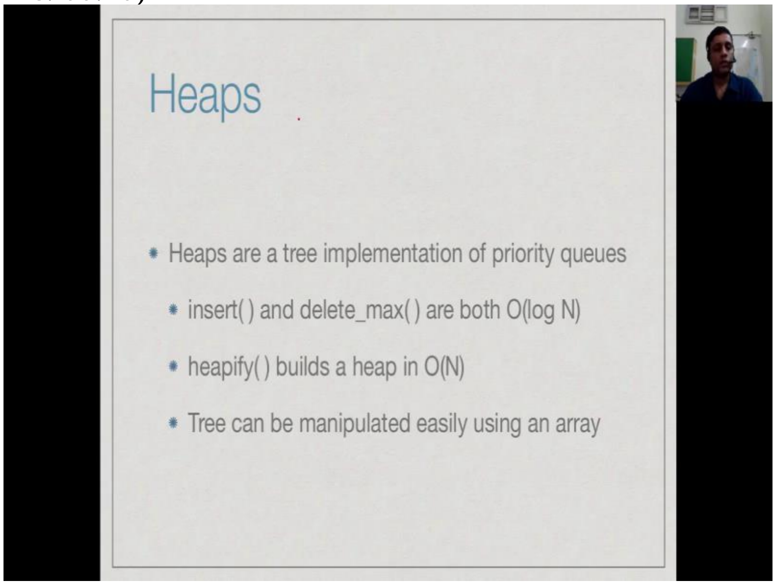
Design and Analysis of Algorithms, Chennai Mathematical Institute

Prof. Madhavan Mukund
Department of Computer Science and Engineering,

Week - 05
Module - 04
Lecture – 36

To complete our discussion of heaps we will see how to use heaps and Dijskstra's algorithm, and also how to use heaps for a different type of sorting.

(Refer Slide Time: 00:10)

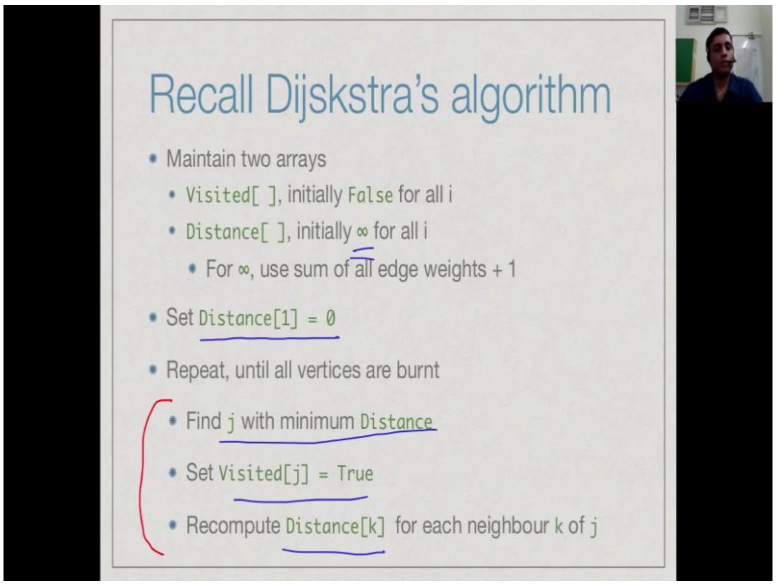


Heaps

- Heaps are a tree implementation of priority queues
  - insert( ) and delete_max( ) are both O(log N)
  - heapify( ) builds a heap in O(N)
  - Tree can be manipulated easily using an array

So, remember that heaps are a tree implementation of priority queues in which insert and delete or both of complexity log N. You can build a heap bottom up in order N time and you can represent it as a tree. And so you can manipulate it in a, I am sorry, you can represent a tree in an array, so you can manipulate it very easily.

(Refer Slide Time: 00:30)

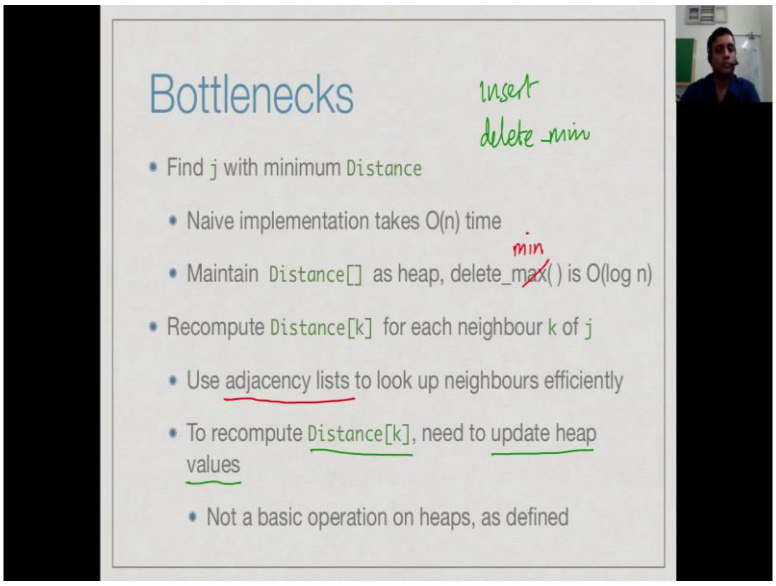


Recall Dijskstra's algorithm

- Maintain two arrays
  - Visited[ ], initially False for all i
  - Distance[ ], initially ∞ for all i
    - For ∞, use sum of all edge weights + 1
- Set Distance[1] = 0
- Repeat, until all vertices are burnt
  - Find j with minimum Distance
  - Set Visited[j] = True
  - Recompute Distance[k] for each neighbour k of j

So, let us go back to Dijskstra's algorithm. So, in Dijskstra's algorithm, we start with the initial vertex and we keep burning these vertices, right, so we keep visiting vertices according to their distance. So, initially we set the distance to be infinity for everything, right. And we use the starting point as, setting the distance to the start vertex as 0, and then we find the smallest unvisited, vertice, vertex, set it to be visited and recompute the distance of each of its neighbours, right. So, the bottlenecks were really at this part of the loop.

