Understanding Trees in Data Structures

We have all watched trees from our childhood. It has roots, stems, branches and leaves. It was observed long back that each leaf of a tree can be traced to root via a unique path. Hence tree structure was used to explain hierarchical relationships, e.g. family tree, animal kingdom classification, etc.

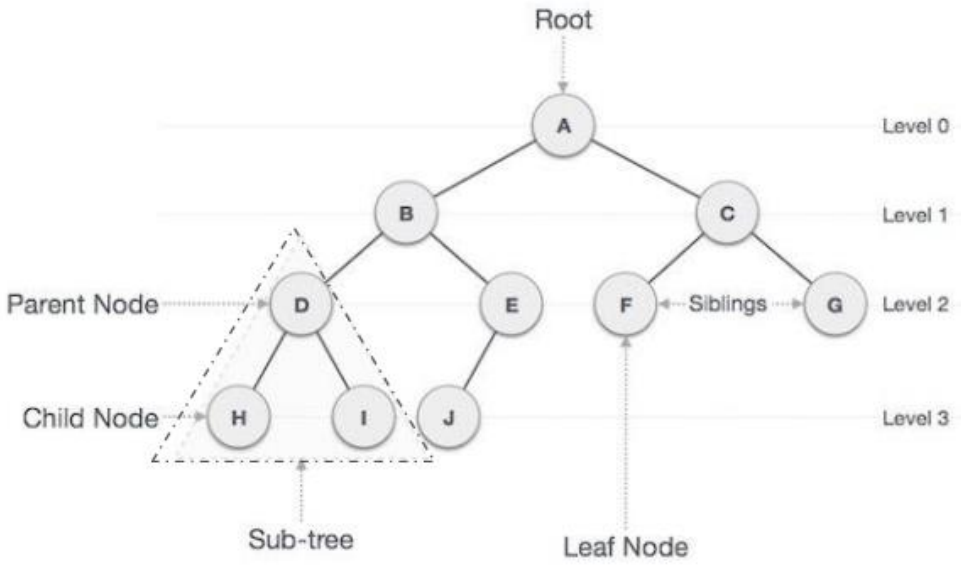
This hierarchical structure of trees is used in Computer science as an abstract data type for various applications like data storage, search and sort algorithms. Let us explore this data type in detail.

Tree Terminology

A tree is a hierarchical data structure defined as a collection of nodes. Nodes represent value and nodes are connected by edges. A tree has the following properties:

1. The tree has one node called root. The tree originates from this, and hence it does not have any parent.
2. Each node has one parent only but can have multiple children.
3. Each node is connected to its children via edge.

Following diagram explains various terminologies used in a tree structure.



Terminology	**Description**	**Example From Diagram**
Root	Root is a special node in a tree. The entire tree originates from it. It does not have a parent.	Node A
Parent Node	Parent node is an immediate predecessor of a node.	B is parent of D & E
Child Node	All immediate successors of a node are its children.	D & E are children of B
Leaf	Node which does not have any child is called as leaf	H, I, J, F and G are leaf nodes
Edge	Edge is a connection between one node to another. It is a line between two nodes or a node and a leaf.	Line between A & B is edge

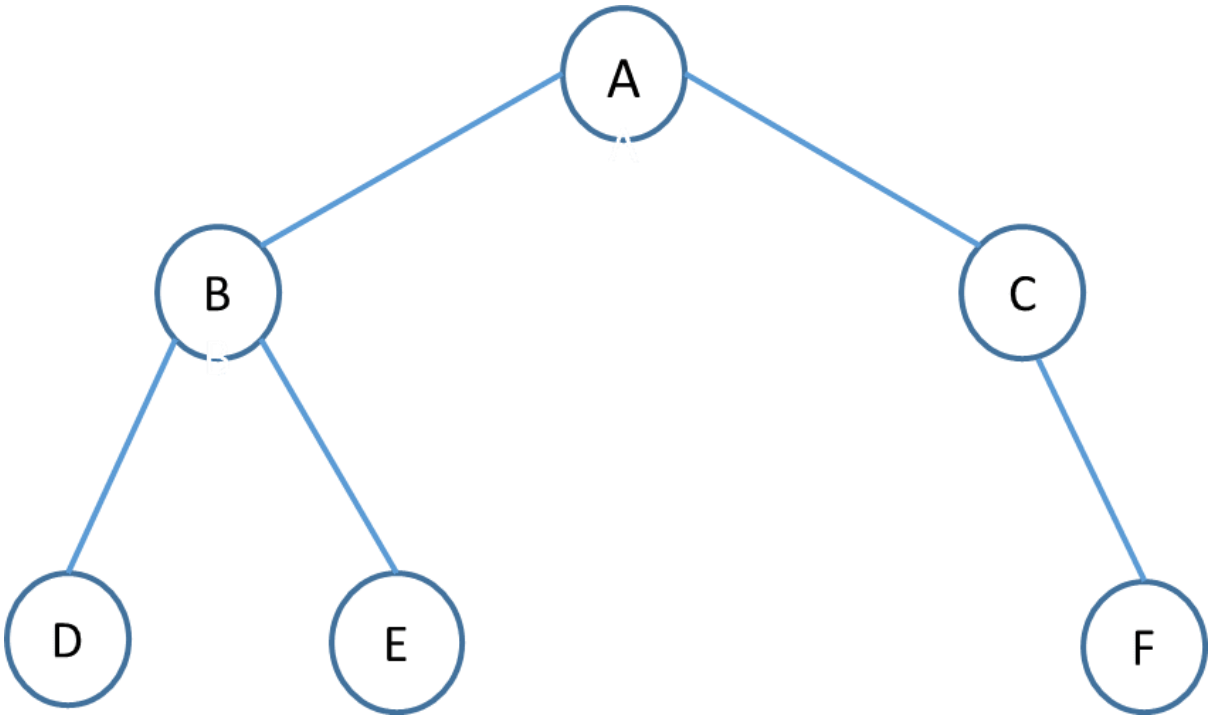
Siblings	Nodes with the same parent are called Siblings.	D & E are siblings
Path / Traversing	Path is a number of successive edges from source node to destination node.	A – B – E – J is path from node A to J
Height of Node	Height of a node represents the number of edges on the longest path between that node and a leaf.	A, B, C, D & E can have height. Height of A is no. of edges between A and H, as that is the longest path, which is 3. Height of C is 1
Levels of node	Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on	Level of H, I & J is 3. Level of D, E, F & G is 2
Degree of Node	Degree of a node represents the maximum number of children of a node.	Degree of D is 2 and of E is 1
Sub tree	Descendants of a node represent subtree.	Nodes D, H, I represent one subtree.

Types of Trees

Types of trees depend on the number of children a node has. There are two major tree types:

- General Tree: A tree in which there is no restriction on the number of children a node has, is called a General tree. Examples are Family tree, Folder Structure.

