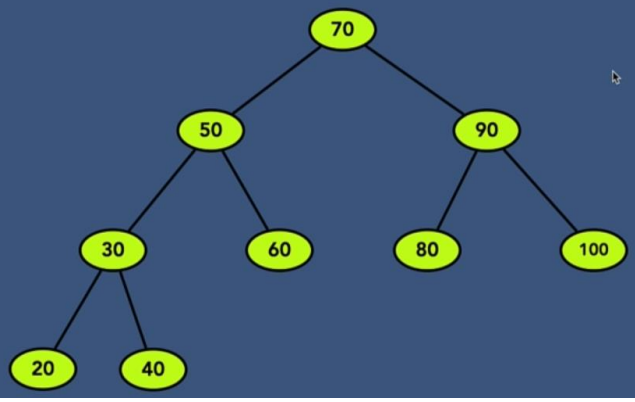
What is an AVL Tree?

An AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.



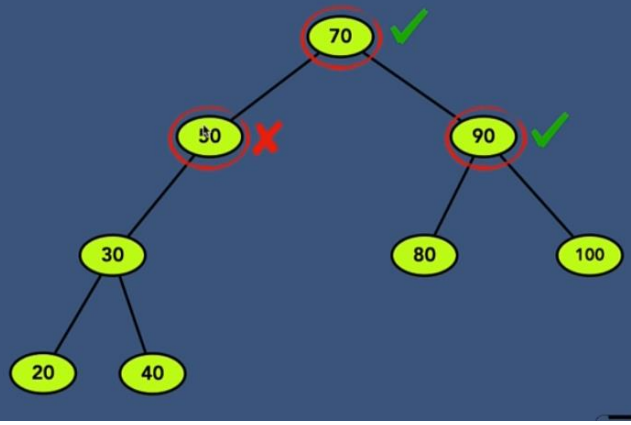
What is an AVL Tree?

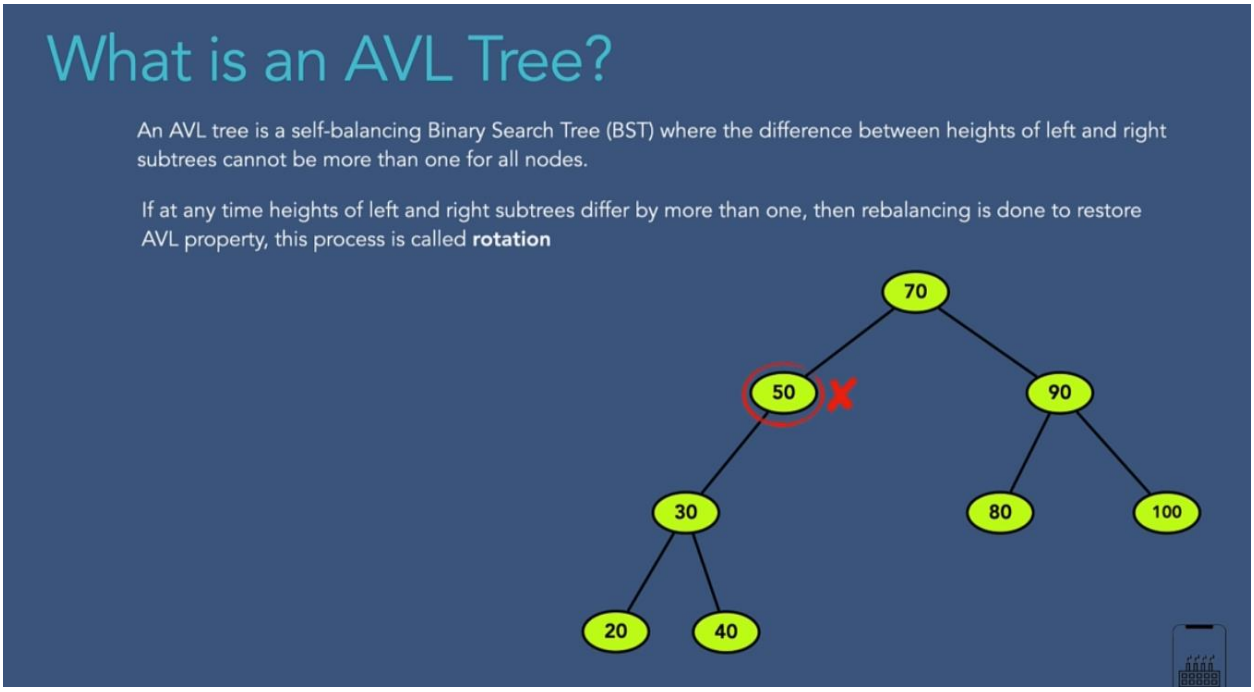
An AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

Height of leftSubtree = 3
Height of rightSubtree = 2
difference = 1

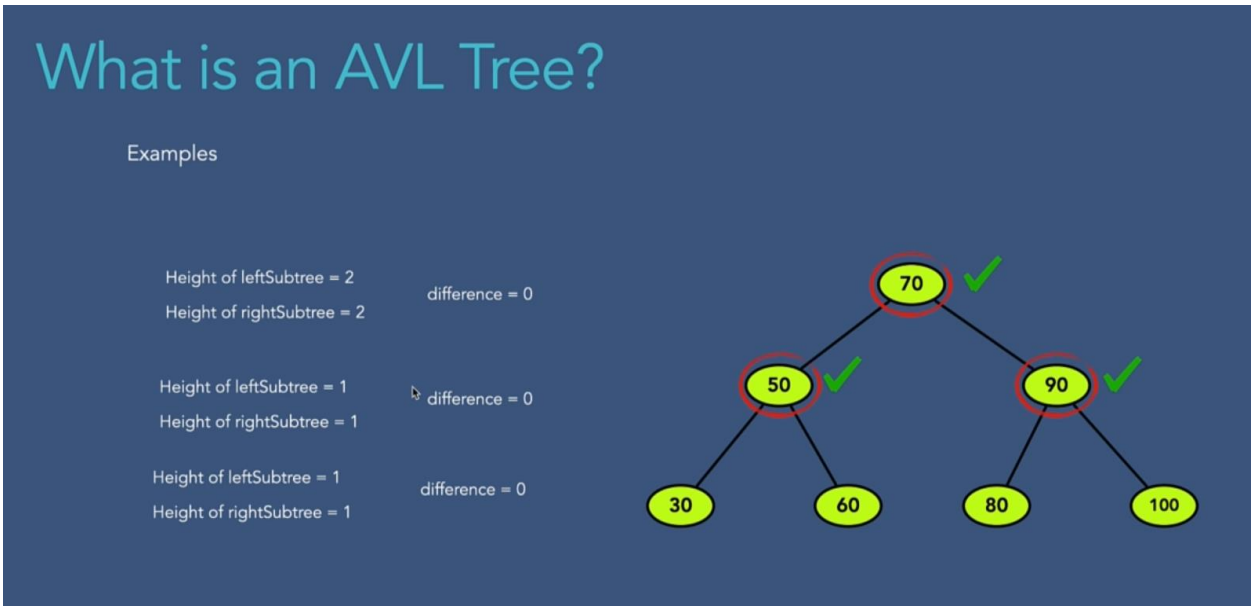
Height of leftSubtree = 1
Height of rightSubtree = 1
difference = 0

Height of leftSubtree = 2
Height of rightSubtree = 0
difference = 2





If at any time heights of left and right subtrees differ by more than one, then rebalancing is done to restore AVL property, this process is called rotation.



What is an AVL Tree?

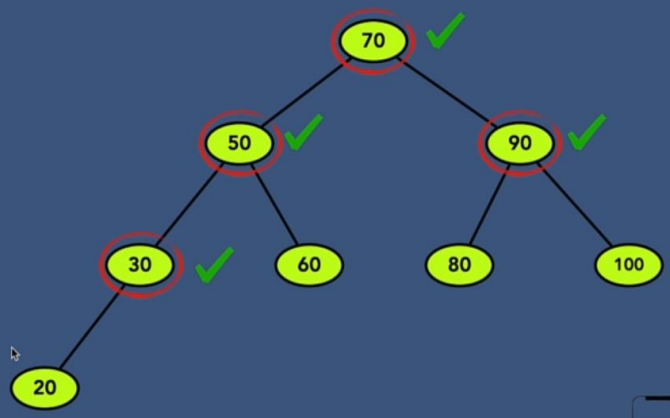
Examples

Height of leftSubtree = 3
Height of rightSubtree = 2
difference = 1

Height of leftSubtree = 2
Height of rightSubtree = 1
difference = 1

Height of leftSubtree = 1
Height of rightSubtree = 1
difference = 0

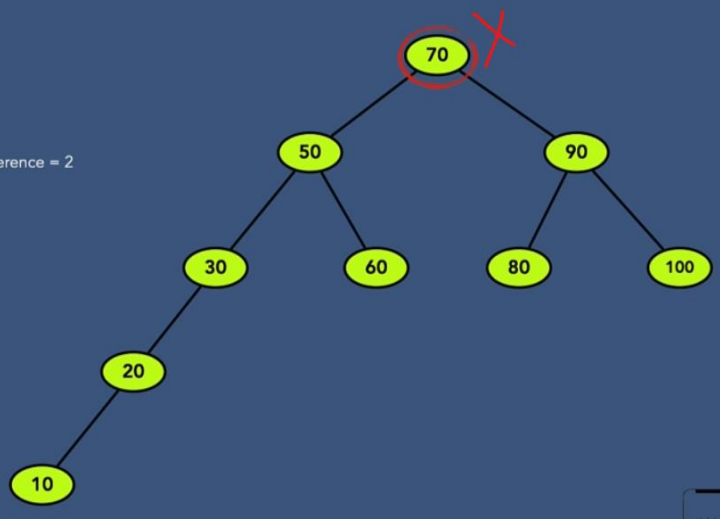
Height of leftSubtree = 1
Height of rightSubtree = 0
difference = 1



What is an AVL Tree?

Examples

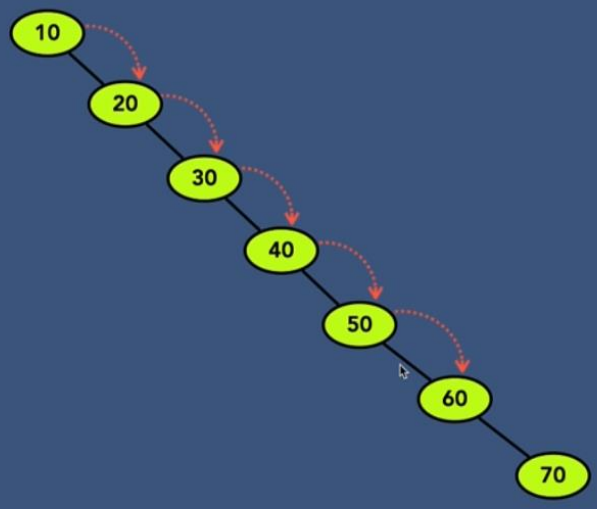
Height of leftSubtree = 4
Height of rightSubtree = 2
difference = 2



What do we need AVL Tree?

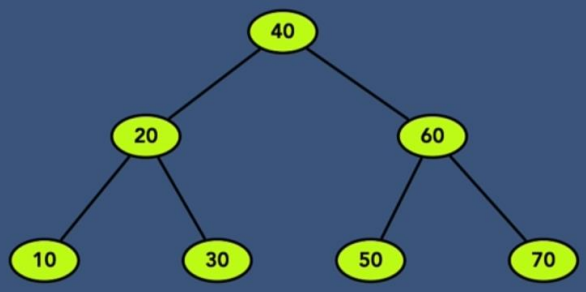
10, 20, 30, 40, 50, 60, 70

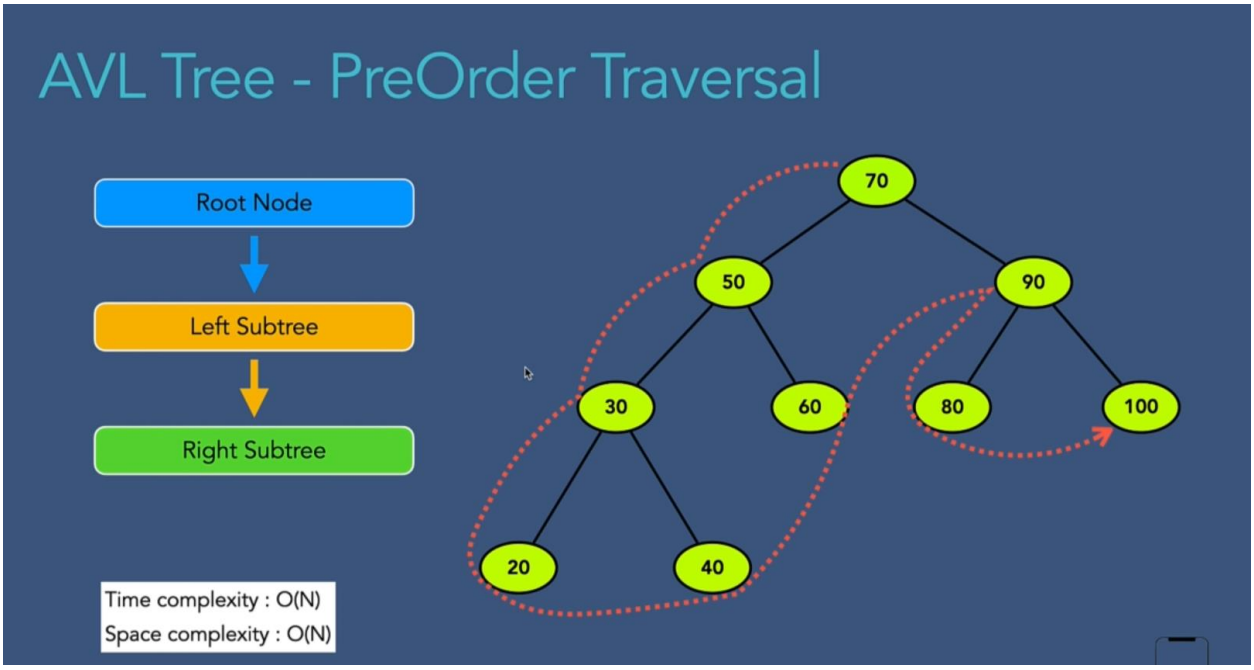
Search for 60

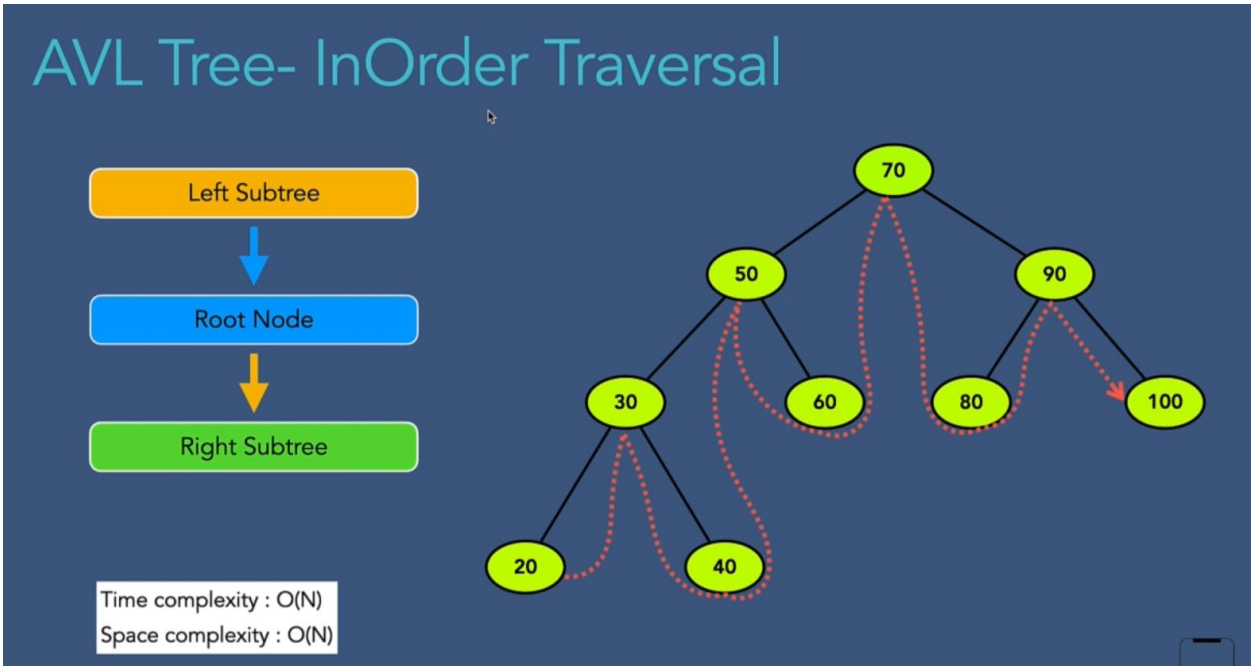


Common Operations on AVL Tree

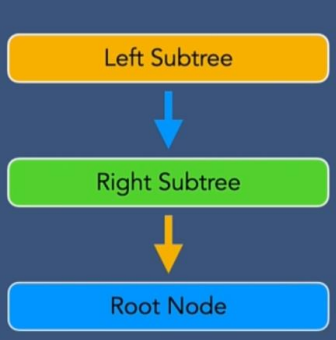
- Creation of AVL trees,
- Search for a node in AVL trees
- Traverse all nodes in AVL trees
- Insert a node in AVL trees
- Delete a node from AVL trees
- Delete the entire AVL trees

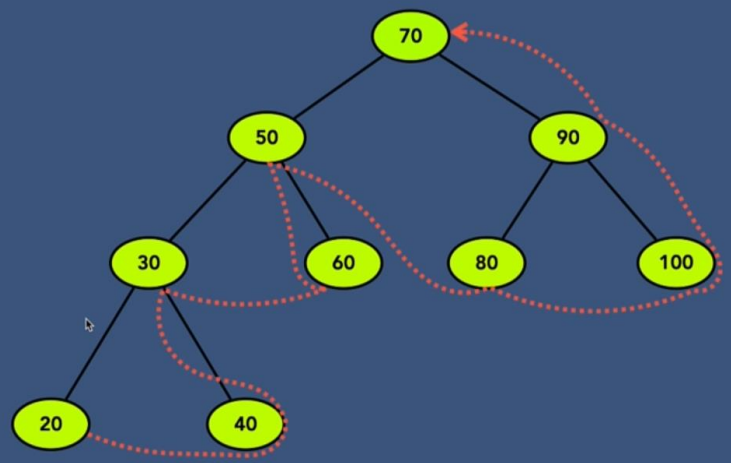



AVL Tree - PreOrder Traversal
Root Node
Left Subtree
Right Subtree
70
50
90
30
60
80
100
20
40
Time complexity : O(N)
Space complexity : O(N)


AVL Tree- InOrder Traversal
Left Subtree
Root Node
Right Subtree
70
50
90
30
60
80
100
20
40
Time complexity : O(N)
Space complexity : O(N)

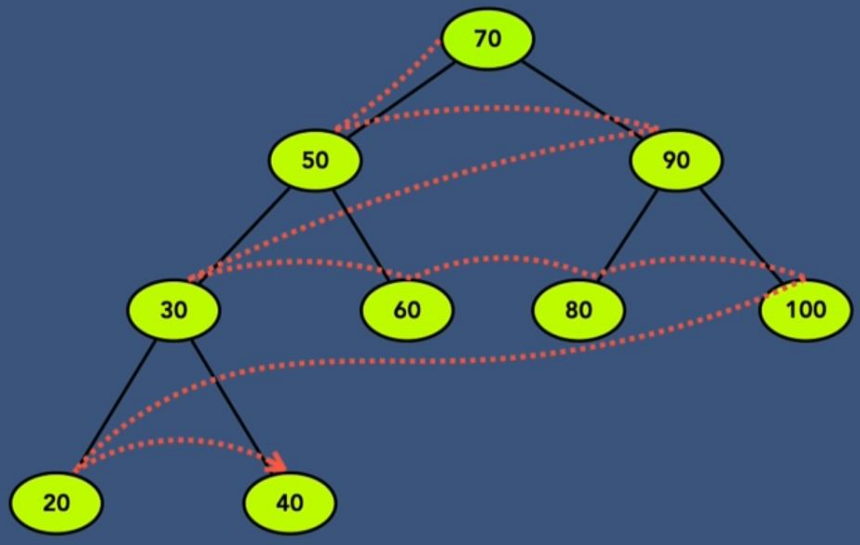
AVL Tree - Post Traversal





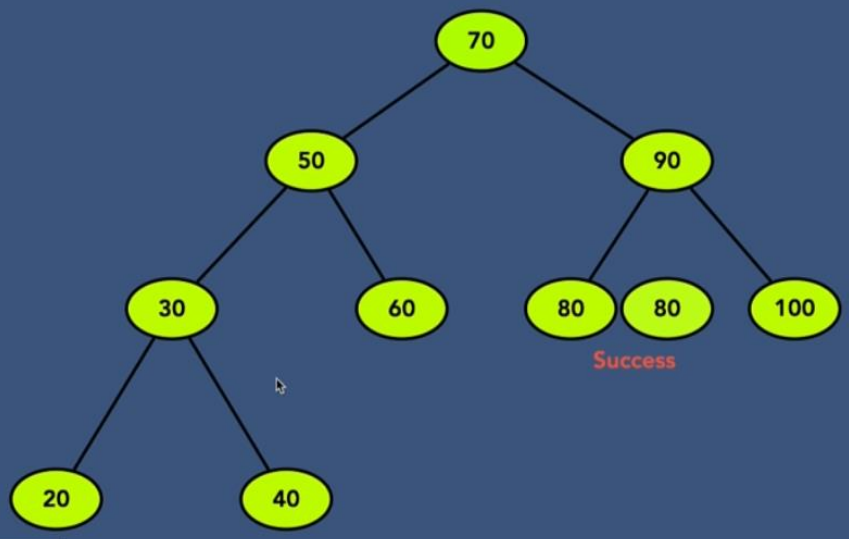
Time complexity : O(N)
Space complexity : O(N)

AVL Tree - LevelOrder Traversal



Time complexity : O(N)
Space complexity : O(N)

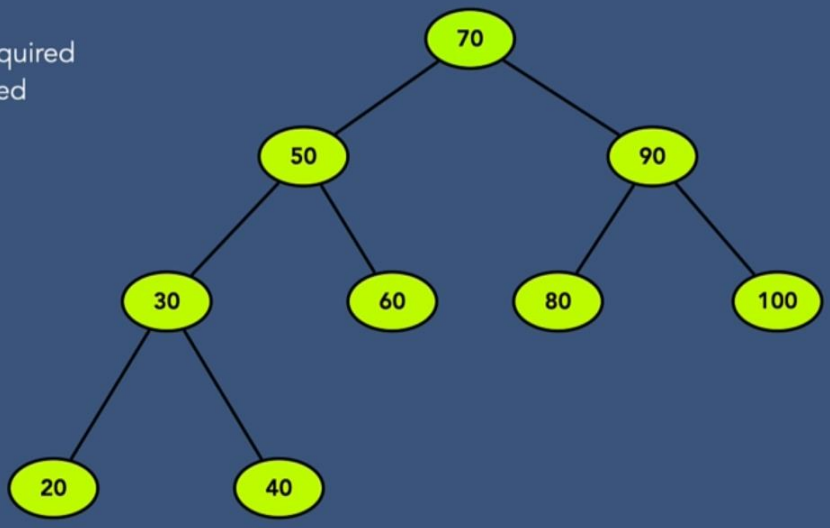
AVL Tree - Search



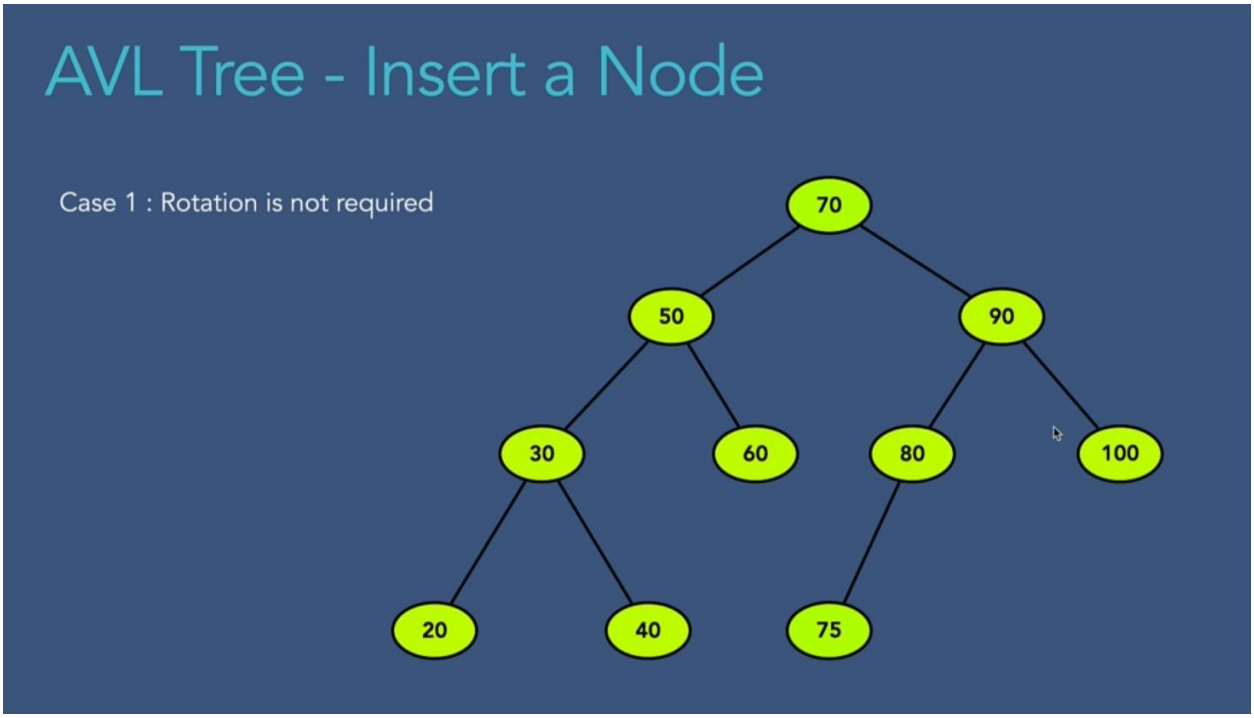
Time complexity : O(LogN)
Space complexity : O(LogN)

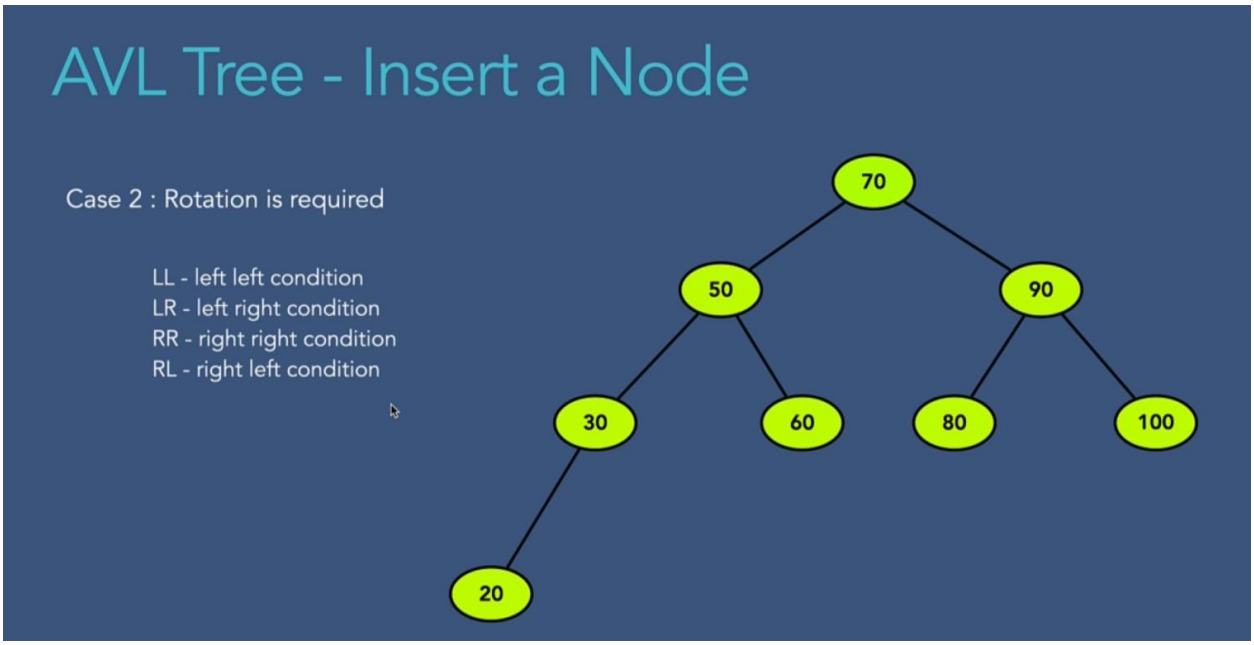
AVL Tree - Insert a Node

Case 1 : Rotation is not required
Case 2 : Rotation is required

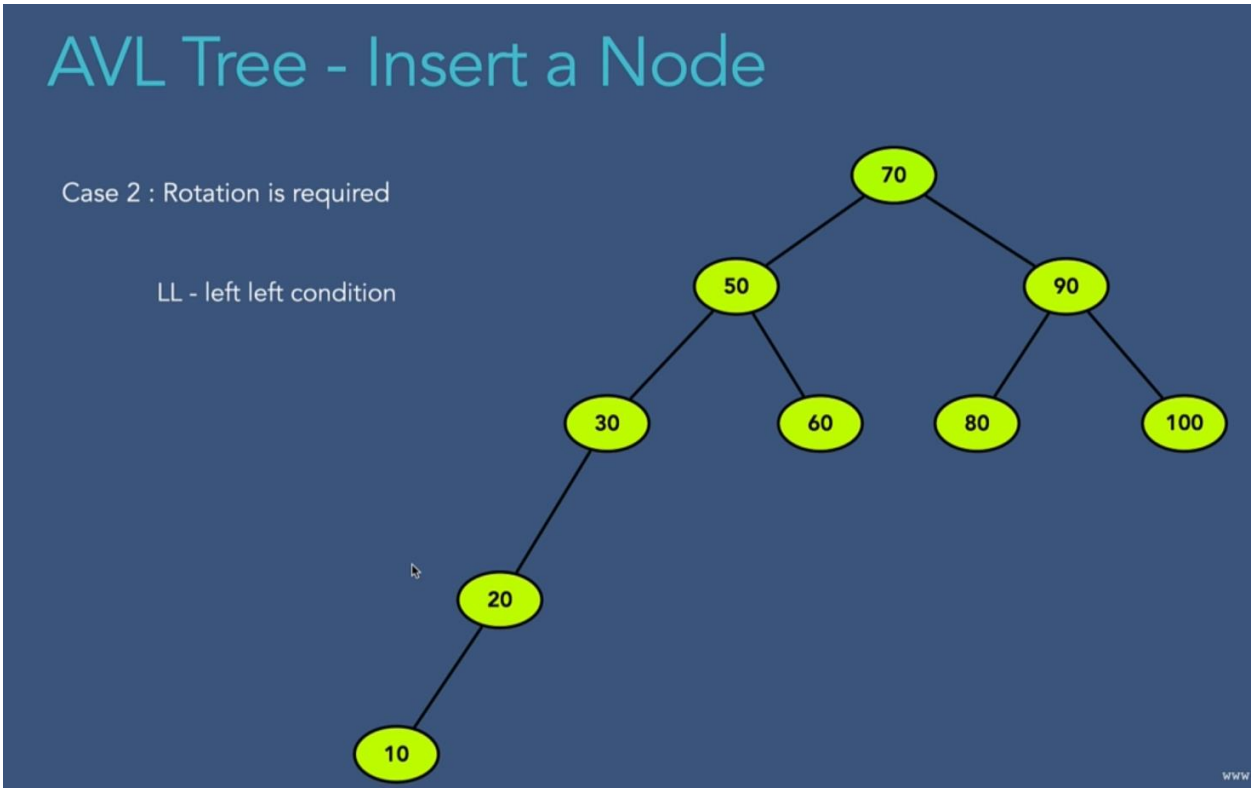


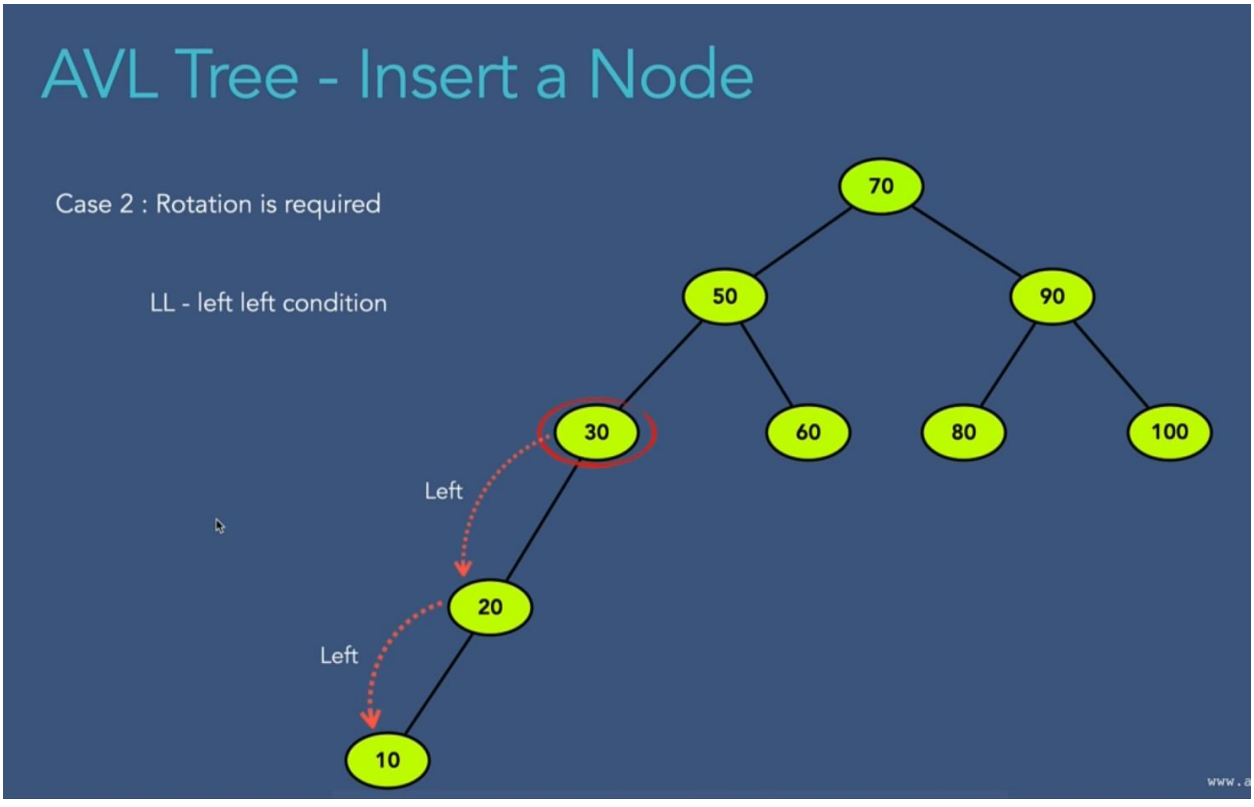
Adding Node 7 to the AVL tree → No rotation required

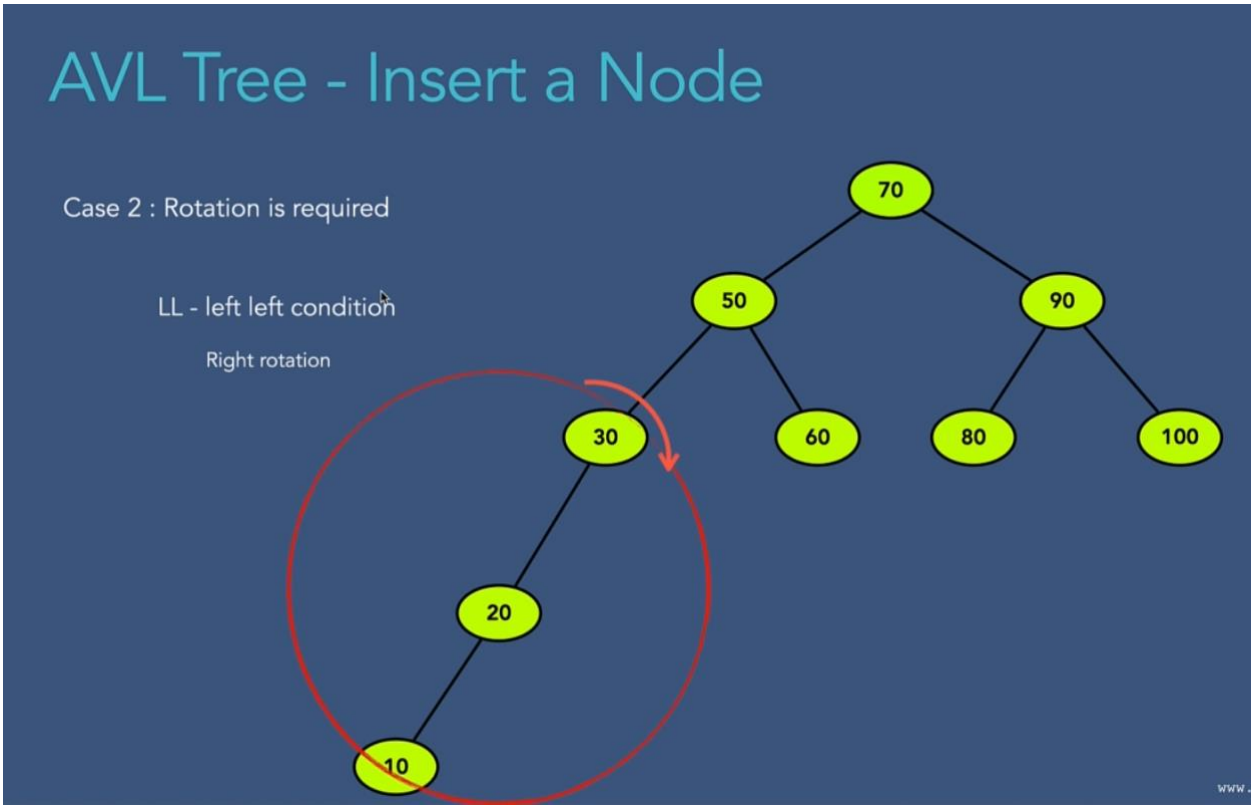




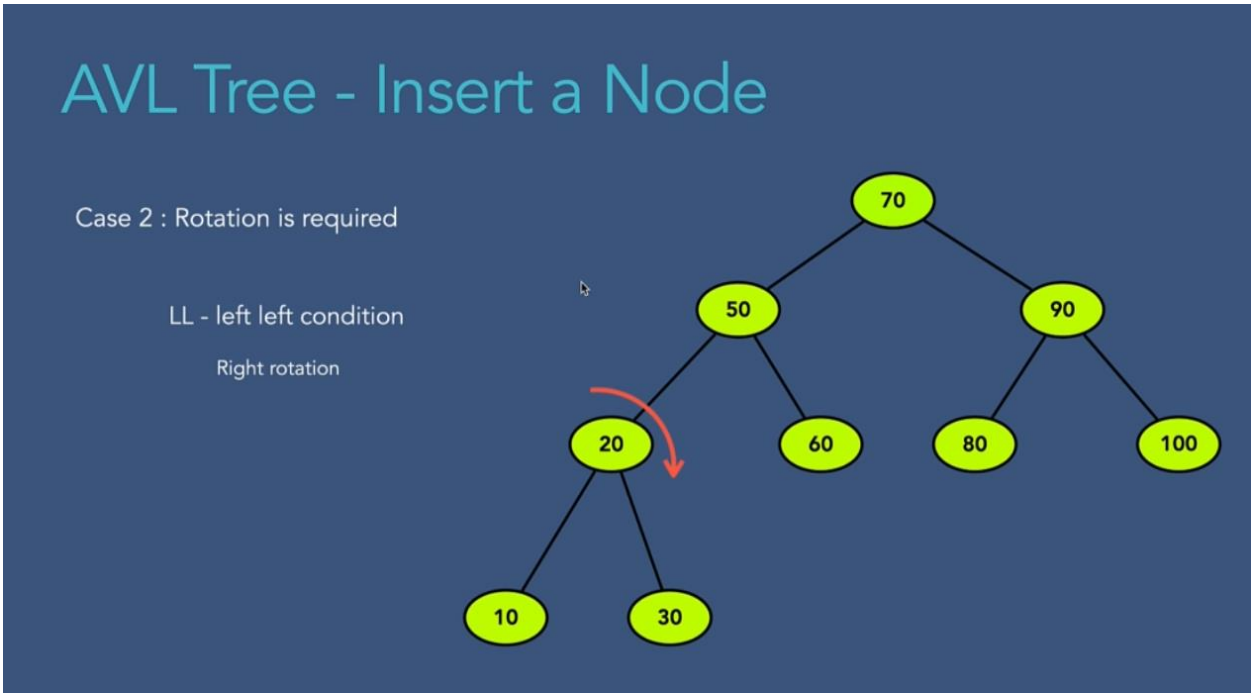
Inserting node 10 to the AVL Tree → Rotation is required