(Refer Slide Time: 01:03)



Bottlenecks

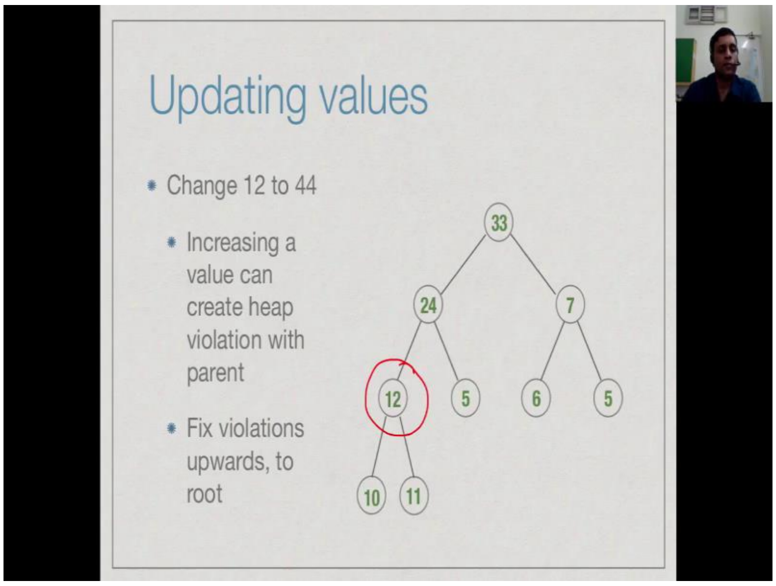
insert
delete_min

- Find j with minimum Distance
  - Naive implementation takes O(n) time
  - Maintain Distance[] as heap, delete_~~max~~ min( ) is O(log n)
- Recompute Distance[k] for each neighbour k of j
  - Use adjacency lists to look up neighbours efficiently
  - To recompute Distance[k], need to update heap values
    - Not a basic operation on heaps, as defined

So, the bottlenecks are first to find the j with the minimum distance, right. So, the naive implementation would take us order n time because we would have to scan all the unvisited vertices, which are in not in any particular order of distance and among them find the minimum, right. So, what seems obvious from what we have discussed so far is that we should maintain distance as a heap and use delete max or actually delete min in this case,because we will want a min heap, ok. So, reduce delete min, but then we have also to recompute the distance.

So, recomputing the distance means examining every neighbour k of j. So, to make sure, that we can look up the neighbours efficiently, we can use adjacency lists so that we do not waste time planning an entire row on adjacency matrix. But the bottleneck here is, that we need to update the distance, that is, we need to get into the heap and change values, so we need to update heap values. Now, we have not really seen how to update heap values, we have only seen insert and delete min and delete max. So, how does update work, right.

(Refer Slide Time: 02:12)

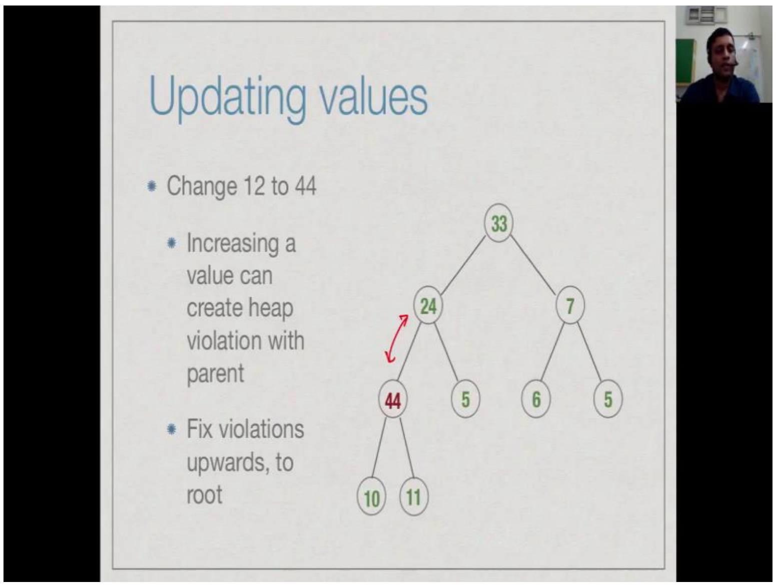


Updating values

- Change 12 to 44
  - Increasing a value can create heap violation with parent
  - Fix violations upwards, to root

So, supposing we want to change this value from 12 to 44, right. If we, if we increase this value, then with respect to its children it cannot get any smaller, right. So, if 12 is bigger than 10 and 11, any larger value will also be bigger than 10 and 11. So, we make it 44. We do not have to look down, but because we make it bigger, it can become bigger than its parent and in fact, it does happen.

