**NPTEL**
**NPTEL ONLINE CERTIFICATION COURSE**

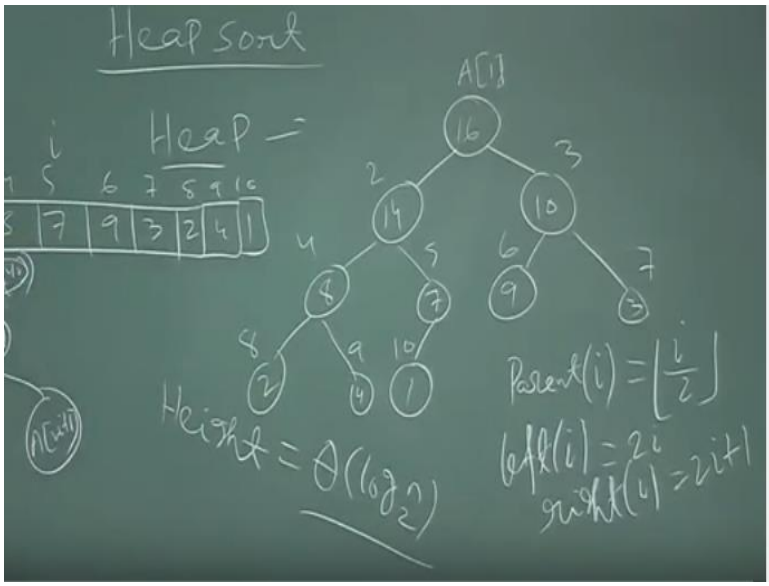
**Course Name**
**Fundamental Algorithms:**
**Design and Analysis**

**By**
**Prof. Sourav Mukhopadhyay**
**Department of Mathematics**
**IIT Kharagpur**

**Lecture 05: Heap Sort**

Okay, so we talk about heap sort, so before that we talk about what is heap?

(Refer Slide Time: 00:28)




So heap is a data structure which is basically an array but it is viewed as a nearly complete binary tree. Array but it is viewed as a binary tree, okay. So it is a, it is basically an array, so suppose let us take an array, say we have an array like this 16, 14, 10, 8, 7, 9, 3, 2, 4, 1, suppose this is an array, this is 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, so there are 10 elements.

So how we can view this array as a binary tree, so we just take the first note A[1] as a route of the tree, okay. Then we take its left child and right child as A[2], A[3] and then A[4], A[5], A[6], A[7], A[8], A[9] second elements to A[10] okay. So we are viewing this binary tree. So A[1] is 16, 14, 10, 8, 7, 9, 3, 2, 4, 1, so this array which any array we can view like this as a tree.

So that means if we have A[i] over here, if this is A[i] so for A[i] what is the left child of A[i] it is basically A[2i] and A[2i+1] is the right child of A[i]. And what is the parent of A[i] is basically A[i/2] I mean ceiling individual part of this. So basically the parent of (i) is basically i/2 and the left child of (i) is basically A[2i] and the right child of (i) is basically A[2i+1].

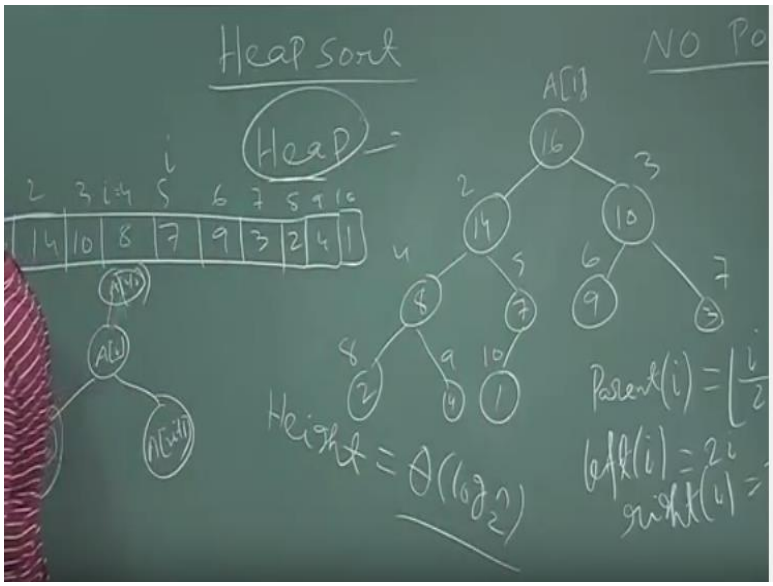
So this is the position of the left and right child of A[i] and this is the position of the parent. So any array we can view it as a binary tree. And this is nearly complete binary tree, because if number of element being is $2^2$ then we have a complete tree, here will stay. So there are 11, 12, 13, 14, if there are 15 elements then we have a complete tree.

