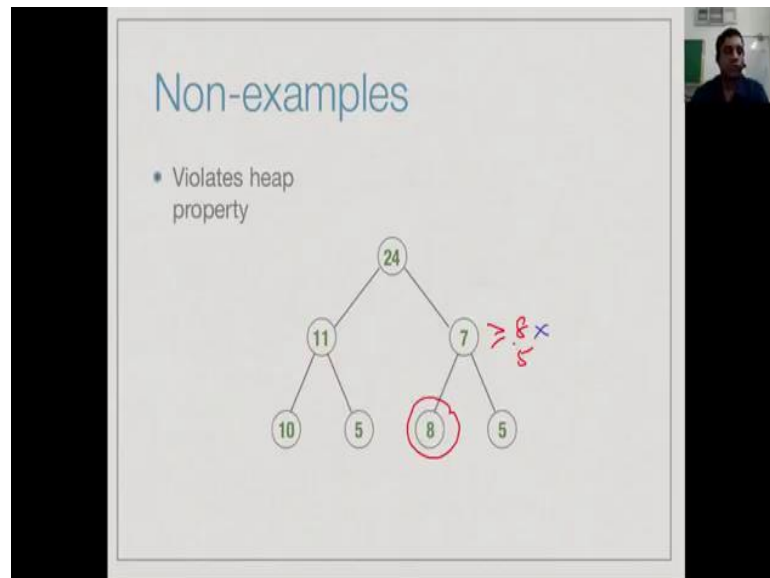
So, here we have something which is not a heap, because the structure is wrong. So, we said that you cannot leave holes, you must go top to bottom left to right, so there should be some node here, before you add the node in the right. So, where is this node? This node is missing, so this structure is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

(Refer Slide Time: 04:51)



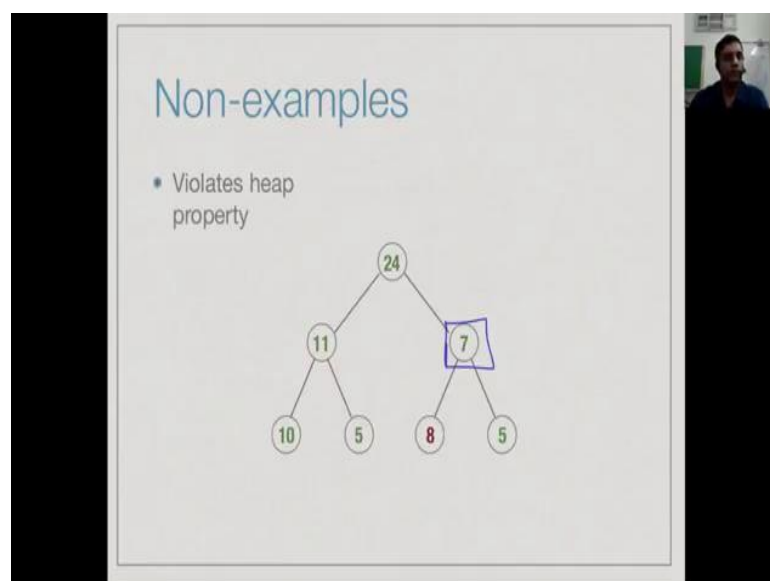
For the same reason, this structure is also not correct, because we have here something which is missing, a node at this level and we started a new line. So, both of these are not a leaf for structural reasons.

(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.

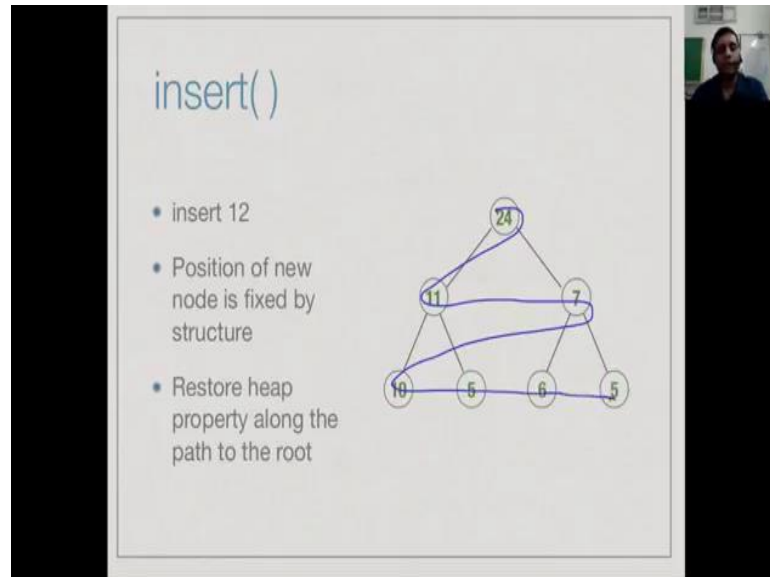
(Refer Slide Time: 05:20)



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

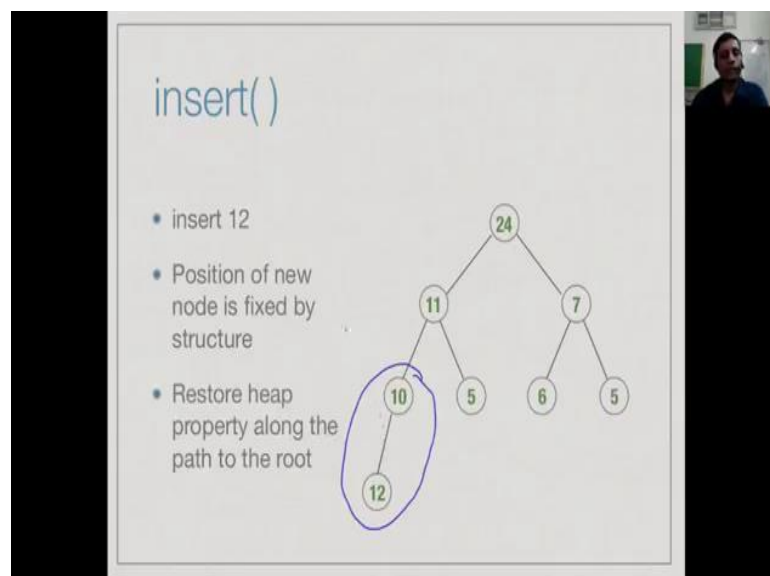
violated. In this case, node that violates the heap property is 7, because one of its children are actually bigger than the node itself.

