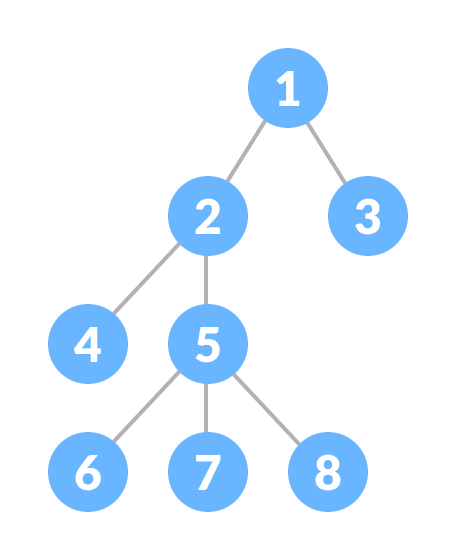
**Tree**

A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges.

A Tree

**Why Tree Data Structure?**

Other data structures such as arrays, linked list, stack, and queue are linear data structures that store data sequentially. In order to perform any operation in a linear data structure, the time complexity increases with the increase in the data size. But, it is not acceptable in today's computational world.

Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure.

**Tree Terminologies**

**Node**

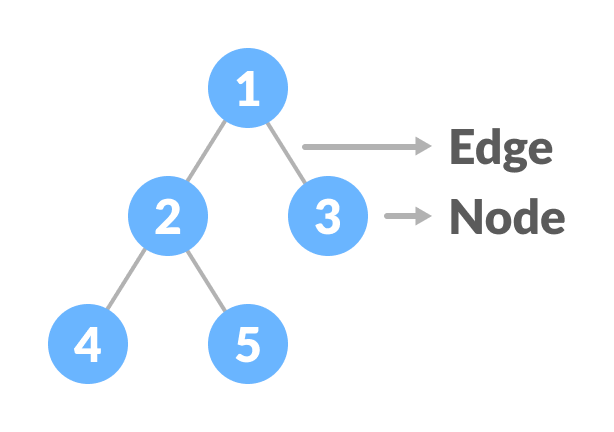
A node is an entity that contains a key or value and pointers to its child nodes.

The last nodes of each path are called **leaf nodes or external nodes** that do not contain a link/pointer to child nodes.

The node having at least a child node is called an **internal node**.

**Edge**

It is the link between any two nodes.

Nodes and edges of a tree

**Root**

It is the topmost node of a tree.

**Height of a Node**

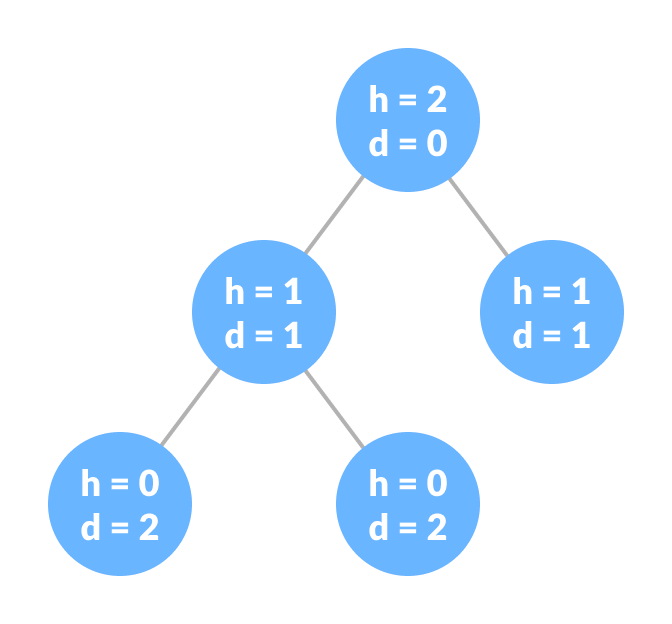
The height of a node is the number of edges from the node to the deepest leaf (ie. the longest path from the node to a leaf node).

**Depth of a Node**

The depth of a node is the number of edges from the root to the node.

**Height of a Tree**

The height of a Tree is the height of the root node or the depth of the deepest node.

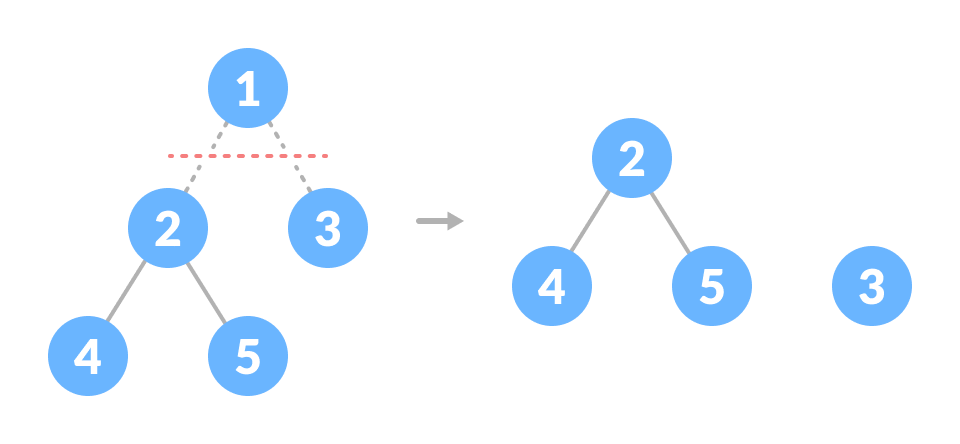
Height and depth of each node in a tree

**Degree of a Node**

The degree of a node is the total number of branches of that node.

**Forest**

A collection of disjoint trees is called a forest.

Creating forest from a tree

You can create a forest by cutting the root of a tree.

**Types of Tree**

1. [Binary Tree](https://www.programiz.com/dsa/binary-tree)
2. [Binary Search Tree](https://www.programiz.com/dsa/binary-search-tree)
3. [AVL Tree](https://www.programiz.com/dsa/avl-tree)
4. [B-Tree](https://www.programiz.com/dsa/b-tree)

**Tree Traversal**

In order to perform any operation on a tree, you need to reach to the specific node. The tree traversal algorithm helps in visiting a required node in the tree.

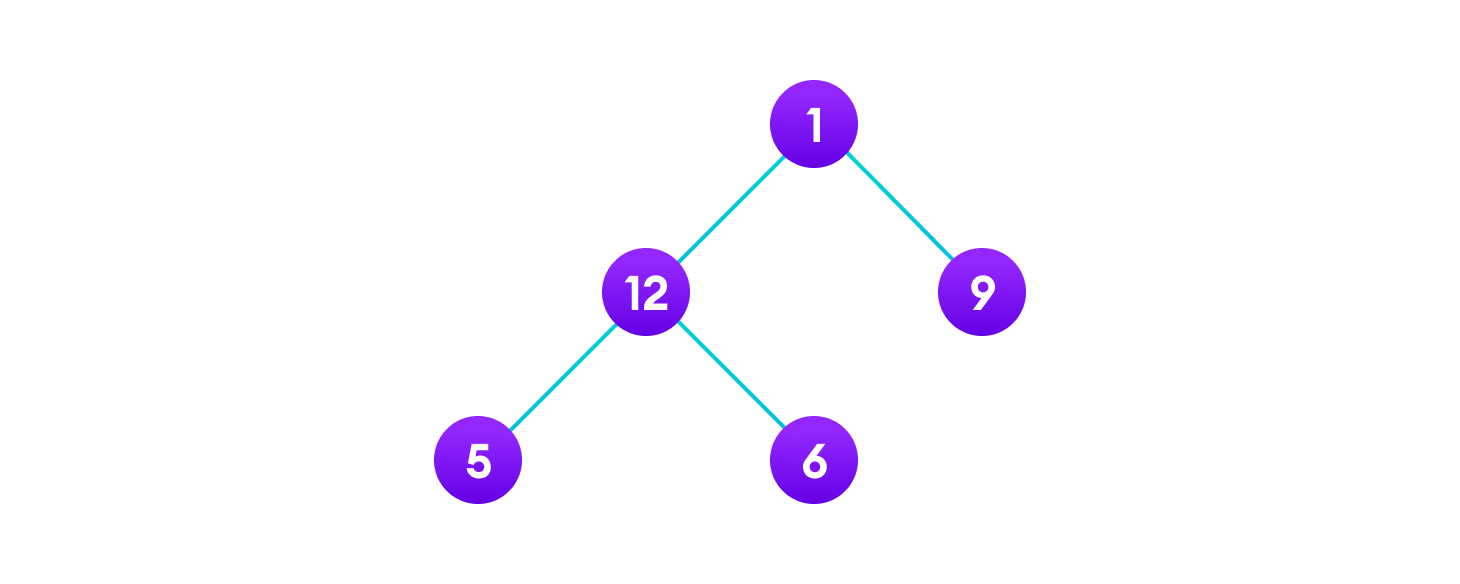
**Tree Applications**

* Binary Search Trees(BSTs) are used to quickly check whether an element is present in a set or not.
* Heap is a kind of tree that is used for heap sort.
* A modified version of a tree called Tries is used in modern routers to store routing information.
* Most popular databases use B-Trees and T-Trees, which are variants of the tree structure we learned above to store their data
* Compilers use a syntax tree to validate the syntax of every program you write.

# Tree Traversal - inorder, preorder and postorder

Traversing a tree means visiting every node in the tree. You might, for instance, want to add all the values in the tree or find the largest one. For all these operations, you will need to visit each node of the tree.

Linear data structures like arrays, [stacks](https://www.programiz.com/data-structures/stack), [queues](https://www.programiz.com/data-structures/queue), and [linked list](https://www.programiz.com/data-structures/linked-list) have only one way to read the data. But a hierarchical data structure like a [tree](https://www.programiz.com/data-structures/trees) can be traversed in different ways.

Tree traversal

Let's think about how we can read the elements of the tree in the image shown above.

Starting from top, Left to right

1 -> 12 -> 5 -> 6 -> 9

Starting from bottom, Left to right

5 -> 6 -> 12 -> 9 -> 1

Although this process is somewhat easy, it doesn't respect the hierarchy of the tree, only the depth of the nodes.

Instead, we use traversal methods that take into account the basic structure of a tree i.e.

struct node {

int data;

struct node\* left;

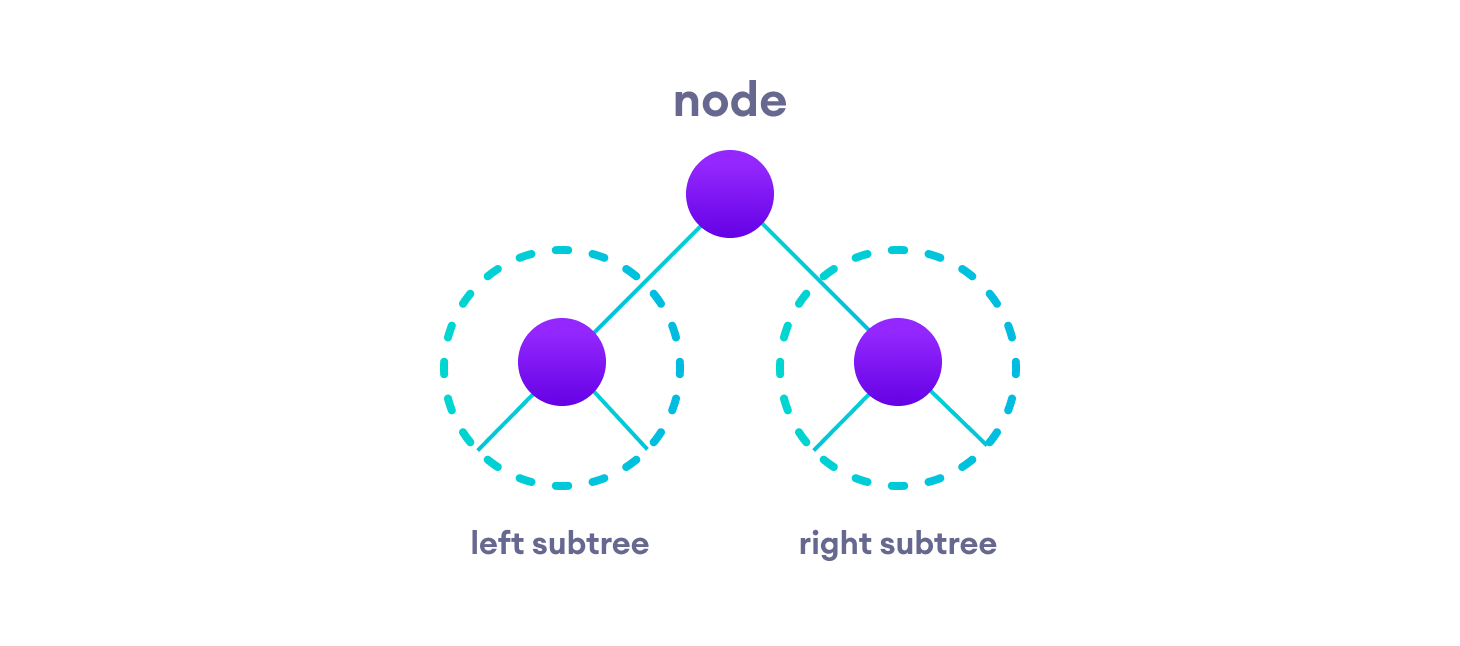
struct node\* right;

}

The struct node pointed to by left and right might have other left and right children so we should think of them as sub-trees instead of sub-nodes.

According to this structure, every tree is a combination of

* A node carrying data
* Two subtrees

Left and Right Subtree

Remember that our goal is to visit each node, so we need to visit all the nodes in the subtree, visit the root node and visit all the nodes in the right subtree as well.

Depending on the order in which we do this, there can be three types of traversal.

## Inorder traversal ( Left->Root->Right)

1. First, visit all the nodes in the left subtree
2. Then the root node
3. Visit all the nodes in the right subtree

inorder(root->left)

display(root->data)

inorder(root->right)

## Preorder traversal

1. Visit root node
2. Visit all the nodes in the left subtree
3. Visit all the nodes in the right subtree

display(root->data)

preorder(root->left)

preorder(root->right)

## Postorder traversal

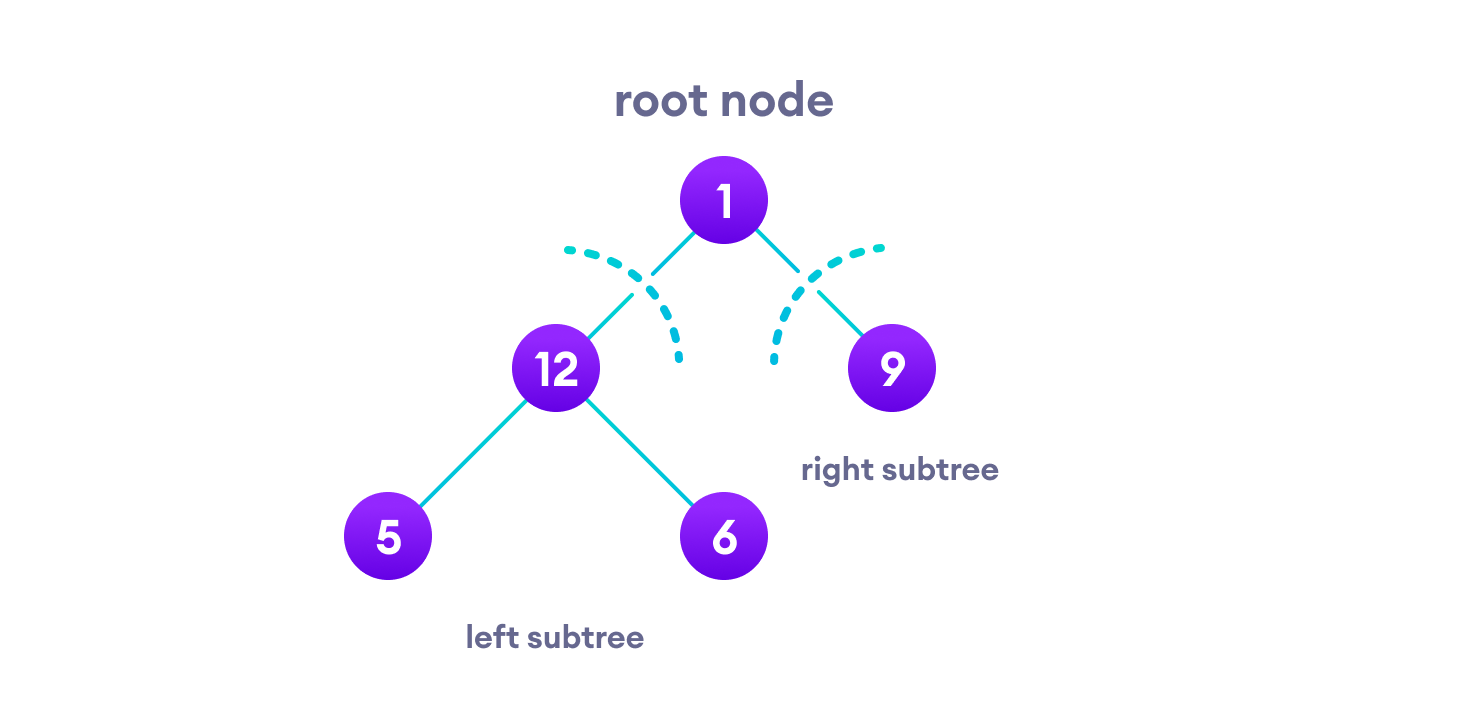
1. Visit all the nodes in the left subtree
2. Visit all the nodes in the right subtree
3. Visit the root node

postorder(root->left)

postorder(root->right)

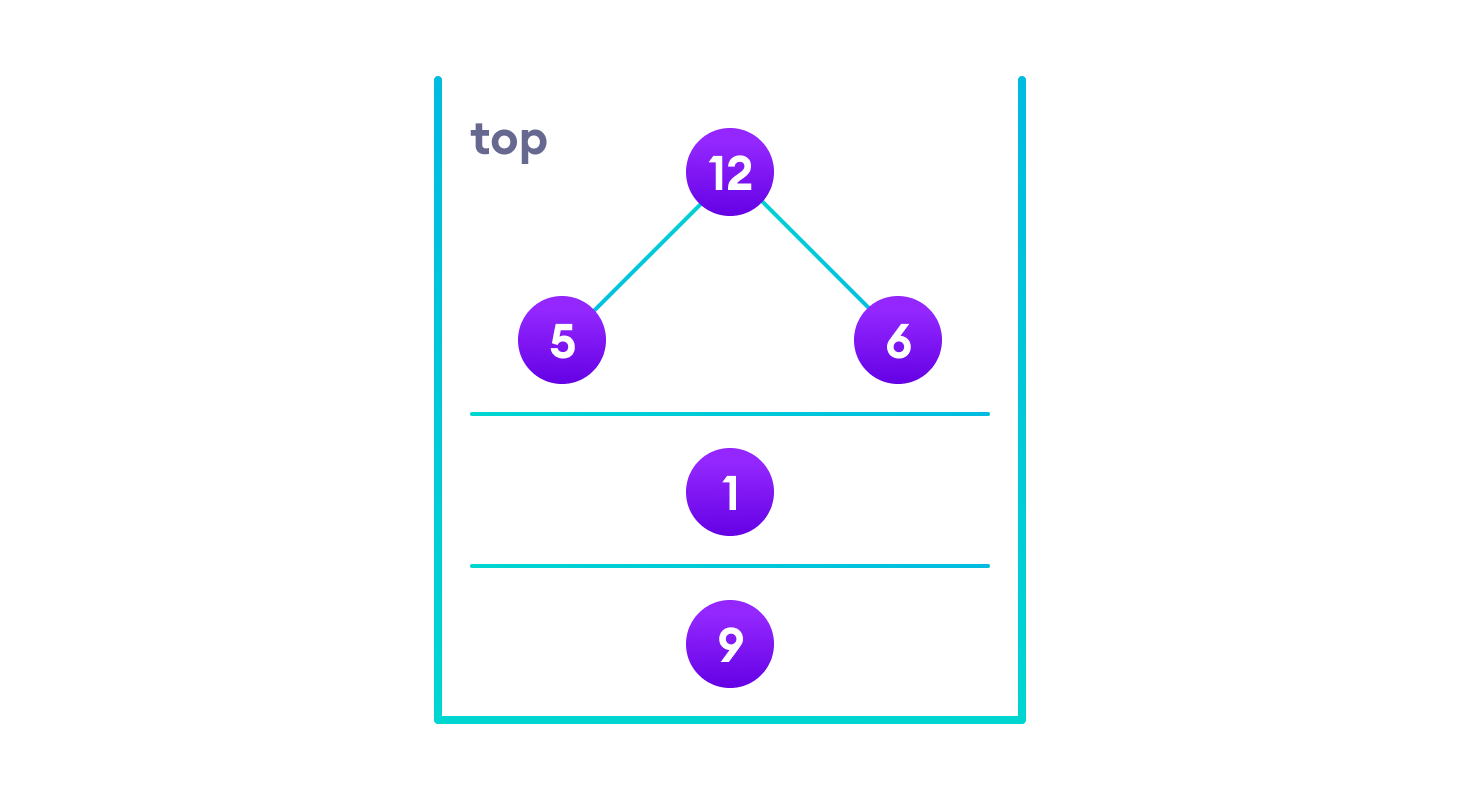
display(root->data)

Let's visualize in-order traversal. We start from the root node.

