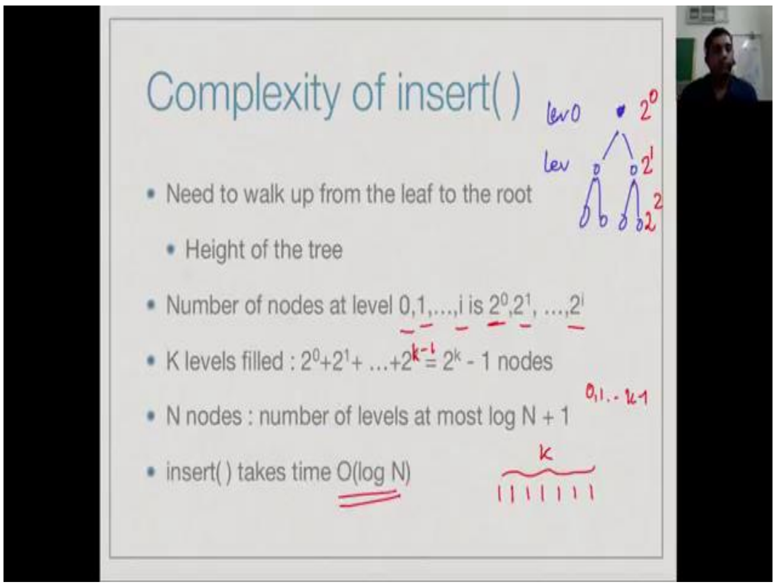
(Refer Slide Time: 09:15)



Complexity of insert( )

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

lev 0
lev
2^0
2^1
2^2
0,1,..k-1
k

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.

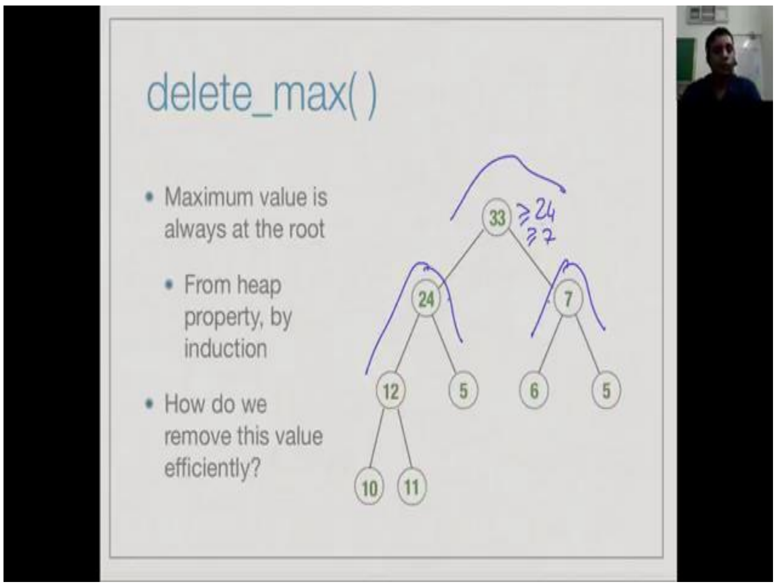
So, we can either counted terms of number of edges or in number of vertices is ((Refer Time: 09:47)) vertices it this soon would be 4, if it is edges it will be 3 does not really matter, but the point is that the longest such path will determine the complexity, because the long at the path the more times 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 to the 0 plus 2 to the 1 plus 2 to the 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 to the power k minus 1, in other wards if I fill up a binary tree for k levels I will have at most 2 to the 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 hide log N.

(Refer Slide Time: 11:46)