(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.

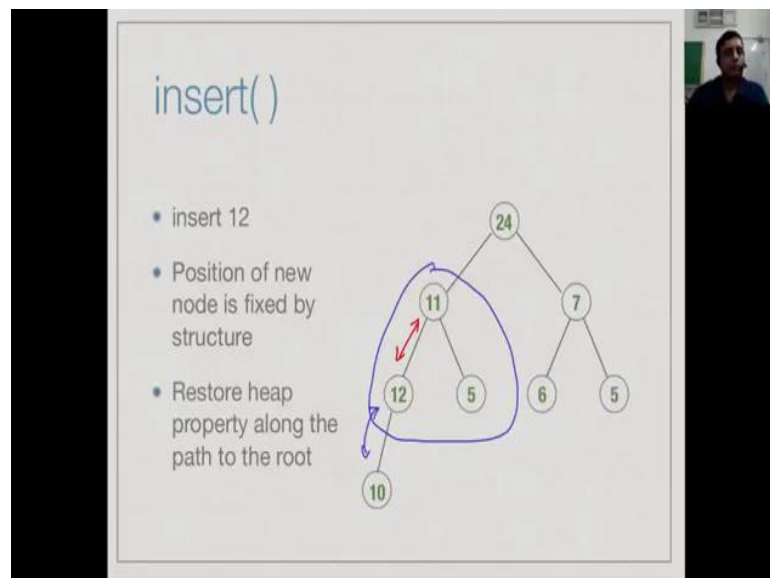
(Refer Slide Time: 06:16)



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.

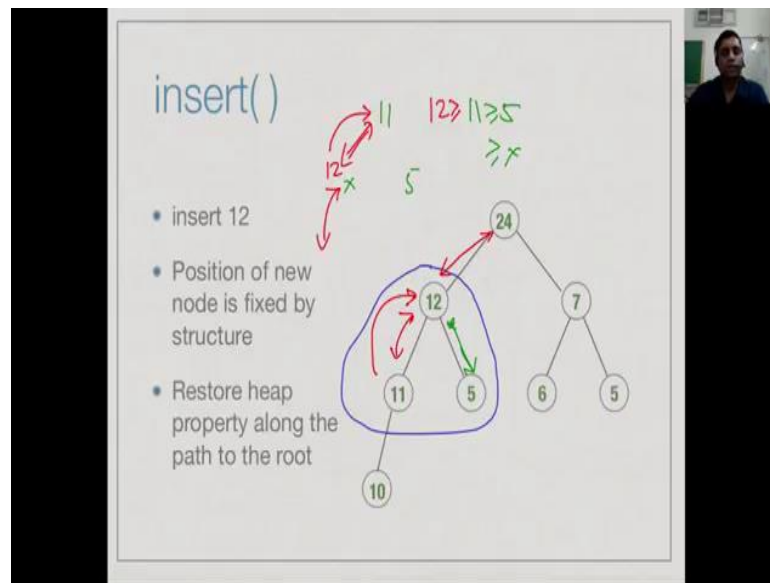
(Refer Slide Time: 06:57)



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)

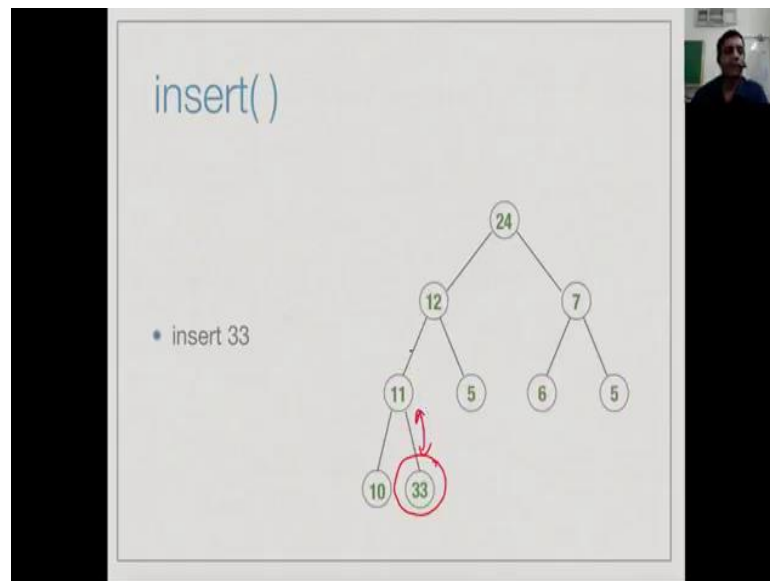


So, then I exchange that, so now I exchange this, now in the process what is happened is that well was gone from here to here. And now, the first thing we have to convince ourselves is that the heap that we local heap property that we just fixed does a need any further fixing. In other words, we have 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.