Left and Right Subtree

We traverse the left subtree first. We also need to remember to visit the root node and the right subtree when this tree is done.

Let's put all this in a [stack](https://www.programiz.com/data-structures/stack) so that we remember.

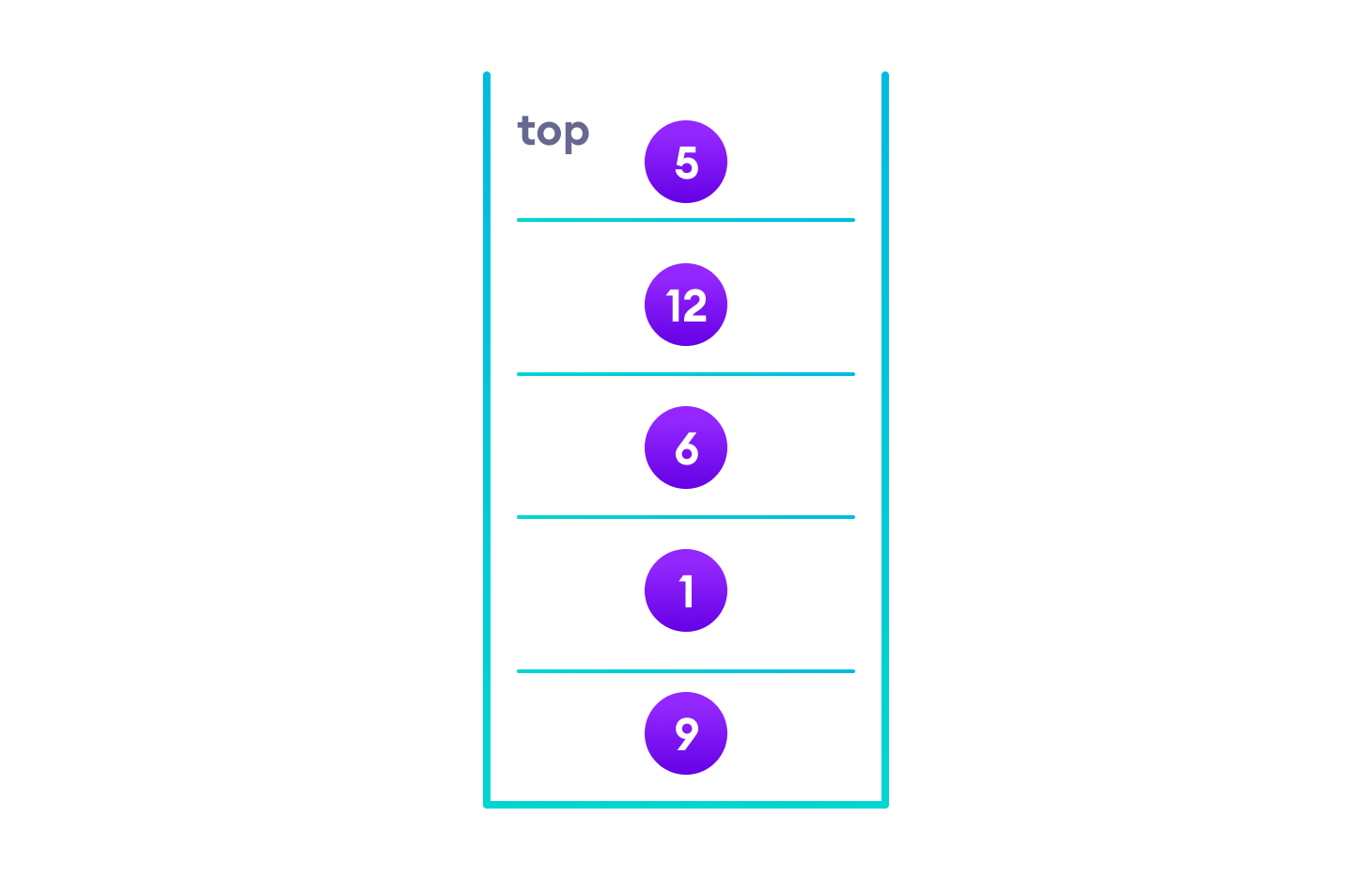
Stack

Now we traverse to the subtree pointed on the TOP of the stack.

Again, we follow the same rule of inorder

Left subtree -> root -> right subtree

After traversing the left subtree, we are left with

Final Stack

Since the node "5" doesn't have any subtrees, we print it directly. After that we print its parent "12" and then the right child "6".

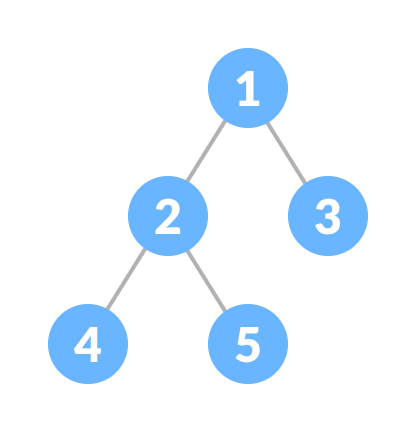
Putting everything on a stack was helpful because now that the left-subtree of the root node has been traversed, we can print it and go to the right subtree.

After going through all the elements, we get the inorder traversal as

5 -> 12 -> 6 -> 1 -> 9

A binary tree is a tree data structure in which each parent node can have at most two children.

For example: In the image below, each element has at most two children.

Binary Tree

**Types of Binary Tree**

**Full Binary Tree**

A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children.

Full Binary Tree

To learn more, please visit [full binary tree](https://www.programiz.com/dsa/full-binary-tree).

**Perfect Binary Tree**

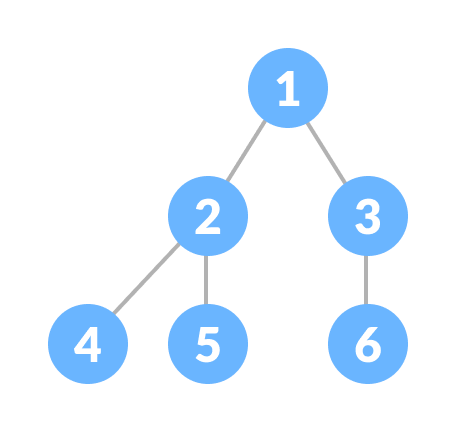
A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.

Perfect Binary Tree

**Complete Binary Tree**

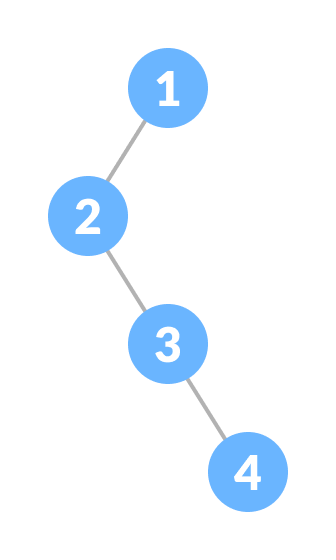
A complete binary tree is just like a full binary tree, but with two major differences

1. Every level must be completely filled
2. All the leaf elements must lean towards the left.
3. The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.

Complete Binary Tree

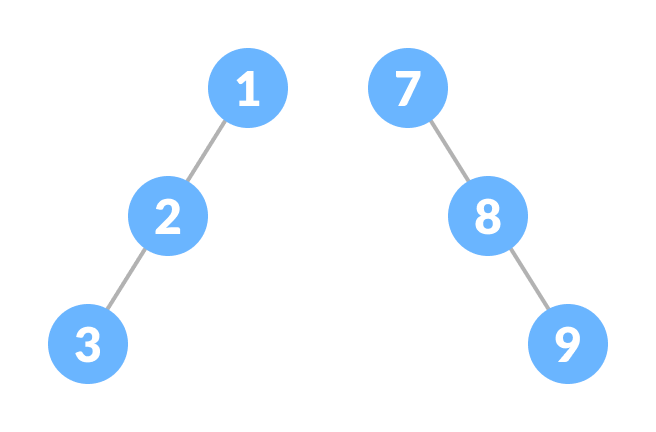
**Degenerate or Pathological Tree**

A degenerate or pathological tree is the tree having a single child either left or right.

Degenerate Binary Tree

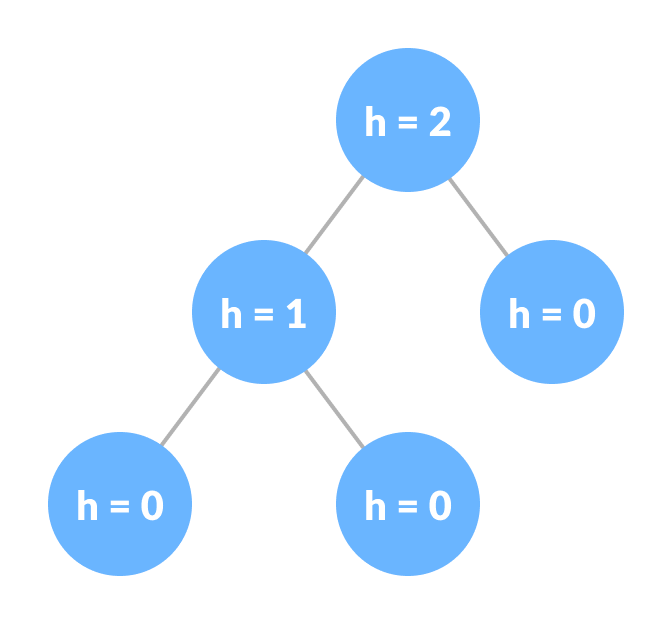
**Skewed Binary Tree**

A skewed binary tree is a pathological/degenerate tree in which the tree is either dominated by the left nodes or the right nodes. Thus, there are two types of skewed binary tree: **left-skewed binary tree** and **right-skewed binary tree**.

Skewed Binary Tree

**Balanced Binary Tree**

It is a type of binary tree in which the difference between the left and the right subtree for each node is either 0 or 1.

Balanced Binary Tree

**Binary Tree Representation**

A node of a binary tree is represented by a structure containing a data part and two pointers to other structures of the same type.

struct node

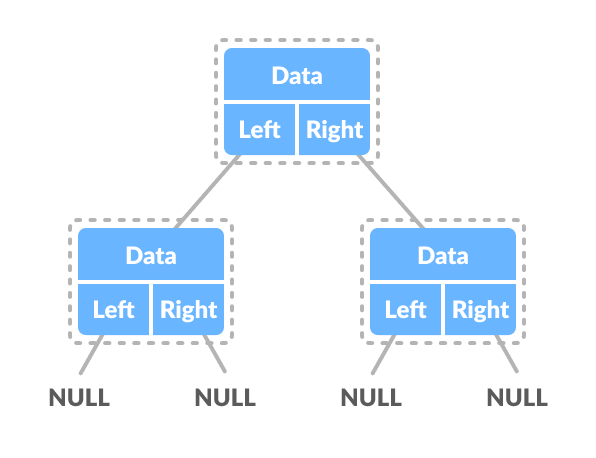
{

int data;

struct node \*left;

struct node \*right;

};



**Full Binary Tree**

A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children.

It is also known as**a proper binary tree**.

Full Binary Tree

**Full Binary Tree Theorems**

Let, i = the number of internal nodes

n = be the total number of nodes

l = number of leaves

λ = number of levels

1. The number of leaves is i + 1.
2. The total number of nodes is 2i + 1.
3. The number of internal nodes is (n – 1) / 2.
4. The number of leaves is (n + 1) / 2.
5. The total number of nodes is 2l – 1.
6. The number of internal nodes is l – 1.
7. The number of leaves is at most 2λ - 1.

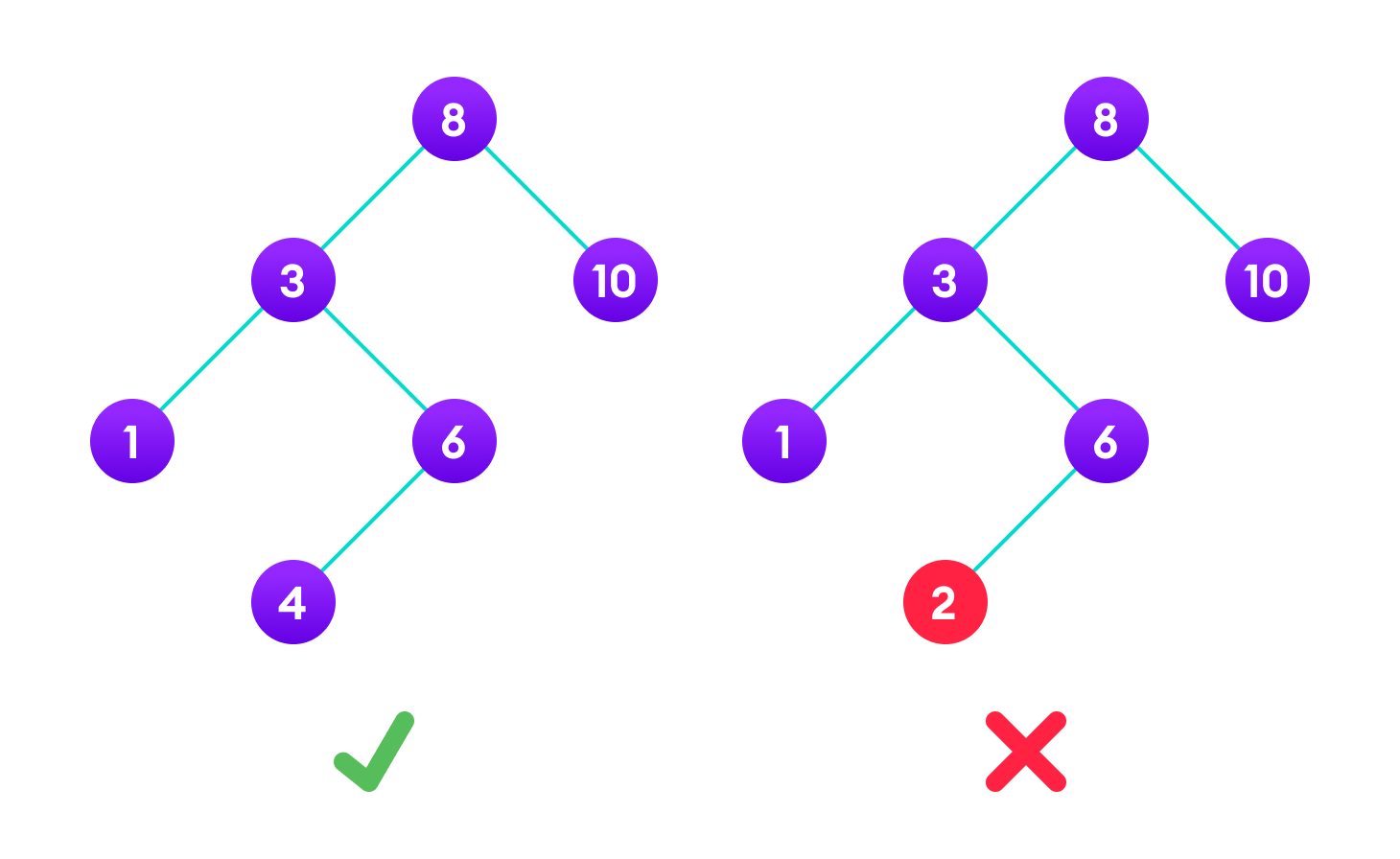
**Binary search tree**

Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.

* It is called a binary tree because each tree node has a maximum of two children.
* It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time.

The properties that separate a binary search tree from a regular [binary tree](https://www.programiz.com/data-structures/trees) is

1. All nodes of left subtree are less than the root node
2. All nodes of right subtree are more than the root node
3. Both subtrees of each node are also BSTs i.e. they have the above two properties

A tree having a right subtree with one value smaller than the root is shown to demonstrate that it is not a valid binary search tree

The binary tree on the right isn't a binary search tree because the right subtree of the node "3" contains a value smaller than it.

There are two basic operations that you can perform on a binary search tree:

**Search Operation**

The algorithm depends on the property of BST that if each left subtree has values below root and each right subtree has values above the root.

If the value is below the root, we can say for sure that the value is not in the right subtree; we need to only search in the left subtree and if the value is above the root, we can say for sure that the value is not in the left subtree; we need to only search in the right subtree.