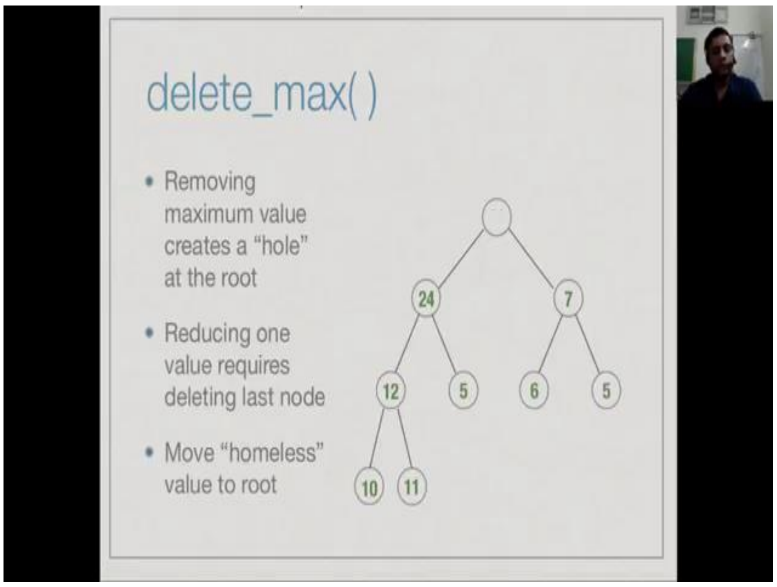
delete_max( )

- Maximum value is always at the root
  - From heap property, by induction
- How do we remove this value efficiently?

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)

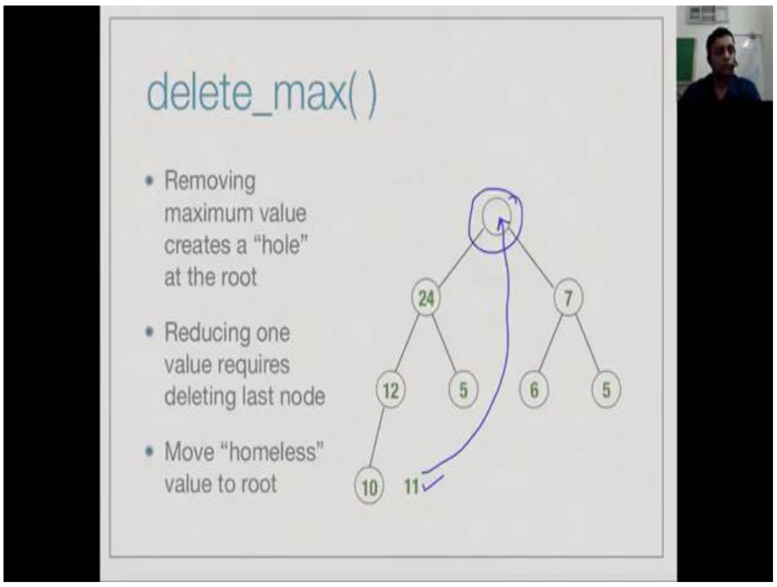


delete_max( )

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

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)

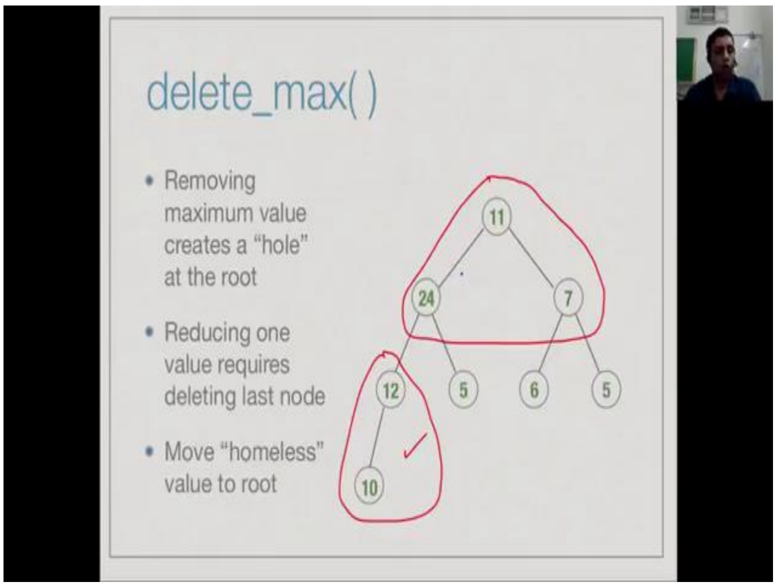


delete_max( )

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

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)