So that is why it is called nearly complete binary tree. So what is the height of this tree, height is basically logn, that is why it is nearly, so it is a good tree. But the advantage of this tree is we are viewing this tree, I mean we are not, I mean we do not need any pointer to implement this tree usually for tree implementation we need a pointer for the left child array, right child array, parent.

But here we need, we do not need that, we do not need, no pointer is required to halve this tree. Because if we know the i, I mean we know if i is 4, then we know the child's are basically 2i to 2i+1. So this tree is just we are viewing like this. So we do not need to have any pointer to implement this tree.

Okay, so this is called heap, this data structure, this array is called heap which is viewed as a tree like this. So this is called heap, okay. Now we introduce a max heap which is a property in this heap, so if every element is greater than from his child.

(Refer Slide Time: 06:04)




So if the key value of every element, so that means, so here it is A[i] and we have the child A[2i], A[2i+1] so if A[i] is greater than maximum of this two A[2i], A[2i+1], that means if the every notes is greater than if from each child the key value of the note is greater than from each child then it is called a, and if this true for all I, if this is true for all i then we call our heap is a max heap, then the heap is a max heap or we can say our heap is having a max heap property.

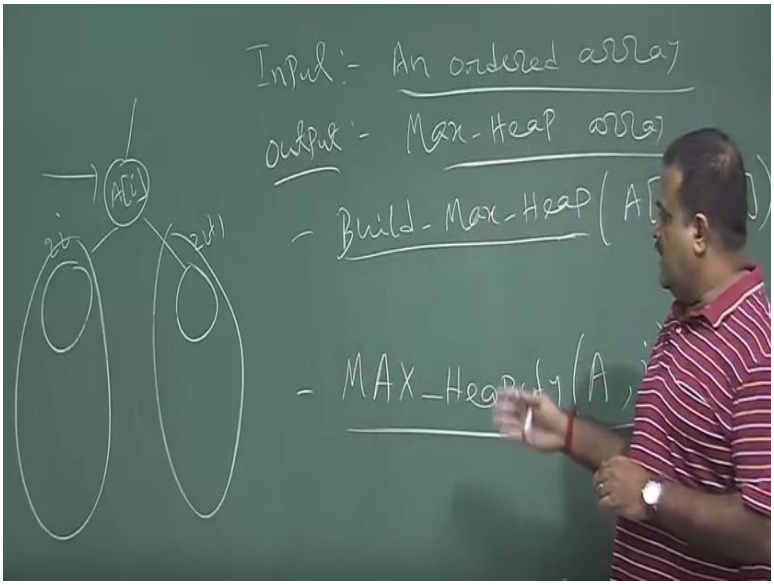
Okay, and similarly we can define the mean if so if the, if any node is less than from value of that node is less than each child, and this true for all node then we call that heap as a mean heap. So we will talk about max heap and we will see how we can use this to have a sorting algorithm, so this heap is nothing to do with sorting, I mean heap we can start independently so this is basically in data structure.

This is used to use for the biotic implementation so, and so now we will talk about how we can have a max heap array from a given array. So is this array is a max heap array, this, in this example, so this array is basically 16, 14, 10, 8, 7, 9, 3, 2, 4, 1, so this is our given array and the question is, is this a max heap array so you have to check whether each node is greater than from

each child so 16 is greater than from each child, 14 is greater than 8 and 7, 8 is greater than this, 7 is greater than this, 10 is greater, yeah.

So this is a max heap array, this array is max heap so it is having the property that each node is bigger than from each child, now but if we have some array which is not max heap array now the question is how we can make that array as a max heap array so the, so the input is any array.