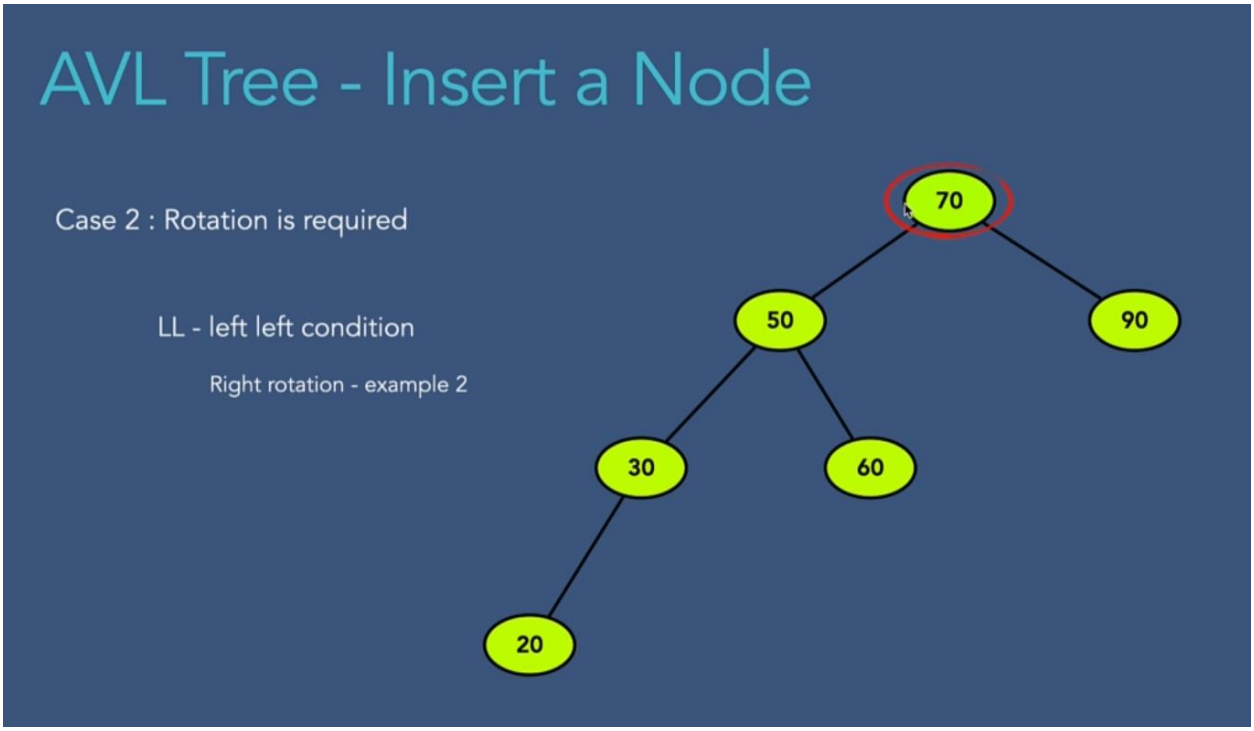




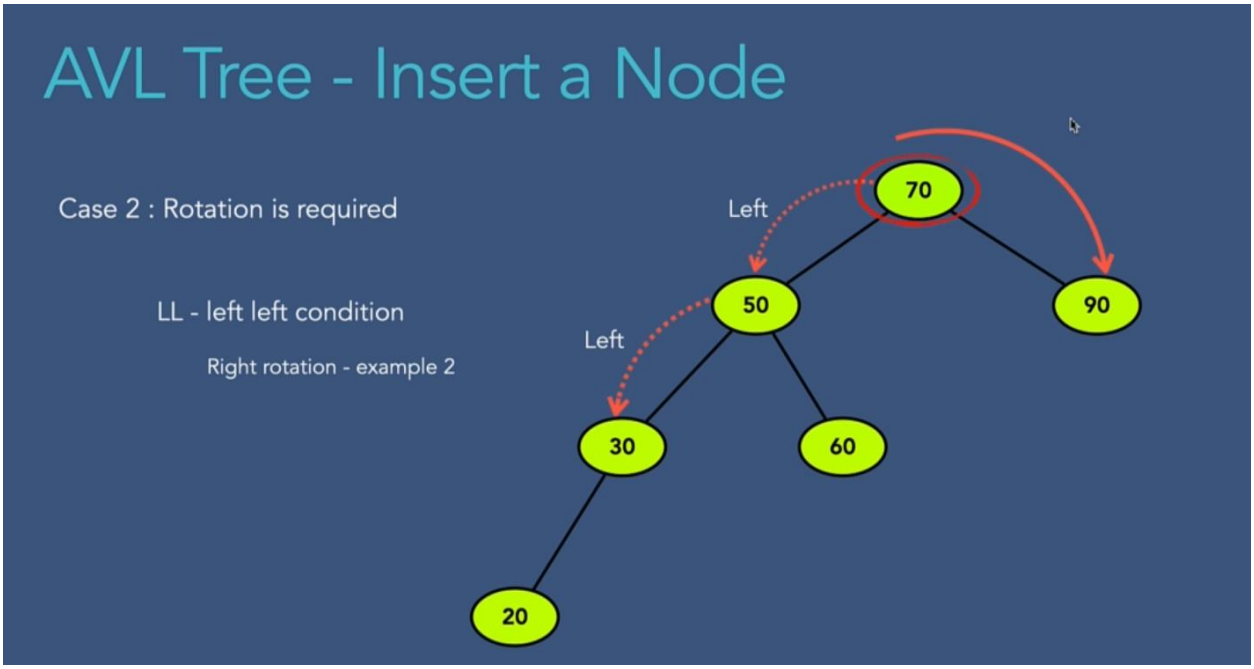
After rotation



Inserting Node 20 to the AVL Tree



We take the grand child of the node which has height as the highest

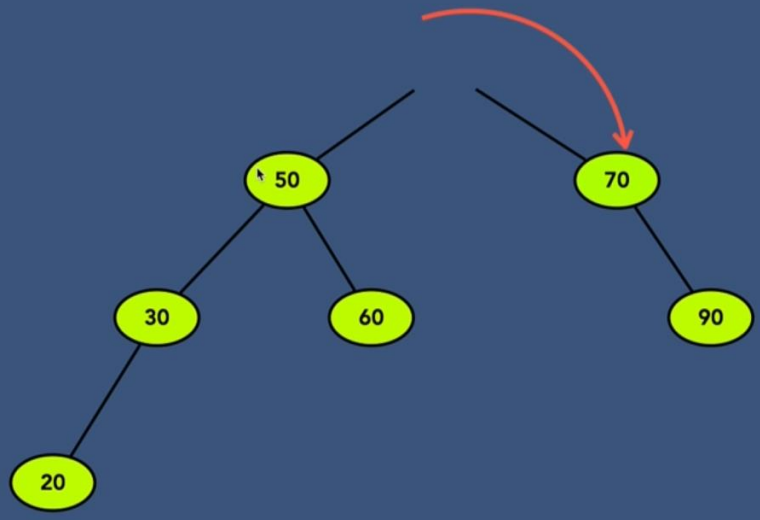


AVL Tree - Insert a Node

Case 2 : Rotation is required

LL - left left condition

Right rotation - example 2

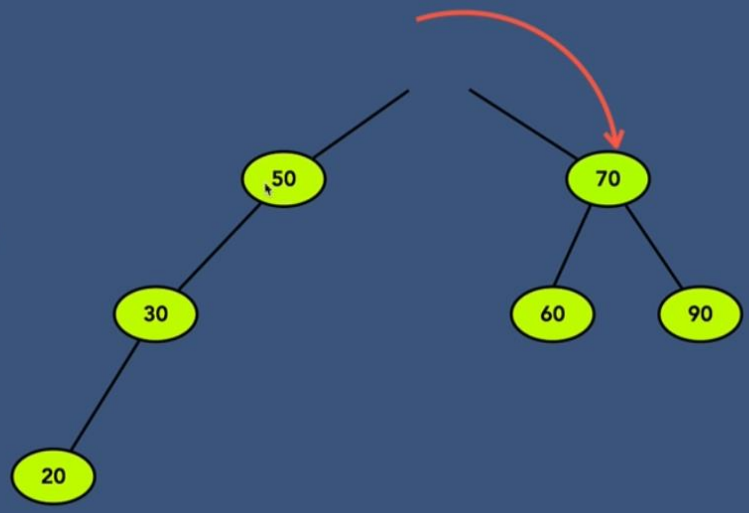


AVL Tree - Insert a Node

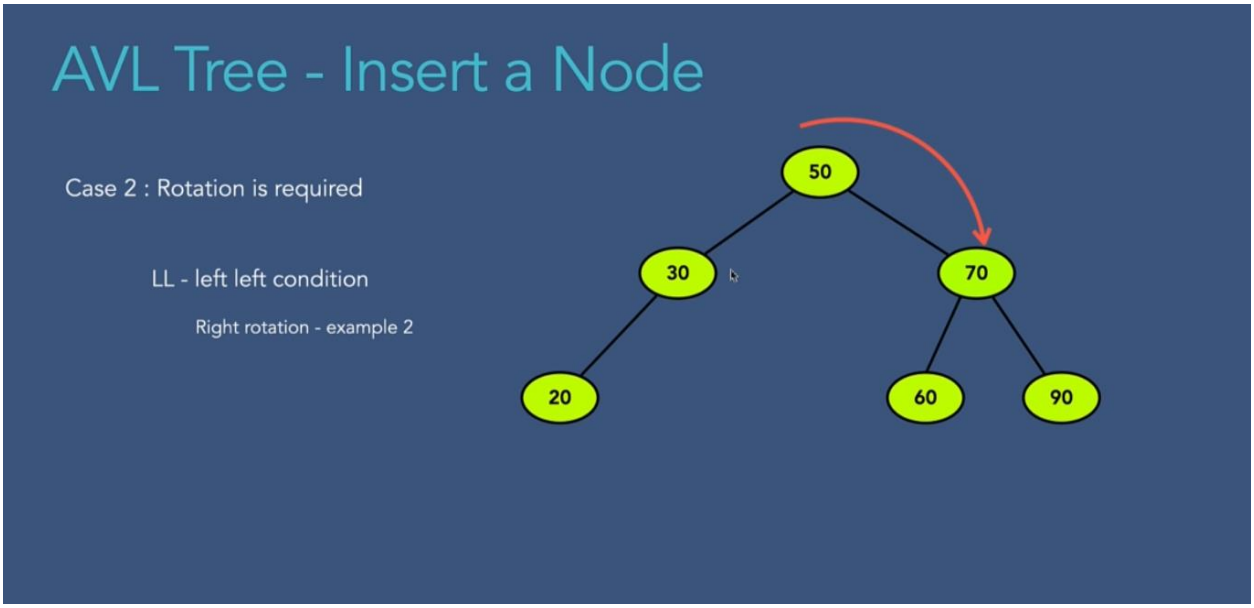
Case 2 : Rotation is required

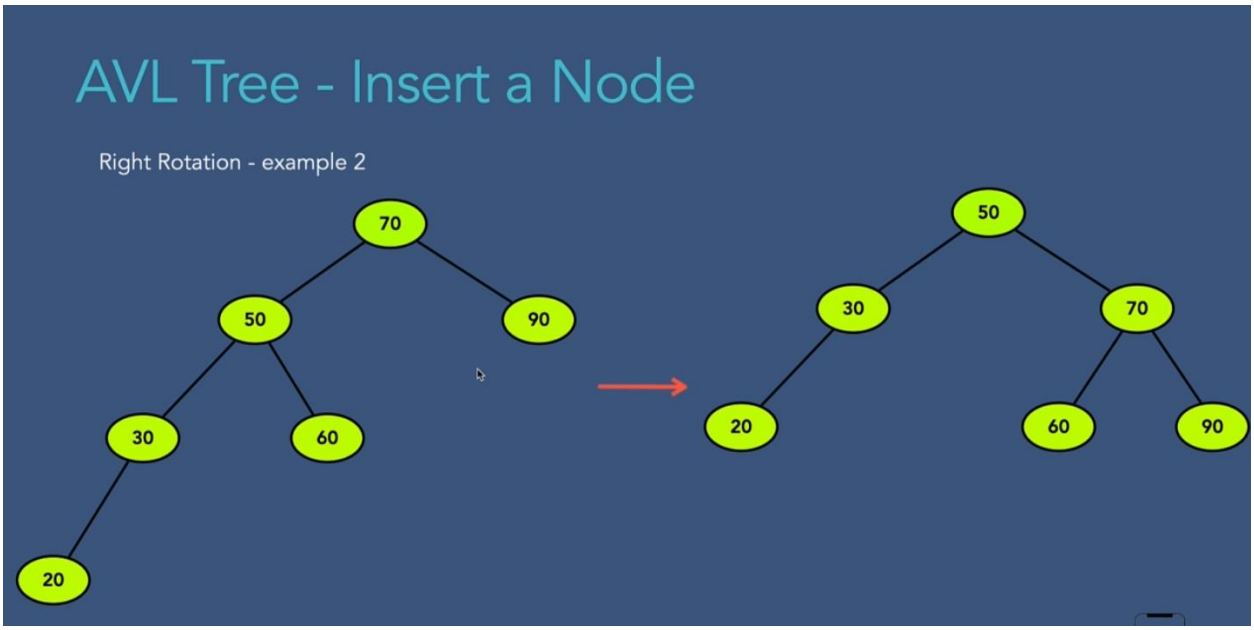
LL - left left condition

Right rotation - example 2



After rotation





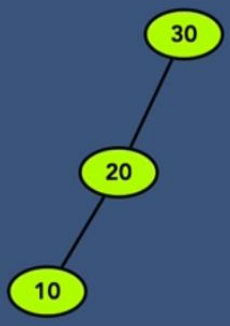
AVL Tree - Insert a Node

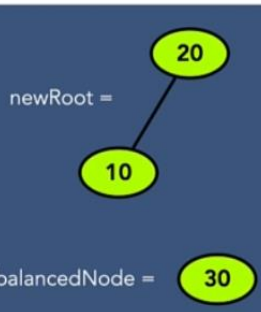
Algorithm of Left Left (LL) Condition

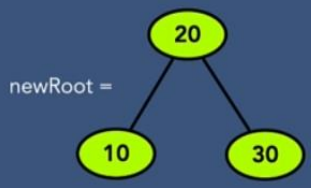
```
rotateRight(disbalancedNode) {
  newRoot = disbalancedNode.leftChild
  disbalancedNode.leftChild = disbalancedNode.leftChild.rightChild
  newRoot.rightChild = disbalancedNode
  update height of disbalancedNode and newRoot
  return newRoot
  }
```

Null

Time complexity : O(1)
Space complexity : O(1)



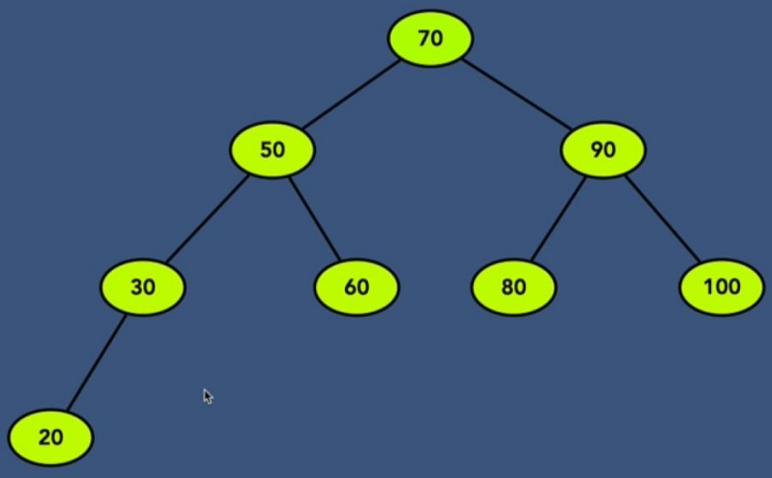




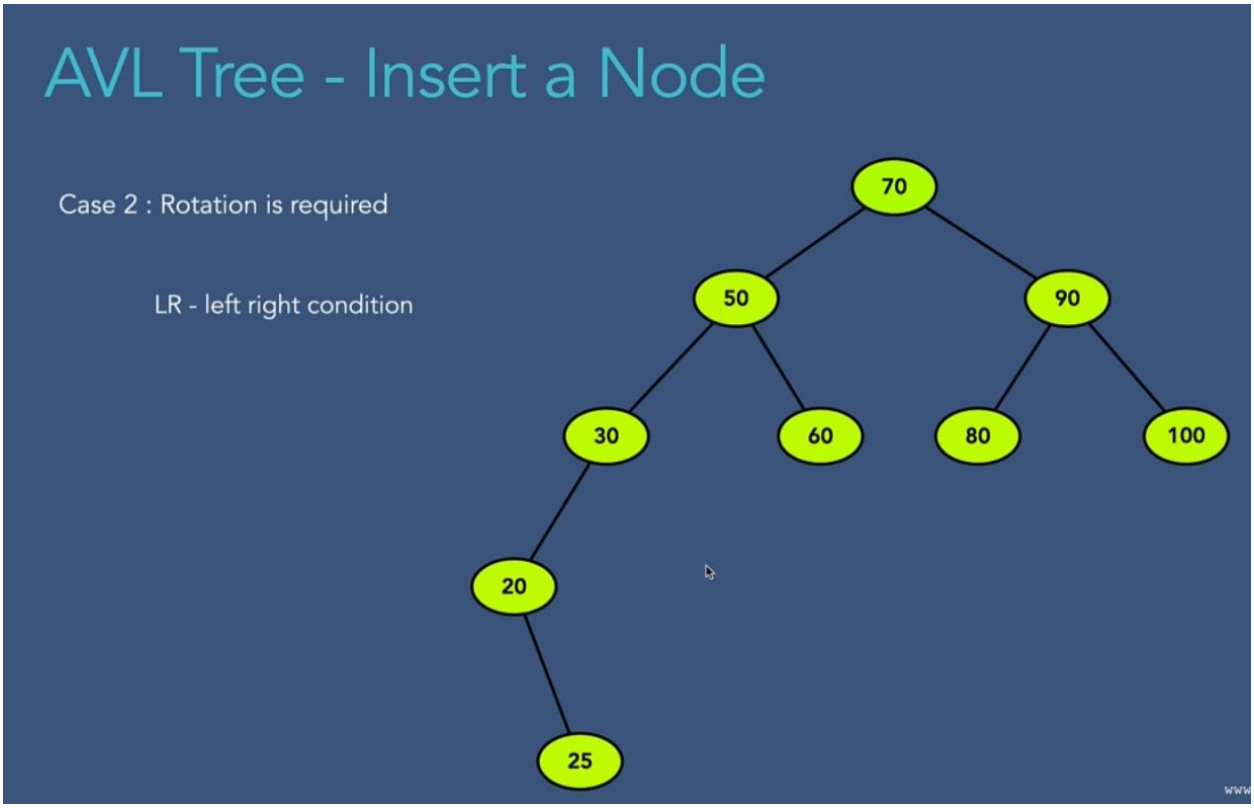
AVL Tree - Insert a Node

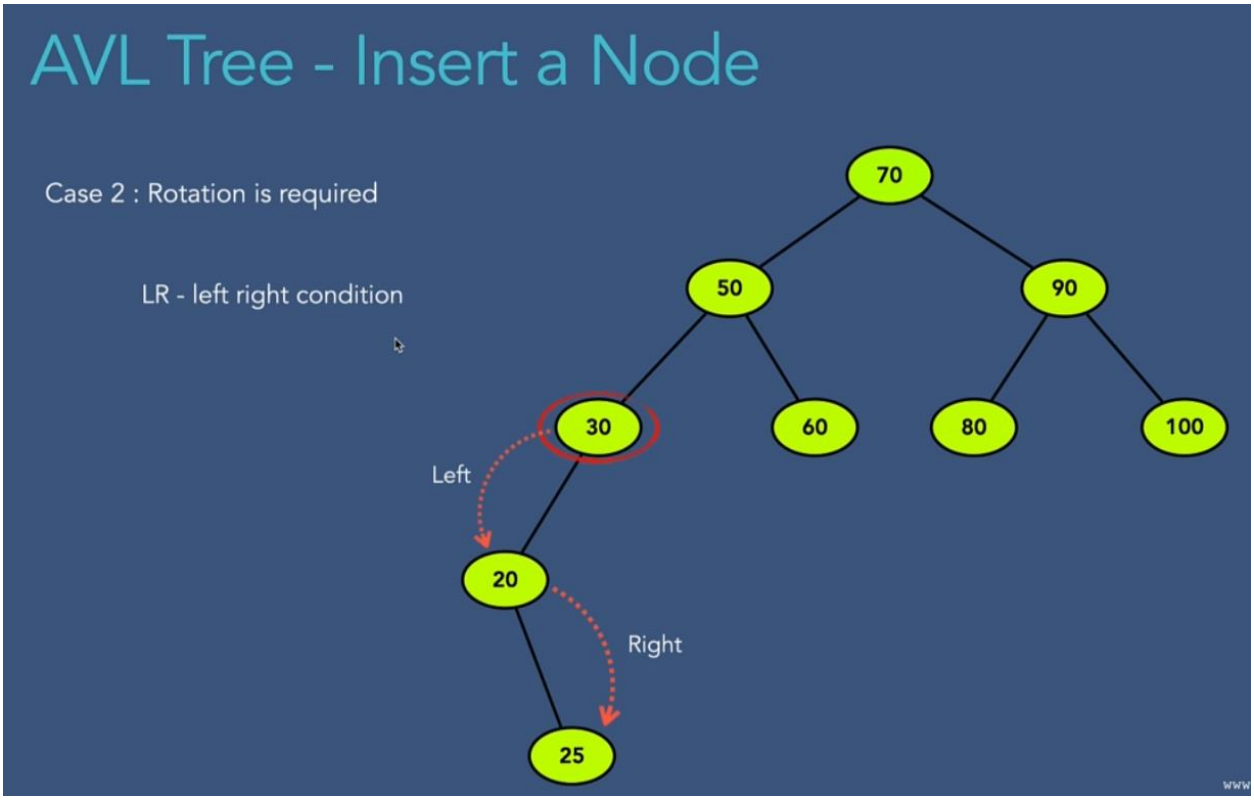
Case 2 : Rotation is required

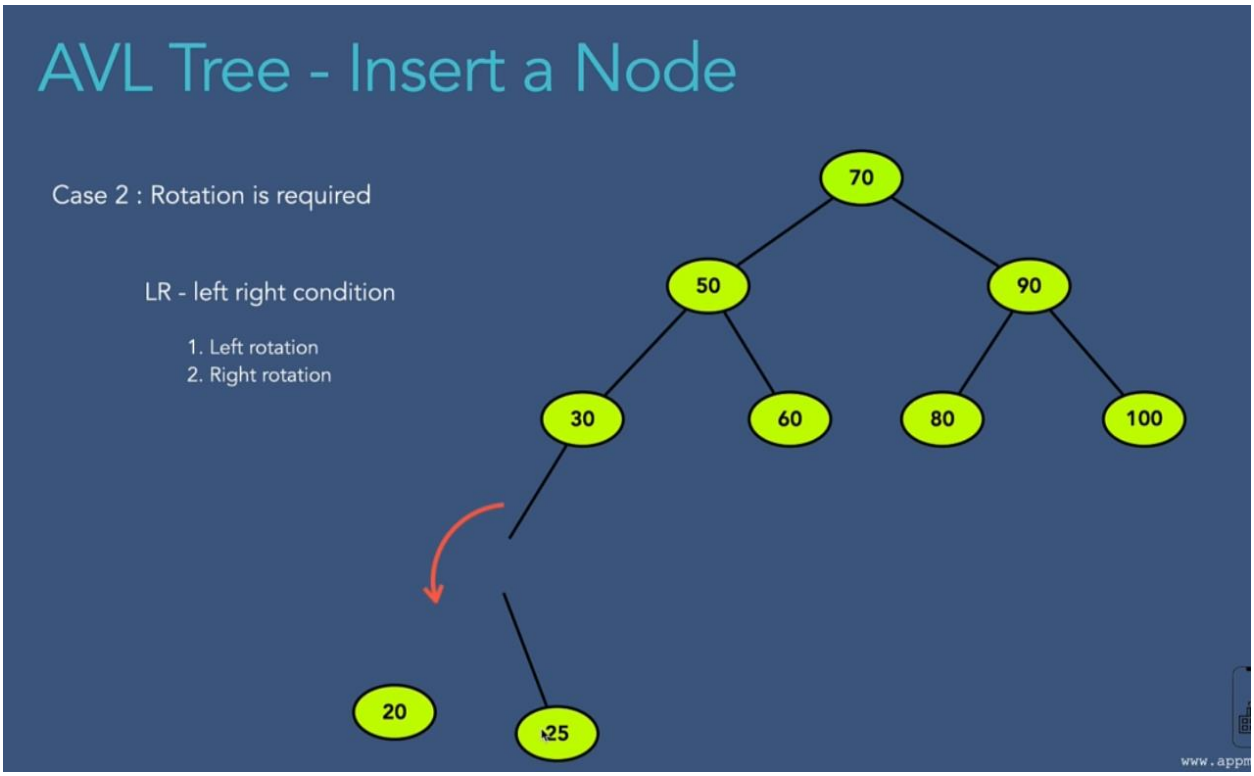
LR - left right condition



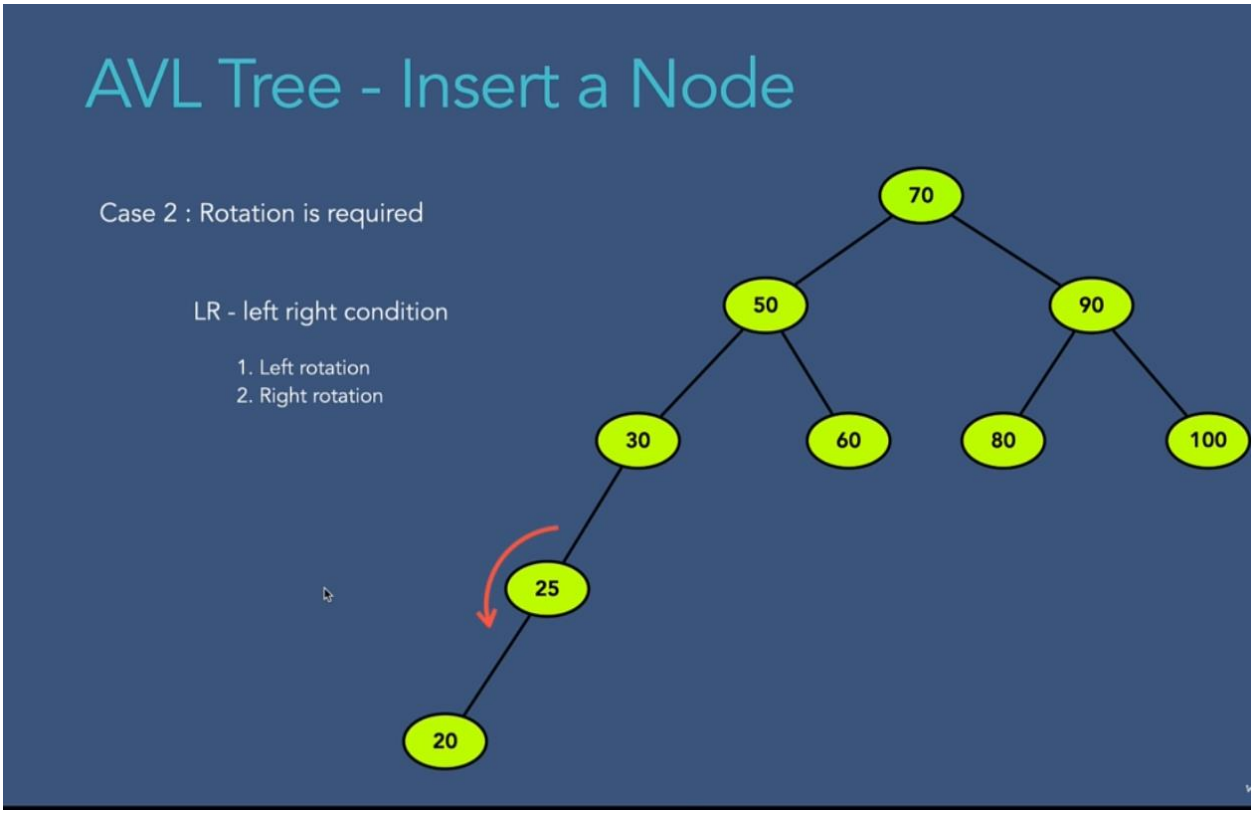
Insertion of Node 25 to AVL Tree

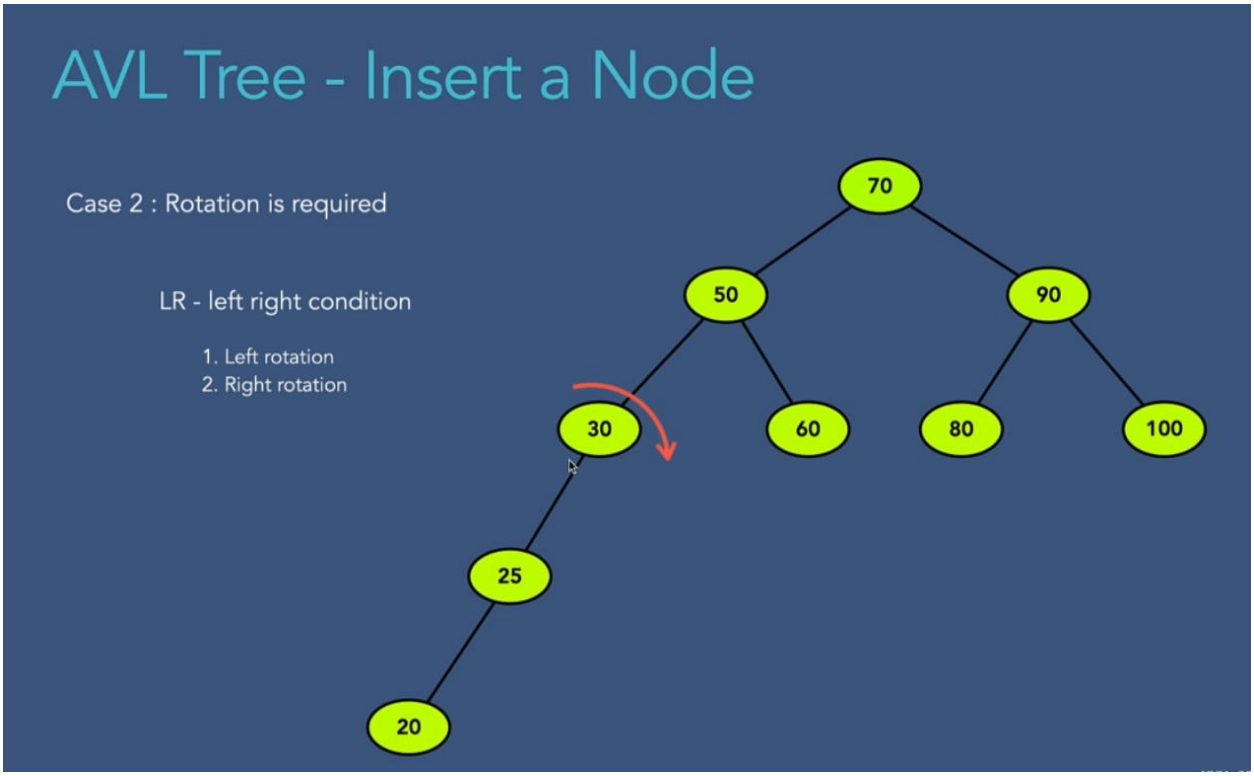




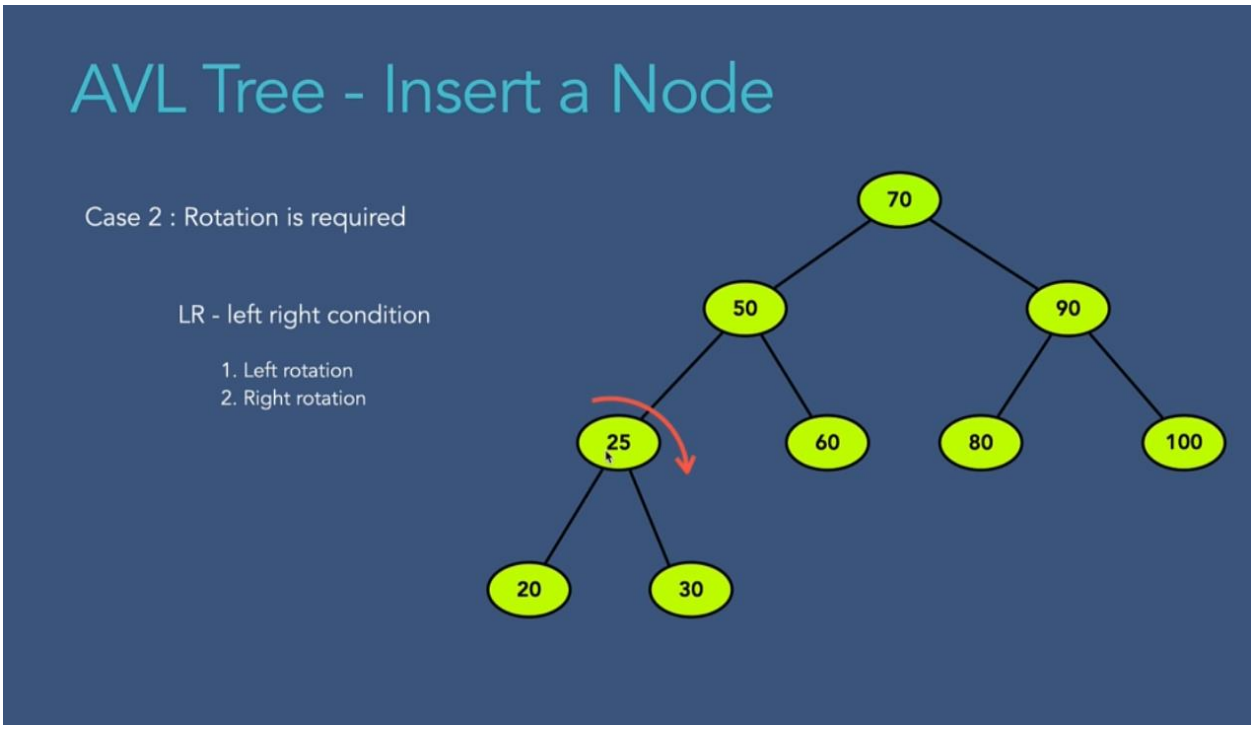


After left rotation





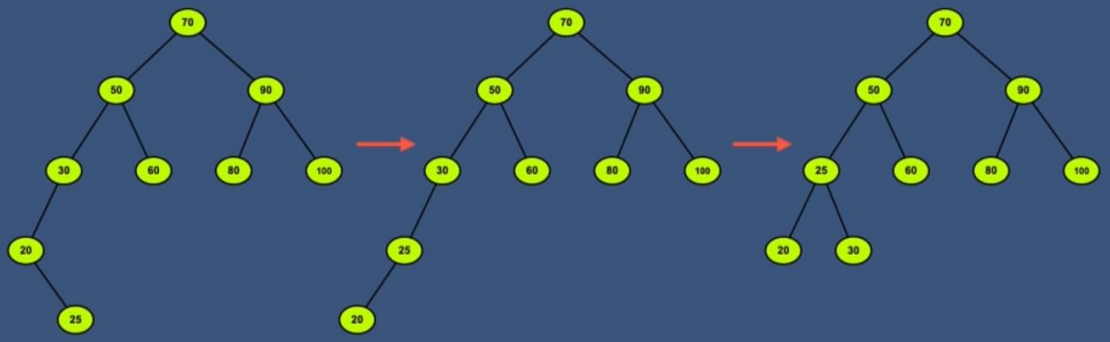
After rotation



AVL Tree - Insert a Node

Case 2 : Left Right Condition

1. Left Rotation
2. Right Rotation

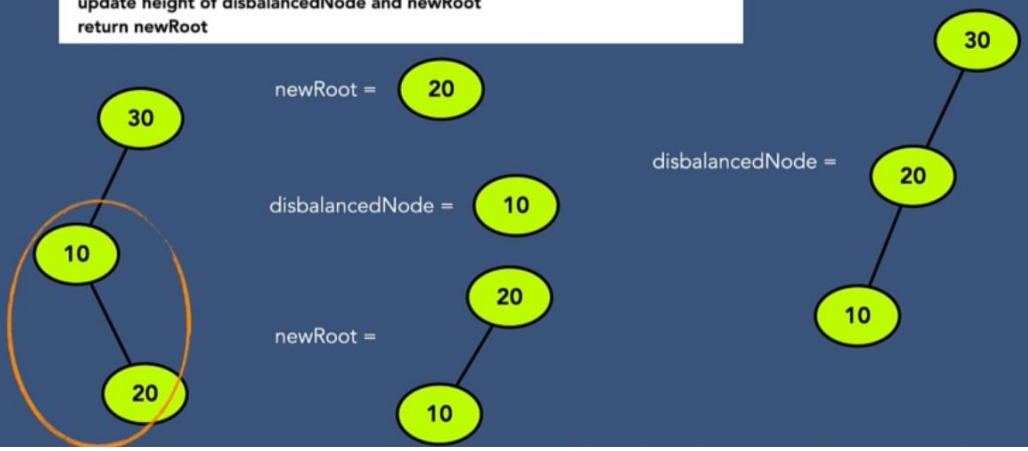


AVL Tree - Insert a Node

Case 2 : Left Right Condition

Step 1 : rotate Left disbalancedNode.leftChild
Step 2 : rotate Right disbalancedNode

```
rotateLeft(disbalancedNode)
  newRoot = disbalancedNode.rightChild
  disbalancedNode.rightChild = disbalancedNode.rightChild.leftChild
  newRoot.leftChild = disbalancedNode
  update height of disbalancedNode and newRoot
  return newRoot
```