(Refer Slide Time: 02:39)

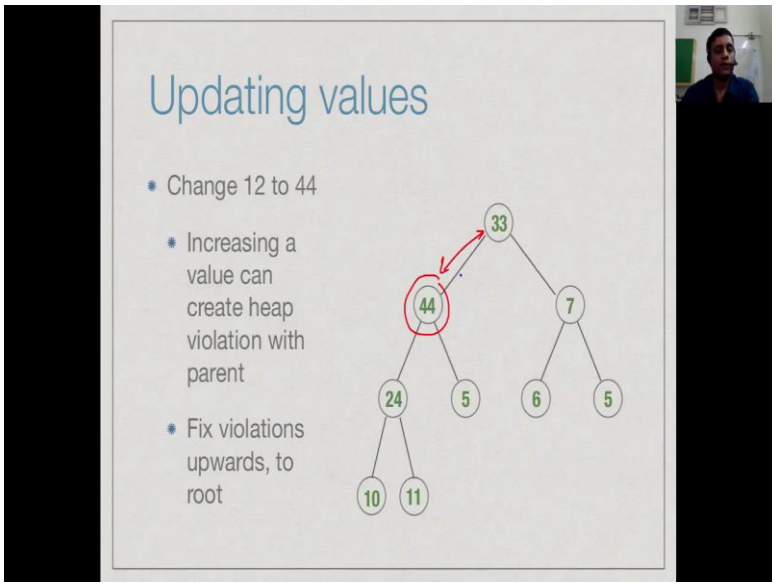


Updating values

- Change 12 to 44
  - Increasing a value can create heap violation with parent
  - Fix violations upwards, to root

So, if we replace, 24, 12 by 44, then we find that there is the heap violation above. So, now, we treat this exactly like we did insert. We look at the parent and so on. So, we fix this violation upwards.

(Refer Slide Time: 02:51)

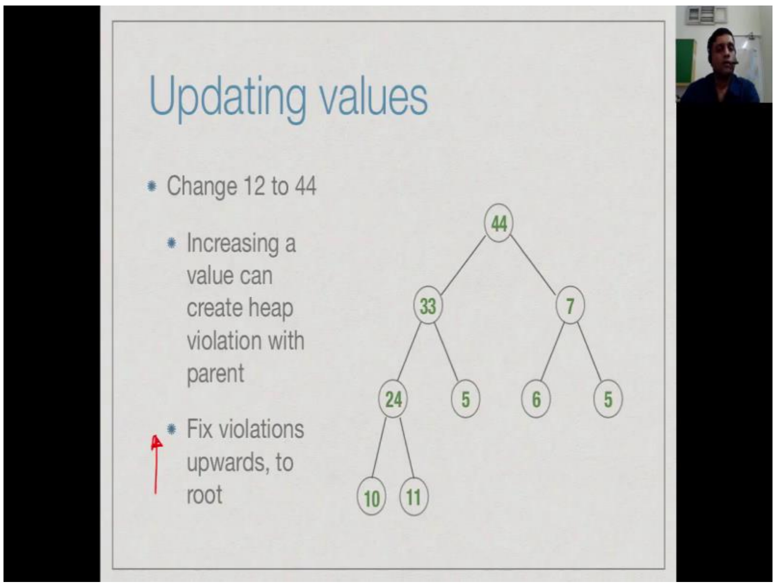


Updating values

- Change 12 to 44
  - Increasing a value can create heap violation with parent
  - Fix violations upwards, to root

So, now we look and we have got 44 here and now we have to check whether it violates anything with its parent, and it does.

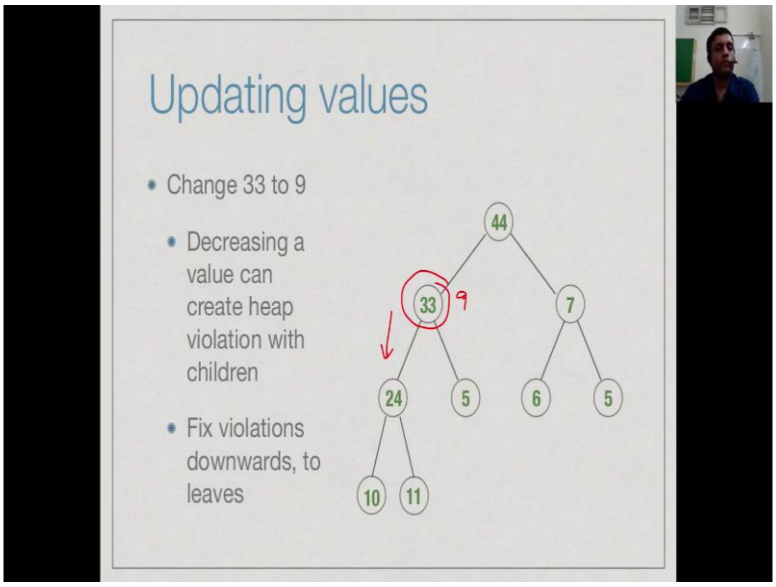
(Refer Slide Time: 03:00)


Updating values

- Change 12 to 44
  - Increasing a value can create heap violation with parent
  - Fix violations upwards, to root

So, exchange that and finally, when it stops, when either there are no more violations or when we reach the root in which case we cannot go up any further, right. So, in increasing a value you fix violations upwards, because in increasing a value you cannot becomes smaller than your children, but you can become bigger than your parent.

