**Trees**

A tree is an undirected graph with no cycles.

Equivalently, a tree it is a connected graph with N nodes and N-1 edges.

3 nodes, 2 edges 3 nodes, 2 edges

Tree is a hierarchical data structure which stores the information naturally in the form of hierarchy unlike linear data structures like, Linked List, Stack, etc. A tree contains nodes(data) and connections(edges) which should not form a cycle.

Following are the few frequently used terminologies for Tree data structure.

**Node —** A node is a structure which may contain a value or condition, or represent a separate data structure.

Root — The top node in a tree, the prime ancestor.

**Child —** A node directly connected to another node when moving away from the root, an immediate descendant.

**Parent —** The converse notion of a child, an immediate ancestor.

**Leaf —** A node with no children.

Internal node — A node with at least one child.

**Edge —** The connection between one node and another.

**Depth —** The distance between a node and the root.

**Level —** the number of edges between a node and the root + 1

**Height —** The number of edges on the longest path between a node and a descendant leaf.

**Breadth —** The number of leaves.

**Sub Tree —** A tree T is a tree consisting of a node in T and all of its descendants in T.

**Binary Tree —** is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child.

**Binary Search Tree —** is a special type of binary tree which has the following properties.

The left subtree of a node contains only nodes with keys lesser than the node’s key.

The right subtree of a node contains only nodes with keys greater than the node’s key.

The left and right subtree each must also be a binary search tree.

For the sake of simplicity, we will use Binary Tree as an example to understand Tree Traversal Algorithms. But those algorithms can be generalised to other types of tree, as well.

**Tree Traversal — Introduction**

“In computer science, tree traversal (also known as tree search) is a form of graph traversal and refers to the process of visiting (checking and/or updating) each node in a tree data structure, exactly once. Such traversals are classified by the order in which the nodes are visited.” — Wikipedia

The definition of Wikipedia is self-explanatory to understand what Tree Traversal mean. But I want to elaborate more about the last line of the definition, which will help us to understand the types of Tree Traversal and how they are different.

Tree Traversal Algorithms can be classified broadly in the following two categories by the order in which the nodes are visited:

**Depth-First Search (DFS) Algorithm:**

It starts with the root node and first visits all nodes of one branch as deep as possible of the chosen Node and before backtracking, it visits all other branches in a similar fashion. There are three sub-types under this, which we will cover in this article.

**Breadth-First Search (BFS) Algorithm:**

It also starts from the root node and visits all nodes of current depth before moving to the next depth in the tree. We will cover one algorithm of BFS type in the upcoming section.

Below is the blueprint of our Node class which will act as the atomic member of the Tree Data Structure. We will call it TreeNode, which is holding data as an integer value, left and right children of the same type(TreeNode). You can use any other data structure to keep as data under the TreeNode.

**Inorder Traversal**

Inorder Traversal is the one the most used variant of DFS(Depth First Search) Traversal of the tree.

As DFS suggests, we will first focus on the depth of the chosen Node and then go to the breadth at that level. Therefore, we will start from the root node of the tree and go deeper-and-deeper into the left subtree with recursive manner.

When we will reach to the left-most node with the above steps, then we will visit that current node and go to the left-most node of its right subtree(if exists).

Same steps should be followed in a recursive manner to complete the inorder traversal. Order of those steps will be like (in recursive function)…

Go to left-subtree

Visit Node

Go to right-subtree

public void inorderTraversal(TreeNode root) {

  if (root != null) {

    inorderTraversal(root.left);

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

    inorderTraversal(root.right);

  }

}

**Preorder Traversal**

Preorder Traversal is another variant of DFS. Where atomic operations in a recursive function, are as same as Inorder traversal but with a different order.

Here, we visit the current node first and then goes to the left sub-tree. After covering every node of the left sub-tree, we will move towards the right sub-tree and visit in a similar fashion.

Order of the steps will be like…

Visit Node

Go to left-subtree

Go to right-subtree

public void preorderTraversal(TreeNode root) {

  if (root != null) {

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

    preorderTraversal(root.left);

    preorderTraversal(root.right);

  }

}

**Postorder Traversal**

Similar goes with Postorder Traversal. Where we visit the left subtree and the right subtree before visiting the current node in recursion.

So, the sequence of the steps will be…

Go to left-subtree

Go to right-subtree

Visit Node

public void postorderTraversal(TreeNode root) {

  if (root != null) {

    postorderTraversal(root.left);

    postorderTraversal(root.right);

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

  }

}

**Level Order Traversal**

This is a different traversal than what we have covered above. Level order traversal follows BFS(Breadth-First Search) to visit/modify every node of the tree.

As BFS suggests, the breadth of the tree takes priority first and then move to depth. In simple words, we will visit all the nodes present at the same level one-by-one from left to right and then move to the next level to visit all the nodes of that level.

public void levelorderTraversal(TreeNode root) {

  if (root == null) {

    return;

  }

  Queue<TreeNode> queue = new LinkedList<>();

  queue.add(root);

  while (!queue.isEmpty()) {

    TreeNode node = queue.remove();

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

    if (node.left != null) {

      queue.add(node.left);

    }

    if (node.right != null) {

      queue.add(node.right);

    }

  }

}

All in one go:

package trees;

/\*\*

 \* @author dmalladi

 \*/

public class TreeNode {

    int data;

    TreeNode left;

    TreeNode right;

    public TreeNode(int data) {

        this.data = data;

        left = right = null;

    }

}

package trees;

public class TreeTraversals {

    public static void main(String[] args) {

        TreeNode tree = new TreeNode(10);

        tree.left = new TreeNode(8);

        tree.right = new TreeNode(20);

        tree.left.left = new TreeNode(4);

        tree.left.right = new TreeNode(6);

        tree.right.left = new TreeNode(15);

        System.out.println("inorder:");

        printInorder(tree);

        System.out.println("postorder:");

        printPostOrder(tree);

        System.out.println("preorder:");

        printPreOrder(tree);

    }

    public static void printInorder(TreeNode root) {

        inorder(root);

        System.out.println();

    }

    public static void printPostOrder(TreeNode root) {

        postOrder(root);

        System.out.println();

    }

    public static void printPreOrder(TreeNode root) {

        preOrder(root);

        System.out.println();

    }

    private static void postOrder(TreeNode root) {

        if (root != null) {

            postOrder(root.left);

            postOrder(root.right);

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

        }

    }

    private static void preOrder(TreeNode root) {

        if (root != null) {

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

            preOrder(root.left);

            preOrder(root.right);

        }

    }

    private static void inorder(TreeNode root) {

        if (root != null) {

            inorder(root.left);

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

            inorder(root.right);

        }

    }

}

**Iterative Inorder Traversal**

Using Stack is the obvious way to traverse tree without recursion. Below is an algorithm for traversing binary tree using stack.

1) Create an empty stack S.

2) Initialize current node as root

3) Push the current node to S and set current = current->left until current is NULL

4) If current is NULL and stack is not empty then

a) Pop the top item from stack.

b) Print the popped item, set current = popped\_item->right

c) Go to step 3.

5) If current is NULL and stack is empty then we are done.

**Code**

public List <Integer> inorderTraversal(TreeNode root) {

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

        Stack <TreeNode> stack = new Stack <> ();

        TreeNode curr = root;

        while (curr != null || !stack.isEmpty()) {

            while (curr != null) {

                stack.push(curr);

                curr = curr.left;

            }

            curr = stack.pop();

            res.add(curr.val);

            curr = curr.right;

        }

        return res;

    }

**Complexity Analysis**

* Time complexity : *O*(*n*).
* Space complexity : *O*(*n*).

**Iterative Preorder Traversal**

Let's start from the root and then at each iteration pop the current node out of the stack and push its child nodes. In the implemented strategy we push nodes into output list following the order Top->Bottom and Left->Right, that naturally reproduces preorder traversal.

Code

public List<Integer> preorderTraversal(TreeNode root) {

        LinkedList<TreeNode> stack = new LinkedList<>();

        LinkedList<Integer> output = new LinkedList<>();

        if (root == null) {

            return output;

        }

        stack.add(root);

        while (!stack.isEmpty()) {

            TreeNode node = stack.pollLast(); //stack.peek();

            output.add(node.data);

            if (node.right != null) {

                stack.add(node.right);

            }

            if (node.left != null) {

                stack.add(node.left);

            }

        }

        return output;

    }

**Complexity Analysis**

* Time complexity : we visit each node exactly once, thus the time complexity is O(*N*), where *N* is the number of nodes, i.e. the size of tree.
* Space complexity : depending on the tree structure, we could keep up to the entire tree, therefore, the space complexity is O(*N*).

**Iterative Postorder Traversal**

**1. Using 1 Stack.** O(n) Time & O(n) Space  
This is similar to in-order using 1 stack. The difference is we keep track of the previously printed node in pre. And we only print a node if its right child is null or equal to pre.

* Push all left nodes into the stack till it hits NULL.
* root = s.peek()
* if root.right = null or pre (Means we have traversed the right subtree already)
  + We print root and pop it from s.
  + Make pre = root
  + root = null (So we dont go down to left child again)
* else
  + root = root.right (Traverse the right subtree before printing root)
* Keep iterating till both the below conditions are met -
  + Stack is empty and
  + Root is NULL.

public List<Integer> postorderTraversal(TreeNode root) {

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

        if(root==null)

            return out;

        TreeNode pre=null;

        Stack<TreeNode> s = new Stack();

        while(root!=null || !s.empty()){

            if(root!=null){

                s.push(root);

                root = root.left;

            }

            else{

                root = s.peek();

                if(root.right==null || root.right==pre){

                    out.add(root.data);

                    s.pop();

                    pre=root;

                    root = null;

                }

                else

                    root = root.right;

            }

        }

        return out;

    }

**2. Using 2 Stacks.** O(n) Time & O(n) Space  
We use two stacks. Stack s is used to find and traverse the child nodes, and path stack keeps track of the path from the root to the current node. (This is usefull in certain problems like [Binary Tree Paths](https://leetcode.com/problems/binary-tree-paths/) and [Path Sum](https://leetcode.com/problems/path-sum/) ).

* Initially we push the root into s.
* Keep iterating with below logic till s is empty.
  + root = s.peek()
  + If the top elements of both the stacks are not the same :
    - Print root and push it into path.
    - Push root's children into s in reverse order. (Remember it's a stack!)
  + When top elements of both stacks are equal. (Which means we hit a deadend, and need to turn back)
    - Pop from both stacks.

public List<Integer> postorderTraversal(TreeNode root) {

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

        if(root == null)

            return out;

        Stack<TreeNode> s = new Stack(), path = new Stack();

        s.push(root);

        while(!s.empty()){

            root = s.peek();

            if(!path.empty() && path.peek()==root){

                out.add(root.data);

            s.pop();

                path.pop();

            }

            else{

                path.push(root);

                if(root.right != null)

                    s.push(root.right);

                if(root.left != null)

                    s.push(root.left);

            }

        }

        return out;

    }

**Another approach:**

1. Create two stacks s an out and push root node onto s
2. While stack s is not empty
   1. op from stack s, current = s.pop
   2. 2. Put current onto stack out.
   3. Put left and right child of current on to stack s
3. Pop everything from out stack and process it.

public void postOrderTraversal(){

        postOrderIterative(root);

    }

    private void postOrderIterative(TreeNode root){

        Stack<TreeNode> out = new Stack<>();

        Stack<TreeNode> s = new Stack<>();

        s.push(root);

        while(!s.empty()){

            Node current = s.pop();

            out.push(current);

            if(current.left != null) s.push(current.left);

            if(current.right != null) s.push(current.right);

        }

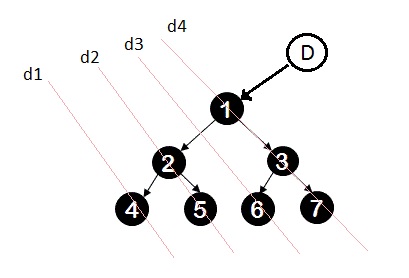
        while(!out.empty()){

            System.out.println(out.pop().data);

        }

    }

**3.Using No Stacks (Morris Traversal).** O(n) Time & O(1) Space  
Instead of using stacks to remember our way back up the tree, we are going to modify the tree to create upwards links. The idea is based on Threaded binary tree.



We reverse each diagonal shown in the picture (d1-d4), print it and re-reverse it.

* Create a dummy node and make dummy.left = root.
* root = dummy
* Iterate till root is null.
  + If root has a left child.
    - Find the inorder predecessor => pre. (Inorder predecessor of root is the right most child of its left child)
      * pre.right = root (Make it point to root).
      * root = root.left.
    - If its already pointing to root (which means we have traversed it already and are on our way up.)
      * Reverse from root.left to pre.
      * Traverse from pre to root.left and print the nodes.
      * Re-reverse it back to normal.
      * pre.right = null.
      * root = root.right.
  + If left child is null
    - root = root.right. (We are climbing up our link.)

public List<Integer> postorderTraversal(TreeNode root) {

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

    if(root == null)

        return out;

    TreeNode dummy = new TreeNode(-1), pre = null;

    dummy.left = root; root = dummy;

    while(root != null){

        if(root.left != null){

            pre = root.left;

            while(pre.right != null && pre.right != root)

                pre=pre.right;

            if(pre.right == null){

                pre.right = root;

                root = root.left;

            }

            else{

                TreeNode node = pre;

                reverse(root.left,pre);

                while(node != root.left){

                    out.add(node.val);

                    node = node.right;

                }

                out.add(node.val);          // Print again since we are stopping at node=root.left

                reverse(pre,root.left);

                pre.right = null;

                root = root.right;

            }

        }

        else{

            root = root.right;

        }

    }

    return out;

}

public void reverse(TreeNode from, TreeNode to){

    if(from == to)

        return;

    TreeNode prev = from, node = from.right;

    while(prev != to){

        TreeNode next = node.right;

        node.right = prev;

        prev = node;

        node = next;

    }

}

Inorder patterns:

An iterative in-order traversal using stack to solve multiple tree problems:

Just remind the iterative in-order traversal.

public List<Integer> inorderTraversal(TreeNode root) {

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

        Stack<TreeNode> nodes = new Stack<>();

        if(root == null) return inorder;

        while(root != null || !nodes.empty()){

            //push left children if available

            while(root != null){

                nodes.push(root);

                root = root.left;

            }

            //retrieve top node and store its right child if exists

            root = nodes.pop();

            inorder.add(root.data);

            root = root.right;

        }

        return inorder;

    }

**Validate if tree is a BST**

For a binary tree to be a BST, the inorder has to be in sorted (ascending) order.

**Iterative approach:**

public bool isValidBST(TreeNode root) {

        if(root == null) return true;

        Stack<TreeNode> s = new Stack<>();

        TreeNode prev = null;

        while(root != null || !s.empty()){

            while(root != null){

                s.push(root);

                root = root.left;

            }

            root = s.pop();

            if(prev != null && prev.data >= root.data)

              return false;

            prev = root;

            root = root.right;

        }

        return true;

    }

**In-order Recursive approach:**

boolean isBST()  {

        return isBSTUtil(root, Integer.MIN\_VALUE,

                               Integer.MAX\_VALUE);

    }

    // Returns true if the given tree is a BST and its

    //  values are >= min and <= max.

    boolean isBSTUtil(TreeNode node, int min, int max)

    {

        // an empty tree is BST

        if (node == null)

            return true;

        // process the root first

        // false if this node violates the min/max constraints

        if (node.data < min || node.data > max)

            return false;

        // process left node then right node

        // otherwise check the subtrees recursively

        //tightening the min/max constraints

        // Allow only distinct values

        return (isBSTUtil(node.left, min, node.data-1) &&

                isBSTUtil(node.right, node.data+1, max));

    }

**Kth smallest element in BST**

Inorder traversal of a BST gives us a sorted order of the items in it. So a simple inorder breaking off at the kth item would give us our answer.

Given a binary search tree, write a function kthSmallest to find the **k**th smallest element in it.

**Example 1:**

**Input:** root = [3,1,4,null,2], k = 1

3

/ \

1 4

\

  2

**Output:** 1

**Example 2:**

**Input:** root = [5,3,6,2,4,null,null,1], k = 3

5

/ \

3 6

/ \

2 4

/

1

**Output:** 3

**Follow up:**  
What if the BST is modified (insert/delete operations) often and you need to find the kth smallest frequently? How would you optimize the kthSmallest routine?

**Constraints:**

* The number of elements of the BST is between 1 to 10^4.
* You may assume k is always valid, 1 ≤ k ≤ BST's total elements.

## **Solution**

#### **How to traverse the tree**

There are two general strategies to traverse a tree:

* Depth First Search (DFS)

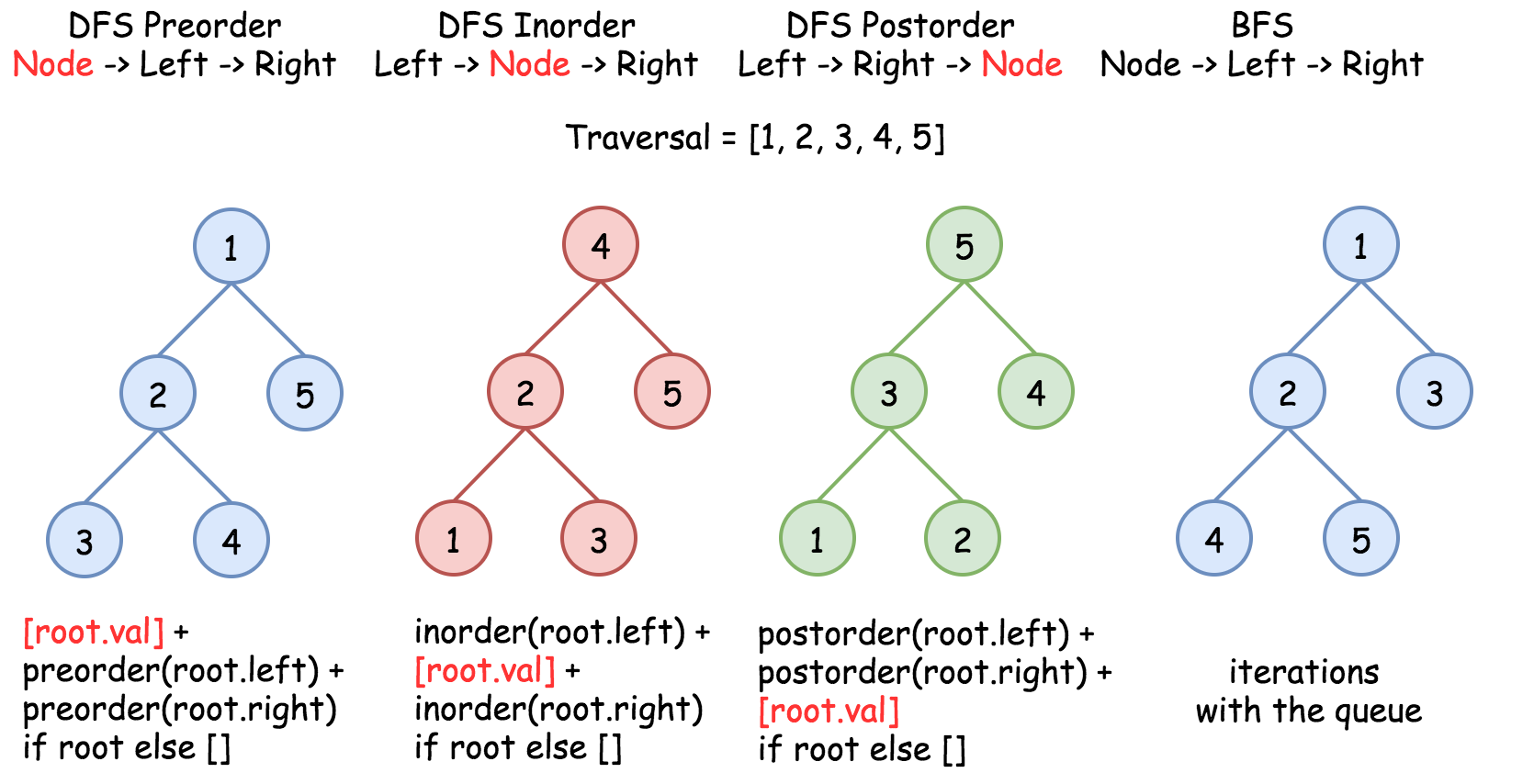
In this strategy, we adopt the depth as the priority, so that one would start from a root and reach all the way down to certain leaf, and then back to root to reach another branch.

The DFS strategy can further be distinguished as preorder, inorder, and postorder depending on the relative order among the root node, left node and right node.

* Breadth First Search (BFS)

We scan through the tree level by level, following the order of height, from top to bottom. The nodes on higher level would be visited before the ones with lower levels.

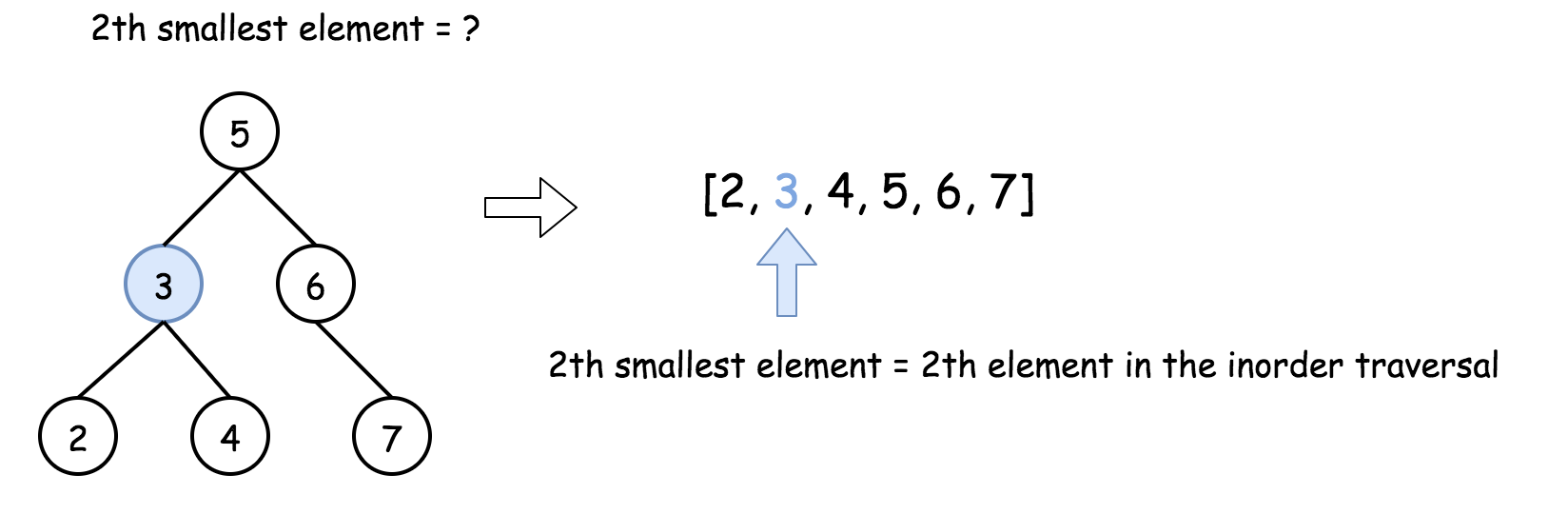
On the following figure the nodes are numerated in the order you visit them, please follow 1-2-3-4-5 to compare different strategies.



To solve the problem, one could use the property of BST : inorder traversal of BST is an array sorted in the ascending order.

#### **Approach 1: Recursion**

It's a very straightforward approach with \mathcal{O}(N)O(*N*) time complexity. The idea is to build an inorder traversal of BST which is an array sorted in the ascending order. Now the answer is the k - 1th element of this array.



public ArrayList<Integer> inorder(TreeNode root, ArrayList<Integer> arr) {

        if (root == null)

            return arr;

        inorder(root.left, arr);

        arr.add(root.val);

        inorder(root.right, arr);

        return arr;

    }

    public int kthSmallest(TreeNode root, int k) {

        ArrayList<Integer> nums = inorder(root, new ArrayList<Integer>());

        return nums.get(k - 1);

    }

One drawback with this approach is that we need to process all elements, even though if we already process the kth smallest element. We can terminate the process once we find the kth smallest element.

Below approach is more efficient than the above one.

public static int kthSmallest(TreeNode root, int k) {

        int[] nums = new int[2];

        inorder(root,nums,k);

        return nums[1];

    }

    public static void inorder(TreeNode root,int[] nums,int k) {

        if (root == null)

            return;

        inorder(root.left,nums,k);

        //process the root

        if (++nums[0] == k) {

            nums[1] = root.data;

            return;

        }

        inorder(root.right,nums,k);

    }

#### **Approach 2: Iteration**

The above recursion could be converted into iteration, with the help of stack. This way one could speed up the solution because there is no need to build the entire inorder traversal, and one could stop after the kth element.

public int kthSmallest(TreeNode root, int k) {

        Stack<TreeNode> s = new Stack<>();

        while(root != null || !s.empty()){

            while(root != null){

                s.push(root);

                root = root.left;

            }

            root = s.pop();

            if(--k == 0)

 return root.data;

            root = root.right;

        }

        return -1;

    }

**Size of a Binary Tree**

Size() function recursively calculates the size of a tree. It works as follows:

Size of a tree = Size of left subtree + 1 + Size of right subtree.

Algorithm:

size(tree)

1. If tree is empty then return 0

2. Else

(a) Get the size of left subtree recursively i.e., call

size( tree->left-subtree)

(a) Get the size of right subtree recursively i.e., call

size( tree->right-subtree)

(c) Calculate size of the tree as following:

tree\_size = size(left-subtree) + size(right-

subtree) + 1

(d) Return tree\_size

#### **Approach 1: Recursion**

1. Start from the root.

2. Size = 1 (for the root) + Size Of left Sub-Tree + Size Of right Sub-Tree

3. solve the left sub-tree and right sub-tree recursively.

public static int size(TreeNode root) {

        if (root == null)

            return 0;

        // inorder size(root.left) + 1 + size(root.right)

        // preorder 1 + size(root.left) + size(root.right)

        // postorder size(root.left) + size(root.right) + 1

        return size(root.left) + 1 + size(root.right);

    }

#### **Approach 2: Iteration**

It's a very straightforward approach with  O(*N*) time complexity. The idea is to build an inorder traversal of Binary Tree and increment the count while poping out the node from stack.

public static int iterativeSize(TreeNode root) {

        int count = 0;

        Stack<TreeNode> stack = new Stack<>();

        while(root != null || !stack.isEmpty()) {

            while(root != null) {

                stack.push(root);

                root = root.left;

            }

            root = stack.pop();

            count++;

            root = root.right;

        }

        return count;

    }

**Maximum depth or Height of a Tree**

Recursively calculate height of left and right subtrees of a node and assign height to the node as max of the heights of two children plus 1. See below pseudo code and program for details.

In post-order traversal, we process the left sub-tree, right sub-tree then we visit the root or process the root. In bottom up approaches, we use post-order traversals, where first processing both left and right subtrees is required before processing the root.

Post-order :

* 1. Process left sub-tree
  2. Process right sub-tree
  3. Process the root

Algorithm:

maxDepth()

1. If tree is empty then return 0

2. Else

(a) Get the max depth of left subtree recursively i.e.,

call maxDepth( tree->left-subtree)

(a) Get the max depth of right subtree recursively i.e.,

call maxDepth( tree->right-subtree)

(c) Get the max of max depths of left and right

subtrees and add 1 to it for the current node.

max\_depth = max(max dept of left subtree,

max depth of right subtree)

+ 1

(d) Return max\_depth

public static int height(TreeNode root) {

        if (root == null)

            return 0;

        //process left sub-tree

        int leftHeight = height(root.left);

        //process right sub-tree

        int rightHeight = height(root.right);

        //process root

        if (leftHeight < rightHeight)

            return rightHeight + 1;

        else

            return leftHeight+1;

    }

More simply,

public static int height(TreeNode root) {

        if (root == null)

            return 0;

        //process left sub-tree

        int leftHeight = height(root.left);

        //process right sub-tree

        int rightHeight = height(root.right);

        //process root

        return Math.max(leftHeight,rightHeight) + 1;

    }

While processing the root, sometimes we add 1 to some variables or recursive subtree values.

**Min depth of a Tree**

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.

**Note:** A leaf is a node with no children.

**Example:**

Given binary tree [3,9,20,null,null,15,7],

3

/ \

9 20

/ \

15 7

return its minimum depth = 2.

## **Solution**

#### **Approach 1: Recursion**

**Algorithm**

The intuitive approach is to solve the problem by recursion. Here we demonstrate an example with the DFS (Depth First Search) strategy.

public static int minDepth(TreeNode root) {

        if (root == null)

            return 0;

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

            return 1;

        // If left subtree is NULL, recur for right subtree

        if (root.left == null)

            return minDepth(root.right) + 1;

        // If right subtree is NULL, recur for left subtree

        if (root.right == null)

            return minDepth(root.left) + 1;

        int leftMinDepth = minDepth(root.left);

        int rightMinDepth = minDepth(root.right);

        return Math.min(leftMinDepth,rightMinDepth) + 1;

    }

Another method,

public int minDepth(TreeNode root) {

        if (root == null) {

        return 0;

        }

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

        return 1;

        }

        int min\_depth = Integer.MAX\_VALUE;

        if (root.left != null) {

        min\_depth = Math.min(minDepth(root.left), min\_depth);

        }

        if (root.right != null) {

        min\_depth = Math.min(minDepth(root.right), min\_depth);

        }

        return min\_depth + 1;

    }

More simply,

public int minDepth(TreeNode root) {

        if(root==null)

            return 0;

        return depth(root);

    }

    public static int depth(TreeNode root)

    {

        if(root==null)

            return Integer.MAX\_VALUE;

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

            return 1;

        int l=depth(root.left);

        int r=depth(root.right);

        return 1+Math.min(l,r);

    }

**Complexity analysis**

* Time complexity: we visit each node exactly once, thus the time complexity is O(*N*), where N*N* is the number of nodes.
* Space complexity: in the worst case, the tree is completely unbalanced, e.g. each node has only one child node, the recursion call would occur N*N* times (the height of the tree), therefore the storage to keep the call stack would be O(*N*). But in the best case (the tree is completely balanced), the height of the tree would be log(*N*). Therefore, the space complexity in this case would be O(log(*N*)).

#### **Approach 2: BFS Iteration**

We iterate the tree level by level, and the first leaf we reach corresponds to the minimum depth. As a result, we do not need to iterate all nodes.

public int minDepth(TreeNode root) {

        int minDepth = 0;

        if (root == null) {

            return minDepth;

        }

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

        TreeNode cur = root;

        q.offer(cur);

        while (!q.isEmpty()) {

            minDepth += 1;

            int n = q.size();

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

                cur = q.poll();

                if (cur.left != null) {

                    q.offer(cur.left);

                }

                if (cur.right != null) {

                    q.offer(cur.right);

                }

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

                    q.clear();

                    break;

                }

            }

        }

        return minDepth;

    }

More simply,

public int minDepth(TreeNode root) {

        if (root == null)   return 0;

        int depth = 1;

        Queue<TreeNode> queue = new ArrayDeque<>();

        queue.offer(root);

        while (!queue.isEmpty()) {

            int size = queue.size();

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

                TreeNode curr = queue.poll();

                if (isLeafNode(curr))   return depth;

                if (curr.left != null)  queue.offer(curr.left);

                if (curr.right != null) queue.offer(curr.right);

            }

            depth += 1;

        }

        return -1;

    }

    private boolean isLeafNode(TreeNode curr) {

        return curr != null && curr.left == null && curr.right == null;

    }

**Find nth node of in-order traversal**

Given the binary tree and you have to find out the n-th node of inorder traversal.

**Example:**

Given binary tree [10,20,30,40,null,50,null],

Input : n = 4

10

/

20 30

/

40 50

Output : 10

Inorder Traversal is : 40 20 50 10 30

Input : n = 3

7

/

2 3

/

8 5

Output : 8

Inorder: 2 7 8 3 5

3th node is 8return its minimum depth = 2.

static int count = 0;

    private static void printNthNode(TreeNode root,int n) {

        if (root == null)

            return;

        printNthNode(root.left,n);

        count++;

        if (count == n) {

            System.out.println(root.data);

        }

        printNthNode(root.right,n);

    }

If you want to return node,

static int count = 0;

    public static TreeNode nthNode(TreeNode root,int n) {

        return nthNodeUtil(root,n);

    }

    private static TreeNode nthNodeUtil(TreeNode root,int n) {

        if (root == null)

            return root;

        TreeNode leftNode = nthNodeUtil(root.left,n);

        if (leftNode != null)

            return leftNode;

        count++;

        if (count == n) {

            return root;

        }

        TreeNode rightNode = nthNodeUtil(root.right,n);

        if (rightNode != null)

            return rightNode;

        return null;

    }

**Find nth node of post-order traversal**

**Find nth node of pre-order traversal**

**Boundary traversal of Binary Tree**

Given a binary tree, return the values of its boundary in **anti-clockwise** direction starting from root. Boundary includes left boundary, leaves, and right boundary in order without duplicate **nodes**.  (The values of the nodes may still be duplicates.)

**Left boundary** is defined as the path from root to the **left-most** node. **Right boundary** is defined as the path from root to the **right-most** node. If the root doesn't have left subtree or right subtree, then the root itself is left boundary or right boundary. Note this definition only applies to the input binary tree, and not applies to any subtrees.

The **left-most** node is defined as a **leaf** node you could reach when you always firstly travel to the left subtree if exists. If not, travel to the right subtree. Repeat until you reach a leaf node.

The **right-most** node is also defined by the same way with left and right exchanged.

**Example 1**

**Input:**

1

\

2

/ \

3 4

**Ouput:**

[1, 3, 4, 2]

**Explanation:**

The root doesn't have left subtree, so the root itself is left boundary.

The leaves are node 3 and 4.

The right boundary are node 1,2,4. Note the anti-clockwise direction means you should output reversed right boundary.

So order them in anti-clockwise without duplicates and we have [1,3,4,2].

**Example 2**

**Input:**

\_\_\_\_1\_\_\_\_\_

/ \

2 3

/ \ /

4 5 6

/ \ / \

7 8 9 10

**Ouput:**

[1,2,4,7,8,9,10,6,3]

**Explanation:**

The left boundary are node 1,2,4. (4 is the left-most node according to definition)

The leaves are node 4,7,8,9,10.

The right boundary are node 1,3,6,10. (10 is the right-most node).

So order them in anti-clockwise without duplicate nodes we have [1,2,4,7,8,9,10,6,3].

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

public List<Integer> boundaryOfBinaryTree(TreeNode root) {

    if(root == null)

        return res;

    res.add(root.val);

    processLeftSubTree(root.left);

    processLeafs(root.left);

    processLeafs(root.right);

    processRightSubTree(root.right);

    return res;

}

private void processLeafs(TreeNode node){

    if(node == null)

        return;

    processLeafs(node.left);

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

        res.add(node.val);

    }

    processLeafs(node.right);

}

private void processLeftSubTree(TreeNode node){

    if(node == null)

        return;

    if(node.left != null){

        res.add(node.val);

        processLeftSubTree(node.left);

    }else if(node.right != null){

        res.add(node.val);

        processLeftSubTree(node.right);

    }

}

private void processRightSubTree(TreeNode node){

    if(node == null)

        return;

    if(node.right != null){

        processRightSubTree(node.right);

        res.add(node.val);

    }else if(node.left != null){

        processRightSubTree(node.left);

        res.add(node.val);

    }

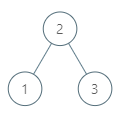
}

**In-order successor of a Binary Search Tree**

Given a binary search tree and a node in it, find the in-order successor of that node in the BST.

The successor of a node p is the node with the smallest key greater than p.val.

**Example 1:**

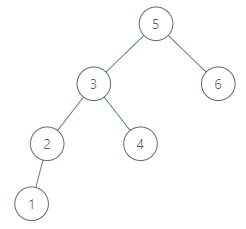


**Input:** root = [2,1,3], p = 1

**Output:** 2

**Explanation:** 1's in-order successor node is 2. Note that both p and the return value is of TreeNode type.

**Example 2:**



**Input:** root = [5,3,6,2,4,null,null,1], p = 6

**Output:** null

**Explanation:** There is no in-order successor of the current node, so the answer is

null.

public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {

        // the successor is somewhere lower in the right subtree

        // successor: one step right and then left till you can

        if (p.right != null) {

            p = p.right;

            while (p.left != null) p = p.left;

            return p;

        }

        // the successor is somewhere upper in the tree

        ArrayDeque<TreeNode> stack = new ArrayDeque<>();

        int inorder = Integer.MIN\_VALUE;

        // inorder traversal : left -> node -> right

        while (!stack.isEmpty() || root != null) {

            // 1. go left till you can

            while (root != null) {

                stack.push(root);

                root = root.left;

            }

            // 2. all logic around the node

            root = stack.pop();

            // if the previous node was equal to p

            // then the current node is its successor

            if (inorder == p.val) return root;

            inorder = root.val;

            // 3. go one step right

            root = root.right;

        }

        // there is no successor

        return null;

    }

More simply,

public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {

        TreeNode candidate = null;

        TreeNode cur = root;

        while (cur != null) {

            if (cur.val > p.val) {

                candidate = cur;

                cur = cur.left;

            } else {

                // cur.val <= p.val

                cur = cur.right;

            }

        }

        return candidate;

    }

Still we need to check if the p.right has the next or not.

public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {

        if (root == null) return null;

        TreeNode next = findRight(p.right);

        if (next != null) return next;

        TreeNode candidate = null;

        while (root != null) {

            if (root.val > p.val) {

                candidate = root;

                root = root.left;

            } else {

                root = root.right;

            }

        }

        return candidate;

    }

    private TreeNode findRight(TreeNode curr) {

        while (curr != null && curr.left != null)

            curr = curr.left;

        return curr;

    }

# Level Order traversal

This pattern is based on the **Breadth First Search (BFS)** technique to traverse a tree.

Any problem involving the traversal of a tree in a level-by-level order can be efficiently solved using this approach. We will use a **Queue** to keep track of all the nodes of a level before we jump onto the next level. This also means that the space complexity of the algorithm will be O(W)*O*(*W*), where ‘W’ is the maximum number of nodes on any level.

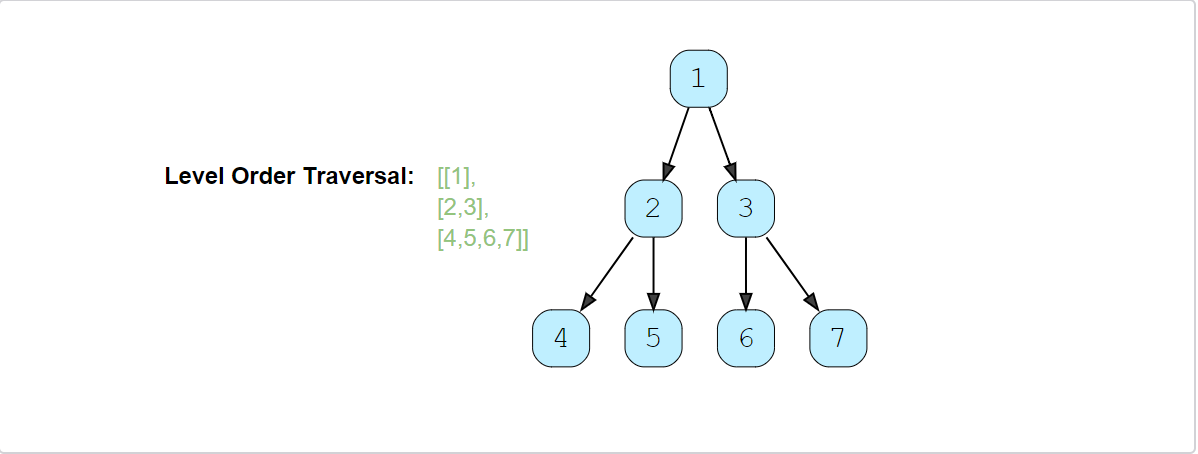
Let’s jump onto our first problem to understand this pattern.

