Binary Search Tree Implementation

Structure of node of a tree

```
class bstnode
{
         public:
        int data;
        bstnode *left;
        bstnode *right;

};
```

Inserting node in Binary Search Tree

```
Step 1: IF TREE = NULL
    Allocate memory for TREE
   SET TREE -> DATA = ITEM
  SET TREE -> LEFT = TREE -> RIGHT = NULL
  ELSE
   IF ITEM < TREE -> DATA
    Insert(TREE -> LEFT, ITEM)
  ELSE
   Insert(TREE -> RIGHT, ITEM)
  [END OF IF]
  [END OF IF]
Step 2: END
```

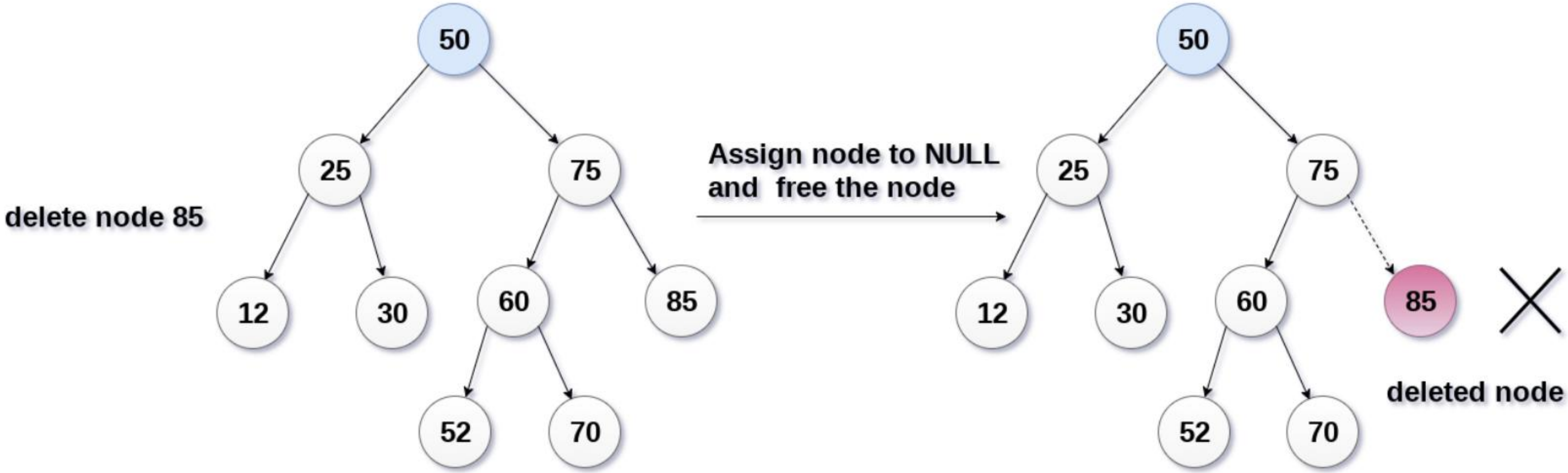
Deletion

Delete function is used to delete the specified node from a binary search tree. However, we must delete a node from a binary search tree in such a way, that the property of binary search tree doesn't violate. There are three situations of deleting a node from binary search tree.

The node to be deleted is a leaf node

It is the simplest case, in this case, replace the leaf node with the NULL and simple free the allocated space.

In the following image, we are deleting the node 85, since the node is a leaf node, therefore the node will be replaced with NULL and allocated space will be freed.



delete node 85

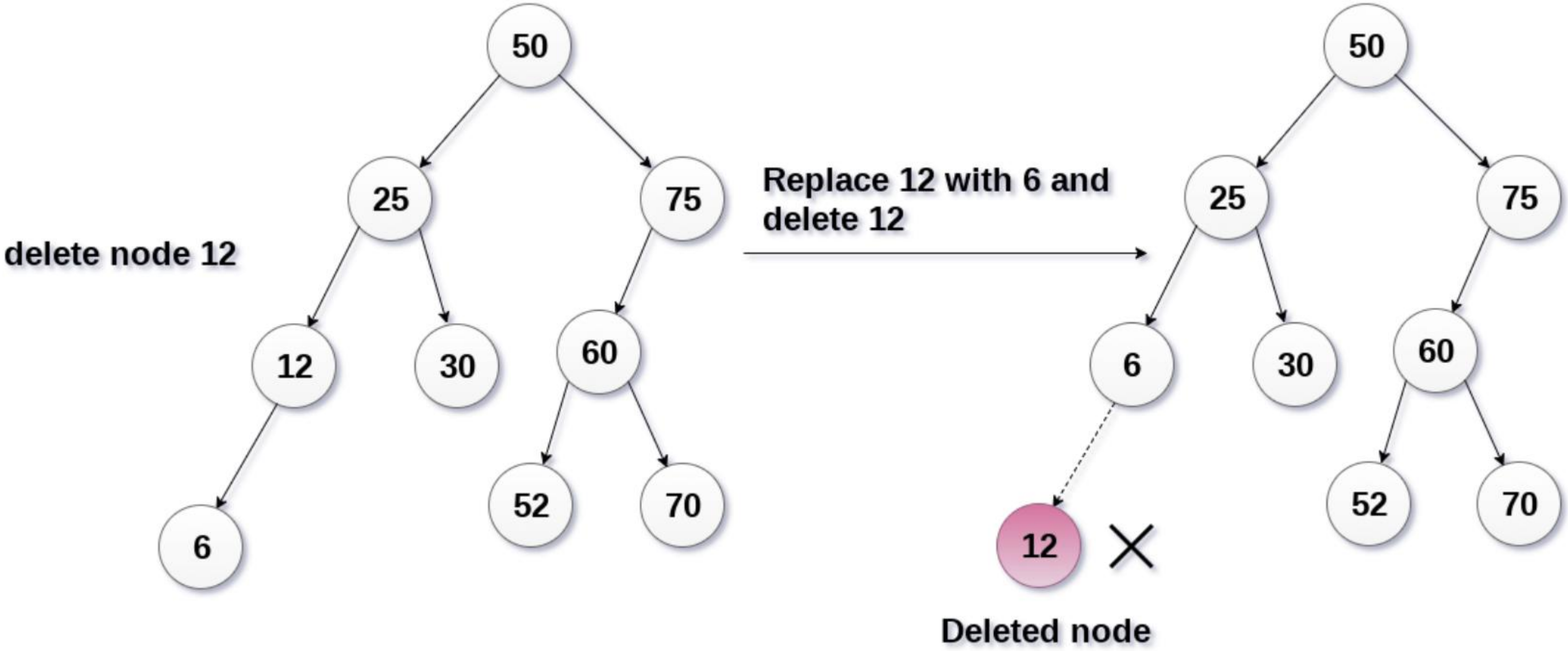
Assign node to NULL and free the node

deleted node

The node to be deleted has only one child.

In this case, replace the node with its child and delete the child node, which now contains the value which is to be deleted. Simply replace it with the NULL and free the allocated space.

In the following image, the node 12 is to be deleted. It has only one child. The node will be replaced with its child node and the replaced node 12 (which is now leaf node) will simply be deleted.



delete node 12

Replace 12 with 6 and delete 12

Deleted node

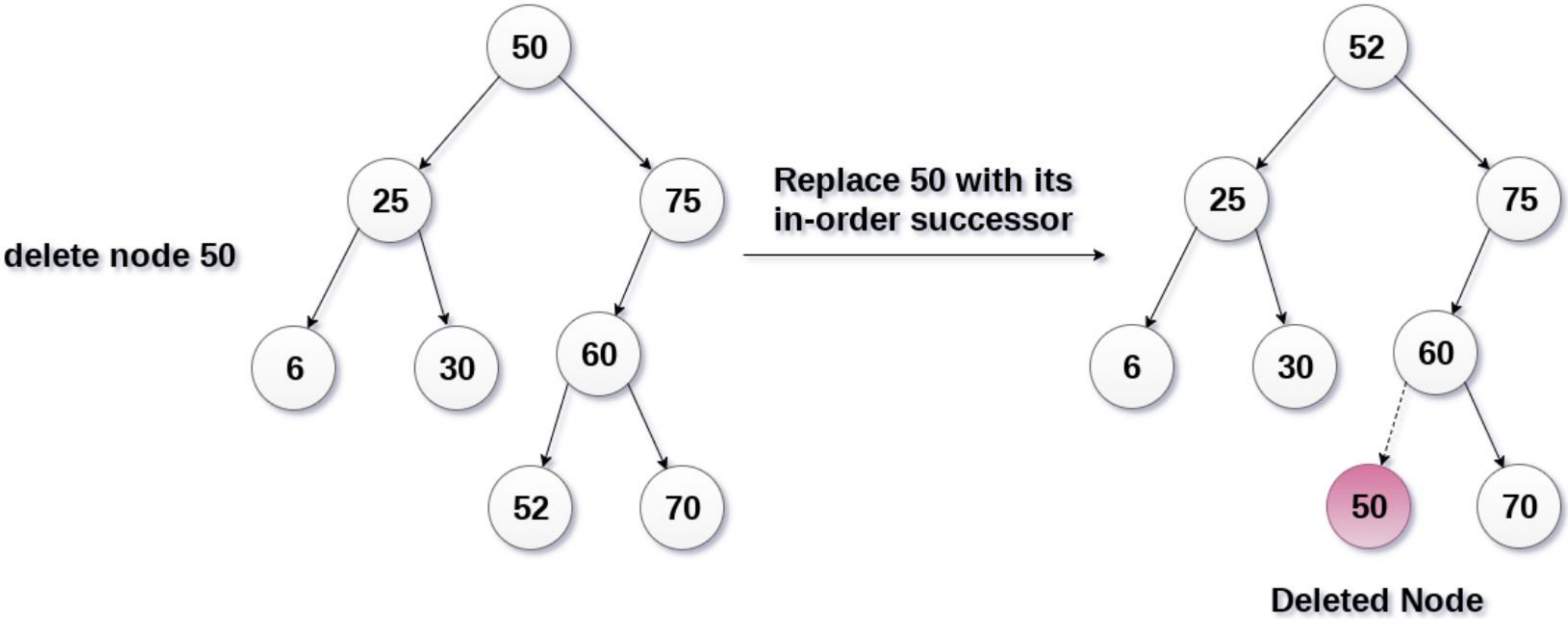
The node to be deleted has two children.

It is a bit complexed case compare to other two cases. However, the node which is to be deleted, is replaced with its in-order successor or predecessor recursively until the node value (to be deleted) is placed on the leaf of the tree. After the procedure, replace the node with NULL and free the allocated space.

In the following image, the node 50 is to be deleted which is the root node of the tree. The in-order traversal of the tree given below.

6, 25, 30, 50, 52, 60, 70, 75.

replace 50 with its in-order successor 52. Now, 50 will be moved to the leaf of the tree, which will simply be deleted.



delete node 50

Replace 50 with its in-order successor

Deleted Node

```
Delete (TREE, ITEM)
Step 1: IF TREE = NULL
   Write "item not found in the tree" ELSE IF ITEM < TREE -> DATA
  Delete(TREE->LEFT, ITEM)
  ELSE IF ITEM > TREE -> DATA
   Delete(TREE -> RIGHT, ITEM)
  ELSE IF TREE -> LEFT AND TREE -> RIGHT
  SET TEMP = findLargestNode(TREE -> LEFT)
  SET TREE -> DATA = TEMP -> DATA
   Delete(TREE -> LEFT, TEMP -> DATA)
  ELSE
   SET TEMP = TREE
   IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
   SET TREE = NULL
  ELSE IF TREE -> LEFT != NULL
  SET TREE = TREE -> LEFT
  ELSE
    SET TREE = TREE -> RIGHT
  [END OF IF]
  FREE TEMP
[END OF IF]
Step 2: END
```

Searching

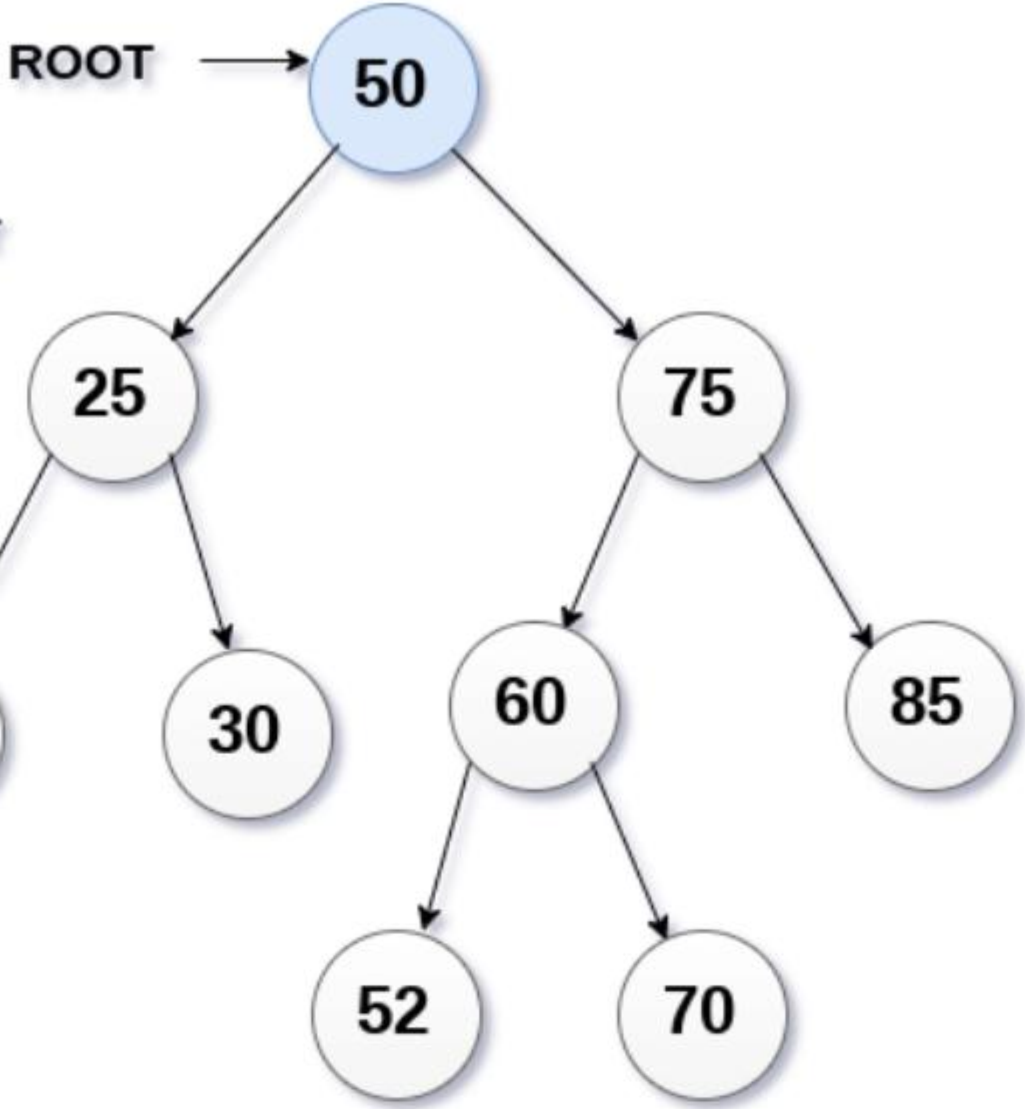
Searching means finding or locating some specific element or node within a data structure. However, searching for some specific node in binary search tree is pretty easy due to the fact that, element in BST are stored in a particular order.

1. Compare the element with the root of the tree.
2. If the item is matched then return the location of the node.
3. Otherwise check if item is less than the element present on root, if so then move to the left sub-tree.
4. If not, then move to the right sub-tree.
5. Repeat this procedure recursively until match found.
6. If element is not found then return NULL.

