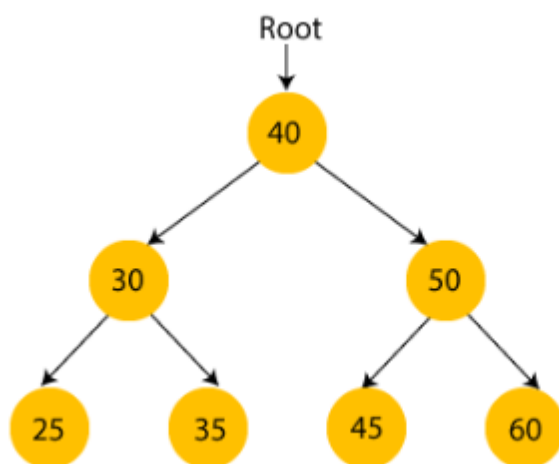


BINARY SEARCH TREE:-

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.

A binary search tree is a binary tree which is either empty or satisfies the following rules:-

- 1) The value of key in the left child or left sub tree is less than the value of the root.
- 2) The value of the key in the right child or right sub tree is more than or equal to the value of the root.
- 3) All the sub tree of the left and right children observe the two rules.



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.

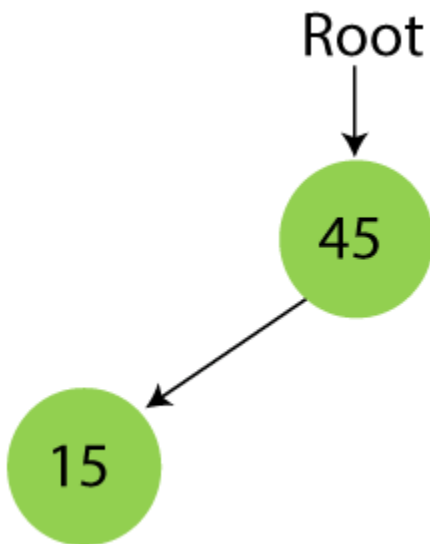
The process of creating the BST is shown below -

Step 1 - Insert 45.



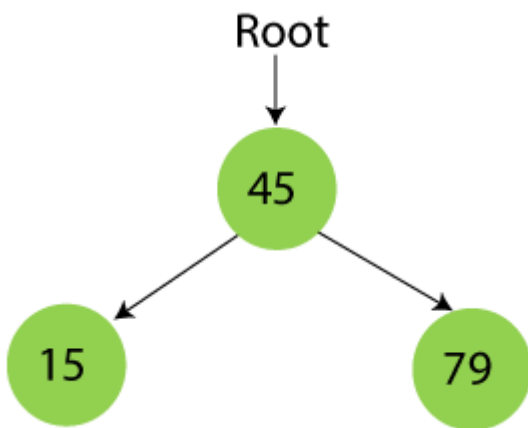
Step 2 - Insert 15.

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



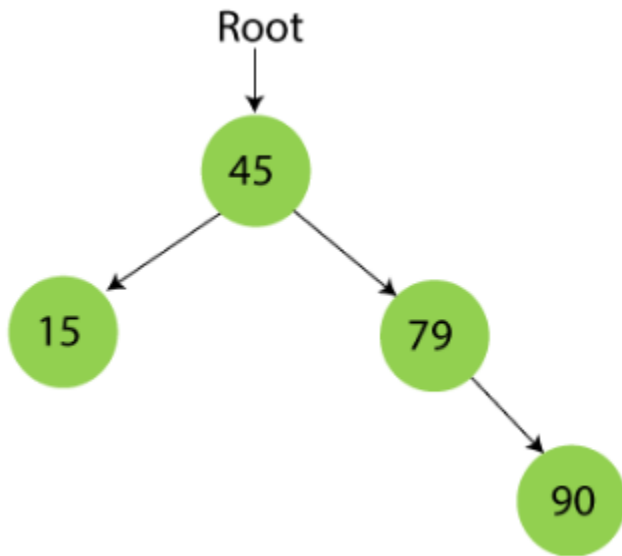
Step 3 - Insert 79.

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



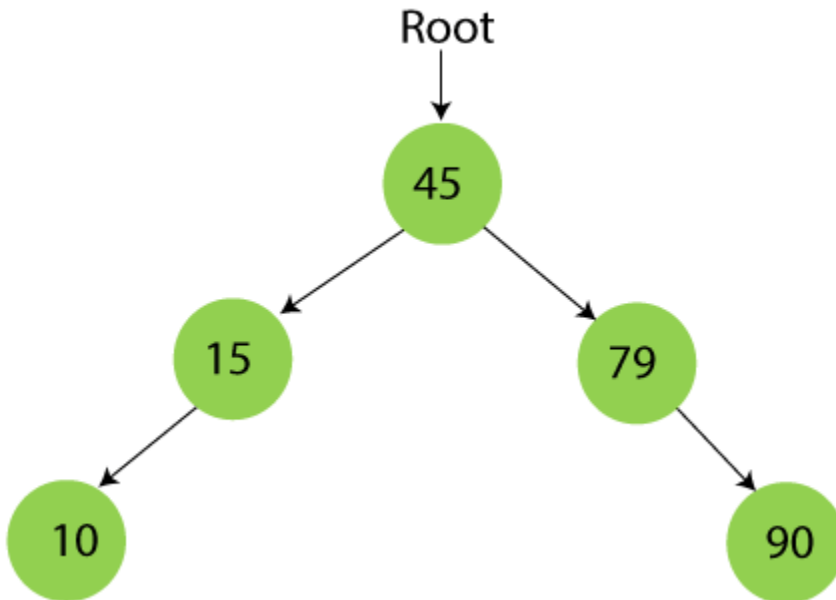
Step 4 - Insert 90.