# Binary Tree Level Order Traversal (easy)

### Problem Statement [#](https://www.educative.io/courses/grokking-the-coding-interview/xV7E64m4lnz#problem-statement)

Given a binary tree, populate an array to represent its level-by-level traversal. You should populate the values of all **nodes of each level from left to right** in separate sub-arrays.

**Example 1:**



**Example 2:**

### Try

import java.util.\*;

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode(int x) {

    val = x;

  }

};

class LevelOrderTraversal {

  public static List<List<Integer>> traverse(TreeNode root) {

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

    // TODO: Write your code here

    return result;

  }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(12);

    root.left = new TreeNode(7);

    root.right = new TreeNode(1);

    root.left.left = new TreeNode(9);

    root.right.left = new TreeNode(10);

    root.right.right = new TreeNode(5);

    List<List<Integer>> result = LevelOrderTraversal.traverse(root);

    System.out.println("Level order traversal: " + result);

  }

}

**Solution**[#](https://www.educative.io/courses/grokking-the-coding-interview/xV7E64m4lnz#solution)

Since we need to traverse all nodes of each level before moving onto the next level, we can use the **Breadth First Search (BFS)** technique to solve this problem.

We can use a Queue to efficiently traverse in BFS fashion. Here are the steps of our algorithm:

1. Start by pushing the root node to the queue.
2. Keep iterating until the queue is empty.
3. In each iteration, first count the elements in the queue (let’s call it levelSize). We will have these many nodes in the current level.
4. Next, remove levelSize nodes from the queue and push their value in an array to represent the current level.
5. After removing each node from the queue, insert both of its children into the queue.
6. If the queue is not empty, repeat from step 3 for the next level.

Let’s take the example-2 mentioned above to visually represent our algorithm:

### Code

import java.util.\*;

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode(int x) {

    val = x;

  }

};

class LevelOrderTraversal {

  public static List<List<Integer>> traverse(TreeNode root) {

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

    if (root == null)

      return result;

    Queue<TreeNode> queue = new LinkedList<>();

    queue.offer(root);

    while (!queue.isEmpty()) {

      int levelSize = queue.size();

      List<Integer> currentLevel = new ArrayList<>(levelSize);

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

        TreeNode currentNode = queue.poll();

        // add the node to the current level

        currentLevel.add(currentNode.val);

        // insert the children of current node in the queue

        if (currentNode.left != null)

          queue.offer(currentNode.left);

        if (currentNode.right != null)

          queue.offer(currentNode.right);

      }

      result.add(currentLevel);

    }

    return result;

  }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(12);

    root.left = new TreeNode(7);

    root.right = new TreeNode(1);

    root.left.left = new TreeNode(9);

    root.right.left = new TreeNode(10);

    root.right.right = new TreeNode(5);

    List<List<Integer>> result = LevelOrderTraversal.traverse(root);

    System.out.println("Level order traversal: " + result);

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/xV7E64m4lnz#time-complexity)

The time complexity of the above algorithm is O(N)*O*(*N*), where ‘N’ is the total number of nodes in the tree. This is due to the fact that we traverse each node once.

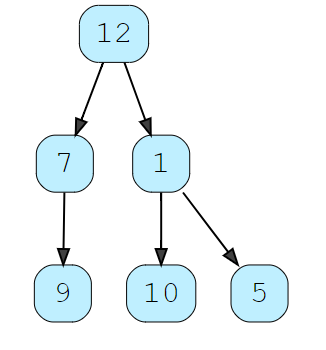
#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/xV7E64m4lnz#space-complexity)

The space complexity of the above algorithm will be O(N)*O*(*N*) as we need to return a list containing the level order traversal. We will also need O(N)*O*(*N*) space for the queue. Since we can have a maximum of N/2*N*/2 nodes at any level (this could happen only at the lowest level), therefore we will need O(N)*O*(*N*) space to store them in the queue.

### Problem Statement 2

Given a binary tree, find its size using level order traversal.

**Example 1:**



Output : 6

### Try

import java.util.\*;

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode(int x) {

    val = x;

  }

};

class LevelOrderSize {

  public static int iterSize(TreeNode root) {

    // TODO: Write your code here

    return 0;

  }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(12);

    root.left = new TreeNode(7);

    root.right = new TreeNode(1);

    root.left.left = new TreeNode(9);

    root.right.left = new TreeNode(10);

    root.right.right = new TreeNode(5);

    List<List<Integer>> result = LevelOrderSize.iterSize(root);

    System.out.println("Level order traversal: " + result);

  }

}

**Solution**[#](https://www.educative.io/courses/grokking-the-coding-interview/xV7E64m4lnz#solution)

Since we need to traverse all nodes of each level before moving onto the next level, we can use the **Breadth First Search (BFS)** technique to solve this problem.

### Code

import java.util.\*;

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode(int x) {

    val = x;

  }

};

class LevelOrderSize {

public static int iterSize(TreeNode root) {

          if (root == null)

            return 0;

        Queue<TreeNode> queue = new LinkedList<>();

        queue.offer(root);

        int count = 0;

        while (!queue.isEmpty()) {

            int levelSize = queue.size();

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

                TreeNode currentNode = queue.poll();

                count++;

                // insert the children of current node in the queue

                if (currentNode.left != null)

                    queue.offer(currentNode.left);

                if (currentNode.right != null)

                    queue.offer(currentNode.right);

            }

        }

        return count;

   }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(12);

    root.left = new TreeNode(7);

    root.right = new TreeNode(1);

    root.left.left = new TreeNode(9);

    root.right.left = new TreeNode(10);

    root.right.right = new TreeNode(5);

    List<List<Integer>> result = LevelOrderSize.iterSize(root);

    System.out.println("Level order traversal: " + result);

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/xV7E64m4lnz#time-complexity)

The time complexity of the above algorithm is O(N)*O*(*N*), where ‘N’ is the total number of nodes in the tree. This is due to the fact that we traverse each node once.

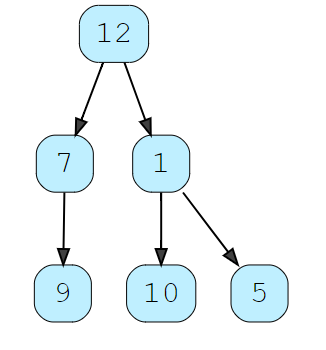
#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/xV7E64m4lnz#space-complexity)

The space complexity of the above algorithm will be O(N)*O*(*N*) as we need to return a list containing the level order traversal. We will also need O(N)*O*(*N*) space for the queue. Since we can have a maximum of N/2*N*/2 nodes at any level (this could happen only at the lowest level), therefore we will need O(N)*O*(*N*) space to store them in the queue.

### Problem Statement 3

Given a binary tree, find all leaf nodes in it.

**Example 1:**



Output : [9,10,5]

### Try

import java.util.\*;

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode(int x) {

    val = x;

  }

};

class LevelOrderTraversal {

  public static List<Integer> leafNodes(TreeNode root) {

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

    // TODO: Write your code here

    return result;

  }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(12);

    root.left = new TreeNode(7);

    root.right = new TreeNode(1);

    root.left.left = new TreeNode(9);

    root.right.left = new TreeNode(10);

    root.right.right = new TreeNode(5);

    List<Integer> result = LevelOrderTraversal.leadNodes(root);

    System.out.println("Level order traversal: " + result);

  }

}

**Solution**[#](https://www.educative.io/courses/grokking-the-coding-interview/xV7E64m4lnz#solution)

Since we need to traverse all nodes of each level before moving onto the next level, we can use the **Breadth First Search (BFS)** technique to solve this problem.

### Code

import java.util.\*;

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode(int x) {

    val = x;

  }

};

class LevelOrderTraversal {

public static List<Integer> leafNodes(TreeNode root) {

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

        if (root == null)

            return result;

        Queue<TreeNode> queue = new LinkedList<>();

        queue.offer(root);

        while (!queue.isEmpty()) {

            int levelSize = queue.size();

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

                TreeNode currentNode = queue.poll();

                // add the node to the result if it is a leaf node

                if (currentNode.left == null && currentNode.right == null)

                    result.add(currentNode.data);

                // insert the children of current node in the queue

                if (currentNode.left != null)

                    queue.offer(currentNode.left);

                if (currentNode.right != null)

                    queue.offer(currentNode.right);

            }

        }

        return result;

    }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(12);

    root.left = new TreeNode(7);

    root.right = new TreeNode(1);

    root.left.left = new TreeNode(9);

    root.right.left = new TreeNode(10);

    root.right.right = new TreeNode(5);

    List<Integer> result = LevelOrderTraversal.leadNodes(root);

    System.out.println("Level order traversal: " + result);

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/xV7E64m4lnz#time-complexity)

The time complexity of the above algorithm is O(N)*O*(*N*), where ‘N’ is the total number of nodes in the tree. This is due to the fact that we traverse each node once.

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/xV7E64m4lnz#space-complexity)

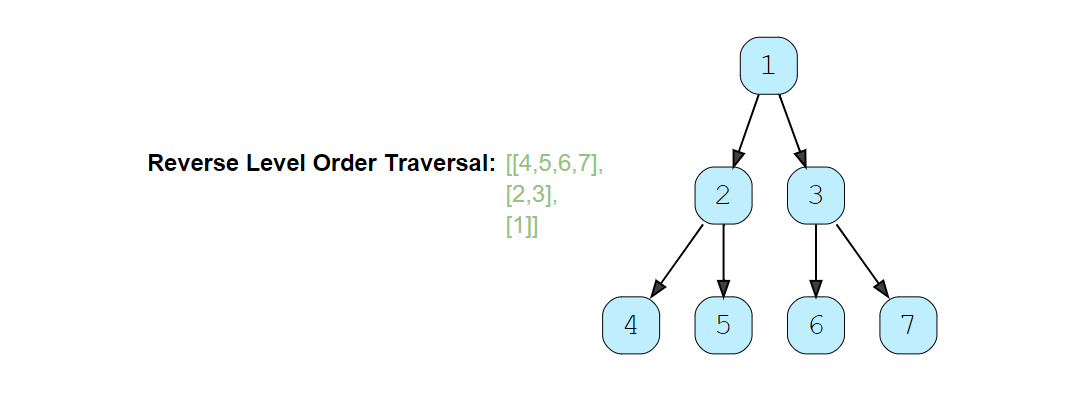
The space complexity of the above algorithm will be O(N)*O*(*N*) as we need to return a list containing the level order traversal. We will also need O(N)*O*(*N*) space for the queue. Since we can have a maximum of N/2*N*/2 nodes at any level (this could happen only at the lowest level), therefore we will need O(N)*O*(*N*) space to store them in the queue.

# Reverse Level Order Traversal

### Problem Statement 4 [#](https://www.educative.io/courses/grokking-the-coding-interview/m2N6GwARL8r#problem-statement)

Given a binary tree, populate an array to represent its level-by-level traversal in reverse order, i.e., the **lowest level comes first**. You should populate the values of all nodes in each level from left to right in separate sub-arrays.

**Example 1:**



**Example 2:**



### Try

import java.util.\*;

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode(int x) {

    val = x;

  }

};

class ReverseLevelOrderTraversal {

  public static List<List<Integer>> traverse(TreeNode root) {

    List<List<Integer>> result = new LinkedList<List<Integer>>();

    // TODO: Write your code here

    return result;

  }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(12);

    root.left = new TreeNode(7);

    root.right = new TreeNode(1);

    root.left.left = new TreeNode(9);

    root.right.left = new TreeNode(10);

    root.right.right = new TreeNode(5);

    List<List<Integer>> result = ReverseLevelOrderTraversal.traverse(root);

    System.out.println("Reverse level order traversal: " + result);

  }

}

### Solution [#](https://www.educative.io/courses/grokking-the-coding-interview/m2N6GwARL8r#solution)

This problem follows the [Binary Tree Level Order Traversal](https://www.educative.io/collection/page/5668639101419520/5671464854355968/5726607939469312/) pattern. We can follow the same **BFS** approach. The only difference will be that instead of appending the current level at the end, we will append the current level at the beginning of the result list.

### Code [#](https://www.educative.io/courses/grokking-the-coding-interview/m2N6GwARL8r#code)

Here is what our algorithm will look like; only the highlighted lines have changed. Please note that, for **Java**, we will use a LinkedList instead of an ArrayList for our result list. As in the case of ArrayList, appending an element at the beginning means shifting all the existing elements. Since we need to append the level array at the beginning of the result list, a LinkedList will be better, as this shifting of elements is not required in a LinkedList.

import java.util.\*;

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode(int x) {

    val = x;

  }

};

class ReverseLevelOrderTraversal {

  public static List<List<Integer>> traverse(TreeNode root) {

    List<List<Integer>> result = new LinkedList<List<Integer>>();

    if (root == null)

      return result;

    Queue<TreeNode> queue = new LinkedList<>();

    queue.offer(root);

    while (!queue.isEmpty()) {

      int levelSize = queue.size();

      List<Integer> currentLevel = new ArrayList<>(levelSize);

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

        TreeNode currentNode = queue.poll();

        // add the node to the current level

        currentLevel.add(currentNode.val);

        // insert the children of current node to the queue

        if (currentNode.left != null)

          queue.offer(currentNode.left);

        if (currentNode.right != null)

          queue.offer(currentNode.right);

      }

      // append the current level at the beginning

      result.add(0, currentLevel);

    }

    return result;

  }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(12);

    root.left = new TreeNode(7);

    root.right = new TreeNode(1);

    root.left.left = new TreeNode(9);

    root.right.left = new TreeNode(10);

    root.right.right = new TreeNode(5);

    List<List<Integer>> result = ReverseLevelOrderTraversal.traverse(root);

    System.out.println("Reverse level order traversal: " + result);

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/m2N6GwARL8r#time-complexity)

The time complexity of the above algorithm is O(N)*O*(*N*), where ‘N’ is the total number of nodes in the tree. This is due to the fact that we traverse each node once.

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/m2N6GwARL8r#space-complexity)

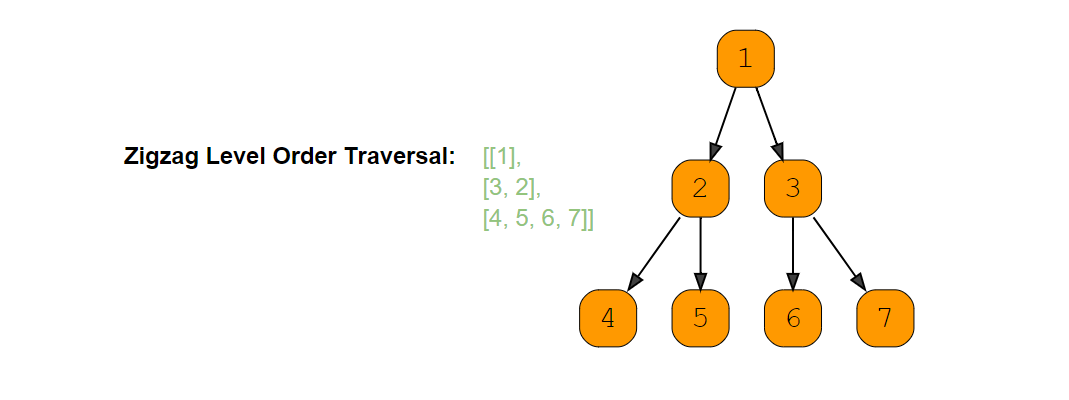
The space complexity of the above algorithm will be O(N)*O*(*N*) as we need to return a list containing the level order traversal. We will also need O(N)*O*(*N*) space for the queue. Since we can have a maximum of N/2*N*/2 nodes at any level (this could happen only at the lowest level), therefore we will need O(N)*O*(*N*) space to store them in the queue.

# Zigzag Traversal

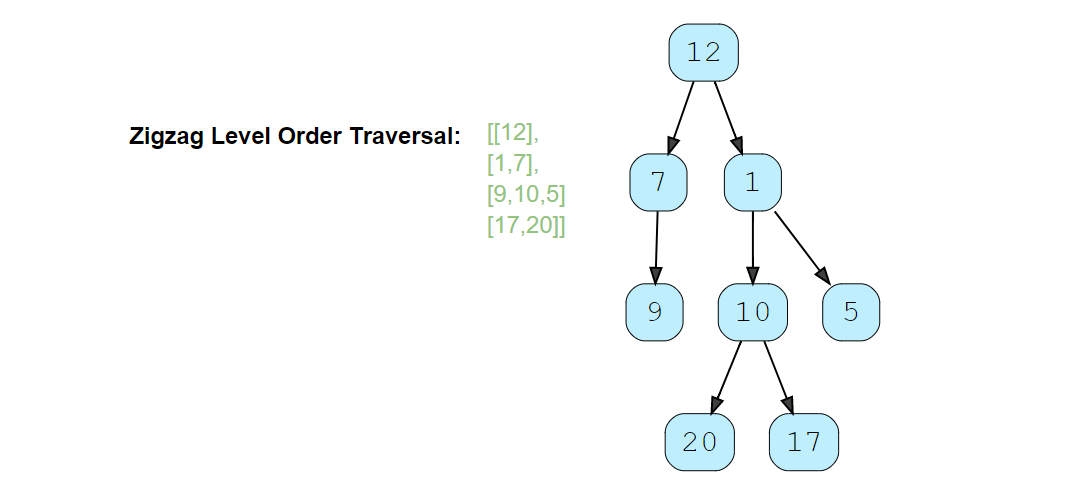
### Problem Statement 5 [#](https://www.educative.io/courses/grokking-the-coding-interview/qVA27MMYYn0#problem-statement)

Given a binary tree, populate an array to represent its zigzag level order traversal. You should populate the values of all **nodes of the first level from left to right**, then **right to left for the next level** and keep alternating in the same manner for the following levels.

**Example 1:**



**Example 2:**



### Try

import java.util.\*;

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode(int x) {

    val = x;

  }

};

class ZigzagTraversal {

  public static List<List<Integer>> traverse(TreeNode root) {

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

    // TODO: Write your code here

    return result;

  }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(12);

    root.left = new TreeNode(7);

    root.right = new TreeNode(1);

    root.left.left = new TreeNode(9);

    root.right.left = new TreeNode(10);

    root.right.right = new TreeNode(5);

    root.right.left.left = new TreeNode(20);

    root.right.left.right = new TreeNode(17);

    List<List<Integer>> result = ZigzagTraversal.traverse(root);

    System.out.println("Zigzag traversal: " + result);

  }

}

### Solution [#](https://www.educative.io/courses/grokking-the-coding-interview/qVA27MMYYn0#solution)

This problem follows the Binary Tree Level order Traversal pattern. We can follow the same **BFS** approach. The only additional step we have to keep in mind is to alternate the level order traversal, which means that for every other level, we will traverse similar to [Reverse Level Order Traversal](https://www.educative.io/collection/page/5668639101419520/5671464854355968/5765606242516992/).

### Code [#](https://www.educative.io/courses/grokking-the-coding-interview/qVA27MMYYn0#code)

Here is what our algorithm will look like, only the highlighted lines have changed:

import java.util.\*;

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode(int x) {

    val = x;

  }

};

class ZigzagTraversal {

  public static List<List<Integer>> traverse(TreeNode root) {

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

    if (root == null)

      return result;

    Queue<TreeNode> queue = new LinkedList<>();

    queue.offer(root);

    boolean leftToRight = true;

    while (!queue.isEmpty()) {

      int levelSize = queue.size();

      List<Integer> currentLevel = new LinkedList<>();

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

        TreeNode currentNode = queue.poll();

        // add the node to the current level based on the traverse direction

        if (leftToRight)

          currentLevel.add(currentNode.val);

        else

          currentLevel.add(0, currentNode.val);

        // insert the children of current node in the queue

        if (currentNode.left != null)

          queue.offer(currentNode.left);

        if (currentNode.right != null)

          queue.offer(currentNode.right);

      }

      result.add(currentLevel);

      // reverse the traversal direction

      leftToRight = !leftToRight;

    }

    return result;

  }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(12);

    root.left = new TreeNode(7);

    root.right = new TreeNode(1);

    root.left.left = new TreeNode(9);

    root.right.left = new TreeNode(10);

    root.right.right = new TreeNode(5);

    root.right.left.left = new TreeNode(20);

    root.right.left.right = new TreeNode(17);

    List<List<Integer>> result = ZigzagTraversal.traverse(root);

    System.out.println("Zigzag traversal: " + result);

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/qVA27MMYYn0#time-complexity)

The time complexity of the above algorithm is O(N)*O*(*N*), where ‘N’ is the total number of nodes in the tree. This is due to the fact that we traverse each node once.

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/qVA27MMYYn0#space-complexity)

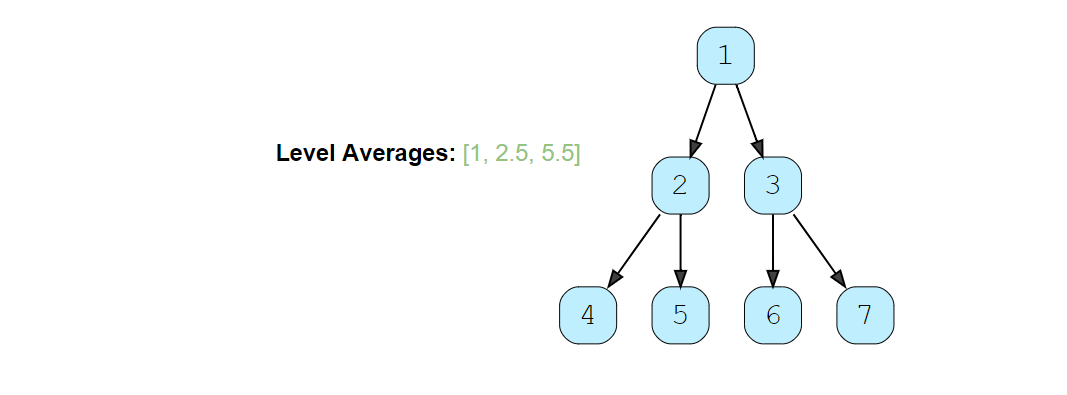
The space complexity of the above algorithm will be O(N)*O*(*N*) as we need to return a list containing the level order traversal. We will also need O(N)*O*(*N*) space for the queue. Since we can have a maximum of N/2*N*/2 nodes at any level (this could happen only at the lowest level), therefore we will need O(N)*O*(*N*) space to store them in the queue.

# Level Averages in a Binary Tree

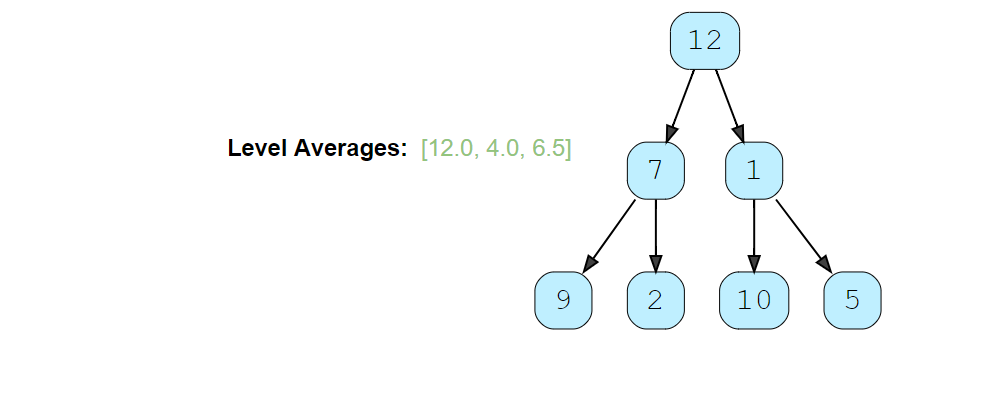
### Problem Statement 6 [#](https://www.educative.io/courses/grokking-the-coding-interview/YQWkA2l67GW#problem-statement)

Given a binary tree, populate an array to represent the **averages of all of its levels**.

**Example 1:**



**Example 2:**



### Try

import java.util.\*;

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode(int x) {

    val = x;

  }

};

class LevelAverage {

  public static List<Double> findLevelAverages(TreeNode root) {

    List<Double> result = new ArrayList<>();

    // TODO: Write your code here

    return result;

  }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(12);

    root.left = new TreeNode(7);

    root.right = new TreeNode(1);

    root.left.left = new TreeNode(9);

    root.left.right = new TreeNode(2);

    root.right.left = new TreeNode(10);

    root.right.right = new TreeNode(5);

    List<Double> result = LevelAverage.findLevelAverages(root);

    System.out.print("Level averages are: " + result);

  }

}

### Solution [#](https://www.educative.io/courses/grokking-the-coding-interview/YQWkA2l67GW#solution)

This problem follows the Binary Tree Level Order traversal pattern. We can follow the same **BFS** approach. The only difference will be that instead of keeping track of all nodes of a level, we will only track the running sum of the values of all nodes in each level. In the end, we will append the average of the current level to the result array.

### Code [#](https://www.educative.io/courses/grokking-the-coding-interview/YQWkA2l67GW#code)

import java.util.\*;

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode(int x) {

    val = x;

  }

};

class LevelAverage {

  public static List<Double> findLevelAverages(TreeNode root) {

    List<Double> result = new ArrayList<>();

    if (root == null)

      return result;

    Queue<TreeNode> queue = new LinkedList<>();

    queue.offer(root);

    while (!queue.isEmpty()) {

      int levelSize = queue.size();

      double levelSum = 0;

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

        TreeNode currentNode = queue.poll();

        // add the node's value to the running sum

        levelSum += currentNode.val;

        // insert the children of current node to the queue

        if (currentNode.left != null)

          queue.offer(currentNode.left);

        if (currentNode.right != null)

          queue.offer(currentNode.right);

      }

      // append the current level's average to the result array

      result.add(levelSum / levelSize);

    }

    return result;

  }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(12);

    root.left = new TreeNode(7);

    root.right = new TreeNode(1);

    root.left.left = new TreeNode(9);

    root.left.right = new TreeNode(2);

    root.right.left = new TreeNode(10);

    root.right.right = new TreeNode(5);

    List<Double> result = LevelAverage.findLevelAverages(root);

    System.out.print("Level averages are: " + result);

  }

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/YQWkA2l67GW#time-complexity)

The time complexity of the above algorithm is O(N)*O*(*N*), where ‘N’ is the total number of nodes in the tree. This is due to the fact that we traverse each node once.

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/YQWkA2l67GW#space-complexity)

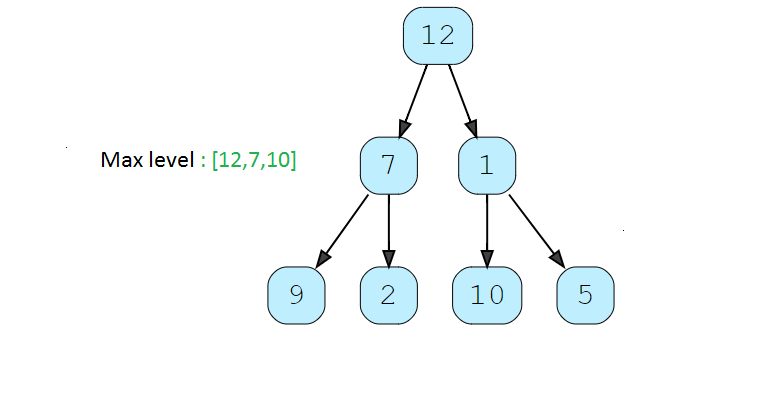
The space complexity of the above algorithm will be *O*(*N*) which is required for the queue. Since we can have a maximum of *N*/2 nodes at any level (this could happen only at the lowest level), therefore we will need *O*(*N*) space to store them in the queue.

# Largest value on each level

### Problem Statement 7 [#](https://www.educative.io/courses/grokking-the-coding-interview/YQWkA2l67GW#problem-statement)

Given a binary tree, populate an array to represent the **largest value of each of its levels**.

**Example 1:**



### Try

import java.util.\*;

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode(int x) {

    val = x;

  }

};

class LevelMax {

  public static List<Integer> findLevelMax(TreeNode root) {

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

    // TODO: Write your code here

    return result;

  }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(12);

    root.left = new TreeNode(7);

    root.right = new TreeNode(1);

    root.left.left = new TreeNode(9);

    root.left.right = new TreeNode(2);

    root.right.left = new TreeNode(10);

    root.right.right = new TreeNode(5);

    List<Integer> result = LevelAverage.findLevelMax(root);

    System.out.print("Level averages are: " + result);

  }

}

### Solution [#](https://www.educative.io/courses/grokking-the-coding-interview/YQWkA2l67GW#solution)

This problem follows the Binary Tree Level Order traversal pattern. We can follow the same **BFS** approach. The only difference will be that instead of keeping track of all nodes of a level, we will only track the max value of the values of all nodes in each level. In the end, we will append the max of the current level to the result array.

### Code [#](https://www.educative.io/courses/grokking-the-coding-interview/YQWkA2l67GW#code)

import java.util.\*;

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode(int x) {

    val = x;

  }

};

class LevelMax {

  public static List<Integer> findLevelMax(TreeNode root) {

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

    if (root == null)

      return result;

    Queue<TreeNode> queue = new LinkedList<>();

    queue.offer(root);

    while (!queue.isEmpty()) {

      int levelSize = queue.size();

      int levelMax = Integer.MIN\_VALUE;

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

        TreeNode currentNode = queue.poll();

        // add the node's value to the running sum

        levelMax = Math.max(levelMax,currentNode.val);

        // insert the children of current node to the queue

        if (currentNode.left != null)

          queue.offer(currentNode.left);

        if (currentNode.right != null)

          queue.offer(currentNode.right);

      }

      // append the current level's average to the result array

      result.add(levelMax);

    }

    return result;

  }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(12);

    root.left = new TreeNode(7);

    root.right = new TreeNode(1);

    root.left.left = new TreeNode(9);

    root.left.right = new TreeNode(2);

    root.right.left = new TreeNode(10);

    root.right.right = new TreeNode(5);

    List<Integer> result = LevelMax.findLevelMax(root);

    System.out.print("Level averages are: " + result);

  }

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/YQWkA2l67GW#time-complexity)

The time complexity of the above algorithm is O(N)*O*(*N*), where ‘N’ is the total number of nodes in the tree. This is due to the fact that we traverse each node once.

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/YQWkA2l67GW#space-complexity)

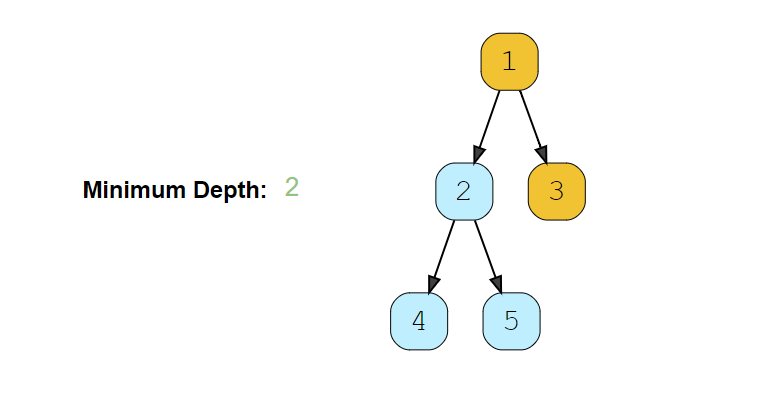
The space complexity of the above algorithm will be *O*(*N*) which is required for the queue. Since we can have a maximum of *N*/2 nodes at any level (this could happen only at the lowest level), therefore we will need *O*(*N*) space to store them in the queue.

# Minimum Depth of a Binary Tree

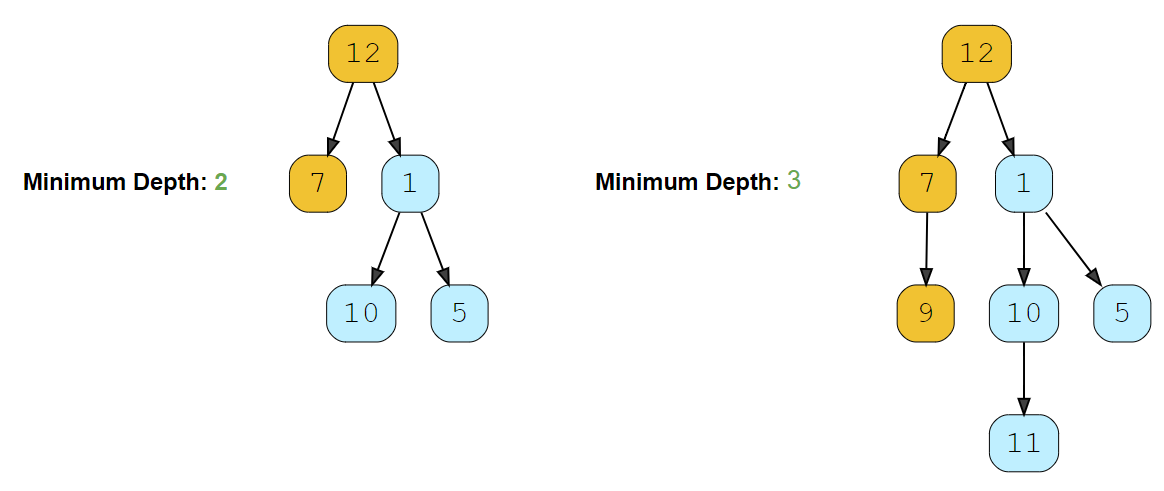
### Problem Statement 7 [#](https://www.educative.io/courses/grokking-the-coding-interview/3jwVx84OMkO#problem-statement)

Find the minimum depth of a binary tree. The minimum depth is the number of nodes along the **shortest path from the root node to the nearest leaf node**.

**Example 1:**



**Example 2:**



### Try

import java.util.\*;

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode(int x) {

    val = x;

  }

};

class MinimumBinaryTreeDepth {

  public static int findDepth(TreeNode root) {

    // TODO: Write your code here

    return -1;

  }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(12);

    root.left = new TreeNode(7);

    root.right = new TreeNode(1);

    root.right.left = new TreeNode(10);

    root.right.right = new TreeNode(5);

    System.out.println("Tree Minimum Depth: " + MinimumBinaryTreeDepth.findDepth(root));

    root.left.left = new TreeNode(9);

    root.right.left.left = new TreeNode(11);

    System.out.println("Tree Minimum Depth: " + MinimumBinaryTreeDepth.findDepth(root));

  }

}

### Solution [#](https://www.educative.io/courses/grokking-the-coding-interview/3jwVx84OMkO#solution)

This problem follows the Binary Tree Level Order Traversal pattern. We can follow the same **BFS** approach. The only difference will be, instead of keeping track of all the nodes in a level, we will only track the depth of the tree. As soon as we find our first leaf node, that level will represent the minimum depth of the tree.

### Code [#](https://www.educative.io/courses/grokking-the-coding-interview/3jwVx84OMkO#code)

import java.util.\*;

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode(int x) {

    val = x;

  }

};

class MinimumBinaryTreeDepth {

  public static int findDepth(TreeNode root) {

    if (root == null)

      return 0;

    Queue<TreeNode> queue = new LinkedList<>();

    queue.add(root);

    int minimumTreeDepth = 0;

    while (!queue.isEmpty()) {

      minimumTreeDepth++;

      int levelSize = queue.size();

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

        TreeNode currentNode = queue.poll();

        // check if this is a leaf node

        if (currentNode.left == null && currentNode.right == null)

          return minimumTreeDepth;

        // insert the children of current node in the queue

        if (currentNode.left != null)

          queue.add(currentNode.left);

        if (currentNode.right != null)

          queue.add(currentNode.right);

      }

    }

    return minimumTreeDepth;

  }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(12);

    root.left = new TreeNode(7);

    root.right = new TreeNode(1);

    root.right.left = new TreeNode(10);

    root.right.right = new TreeNode(5);

    System.out.println("Tree Minimum Depth: " + MinimumBinaryTreeDepth.findDepth(root));

    root.left.left = new TreeNode(9);

    root.right.left.left = new TreeNode(11);

    System.out.println("Tree Minimum Depth: " + MinimumBinaryTreeDepth.findDepth(root));

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/3jwVx84OMkO#time-complexity)

The time complexity of the above algorithm is O(N)*O*(*N*), where ‘N’ is the total number of nodes in the tree. This is due to the fact that we traverse each node once.

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/3jwVx84OMkO#space-complexity)

The space complexity of the above algorithm will be O(N)*O*(*N*) which is required for the queue. Since we can have a maximum of N/2*N*/2 nodes at any level (this could happen only at the lowest level), therefore we will need O(N)*O*(*N*) space to store them in the queue.

# Minimum Depth of a Binary Tree

### Problem Statement 8 [#](https://www.educative.io/courses/grokking-the-coding-interview/3jwVx84OMkO#problem-statement)

Find the maximum depth of a binary tree (height).

### Solution [#](https://www.educative.io/courses/grokking-the-coding-interview/3jwVx84OMkO#solution)

This problem follows the Binary Tree Level Order Traversal pattern. We can follow the same **BFS** approach. The only difference will be, instead of keeping track of all the nodes in a level, we will only track the depth of the tree. As soon as we find our first leaf node, that level will represent the minimum depth of the tree.

### Code [#](https://www.educative.io/courses/grokking-the-coding-interview/3jwVx84OMkO#code)

import java.util.\*;

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode(int x) {

    val = x;

  }

};

class MaximumBinaryTreeDepth {

  public static int findDepth(TreeNode root) {

    if (root == null)

      return 0;

    Queue<TreeNode> queue = new LinkedList<>();

    queue.add(root);

    int maximumTreeDepth = 0;

    while (!queue.isEmpty()) {

      maximumTreeDepth++;

      int levelSize = queue.size();

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

        TreeNode currentNode = queue.poll();

        // insert the children of current node in the queue

        if (currentNode.left != null)

          queue.add(currentNode.left);

        if (currentNode.right != null)

          queue.add(currentNode.right);

      }

    }

    return maximumTreeDepth;

  }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(12);

    root.left = new TreeNode(7);

    root.right = new TreeNode(1);

    root.right.left = new TreeNode(10);

    root.right.right = new TreeNode(5);

    System.out.println("Tree Maximum Depth: " + MaximumBinaryTreeDepth.findDepth(root));

    root.left.left = new TreeNode(9);

    root.right.left.left = new TreeNode(11);

    System.out.println("Tree Maximum Depth: " + MaximumBinaryTreeDepth.findDepth(root));

  }

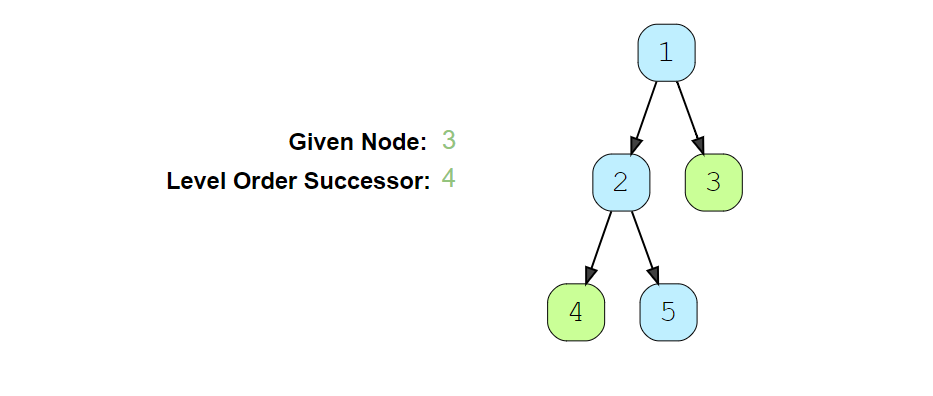
}

# Level Order Successor

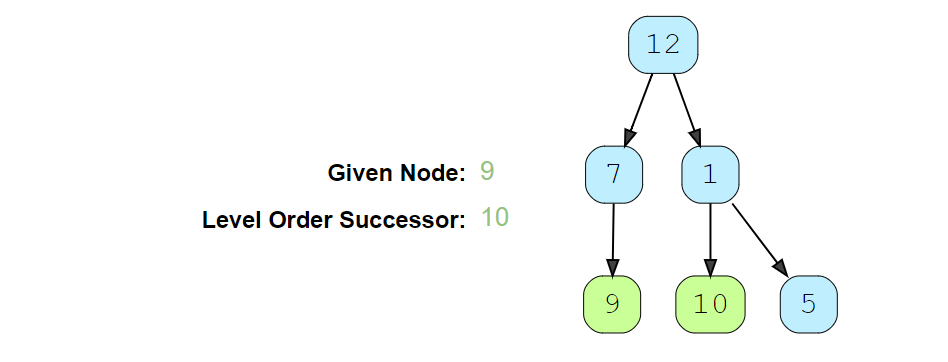
### Problem Statement 9 [#](https://www.educative.io/courses/grokking-the-coding-interview/7nO4VmA74Lr#problem-statement)

Given a binary tree and a node, find the level order successor of the given node in the tree. The level order successor is the node that appears right after the given node in the level order traversal.