(Refer Slide Time: 09:18)



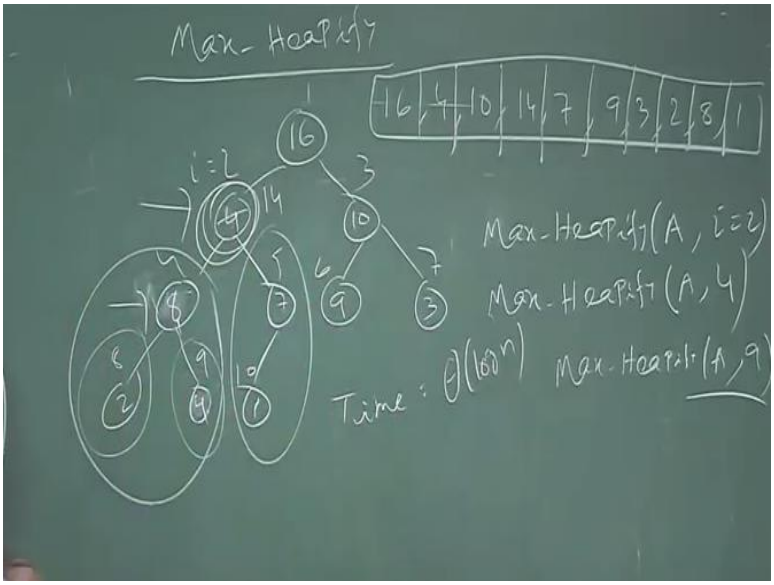

And an ordered array and the odd we want to make it output will be our goal is to build, make it max heap array.

Okay, so this we want to discuss how we can do that, so for this we need two subroutine, one is called build max heap so this is the, I will go for this so it is taken, it will take an array and it will convert into max heap array and in the build max heap we will use a subroutine which is called max heapify A, i suppose we have a, this is how our A,i and this is 2i, I mean this is 2i+1 that is then that means the left sub array and this is the right sub array of A, i. Now we are calling this max heap, max heapify of this array at the point A.

So for this pseudo code or for this code we need to have assumption that assumption is, so we assume this left sub array is already max heapify, right sub array is already max heapify, so we have to handle this here, so here we have a violation of the max heapify so this code can handle only one violation so everything is okay here, everything is okay here, although we may have a violation at this level.

So that is how we can handle this, is this clear? So this code can one handle only under this assumption that the left sub array of i is max heapify, right sub array of i is max heapify, only thing we may have a problem over here so that we have to handle, so that is our, that is by the max heapify code, so let us see what is the max heapify, how it is working?

(Refer Slide Time: 12:14)




Okay, we will take an example by which we will suppose we have this array 4, 10, 14, 7, 9, 3, 2, 8, 1, so this is an array like this 16, 4, 10, 14, 7, 9, 3, 2, 8, 1, so this is our input given array, okay. Now this is 1, 2, 3, 4, 5, 6 these are the indices 7, 8, 9, 10 now if you look at this note. So if you look at this note this is our i say, i=2 this note you have a problem, what is the problem?

Because this note is not greater than from this child it is less than even both of the child, so it is not a max heap, max heap property is violating here, but if you look at the left sub heading and if you look at the right side of this $i^{th}$ node this is max heap array this is max-heap, so no violation in these two, only violation we have here so that is can be handle by this code. So what do we call here? We call the max heapify at the point i = 2.

This is the so we have a violation here, but for that we need to have everything is okay here, everything is okay here, that is there, okay. So now we how we can handle this, how we can handle this violation? So we check this two whichever is the maximum we replace that with this, so 14 is the maximum so what we do? We replace 14 by 4 so 14 will go here and 4 will come here, okay.

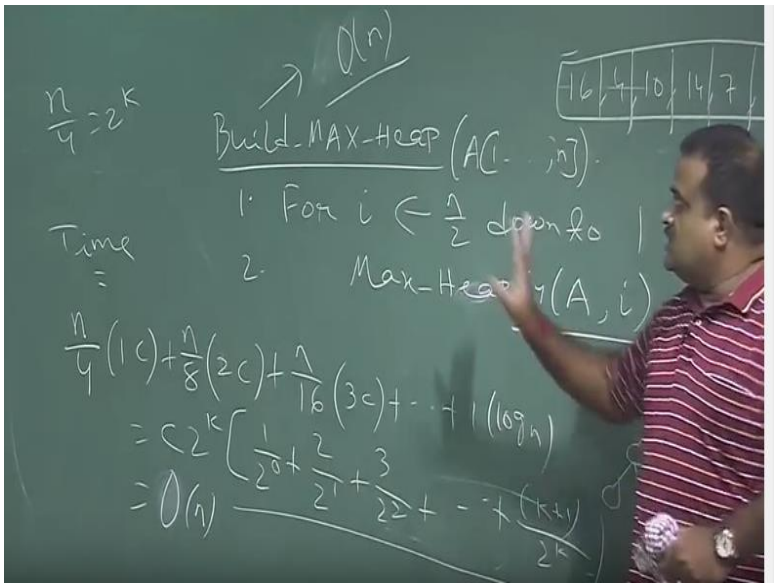
So now this is basically 4, now once 4 will come here then we may need to check whether this is violating the max heapify property here a lot, this is this will problem will come down, so now here also it is not but it is not max heapify, so what we need to call? We need to call again the max heapify (A, i=4). So then if we call this when we are calling then we are making sure make sure that these two part is max heapify.