Item = 60

ITEM > ROOT -> DATA

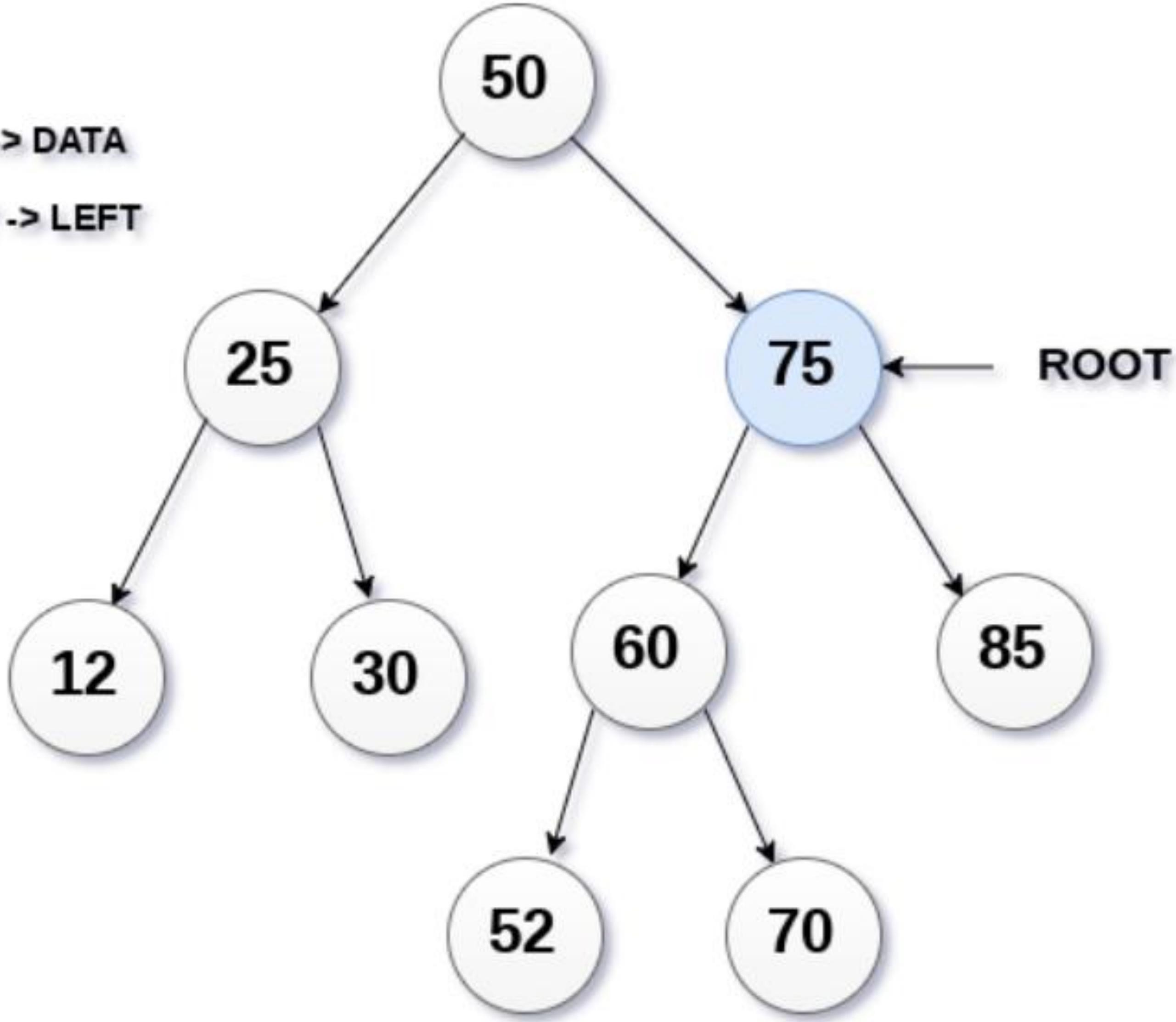
ROOT = ROOT -> RIGHT



Item = 60

ITEM < ROOT -> DATA

ROOT = ROOT -> LEFT

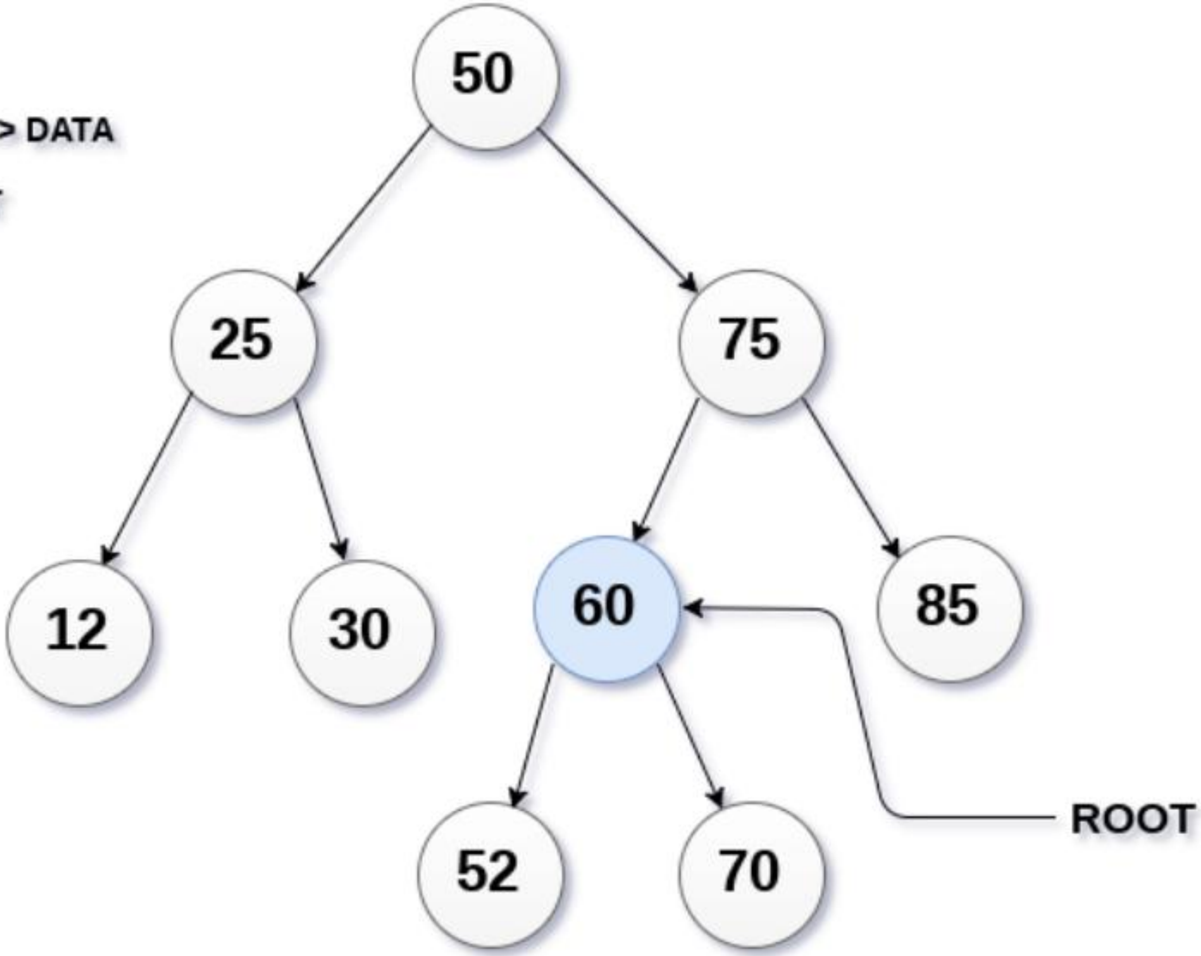


STEP 2

Item = 60

ITEM = ROOT -> DATA

RETURN ROOT



STEP 3

Search (ROOT, ITEM)

```
Step 1: IF ROOT -> DATA = ITEM OR ROOT = NULL
    Return ROOT
   ELSE
   IF ROOT < ROOT -> DATA
   Return search(ROOT -> LEFT, ITEM)
  ELSE
   Return search(ROOT -> RIGHT,ITEM)
  [END OF IF]
  [END OF IF]
Step 2: END
```

Pre-order

A pre-order traversal on a tree performs the following steps starting from the root:

1) Return the root node value.
2) Traverse the left subtree by recursively calling the pre-order function.
3) Traverse the right subtree by recursively calling the pre-order function.

In-order

An in-order traversal on a tree performs the following steps starting from the root:

1) Traverse the left subtree by recursively calling the in-order function.
2) Return the root node value.
3) Traverse the right subtree by recursively calling the in-order function.

Post-order

A post-order traversal on a tree performs the following steps starting from the root:

1) Traverse the left subtree by recursively calling the post-order function.
2) Traverse the right subtree by recursively calling the post-order function.
3) Return the root node value.

Algorithm for finding minimum or maximum element in Binary Search Tree

As we know the Property of Binary search tree. This quite simple

Approch for finding minimum element:

1. Traverse the node from root to left recursively until left is NULL.
2. The node whose left is NULL is the node with minimum value.

Approch for finding maximum element:

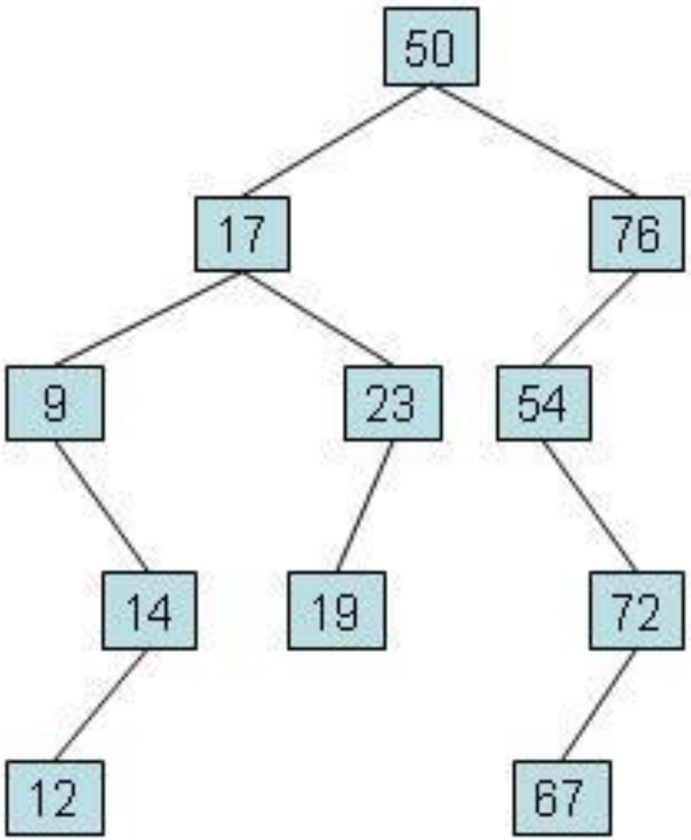
1. Traverse the node from root to right recursively until right is NULL.
2. The node whose right is NULL is the node with maximum value.

3. Define another class which has an attribute root.

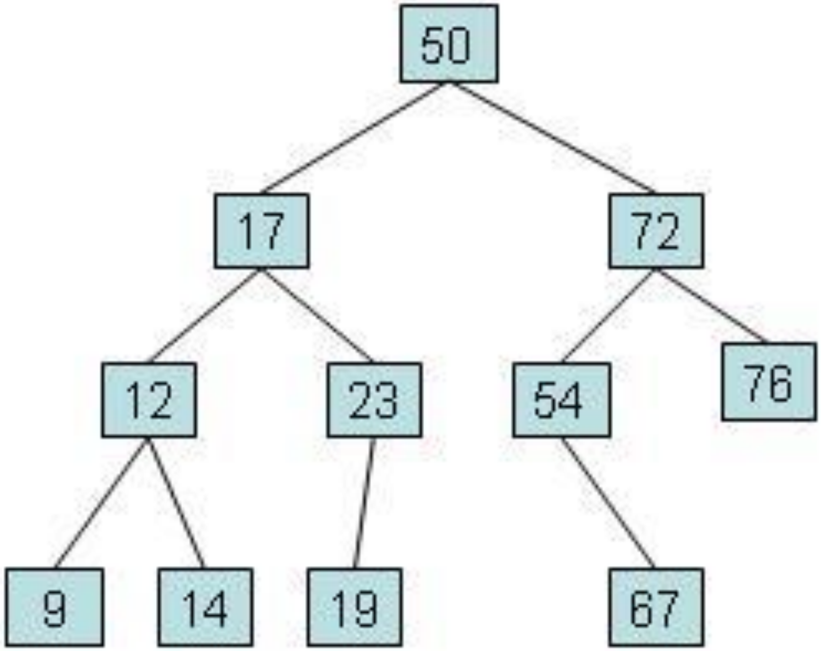
- **A. Root represents the root node of the tree which is initialized to null.**
- **B. largestElement() will find out the largest node in the binary tree:**
    - **I. It checks whether the root is null, which means the tree is empty.**
    - **II. If the tree is not empty, define a variable max that will store temp's data.**
    - **III. Find out the maximum node in the left subtree by calling largestElement() recursively. Store that value in leftMax. Compare the value of max with leftMax and store the maximum of two to max.**
    - **IV. Find out the maximum node in right subtree by calling largestElement() recursively. Store that value in rightMax. Compare the value of max with rightMax and store the maximum of two to max.**
    - **V. In the end, max will hold the largest node in the binary tree.**

Self-Balancing Binary Search Tree

A self-balancing binary search tree or height-balanced binary search tree is a binary search tree (BST) that attempts to keep its height, or the number of levels of nodes beneath the root, as small as possible at all times, automatically.



An unbalanced tree



The same tree after
being height-balanced

What is the major disadvantage of an ordinary BST?

- Most operations on a BST take time proportional to the height of the tree, so it is desirable to keep the height small.
- Self-balancing binary trees solve this problem by performing transformations on the tree at key times, in order to reduce the height. Although a certain overhead is involved, it is justified in the long run by ensuring fast execution of later operations.
- The height must always be at most the ceiling of log_2n.
- Balanced BSTs are not always so precisely balanced, since it can be expensive to keep a tree at minimum height at all times; instead, most algorithms keep the height within a constant factor of this lower bound.

Implementations:

- AA tree
- AVL tree
- Red-black tree
- Scapegoat tree
- Splay tree
- Treap

Self-Balancing BST Applications:

- They can be used to construct and maintain ordered lists, such as priority queues.
- They can also be used for associative arrays; key-value pairs are inserted with an ordering based on the key alone. In this capacity, self-balancing BSTs have a number of advantages and disadvantages over their main competitor, hash tables.
- They can be easily extended to record additional information or perform new operations. These extensions can be used, for example, to optimize database queries or other list-processing algorithms.

AVL Tree

AVL tree is a self-balancing binary search tree in which each node maintains extra information called a balance factor whose value is either -1, 0 or +1.

AVL tree got its name after its inventor Georgy Adelson-Velsky and Landis.

What is an AVL Tree?

An AVL tree is a subtype of binary search tree. Named after it's inventors Adelson, Velskii and Landis, AVL trees have the property of dynamic self-balancing in addition to all the properties exhibited by binary search trees.

A BST is a data structure composed of nodes. It has the following guarantees:

- Each tree has a root node (at the top).
- The root node has zero, one or two child nodes.
- Each child node has zero, one or two child nodes, and so on.
- Each node has up to two children.
- For each node, its left descendants are less than the current node, which is less than the right descendants.

