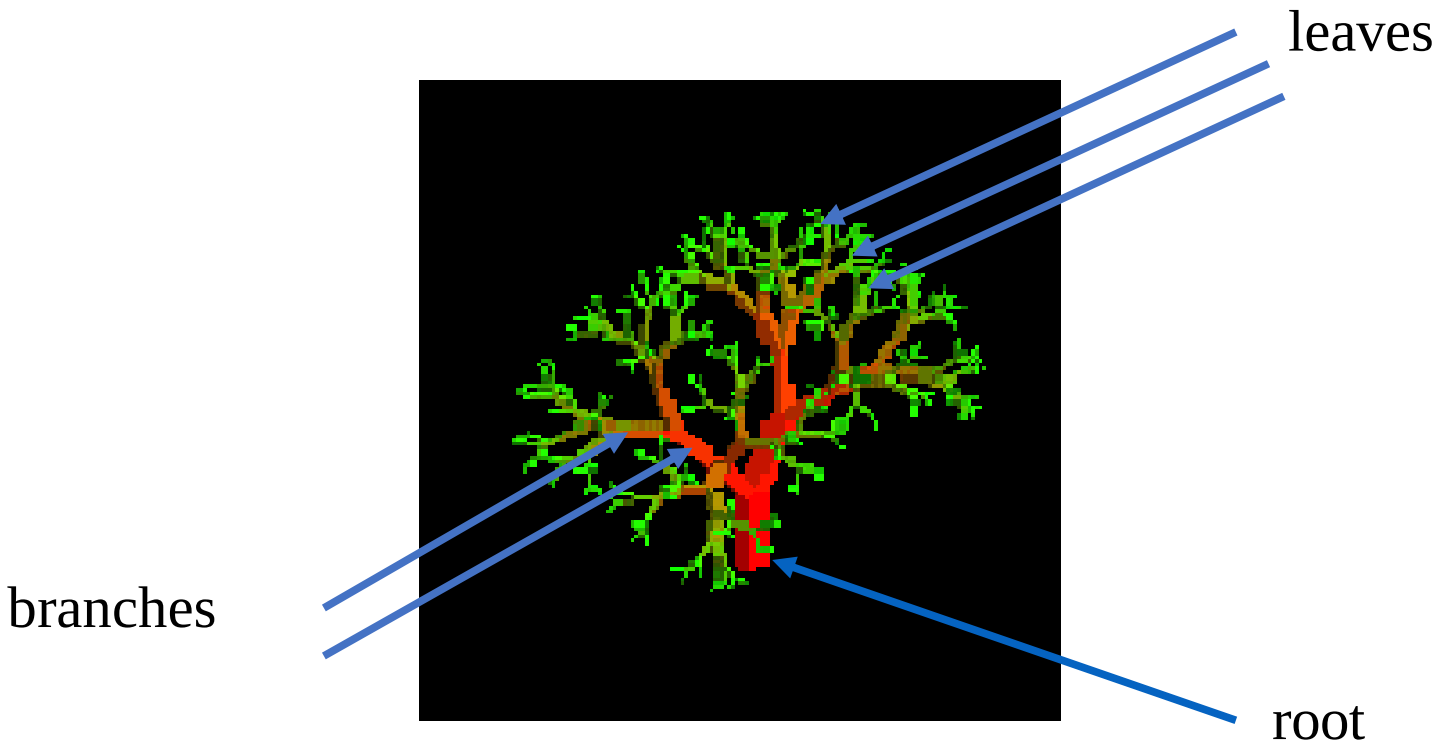
Trees

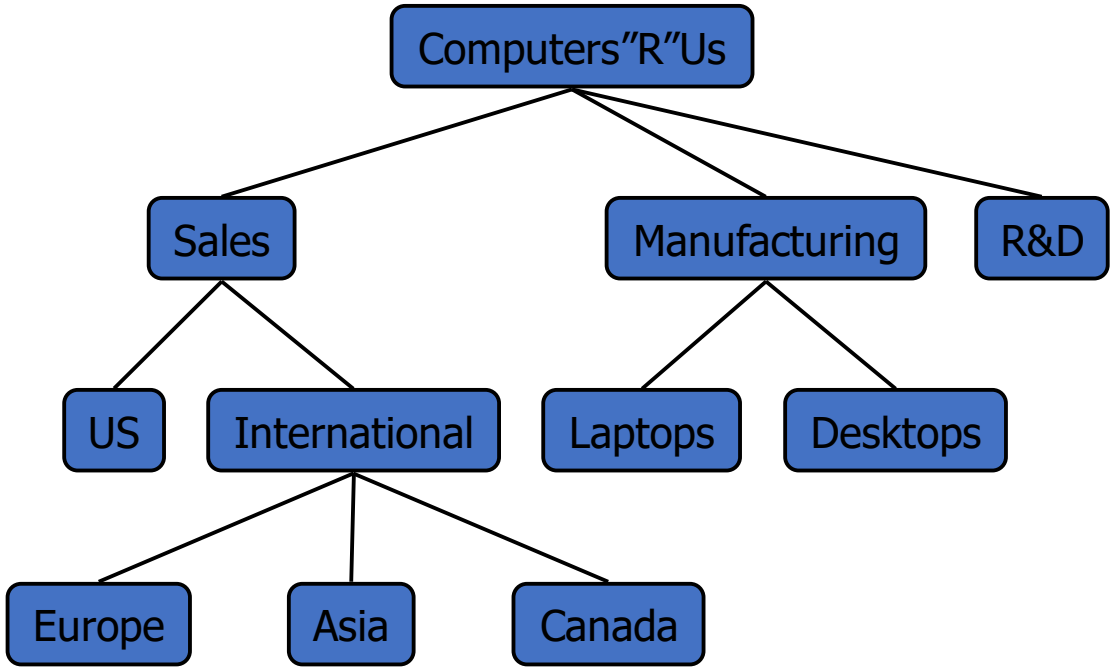
Trees

Nature View of a Tree



What is a Tree

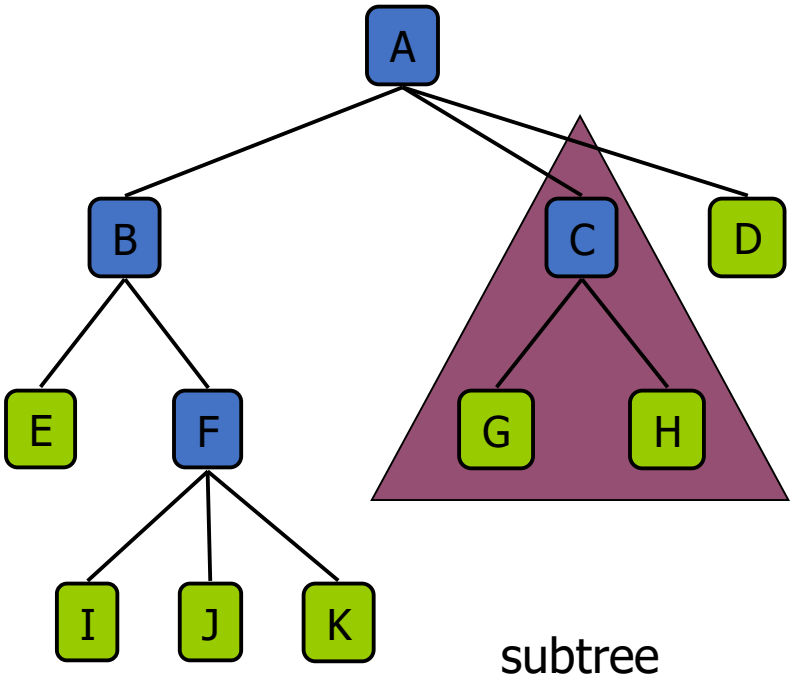
- A tree is a finite nonempty set of elements.
- It is an abstract model of a hierarchical structure.
- consists of nodes with a parent-child relation.
- Applications:
  - Organization charts
  - File systems
  - Programming environments



Tree Terminology

- **Root**: node without parent (A)
- **Siblings**: nodes share the same parent
- **Internal node**: node with at least one child (A, B, C, F)
- **External node** (leaf): node without children (E, I, J, K, G, H, D)
- **Ancestors** of a node: parent, grandparent, grand-grandparent, etc.
- **Descendant** of a node: child, grandchild, grand-grandchild, etc.
- **Depth of the tree-** no. of edges from root to leaf node in max path.
- **Height** of a tree: no. of edges from leaf node to root node in max path.

- **Subtree**: tree consisting of a node and its descendants



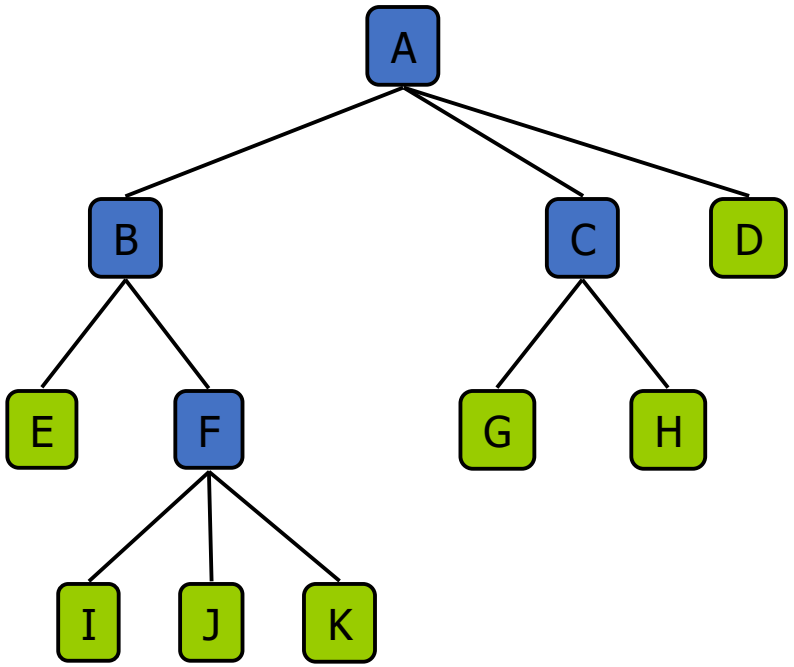
Tree Terminology

Depth of the node – no. of edges from root node to that node.

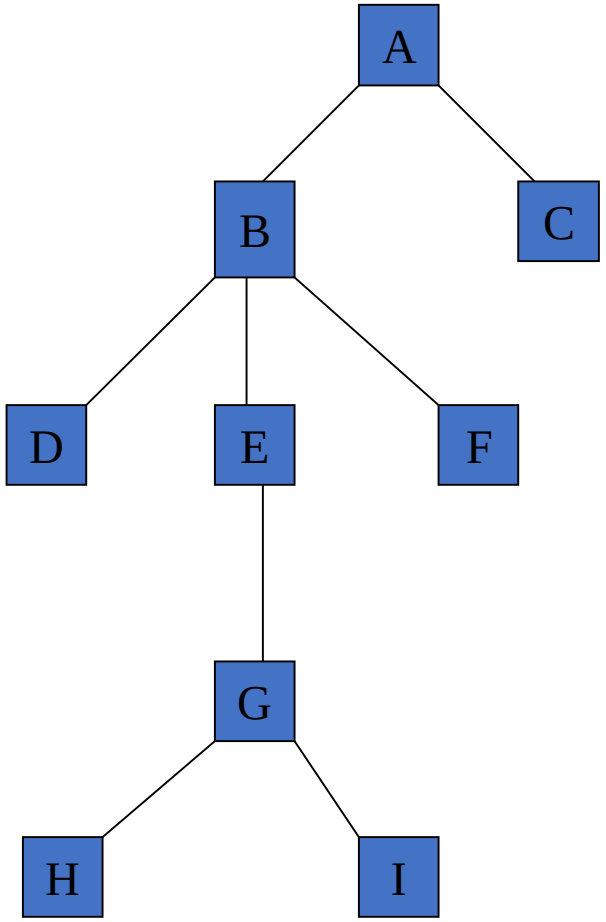
Height of the node – no. of edges from leaf node to that node.

Degree of a node: the number of its children

Degree of a tree: the maximum degree



Tree Properties



Property	Value
Number of nodes	
Height	
Root Node	
Leaves	
Interior nodes	
Ancestors of H	
Descendants of B	
Siblings of E	
Right subtree of A	
Degree of this tree	

Tree ADT

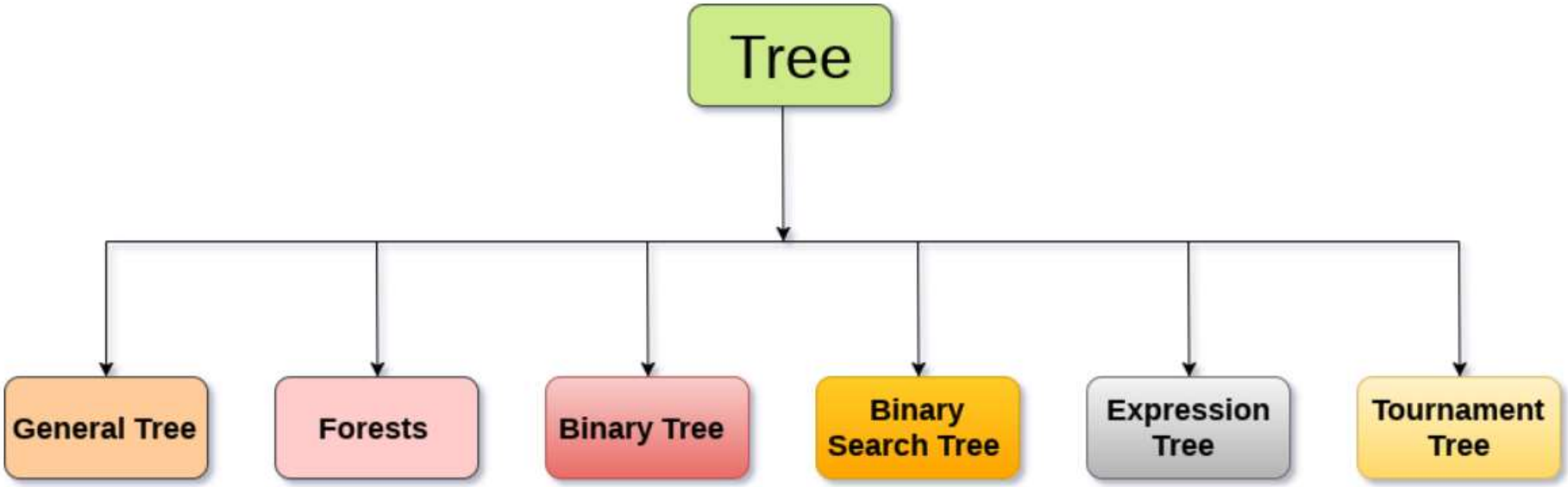
- We use positions to abstract nodes
- Generic methods:
  - integer **size**()
  - boolean **isEmpty**()
  - objectIterator **elements**()
  - positionIterator **positions**()
- Accessor methods:
  - position **root**()
  - position **parent**(p)
  - positionIterator **children**(p)

- Query methods:
  - boolean **isInternal**(p)
  - boolean **isExternal**(p)
  - boolean **isRoot**(p)
- Update methods:
  - **swapElements**(p, q)
  - object **replaceElement**(p, o)
- Additional update methods may be defined by data structures implementing the Tree ADT

Trees

Types of Tree

The tree data structure can be classified into six different categories.