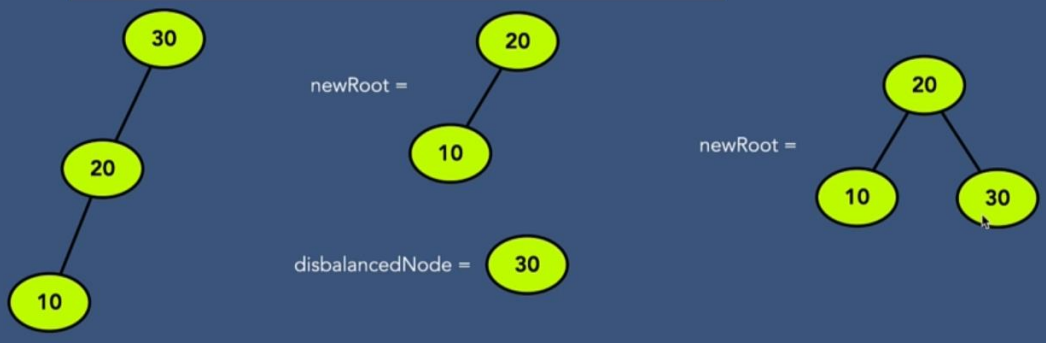


AVL Tree - Insert a Node

Case 2 : Left Right Condition

Step 1 : rotate Left disbalancedNode.leftChild
Step 2 : rotate Right disbalancedNode

```
rotateRight(disbalancedNode)
  newRoot = disbalancedNode.leftChild
  disbalancedNode.leftChild = disbalancedNode.leftChild.rightChild
  newRoot.rightChild = disbalancedNode
  update height of disbalancedNode and newRoot
  return newRoot
```



AVL Tree - Insert a Node

Case 2 : Left Right Condition

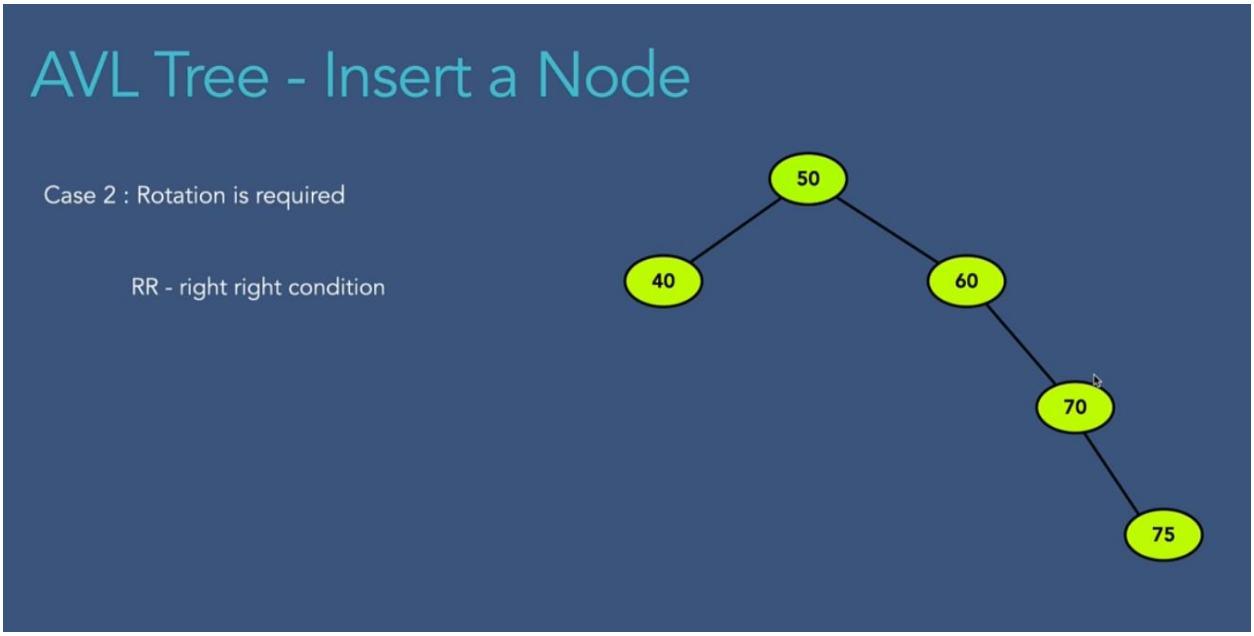
Step 1 : rotate Left disbalancedNode.leftChild
Step 2 : rotate Right disbalancedNode

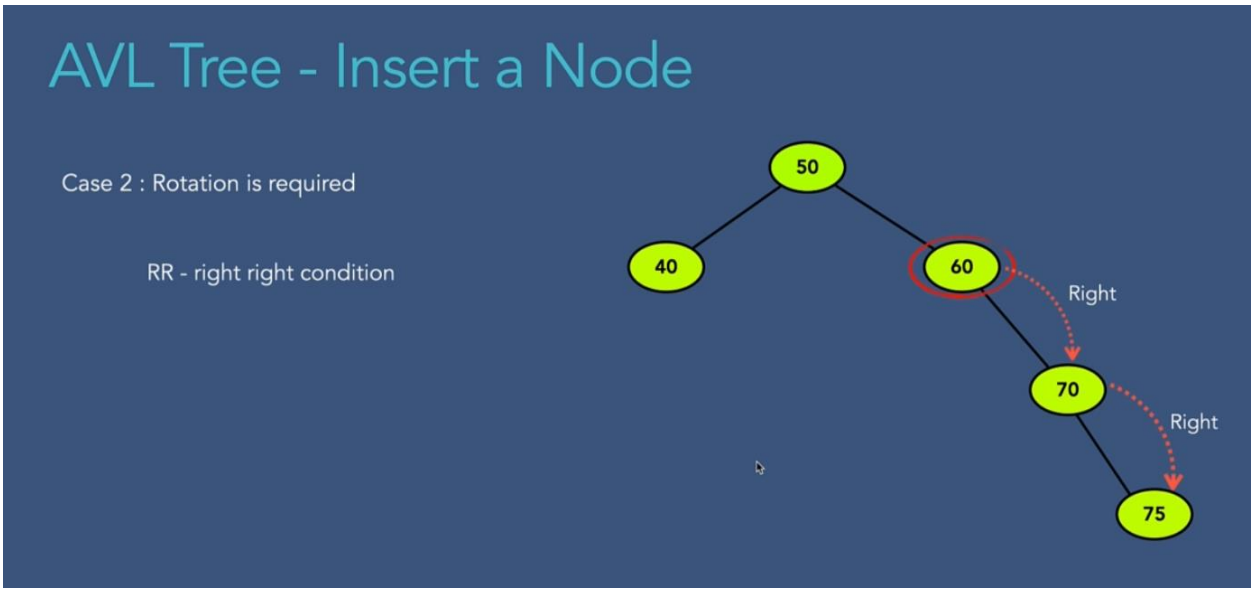
```
rotateLeft(disbalancedNode)
  newRoot = disbalancedNode.rightChild
  disbalancedNode.rightChild = disbalancedNode.rightChild.leftChild
  newRoot.leftChild = disbalancedNode
  update height of disbalancedNode and newRoot
  return newRoot
```

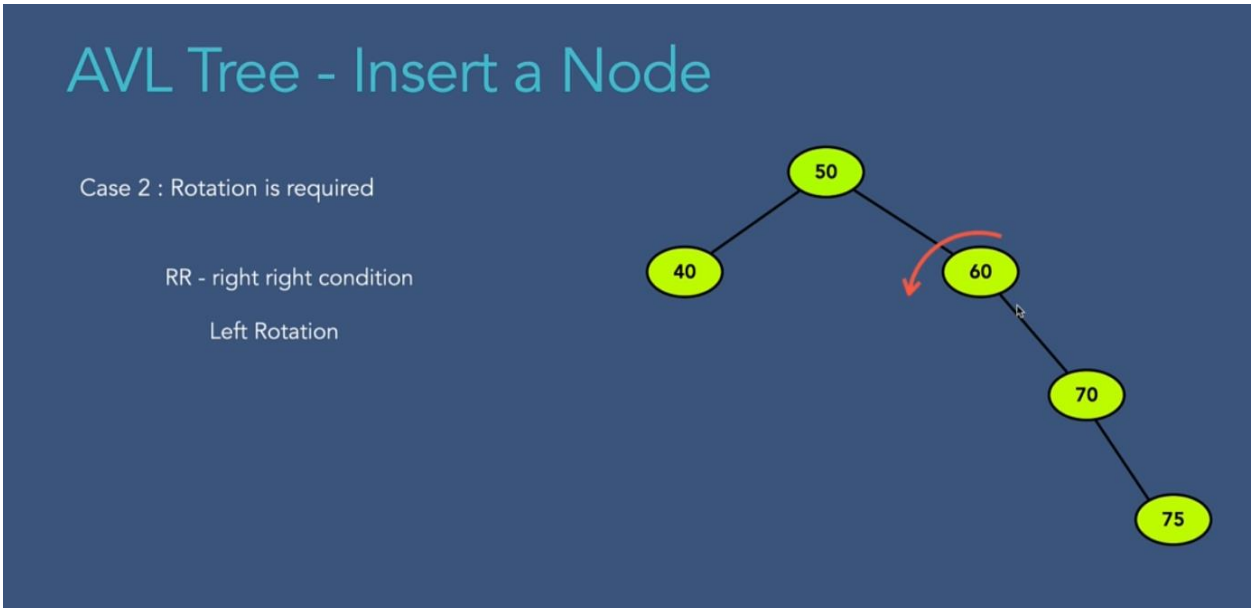
```
rotateRight(disbalancedNode)
  newRoot = disbalancedNode.leftChild
  disbalancedNode.leftChild = disbalancedNode.leftChild.rightChild
  newRoot.rightChild = disbalancedNode
  update height of disbalancedNode and newRoot
  return newRoot
```

Time complexity : O(1)
Space complexity : O(1)

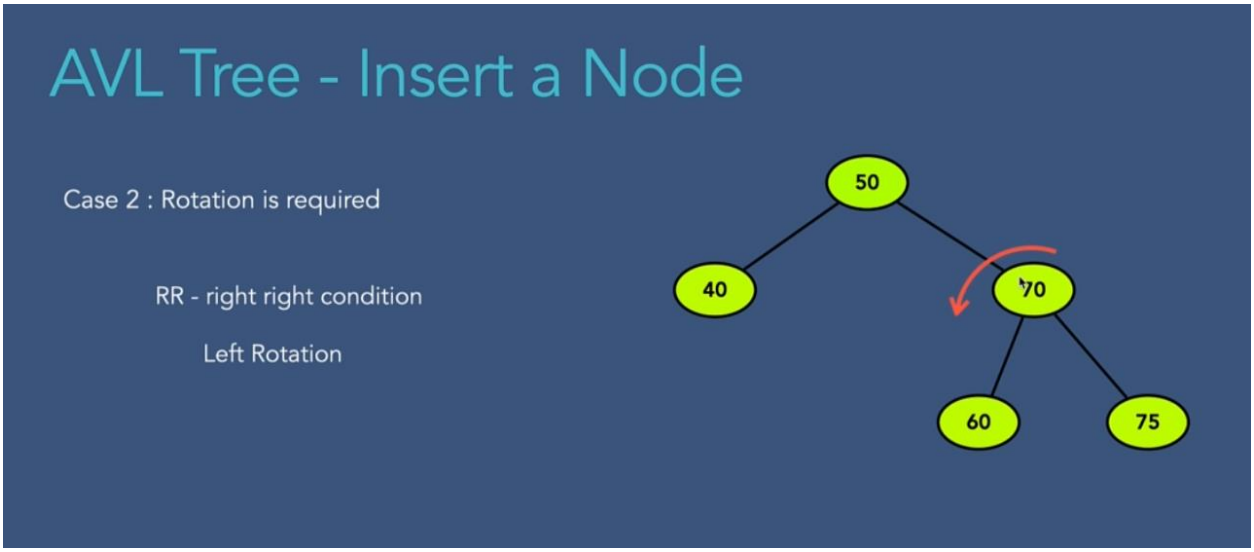
Insertion of Node 75 to the AVL Tree



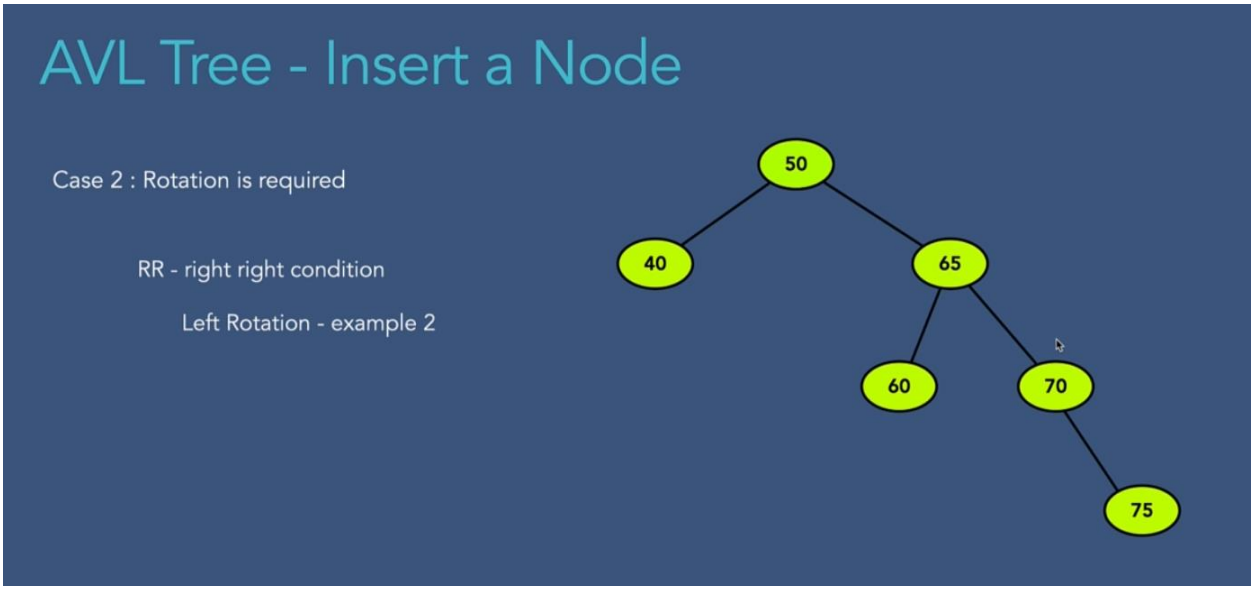


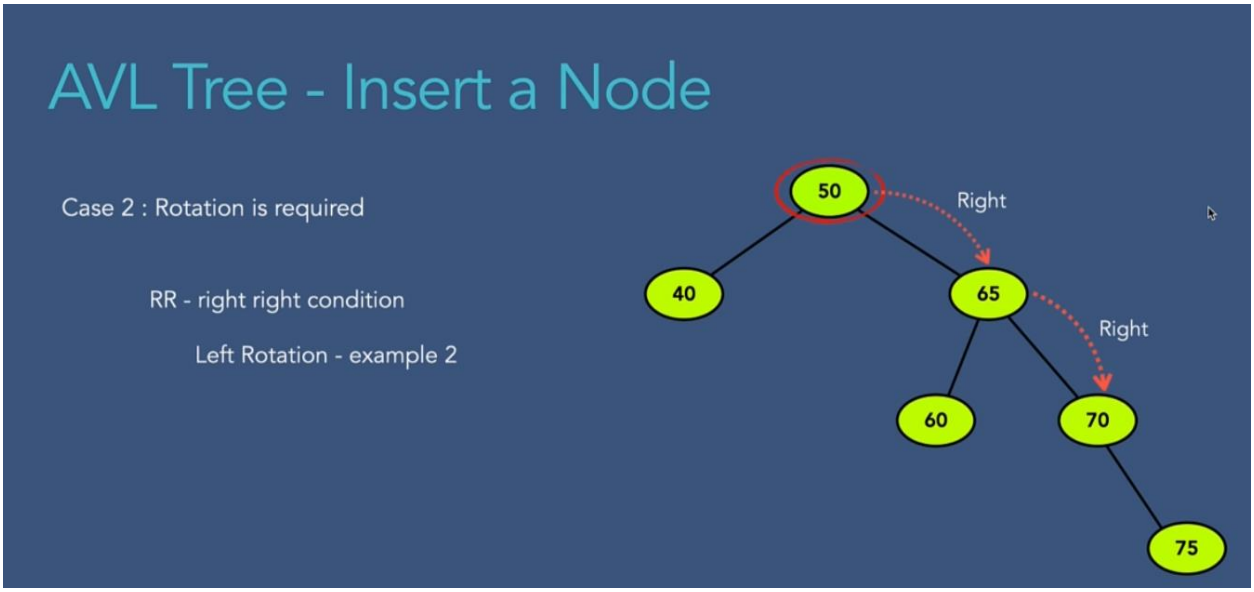


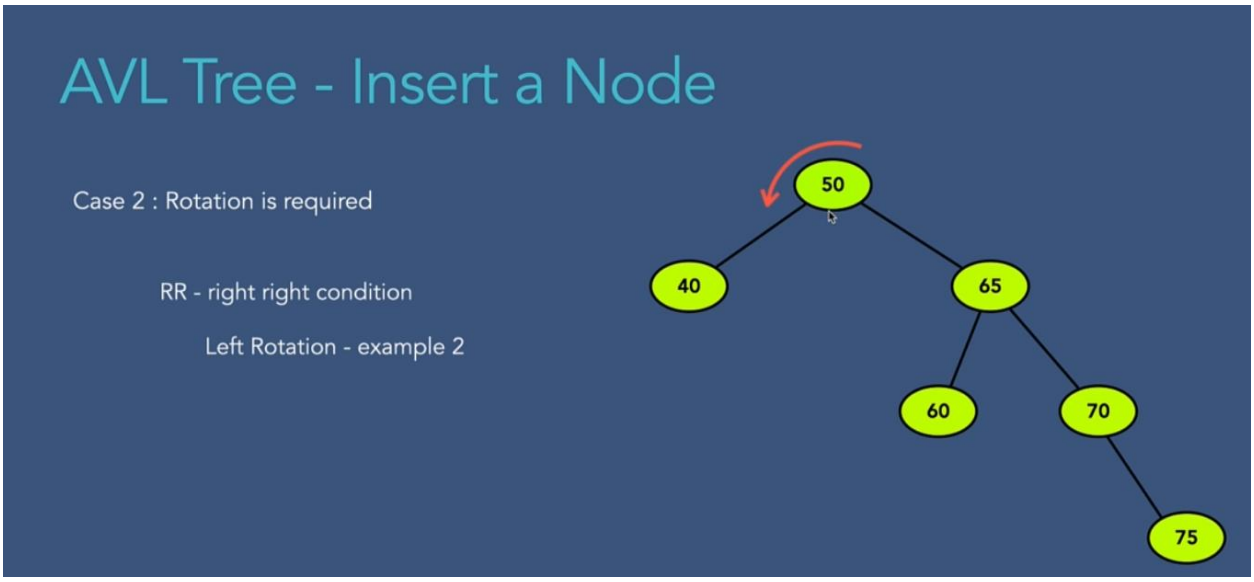
After rotation



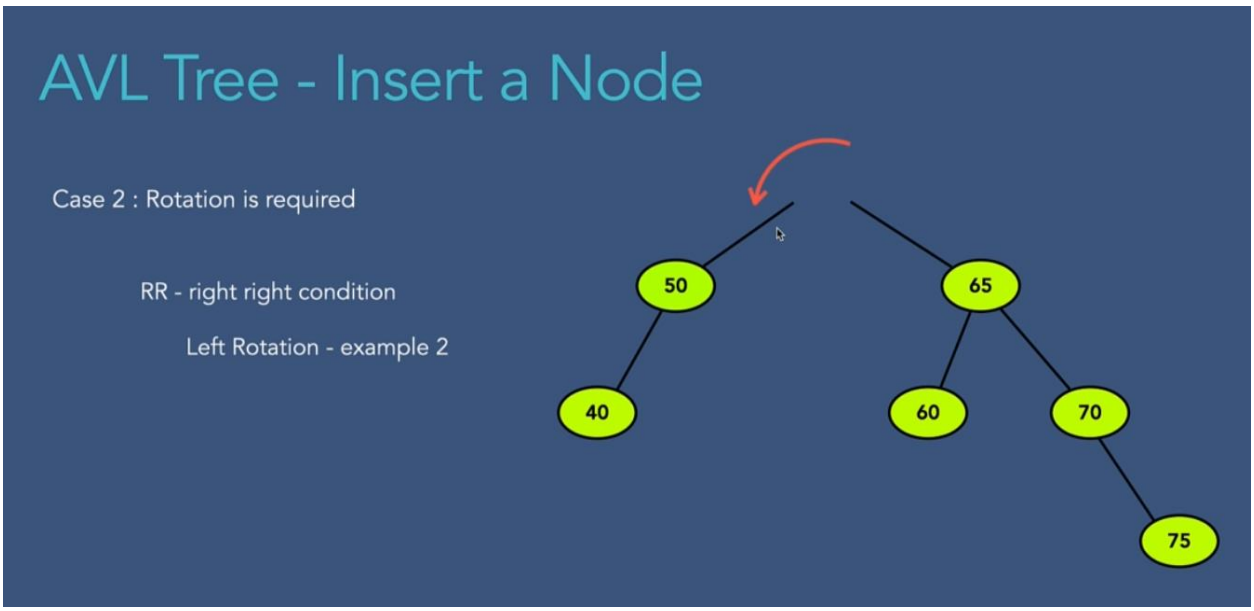
Again insertion of Node 75 to the AVL Tree







Here the root node is unbalanced (Height of left tree is 1 and the height of the right tree is 3. So the difference is 3-1=2 → so unbalanced)

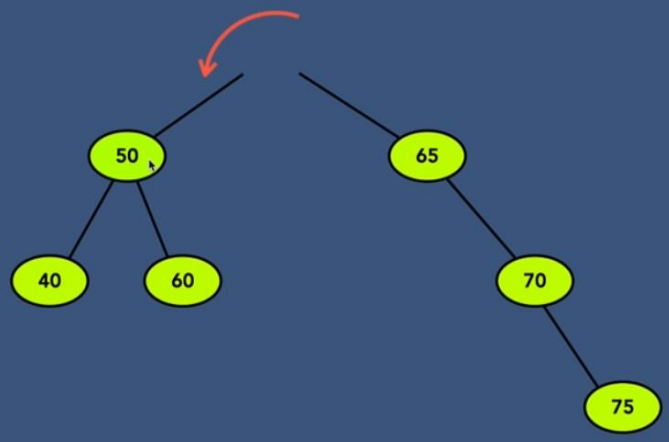


AVL Tree - Insert a Node

Case 2 : Rotation is required

RR - right right condition

Left Rotation - example 2

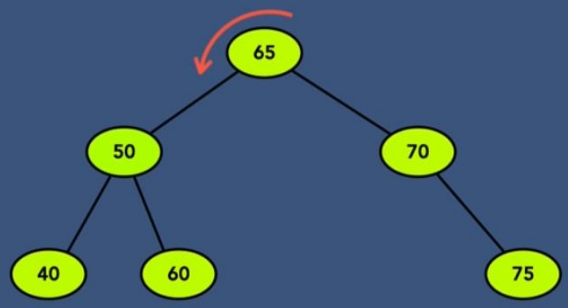


AVL Tree - Insert a Node

Case 2 : Rotation is required

RR - right right condition

Left Rotation - example 2

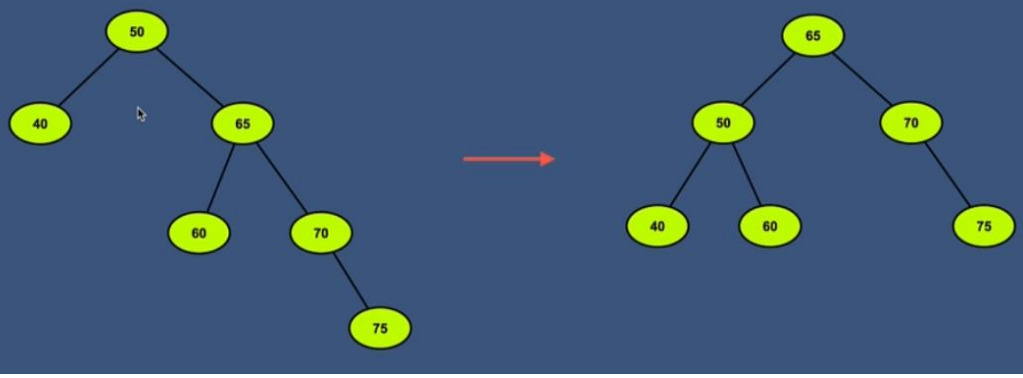


AVL Tree - Insert a Node

Case 2 : Rotation is required

RR - right right condition

Left Rotation



AVL Tree - Insert a Node

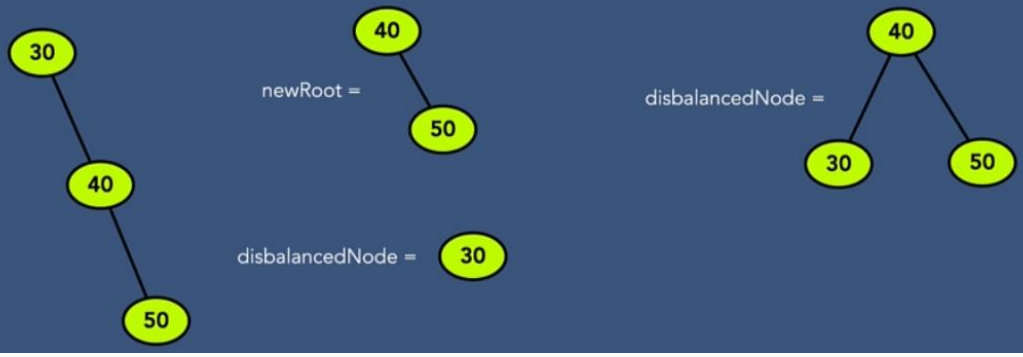
RR - right right condition

- rotate Left disbalancedNode

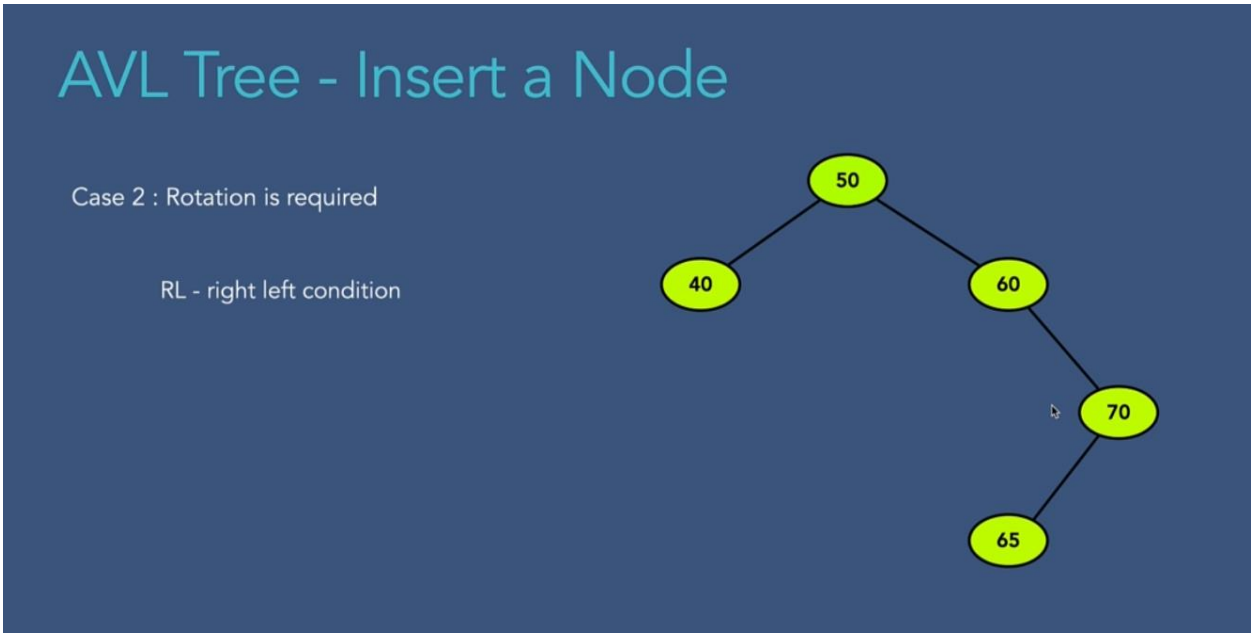
```
rotateLeft(disbalancedNode)
  newRoot = disbalancedNode.rightChild
  disbalancedNode.rightChild = disbalancedNode.rightChild.leftChild
  newRoot.leftChild = disbalancedNode
  update height of disbalancedNode and newRoot
  return newRoot
```

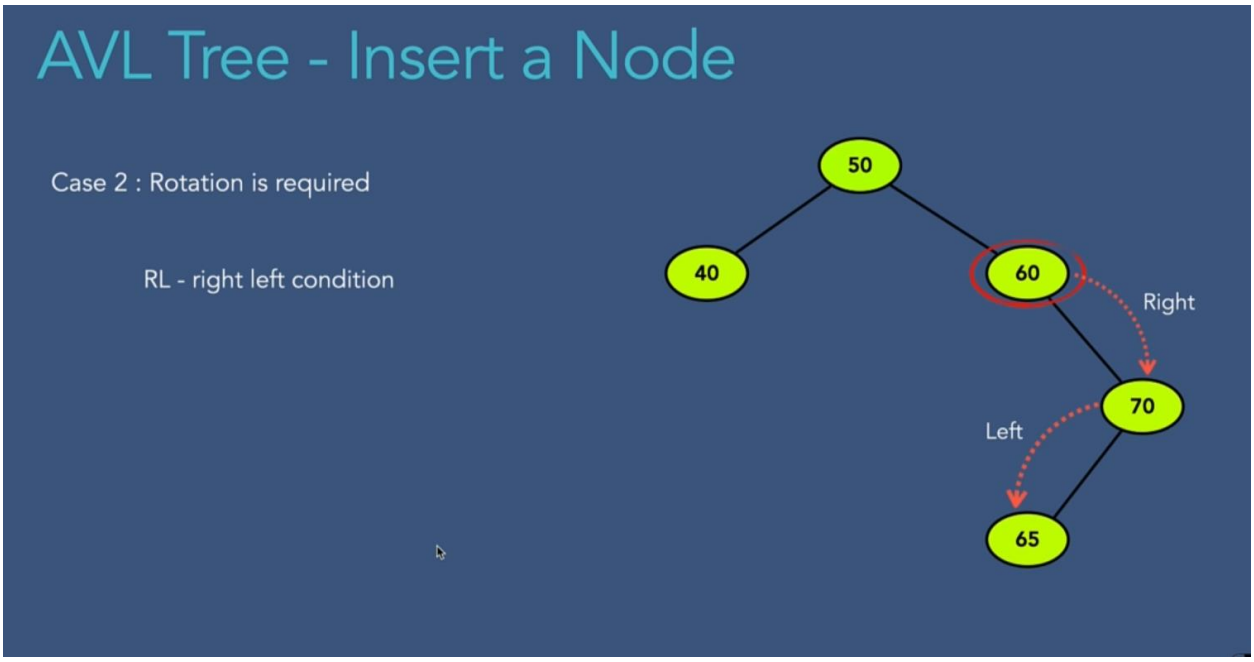
Time complexity : O(1)

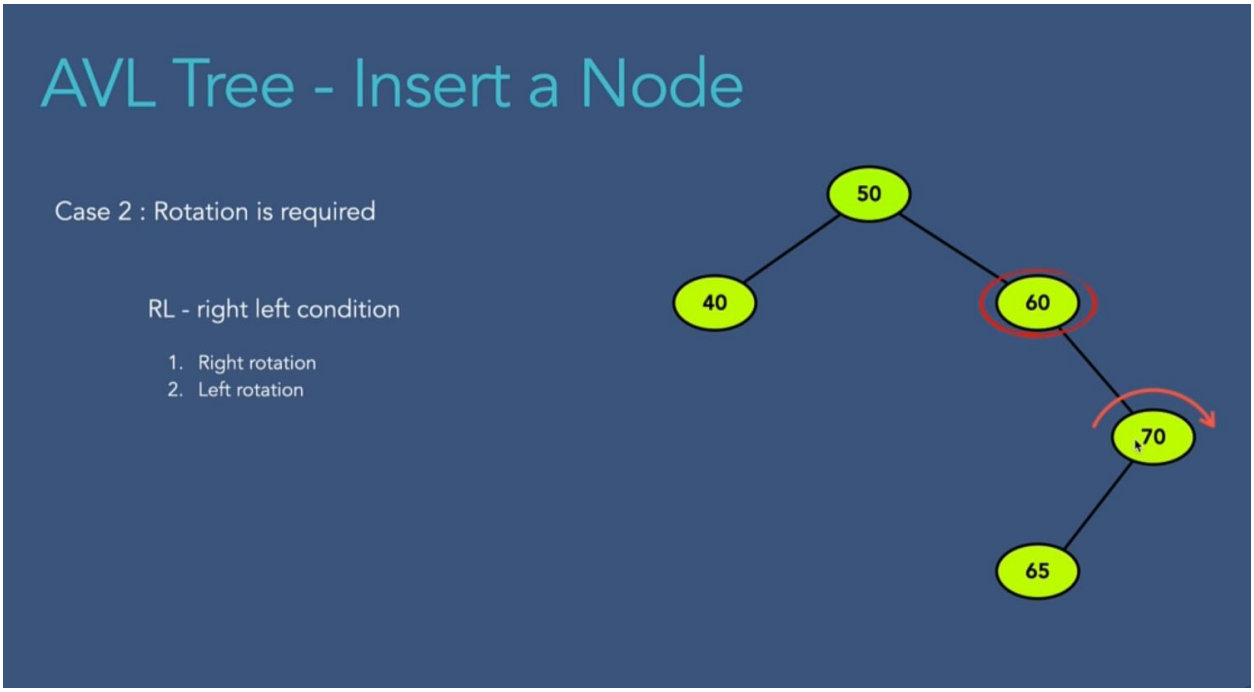
Space complexity : O(1)



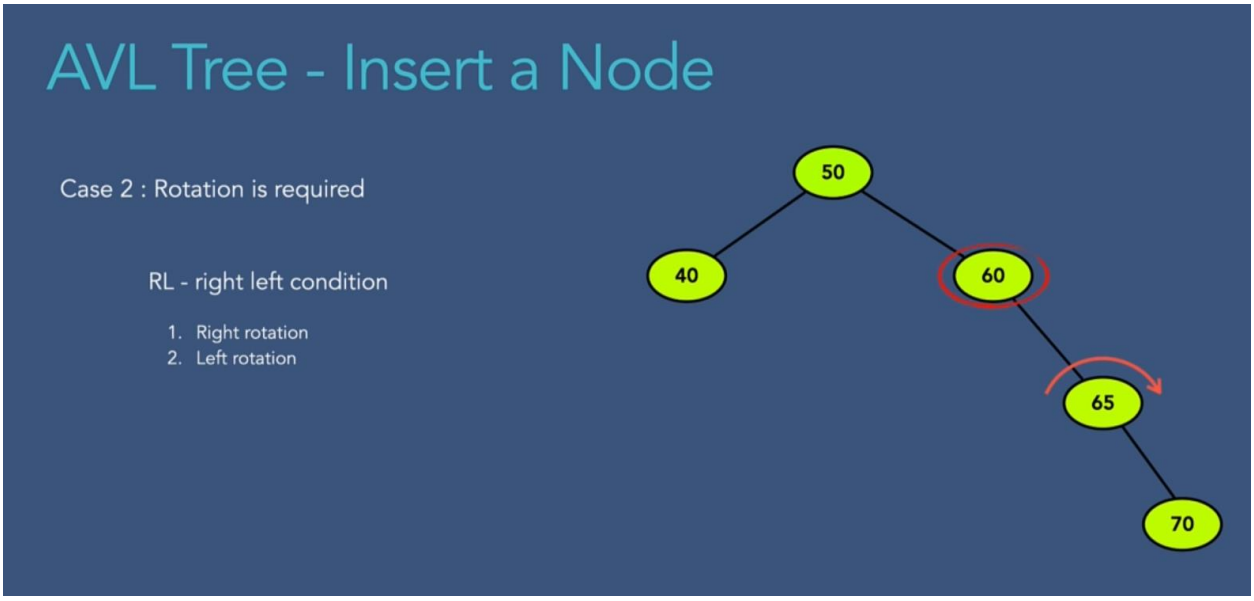
Insertion of 65 to the AVL Tree







After right rotation



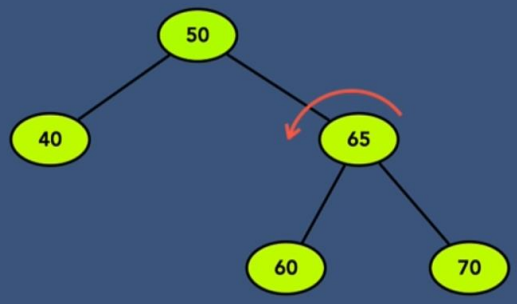
After left rotation

AVL Tree - Insert a Node

Case 2 : Rotation is required

RL - right left condition

1. Right rotation
2. Left rotation



AVL Tree - Insert a Node

RL - right left condition

Step 1 : rotate Right disbalancedNode.rightChild
Step 2 : rotate Left disbalancedNode

```
rotateRight(disbalancedNode)
  newRoot = disbalancedNode.leftChild
  disbalancedNode.leftChild = disbalancedNode.leftChild.rightChild
  newRoot.rightChild = disbalancedNode
  update height of disbalancedNode and newRoot
  return newRoot
```