**Algorithm:**

If root == NULL

return NULL;

If number == root->data

return root->data;

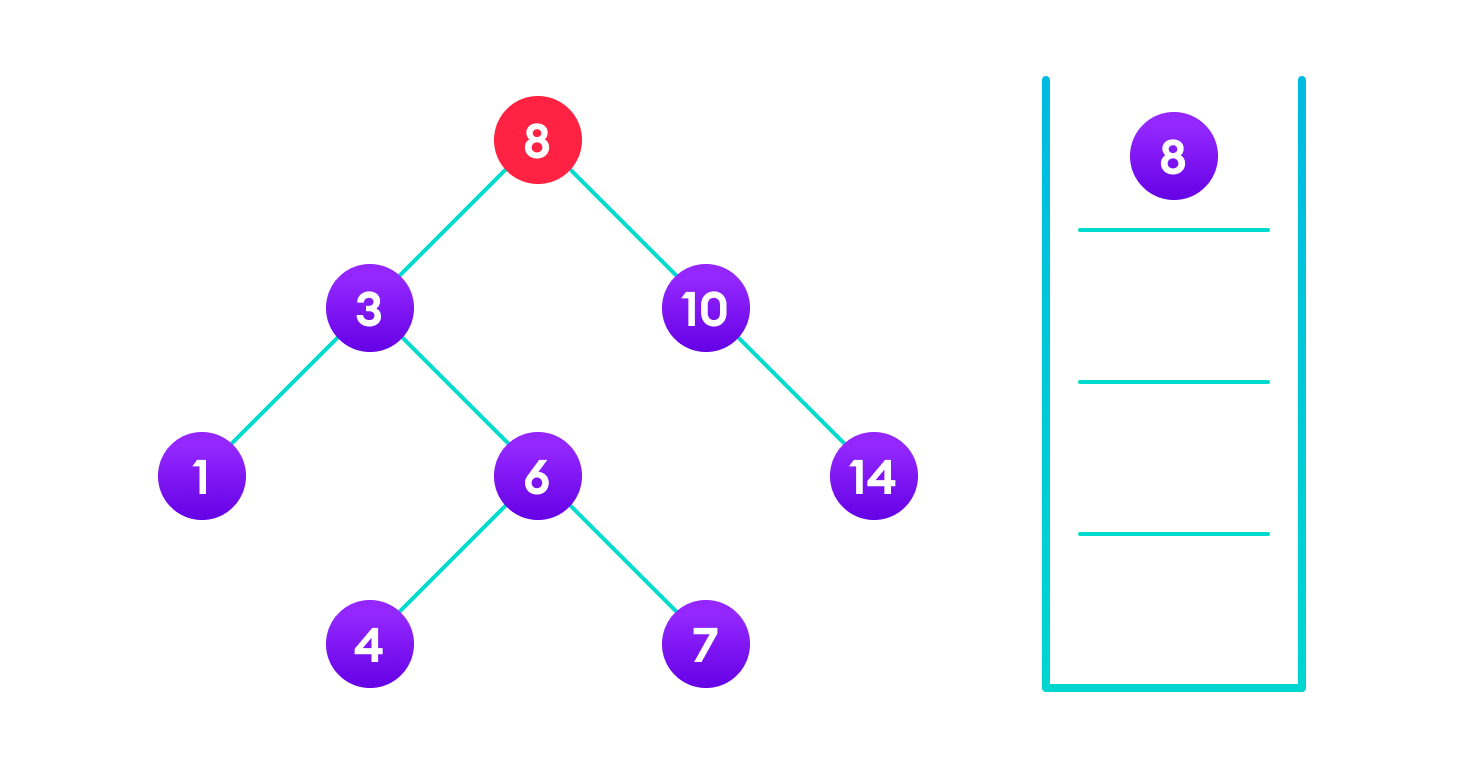
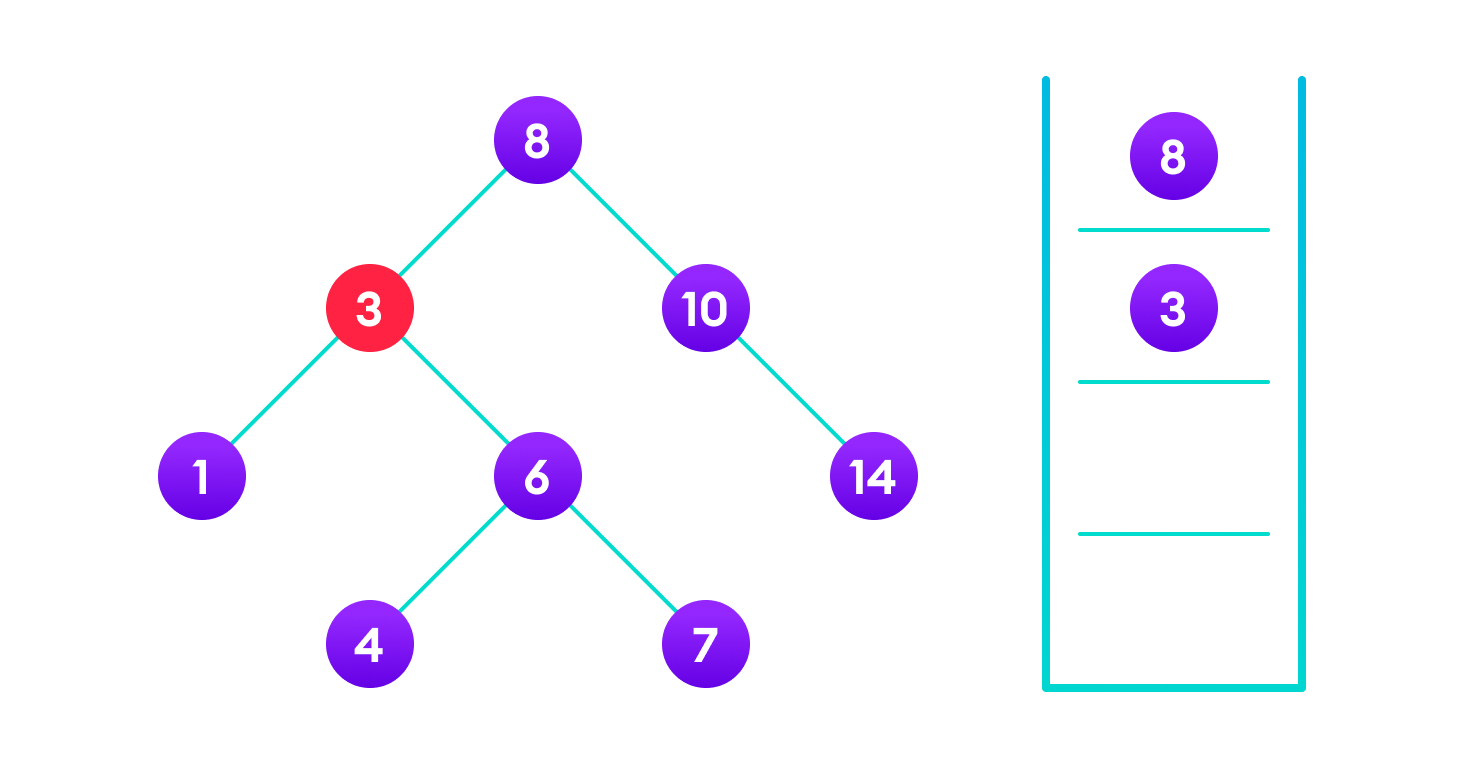
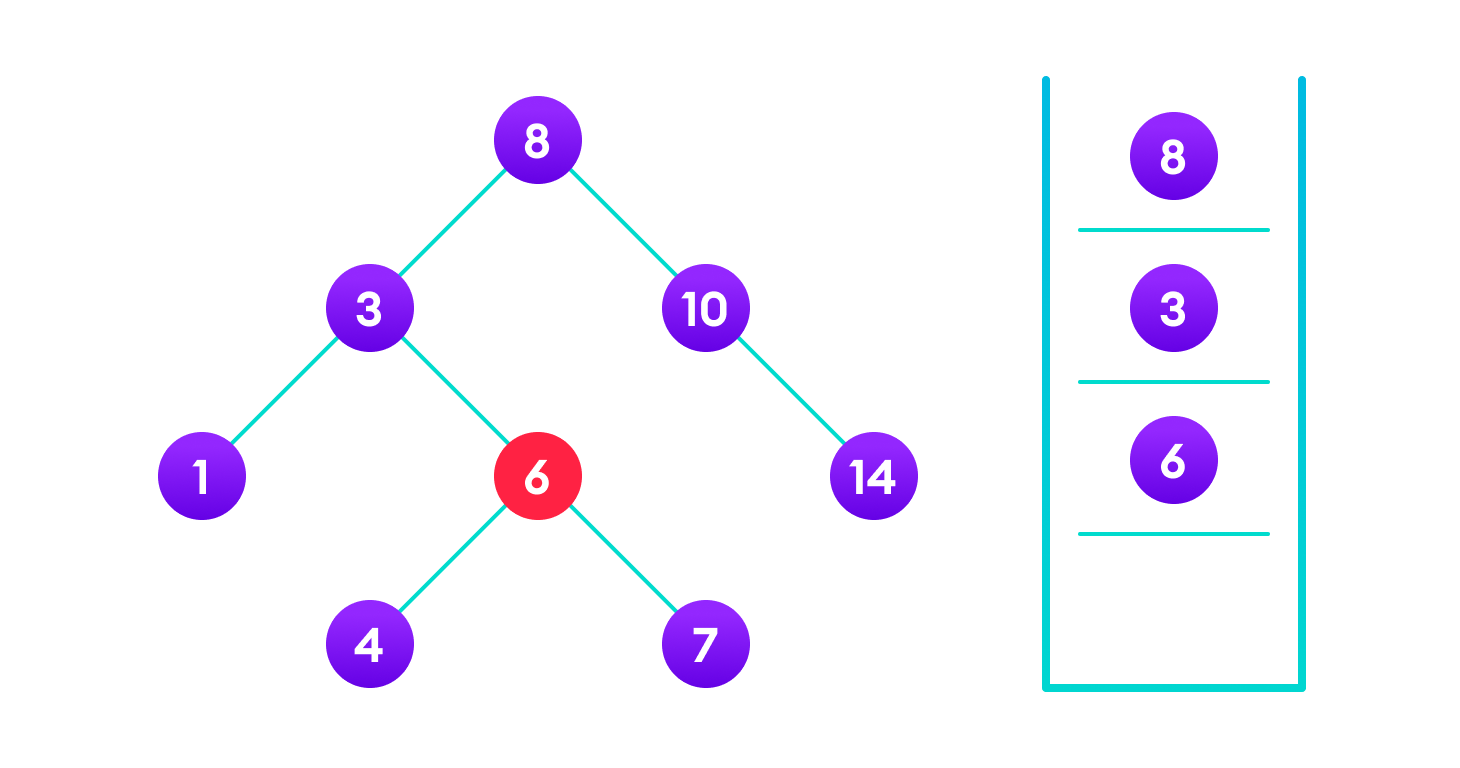
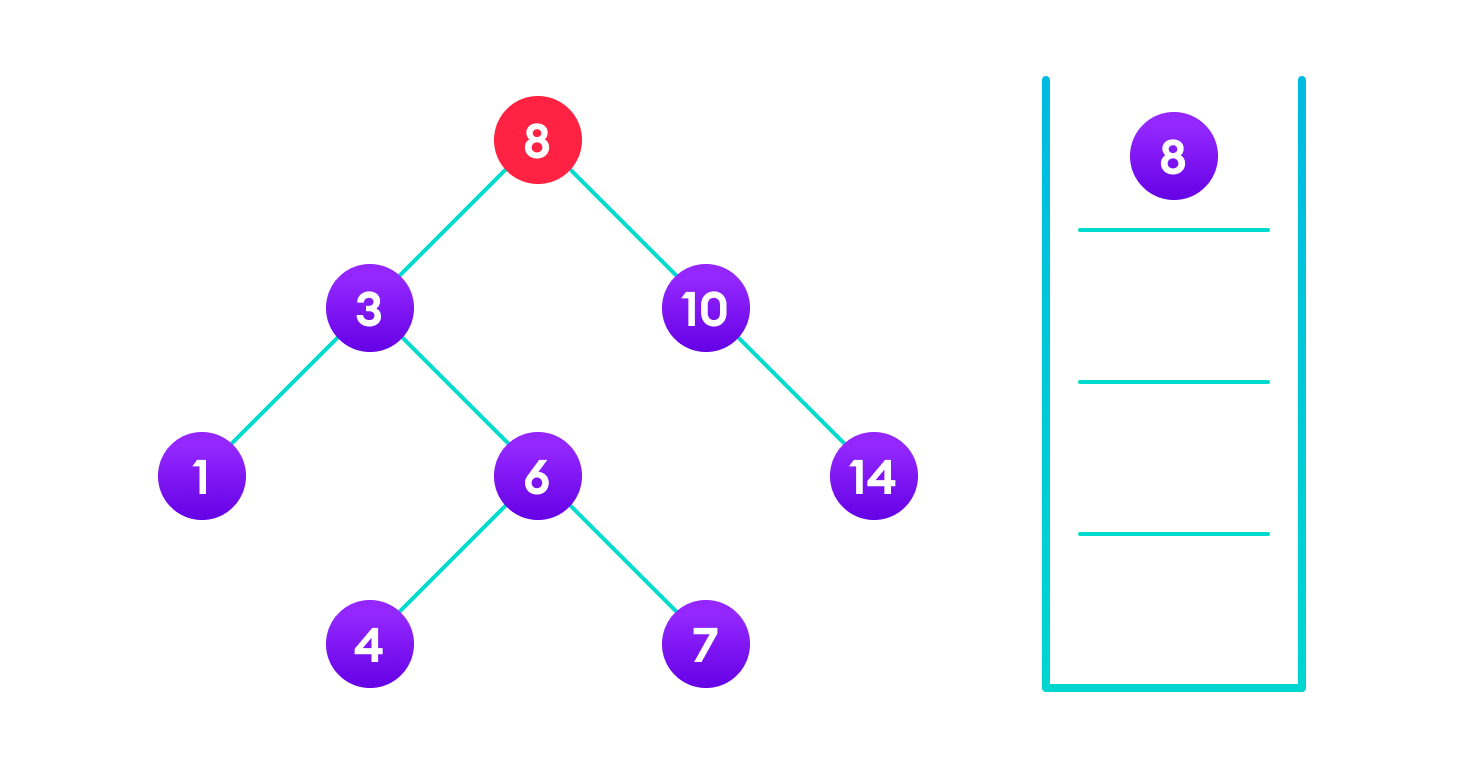
If number < root->data

return search(root->left)

If number > root->data

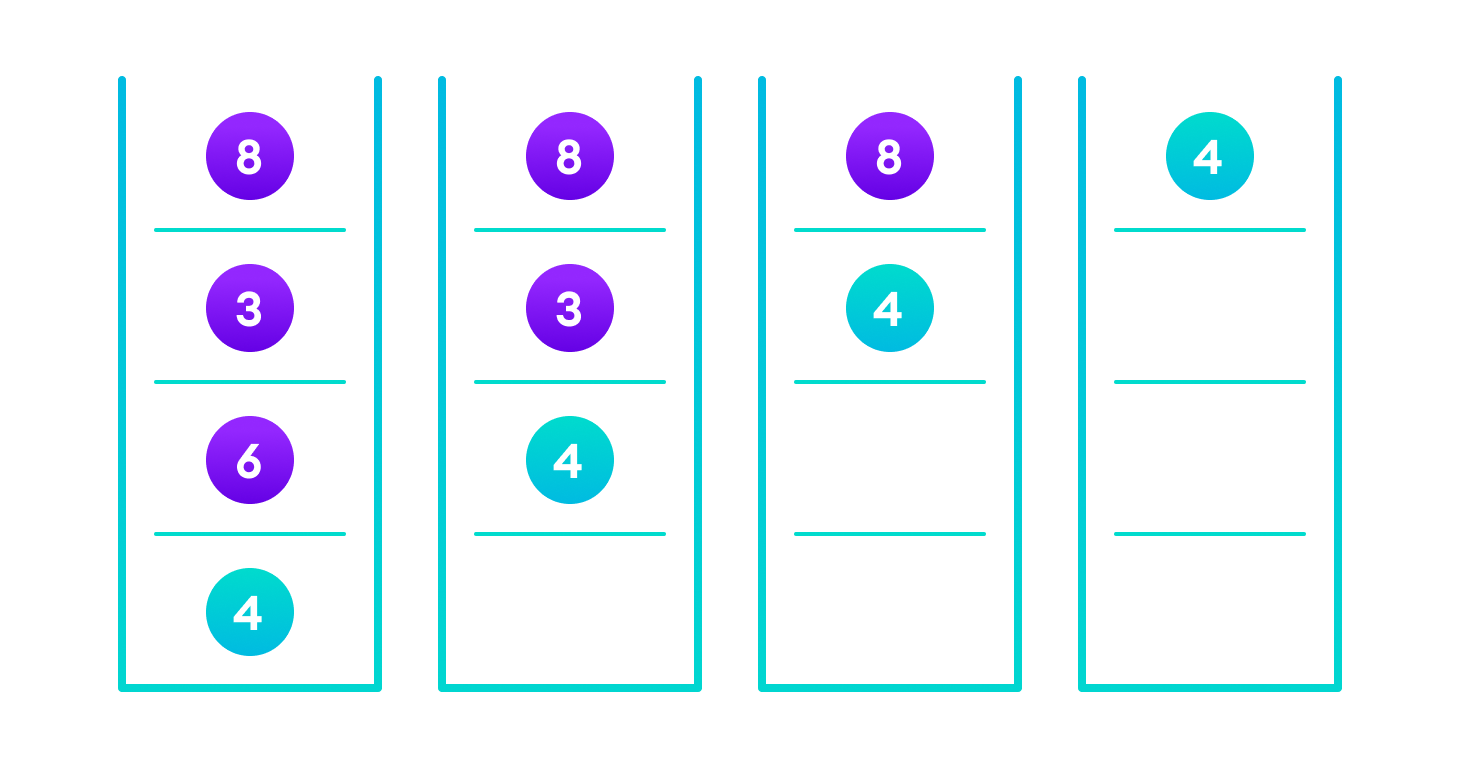
return search(root->right)

Let us try to visualize this with a diagram.

4 is not found so, traverse through the left subtree of 84 is not found so, traverse through the right subtree of 34 is not found so, traverse through the left subtree of 64 is found

If the value is found, we return the value so that it gets propagated in each recursion step as shown in the image below.

If you might have noticed, we have called return search(struct node\*) four times. When we return either the new node or NULL, the value gets returned again and again until search(root) returns the final result.

If the value is found in any of the subtrees, it is propagated up so that in the end it is returned, otherwise null is returned

If the value is not found, we eventually reach the left or right child of a leaf node which is NULL and it gets propagated and returned.

**Insert Operation**

Inserting a value in the correct position is similar to searching because we try to maintain the rule that the left subtree is lesser than root and the right subtree is larger than root.

We keep going to either right subtree or left subtree depending on the value and when we reach a point left or right subtree is null, we put the new node there.

**Algorithm:**

If node == NULL

return createNode(data)

if (data < node->data)