**Example 1:**



**Example 2:**



### Try

import java.util.\*;

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode(int x) {

    val = x;

  }

};

class LevelOrderSuccessor {

  public static TreeNode findSuccessor(TreeNode root, int key) {

    // TODO: Write your code here

    return null;

  }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(12);

    root.left = new TreeNode(7);

    root.right = new TreeNode(1);

    root.left.left = new TreeNode(9);

    root.right.left = new TreeNode(10);

    root.right.right = new TreeNode(5);

    TreeNode result = LevelOrderSuccessor.findSuccessor(root, 12);

    if (result != null)

      System.out.println(result.val + " ");

    result = LevelOrderSuccessor.findSuccessor(root, 9);

    if (result != null)

      System.out.println(result.val + " ");

  }

}

### Solution [#](https://www.educative.io/courses/grokking-the-coding-interview/7nO4VmA74Lr#solution)

This problem follows the Binary Tree Level Order Trversal pattern. We can follow the same **BFS** approach. The only difference will be that we will not keep track of all the levels. Instead we will keep inserting child nodes to the queue. As soon as we find the given node, we will return the next node from the queue as the level order successor.

### Code [#](https://www.educative.io/courses/grokking-the-coding-interview/7nO4VmA74Lr#code)

import java.util.\*;

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode(int x) {

    val = x;

  }

};

class LevelOrderSuccessor {

  public static TreeNode findSuccessor(TreeNode root, int key) {

    if (root == null)

      return null;

    Queue<TreeNode> queue = new LinkedList<>();

    queue.offer(root);

    while (!queue.isEmpty()) {

      TreeNode currentNode = queue.poll();

      // insert the children of current node in the queue

      if (currentNode.left != null)

        queue.offer(currentNode.left);

      if (currentNode.right != null)

        queue.offer(currentNode.right);

      // break if we have found the key

      if (currentNode.val == key)

        break;

    }

    return queue.peek();

  }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(12);

    root.left = new TreeNode(7);

    root.right = new TreeNode(1);

    root.left.left = new TreeNode(9);

    root.right.left = new TreeNode(10);

    root.right.right = new TreeNode(5);

    TreeNode result = LevelOrderSuccessor.findSuccessor(root, 12);

    if (result != null)

      System.out.println(result.val + " ");

    result = LevelOrderSuccessor.findSuccessor(root, 9);

    if (result != null)

      System.out.println(result.val + " ");

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/7nO4VmA74Lr#time-complexity)

The time complexity of the above algorithm is *O*(*N*), where ‘N’ is the total number of nodes in the tree. This is due to the fact that we traverse each node once.

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/7nO4VmA74Lr#space-complexity)

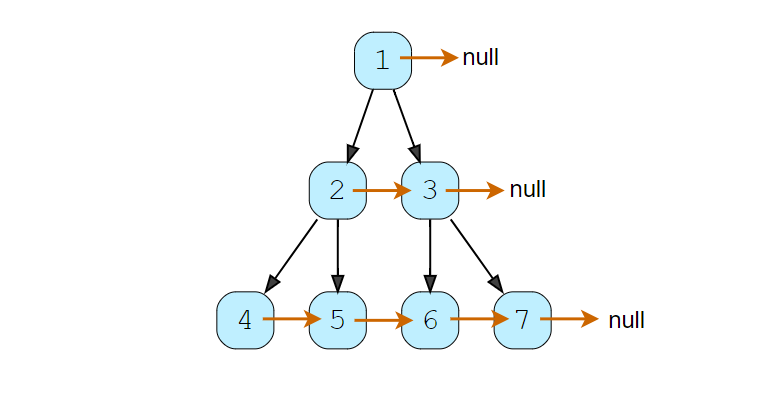
The space complexity of the above algorithm will be *O*(*N*) which is required for the queue. Since we can have a maximum of *N*/2 nodes at any level (this could happen only at the lowest level), therefore we will need *O*(*N*) space to store them in the queue.

# Connect Level Order Siblings

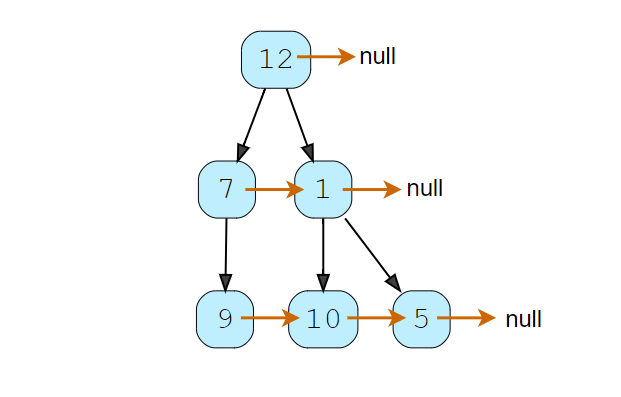
### Problem Statement 10 [#](https://www.educative.io/courses/grokking-the-coding-interview/m2YYxXDOJ03#problem-statement)

Given a binary tree, connect each node with its level order successor. The last node of each level should point to a null node.

**Example 1:**



**Example 2:**



### Try

import java.util.\*;

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode next;

  TreeNode(int x) {

    val = x;

    left = right = next = null;

  }

  // level order traversal using 'next' pointer

  void printLevelOrder() {

    TreeNode nextLevelRoot = this;

    while (nextLevelRoot != null) {

      TreeNode current = nextLevelRoot;

      nextLevelRoot = null;

      while (current != null) {

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

        if (nextLevelRoot == null) {

          if (current.left != null)

            nextLevelRoot = current.left;

          else if (current.right != null)

            nextLevelRoot = current.right;

        }

        current = current.next;

      }

      System.out.println();

    }

  }

};

class ConnectLevelOrderSiblings {

  public static void connect(TreeNode root) {

    // TODO: Write your code here

  }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(12);

    root.left = new TreeNode(7);

    root.right = new TreeNode(1);

    root.left.left = new TreeNode(9);

    root.right.left = new TreeNode(10);

    root.right.right = new TreeNode(5);

    ConnectLevelOrderSiblings.connect(root);

    System.out.println("Level order traversal using 'next' pointer: ");

    root.printLevelOrder();

  }

}

### Solution [#](https://www.educative.io/courses/grokking-the-coding-interview/m2YYxXDOJ03#solution)

This problem follows the Binary Tree Level Order Traversal pattern. We can follow the same **BFS** approach. The only difference is that while traversing a level we will remember the previous node to connect it with the current node.

### Code

import java.util.\*;

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode next;

  TreeNode(int x) {

    val = x;

    left = right = next = null;

  }

  // level order traversal using 'next' pointer

  public void printLevelOrder() {

    TreeNode nextLevelRoot = this;

    while (nextLevelRoot != null) {

      TreeNode current = nextLevelRoot;

      nextLevelRoot = null;

      while (current != null) {

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

        if (nextLevelRoot == null) {

          if (current.left != null)

            nextLevelRoot = current.left;

          else if (current.right != null)

            nextLevelRoot = current.right;

        }

        current = current.next;

      }

      System.out.println();

    }

  }

};

class ConnectLevelOrderSiblings {

  public static void connect(TreeNode root) {

    if (root == null)

      return;

    Queue<TreeNode> queue = new LinkedList<>();

    queue.offer(root);

    while (!queue.isEmpty()) {

      TreeNode previousNode = null;

      int levelSize = queue.size();

      // connect all nodes of this level

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

        TreeNode currentNode = queue.poll();

        if (previousNode != null)

          previousNode.next = currentNode;

        previousNode = currentNode;

        // insert the children of current node in the queue

        if (currentNode.left != null)

          queue.offer(currentNode.left);

        if (currentNode.right != null)

          queue.offer(currentNode.right);

      }

    }

  }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(12);

    root.left = new TreeNode(7);

    root.right = new TreeNode(1);

    root.left.left = new TreeNode(9);

    root.right.left = new TreeNode(10);

    root.right.right = new TreeNode(5);

    ConnectLevelOrderSiblings.connect(root);

    System.out.println("Level order traversal using 'next' pointer: ");

    root.printLevelOrder();

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/m2YYxXDOJ03#time-complexity)

The time complexity of the above algorithm is O(N)*O*(*N*), where ‘N’ is the total number of nodes in the tree. This is due to the fact that we traverse each node once.

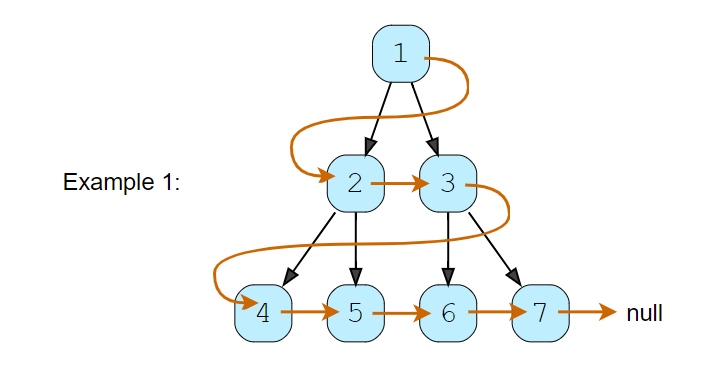
#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/m2YYxXDOJ03#space-complexity)

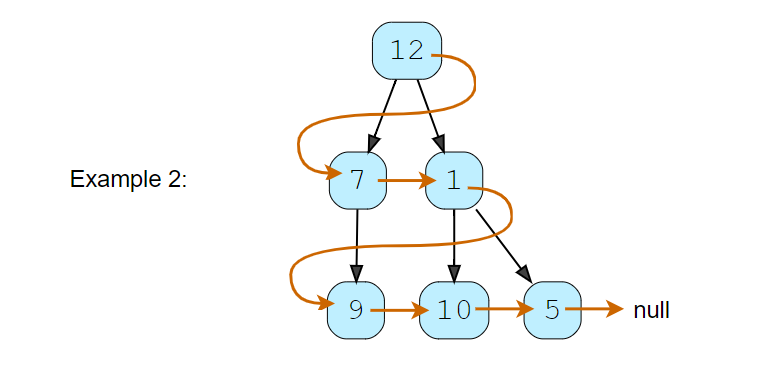
The space complexity of the above algorithm will be O(N)*O*(*N*), which is required for the queue. Since we can have a maximum of N/2*N*/2 nodes at any level (this could happen only at the lowest level), therefore we will need O(N)*O*(*N*) space to store them in the queue.

### Connect All Level Order Siblings

**Problem Statement 11**

Given a binary tree, connect each node with its level order successor. The last node of each level should point to the first node of the next level.





### Try

import java.util.\*;

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode next;

  TreeNode(int x) {

    val = x;

    left = right = next = null;

  }

};

class ConnectAllSiblings {

  public static void connect(TreeNode root) {

    // TODO: Write your code here

  }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(12);

    root.left = new TreeNode(7);

    root.right = new TreeNode(1);

    root.left.left = new TreeNode(9);

    root.right.left = new TreeNode(10);

    root.right.right = new TreeNode(5);

    ConnectAllSiblings.connect(root);

    // level order traversal using 'next' pointer

    TreeNode current = root;

    System.out.println("Traversal using 'next' pointer: ");

    while (current != null) {

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

      current = current.next;

    }

  }

}

### Solution [#](https://www.educative.io/courses/grokking-the-coding-interview/qVxy1qop77p#solution)

This problem follows the Binary Tree Level Traversal pattern. We can follow the same **BFS** approach. The only difference will be that while traversing we will remember (irrespective of the level) the previous node to connect it with the current node.

### Code

import java.util.\*;

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode next;

  TreeNode(int x) {

    val = x;

    left = right = next = null;

  }

  // tree traversal using 'next' pointer

  public void printTree() {

    TreeNode current = this;

    System.out.print("Traversal using 'next' pointer: ");

    while (current != null) {

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

      current = current.next;

    }

  }

};

class ConnectAllSiblings {

  public static void connect(TreeNode root) {

    if (root == null)

      return;

    Queue<TreeNode> queue = new LinkedList<>();

    queue.offer(root);

    TreeNode currentNode = null, previousNode = null;

    while (!queue.isEmpty()) {

      currentNode = queue.poll();

      if (previousNode != null)

        previousNode.next = currentNode;

      previousNode = currentNode;

      // insert the children of current node in the queue

      if (currentNode.left != null)

        queue.offer(currentNode.left);

      if (currentNode.right != null)

        queue.offer(currentNode.right);

    }

  }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(12);

    root.left = new TreeNode(7);

    root.right = new TreeNode(1);

    root.left.left = new TreeNode(9);

    root.right.left = new TreeNode(10);

    root.right.right = new TreeNode(5);

    ConnectAllSiblings.connect(root);

    root.printTree();

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/qVxy1qop77p#time-complexity)

The time complexity of the above algorithm is O(N)*O*(*N*), where ‘N’ is the total number of nodes in the tree. This is due to the fact that we traverse each node once.

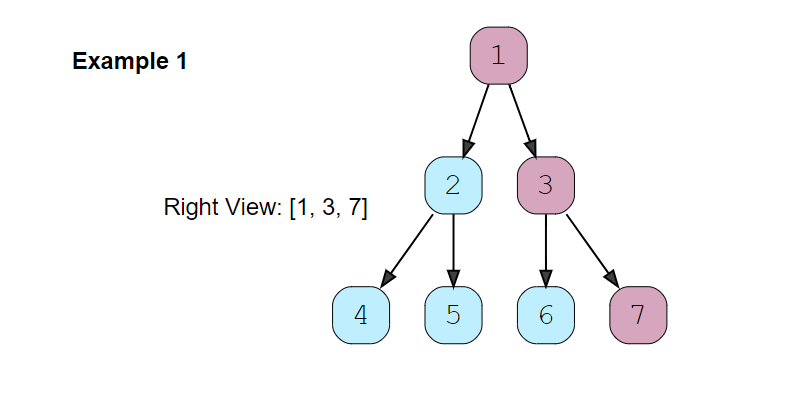
#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/qVxy1qop77p#space-complexity)

The space complexity of the above algorithm will be O(N)*O*(*N*) which is required for the queue. Since we can have a maximum of N/2*N*/2 nodes at any level (this could happen only at the lowest level), therefore we will need O(N)*O*(*N*) space to store them in the queue.

### Right View of a Binary Tree

**Problem Statement 12**

Given a binary tree, return an array containing nodes in its right view. The right view of a binary tree is the set of **nodes visible when the tree is seen from the right side**.





### Try

import java.util.\*;

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode(int x) {

    val = x;

  }

};

class RightViewTree {

  public static List<TreeNode> traverse(TreeNode root) {

    List<TreeNode> result = new ArrayList<>();

    // TODO: Write your code here

    return result;

  }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(12);

    root.left = new TreeNode(7);

    root.right = new TreeNode(1);

    root.left.left = new TreeNode(9);

    root.right.left = new TreeNode(10);

    root.right.right = new TreeNode(5);

    root.left.left.left = new TreeNode(3);

    List<TreeNode> result = RightViewTree.traverse(root);

    for (TreeNode node : result) {

      System.out.print(node.val + " ");

    }

  }

}

### Solution [#](https://www.educative.io/courses/grokking-the-coding-interview/gxVWnvZjMn9#solution)

This problem follows the Binary Tree Level Order traversal pattern. We can follow the same **BFS** approach. The only additional thing we will be do is to append the last node of each level to the result array.

### Code

import java.util.\*;

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode(int x) {

    val = x;

  }

};

class RightViewTree {

  public static List<TreeNode> traverse(TreeNode root) {

    List<TreeNode> result = new ArrayList<>();

    if (root == null)

      return result;

    Queue<TreeNode> queue = new LinkedList<>();

    queue.offer(root);

    while (!queue.isEmpty()) {

      int levelSize = queue.size();

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

        TreeNode currentNode = queue.poll();

        // if it is the last node of this level, add it to the result

        if (i == levelSize - 1)

          result.add(currentNode);

        // insert the children of current node in the queue

        if (currentNode.left != null)

          queue.offer(currentNode.left);

        if (currentNode.right != null)

          queue.offer(currentNode.right);

      }

    }

    return result;

  }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(12);

    root.left = new TreeNode(7);

    root.right = new TreeNode(1);

    root.left.left = new TreeNode(9);

    root.right.left = new TreeNode(10);

    root.right.right = new TreeNode(5);

    root.left.left.left = new TreeNode(3);

    List<TreeNode> result = RightViewTree.traverse(root);

    for (TreeNode node : result) {

      System.out.print(node.val + " ");

    }

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/gxVWnvZjMn9#time-complexity)

The time complexity of the above algorithm is O(N)*O*(*N*), where ‘N’ is the total number of nodes in the tree. This is due to the fact that we traverse each node once.

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/gxVWnvZjMn9#space-complexity)

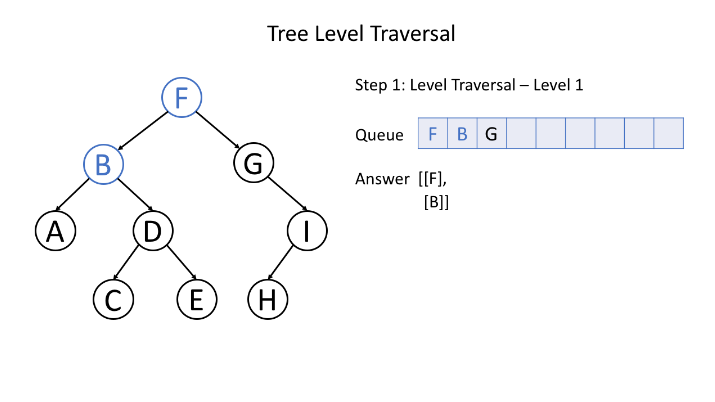
The space complexity of the above algorithm will be O(N)*O*(*N*) as we need to return a list containing the level order traversal. We will also need O(N)*O*(*N*) space for the queue. Since we can have a maximum of N/2*N*/2 nodes at any level (this could happen only at the lowest level), therefore we will need O(N)*O*(*N*) space to store them in the queue.

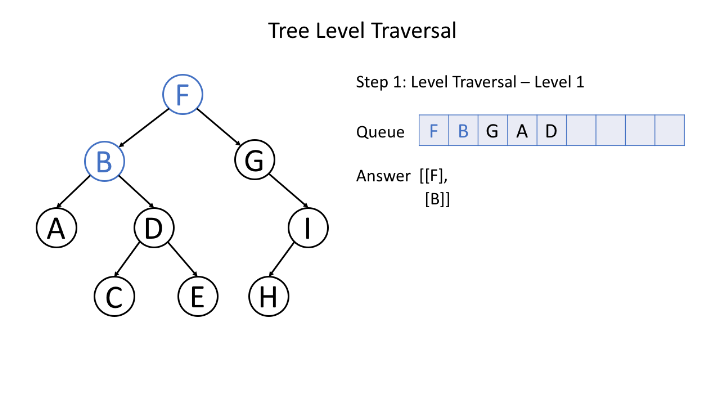
### Left View of a Binary Tree

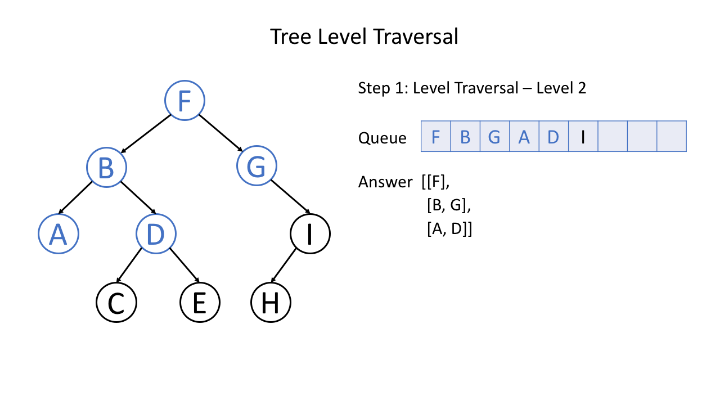
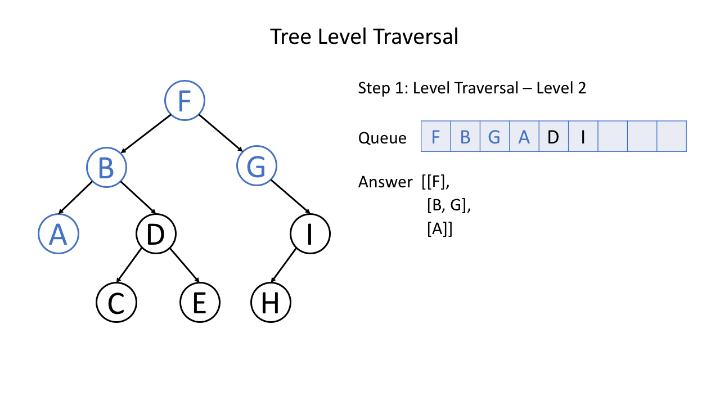
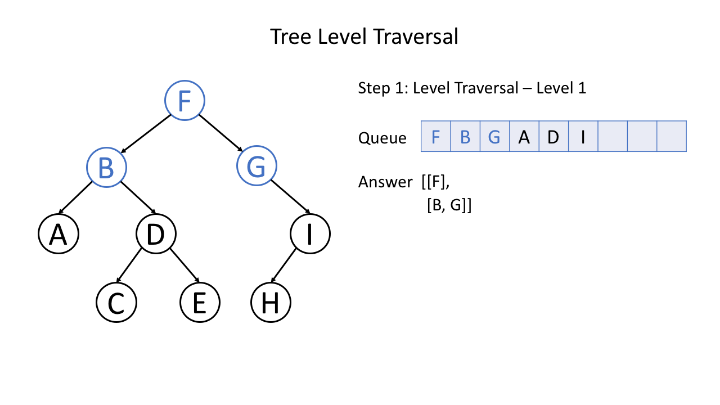
**Problem Statement 13**

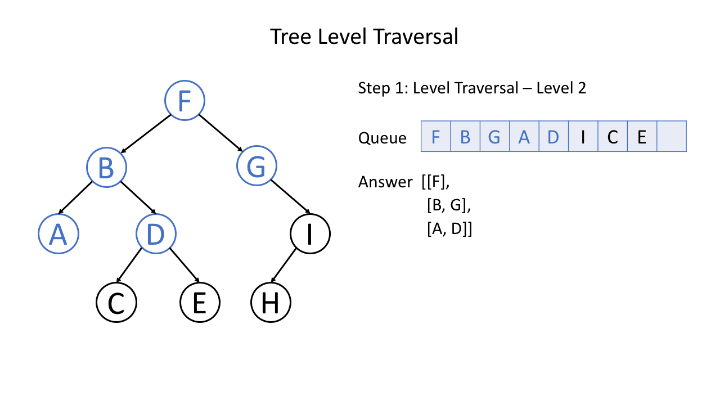
Given a binary tree, return an array containing nodes in its left view. The left view of a binary tree is the set of **nodes visible when the tree is seen from the left side**.

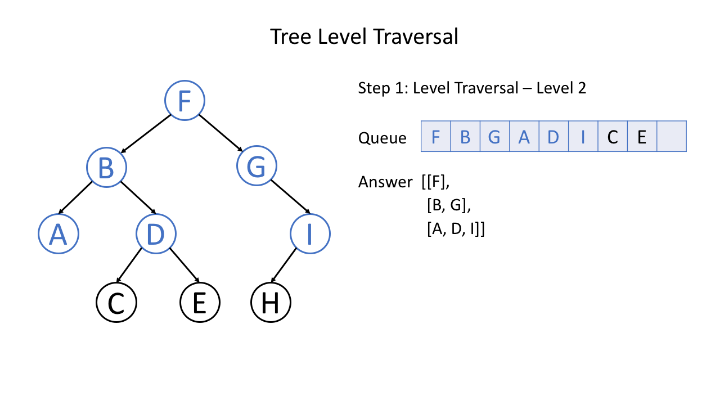
**Solution:** We will be following a similar approach, but instead of appending the last element of each level we will be appending the first element of each level to the output array.

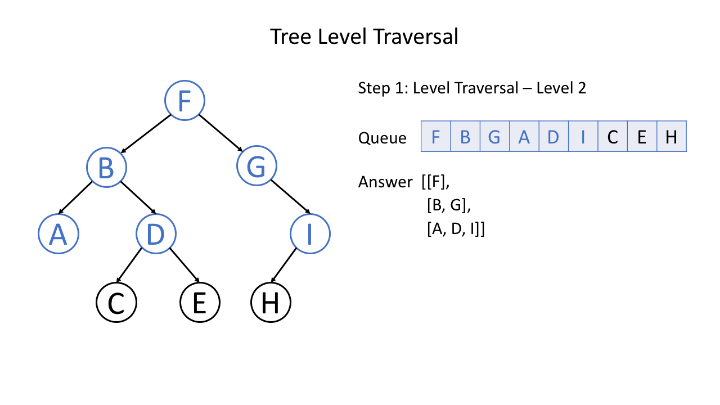




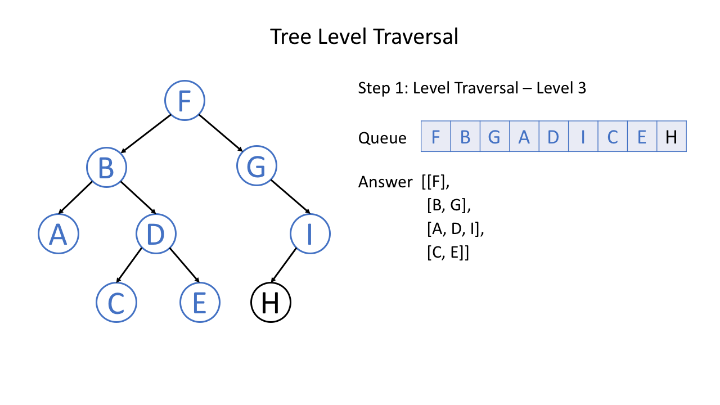


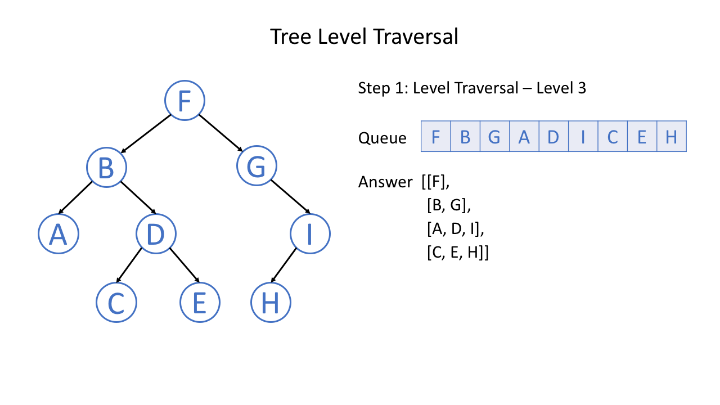


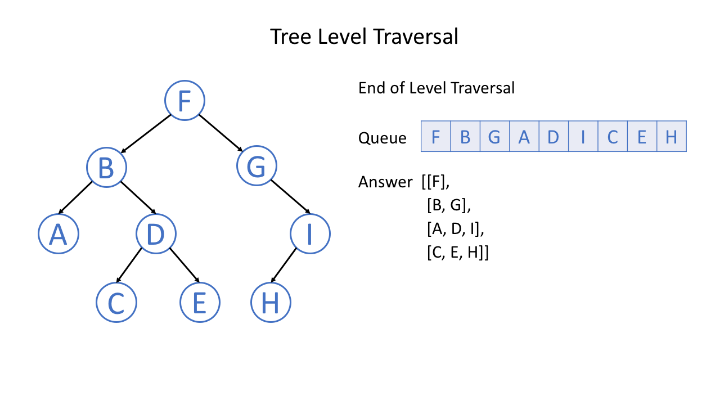












Typically, we use a queue to help us to do BFS.

We can also use BFS to traverse a graph. For example, we can use BFS to find a path, especially the shortest path, from a start node to a target node.

We can use BFS, in even more abstract scenarios, to traverse all the possible statuses. In this case, we can regard the statuses as the nodes in the graph while the legal transition paths as the edges in the graph.

**BFS – Template**

It will be important to determine the nodes and the edges before doing BFS in a specific question. Typically, the node will be an actual node or a status while the edge will be an actual edge or a possible transition.

*Template I*

Here we provide a pseudocode as a template:

/\*\*

 \* Return the length of the shortest path between root and target node.

 \*/

int BFS(Node root, Node target) {

    Queue<Node> queue;  // store all nodes which are waiting to be processed

    int step = 0;       // number of steps neeeded from root to current node

    // initialize

    add root to queue;

    // BFS

    while (queue is not empty) {

        step = step + 1;

        // iterate the nodes which are already in the queue

        int size = queue.size();

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

            Node cur = the first node in queue;

            return step if cur is target;

            for (Node next : the neighbors of cur) {

                add next to queue;

            }

      remove the first node from queue;

        }

    }

    return -1;          // there is no path from root to target

}

1. As shown in the code, in each round, the nodes in the queue are the nodes which are waiting to be processed.
2. After each outer while loop, we are one step farther from the root node. The variable step indicates the distance from the root node and the current node we are visiting.

### *Template II*

Sometimes, it is important to make sure that we never visit a node twice. Otherwise, we might get stuck in an infinite loop. If so, we can add a hash set to the code above to solve this problem. Here is the pseudocode after modification:

  /\*\*

 \* Return the length of the shortest path between root and target node.

 \*/

int BFS(Node root, Node target) {

    Queue<Node> queue;  // store all nodes which are waiting to be processed

    Set<Node> visited;  // store all the nodes that we've visited

    int step = 0;       // number of steps neeeded from root to current node

    // initialize

    add root to queue;

    add root to visited;

    // BFS

    while (queue is not empty) {

        step = step + 1;

        // iterate the nodes which are already in the queue

        int size = queue.size();

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

            Node cur = the first node in queue;

            return step if cur is target;

            for (Node next : the neighbors of cur) {

                if (next is not in used) {

                    add next to queue;

                    add next to visited;

                }

            }

      remove the first node from queue;

        }

    }

    return -1;          // there is no path from root to target

}

There are two cases you don't need the hash set used:

1. You are absolutely sure there is no cycle, for example, in tree traversal;
2. You do want to add the node to the queue multiple times.

Problems

### All Nodes Distance K in Binary Tree

**Problem Statement 14**

We are given a binary tree (with root node root), a target node, and an integer value K.

Return a list of the values of all nodes that have a distance K from the target node.  The answer can be returned in any order.

**Example 1:**

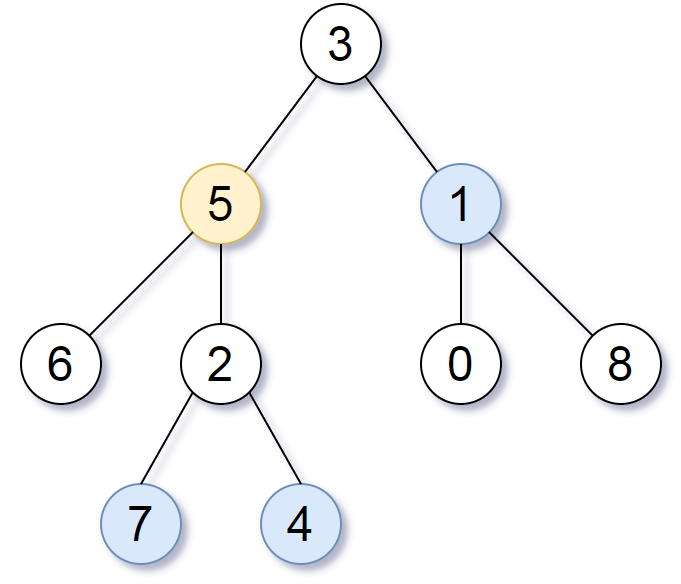
**Input:** root = [3,5,1,6,2,0,8,null,null,7,4], target = 5, K = 2

**Output:** [7,4,1]

**Explanation:**

The nodes that are a distance 2 from the target node (with value 5)

have values 7, 4, and 1.



Note that the inputs "root" and "target" are actually TreeNodes.

The descriptions of the inputs above are just serializations of these objects.

 /\*\*

 \* Definition for a binary tree node.

 \* public class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode(int x) { val = x; }

 \* }

 \*/

class Solution {

    public List<Integer> distanceK(TreeNode root, TreeNode target, int K) {

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

        if (root == null)

            return result;

        if (K == 0) {

            result.add(target.val);

            return result;

        }

        HashSet<TreeNode> seen = new HashSet<>();

        TreeNode parent = getParent(root,target);

        Queue<TreeNode> queue = new LinkedList<>();

        if (parent != null) {

            queue.offer(parent);

            seen.add(parent);

        }

        if (target.left != null) {

            queue.offer(target.left);

            seen.add(target.left);

        }

        if (target.right != null) {

            queue.offer(target.right);

            seen.add(target.right);

        }

        seen.add(target);

        int level = 0;

        while(!queue.isEmpty()) {

            level++;

            int size = queue.size();

            if (level == K) {

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

                    result.add(queue.poll().val);

                }

                return result;

            }

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

                TreeNode tempNode = queue.poll();

                TreeNode tempParent = getParent(root,tempNode);

                if (tempParent != null && !seen.contains(tempParent)) {

                    queue.offer(tempParent);

                    seen.add(tempParent);

                }

                if (tempNode.left != null && !seen.contains(tempNode.left)) {

                    queue.offer(tempNode.left);

                    seen.add(tempNode.left);

                }

                if ( tempNode.right != null && !seen.contains(tempNode.right)) {

                    queue.offer(tempNode.right);

                    seen.add(tempNode.right);

                }

            }

        }

        return result;

    }

    public TreeNode getParent(TreeNode root,TreeNode target) {

        if (root == null)

            return null;

        if (root == target)

            return null;

        if (root.left == target || root.right == target)

            return root;

        TreeNode leftNode = getParent(root.left,target);

        if (leftNode != null)

            return leftNode;

        TreeNode rightNode = getParent(root.right,target);

        return rightNode;

    }

}

More simply,

class Solution {

    Map<TreeNode, TreeNode> parent;

    public List<Integer> distanceK(TreeNode root, TreeNode target, int K) {

        parent = new HashMap();

        dfs(root, null);

        Queue<TreeNode> queue = new LinkedList();

        queue.add(null);

        queue.add(target);

        Set<TreeNode> seen = new HashSet();

        seen.add(target);

        seen.add(null);

        int dist = 0;

        while (!queue.isEmpty()) {

            TreeNode node = queue.poll();

            if (node == null) {

                if (dist == K) {

                    List<Integer> ans = new ArrayList();

                    for (TreeNode n: queue)

                        ans.add(n.val);

                    return ans;

                }

                queue.offer(null);

                dist++;

            } else {

                if (!seen.contains(node.left)) {

                    seen.add(node.left);

                    queue.offer(node.left);

                }

                if (!seen.contains(node.right)) {

                    seen.add(node.right);

                    queue.offer(node.right);

                }

                TreeNode par = parent.get(node);

                if (!seen.contains(par)) {

                    seen.add(par);

                    queue.offer(par);

                }

            }

        }

        return new ArrayList<Integer>();

    }

    public void dfs(TreeNode node, TreeNode par) {

        if (node != null) {

            parent.put(node, par);

            dfs(node.left, node);

            dfs(node.right, node);

        }

    }

}

### Open the Lock

**Problem Statement 15**

ou have a lock in front of you with 4 circular wheels. Each wheel has 10 slots: '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'. The wheels can rotate freely and wrap around: for example we can turn '9' to be '0', or '0' to be '9'. Each move consists of turning one wheel one slot.

The lock initially starts at '0000', a string representing the state of the 4 wheels.

You are given a list of deadends dead ends, meaning if the lock displays any of these codes, the wheels of the lock will stop turning and you will be unable to open it.

Given a target representing the value of the wheels that will unlock the lock, return the minimum total number of turns required to open the lock, or -1 if it is impossible.

**Example 1:**

**Input:** deadends = ["0201","0101","0102","1212","2002"], target = "0202"

**Output:** 6

**Explanation:**

A sequence of valid moves would be "0000" -> "1000" -> "1100" -> "1200" -> "1201" -> "1202" -> "0202".

Note that a sequence like "0000" -> "0001" -> "0002" -> "0102" -> "0202" would be invalid,

because the wheels of the lock become stuck after the display becomes the dead end "0102".

**Example 2:**

**Input:** deadends = ["8888"], target = "0009"

**Output:** 1

**Explanation:**

We can turn the last wheel in reverse to move from "0000" -> "0009".

**Example 3:**

**Input:** deadends = ["8887","8889","8878","8898","8788","8988","7888","9888"], target = "8888"

**Output:** -1

**Explanation:**

We can't reach the target without getting stuck.

