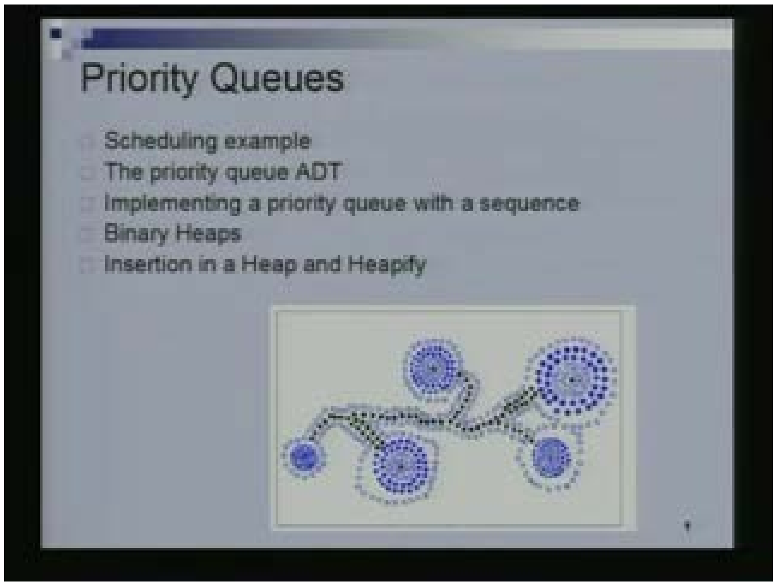
Data Structures and Algorithms

Dr. Naveen Garg
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 20
Priority Queues

Today we are going to look at the priority queue abstract data type. I am going to motivate it with the scheduling example then we are going to look at what the data type is, we are going to see how to implement the priority queue with the sequence then the key thing today we are going to do is to introduce the concept of binary heap. We are going to see how to do procedures for insertion in a heap and procedure called heapify which will be using in subsequent classes to see how to find to delete the minimum element in the binary heap and for other operations on heap.

(Refer Slide Time: 01:48)

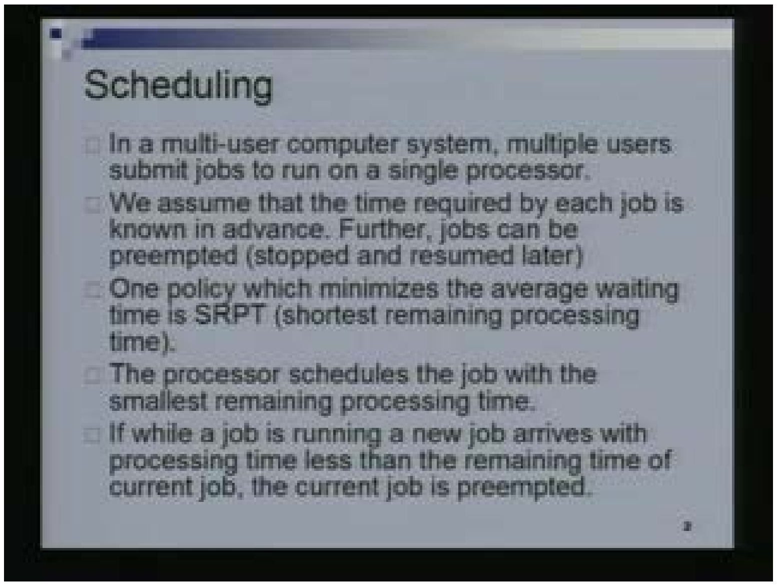


Priority Queues

- Scheduling example
- The priority queue ADT
- Implementing a priority queue with a sequence
- Binary Heaps
- Insertion in a Heap and Heapify

So we have a multi user computer system, that's the system in which there was multiple users who can submit jobs at different points in time.

(Refer Slide Time: 03:16)



Scheduling

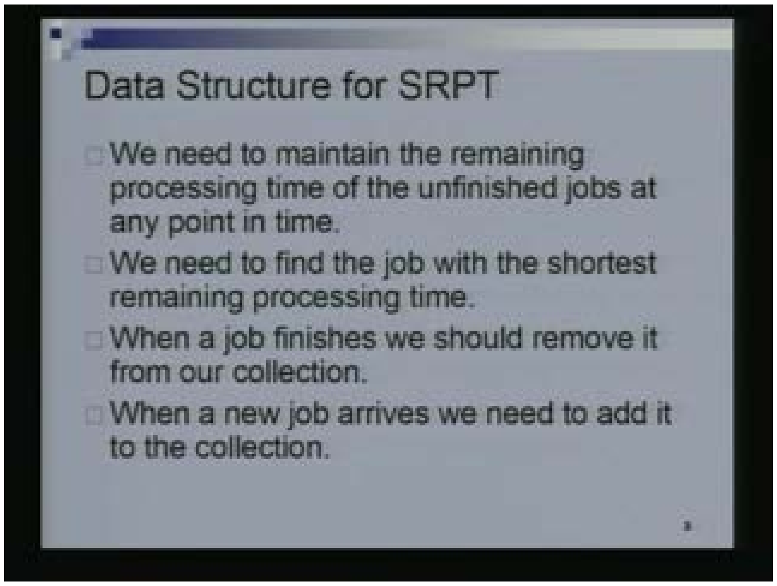
- In a multi-user computer system, multiple users submit jobs to run on a single processor.
- We assume that the time required by each job is known in advance. Further, jobs can be preempted (stopped and resumed later)
- One policy which minimizes the average waiting time is SRPT (shortest remaining processing time).
- The processor schedules the job with the smallest remaining processing time.
- If while a job is running a new job arrives with processing time less than the remaining time of current job, the current job is preempted.

2

We assume that when a job arrives, we know in advance how much time the job is going to take on the particular system that we have. Further we are going to assume that the jobs can get preempted that is that job is running, it can be stopped some other jobs can be schedule on the processors and then later this job which was stopped can be resumed. One policy which minimizes the average waiting time of job is what is called the srpt. The srpt rule, its stands for shortest remaining processing time. What this policy says is that the processor schedules the job which is the smallest at any point in time which has smallest remaining processing time at any point in time.

So to begin with this processor has the bunch of job, it will take the smallest job amongst that bunch and schedule it. May be the job finishes then it picks up the next smallest jobs in this collection it has and schedules that one but suppose you have a job which is running and then in mid way some other job comes in which has the processing time which is smaller than the remaining processing time of this job, the one that is currently running. If such is a case then we are going to interrupt the currently running job and schedule this new job that's given. This is the setting in which interruption happens in which preemption happens. This is just a motivation we are not going to be looking at this srpt rule any more carefully than this but if you were to implement this kind of a policy, what kind of a data structure would you need that's what we are going to look at.

(Refer Slide Time: 04:02)



Data Structure for SRPT

- We need to maintain the remaining processing time of the unfinished jobs at any point in time.
- We need to find the job with the shortest remaining processing time.
- When a job finishes we should remove it from our collection.
- When a new job arrives we need to add it to the collection.

3

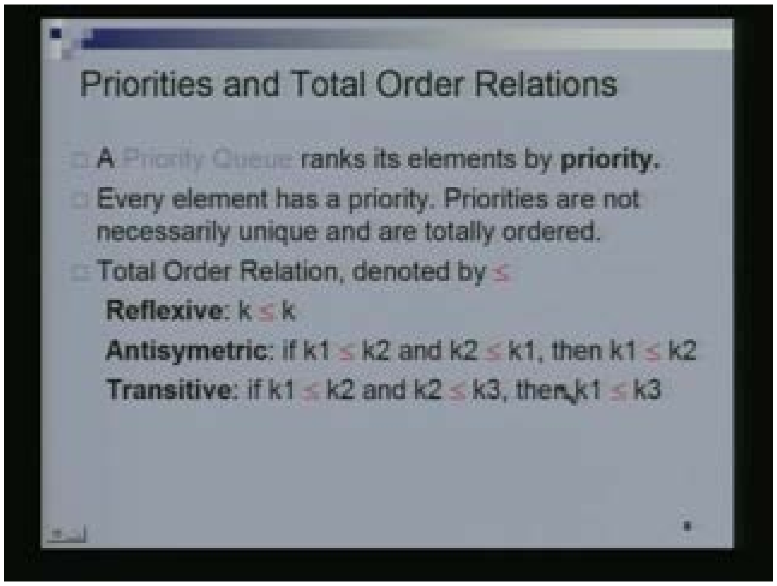
First we some how need to maintain the remaining processing time of the unfinished job, jobs that have not been completely finished. They are still with us, they still need to be scheduled for some period of time. We need to maintain that collection. Amongst these set of jobs, we need to find out the one which has the shortest remaining processing time because we need that so that according we can schedule the job according to the rule that we are following. When a finishes we would like to remove it from this collection and when a new job arrives we would like to add it to this collection.

These are the kind of operations that we need to do on our data structures to be able to implement the srpt policies. One data type which will let us do all of this, what's called the priority queues. It is data type to maintain a set of elements each of which has an associated priority. The priority queue data type supports the following operations. The first operation is one of inserting an element. The x is an element, we are going to insert it to our collection s, so that new collection is now going to be s union x. Of course this element has certain priority. When we are inserting the element, we will also have to take care of the priority of that element. The minimum operation returns the element of s with this smallest priority.

Priority perhaps is not the best word here because when you say each element has a priority, you would like to say that you would like to get the element of the largest priority. It's a bit of misnomer here but that's what we will work with, so in the scheduling example it made sense to remove the one with the minimum processing time remaining. We will continue with the minimum, the priority really is the misnomer here and then we have the delete min operation. The delete main operation returns the element with the minimum priority and also removes it from the collection.

