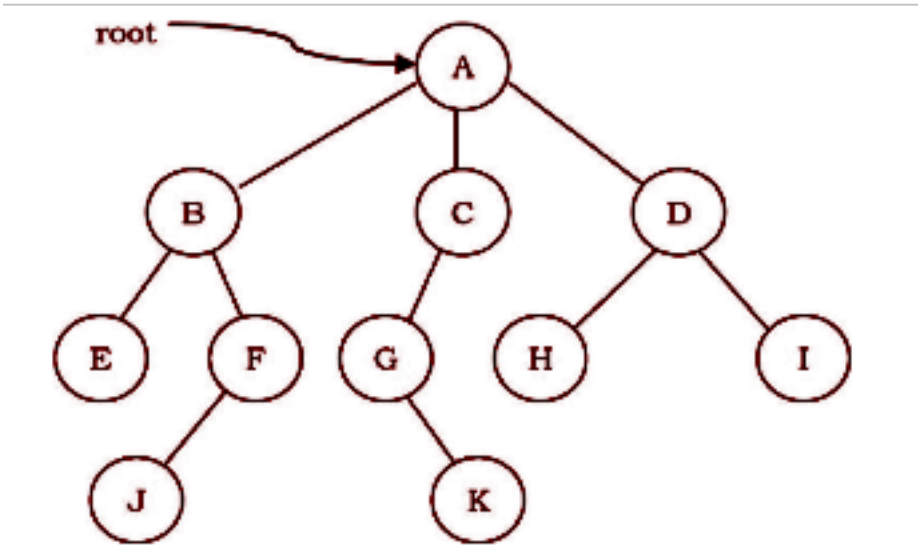
Binary Trees

What is A Tree?

- A tree is a data structure similar to a linked list but instead of each node pointing simply to the next node in a linear fashion, each node points to several nodes.
- A tree is an example of a non- linear data structure.
- A tree structure is a way of representing the hierarchical nature of a structure in a graphical form.

Terminology Of Trees

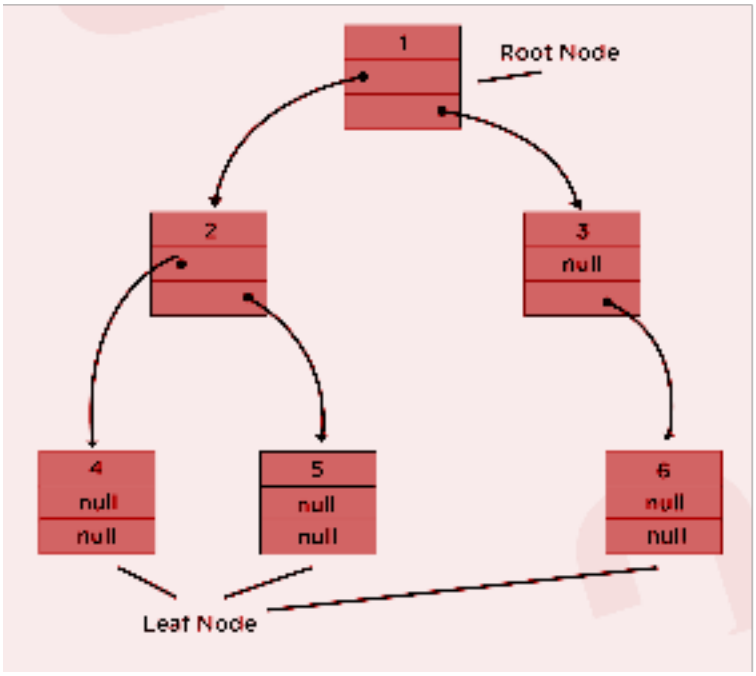


- The root of a tree is the node with no parents. There can be at most one root node in a tree **(node A in the above example)**.
- An **edge** refers to the link from a parent to a child **(all links in the figure)**.
- A node with no children is called a **leaf node** **(E, J, K, H, and I)**.
- The children nodes of the same parent are called **siblings** **(B, C, D are siblings of parent A, and E, F are siblings of parent B)**.

