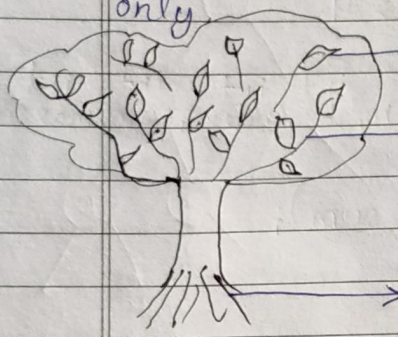
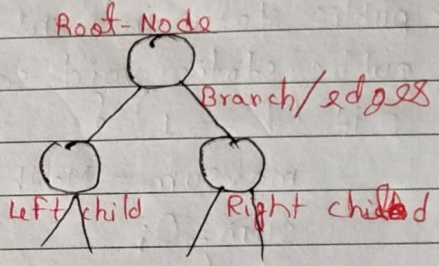
BINARY TREES - I

* Binary tree is a Hierarchical data structure.
Ex → Family Tree

* In Binary tree, at max we can have 2 children only.





* "A Binary tree is a hierarchical data structure in which each node has at most two children i.e., left child & right child."

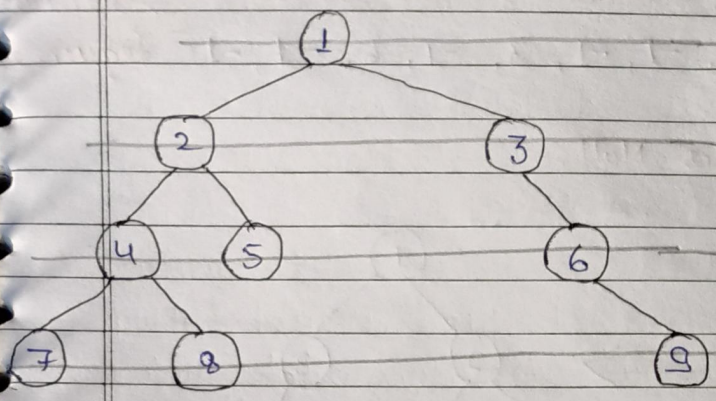
Node – Each element in a Binary tree.
Data – Value stored in the node.
Left Child – Pointer or Reference to left of the current node.
Right Child – Pointer or Reference to right of the current node.
Root Node – Topmost node in the tree.
Leaf Nodes – Nodes that do not have any children.

Height of Tree – no. of edges from root to deepest leaf node.

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

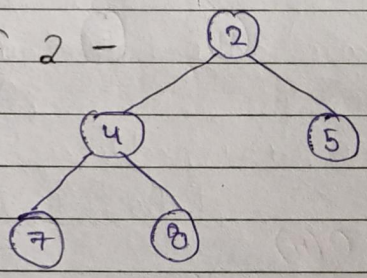
Subtree – Any node in tree, along with its descendants, forms a subtree. The left & right children of a node are considered subtrees of that node.

★ LEVELS & SUBTREE IN A TREE

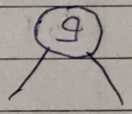


Level - 1 , Depth - 1
Level - 2 , Depth - 2
Level - 3 , Depth - 3
Level - 4 , Depth - 4

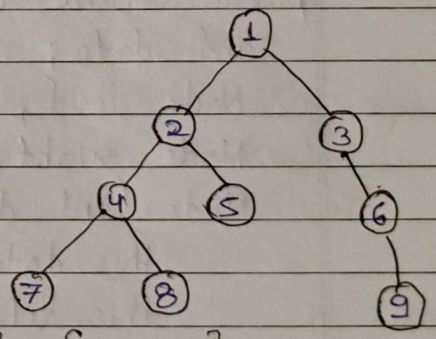
* Subtree of 2 -



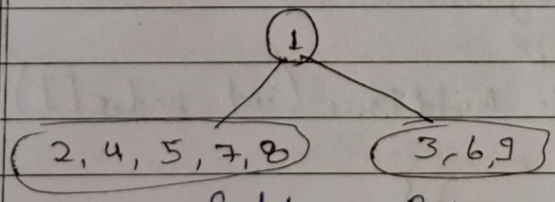
* Subtree of 9 -



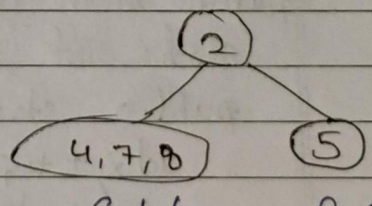
★ QUIZ



a) Children of 4 : 7 & 8
b) No. of leaves : 4 (7, 8, 5, 9)
c) Parent of 6 : 3
d) Level of 2 : 2
e) Subtrees of 1 ~~& 2~~ : {2, 4, 5, 7, 8}, {3, 6, 9}
f) Ancestors of 8 : (4, 2, 1)
g) Subtrees of 2 : {4, 7, 8} {5}.



