Tree based DSA (I)

By: Akatsuki Temoka

Topics

- Tree Data Structure
- Tree Traversal
- Binary Tree
- Full Binary Tree
- Perfect Binary Tree
- Complete Binary Tree
- Balanced Binary Tree
- Binary Search Tree
- AVL Tree

Binary Tree

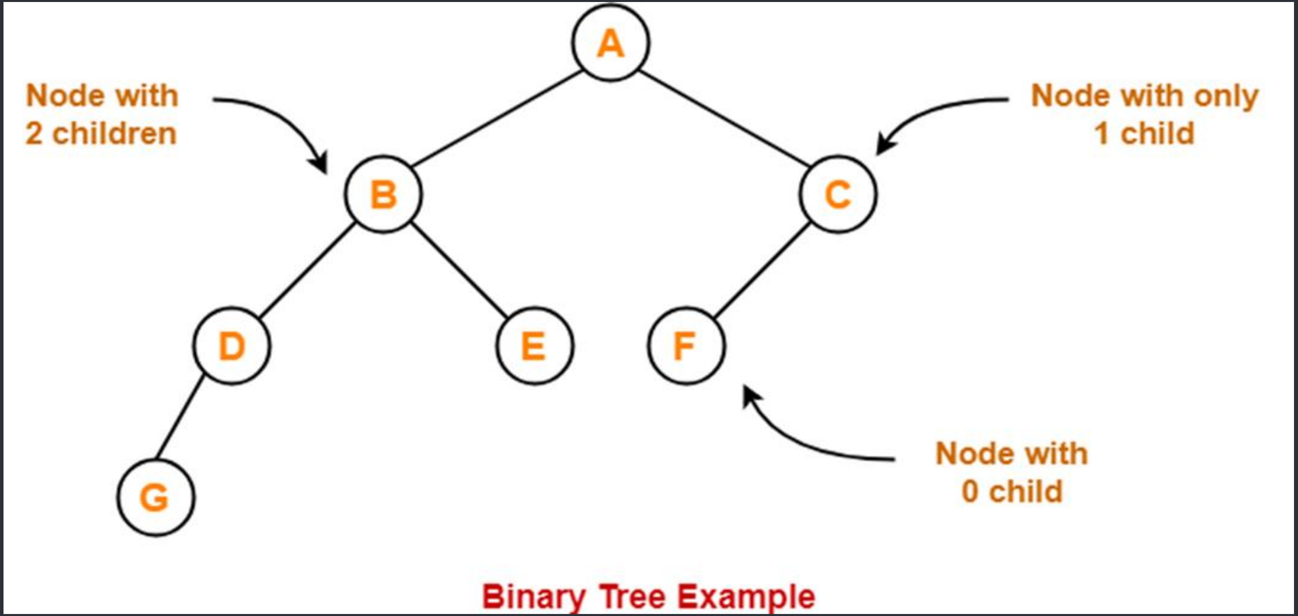
data item

- address of left child

- address of right child

A binary tree is a tree data structure in which each parent node can have at most two children. Each node of a binary tree consists of three items:

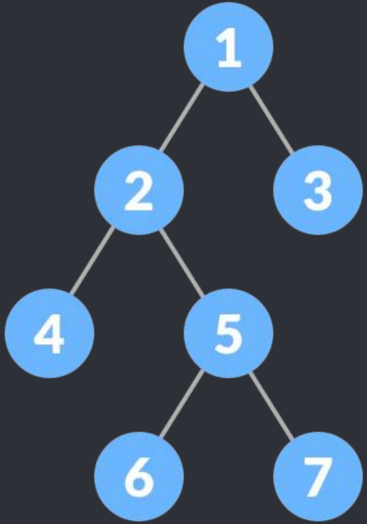
- data item
- address of left child
- address of right child



Types of Binary Tree

1. Full Binary Tree

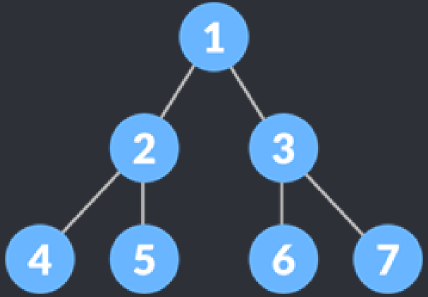
A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children.



Types of Binary Tree

2. 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.



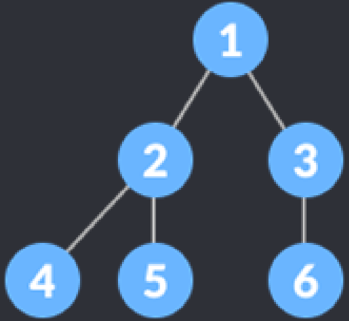
3. Complete Binary Tree

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

Every level must be completely filled

All the leaf elements must lean towards the left.

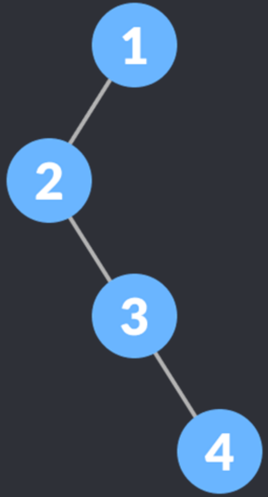
The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.

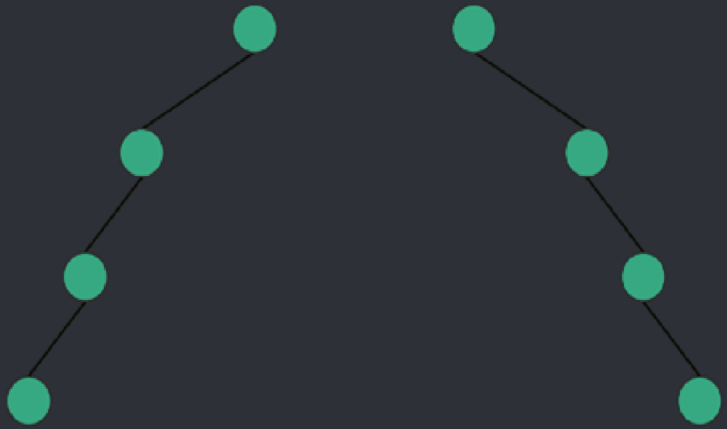


Types of Binary Tree

4. Degenerate or Pathological Tree

A Tree where every internal node has one child. Such trees are performance-wise same as linked list. A degenerate or pathological tree is the tree having a single child either left or right.

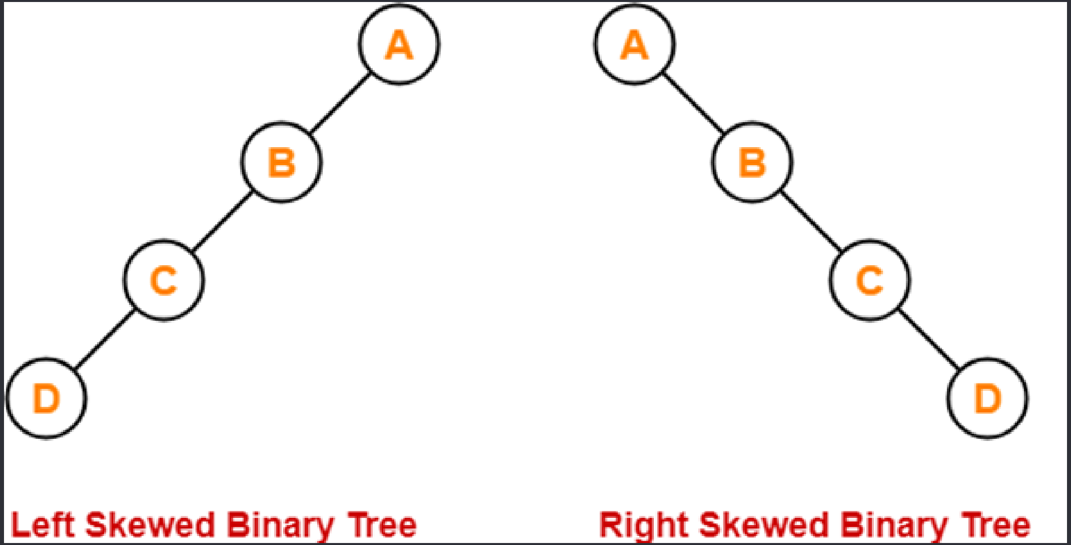




Types of Binary Tree

5. Skewed Binary Tree

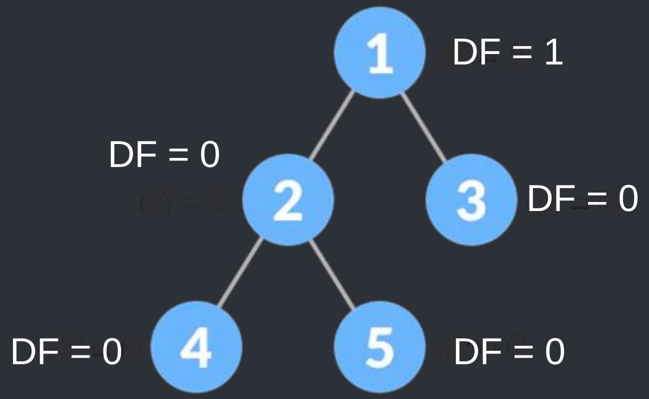
A skewed binary tree is a pathological/degenerate tree in which the tree is either dominated by the left nodes or the right nodes. Thus, there are two types of skewed binary tree: left-skewed binary tree and right-skewed binary tree.



Types of Binary Tree

6. Balanced Binary Tree

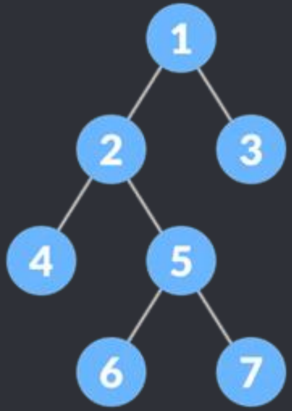
It is a type of binary tree in which the difference between the height of the left and the right subtree for each node is either 0 or 1.



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 Theorems

The number of leaves is i + 1.

The total number of nodes is 2i + 1.

The number of internal nodes is (n – 1) / 2.

The number of leaves is (n + 1) / 2.

The total number of nodes is 2l – 1.

The number of internal nodes is l – 1.

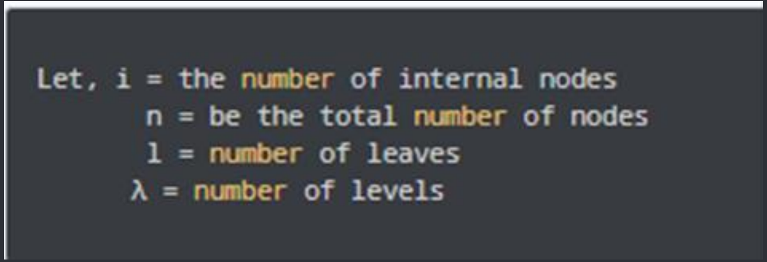
The number of leaves is at most 2λ - 1.

