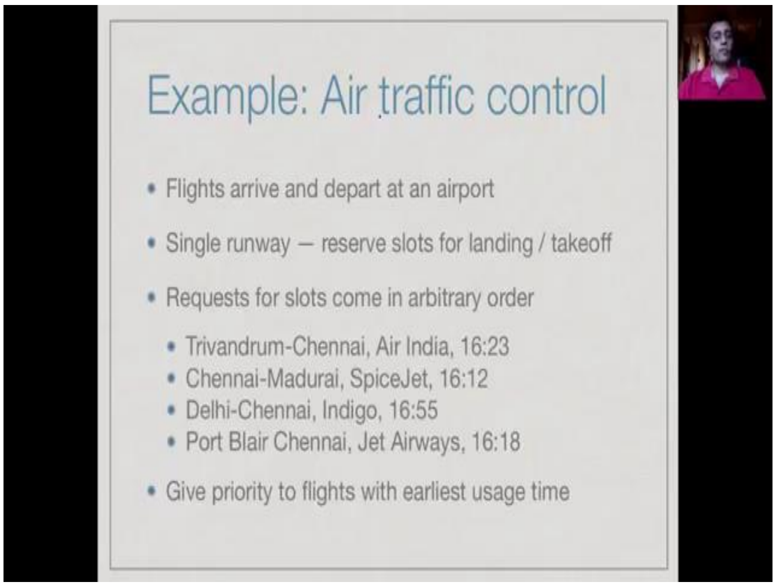
Design and Analysis of Algorithms
Prof. Madhavan Mukund
Chennai Mathematical Institute

Module – 01
Lecture - 39
Search Trees

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

(Refer Slide Time: 00:05)



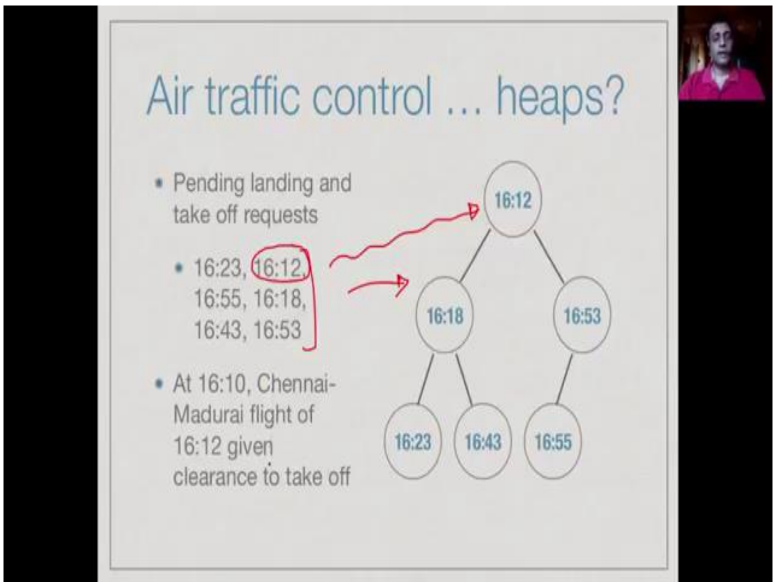
Example: Air traffic control

- 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)

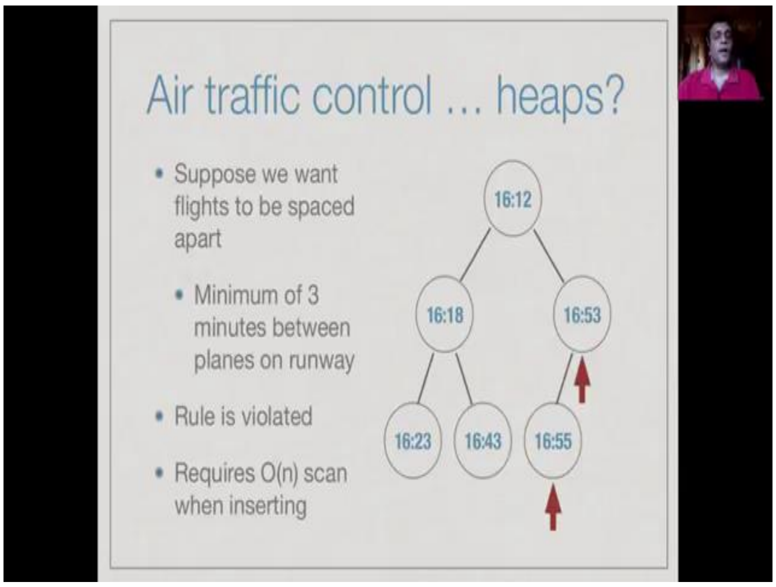


Air traffic control ... heaps?

- 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?

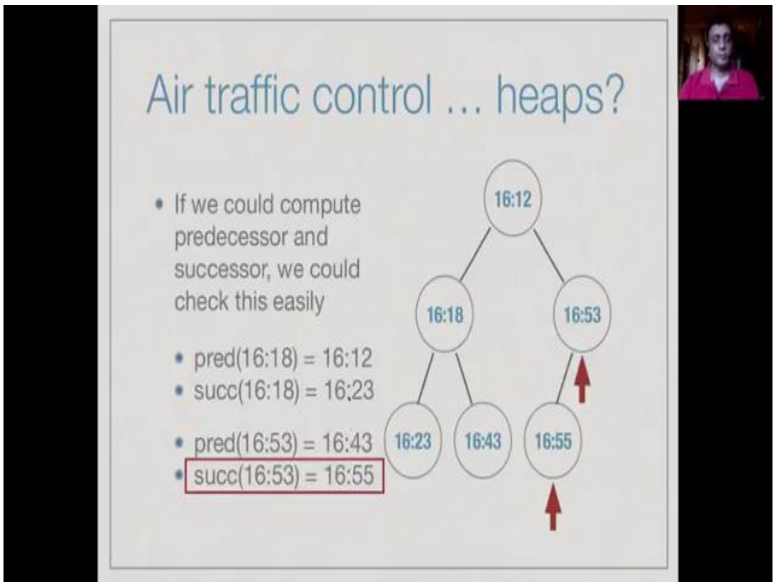
- Suppose we want flights to be spaced apart
  - Minimum of 3 minutes between planes on runway
- Rule is violated
- Requires O(n) scan when inserting

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?

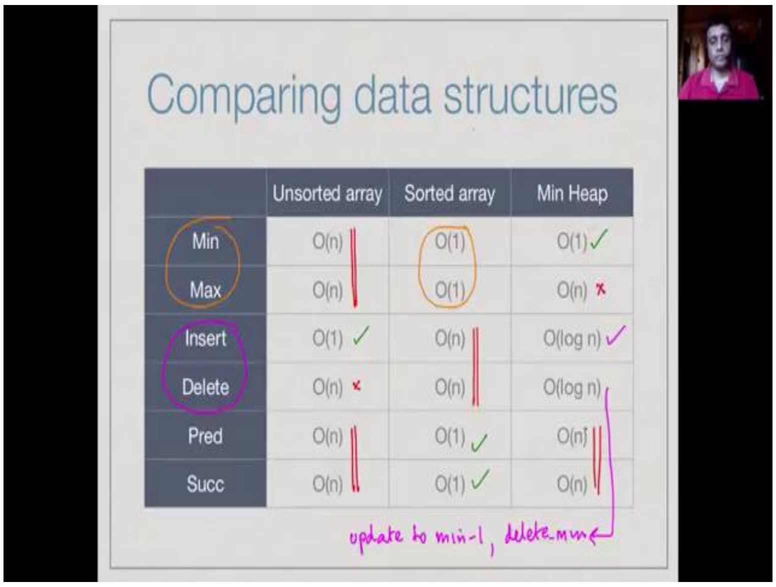
- If we could compute predecessor and successor, we could check this easily
  - pred(16:18) = 16:12
  - succ(16:18) = 16:23
  - pred(16:53) = 16:43
  - succ(16:53) = 16:55

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.

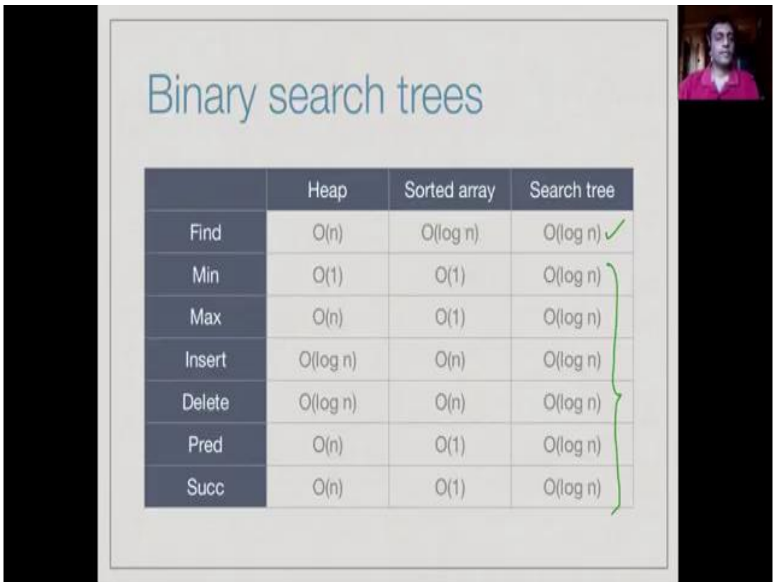
The worst is an unsorted array, because it will be anywhere and both look at linear time and then depending on whether we have a min heap or the max heap, one of them is good and the other one is bad. So, in a min heap we can find the minimum with the root, the maximum could be anywhere and likewise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

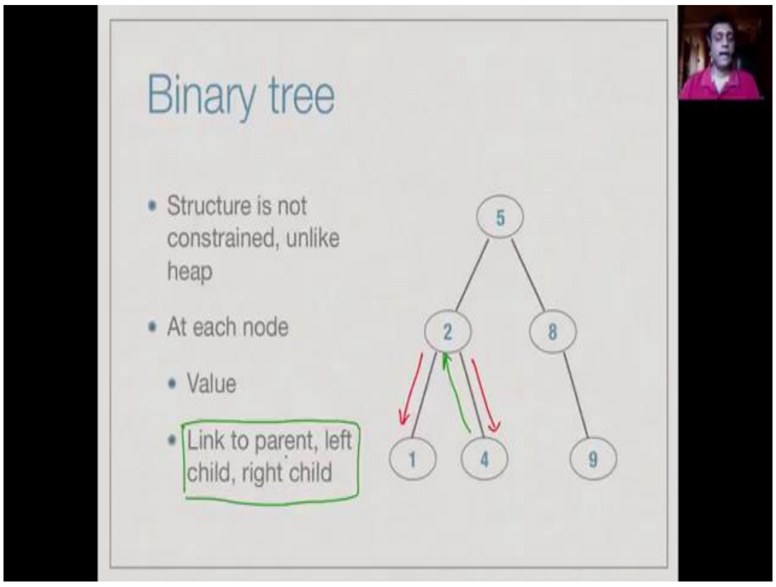


Binary search trees

	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)



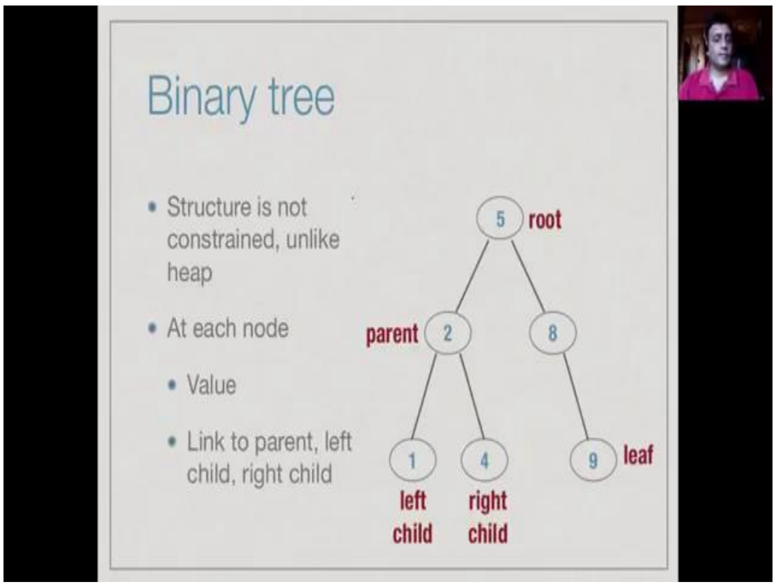
Binary tree

- 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)



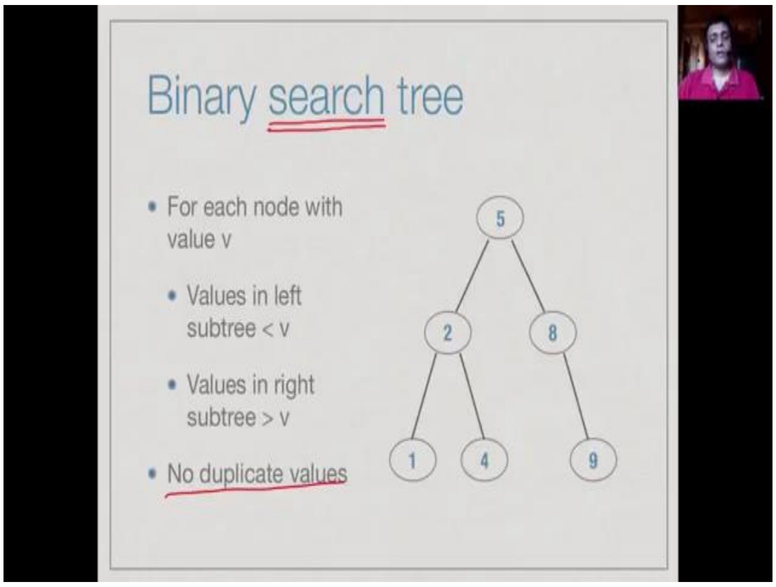
Binary tree

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

5
root
parent
2
8
1
4
9
leaf
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)