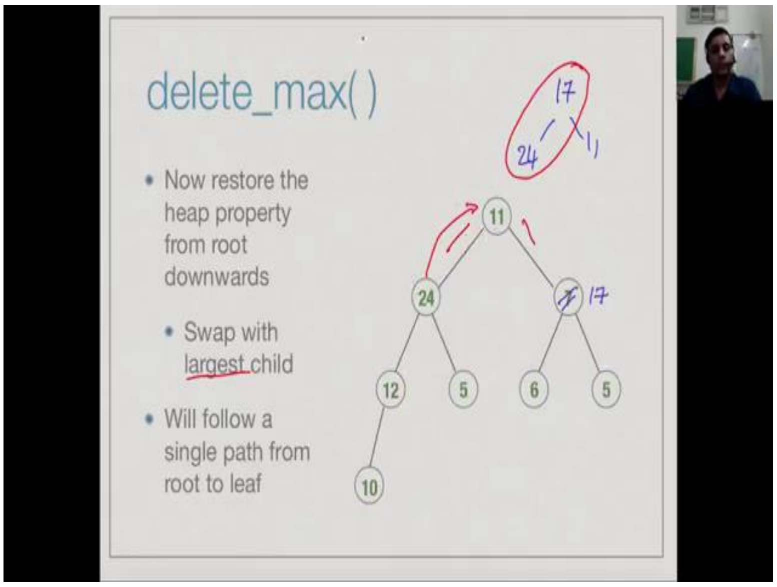
delete_max( )

- 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)



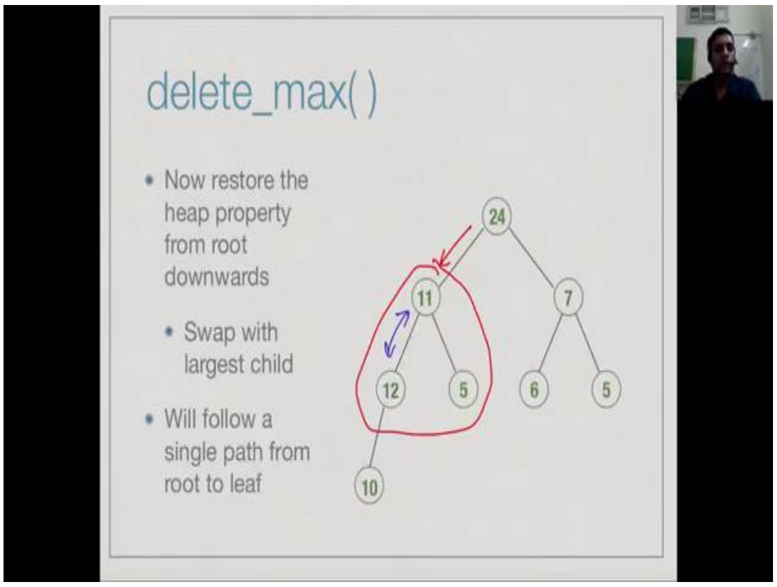
delete_max( )

- 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)

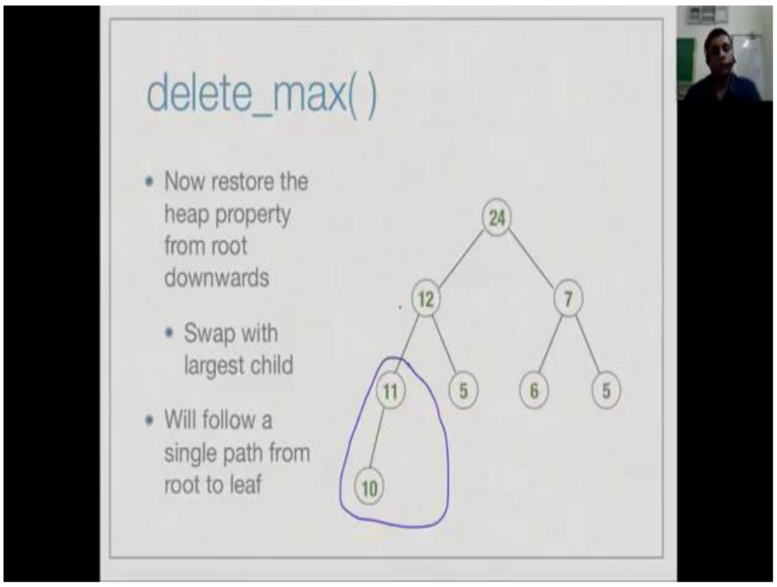


delete_max( )

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

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)

