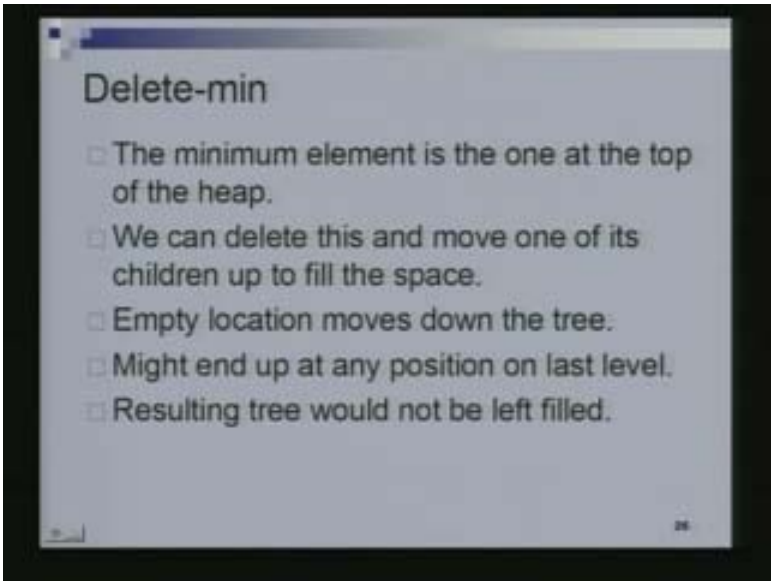
Data Structures and Algorithms
Dr. Naveen Garg
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi
Lecture – 21
Binary Heaps

We will continue our discussion of binary heaps in this class. Recall that in the last class we saw what a binary heap was, we also looked at the operations of insertion and heapify on a binary heap. To recall a binary heap has two properties two critical properties, one is the structural property where we require that the structure of the heap be similar to that of a complete binary tree. So all the levels except the last level of full and even the last level is what we called left full that is all the nodes in the last level are as left as possible. The other property was the heap property which was that for any node, the priority of that node should be less than or equal to the priority of its two children.

(Refer Slide Time: 02:05)

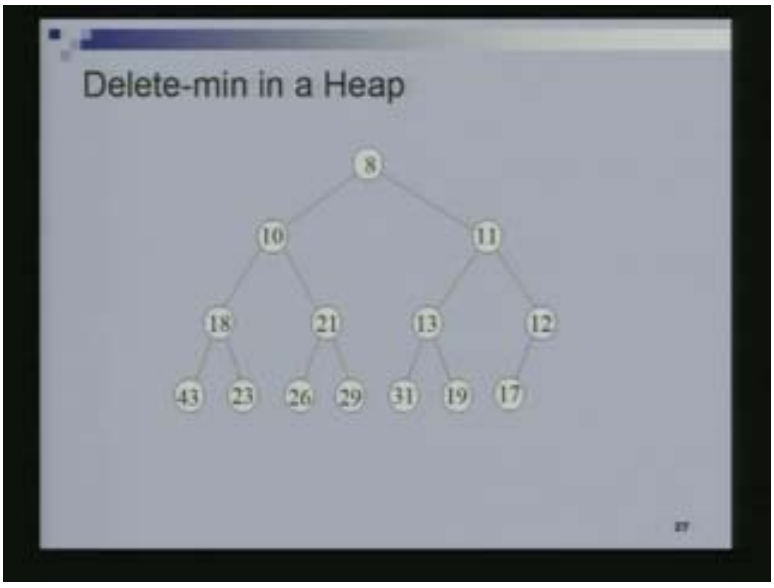
So today we are going to look at the operations of deleting the minimum element from a binary heap and building a heap. It is very easy to build a heap using repeated insertions but today we are going to do a build operation, it just takes linear time. Then we are also going to see how to use binary heaps to do sorting and that procedure is called heap sort. So recall that the minimum element is the one at the top of the heap.

(Refer Slide Time: 02:35)



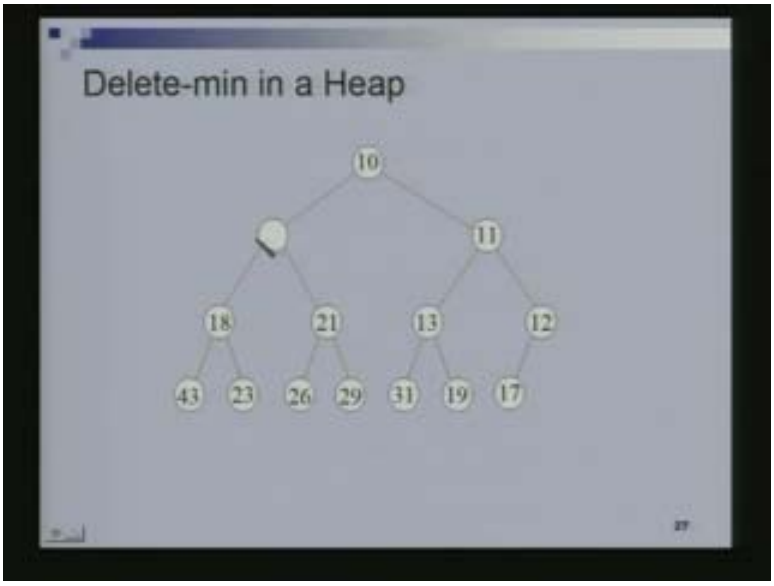
So one way of doing a delete min is to just remove this element and then we have an empty space at the root. So we have to fill up this empty space and to fill up the empty space it would be natural to promote one of the two children of this root element to fill this empty space. So if you were to do that then this empty space would move down into the tree, move down the tree and it might end up at any particular location in the last level. So that the resulting tree might not be left filled. So just to illustrate what I am saying here let us look at this picture.

