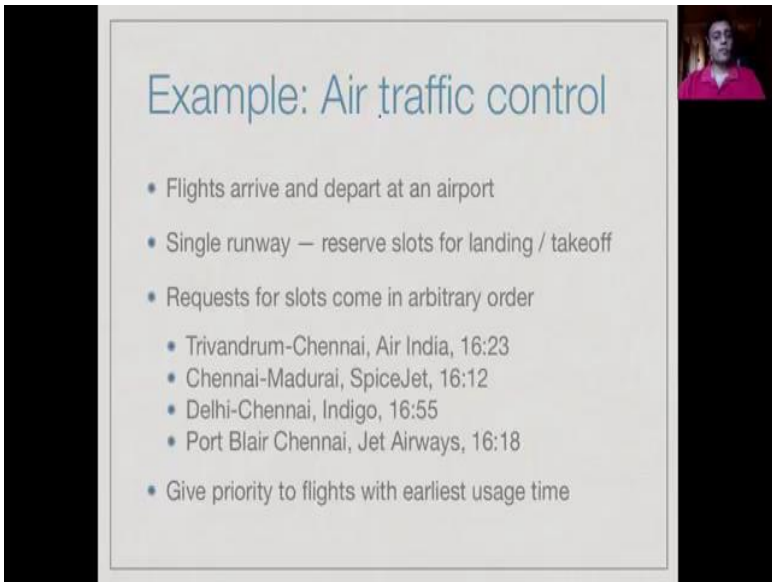
Design and Analysis of Algorithms, Chennai Mathematical Institute
Prof. Madhavan Mukund
Department of Computer Science and Engineering,

Module – 01
Lecture - 39
Search Trees

We now look at another data structure called a Search Tree.

(Refer Slide Time: 00:05)

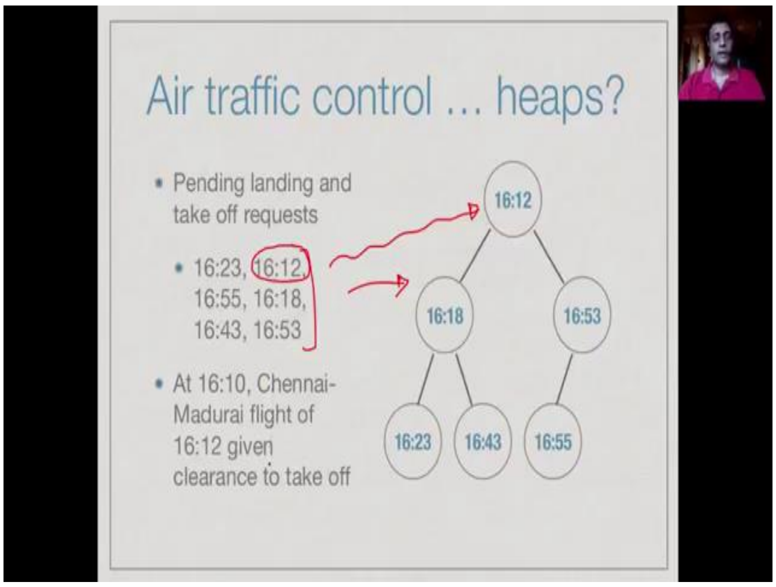

Example: Air traffic control

- Flights arrive and depart at an airport
- Single runway — reserve slots for landing / takeoff
- Requests for slots come in arbitrary order
  - Trivandrum-Chennai, Air India, 16:23
  - Chennai-Madurai, SpiceJet, 16:12
  - Delhi-Chennai, Indigo, 16:55
  - Port Blair Chennai, Jet Airways, 16:18
- Give priority to flights with earliest usage time

So, to motivate search trees let us consider the following example. So, supposing you are an air traffic controller and you controlling access to a single run way, flights have to land and takeoff. So, the air traffic control tower receives request from the air craft about the expected arrival and departure times and these request come at random time. So, it is not that they come in order of the time of the actual event.

So, we might first get a request for a landing in Chennai at 16:23 and later on we might get a request for a takeoff at 16:12 which is actually earlier. And then, we might get another arrival information at 16:55, then other earlier arrival information at 16:18 and the strategy with the control tower is supposed to follow is to give priority to the flight with the earliest usage time.

(Refer Slide Time: 00:59)

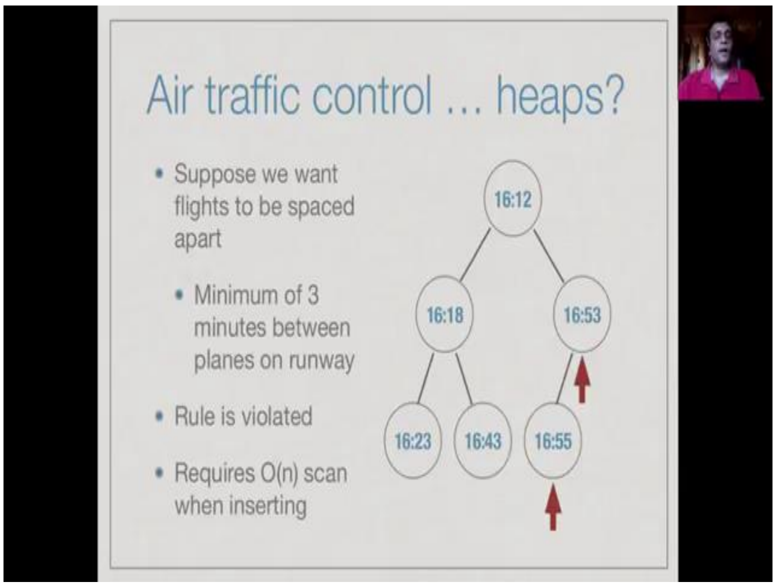


Air traffic control … heaps?

- Pending landing and take off requests
  - 16:23, 16:12, 16:55, 16:18, 16:43, 16:53
- At 16:10, Chennai-Madurai flight of 16:12 given clearance to take off

So, this is a priority queue, each request goes into the queue, but it has a priority depending on the expected time that it is going to take place. So, we could give a min heap representation and if you take the 6 request in the order that they come we insert them into the min heap, then we will get the min heap that we see on the right hand side. And so in this, because 16:12 happens to be the earliest event among these 6, it will automatically float up to the route and so it will be available to us is the next went to the process and then maybe 2 minutes before that event may happens, we get send signal to the flight giving a clearance to take off.