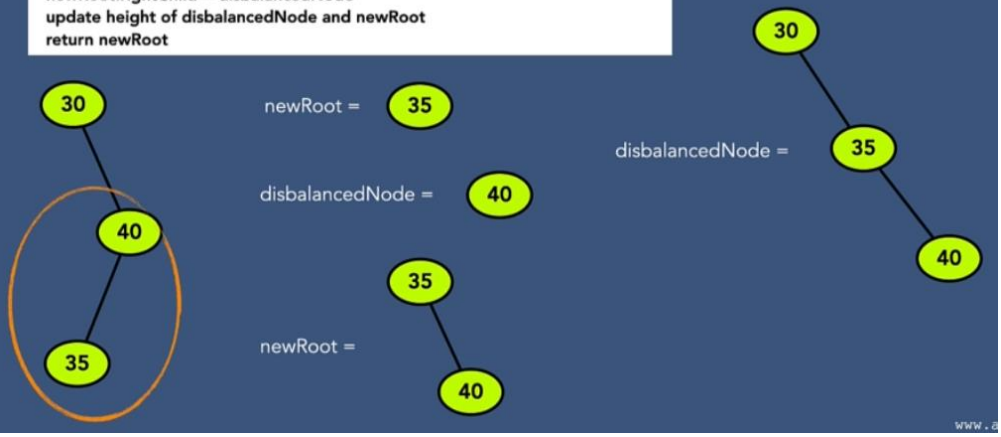
```
rotateLeft(disbalancedNode)
  newRoot = disbalancedNode.rightChild
  disbalancedNode.rightChild = disbalancedNode.rightChild.leftChild
  newRoot.leftChild = disbalancedNode
  update height of disbalancedNode and newRoot
  return newRoot
```

AVL Tree - Insert a Node

RL - right left condition

Step 1 : rotate Right disbalancedNode.rightChild
Step 2 : rotate Left disbalancedNode

```
rotateRight(disbalancedNode)
  newRoot = disbalancedNode.leftChild
  disbalancedNode.leftChild = disbalancedNode.leftChild.rightChild
  newRoot.rightChild = disbalancedNode
  update height of disbalancedNode and newRoot
  return newRoot
```

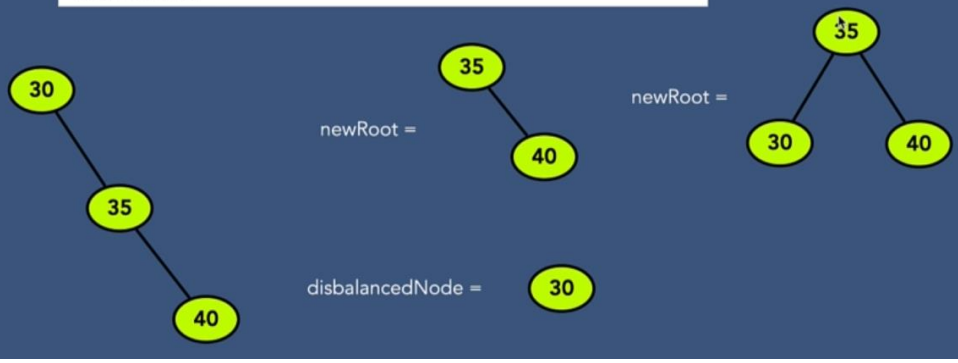


AVL Tree - Insert a Node

RL - right left condition

Step 1 : rotate Right disbalancedNode.rightChild
Step 2 : rotate Left disbalancedNode


```
rotateLeft(disbalancedNode)
  newRoot = disbalancedNode.rightChild
  disbalancedNode.rightChild = disbalancedNode.rightChild.leftChild
  newRoot.leftChild = disbalancedNode
  update height of disbalancedNode and newRoot
  return newRoot
```



AVL Tree - Insert a Node

RL - right left condition

Step 1 : rotate Right disbalancedNode.rightChild
Step 2 : rotate Left disbalancedNode

```
rotateRight(disbalancedNode)
  newRoot = disbalancedNode.leftChild
  disbalancedNode.leftChild = disbalancedNode.leftChild.rightChild
  newRoot.rightChild = disbalancedNode
  update height of disbalancedNode and newRoot
  return newRoot
```

```
rotateLeft(disbalancedNode)
  newRoot = disbalancedNode.rightChild
  disbalancedNode.rightChild = disbalancedNode.rightChild.leftChild
  newRoot.leftChild = disbalancedNode
  update height of disbalancedNode and newRoot
  return newRoot
```

Time complexity : O(1)
Space complexity : O(1)

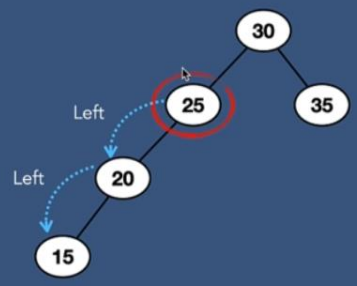
AVL Tree - Insert a Node (All together)

Case 1 : Rotation is not required
Case 2 : Rotation is required

- Left Left condition (LL)
- Left Right condition (LR)
- Right Right condition (RR)
- Right Left condition (RL)

AVL Tree - Insert a Node (All together)

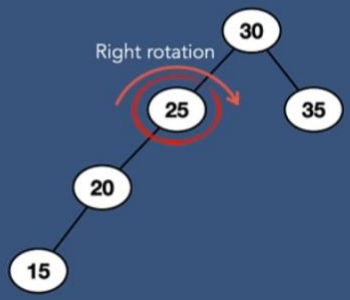
30,25,35,20,15,5,10,50,60,70,65



- Left Left condition (LL)
- Left Right condition (LR)
- Right Right condition (RR)
- Right Left condition (RL)

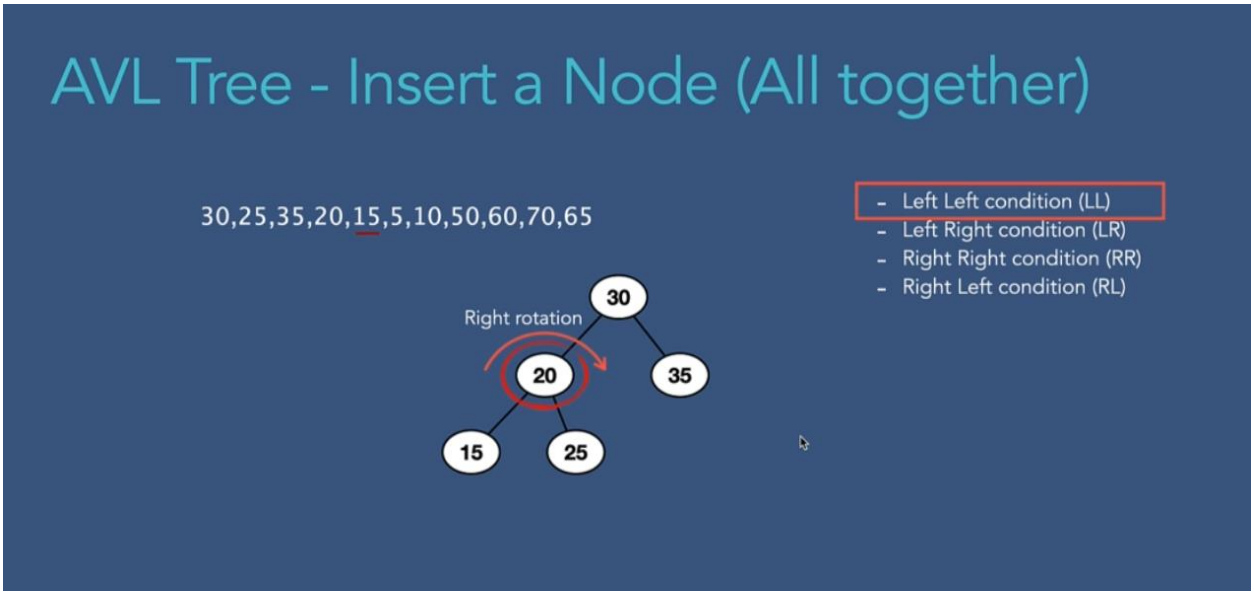
AVL Tree - Insert a Node (All together)

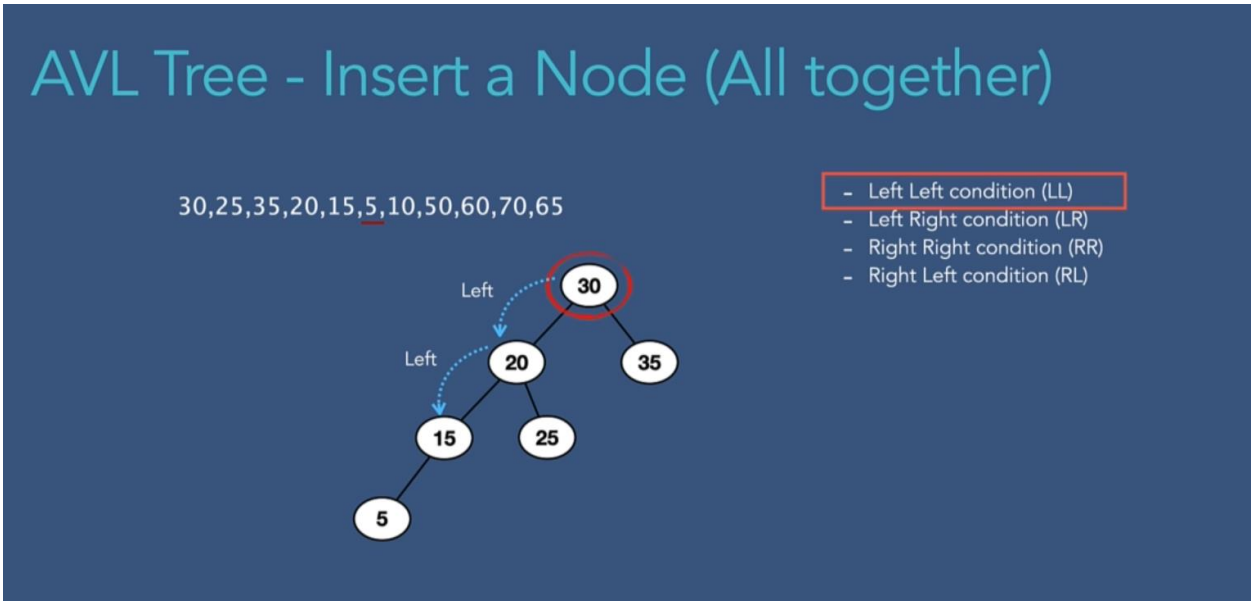
30,25,35,20,15,5,10,50,60,70,65



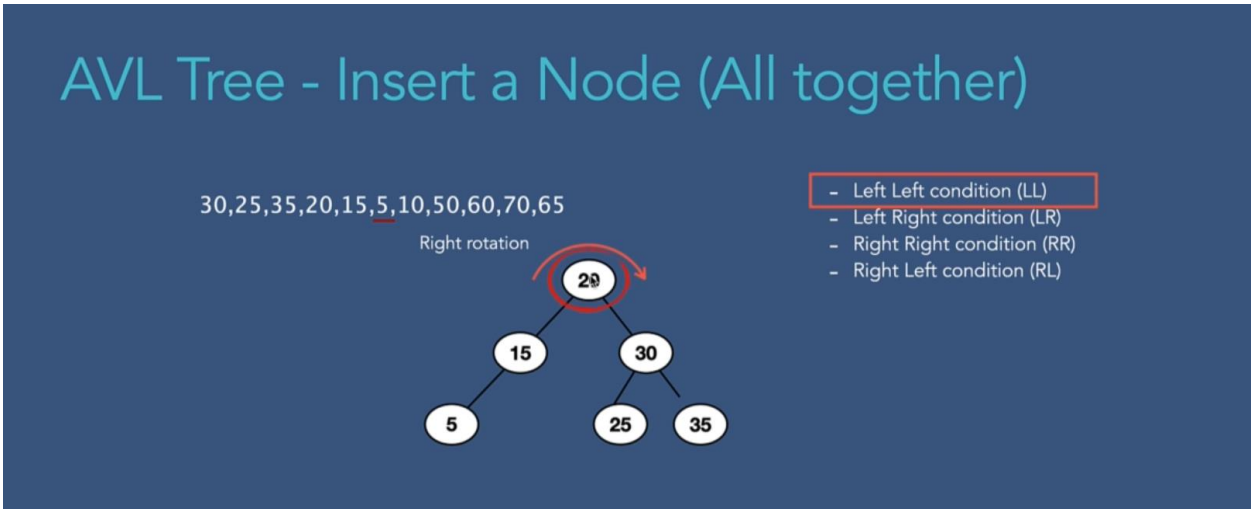
- Left Left condition (LL)
- Left Right condition (LR)
- Right Right condition (RR)
- Right Left condition (RL)

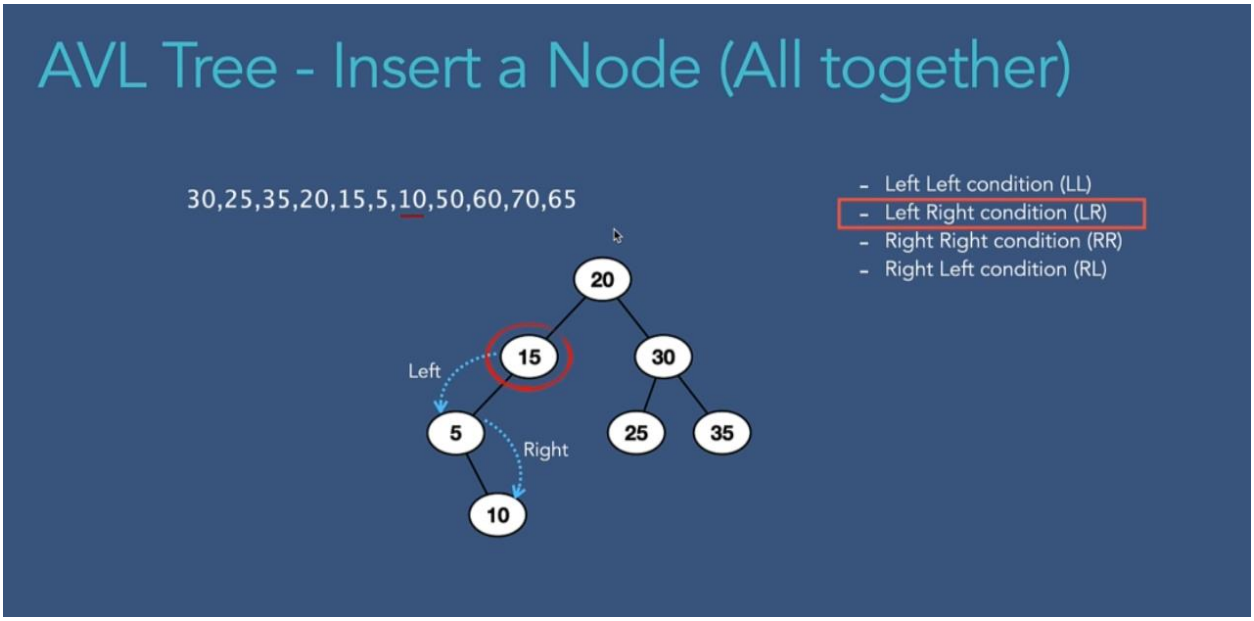
After rotation

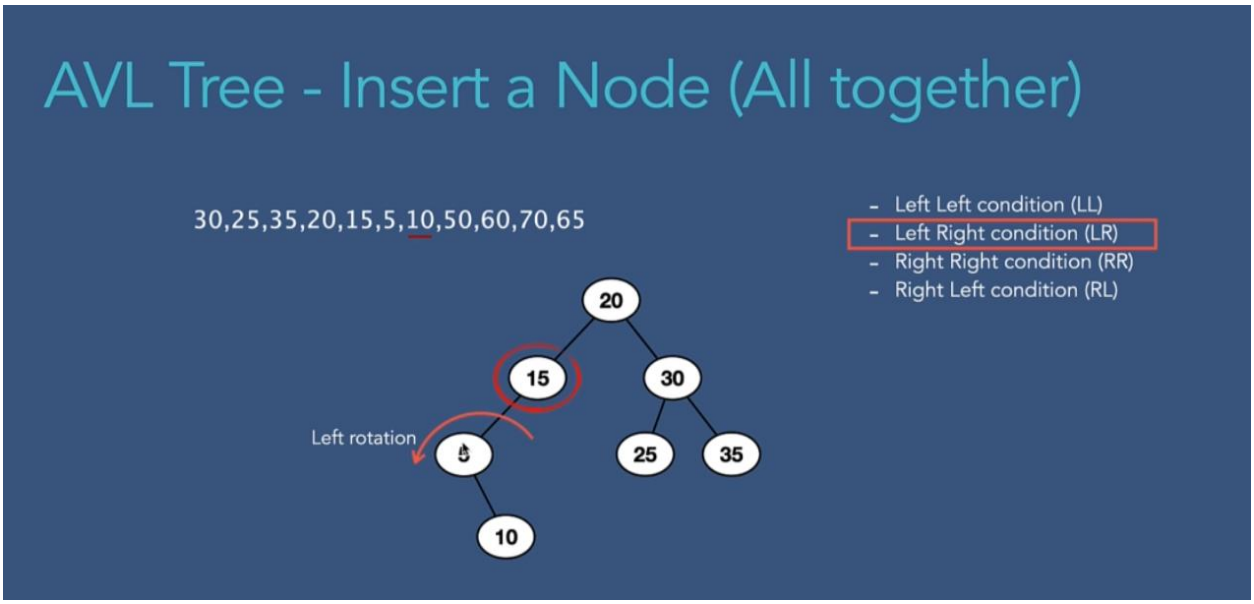




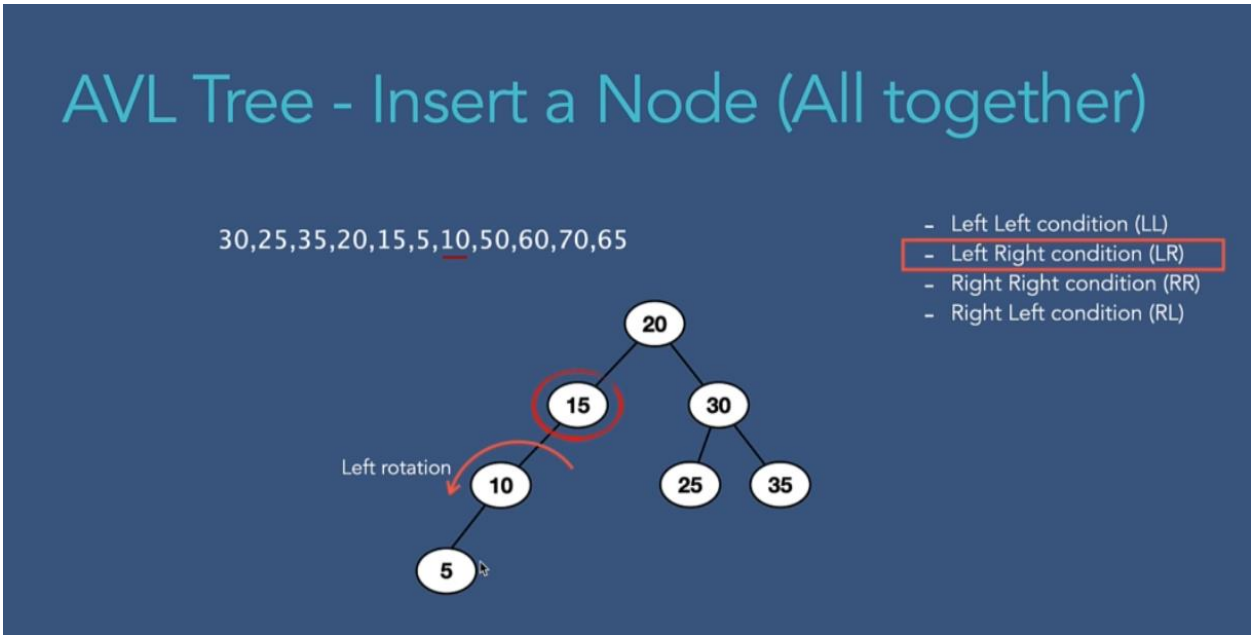
After rotation

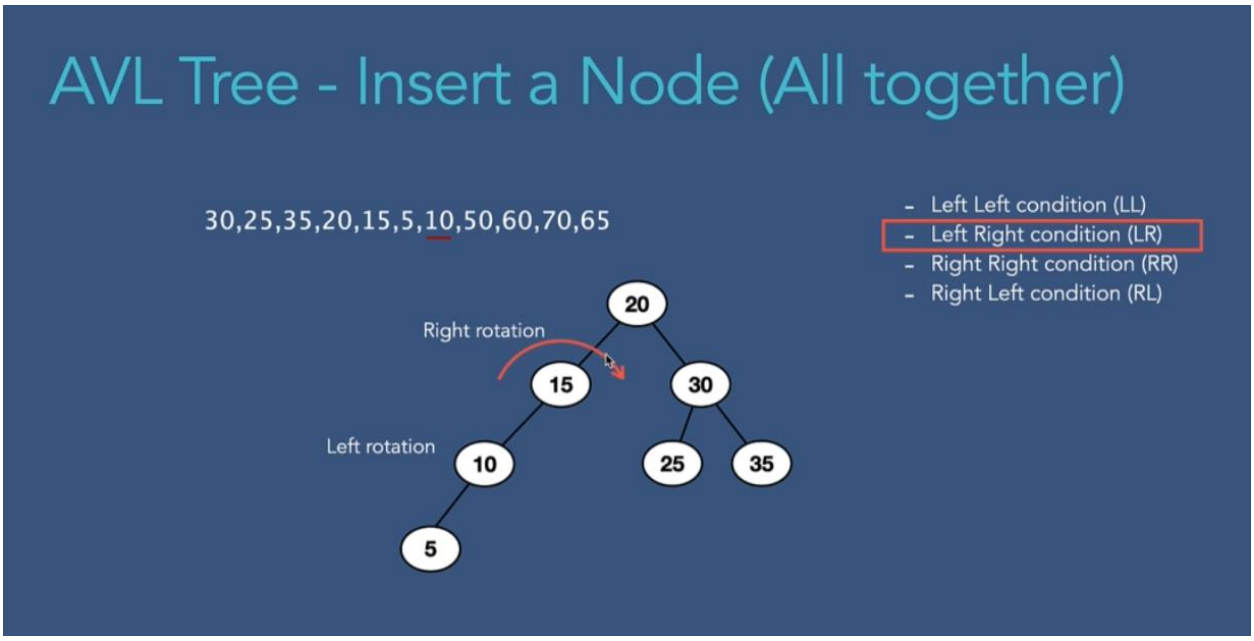




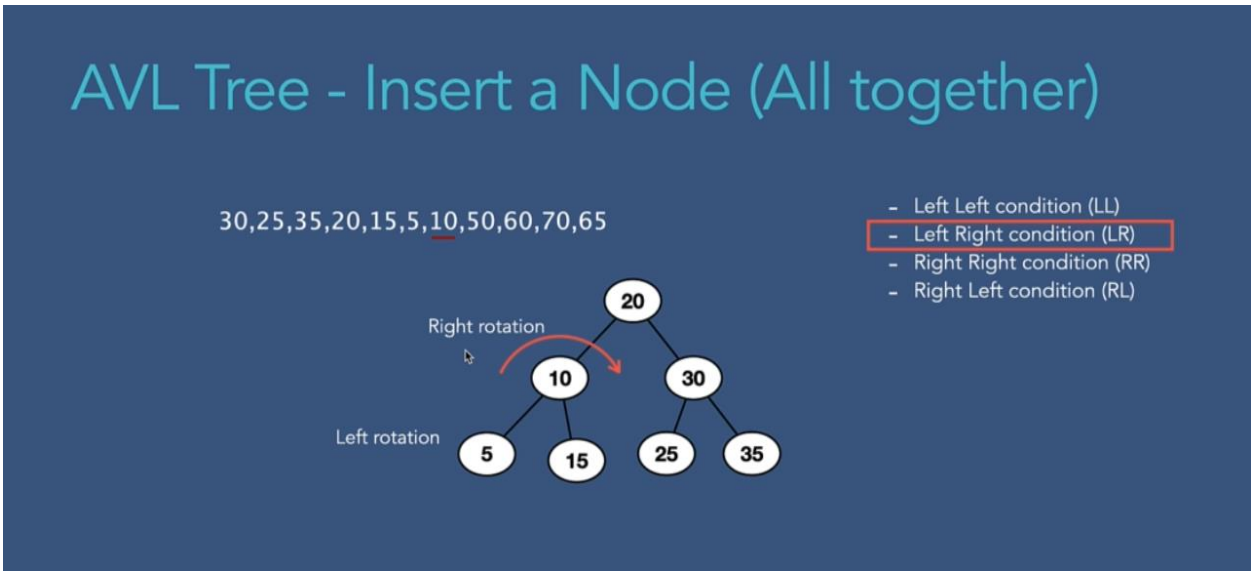


After left rotation





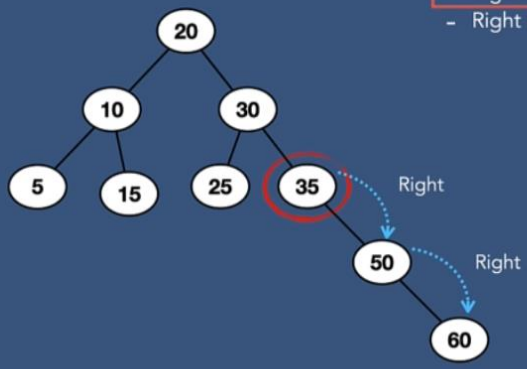
After right rotation



AVL Tree - Insert a Node (All together)

30,25,35,20,15,5,10,50,60,70,65

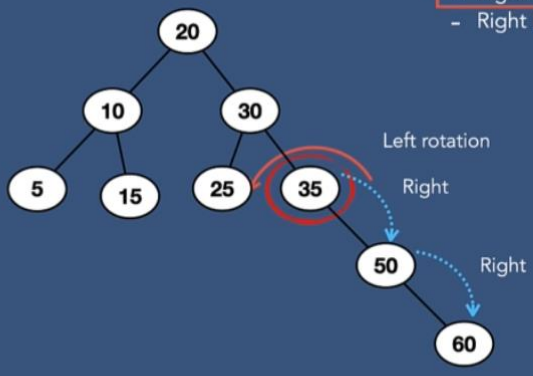
- Left Left condition (LL)
- Left Right condition (LR)
- Right Right condition (RR)
- Right Left condition (RL)



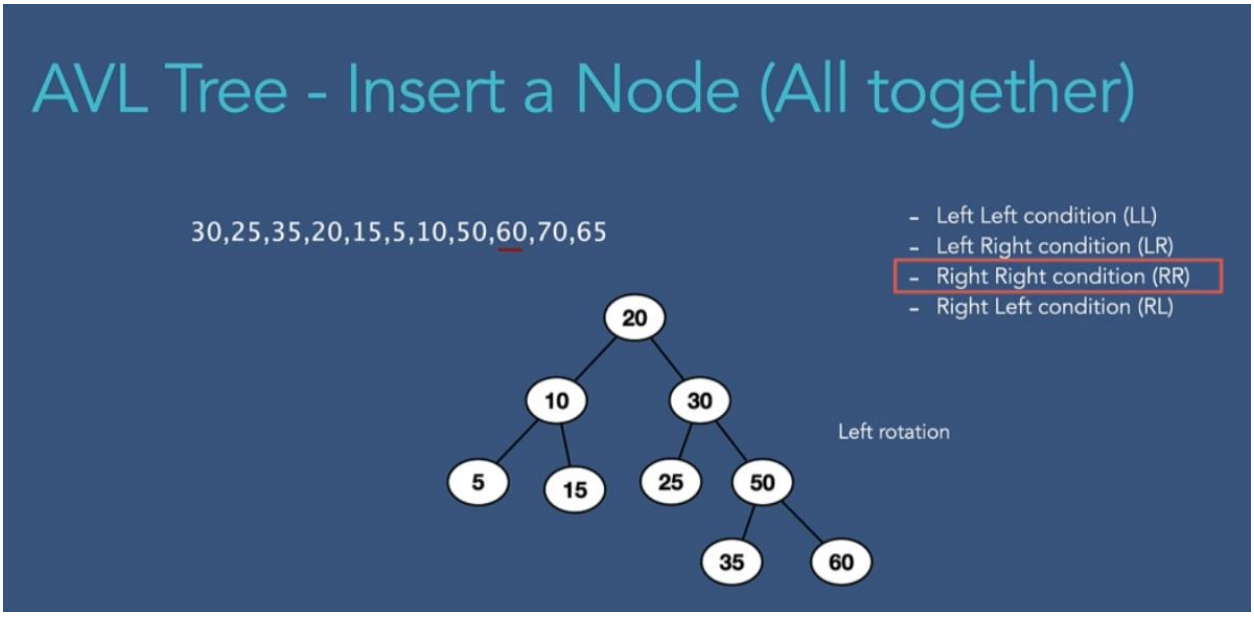
AVL Tree - Insert a Node (All together)

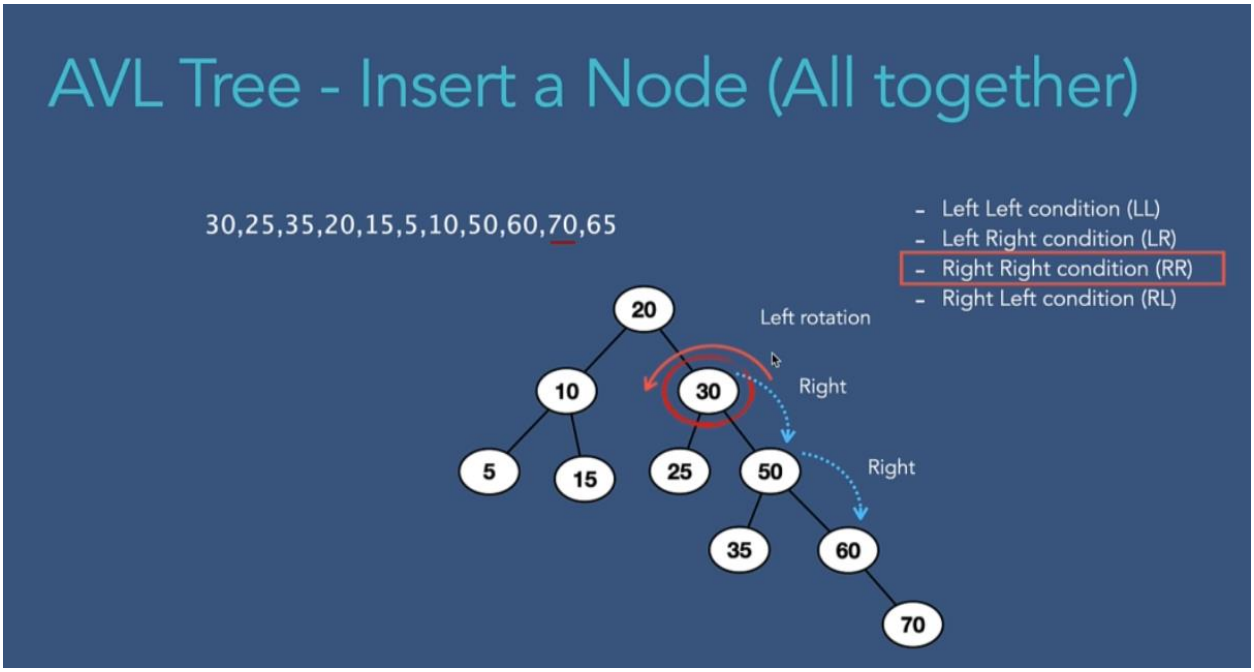
30,25,35,20,15,5,10,50,60,70,65

- Left Left condition (LL)
- Left Right condition (LR)
- Right Right condition (RR)
- Right Left condition (RL)