Trees

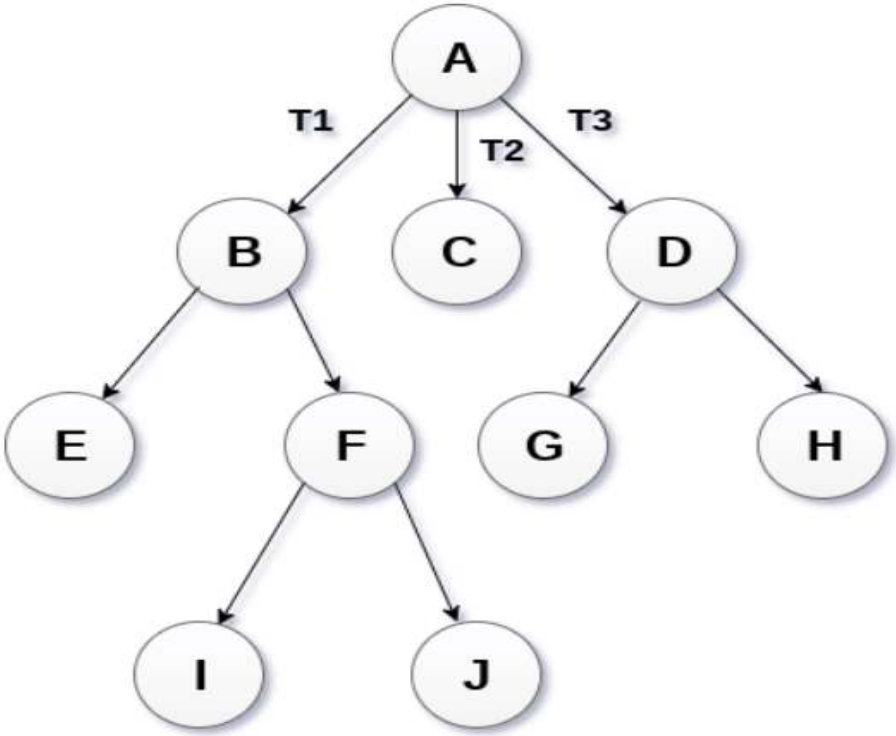
General Tree

General Tree stores the elements in a hierarchical order in which the top level element is always present at level 0 as the root element. All the nodes except the root node are present at number of levels. The nodes which are present on the same level are called siblings while the nodes which are present on the different levels exhibit the parent-child relationship among them. A node may contain any number of sub-trees.

Forests

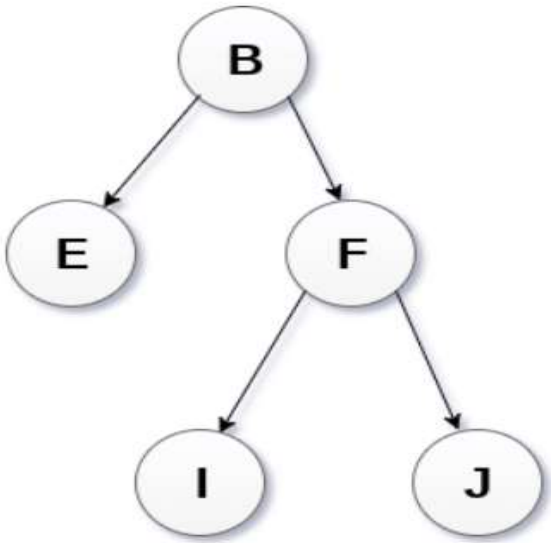
Forest can be defined as the set of disjoint trees which can be obtained by deleting the root node and the edges which connects root node to the first level node.

Trees

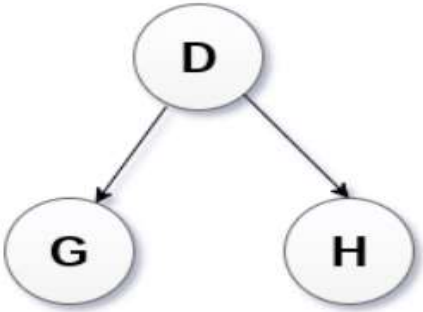


General Tree

Forest 2



Forest 1



Forest 3

Trees

Binary Tree

Binary tree is a data structure in which each node can have at most 2 children. The node present at the top most level is called the root node. A node with the 0 children is called leaf node. Binary Trees are used in the applications like expression evaluation and many more.

Binary Search Tree

Binary search tree is an ordered binary tree. All the elements in the left sub-tree are less than the root while elements present in the right sub-tree are greater than or equal to the root node element. Binary search trees are used in most of the applications of computer science domain like searching, sorting, etc.

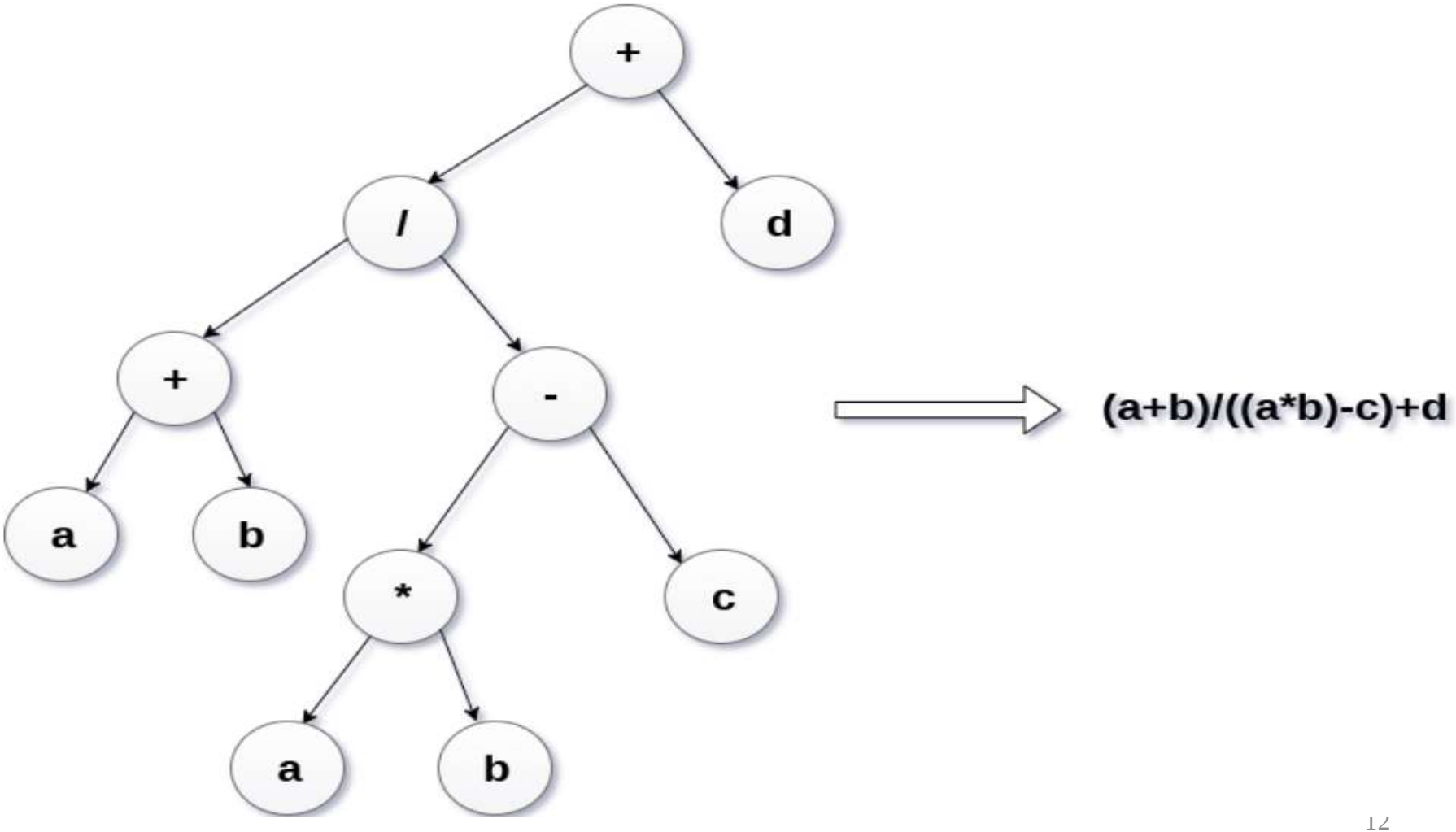
Expression Tree

Expression trees are used to evaluate the simple arithmetic expressions. Expression tree is basically a binary tree where internal nodes are represented by operators while the leaf nodes are represented by operands.

Trees

Q. Construct an expression tree by using the following algebraic expression.

(a + b) / (a*b - c) + d

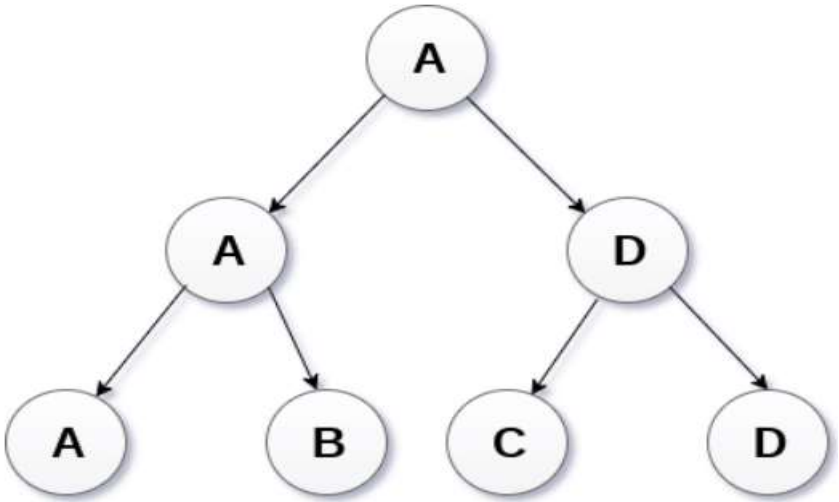


Trees

Tournament Tree

Tournament tree are used to record the winner of the match in each round being played between two players. Tournament tree can also be called as selection tree or winner tree. External nodes represent the players among which a match is being played while the internal nodes represent the winner of the match played. At the top most level, the winner of the tournament is present as the root node of the tree.

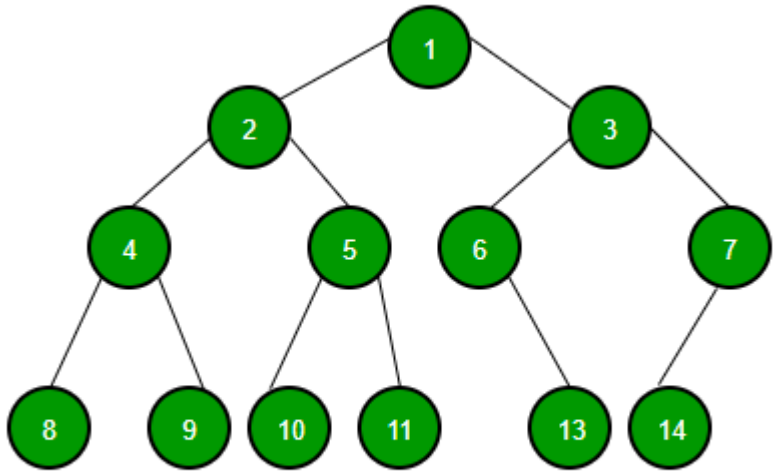
For example, tree of a chess tournament being played among 4 players is shown as follows. However, the winner in the left sub-tree will play against the winner of right sub-tree.



Binary Tree

A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

A Binary Tree node contains following parts.
1.Data
2.Pointer to left child
3.Pointer to right child



Binary Tree(Properties)

1) The maximum number of nodes at level 'l' of a binary tree is 2^l.

Here level is the number of nodes on the path from the root to the node (including root and node). Level of the root is 0.

This can be proved by induction.

For root, l = 0, number of nodes = $2^0 = 1$

Assume that the maximum number of nodes on level 'l' is 2^l

Since in Binary tree every node has at most 2 children, next level would have twice nodes, i.e. $2 * 2^l$

2. The Maximum number of nodes in a binary tree of height 'h' is $2^{h+1} – 1$.

3. In a Binary Tree with N nodes, minimum possible height or the minimum number of levels is?

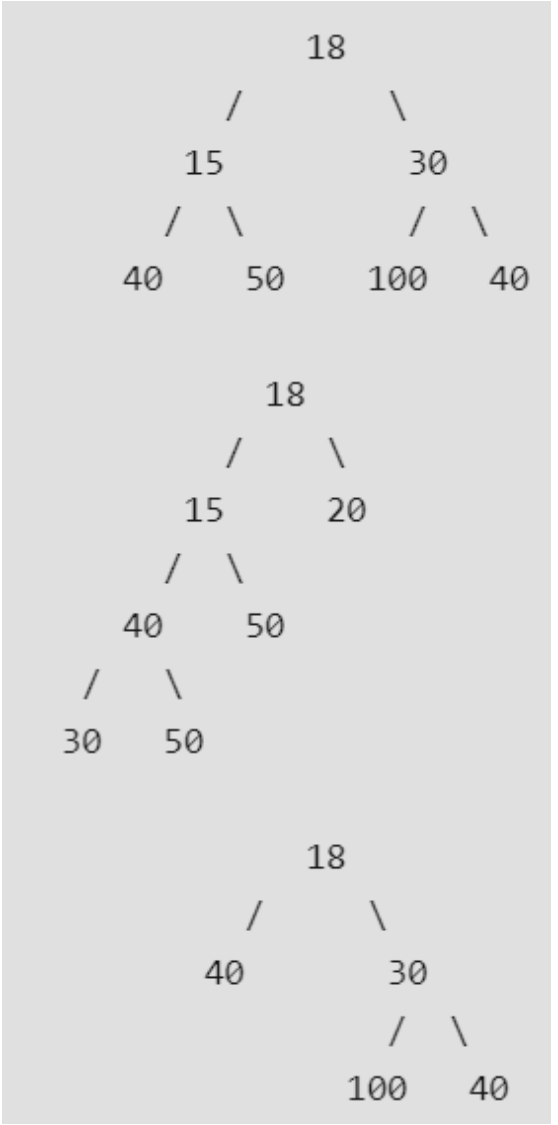
$$\log_2(N+1) -1$$

Types of Binary Tree

1. Complete Binary Tree
2. Full Binary Tree
3. Balanced Binary Tree

Trees

Full Binary Tree A Binary Tree is a full binary tree if every node has 0 or 2 children. The following are the examples of a full binary tree. We can also say a full binary tree is a binary tree in which all nodes except leaf nodes have two children.



In a Full Binary Tree, number of leaf nodes is the number of internal nodes plus 1

$$L = I + 1$$

Where L = Number of leaf nodes, I = Number of internal nodes

FULL Binary Tree theorem

Theorem: Let T be a nonempty, full binary tree Then:

(a) If T has I internal nodes, the number of leaves is L = I + 1.

(b) If T has I internal nodes, the total number of nodes is N = 2I + 1.

(c) If T has a total of N nodes, the number of internal nodes is I = (N – 1)/2.

(d) If T has a total of N nodes, the number of leaves is L = (N + 1)/2.

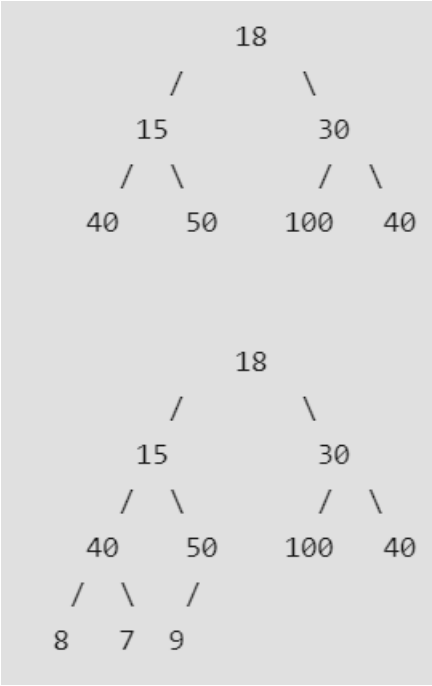
(e) If T has L leaves, the total number of nodes is N = 2L – 1.

(f) If T has L leaves, the number of internal nodes is I = L – 1.