The way minimum and delete min differs is that while minimum only returns the minimum element it doesn't delete it from the collection, delete min also deletes it from collection. These are the three basic operations that the priority queue is supposed to support. Of course there could be many other operations, like for all other container classes that we have seen so far we have seen methods like is empty, is size and so on or size, is empty and so on. One thing that we require on the priorities is a total order, let me come to that.

(Refer Slide Time: 08:11)



Priorities and Total Order Relations

- A Priority Queue ranks its elements by **priority.**
- Every element has a priority. Priorities are not necessarily unique and are totally ordered.
- Total Order Relation, denoted by $\leq$

  Reflexive: $k \leq k$

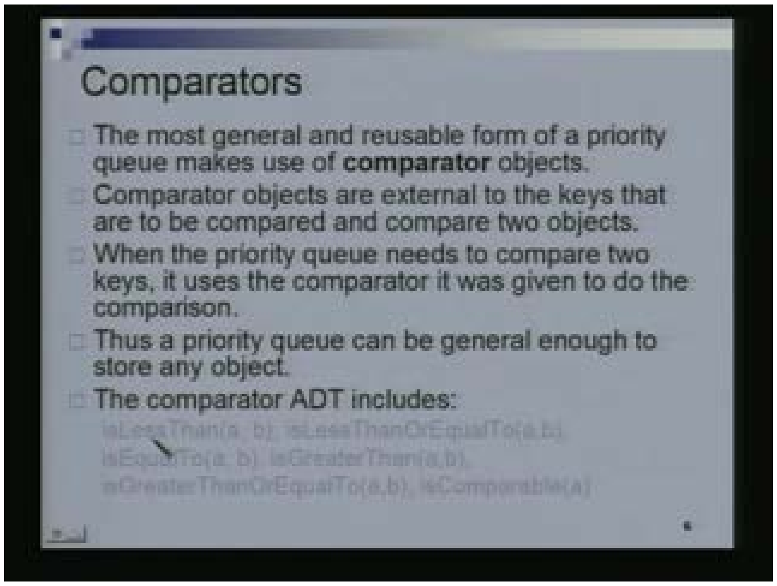
  Antisymetric: if $k1 \leq k2$ and $k2 \leq k1$, then $k1 \leq k2$

  Transitive: if $k1 \leq k2$ and $k2 \leq k3$, then $k1 \leq k3$

Recall the priority queue ranks its element by priority. Every element has a priority, priority need not necessarily be unique but they are totally ordered. That is given two priorities I can compare them, I can decide whether one is smaller than the other or larger than the other or they are both equal. There is the total order which will lets say denote by less than or equal to on the priorities. These priorities need not be integer, they could be anything at all but there is definitely some kind of a total order on these priorities. In particular this relation less than or equal to is reflexive that is priority k is less than or equal to k. it is antisymmetric which means that if k1 is less than k2 and k2 is less than k1 then k1 is also less than or equal to k2. The k1 is equal to k2, this is the error here and k1 k2 should be the same and its transitive. If k1 is less than k2 and k2 is less than k three then k1 less than k3.

These are the properties of total order relation and we will assume that the priorities that we have satisfies the total order deletion. The most general and reusable form of priority queue uses comparator objects. What do I means by this? Recall that we said that in a priority queue, each element has priority associated with it and the priorities there is a total order on the priorities. If I want to put any kind of an object in to my priority queue, I should have a mechanism of comparing the priorities of the objects in the queue.

(Refer Slide Time: 10:05)



Comparators

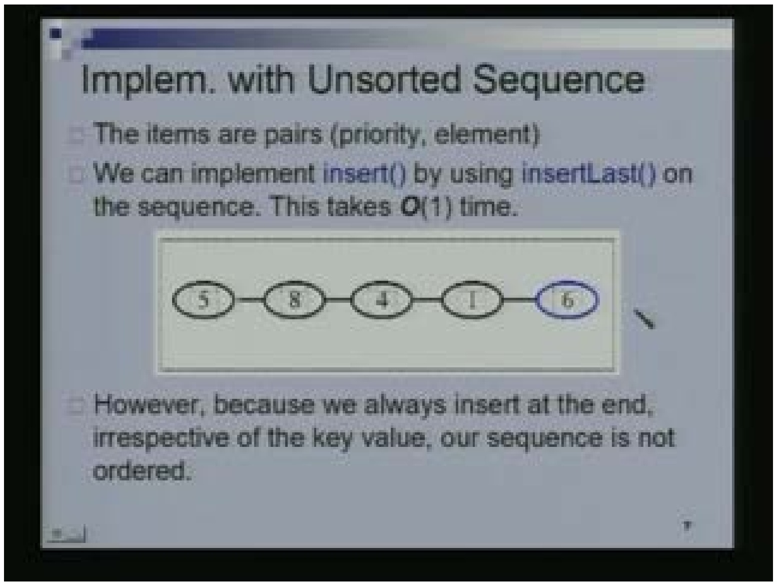
- The most general and reusable form of a priority queue makes use of **comparator** objects.
- Comparator objects are external to the keys that are to be compared and compare two objects.
- When the priority queue needs to compare two keys, it uses the comparator it was given to do the comparison.
- Thus a priority queue can be general enough to store any object.
- The comparator ADT includes:

  isLessThan(a, b), isLessThanOrEqualTo(a,b), isEqualTo(a, b), isGreaterThan(a,b), isGreaterThanOrEqualTo(a,b), isComparable(a)

We can have such a comparator object which will help us to do this comparison. So that comparator object all it does is it specifies the methods which will help us compare to priorities. In this manner we can ensure that our priority queue that we would have one implementation of priority queue and we can use same implementation to store any kind of object because we would also specify this comparator object which would help us compare the priorities of any two object that we decide to put in the collection.

The comparator abstract data type would include methods like is less than, is less than or equal to, is equal to, is greater than, is greater than or equal to and whether it is comparable or not. Let’s first look at an implementation of priority queue using an unsorted sequence. Recall the items that we are trying to insert in to our sequence or pairs of priority and our element. We can implement the insert by using just say insert last.

(Refer Slide Time: 11:03)



Implem. with Unsorted Sequence

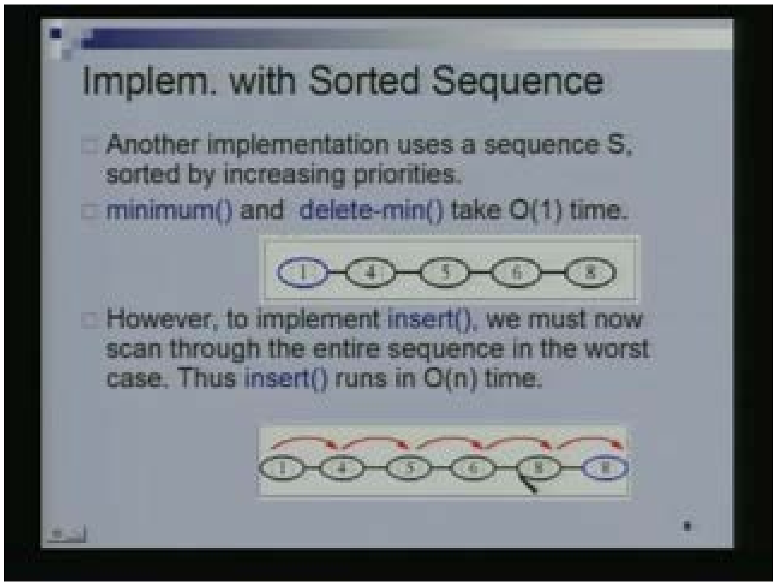
- The items are pairs (priority, element)
- We can implement insert() by using insertLast() on the sequence. This takes **O**(1) time.

- However, because we always insert at the end, irrespective of the key value, our sequence is not ordered.

The insert operation of the priority queue, to do this we will just insert the element that is the item at the very end of our sequence. In this case six was the element that was to be inserted. Let’s say we went and put it at the very end. This takes only the constant amount of time but if I do such a thing, if we always insert at the end irrespective of the value of the priority then our sequence is not ordered any more. Since it is not ordered according to the priorities, we are going to have problems when we are trying to find a minimum or when we are trying to delete the minimum element, the element with the minimum priority from this collection. In that case we will have to look at all the elements of these sequences. For instance if I have to find the minimum then I would have to start right from here and traverse the sequence till I reach the element with the minimum priority which is let’s say one here. That’s to find the minimum.

Similarly for delete min because I will have to get to here, I will have to remove the element and then I would have to find out the new minimum element. So worst case time complexity of minimum and delete min therefore becomes ordering. That’s not very good, here all the insertion takes constant time we are saying that both for minimum and for delete min we are going to take order n time. How about using the sorted sequence to try and implement a priority queue.

(Refer Slide Time: 13:20)



Implem. with Sorted Sequence

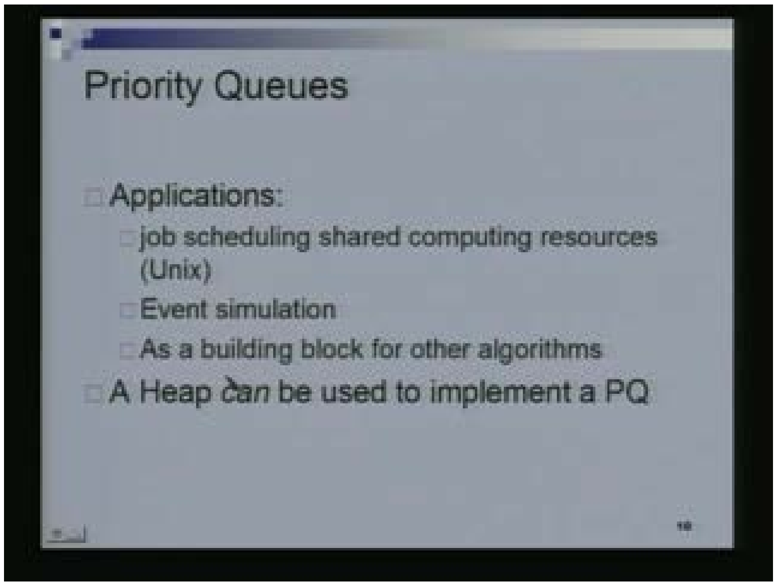
- Another implementation uses a sequence S, sorted by increasing priorities.
- minimum() and delete-min() take O(1) time.
- However, to implement insert(), we must now scan through the entire sequence in the worst case. Thus insert() runs in O(n) time.

