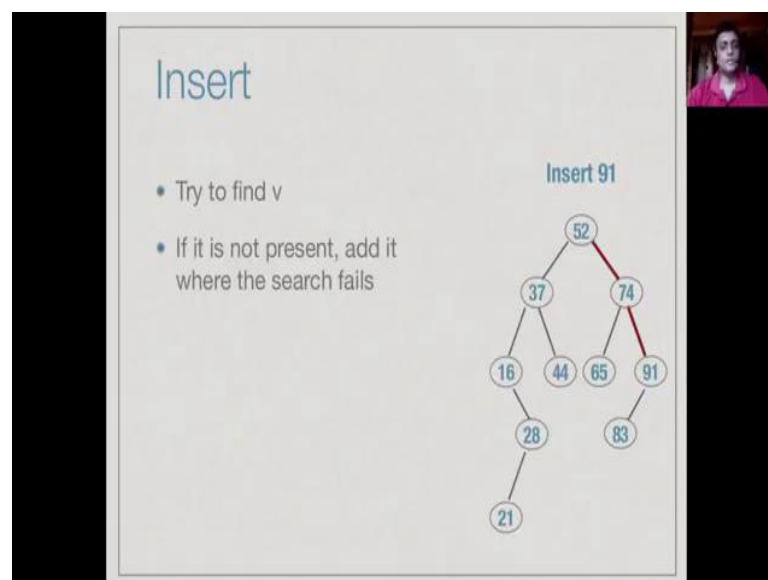


So, now let us see how to insert a value, inserting a value in a search tree fairly straight forward, there is only one place because of remember that the tree is listed in order will give us a sorted list. So, now after adding this value it must second give us a sorted list, so it is like saying that they was only one place to insert a value in a sorted list. So, is like insertion in a sorted 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 instance if you want to insert 21, then we will walk down the tree and we will look for the place where we should find 21. So, it is smaller than 52 we go left, smaller than 37 we go left, bigger than 16 we go right and then we find that at 28 there is no value of 21, because there is no left, so this since it should be to the left of 28, we add it there.

(Refer Slide Time: 22:45)

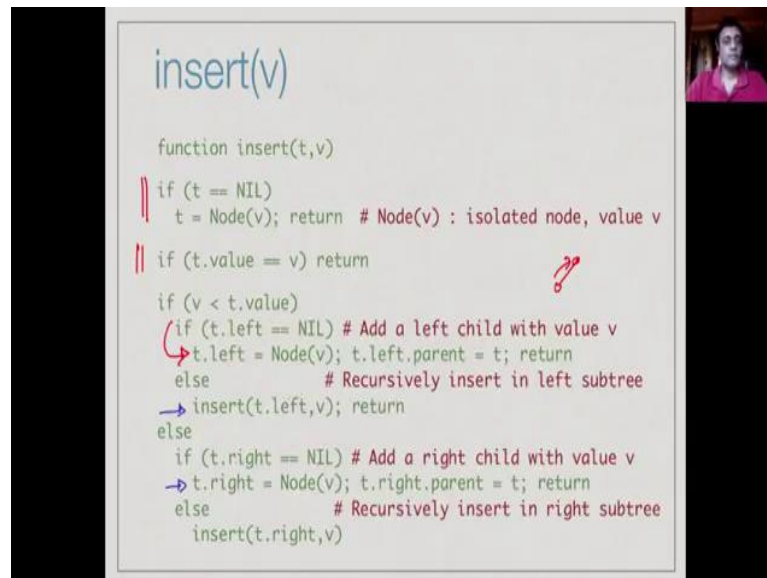


Now, for instance if we want to insert 65, then we will go right from 52 and now it is 74 we will look left, where there is nothing to the left. So, we will insert ((Refer Time: 22:59)), so it can happen at any position in the tree does not have to be at a leaf node like a 28, 28 was already a leaf and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



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 $\text{pred}(v)$ or $\text{succ}(v)$
 - Delete $\text{pred}(v)$ / $\text{succ}(v)$
 - Either leaf or only one child