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



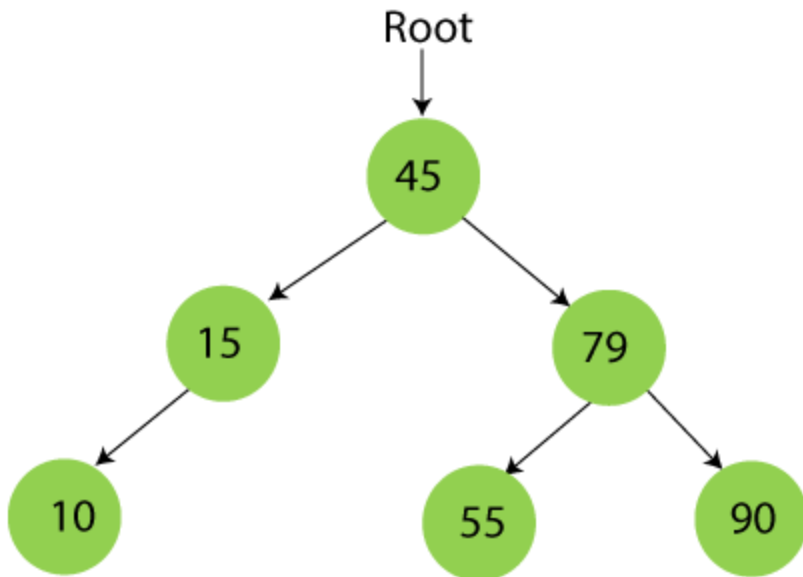
Step 5 - Insert 10.

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



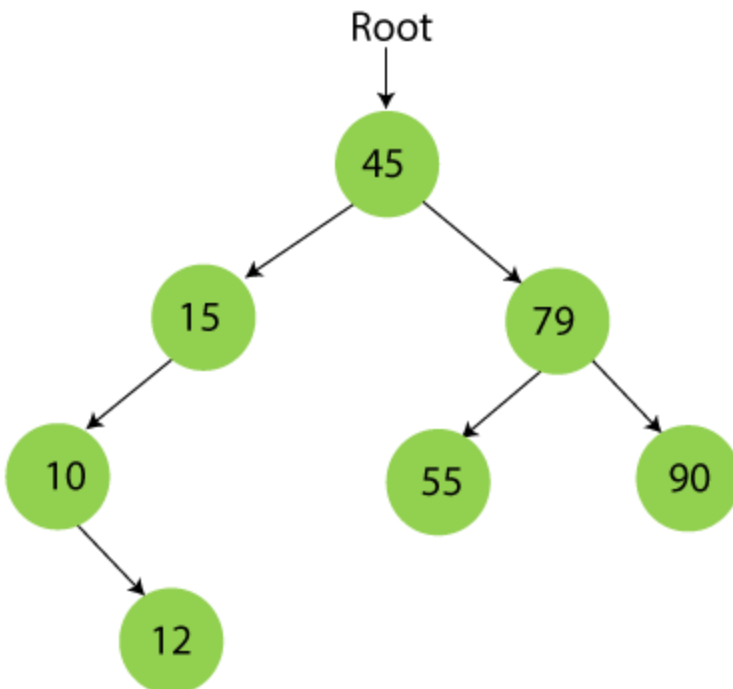
Step 6 - Insert 55.

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



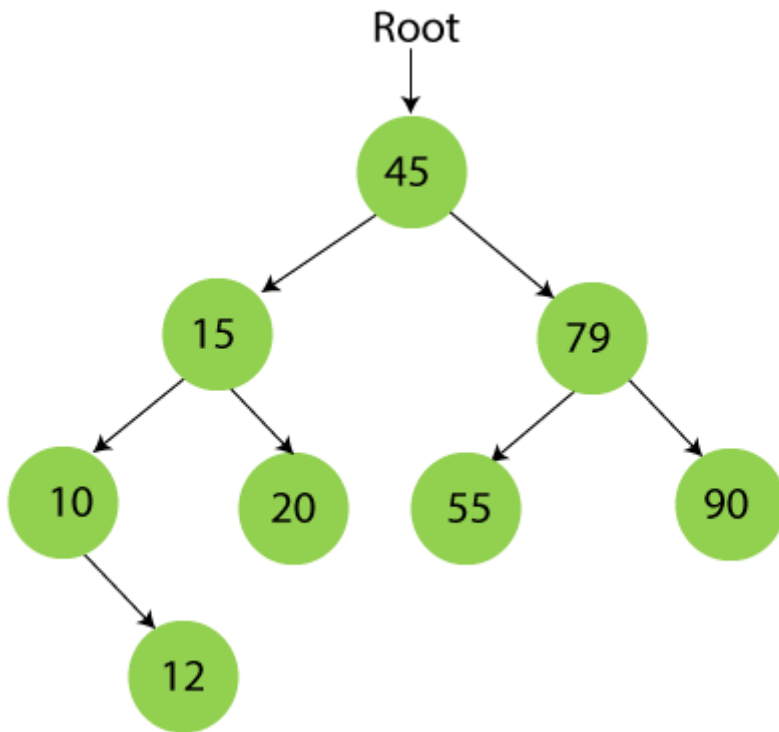
Step 7 - Insert 12.