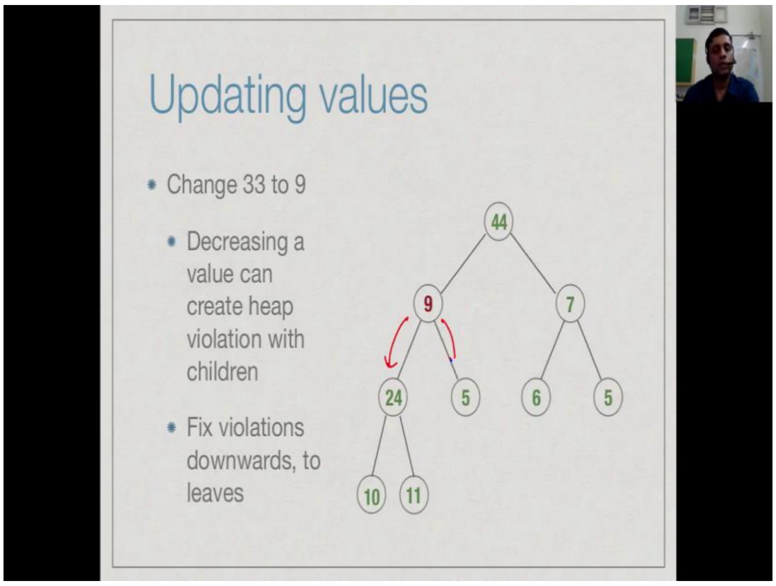
(Refer Slide Time: 03:17)


Updating values

- Change 33 to 9
  - Decreasing a value can create heap violation with children
  - Fix violations downwards, to leaves

The other type of change is to decrease the value. So, supposing I take this 33 and I make it 9. Now, again by the same logic it was 33, was smaller than 44, 9 will also be smaller than 44. Any value smaller than 33 cannot create a violation up. So, I must look down, right. So, if I bring it down to 9, the number is the problem between 33 and 24.

(Refer Slide Time: 03:36)

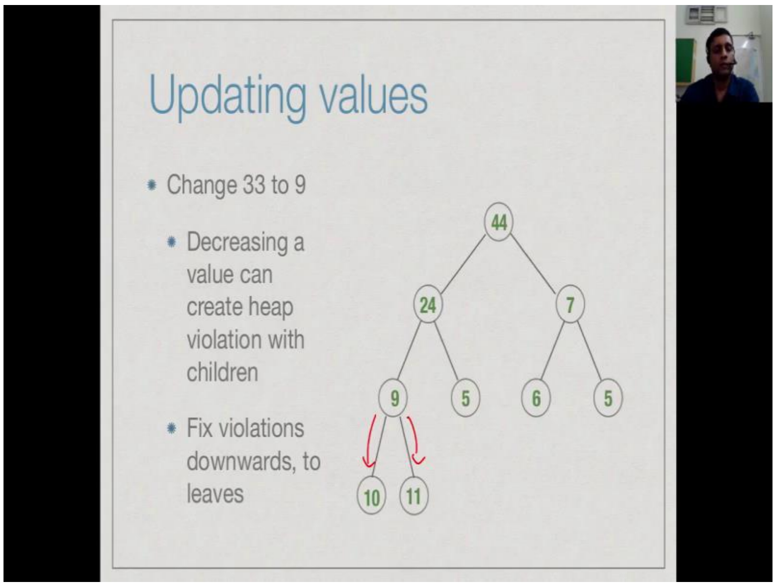


Updating values

- Change 33 to 9
  - Decreasing a value can create heap violation with children
  - Fix violations downwards, to leaves

So, if this is my new value, I have to check with its two children and take the biggest one up. In this case it is 24, right.

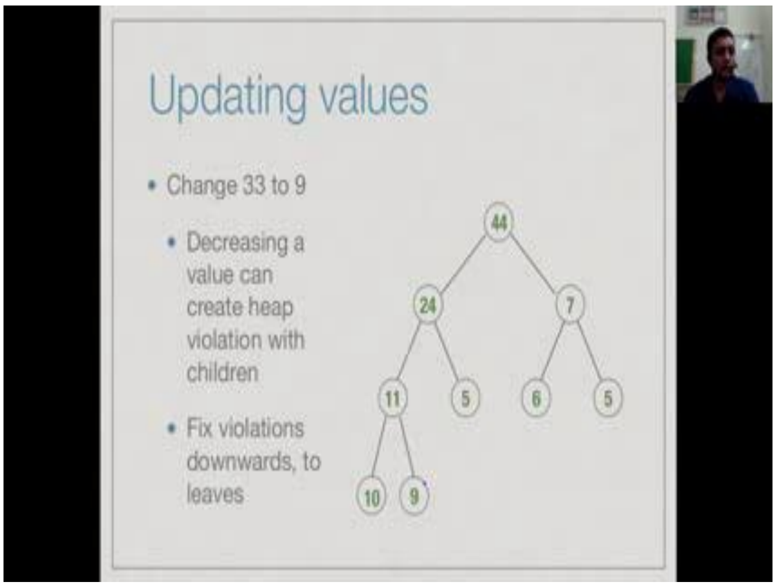
(Refer Slide Time: 03:43)