Subtree of 1



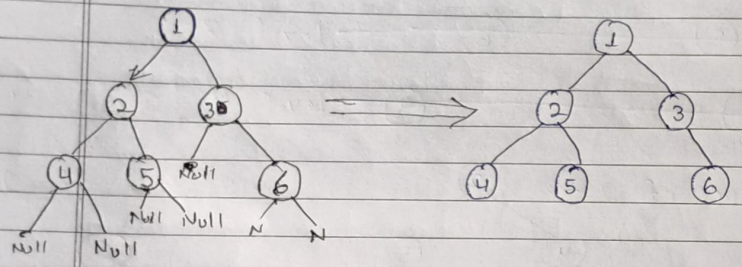
Subtree of 2.

★ BUILD TREE PREORDER

Node → left subtree → right subtree ⇒ Preorder

1, 2, 4, -1, -1, 5, -1, -1, 3, -1, 6, -1, -1 } Sequence

-1 indicates Null.



⇒ Time Complexity → O(n)

```
public static class Node {
    int data;
    Node left;
    Node right;
    Node (int data) {
        this.data = data;
        this.left = null;
        this.right = null;
    }
}
public static class BinaryTree {
    static int idx = -1;
    public static Node buildTree (int nodes[]) {
        idx++;
        if (nodes[idx] == -1) {
            return null;
        }
        Node newNode = new Node (nodes[idx]);
        newNode.left = buildTree (nodes);
        newNode.right = buildTree (nodes);
        return newNode;
    }
}
public static void main (String args[]) {
    int nodes[] = {1, 2, 4, -1, -1, 5, -1, -1, 3, -1, 6, -1, -1};
    BinaryTree tree = new BinaryTree();
    Node root = tree.buildTree (nodes);
    Sy so (root.data);
}
```

Output : 1

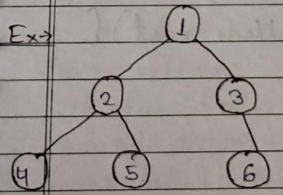
★ TREE TRAVERSAL

Three types of traversal are :-

① PreOrder ② InOrder ③ PostOrder

a) PreOrder ⇒ Root → Left Subtree → Right Subtree
b) InOrder ⇒ Left Subtree → Root → Right Subtree
c) PostOrder ⇒ Left Subtree → Right Subtree → Root

Ex →



* PreOrder Traversal // O(n)
⇒ 1 2 4 5 3 6

* InOrder Traversal // O(n)
⇒ 4 2 5 1 3 6

* PostOrder Traversal // O(n)
⇒ 4 5 2 6 3 1

In Trees, we use Recursion.

We use recursion, in tree we go child leaves to root.

Page No. 355
Date : / /

⇒ // Pre-Order

```
public static void preorder (Node root) {
    if (root == null) {
        return;
    }
    System.out.print (root.data + " ");
    preorder (root.left);
    preorder (root.right);
}
```

⇒ // In-Order

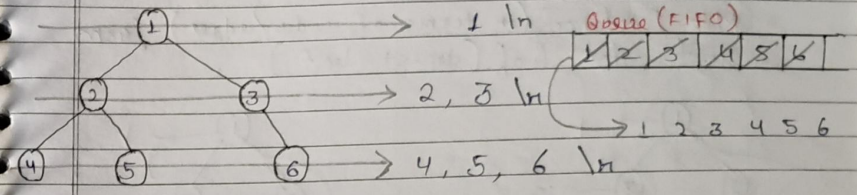
```
public static void inorder (Node root) {
    if (root == null) {
        return;
    }
    inorder (root.left);
    System.out.print (root.data + " ");
    inorder (root.right);
}
```

⇒ // Post-Order

```
public static void postorder (Node root) {
    if (root == null) {
        return;
    }
    postorder (root.left);
    postorder (root.right);
    System.out.print (root.data + " ");
}
```

Page No. 356
Date : / /

★ LEVEL ORDER TRAVERSAL



∴ Level-wise print :- 1 2 3 4 5 6

* In Pre, In & Post Orders, we follow the Depth First Search (DFS).

* In Level Order, we follow :- Breadth First Search (BFS).