Let, i = the number of internal nodes

n = be the total number of nodes

l = number of leaves

λ = number of levels

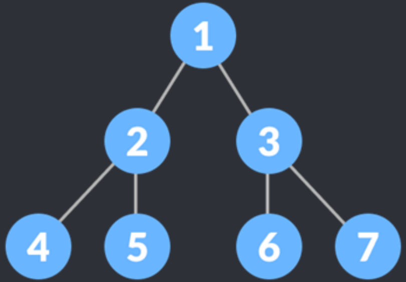


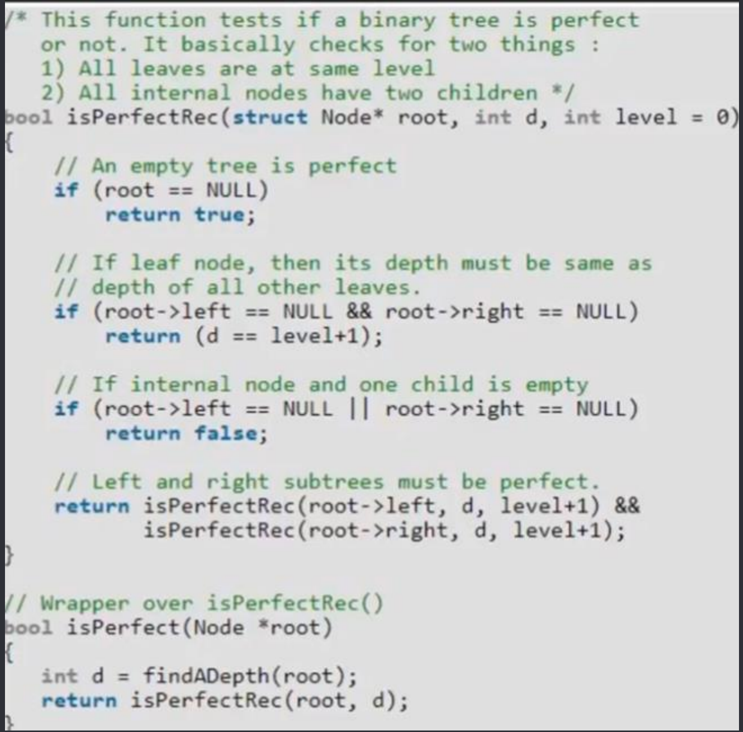
```
Let, i = the number of internal nodes
       n = be the total number of nodes
       l = number of leaves
      λ = number of levels
```

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.

  A Perfect Binary Tree of height h (where height is number of nodes on path from root to leaf) has 2h – 1 nodes.





```
/* This function tests if a binary tree is perfect
   or not. It basically checks for two things :
   1) All leaves are at same level
   2) All internal nodes have two children */
bool isPerfectRec(struct Node* root, int d, int level = 0)
{
    // An empty tree is perfect
    if (root == NULL)
        return true;

    // If leaf node, then its depth must be same as
    // depth of all other leaves.
    if (root->left == NULL && root->right == NULL)
        return (d == level+1);

    // If internal node and one child is empty
    if (root->left == NULL || root->right == NULL)
        return false;

    // Left and right subtrees must be perfect.
    return isPerfectRec(root->left, d, level+1) &&
           isPerfectRec(root->right, d, level+1);
}

// Wrapper over isPerfectRec()
bool isPerfect(Node *root)
{
   int d = findADepth(root);
   return isPerfectRec(root, d);
}
```

root=1	root= 2	root= 4
d= 3	d= 3	d=3
level= 0	level= 1	level=2

Perfect Binary Tree

root=1
d= 3
level= 0

root= 2
d= 3
level= 1

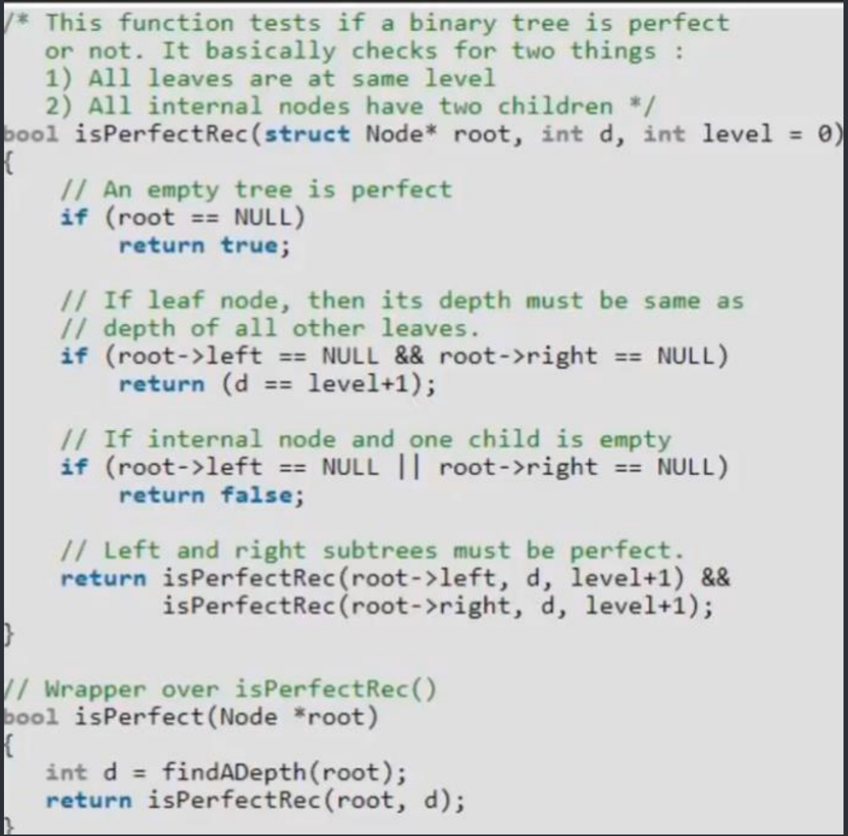
root= 5
d=3
level=2

root=1
d= 3
level= 0

root= 3
d= 3
level= 1

root= 5
d=3
level=2

root= 1
d= 3
level= 0



```
/* This function tests if a binary tree is perfect
   or not. It basically checks for two things :
   1) All leaves are at same level
   2) All internal nodes have two children */
bool isPerfectRec(struct Node* root, int d, int level = 0)
{
    // An empty tree is perfect
    if (root == NULL)
        return true;

    // If leaf node, then its depth must be same as
    // depth of all other leaves.
    if (root->left == NULL && root->right == NULL)
        return (d == level+1);

    // If internal node and one child is empty
    if (root->left == NULL || root->right == NULL)
        return false;

    // Left and right subtrees must be perfect.
    return isPerfectRec(root->left, d, level+1) &&
           isPerfectRec(root->right, d, level+1);
}

// Wrapper over isPerfectRec()
bool isPerfect(Node *root)
{
   int d = findADepth(root);
   return isPerfectRec(root, d);
}
```

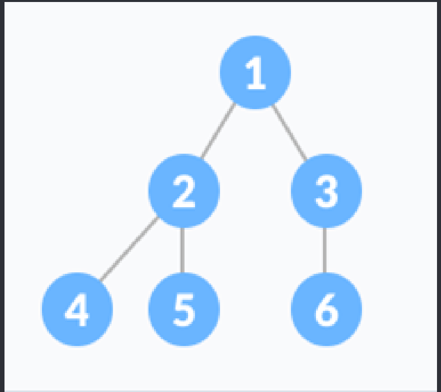
Complete Binary Tree

- A complete binary tree is a binary tree in which all the levels are completely filled except possibly the lowest one, which is filled from the left.

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

  1. All the leaf elements must lean towards the left.

  2. 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.



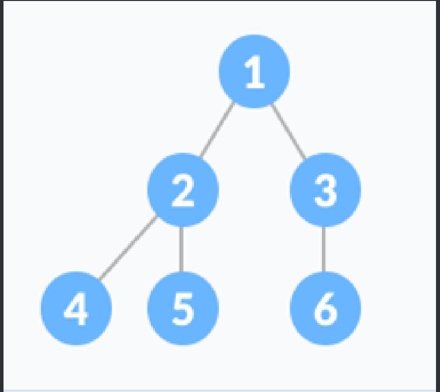
Complete Binary Tree

Complete Binary Tree

- A complete binary tree is a binary tree in which all the levels are completely filled except possibly the lowest one, which is filled from the left.

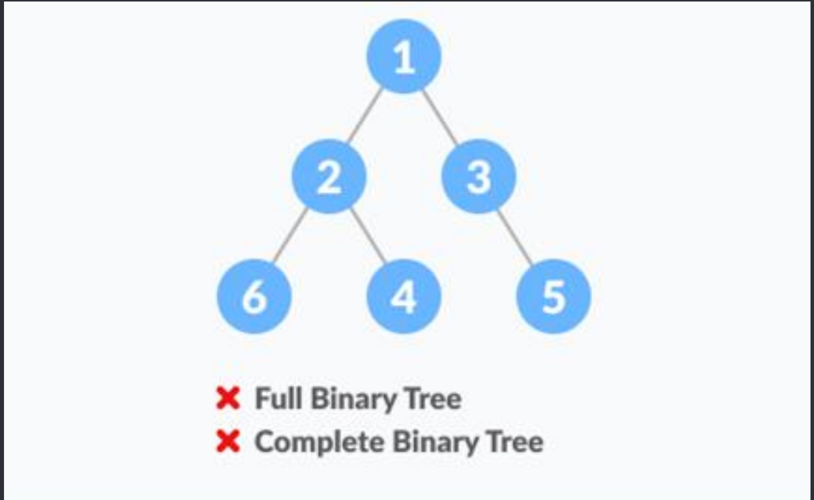
A complete binary tree is just like a full binary tree, but with two major differences

1. All the leaf elements must lean towards the left.
2. 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.

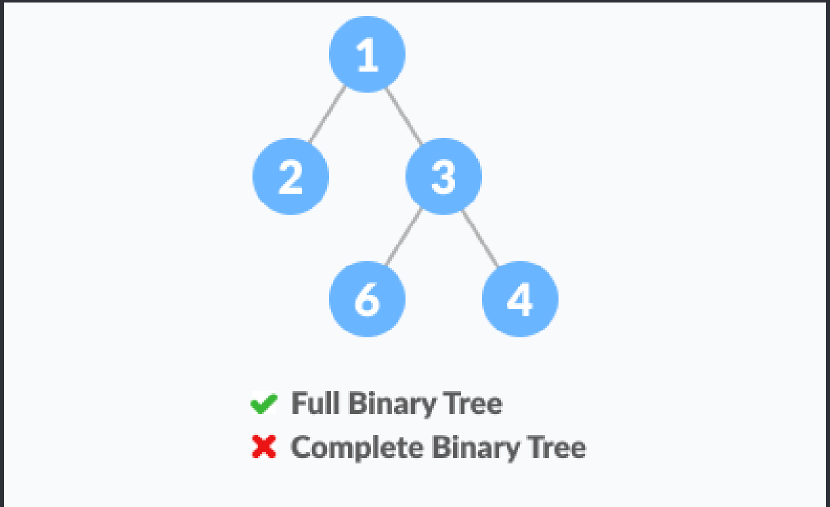


Complete Binary Tree

Full Binary Tree vs Complete Binary Tree

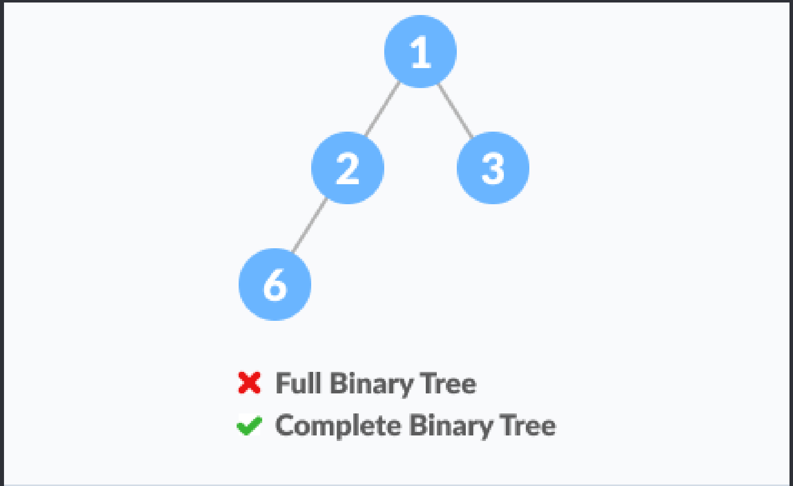


Comparison between full binary tree and complete binary tree

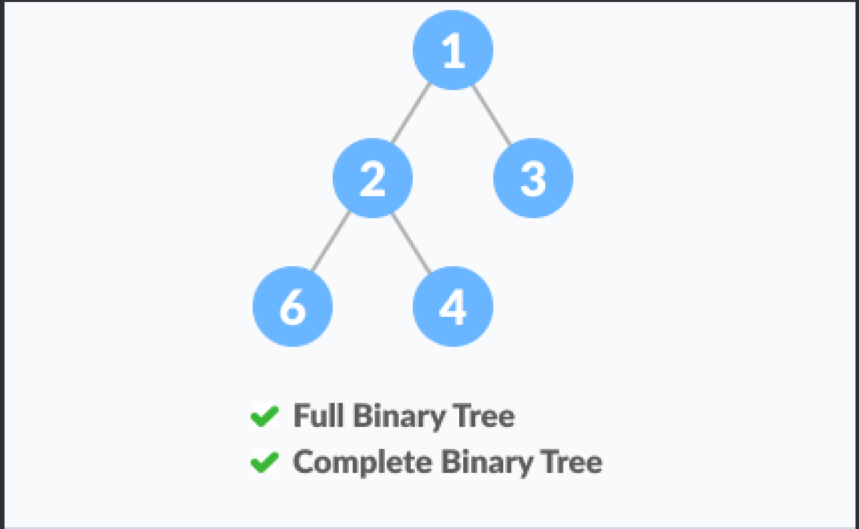


Comparison between full binary tree and complete binary tree

Full Binary Tree vs Complete Binary Tree



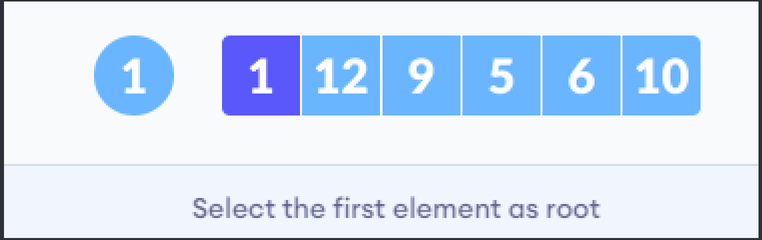
Comparison between full binary tree and complete binary tree