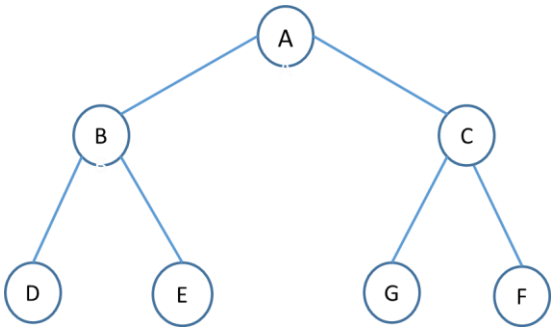
- Binary Tree: In a Binary tree, every node can have at most 2 children, left and right. In diagram below, B & D are left children and C, E & F are right children.



Binary trees are further divided into many types based on its application.

- Full Binary Tree: If every node in a tree has either 0 or 2 children, then the tree is called a full tree. The tree in the above diagram is **not a full** binary tree as node C has only the right child.

- Perfect Binary tree: It is a binary tree in which all interior nodes have two children and all leaves have the same depth or same level.



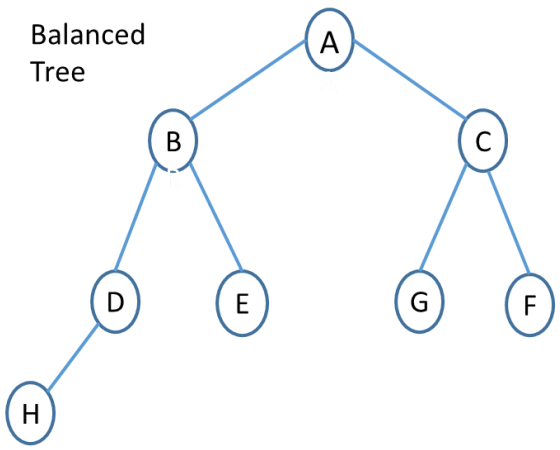
In perfect full binary tree,

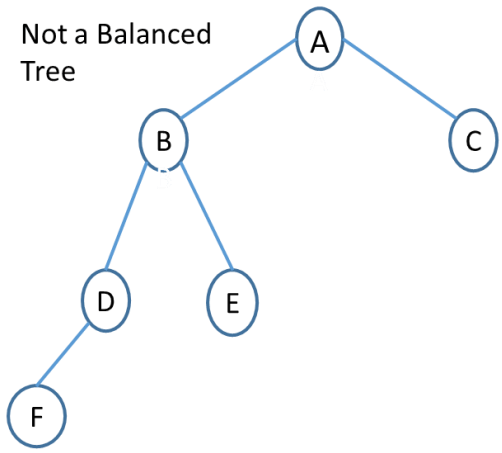
leaf nodes(l) = 2^h and

total number of nodes(n) = $2^{h+1} - 1$

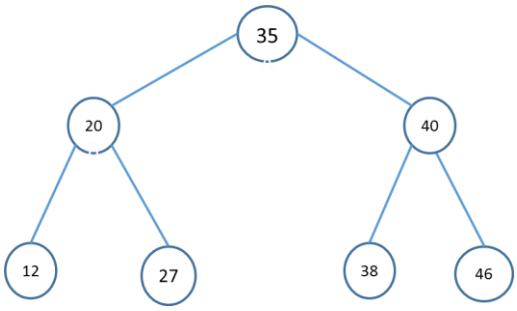
where, n is number of nodes, h is height of tree and l is number of leaf nodes. In the above diagram, h is 2 so leaves will be 4 and nodes will $2^3 - 1$ which is 7.

- Balanced Tree: If the height of the left and right subtree at any node differs at most by 1, then the tree is called a balanced tree.





- Binary Search Tree: It is a binary tree with binary search property. Binary search property states that the value or key of the left node is less than its parent and value or key of right node is greater than its parent. And this is true for all nodes.



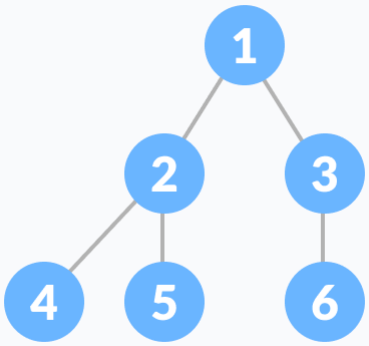
Binary search trees are used in various searching and sorting algorithms. There are many variants of binary search trees like AVL tree, B-Tree, Red-black tree, etc.

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.

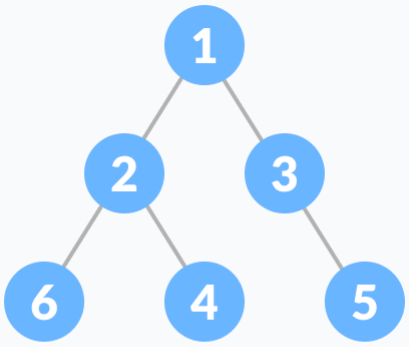
A complete binary tree is just like a full binary tree, but with two major differences

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.



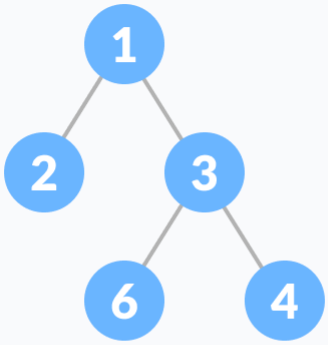
Complete Binary Tree

Full Binary Tree vs Complete Binary Tree



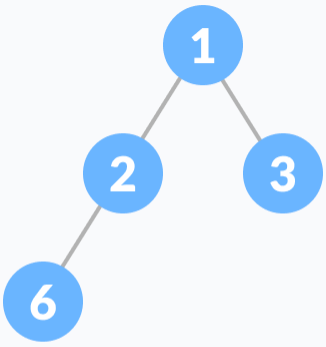
✖ Full Binary Tree
✖ Complete Binary Tree

Comparison between full binary tree and complete binary tree



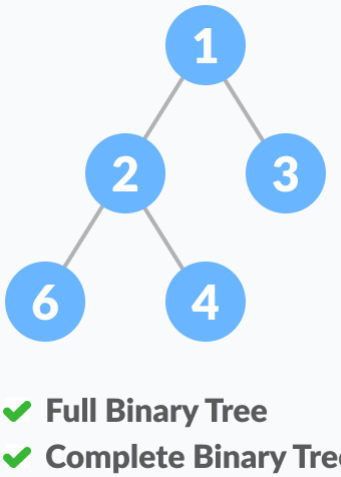
✔ Full Binary Tree
✖ Complete Binary Tree