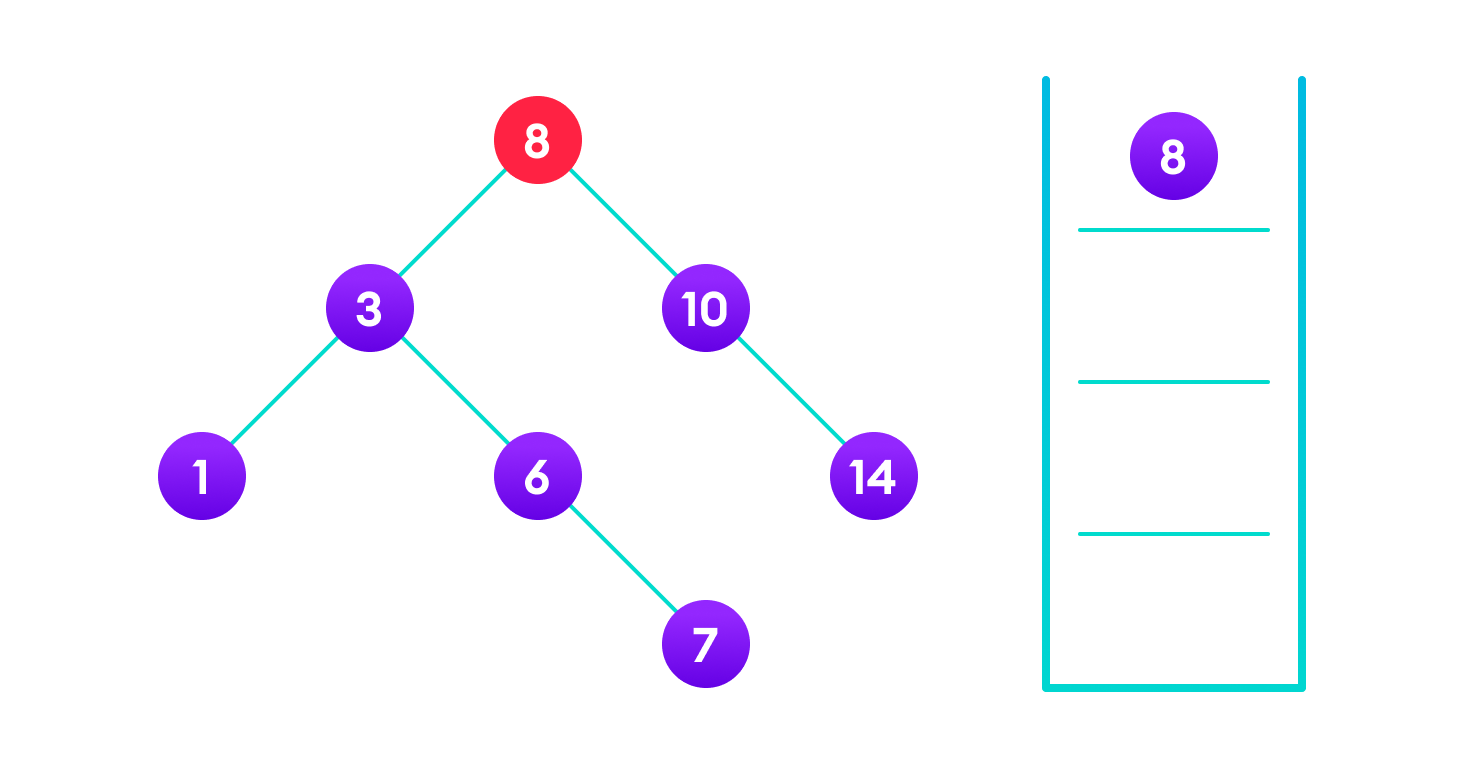
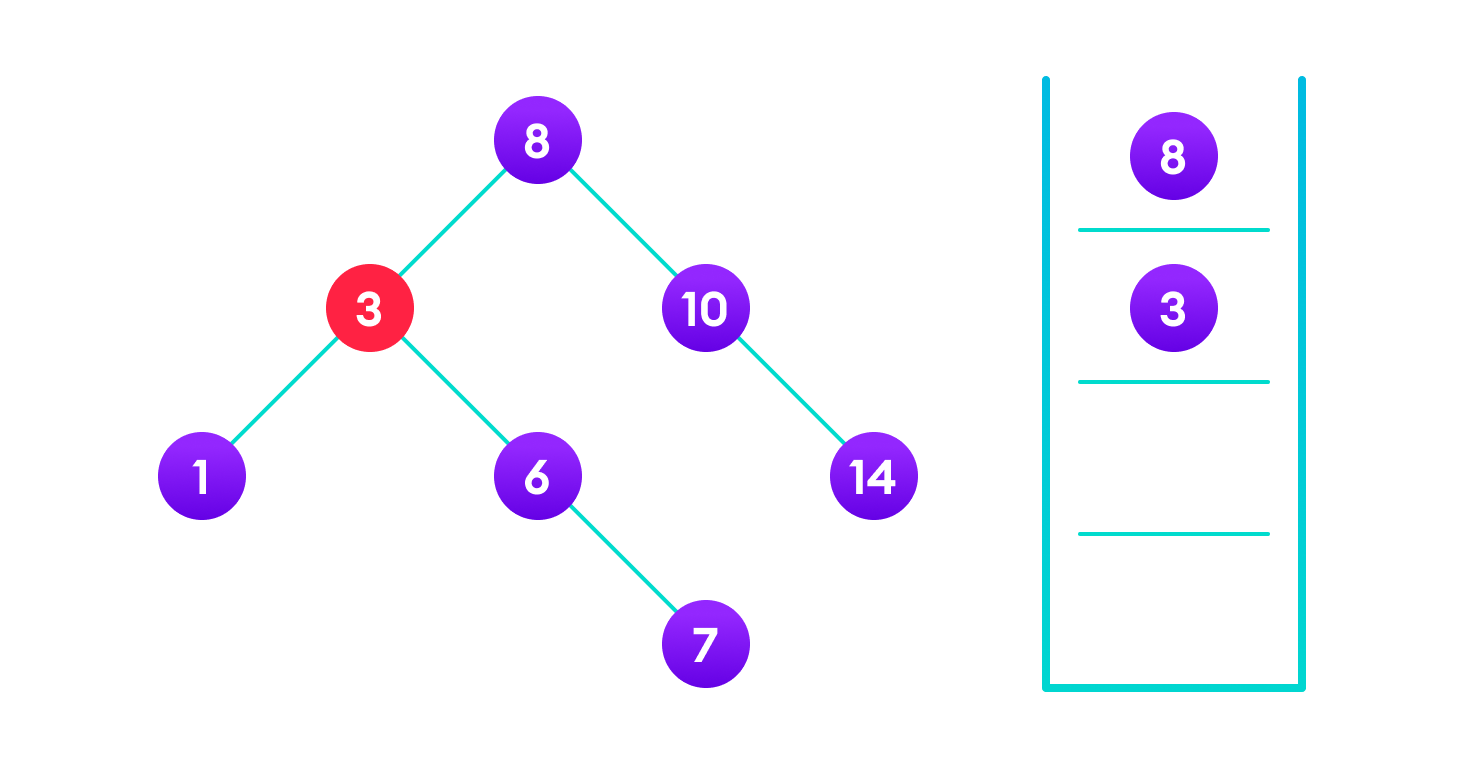
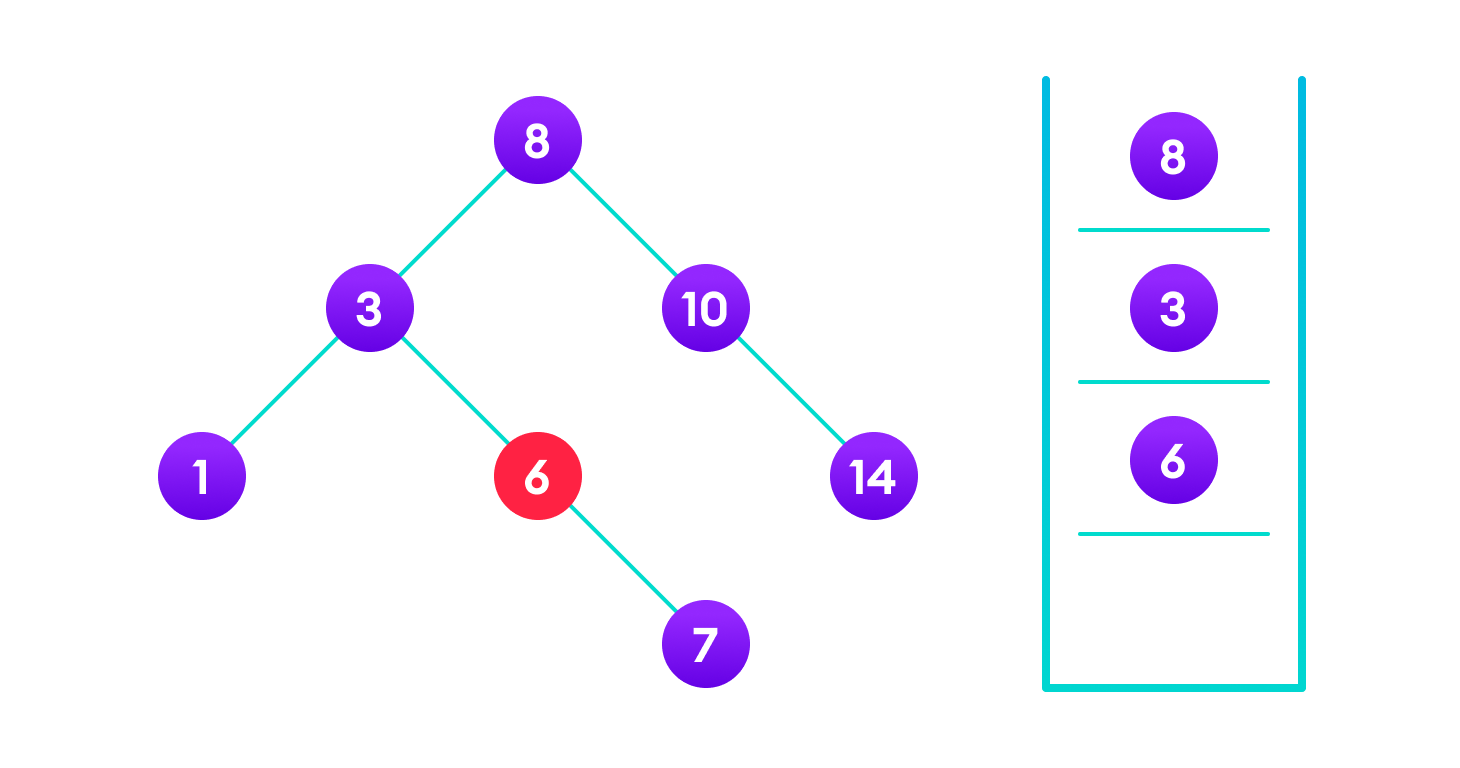
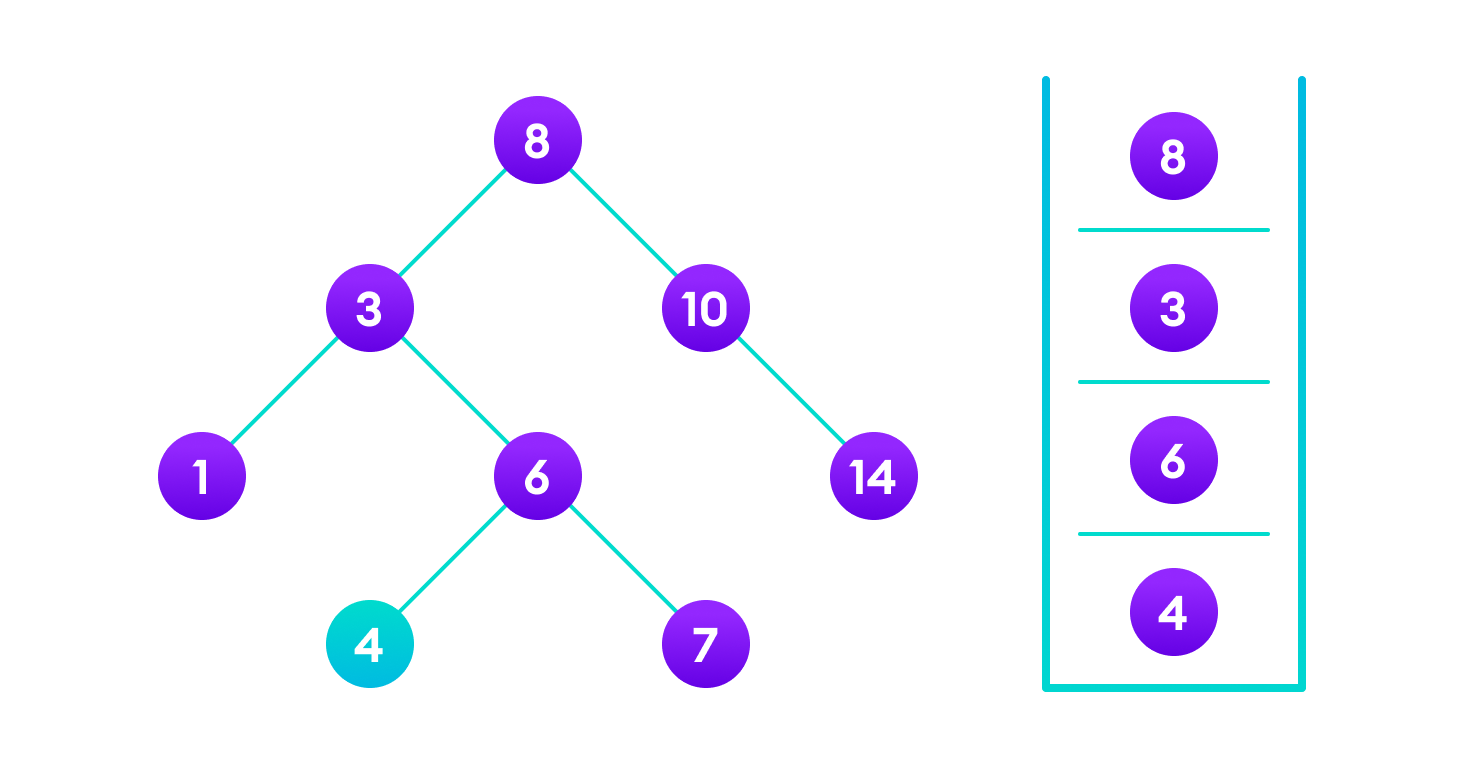
node->left = insert(node->left, data);

else if (data > node->data)

node->right = insert(node->right, data);

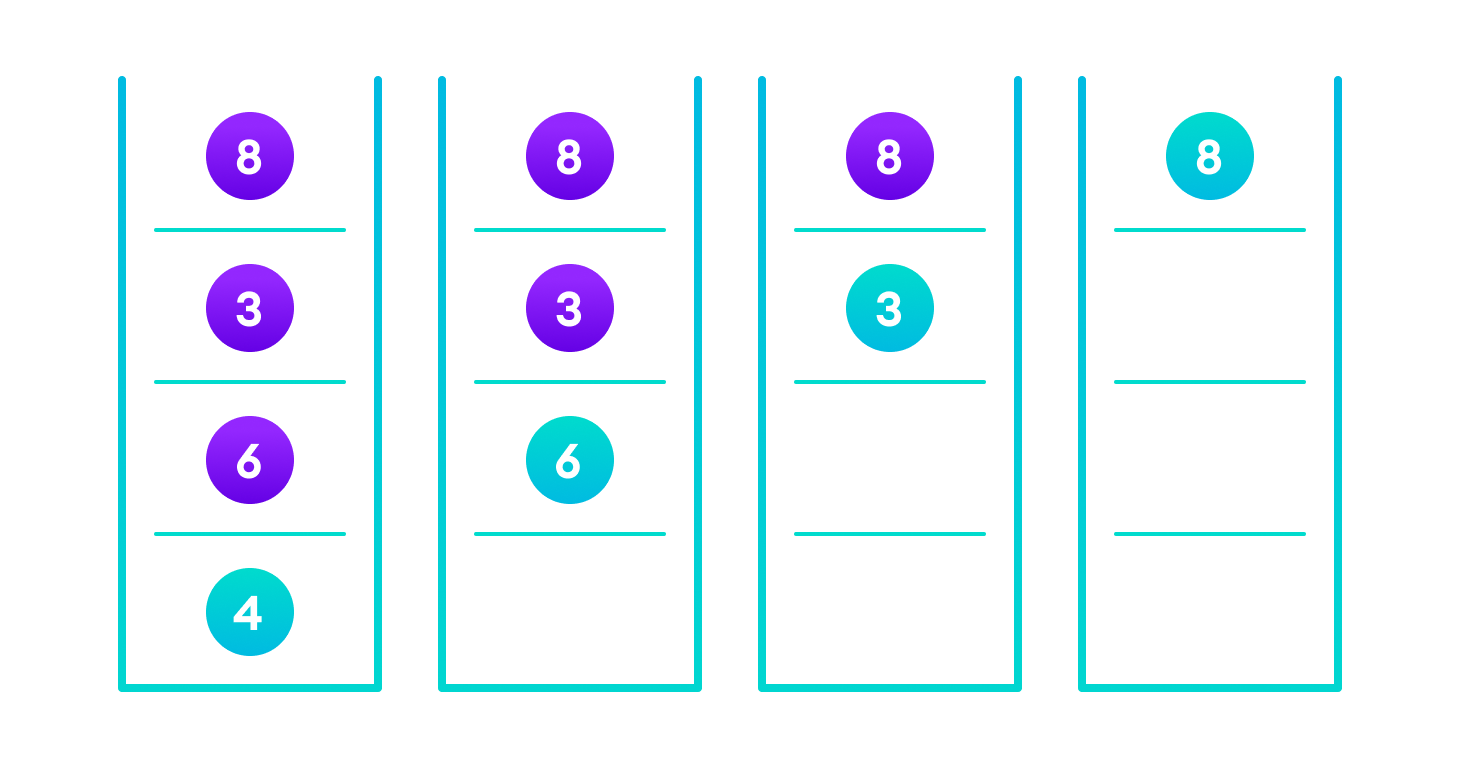
return node;

The algorithm isn't as simple as it looks. Let's try to visualize how we add a number to an existing BST.

4<8 so, transverse through the left child of 84>3 so, transverse through the right child of 84<6 so, transverse through the left child of 6Insert 4 as a left child of 6

We have attached the node but we still have to exit from the function without doing any damage to the rest of the tree. This is where the return node; at the end comes in handy. In the case of NULL, the newly created node is returned and attached to the parent node, otherwise the same node is returned without any change as we go up until we return to the root.

This makes sure that as we move back up the tree, the other node connections aren't changed.

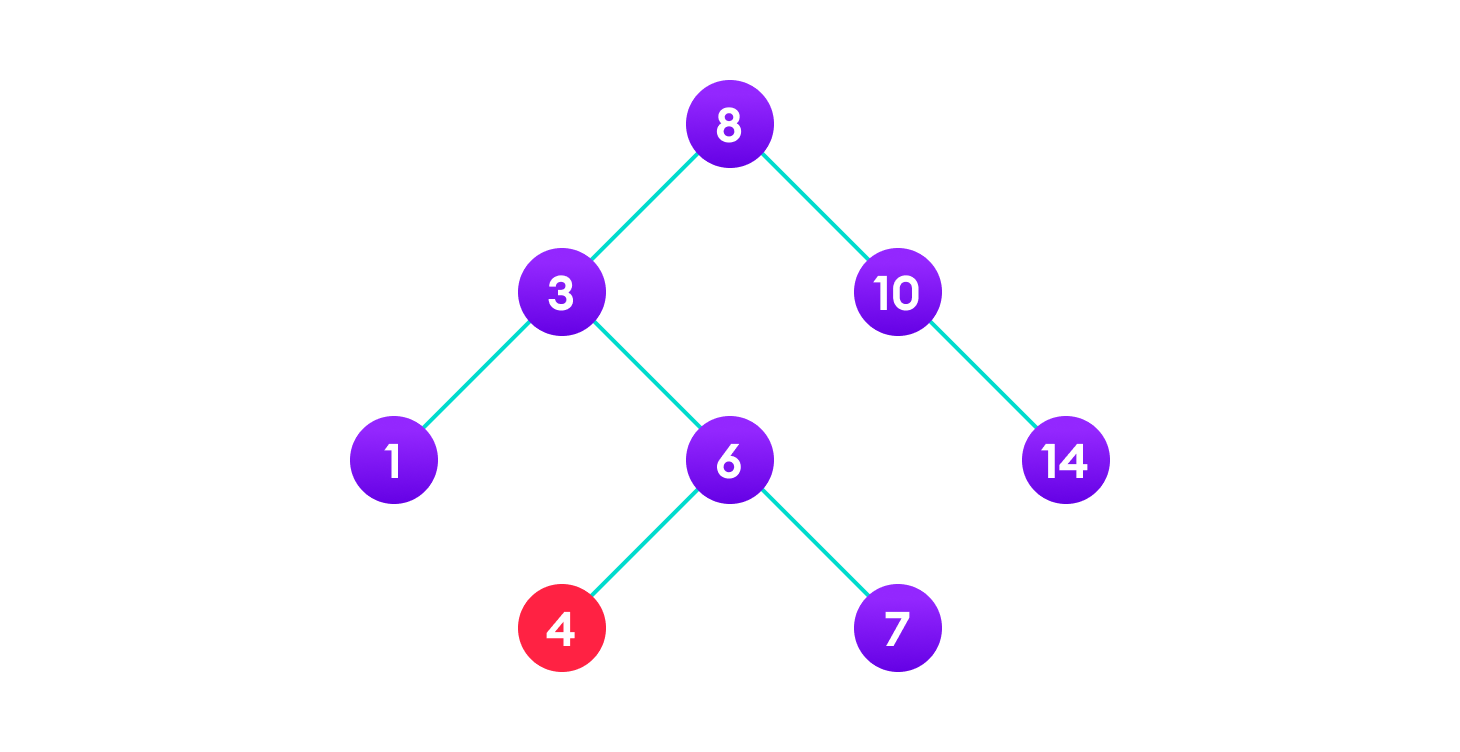
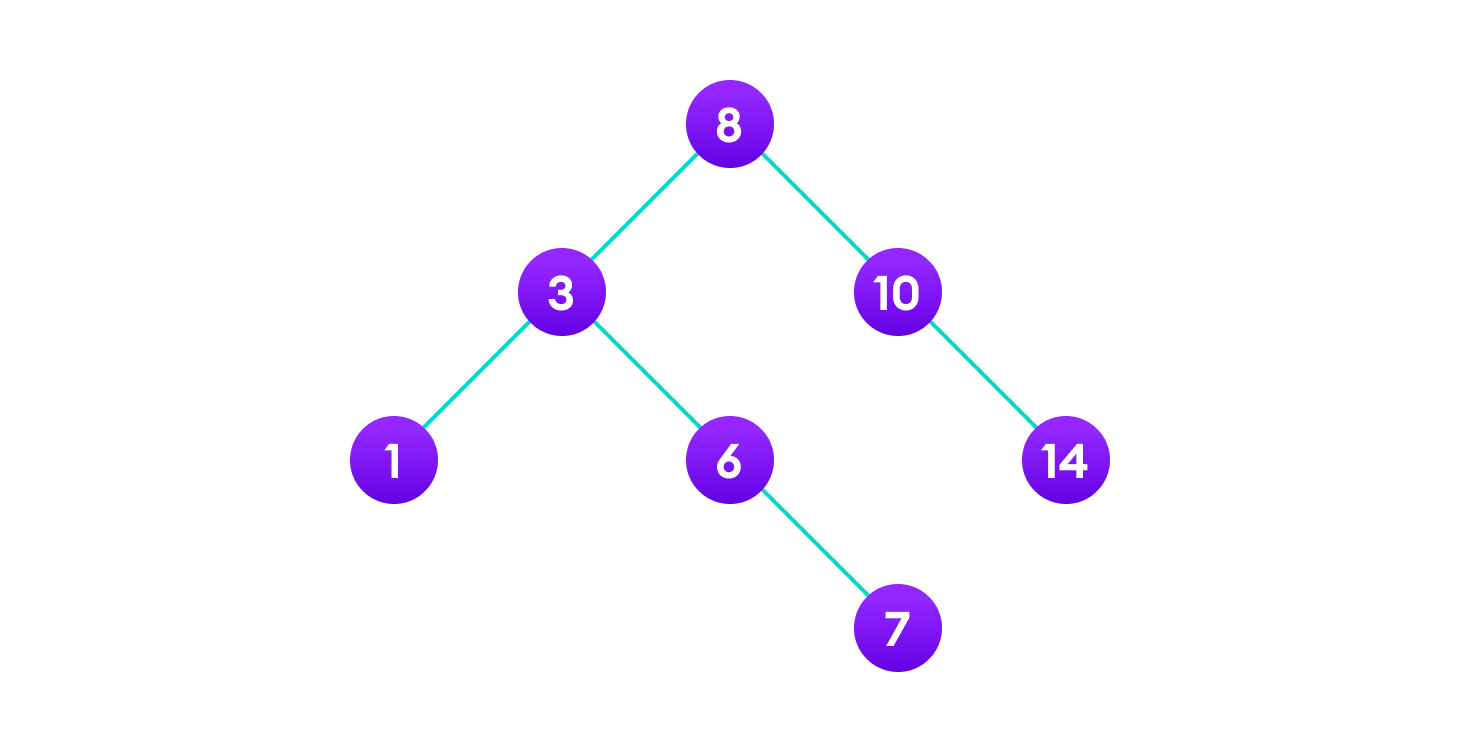
Image showing the importance of returning the root element at the end so that the elements don't lose their position during the upward recursion step.