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



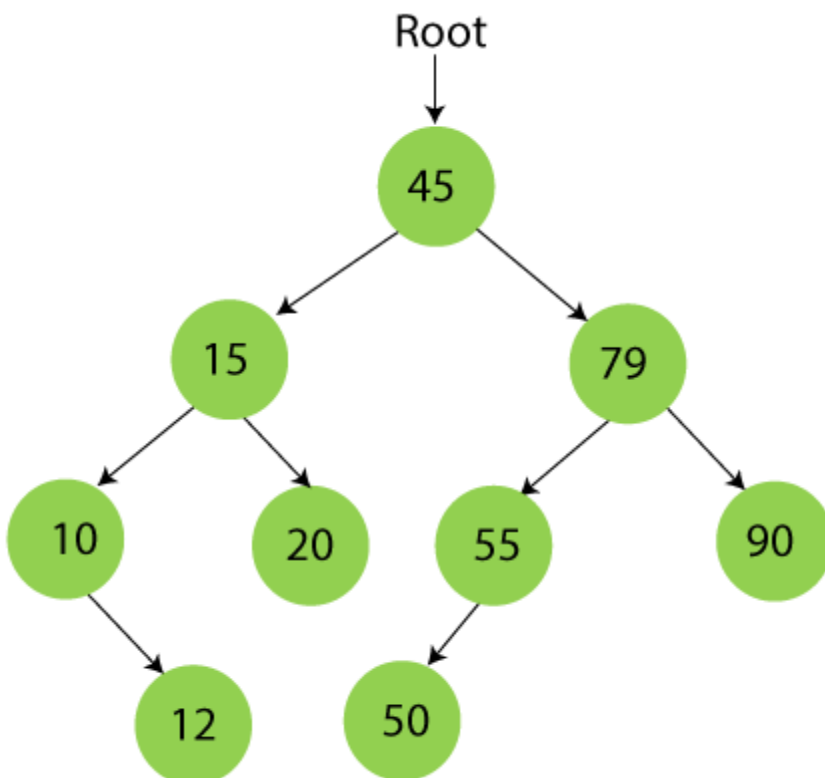
Step 8 - Insert 20.

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 sub tree of 55.



Now, the creation of binary search tree is completed.

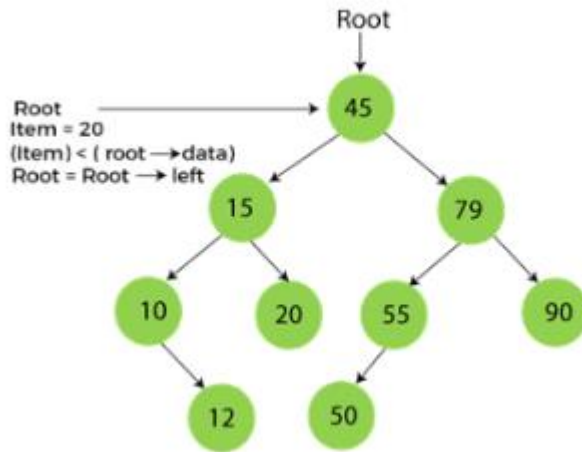
Searching in Binary search tree

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 –

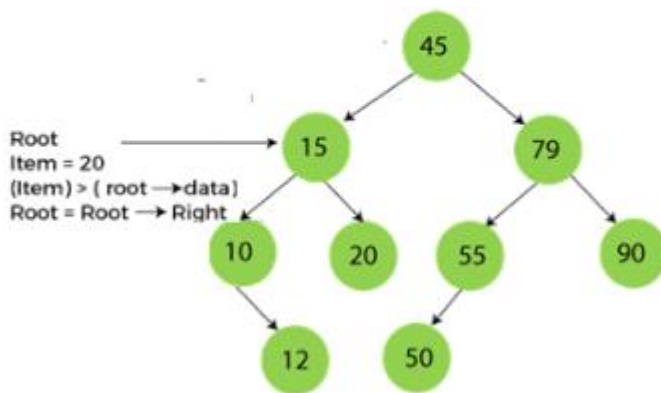
1. First, compare the element to be searched with the root element of the tree.
2. If root is matched with the target element, then return the node's location.
3. If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the 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.

Suppose we have to find node 20 from the below tree.

