

## UNIT – III

### Search Trees:

Binary Search Trees, Definition, Implementation, Operations- Searching, Insertion and Deletion, AVL Trees, Definition, Height of an AVL Tree, Operations – Insertion, Deletion and Searching, Red –Black, Splay Trees.

## Binary Search tree

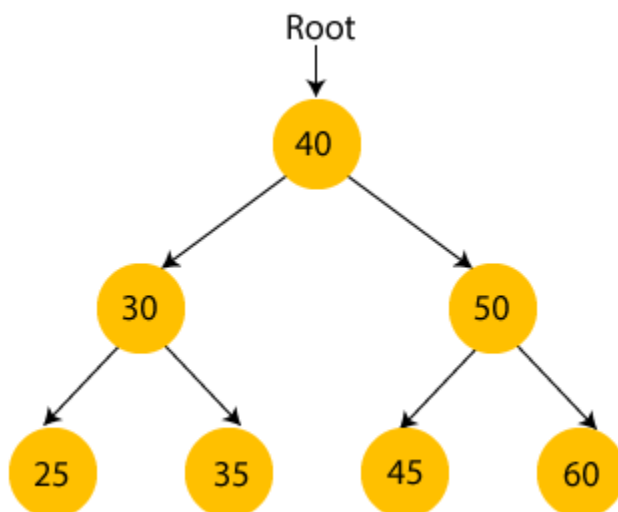
In this article, we will discuss the Binary search tree. This article will be very helpful and informative to the students with technical background as it is an important topic of their course.

Before moving directly to the binary search tree, let's first see a brief description of the tree.

### What is a Binary Search tree?

A binary search tree follows some order to arrange the elements. In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node. This rule is applied recursively to the left and right subtrees of the root.

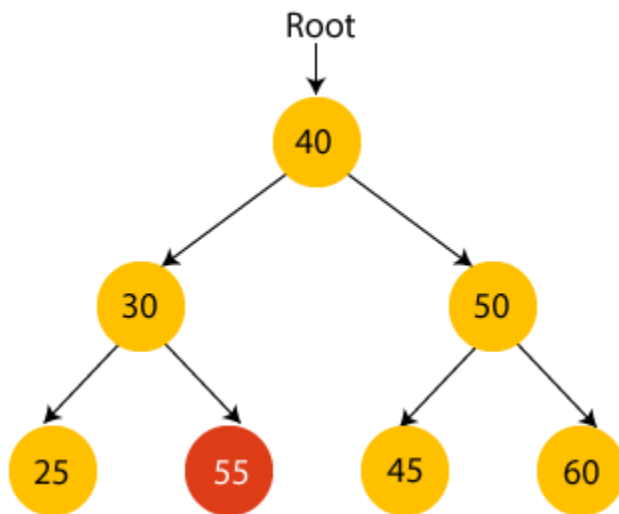
Let's understand the concept of Binary search tree with an example.



In the above figure, we can observe that the root node is 40, and all the nodes of the left subtree are smaller than the root node, and all the nodes of the right subtree are greater than the root node.

Similarly, we can see the left child of root node is greater than its left child and smaller than its right child. So, it also satisfies the property of binary search tree. Therefore, we can say that the tree in the above image is a binary search tree.

Suppose if we change the value of node 35 to 55 in the above tree, check whether the tree will be binary search tree or not.



In the above tree, the value of root node is 40, which is greater than its left child 30 but smaller than right child of 30, i.e., 55. So, the above tree does not satisfy the property of Binary search tree. Therefore, the above tree is not a binary search tree.

## Advantages of Binary search tree

- Searching an element in the Binary search tree is easy as we always have a hint that which subtree has the desired element.
- As compared to array and linked lists, insertion and deletion operations are faster in BST.

## Example of creating a binary search tree

Now, let's see the creation of binary search tree using an example.

Suppose the data elements are - **45, 15, 79, 90, 10, 55, 12, 20, 50**

- First, we have to insert **45** into the tree as the root of the tree.
- Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.
- Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.

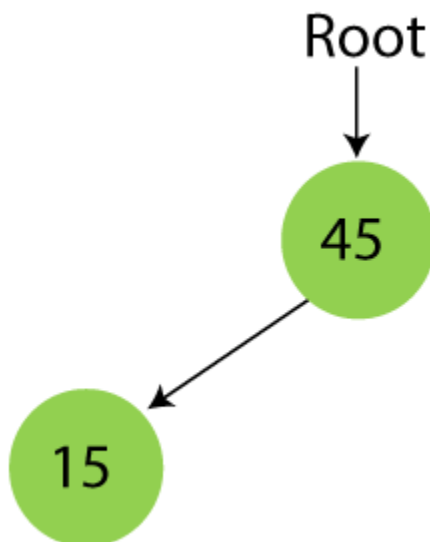
Now, let's see the process of creating the Binary search tree using the given data element. The process of creating the BST is shown below -

**Step 1 - Insert 45.**



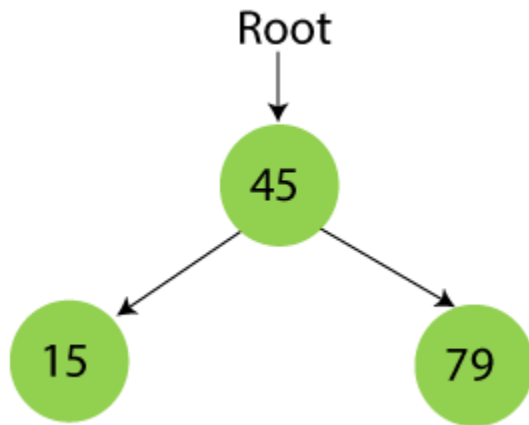
**Step 2 - Insert 15.**

As 15 is smaller than 45, so insert it as the root node of the left subtree.



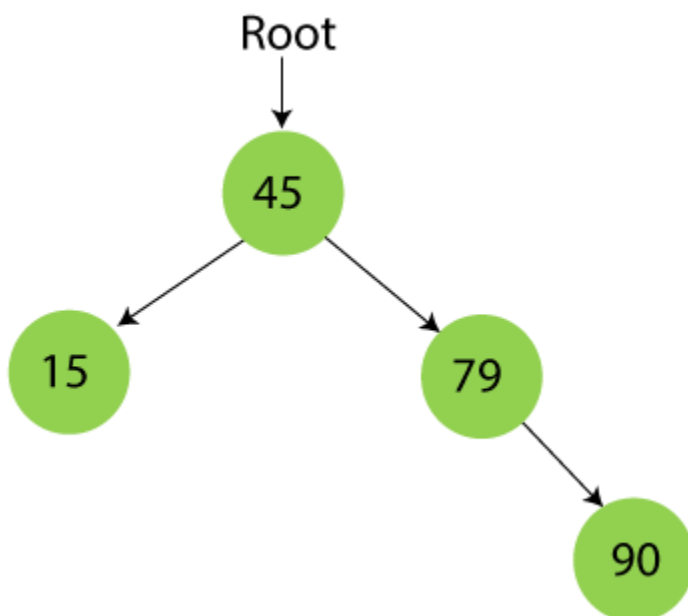
**Step 3 - Insert 79.**

As 79 is greater than 45, so insert it as the root node of the right subtree.



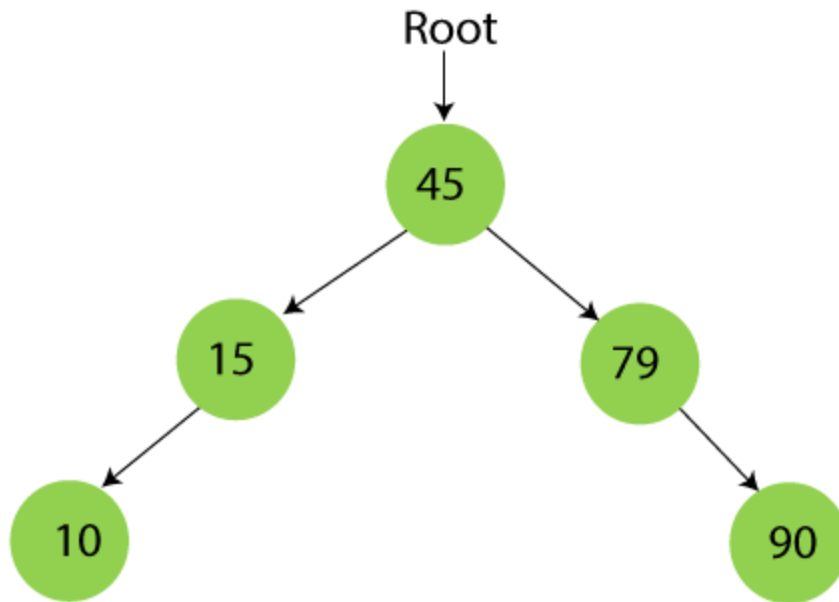
**Step 4 - Insert 90.**

90 is greater than 45 and 79, so it will be inserted as the right subtree of 79.



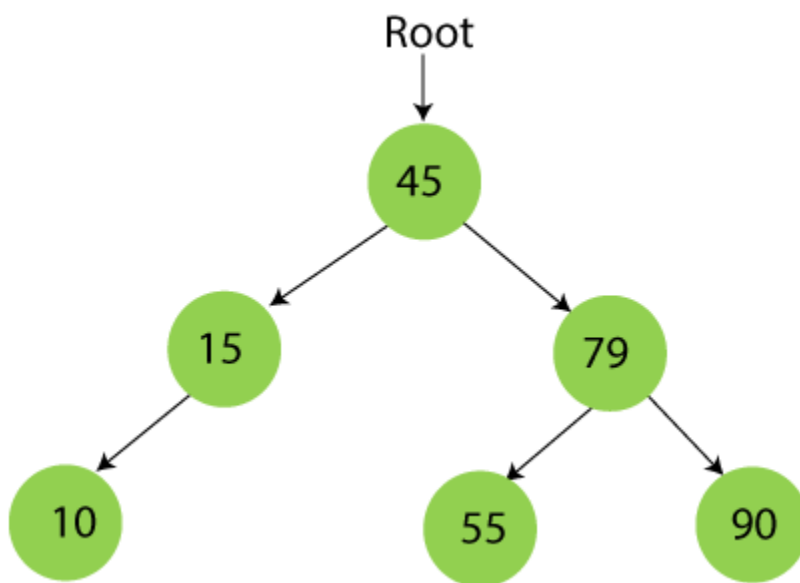
**Step 5 - Insert 10.**

10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.



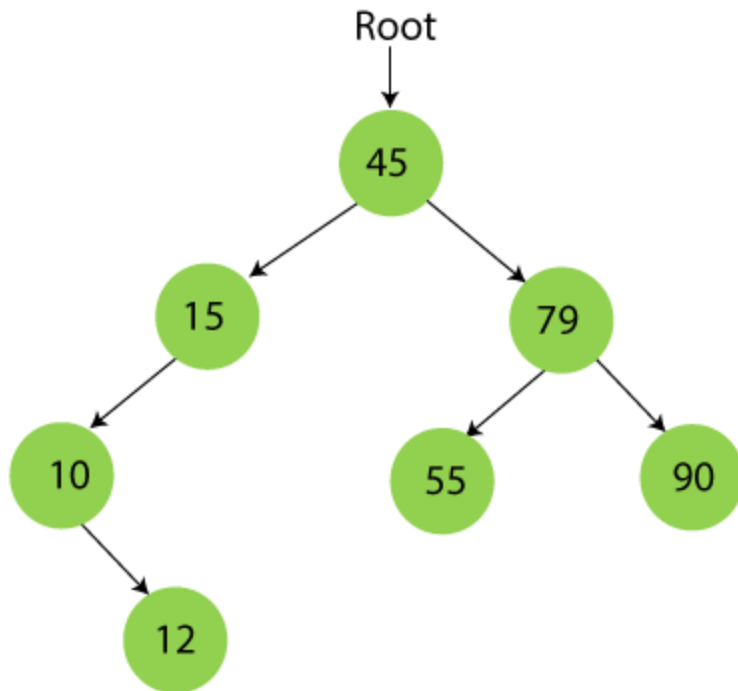
**Step 6 - Insert 55.**

55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.



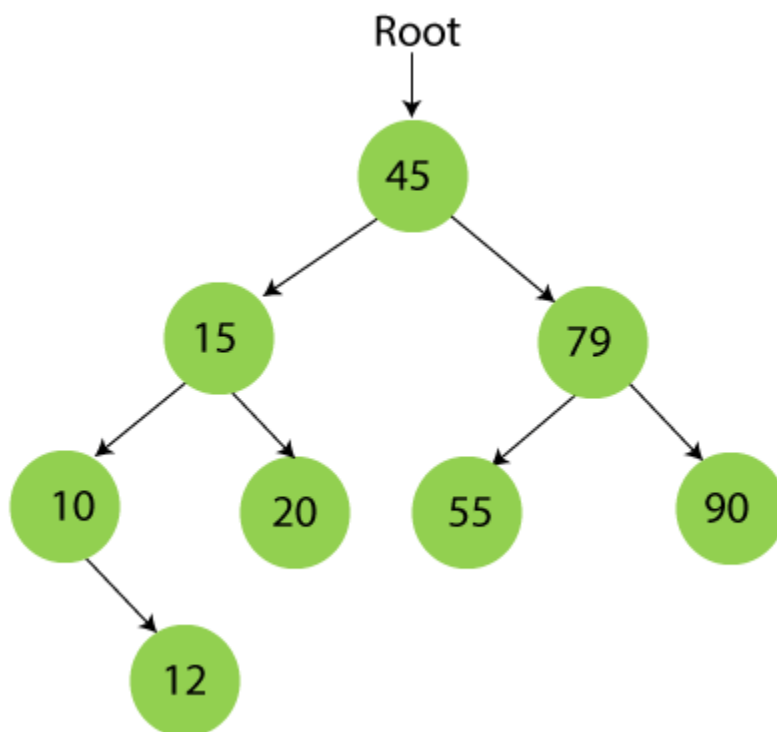
**Step 7 - Insert 12.**

12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right subtree of 10.



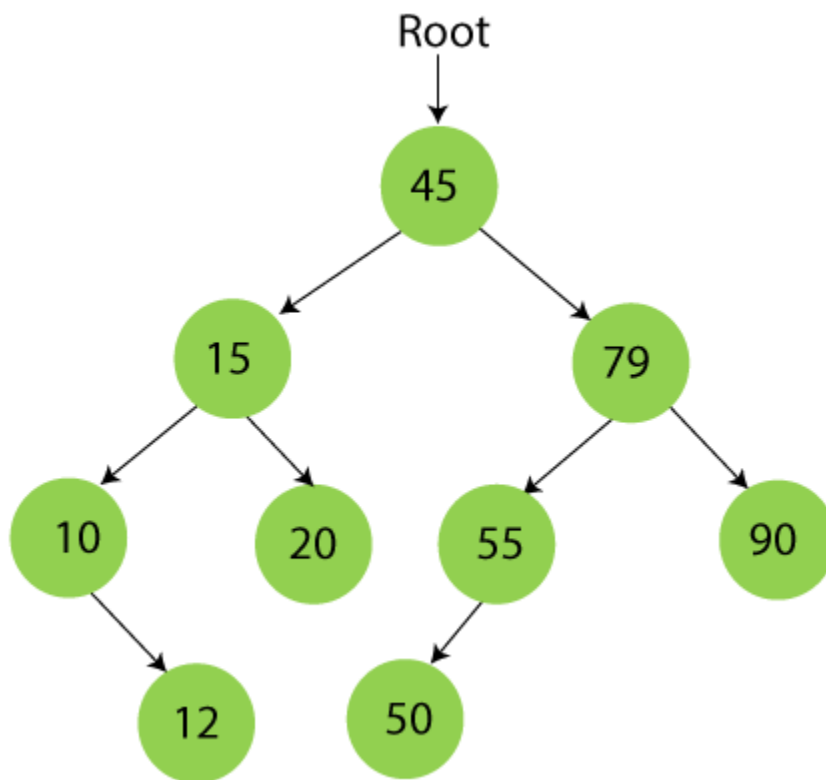
**Step 8 - Insert 20.**

20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree of 15.



### Step 9 - Insert 50.

50 is greater than 45 but smaller than 79 and 55. So, it will be inserted as a left subtree of 55.



Now, the creation of binary search tree is completed. After that, let's move towards the operations that can be performed on Binary search tree.

We can perform insert, delete and search operations on the binary search tree.

Let's understand how a search is performed on a binary search tree.

## Searching in Binary search tree

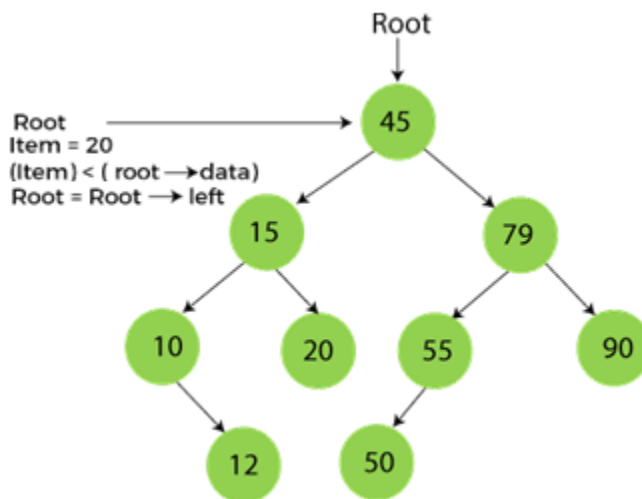
Searching means to find or locate a specific element or node in a data structure. In Binary search tree, searching a node is easy because elements in BST are stored in a specific order. The steps of searching a node in Binary Search tree are listed as follows -

1. First, compare the element to be searched with the root element of the tree.
2. If root is matched with the target element, then return the node's location.