Trees

Complete Binary Tree: A Binary Tree is a complete Binary Tree if all the levels are completely filled except possibly the last level and the last level has all keys as left as possible

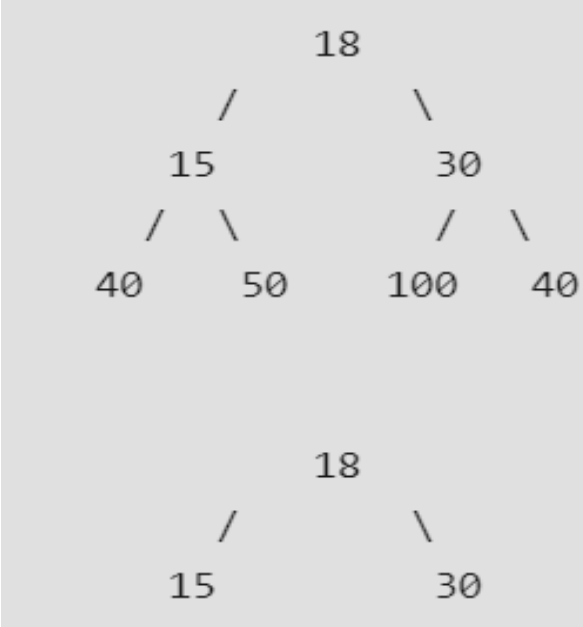
The following are examples of Complete Binary Trees



Trees

Perfect Binary Tree: A Binary tree is a Perfect Binary Tree in which all the internal nodes have two children and all leaf nodes are at the same level.

The following are the examples of Perfect Binary Trees.



A Perfect Binary Tree of height h (where the height of the binary tree is the longest path from the root node to any leaf node in the tree) has $2^{h+1} - 1$ node.

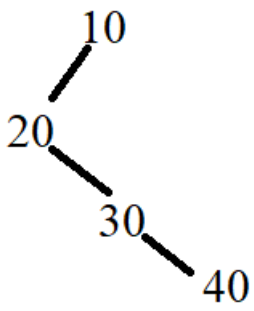
Trees

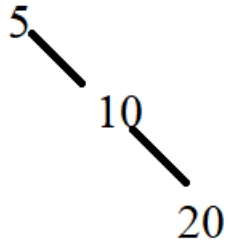
Balanced Binary Tree

A binary tree is balanced if the height of the tree is O(Log n) where n is the number of nodes. For Example, the AVL tree maintains O(Log n) height by making sure that the difference between the heights of the left and right subtrees is almost 1. Red-Black trees maintain O(Log n) height by making sure that the number of Black nodes on every root to leaf paths is the same and there are no adjacent red nodes. Balanced Binary Search trees are performance-wise good as they provide O(log n) time for search, insert and delete

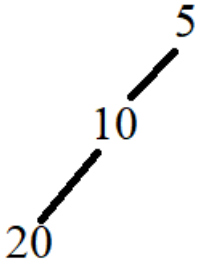
Trees

A degenerate (or pathological) tree: A Tree where every internal node has one child. Such trees are performance-wise same as linked list.





right skewed tree



left skewed tree

Trees

A perfect binary tree of height h is a binary tree where:

1. all leaf nodes have the same depth, h, and
2. all other nodes are full nodes.

A perfect binary tree of height 5 is shown in Figure 1.

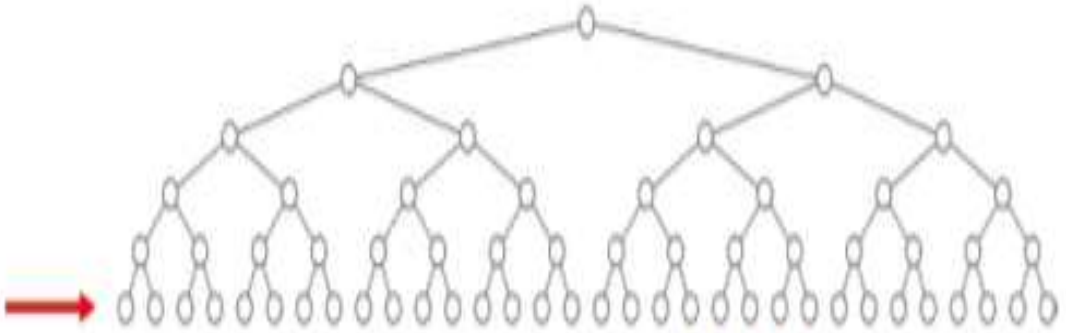


Figure 1. A perfect binary tree of height $h = 5$.

Trees

A recursive definition of a perfect binary tree is:

1. A single node with no children is a perfect binary tree of height $h = 0$,
2. A perfect binary tree with height $h > 0$ is a node where both sub-trees are non-overlapping perfect binary trees of height $h - 1$.

Perfect binary trees of heights 0, 1, 2, 3 and 4 are shown in Figure 2.

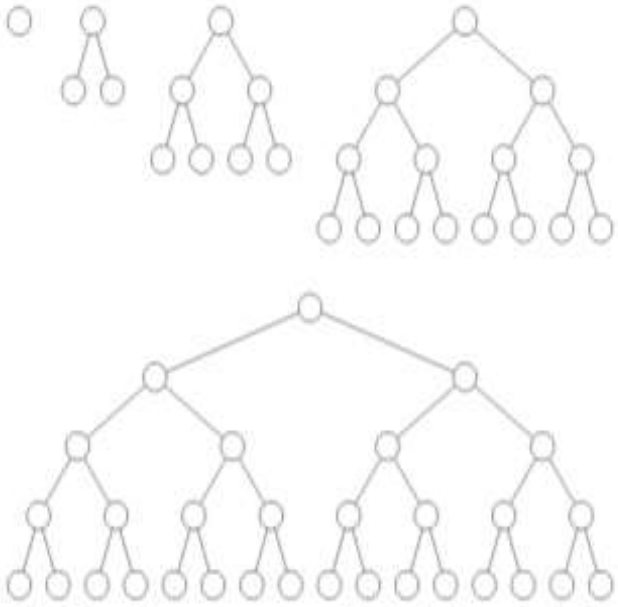


Figure 2. Perfect binary trees of height $h = 0, 1, 2, 3$, and 4.

Binary Tree Representations

A binary tree data structure is represented using two methods. Those methods are as follows...

Array Representation

Linked List Representation

Trees

1. Array Representation of Binary Tree

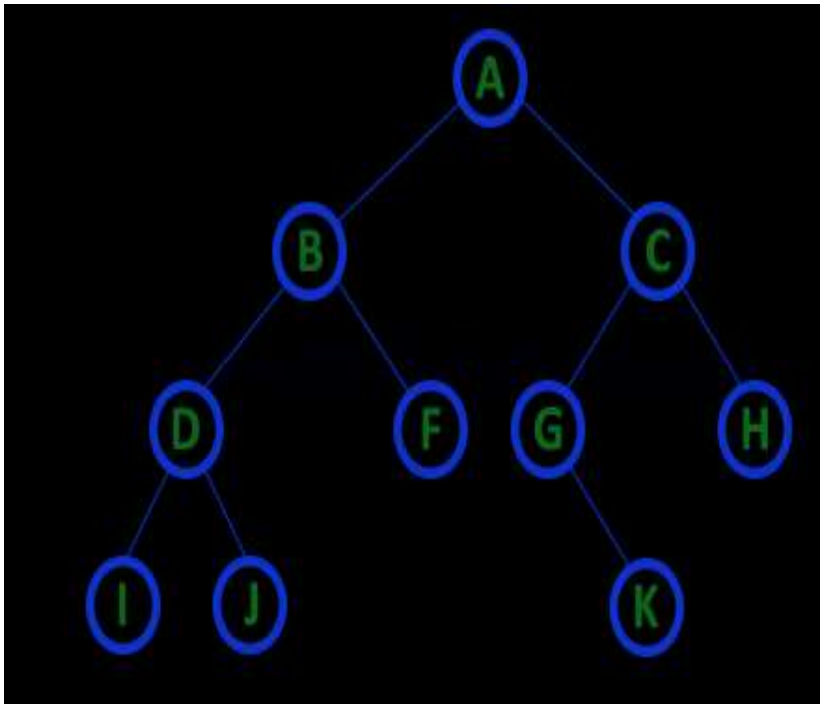
In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.

a) Root node is stored in TREE[1]

b) If a node N occupies TREE[k], then:

   i. Its left child is stored in TREE[2k]

   ii. Its right child is stored in TREE[2k+1]



Example:

1) Root node A is stored at index 1, its left child at index 2 and right child at index 3.

2) Node F is at index 5 and has no child, so index 10 and 11 are empty.

3) Node G is at index 6, its right child at index 2K+1= 13

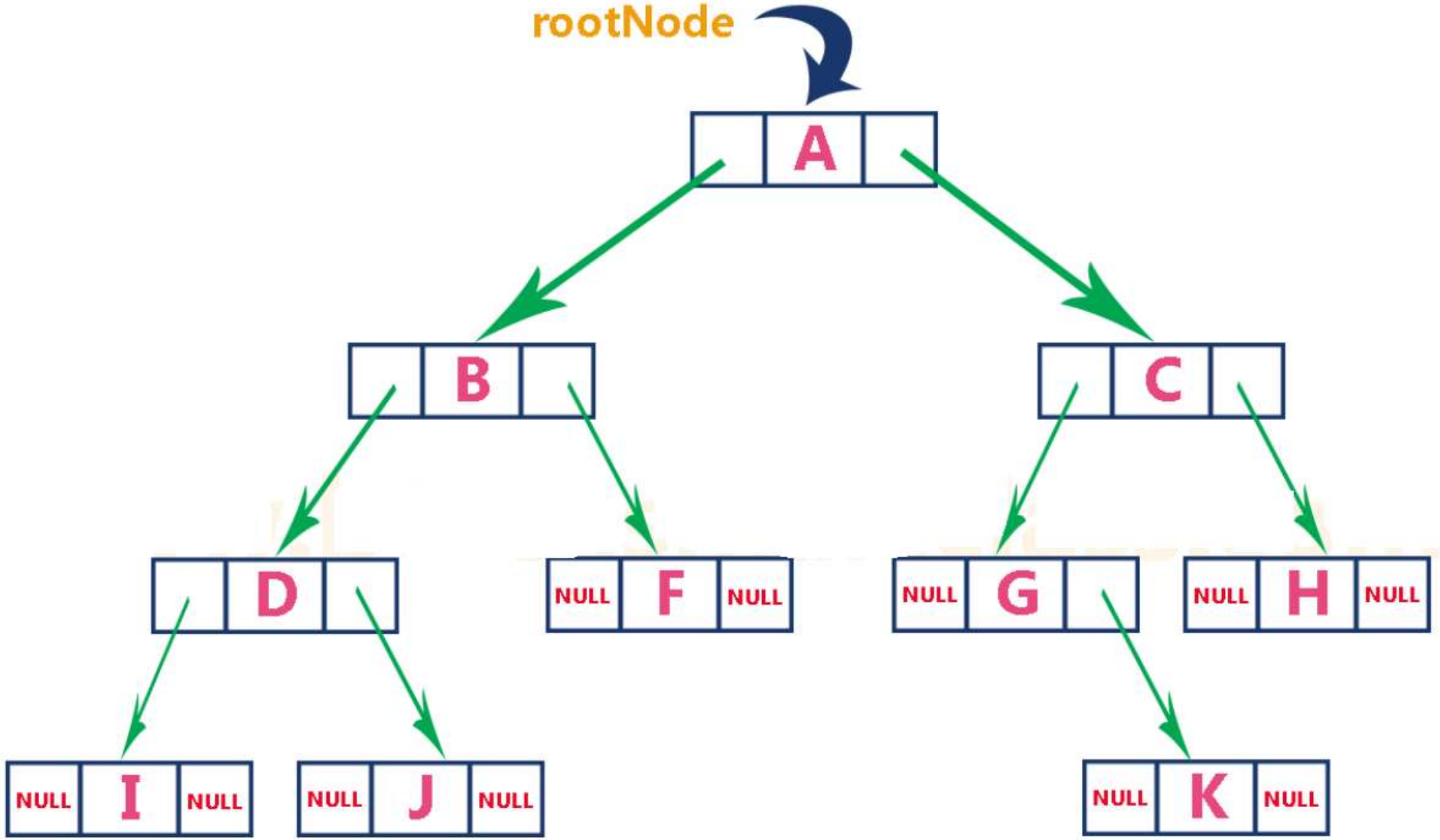
A	B	C	D	F	G	H	I	J	-	-	-	K	-	-	-	-	-	-	-	-

Trees

2. Linked List Representation of Binary Tree

We use a double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.

Trees

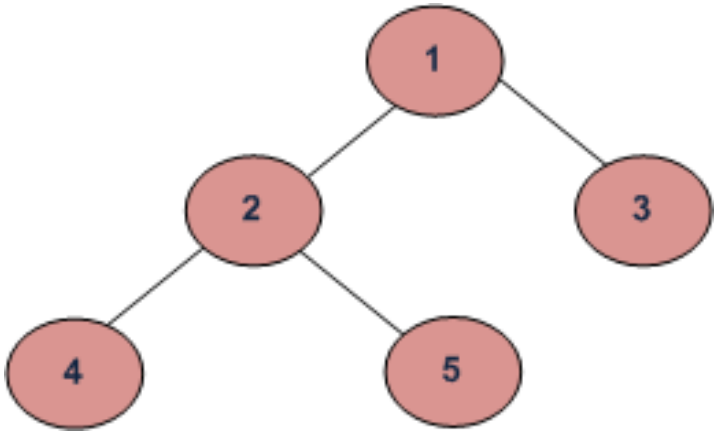

rootNode
A
B
C
D
NULL
F
NULL
NULL
G
NULL
NULL
H
NULL
NULL
I
NULL
NULL
J
NULL
NULL
K
NULL

Binary Tree Traversal

- Preorder (Node,Left,Right)
- Postorder (Left, Right, Node)
- Inorder(Left, Node, Right)

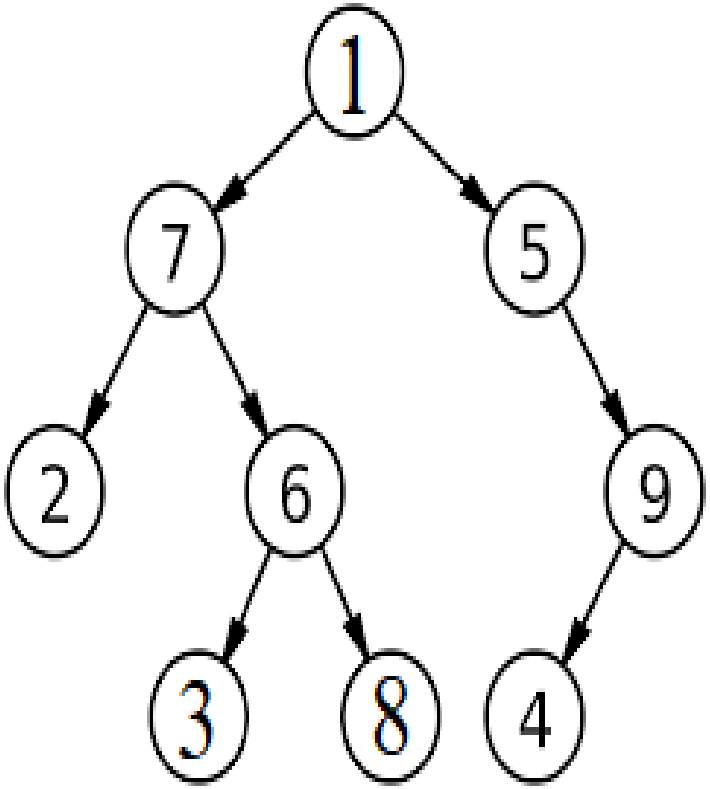
SN	Traversal	Description
1	Pre-order Traversal	Traverse the root first then traverse into the left sub-tree and right sub-tree respectively. This procedure will be applied to each sub-tree of the tree recursively.
2	In-order Traversal	Traverse the left sub-tree first, and then traverse the root and the right sub-tree respectively. This procedure will be applied to each sub-tree of the tree recursively.
3	Post-order Traversal	Traverse the left sub-tree and then traverse the right sub-tree and root respectively. This procedure will be applied to each sub-tree of the tree recursively.

Trees



(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

Trees



PreOrder : (NLR)

Ans – 1 7 2 6 3 8 5 9 4

InOrder: (LNR)

Ans – 2 7 3 6 8 1 5 9 4

PostOrder: (LRN)

Ans - 2 3 8 6 7 4 9 5 1

Trees

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)

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)

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.