* \n (next line) → null

| 1 | null | 2 | 3 | null | 4 | 5 | 6 | null |

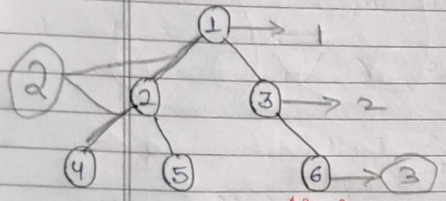
// O(n)

⇒ // Level Order

```
public static void levelOrder (Node root) {
    if (root == null)
        return;
    Queue <Node> q = new LinkedList <> ();
    q.add (root);    q.add (null);
    while (!q.isEmpty)
        Node currNode = q.remove();
        if (currNode == null)
            Syso();
            if (q.isEmpty())
                break
            else
                q.add (null);
        else
            Syo (currNode.data + " ");
            if (currNode.left != null)
                q.add (currNode.left);
            if (currNode.right != null)
                q.add (currNode.right);
    }
```

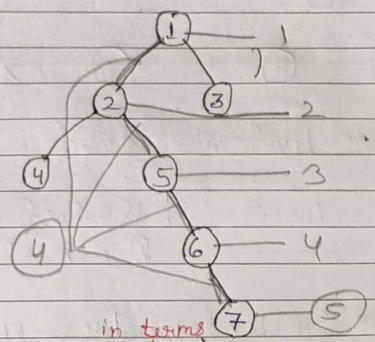
★ HEIGHT OF A TREE

* Max-Distance (in terms of nodes/edges) from root to leaf (depest leaf)



in terms of node

* Height (nodes) → 3
* Height (edges) → 2



in terms

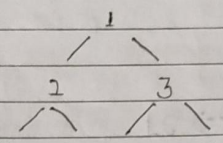
* Height (nodes) → 5
* Height (edges) → 4

(In Terms of node)

⇒
```
public static class Node {          //O(n)
    Node left, right;
    public Node (int data) {
        this.data = data;
        this.left = null;
        this.right = null;
    }
}
public static int height (Node root) {
    if (root == null) {
        return 0;
    }
    int lh = height (root.left);
    int rh = height (root.right);
    return Math.max (lh, rh) + 1;
}
```

→ it will show errors. Use Build tree code part.

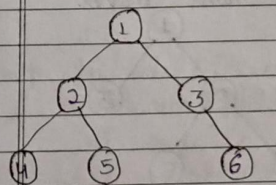

```
public static void main (String args[]) {
```




```
    Node root = new Node (1);
    root.left = new Node (2);
    root.right = new Node (3);
    root.left.left = new Node (4);
    root.left.right = new Node (5);
    root.right.left = new Node (6);
    root.right.right = new Node (7);
    Syso (height (root));
}
```

Output : 3

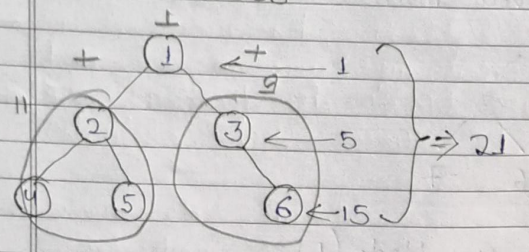
★ COUNT OF NODES



⇒ Count = 6

⇒
```
public static int count (Node root) {
    if (root == null) {
        return 0;
    }
    int leftCount = count (root.left);
    int rightCount = count (root.right);
    return leftCount + rightCount + 1;
}
```

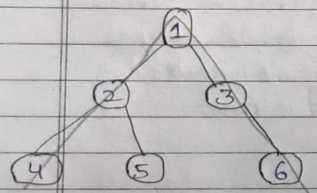
★ SUM OF NODES



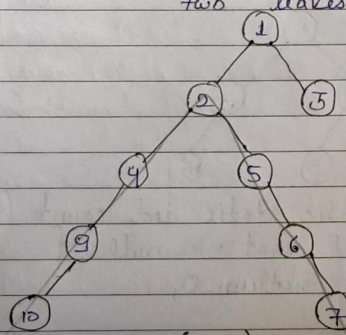

```
⇒ public static int sum(Node root) {
    if (root == null) {
        return 0;
    }
    int leftSum = sum(root.left);
    int rightSum = sum(root.right);
    return leftSum + rightSum + root.data;
}
```

★ DIAMETER OF A TREE

→ no. of nodes in the longest path b/t two leaves.



* Diameter = 5 (4 to 6)

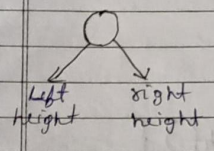


* Diameter (9-7) = 6
* Diameter (9-3) = 5
* Diameter (7-3) = 6
* Diameter (10-7) = 7 (Diameter of tree)

Case 1: पूरे Tree का Diameter root node के through Traverse करे या।

Case 2: Diameter subtree main Traverse करे।

* Approach 1 – // $O(n^2)$



// Diameter passes through root
→ D = lh + rh + 1.
// Diameter doesn't passes root
→ lD ; rD
// Comparing
→ lD, rD & D
// The largest one will be Diameter of tree.