(Refer Slide Time: 01:42)



Air traffic control … heaps?

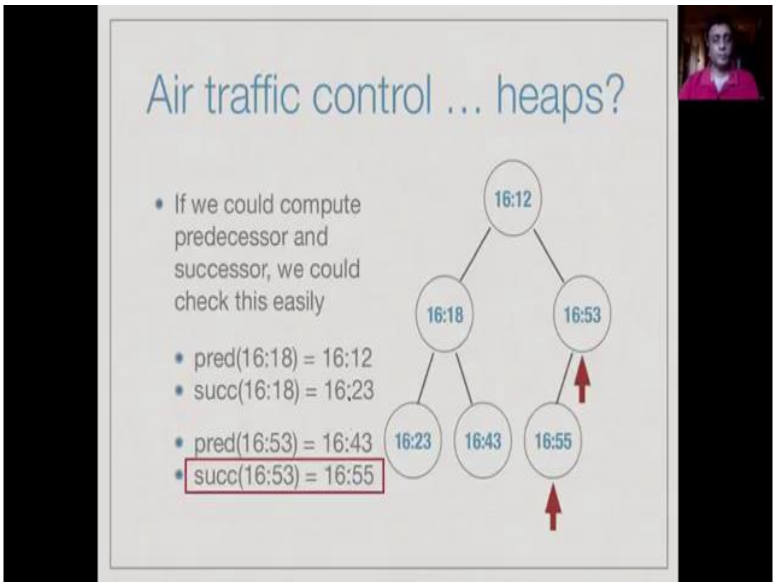
- Suppose we want flights to be spaced apart
  - Minimum of 3 minutes between planes on runway
- Rule is violated
- Requires O(n) scan when inserting

So, suppose we have an extra requirement on this, not only do we have to want to give priority to the earliest event to happen, but we also want to impose some minimum separation of planes to avoid any possibility of collusions on the run way. So, we say that 2 planes accessing the run way must be a minimum of 3 minutes apart. Now, in this example we can see that the rule is violated by the 2 events at 16:53 and 16:55.

So, what happens is that we should when we insert or we accept the request for 16:55 assuming it came after 16:53. At this point, we should check that whether it is at least 3 minutes apart from all the current bending request, if not we should send a message to the pilot saying 16:55 is not possible, you must move your takeoff or landing to 16:56 or later. Now, unfortunately in the heap data structure there is no easy way to do this. So, when 16:55 is added to the heap, we have to scan it against all the elements in the heap in order to find out whether it is within 3 minutes or any of them.

So, this would be a linear scan, so all other operations insert and delete on a heap for logarithmic. But, this if you want to impose this constraint that move to elements in the heap of within 3 minutes of each other, then we would add an extra linear time cost to every insert and this would then make a heap unviolable for this kind of data.

(Refer Slide Time: 03:10)



Air traffic control … heaps?

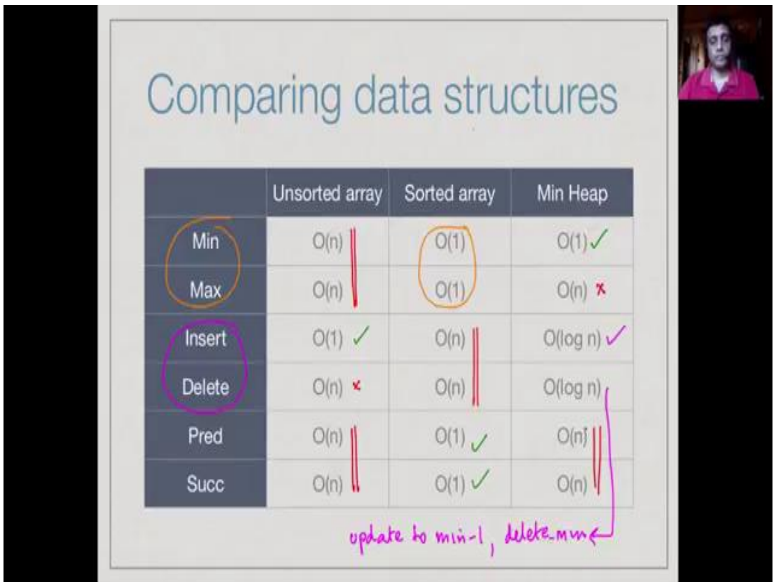
- If we could compute predecessor and successor, we could check this easily
  - pred(16:18) = 16:12
  - succ(16:18) = 16:23
  - pred(16:53) = 16:43
  - succ(16:53) = 16:55

So, one way to compute this kind of requirement is to be able to check the predecessor and the successor. In other words, given the list of value set we have for any value can we quickly compute what is the previous in terms of the nearest a smaller value and then next successor what is the nearest bigger value. So, if you look at 16:18 for example, then the next smaller value is here, so this is the predecessor 16:12 and the next bigger value is here, so this is the successor.

Now, this happens to be it looks like there is some nice relationship within the heap in terms of the tree structure between predecessor and successor. But, if you look at a different value for example, 16:53 then we will find that the predecessor lies in a different branch of the tree and the successor lies in the same branch. So, there is no direct relationship between the tree structure of the heap and the predecessor and the successor relation that we need.

However, the important thing to notice if we could compute predecessor and successor, then we can solve this problem. Because, once we look at the successor or the predecessor and we find something which is within 3 minutes, then we can fix, flaw the warning and signal to that aircraft that request is not turn away.

(Refer Slide Time: 04:30)



Comparing data structures

	Unsorted array	Sorted array	Min Heap
Min	O(n)	O(1)	O(1) ✓
Max	O(n)	O(1)	O(n) ×
Insert	O(1) ✓	O(n)	O(log n) ✓
Delete	O(n) ×	O(n)	O(log n)
Pred	O(n)	O(1) ✓	O(n)
Succ	O(n)	O(1) ✓	O(n)

update to min-1, delete-min

So, if you want to maintain this information, let us try and look at the different data structures we have seen, so far. So, we have unsorted array, sorted arrays and min heaps, now in this particular structure we are not going to look at directly a delete min or a delete max. But, two separate operations one which checks the minimum and maximum and one which deletes an element arbitrarily. So, for min and max as far as min and max go, then it is clear that the best data structure is sorted array, because these values are at the extreme ends of the array.