**Deletion Operation**

There are three cases for deleting a node from a binary search tree.

**Case I**

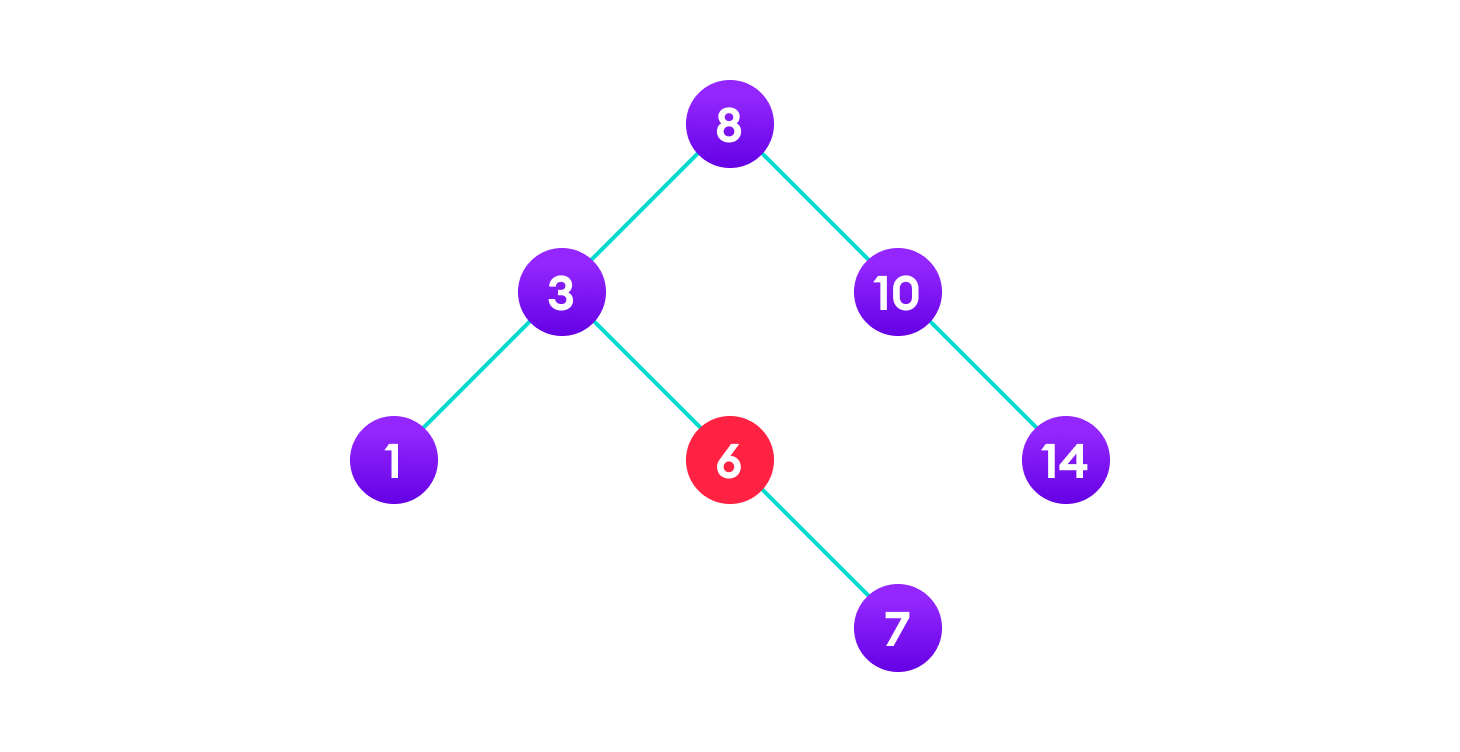
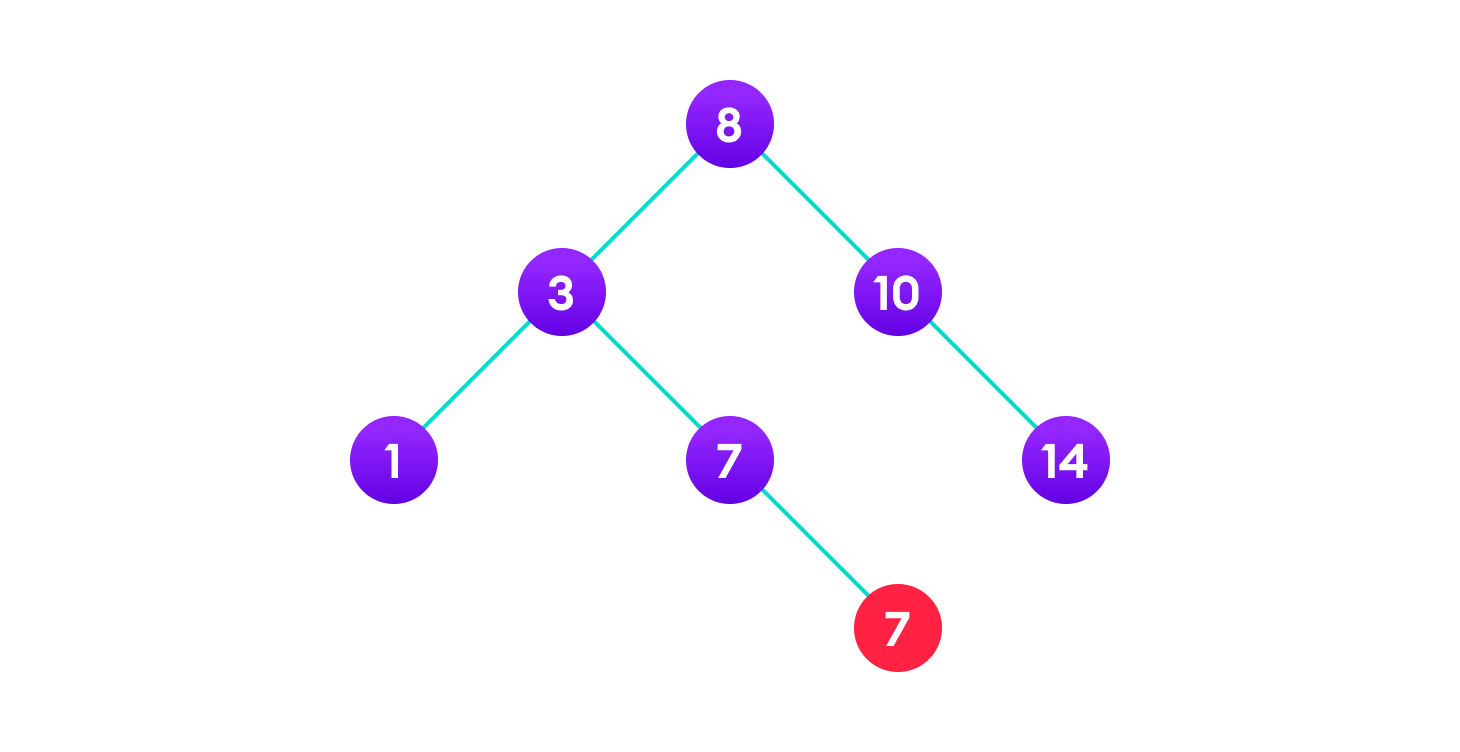
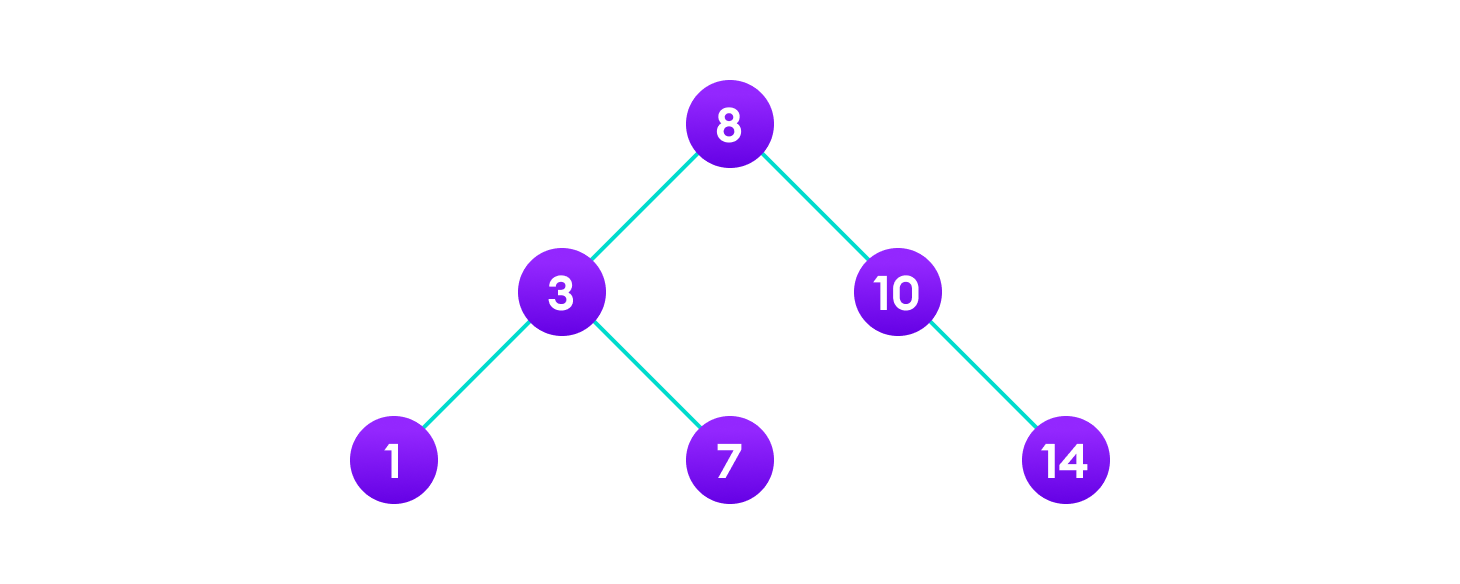
In the first case, the node to be deleted is the leaf node. In such a case, simply delete the node from the tree.

4 is to be deletedDelete the node

**Case II**

In the second case, the node to be deleted lies has a single child node. In such a case follow the steps below:

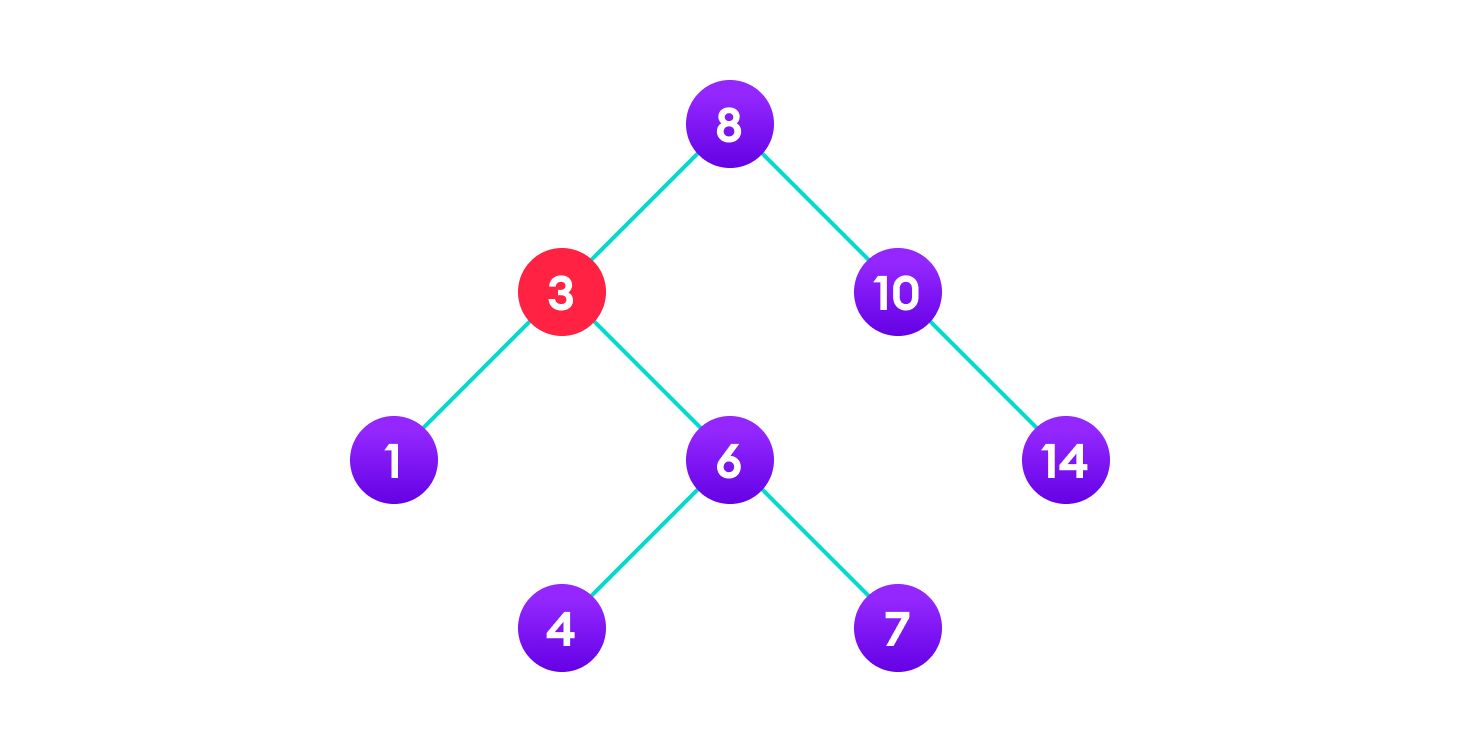
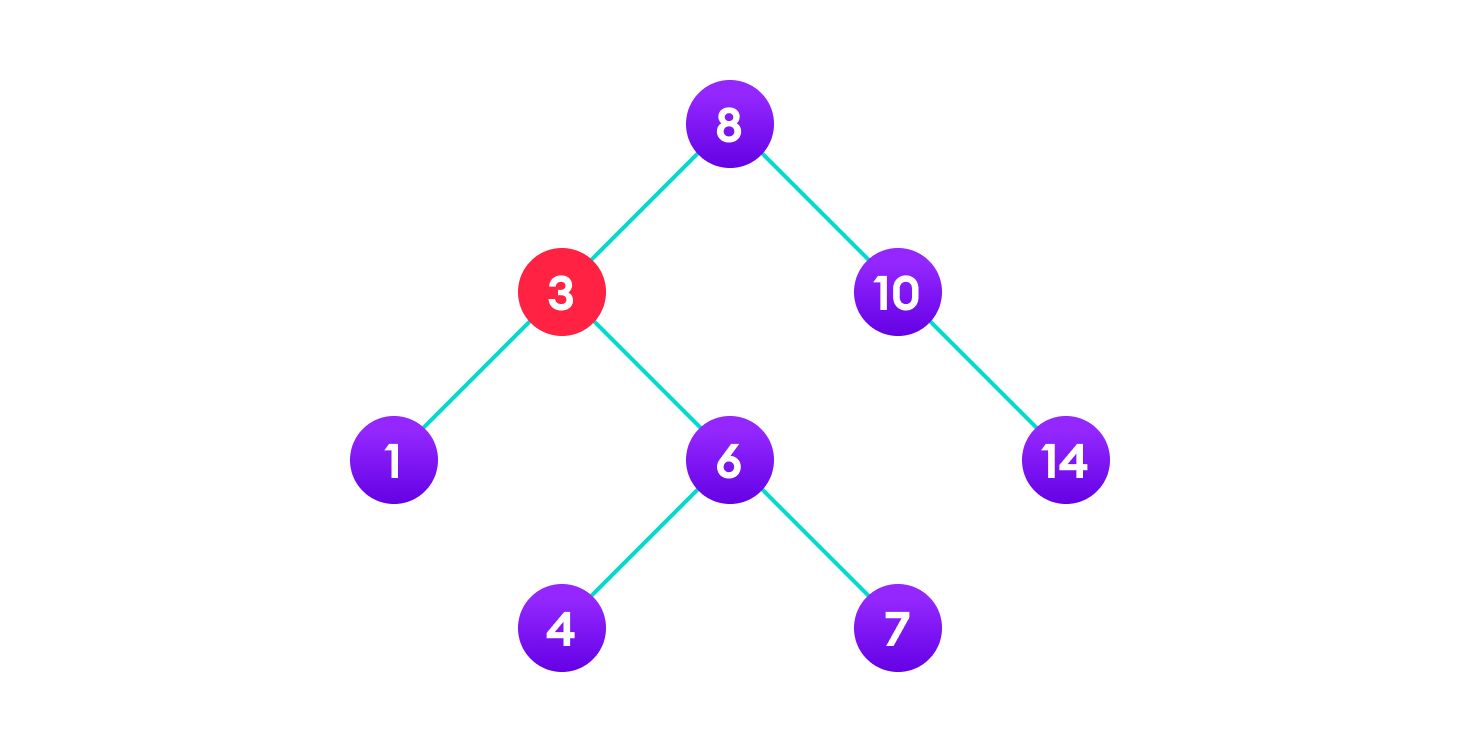
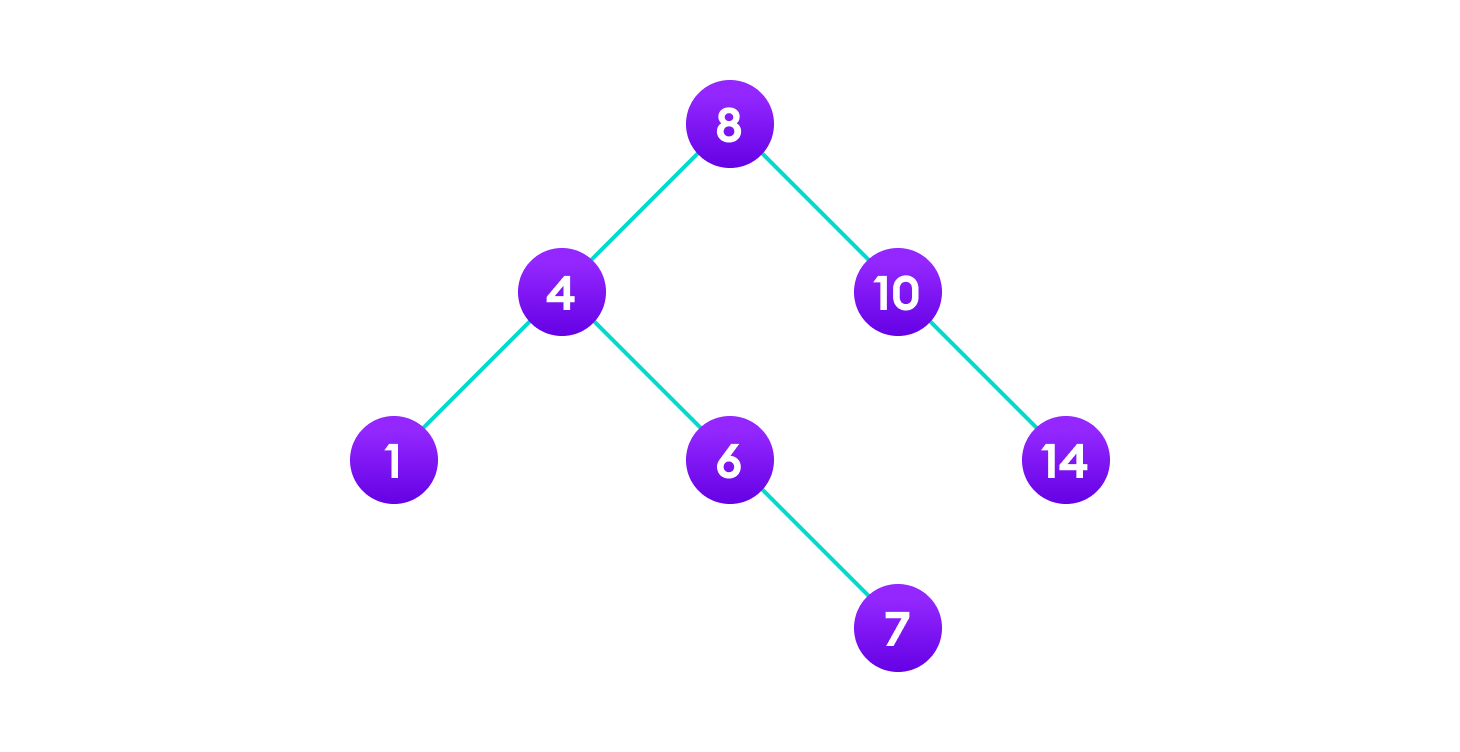
1. Replace that node with its child node.
2. Remove the child node from its original position.