⇒ //Inside Height of a Tree (code) we're writing it.

```
public static int diameter(Node root) {
    if (root == null) {
        return 0;
    }
    int leftDiam = diameter(root.left);
    int leftHt = height(root.left);
    int rightDiam = diameter(root.right);
    int rightHt = height(root.right);
    int selfDiam = leftHt + rightHt + 1;

    return Math.max(selfDiam, Math.max(leftDiam,
                                       rightDiam));
}
```

* Approach 2 - O(n)

→ In this approach, height & Diameter will be done in a same function.

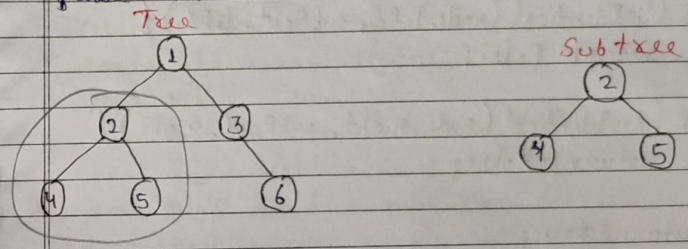
→ इस Approach में Height साथ-साथ में ही Calculate होगा, अलग से नहीं।

⇒
```
static class Info {
    int diam;
    int ht;
    public Info (int diam, int ht) {
        this.diam = diam;
        this.ht = ht;
    }
}
public static Info diameter (Node root) {
    if (root == null) {
        return new Info (0, 0);
    }
    Info leftInfo = diameter (root.left);
    Info rightInfo = diameter (root.right);
    int diam = Math.max (Math.max (leftInfo.diam,
                rightInfo.diam), leftInfo.ht +
                rightInfo.ht + 1);
    int ht = Math.max (leftInfo.ht, rightInfo.ht)+1;
    return new Info (diam, ht);
}
public static void main (String args[]) {

    Info result = diameter (root);
    Sysо (result.diam);
}
```


BINARY TREES - II

★ SUBTREE OF ANOTHER TREE - Given the roots of two binary trees root & subRoot, return true if there is a subtree of root with the same structure & node value of subRoot & false otherwise.



* Approach → if (subroot == node of any tree) & the child node are identical return true;

① Find subRoot in tree → ~~traversal~~ (root.data == subRoot.data)

② Check identical (subtree, node subtree)

→ Non-identical Condition

① Node.data != subRoot.data
② node = null || subRoot = null
③ leftSubtree → non-identical
④ rightSubtree → non-identical

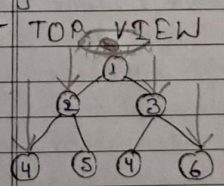
⇒
```
public static boolean isIdentical (Node node, Node subRoot){
  if (node == null && subRoot == null) {
      return true;
  } else if (node == null || subRoot == null || node.data != subRoot.data){
      return false;
  }
  if (!isIdentical (node.left, subRoot.left) {
      return false;
  }
  if (!isIdentical (node.right, subRoot.right) {
      return false;
  }
  return true;
}
public static boolean isSubtree (Node root, Node subRoot){
  if (root == null) {
      return false;
  }
  if (root.data == subRoot.data) {
      if (isIdentical (root, subRoot)){
          return true;
      }
  }
  return isSubtree (root.left, subRoot) ||
         isSubtree (root.right, subRoot);
}
```

//NEXT PAGE

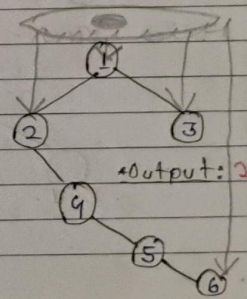

```
public static void main (String[] args) {
/*
        1
       / \
      2   3
     / \ / \
    4  5 6  7
*/
  Node root = new Node (1);
  root.left = new Node (2);
  root.right = new Node (3);
  root.left.left = new Node (4);
  root.left.right = new Node (5);
  root.right.left = new Node (6);
  root.right.right = new Node (7);

/**
      2
     / \
    4   5
*/
  Node subRoot = new Node (2);
  subRoot.left = new Node (4);
  subRoot.right = new Node (5);
  Syso (isSubtree (root, subRoot));
}
```

★ TOP VIEW OF A TREE



*Output: 4, 2, 1, 3, 6



*Output: 2, 1, 3, 6

* HashMap in Java (Briefly)

⇒ map (key, value)
↓
unique

Ex⇒

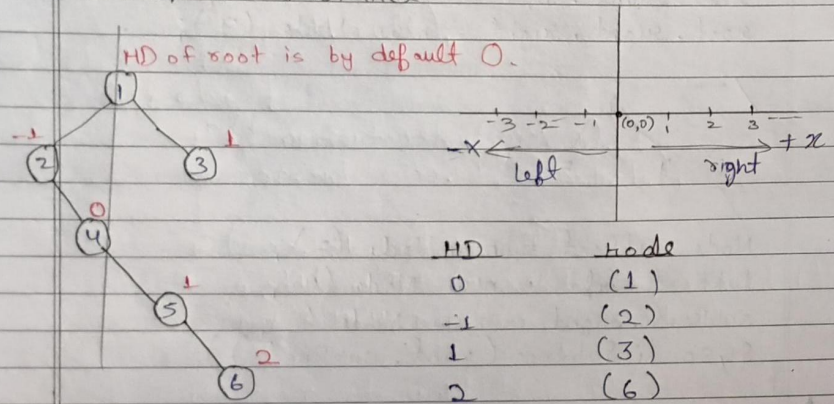
Countries	Population
India	500
china	100
Pakistan	200
Nepal	200