Trees

Construct Tree from given Inorder and Postorder traversals

1. Pickup last key of post order sequence and create node of each.
2. Create left subtree of root node recursively by selecting keys which are preceding the root node in Inorder.
3. Create right subtree of root node recursively by selecting keys which are following the root node in Inorder

Trees

Construct Tree from given Inorder and Postorder traversals

Inorder: 1 2 3 4 5 7 8
Postorder: 1 3 4 2 7 8 5

Trees

Construct Tree from given Inorder and Preorder traversals

1. Pickup first key of pre order sequence and create node of each.

2. Create left subtree of root node recursively by selecting keys which are preceding the root node in Inorder.

3. Create right subtree of root node recursively by selecting keys which are following the root node in Inorder

Trees

Construct Tree from given Inorder and Preorder traversals

Inorder: 1 2 3 4 5 7 8
Preorder: 5 2 1 4 3 8 7

Trees

1. Write the C function to count number of nodes in a binary tree.

```
int countnodes(struct node *r)
{
        if(r==NULL)
                return 0;
        return countnodes(r→left)+ countnodes(r→right)+1;
}
```

Trees

2. Write the C function to count number of leaf nodes in a binary tree.

```
int countleafnode(struct node *r)
{
        if(r==NULL)
                return 0;
        if(r→left==NULL&&r→right==NULL)
                return 1;

        return countleafnode(r→left)+ countleafnode(r→right);
}
```

Trees

3. Write the C function to count number of non-leaf nodes in a binary tree.

```
int countleafnode(struct node *r)
{
        if(r==NULL)
                return 0;
        if(r→left==NULL&&r→right==NULL)
                return 0;

        return countleafnode(r→left)+ countleafnode(r→right)+1;
}
```

Trees

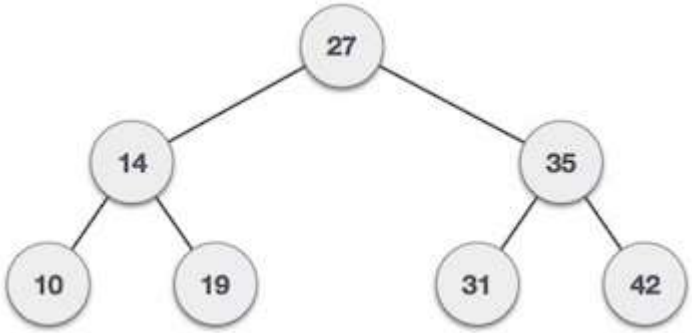
4. Write the C function to count height of the binary tree.

```
int height(struct node *r)
{
        if(r==NULL)
                return 0;
        lh = height(r→left);
        rh = height(r→right);
        if(lh>=rh)
            return lh+1;
        else
             return rh+1;
}
```

Binary Search Tree Representation

Binary Search tree exhibits a special behavior.

A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value.



Binary Search Tree Representation

Construct all possible Binary Search tree by inserting following keys in any order.

10, 20,30

BST

Tree Node
The code to write a tree node would be similar to what is given below. It has a data part and references to its left and right child nodes.

```
struct node {
   int data;
   struct node *leftChild;
   struct node *rightChild;
};
```

BST Basic Operations

The basic operations that can be performed on a binary search tree data structure, are the following −

•**Insert** − Inserts an element in a tree/create a tree.

•**Search** − Searches an element in a tree.

•**Preorder Traversal** − Traverses a tree in a pre-order manner.

•**Inorder Traversal** − Traverses a tree in an in-order manner.

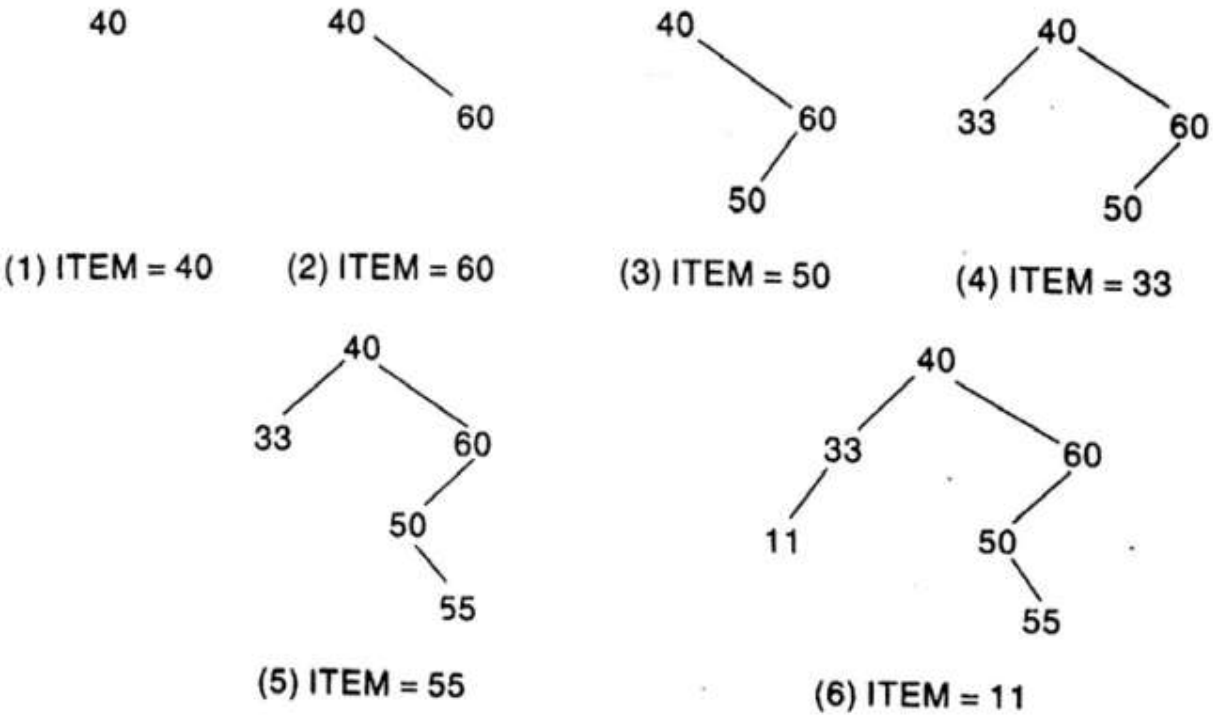
•**Postorder Traversal** − Traverses a tree in a post-order manner.

We shall learn creating (inserting into) a tree structure and searching a data item in a tree in this chapter. We shall learn about tree traversing methods in the coming chapter.

Insertion in BST

- Insert the following elements in empty BST:

  40, 60, 50, 33, 55,11



Insertion in BST

```
struct node* insert(struct node* node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL)
        return newNode(key);


    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    return node;
}
```

BST Insertion Operation

1. To insert element in degenerate tree, it requires order of n times.

2. To insert an element in Complete Binary search tree in the worst case, the no. comparison required are bounded by $O(\log_2 n)$.

Trees

BST Search Operation

```
struct node* search(struct node* root, int key)
{
    // Base Cases: root is null or key is present at root
    if (root == NULL || root->key == key)
       return root;

    // Key is greater than root's key
    if (root->key < key)
       return search(root->right, key);

    // Key is smaller than root's key
    return search(root->left, key);
}
```

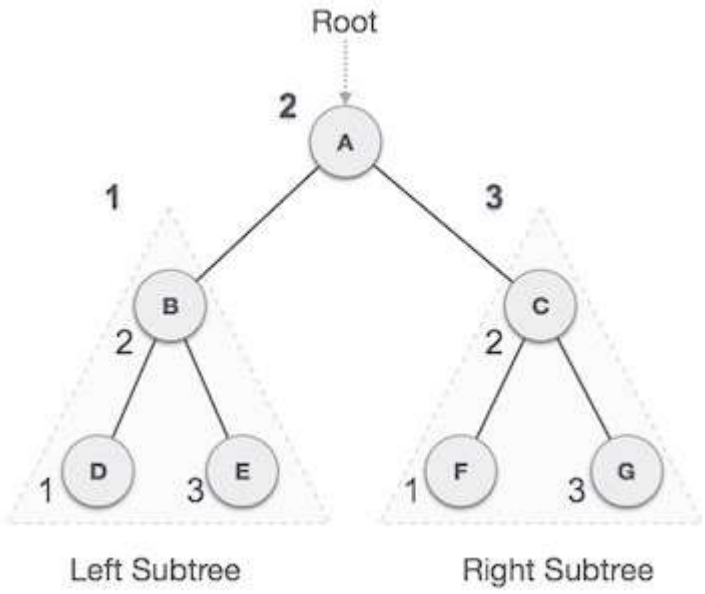
Trees

There are three ways which we use to traverse a tree

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

In-order Traversal

Algorithm
Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Visit root node.
Step 3 − Recursively traverse right subtree.

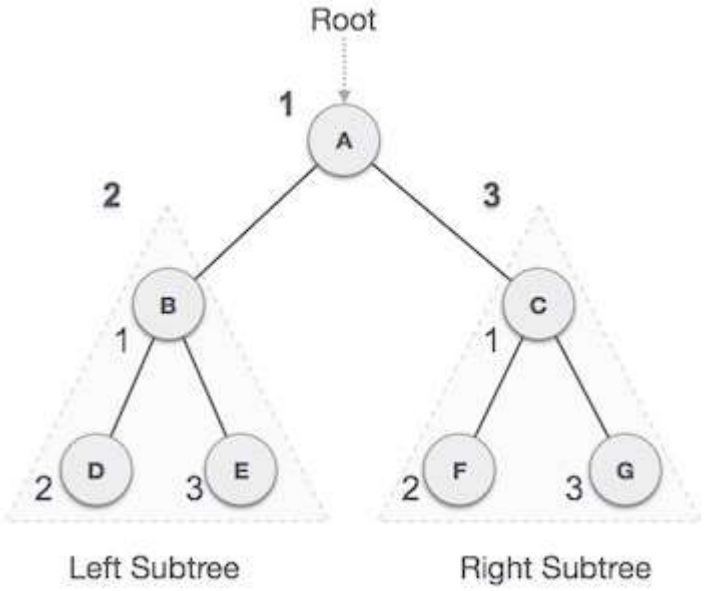


We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be −

$$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$$

Pre-order Traversal

Algorithm
Until all nodes are traversed −
Step 1 − Visit root node.
Step 2 − Recursively traverse left subtree.
Step 3 − Recursively traverse right subtree.

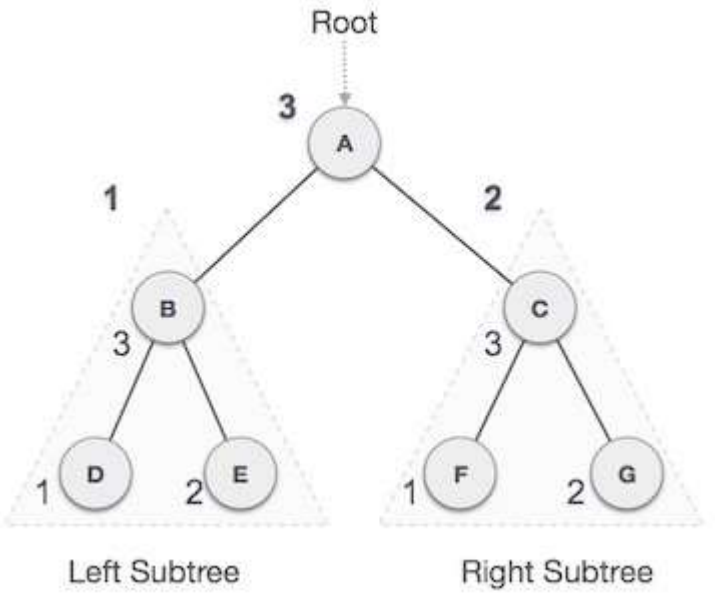


We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

$$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$$

Post-order Traversal

Algorithm
Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Recursively traverse right subtree.
Step 3 − Visit root node.



We start from **A**, and following Post-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

$$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$$

Binary Search Tree Deletion

Case 1 : Deleting the leaf node from BST

Task 1 : Finding key takes either log2(n) or O(n) times.

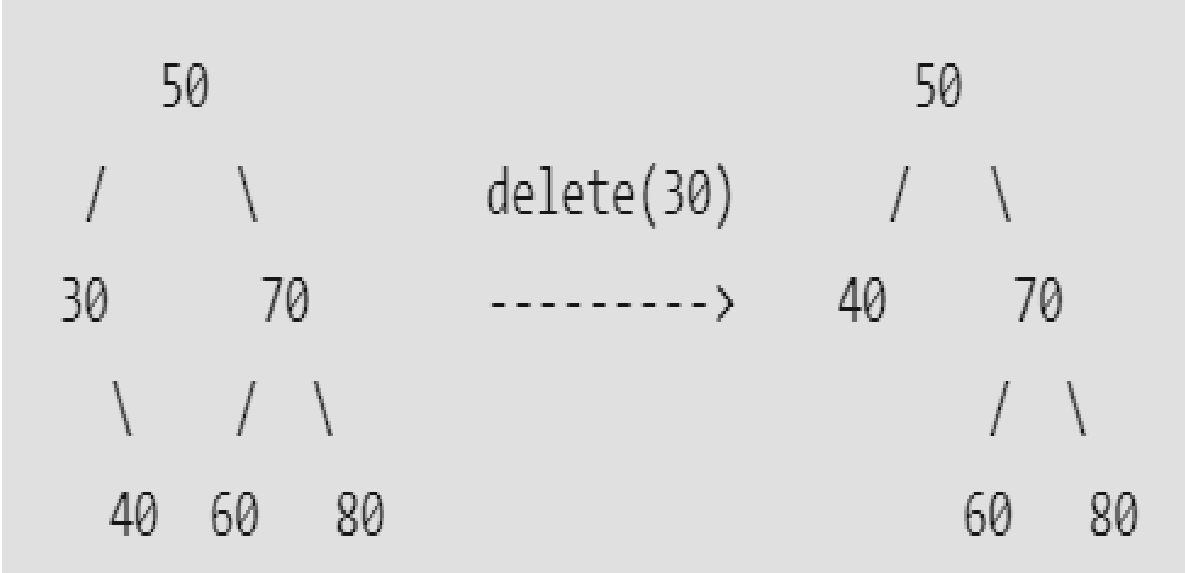
Task 2 : Identify leaf node parent pointer which is pointing to it and make it NULL.

Binary Search Tree Deletion

Case 2 : Deleting the node which has only one child

Task 1 : Finding key takes either log2(n) or O(n) times.

Task 2 : Copy the child to the node and delete the child

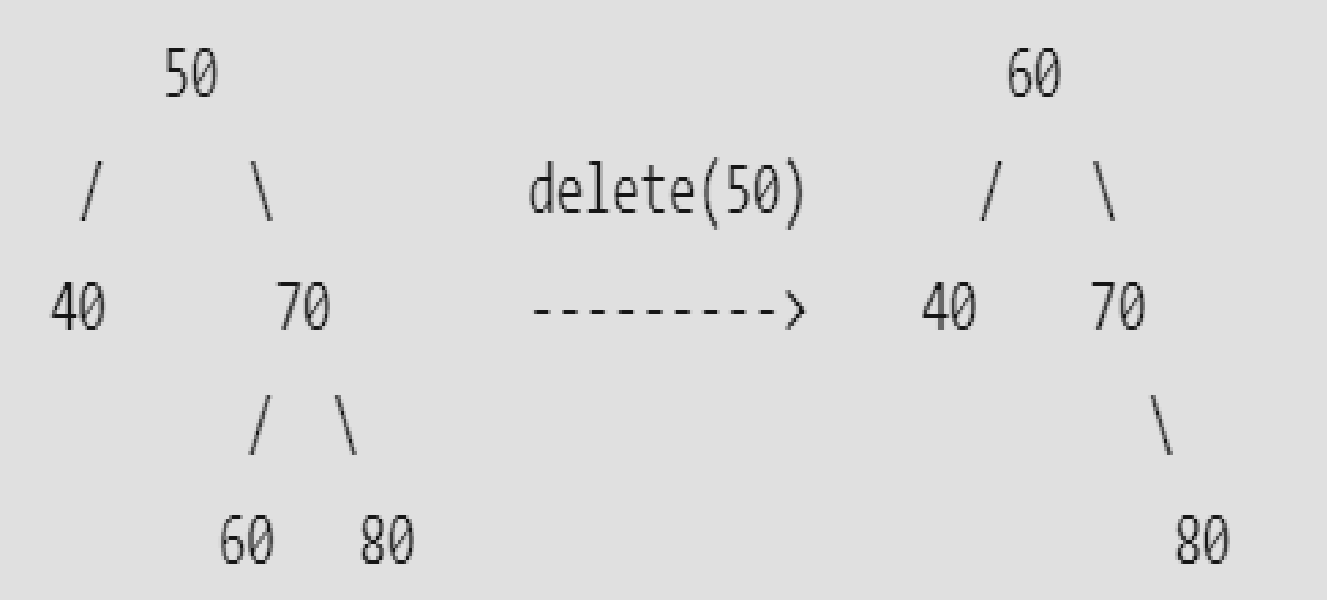


Binary Search Tree Deletion

Case 3 : Deleting the node which has left subtree or right subtree.

