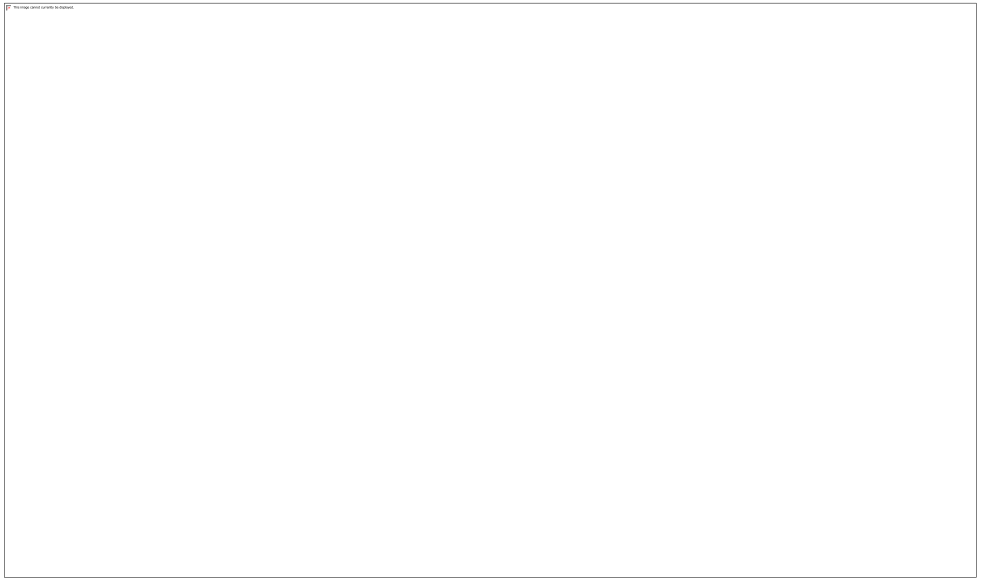
Tree - 2

Ordered Trees

Definition

An **ordered tree is an oriented tree in which the children of a node are somehow "ordered."**

If T1 and T2 are ordered trees then T1 ≠ T2 else T1 = T2.



Types of Ordered Trees

There are several types of ordered trees:

- k-ary tree
- Binomial tree
- Fibonacci tree

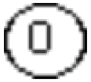
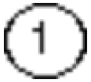
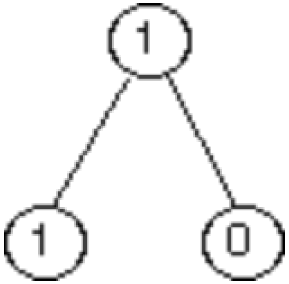
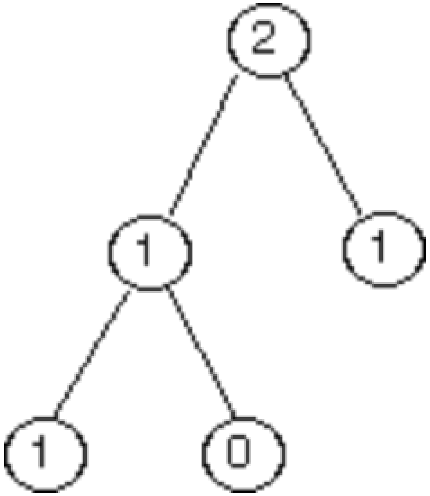
Fibonoacci Trees

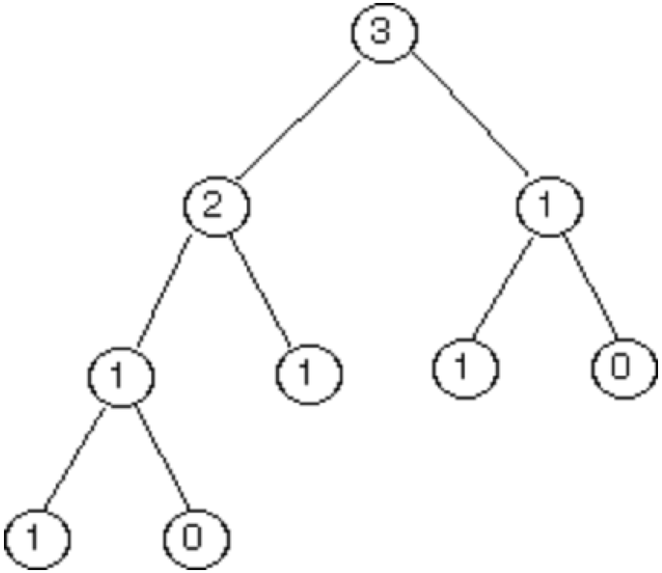
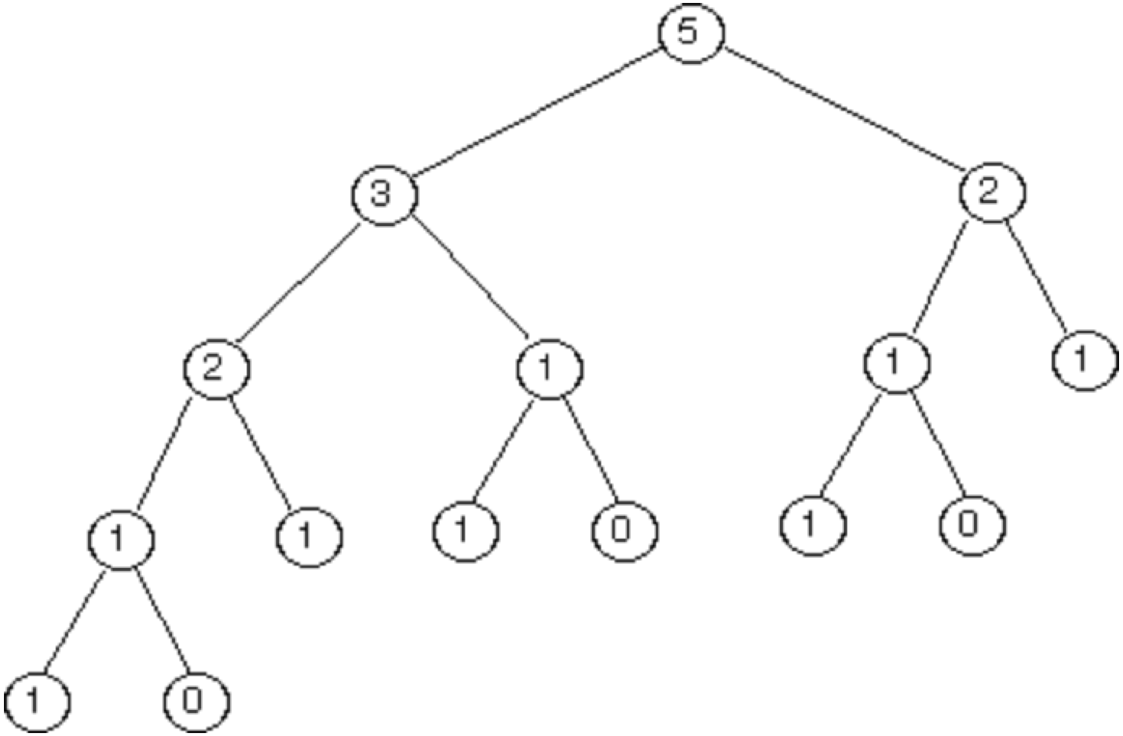
A **Fibonacci Tree** (F_k) is defined by

- F_0 is the empty tree
- F_1 is a tree with only one node
- F_{k+2} is a node whose left subtree is a F_{k+1} tree and whose right subtree is a F_k tree.

The Fibonacci sequence is defined as follows: **F0 = 0, F1 = 1**, and **each subsequent number in the sequence is the sum of the previous two**.

The root of a Fibonacci tree should contain the value of the nth Fibonacci number the left subtree should be the tree representing the computation of the n-1st Fibonacci number, and the right subtree should be the tree representing the computation of the n-2nd Fibonacci number. The first few Fibonacci trees appear below.

Function	Graph
`fibtree(0)`	
`fibtree(1)`	
`fibtree(2)`	
`fibtree(3)`	

fibtree(4)	
fibtree(5)	

Binomial Trees

The **Binomial Tree** (B_k) **consists of a node with k children**. The **first child is the root of a B_{k-1} tree**, the **second is the root of a B_{k-2} tree,** etc.

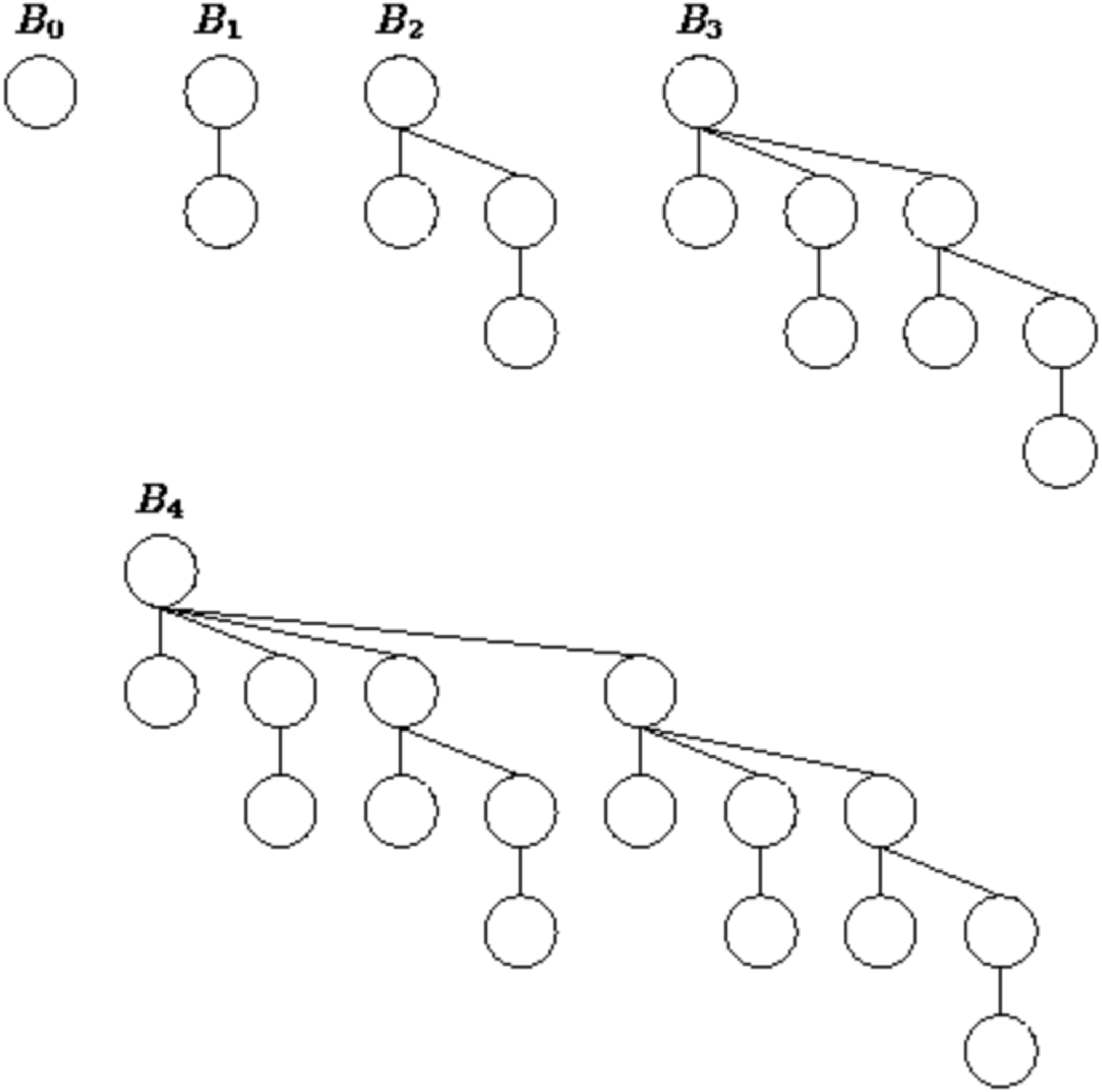
A binomial tree is a general tree with a very special shape:

Definition (Binomial Tree) The *binomial tree of order* with root *R* is the tree defined as follows

If *k*=0, . $B_k = B_0 = \{R\}$. i.e., **the binomial tree of order zero consists of a single node, *R*.**

If *k*>0 $B_k = \{R, B_0, B_1, \ldots, B_{k-1}\}$. i.e., **the binomial tree of order *k*>0 comprises the root *R*, and *k* binomial** $B_0, B_1, \ldots, B_{k-1}$.

Figure shows the **first five binomial trees, B_0 … B_4**. It follows directly from Definition that the **root of B_k** , the **binomial tree of order *k*, has degree *k***. Since *k* **may arbitrarily large, so too can the degree of the root.** Furthermore, the **root of a binomial tree has the largest fanout of any of the nodes in that tree.**

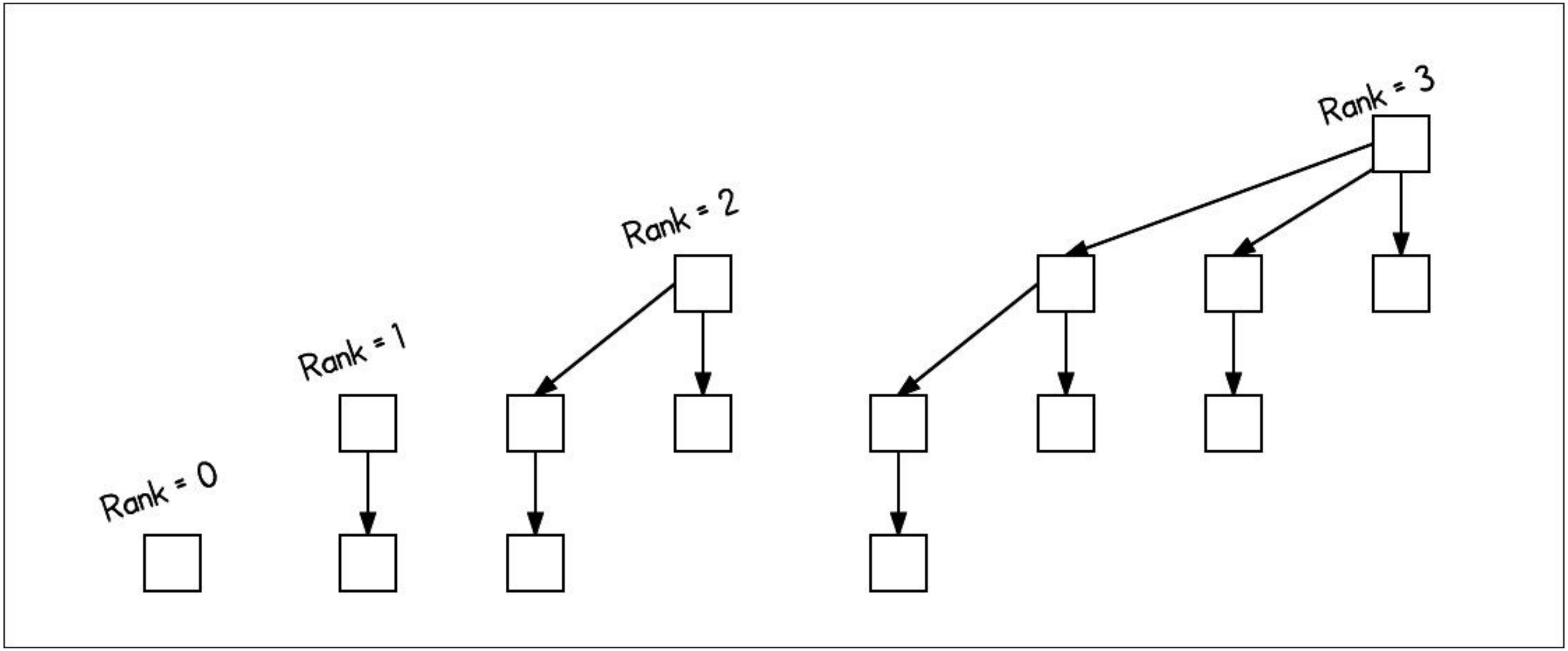

B_0
B_1
B_2
B_3
B_4

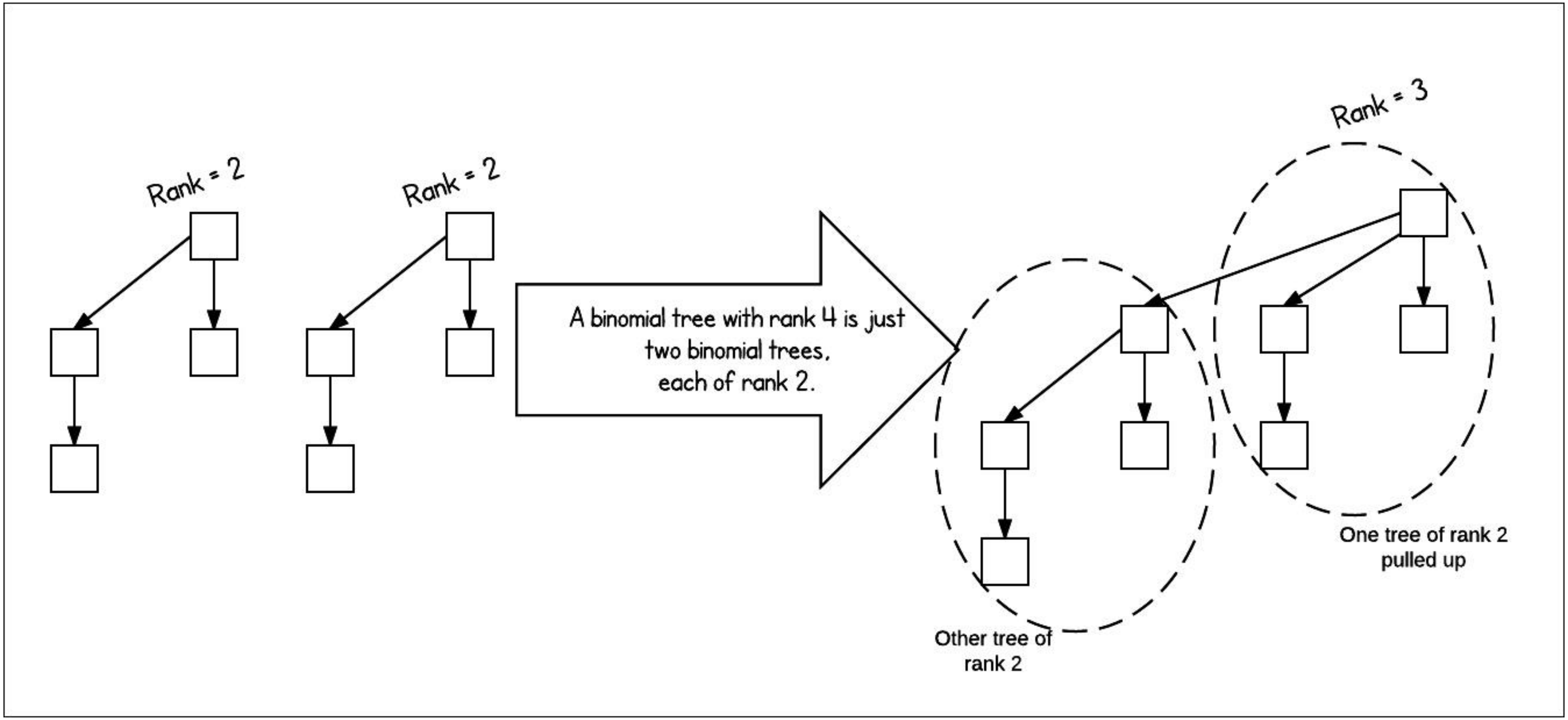
The number of nodes in a binomial tree of order *k* is a function of *k*:

Theorem The binomial tree of order *k*, $\boldsymbol{B_K}$ contains $\boldsymbol{2^K}$ nodes.

The first binomial tree has just one node. Its height is equal to **0**. A binomial tree of height **1** is formed from two binomial trees, each of height **0**. A binomial tree of height **2** is formed from two binomial trees, each of height **1**.

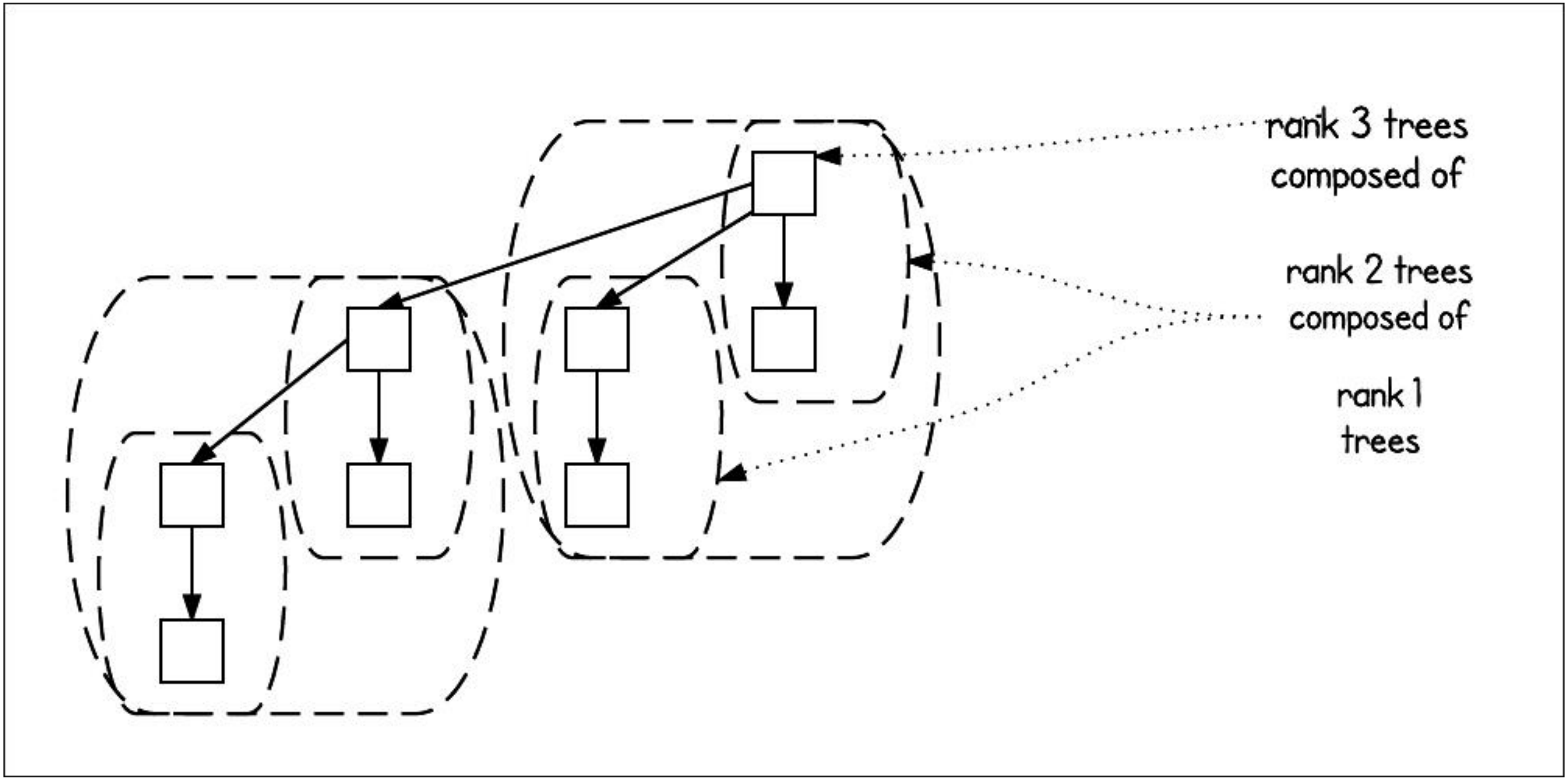
The tree is defined in terms of itself, recursively. For example, the following figure shows two binomial trees of rank **2**. When the right tree is pulled up, the left tree becomes its left child. The resulting tree is of rank 3 with $\mathbf{2^3 = 8}$ nodes.


Rank = 0
Rank = 1
Rank = 2
Rank = 3

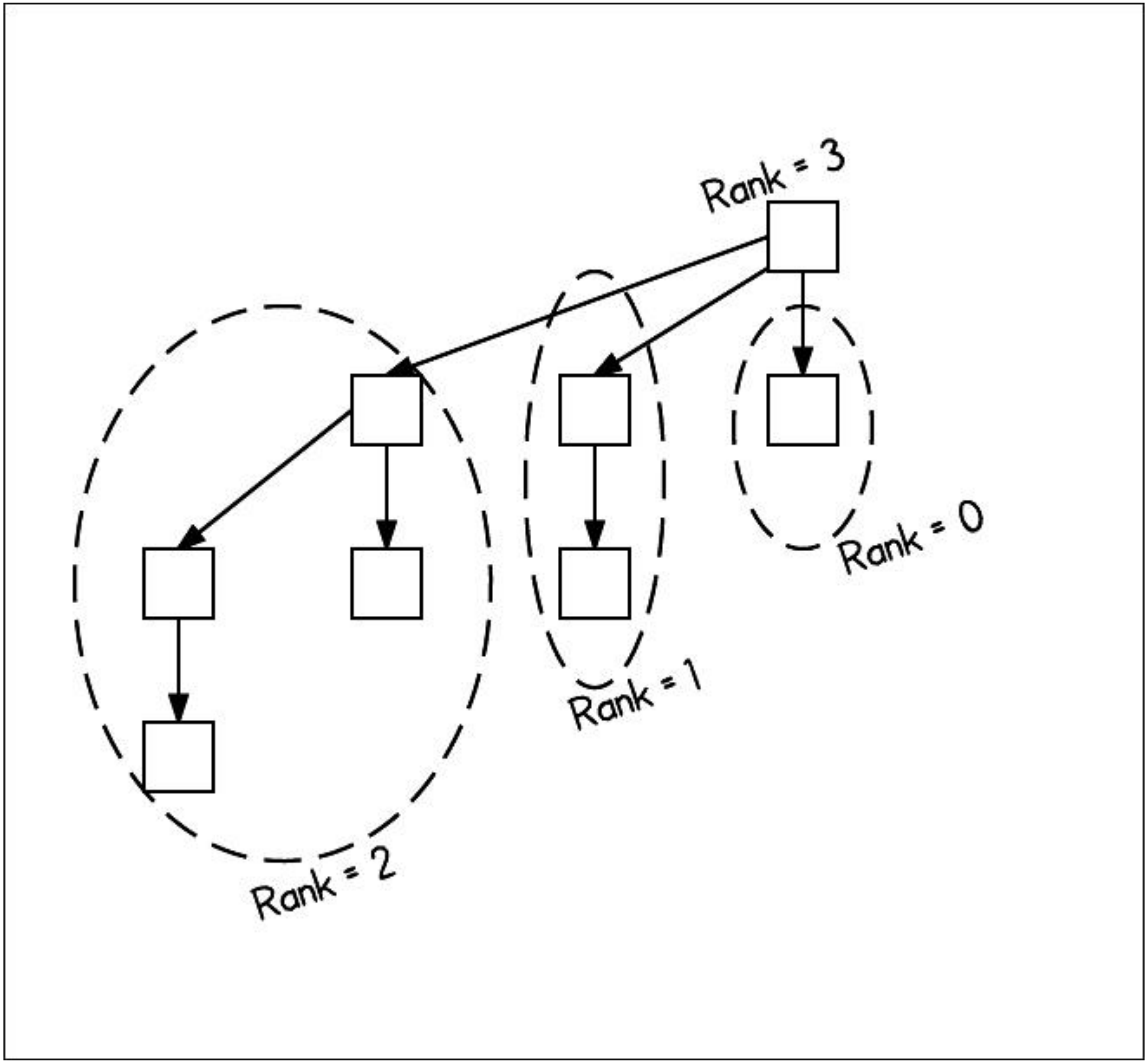

Rank = 2
Rank = 2
A binomial tree with rank 4 is just
two binomial trees,
each of rank 2.
Rank = 3
One tree of rank 2
pulled up
Other tree of
rank 2

The definition of binomial trees could also be stated as follows:

a **binomial tree with rank k is composed of subtrees with rank (k-1)**. Here is a pictorial way of stating this:



You can also look at a binomial tree as a list of binary trees.



k-ary Trees

A **k-ary tree is a tree in which the children of a node appear at distinct index positions in 0..k-1.** *This means the maximum number of children for a node is k*.

Some k-ary trees have special names

2-ary trees are called **binary trees**.

3-ary trees are called **trinary trees** or **ternary trees**.

1-ary trees are called **lists**.