Comparison between full binary tree and complete binary tree

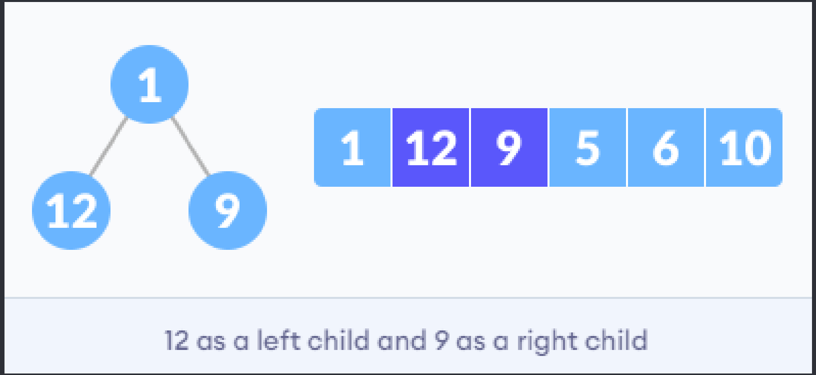
How a Complete Binary Tree is Created?

1. Select the first element of the list to be the root node. (no. of elements on level-I: 1)



Select the first element as root

2. Put the second element as a left child of the root node and the third element as the right child. (no. of elements on level-II: 2)



12 as a left child and 9 as a right child

BALANCED BINARY TREE

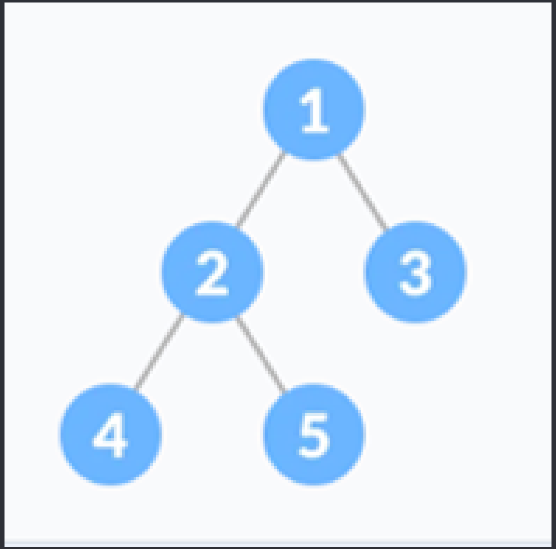
A balanced binary tree, also referred to as a height-balanced binary tree, is defined as a binary tree in which the height of the left and right subtree of any node differ by not more than 1.

Following are the conditions for a height-balanced binary tree:

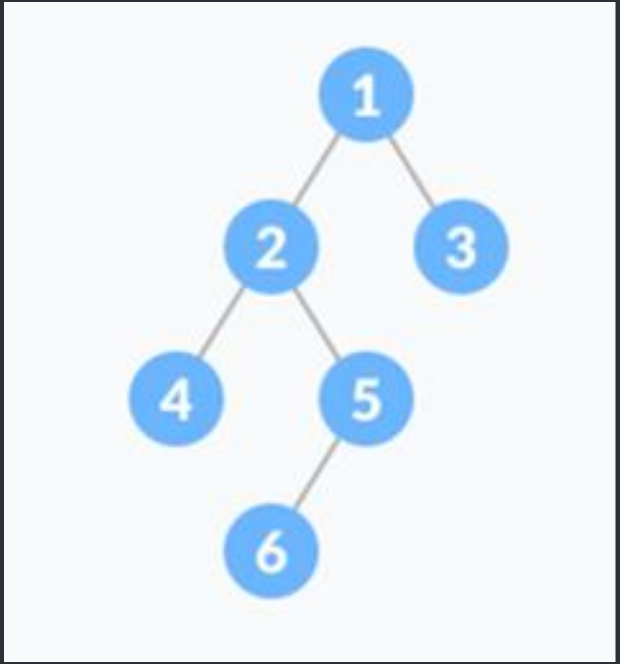
1. difference between the left and the right subtree for any node is not more than one
2. the left subtree is balanced
3. the right subtree is balanced

BALANCED BINARY TREE

First Example

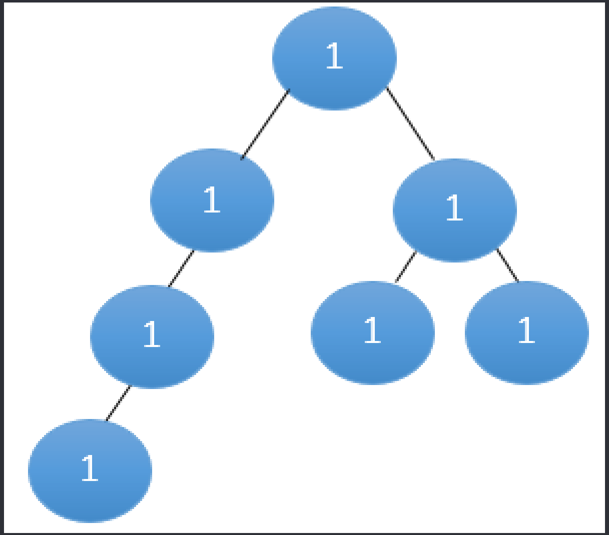


Second Example



BALANCED BINARY TREE

EXAMPLE NO. 3



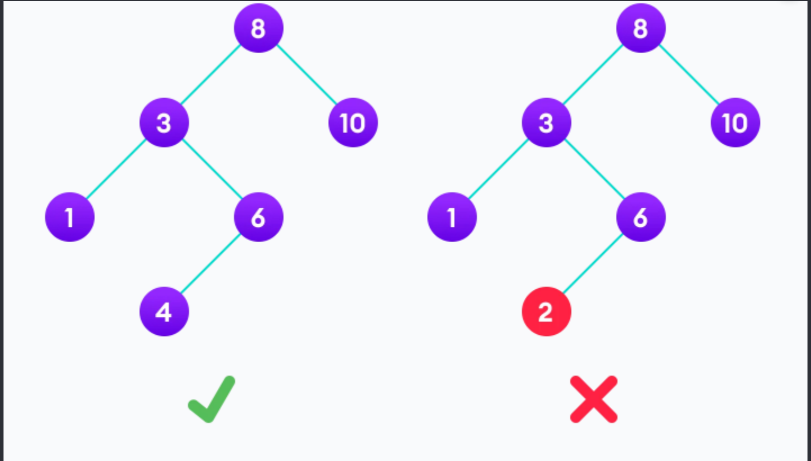
```
Bool check(node)
    if node == null
        return true
    Lh=findHleft(node->left) // left subtree
    Rh=findHright(node->right) // right subtree

    If(abs(rh-lh)>1) return false;

    Bool left = check(node->left);
    Bool right = check(node->right);

    If(!left || !right)return false;

    return true;
```

Binary Search Tree(BST)

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

The properties that separate a binary search tree from a regular binary tree is

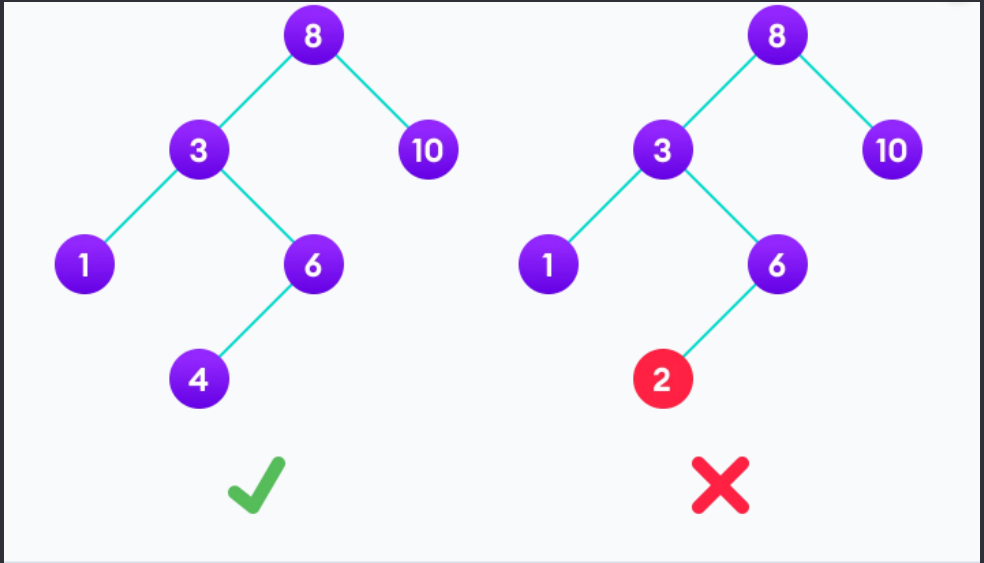
1. All nodes of left subtree are less than the root node
2. All nodes of right subtree are more than the root node
3. Both subtrees of each node are also BSTs i.e. they have the above two properties



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

Binary Search Tree(BST)

- 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.



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

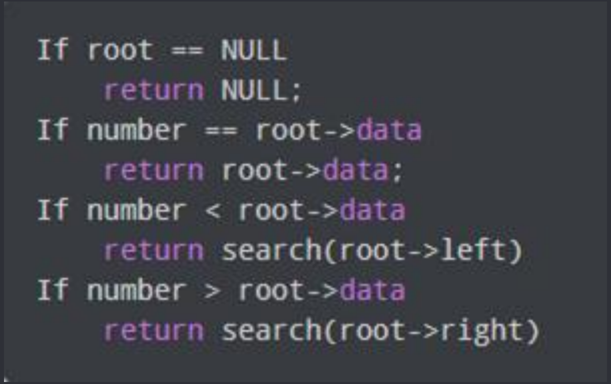
There are two basic operations that you can perform on a binary search tree:

Binary Search Tree(BST)

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.