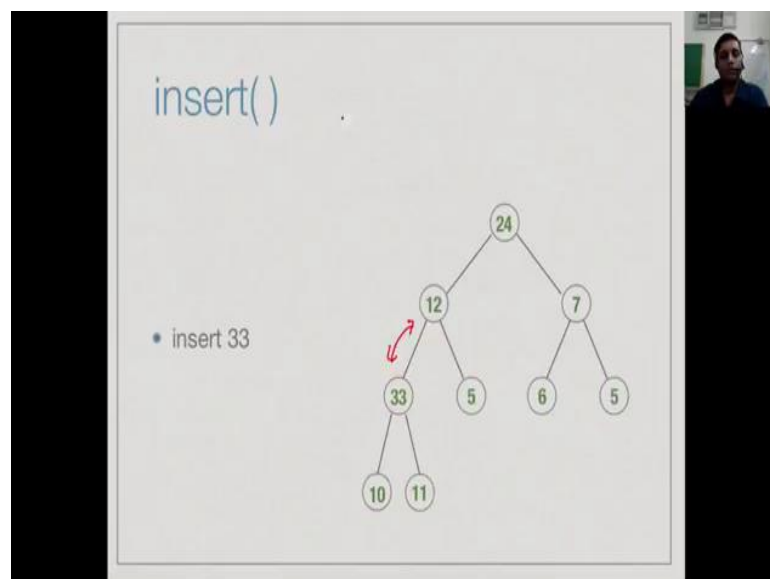
So, now I got 12 up to this point, so now I need to check whether there is any further violation and it turns out the 12 is smaller than 24. So, I can stop, so this was the result of inserting 12 into this heap.

(Refer Slide Time: 08:41)



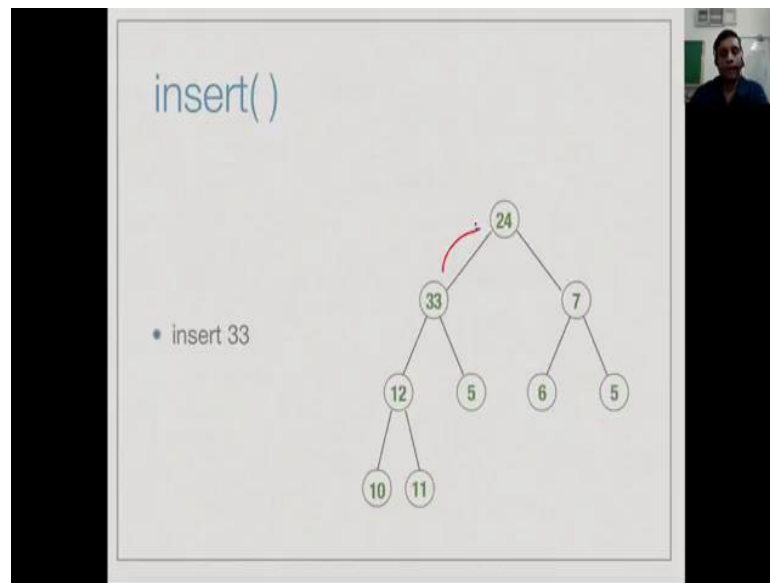
Now, I can insert a new node 33 just to convince as we know how to do, so as before if I insert 33. So, this was the result of the heap after 12 and now I have inserted a 33 here, so it has to be the right because that is a next node in the structure and I put the value, but it violates the heap property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

(Refer Slide Time: 09:00)



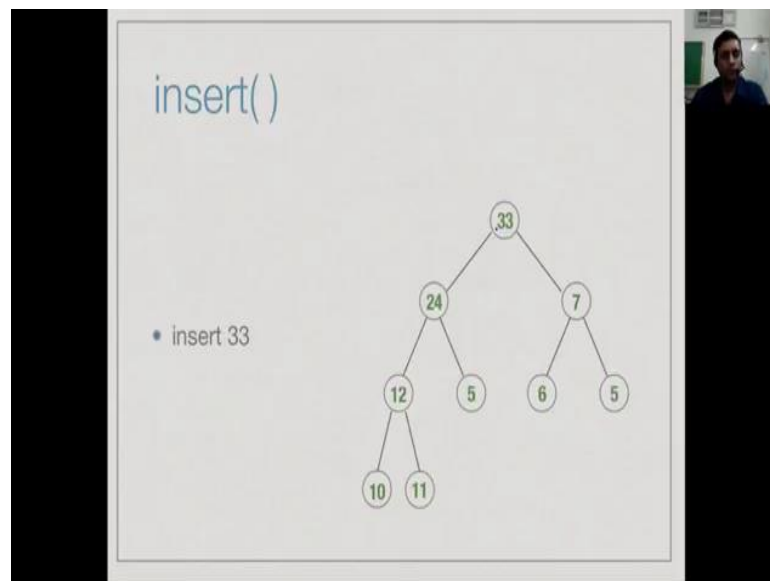
And then again I have a violation here, so I swap it up again.

(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.

(Refer Slide Time: 09:15)

**Complexity of insert()**

- Need to walk up from the leaf to the root
- Height of the tree
- Number of nodes at level  $0, 1, \dots, i$  is  $2^0, 2^1, \dots, 2^i$
- K levels filled :  $2^0 + 2^1 + \dots + 2^{k-1} = 2^k - 1$  nodes
- N nodes : number of levels at most  $\log N + 1$
- insert() takes time  $O(\log N)$

Handwritten notes:   
 - Level labels: lev 0, lev 1, lev 2   
 - Tree diagram with nodes labeled  $2^0, 2^1, 2^2$    
 - Red underlines under  $2^0, 2^1, \dots, 2^i$    
 - Red  $0, 1, \dots, k-1$    
 - Red  $k$  with a bracket   
 - Red  $|||||$

So, how long does this take? So, every time we saw every time we do an insert, we start at a leaf node a new leaf node that we create and we walk up to the root. So, the worst case of such a thing it depend on the worst case height of the tree, we have to bound the height of the tree, the height of the tree by definition if I have a tree like this. So, the height of the tree is a longest search path, the length of the longest path from the root to them off.