3. If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the left subtree.
4. If it is larger than the root element, then move to the right subtree.
5. Repeat the above procedure recursively until the match is found.
6. If the element is not found or not present in the tree, then return NULL.

Now, let's understand the searching in binary tree using an example. We are taking the binary search tree formed above. Suppose we have to find node 20 from the below tree.

**Step1:**

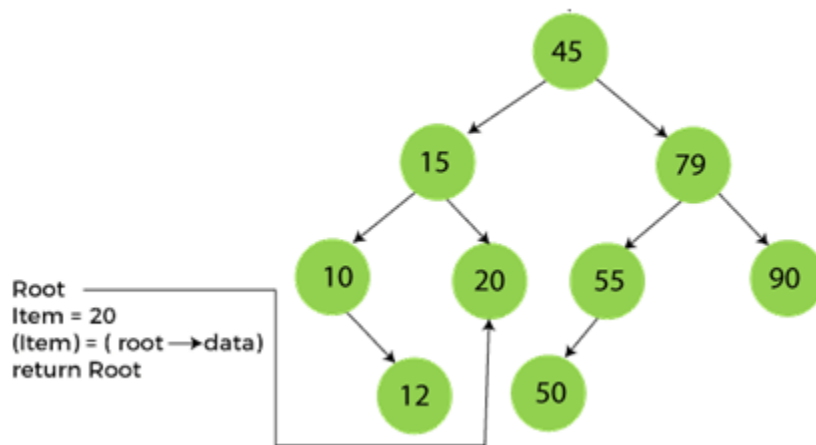


**Step2:**





### Step3:



Now, let's see the algorithm to search an element in the Binary search tree.

### Algorithm to search an element in Binary search tree

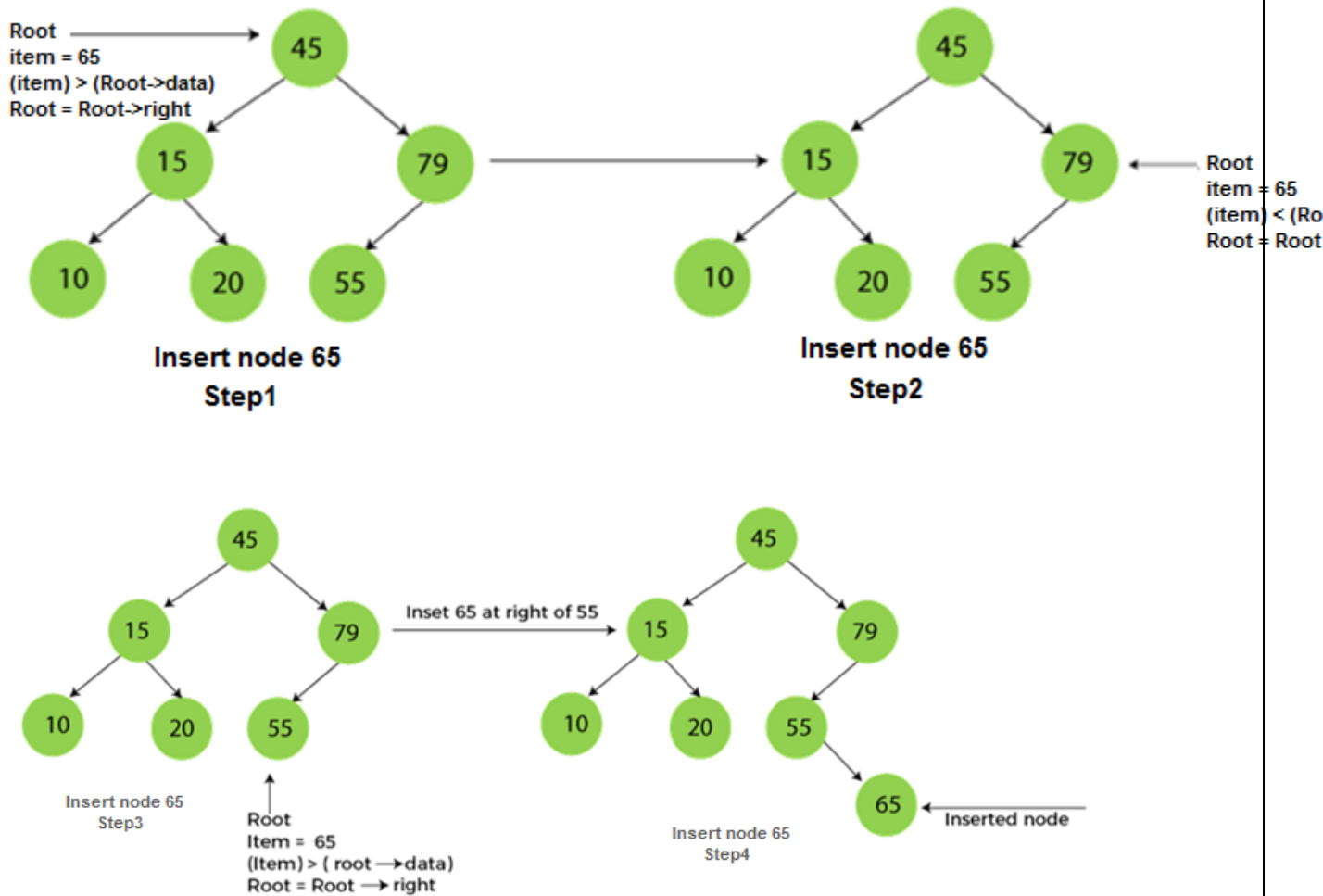
1. Search (root, item)
2. Step 1 - if (item = root → data) or (root = NULL)
3. return root
4. else if (item < root → data)
5. return Search(root → left, item)
6. else
7. return Search(root → right, item)
8. END if
9. Step 2 - END

Now let's understand how the deletion is performed on a binary search tree. We will also see an example to delete an element from the given tree.

### Insertion in Binary Search tree

A new key in BST is always inserted at the leaf. To insert an element in BST, we have to start searching from the root node; if the node to be inserted is less than the root node, then search for an empty location in the left subtree. Else, search for the empty location in the right subtree and insert the data. Insert in BST is similar to searching, as we always have to maintain the rule that the left subtree is smaller than the root, and right subtree is larger than the root.

Now, let's see the process of inserting a node into BST using an example.



## Deletion in Binary Search tree

In a binary search tree, we must delete a node from the tree by keeping in mind that the property of BST is not violated. To delete a node from BST, there are three possible situations occur -

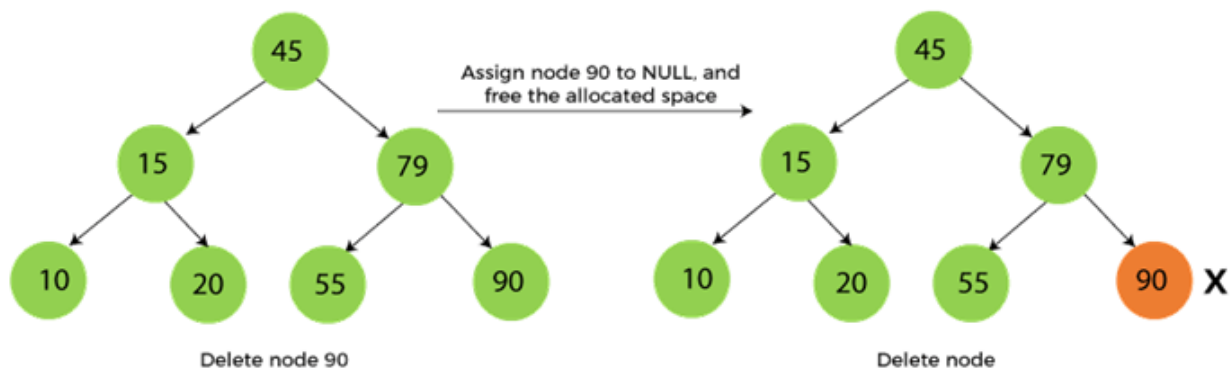
- The node to be deleted is the leaf node, or,
- The node to be deleted has only one child, and,
- The node to be deleted has two children

We will understand the situations listed above in detail.

### When the node to be deleted is the leaf node

It is the simplest case to delete a node in BST. Here, we have to replace the leaf node with NULL and simply free the allocated space.

We can see the process to delete a leaf node from BST in the below image. In below image, suppose we have to delete node 90, as the node to be deleted is a leaf node, so it will be replaced with NULL, and the allocated space will free.



## The complexity of the Binary Search tree

Let's see the time and space complexity of the Binary search tree. We will see the time complexity for insertion, deletion, and searching operations in best case, average case, and worst case.

### 1. Time Complexity

Operations	Best case time complexity	Average case time complexity	Worst case time complexity
Insertion	$O(\log n)$	$O(\log n)$	$O(n)$
Deletion	$O(\log n)$	$O(\log n)$	$O(n)$
Search	$O(\log n)$	$O(\log n)$	$O(n)$

Where 'n' is the number of nodes in the given tree.

## 2. Space Complexity

Operations	Space complexity
Insertion	$O(n)$
Deletion	$O(n)$
Search	$O(n)$

- The space complexity of all operations of Binary search tree is  $O(n)$ .

## AVL Tree

AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.

AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

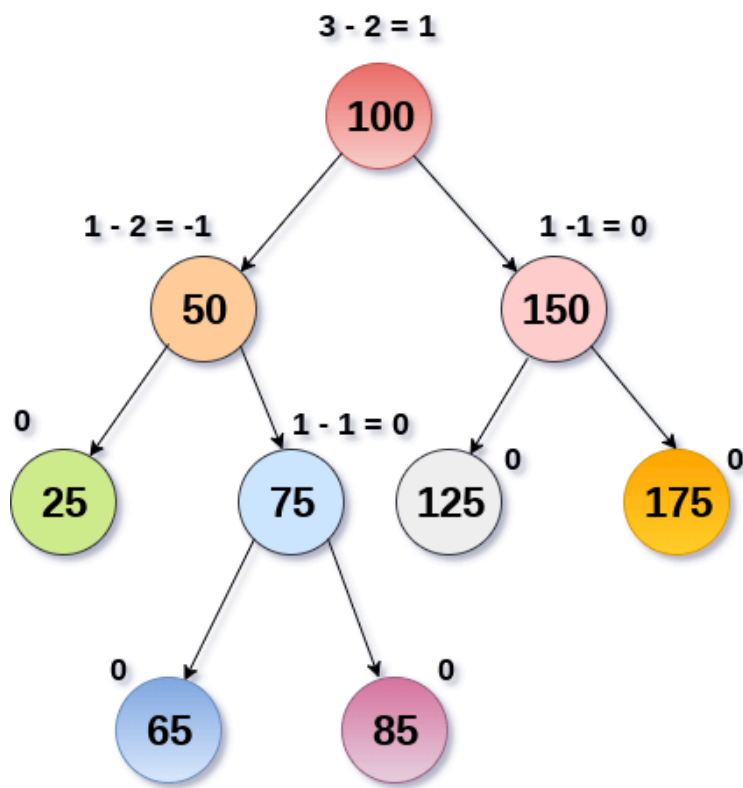
**Balance Factor (k) = height (left(k)) - height (right(k))**

If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.

If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.

If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

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.



**AVL Tree**

## 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 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.

SN	Operation	Description
1	<a href="#">Insertion</a>	Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. 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.
2	<a href="#">Deletion</a>	Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the 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.

## AVL Rotations

We perform rotation in AVL tree only in case if Balance Factor is other than **-1, 0, and 1**. There are basically four types of rotations which are as follows:

1. L L rotation: Inserted node is in the left subtree of left subtree of A
2. R R rotation : Inserted node is in the right subtree of right subtree of A
3. L R rotation : Inserted node is in the right subtree of left subtree of A

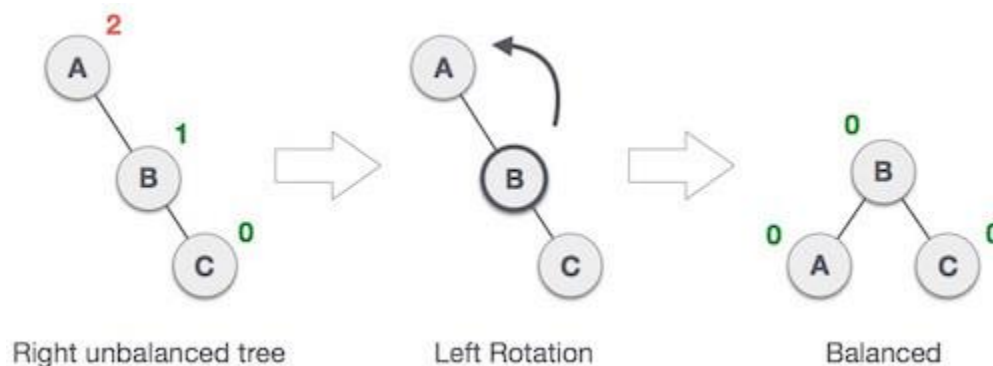
#### 4. R L rotation : Inserted node is in the left subtree of right subtree of A

Where node A is the node whose balance Factor is other than -1, 0, 1.

The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations. For a tree to be unbalanced, minimum height must be at least 2, Let us understand each rotation

### 1. RR Rotation

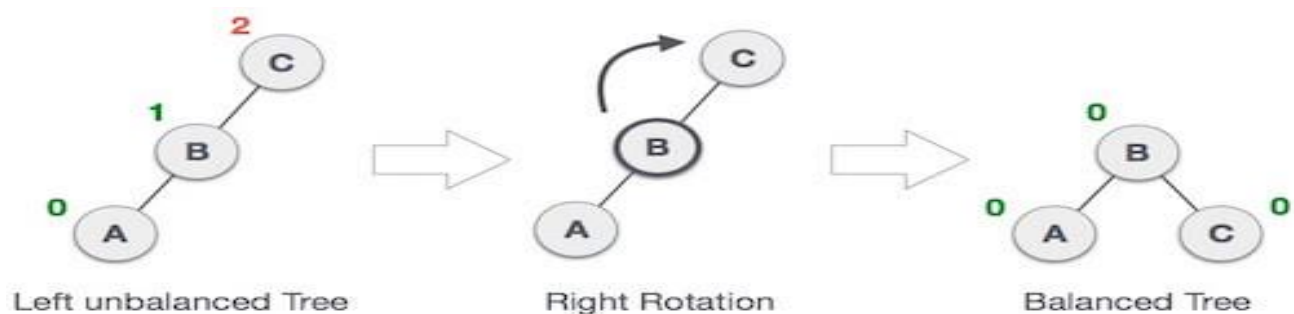
When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, [RR rotation](#) is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



In above example, node A has balance factor -2 because a node C is inserted in the right subtree of A right subtree. We perform the RR rotation on the edge below A.

### 2. LL Rotation

When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, [LL rotation](#) is clockwise rotation, which is applied on the edge below a node having balance factor 2.

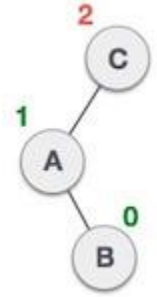
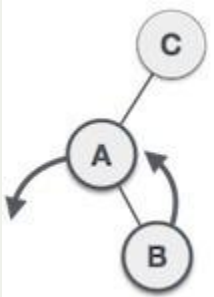
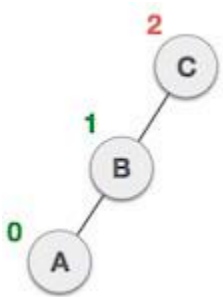


In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

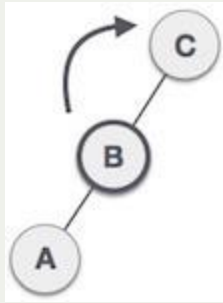
### 3. LR Rotation

Double rotations are bit tougher than single rotation which has already explained above. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

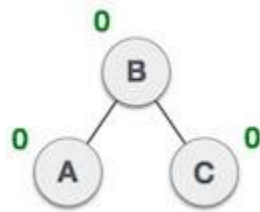
**Let us understand each and every step very clearly:**

State	Action
	<p>A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C</p>
	<p>As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node A, has become the left subtree of B.</p>
	<p>After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of C</p>





Now we perform LL clockwise rotation on full tree, i.e. on node C. node **C** has now become the right subtree of node B, A is left subtree of B



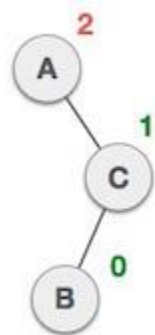
Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now.

## 4. RL Rotation

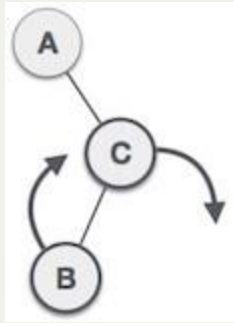
As already discussed, that double rotations are bit tougher than single rotation which has already explained above. R L rotation = LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

### State

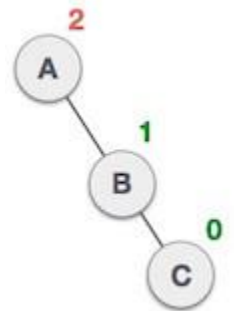
### Action



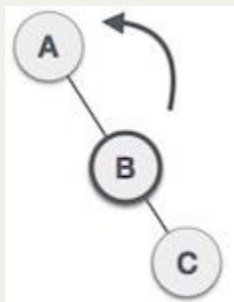
A node **B** has been inserted into the left subtree of **C** the right subtree of **A**, because of which A has become an unbalanced node having balance factor - 2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A



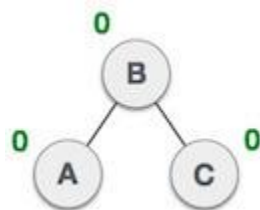
As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at **C** is performed first. By doing RR rotation, node **C** has become the right subtree of **B**.