1
4
5
6
8
1
4
5
6
8
8

Now we are going to have a sequence in which the elements are sorted with increasing priorities that is the elements with the smallest priority is straight sitting right at the front of our sequence. In which case minimum and delete min I am just going to take a constant amount of time. The minimum element as I said is the element at the front of our sequence. So you just be able to retrieve the minimum element in constant time.

Similarly to delete it, you just have to delete the minimum element and then delete the minimum element from the front of the sequence. This is what your sequence would look like now, note that the element with the minimum priority is sitting right at the front here but now the problem is if you have to do the insert operation. If you have to insert lets say an element with priority 7, I will have to traverse the sequence till I come to the right position and put the element there. In the worst case insert could now take order end time. In this example I am inserting 8 which can be inserted either before this 8 or after this 8. In this case I am taking time proportional to the length of the sequence. Priority queues find many applications.

(Refer Slide Time: 13:34)



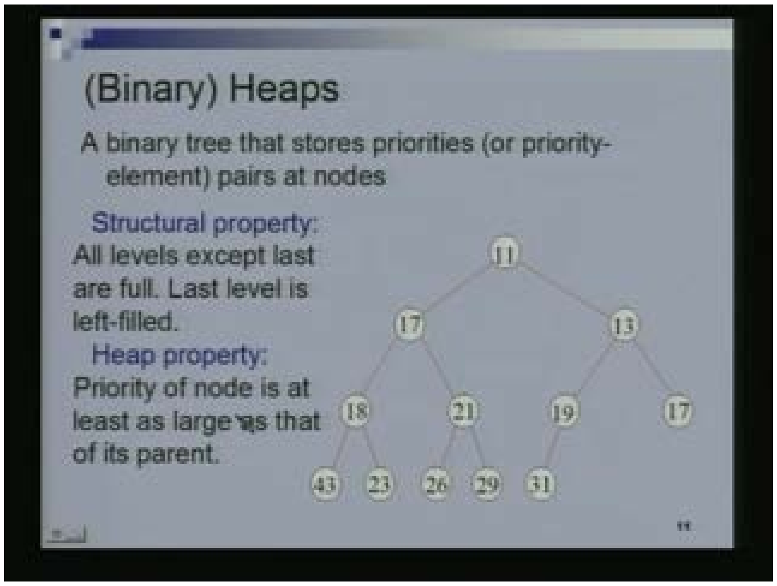
Priority Queues

- Applications:
  - job scheduling shared computing resources (Unix)
  - Event simulation
  - As a building block for other algorithms
- A Heap *can* be used to implement a PQ

I gave you an example of scheduling. They are also used in discreet event simulation and they are used as a building block for many other algorithms. we are going to be seeing more algorithm in this course lets says dijkstra's algorithm which require the priority queue data structure to be able to efficiently implement the algorithm. Today what we are going to see is how to use a data structure called a heap, to implement a priority queue.

We saw the two methods to implement the priority queue and unsorted sequence and sorted sequence and both of them had their problems. In case of unsorted sequence the delete min operation and minimum operation would take order n time and in the case of a sorted sequence, the insert operation would take order in time while the other operations are constant time operation. We would like to have a data structure which would do all these three operations insert, delete min and minimum very efficiently.

(Refer Slide Time: 14:39)



(Binary) Heaps

A binary tree that stores priorities (or priority-element) pairs at nodes

Structural property: All levels except last are full. Last level is left-filled.

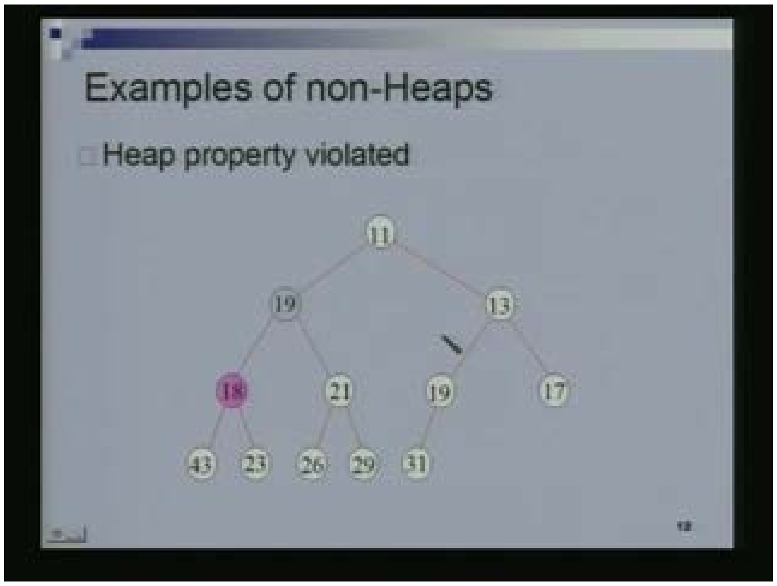
Heap property: Priority of node is at least as large as that of its parent.

What is a heap? When we say heap we typically mean what's called a binary heap. Binary heap is a binary tree that stores the priorities or the priority element pairs at the nodes. In the rest of the class I am just going to show the nodes as containing the priority of the element and the element is sitting somewhere here also. They could be storing only priorities of the priority element pairs. Now heap has to two properties. One is the structure properties. In this binary tree all levels are full. This is level zero, it can have only one node, it has one node. Level 1 can have 2 nodes, it has 2 nodes. Level 2 can have 4 nodes, it has 4 nodes. Level 3 can have 8 nodes it has only 5 nodes.

So all levels except the last levels are full and the last level is what we will call left fill which means that all the nodes on the last level are as left as possible. They could be 8 elements there. There only 5 but these 5 nodes would be the ones which are the first five if I were to move left to right. That's the structural property of a binary. The other properties what I call a heap property. The heap property simply says the following. The priority of the node is at least as largest that of its parent. For any node its priority has to be larger than the priorities of its parent.

This picture here that satisfy both of the properties, structure as well as heap. Consider any node, its priorities is at least as larger then the priorities of its parent. You can take any node. I can restate this, I can also say it in the following manner. The priority of any node is less than or equal to the priorities of its children. Sometimes we will also think of it in this manner, priority of any node is less than or equal to the priorities of its children. There might be two children, there might be only one child as is the case here. In this case priority of this node has to be less than the priority of its lone child. Let's look at example which are not heap.

(Refer Slide Time: 17:01)

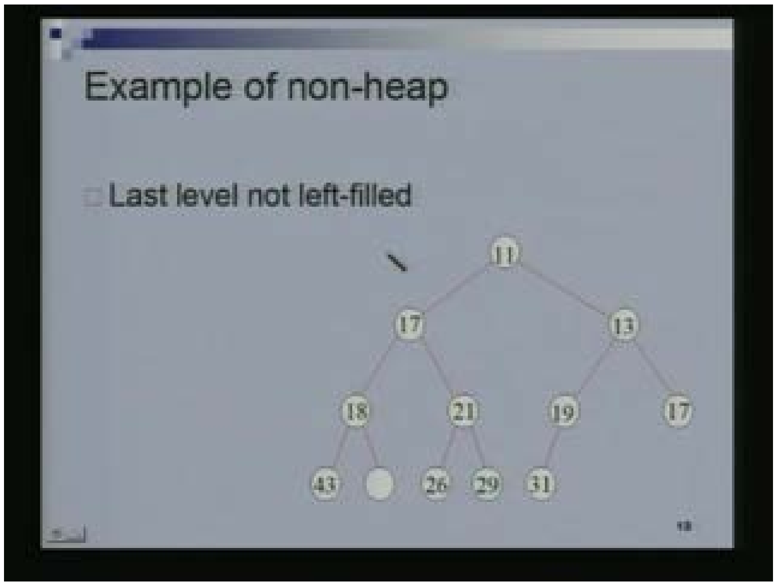


Examples of non-Heaps

- Heap property violated

This is not a heap because if you look at this node, we require that it have a priority which is larger than the priority of its parent, at least as large as the priority of its parent but its priority is 18, its priority is 19. This violates the heap property. This does not violate the heap property but as you can see this node is empty, so this last level is not left filled. We would like that if there were only 4 nodes at the last level then they should appear as the first 4 nodes of this binary tree.

(Refer Slide Time: 17:24)



Example of non-heap

- Last level not left-filled

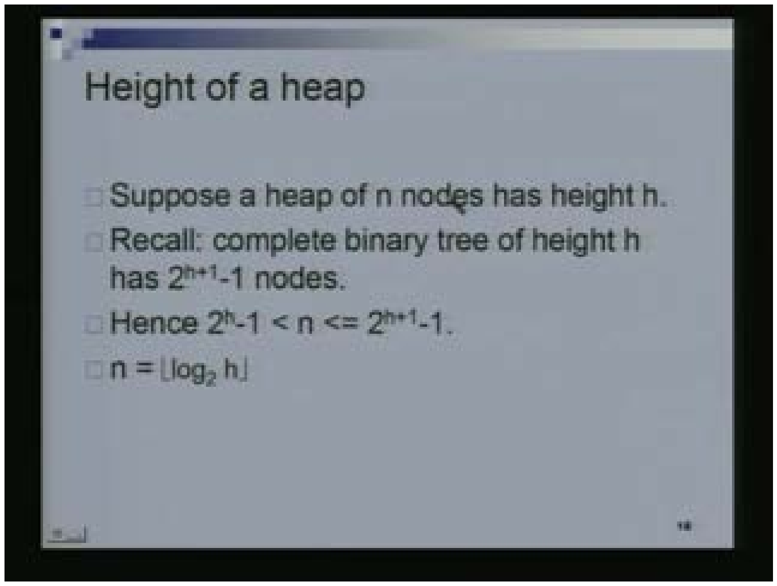
This appears as the first four nodes of the last level of this binary tree (Refer Slide Time: 17:51). How does one find the minimum element in a heap? Suppose I give you a heap. What's a heap? Recall as this two properties, one is the structural property the other is the heap property.