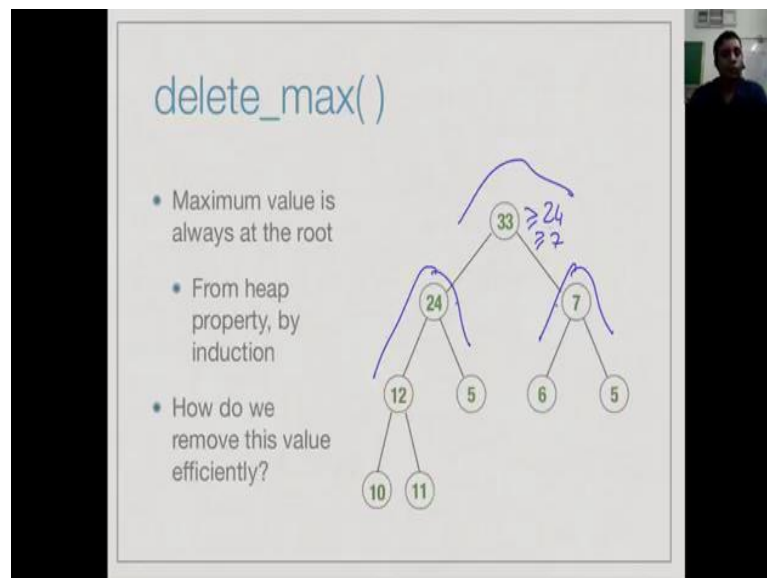
So, we can either counted terms of number of edges or in number of vertices is ((Refer Time: 09:47)) vertices it this soon would be 4, if it is edges it will be 3 does not really matter, but the point is that the longest such path will determine the complexity, because the long at the path the more times I am in need to swap on the via. So, what can say about the height of the heap, so the first thing to notice is that in a heap because of the way that we are done it. So, at the root node we have at level 0 we call this level 0, we have exactly one node at level 1 at most we have 2 nodes.

So, we can write is as 2 to the power of 0, this is 2 to the power 1 of course, each of these will have 1. So, we will double, so at every level the number of nodes doubles, because each of the previous level has two children at most. So, we have to swap, so in this way we have number of nodes at level 0 is 2 to the power of 0 at level 1 is 2 to the power 1 at any level  $i$  is 2 to the power  $i$ . So, if you have  $k$  levels, then the levels are 0, 1 up to  $k$  minus 1 from work we just said that is 2 to the 0 plus 2 to the 1 plus 2 to the  $k$ , the  $k$  minus 1 ((Refer time: 10:55)).

So, it will be  $2^0$  plus  $2^1$  plus  $2^k$  minus 1, now this and one way to think about it this is a binary number with  $k$  1s. So, binary number  $k$  1s is just  $2^k$  minus 1, in other words if I fill up a binary tree for  $k$  levels I will have at most  $2^k$  minus 1 nodes. So, therefore, the number of nodes is exponential are number of levels. So, therefore, if I have the number of nodes in the number of levels must be logarithmic ((Refer Time: 11:31)).

And the number of levels is what determines, the logarithmic length of the longest path and therefore, insert an any heap will take time  $\log$  of  $N$ , because there is every path is going to be guaranty to be of  $\log N$ .

(Refer Slide Time: 11:46)



So, the other operation that we need to implement for a priority queue is to delete the maximum. So, the first question is where is the maximum in a heap? So, the claim is that the maximums always at the root, why is that because if I start anywhere I know that among the any 3 nodes the maximum is that the top. So, if I look at 33 for example, 33 is bigger than 24 and 7, but inductively I know that 24 must be the biggest node in this sub tree and 7 must be the biggest node in the sub tree. So, therefore, 34 is 33 bigger than both it must be the biggest node overall in sent directly.

So, the module is the mod 3 maximum values already at the root. So, now the question is if the maximum values at the root, how do we remove it from the tree efficiently.

(Refer Slide Time: 12:39)

### delete\_max()

- Removing maximum value creates a "hole" at the root
- Reducing one value requires deleting last node
- Move "homeless" value to root

```
graph TD; Root(( )) --- 24((24)); Root --- 7((7)); 24 --- 12((12)); 24 --- 5_1((5)); 12 --- 10((10)); 12 --- 11((11)); 7 --- 6((6)); 7 --- 5_2((5));
```

So, let say you remove the maximum value, so then this leaf has to the hole, we do not have any value at the root now, at the same time because we have remove the value we have reduce the number of values in the tree by one. But, we said that the structure of the tree is fixed, if you reduced by one we cannot remove the root, we must remove the last node going on this left to right top to bottom order.

(Refer Slide Time: 13:05)

### delete\_max()

- Removing maximum value creates a "hole" at the root
- Reducing one value requires deleting last node
- Move "homeless" value to root

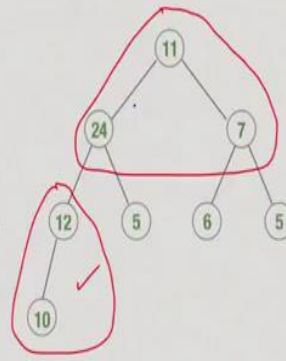
```
graph TD; Root(( )) --- 24((24)); Root --- 7((7)); 24 --- 12((12)); 24 --- 5_1((5)); 12 --- 10((10)); 12 --- 11((11)); 7 --- 6((6)); 7 --- 5_2((5));
```

So, we must in fact to remove the node here this node as to go, so now we have a value which is homeless, it does not have a root node to belong to and we have a home which is empty. So, what we will do is we will move this the 11 to the root.