Updating values

- Change 33 to 9
  - Decreasing a value can create heap violation with children
  - Fix violations downwards, to leaves

(Refer Slide Time: 03:48)

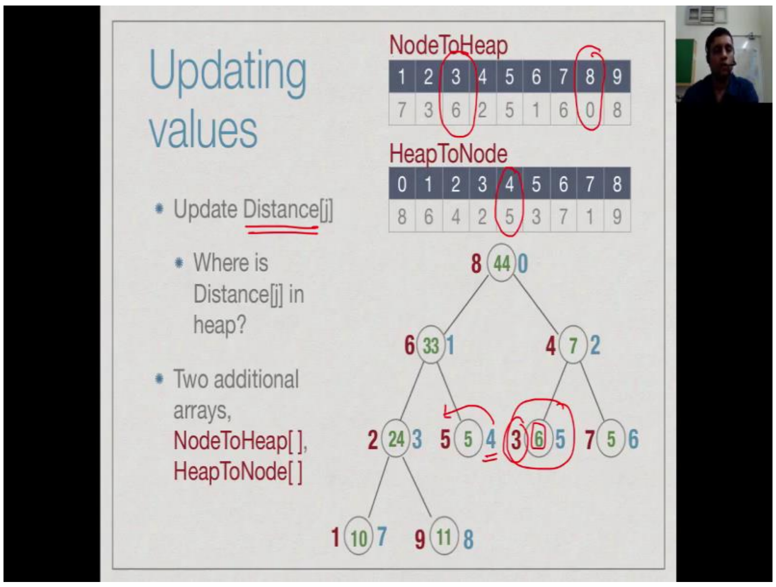


Updating values

- Change 33 to 9
  - Decreasing a value can create heap violation with children
  - Fix violations downwards, to leaves

So, now having come here, I need to now check its two children. And now, take the biggest one up in which case it is 11, and so this is how it is, right. So, when I update values and decrease the value, I have to fix violations downwards because reducing a value cannot make it bigger than its parent, but it can make it smaller than one of its two children.

(Refer Slide Time: 04:04)



Updating values

- Update Distance[j]
  - Where is Distance[j] in heap?
- Two additional arrays, NodeToHeap[ ], HeapToNode[ ]

NodeToHeap

1	2	3	4	5	6	7	8	9
7	3	6	2	5	1	6	0	8

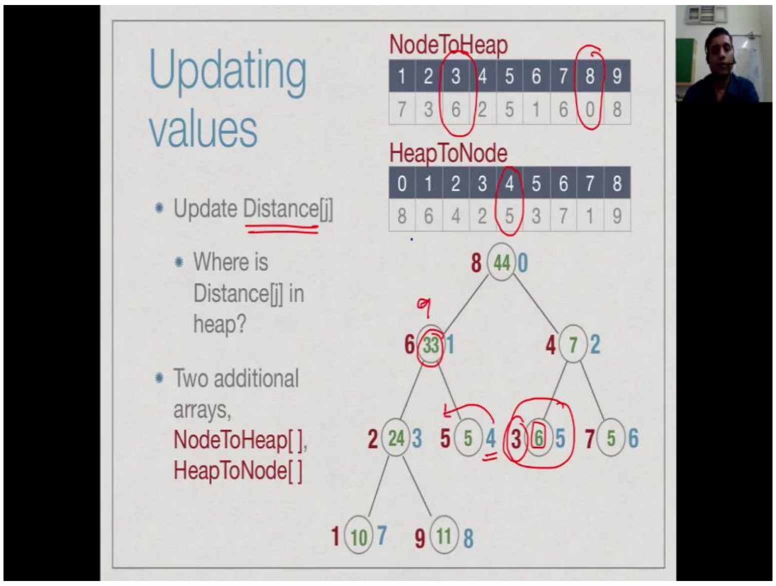
HeapToNode

0	1	2	3	4	5	6	7	8
8	6	4	2	5	3	7	1	9

So, if you look at Dijskstra’s algorithm, the way it works is, we take a vertex j and say, update this distance, right. So, we have to update the distance of some vertex j, which is somewhere in the heap, right. So, what our previous example showed up is, if we, if we put our finger on the node in the heap and change its value, we know how to adjust the heap. But how do we find where j is in the heap, right? So, where is j located? So, we need this extra information to be kept separately, right. So, we would keep two new arrays pointing from the nodes, which are one to end to the heap, which is 0 to N minus 1 and vice versa from the heap which is 0 to N to the nodes.

So, so here in this picture, right, these two things are drawn in different colors. So, the red labels against a node indicate the vertex. So, 44 represents the distance of vertex 8 and to indicate that we have an array, saying that the node ((Refer Time: 05:06)), saying that the node 8 in the graph is vertex 0 in the heap. Similarly, if I look at this it says, that this vertex is 3. So, the vertex 3 in my graph is node 6 in my heap. Conversely, if I am in the heap and I am if I am at node 4, which is represented loop, then which vertex does this correspond to? So, it says, that the node 4 in the heap corresponds to vertex 5. So, from the heap given the index of the node in the heap, which node in the graph does it correspond to? So, we have the extra things which we set of and we have to update this when we do our swaps or ((Refer Time: 05:49)), right.