**Example 4:**

**Input:** deadends = ["0000"], target = "8888"

**Output:** -1

class Solution {

    public int openLock(String[] deadends, String target) {

        if (target == null || target.length() == 0)

            return -1;

        HashSet<String> visited = new HashSet<>();

        HashSet<String> deadLocks = new HashSet<>(Arrays.asList(deadends));

        if (deadLocks.contains("0000"))

            return -1;

        Queue<String> queue = new LinkedList<>();

        queue.add("0000");

        visited.add("0000");

        int steps = 0;

        while(!queue.isEmpty()) {

            int size = queue.size();

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

                String curr = queue.poll();

                if (curr.equals(target))

                    return steps;

                for (int j = 0; j < curr.length(); j++) {

                    int digit = curr.charAt(j) - '0';

                    //forward direction

                    int forward = (digit+1)%10;

                    String newState = curr.substring(0,j) + forward + curr.substring(j+1);

                    if (!visited.contains(newState) && !deadLocks.contains(newState)) {

                        queue.add(newState);

                        visited.add(newState);

                    }

                    //backward direction

                    int backward = (digit+9)%10;

                    String newState2 = curr.substring(0,j) + backward + curr.substring(j+1);

                    if (!visited.contains(newState2) && !deadLocks.contains(newState2)) {

                        queue.add(newState2);

                        visited.add(newState2);

                    }

                }

            }

            steps++;

        }

        return -1;

    }

}

More simply,

class Solution {

    public int openLock(String[] deadends, String target) {

        Set<String> dead = new HashSet();

        for (String d: deadends) dead.add(d);

        Queue<String> queue = new LinkedList();

        queue.offer("0000");

        queue.offer(null);

        Set<String> seen = new HashSet();

        seen.add("0000");

        int depth = 0;

        while (!queue.isEmpty()) {

            String node = queue.poll();

            if (node == null) {

                depth++;

                if (queue.peek() != null)

                    queue.offer(null);

            } else if (node.equals(target)) {

                return depth;

            } else if (!dead.contains(node)) {

                for (int i = 0; i < 4; ++i) {

                    for (int d = -1; d <= 1; d += 2) {

                        int y = ((node.charAt(i) - '0') + d + 10) % 10;

                        String nei = node.substring(0, i) + ("" + y) + node.substring(i+1);

                        if (!seen.contains(nei)) {

                            seen.add(nei);

                            queue.offer(nei);

                        }

                    }

                }

            }

        }

        return -1;

    }

}

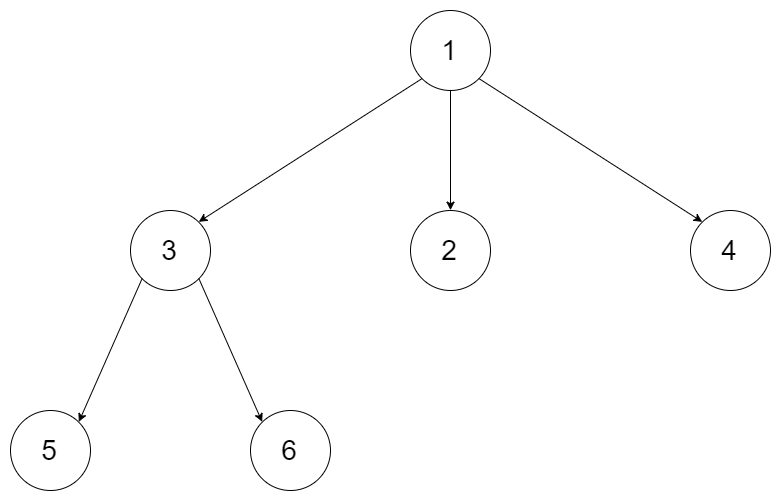
### N-ary Tree Level Order Traversal

**Problem Statement 16**

Given an n-ary tree, return the *level order* traversal of its nodes' values.

*Nary-Tree input serialization is represented in their level order traversal, each group of children is separated by the null value (See examples).*

**Example 1:**



**Input:** root = [1,null,3,2,4,null,5,6]

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

**Example 2:**



**Input:** root = [1,null,2,3,4,5,null,null,6,7,null,8,null,9,10,null,null,11,null,12,null,13,null,null,14]

**Output:** [[1],[2,3,4,5],[6,7,8,9,10],[11,12,13],[14]]

**Constraints:**

* The height of the n-ary tree is less than or equal to 1000
* The total number of nodes is between [0, 10^4]

/\*

// Definition for a Node.

class Node {

    public int val;

    public List<Node> children;

    public Node() {}

    public Node(int \_val,List<Node> \_children) {

        val = \_val;

        children = \_children;

    }

};

\*/

class Solution {

    public List<List<Integer>> levelOrder(Node root) {

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

        if (root == null)

            return result;

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

        queue.offer(root);

        queue.offer(null);

        Node temp = null;

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

        while(!queue.isEmpty()) {

            temp = queue.poll();

            if (temp == null) {

                result.add(list);

                if (queue.isEmpty()) {

                    break;

                }

                else {

                    queue.offer(null);

                    list = new ArrayList<>();

                }

            }

            else {

                list.add(temp.val);

                if (temp.children.size() != 0) {

                    for (Node node : temp.children) {

                        queue.offer(node);

                    }

                }

            }

        }

        return result;

    }

}

More simply,

class Solution {

    public List<List<Integer>> levelOrder(Node root) {

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

        levelOrderUtil(root, levelOrder, 0);

        return levelOrder;

    }

    public void levelOrderUtil(Node root, List<List<Integer>> levelOrder, int level) {

        if (root == null) {

            return;

        }

        if (levelOrder.size() <= level) {

            levelOrder.add(new ArrayList<>());

        }

        levelOrder.get(level).add(root.val);

        for (Node child : root.children) {

            levelOrderUtil(child, levelOrder, level + 1);

        }

    }

}

### Clone N-ary Tree

**Problem Statement 17**

Given a root of an N-ary tree, return a [**deep copy**](https://en.wikipedia.org/wiki/Object_copying#Deep_copy) (clone) of the tree.

Each node in the n-ary tree contains a val (int) and a list (List[Node]) of its children.

class Node {

public int val;

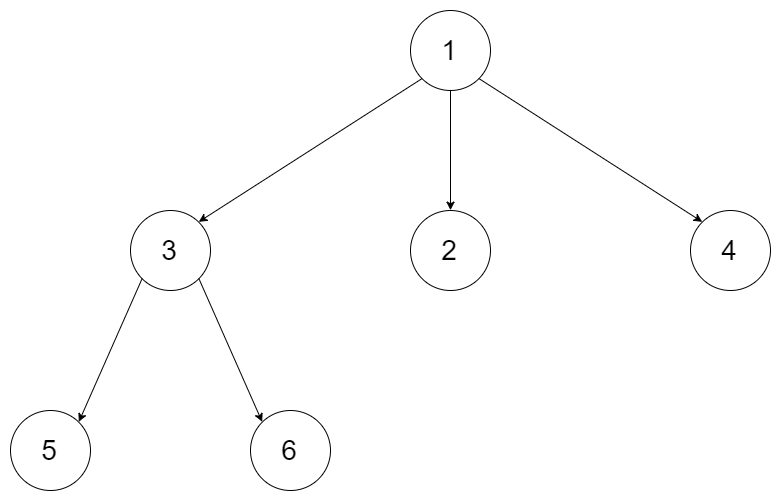
public List<Node> children;

}

Nary-Tree input serialization is represented in their level order traversal, each group of children is separated by the null value (See examples).

**Follow up:**Can your solution work for the [graph problem](https://leetcode.com/problems/clone-graph/)?

**Example 1:**



**Input:** root = [1,null,3,2,4,null,5,6]

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

**Example 2:**



**Input:** root = [1,null,2,3,4,5,null,null,6,7,null,8,null,9,10,null,null,11,null,12,null,13,null,null,14]

**Output:** [1,null,2,3,4,5,null,null,6,7,null,8,null,9,10,null,null,11,null,12,null,13,null,null,14]

class Solution {

    public Node cloneTree(Node root) {

        if(root == null) return null;

        Node new\_node = new Node(root.val);

        Queue<Node> q1 = new LinkedList<Node>();

        Queue<Node> q2 = new LinkedList<Node>();

        q1.offer(root);

        q2.offer(new\_node);

        while(!q1.isEmpty()){

            int size = q1.size();

            for(int i=0;i<size;++i){

                Node t1 = q1.poll();

                Node t2 = q2.poll();

                List<Node> kids = t1.children;

                for(int j=0;j<kids.size();++j){

                    Node new\_k = new Node(kids.get(j).val);

                    t2.children.add(new\_k);

                    q1.offer(kids.get(j));

                    q2.offer(new\_k);

                }

            }

        }

        return new\_node;

    }

}

Alternate solution,

class Solution {

    // same code will work perfectly fine for graph as well

    public Node cloneTree(Node root) {

        if (root == null) {

            return null;

        }

        Map<Node, Node> originalToClone = new HashMap<>();

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

        q.add(root);

        originalToClone.put(root, new Node(root.val));

        while(!q.isEmpty()) {

            Node node = q.poll();

            Node clone = originalToClone.get(node);

            for (Node child: node.children) {

                if (!originalToClone.containsKey(child)) {

                    Node adj = new Node(child.val);

                    originalToClone.put(child, adj);

                    q.add(child);

                }

                clone.children.add(originalToClone.get(child));

            }

        }

        return originalToClone.get(root);

    }

}

DFS Approach,

class Solution {

    public Node cloneTree(Node root) {

        if (root == null) return null;

        Node cloneRoot = new Node(root.val);

        for (Node adj: root.children) {

            cloneRoot.children.add(cloneTree(adj));

        }

        return cloneRoot;

    }

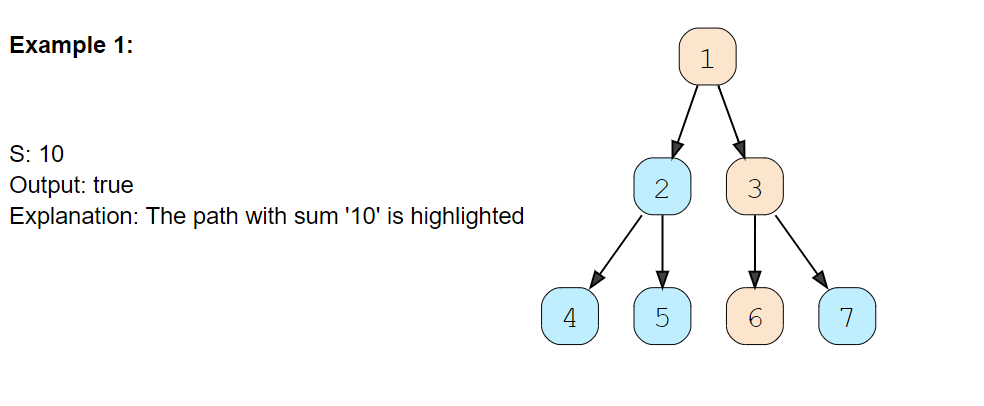
}

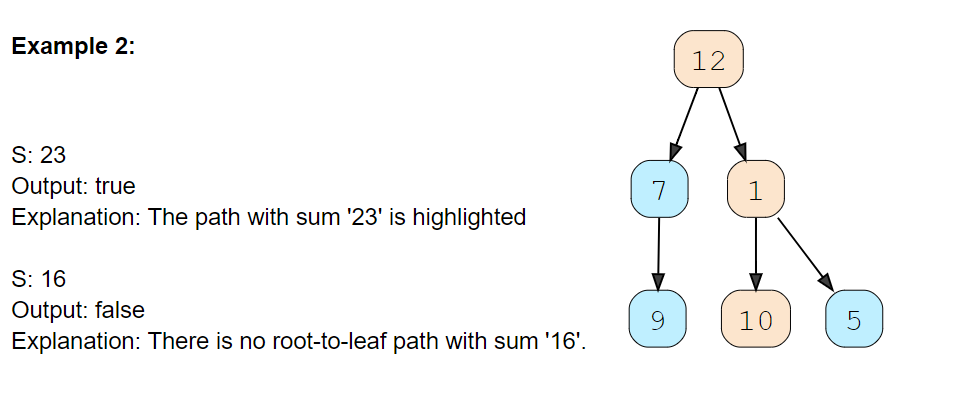
**Some more DFS problems**

# **Binary Tree Path Sum**

### Problem Statement [#](https://www.educative.io/courses/grokking-the-coding-interview/RMlGwgpoKKY#problem-statement)

Given a binary tree and a number ‘S’, find if the tree has a path from root-to-leaf such that the sum of all the node values of that path equals ‘S’.





**Solution**[#](https://www.educative.io/courses/grokking-the-coding-interview/RMlGwgpoKKY#solution)

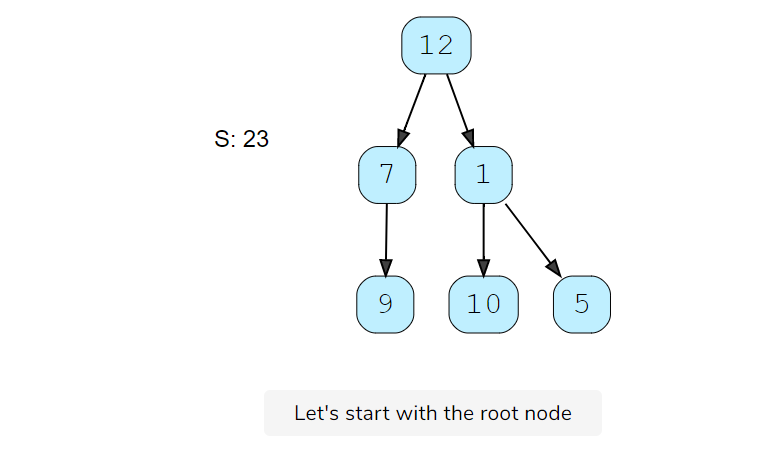
As we are trying to search for a root-to-leaf path, we can use the **Depth First Search (DFS)** technique to solve this problem.

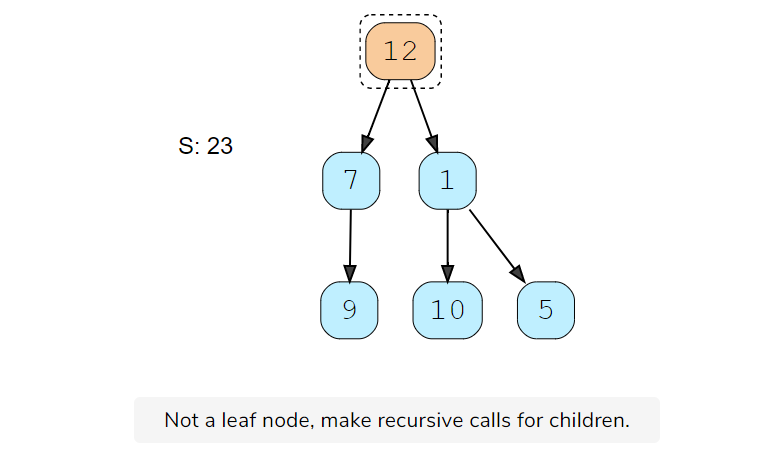
To recursively traverse a binary tree in a DFS fashion, we can start from the root and at every step, make two recursive calls one for the left and one for the right child.

Here are the steps for our Binary Tree Path Sum problem:

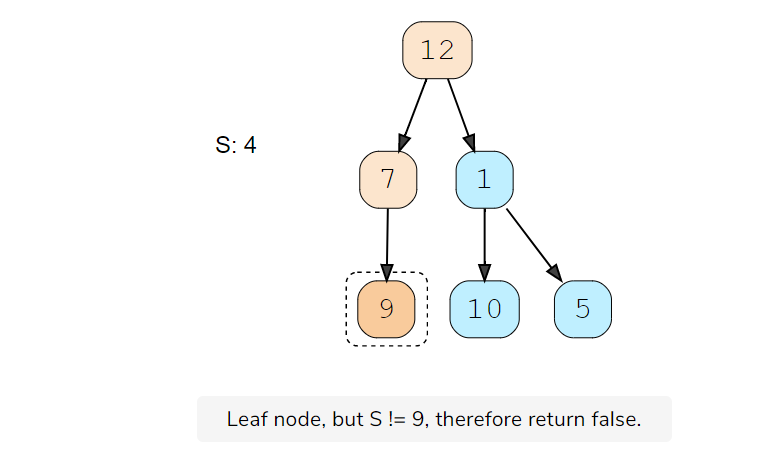
1. Start DFS with the root of the tree.
2. If the current node is not a leaf node, do two things:
   * Subtract the value of the current node from the given number to get a new sum => S = S - node.value
   * Make two recursive calls for both the children of the current node with the new number calculated in the previous step.
3. At every step, see if the current node being visited is a leaf node and if its value is equal to the given number ‘S’. If both these conditions are true, we have found the required root-to-leaf path, therefore return true.
4. If the current node is a leaf but its value is not equal to the given number ‘S’, return false.

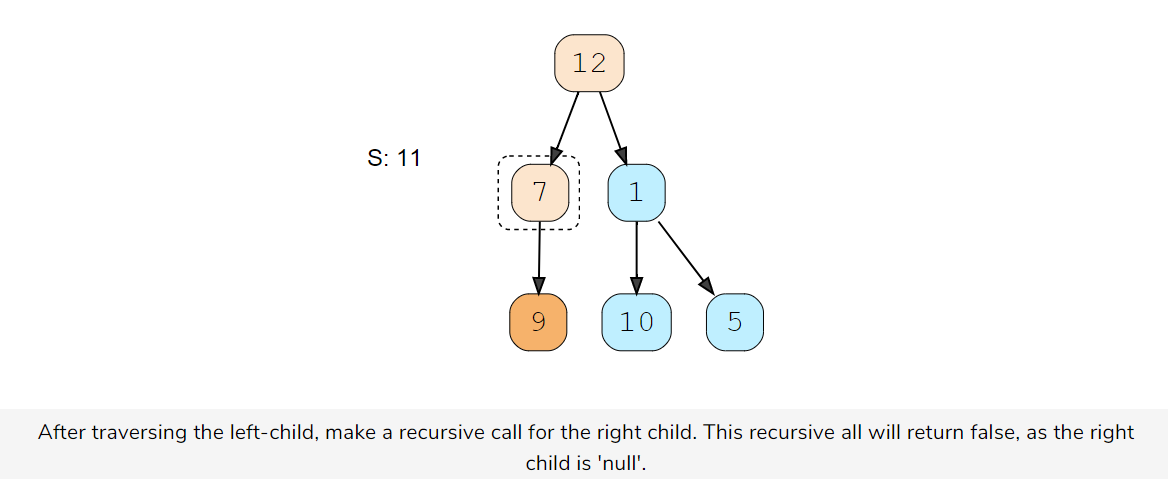
Let’s take the example-2 mentioned above to visually see our algorithm:

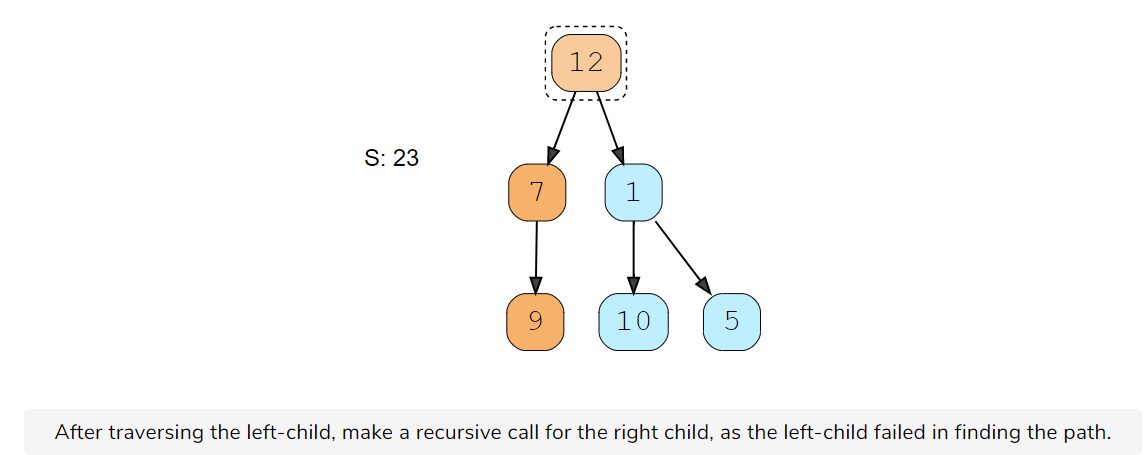


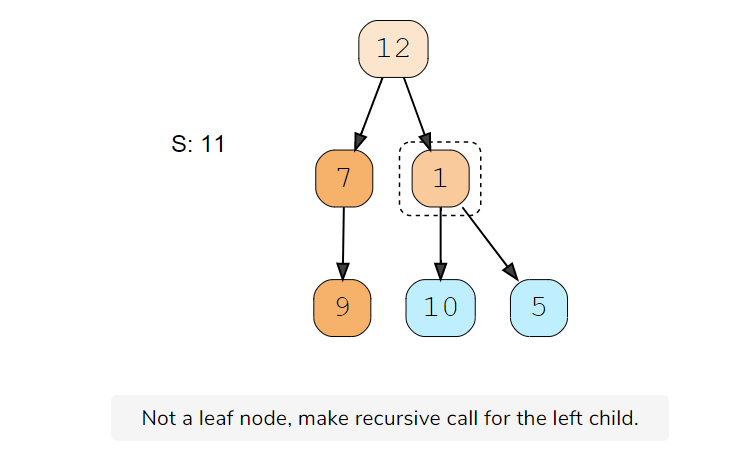


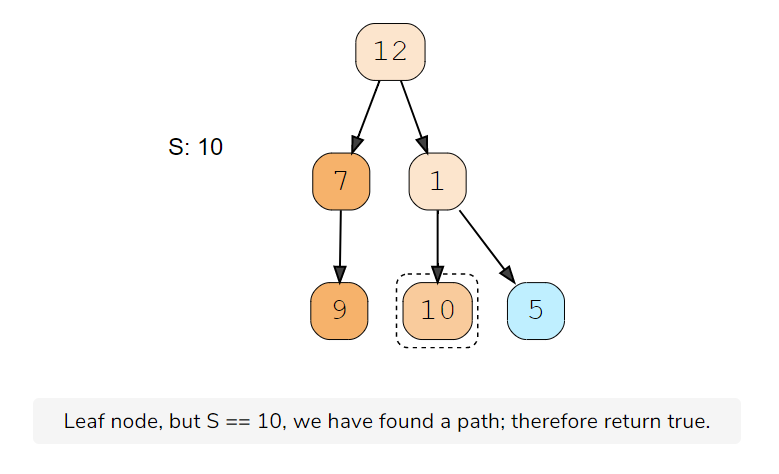


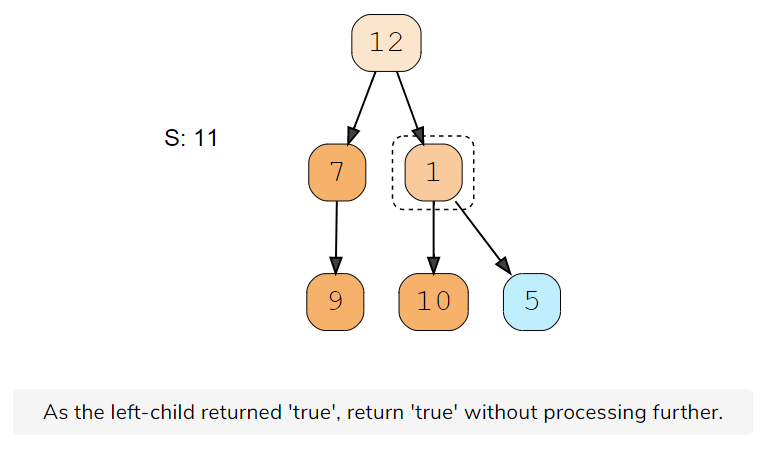


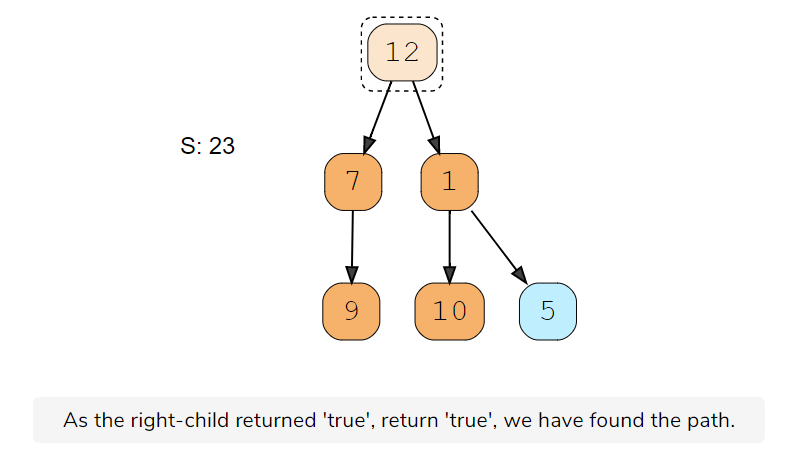












### Code [#](https://www.educative.io/courses/grokking-the-coding-interview/RMlGwgpoKKY#code)

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode(int x) {

    val = x;

  }

};

class TreePathSum {

  public static boolean hasPath(TreeNode root, int sum) {

    if (root == null)

      return false;

    // if the current node is a leaf and its value is equal to the sum, we've found a path

    if (root.val == sum && root.left == null && root.right == null)

      return true;

    // recursively call to traverse the left and right sub-tree

    // return true if any of the two recursive call return true

    return hasPath(root.left, sum - root.val) || hasPath(root.right, sum - root.val);

  }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(12);

    root.left = new TreeNode(7);

    root.right = new TreeNode(1);

    root.left.left = new TreeNode(9);

    root.right.left = new TreeNode(10);

    root.right.right = new TreeNode(5);

    System.out.println("Tree has path: " + TreePathSum.hasPath(root, 23));

    System.out.println("Tree has path: " + TreePathSum.hasPath(root, 16));

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/RMlGwgpoKKY#time-complexity)

The time complexity of the above algorithm is O(N)*O*(*N*), where ‘N’ is the total number of nodes in the tree. This is due to the fact that we traverse each node once.

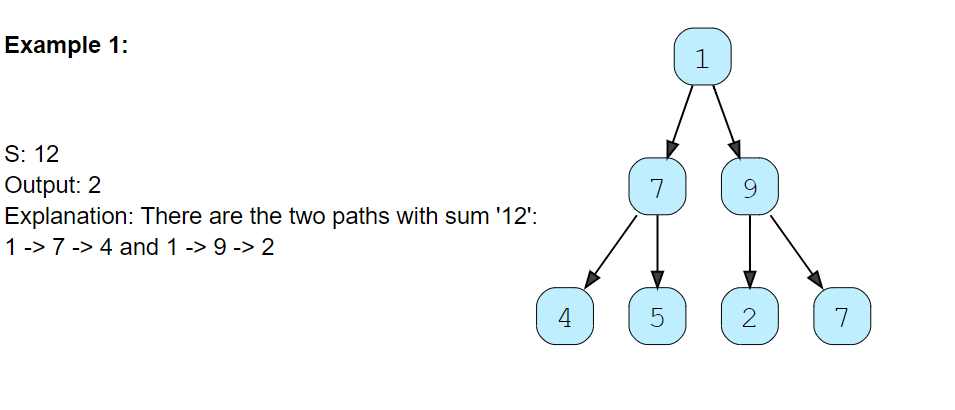
#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/RMlGwgpoKKY#space-complexity)

The space complexity of the above algorithm will be O(N)*O*(*N*) in the worst case. This space will be used to store the recursion stack. The worst case will happen when the given tree is a linked list (i.e., every node has only one child).

# **All Paths for a Sum**

### Problem Statement 2 [#](https://www.educative.io/courses/grokking-the-coding-interview/B815A0y2Ajn#problem-statement)

Given a binary tree and a number ‘S’, find all paths from root-to-leaf such that the sum of all the node values of each path equals ‘S’.



### Solution [#](https://www.educative.io/courses/grokking-the-coding-interview/B815A0y2Ajn#solution)

This problem follows the Binary Tree Path Sum pattern. We can follow the same **DFS** approach. There will be two differences:

1. Every time we find a root-to-leaf path, we will store it in a list.
2. We will traverse all paths and will not stop processing after finding the first path.

### Code [#](https://www.educative.io/courses/grokking-the-coding-interview/B815A0y2Ajn#code)

import java.util.\*;

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode(int x) {

    val = x;

  }

};

class FindAllTreePaths {

  public static List<List<Integer>> findPaths(TreeNode root, int sum) {

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

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

    findPathsRecursive(root, sum, currentPath, allPaths);

    return allPaths;

  }

  private static void findPathsRecursive(TreeNode currentNode, int sum, List<Integer> currentPath,

      List<List<Integer>> allPaths) {

    if (currentNode == null)

      return;

    // add the current node to the path

    currentPath.add(currentNode.val);

    // if the current node is a leaf and its value is equal to sum, save the current path

    if (currentNode.val == sum && currentNode.left == null && currentNode.right == null) {

      allPaths.add(new ArrayList<Integer>(currentPath));

    } else {

      // traverse the left sub-tree

      findPathsRecursive(currentNode.left, sum - currentNode.val, currentPath, allPaths);

      // traverse the right sub-tree

      findPathsRecursive(currentNode.right, sum - currentNode.val, currentPath, allPaths);

    }

    // remove the current node from the path to backtrack,

    // we need to remove the current node while we are going up the recursive call stack.

    currentPath.remove(currentPath.size() - 1);

  }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(12);

    root.left = new TreeNode(7);

    root.right = new TreeNode(1);

    root.left.left = new TreeNode(4);

    root.right.left = new TreeNode(10);

    root.right.right = new TreeNode(5);

    int sum = 23;

    List<List<Integer>> result = FindAllTreePaths.findPaths(root, sum);

    System.out.println("Tree paths with sum " + sum + ": " + result);

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/B815A0y2Ajn#time-complexity)

The time complexity of the above algorithm is O(N^2)*O*(*N*​2​​), where ‘N’ is the total number of nodes in the tree. This is due to the fact that we traverse each node once (which will take O(N)*O*(*N*)), and for every leaf node we might have to store its path which will take O(N)*O*(*N*).

We can calculate a tighter time complexity of O(NlogN)*O*(*NlogN*) from the space complexity discussion below.

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/B815A0y2Ajn#space-complexity)

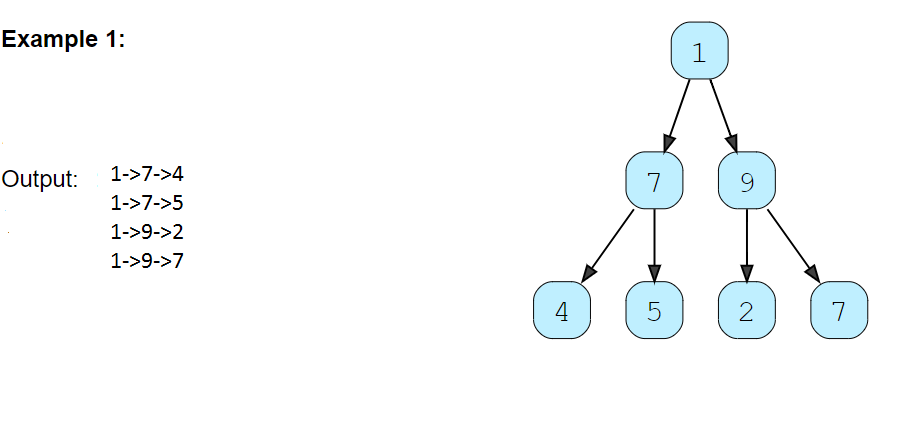
If we ignore the space required for the allPaths list, the space complexity of the above algorithm will be O(N)*O*(*N*) in the worst case. This space will be used to store the recursion stack. The worst case will happen when the given tree is a linked list (i.e., every node has only one child).

How can we estimate the space used for the allPaths array? Take the example of the following balanced tree:

# **All root-to-leaf paths**

### Problem Statement 3 [#](https://www.educative.io/courses/grokking-the-coding-interview/B815A0y2Ajn#problem-statement)

Given a binary tree, return all root-to-leaf paths



### Solution [#](https://www.educative.io/courses/grokking-the-coding-interview/B815A0y2Ajn#solution)

This problem follows the Binary Tree Path Sum pattern. We can follow the same **DFS** approach. There will be two differences:

1. Every time we find a root-to-leaf path, we will store it in a list.
2. We will traverse all paths and will not stop processing after finding the first path.

### Code [#](https://www.educative.io/courses/grokking-the-coding-interview/B815A0y2Ajn#code)

import java.util.\*;

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode(int x) {

    val = x;

  }

};

class FindAllTreePaths {

  public static List<List<Integer>> findPaths(TreeNode root) {

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

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

    findPathsRecursive(root, sum, currentPath, allPaths);

    return allPaths;

  }

  private static void findPathsRecursive(TreeNode currentNode, List<Integer> currentPath,

      List<List<Integer>> allPaths) {

    if (currentNode == null)

      return;

    // add the current node to the path

    currentPath.add(currentNode.val);

    // if the current node is a leaf , save the current path

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

      allPaths.add(new ArrayList<Integer>(currentPath));

    } else {

      // traverse the left sub-tree

      findPathsRecursive(currentNode.left, currentPath, allPaths);

      // traverse the right sub-tree

      findPathsRecursive(currentNode.right, currentPath, allPaths);

    }

    // remove the current node from the path to backtrack,

    // we need to remove the current node while we are going up the recursive call stack.

    currentPath.remove(currentPath.size() - 1);

  }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(12);

    root.left = new TreeNode(7);

    root.right = new TreeNode(1);

    root.left.left = new TreeNode(4);

    root.right.left = new TreeNode(10);

    root.right.right = new TreeNode(5);

    List<List<Integer>> result = FindAllTreePaths.findPaths(root);

    System.out.println("Tree paths : " + result);

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/B815A0y2Ajn#time-complexity)

The time complexity of the above algorithm is O(N^2)*O*(*N*​2​​), where ‘N’ is the total number of nodes in the tree. This is due to the fact that we traverse each node once (which will take O(N)*O*(*N*)), and for every leaf node we might have to store its path which will take O(N)*O*(*N*).

We can calculate a tighter time complexity of O(NlogN)*O*(*NlogN*) from the space complexity discussion below.

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/B815A0y2Ajn#space-complexity)

If we ignore the space required for the allPaths list, the space complexity of the above algorithm will be O(N)*O*(*N*) in the worst case. This space will be used to store the recursion stack. The worst case will happen when the given tree is a linked list (i.e., every node has only one child).

# **Root-to-leaf path with maximum sum**

### Problem Statement 4 [#](https://www.educative.io/courses/grokking-the-coding-interview/B815A0y2Ajn#problem-statement)

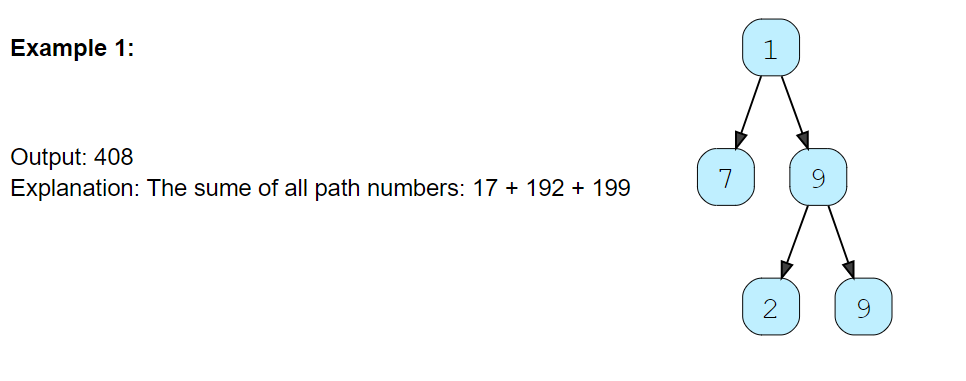
Given a binary tree, return root-to-leaf path with maximum sum

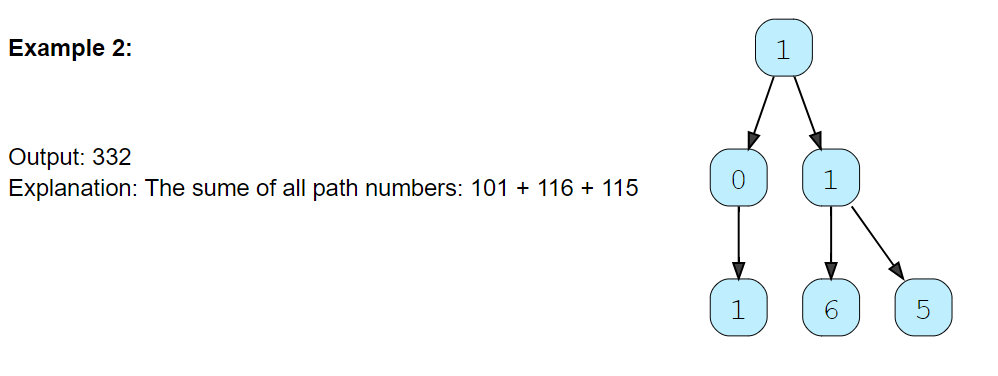
Solution: We need to find the path with the maximum sum. As we traverse all paths, we can keep track of the path with the maximum sum.

# **Sum of Path Numbers**

### Problem Statement 5 [#](https://www.educative.io/courses/grokking-the-coding-interview/YQ5o5vEXP69#problem-statement)

Given a binary tree where each node can only have a digit (0-9) value, each root-to-leaf path will represent a number. Find the total sum of all the numbers represented by all paths.





### Solution [#](https://www.educative.io/courses/grokking-the-coding-interview/YQ5o5vEXP69#solution)

This problem follows the Binary Tree Path sum pattern. We can follow the same **DFS** approach. The additional thing we need to do is to keep track of the number representing the current path.

How do we calculate the path number for a node? Taking the first example mentioned above, say we are at node ‘7’. As we know, the path number for this node is ‘17’, which was calculated by: 1 \* 10 + 7 => 17. We will follow the same approach to calculate the path number of each node.

### Code

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode(int x) {

    val = x;

  }

};

class SumOfPathNumbers {

  public static int findSumOfPathNumbers(TreeNode root) {

    return findRootToLeafPathNumbers(root, 0);

  }

  private static int findRootToLeafPathNumbers(TreeNode currentNode, int pathSum) {

    if (currentNode == null)

      return 0;

    // calculate the path number of the current node

    pathSum = 10 \* pathSum + currentNode.val;

    // if the current node is a leaf, return the current path sum.

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

      return pathSum;

    }

    // traverse the left and the right sub-tree

    return findRootToLeafPathNumbers(currentNode.left, pathSum) +

           findRootToLeafPathNumbers(currentNode.right, pathSum);

  }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(1);

    root.left = new TreeNode(0);

    root.right = new TreeNode(1);

    root.left.left = new TreeNode(1);

    root.right.left = new TreeNode(6);

    root.right.right = new TreeNode(5);

    System.out.println("Total Sum of Path Numbers: " + SumOfPathNumbers.findSumOfPathNumbers(root));

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/YQ5o5vEXP69#time-complexity)

The time complexity of the above algorithm is O(N)*O*(*N*), where ‘N’ is the total number of nodes in the tree. This is due to the fact that we traverse each node once.

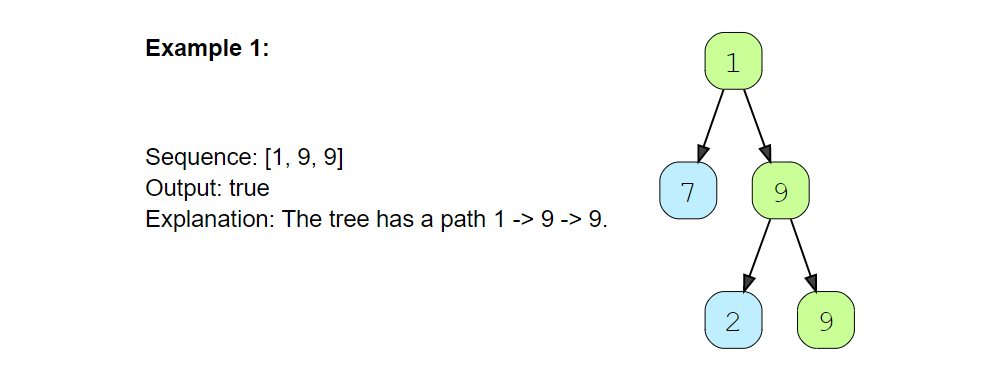
#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/YQ5o5vEXP69#space-complexity)

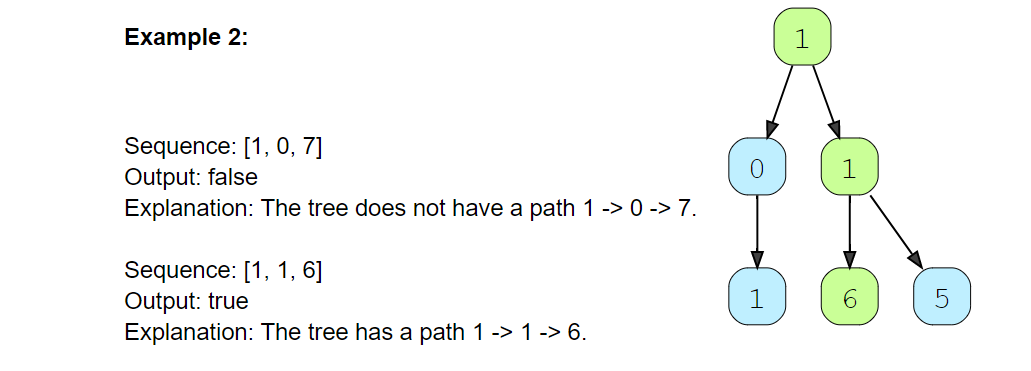
The space complexity of the above algorithm will be O(N)*O*(*N*) in the worst case. This space will be used to store the recursion stack. The worst case will happen when the given tree is a linked list (i.e., every node has only one child).