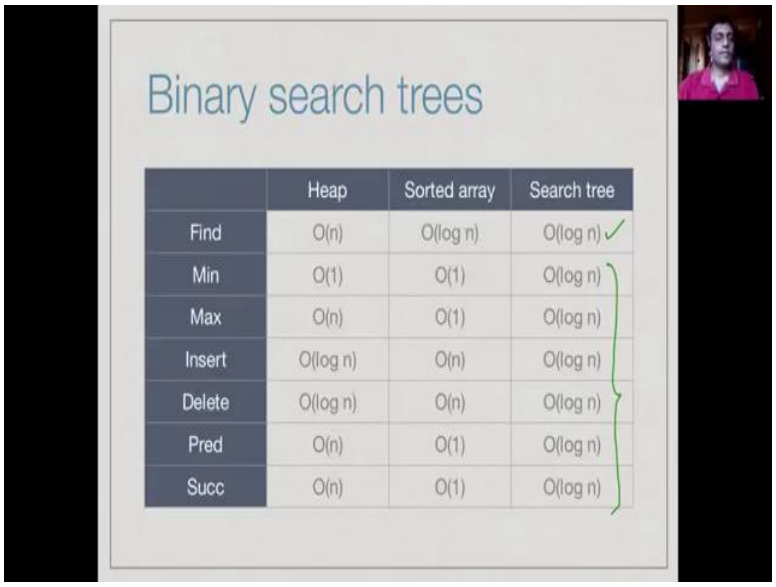
The worst is an unsorted array, because it will be anywhere and both look at linear time and then depending on whether we have a min heap or the max heap, one of them is good and the other one is bad. So, in a min heap we can find the minimum with the root, the maximum could be anywhere and likewise in a max heap a maximum would be at the root, minimum could be anywhere.

So, if you look at the next set of operations, this is insert and delete, now here it turns out that an unsorted array is good for insert, because, we can just tick it at the end. For delete, we have to go through the array and look for the value and remove it, so it will take linear time. In a sorted array both will be slow, because we have to either insert in the correct position, like in insertion sort or when we delete we have to compress the array which will take linear time.

Now, we know that in a min heap insert is logarithmic, delete is also logarithmic even though this is not delete min. So, this can be broken up a two steps, so update to the current minimum overall minus 1 and then delete. So, you can take the current value that you want to delete, reset it is value, so that it becomes the minimum, in which case as we saw when you update to a smaller value, it will propagate up to the root. And then, if you delete min that value is disappear and so we would effectively in two steps of log n each we would deleted it, so overall it is log n.

And now as we just said predecessor and successor in a sorted list they are adjust, so these are both constant operations. On the other hand, in an unsorted list we have to look through the whole array and we also said in a heap, we have to scan. Because, there is no particular order in which the adjacent elements are stored in the heap.

(Refer Slide Time: 06:59)

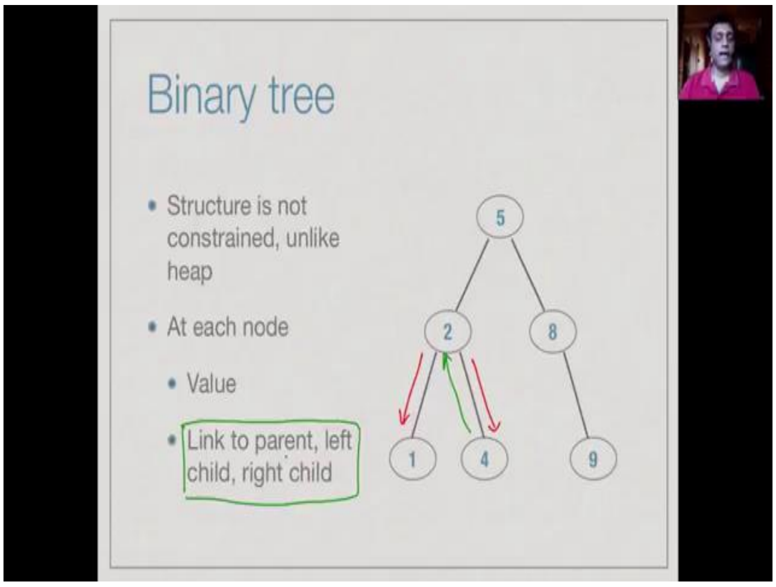


Binary search trees

	Heap	Sorted array	Search tree
Find	O(n)	O(log n)	O(log n) ✓
Min	O(1)	O(1)	O(log n)
Max	O(n)	O(1)	O(log n)
Insert	O(log n)	O(n)	O(log n)
Delete	O(log n)	O(n)	O(log n)
Pred	O(n)	O(1)	O(log n)
Succ	O(n)	O(1)	O(log n)

So, what we are going to look at today and binary search trees and in a binary search tree we are going to argue that all of these operations are actually logarithmic, provided we maintain the binary search tree with a good structure. And also we will look at an operation called find, which searches for a value and this will also be logarithmic. So, we will simultaneously be able to optimize, all these 7 operations in a binary search tree.

(Refer Slide Time: 07:27)



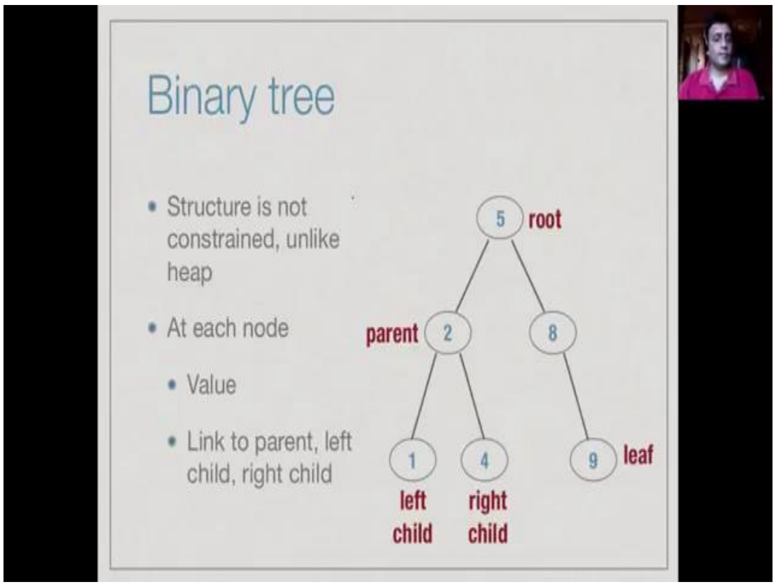
Binary tree

- Structure is not constrained, unlike heap
- At each node
  - Value
  - Link to parent, left child, right child

So, a binary tree is just a tree with a root in which every node has either 1, 2 or 3 children, the heap poses a very special kind of binary tree, which we filled up top to bottom left to right, but in general a binary tree has no constraint. So, we have values at each node and we will assume that not only do we have way to go from the parent to the child, but we also have a way to go from the child to the parent.