(Refer Slide Time: 13:25)

### delete\_max()

- Removing maximum value creates a "hole" at the root
- Reducing one value requires deleting last node
- Move "homeless" value to root



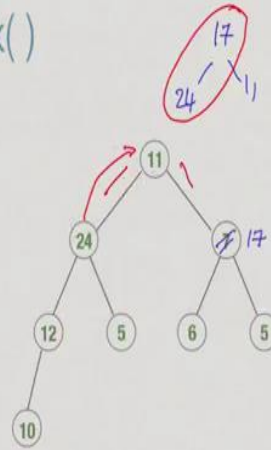
Now, unfortunately because we are disrupt the heap order by doing this taking some arbitrary node from leaf moving into the root, we do not know whether we have the heap property satisfied or not. Now, the only place where the heap property can be violated at the root, because everywhere else the local neighborhood does not touch that it is operation, you only other place their neighborhood was touch to is here, but always did was remove a node. If you remove a leaf then it cannot violate a heap property, because for the upper node it is already bigger than both this tree.

So, this... So, the only place we could have a violation is here and indeed we do, because 11 and 24 on the wrong of them.

(Refer Slide Time: 14:01)

### delete\_max()

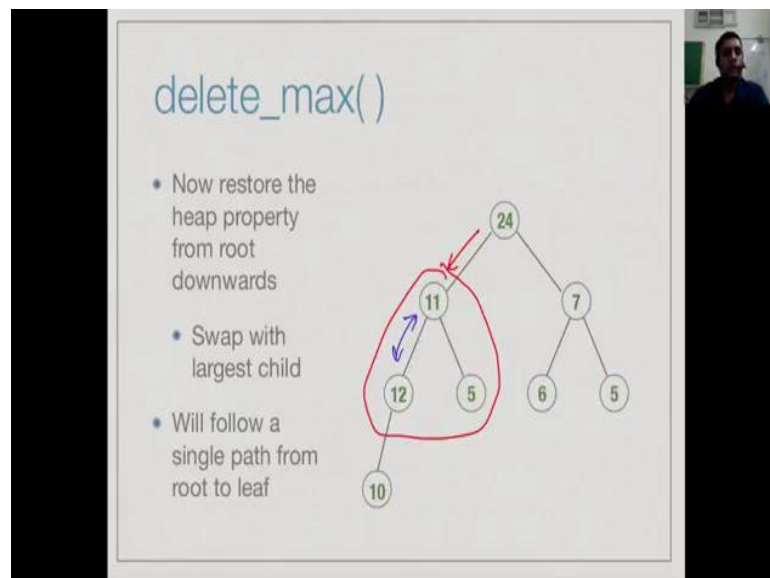
- Now restore the heap property from root downwards
- Swap with largest child
- Will follow a single path from root to leaf



So, we will start now the storing the heap property downwards, when we inserted we did upstairs, it start here and look at this and then we will look at both directions and we take of the bigger one of the two and move it up. So, we swap with the largest child, suppose for instance that this admin not 7, but 17 then what could I happened, if you are move 17 up his viewed about 17 the 11 and 24 and this would not a fix the heap property, because this should see they want.

So, we must take the bigger of the two and move it up, because among these three the biggest value must be at the top that is the definition of the max heap property, so we exchange the 11 and the 24.

(Refer Slide Time: 14:44)



So, 24 goes up to the root and then 11 has come in this direction, so we must second check whether this part which has now been disturb satisfies the heap property. And of course, in this case it does not because 11 and 12 are not in the correct thing. So, again among these 3 I have to take the maximum value up, so I take the 12 up and move the 11 down.



(Refer Slide Time: 15:04)

### delete\_max()

- Now restore the heap property from root downwards
- Swap with largest child
- Will follow a single path from root to leaf

```
graph TD; 24((24)) --- 12((12)); 24 --- 7((7)); 12 --- 11((11)); 12 --- 5_12((5)); 11 --- 10((10)); 7 --- 6((6)); 7 --- 5_7((5));
```

And now I have to check this section whether this heap property is satisfied, here it is satisfied now we want stop. So, in delete max I start from the root and I walk downwards.

(Refer Slide Time: 15:16)

### delete\_max()

- Will follow a single path from root to leaf
- Cost proportional to height of tree
- $O(\log N)$

```
graph TD; Root(( )) --- 12((12)); Root --- 7((7)); 12 --- 11((11)); 12 --- 5_12((5)); 11 --- 10((10)); 7 --- 6((6)); 7 --- 5_7((5));
```

So, supposing we do this again, then I remove 24.

(Refer Slide Time: 15:25)

delete\_max()

- Will follow a single path from root to leaf
- Cost proportional to height of tree
- $O(\log N)$

10

And then I move the 10 from the last leaf to the top.

(Refer Slide Time: 15:28)

delete\_max()

- Will follow a single path from root to leaf
- Cost proportional to height of tree
- $O(\log N)$

10

Then, I again have to fix this problem, so I exchange the 10 and the 12.

(Refer Slide Time: 15:34)

delete\_max()

- Will follow a single path from root to leaf
- Cost proportional to height of tree
- $O(\log N)$

```
graph TD; 12((12)) --- 10((10)); 12 --- 7((7)); 10 --- 11((11)); 10 --- 5_1((5)); 7 --- 6((6)); 7 --- 5_2((5));
```

Then, I have to look at this and fix these problems the biggest of this trees is 11 I fix the 10 and the 11 and I get it.