# **Path with Given Sequence**

### Problem Statement 6

### Given a binary tree and a number sequence, find if the sequence is present as a root-to-leaf path in the given tree. [#](https://www.educative.io/courses/grokking-the-coding-interview/YQ5o5vEXP69#problem-statement)





### Solution [#](https://www.educative.io/courses/grokking-the-coding-interview/m280XNlPOkn#solution)

This problem follows the Binary Tree path sum pattern. We can follow the same **DFS** approach and additionally, track the element of the given sequence that we should match with the current node. Also, we can return false as soon as we find a mismatch between the sequence and the node value.

import java.util.\*;

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode(int x) {

    val = x;

  }

};

class PathWithGivenSequence {

  public static boolean findPath(TreeNode root, int[] sequence) {

    if (root == null)

      return sequence.length == 0;

    return findPathRecursive(root, sequence, 0);

  }

  private static boolean findPathRecursive(TreeNode currentNode, int[] sequence, int sequenceIndex) {

    if (currentNode == null)

      return false;

    if (sequenceIndex >= sequence.length || currentNode.val != sequence[sequenceIndex])

      return false;

    // if the current node is a leaf, add it is the end of the sequence, we have found a path!

    if (currentNode.left == null && currentNode.right == null && sequenceIndex == sequence.length - 1)

      return true;

    // recursively call to traverse the left and right sub-tree

    // return true if any of the two recusrive call return true

    return findPathRecursive(currentNode.left, sequence, sequenceIndex + 1)

        || findPathRecursive(currentNode.right, sequence, sequenceIndex + 1);

  }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(1);

    root.left = new TreeNode(0);

    root.right = new TreeNode(1);

    root.left.left = new TreeNode(1);

    root.right.left = new TreeNode(6);

    root.right.right = new TreeNode(5);

    System.out.println("Tree has path sequence: " + PathWithGivenSequence.findPath(root, new int[] { 1, 0, 7 }));

    System.out.println("Tree has path sequence: " + PathWithGivenSequence.findPath(root, new int[] { 1, 1, 6 }));

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/m280XNlPOkn#time-complexity)

The time complexity of the above algorithm is O(N)*O*(*N*), where ‘N’ is the total number of nodes in the tree. This is due to the fact that we traverse each node once.

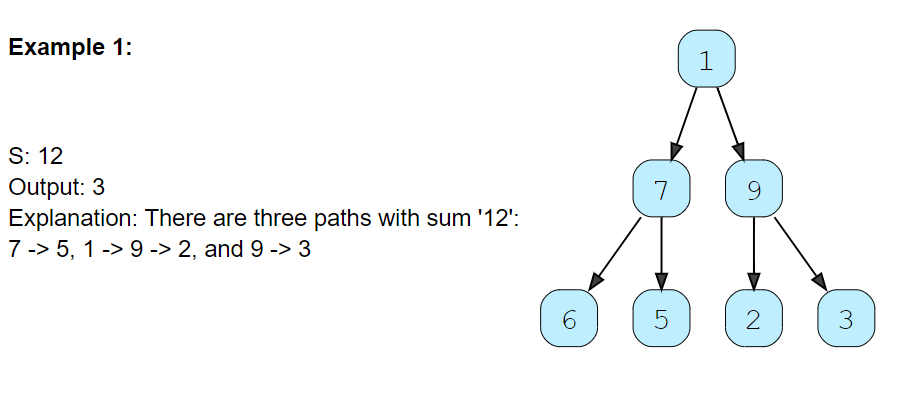
#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/m280XNlPOkn#space-complexity)

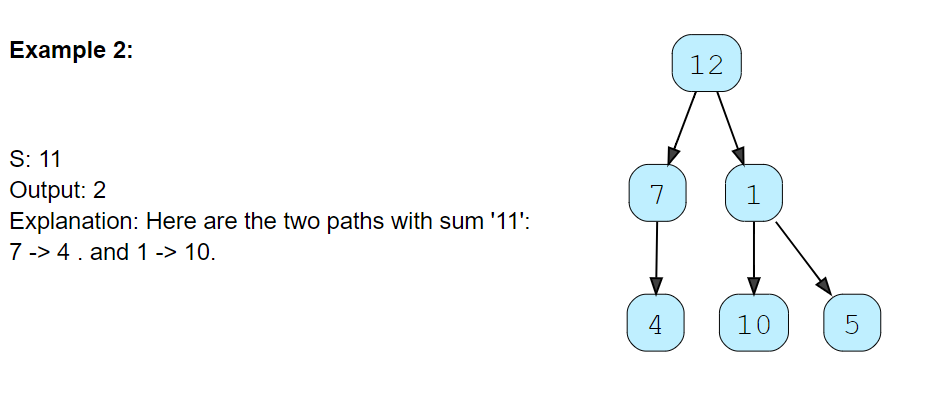
The space complexity of the above algorithm will be O(N)*O*(*N*) in the worst case. This space will be used to store the recursion stack. The worst case will happen when the given tree is a linked list (i.e., every node has only one child).

# **Count Paths for a Sum**

### Problem Statement 7

Given a binary tree and a number ‘S’, find all paths in the tree such that the sum of all the node values of each path equals ‘S’. Please note that the paths can start or end at any node but all paths must follow direction from parent to child (top to bottom).





### Solution [#](https://www.educative.io/courses/grokking-the-coding-interview/xV2J7jvN1or#solution)

This problem follows the Binary Tree Path Sum pattern. We can follow the same **DFS** approach. But there will be four differences:

1. We will keep track of the current path in a list which will be passed to every recursive call.
2. Whenever we traverse a node we will do two things:
   * Add the current node to the current path.
   * As we added a new node to the current path, we should find the sums of all sub-paths ending at the current node. If the sum of any sub-path is equal to ‘S’ we will increment our path count.
3. We will traverse all paths and will not stop processing after finding the first path.
4. Remove the current node from the current path before returning from the function. This is needed to **Backtrack** while we are going up the recursive call stack to process other paths.

### Code [#](https://www.educative.io/courses/grokking-the-coding-interview/xV2J7jvN1or#code)

import java.util.\*;

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode(int x) {

    val = x;

  }

};

class CountAllPathSum {

  public static int countPaths(TreeNode root, int S) {

    List<Integer> currentPath = new LinkedList<>();

    return countPathsRecursive(root, S, currentPath);

  }

  private static int countPathsRecursive(TreeNode currentNode, int S, List<Integer> currentPath) {

    if (currentNode == null)

      return 0;

    // add the current node to the path

    currentPath.add(currentNode.val);

    int pathCount = 0, pathSum = 0;

    // find the sums of all sub-paths in the current path list

    ListIterator<Integer> pathIterator = currentPath.listIterator(currentPath.size());

    while (pathIterator.hasPrevious()) {

      pathSum += pathIterator.previous();

      // if the sum of any sub-path is equal to 'S' we increment our path count.

      if (pathSum == S) {

        pathCount++;

      }

    }

    // traverse the left sub-tree

    pathCount += countPathsRecursive(currentNode.left, S, currentPath);

    // traverse the right sub-tree

    pathCount += countPathsRecursive(currentNode.right, S, currentPath);

    // remove the current node from the path to backtrack,

    // we need to remove the current node while we are going up the recursive call stack.

    currentPath.remove(currentPath.size() - 1);

    return pathCount;

  }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(12);

    root.left = new TreeNode(7);

    root.right = new TreeNode(1);

    root.left.left = new TreeNode(4);

    root.right.left = new TreeNode(10);

    root.right.right = new TreeNode(5);

    System.out.println("Tree has path: " + CountAllPathSum.countPaths(root, 11));

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/xV2J7jvN1or#time-complexity)

The time complexity of the above algorithm is O(N^2)*O*(*N*​2​​) in the worst case, where ‘N’ is the total number of nodes in the tree. This is due to the fact that we traverse each node once, but for every node, we iterate the current path. The current path, in the worst case, can be O(N)*O*(*N*) (in the case of a skewed tree). But, if the tree is balanced, then the current path will be equal to the height of the tree, i.e., O(logN)*O*(*logN*). So the best case of our algorithm will be O(NlogN)*O*(*NlogN*).

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/xV2J7jvN1or#space-complexity)

The space complexity of the above algorithm will be O(N)*O*(*N*). This space will be used to store the recursion stack. The worst case will happen when the given tree is a linked list (i.e., every node has only one child). We also need O(N)*O*(*N*) space for storing the currentPath in the worst case.

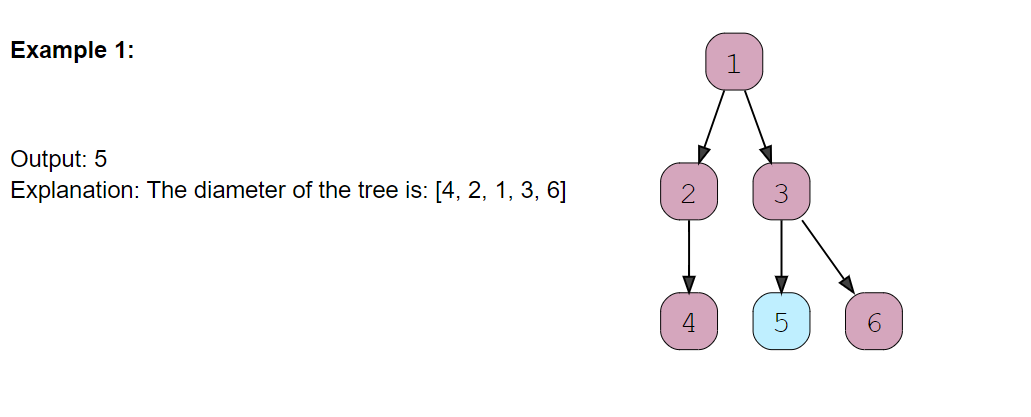
Overall space complexity of our algorithm is O(N)*O*(*N*).

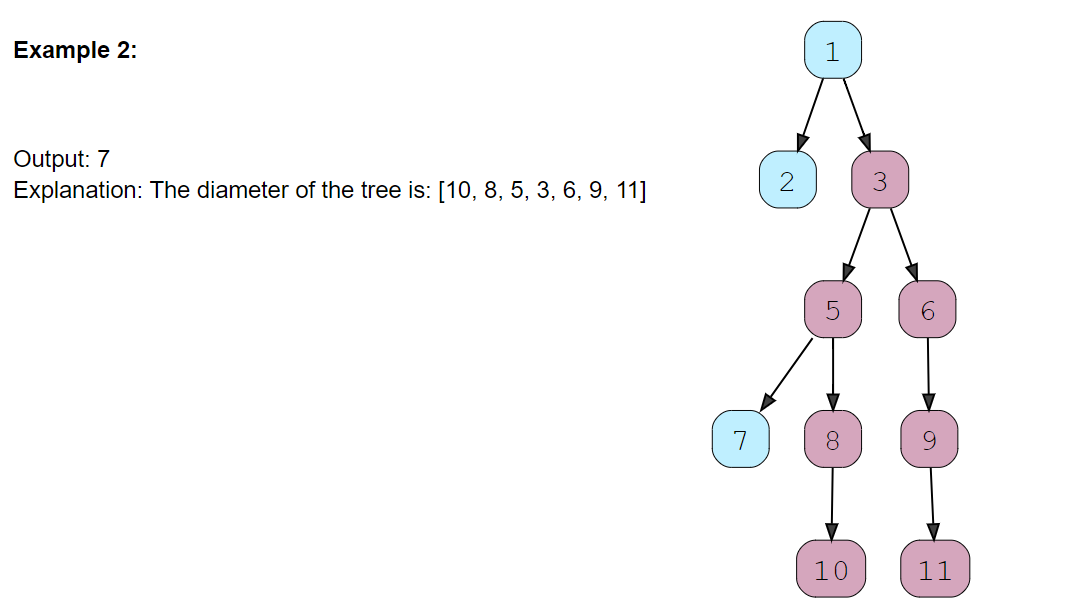
### Tree Diameter (medium)

### Problem Statement 7

Given a binary tree, find the length of its diameter. The diameter of a tree is the number of nodes on the **longest path between any two leaf nodes**. The diameter of a tree may or may not pass through the root.

Note: You can always assume that there are at least two leaf nodes in the given tree.





### Solution [#](https://www.educative.io/courses/grokking-the-coding-interview/JYVW7l2L4EJ#solution)

This problem follows the Binary Tree Path sum pattern. We can follow the same **DFS** approach. There will be a few differences:

1. At every step, we need to find the height of both children of the current node. For this, we will make two recursive calls similar to **DFS**.
2. The height of the current node will be equal to the maximum of the heights of its left or right children, plus ‘1’ for the current node.
3. The tree diameter at the current node will be equal to the height of the left child plus the height of the right child plus ‘1’ for the current node: diameter = leftTreeHeight + rightTreeHeight + 1. To find the overall tree diameter, we will use a class level variable. This variable will store the maximum diameter of all the nodes visited so far, hence, eventually, it will have the final tree diameter.

### Code [#](https://www.educative.io/courses/grokking-the-coding-interview/JYVW7l2L4EJ#code)

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode(int x) {

    val = x;

  }

};

class TreeDiameter {

  private static int treeDiameter = 0;

  public static int findDiameter(TreeNode root) {

    calculateHight(root);

    return treeDiameter;

  }

  private static int calculateHight(TreeNode currentNode) {

    if (currentNode == null)

      return 0;

    int leftTreeHeight = calculateHight(currentNode.left);

    int rightTreeHeight = calculateHight(currentNode.right);

    // diameter at the current node will be equal to the height of left subtree +

    // the height of right sub-trees + '1' for the current node

    int diameter = leftTreeHeight + rightTreeHeight + 1;

    // update the global tree diameter

    treeDiameter = Math.max(treeDiameter, diameter);

    // height of the current node will be equal to the maximum of the hights of

    // left or right subtrees plus '1' for the current node

    return Math.max(leftTreeHeight, rightTreeHeight) + 1;

  }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(1);

    root.left = new TreeNode(2);

    root.right = new TreeNode(3);

    root.left.left = new TreeNode(4);

    root.right.left = new TreeNode(5);

    root.right.right = new TreeNode(6);

    System.out.println("Tree Diameter: " + TreeDiameter.findDiameter(root));

    root.left.left = null;

    root.right.left.left = new TreeNode(7);

    root.right.left.right = new TreeNode(8);

    root.right.right.left = new TreeNode(9);

    root.right.left.right.left = new TreeNode(10);

    root.right.right.left.left = new TreeNode(11);

    System.out.println("Tree Diameter: " + TreeDiameter.findDiameter(root));

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/JYVW7l2L4EJ#time-complexity)

The time complexity of the above algorithm is O(N)*O*(*N*), where ‘N’ is the total number of nodes in the tree. This is due to the fact that we traverse each node once.

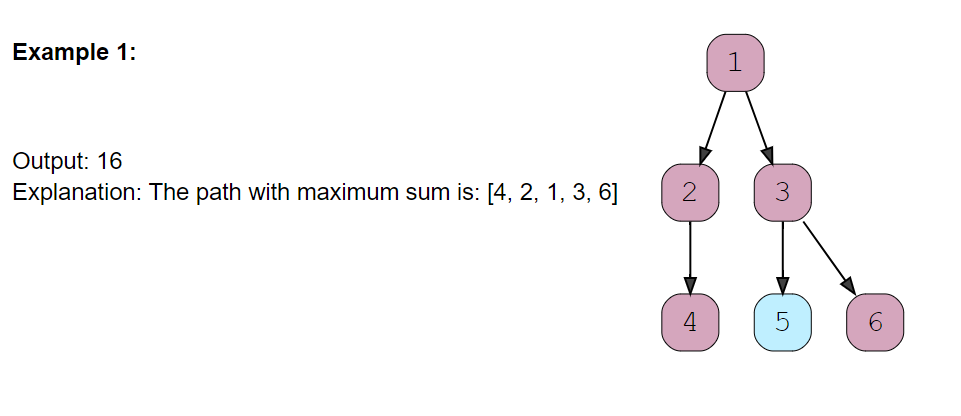
#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/JYVW7l2L4EJ#space-complexity)

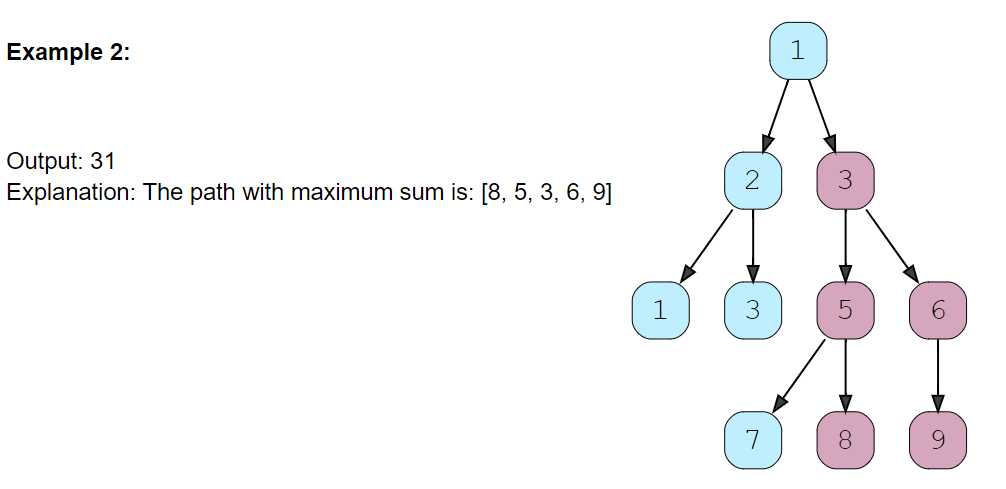
The space complexity of the above algorithm will be O(N)*O*(*N*) in the worst case. This space will be used to store the recursion stack. The worst case will happen when the given tree is a linked list (i.e., every node has only one child).

### Path with Maximum Sum

### Problem Statement 8

### Find the path with the maximum sum in a given binary tree. Write a function that returns the maximum sum. A path can be defined as a ****sequence of nodes between any two nodes**** and doesn’t necessarily pass through the root.





### Solution [#](https://www.educative.io/courses/grokking-the-coding-interview/xVPgnOvWVJq#solution)

This problem follows the Binary Tree Path Sum pattern and shares the algorithmic logic with [Tree Diameter](https://www.educative.io/collection/page/5668639101419520/5671464854355968/5691878833913856/). We can follow the same **DFS** approach. The only difference will be to ignore the paths with negative sums. Since we need to find the overall maximum sum, we should ignore any path which has an overall negative sum.

### Code [#](https://www.educative.io/courses/grokking-the-coding-interview/xVPgnOvWVJq#code)

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode(int x) {

    val = x;

  }

};

class MaximumPathSum {

  private static int globalMaximumSum;

  public static int findMaximumPathSum(TreeNode root) {

    globalMaximumSum = Integer.MIN\_VALUE;

    findMaximumPathSumRecursive(root);

    return globalMaximumSum;

  }

  private static int findMaximumPathSumRecursive(TreeNode currentNode) {

    if (currentNode == null)

      return 0;

    int maxPathSumFromLeft = findMaximumPathSumRecursive(currentNode.left);

    int maxPathSumFromRight = findMaximumPathSumRecursive(currentNode.right);

    // ignore paths with negative sums, since we need to find the maximum sum we should

    // ignore any path which has an overall negative sum.

    maxPathSumFromLeft = Math.max(maxPathSumFromLeft, 0);

    maxPathSumFromRight = Math.max(maxPathSumFromRight, 0);

    // maximum path sum at the current node will be equal to the sum from the left subtree +

    // the sum from right subtree + val of current node

    int localMaximumSum = maxPathSumFromLeft + maxPathSumFromRight + currentNode.val;

    // update the global maximum sum

    globalMaximumSum = Math.max(globalMaximumSum, localMaximumSum);

    // maximum sum of any path from the current node will be equal to the maximum of

    // the sums from left or right subtrees plus the value of the current node

    return Math.max(maxPathSumFromLeft, maxPathSumFromRight) + currentNode.val;

  }

  public static void main(String[] args) {

    TreeNode root = new TreeNode(1);

    root.left = new TreeNode(2);

    root.right = new TreeNode(3);

    System.out.println("Maximum Path Sum: " + MaximumPathSum.findMaximumPathSum(root));

    root.left.left = new TreeNode(1);

    root.left.right = new TreeNode(3);

    root.right.left = new TreeNode(5);

    root.right.right = new TreeNode(6);

    root.right.left.left = new TreeNode(7);

    root.right.left.right = new TreeNode(8);

    root.right.right.left = new TreeNode(9);

    System.out.println("Maximum Path Sum: " + MaximumPathSum.findMaximumPathSum(root));

    root = new TreeNode(-1);

    root.left = new TreeNode(-3);

    System.out.println("Maximum Path Sum: " + MaximumPathSum.findMaximumPathSum(root));

  }

}

**Construction and Conversion**

### Flatten Binary Tree to Linked List

### Problem Statement 1

Given a binary tree, flatten it to a linked list in-place.

For example, given the following tree:

1

/ \

2 5

/ \ \

3 4 6

The flattened tree should look like:

1

\

2

\

3

\

4

\

5

\

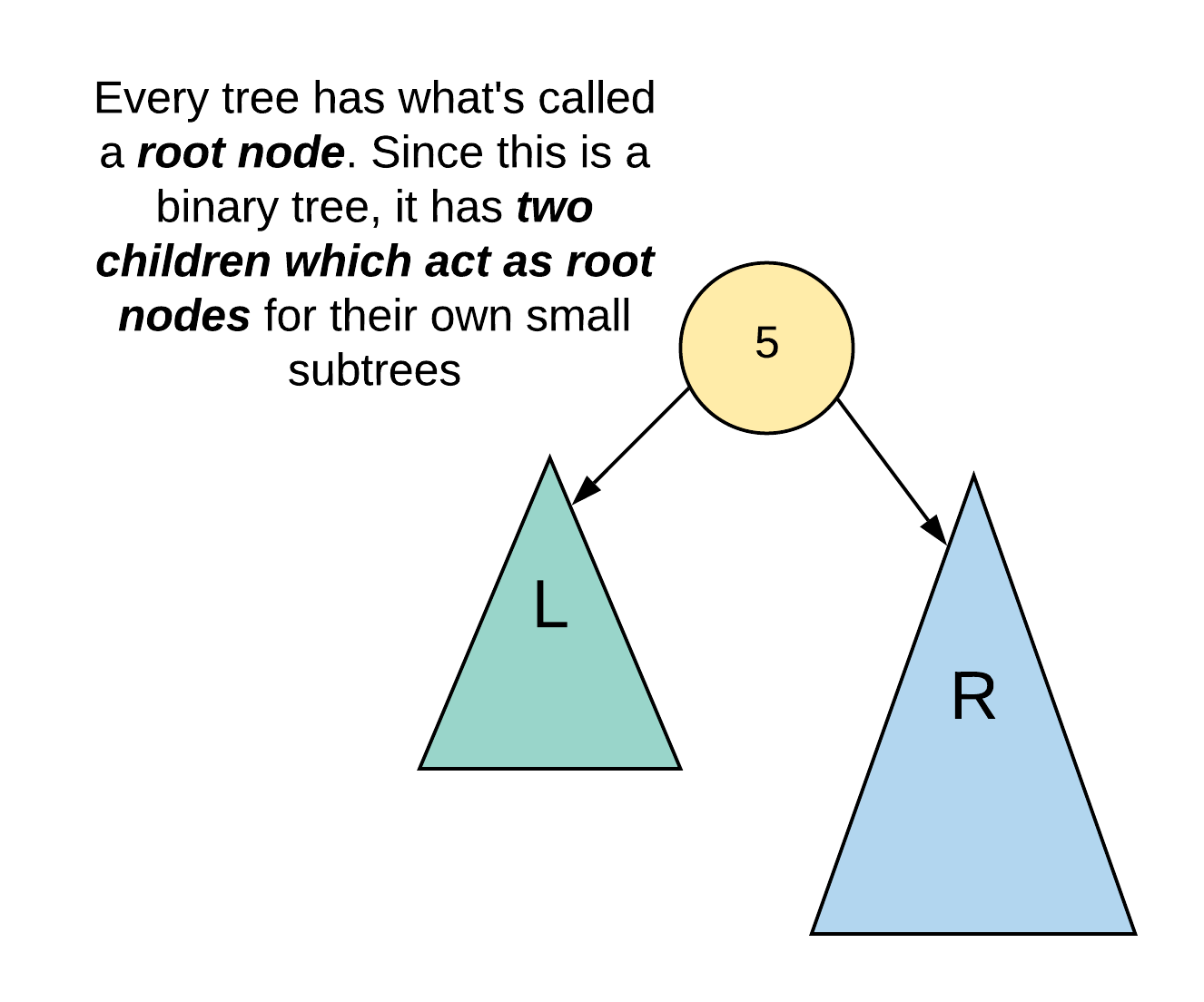
6

## **Solution**

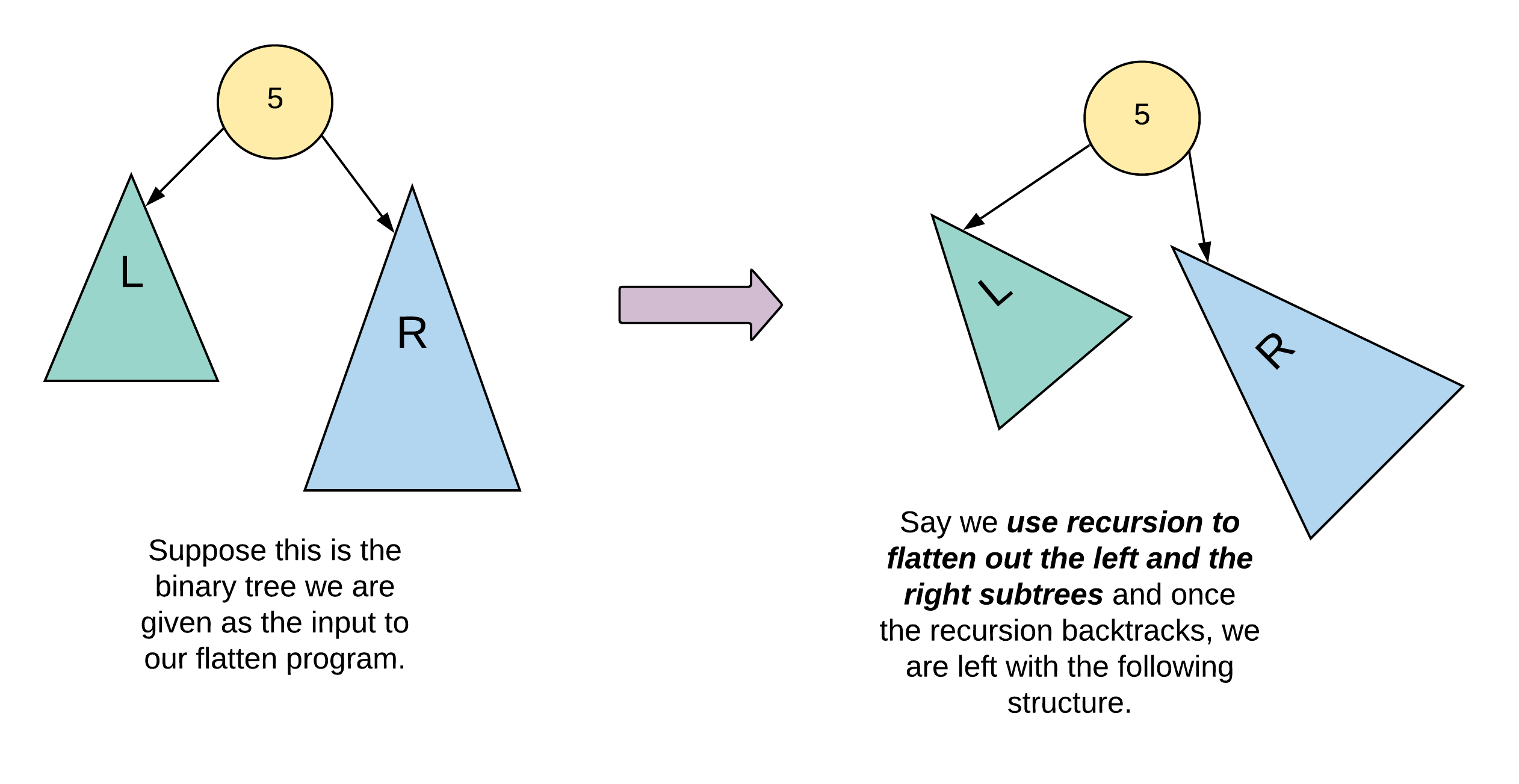
#### **Approach 1: Recursion**

**Intuition**

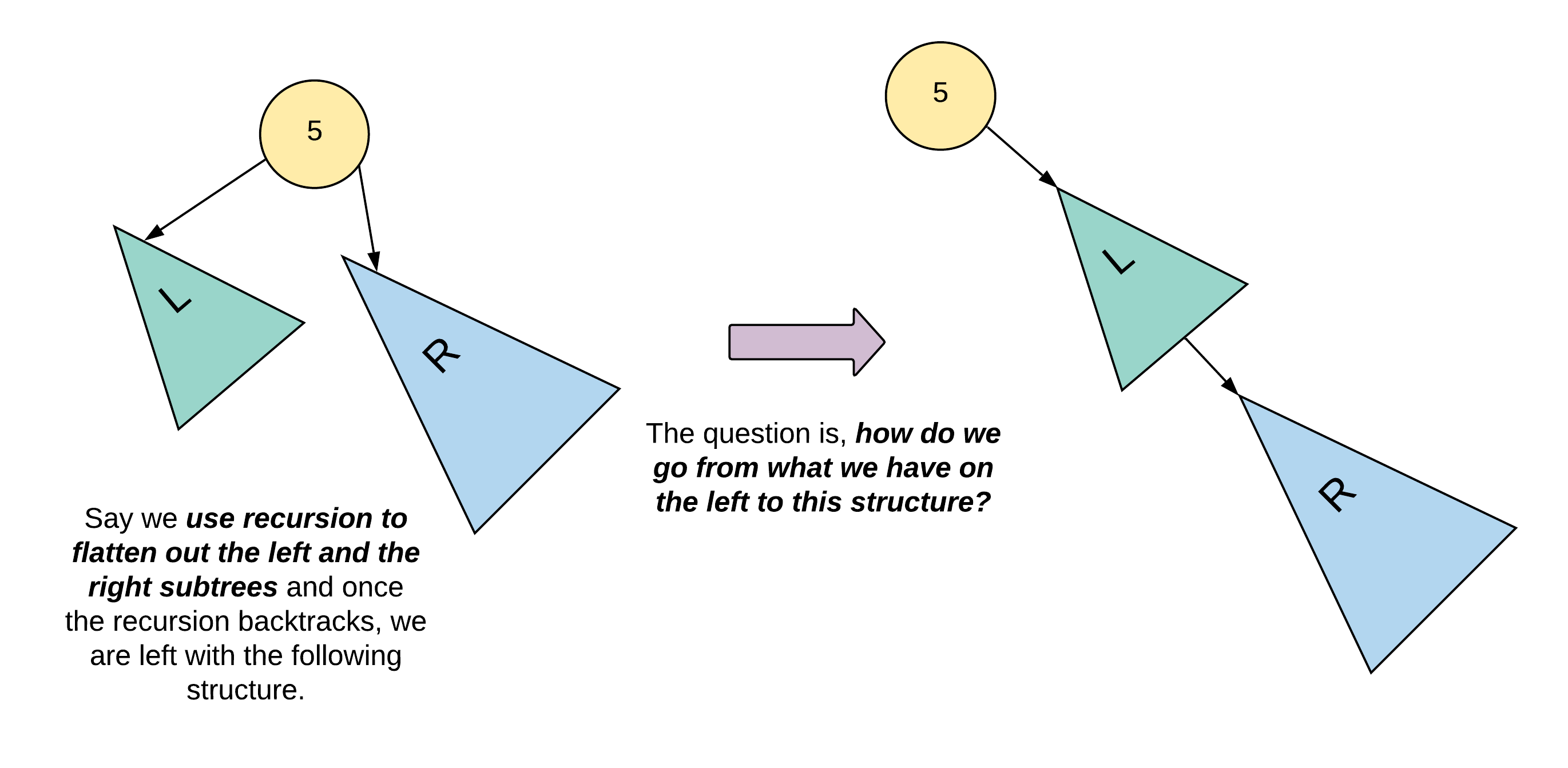
A common strategy for tree modification problems is recursion. A tree is a recursive structure. Every node gets to be a root node of some tree and that tree further has a bunch of smaller subtrees each with their own root nodes. So, when it comes to problems where the structure of the tree has to be modified or we have to traverse the tree in general, recursion is one of the top approaches that comes to mind simply because it's easy enough to code up and also is very intuitive to understand. Let's quickly look at a binary tree structure and then we will talk about how we can solve this problem using a recursive strategy.



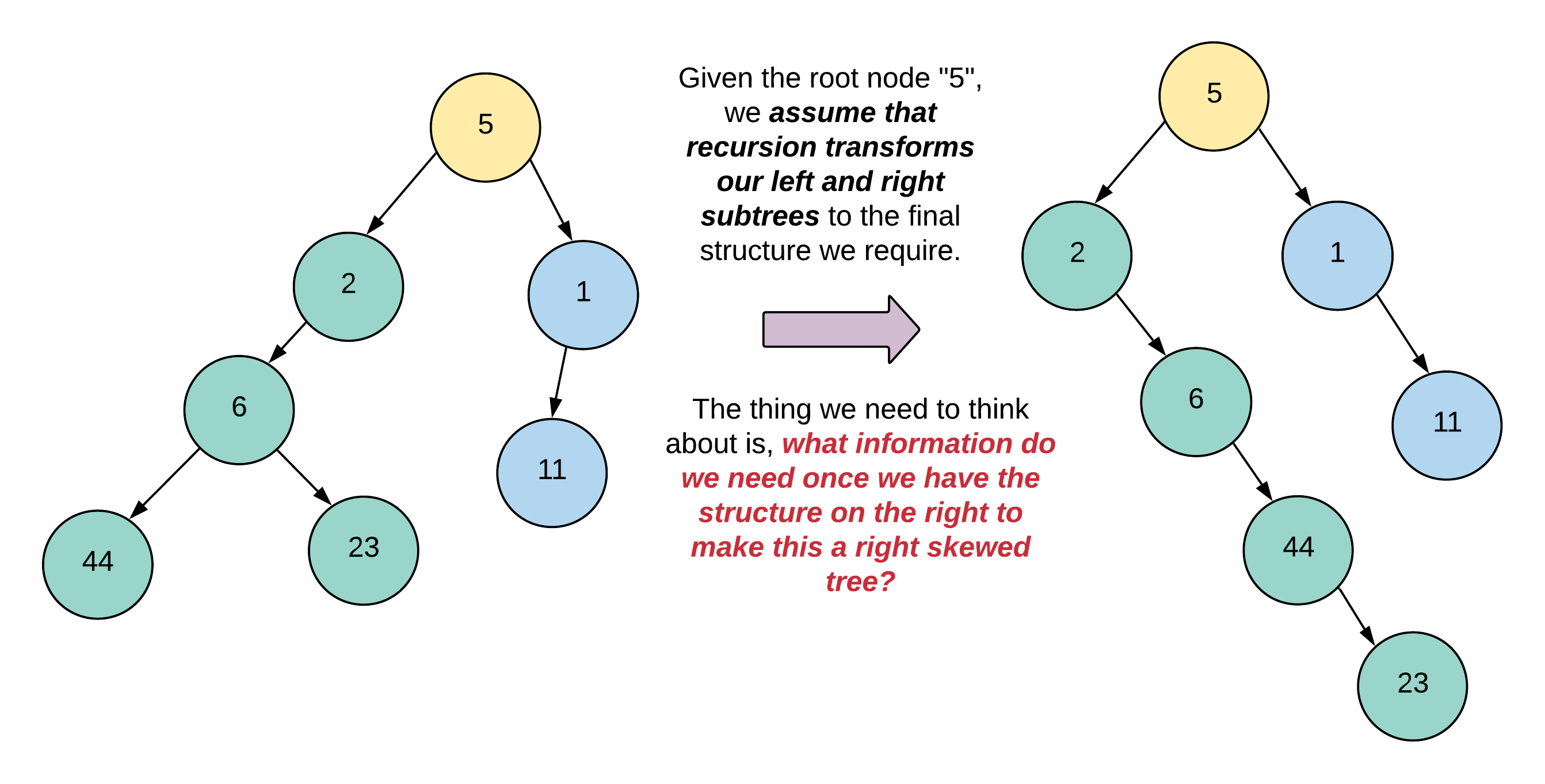
We haven't drawn the entire tree in the above image so as to build that intuition for that recursive solution. The main idea behind a recursive solution is that we use the solutions for subproblems to solve an uber level problem. In the case of a tree, the subtrees are essentially our subproblems. So, a recursive solution for this problem is essentially based on the idea that assuming we have already transformed the left and the right halves of a given root node, how do we establish or modify the necessary connections so that we get a right skewed tree overall. Let's look at what this means diagrammatically to have a better understanding.