Task 1 : Finding key takes either log2(n) or O(n) times.

Task 2 : Replace the deleted node with the max of LST or min of RST.



AVL Tree

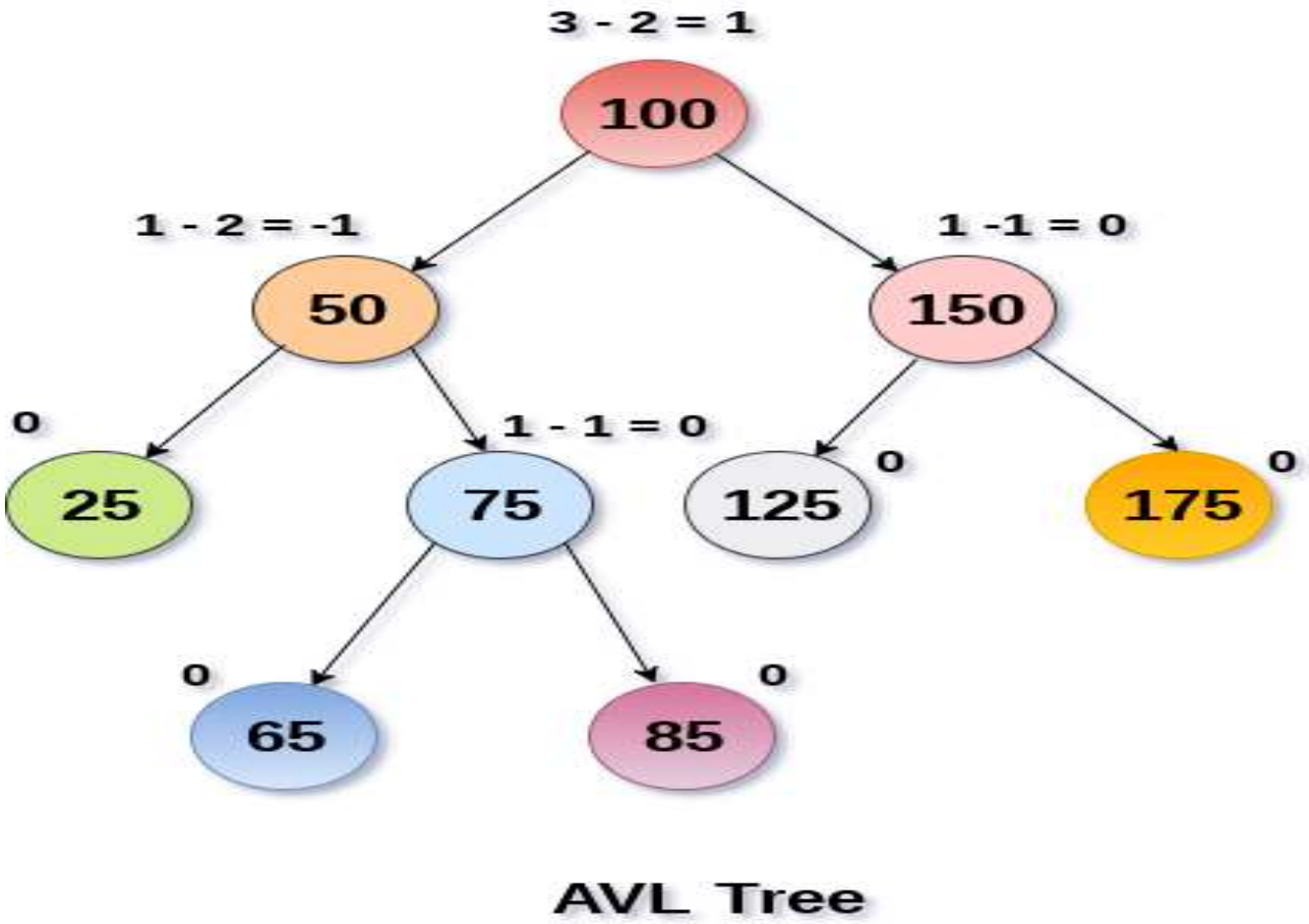
AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.

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.

$$\text{BalanceFactor} = \text{height(left-subtree)} - \text{height(right-subtree)}$$

Trees



AVL Tree

Balance Factor = height(left-subtree) − height(right-subtree)

Trees

Complexity

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

AVL Rotations

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

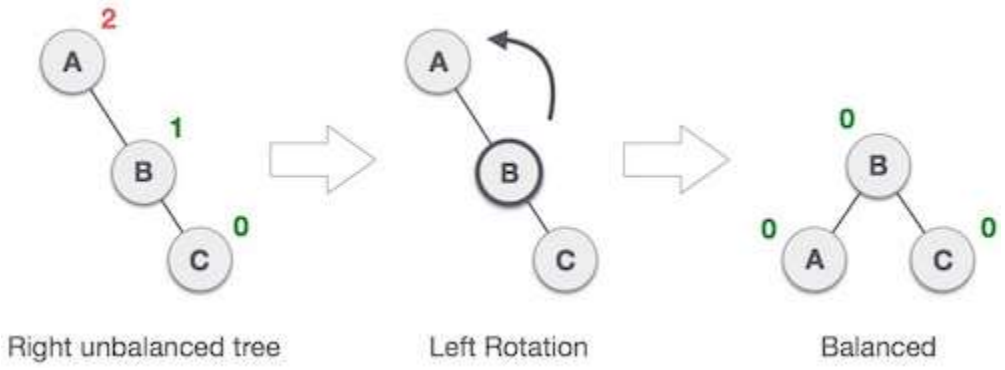
1.L L rotation: Inserted node is in the left subtree of left subtree of A
2.R R rotation : Inserted node is in the right subtree of right subtree of A
3.L R rotation : Inserted node is in the right subtree of left subtree of A
4.R L rotation : Inserted node is in the left subtree of right subtree of A
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.

Left Rotation/RR Rotation

If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation −

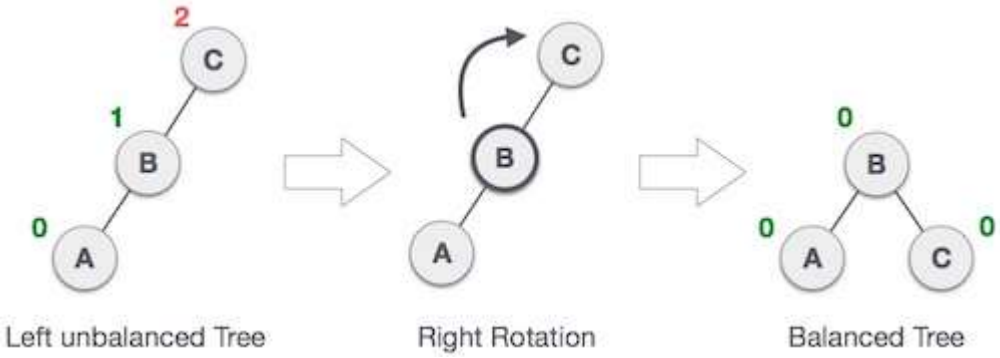
In our example, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A** the left-subtree of B.



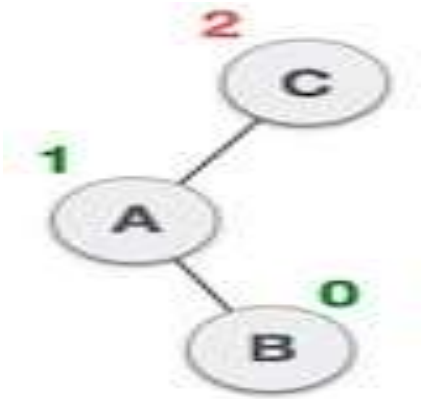
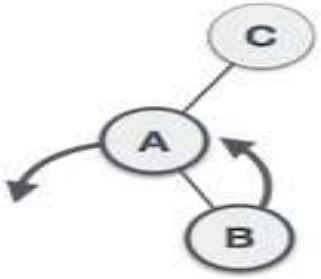
Right Rotation/LL Rotation

AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.

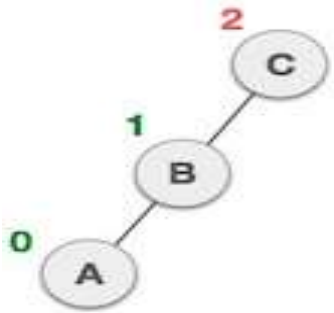
As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.



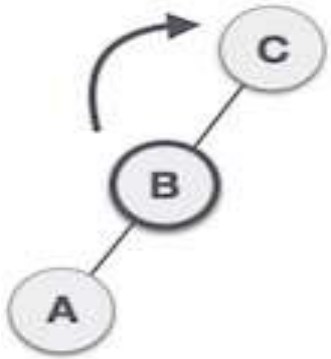
Left-Right Rotation

State	Action
	A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.
	We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**.

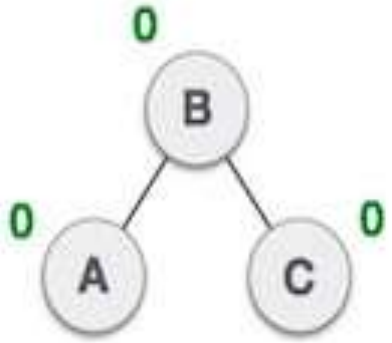
Trees



Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree.



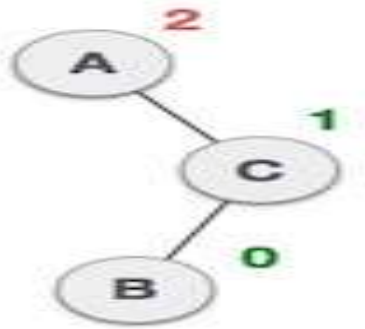
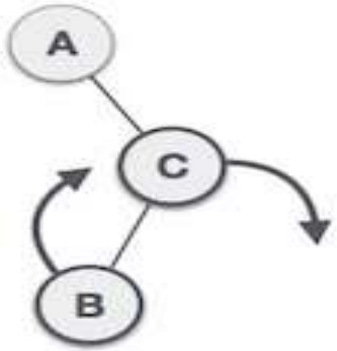
We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.



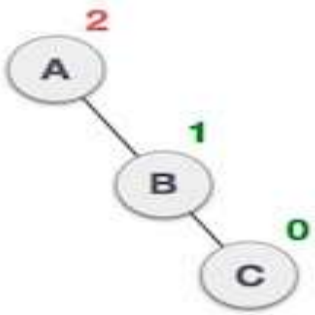
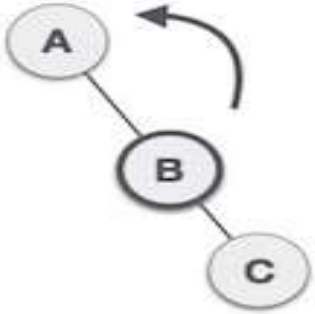
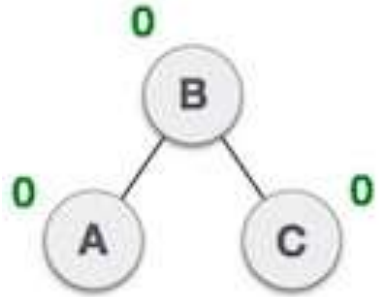
The tree is now balanced.

Right-Left Rotation

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

State	Action
	A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2.
	First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**.

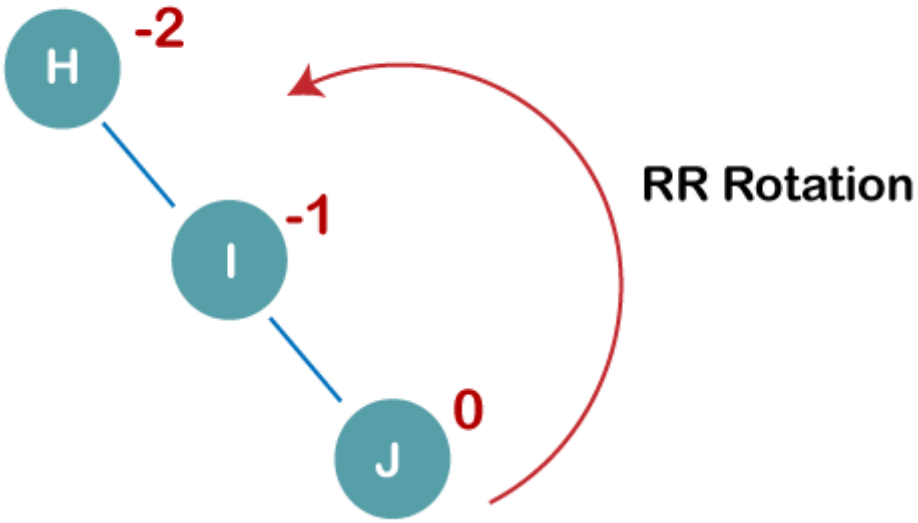
Trees

	Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation.
	A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**.
	The tree is now balanced.

Trees

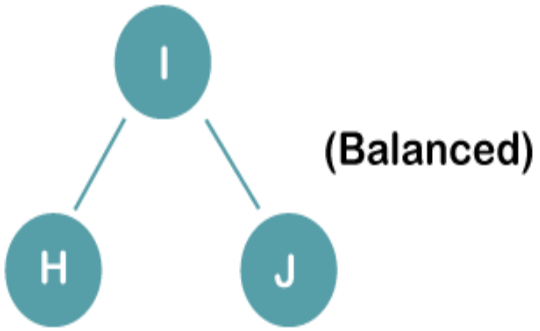
Q: Construct an AVL tree having the following elements
H, I, J, B, A, E, C, F, D, G, K, L

1. Insert H, I, J

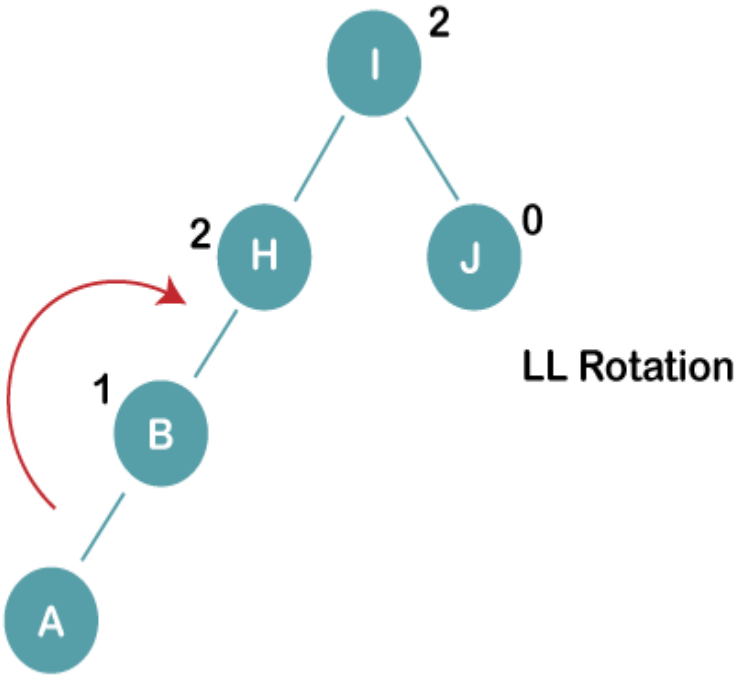


On inserting the above elements, especially in the case of H, the BST becomes unbalanced as the Balance Factor of H is -2. Since the BST is right-skewed, we will perform RR Rotation on node H.