(Refer Slide Time: 15:39)

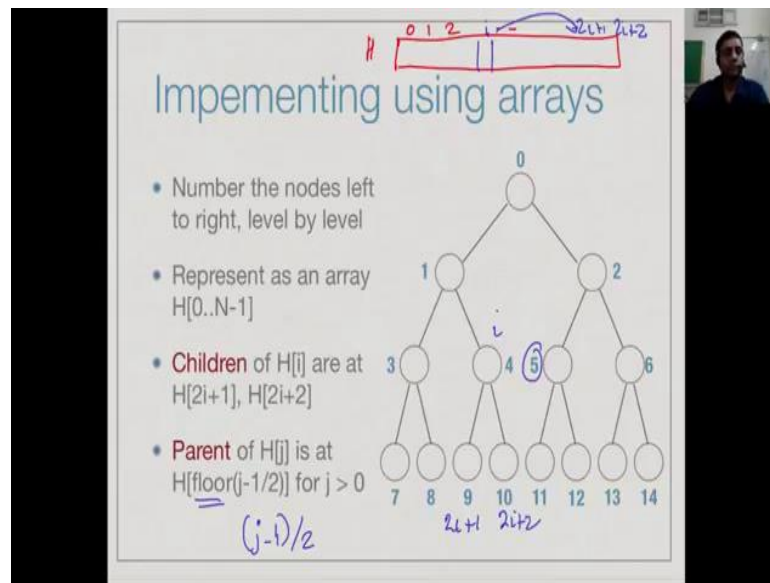
delete\_max()

- Will follow a single path from root to leaf
- Cost proportional to height of tree
- $O(\log N)$

```
graph TD; 12((12)) --- 11((11)); 12 --- 7((7)); 11 --- 10((10)); 11 --- 5_1((5)); 7 --- 6((6)); 7 --- 5_2((5));
```

So, now by making sure that I take the biggest one of again I do not walk down the other directions. So, I am always walking down as single path, so once again just like insert the cost is proportional to the height. And since we know that in a heap the height is logarithmic, delete max is also an order  $\log N$  operation.

(Refer Slide Time: 16:00)



So, what we have done is, we have shown that a heap actually does both delete max and insert in  $\log N$  time. Now, and other very nice property about heaps is that we do not actually need to maintain a very complex tree like structure, we can actually do heaps in arrays to do this we observe that we can canonically number all the nodes in a heap, we start number in the root by 0, the first node we fill below the root by 1, the other child 2 and so on.

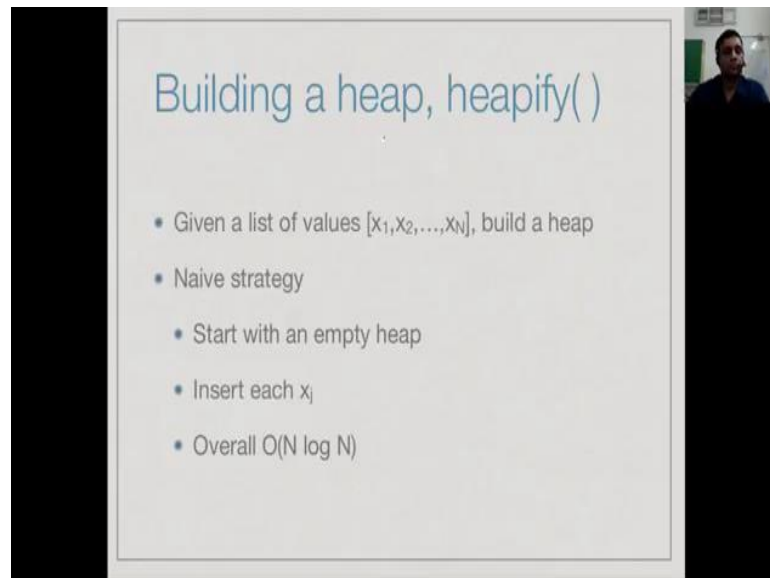
So, I have a numbering 0, 1, 2, so I can actually represent this heap as an array heap which has this is 0, 1, 2 and so on. Now, in this the claim is the define at a position  $i$ , so if I have some position  $i$ , then the children of this are  $2i+1$  and  $2i+2$  you can check this everywhere. So, therefore, if I want to actually go to a heap and ask something about the heap property, then I will just look at the position  $i$ , then I will jump ahead position  $2i+1$  and  $2i+2$ .

So, completely using the array alone within the array I can look up the children of a node and by inverting this operation, if I look at  $j$  minus 1 by 2 and take the floor of that then I will come back this. So, with the child is  $2i+1$   $2i+2$  then the parent is  $j$  minus 1 by 2 and then it might be fractional, so take the integer parts. So, floor means take the integer part of  $j$  minus 1 by 2. So, this is not  $j$  minus half, so it has  $j$  minus 1 the whole by 2.

So, for example, for 12  $j$  minus 1 is the 11, the 11 by 2 is 5 and half floor of that is 5, so the parent at 12 is 5. So, you can check that this is form, so therefore I can now do all my

heap manipulation with in an array, which is very convenient I just have to write an array and then whenever I do these operations which involved walking up and down the heap I will just uses  $2i + 1$   $2i + 2$  formula what are use this floor of  $j$  minus 1 by 2 formula.