So, every node we are going to assume will have a link to it is parent, left child and right child of course, these links maybe missing, but if they are there then they all three. So, we will can point up the tree and to the two nodes below it or down the tree.

(Refer Slide Time: 08:10)

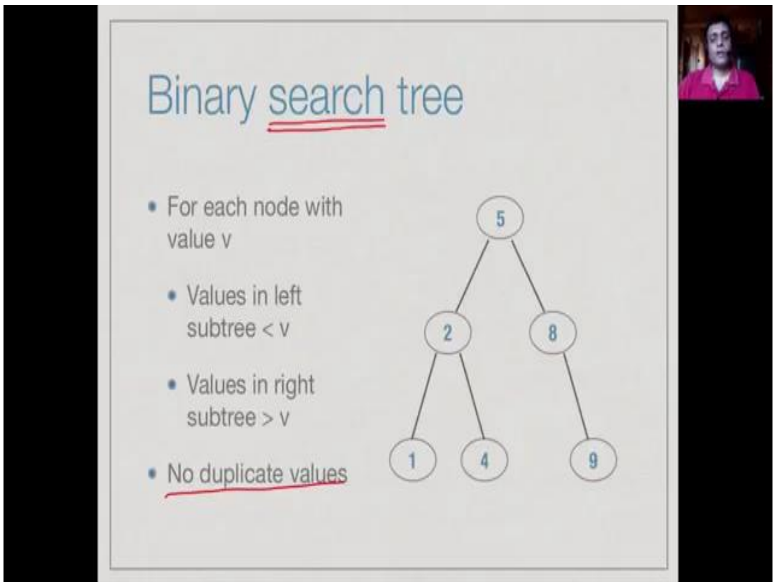


Binary tree

- Structure is not constrained, unlike heap
- At each node
  - Value
  - Link to parent, left child, right child

So, just in terms of terminology the root is the top most node, it has no parent, the leaf is any node which has no children and at any given point, if you look at a node, then it is a parent of it is left child and it is right child.

(Refer Slide Time: 08:28)



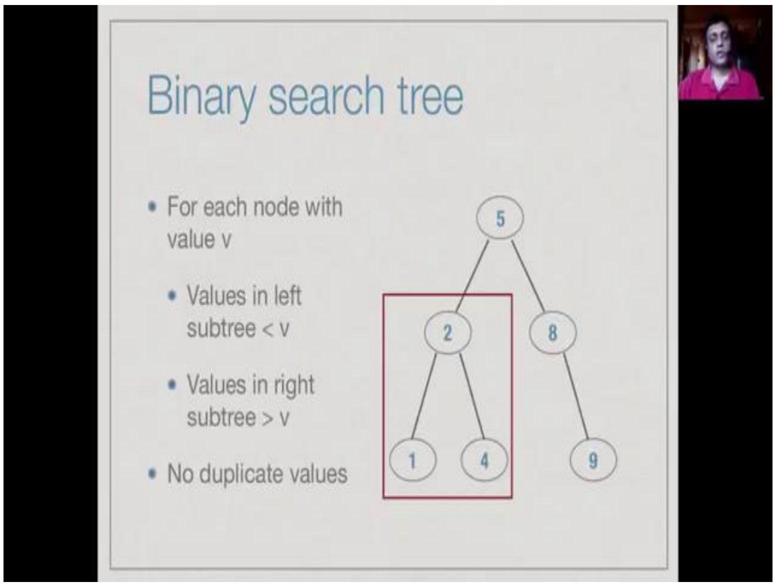
Binary search tree

- For each node with value v
  - Values in left subtree < v
  - Values in right subtree > v
- No duplicate values

So, now we take a binary tree and we further impose a constraint of the values. So, remember like in a heap we first at as structural condition that at a tree is filled top to bottom left to right and then we said, there is a heap property which says max heap or min heap, either a node is bigger than it is 2 children or it is smaller than it is 2 children. So, they we have a property on the structure and they was a property on the value, so how they are arrange with this respect to each other.

So, in a binary search tree, the binary tree could have an arbitrary structure that the values are arranged in a specific way. So, for each node with a value v, all the nodes below v, smaller than v are in the left sub tree and all the values bigger than v are in the rights sub tree. And we typically we will assume that there are no duplicate values.

(Refer Slide Time: 09:20)

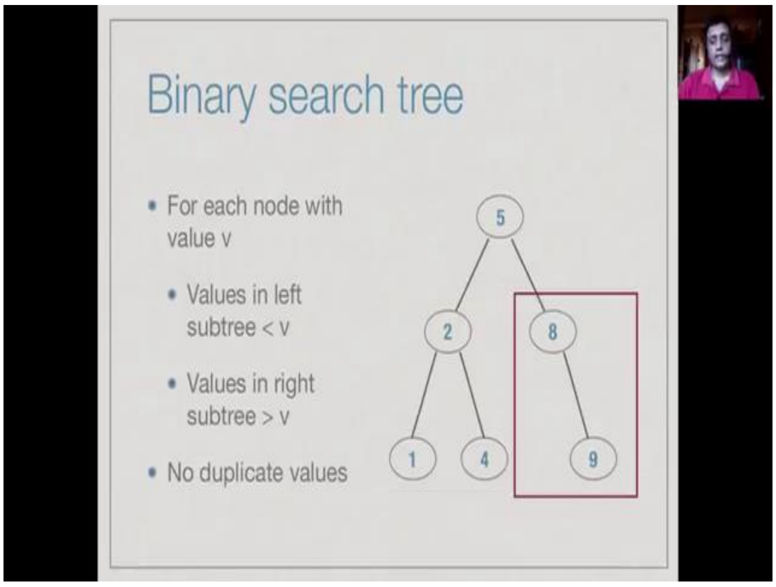


Binary search tree

- For each node with value v
  - Values in left subtree < v
  - Values in right subtree > v
- No duplicate values

So, for example, here if you look at a root node 5, then the left sub tree of 5 contains values 1 to 4 which are all the nodes in these tree smaller than 5.