→ Syntax for Creating map:

```
// import java.util.HashMap;
// HashMap <Data-type of key, Value> map = new HashMap();
```

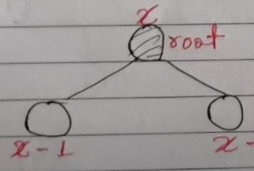
O(1) → In HashMap, to add (put) → map.put(key, value)
O(1) → In HashMap, to Get → map.get(key)

* Horizontal Distance

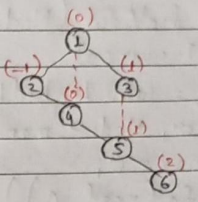


HD	Node
0	(1)
-1	(2)
1	(3)
2	(6)

∴ Top View = 1 2 3 6



* Approach



map

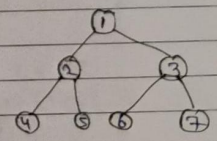
key (HD)	Value (node)
(0)	1
(-1)	2
(1)	3
(2)	6

→ Level Order Traversal.
→ min = ∅ -1
→ max = ∅ ✗ 2

→ loop (from min to max)
-1 → map.get(-1) → 2
0 → map.get(0) → 1
1 → map.get(01) → 3
2 → map.get(2) → 6
Output

⇒

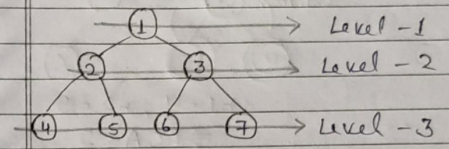
```
static class Info {
    Node node;
    int hd;
    public Info (Node node, int hd) {
        this.node = node;
        this.hd = hd;
    }
}
public static void main (String args[]) {
    Node root = new Node(1);
    root.left = new Node(2);
    root.right = new Node(3);
    root.left.left = new Node(4);
    root.left.right = new Node(5);
    root.right.left = new Node(6);
    root.right.right = new Node(7);
    topView(root); // 4 2 1 3 7
}
```



```
public static void topView (Node root) {
  //Level Order Traversal
  Queue<Info> q = new LinkedList<>();
  HashMap<Integer, Node> map = new HashMap<>();
  int min = 0, max = 0;
  q.add (new Info (root, 0));
  q.add (null);
  while (!q.isEmpty()) {
    Info curr = q.remove();
    if (curr == null) {
      if (q.isEmpty()) {
        break;
      } else {
        q.add(null);
      }
    } else {
      if (!map.containsKey(curr.hd)) {
        map.put (curr.hd, curr.node);
      }
      if (curr.node.left != null) {
        q.add (new Info (curr.node.left, curr.hd-1));
        min = Math.min (min, curr.hd-1);
      }
      if (curr.node.right != null) {
        q.add (new Info (curr.node.right, curr.hd+1));
        max = Math.max (max, curr.hd+1);
      }
    }
  }
  for (int i=min; i<=max; i++) {
    System.out.print (map.get(i).data + " ");
  }
  System.out();
}
```


BINARY TREES - III

★ Kth LEVEL OF A TREE



→ K = 3 // K is Level.

→ Output: 4, 5, 6, 7

* Approach - O(n)

→ PreOrder Traversal.

→ Root → Leftsubtree → Rightsubtree

* Pseudo Code

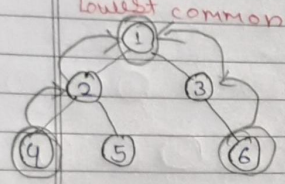
```
KLevel (root, level, k)
if (root == null)
    return;
if (level == k) sprint(root.data) => return
KLevel (root.left, level+1, k)
KLevel (root.right, level+1, k)
```

```
public static void KLevel (Node root, int level, int k) {
  if (root == null) {
    return;
  }
  if (level == k) {
    System.out.print (root.data + " ");
    return;
  }
  KLevel (root.left, level+1, k);
  KLevel (root.right, level+1, k);
}
```

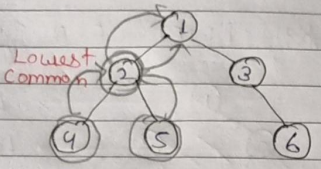
Input: int k = 3

KLevel (root, 1, k);

★ LOWEST COMMON ANCESTOR (Approach-1)