After performing LL rotation, node **A** is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A.



Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node A. node **C** has now become the right subtree of node B, and node A has become the left subtree of B.

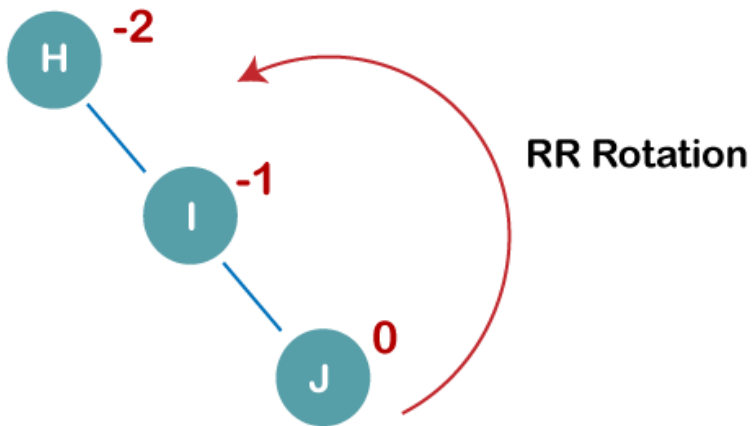


Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.

**Q: Construct an AVL tree having the following elements**

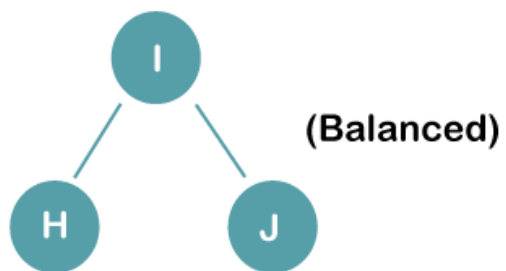
**H, I, J, B, A, E, C, F, D, G, K, L**

**1. Insert H, I, J**

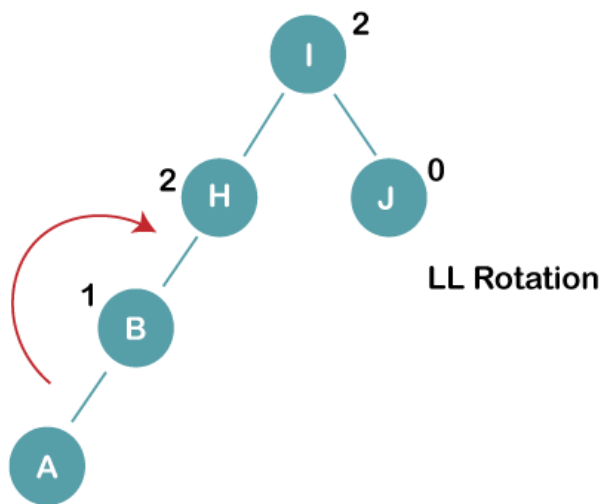


On inserting the above elements, especially in the case of H, the BST becomes unbalanced as the Balance Factor of H is -2. Since the BST is right-skewed, we will perform RR Rotation on node H.

**The resultant balance tree is:**

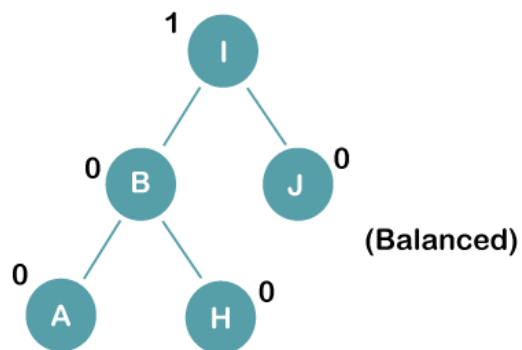


**2. Insert B, A**

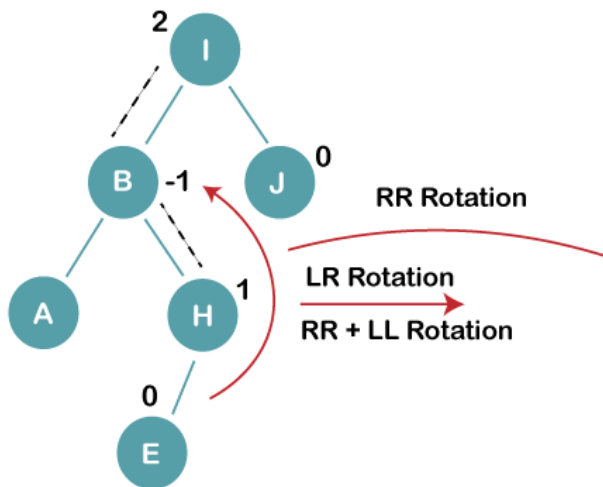


On inserting the above elements, especially in case of A, the BST becomes unbalanced as the Balance Factor of H and I is 2, we consider the first node from the last inserted node i.e. H. Since the BST from H is left-skewed, we will perform LL Rotation on node H.

**The resultant balance tree is:**



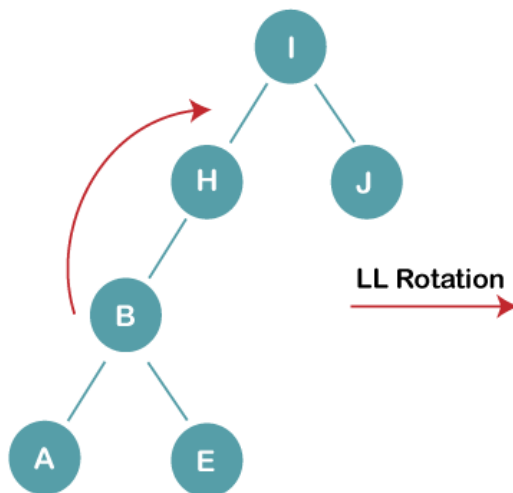
**3. Insert E**



On inserting E, BST becomes unbalanced as the Balance Factor of I is 2, since if we travel from E to I we find that it is inserted in the left subtree of right subtree of I, we will perform LR Rotation on node I. LR = RR + LL rotation

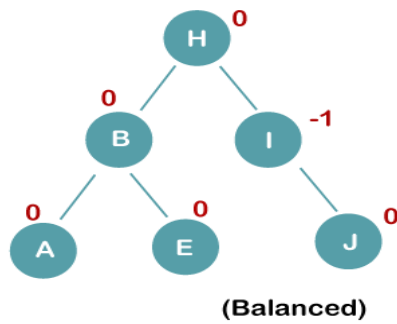
### 3 a) We first perform RR rotation on node B

The resultant tree after RR rotation is:

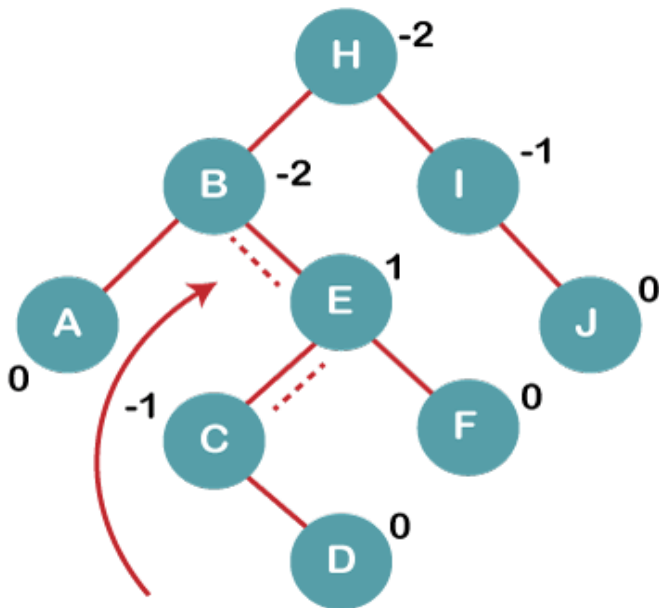


### 3b) We first perform LL rotation on the node I

The resultant balanced tree after LL rotation is:



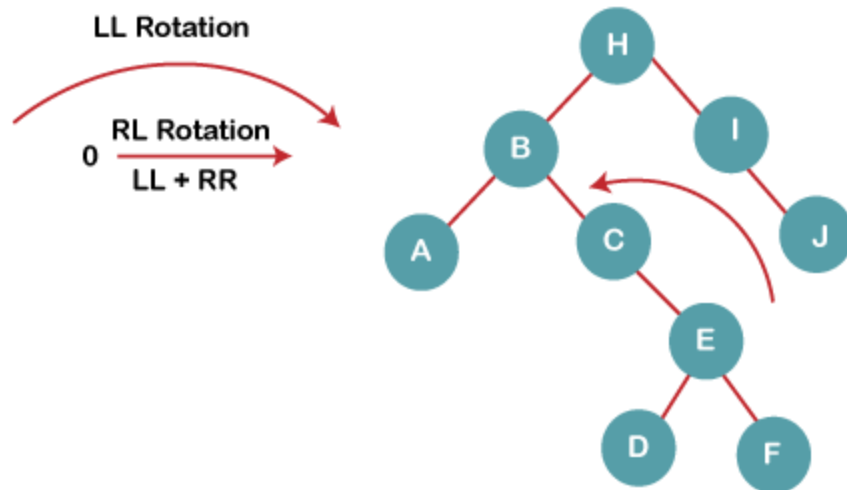
#### 4. Insert C, F, D



On inserting C, F, D, BST becomes unbalanced as the Balance Factor of B and H is -2, since if we travel from D to B we find that it is inserted in the right subtree of left subtree of B, we will perform RL Rotation on node I. RL = LL + RR rotation.

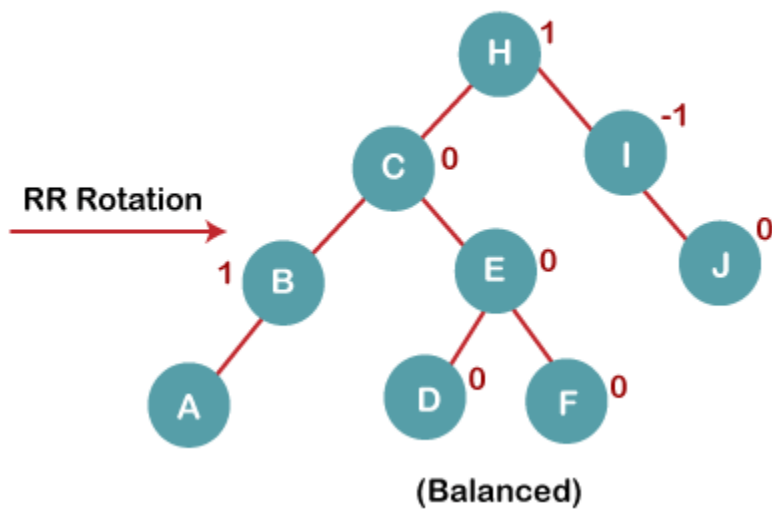
#### 4a) We first perform LL rotation on node E

The resultant tree after LL rotation is:

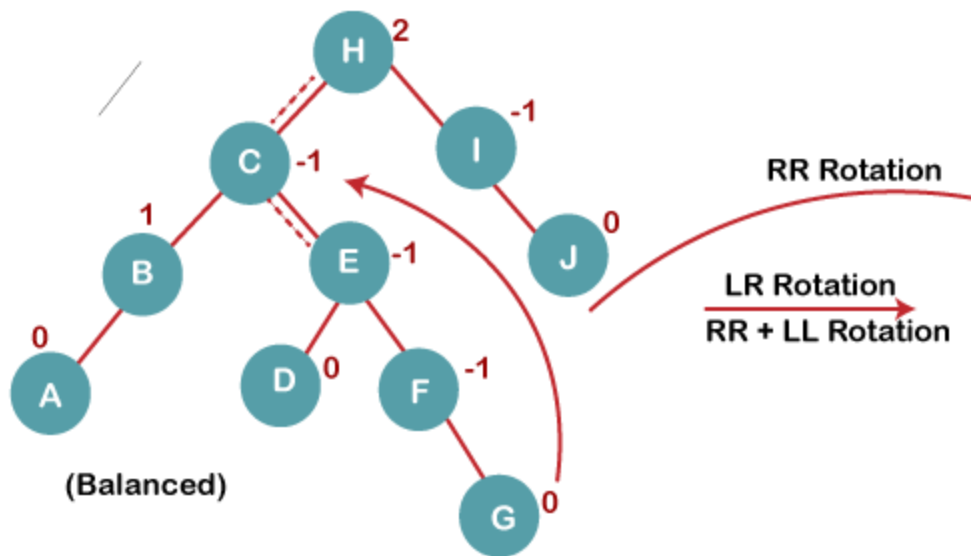


4b) We then perform RR rotation on node B

The resultant balanced tree after RR rotation is:



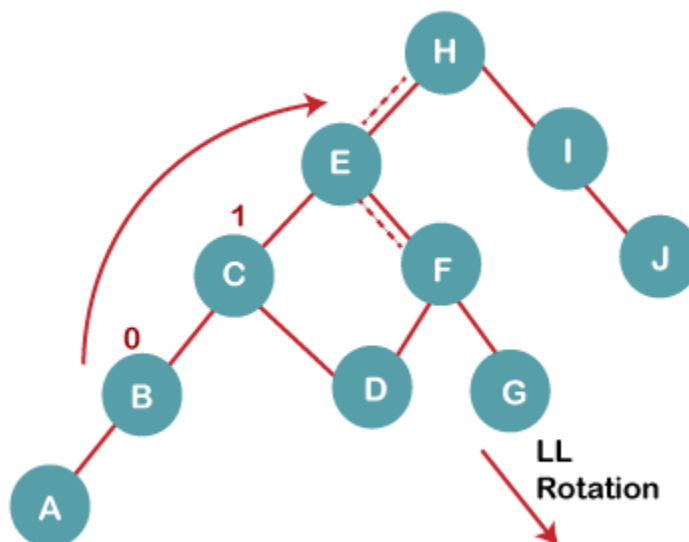
5. Insert G



On inserting G, BST become unbalanced as the Balance Factor of H is 2, since if we travel from G to H, we find that it is inserted in the left subtree of right subtree of H, we will perform LR Rotation on node I. LR = RR + LL rotation.

#### 5 a) We first perform RR rotation on node C

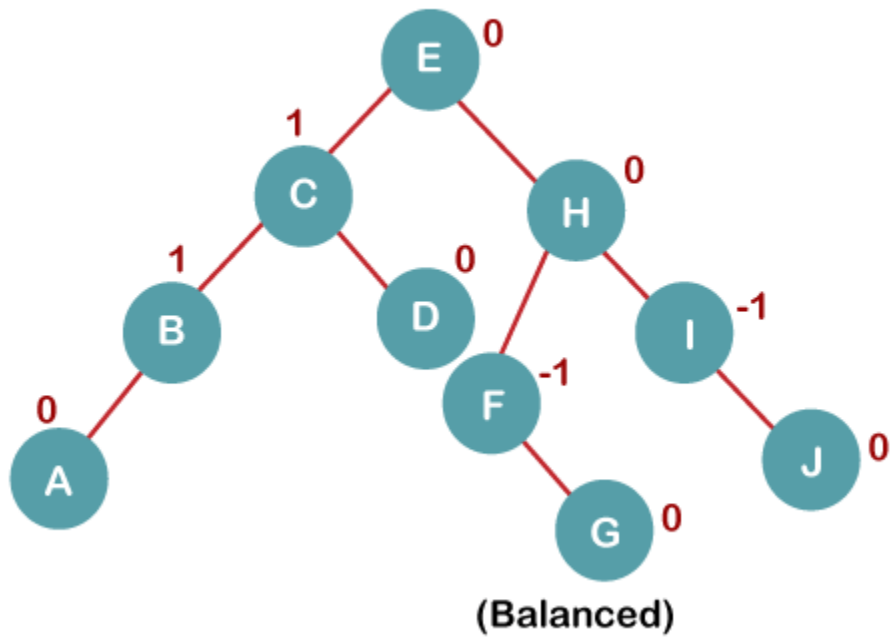
The resultant tree after RR rotation is:



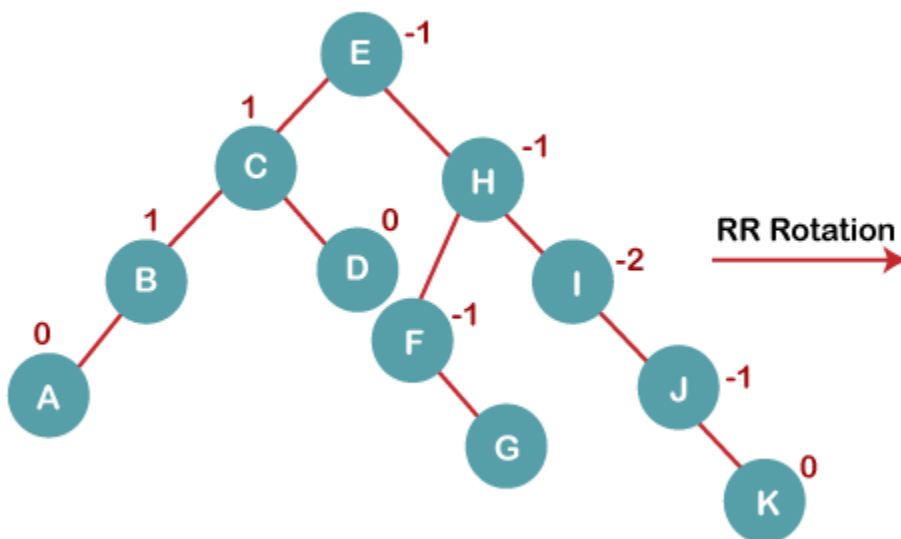


5 b) We then perform LL rotation on node H

The resultant balanced tree after LL rotation is:

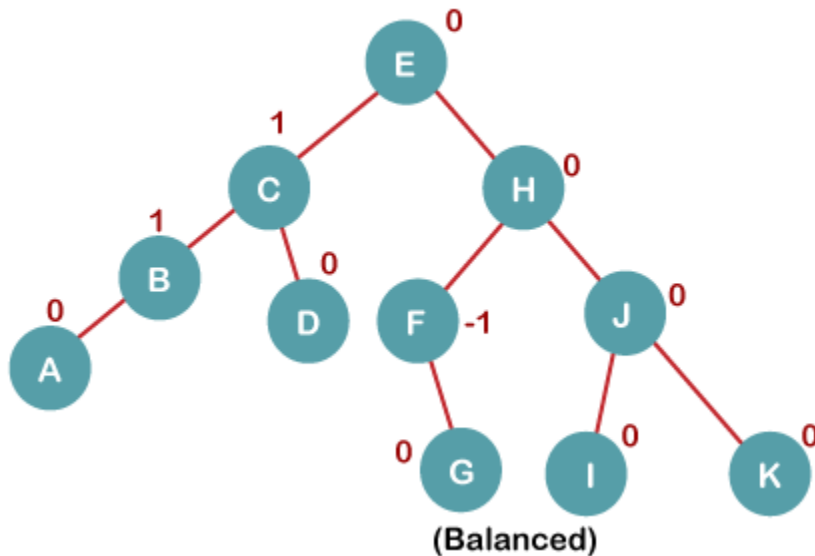


6. Insert K



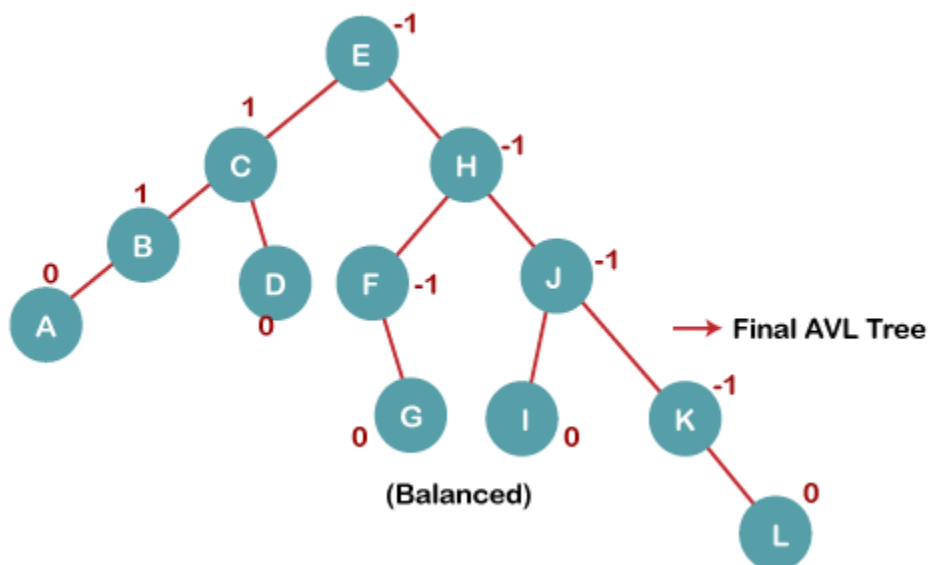
On inserting K, BST becomes unbalanced as the Balance Factor of I is -2. Since the BST is right-skewed from I to K, hence we will perform RR Rotation on the node I.

**The resultant balanced tree after RR rotation is:**



## 7. Insert L

On inserting the L tree is still balanced as the Balance Factor of each node is now either, -1, 0, +1. Hence the tree is a Balanced 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

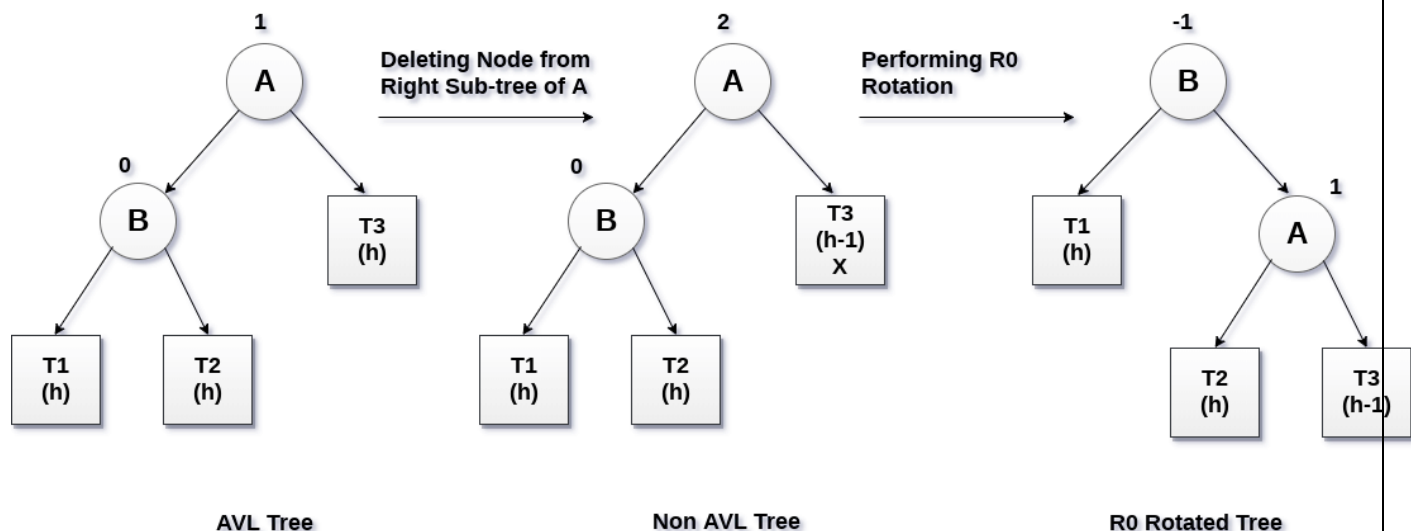
left sub-tree of the critical node, then L rotation needs to be applied else if, the node which is to be deleted is present in the right sub-tree of the critical node, the R rotation will be applied.

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 )

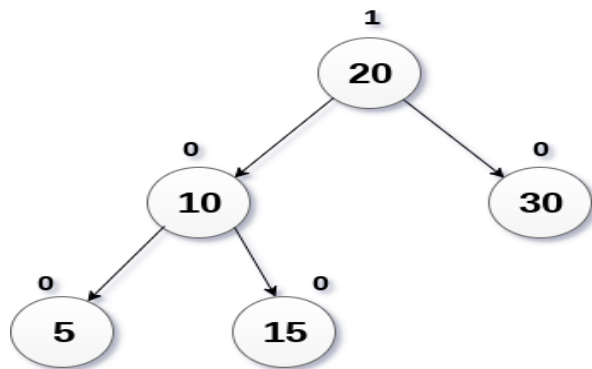
If the node B has 0 balance factor, and the balance factor of node A disturbed upon deleting the node X, then the tree will be rebalanced by rotating tree using R0 rotation.

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.



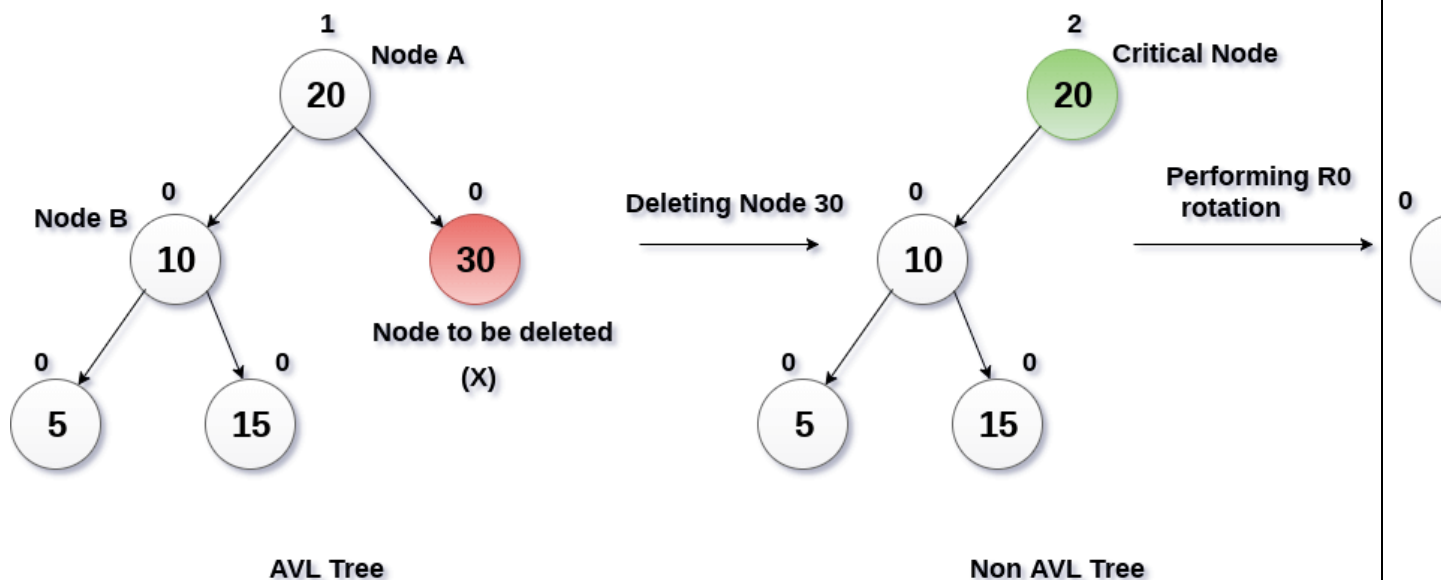
## Example:

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



## Solution

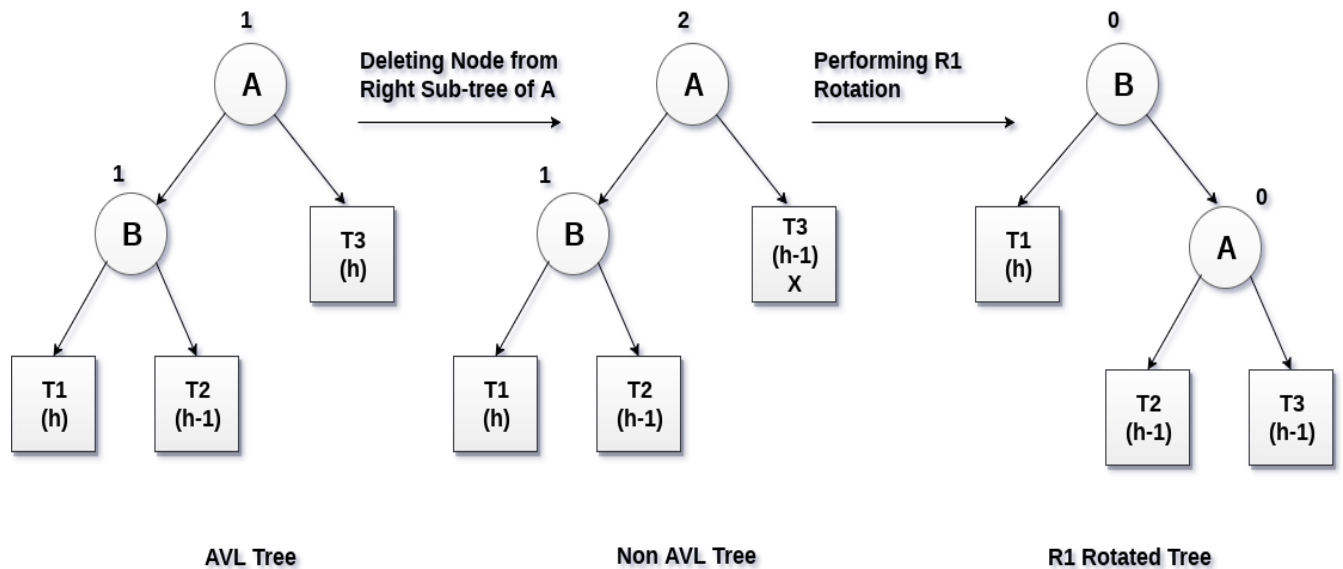
In this case, the node B has balance factor 0, therefore the tree will be rotated by using R0 rotation as shown in the following image. The node B(10) becomes the root, while the node A is moved to its right. The right child of node B will now become the left child of node A.



## R1 Rotation (Node B has balance factor 1)

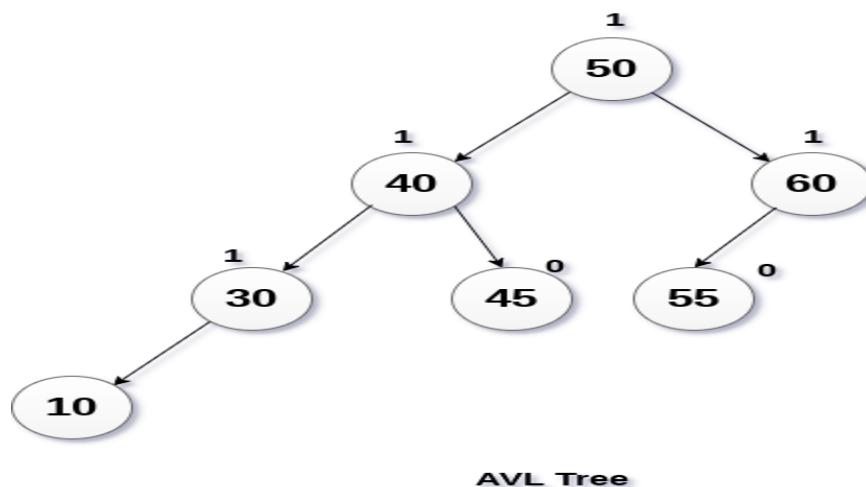
R1 Rotation is to be performed if the balance factor of Node B is 1. In R1 rotation, the critical node A is moved to its right having sub-trees T2 and T3 as its left and right child respectively. T1 is to be placed as the left sub-tree of the node B.

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



## Example

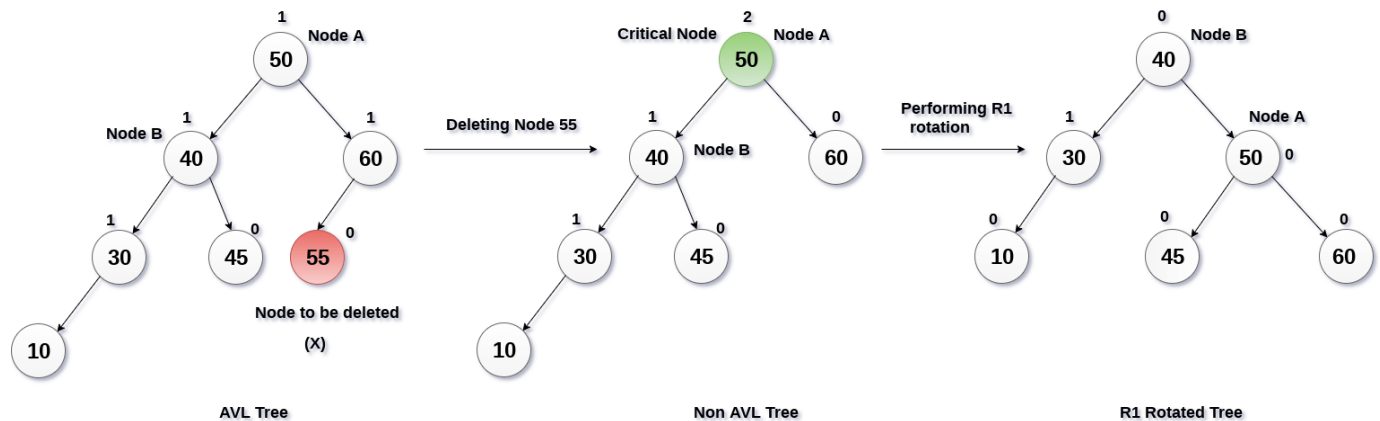
Delete Node 55 from the AVL tree shown in the following image.



## 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.

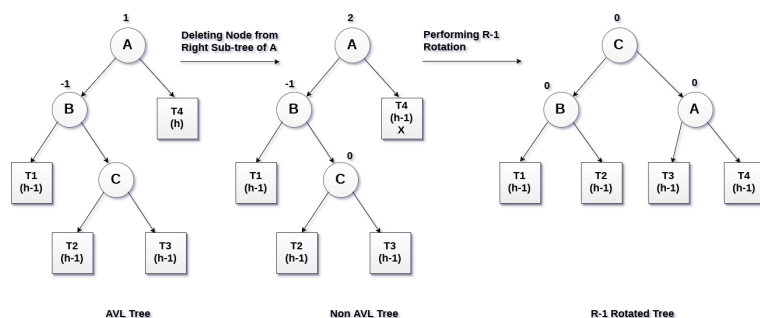


## R-1 Rotation (Node B has balance factor -1)

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.