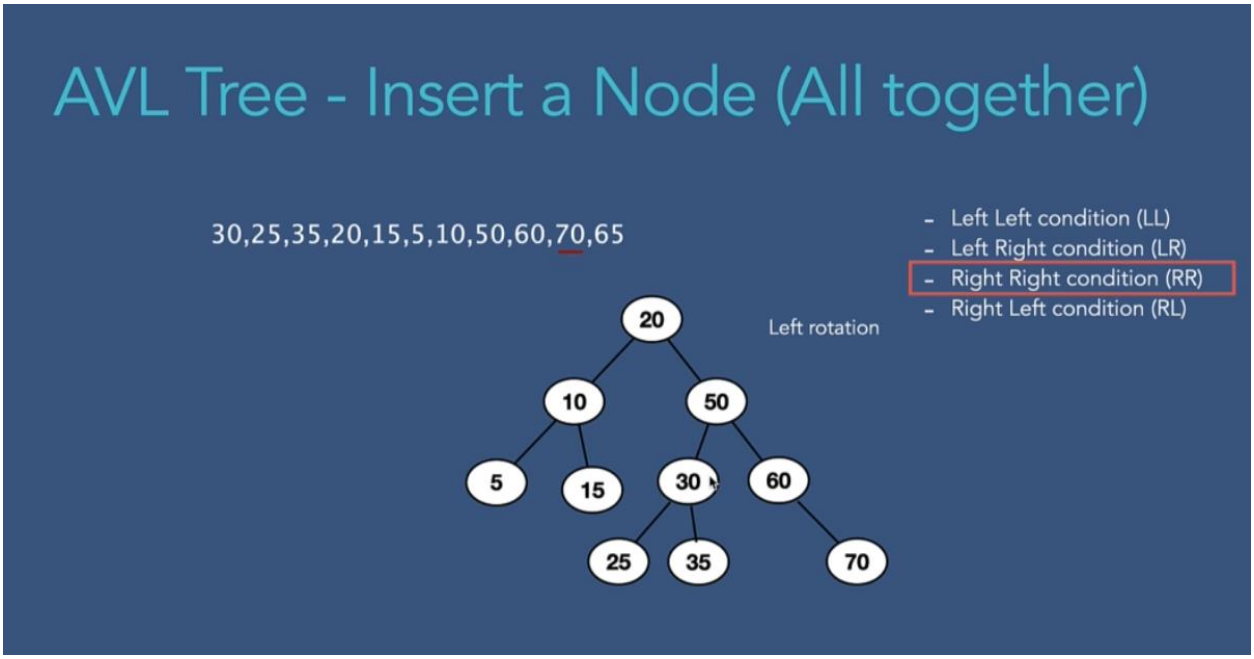


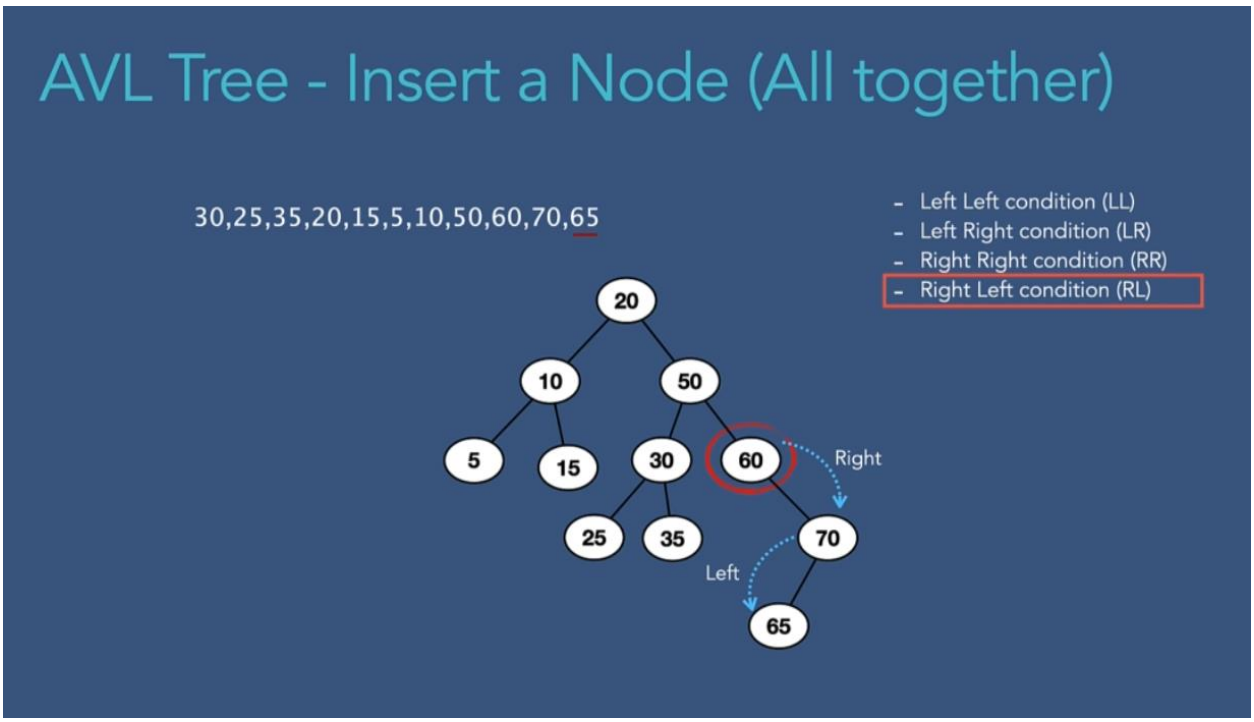
After rotation

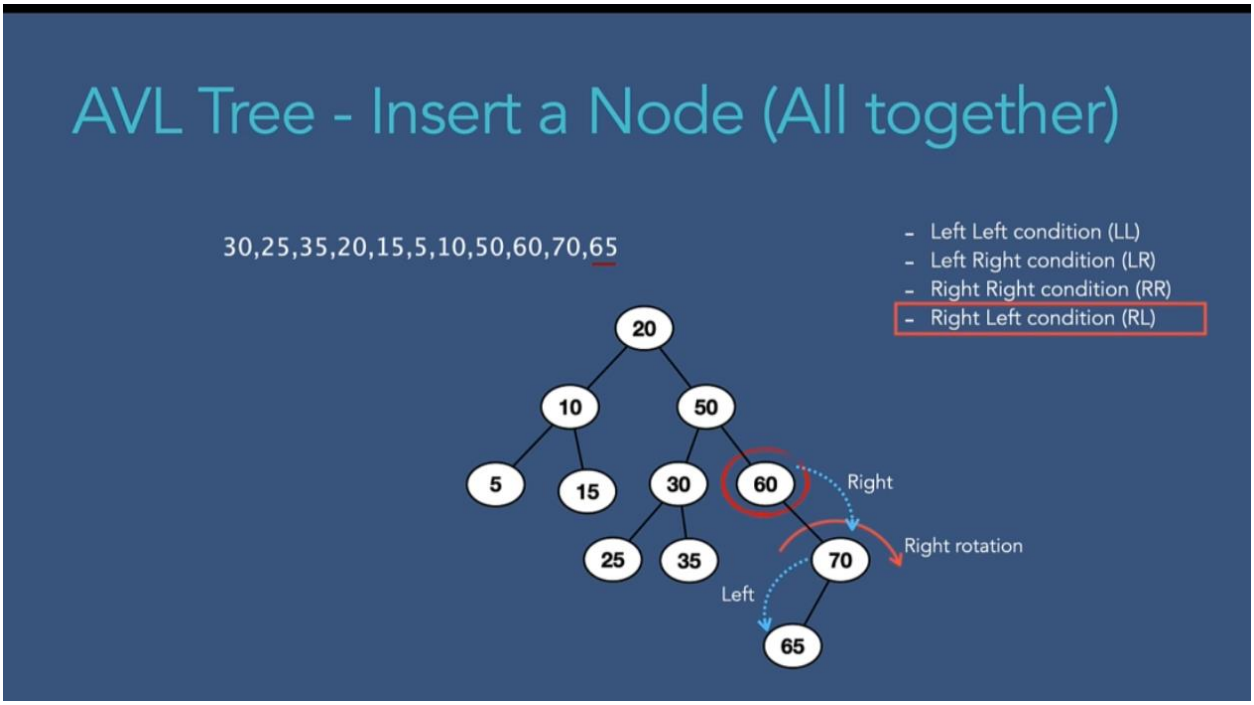




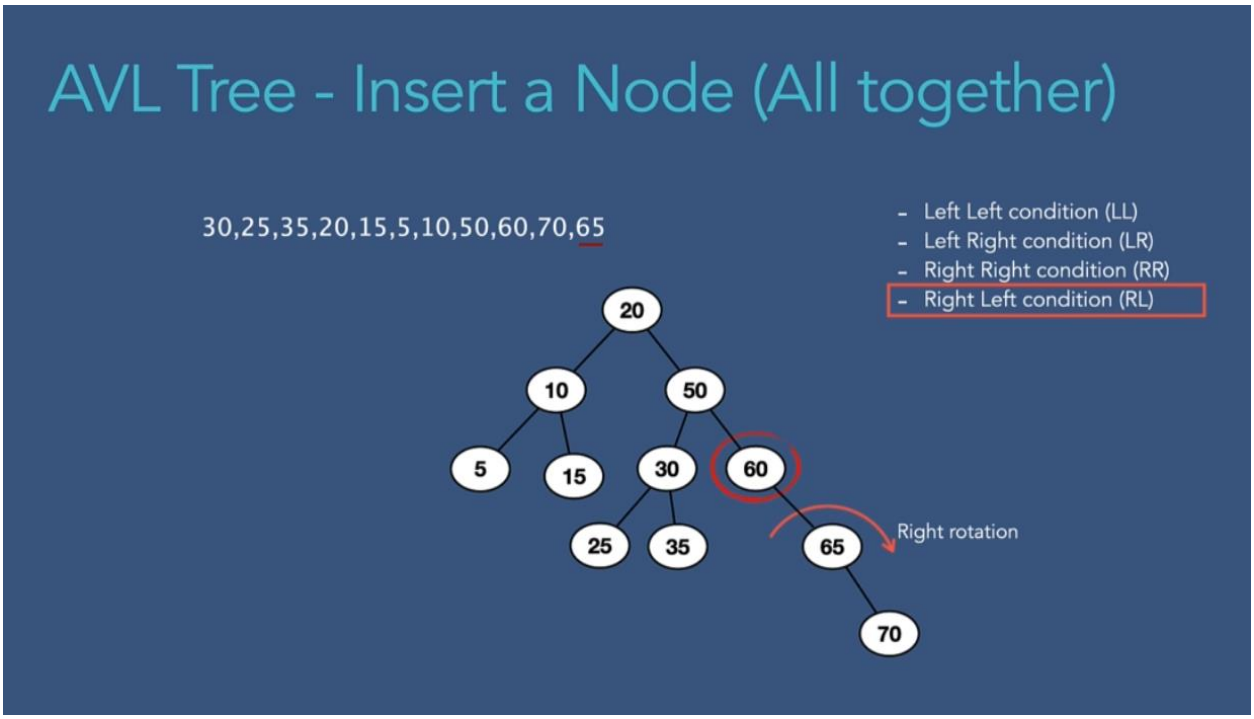
After left rotation

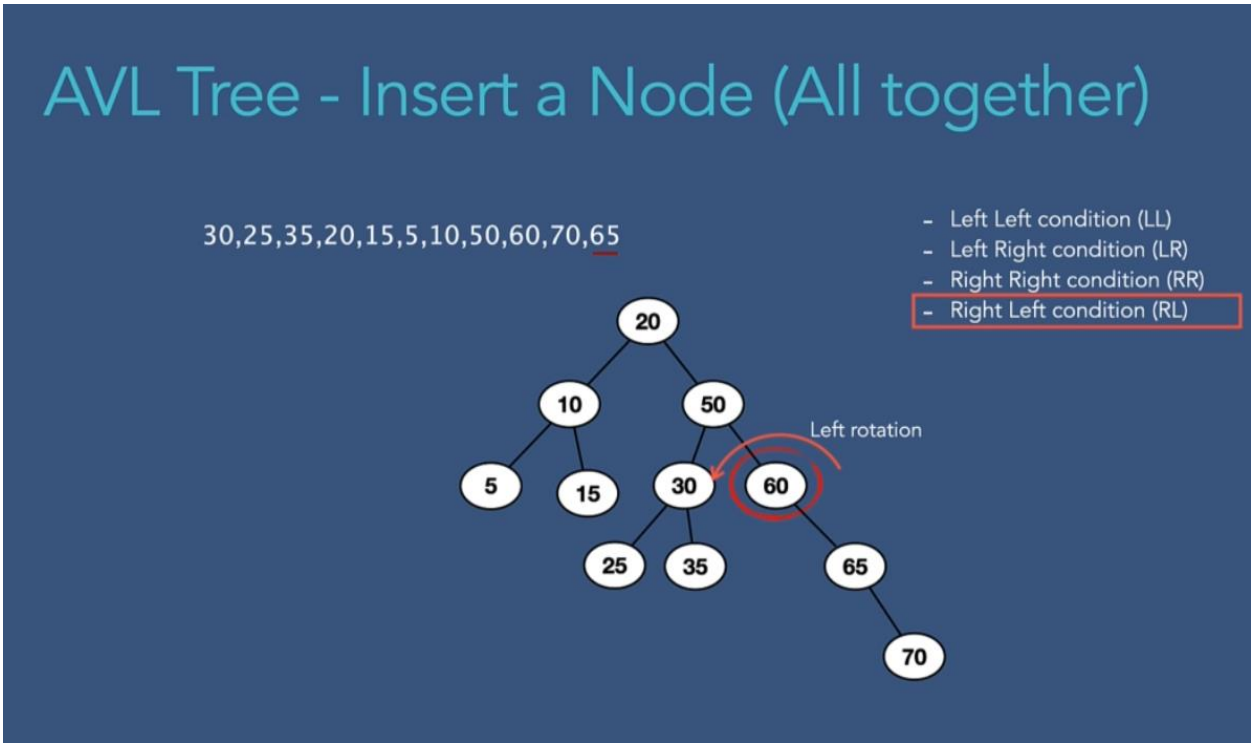




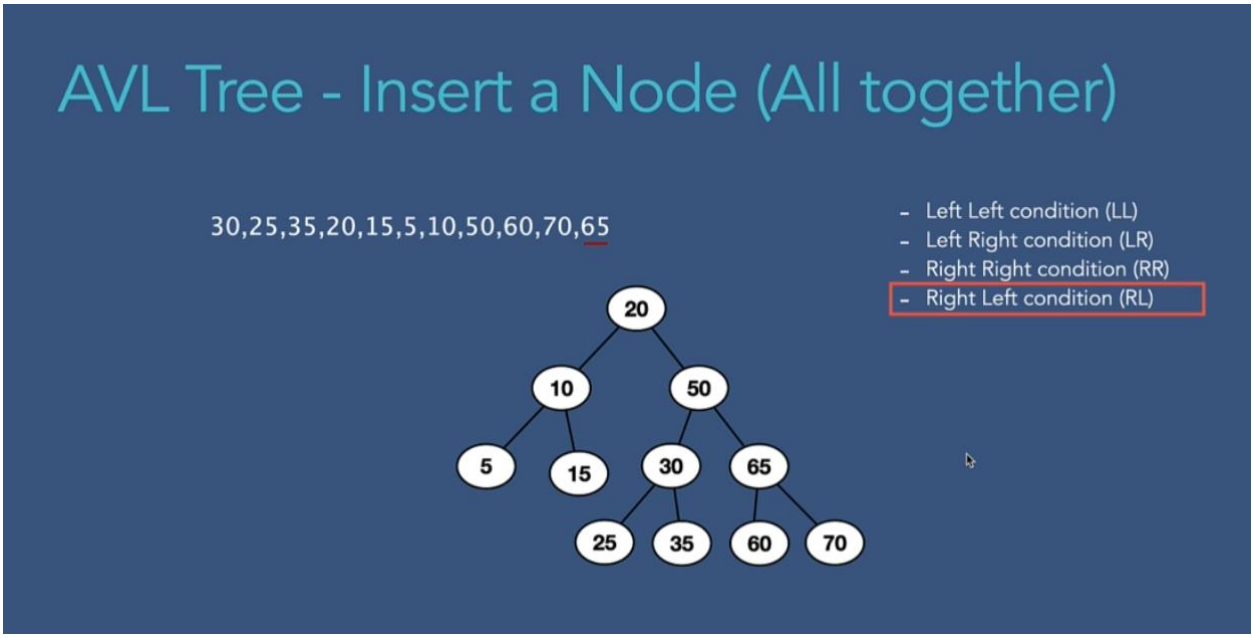


After right rotation



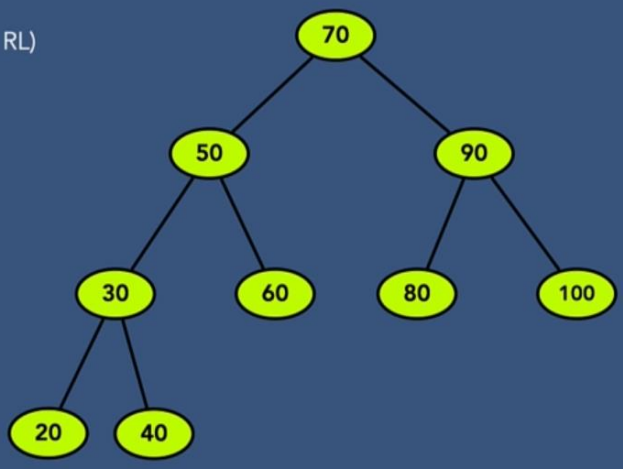


After left rotation



AVL Tree - Delete a Node

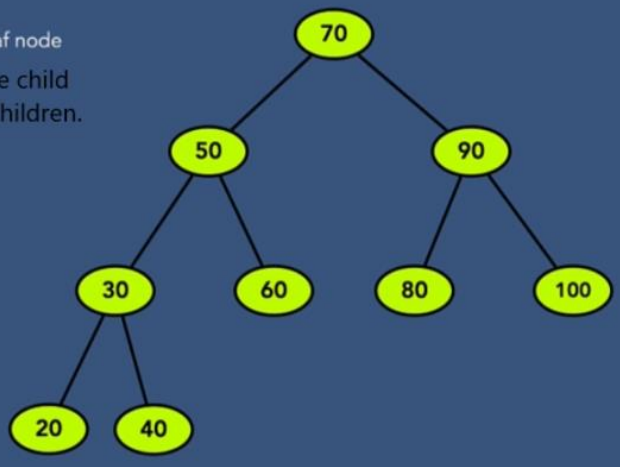
Case 1 - Rotation is not required
Case 2 - Rotation is required (LL, LR, RR, RL)



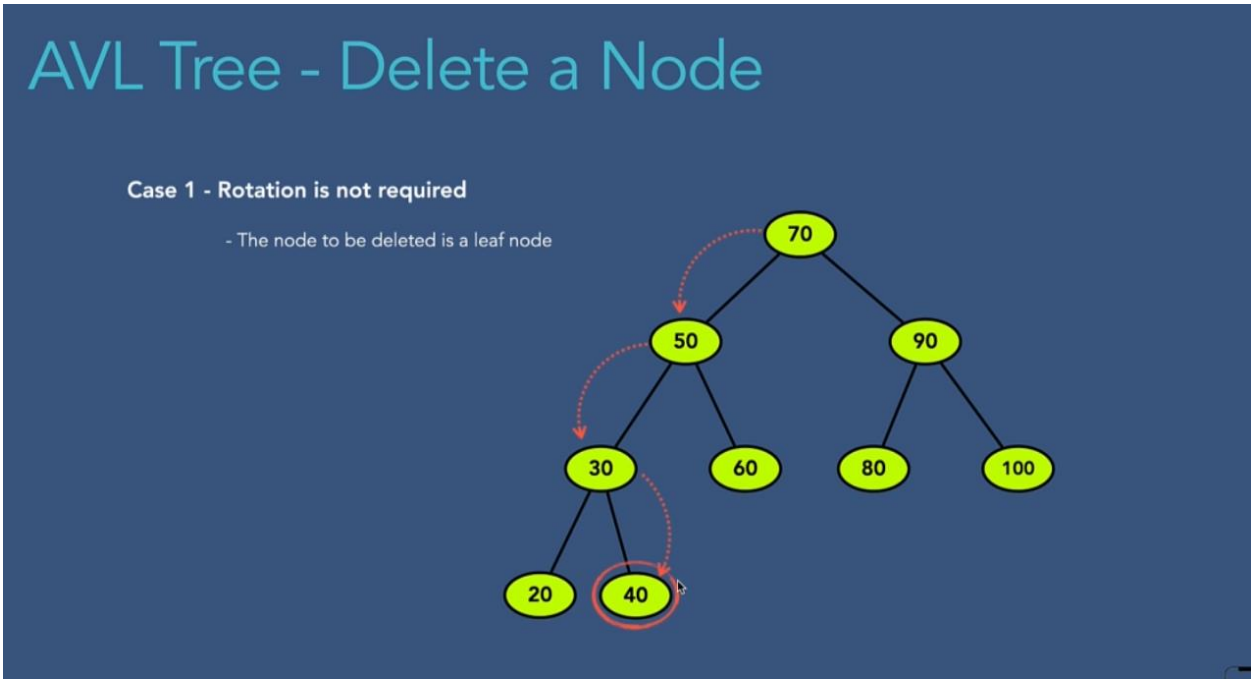
AVL Tree - Delete a Node

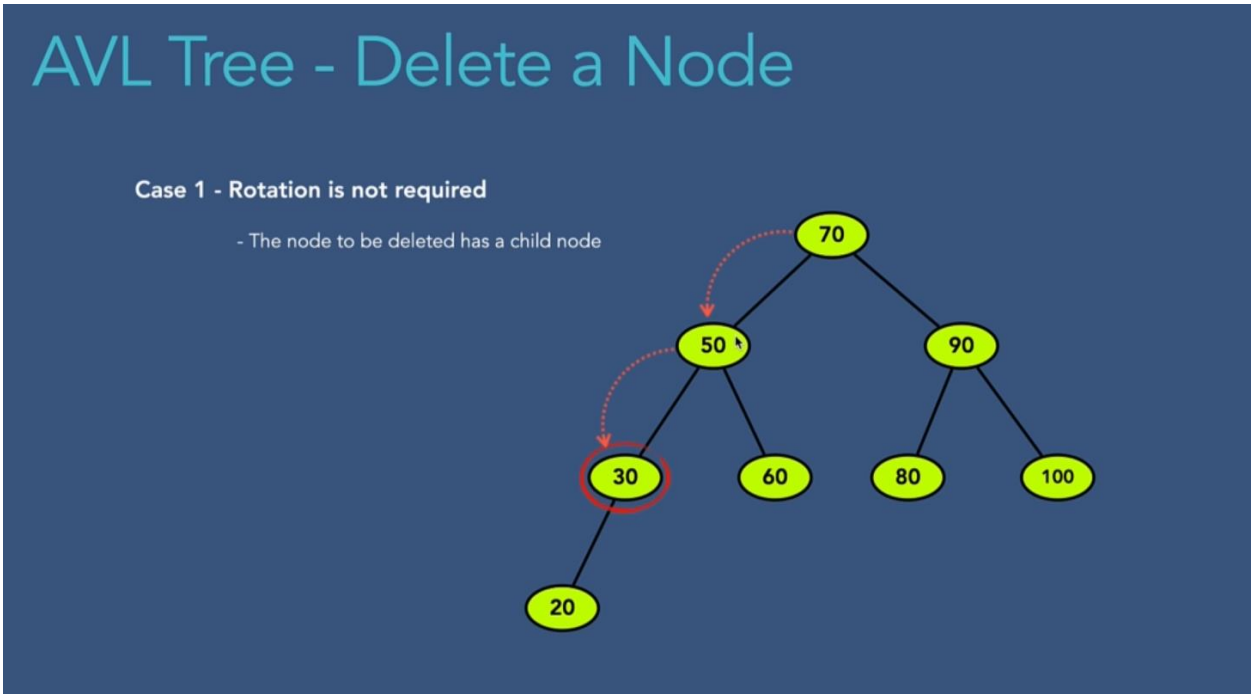
Case 1 - Rotation is not required

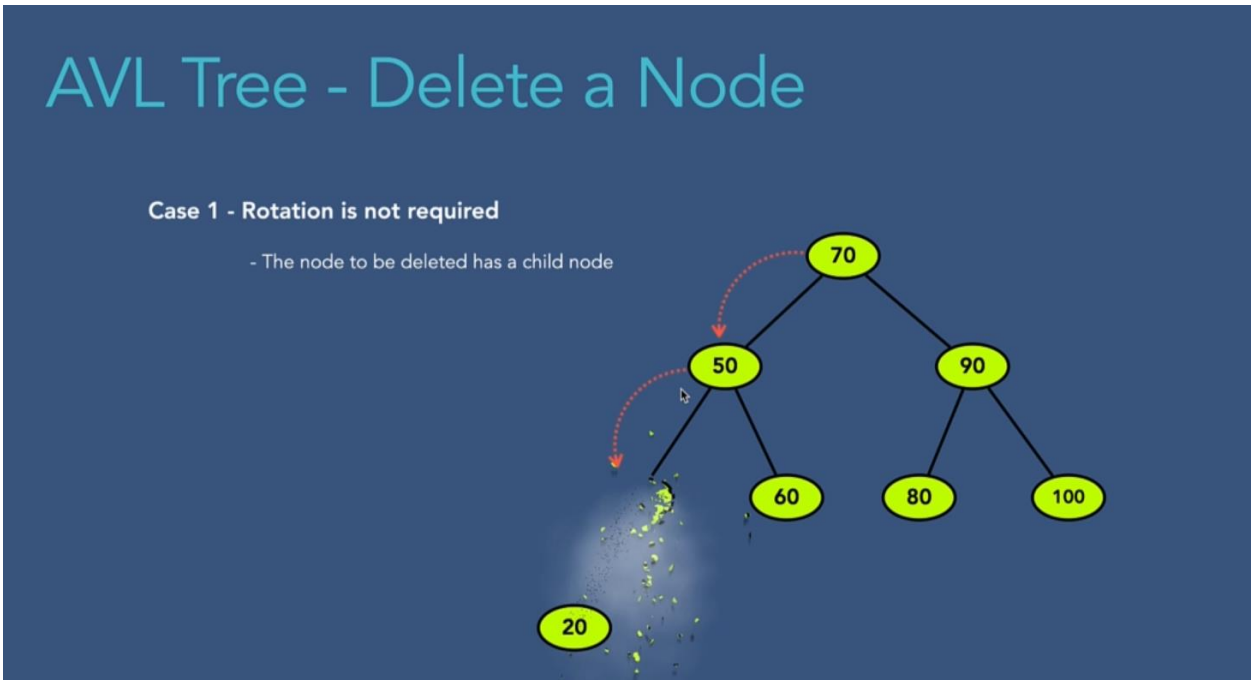
- The node to be deleted is a leaf node
- Node to be deleted has one child
- Node to be deleted has 2 children.



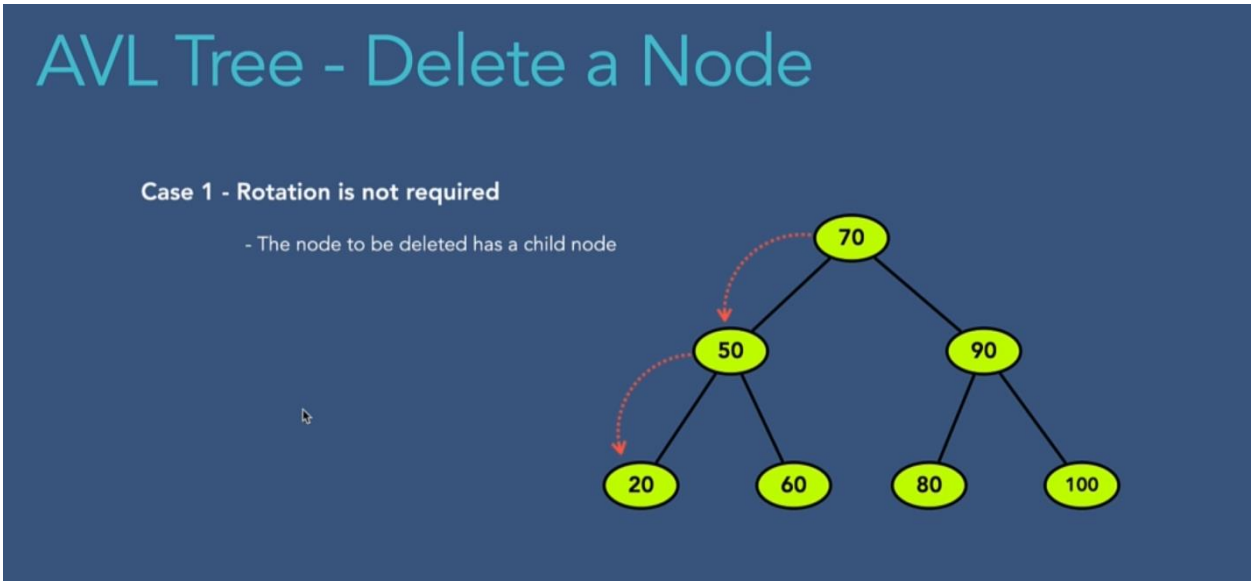
Deletion of 40 from the AVL Tree



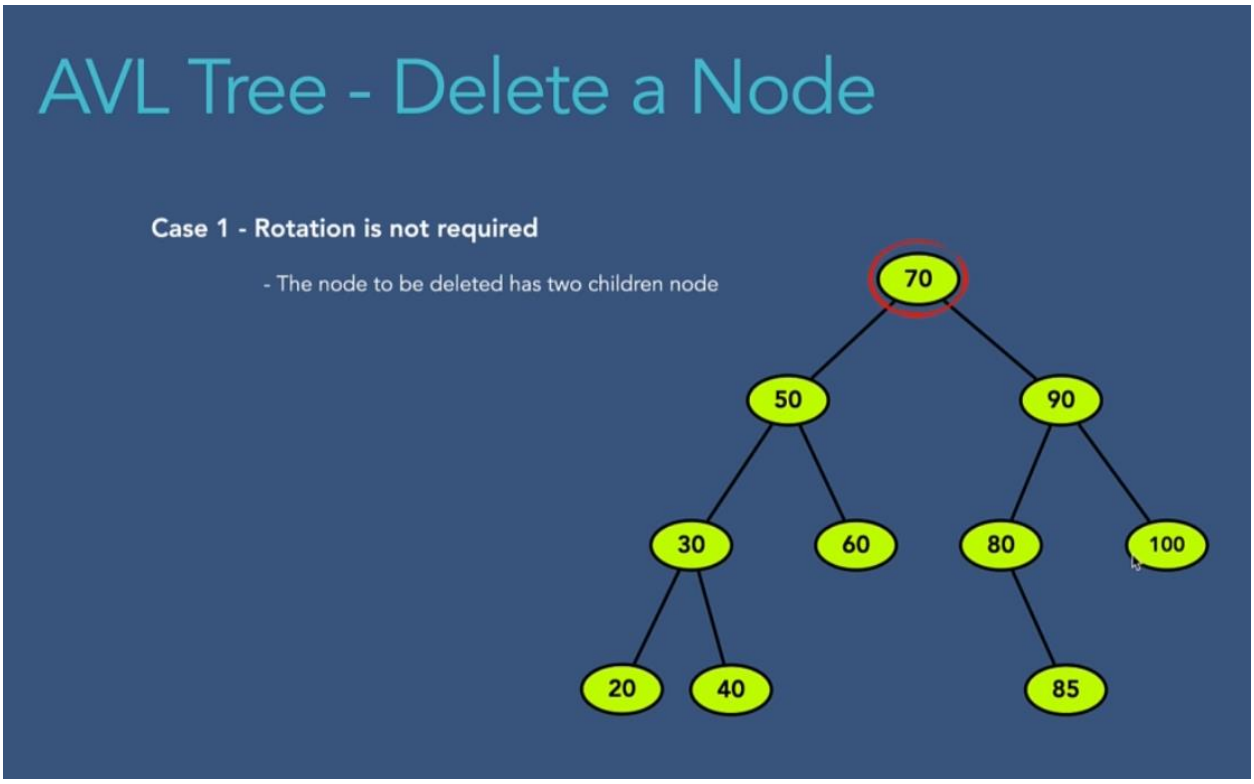




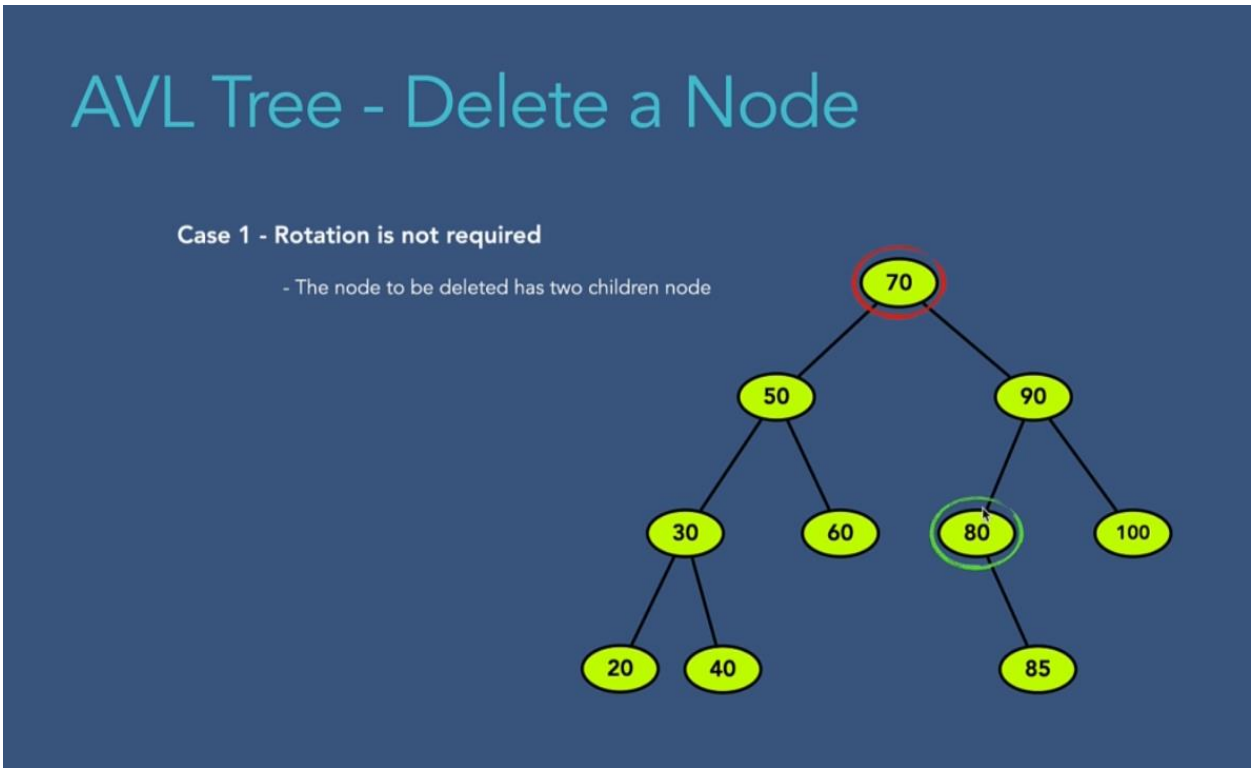
After deletion of node 30



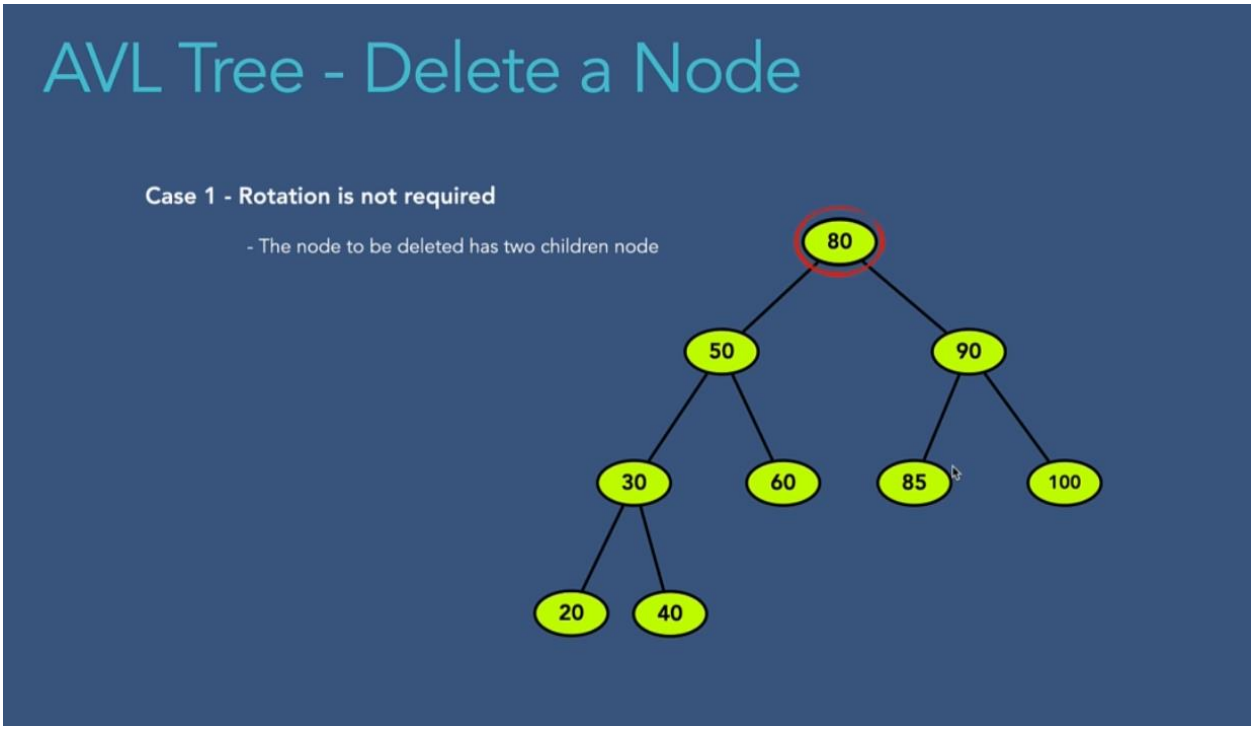
Node to be deleted is 70 which is the right node. And we have to find the successor for the root node which is the minimum element in the right subtree



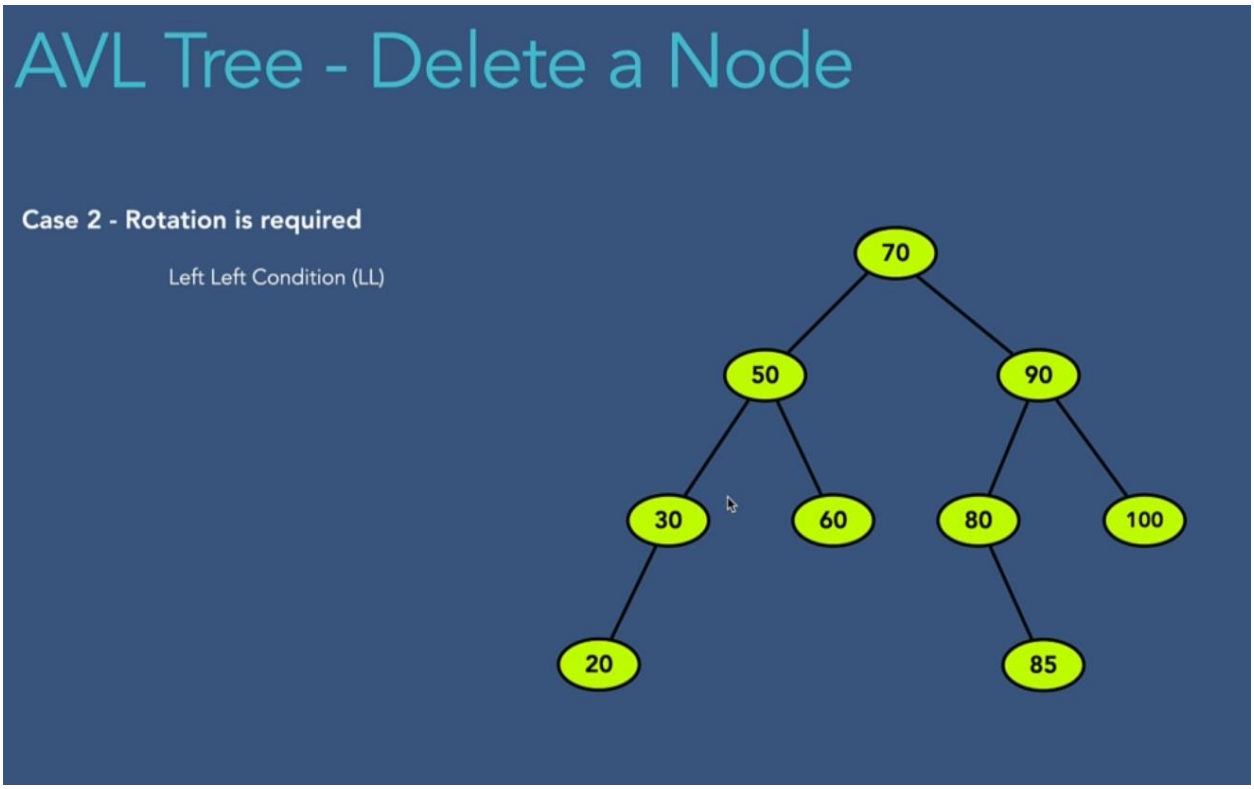
Node 80 is the minimum node in the right sub tree



After the deletion of the root node



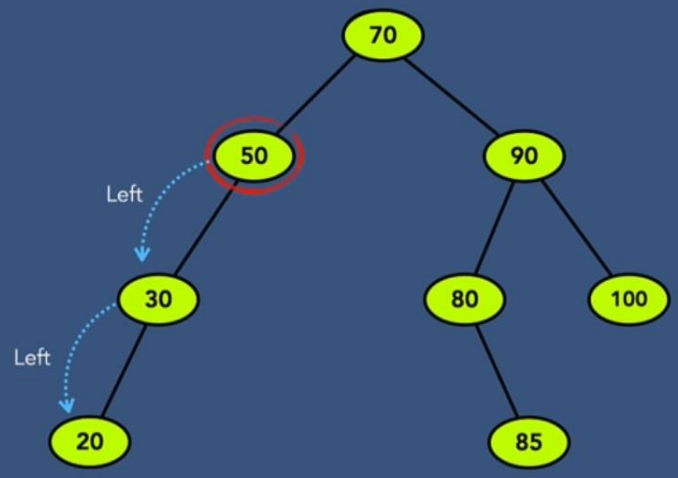
To delete Node 60 from the AVL Tree



AVL Tree - Delete a Node

Case 2 - Rotation is required

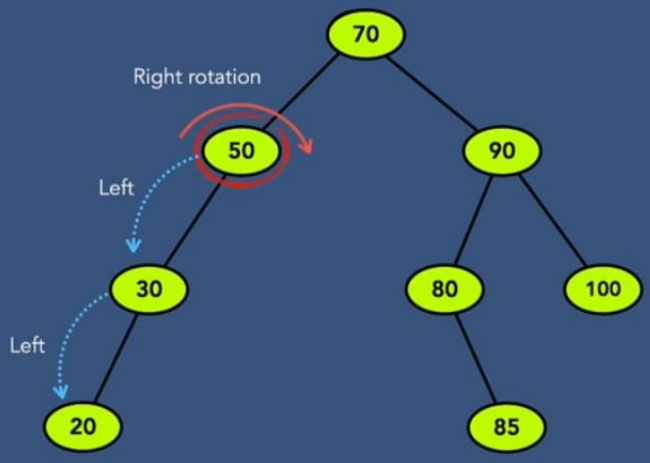
Left Left Condition (LL)