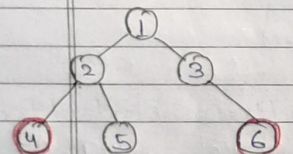


n1 = 4, n2 = 6
⇒ LCA = 1



n1 = 4, n2 = 5
⇒ LCA = 2

* Approach O(n)



→ path 1 (n1) : | 1 | 2 | 4 |
→ path 2 (n2) : | 1 | 3 | 6 |

{ Root to node

∴ LCA = 1

Ex: n1 = 4, n2 = 5
→ path (n1) : | 1 | 2 | 4 |
→ path (n2) : | 1 | 2 | 5 |

compare
LCA

∴ LCA = 2

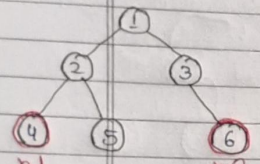
(1) Root to node — Array List
getPath (root, node, AL path)

(2) for (i = 0 to path1.length & path2.length)
Last common ancestor
means
Last common node.


```
⇒ public static boolean getPath (Node root, int n, ArrayList
                                 <Node> path){
    if (root == null) {
        return false;
    }
    path.add (root);
    if (root.data == n) {
        return true;
    }
    boolean foundLeft = getPath (root.left, n, path);
    boolean foundRight = getPath (root.right, n, path);
    if (foundLeft || foundRight) {
        return true;
    }
    path.remove (path.size () - 1);
    return false;
}
public static Node lca (Node root, int n1, int n2) {
    Arraylist <Node> path1 = new Arraylist <>();
    Arraylist <Node> path2 = new Arraylist <>();
    getPath (root, n1, path1);   getPath (root, n2, path2);
    //Last Common Ancestor
    int i = 0
    for (; i<path1.size() && i<path2.size(); i++)
        if (path1.get (i) != path2.get (i))
            break;
    //Last equal node -> i-1th
    Node lca = path1.get (i-1);
    return lca;
}
```

Input: int n1 = 4, n2 = 7
Syso (lca (root, n1, n2).data);

★ LOWEST COMMON ANCESTOR (Approach-2)



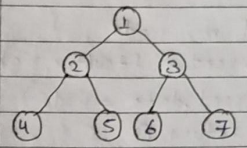
* LCA = 1st ancestor or node
↓ जिसके अंडर
subtree n1 & n2 exist

① root = null
root = n1
root = n2
} return root
⇒ left LCA
→ Right LCA

② rightLCA = null → leftLCA; leftLCA = null → rightLCA
③ right & left != null → return root

⇒
```
public static Node lca2 (Node root, int n1, int n2) {
  if (root == null || root.data == n1 || root.data == n2) {
    return root;
  }
  Node leftLCA = lca2 (root.left, n1, n2);
  Node rightLca = lca2 (root.right, n1, n2);
  // leftLca = valid, rightLca = null
  if (rightLca == null) {
    return leftLca;
  }
  if (leftLca == null) {
    return rightLca;
  }
  return root;
}
```


★ MIN DISTANCE BETWEEN NODES

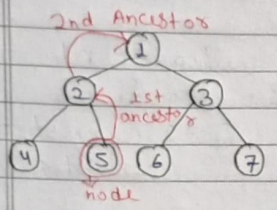


① Dist 1 = lca to n1
② Dist 2 = lca to n2
③ Dist 1 + Dist 2 = min. dist b/t two nodes.

* n1 = 4 , * n2 = 6
→ Distance = 4

⇒
```
public static int lcaDist (Node root, int n) {
  if (root == null) {
    return -1;
  }
  if (root.data == n) {
    return 0;
  }
  int leftDist = lcaDist (root.left, n);
  int rightDist = lcaDist (root.right, n);
  if (leftDist == -1 && rightDist == -1) {
    return -1;
  } else if (leftDist == -1) {
    return rightDist + 1;
  } else {
    return leftDist + 1;
  }
}
public static int minDist (Node root, int n1, int n2) {
  Node lca = lca2 (root, n1, n2);
  int dist1 = lcaDist (lca, n1);
  int dist2 = lcaDist (lca, n2);
  return dist1 + dist2;
}
```

★ Kth ANCESTOR OF NODE



S1: find my node
↳ root, leftsubtree, rightsubtree

S2: if (root.data = node)
return 0

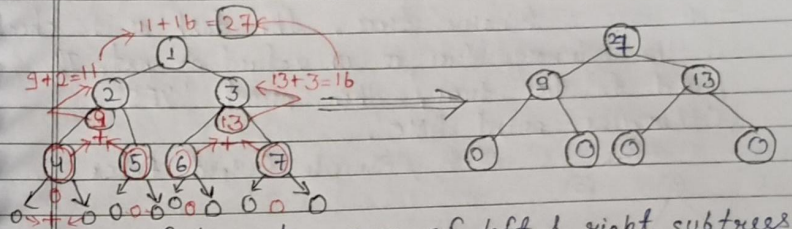
S3: Calculate right Dist, left Dist.
* -1, -1 return → -1
* Valid value → (dist + 1) == k
print root.data