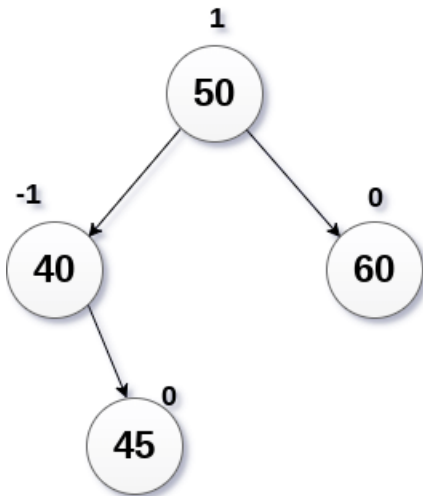
The sub-trees T1, T2 becomes the left and right sub-trees of B whereas, T3, T4 become the left and right sub-trees of A.

The process involved in R-1 rotation is shown in the following image.



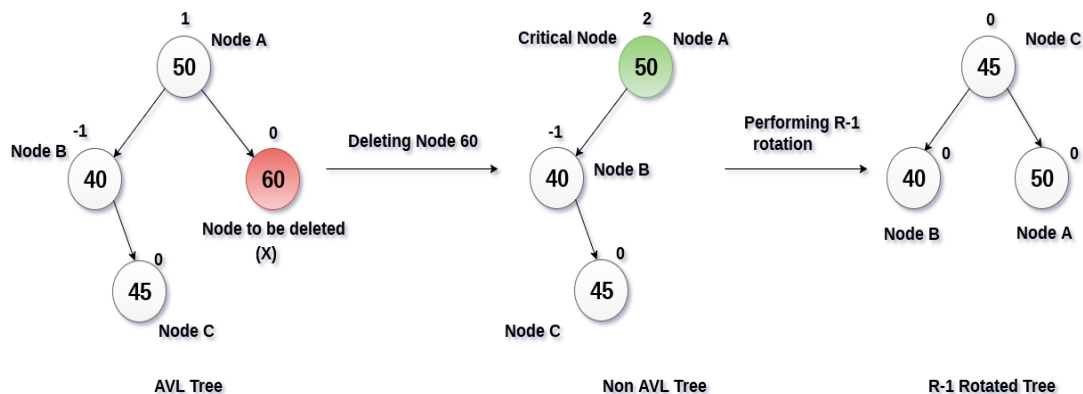
## Example

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.



## Search Operation:

The search operation in an AVL tree with parent pointers is similar to the [search operation in a normal Binary Search Tree](#). Follow the steps below to perform search operation:

- Start from the root node.
- If the root node is **NULL**, return **false**.
- Check if the current node's value is equal to the value of the node to be searched. If yes, return **true**.
- If the current node's value is less than searched key then recur to the right subtree.
- If the current node's value is greater than searched key then recur to the left subtree.

## Red Black Tree

A Red Black Tree is a category of the self-balancing binary search tree. It was created in 1972 by Rudolf Bayer who termed them "**symmetric binary B-trees**."

A red-black tree is a Binary tree where a particular node has color as an extra attribute, either red or black. By check the node colors on any simple path from the root to a leaf, red-black trees secure that no such path is higher than twice as long as any other so that the tree is generally balanced.

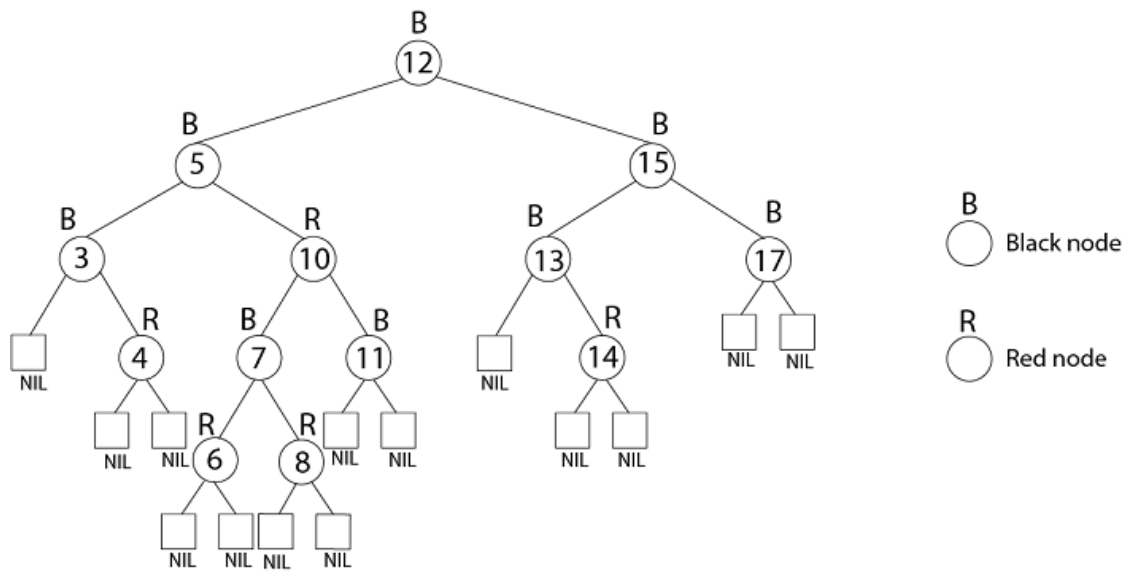
## Properties of Red-Black Trees

A red-black tree must satisfy these properties:

1. The root is always black.
2. A nil is recognized to be black. This factor that every non-NIL node has two children.
3. **Black Children Rule:** The children of any red node are black.
4. **Black Height Rule:** For particular node  $v$ , there exists an integer  $bh(v)$  such that specific downward path from  $v$  to a nil has correctly  $bh(v)$  black real (i.e. non-nil) nodes. Call this portion the black height of  $v$ . We determine the black height of an RB tree to be the black height of its root.

A tree  $T$  is an almost red-black tree (ARB tree) if the root is red, but other conditions above hold.



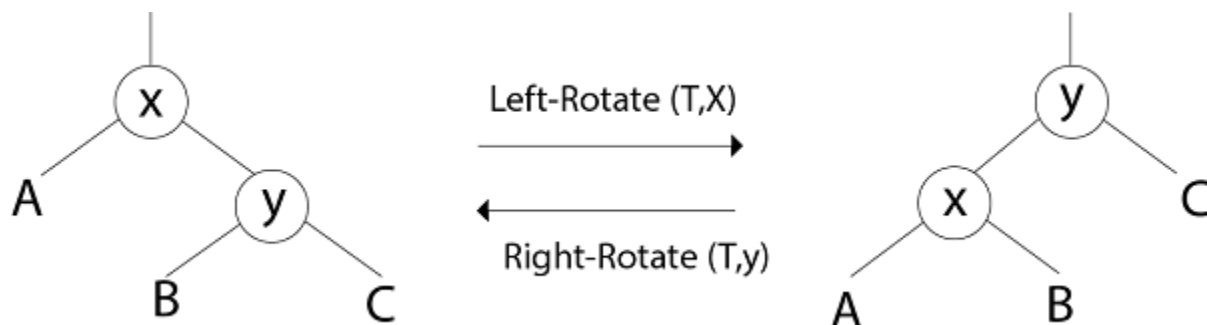


## Operations on red-black Trees:

The search-tree operations TREE-INSERT and TREE-DELETE, when runs on a red-black tree with  $n$  keys, take  $O(\log n)$  time. Because they customize the tree, the conclusion may violate the red-black properties. To restore these properties, we must change the color of some of the nodes in the tree and also change the pointer structure.

### 1. Rotation:

Restructuring operations on red-black trees can generally be expressed more clearly in details of the rotation operation.



### 2. Insertion:

- Insert the new node the way it is done in Binary Search Trees.
- Color the node red

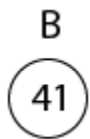
- If an inconsistency arises for the red-black tree, fix the tree according to the type of discrepancy.

A discrepancy can decision from a parent and a child both having a red color. This type of discrepancy is determined by the location of the node concerning grandparent, and the color of the sibling of the parent.

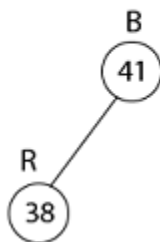
**Example:** Show the red-black trees that result after successively inserting the keys 41,38,31,12,19,8 into an initially empty red-black tree.

**Solution:**

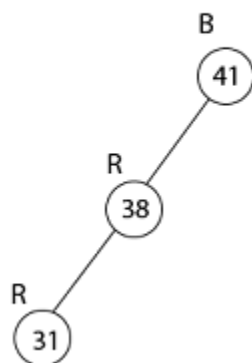
Insert 41



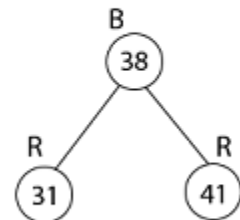
◀ Insert 38



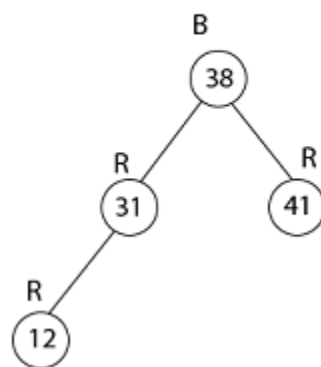
◀ Insert 31



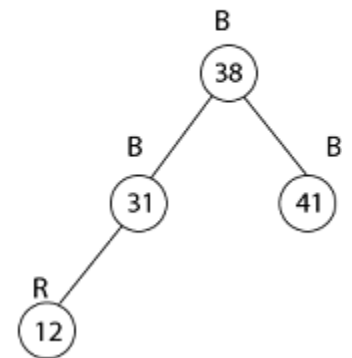
Case 3 →



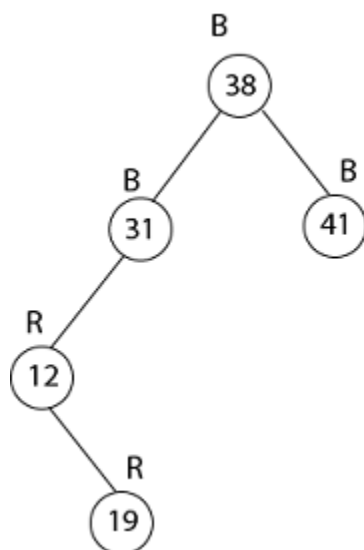
◀ Insert 12



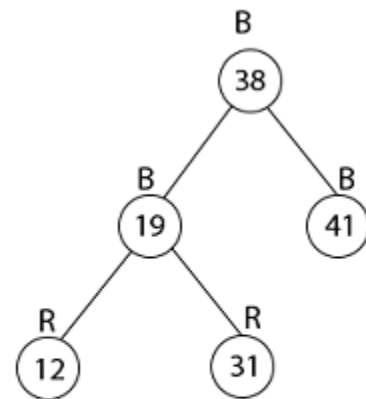
Case 1 →



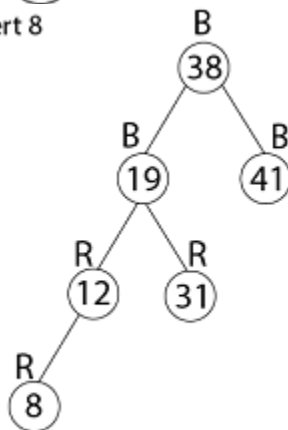
Insert 19



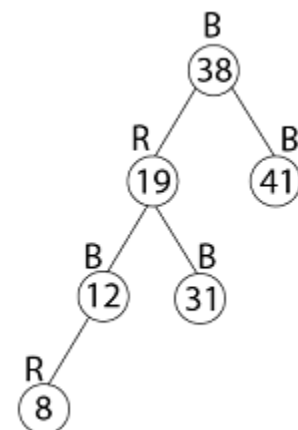
Case 2,3 →



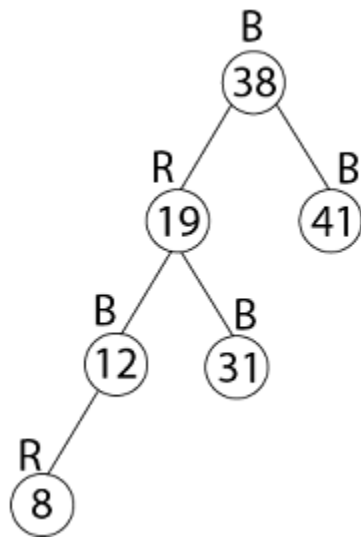
◀ Insert 8



Case-1 →



Thus the final tree is



### 3. Deletion:

First, search for an element to be deleted

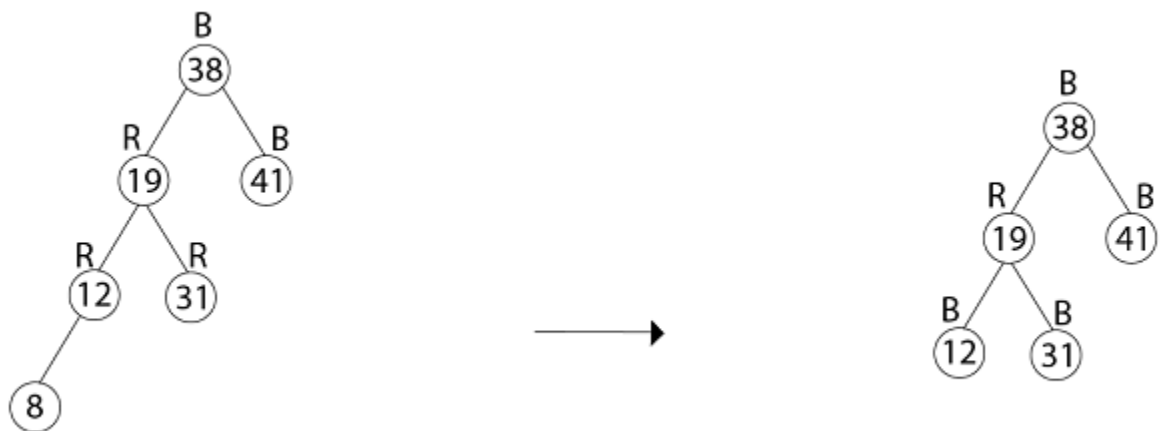
- If the element to be deleted is in a node with only left child, swap this node with one containing the largest element in the left subtree. (This node has no right child).
- If the element to be deleted is in a node with only right child, swap this node with the one containing the smallest element in the right subtree (This node has no left child).
- If the element to be deleted is in a node with both a left child and a right child, then swap in any of the above two ways. While swapping, swap only the keys but not the colors.
- The item to be deleted is now having only a left child or only a right child. Replace this node with its sole child. This may violate red constraints or black constraint. Violation of red constraints can be easily fixed.
- If the deleted node is black, the black constraint is violated. The elimination of a black node y causes any path that contained y to have one fewer black node.
- Two cases arise:
  - The replacing node is red, in which case we merely color it black to make up for the loss of one black node.
  - The replacing node is black.

The strategy RB-DELETE is a minor change of the TREE-DELETE procedure. After splicing out a node, it calls an auxiliary procedure RB-DELETE-FIXUP that changes colors and performs rotation to restore the red-black properties.

**Example:** In a previous example, we found that the red-black tree that results from successively inserting the keys 41,38,31,12,19,8 into an initially empty tree. Now show the red-black trees that result from the successful deletion of the keys in the order 8, 12, 19,31,38,41.

**Solution:**

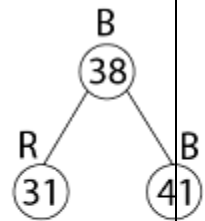
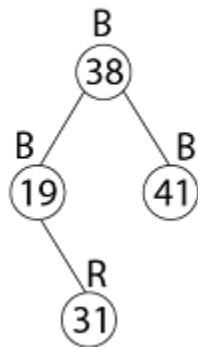
◀ Delete 8



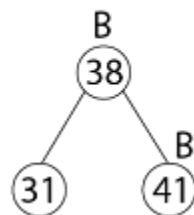
◀ Delete 12



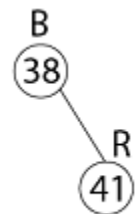
◀ Delete 19



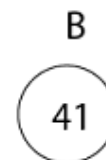
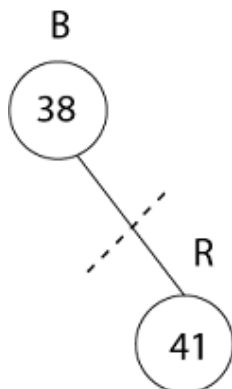
◀ Delete 31



Case-2 →



**Delete 38**



**Delete 41**

No tree

## Splay Tree

Splay trees are the self-balancing or self-adjusted binary search trees. In other words, we can say that the splay trees are the variants of the binary search trees. The prerequisite for the splay trees that we should know about the binary search trees.

As we already know, the time complexity of a binary search tree in every case. The time complexity of a binary search tree in the average case is  **$O(\log n)$**  and the time complexity in the worst case is  $O(n)$ . In a binary search tree, the value of the left subtree is smaller than the root node, and the value of the right subtree is greater than the root node; in such case, the time complexity would be  **$O(\log n)$** . If the binary tree is left-skewed or right-skewed, then the time complexity would be  $O(n)$ . To limit the skewness, the [AVL and Red-Black tree](#) came into the picture, having  **$O(\log n)$**  time complexity for all the operations in all the cases. We can also improve this time complexity by doing more practical implementations, so the new Tree [data structure](#) was designed, known as a Splay tree.