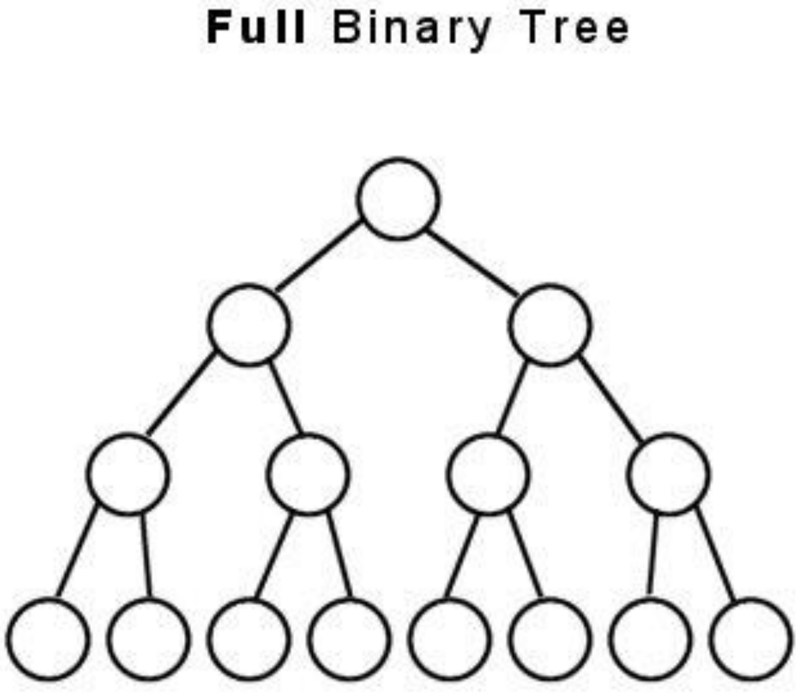
You have to draw k-ary trees carefully. In a binary tree, if a node has one child, you can't draw a vertical line from the parent to the child. Every child in a binary tree is either a left or a right child;

Complete Binary Tree

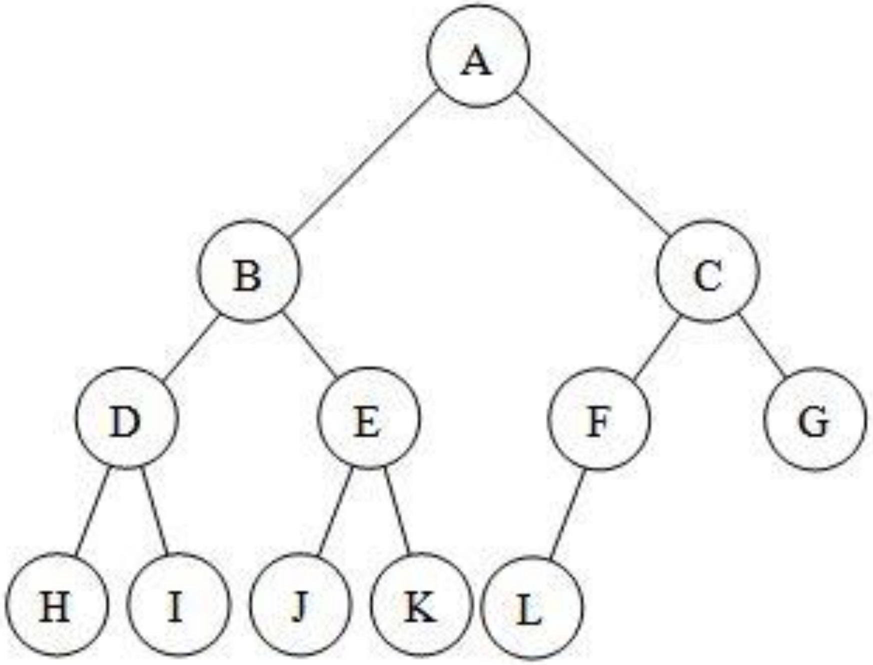
A complete binary tree is a binary tree in which all the levels are completely filled except possibly the lowest one, which is filled from the left.

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

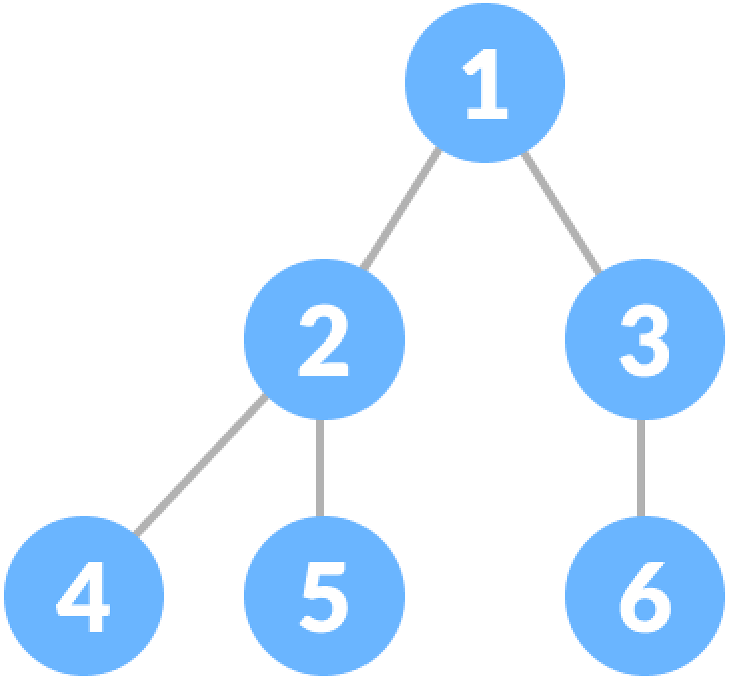
- All the leaf elements must lean towards the left.
- The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.



A **full binary tree** (sometimes proper binary tree or 2-tree) is a tree in which every node other than the leaves has two children.

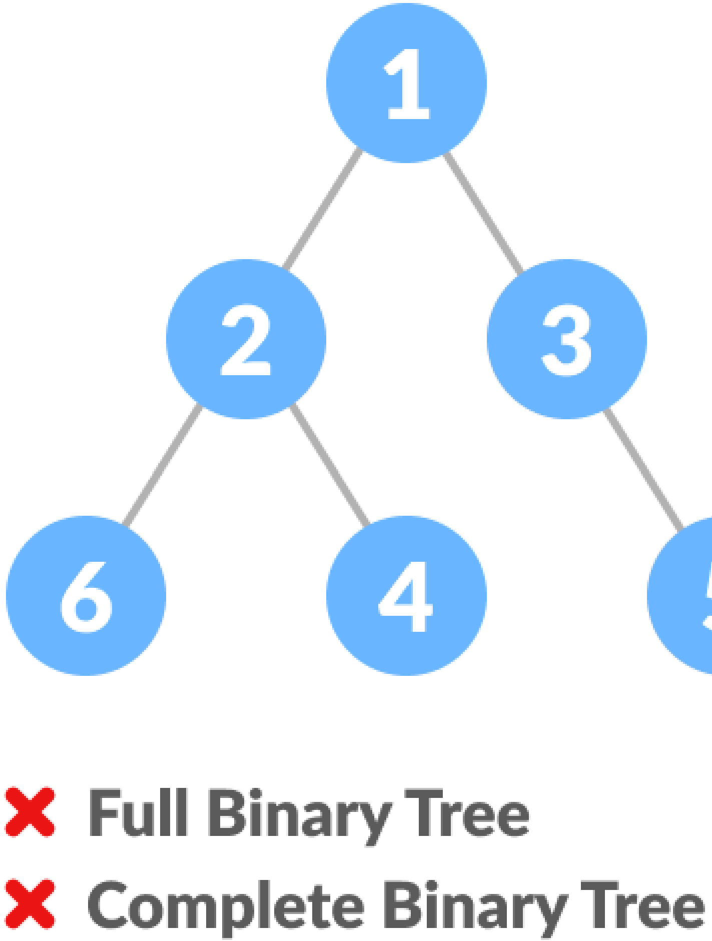


A **complete binary tree** is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

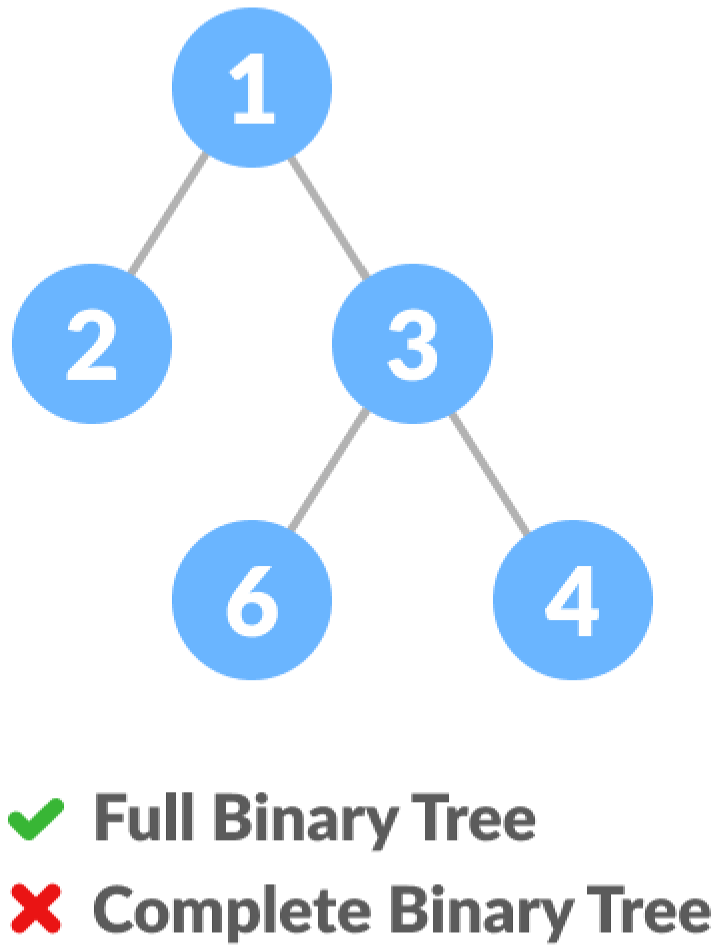


Complete Binary Tree

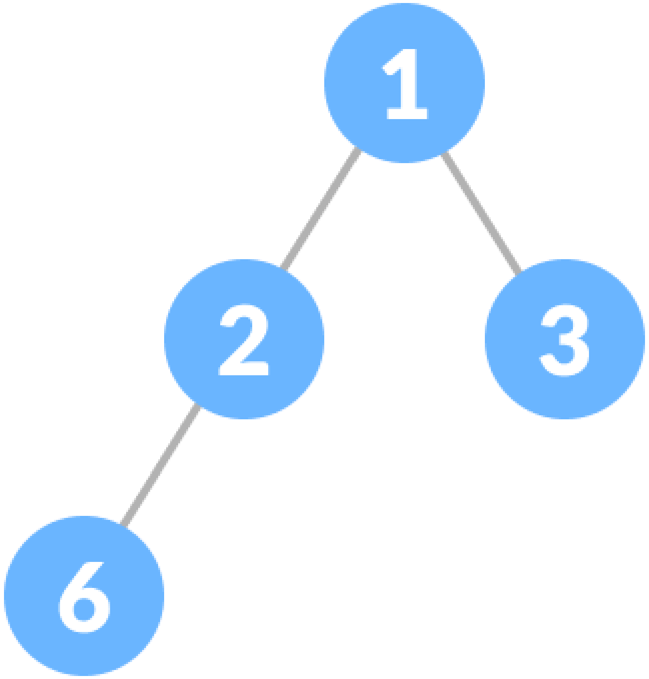
Full Binary Tree vs Complete Binary Tree



Comparison between full binary tree and complete binary tree



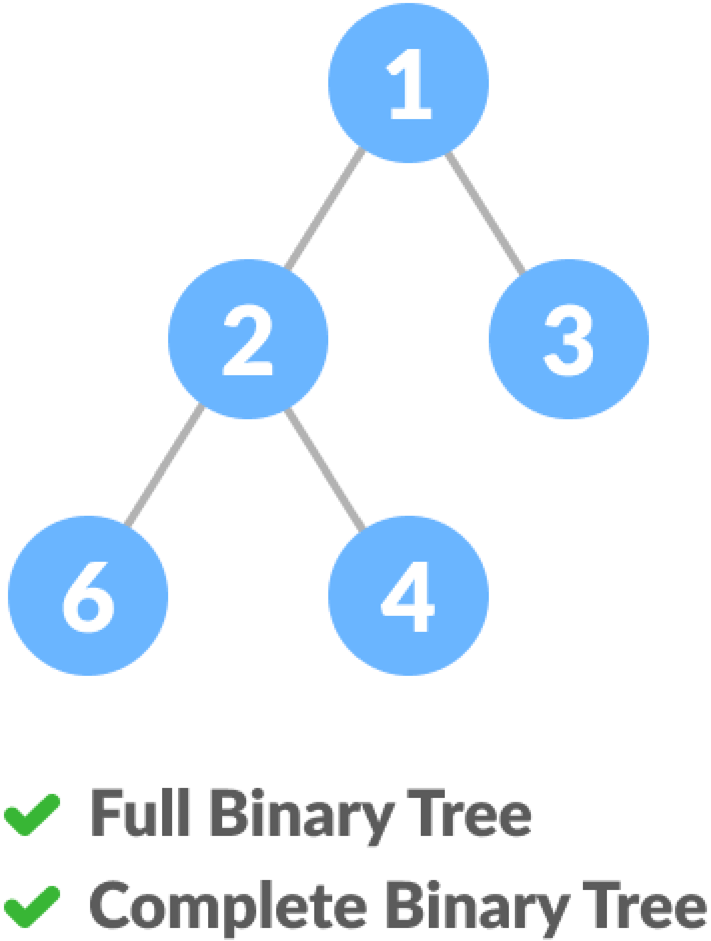
Comparison between full binary tree and complete binary tree



✗ Full Binary Tree

✓ Complete Binary Tree

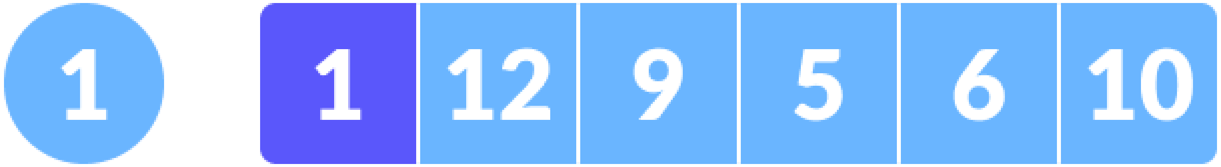
Comparison between full binary tree and complete binary tree



Comparison between full binary tree and complete binary tree

How a Complete Binary Tree is Created?

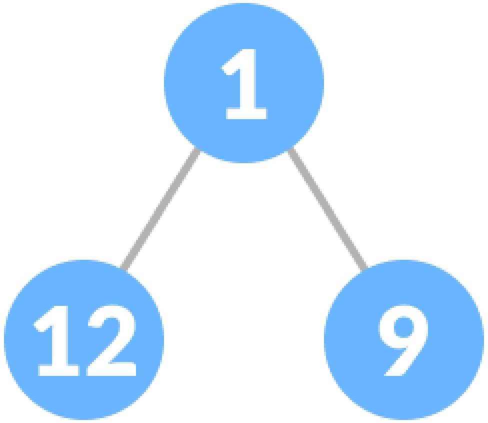
Select the first element of the list to be the root node. (no. of elements on level-I: 1)



1	12	9	5	6	10

Select the first element as root

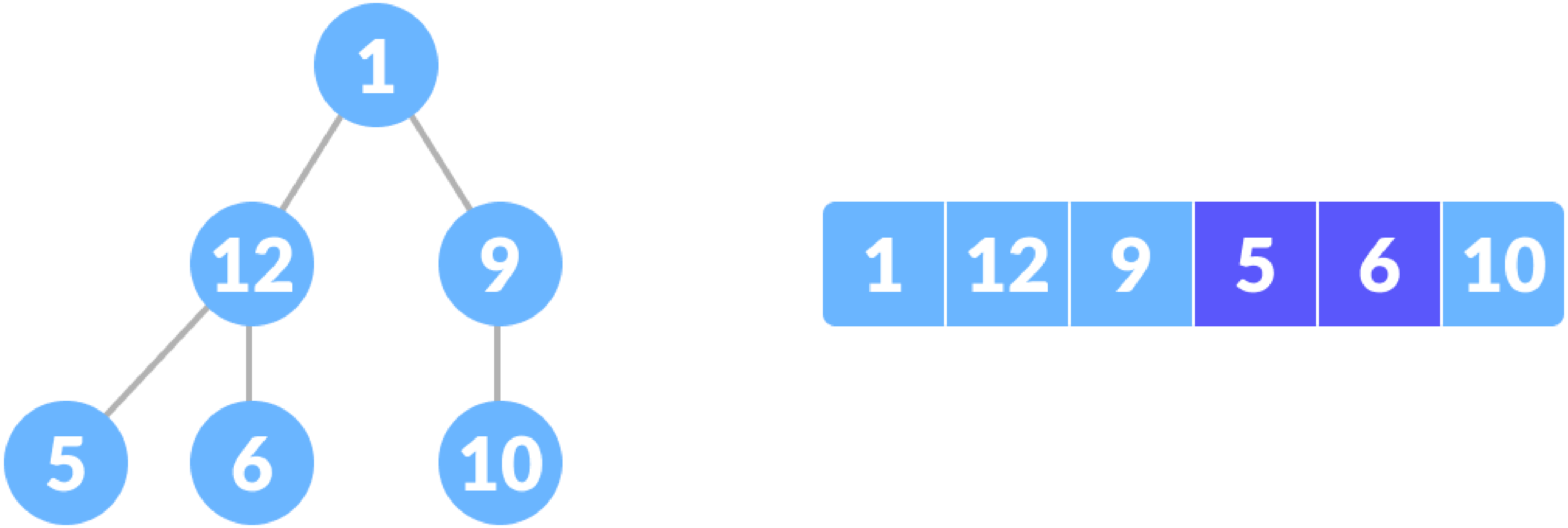
Put the second element as a left child of the root node and the third element as the right child. (no. of elements on level-II: 2)



1	12	9	5	6	10

12 as a left child and 9 as a right child

Put the next two elements as children of the left node of the second level. Again, put the next two elements as children of the right node of the second level (no. of elements on level-III: 4) elements). Keep repeating until you reach the last element.



1	12	9	5	6	10

5 as a left child and 6 as a right child

Binary Search tree

A binary search tree is a binary tree with a special property called the BST-property, which is given as follows:

For all nodes x and y, if y belongs to the left subtree of x, then the key at y is less than the key at x, and if y belongs to the right subtree of x, then the key at y is greater than the key at x.

We will assume that the keys of a BST are pairwise distinct.

Each node has the following attributes:

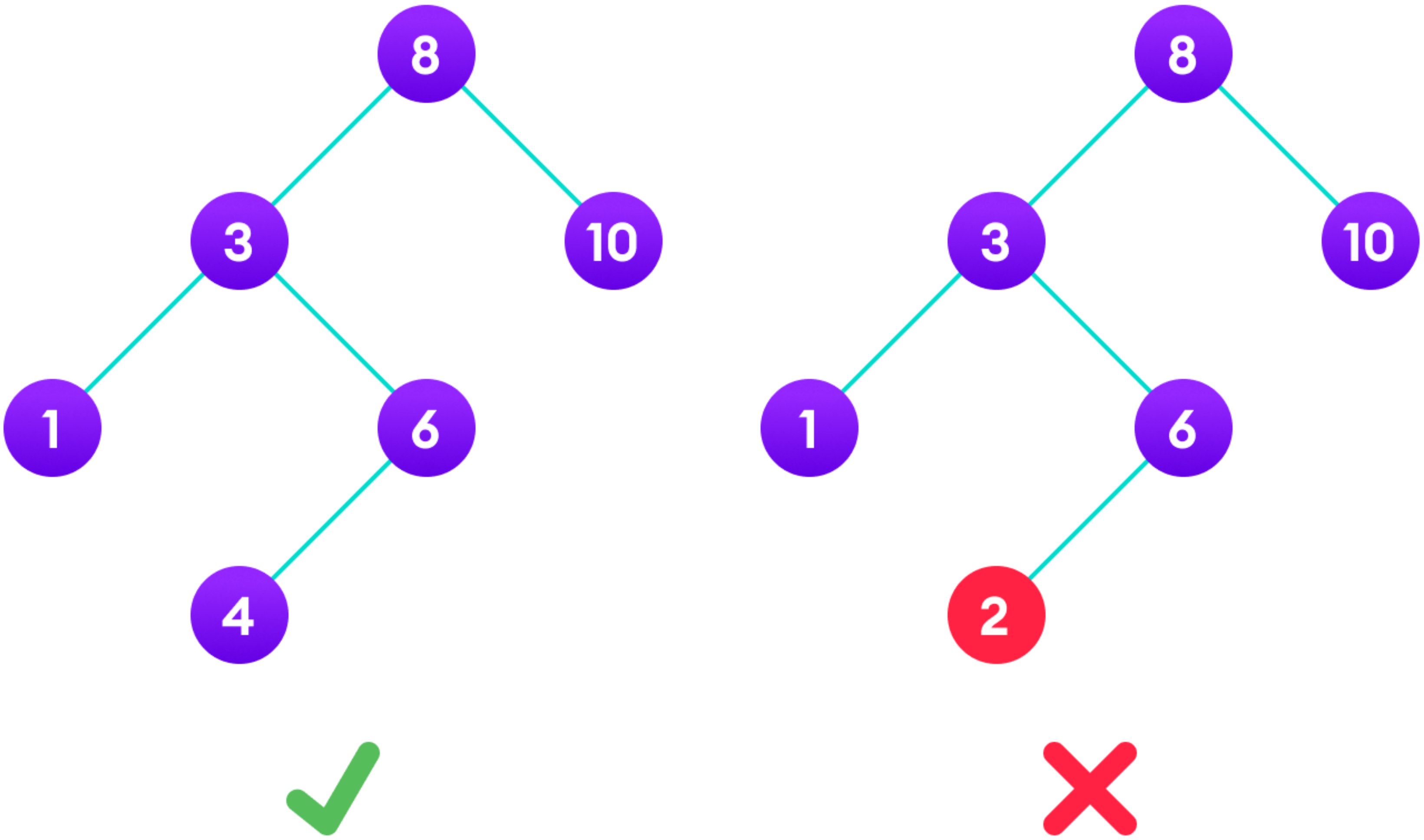
- p, left, and right, which are pointers to the parent, the left child, and the right child, respectively, and
- key, which is key stored at the node.

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

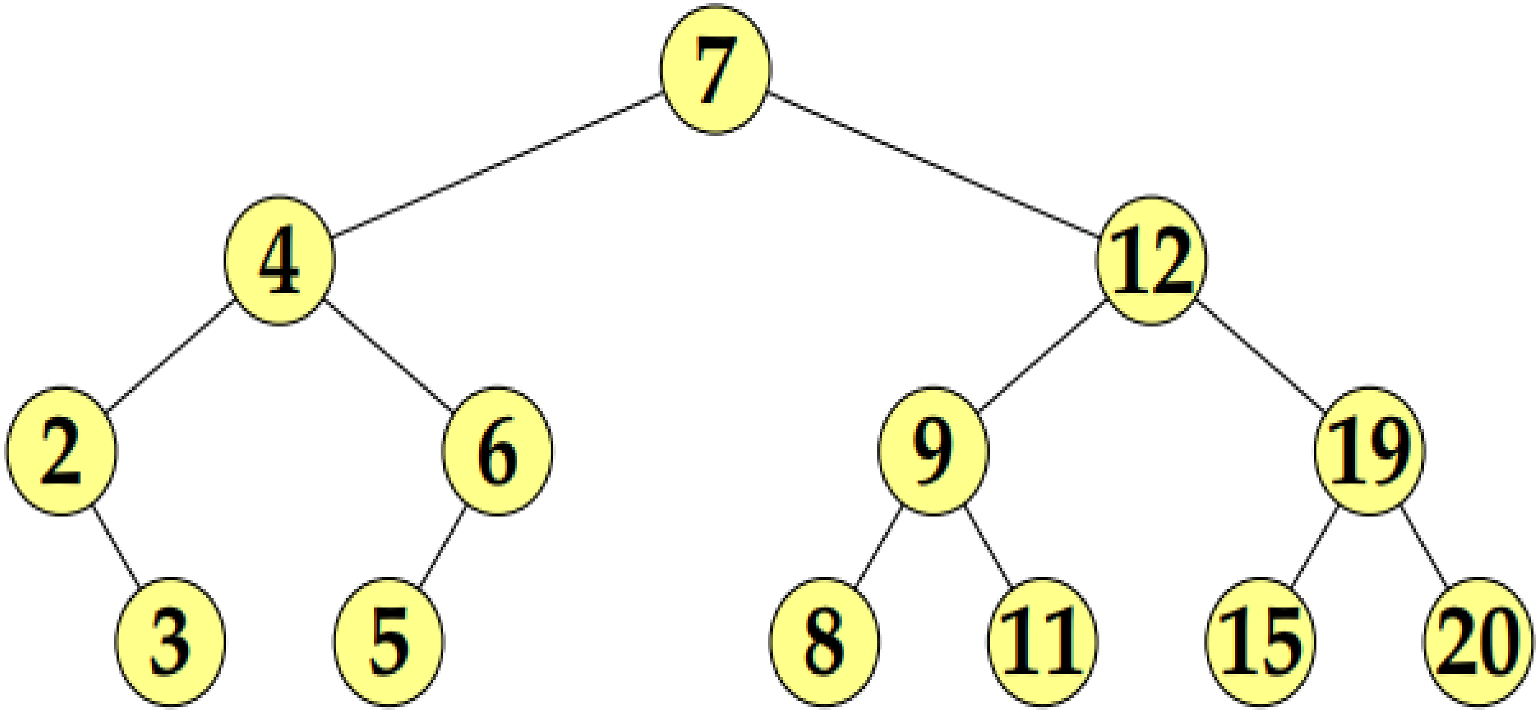
- It is called a **binary tree because each tree node has a maximum of two children.**
- **It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time**.

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

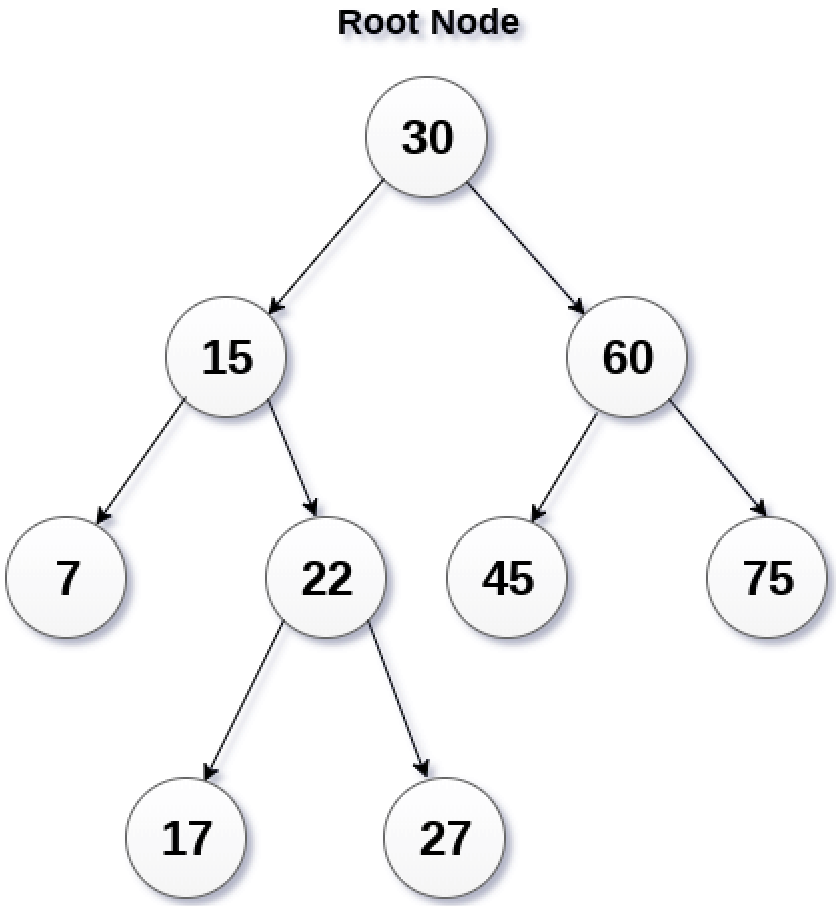


A tree having a right subtree with one value smaller than the root is shown to demonstrate that it is not a valid binary search tree


7
4
12
2
6
9
19
3
5
8
11
15
20

Binary Search Tree

1. Binary Search tree can be **defined as a class of binary trees, in which the nodes are arranged in a specific order. This is also called ordered binary tree.**
2. In a binary search tree, **the value of all the nodes in the left sub-tree is less than the value of the root.**
3. Similarly**, value of all the nodes in the right sub-tree is greater than or equal to the value of the root.**
4. This **rule will be recursively applied to all the left and right sub-trees of the root.**



Binary Search Tree

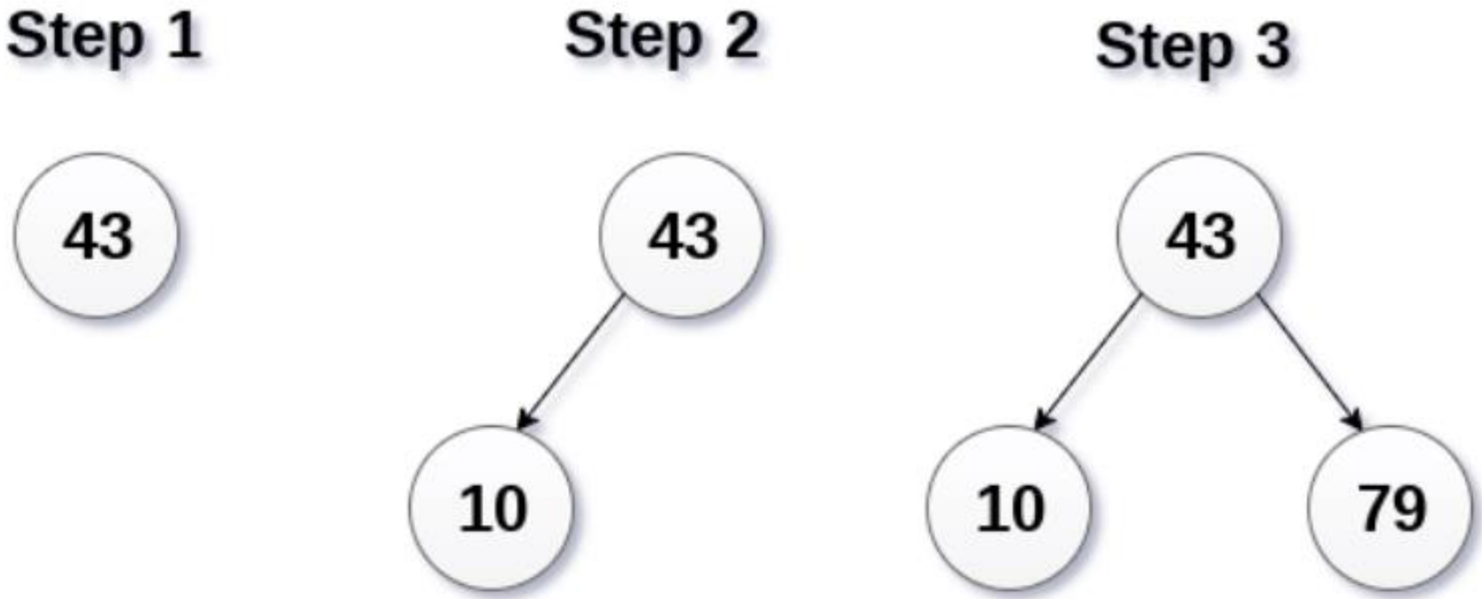
Advantages of using binary search tree

1. **Searching become very efficient in a binary search tree since**, we get a hint at each step, about which sub-tree contains the desired element.
2. **The binary search tree is considered as efficient data structure in compare to arrays and linked lists. In searching process, it removes half sub-tree at every step**. Searching for an element in a binary search tree takes o($\log_2$n) time. In worst case, the time it takes to search an element is 0(n).
3. **It also speed up the insertion and deletion operations as compare to that in array and linked list.**

Create the binary search tree using the following data elements.