Comparison between full binary tree and complete binary tree



✖ Full Binary Tree
✔ Complete Binary Tree

Comparison between full binary tree and complete binary tree

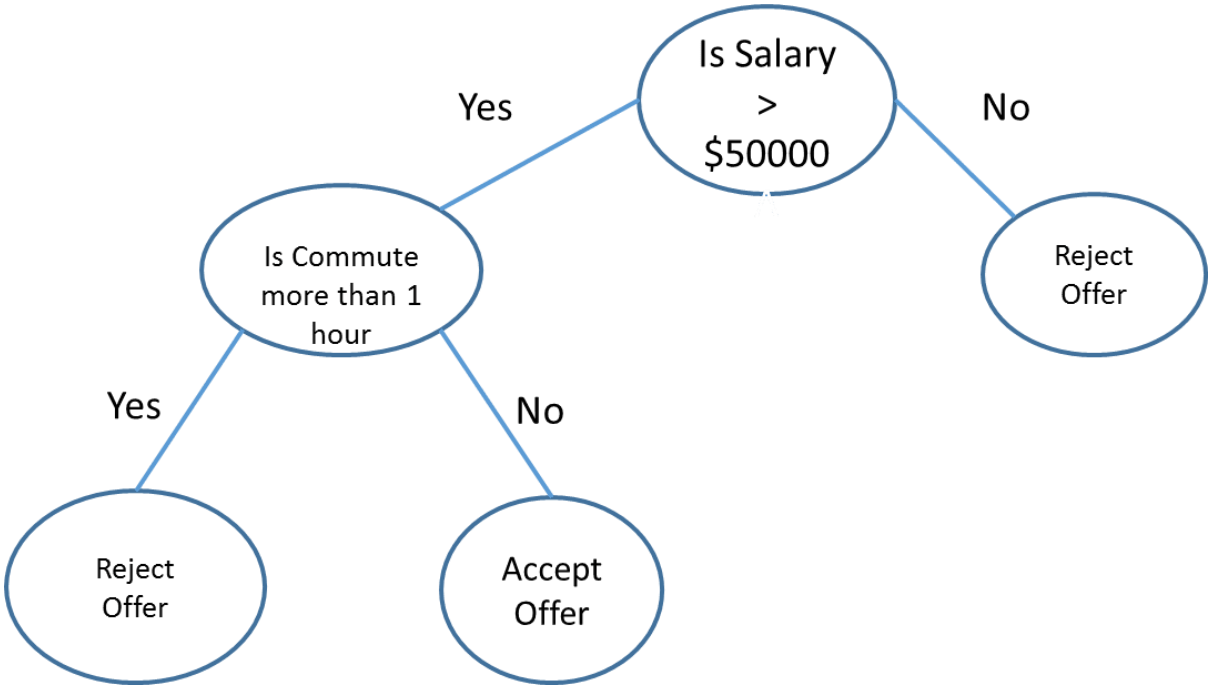


Comparison between full binary tree and complete binary tree

and complete binary tree

Trees in Data Science

A Tree structure is used in predictive modelling. It is usually called a Decision tree. In the Decision tree, each internal node represents a test or condition on a predictive variable, and edge gives various possible answers to this test. Leaf node gives the outcome of all tests on a path. Decision tree for accepting or rejecting job offer is as follows:

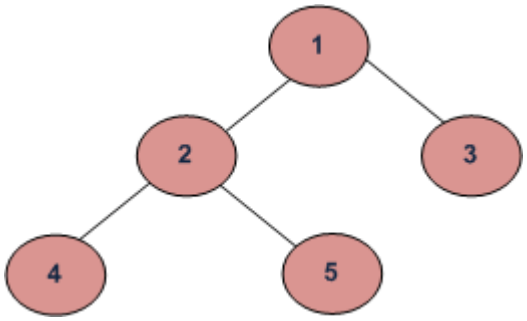


Deciding whether to accept or reject a job is based on two parameters: salary and commute time. Nodes specify conditions on which these two parameters are tested. The priority of the condition depends on how close it is to root. The most affecting parameter is tested first, in this case, salary. So that is the 1st split at the root. Then the next relevant condition. Hence, decision trees not only help in finding the decisions, but it also does in the fastest way.

There are many algorithms like CART (Classification And Regression Tree), Random forest, which helps in building models.

Tree Traversals (Inorder, Preorder and Postorder)

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.



Depth First Traversals:
(a) Inorder (Left, Root, Right) : 4 2 5 1 3
(b) Preorder (Root, Left, Right) : 1 2 4 5 3
(c) Postorder (Left, Right, Root) : 4 5 2 3 1

Inorder Traversal

```
Algorithm Inorder(tree)
```

1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

Uses of Inorder
In the case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal s reversed can be used.
Example: In order traversal for the above-given figure is 4 2 5 1 3.

Preorder Traversal

```
Algorithm Preorder(tree)
```

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

Uses of Preorder
Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on an expression tree.
Please see http://en.wikipedia.org/wiki/Polish_notation know why prefix expressions are useful.

Example: Preorder traversal for the above-given figure is 1 2 4 5 3.

Postorder Traversal:

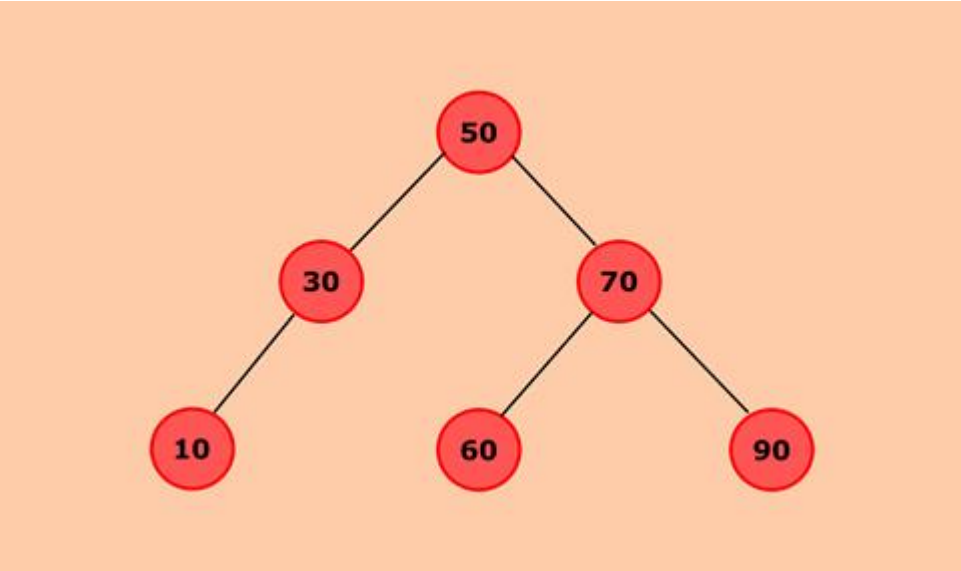
```
Algorithm Postorder(tree)
```

1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

Uses of Postorder
Postorder traversal is used to delete the tree. Postorder traversal is also useful to get the postfix expression of an expression tree. Please see http://en.wikipedia.org/wiki/Reverse_Polish_notation for the usage of postfix expression.
Example: Postorder traversal for the above-given figure is 4 5 2 3 1.

Cconstruct a Binary Search Tree and perform deletion and In-order traversal

In this program, we need to create a binary search tree, delete a node from the tree, and display the nodes of the tree by traversing the tree using in-order traversal. In in-order traversal, for a given node, first, we traverse the left child then root then right child (**Left -> Root -> Right**).



In **Binary Search Tree**, all nodes which are present to the left of root will be less than root node and nodes which are present to the right will be greater than the root node.