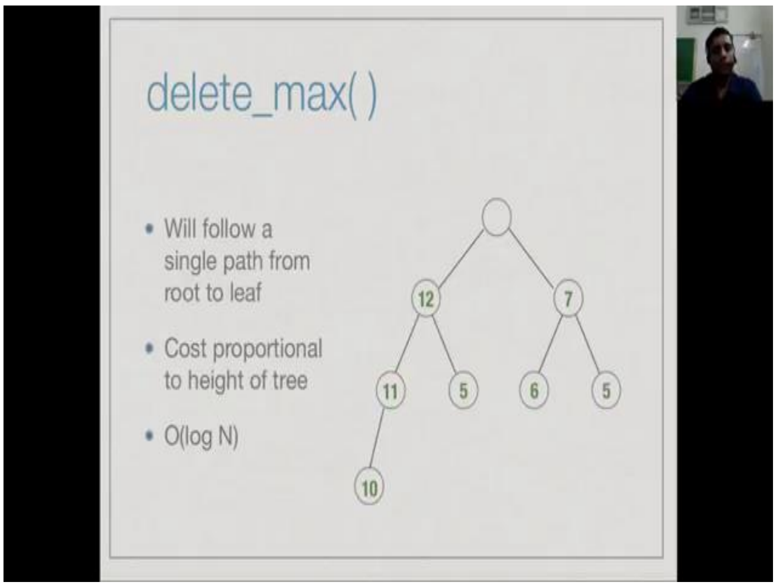


delete_max( )

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

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)

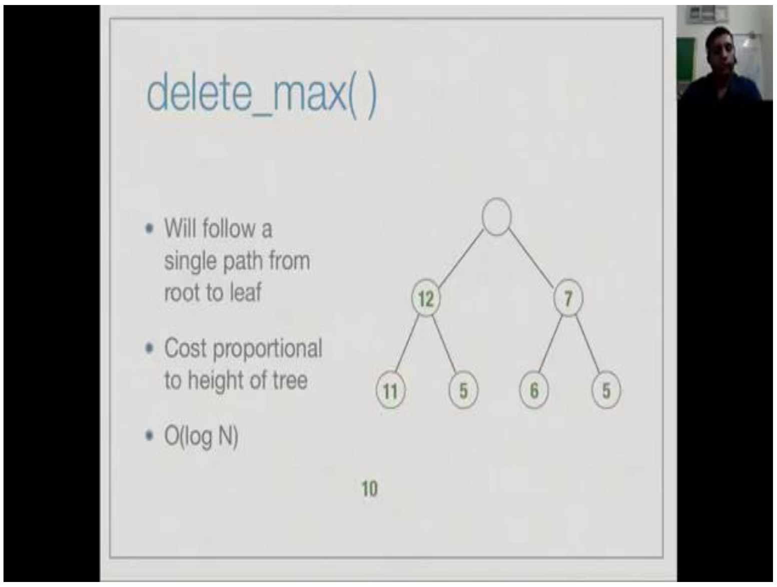


delete_max( )

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

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

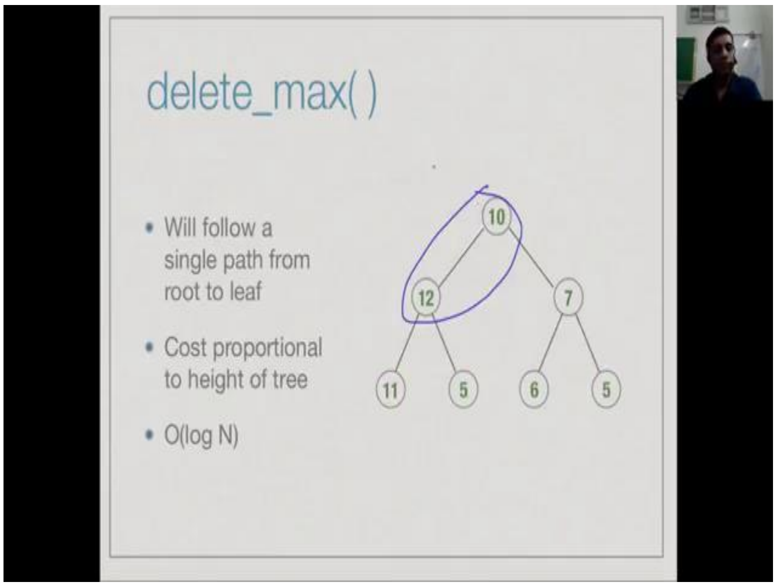
(Refer Slide Time: 15:25)


delete_max( )