We need this is only one element so it is max heapify, this is the leap node. So now what we do, we exchange this with the maximum of these two so maximum is 8 so you exchange 8 and 4 and then this is, there is no further this is the leap node so we stop here, so this is the max heapify code, I mean the execution of the max heapify. So what is the time complexity for this? So time is basically log n.

Because if our height of the tree is n we may have to start from the root, okay. So it is basically log n height of the tree. So every time we are exchanging the maximum hallow and we are just calling the max heapify again, now again we have to call the max heapify here at (A, 9) but this is the leap node it is already maximum so we will stop here, so this is the max heapify sub routine but this is, this can handle only if two part is the left part of the array is max heap and right part of the array is max heap property.

Only we can handle one violation, okay. So now how we can use this to build a max heap array so that we will now discuss, okay.

(Refer Slide Time: 16:56)




So next we will talk about build max heap, so it is taken an array A of size n, okay it is very simple code two line code, this is the simplest code I have ever seen so this is for this is for I ← n/2 down to 1 we call the max heapify (A, i) that is it, so this is the pseudo code for build max heap.

Now the question is why you are starting from n/2 to 1, why not? So this is our array, array is from 1 to n, so the question is n/2 then n/2+1 ….. n question is we are not looking at this, this elements we are just calling this max heapify, all the from n/2 so what is the result for that, because see look at this array so this is a1, 2, 3 like this, this is our, anyway maybe we have up to this okay so this is 1 2 3 4 5 6 so can you tell me what is this element basically, n/2 to n/a n/ 2 + 1 to n, these are all basically leave notes because if you take any notes from if you take any index from here then two height y is out of the range, if you.

So these are all leave notes, these are all leaps so leaps is already back simplify, we have already one note it is already max so we do not need to care about the leaps, but these are the notes so this, this, this, this, this…..these are all from n/2 to a n. Now only thing, now this is the note at, this is n/2 this is n/2, this the first note which is having at least one child, okay, now when you do to care from this because this maybe having some higher lesson of max f so we need to call the max f here, now when we are calling you are insuring that this is maxi because it is only all the.

If we up to child also then all the leap notes they are already maxi so when you need to call maxi if or this like this, like this where we will continue and in rows to the relief root, so this is the code for this, okay, so what is the time complexity for this, what is the time for this code? Intuitively it is maybe your thinking that it is n log n because there are n notes and each, each in the it maybe the log n is the height but you see here.

Not for all notes we are spending log n time because these are the notes which is just having one comparison, because for this note we are just comparing one time maxi call, comparing this maximum and that is it, so for this fast level note we are just comparing one, so how many such notes are there which is in this level, I mean we can say this is from here it is level one so there are n/4 note, so there are n/4 note which is are at these level so how many comparison we are doing for this?

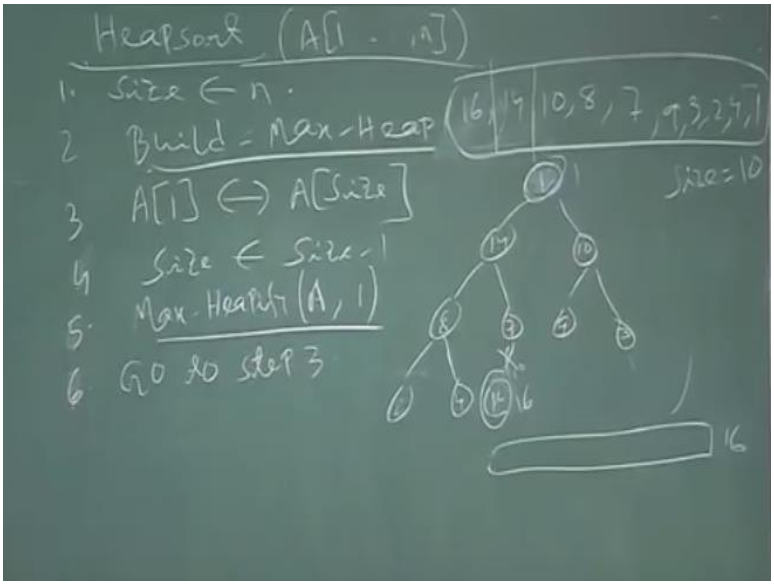
We are doing constant comparison so n/4 x 1 x c and in the next level when you are calling the maxi heapify again so for them we may need to go down twice, okay so how many such notes are there, n/8 x 2c the constant comparison we may say, then like this n/16 x 3c like this, dot, dot, dot, but at the root level root has to come down for the root level calling the max heapify it has to come log n time so this is the, the exact time for this code.