AVL trees have an additional guarantee:

The difference between the depth of right and left subtrees cannot be more than one. In order to maintain this guarantee, an implementation of an AVL will include an algorithm to rebalance the tree when adding an additional element would upset this guarantee.

Balance Factor (k) = height (left(k)) - height (right(k))

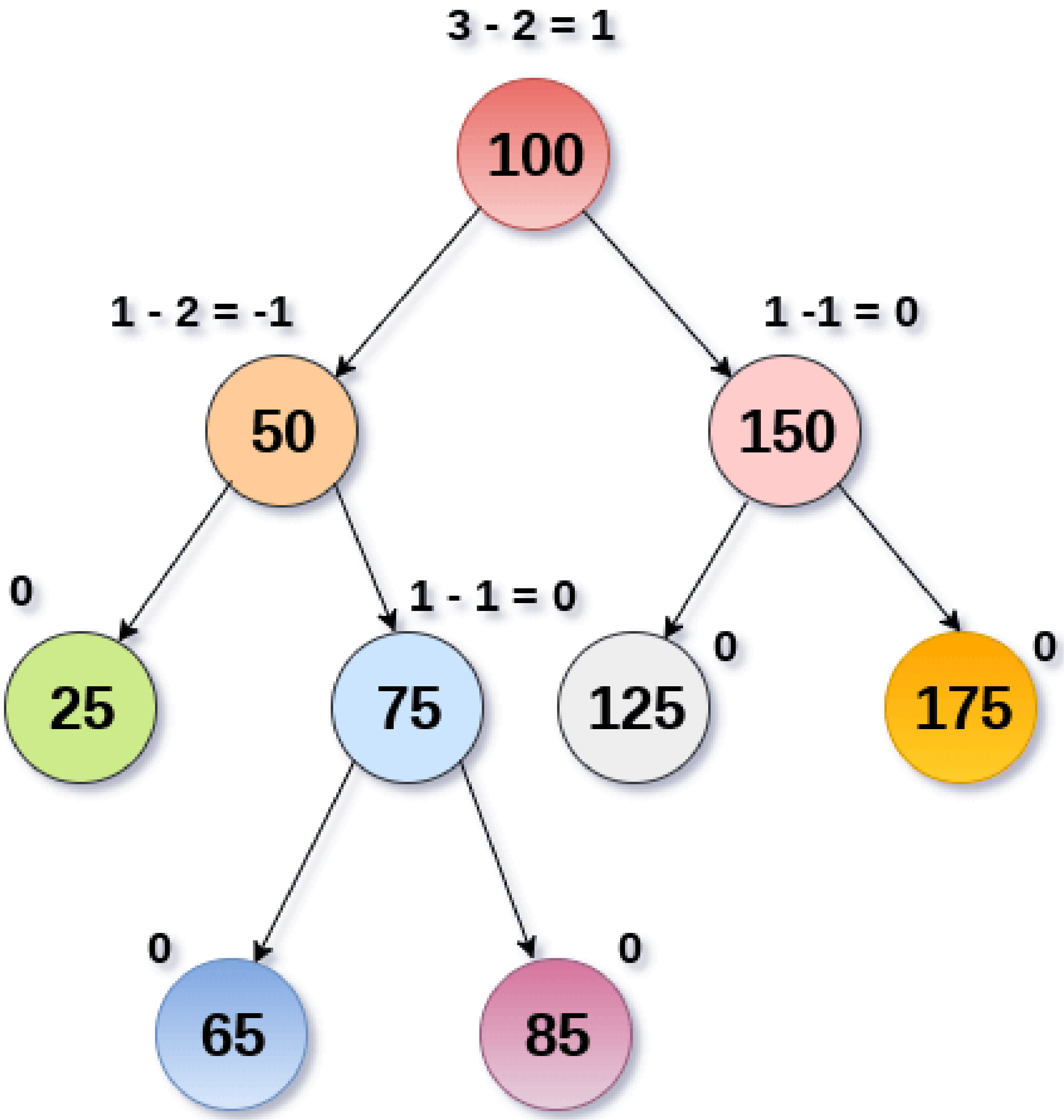
1. If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.
2. If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.
3. If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.
4. An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. therefore, it is an example of AVL tree.

Balance Factor

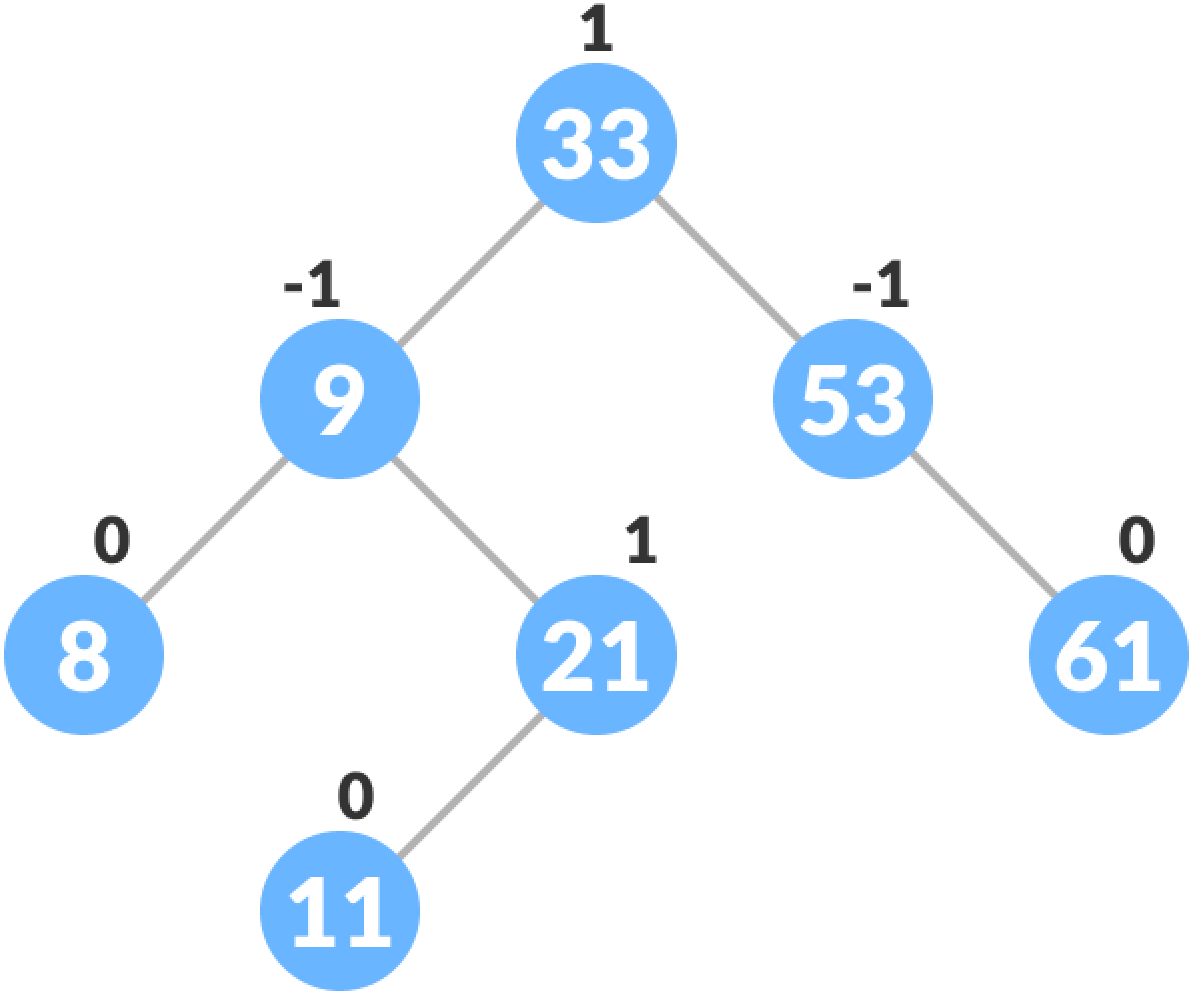
Balance factor of a node in an AVL tree is the difference between the height of the left subtree and that of the right subtree of that node.

Balance Factor = (Height of Left Subtree - Height of Right Subtree) or (Height of Right Subtree - Height of Left Subtree)

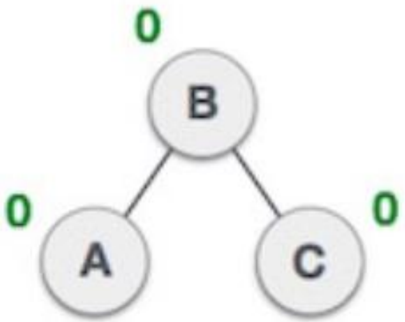
The self balancing property of an avl tree is maintained by the balance factor. The value of balance factor should always be -1, 0 or +1.



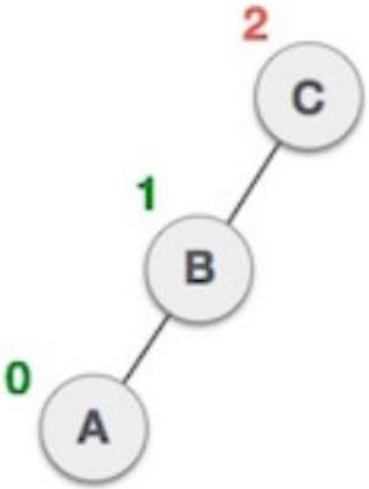
AVL Tree



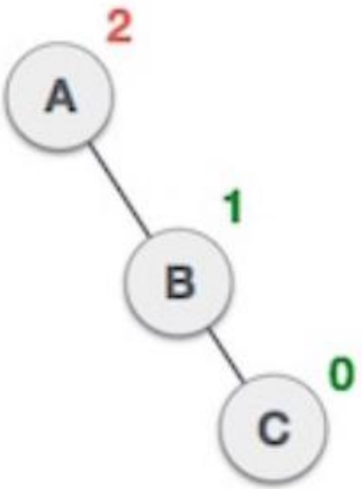
Avl tree



Balanced



Not balanced



Not balanced

Operations on AVL tree

Due to the fact that, AVL tree is also a binary search tree therefore, all the operations are performed in the same way as they are performed in a binary search tree. Searching and traversing do not lead to the violation in property of AVL tree. However, insertion and deletion are the operations which can violate this property and therefore, they need to be revisited.

Complexity

Algorithm	Average case	Worst case
Space	o(n)	o(n)
Search	o(log n)	o(log n)
Insert	o(log n)	o(log n)
Delete	o(log n)	o(log n)

Operations on an AVL tree

Various operations that can be performed on an AVL tree are:

Rotating the subtrees in an AVL Tree

In rotation operation, the positions of the nodes of a subtree are interchanged.

There are two types of rotations:

Rotation operations are used to make a tree balanced.There are four rotations and they are classified into two types:

Single Left Rotation (LL Rotation)

In LL Rotation every node moves one position to left from the current position.

Single Right Rotation (RR Rotation)

In RR Rotation every node moves one position to right from the current position.

Left Right Rotation (LR Rotation)

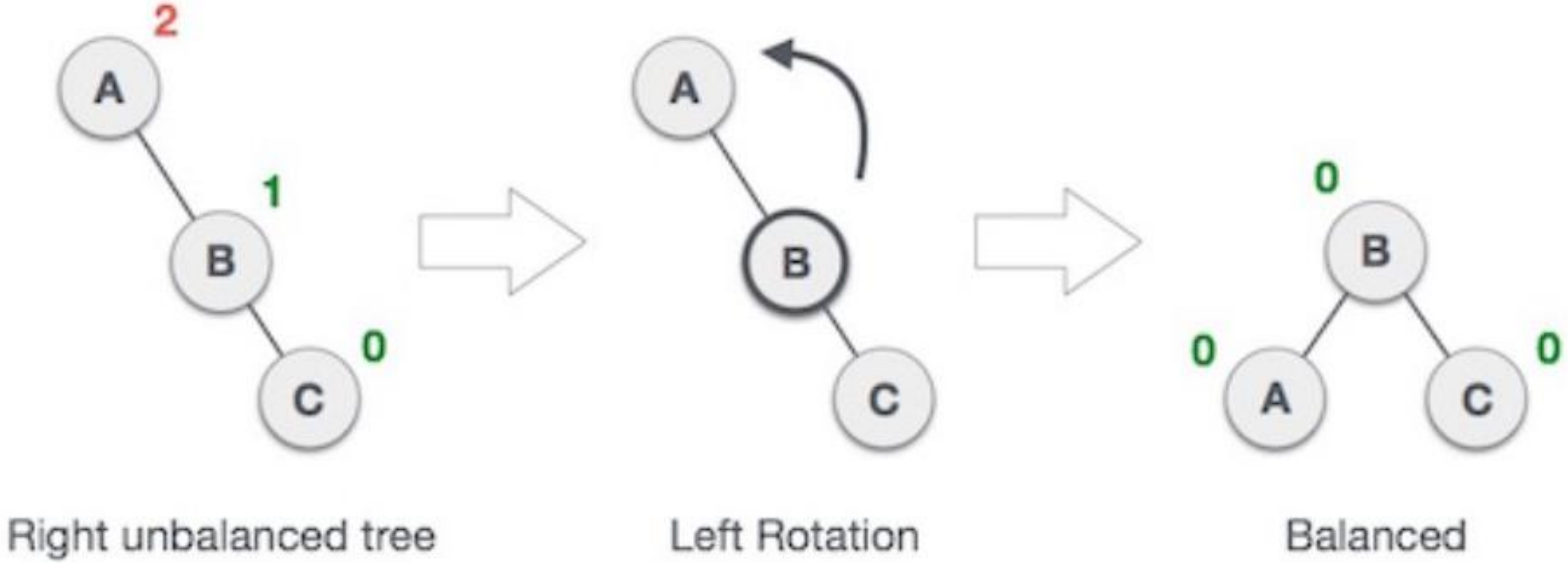
The LR Rotation is combination of single left rotation followed by single right rotation. In LR Rotation, first every node moves one position to left then one position to right from the current position.

Right Left Rotation (RL Rotation)

The RL Rotation is combination of single right rotation followed by single left rotation. In RL Rotation, first every node moves one position to right then one position to left from the current position.

Left Rotation

If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation –



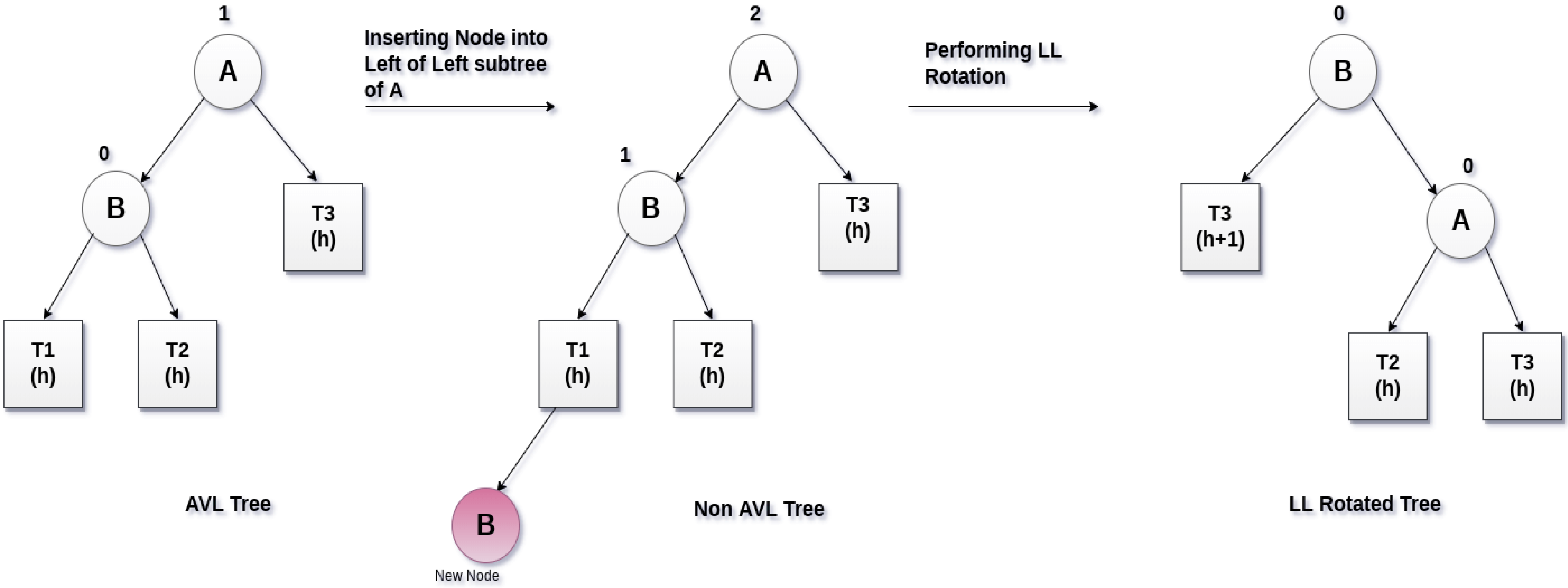
LL Rotation

The tree shown in following figure is an AVL Tree, however, we,need to insert an element into the left of the left sub-tree of A. the tree can become unbalanced with the presence of the critical node A.