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

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

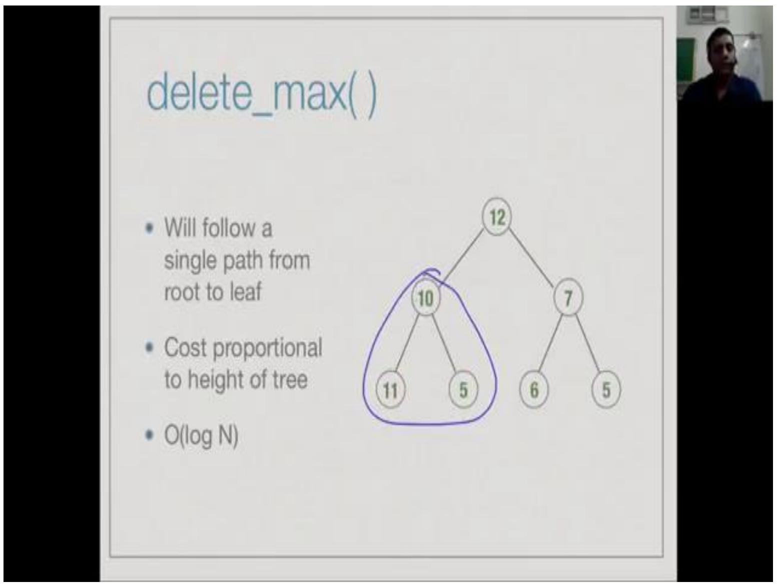
(Refer Slide Time: 15:28)


delete_max( )

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

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

(Refer Slide Time: 15:34)

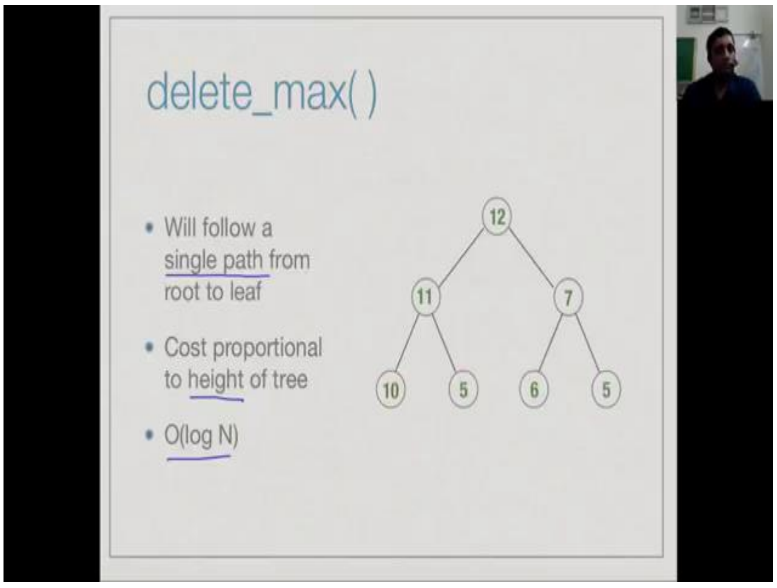


delete_max( )

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

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)

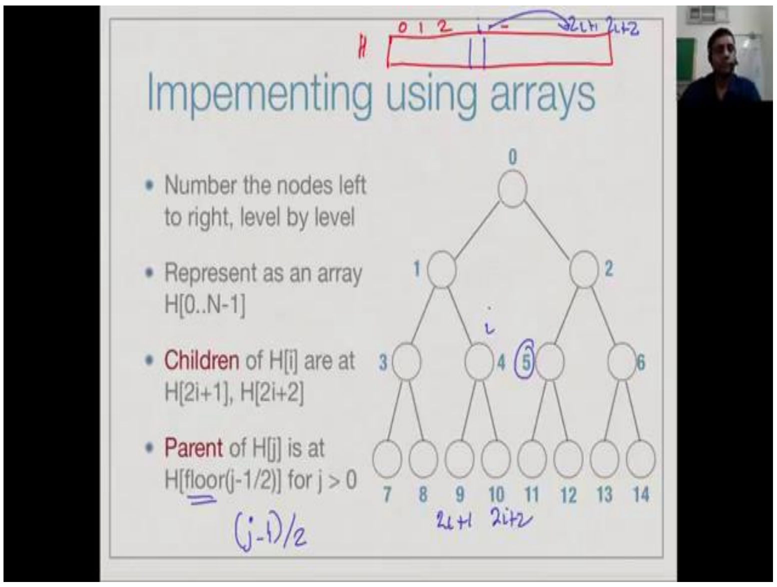


delete_max( )