(Refer Slide Time: 09:26)

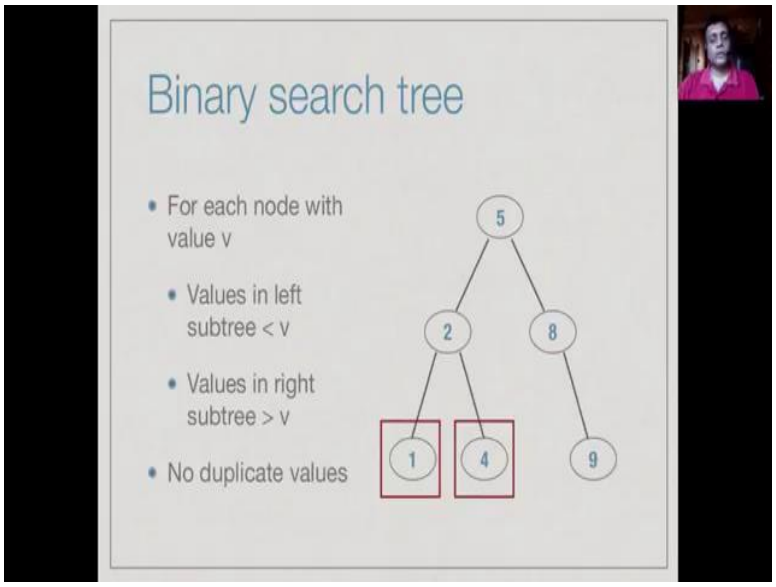


Binary search tree

- For each node with value v
  - Values in left subtree < v
  - Values in right subtree > v
- No duplicate values

And the right sub tree contains 8 and 9, now this is the property that is recursively satisfied throughout the tree.

(Refer Slide Time: 09:33)

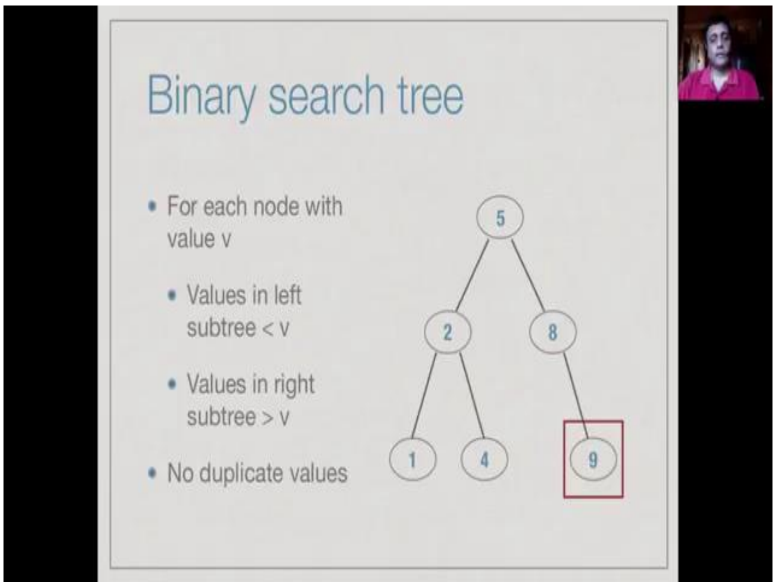


Binary search tree

- For each node with value v
  - Values in left subtree < v
  - Values in right subtree > v
- No duplicate values

So, if you look at the node 2 for example, it is left child, left sub trees just a single node 1 which is value smaller than in the tree rooted at 2 and it is right sub trees the value 4 which is the right it is a bigger than 2, the only value bigger than 2 this sub tree.

(Refer Slide Time: 09:49)

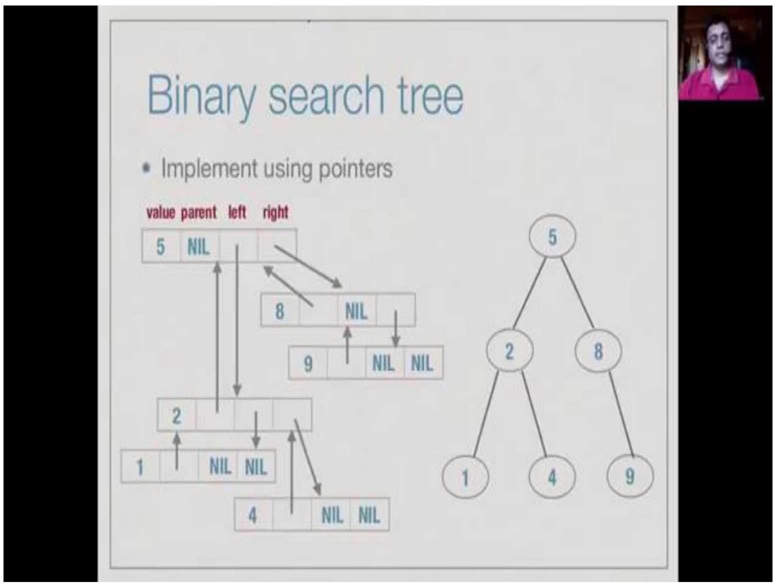


Binary search tree

- For each node with value v
  - Values in left subtree < v
  - Values in right subtree > v
- No duplicate values

Likewise, if you look at 8 in this only one value other than 8 in the sub tree starting at 8 has the value 9 and since it is we go in the right sub tree and we have the left sub tree empty. So, we can have this kind of gaps unlike in heaps, we can have a gap, but we have this uniform property that at any node in the sub tree all the value smaller than the node are on the left, and all the values bigger around the right.