Delete 65

```
graph TD; 52((52)) --> 37((37)); 52 --> 74((74)); 37 --> 16((16)); 37 --> 44((44)); 16 --> 28((28)); 28 --> 21((21)); 74 --> 65((65)); 74 --> 91((91)); 65 --> 83((83)); style 65 stroke:#f00
```

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 instance 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 $\text{pred}(v)$ or $\text{succ}(v)$
 - Delete $\text{pred}(v)$ / $\text{succ}(v)$
 - Either leaf or only one child

Delete 65

```
graph TD; 52((52)) --> 37((37)); 52 --> 74((74)); 37 --> 16((16)); 37 --> 44((44)); 16 --> 28((28)); 28 --> 21((21)); 74 --> 65((65)); 74 --> 91((91)); 65 --> 83((83)); style 65 stroke:#f00
```

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 $\text{pred}(v)$ or $\text{succ}(v)$
 - Delete $\text{pred}(v)$ / $\text{succ}(v)$
 - Either leaf or only one child

Delete 74

```
graph TD; 52((52)) --- 37((37)); 52 --- 74((74)); 37 --- 16((16)); 37 --- 44((44)); 16 --- 28((28)); 28 --- 21((21)); 74 --- 91((91)); 91 --- 83((83));
```

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 $\text{pred}(v)$ or $\text{succ}(v)$
 - Delete $\text{pred}(v)$ / $\text{succ}(v)$
 - Either leaf or only one child

Delete 37

```
graph TD; 52((52)) --- 37((37)); 52 --- 91((91)); 37 --- 16((16)); 37 --- 44((44)); 16 --- 28((28)); 28 --- 21((21)); 91 --- 83((83));
```

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 $\text{pred}(v)$ or $\text{succ}(v)$
 - Delete $\text{pred}(v)$ / $\text{succ}(v)$
 - Either leaf or only one child

Delete 37

```
graph TD; 52((52)) --- 37((37)); 52 --- 91((91)); 37 --- 16((16)); 37 --- 44((44)); 16 --- 28((28)); 28 --- 21((21)); 91 --- 83((83)); style 37 stroke:#f00,stroke-width:2px;
```

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 $\text{pred}(v)$ or $\text{succ}(v)$
 - Delete $\text{pred}(v)$ / $\text{succ}(v)$
 - Either leaf or only one child

Delete 37

```
graph TD; 52((52)) --- 37((37)); 52 --- 91((91)); 37 --- 16((16)); 37 --- 44((44)); 16 --- 28((28)); 28 --- 21((21)); 91 --- 83((83)); style 37 stroke:#f00,stroke-width:2px;
```

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 $\text{pred}(v)$ or $\text{succ}(v)$
 - Delete $\text{pred}(v)$ / $\text{succ}(v)$
 - Either leaf or only one child

Delete 37

```
graph TD; 52((52)) --> 28((28)); 52 --> 91((91)); 28 --> 16((16)); 28 --> 44((44)); 16 --> 21((21)); 16 --> 28((28)); 44 --> 83((83));
```

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 sorted thing. So, if we have some list of sorted 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 sorted 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 $\text{pred}(v)$ or $\text{succ}(v)$
 - Delete $\text{pred}(v)$ / $\text{succ}(v)$
 - Either leaf or only one child

Delete 37

```
graph TD; 52((52)) --> 28((28)); 52 --> 91((91)); 28 --> 16((16)); 28 --> 44((44)); 16 --> 21((21)); 91 --> 83((83));
```

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.