(Refer Slide Time: 18:08)



Building a heap, heapify( )

- Given a list of values  $[x_1, x_2, \dots, x_N]$ , build a heap
- Naive strategy
  - Start with an empty heap
  - Insert each  $x_i$
  - Overall  $O(N \log N)$

So, how do we start this whole process of, how do we build a heap from a given set of values. So, a very naive strategy would work as follows and given a set of values  $n$  values  $x_1$  to  $x_n$ . So, I start with an empty heap and then I insert  $x_1$ , so I have a heap of size 1 then I insert  $x_2$ , so now I heap of size 2 and so on. So, I do  $n$  inserts and each insert takes  $\log N$  time at most, so we will take less time we will take  $\log i$  time if I have insert it ((Refer Time: 18:39)), so for but let us take  $\log N$  as an upper bound. So, overall if I insert these  $N$  elements I will build the heap also and  $N \log N$  time.

(Refer Slide Time: 18:49)

**Better heapify( )**

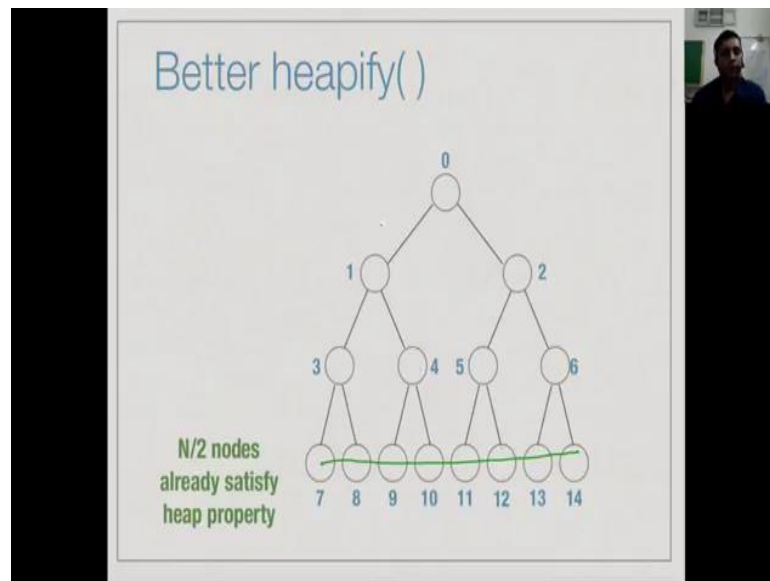
- Set up the array as  $[x_1, x_2, \dots, x_N]$
- Leaf nodes trivially satisfy heap property
- Second half of array is already a valid heap
- Assume leaf nodes are at level  $k$
- For each node at level  $k-1, k-2, \dots, 0$ , fix heap property
- As we go up, the number of steps per node goes up by 1, but the number of nodes per level is halved
- Cost turns out to be  $O(N)$  overall

Now, in fact it turns out that there is a better way to do this, so if I look at any array I can think of this as a heap I can just imagine that it is ordered. Because, every array as a heap interpretation, I can imagine that this is how the array looks, if I think of it the heap of course, it does not satisfy the heap property. But, but this is how it would look if I arranged as the heap. Now, in this anything which is at the leaf level does not need to be checked, because it has no children all leaves trivially satisfied the heap property.

So, I need to start fixing things only at the previous level, so I work back to this. So, I come here and  $x_3$  I fix the heap property with respect to its children, when at  $x_2$  I fix the property with respect to each child, in the process something goes up and down to only one level, then I will come to  $x_1$  and I will fix its problem, now this might involve 2 levels. So, for each level  $k-1, k-2, \dots$  that is. So, leaves are at level  $k$  at level  $k-1, k-2$  on up to the root, we fix the heap property.

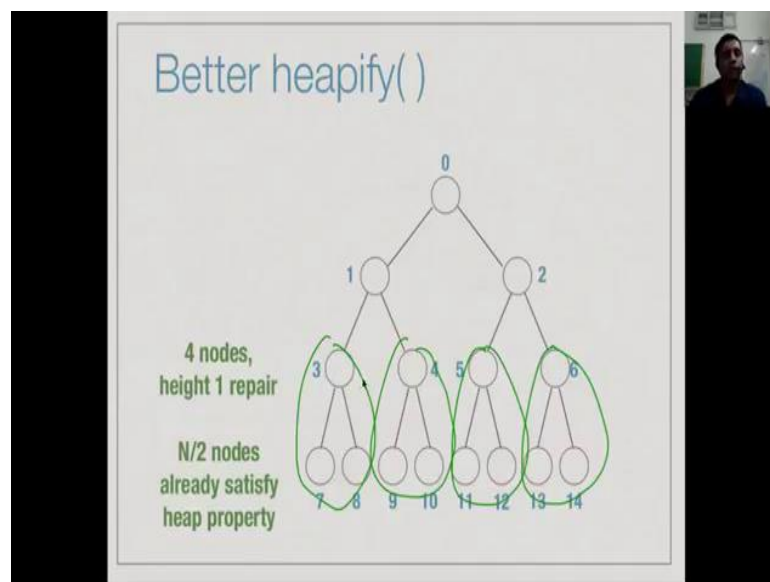
So, as we go up fixing the heap property means, walking down like we did for delete max, walking down to the leaf. So, each level we go up the length of this path increases by 1, but because the levels double as we go down they halve as we go up. So, the number of nodes for which we have to check this extra length path goes down by factor of 2. So, now if you do the analysis should we are not going to do exactly, it turns out that in this process the number of updates you need to make a heap is actually only order  $N$ . So, if you use this bottom up heapification, then it will be an order  $N$  procedure.