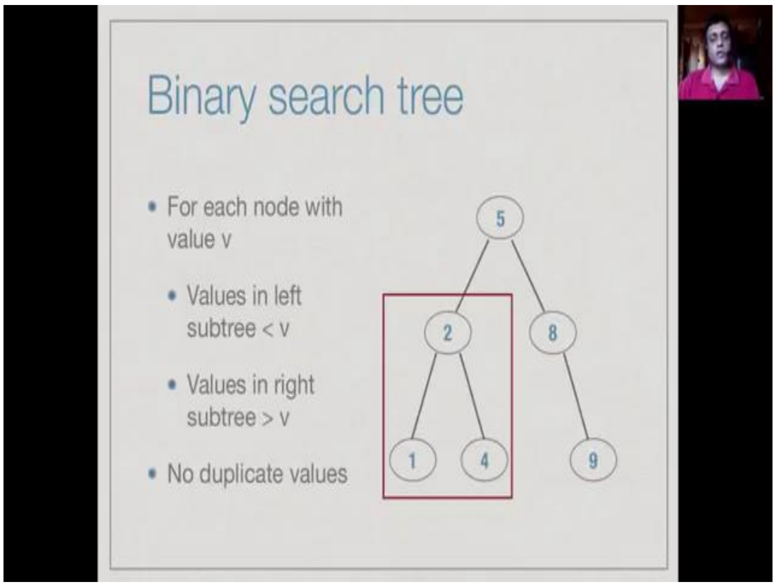
Binary search tree

- 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)

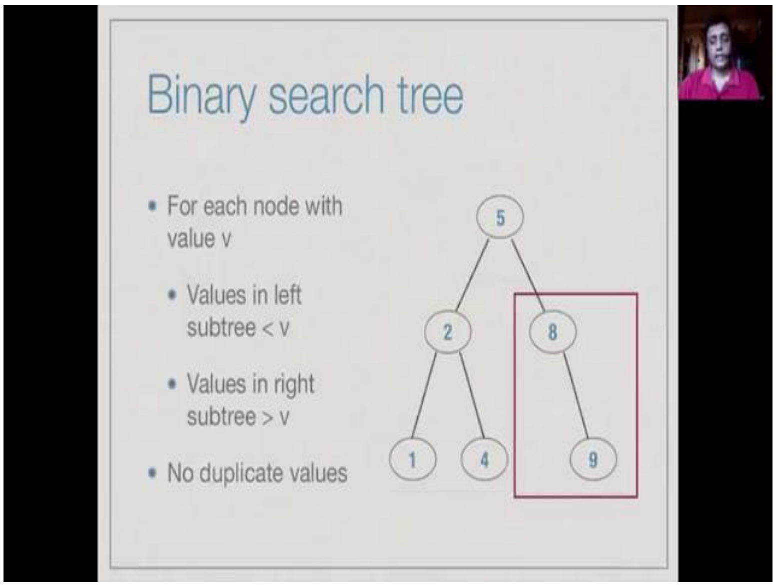


Binary search tree

- 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)

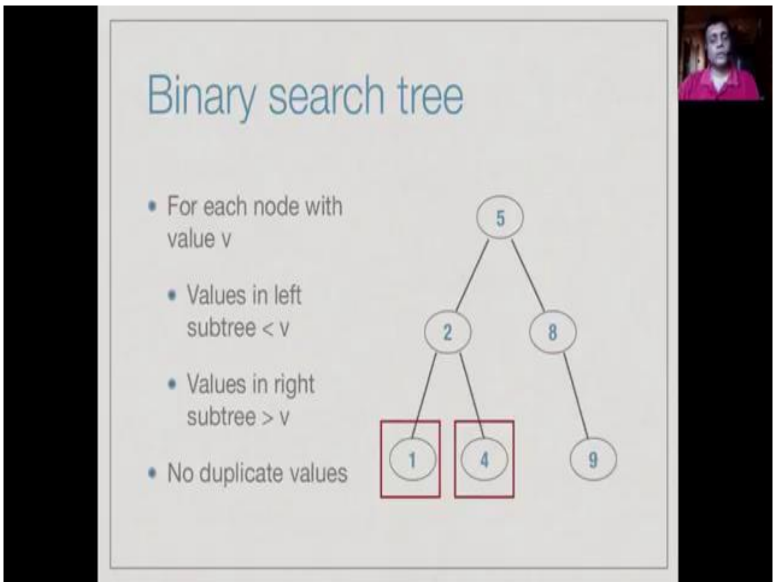


Binary search tree

- 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)

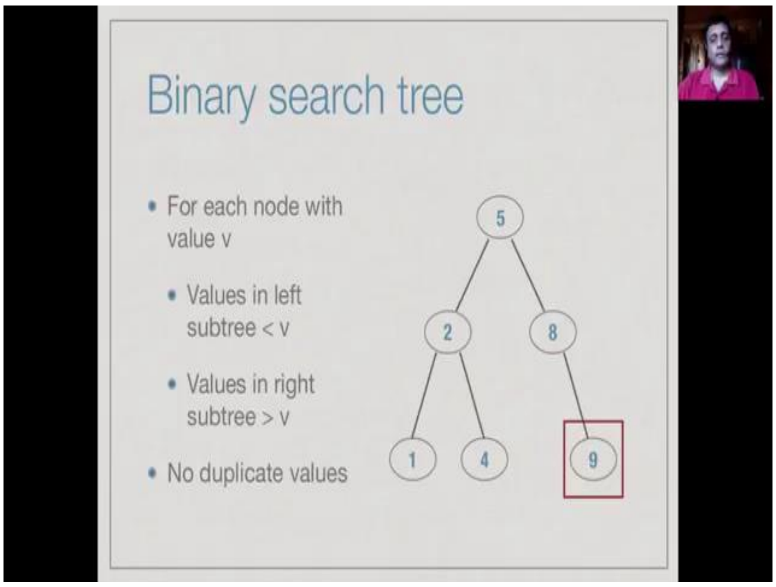


Binary search tree

- 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)

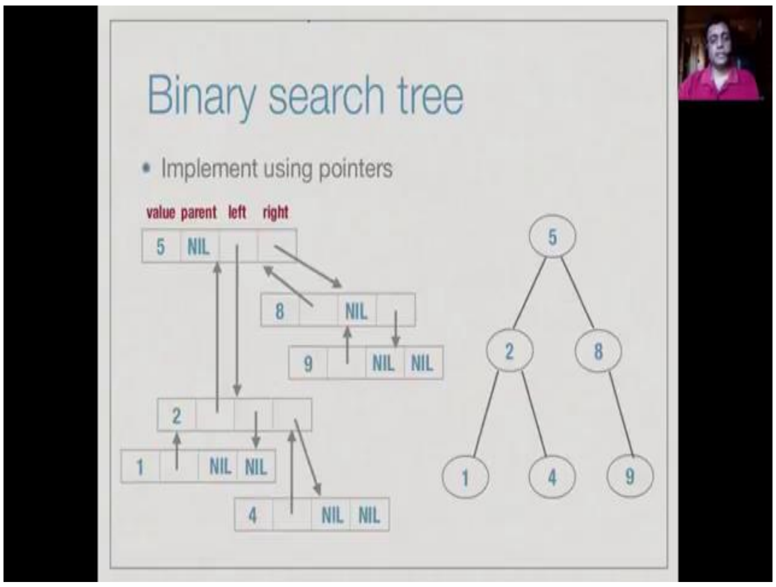


Binary search tree

- 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)

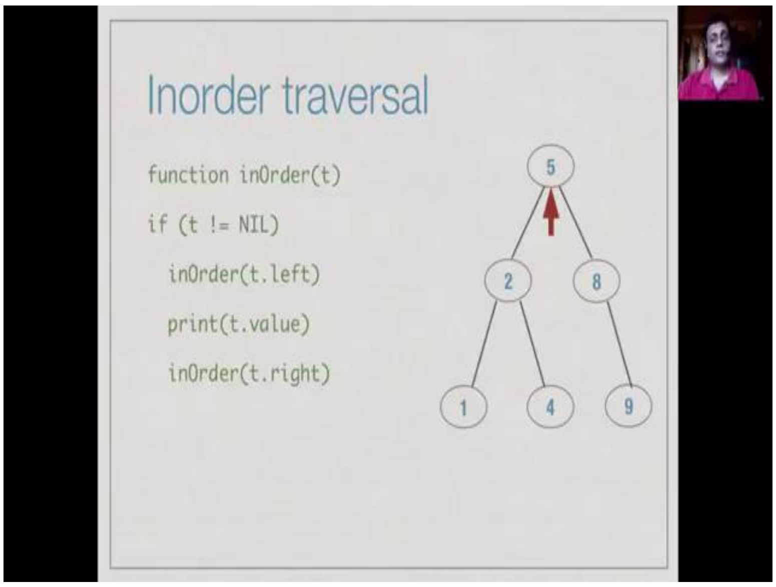


Binary search tree

- 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)

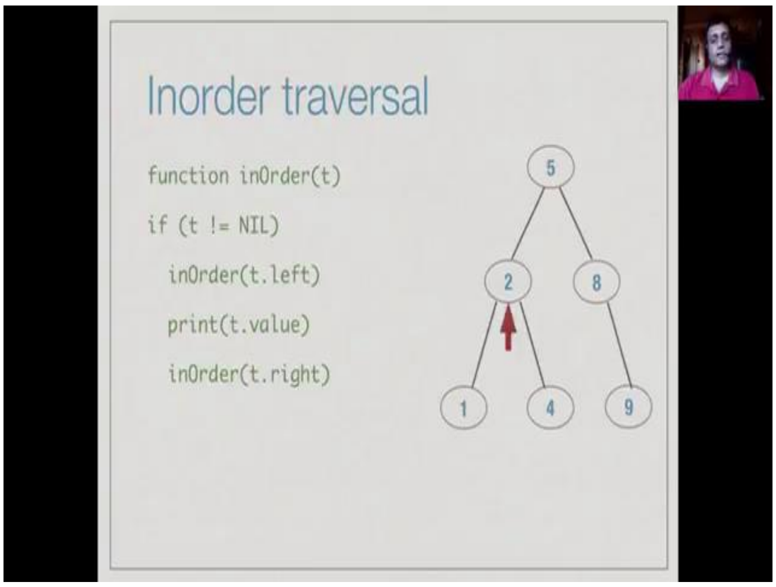


Inorder traversal

```
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)

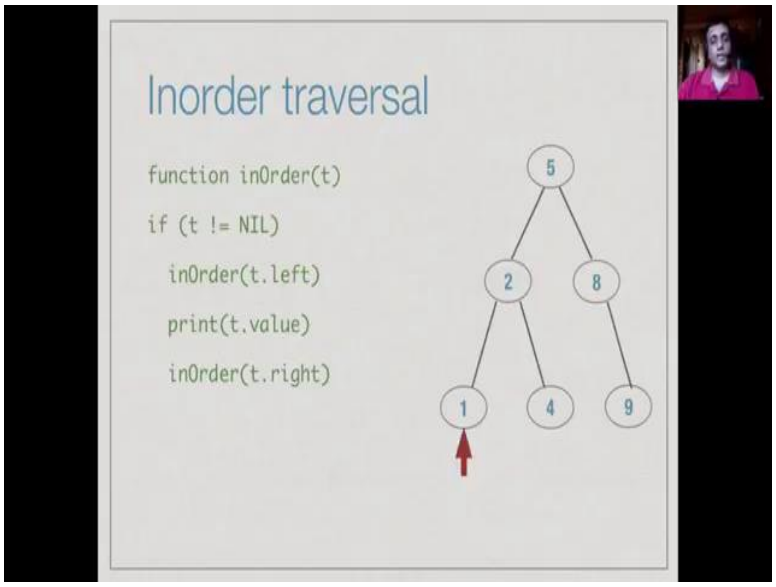


Inorder traversal

```
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)