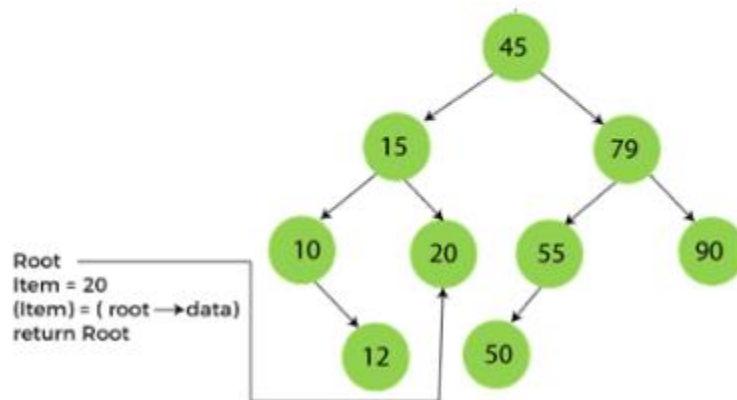
Step1:



Step2:



Step3:



Algorithm to search an element in Binary search tree

Search (root, item)

Step 1 - if (item = root → data) or (root = NULL)

return root

else if (item < root → data)

return Search(root → left, item)

else

return Search(root → right, item)

END if

Step 2 - END

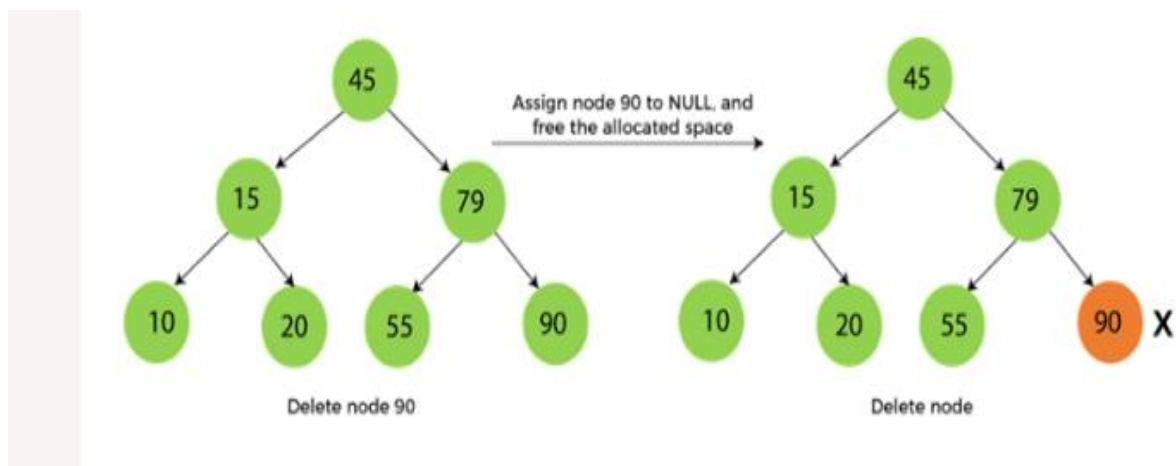
Deletion in Binary Search tree

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

1) When the node to be deleted is the leaf node

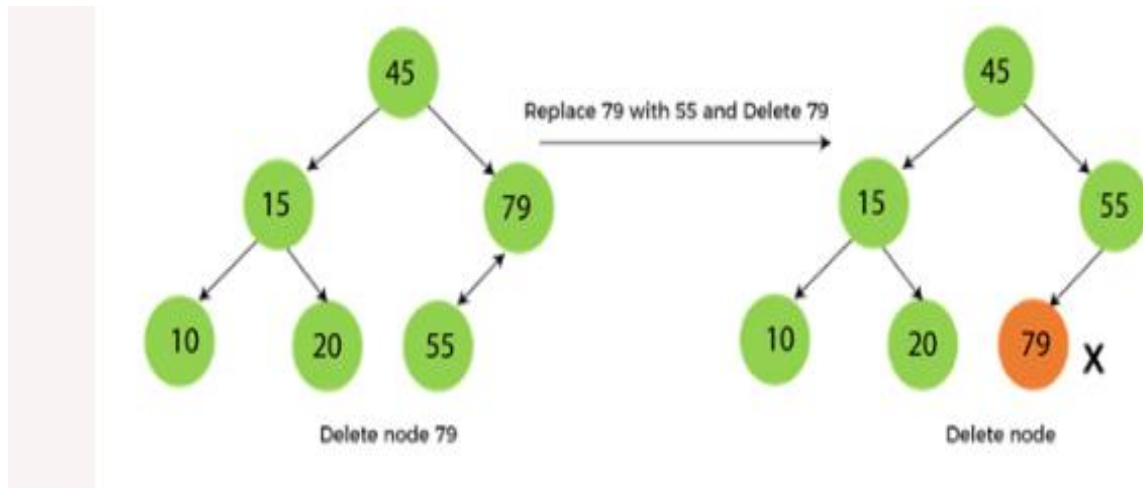
suppose we have to delete node 90, as the node to be deleted is a leaf node, so it will be replaced with NULL, and the allocated space will free.



2) When the node to be deleted has only one child

suppose we have to delete the node 79, as the node to be deleted has only one child, so it will be replaced with its child 55.

So, the replaced node 79 will now be a leaf node that can be easily deleted.