Insertion:

- If the value of the new node is less than the root node then, it will be inserted to the left subtree.
- If the value of the new node is greater than root node then, it will be inserted to the right subtree.

Deletion:

- If the node to be deleted is a leaf node then, parent of that node will point to null. For eg. If we delete 90, then parent node 70 will point to null.
- If the node to be deleted has one child node, then child node will become a child node of the parent node. For eg. If we delete 30, then node 10 which was left child of 30 will become left child of 50.
- If the node to be deleted has two children then, we find the node(minNode) with minimum value from the right subtree of that current node. The current node will be replaced by its successor(minNode).

Algorithm

- Define Node class which has three attributes namely: **data**, **left** and **right**. Here, left represents the left child of the node and right represents the right child of the node.
- When a node is created, data will pass to the data attribute of the node and both left and right will be set to null.
- Define another class which has an attribute root.
    - Root represents the root node of the tree and initializes it to null.

a. insert() will insert the new value into a binary search tree:

- It checks whether root is null, which means tree is empty. New node will become root node of tree.
- If tree is not empty, it will compare value of new node with root node. If value of new node is greater than root, new node will be inserted to right subtree. Else, it will be inserted in left subtree.

a. deleteNode() will delete a particular node from the tree:

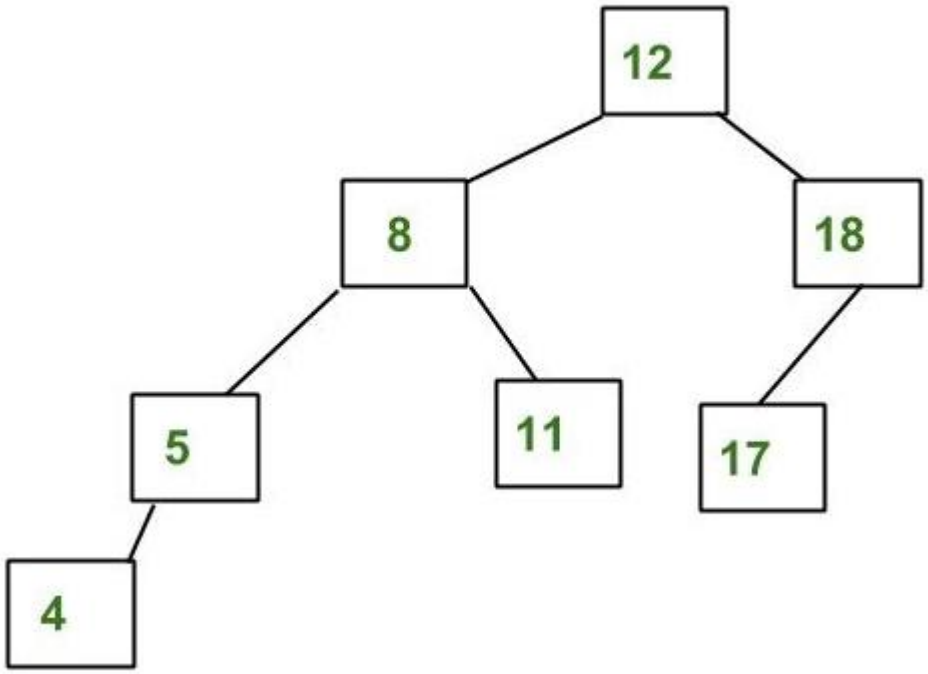
- If value of node to be deleted is less than root node, search node in left subtree. Else, search in right subtree.

- If node is found and it has no children, then set the node to null.
- If node has one child then, child node will take position of node.
- If node has two children then, find a minimum value node from its right subtree. This minimum value node will replace the current node.

AVL Tree

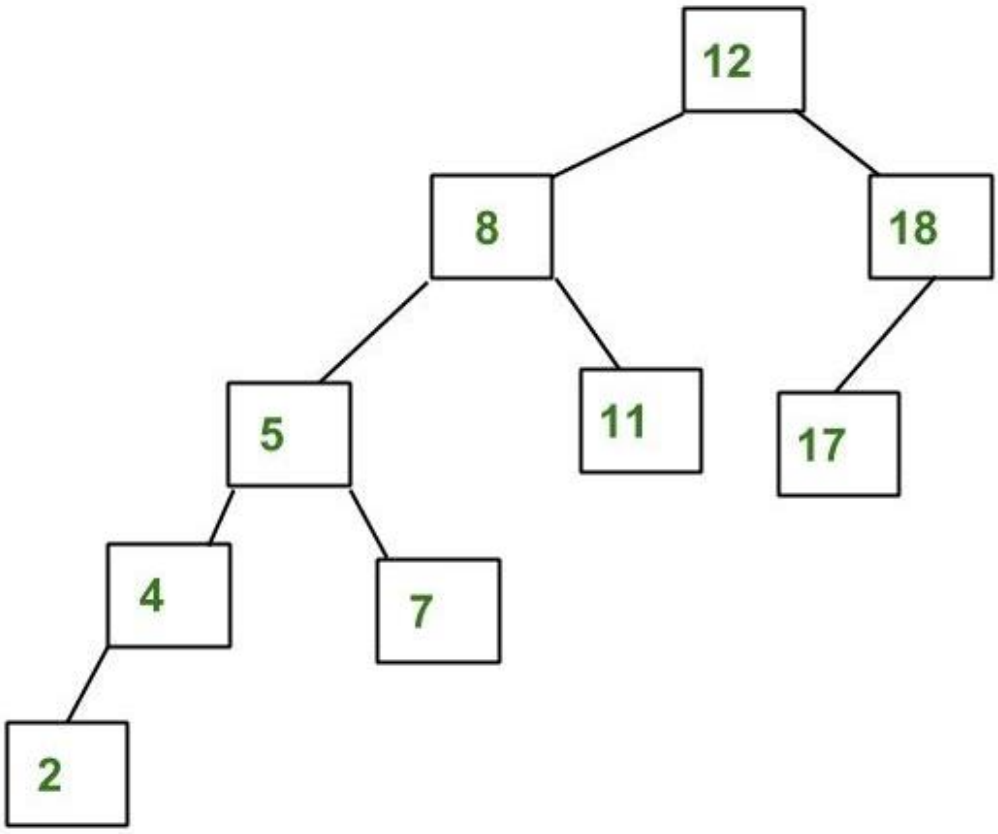
AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

An Example Tree that is an AVL Tree



The above tree is AVL because differences between heights of left and right subtrees for every node is less than or equal to 1.

An Example Tree that is NOT an AVL Tree



The above tree is not AVL because differences between heights of left and right subtrees for 8 and 12 is greater than 1.

Why AVL Trees?
Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that height of the tree remains O(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations. The height of an AVL tree is always O(Logn) where n is the number of nodes in the tree

Insertion
To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).
1) Left Rotation
2) Right Rotation

```
T1, T2 and T3 are subtrees of the tree

rooted with y (on the left side) or x (on

the right side)
```

```
     y                               x
    / \     Right Rotation          /  \
   x   T3   - - - - - - - - >      T1   y
  / \       < - - - - - - - -          / \
 T1  T2     Left Rotation            T2  T3
```

Keys in both of the above trees follow the

following order

keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)

So BST property is not violated anywhere.