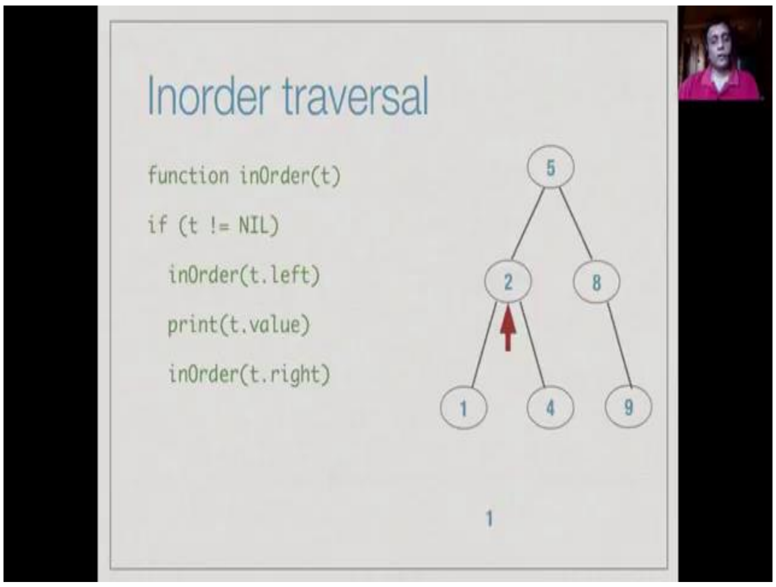


Inorder traversal

```
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)

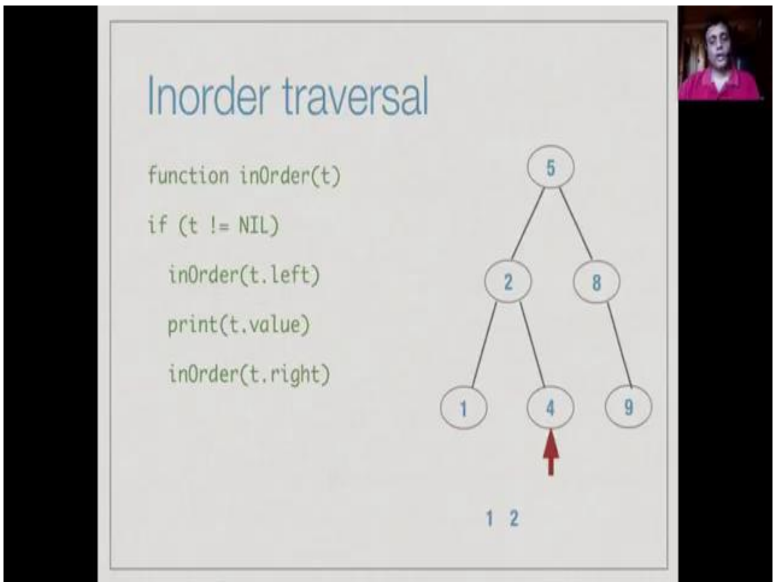


Inorder traversal

```
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)

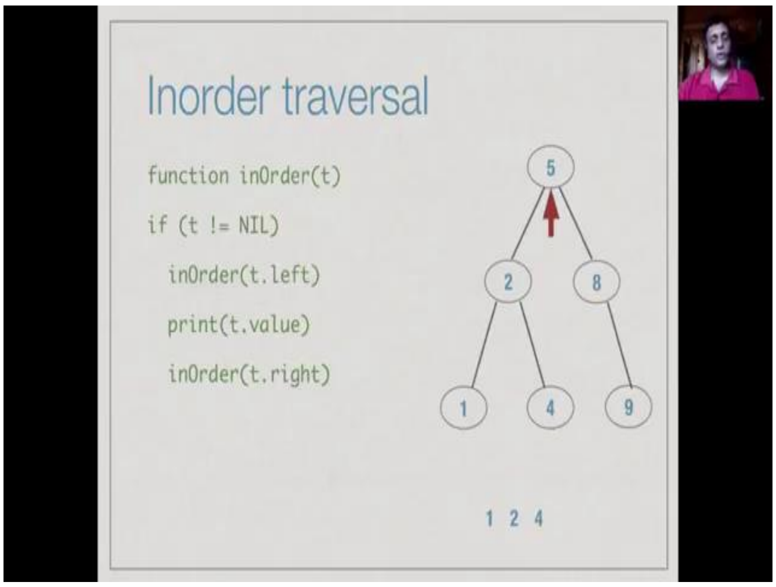


Inorder traversal

```
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)

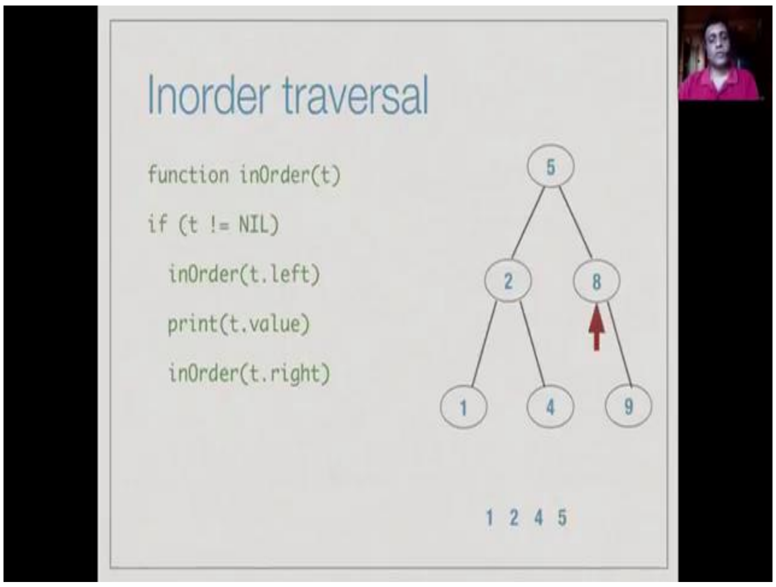


Inorder traversal

```
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)

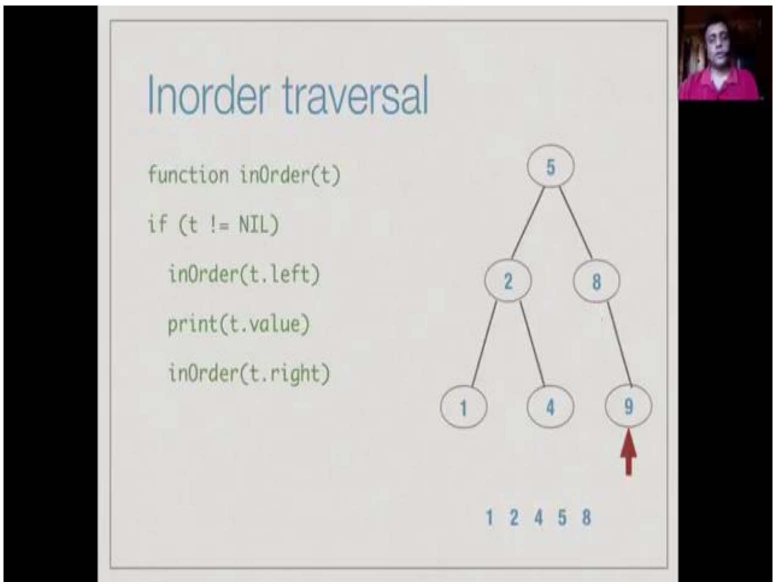


Inorder traversal

```
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)

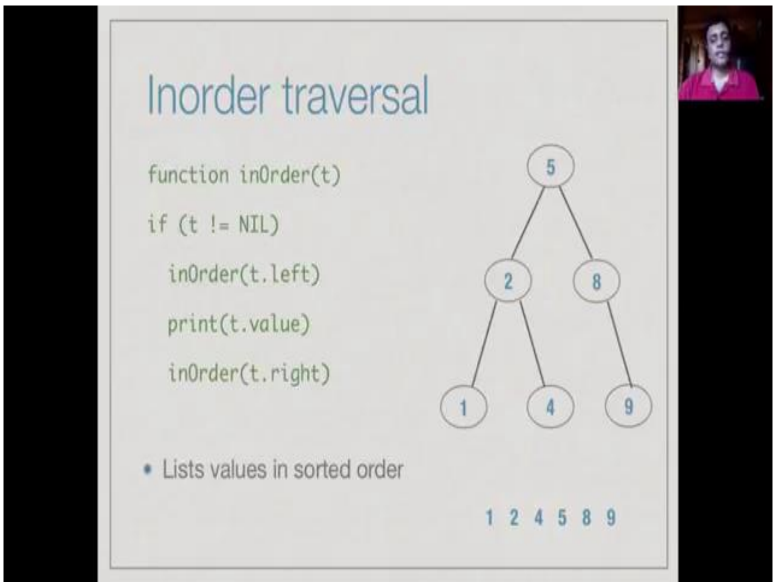


Inorder traversal

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

And finally, will print 9.

(Refer Slide Time: 11:33)



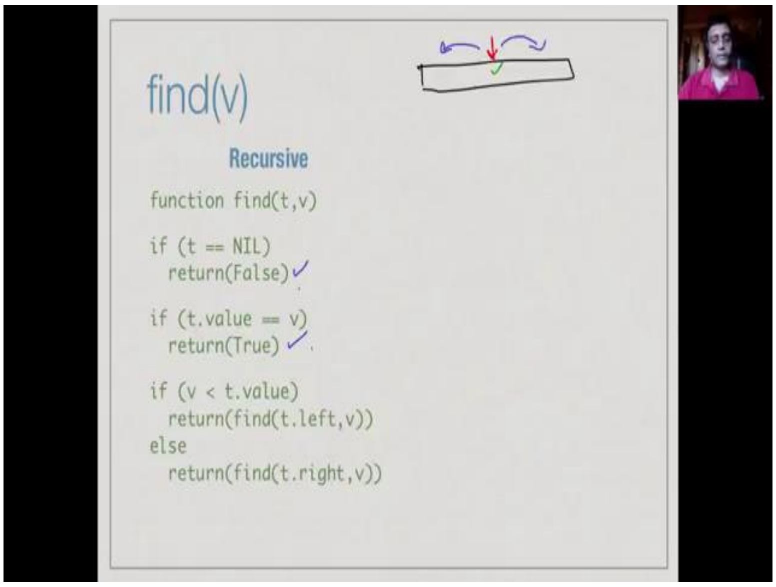
Inorder traversal

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

- Lists values in sorted order

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)

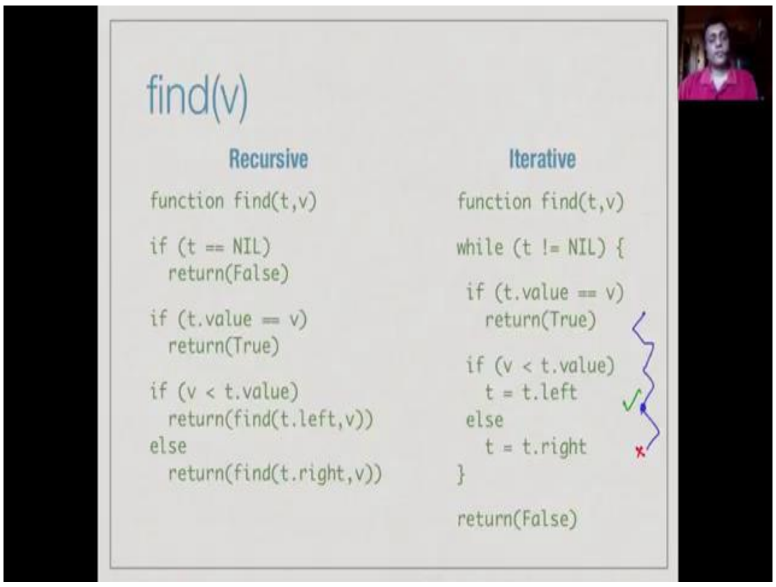
Recursive

```
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.

(Refer Slide Time: 12:59)



find(v)

Recursive

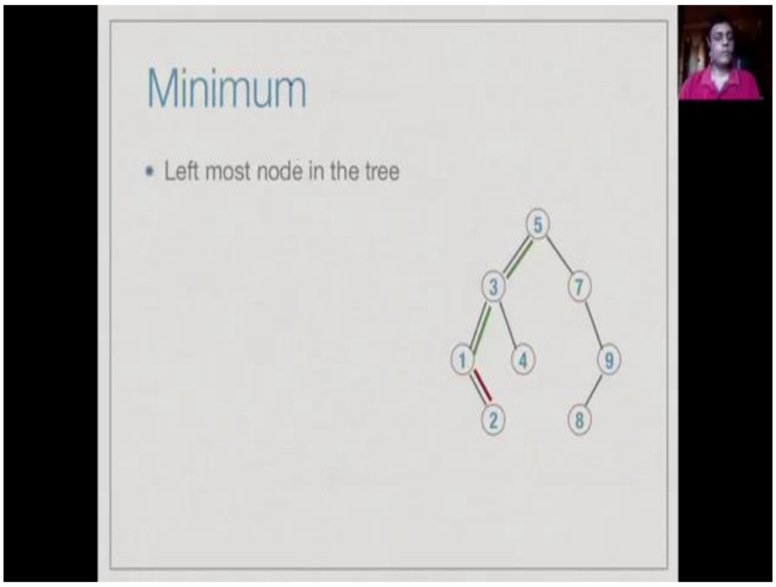
```
function find(t,v)

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))
```

Iterative

```
function find(t,v)

while (t != NIL) {

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

 if (v < t.value)
   t = t.left
 else
   t = t.right
}

return(False)
```

Now, like binary search you can do iterative version of this, so you start at the root and then so long as it is not nil you trust on the path. So, the current values we return true; otherwise, you walked on to the left are you actually start to the root and you kind of trace a path you can values are that find. And then when you reach are node you say yes, on the other hand if you reach a point where you cannot going further, if you run out of nodes, if you come all the way down and then you say there is no extension, where I can find it then we will safe also. So, this is the simple recursive and iterative version of find.