- The set of all nodes at a given depth is called the **level** of the tree **(B, C, and D are the same level)**. The root node is at level zero.
- The **depth** of a node is the length of the path from the root to the node **(depth of G is 2, *A -> C –> G*)**.
- The **height** of a node is the length of the path from that node to the deepest node.
- The **height** of a tree is the length of the path from the root to the deepest node in the tree.
- A (rooted) tree with only one node (the root) has a height of zero.

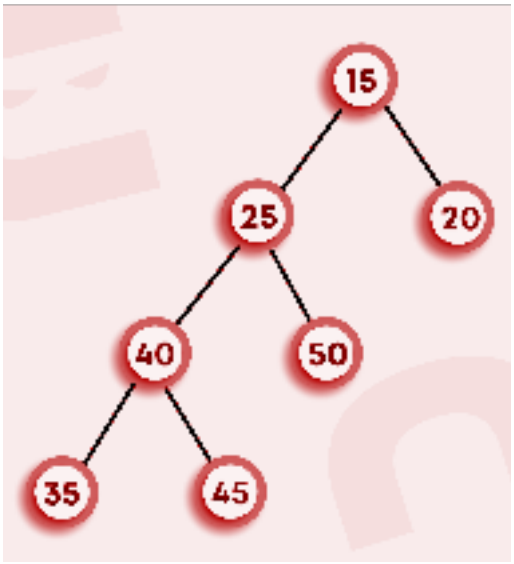
Binary Trees

- A generic tree with at most two child nodes for each parent node is known as a binary tree.
- A binary tree is made of nodes that constitute a **left** pointer, a **right** pointer, and a data element. The **root** pointer is the topmost node in the tree.
- The left and right pointers recursively point to smaller **subtrees** on either side.
- An empty tree is also a valid binary tree.
- *A formal definition is:* A **binary tree** is either empty (represented by a null pointer), or is made of a single node, where the left and right pointers (recursive definition ahead) each point to a **binary tree**.



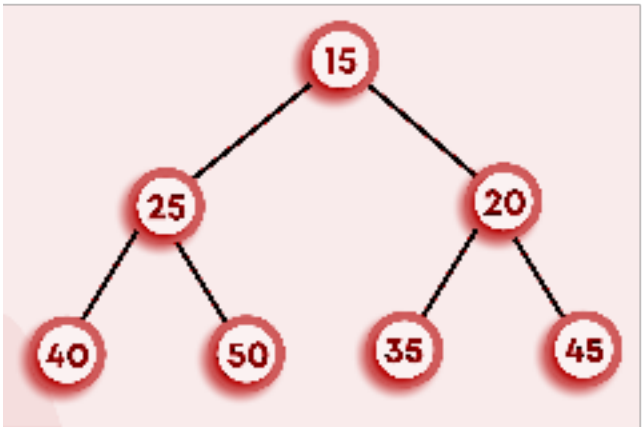
Types of binary trees:

Full binary trees: A binary tree in which every node has 0 or 2 children is termed as a full binary tree.

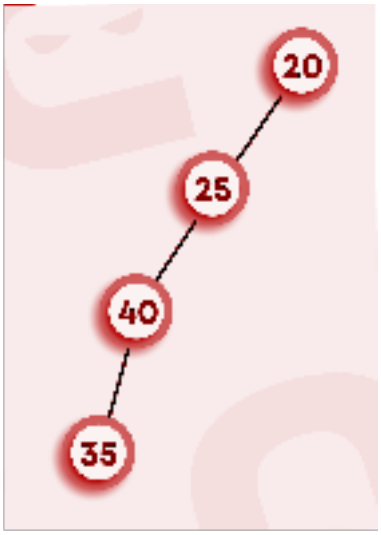


Complete binary tree: A complete binary tree has all the levels filled except for the last level, which has all its nodes as much as to the left.