43, 10, 79, 90, 12, 54, 11, 9, 50

- Insert 43 into the tree as the root of the tree.
- Read the next element, if it is lesser than the root node element, insert it as the root of the left sub-tree.
- Otherwise, insert it as the root of the right of the right sub-tree.
- The process of creating BST by using the given elements, is shown in the image below.

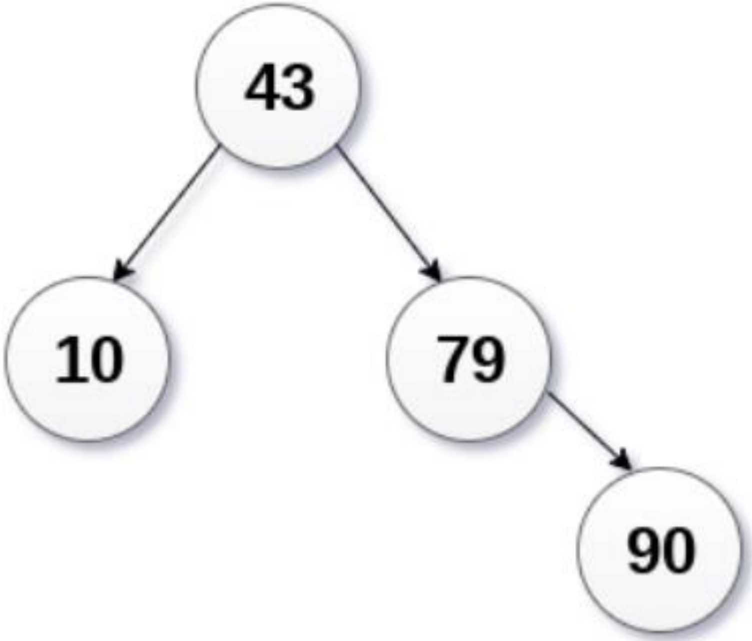


Step 1

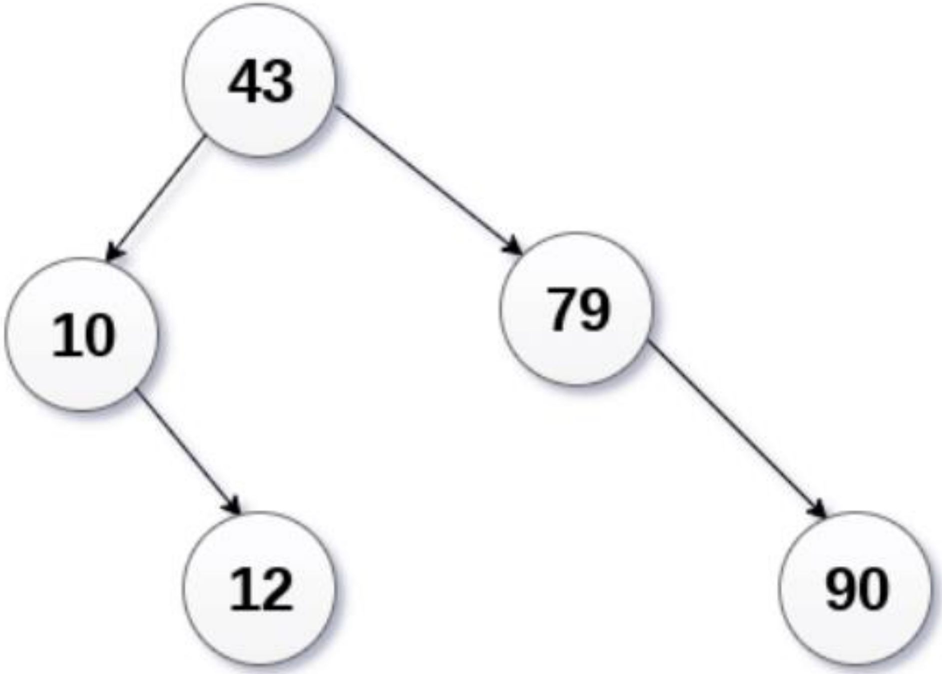
Step 2

Step 3

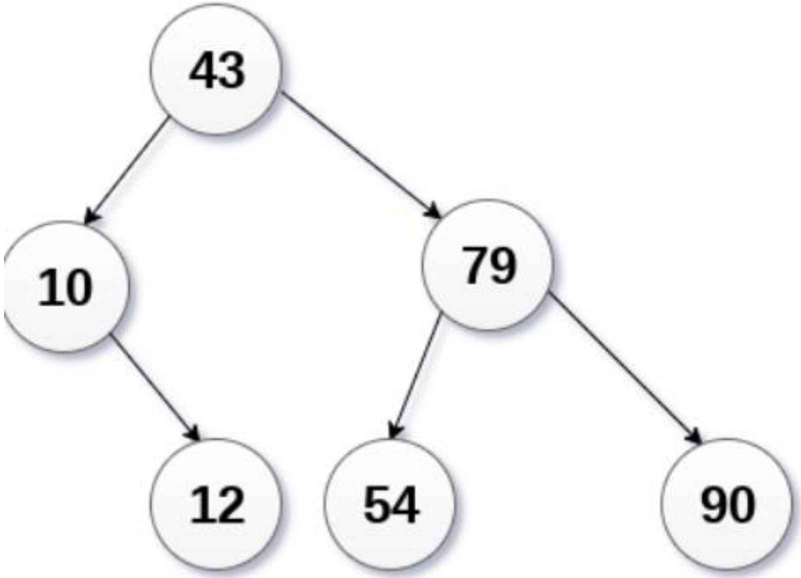
Step 4



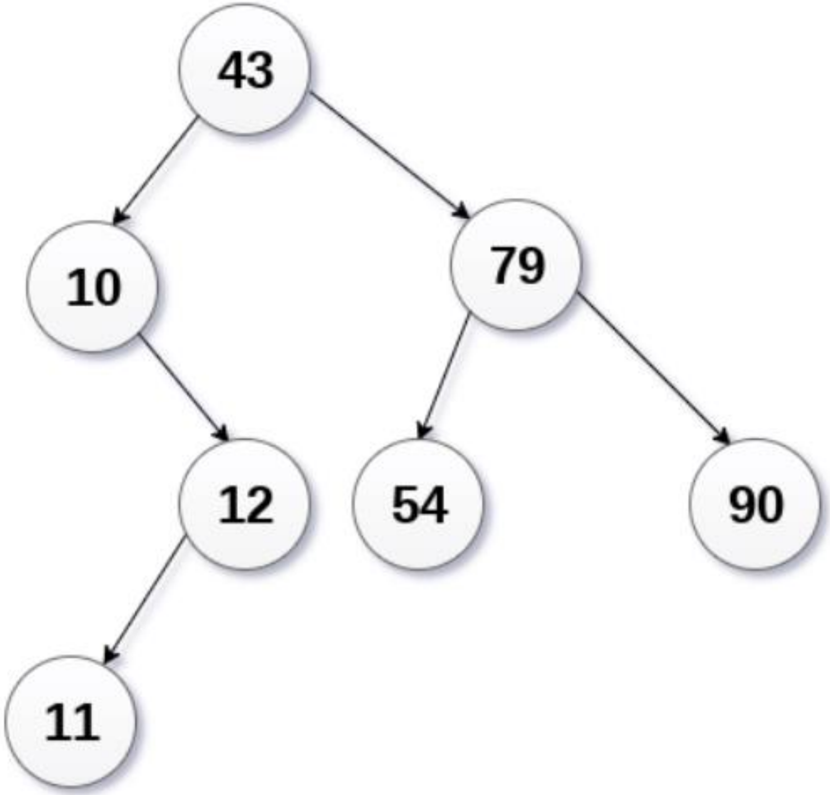
Step 5



Step 6

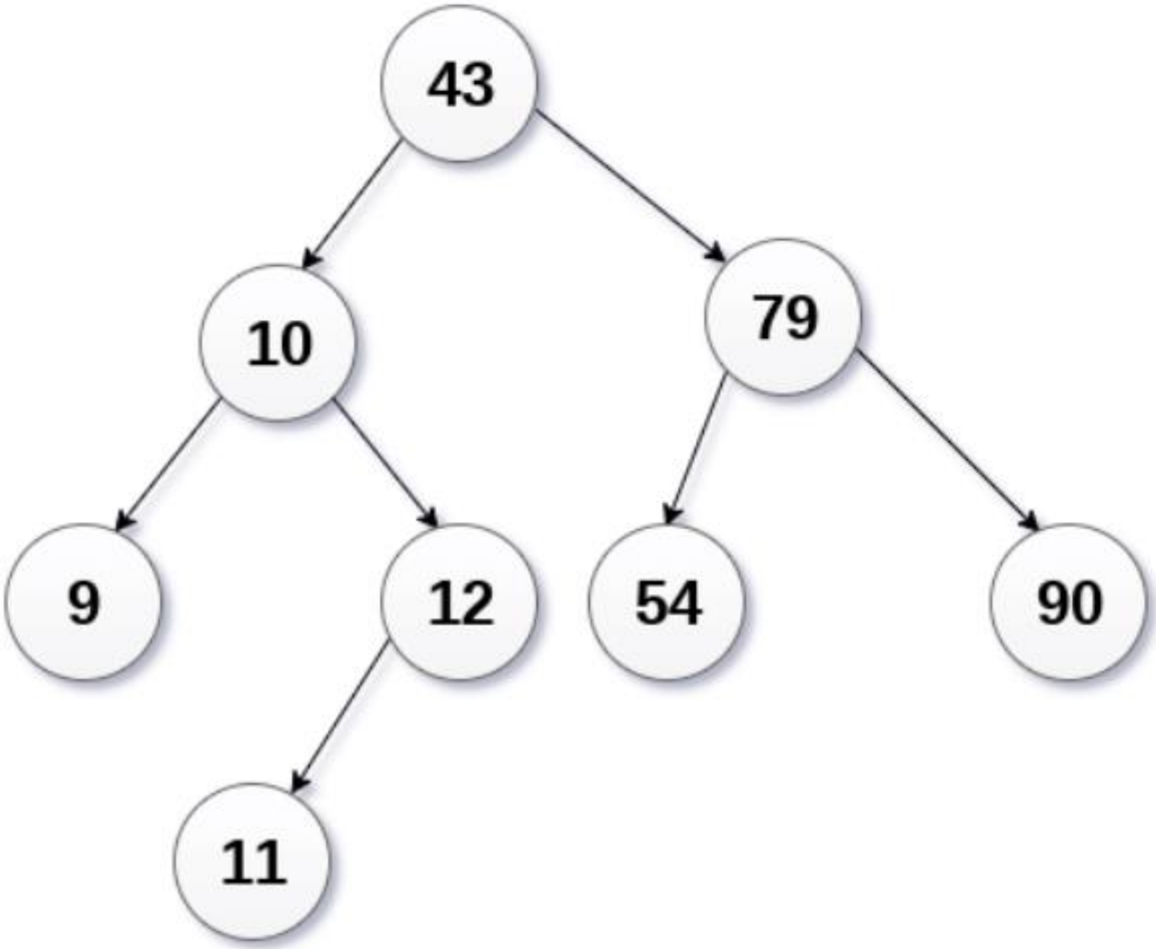


Step 7

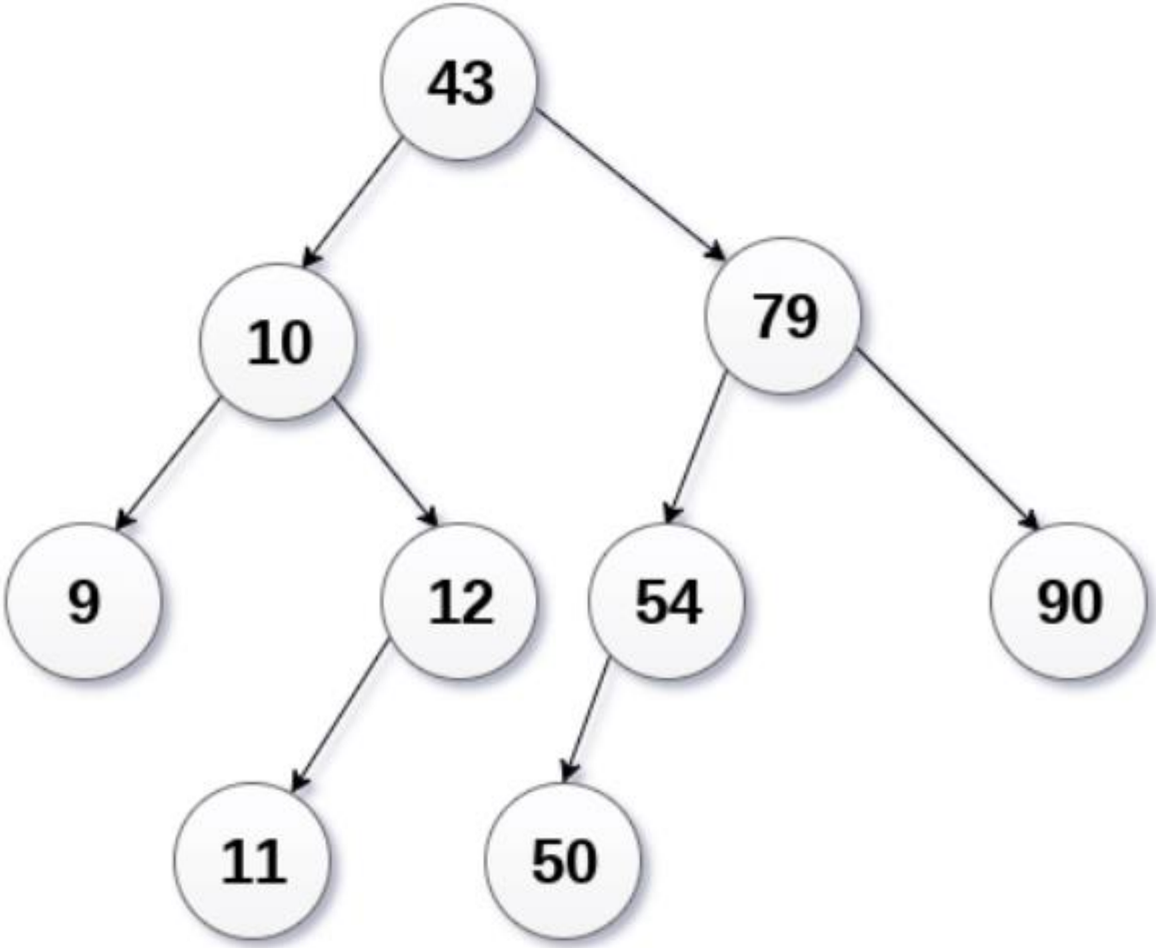


Step 8

Step 8



Step 9



Operations on Binary Search Tree

There are many operations which can be performed on a binary search tree.

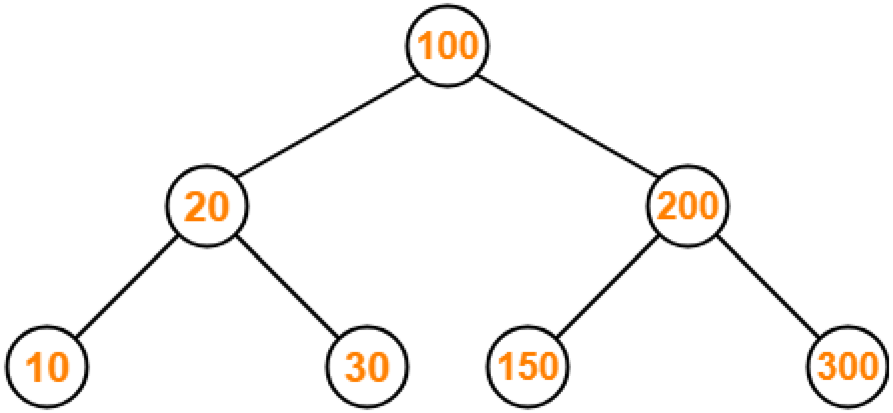
SN	Operation	Description
1	Searching in BST	Finding the location of some specific element in a binary search tree.
2	Insertion in BST	Adding a new element to the binary search tree at the appropriate location so that the property of BST do not violate.
3	Deletion in BST	Deleting some specific node from a binary search tree. However, there can be various cases in deletion depending upon the number of children, the node have.

Types of Traversals

The Binary Search Tree can be traversed in the following ways:

- Pre-order Traversal
- In-order Traversal
- Post-order Traversal

1. The *pre-order* traversal will visit nodes in **Parent-LeftChild-RightChild** order.
2. The *in-order* traversal will visit nodes in **LeftChild-Parent-RightChild** order. In this way, the tree is traversed in an ascending order of keys.
3. The *post-order* traversal will visit nodes in **LeftChild-RightChild-Parent** order.



Binary Search Tree

Preorder Traversal-

100 , 20 , 10 , 30 , 200 , 150 , 300

Inorder Traversal-

10 , 20 , 30 , 100 , 150 , 200 , 300

Postorder Traversal-

10 , 30 , 20 , 150 , 300 , 200 , 100

Notes :

- **Inorder traversal of a binary search tree always yields all the nodes in increasing order.**

Unlike Binary Trees,

- **A binary search tree can be constructed using only preorder or only postorder traversal result**.
- **This is because inorder traversal can be obtained by sorting the given result in increasing order**.

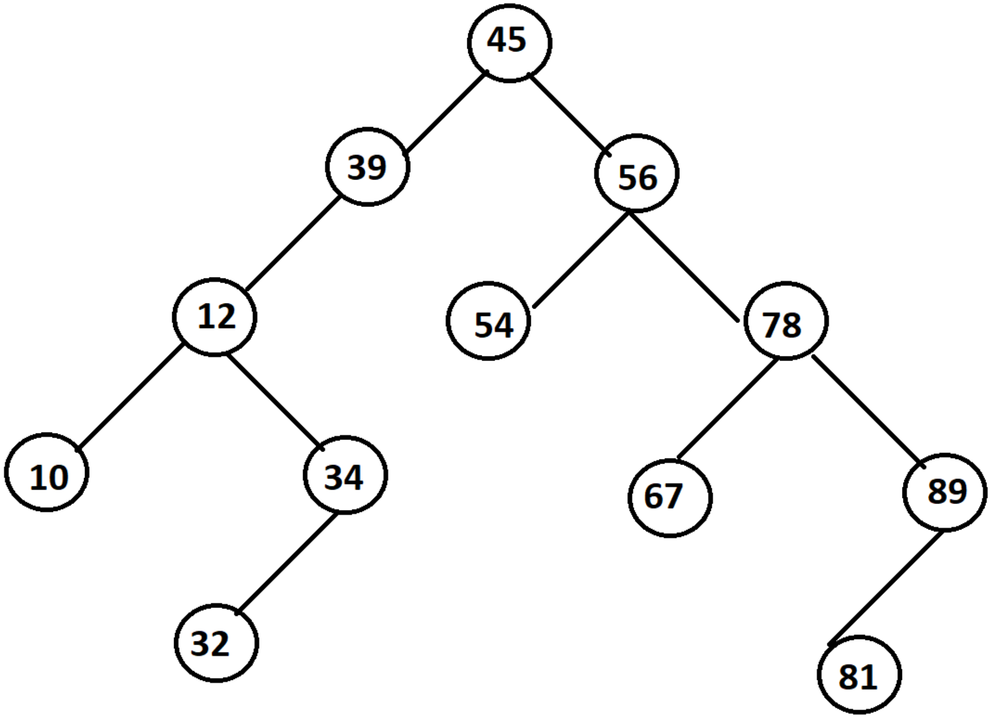
Q. Create a Binary Search Tree using following sequence of numbers 7 , 5 , 1 , 8 , 3 , 6 , 0 , 9 , 4 , 2.

Q. Create a Binary Search Tree using following sequence of numbers 30 , 20 , 10 , 15 , 25 , 23 , 39 , 35 , 42

Q. Create a Binary Search Tree using following sequence of numbers 98, 2, 48, 12, 56, 32, 4, 67, 23, 55, 46.

(a) Insert 21, 39, 45, 54, and 63 into the tree.

(b) Delete values 23, 56, 2, and 45 from the tree.



Q. (i) Find the result of in-order, preorder, and post order traversals.

(ii) Insert 11, 22, 33, 44, 55, 66, and 77 in the tree.

Rebuild a binary tree from Inorder and Preorder traversals

The following procedure demonstrates on how to rebuild tree from given inorder and preorder traversals of a binary tree:

- Preorder traversal visits Node, left subtree, right subtree recursively
- Inorder traversal visits left subtree, node, right subtree recursively
- Since we know that the first node in Preorder is its root, we can easily locate the root node in the inorder traversal and hence we can obtain left subtree and right subtree from the inorder traversal recursively

Preorder Traversal: 1 2 4 8 9 10 11 5 3 6 7

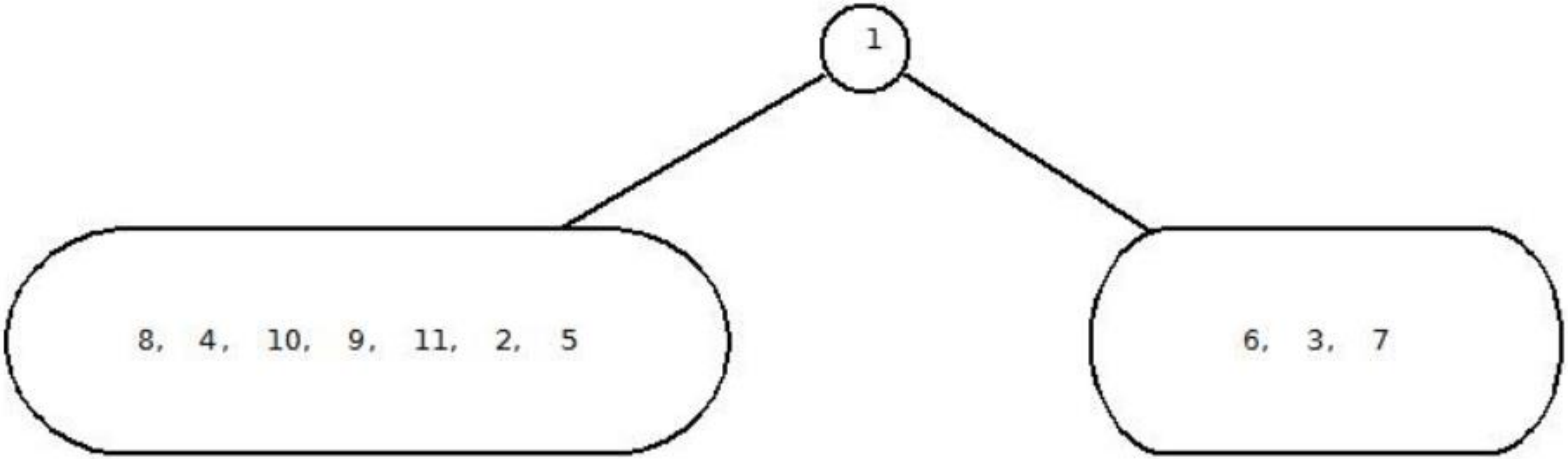
Inorder Traversal: 8 4 10 9 11 2 5 1 6 3 7

Iteration 1:

Root – {1}

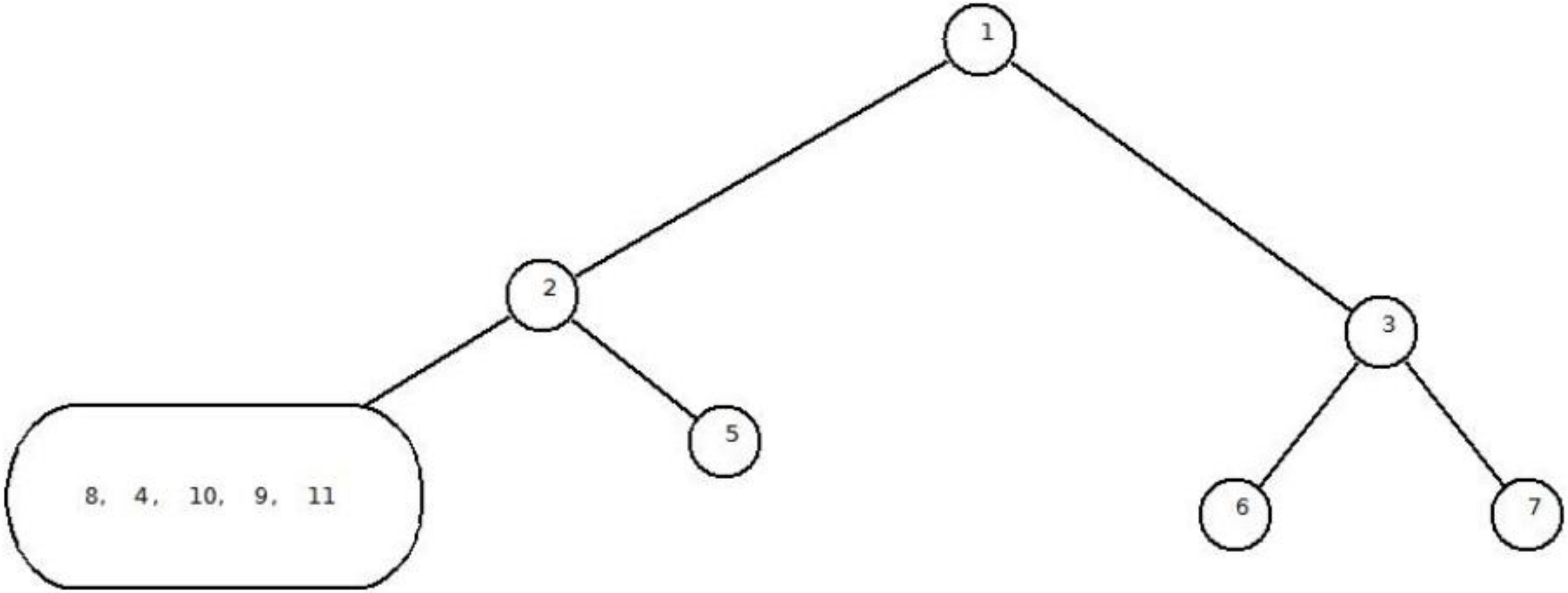
Left Subtree – {8,4,10,9,11,2,5}

Right Subtree – {6,3,7}



Iteration 2:

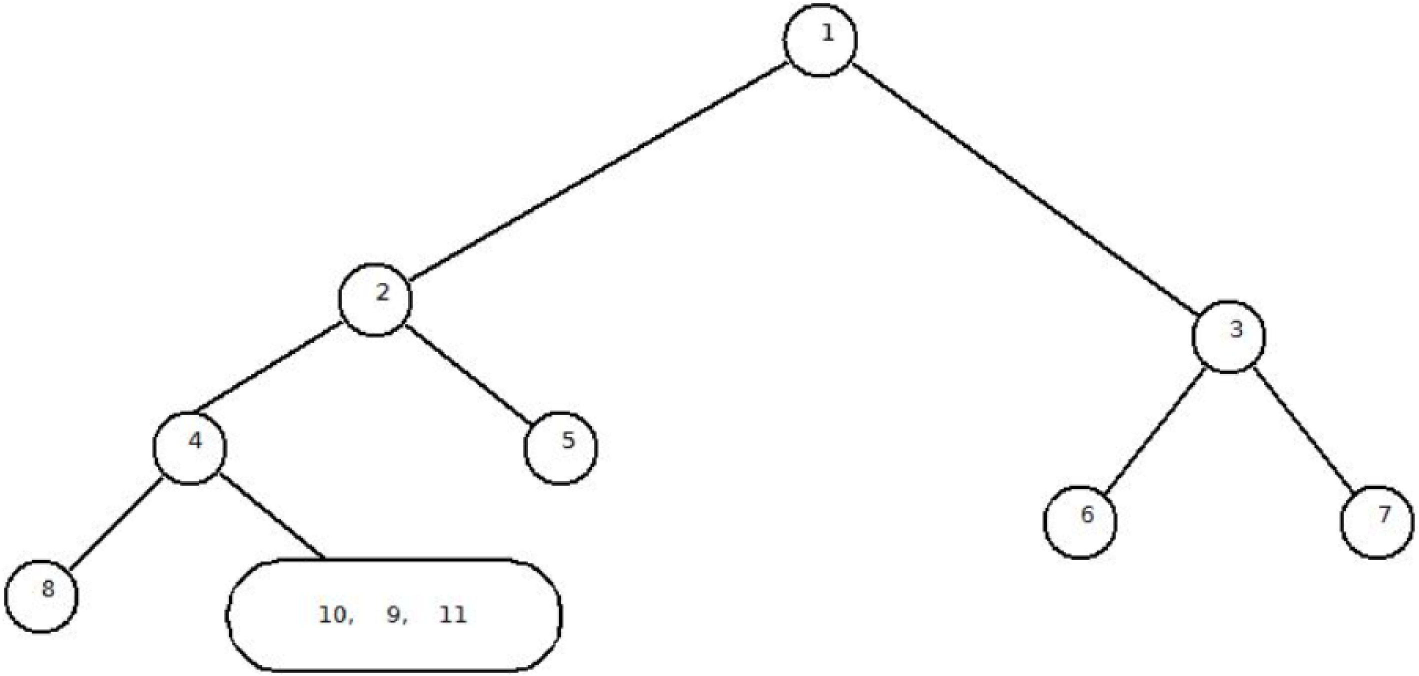
Root – {2}
Left Subtree – {8,4,10,9,11}
Right Subtree – {5}
	Root – {3}
	Left Subtree – {6}
	Right Subtree – {7}



Iteration 3:

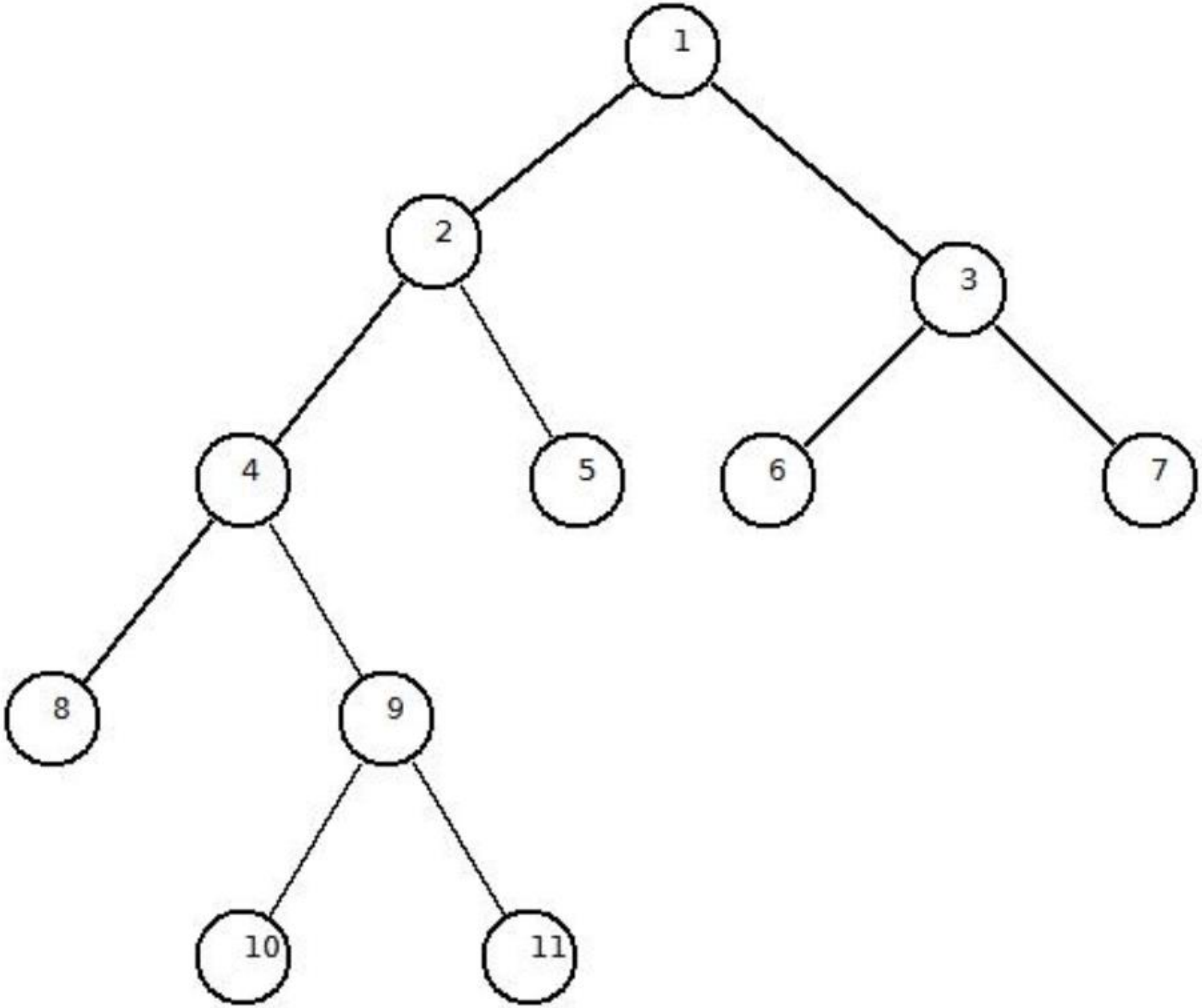
Root – {2}
Left Subtree – {8,4,10,9,11}
Right Subtree – {5}
		Root – {3}
		Left Subtree – {6}
		Right Subtree – {7}

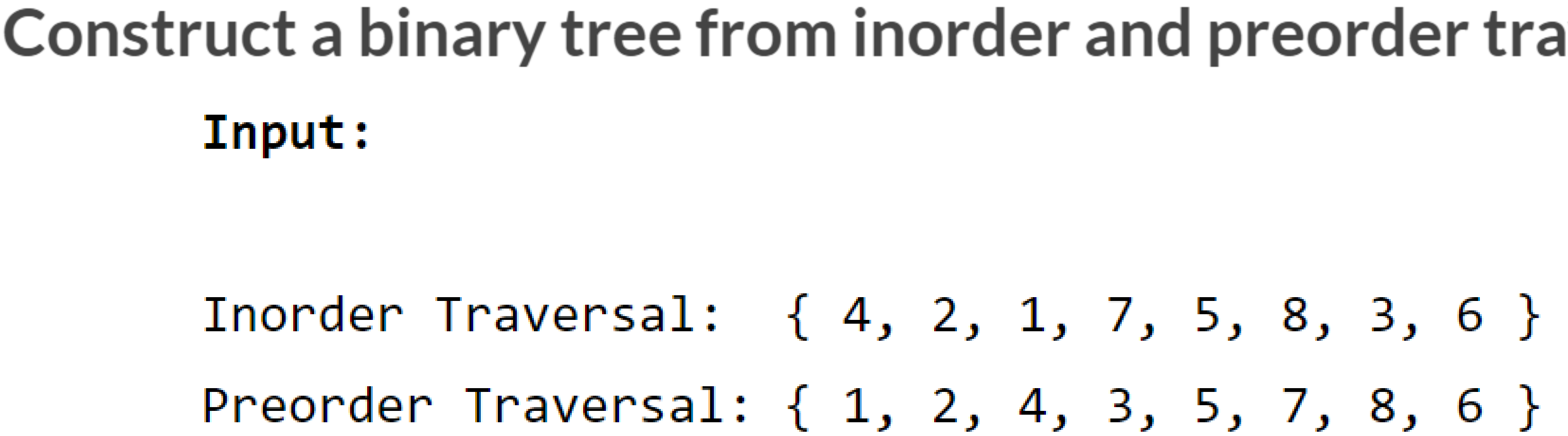
Root – {4}
Left Subtree – {8}
Right Subtree – {10,9,11}
	Done	Done



Iteration 4:

<table>
<tr><td colspan="3">Root – {2}
Left Subtree – {8,4,10,9,11}
Right Subtree – {5}</td><td>Root – {3}
Left Subtree – {6}
Right Subtree – {7}</td></tr>
<tr><td colspan="2">Root – {4}
Left Subtree – {8}
Right Subtree – {10,9,11}</td><td>Done</td><td>Done</td></tr>
<tr><td>Done</td><td>R – {9}
Left ST – {10}
Right ST- {11}</td><td>Done</td><td>Done</td></tr>
</table>


1
2
3
4
5
6
7
8
9
10
11



Construct a binary tree from inorder and preorder traversal

```
Input:


Inorder Traversal:  { 4, 2, 1, 7, 5, 8, 3, 6 }
Preorder Traversal: { 1, 2, 4, 3, 5, 7, 8, 6 }
```

Inorder sequence: D B E A F C

Preorder sequence: A B D E C F

inorder = {2,5,6,10,12,14,15};

preorder = {10,5,2,6,14,12,15};

Construct Binary Tree from Inorder and Postorder Traversal

in-order: 4 2 5 (1) 6 7 3 8

post-order: 4 5 2 6 7 8 3 (1)

AVL Tree

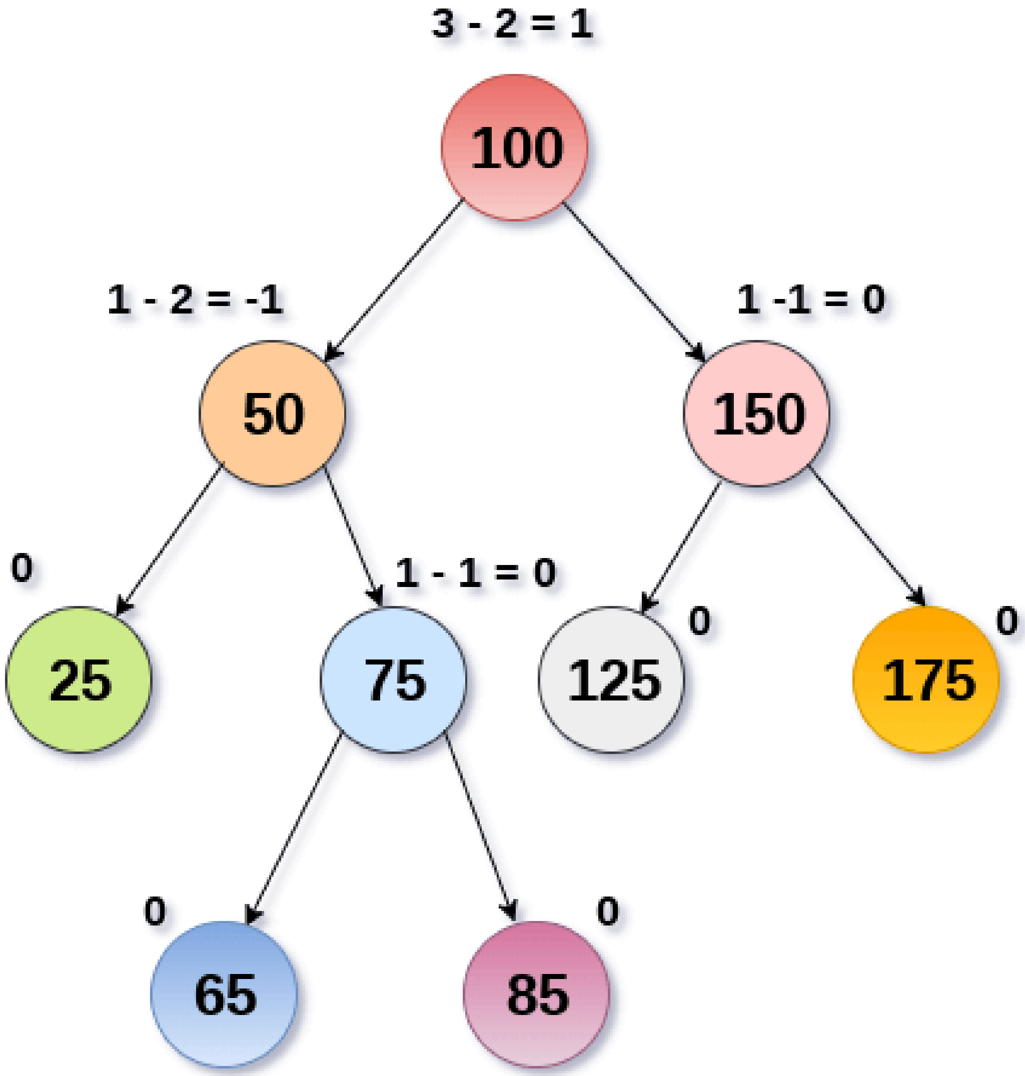
- AVL tree is a **self-balancing Binary Search Tree (BST)** where the **difference between heights of left and right subtrees cannot be more than one for all nodes.** AVL tree is a self-balancing binary search tree in which **each node maintains extra information called a balance factor whose value is either -1, 0 or +1.**
- AVL tree got its name after its inventor **Georgy Adelson-Velsky and Landis.**

- **AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962**. The tree is named AVL in honour of its inventors.
- AVL Tree can be defined as **height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree**.
- **Tree is said to be balanced if balance factor of each node is in between -1 to 1**, otherwise, the tree will be unbalanced and need to be balanced.

Balance Factor

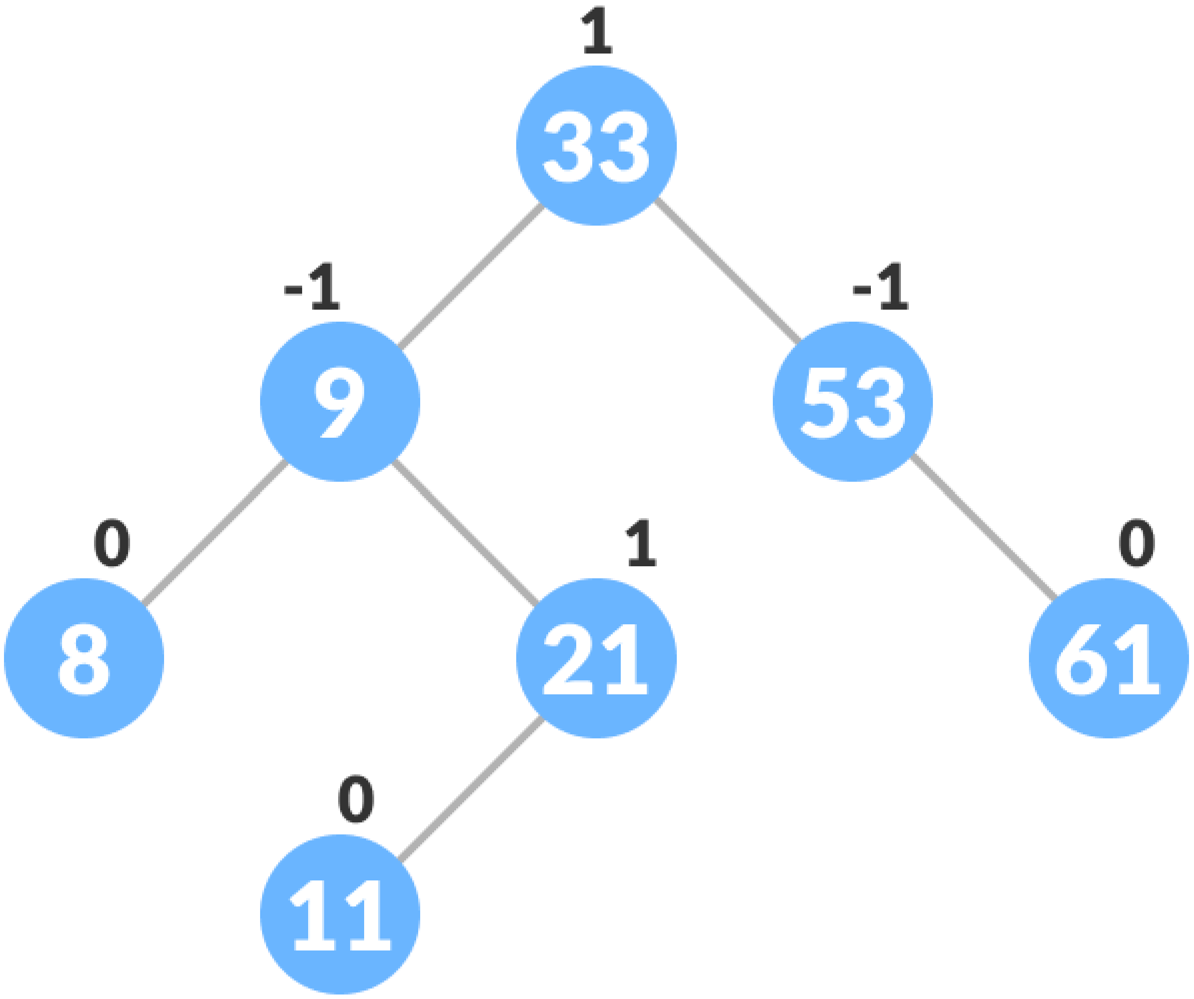
- Balance factor of a node in an AVL tree is the difference between the height of the left subtree and that of the right subtree of that node.
- **Balance Factor = (Height of Left Subtree - Height of Right Subtree) or (Height of Right Subtree - Height of Left Subtree)**
- The self balancing property of an avl tree is maintained by the balance factor. **The value of balance factor should always be -1, 0 or +1.**

- **Balance Factor (k) = height (left(k)) - height (right(k))**
- **If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.**
- **If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.**
- **If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.**
- An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. therefore, it is an example of AVL tree.



AVL Tree

An example of a balanced avl tree is:



Complexity

Algorithm	Average case	Worst case
Space	o(n)	o(n)
Search	o(log n)	o(log n)
Insert	o(log n)	o(log n)
Delete	o(log n)	o(log n)

Why AVL Tree?

AVL tree controls the height of the binary search tree by not letting it to be skewed. The time taken for all operations in a binary search tree of height h is **O(h)**. However, it can be extended to **O(n)** if the BST becomes skewed (i.e. worst case). By limiting this height to log n, AVL tree imposes an upper bound on each operation to be **O(log n)** where n is the number of nodes.

AVL Rotations

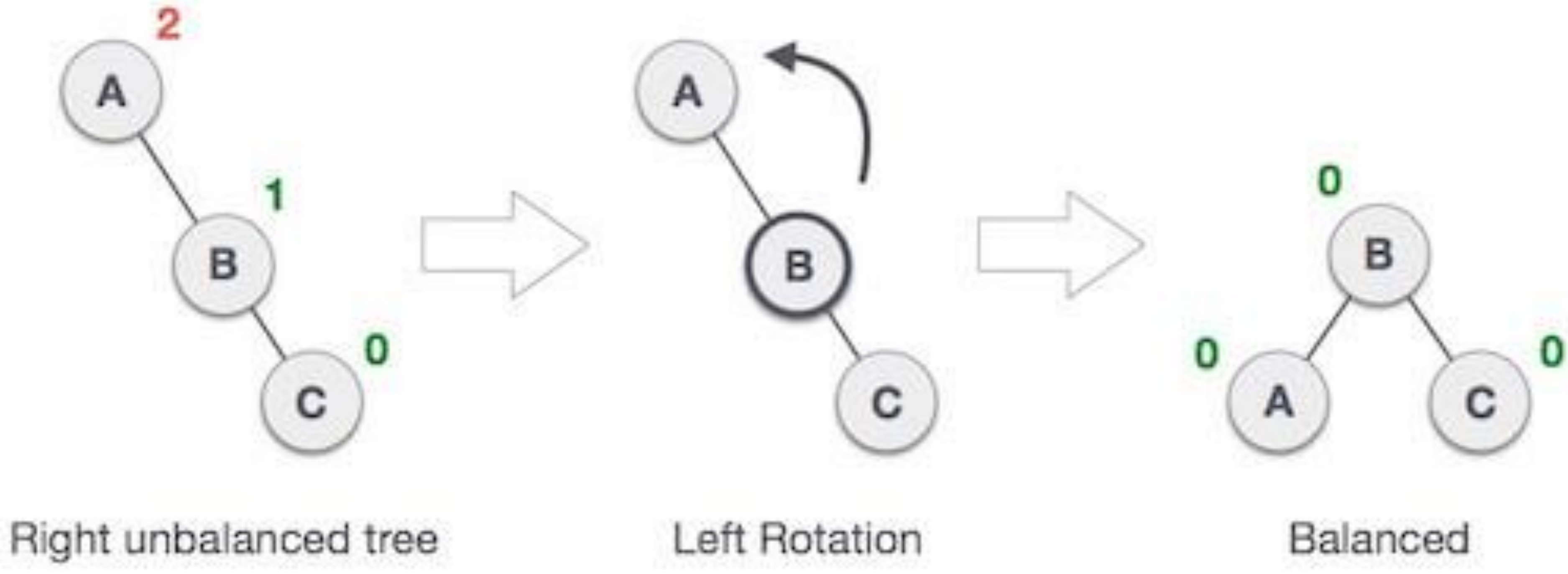
We perform rotation in AVL tree only in case if Balance Factor is other than **-1, 0, and 1**. There are basically four types of rotations which are as follows:

- L L rotation: Inserted node is in the left subtree of left subtree of A
- R R rotation : Inserted node is in the right subtree of right subtree of A
- **L R rotation : Inserted node is in the right subtree of left subtree of A**
- **R L rotation : Inserted node is in the left subtree of right subtree of A**

Where node A is the node whose balance Factor is other than -1, 0, 1. The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations. For a tree to be unbalanced, minimum height must be at least 2, Let us understand each rotation

1. RR Rotation

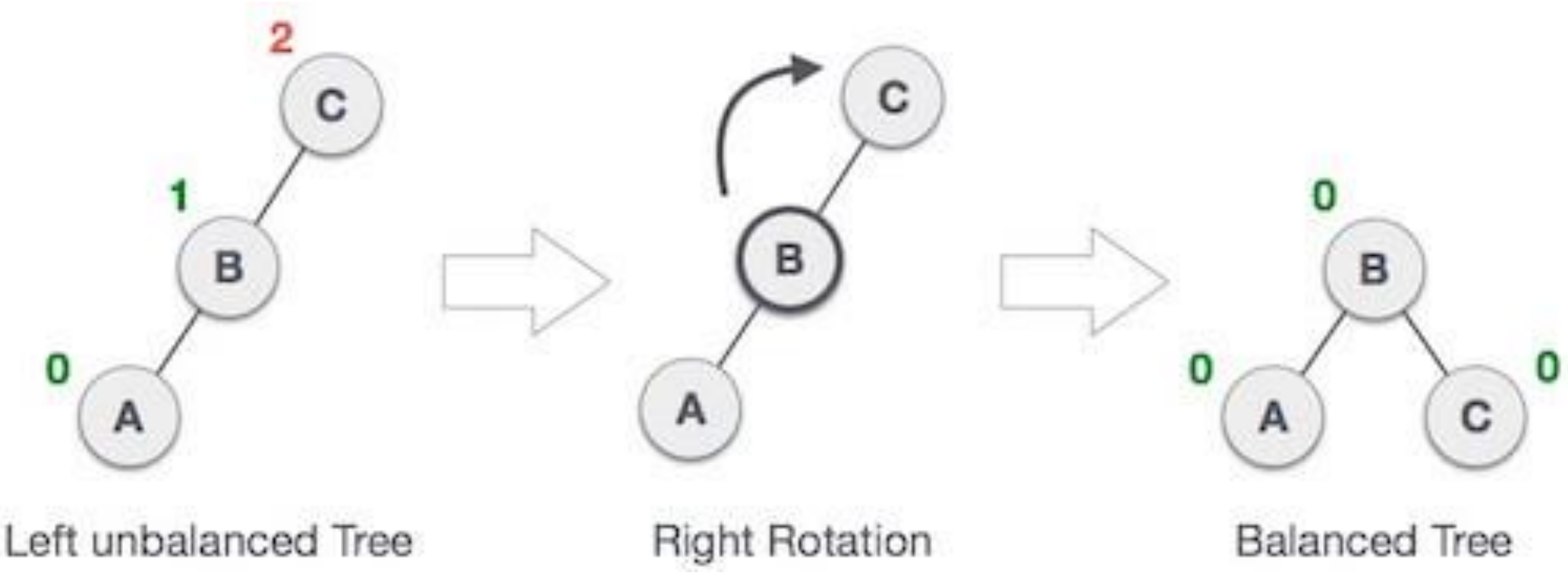
When **BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2**



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

2. LL Rotation

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



Left unbalanced Tree

Right Rotation

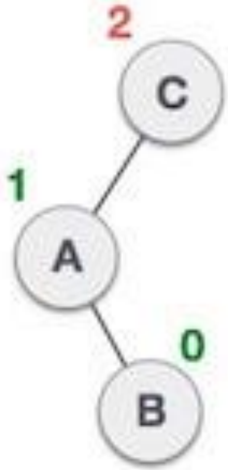
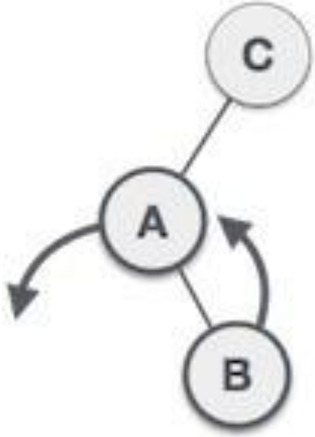
Balanced Tree

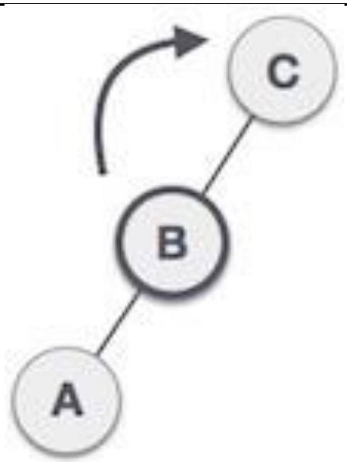
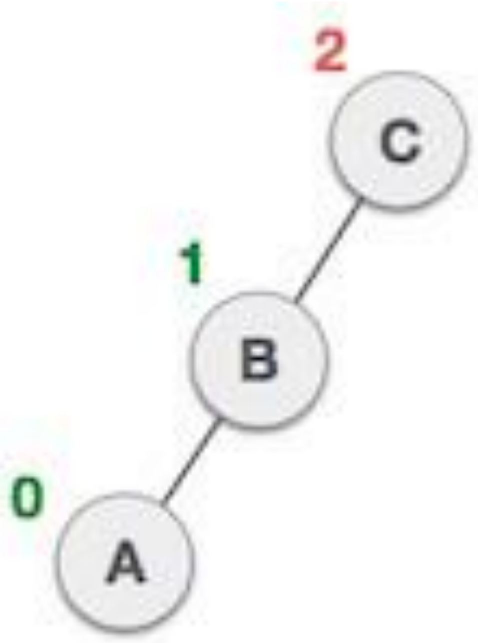
In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

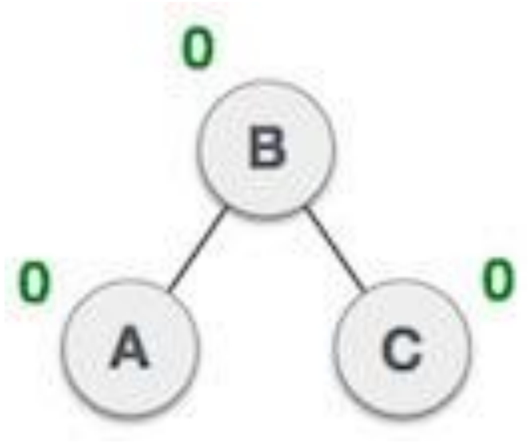
3. LR Rotation

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

Let us understand each and every step very clearly:

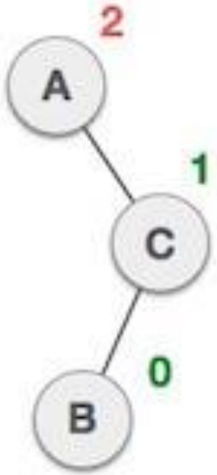
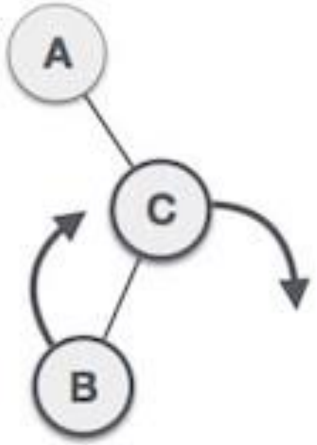
State	Action
	A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: **Inserted node is in the right subtree of left subtree of C**
	As **LR rotation = RR + LL rotation**,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

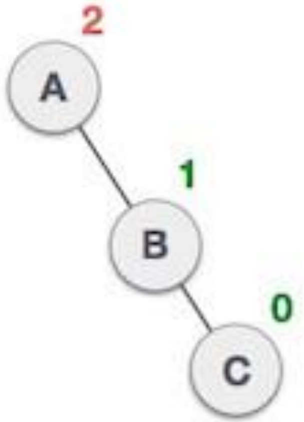
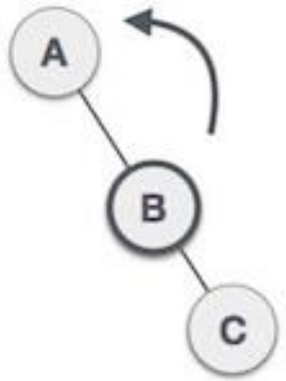
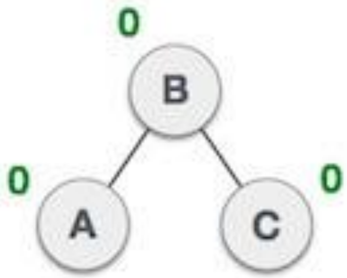
	After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of C
	Now we perform LL clockwise rotation on full tree, i.e. on node C. node C has now become the right subtree of node B, A is left subtree of B

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

RL Rotation

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

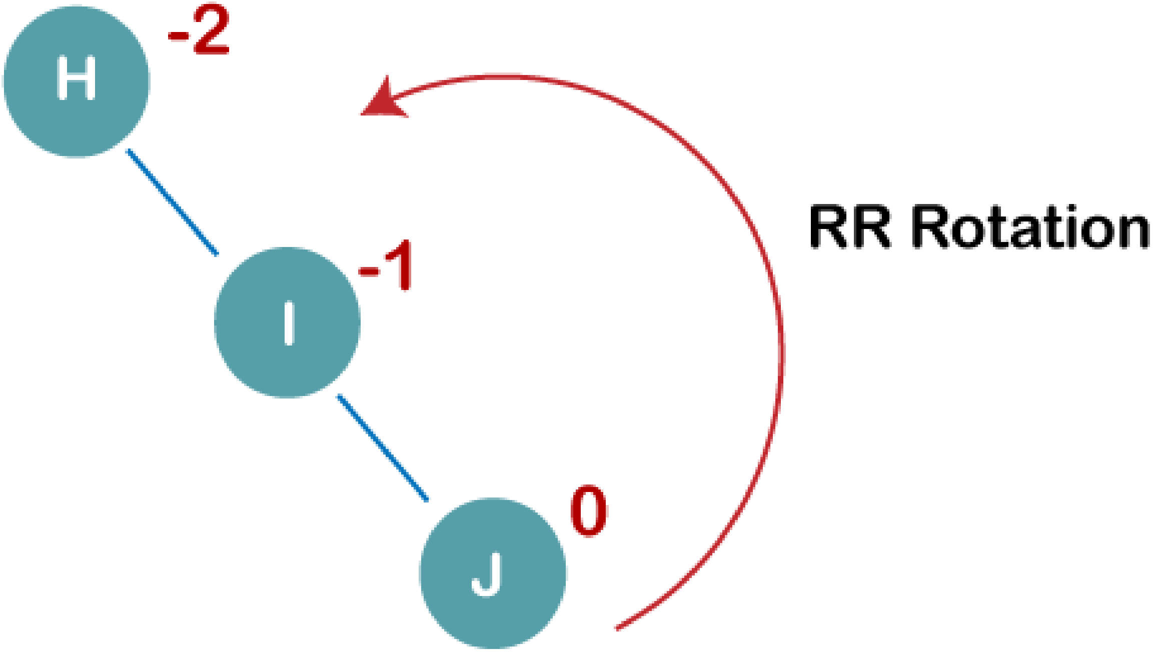
State	Action
	A node B has been inserted into the left subtree of C the right subtree of A, because of which A has become an unbalanced node having balance factor - 2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A
	As **RL rotation = LL rotation + RR rotation**, hence, LL (clockwise) on subtree rooted at C is performed first. By doing RR rotation, node C has become the right subtree of B.