The resultant balance tree is:



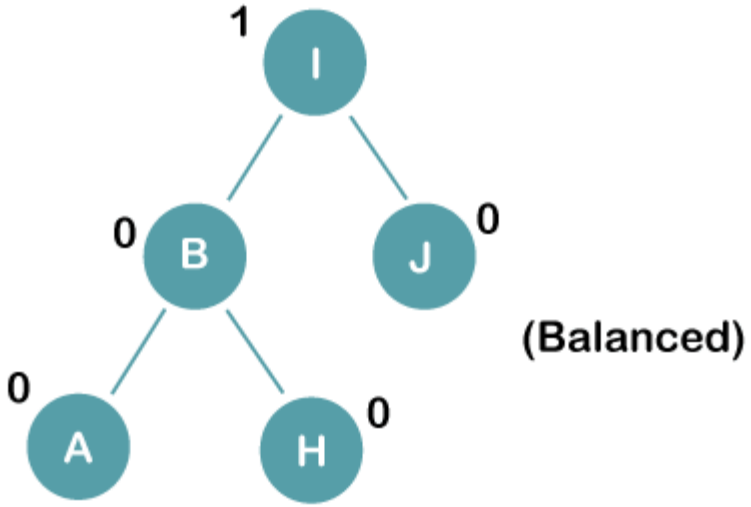
2. Insert B, A



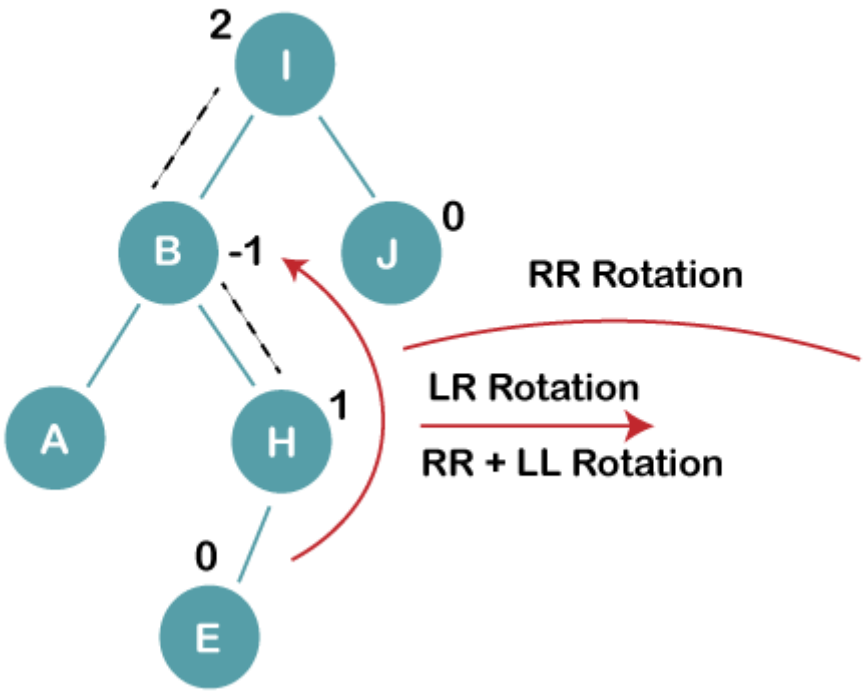
Trees

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

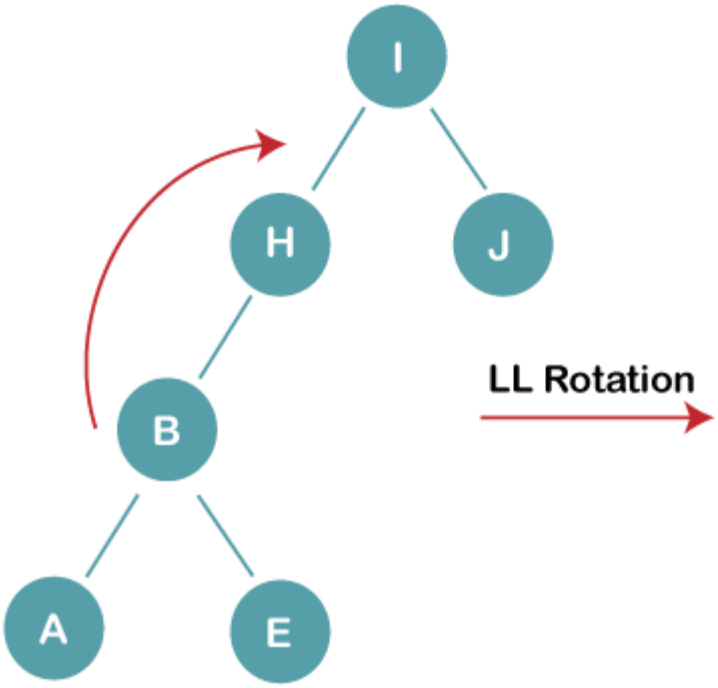


On inserting E,

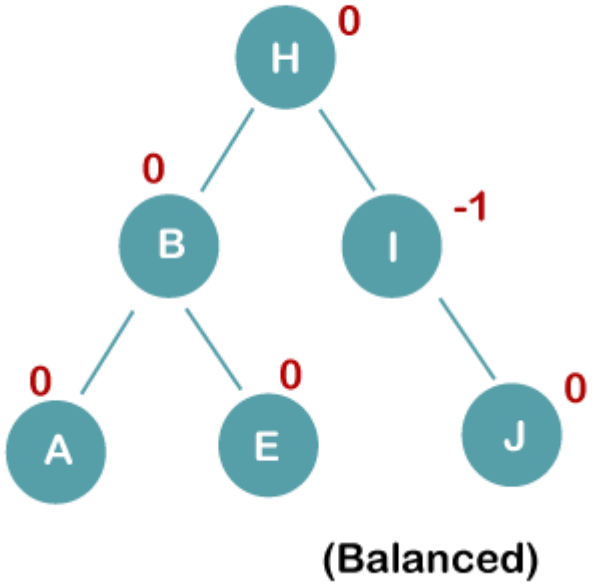
BST becomes unbalanced as the Balance Factor of I is 2, since if we travel from E to I we find that it is inserted in the left subtree of right subtree of I,

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:



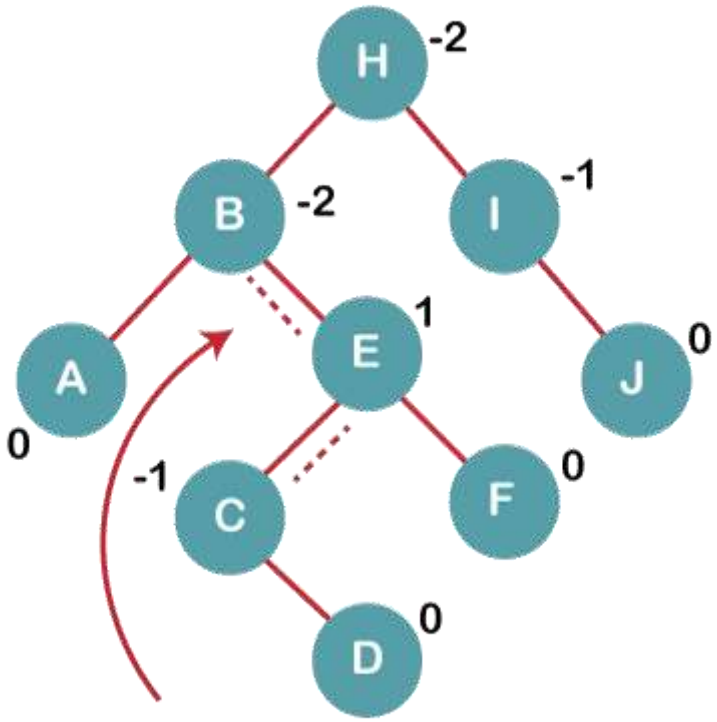
3b) We first perform LL rotation on the node I
The resultant balanced tree after LL rotation is:



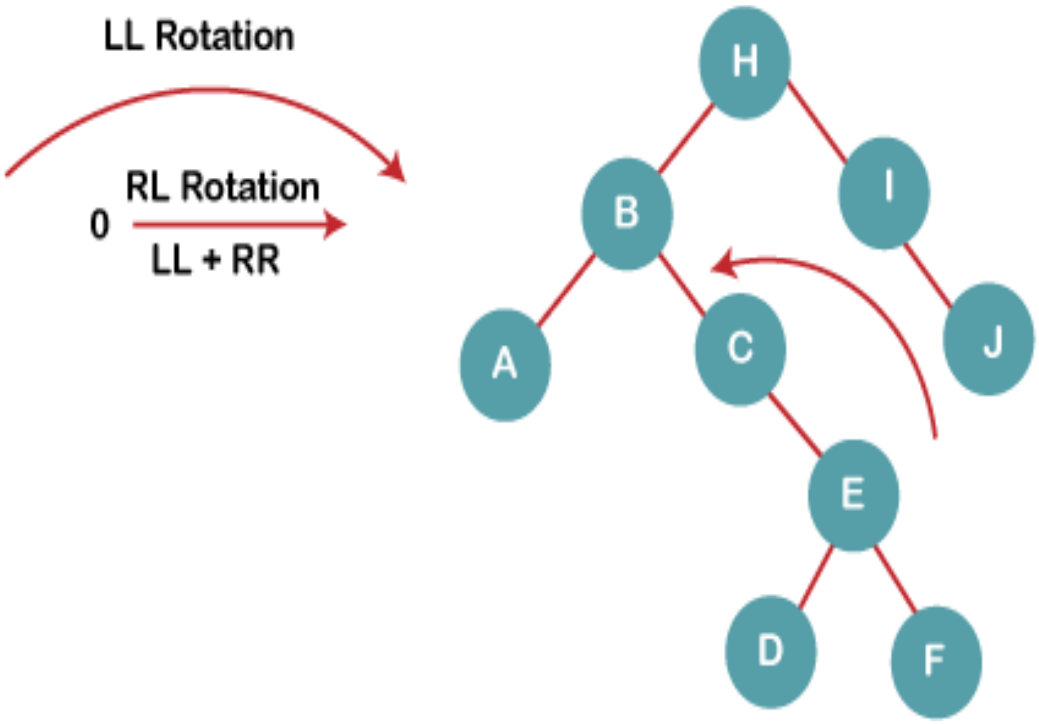
4. Insert C, F, D

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,

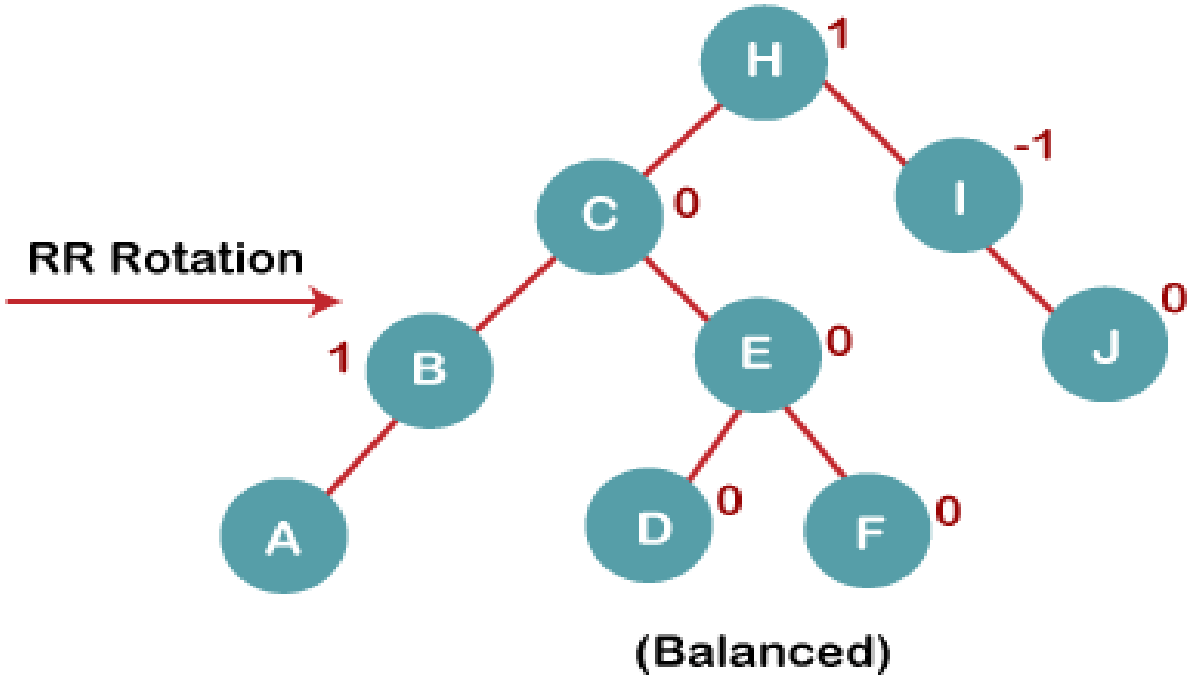
we will perform RL Rotation on node I. RL = LL + RR rotation.



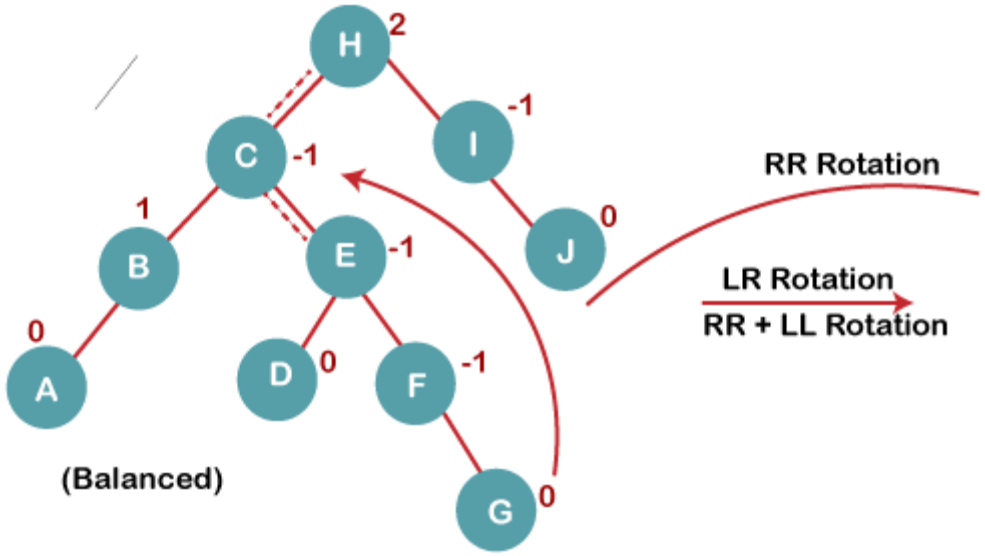
4a) We first perform LL rotation on node E
The resultant tree after LL rotation is:



4b) We then perform RR rotation on node B
The resultant balanced tree after RR rotation is:

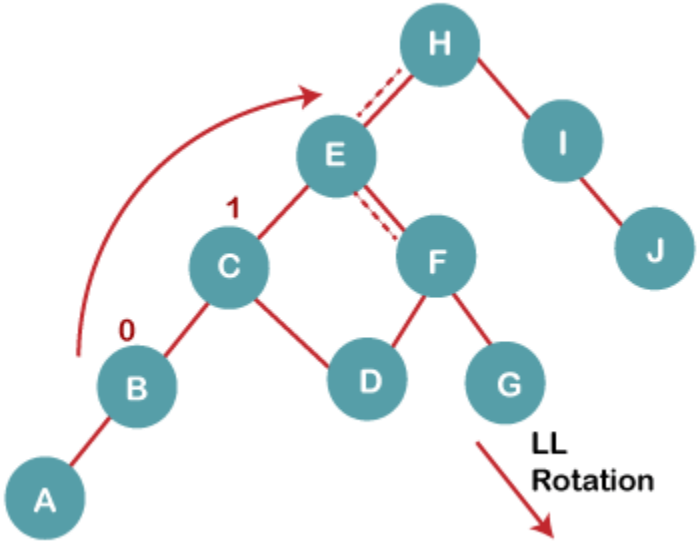


5. Insert G

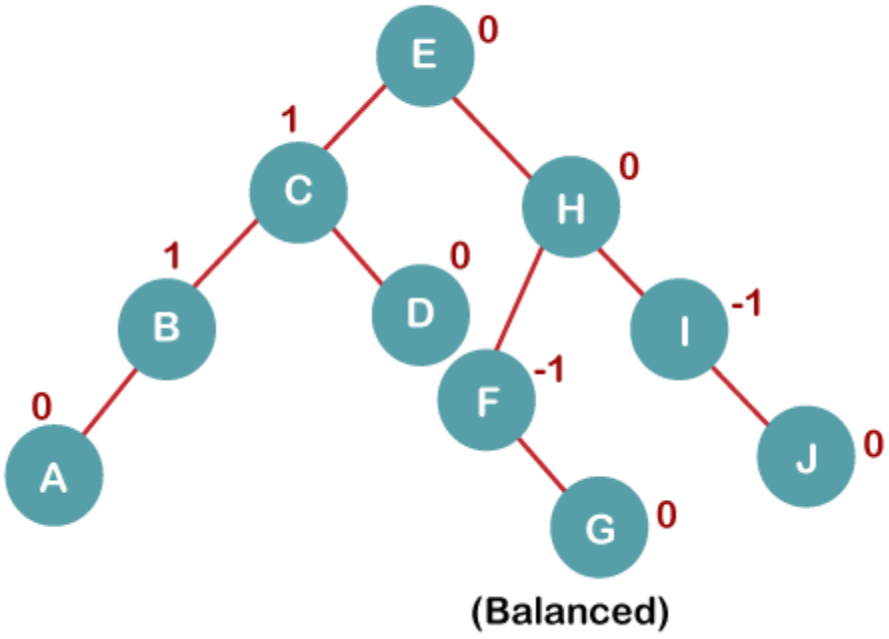


On inserting G, BST become unbalanced as the Balance Factor of H is 2, since if we travel from G to H, we find that it is inserted in the left subtree of right subtree of H, we will perform LR Rotation on node I. LR = RR + LL rotation.