Perfect binary tree: A binary tree is termed perfect when all its internal nodes have two children along with the leaf nodes that are at the same level.

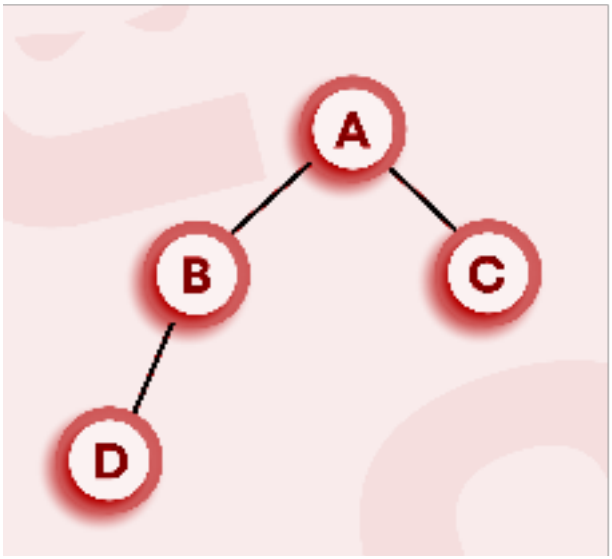


A degenerate tree: In a degenerate tree, each internal node has only one child.



The tree shown above is degenerate. These trees are very similar to linked-lists.

Balanced binary tree: A binary tree in which the difference between the depth of the two subtrees of every node is at most one is called a balanced binary tree.



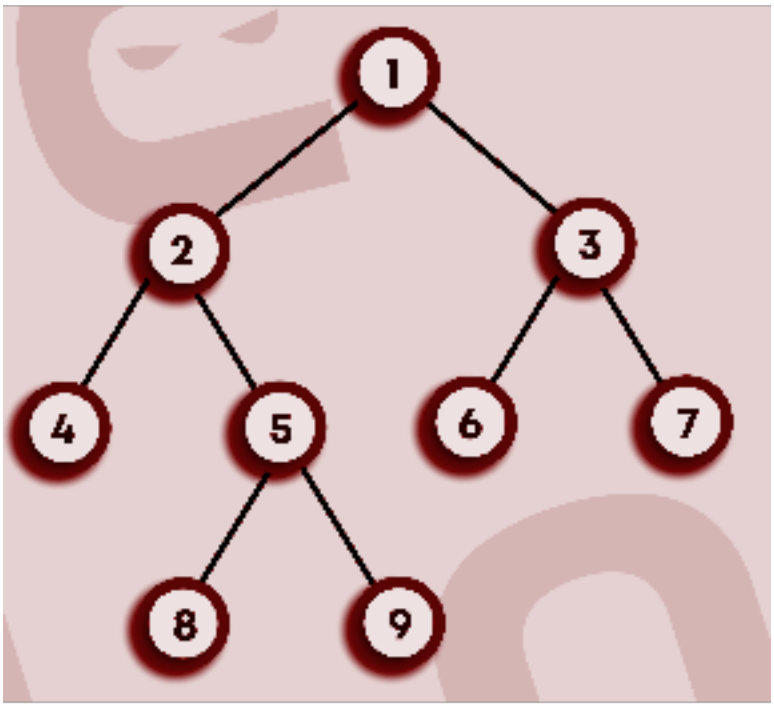
Binary tree representation:

Binary trees can be represented in two ways:

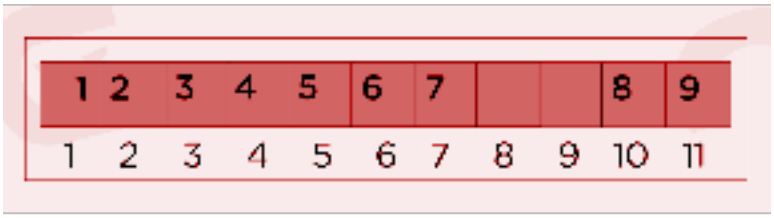
Sequential representation

- This is the most straightforward technique to store a tree data structure. An array is used to store the tree nodes.
- The number of nodes in a tree defines the size of the array.
- The root node of the tree is held at the first index in the array.
- In general, if a node is stored at the **i^{th}** location, then its **left** and **right** child are kept at **$(2i)^{th}$** and **$(2i+1)^{th}$** locations in the array, respectively.