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

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)



Impementing using arrays

- Number the nodes left to right, level by level
- Represent as an array H[0..N-1]
- Children of H[i] are at H[2i+1], H[2i+2]
- Parent of H[j] is at H[floor(j-1/2)] for j > 0

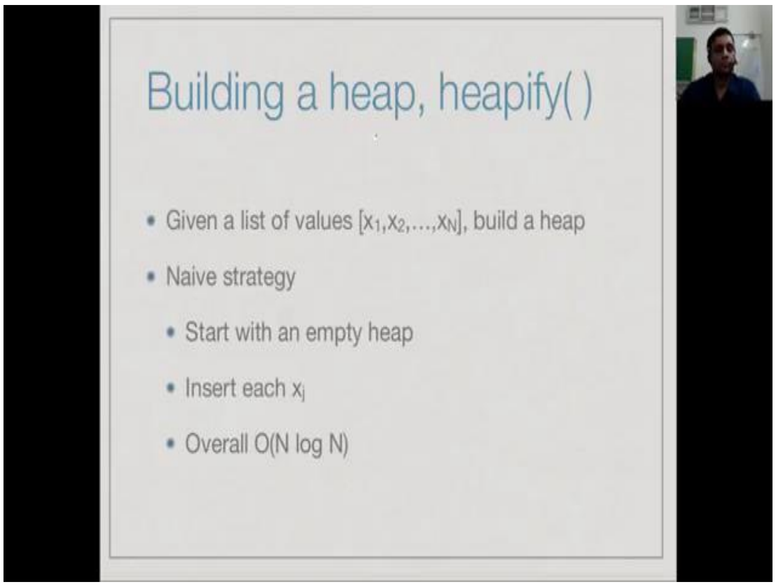
So, what we have done is, we have shown that a heap actually does both delete max and insert in log N time. Now, and other very nice property about heaps is that we do not actually need to maintain a very complex tree likes structure, we can actually do heaps in arrays to do this we observe that we can 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 arrays 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 2 i plus 1 and 2 i plus 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 2 i plus 1 and 2 i plus 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 2 i plus 1 2 i plus 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 2 i plus 1 2 i plus 2 formula what are use this floor of j minus 1 by 2 formula.

(Refer Slide Time: 18:08)

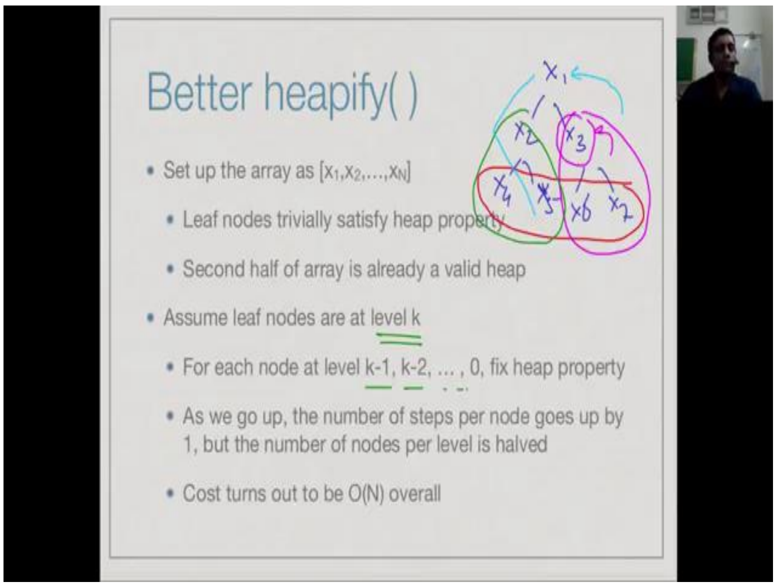


Building a heap, heapify( )

- Given a list of values $[x_1, x_2, \ldots, x_N]$, build a heap
- Naive strategy
  - Start with an empty heap
  - Insert each x_j
  - 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 $[x_1, x_2, \ldots, x_N]$
  - 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, … , 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