(Refer Slide Time: 05:50)



Updating values

- Update Distance[j]
  - Where is Distance[j] in heap?
- Two additional arrays, NodeToHeap[ ], HeapToNode[ ]

NodeToHeap

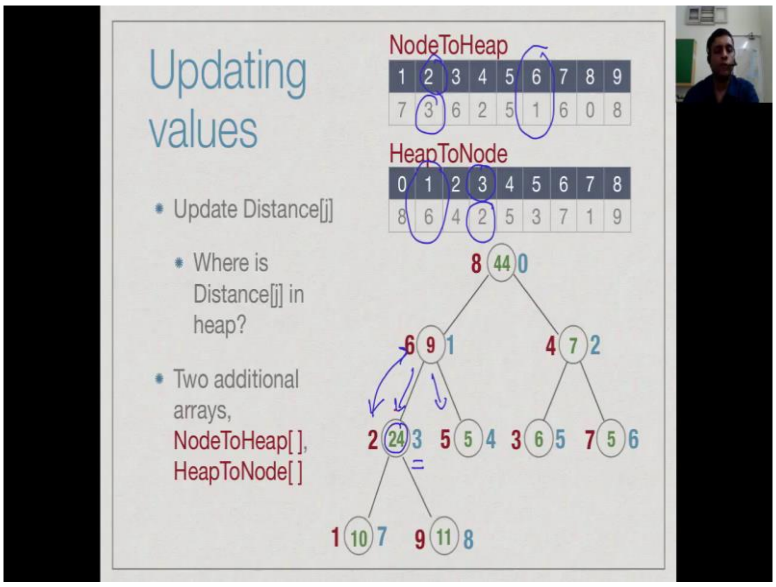
1	2	3	4	5	6	7	8	9
7	3	6	2	5	1	6	0	8

HeapToNode

0	1	2	3	4	5	6	7	8
8	6	4	2	5	3	7	1	9

So, now for instance, supposing we have this previous thing and we want to make this 33 into 9.

(Refer Slide Time: 05:57)



Updating values

- Update Distance[j]
  - Where is Distance[j] in heap?
- Two additional arrays, NodeToHeap[ ], HeapToNode[ ]

NodeToHeap

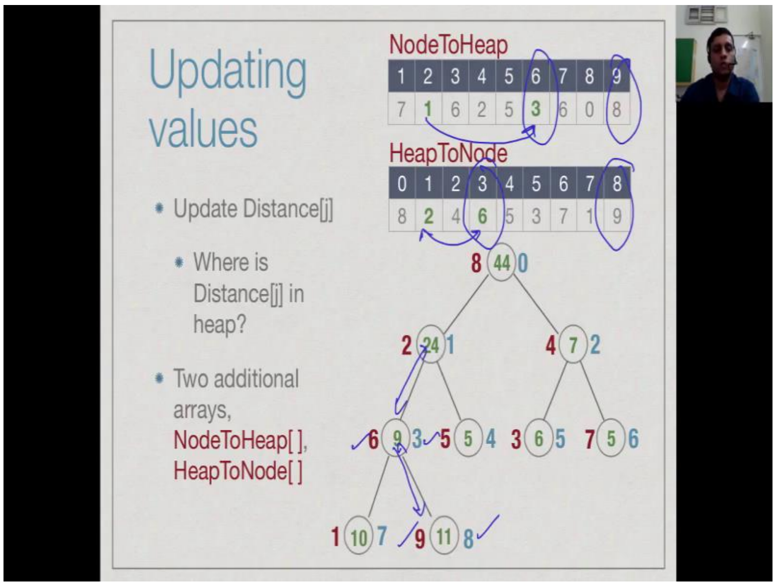
1	2	3	4	5	6	7	8	9
7	3	6	2	5	1	6	0	8

HeapToNode

0	1	2	3	4	5	6	7	8
8	6	4	2	5	3	7	1	9

So, our goal is to reduce 33 to 9, which we did in the previous set. Now, let us do so. So, now, when I do this, I need to go down to its two children in the heap and recognize that this 24 must change. Now, since 24 must change, I must know also how to update it. So, 24 is node 3 in the heap. So, node 3 is vertex 2. So, I have to go to vertex 2 here. So, I need to update these entries. So, I need to update 6 and 1, and 3 and 2. So, therefore, when I exchange 9 and 24, I must also exchange 6 and 2. I must say, that now vertex node, vertex 2 in my graph is now at node 1 in the tree and vertex 6 in my graph is now at node 3 in the tree. Conversely, node 1 in the tree points to vertexes, node 3 in the tree points to vertex, points to vertex 2, node 3 points to vertex ((Refer Time: 06:55)), right.

(Refer Slide Time: 06:57)



Updating values

- Update Distance[j]
  - Where is Distance[j] in heap?
- Two additional arrays, NodeToHeap[ ], HeapToNode[ ]