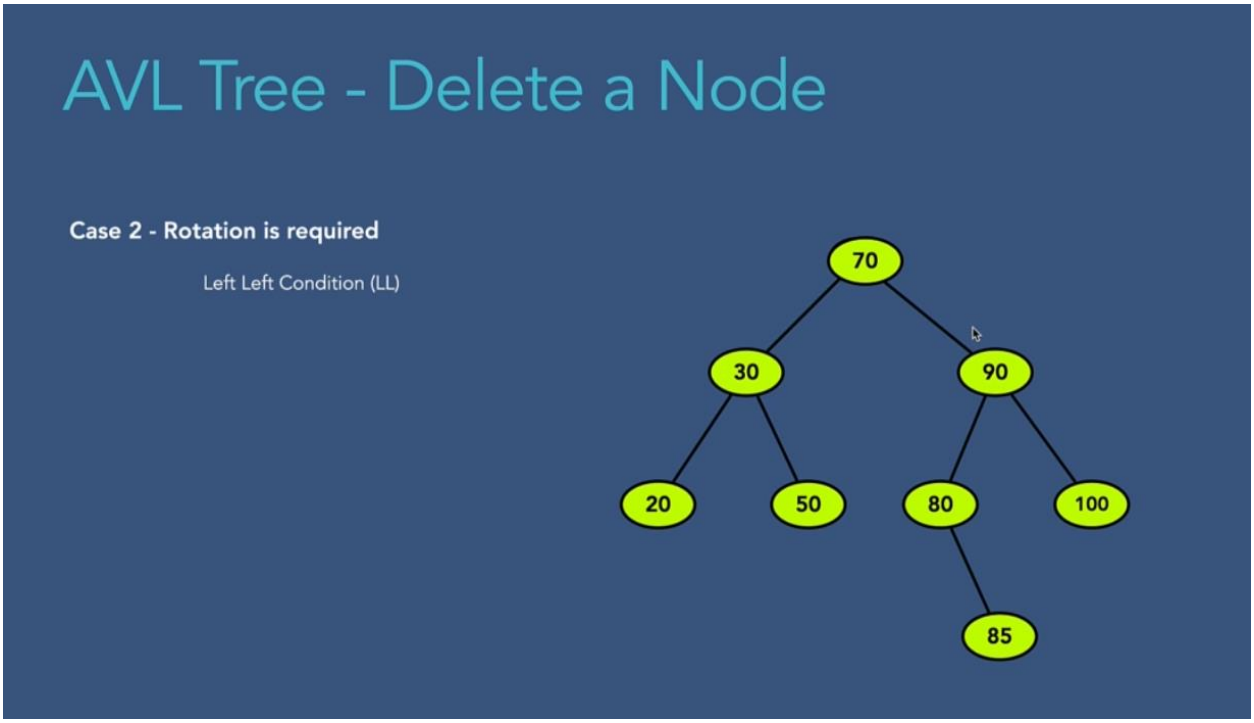
AVL Tree - Delete a Node

Case 2 - Rotation is required

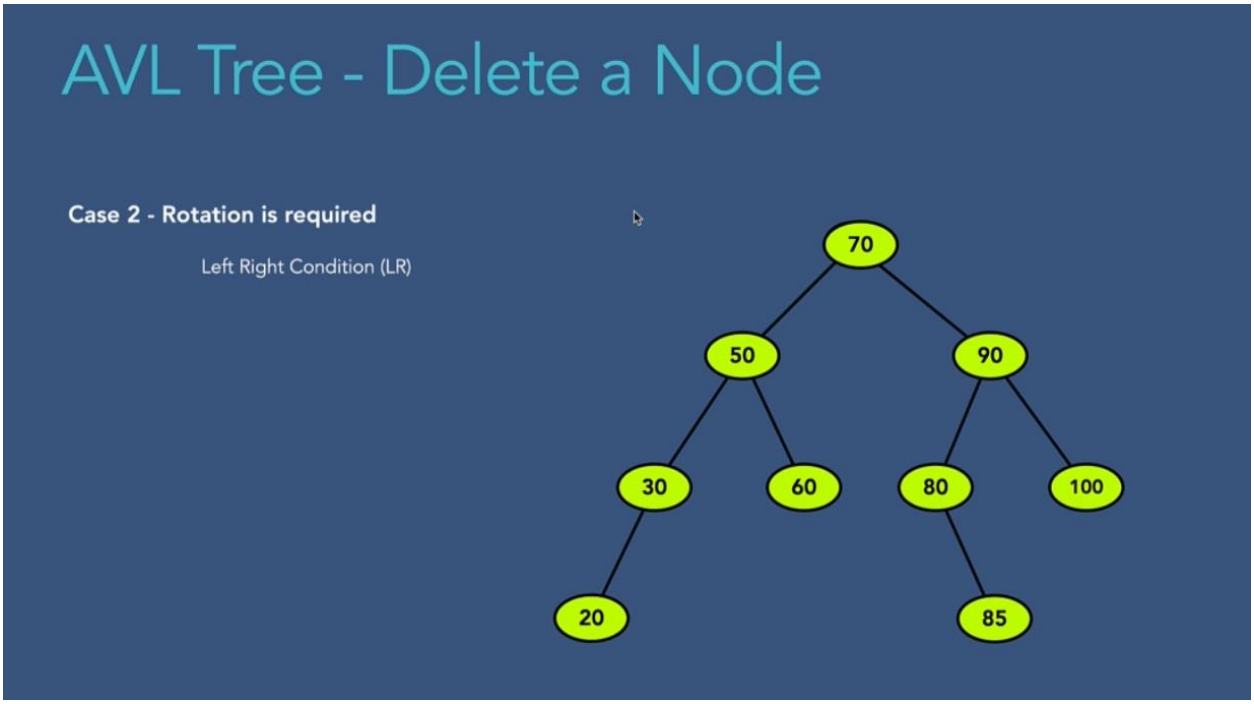
Left Left Condition (LL)



After right rotation



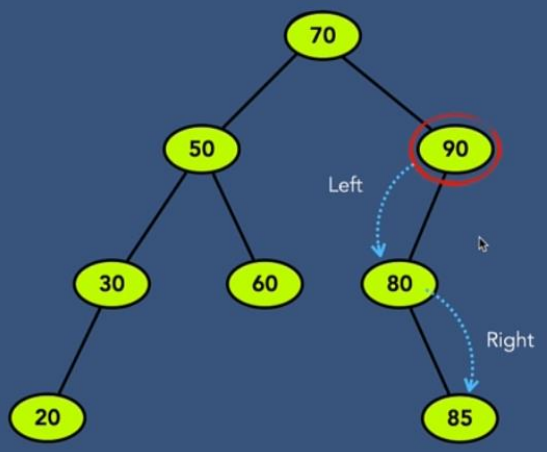
To delete Node 100 from the AVL Tree



AVL Tree - Delete a Node

Case 2 - Rotation is required

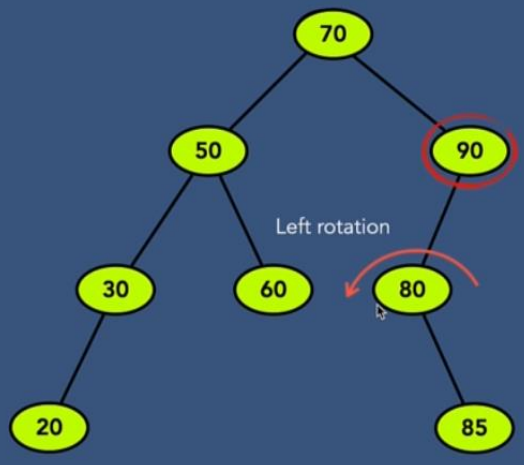
Left Right Condition (LR)



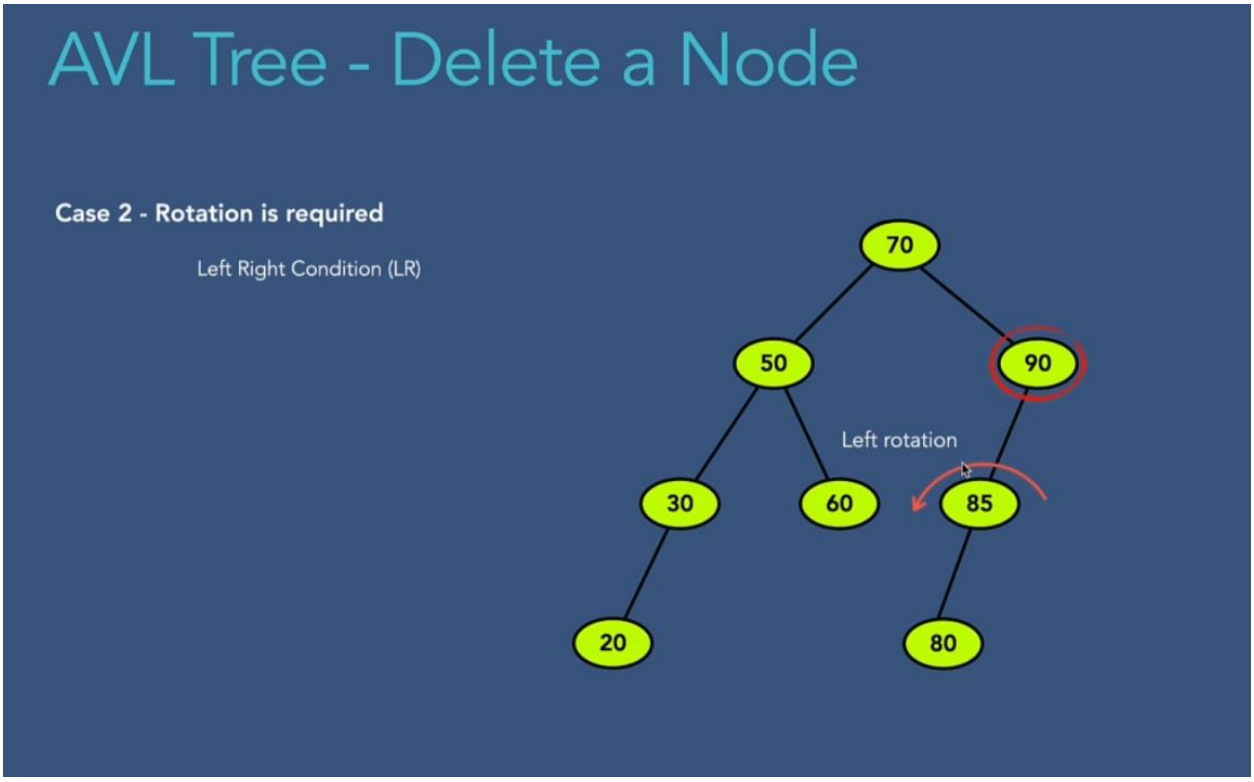
AVL Tree - Delete a Node

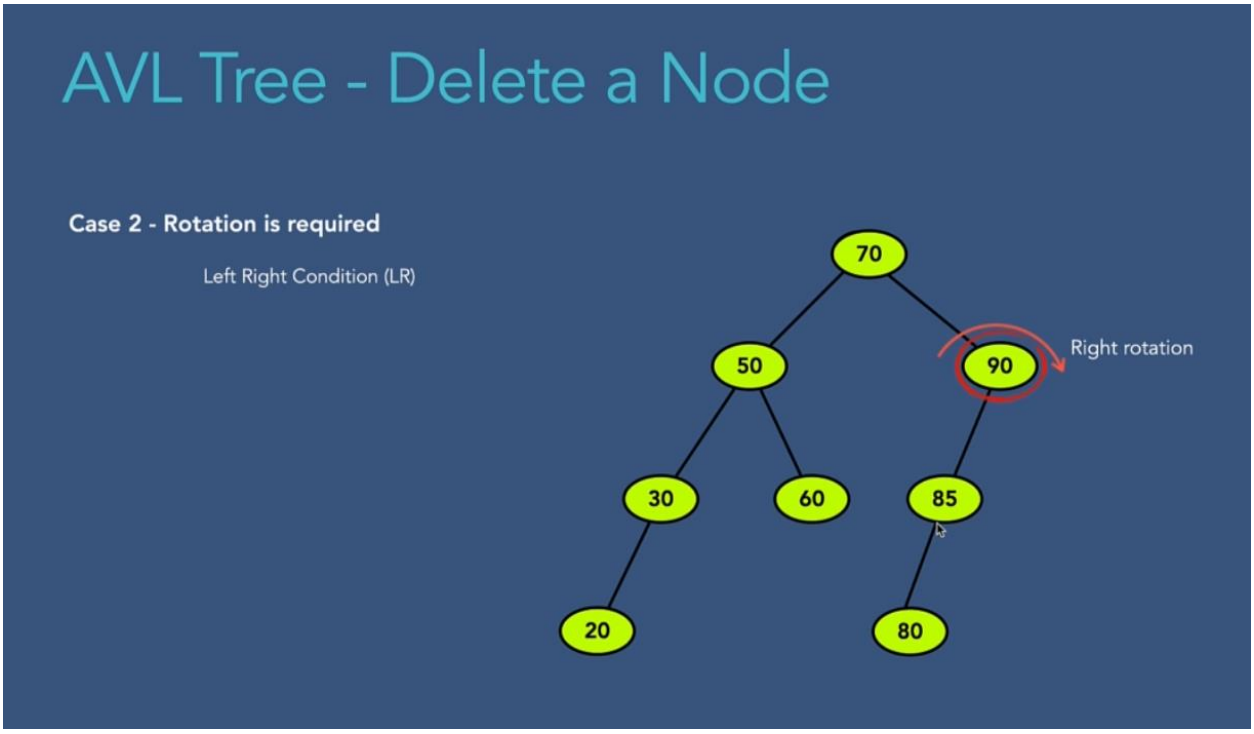
Case 2 - Rotation is required

Left Right Condition (LR)

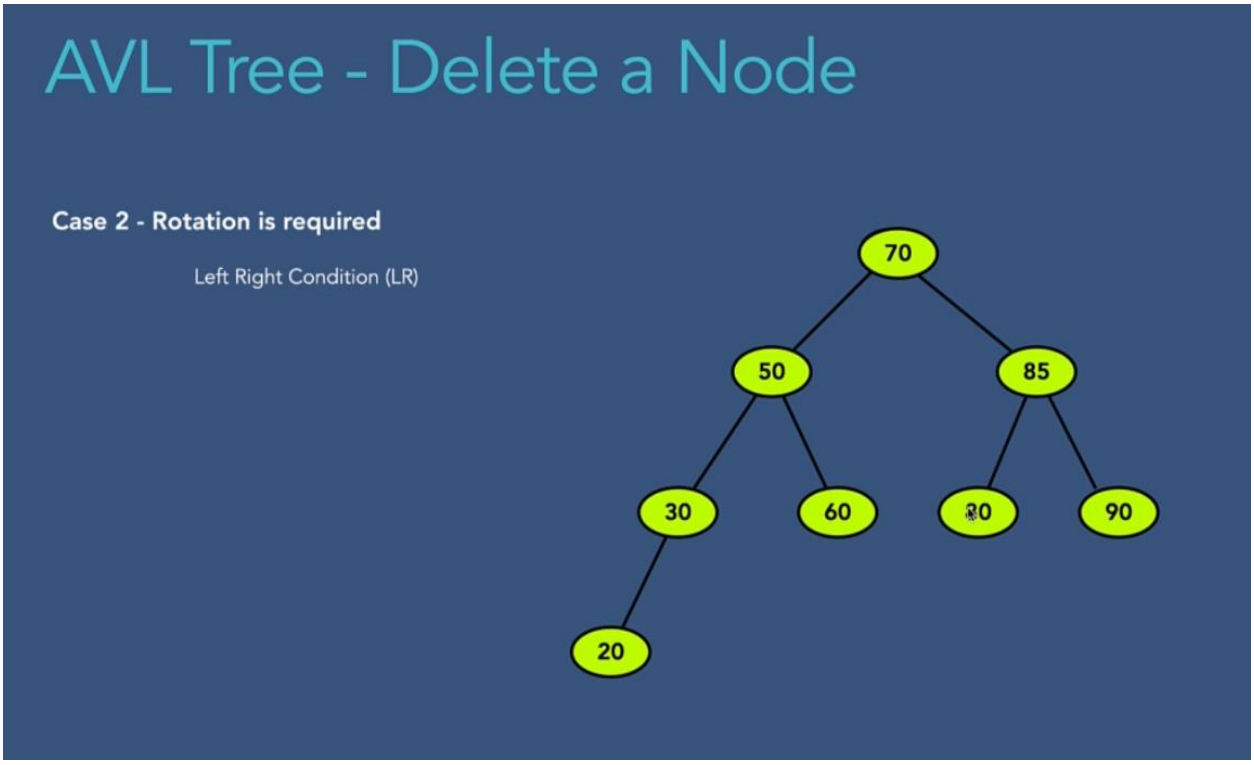


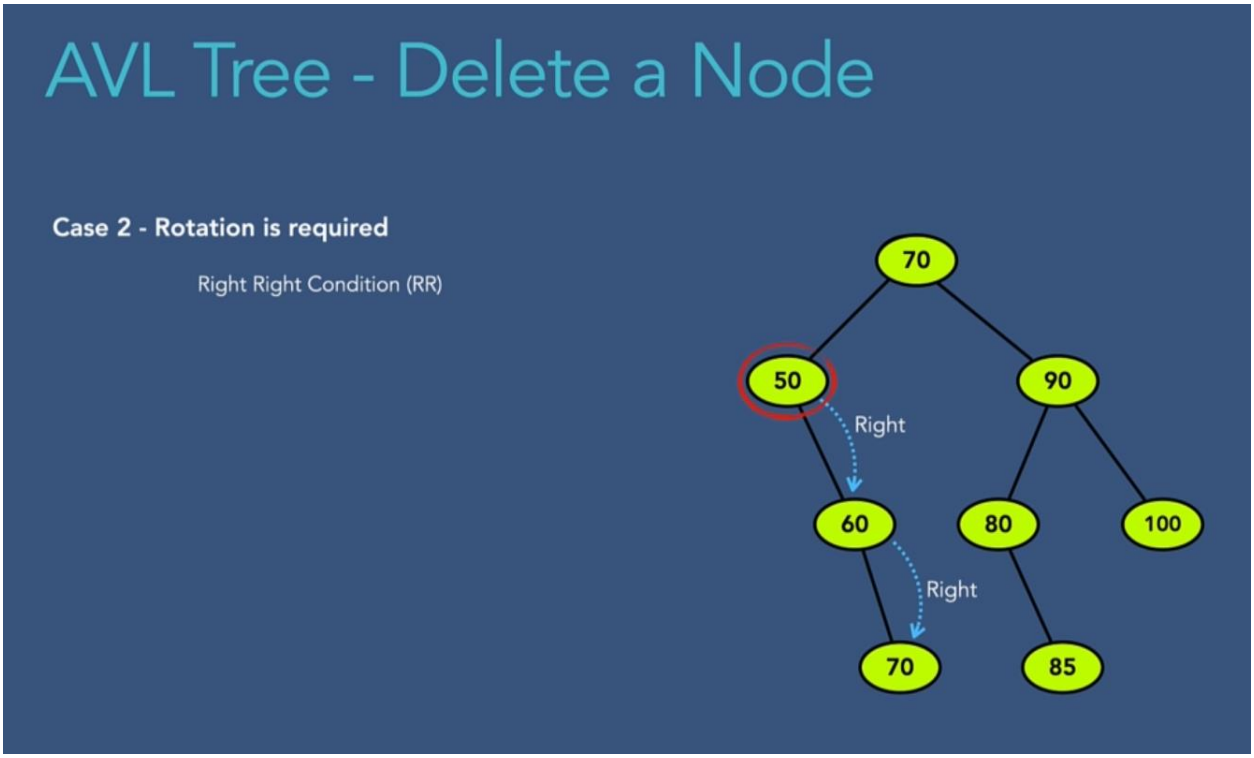
After left rotation

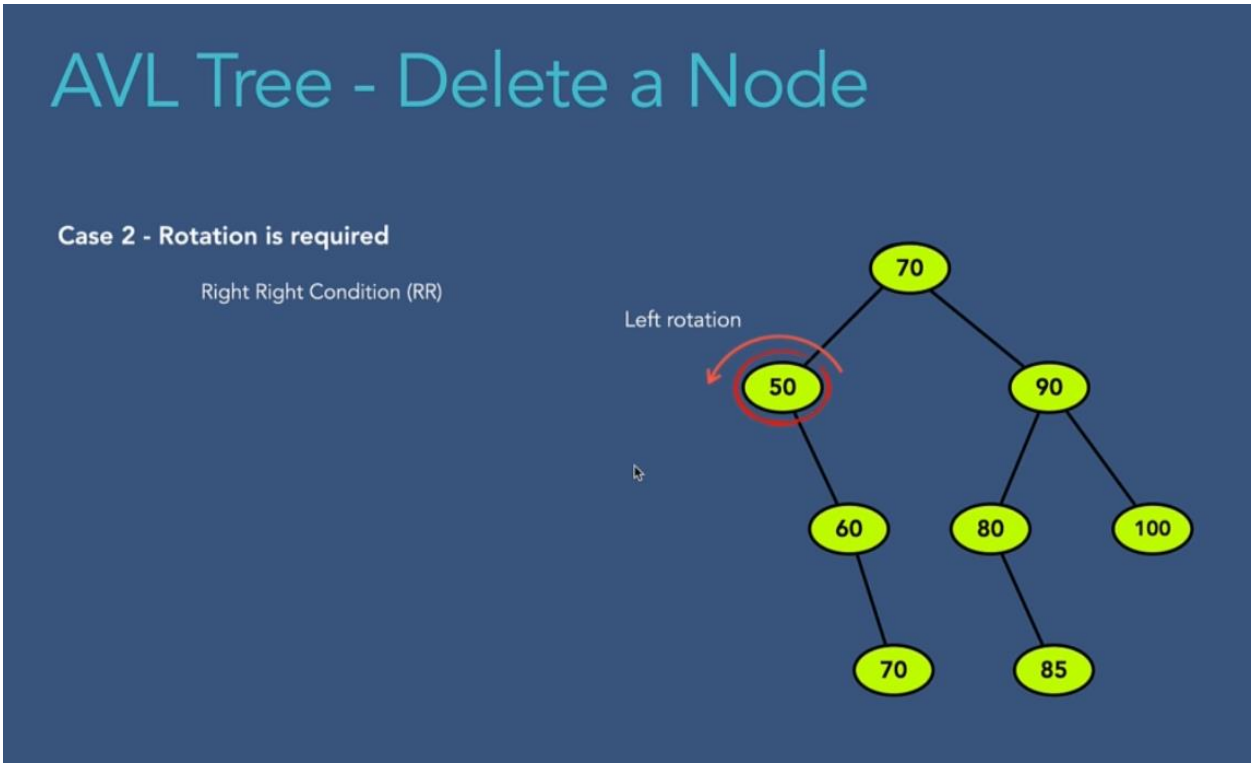




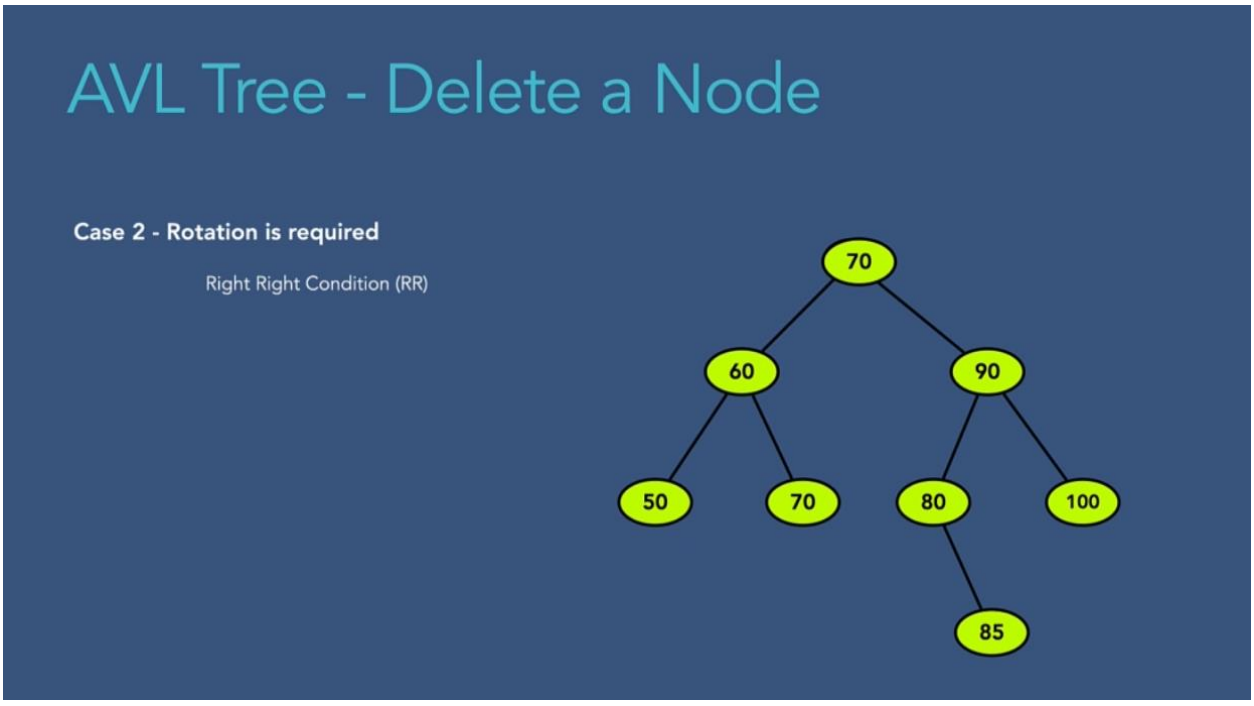
After successful deletion



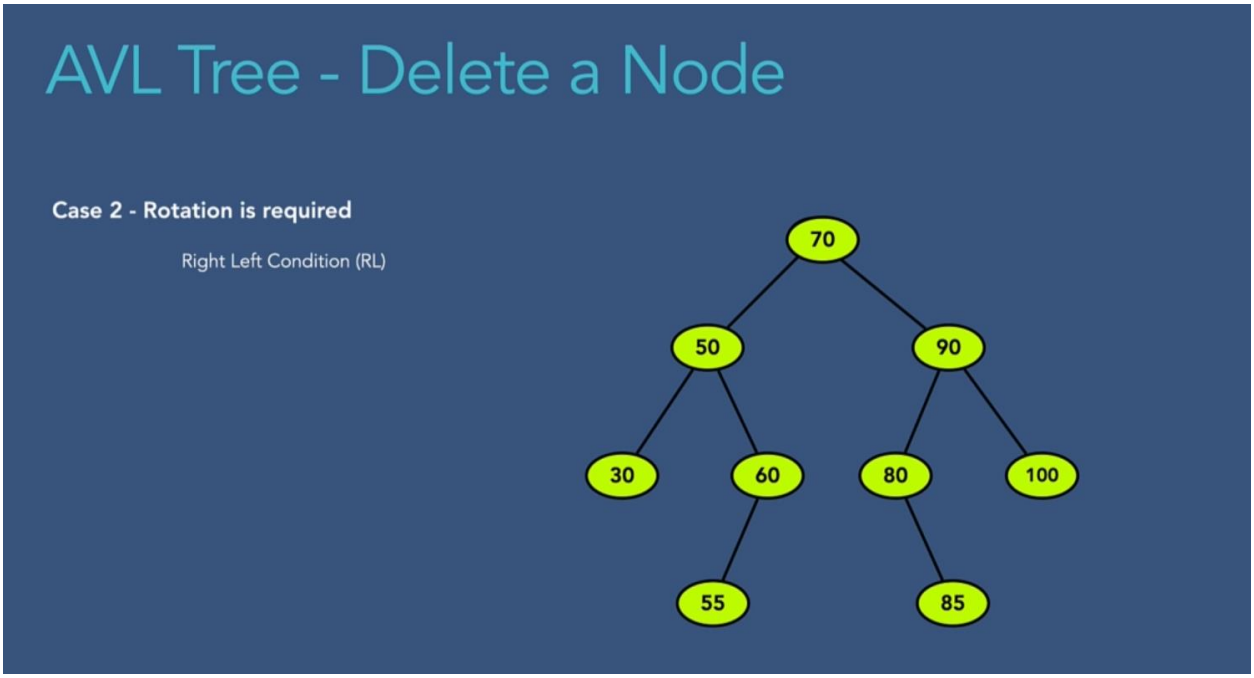


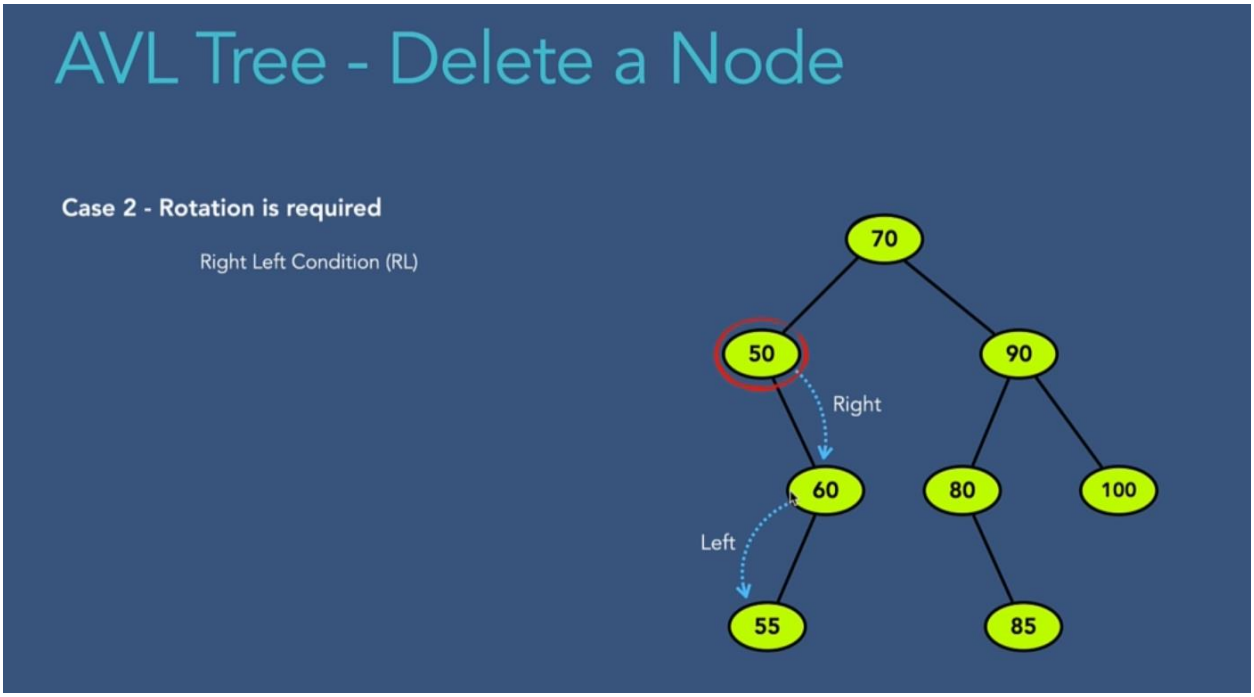


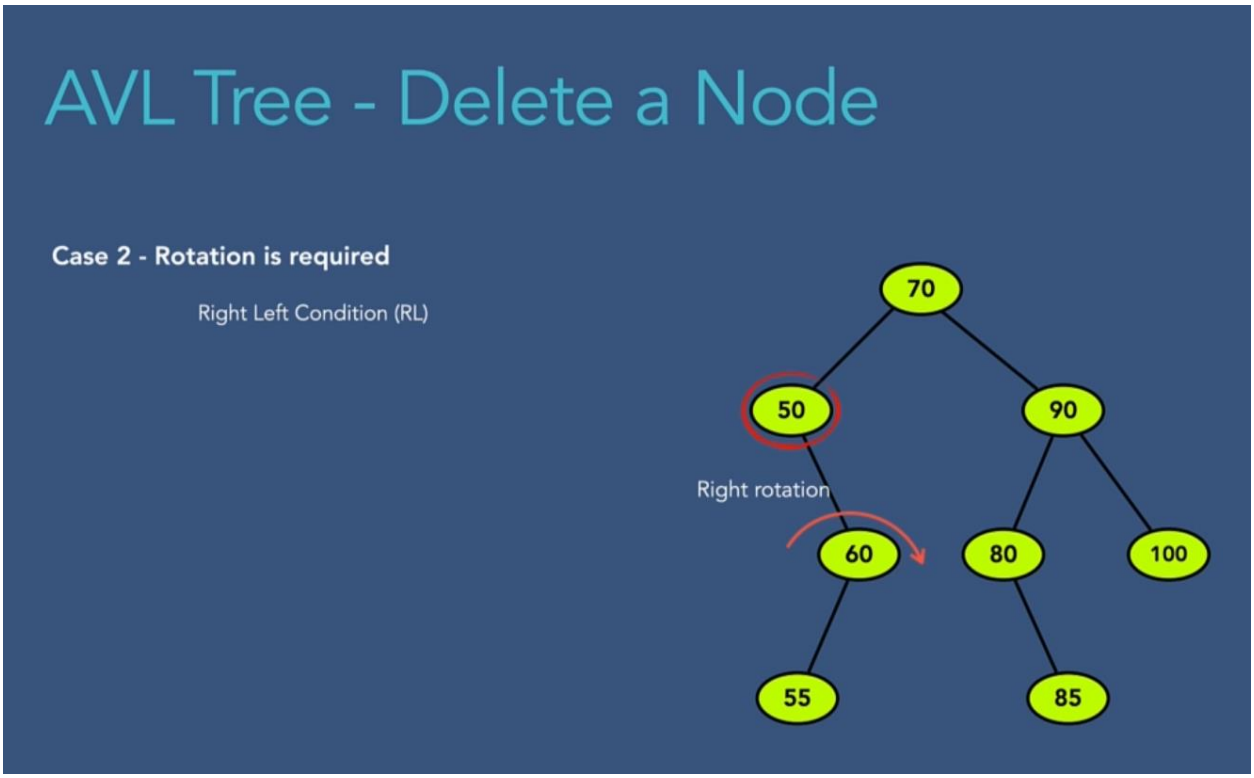
After left rotation



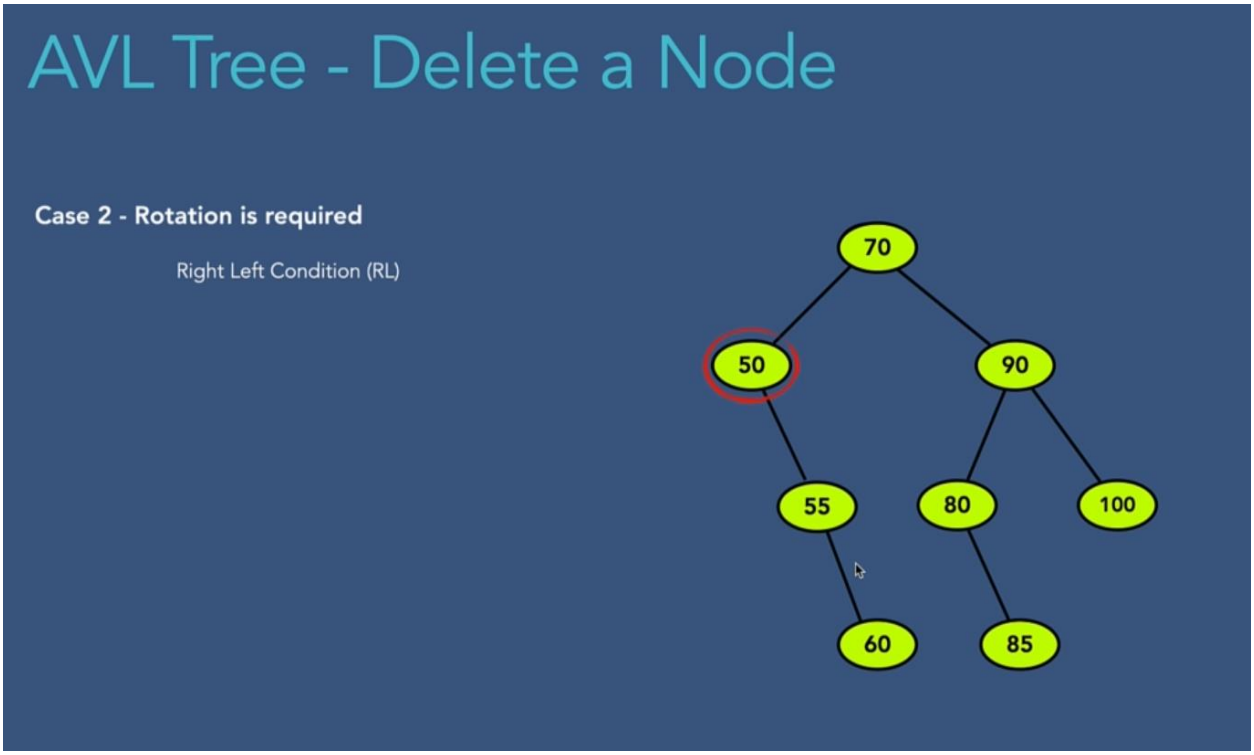
Again deletion of Node 30 from the AVL Tree

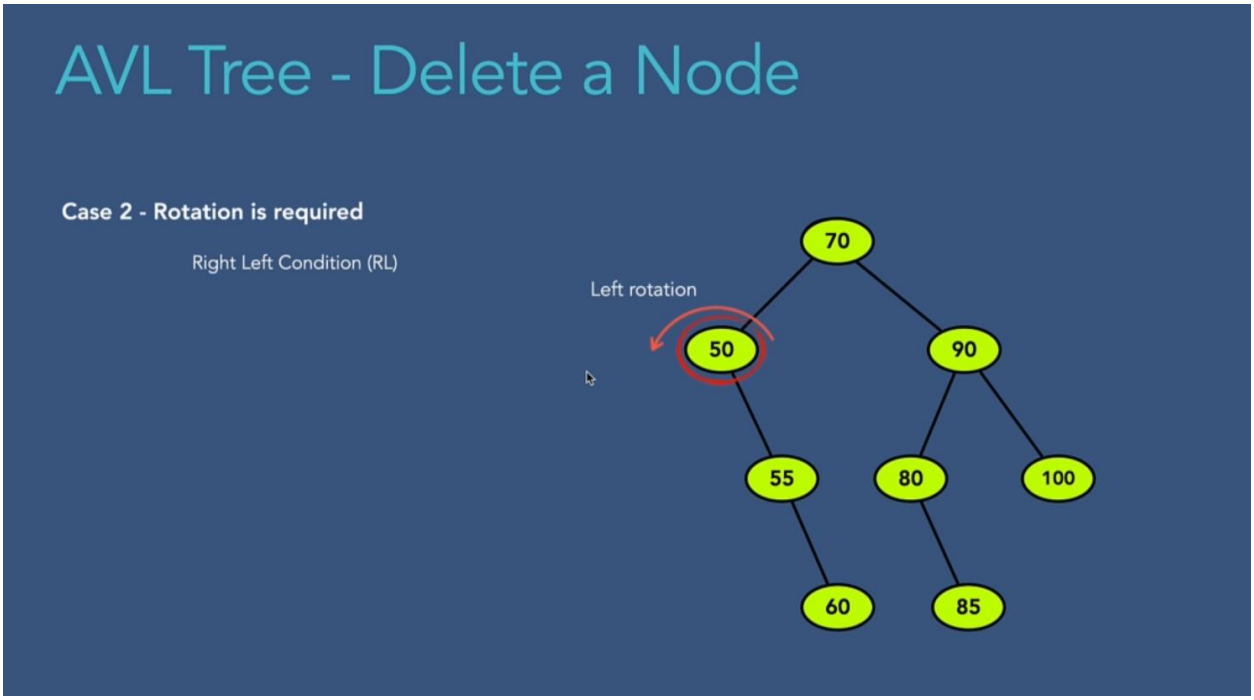




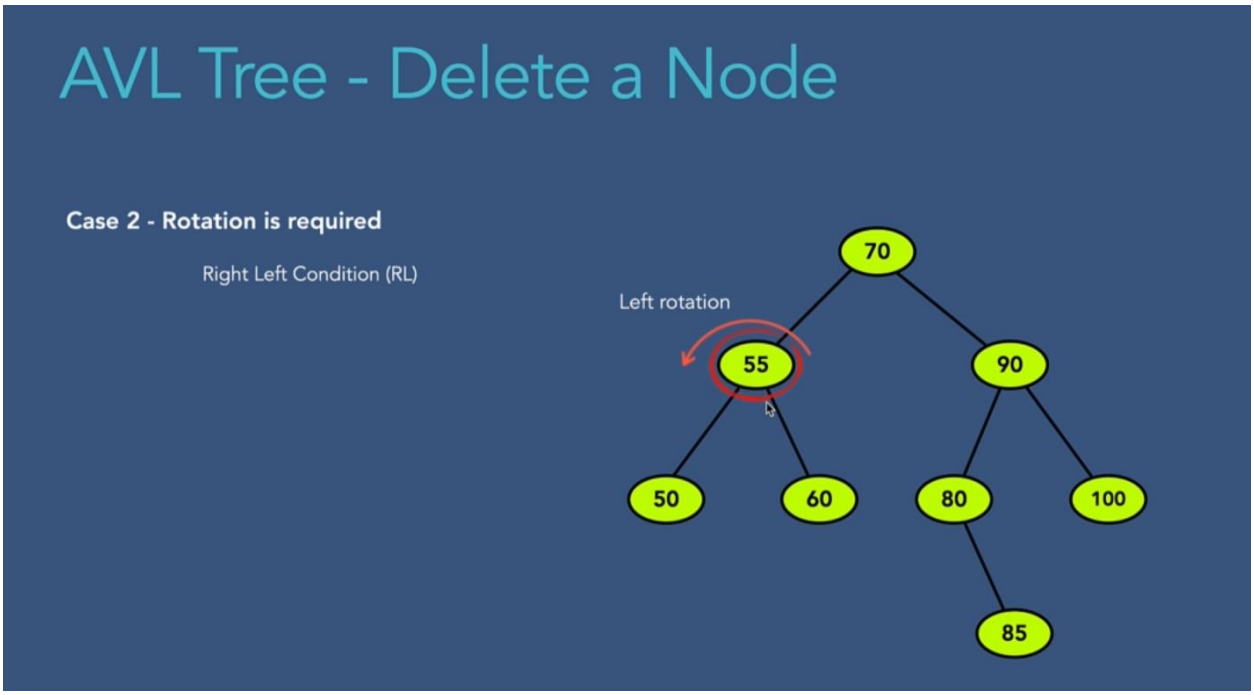


After right rotation

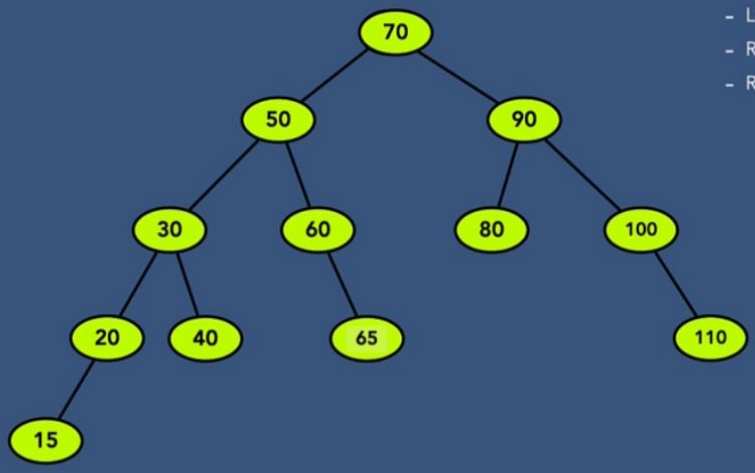




After left rotation



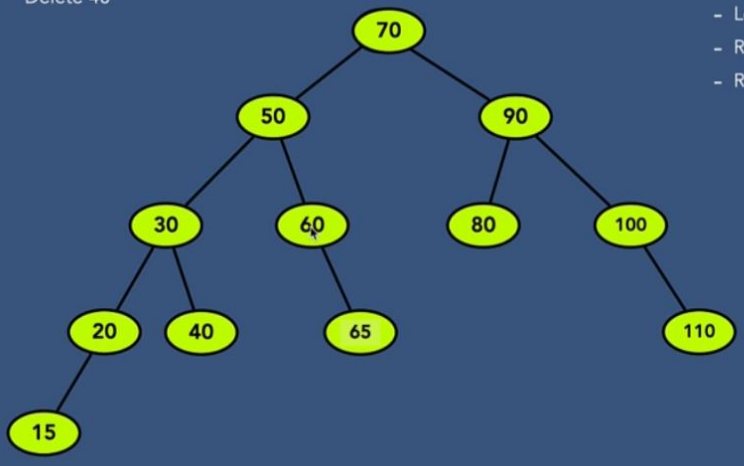
AVL Tree - Delete a Node (all together)