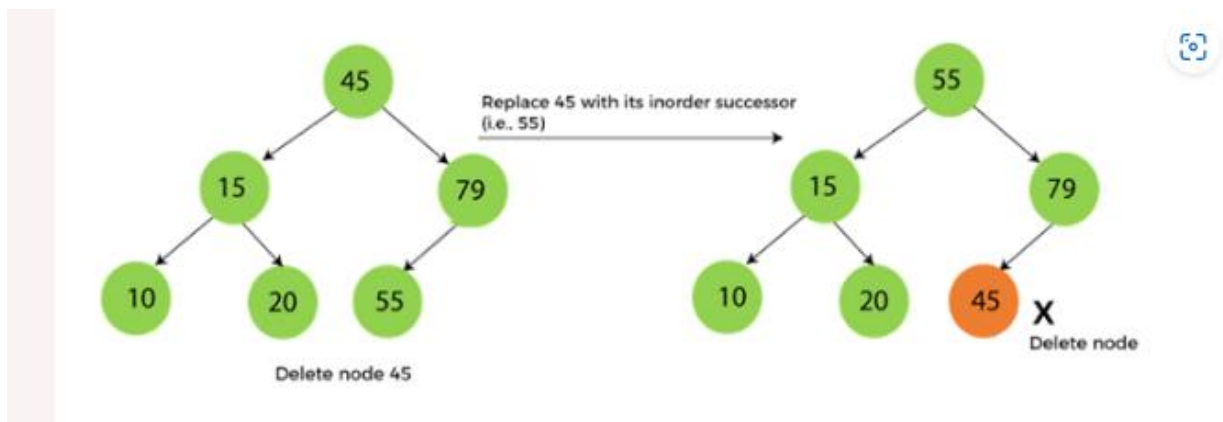
When the node to be deleted has two children

In such a case, the steps to be followed are listed as follows -

- First, find the inorder successor of the node to be deleted.
- After that, replace that node with the inorder successor until the target node is placed at the leaf of tree.
- And at last, replace the node with NULL and free up the allocated space.

We have to delete node 45 that is the root node, as the node to be deleted has two children, so it will be replaced with its inorder

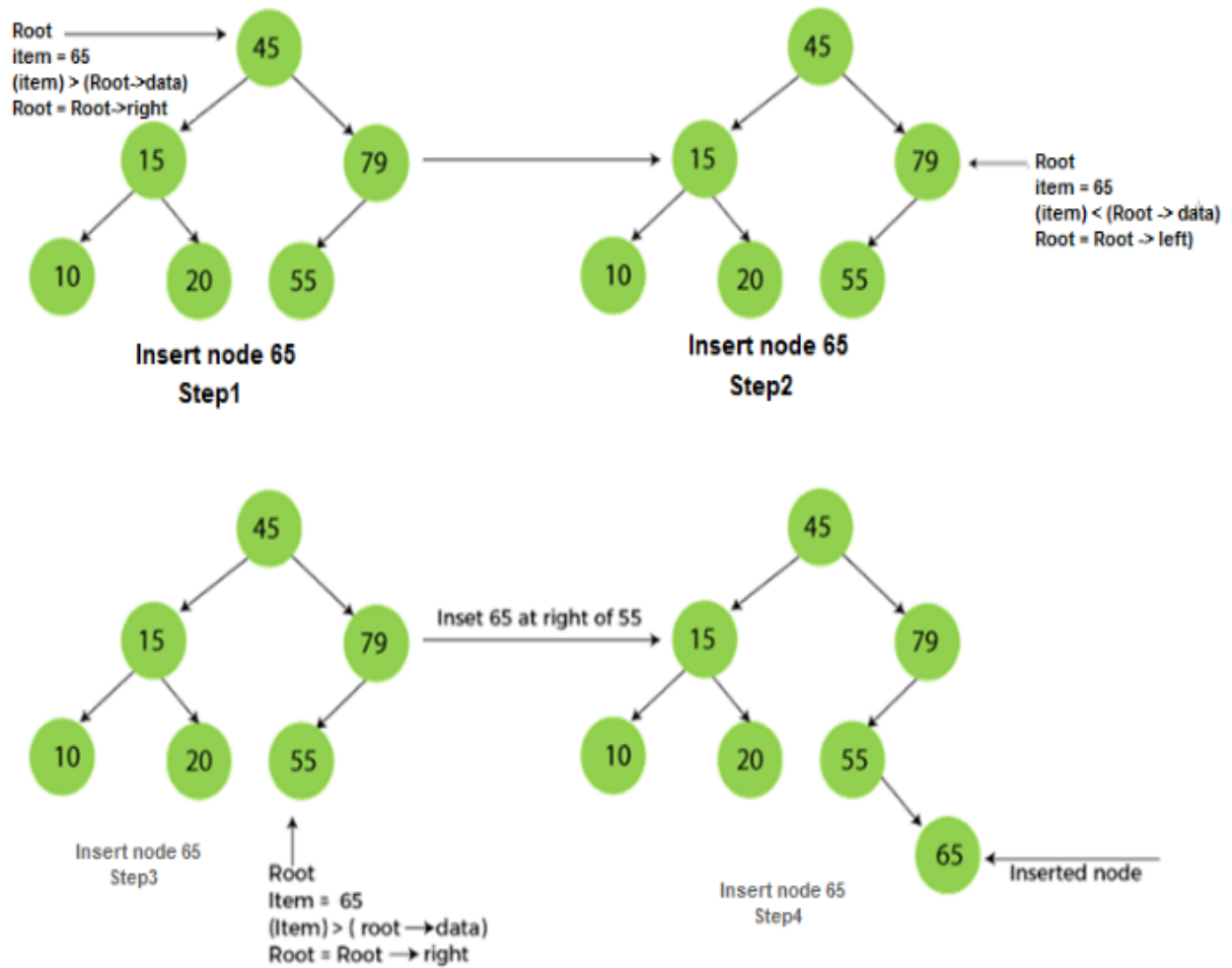
successor. Now, node 45 will be at the leaf of the tree so that it can be deleted easily.