*node = 5, k = 2
⇒ ans = 1
it is basically Distance

```
⇒ public static int KAncestor (Node root, int n, int k){
    if (root == null) {
        return -1;
    }
    if (root.data == n) {
        return 0;
    }
    int leftDist = KAncestor (root.left, n, k);
    int rightDist = KAncestor (root.right, n, k);
    if (leftDist == -1 && rightDist == -1) {
        return -1;
    }
    int max = Math.max (leftDist, rightDist);
    if (max+1 == k) {
        Syso (root.data);
    }
    return max+1;
}
```

* Input : int n = 5, k = 2;
KAncestor (root, n, k)
* Output : 1

★ TRANSFORM TO SUM TREE - //O(n)



* Each node = sum of left & right subtrees

S1: if (root == null)
return 0.

S2: Calculate left child, right child;

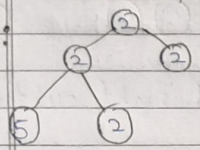
S3: data = root.data
root.data = root.left + leftchild + root.right + rightchild;
return data;

```
⇒ public static int transform (Node root) {
    if (root == null)
        return 0;
    int leftChild = transform (root.left);
    int rightChild = transform (root.right);
    int data = root.data;
    int newleft = root.left == null ? 0 : root.left.data;
    int newRight = root.right == null ? 0 : root.right.data;
    root.data = newleft + leftChild + newRight + rightChild;
    return data;
}
public static void preorder (Node root) {
    if (root == null)
        return;
    System.out.print (root.data + " ");
    preorder (root.left);
    preorder (root.right);
}
```

QUES 1: Check if a Binary Tree is univalued or not

– We have a binary tree, the task is to check if the binary tree is univalued or not. If found to be true, then print "YES". Otherwise, print "NO".

Sample Input:



Sample Output: false

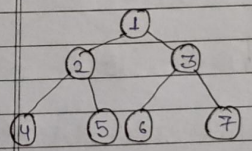
Sol.

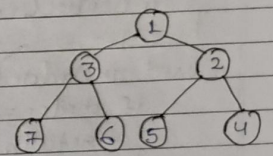
```
static class Node {          // TC → O(n)
    int data;                // SC → O(1)
    Node left;
    Node right;
}
static Node newNode (int data) {
    Node temp = new Node ();
    temp.data = data;
    temp.left = temp.right = null;
    return (temp);
}
static boolean isUnivalTree (Node root) {
    if (root == null)
        return true;
    if (root.left != null && root.data != root.left.data)
        return false;
    if (root.right != null && root.data != root.right.data)
        return false;
    return isUnivalTree (root.left) &&
           isUnivalTree (root.right);
}
```



```
public static void main (String[] args) {
    Node root = newNode (1);
    root.left = newNode (1);   root.right = newNode (1);
    root.left.left = newNode (1);
    root.left.right = newNode (1);
    root.right.right = newNode (1);
    if (isUnivalTree (root)) {
        System.out.print ("YES");
    } else {
        System.out.print ("NO");
    }
}
```

QUES 2. Invert Binary Tree – Mirror of Binary Tree T is another binary tree M(T) with left & right children of all non-leaf nodes interchanged.





Sol →

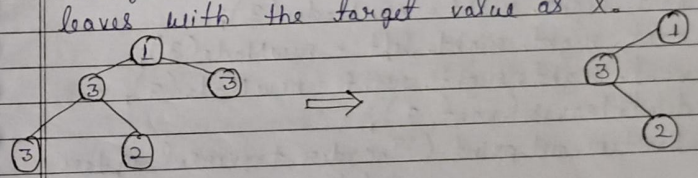
```
class Node {
    int data;
    Node left, right;
    public Node (int item) {
        data = item;
        left = right = null;
    }
}
```

```
class Solution {
  Node root;                          // TC → O(n)
  void mirror() {
     root = mirror (root);            // SC → O(n)
  }
  Node mirror (Node node) {
     if (node == null) {
        return node;
     }
     Node left = mirror (node.left);
     Node right = mirror (node.right);
     node.left = right;
     node.right = left;
     return node;
  }
  void inOrder() {
     inOrder (root);
  }
  void inOrder (Node node) {
     if (node == null) {
        return;
     }
     inOrder (node.left);
     System.out.print (node.data + " ");
     inOrder (node.right);
  }
  psvm {
     BinaryTree tree = new BinaryTree();
     tree.root = new Node (1);
     tree.root.left = new Node (2);
     tree.root.right = new Node (3);
```



```
     tree.root.left.left = new Node (4);
     tree.root.left.right = new Node (5);
     System.out.println ("Inorder Traversal : ");
     tree.inOrder();
     Syso();
     tree.mirror();
     Syso ("Inorder Traversal : ");
     tree.inOrder();
  }
}
```

QUES 3. Delete Leaf nodes with values as x - We have a binary tree & a target integer x, delete all the leaf nodes having value as x. Also, delete the newly formed leaves with the target value as x.



Sol →

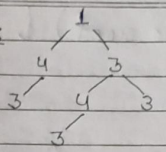
```
static class Node {
   int data;
   Node left, right;
}
static Node newNode (int data) {
   Node newNode = new Node();
   newNode.data = data;
   newNode.left = null;
   newNode.right = null;
   return (newNode);
}
```

```
static Node deleteLeaves (Node root, int x) {
  if (root == null) {
     return null;
  }
  inOrder (root.left);
  System.out.print (root.data + " ");
  inOrder (root.right);
}

public static void main (String[] args) {
  Node root = newNode (10);
  root.left = newNode (3);
  root.right = newNode (10);
  root.right = root.left.left = newNode (3);
  root.left.right = newNode (1);
  root.right.right = newNode (3);
  root.right.right.left = newNode(3);
  root.right.right.right = newNode (3);
  deleteLeaves (root, 3);
  System.out.print ("Inorder traversal after
                     deletion: ");
  inOrder (root);
}
```

QUES 4. Find all Duplicate Subtrees – We have a binary tree, find all duplicate subtrees. For each duplicate subtree, we only need to return the root node of any one of them. Two trees are duplicates if they have the same structure with the same node values.

Input:



Output: 4-3, 3

Sol →

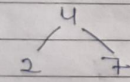
```
static HashMap <String, Integer> m;
static class Node {                    // Time Complexity - O(n*n)
  int data;                            // Space Complexity - O(n*n)
  Node left;
  Node right;
  Node (int data) {
     this.data = data;
     left = null;
     right = null;
  }
}
static String inorder (Node node) {
  if (node == null) {
     return "";
  }
  String str = "(";
  str += inorder (node.left);
  str += Integer.toString (node.data);
  str += inorder (node.right);
  str += ")";
  if (m.get(str) != null && m.get (str) == 1) {
     System.out.print (node.data + " ");
  if (m.containsKey (str))
     m.put (str, m.get (str) + 1);
  else
     m.put (str, 1);
  return str;
}
```

```
static void printAllDuplicates (Node root) {
    m = new HashMap<>();
    inOrder (root);
}

public static void main (String[] args) {
    Node root = null;
    root = new Node (1);
    root.left = new Node (2);
    root.right = new Node (3);
    root.left.left = new Node (4);
    root.right.left = new Node (2);
    root.right.left.left = new Node (4);
    root.right.right = new Node (4);
    printAllDuplicates (root);
}
```

QUES 5. Maximum Path Sum in a Binary Tree – We have a binary tree, find the maximum path sum. The path may start & end at any node in the tree.

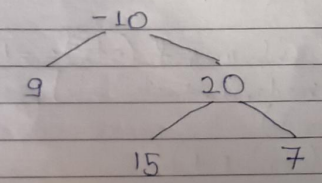
Input 1:



Output 1 : 13

4 + 7 + 2 = (13)

Input 2:



Output 2 : 42

20 + 15 + 7 + 9 - 10 = (41)

Sol →

```
class Node {
    int data;                    // Time Complexity - O(H)
    Node left, right;            // Space Complexity - O(1)
    public Node (int item) {
        data = item;
        left = right = null;
    }
}
class Res {
    public int val;
}
class Solution {
    Node root;
    int findMaxUtil (Node node, Res res) {
        if (node == null) {
            return 0;
        }
        int l = findMaxUtil (node.left, res);
        int r = findMaxUtil (node.right, res);
        int max_single = Math.max (Math.max (l, r) +
                                   node.data, node.data);
        int max_top = Math.max (max_single, l + r + node.data);
        res.val = Math.max (res.val, max_top);
        return max_single;
    }
    int findMaxSum (Node node) {
        Res res = new Res();
        res.val = Integer.MIN_VALUE;
        findMaxUtil (node, res);
        return res.val;
    }
```

```
int FindMaxSum() {
    return FindMaxSum(root);
}

public static void main(String args[]) {
    Solution tree = new Solution();
    tree.root = new Node(10);
    tree.root.left = new Node(2);
    tree.root.right = new Node(10);
    tree.root.left.left = new Node(20);
    tree.root.left.right = new Node(1);
    tree.root.right.right = new Node(-25);
    tree.root.right.right.left = new Node(3);
    tree.root.right.right.right = new Node(4);
    Syso("Maximum Path Sum : " + tree.FindMaxSum());
}
}
```