(Refer Slide Time: 13:34)

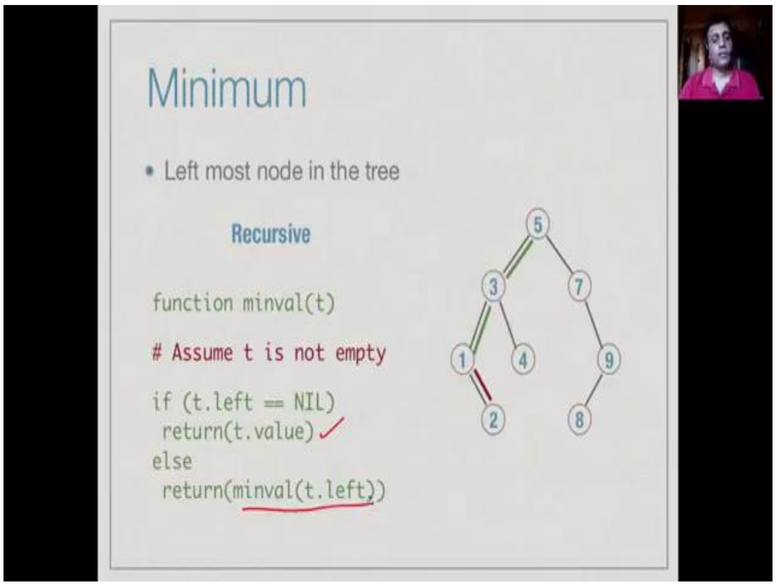


Minimum

- Left most node in the tree

So, the next two operations insert we want to do your finding the minimum and the maximum. So, the minimum node in the tree is the left most node, in other wards it is the node that you reach if you go and as for as you can follow only left edges. So, in this case from 5 for go left I go to 3, from 3 if I go to left I go to 1 and then I cannot go left there is something below one it is on the right, but never the less in cannot go left one must be the minimum element on the tree, because everything that is smaller than something will be 2 is left, so if I cannot go left is nothing smaller than 1. So, this is the reason why the left most value in the tree will be the minimum.

(Refer Slide Time: 14:14)



Minimum

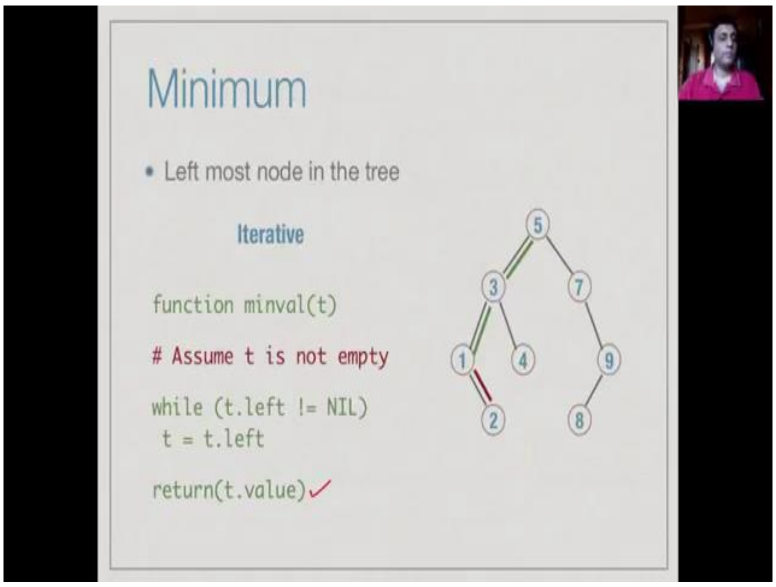
- Left most node in the tree

Recursive

```
function minval(t)

# Assume t is not empty

if (t.left == NIL)
 return(t.value)
else
 return(minval(t.left))
```

So, we can easily find it recursively, now we will typically use this assuming the trees not empty, it simplifies a little bit of coding. So, assuming the trees not empty we do not have to check any special condition and given a error when it is not empty, when it is empty. So, we assume it is not empty, so if we can go left we can we do, so if the t dot left is nil then this is the minimum value and we return otherwise, we recursively search for the minimum value on the left.

(Refer Slide Time: 14:47)



Minimum

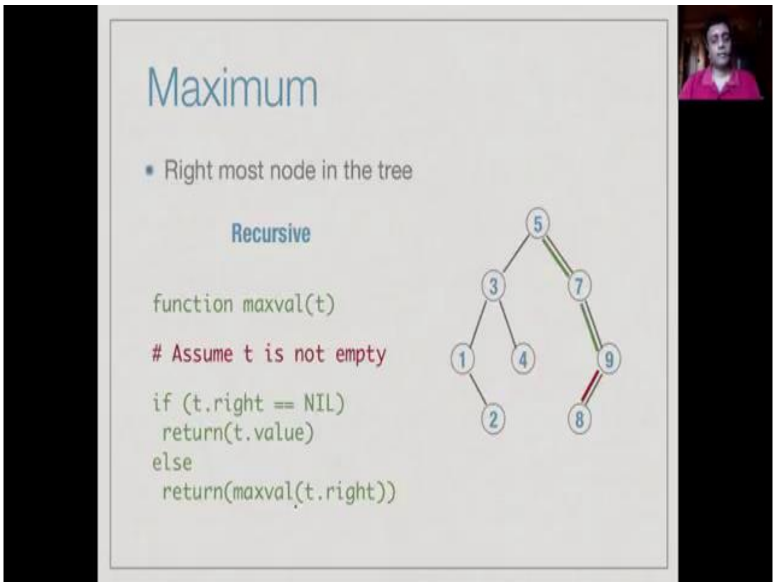
- Left most node in the tree

Iterative

```
function minval(t)
# Assume t is not empty
while (t.left != NIL)
 t = t.left
return(t.value)✓
```

And again there is the very simple iterative version, we start with the current thing and we keep going left as long as we can. So, long as we do not it will nil and when reach this point why t dot left is nil, we come out with this loop and you return the value at this point, so here from since we would start with 5 t dot left is 3, t dot left is 1, t dot left is nil. So, this at this point we stop, so t is pointing to 1, so therefore t dot value is 1 and this is the value by return, so we can find the minimum.

(Refer Slide Time: 15:14)



Maximum

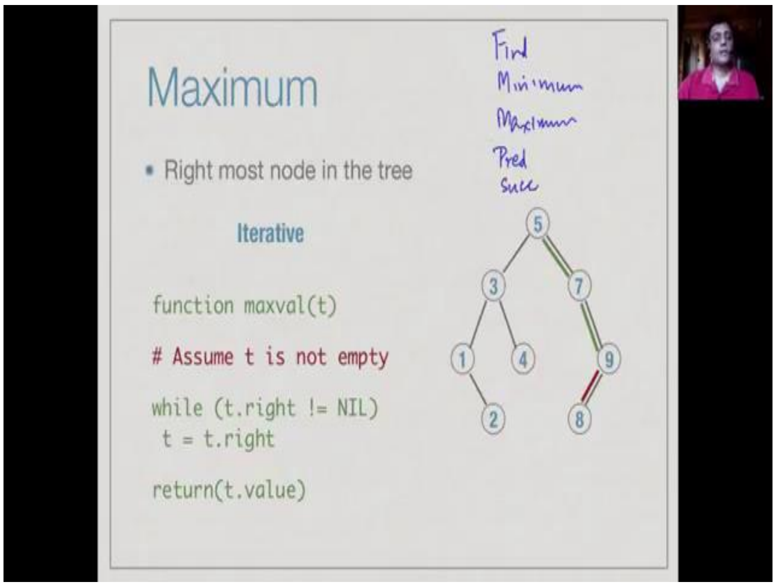
- Right most node in the tree

Recursive

```
function maxval(t)
# Assume t is not empty
if (t.right == NIL)
 return(t.value)
else
 return(maxval(t.right))
```

So, symmetrically the maximum is the right most value on the tree, as you go right you go bigger, if you cannot go right any more this is nothing which is bigger than that node. So, here from 5 we can go to 7 to 9 and 9 is the biggest value, because a next node which all though it is below 9 in the tree it is through the left and therefore, smaller than 9. So, we have a symmetric recursive solution maxval, we checks the right if the right is nil then we are at the maximum we return the value if the right is not nil, then we recursively search the right for the maximum.

(Refer Slide Time: 15:46)



Maximum

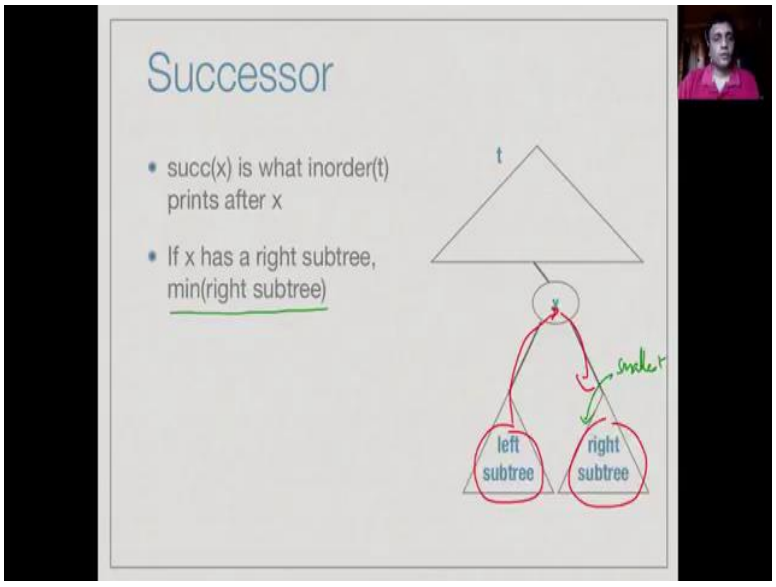
- Right most node in the tree

Iterative

```
function maxval(t)
# Assume t is not empty
while (t.right != NIL)
 t = t.right
return(t.value)
```

And again iterative version of the same, we just follow chase point as to the right. So, as long as right t dot right is not nil, we move from t to t dot right and when we cannot move any more here of the maximum value, so we return that value. So, this point we have done find minimum and maximum in a search tree. So, these three operations we have done, now if you remember we have do predecessor and successor.

(Refer Slide Time: 16:20)



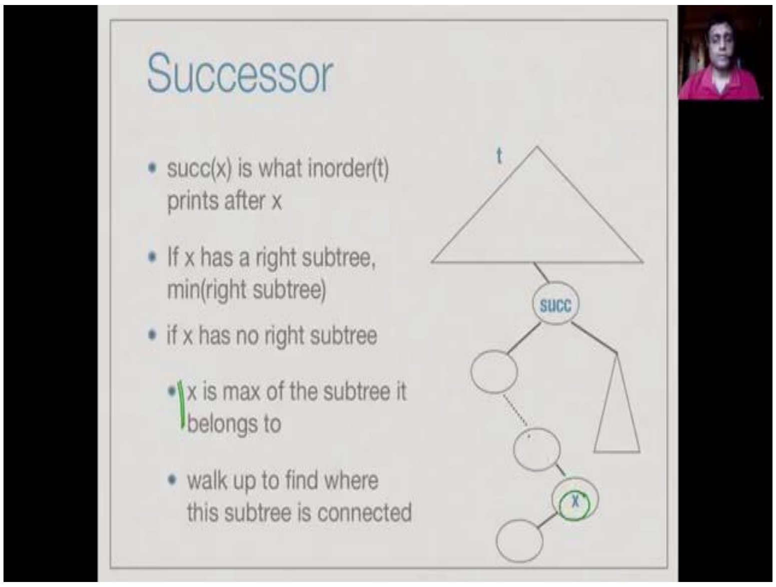
Successor

- succ(x) is what inorder(t) prints after x
- If x has a right subtree, min(right subtree)

So, let us first look at successor, so recall that the successor of x in the list, supposed to be the next value, the next smallest value after x the list. So, if we print it out in sorted order, it will be the value that would appear to the right of x and the in order traversal of a tree prints out the values in sorted order. So, it is effectively what in order would print immediately after x. So, we know that the way that in order works, it prints the left sub tree then it prints x, then it prints right sub tree.

So, if you want to one that comes immediately after x, it will be the first value here. So, the smallest value in the right sub tree, in other words the minimum of the right sub tree and we already know how to compute the minimum of a tree, but they could be a possibility that we want to successor of a value with does not have a right sub tree, then what we do.

(Refer Slide Time: 17:16)

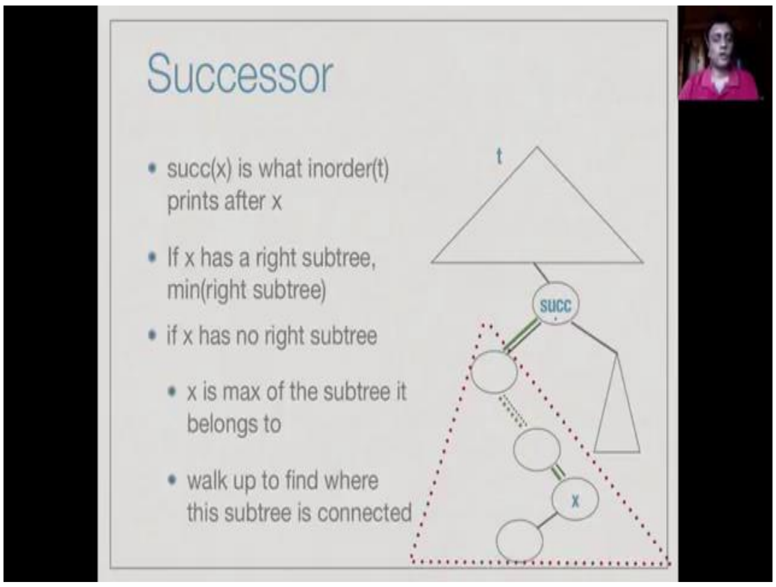


Successor

- succ(x) is what inorder(t) prints after x
- If x has a right subtree, min(right subtree)
- if x has no right subtree
  - x is max of the subtree it belongs to
  - walk up to find where this subtree is connected

So, the first observation is that if a value has no right sub tree, then it must be the maximum of the sub tree to which it belongs. So, in this case we look at the value x, it has no right sub trees, so it is the maximum of some sub tree.