Consider the following binary tree:



The array representation of the above binary tree is as follows:



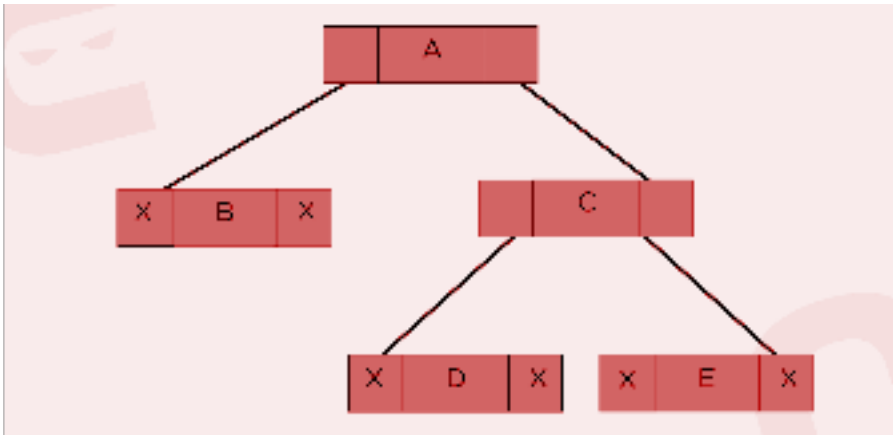
As discussed above, we see that the left and right child of each node is stored at locations **2*(nodePosition)** and **2*(nodePosition)+1**, respectively.

For Example, The location of node 3 in the array is 3. So its left child will be placed at **2*3 = 6**. Its right child will be at the location **2*3 +1 = 7**. As we can see in the array, children of 3, which are 6 and 7, are placed at locations 6 and 7 in the array.

Note: The sequential representation of the tree is not preferred due to the massive amount of memory consumption by the array.

Linked list representation:

In this type of model, a linked list is used to store the tree nodes. The nodes are connected using the parent-child relationship like a tree. The following diagram shows a linked list representation for a tree.

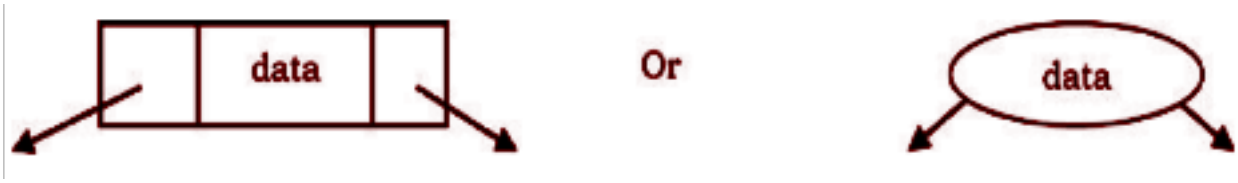


As shown in the above representation, each linked list node has three components:

- Pointer to the left child
- Data
- Pointer to the right child

Note: If there are no children for a given node (leaf node), then the left and right pointers for that node are set to **null**.

Let's now check the implementation of the **Binary tree class**.




```
class BinaryTreeNode<T> {
      T data;                // To store data
      BinaryTreeNode left; // for storing the reference to left pointer
      BinaryTreeNode right; // for storing the reference to right pointer
            // Constructor
      BinaryTreeNode(T data) {
            this.data = data;    // Initializes data of the node
```

```
            this.left = null;// initializes left and right pointers to null
            this.right = null;
      }
}
```

Operations on Binary Trees

Basic Operations

- Inserting an element into a tree
- Deleting an element from a tree
- Searching for an element
- Traversing the tree

Auxiliary Operations

- Finding the size of the tree
- Finding the height of the tree
- Finding the level which has the maximum sum and many more...