Steps to follow for insertion

Let the newly inserted node be w

1) Perform standard BST insert for w.

2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.

3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:

a) y is left child of z and x is left child of y (Left Left Case)

b) y is left child of z and x is right child of y (Left Right Case)

c) y is right child of z and x is right child of y (Right Right Case)

d) y is right child of z and x is left child of y (Right Left Case)

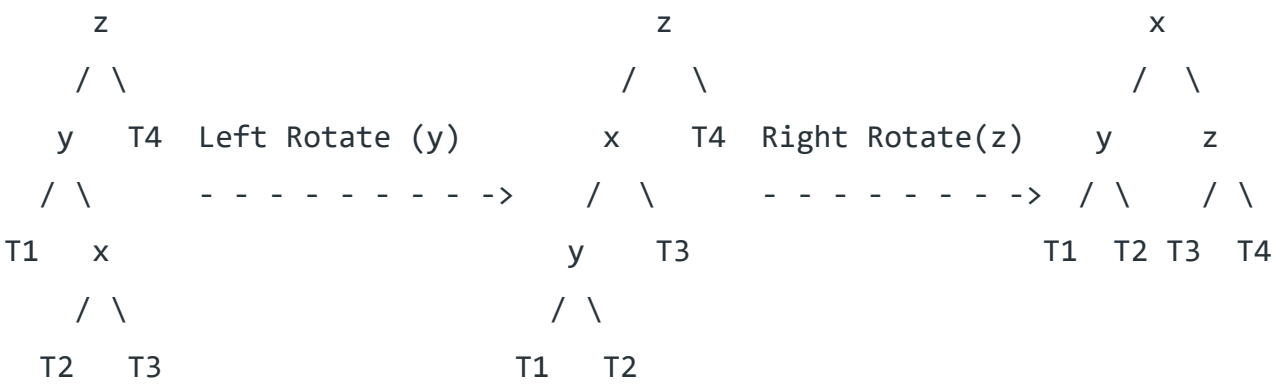
Following are the operations to be performed in above mentioned 4 cases. In all of the cases, we only need to re-balance the subtree rooted with z and the complete tree becomes balanced as the height of subtree (After appropriate rotations) rooted with z becomes same as it was before insertion.

a) Left Left Case

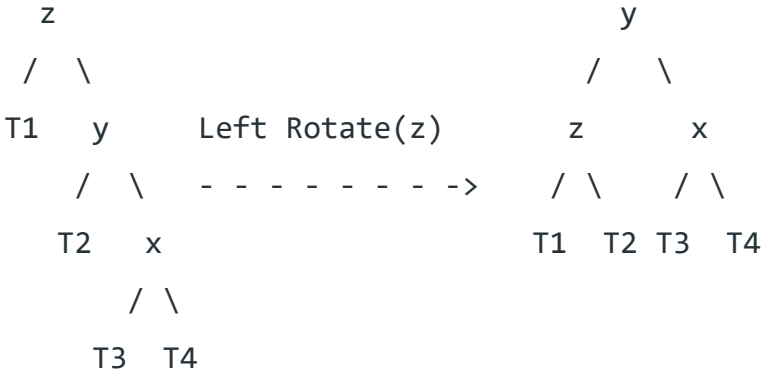
T1, T2, T3 and T4 are subtrees.

```
        z                                      y
       / \                                    /   \
      y   T4      Right Rotate (z)           x      z
     / \          - - - - - - - - ->        /  \    /  \
    x   T3                                T1  T2  T3  T4
   / \
 T1   T2
```