The node whose balance factor doesn't lie between -1 and 1, is called critical node.

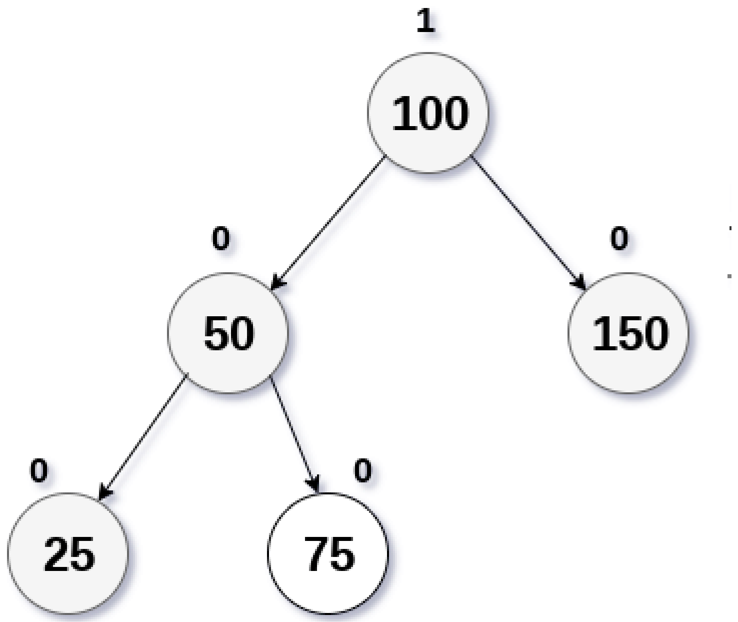
In order to rebalance the tree, LL rotation is performed as shown in the following diagram.

The node B becomes the root, with A and T3 as its left and right child. T1 and T2 becomes the left and right sub-trees of A.



Example :

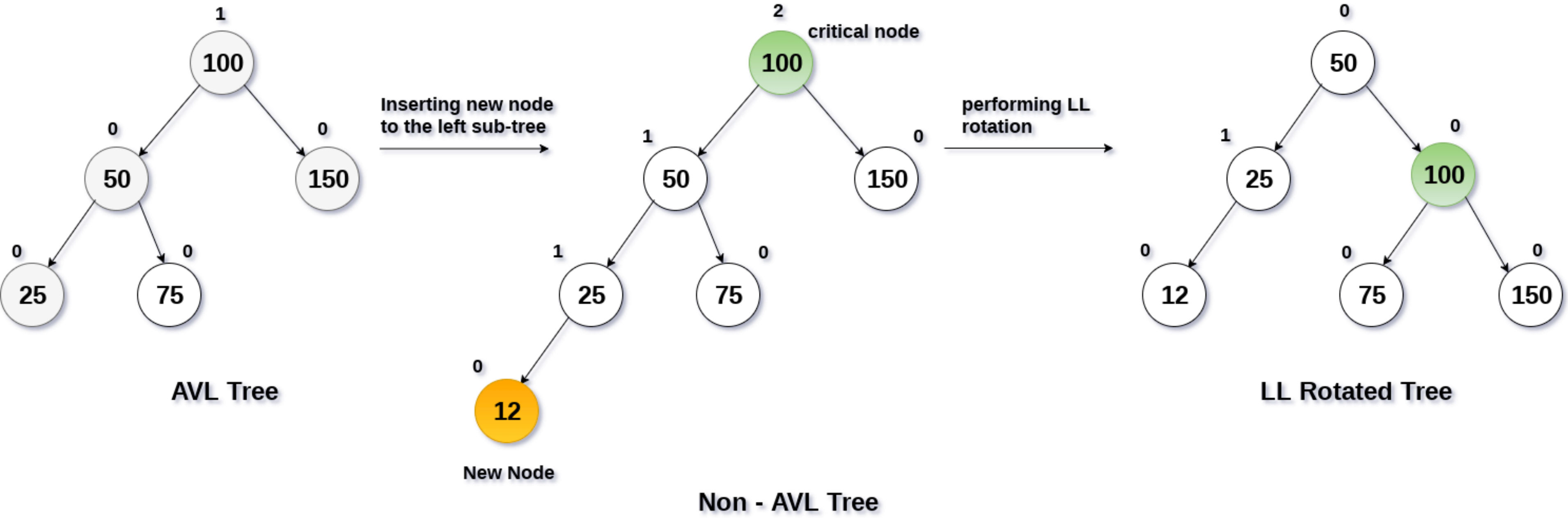
Insert the node with value 12 into the tree shown in the following figure.



AVL Tree

Solution:

- 12 will be inserted to the left of 25 and therefore, it disturbs the AVLness of the tree. The tree needs to be rebalanced by rotating it through LL rotation.
- Here, the critical node 100 will be moved to its right, and the root of its left sub-tree (B) will be the new root node of the tree.
- The right sub-tree of B i.e. T2 (with root node 75) will be place to the left of Node A (with value 100).
- By following this procedure, the tree will be rebalanced and therefore, it will be an AVL tree produced after inserting 12 into it.



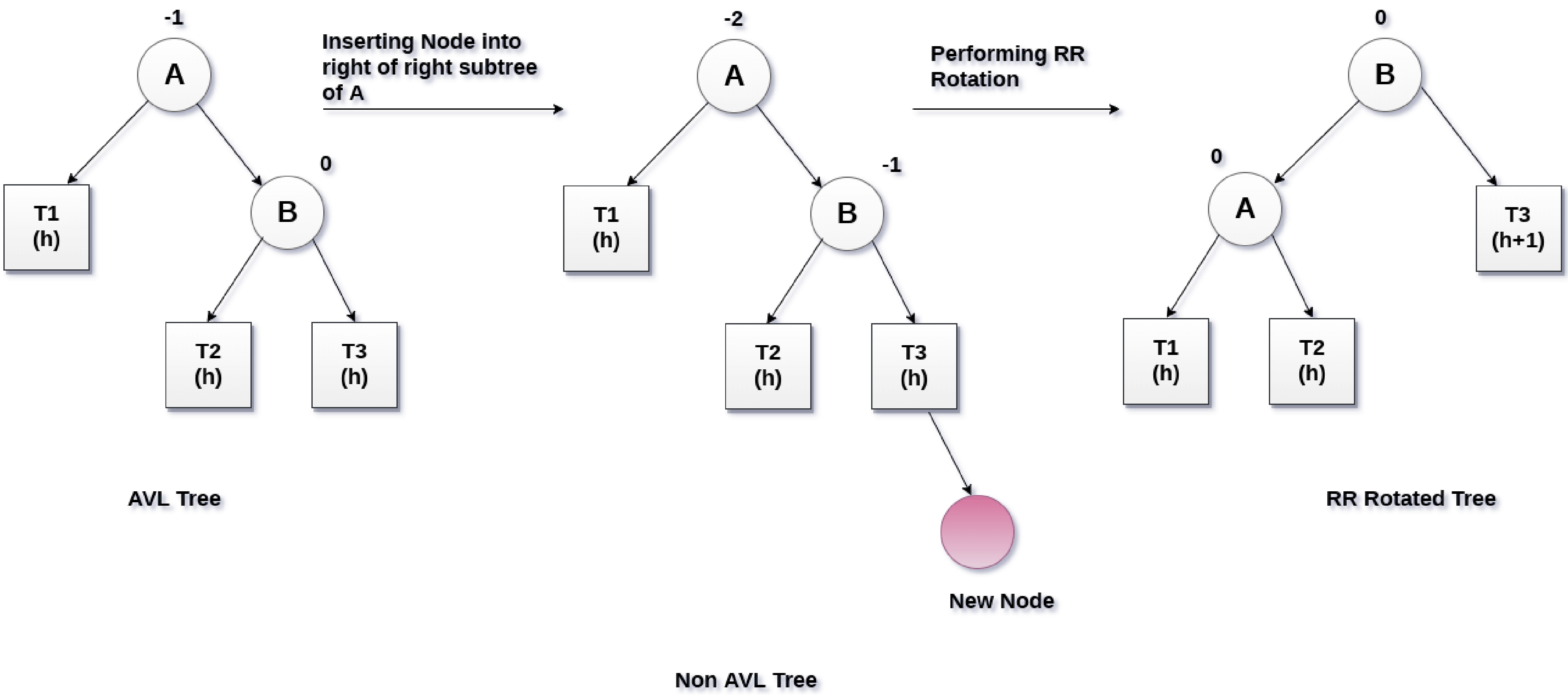
AVL Tree

Non - AVL Tree

LL Rotated Tree

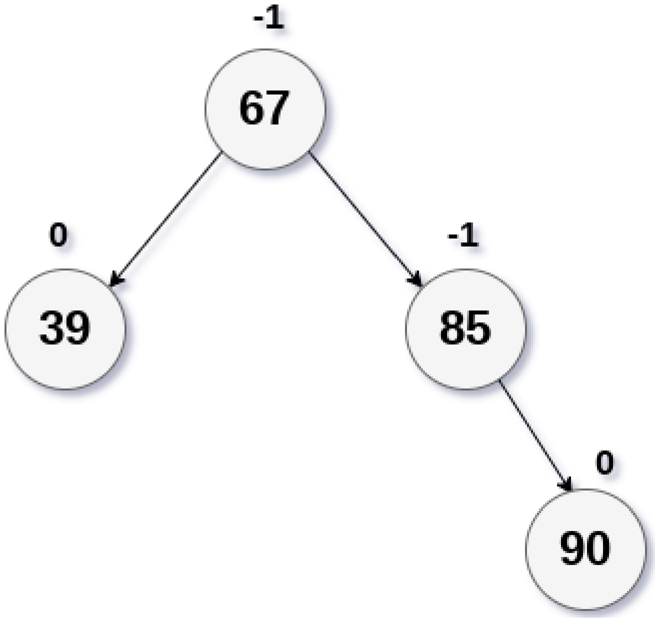
RR Rotation

- If the node is inserted into the right of the right sub-tree of a node A and the tree becomes unbalanced then, in that case, RR rotation will be performed as shown in the following diagram.
- While the rotation, the node B becomes the root node of the tree. The critical node A will be moved to its left and becomes the left child of B.
- The sub-tree T3 becomes the right sub-tree of A. T1 and T2 becomes the left and right sub-tree of node A.


-1
A
0
B
T1
(h)
T2
(h)
T3
(h)
AVL Tree
Inserting Node into
right of right subtree
of A
-2
A
-1
B
T1
(h)
T2
(h)
T3
(h)
New Node
Non AVL Tree
Performing RR
Rotation
0
B
0
A
T1
(h)
T2
(h)
T3
(h+1)
RR Rotated Tree

Example

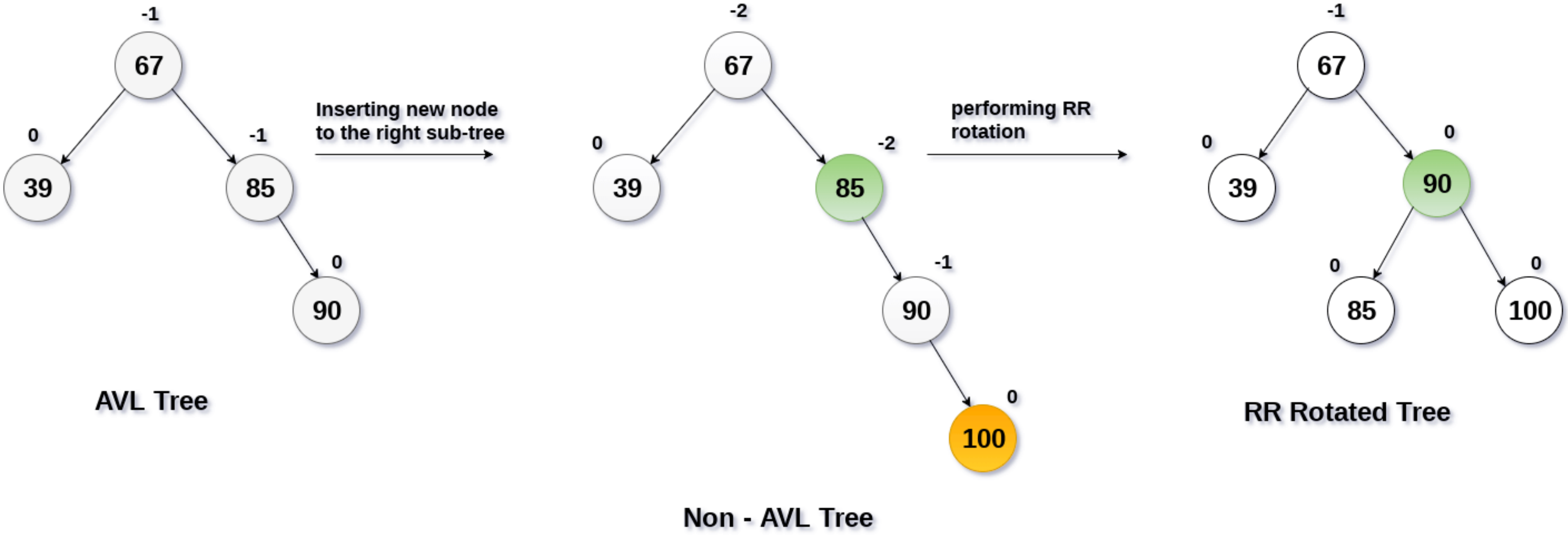
Insert 90 into the AVL Tree shown in the figure.



AVL Tree

Solution :

- 90 is inserted in to the right of the right sub-tree. In this case, critical node A will be 85, which is the closest ancestor to the new node, whose balance factor is disturbed. Therefore, we need to rebalance the tree by applying RR rotation onto it.
- The node B will be the node 90 , which will become the root node of this sub-tree. The critical node 85 will become its left child, in order to produce the rebalanced tree which is now an AVL tree.