6 is to be deletedcopy the value of its child to the node and delete the childFinal tree

**Case III**

In the third case, the node to be deleted has two children. In such a case follow the steps below:

1. Get the inorder successor of that node.
2. Replace the node with the inorder successor.
3. Remove the inorder successor from its original position.

3 is to be deletedCopy the value of the inorder successor (4) to the node

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.

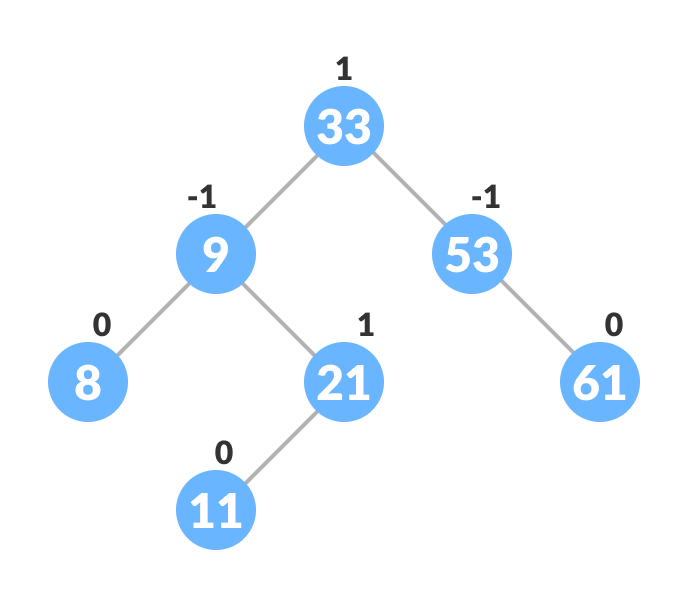
**Balance Factor**

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.

An example of a balanced avl tree is:

Avl tree

**Operations on an AVL tree**

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

**Rotating the subtrees in an AVL Tree**

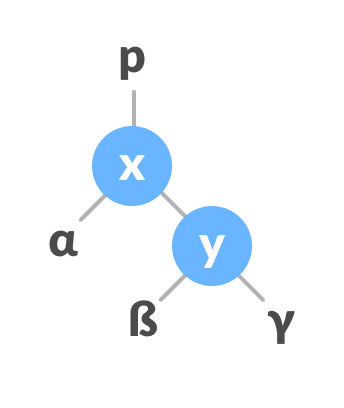
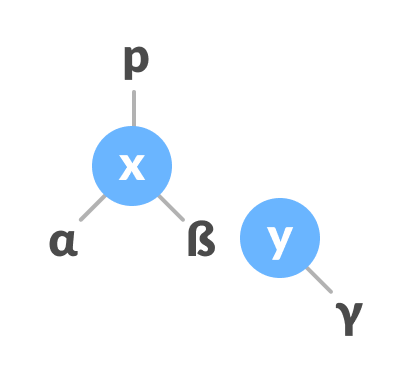
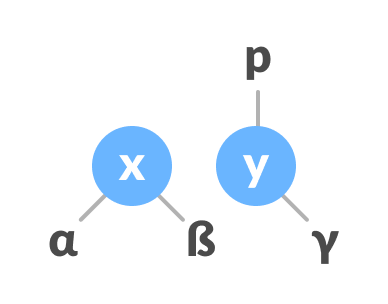
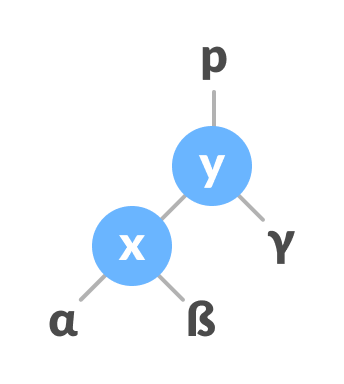
In rotation operation, the positions of the nodes of a subtree are interchanged.

There are two types of rotations:

**Left Rotate**

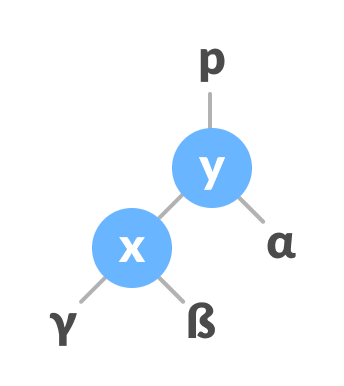
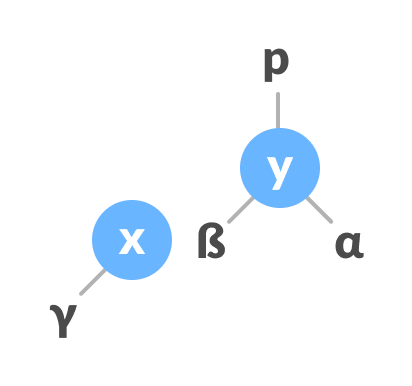
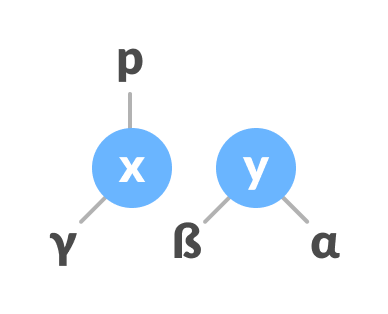
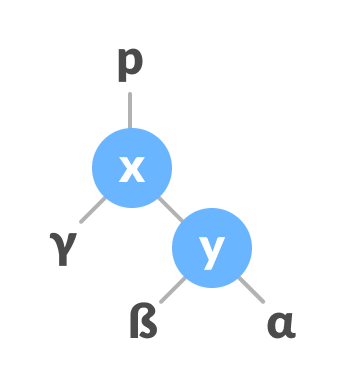
In left-rotation, the arrangement of the nodes on the right is transformed into the arrangements on the left node.

Algorithm

1. Let the initial tree be:Left rotate
2. If y has a left subtree, assign x as the parent of the left subtree of y.Assign x as the parent of the left subtree of y
3. If the parent of x is NULL, make y as the root of the tree.
4. Else if x is the left child of p, make y as the left child of p.
5. Else assign y as the right child of p.Change the parent of x to that of y
6. Make y as the parent of x.Assign y as the parent of x.

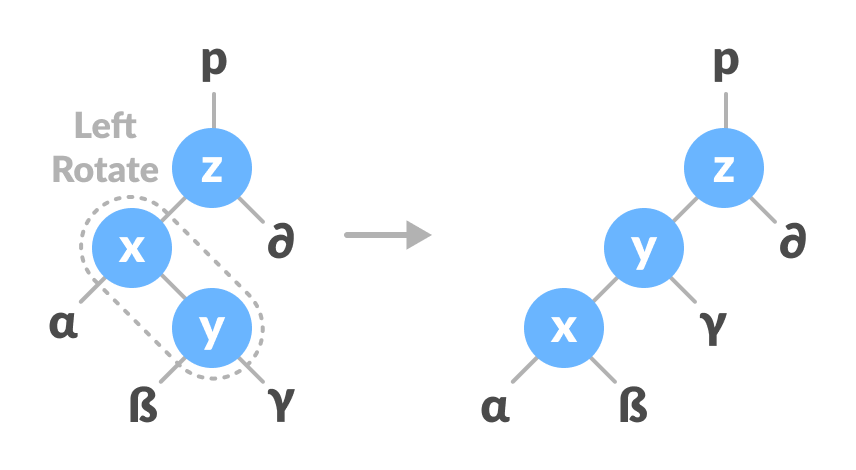
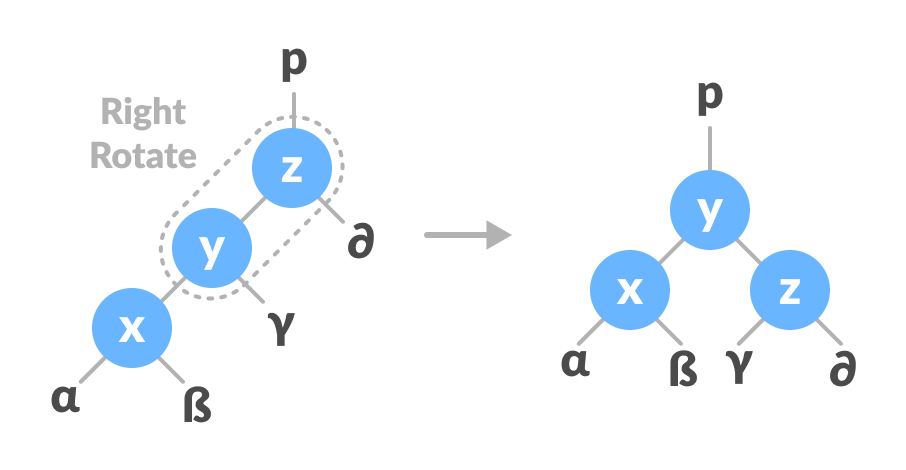
**Right Rotate**

In left-rotation, the arrangement of the nodes on the left is transformed into the arrangements on the right node.

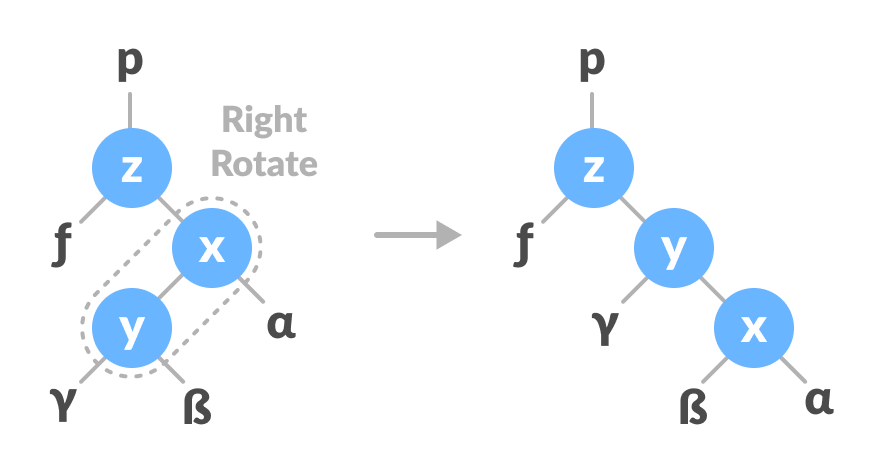
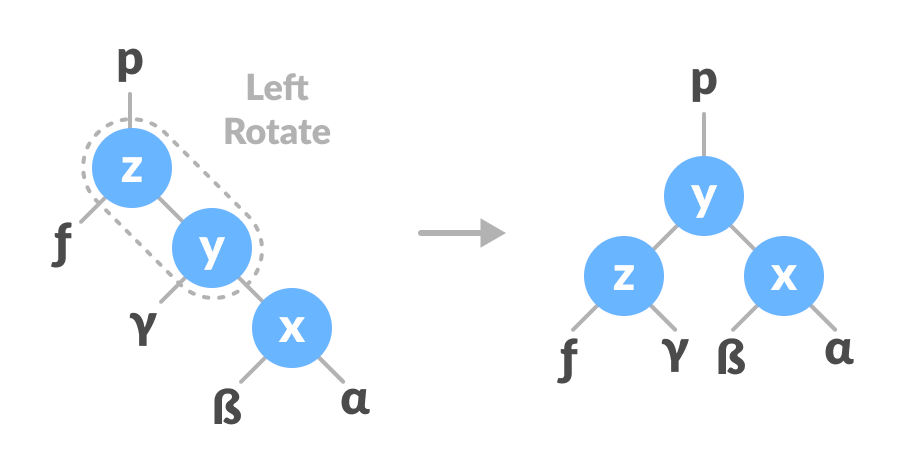
1. Let the initial tree be:Initial tree
2. If x has a right subtree, assign y as the parent of the right subtree of x.Assign y as the parent of the right subtree of x
3. If the parent of y is NULL, make x as the root of the tree.
4. Else if y is the right child of its parent p, make x as the right child of p.
5. Else assign x as the left child of p.Assign the parent of y as the parent of x.
6. Make x as the parent of y.Assign x as the parent of y

**Left-Right and Right-Left Rotate**

In left-right rotation, the arrangements are first shifted to the left and then to the right.