(Refer Slide Time: 03:14)



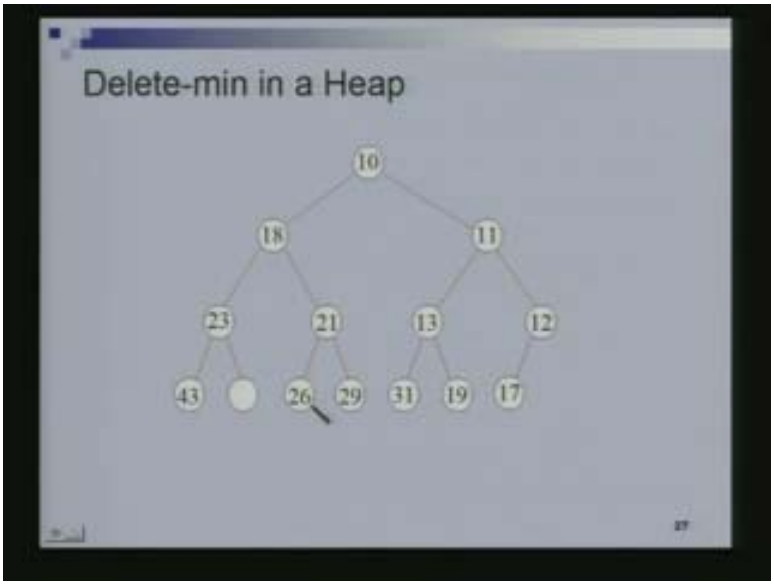
We are trying to delete a minimum element from the heap which is eight. So suppose I were to do that. So this creates an empty location here, now I am going to take the smaller of the two children element and push it up here, move it up here. So 10 is the smaller one I move 10 up there, so I have an empty location now here.

(Refer Slide Time: 3:36)



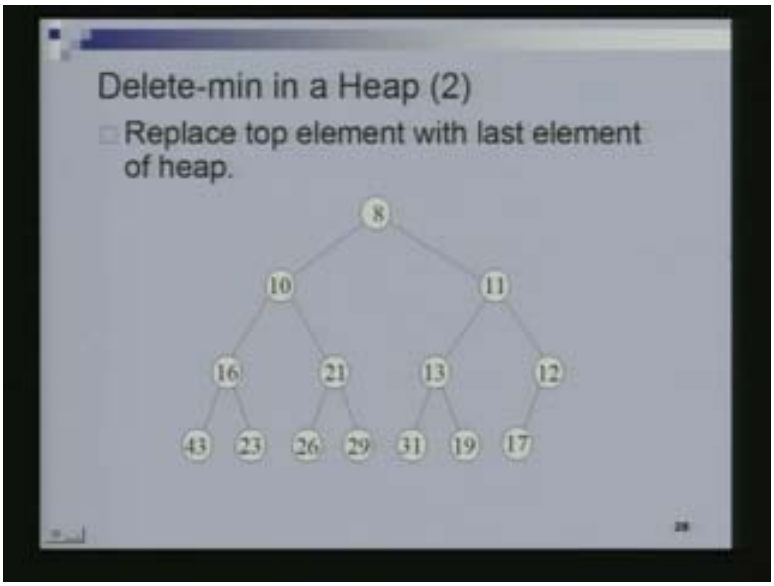
So it is natural to take the smaller of these two and move it up there and this moves the empty location here and now it is natural to take the smaller of these two, 23 and 43 and move it up there at that empty location. So that now we have an empty location here.

(Refer Slide Time: 03:59)



Now while this does satisfy the heap property. This is not a heap because it doesn't satisfy the structural property of a heap. The elements at the level are not left filled. There is an empty slot here. So we can really do delete min in this manner but this is close to what we will be doing in our delete min and let's see what is the right procedure for doing a delete min. So we need to get rid of 8.

(Refer Slide Time: 04:35)



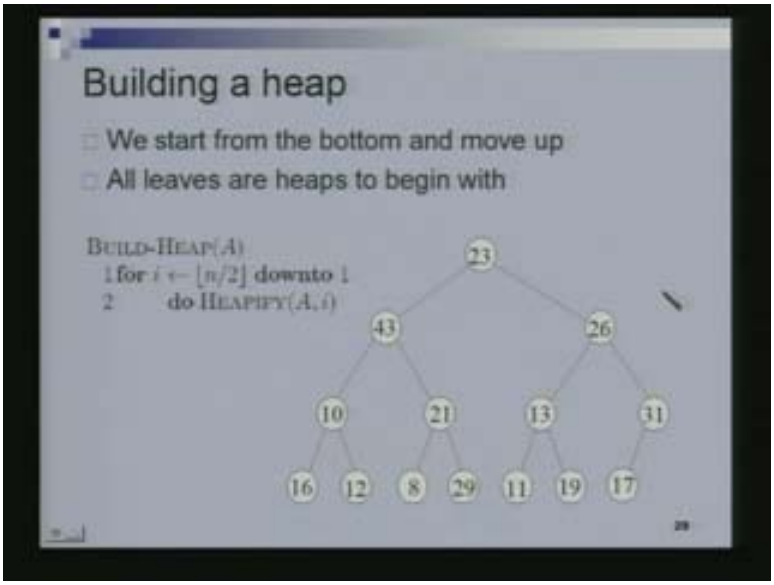
Now when we get rid of 8, the number of elements in this heap is going to reduce. So structurally this heap should not be having this node any more here.