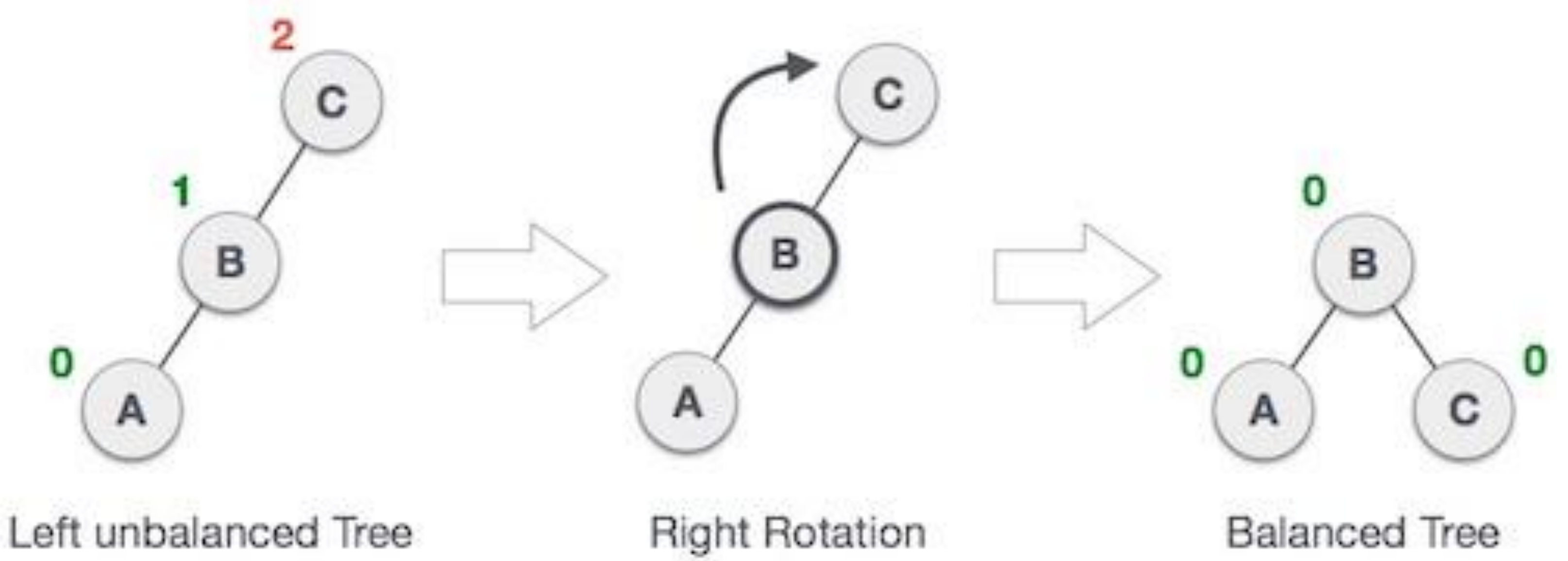
AVL Tree

Non - AVL Tree

RR Rotated Tree

Right Rotation

AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.



LR Rotation

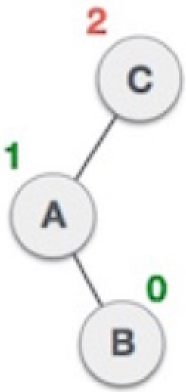
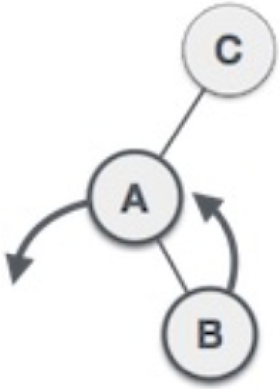
LR rotation is to be performed if the new node is inserted into the right of the left sub-tree of node A.

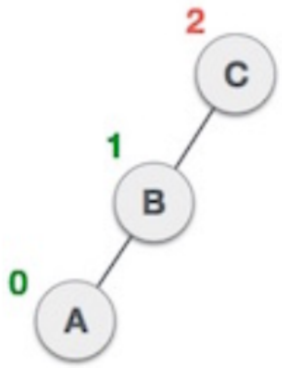
In LR rotation, node C (as shown in the figure) becomes the root node of the tree, while the node B and A becomes its left and right child respectively.

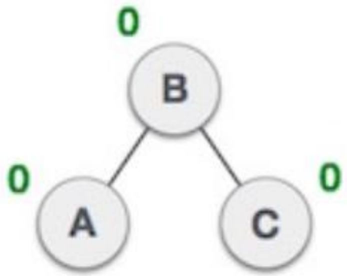
T1 and T2 becomes the left and right sub-tree of Node B respectively whereas, T3 and T4 becomes the left and right sub-tree of Node A.

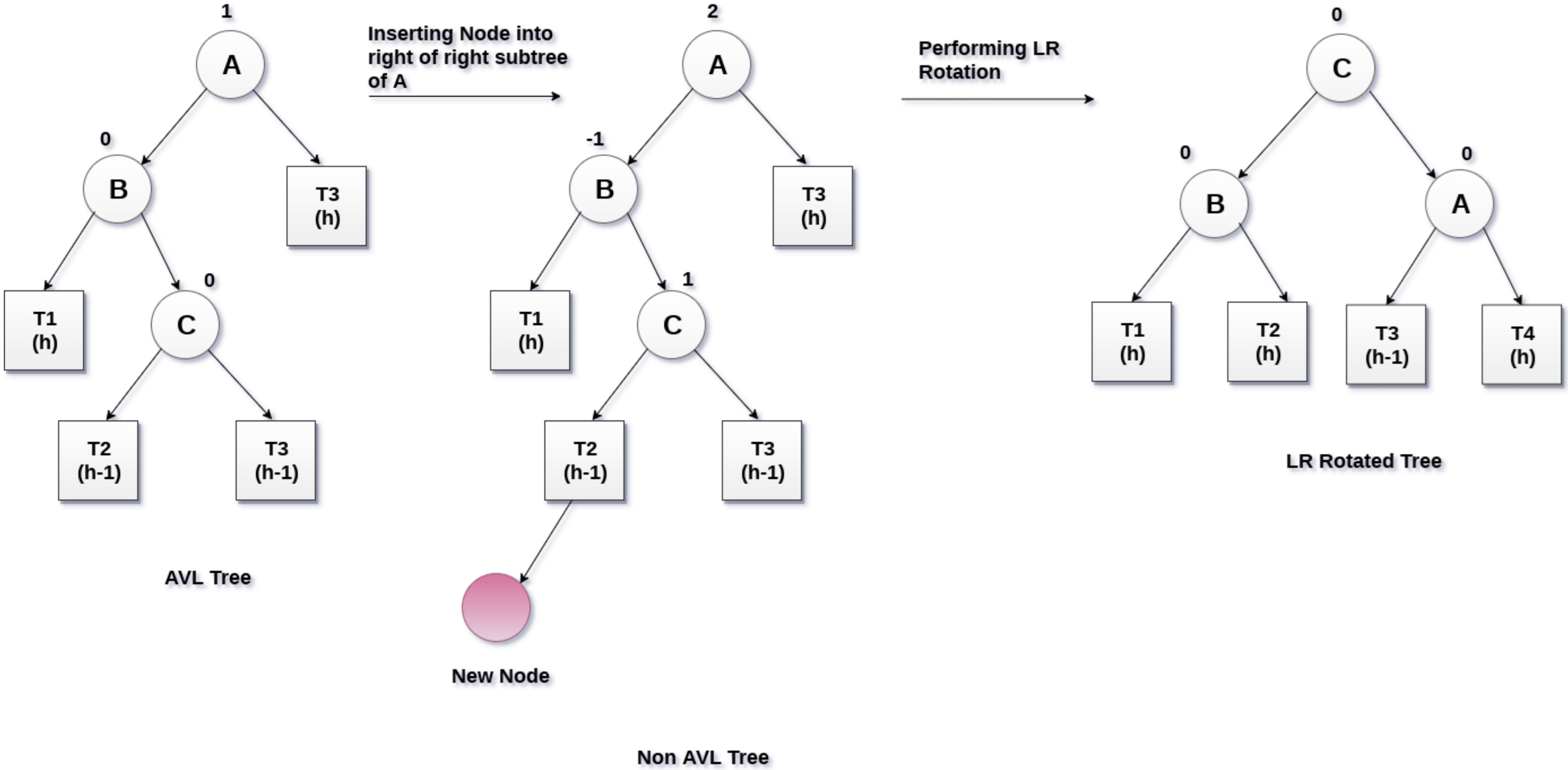
Left-Right Rotation

Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

State	Action
	A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.
	We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**.

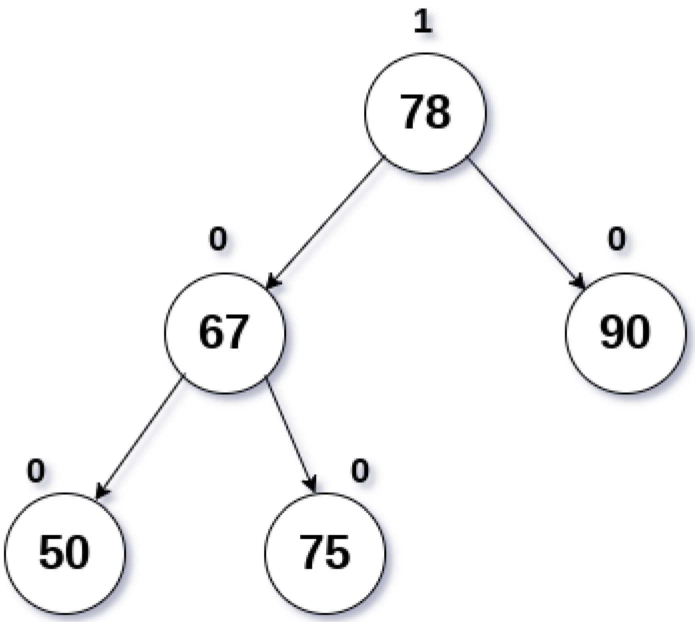
	Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree.

	The tree is now balanced.


1
A
0
B
T3
(h)
T1
(h)
0
C
T2
(h-1)
T3
(h-1)
AVL Tree
Inserting Node into right of right subtree of A
2
A
-1
B
T3
(h)
T1
(h)
1
C
T2
(h-1)
T3
(h-1)
New Node
Non AVL Tree
Performing LR Rotation
0
C
0
B
0
A
T1
(h)
T2
(h)
T3
(h-1)
T4
(h)
LR Rotated Tree

Example:

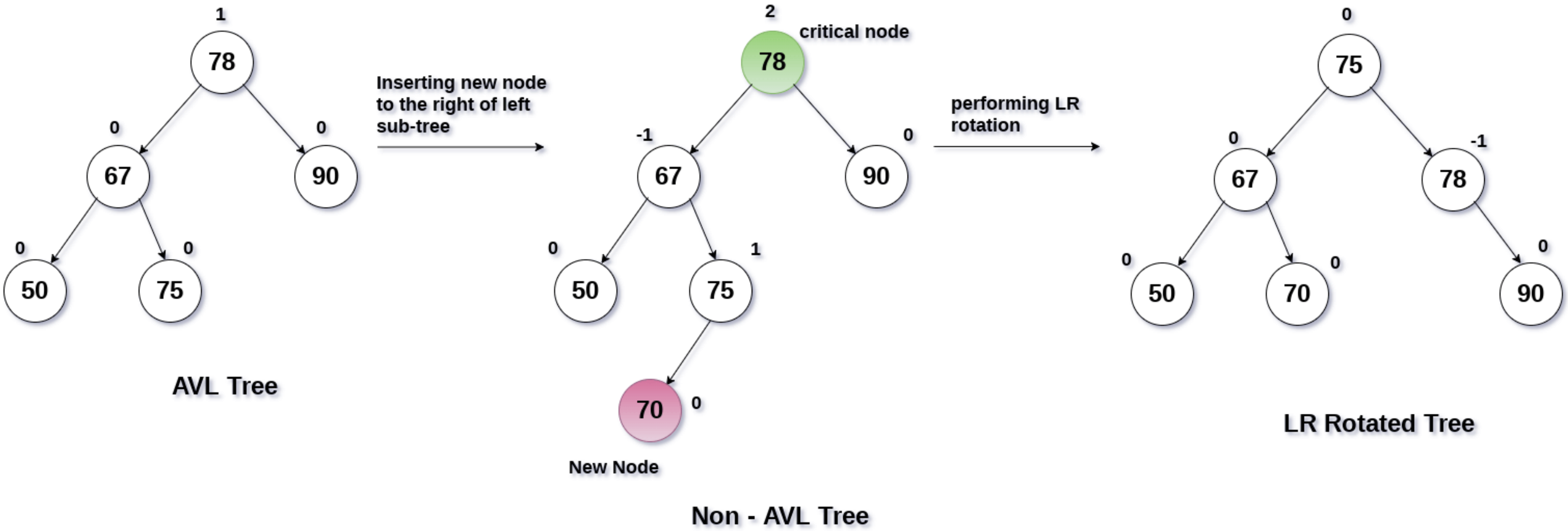
Insert the node with value 70 into the tree shown in the following data structure.



AVL Tree

Solution

- According to the property of binary search tree, the node with value 70 is inserted into the right of the left sub-tree of the root of tree.
- As shown in the figure, the balance factor of the root node disturbed upon inserting 70 and this becomes the critical node A.
- To rebalance the tree, LR rotation is to be performed. Node C i.e. 75 will become the new root node of the tree with B and A as its left and right child respectively.
- Sub-trees T1, T2 become the left and right sub-tree of B while sub-trees T3, T4 become the left and right sub-tree of A.
- The procedure is shown in the following figure.



AVL Tree

Non - AVL Tree

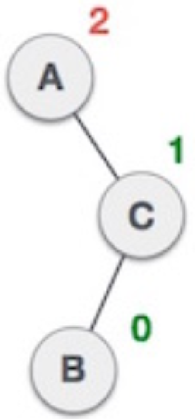
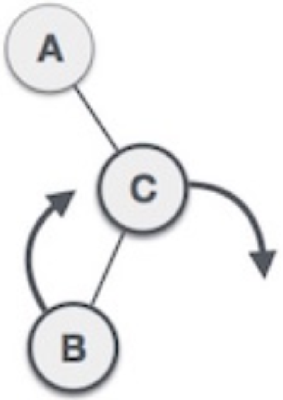
LR Rotated Tree

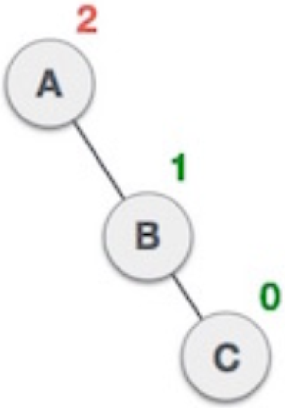
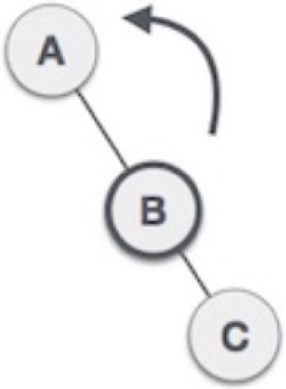
RL Rotation

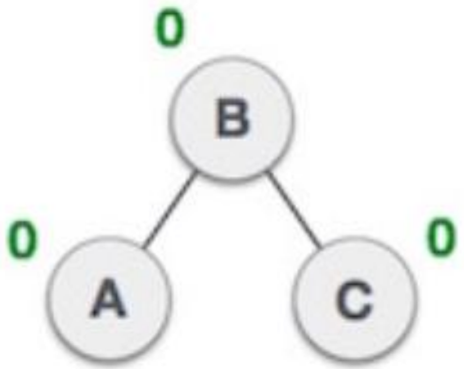
- RL rotations is to be performed if the new node is inserted into the left of right sub-tree of the critical node A. Let us consider, Node B is the root of the right sub-tree of the critical node, Node C is the root of the sub-tree in which the new node is inserted.
- Let T1 be the left sub-tree of the critical node A, T2 and T3 be the left and right sub-tree of Node C respectively, sub-tree T4 be the right sub-tree of Node B.
- Since, RL rotation is the mirror image of LR rotation. In this rotation, the node C becomes the root node of the tree with A and B as its left and right children respectively. Sub-trees T1 and T2 becomes the left and right sub-trees of A whereas, T3 and T4 becomes the left and right sub-trees of B.