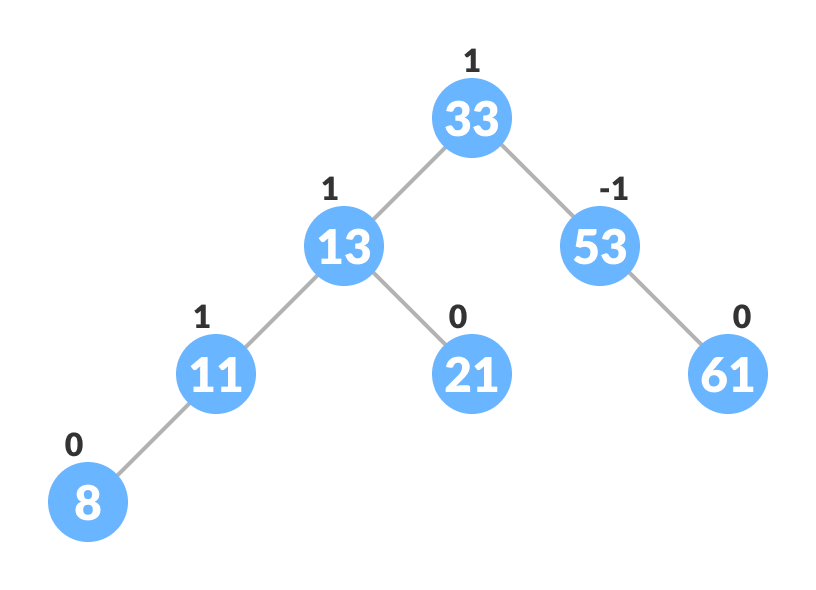
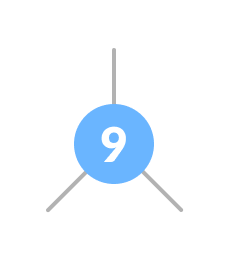
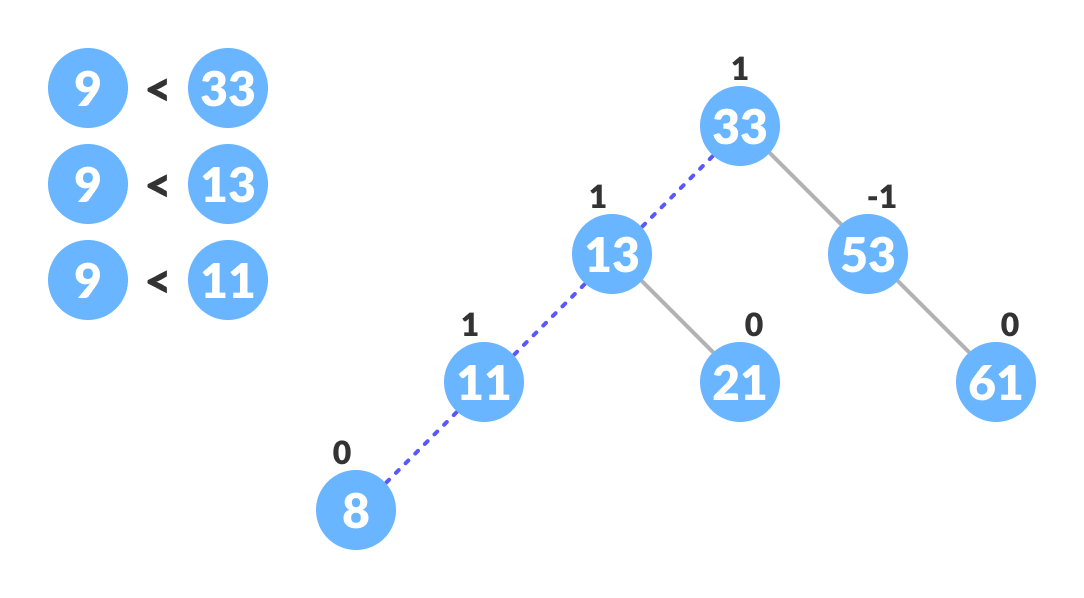
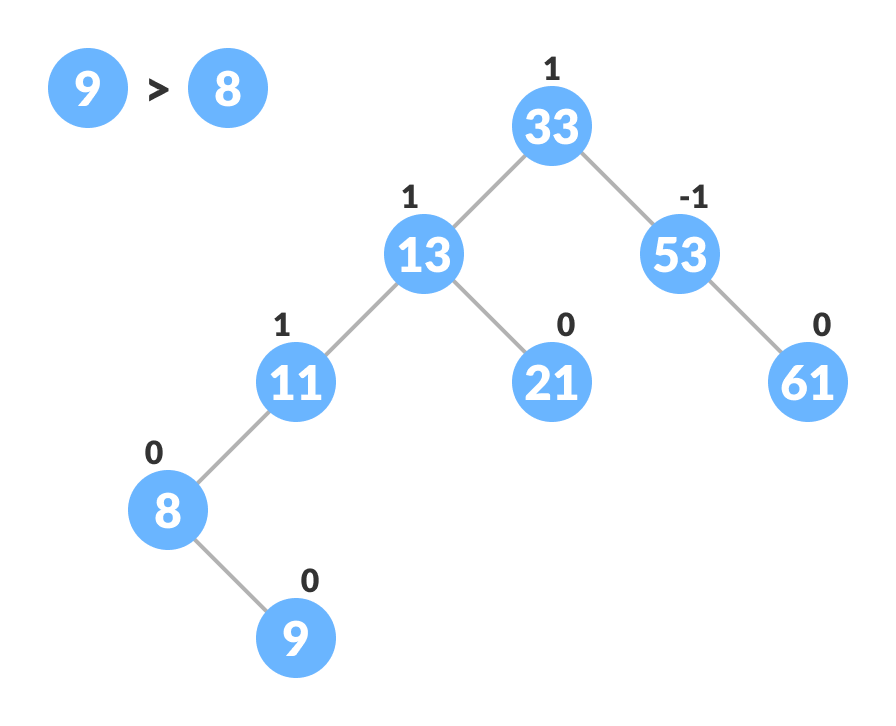
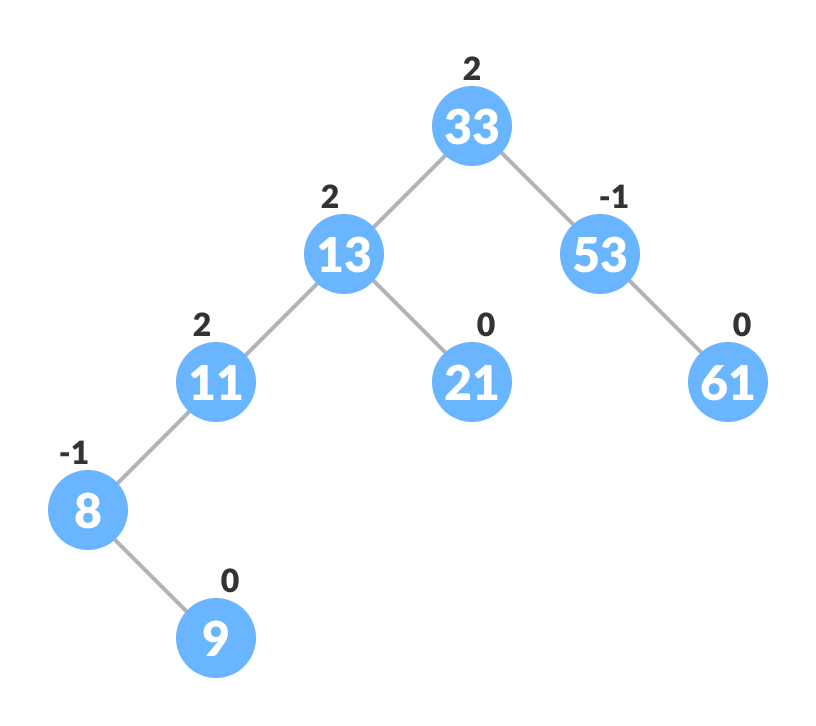
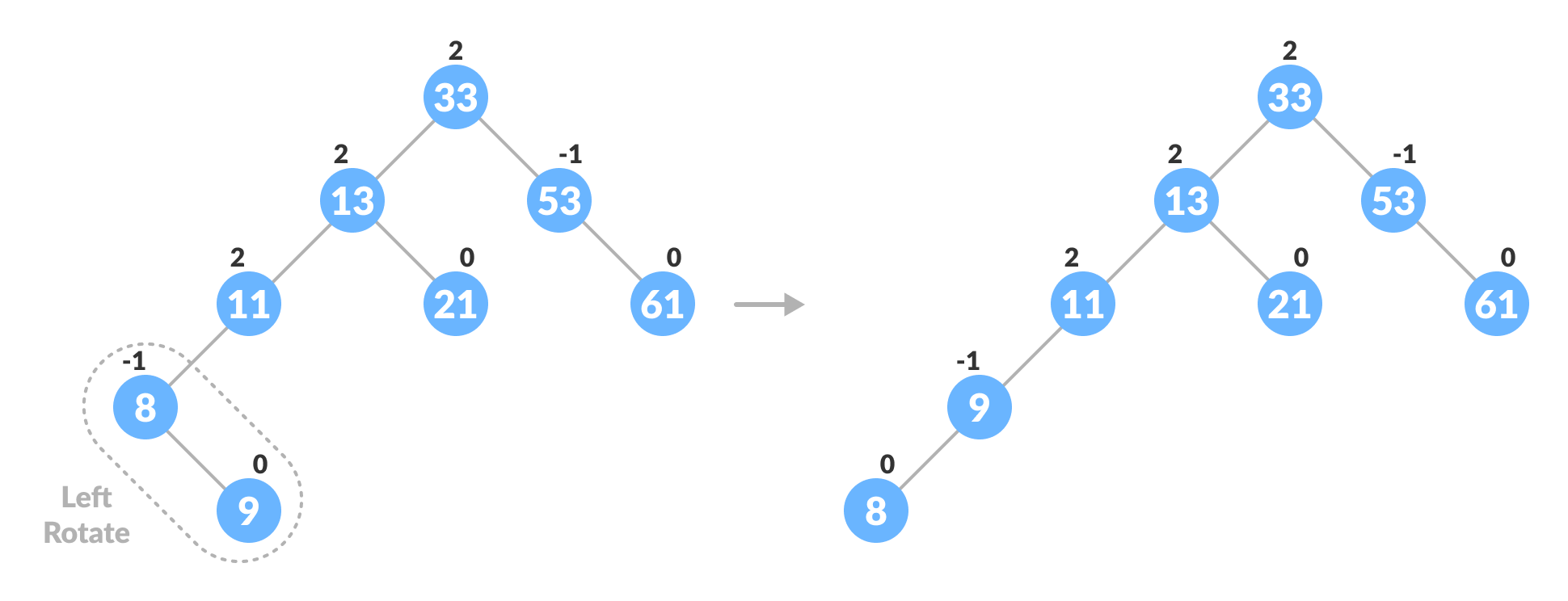
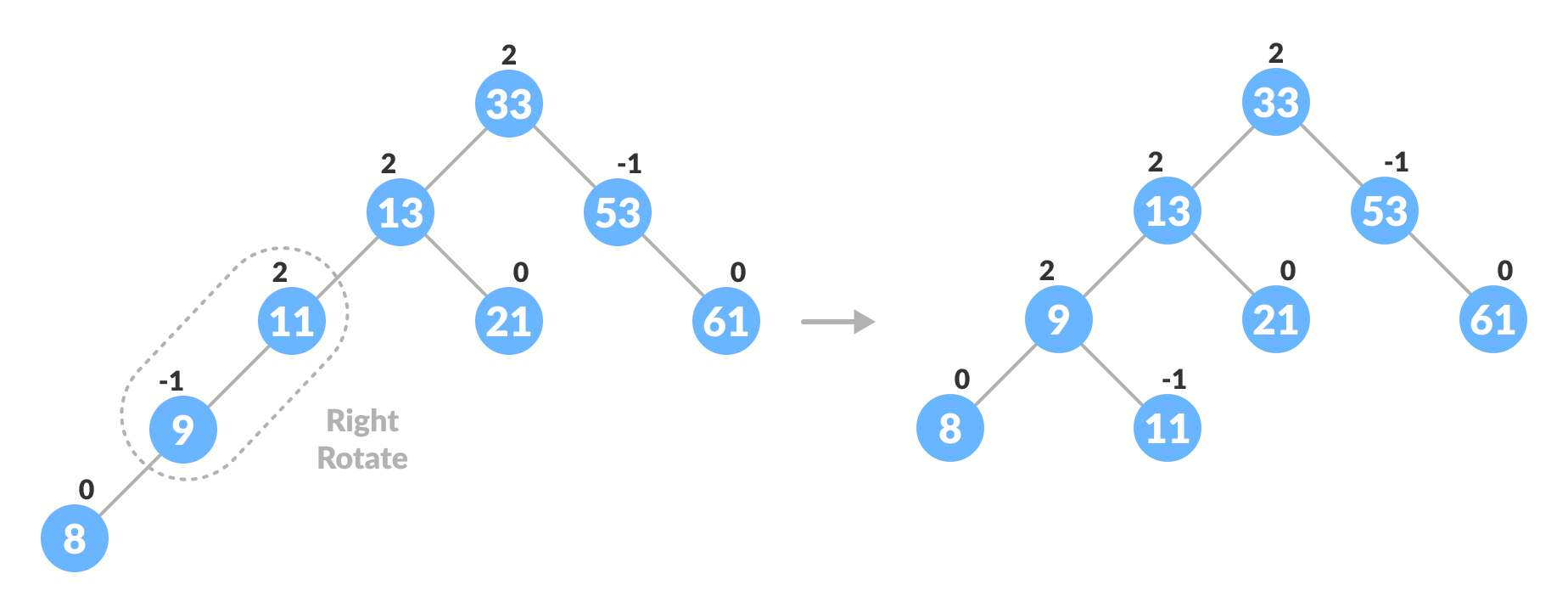
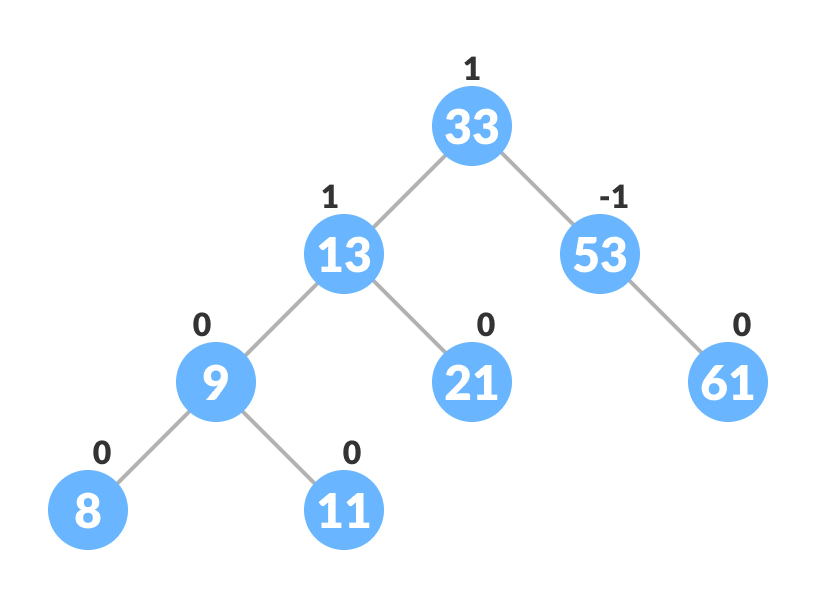
1. Do left rotation on x-y.Left rotate x-y
2. Do right rotation on y-z.Right rotate z-y

In right-left rotation, the arrangements are first shifted to the right and then to the left.

1. Do right rotation on x-y.Right rotate x-y
2. Do left rotation on z-y.Left rotate z-y

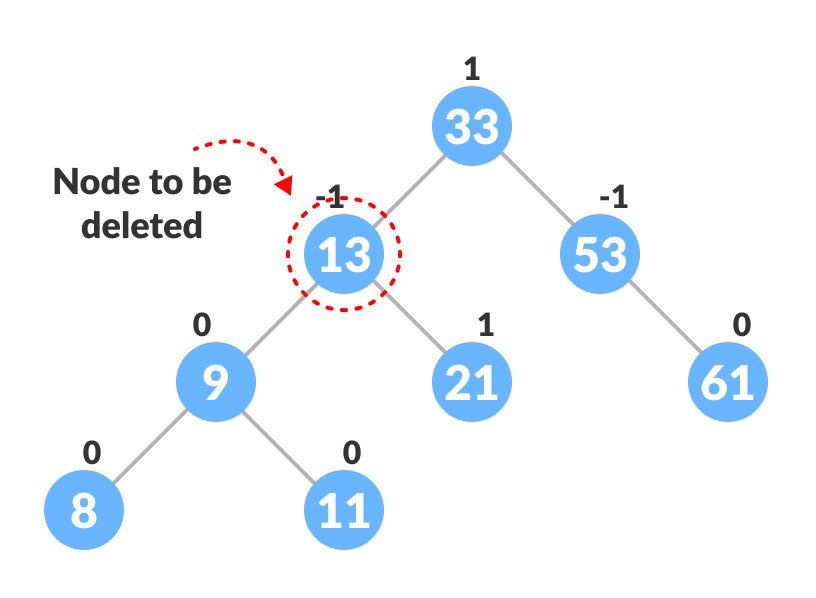
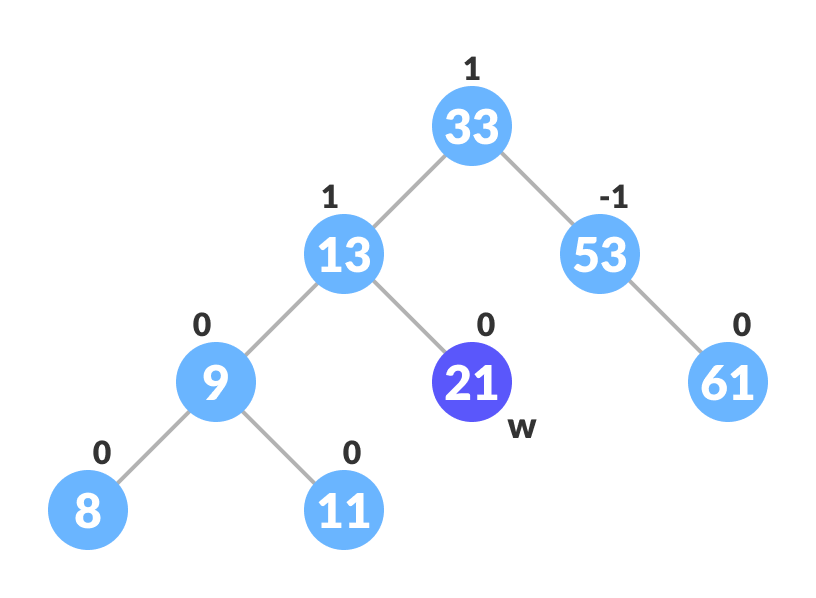
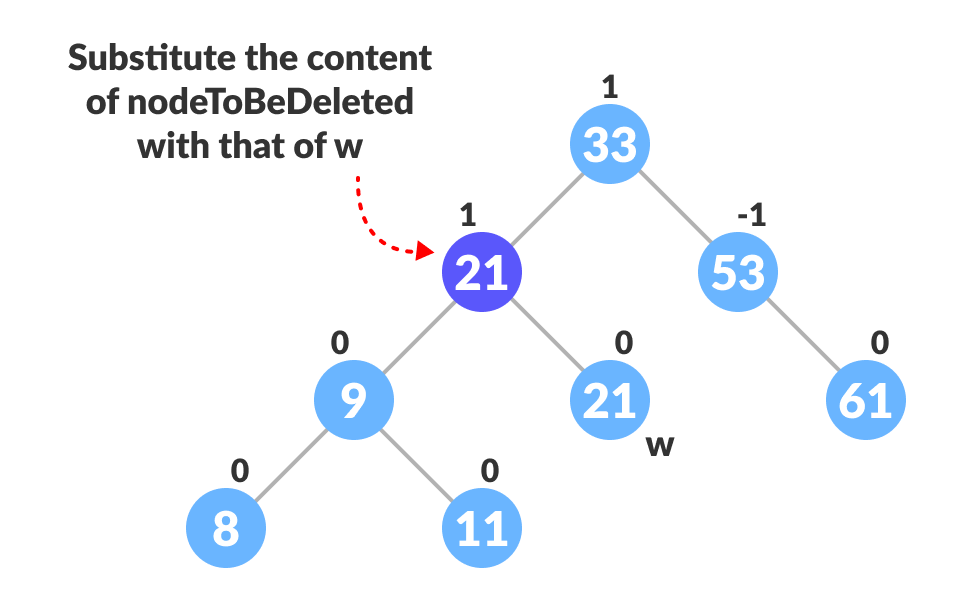
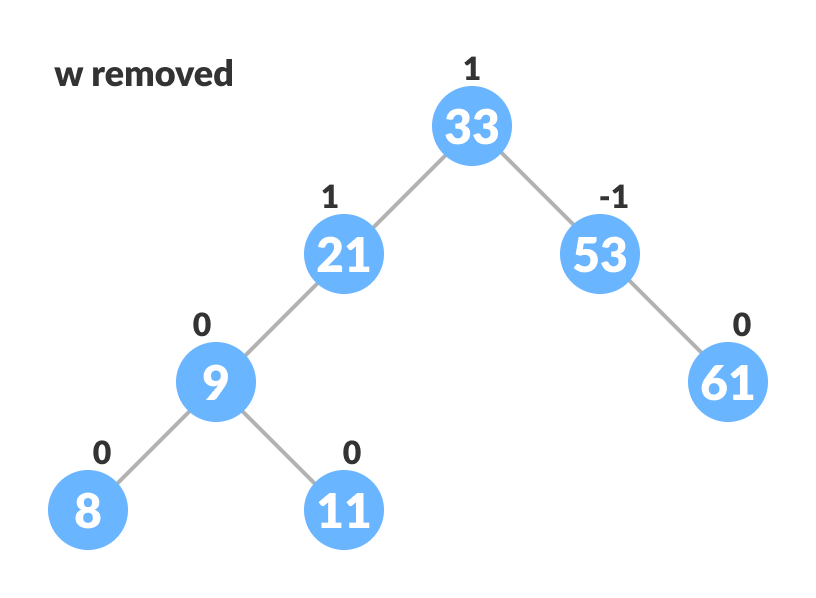
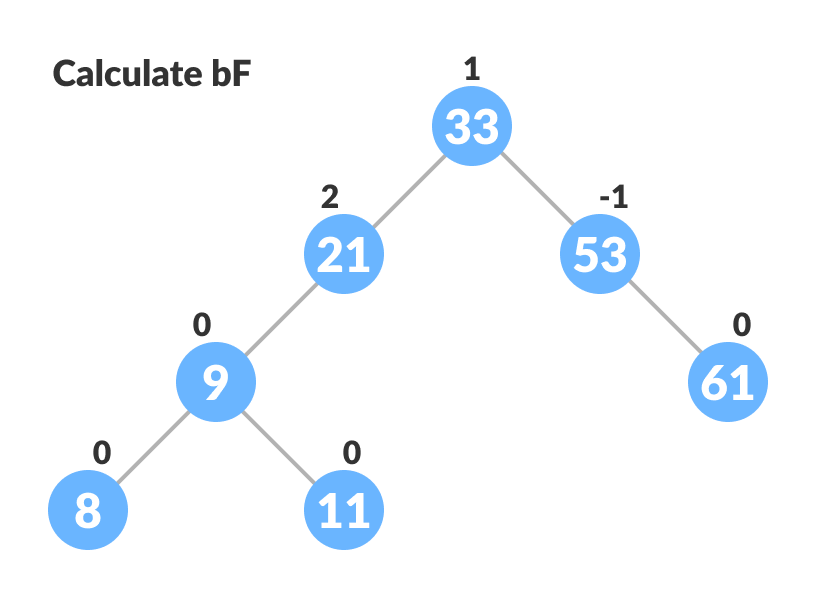
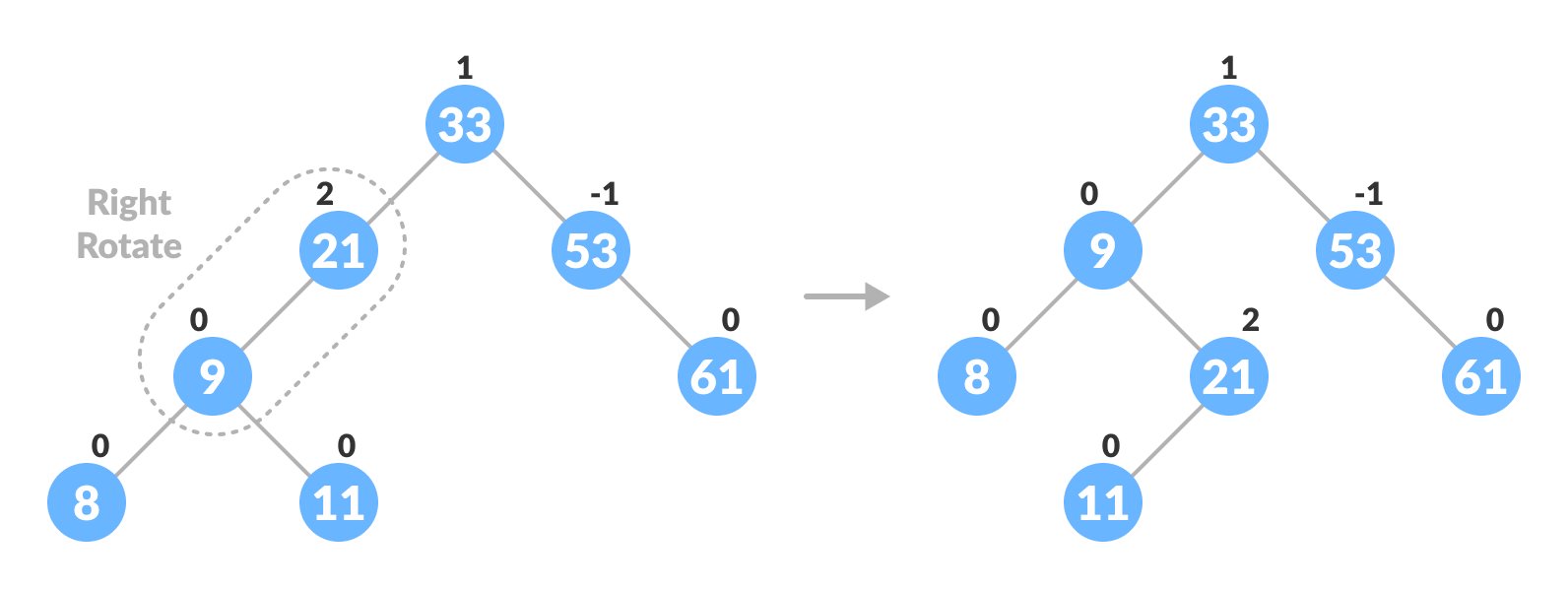
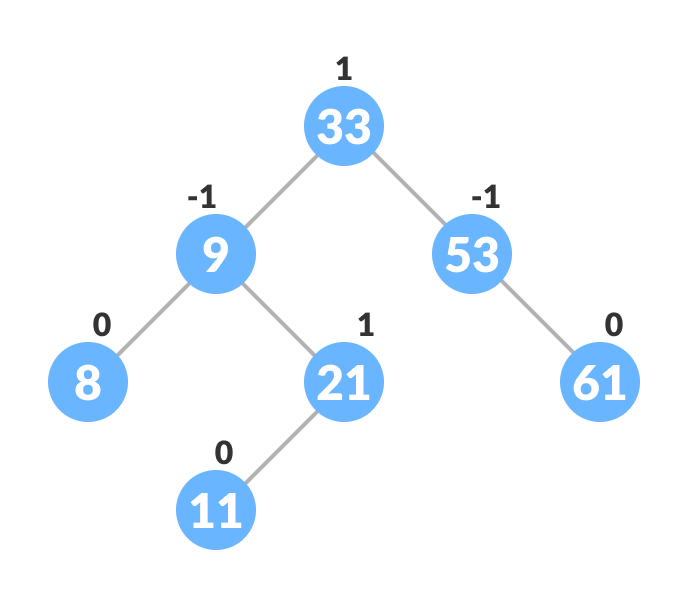
**Algorithm to insert a newNode**