Algorithm:

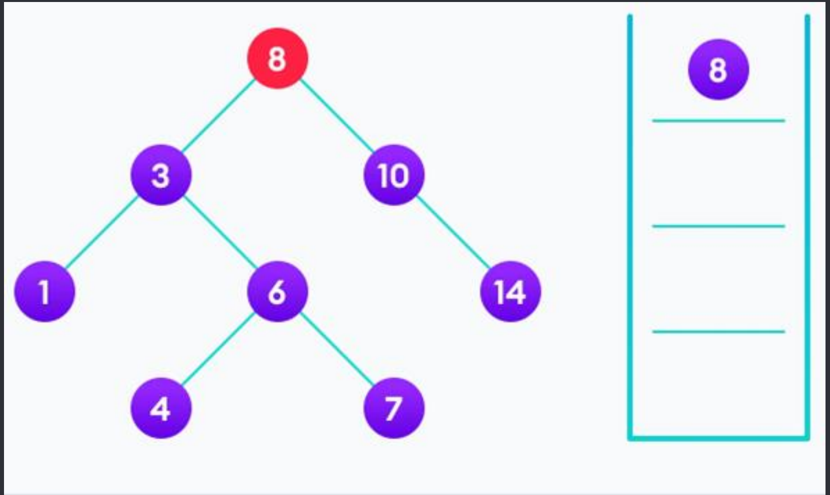


```
If root == NULL
    return NULL;
If number == root->data
    return root->data;
If number < root->data
    return search(root->left)
If number > root->data
    return search(root->right)
```

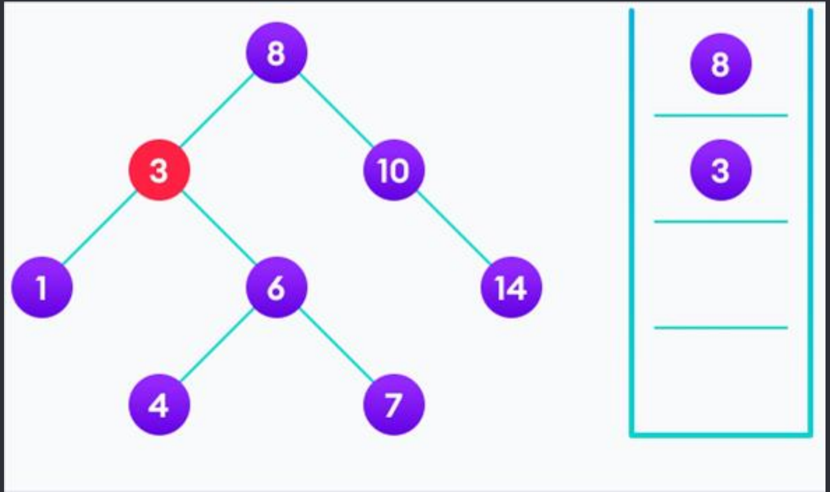
If the value is below the root, we can say for sure that the value is not in the right subtree; we need to only search in the left subtree and if the value is above the root, we can say for sure that the value is not in the left subtree; we need to only search in the right subtree.

Binary Search Tree(BST)

Let us try to visualize this with a diagram.



4 is not found so, traverse through the left subtree of 8



4 is not found so, traverse through the right subtree of 3

Binary Search Tree(BST)

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.

Binary Search Tree(BST)

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.

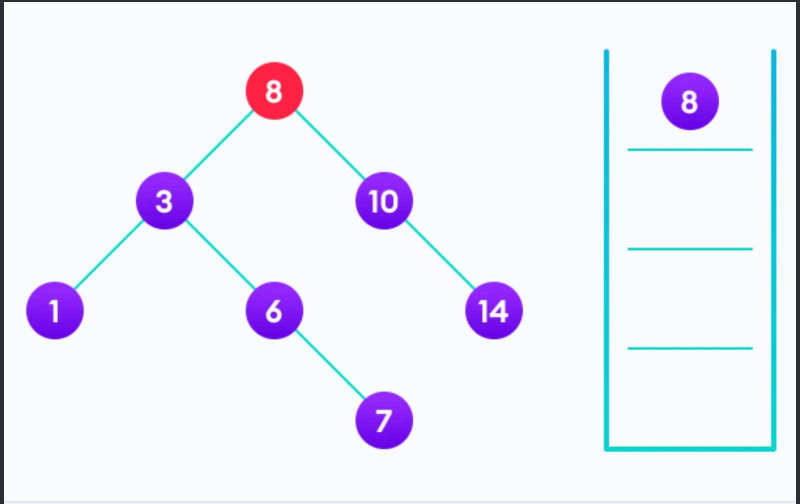
We keep going to either right subtree or left subtree depending on the value and when we reach a point left or right subtree is null, we put the new node there.

Algorithm:

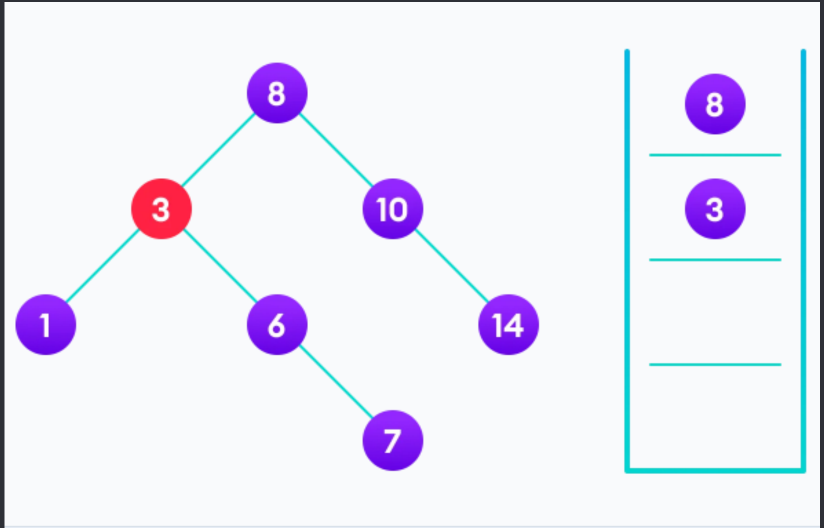
```
If node == NULL
    return createNode(data)
if (data < node->data)
    node->left  = insert(node->left, data);
else if (data > node->data)
    node->right = insert(node->right, data);
return node;
```

Binary Search Tree(BST)

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



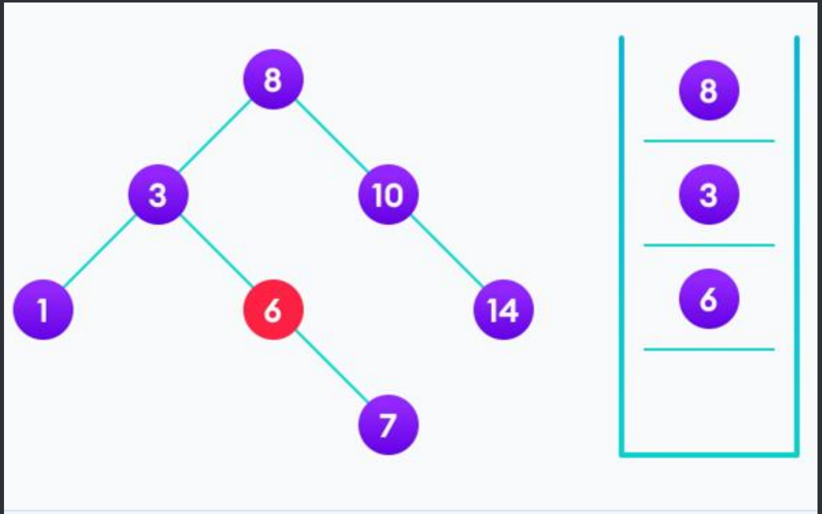
4<8 so, transverse through the left child of 8



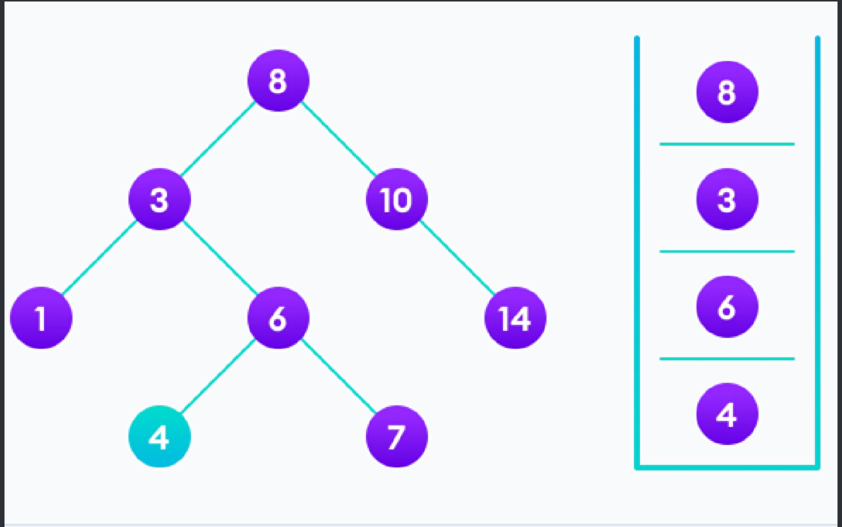
4>3 so, transverse through the right child of 8

Binary Search Tree(BST)

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



4<6 so, transverse through the left child of 6



Insert 4 as a left child of 6

Binary Search Tree(BST)

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.

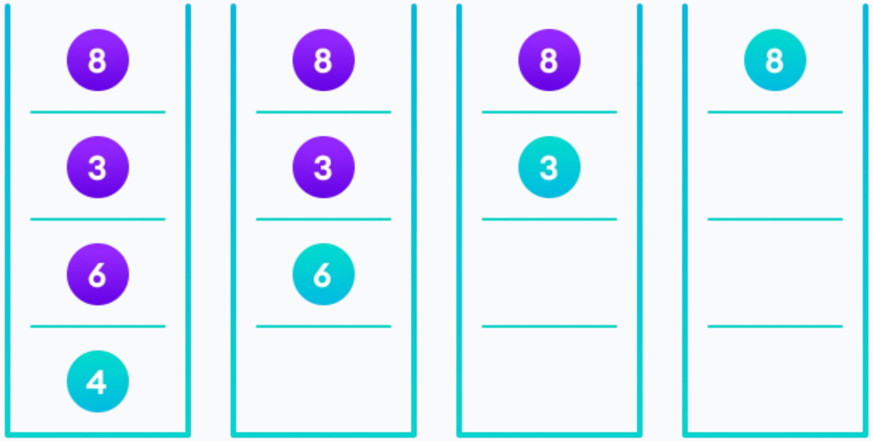


Image showing the importance of returning the root element at the end so that the elements don't lose their position during the upward recursion step.

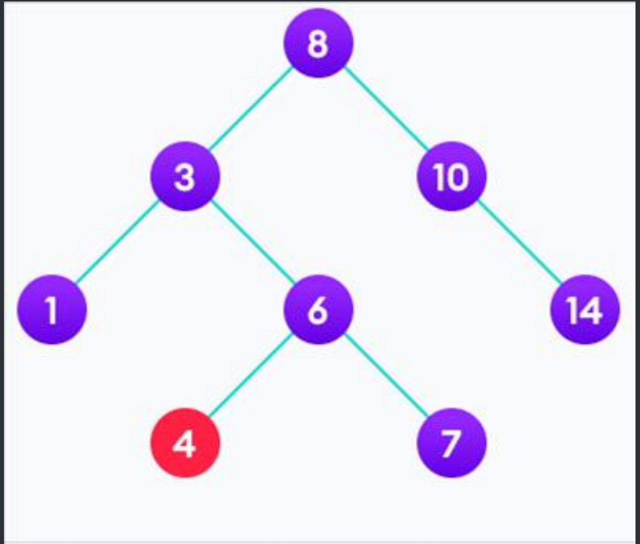
Binary Search Tree(BST)

Deletion Operation

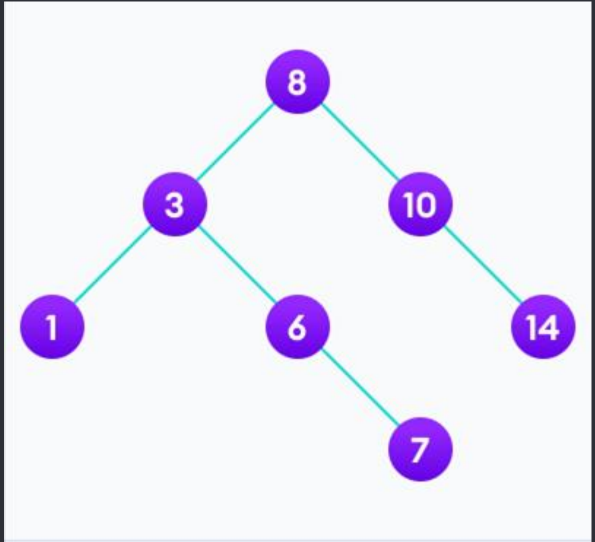
There are three cases for deleting a node from a binary search tree.