	After performing LL rotation, node A is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A.
	Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node A. node C has now become the right subtree of node B, and node A has become the left subtree of B.
	Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.

Construct an AVL tree having the following elements

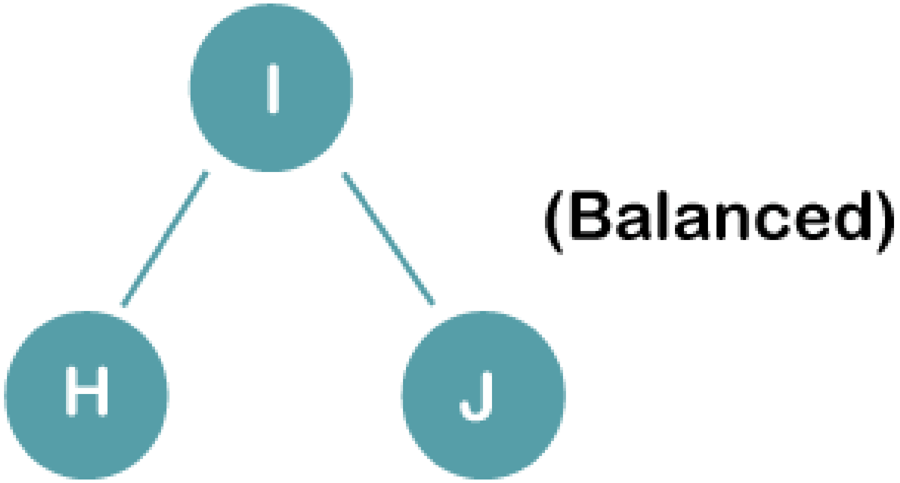
H, I, J, B, A, E, C, F, D, G, K, L

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

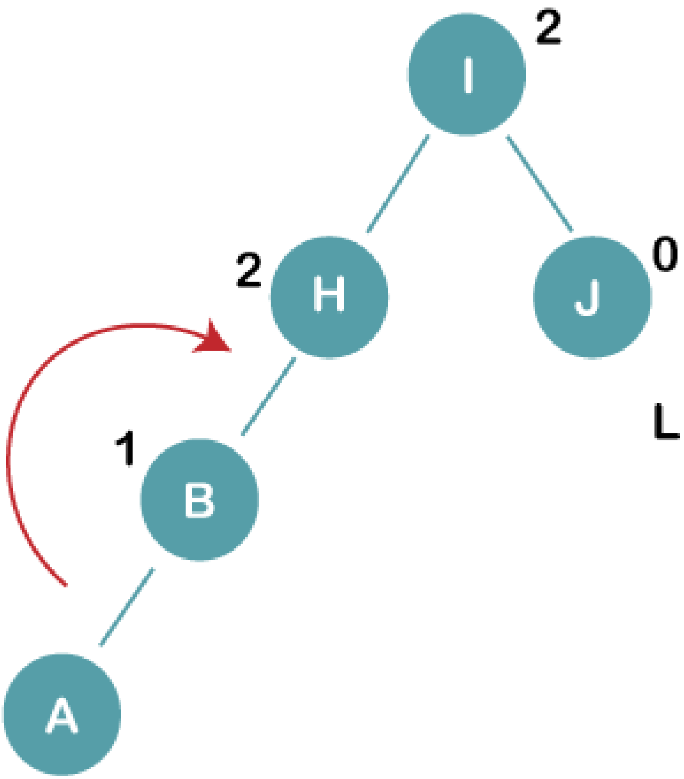


On inserting the above elements, especially in the case of H, the BST becomes unbalanced as the Balance Factor of H is -2. Since the BST is right-skewed, we will perform RR Rotation on node H.

The resultant balance tree is:



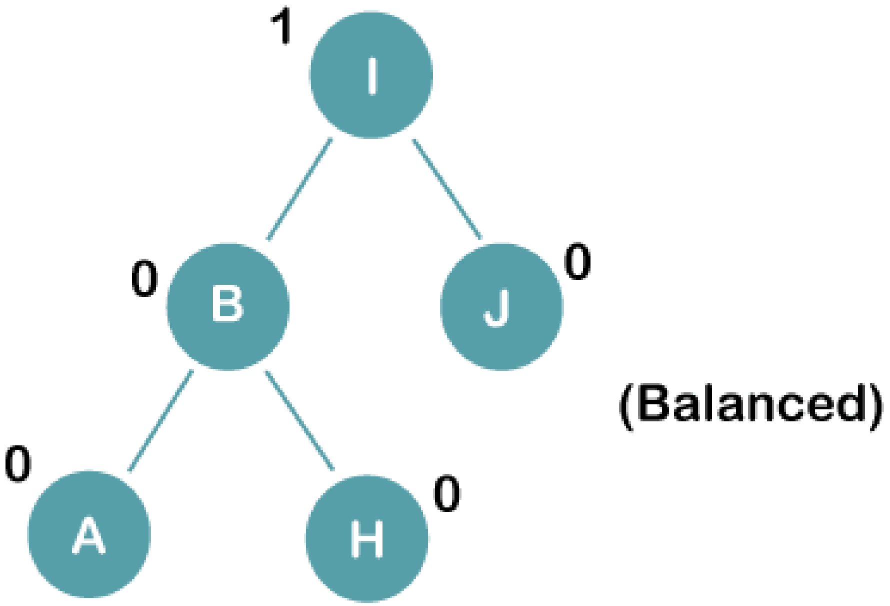
2. Insert B, A



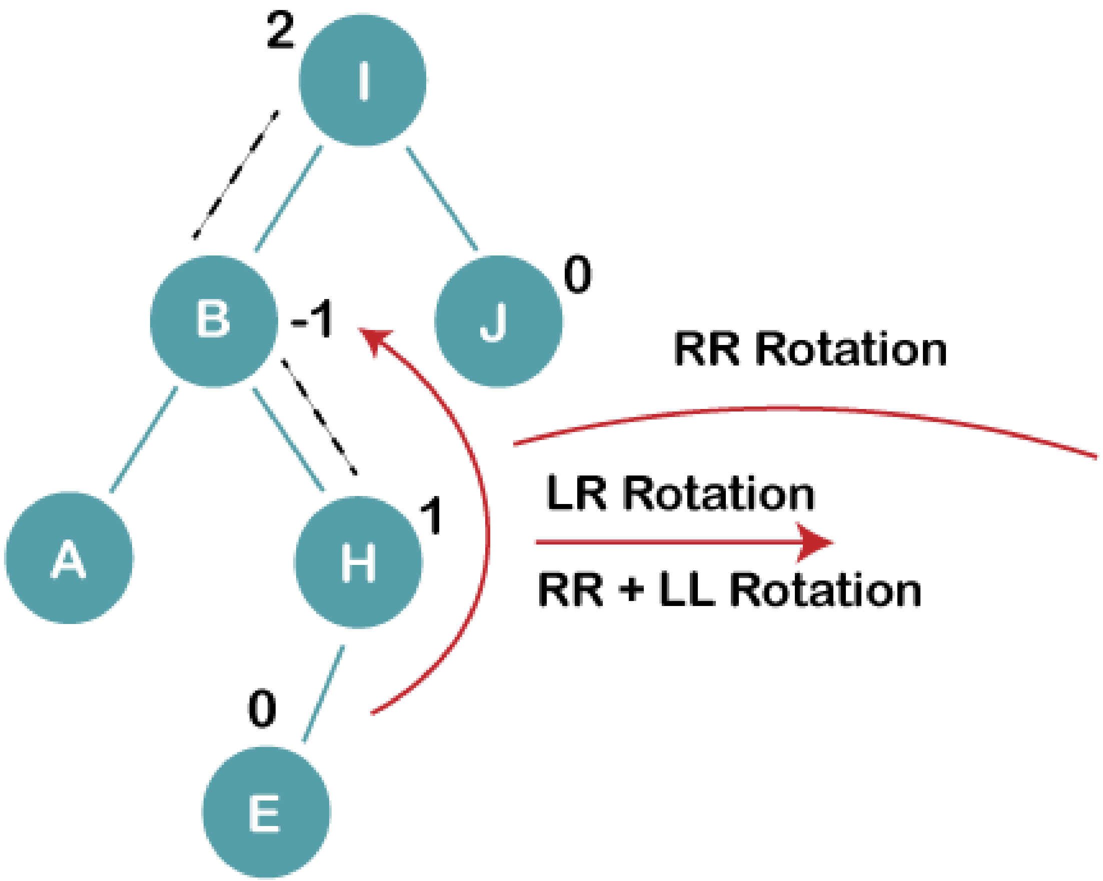
LL Rotation

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

The resultant balance tree is:



3. Insert E

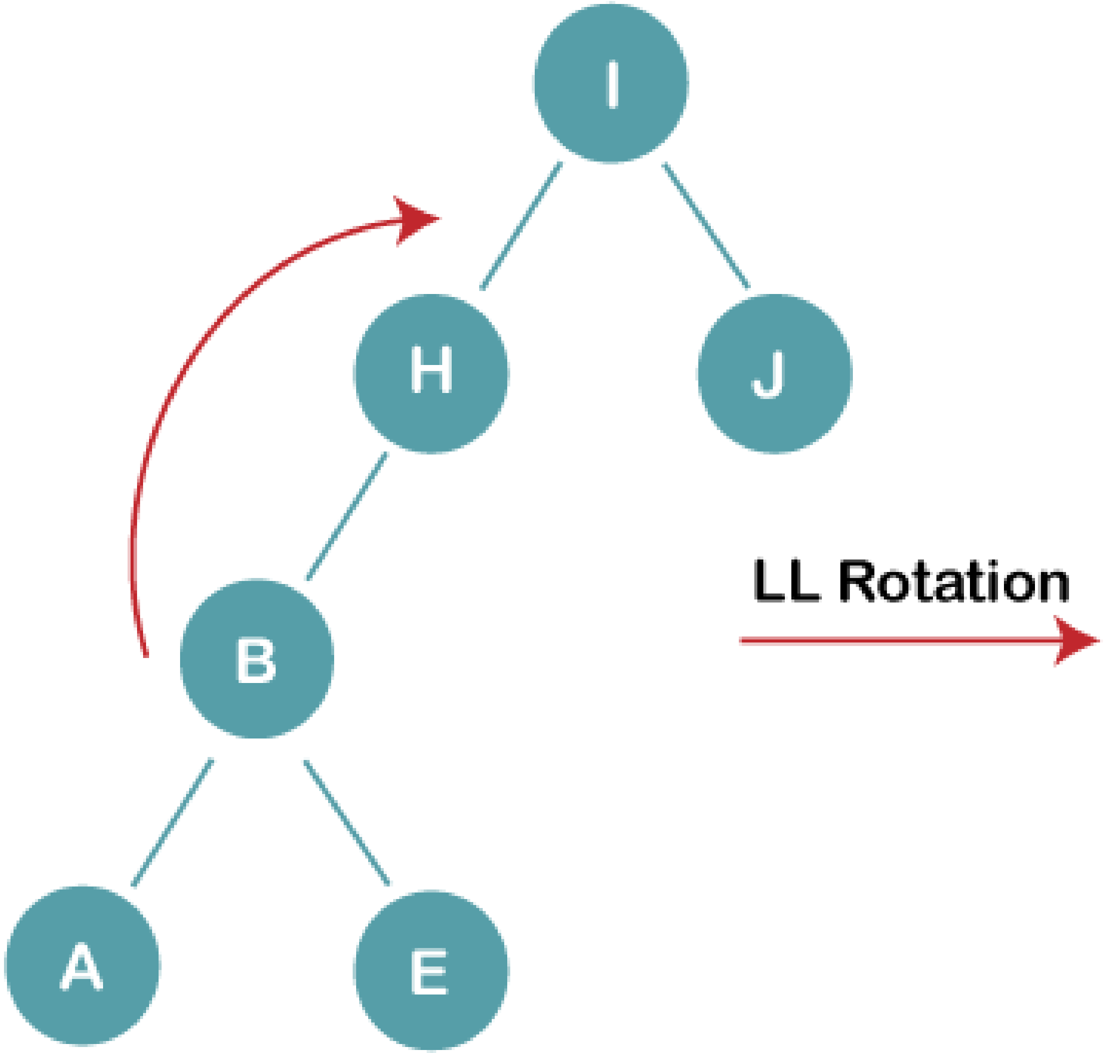


On inserting E, BST becomes unbalanced as the Balance Factor of I is 2, since if we travel from E to I we find that it is inserted in the left subtree of right subtree of I, we will perform LR Rotation on node I.

LR = RR + LL rotation

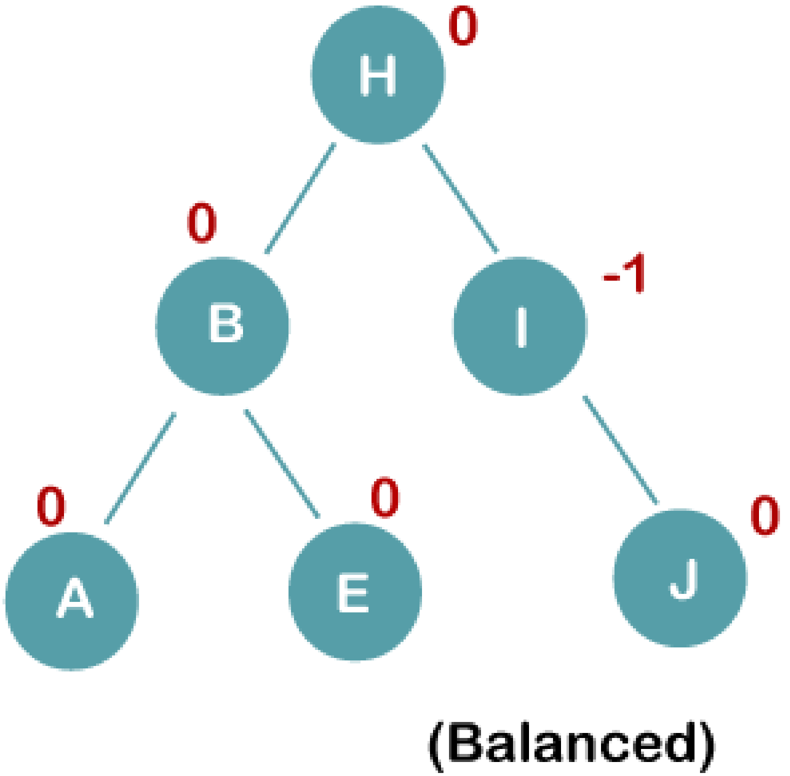
3 a) We first perform RR rotation on node B

The resultant tree after RR rotation is:

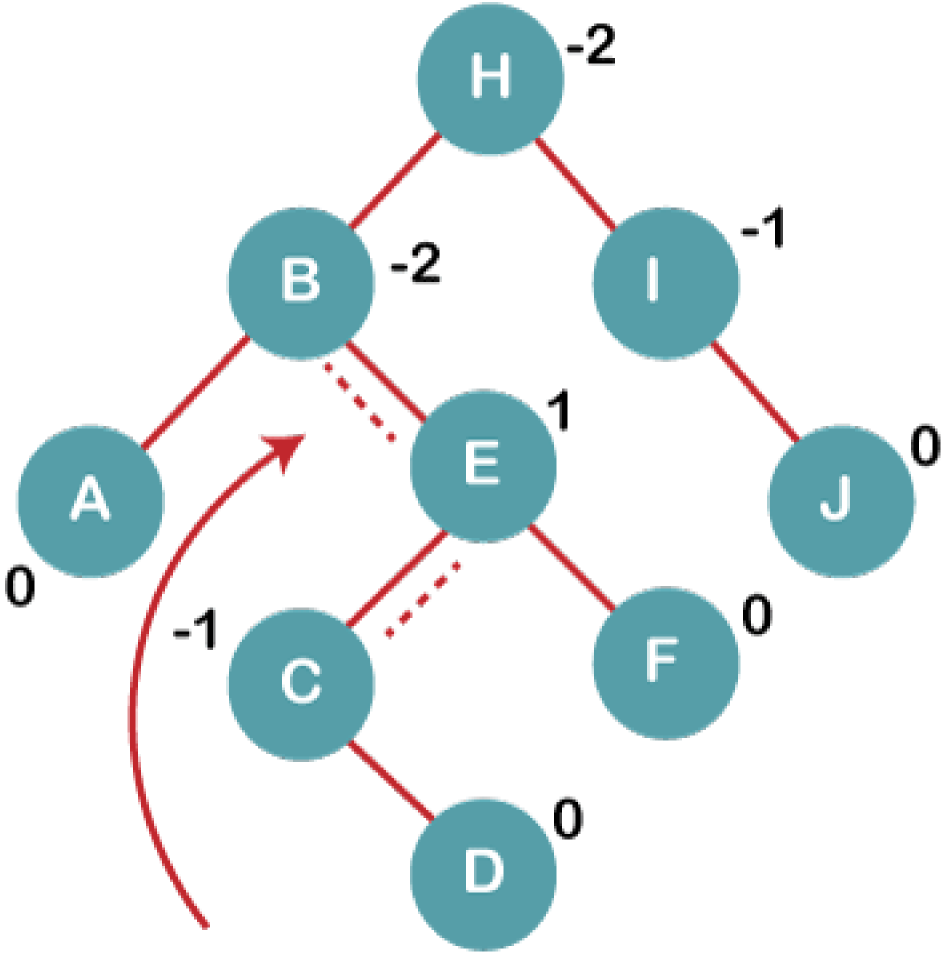

I
H
J
B
A
E
LL Rotation

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

The resultant balanced tree after LL rotation is:



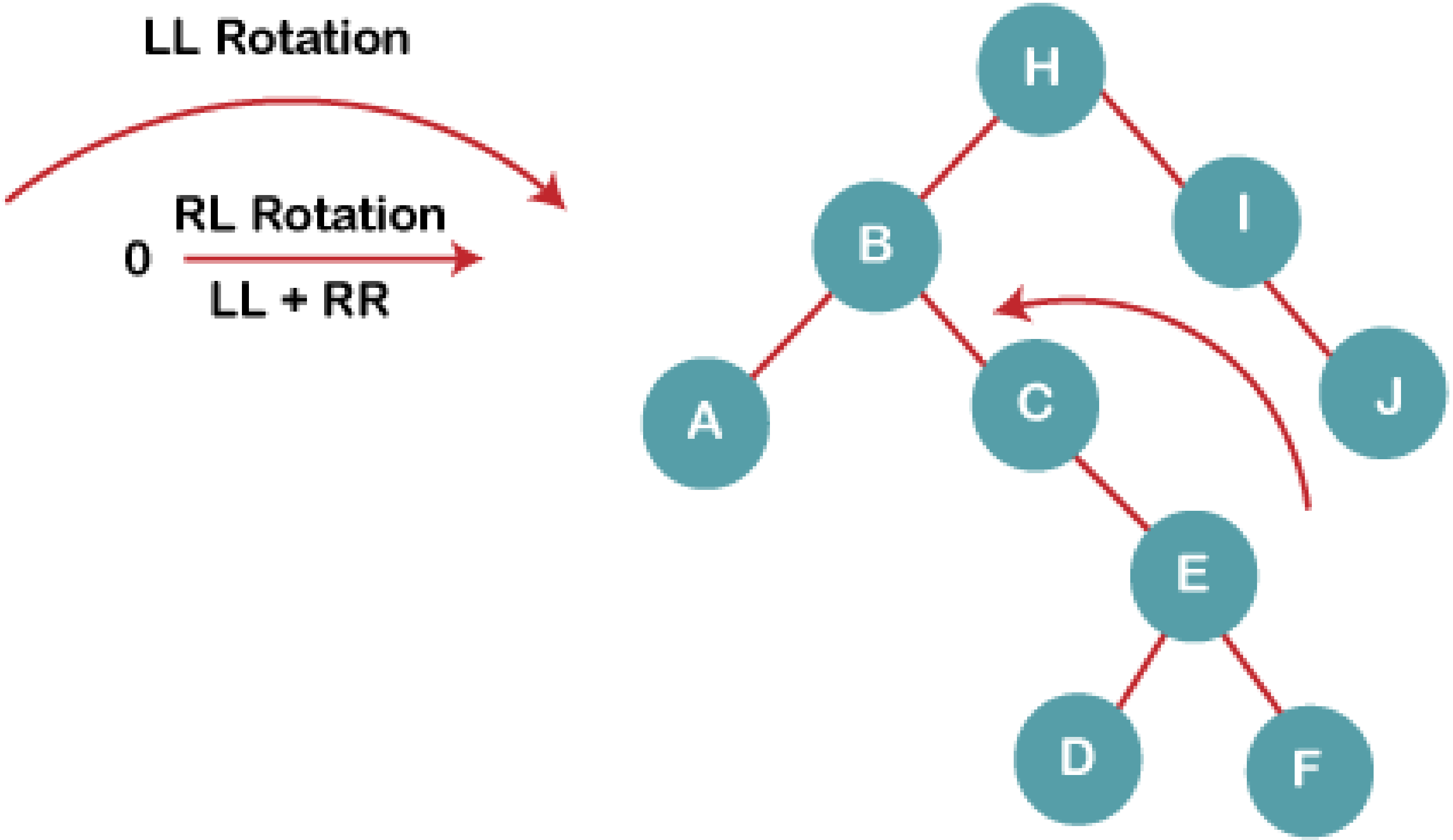
4. Insert C, F, D



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

4a) We first perform LL rotation on node E

The resultant tree after LL rotation is:



LL Rotation

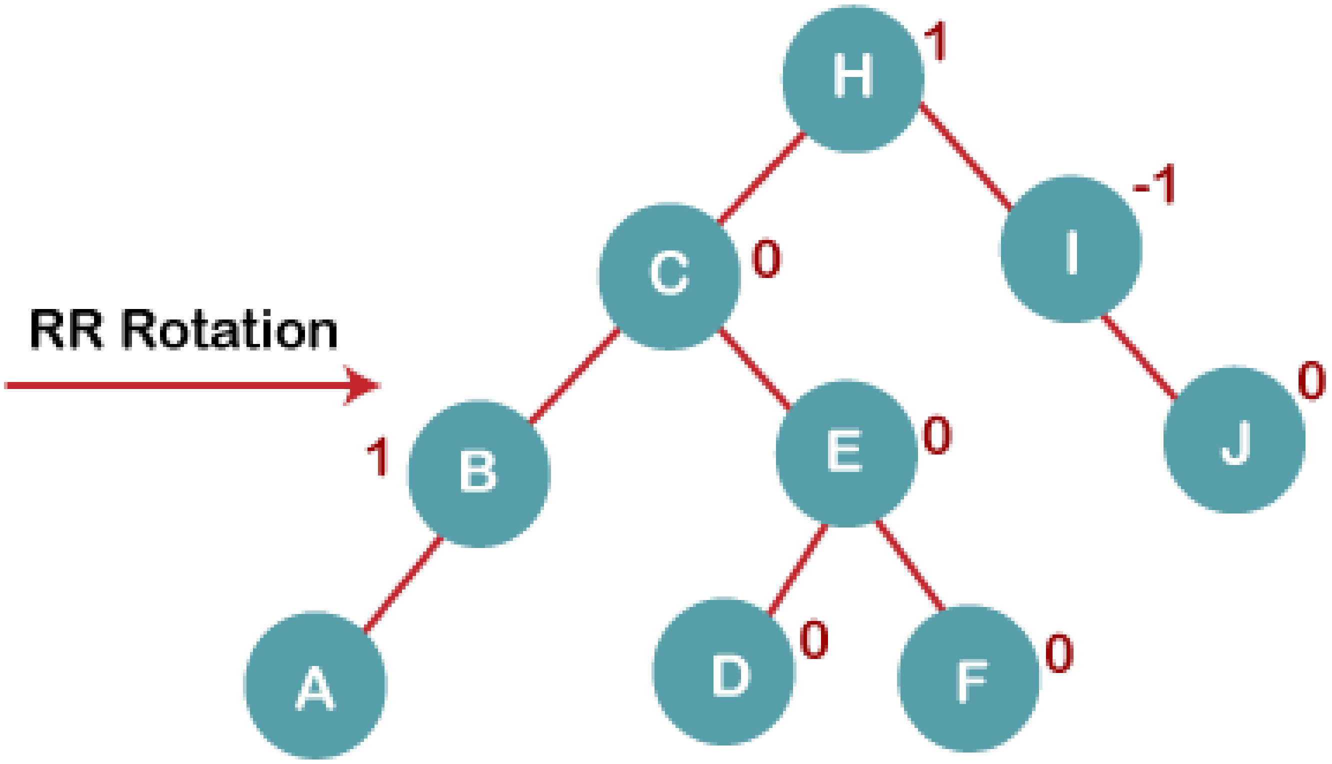
RL Rotation

0

LL + RR

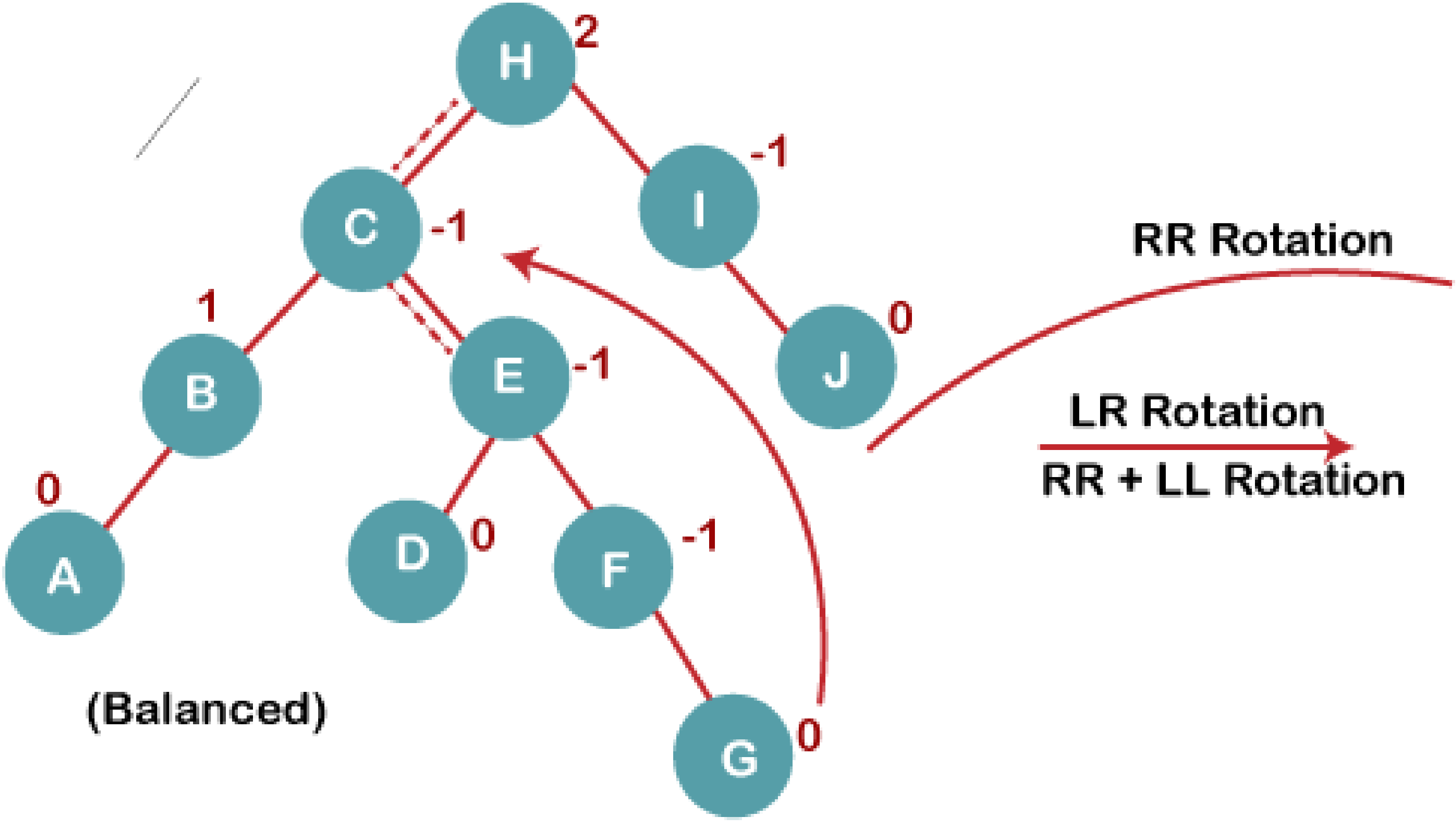
4b) We then perform RR rotation on node B

The resultant balanced tree after RR rotation is:



(Balanced)