So it makes sense to move this last element here, so we knock off eight and then we move this last element at this place and remove this node. Now structurally this is a heap but it doesn't satisfy the heap property now. So to make sure if it satisfy the heap property we have to adjust the contents of the various nodes but now note the following interesting thing. This sub tree is a heap and so is this sub tree is a heap. The heap property is violated only at this node, this does not have a priority less than the priority of two children. So but we know for procedure for taking care of this problem. If this is a heap and this is a heap, all we have to do is to run a heapify procedure on this particular node. So we just have to do heapify one and recall what does heapify one do. Heapify one we saw also saw this in the last class. Heapify one would take the minimum of these two which is 10, swap 10 and 17. The heap property is valid at this node but it's now violated at this node.

So once again we are going to take the smaller of its two children 16 and swap it with 17. Now the heap property is valid at this node but while it's also valid at this node because if I look at the two children, they are both larger than 17. So this entire thing is now a heap and we have deleted the minimum element. So this is the delete min procedure.

To recap in the delete min procedure we are going to remove the minimum element, take the last element of the heap which means go to the last level and take the right most element. In the array implementation this just corresponds to the last element in the array. Take that element and put it at the root and then just do a heapify of the root, heapify one. So this would make this entire thing a heap once again.

(Refer Slide Time: 07:00)



So this was the delete min procedure. Fairly simple all it required was the heapify procedure. We are now going to see another application of heapify procedure and that is to create a heap.

One way of building a heap is to just repeatedly insert the elements in the heap. So we could insert the first element, second element and the third element and so on and on and how much time does this procedure take? So recall that to insert an element we take log of the number of elements that are already in the heap. So to insert the first element we will take order log one time, for the second element we would take order log two time, for the third we will take order log 3 time and so on and on all the way up to n, if you were to insert n elements the last element would take order log n time to insert. So if you some up this series it's exactly log of n factorial which is the same as n log n. This simple minded method of creating a heap by repeated insertion takes order n log n time.

So we are going to look at this other method of creating a heap which is going to take only linear time and the key to this is to create the heap bottom up. So note that this point, this is not the heap. These are the elements that were given to me, I just put them at arbitrary locations in this heap. So in the array so it just means that since we implementing heaps using array, so we just put all the elements that we have to make into a heap, we just put them into the array. Now we are going to create the heap bottom up, so note that these are already heaps.

Look at this sub tree rooted here which is just this node itself, this is a heap because it doesn't have a child, any children. So it does satisfy the heap property. So these individual leaves are heaps so no problem here. Now what we are going to do when we say we are going to create the heaps bottom up is we are going to make a heap out of this. So we would want that the sub tree rooted at this node also becomes a heap. What is the sub tree rooted at this node? It's this sub tree. How will we make this into a heap? This is already a heap, we have to make this entire thing into a heap. So we will just run a heapify procedure on this and that is what happening here. This element would be at location n by 2, floor of n by 2. We are going to run a heapify procedure on this file.

What would heapify do? Heapify would just compare this with its two children whichever is the smaller child, so it has only one child, so it will just take this smaller child and swap it here. Now this entire thing is a heap. Now we are going to look at this element, so we are going to make a heap out of all of these four sub trees now. So we are basically go to this element, we are going to look at the sub tree rooted at this element and make it a heap. So what is a sub tree rooted at this element? It has these 3 nodes in it 13, 11 and 19. How do I make a heap? I run heapify on this. To run heapify the first thing that heapify does is it takes this, looks at the two children, takes the smaller of T children and swaps it with this. So I am going to make a heap out of this. The smaller of the two children is 11 so I am going to swap 11 and 13. So this also now becomes a heap.

Similarly I now need to make this, the sub tree rooted here a heap. The smaller of the two children is 8, I swap 8 and 21. Now I need to make the sub tree rooted at this node a heap, these are the two children I want to make this a heap but note this is already a heap. This is a heap, this is a heap and this entire thing is a heap because the heap property is also satisfied here (Refer Slide Time: 11:15). So now I have these 4 heaps, these are all heaps. Now I am in a position to run a heapify operation here.

Why run a heapify operation here? This is a heap, this is a heap so I can now run a heapify operation on this one and make this entire thing a heap. To see the advantage of heapify, this is already a heap, this is already a heap so I can make a heap out of this. So to make a heap out of this when I run heapify what does it do? So I am going to make the sub tree rooted at this node a heap now. To do that I have to take the smaller of these two and swap it there so that is 11 and 26 swapped and now recall we are doing heapify. So heapify would bubble up the element all the way to the bottom if need be. So as a consequence of this swap, the heap property is violated at this one. This node now has a larger priority than its two children.