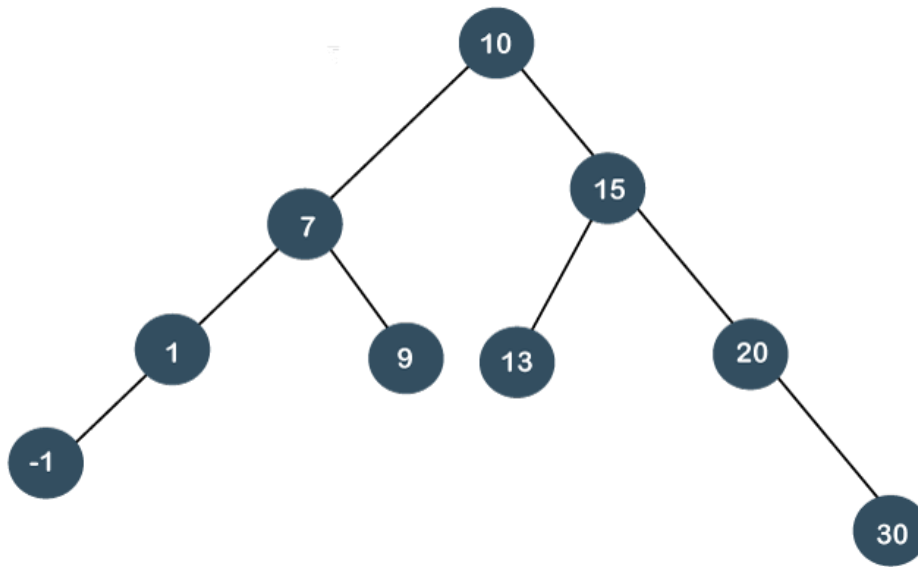
## What is a Splay Tree?

A splay tree is a self-balancing tree, but [AVL](#) and [Red-Black trees](#) are also self-balancing trees then. What makes the splay tree unique two trees. It has one extra property that makes it unique is splaying.

A splay tree contains the same operations as a [Binary search tree](#), i.e., Insertion, deletion and searching, but it also contains one more operation, i.e., splaying. So. all the operations in the splay tree are followed by splaying.

Splay trees are not strictly balanced trees, but they are roughly balanced trees. Let's understand the search operation in the splay-tree.

Suppose we want to search 7 element in the tree, which is shown below:



To search any element in the splay tree, first, we will perform the standard binary search tree operation. As 7 is less than 10 so we will come to the left of the root node. After performing the search operation, we need to perform splaying. Here splaying means that the operation that we are performing on any element should become the root node after performing some rearrangements. The rearrangement of the tree will be done through the rotations.

*Note: The splay tree can be defined as the self-adjusted tree in which any operation performed on the element would rearrange the tree so that the element on which operation has been performed becomes the root node of the tree.*

## Rotations

**There are six types of rotations used for splaying:**

1. Zig rotation (Right rotation)
2. Zag rotation (Left rotation)
3. Zig zag (Zig followed by zag)
4. Zag zig (Zag followed by zig)
5. Zig zig (two right rotations)
6. Zag zag (two left rotations)

**Factors required for selecting a type of rotation**



**The following are the factors used for selecting a type of rotation:**

- Does the node which we are trying to rotate have a grandparent?
- Is the node left or right child of the parent?
- Is the node left or right child of the grandparent?

## Cases for the Rotations

**Case 1:** If the node does not have a grand-parent, and if it is the right child of the parent, then we carry out the left rotation; otherwise, the right rotation is performed.

**Case 2:** If the node has a grandparent, then based on the following scenarios; the rotation would be performed:

**Scenario 1:** If the node is the right of the parent and the parent is also right of its parent, then **zig zig right right rotation** is performed.

**Scenario 2:** If the node is left of a parent, but the parent is right of its parent, then **zig zag right left rotation** is performed.

**Scenario 3:** If the node is right of the parent and the parent is right of its parent, then **zig zig left left rotation** is performed.

**Scenario 4:** If the node is right of a parent, but the parent is left of its parent, then **zig zag right-left rotation** is performed.

**Now, let's understand the above rotations with examples.**

To rearrange the tree, we need to perform some rotations. The following are the types of rotations in the splay tree:

- **Zig rotations**

The zig rotations are used when the item to be searched is either a root node or the child of a root node (i.e., left or the right child).

**The following are the cases that can exist in the splay tree while searching:**

**Case 1:** If the search item is a root node of the tree.

**Case 2:** If the search item is a child of the root node, then the two scenarios will be there:

1. If the child is a left child, the right rotation would be performed, known as a zig right rotation.
2. If the child is a right child, the left rotation would be performed, known as a zig left rotation.

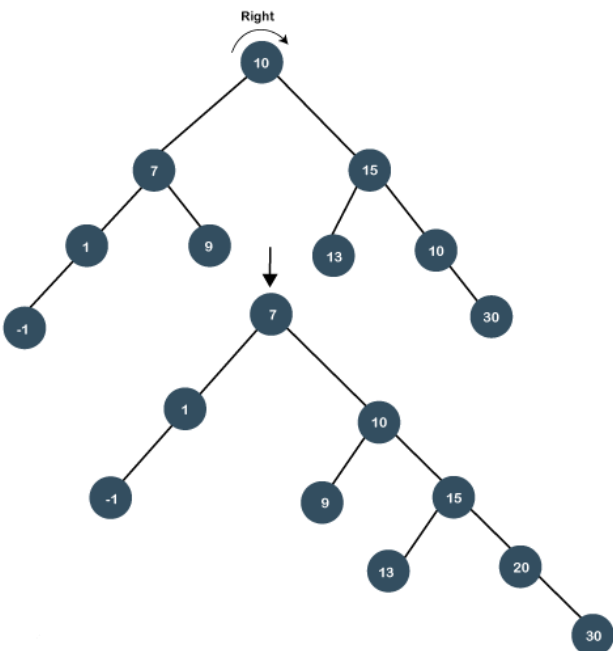
**Let's look at the above two scenarios through an example.**

**Consider the below example:**

In the above example, we have to search 7 element in the tree. We will follow the below steps:

**Step 1:** First, we compare 7 with a root node. As 7 is less than 10, so it is a left child of the root node.

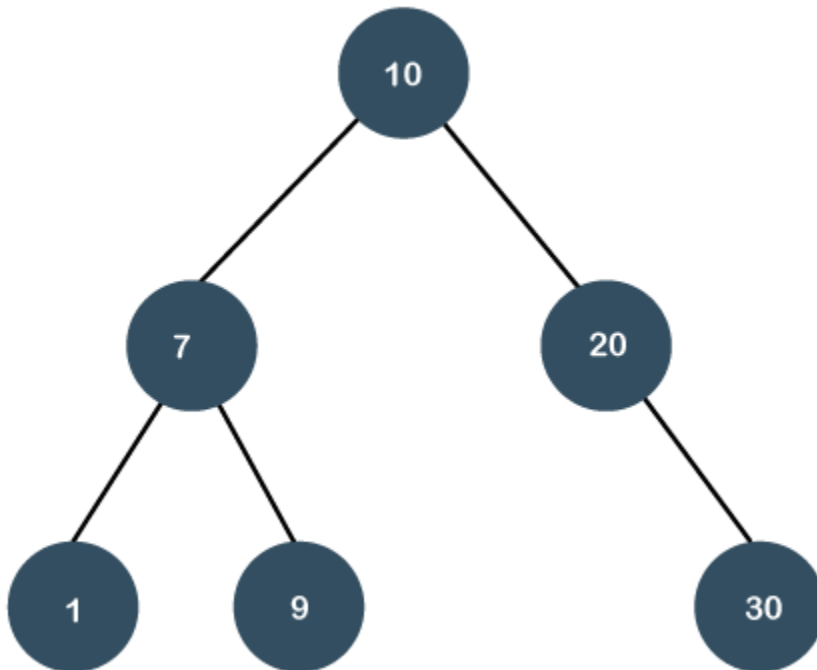
**Step 2:** Once the element is found, we will perform splaying. The right rotation is performed so that 7 becomes the root node of the tree, as shown below:



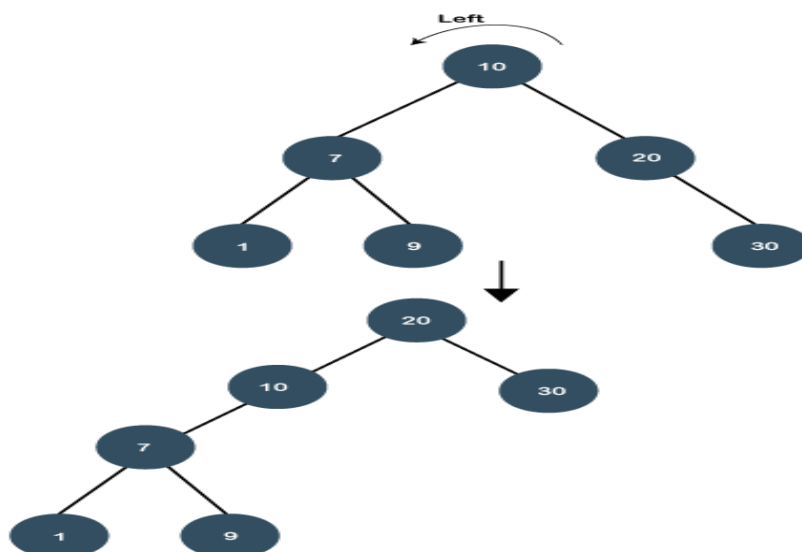
**Let's consider another example.**

In the above example, we have to search 20 element in the tree. We will follow the below steps:

**Step 1:** First, we compare 20 with a root node. As 20 is greater than the root node, so it is a right child of the root node.



**Step 2:** Once the element is found, we will perform splaying. The left rotation is performed so that 20 element becomes the root node of the tree.



- **Zig zig rotations**

Sometimes the situation arises when the item to be searched is having a parent as well as a grandparent. In this case, we have to perform four rotations for splaying.

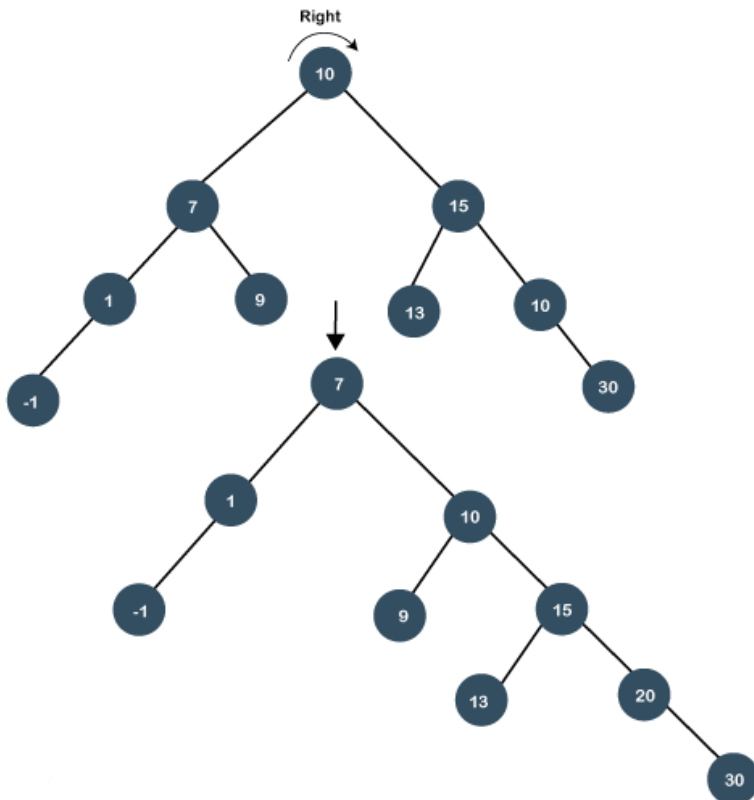
Let's understand this case through an example.

Suppose we have to search 1 element in the tree, which is shown below:

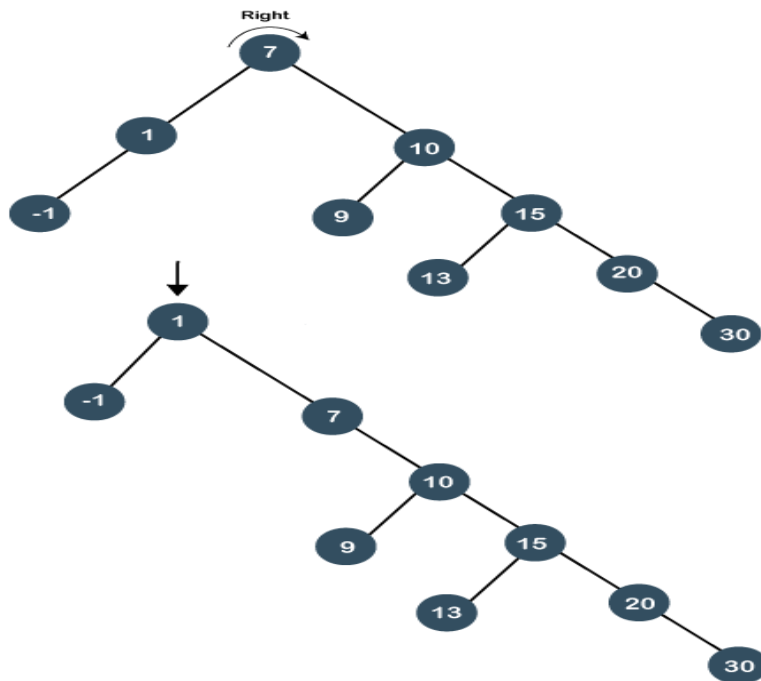
**Step 1:** First, we have to perform a standard BST searching operation in order to search the 1 element. As 1 is less than 10 and 7, so it will be at the left of the node 7. Therefore, element 1 is having a parent, i.e., 7 as well as a grandparent, i.e., 10.

**Step 2:** In this step, we have to perform splaying. We need to make node 1 as a root node with the help of some rotations. In this case, we cannot simply perform a zig or zag rotation; we have to implement zig zig rotation.

In order to make node 1 as a root node, we need to perform two right rotations known as zig zig rotations. When we perform the right rotation then 10 will move downwards, and node 7 will come upwards as shown in the below figure:



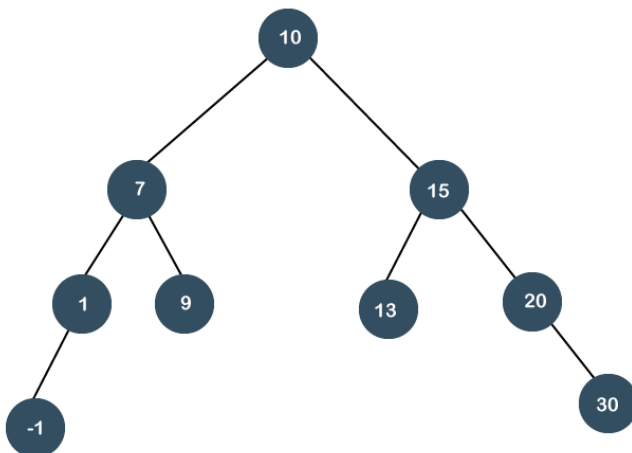
Again, we will perform zig right rotation, node 7 will move downwards, and node 1 will come upwards as shown below:



As we observe in the above figure that node 1 has become the root node of the tree; therefore, the searching is completed.

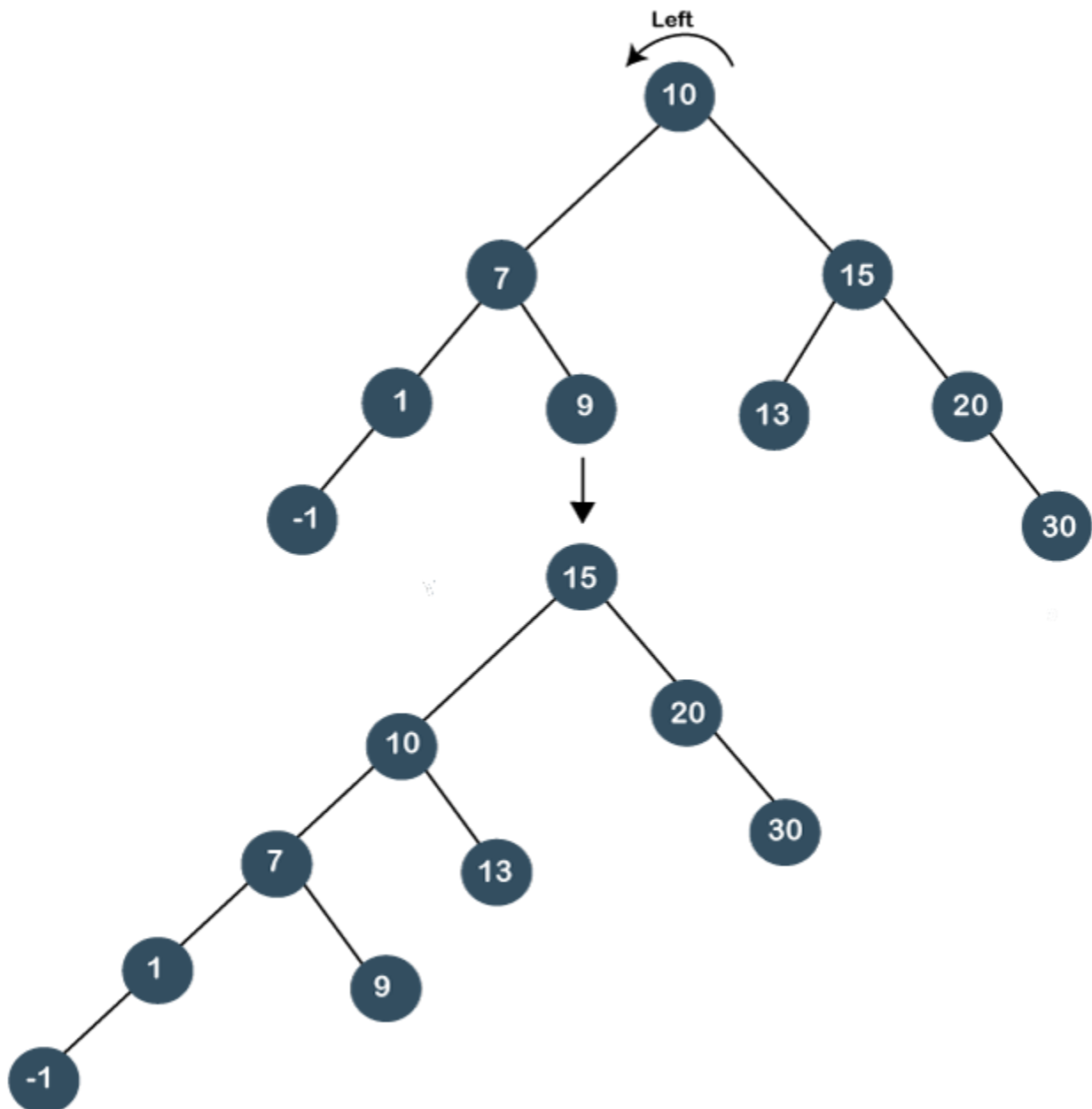
**Suppose we want to search 20 in the below tree.**