(Refer Slide Time: 20:34)



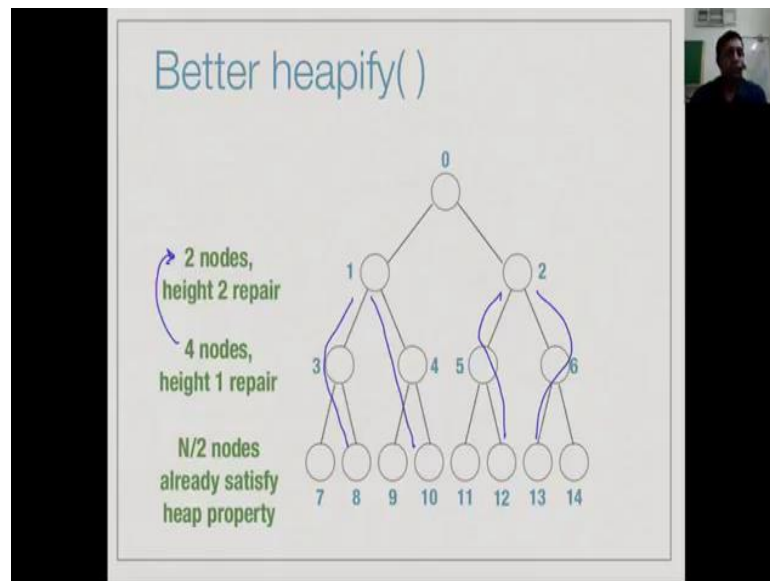
So, just to get a guider picture what is going on, so let us assume that we had 15 elements I had list and we actually through it do it out like this and the clam is that these  $n$  minus 2 node  $n$  by 2 nodes, which is roughly have it is actually 8 out of 15 already satisfied the heap property, so this is nothing to be done.

(Refer Slide Time: 20:50)



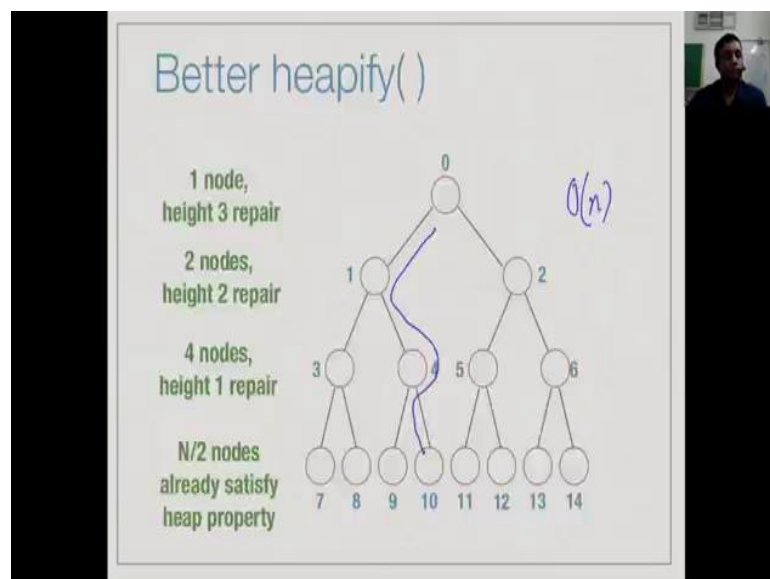
So, then I go one level up and I fix these, when I fix these I have to do it for 4 nodes and each of them the repair will involve one swap at most or no swaps, worst case will ((Refer Time: 21:02)) so on.

(Refer Slide Time: 21:05)



Then, I will up one level and now for each of these nodes I will have to possibly go down 2 paths of length 2. So, each of them will involve a height 2 repair that is 2 step of thing better than all from 4 nodes I have gone to 2 nodes. So, the only half is mini nodes which required only one step node.

(Refer Slide Time: 21:25)



And then finally, when I go to the root I might have to do kind of fix which involves swapping down to the last leaf, so 3 swaps. But, this only one node which does this, so therefore since there is trade of that the number of nodes to be fix is halving and the length is only increasing by one it turns out that this whole operand needs only order  $N$  time.



(Refer Slide Time: 21:45)

**Summary**

- Heaps are a tree implementation of priority queues
- `insert()` and `delete_max()` are both  $O(\log N)$
- `heapify()` builds a heap in  $O(N)$
- Tree can be manipulated easily using an array
- Can invert the heap condition
- Each node is **smaller** than its children
- Min-heap, for `insert()`, `delete_min()`

Handwritten notes:  $v_1 \leq v_2, v_3$  and a diagram showing a node  $v_1$  with children  $v_2$  and  $v_3$ .

So, to summarize go to we have seen is that heaps implement priority queues using special balance trees, in these tree both insert and delete max or logarithmic we can use this bottom up heapify to actually build a heap and order  $N$  time. And what is most useful is that this heap can actually be manipulated very simply as an array, now one thing which we can do is to invert the heap condition. So, we can say that whenever we see  $v_1$ ,  $v_2$  and  $v_3$  we want  $v_1$  to be smaller than both  $v_2$  and  $v_3$ .

So, this is what is called a min heap, what we have been doing, so far is a max heap. So, sometimes you want to keep track of the smallest priority and remove the smallest priority item, just think of how for example, you rank people in an exam. So, you are somebody if in a competitive exam, the smaller the rank the higher the priority. So, if you have rank 1 then you have highest priority. So, this some situation it is natural to think of smaller numbers is higher priority.

So, you want have to do anything very much, we just have to change the heap root to be minimum. So, that the each node is smaller than as two children and then everything would work exactly as we have done, so far. So, we have two types of heaps, you have max heaps and you have min heaps and all the differs in max heaps and min heaps is the heap condition on the nodes and the corresponds you whether the operation you implement is delete min and delete max.