(Refer Slide Time: 17:29)



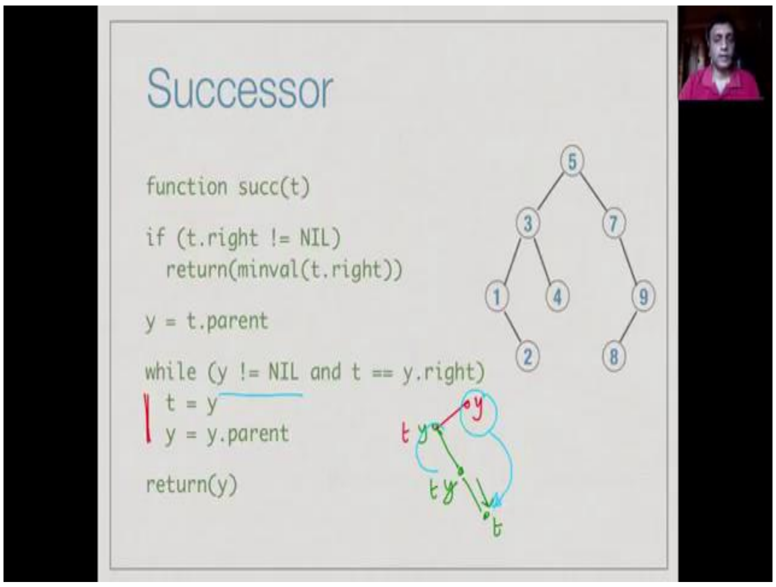
Successor

- succ(x) is what inorder(t) prints after x
- If x has a right subtree, min(right subtree)
- if x has no right subtree
  - x is max of the subtree it belongs to
  - walk up to find where this subtree is connected

So, we have to first identify this sub tree, so how do we identify the sub tree, will be wake up and find out how this sub tree is connected. So, we get to x there following is sequence of right pointers. So, we follow those right pointers backwards and when we a follow a left pointer, we know that we have come out of this sub tree. So, therefore that node must be the root of the sub tree below is this is the left sub tree, therefore that is the next value.

So, this entire red, block sub tree will be printed out and shorted order ending with x, after this the root will be printed thus which is this node which we have label as successor. So, therefore, that is the node that we want took a designate in the success for x.

(Refer Slide Time: 18:10)



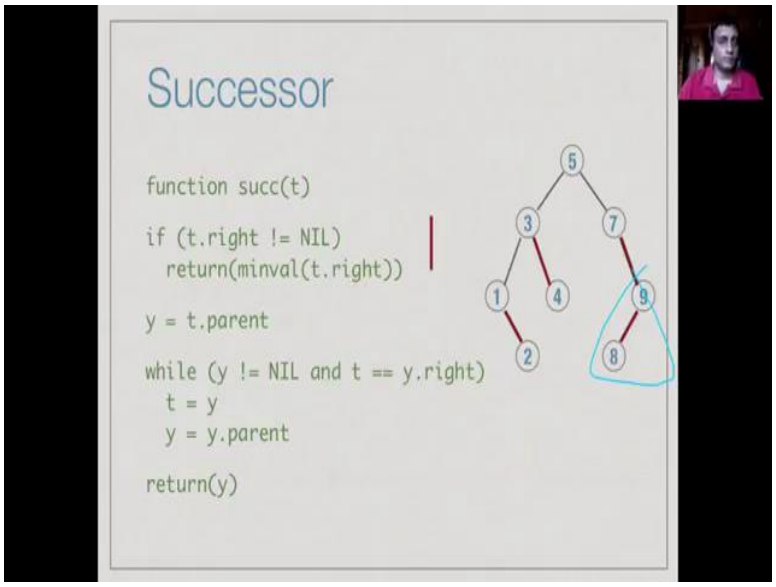
Successor

```
function succ(t)

if (t.right != NIL)
  return(minval(t.right))

y = t.parent

while (y != NIL and t == y.right)
  t = y
  y = y.parent

return(y)
```

So, here is a simple pseudo code for this, so we want to find the successor of node t. Now, if this node has a right pointer, then we just return the minimum value of the right sub tree, on the other hand if it does not have a right, then we need to go up and find the first place where the part turns right. So, we start with t and then we compute it is parent which we call y now. So, long as this direction is right, we keep moving up's we go to one more parent and we move t up's, so that is what happening in this loop here.

So, we move t and y up and then at some point we have move t and y will go this way. So, it will be y dot left and therefore, this now this y is the successor of this original node t and then there is one special case where we have reach the top and we do not ever turn right and that is case where we as you as started from the maximum value overall in the tree, in which case we reach in then it has no successive of we will just return nil.

(Refer Slide Time: 19:19)

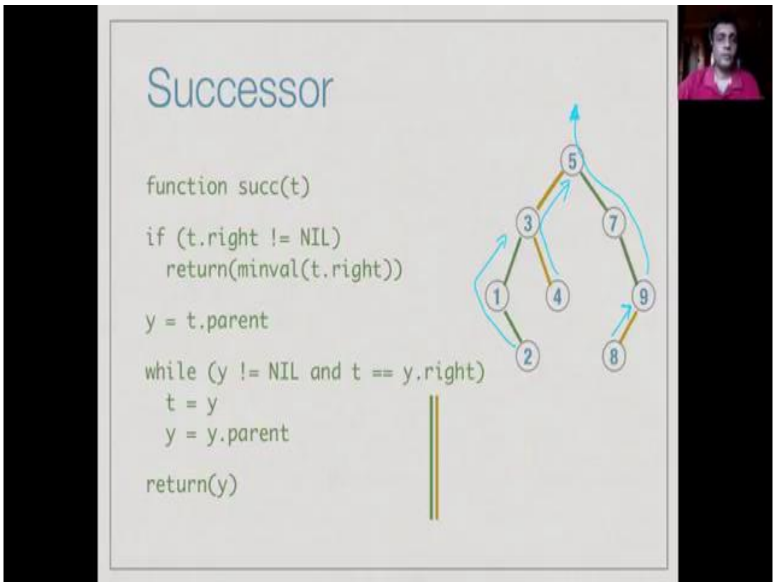


Successor

```
function succ(t)

if (t.right != NIL)
  return(minval(t.right))

y = t.parent

while (y != NIL and t == y.right)
  t = y
  y = y.parent

return(y)
```

So, let us look at this particular thing, so for 3 for incidents the right sub tree exists 4 and this smaller is value there is 4. Therefore, there is a success of for 1 the smallest values 2, so that is the successor for 7 the minimum value in this sub tree is 8. So, 8 is a successor or 7, so these all come out of the basic case for you have a right sub tree.

(Refer Slide Time: 19:42)

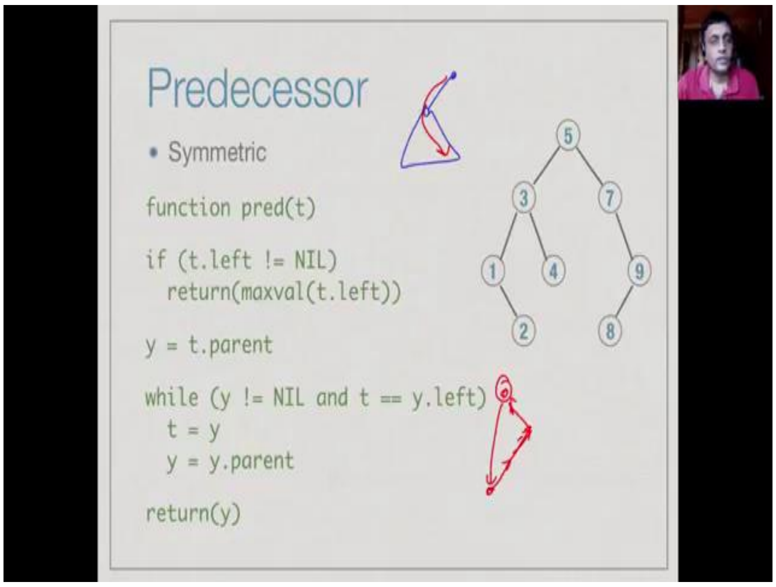


Successor

```
function succ(t)

if (t.right != NIL)
  return(minval(t.right))

y = t.parent

while (y != NIL and t == y.right)
  t = y
  y = y.parent

return(y)
```

On the other hand, if you do not have a right sub tree then for instance you started 2, then you wake up and that they where you turn right, you find that 3 is the successive or from 4 you wake up and you very term right you find that 5 is successive. Similarly, for I 8 you immediately turn right, so 9 is successive and finally, for 9 you will come in and you will ((Refer time: 20:02)) this nil case. So, 9 will say that there is no successive, because it is the large possible value.

(Refer Slide Time: 20:09)



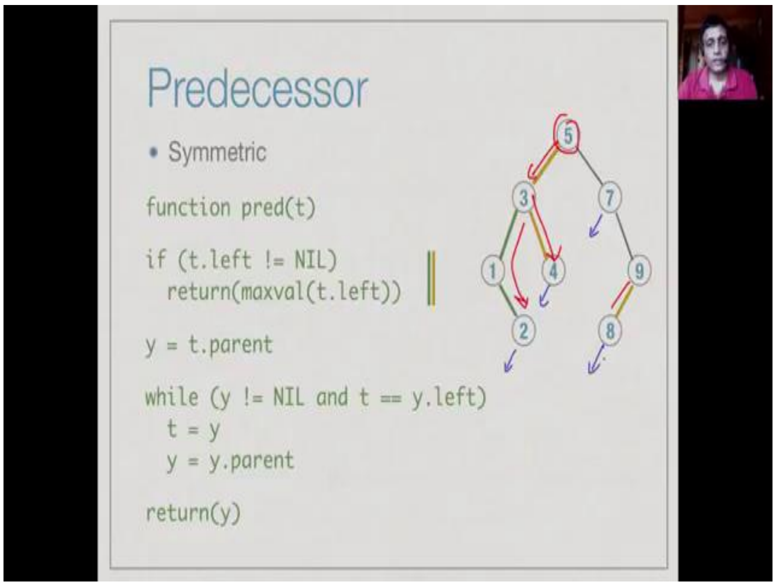
Predecessor

- Symmetric

```
function pred(t)

if (t.left != NIL)
  return(maxval(t.left))

y = t.parent

while (y != NIL and t == y.left)
  t = y
  y = y.parent

return(y)
```

So, the situation for the predecessor is symmetric, so if we have a left sub tree, then the predecessor is the maximum value in this. So, we go down left and then we go all the way right and if we do not have a left sub tree, then it means that this is already the minimum value in it is sub trees. So, we walk back up following these sequence of left pointers, until we turn the other way and then this node we have turn is predecessors.

(Refer Slide Time: 20:41)



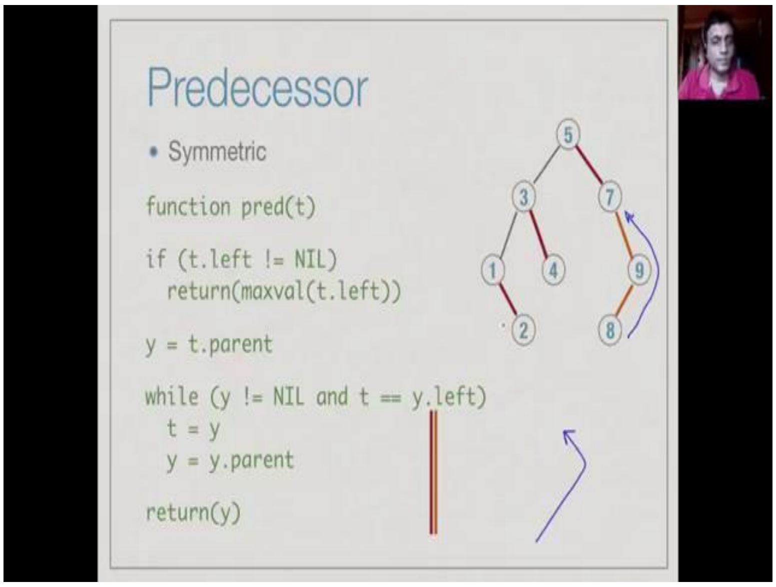
Predecessor

- Symmetric

```
function pred(t)

if (t.left != NIL)
  return(maxval(t.left))

y = t.parent

while (y != NIL and t == y.left)
  t = y
  y = y.parent

return(y)
```

So, for example if you look at these values which have left sub trees, so for 5 we go down the left sub tree and we find the right most value which is 4. So, 4 is the predecessor are 5, likewise 2 is the predecessor of 3 and 8 is the predecessor of 9, now we have this other value. So, which we cannot to left for example, we cannot go left it 2, we cannot go left it 4, we cannot go left it 7, cannot go left it 8.