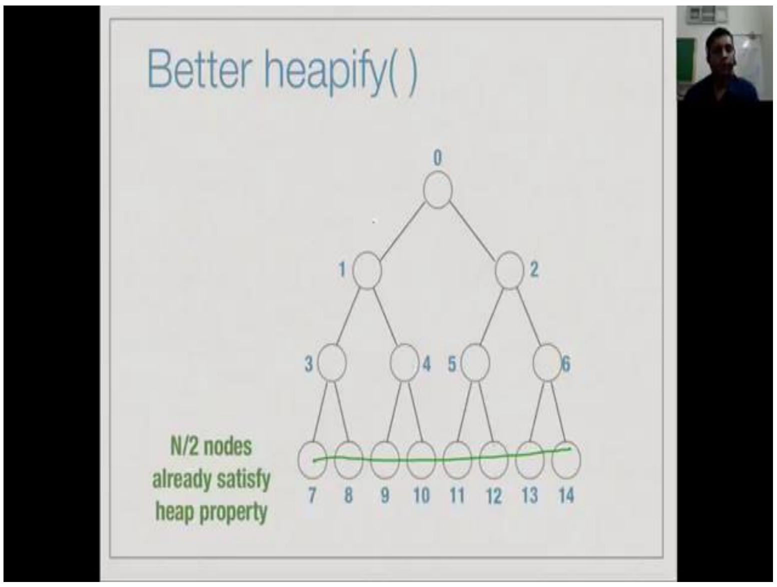
x1
x2
x3
x4
x5
x6
x7

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 check, 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 respective it is children, when at x 2 at fix the property with respective each children, in the process something a up and down to only one level, then I will come to x 1 and I will fix it is problem, now this might involved 2 levels. So, for each level k minus 1 k minus 2 that it. So, leaves are at level k at level k minus 1 k minus 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 have as we go up. So, the number of nodes for which we have to check this extra length path goes down by fact row. So, now if you do the analysis should we are not going to do exactly, it turns out at 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.

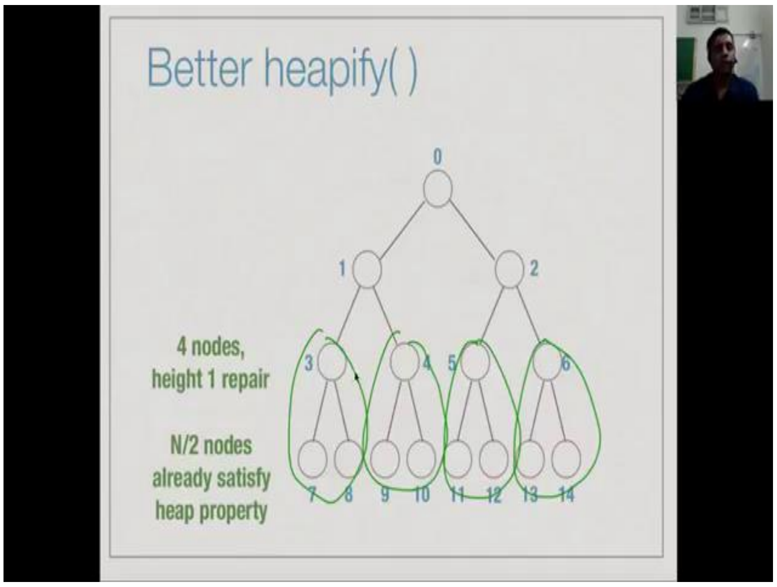
(Refer Slide Time: 20:34)



Better heapify( )

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.

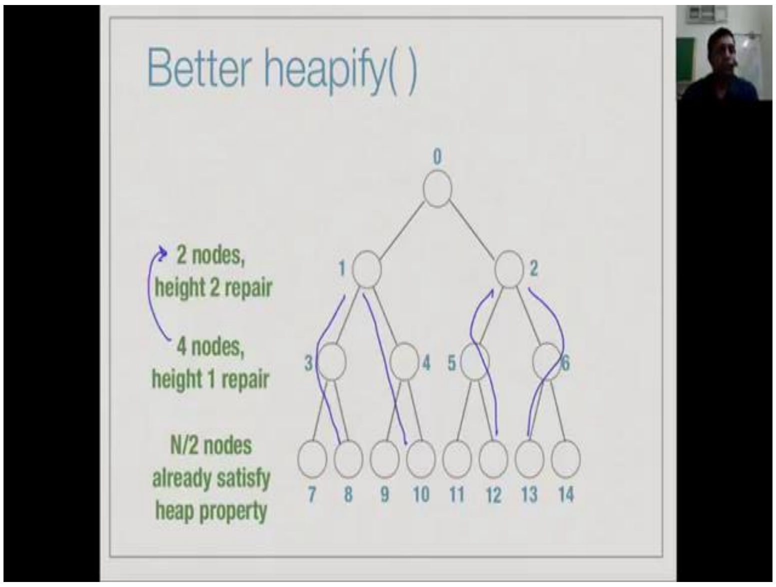
(Refer Slide Time: 20:50)