So once again we are going to take smaller of these two and swap it with this and now this entire thing is a heap sort. This was just a heapify operation on this node. The heapify operation on this node just doesn't stop with one swap. It will swap and if need be bubble the element down and that's what happened here. Now I need to run a heapify operation on this node to make this entire thing a heap. This is already a heap, this is already a heap, I need to make this entire thing a heap. So once again it will take the smaller of these two which is 8 in this case and swap it with 43 and now this is not a heap because the heap property is violated here. So I need to run a heapify essentially on this one or actually we are just part of the larger heapify, so we are doing the heapify. We took the smaller of the two children swapped, now we come to this node.

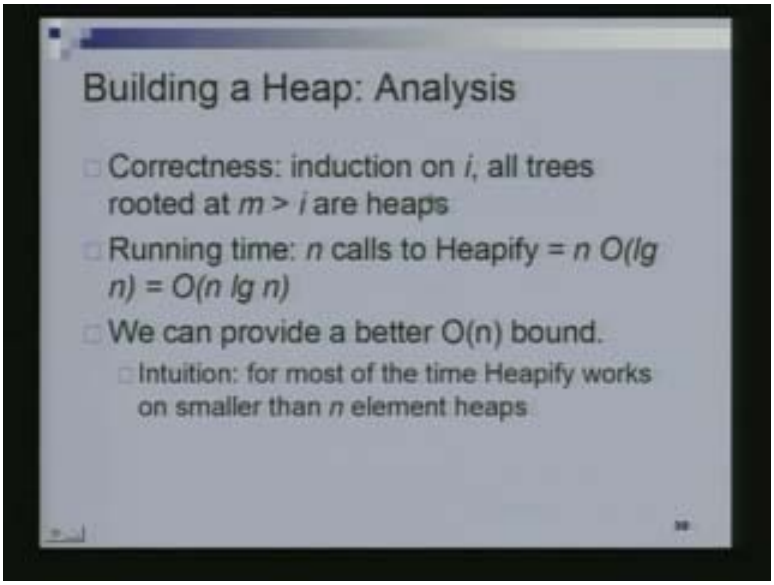
If the heap property is validated here which it is, I will take the smaller of the two children, swap with it 43, so 21 and 43 get swapped. So now this is a heap, this entire thing is a heap and all that remains is to make this entire thing a heap. The heap property is violated here so I run a heapify operation on this. The heapify operation will take the smaller of these two nodes and swap it with 23. So it's going to happen now, 8 and 23 get exchanged.

Now the heapify continues it just doesn't stop with this because now this is not a heap we have changed the content of this node. So we are going to take the smaller of these two and swap it with 23 and then now this is not a heap. So we go and take the smaller of these two and swap it with 23 so we get that. Since this does not have any other children we are done. So this entire thing now becomes a heap. So that was the build heap procedure. The build heap procedure all it is saying is essentially just go down, so recall this was 1, 2, 3, 4, 5, 6, this is how the elements are laid out in the array. If this is element n then recall that its parent is going to be at location n by 2. So this element on which we first have to run the heapify procedure is at location n by 2.

So we first run the heapify on this then we run heapify on this, then we run the heapify here, then we run the heapify here and after we have done all of these we know that these are already heaps. So we can now go and run the heapify here and then the heapify here and then the heapify here. The right order of running heapify is really to first run the heapify on all of these nodes then on all of these nodes and then on all of these nodes and that is exactly what is being done here through this single for loop.

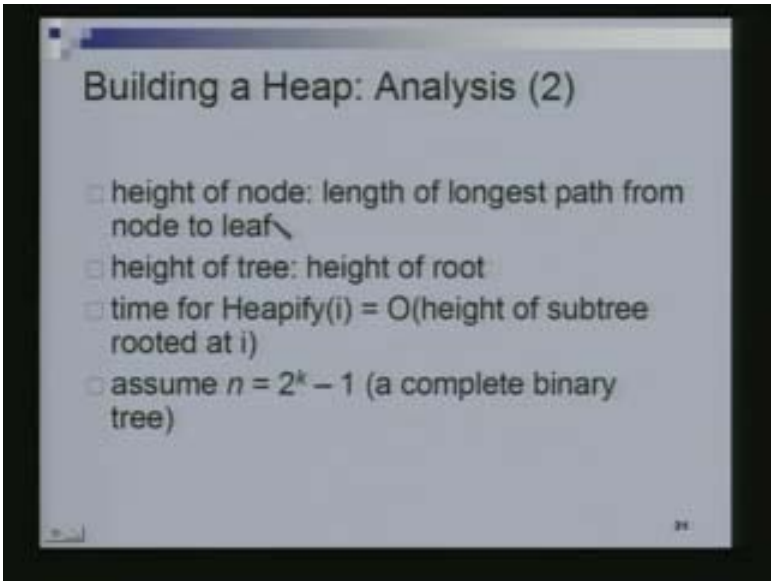
We first run the heapify on all of these then on all of these then all of these. Of course this here is saying that you first run the heapify here then here, then here, then here (Refer Slide Time: 15:10) this is not necessarily required. You could also run the heapify in this order but remember that the heapify has to be run first on this level only then can you proceed to run the heapify here. Why is that? That is required because to run the heapify on this node these two should already be heaps and this can be a heap only after you have run a heapify here. So this is the entire build heap procedure. We are using the heapify sub routine crucially which takes as parameter the location at which you want to run heapify.