(Refer Slide Time: 21:09)



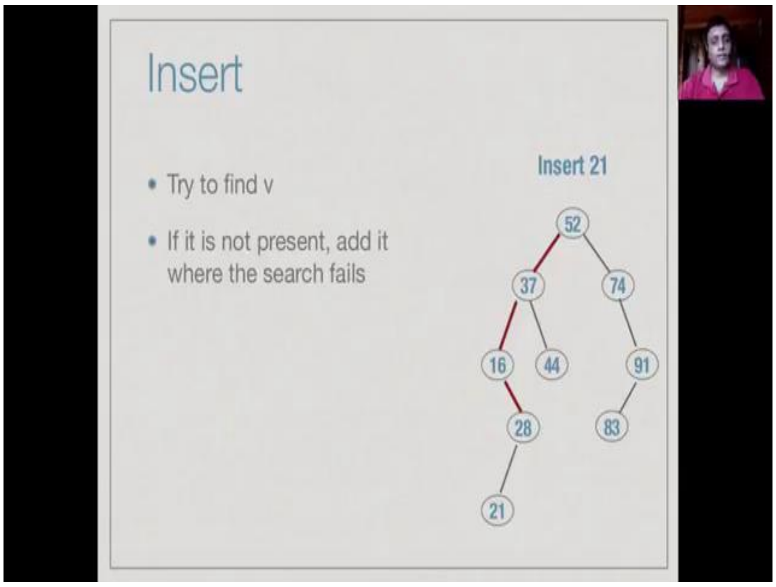
Predecessor

- Symmetric

```
function pred(t)

if (t.left != NIL)
  return(maxval(t.left))

y = t.parent

while (y != NIL and t == y.left)
  t = y
  y = y.parent

return(y)
```

So, what do we do in these cases we started 2 for example, and then we try to go up and where we turn right there we find. So, we come here and we find that one is a predecessor simulate from 4 if you go and so we a goal is to go right and then where we turn left if find predecessor. So, immediately turn left of these three nodes and then 8 we go right and where we turn left we find the predecessor. So, this is how the predecessor works it is exactly symmetric to the successive function.

(Refer Slide Time: 21:46)



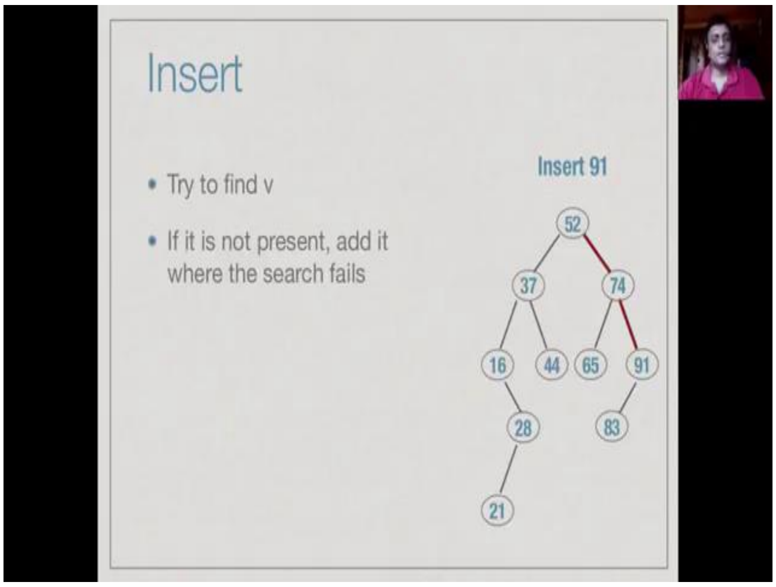
Insert

- Try to find v
- If it is not present, add it where the search fails

So, now let us see how to insert a value, inserting a value in a search tree fairly straight forward, there is only one place because of remember that the tree is listed in order will give us a shorted list. So, now after adding this value it must second give us a shorted list, so it is like saying that they was only one place to insert a value in a shorted list. So, is like insertion in a shorted list, except we have to find that correct place in the tree to hang it.

So, that when we do this in order traversal, it would be in the correct order when we list it out. And it turns out that basically we have to find out where it should be by searching for it and if it is not present, we add it by this search fields. So, for instants if you want to insert 21, then we will walked down the tree and we will look for the place where we should find 21. So, it is smaller than 52 we go a left, smaller than 37 we go a left, bigger than 16 we go right and then we find that at 28 those no value of 21, because there is no left, so this since it should be to the left it 28, we add it that.

(Refer Slide Time: 22:45)



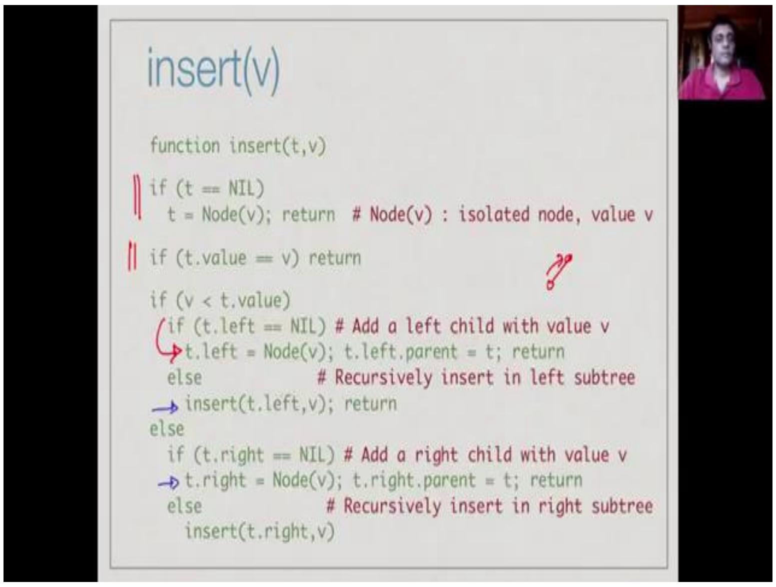
Insert

- Try to find v
- If it is not present, add it where the search fails

Now, for instants if we want to insert 65, then we will go right from 52 and now it is 74 we will look left, where they is nothing to the left. So, we will insert ((Refer Time: 22:59)), so it can happen at any position in the tree does not have to be at a leaf node like a 28, 28 was already a leaf and be inserted 21 below it. But, 74 had a right child, but we inserted something to it is left, because left child ((Refer Time: 23:11)).

Now, it could be that we said try to insert a value that is already there, for instants in my prompt to insert 91. So, we look for 91 and we find it, so we will interpret the insertion of 91 has something which does not disturbed it, because we earlier said we do not want to have duplicate values. So, when we insert we try to find if the fine fields we insert, if find the succeeds we do nothing.

(Refer Slide Time: 23:35)



insert(v)

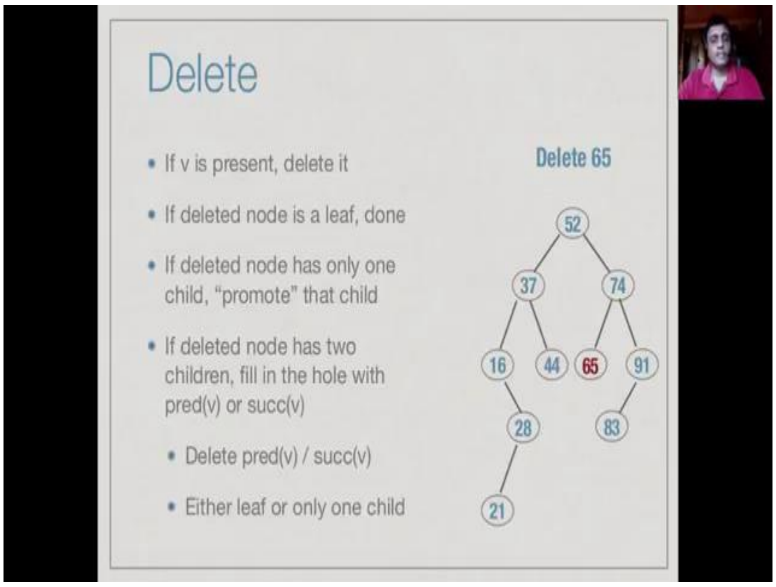
```
function insert(t,v)

if (t == NIL)
  t = Node(v); return  # Node(v) : isolated node, value v

if (t.value == v) return

if (v < t.value)
  if (t.left == NIL) # Add a left child with value v
    t.left = Node(v); t.left.parent = t; return
  else               # Recursively insert in left subtree
    insert(t.left,v); return
else
  if (t.right == NIL) # Add a right child with value v
    t.right = Node(v); t.right.parent = t; return
  else                # Recursively insert in right subtree
    insert(t.right,v)
```

So, this is very simple recursive way to do this, so first of all we have a special case which such as that trees empty, then we just create a new node and point to that node. So, this is an isolated node which has only a value v parent left and right to the all be nil, on the other hand if we find it, then we do nothing, now we have not found it. So, now if the values smaller then the value if the current node and if we have no left child, then we actually insert.

If we have to go left and we cannot go left, then we create a new node to are left and we make that node point here by it is parents. So, we create a new node here and we say that it is parent is ourselves. So, t dot left dot parent is t; otherwise, we just recursively insert 2 are left. So, if you do have a left we recursively insert, if we do not have a left then we create a node with v, this is the actually insert operation and we make it point towards through the parent.

Likewise, if we do want to go right and there is no right, we insert it to the right and we make it point to ourselves through it is parent; otherwise, we recursively insert to the right, so is the insert to the very straight forward function.

(Refer Slide Time: 24:51)

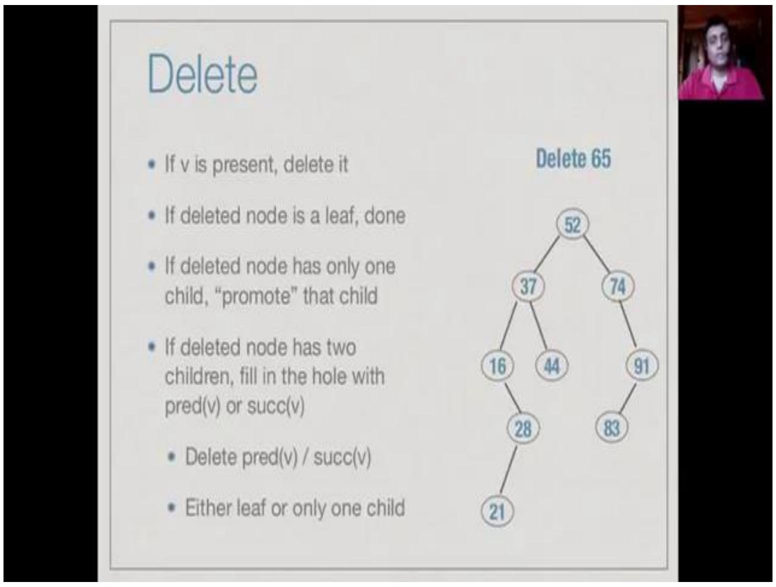


Delete

- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, "promote" that child
- If deleted node has two children, fill in the hole with pred(v) or succ(v)
  - Delete pred(v) / succ(v)
  - Either leaf or only one child

So, how do we delete a node, so with delete says, but we given a value v which we find v in the tree, you must delete the node containing. So, the basic case that is very simple to handle is one the node is a leaf node, because then we can just delete it and then it is just falls of the tree at the bottom. So, for instants if you want to delete 65, then we will search for 65 find it by the usual mechanism of following the left and right paths.

(Refer Slide Time: 25:24)

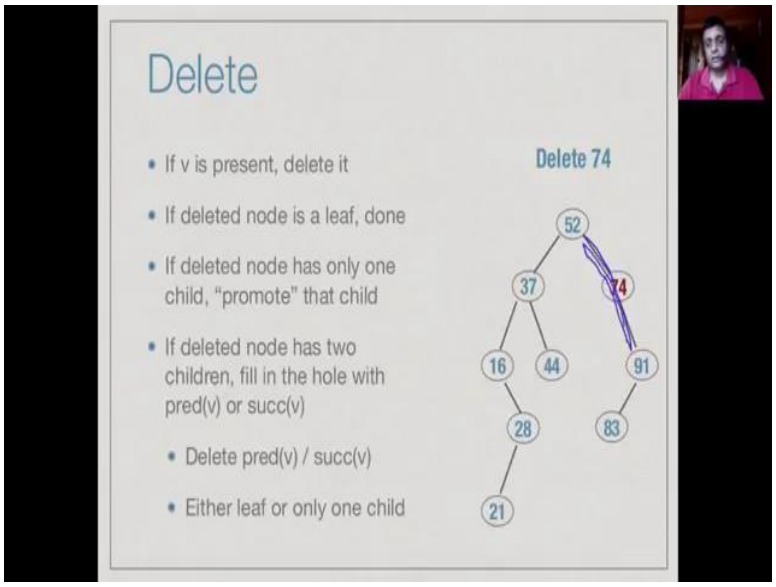


Delete

- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, "promote" that child
- If deleted node has two children, fill in the hole with pred(v) or succ(v)
  - Delete pred(v) / succ(v)
  - Either leaf or only one child

And then since it is a leaf node we can just remove this node from the tree and nothing happens, it remains valid search. Now, sometimes a deleted node might have only one child, for instance supposing we deletes 74.