5. Insert G

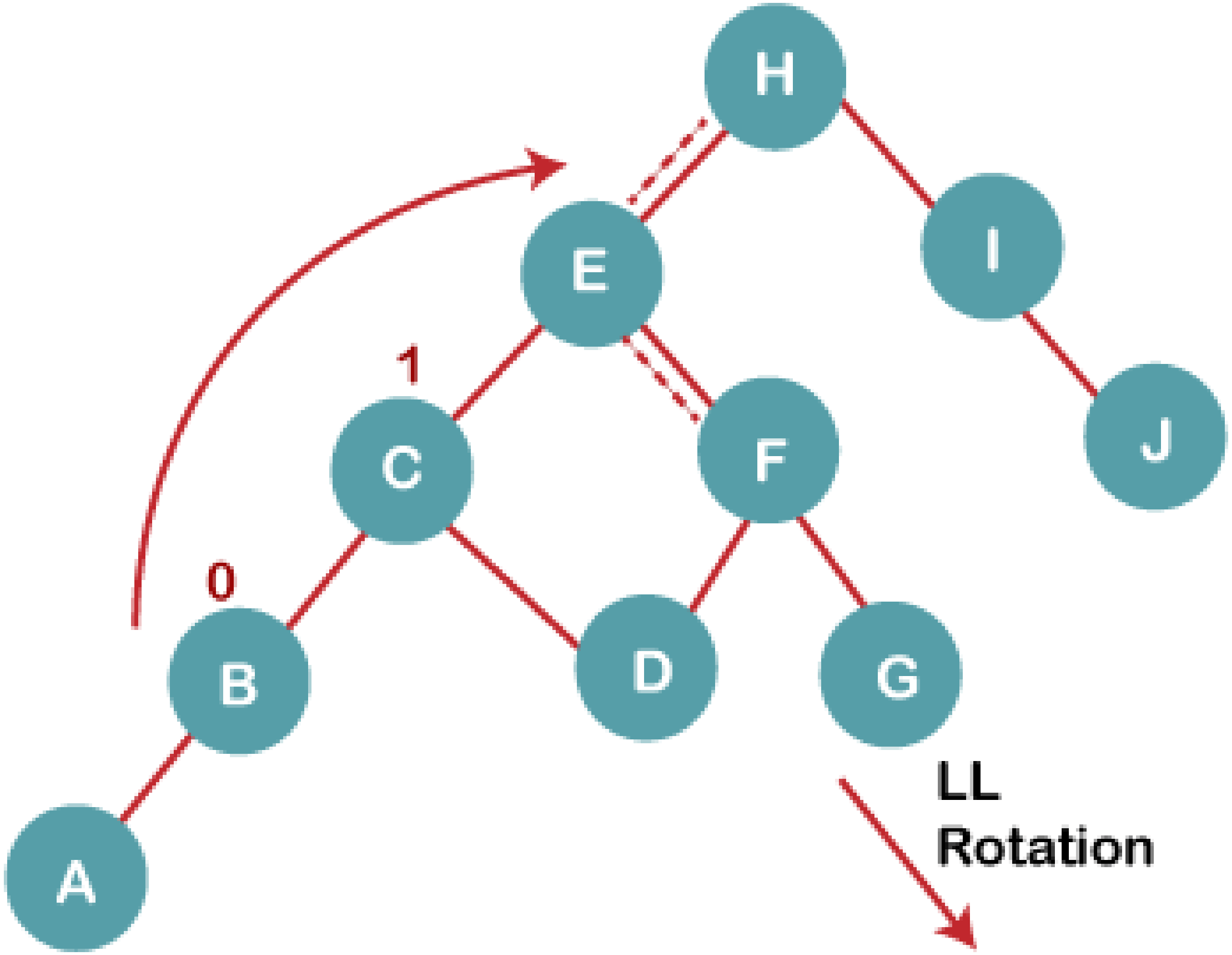


H
2
C
-1
I
-1
RR Rotation
1
B
E
-1
J
0
LR Rotation
RR + LL Rotation
0
A
D
0
F
-1
(Balanced)
G
0

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

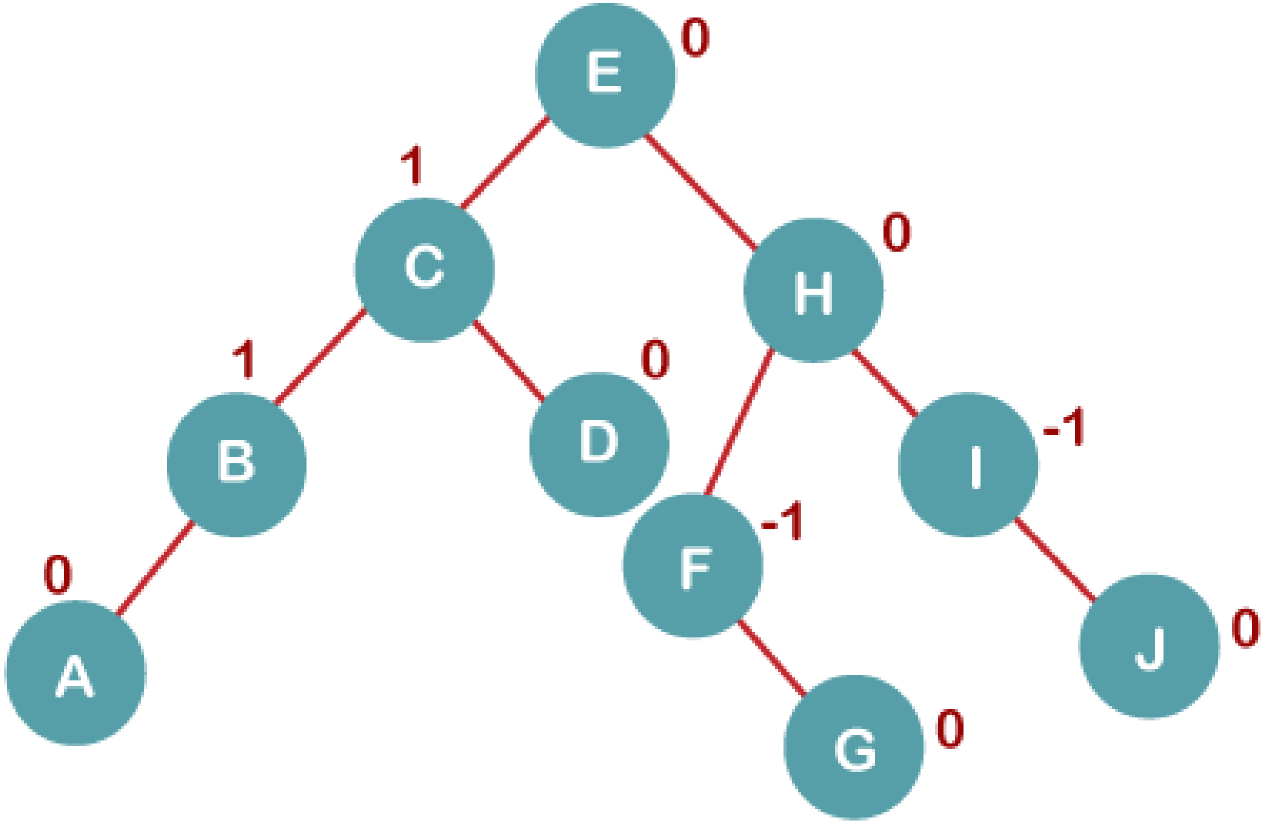
5 a) We first perform RR rotation on node C

The resultant tree after RR rotation is:


H
E
I
1
C
F
J
0
B
D
G
A
LL
Rotation

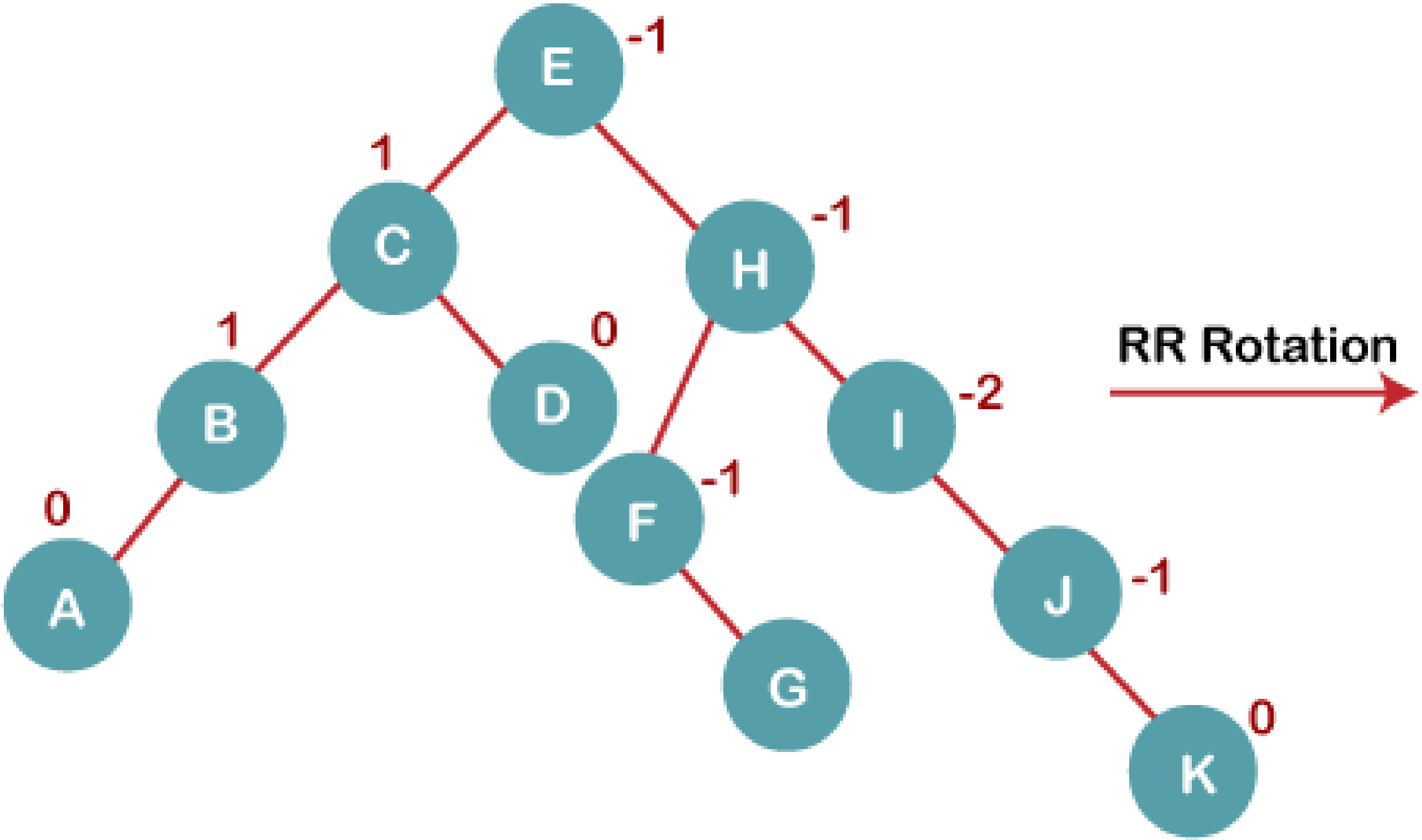
5 b) We then perform LL rotation on node H

The resultant balanced tree after LL rotation is:



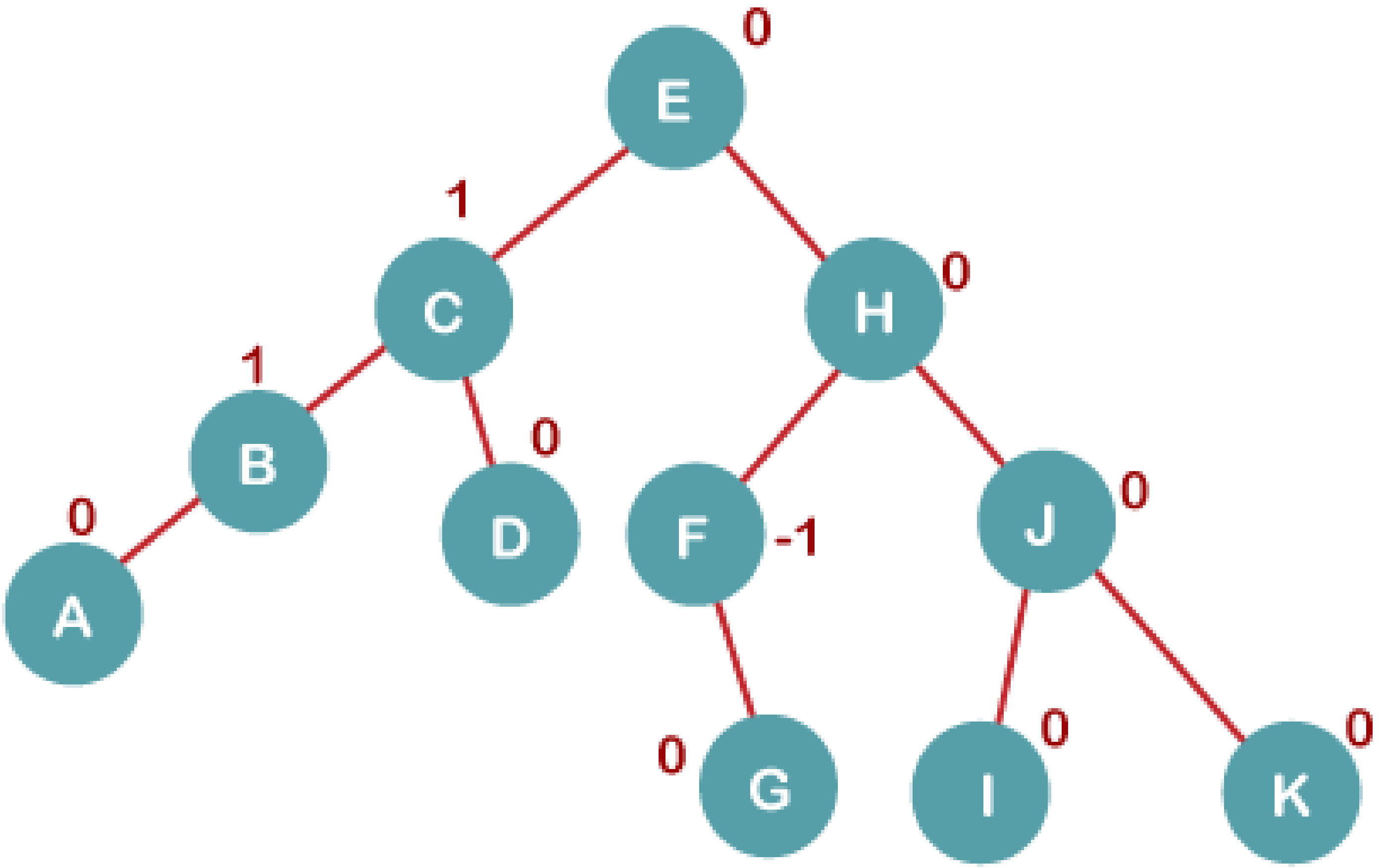
(Balanced)

6. Insert K



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

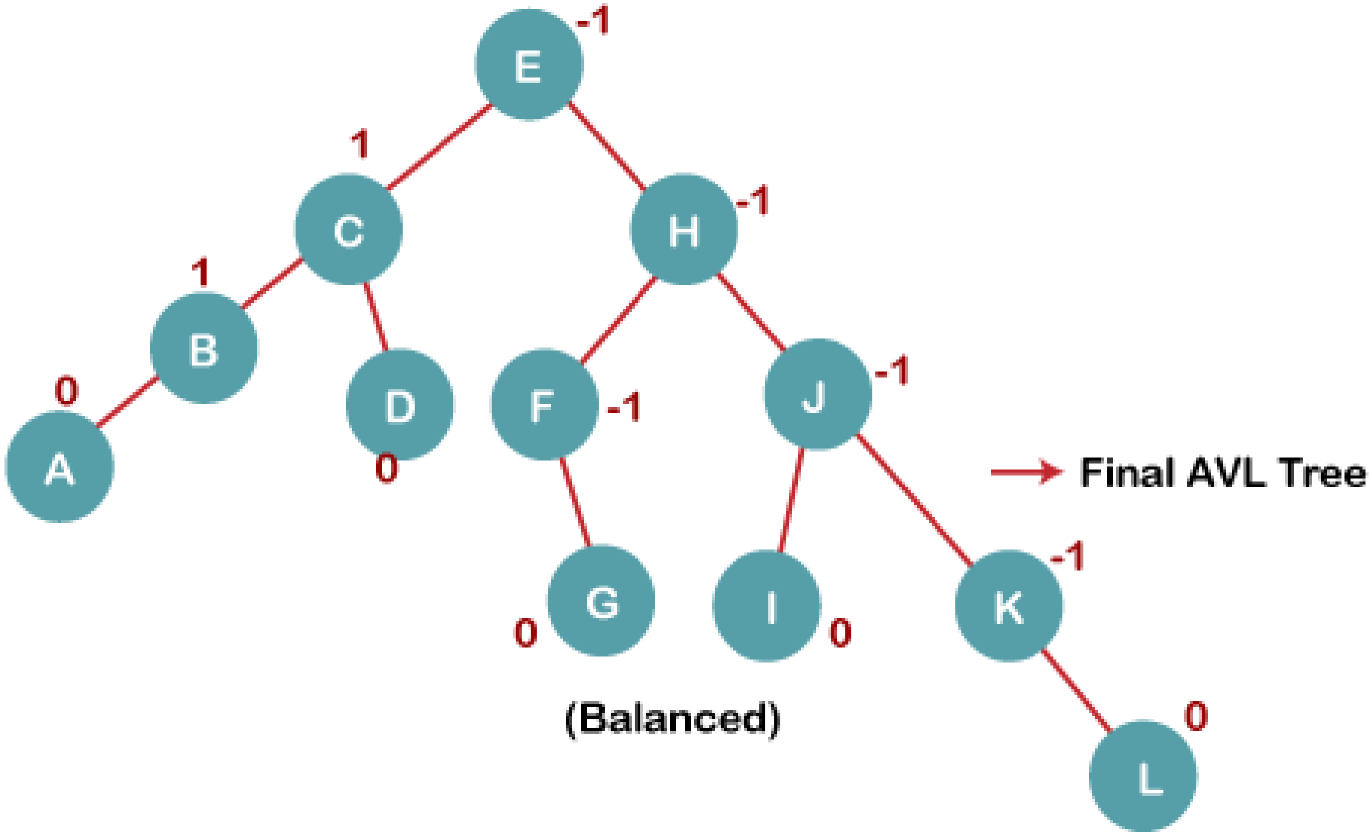
The resultant balanced tree after RR rotation is:



(Balanced)

7. Insert L

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



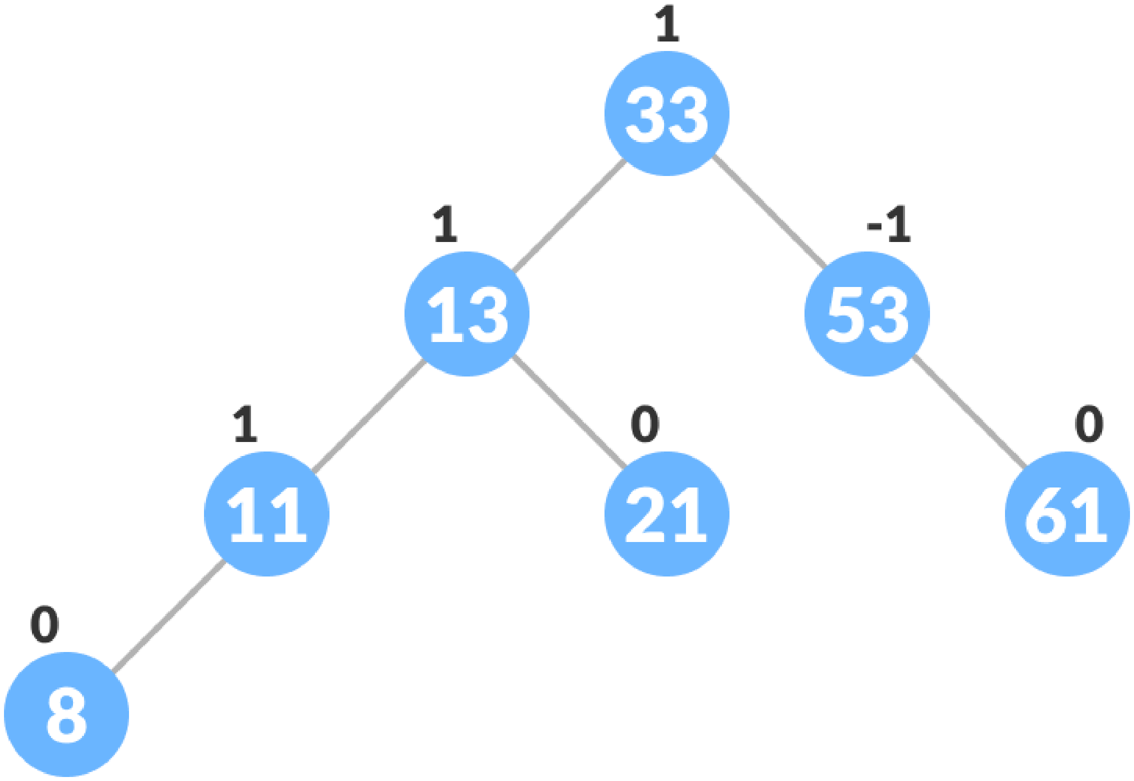
Link to create an AVL Tree

https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

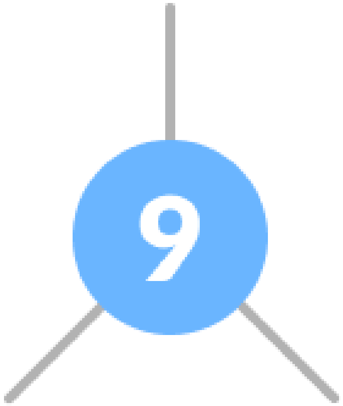
Algorithm to insert a newNode

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

Let the initial tree be:



Let the node to be inserted be:

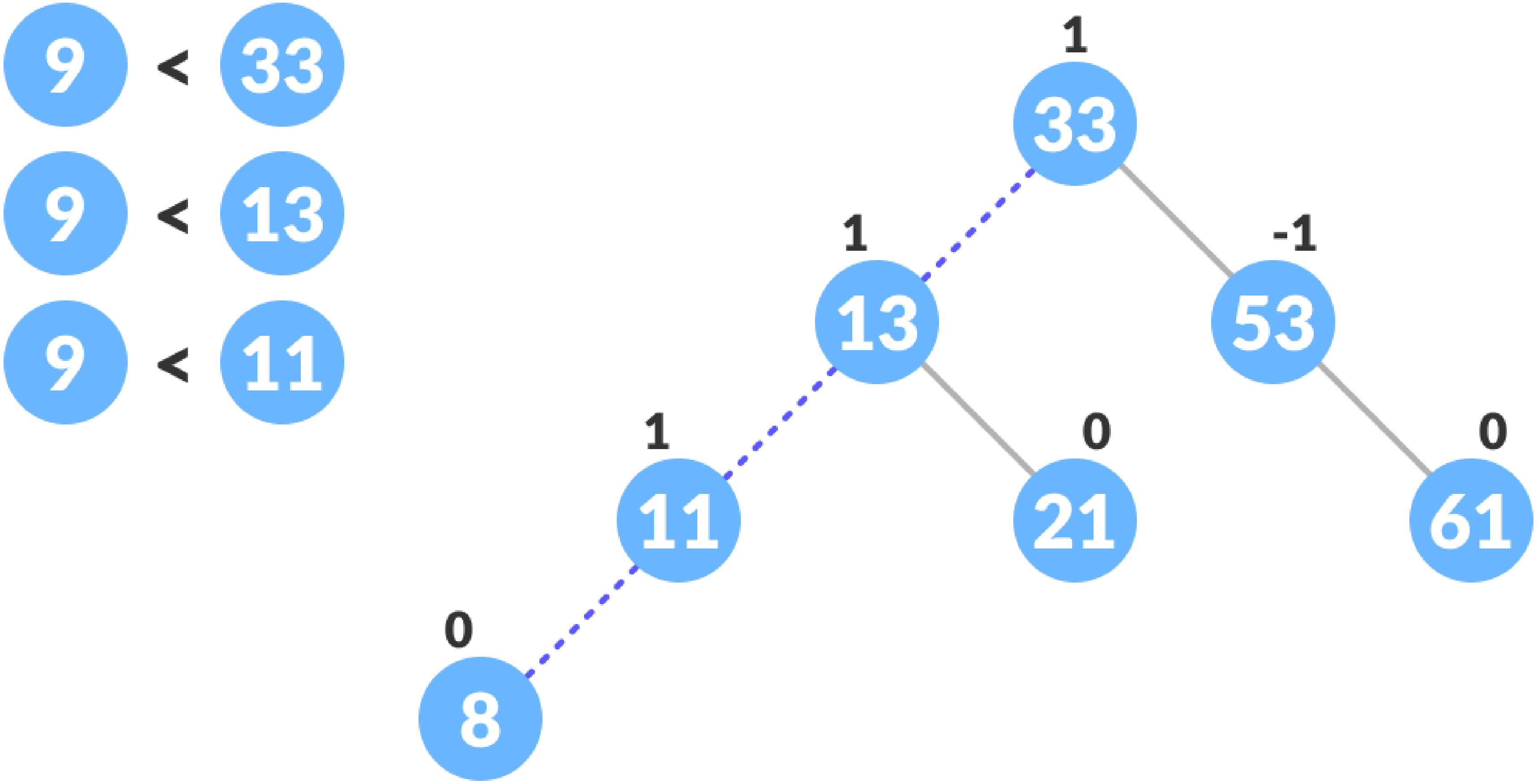


Go to the appropriate leaf node to insert a **newNode** using the following recursive steps. Compare **newKey** with **rootKey** of the current tree.

(a) If **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.

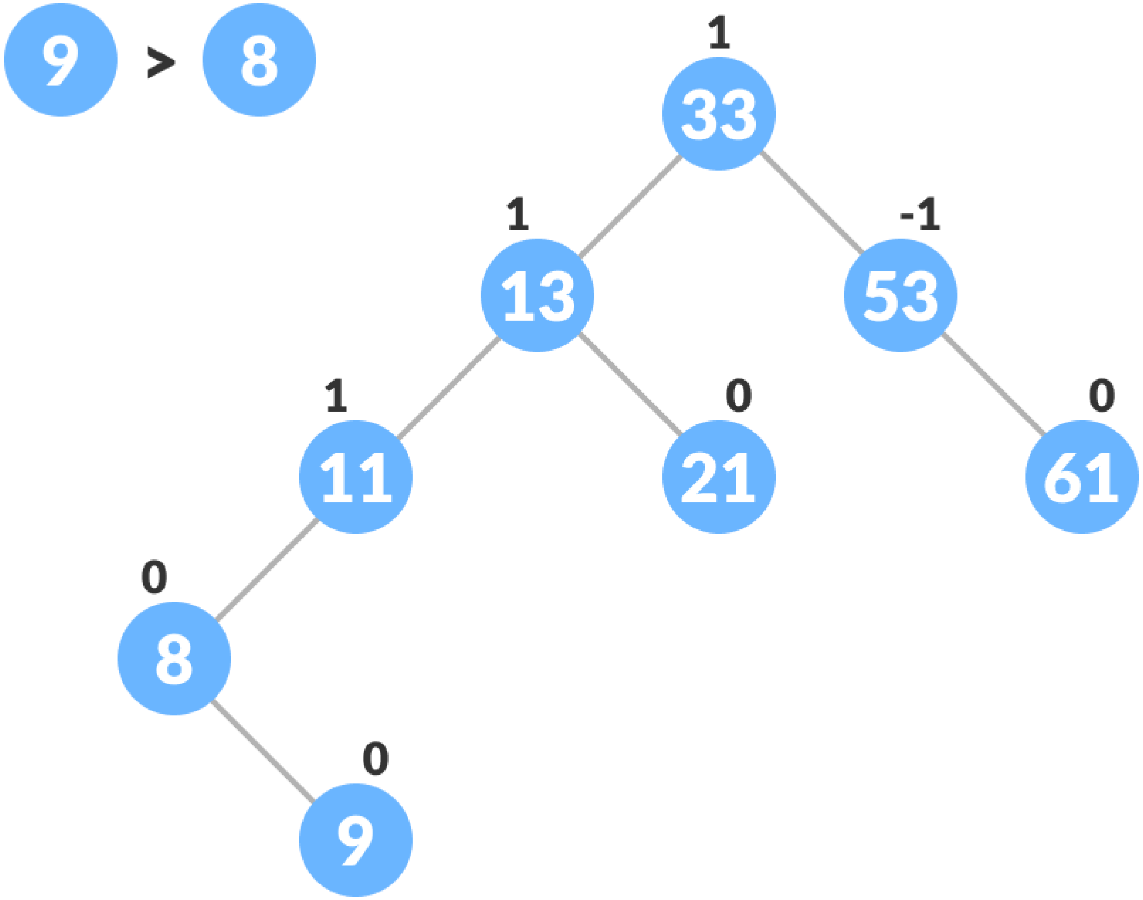
(c) Else, return **leafNode**.


9 < 33
9 < 13
9 < 11
1
33
1
13
-1
53
1
11
0
21
0
61
0
8

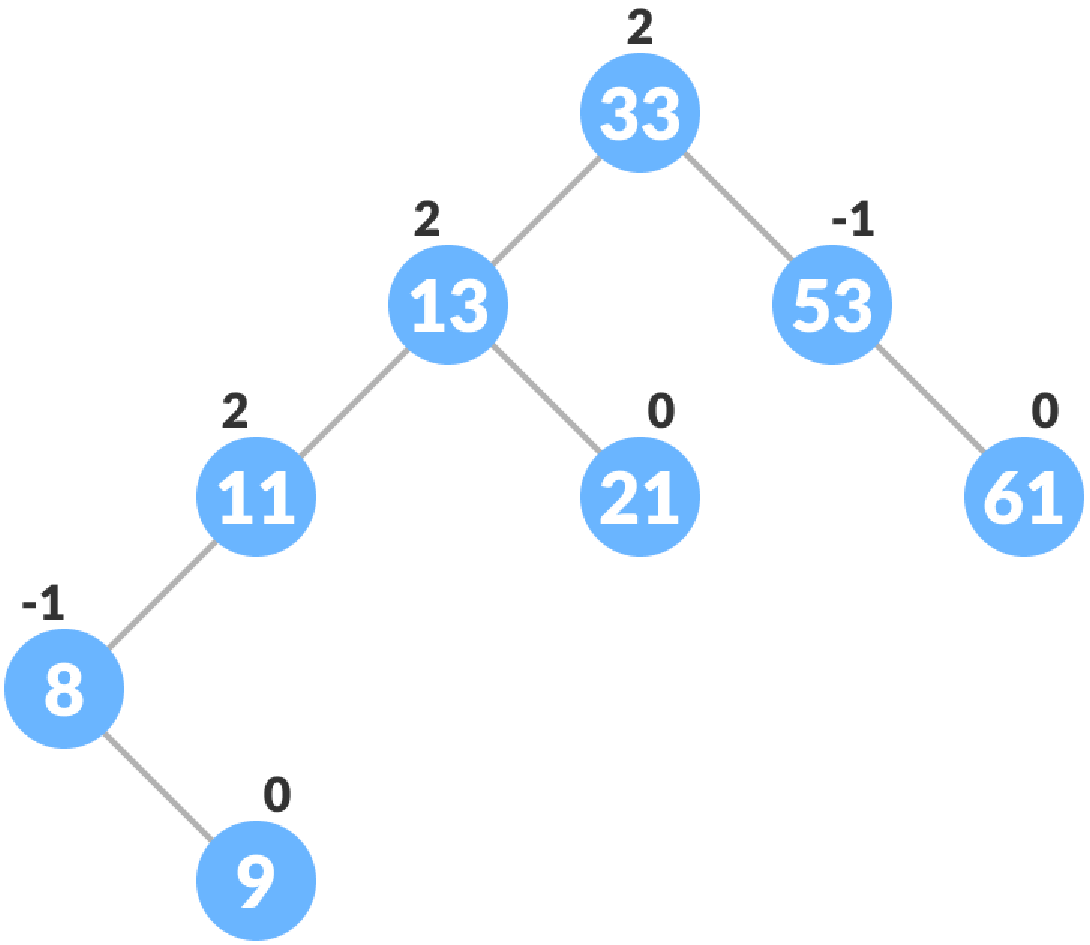
(3) Compare **leafKey** obtained from the above steps with **newKey**:

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

(b) Else, make **newNode** as **rightChild** of **leafNode**



(4) Update balanceFactor of the nodes.