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.



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	$O(\log n)$	$O(\log n)$	$O(n)$

2. Space Complexity

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

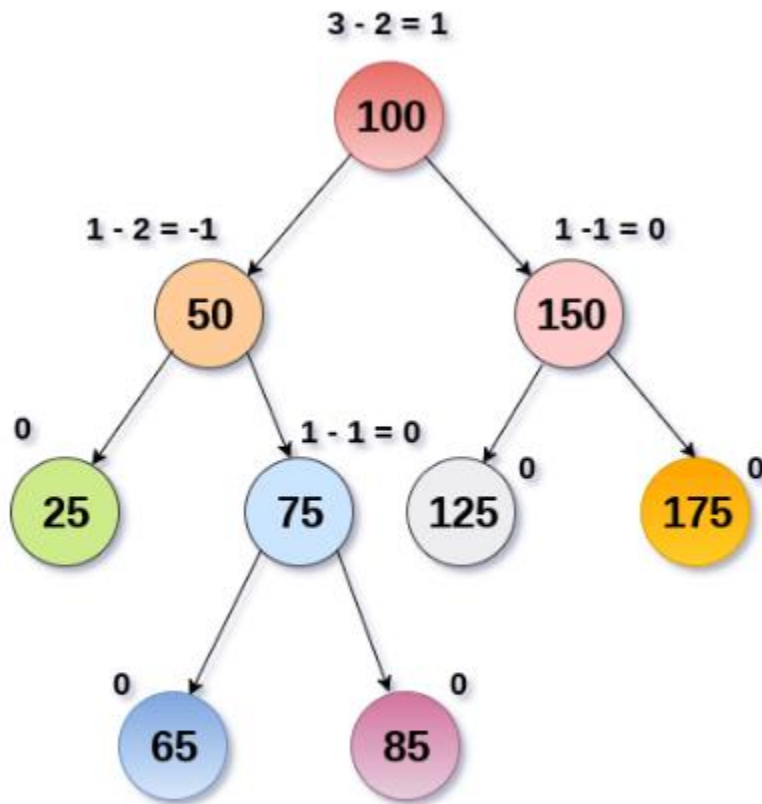
AVL Tree

AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honor 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 0, -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

Balance factor = height of left sub tree – height of right sub 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	$o(\log n)$	$o(\log n)$

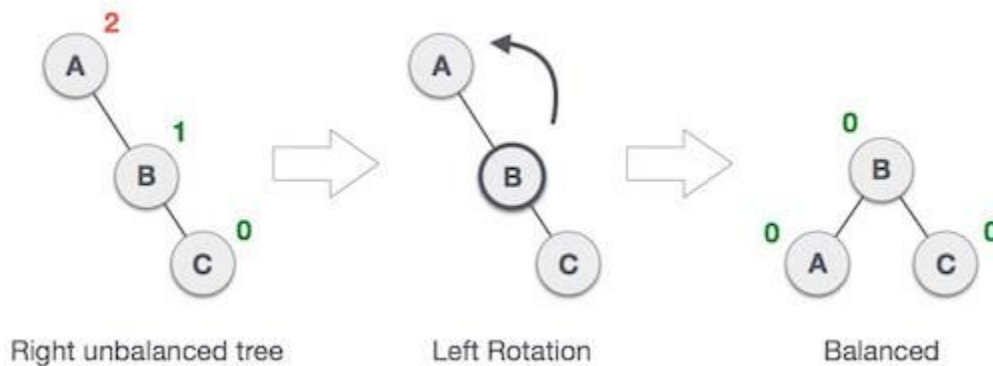
AVL Rotations

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

1. RR Rotation

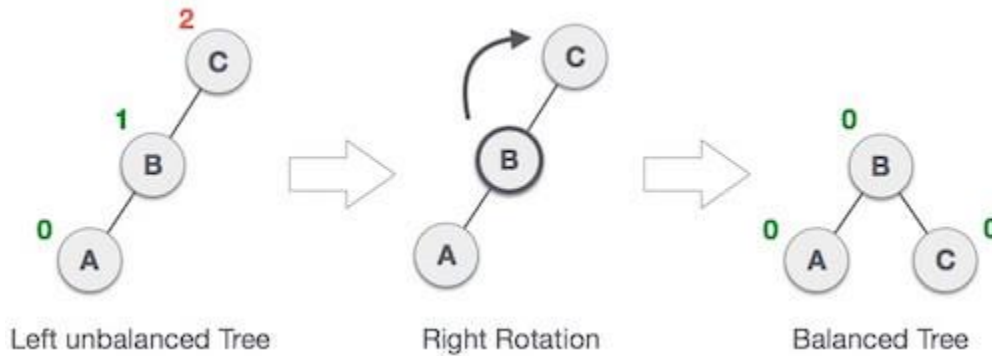
When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, **RR rotation** is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



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.