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 deleted



Delete the node

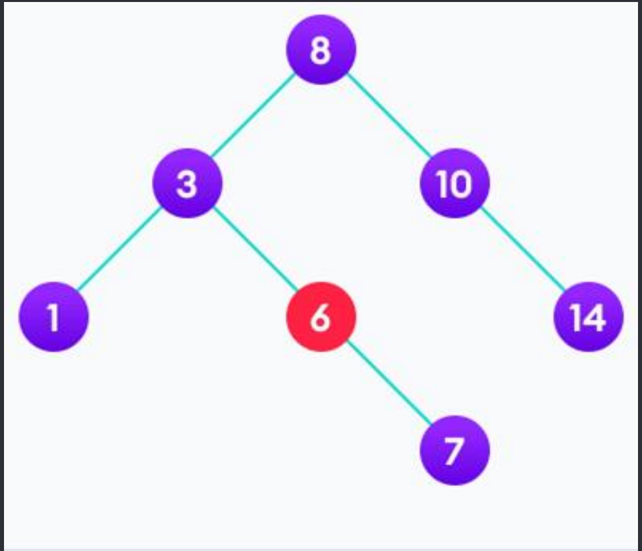
Binary Search Tree(BST)

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.

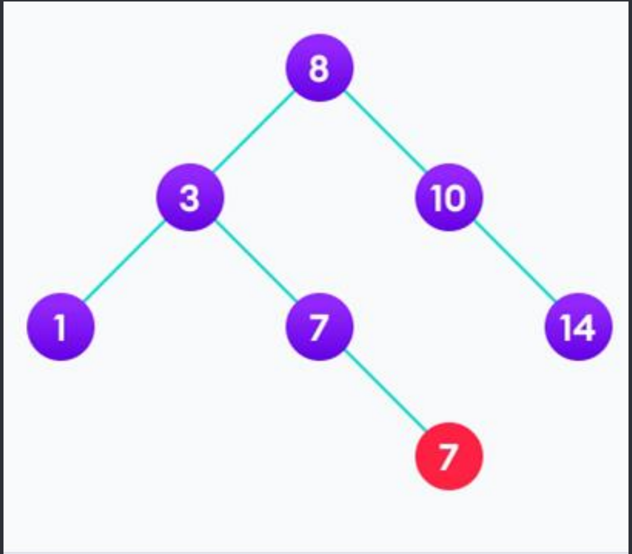


6 is to be deleted

Binary Search Tree(BST)

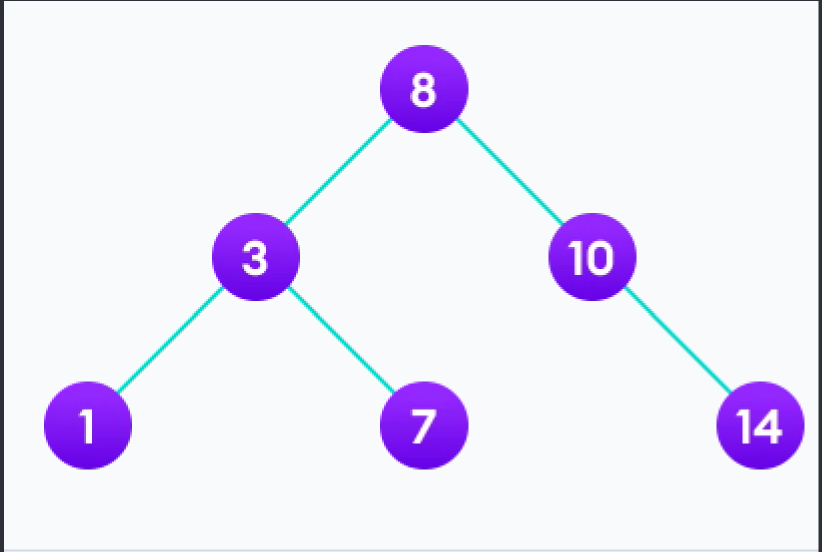
- Case II

Fig.2



copy the value of its child to the node and delete the child

Fig.3



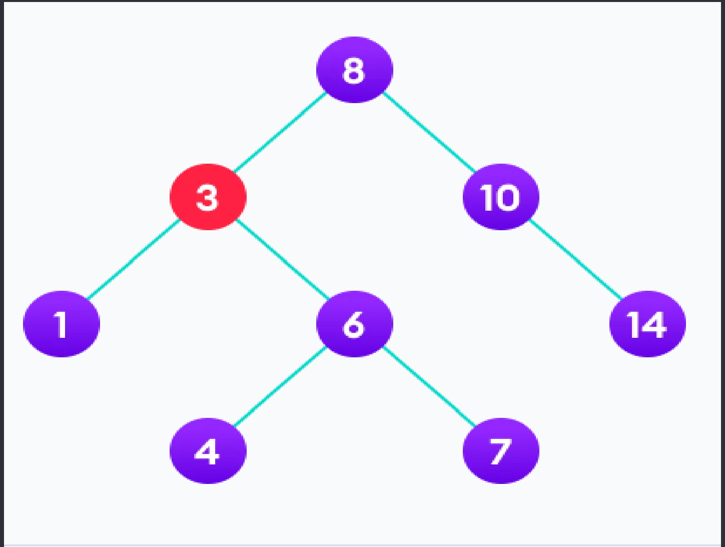
Final tree

Binary Search Tree(BST)

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.

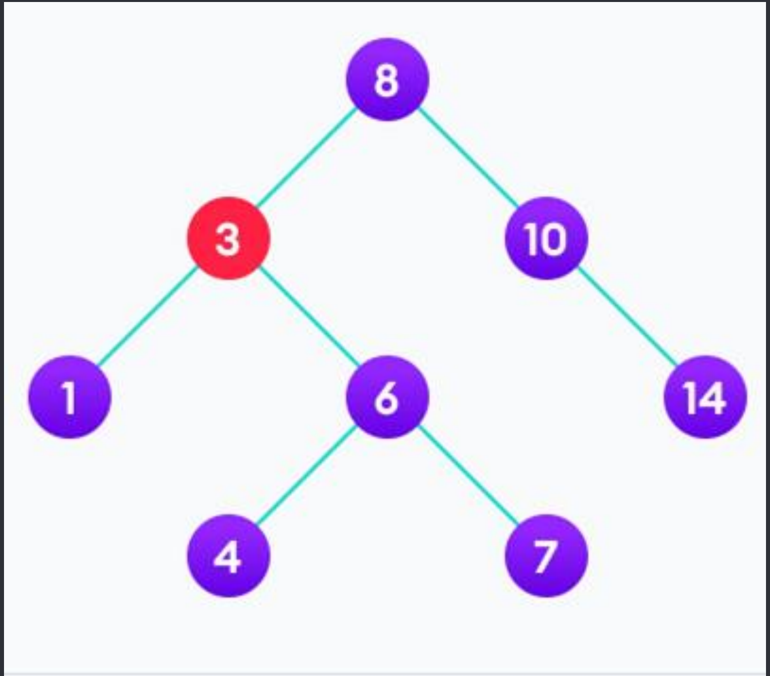


3 is to be deleted

Binary Search Tree(BST)

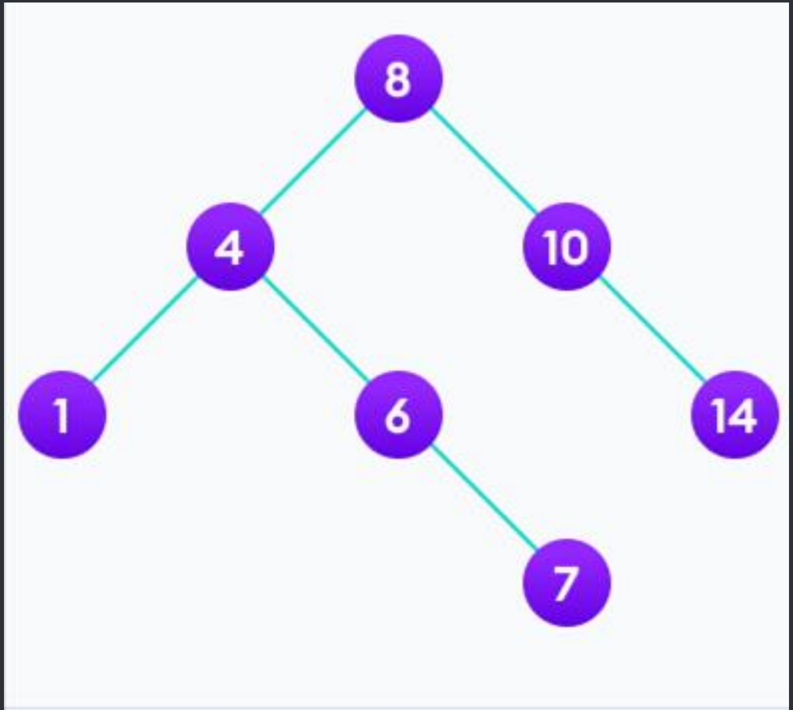
- Case III

Fig.2



Copy the value of the inorder successor (4) to the node

Fig.3



Delete the inorder successor

AVL Tree

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

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

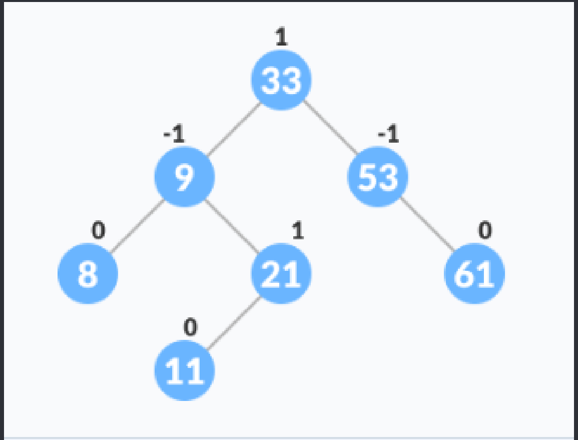
Balance Factor

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:

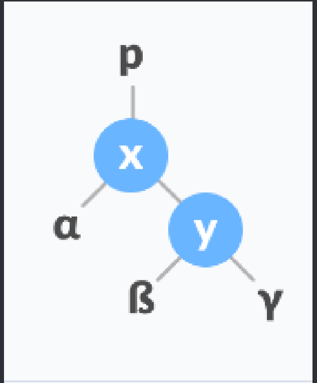
1. Left Rotate
2. Right Rotate

Left Rotate

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

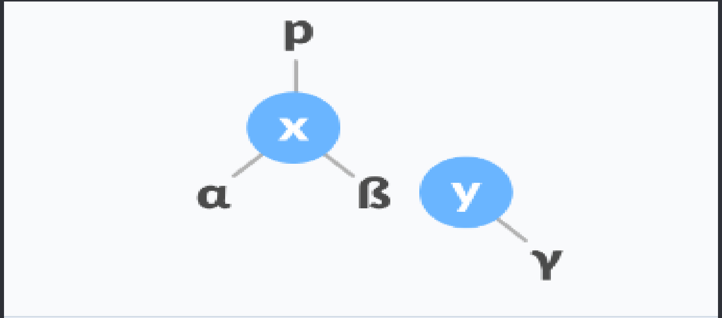
Algorithm

1. Let the initial tree be:



Left rotate

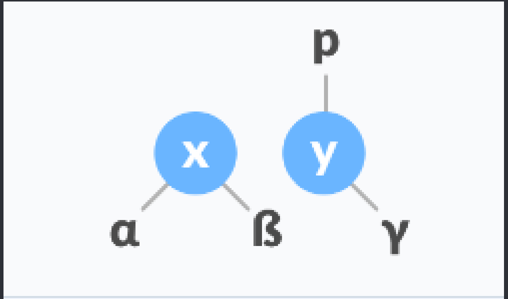
2. If y has a left subtree, assign x as the parent of the left subtree of y.