Suppose I give you a heap, how does one found the minimum element? The claim is the minimum element, the element with the smallest priority always sits at the top of the heap. The top of the heap I mean the node here at the root and as you can see such as the case in the picture. The element with this minimum priority is the element by 11 and its sitting in top of the tree.

Why is this the case? Why can’t the minimum not be some where else? If it is not at the root of the heap or at the top of the heap, it could be somewhere in the middle, somewhere inside the heap but then it has a parent and the heap property says that the priority of the parent is at least as large as the priority of the particular node then the parent would have a larger priority. The heap property says that the priority of the parent is no larger than the priority of the node. Now if such is the case then the parent would have even smaller priority and so the element that I was talking of was not the one with the smallest priority.

Just recap what I said, if the element with the smallest priority was sitting else where in the heap then it would have a parent with smaller priority and this would violate the heap property. This larger should really leads smaller, it would have parent with the smaller priority and that would violate the heap property. The minimum can be done in constant time because we just need to go to the root element to find the element with the smallest priority. What is the height of the heap? Suppose I give you a heap on n nodes, what is its height going to be?

(Refer Slide Time: 20:08)



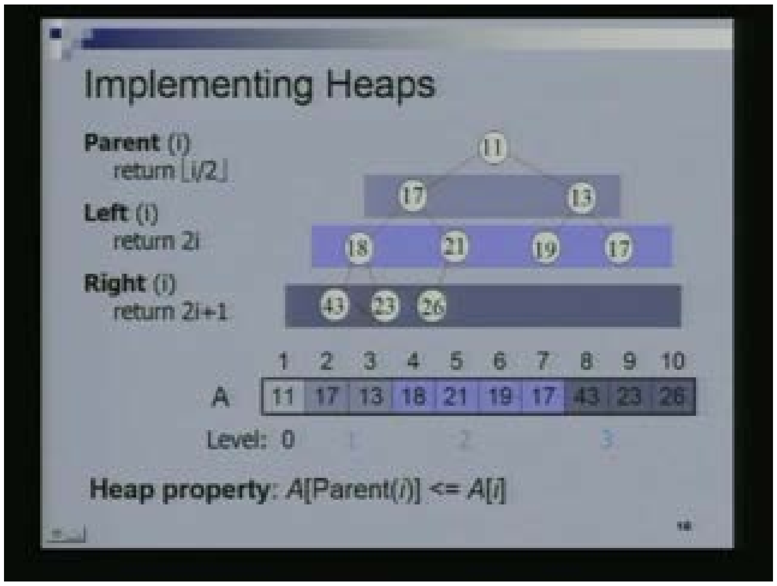
Height of a heap

- Suppose a heap of n nodes has height h.
- Recall: complete binary tree of height h has $2^{h+1}-1$ nodes.
- Hence $2^h-1 < n <= 2^{h+1}-1$.
- $n = \lfloor \log_2 h \rfloor$

Recall our discussion on binary tree, if you have a complete binary tree of height h then it has exactly two to the h plus one minus one nodes. If we have a heap of n nodes with height h, note that since the height of the heap is h, it has more nodes than a complete binary tree of height h minus one. A complete binary tree of height h -1 has so many nodes in it. The number of nodes in the heap is larger than the number of nodes in the complete binary tree of height h -1.

A complete binary tree of height h has 2 to the h plus one minus one nodes and the number of nodes in the heap is less than or equal to this quantity. This inequality holds, n is strictly larger than the two to the h minus one and less than or equal to two to the h plus one minus one. I can replace this as n equals the floor of the log of h. This notation here really means that log of h which need not be an integer, we are just rounding it down to the nearest integer. Hope this is clear to everyone. This is the mistake, this should read h equals log of n. The height of a heap is the log of the number of nodes. Just interchange this h and n, this is error on this slide. If n lies between two to the h and two to the h plus one minus one then h is log of n. How does one implement the heap?

(Refer Slide Time: 22:04)



Implementing Heaps

Parent (i)
  return $\lfloor i/2 \rfloor$

Left (i)
  return 2i

Right (i)
  return 2i+1

	1	2	3	4	5	6	7	8	9	10
A	11	17	13	18	21	19	17	43	23	26

Level: 0 1 2 3

Heap property: $A[\text{Parent}(i)] \le A[i]$

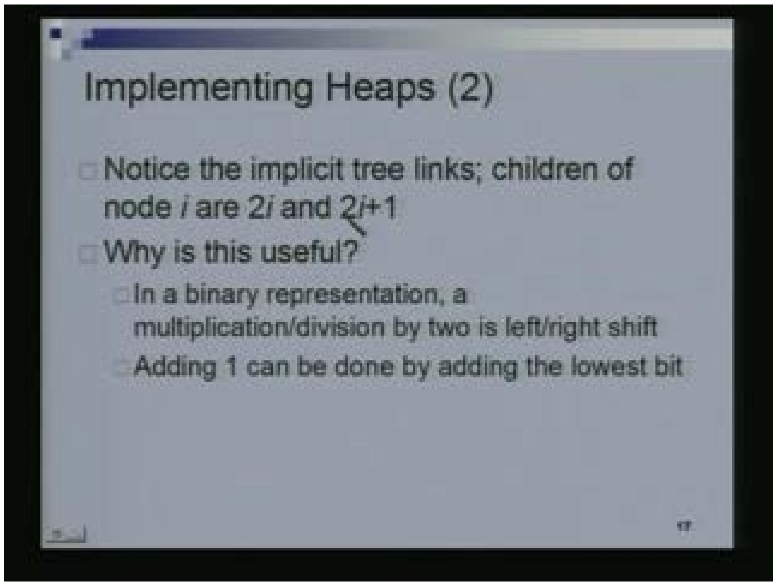
Recall that till now we were saying that heap is this kind of a binary tree. We can of course implement the heap using the binary tree but it’s much simpler to implement the heap using an array. Let’s see what I mean here. This is level zero of my heap, this is level one, level two and level three of the binary tree corresponding to this heap. I am going to put these nodes of binary tree in to an array with the root node sitting at the first location in the array. My array is going to be indexed starting from location one. So 11 comes here 17 and 13, the next level would follow and we would go left to right. 17 would be the next element, 13 would be one after that then 18, 21, 19, 17, 43, 23, 26.

This is level zero of the binary tree, this is level one of the binary tree, this is level two, this is level and this is level three of the binary tree. What’s the advantage of putting the nodes of a heap in to an array like this? Suppose I look at a node at location five lets say, node 21 and I want to find out the parent of this node in the heap. Then all I have to do is take 5 divided by 2 take the floors of 5 by 2 floor is 2. The parent of 21 in this heap is going to be 17 which is the case here. We can do it for any other node let’s take node 26. The 26 is at location 10, 10 by 2 is 5 whose floor is also 5. The parent of 26 that location 5 and is 21. You can quickly determine the parent of any node, if you know the location of the node. Parent is just at location i by two if the node is at location i.

Conversely if the node is at location i and I want to figure out its two children then the two children, the left child would be at location two i then the right child would be location two i +1. We can work out this math it will be very easy, I will just show it with an example here.

If I look at node thirteen it's at location three, its two children then should be location 6 and 7. Two children are 19 and 17 and they are at location 6 and 7. Given the location of any particular node, I can just quickly figure out what is the location of the parent, what are the locations of the two children of that node. Those are the only thing that are really need to do in heap and the heap property is really saying that if i is the location, parent of i is the location of its parent and a of parent of i, a is the array which is holding these priorities. So its priority of the parent of i. The priority of the parent is less than or equal to the priority of the node itself. That's what the property is. We implicitly are maintaining the tree links. The parent child relationships are getting implicitly maintained. the children of a node i are at location two i and two i plus one and this is very useful in binary, in multiplying by two just corresponds to a left shift and multiplying by two and adding one is equally simple.

(Refer Slide Time: 25:53)

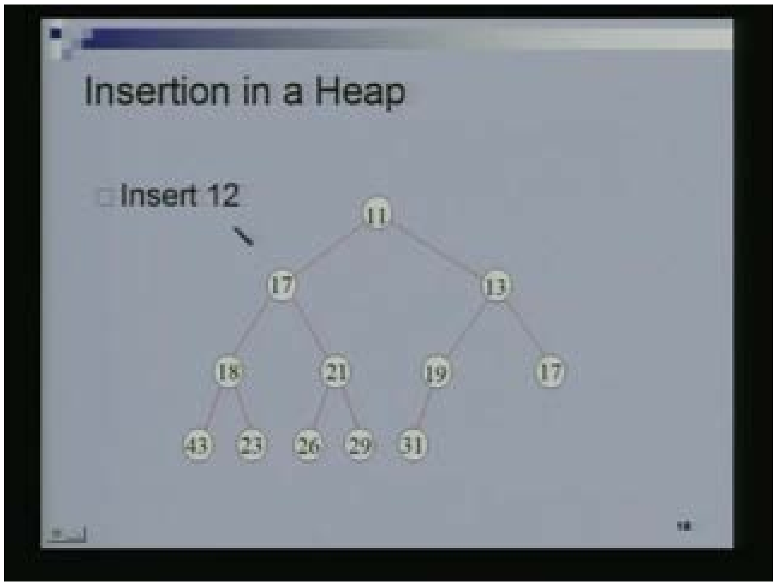


Implementing Heaps (2)

- Notice the implicit tree links; children of node *i* are 2*i* and 2*i*+1
- Why is this useful?
  - In a binary representation, a multiplication/division by two is left/right shift
  - Adding 1 can be done by adding the lowest bit

One can quickly figure out the children of a node or also the parent of the node by just doing one left or right shift and an increment operation. This makes the operation really fast, when you trying to access the parent or the children of a particular node. Let's now see how one would insert an element in to the heap.