(Refer Slide Time: 15:53)



So let's analyze the build heap procedure so to prove correctness and I am not going to discuss it in detail here. You can do an induction on i and the induction claim would be that all trees rooted at locations m which are more than i are already heaps. So given this induction statement, with this induction hypothesis you can do the induction step and prove the correctness of the build heap procedure. A simple minded approach to do the running time computation would just be as follows. There are n calls that we make to the heapify procedure or n by 2 calls that we make to the heapify procedure. Each one of them we saw in the last class takes in the worst case order log n time. So the total time taken by the build heap procedure is n log n but we said the build heap takes order n time and we can actually prove a better bound of order n. The intuition for this is most often we are doing the heapify procedure on heaps which are very small and let us see what this really means. So let's define the height of a node as the length of the longest path from the node to a particular leaf.

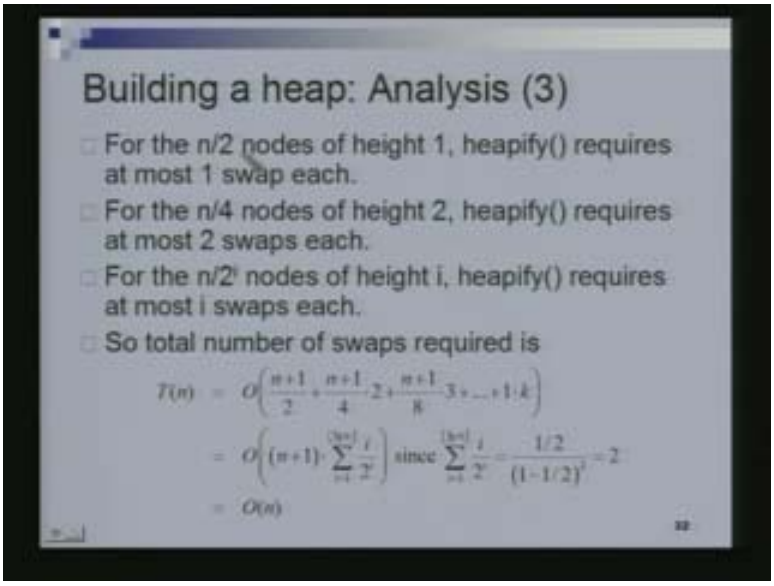
(Refer Slide Time: 17:22)



So the height of this node is one, the height of this node is 2 and the height of this node is 3. We will call the leaves at height zero, we will say that the height of the leaves is zero. So 1, 2 and 3 and the height of the tree is just the maximum height of any node which is therefore 3. So the height of the tree is the same as the height of the root and its 3 in that example.

Now the time for heapify, if I do a heapify on i on location i, the time for heapify is just the height of the sub tree rooted at that node i because in heapify as you recall, we might have to move the element all the way down but if the height of the sub tree is some quantity h then we can only move it h levels down and so the time is proportional to just the height of the sub tree. This we will have to remember that the time for heapify on i is just the height of the sub tree rooted at i. We are also going to assume that the number of nodes in the heap is of the form 2 to the k - 1 so that it is a complete binary tree. This will only help us simply the analysis, it is not really required to we can also do without this.

(Refer Slide Time: 18:48)

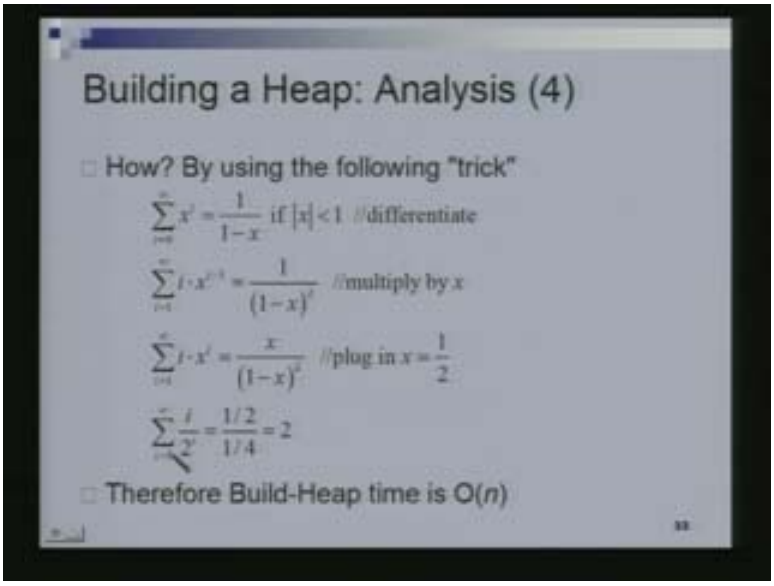


If the number of nodes was of the kind 2 to the k-1 then we know that there are roughly n by 2 nodes of height one and for each of these nodes, we require only one swap. These are height one nodes, the sub tree rooted at these nodes is height one. So we require only n by two swaps. For the n by 4 nodes at height 2, the number of nodes is half the number of the nodes at height 1, height 3 the number of nodes is half the number of nodes at height 2 and so on and on. So at height i there are n by 2 to the i nodes and for each of these nodes you require at most i swaps. So the total number of swaps that are required by swaps I mean the time, so you can count the time required by heapify in terms of the number of times you have to swap the location of two elements.