Better heapify( )

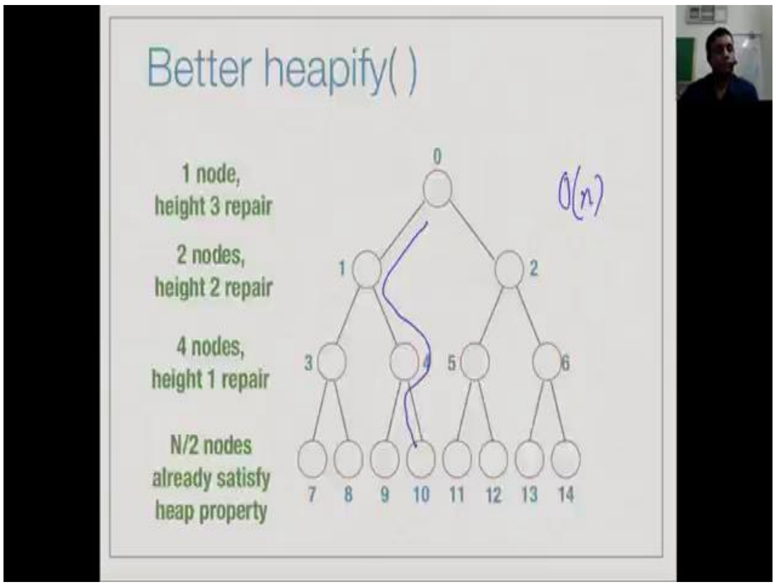
So, then I go one level up and I fix these, when I fix these I have to do it for 4 nodes and each of them the repair will involve one swap at most or no swaps, worst case will ((Refer Time: 21:02)) so on.

(Refer Slide Time: 21:05)


Better heapify( )

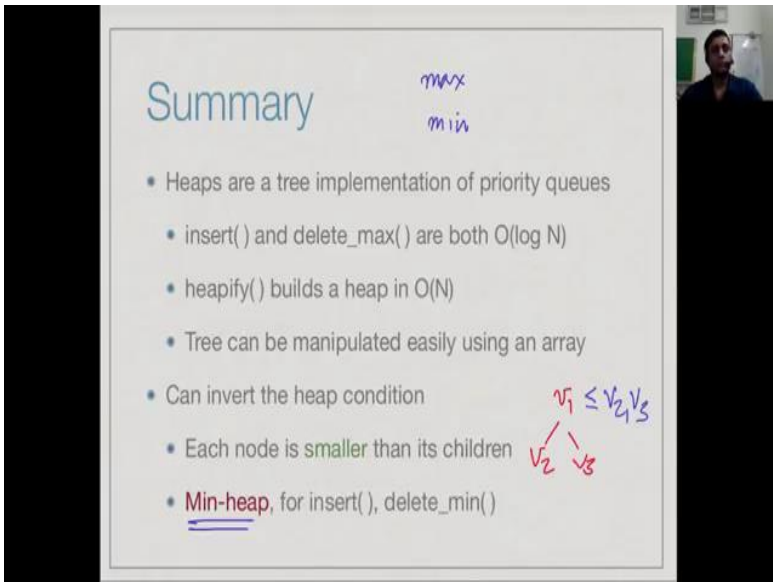
Then, I will up one level and now for each of these nodes I will have to possibly go down 2 paths of length 2. So, each of them will involve a height 2 repair that is 2 step of thing better than all from 4 nodes I have gone to 2 nodes. So, the only half is mini nodes which required only one step node.

(Refer Slide Time: 21:25)


Better heapify( )

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

max

min

- 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( )

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 be 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 for 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 for. 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.