(Refer Slide Time: 10:13)

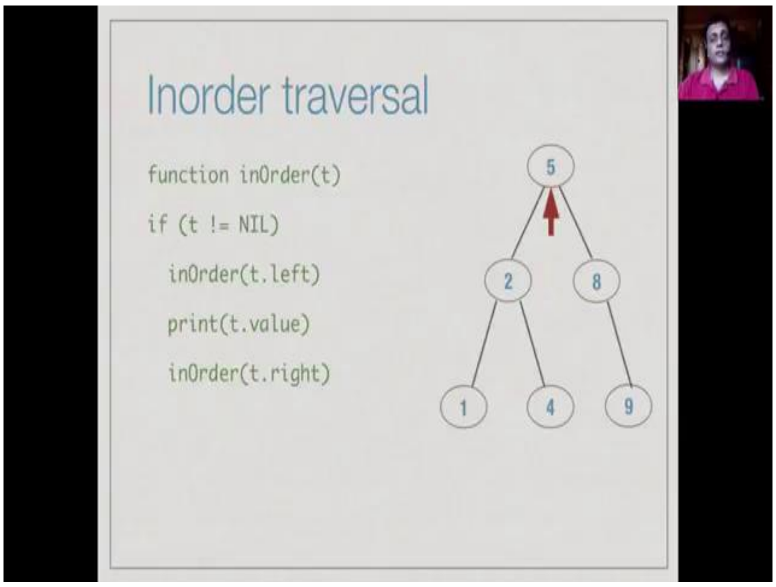


Binary search tree

- Implement using pointers

So, typically we would implement this as a link structure, where we have each node with 3 fields other than the value. So, we have a pointed to the parent, a pointed to the left child and a pointed to the right child.

(Refer Slide Time: 10:31)

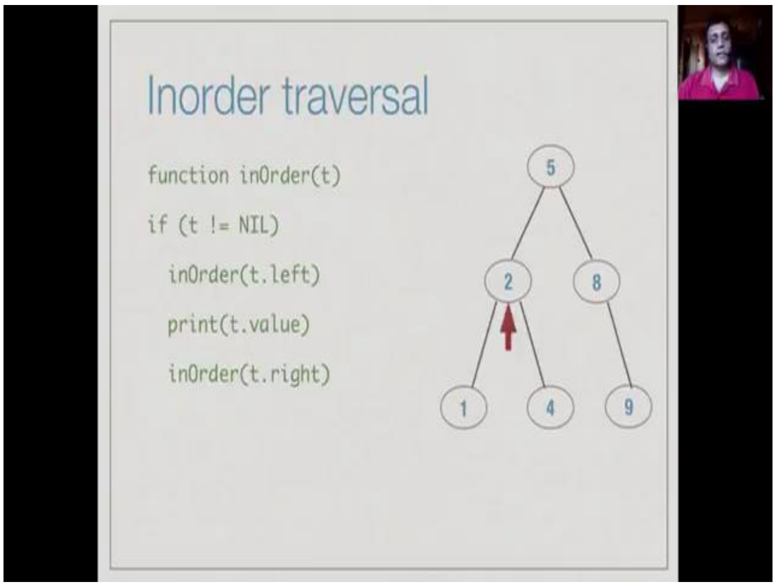


Inorder traversal

```
function inOrder(t)
if (t != NIL)
  inOrder(t.left)
  print(t.value)
  inOrder(t.right)
```

So, the first thing that we can do with a binary search tree is to list out it is values in sorted order, this is called a in order traversal. So, in order traversal is recursive traversal it is a way of walking through the tree. So, that we first walk through the left sub tree, then the current node and then the right sub tree. For example, if you perform a in order traversal here, we start at the root and then we have to first walk on the lefts up tree, so we walk to the left child.

(Refer Slide Time: 11:03)

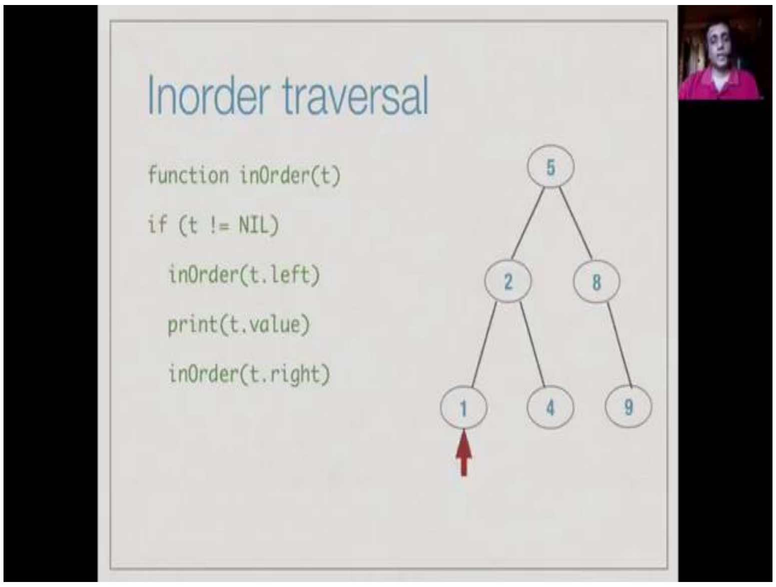


Inorder traversal

```
function inOrder(t)
if (t != NIL)
  inOrder(t.left)
  print(t.value)
  inOrder(t.right)
```

And then, once again we must walk on to the left of tree, so you walk to the left child.