A newNode is always inserted as a leaf node with balance factor equal to 0.

1. Let the initial tree be:Initial tree for insertion  
   Let the node to be inserted be:New node
2. Go to the appropriate leaf node to insert a newNode using the following recursive steps. Compare newKey with rootKey of the current tree.
   1. If newKey < rootKey, call insertion algorithm on the left subtree of the current node until the leaf node is reached.
   2. Else if newKey > rootKey, call insertion algorithm on the right subtree of current node until the leaf node is reached.
   3. Else, return leafNode.Finding the location to insert newNode
3. Compare leafKey obtained from the above steps with newKey:
   1. If newKey < leafKey, make newNode as the leftChild of leafNode.
   2. Else, make newNode as rightChild of leafNode.Inserting the new node
4. Update balanceFactor of the nodes.Updating the balance factor after insertion
5. If the nodes are unbalanced, then rebalance the node.
   1. If balanceFactor > 1, it means the height of the left subtree is greater than that of the right subtree. So, do a right rotation or left-right rotation
      1. If newNodeKey < leftChildKey do right rotation.
      2. Else, do left-right rotation.Balancing the tree with rotationBalancing the tree with rotation
   2. If balanceFactor < -1, it means the height of the right subtree is greater than that of the left subtree. So, do right rotation or right-left rotation
      1. If newNodeKey > rightChildKey do left rotation.
      2. Else, do right-left rotation
6. The final tree is:Final balanced tree

**Algorithm to Delete a node**

A node is always deleted as a leaf node. After deleting a node, the balance factors of the nodes get changed. In order to rebalance the balance factor, suitable rotations are performed.

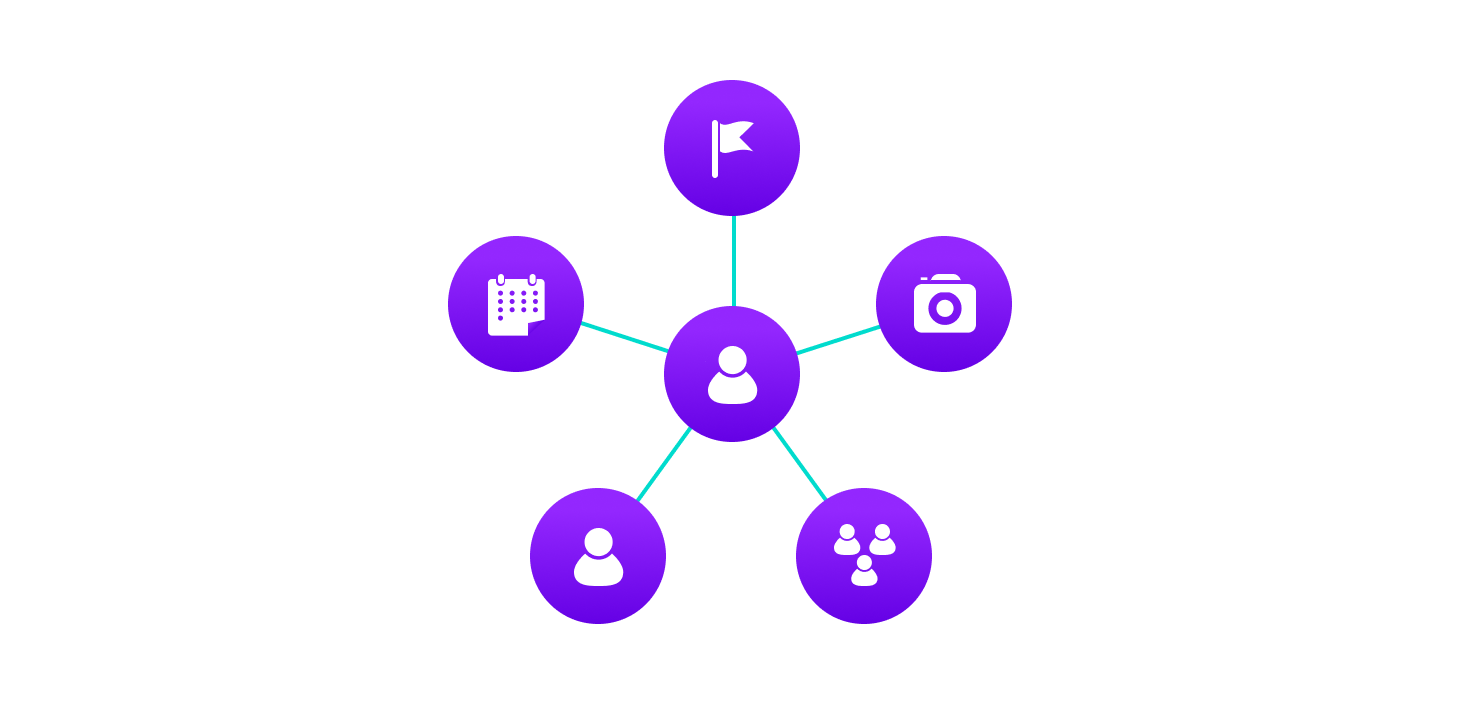
1. Locate nodeToBeDeleted (recursion is used to find nodeToBeDeleted in the code used below).Locating the node to be deleted
2. There are three cases for deleting a node:
   1. If nodeToBeDeleted is the leaf node (ie. does not have any child), then remove nodeToBeDeleted.
   2. If nodeToBeDeleted has one child, then substitute the contents of nodeToBeDeleted with that of the child. Remove the child.
   3. If nodeToBeDeleted has two children, find the inorder successor w of nodeToBeDeleted (ie. node with a minimum value of key in the right subtree).Finding the successor
      1. Substitute the contents of nodeToBeDeleted with that of w.Substitute the node to be deleted
      2. Remove the leaf node w.Remove w
3. Update balanceFactor of the nodes.Update bf
4. Rebalance the tree if the balance factor of any of the nodes is not equal to -1, 0 or 1.
   1. If balanceFactor of currentNode > 1,
      1. If balanceFactor of leftChild >= 0, do right rotation.Right-rotate for balancing the tree
      2. Else do left-right rotation.
   2. If balanceFactor of currentNode < -1,
      1. If balanceFactor of rightChild <= 0, do left rotation.
      2. Else do right-left rotation.
5. The final tree is:Avl tree final

**Graph**

A graph data structure is a collection of nodes that have data and are connected to other nodes.

Let's try to understand this through an example. On facebook, everything is a node. That includes User, Photo, Album, Event, Group, Page, Comment, Story, Video, Link, Note...anything that has data is a node.

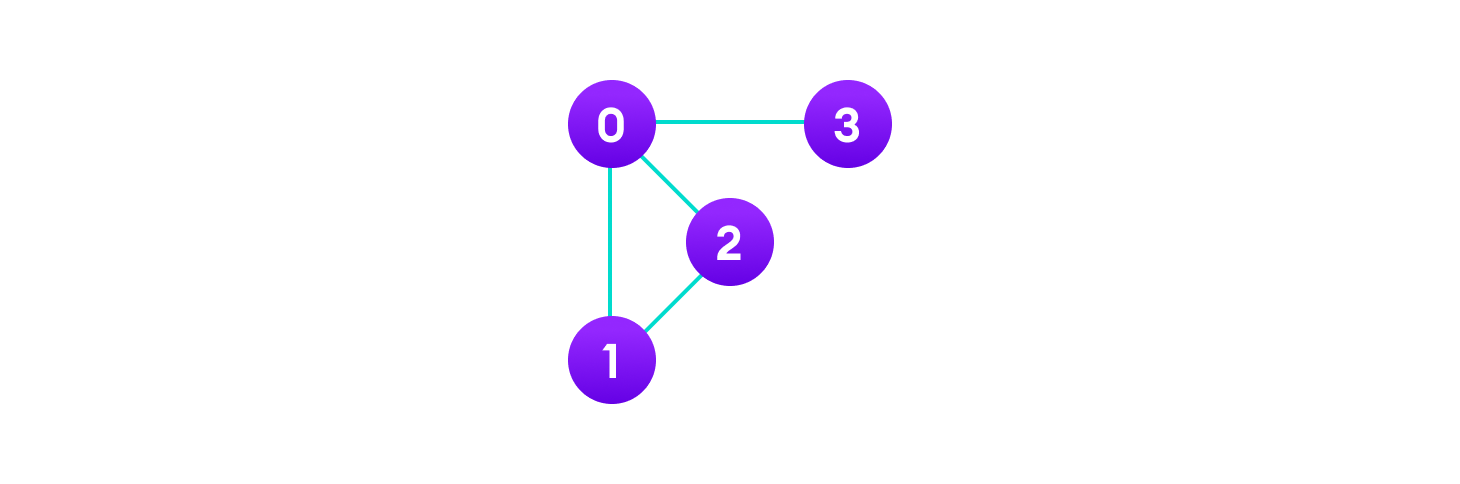
Every relationship is an edge from one node to another. Whether you post a photo, join a group, like a page, etc., a new edge is created for that relationship.

Example of graph data structure

All of facebook is then a collection of these nodes and edges. This is because facebook uses a graph data structure to store its data.

More precisely, a graph is a data structure (V, E) that consists of

* A collection of vertices V
* A collection of edges E, represented as ordered pairs of vertices (u,v)

Vertices and edges

In the graph,

V = {0, 1, 2, 3}

E = {(0,1), (0,2), (0,3), (1,2)}

G = {V, E}

**Graph Terminology**

* **Adjacency**: A vertex is said to be adjacent to another vertex if there is an edge connecting them. Vertices 2 and 3 are not adjacent because there is no edge between them.
* **Path**: A sequence of edges that allows you to go from vertex A to vertex B is called a path. 0-1, 1-2 and 0-2 are paths from vertex 0 to vertex 2.
* **Directed Graph**: A graph in which an edge (u,v) doesn't necessarily mean that there is an edge (v, u) as well. The edges in such a graph are represented by arrows to show the direction of the edge.

**Graph Representation**

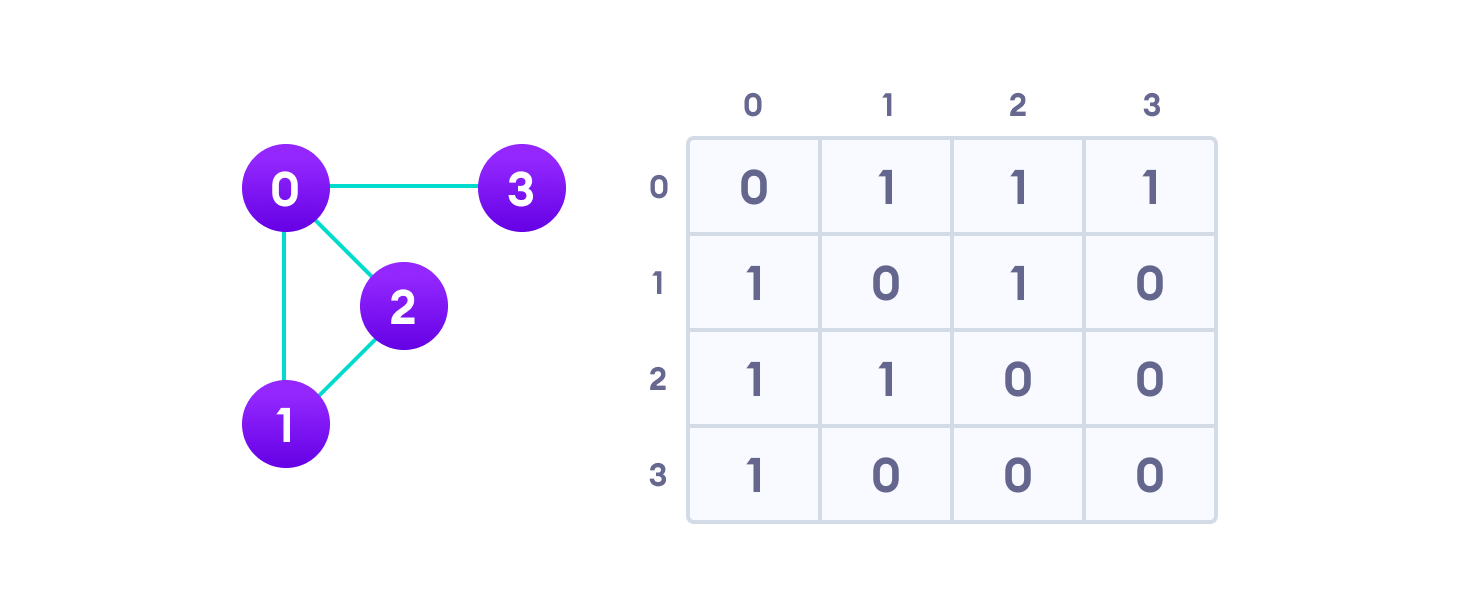
Graphs are commonly represented in two ways:

**1. Adjacency Matrix**

An adjacency matrix is a 2D array of V x V vertices. Each row and column represent a vertex.

If the value of any element a[i][j] is 1, it represents that there is an edge connecting vertex i and vertex j.

The adjacency matrix for the graph we created above is

Graph adjacency matrix

Since it is an undirected graph, for edge (0,2), we also need to mark edge (2,0); making the adjacency matrix symmetric about the diagonal.

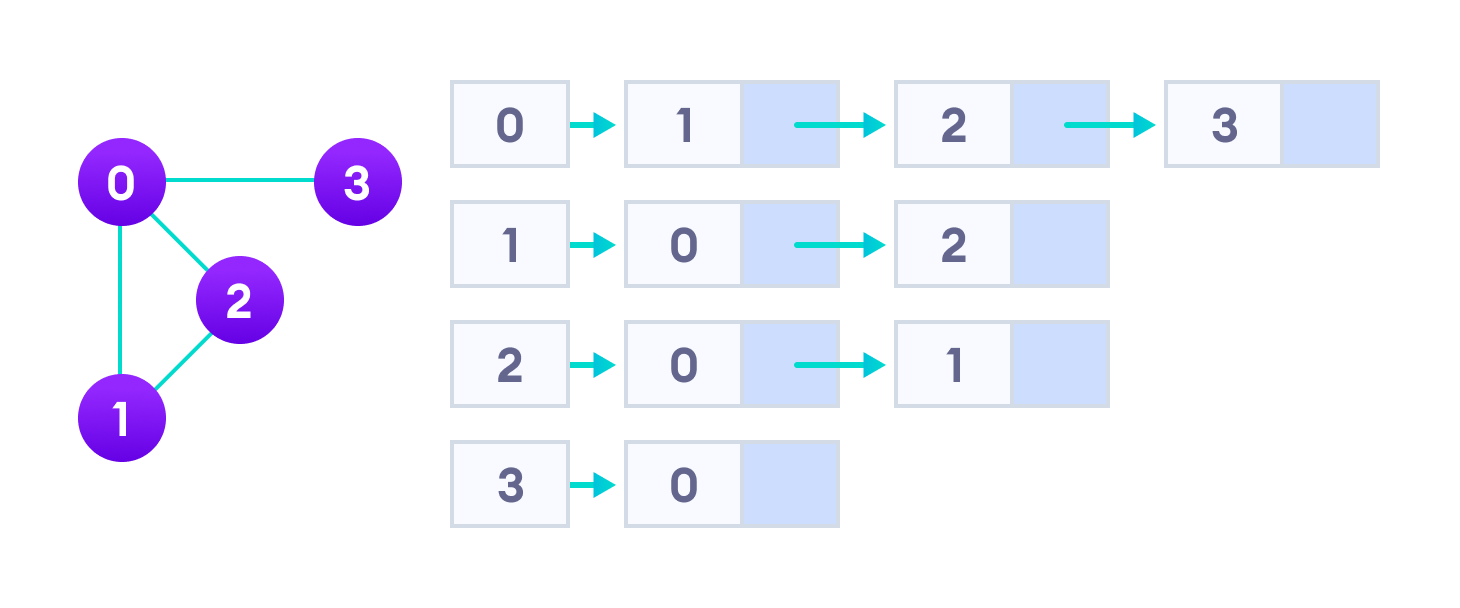
Edge lookup(checking if an edge exists between vertex A and vertex B) is extremely fast in adjacency matrix representation but we have to reserve space for every possible link between all vertices(V x V), so it requires more space.

**2. Adjacency List**

An adjacency list represents a graph as an array of linked lists.

The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

The adjacency list for the graph we made in the first example is as follows:

Adjacency list representation

An adjacency list is efficient in terms of storage because we only need to store the values for the edges. For a graph with millions of vertices, this can mean a lot of saved space.

**Graph Operations**

The most common graph operations are:

* Check if the element is present in the graph
* Graph Traversal
* Add elements(vertex, edges) to graph
* Finding the path from one vertex to another