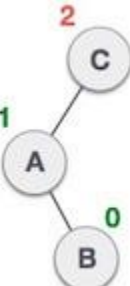
2. LL Rotation

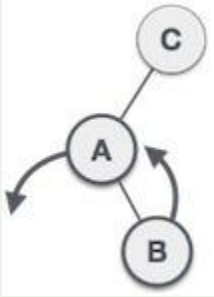
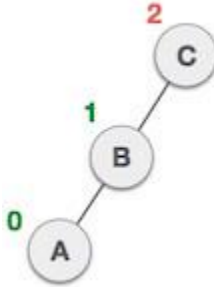
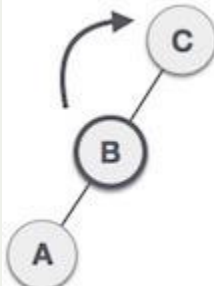
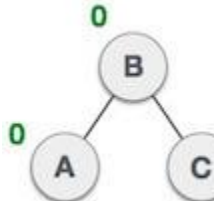
When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, **LL rotation** is clockwise rotation, which is applied on the edge below a node having balance factor 2.



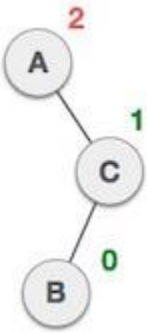
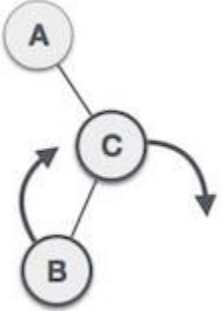
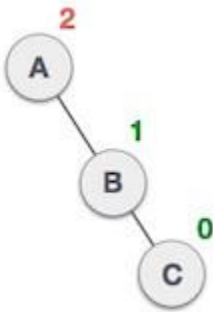
In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

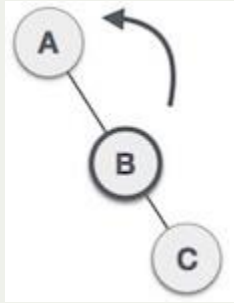
3. LR Rotation:- LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on sub tree and then LL rotation is performed on full tree.

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>

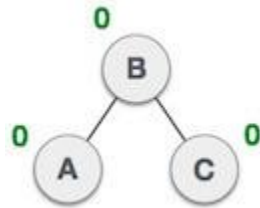
	<p>As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node A, has become the left subtree of B.</p>
	<p>After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of C</p>
	<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</p>
	<p>Balance factor of each node is now either</p>

5. **RL 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
	<p>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</p>
	<p>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.</p>
	<p>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.</p>



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.

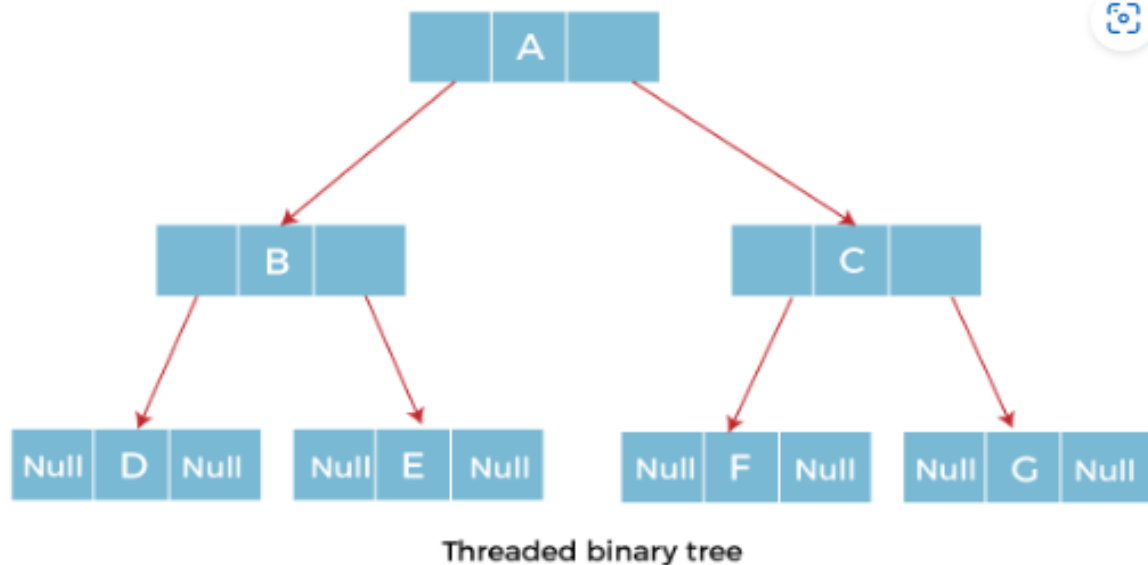


Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.