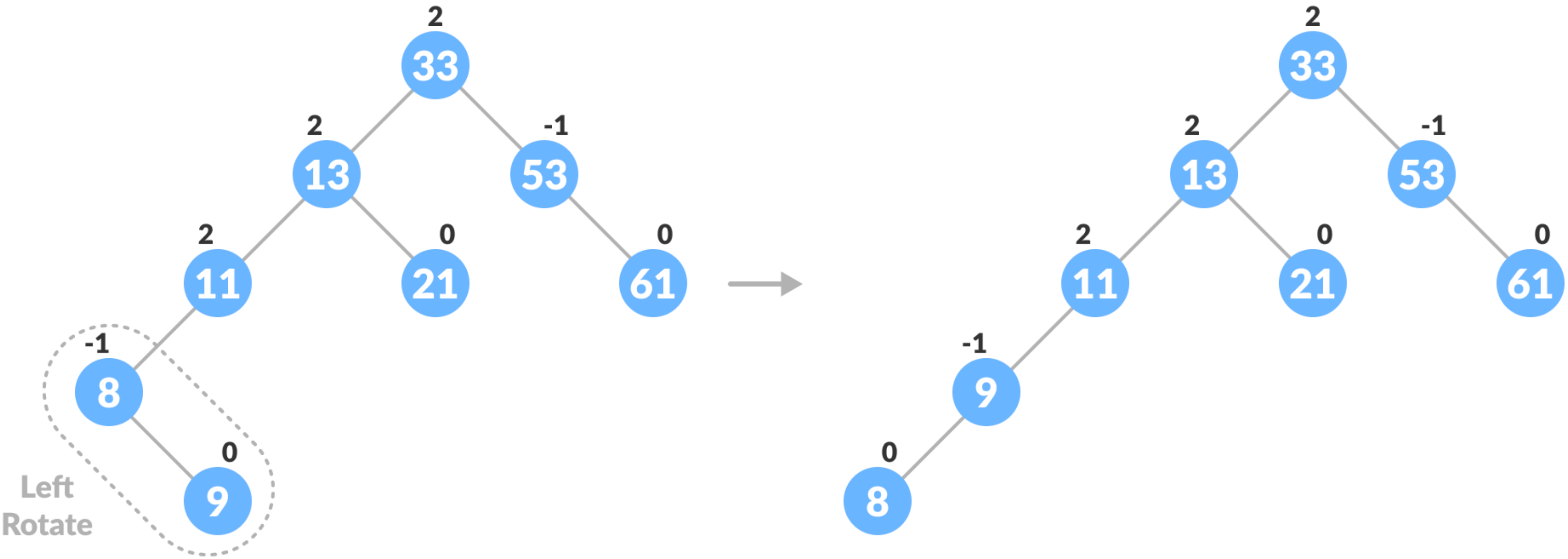


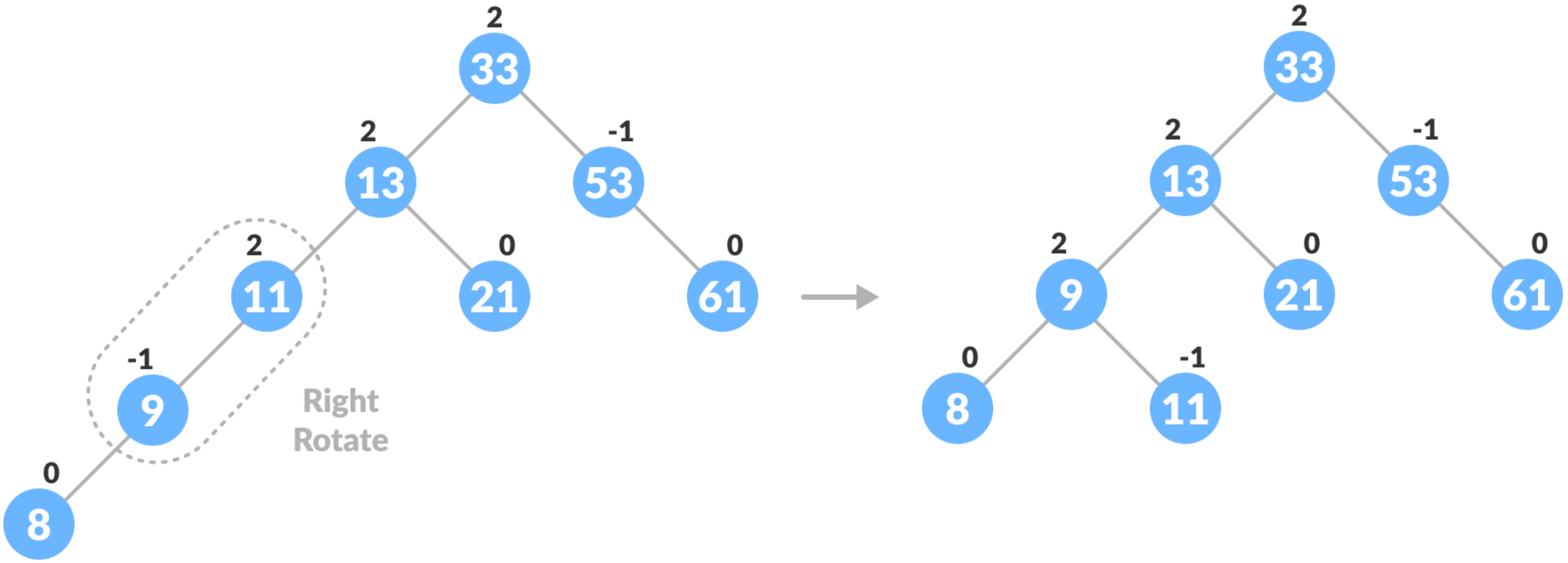
(5) If the nodes are unbalanced, then rebalance the node.

(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.



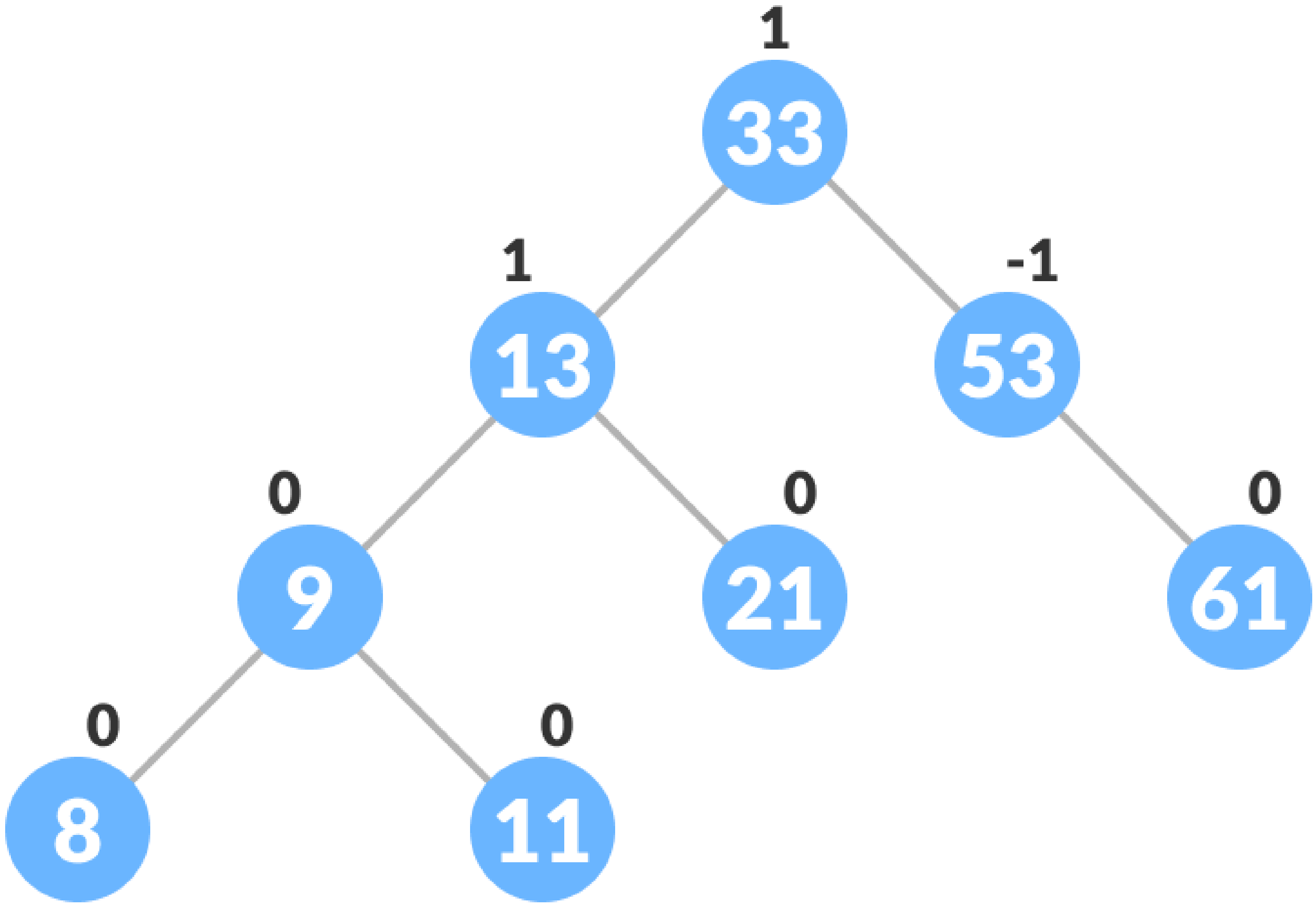


(b) If balanceFactor < -1, it means the height of the right subtree is greater than that of the left subtree. So, do right rotation or right-left rotation

(a) If newNodeKey > rightChildKey do left rotation.

(b) Else, do right-left rotation

6. The final tree is



Why AVL Trees?

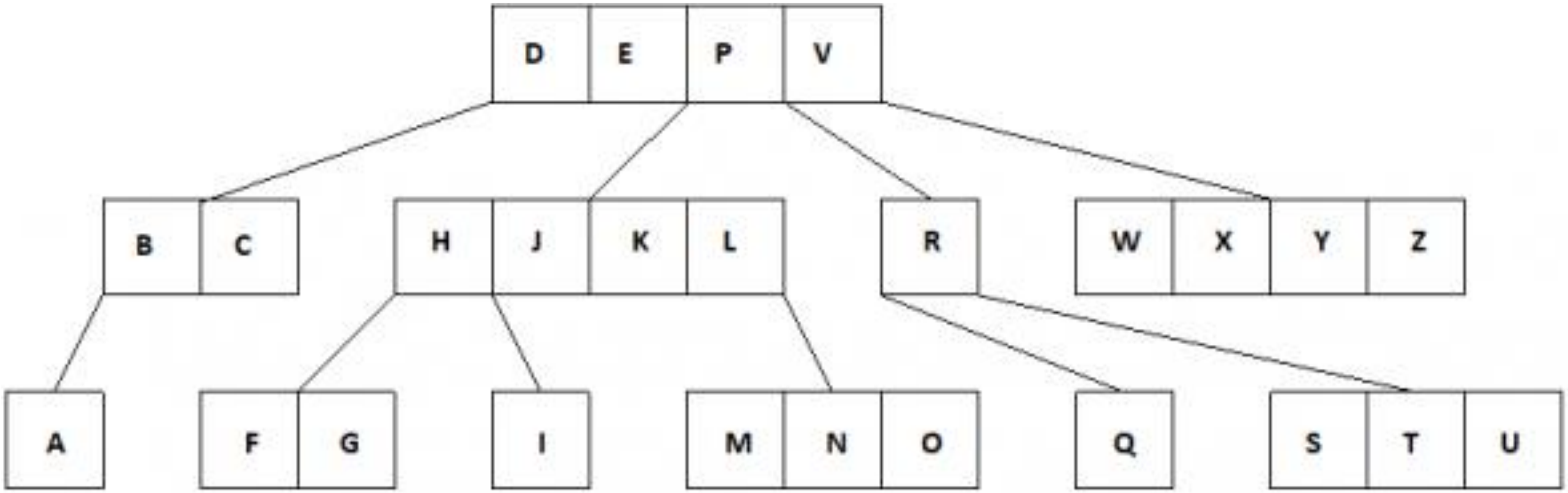
Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that height of the tree remains O(Logn) after every insertion and deletion, then we can guarantee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

Multi-Way Trees

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.

B-Trees

In search trees like binary search tree, AVL Tree, Red-Black tree, etc., every node contains only one value (key) and a maximum of two children. But there is a special type of search tree called B-Tree in which a node contains more than one value (key) and more than two children. B-Tree was developed in the year 1972 by **Bayer and McCreight** with the name ***Height Balanced m-way Search Tree***. Later it was named as B-Tree.

- B Tree is a specialized m-way tree that can be widely used for disk access. A B-Tree of order m can have at most m-1 keys and m children. One of the main reason of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.
- A B-tree is a tree data structure that keeps data sorted and allows searches, insertions, and deletions in logarithmic amortized time. Unlike self-balancing binary search trees, it is optimized for systems that read and write large blocks of data. It is most commonly used in database and file systems.

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

B-Tree of Order m has the following properties...

Property #1 - All **leaf nodes** must be **at same level**.

Property #2 - All nodes except root must have at least **[m/2]-1** keys and maximum of **m-1** keys.

Property #3 - All non leaf nodes except root (i.e. all internal nodes) must have at least **m/2** children.

Property #4 - If the root node is a non leaf node, then it must have **atleast 2** children.

Property #5 - A non leaf node with **n-1** keys must have **n** number of children.

Property #6 - All the **key values in a node** must be in **Ascending Order**.

A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.

- Every node in a B-Tree contains at most m children.
- Every node in a B-Tree except the root node and the leaf node contain at least m/2 children.
- The root nodes must have at least 2 nodes.
- All leaf nodes must be at the same level.

Important properties of a B-tree:

- B-tree nodes have many more than two children.
- A B-tree node may contain more than just a single element.

- It is not necessary that, all the nodes contain the same number of children but, each node must have m/2 number of nodes.
- A B tree of order 4 is shown in the following image.

The set formulation of the B-tree rules:

Every B-tree depends on a positive constant integer called MINIMUM, which is used to determine how many elements are held in a single node.

Rule 1: The root can have as few as one element (or even no elements if it also has no children); every other node has at least MINIMUM elements.

Rule 2: The maximum number of elements in a node is twice the value of MINIMUM.

Rule 3: The elements of each B-tree node are stored in a partially filled array, sorted from the smallest element (at index 0) to the largest element (at the final used position of the array).

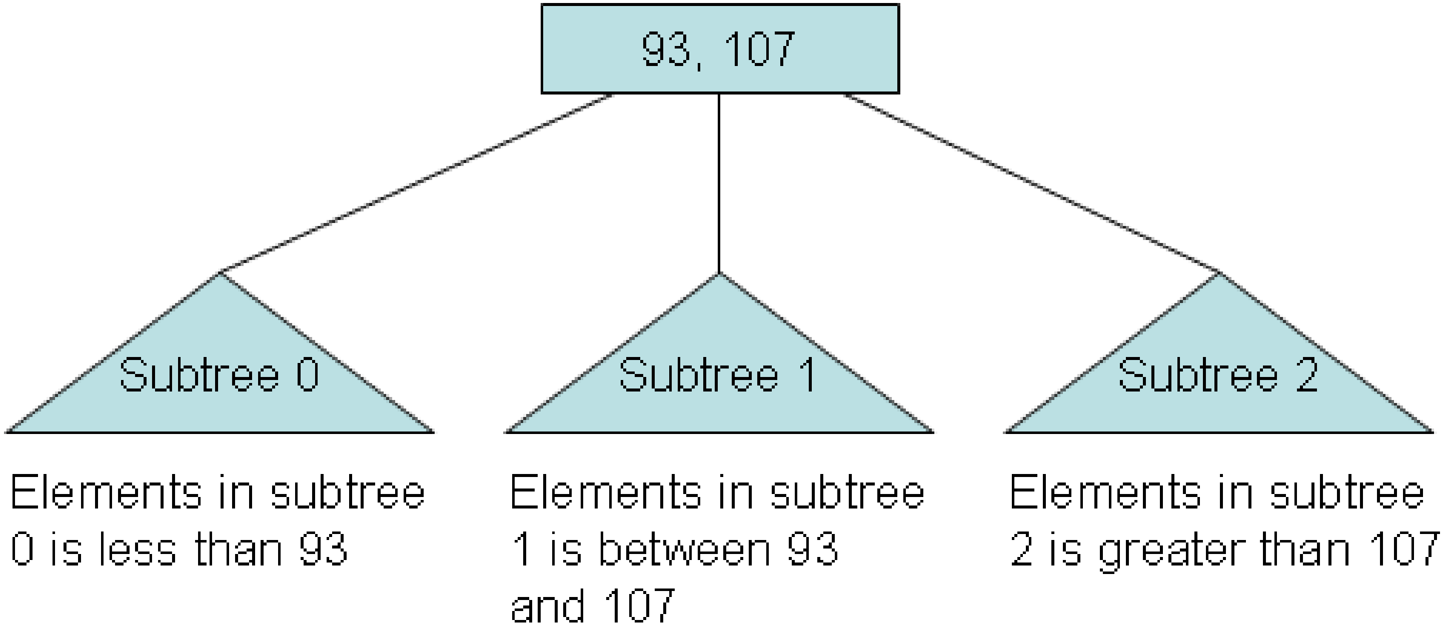
Rule 4: The number of subtrees below a nonleaf node is always one more than the number of elements in the node.

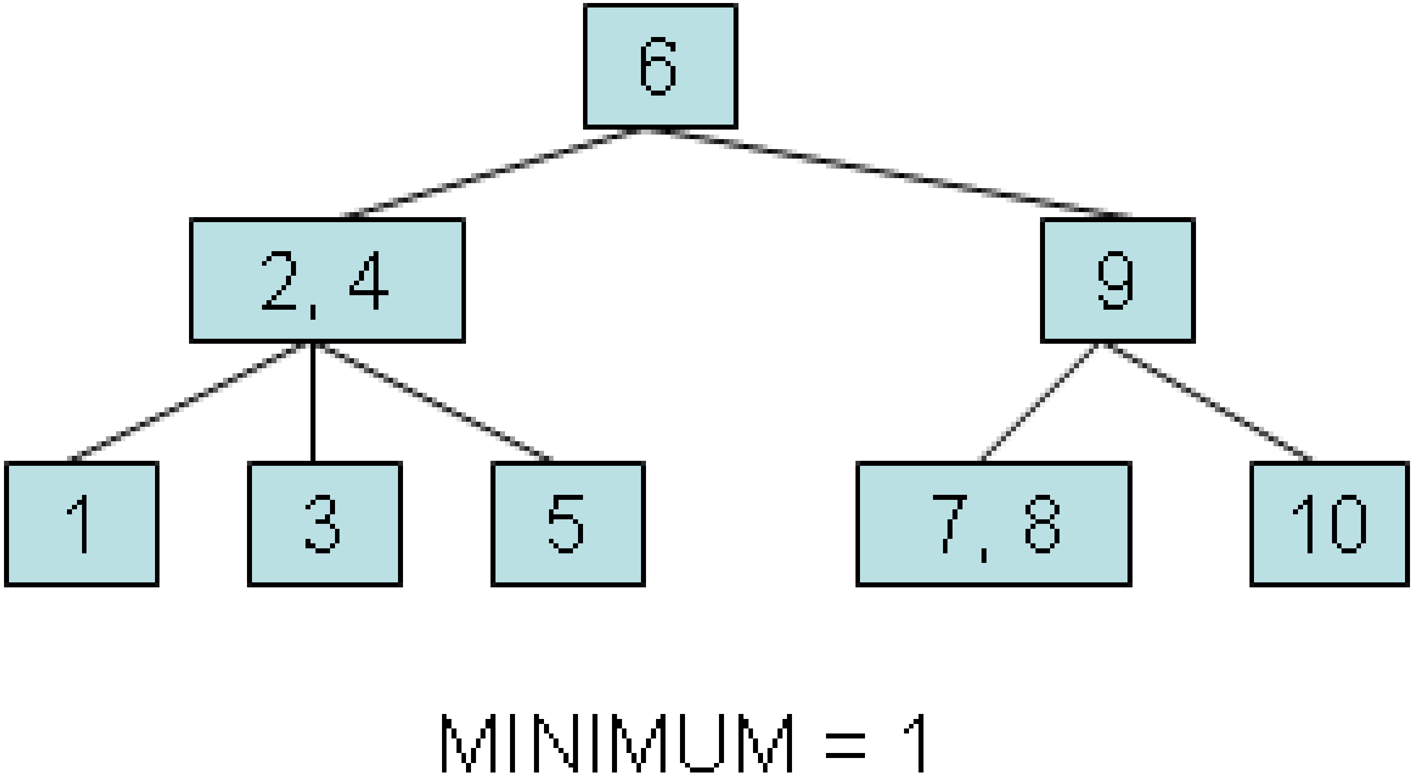
- Subtree 0, subtree 1, ...

Rule 5: For any nonleaf node:

- An element at index *i* is greater than all the elements in subtree number *i* of the node, and
- An element at index *i* is less than all the elements in subtree number *i* + 1 of the node.

Rule 6: Every leaf in a B-tree has the same depth. Thus it ensures that a B-tree avoids the problem of a unbalanced tree.


93, 107
Subtree 0
Subtree 1
Subtree 2
Elements in subtree 0 is less than 93
Elements in subtree 1 is between 93 and 107
Elements in subtree 2 is greater than 107



https://www.cs.usfca.edu/~galles/visualization/BTree.html

Insertion Operation in B-Tree

In a B-Tree, a new element must be added only at the leaf node. That means, the new keyValue is always attached to the leaf node only. The insertion operation is performed as follows...

Step 1 - Check whether tree is Empty.

Step 2 - If tree is **Empty**, then create a new node with new key value and insert it into the tree as a root node.

Step 3 - If tree is **Not Empty**, then find the suitable leaf node to which the new key value is added using Binary Search Tree logic.

Step 4 - If that leaf node has empty position, add the new key value to that leaf node in ascending order of key value within the node.

Step 5 - If that leaf node is already full, **split** that leaf node by sending middle value to its parent node. Repeat the same until the sending value is fixed into a node.

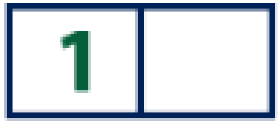
Step 6 - If the spilting is performed at root node then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Example

Construct a **B-Tree of Order 3** by inserting numbers from 1 to 10.

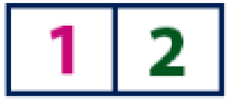
insert(1)

Since '1' is the first element into the tree that is inserted into a new node. It acts as the root node.



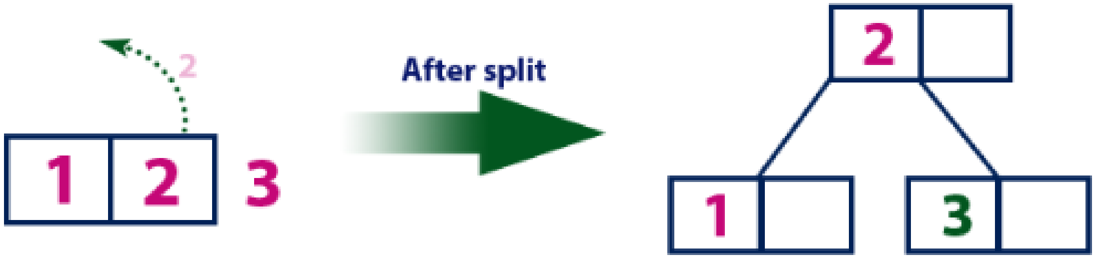
insert(2)

Element '2' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node has an empty position. So, new element (2) can be inserted at that empty position.



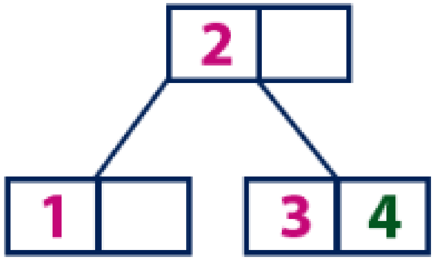
insert(3)

Element '3' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node doesn't has an empty position. So, we split that node by sending middle value (2) to its parent node. But here, this node doesn't has parent. So, this middle value becomes a new root node for the tree.



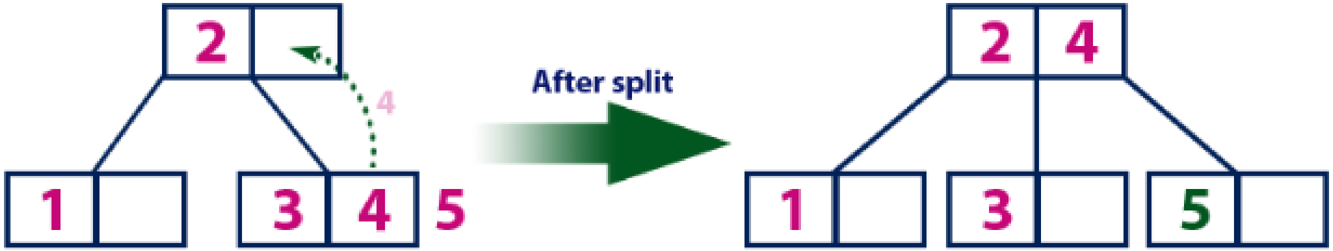
insert(4)

Element '4' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node with value '3' and it has an empty position. So, new element (4) can be inserted at that empty position.



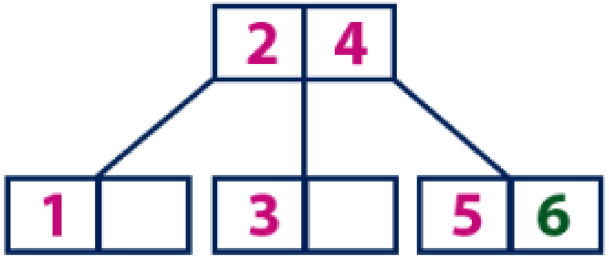
insert(5)

Element '5' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (4) to its parent node (2). There is an empty position in its parent node. So, value '4' is added to node with value '2' and new element '5' added as new leaf node.



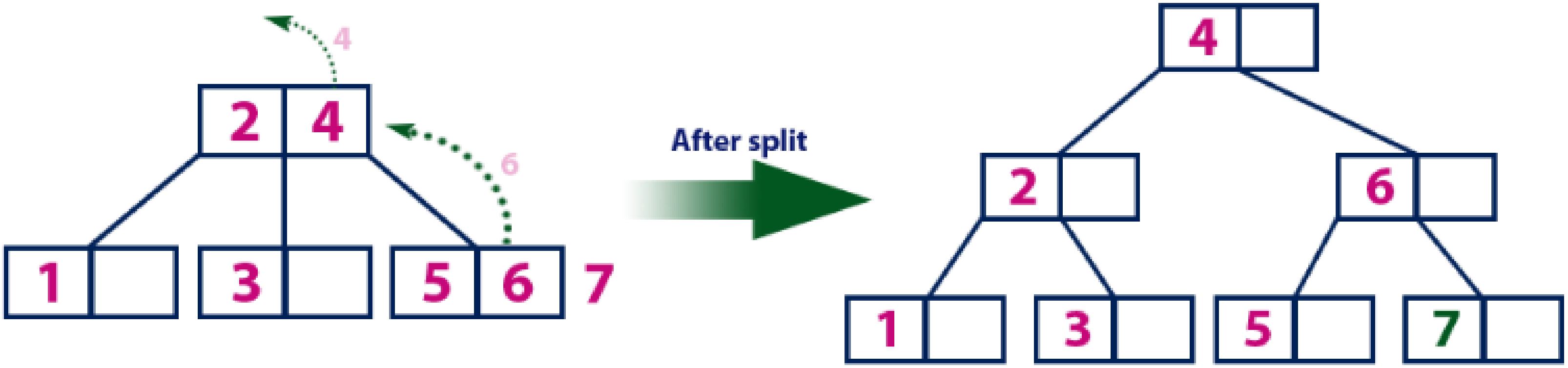
insert(6)

Element '6' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node with value '5' and it has an empty position. So, new element (6) can be inserted at that empty position.



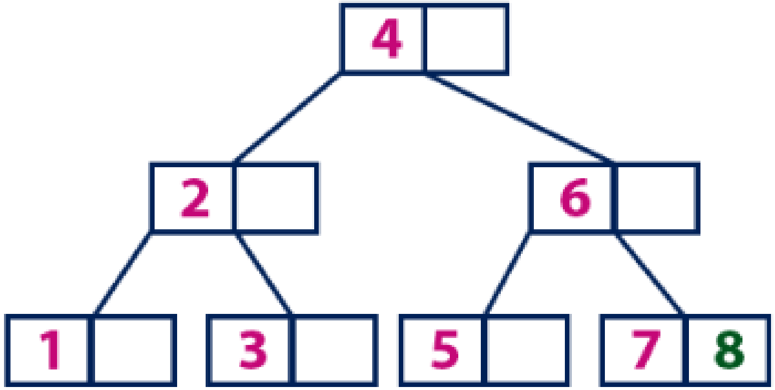
insert(7)

Element '7' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (6) to its parent node (2&4). But the parent (2&4) is also full. So, again we split the node (2&4) by sending middle value '4' to its parent but this node doesn't have parent. So, the element '4' becomes new root node for the tree.