NodeToHeap

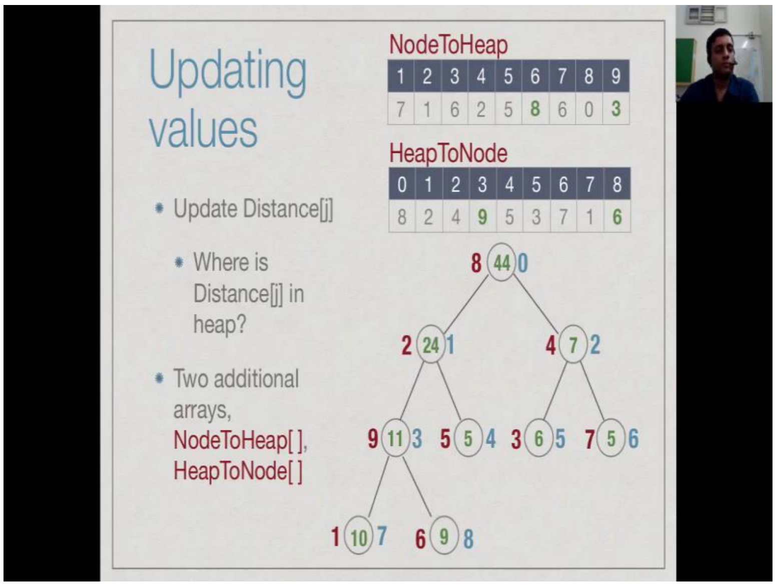
1	2	3	4	5	6	7	8	9
7	1	6	2	5	3	6	0	8

HeapToNode

0	1	2	3	4	5	6	7	8
8	2	4	6	5	3	7	1	9

So, after this update, basically, these two values have got exchanged from what they were previously, ok. So, in addition to swapping at this level, I also have to recognize that this swap had. Now, I do one more swap. Because 9 and 11 had the swaps, now I will swap the entries for 3, 6 and 8, 9. And similarly, 6, 3 and 9, 8, right. So, the entries corresponds to vertex 6 and vertex 9 and node 3 and node 8, right. So, vertex 6, vertex 9, node 3 and node 8, these have to be swapped, right.

(Refer Slide Time: 07:26)



Updating values

- Update Distance[j]
  - Where is Distance[j] in heap?
- Two additional arrays, NodeToHeap[ ], HeapToNode[ ]

NodeToHeap

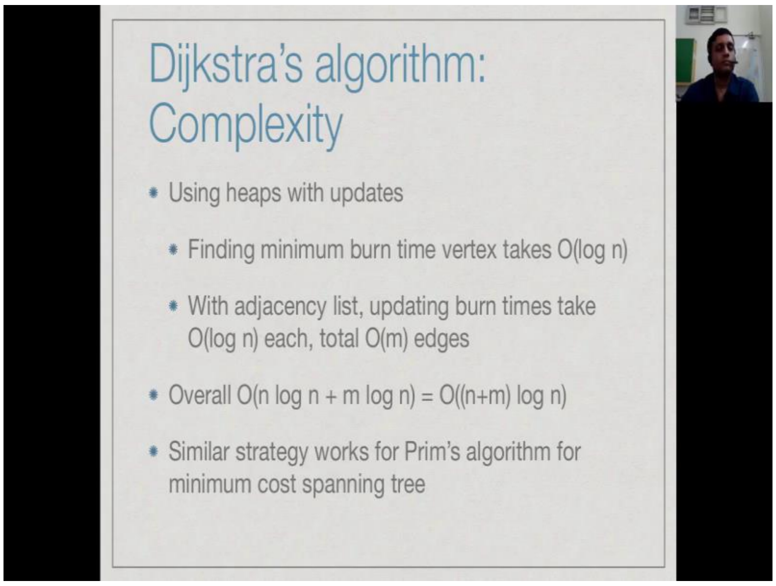
1	2	3	4	5	6	7	8	9
7	1	6	2	5	8	6	0	3

HeapToNode

0	1	2	3	4	5	6	7	8
8	2	4	9	5	3	7	1	6

So, by keeping this to extra arrays, right, I can do these updates very easily, because I have a way of going backwards and forwards between the heap index, and the node index. And unless I do this, I cannot really use for Dijskstra's algorithm because Dijskstra’s algorithm will ask us to update value in the heap, but I need to know which value in the heap I need to update. The update can be done using this upward or downward manipulation exactly like insert or delete max, but the real problem is identifying where the update starts.

(Refer Slide Time: 07:55)



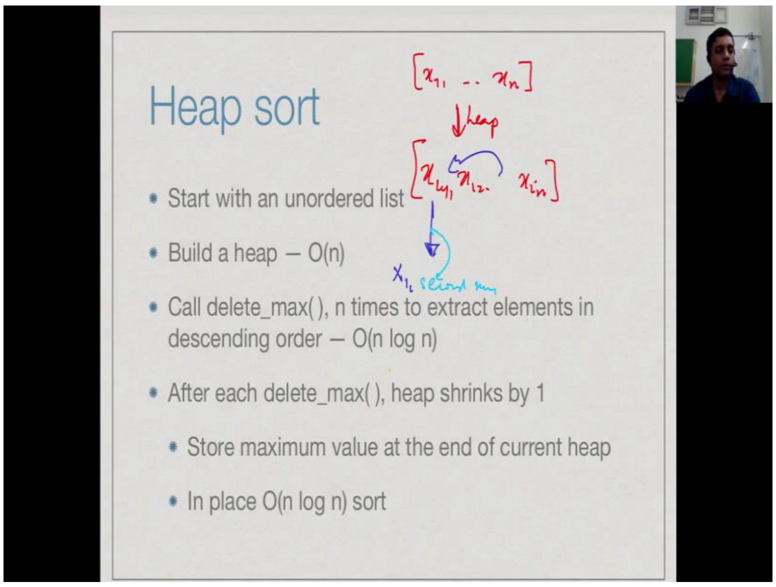
Dijkstra's algorithm: Complexity

- Using heaps with updates
  - Finding minimum burn time vertex takes O(log n)
  - With adjacency list, updating burn times take O(log n) each, total O(m) edges
- Overall O(n log n + m log n) = O((n+m) log n)
- Similar strategy works for Prim's algorithm for minimum cost spanning tree

So, as we saw before, now we can use this heap with this update operation and find the minimum time vertex and log n time. And because we have adjacency list, right, overall the loops updating the burn times takes log n time per edge, and there are totally order m edges. So, overall we get n log n for updating, for finding the minimum n times and m log n for updating the minimum m times. So, n plus n log n.