Right-Left Rotation

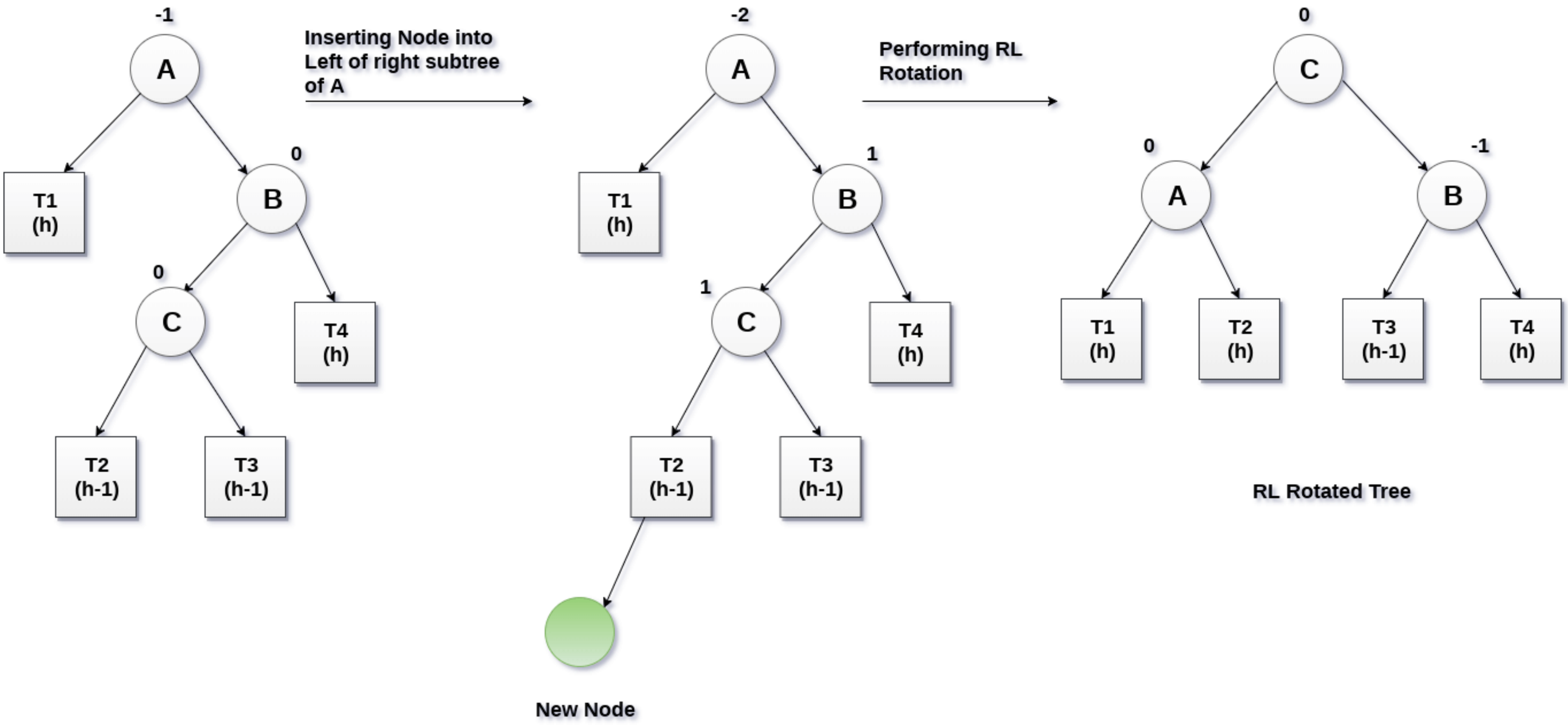
The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

State	Action
	A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2.
	First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**.

	Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation.
	A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**.

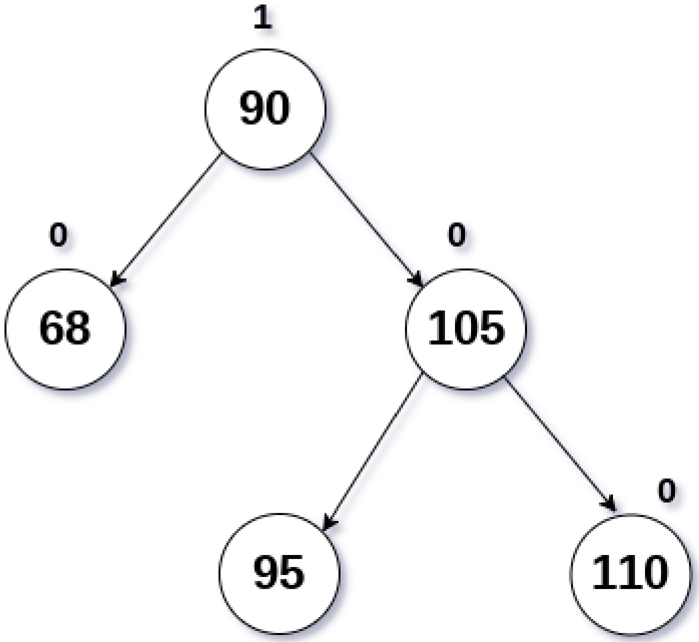
	The tree is now balanced.

The process of RL rotation is shown in the following image.



Example

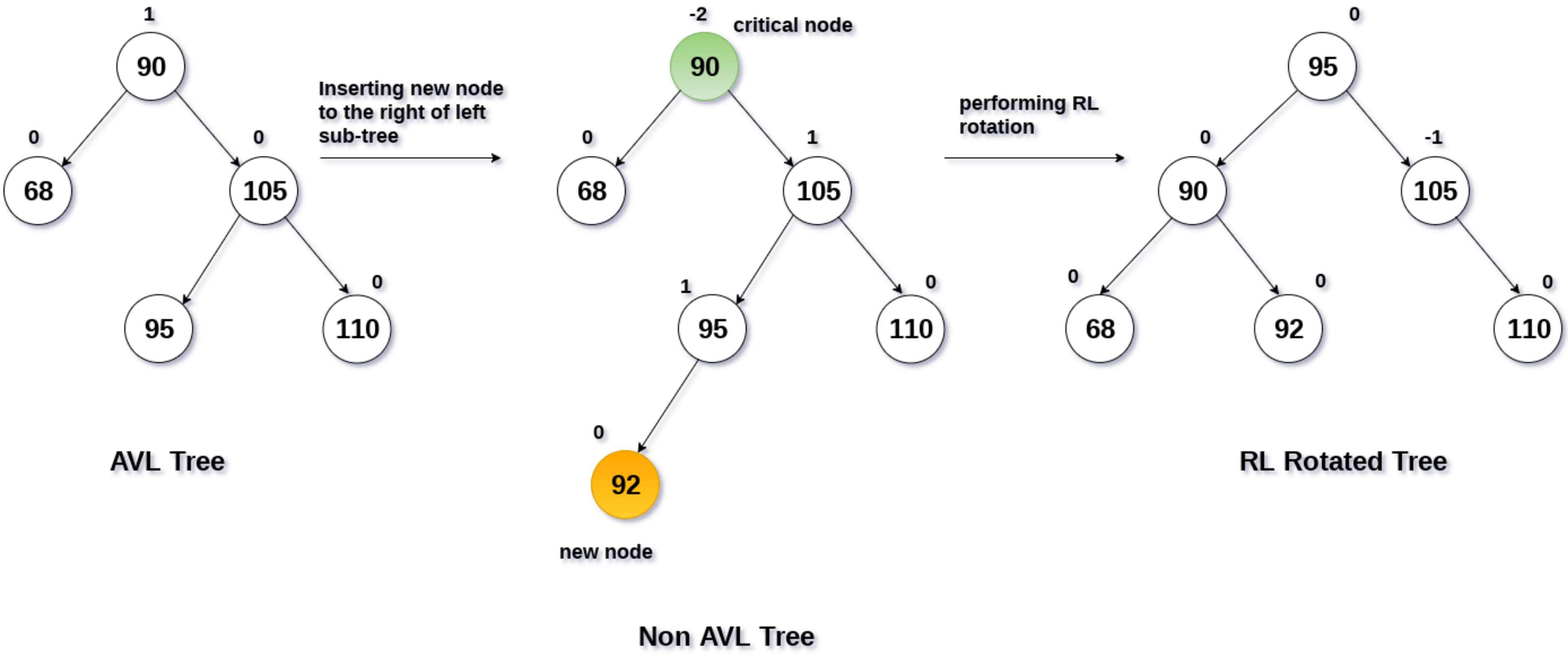
Insert node with the value 92 into the tree shown in the following figure.



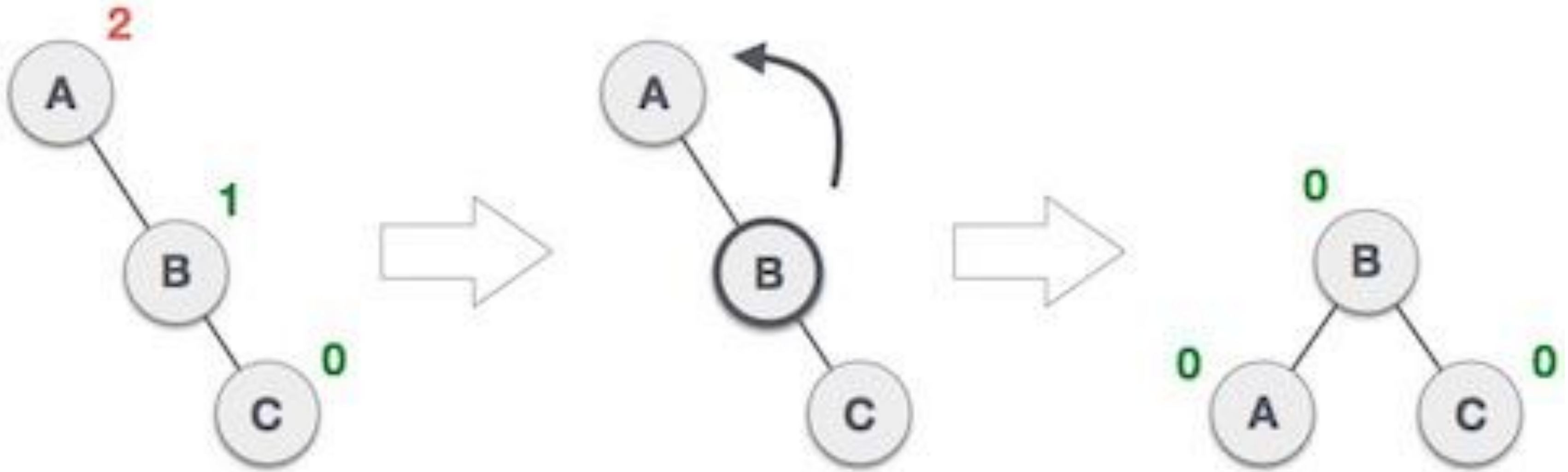
AVL Tree

Solution :

inserting 92 disturbs the balance factor of the node 92 and it becomes the critical node A with 105 as the node B and 95 with the node C. In RL rotation, C becomes the root of the tree (as shown in the figure) with node A (90) and B (105) as its left and right children respectively. The tree will be rotated as shown in the figure.



Right Rotation And Left Rotation



Right unbalanced tree

Left Rotation

Balanced

AVL Insertion Process

You will do an insertion similar to a normal Binary Search Tree insertion. After inserting, you fix the AVL property using left or right rotations.

- If there is an imbalance in left child of right subtree, then you perform a left-right rotation.
- If there is an imbalance in left child of left subtree, then you perform a right rotation.
- If there is an imbalance in right child of right subtree, then you perform a left rotation.
- If there is an imbalance in right child of left subtree, then you perform a right-left rotation.

An AVL tree is a self-balancing binary search tree. An AVL tree is a binary search tree which has the following properties: ->The sub-trees of every node differ in height by at most one. ->Every sub-tree is an AVL tree.

AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the Balance Factor. The height of an AVL tree is always O(Logn) where n is the number of nodes in the tree.

Insertion

Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. The new node is added into AVL tree as the leaf node. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing.

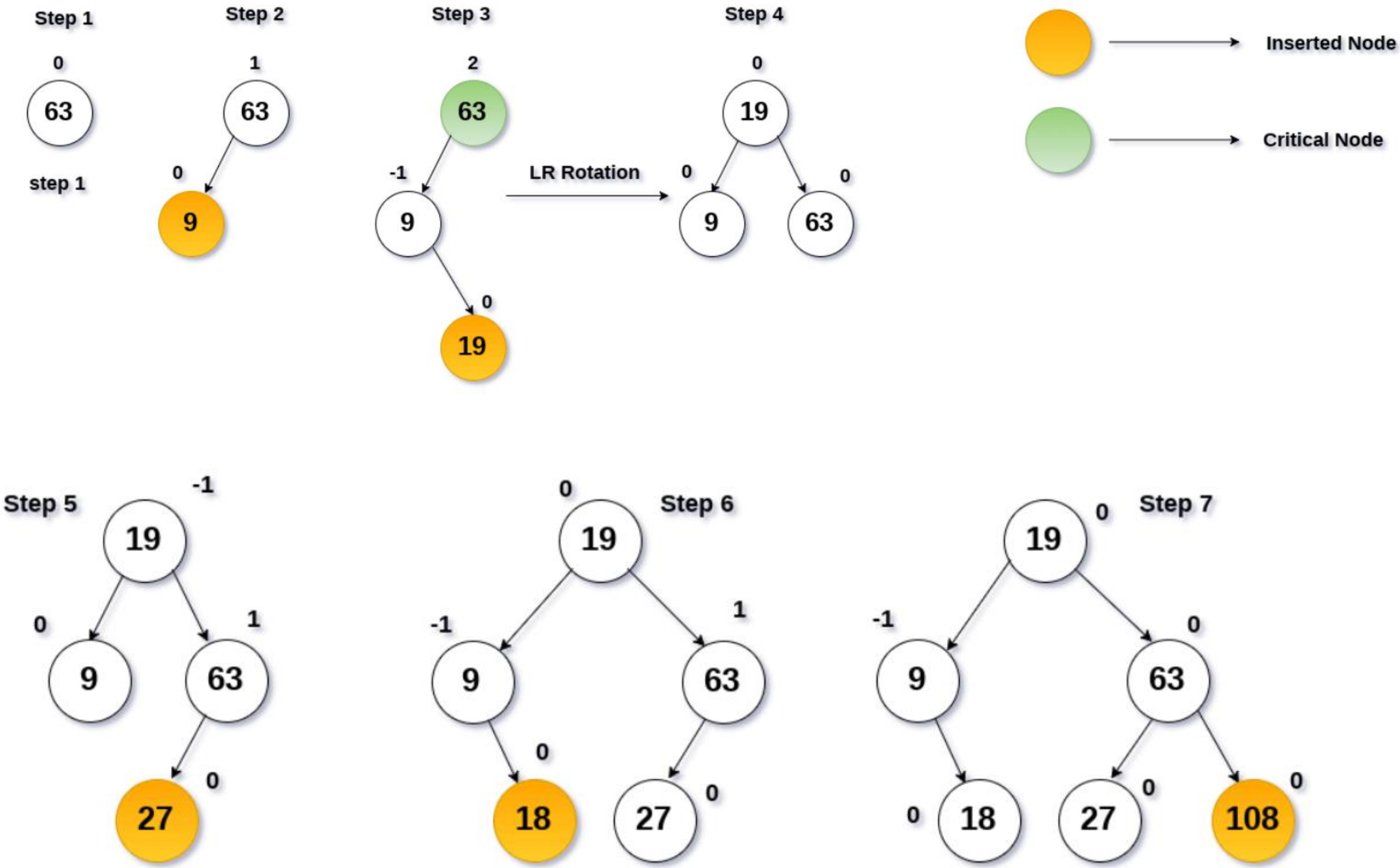
The tree can be balanced by applying rotations. Rotation is required only if, the balance factor of any node is disturbed upon inserting the new node, otherwise the rotation is not required.