Assign x as the parent of the left subtree of y

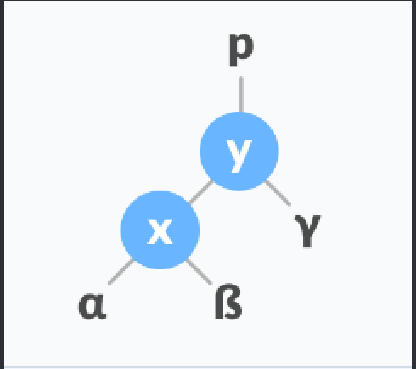
Left Rotate

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.

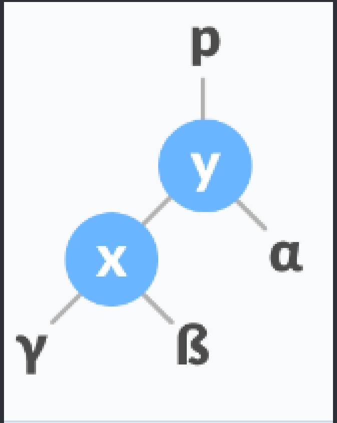


Assign y as the parent of x.

Right Rotate

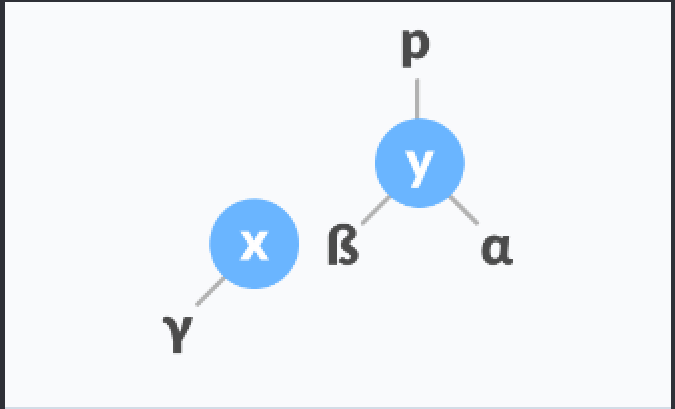
In left-rotation, the arrangement of the nodes on the left is transformed into the arrangements on the right node.

1. Let the initial tree be:



Initial tree

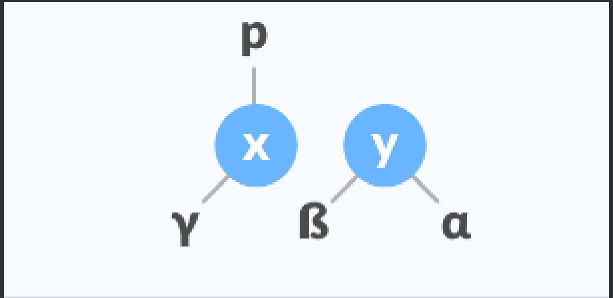
2. If x has a right subtree, assign y as the parent of the right subtree of x.



Assign y as the parent of the right subtree of x

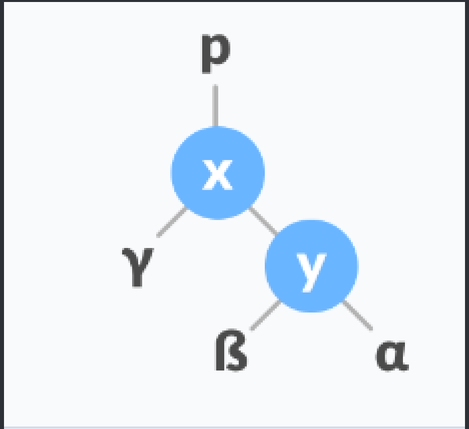
Right Rotate

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

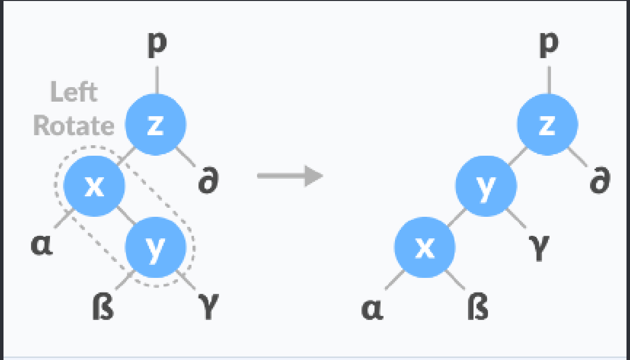


Assign x as the parent of y

Left-Right and Right-Left Rotate

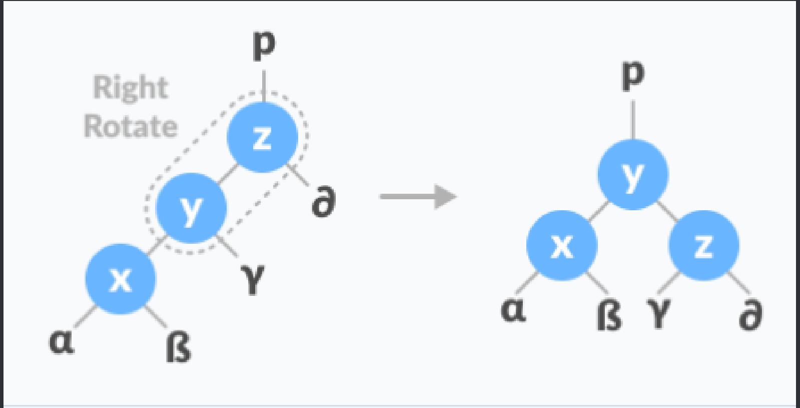
In left-right rotation, the arrangements are first shifted to the left and then to the right.

1. Do left rotation on x-y.



Left rotate x-y

2. Do right rotation on y-z.

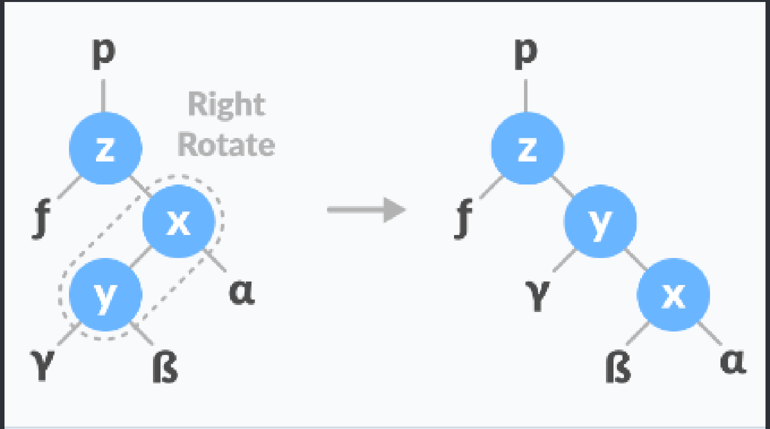


Right rotate z-y

Left-Right and Right-Left Rotate

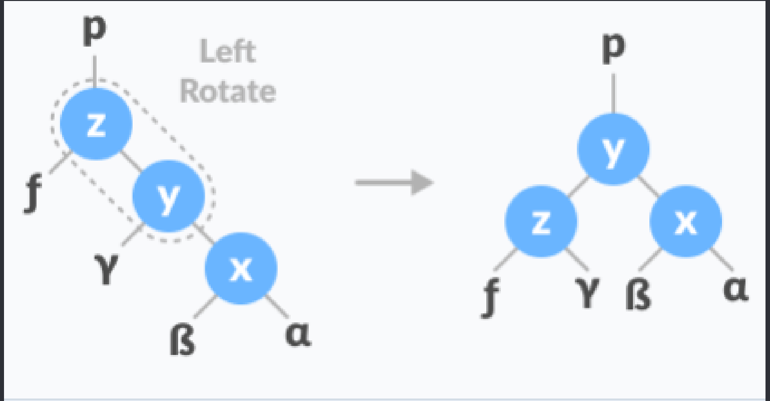
In right-left rotation, the arrangements are first shifted to the right and then to the left.

1. Do right rotation on x-y.



Right rotate x-y

2. Do left rotation on z-y.

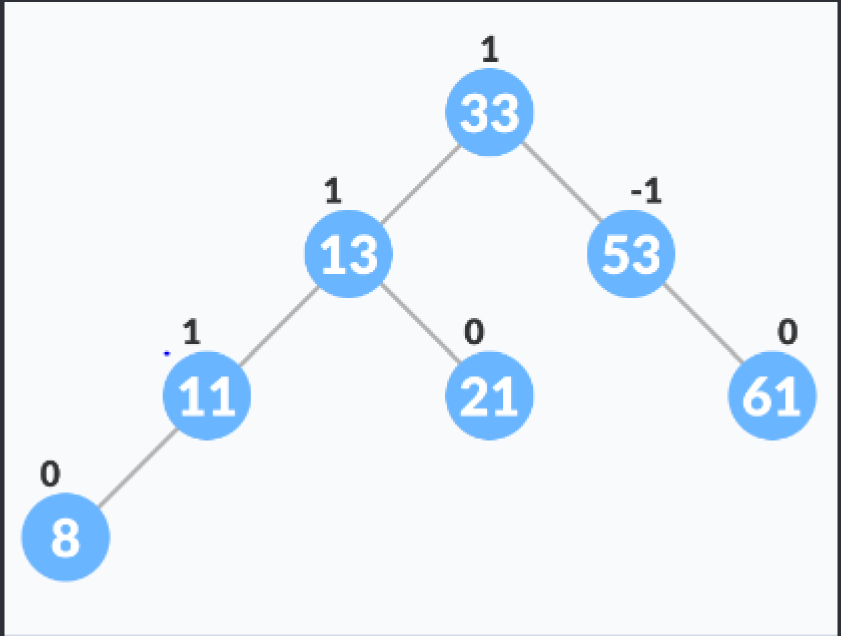


Left rotate z-y

Algorithm to insert a newNode

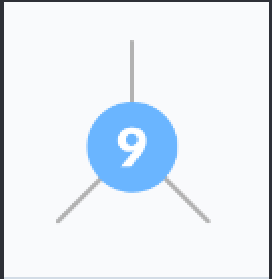
A newNode is always inserted as a leaf node with balance factor equal to 0.

Let the initial tree be:



Initial tree for insertion

Let the node to be inserted be:



New node

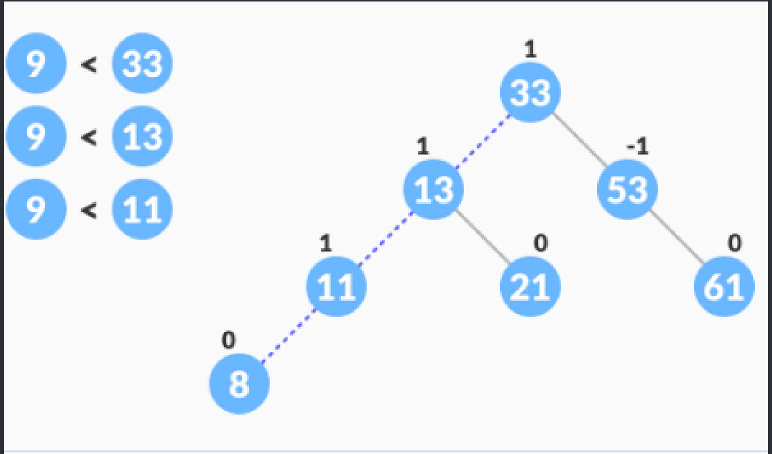
Algorithm to insert a newNode

2. Go to the appropriate leaf node to insert a newNode using the following recursive steps. Compare newKey with rootKey of the current tree.

a. newKey < rootKey, call insertion algorithm on the left subtree of the current node until the leaf node is reached.

b. Else if newKey > rootKey, call insertion algorithm on the right subtree of current node until the leaf node is reached.

c. Else, return leafNode.