THREADED BINARY TREE

In threaded binary tree the special pointer called thread is used to point to nodes higher in the tree.

In the linked representation of binary trees, more than one half of the link fields contain NULL values which results in wastage of storage space. If a binary tree consists of n nodes then $n+1$ link fields contain NULL values. So in order to effectively manage the space, a method was devised by Perlis and Thornton in which the NULL links are replaced with special links known as threads. Such binary trees with threads are known as **threaded binary trees**. Each node in a threaded binary tree either contains a link to its child node or thread to other nodes in the tree.



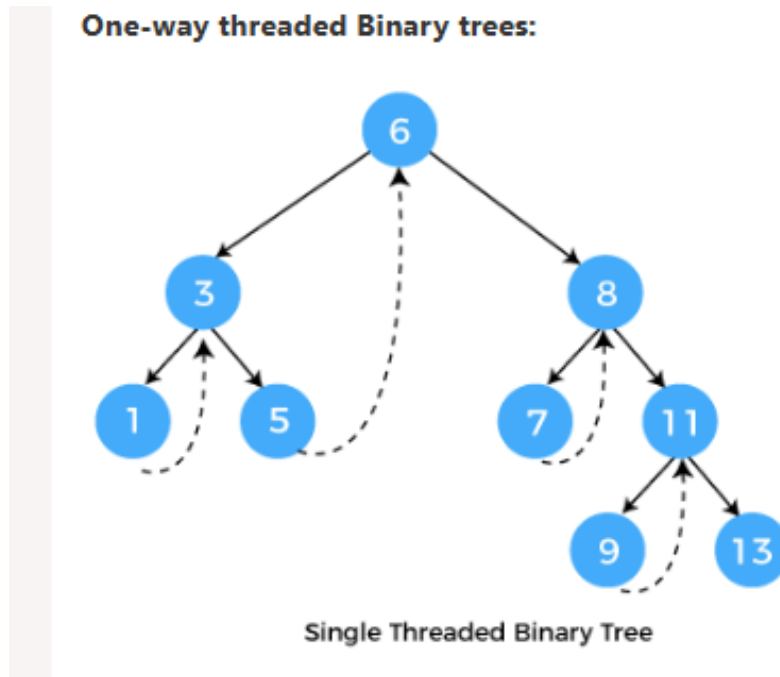
Types of Threaded Binary Tree

There are two types of threaded Binary Tree:

- One-way threaded Binary Tree
- Two-way threaded Binary Tree

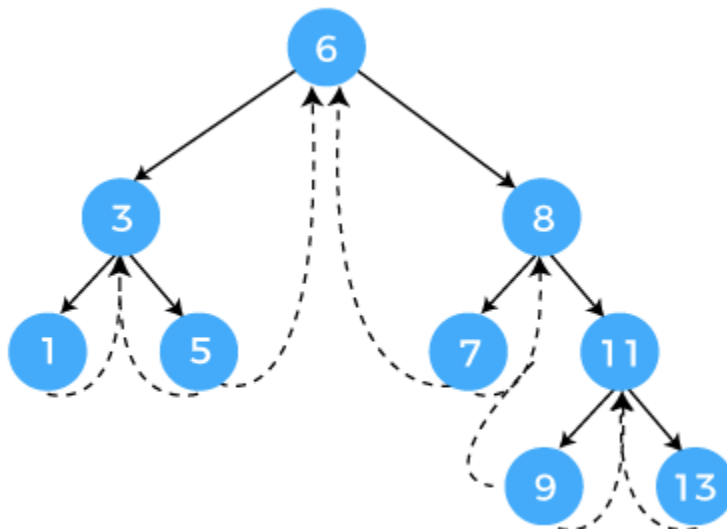
1) One-way threaded Binary Tree:- In one-way threaded binary trees, a thread will appear either in the right or left link field of a node. If it appears in the right link field of a node then it will point to the next node that will appear on performing in order traversal. Such trees are called **Right**

threaded binary trees. If thread appears in the left field of a node then it will point to the nodes inorder predecessor. Such



trees are called **Left threaded binary trees.** Left threaded binary trees are used less often as they don't yield the last advantages of right threaded binary trees. In one-way threaded binary trees, the right link field of last node and left link field of first node contains a NULL. In order to distinguish threads from normal links they are represented by dotted lines.

2. **Two-way threaded Binary trees:** - The right link field of a node containing NULL values is replaced by a thread that points to nodes inorder successor and left field of a node containing NULL values is replaced by a thread that points to nodes in order predecessor.



Double Threaded Binary Tree

B Tree: - B-tree is a self balance search tree in which entry node contains multiple keys and has more than two children.

B-tree of order m has the following properties.

1. All leaf nodes must be at same level.
2. All nodes except root must have at least $m/2 - 1$ keys and minimum of $m-1$ keys.

3. Each node has a maximum of m children and a minimum of $m/2$ children.
4. All the key value in a node must be in ascending order.