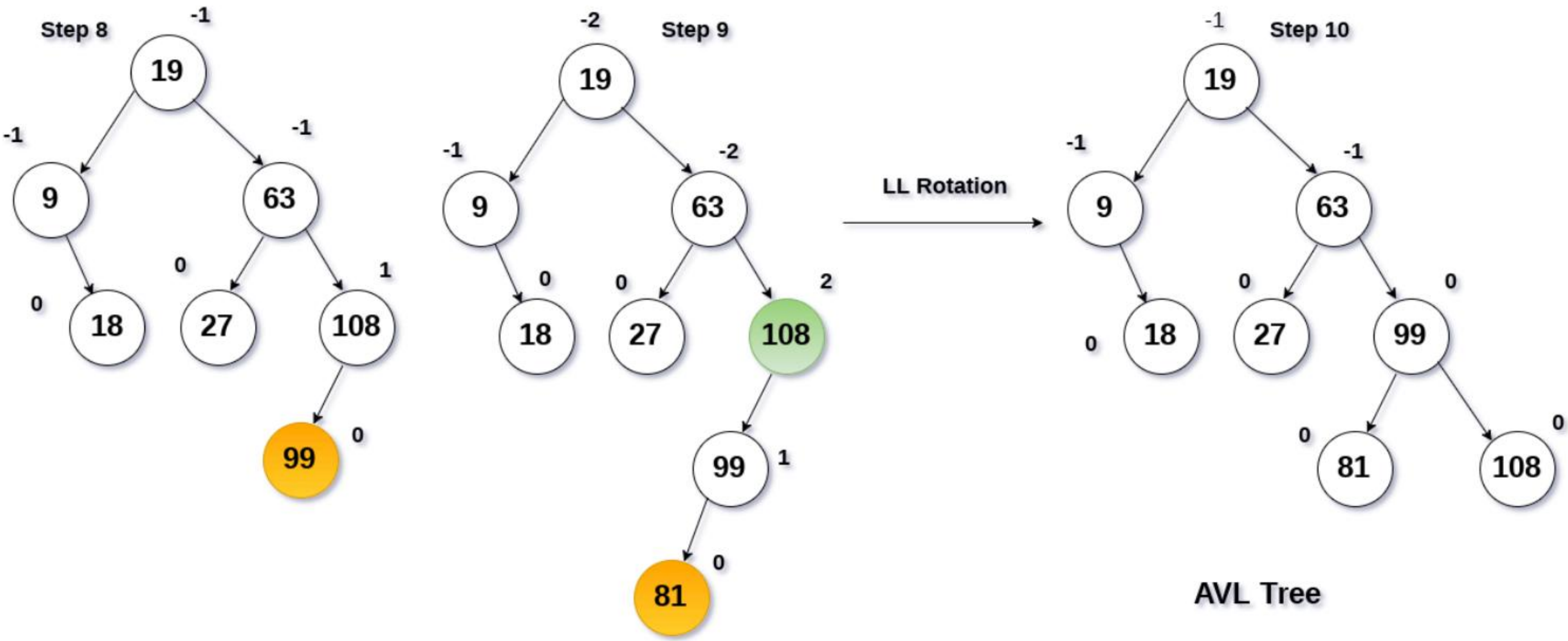
Depending upon the type of insertion, the Rotations are categorized into four categories.

Construct an AVL tree by inserting the following elements in the given order.

63, 9, 19, 27, 18, 108, 99, 81

- The process of constructing an AVL tree from the given set of elements is shown in the following figure.
- At each step, we must calculate the balance factor for every node, if it is found to be more than 2 or less than -2, then we need a rotation to rebalance the tree. The type of rotation will be estimated by the location of the inserted element with respect to the critical node.
- All the elements are inserted in order to maintain the order of binary search tree.


Step 1
0
63
step 1
Step 2
1
63
0
9
Step 3
2
63
-1
9
0
19
LR Rotation
Step 4
0
19
0
9
0
63
Inserted Node
Critical Node
Step 5
-1
19
0
9
1
63
27
0
Step 6
0
19
-1
9
1
63
0
18
27
0
Step 7
19
0
-1
9
0
63
0
18
27
0
108
0


Step 8
-1
19
-1
9
-1
63
0
18
0
27
1
108
0
99
Step 9
-2
19
-1
9
-2
63
0
18
0
27
2
108
1
99
0
81
LL Rotation
Step 10
-1
19
-1
9
-1
63
0
18
0
27
0
99
0
81
0
108

AVL Tree

Deletion in AVL Tree

Deleting a node from an AVL tree is similar to that in a binary search tree. Deletion may disturb the balance factor of an AVL tree and therefore the tree needs to be rebalanced in order to maintain the AVLness. For this purpose, we need to perform rotations. The two types of rotations are L rotation and R rotation. Here, we will discuss R rotations. L rotations are the mirror images of them.

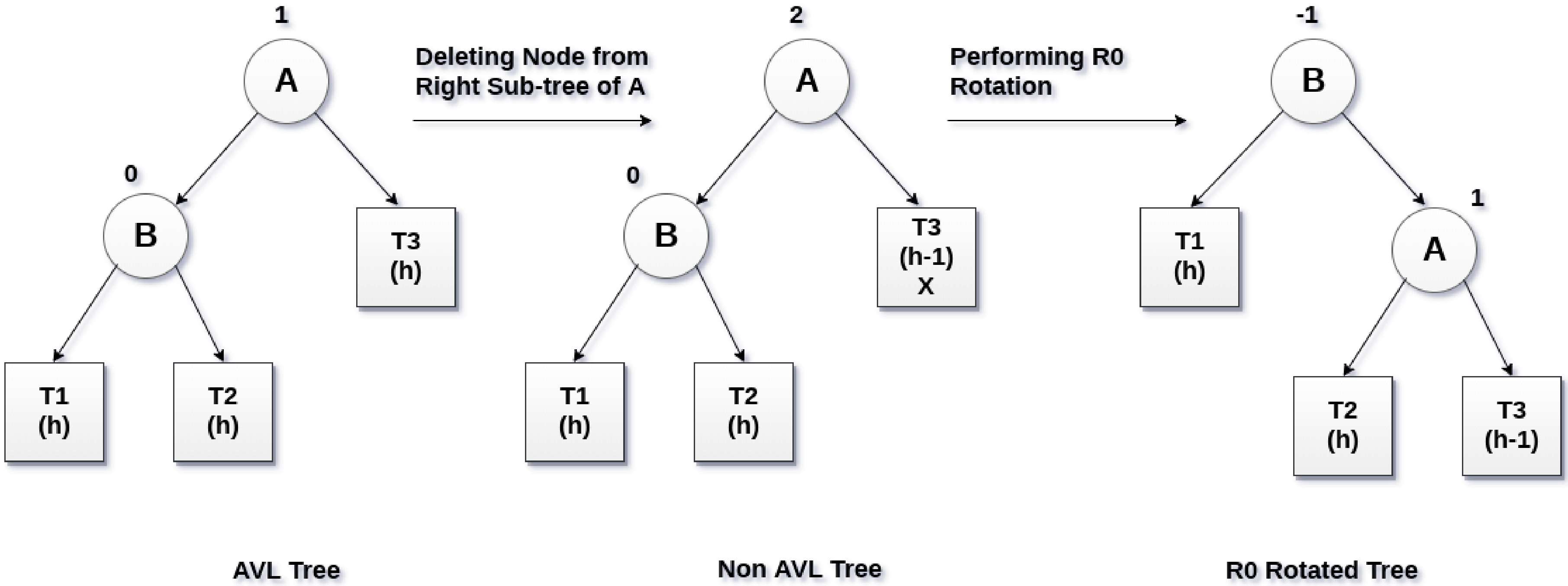
If the node which is to be deleted is present in the left sub-tree of the critical node, then L rotation needs to be applied else if, the node which is to be deleted is present in the right sub-tree of the critical node, the R rotation will be applied.

Let us consider that, A is the critical node and B is the root node of its left sub-tree. If node X, present in the right sub-tree of A, is to be deleted, then there can be three different situations:

R0 rotation (Node B has balance factor 0)

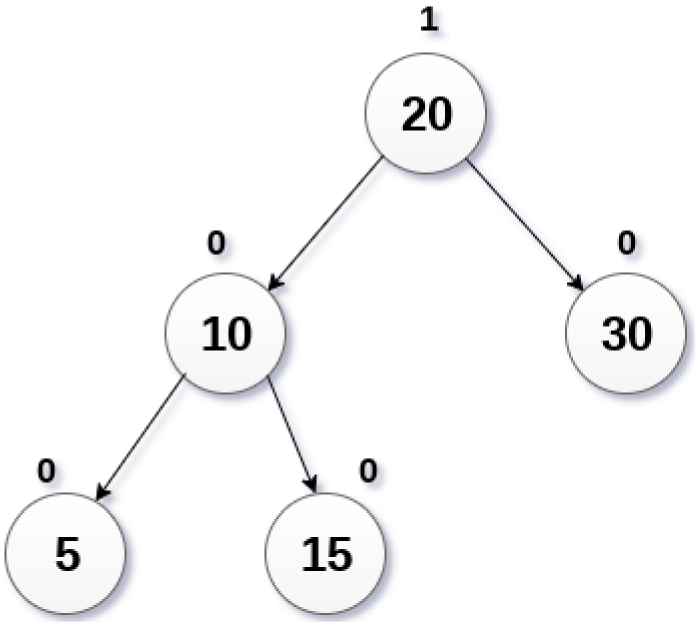
If the node B has 0 balance factor, and the balance factor of node A disturbed upon deleting the node X, then the tree will be rebalanced by rotating tree using R0 rotation.

The critical node A is moved to its right and the node B becomes the root of the tree with T1 as its left sub-tree. The sub-trees T2 and T3 becomes the left and right sub-tree of the node A. the process involved in R0 rotation is shown in the following image.


1
A
0
B
T3
(h)
T1
(h)
T2
(h)
Deleting Node from
Right Sub-tree of A
2
A
0
B
T3
(h-1)
X
T1
(h)
T2
(h)
Performing R0
Rotation
-1
B
T1
(h)
1
A
T2
(h)
T3
(h-1)
AVL Tree
Non AVL Tree
R0 Rotated Tree

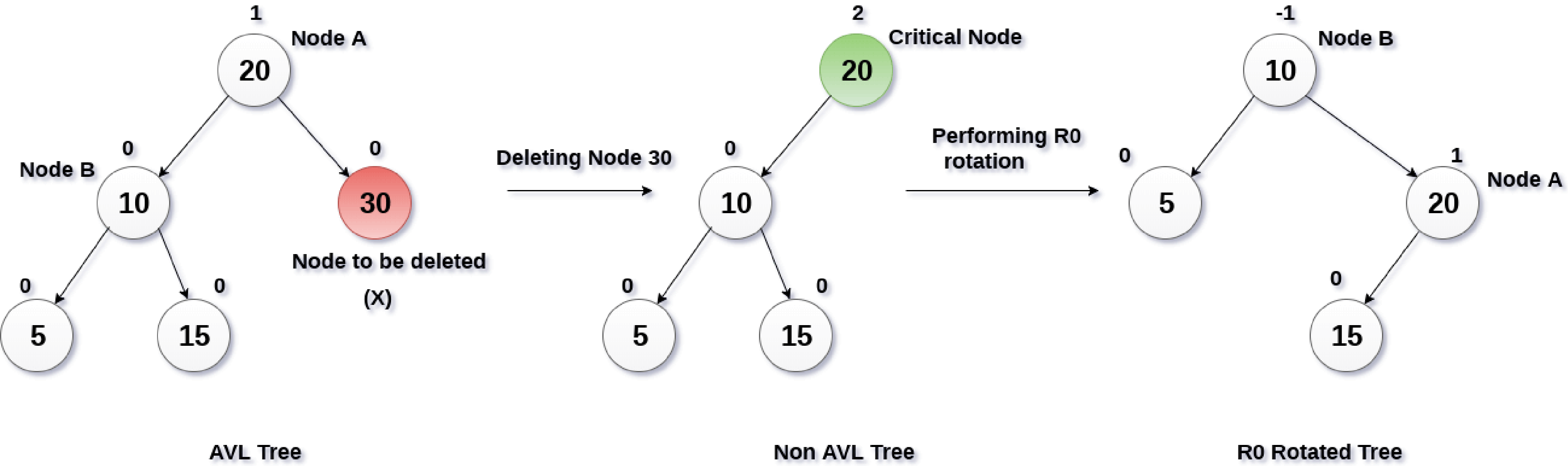
Example:

Delete the node 30 from the AVL tree shown in the following image.



Solution

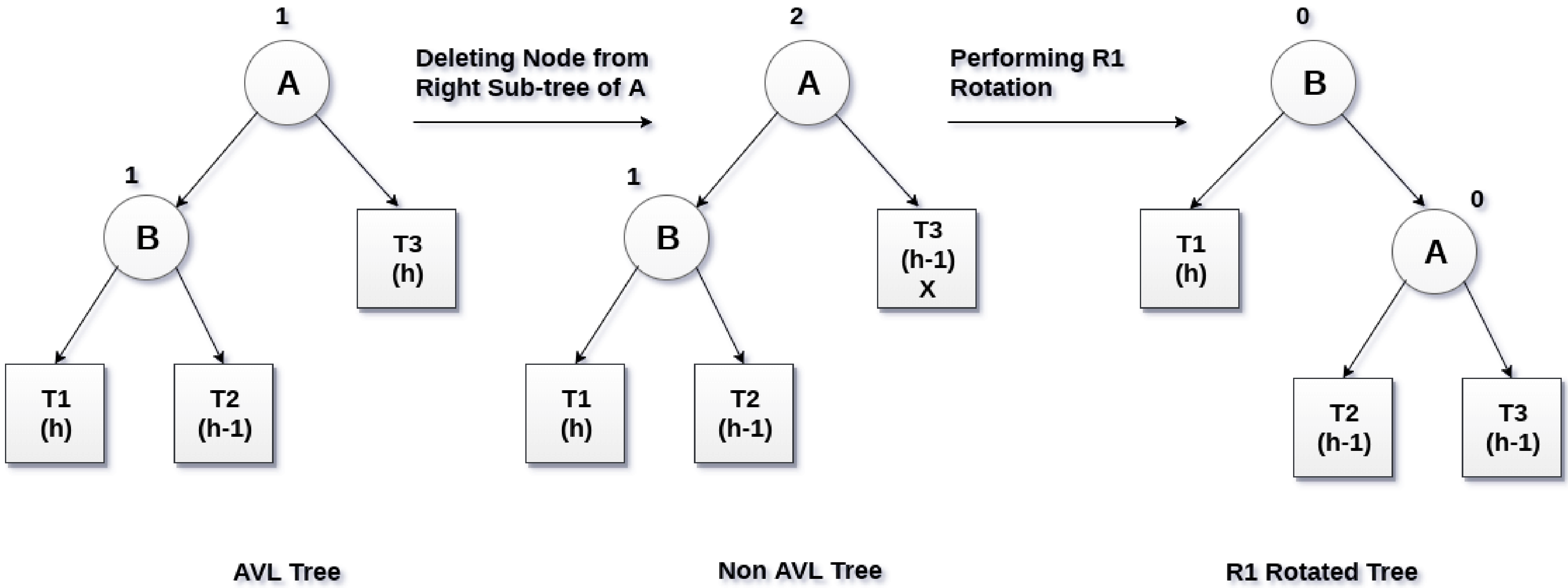
In this case, the node B has balance factor 0, therefore the tree will be rotated by using R0 rotation as shown in the following image. The node B(10) becomes the root, while the node A is moved to its right. The right child of node B will now become the left child of node A.