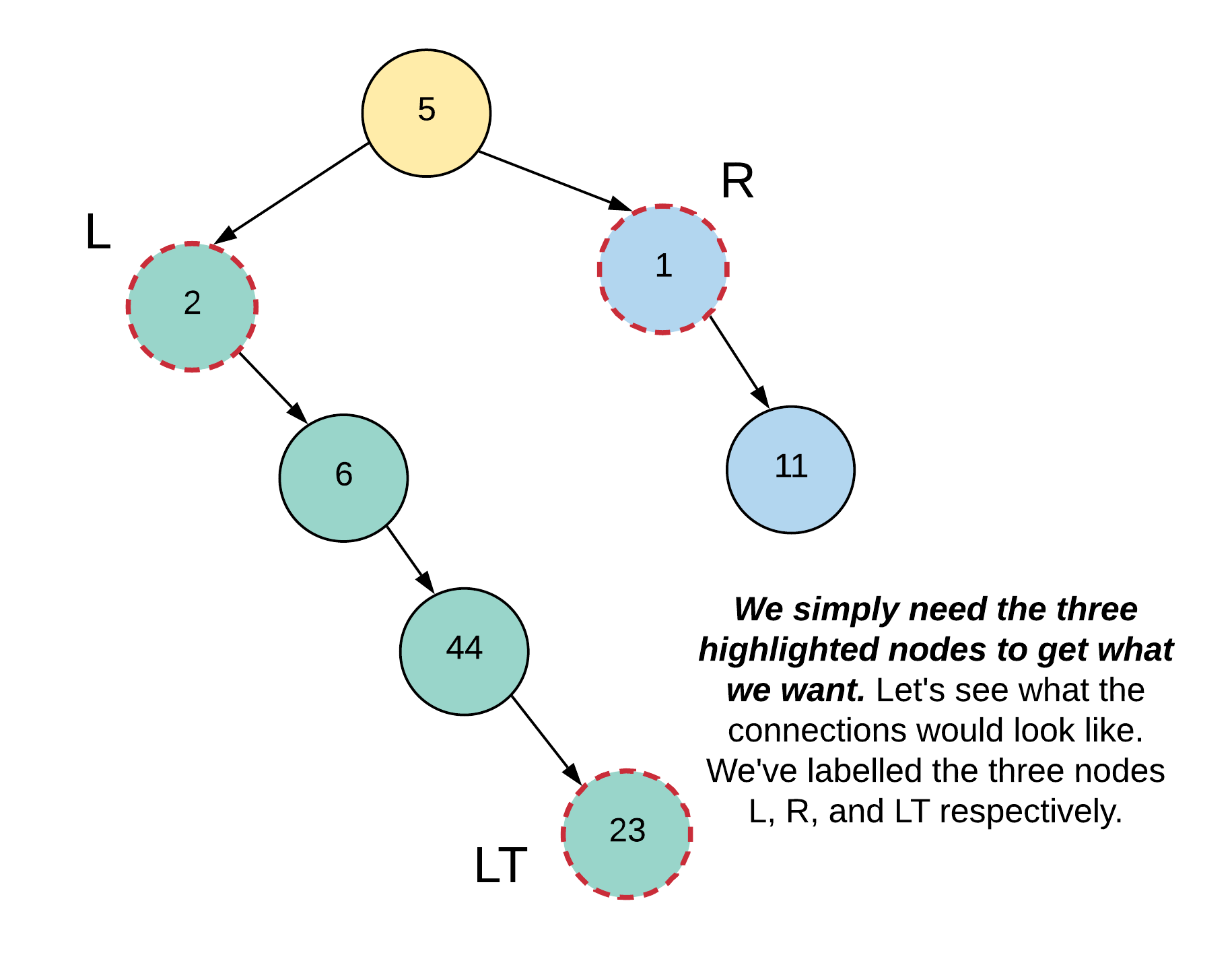
In the above figure, we simply showcase the root node of a tree and its left and right subtrees. A great way to think about recursion here is that we "suppose" that recursion does all the hard work for us and flattens out the left and the right subtrees as shown in the figure. What is it that we have to do then to get our final result? We need a right skewed tree, right? Well, we simply have to shuffle around some pointers to get our final result as shown below.

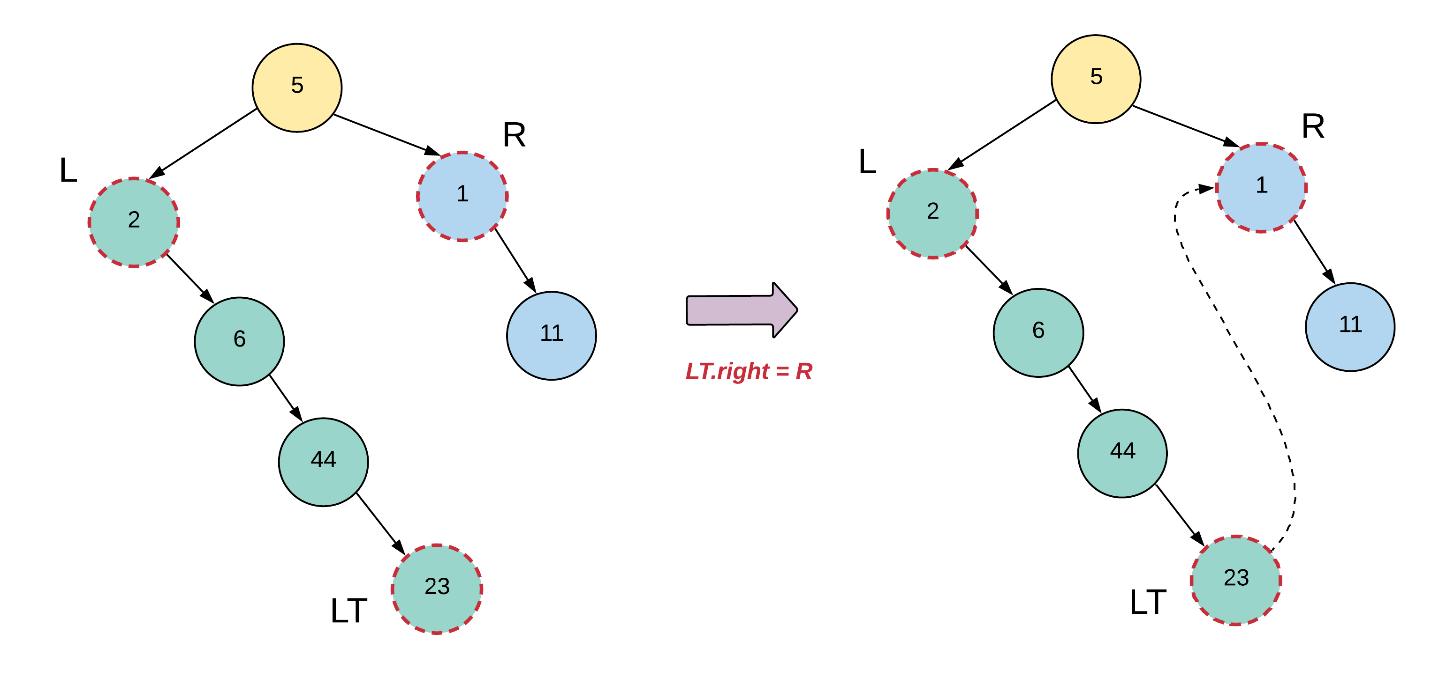


Let's dive a bit deeper and take a look at an exact tree now to see what exact connections we'll need to establish exactly for this work.

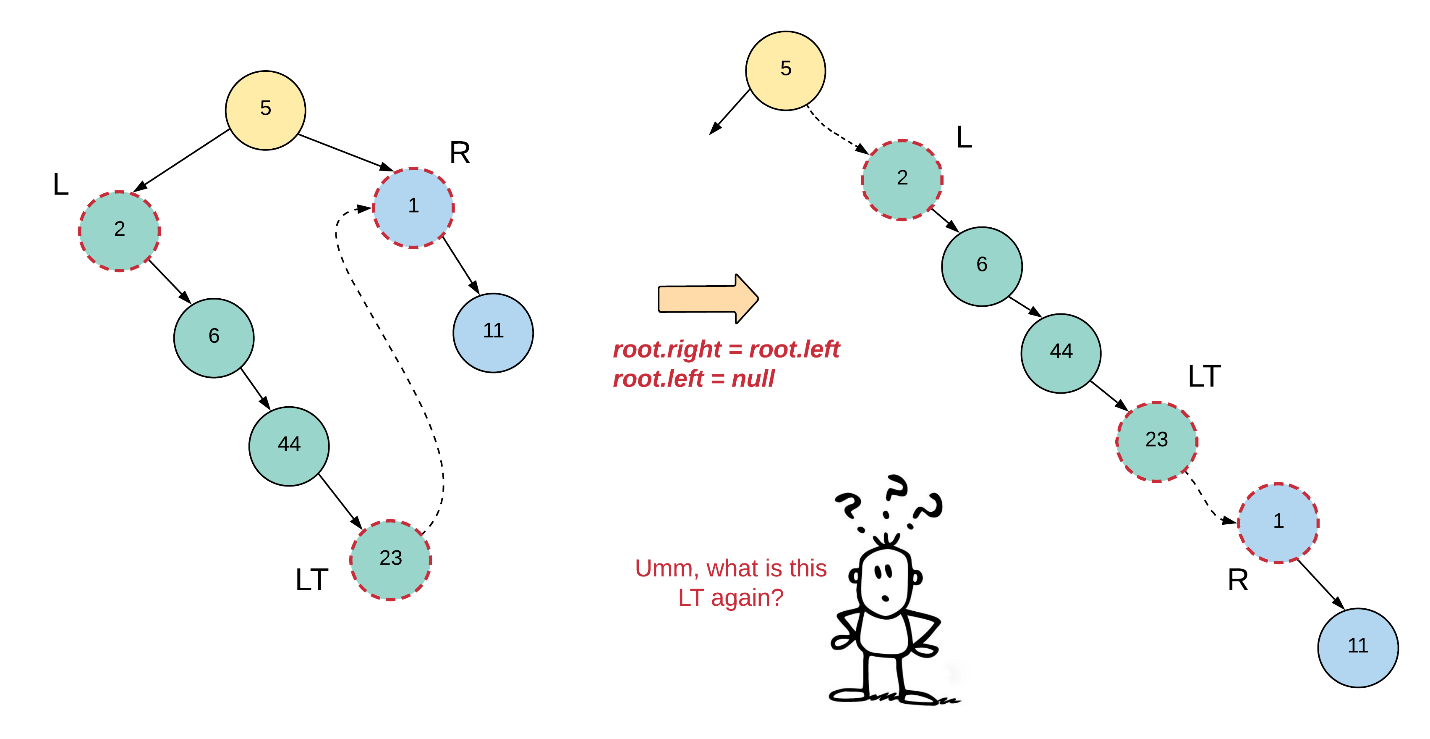


The figure shown below essentially highlights the exact set of nodes that are required for re-wiring the tree to our final right skewed tree. We've marked them "L" for left, "R" for right, and LT for "left tail". We'll get to the reason as to why we call that third node "left tail", later.





And finally, let's see how our tree looks like once we rewire the "L" and the "R" nodes properly as well.



Yeah, we haven't exactly explained what this left tail means. So, if you go back a few figures to the point where we had the left and the right subtrees all flattened out and we hadn't done any pointer manipulation yet, you'll notice that each subtree actually looks like a Linked List. Every linked list has a head node and in this case, we also need the tail node. Once recursion does the hard work for us and flattens out the subtrees, we will essentially get two linked lists and we need the tail end of the left one to attach it to the right one. Let's see what all information we will need in our recursive function at a given node.

* *\_node\_* = The current node
* *\_leftChild\_* = the left child of our current node
* *\_rightChild\_* = the right child of our current node
* *\_leftTail\_* = The tail node of the flattened out left subtree
* *\_rightTail\_* = The tail node of the fully formed tree rooted at *\_node\_*. This information is needed by the parent recursive calls since the tree rooted at the current node can be some other's node's left subtree or right subtree.

We have all the information available with us except the tail nodes. That's something that our recursion function will have to return. So, a recursion call for a given node will return the tail node of the flattened out tree. In our example, we will return the node 11 as the tail end of our final flattened out tree.

**Algorithm**

1. We'll have a separate function for flattening out the tree since the main function provided in the problem isn't supposed to return anything and our algorithm will return the tail node of the flattened out tree.
2. For a given node, we will recursively flatten out the left and the right subtrees and store their corresponding tail nodes in leftTail and rightTail respectively.
3. Next, we will make the following connections (only if there is a left child for the current node, else the leftTail would be null)
4. leftTail.right = node.right
5. node.right = node.left
6. node.left = None

1. Next we have to return the tail of the final, flattened out tree rooted at node. So, if the node has a right child, then we will return the rightTail, else, we'll return the leftTail.

### Code [#](https://www.educative.io/courses/grokking-the-coding-interview/xVPgnOvWVJq#code)

/\*

public class TreeNode {

    public int data;

    public TreeNode left;

    public TreeNode right;

    public TreeNode(int data) {

        this.data = data;

        left = right = null;

    }

}

 \*/

public class TreeFlatten {

    public static void flatten(TreeNode root) {

        flattenTree(root);

    }

    private static TreeNode flattenTree(TreeNode root) {

        if (root == null)

            return null;

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

            return root;

        TreeNode leftTail = flattenTree(root.left);

        TreeNode rightTail = flattenTree(root.right);

        if (leftTail != null) {

            leftTail.right = root.right;

            root.right = root.left;

            root.left = null;

        }

        return rightTail != null ? rightTail : leftTail;

    }

    public static void printTree(TreeNode root) {

        if (root != null) {

            printTree(root.left);

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

            printTree(root.right);

        }

    }

    public static void main(String[] args) {

        TreeNode root = new TreeNode(1);

        root.left = new TreeNode(2);

        root.left.left = new TreeNode(3);

        root.left.right = new TreeNode(4);

        root.right = new TreeNode(5);

        root.right.right = new TreeNode(6);

        printTree(root);

        flatten(root);

        System.out.println();

        printTree(root);

    }

}

**Complexity Analysis**

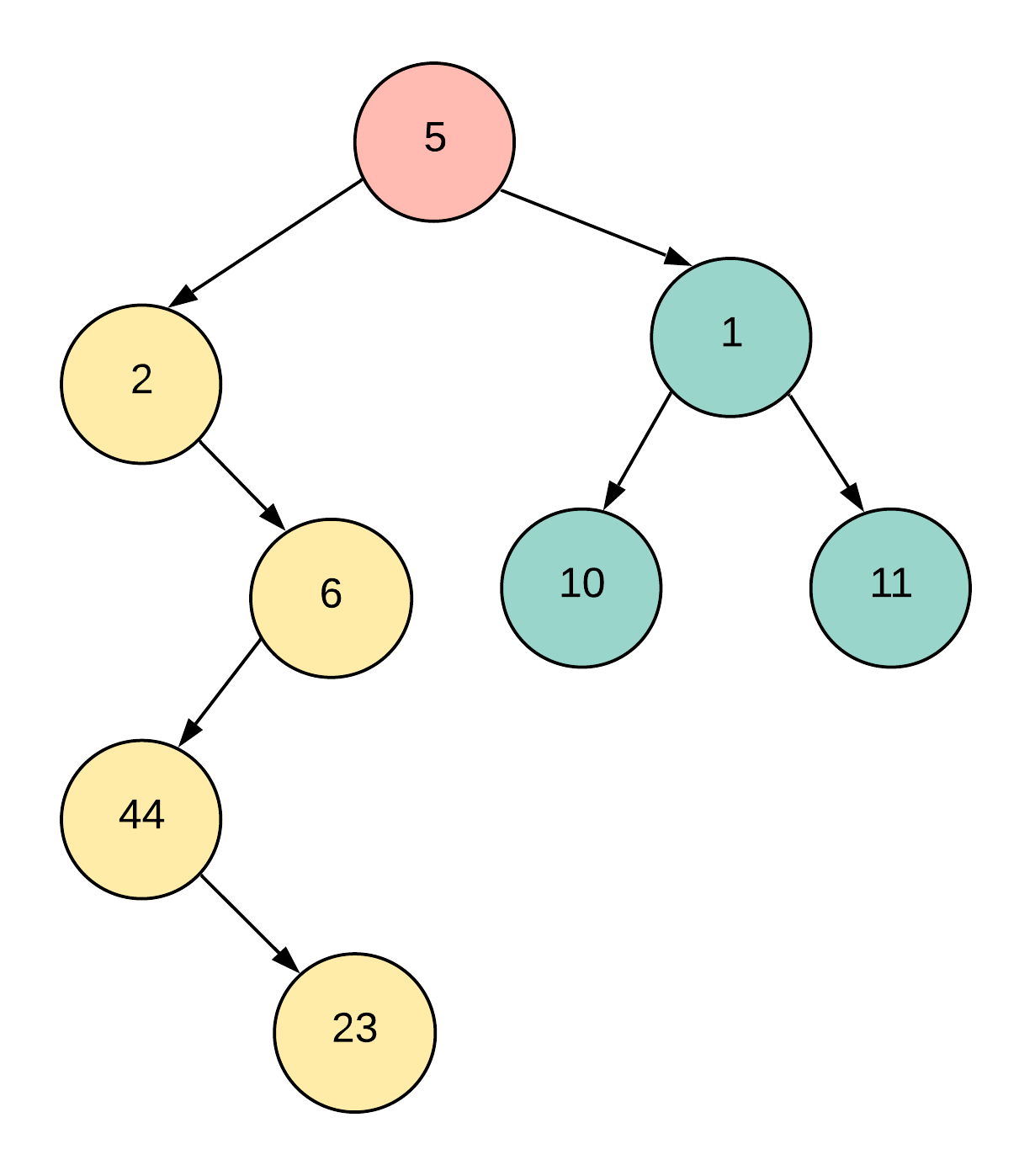
* Time Complexity: O(N)*O*(*N*) since we process each node of the tree exactly once.
* Space Complexity: O(N)*O*(*N*) which is occupied by the recursion stack. The problem statement doesn't mention anything about the tree being balanced or not and hence, the tree could be e.g. left skewed and in that case the longest branch (and hence the number of nodes in the recursion stack) would be N*N*

#### **Approach 2: O(1) Iterative Solution**

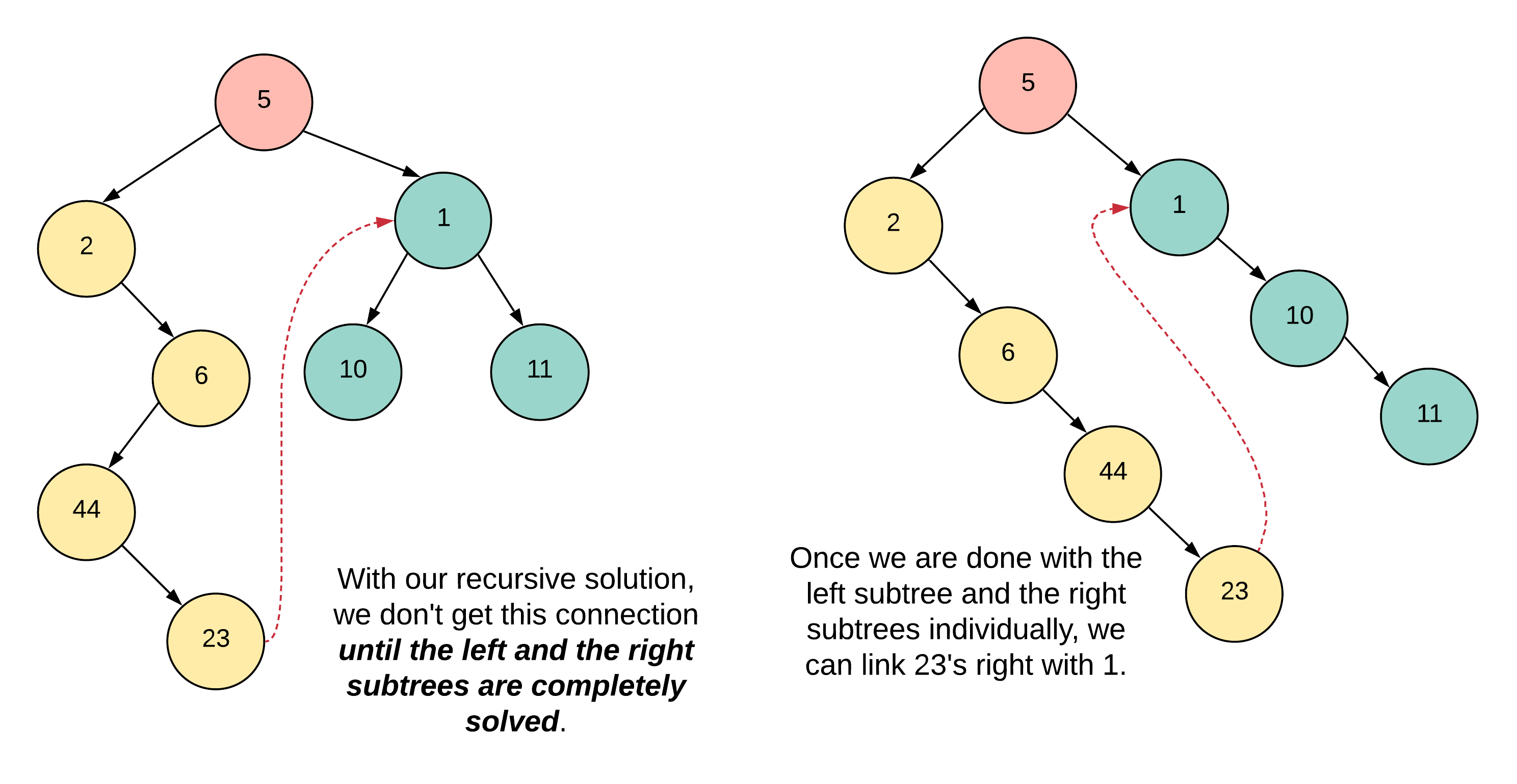
**Intuition**

We'll get to the intuition for this approach in a bit, but first let's talk about the motivation. For any kind of tree traversal, we always have the easiest of solutions which is based on recursion. Next, we have a custom stack based iterative version of the same solution. Finally, we want a tree traversal that doesn't use any kind of additional space at all. There is a well known tree traversal out there that doesn't use any additional space at all. It's known as Morris Traversal. Our solution is based off of the same ideology, but Morris Traversal is not a pre-requisite here.

To understand what's difference between the nodes processing of this approach and basic recursion, let's look at a sample tree.



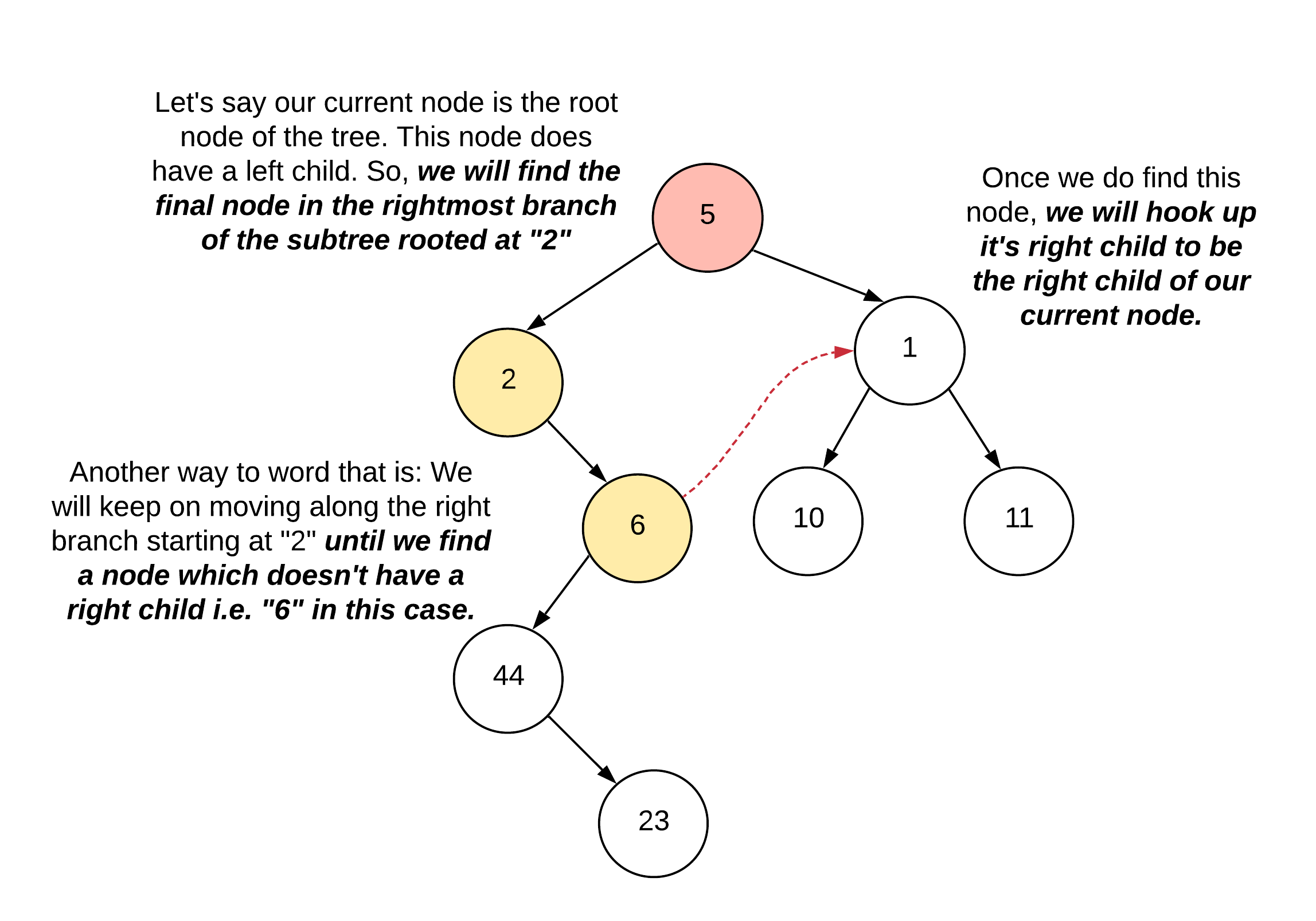
With recursion, we only re-wire the connections for the "current node" once we are already done processing the left and the right subtrees completely. Let's see what that looks like in a figure.



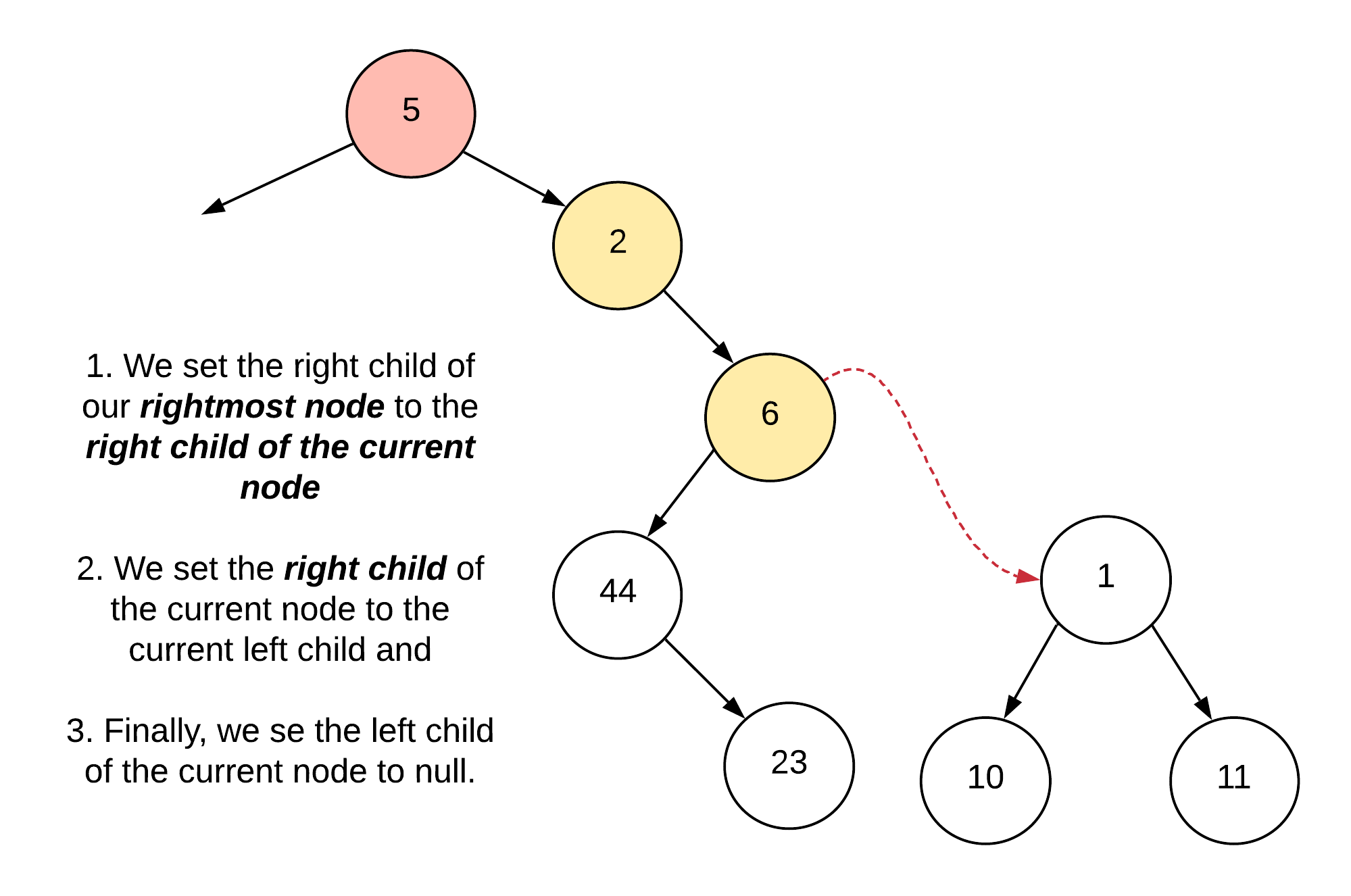
However, the postponing of rewiring of connections on the current node until the left subtree is done, is basically what recursion is. Recursion is all about postponing decisions until something else is completed. In order for us to be able to postpone stuff, we need to use the stack. However, in our current approach we want to get rid of the stack altogether. So, we will have to come up with a greedy way that will be costlier in terms of time, but will be space efficient in achieving the same results.

For a current node, we will check if it has a left child or not. If it does, we will find the last node in the rightmost branch of the subtree rooted at this left child. Once we find this "rightmost" node, we will hook it up with the right child of the current node.

Let's look at this idea on our current sample tree.



This might not make a lot of sense just yet. But, bear with me and read on. Let's see what connections we need to establish or shuffle once we find that "rightmost node". We are highlighting "rightmost" here because technically, even without knowing this approach, the node 23 would have made much more sense here, right? Instead, we are doing some vodoo with the node 6. God knows why!



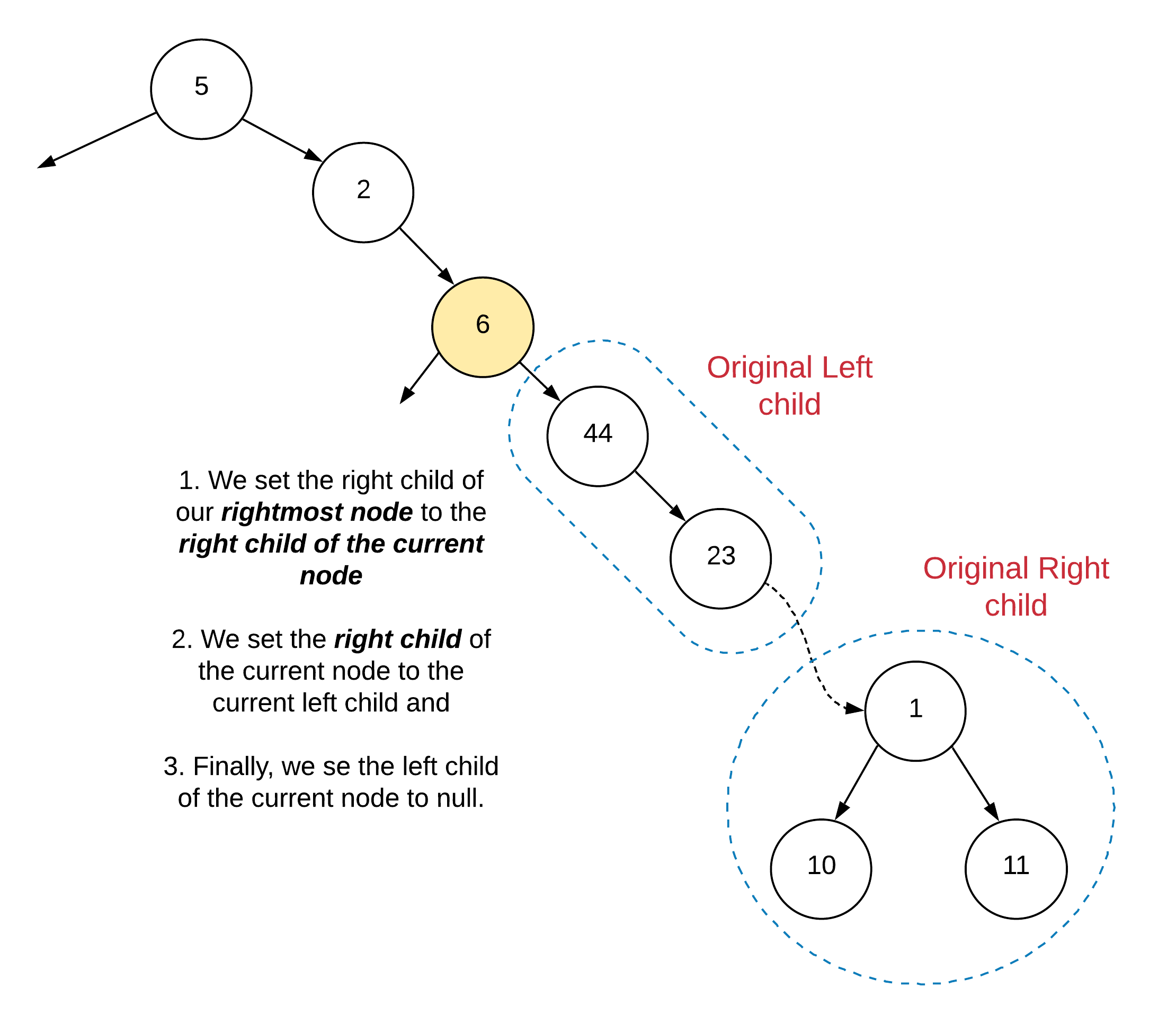
As mentioned in the previous paragraph, this figure would make much more sense if we had just found out the node 23, and set its right child to 1 instead of doing all this with 6. Why did we do that you might ask? Well, it's an optimization of sorts. To find the actual "rightmost" node of subtree, we might have to potentially traverse most of that subtree. Like in our example. To actually get to the node 23, we would have had to traverse all of the nodes: 2, 6, 44, 23. Instead, we simply stop at the node 6. We'll see why that also achieves our final purpose. For now, let's move on.

By doing the following operation for every node, we are simply try ing to move stuff to the right hand side one step at a time. The reason we used the node 6 in the above example and not 23 is the very reason we called this approach somewhat greedy.

Processing of the node 2 is simple since it doesn't have a left child at all. So we have nothing to do here. Let's come over to the node 6 since this is where things get interesting and start to make sense. We'll again use the same logic as before.

For a current node, we will check if it has a left child or not. If it does, we will find the last node in the rightmost branch of the subtree rooted at this left child. Once we find this "rightmost" node, we will hook it up with the right child of the current node.

As we can clearly see from the previous figures, the rightmost node here would be 23. So, let's look at the tree after we are done rewiring the connections.



Now this looks just like the tree after the recursion would have completed on the left subtree and we rewired the connections, right? Exactly!. The reason we stopped at the first rightmost node with no right child was because we would eventually end up rightyfying all the subtrees through that connection. Even though before we didn't hook up the node 23, we were able to do it when we arrived at the node 6 here.

**Algorithm**

1. So basically, this is going to be a super short algorithm and a short-er implementation :)
2. We use a pointer for traversing the nodes of our tree starting from the root. We have a loop that keeps going until the node pointer becomes null which is when we would be done processing the entire tree.
3. For every node we check if it has a left child or not. If it doesn't we simply move on to the right hand side i.e.

node = node.right

1. If the node does have a left child, we find the first node on the rightmost branch of the left subtree which doesn't have a right child i.e. the almost rightmost node.
2. rightmost = node.left
3. while rightmost != null:

rightmost = rightmost.right

1. Once we find this rightmost node, we rewire the connections as explained in the intuition section.
2. rightmost.right = node.right
3. node.right = node.left

node.left = null

1. And we move on to the right node to continue processing of our tree.

### Code [#](https://www.educative.io/courses/grokking-the-coding-interview/xVPgnOvWVJq#code)

public class TreeFlattenIterative {

    public static void flatten(TreeNode root) {

        if (root == null)

            return;

        TreeNode node = root;

        while (node != null) {

            // If the node has a left child

            if (node.left != null) {

                // Find the rightmost node

                TreeNode rightmost = node.left;

                while (rightmost.right != null) {

                    rightmost = rightmost.right;

                }

                // rewire the connections

                rightmost.right = node.right;

                node.right = node.left;

                node.left = null;

            }

            // move on to the right side of the tree

            node = node.right;

        }

    }

    public static void printTree(TreeNode root) {

        if (root != null) {

            printTree(root.left);

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

            printTree(root.right);

        }

    }

    public static void main(String[] args) {

        TreeNode root = new TreeNode(1);

        root.left = new TreeNode(2);

        root.left.left = new TreeNode(3);

        root.left.right = new TreeNode(4);

        root.right = new TreeNode(5);

        root.right.right = new TreeNode(6);

        printTree(root);

        flatten(root);

        System.out.println();

        printTree(root);

    }

}

### Symmetric Tree

### Problem Statement 2

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).

For example, this binary tree [1,2,2,3,4,4,3] is symmetric:

1

/ \

2 2

/ \ / \

3 4 4 3

But the following [1,2,2,null,3,null,3] is not:

1

/ \

2 2

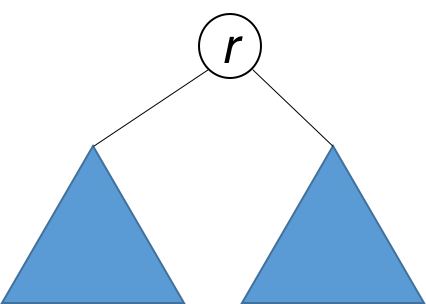
\ \

3 3

## **Solution**

#### **Approach 1: Recursive**

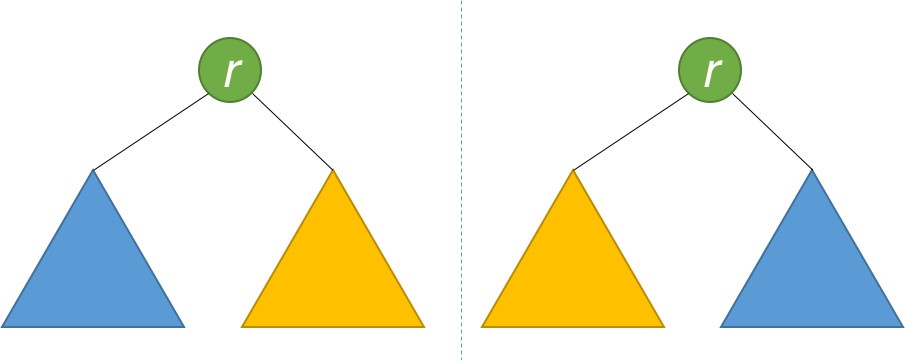
A tree is symmetric if the left subtree is a mirror reflection of the right subtree.



Therefore, the question is: when are two trees a mirror reflection of each other?

Two trees are a mirror reflection of each other if:

1. Their two roots have the same value.
2. The right subtree of each tree is a mirror reflection of the left subtree of the other tree.



This is like a person looking at a mirror. The reflection in the mirror has the same head, but the reflection's right arm corresponds to the actual person's left arm, and vice versa.

The explanation above translates naturally to a recursive function as follows.

### Code [#](https://www.educative.io/courses/grokking-the-coding-interview/xVPgnOvWVJq#code)

public class SymmetricTree {

    public static boolean isSymmetry(TreeNode root) {

        return isMirror(root,root);

    }

    private static boolean isMirror(TreeNode root1,TreeNode root2) {

        if (root1 == null && root2 == null)

            return true;

        if (root1 == null || root2 == null)

            return false;

        return root1.data == root2.data && isMirror(root1.left,root2.right)

                && isMirror(root1.right,root2.left);

    }

    public static void printTree(TreeNode root) {

        if (root != null) {

            printTree(root.left);

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

            printTree(root.right);

        }

    }

    public static void main(String[] args) {

        TreeNode root = new TreeNode(1);

        root.left = new TreeNode(2);

        root.left.left = new TreeNode(3);

        root.left.right = new TreeNode(4);

        root.right = new TreeNode(2);

        root.right.left = new TreeNode(4);

        root.right.right = new TreeNode(3);

        printTree(root);

        if (isSymmetry(root)) {

            System.out.println("symmetry");

        }

        else {

            System.out.println("not a symmetry");

        }

    }

}

**Complexity Analysis**

* Time complexity : O(n)*O*(*n*). Because we traverse the entire input tree once, the total run time is O(n)*O*(*n*), where n*n* is the total number of nodes in the tree.
* Space complexity : The number of recursive calls is bound by the height of the tree. In the worst case, the tree is linear and the height is in O(n)*O*(*n*). Therefore, space complexity due to recursive calls on the stack is O(n)*O*(*n*) in the worst case.