The time required by the heapify procedure is just proportional to this to the number of swaps. So we are just counting the number of swaps. What is the total number of swaps required then? It is n by 2 times 1+ n by 4 times 2+ n by 2 to the i times i. So this is what the sum would look like, it is basically n +1 or n times summation i over 2 to the i, as i goes from 1 through log n. Why log n, because that's the height of the heap.

It is this sum that we are really interested in and now this summation here, summation i going from 1 through log n. i over 2 to the i is just 2 and I will show you why in a second. So if this is just a constant then this entire thing just becomes order n. So this is what we said earlier, heapify all though we are making n calls to the heapify procedure, most of the calls are being done on heaps which are very small. n by 2 calls have been done on heaps of size of height 1, n by 4 calls have been done on heaps of height 2 and since the time taken for heapify is proportional to the height of the tree on which the heapify procedure is being called much less time is spent here then just saying n by 4 log n which is a crude upper bound.

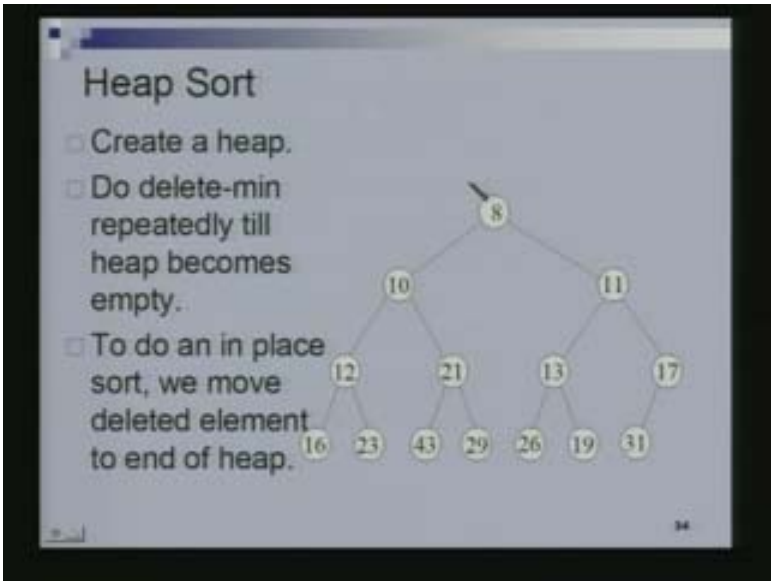
(Refer Slide Time: 21:37)



I have to argue that the summation i over to 2 to the i is 2 and this is a simple argument for that. So recall that summation x to the i, i going from zero through infinity is 1 over 1 - x, if x is less than one. Now if I just differentiate this I get i times x to the i -1, i going from one through infinity. I have dropped the i equal zero term because that would now contribute to zero. The differential of the right hand side is 1 over 1- x square.

Now I multiply both sides by x to get i times x to the i equals x over 1- x square and now i plug in x equals half. So I get i over 2 to the i, i going from one through infinity equals 1 over 2 divide 1 over 4 which is equal to 2 and if you recall what we required was that this sum go from one through log n. So that then is only going to be less than or equal to 2 or strictly less than 2. So the sum from 1 through infinity is 2. So that completes analysis to show that one can build a heap in just order n time. The key thing here was that we build heap procedure went bottom up, it first created smaller heaps and then combine them into larger heaps by just using the heapify procedure repeatedly.

(Refer Slide Time: 23:09)



Today we have seen how to delete the minimum element from a heap. We have also seen how to build a heap in linear time, in order n time. Now we are going to see how to sort using heaps. So give you a bunch of elements and I want to put those elements in increasing order. So what would be one way of doing it using a heap? I could just do the following, I could take those elements, build a heap using them. Then I could repeatedly remove the smaller element from the heap. So that is exactly what I have said here. I first create a heap, this can be done in order n time as we just saw then I repeatedly remove the minimum element from the heap till the heap becomes empty. So how many elements would I have to remove in all? There are n elements in the heap, I am going to be removing n elements in all and each of the delete min procedure, each of the delete min steps requires order log n time. So I do n steps here, so the total required would be order n log n and this requires order n times, so the total time required is order n log n.

Now suppose you have to do an in place sort by in place sort I mean that you are given an array of n elements and you are given no other space. This is all the space you have and you just want to swap the elements in this array so that finally you have a sorted sequence. Recall that our heaps are implemented using arrays. So we first create a heap, for that we did not require any additional space. Starting with the initial array, just create a heap in that array.

Now the contents of the array are basically 8, 10, 11, 12 so on and on, in this order sitting in the array. Now when I delete the minimum element, where does it go? Where would I put this element, because I don't have any additional space. So now what we are going to do is when we delete this element we are going to put it at the end of the heap. So recall that for deleting the element, I am going to move this element here to the top, so which means that this location in the array is now available to me.