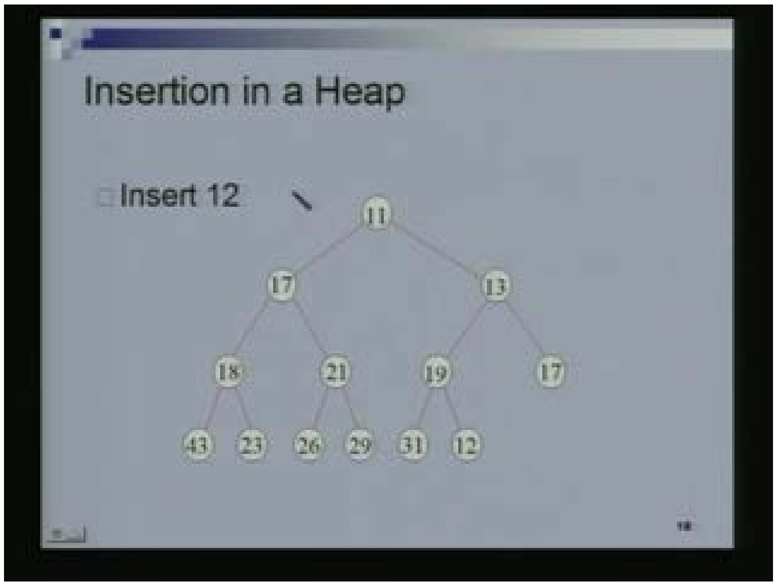
(Refer Slide Time: 26:54)


Insertion in a Heap

- Insert 12

Till now the only operation we have seen is to find the minimum element in the heap and that we said can be done in constant time and because minimum element in a heap always sits at the root of the heap. Suppose I wanted to insert twelve in to this heap. What would I do? Right now this heap as 3, 7 and 12 nodes in it. If I insert 12 this heap is going to have 13 elements in it. If it has 13 elements in it then structurally I should have another node here. That’s the first thing I would do. I would first create a node here and let me now put 12 in to it.

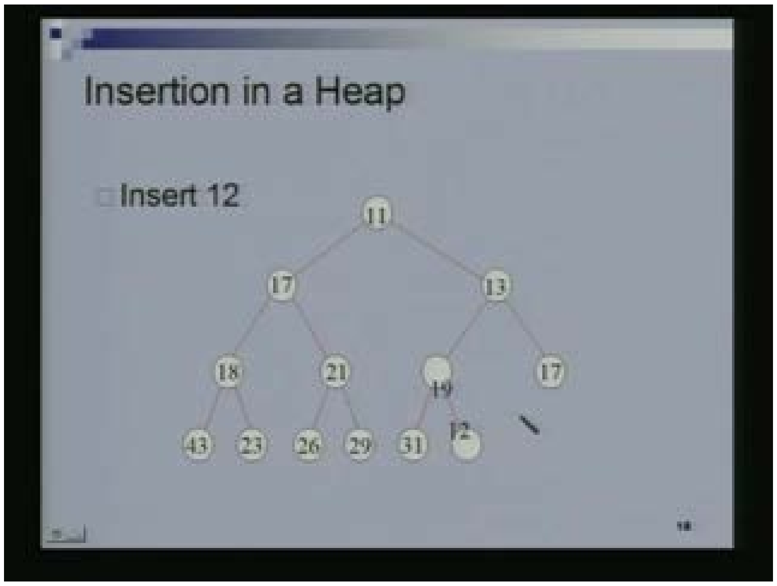
(Refer Slide Time: 27:28)


Insertion in a Heap

- Insert 12

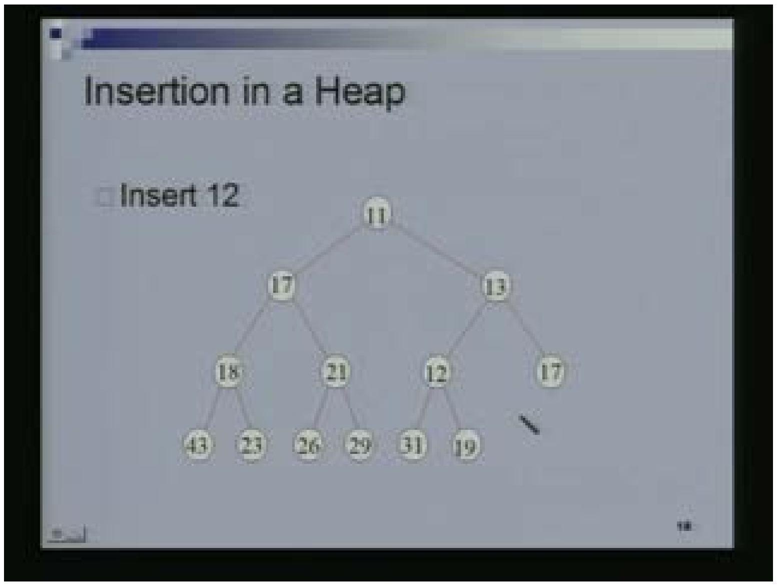
Of course structurally this does satisfy the structural property of heap but it is not heap because it violates the heap property. This node should have lesser priority than its two children it does not, it has higher priority than this child. We need to take care of that. Since this is lesser priority than this we need to swap this thing.

(Refer Slide Time: 27:55)


Insertion in a Heap

- Insert 12

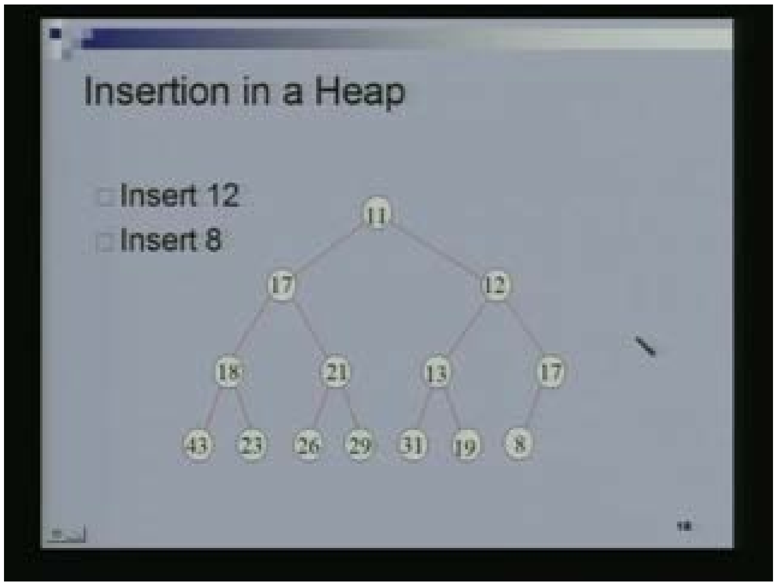
(Refer Slide Time: 28:08)


Insertion in a Heap

- Insert 12

We do that in this manner we swap this two element. Now the heap property is satisfied here, no problem. This has lower priority than its two children but it's violated here. This does not have lower priority than its two children. We should then swap this and this, we do that. Heap property also now satisfied here. Since we change the content of this node, could it be that the heap property is violated at this node. Let's look this has priority eleven, this has 17, this has 12. This has lesser priority than both its children, the heap property is satisfied here no problem at all. We are done with the insertion of 12. All we did was create a new node, put twelve here and then move twelve up, so as to maintain the heap property. We saw that 12 had a lower priority than its parent. So we swap 12 and its parent then once again we saw that 12 had reached here. 12 had a lower priority than its parent so once again we swapped 12 and its parent. 12 now reached here, but now 12 does not have lower priority than its parent, so we stop. Let's see if we were to insert another element. Let me insert 8 in to this heap.

(Refer Slide Time: 29:24)

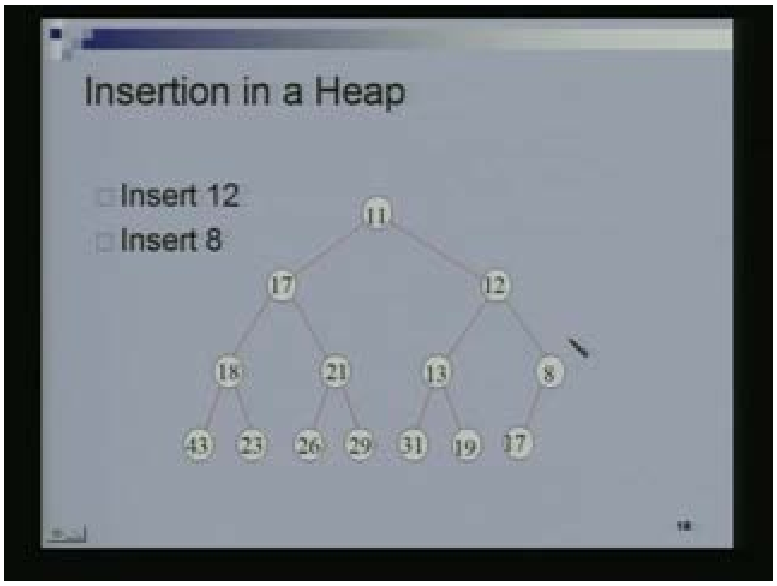


Insertion in a Heap

- Insert 12
- Insert 8

Once again 8 would come at this location, so I have put in 8 here. 8 has a lower priority than its parent this violates the heap property, so we would swap 8 and 17.