In order to search 20, we need to perform two left rotations. Following are the steps required to search 20 node:

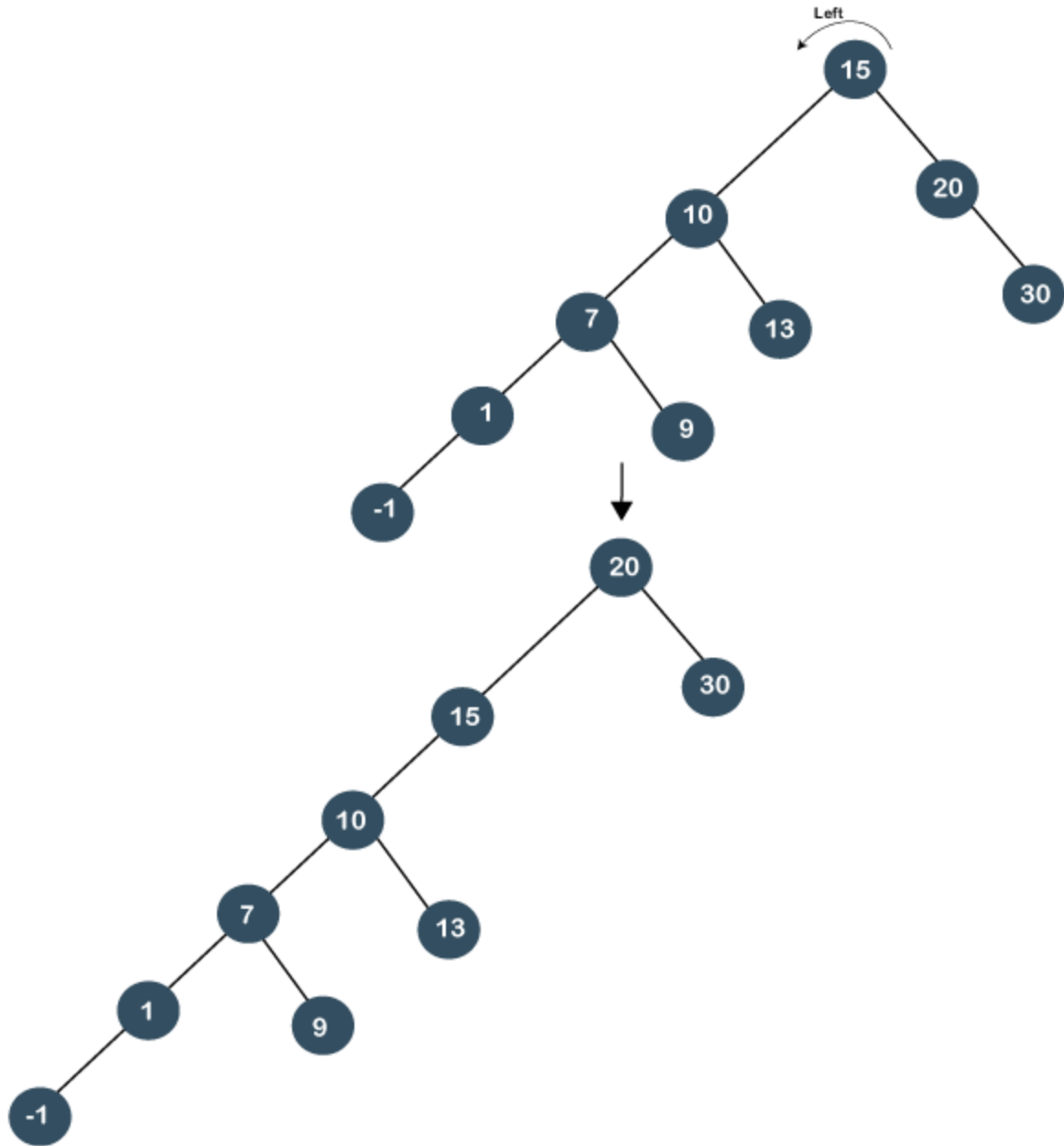


**Step 1:** First, we perform the standard BST searching operation. As 20 is greater than 10 and 15, so it will be at the right of node 15.

**Step 2:** The second step is to perform splaying. In this case, two left rotations would be performed. In the first rotation, node 10 will move downwards, and node 15 would move upwards as shown below:



In the second left rotation, node 15 will move downwards, and node 20 becomes the root node of the tree, as shown below:



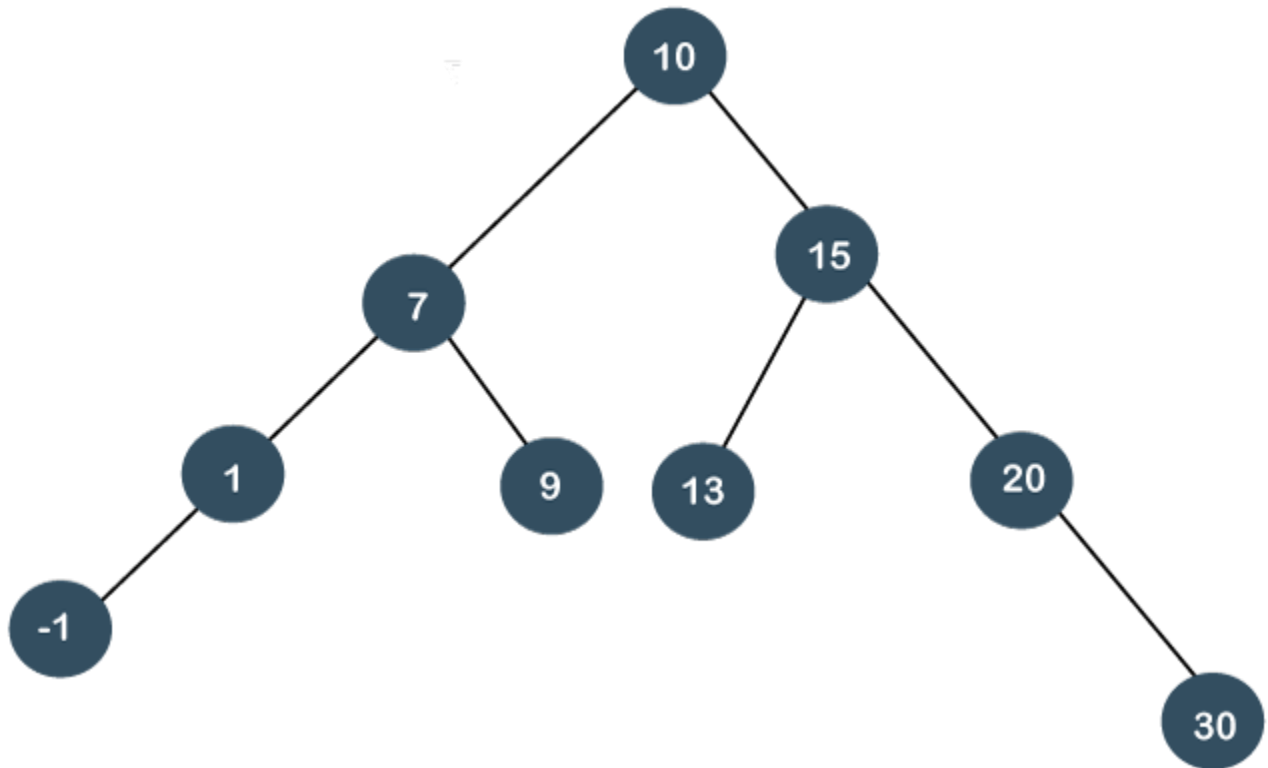
As we have observed that two left rotations are performed; so it is known as a zig zig left rotation.

- **Zig zag rotations**

Till now, we have read that both parent and grandparent are either in RR or LL relationship. Now, we will see the RL or LR relationship between the parent and the grandparent.

**Let's understand this case through an example.**

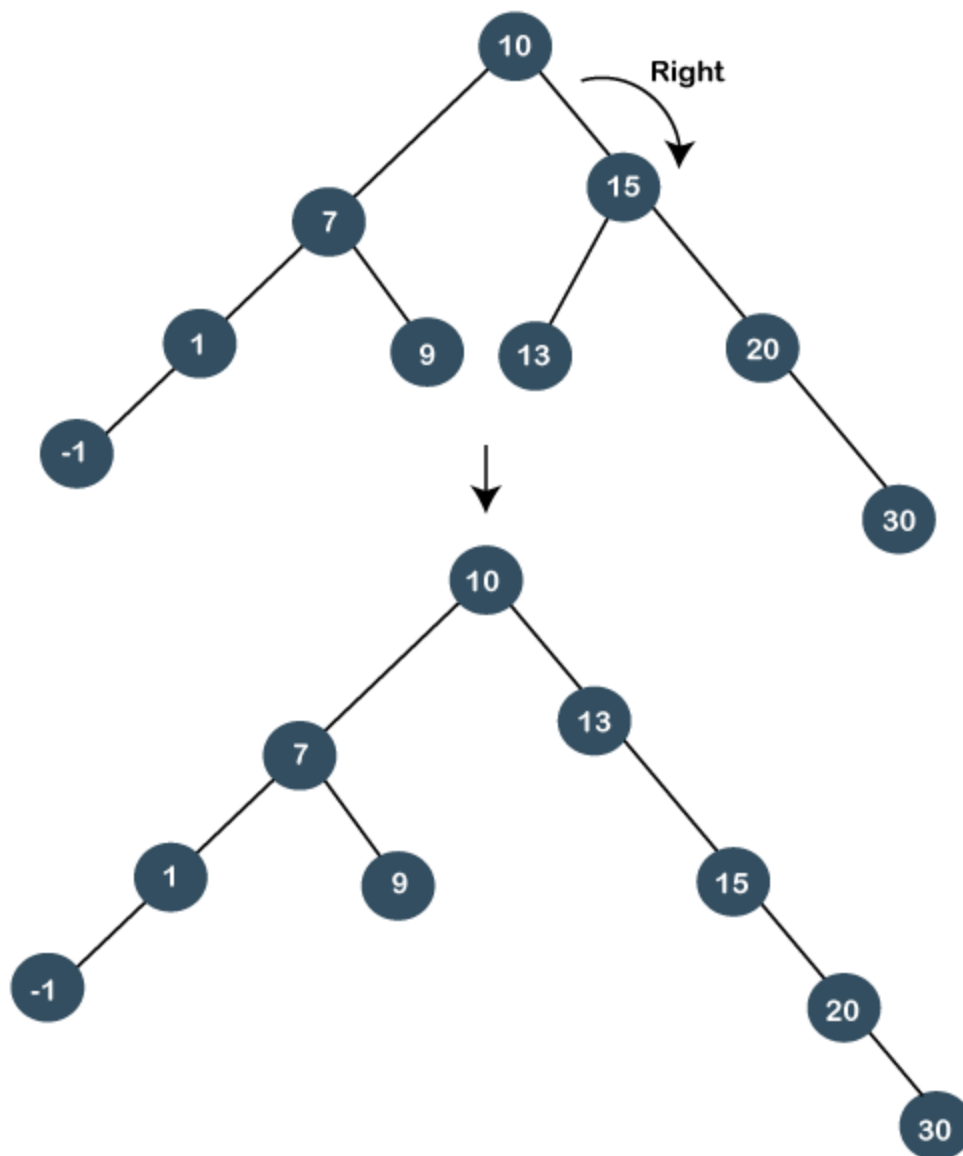
Suppose we want to search 13 element in the tree which is shown below:



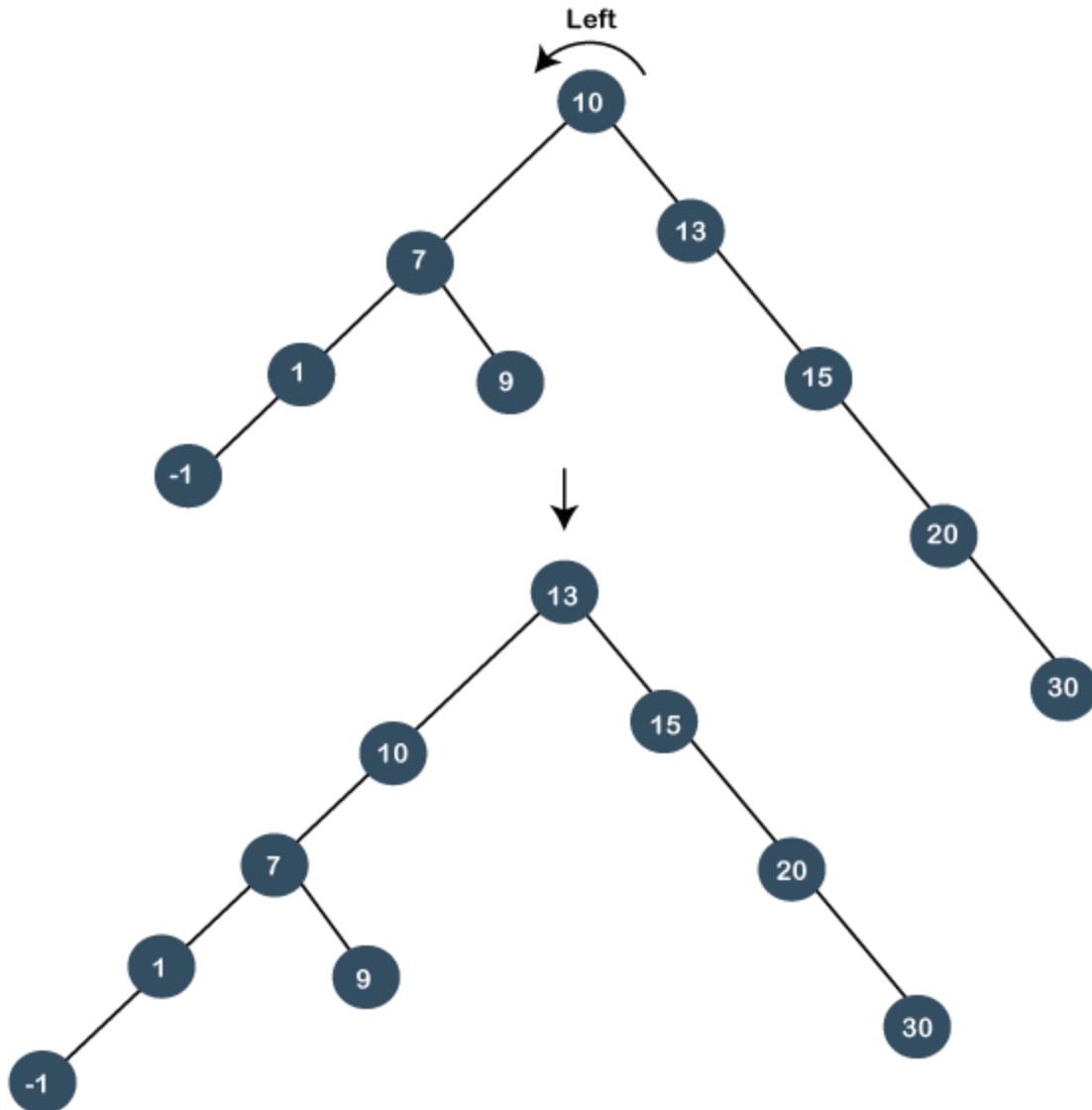
**Step 1:** First, we perform standard BST searching operation. As 13 is greater than 10 but less than 15, so node 13 will be the left child of node 15.