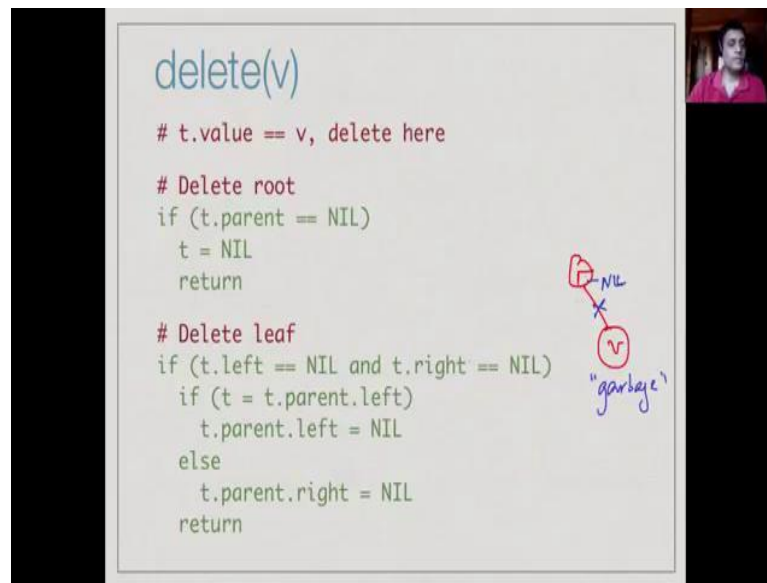
(Refer Slide Time: 27:58)

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.

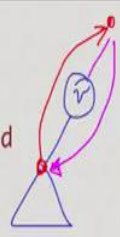
(Refer Slide Time: 29:54)

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.

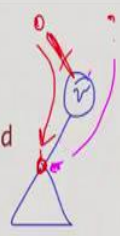
(Refer Slide Time: 30:54)

delete(v)

Delete node with one child

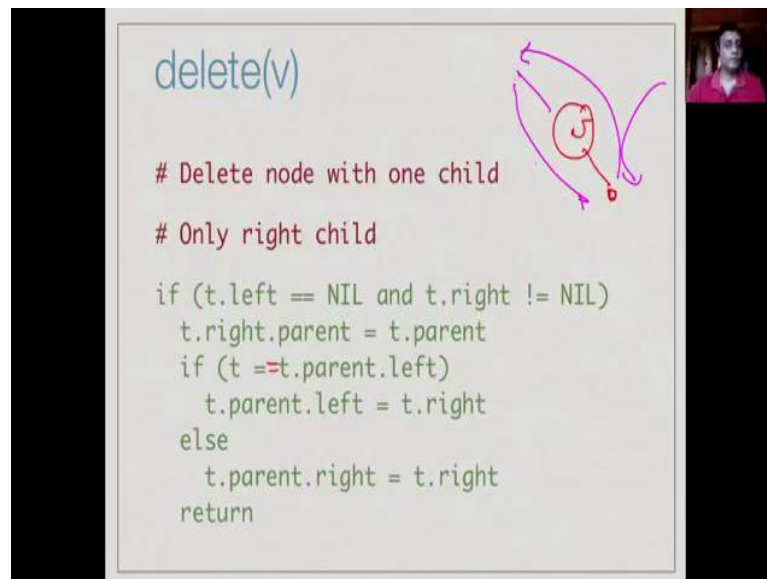
Only left child

```
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)



The slide displays a code snippet for the `delete(v)` function, specifically for the case where a node has only a right child. The code is as follows:

```
delete(v)

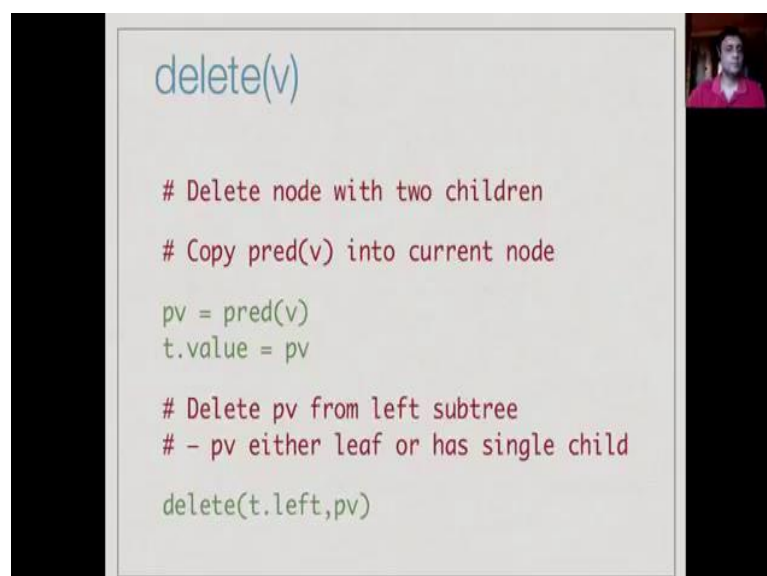
# Delete node with one child
# Only right child