(Refer Slide Time: 25:35)

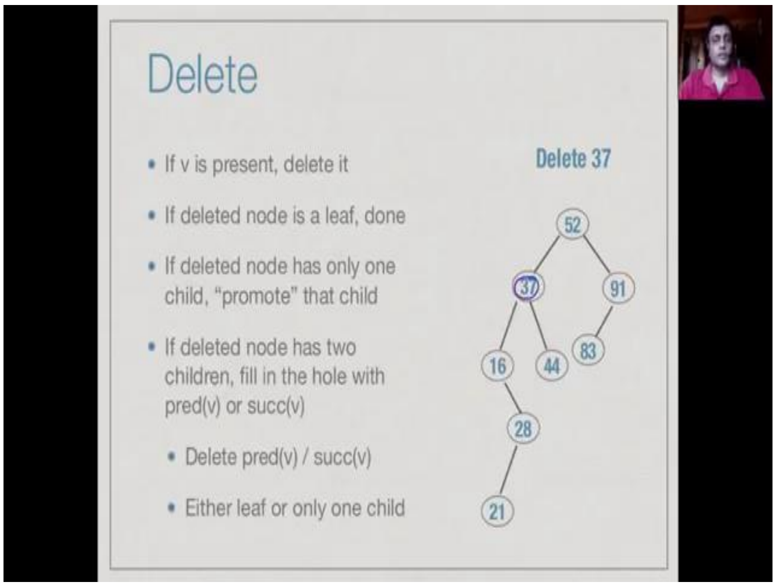


Delete

- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, "promote" that child
- If deleted node has two children, fill in the hole with pred(v) or succ(v)
  - Delete pred(v) / succ(v)
  - Either leaf or only one child

So, we come down to 74, now we want to delete this node, but it has the right child, but now what we can do, you can promote this child, so we can just kind of pertain that this link goes directly through this to 91.

(Refer Slide Time: 25:51)

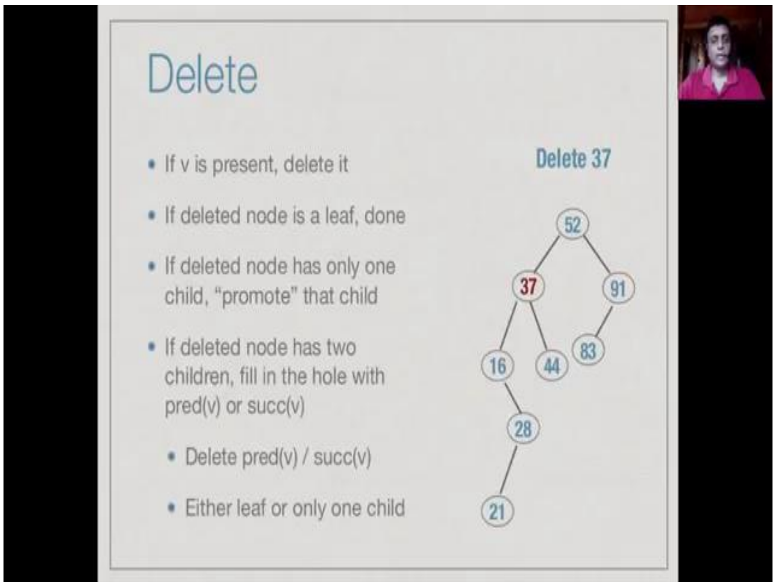


Delete

- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, "promote" that child
- If deleted node has two children, fill in the hole with pred(v) or succ(v)
  - Delete pred(v) / succ(v)
  - Either leaf or only one child

So, you just eliminate that node in between and the directly connect 52 to the successor of the node that is going to be delete. So, if there is only one child in this no problem, now what if the child delete, child is 2 deleted node as 2 children. So, supposing you want a delete 37 that is this one.

(Refer Slide Time: 26:12)

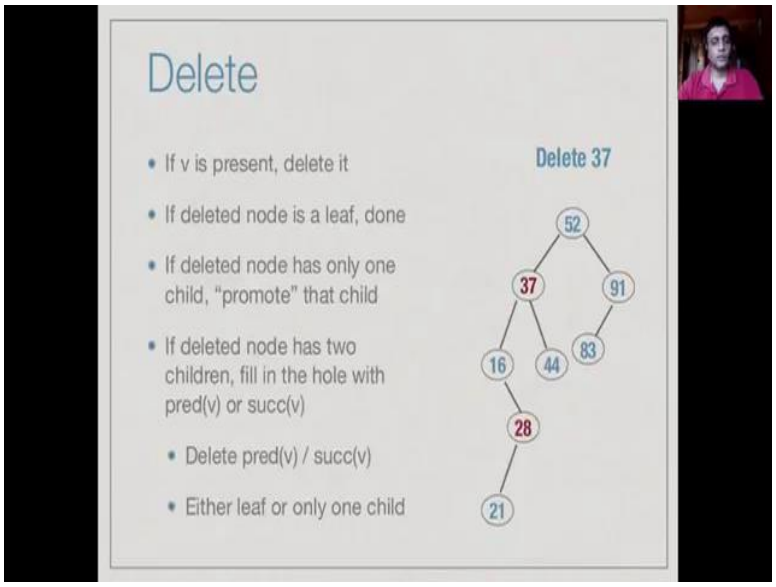


Delete

- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, "promote" that child
- If deleted node has two children, fill in the hole with pred(v) or succ(v)
  - Delete pred(v) / succ(v)
  - Either leaf or only one child

So, we identify 37 now 37 must go to at we cannot arbitrarily re structure the tree, because we will have 2 children and we do not know what to do with think. So, now one strategy that works is to make a whole there to remove the 37 and replace it by either it is predecessor or successive. So, supposing we identify it is predecessor with predecessor remember will be the biggest node a maximum in it is left sub trees.

(Refer Slide Time: 26:40)

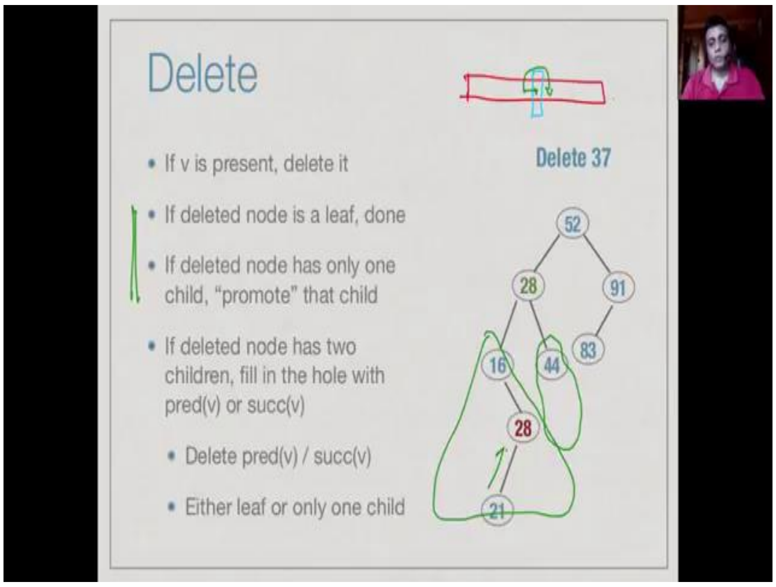


Delete

- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, "promote" that child
- If deleted node has two children, fill in the hole with pred(v) or succ(v)
  - Delete pred(v) / succ(v)
  - Either leaf or only one child

So, here it will be 28, so what we will do is, we will copy the 28 to this node which is to be deleted.

(Refer Slide Time: 26:43)


Delete

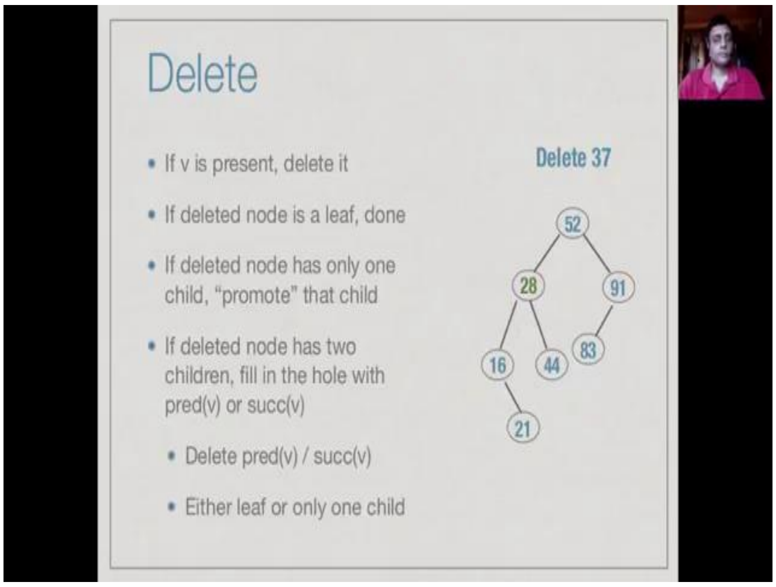
- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, "promote" that child
- If deleted node has two children, fill in the hole with pred(v) or succ(v)
    - Delete pred(v) / succ(v)
    - Either leaf or only one child

So, 37 the node is not be deleted, it is value has been replace by 28, now why is the value valid, because we want to preserve the shorted thing. So, if we have some list of shorted values and I delete something here, then what happens is that everything is to the right of the predecessor. So, I have basically move the predecessor to this point, where preserve the order between these elements in the shorted order.

So, moving the predecessor here I guaranty there anything that is to their height, remains bigger then this node and everything to the left remains smaller then this node, because that was the biggest values. Now of course, I have two copies at 28, so I am was remove that 28, so then I will focus on the left sub tree and I delete this predecessor value. But, the good thing of at the predecessor value is that is the right most valued, right most value means either it is a leaf or it has only left child.

So, we have back in the simpler case one of these two cases. So, we can just delete the predecessor of the using one of these two cases, either it is going to be a leaf it is just if also or I am going to promote this 21.

(Refer Slide Time: 27:46)

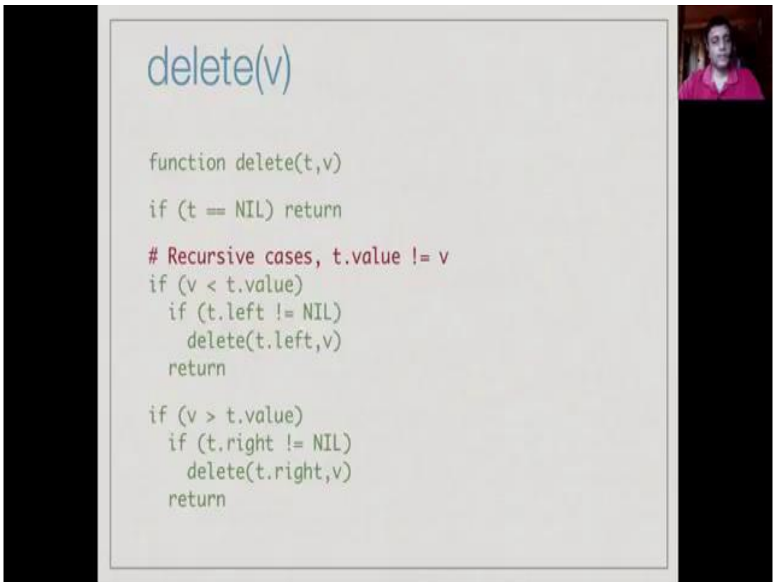


Delete

- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, "promote" that child
- If deleted node has two children, fill in the hole with pred(v) or succ(v)
  - Delete pred(v) / succ(v)
  - Either leaf or only one child

So, in this case the 21 will get on the right, so deleting in general consist of moving the predecessor to the current value and then deleting the predecessor in the left sub tree.

(Refer Slide Time: 27:58)

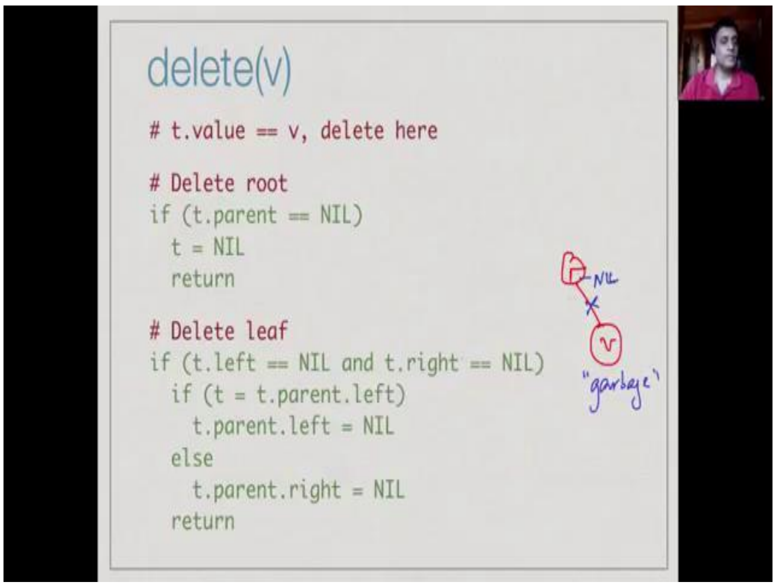


delete(v)

```
function delete(t,v)

if (t == NIL) return

# Recursive cases, t.value != v
if (v < t.value)
  if (t.left != NIL)
    delete(t.left,v)
  return

if (v > t.value)
  if (t.right != NIL)
    delete(t.right,v)
  return
```

So, it is a long piece of code, because we have been many cases, so first of all you with empty tree we do not nothing to we cannot delete. If form the other hand that value at the current node is not v, so if it is strictly less then, then it there is something to the left we delete from there, if it is strictly greater then at there is something to the right we delete from there, so these are the two recursive cases. So, if it is not strictly less than and not strictly greeter then in must be equal and we have to do some real deletes. So, now we have this three cases.