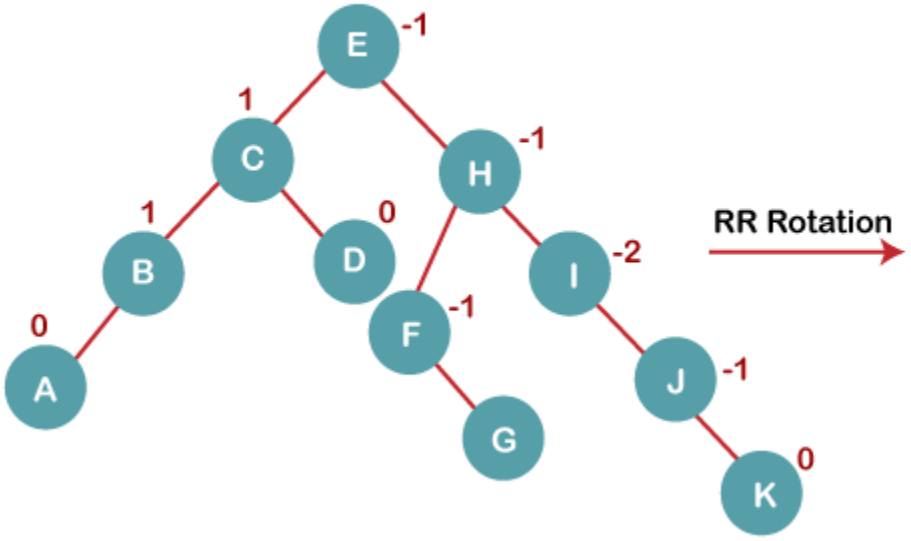
5 a) We first perform RR rotation on node C
The resultant tree after RR rotation is:



5 b) We then perform LL rotation on node H
The resultant balanced tree after LL rotation is:



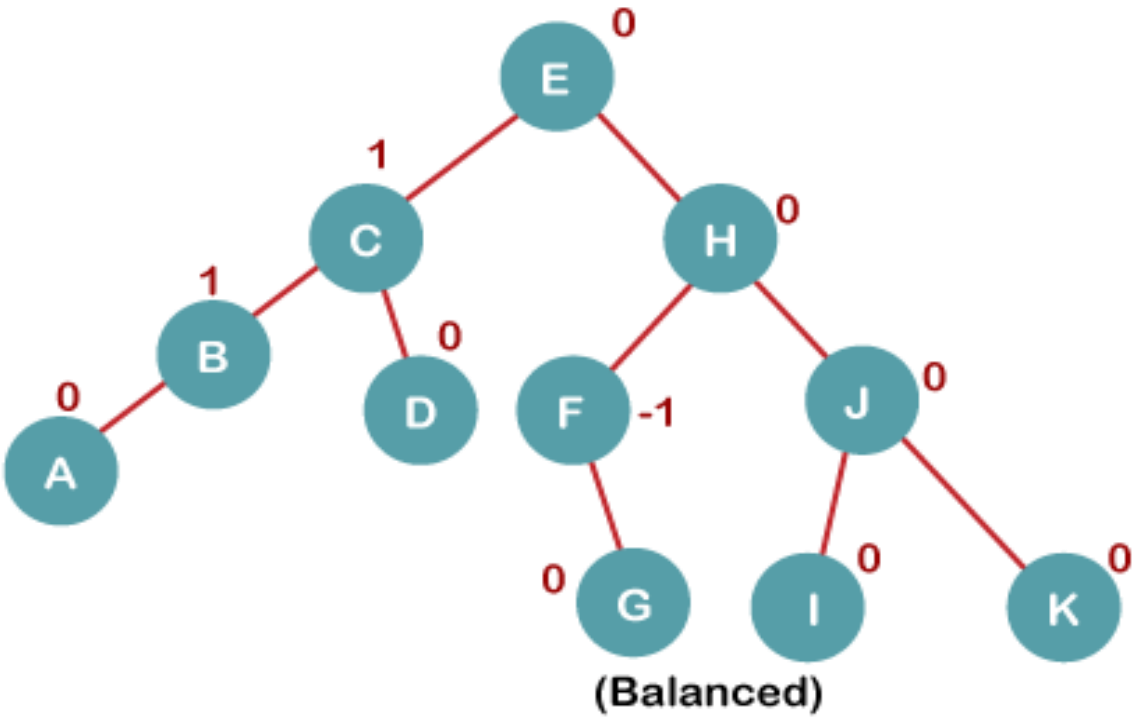
6. Insert K



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.

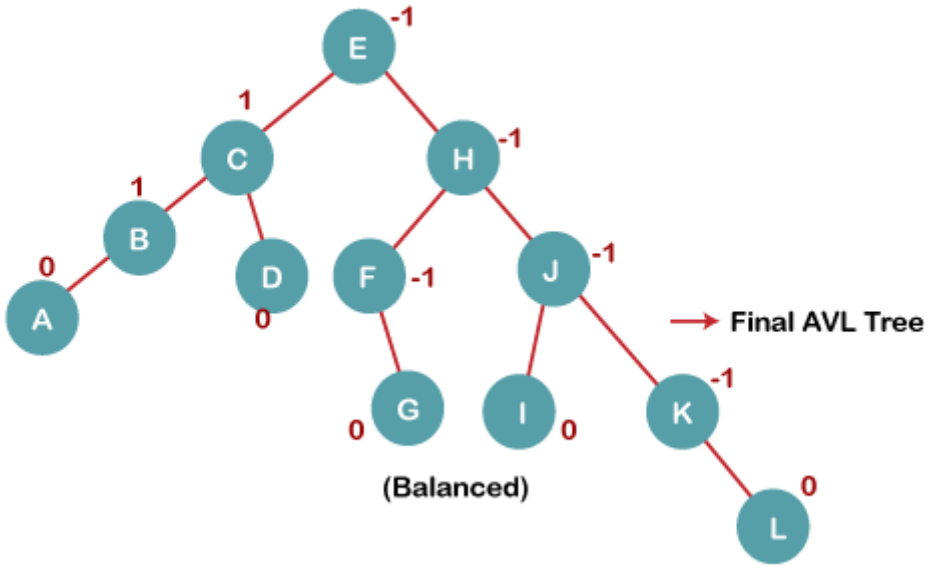
Trees

The resultant balanced tree after RR rotation is:



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



Deletion in AVL Tree

Deleting a node from an AVL tree is similar to that in a binary search tree. Deletion may disturb the balance factor of an AVL tree and therefore the tree needs to be rebalanced in order to maintain the AVLness. For this purpose, we need to perform rotations. The two types of rotations are L rotation and R rotation. Here, we will discuss R rotations. L rotations are the mirror images of them.

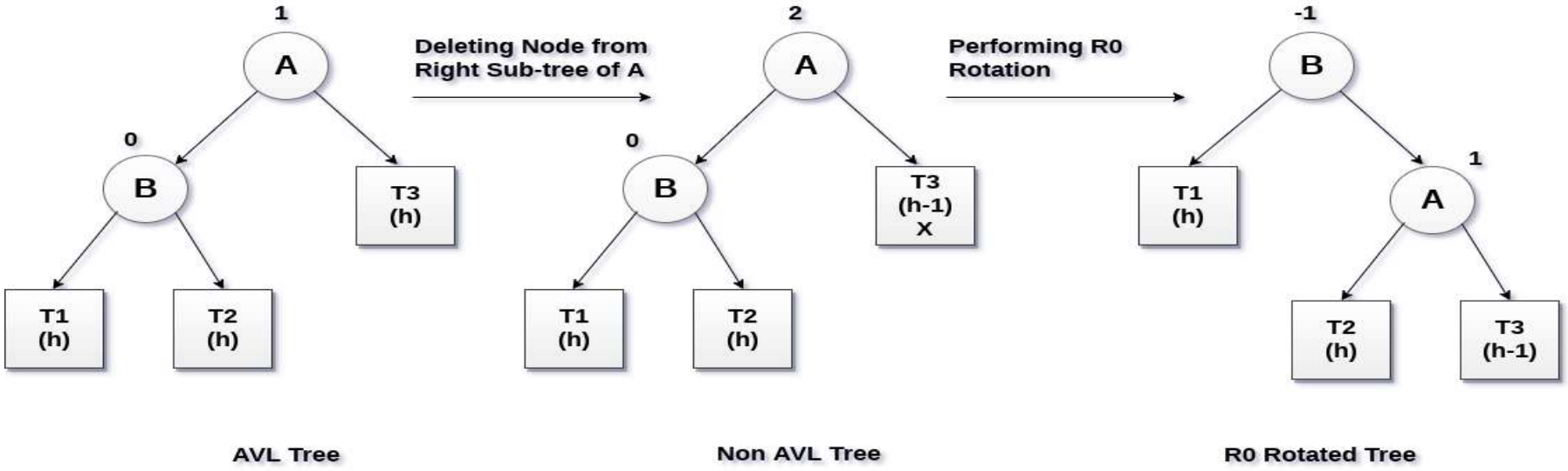
If the node which is to be deleted is present in the left sub-tree of the critical node, then L rotation needs to be applied else if, the node which is to be deleted is present in the right sub-tree of the critical node, the R rotation will be applied.

Let us consider that, A is the critical node and B is the root node of its left sub-tree. If node X, present in the right sub-tree of A, is to be deleted, then there can be three different situations:

R0 rotation (Node B has balance factor 0)

If the node B has 0 balance factor, and the balance factor of node A disturbed upon deleting the node X, then the tree will be rebalanced by rotating tree using R0 rotation.

The critical node A is moved to its right and the node B becomes the root of the tree with T1 as its left sub-tree. The sub-trees T2 and T3 becomes the left and right sub-tree of the node A. the process involved in R0 rotation is shown in the following image.

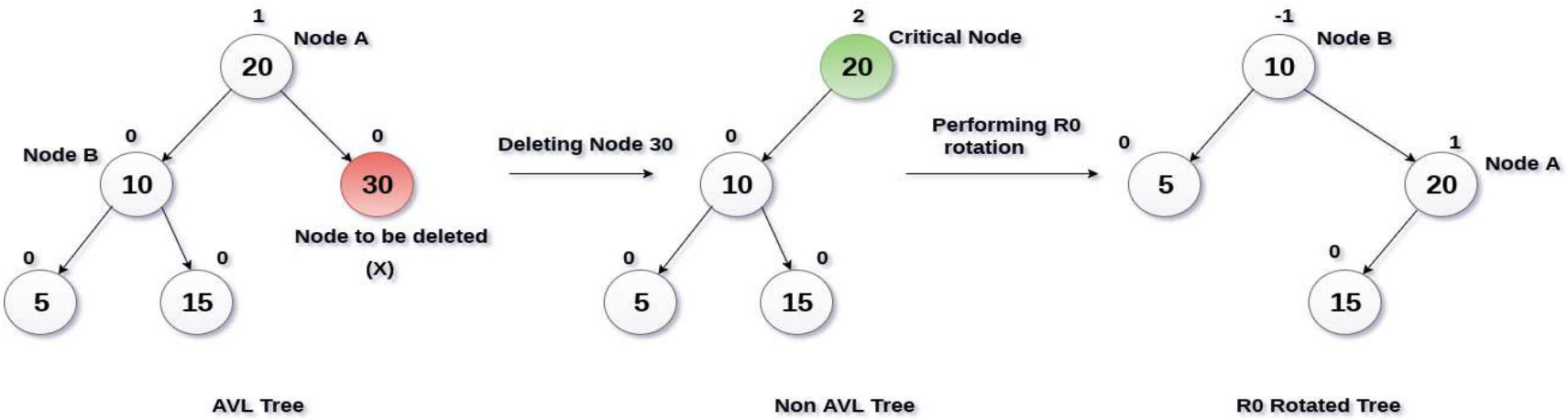


Example:

Delete the node 30 from the AVL tree shown in the following image.

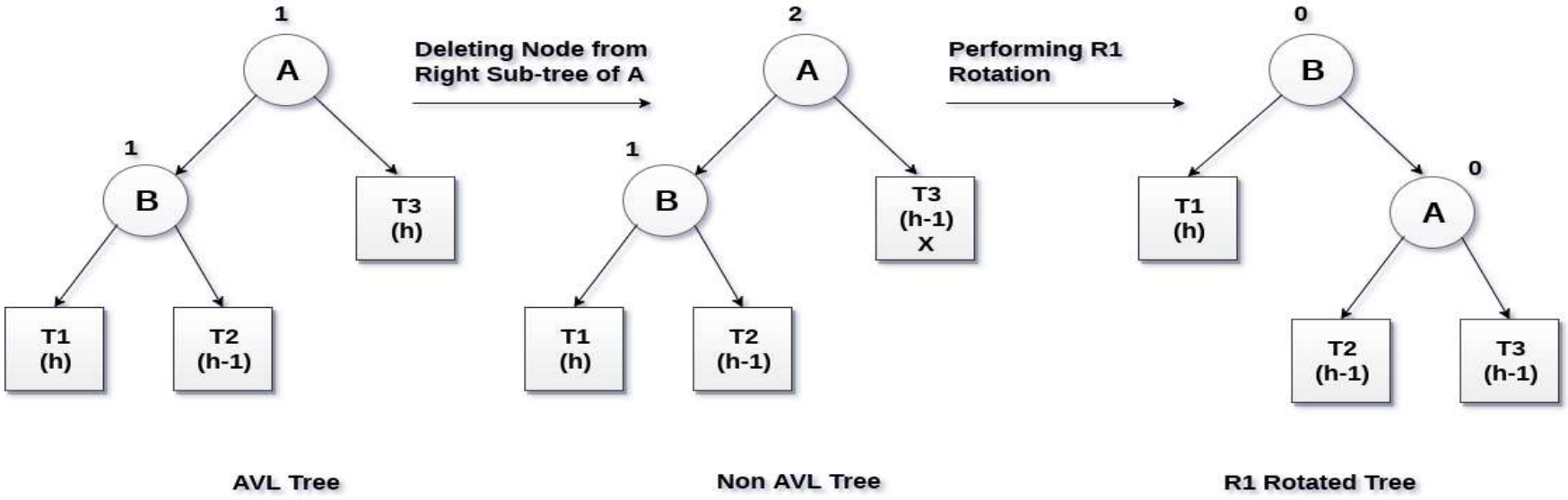
Solution

In this case, the node B has balance factor 0, therefore the tree will be rotated by using R0 rotation as shown in the following image. The node B(10) becomes the root, while the node A is moved to its right. The right child of node A will now become the left child of node A.



