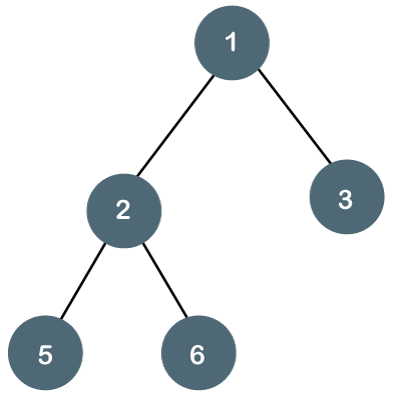
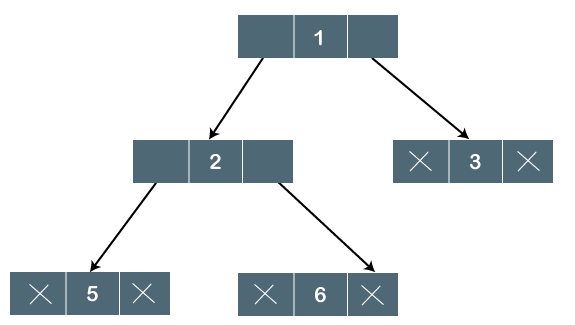
**Theory**

The Binary tree means that the node can have maximum two children. Here, binary name itself suggests that 'two'; therefore, each node can have either 0, 1 or 2 children.



The above tree is a binary tree because each node contains the utmost two children. The logical representation of the above tree is given below:



In the above tree, node 1 contains two pointers, i.e., left and a right pointer pointing to the left and right node respectively. The node 2 contains both the nodes (left and right node); therefore, it has two pointers (left and right). The nodes 3, 5 and 6 are the leaf nodes, so all these nodes contain ****NULL**** pointer on both left and right parts.

## Properties of Binary Tree

* At each level of i, the maximum number of nodes is 2i.
* The height of the tree is defined as the longest path from the root node to the leaf node. The tree which is shown above has a height equal to 3. Therefore, the maximum number of nodes at height 3 is equal to (1+2+4+8) = 15. In general, the maximum number of nodes possible at height h is (20 + 21 + 22+….2h) = 2h+1 -1.
* The minimum number of nodes possible at height h is equal to ****h+1****.
* If the number of nodes is minimum, then the height of the tree would be maximum. Conversely, if the number of nodes is maximum, then the height of the tree would be minimum.

If there are 'n' number of nodes in the binary tree.

****The minimum height can be computed as:****

As we know that,

n = 2h+1 -1

n+1 = 2h+1

Taking log on both the sides,

log2(n+1) = log2(2h+1)

log2(n+1) = h+1

****h = log2(n+1) - 1****

****The maximum height can be computed as:****

As we know that,

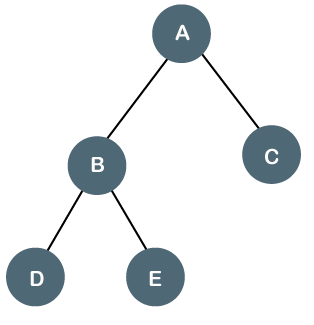
n = h+1

****h= n-1****

## Types of Binary Tree

**Full/ proper/ strict Binary tree**

The full binary tree can also be defined as the tree in which each node must contain 2 children except the leaf nodes.



****Properties of Full Binary Tree****

* The number of leaf nodes is equal to the number of internal nodes plus 1. In the above example, the number of internal nodes is 2; therefore, the number of leaf nodes is equal to 3.
* The maximum number of nodes is the same as the number of nodes in the binary tree, i.e., 2h+1 -1.
* The minimum number of nodes in the full binary tree is 2\*h-1.
* The minimum height of the full binary tree is ****log2(n+1) - 1.****
* The maximum height of the full binary tree can be computed as:

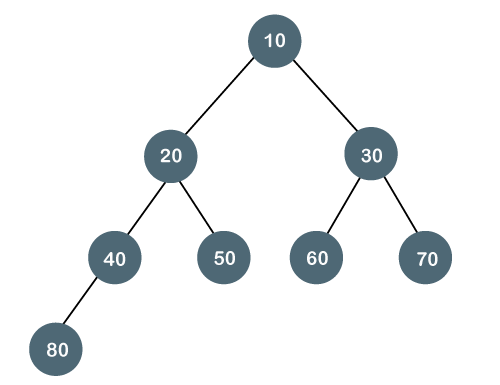
n= 2\*h - 1

n+1 = 2\*h

****h = n+1/2****

****Complete Binary Tree****

**T**he complete binary tree is a tree in which all the nodes are completely filled except the last level. In the last level, all the nodes must be as left as possible. In a complete binary tree, the nodes should be added from the left.

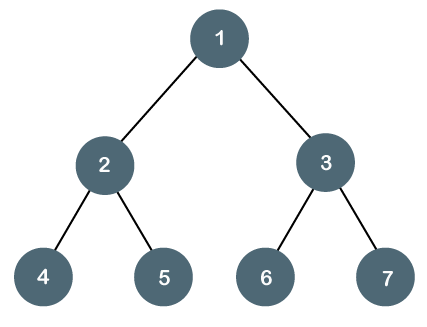


****Properties of Complete Binary Tree****

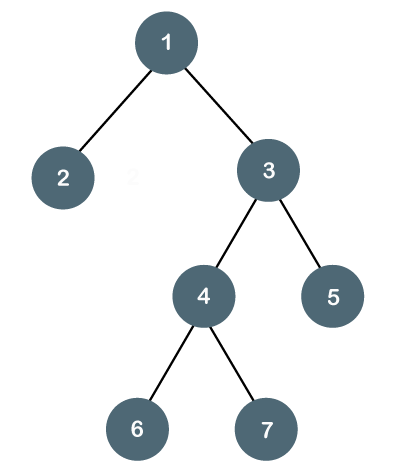
* The maximum number of nodes in complete binary tree is 2h+1 - 1.
* The minimum number of nodes in complete binary tree is 2h.
* The minimum height of a complete binary tree is ****log2(n+1) - 1.****
* The maximum height of a complete binary tree is

****Perfect Binary Tree****

A tree is a perfect binary tree if all the internal nodes have 2 children, and all the leaf nodes are at the same level.



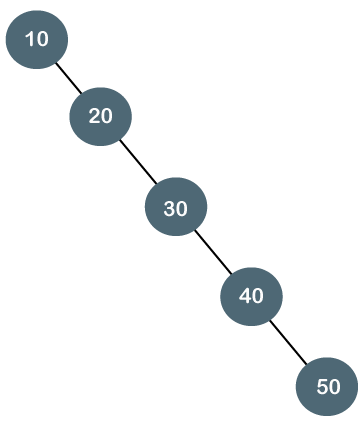
The below tree is not a perfect binary tree because all the leaf nodes are not at the same level.



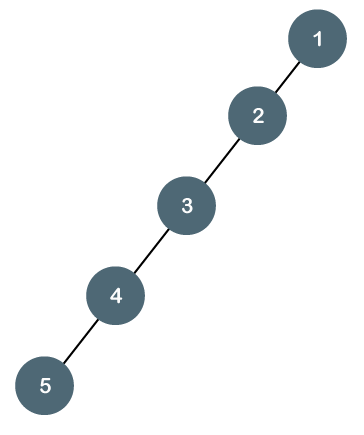
Note: All the perfect binary trees are the complete binary trees as well as the full binary tree, but vice versa is not true, i.e., all complete binary trees and full binary trees are the perfect binary trees.

### Degenerate Binary Tree

The degenerate binary tree is a tree in which all the internal nodes have only one children.



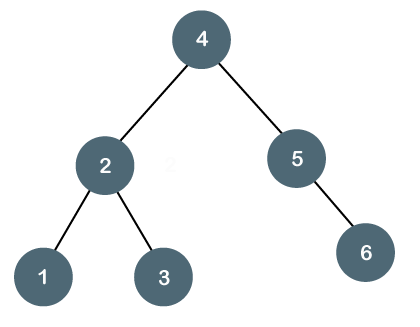
The above tree is a degenerate binary tree because all the nodes have only one child. It is also known as a right-skewed tree as all the nodes have a right child only.



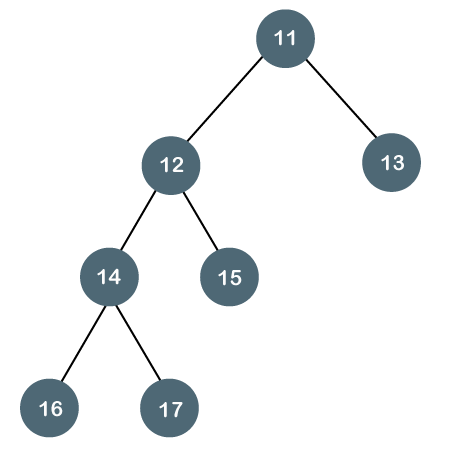
The above tree is also a degenerate binary tree because all the nodes have only one child. It is also known as a left-skewed tree as all the nodes have a left child only.

****Balanced Binary Tree****

The balanced binary tree is a tree in which both the left and right trees differ by atmost 1. For example, **AVL** and **Red-Black trees** are balanced binary tree.



The above tree is a balanced binary tree because the difference between the left subtree and right subtree is zero.



The above tree is not a balanced binary tree because the difference between the left subtree and the right subtree is greater than 1.

**Building Tree**

**Recursively**

// Method to create a binary tree recursively using user input from Scanner

Node createTree(Scanner sc) {

    System.out.println("Enter data: "); // Prompt the user to enter data for the current node

    int data = sc.nextInt(); // Read the input data

    // base case: if input data is -1, return null indicating end of branch

    if(data==-1) {

        return null;

    }

    // Create a new node with the input data

    Node n = new Node(data);

    // Recursively create left subtree

    n.left = createTree(sc);

    // Recursively create right subtree

    n.right = createTree(sc);

    return n; // Return the constructed node

}

**From level order**

// This function builds a binary tree from a given level order traversal represented by an integer array.

// The array should contain -1 for missing nodes in the tree.

// It returns the root of the constructed binary tree.

Node buildFromLevelOrder(int[] lvlOrder) {

    // Check if the input array is null or empty

    if(lvlOrder==null || lvlOrder.length==0) {

        return null; // Return null if the array is empty

    }

    // Create a queue to store nodes during construction

    Queue<Node> q = new LinkedList<>();

    // Create the root node from the first element of the level order traversal array

    Node root = new Node(lvlOrder[0]);

    q.add(root); // Add the root node to the queue

    // Variable to keep track of the index in the level order array

    int i=1;

    // Continue until the queue is empty or all elements of the level order array are processed

    while(!q.isEmpty() && i<lvlOrder.length) {

        // Remove the front node from the queue

        Node current = q.poll();

        // Add left child if it exists (and is not -1)

        if(i<lvlOrder.length && lvlOrder[i]!=-1) {

            current.left = new Node(lvlOrder[i]); // Create the left child node

            q.add(current.left); // Add the left child to the queue

        }

        i++; // Move to the next element in the level order array

        // Add right child if it exists (and is not -1)

        if(i<lvlOrder.length && lvlOrder[i]!=-1) {

            current.right = new Node(lvlOrder[i]); // Create the right child node

            q.add(current.right); // Add the right child to the queue

        }

        i++; // Move to the next element in the level order array

    }

    // Return the root of the constructed binary tree

    return root;

}

**Traversal**

**Level Order**

void levelOrder(Node root) {

        Queue<Node> q = new LinkedList<>();

        q.add(root); // Add the root node to the queue

        while(!q.isEmpty()) { // Continue the loop until the queue is empty

            Node temp = q.poll(); // Remove and retrieve the front node from the queue

            System.out.print(temp.data); // Print the data of the current node

            if(temp.left!=null && temp.right!=null) { // Check if the current node has both left and right children

                q.add(temp.left); // Add the left child to the queue if it exists

                q.add(temp.right); // Add the right child to the queue if it exists

            }

        }

    }

**Zig-zag traversal**

public List<List<Integer>> zigzagLevelOrder(TreeNode root) {

    // Initialize the result list

    List<List<Integer>> ans = new ArrayList<>();

    // If the root is null, return an empty result

    if(root == null){

        return ans;

    }

    // Initialize a deque for traversal

    Deque<TreeNode> dq = new LinkedList<>();

    // Initialize a boolean flag to indicate traversal direction

    boolean leftToRight = true;

    // Add the root to the deque

    dq.add(root);

    // Traverse the tree

    while(!dq.isEmpty()){

        // Get the number of nodes at the current level

        int size = dq.size();

        // Initialize a list to store values of nodes at the current level

        List<Integer> temp = new ArrayList<>();

        // Iterate through the nodes at the current level

        for(int i = 0; i < size; i++){

            TreeNode t;

            // Poll the node from the front or back of the deque based on traversal direction

            if(leftToRight){

                t=dq.pollFirst();

            }

            else{

                t=dq.pollLast();

            }

            // Add the value of the node to the list

            temp.add(t.val);

            // Add the children of the current node to the deque based on traversal direction

            if(leftToRight){

                if(t.left!=null){

                    dq.addLast(t.left);

                }

                if(t.right!=null){

                    dq.addLast(t.right);

                }

            }

            else{

                if(t.right!=null){

                    dq.addFirst(t.right);

                }

                if(t.left!=null){

                    dq.addFirst(t.left);

                }

            }

        }

        // Add the list of values at the current level to the result

        ans.add(temp);

        // Toggle the traversal direction

        leftToRight = !leftToRight;

    }

    // Return the result

    return ans;

}

**TC :** O( N )

**SC :** O( N+H ), N= no of nodes and H=height

**Pre Order**

// Perform a pre-order traversal of the binary tree starting from the given root node.

// Print the data of each node in the order: current node, left subtree, right subtree.

void preOrder(Node root) {

    // Base case: if the root is null, return as there's nothing to process

    if(root==null) {

        return;

    }

    // Print the data of the current node

    System.out.print(root.data+" ");

    // Recursively traverse the left subtree

    preOrder(root.left);

    // Recursively traverse the right subtree

    preOrder(root.right);

}

**In Order**

// Perform an in-order traversal of the binary tree starting from the given root node.

void inOrder(Node root) {

    // Base case: if the root is null, return without further processing

    if(root==null) {

        return;

    }

    // Recursively traverse the left subtree

    inOrder(root.left);

    // Print the data of the current node

    System.out.print(root.data+" ");

    // Recursively traverse the right subtree

    inOrder(root.right);

}

**Post Order**

// Function to traverse a binary tree in post-order fashion

void postOrder(Node root) {

    // Base case: if the root is null, return

    if(root==null) {

        return;

    }

    // Recursively traverse the left subtree in post-order

    postOrder(root.left);

    // Recursively traverse the right subtree in post-order

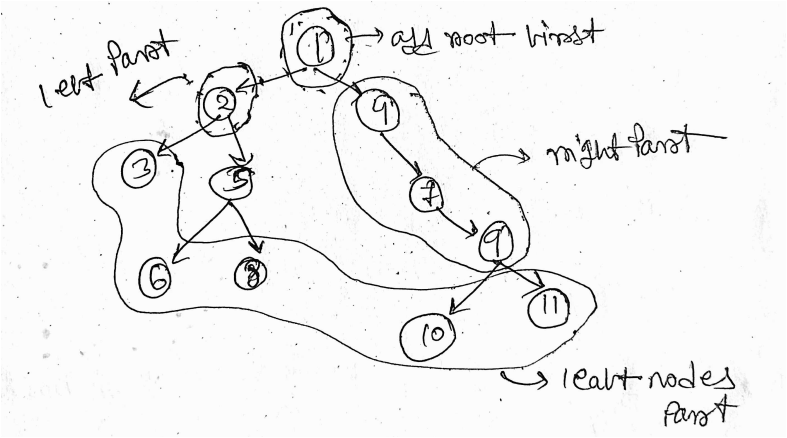
    postOrder(root.right);

    // Print the data of the current node

    System.out.print(root.data+" ");

}

**Boundary Traversal**



private void traverseLeft(Node node, ArrayList<Integer> ans){

    // Base case: if the current node is null or it's a leaf node, return

    if(node == null || (node.left == null && node.right == null)){

        return;

    }

    // Add the current node's value to the list

    ans.add(node.data);

    // Traverse left if exists, otherwise traverse right

    if(node.left != null){

        traverseLeft(node.left, ans);

    }

    else{

        traverseLeft(node.right, ans);

    }

}

private void traverseLeaf(Node node, ArrayList<Integer> ans){

    // Base case: if the current node is null, return

    if(node == null){

        return;

    }

    // If the current node is a leaf, add its value to the list

    if(node.left == null && node.right == null){

        ans.add(node.data);

    }

    // Recursively traverse left and right subtrees

    traverseLeaf(node.left, ans);

    traverseLeaf(node.right, ans);

}

private void traverseRight(Node node, ArrayList<Integer> ans){

    // Base case: if the current node is null or it's a leaf node, return

    if(node == null || (node.left == null && node.right == null)){

        return;

    }

    // Traverse right if exists, otherwise traverse left

    if(node.right != null){

        traverseRight(node.right, ans);

    }

    else{

        traverseRight(node.left, ans);

    }

    // Add the current node's value to the list after traversing

    ans.add(node.data);

}

ArrayList<Integer> boundary(Node node)

{

    ArrayList<Integer> ans = new ArrayList<>();

    // If the root is null, return empty list

    if(node == null){

        return ans;

    }

    // Add the root node's value to the list

    ans.add(node.data);

    // Traverse left boundary

    traverseLeft(node.left, ans);

    // Traverse leaf nodes in left and right subtrees

    traverseLeaf(node.left, ans);

    traverseLeaf(node.right, ans);

    // Traverse right boundary

    traverseRight(node.right, ans);

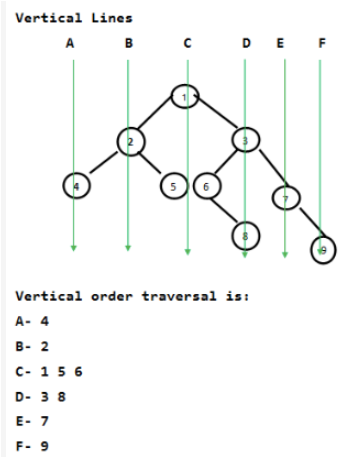
    return ans;

}

**TC :** O( N )

**SC :** O( N )

**Vertical traversal**



class MyPair {

    Node node;

    int hd;

    MyPair(Node node, int hd) {

        this.node = node;

        this.hd = hd;

    }

}

class Solution {

    // Function to find the vertical order traversal of Binary Tree.

    static ArrayList<Integer> verticalOrder(Node root) {

        ArrayList<Integer> ans = new ArrayList<Integer>(); // Initialize list to store the result

        Queue<MyPair> q = new LinkedList<MyPair>(); // Initialize a queue to perform level order traversal

        Map<Integer, ArrayList<Integer>> mp = new TreeMap<>(); // Initialize a map to store nodes at each horizontal distance

        if (root == null) {

            return ans; // Return empty list if tree is empty

        }

        q.add(new MyPair(root, 0)); // Add root with horizontal distance 0 to the queue

        // Perform level order traversal

        while (!q.isEmpty()) {

            MyPair temp = q.poll(); // Dequeue a node

            Node temp\_node = temp.node; // Extract node from MyPair

            int temp\_hd = temp.hd; // Extract horizontal distance from MyPair

            // Check if the horizontal distance exists in the map

            if (mp.containsKey(temp\_hd)) {

                // If exists, get the list of nodes at that horizontal distance

                ArrayList<Integer> al = mp.get(temp\_hd);

                al.add(temp\_node.data); // Add current node's data to the list

                mp.put(temp\_hd, al); // Update the map

            } else {

                // If horizontal distance doesn't exist, create a new list

                ArrayList<Integer> al = new ArrayList<>();

                al.add(temp\_node.data); // Add current node's data to the list

                mp.put(temp\_hd, al); // Put the list in the map

            }

            // Enqueue left child if exists with adjusted horizontal distance

            if (temp\_node.left != null) {

                q.add(new MyPair(temp\_node.left, temp\_hd - 1));

            }

            // Enqueue right child if exists with adjusted horizontal distance

            if (temp\_node.right != null) {

                q.add(new MyPair(temp\_node.right, temp\_hd + 1));

            }

        }

        // Traverse the map to collect nodes in vertical order

        for (Map.Entry<Integer, ArrayList<Integer>> m : mp.entrySet()) {

            ArrayList<Integer> al = m.getValue(); // Get list of nodes at current horizontal distance

            for (int i = 0; i < al.size(); i++) {

                ans.add(al.get(i)); // Add each node's data to the result list

            }

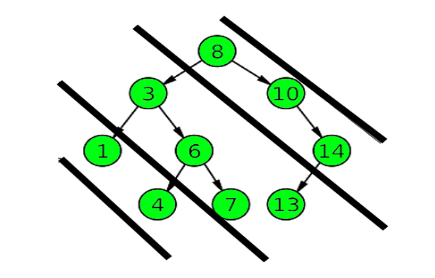
        }

        return ans; // Return the result list

    }

}

**Diagonal traversal**



If a node has a left child then the left child and the root will be in a different diagonal, but the right child of the node will be in the same diagonal as the root.

public ArrayList<Integer> diagonal(Node root)

{

    // Initialize an ArrayList to store the diagonal elements

    ArrayList<Integer> ans = new ArrayList<>();

    // Check if the root is null, if so, return an empty list

    if(root == null){

        return ans;

    }

    // Initialize a queue to store the left children of the nodes

    Queue<Node> leftQueue = new LinkedList<>();

    // Initialize a current node pointer and set it to the root

    Node cur = root;

    // Loop until the current node is not null

    while(cur != null){

        // Add the data of the current node to the result list (diagonal element)

        ans.add(cur.data);

        // If the left child of the current node exists, add it to the leftQueue

        if(cur.left != null){

            leftQueue.add(cur.left);

        }

        // If the right child of the current node exists, move to the right child

        if(cur.right != null){

            cur = cur.right;

        }

        // If the right child does not exist

        else{

            // If there are elements in the leftQueue, update the current node to the front element of the leftQueue and remove it from the queue

            if(!leftQueue.isEmpty()){

                cur = leftQueue.peek();

                leftQueue.remove();

            }

            // If the leftQueue is empty, set the current node to null to exit the loop

            else{

                cur = null;

            }

        }

    }

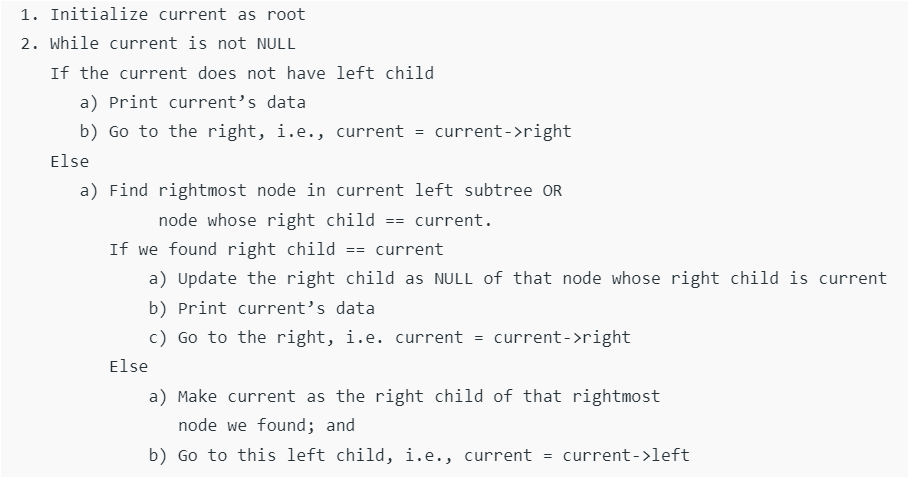
    // Return the diagonal elements

    return ans;

}

**Morris traversal for Inorder**

Morris Traversal is a space-efficient algorithm for traversing binary trees in Inorder without using [Recursion](https://www.codingninjas.com/studio/library/what-is-recursion" \t "https://www.codingninjas.com/studio/library/_blank) or a stack. It modifies the tree structure by adding temporary links between nodes to traverse the tree in a specific order. It is not commonly used in practice due to its higher time complexity, but it can be useful in certain memory-constrained environments.



**Time Complexity :** O(n), where n is the number of nodes in the binary tree.

This is because each node is visited at most twice: once to establish the threading and once to visit the node during the traversal.

There are no nested loops or recursion, and each node is visited exactly once in a constant amount of time.

**Space Complexity :** O(1), meaning it uses constant extra space.

void MorrisTraversal(tNode root)

{

    tNode current, pre;

    if (root == null)

        return;

    current = root;

    while (current != null)

    {

        if (current.left == null)

        {

            System.out.print(current.data + " ");

            current = current.right;

        }

        else {

            /\* Find the inorder

               predecessor of current

            \*/

            pre = current.left;

            while (pre.right != null

                   && pre.right != current)

                pre = pre.right;

            /\* Make current as right

               child of its

             \* inorder predecessor \*/

            if (pre.right == null) {

                pre.right = current;

                current = current.left;

            }

            /\* Revert the changes made

               in the 'if' part

               to restore the original

               tree i.e., fix

               the right child of predecessor\*/

            else

            {

                pre.right = null;

                System.out.print(current.data + " ");

                current = current.right;

            }

        }

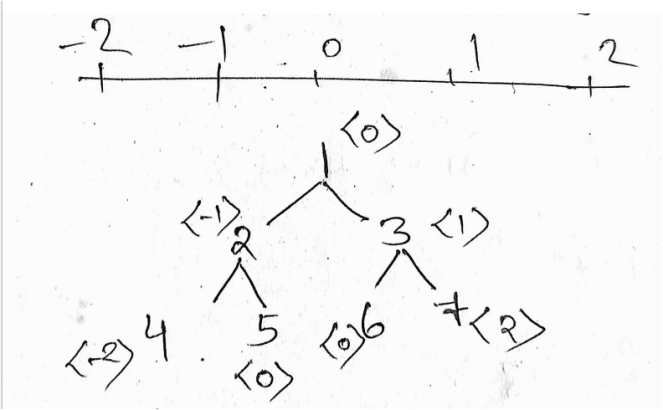
    }

}

**Different view**

**Top view**

Same as vertical order traversal, but adding only a single node for every horizontal distance in the map.



It can be considered as, for every horizontal distance, first entry hide all the rest entry. For instance, for <0> 1 hide later entry 5 and 6.

**Bottom view**

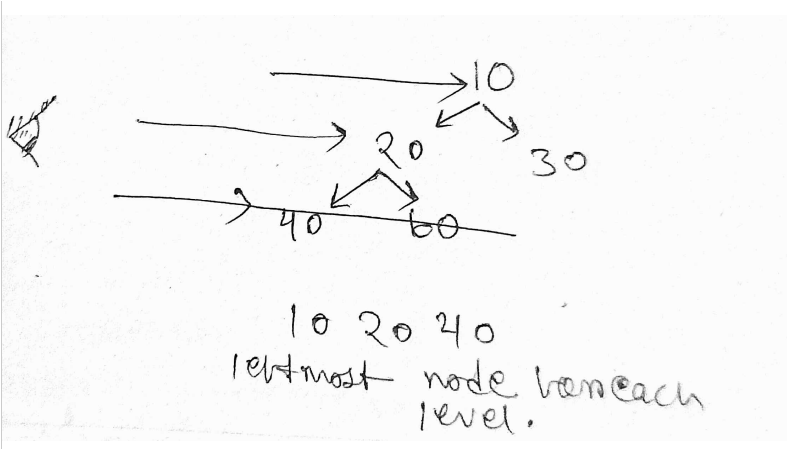
Same as vertical order traversal, but just adding in map without any condition checking.

Because map.put() will store the latest value, all the previous value will get replaced.so, we will get only bottom nodes.

**Left view**

Keep the track of current leve.

If the currrent level== ans.size() add new node.By doing this we are addding this only leftmost node for each level.



// Recursive function to traverse the binary tree and populate the left view elements.

void solve(Node root, ArrayList<Integer> ans, int lvl) {

    // Base case: if the current node is null, return.

    if (root == null) {

        return;

    }

    // If the current level is equal to the size of the answer list,

    // it means we've reached a new level in the tree.

    if (lvl == ans.size()) {

        // Add the data of the current node to the answer list,

        // as it represents the leftmost node at this level.

        ans.add(root.data);

    }

    // Recursively traverse the left and right subtrees, incrementing the level by 1.

    solve(root.left, ans, lvl + 1);

    solve(root.right, ans, lvl + 1);

}

// Function to return a list containing elements of the left view of the binary tree.

ArrayList<Integer> leftView(Node root) {

    // Initialize an empty ArrayList to store the left view elements.

    ArrayList<Integer> ans = new ArrayList<Integer>();

    // Call the solve function to populate the left view elements.

    // Start with the root node and level 0.

    solve(root, ans, 0);

    // Return the list containing the left view elements.

    return ans;

}

**Right view**

Same as left view. Just recursive call to right subtree would be before left subtree.

**Count Leaf Nodes**

public static void inOrder(BinaryTreeNode<Integer> root, AtomicInteger count){

    if(root==null){

      return;

    }

    inOrder(root.left, count);

    if(root.left==null && root.right==null){

        // automatically increment and return new value

        // in place modification

      count.incrementAndGet();

    }

    inOrder(root.right,count);

}

public static int noOfLeafNodes(BinaryTreeNode<Integer> root) {

    // using AtomicInteger to change value in place

    AtomicInteger count= new AtomicInteger(0);

    inOrder(root, count);

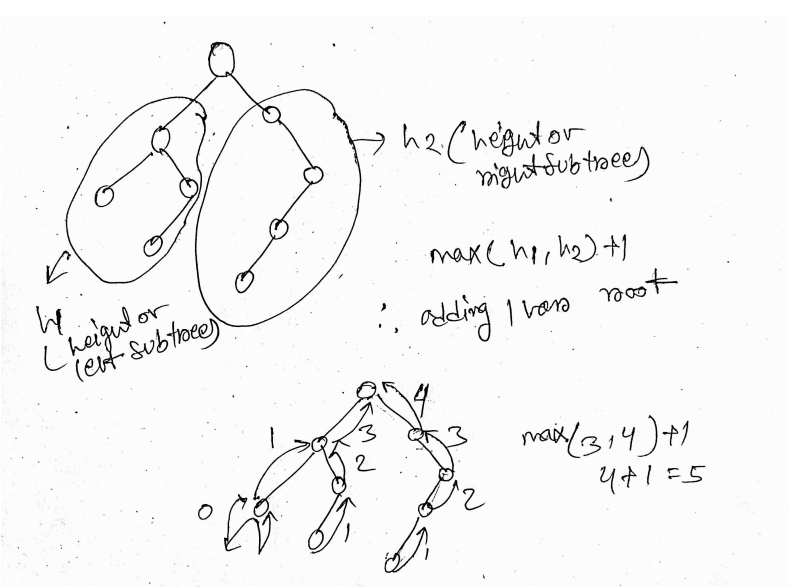
    // getting value of count object

    return count.get();

}

Time complexity: O( N ) Space complexity: O( N )

**179\_Height of a tree**



Time complexity: O( N ) Space complexity: O( height )

If the tree is left or rigth skewed tree then SC would be O( N )

public static int heightOfBinaryTree(TreeNode root) {

        if(root==null){

            return 0;

        }

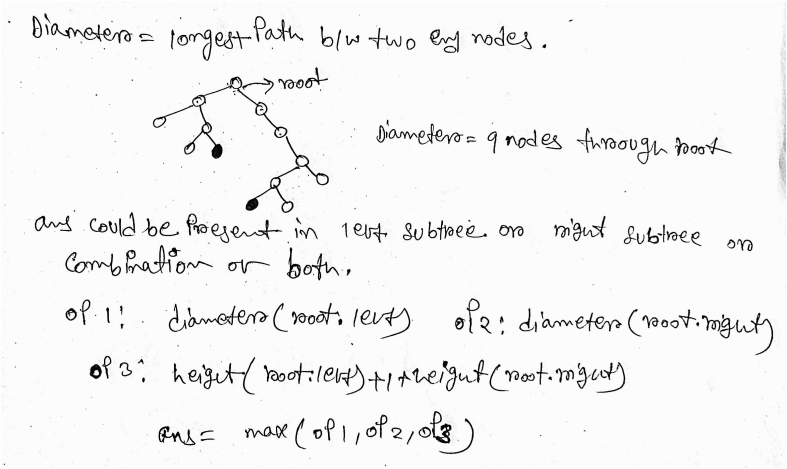
        int height=Math.max(heightOfBinaryTree(root.left),

        heightOfBinaryTree(root.right)) +1;

        return height;

    }

**180\_Diameter of a tree**



public int height(Node root) {

  if(root==null){

      return 0;

  }

  int height=Math.max(height(root.left),height(root.right)) +1;

  return height;

}

// Function to return the diameter of a Binary Tree.

int diameter(Node root) {

  if(root==null){

      return 0;

  }

  int d1=diameter(root.left);

  int d2=diameter(root.right);

  int d3=height(root.left)+1+height(root.right);

  return Math.max(d1, Math.max(d2,d3));

}

**TC :** O(N^2),O(N) time will be needed to traverse the tree and for each node of the tree another O(N) will be needed to find the height of its left and right subtree.

**SC :** O(N)

To do this in O(N) time, we can calculate diameter and height together during recursivecall.

1. DiaPair Class: This class is a simple data structure that holds two properties: height and diameter. It is used to return both the height and diameter of a subtree from the diaFast method.
2. diaFast Method:
   1. If the root is null, it returns a DiaPair object with both height and diameter set to 0.
   2. It recursively calls diaFast for the left and right subtrees to compute their heights and diameters.
   3. It calculates three options for the diameter:
      1. op1: Diameter of the left subtree.
      2. op2: Diameter of the right subtree.
      3. op3: Diameter passing through the current root node, calculated as the sum of heights of left and right subtrees plus 1.
   4. It calculates the temporary height (temp\_h) of the current subtree as 1 plus the maximum of heights of the left and right subtrees.
   5. It calculates the temporary diameter (temp\_d) of the current subtree as the maximum of op1, op2, and op3.
   6. It creates a new DiaPair object ans with temp\_h and temp\_d as its properties and returns it

**TC :** O(N) **SC :** O(H)

class DiaPair{

  int height;

  int diameter;

  DiaPair(){

  }

  DiaPair(int height, int diameter){

      this.height=height;

      this.diameter=diameter;

  }

}

class Solution {

  public static DiaPair diaFast(Node root){

      // Base case:

      if(root==null){

          DiaPair dp = new DiaPair(0,0);

          return dp;

      }

      // Recursively compute DiaPair for the left subtree

      DiaPair left = diaFast(root.left);

      // Recursively compute DiaPair for the right subtree

      DiaPair right = diaFast(root.right);

      // Calculate three options for diameter:

      int op1 = left.diameter; // Diameter of the left subtree

      int op2 = right.diameter; // Diameter of the right subtree

      int op3 = left.height + right.height + 1; // Diameter passing through the current root

      // Calculate temporary height of the current subtree

      int temp\_h = 1 + Math.max(left.height, right.height);

      // Calculate temporary diameter of the current subtree

      int dtemp\_d = Math.max(op1, Math.max(op2, op3));

      DiaPair ans = new DiaPair(temp\_h, dtemp\_d);

      return ans;

  }

  int diameter(Node root) {

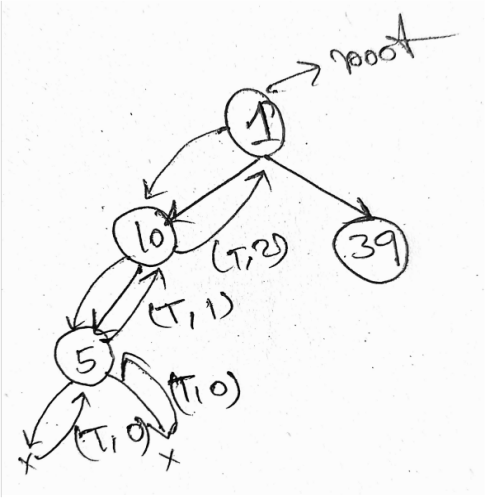
      return diaFast(root).diameter;

  }

}

**190\_Check if a tree is balanced or not**

A tree is height balanced if difference between heights of left and right subtrees is not more than one for all nodes of tree.



class BalPair{

  boolean balance; // Indicates whether the tree is balanced or not

  int height; // Height of the tree

  // Default constructor

  BalPair(){

  }

  // Parameterized constructor

  BalPair(boolean balance, int height){

      this.balance=balance;

      this.height=height;

  }

}

class Tree

{

  // Recursive function to determine balance and height of a subtree

  private BalPair balFast(Node root){

      // Base case

      if(root==null){

          BalPair bp = new BalPair(true,0);

          return bp;

      }

      // Recursively get balance and height of left and right subtrees

      BalPair left = balFast(root.left);

      BalPair right = balFast(root.right);

      // Extract balance status and heights of left and right subtrees

      boolean leftAns = left.balance;

      boolean rightAns = right.balance;

      // Calculate height difference between left and right subtrees

      int diff = Math.abs(left.height-right.height);

      // Create a new BalPair object to store balance and height information

      BalPair ans = new BalPair();

      // Set height of current node by taking maximum of left and right subtree heights, plus 1

      ans.height = Math.max(left.height,right.height)+1;

      // Check if both subtrees are balanced and their height difference is at most 1

      if(leftAns && rightAns && diff<=1){

          ans.balance = true; // Set balance to true if conditions are met

      }

      else{

          ans.balance = false; // Otherwise, set balance to false

      }

      return ans;

  }

  boolean isBalanced(Node root)

  {

      return balFast(root).balance;

  }

}

**TC :** O(N) **SC :** O(N)

**Check if Identical Tree**

public boolean isSameTree(TreeNode p, TreeNode q) {

    // Base case: both trees are empty, so they are identical

    if(p==null && q==null){

        return true;

    }

    // If one tree is empty and the other is not, they are not identical

    if(p!=null && q==null){

        return false;

    }

    if(p==null && q!=null){

        return false;

    }

    // Recursively check left and right subtrees

    boolean left = isSameTree(p.left,q.left);

    boolean right = isSameTree(p.right,q.right);

    // Check if current nodes' values are equal

    boolean data = p.val==q.val?true:false;

    // If left subtree is identical, right subtree is identical,

    // and current nodes' values are equal, return true

    if(left && right && data){

        return true;

    }

    // Otherwise, the trees are not identical

    else{

        return false;

    }

}

**Check if Sum tree**

Given a Binary Tree. Return true if, for every node X in the tree other than the leaves, its value is equal to the sum of its left subtree's value and its right subtree's value. Else return false.

class Pair{

    boolean isTrue; // Boolean flag indicating if the tree is a sum tree

    int sum; // Sum of the subtree rooted at the current node

    // Default constructor

    Pair(){

    }

    // Parameterized constructor to initialize the Pair with given values

    Pair(boolean isTrue, int sum){

        this.isTrue = isTrue;

        this.sum = sum;

    }

}

// Define a class named Solution which contains a method to check if a given binary tree is a Sum Tree

class Solution

{

    // Private method to recursively compute the sum and check if the tree is a Sum Tree

    private Pair sumFast(Node root){

        // Base case: If the root is null, return a Pair indicating true and sum as 0

        if(root == null){

            Pair p = new Pair(true, 0);

            return p;

        }

        // Base case: If the root has no children, return a Pair indicating true and sum as root's data

        if(root.left == null && root.right == null){

            Pair p = new Pair(true, root.data);

            return p;

        }

        // Recursively compute sum and check if left subtree and right subtree are sum trees

        Pair leftAns = sumFast(root.left);

        Pair rightAns = sumFast(root.right);

        // Create a new Pair to store the result for the current node

        Pair ans = new Pair();

        // Calculate the sum for the current node by adding sums of left and right subtrees along with the current node's data

        ans.sum = leftAns.sum + rightAns.sum + root.data;

        // Check if the current subtree rooted at the current node is a sum tree

        boolean data = leftAns.sum + rightAns.sum == root.data ? true : false;

        if(leftAns.isTrue && rightAns.isTrue && data){

            ans.isTrue = true; // If both left and right subtrees are sum trees and satisfy the sum condition, set isTrue to true

        }

        else{

            ans.isTrue = false; // Otherwise, set isTrue to false

        }

        return ans; // Return the result for the current subtree

    }

    // Public method to check if the given binary tree is a sum tree

    boolean isSumTree(Node root)

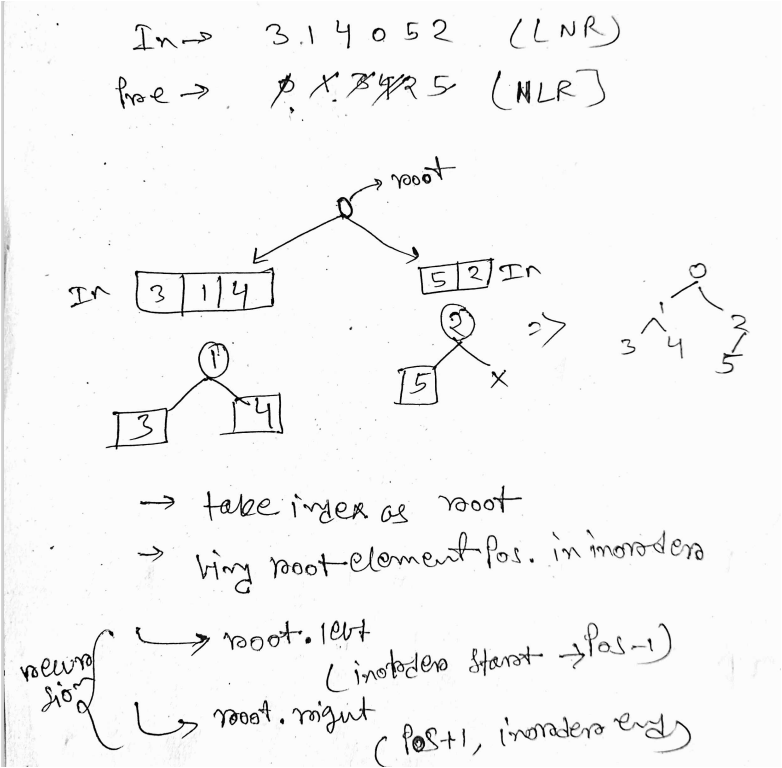
    {

        return sumFast(root).isTrue; // Call the private method sumFast and return the boolean flag indicating if the tree is a sum tree

    }

}

**196\_Construct Binary tree from Inorder and preorder traversal**



// Helper method to create a mapping of values to their indices in the inorder array

    private static void createMapping(HashMap<Integer,Integer> nodeToMap, int inorder[], int n){

        for(int i=0; i<n; i++){

            nodeToMap.put(inorder[i], i);

        }

    }

    // Recursive helper method to construct the binary tree

    private static TreeNode solve(int inorder[], int preorder[], int[] preIndex, int inStart, int inEnd, HashMap<Integer,Integer> nodeToMap, int n){

        // Base cases: if preIndex exceeds the length of preorder or inorder start is greater than end, return null

        if(preIndex[0]>=n || inStart>inEnd){

            return null;

        }

        // Create a new TreeNode with the current element from preorder

        int element = preorder[preIndex[0]];

        TreeNode root = new TreeNode(element);

        // Find the position of the current element in the inorder array using the mapping

        int position = nodeToMap.get(element);

        preIndex[0]++; // Move to the next element in preorder

        // Recursively build left and right subtrees

        root.left = solve(inorder, preorder, preIndex, inStart, position-1, nodeToMap, n);

        root.right = solve(inorder, preorder, preIndex, position+1, inEnd, nodeToMap, n);

        return root;

    }

    // Main method to build the binary tree using preorder and inorder arrays

    public TreeNode buildTree(int[] preorder, int[] inorder) {

        // Create a hashmap to store the mapping of values to their indices in inorder array

        HashMap<Integer,Integer> nodeToMap = new HashMap<>();

        int n = preorder.length; // Length of preorder array

        createMapping(nodeToMap, inorder, n); // Create the mapping

        int[] preIndex = new int[1]; // Array to hold the preorder index (mutable)

        // Call the solve method to construct the binary tree

        TreeNode root = solve(inorder, preorder, preIndex, 0, n-1, nodeToMap, n);

        return root; // Return the root of the constructed binary tree

    }

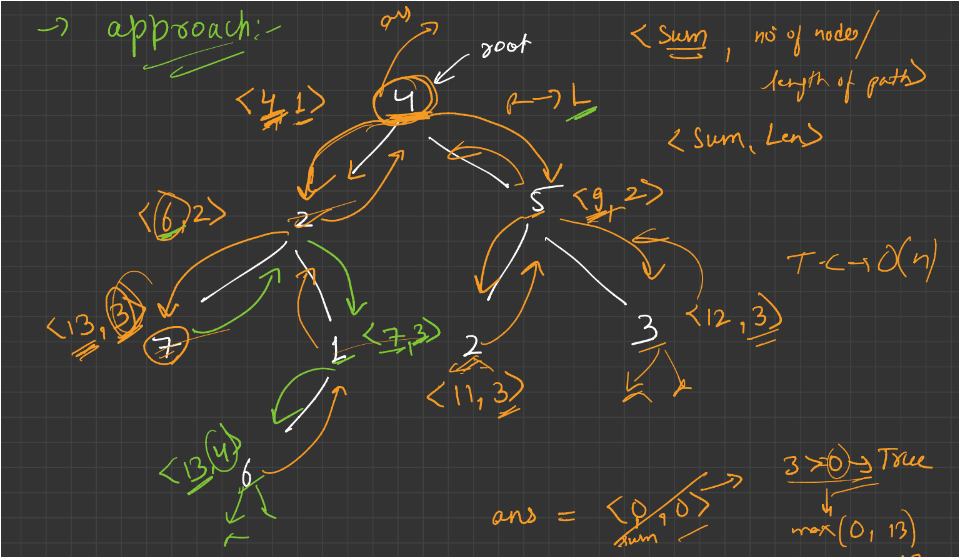
**Construct Binary tree from Inorder and postorder traversal**

Same as inorder-preorder just,

Postindex = n-1

Right recursive call should be before left call

**202\_Sum of Nodes on the Longest path from root to leaf node**



private void solve(Node root, int len, int sum, int[] maxLen, int[] maxSum) {

    // Base case: if the current node is null

    if (root == null) {

        // If the length of the current path is greater than the maximum length found so far

        if (len > maxLen[0]) {

            // Update the maximum length and corresponding maximum sum

            maxLen[0] = len;

            maxSum[0] = sum;

        }

        // If the length of the current path is equal to the maximum length found so far

        else if (len == maxLen[0]) {

            // Update the maximum sum if the current sum is greater

            maxSum[0] = Math.max(sum, maxSum[0]);

        }

        // Exit the function

        return;

    }

    // Recursively traverse the left subtree

    solve(root.left, len + 1, sum + root.data, maxLen, maxSum);

    // Recursively traverse the right subtree

    solve(root.right, len + 1, sum + root.data, maxLen, maxSum);

}

// Function to find the sum of the long root-to-leaf path in a binary tree

public int sumOfLongRootToLeafPath(Node root) {

    // Arrays to store the maximum length and maximum sum found so far

    int[] maxLen = new int[1];

    int[] maxSum = new int[1];

    // Call the recursive function to solve the problem

    solve(root, 0, 0, maxLen, maxSum);

    // Return the maximum sum found

    return maxSum[0];

}

**TC : O( N )**

**SC : O( H )**

**207\_LCA in a Binary tree**

// Function to find the lowest common ancestor (LCA) of two nodes in a binary tree.

Node lca(Node root, int n1, int n2) {

    // If the current node is null, return null indicating that no ancestor is found.

    if (root == null) {

        return null;

    }

    // If the current node's data matches either n1 or n2, return the current node as an ancestor.

    if (root.data == n1 || root.data == n2) {

        return root;

    }

    // Recursively find the LCA in the left subtree.

    Node leftAns = lca(root.left, n1, n2);

    // Recursively find the LCA in the right subtree.

    Node rightAns = lca(root.right, n1, n2);

    // If both left and right subtrees return a non-null value, then the current node is the LCA.

    if (leftAns != null && rightAns != null) {

        return root;

    }

    // If only the left subtree returns a non-null value, return its result as the LCA.

    else if (leftAns != null && rightAns == null) {

        return leftAns;

    }

    // If only the right subtree returns a non-null value, return its result as the LCA.

    else if (leftAns == null && rightAns != null) {

        return rightAns;

    }

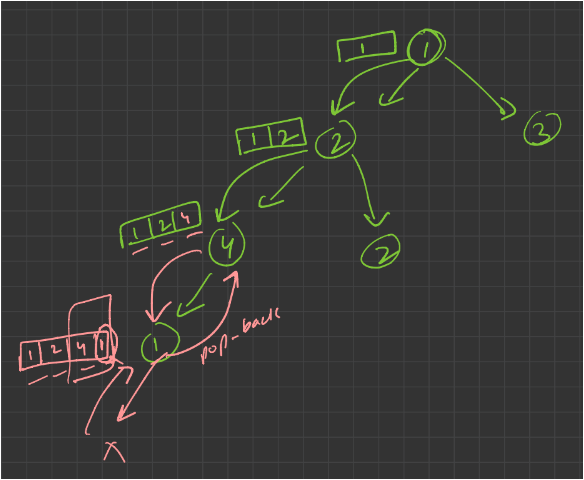
    // If both left and right subtrees return null, then the current node is not the LCA.

    else {

        return null;

    }}

**206\_count all "K" Sum paths in a Binary tree**



This basic approach effectively explores all possible paths in the binary tree and counts those whose sum matches the target k. However, it may TLE.

**TC : O( N^2 )**

private void solve(Node root, int k, ArrayList<Integer> path, int[] count) {

        // Base case: If the current node is null, return.

        if (root == null) {

            return;

        }

        // Add the current node's data to the path.

        path.add(root.data);

        // Recursively explore the left subtree.

        solve(root.left, k, path, count);

        // Recursively explore the right subtree.

        solve(root.right, k, path, count);

        // Calculate the sum of elements in the current path.

        int sum = 0;

        for (int i = path.size() - 1; i >= 0; i--) {

            sum += path.get(i);

            // If the sum equals k, increment the count.

            if (sum == k) {

                count[0] = count[0] + 1;

            }

        }

        // Backtrack: Remove the last element from the path.

        path.remove(path.size() - 1);

    }

    // Public method to find the number of paths in the binary tree

    // whose elements sum up to k.

    public int sumK(Node root, int k) {

        // Create an empty list to store the current path.

        ArrayList<Integer> path = new ArrayList<>();

        // Create an array to store the count. Using an array allows passing it by reference.

        int[] count = new int[1];

        // Call the private helper method to find the paths.

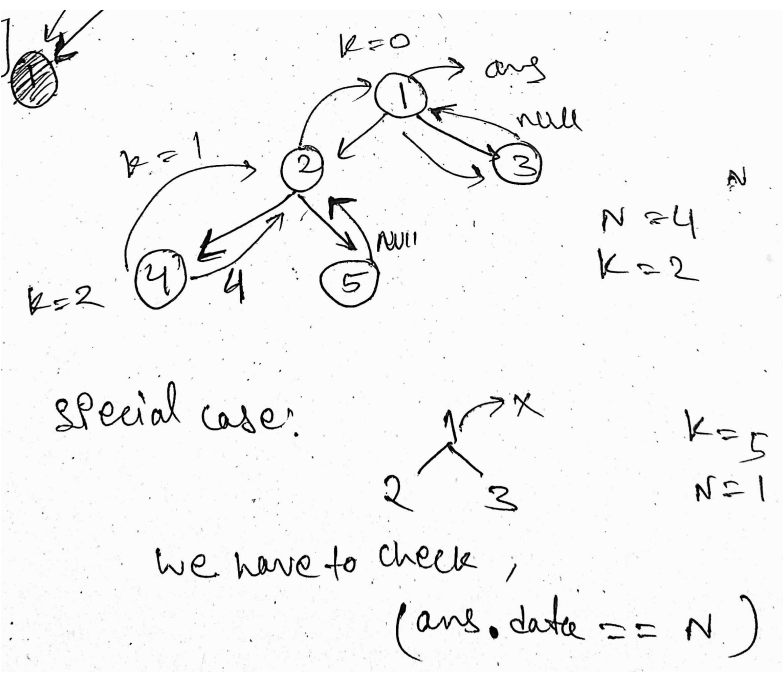
        solve(root, k, path, count);

        // Return the count of paths found.

        return count[0];

    }

**209\_Kth Ancestor of node**



// Utility function to find the kth ancestor of a given node in a binary tree.

    private Node kthAncestor\_util(Node root, int[] temp\_k, int node) {

        // Base case: If the current node is null, return null.

        if (root == null) {

            return null;

        }

        // If the current node's data matches the target node, return the current node.

        if (root.data == node) {

            return root;

        }

        // Recursively search for the target node in the left subtree.

        Node leftAns = kthAncestor\_util(root.left, temp\_k, node);

        // Recursively search for the target node in the right subtree.

        Node rightAns = kthAncestor\_util(root.right, temp\_k, node);

        // If the target node is found in the left subtree but not in the right subtree.

        if (leftAns != null && rightAns == null) {

            // Decrement the kth ancestor counter.

            temp\_k[0] = temp\_k[0] - 1;

            // If the counter reaches 0 or below, return the current node as the kth ancestor.

            if (temp\_k[0] <= 0) {

                // Lock the answer to prevent further processing.

                temp\_k[0] = Integer.MAX\_VALUE;

                return root;

            }

            // Otherwise, return the left ancestor.

            return leftAns;

        }

        // If the target node is found in the right subtree but not in the left subtree.

        if (rightAns != null && leftAns == null) {

            // Decrement the kth ancestor counter.

            temp\_k[0] = temp\_k[0] - 1;

            // If the counter reaches 0 or below, return the current node as the kth ancestor.

            if (temp\_k[0] <= 0) {

                // Lock the answer to prevent further processing.

                temp\_k[0] = Integer.MAX\_VALUE;

                return root;

            }

            // Otherwise, return the right ancestor.

            return rightAns;

        }

        // If the target node is not found in either subtree, return null.

        return null;

    }

    // Main method to find the kth ancestor of a given node in a binary tree.

    public int kthAncestor(Node root, int k, int node) {

        // Array to store the kth ancestor counter.

        int[] temp\_k = new int[1];

        temp\_k[0] = k;

        // Call the utility function to find the kth ancestor.

        Node ans = kthAncestor\_util(root, temp\_k, node);

        // Special cases:

        // 1. If no ancestor found or the found ancestor is the root, return -1.

        // 2. If the ancestor is found, return its data.

        if (ans == null || ans.data == node) {

            return -1;

        } else {

            return ans.data;

        }

    }

**205\_Maximum Sum of nodes in Binary tree such that no two are adjacent**

1

/ \

2 3

/ / \

4 5 6

Output: 16

Explanation: The maximum sum is sum of

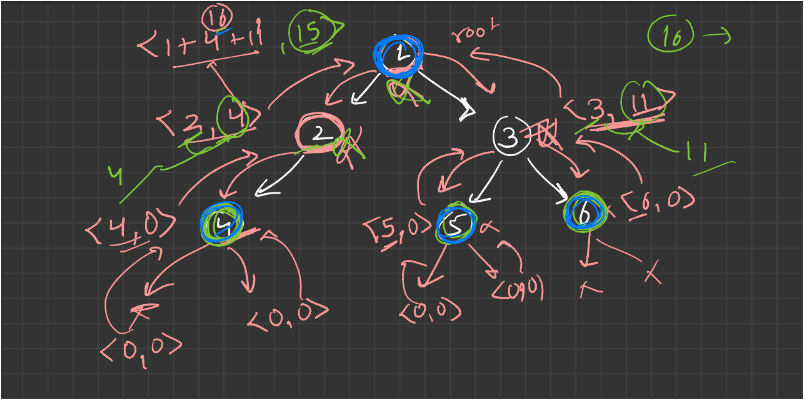
nodes 1 4 5 6 , i.e 16. These nodes are

non adjacent.

Use pair to store inSum and exSum;

inSum = max sum by including nodes at current level

exSum = max sum by excluding nodes at current level



// Recursive method to solve the problem.

    static Pair solve(Node root){

        // If the current node is null, return a Pair with both sums as 0.

        if(root==null){

            return new Pair(0,0);

        }

        // Recursively solve for the left subtree.

        Pair left = solve(root.left);

        // Recursively solve for the right subtree.

        Pair right = solve(root.right);

        // Create a Pair to hold the result.

        Pair res = new Pair();

        // Calculate the sum including the current node.

        res.inSum = root.data+left.exSum+right.exSum;

        // Calculate the sum excluding the current node.

        res.exSum = Math.max(left.inSum, left.exSum) + Math.max(right.inSum, right.exSum);

        // Return the result Pair.

        return res;

    }

    // Function to return the maximum sum of non-adjacent nodes.

    static int getMaxSum(Node root)

    {

        // Call the solve method to get the Pair with maximum sums.

        Pair ans = solve(root);

        // Return the maximum of the two sums from the result Pair.

        return Math.max(ans.inSum, ans.exSum);

}

**Burning Tree**

Step 1: create parent mapping

Step 2: find target node

Step 3:

Maintain a visited array

While traversing, check if new nodes being added to the tree , time++

**TC : O( Nlog(N) )**

**SC : O( N )**

// HashMap to store parent-child relationship

    static HashMap<Node, Node> childToParent = new HashMap<>();

    // Method to create mapping from nodes to their parents

    private static Node createMapping(Node root, int target) {

        Node res = null; // Initialize the result node to null

        Queue<Node> q = new LinkedList<>(); // Create a queue for BFS traversal

        q.add(root);

        // Since the root has no parent, put it into the map with a null parent

        childToParent.put(root, null);

        while (!q.isEmpty()) {

            Node cur = q.remove();

            // Update the result node if it matches the target

            if (cur.data == target) {

                res = cur;

            }

            // If the current node has a left child, put it into the map with the current node as its parent

            if (cur.left != null) {

                childToParent.put(cur.left, cur);

                q.add(cur.left); // Enqueue the left child

            }

            // If the current node has a right child, put it into the map with the current node as its parent

            if (cur.right != null) {

                childToParent.put(cur.right, cur);

                q.add(cur.right);

            }

        }

        return res;

    }

    private static void burnTree(Node targetNode, int[] ans) {

        // HashMap to keep track of visited nodes

        HashMap<Node, Boolean> visited = new HashMap<>();

        Queue<Node> q = new LinkedList<>();

        boolean flag = false; // Flag to check if any node is burned in each iteration

        q.add(targetNode);

        visited.put(targetNode, true);

        while (!q.isEmpty()) {

            int size = q.size();

            flag = false; // Reset flag for each iteration

            // Process all nodes at the current level

            for (int i = 0; i < size; i++) {

                Node cur = q.remove();

                // Check if the left child exists and is not visited

                if (cur.left != null && !visited.getOrDefault(cur.left, false)) {

                    flag = true; // Set flag to true as the left child is burned

                    q.add(cur.left); // Enqueue the left child

                    visited.put(cur.left, true); // Mark the left child as visited

                }

                // do same for right child

                if (cur.right != null && !visited.getOrDefault(cur.right, false)) {

                    flag = true;

                    q.add(cur.right);

                    visited.put(cur.right, true);

                }

                // do same for parent node

                if (childToParent.get(cur) != null && !visited.getOrDefault(childToParent.get(cur), false)) {

                    flag = true; // Set flag to true as the parent node is burned

                    q.add(childToParent.get(cur));

                    visited.put(childToParent.get(cur), true);

                }

            }

            // If any node is burned in the current iteration, increment the burning time

            if (flag) {

                ans[0]++;

            }

        }

    }

    public static int minTime(Node root, int target) {

        // Find the target node in the tree

        Node targetNode = createMapping(root, target);

        int[] ans = {0}; // Array to store the burning time

        burnTree(targetNode, ans);

        return ans[0]; // Return the burning time

    }

**Flatten a tree into linkedlist using O(1) space**

For constant space we will use concept of morris traversal.

**TC : O( N )**

**SC : O( 1 )**

public void flatten(TreeNode root) {

    // Check if the root is null

    if(root==null){

        return; // If so, return as there's nothing to flatten

    }

    TreeNode cur = root; // Start from the root node

    TreeNode pre = null; // Initialize a predecessor node

    while(cur != null){

        // If the current node has a left child

        if(cur.left != null){

            pre = cur.left; // Find the rightmost node in the left subtree (predecessor)

            while(pre.right != null){

                pre = pre.right; // Traverse to the rightmost node

            }

            // Link the right pointer of predecessor to the current node's right subtree

            pre.right = cur.right;

            // Move the left subtree to the right

            cur.right = cur.left;

            // Clear the left subtree after moving it to the right

            cur.left = null;

        }

        // Move to the right child

        cur = cur.right;

    }

}