(Refer Slide Time: 29:34)

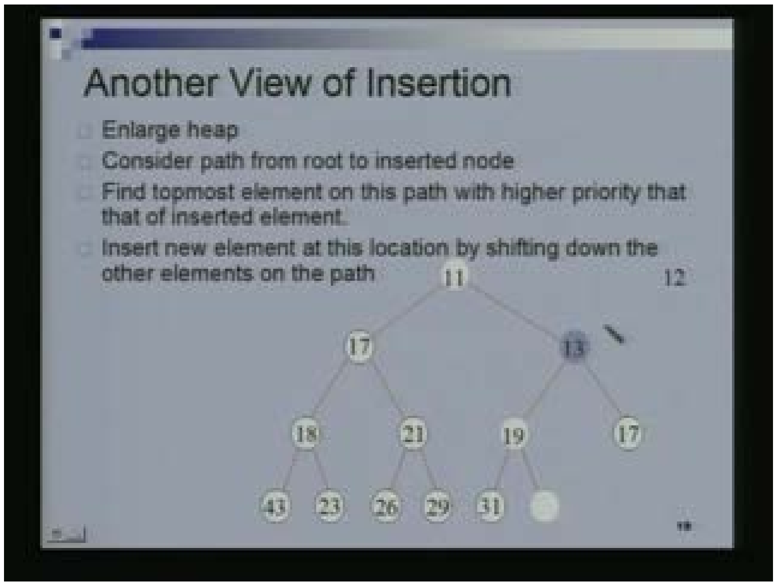


Insertion in a Heap

- Insert 12
- Insert 8

Once again 8 has the lower priority than its parent, so we would swap 8 and 12. Once again this has lower priority than its parent so we swap 8 and 11 and now we were done. This is now a heap with 8, the element that we inserted last. Now coming at the root of the heap and in fact it should come there because this is the minimum priority element in the entire thing. Let's look at insertion again from the slightly different point of view.

(Refer Slide Time: 31:48)



Another View of Insertion

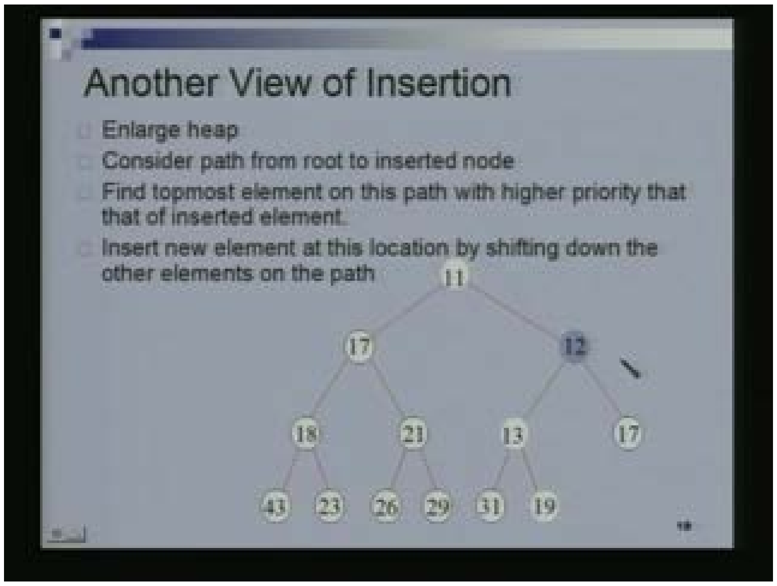
- Enlarge heap
- Consider path from root to inserted node
- Find topmost element on this path with higher priority that that of inserted element.
- Insert new element at this location by shifting down the other elements on the path

11
12
17
13
18
21
19
17
43
23
26
29
31

We again have the same heap as before, we are trying to insert 12. First we enlarge the heap which means that we create the new node where we create the new node because we know that the new heap now has 13 elements in it and so this is what it structure should be like. Now we consider the path from the root to inserted node which is this slight colored path. So this was the inserted node and we just march up, we just follow, go from this node to its parent, to its parent and to its parent. This is path from the root to the inserted node and on this path, we find the top most element with the priority higher than the priority of the inserted element. What are the nodes on this path? These are these three nodes, they have priorities 19, 13 and 11.

The top most element on this path with higher priority, the element we are inserting is priority 12. This is the top most element but it has priority lesser than 12. This is the next highest element and this has priority more than 12. This is the top most element of the path with higher priority than that of the inserted element. This is the element, so what are we going to do now. We are going to insert 12 at the location but where would 13 go? 13 would get pushed down and where would 19 go? 19 would get pushed down like this.

(Refer Slide Time: 32:03)

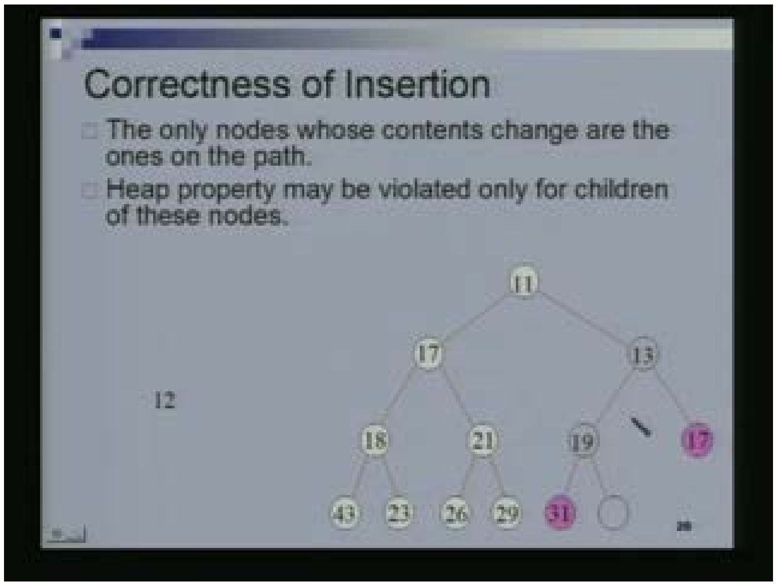


Another View of Insertion

- Enlarge heap
- Consider path from root to inserted node
- Find topmost element on this path with higher priority that that of inserted element.
- Insert new element at this location by shifting down the other elements on the path

This is exactly the same procedure as before. If you look at the slide that we got earlier, the elements where at the same location except there we started 12 here and we bubbled it up. Now I say that I am going to directly figure out where 12 is going to come and I am going to move elements down. Let's argue correctness. Let's argue that the procedure of the insertion is really correct and for this we look at this other view of insertion that we saw in the previous slide.

(Refer Slide Time: 33:37)

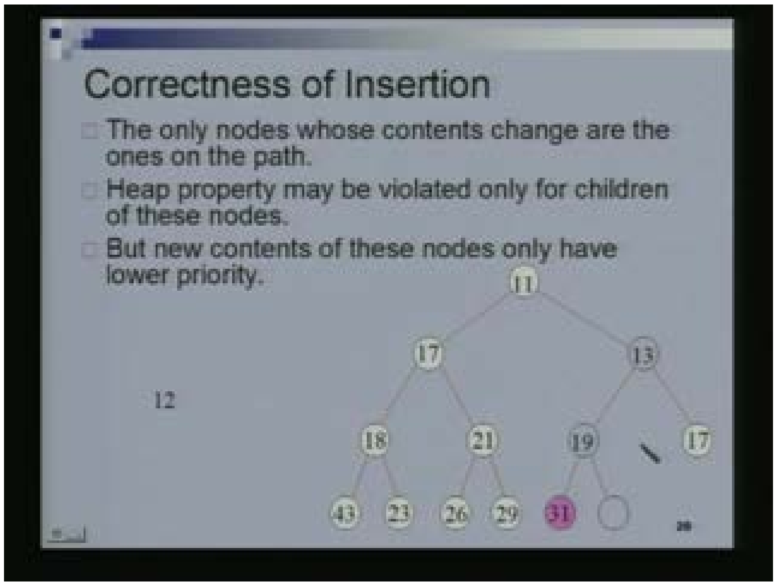


Correctness of Insertion

- The only nodes whose contents change are the ones on the path.
- Heap property may be violated only for children of these nodes.

The first claim is that the only nodes whose contents changes on the ones on this path. This is the path that we considered which is the path from the newly inserted node to the root and we are changing only the contents of this. We are in fact only changing the contents of this part of the path. If the heap property is getting violated, the heap property can be violated only at the children of these nodes. It could that for this node, since we are modifying this content it might be that the new priority of the parent is larger than the priority of this guy or new priority of the parent of this node is larger than the priority of this node that would be violation of the heap property. So it is only for these two pink color nodes that the heap properties might be violated.

(Refer Slide Time: 33:45)



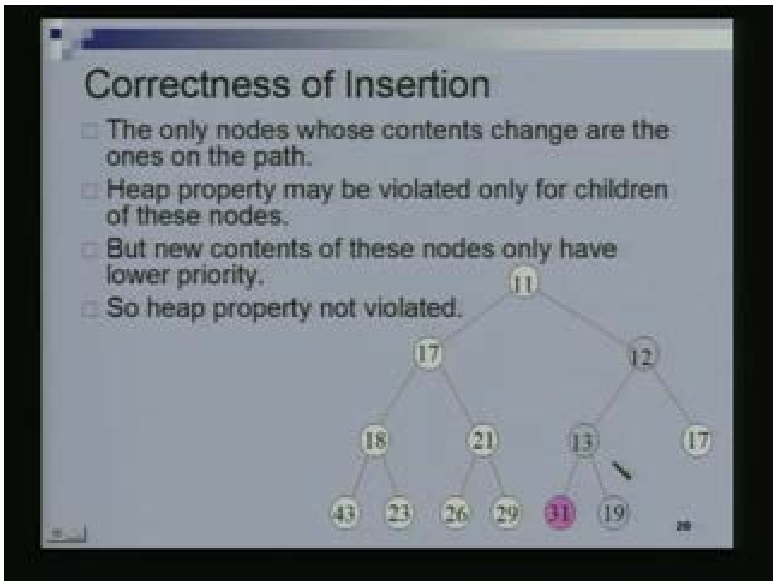
Correctness of Insertion

- The only nodes whose contents change are the ones on the path.
- Heap property may be violated only for children of these nodes.
- But new contents of these nodes only have lower priority.