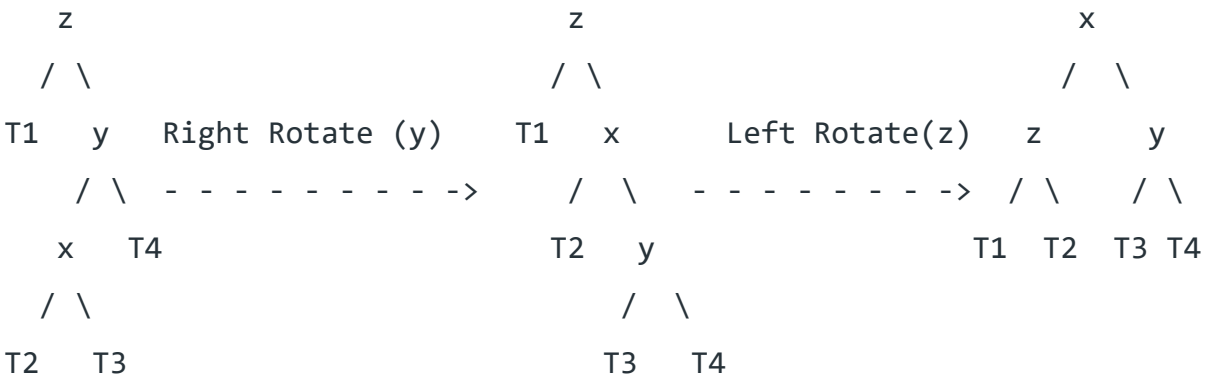
b) Left Right Case



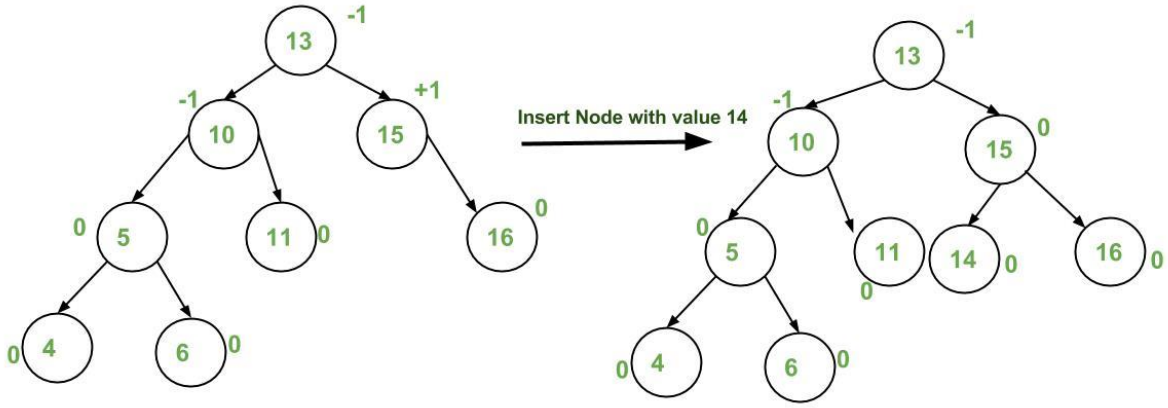
c) Right Right Case

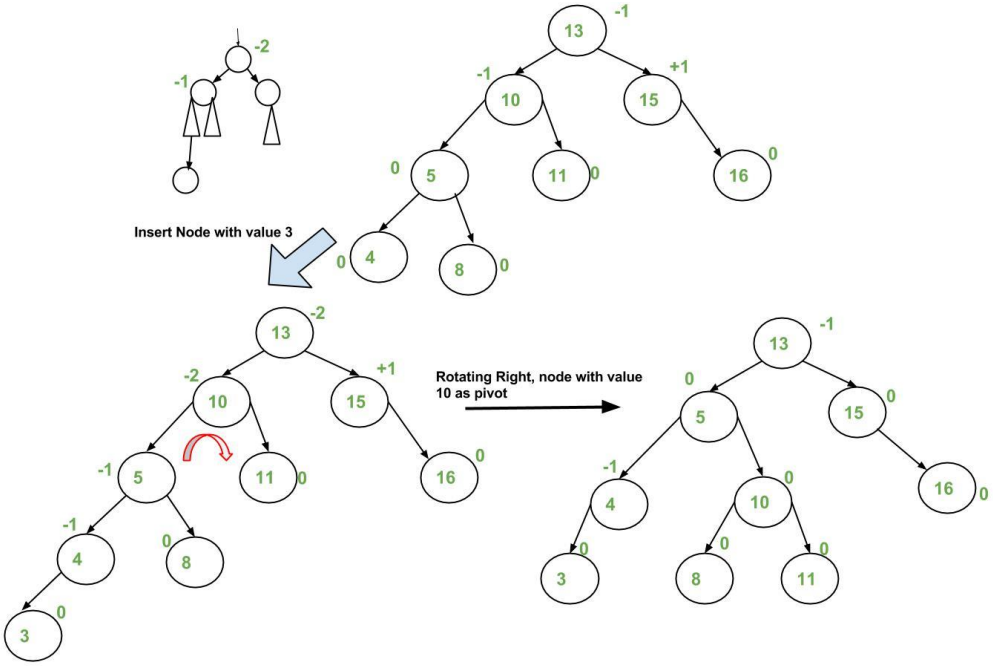


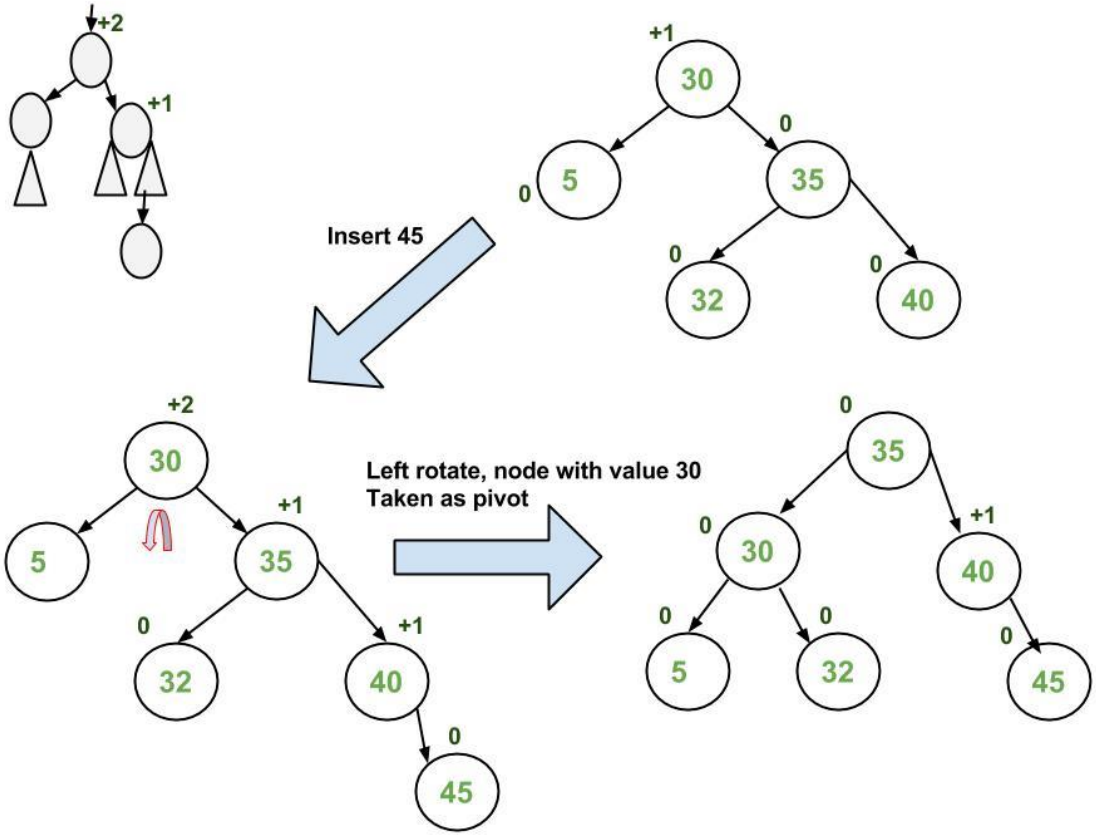
d) Right Left Case

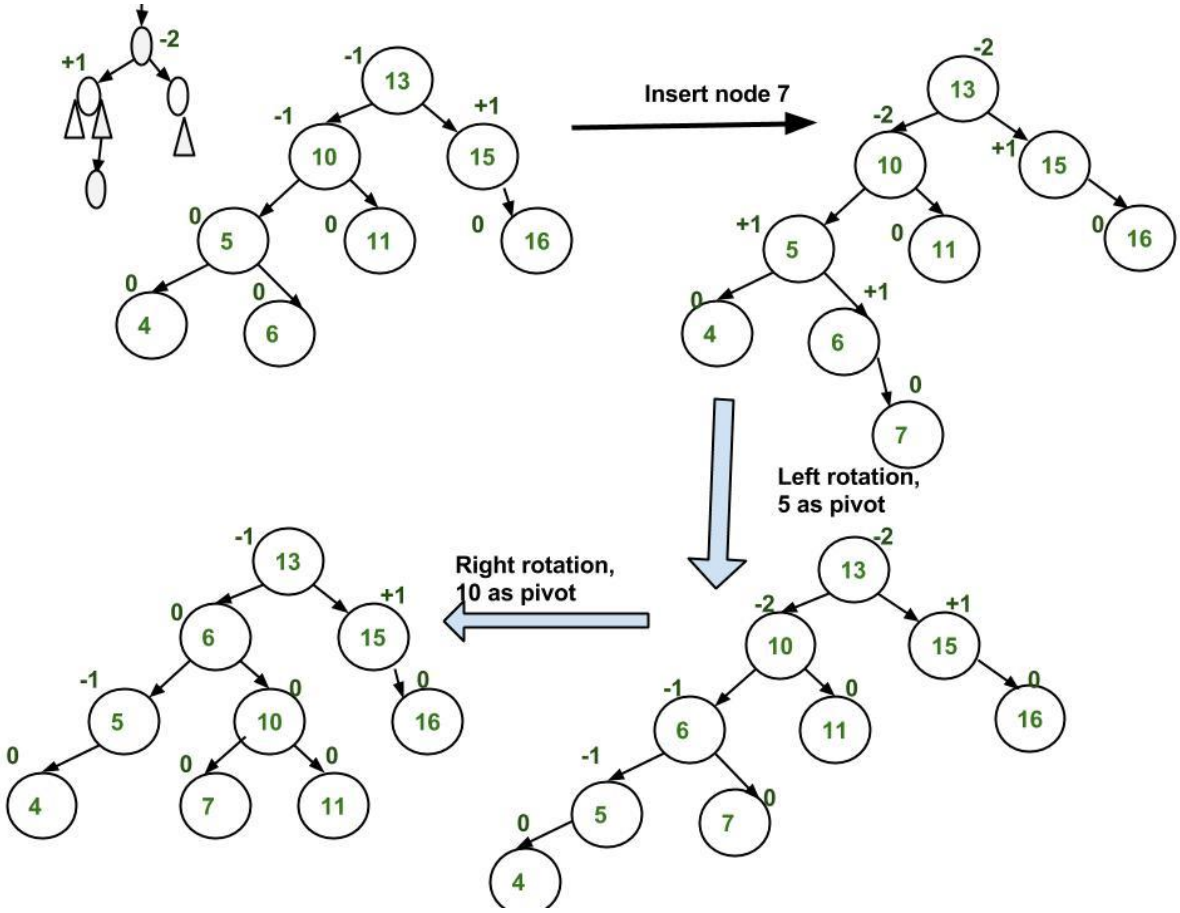


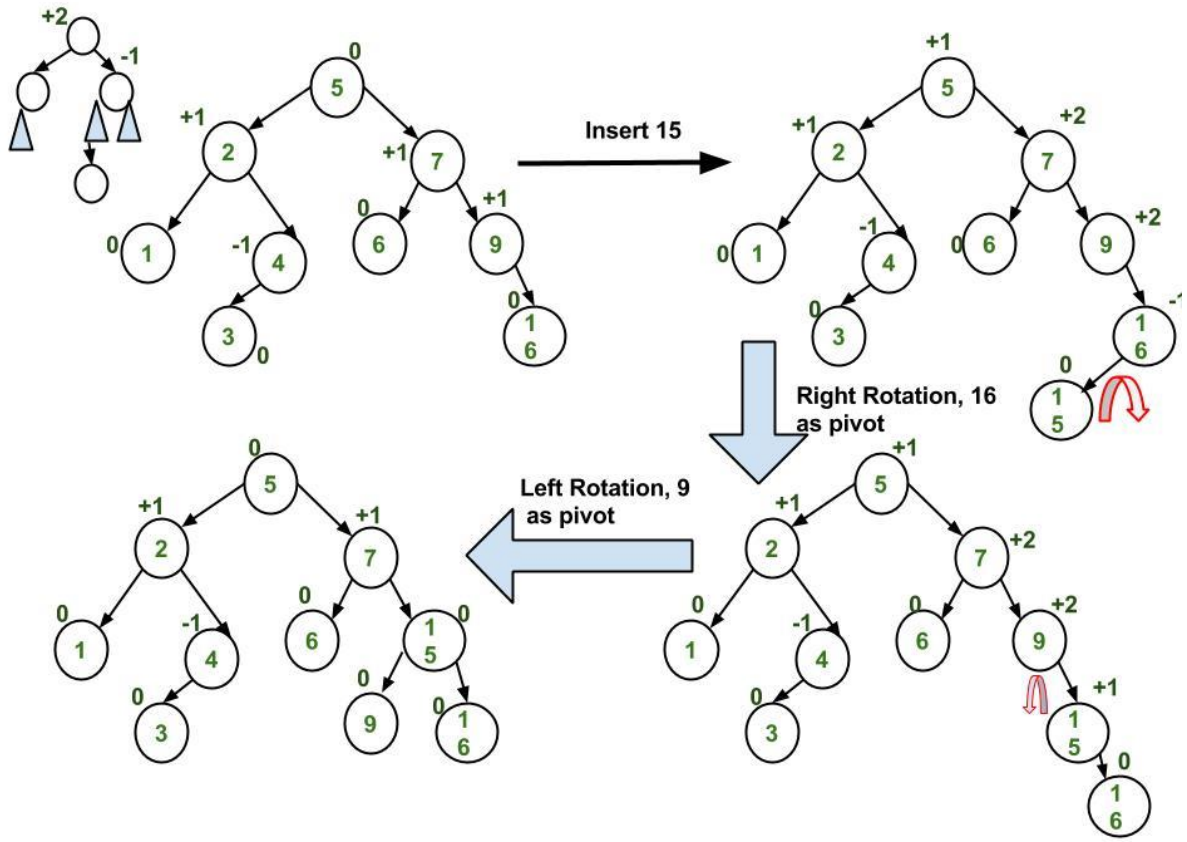
Insertion Examples:


13
-1
10
-1
15
+1
5
0
11
0
16
0
4
0
6
0
Insert Node with value 14
13
-1
10
-1
15
0
5
0
11
0
14
0
16
0
4
0
6
0


-2
-1
13
-1
10
-1
15
+1
5
0
11
0
16
0
4
0
8
0
Insert Node with value 3
13
-2
10
-2
15
+1
5
-1
11
0
16
0
4
-1
8
0
3
0
Rotating Right, node with value
10 as pivot
13
-1
5
0
15
0
4
-1
10
0
16
0
3
0
8
0
11
0


+2
+1
Insert 45
+1
30
0
5
35
0
0
32
40
0
+2
30
5
+1
35
0
32
+1
40
0
45