Now if you take $n/4 = 2^k$ then this is coming out to be seen to $2^k$ $1/2^0$ + 2/21+3/ $2^2$ + dot, dot, dot, $k + 1/k^k$, okay, now this term is basically a constant so these constant can be plot with this and this basically a order of we O(n), okay. So this is the linear time, linear time algorithm so time complexity is this 0(n), okay, so now we will, so this is the build max heap so we have given an

array, any unwanted array how we can make it maxi heap vary, so now we will see how we can use this maxi heap array to have a sorting algorithm, so far we have not talked about.

This, we start the heap which is nothing to do with sorting but we now we will see how we can use this to have a sorting algorithm and that is called heap sort, okay.

(Refer Slide Time: 23:37)




So we have a array we need to, this is sorting problem so we have a array of size n and you need to sort it so what we do, we define the size of the array, this is a n, so now we call the build max heap on this array, so once we call the build max heap on this array so it will give us a, it will give us a max heap array where maximum is belonging to A[1], so like this, okay like this, okay. So maybe you have some more element at this what is the n, now what we do we just exchange this A[1] and A[n], we just exchange A[1] with the, with the last element, and we reduce the size of the array by 1, okay that means the last element is, is in its position.

Now again we, so while we are swapping so everything was fine here, so we now this is, this was a new element, so this may get violation of the max heapify, so then but for there we know this is okay, this is okay, so now we need to call max heapify at this level, okay. And then after calling

the max heapify we will go to, sorry this is 5, this is 6, we will go to step 3, and we continue this until the size become 1. So this is the code, so we can take a quick example how this is working, okay, so this was our array so 16,14,10, 8, 7, 9, 3, 2, 4, 1 so we can just draw the, so anyway this is the max heap array 14,10,8,7, 9, 3, 2, 4, 1, okay.

So we may have a general array but we can make it a max heap array, so suppose we have array so you make it max heap array so we got this. So after this step we have this, now what we do our size here is 10, so what we do we extends this, this is the 10 and this is 1 we extends this with this, so 16 will be coming here, 16 and 1 will be going here, so this is now 1, okay. Now we just cut this link. So 16 is in correct position we discuss this, so this, this, this, this tree we are viewing, this tree is does not exist, we are not implementing this, everything is we are doing in a, in this array. So that is why it is called impulse sort.

We do not have, we do not need to have extra memory or something. Everything we are doing here, so this is now 1 so now this, this, this may violate the max heap property, so we again do the max heapify for this, so this we will exchange with this so 14 and 1 will come here, so this 1 will come here 8 will go here, so this 1 will come here 4 will go there, so then we will reduce the size, now the size is 9, so now so now this is our, so now this 14 we exchange with this 1 again, so this is 14 now, so this is 1 again so this 14 we cut this link and we reduce the size so this way we continue and at the end we got the sorted array.

So we just forget this part because this is already in its position so like this we can, so what is the time complexity, so this will linear time and here we have a for loop this is on the size of the array n, and each time we are calling the max heapify this may take logn because up to n/2 it is taking logn, after that we are, so this will take n logn, so in total time is $\Theta$(n logn) so this is same time complexity as merged sort but this is better in the sense, this is impulse sort, so we are not using any extra memory for this, everything we do here, doing in the array only thing we are viewing these array as a tree.

We are not implementing this tree, this tree is just our view, so we are everything we are doing in the array itself, so this is in place. But for the merged sort we need to have the extra array for the merge subroutine in a linear time, okay. Thank you.