What is happening? What are the new contents of the parent of 31 and 17? 31 is 19, what is the new contents of this going to be? If you recall 19 moved down, 13 was coming here and 12 was reaching there. The new content of this node is 13 which is only going to be less than this. The new content of any node are only going to be either the newly inserted element or the parent of that node. The new content of this is going to be the content of its parent and the content of the parent is already smaller and has the lower priority that is the priority of this guy.

The priority of each node is only going to be reduce, so the priority of these guy is going to reduce because 13 is going to move here. The priority of this guy is going to reduce because newly inserted element 12 is going to come here and recall from our choice of path that this was picked as a node with higher priority than the priority of the element that was getting inserted. This is higher than this. The priority of this is also going to reduce as the consequence of insertion procedure. The priority of all of these nodes only going to reduce the consequence of insertion procedure and ones the heap property would not get violated.

(Refer Slide Time: 35:56)



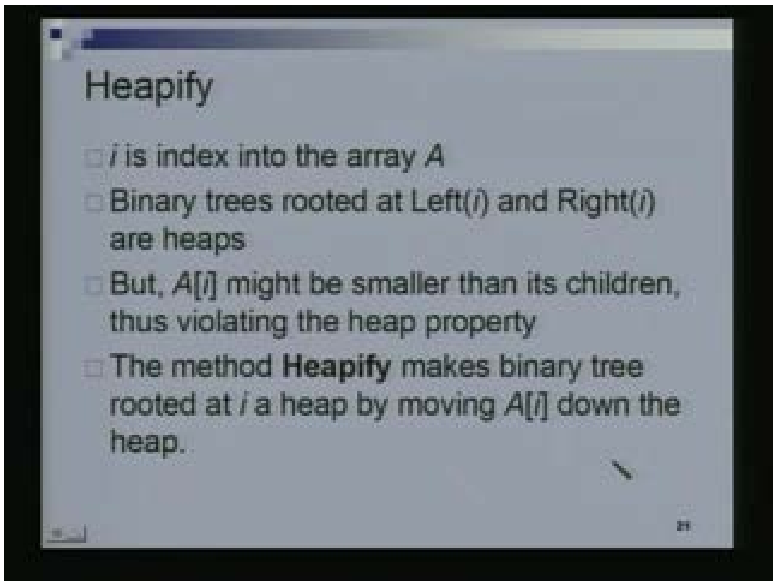
Correctness of Insertion

- The only nodes whose contents change are the ones on the path.
- Heap property may be violated only for children of these nodes.
- But new contents of these nodes only have lower priority.
- So heap property not violated.

These are going to be the new priorities of these nodes. We said that heap property could be violated for this guy or for this guy. This should also have been pink, it could be violated for this or this but earlier its parent had a priority 19. Now it has a priority 13. Earlier the parent of this guy had a priority of 13 now it has a priority of 12, the priority is only reducing. If there were only reducing then earlier if this had a priority which was smaller than the priority of this guy, it continues to have priority which is smaller than the priority of this guy.

If earlier this had a priority which is smaller than that of this guy, it continues to have a priority which is smaller than the priority of this guy. Heap property is not violated at all and so what we get after insertion is still a heap. I am now going to look at another procedure called heapify which we are going to use as I mentioned at the beginning of the class, we are going to use for the other methods that we have to do on a heap.

(Refer Slide Time: 36:04)



Heapify

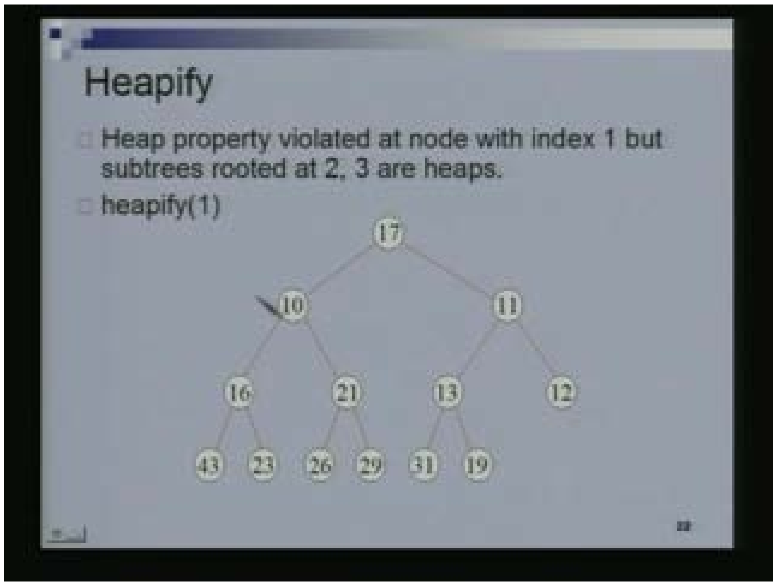
- *i* is index into the array *A*
- Binary trees rooted at Left(*i*) and Right(*i*) are heaps
- But, *A*[*i*] might be smaller than its children, thus violating the heap property
- The method **Heapify** makes binary tree rooted at *i* a heap by moving *A*[*i*] down the heap.

21

I am now assuming that my heap is kept in an array A and i is an index in to this array so heapify take the parameter as an index in to the array A. Then the binary tree is rooted at the two children of i, left i and right i are the two children of i. The binary trees rooted at these two locations are already heaps but it might be the case that the heap property is violated at this node i that is A[i] might be larger than its children. A[i] might be larger than its children and this violates the heap property.

Now heapify is the method which tries to maintain the heap property, makes the binary tree rooted at i a heap by suitable modification and will see what this is in a second. Let's look at this structure. This is not a heap because this node right here at the very top has priority which is larger than the priorities of its two children.

(Refer Slide Time: 38:47)



Heapify

- Heap property violated at node with index 1 but subtrees rooted at 2, 3 are heaps.
- heapify(1)

But this binary tree, if I were to just restrict myself to this part which is a heap. Why it does satisfy the heap property? Every node has a priority which is smaller than the priority of its two children. This has smaller priority than its two children, this has smaller priority than its two children, this has smaller priority than its two children. Similarly this here is a heap, structurally as well as it does satisfy the heap property. Each node has priority smaller than its two children. These two are already heaps but this entire thing is not a heap because the heap property is getting violated at this node (Refer Slide Time: 38:23).

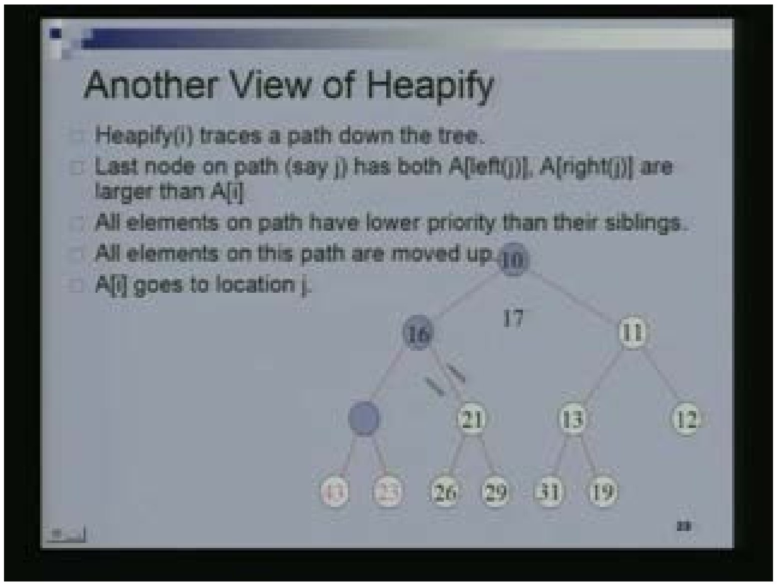
This does not have priority smaller than the priority of its two children and some how we want to make this entire thing a heap. We can invoke the heapify procedure. Heap property is violated at node with index one which is this node but the sub tree is rooted at two, two is this and three is this. So sub tree is rooted at two which is this and this are heaps. We would invoke the heapify procedure with one because we want to make this entire thing a heap. For heapify remember we require that whichever node heapify is invoked on, the sub tree is rooted at the two children of that node are already heaps. That's a very crucial part for heapify only then we would heapify it. If heapify is invoked at one, it can be invoked at one only because the sub trees rooted at two children of node one, this and this the sub tree is rooted here are already heaps.

Let's see how heapify works. Heapify is going to look at the two children of this node where the heap property is correctly violated. You have to look at the two children which are these 10 and 11 and it would take this smaller of these two children and swap it with 17.

The smaller is 10, it would swap 10 with 17. Now the heap property heap property is valid at this node. This has priority less than the priority of its two children but the heap property is violated at this node because this does not have priority smaller than the priority of these two children. Once again we are going to do the same thing. We are going to look at the two children, take the smaller one amongst them which is 16 and swap it with this. The heap property is now valid at this node but is the heap property valid at this node? (Refer Slide Time: 40:39) Because we also changed the content of this node. It is, because now when I look at the two children of this, they are 23 and 43 both of which have priority higher than 17 and so the heap property is violated here. Now this entire thing is heap, this is the heap because the heap property is now valid at every node in this tree. Recall, we have changed only the contents of these nodes. That's the heapify procedure.