- Left Left Condition (LL)
- Left Right Condition (LR)
- Right Right Condition (RR)
- Right Left Condition (RL)

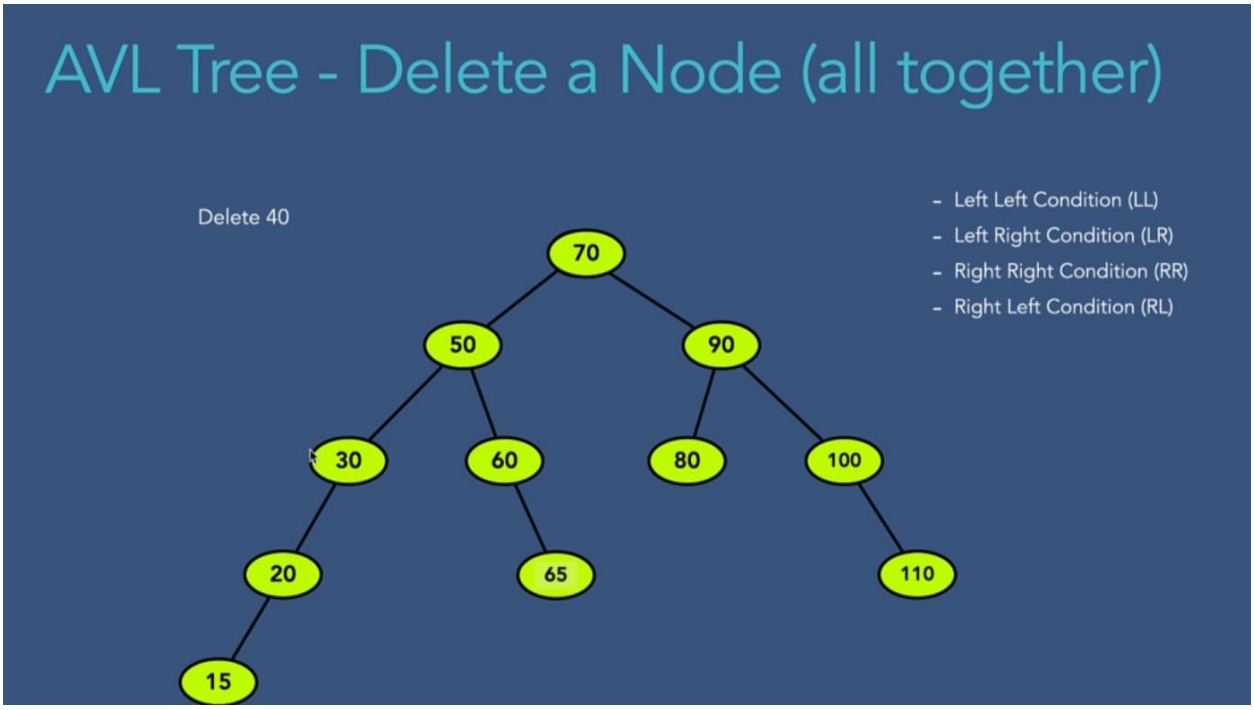
AVL Tree - Delete a Node (all together)

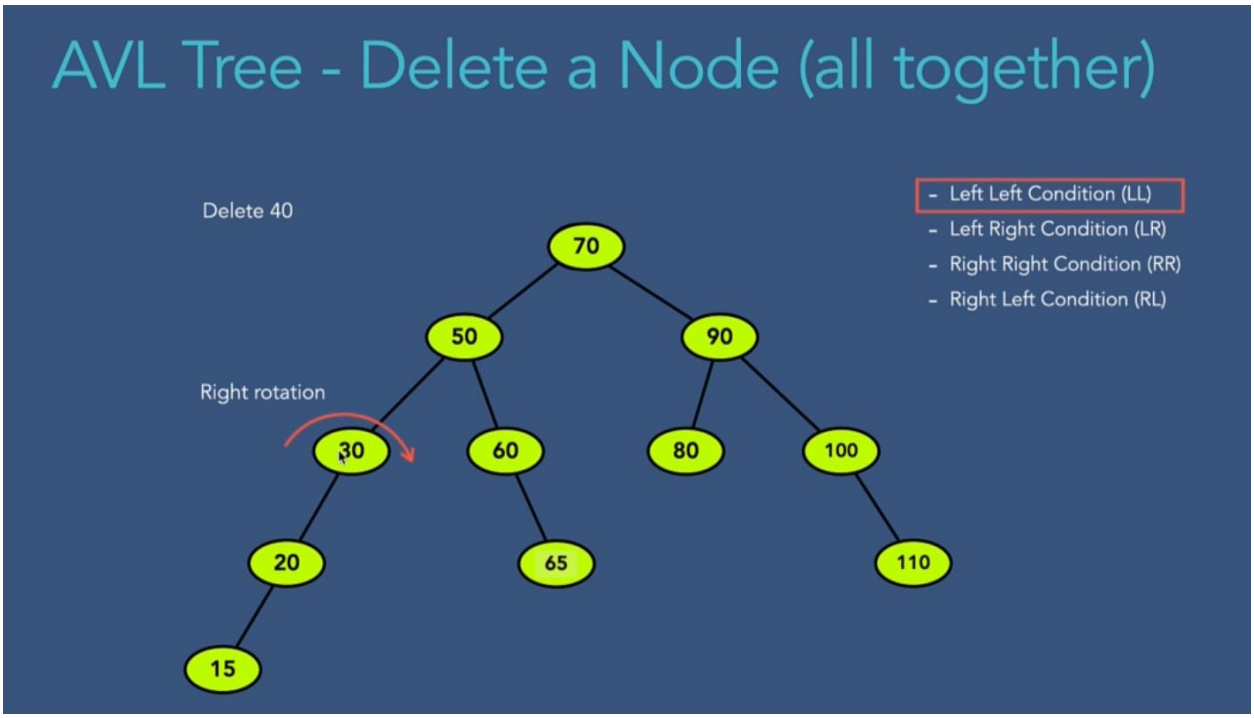
Delete 40



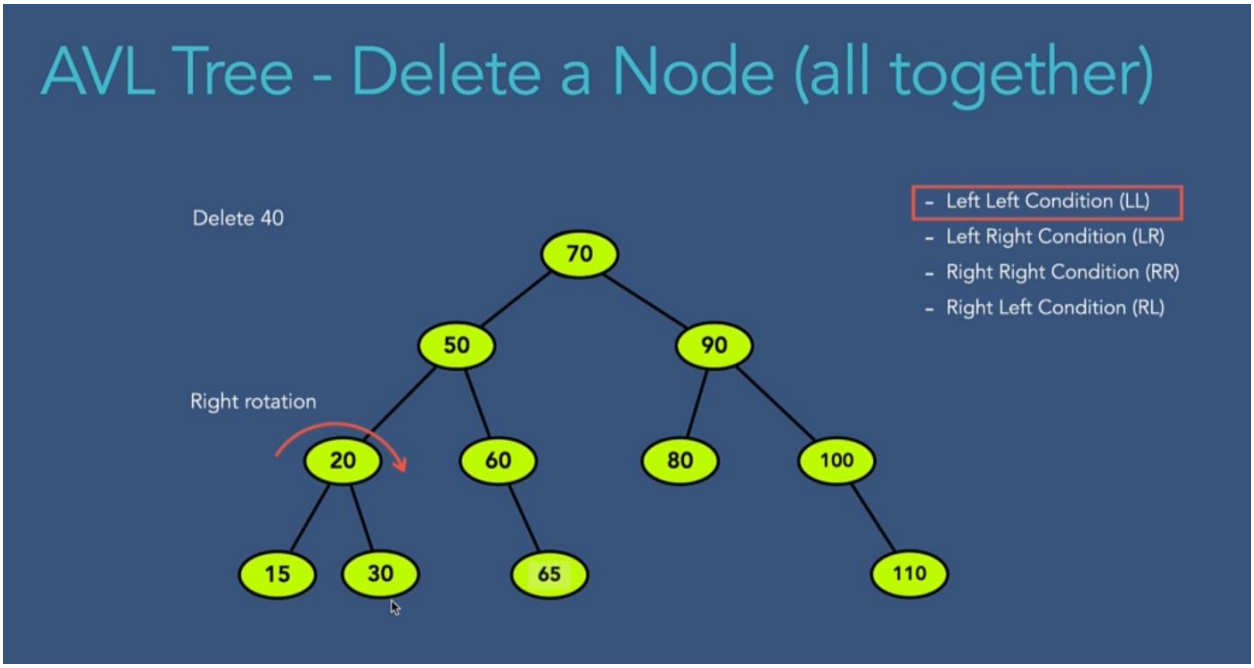
- Left Left Condition (LL)
- Left Right Condition (LR)
- Right Right Condition (RR)
- Right Left Condition (RL)

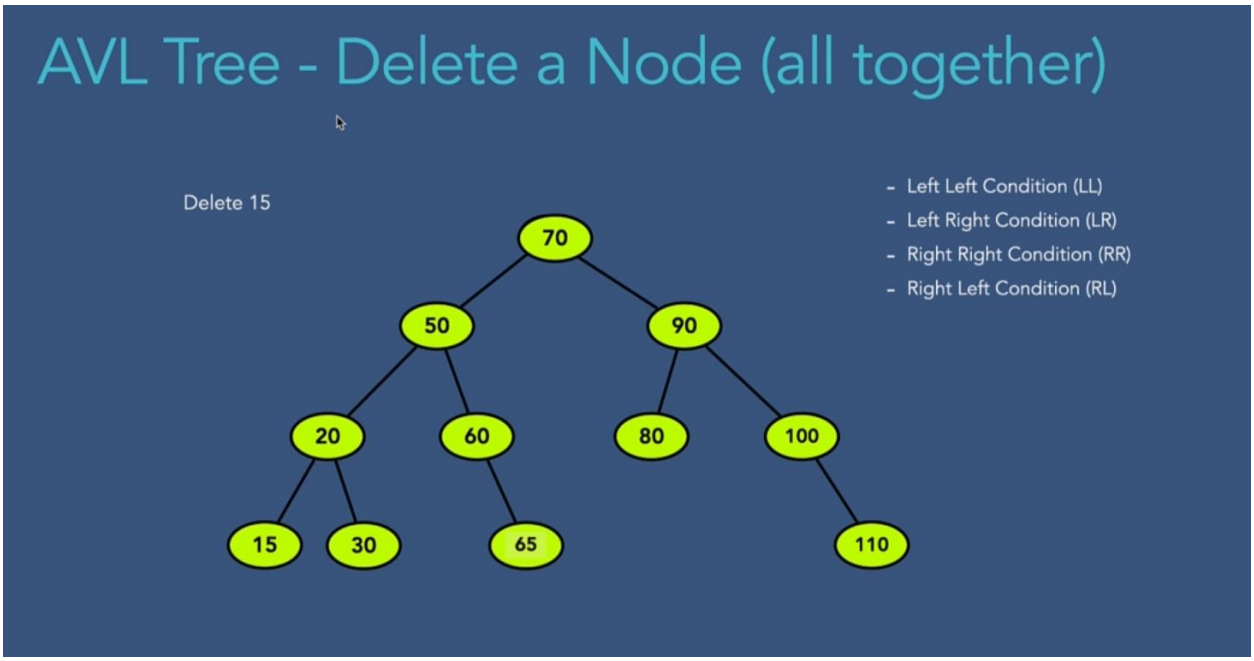
Deleted Node 40 from the AVL Tree



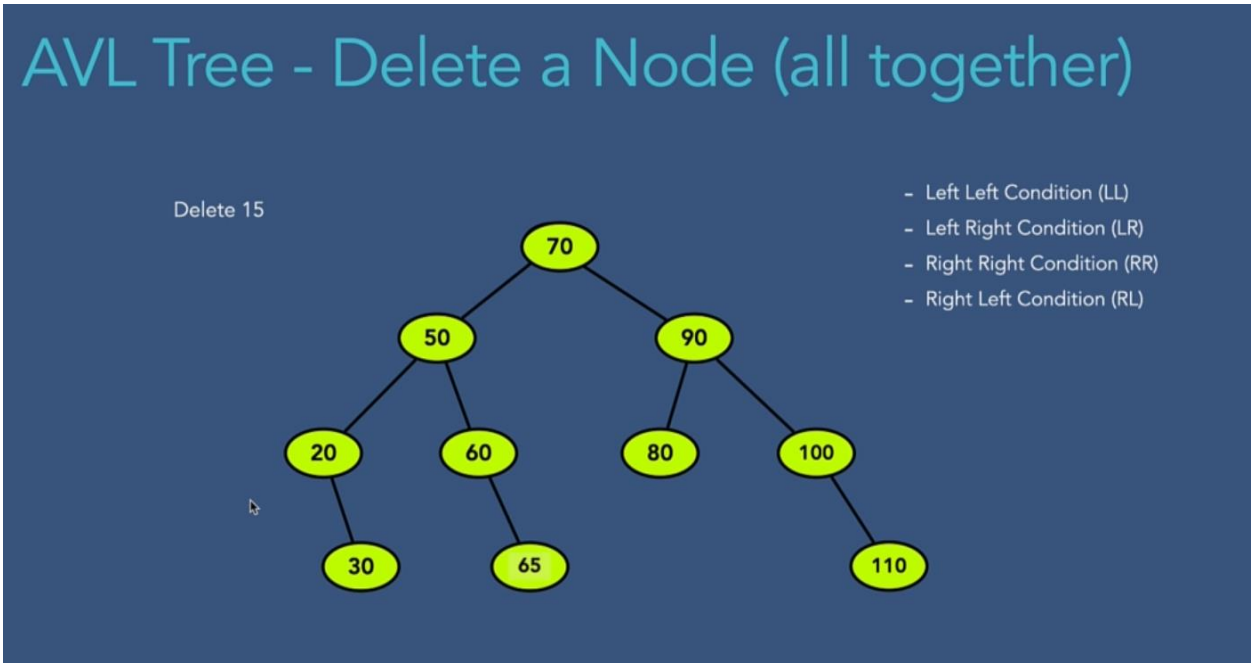


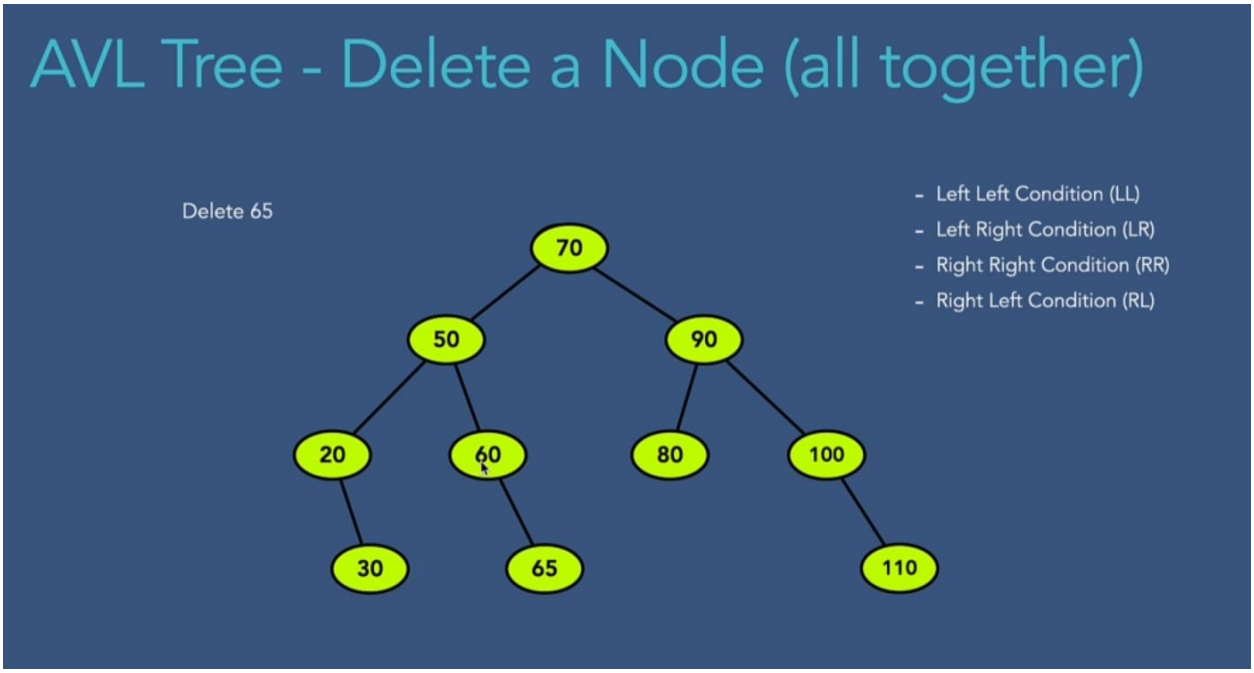
After right rotation



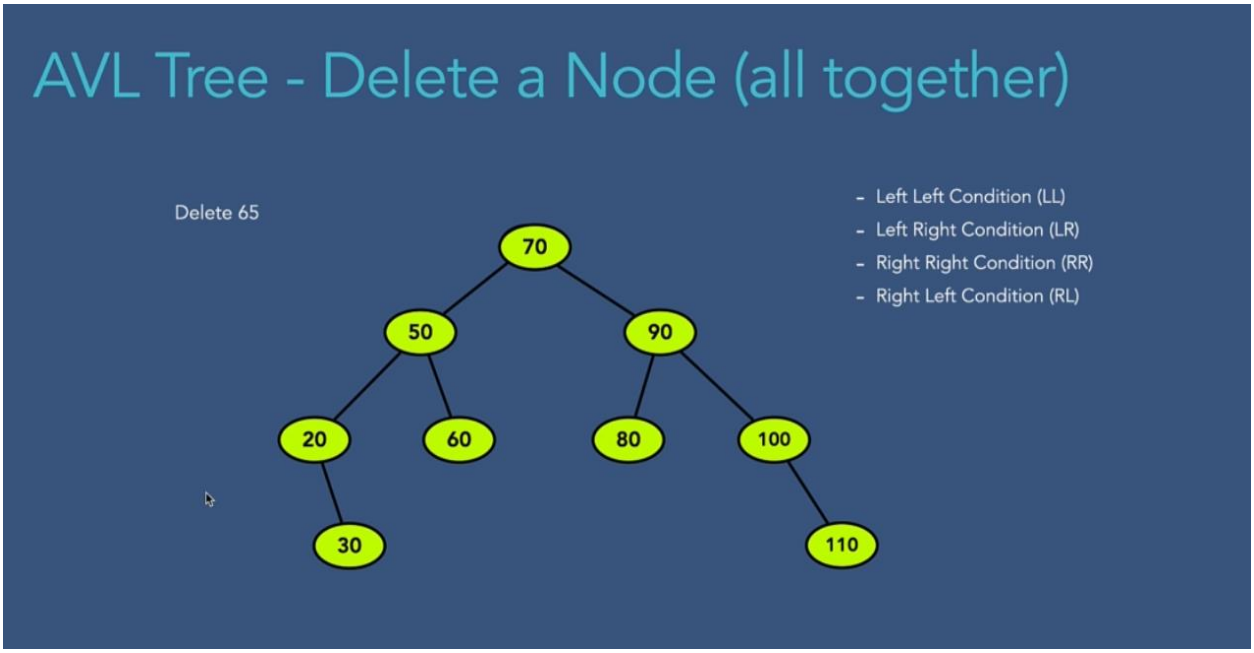


Deleted Node 15 from the AVL Tree → No rotation is required in this case

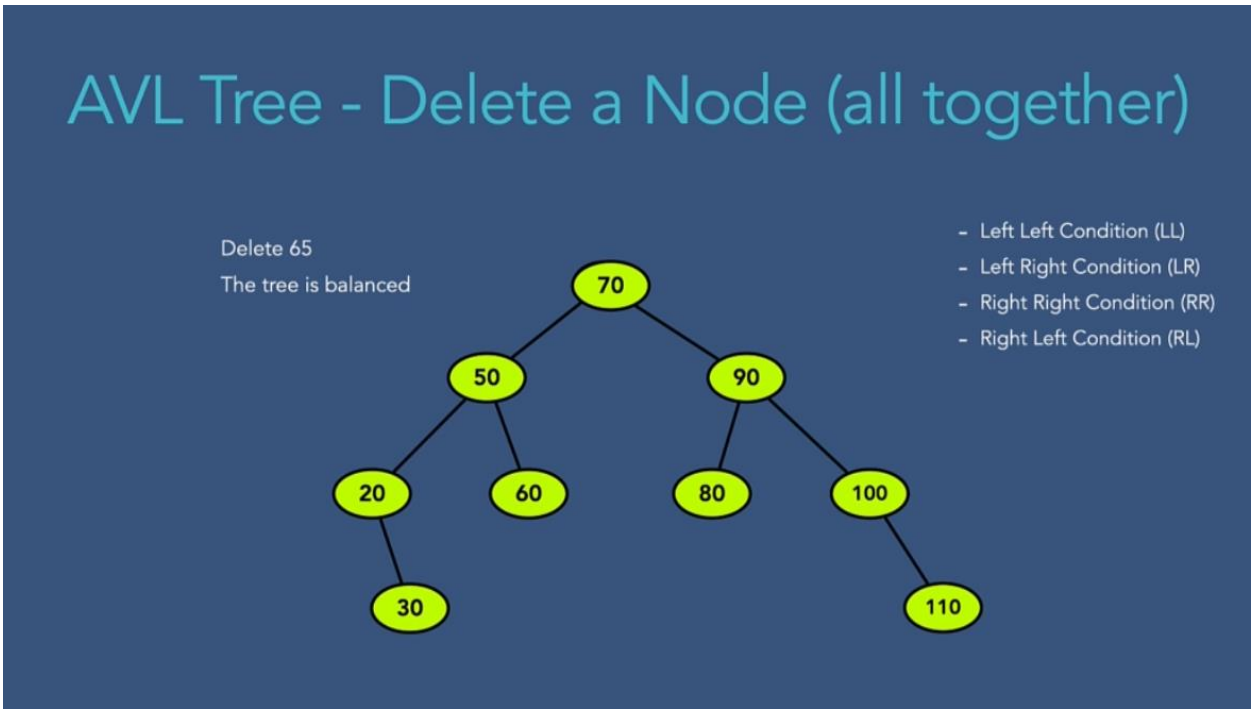


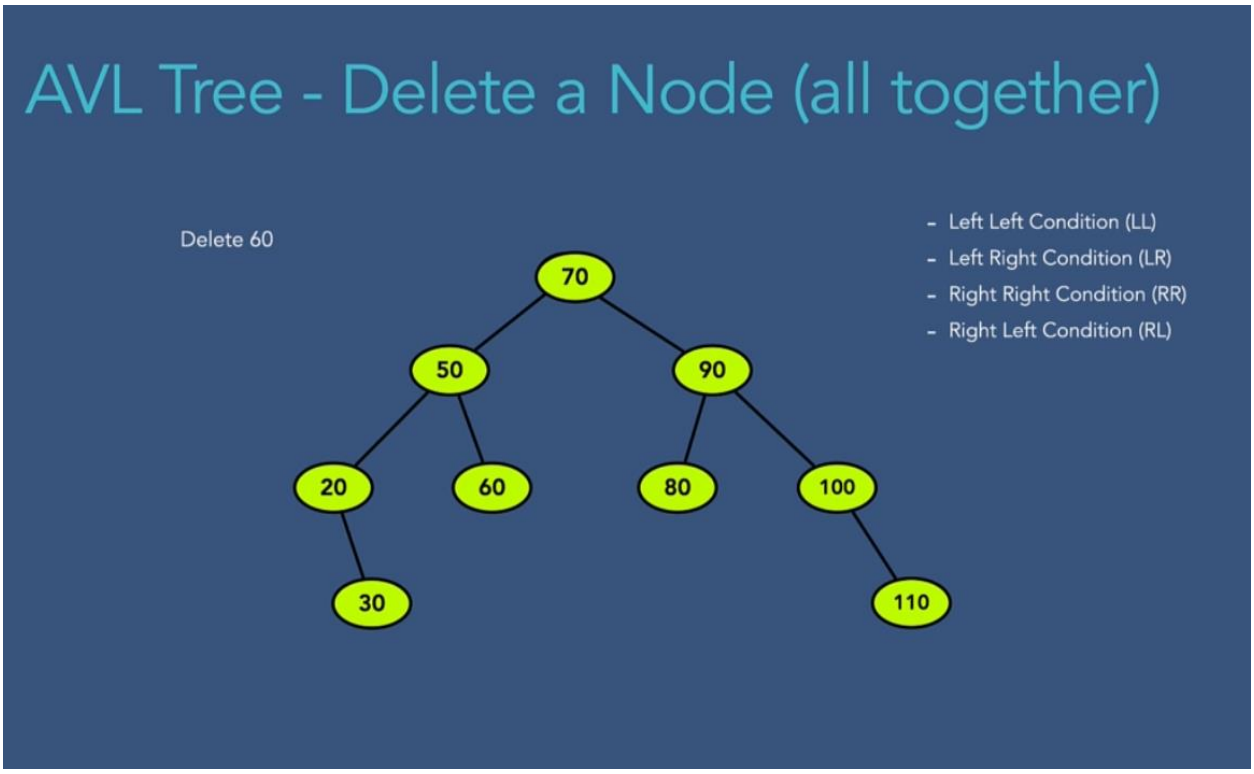


Deleted Node 65 from the AVL Tree

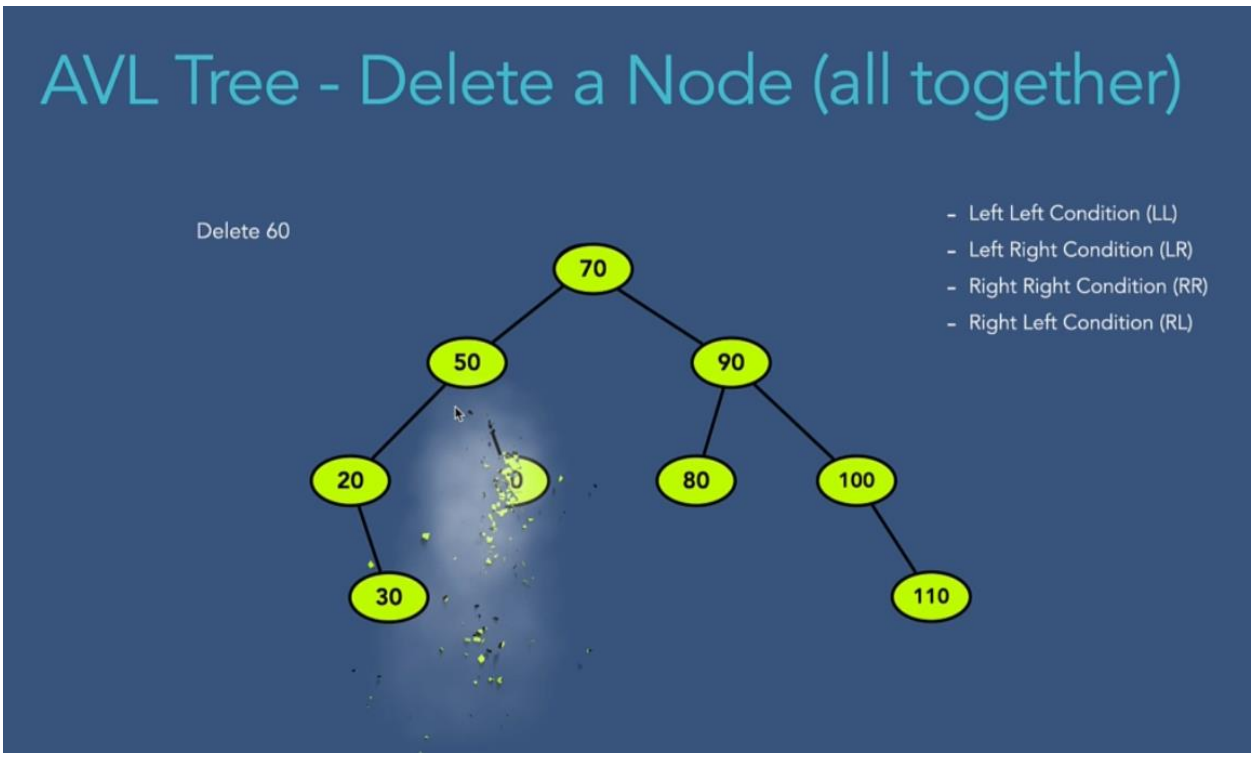


AVL Tree is balanced so no rotation is required





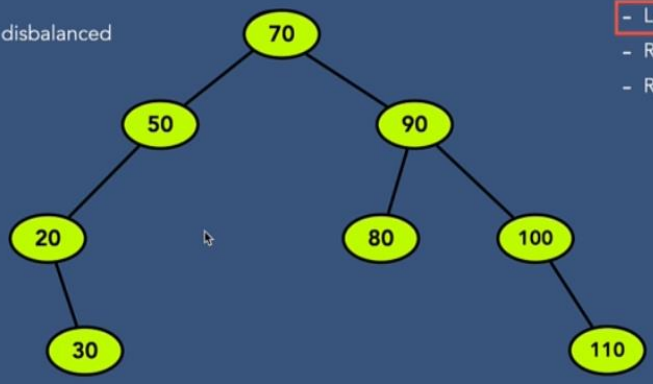
Deleting Node 60 from the AVL Tree



AVL Tree - Delete a Node (all together)

Delete 60

The tree is disbalanced



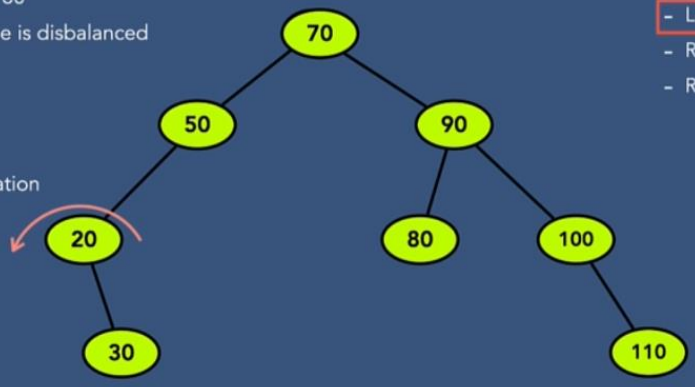
- Left Left Condition (LL)
- Left Right Condition (LR)
- Right Right Condition (RR)
- Right Left Condition (RL)

AVL Tree - Delete a Node (all together)

Delete 60

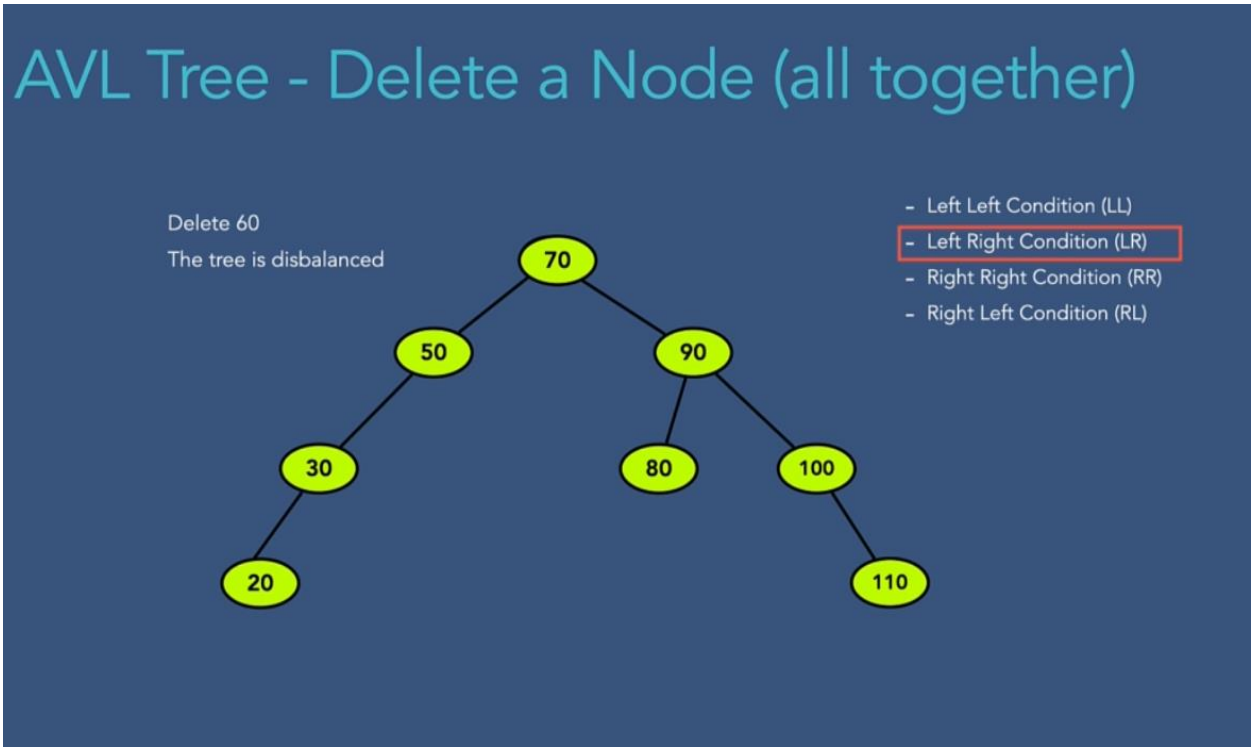
The tree is disbalanced

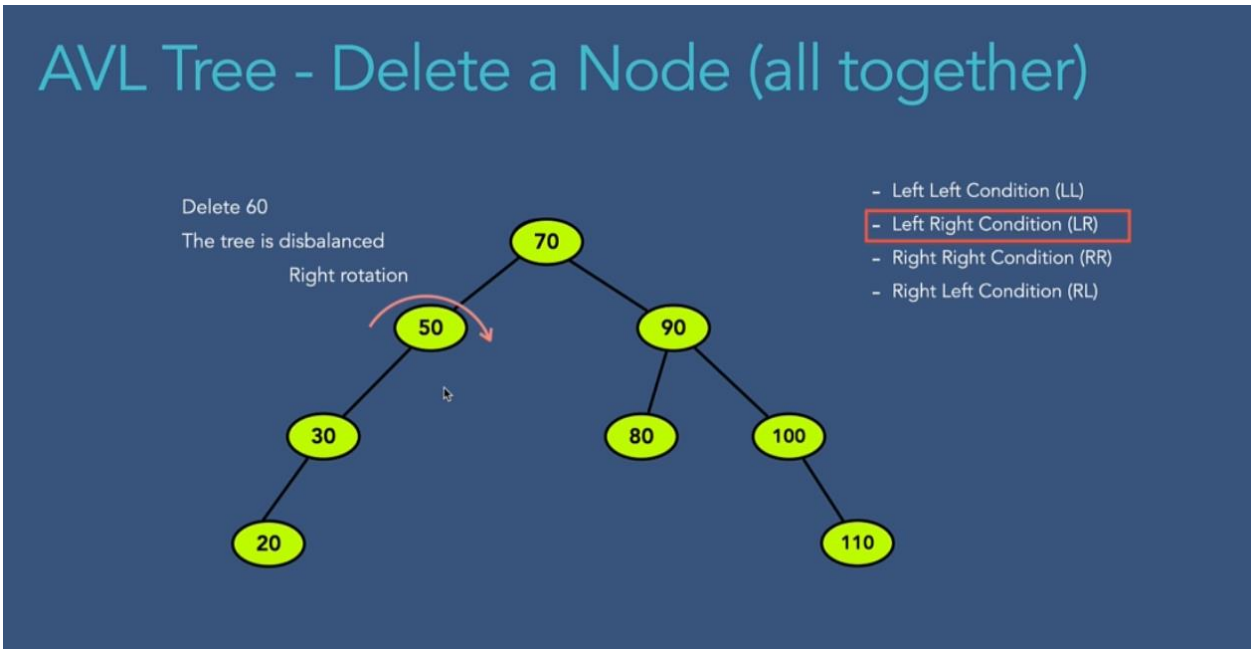
Left rotation



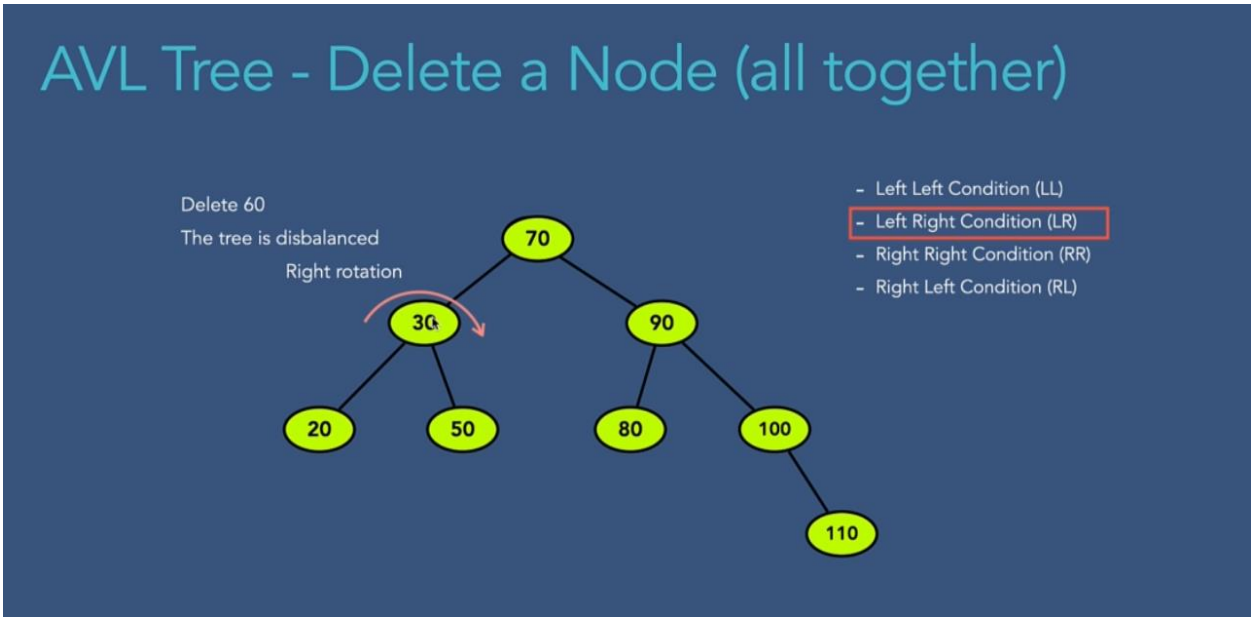
- Left Left Condition (LL)
- Left Right Condition (LR)
- Right Right Condition (RR)
- Right Left Condition (RL)