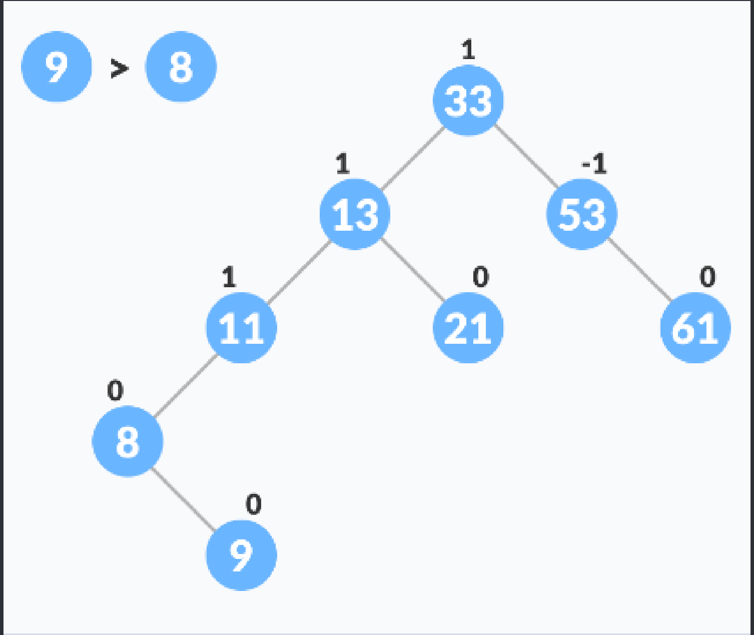
Finding the location to insert newNode

Algorithm to insert a newNode

3. Compare leafKey obtained from the above steps with newKey:

a. If newKey < leafKey, make newNode as the leftChild of leafNode.

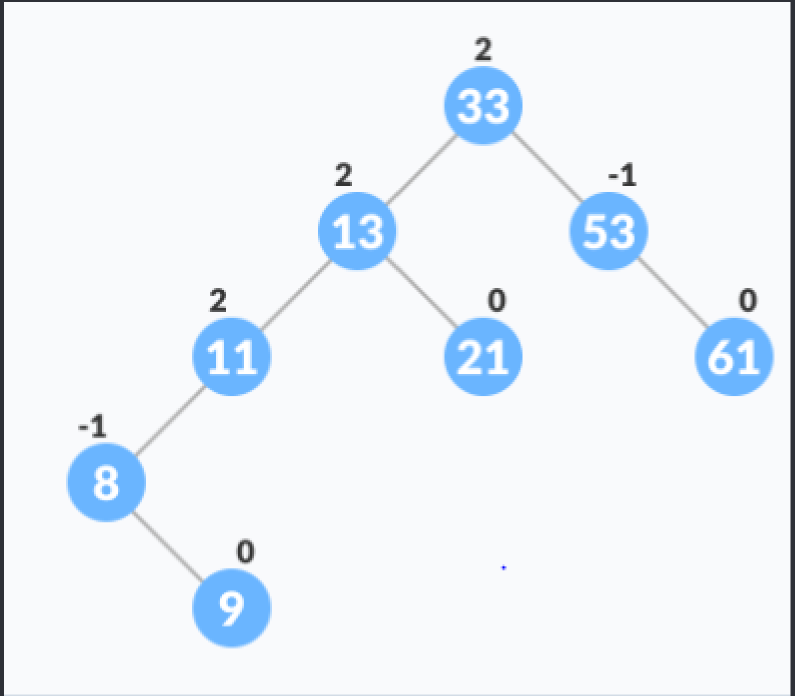
b. Else, make newNode as rightChild of leafNode.



Inserting the new node

Algorithm to insert a newNode

4. Update balanceFactor of the nodes.



Updating the balance factor after insertion

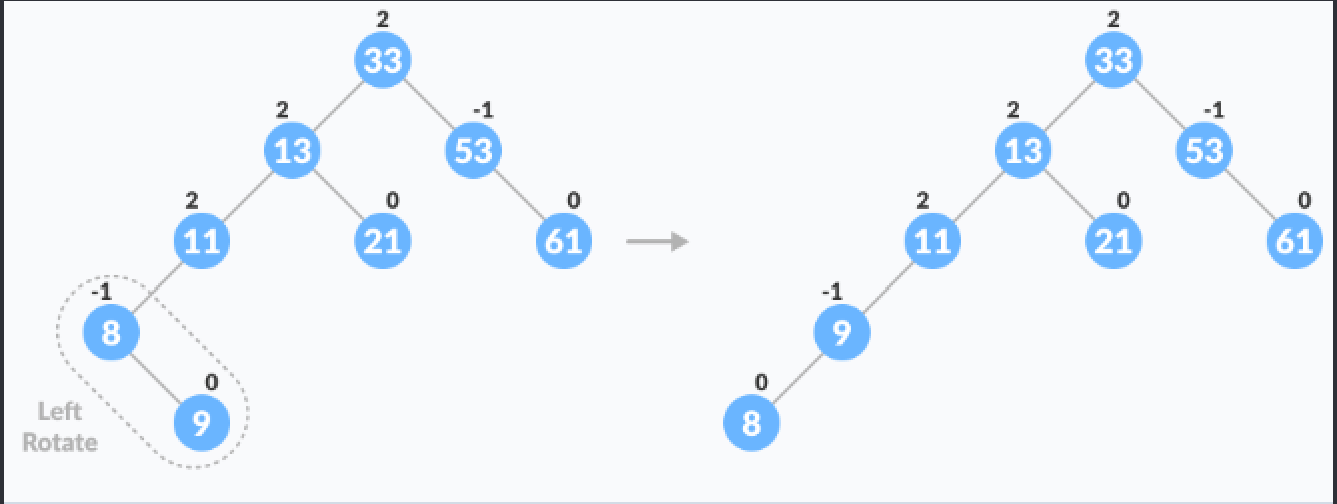
Algorithm to insert a newNode

5. If the nodes are unbalanced, then rebalance the node.

A. 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

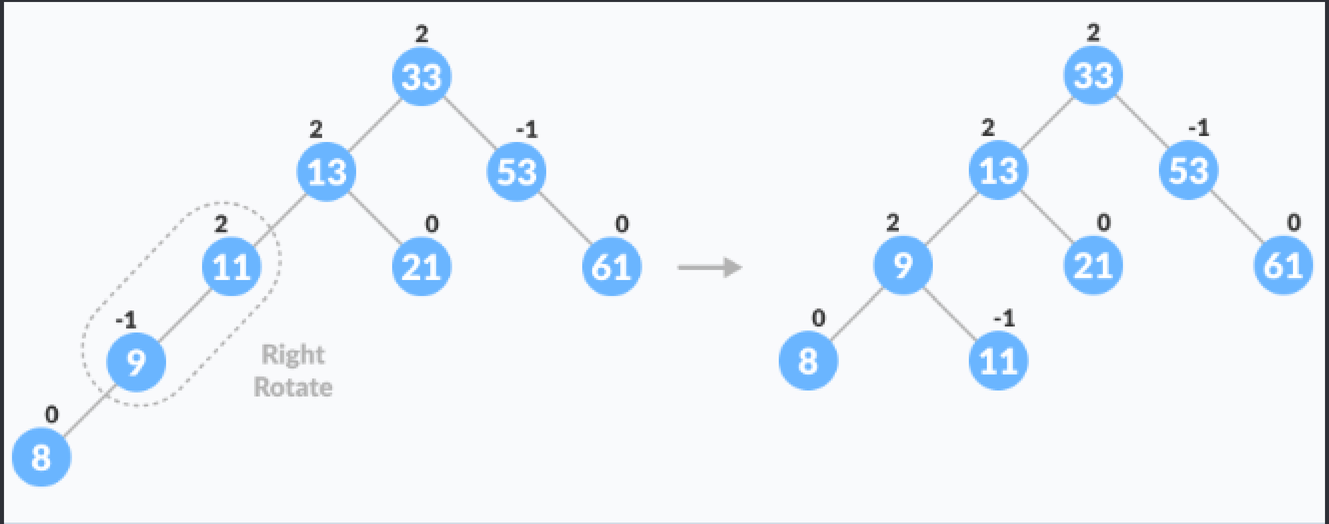
a. If newNodeKey < leftChildKey do right rotation.

b. Else, do left-right rotation.



Balancing the tree with rotation

Algorithm to insert a newNode



Balancing the tree with rotation

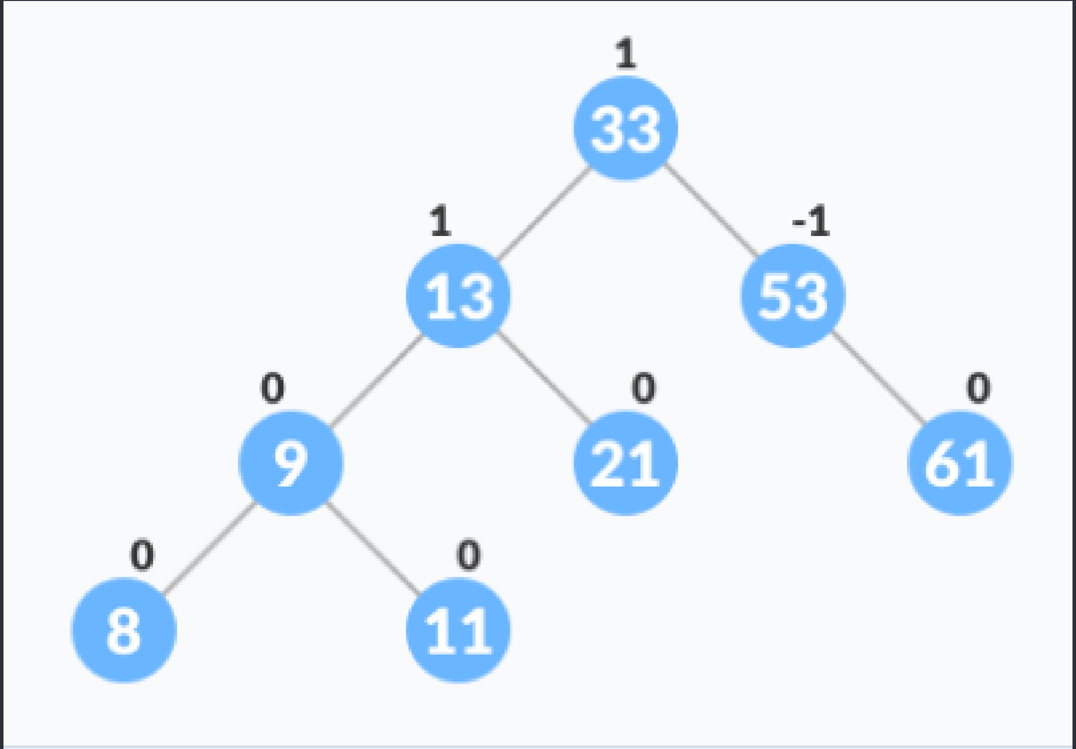
B. If balanceFactor < -1, it means the height of the right subtree is greater than that of the left subtree. So, do right rotation or right-left rotation

a. If newNodeKey > rightChildKey do left rotation.

b. Else, do right-left rotation

Algorithm to insert a newNode

The final tree is:

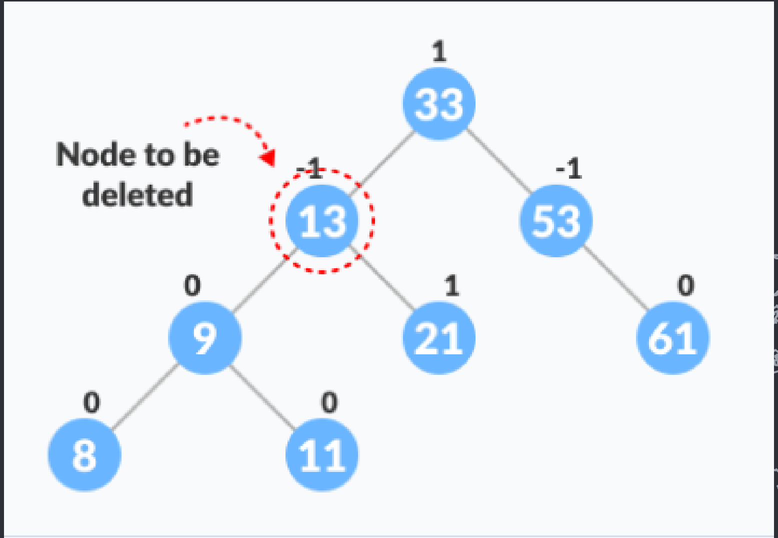


Final balanced tree

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

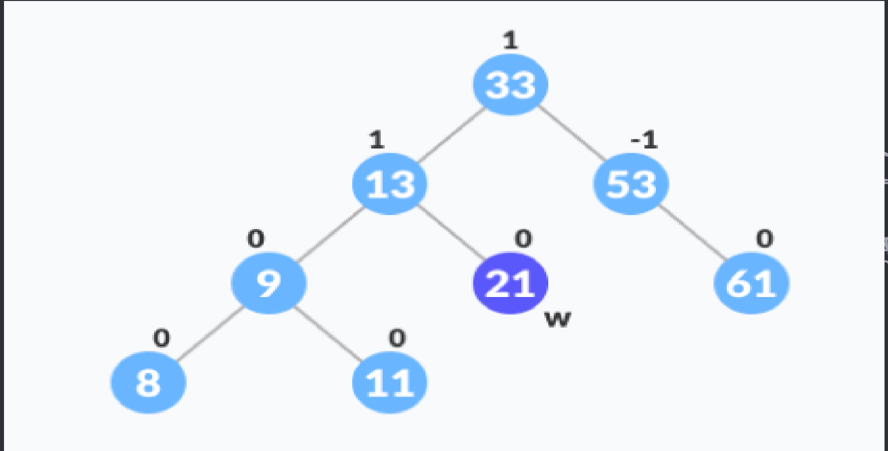
Algorithm to Delete a node

2. There are three cases for deleting a node:

a. If nodeToBeDeleted is the leaf node (ie. does not have any child), then remove nodeToBeDeleted.

b. If nodeToBeDeleted has one child, then substitute the contents of nodeToBeDeleted with that of the child. Remove the child.

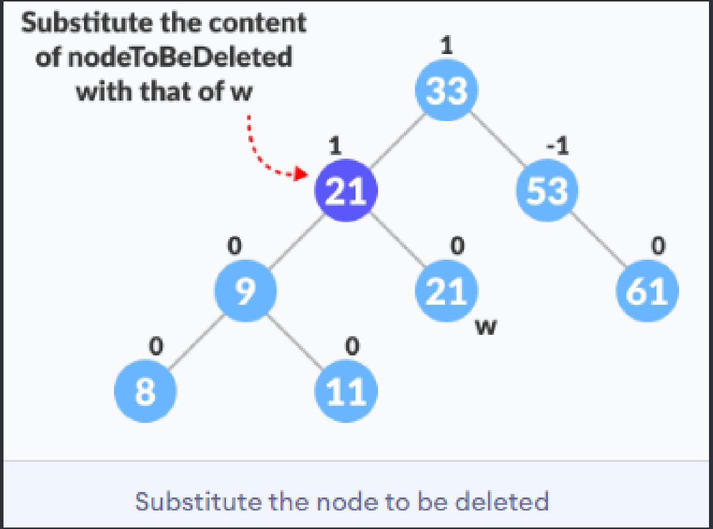
c. If nodeToBeDeleted has two children, find the inorder successor w of nodeToBeDeleted (ie. node with a minimum value of key in the right subtree).



Finding the successor

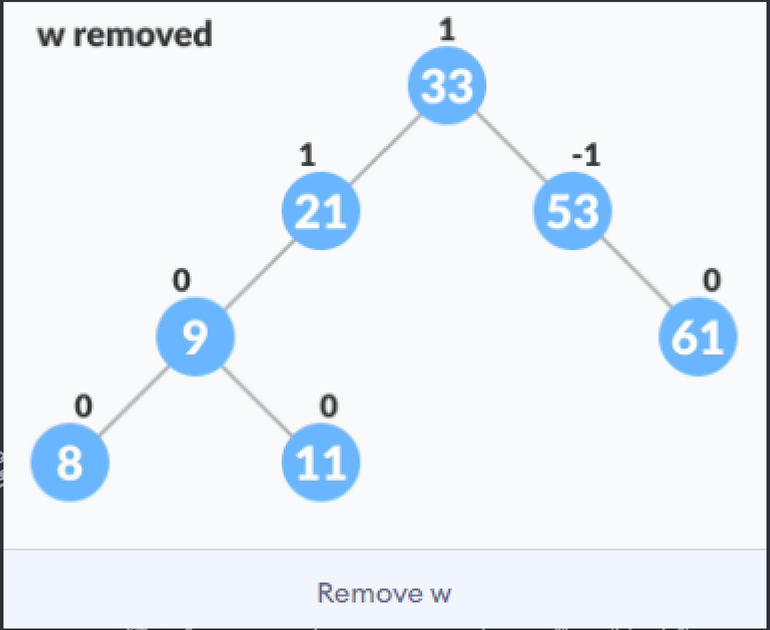
Algorithm to Delete a node

a. Substitute the contents of nodeToBeDeleted with that of w.



Substitute the node to be deleted

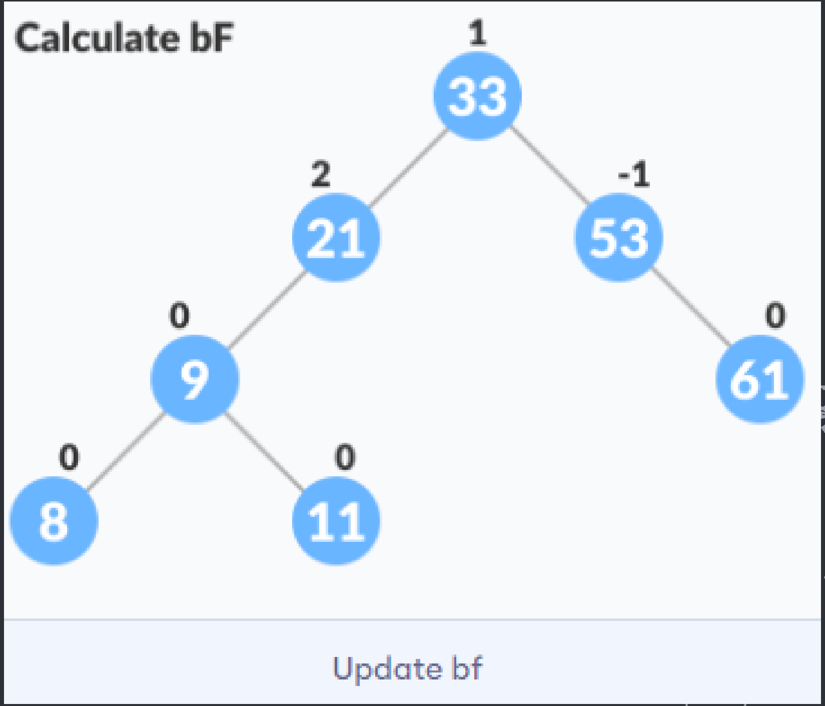
b. Remove the leaf node w.



Remove w

Algorithm to Delete a node

3. Update balanceFactor of the nodes.

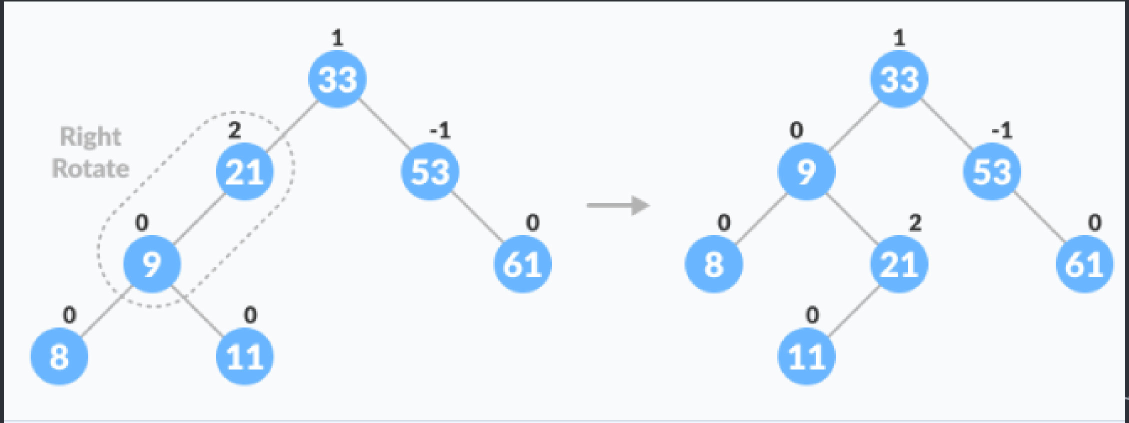


Algorithm to Delete a node

4. Rebalance the tree if the balance factor of any of the nodes is not equal to -1, 0 or 1.

A. If balanceFactor of currentNode > 1,

a. If balanceFactor of leftChild >= 0, do right rotation.



Right-rotate for balancing the tree

b. Else do left-right rotation.

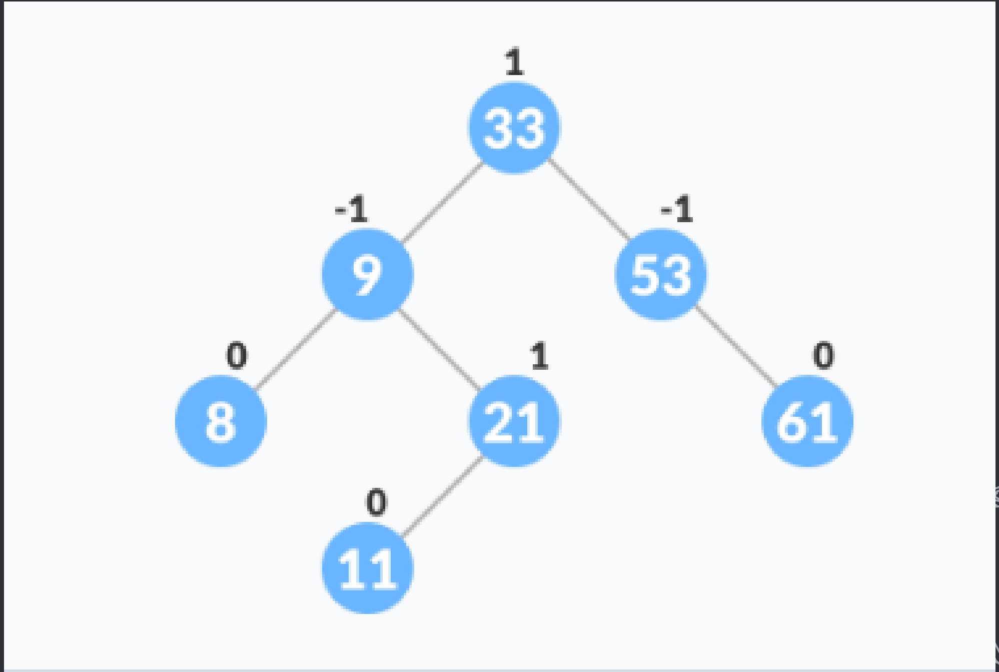
B. If balanceFactor of currentNode < -1,

a. If balanceFactor of rightChild <= 0, do left rotation.

b. Else do right-left rotation.

Algorithm to Delete a node

- 5. The final tree is:



Avl tree final

Team Presentation

Chavez, John Michael
Gabinete, Roland Paul
Sababoro, Maylien
Mayordo, Rex
Lapinid, Janferson
Ramirez, Dian Gabriel

FOR LISTENING!!