(Refer Slide Time: 11:07)

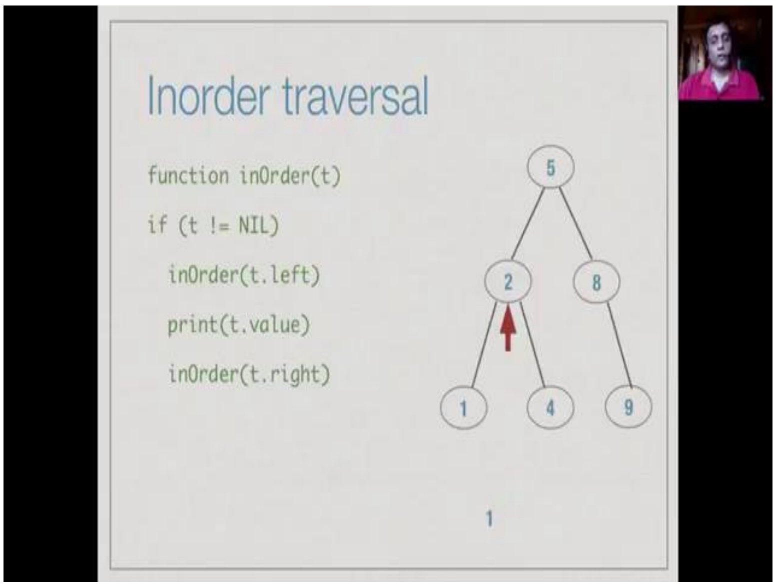


Inorder traversal

```
function inOrder(t)
if (t != NIL)
  inOrder(t.left)
  print(t.value)
  inOrder(t.right)
```

And now, there are no left child, so now we will print out 1.

(Refer Slide Time: 11:10)

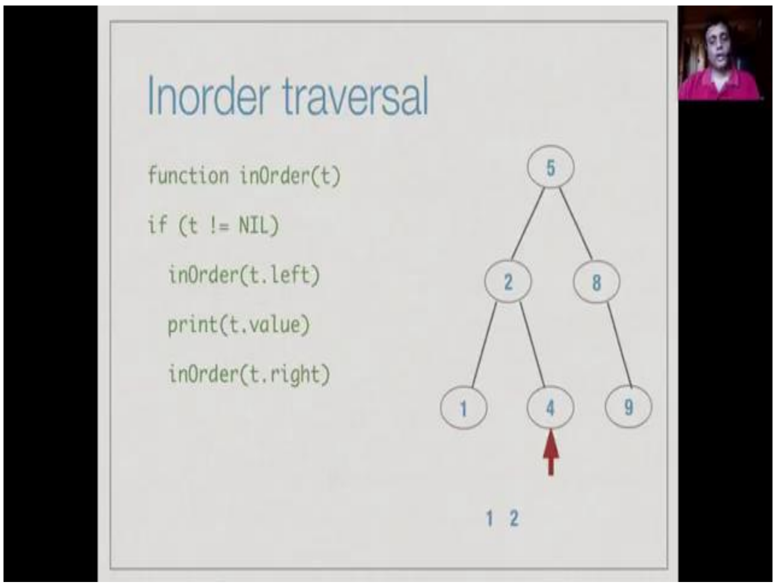


Inorder traversal

```
function inOrder(t)
if (t != NIL)
  inOrder(t.left)
  print(t.value)
  inOrder(t.right)
```

And move backup, because is no right child, now we print out 2.

(Refer Slide Time: 11:15)

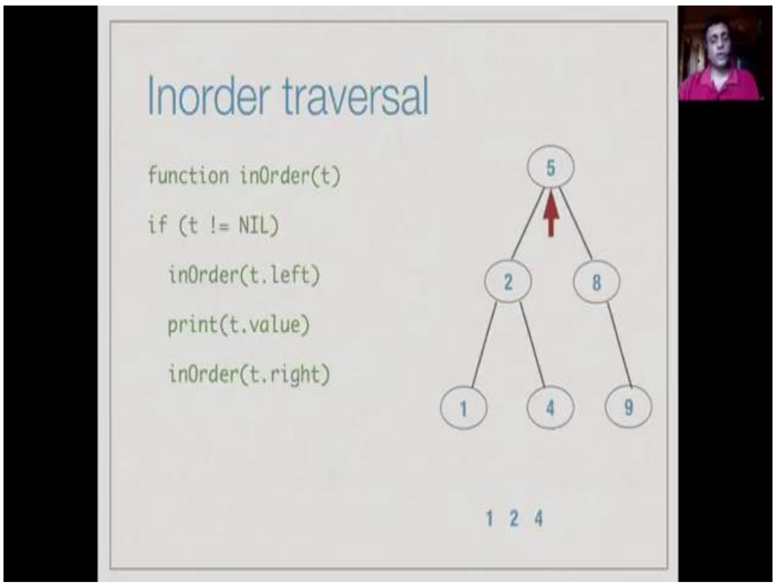


Inorder traversal

```
function inOrder(t)
if (t != NIL)
  inOrder(t.left)
  print(t.value)
  inOrder(t.right)
```

And go down to the right sub tree and now again because 4 has no left or right sub tree, so will print out and go backup.

(Refer Slide Time: 11:20)

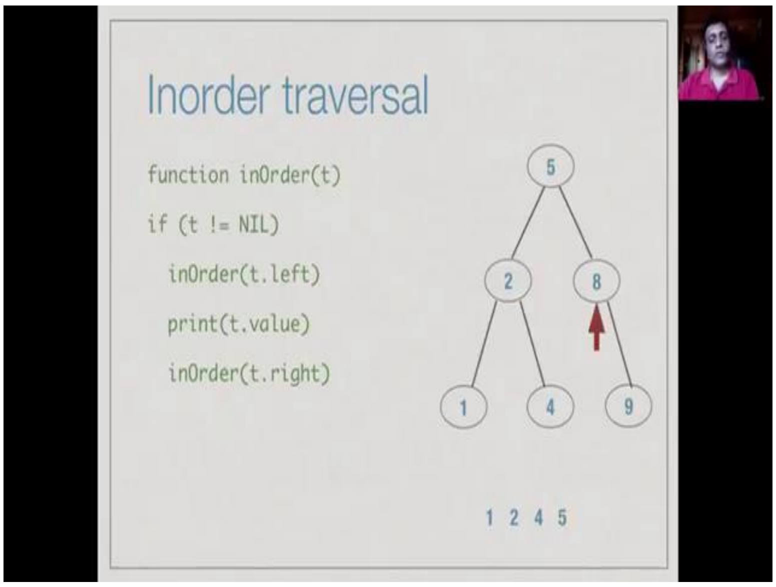


Inorder traversal

```
function inOrder(t)
if (t != NIL)
  inOrder(t.left)
  print(t.value)
  inOrder(t.right)
```

And since we already finish to we will end of that the root.