1
Node A
20
0
Node B
10
0
30
Node to be deleted
(X)
0
5
0
15
AVL Tree
Deleting Node 30
2
Critical Node
20
0
10
0
5
0
15
Non AVL Tree
Performing R0
rotation
-1
Node B
10
0
5
1
Node A
20
0
15
R0 Rotated Tree

R1 Rotation (Node B has balance factor 1)

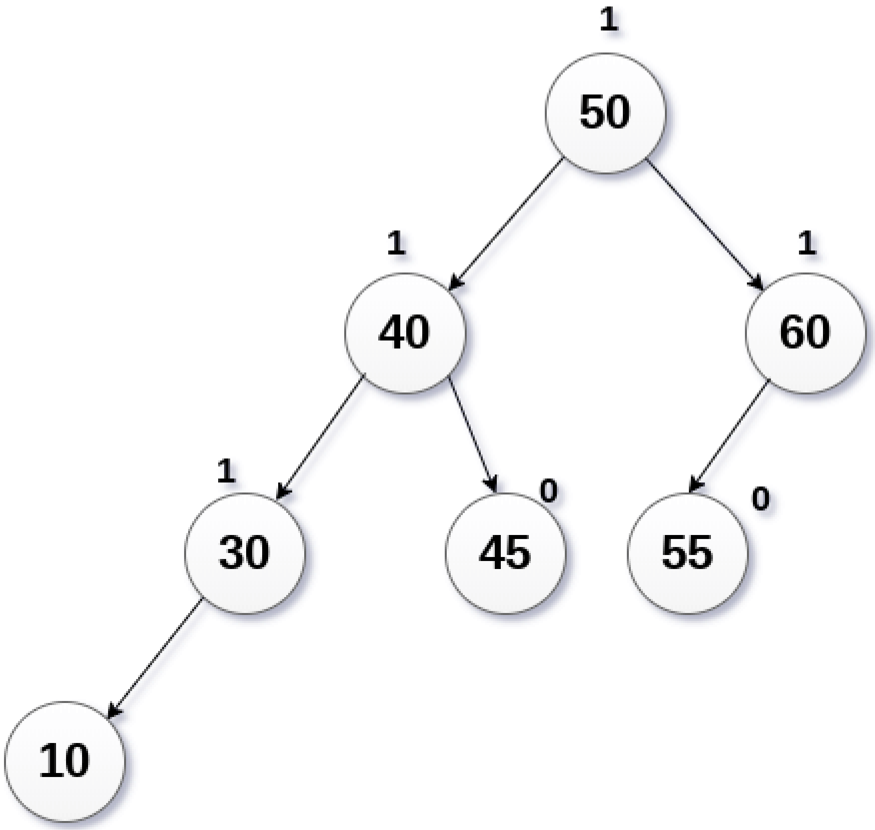
R1 Rotation is to be performed if the balance factor of Node B is 1. In R1 rotation, the critical node A is moved to its right having sub-trees T2 and T3 as its left and right child respectively. T1 is to be placed as the left sub-tree of the node B.

The process involved in R1 rotation is shown in the following image.


1
A
1
B
T1
(h)
T2
(h-1)
T3
(h)
Deleting Node from
Right Sub-tree of A
2
A
1
B
T1
(h)
T2
(h-1)
T3
(h-1)
X
Performing R1
Rotation
0
B
T1
(h)
0
A
T2
(h-1)
T3
(h-1)
AVL Tree
Non AVL Tree
R1 Rotated Tree

Example

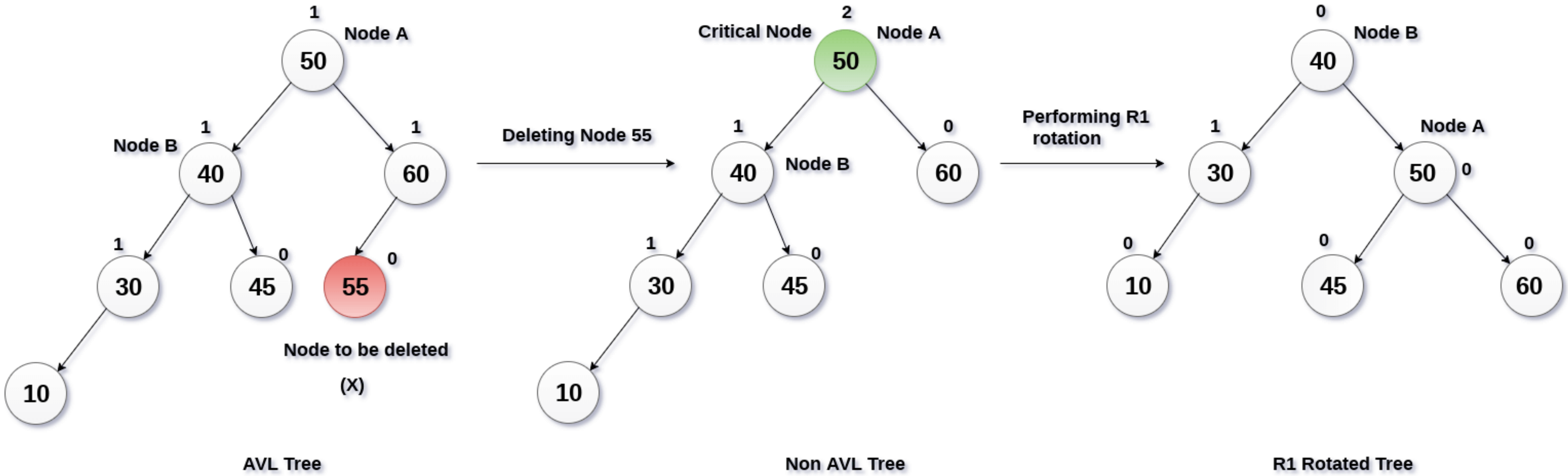
Delete Node 55 from the AVL tree shown in the following image.



AVL Tree

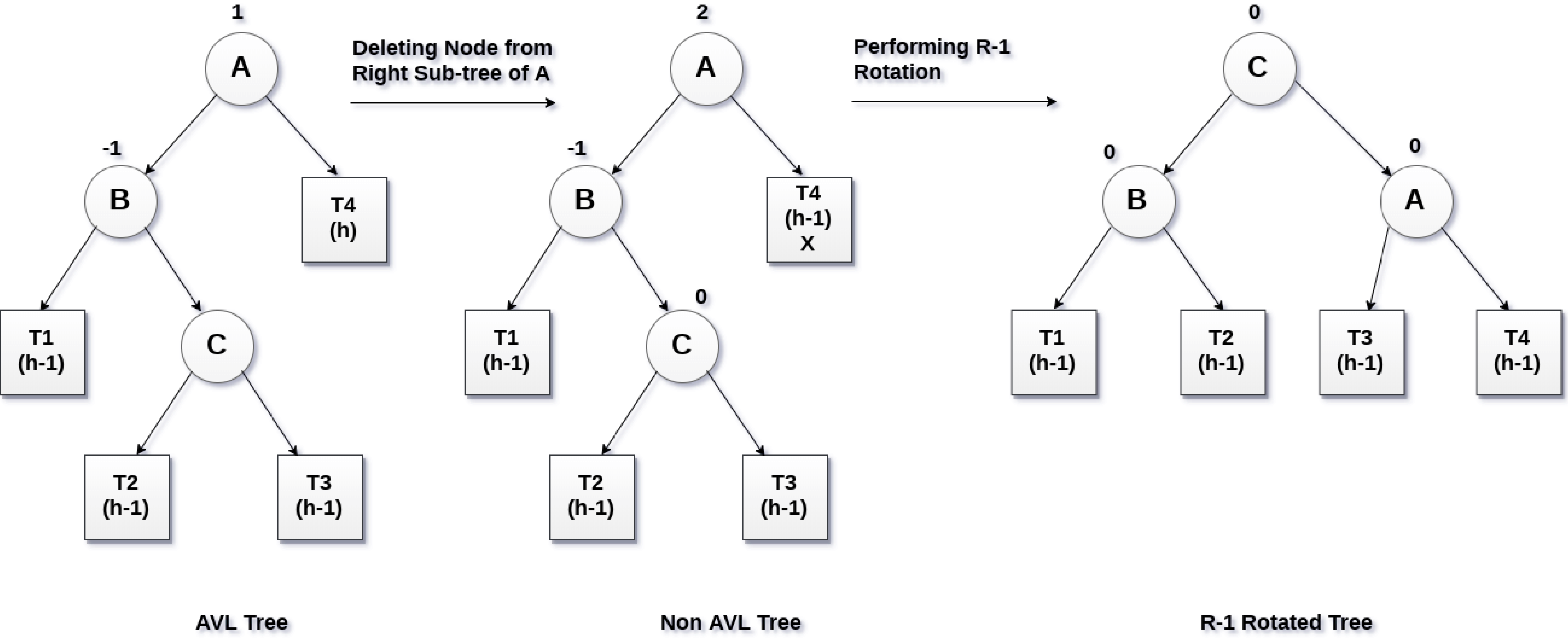
Solution :

Deleting 55 from the AVL Tree disturbs the balance factor of the node 50 i.e. node A which becomes the critical node. This is the condition of R1 rotation in which, the node A will be moved to its right (shown in the image below). The right of B is now become the left of A (i.e. 45). The process involved in the solution is shown in the following image.


1
Node A
50
Node B
1
40
1
60
1
30
45
0
55
0
Node to be deleted
(X)
10
AVL Tree
Deleting Node 55
Critical Node
2
Node A
50
1
40
Node B
0
60
1
30
45
0
10
Non AVL Tree
Performing R1
rotation
0
Node B
40
1
30
Node A
50
0
0
10
0
45
0
60
R1 Rotated Tree

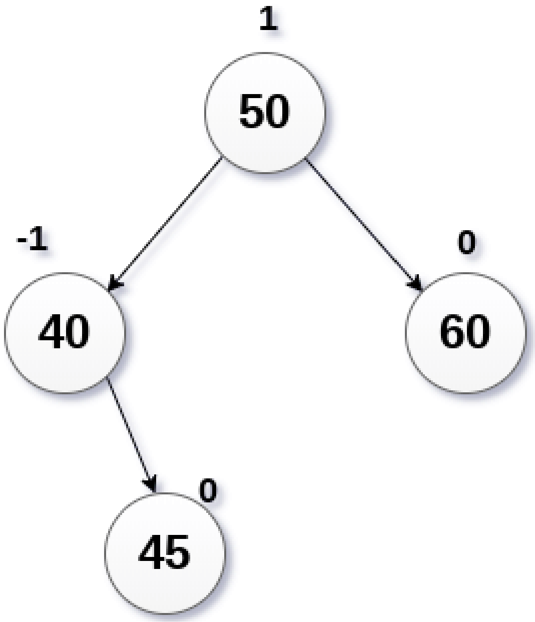
R-1 Rotation (Node B has balance factor -1)

- R-1 rotation is to be performed if the node B has balance factor -1. This case is treated in the same way as LR rotation. In this case, the node C, which is the right child of node B, becomes the root node of the tree with B and A as its left and right children respectively.
- The sub-trees T1, T2 becomes the left and right sub-trees of B whereas, T3, T4 become the left and right sub-trees of A.
- The process involved in R-1 rotation is shown in the following image.


1
A
-1
B
T4
(h)
T1
(h-1)
C
T2
(h-1)
T3
(h-1)
AVL Tree
Deleting Node from
Right Sub-tree of A
2
A
-1
B
T4
(h-1)
X
T1
(h-1)
0
C
T2
(h-1)
T3
(h-1)
Non AVL Tree
Performing R-1
Rotation
0
C
0
B
0
A
T1
(h-1)
T2
(h-1)
T3
(h-1)
T4
(h-1)
R-1 Rotated Tree

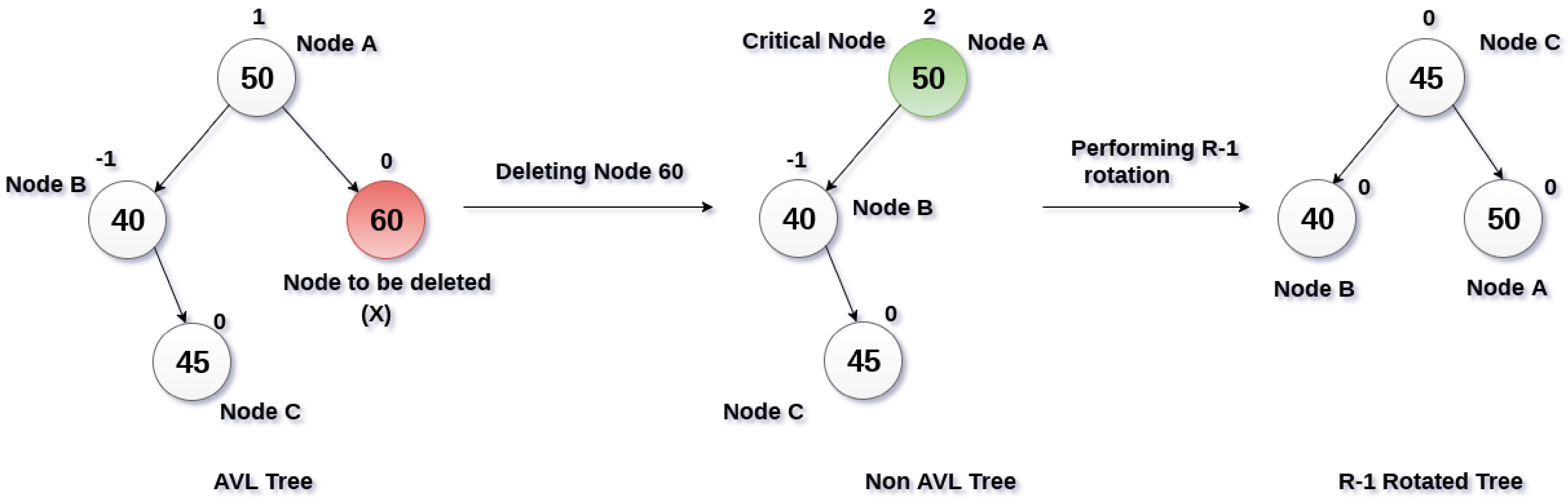
Example

Delete the node 60 from the AVL tree shown in the following image.



Solution:

in this case, node B has balance factor -1. Deleting the node 60, disturbs the balance factor of the node 50 therefore, it needs to be R-1 rotated. The node C i.e. 45 becomes the root of the tree with the node B(40) and A(50) as its left and right child.


1
Node A
50
Node B
-1
40
0
60
Node to be deleted
(X)
0
45
Node C
AVL Tree
Deleting Node 60
Critical Node
2
50
Node A
-1
40
Node B
0
45
Node C
Non AVL Tree
Performing R-1
rotation
0
45
Node C
40
0
50
0
Node B
Node A
R-1 Rotated Tree

Why AVL Tree ?

AVL tree controls the height of the binary search tree by not letting it to be skewed. The time taken for all operations in a binary search tree of height h is **O(h)**. However, it can be extended to **O(n)** if the BST becomes skewed (i.e. worst case). By limiting this height to log n, AVL tree imposes an upper bound on each operation to be **O(log n)** where n is the number of nodes.