Left rotate, node with value 30
Taken as pivot
0
35
0
30
+1
40
0
0
5
32
0
45


Insert node 7
Left rotation,
5 as pivot
Right rotation,
10 as pivot



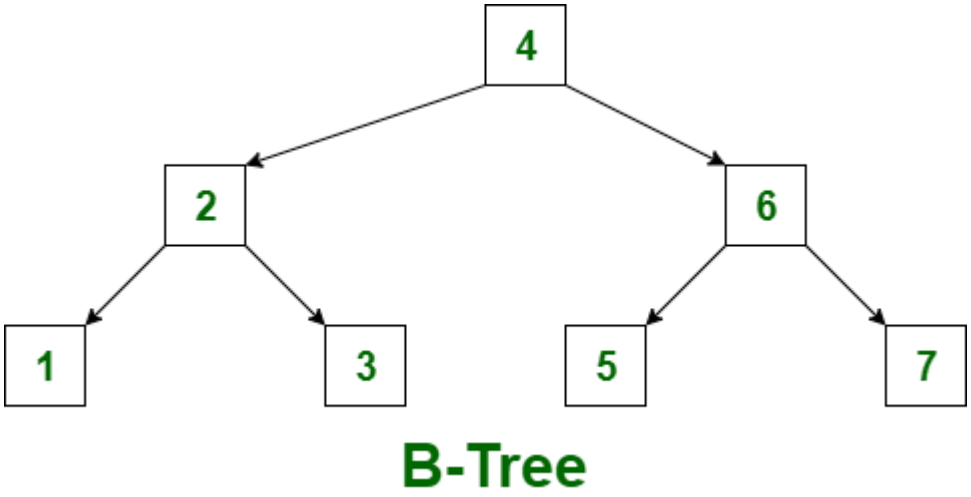
Difference between B tree and B+ tree

B-Tree:

B-Tree is known as a self-balancing tree as its nodes are sorted in the inorder traversal. In B-tree, a node can have more than two children. B-tree has a height of logM N (Where 'M' is the order of tree and N is the number of nodes). And the height is adjusted automatically at each update. In the B-tree data is sorted in a specific order, with the lowest value on the left and the highest value on the right. To insert the data or key in B-tree is more complicated than a binary tree.