R1 Rotation (Node B has balance factor 1)

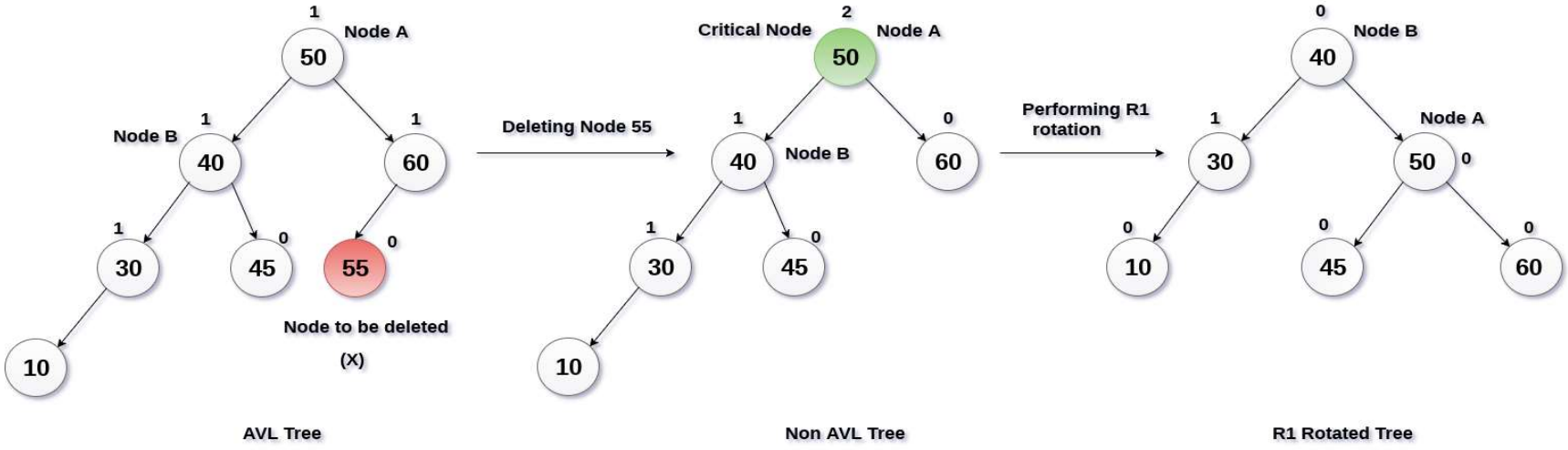
R1 Rotation is to be performed if the balance factor of Node B is 1. In R1 rotation, the critical node A is moved to its right having sub-trees T2 and T3 as its left and right child respectively. T1 is to be placed as the left sub-tree of the node B.
The process involved in R1 rotation is shown in the following image.



Example

Delete Node 55 from the AVL tree shown in the following image.

Deleting 55 from the AVL Tree disturbs the balance factor of the node 50 i.e. node A which becomes the critical node. This is the condition of R1 rotation in which, the node A will be moved to its right (shown in the image below). The right of B is now become the left of A (i.e. 45). The process involved in the solution is shown in the following image.

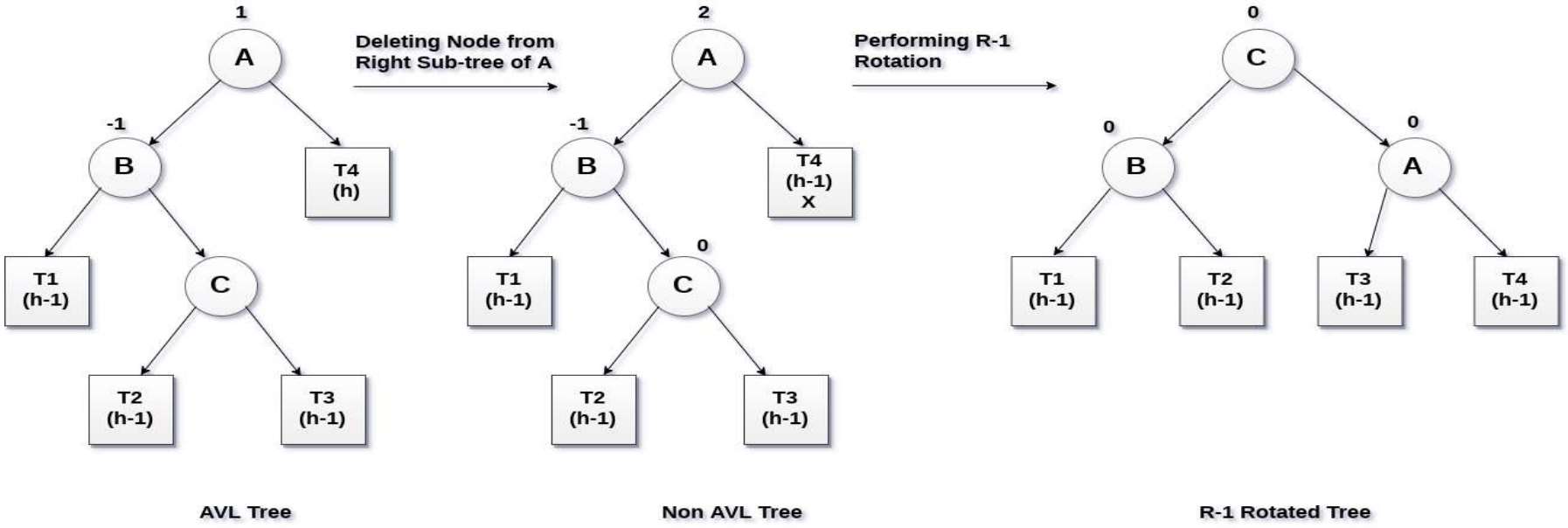


R-1 Rotation (Node B has balance factor -1)

R-1 rotation is to be performed if the node B has balance factor -1. This case is treated in the same way as LR rotation. In this case, the node C, which is the right child of node B, becomes the root node of the tree with B and A as its left and right children respectively.

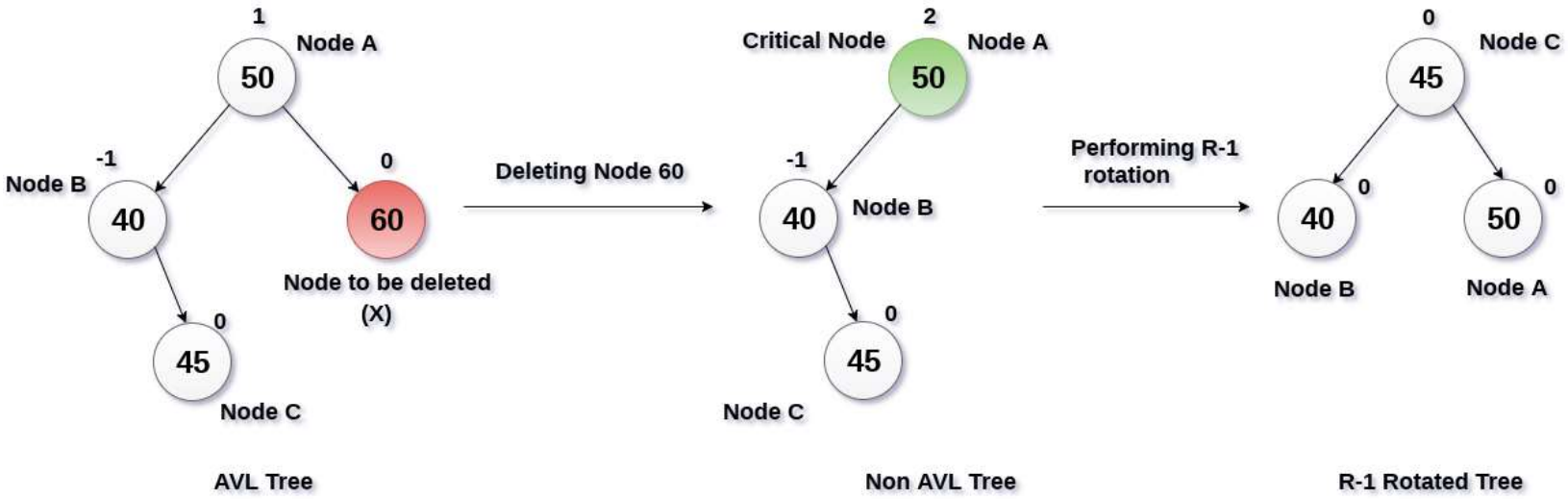
The sub-trees T1, T2 becomes the left and right sub-trees of B whereas, T3, T4 become the left and right sub-trees of A.

The process involved in R-1 rotation is shown in the following image.



Example

Delete the node 60 from the AVL tree shown in the following image.



Weekly Assignment

1. Prove that number of nodes in a binary tree of height h is 2h+1-1.
2. Prove that minimum height of a binary tree with node n is log2 n.
3. Implement iterative functions for preorder, inorder and postorder traversal.
4. Construct a binary tree from given tree traversals-
   Preorder: G,B,Q,A,C,K,F,P,D,E,R,H
   Inorder: Q,B,K,C,F,A,G,P,E,D,H,R
5. Construct a binary tree from given tree traversals-
   Postorder: 4,5,2,6,7,3,1
   Inorder: 4,2,5,1,6,3,7
6. Construct a binary tree for the following expressions-

I. (A<B)&&(B<C)&&(C-D)

II. A+(B+C*D+E)+F/G

Weekly Assignment

7. Generate a binary search tree by inserting following integers
   50, 15, 62, 5, 20, 58, 91, 3, 8, 37, 60, 24.
8. Write a function to delete a node from binary search tree.
9. Write properties of B-Tree. Construct a B tree of order 5 by inserting following data:
   H,D,K,Z,B,P,Q,E,A,S,W,T,C,L,N,Y,M.
10. With the help of diagrams explain all the rotations of AVL tree. Construct tree for the following list of elements:
    3,5,11,8,4,1,12,7,2,6,10.

MCQ s

1. How many children does a binary tree have?
   a) 2
   b) any number of children
   c) 0 or 1 or 2
   d) 0 or 1

2. What is/are the disadvantages of implementing tree using normal arrays?
   a) difficulty in knowing children nodes of a node
   b) difficult in finding the parent of a node
   c) have to know the maximum number of nodes possible before creation of trees
   d) difficult to implement

MCQ s

3. What must be the ideal size of array if the height of tree is ‘l’?

a) 2^l-1

b) l-1

c) l

d) 2l

4. What are the children for node ‘w’ of a complete-binary tree in an array representation?

a) 2w and 2w+1

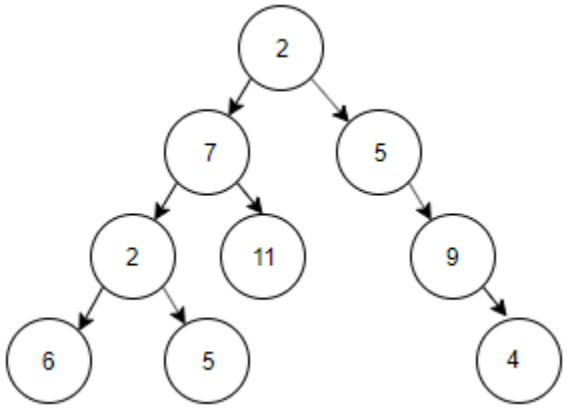
b) 2+w and 2-w

c) w+1/2 and w/2

d) w-1/2 and w+1/2

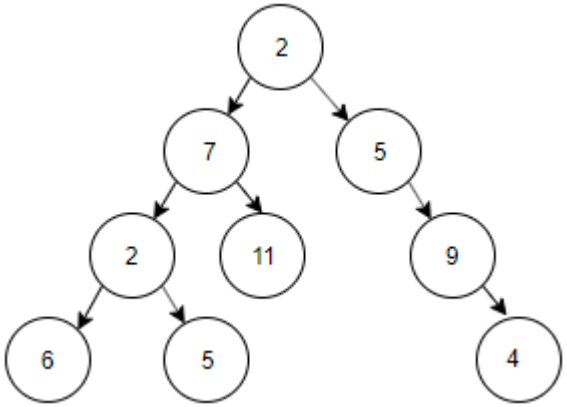
MCQ s

5. For the tree below, write the pre-order traversal.



a) 2, 7, 2, 6, 5, 11, 5, 9, 4
b) 2, 7, 5, 2, 6, 9, 5, 11, 4
c) 2, 5, 11, 6, 7, 4, 9, 5, 2
d) 2, 7, 5, 6, 11, 2, 5, 4, 9

6. For the tree below, write the post-order traversal.



a) 2, 7, 2, 6, 5, 11, 5, 9, 4
b) 2, 7, 5, 2, 6, 9, 5, 11, 4
c) 2, 5, 11, 6, 7, 4, 9, 5, 2
d) 2, 7, 5, 6, 11, 2, 5, 4, 9

MCQ s

7. What is the maximum number of children that a binary tree node can have?

a) 0

b) 1

c) 2

d) 3

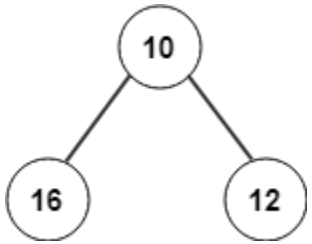
8. The following given tree is an example for?

a) Binary tree

b) Binary search tree

c) Fibonacci tree

d) AVL tree



MCQ s

9. A binary tree is a rooted tree but not an ordered tree.

a) True

b) False

10. What is the traversal strategy used in the binary tree?

a) depth-first traversal

b) breadth-first traversal

c) random traversal

d) Priority traversal

11. How many types of insertion are performed in a binary tree?

a) 1

b) 2

c) 3

d) 4

Expected Questions for University Exam

1. Define Binary search tree. How it is different from binary tree?
2. Suppose the following list of letters is inserted in order into empty binary search tree: J, R, D, G, T, E, M, H, P, A, F, Q

(i) Construct the binary Search Tree

(ii) Find the in-order, Pre-order and Post-order Traversal of the BST created.

3. Discuss the advantages of using B-Tree. Insert the following Information 86, 23, 91, 4, 67, 18, 32, 54, 46, 96, 45 into an empty B-Tree with order 4.

4. Construct the binary tree using following in-order and post-order traversal.

In-order : DBMINEAFCJGK

Post-order : ABDEIMNCFGJK

Expected Questions for University Exam

5. What is a height balanced Tree? Why height balancing of Tree is required?

Create an AVL Tree for the following elements: a, z, b, y, c, x, d, w, e, v, f

6. What is binary tree? Explain.

7. What is binary search tree? Write the important applications of binary search tree. Write algorithm to delete a node from a binary search tree.

Summary

- Unlike Array and Linked List, which are linear data structures, tree is hierarchical (or non-linear) data structure.
- One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer file system
- Binary Search Tree is a tree that allows fast search, insert, delete on a sorted data. It also allows finding closest item

References

[1] Aaron M. Tenenbaum, Yedidyah Langsam and Moshe J. Augenstein, "Data Structures Using C and C++", PHI Learning Private Limited, Delhi India

[2] Horowitz and Sahani, "Fundamentals of Data Structures", Galgotia Publications Pvt Ltd Delhi India.

[3] Lipschutz, "Data Structures" Schaum's Outline Series, Tata McGraw-hill Education (India) Pvt. Ltd.

[4] Thareja, "Data Structure Using C" Oxford Higher Education.

[5] AK Sharma, "Data Structure Using C", Pearson Education India.

Thank you