It is free so I am going to use this location to move 8 to put 8 in. Essentially I am swapping 8 and 31 and now this location is really not part of the heap. The heap is this now and actually this is not a heap because it doesn't satisfy the heap property. So to do a heapify, to make this into a heap so which means that the same as before. I take the smaller of the two children's, swap the element with that now again heap property validated here, smaller of these two and swap and now this is again a heap. So all I have done is, so this was essentially the delete min procedure. I have done a delete min, I have removed 8 but I have kept the 8 in my array at the very end.

So this is in place sorting. We are not using any additional space, we delete the element but then we keep it in the array in an empty location and now once again we do a delete min. so we delete 10 and 19 is going to come here. So this location is going to become empty so ten and nineteen essentially are getting swapped. This location now goes away from the heap, so this is not marked which essentially means that this is the part of my heap.

Of course I once again need to ensure that the heap property is satisfied, so I am going to do the swaps, 11 and 19 get swapped, 13 and 19 get swapped and 10 and 19 are not going to get swapped because this is not part of the heap any more, the heap is just this. So the heap property is satisfied here because this is only one child and it has priority less than the priority of its child, its lone child. So this is now a heap.

So once again I do the delete min, so 11 is going to be deleted, 26 is going to come here, 11 is going to come here because this location is now going to be empty and so this is what is going to happen. This location is going to go out of the heap now because the heap is only this part and once again we are going to do the swaps. We are going to do a heapify at the root so as to convert this into a heap and that is what we are doing now and now this is a heap.

As you can see the last element is the minimum element. The second last element is the second minimum and so on and on. Eventually this is what is going to happen, we are going to have a sorted sequence but in decreasing order and that can easily be reversed in linear time to get a sorted sequence in increasing order. So I will just continue the procedure, we swap the last element with the root and then this element goes away from the heap. Now we are just going to do the swapping so 13 and 29 get swapped, here the smaller of these two, 17 and 19; 17 is going to get swapped with 29 and now this is again a heap furnace.

Now the minimum element is sitting here, 13 is going to swapped with the last element, this element is going to go out of the heap now. It fades away out of the heap and now we are going to ensure the heap property by doing the necessary swaps. So smaller is 16, it got swapped, smaller is 21 it got swapped and now this is a heap because these two are not part of the heap. So heap is only this thing, this is the smart. So once again you swap the last element in the heap with that root element. So 16 comes here and 26 comes there, this goes away. It is not part of the heap any more and then we are going to ensure the heap property by doing the necessary swaps.

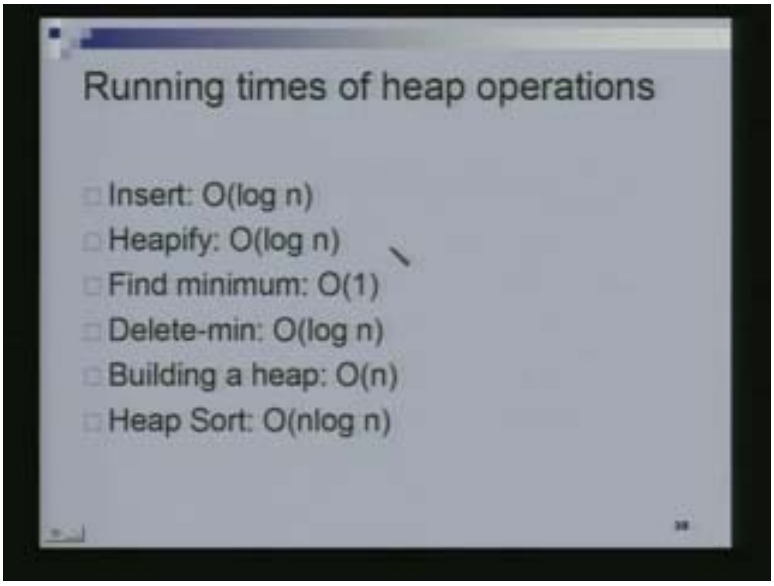
So 17 and 26 get swapped and now the heap property is violated here because this has higher priority than one of its children so 19 and 26 are going to get swapped and now this is a heap. This again is now going to swap with the last element, 17 and 31 get swapped, 17 is going to drop off from the heap, 17 drops off and now we are going to ensure that this entire thing is a heap by taking the smaller of its children, swapping heap property validated, take the smaller of these two and swap.

The heap property is now valid. This is a heap so this is a minimum element, swap it with the last element, 19 and 29 get swapped. This element goes away from the array, gets dropped and now we have to ensure that this is a heap. The heap property is violated here, take the smaller of these two, 21 and 29 are swapped. The heap property is still violated; take the smaller of these two, 23 swap it with 29. This entire thing is now a heap. So we are going to remove the minimum element which is 21 we remove it, 31 goes in this place and since this location is now empty, 21 comes at this location.

Now we need to ensure that this is a heap, 21 dropped away now take the smaller of these two children twenty three, thirty one. Swap 23 and 21 and then take the smaller of the two children and swap it with 31. So now this is a heap, so the minimum element is 23 we are now going to remove the minimum element which essentially means swap it with the last. This is not part of the heap anymore and once again ensure the heap property by swapping and this is now a heap. The smallest element is 26, exchange 26 and 31 remove it from the heap and now ensure the heap by doing the necessary swaps. So the smaller of these two children's is 29, so 29 and 31 are going to be swapped. This is a heap now, smallest element is 29.