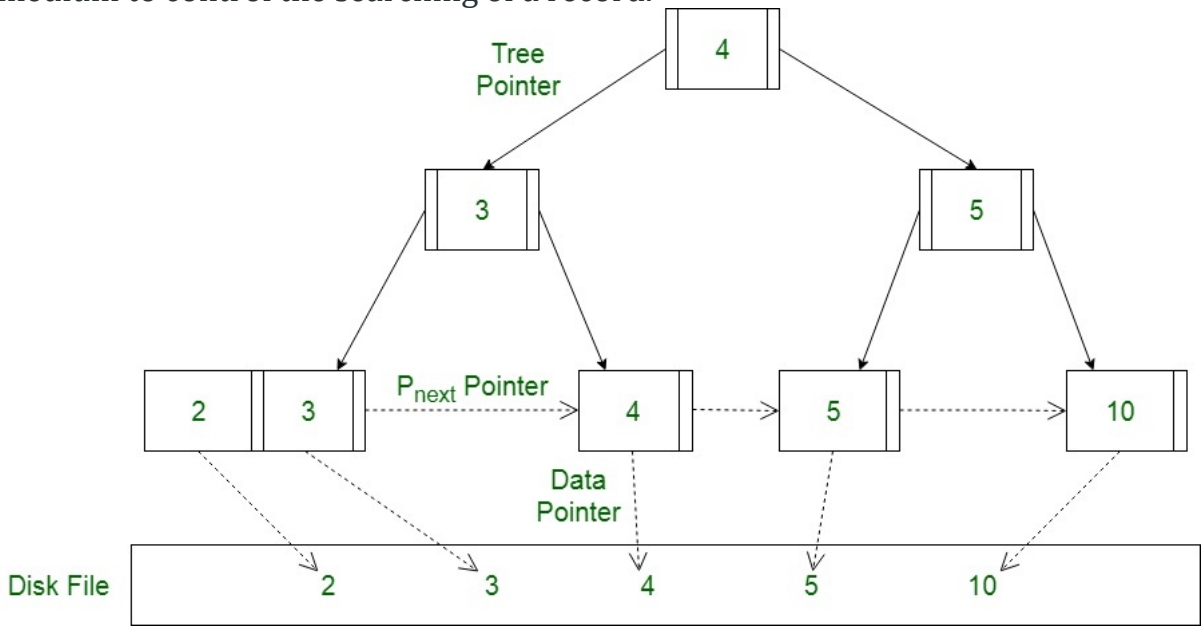
There are some conditions that must be hold by the B-Tree:

- All the leaf nodes of the B-tree must be at the same level.
- Above the leaf nodes of the B-tree, there should be no empty sub-trees.
- B- tree's height should lie as low as possible.



B+ Tree

B+ tree eliminates the drawback B-tree used for indexing by storing data pointers only at the leaf nodes of the tree. Thus, the structure of leaf nodes of a B+ tree is quite different from the structure of internal nodes of the B tree. It may be noted here that, since data pointers are present only at the leaf nodes, the leaf nodes must necessarily store all the key values along with their corresponding data pointers to the disk file block, in order to access them. Moreover, the leaf nodes are linked to providing ordered access to the records. The leaf nodes, therefore form the first level of the index, with the internal nodes forming the other levels of a multilevel index. Some of the key values of the leaf nodes also appear in the internal nodes, to simply act as a medium to control the searching of a record.



Difference between B-tree and B+ tree:

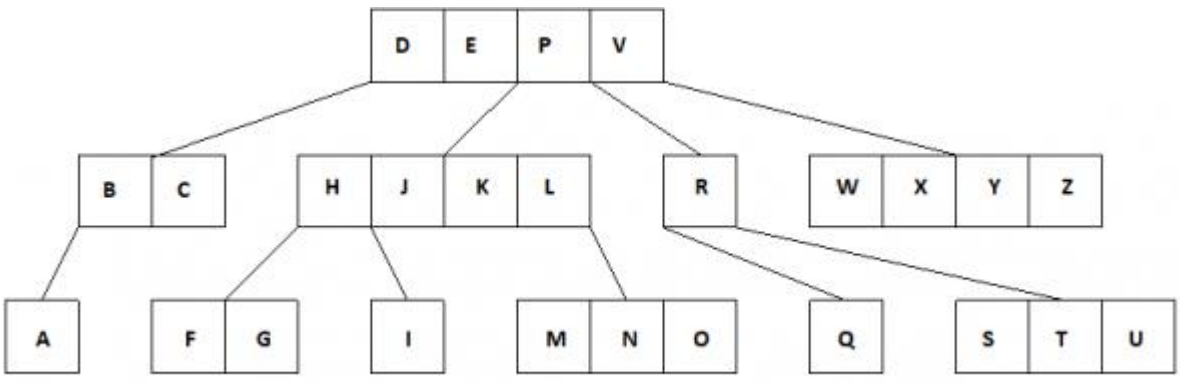
S.NO	B tree	B+ tree
1.	All internal and leaf nodes have data pointers	Only leaf nodes have data pointers
2.	Since all keys are not available at leaf, search often takes more time.	All keys are at leaf nodes, hence search is faster and accurate..
3.	No duplicate of keys is maintained in the tree.	Duplicate of keys are maintained and all nodes are present at leaf.
4.	Insertion takes more time and it is not predictable sometimes.	Insertion is easier and the results are always the same.
5.	Deletion of internal node is very complex and tree has to undergo lot of transformations.	Deletion of any node is easy because all node are found at leaf.
6.	Leaf nodes are not stored as structural linked list.	Leaf nodes are stored as structural linked list.
7.	No redundant search keys are present..	Redundant search keys may be present..

MultiWay tree

A multiway tree is defined as a tree that can have more than two children. If a multiway tree can have maximum m children, then this tree is called as multiway tree of order m (or an m-way tree).

As with the other trees that have been studied, the nodes in an m-way tree will be made up of m-1 key fields and pointers to children.

multiway tree of order 5



To make the processing of m-way trees easier some type of constraint or order will be imposed on the keys within each node, resulting in a multiway search tree of order m (or an m-way search tree). By definition an m-way search tree is a m-way tree in which following condition should be satisfied −

- Each node is associated with m children and m-1 key fields
- The keys in each node are arranged in ascending order.
- The keys in the first j children are less than the j-th key.
- The keys in the last m-j children are higher than the j-th key

Examples for tree traversal,

Design binary search tree, and find preorder, postorder and preorder traversal

30,40,25,22,42,35,36,7,10,27,21,40,45