Once again to recap, the heapify procedure can be invoked at the node i, only if the two children of this node or the sub tree rooted at the two children of this node are already heaps. Once again we will present the second view of heapify which will help us prove the correctness of heapify very easily.

(Refer Slide Time: 43:55)



Another View of Heapify

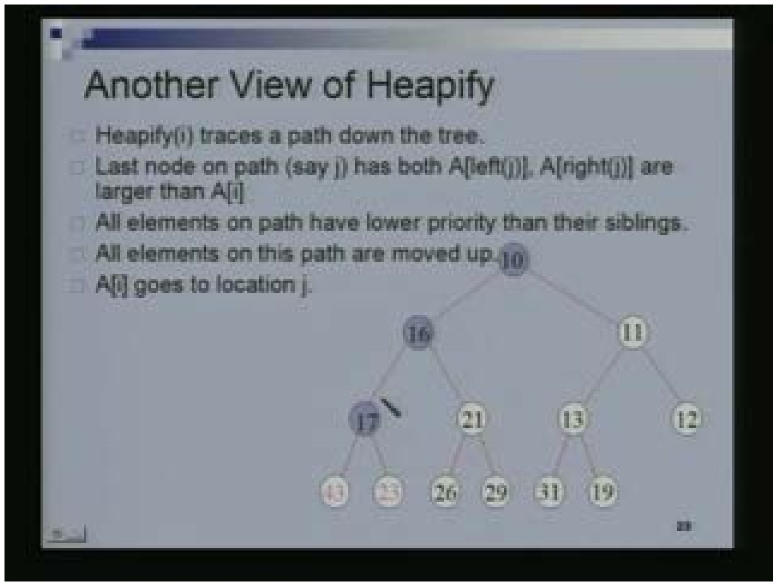
- Heapify(i) traces a path down the tree.
- Last node on path (say j) has both A[left(j)], A[right(j)] are larger than A[i]
- All elements on path have lower priority than their siblings.
- All elements on this path are moved up.
- A[i] goes to location j.

Heapify essentially is tracing a path down the tree. In our previous example, this path was this blue colored nodes. These were the node whose context we modified. If I look at the last node on the path suppose I call it node j then the left and the right child of j have priority which was larger than the priority of node i.

i is this node (Refer Slide Time: 42:22), i is in our example is one so 43 and 23 are both larger than 17. All elements on this path have lower priority than their siblings. What are the siblings? Sibling of 16 is 21, sibling of 10 is 11, 17 does not really have a sibling. 10 is less than 11, 16 is less than 21 and why is this true and why is that the case that the priorities for all nodes on this path have priorities less than the priority of the sibling. That's because of our choice of the path. We came left because we compared 10 and 11 and we picked 10, so that we could then swap it with here and then when we came here, we compared 16 and 21 and we picked 16.

Just from the way we constructed our path this statement really follows. What are they really doing in heapify? In heapify what we are really doing is that we are moving these elements up, let me show you how. I am just showing it in one step now but what we saw earlier was a sequence of step. This was the net result, 10 moved up, 16 moved up and this 17 really came at location j. It came at this location. This was the net result of our heapify step.

(Refer Slide Time: 44:00)



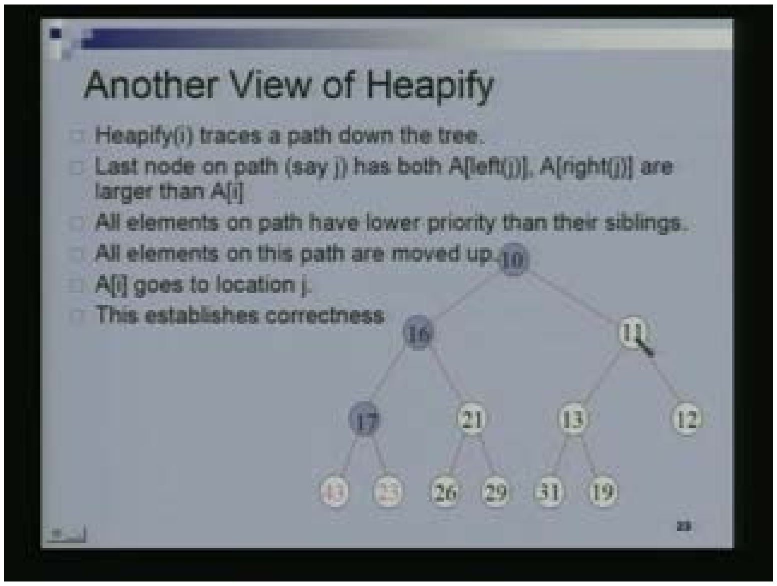
Another View of Heapify

- Heapify(i) traces a path down the tree.
- Last node on path (say j) has both A[left(j)], A[right(j)] are larger than A[i]
- All elements on path have lower priority than their siblings.
- All elements on this path are moved up.
- A[i] goes to location j.

Now once again the same thing is happening. 16 was here, it was less than its sibling and it moved up here. Now 16 is less than 21, similarly 10. 10 was less than 11, 10 was less than its sibling, 10 moved up here. Now it has priority less than its child and 17 which was here because of this property that the last node on the path in which both of its children have probability which was larger than the A[i]. The 17 is going to have lower priority than both of these.

While the contents of these three nodes have changed, the heap property is satisfied at all of these three nodes. This is going to have lesser priority than its two children because of this property that we mention here. This is going to have lesser priority than its two children because of this priority, this 16 which was here which was earlier a sibling of 21, had a lesser priority than 21. Now it has become a parent and similarly for 10. 10 will have a lesser priority than this guy because this was the smaller priority node amongst the two siblings, 10 and 11.

(Refer Slide Time: 45:36)

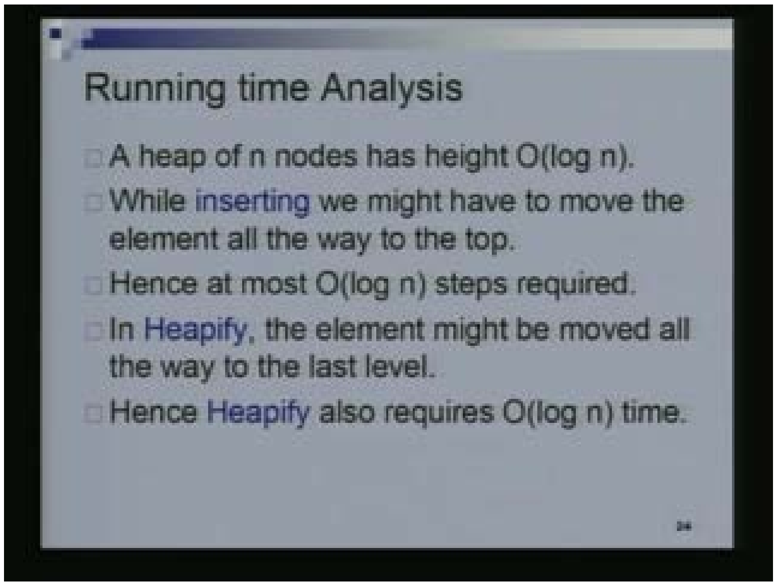


Another View of Heapify

- Heapify(i) traces a path down the tree.
- Last node on path (say j) has both A[left(j)], A[right(j)] are larger than A[i]
- All elements on path have lower priority than their siblings.
- All elements on this path are moved up.
- A[i] goes to location j.
- This establishes correctness

This essentially establishes the correctness of the heapify procedure. What you get as a result of heapify is still a heap. Heapify is a really a mechanism to rebuild a heap. If due to our operations, the heap property is violated at a particular node then we can heapify at that particular node. Assuming of course that the two sub trees, the two child sub trees which are rooted at that node have the heap property then we can do heapify on that node and get a larger heap. Let’s do a quick runtime analysis of the various procedures that we have seen today.

(Refer Slide Time: 47:11)



Running time Analysis

- A heap of n nodes has height O(log n).
- While inserting we might have to move the element all the way to the top.
- Hence at most O(log n) steps required.
- In Heapify, the element might be moved all the way to the last level.
- Hence Heapify also requires O(log n) time.

24

Recall that a heap of n nodes has height order log n. when we are inserting an element, the insertion procedure was that we would create a new node and then we would move the element up the heap. In the worst case, the element might move all the way up to the top. if its moves all the way up to the top then we require time proportional to the height of the heap but an n node heap has the height at most log n. We require at most order log n steps to do an insertion. The other procedure we looked at today was the heapify procedure.

In the heapify procedure on the other hand, the element moved down the heap. We took the element so if we are invoking heapify at node i, we looked at node i we looked at two children of node i, we took the smaller of the two children and swapped it with node i and continued the process till the element came down and the heap property was eventually satisfied. In the worst case the element might moved down as many steps as the height of the heap. The height of the heap is only order log n. In the worst case, the time taken for the heapify is also order log n.

The two operation that we looked today insert and heapify, both require only order log n time. We also looked at the operation of minimum which was just to find the minimum element in the heap without removing it. Just to find the minimum element, we could do it in constant time because all we have do was to go to the root element of the heap and return the value of that, return the element stored at the root. Minimum takes constant time and insertion takes order log n time. Heapify also takes order log n time and in the next class we are going to use heapify to remove the minimum element from the heap and we will be able to do that in order log n time too.

With that we end to today’s discussion. In the next class we are going to see how to use heapify to remove the minimum element, the delete min operation. We also going to see how to create a heap quickly in order n time and we are going to see how heaps can be used to do sorting efficiently.Thank you.