#### **Approach 2: Iterative**

Instead of recursion, we can also use iteration with the aid of a queue. Each two consecutive nodes in the queue should be equal, and their subtrees a mirror of each other. Initially, the queue contains root and root. Then the algorithm works similarly to BFS, with some key differences. Each time, two nodes are extracted and their values compared. Then, the right and left children of the two nodes are inserted in the queue in opposite order. The algorithm is done when either the queue is empty, or we detect that the tree is not symmetric (i.e. we pull out two consecutive nodes from the queue that are unequal).

public static boolean isSymmetryIterative(TreeNode root) {

        Queue<TreeNode> queue = new LinkedList<>();

        queue.add(root);

        queue.add(root);

        while(!queue.isEmpty()) {

            TreeNode node1 = queue.poll();

            TreeNode node2 = queue.poll();

            if (node1 == null && node2 == null)

                continue;

            if (node1 == null || node2 == null)

                return false;

            if (node1.data != node2.data)

                return false;

            queue.add(node1.left);

            queue.add(node2.right);

            queue.add(node1.right);

            queue.add(node2.left);

        }

        return true;

    }

}

**Complexity Analysis**

* Time complexity : O(n)*O*(*n*). Because we traverse the entire input tree once, the total run time is O(n)*O*(*n*), where n*n* is the total number of nodes in the tree.
* Space complexity : There is additional space required for the search queue. In the worst case, we have to insert O(n)*O*(*n*) nodes in the queue. Therefore, space complexity is O(n)*O*(*n*).

### Convert Sorted List to Binary Search Tree

### Problem Statement 3

Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST.

For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of *every* node never differ by more than 1.

**Example:**

Given the sorted linked list: [-10,-3,0,5,9],

One possible answer is: [0,-3,9,-10,null,5], which represents the following height balanced BST:

0

/ \

-3 9

/ /

-10 5

## **Solution**

#### **Approach 1: Recursion**

**Intuition**

The important condition that we have to adhere to in this problem is that we have to create a height balanced binary search tree using the set of nodes given to us in the form of a linked list. The good thing is that the nodes in the linked list are sorted in ascending order.

As we know, a binary search tree is essentially a rooted binary tree with a very special property or relationship amongst its nodes. For a given node of the binary search tree, it's value must be \ge≥ the value of all the nodes in the left subtree and \le≤ the value of all the nodes in the right subtree. Since a binary tree has a recursive substructure, so does a BST i.e. all the subtrees are binary search trees in themselves.

The main idea in this approach and the next is that:

the middle element of the given list would form the root of the binary search tree. All the elements to the left of the middle element would form the left subtree recursively. Similarly, all the elements to the right of the middle element will form the right subtree of the binary search tree. This would ensure the height balance required in the resulting binary search tree.

**Algorithm**

1. Since we are given a linked list and not an array, we don't really have access to the elements of the list using indexes. We want to know the middle element of the linked list.
2. We can use the two pointer approach for finding out the middle element of a linked list. Essentially, we have two pointers called slow\_ptr and fast\_ptr. The slow\_ptr moves one node at a time whereas the fast\_ptr moves two nodes at a time. By the time the fast\_ptr reaches the end of the linked list, the slow\_ptr would have reached the middle element of the linked list. For an even sized list, any of the two middle elements can act as the root of the BST.
3. Once we have the middle element of the linked list, we disconnect the portion of the list to the left of the middle element. The way we do this is by keeping a prev\_ptr as well which points to one node before the slow\_ptr i.e. prev\_ptr.next = slow\_ptr. For disconnecting the left portion we simply do prev\_ptr.next = None
4. We only need to pass the head of the linked list to the function that converts it to a height balances BST. So, we recurse on the left half of the linked list by passing the original head of the list and on the right half by passing slow\_ptr.next as the head.

/\*\*

 \* Definition for singly-linked list. public class ListNode { int val; ListNode next; ListNode(int

 \* x) { val = x; } }

 \*/

/\*\*

 \* Definition for a binary tree node. public class TreeNode { int val; TreeNode left; TreeNode

 \* right; TreeNode(int x) { val = x; } }

 \*/

class Solution {

  private ListNode findMiddleElement(ListNode head) {

    // The pointer used to disconnect the left half from the mid node.

    ListNode prevPtr = null;

    ListNode slowPtr = head;

    ListNode fastPtr = head;

    // Iterate until fastPr doesn't reach the end of the linked list.

    while (fastPtr != null && fastPtr.next != null) {

      prevPtr = slowPtr;

      slowPtr = slowPtr.next;

      fastPtr = fastPtr.next.next;

    }

    // Handling the case when slowPtr was equal to head.

    if (prevPtr != null) {

      prevPtr.next = null;

    }

    return slowPtr;

  }

  public TreeNode sortedListToBST(ListNode head) {

    // If the head doesn't exist, then the linked list is empty

    if (head == null) {

      return null;

    }

    // Find the middle element for the list.

    ListNode mid = this.findMiddleElement(head);

    // The mid becomes the root of the BST.

    TreeNode node = new TreeNode(mid.val);

    // Base case when there is just one element in the linked list

    if (head == mid) {

      return node;

    }

    // Recursively form balanced BSTs using the left and right halves of the original list.

    node.left = this.sortedListToBST(head);

    node.right = this.sortedListToBST(mid.next);

    return node;

  }

}

**Complexity Analysis**

* Time Complexity: O(N\log N)*O*(*N*log*N*). Suppose our linked list consists of N*N* elements. For every list we pass to our recursive function, we have to calculate the middle element for that list. For a list of size N*N*, it takes N / 2*N*/2 steps to find the middle element i.e. O(N)*O*(*N*) to find the mid. We do this for **every** half of the original linked list. From the looks of it, this seems to be an O(N^2)*O*(*N*2) algorithm. However, on closer analysis, it turns out to be a bit more efficient than O(N^2)*O*(*N*2).

Let's look at the number of operations that we have to perform on each of the halves of the linked list. As we mentioned earlier, it takes N/2*N*/2 steps to find the middle of a linked list with N*N* elements. After finding the middle element, we are left with two halves of size N / 2*N*/2 each. Then, we find the middle element for both of these halves and it would take a total of 2 \times N / 42×*N*/4 steps for that. And similarly for the smaller sublists that keep forming recursively. This would give us the following series of operations for a list of size N*N*.

\begin{aligned} \frac{N}{2} + 2 \cdot \frac{N}{4} + 4 \cdot \frac{N}{8} + 8 \cdot \frac{N}{16} \; \ldots \end{aligned}2*N*​+2⋅4*N*​+4⋅8*N*​+8⋅16*N*​…​

Essentially, this is done \log Nlog*N* times since we split the linked list in half every time. Hence, the above equation becomes:

\begin{aligned} &\sum\_{i = 1}^{\log N} 2^{i - 1} \cdot \frac{N}{2^i} \\ = \; &\sum\_{i = 1}^{\log N}\frac{N}{2} \\ = \; &\frac{N}{2} \; \log N \\ = \; &O(N\log N) \end{aligned}===​*i*=1∑log*N*​2*i*−1⋅2*iN*​*i*=1∑log*N*​2*N*​2*N*​log*NO*(*N*log*N*)​

* Space Complexity: O(\log N)*O*(log*N*). Since we are resorting to recursion, there is always the added space complexity of the recursion stack that comes into picture. This could have been O(N)*O*(*N*) for a skewed tree, but the question clearly states that we need to maintain the height balanced property. This ensures the height of the tree to be bounded by O(\log N)*O*(log*N*). Hence, the space complexity is O(\log N)*O*(log*N*).

The main problem with the above solution seems to be the middle element computation. That takes up a lot of unnecessary time and this is due to the nature of the linked list data structure.

#### **Approach 2: Inorder Simulation**

**Intuition**

As we know, there are three different types of traversals for a binary tree:

* Inorder
* Preorder and
* Postorder traversals.

The inorder traversal on a binary search tree leads to a very interesting outcome.

Elements processed in the inorder fashion on a binary search tree turn out to be sorted in ascending order.

The approach listed here make use of this idea to formulate the construction of a binary search tree. The reason we are able to use this idea in this problem is because we are given a sorted linked list initially.

Before looking at the algorithm, let us look at how the inorder traversal actually leads to a sorted order of nodes' values.

The critical idea based on the inorder traversal that we will exploit to solve this problem, is:

We know that the leftmost element in the inorder traversal has to be the head of our given linked list. Similarly, the next element in the inorder traversal will be the second element in the linked list and so on. This is made possible because the initial list given to us is sorted in ascending order.

Now that we have an idea about the relationship between the inorder traversal of a binary search tree and the numbers being sorted in ascending order, let's get to the algorithm.

**Algorithm**

Let's quickly look at a pseudo-code to make the algorithm simple to understand.

➔ function formBst(start, end)

➔ mid = (start + end) / 2

➔ formBst(start, mid - 1)

➔

➔ TreeNode(head.val)

➔ head = head.next

➔

➔ formBst(mid + 1, end)

➔

1. Iterate over the linked list to find out it's length. We will make use of two different pointer variables here to mark the beginning and the end of the list. Let's call them start and end with their initial values being 0 and length - 1 respectively.
2. Remember, we have to simulate the inorder traversal here. We can find out the middle element by using (start + end) / 2. Note that we don't really find out the middle node of the linked list. We just have a variable telling us the index of the middle element. We simply need this to make recursive calls on the two halves.
3. Recurse on the left half by using start, mid - 1 as the starting and ending points.
4. The invariance that we maintain in this algorithm is that whenever we are done building the left half of the BST, the head pointer in the linked list will point to the root node or the middle node (which becomes the root). So, we simply use the current value pointed to by head as the root node and progress the head node by once i.e. head = head.next
5. We recurse on the right hand side using mid + 1, end as the starting and ending points.

class Solution {

  private ListNode head;

  private int findSize(ListNode head) {

    ListNode ptr = head;

    int c = 0;

    while (ptr != null) {

      ptr = ptr.next;

      c += 1;

    }

    return c;

  }

  private TreeNode convertListToBST(int l, int r) {

    // Invalid case

    if (l > r) {

      return null;

    }

    int mid = (l + r) / 2;

    // First step of simulated inorder traversal. Recursively form

    // the left half

    TreeNode left = this.convertListToBST(l, mid - 1);

    // Once left half is traversed, process the current node

    TreeNode node = new TreeNode(this.head.val);

    node.left = left;

    // Maintain the invariance mentioned in the algorithm

    this.head = this.head.next;

    // Recurse on the right hand side and form BST out of them

    node.right = this.convertListToBST(mid + 1, r);

    return node;

  }

  public TreeNode sortedListToBST(ListNode head) {

    // Get the size of the linked list first

    int size = this.findSize(head);

    this.head = head;

    // Form the BST now that we know the size

    return convertListToBST(0, size - 1);

  }

}

**Complexity Analysis**

* Time Complexity: The time complexity is still O(N)*O*(*N*) since we still have to process each of the nodes in the linked list once and form corresponding BST nodes.
* Space Complexity: O(\log N)*O*(log*N*) since now the only extra space is used by the recursion stack and since we are building a height balanced BST, the height is bounded by \log Nlog*N*.

### Convert Sorted Array to BST

### Problem Statement 4

Given an array where elements are sorted in ascending order, convert it to a height balanced BST.

For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of every node never differ by more than 1.

**Example:**

Given the sorted array: [-10,-3,0,5,9],

One possible answer is: [0,-3,9,-10,null,5], which represents the following height balanced BST:

0

/ \

-3 9

/ /

-10 5

## **Solution**

#### **How to Traverse the Tree. DFS: Preorder, Inorder, Postorder; BFS.**

There are two general strategies to traverse a tree:

* Depth First Search (DFS)

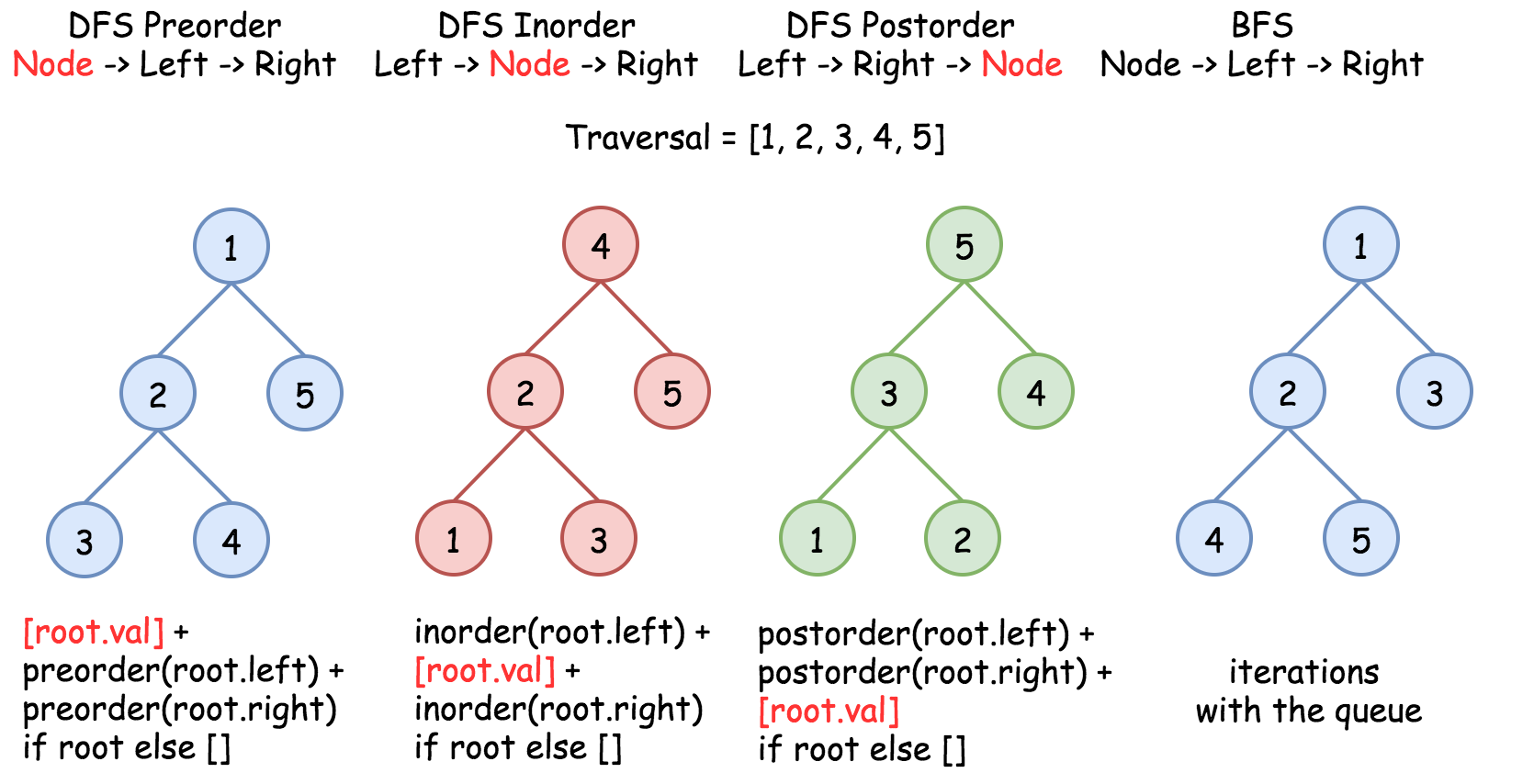
In this strategy, we adopt the depth as the priority, so that one would start from a root and reach all the way down to certain leaf, and then back to root to reach another branch.

The DFS strategy can further be distinguished as preorder, inorder, and postorder depending on the relative order among the root node, left node and right node.

* Breadth First Search (BFS)

We scan through the tree level by level, following the order of height, from top to bottom. The nodes on higher level would be visited before the ones with lower levels.

On the following figure the nodes are enumerated in the order you visit them, please follow 1-2-3-4-5 to compare different strategies.



#### **Construct BST from Inorder Traversal: Why the Solution is Not Unique**

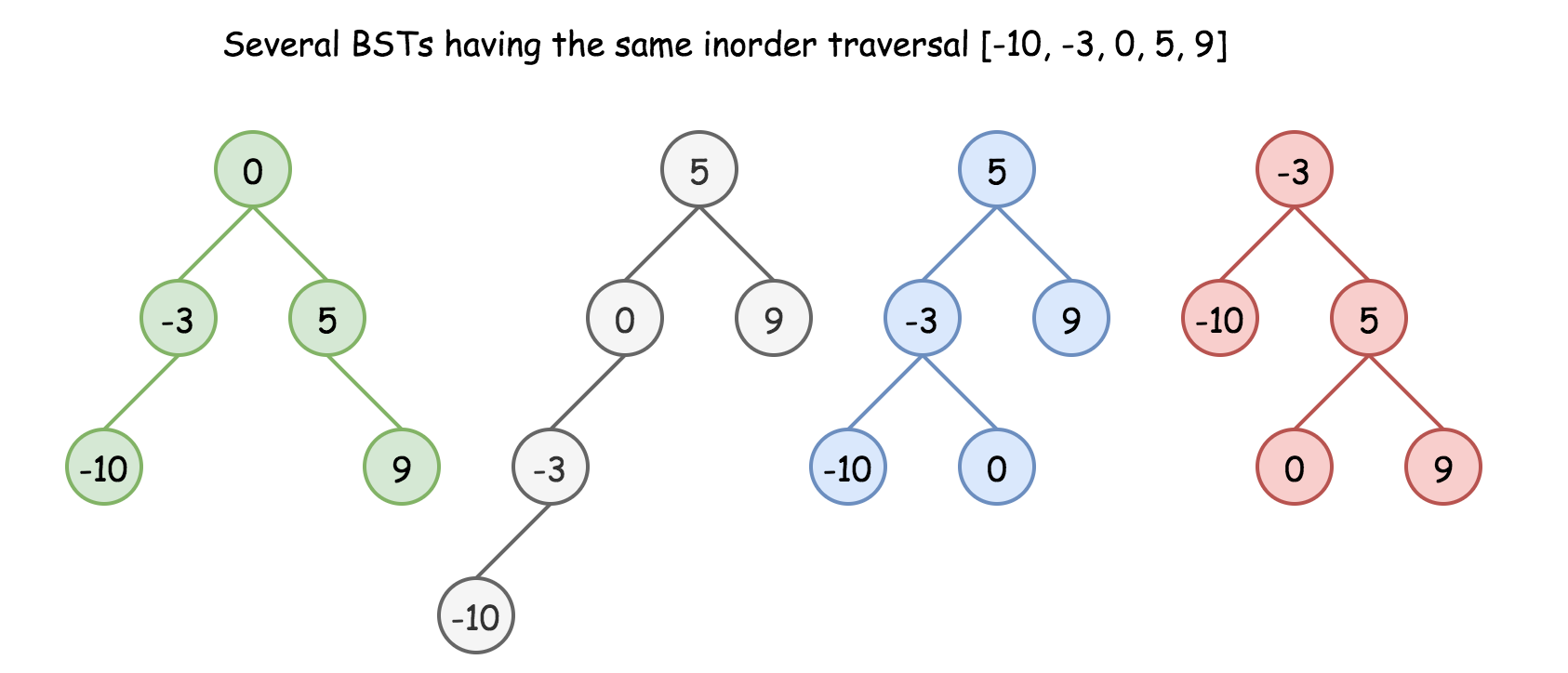
It's known that [inorder traversal of BST is an array sorted in the ascending order](https://leetcode.com/articles/delete-node-in-a-bst/).

Having sorted array as an input, we could rewrite the problem as Construct Binary Search Tree from Inorder Traversal.

Does this problem have a unique solution, i.e. could inorder traversal be used as a unique identifier to encore/decode BST? The answer is no.

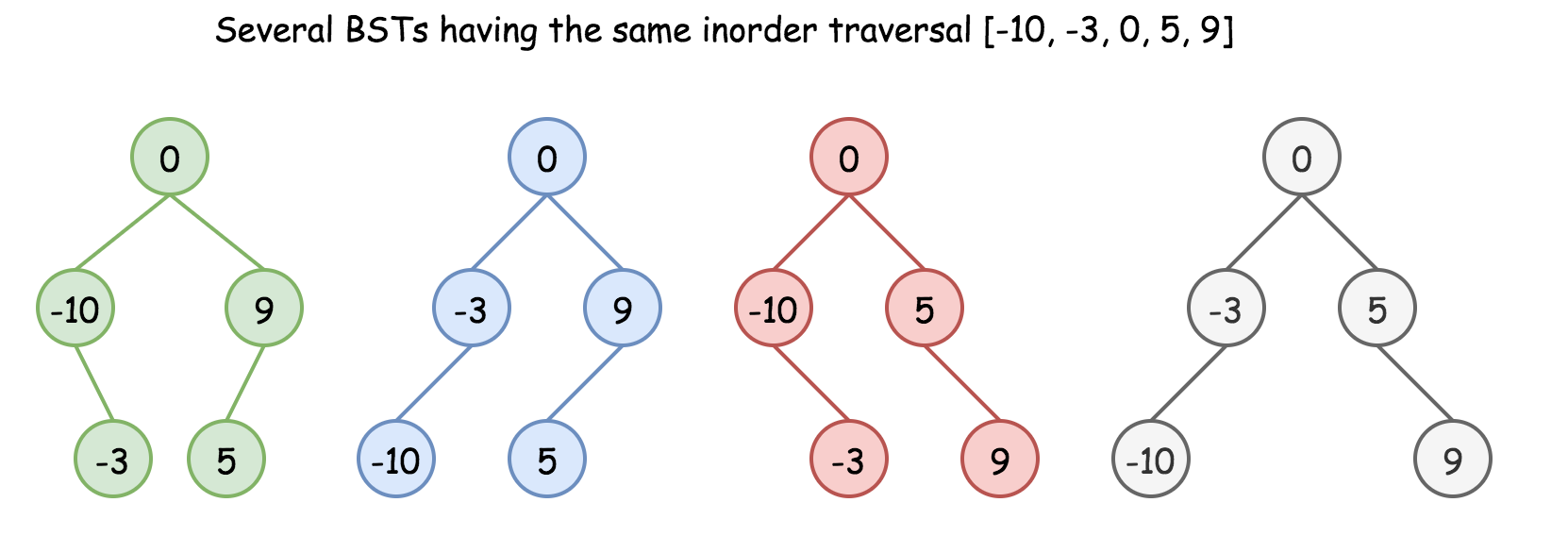
Here is the funny thing about BST. Inorder traversal is not a unique identifier of BST. At the same time both preorder and postorder traversals are unique identifiers of BST. [From these traversals one could restore the inorder one](https://leetcode.com/articles/construct-bst-from-preorder-traversal/): inorder = sorted(postorder) = sorted(preorder), and [inorder + postorder or inorder + preorder are both unique identifiers of whatever binary tree](https://leetcode.com/articles/construct-binary-tree-from-inorder-and-postorder-t/).

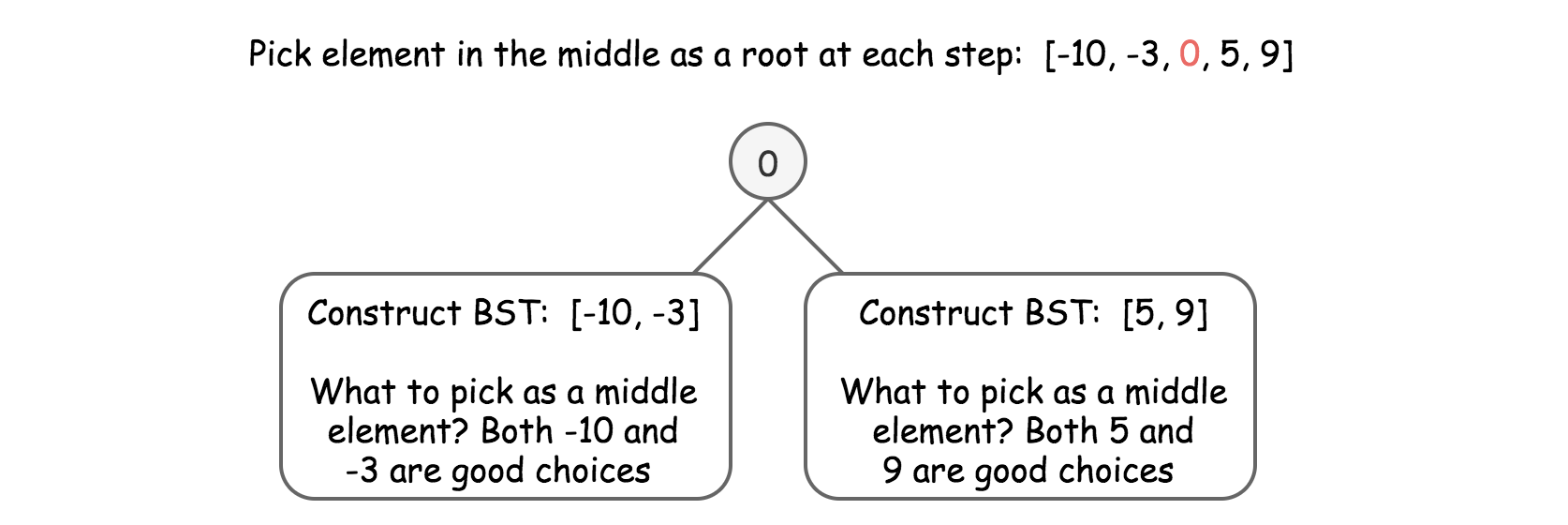
So, the problem "sorted array -> BST" has multiple solutions.



Here we have an additional condition: *the tree should be height-balanced*, i.e. the depths of the two subtrees of every node never differ by more than 1.

Does it make the solution to be unique? Still no.

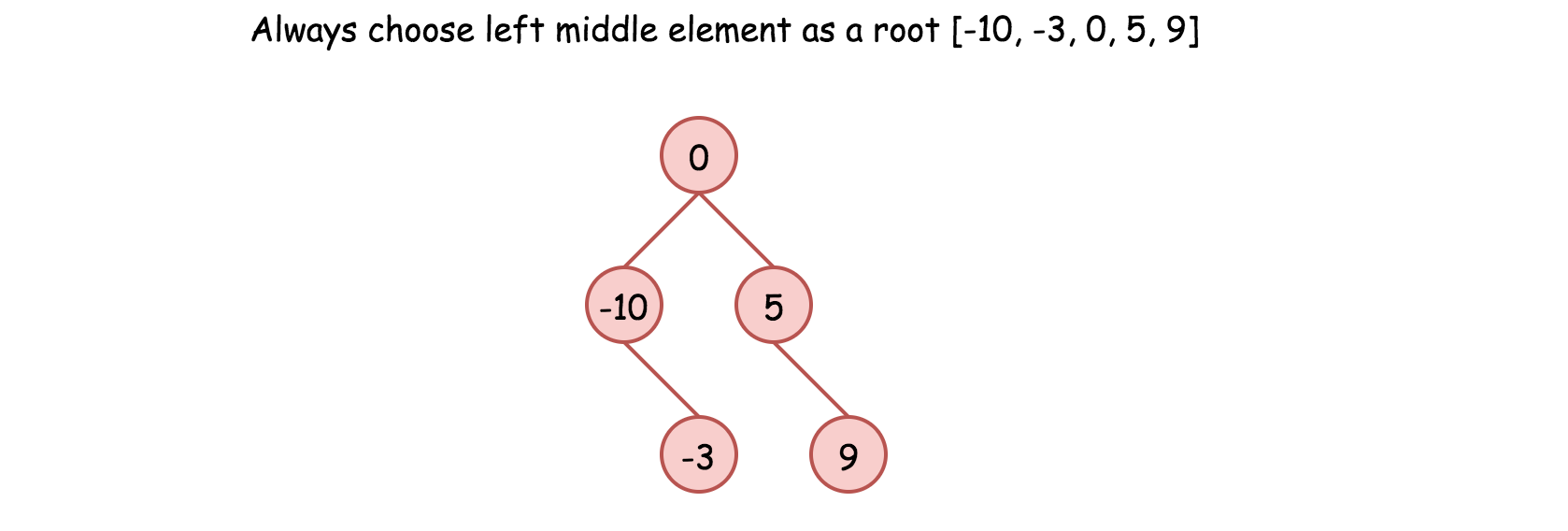




One could choose left middle element, or right middle one, and both choices will lead to different height-balanced BSTs. Let's see that in practice: in Approach 1 we will always pick up left middle element, in Approach 2 - right middle one. That will generate different BSTs but both solutions will be accepted.

#### **Approach 1: Preorder Traversal: Always Choose Left Middle Node as a Root**

**Algorithm**



* Implement helper function helper(left, right), which constructs BST from nums elements between indexes left and right:
  + If left > right, then there is no elements available for that subtree. Return None.
  + Pick left middle element: p = (left + right) // 2.
  + Initiate the root: root = TreeNode(nums[p]).
  + Compute recursively left and right subtrees: root.left = helper(left, p - 1), root.right = helper(p + 1, right).
* Return helper(0, len(nums) - 1).

class Solution {

  int[] nums;

  public TreeNode helper(int left, int right) {

    if (left > right) return null;

    // always choose left middle node as a root

    int p = (left + right) / 2;

    // preorder traversal: node -> left -> right

    TreeNode root = new TreeNode(nums[p]);

    root.left = helper(left, p - 1);

    root.right = helper(p + 1, right);

    return root;

  }

  public TreeNode sortedArrayToBST(int[] nums) {

    this.nums = nums;

    return helper(0, nums.length - 1);

  }

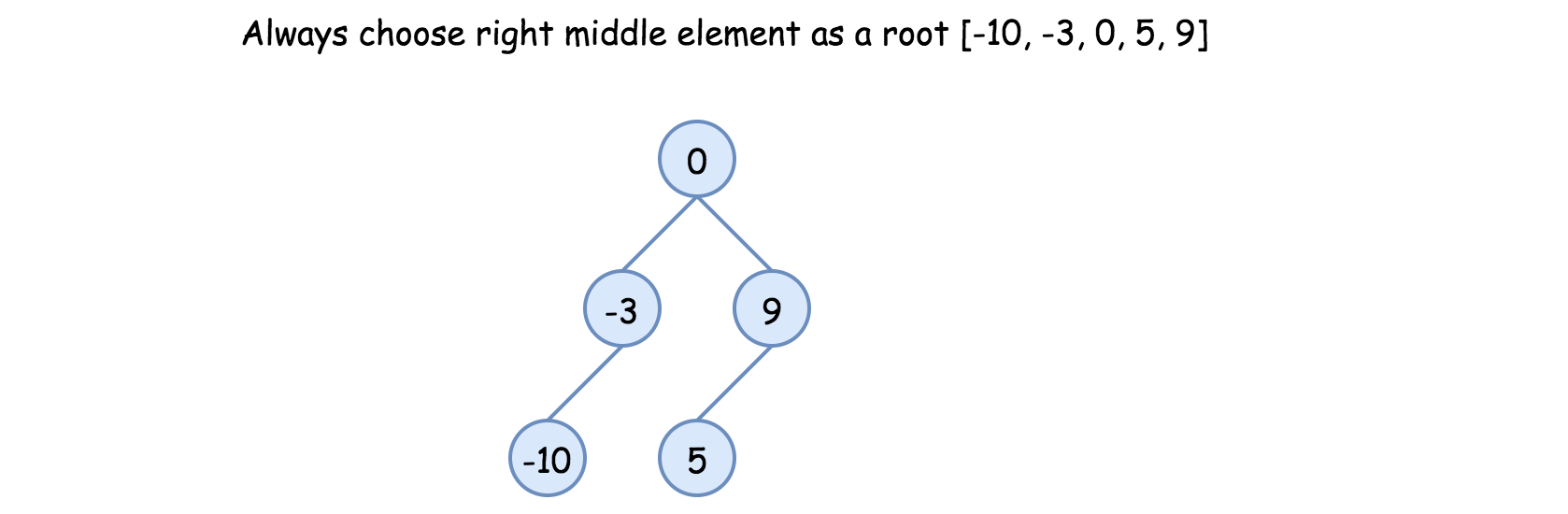
}

**Complexity Analysis**

* Time complexity: O(*N*) since we visit each node exactly once.
* Space complexity:  O(*N*).  O(*N*) to keep the output, and  O(log*N*) for the recursion stack.

#### **Approach 2: Preorder Traversal: Always Choose Right Middle Node as a Root**

**Algorithm**



* Implement helper function helper(left, right), which constructs BST from nums elements between indexes left and right:
  + If left > right, then there is no elements available for that subtree. Return None.
  + Pick right middle element:
    - p = (left + right) // 2.
    - If left + right is odd, add 1 to p-index.
  + Initiate the root: root = TreeNode(nums[p]).
  + Compute recursively left and right subtrees: root.left = helper(left, p - 1), root.right = helper(p + 1, right).
* Return helper(0, len(nums) - 1).

class Solution {

  int[] nums;

  public TreeNode helper(int left, int right) {

    if (left > right) return null;

    // always choose right middle node as a root

    int p = (left + right) / 2;

    if ((left + right) % 2 == 1) ++p;

    // preorder traversal: node -> left -> right

    TreeNode root = new TreeNode(nums[p]);

    root.left = helper(left, p - 1);

    root.right = helper(p + 1, right);

    return root;

  }

  public TreeNode sortedArrayToBST(int[] nums) {

    this.nums = nums;

    return helper(0, nums.length - 1);

  }

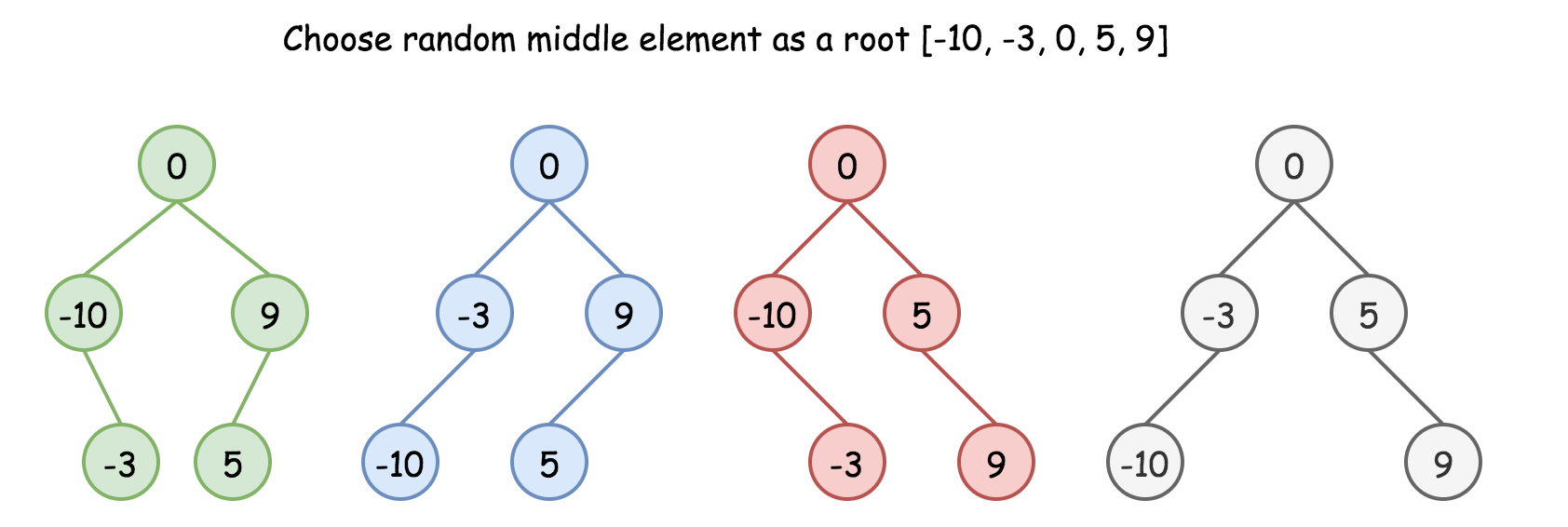
}

**Complexity Analysis**

* Time complexity:  O(*N*) since we visit each node exactly once.
* Space complexity:  O(*N*).  O(*N*) to keep the output, and  O(log*N*) for the recursion stack

#### **Approach 3: Preorder Traversal: Choose Random Middle Node as a Root**

This one is for fun. Instead of predefined choice we will pick randomly left or right middle node at each step. Each run will result in different solution and they all will be accepted.



**Algorithm**

* Implement helper function helper(left, right), which constructs BST from nums elements between indexes left and right:
  + If left > right, then there is no elements available for that subtree. Return None.
  + Pick random middle element:
    - p = (left + right) // 2.
    - If left + right is odd, add randomly 0 or 1 to p-index.
  + Initiate the root: root = TreeNode(nums[p]).
  + Compute recursively left and right subtrees: root.left = helper(left, p - 1), root.right = helper(p + 1, right).
* Return helper(0, len(nums) - 1).

class Solution {

    int[] nums;

    Random rand = new Random();

    public TreeNode helper(int left, int right) {

        if (left > right) return null;

        // choose random middle node as a root

        int p = (left + right) / 2;

        if ((left + right) % 2 == 1) p += rand.nextInt(2);

        // preorder traversal: node -> left -> right

        TreeNode root = new TreeNode(nums[p]);

        root.left = helper(left, p - 1);

        root.right = helper(p + 1, right);

        return root;

    }

    public TreeNode sortedArrayToBST(int[] nums) {

        this.nums = nums;

        return helper(0, nums.length - 1);

    }

}

**Complexity Analysis**

* Time complexity:  O(*N*) since we visit each node exactly once.
* Space complexity:  O(*N*).  O(*N*) to keep the output, and  O(log*N*) for the recursion stack.

### Convert Binary Search Tree to Sorted Doubly Linked List

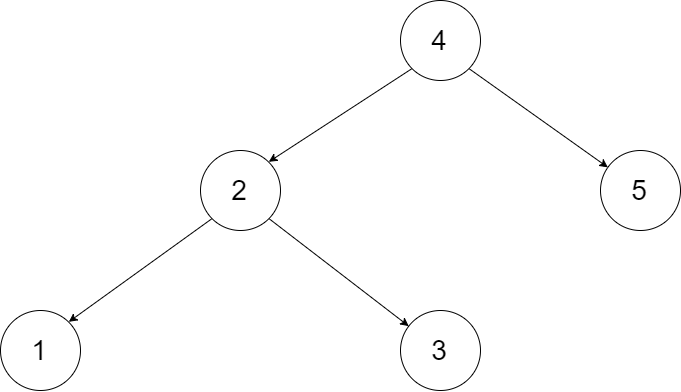
### Problem Statement 5

Convert a **Binary Search Tree** to a sorted **Circular Doubly-Linked List** in place.