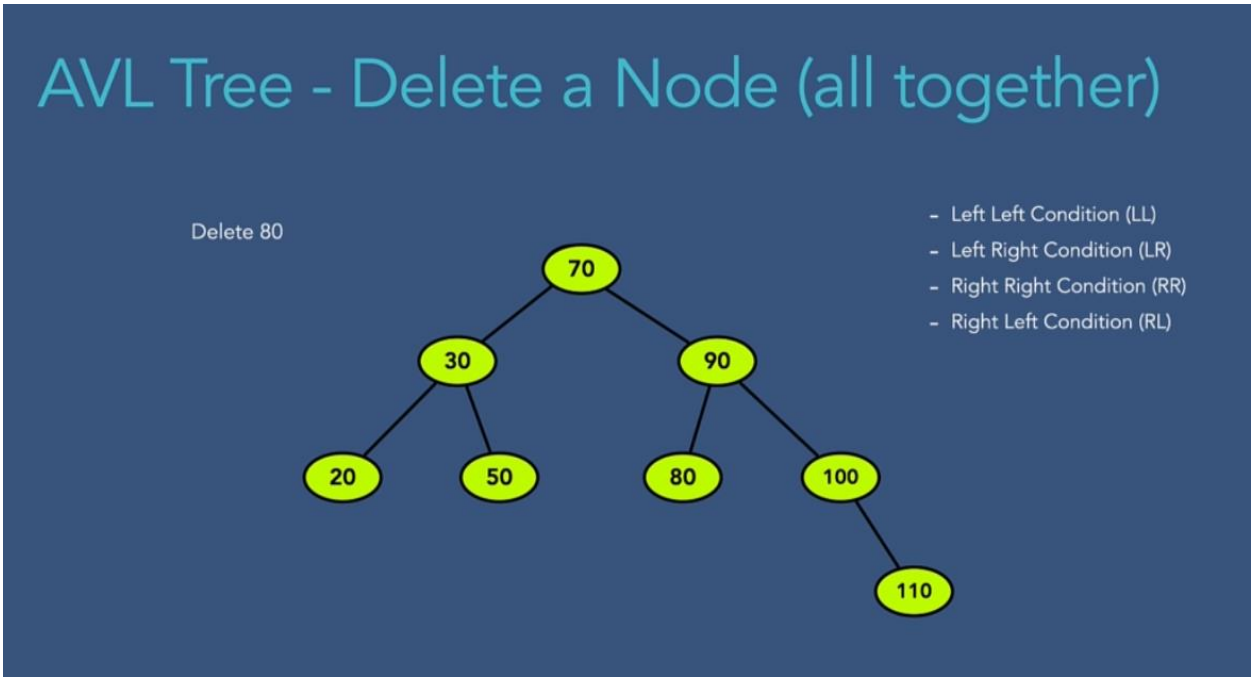
After left rotation



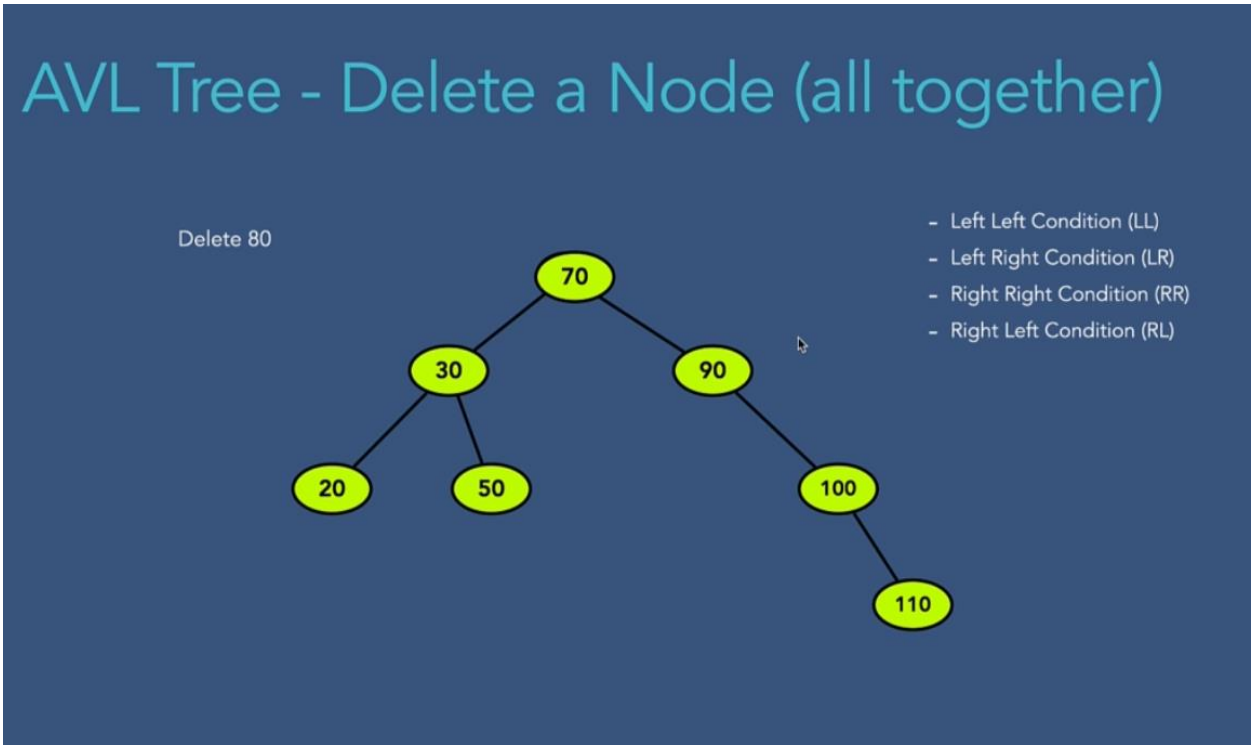


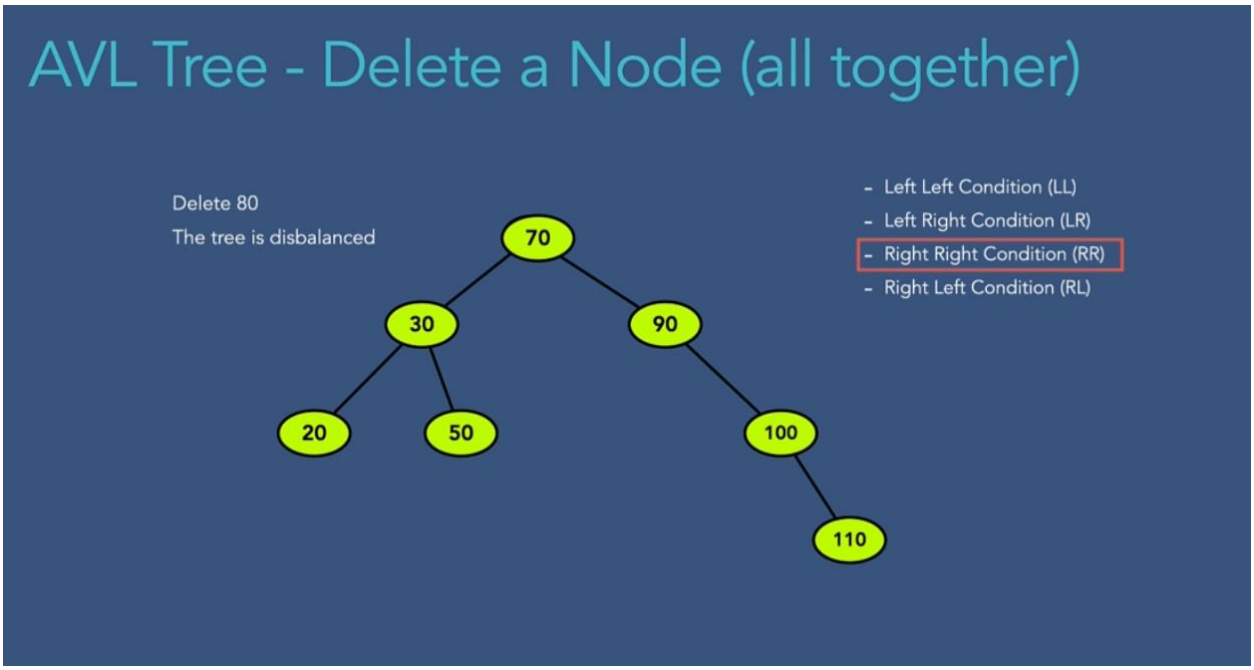
After right rotation

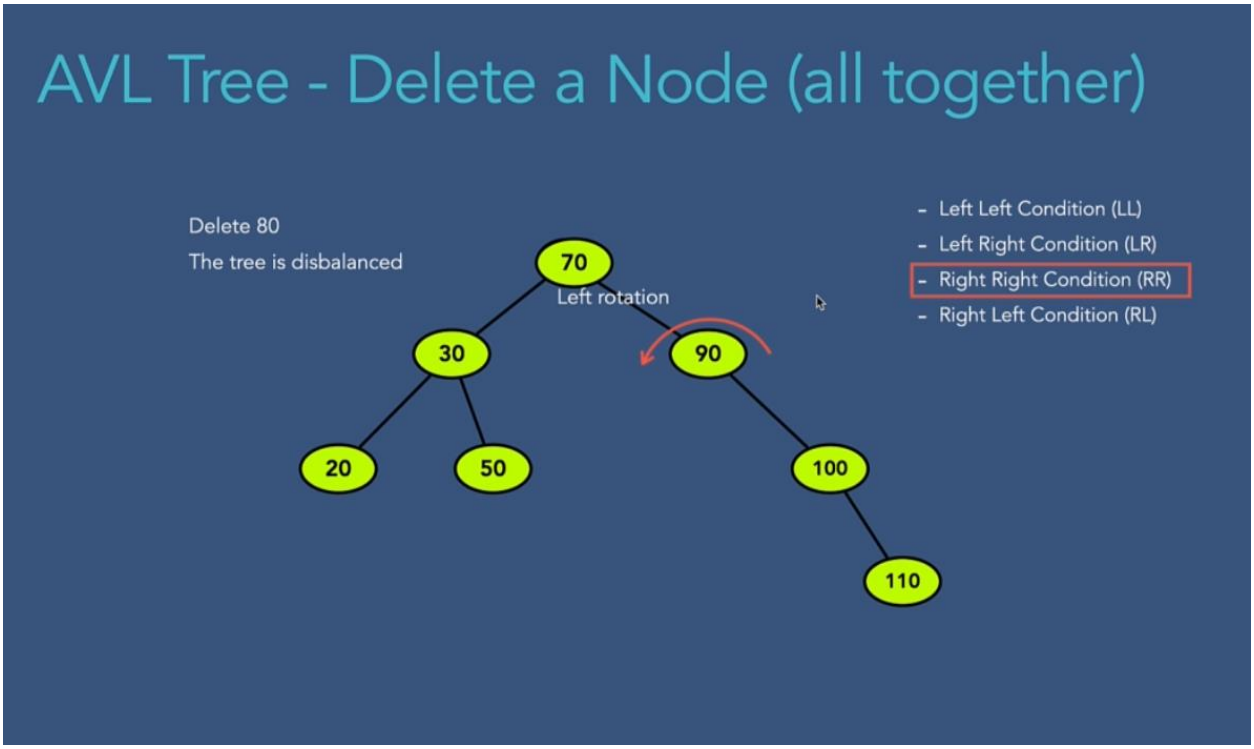




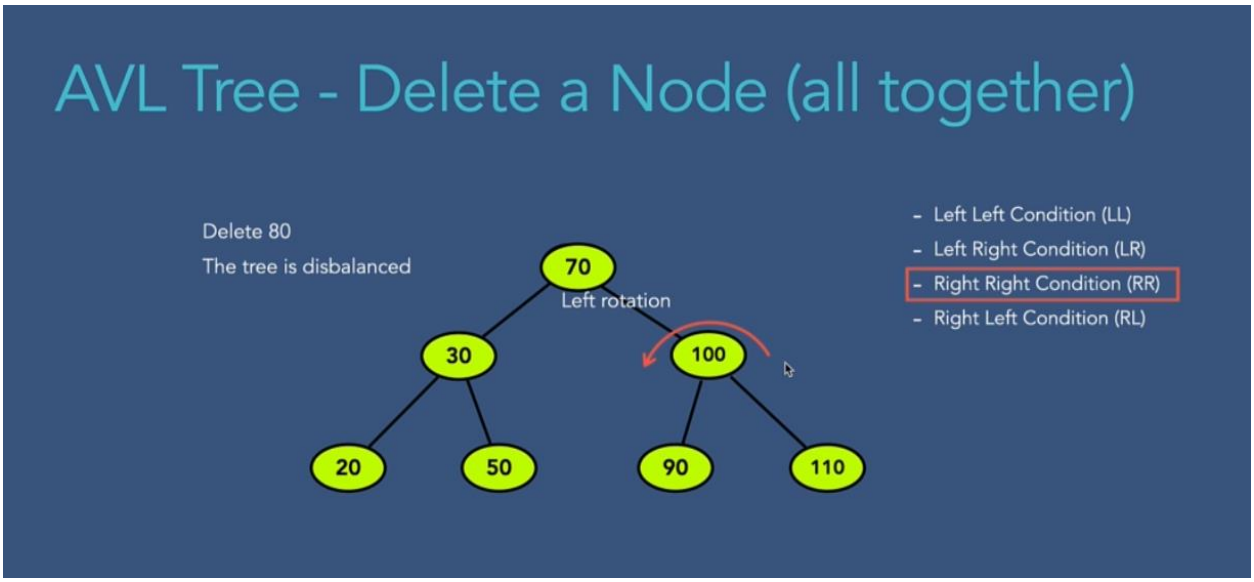
Deleted 80 from the AVL Tree



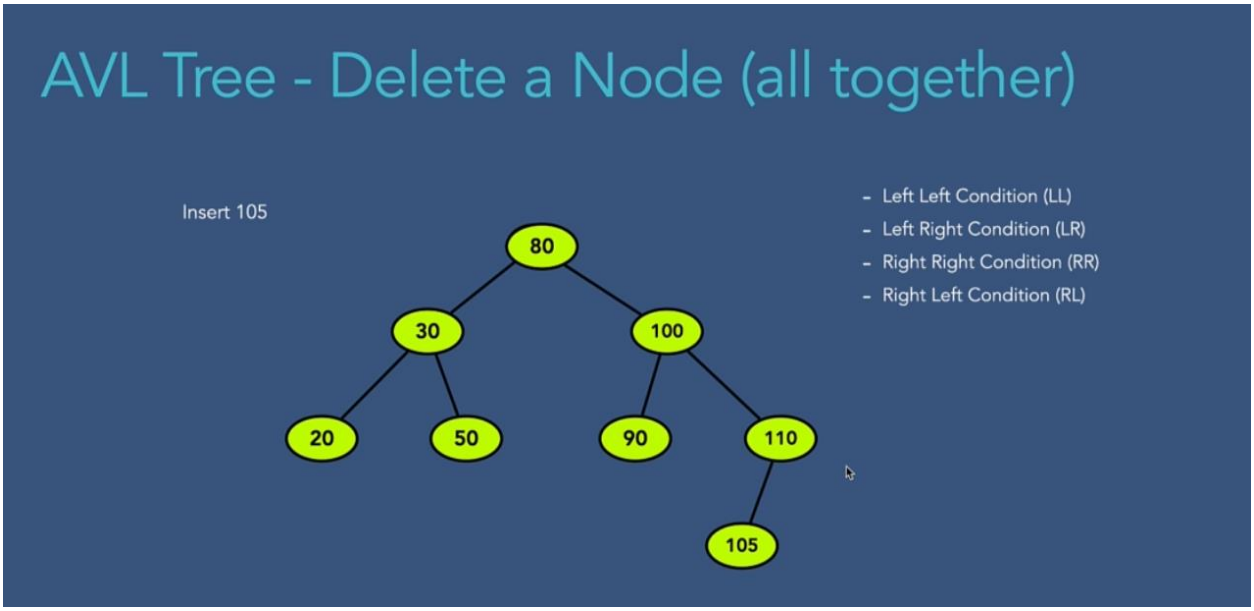


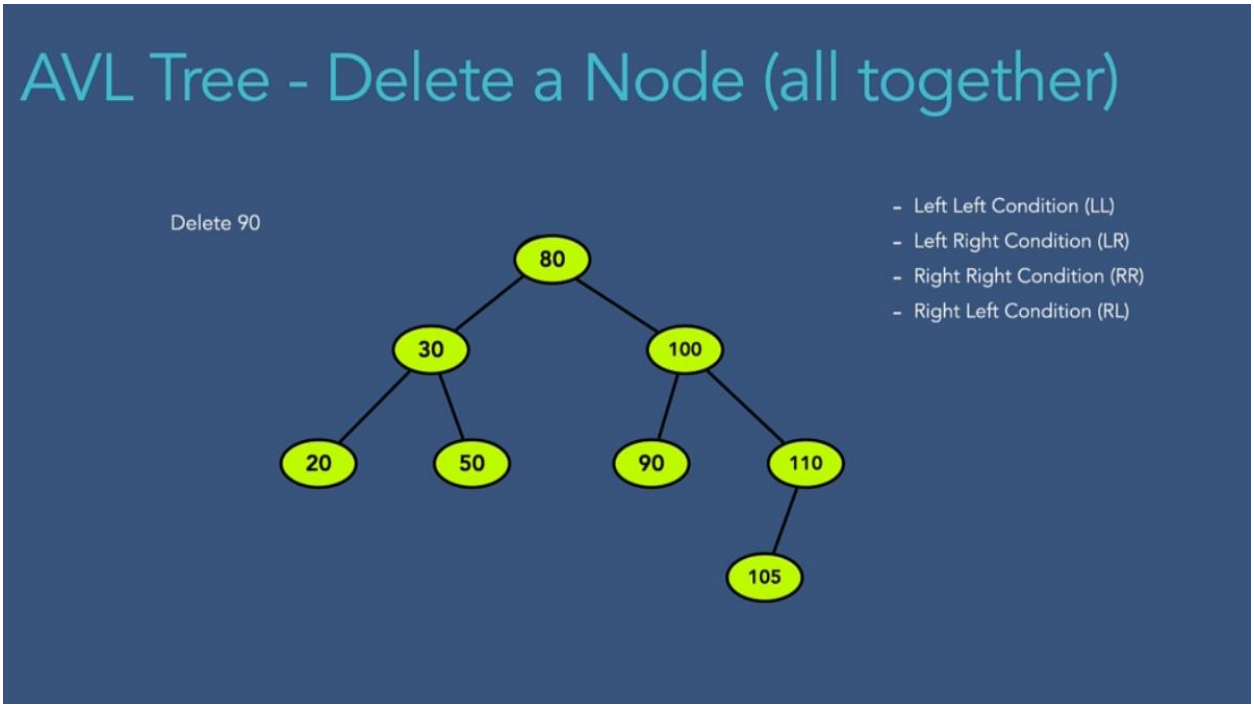


After left rotation

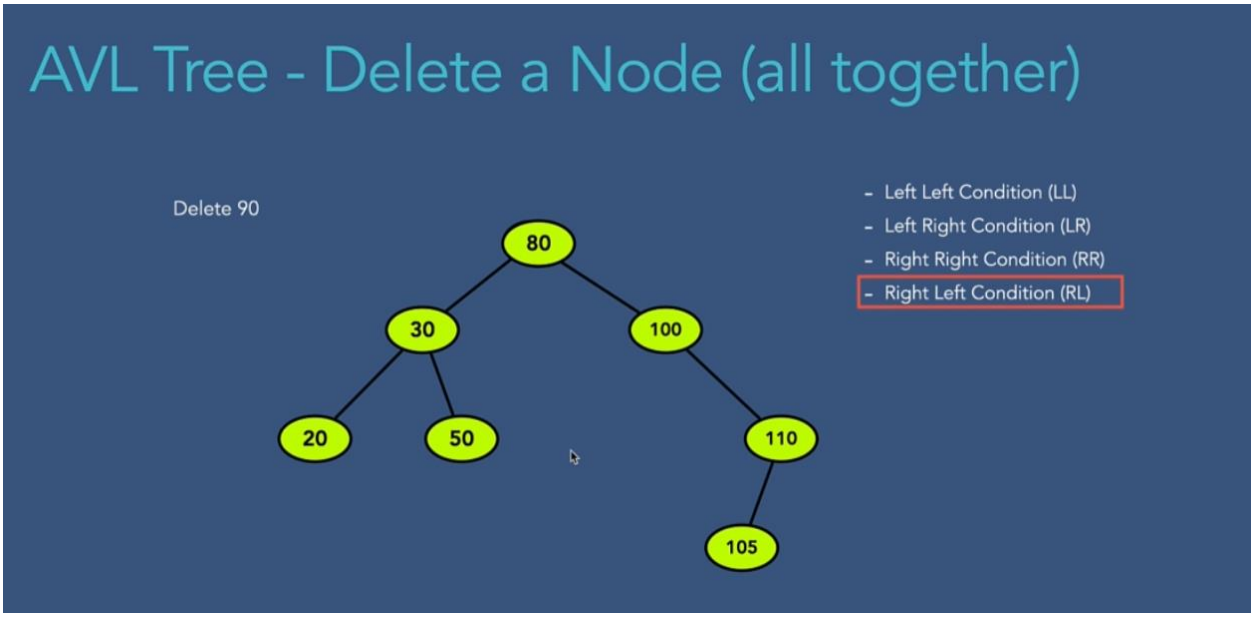


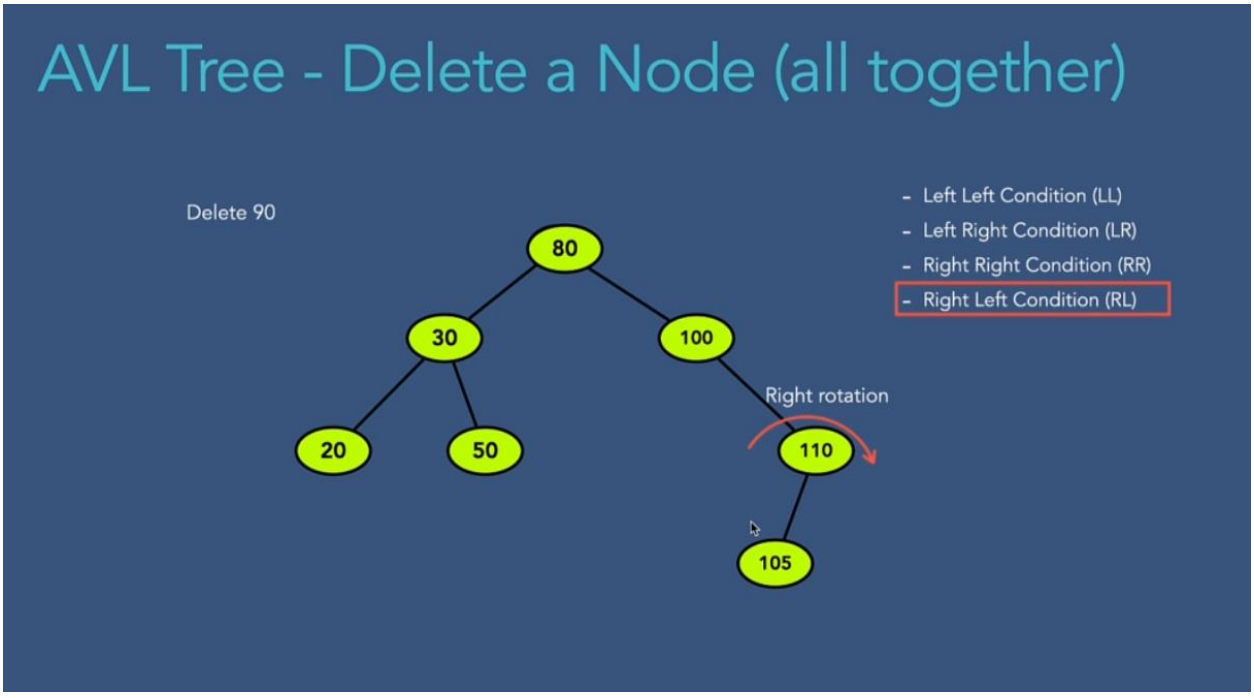
Inserted Node 105 to the AVL Tree



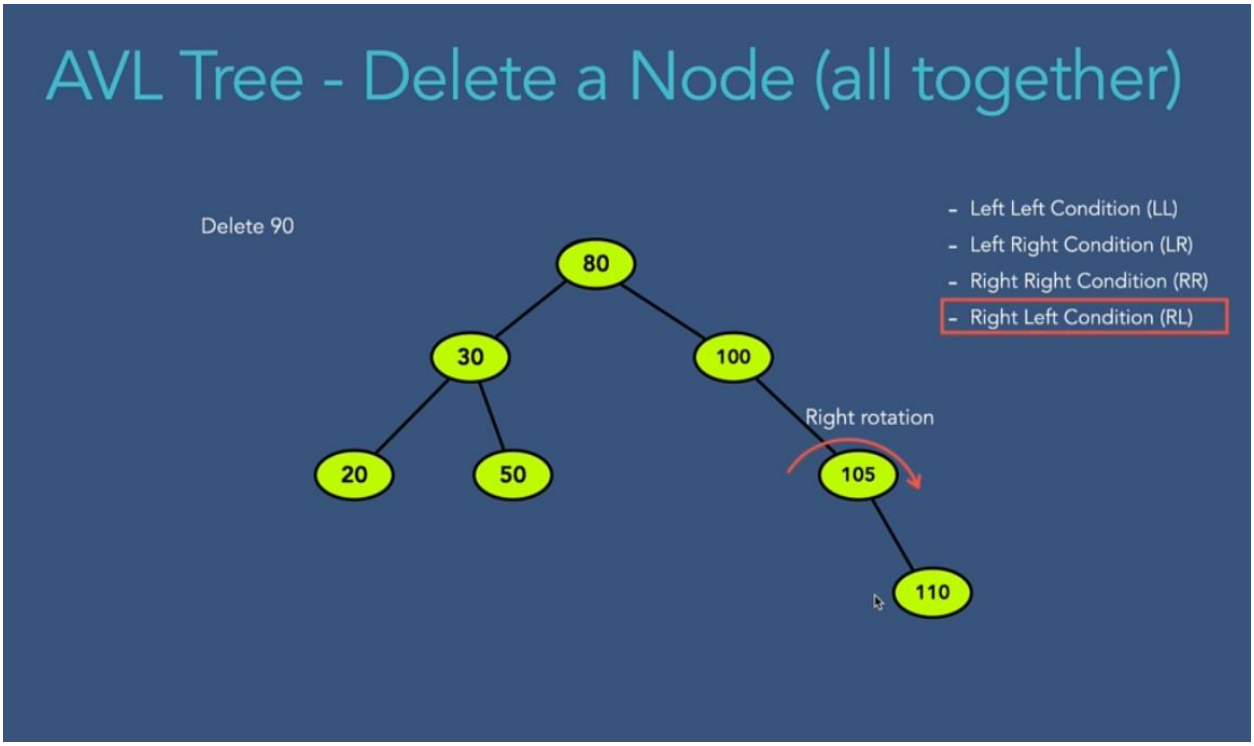


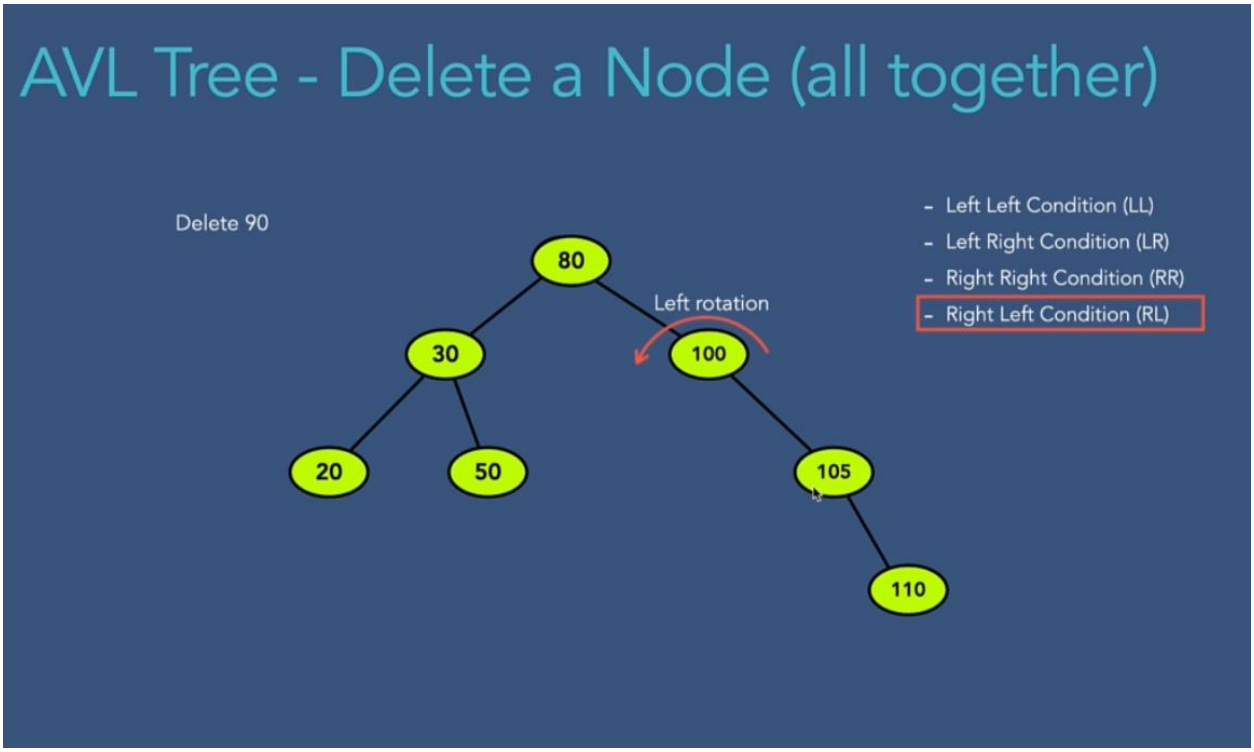
Deleted Node 90 from the AVL Tree



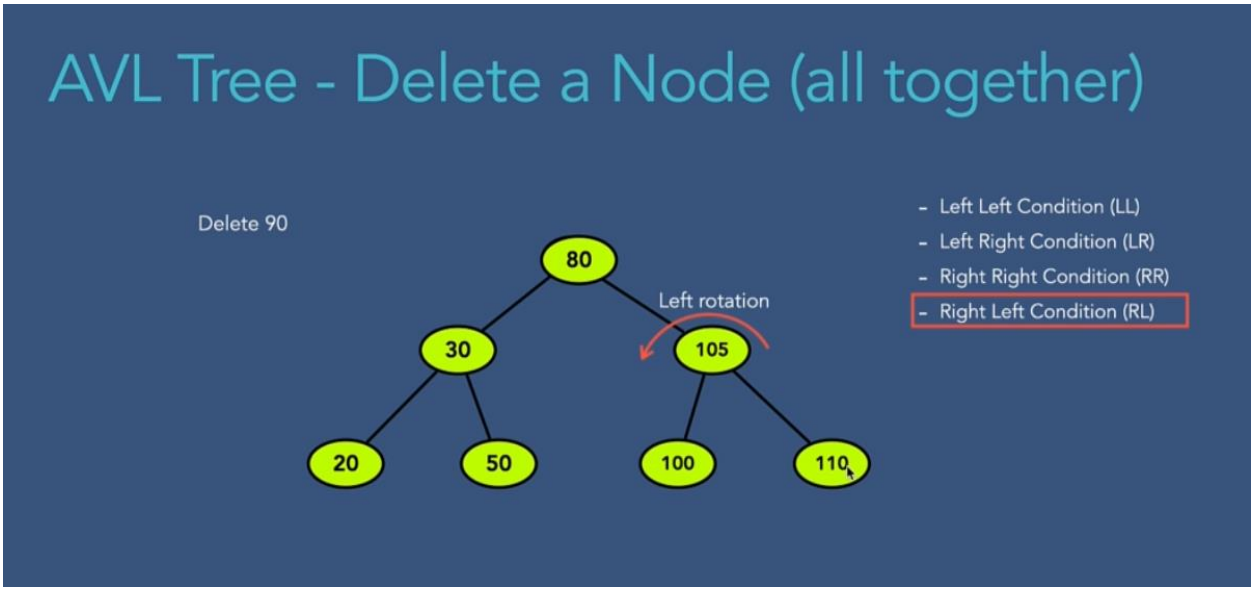


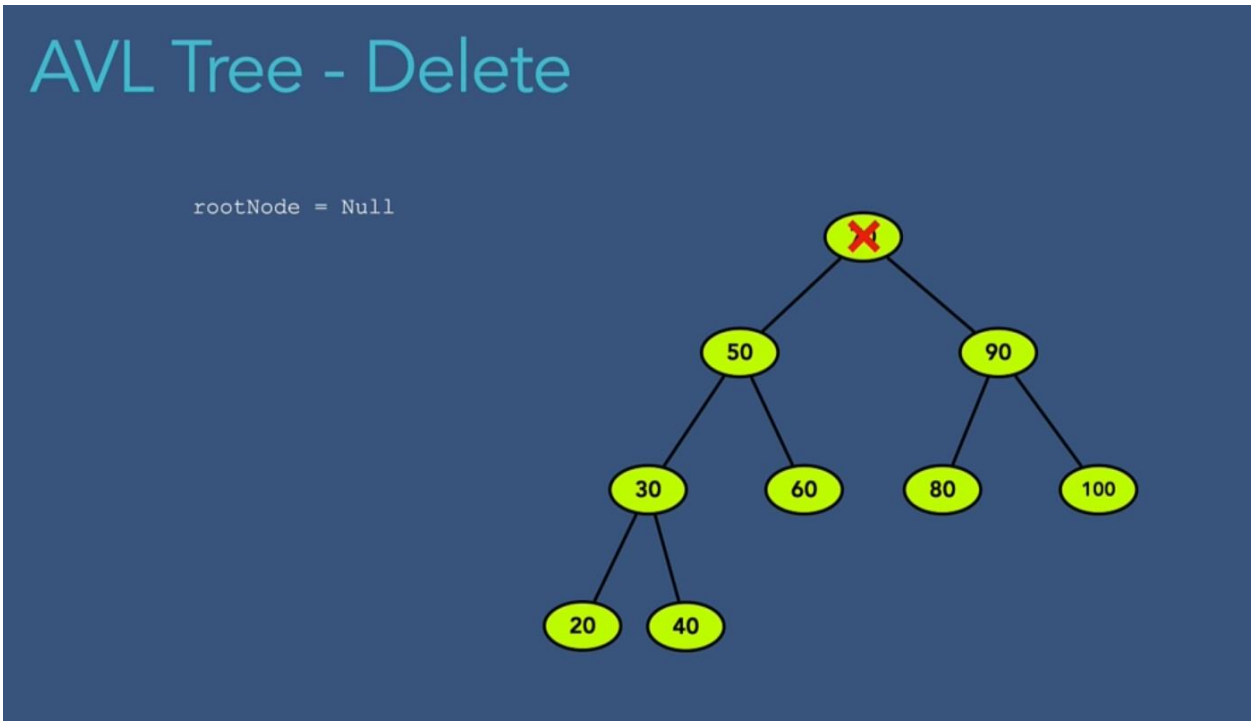
After right rotation





After left rotation





Time and Space Complexity of AVL

	Time complexity	Space complexity
Create AVL	O(1)	O(1)
Insert a node AVL	O(logN)	O(logN)
Traverse AVL	O(N)	O(N)
Search for a node AVL	O(logN)	O(logN)
Delete node from AVL	O(logN)	O(logN)
Delete Entire AVL	O(1)	O(1)