(Refer Slide Time: 28:28)



delete(v)

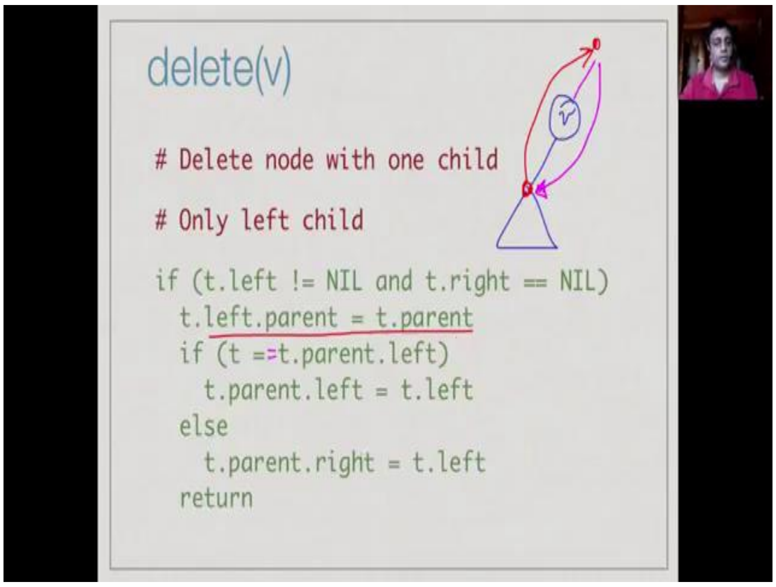
```
# t.value == v, delete here

# Delete root
if (t.parent == NIL)
  t = NIL
  return

# Delete leaf
if (t.left == NIL and t.right == NIL)
  if (t = t.parent.left)
    t.parent.left = NIL
  else
    t.parent.right = NIL
  return
```

So, the first case actually breaks up in two cases, if it is a leaf it could be a leaf because it is the root, the root and which is only one node. So, we will treat that slightly differently, so we will say that if it has no parent that is it is the root, then deleting the value actually makes it tree empty. So, we just a reset t to t the empty tree nil and the return. If on the other hand, it is a leaf node it has no children, then what we do is we try to deleted by just setting the parents value to be nil.

So, supposing I come here and this my leaf node and I want to delete this, then I look up and then I basically said the right pointer of this to be nil, which is essentially skills of this link and says that there is no right pointers. So, if the current node is the left child of it is parent as said the left child with the parent to nil. Otherwise, said the right child with the parent to null, so this of course, create some garbage because this node is now an accessible, in accessible from the tree.

But, we assume that this should be recovered and we do not at you worry about it, if you are doing this in a language likes see, then you have to be very careful to this store this back to the free space. But, in a other language which have garbage collection, this will they automatically be restored and garbage collection takes a over, but the point is that we have just simply removing it by resetting are parents point at be there, so this is the leaf case.

(Refer Slide Time: 29:54)



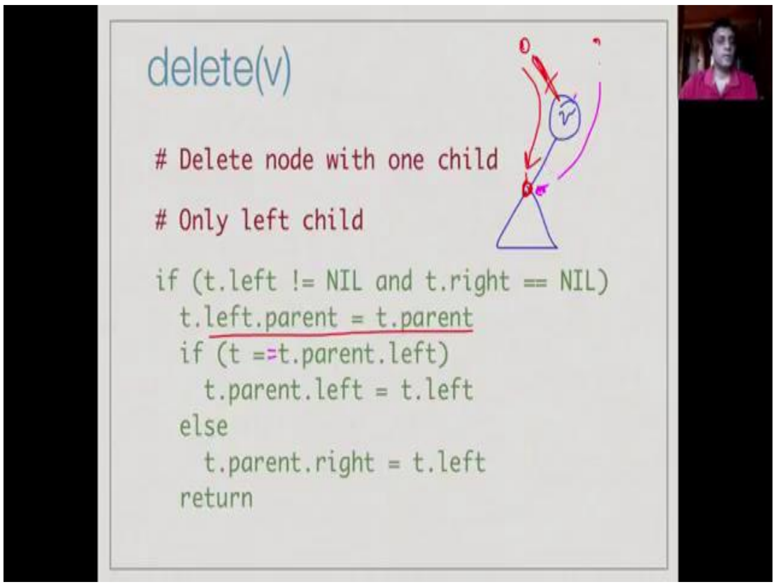
delete(v)

```
# Delete node with one child

# Only left child

if (t.left != NIL and t.right == NIL)
  t.left.parent = t.parent
  if (t ==t.parent.left)
    t.parent.left = t.left
  else
    t.parent.right = t.left
  return
```

So, having consider into the leaf case, now let us consider the case if I will delete a node with a only one child. So, the first case we will look at this, when we have trying to delete a node which has only a left sub tree. So, left is not nil, right is nil, so what we do first of all is we look at this imitate left successor. So, this has a parent somewhere about it, so if first make this left child point directly to the parent.

So, t dot left dot parent is t dot parent, then we look up and we decided whether this is the left parent or left child or a right child which parent. So, supposing the node we have is sitting to the left of it is parent, if t dot parent are left is t, then what we do is we make this point directly.

(Refer Slide Time: 30:54)

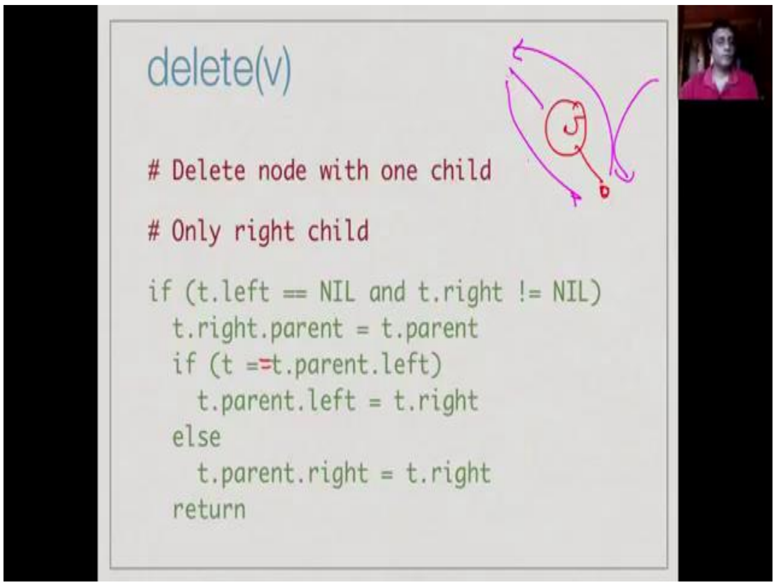


delete(v)

```
# Delete node with one child

# Only left child

if (t.left != NIL and t.right == NIL)
  t.left.parent = t.parent
  if (t ==t.parent.left)
    t.parent.left = t.left
  else
    t.parent.right = t.left
  return
```

Now, on the other hand if it was not like this, but it was the other way, so this happen to be a right child of which parent, then the right child should not point directly, the right child of the parent is now my left child. So, this the way to splices out a remove this thing and promote the child they only child are we do a symmetrical thing if it is only the right child.

(Refer Slide Time: 31:14)

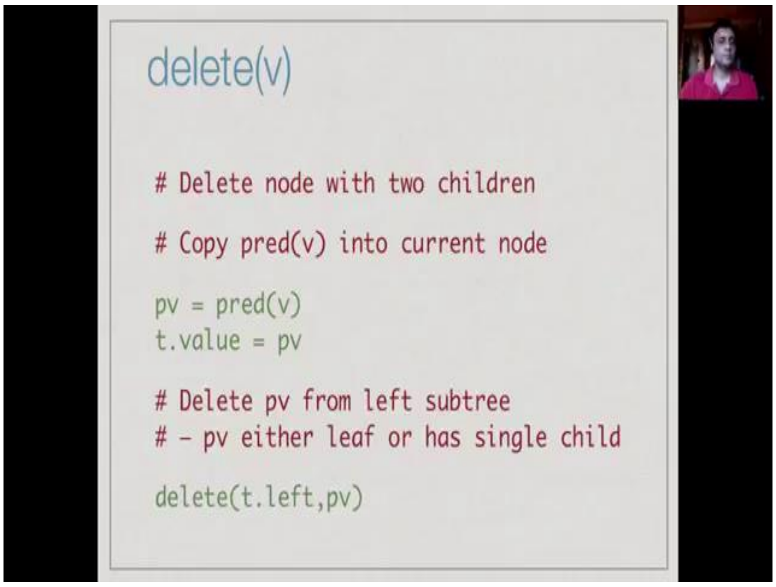


delete(v)

```
# Delete node with one child

# Only right child

if (t.left == NIL and t.right != NIL)
  t.right.parent = t.parent
  if (t ==t.parent.left)
    t.parent.left = t.right
  else
    t.parent.right = t.right
  return
```

So, we first take the right child and then we restart it is parent to be your own parent and there after that depending on the case, we either make at this point like this, at this point like this. So, either t dot parent of left is t dot right, t dot parent of right is t dot left. So, if I have only one child we just remove the node if effectively from the tree.

(Refer Slide Time: 31:47)

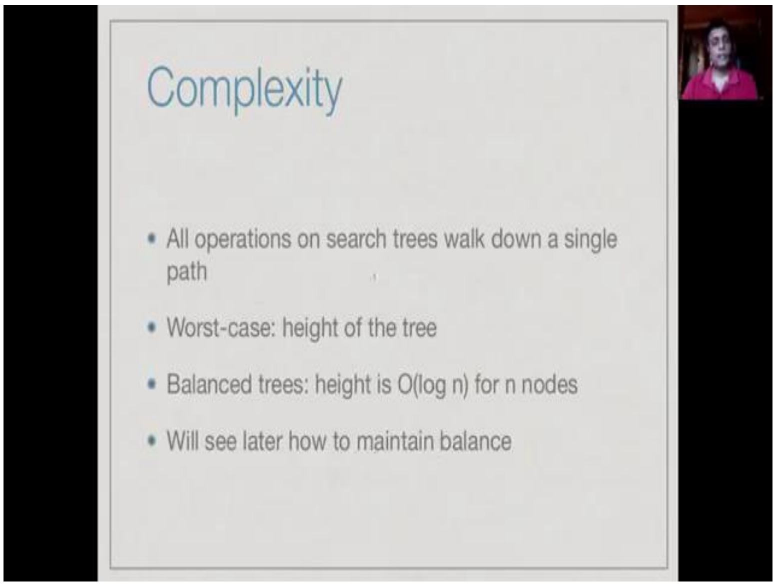


delete(v)

```
# Delete node with two children

# Copy pred(v) into current node

pv = pred(v)
t.value = pv

# Delete pv from left subtree
# - pv either leaf or has single child

delete(t.left,pv)
```

And finally, if we have a node with two children, what we do is if we first compute the predecessor at v and then we said the current nodes value to be the predecessor value and now we know that this value is duplicate. So, we go the left child and deleted are remember that when we deleted here, the left child we have deleting what we know is the maximum value. So, therefore it will have either no children it will be a leaf or it will have a single child. So, it will not come back to this case it will just stop at the earlier two cases and it will get successfully deleted. So, there is no problem it is just calling delete again.

(Refer Slide Time: 32:22)

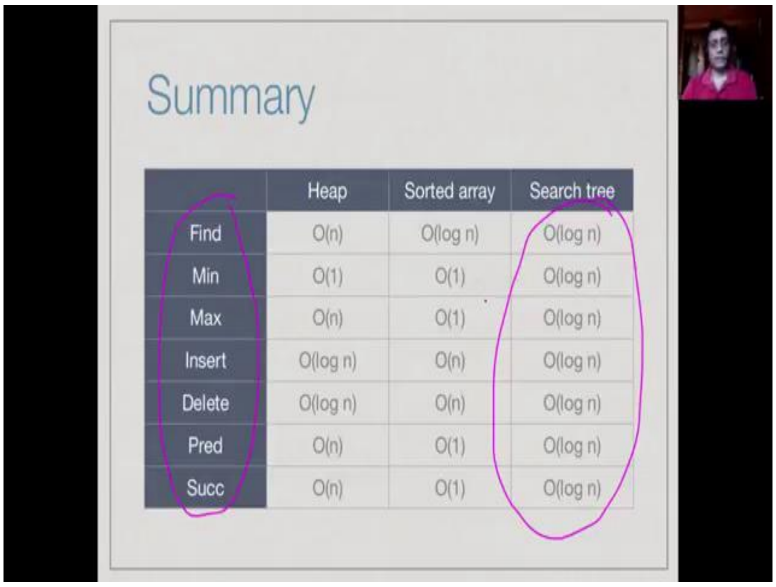


Complexity

- All operations on search trees walk down a single path
- Worst-case: height of the tree
- Balanced trees: height is O(log n) for n nodes
- Will see later how to maintain balance

So, all these operations that we have to describe now walk down a single path. So, therefore, the worst case complex to any one of these operations is the height of the current given tree. And if we have maintain some kind of a balance tree like a heat we saw, then the height is logarithmic in this sides. So, we have n nodes the height is order log n.

(Refer Slide Time: 32:48)



Summary

	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, you will see in the next lecture how to maintain the balance but, assuming that we maintain in the balance we have succeeded in what we have achieve, what it wanted to achieve which is we wanted all these 7 operations to be simultaneously efficient and there all now log n time. Because, each of them can be achieved in one traversal up or down a path in the tree.