So I am going to remove this and 43 is going to come at this location, so 43 comes here 29comes here, this is not part of the heap anymore so I drop this off and now I need to ensure that this is a heap. So this now has only one child, this is the remaining part of the heap, this has only one child and which is smaller, so I need to swap them. So this is a heap now. So the minimum element is 31, I remove this minimum element and the last element is 43, it will come at this place. So 43 comes here, 31 is removed and it is put at the location of the last place because this location now gets empty. So essentially that corresponds to 31 and 43 getting swapped again and this element going away from the heap. Now this is the only element left in the heap and so it is a heap. I do a delete min which means I remove this element and it's not part of the heap anymore. So this is what we get, as you can see this is a sorted sequence in decreasing order now if you read it like this and this is how we do heap sort.

(Refer Slide Time: 33:40)



So let's quickly summarize the running times of the various heap operations that we have seen so far. The last thing we saw was the heap sort which takes a total time of order n log n. Why did it take a time of n log n? This was because it was a two step process, first we created a heap. This took only order n time and then we did delete min repeatedly till the heap becomes empty. So the first time we did the delete min operation we spent log n time, the second time we did a delete min operation we spent order log n - 1 time let's say because the size of the heap reduces and so once again we have a series of this kind log n plus log n -1 plus log n-2 plus log n -3 going all the way down to a one but the sum of this series is log n factorial which is the same as n log n. The total time taken by this system is order n log n.

So while this is only order n, the total time taken by this step is order n log n and that implies that heap sort takes a total time of order n log n. Building a heap, we saw a bottom up procedure for building a heap. So there are two ways of building a heap, one is repeated insertion. Repeated insertion we insert one element at a time and we argued that takes total time of n log n and you can actually come up with the examples were it takes that kind of time. So repeated insertions would take n log n time but if we did this bottom up process of building the heap where in the leaf elements are already heap, the sub trees of height one you made them a heap then you made the sub trees of height two a heap then you made the sub trees of height 3 a heap and so on and you repeatedly use the heapify procedure to be able to do that.

So if you were to do it this way this bottom up construction of a heap this takes order n time. The delete min operation which we also sorted is takes only order log n time this was because the delete min operation is a two-step thing. First we remove the minimum element which is sitting at the root, take the last element in the heap.

So in the array implementation this corresponds to the very last element in the array and put it at the root location, put it at the location of the first element which we have removed which was the minimum element. So once you did that, now this is not a heap because the heap property could be violated at the root but the two sub trees the two children of the root, the left child and the right child and the two sub trees rooted at this two children are heaps. So we can invoke the heapify procedure on the root.

Heapify we have already discussed and we are going to recap today, a recap just now takes only order log n time. So the total time taken for delete min then is log n for the heapify and constant time to do this swap of moving the last element to the very first location. So total time taken is order log n so this was delete min where you are removing the minimum element. If you just wanted to find what the element was, the element with the least priority that we said is the element sitting at the root node. Now that we can just directly access and so finding the minimum element just take constant time.

In the last class we saw the heapify operation we also saw it repeatedly in the delete min and the build heap procedure today and also in heap sort actually. Heapify is really crucial operations and we saw that it takes only log n time this is because heapify we are bubbling the element down the tree and the worst case we might have to bubble it all the way down but since the height of the tree is no more than log n, it will take no more than log n steps to do that.

In the insertion process on the other hand we are moving the element up the tree, so first we decide what the new structure of the tree is, so we add an additional node, we put the element there and then we keep moving it up till the property at all the node is not satisfied. So we keep moving it up so it bubbles up the tree. Since once again the height of the tree is only log n. in the worst case we might be bubbling up the tree at most log n levels and so the total time taken by the insert procedure is at most order log n. So as you can see, if i use a heap to implement the priority queue data structure then the worst case time complexity of any of the operations, so forget heap sort because the typical operations that we are doing are insert, find min and delete min. These were the three operations we started of with and while find min is done very quickly, it’s just constant time. Insert and delete min also don’t take too much time they take only order log n time.

Compare this with the implementation we had done using a sorted sequence in an unsorted sequence. In the case of an unsorted sequence we said insert would take constant time but find min and delete min will both take order n time. In the case of a sorted sequence we said that insert would take order n time while both find min and delete min could be done in constant time. So it’s not the case, in some settings you might be interested in implementing a heap, implementing a priority queue using a sorted sequence and which would those settings be. If you were implementing it using a sorted sequence then as I said both find min and delete min just take constant amount of time but insert takes a lot of time. It takes order n time. If you had a setting where you were doing a lot of find min operations, very few insert operations then it might make sense to use a sorted sequence.

So depending upon the application one has, depending upon the settings one is in, one might have to choose between I mean the different ways of implementing a priority queue. So we have looked at three ways heap, sorted sequences and unsorted sequences and depending upon which operation of occurring more often, one might have to choose an appropriate implementation. With this I am going to end today's class. Today we looked at the other operations on heap. In particular we looked at the delete min operation and the operation for building a heap in a linear time. We also saw how to use a heap to do sorting in order n log n time only. This sorting that we saw was an in place sorting algorithm.
Thank you.