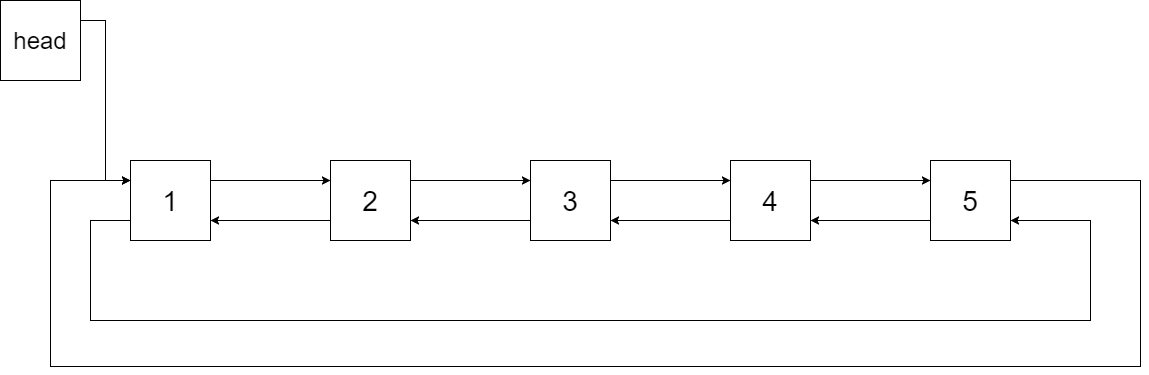
You can think of the left and right pointers as synonymous to the predecessor and successor pointers in a doubly-linked list. For a circular doubly linked list, the predecessor of the first element is the last element, and the successor of the last element is the first element.

We want to do the transformation **in place**. After the transformation, the left pointer of the tree node should point to its predecessor, and the right pointer should point to its successor. You should return the pointer to the smallest element of the linked list.

**Example 1:**

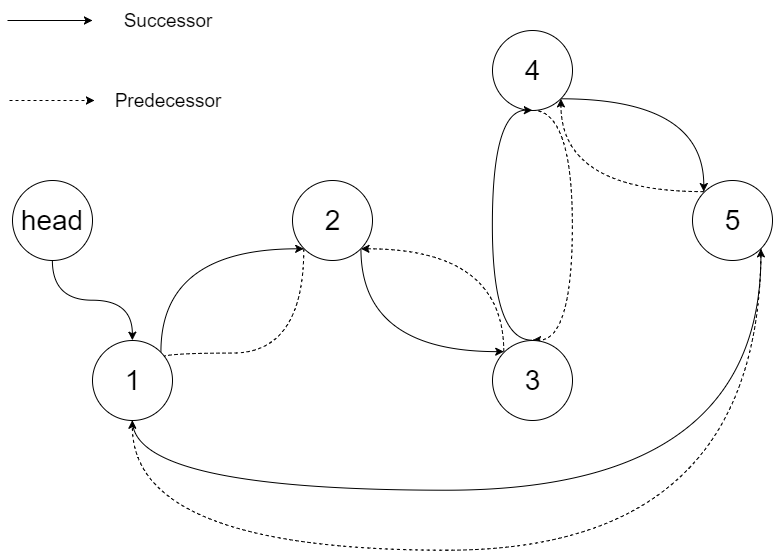


**Input:** root = [4,2,5,1,3]



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

**Explanation:** The figure below shows the transformed BST. The solid line indicates the successor relationship, while the dashed line means the predecessor relationship.



**Example 2:**

**Input:** root = [2,1,3]

**Output:** [1,2,3]

**Example 3:**

**Input:** root = []

**Output:** []

**Explanation:** Input is an empty tree. Output is also an empty Linked List.

**Example 4:**

**Input:** root = [1]

**Output:** [1]

## **Solution**

#### **How to traverse the tree**

There are two general strategies to traverse a tree:

* Depth First Search (DFS)

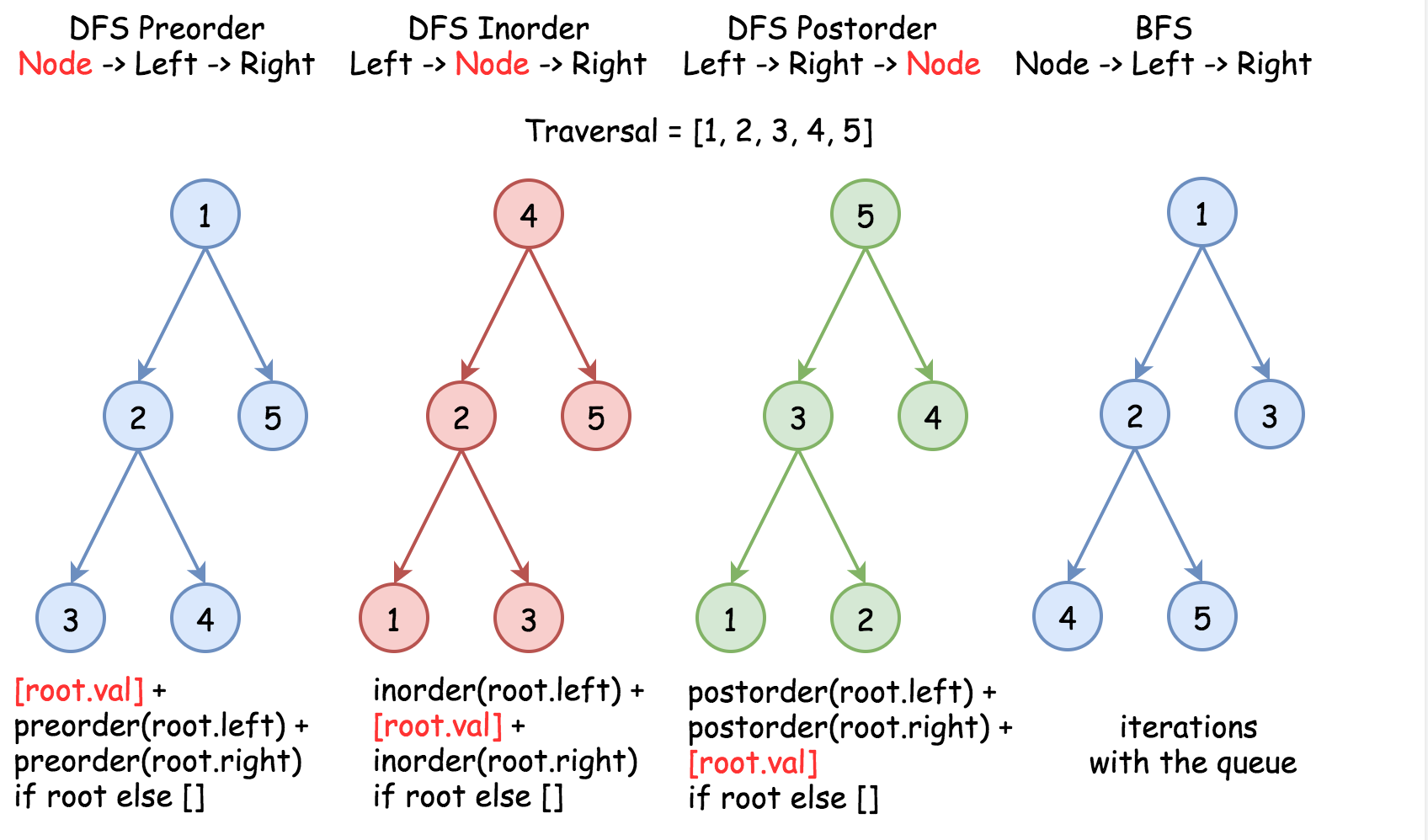
In this strategy, we adopt the depth as the priority, so that one would start from a root and reach all the way down to certain leaf, and then back to root to reach another branch.

The DFS strategy can further be distinguished as preorder, inorder, and postorder depending on the relative order among the root node, left node and right node.

* Breadth First Search (BFS)

We scan through the tree level by level, following the order of height, from top to bottom. The nodes on higher level would be visited before the ones with lower levels.

On the following figure the nodes are numerated in the order you visit them, please follow 1-2-3-4-5 to compare different strategies.



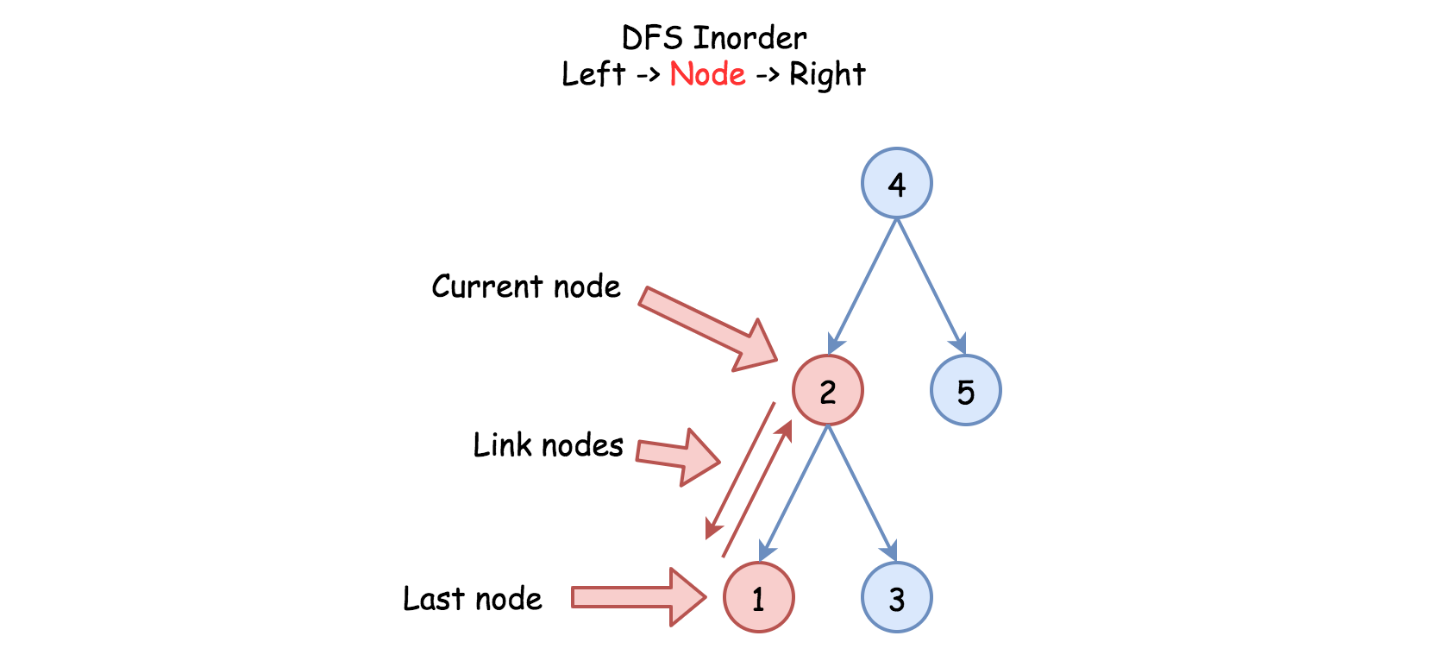
Here the problem is to implement DFS inorder traversal in a textbook recursion way because of in-place requirement.

#### **Approach 1: Recursion**

**Algorithm**

Standard inorder recursion follows left -> node -> right order, where left and right parts are the recursion calls and node part is where all processing is done.

Processing here is basically to link the previous node with the current one, and because of that one has to track the last node which is the largest node in a new doubly linked list so far.



One more detail : one has to keep the first, or the smallest, node as well to close the ring of doubly linked list.

Here is the algorithm :

* Initiate the first and the last nodes as nulls.
* Call the standard inorder recursion helper(root) :
  + If node is not null :
    - Call the recursion for the left subtree helper(node.left).
    - If the last node is not null, link the last and the current node nodes.
    - Else initiate the first node.
    - Mark the current node as the last one : last = node.
    - Call the recursion for the right subtree helper(node.right).
* Link the first and the last nodes to close DLL ring and then return the first node.

class Solution {

  // the smallest (first) and the largest (last) nodes

  Node first = null;

  Node last = null;

  public void helper(Node node) {

    if (node != null) {

      // left

      helper(node.left);

      // node

      if (last != null) {

        // link the previous node (last)

        // with the current one (node)

        last.right = node;

        node.left = last;

      }

      else {

        // keep the smallest node

        // to close DLL later on

        first = node;

      }

      last = node;

      // right

      helper(node.right);

    }

  }

  public Node treeToDoublyList(Node root) {

    if (root == null) return null;

    helper(root);

    // close DLL

    last.right = first;

    first.left = last;

    return first;

  }

}

**Complexity Analysis**

* Time complexity :  O(*N*) since each node is processed exactly once.
* Space complexity :  O(*N*). We have to keep a recursion stack of the size of the tree height, which is  O(log*N*) for the best case of completely balanced tree and  O(*N*) for the worst case of completely unbalanced tree.

### Convert a given tree to its Sum Tree

### Problem Statement 7

Given a Binary Tree where each node has positive and negative values. Convert this to a tree where each node contains the sum of the left and right sub trees in the original tree. The values of leaf nodes are changed to 0.

For example, the following tree

**Example:**

10

/ \

-2 6

/ \ / \

8 -4 7 5

should be changed to

20(4-2+12+6)

/ \

4(8-4) 12(7+5)

/ \ / \

0 0 0 0

## **Solution**

Do a traversal of the given tree. In the traversal, store the old value of the current node, recursively call for left and right subtrees and change the value of current node as sum of the values returned by the recursive calls. Finally return the sum of new value and value (which is sum of values in the subtree rooted with this node).

public class SumTree {

    public static int sumTree(TreeNode root) {

        if (root == null)

            return 0;

        int val = root.data;

        root.data = sumTree(root.left) + sumTree(root.right);

        return root.data + val;

    }

    public static void printTree(TreeNode root) {

        if (root != null) {

            printTree(root.left);

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

            printTree(root.right);

        }

    }

    public static void main(String[] args) {

        TreeNode root = new TreeNode(10);

        root.left = new TreeNode(-2);

        root.left.left = new TreeNode(8);

        root.left.right = new TreeNode(-4);

        root.right = new TreeNode(6);

        root.right.left = new TreeNode(7);

        root.right.right = new TreeNode(5);

        printTree(root);

        sumTree(root);

        System.out.println();

        printTree(root);

    }

}

### Convert BST to Greater Tree

### Problem Statement 8

Given a Binary Search Tree (BST), convert it to a Greater Tree such that every key of the original BST is changed to the original key plus sum of all keys greater than the original key in BST.

**Example:**

**Input:** The root of a Binary Search Tree like this:

5

/ \

2 13

**Output:** The root of a Greater Tree like this:

18

/ \

20 13

#### **Initial Thoughts**

This question asks us to modify an asymptotically linear number of nodes in a given binary search tree, so a very efficient solution will visit each node once. The key to such a solution would be a way to visit nodes in descending order, keeping a sum of all values that we have already visited and adding that sum to the node's values as we traverse the tree. This method for tree traversal is known as a reverse in-order traversal, and allows us to guarantee visitation of each node in the desired order. The basic idea of such a traversal is that before visiting any node in the tree, we must first visit all nodes with greater value. Where are all of these nodes conveniently located? In the right subtree.

#### **Approach #1 Recursion**

**Intuition**

One way to perform a reverse in-order traversal is via recursion. By using the call stack to return to previous nodes, we can easily visit the nodes in reverse order.

**Algorithm**

For the recursive approach, we maintain some minor "global" state so each recursive call can access and modify the current total sum. Essentially, we ensure that the current node exists, recurse on the right subtree, visit the current node by updating its value and the total sum, and finally recurse on the left subtree. If we know that recursing on root.right properly updates the right subtree and that recursing on root.left properly updates the left subtree, then we are guaranteed to update all nodes with larger values before the current node and all nodes with smaller values after.

class Solution {

    private int sum = 0;

    public TreeNode convertBST(TreeNode root) {

        if (root != null) {

            convertBST(root.right);

            sum += root.val;

            root.val = sum;

            convertBST(root.left);

        }

        return root;

    }

}

**Complexity Analysis**

* Time complexity :  O(*n*)

A binary tree has no cycles by definition, so convertBST gets called on each node no more than once. Other than the recursive calls, convertBST does a constant amount of work, so a linear number of calls to convertBST will run in linear time.

* Space complexity :  O(*n*)

Using the prior assertion that convertBST is called a linear number of times, we can also show that the entire algorithm has linear space complexity. Consider the worst case, a tree with only right (or only left) subtrees. The call stack will grow until the end of the longest path is reached, which in this case includes all *n* nodes.

#### **Approach #2 Iteration with a Stack**

**Intuition**

If we don't want to use recursion, we can also perform a reverse in-order traversal via iteration and a literal stack to emulate the call stack.

**Algorithm**

One way to describe the iterative stack method is in terms of the intuitive recursive solution. First, we initialize an empty stack and set the current node to the root. Then, so long as there are unvisited nodes in the stack or node does not point to null, we push all of the nodes along the path to the rightmost leaf onto the stack. This is equivalent to always processing the right subtree first in the recursive solution, and is crucial for the guarantee of visiting nodes in order of decreasing value. Next, we visit the node on the top of our stack, and consider its left subtree. This is just like visiting the current node before recursing on the left subtree in the recursive solution. Eventually, our stack is empty and node points to the left null child of the tree's minimum value node, so the loop terminates.

class Solution {

    public TreeNode convertBST(TreeNode root) {

        int sum = 0;

        TreeNode node = root;

        Stack<TreeNode> stack = new Stack<TreeNode>();

        while (!stack.isEmpty() || node != null) {

            /\* push all nodes up to (and including) this subtree's maximum on

             \* the stack. \*/

            while (node != null) {

                stack.add(node);

                node = node.right;

            }

            node = stack.pop();

            sum += node.val;

            node.val = sum;

            /\* all nodes with values between the current and its parent lie in

             \* the left subtree. \*/

            node = node.left;

        }

        return root;

    }

}

### Flip Binary Tree

### Problem Statement 9

Given a binary tree, the task is to flip the binary tree towards right direction that is clockwise. See below examples to see the transformation.

In the flip operation, left most node becomes the root of flipped tree and its parent become its right child and the right sibling become its left child and same should be done for all left most nodes recursively.

**Example:**

**Input:** The root of a Binary Tree like this:

1

/ \

2 3

/ \ / \

4 5 6 7

**Output:** The root of a Greater Tree like this:

4

/ \

5 2

/ \

3 1

/ \

6 7

#### **Approach #1 Recursion**

Below is main rotation code of a subtree

root->left->left = root->right;

root->left->right = root;

root->left = NULL;

root->right = NULL;

class TreeNode {

  int val;

  TreeNode left;

  TreeNode right;

  TreeNode(int x) {

    val = x;

  }

};

public class FlipTree {

    public static TreeNode flipTree(TreeNode root) {

        if (root == null)

            return null;

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

            return root;

        TreeNode flipRoot = flipTree(root.left);

        root.left.left = root.right;

        root.left.right = root;

        root.left = root.right = null;

        return flipRoot;

    }

    public static void printTree(TreeNode root) {

        if (root != null) {

            printTree(root.left);

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

            printTree(root.right);

        }

    }

    public static void main(String[] args) {

        TreeNode root = new TreeNode(1);

        root.left = new TreeNode(2);

        root.left.left = new TreeNode(4);

        root.left.right = new TreeNode(5);

        root.right = new TreeNode(3);

        root.right.left = new TreeNode(6);

        root.right.right = new TreeNode(7);

        printTree(root);

        TreeNode flipRoot = flipTree(root);

        System.out.println();

        printTree(flipRoot);

    }

}

#### **Approach #2 Iteration**

The iterative solution follows the same approach as the recursive one, the only thing we need to pay attention to is to save the node information that will be overwritten.

public static TreeNode flipTreeIterative(TreeNode root)

{

    // Initialization of pointers

    TreeNode curr = root;

    TreeNode next = null;

    TreeNode temp = null;

    TreeNode prev = null;

    // Iterate through all left nodes

    while(curr != null)

    {

        next = curr.left;

        // Swapping nodes now, need

        // temp to keep the previous

        // right child

        // Making prev's right

        // as curr's left child

        curr.left = temp;

        // Storing curr's right child

        temp = curr.right;

        // Making prev as curr's

        // right child

        curr.right = prev;

        prev = curr;

        curr = next;

    }

    return prev;

}

### Sum of all nodes in left subtree

### Problem Statement 10

Given a Binary Tree, change the value in each node to sum of all the values in the nodes in the left subtree including its own.

**Example:**

**Input :**

**1**

**/ \**

**2 3**

**Output :**

**3**

**/ \**

**2 3**

Input

1

/ \

2 3

/ \ \

4 5 6

Output:

12

/ \

6 3

/ \ \

4 5 6

## **Solution**

The idea is to traverse the given tree in bottom up manner. For every node, recursively compute sum of nodes in left and right subtrees. Add sum of nodes in left subtree to current node and return sum of nodes under current subtree.

class TreeNode {

  int data;

  TreeNode left;

  TreeNode right;

  TreeNode(int x) {

    data = x;

  }

};

public class LeftSumTree {

    public static int update(TreeNode root) {

        if (root == null)

            return 0;

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

            return root.data;

        int leftSum = update(root.left);

        int rightSum = update(root.right);

        root.data = root.data + leftSum;

        return root.data + rightSum;

    }

    public static void printTree(TreeNode root) {

        if (root != null) {

            printTree(root.left);

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

            printTree(root.right);

        }

    }

    public static void main(String[] args) {

        TreeNode root = new TreeNode(1);

        root.left = new TreeNode(2);

//        root.left.left = new TreeNode(4);

//        root.left.right = new TreeNode(5);

        root.right = new TreeNode(3);

//        root.right.left = new TreeNode(6);

//        root.right.right = new TreeNode(7);

        printTree(root);

       update(root);

        System.out.println();

        printTree(root);

    }

}

### Merge Two Binary Trees

### Problem Statement 11

Given two binary trees and imagine that when you put one of them to cover the other, some nodes of the two trees are overlapped while the others are not.

You need to merge them into a new binary tree. The merge rule is that if two nodes overlap, then sum node values up as the new value of the merged node. Otherwise, the NOT null node will be used as the node of new tree.

**Example 1:**

**Input:**

Tree 1 Tree 2

1 2

/ \ / \

3 2 1 3

/ \ \

5 4 7

**Output:**

Merged tree:

3

/ \

4 5

/ \ \

5 4 7

## **Solution**

#### **Approach #1 Using Recursion**

We can traverse both the given trees in a preorder fashion. At every step, we check if the current node exists(isn't null) for both the trees. If so, we add the values in the current nodes of both the trees and update the value in the current node of the first tree to reflect this sum obtained. At every step, we also call the original function mergeTrees() with the left children and then with the right children of the current nodes of the two trees. If at any step, one of these children happens to be null, we return the child of the other tree(representing the corresponding child subtree) to be added as a child subtree to the calling parent node in the first tree. At the end, the first tree will represent the required resultant merged binary tree.

public TreeNode mergeTrees(TreeNode t1, TreeNode t2) {

        if (t1 == null)

            return t2;

        if (t2 == null)

            return t1;

        t1.val += t2.val;

        t1.left = mergeTrees(t1.left, t2.left);

        t1.right = mergeTrees(t1.right, t2.right);

        return t1;

}

**Complexity Analysis**

* Time complexity : O(m). A total of m nodes need to be traversed. Here, m*m* represents the minimum number of nodes from the two given trees.
* Space complexity : O(m). The depth of the recursion tree can go upto m in the case of a skewed tree. In average case, depth will be O(logm).

#### **Approach #2 Iterative Method**

**Algorithm**

In the current approach, we again traverse the two trees, but this time we make use of a stack*stack* to do so instead of making use of recursion. Each entry in the stack*stack* strores data in the form [node\_{tree1}, node\_{tree2}][*nodetree*1​,*nodetree*2​]. Here, node\_{tree1}*nodetree*1​ and node\_{tree2}*nodetree*2​ are the nodes of the first tree and the second tree respectively.

We start off by pushing the root nodes of both the trees onto the stack*stack*. Then, at every step, we remove a node pair from the top of the stack. For every node pair removed, we add the values corresponding to the two nodes and update the value of the corresponding node in the first tree. Then, if the left child of the first tree exists, we push the left child(pair) of both the trees onto the stack. If the left child of the first tree doesn't exist, we append the left child(subtree) of the second tree to the current node of the first tree. We do the same for the right child pair as well.

If, at any step, both the current nodes are null, we continue with popping the next nodes from the stack*stack*.

public TreeNode mergeTrees(TreeNode t1, TreeNode t2) {

        if (t1 == null)

            return t2;

        Stack < TreeNode[] > stack = new Stack < > ();

        stack.push(new TreeNode[] {t1, t2});

        while (!stack.isEmpty()) {

            TreeNode[] t = stack.pop();

            if (t[0] == null || t[1] == null) {

                continue;

            }

            t[0].val += t[1].val;

            if (t[0].left == null) {

                t[0].left = t[1].left;

            } else {

                stack.push(new TreeNode[] {t[0].left, t[1].left});

            }

            if (t[0].right == null) {

                t[0].right = t[1].right;

            } else {

                stack.push(new TreeNode[] {t[0].right, t[1].right});

            }

        }

        return t1;

}

**Complexity Analysis**

* Time complexity : O(n). We traverse over a total of n nodes. Here, nrefers to the smaller of the number of nodes in the two trees.
* Space complexity : O(n). The depth of stack can grow upto n in case of a skewed tree.

### Flip Equivalent Binary Trees

### Problem Statement 12

For a binary tree T, we can define a flip operation as follows: choose any node, and swap the left and right child subtrees.

A binary tree X is *flip equivalent* to a binary tree Y if and only if we can make X equal to Y after some number of flip operations.

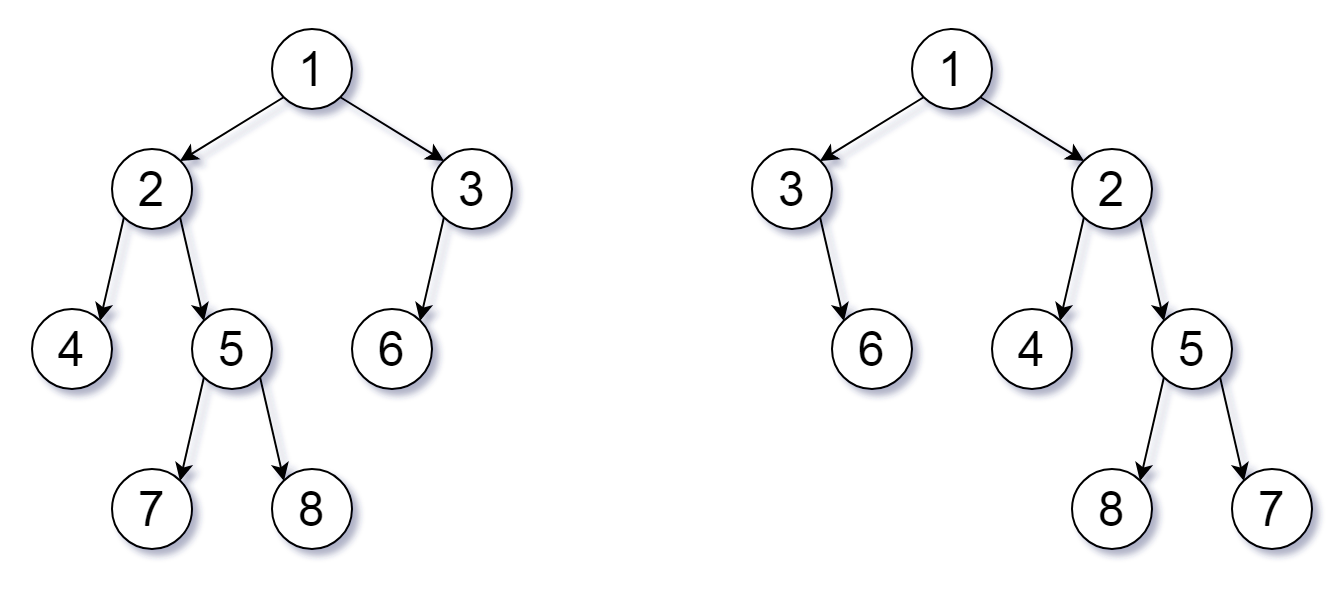
Write a function that determines whether two binary trees are *flip equivalent*.  The trees are given by root nodes root1 and root2.

**Example 1:**

**Input:** root1 = [1,2,3,4,5,6,null,null,null,7,8], root2 = [1,3,2,null,6,4,5,null,null,null,null,8,7]

**Output:** true

**Explanation:** We flipped at nodes with values 1, 3, and 5.



## **Solution**

#### **Approach 1: Recursion**

**Intuition**

If root1 and root2 have the same root value, then we only need to check if their children are equal (up to ordering.)

**Algorithm**

There are 3 cases:

* If root1 or root2 is null, then they are equivalent if and only if they are both null.
* Else, if root1 and root2 have different values, they aren't equivalent.
* Else, let's check whether the children of root1 are equivalent to the children of root2. There are two different ways to pair these children.

public boolean flipEquiv(TreeNode root1, TreeNode root2) {

        if (root1 == root2)

            return true;

        if (root1 == null || root2 == null || root1.val != root2.val)

            return false;

        return (flipEquiv(root1.left, root2.left) && flipEquiv(root1.right, root2.right) ||

                flipEquiv(root1.left, root2.right) && flipEquiv(root1.right, root2.left));

}

**Complexity Analysis**

* Time Complexity: *O*(*min*(*N*1​,*N*2​)), where *N*1​,*N*2​ are the lengths of root1 and root2.
* Space Complexity: *O*(*min*(*H*1​,*H*2​)), where *H*1​,*H*2​ are the heights of the trees of root1 and root2.

#### **Approach 2: Canonical Traversal**

**Intuition**

Flip each node so that the left child is smaller than the right, and call this the canonical representation. All equivalent trees have exactly one canonical representation.

**Algorithm**

We can use a depth-first search to compare the canonical representation of each tree. If the traversals are the same, the representations are equal.

When traversing, we should be careful to encode both when we enter or leave a node.

class Solution {

    public boolean flipEquiv(TreeNode root1, TreeNode root2) {

        List<Integer> vals1 = new ArrayList();

        List<Integer> vals2 = new ArrayList();

        dfs(root1, vals1);

        dfs(root2, vals2);

        return vals1.equals(vals2);

    }

    public void dfs(TreeNode node, List<Integer> vals) {

        if (node != null) {

            vals.add(node.val);

            int L = node.left != null ? node.left.val : -1;

            int R = node.right != null ? node.right.val : -1;

            if (L < R) {

                dfs(node.left, vals);

                dfs(node.right, vals);

            } else {

                dfs(node.right, vals);

                dfs(node.left, vals);

            }

            vals.add(null);

        }

    }

}

**Complexity Analysis**

* Time Complexity:  *O*(*N*1​+*N*2​), where *N*1​,*N*2​ are the lengths of root1 and root2. (In Python, this is min(*N*1​,*N*2​).)
* Space Complexity: *O*(*N*1​+*N*2​). (In Python, this is min(*H*1​,*H*2​), where H\_1, *H*1​,*H*2​ are the heights of the trees of root1 and root2.)

### MISC

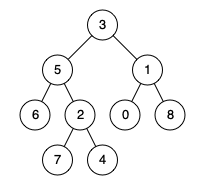
### Lowest Common Ancestor of a Binary Tree

### Problem Statement 1

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

According to the [definition of LCA on Wikipedia](https://en.wikipedia.org/wiki/Lowest_common_ancestor): “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow **a node to be a descendant of itself**).”

Given the following binary tree:  root = [3,5,1,6,2,0,8,null,null,7,4]



**Example 1:**

**Input:** root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1

**Output:** 3

**Explanation:** The LCA of nodes 5 and 1 is 3.

**Example 2:**

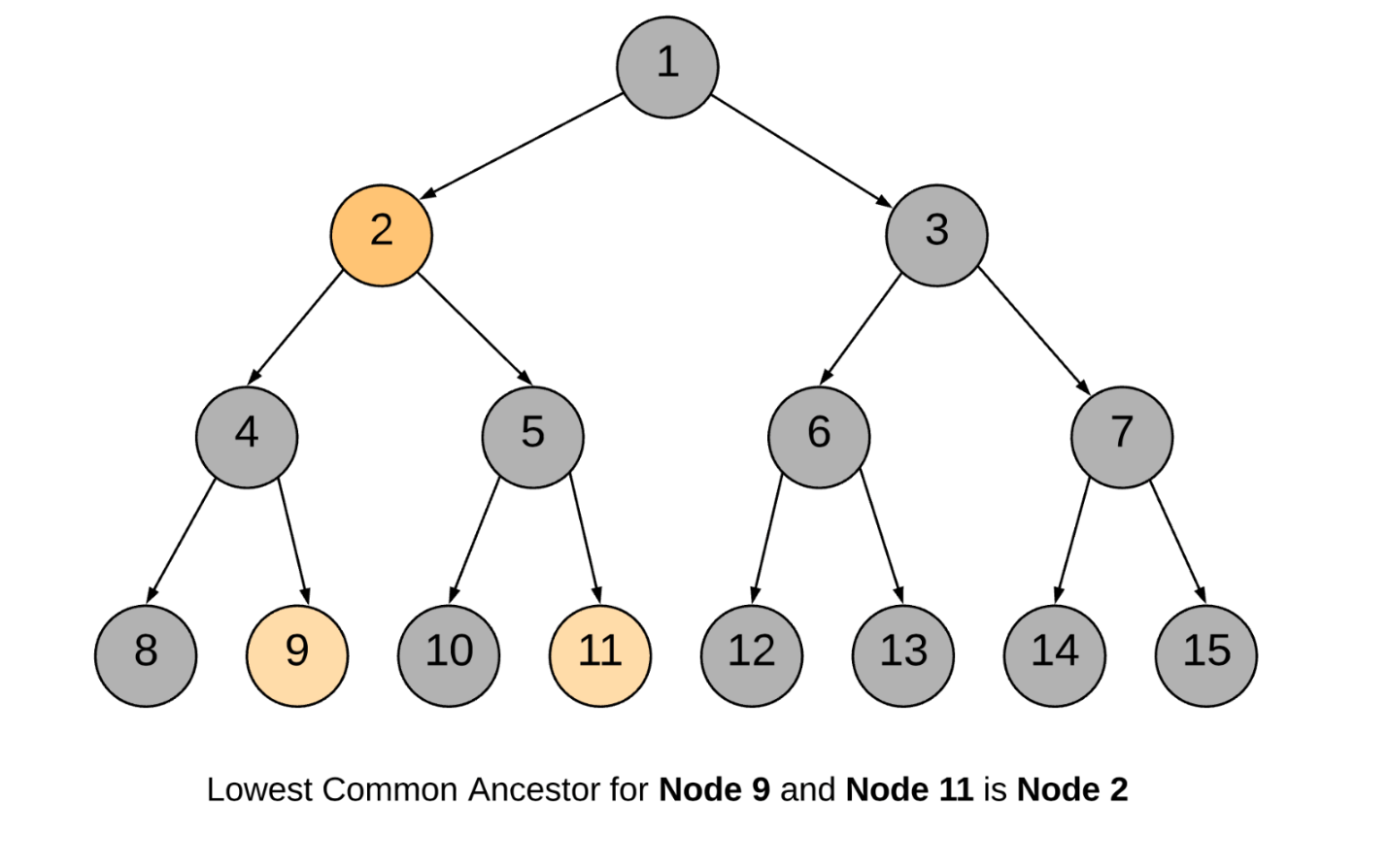
**Input:** root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4

**Output:** 5

**Explanation:** The LCA of nodes 5 and 4 is 5, since a node can be a descendant of itself according to the LCA definition.

## **Solution**

First the given nodes p and q are to be searched in a binary tree and then their lowest common ancestor is to be found. We can resort to a normal tree traversal to search for the two nodes. Once we reach the desired nodes p and q, we can backtrack and find the lowest common ancestor.



#### **Approach 1: Recursive Approach**

**Intuition**

The approach is pretty intuitive. Traverse the tree in a depth first manner. The moment you encounter either of the nodes p or q, return some boolean flag. The flag helps to determine if we found the required nodes in any of the paths. The least common ancestor would then be the node for which both the subtree recursions return a True flag. It can also be the node which itself is one of p or q and for which one of the subtree recursions returns a True flag.

Let us look at the formal algorithm based on this idea.

**Algorithm**

1. Start traversing the tree from the root node.
2. If the current node itself is one of p or q, we would mark a variable mid as True and continue the search for the other node in the left and right branches.
3. If either of the left or the right branch returns True, this means one of the two nodes was found below.
4. If at any point in the traversal, any two of the three flags left, right or mid become True, this means we have found the lowest common ancestor for the nodes p and q.

Following is the sequence of nodes that are followed in the recursion:

1 --> 2 --> 4 --> 8

BACKTRACK 8 --> 4

4 --> 9 (ONE NODE FOUND, return True)

BACKTRACK 9 --> 4 --> 2

2 --> 5 --> 10

BACKTRACK 10 --> 5

5 --> 11 (ANOTHER NODE FOUND, return True)

BACKTRACK 11 --> 5 --> 2

2 is the node where we have left = True and right = True and hence it is the lowest common ancestor.

class Solution {

    private TreeNode ans;

    public Solution() {

        // Variable to store LCA node.

        this.ans = null;

    }

    private boolean recurseTree(TreeNode currentNode, TreeNode p, TreeNode q) {

        // If reached the end of a branch, return false.

        if (currentNode == null) {

            return false;

        }

        // Left Recursion. If left recursion returns true, set left = 1 else 0

        int left = this.recurseTree(currentNode.left, p, q) ? 1 : 0;

        // Right Recursion

        int right = this.recurseTree(currentNode.right, p, q) ? 1 : 0;

        // If the current node is one of p or q

        int mid = (currentNode == p || currentNode == q) ? 1 : 0;

        // If any two of the flags left, right or mid become True

        if (mid + left + right >= 2) {

            this.ans = currentNode;

        }

        // Return true if any one of the three bool values is True.

        return (mid + left + right > 0);

    }

    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {

        // Traverse the tree

        this.recurseTree(root, p, q);

        return this.ans;

    }

}

**Complexity Analysis**

* Time Complexity: *O*(*N*), where *N* is the number of nodes in the binary tree. In the worst case we might be visiting all the nodes of the binary tree.
* Space Complexity: *O*(*N*). This is because the maximum amount of space utilized by the recursion stack would be *N* since the height of a skewed binary tree could be *N*.

#### **Approach 2: Iterative using parent pointers**

**Intuition**

If we have parent pointers for each node we can traverse back from p and q to get their ancestors. The first common node we get during this traversal would be the LCA node. We can save the parent pointers in a dictionary as we traverse the tree.

**Algorithm**

1. Start from the root node and traverse the tree.
2. Until we find p and q both, keep storing the parent pointers in a dictionary.
3. Once we have found both p and q, we get all the ancestors for p using the parent dictionary and add to a set called ancestors.
4. Similarly, we traverse through ancestors for node q. If the ancestor is present in the ancestors set for p, this means this is the first ancestor common between p and q (while traversing upwards) and hence this is the LCA node.

class Solution {

    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {

        // Stack for tree traversal

        Deque<TreeNode> stack = new ArrayDeque<>();

        // HashMap for parent pointers

        Map<TreeNode, TreeNode> parent = new HashMap<>();

        parent.put(root, null);

        stack.push(root);

        // Iterate until we find both the nodes p and q

        while (!parent.containsKey(p) || !parent.containsKey(q)) {

            TreeNode node = stack.pop();

            // While traversing the tree, keep saving the parent pointers.

            if (node.left != null) {

                parent.put(node.left, node);

                stack.push(node.left);

            }

            if (node.right != null) {

                parent.put(node.right, node);

                stack.push(node.right);

            }

        }

        // Ancestors set() for node p.

        Set<TreeNode> ancestors = new HashSet<>();

        // Process all ancestors for node p using parent pointers.

        while (p != null) {

            ancestors.add(p);

            p = parent.get(p);

        }

        // The first ancestor of q which appears in

        // p's ancestor set() is their lowest common ancestor.

        while (!ancestors.contains(q))

            q = parent.get(q);

        return q;

    }

}