(Refer Slide Time: 11:26)

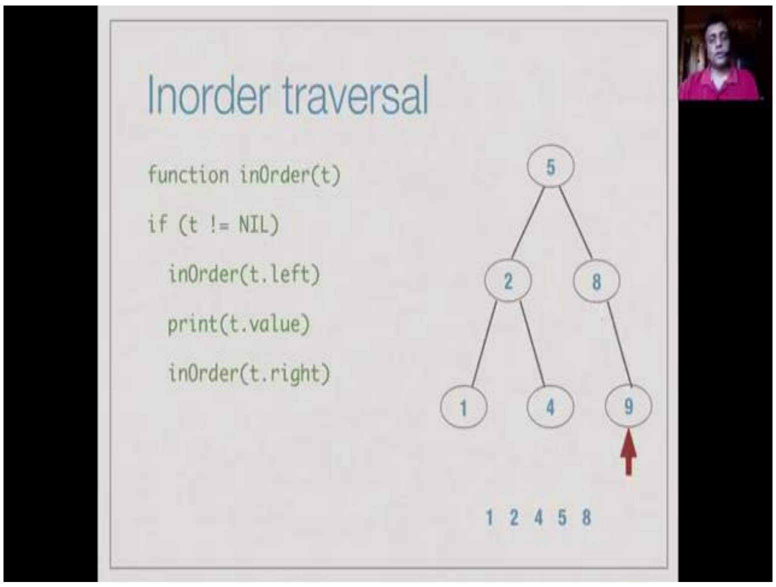


Inorder traversal

```
function inOrder(t)
if (t != NIL)
  inOrder(t.left)
  print(t.value)
  inOrder(t.right)
```

Have we will print 5 move to the sub tree, then because there is no left sub tree will print 8 and go to the right sub tree.

(Refer Slide Time: 11:30)



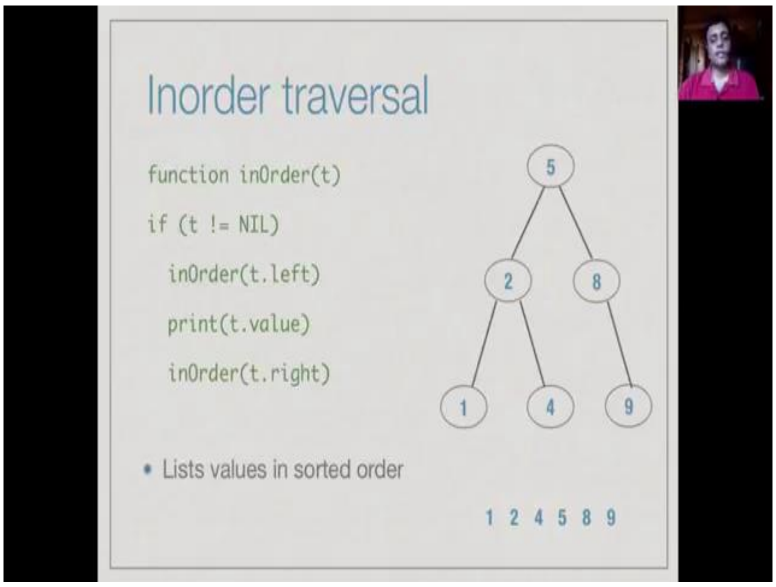
Inorder traversal

```
function inOrder(t)
if (t != NIL)
  inOrder(t.left)
  print(t.value)
  inOrder(t.right)
```

1 2 4 5 8

And finally, will print 9.

(Refer Slide Time: 11:33)



Inorder traversal

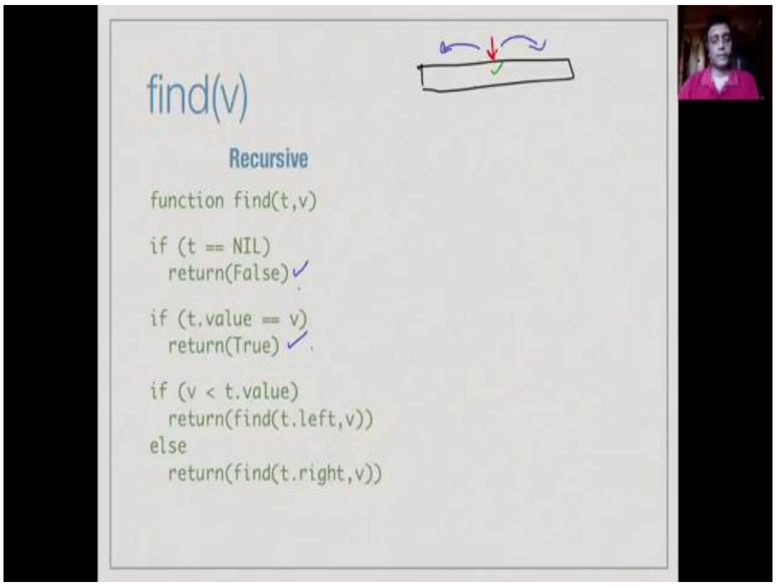
```
function inOrder(t)
if (t != NIL)
  inOrder(t.left)
  print(t.value)
  inOrder(t.right)
```

- Lists values in sorted order

1 2 4 5 8 9

So, it easy to see the because of this property, we know that everything to the left is smaller than that value and everything to the right is bigger than the value. So, this is like a recursive, in this is very similar to the partitioning of quick sort in a sense. So, we have already partition the values of the smaller values are to the left. So, there are bigger values to the right and if you recursively follow this partitioning to list out we will; obviously, get the values in sorted array.

(Refer Slide Time: 11:59)



find(v)

Recursive

```
function find(t,v)

if (t == NIL)
  return(False)

if (t.value == v)
  return(True)

if (v < t.value)
  return(find(t.left,v))
else
  return(find(t.right,v))
```

So, now searching for a tree a value in a binary search tree is very much like binary search. Remember, in binary search you have an array and then we start at the midpoint and then if you find it you say yes; otherwise, if it is smaller you go and to left; otherwise, one is bigger we look at the right. So, we have a very similar thing, so we want to find a value v in a tree, if the tree is empty of course, we say that is not there, so we return false.

If the current node has the value, then we have founded and return 2, on the other hand if have you not founded it, then since we have a tree we look to see whether the value is smaller than the current value, it is smaller than we recursively search in the left and return whatever we find there with it found there we say to true, it is not found there which we say to the tree; otherwise, we search recursively the right, so this is very similar binary search.