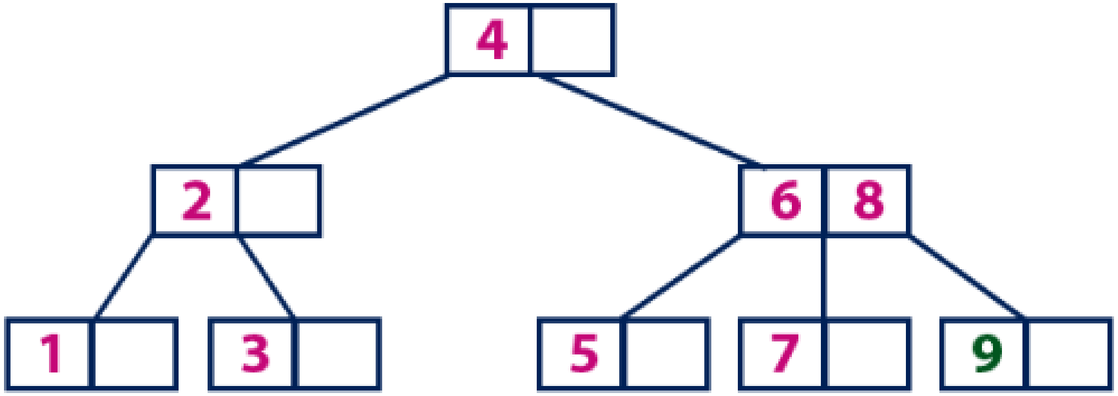
insert(8)

Element '8' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '8' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7) and it has an empty position. So, new element (8) can be inserted at that empty position.



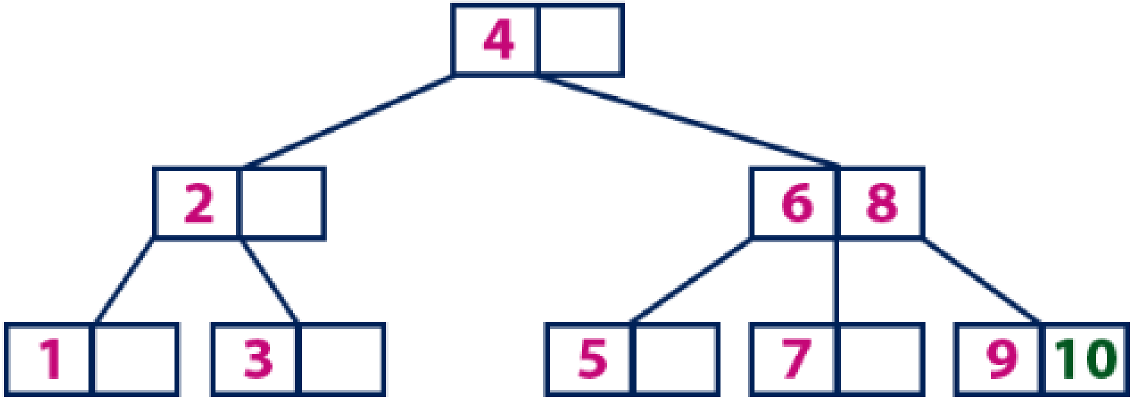
insert(9)

Element '9' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '9' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7 & 8). This leaf node is already full. So, we split this node by sending middle value (8) to its parent node. The parent node (6) has an empty position. So, '8' is added at that position. And new element is added as a new leaf node.

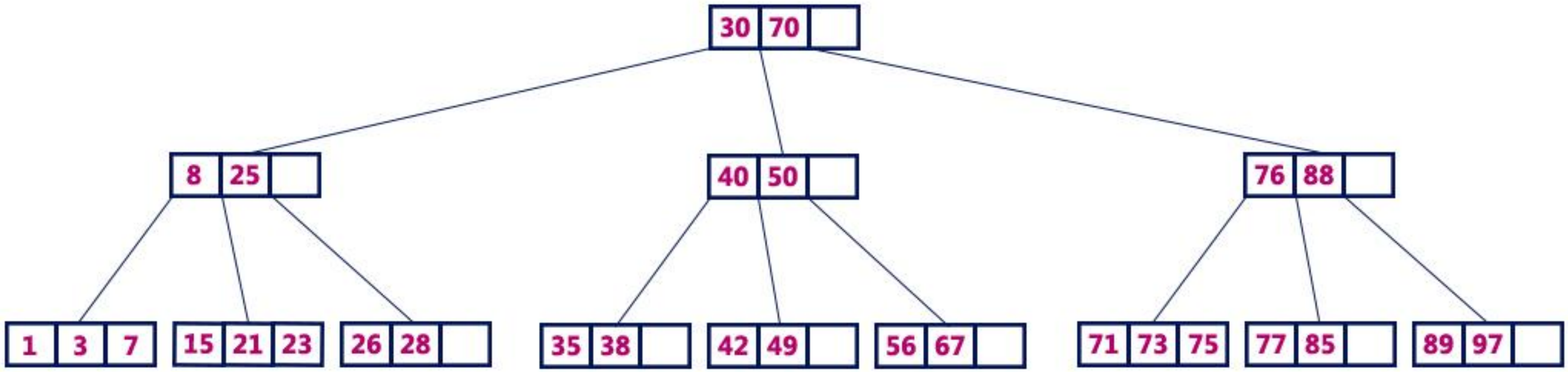


insert(10)

Element ‘10’ is larger than root node ‘4’ and it is not a leaf node. So, we move to the right of ‘4’. We reach to a node with values ‘6 & 8 ’. ‘10’ is larger than ‘6 & 8’ and it is also not a leaf node. So, we move to the right of ‘8’. We reach to a leaf node (9). This leaf node has an empty position. So, new element ‘10’ is added at that empty position.



B-Tree of Order 4

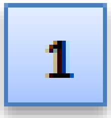


Example

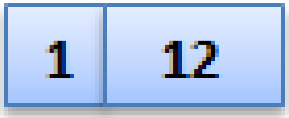
key :- 1,12,8,2,25,6,14,28,17,7,52,16,48,68,3,26,29,53,55,45,67.

Order = 5

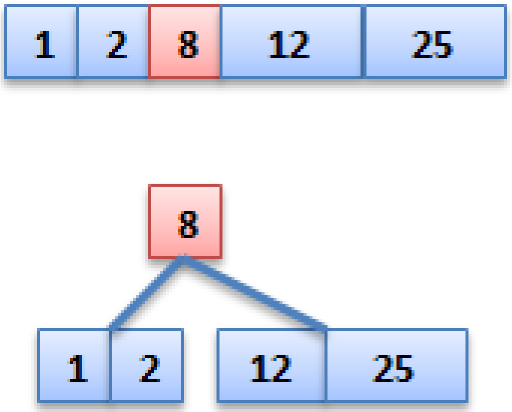
Step1. Add first key as root node.



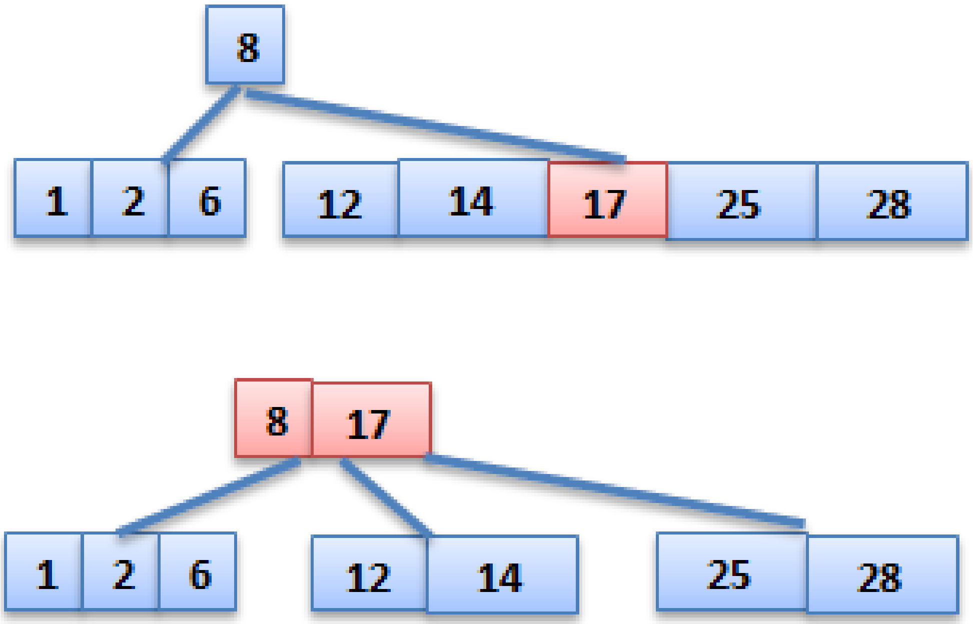
Step2. Add next key at the appropriate place in sorted order.

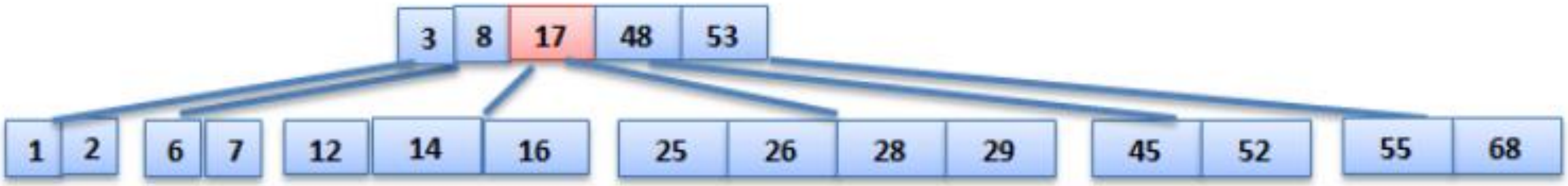


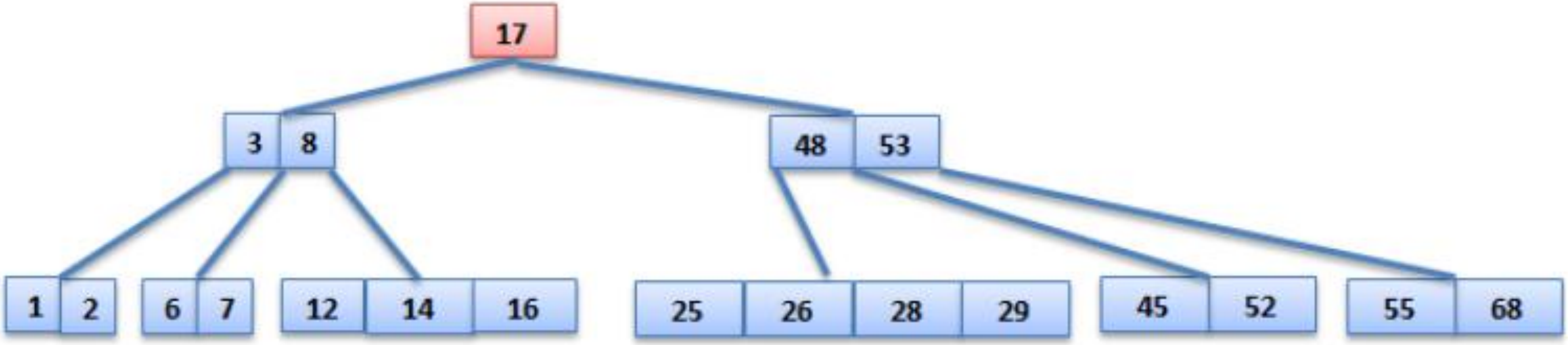
Step3. Same process applied until root node full. if root node full then spliting process applied.



Some importent steps




3 8 17 48 53
1 2
6 7
12 14 16
25 26 28 29
45 52
55 68

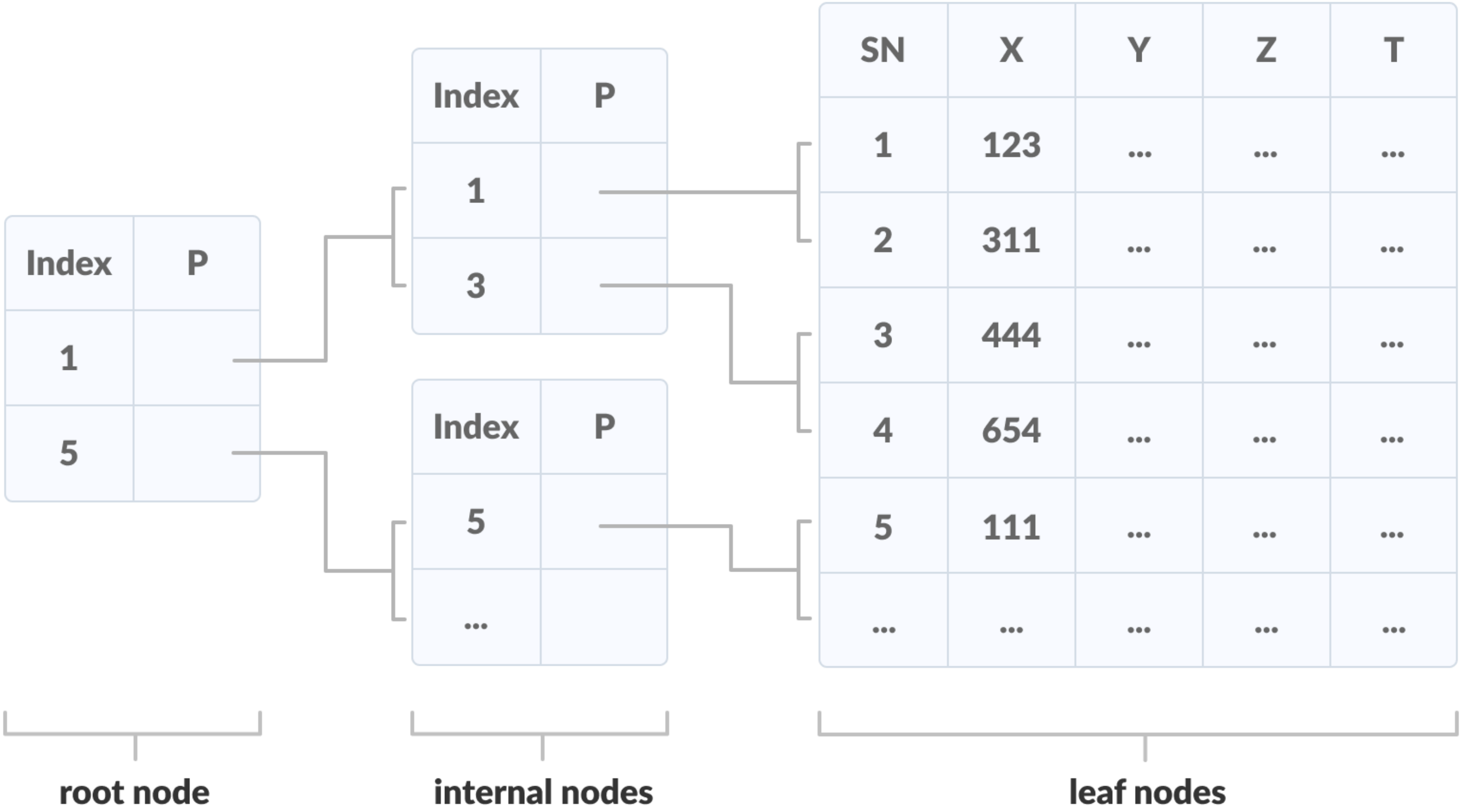

17
3 8
48 53
1 2
6 7
12 14 16
25 26 28 29
45 52
55 68

B+ Tree

- A B+ tree is an advanced form of a self-balancing tree in which all the values are present in the leaf level.
- B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.
- In B Tree, Keys and records both can be stored in the internal as well as leaf nodes. Whereas, in B+ tree, records (data) can only be stored on the leaf nodes while internal nodes can only store the key values.

- The leaf nodes of a B+ tree are linked together in the form of a singly linked lists to make the search queries more efficient.
- B+ Tree are used to store the large amount of data which can not be stored in the main memory. Due to the fact that, size of main memory is always limited, the internal nodes (keys to access records) of the B+ tree are stored in the main memory whereas, leaf nodes are stored in the secondary memory.

In multilevel indexing, the index of indices is created as in figure below. It makes accessing the data easier and faster.



Index	P
1	
5	

Index	P
1	
3	

Index	P
5	
...	

SN	X	Y	Z	T
1	123	...	...	...
2	311	...	...	...
3	444	...	...	...
4	654	...	...	...
5	111	...	...	...
...	...	...	...	...

root node

internal nodes

leaf nodes

Advantages of B+ Tree

- Records can be fetched in equal number of disk accesses.
- Height of the tree remains balanced and less as compare to B tree.
- We can access the data stored in a B+ tree sequentially as well as directly.
- Keys are used for indexing.
- Faster search queries as the data is stored only on the leaf nodes.

B Tree VS B+ Tree

SN	B Tree	B+ Tree
1	Search keys can not be repeatedly stored.	Redundant search keys can be present.
2	Data can be stored in leaf nodes as well as internal nodes	Data can only be stored on the leaf nodes.
3	Searching for some data is a slower process since data can be found on internal nodes as well as on the leaf nodes.	Searching is comparatively faster as data can only be found on the leaf nodes.
4	Deletion of internal nodes are so complicated and time consuming.	Deletion will never be a complexed process since element will always be deleted from the leaf nodes.
5	Leaf nodes can not be linked together.	Leaf nodes are linked together to make the search operations more efficient.

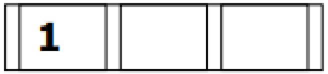
- A balanced tree
- Each node can have at most *m* key fields and *m+1* pointer fields
- Half-full must be satisfied (except root node):
- m is even and m=2d
  - Leaf node half full: at least d entries
  - Non-leaf node half full: at least d entries
- m is odd and m = 2d+1
  - Leaf node half full: at least d+1 entries
  - Non-leaf node half full: at least d entries (i.e., d+1 pointers)

Show the tree after insertions

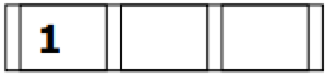
- Suppose each B+-tree node can hold up to 4 pointers and 3 keys.
- m=3 (odd), d=1
- Half-full (for odd m value)
  - Leaf node, at least 2 (d+1) entries
  - Non-leaf nodes, at least 2 (d+1) pointers (1 entry)
- Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10

Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10

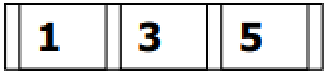
- Insert 1



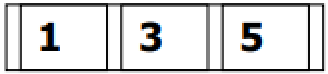
Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10



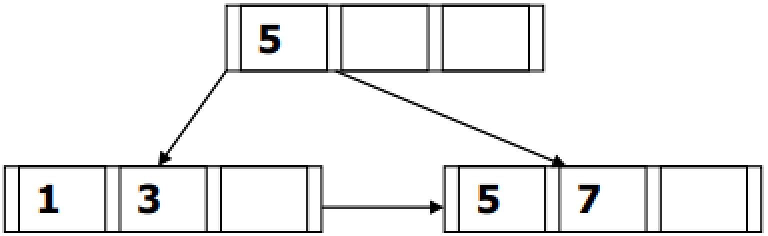
- Insert 3, 5



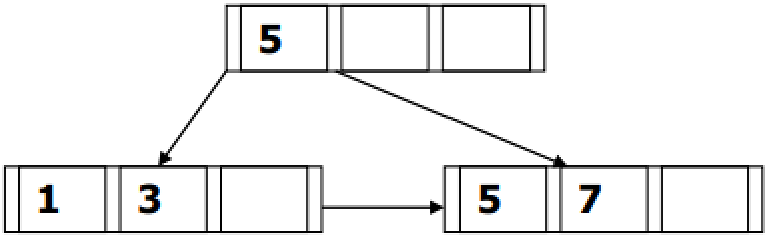
Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10



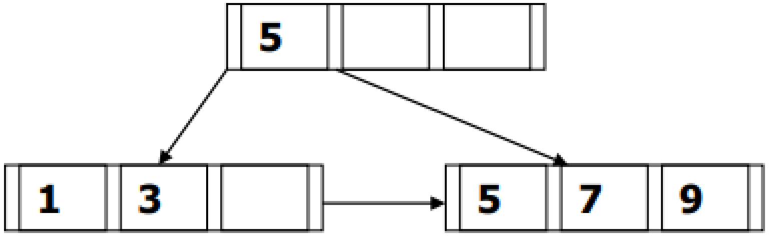
- Insert 7



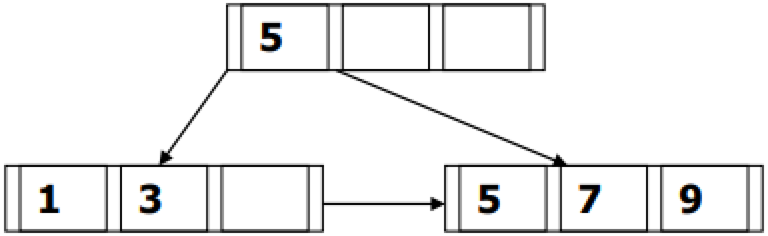
Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10



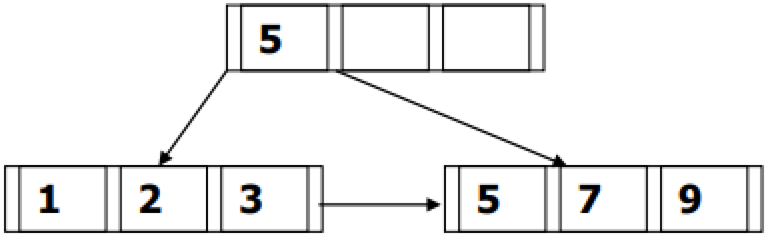
- Insert 9



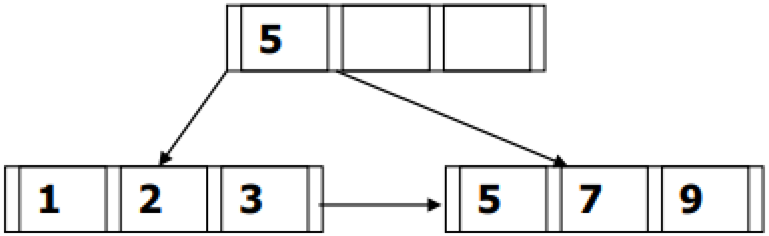
Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10



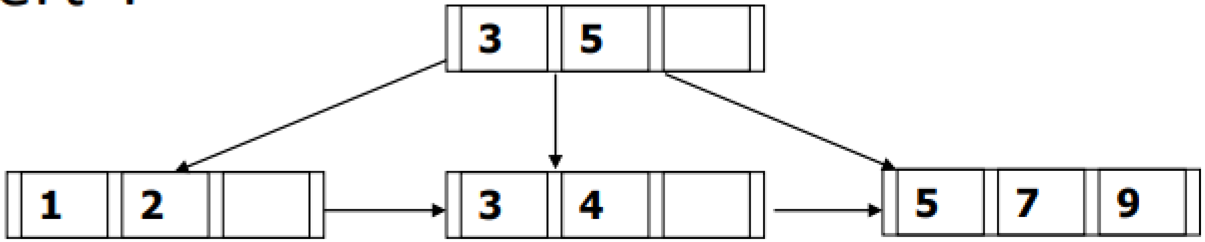
- Insert 2



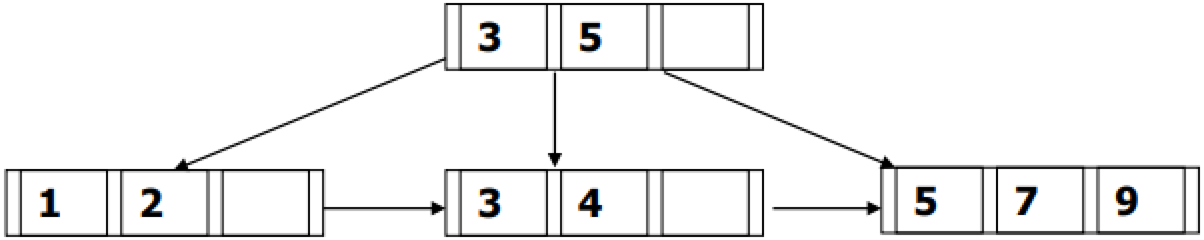
Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10



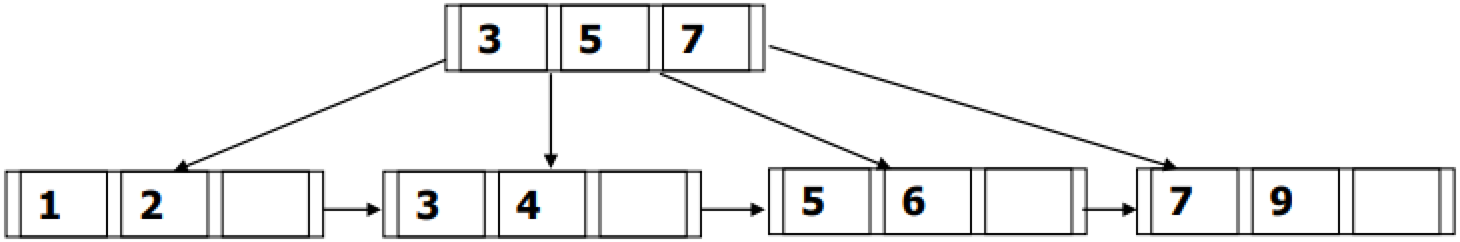
- Insert 4



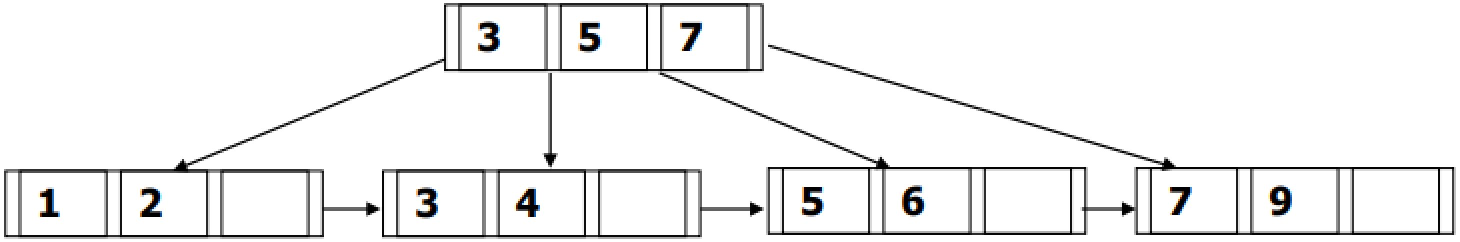
Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10



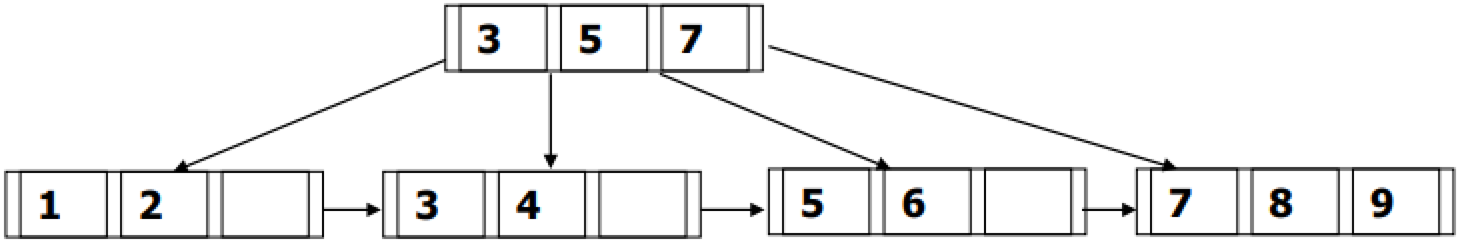
- Insert 6



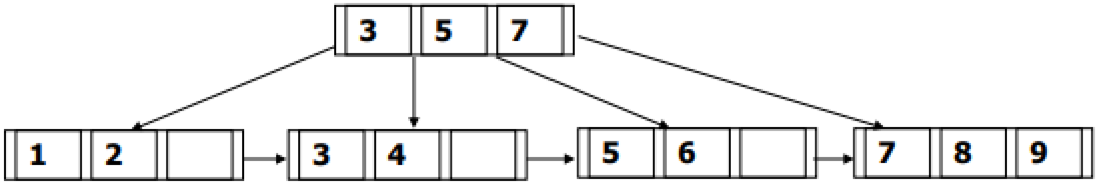
Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10



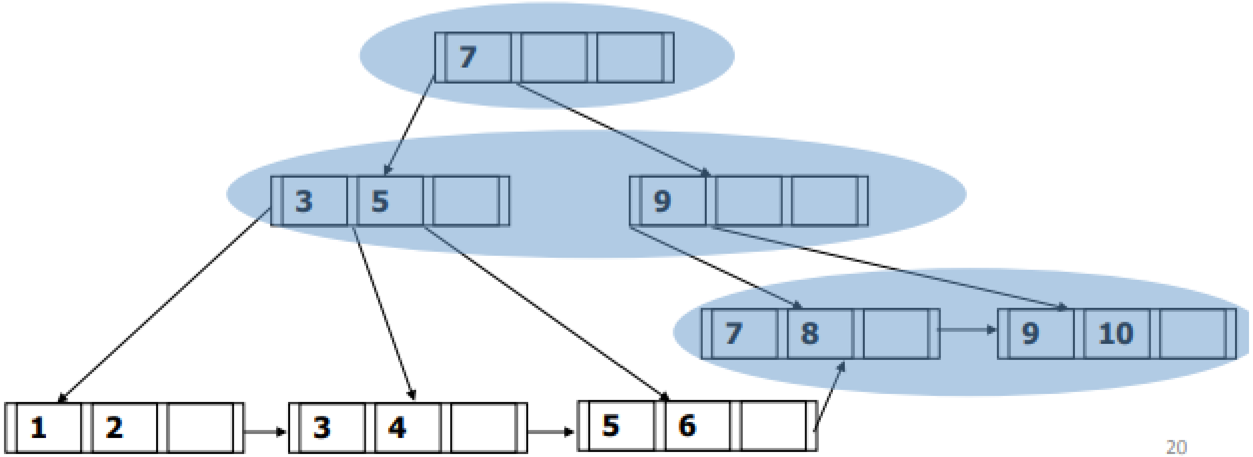
- Insert 8



Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10



- Insert 10


7
3
5
9
7
8
9
10
1
2
3
4
5
6