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
```

Handwritten in pink on the right side of the slide is a diagram of a node labeled 't' with a right child labeled 'p'. Arrows indicate the update of the parent pointer for the right child to be the same as the parent of 't'.

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.parent.left` is `t.right`, `t.parent.right` is `t.left`. So, if I have only one child we just remove the node if effectively from the tree.

(Refer Slide Time: 31:47)



The slide displays a code snippet for the `delete(v)` function, specifically for the case where a node has two children. The code is as follows:

```
delete(v)

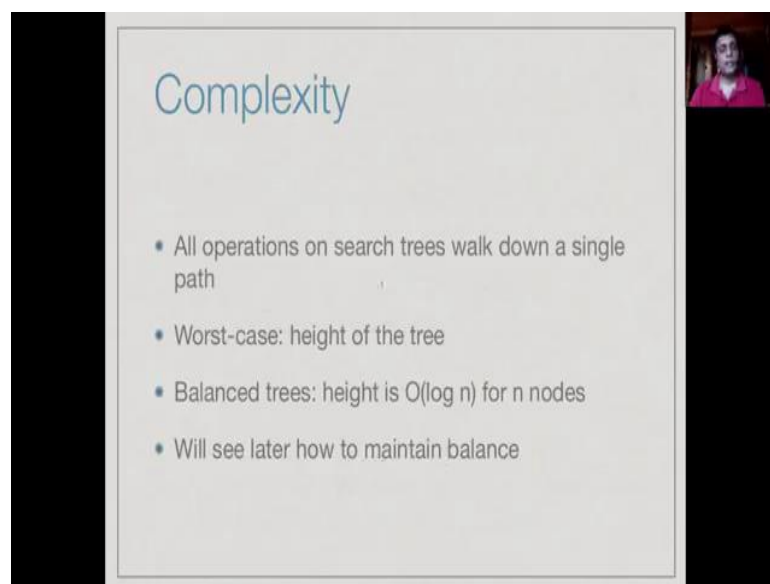
# Delete node with two children
# Copy pred(v) into current node

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)

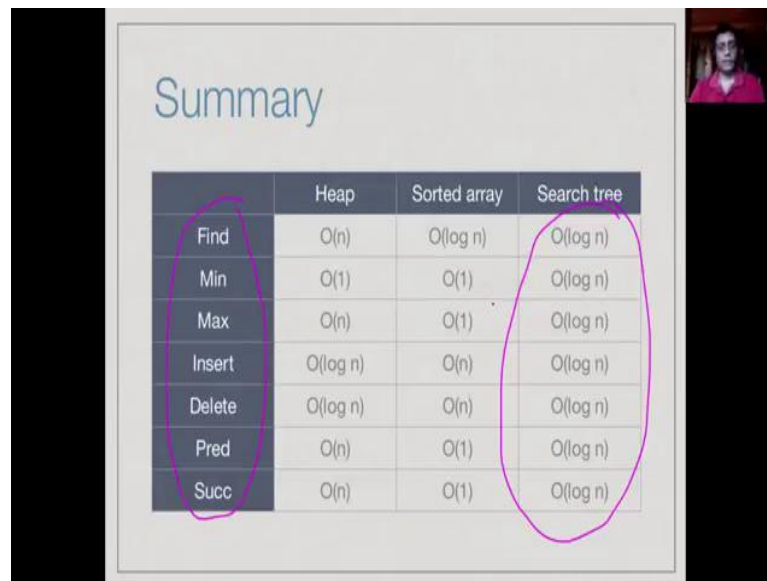


Complexity

- 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 achieved, 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.