You can use the similar strategy for Prim's algorithm. In Prim's algorithm the only distance, the notion of distance is not the same. We do not accumulate the distance, we look at the cost of the actual edge. But still, we need to attract the minimum cost edge connecting a node to a tree, to the current tree and then we need to update these things after we add this edge to the tree, right. So, exactly the same idea of using a min heap with updates we will use. So, getting Prim's algorithm also down to the same complexity as Dijskstra's algorithm.

(Refer Slide Time: 08:55)



Heap sort

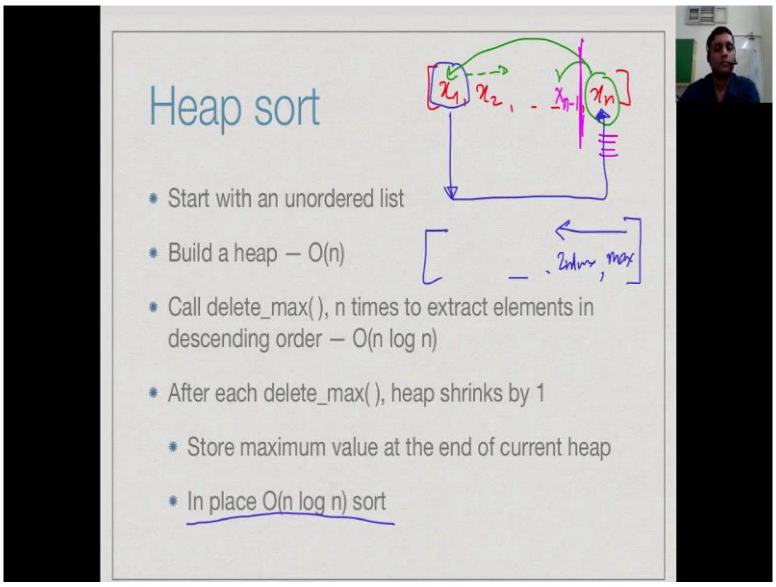
- Start with an unordered list
- Build a heap — O(n)
- Call delete_max( ), n times to extract elements in descending order — O(n log n)
- After each delete_max( ), heap shrinks by 1
  - Store maximum value at the end of current heap
  - In place O(n log n) sort

So, before we leave heaps, let us see how to use heaps to sort. So, we want to sort a list of values. So, what we can do is, we can first build a heap, right. So, we now start with some arbitrary sequence of values, x 1 to x n, then we build a heap and we possibly get reordered in some way of x i 1, x i 2, x i n, but this is not in sorted way, this is in ((Refer Time: 09:21)), that is, we know, that the maximum is at the left and so on. Now, I do it delete max and I put this guy out, right. So, I know that this is the maximum.

Then, after this, this something will come to the front. At the next point, that will come out, and then I will get the second max and so on, right. So, if I do delete max n times, clearly, at each point I get the next maximum. So, I am extracting elements from the tree in descending order, and so I get a sorted output. I can reverse set, I can keep in ascending order, it does not matter, but I am just extracting the elements in a particular order. And so this is a trivial sorting algorithm.

Now, each extraction takes log n time because it is a delete time delete max, right, or a delete max or a delete min depending on what type of heap for using. So, you do n such extraction. So, in log n time you can get the elements out in sorted order and to put them in, you took only order n time. So, overall we sorted n log n time.

(Refer Slide Time: 10:19)



Heap sort

- Start with an unordered list
- Build a heap — O(n)
- Call delete_max( ), n times to extract elements in descending order — O(n log n)
- After each delete_max( ), heap shrinks by 1
  - Store maximum value at the end of current heap
  - In place O(n log n) sort

There is a small subtle thing that you can ask about this. The question is, where these values go, right. So, initially, in the first iteration, if I have my heap looking like this, this is my maximum. So, it looks like I have to put in a new list, but what happens after this step is, that this value gets inserted here and then it percolates down by the heap if I fix operation, right. So, when I delete the value at the root, I take the last leaf, put its value in the root and then I put it down so that all the heap property ((Refer Time: 10:54)). And after this, now my heap only look like this because this value is gone. So, my heap ends with x minus n minus 1.

So, I have a place in my array, which is not being used any more for the next delete max because the next delete max involves only n minus 1 operation. So, therefore, I can, in fact, at this point go and put this value back into the heap at this position and reassure it is not much, right. So, now what will happen is, I will have the max coming here, then I will have the next second max coming here and so on. So, if I keep doing delete max, I will be propagating from the right the value in descending order and therefore, finally, it may equal actually end up being resorted in ascending order, right.

So, this gives me a in place order n log n swap, right, by just making sure, that when I delete the value instead of throwing it away or putting into a new list, I just put it into the place it was vacated right now by seeking the heap by one value, and then I automatically get an output in sorted way, right. So, heaps can be used to do a very different kind of order n log n short in place.