**Step 2:** Since node 13 is at the left of 15 and node 15 is at the right of node 10, so RL relationship exists. First, we perform the right rotation on node 15, and 15 will move downwards, and node 13 will come upwards, as shown below:





Still, node 13 is not the root node, and 13 is at the right of the root node, so we will perform left rotation known as a zag rotation. The node 10 will move downwards, and 13 becomes the root node as shown below:

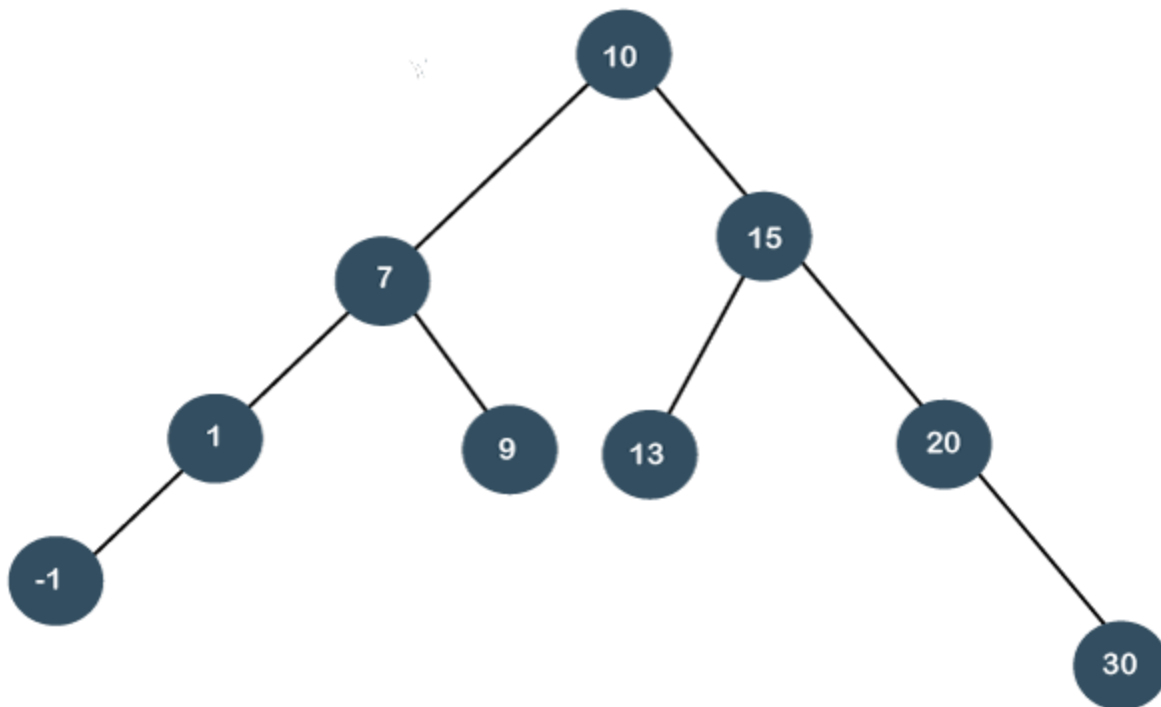


As we can observe in the above tree that node 13 has become the root node; therefore, the searching is completed. In this case, we have first performed the zig rotation and then zag rotation; so, it is known as a zig zag rotation.

- **Zag zig rotation**

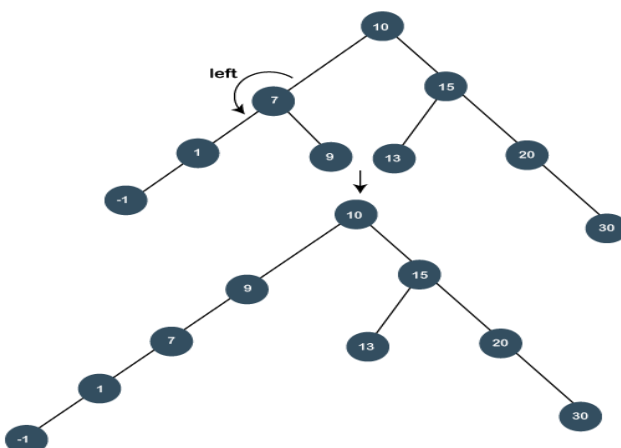
**Let's understand this case through an example.**

Suppose we want to search 9 element in the tree, which is shown below:

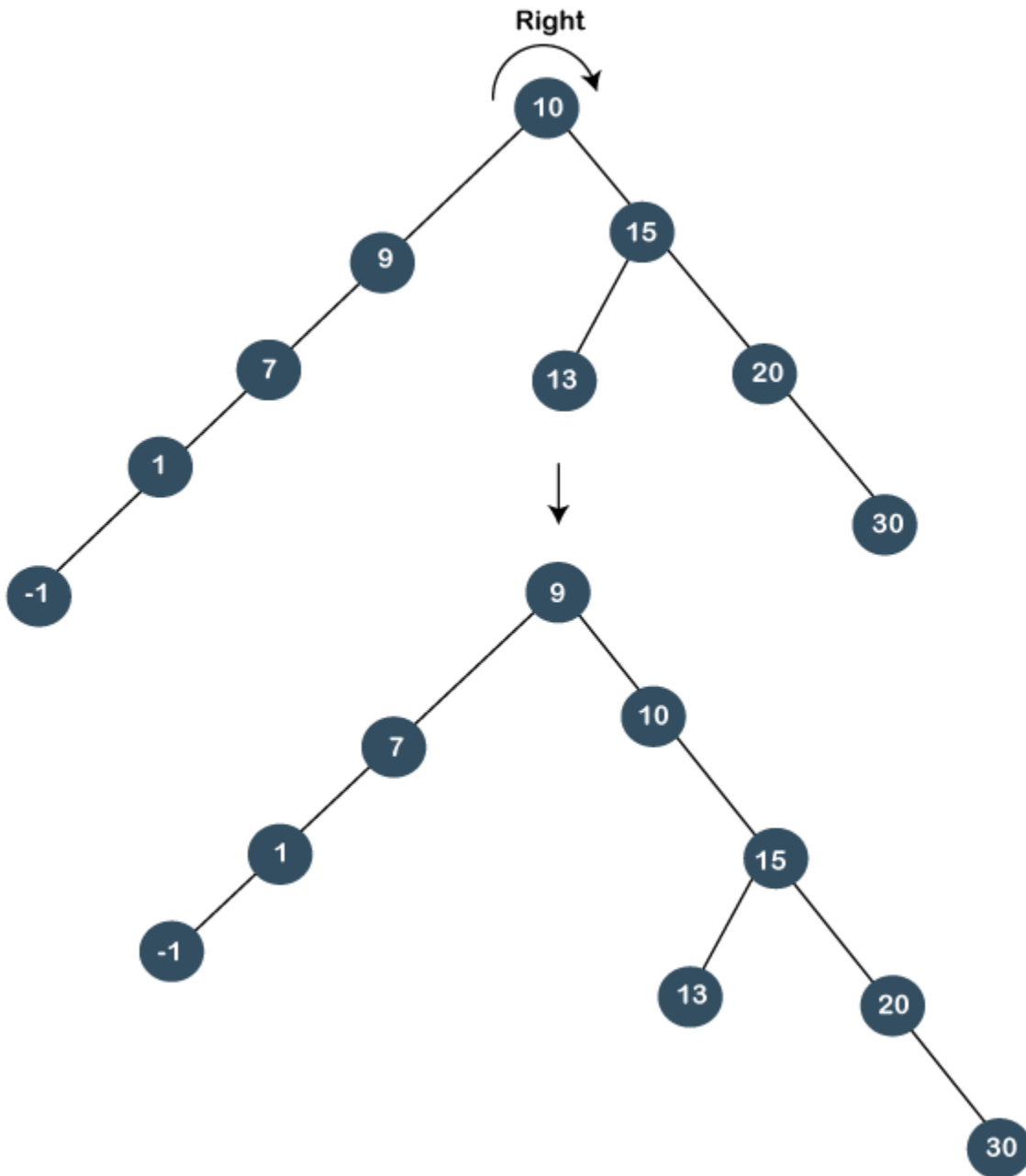


**Step 1:** First, we perform the standard BST searching operation. As 9 is less than 10 but greater than 7, so it will be the right child of node 7.

**Step 2:** Since node 9 is at the right of node 7, and node 7 is at the left of node 10, so LR relationship exists. First, we perform the left rotation on node 7. The node 7 will move downwards, and node 9 moves upwards as shown below:



Still the node 9 is not a root node, and 9 is at the left of the root node, so we will perform the right rotation known as zig rotation. After performing the right rotation, node 9 becomes the root node, as shown below:



As we can observe in the above tree that node 13 is a root node; therefore, the searching is completed. In this case, we have first performed the zag rotation (left rotation), and then zig rotation (right rotation) is performed, so it is known as a zag zig rotation.

## Advantages of Splay tree

- In the splay tree, we do not need to store the extra information. In contrast, in AVL trees, we need to store the balance factor of each node that requires extra space, and Red-Black trees also require to store one extra bit of information that denotes the color of the node, either Red or Black.
- It is the fastest type of Binary Search tree for various practical applications. It is used in **Windows NT and GCC compilers**.
- It provides better performance as the frequently accessed nodes will move nearer to the root node, due to which the elements can be accessed quickly in splay trees. It is used in the cache implementation as the recently accessed data is stored in the cache so that we do not need to go to the memory for accessing the data, and it takes less time.

## Drawback of Splay tree

The major drawback of the splay tree would be that trees are not strictly balanced, i.e., they are roughly balanced. Sometimes the splay trees are linear, so it will take  $O(n)$  time complexity.

### Insertion operation in Splay tree

In the **insertion** operation, we first insert the element in the tree and then perform the splaying operation on the inserted element.

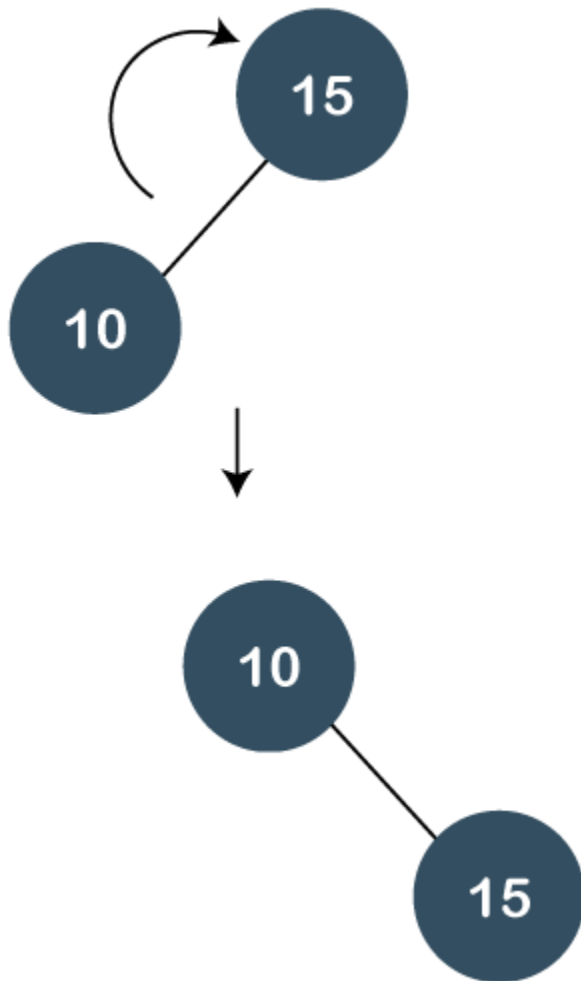
**15, 10, 17, 7**

**Step 1:** First, we insert node 15 in the tree. After insertion, we need to perform splaying. As 15 is a root node, so we do not need to perform splaying.



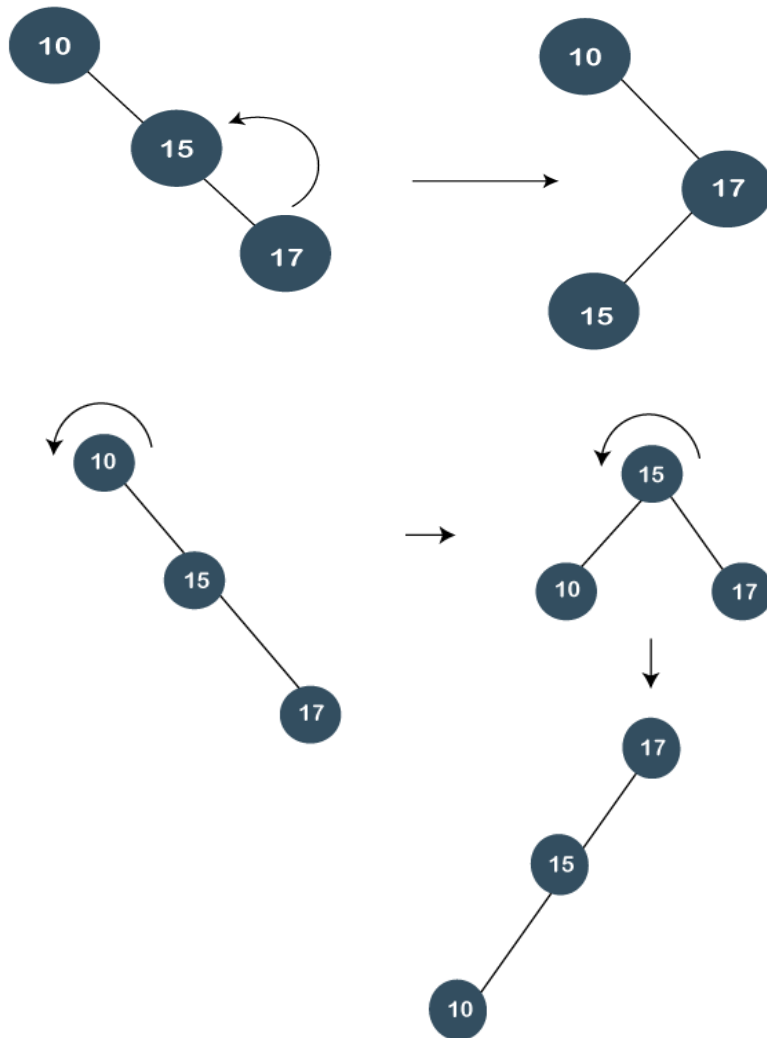
**Step 2:** The next element is 10. As 10 is less than 15, so node 10 will be the left child of node 15, as shown below:

Now, we perform **splaying**. To make 10 as a root node, we will perform the right rotation, as shown below:



**Step 3:** The next element is 17. As 17 is greater than 10 and 15 so it will become the right child of node 15.

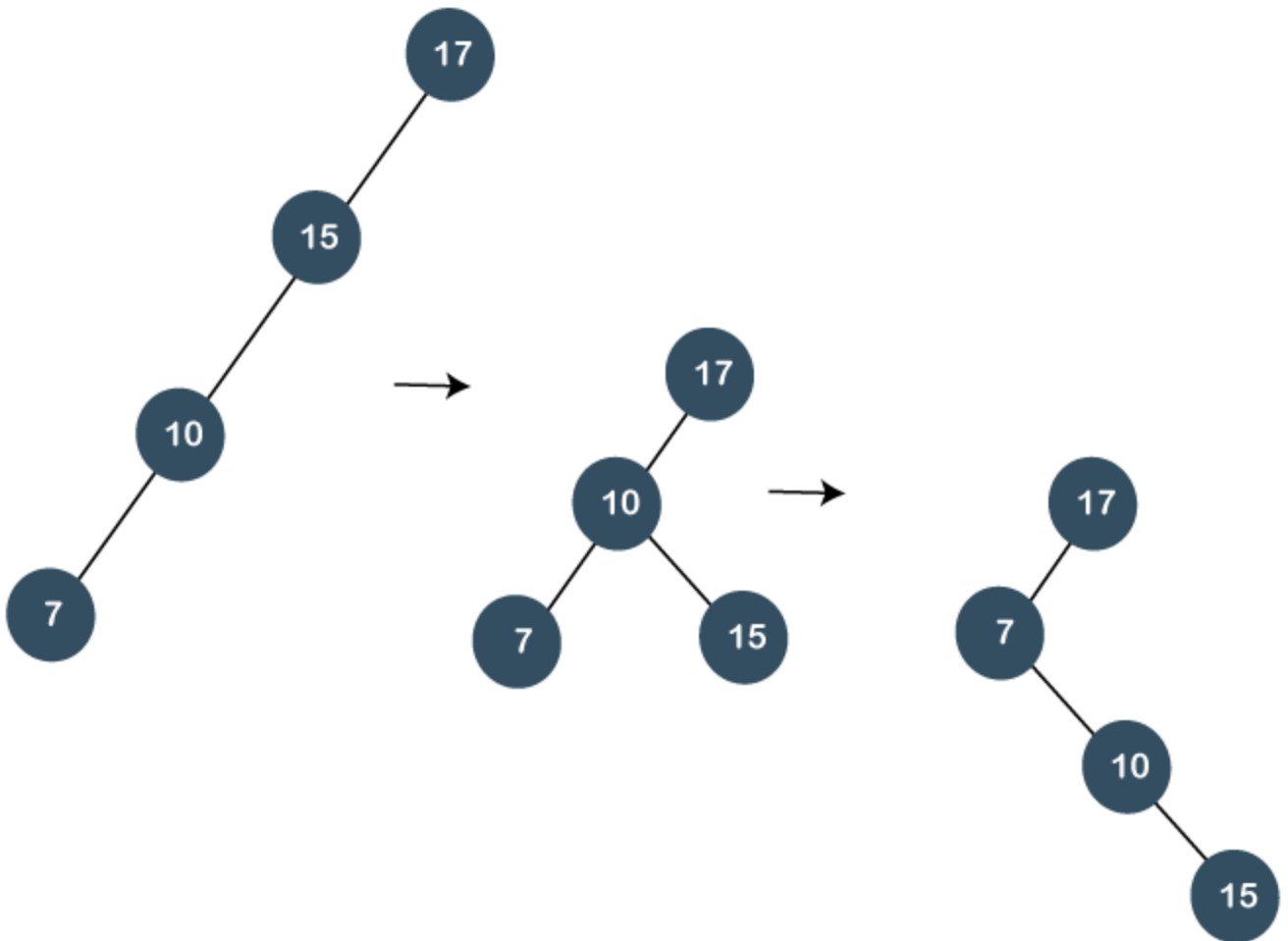
Now, we will perform splaying. As 17 is having a parent as well as a grandparent so we will perform zig zig rotations.



In the above figure, we can observe that 17 becomes the root node of the tree; therefore, the insertion is completed.

**Step 4:** The next element is 7. As 7 is less than 17, 15, and 10, so node 7 will be left child of 10.

Now, we have to splay the tree. As 7 is having a parent as well as a grandparent so we will perform two right rotations as shown below:



Still the node 7 is not a root node, it is a left child of the root node, i.e., 17. So, we need to perform one more right rotation to make node 7 as a root node as shown below:

