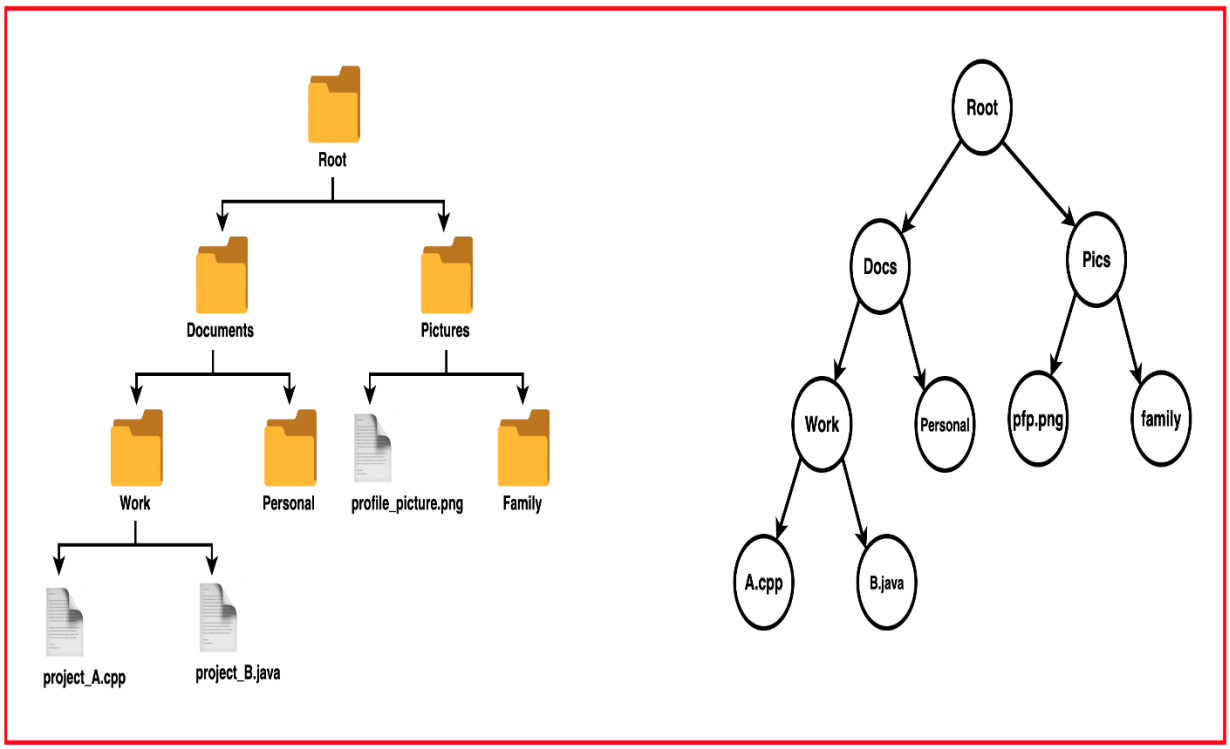
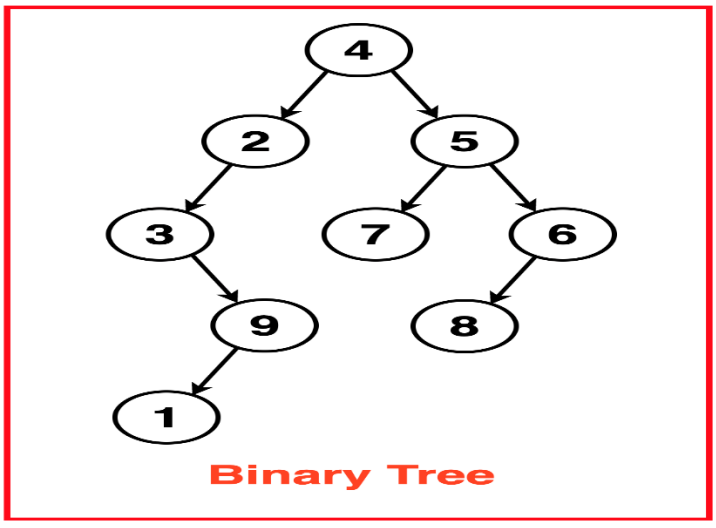
Introduction to Trees

In the world of data structures and algorithms, understanding binary trees lays the groundwork for hierarchical organisation and efficient data manipulation.

Up until now, we have studied array, linked list, stack and queues which are the fundamental linear data structures. Binary Trees are a different data structure and allow hierarchical organisation and structure of multi-level sequences. This resembles a tree with branching at each node expanding the tree in a non-linear fashion.

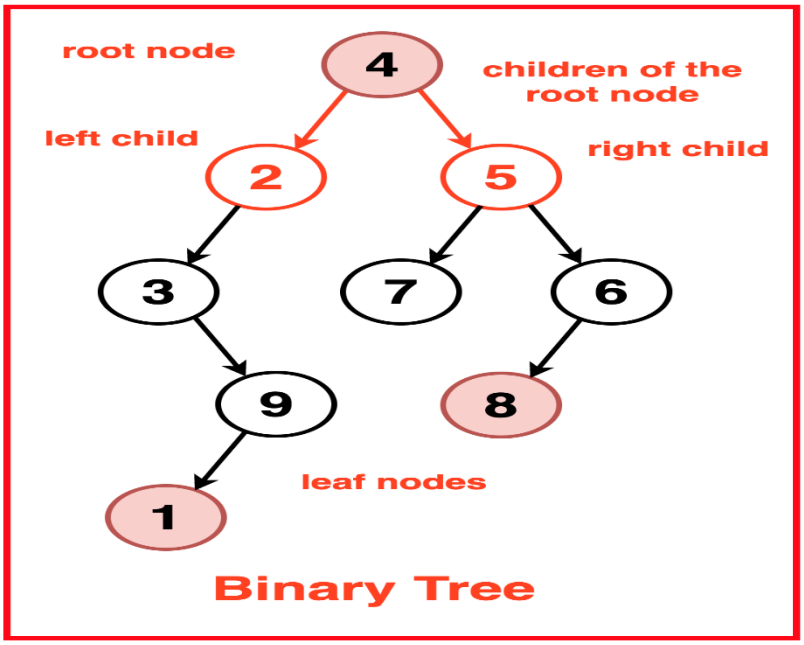


Just as the folders, subfolders and files are hierarchically arranged in your computer's file system, the binary tree has a similar structure with nodes representing folders and their children nodes representing the sub directory or files inside it.



Binary Tree: Binary tree is a fundamental hierarchical data structure in computer science that comprises nodes arranged in a tree-like structure. It consists of nodes, where each node can have at most two children nodes, known as the left child and the right child.

Nodes: Each node in a binary tree contains a piece of data, often referred to as the node's value or key. This node also contains references and pointers to its left and right children so that we can traverse from one node to another in a hierarchical order.



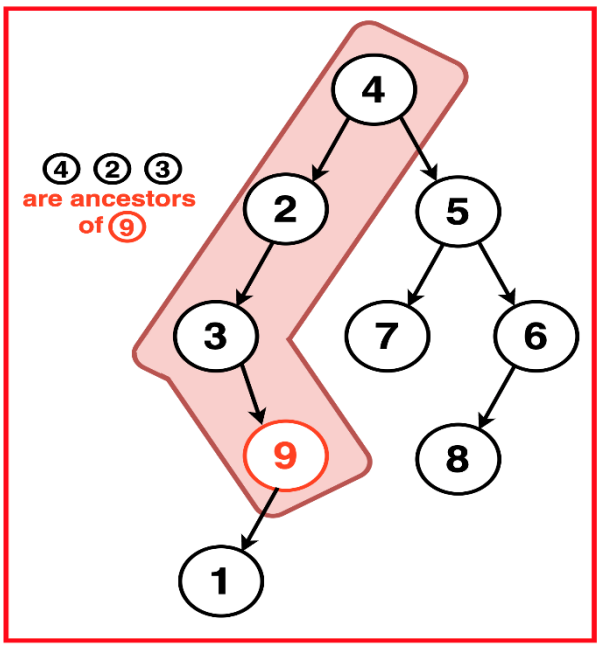
Root Node: is the topmost node of a binary tree from which all other nodes stem out. This serves as the entry point for traversing the tree structure.

Children Nodes are the nodes directly connected to a parent node. In a binary tree, a parent node can have zero, one or two children nodes, each situated to the left or right of the parent node.

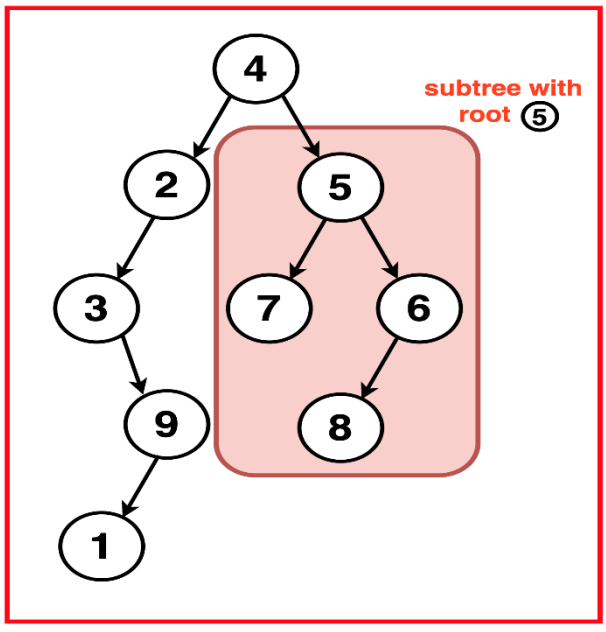
Leaf Nodes are the nodes that do not have any children. These nodes lie on the outermost ends of the tree branches and are the terminal points of the traversal.

Ancestor: An ancestor of a node N is any node on the path from the root of the tree to N, excluding N itself.

Descendant: A descendant of a node N is any node in the subtree rooted at N, excluding N itself.



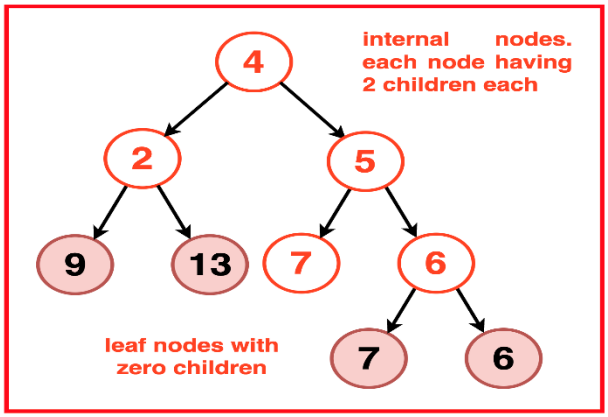
Sub-tree: A subtree is a part of a tree that is itself a tree, consisting of a specific node and all of its descendants



Types of BT (Binary trees):

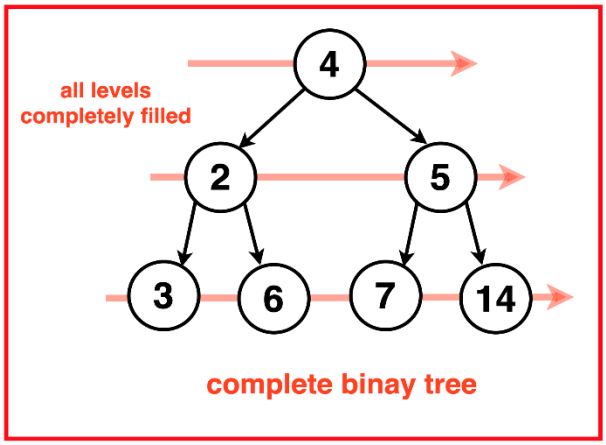
1. Full Binary Tree:

- Also known as a Strict Binary Tree.
- Every node has either zero or two children.



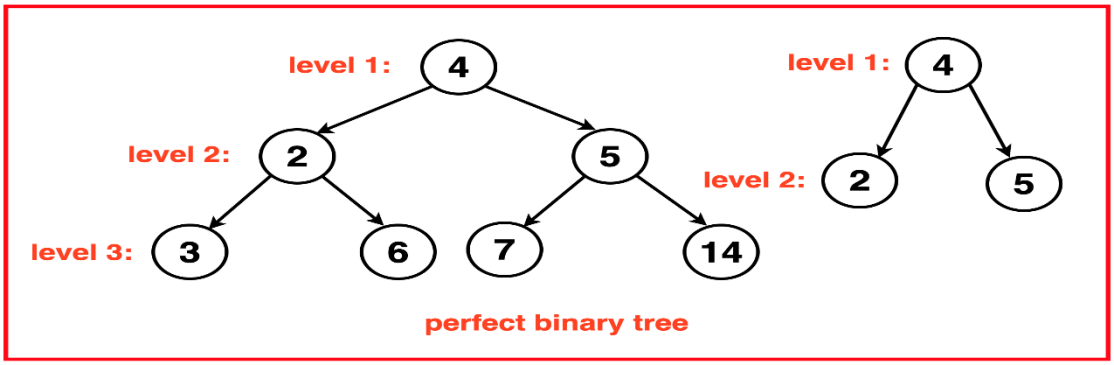
2. Complete Binary Tree:

All levels of the tree, except possibly the last one, are fully filled. If the last level is not completely filled, it is filled from left to right, ensuring that nodes are positioned as far left as possible.



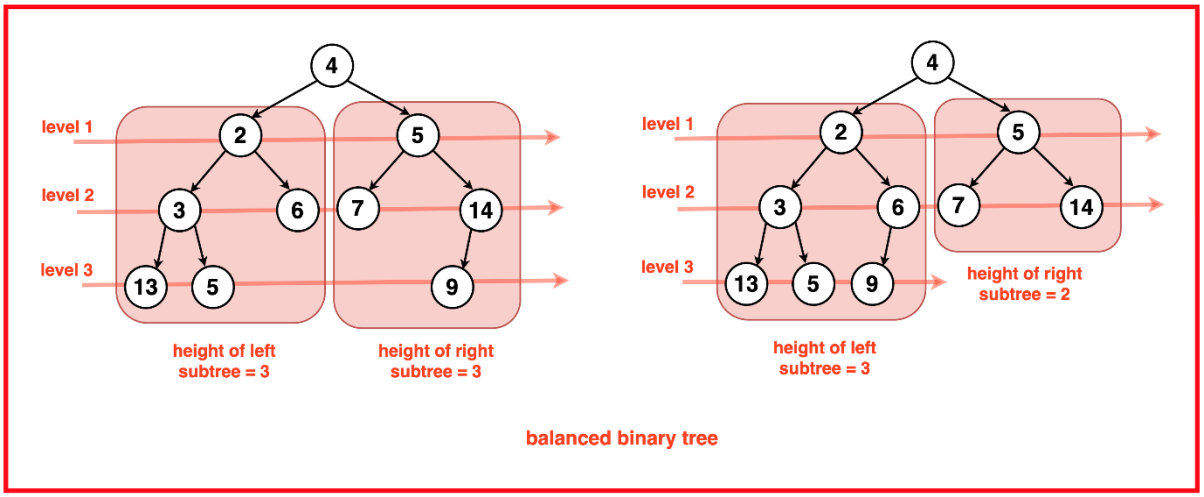
3. Perfect Binary Tree:

A Perfect Binary tree is a type of Binary Tree where all leaf nodes are at the same level

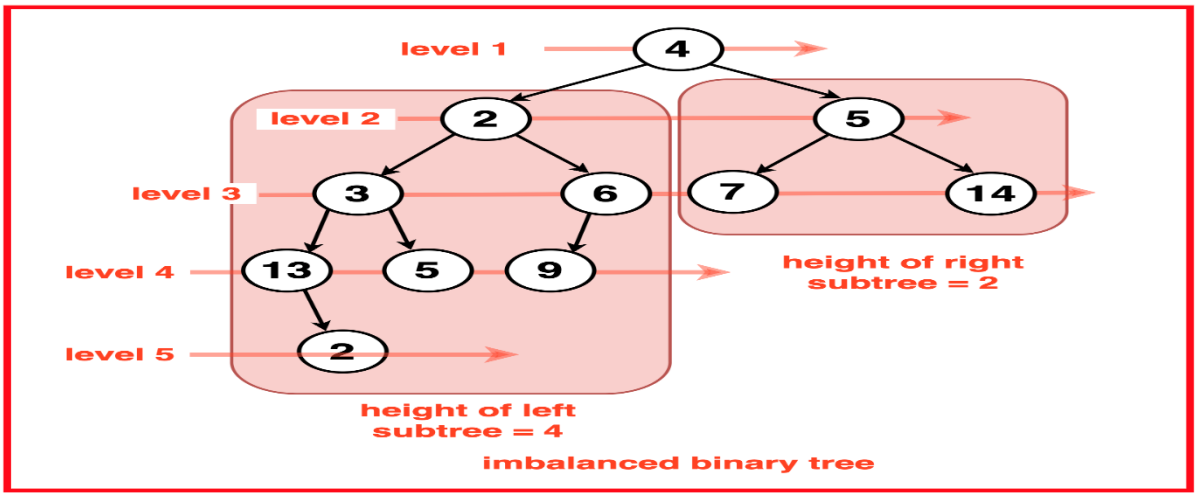


4. Balanced Binary Tree:

A Balanced Binary tree is a type of Binary Tree where the heights of the two subtrees of any node differ by at most one. This property ensures that the tree remains relatively well-balanced, preventing the tree from becoming highly skewed or degenerate.



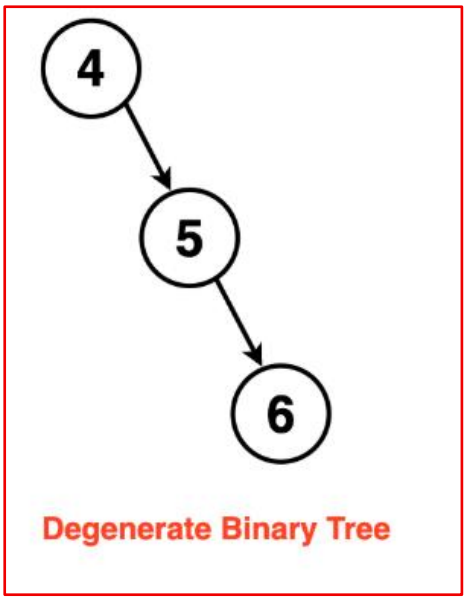
In a balanced binary tree, the height of the tree should be log2N at maximum, where N is the number of nodes.



5. Degenerate Tree:

A Degenerate Tree is a Binary Tree where the nodes are arranged in a single path leaning to the right or left. The tree resembles a linked list in its structure where each node points to the next node in a linear fashion.

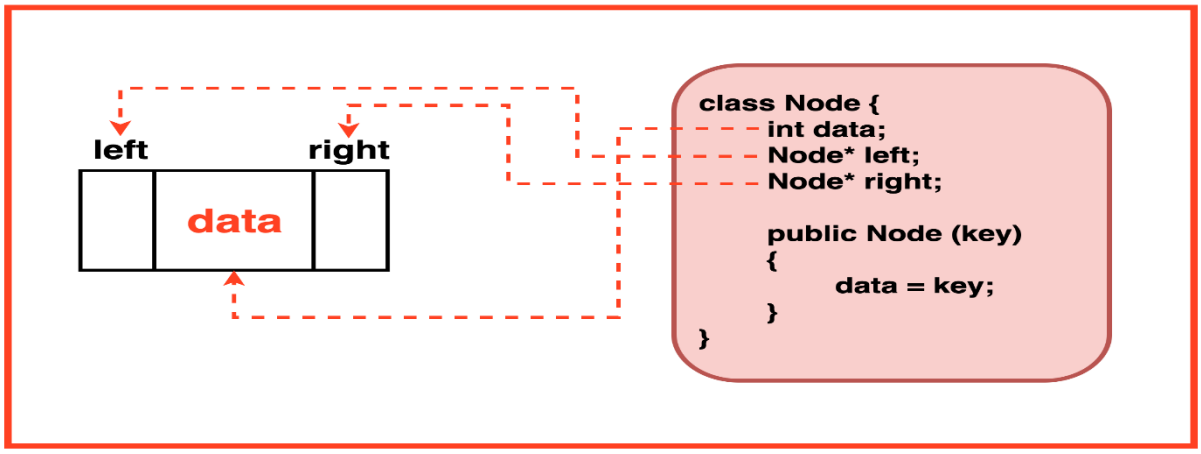
Each level of this tree only has one node.



Binary Tree Representation in Java

In Java, a binary tree is structured using references to other nodes, forming a hierarchical arrangement where each node can refer to at most two other nodes: a left child and a right child.

This reference-based approach establishes connections between nodes, enabling traversal and navigation within the tree structure.



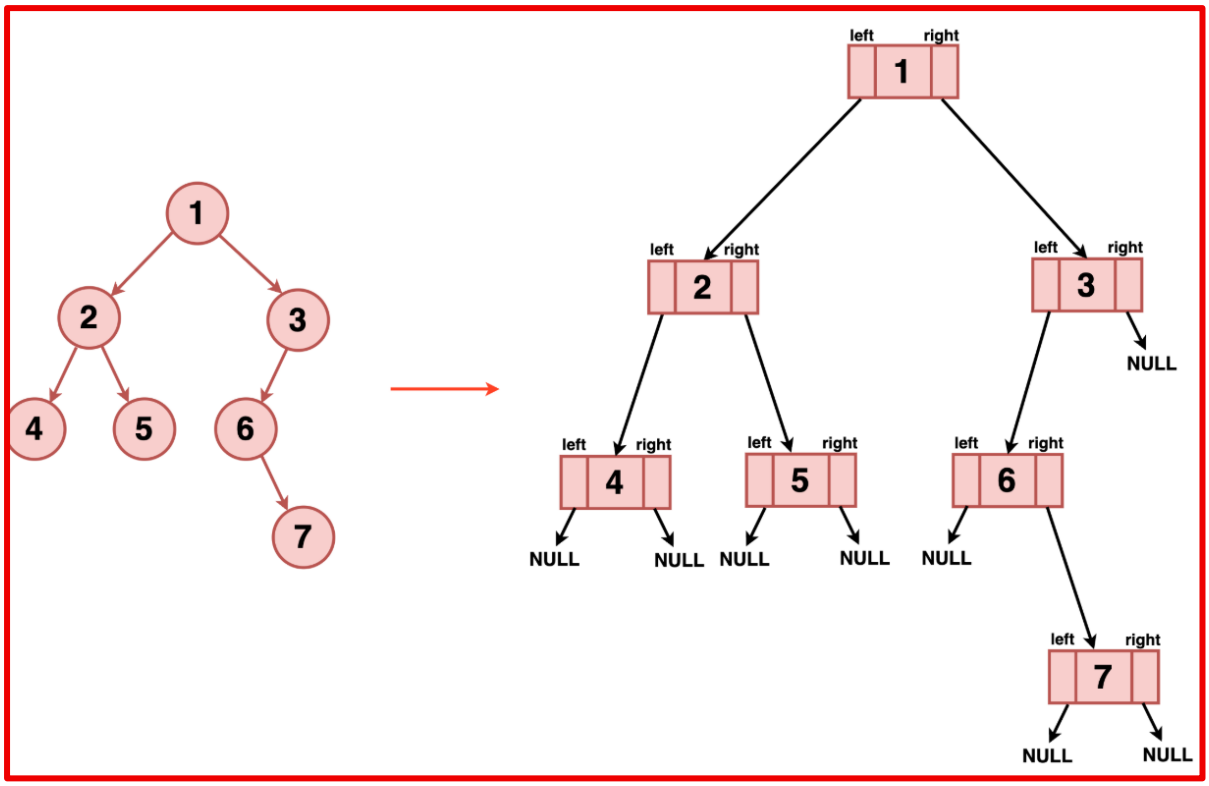
Node Structure:

In Java, a Binary Tree node is represented using a class that encapsulates the attributes of a node:

- **Data Component:** This holds the value of the node, which could be of any data type (e.g., integer, string, object).

- **Pointers to Children:** Two reference variables ie. ‘left’ and ‘right’ point to the left and right child nodes respectively. These references store the memory addresses of child nodes, allowing traversal and access to further nodes in the tree structure.

In Java, references to objects act similarly to pointers in C++, allowing nodes to refer to other nodes without direct memory manipulation.



Node Constructor:

In Java, the constructor of the Node class initialises a node with a specific value and sets its left and right references to null to signify that it doesn't have any children initially.

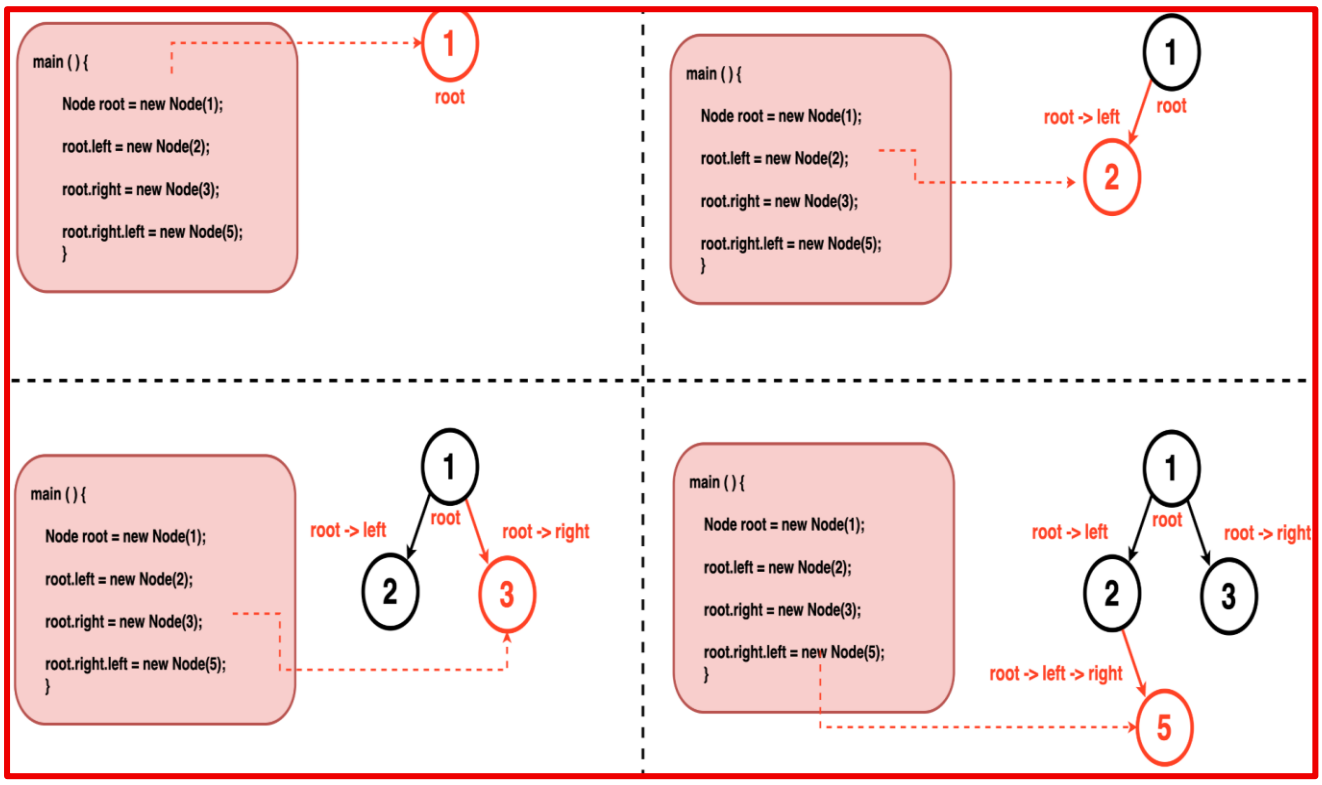
Within the constructor:

- **data = val;** :Sets the data of the node to the provided value (val). This assigns the input integer to the node's data.
- **left = right = NULL;** : Initialises both left and right references as null. This initialization ensures that when a node is created, it does not have any immediate connections to other nodes, indicating the absence of left and right children.

Node Connection:

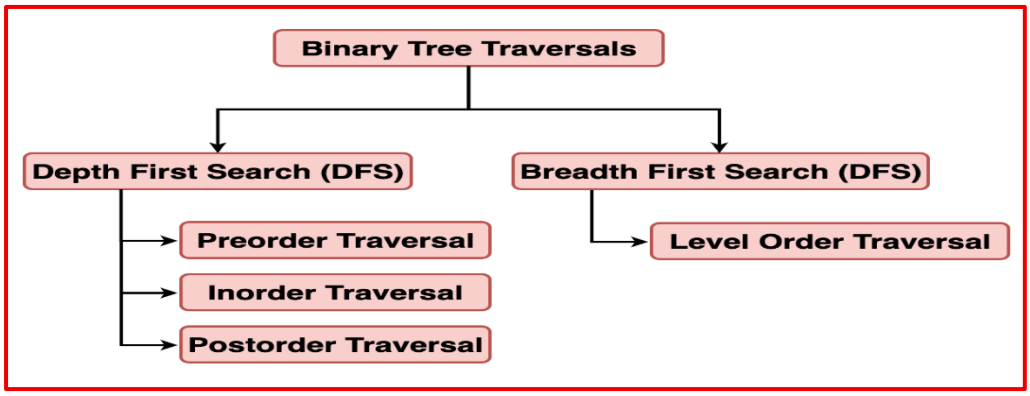
Java utilises references between nodes allowing them to refer to other nodes. When creating a new node, memory is allocated for the node object, and the node’s data is stored within it.

References (`left` and `right`) are initialised as `null` to indicate that the node doesn't currently have any children, The nodes are connected by assigning references of a parent node to its respective left and right child nodes.



Tree traversals:

Traversals in hierarchical data structures like Binary Trees can be broadly classified into two categories: Depth-First Search (DFS) and Breadth-First Search (BFS). Each of them utilises a different strategy to visit the nodes within the tree.



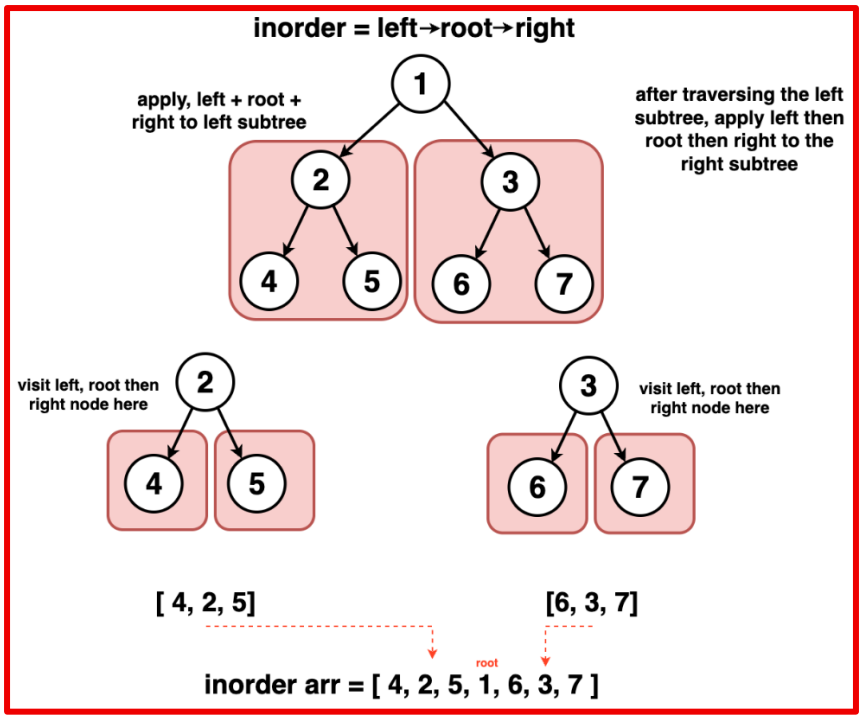
Depth-First Search (DFS) explores a binary tree by going as deeply as possible along each branch before backtracking.

- It starts from the root and explores as deeply as possible along each branch, visiting nodes until it reaches a leaf node. then backtracks to the most recent unexplored node and continues until all nodes are visited.
- The order in which we visit a node determines if that traversal would be preorder, inorder and postorder.
- DFS uses recursion or a stack to traverse deeper levels of the tree before visiting nodes at the same level.

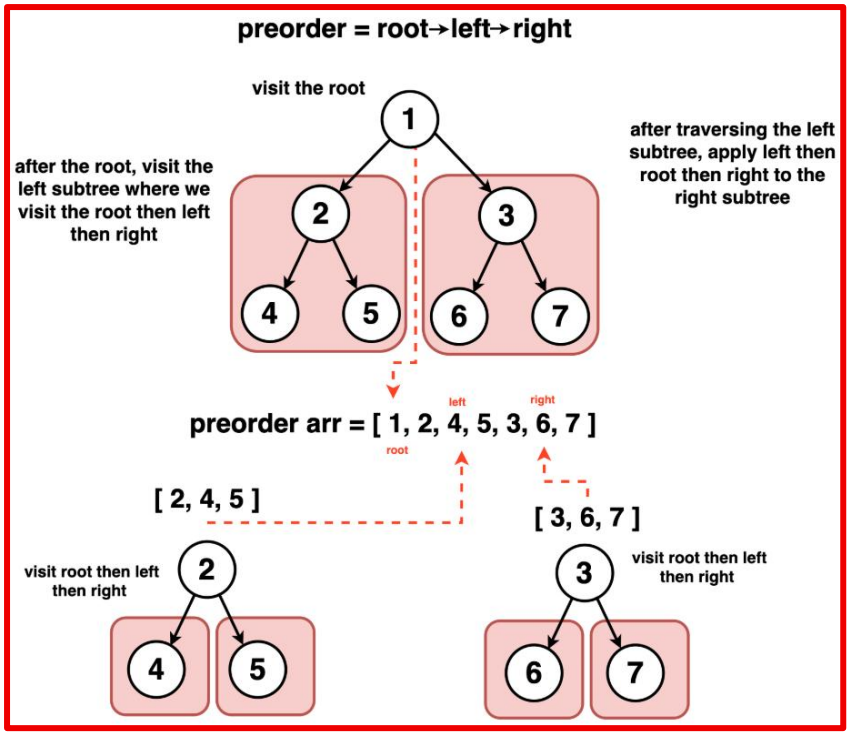
Breadth-First Search (BFS) explores a binary tree level by level, visiting all nodes at a given level before processing to the next level.

- It starts from the root and visits all nodes at level 0, then proceeds to level 1, level 2, and so on. Nodes at a level are visited from left to right.
- BFS uses a queue data structure to maintain nodes at each level, ensuring that nodes at higher levels are visited moving to lower levels.

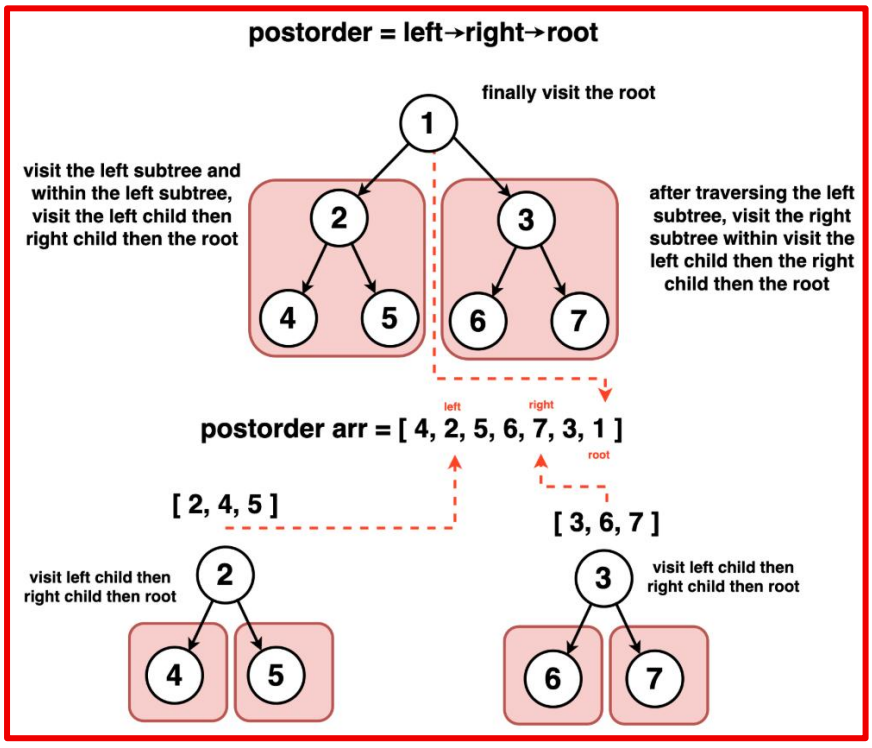
In-order Traversal is the type of Depth First Traversal where nodes are visited in the order: Left, Root, Right.



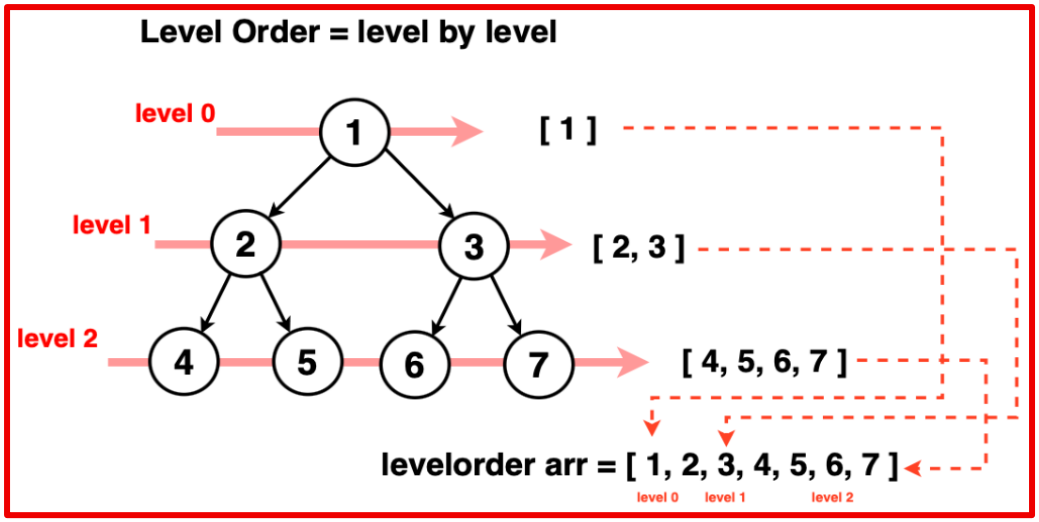
Preorder Traversal is the type of Depth First Traversal where nodes are visited in the order: Root, Left then Right.



Post-order Traversal is the type of Depth First Traversal where nodes are visited in the order: Left, Right then Root.



Level Order Traversal is the type of Breadth First Traversal where nodes are visited level by level, exploring each level completely before moving to the next level.



Problem Statement: Given the root of a Binary Tree, write a recursive function that returns an array-List containing the preorder traversal of the tree.

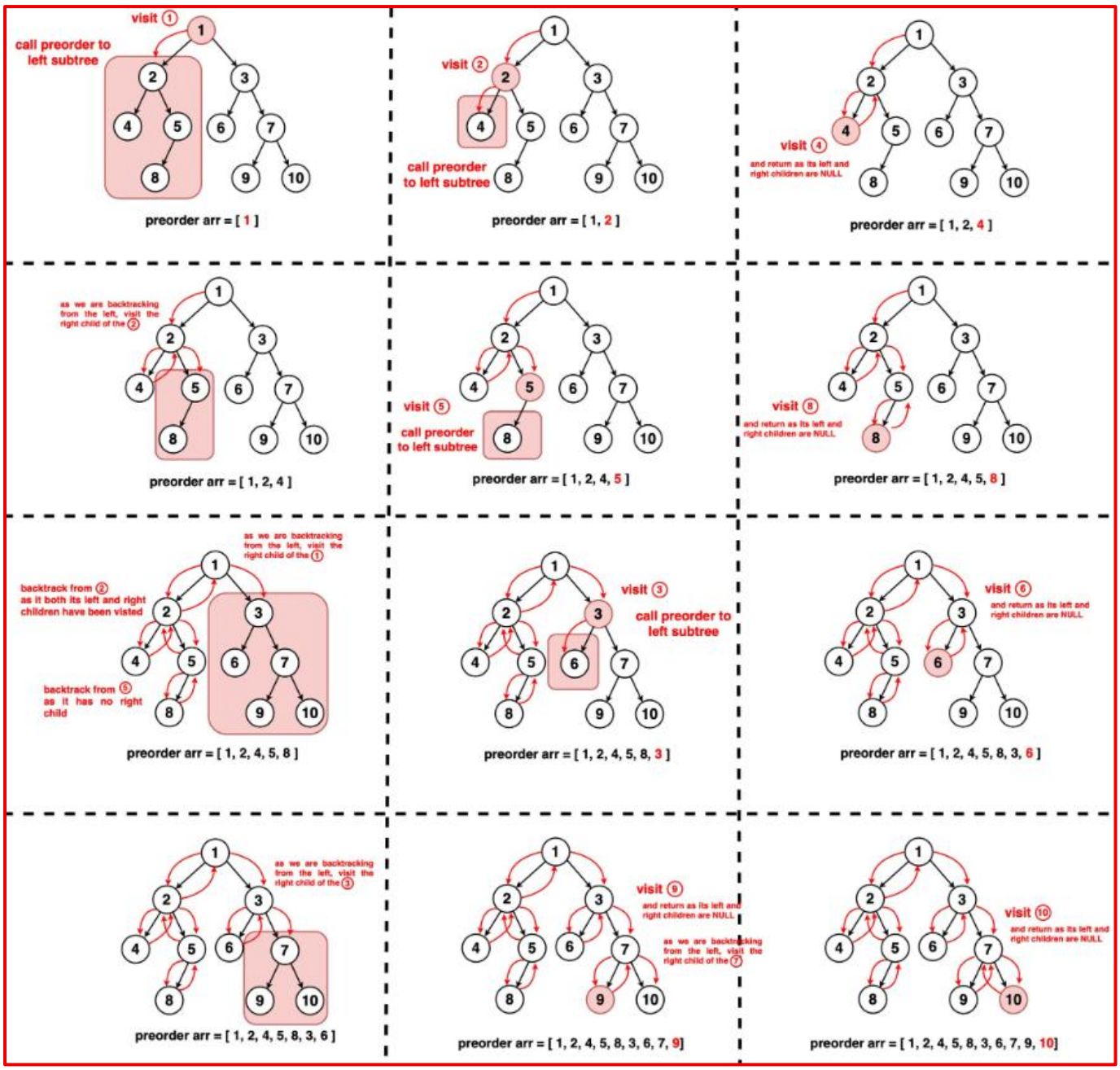
Recursive approach

The algorithm first visits the root node then in the preorder traversal, we visit (ie. add to the arrayList) the current node by accessing its value then we recursively traverse the left subtree in the same manner. We repeat these steps for the left subtree then when we return to the current node, we recursively travel to the right subtree in a preorder manner as well. The sequence of steps in preorder traversal follows: Root, Left, Right.

Base Case: If the current node is null, it means we have reached the end of a subtree and there are no further nodes to explore. Hence the recursive function stops and we return from that particular recursive call.

Recursive Function:

- **Process Current Node:** The recursive function begins by processing ie. adding to the array-list.
- **Traverse Left Subtree:** Recursively traverse the left subtree by invoking the preorder function on the left child of the current node. This step continues the exploration of nodes in a depth first manner.
- **Traverse Right Subtree:** After traversing the entire left subtree,we traverse the right subtree recursively. We once again invoke the preorder function, but this time on the right child of the current node.



Code:

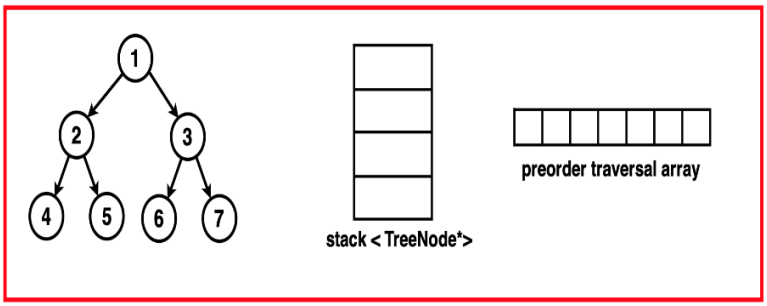
```
static void preOrderTraversal(Node root){
    if (root == null) return;

    System.out.print(root.data+", ");
    preOrderTraversal(root.left);
    preOrderTraversal(root.right);
}
```

Time Complexity: O(N) where N is the number of nodes in the binary tree as each node of the binary tree is visited exactly once.

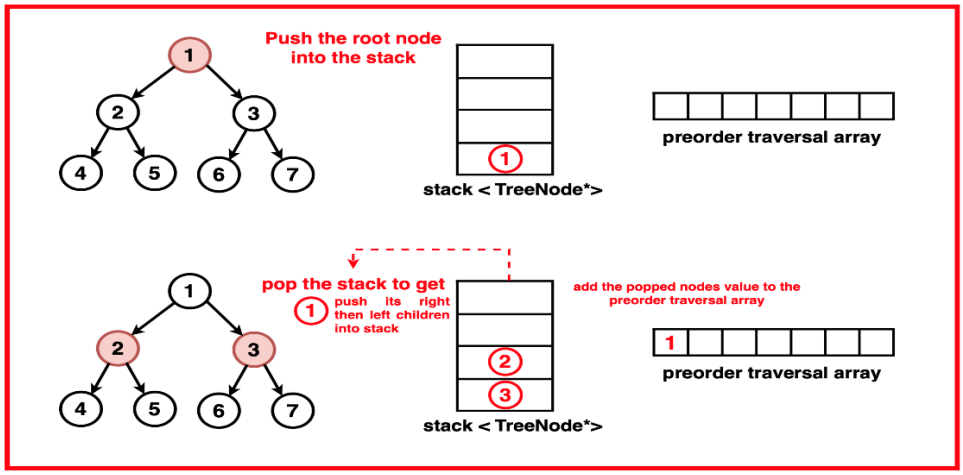
Space Complexity: O(N) where N is the number of nodes in the binary tree. This is because the recursive stack uses an auxiliary space which can potentially hold all nodes in the tree when dealing with a skewed tree (all nodes have only one child), consuming space proportional to the number of nodes.In the average case or for a balanced tree, the maximum number of nodes that could be in the stack at any given time would be roughly the height of the tree hence **O(log2N).**

Iterative approach

Step 1: Initialise an empty Array-list to store the preorder traversal result. Create a stack to store the nodes during traversal and push the root node onto the stack. Check, if the root is null, return an empty traversal result if true.



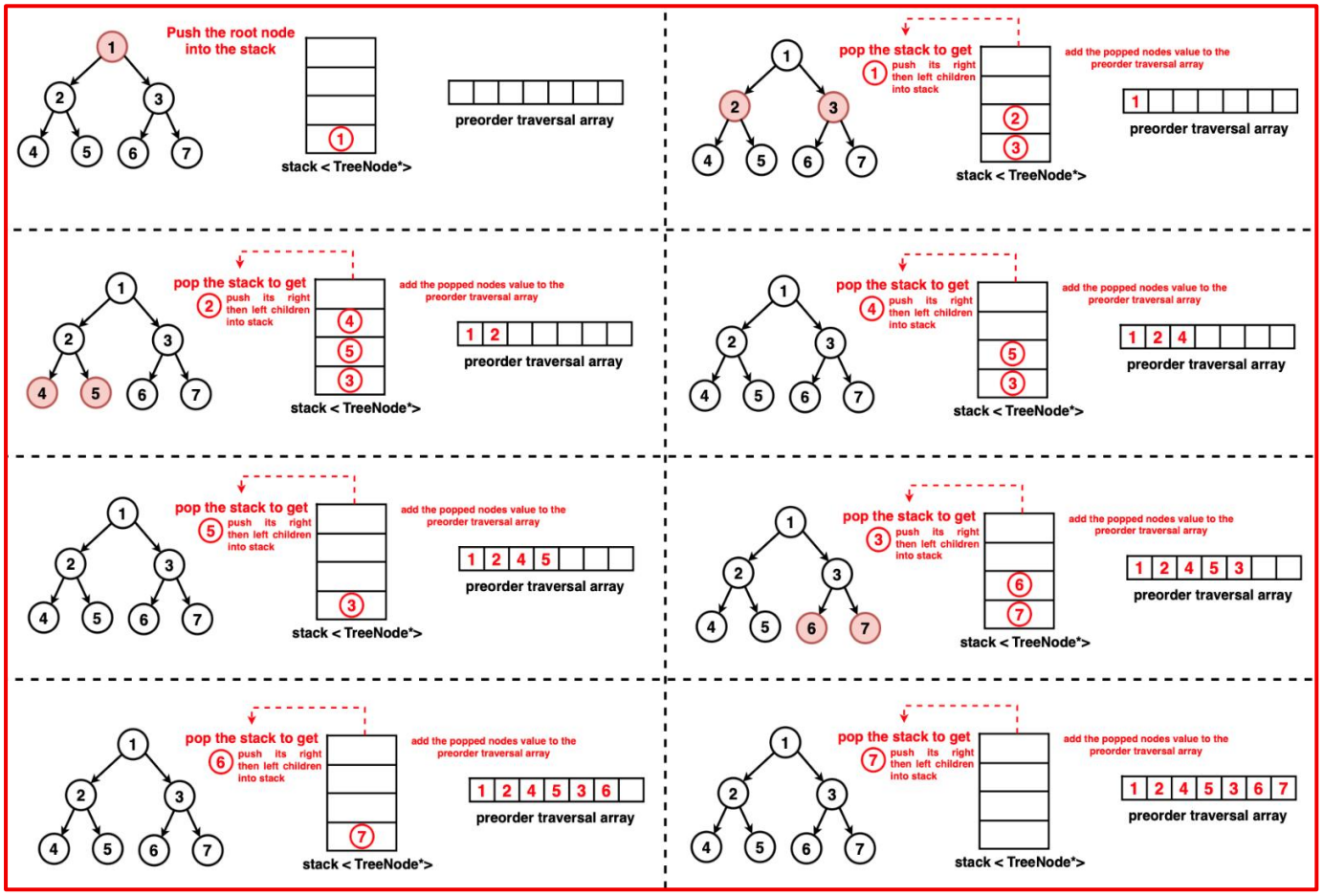
Step 2: Push the root of the binary tree into the stack.



Step 3: While the stack is not empty:

- Get the current node from the top of the stack.
- Remove the node from the stack.
- Add the node's value to the preorder traversal result.

- First, push the right child onto the track if it exists.
- Secondly, push the left child onto the stack if it exists.



Step 4: Return the 'preorder' traversal result.

Code:

```
static List<Integer> iteratively(Node root){  1 usage
    List<Integer> res = new ArrayList<>();
    if (root == null) return res;

    Stack<Node> s = new Stack<>();
    s.add(root);
    while (!s.isEmpty()){
        Node temp = s.pop();
        res.add(temp.data);

        if (temp.right != null)  s.add(temp.right);
        if (temp.left != null) s.add(temp.left);
    }
    return res;
}
```

Time Complexity: O(N) where N is the number of nodes in the binary tree. Every node of the binary tree is visited exactly once, and for each node, the operations performed (pushing and popping from the stack, accessing node values, etc.) are constant time operations.

Space Complexity: O(N) where N is the number of nodes in the binary tree. This is because the stack can potentially hold all nodes in the tree when dealing with a skewed tree (all nodes have only one child), consuming space proportional to the number of nodes.

Problem Statement: Given the root of a Binary Tree, write a recursive function that returns an array-List containing the in-order traversal of the tree.

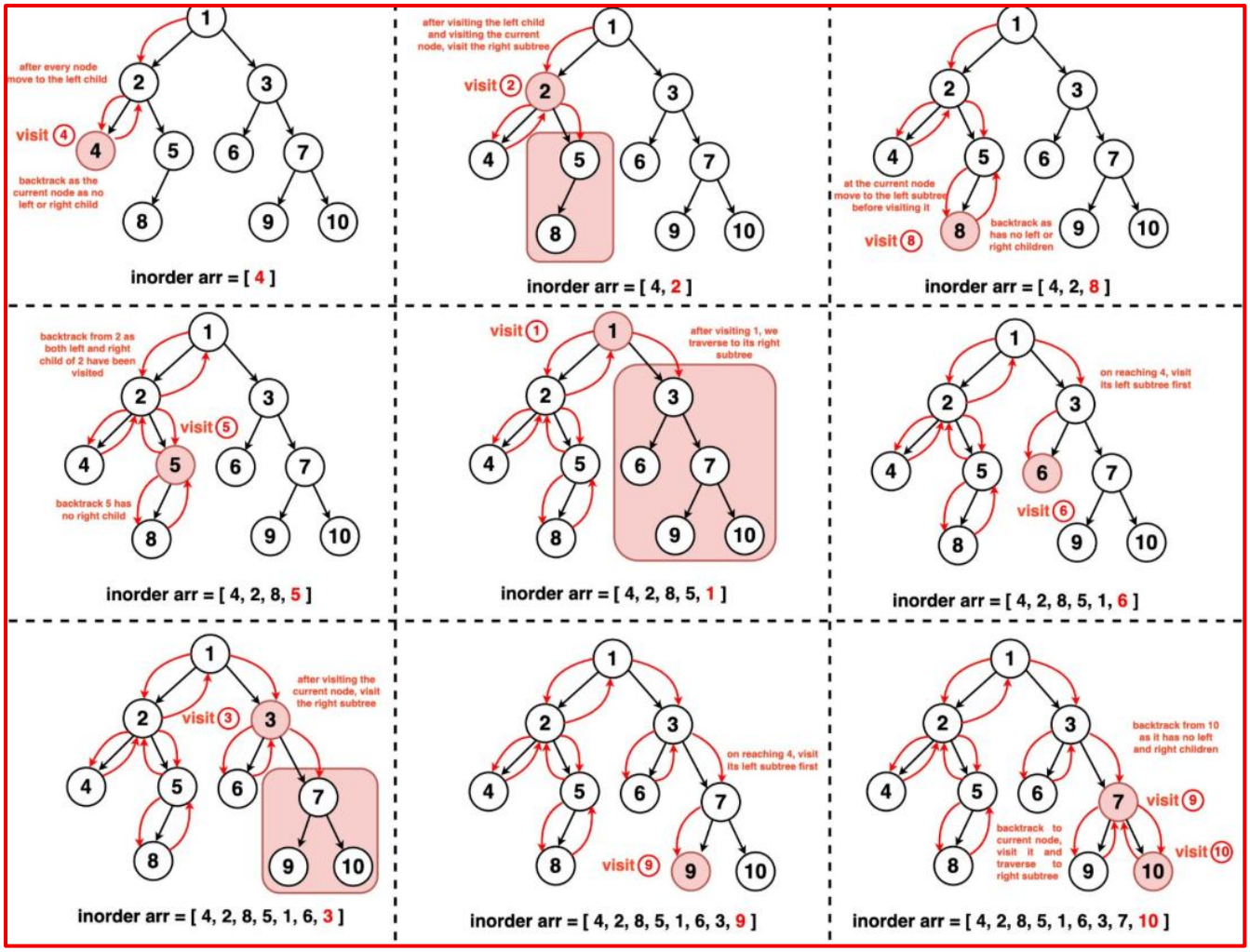
Optimal Approach

In-order traversal stands as one of the depth-first traversal techniques for navigating nodes in a binary tree. The method starts by exploring the left subtree recursively in the following order: left child, root node, right child. Initially, it traverses the left subtree until reaching the leftmost node. Upon reaching a null node, signifying the end of a subtree, the recursive process halts. then, we visit the current node that had invoked the in-order on its left child. After visiting the current node we visit the right subtree in the in-order manner as well.

Base Case: If the current node is null, it means we have reached the end of a subtree and there are no further nodes to explore. Hence the recursive function stops and we return from that particular recursive call.

Recursive Function:

- **Traverse Left Subtree:** Recursively traverse the left subtree by invoking the preorder function on the left child of the current node. This step continues the exploration of nodes in a depth first manner.
- **Process Current Node:** The recursive function begins by processing ie. adding to the array or printing the current node.
- **Traverse Right Subtree:** After traversing the entire left sub-tree, we traverse the right subtree recursively. We once again invoke the preorder function, but this time on the right child of the current node.



Code:

```
static void inOrderTraversal(Node root){
    if (root == null) return;

    inOrderTraversal(root.left);
    System.out.print(root.data+", ");
    inOrderTraversal(root.right);
}
```

Time Complexity: O(N) where N is the number of nodes in the binary tree as each node of the binary tree is visited exactly once.

Space Complexity: O(N) where N is the number of nodes in the binary tree. This is because the recursive stack uses an auxiliary space which can potentially hold all nodes in the tree when dealing with a skewed tree (all nodes have only one child), consuming space proportional to the number of nodes.In the average case or for a balanced tree, the maximum number of nodes that could be in the stack at any given time would be roughly the height of the tree hence **O(log2N).**

Iterative Approach

Algorithm

- Start with an empty result list.
- Initialize an empty stack.
- Set a pointer temp to the root node.
- Repeat until the stack is empty **and** temp is null:
  - If temp is not null:
    - Push temp onto the stack.
    - Move temp to its left child.
  - Otherwise (when temp is null):
    - Pop a node from the stack.
    - Add the popped node's value to the result list.
    - Move temp to the right child of the popped node.
- Once the loop finishes, return the result list.

Code:

```
static List<Integer> iteratively(Node root){  1 usage
    List<Integer> res = new ArrayList<>();
    if (root == null) return res;

    Node temp = root;
    Stack<Node> s = new Stack<>();
    while (true){
        if (temp != null){
            s.add(temp);
            temp = temp.left;
        }
        else{
            if (s.isEmpty()) break;
            temp = s.pop();
            res.add(temp.data);
            temp = temp.right;
        }
    }
    return res;
}
```

Time Complexity: O(N) (each node pushed & popped once).

Space Complexity: O(N) (stack in worst case, skewed tree).

Problem Statement: Given the root of a Binary Tree, write a recursive function that returns an array-List containing the post-order traversal of the tree.

Optimal Approach

Post-order traversal, another depth-first method in tree exploration, follows a sequence where the algorithm first explores the left subtree, then the right subtree, and finally visits the root node. In post-order traversal, we visit (or add to the array) the current node after traversing both its left and right sub-trees. The sequence of steps in preorder traversal follows: Left, Right, Root.

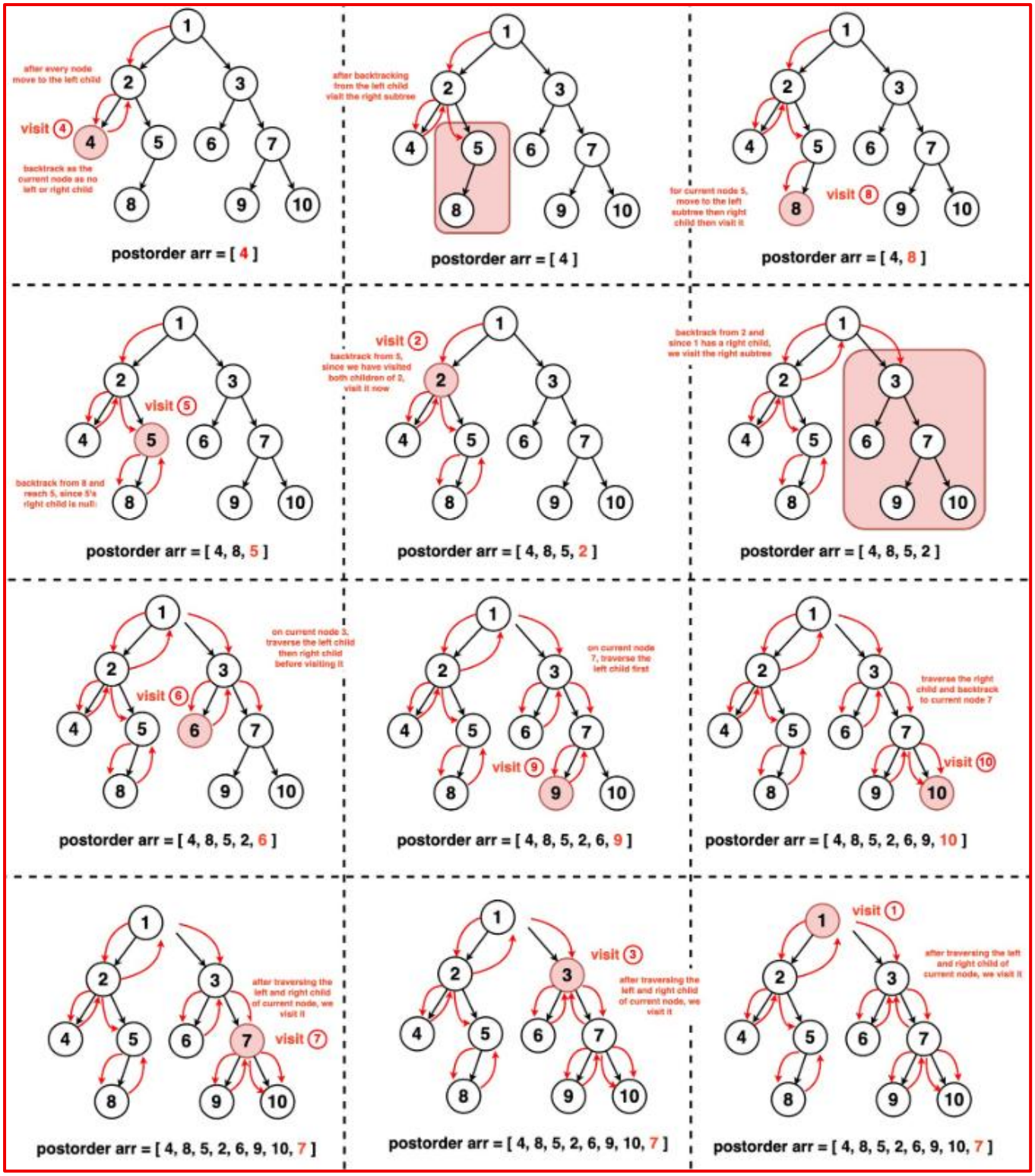
Base Case: If the current node is null, it means we have reached the end of a subtree and there are no further nodes to explore. Hence the recursive function stops and we return from that particular recursive call.

Recursive Function:

- **Traverse Left Subtree:** Explore the left subtree by recursively invoking the post-order function on the left child of the current node, ensuring a depth-first approach to node exploration.
- **Traverse Right Subtree:** After fully traversing the left subtree, we move on to the right subtree, invoking the post-order function on the right child of the current node.
- **Process Current Node:** After exploring the children of the current node, we process it by adding its value to the post-order traversal array.

Time Complexity: O(N) where N is the number of nodes in the binary tree as each node of the binary tree is visited exactly once.

Space Complexity: O(N) where N is the number of nodes in the binary tree. This is because the recursive stack uses an auxiliary space which can potentially hold all nodes in the tree when dealing with a skewed tree (all nodes have only one child), consuming space proportional to the number of nodes.In the average case or for a balanced tree, the maximum number of nodes that could be in the stack at any given time would be roughly the height of the tree hence **O(log2N).**


after every node move to the left child
visit 4
backtrack as the current node as no left or right child
postorder arr = [4]
after backtracking from the left child visit the right subtree
postorder arr = [4]
for current node 5, move to the left subtree then right child then visit it
visit 8
postorder arr = [4, 8]
visit 5
backtrack from it and reach 5, since 5's right child is null
postorder arr = [4, 8, 5]
visit 2
backtrack from 5, since we have visited both children of 2, visit it now
postorder arr = [4, 8, 5, 2]
backtrack from 2 and since 1 has a right child, we visit the right subtree
postorder arr = [4, 8, 5, 2]
visit 6
on current node 3, traverse the left child then right child before visiting it
postorder arr = [4, 8, 5, 2, 6]
on current node 7, traverse the left child first
visit 9
postorder arr = [4, 8, 5, 2, 6, 9]
traverse the right child and backtrack to current node 7
visit 10
postorder arr = [4, 8, 5, 2, 6, 9, 10]
after traversing the left and right child of current node, we visit it
visit 7
postorder arr = [4, 8, 5, 2, 6, 9, 10, 7]
visit 3
after traversing the left and right child of current node, we visit it
postorder arr = [4, 8, 5, 2, 6, 9, 10, 7]
visit 1
after traversing the left and right child of current node, we visit it
postorder arr = [4, 8, 5, 2, 6, 9, 10, 7]

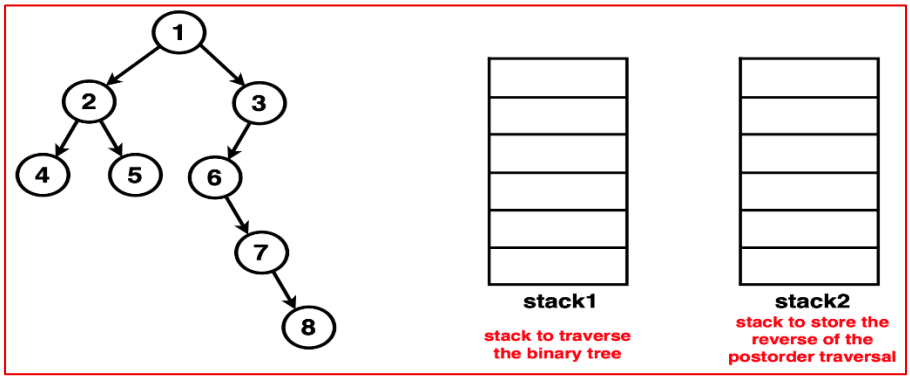
Code:

```java
static void postOrderTraversal(Node root){
    if (root == null) return;

    postOrderTraversal(root.left);
    postOrderTraversal(root.right);
    System.out.print(root.data+", ");
}
```

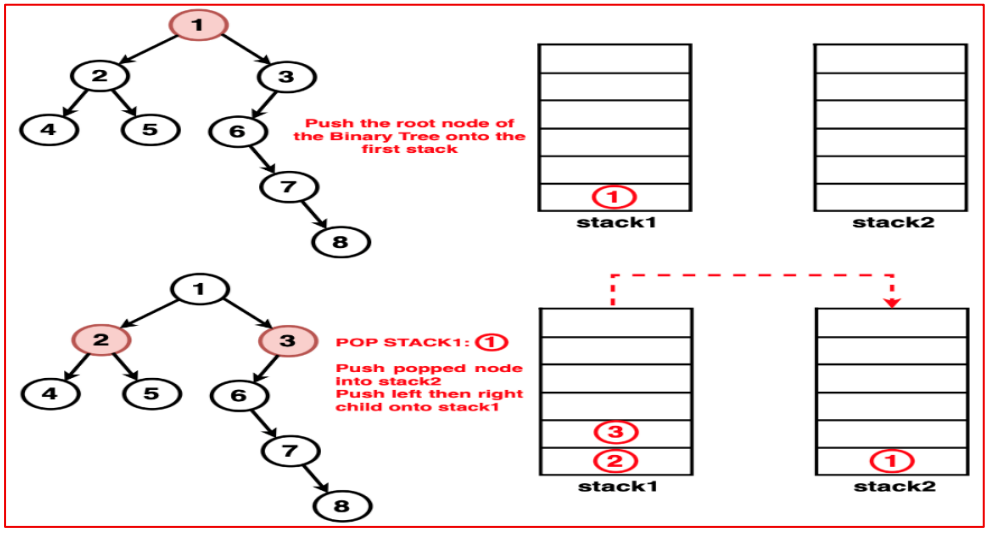
Iterative Approach Using two stacks

Algorithm:

Step 1: Create two stacks: one for holding nodes and another for storing the final post-order traversal sequence. Initialise an array-list to store the traversal sequence.

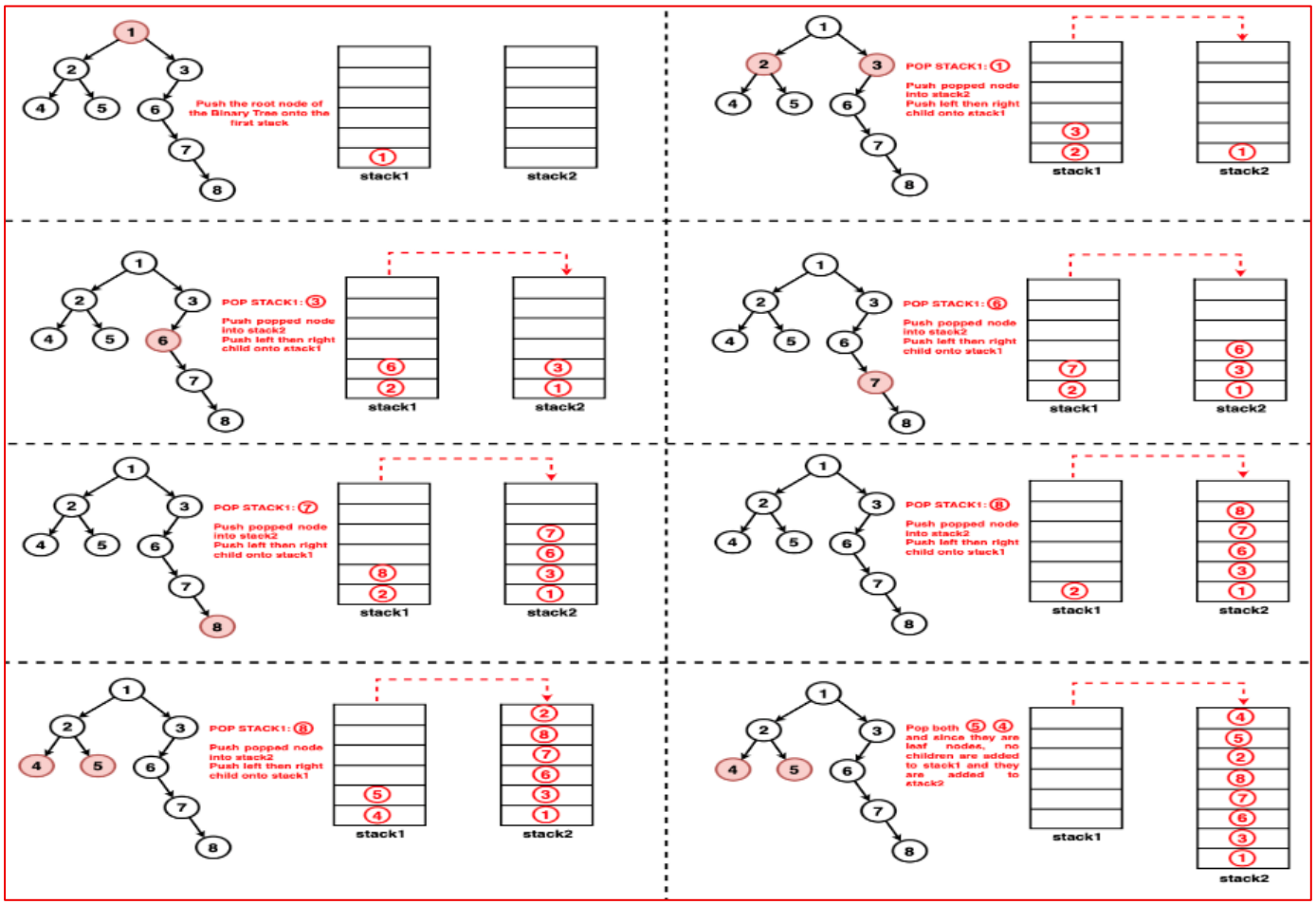


Step 2: Push the root node to the first stack.

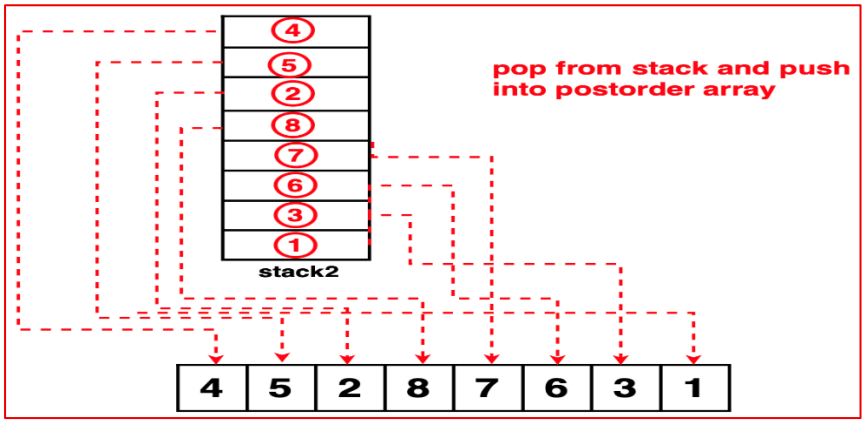


Step 3: Process the nodes until the first stack is empty:

- Pop a node from the top of the first stack.
- Push this node onto the second stack.
- Push its left child (if it exists) onto the first stack.
- Push its right child (if it exists) onto the first stack.



Step 4: Once the first stack is empty, retrieve the nodes in the post-order sequence by popping nodes from the second stack one by one and store them in the Array-List.



Code:

```
static List<Integer> iterativelyUsingTwoStacks(Node root){
    Stack<Node> s1 = new Stack<>();
    Stack<Node> s2 = new Stack<>();
    List<Integer> res = new ArrayList<>();

    if (root == null) return res;

    s1.add(root);
    while (!s1.isEmpty()){
        Node temp = s1.pop();
        s2.add(temp);
        if (temp.left != null) s1.add(temp.left);
        if (temp.right != null) s1.add(temp.right);
    }

    while (!s2.isEmpty()){
        res.add(s2.pop().data);
    }
    return res;
}
```

Time Complexity: O(2N) where N is the number of nodes in the Binary Tree. The traversal process visits each node in the Binary Tree exactly once to push into stack1 and stack2. Then after the tree is traversed and the nodes are popped from stack2 to push into the Array-list.

Space Complexity: O(2N) where N is the number of nodes in the Binary Tree. The space occupied by the two stacks depend on the height of the binary tree. In the worst-case scenario, if the tree is skewed, the space complexity would be O(N) as both stacks could potentially hold all nodes at different points during traversal.

Iterative Approach Using One stacks

Algorithm:

1. Create an empty result list.

2. If the root is null, return the result list.

3. Initialize an empty stack.

4. Set a pointer current to the root.

5. Repeat until both current is null and the stack is empty:

- If current is not null:
    - Push current onto the stack.
    - Move current to its left child.

- Otherwise (when current is null):
    - Look at the right child of the node on top of the stack.
    - If the right child is null:
        - Pop the node from the stack and add its value to the result list.
        - Keep popping while the popped node is the right child of the new top of the stack, adding each to the result.
    - If the right child is not null:
        - Move current to that right child.

6. Return the result list.

Code:

```
static List<Integer> iterativelyUsingOneStacks(Node root){  1 usage
    List<Integer> res = new ArrayList<>();
    if (root == null) return res;

    Node current = root;
    Stack<Node> s = new Stack<>();

    while (current != null || !s.isEmpty()){
        if (current != null){
            s.add(current);
            current = current.left;
        }
        else{
            Node temp = s.peek().right;
            if (temp == null){
                temp = s.pop();
                res.add(temp.data);

                while (!s.isEmpty() && temp == s.peek().right){
                    temp = s.pop();
                    res.add(temp.data);
                }
            }
            else current = temp;
        }
    }
    return res;
}
```

Time Complexity: O(N) (each node is pushed and popped once).

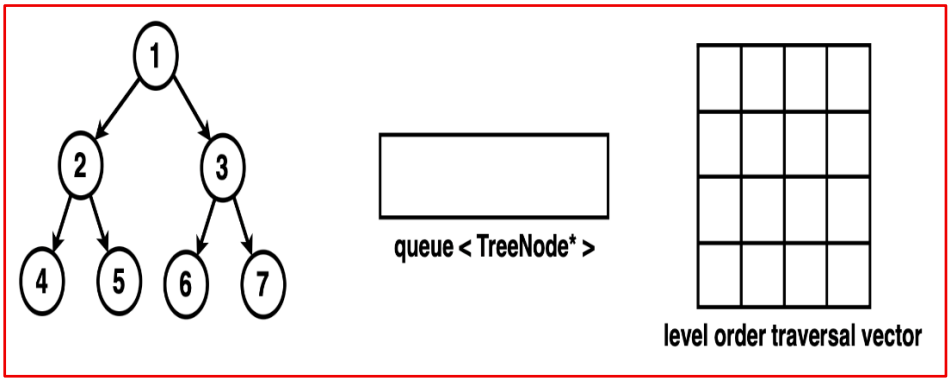
Space Complexity: O(N) (stack can grow up to height of tree in worst case).

Problem Statement: Given the root of a Binary Tree, returns an array containing the level order traversal of the tree.

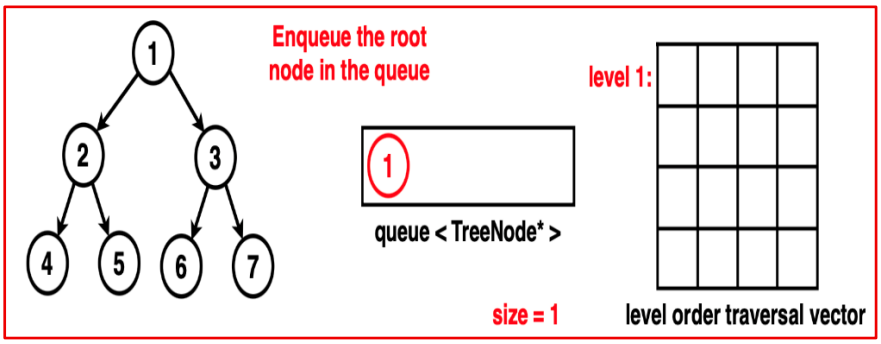
Optimal Approach

Algorithm:

Step 1: Initialise an empty queue data structure to store the nodes during traversal. Create a 2D Array-List of Integers to store the level order traversal. If the tree is empty, return this empty 2D list.



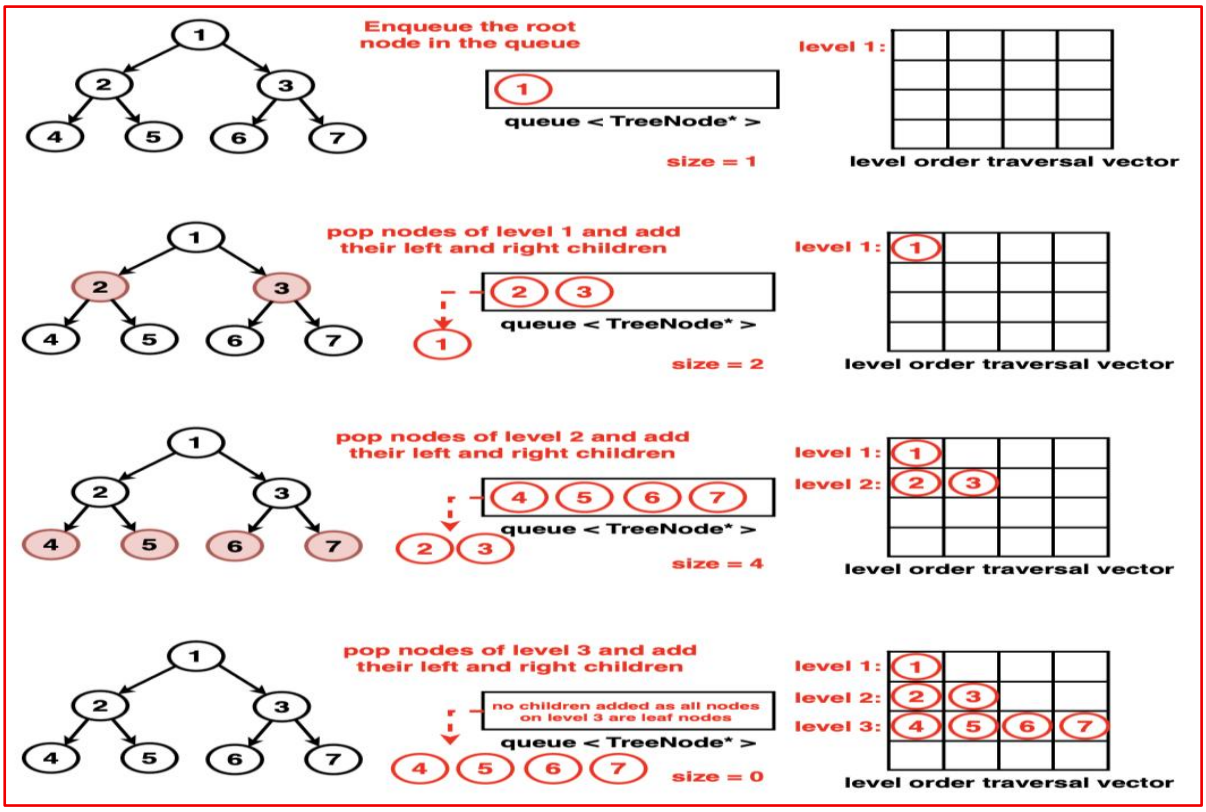
Step 2: Add (Enqueue) the root node of the binary tree to the queue.



Step 3: Iterate until the queue is empty:

- Get the current size of the queue. This size indicates the number of nodes at the current level.
- Create an Array-List 'level' to store the nodes at the current level.
- Iterate through 'size' number of nodes at the current level:
    - Remove (Pop) the front node from the queue.
    - Add the left and right child nodes of the current node (if they exist) into the queue.
    - Store the node's value in the 'level' Array-list.

- After processing all the nodes at the current level, add the ‘level’ Array-List to the ‘ans’ 2D Array-List, representing the current level.



Step 4: Once the traversal loop completes ie. all levels have been processed, return the ‘ans’ 2D Array-list containing the level-order traversal.

Code:

```java
static ArrayList<ArrayList<Integer>> optimal(Node root){  1 usage
    ArrayList<ArrayList<Integer>> res = new ArrayList<>();
    if (root == null) return res;

    Queue<Node> q = new LinkedList<>();
    q.add(root);
    while (!q.isEmpty()){
        int size = q.size();
        ArrayList<Integer> level = new ArrayList<>();

        for (int i=0;i<size;i++){
            Node temp = q.remove();
            if(temp.left != null) q.add(temp.left);
            if (temp.right != null) q.add(temp.right);
            level.add(temp.data);
        }
        res.add(level);
    }
    return res;
}
```

Time Complexity: O(N) where N is the number of nodes in the binary tree. Each node of the binary tree is enqueued and dequeued exactly once, hence all nodes need to be processed and visited.

Space Complexity: O(2N) where N is the number of nodes in the binary tree. In the worst case, the queue has to hold all the nodes of the last level of the binary tree. The resultant Array-list also stores the values of the nodes level by level and hence contains all the nodes of the tree contributing to O(N) space as well.

Problem Statement: Given the root of a Binary Tree, return the preorder, in-order and post-order traversal sequence of the given tree by making just one traversal.

Optimal Approach:

Algorithm:

Step 1: Create a 2D list for storing results.

Step 1: If the input root is null (i.e., the tree is empty), return 2D result list.

Step 3: Create three separate Array-lists to store nodes for each traversal type:

In-order list – will store nodes visited in in-order.

Pre-order list – will store nodes visited in pre-order.

Post-order list – will store nodes visited in post-order.

Step 3: Prepare the stack

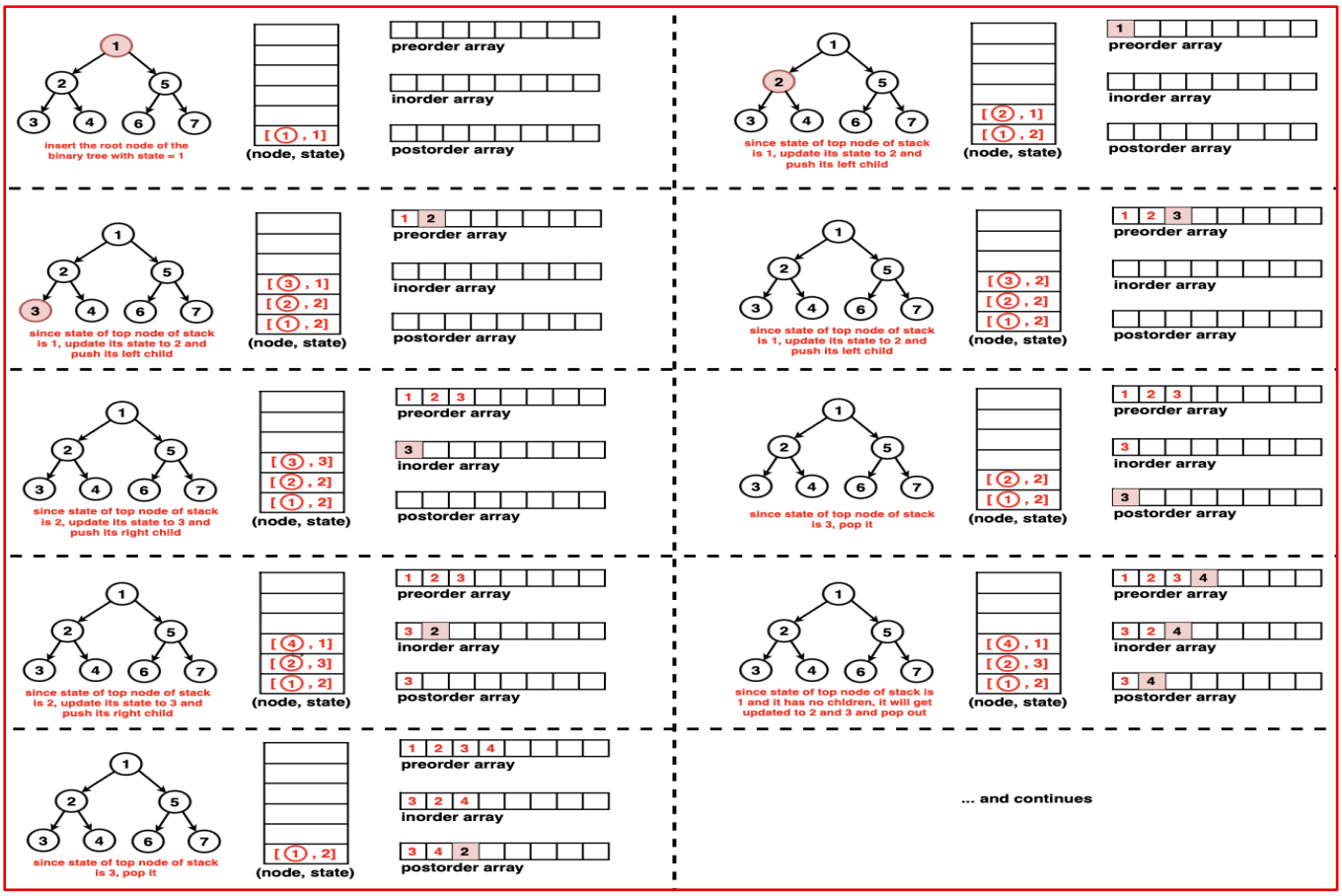
- Use a stack to simulate recursive traversal.
- Each element of the stack stores:
    - A reference to a **node**.
    - A **state** value indicating the progress for that node:
        - **State 1** → node is ready for pre-order processing
        - **State 2** → node is ready for in-order processing
        - **State 3** → node is ready for post-order processing
- Push the root node onto the stack with **state = 1**.

Step 4: Iterative traversal using the stack

- Loop while the stack is not empty:
    1. Pop the top element (node + state) from the stack.
    2. Preorder step (state = 1)
        - Add the node's value to the preorder list.
        - Increment the node's state to 2 and push it back onto the stack.
        - If the node has a left child, push the left child onto the stack with state = 1.
    3. In-order step (state = 2)
        - Add the node's value to the in-order list.
        - Increment the node's state to 3 and push it back onto the stack.
        - If the node has a right child, push the right child onto the stack with state = 1.
    4. Post-order step (state = 3)
        - Add the node's value to the post-order list.
        - Node is fully processed, so do not push it back.

Step 5: Add three list into 2D list.

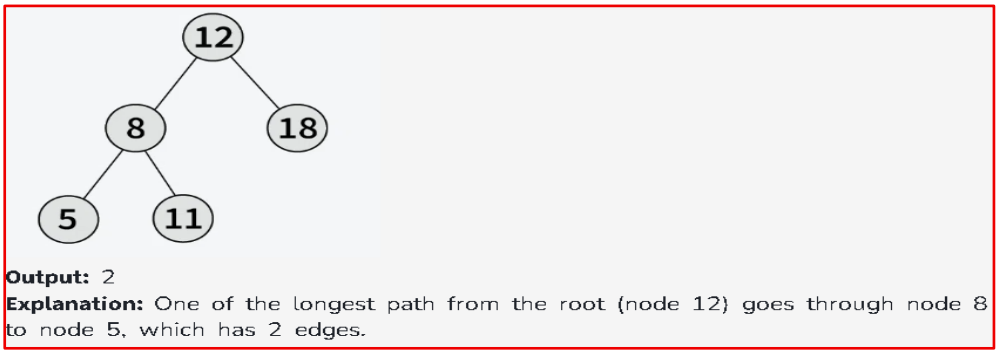
Step 6: Return the 2D result



Time Complexity: O(3N) where N is the number of nodes in the Binary Tree. Each node is processed once for each traversal type (pre-order, in-order, and post-order). Hence, the algorithm effectively visits each node three times in total across the three traversal types.

Space Complexity: O(4N) where N is the number of nodes in the Binary Tree. A stack to maintain traversal states, requiring additional space proportional to the maximum number of nodes in the stack at any point during traversal. Three Array-lists to store the preorder, in-order and post-order traversal.

Problem Statement: Given the root of a Binary Tree, return the height (Maximum depth) of the tree. The height of the tree is equal to the number of nodes on the longest path from root to a leaf.



Output: 2
Explanation: One of the longest path from the root (node 12) goes through node 8 to node 5, which has 2 edges.

Optimal Approach - 1: Using Recursion

Step 1: **Handle the base case**

- If the current node (root) is null, return **0**.
- This means: an empty tree has height 0 (no nodes, no depth).

Step 2: Recursive calls

- Recursively calculate the height of the **left subtree** and store it as lh.
- Recursively calculate the height of the **right subtree** and store it as rh.

Step 3: Compare subtree heights

- Since the height of a tree is determined by the **longest path**, take the maximum of lh and rh.

Step 4: Add current node's contribution

- Add **1** to the maximum of left and right heights to account for the **current node** (root of this subtree).

Step 5: Return the result

- The value returned represents the **height (or maximum depth)** of the subtree rooted at this node.
- When the recursion unwinds back to the original root, the final value is the **height of the entire tree**.

Code:

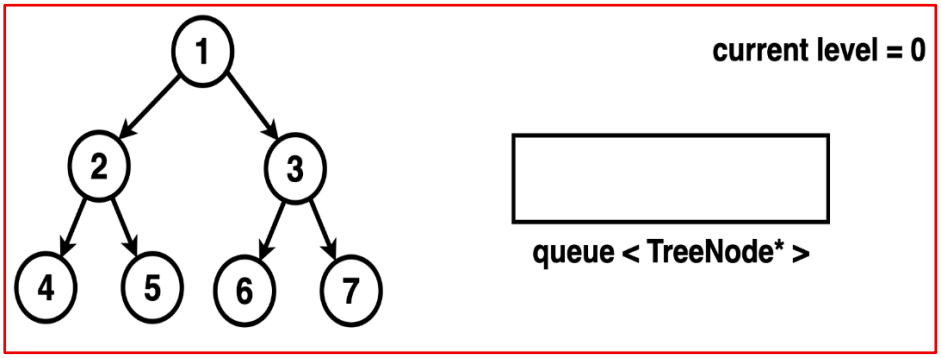
```
static int recursively(Node root){  3 usages
    if (root == null) return 0;

    int lh = recursively(root.left);
    int rh = recursively(root.right);
    return 1 + Math.max(lh,rh);
}
```

Time Complexity = O(N) where N = number of nodes in the binary tree. Because each node is visited exactly once.

Space Complexity = O(H) where H = height of the tree (worst-case N, best-case log N).

Optimal Approach - 2: Iteratively using Queue

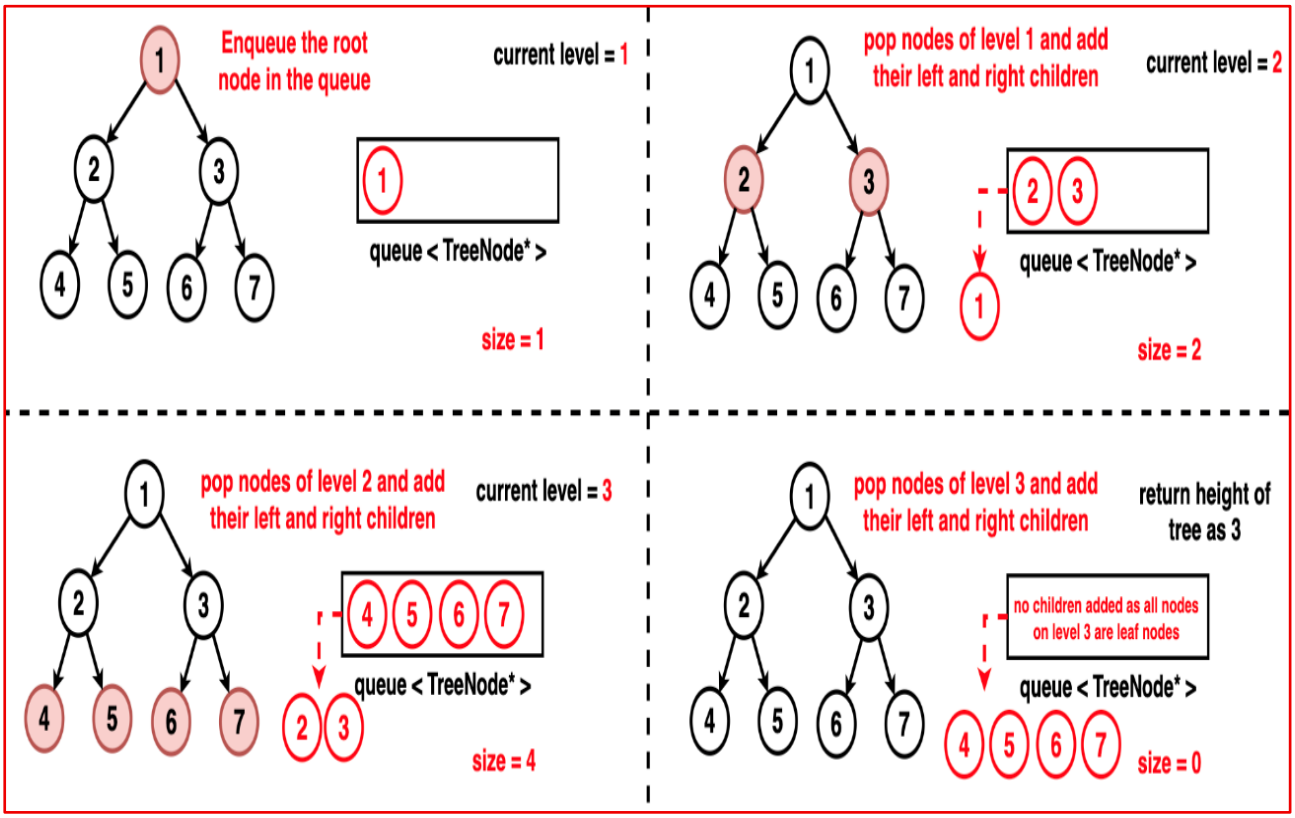
Step 1: Initialise a queue for level order traversal and a variable `level` to track the depth. Check, if the root is null, if so, return the answer as 0 indicating an empty tree.



Step 2: Insert the root of the Binary Tree into the queue and set the level as 0.

Step 3: Begin a loop that continues until the queue becomes empty where at each level:

- Increment `level` by 1, indicating we are moving to the next level.
- Determine the number of nodes at the current level by storing the size of the queue.
- Iterate over the number of nodes equal to the size of the queue and at each node, Pop it from front of the queue and push its left and right children (if they exist).



Step 4: After the queue loop gets over, return the `level` variable representing the maximum depth of the tree calculated during the level order traversal.

Code:

```
static int iteratively(Node root){  1 usage
    if (root == null) return 0;

    int level = 0;
    Queue<Node> q = new LinkedList<>();
    q.add(root);
    while (!q.isEmpty()){
        int size = q.size();
        level += 1;
        for (int i=0;i<size;i++) {
            Node temp = q.remove();
            if (temp.left != null) q.add(temp.left);
            if (temp.right != null) q.add(temp.right);
        }
    }
    return level;
}
```

Time Complexity: O(N) where N is the number of nodes in the Binary Tree. This complexity arises from visiting each node exactly once during the traversal.

Space Complexity: O(N) where N is the number of nodes in the Binary Tree because in the worst-case scenario the tree is balanced and has N/2 nodes in its last level which will have to be stored in the queue.

Problem Statement: Given a binary tree, find its minimum depth. The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

Optimal Approach – 1: Using recursion

Step 1: Handle the empty tree

- If the current node is null, return **0**.

Step 2: Handle single-child nodes

- If the left child is null but the right child exists:
    - Ignore the left (null) path because it does not lead to a leaf.
    - Recursively compute the minimum depth from the right child and add 1 for the current node.
- If the right child is null but the left child exists:

- Ignore the right (null) path.
- Recursively compute the minimum depth from the left child and add 1.

Step 3: Handle nodes with both children

- If the node has both left and right children:
  - Recursively compute the minimum depth of the left subtree.
  - Recursively compute the minimum depth of the right subtree.
  - Take the minimum of the two depths and add 1 for the current node.

Step 4: Return result

- Each recursive call returns the minimum depth for that subtree.
- When the recursion unwinds back to the root, the final return value is the minimum depth of the entire tree.

Code:

```
static int recursively(Node root){  5 usages
    if (root == null) return 0;
    if (root.left == null) return 1 + recursively(root.right);
    if (root.right == null) return 1 + recursively(root.left);
    return 1 + Math.min(recursively(root.left), recursively(root.right));
}
```

Time Complexity = O(N) where N = number of nodes in the binary tree. Because each node is visited exactly once.

Space Complexity = O(H) where H = height of the tree (worst-case N, best-case log N).

Optimal Approach – 1: Iteratively using Queue

Step 1: Handle empty tree

- If the root node is null, return 0, because an empty tree has minimum depth 0.

Step 2: Initialize BFS traversal

- Create a queue to perform level-order traversal.
- Insert the root node into the queue.
- Initialize a level counter to 0. This will track the current depth.

Step 3: Traverse the tree level by level

- While the queue is not empty:
    1. Determine the number of nodes at the current level (size = queue.size()).
    2. Increment the level counter by 1.
    3. Process each node at this level:
        - Remove the node from the queue.
        - Check if it's a leaf node (both left and right are null).
            - If it is a leaf → return the current level. This is the minimum depth.
        - If left child exists → enqueue left child.
        - If right child exists → enqueue right child.

Step 4: Continue until a leaf is found

- The BFS ensures that the first leaf encountered is at the minimum depth, because BFS explores level by level.
- Once a leaf is found, the function immediately returns the depth.

Step 5: Return the result

- The returned level is the minimum depth of the tree.

Code:

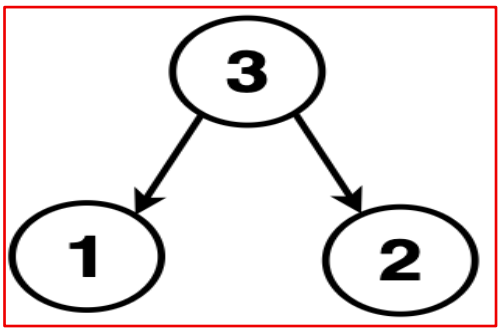
```
static int iteratively(Node root){  1 usage
    if (root == null) return 0;

    int level = 0;
    Queue<Node> q = new LinkedList<>();
    q.add(root);
    while (!q.isEmpty()){
        int size = q.size();
        level += 1;
        for (int i=0;i<size;i++) {
            Node temp = q.remove();

            if (temp.left == null && temp.right == null) return level;
            if (temp.left != null) q.add(temp.left);
            if (temp.right != null) q.add(temp.right);
        }
    }
    return level;
}
```

Time Complexity: O(N) where N is the number of nodes in the Binary Tree. This complexity arises from visiting each node exactly once during the traversal.

Space Complexity: O(N) where N is the number of nodes in the Binary Tree because in the worst-case scenario the tree is balanced and has N/2 nodes in its last level which will have to be stored in the queue.

Problem Statement: Given a Binary Tree, return true if it is a Balanced Binary Tree else return false. A Binary Tree is balanced if, for all nodes in the tree, the difference between left and right subtree height is not more than 1.

Input: 3 9 20 -1 -1 15 7



Output: True.

Explanation: The difference in the height of left and right subtree is 1 hence the tree is balanced.

Brute-Force:

Step – 1: Base Case:

- If the current node is null (i.e., the tree or subtree is empty), consider it balanced.

Step – 2: Compute Heights:

- Calculate the height of the left subtree.
- Calculate the height of the right subtree.

Step – 2: Check Balance at Current Node:

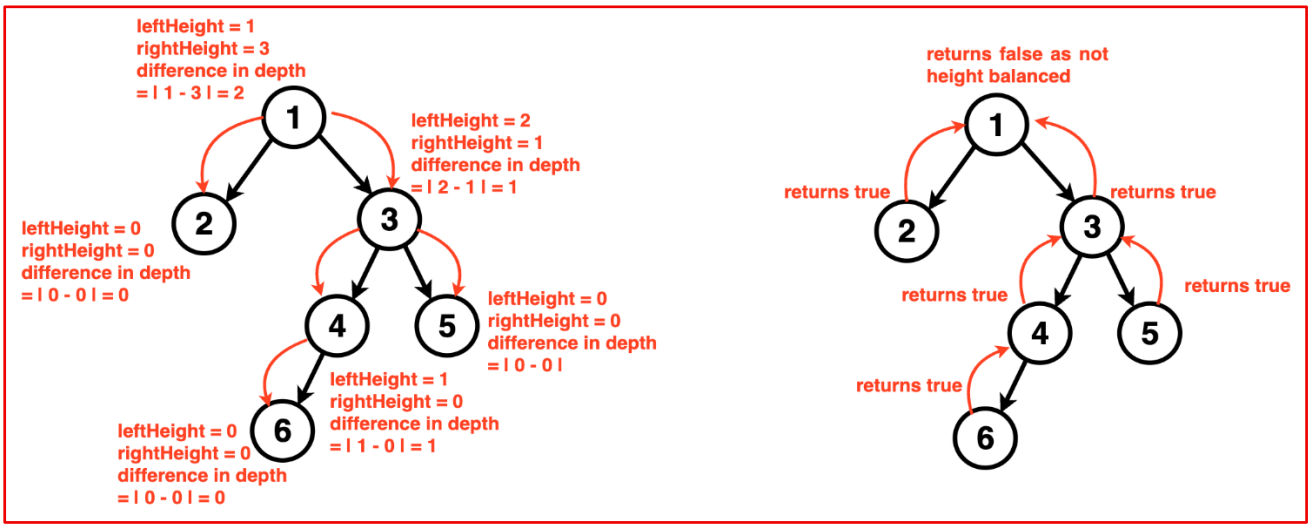
- Find the difference between the heights of the left and right subtrees.
- If the difference is greater than 1, the tree is unbalanced.

Step – 2: Check Subtrees Recursively:

- Recursively check whether the left subtree is balanced.
- Recursively check whether the right subtree is balanced.

Step – 2: Return Result:

- If both subtrees are balanced and the current node's height difference is ≤ 1, the tree is balanced.
- Otherwise, it is unbalanced.



Code:

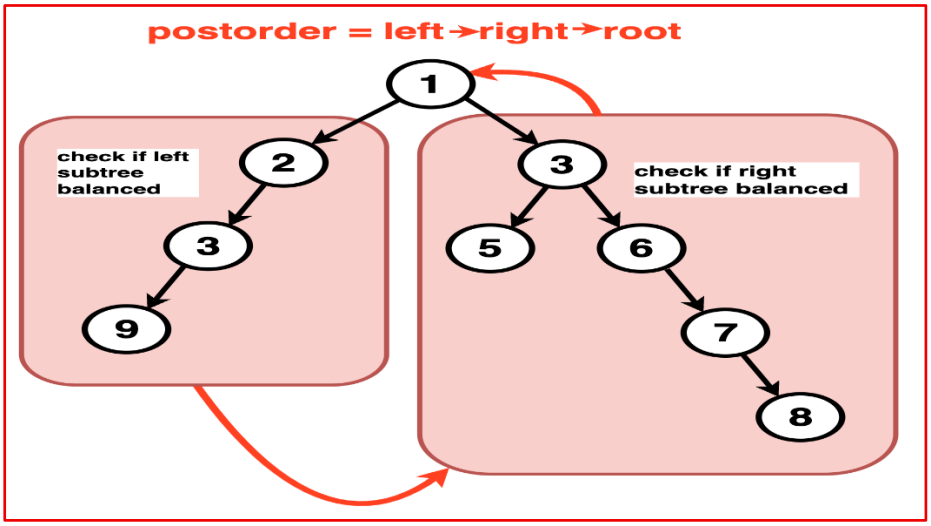
```
static int getHeight(Node root){  4 usages
    if (root == null) return 0;

    int lh = getHeight(root.left);
    int rh = getHeight(root.right);
    return 1 + Math.max(lh,rh);
}
static boolean bruteForce(Node root){  3 usages
    if (root == null) return true;

    int lh = getHeight(root.left);
    int rh = getHeight(root.right);

    if (Math.abs(lh-rh) > 1) return false;

    boolean left = bruteForce(root.left);
    boolean right = bruteForce(root.right);

    if (!left || !right) return false;
    return true;
}
```

Time Complexity: O(N^2) where N is the number of nodes in the Binary Tree. This arises as we calculate the height of each node and to calculate the height for each node, we traverse the tree which is proportional to the number of nodes. Since this calculation is performed for each node in the tree, the complexity becomes: O (N x N) ~ O(N^2).

Space Complexity: O(h) Space is used by the recursion stack. For a tree of height (h), the recursion stack can go up to h. In the worst case (skewed tree), h = n.

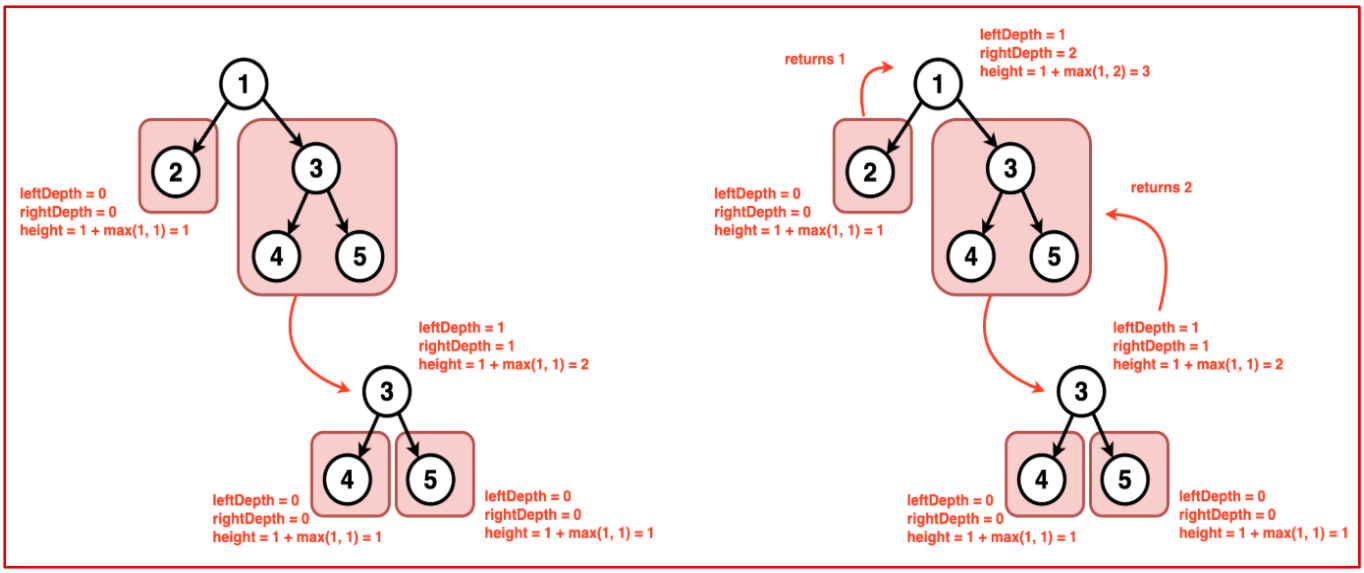
Optimal:

Step 1: Traverse the Binary Tree is post-order manner using recursion. Visit left subtree, then right subtree, and finally the root node.



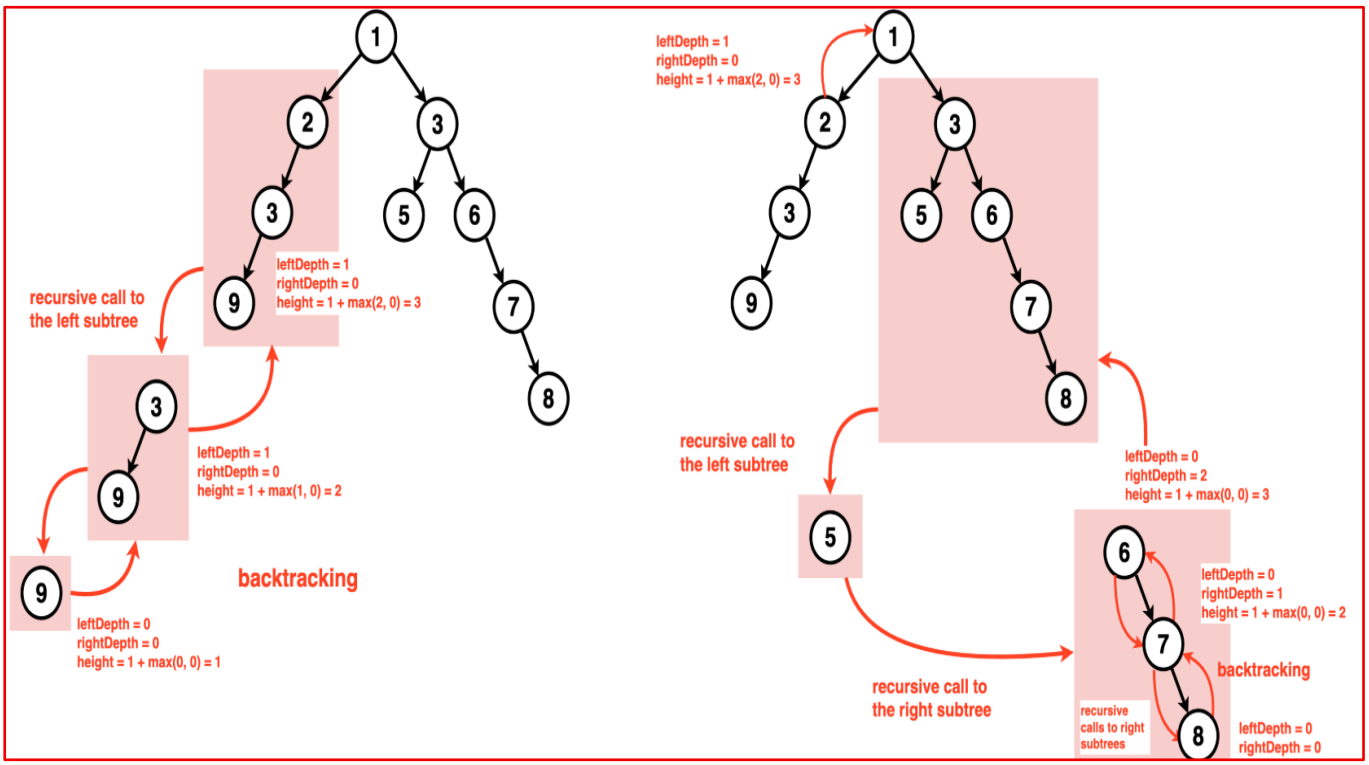
Step 2: During the traversal, for each node, calculate the heights of the its left and right subtrees. Use the obtained subtree heights to validate the balance conditions for the current node.

Step 3: At each node, if the absolute difference between the heights of the left and right subtrees is greater than 1 or if any subtree is unbalanced (returns -1), return -1 immediately indicating an unbalanced tree.



Step 4: If the tree is balanced, return the height of the current node by considering the maximum height of its left and right subtree plus 1 accounting for the current node.

Step 5: Complete the traversal until all nodes are visited and return the final result - either the height of the entire tree if balanced or -1 if unbalanced.



Code:

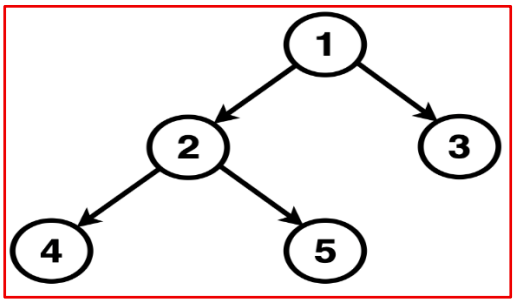
```java
static int optimal(Node root){  3 usages
    if (root==null) return 0;

    int lh = optimal(root.left);
    if(lh == -1) return -1;

    int rh = optimal(root.right);
    if(rh == -1) return -1;

    if (Math.abs(lh-rh) > 1) return -1;
    return 1+Math.max(lh,rh);
}
```

Time Complexity = O(N) where N = number of nodes in the binary tree. Because each node is visited exactly once.

Space Complexity = O(H) where H = height of the tree (worst-case N, best-case log N).

Problem Statement: Given the root of the Binary Tree, return the length of its diameter. The Diameter of a Binary Tree is the longest distance between any two nodes of that tree. This path may or may not pass through the root.

Example:

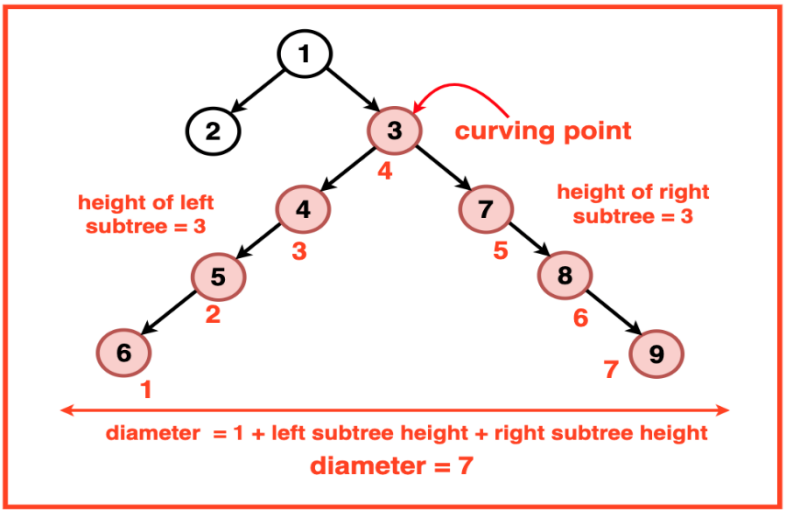


Output: 4

Explanation: The distance between the leftmost node 4 and the rightmost node 3 is 4, since this is the longest horizontal distance of the binary tree, hence its diameter.

Brute-Force:

To find the diameter of a binary tree, we can think of every node as a potential `Curving Point` of the diameter path. This Curving Point is the node along the diameter path that holds the maximum height and from where the path curves uphill and downhill.

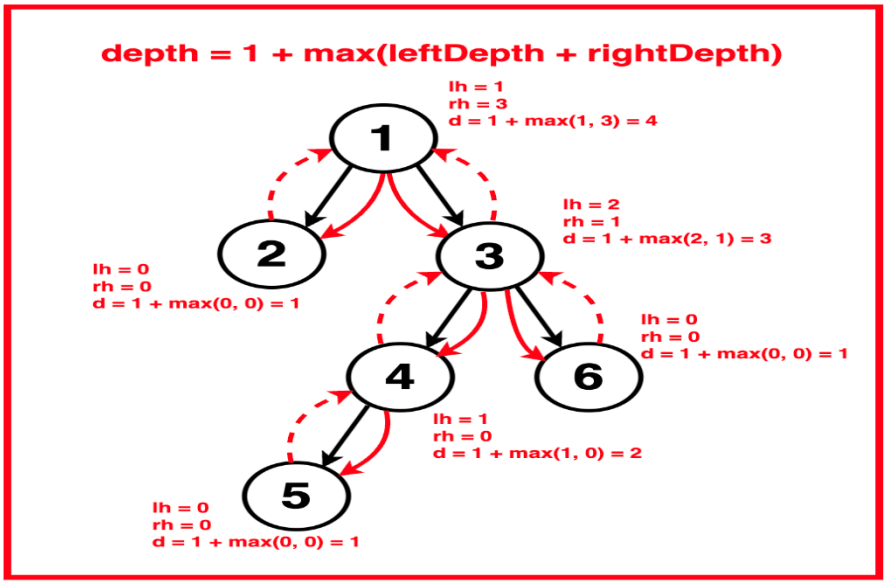


Hence, we can see that the diameter at a specific curving point is determined by adding the height of the left subtree to the height of the right subtree and adding 1 to account for the level of the curving point. Diameter = 1 + Left Subtree Height + Right Subtree Height Therefore, we can traverse the tree recursively considering each node as a potential Curving Point and calculate the height of the left and right subtrees at each node. This will give us the diameter for the current Curving Point. Throughout the traversal, we track the maximum diameter encountered and return it as the overall diameter of the Binary Tree.

Algorithm:

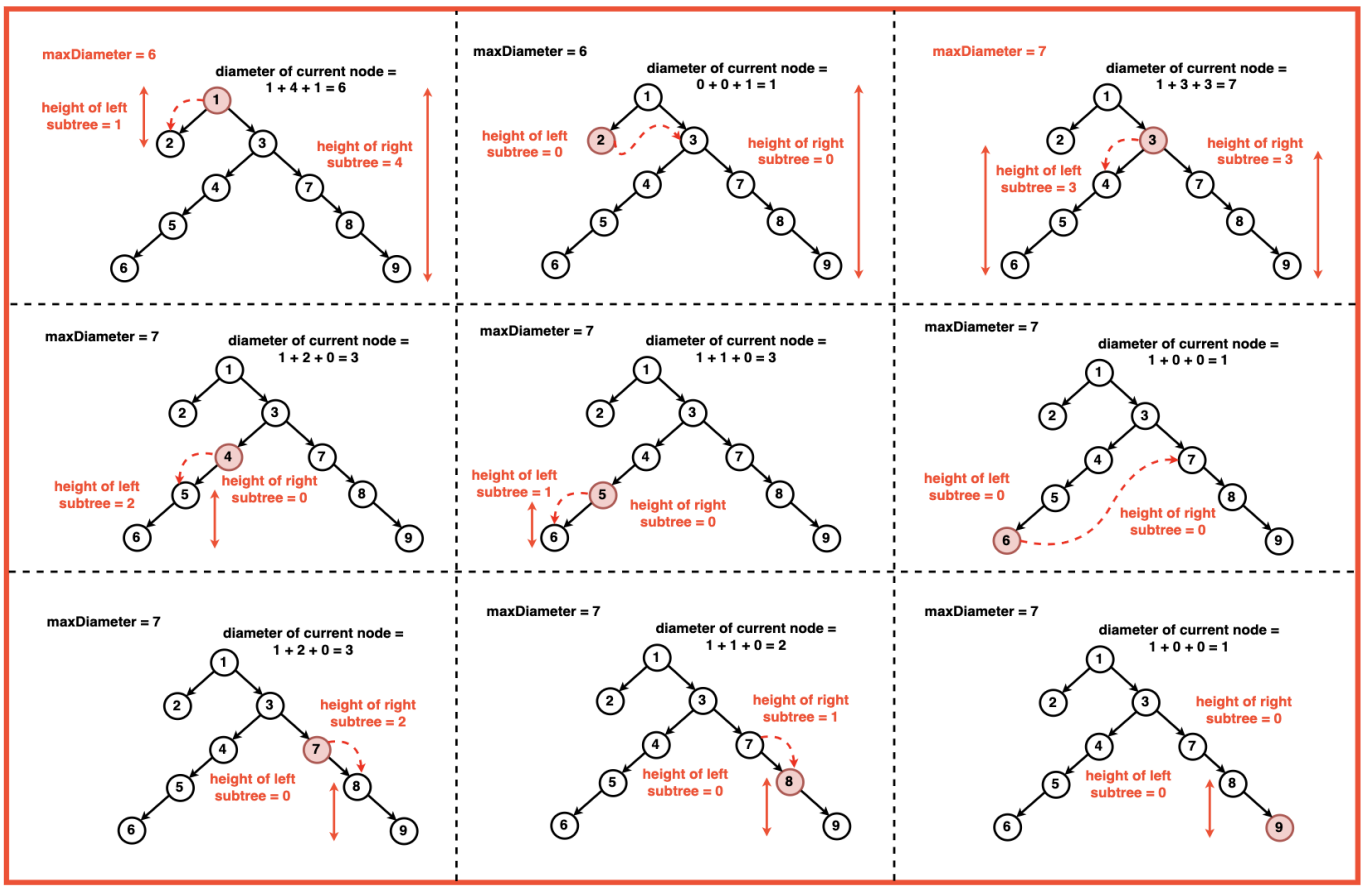
Step 1: Create a global variable `diameter` to store the maximum diameter encountered. At every node, we will take the maximum of this variable and the current diameter to update it to be the maximum amongst all.

Step 2: Define the recursive function calculate Height that takes in a node as an argument and calculate its [height in the Binary Tree.]



Step 3: Recursively start traversing from the root, consider the current node to be a potential Curving Point and for each node:

- Recursively calculate the height of its left and right subtrees.
- Compute the diameter at the current node by summing heights of the left and right subtrees.
- Update the global variable diameter as the max of the current diameter and the largest diameter encountered so far.



Step 4: Return the maximum diameter found during traversal as the result.

🔑 Intuition Behind lh + rh

At every node, you ask:

👉 *"What is the longest path that passes through this node?"*

- Height of left subtree = longest path downwards in the left direction.
- Height of right subtree = longest path downwards in the right direction.

So, the longest path through the current node =

leftHeight + rightHeight

This covers the case where the diameter passes through the current node.

Why Not More Than lh + rh?

Because any path that goes through the current node must go down the left side + down the right side.

- If you tried to include more than lh + rh, you'd be double-counting nodes.
- Example: If lh = 3 and rh = 2, the longest path through the current node is 3 + 2 = 5.

Code:

```
static int diameter1 = 0;  4 usages
static int findHeight(Node root){  4 usages
    if (root == null) return 0;
    int lh = findHeight(root.left);
    int rh = findHeight(root.right);
    return 1+Math.max(lh,rh);
}
static int bruteForce(Node root){  3 usages
    if (root == null) return 0;
    int lh = findHeight(root.left);
    int rh = findHeight(root.right);
    diameter1 = Math.max(diameter1,lh+rh);

    bruteForce(root.left);
    bruteForce(root.right);

    return diameter1;
}
```

Time Complexity: O(N*N) where N is the number of nodes in the Binary Tree.

- This arises as we calculate the diameter of each node and to calculate the height of its left and right children, we traverse the tree which is proportional to the number of nodes.
- Since this calculation is performed for each node in the tree, the complexity becomes: O (N x N) ~ O(N2).

Space Complexity: O (1) as no additional data structures or memory is allocated O(H): Recursive Stack Space is used to calculate the height of the tree at each node which is proportional to the height of the tree. The recursive nature of the getHeight function, which incurs space on the call stack for each recursive call until it reaches the leaf nodes or the height of the tree.

Optimal:

The O(N2) time complexity of the previous approach can be optimised by eliminating the steps of repeatedly calculating subtree heights. The heights of the left and right subtrees are computed multiple times for each node, which leads to redundant calculations. Instead, we can compute these heights in a bottom-up manner. The Post-order method allows us to validate balance conditions efficiently during the traversal.

Algorithm:

Step-1: Base Case

- If the current node is null, return height = 0.
- This indicates that an empty subtree contributes no height to the diameter.

Step-2: Recursively Compute Subtree Heights

- Compute the height of the left subtree.
- Compute the height of the right subtree.

Step-3: Update Diameter

- The longest path that passes through the current node = left-Height + right-Height.
- Update the global ‘diameter’ if this path is longer than the current maximum.

Step-4: Return Height of Current Node

- Height of the current node = 1 + max (left-Height, right-Height).
- This height is used by the parent node to compute its diameter.

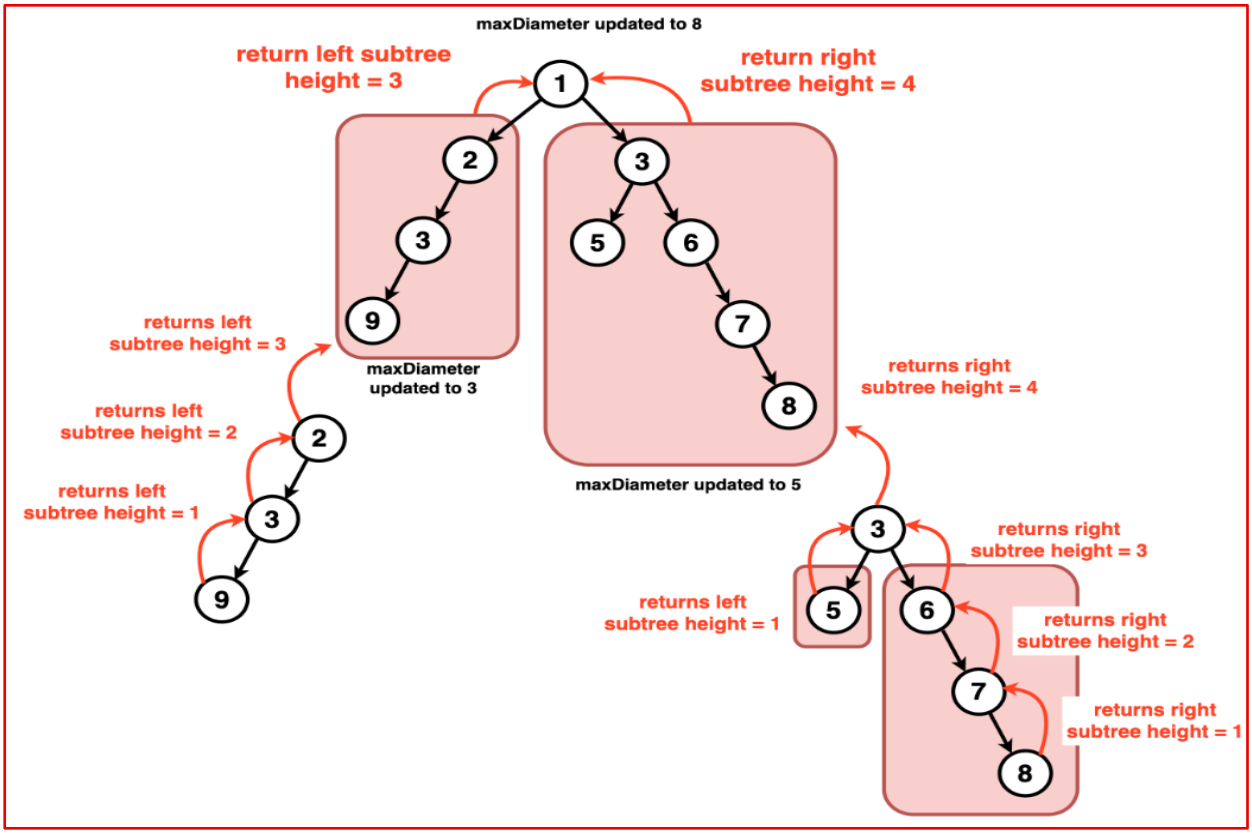
Step-5: Initialize and Call Helper Function

- Reset diameter = 0 before starting.
- Call the helper function on the root node.

Step-6: Return Final Diameter

- After traversal, diameter holds the length of the longest path in the tree.

- Return diameter as the final answer.



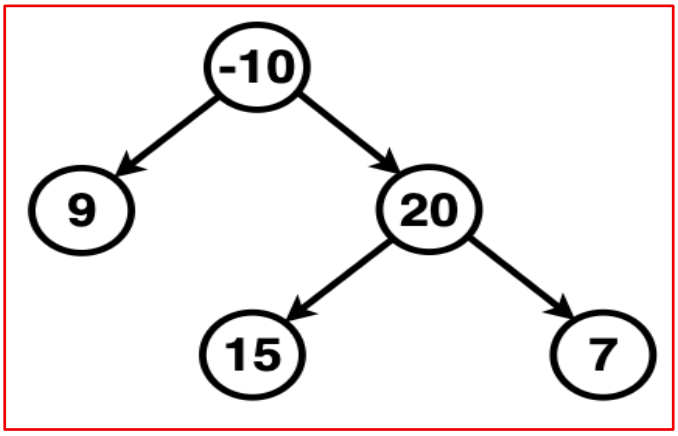
Code:

```
class Solution {
    static int diameter = 0;
    static int optimal(TreeNode root){
        if (root == null) return 0;
        int lh = optimal(root.left);
        int rh = optimal(root.right);
        diameter = Math.max(diameter,lh+rh);
        return 1+ Math.max(lh,rh);
    }

    public int diameterOfBinaryTree(TreeNode root) {
        diameter = 0;
        optimal(root);
        return diameter;
    }
}
```

Time Complexity: O(n) → Each node is visited exactly once, where n = Total number of nodes.

Space Complexity: O(h) → Recursion stack, where h is the height of the tree. Worst case (skewed tree): O(n) Best case (balanced tree): O (log n)

Problem Statement: Given a Binary Tree, determine the maximum sum achievable along any path within the tree. A path in a binary tree is defined as a sequence of nodes where each pair of adjacent nodes is connected by an edge. Nodes can only appear once in the sequence, and the path is not required to start from the root. Identify and compute the maximum sum possible along any path within the given binary tree.



Output: 42

Explanation: Out of all the paths possible in the Binary Tree, 15 -> 20 -> 7 has the greatest sum ie. 42.

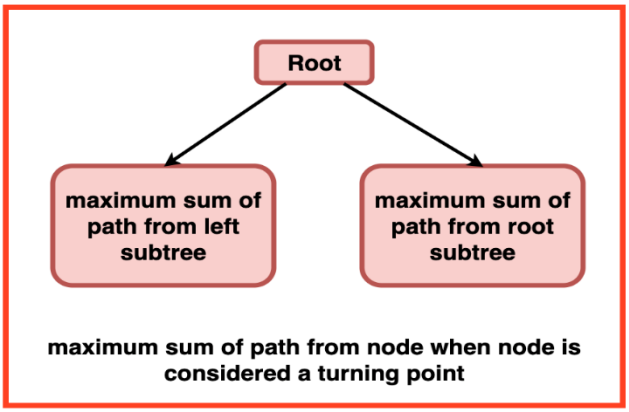
Brute-force:

The brute-force way is “try every node as the turning point”.

- Pick a node N.
- Find the max path sum going down the left (maxDownPath(N.left)).
- Find the max path sum going down the right (maxDownPath(N.right)).
- Add N.val + left + right → this is the max path sum passing through N.
- Then do the same for the left subtree and right subtree recursively, because maybe the max path is entirely inside the left or right subtree.
- Take the maximum of these three.

Optimal:

To find the max path sum of a binary tree, we can think of every node as a potential `Curving Point` of the path along which we find the sum. The maximum sum of a path through a turning point (like a curve) can be found by adding the maximum sum achievable in the left subtree, the right subtree, and the value of the turning point.



We can recursively traverse the tree, considering each node as a potential turning point and storing the maximum value (our final answer) in a reference variable. The recursive function should be defined in such a way that we consider both the possibilities:

- When the current node is the turning point and in this scenario we calculate the maximum path sum taking into sum contributions from both the left and right subtrees along with the value of the current node.
- When the current node is not the turning point, we consider only the left or the right subtree. The maximum of the two is returned as the maximum path sum of that subtree.

Algorithm:

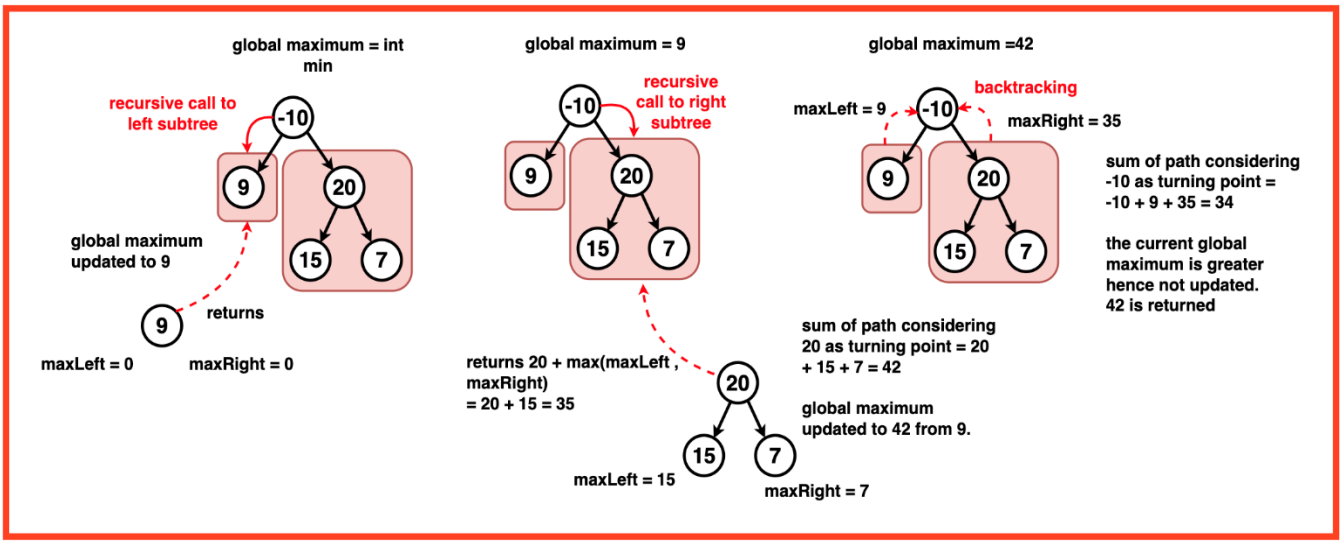
Step 1: Initialise the variable ‘maxSum’ to the minimum possible integer value. This ensures that the algorithm correctly updates maxi with the first encountered valid path sum (even if its negative) and subsequently updates it whenever a larger path sum is found.

Step 2: Call the recursive function `optimal ` with the root of the binary tree.

Step 3: Base case: If the current node is null, return 0.

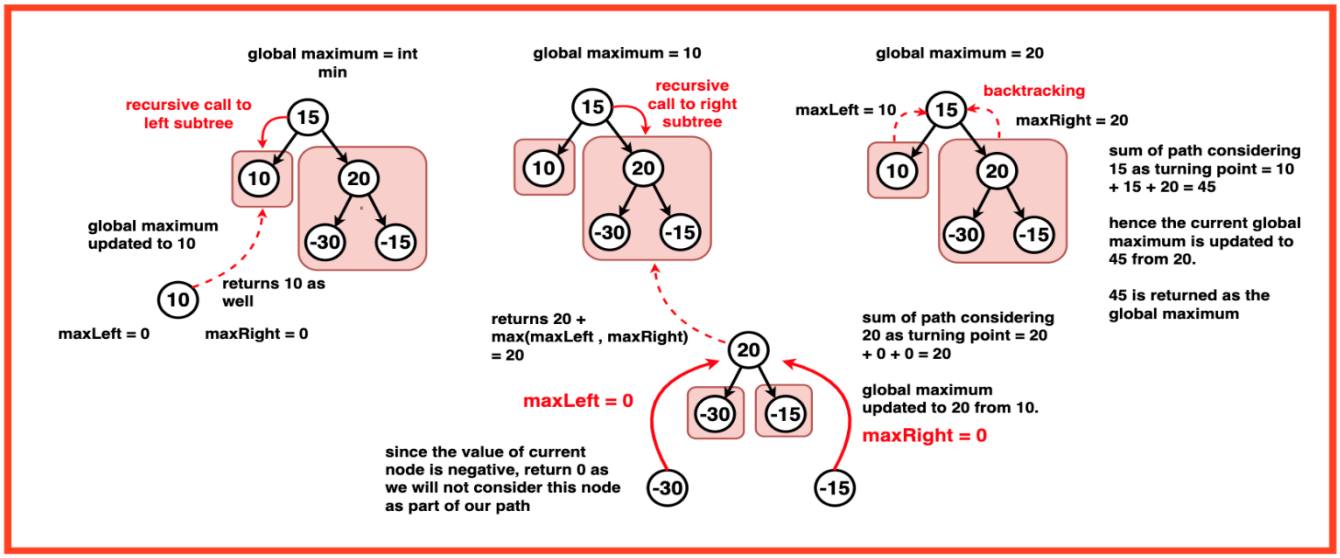
Step 4: Calculate the maximum path sum for the left and right subtree using recursion.

Step 5: Update the overall maximum path sum (maxSum) by considering the sum of the left and right subtree paths plus the current node's value. This represents the sum of the path that includes the current node. This sum is used to update the overall maximum path sum (maxSum) when the current node serves as the turning point in the path.



Step 6: Return the maximum sum considering only one branch (either left or right) along with the current node. This represents the maximum sum considering only one branch (either left or right) along with the current node. This value is returned by the recursive function to contribute to the calculation of the maximum path sum in the binary tree.

Case Considering Negative Leaf Nodes:



Code:

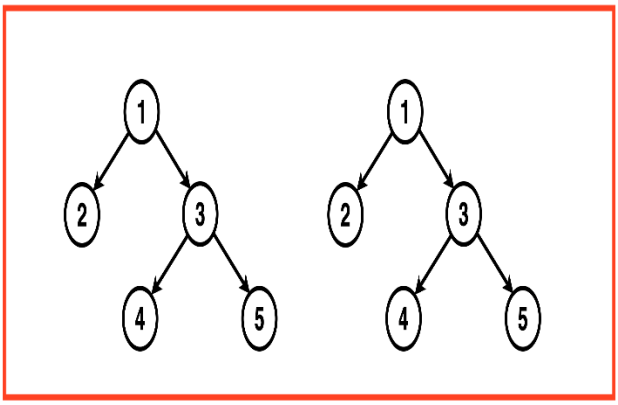
```java
class Solution {
    static int maxSum = 0;
    static int optimal(TreeNode root){
        if (root == null) return 0;

        int leftSum = optimal(root.left);
        if(leftSum<0) leftSum = 0;

        int rightSum = optimal(root.right);
        if(rightSum<0) rightSum = 0;

        maxSum = Math.max(maxSum,leftSum+rightSum+root.val);
        return root.val + Math.max(leftSum,rightSum);
    }
    public int maxPathSum(TreeNode root) {
        maxSum = Integer.MIN_VALUE;
        optimal(root);
        return maxSum;
    }
}
```

Time Complexity: O(N) where N is the number of nodes in the Binary Tree. This complexity arises from visiting each node exactly once during the recursive traversal.

Space Complexity: O(1) as no additional space or data structures is created that is proportional to the input size of the tree. O(H) Recursive Stack Auxiliary Space : The recursion stack space is determined by the maximum depth of the recursion, which is the height of the binary tree denoted as H. In the balanced case it is log2N and in the worst case its N.

Given the roots of two binary trees p and q, write a function to check if they are the same or not. Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.

Input: Binary Tree 1: 1 2 3 -1 -1 4 5, Binary Tree 2: 1 2 3 -1 -1 4 5



Output: True, these trees are identical.

Explanation: Two trees are said to be identical if these three conditions are met for every pair of nodes:

- Value of a node in the first tree is equal to the value of the corresponding node in the second tree.
- Left subtree of this node is identical to the left subtree of the corresponding node.
- Right subtree of this node is identical to the right subtree of the corresponding node.

Optimal:

To determine if two binary trees are identical, we can follow a recursive approach. We traverse both trees in the preorder manner. We can traverse it by using any tree traversals.

The idea is to traverse both trees simultaneously, comparing the values of corresponding nodes at each step. We need to ensure that the left subtree of each node in the first tree is identical to the left subtree of the corresponding node in the second tree, and similarly for the right subtrees.

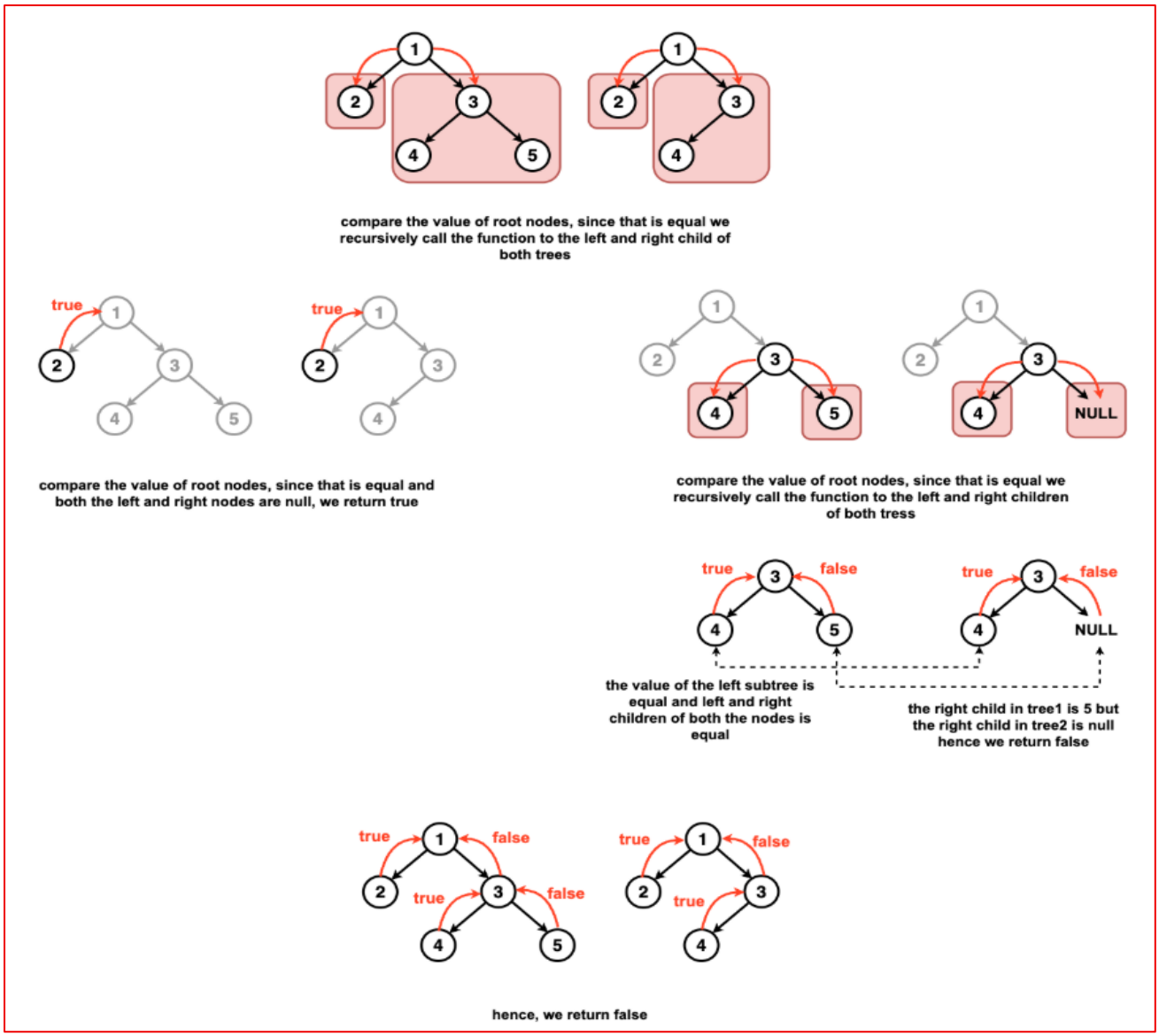
Algorithm:

1. Start at the root nodes of both trees (root1 and root2).
2. **Check base cases:**
   - If both nodes are null, return true → empty trees are identical.
   - If one node is null and the other is not, return false → trees are not identical.
3. **Check the current nodes' values:**
   - Store the result in a boolean variable.
4. **Recursively check the left subtrees:**
   - Store the result in a boolean variable, e.g., left
5. **Recursively check the right subtrees:**
   - Store the result in a boolean variable, e.g., right
6. **Combine the results:**
   - Return value && left && right
   - This ensures the trees are identical only if the current node matches and left subtree matches and right subtree matches.

Code:

```
static boolean optimal(Node root1,Node root2){  3 usages
    if (root1 == null && root2 == null) return true;
    if (root1 == null || root2 == null) return false;

    boolean value = (root1.data == root2.data);
    boolean left = optimal(root1.left,root2.left);
    boolean right = optimal(root1.right,root2.right);
    return value && left && right;
}
```

Dry-run:

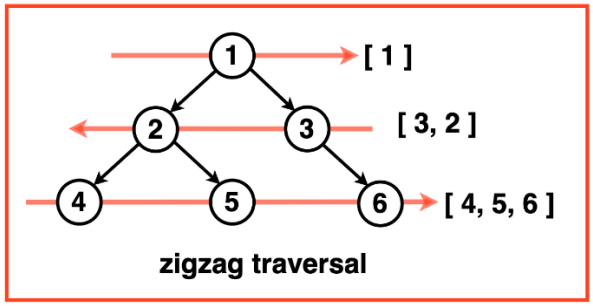


Time Complexity: O(N+M) where N is the number of nodes in the first Binary Tree and M is the number of nodes in the second Binary Tree. This complexity arises from visiting each node of the two binary nodes during their comparison.

Space Complexity: O(1) as no additional space or data structures is created that is proportional to the input size of the tree. O(H) Recursive Stack Auxiliary Space: The recursion stack space is determined by the maximum depth of the recursion, which is the height of the binary tree denoted as H. In the balanced case it is log2N and in the worst case (its N).

Given a binary tree with n nodes. You have to find the zig-zag level order traversal of the binary tree. In zig zag traversal starting from the first level go from left to right for odd-numbered levels and right to left for even-numbered levels.

Input: Binary Tree: 1 2 3 4 5 -1 6



Output: [[1],[3, 2],[4, 5, 6]]

Explanation:

- Level 1 (Root): Visit the root node 1 from left to right. Zigzag Traversal: [1]
- Level 2: Visit nodes at this level in a right-to-left order. Zigzag Traversal: [1], [3, 2]
- Level 3: Visit nodes at this level in a left-to-right order. Zigzag Traversal: [1], [3, 2], [4, 5, 6]

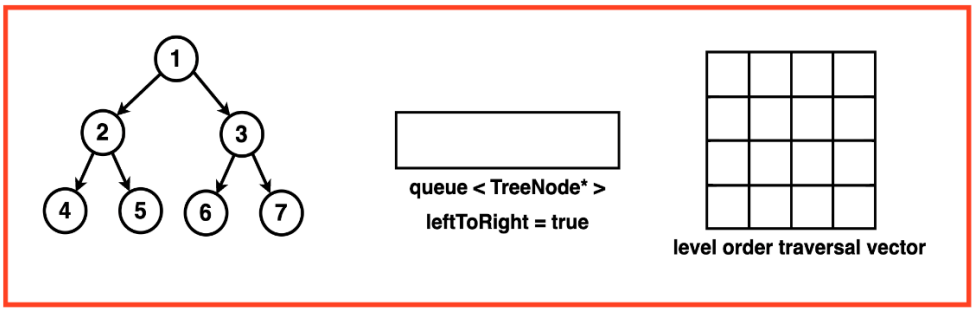
Optimal:

Zigzag traversal is a modification of the traditional level order traversal in a binary tree. Level Order Traversal explores does at each level from left or right but zigzag traversal adds a twist by alternating the direction of exploration. At odd levels, we proceed from left to right but for even levels the order is reversed, from right to left. This is achieved by introducing a `leftToRight` flag which controls the order in which nodes are processed at each level. When `leftToRight` is true, nodes are inserted into the level vector from left to right and when its false, nodes are inserted right to left.

Algorithm:

Step 1: Initialise an empty queue data structure to store the nodes during traversal. Create a 2D arrayList to store the level order traversal. If the tree is empty, return this empty 2D list.

Step 2: Create a `leftToRight` flag to keep track of the direction of traversal. When `leftToRight` is true, nodes are inserted into the level vector from left to right and when its false, nodes are inserted right to left.



Step 3: Enqueue (Add) the root node.

Step 4: Iterate until the queue is empty:

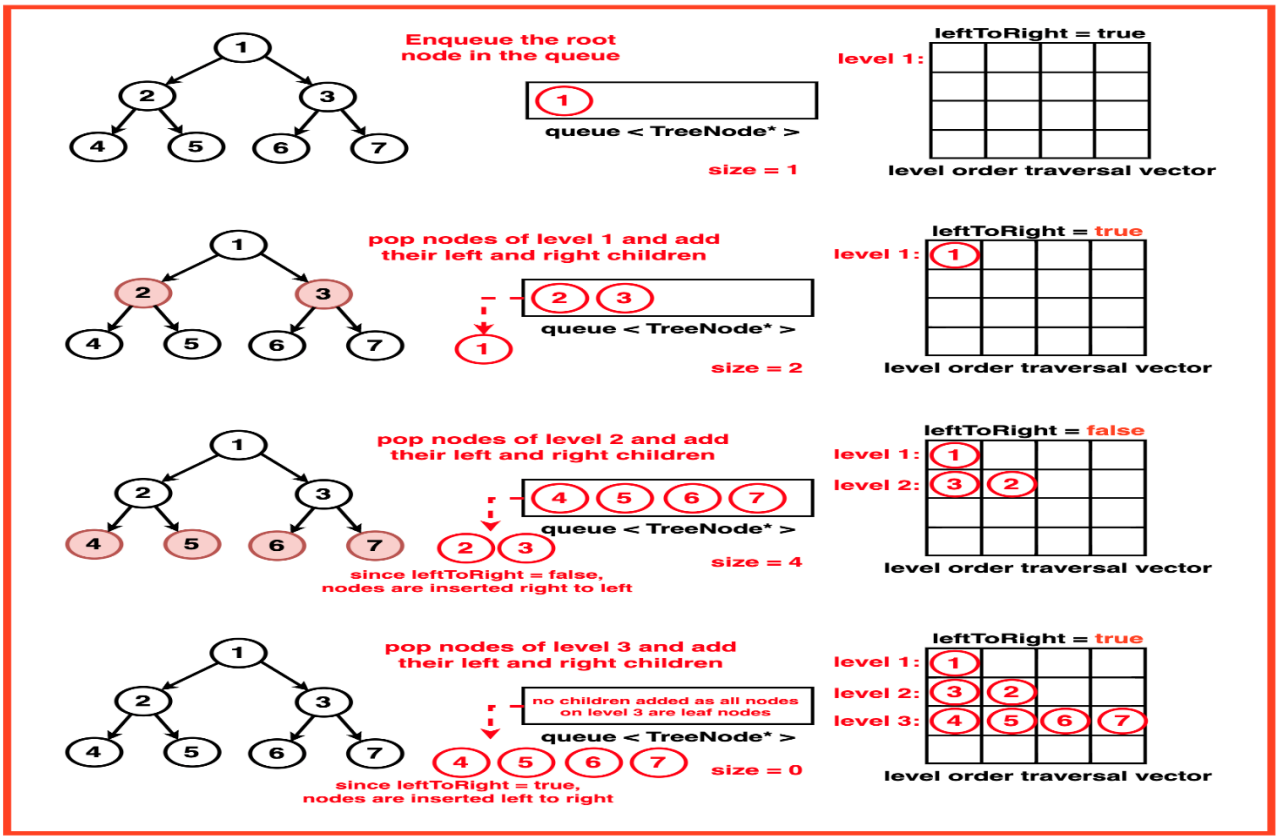
- Get the current size of the queue. This size indicates the number of nodes at the current level.
- Create an array ‘row’ to store the nodes at the current level.

Step 5: Iterate through ‘size’ number of nodes at the current level:

- Pop the front node from the queue.
- Store the node’s value in the ‘row’ array. Determine the index to insert the node’s value based on the traversal direction ‘leftToRight’.

If ‘leftToRight’ is true, the index is set to ‘i’ which means the node’s value will be inserted form left to right. If ‘rightToLeft’ is false, the index is set to size - 1 - i, meaning the node’s value will be inserted from right to left.

Step 6: Enqueue the left and right child nodes of the current node (if they exist) into the queue.



Step 7: After processing all the nodes at the current level, add the 'row' array to the 'ans' 2D list, representing the current level. Reverse the direction of traversal for the next level by updating the 'leftToRight' flag to its opposite value. This toggling ensures that the nodes at the next level will be processed in the opposite direction, alternating between left-to-right and right-to-left.

Step 8: Once the traversal loop completes return the 'ans' 2D list.

Time Complexity: O(N) where N is the number of nodes in the binary tree. Each node of the binary tree is enqueued and dequeued exactly once, hence all nodes need to be processed and visited. Processing each node takes constant time operations which contributes to the overall linear time complexity.

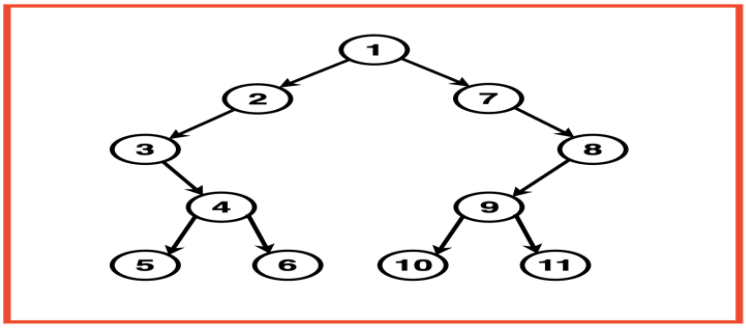
Space Complexity: O(N) where N is the number of nodes in the binary tree. In the worst case, the queue has to hold all the nodes of the last level of the binary tree, the last level could at most hold N/2 nodes hence the space complexity of the queue is proportional to O(N). The resultant vector answer also stores the values of the nodes level by level and hence contains all the nodes of the tree contributing to O(N) space as well.

Code:

```java
static List<List<Integer>> optimal(Node root){  1 usage
    List<List<Integer>> res = new ArrayList<>();
    if (root==null) return res;

    boolean leftToRight = true;
    Queue<Node> q = new LinkedList<>();
    q.add(root);
    while (!q.isEmpty()){
        int s = q.size();
        Integer[] row = new Integer[s];
        for (int i = 0; i < s; i++) {
            Node temp = q.remove();

            int index = (leftToRight) ? i : s-1-i;
            row[index] = temp.data;

            if (temp.left != null) q.add(temp.left);
            if (temp.right != null) q.add(temp.right);
        }
        leftToRight = !leftToRight;
        res.add(Arrays.asList(row));
    }
    return res;
}
```

Given a Binary Tree, perform the boundary traversal of the tree. The boundary traversal is the process of visiting the boundary nodes of the binary tree in the anticlockwise direction, starting from the root.

Input: 1 2 7 3 -1 -1 8 -1 4 9 -1 5 6 10 11



Output: Boundary Traversal: [1, 2, 3, 4, 5, 6, 10, 11, 9, 8, 7]

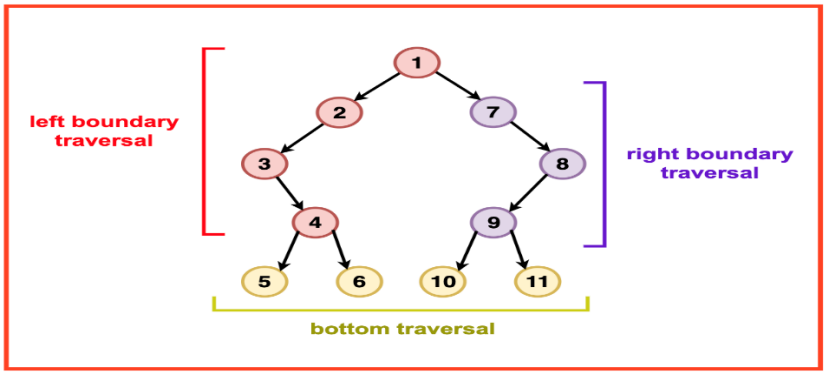
Explanation: The boundary traversal of a binary tree involves visiting its boundary nodes in an anticlockwise direction:

- Starting from the root, we traverse from: 1
- The left side traversal includes the nodes: 2, 3, 4

- The bottom traversal includes the leaf nodes: 5, 6, 10, 11
- The right-side traversal includes the nodes: 9, 8, 7
- We return to the root and the boundary traversal is complete.

Optimal:

The boundary traversal algorithm should be divided into three main parts traversed in the anti-clockwise direction:



Left Boundary: Traverse the left boundary of the tree. Start from the root and keep moving to the left child; if unavailable, move to the right child. Continue this until we reach a leaf node.

Bottom Boundary: Traverse the bottom boundary of the tree by traversing the leaf nodes using a simple preorder traversal. We check if the current node is a lead, and if so, its value is added to the boundary traversal array.

Right Boundary: The right boundary is traversed in the reverse direction, similar to the left boundary traversal starting from the root node and keep moving to the right child; if unavailable, move to the left child. Nodes that are not leaves are pushed into the right boundary array from end to start to ensure that they are added in the reverse direction.

Algorithm:

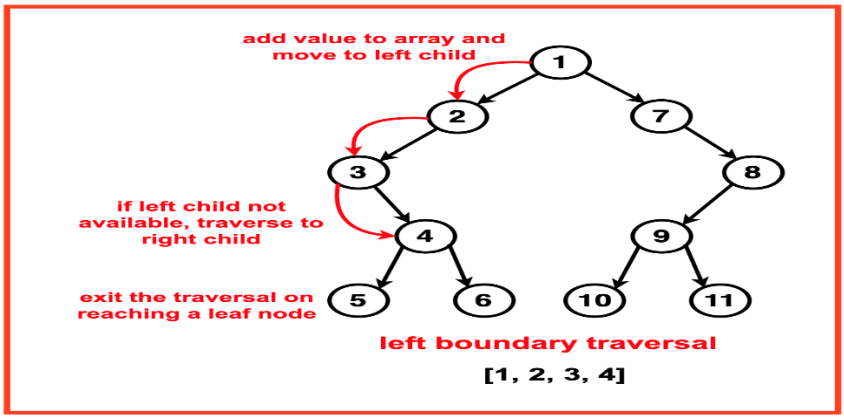
Step 1: Initialise an empty Arraylist to store the boundary traversal nodes.

Step 2: Create a helper function to check if a node is a leaf. This is to avoid cases where there will be an overlap in the traversal of nodes. We exclude leaf nodes when adding left and right boundaries as they will already be added when in the bottom boundary.

Step 3: Initialise a function `addLeftBoundary`.

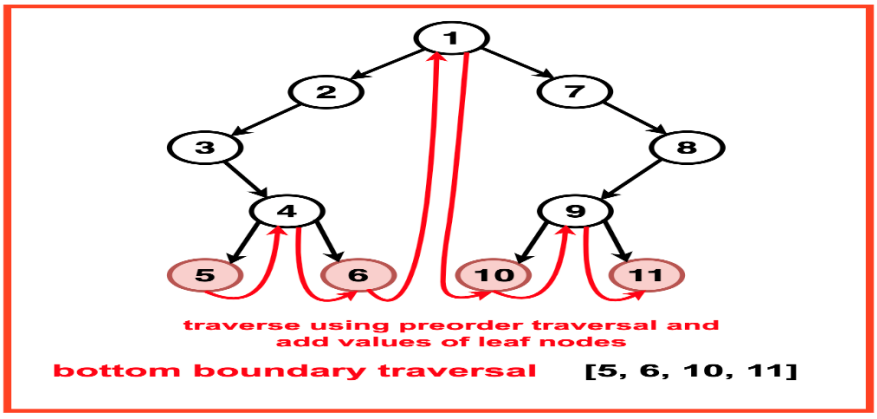
- Start from the root of the tree.

- Traverse down the left side of the tree until we reach a leaf node. For each non-leaf node, add its value to the result Arraylist.
- Traverse to its left child. If unavailable, call its right child.



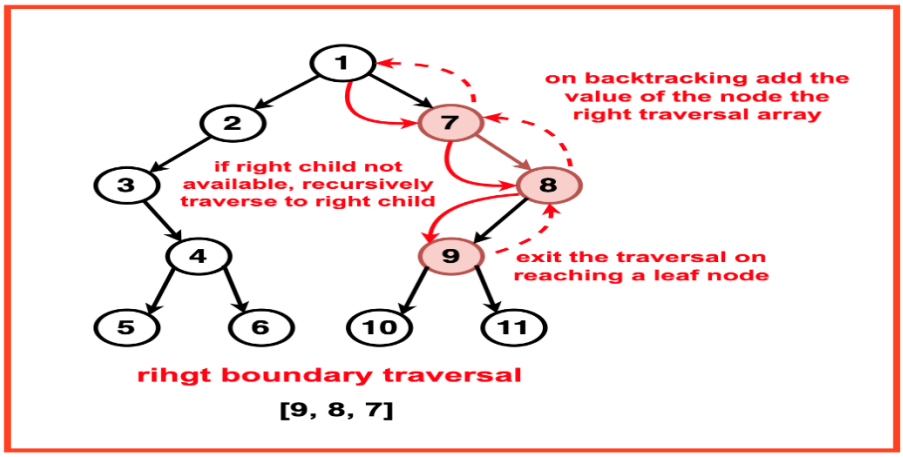
Step 4: Implement a recursive function `addLeafNodes`.

- If the current node is a leaf, add its value to the result Arraylist.
- Recursively travel to the current nodes left and right subtrees in a preorder fashion.



Step 5: Implement a function `addRightBoundary`.

- Start from the root of the tree.
- Create a temporary AraryList.
- Traverse to the right most side of the tree until we reach a leaf node.
- For each non-leaf node, add it's value in temporary list & move to right child if available if not then move to left child.
- Add the temporary list in reverse order in the result list.



Code:

```
static boolean isLeaf(Node curr){
    if (curr.left == null && curr.right == null) return true;
    return false;
}
static void addLeftBoundary(Node curr, ArrayList<Integer> res){
    while (curr != null){
        if (!isLeaf(curr)) res.add(curr.data);
        if (curr.left != null) curr = curr.left;
        else curr = curr.right;
    }
}
static void addLeafNodes(Node root, ArrayList<Integer> res){
    if (isLeaf(root)){
        res.add(root.data);
        return;
    }
    if (root.left != null) addLeafNodes(root.left,res);
    if (root.right != null) addLeafNodes(root.right,res);
}
static void addRightBoundary(Node curr, ArrayList<Integer> res){
    ArrayList<Integer> l = new ArrayList<>();
    while (curr != null){
        if (!isLeaf(curr)) l.add(curr.data);
        if (curr.right != null) curr = curr.right;
        else curr = curr.left;
    }
    for (int i=l.size()-1;i>=0;i--) res.add(l.get(i));
}
static List<Integer> optimal(Node root){
    ArrayList<Integer> res = new ArrayList<>();
    if (!isLeaf(root)) res.add(root.data);
    addLeftBoundary(root.left,res);
    addLeafNodes(root,res);
    addRightBoundary(root.right,res);
    return res;
}
```

Time Complexity: O(N) where N is the number of nodes in the Binary Tree.

- Adding the left boundary of the Binary Tree results in the traversal of the left side of the tree which is proportional to the the height of the three hence O(H) ie. O(log2N). In the worst case that the tree is skewed the complexity would be O(N).

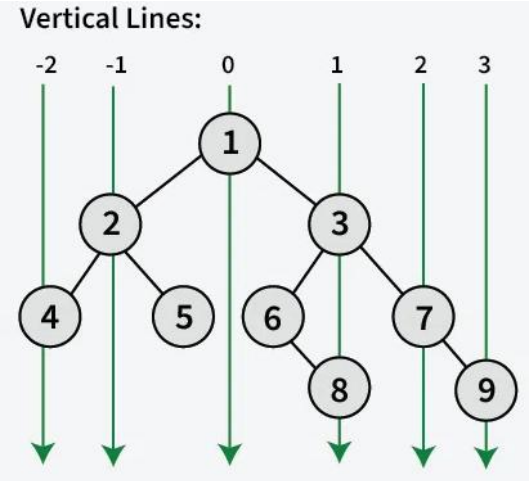
- For the bottom traversal of the Binary Tree, traversing the leaves is proportional to O(N) as preorder traversal visits every node once.
- Adding the right boundary of the Binary Tree results in the traversal of the right side of the tree which is proportional to the the height of the three hence O(H) ie. O(log2N). In the worst case that the tree is skewed the complexity would be O(N).

Since all these operations are performed sequentially, the overall time complexity is dominated by the most expensive operation, which is O(N).

Space Complexity: O(N) where N is the number of nodes in the Binary Tree to store the boundary nodes of the tree. O(H) or O(log2N) Recursive stack space while traversing the tree. In the worst case scenario the tree is skewed and the auxiliary recursion stack space would be stacked up to the maximum depth of the tree, resulting in an O(N) auxiliary space complexity.

Given a root of a Binary Tree, find the vertical traversal of it starting from the leftmost level to the rightmost level. If there are multiple nodes passing through a vertical line, then they should be printed as they appear in level order traversal of the tree.

Input: [1, 2, 3, 4, 5, 6, 7, N, N, N, N, N, 8, N, 9]



Output: [[4], [2], [1, 5, 6], [3, 8], [7], [9]]

Optimal:

Algorithm:

- **Initialize tracking variables:**
  - Keep track of the minimum and maximum column indices encountered while traversing the tree
- **Create a mapping structure:**

- Use a map to store a list of node values for each column. The key is the column number, and the value is the list of nodes in that column.

- **Set up a queue for level-order traversal:**
  - Use a queue to perform a breadth-first traversal (level-order traversal) of the tree.
  - Each element in the queue should store three pieces of information:
    1. The current node.
    2. Its level (or depth) in the tree.
    3. Its column index.
- **Start traversal from the root:**
  - Add the root node to the queue with level 0 and column 0.
- **Process nodes in the queue:**
  - While the queue is not empty:
    - Remove a node from the front of the queue.
    - Update the minimum and maximum column values based on the node's column index.
    - Add the node's value to the list corresponding to its column in the map.
    - If the node has a left child, add it to the queue with:
      - Level incremented by 1.
      - Column decremented by 1 (moving left).
    - If the node has a right child, add it to the queue with:
      - Level incremented by 1.
      - Column incremented by 1 (moving right).
- **Build the final result:**
  - Initialize an empty list to store the final vertical order traversal.
  - Loop from the minimum column to the maximum column:
    - Retrieve the list of nodes from the map for that column.
    - Add this list to the final result.
- **Return the result**

Code:

```
static ArrayList<ArrayList<Integer>> verticalOrder(Node root) {
    int minCol = 0, maxCol = 0;
    Map<Integer,ArrayList<Integer>> mp = new HashMap<>();
    Queue<Triplet> q = new LinkedList<>();
    q.add(new Triplet(root,0,0));
    while (!q.isEmpty()){
        int s = q.size();
        for (int i=0;i<s;i++){
            Triplet p = q.remove();

            if(p.column > maxCol) maxCol = p.column;
            if(p.column < minCol) minCol = p.column;

            mp.putIfAbsent(p.column, new ArrayList<>());
            mp.get(p.column).add(p.temp.data);

            if (p.temp.left != null) q.add(new Triplet(p.temp.left,p.level+1,p.column-1));
            if (p.temp.right != null) q.add(new Triplet(p.temp.right,p.level+1,p.column+1));
        }
    }

    ArrayList<ArrayList<Integer>> res = new ArrayList<>();
    for (int col = minCol; col <= maxCol; col++) {
        res.add(mp.get(col));
    }
    return res;
```

T.C: O(n) + O(n)

O(n) for level order traversal & another O(n) for adding node value into ArrayList.

S.C: O(n/2) + O(n) + O(n)

Queue for BFS:

- At most, the queue stores all nodes at the current level.
- In the worst case (for a complete binary tree), the last level can have about n/2 nodes, so space is O(n).

Map for storing columns:

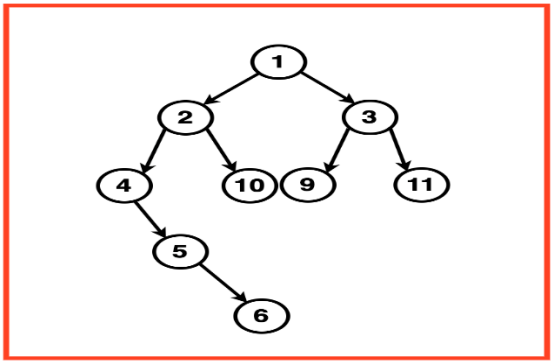
- The map stores all nodes, grouped by column. Total elements stored = n nodes.
- Space for keys (columns) ≤ n in worst case.
- So, map space = O(n).

Result list:

- The final result stores all nodes in a nested list. Total elements = n.
- So, space = O(n).

Given the root of a binary tree, calculate the vertical order traversal of the binary tree. For each node at position (row, col), its left and right children will be at positions (row + 1, col - 1) and (row + 1, col + 1) respectively. The root of the tree is at (0, 0). The vertical order traversal of a binary tree is a list of top-to-bottom orderings for each column index starting from the leftmost column and ending on the rightmost column. There may be multiple nodes in the same row and same column. In such a case, sort these nodes by their values. Return *the vertical order traversal of the binary tree*.

Input: 1 2 3 4 10 9 11 -1 5 -1 -1 -1 -1 -1 -1 -1 6



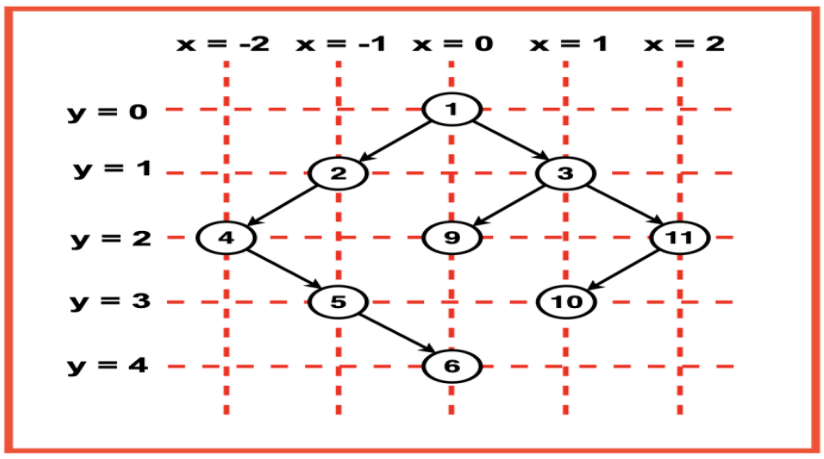
Output: Vertical Order Traversal: [[4],[2, 5], [1, 10, 9, 6],[3],[11]]

Explanation: Vertical Levels from left to right:

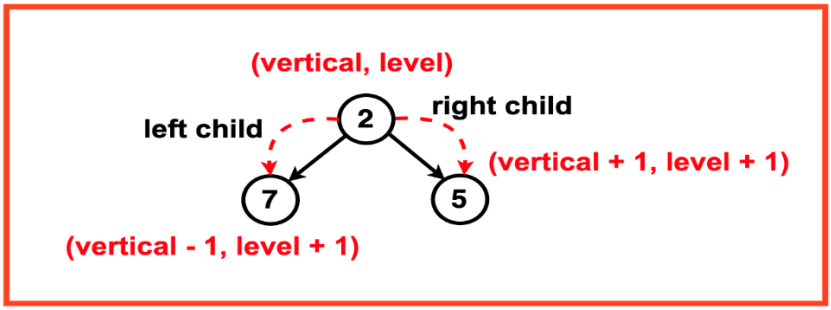
- Level -2: [4]
- Level -1: [2]
- Level 0: [1, 9,10, 6] (Overlapping nodes are added in sorted order)
- Level 1: [3]
- Level 2: [11]

Optimal:

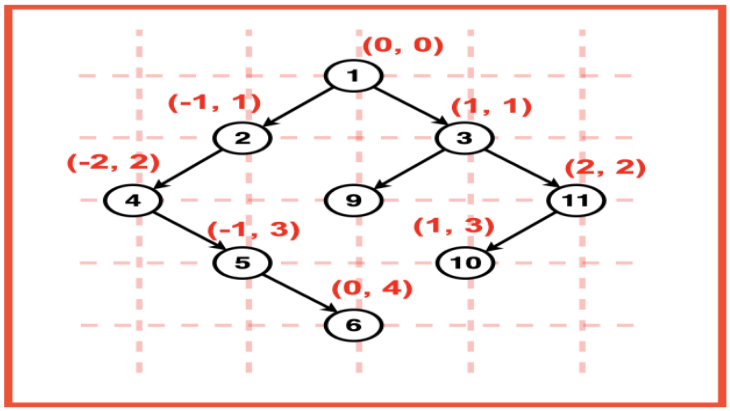
We can assign a vertical and level to every node. This will help us in categorising nodes based on their position in the binary tree. **Vertical Coordinates (x):** The vertical coordinate, denoted as 'x', represents the vertical column in the tree. It essentially signifies the horizontal position of a node in relation to its parent. Nodes with the same 'x' value are aligned vertically, forming a column. **Level Coordinates (y):** The level coordinate, denoted as 'y', represents the depth or level of a node in the tree. It signifies the vertical position of a node within the hierarchy of levels. As we traverse down the tree, the 'y' value increases, indicating a deeper level.



We create a map that serves as our organisational structure. The map is based on the vertical and level information of each node. The vertical information, represented by 'x', signifies the vertical column, while the level information, denoted as 'y', acts as the key within the nested map. This nested map utilises a multiset to ensure that node values are stored in a unique and sorted order. With our map structure in place, we initiate a level order BFS traversal using a queue. Each element in the queue is a pair containing the current node and its corresponding vertical and level coordinates. Starting with the root node, we enqueue it with initial vertical and level values (0, 0). During traversal, for each dequeued node, we update the map by inserting the node value at its corresponding coordinates and enqueue its left and right children with adjusted vertical and level information. When traversing to the left child, the vertical value decreases by 1 and the level increases by 1, while traversal to the right child leads to an increase in both vertical and level by 1.

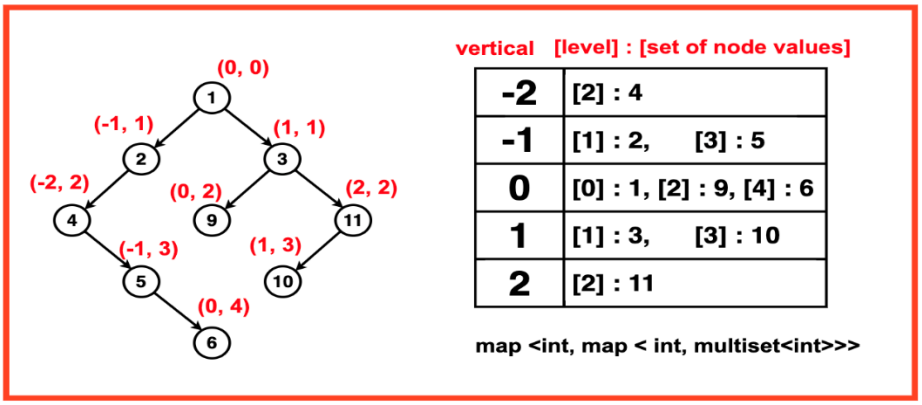


After completing the BFS traversal, we prepare the final result vector. We iterate through the map, creating a column vector for each vertical column. This involves gathering node values from the multiset and inserting them into the column vector. These column vectors are then added to the final result vector, resulting in a 2D representation of the vertical order traversal of the binary tree.

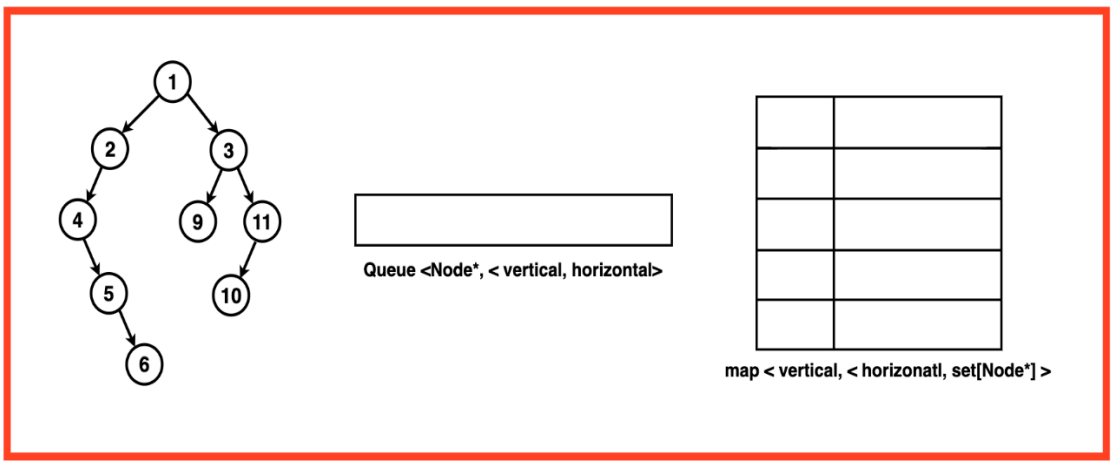


Algorithm:

Step 1: Create an empty map to store the nodes based on their vertical and horizontal levels. The key of the map 'x' represents the vertical column, and the nested map uses 'y' as the key for the level. Initialise a 'PriorityQueue' to store node values at a specific vertical and level to ensure unique and sorted order of nodes.

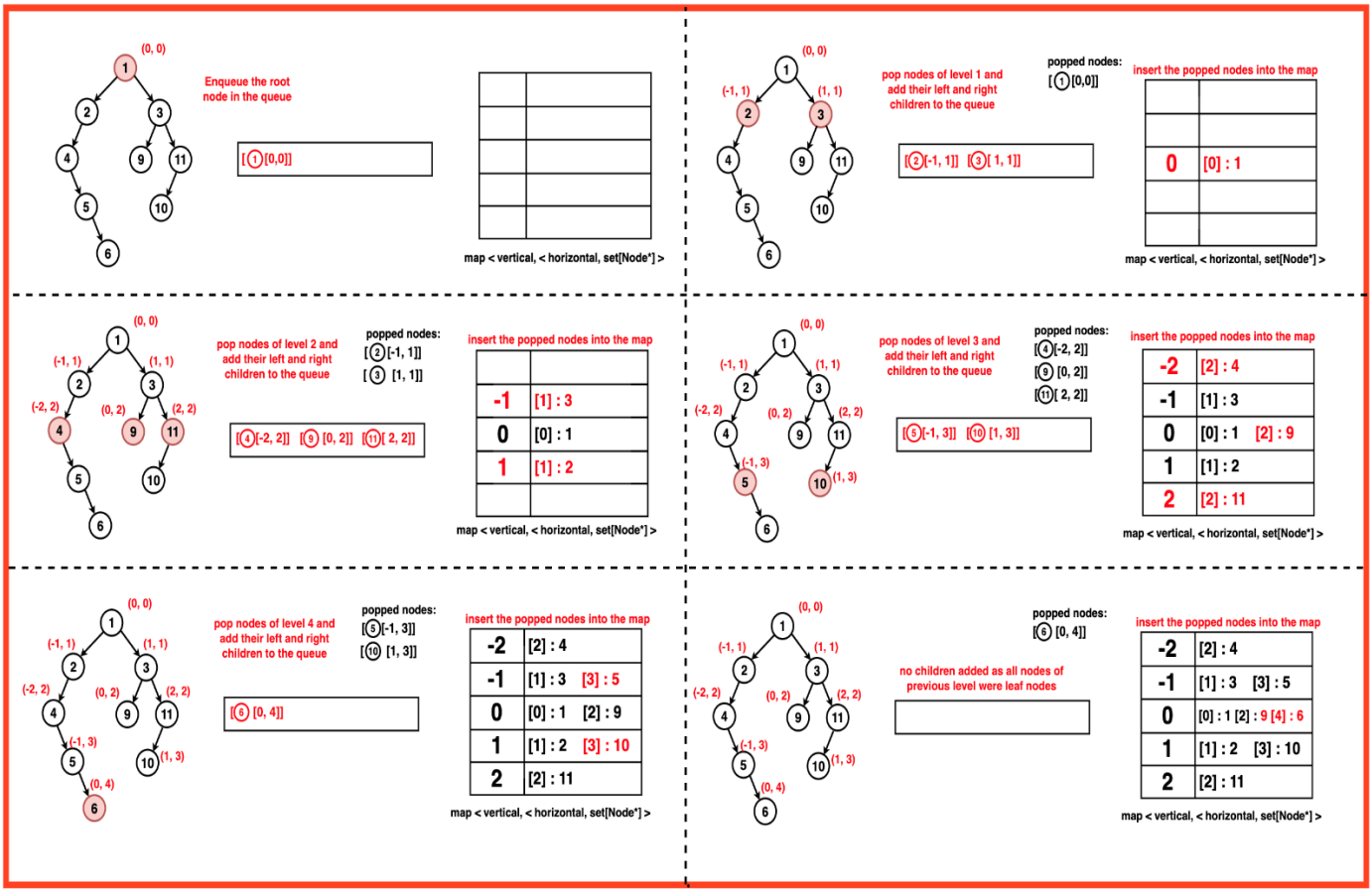


Step 2: Initialise a queue for level order BFS traversal. Each element in the queue should be a pair containing the current node and its vertical and level order information as x and coordinates. Enqueue the root node into the queue with its initial vertical and level order values as (0, 0)

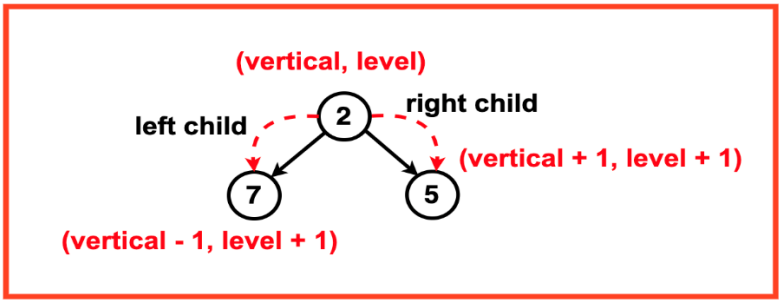


Step 3: While the queue is not empty, pop the front node of the queue:

- Get this nodes vertical ie. ‘x’ and level order ‘y’ information.
- Insert this node into the map at its corresponding coordinate.
- Push the left and right child of the node with their updated horizontal distance and level order.



For the left child, decrement the vertical value ‘x’ by 1 to indicate a move towards the left. Increment the level value ‘y’ by 1 to indicate a move down to the next level. For the right child, increment the vertical value ‘x’ by 1 to indicate a move towards the right. Increment the level value ‘y’ by 1 to indicate a move down to the next level.



Enqueue both the left and right children along with their updated vertical and level information into the queue.

Step 4: After the BFS traversal using the queue is complete, initialise a final result 2D ArrayList 'ans'.

- Iterate through the map, creating a column 1D ArrayList for each vertical column. Gather the node values from the PriorityQueue and insert them into the column vector.
- Add these column vectors to the final result vector 'ans'.

Step 5: Return the 2D List `ans` representing the vertical order traversal of the binary tree.

T.C = O (n*logn * logn * logn) + O(nlogn) n: number of nodes in the tree.

- Postorder Traversal performed using BFS as a time complexity of O(N) as we are visiting each and every node once.
- Insertion into **TreeMap**: **O (log C)** where C = number of unique columns. Worst case, C = n → **O (log n)**.
- Insertion into an **inner TreeMap** keyed by level: **O (log L)** where L = number of levels in that column.Worst case, L ≤ n → **O (log n)**.
- Adding into a **PriorityQueue**: **O (log k)** where k = number of nodes at same (column, level). In worst case, k = n → **O (log n)**.
- Each node's value goes into exactly one priority queue during BFS. pq.poll() = O (log k), where k = size of that Priority Queue. In the worst case, one PQ could contain n elements → each poll = O (log n).
  So extracting all n elements = O (n log n).

S.C = O(n) + O(n) + O(n) where n: number of nodes in the tree.

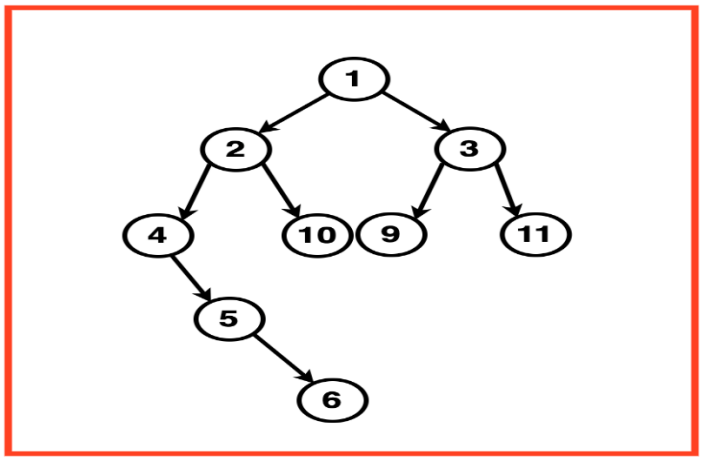
- Queue for BFS = **O(n)**
- Nested TreeMap + PQs = each node stored once = **O(n)**
- Result list = **O(n)**

Code:

```
static List<List<Integer>> optimal(Node root){  1 usage
    TreeMap<Integer,TreeMap<Integer, PriorityQueue<Integer>>> mp = new TreeMap<>();
    Queue<Triplet> q = new LinkedList<>();
    q.add(new Triplet(root, level: 0, column: 0));
    while (!q.isEmpty()){
        int s = q.size();
        for (int i=0;i<s;i++){
            Triplet p = q.remove();
            mp.putIfAbsent(p.column, new TreeMap<>());
            mp.get(p.column).putIfAbsent(p.level, new PriorityQueue<>());
            mp.get(p.column).get(p.level).add(p.temp.data);

            if (p.temp.left != null) q.add(new Triplet(p.temp.left, level: p.level+1, column: p.column-1));
            if (p.temp.right != null) q.add(new Triplet(p.temp.right, level: p.level+1, column: p.column+1));
        }
    }

    List<List<Integer>> res = new ArrayList<>();
    for (TreeMap<Integer, PriorityQueue<Integer>> innerMap : mp.values()) {
        List<Integer> col = new ArrayList<>();
        for (PriorityQueue<Integer> pq : innerMap.values()) {
            while (!pq.isEmpty())
                col.add(pq.poll());
        }
        res.add(col);
    }
    return res;
}
```

Given a Binary Tree, return its Top View. The Top View of a Binary Tree is the set of nodes visible when we see the tree from the top.

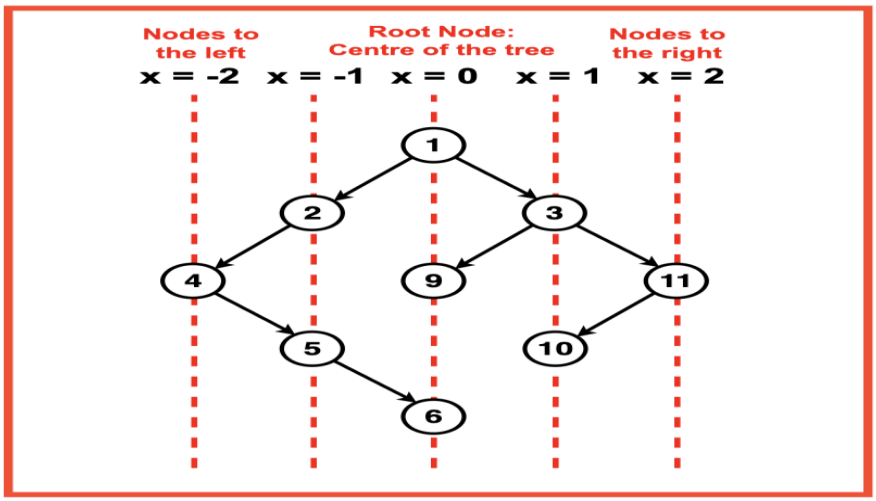
Input: 1 2 3 4 10 9 11 -1 5 -1 -1 -1 -1 -1 -1 -1 6



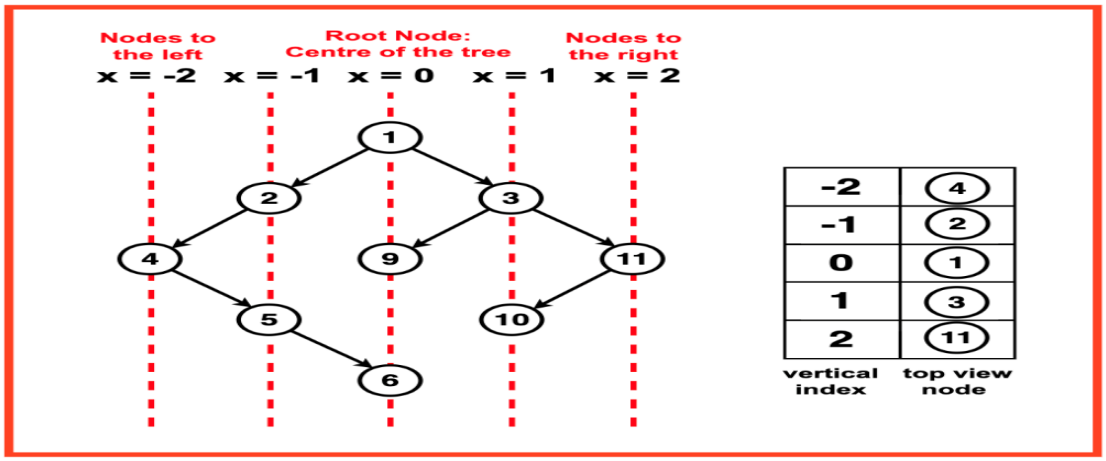
Output: Top View: [4, 2, 1, 3, 11]

Optimal:

To imagine the Binary Tree from above, we visualise vertical lines passing through the tree. Each vertical line represents a unique vertical position. Nodes to the right of the tree's centre are assigned positive vertical indexes. As we move to the right, the vertical index increases. Nodes to the left of the tree's centre are assigned negative vertical indexes. As we move to the left, the vertical index decreases.



We use a map data structure to store the nodes corresponding to each vertical level of the tree. Against each vertical level, the node highest in the tree at that vertical level is added by traversing the tree level order wise (BFS).

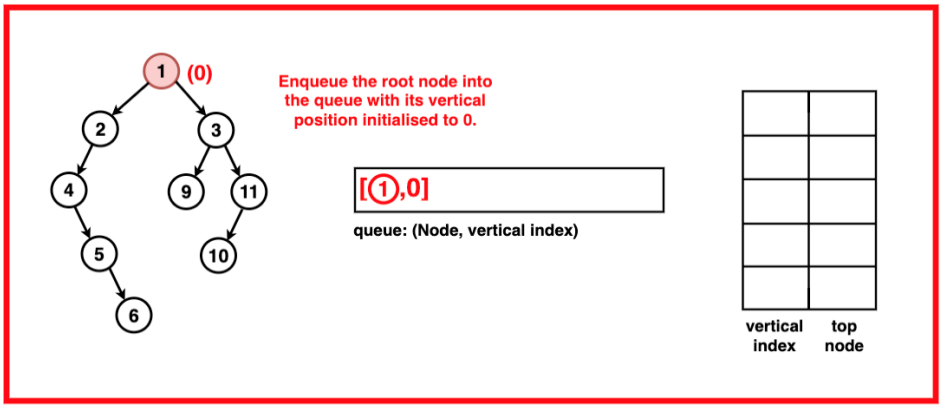


Algorithm:

Step 1: Initialize variables minCol and maxCol to track the leftmost and rightmost columns encountered in the tree.

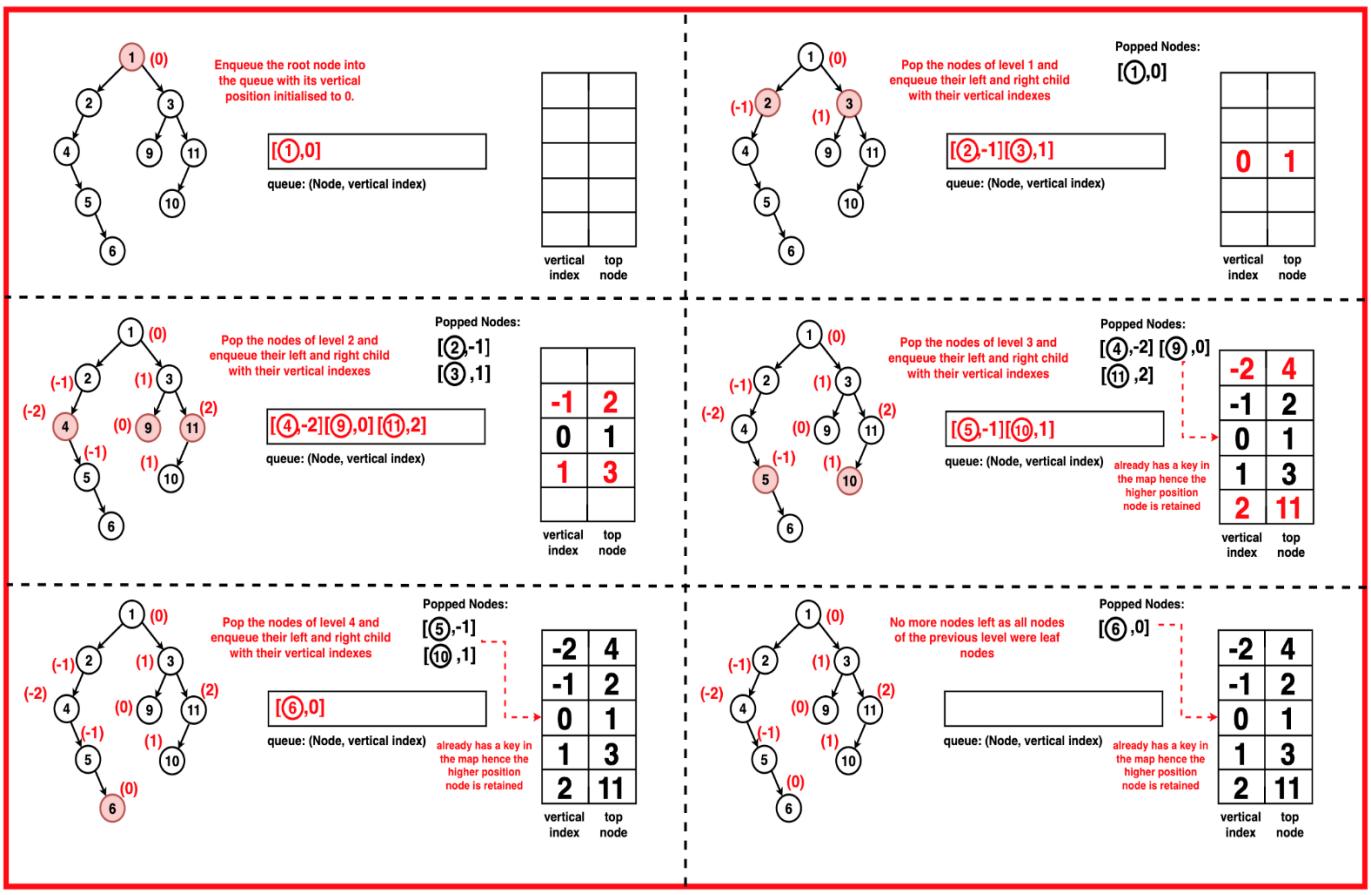
Step 2: Create a map to store the top view of nodes based on their vertical positions. The key of this map is the vertical index and the value is the node's data.

Step 3: Initialise a queue to perform breadth first traversal. Each element in the queue stores, the current node & Its column index. Add the root node with column 0 to the queue.



Step 4: While the queue is not empty,

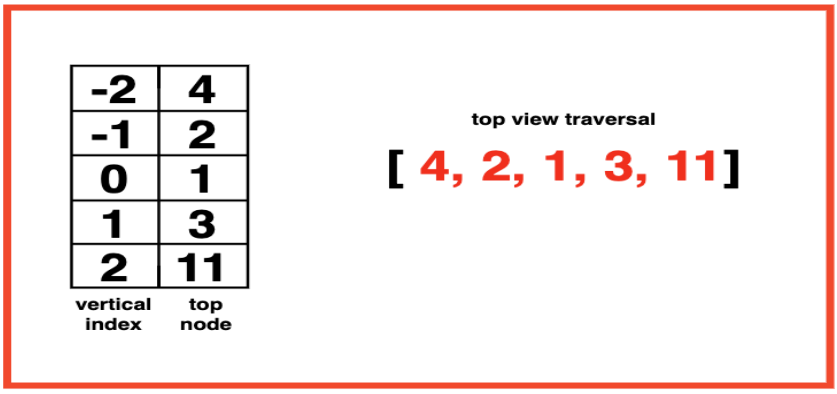
- pop the front node of the queue.
- Update minCol and maxCol based on the node's column.
- If the column is not already in the map, add the node's value (putIfAbsent) → ensures topmost node per column is stored.
- If the node has a left child, add it to the queue with column index current column - 1.
- If the node has a right child, add it to the queue with column index current column + 1.



Step 5: Initialize an empty list for the result.

- Loop from minCol to maxCol.
- Retrieve the value from the map for that column. Append it to the result list.

Step 6: Return the result



Code:

```java
class Pair{
    Node temp;
    int column;

    Pair(Node temp,int column){
        this.temp = temp;
        this.column = column;
    }
}
class Solution {
    static ArrayList<Integer> topView(Node root) {
        int minCol = 0, maxCol = 0;
        Map<Integer,Integer> mp = new HashMap<>();
        Queue<Pair> q = new LinkedList<>();
        q.add(new Pair(root,0));
        while (!q.isEmpty()){
            int s = q.size();
            for (int i=0;i<s;i++){
                Pair p = q.remove();
                if(p.column > maxCol) maxCol = p.column;
                if(p.column < minCol) minCol = p.column;
                mp.putIfAbsent(p.column, p.temp.data);

                if (p.temp.left != null) q.add(new Pair(p.temp.left,p.column-1));
                if (p.temp.right != null) q.add(new Pair(p.temp.right,p.column+1));
            }
        }
        ArrayList<Integer> res = new ArrayList<>();
        for (int col = minCol; col <= maxCol; col++) {
            res.add(mp.get(col));
        }
        return res;
    }
}
```

T.C: O(n)

BFS Traversal: Each node is visited exactly once → O(n).

Map operations:

- putIfAbsent and get in HashMap are O (1) on average.
- Across all nodes → O (n).

Building the result list:

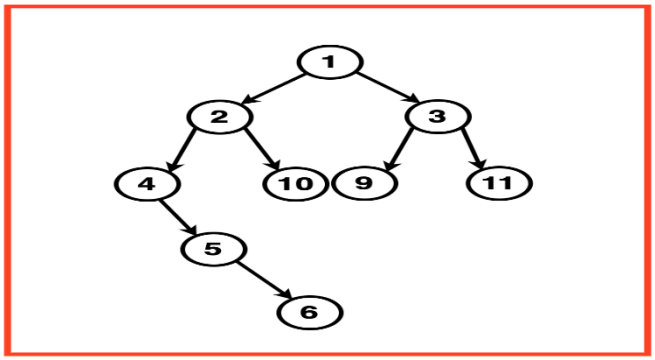
- Loop from minCol to maxCol → number of columns = c (≤ n).
- Accessing the map is O (1) per column → O (c) → O (n) in worst case.

S.C: O(n)

- Queue for BFS → O (width of tree) ≤ O(n)
- Map → O (number of columns) ≤ O(n)
- Result list → O (number of columns) ≤ O(n)

Given a Binary Tree, return its Bottom View. The Bottom View of a Binary Tree is the set of nodes visible when we see the tree from the bottom.

Input: 1 2 3 4 10 9 11 -1 5 -1 -1 -1 -1 -1 -1 -1 6

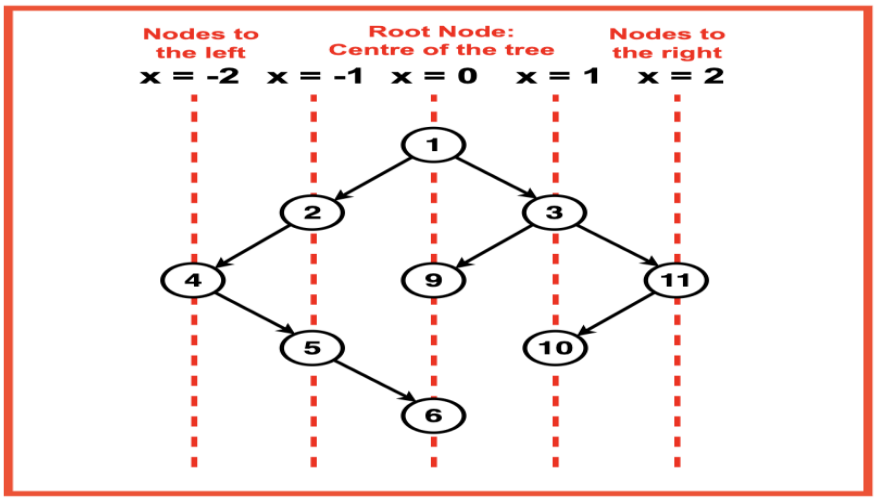


Output: [4, 5, 6, 3, 11]

Explanation: The bottom view of the binary tree would comprise of the nodes that are the last encountered nodes for each vertical index.

Optimal:

To imagine the Binary Tree from above, we visualise vertical lines passing through the tree. Each vertical line represents a unique vertical position. Nodes to the right of the tree's centre are assigned positive vertical indexes. As we move to the right, the vertical index increases. Nodes to the left of the tree's centre are assigned negative vertical indexes. As we move to the left, the vertical index decreases.



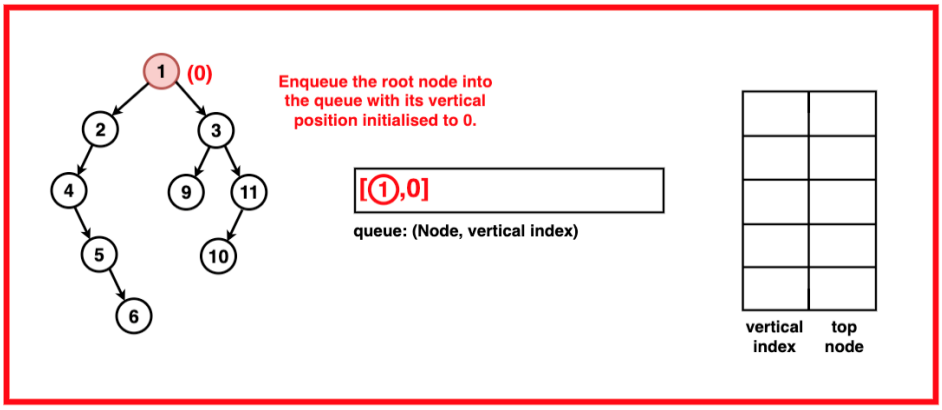
We use a map data structure to store the nodes corresponding to each vertical level of the tree. Against each vertical level, the node lowest in the tree at that vertical level is added by traversing the tree level order wise (BFS).

Algorithm:

Step 1: Initialize variables minCol and maxCol to track the leftmost and rightmost columns encountered in the tree.

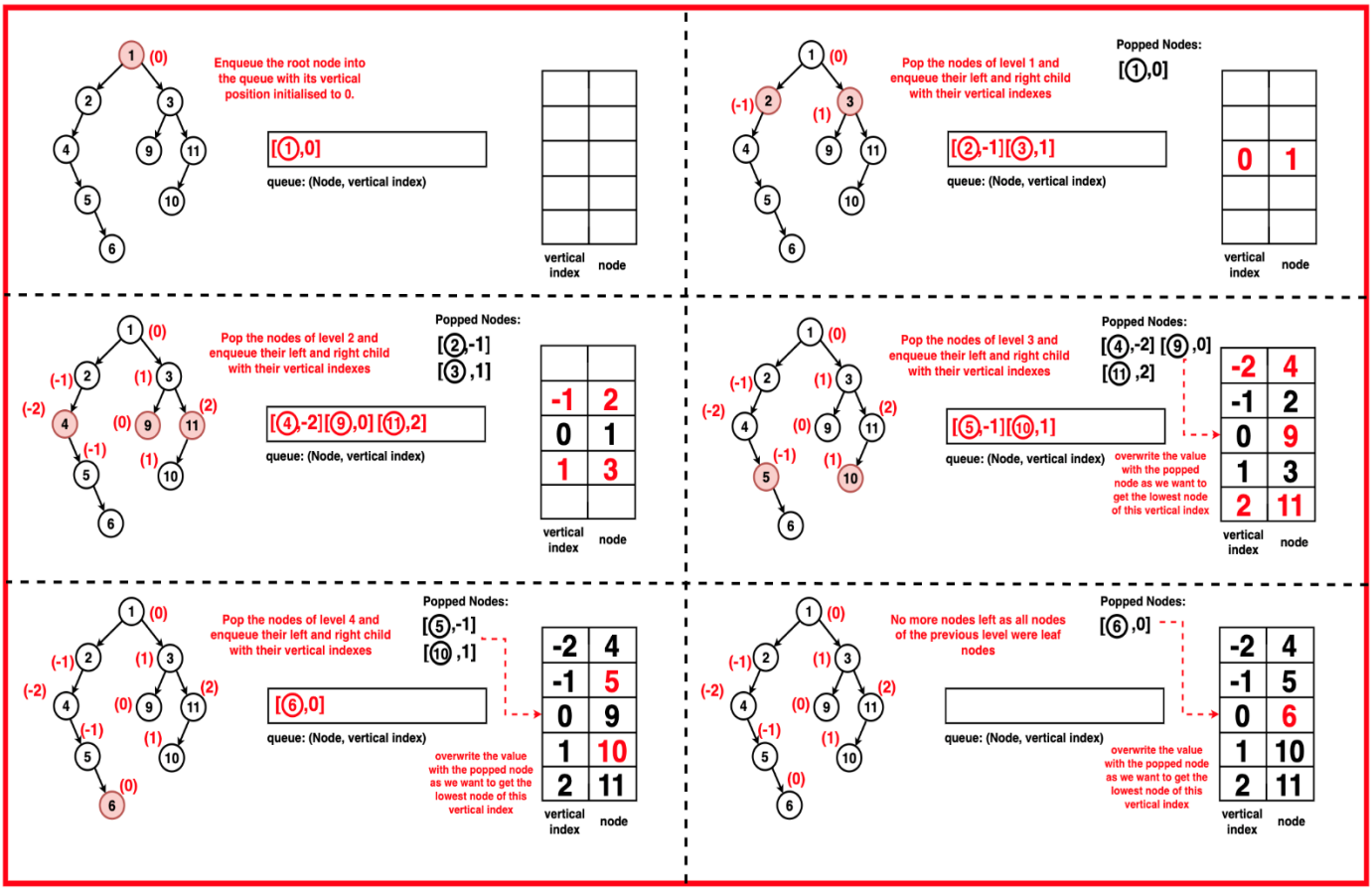
Step 2: Create a map to store the top view of nodes based on their vertical positions. The key of this map is the vertical index and the value is the node's data.

Step 3: Initialise a queue to perform breadth first traversal. Each element in the queue stores, the current node & Its column index. Add the root node with column 0 to the queue.



Step 4: While the queue is not empty,

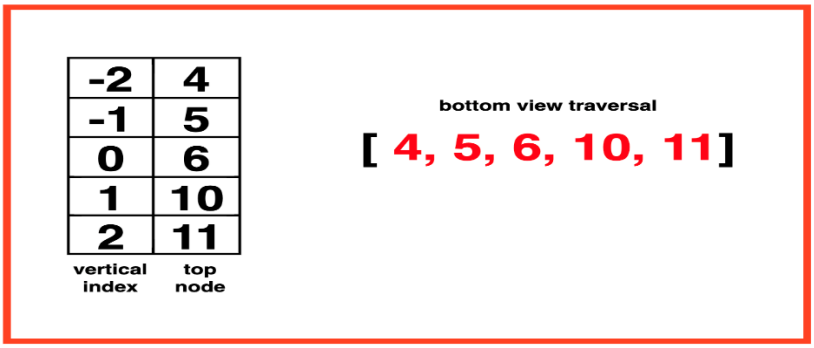
- pop the front node of the queue.
- Update minCol and maxCol based on the node's column.
- If the column is not already in the map, add the node's data to the map. This means that this node is the first node encountered at this vertical position during the traversal.
- If the column is already in the map, it implies that a node higher in the tree with the same vertical position has already been processed. Overwrite this position with the current node as we want to get the lowest node of that vertical index.
- If the node has a left child, add it to the queue with column index current column - 1.
- If the node has a right child, add it to the queue with column index current column + 1.



Step 5: Initialize an empty list for the result.

- Loop from minCol to maxCol.
- Retrieve the value from the map for that column. Append it to the result list.

Step 6: Return the result



Code:

```java
class Pair{
    Node temp;
    int column;

    Pair(Node temp,int column){
        this.temp = temp;
        this.column = column;
    }
}
class Solution {
    public ArrayList<Integer> bottomView(Node root) {
        // Code here
        int minCol = 0, maxCol = 0;
        Map<Integer,Integer> mp = new HashMap<>();
        Queue<Pair> q = new LinkedList<>();
        q.add(new Pair(root,0));
        while (!q.isEmpty()){
            int s = q.size();
            for (int i=0;i<s;i++){
                Pair p = q.remove();
                if(p.column > maxCol) maxCol = p.column;
                if(p.column < minCol) minCol = p.column;
                mp.put(p.column, p.temp.data);

                if (p.temp.left != null) q.add(new Pair(p.temp.left,p.column-1));
                if (p.temp.right != null) q.add(new Pair(p.temp.right,p.column+1));
            }
        }
        ArrayList<Integer> res = new ArrayList<>();
        for (int col = minCol; col <= maxCol; col++) {
            res.add(mp.get(col));
        }
        return res;
    }
}
```

T.C: O(3n)

BFS Traversal: Each node is visited exactly once → O(n).

Map operations (put): Done once per node → **O(n)**

Building the result list:

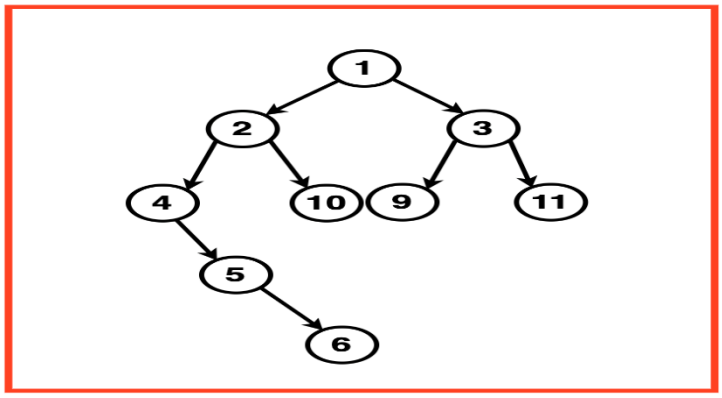
- Loop from minCol to maxCol → number of columns = c (≤ n).
- Accessing the map is O (1) per column → O (c) → O (n) in worst case.

S.C: O(3n)

- Queue for BFS → O (width of tree) ≤ O(n/2)
- Map → O (number of columns) ≤ O(n)
- Result list → O (number of columns) ≤ O(n)

Given a Binary Tree, return its right and left views. The Right View of a Binary Tree is a list of nodes that can be seen when the tree is viewed from the right side. The Left View of a Binary Tree is a list of nodes that can be seen when the tree is viewed from the left side.

Input: 1 2 3 4 10 9 11 -1 5 -1 -1 -1 -1 -1 -1 -1 6

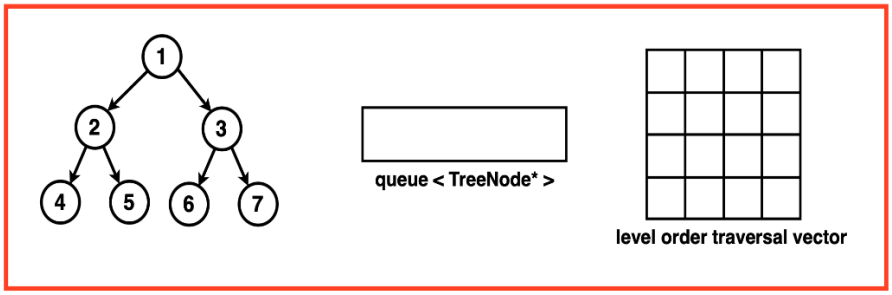


Output: Left View: [1, 2, 4, 5, 6]

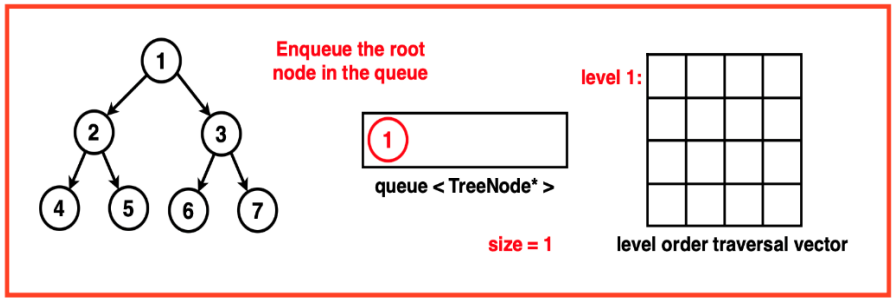
Brute-Force: Using level order traversal

Algorithm:

Step 1: Initialise an empty queue data structure to store the nodes during traversal. Create a 2D arrayList to store the level order traversal. If the tree is empty, return this empty 2D vector.



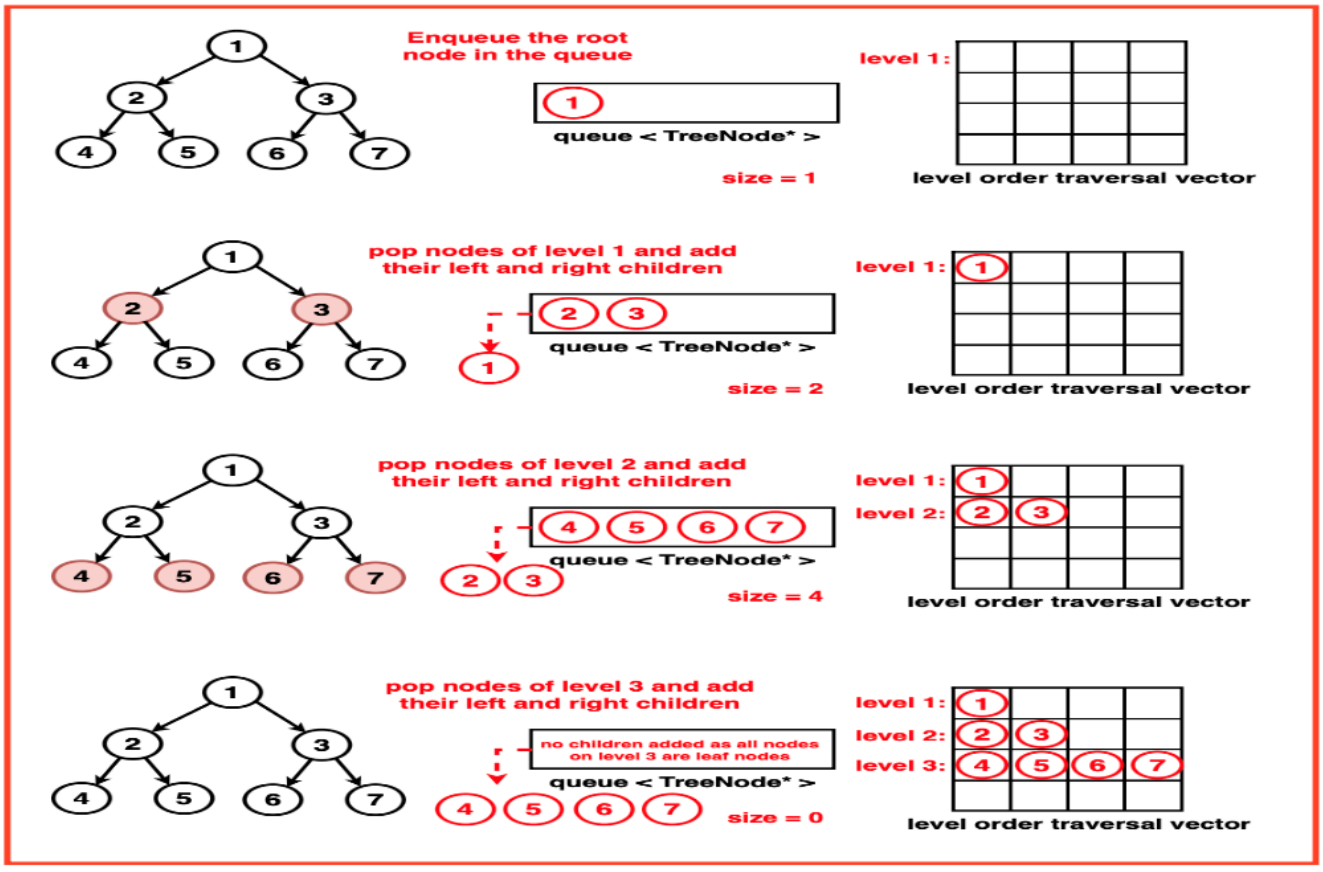
Step 2: Add the root node of the binary tree to the queue.



Step 3: Iterate until the queue is empty:

- Get the current size of the queue. This size indicates the number of nodes at the current level.
- Create a ArrayList‘level’ to store the nodes at the current level.

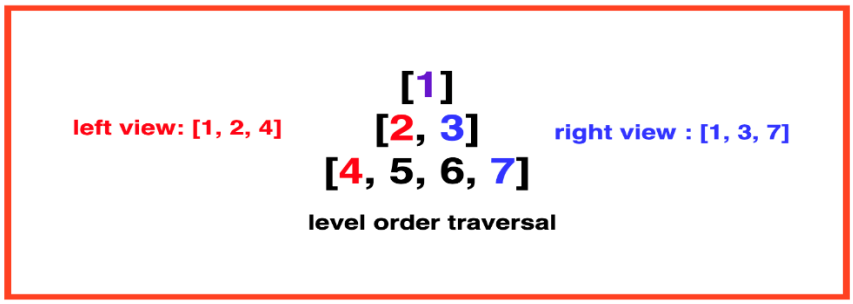
- Iterate through ‘size’ number of nodes at the current level:
    - Pop the front node from the queue.
    - Store the node’s value in the level vector.
    - Enqueue the left and right child nodes of the current node (if they exist) into the queue.
- After processing all the nodes at the current level, add the ‘level’ ArrayList to the ‘ans’ 2D vector, representing the current level.



Step 4: Once the traversal loop completes, the 'ans' 2D ArrayList now contains the level order traversal of the binary tree. To obtain the left view and right view we use each level's vector in the 'ans' vector.

Left View: For each level, extract the first element from the ArrayList and store it in a separate array. Return this array as the final left view of the binary tree.

Right View: For each level, extract the last element from the ArrayList and store it in a separate array. Return this array as the final right view of the binary tree.



Code:

```
static List<Integer> bruteForce(Node root){  1 usage
    List<Integer> res = new ArrayList<>();
    if (root == null) return res;

    Queue<Node> q = new LinkedList<>();
    List<List<Integer>> levelOrder = new ArrayList<>();
    q.add(root);
    while (!q.isEmpty()){
        int s = q.size();
        List<Integer> level = new ArrayList<>();
        for (int i=0;i<s;i++){
            Node temp = q.remove();
            level.add(temp.data);
            if (temp.left != null) q.add(temp.left);
            if (temp.right != null) q.add(temp.right);
        }
        levelOrder.add(level);
    }
    for (List<Integer> l : levelOrder){
        res.add(l.getLast());
    }
    return res;
}
```

Time Complexity: O(N) where N is the number of nodes in the binary tree. Each node of the binary tree is enqueued and dequeued exactly once, hence all nodes need to be processed and visited. Processing each node takes constant time operations which contributes to the overall linear time complexity.

Space Complexity: O(N/2) + O(N) where N is the number of nodes in the binary tree. In the worst case, the queue has to hold all the nodes of the last level of the binary tree, the last level could at most hold N/2 nodes hence the space complexity of the queue is proportional to O(N). The resultant vector answer also stores the values of the nodes level by level and hence contains all the nodes of the tree contributing to O(N) space as well.

Optimal: Using Recursion

To get the left and right view of a Binary Tree, we perform a depth-first traversal of the Binary Tree while keeping track of the level of each node. For both the left and right view, we'll ensure that only the first node encountered at each level is added to the result vector.

Algorithm for Left View

Step 1: Initialise an empty ArrayList `res` to store the left view nodes.

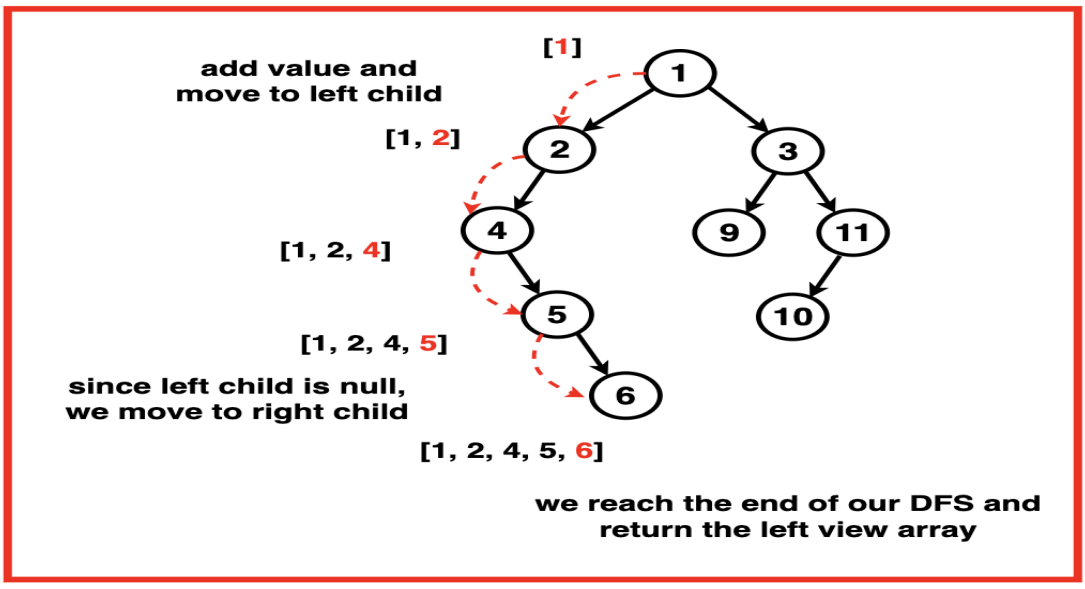
Step 2: Implement a recursive depth-first traversal of the binary tree.

Base Case: Check if the current node is null, if true, return the function as we have reached the end of that particular vertical level.

Recursive Function: The recursive function takes in arguments the current node of the Binary Tree, its current level and the result ArrayList.

- We check if the size of the result ArrayList is equal to the current level.
- If true, it means that we have not yet encountered any node at this level in the result vector. Add the value of the current node to the result vector.
- Recursively call the function for the current nodes left then right child with an increased level ie. level + 1.
- We call the left child first as we want to traverse the left most nodes. In cases where there is no left child, the recursion function backtracks and explores the right child.

Step 3: The recursion continues until it reaches the base case. Return the result ArrayList at the end.



Code:

```java
class Solution {
    static void recursively(Node root,int level,List<Integer> res){
        if (root == null) return ;
        if (level == res.size())
            res.add(root.data);

        recursively(root.left,level+1,res);
        recursively(root.right,level+1,res);
    }

    ArrayList<Integer> leftView(Node root) {
        ArrayList<Integer> res = new ArrayList<>();
        if(root== null)
            return res;

        recursively(root,0,res);
        return res;
    }
}
```

T.C: O(n) As we are visiting each node once.

S.C: O(2h) Stores one value per level. Recursion stack: Depth of recursion = height of tree h.

Algorithm for Right View

Step 1: Initialise an empty ArrayList `res` to store the left view nodes.

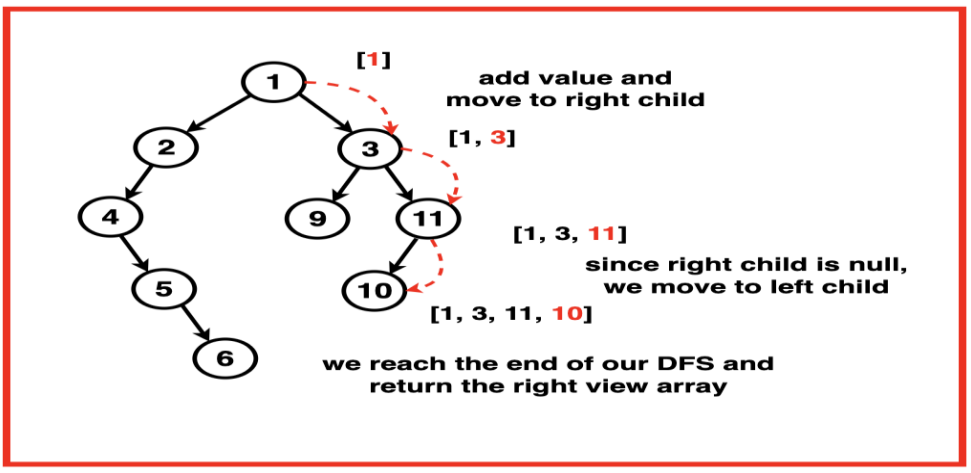
Step 2: Implement a recursive depth-first traversal of the binary tree.

Base Case: Check if the current node is null, if true, return the function as we have reached the end of that particular vertical level.

Recursive Function: The recursive function takes in arguments the current node of the Binary Tree, its current level and the result ArrayList.

- We check if the size of the result vector is equal to the current level.
- If true, it means that we have not yet encountered any node at this level in the result vector. Add the value of the current node to the result ArrayList.
- Recursively call the function for the current nodes right then left child with an increased level ie. level + 1.
- We call the right child first as we want to traverse the right most nodes. In cases where there is no right child, the recursion function backtracks and explores the left child.

Step 3: The recursion continues until it reaches the base case. Return the result ArrayList at the end.



Code:

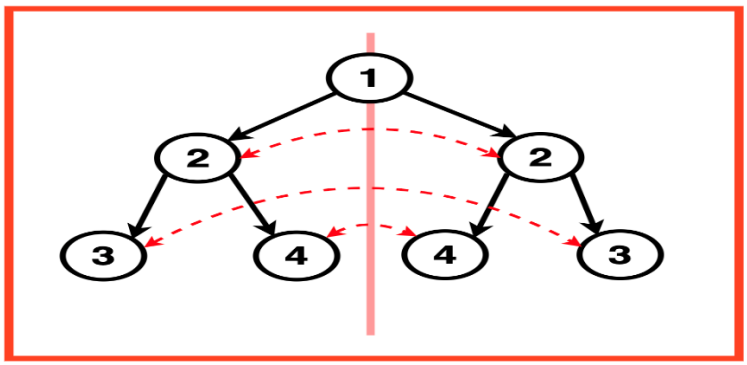
```
class Solution {
    static void recursively(Node root,int level,List<Integer> res){
        if (root == null) return ;
        if (level == res.size())
            res.add(root.data);

        recursively(root.right,level+1,res);
        recursively(root.left,level+1,res);
    }
    ArrayList<Integer> rightView(Node root) {
        ArrayList<Integer> res = new ArrayList<>();
        if(root== null)
            return res;

        recursively(root,0,res);
        return res;
    }
}
```

T.C: O(n) As we are visiting each node once.

S.C: O(2h) Stores one value per level. Recursion stack: Depth of recursion = height of tree h.

Given a Binary Tree, determine whether the given tree is symmetric or not. A Binary Tree would be Symmetric, when its mirror image is exactly the same as the original tree. If we were to draw a vertical line through the centre of the tree, the nodes on the left and right side would be mirror images of each other.

Input: 1 2 2 3 4 4 3



Output: True, this tree is symmetric.

Explanation: If we were to draw a vertical line through the centre of the tree, dividing it into left and right parts, we observe that the nodes on the left and right sides are mirror images of each other.

- The root node (1) is at the centre.
- The left subtree has a node (2) on the left, and the right subtree has a corresponding node (2) on the right.
- Further, the left subtree of (2) has nodes (3) and (4) from left to right, while the right subtree of (2) has nodes (4) and (3) from right to left.

This mirroring pattern continues throughout the tree. The left and right subtrees are symmetrically arranged with respect to the central vertical line. Therefore, the given binary tree is symmetric.

Optimal:

For a binary tree to be symmetric:

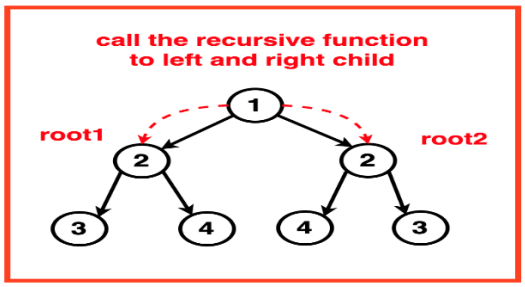
- The root node and its two subtrees (left and right) must have the same value.
- The left subtree of the root should be a mirror image of the right subtree.
- This mirroring should be consistent throughout the entire tree, not just at the root level.

When recursively checking the left and right subtrees for symmetry in a binary tree, the traversals are mirrored. Specifically, the algorithm compares the left child of the left subtree with the right child of the right subtree and the right child of the left subtree with the left child of the right subtree.

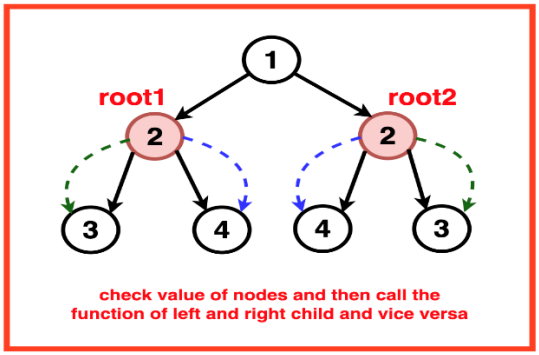
Algorithm:

Step 1: If the tree is empty, it is considered symmetric by default and we return true.

Step 2: If the tree is not empty, we call a utility function `optimal `, passing the left and right subtrees of the root. This utility function handles the recursive checks for symmetry.



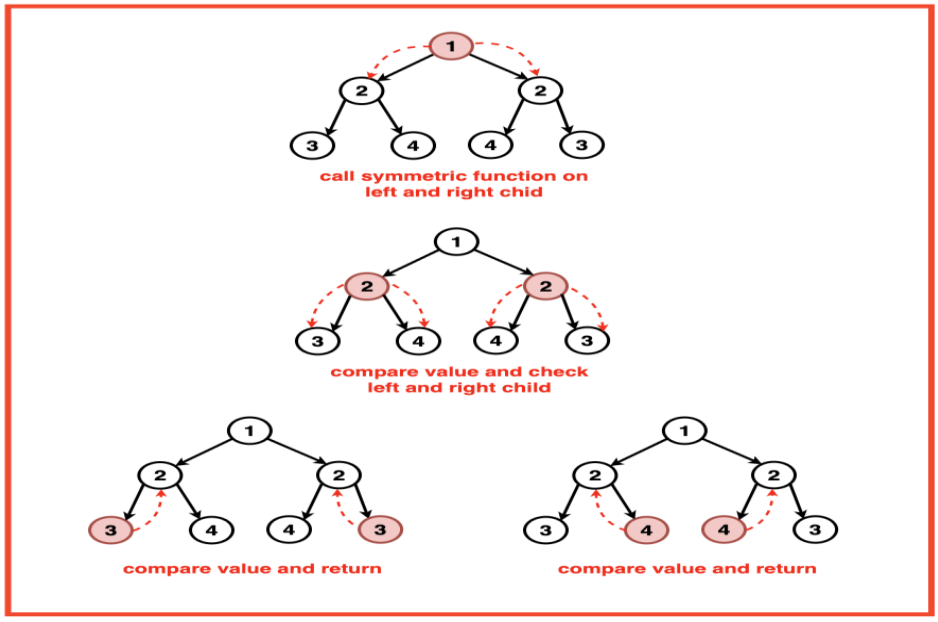
Base Case: The base case for recursion is when both the left and right subtrees are empty, indicating a symmetric structure and we return true. If only one of the subtrees is empty (while the other is not), we return false as this violates the conditions of symmetry.



Check for Symmetry:

- Compare the values of the current nodes from the left and right subtrees. For the binary tree to be symmetric, the corresponding nodes received should have equal values.
- Recursively check the symmetry of these subtrees. We check if the left subtree of the left node is symmetric with the right subtree of the right node.
- Similarly, also check the symmetry of the right subtree of the left node with the left subtree of the right node.

Hence, we compare the node values and recursively explore the left and right subtrees in a mirrored fashion.



Step 3: The final result of the 'optimal' function is based on the outcome of the utility function 'optimal' recursive function for the roots left and right subtree.

Code:

```
static boolean optimal(Node r1, Node r2){  3 usages
    if(r1==null || r2==null){
        if(r1==null && r2==null) return true;
        return false;
    }
    if(r1.data != r2.data) return false;
    return optimal(r1.left,r2.right) && optimal(r1.right,r2.left);
}
```

Time Complexity: O(N) where N is the number of nodes in the Binary Tree. This complexity arises from visiting each node exactly once during the traversal.

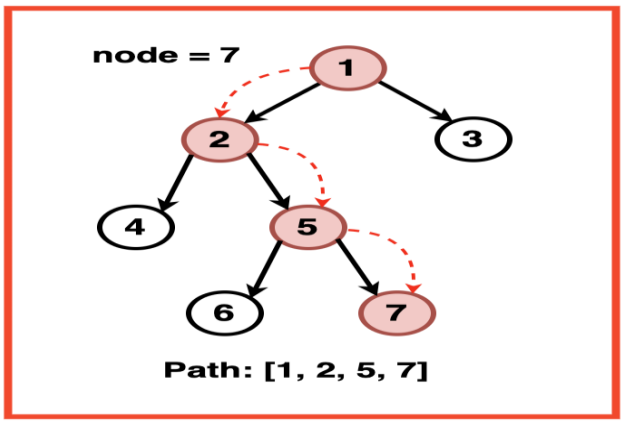
Space Complexity: O(1) as no additional data structures or memory is allocated.

- O(H): Recursive Stack Space, where H: height of tree

Given a Binary Tree and a target node. Return the path from the root node to the given leaf node.

- **No two nodes in the tree have the same data value.**
- **It is assured that the given node is present and a path always exists.**

Input: 1 2 3 4 5 -1 -1 -1 -1, Node: 7



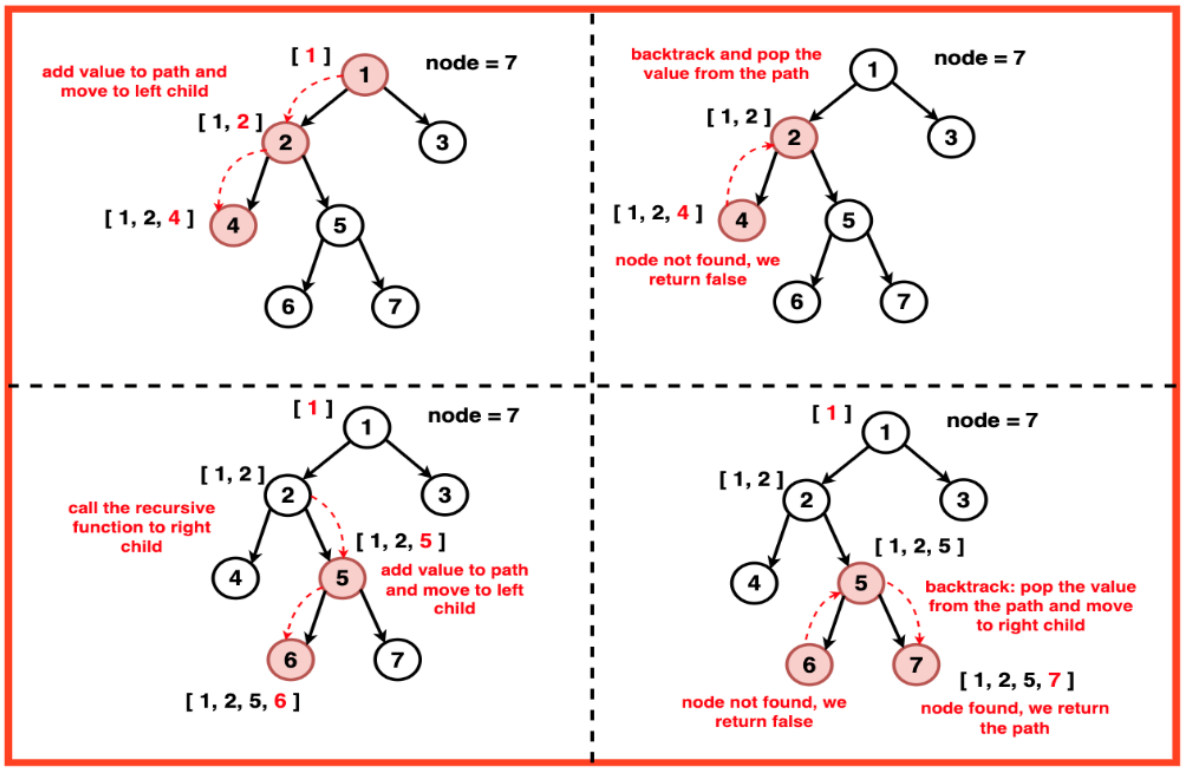
Output: [1, 2, 5, 7]

Optimal

Algorithm:

Step 1: Initialise an empty ArrayList to store the current path.

Step 2: Initialise a recursive function to explore the Binary Tree using Depth First Search. Starting from the root node, we traverse the tree using the inorder sequence.

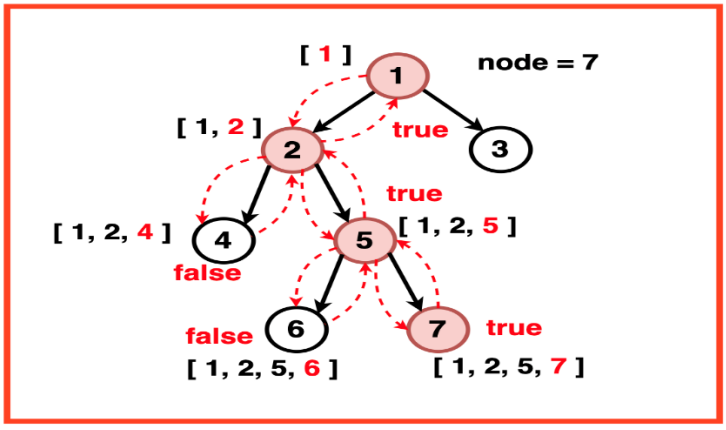


Base Case: If the current node is null then we return false, indicating the end of the path.

Step 3: Recursive Calls:

- During the recursive exploration, the recursive function appends the current node's data value to the Arraylist.

- It checks if the current node's value matches the target value x. If it does, the function returns true, indicating the completion of the path to the target node.
- We then call the function on the left and right children of the current node.



Step 4: Backtracking:

- If the target is not found in either subtree, remove the last element (the current node's value) from the path list to undo the choice (backtracking).
- Return false to indicate that this path does not lead to the target.

Step 5: In the end, we return the ArrayList containing the path from the root to the given node.

Node:

```
static boolean optimal(Node root,int target,List<Integer> res){  3 usages
    if (root == null) return false;
    res.add(root.data);

    if (root.data == target) return true;

    if (optimal(root.left, target, res) || optimal(root.right, target, res))
        return true;

    res.removeLast();
    return false;
}
```

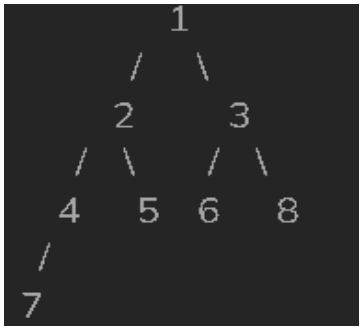
Time Complexity: O(N) where N is the number of nodes in the binary tree as each node of the binary tree is visited exactly once.

Space Complexity: O(1) as no additional data structures or memory is allocated.

- O(H): Recursive Stack Space, where H: height of tree

Find the path from the root to a target node in a binary tree, storing the path in reverse order (target → root).

Input: [1,2,3,4,5,6,8,7], target = 7



Output: [4 2 1]
Explanation: The given target is 7, if we go above the level of node 7, then we find 4, 2 and 1. Hence the ancestors of node 7 are 4 2 and 1

Optimal:

Algorithm

Step 1: Base condition (null node)

- If the current node is null, return false because no path can be found here.

Step 2: Check if current node is target

- If the current node's value equals the target value, return true (target found).

Step 3: Search in the left subtree

- Recursively call the function on the left child.
- If the left call returns true (meaning target found in left subtree):
    - Add the current node's value to the result list (res).
    - Return true to propagate success upward.

Step 4: Search in the right subtree

- If target wasn't found in left, recursively call on the right child.
- If the right call returns true:
    - Add the current node's value to the result list (res).
    - Return true.

Step 5: Not found in either subtree

- If neither left nor right recursive calls found the target, return false.

👉 At the end, res will contain the path from the **target node up to the root** (reverse order).

Code:

```
static boolean optimal(Node root, ArrayList<Integer> res , int target){
    if(root==null) return false;
    if(root.data == target) return true;

    boolean l = optimal(root.left,res,target);
    if(l==true) {
        res.add(root.data);
        return true;
    }
    boolean r = optimal(root.right,res,target);
    if(r==true) {
        res.add(root.data);
        return true;
    }

    return l || r;
}

public ArrayList<Integer> Ancestors(Node root, int target) {
    ArrayList<Integer> res = new ArrayList<>();
    optimal(root,res,target);
    return res;
}
```

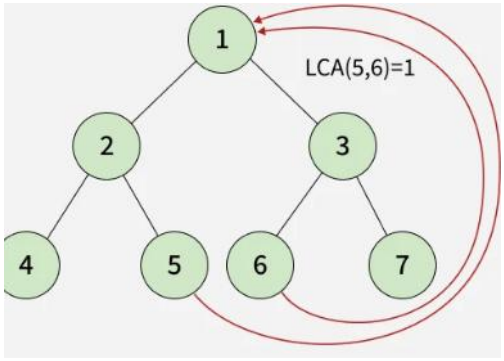
Time Complexity: O(N) where N is the number of nodes in the binary tree as each node of the binary tree is visited exactly once.

Space Complexity: O(1) as no additional data structures or memory is allocated.

- O(H): Recursive Stack Space, where H: height of tree

Given a root binary tree with all unique values and two nodes value, n1 and n2. The task is to find the lowest common ancestor of the given two nodes. We may assume that either both n1 and n2 are present in the tree or none of them are present.
Note: LCA is the first common ancestor of both the nodes n1 and n2 from bottom of tree.

Input: root = [1, 2, 3, 4, 5, 6, 7], n1 = 5, n2 = 6



Output: 1

Brute-Force:

✅ **Implementation Steps**

Part 1: Collect paths for both target nodes

1. Start a DFS (optimal) from the root.
2. Keep a temporary list temp that tracks the current path from root down to the current node.
3. For each visited node:
    - Add it to temp.
    - If the node is either target1 or target2, store a copy of the current path into res.
4. Recursively explore left and right children.
5. After exploring, remove the current node from temp (backtracking).
6. After DFS finishes, res contains exactly two paths:
    - res.get(0) → path from root to p.
    - res.get(1) → path from root to q.

Part 2: Compare the two paths

7. Initialize 'common' as the root (first node of either path).
8. Compare nodes from the beginning of both paths step by step:
    - If nodes at the same index are equal, update common.
    - Stop once paths diverge.
9. Return the last common node as the Lowest Common Ancestor (LCA).

Code:

```java
class Solution {
    static void optimal(TreeNode root, TreeNode target1, TreeNode target2, ArrayList<TreeNode> temp, ArrayList<ArrayList<TreeNode>> res) {
        if (root==null) return;
        temp.add(root);
        if (root == target1 || root == target2)
            res.add(new ArrayList<>(temp));
        optimal(root.left,target1,target2,temp,res);
        optimal(root.right,target1,target2,temp,res);
        temp.removeLast();
    }
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        ArrayList<ArrayList<TreeNode>> res = new ArrayList<>();
        optimal(root,p,q,new ArrayList<>(),res);
        TreeNode common = res.get(0).get(0);
        if (res.get(0).size()<res.get(1).size()){
            for (int i=0;i<res.get(0).size();i++){
                if (res.get(0).get(i) == res.get(1).get(i))
                    common = res.get(0).get(i);
            }
        }
        else {
            for (int i=0;i<res.get(1).size();i++){
                if (res.get(1).get(i) == res.get(0).get(i))
                    common = res.get(1).get(i);
            }
        }
        return common;
    }
}
```

Complexity Analysis = O (N) + O(H)

- **DFS to collect paths**
    - Each node is visited at most once.
    - Time = **O(N)**, where N = number of nodes.
- **Path comparison**
    - Path length is at most the height of the tree (H).
    - Comparing two paths takes **O(H)**.

Space Complexity = O(H) + O(2H) + O(H)

- temp uses at most O(H) space during recursion.
- res stores 2 paths, each up to length H → O(H).
- Recursion stack = O(H).

Optimal:

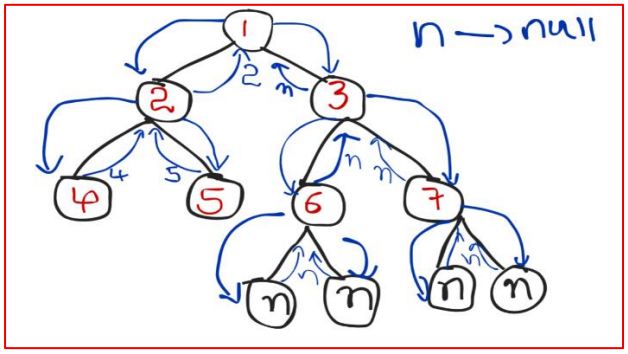
✅ Implementation Steps

1. **Base Case:**
    - If the current node is null, return null (no ancestor here).

- If the current node matches either target1 or target2, return the current node (because one of the targets is found).

2. **Recursive Search:**
   - Recursively search the **left subtree** and store the result in 'l'.
   - Recursively search the **right subtree** and store the result in 'r'.
3. **Analyse Returned Values:**
   - If both 'l' and 'r' are non-null, it means:
     - One target was found in the left subtree, and the other in the right subtree.
     - Therefore, the current node is their Lowest Common Ancestor → return root.
   - If both are null, return null (no target found in this branch).
   - If only one is non-null, return the non-null node (it's either a target itself or an ancestor of the target).

Dry-run (LCA OF 4,5)



Code:

```
static Node optimal(Node root,Node target1,Node target2){
    if (root==null) return null;
    if (root == target1 || root==target2)
        return root;

    Node l = optimal(root.left,target1,target2);
    Node r = optimal(root.right,target1,target2);

    if (l!=null && r!=null) return root;
    else if (l==null && r==null) return null;
    else if (l == null) return r;
    return l;
}
```

Time Complexity: O(N)

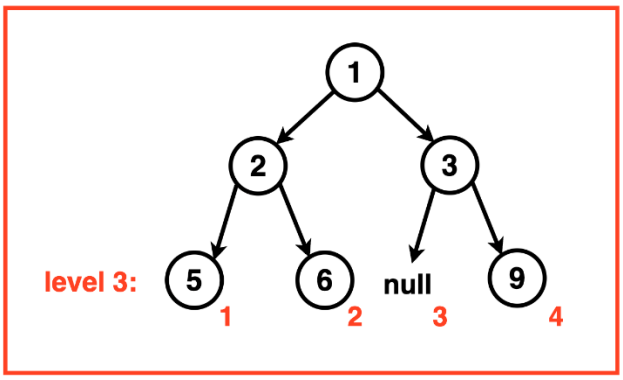
- Each node is visited at most once.
- Total = O(N), where N = number of nodes.

Space Complexity: O(H)

- Recursion stack depth = O(H), where H = height of the tree.

Given the root of a binary tree, return the maximum width of the given tree. The maximum width of a tree is the maximum width among all levels. The width of one level is defined as the length between the end-nodes (the leftmost and rightmost non-null nodes), where the null nodes between the end-nodes that would be present in a complete binary tree extending down to that level are also counted into the length calculation.

Input: [1 2 3 5 6 null 9]



Output: Maximum Width: 4

Explanation: Level 3 is the widest level of the Binary Tree and whose end-to-end width is 4 comprising of nodes: {5, 6, null, 9}.

Brute Force:

Idea: Use complete-binary-tree indexing directly.

Works: Small/moderate trees.

Algorithm

1. **Initialization**
    - You define a Pair class that stores:
        - A reference to a tree node.

        - Its corresponding **index**.
    - Create a queue to perform **level-order traversal (BFS)**.
    - Push the root node with index 0.
2. **Level-wise Processing**
    - While the queue is not empty:
        - Take note of the **index of the first node** in the queue (first).
        - Take note of the **index of the last node** in the queue (last).
        - Compute the **width of the current level** as last - first + 1.
        - Update the maximum width seen so far.
3. **Expanding Nodes**
    - Process all nodes in the current level (loop size times where size = q.size()):
        - Remove a node from the queue along with its index.
        - If it has a left child, assign it index 2*index + 1 and push into the queue.
        - If it has a right child, assign it index 2*index + 2 and push into the queue.

*(These indices are based on the "binary tree as an array" concept, where for a node at index i, left = 2*i+1 and right = 2*i+2).*

4. **Termination**
    - Continue until all levels are processed.
    - The stored width will contain the **maximum width across all levels**.
    - Return that value.

Code:

```java
class Pair{
    TreeNode root;
    int index;
    Pair(TreeNode root,int index){
        this.root = root;
        this.index = index;
    }
}
class Solution {
    public int widthOfBinaryTree(TreeNode root) {
        int width = 1;
        Queue<Pair> q = new LinkedList<>();
        q.add(new Pair(root,0));
        while (!q.isEmpty()){
            int first = ((LinkedList<Pair>) q).peekFirst().index;
            int last = ((LinkedList<Pair>) q).peekLast().index;
            width = Math.max(width,last-first+1);
            int s = q.size();
            for (int i = 0; i < s; i++){
                Pair p = q.remove();
                TreeNode temp = p.root; int index = p.index;
                if (temp.left != null)  q.add(new Pair(temp.left,2*index+1));
                if (temp.right != null) q.add(new Pair(temp.right,2*index+2));
            }
        }
        return width;
    }
}
```

Time Complexity = O(N)

- Each node is inserted into the queue **once** and removed **once**.
- Operations like peekFirst(), peekLast(), and adding/removing from the queue are **O(1)**.
- Therefore, the total time complexity is: O(N), where N is the number of nodes in the tree.

Space Complexity = O(N)

- The queue can store at most all nodes in a level.
- In the worst case (a complete binary tree), this is about N/2 nodes.
- So, the auxiliary space is: O(N)

Limitation

- Because we are directly using indices (2*index+1, 2*index+2) with an int, for a deep tree (depth > 31), these indices can overflow.
- So, while the time/space complexity is fine, correctness may break if node count is very large (close to 10^5 and depth is skewed).

Optimal:

Idea: Use indexing, but **normalize each level** to avoid overflow.

Works: Both small and very deep trees.

Algorithm:

Step 1: Initialize a variable `width` to store the maximum width.

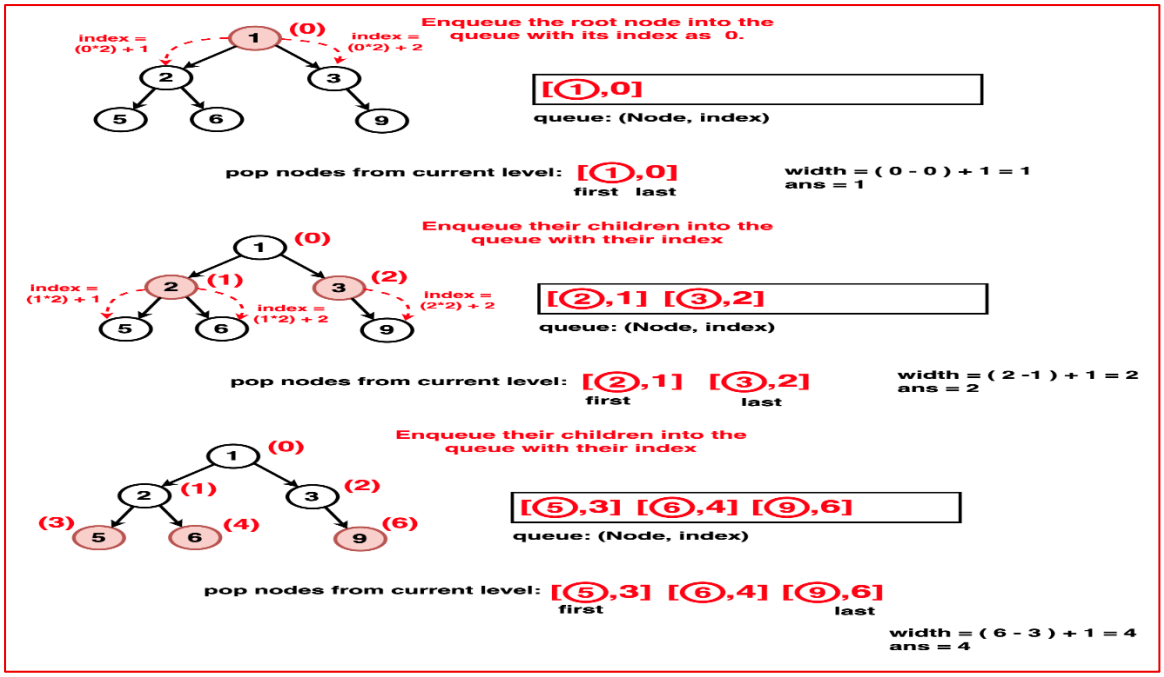
Step 2: Create a queue to perform level-order traversal and each element of this queue would be a pair containing a node and its vertical index. Push the root node and its position (initially 0) into the queue.

Step 3: While the queue is not empty, perform the following steps:

- Get the number of nodes at the current level (size).
- Get the position of the front node in the current level which is the leftmost minimum index at that level.
- Initialize variables first and last to store the first and last positions of nodes in the current level.

Step 4: Backtracking: For each node in the current level:

- Calculate the current position relative to the minimum position in the level.
- Get the current node (node) from the front of the queue.
- If this is the first node in the level, update the first variable.
- If this is the last node in the level, update the last variable.
- Enqueue the left child of the current node with index: (2 x current index – 1).
- Enqueue the right child of the current node with index: (2 x current index + 1).



Step 5: Update the maximum 'width' by calculating the difference between the first and last positions, and adding 1.

Step 6: Repeat the level-order traversal until all levels are processed. The final value of `width` represents the maximum width of the binary tree, return it.

Code:

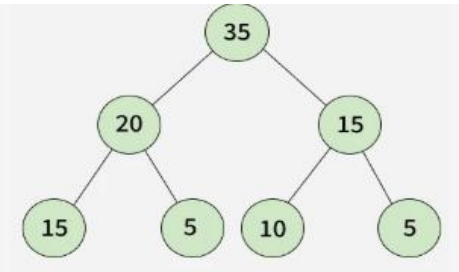
```java
class Pair{
    TreeNode root;
    int index;
    Pair(TreeNode root,int index){
        this.root = root;
        this.index = index;
    }
}
class Solution {
    public int widthOfBinaryTree(TreeNode root) {
        int width = 1;
        Queue<Pair> q = new LinkedList<>();
        q.add(new Pair(root,0));
        while (!q.isEmpty()){
            int size = q.size();
            int min = q.peek().index;
            int first=0, last=0;
            for (int i = 0;i<size;i++){
                Pair p = q.remove();
                int curr_ind = p.index - min;

                if (i==0) first = curr_ind;
                if (i==size-1) last = curr_ind;
                if (p.root.left != null)  q.add(new Pair(p.root.left,2*curr_ind+1));
                if (p.root.right != null) q.add(new Pair(p.root.right,2*curr_ind+2));
            }
            width = Math.max(width,last-first+1);
        }
        return width;
    }
}
```

Time Complexity: O(N) where N is the number of nodes in the binary tree. Each node of the binary tree is enqueued and dequeued exactly once, hence all nodes need to be processed and visited. Processing each node takes constant time operations which contributes to the overall linear time complexity.

Space Complexity: O(N) where N is the number of nodes in the binary tree. In the worst case, the queue has to hold all the nodes of the last level of the binary tree, the last level could at most hold N/2 nodes hence the space complexity of the queue is proportional to O(N).

Given the root of a binary tree, determine whether the tree satisfies the Children Sum Property. In this property, each non-leaf node must have a value equal to the sum of its left and right children's values. A NULL child is considered to have a value of 0, and all leaf nodes are considered valid by default.
Return true if every node in the tree satisfies this condition, otherwise return false.

Input: root = [35, 20, 15, 15, 5, 10, 5]



Output: True

Explanation: Here, every node is sum of its left and right child.

Optimal:

1. Base Case 1:

- If root == null, return true (empty tree satisfies the property).

2.Base Case 2:

- If root is a leaf node return true.

3.Recursive Step:

- Recursively check left subtree: l = isSumProperty(root.left)
- Recursively check right subtree: r = isSumProperty(root.right)

4. Compute Sum of Children:

- Initialize sum = 0
- If root.left exists, add root.left.data to sum
- If root.right exists, add root.right.data to sum

5.Check Current Node:

- If sum != root.data, return false (property violated at this node).

6.Combine Results:

- Return l && r (property holds only if both subtrees satisfy it)

Code:

```java
static Boolean optimal(Node root){  3 usages
    if (root==null) return true;
    if (root.left == null && root.right==null) return true;

    boolean l = optimal(root.left);
    boolean r = optimal(root.right);

    int sum = 0;
    if (root.right != null) sum += root.right.data;
    if (root.left != null) sum += root.left.data;
    if (sum != root.data) return false;

    return l && r;
}
```

Time Complexity: O(n)

- Each node is visited exactly once.

Space Complexity: O(h)

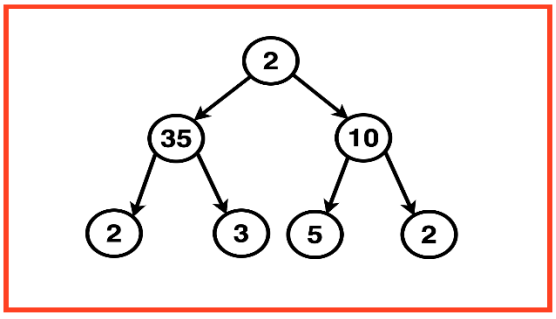
- Due to recursion stack; h = height of tree (O(log n) balanced, O(n) skewed).

Given a Binary Tree, convert the value of its nodes to follow the Children Sum Property. The Children Sum Property in a binary-tree states that for every node, the sum of its children's values (if they exist) should be equal to the node's value. If a child is missing, it is considered as having a value of 0.

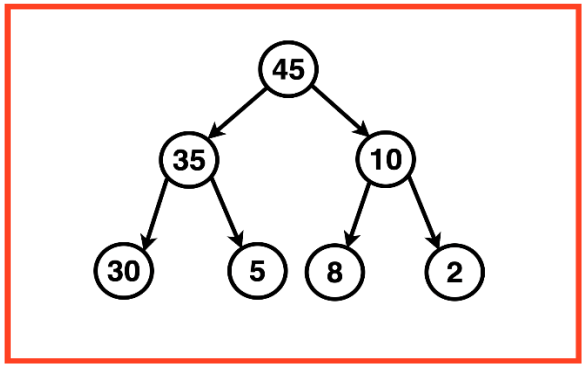
Note:

- The node values can be increased by any positive integer any number of times, but decrementing any node value is not allowed.
- A value for a NULL node can be assumed as 0.
- We cannot change the structure of the given binary tree.

Input:

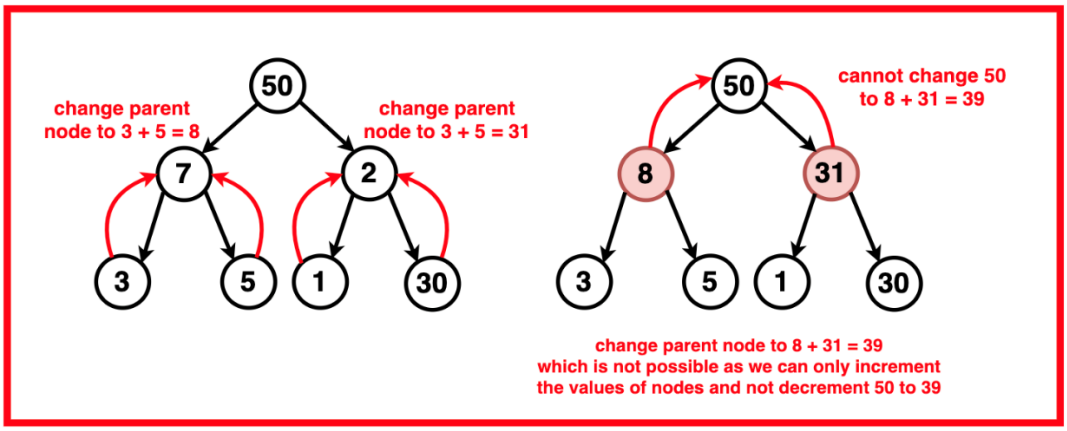


Output:

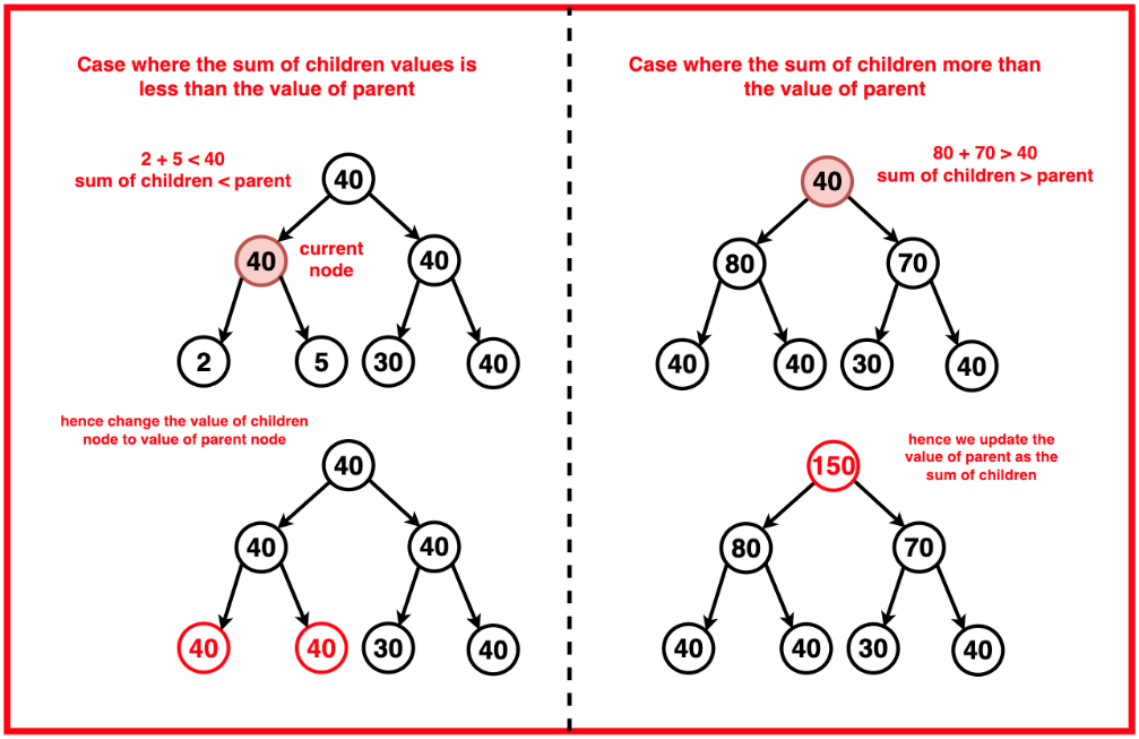


Optimal:

The constraint is that we cannot decrease the value of any node, only increase it. Also, the structure of the binary tree cannot be changed. If we follow a bottom-up approach and try to adjust parent values based on children, we may reach a situation where the sum of children exceeds the parent's value, requiring us to decrease the parent's value, which is not allowed.



With just a bottom-up approach, we cannot guarantee that the Children Sum Property will be satisfied at each level. It might work for some cases but not for all. There's no clear strategy to ensure that the property holds true for the entire tree. A key insight here is that there's no restriction on how much we can increase the value of each node. Hence, we have the flexibility to adjust the values as needed to ensure that the Children Sum Property holds true at every node in the tree.



This means that if the sum of the values of a node's children is less than the node's value, we can simply increase the values of the children (and potentially the grandchildren and so on) until the property is satisfied. Using recursive calls, we traverse the binary tree, addressing each node and its children iteratively. At each step, we calculate the sum of the children's values and compare it with the parent node's value.

Algorithm:

1.Base Case:

- If root == null, return.
- If root is a leaf node, return.

2.Compute Sum of Children:

- Initialize child = 0.
- If root.left != null, add root.left.data to child.
- If root.right != null, add root.right.data to child.

3.Adjust Children or Root:

- If child < root.data:
    - If root.left != null, set root.left.data = root.data.
    - If root.right != null, set root.right.data = root.data.
- Else (child ≥ root.data):

- Set root.data = child.

4.Recursive Calls:

- Call optimal(root.left)
- Call optimal(root.right)

5.Fix Root After Recursion:

- Compute sum = 0
  - If root.left != null, add root.left.data
  - If root.right != null, add root.right.data
- Set root.data = sum

Code:

```
static void optimal(Node root){  3 usages
    if (root==null) return;
    if (root.left == null && root.right == null) return;

    int child = 0;
    if (root.left != null)  child += root.left.data;
    if (root.right != null)  child += root.right.data;

    if (child < root.data){
        if (root.left != null) root.left.data = root.data;
        if (root.right != null) root.right.data = root.data;
    }
    else root.data = child;

    optimal(root.left);
    optimal(root.right);

    int sum = 0;
    if (root.left != null) sum += root.left.data;
    if (root.right != null) sum += root.right.data;
    root.data = sum;
}
```

Time Complexity: O(n)

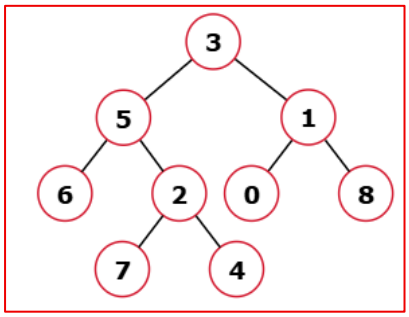
- Each node is visited exactly once.

Space Complexity: O(h)

- Due to recursion stack; h = height of tree (O(log n) balanced, O(n) skewed).

Given the root of a binary tree, a 'target' node, and an integer k. Return an arrayList of the values of all nodes that have a distance k from the target node. The answer can be returned in any order.

Input: root = [3, 5, 1, 6, 2, 0, 8, N, N, 7, 4], target = 5, k = 2



Output: [7,4,1]

Explanation: The nodes that are a distance 3 from the target node (with value 5) have values 7, 4 and 1.

Optimal:

In a binary tree, each node only has references to its children. To find all nodes at a specific distance K from a given target node, we must be able to move both **downward** (to children) and **upward** (to parent). Since the binary tree doesn't store parent links, we simulate this by converting the tree into an **undirected graph**. This allows traversal in all directions: left, right, and parent. Once we treat the tree as a graph, we perform a standard **Breadth-First Search (BFS)** starting from the target node and collect all nodes that are exactly K steps away.

Algorithm:

- **Build the Parent Map:**
  Traverse the entire binary tree using either BFS or DFS and map each node to its parent node. This step effectively adds upward connectivity to the tree, allowing access to parent nodes during traversal.
    - Create a Map(parentTrack) where each key is a child node and its value is the corresponding parent node.
    - This transforms the binary tree into an undirected structure where each node can move left, right, or up.
- **Perform BFS from the Target Node:**
  Begin a level-order (BFS) traversal from the target node to explore all nodes at increasing distances.
    - Initialize a queue with the target node and a set to track visited nodes.

- For each node at the current BFS level:
    - Visit the left child if it exists and hasn't been visited.
    - Visit the right child if it exists and hasn't been visited.
    - Visit the parent (from the parentTrack) if it exists and hasn't been visited.
- Repeat the process level by level until distance K is reached.

- **Store the Result:** When the BFS reaches distance K, all nodes remaining in the queue are exactly K edges away from the target node.
    - Store the values of these nodes in a ArrayList(result) and return it.

Code:

```
static void markParents(Node root,Map<Node,Node> parentTrack){
    Queue<Node> q = new LinkedList<>();
    q.add(root);
    parentTrack.put(root,null);
    while (!q.isEmpty()){
        int size = q.size();
        for (int i=0; i<size; i++){
            Node temp = q.remove();
            if (temp.left != null){
                parentTrack.put(temp.left,temp);
                q.add(temp.left);
            }
            if (temp.right != null){
                parentTrack.put(temp.right,temp);
                q.add(temp.right);
            }
        }
    }
}
```

```java
static List<Integer> optimal(Node root,Node target,int k){
    Map<Node,Node> parentTrack = new HashMap<>();
    markParents(root,parentTrack);

    Queue<Node> q = new LinkedList<>();
    Map<Node, Boolean> visited = new HashMap<>();
    int currDist = 0;
    q.add(target);
    visited.put(target,true);
    while (!q.isEmpty()){
        int size = q.size();
        if (currDist == k) break;
        for (int i=0; i < size; i++){
            Node temp = q.remove();

            Node upward = parentTrack.get(temp);
            if(upward != null && !visited.containsKey(upward)){
                q.add(upward);
                visited.put(upward,true);
            }
            if (temp.left != null && !visited.containsKey(temp.left)){
                q.add(temp.left);
                visited.put(temp.left,true);
            }
            if (temp.right != null && !visited.containsKey(temp.right)){
                q.add(temp.right);
                visited.put(temp.right,true);
            }
        }
        currDist += 1;
    }
    List<Integer> res = new ArrayList<>();
    while (!q.isEmpty()){
        Node temp = q.remove();
        res.add(temp.data);
    }
    return res;
}
```

Time Complexity: O(2n) where n = number of nodes.

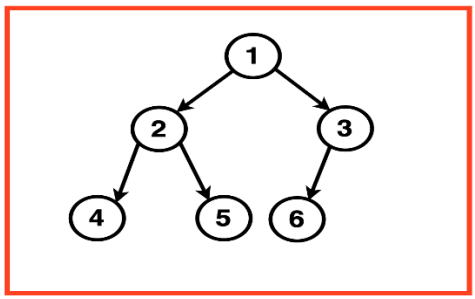
- O(n) for BFS to mark parents.
- O(n) for BFS from target.

Space Complexity: O(5n) where n = number of nodes.

- O(n) for parent map.
- O(n) for visited map.
- O(2n) for BFS queue as we are doing level order traversal 2 times one in the optimal() & another in the markParents().
- O(n) for ArrayList(res) in worst case it can store all the nodes.

Given a Complete Binary Tree, count and return the number of nodes in the given tree. A Complete Binary Tree is a binary tree in which all levels are completely filled, except possibly for the last level, and all nodes are as left as possible.

Input: 1 2 3 4 5 6



Output: 6

Explanation: There are 6 nodes in this Binary Tree.

Brute-force:

A brute force approach would be to traverse the tree using In-order (or any) traversal and count the number of nodes as we are traversing the tree. In In-order traversal, we visit the left subtree first, then the current node, and finally the right subtree. By incrementing the counter for each visited node, we effectively count all nodes in the binary tree.

Algorithm

Step 1: Base Case

- If root is null, return 0 (An empty tree has zero nodes).

Step 2: Recursive Case

- Recursively count the number of nodes in the **left**
- Recursively count the number of nodes in the **right subtree**

Step 3: Compute Total for Current Node

- Total nodes for the current subtree = 1 + leftCount + rightCount
    - 1 account for the current root node.

Step 4: Return the Result

- Return the total nodes calculated in Step 3 to the caller.

Code:

```java
static int bruteForce(Node root){  3 usages
    if (root == null) return 0;

    int leftCount = bruteForce(root.left);
    int rightCount = bruteForce(root.right);

    return 1 + leftCount + rightCount;
}
```

Time Complexity = O(n)

- Each node is visited **exactly once** → **O(n)**, where n is the number of nodes.

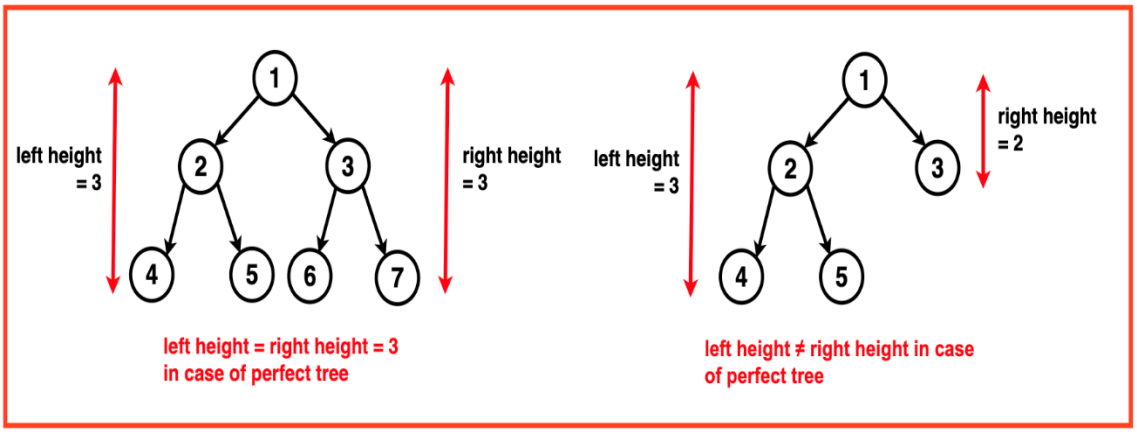
Space Complexity = O(h)

- Recursive stack space = **O(h)**, where h is the height of the tree.

Optimal:

Given that the binary is a complete binary tree, we can exploit its properties to optimise the algorithm and achieve a better time complexity. In a complete binary tree, the last level may not be completely filled, but the nodes are positioned from left to right. This property allows us to determine the number of nodes using just the height. The relationship between the height of the binary tree (h) and the maximum number of nodes it can have, denoted by the formula: Maximum Number of Nodes: 2^h-1.

If the last level of a binary tree is perfectly filled, known as a perfect binary tree, the count of nodes can be determined by the formula: 2h-1, where h is the height. To check if the last level of the Binary Tree is filled or not, we can compare the left and right heights of the tree.



- If the left height equals right height, it indicates that the last level is completely filled.

- If the left height does not equal right height, the last level is not completely filled.

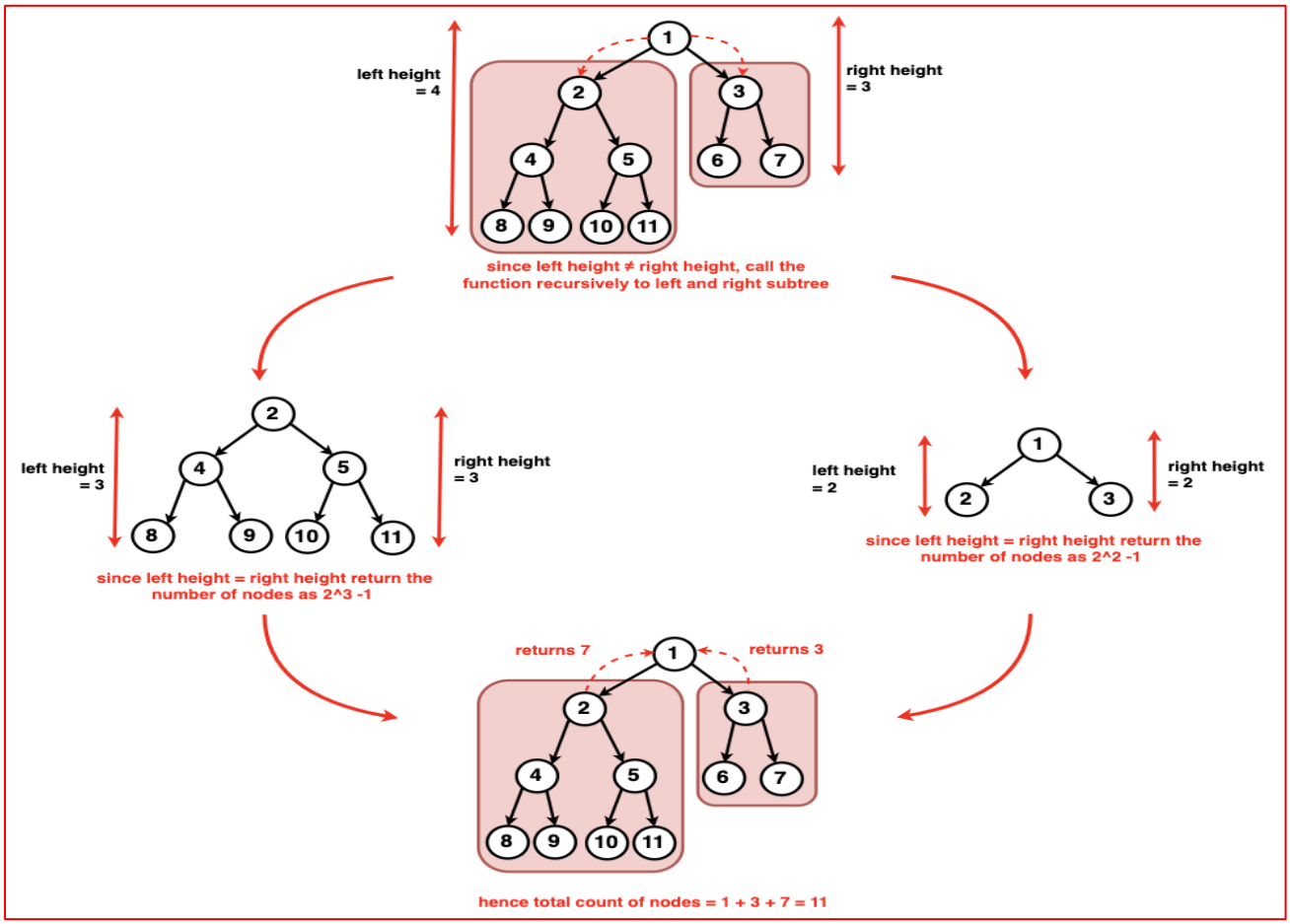
In the case where left height and right height differ, we can employ a recursive approach. We recursively calculate the number of nodes in the left subtree and in the right subtree, and then return the total count as 1 + leftNodes + rightNodes. If the height of the left subtree is equal to the height of the right subtree, we can directly calculate using the 2h-1 formula.

Algorithm:

Step 1: Base Case If the given node is null, we return 0.

Step 2: Recursive Calls: Recursively find the left height and right height of the Binary Tree.

Step 3: Comparison: If the left height is equal to the right height implies that the tree's last level is completely filled. Return the count of nodes using the formula: return (2^lh) – 1.

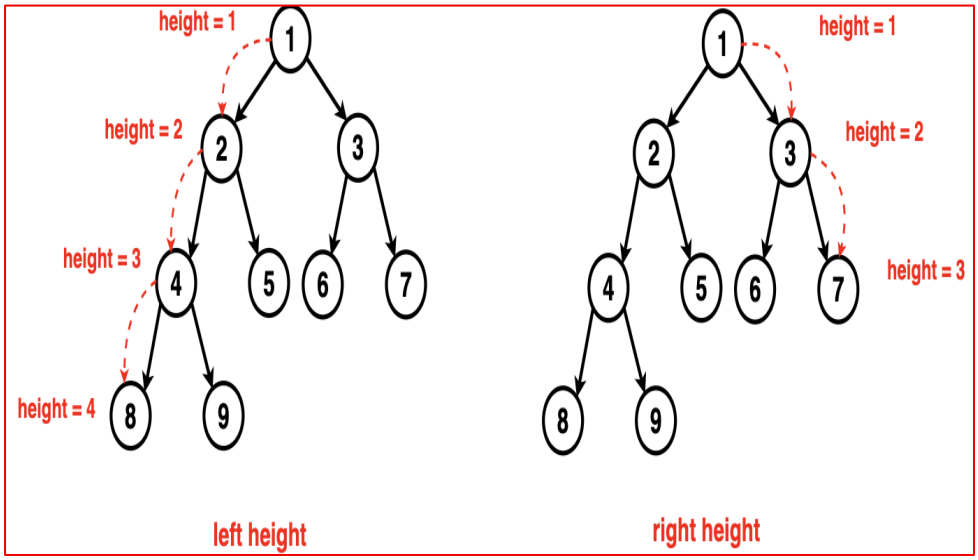


Step 4: If the left height is not equal to the right height implies that the tree's last level is not completely filled. Recursively call the function to the left and right subtree and return the final number of nodes as 1 + countNodes(root->left) + countNodes(root->right).

Step 5: Implement the find left height and right height functions.

- Start with the variable height set to 0.

- Use a while loop to traverse the left/right side of the tree incrementing the height until reaching a leaf node.
- Return the calculated height.



Code:

```
static int leftHeight(TreeNode root){
    int height = 0;
    while (root != null){
        height += 1;
        root = root.left;
    }
    return height;
}
static int rightHeight(TreeNode root){
    int height = 0;
    while (root != null){
        height += 1;
        root = root.right;
    }
    return height;
}
public int countNodes(TreeNode root) {
    if (root == null) return 0;

    int leftHeight = 1 + leftHeight(root.left);
    int rightHeight = 1 + rightHeight(root.right);

    if (leftHeight == rightHeight) return (int)Math.pow(2,leftHeight) - 1;

    int l = countNodes(root.left);
    int r = countNodes(root.right);
    return 1+l+r;
}
```

Time Complexity: O(log N * log N) where N is the number of nodes in the Binary Tree.

- The calculation of leftHeight and rightHeight takes O (log N) time.

- In the worst case, when encountering the second case (leftHeight != rightHeight), the recursive calls are made at most log N times (the height of the tree).

Space Complexity: O(H) where H is the height of tree.

- The space complexity is determined by the maximum depth of the recursion stack, which is equal to the height of the binary tree.
- Since the given tree is a complete binary tree, the height will always be log N.
- Therefore, the space complexity is O (log N).

Can we create a unique Binary tree with pre-order & post-order traversal?

Answer: No (except for full binary trees).

Reason:

In preorder (Root → Left → Right) and post-order (Left → Right → Root), we know:

- The first element of preorder = root.
- The last element of post-order = root.

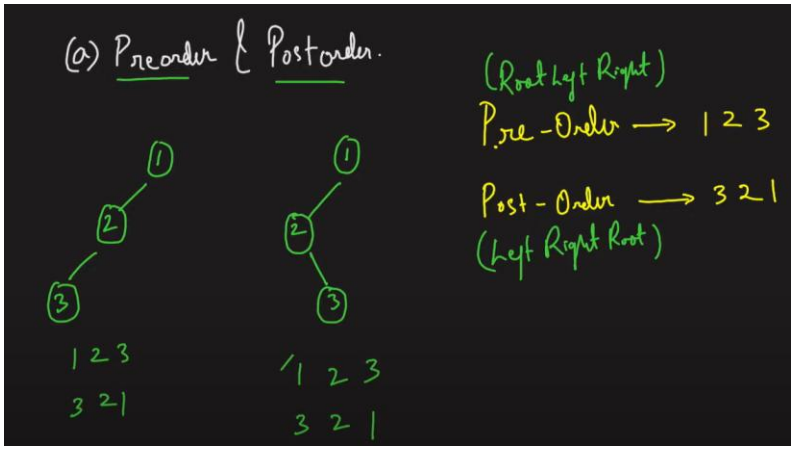
But… how do we know **where the left subtree ends and the right subtree begins**?
Without in-order traversal, that boundary is missing.

So, if a node has only **one child**, we can't know if it was a **left child** or a **right child**.
That's why multiple trees can give the same preorder & post-order.

Why it works for Full Binary Trees?

👉 A full binary tree means:

- Every node has either 0 or 2 children (never exactly 1).
- If a node has children, it must have two.
- So, once you locate the left child (using preorder), the right child is guaranteed to exist.
- That allows you to uniquely partition nodes between left and right subtrees.



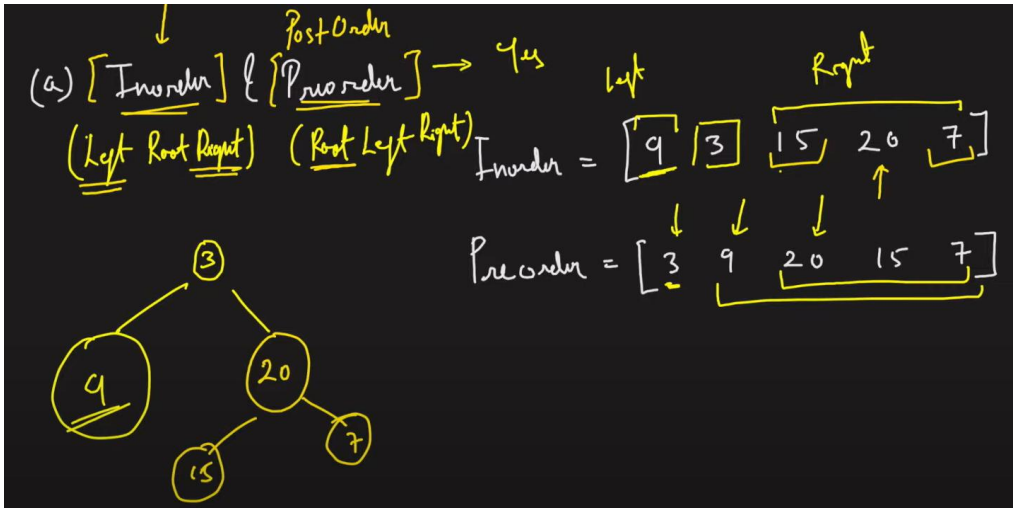
Can we create a unique Binary tree with in-order & post-order or in-order & pre-order traversal?

Answer: Yes

Reason:

- Preorder gives you the root (first element) & Post-order gives you the root (last element).
- In in-order, you can locate that root, which splits the sequence into left subtree and right subtree.
- Recurse on each half.

Since the split is clear, the tree is uniquely determined.

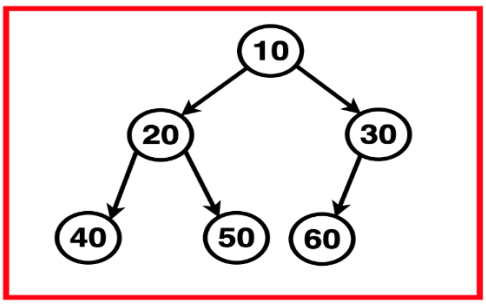


Given two arrays representing the inorder and preorder traversals of a binary tree, your task is to construct the binary tree and return its root.

Note: The inorder and preorder traversals contain unique values.

Input: Inorder: [40, 20, 50, 10, 60, 30], Preorder: [10, 20, 40, 50, 30, 60]

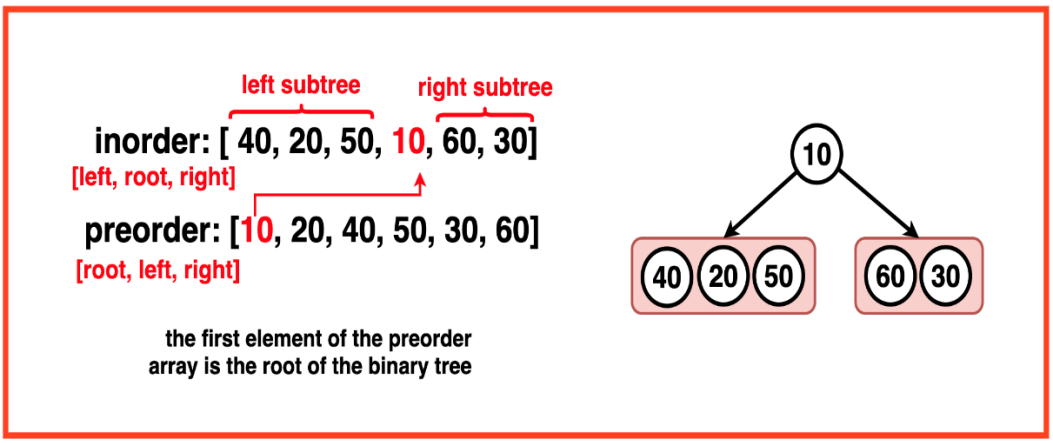
Output:



Explanation: The unique Binary Tree hence created has inorder traversal: [40, 20, 50, 10, 60, 30] and preorder traversal: [10, 20, 40, 50, 30, 60].

Optimal:

In-order traversal allows us to identify a node and its left and right subtrees, while preorder traversal ensures we always encounter the root node first. Leveraging these properties, we can uniquely construct a binary tree. The core of our approach lies in a recursive algorithm that creates one node at a time. We locate this root node in the in-order traversal, which splits the array into the left and right subtrees.

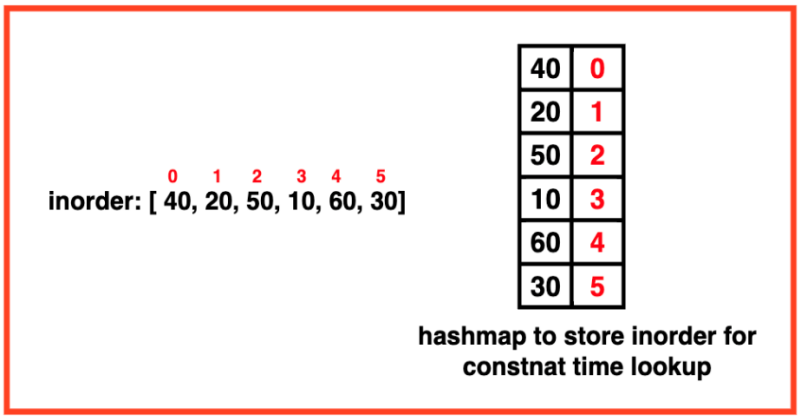


The in-order array keeps getting divided into left and right subtrees hence to avoid unnecessary array duplication, we use variables (inStart, inEnd) and (preStart, preEnd) on the inorder and preorder array respectively. These variables effectively define the boundaries of the current subtree within the original inorder and preorder traversals. Everytime we encounter the root of a subtree via preorder traversal, we locate its position in

the inorder array to get the left and right subtrees. So, to save complexity on the linear look up, we employ a hashmap to store the index of each element in the inorder traversal. This transforms the search operation into a constant-time lookup.

Algorithm:

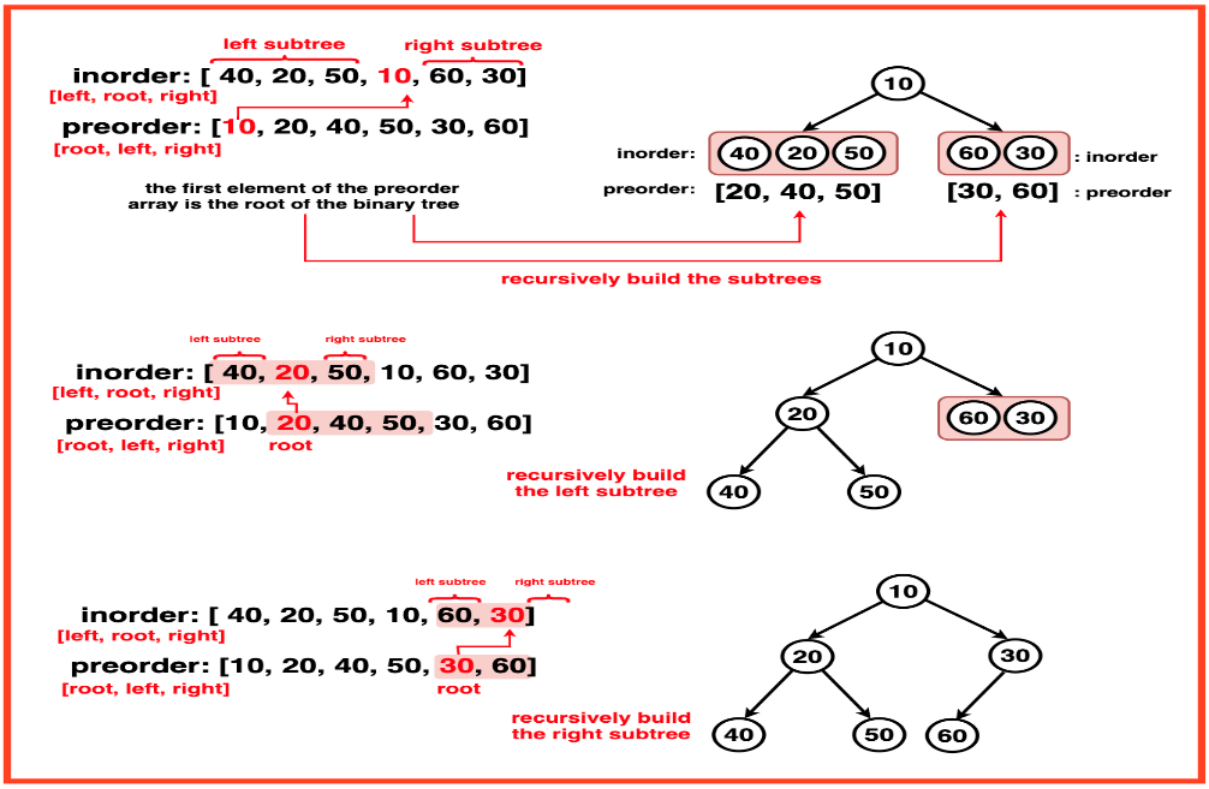
Step 1: Create an empty map to store the indices of elements in the inorder traversal. Iterate through each element in the inorder traversal and store its index in the map (inMap) using the element as the key and its index as the value.



Step 2: Create a recursive helper function `construction ` with the following parameters:

- Preorder array.
- Start index of preorder (preStart), initially set to 0
- End index of preorder (preEnd), initially set to preorder.size() - 1.
- Inorder array.
- Start index of inorder (inStart), initially set to 0.
- End index of inorder (inEnd), initially set to inorder.size() - 1.
- Map for efficient root index lookup in the inorder traversal.

Step 3: Base Case: Check if preStart is greater than preEnd or inStart is greater than inEnd. If true, return NULL, indicating an empty subtree/node.



Step 4: The root node for the current subtree is the first element in the preorder traversal (preorder[preStart]). Find the index of this root node in the inorder traversal using the map (inMap[rootValue]). This is the rootIndex.

Step 5: Hence, the left subtree ranges from inStart to rootIndex. Subtracting these indexes gives us the leftSubtreeSize.

Step 6: Make two recursive calls to 'construction' to build the left and right subtrees: For the left subtree:

- Update preStart to preStart + 1 (moving to the next element in preorder)
- Update preEnd to preStart + leftSubtreeSize (end of left subtree in preorder)
- Update inEnd to rootIndex - 1 (end of left subtree in inorder)

For the right subtree:

- Update preStart to preStart + leftSubtreeSize + 1 (moving to the next element after the left subtree)
- Update preEnd to the original preEnd (end of right subtree in preorder)
- Update inStart to rootIndex + 1 (start of right subtree in inorder)

Step 7: Return the root node constructed for the current subtree. The function returns the root of the entire binary tree constructed from the preorder and inorder traversals.

Code:

```
static Node construction(int[] inorder, int inStart, int inEnd, Map<Integer,Integer> inMap,int[] preorder, int preStart,int preEnd)
{
    if (inStart>inEnd || preStart>preEnd) return null;

    Node root = new Node(preorder[preStart]);
    int inRoot = inMap.get(root.data);
    int numsLeft = inRoot - inStart;

    root.left = construction(inorder,inStart, inEnd: inRoot-1,inMap,preorder, preStart: preStart+1, preEnd: preStart+numsLeft);
    root.right = construction(inorder, inStart: inRoot+1,inEnd,inMap,preorder, preStart: preStart+numsLeft+1,preEnd);
    return root;
}

static Node optimal(int[] inorder, int[] preorder){  1 usage
    HashMap<Integer,Integer> inMp = new HashMap<>();
    for(int i=0;i<inorder.length;i++)
        inMp.put(inorder[i],i);

    return construction(inorder, inStart: 0, inEnd: inorder.length-1,inMp,preorder, preStart: 0, preEnd: preorder.length-1);
}
```

Time Complexity: O(2n)

- **Building the map** -> Runs once for all elements → O(n).
- **Recursive construction**
    - Each recursive call creates one node.
    - Each node is created exactly once (since preorder is traversed sequentially).
    - For every node, we do:
    - Constant time lookup in the map → O (1).
    - Constant amount of arithmetic + recursion setup.
    - So, total across all nodes → O(n).

Space Complexity: O(n) + O(h)

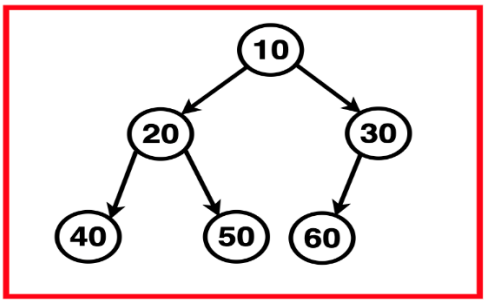
1. **HashMap** → stores indices of all nodes → O(n).
2. **Recursive call stack**: O(h) where h is the height of tree.

Given two arrays representing the inorder and preorder traversals of a binary tree, your task is to construct the binary tree and return its root.

Note: The inorder and preorder traversals contain unique values.

Input: Inorder: [40, 20, 50, 10, 60, 30], Preorder: [10, 20, 40, 50, 30, 60]

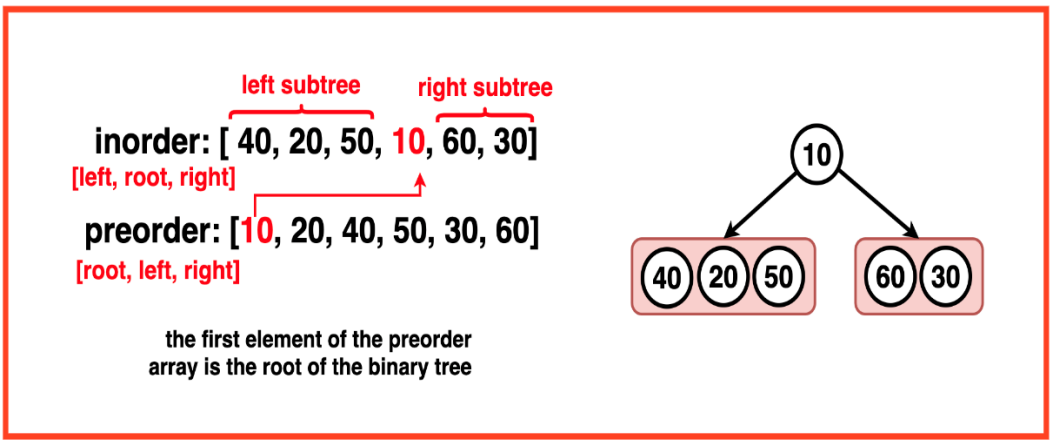
Output:



Explanation: The unique Binary Tree hence created has inorder traversal: [40, 20, 50, 10, 60, 30] and preorder traversal: [10, 20, 40, 50, 30, 60].

Optimal:

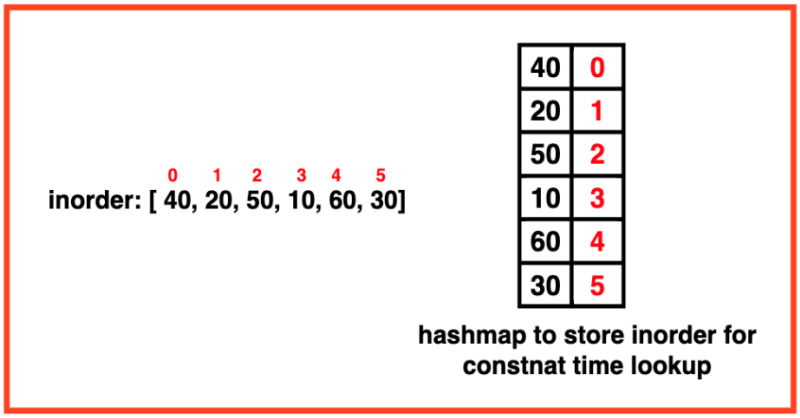
In-order traversal allows us to identify a node and its left and right subtrees, while preorder traversal ensures we always encounter the root node first. Leveraging these properties, we can uniquely construct a binary tree. The core of our approach lies in a recursive algorithm that creates one node at a time. We locate this root node in the in-order traversal, which splits the array into the left and right subtrees.



The in-order array keeps getting divided into left and right subtrees hence to avoid unnecessary array duplication, we use variables (inStart, inEnd) and (preStart, preEnd) on the inorder and preorder array respectively. These variables effectively define the boundaries of the current subtree within the original inorder and preorder traversals. Everytime we encounter the root of a subtree via preorder traversal, we locate its position in the inorder array to get the left and right subtrees. So, to save complexity on the linear look up, we employ a hashmap to store the index of each element in the inorder traversal. This transforms the search operation into a constant-time lookup.

Algorithm:

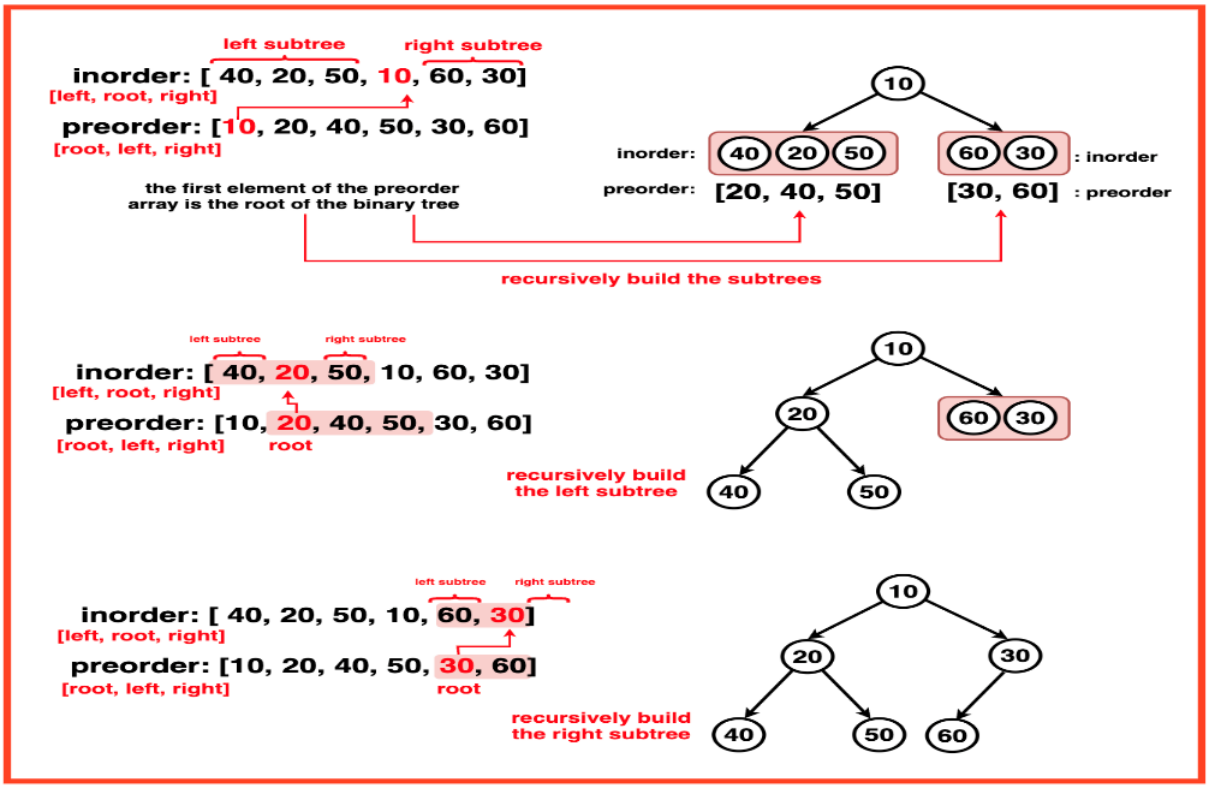
Step 1: Create an empty map to store the indices of elements in the inorder traversal. Iterate through each element in the inorder traversal and store its index in the map (inMap) using the element as the key and its index as the value.



Step 2: Create a recursive helper function `construction ` with the following parameters:

- Preorder array.
- Start index of preorder (preStart), initially set to 0
- End index of preorder (preEnd), initially set to preorder.size() - 1.
- Inorder array.
- Start index of inorder (inStart), initially set to 0.
- End index of inorder (inEnd), initially set to inorder.size() - 1.
- Map for efficient root index lookup in the inorder traversal.

Step 3: Base Case: Check if preStart is greater than preEnd or inStart is greater than inEnd. If true, return NULL, indicating an empty subtree/node.



Step 4: The root node for the current subtree is the first element in the preorder traversal (preorder[preStart]). Find the index of this root node in the inorder traversal using the map (inMap[rootValue]). This is the rootIndex.

Step 5: Hence, the left subtree ranges from inStart to rootIndex. Subtracting these indexes gives us the leftSubtreeSize.

Step 6: Make two recursive calls to 'construction' to build the left and right subtrees: For the left subtree:

- Update preStart to preStart + 1 (moving to the next element in preorder)
- Update preEnd to preStart + leftSubtreeSize (end of left subtree in preorder)
- Update inEnd to rootIndex - 1 (end of left subtree in inorder)

For the right subtree:

- Update preStart to preStart + leftSubtreeSize + 1 (moving to the next element after the left subtree)
- Update preEnd to the original preEnd (end of right subtree in preorder)
- Update inStart to rootIndex + 1 (start of right subtree in inorder)

Step 7: Return the root node constructed for the current subtree. The function returns the root of the entire binary tree constructed from the preorder and inorder traversals.

Code:

```
static Node construction(int[] inorder, int inStart, int inEnd, Map<Integer,Integer> inMap,int[] preorder, int preStart,int preEnd)
{
    if (inStart>inEnd || preStart>preEnd) return null;

    Node root = new Node(preorder[preStart]);
    int inRoot = inMap.get(root.data);
    int numsLeft = inRoot - inStart;

    root.left = construction(inorder,inStart, inEnd: inRoot-1,inMap,preorder, preStart: preStart+1, preEnd: preStart+numsLeft);
    root.right = construction(inorder, inStart: inRoot+1,inEnd,inMap,preorder, preStart: preStart+numsLeft+1,preEnd);
    return root;
}

static Node optimal(int[] inorder, int[] preorder){  1 usage
    HashMap<Integer,Integer> inMp = new HashMap<>();
    for(int i=0;i<inorder.length;i++)
        inMp.put(inorder[i],i);

    return construction(inorder, inStart: 0, inEnd: inorder.length-1,inMp,preorder, preStart: 0, preEnd: preorder.length-1);
}
```

Time Complexity: O(2n)

- **Building the map** -> Runs once for all elements → O(n).
- **Recursive construction**
    - Each recursive call creates one node.
    - Each node is created exactly once (since preorder is traversed sequentially).
    - For every node, we do:
    - Constant time lookup in the map → O (1).
    - Constant amount of arithmetic + recursion setup.
    - So, total across all nodes → O(n).

Space Complexity: O(n) + O(h)

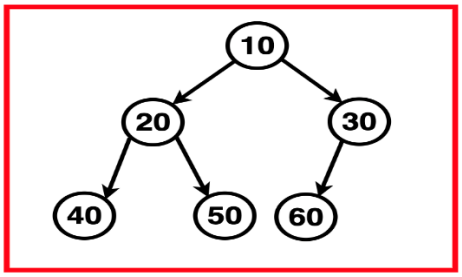
3. **HashMap** → stores indices of all nodes → O(n).
4. **Recursive call stack**: O(h) where h is the height of tree.

Given two arrays representing the inorder and postorder traversals of a binary tree, your task is to construct the binary tree and return its root.

Note: The inorder and postorder traversals contain unique values.

Input: Inorder: [40, 20 , 50, 10, 60, 30], Postorder: [40, 50, 20, 60, 30, 10]

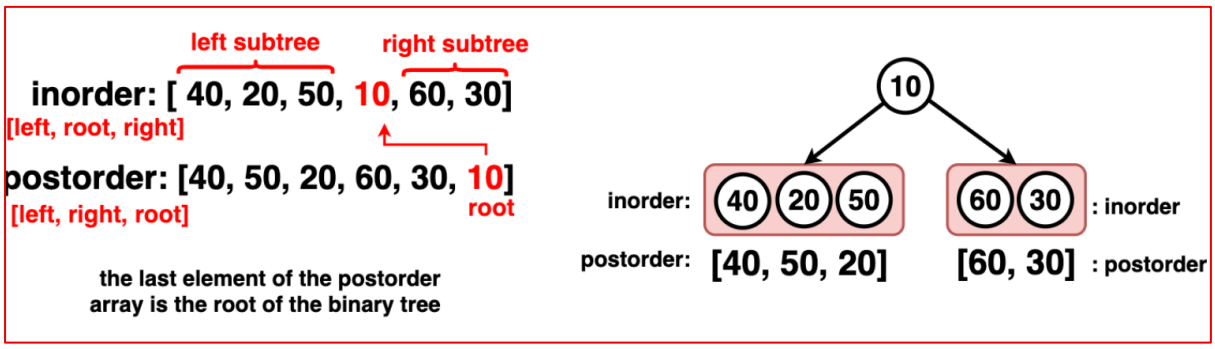
Output:



Explanation: The unique Binary Tree hence created has inorder traversal: [40, 20, 50, 10, 60, 30] and postorder traversal: [40, 50, 20, 60, 30, 10].

Optimal:

We identify a node from the postorder array, locate it in the inorder traversal which splits the array into the left and right subtree.



To avoid unnecessary array duplication, we use variables (inStart, inEnd) and (postStart, postEnd) on the inorder and postorder arrays, respectively. These variables effectively define the boundaries of the current subtree within the original inorder and postorder traversals.

Similar to the previous algorithm, we employ a hashmap to store the index of each element in the inorder traversal, transforming the search operation into a constant-time lookup.

Algorithm:

Step 1: Create an empty map to store the indices of elements in the inorder traversal. Iterate through each element in the inorder traversal and store its index in the map (inMap) using the element as the key and its index as the value.

Step 2: Create a recursive helper function buildTree with the following parameters:

- Postorder array.
- Start index of postorder (postStart), initially set to 0
- End index of postorder (postEnd), initially set to postorder.size() - 1
- Inorder array.

- Start index of inorder (inStart), initially set to 0
- End index of inorder (inEnd), initially set to inorder.size() - 1
- Map for efficient root index lookup in the inorder traversal.

Step 3: Base Case: Check if postStart is greater than postEnd or inStart is greater than inEnd. If true, return NULL, indicating an empty subtree/node.

Step 4: The root node for the current subtree is the last element in the postorder traversal (postorder[postEnd]).

Step 5: Find the index of this root node in the inorder traversal using the map (inMap[rootValue]). This is the rootIndex.

Step 6: Hence, the left subtree ranges from inStart to rootIndex. Subtracting these indexes gives us the leftSubtreeSize.

Step 7: Make two recursive calls to buildTree to build the left and right subtrees:

For the left subtree:

- Update postStart to postEnd - leftSubtreeSize (moving to the next element in postorder)
- Update postEnd to postEnd - 1 (end of left subtree in postorder)
- Update inEnd to rootIndex - 1 (end of left subtree in inorder)

For the right subtree:

- Update postStart to postStart (moving to the next element in postorder)
- Update postEnd to postEnd - leftSubtreeSize - 1 (end of right subtree in postorder)
- Update inStart to rootIndex + 1 (start of right subtree in inorder)

Step 8: Return the root node constructed for the current subtree.

Code:

```java
static Node construction(int[] inorder, int inStart, int inEnd, Map<Integer,Integer> inMap,int[] postorder, int postStart,int postEnd)
{
    if (inStart>inEnd || postStart>postEnd) return null;

    Node root = new Node(postorder[postEnd]);
    int inRoot = inMap.get(root.data);
    int numsLeft = inRoot - inStart;

    root.left = construction(inorder,inStart, inEnd: inRoot-1,inMap,postorder,postStart, postEnd: postStart+numsLeft-1);
    root.right = construction(inorder, inStart: inRoot+1,inEnd,inMap,postorder, postStart: postStart+numsLeft, postEnd: postEnd-1);
    return root;
}

static Node optimal(int[] inorder, int[] postorder){  1 usage
    HashMap<Integer,Integer> inMp = new HashMap<>();
    for(int i=0;i<inorder.length;i++)
        inMp.put(inorder[i],i);

    return construction(inorder, inStart: 0, inEnd: inorder.length-1,inMp,postorder, postStart: 0, postEnd: postorder.length-1);
}
```

Time Complexity: O(2n)

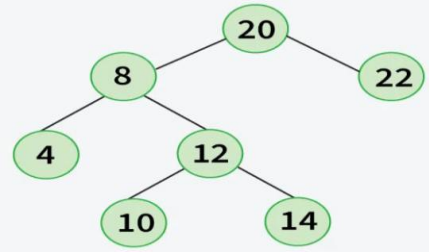
- **Building the map** -> Runs once for all elements → O(n).
- **Recursive construction**
    - Each recursive call creates one node.
    - Each node is created exactly once (since preorder is traversed sequentially).
    - For every node, we do:
    - Constant time lookup in the map → O (1).
    - Constant amount of arithmetic + recursion setup.
    - So, total across all nodes → O(n).

Space Complexity: O(n) + O(h)

5. **HashMap** → stores indices of all nodes → O(n).
6. **Recursive call stack**: O(h) where h is the height of tree.

Given inorder and level-order traversals of a Binary Tree, construct the Binary Tree and return the root Node.

Input: in[] = {4, 8, 10, 12, 14, 20, 22}; level[] = {20, 8, 22, 4, 12, 10, 14};
Output:



Optimal:

Algorithm

Step 1: Build a map for inorder indexes

1. Create a HashMap<Integer, Integer> inMap.
2. For each element in inorder[], store (value → index) in the map.

Step 2: Initialize root node and queue

1. Take the first element of levelorder[] as root.
2. Create a queue q.
3. Enqueue a Data object containing:
    - root node
    - start index = 0
    - end index = inorder.length - 1

Step 3: BFS-like construction
While the queue is not empty:

1. Dequeue one Data object (curr).
2. Let node = curr.root, inStart = curr.start, inEnd = curr.end.
3. Find the index of node.data in inorder using inMap. Call it currIndex.

Step 4: Create left child if exists

1. If inStart < currIndex:
    - Take the next element from levelorder.
    - Create a new node as left child.
    - Enqueue a new Data object for the left child with range [inStart … currIndex-1].

Step 5: Create right child if exists

1. If currIndex < inEnd:
    - Take the next element from levelorder.
    - Create a new node as right child.
    - Enqueue a new Data object for the right child with range [currIndex+1 … inEnd].

Step 7: Return root

Finally, return the root node of the constructed tree.

Code:

```
class Data{
    Node root;
    int start;
    int end;
    Data(Node root,int start, int end){
        this.root = root;
        this.start = start;
        this.end = end;
    }
}
class GfG {
    Node buildTree(int inorder[], int levelorder[]) {
        HashMap<Integer,Integer> inMp = new HashMap<>();
        for (int i=0;i<inorder.length;i++) inMp.put(inorder[i],i);
        int levelIndex = 0;
        Node root = new Node(levelorder[levelIndex++]);
        Queue<Data> q = new LinkedList<>();
        q.add(new Data(root,0,inorder.length-1));
        while (!q.isEmpty()){
            Data curr = q.remove();
            Node node = curr.root;
            int inStart = curr.start, inEnd = curr.end;
            int currIndex = inMp.get(node.data);
            if (inStart < currIndex){
                Node temp = new Node(levelorder[levelIndex++]);
                node.left = temp;
                q.add(new Data(temp,inStart,currIndex-1));
            }
            if (currIndex < inEnd){
                Node temp = new Node(levelorder[levelIndex++]);
                node.right = temp;
                q.add(new Data(temp,currIndex+1,inEnd));
            }
        }
        return root;
    }
}
```

Time: O(2n)

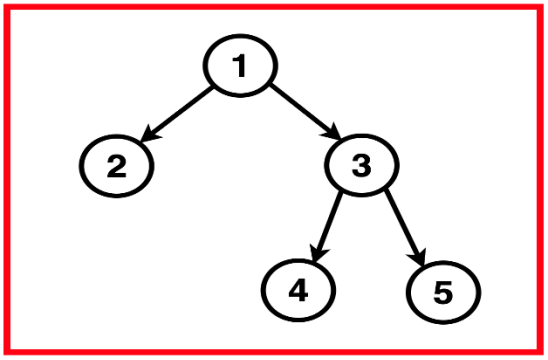
- Building inMap → O(n)
- Processing each node once → O(n)

Space: O(2n)

- HashMap for inorder indexes → O(n)
- Queue for level-order construction → O(n)

Given a Binary Tree, design an algorithm to serialise and deserialise it. There is no restriction on how the serialisation and deserialization takes place. But it needs to be ensured that the serialised binary tree can be deserialized to the original tree structure. Serialisation is the process of translating a data structure or object state into a format that can be stored or transmitted (for example, across a computer network) and reconstructed later. The opposite operation, that is, extracting a data structure from stored information, is deserialization.

Input: Binary Tree: 1 2 3 -1 -1 4 5

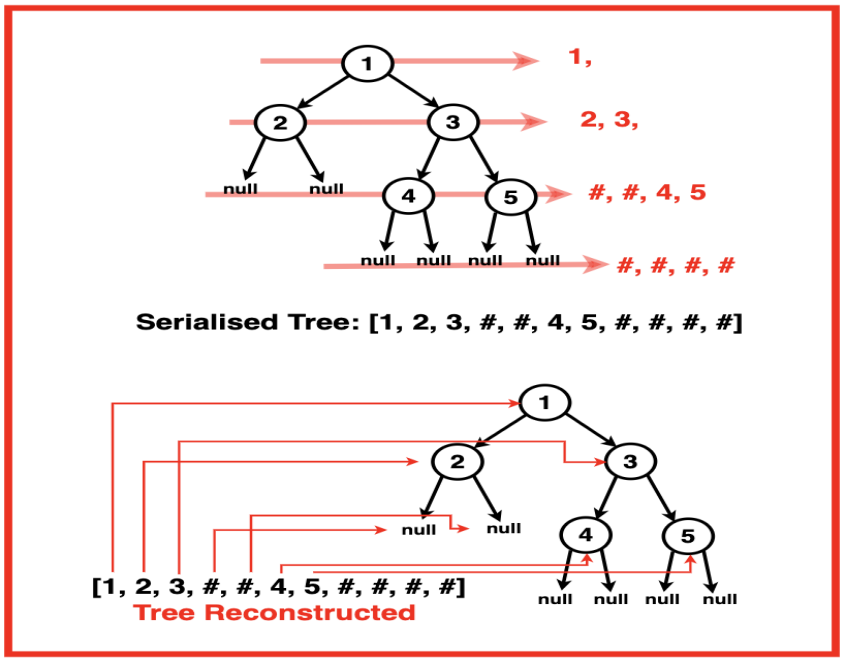


Output: After Serialisation: 1 2 3 # # 4 5 # # # #

After Deserialization: (Original Tree Back)

Explanation: Any algorithm that compresses this binary tree to a string which can be transmitted and from which the binary tree can be reconstructed later can be used.

Here we have used a serialisation algorithm based on level order traversal where spaces separates the nodes and # denotes null nodes.



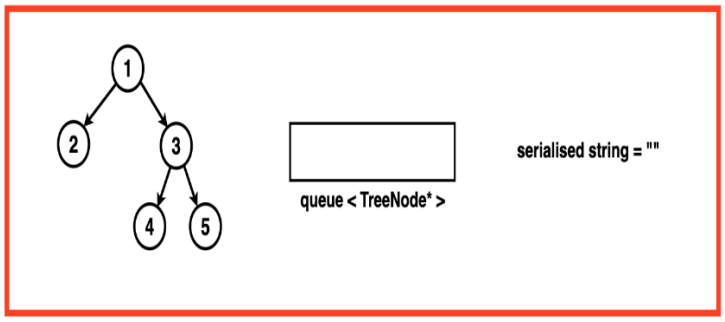
Optimal:

Serialisation:

Step 1: Check if the tree is empty: return an empty string.

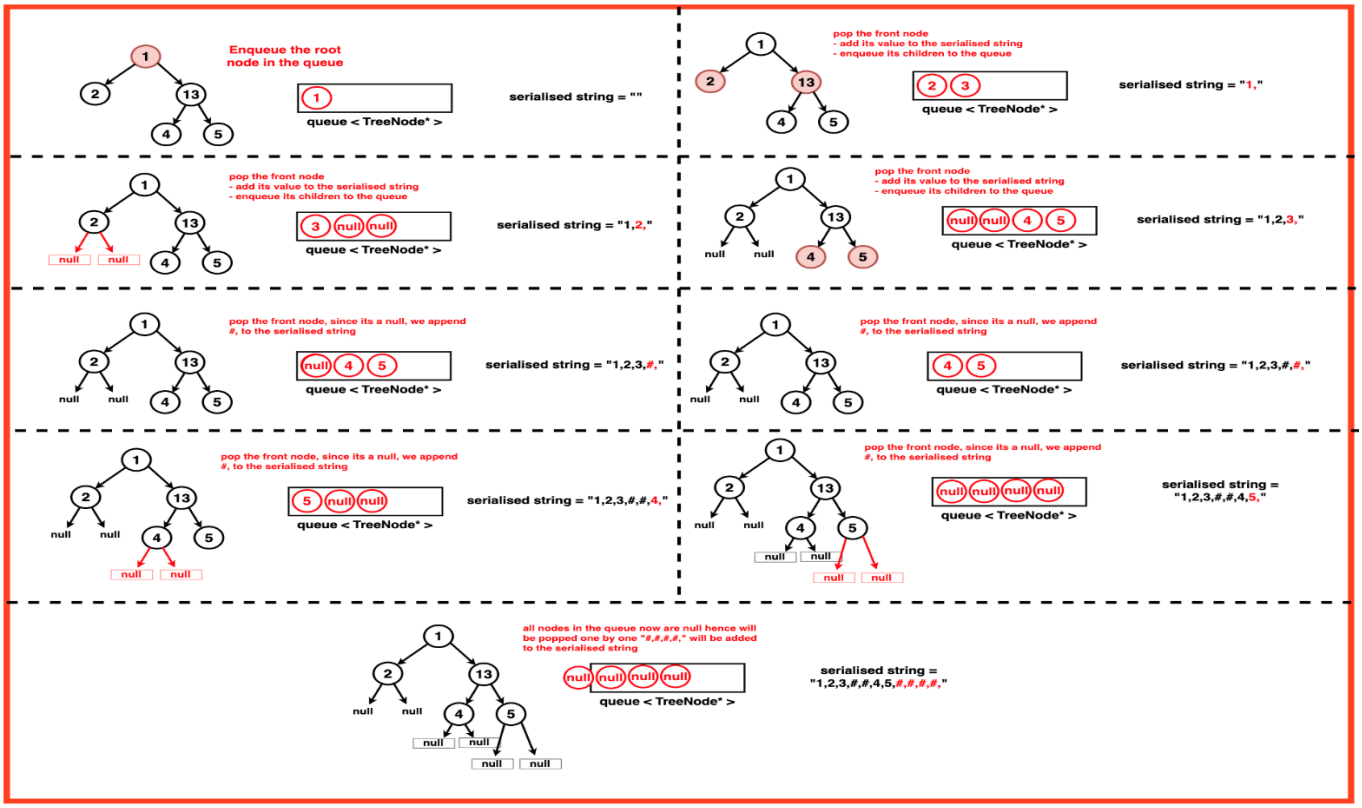
Step 2: Initialise an empty string: This string will store the serialised binary tree.

Step 3: Use a queue for level-order traversal: Initialise a queue and enqueue the root.



Step 4: Within the level-order traversal loop:

- Dequeue a node from the queue.
- If the node is null, append "#" to the string.
- If the node is not null, append its data value along with a ‘,’ (comma) to the string. This comma acts as a delimiter that separates the different node values in the string. Enqueue its left and right children.



Step 5: Return the final string containing the serialised representation of the tree.

Code:

```
static String serialize(Node root){  1 usage
    if (root == null) return "";

    StringBuilder res = new StringBuilder();
    Queue<Node> q = new LinkedList<>();
    q.add(root);
    while (!q.isEmpty()){
        Node temp = q.remove();
        if (temp==null){
            res.append("#,");
        }
        else {
            res.append(temp.data + ",");
            q.add(temp.left);
            q.add(temp.right);
        }
    }
    return res.toString();
}
```

Time Complexity: O(n)

- Each node is enqueued and dequeued once → **O(n)**, where n = number of nodes.
- Appending to StringBuilder is amortized O(1).

Space Complexity: O(2n)

- Queue can hold at most O(n) nodes.
- Output string stores values of n nodes + null markers (#), so O(n).

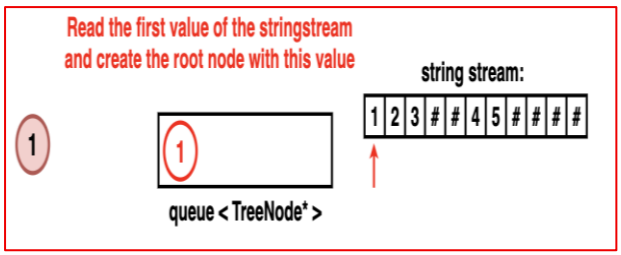
Deserialization:

Step 1: If the serialised data is empty: return null.

Step 2: Split data by "," into an array values[].

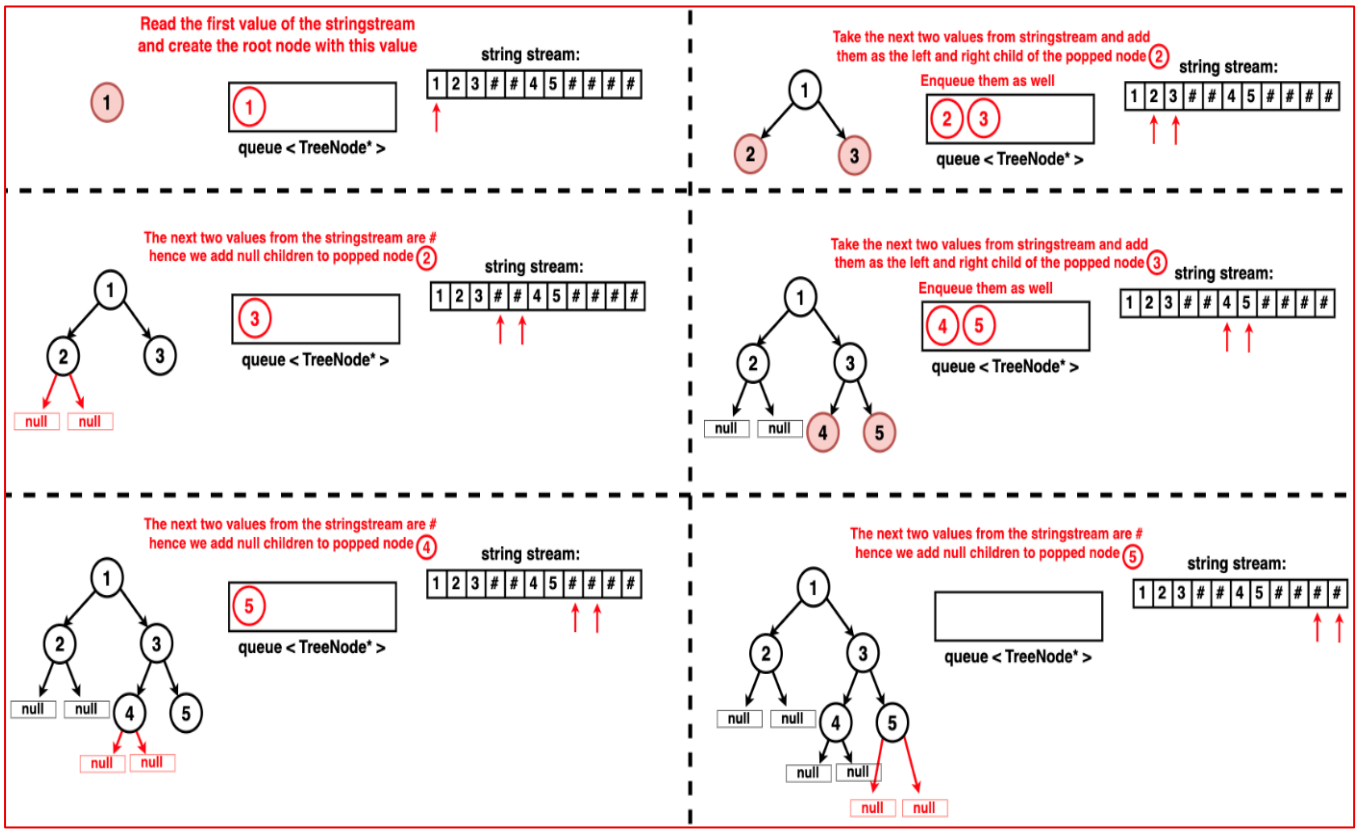
Step 3: Convert values[i] to integer and create the root node.

Step 4: Use a queue for level-order traversal: Initialise a queue and enqueue the root.



Step 5: While the queue is not empty:

- Dequeue a node from the queue.
- Read the value for the left child from the values.
- If it is "#", set the left child to null. If it's not "#", create a new node with the value and set it as the left child.
- Read the next value in the values for the right child.
- If it is "#", set the right child to null. If it's not "#", create a new node with the value and set it as the right child.
- Enqueue the left and right children into the queue for further traversal.



Step 6: Return the reconstructed root: The final result is the root of the reconstructed tree.

Code:

```java
static Node deserialize(String data){  1 usage
    if (data == "") return null;

    String[] values = data.split( regex: ",");
    int i = 0;

    Queue<Node> q = new LinkedList<>();
    Node root = new Node(Integer.parseInt(values[i++]));
    q.add(root);
    while (!q.isEmpty()){
        Node temp = q.remove();
        if (!values[i].equals("#")){
            Node l = new Node(Integer.parseInt(values[i]));
            temp.left = l;
            q.add(l);
        }
        if (!values[++i].equals("#")){
            Node r = new Node(Integer.parseInt(values[i]));
            temp.right = r;
            q.add(r);
        }
        ++i;
    }
    return root;
}
```

Time Complexity: O(2n)

- Each value in values[] is processed exactly once → **O(n)**.
- Splitting the string → O(n).

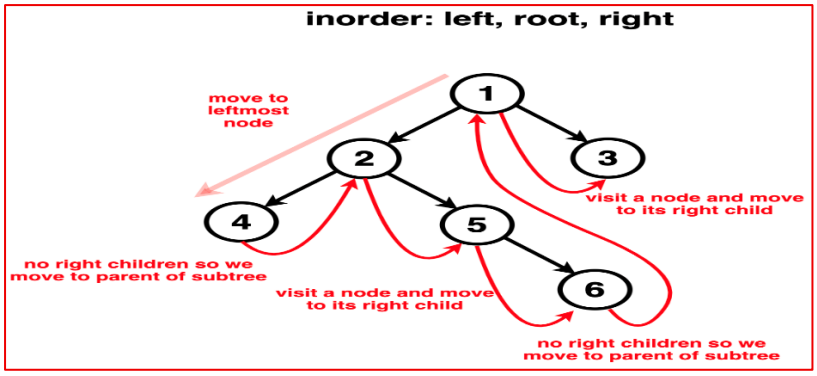
Space Complexity: O(3n)

- Queue holds at most O(n) nodes.
- values[] array size is O(n).
- Tree reconstruction also takes O(n).

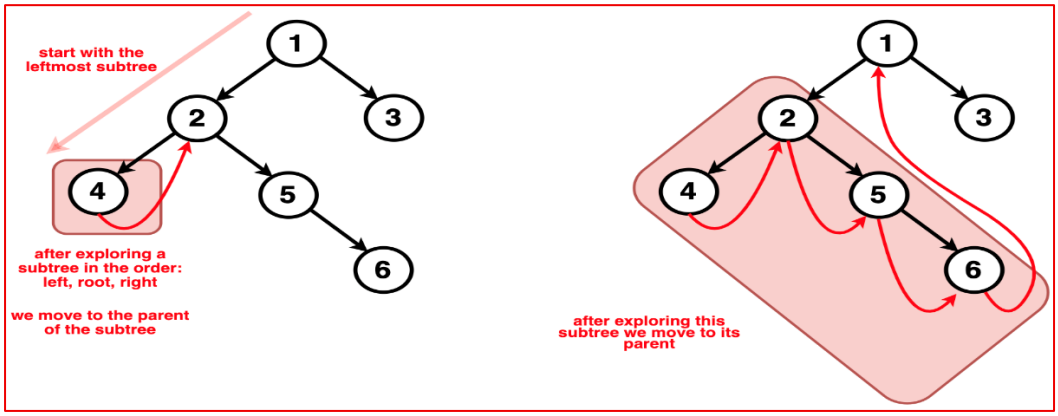
Given a Binary Tree, implement Morris Inorder Traversal and return the arrayList containing its inorder sequence.

Morris Traversal is a tree traversal algorithm that allows for an in-order traversal of a binary tree without using recursion or a stack. It uses threading to traverse the tree efficiently. The key idea is to establish a temporary link between the current node and its in-order successor

Optimal:



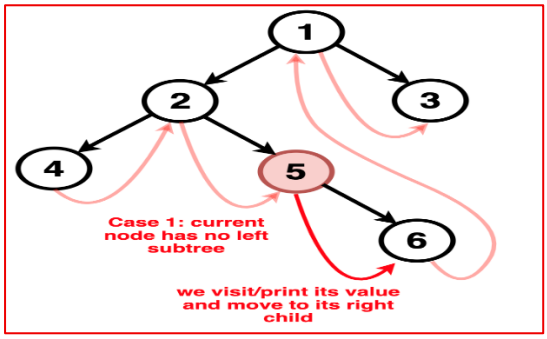
The inorder predecessor of a node is the rightmost node in the left subtree. So when we traverse the left subtree, we encounter a node whose right child is null, this is the last node in that subtree.Hence, we observe a pattern whenever we are at the last node of a subtree such that the right child is pointing to none, we move to the parent of this subtree.



When we are currently at a node, the following cases can arise:

Case 1: The current node has no left subtree.

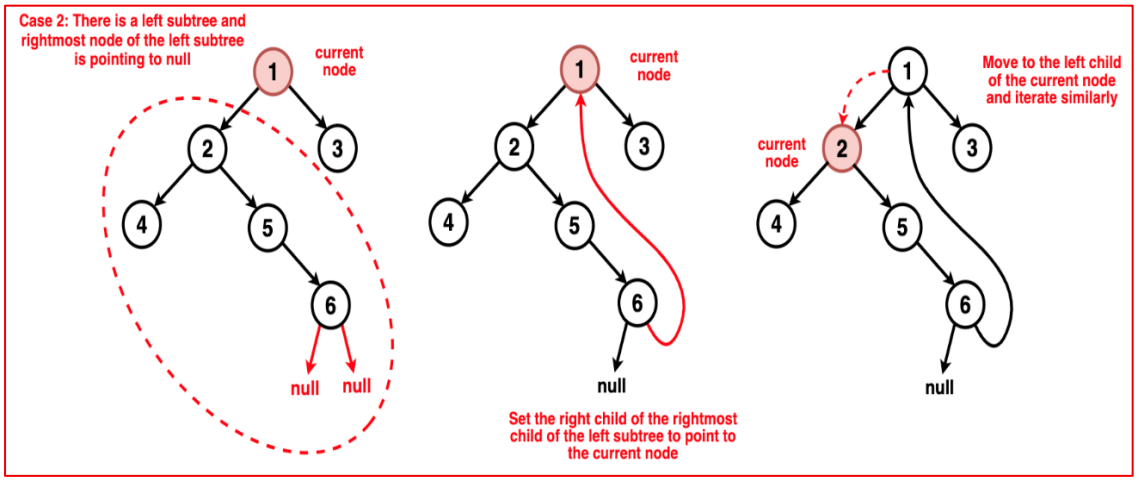
- Print the value of the current node.
- Then, go to the right child of the current node.



Case 2: There is a left subtree, and the right-most child of this left subtree is pointing to null.

- Set the right-most child of the left subtree to point to the current node.

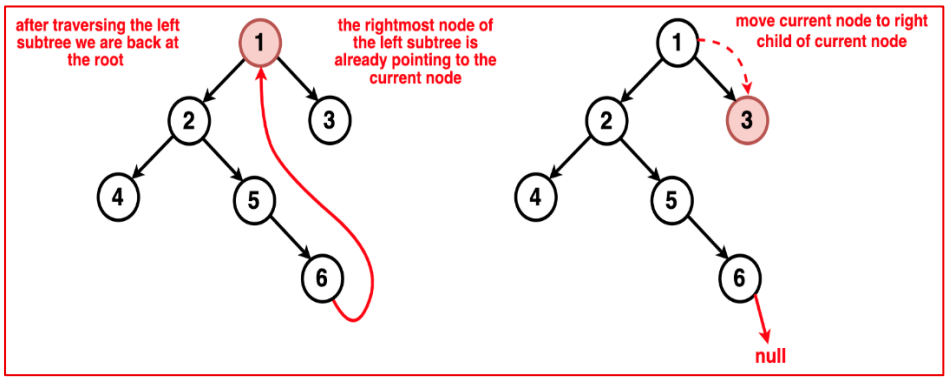
- Move to the left child of the current node.



In this case, we haven't visited the left subtree yet. We establish a temporary link from the rightmost node of the left subtree to the current node. This link helps us later to identify when we've completed the in-order traversal of the left subtree. After setting the link, we move to the left child to explore the left subtree.

Case 3: There is a left subtree, and the right-most child of this left subtree is already pointing to the current node.

- Print the value of the current node.
- Revert the temporary link (set it back to null).
- Move to the right child of the current node.



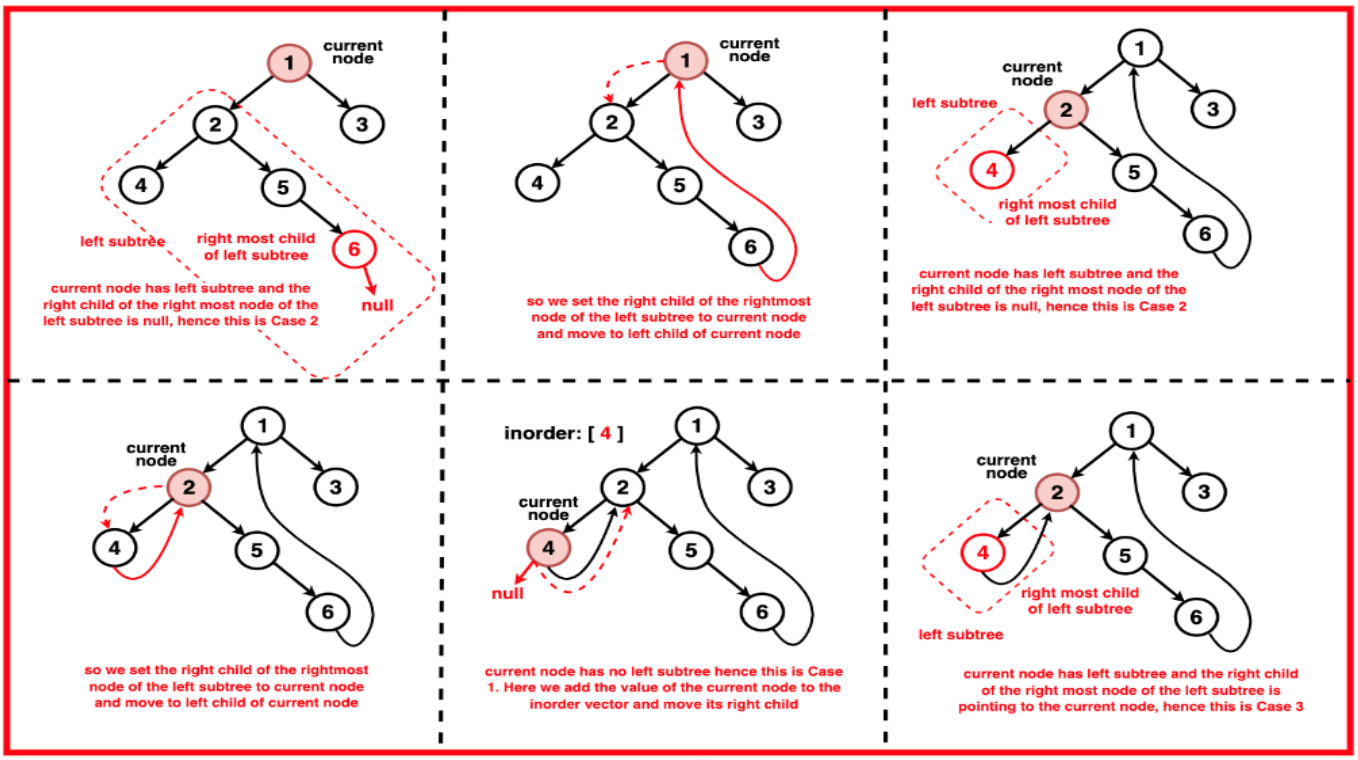
This case is crucial for maintaining the integrity of the tree structure. If the right-most child of the left subtree is already pointing to the current node, it means we've completed the in-order traversal of the left subtree. We print the value of the current node and then revert the temporary link to restore the original tree structure. Finally, we move to the right child to continue the traversal.

Note: The temporary links added in Case 2 are essential for identifying the completion of the left subtree in Case 3. It's critical to revert these links to avoid altering the original structure of the tree.

Algorithm:

Step 1: Initialise a 'current' to traverse the tree. Set 'current' to the root of the Binary Tree.

Step 2: While the 'current' is not null: If the current node has no left child, print the current node's value and move to the right child ie. set the current to its right child.



Step 3: If the current node has a left child, we find the in-order predecessor of the current node. This in-order predecessor is the rightmost node in the left subtree or the left subtree's rightmost node.

If the right child of the in-order predecessor is null:

- Set the right child to the current node.
- Move to the left child (i.e., set current to its left child).

If the right child of the in-order predecessor is not null:

- Revert the changes made in the previous step by setting the right child as null.
- Print the current node's value.
- Move to the right child (i.e., set current to its right child).

Repeat steps 2 and 3 until the end of the tree is reached.

Code:

```
static ArrayList<Integer> inOrder(Node root){  1 usage
    ArrayList<Integer> res = new ArrayList<>();
    Node current = root;

    while (current != null){
        if (current.left == null){
            res.add(current.data);
            current = current.right;
        }
        else {
            Node temp = current.left;
            while (temp.right != null && temp.right != current)
                temp = temp.right;

            if (temp.right == null){
                temp.right = current;
                current = current.left;
            }
            else{
                temp.right = null;
                res.add(current.data);
                current = current.right;
            }
        }
    }
    return res;
}
```

Time Complexity: O(2N) where N is the number of nodes in the Binary Tree.

- The time complexity is linear, as each node is visited at most twice (once for establishing the temporary link and once for reverting it).

Space Complexity: O (1)

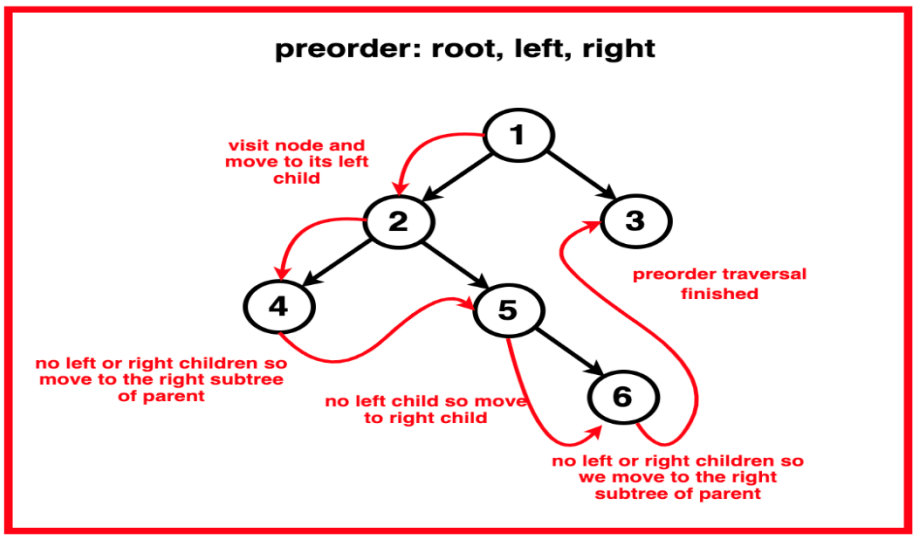
Given a Binary Tree, implement Morris Preorder Traversal and return the array containing its preorder sequence.

Optimal:

We extend Morris Inorder Traversal to Preorder Morris Traversal and modify the algorithm to print the current node's value before moving to the left child when the right child of the inorder predecessor is null.

This change ensures that the nodes are processed in the desired order for Preorder Traversal. The basic structure of Morris Traversal remains constant, but the printing step is

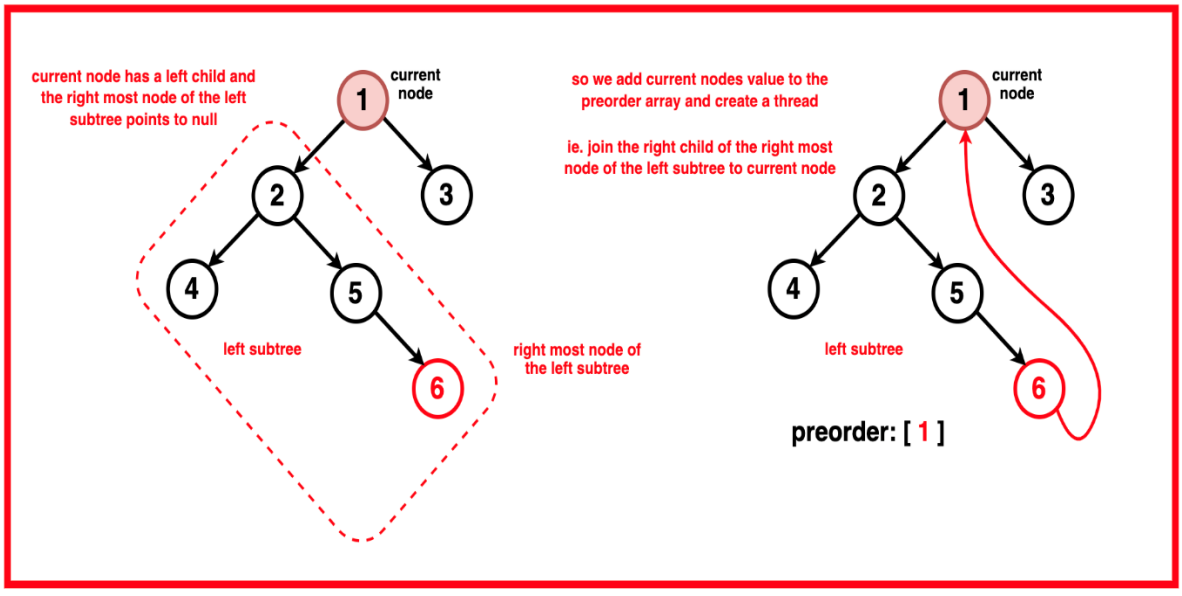
adjusted, resulting in a Preorder Traversal that is still in-place and has a constant space complexity.



In Morris Inorder Traversal, we are traversing the tree in the way: Left, Root, Right. In Morris Preorder traversal we want to traverse the tree in the way: Root, Left, Right. Therefore, the following code changes are required:

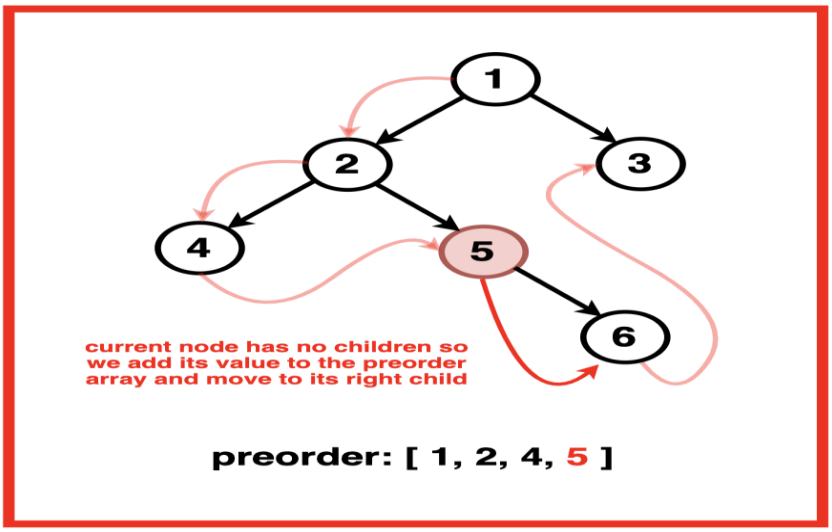
When the current node has a left child:

In Morris Inorder Traversal, a new thread is created by establishing a temporary link between the current node and its in-order predecessor. In Morris Preorder Traversal, we want to print the root before visiting the left child. Therefore, after setting the thread (establishing the link), we print the current node's value before moving it to its left child.



When the current node has no left child:

This case remains unchanged from Morris Inorder Traversal. If the current node has no left child, there is nothing to visit on the left side. In both Inorder and Preorder traversals, we want to print the current node's value and move to the right child. Therefore, there is no code modification needed for this scenario.



Algorithm:

Step 1: Initialise a current to traverse the tree. Set current to the root of the Binary Tree.

Step 2: While the current is not null: If the current node has no left child, print the current node's value and move to the right child ie. set current to its right child.

Step 3: If the current node has a left child, we find the in-order predecessor of the current node. This in-order predecessor is the rightmost node in the left subtree or the left subtree's rightmost node.

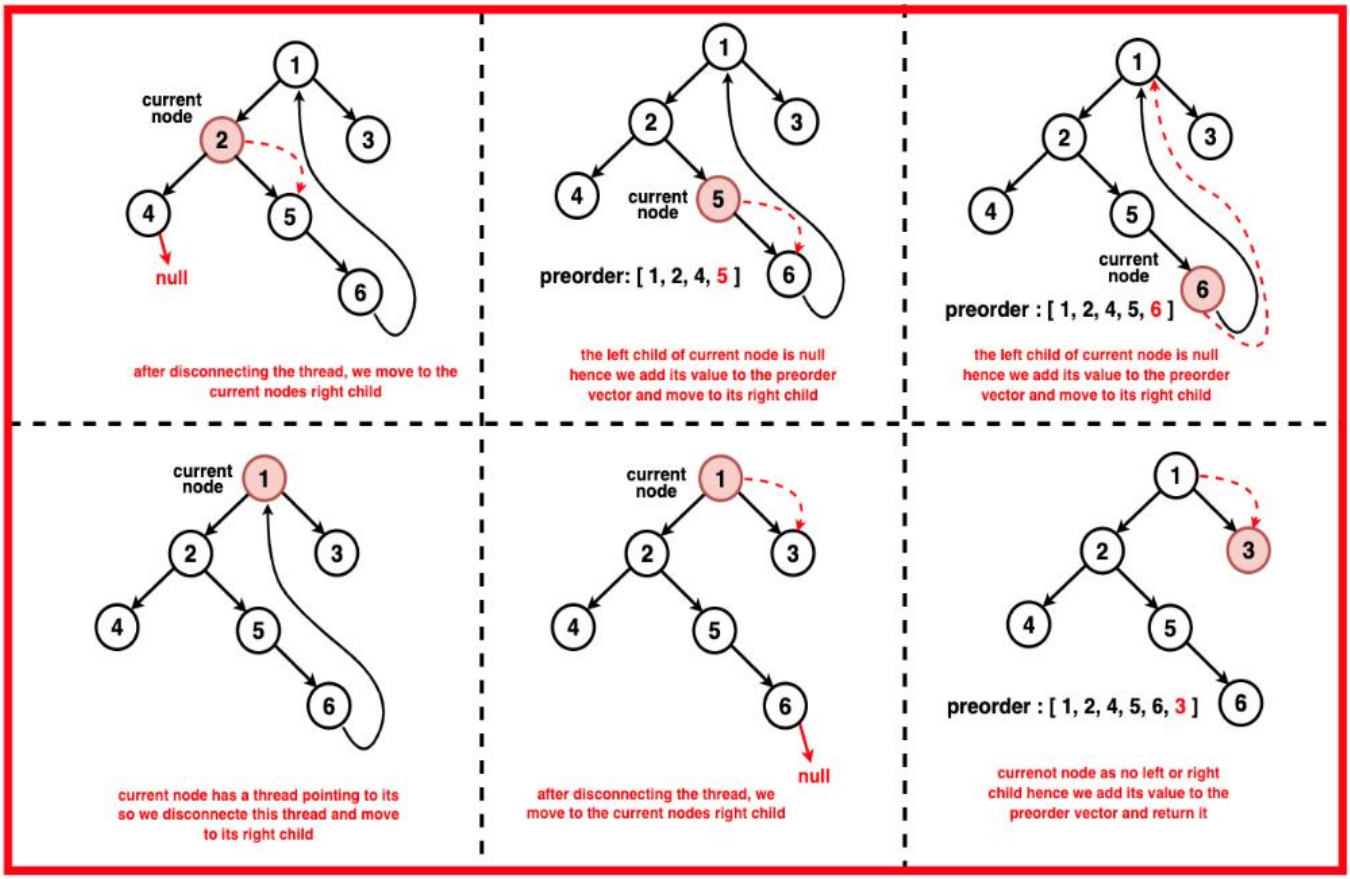
If the right child of the in-order predecessor is null:

- Set the right child to the current node.
- Print the value of the current node as preorder traverses the tree in the inorder: Root, Left then Right.
- Move to the left child (i.e., set current to its left child).

If the right child of the in-order predecessor is not null:

- Revert the changes made in the previous step by setting the right child as null.
- Move to the right child (i.e., set current to its right child).

Repeat steps 2 and 3 until the end of the tree is reached.



Code:

```
static ArrayList<Integer> preOrder(Node root){  1 usage
    ArrayList<Integer> res = new ArrayList<>();
    Node current = root;

    while (current != null){
        if (current.left == null){
            res.add(current.data);
            current = current.right;
        }
        else {
            Node temp = current.left;
            while (temp.right != null && temp.right != current)
                temp = temp.right;

            if (temp.right == null){
                temp.right = current;
                res.add(current.data);
                current = current.left;
            }
            else{
                temp.right = null;
                current = current.right;
            }
        }
    }
    return res;
}
```

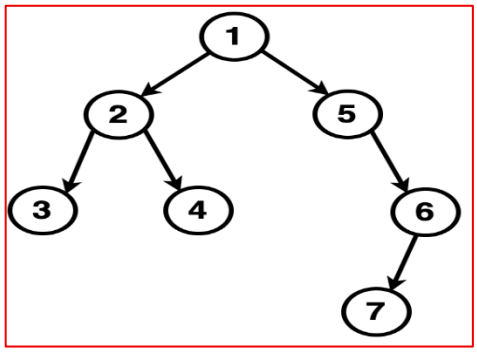
Time Complexity: O(2N) where N is the number of nodes in the Binary Tree.

- The time complexity is linear, as each node is visited at most twice (once for establishing the temporary link and once for reverting it).

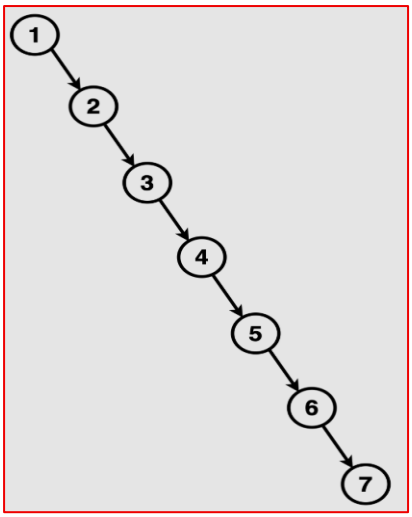
Space Complexity: O (1)

Given a Binary Tree, convert it to a Linked List where the linked list nodes follow the same order as the pre-order traversal of the binary tree. Use the right pointer of the Binary Tree as the 'next' pointer for the linked list and set the left pointer to null. Do this in place and do not create extra nodes.

Input: [1,2, 5, 3, 4, null, 6, null, null, null, null,7, null]



Output: [1, null, 2, null, 3, null, 4, null, 5, null, 6, null, 7]

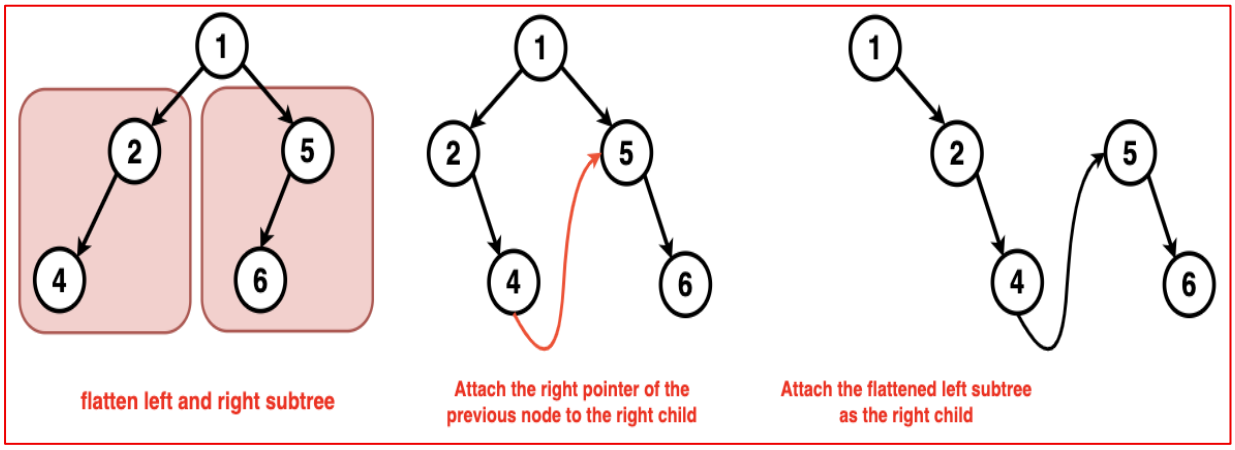


Explanation: The given binary tree has a preorder traversal order of [1, 2, 3, 4, 5, 6, 7]. To flatten the tree, we convert it into a linked list. In this linked list, the nodes follow the same order as the preorder traversal. The right pointer of each node in the linked list serves the function of the 'next' pointer. Additionally, we set the left pointer of each node to null. This process is done in place, meaning we modify the existing binary tree structure to achieve the desired linked list representation.

Brute-force: Using recursion

The intuition behind this approach is to perform a reverse pre-order traversal where, instead of simply visiting nodes, we flatten the tree into a linked list as we traverse it.

We start at the root of the tree and recursively do the following for each node, we first ensure that the right subtree is flattened into a linked list. This means that all nodes in the right subtree are processed and attached to the linked list in the correct order.



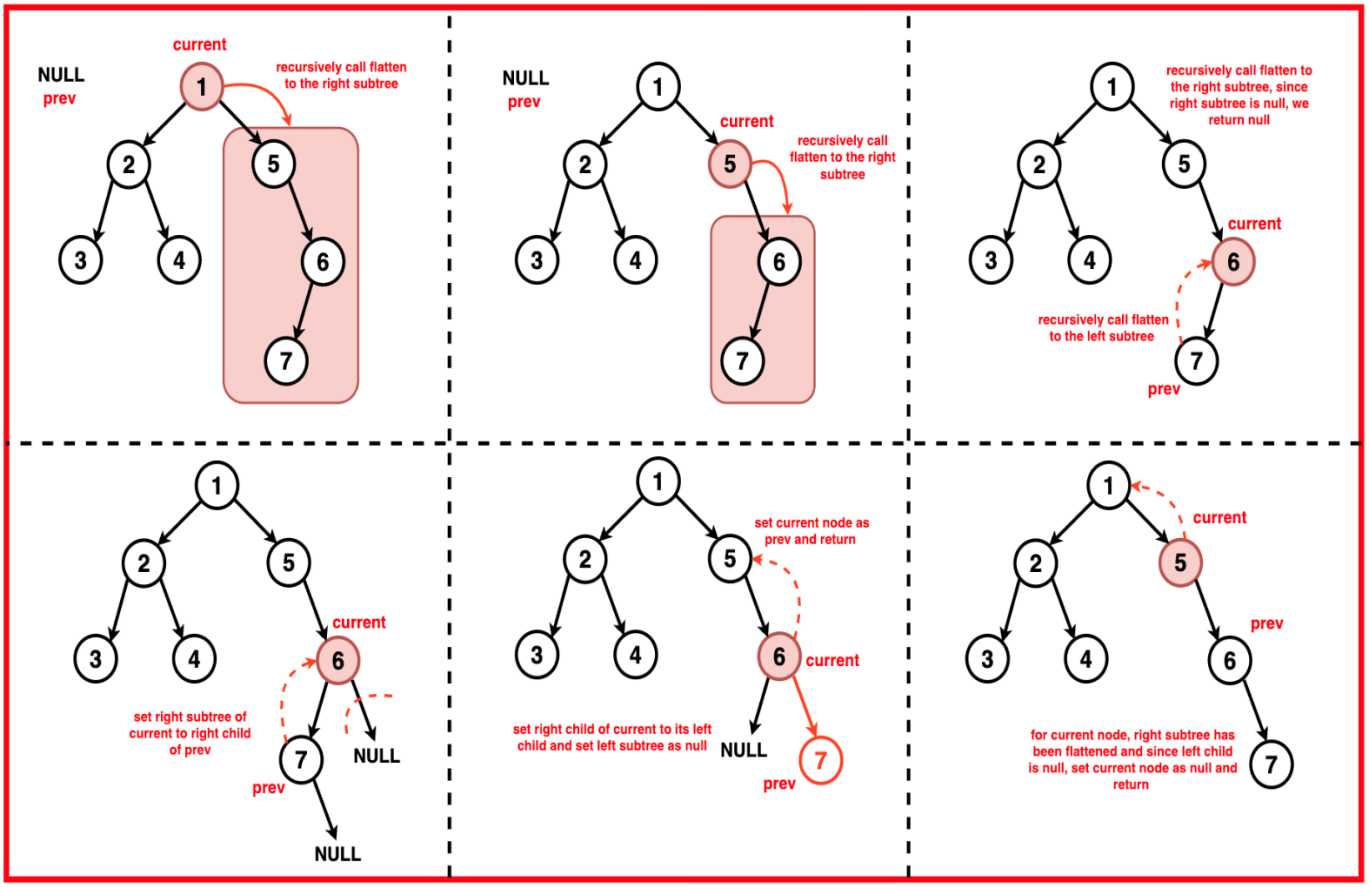
Next, we do the same for the left subtree. This ensures that all nodes in the left subtree are processed and attached to the linked list in the correct order. Once both subtrees are flattened, we attach the flattened left subtree as the right child of the current node. Since we're using the right pointer of the binary tree as the next pointer for the linked list, this effectively attaches the left subtree to the current node in the linked list. Finally, we attach the flattened right subtree to the rightmost node of the flattened left subtree. This ensures that the right subtree is properly attached to the end of the linked list.

Algorithm

Step 1: Initialise a global variable `prev` to keep track of the previously processed node. Initially set it to null.

Step 2: Base Case: If the current node is null, return.

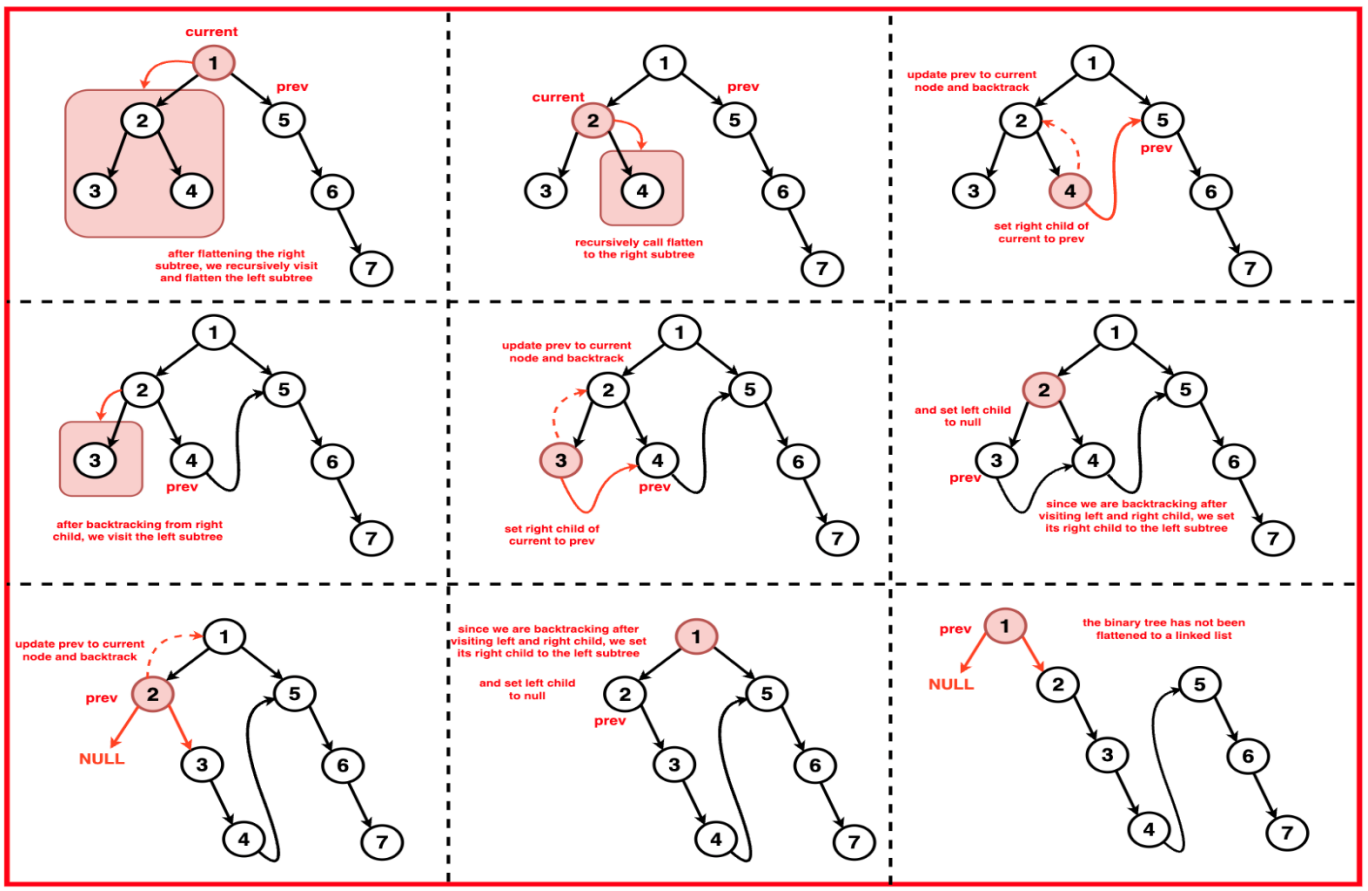
Step 3: Flatten the Right and Left Subtree: Recursively flatten the right and left subtree of the current node by calling the flatten function on the current node's right and left child.



Step 4: Attach the Right Subtree to the Flattened Left Subtree: Set the right child of the current node to the value of `prev` since `prev` points to the rightmost node in the flattened left subtree. This effectively attaches the right subtree to the right of the rightmost node of the left subtree.

Step 5: Attach the Left Subtree as Right Child: Set the right child of the current node to the left subtree.

- Set the left child of the current node to null since we are flattening the binary tree to a linked list and there should be no left child.



Step 6: Update `prev` to the current node for the next iteration and recursion step.

Code:

```
static Node prev = null;  2 usages
static void recursion(Node root){  2 usages
    if (root == null) return;

    recursion(root.right);
    recursion(root.left);
    root.right = prev;
    root.left = null;
    prev = root;
}
```

Time Complexity: O(N) where N is the number of nodes in the Binary Tree. Each node of the binary node is visited exactly once.

Space Complexity: O(H) where H is the height of Binary Tree. (Stack space)

Better Solution: Iterative

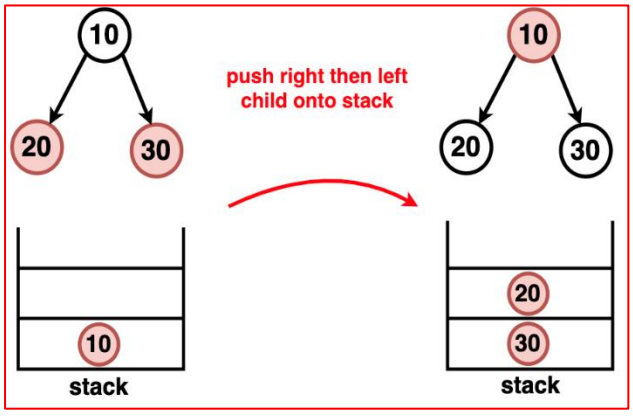
Algorithm

Step 1: Base Case: If the root node is null, we return as there is no tree to flatten.

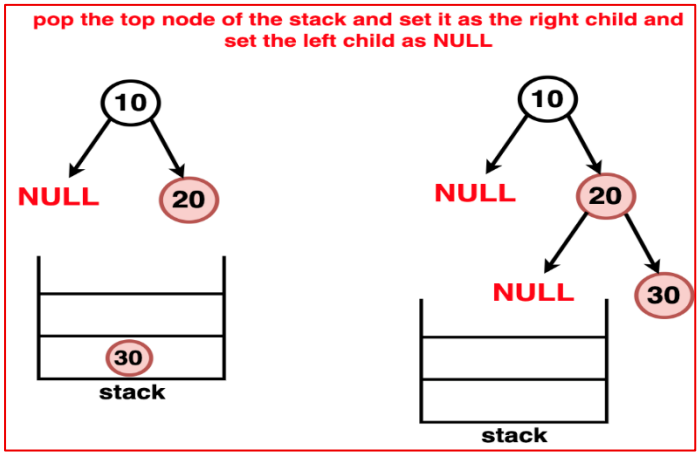
Step 2: Initialise Stack: Create a stack & push the root node onto the stack.

Step 3: Iterative Traversal with Stack: While the stack is not empty, repeat the following steps:

- Pop the top node from the stack.
- If the popped node has a right child, push it onto the stack. This ensures that the right child is processed after the left child nodes.
- If the popped node has a left child, push it onto the stack.



- If the stack is not empty after pushing the left child, connect the right pointer of the current node (popped from the stack) to the top node of the stack. This creates the linked list structure by setting the next pointer.
- Set the left pointer of the current node to null.



Step 4: Once the stack becomes empty, the traversal is complete and the binary tree is flattened into linked list structure.

Code:

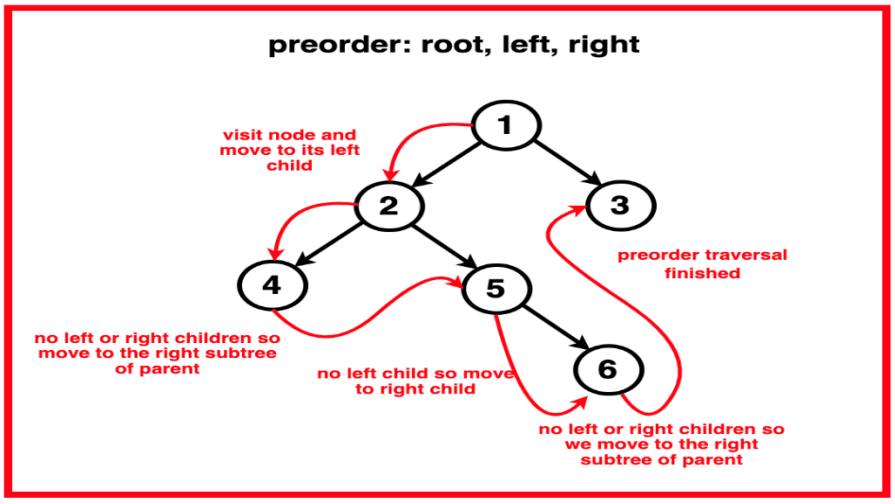
```
static void iterative(Node root){  no usages
    if(root == null) return;

    Stack<Node> st = new Stack<>();
    st.add(root);
    while (!st.isEmpty()){
        Node curr = st.pop();
        if (curr.right != null) st.add(curr.right);
        if (curr.left != null) st.add(curr.left);
        if (!st.isEmpty()) curr.right = st.peek();
        curr.left = null;
    }
}
```

Time Complexity: O(N) where N is the number of nodes in the Binary Tree. Each node of the binary node is visited exactly once.

Space Complexity: O(N) where H is the height of Binary Tree as we are using Stack as an additional data structure.

Optimal: Using threaded binary tree

Morris Traversal is an algorithm that allows preorder/inorder tree traversal without using any extra space for stack or recursion. It utilises threaded binary trees to traverse the tree without requiring a stack for saving the path.



Algorithm

Step 1: If the root of the tree is empty, return.

Step 2: Start with the current node as the root of the tree.

Step 3: While the current node is not null, we traverse the tree:

- If the current node has a left child:
    - Find the rightmost node in the left subtree.
    - Connect the rightmost node of the left subtree to the current node's right child.
    - Update the current node's right child to be its left child.
    - Set the current node's left child to null.
- Move to the current node's right child.

Step 4: Repeat until all nodes are processed.

Code:

```
static void usingThreadedBinaryTree(Node root){
    if (root == null) return;

    Node curr = root;
    while (curr != null){
        if (curr.left != null){
            Node prev = curr.left;
            while (prev.right != null)
                prev = prev.right;

            prev.right = curr.right;
            curr.right = curr.left;
        }
        curr.left = null;
        curr = curr.right;
    }
}
```

Time Complexity: O(n)

Each node is visited a constant number of times (finding the rightmost node can amortize to O(n) total).

Space Complexity: O(1)

Given a Binary Tree, find maximum and minimum elements in it.

Optimal: For minimum

Algorithm

Step 1: Start at the root of the binary tree.

Step 2: If the current node is null (tree is empty or reached a child of a leaf), return a very large number.

Step 3: If the current node is a leaf node (both left and right children are null), return the value of this node.

Step 4: Recursively find the minimum value in the left & right subtree.

Step 5: Return the smallest value among the current node, the left subtree minimum, and the right subtree minimum.

```
public static int findMin(Node root) {
    if(root == null) return Integer.MAX_VALUE;
    else if(root.left == null && root.right == null) return root.data;

    int l = findMin(root.left);
    int r = findMin(root.right);

    return Math.min(root.data, Math.min(l, r));
}
```

Time: O(n) — every node is visited once.

Space: O(h) — recursive call stack, where h is the height of the tree.

Optimal: For maximum

Algorithm

Step 1: Start at the root of the binary tree.

Step 2: If the current node is null (tree is empty or reached a child of a leaf), return a very small number.

Step 3: If the current node is a leaf node (both left and right children are null), return the value of this node.

Step 4: Recursively find the minimum value in the left & right subtree.

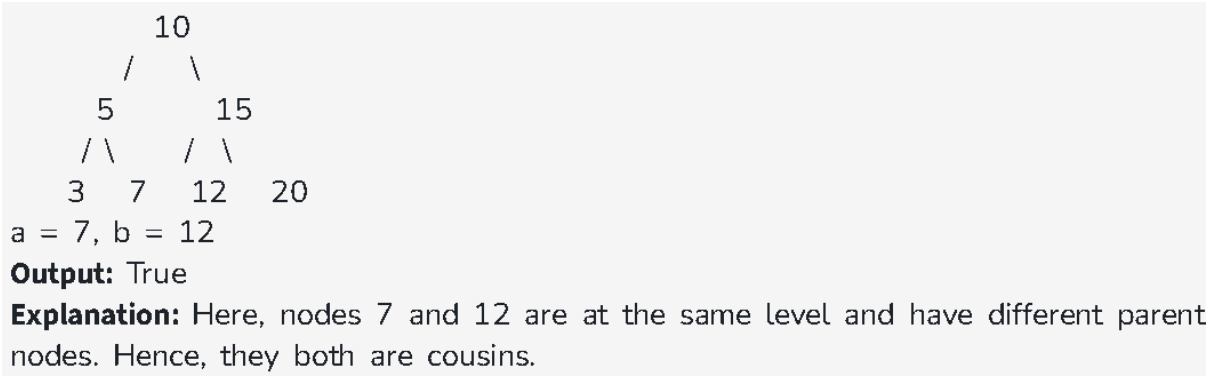
Step 5: Return the largest value among the current node, the left subtree minimum, and the right subtree minimum.

```java
public static int findMax(Node root) {
    if(root == null) return Integer.MIN_VALUE;
    else if(root.left == null && root.right == null) return root.data;

    int l = findMax(root.left);
    int r = findMax(root.right);

    return Math.max(root.data, Math.max(l, r));
}
```

Time: O(n) — every node is visited once.

Space: O(h) — recursive call stack, where h is the height of the tree.

Given a binary tree (having distinct node values) root and two node values. Check whether or not the two nodes with values a and b are cousins.
Note: Two nodes of a binary tree are cousins if they have the same depth with different parents.



a = 7, b = 12
Output: True
Explanation: Here, nodes 7 and 12 are at the same level and have different parent nodes. Hence, they both are cousins.

Optimal Approach:

Algorithm:

Step 1: Initialize data structures

1. Create a map ‘parentMap’ to store the parent of each node.
2. Create a map ‘levelMap’ to store the level (depth) of each node.
3. Initialize two variables t1 and t2 to null. These will point to nodes with values a and b.
4. Set the root’s parent as null and its level as 0.
5. Create a queue for level-order traversal (BFS) and add the root node to it.

Step 2: Level-order traversal (BFS)

1. While the queue is not empty:

1. Increment the current level by 1 (moving to the next tree level).
2. For each node at the current level:
    1. Remove the node from the queue.
    2. If its value equals a, set t1 to this node.
    3. If its value equals b, set t2 to this node.
    4. If the node has a left child:
        - Store the current node as the parent of the left child in 'parentMap'.
        - Store the level of the left child in 'levelMap'.
        - Add the left child to the queue.
    5. If the node has a right child:
        - Store the current node as the parent of the right child in 'parentMap'.
        - Store the level of the right child in 'levelMap'.
        - Add the right child to the queue.

Step 3: Check cousin conditions

1. After BFS, both t1 and t2 will point to the nodes with values a and b.
2. Fetch the **parents** of t1 and t2 from parentMap.
3. Fetch the **levels** of t1 and t2 from levelMap.
4. Nodes a and b are **cousins** if:
    - Their **parents are different** AND
    - Their **levels are equal**

Step 4: Return result

- If the conditions are satisfied, return true.
- Otherwise, return false.

```java
public boolean isCousins(Node root, int a, int b) {
    Map<Node,Node> parentMap = new HashMap<>();
    Map<Node,Integer> levelMap = new HashMap<>();
    Node t1 = null, t2 = null;
    int level = 0;

    parentMap.put(root,null);
    levelMap.put(root,level);
    Queue<Node> q = new LinkedList<>();
    q.add(root);
    while (!q.isEmpty()){
        int size = q.size();
        level += 1;
        for (int i=0; i<size; i++){
            Node temp = q.remove();
            if(temp.data == a) t1 = temp;
            if(temp.data == b) t2 = temp;

            if (temp.left != null){
                parentMap.put(temp.left,temp);
                levelMap.put(temp.left,level);
                q.add(temp.left);
            }
            if (temp.right != null){
                parentMap.put(temp.right,temp);
                levelMap.put(temp.right,level);
                q.add(temp.right);
            }
        }
    }
    if(parentMap.get(t1) != parentMap.get(t2) && levelMap.get(t1) == levelMap.get(t2))
        return true;
    return false;
}
```

Time Complexity: O(3n)

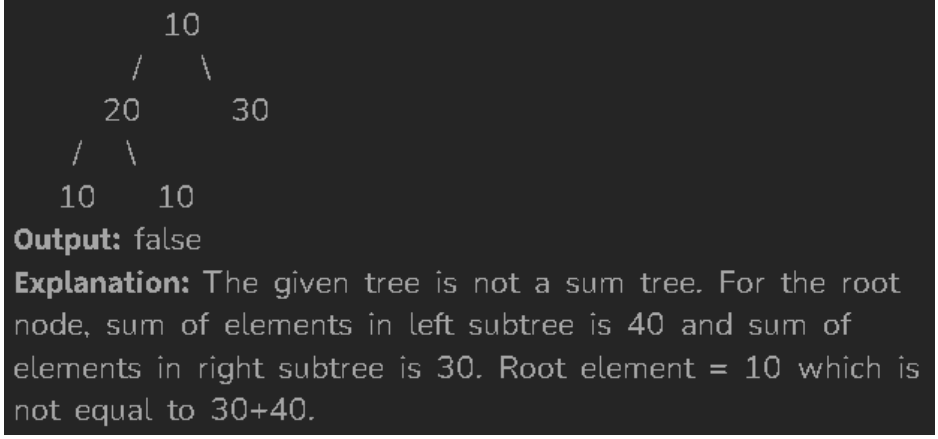
- **BFS traversal:** Each node is enqueued once and dequeued once: O(n)
- **Map operations:** Since each node is inserted once in both maps: O(2n)

Space Complexity: O(3n)

- **Queue q** → holds at most all nodes at a single level: O(n)
- **parentMap** → stores n nodes → O(n)
- **levelMap** → stores n nodes → O(n)

Given a Binary Tree. Check for the Sum Tree for every node except the leaf node. Return true if it is a Sum Tree otherwise, return false. A Sum-Tree is a Binary Tree where the value of a node is equal to the sum of the nodes present in its left subtree and right subtree. An empty tree is also a Sum Tree as the sum of an empty tree can be considered to be 0. A leaf node is also considered a Sum Tree.

Input:



Output: false

Explanation: The given tree is not a sum tree. For the root node, sum of elements in left subtree is 40 and sum of elements in right subtree is 30. Root element = 10 which is not equal to 30+40.

Optimal:

Algorithm:

Step 1: Define helper structure

- Create a structure (class Pair) to hold **two values** for each node:
    1. isSumTree → whether the subtree rooted at this node satisfies the Sum Tree property
    2. sum → sum of all node values in the subtree

Step 2: Recursive helper function

- Define a recursive function that processes a node and returns its corresponding Pair.

Base cases

1. If the node is null → return (true, 0)
    - A null subtree is considered a Sum Tree with sum 0.
2. If the node is a **leaf node** → return (true, node.data)
    - Leaf nodes are automatically Sum Trees.

Step 3: Recursive step

- For a non-leaf node:
    1. Recursively call the helper function on the **left child**, get (isSumLeft, sumLeft)
    2. Recursively call the helper function on the **right child**, get (isSumRight, sumRight)
    3. Compute whether the **current node satisfies Sum Tree property**:
        - Current node is a Sum Tree if:
            - Left subtree is a Sum Tree
            - Right subtree is a Sum Tree
            - Node's value equals sumLeft + sumRight
    4. Compute **total sum of the subtree** rooted at this node:

- totalSum = node.value + sumLeft + sumRight

5. Return (isCurrentSumTree, totalSum)

Step 4: Main function

- Call the recursive helper on the root node → get (isSumTree, sum)
- Return isSumTree as the result

```
class Pair {
    boolean isSumTree;
    int sum;
    Pair(boolean isSumTree, int sum){
        this.isSumTree = isSumTree;
        this.sum = sum;
    }
}

class Solution {

     static Pair optimal(Node root){
        if(root == null) return new Pair(true, 0);
        if(root.left == null && root.right==null) return new Pair(true, root.data);

        Pair left = optimal(root.left);
        Pair right = optimal(root.right);

        boolean isCurrentSumTree = left.isSumTree && right.isSumTree && (root.data == left.sum + right.sum);
        int totalSum = root.data + left.sum + right.sum;

        return new Pair(isCurrentSumTree, totalSum);
    }

    boolean isSumTree(Node root) {
        Pair res = optimal(root);
        return res.isSumTree;
    }
}
```

Time complexity: O(n)

- Each node is visited exactly once → O(n), where n = number of nodes

Space complexity: O(h)

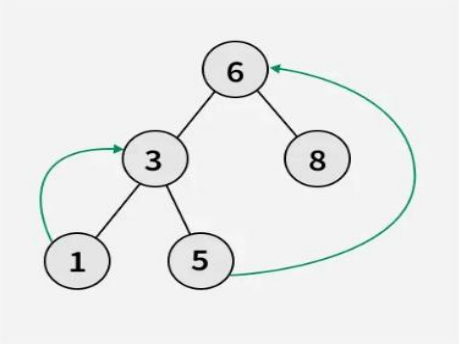
- Recursion stack uses space = height of tree → O(h)

You're given a binary tree where each node has three pointers: left, right, random (can point to *any* node in the tree or null). You need to clone (deep copy) the entire tree — i.e., make a new identical tree where:

- **Each node's data is the same.**
- **Each new node's left, right, and random pointers correctly point to *the corresponding cloned nodes*.**

Note: The output is 1 if the tree is cloned successfully. Otherwise, the output is 0.

Input:



Output: 1
Explanation: The tree was cloned successfully.

Optimal:

When cloning a tree with random pointers, the challenge is:

Random pointers can point to *any node*, not just children.

If we try to copy everything in one traversal, we'll hit a problem — we might not have created the node yet that the random pointer points to.

So, we need to **first ensure all nodes exist**, then **set up their connections**.

Algorithm:

Step 1: Initialize

1. Create an empty HashMap<Tree, Tree> mp to store mappings between original and cloned nodes.

Step 2: Create clone nodes (structure creation)

2. Call a recursive function 'helper' which:
    - If root is null, return.
    - Create a new node temp with the same data as root. Put this mapping into the map.
    - Recursively call left & right node.

✅ After this step, all clone nodes are created and stored in the map,
but their left, right, and random pointers are not yet connected.

Step 3: Connect pointers

3. Call another recursive function 'optimal' which:
    - If root is null, return null.

- Get the cloned node from the map: copy = mp.get(root).
- Recursively connect:
  - copy.left = optimal (root.left, mp)
  - copy.right = optimal (root.right, mp)
  - copy.random = mp.get(root.random)
- Return the cloned node copy.

✅ This step correctly connects each cloned node's left, right, and random pointers using the map.

Step 4: Return the cloned tree

4. In the main function cloneTree(root):
   - Create the map: Map<Tree, Tree> mp = new HashMap<>();
   - Call helper(root, mp) to create all nodes.
   - Return optimal(root, mp) which returns the root of the cloned tree.

```
class Tree{
    int data;
    Tree left,right,random;
    Tree(int d){
        data=d;
        left=null;
        right=null;
        random=null;
    }
}
public class Solution {
    static Tree helper(Tree root, Map<Tree,Tree> mp){
        if(root == null) return null;
        Tree copy = mp.get(root);
        copy.left = helper(root.left, mp);
        copy.right = helper(root.right, mp);
        copy.random = mp.get(root.random);

        return copy;
    }
    static void optimal(Tree root, Map<Tree,Tree> mp){
        if(root == null) return;
        Tree temp = new Tree(root.data);
        mp.put(root,temp);
        optimal(root.left,mp);
        optimal(root.right,mp);
    }

    public Tree cloneTree(Tree root) {
        Map<Tree,Tree> mp = new HashMap<>();
        optimal(root,mp);
        return helper(root,mp);
    }
}
```

Time complexity: O(2n) where ‘n’ is the number of nodes

- Each node is visited twice (once in each recursive function)

Space Complexity: O(n)+O(H) where ‘n’ is the number of nodes & ‘H’ is the height of tree.

- HashMap + recursion stack