Print Tree Recursively

Let's first write a program to print a binary tree recursively. Follow the comments in the code below:

```
public void printTree(BinaryTreeNode<Integer> root) {
      if (root == null) {        // Base case
            return;
      }
      System.out.print(root.data + ":"); //printing the data at root node
      if (root.left != null) {   // checking if left not null
            System.out.print("L" + root.left.data);
      }

      if (root.right != null) { // checking if right not null
            System.out.print("R" + root.right.data);
      }
      System.out.println();
      printTree(root.left);//Now recursively, call left and right subtrees
      printTree(root.right);
}
```

Input Binary Tree

We will be following the level-wise order for taking input and -1 denotes the **null** pointer.

```
public BinaryTreeNode<Integer> takeInput() {
      System.out.print("Enter data:");
      int rootData = s.nextInt();            // taking data as input
      if (rootData == -1) {        // if the data is -1, means null pointer
            return null;
      }
  // Dynamically create root Node which calls constructor of the same class
      BinaryTreeNode<Integer> root = new BinaryTreeNode<>(rootData);
           // Recursively calling over left subtree
      BinaryTreeNode<Integer> leftChild = takeInput();
           // Recursively calling over right subtree
      BinaryTreeNode<Integer> rightChild = takeInput();
      root.left = leftChild; // now allotting left and right childs to root
      root.right = rightChild;
      return root;
}
```

Count nodes

- Unlike the Generic trees, where we need to traverse the children vector of each node, in binary trees, we just have at most left and right children for each node.
- Here, we just need to recursively call on the right and left subtrees independently with the condition that the node pointer is not null.
- Follow the comments in the upcoming code for better understanding:

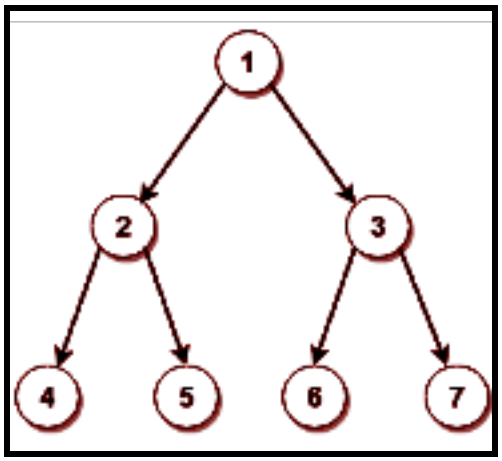
```
public int numNodes(BinaryTreeNode<Integer> root) {
      if (root == null) {    // Condition to check if the node is not null
            return 0;        // counted as zero if so
      }
      return 1 + numNodes(root.left) + numNodes(root.right);
      //recursive calls on left and right subtrees with addition of 1(for
      //counting current node)
}
```

Binary tree traversal

Following are the ways to traverse a binary tree and their orders of traversal:

- **Preorder traversal** : ROOT -> LEFT -> RIGHT
- **Postorder traversal** : LEFT -> RIGHT-> ROOT
- **Inorder traversal** : LEFT -> ROOT -> RIGHT

Some examples of the above-stated traversal methods:



- **Preorder traversal:** 1, 2, 4, 5, 3, 6, 7
- **Postorder traversal:** 4, 5, 2, 6, 7, 3, 1
- **Inorder traversal:** 4, 2, 5, 1, 6, 3, 7

Let's look at the code for inorder traversal, below:

```
public void inorder(BinaryTreeNode<Integer> root) {
      if (root == null) {   // Base case when node's value is null
            return;
      }
      inorder(root.left); //Recursive call over left part as it needs
                          // to be printed first
      System.out.print(root.data);  // Now printed root's data
      inorder(root.right); //Finally recursive call made over right subtree
}
```

Now, from this inorder traversal code, try to code preorder and postorder traversal yourselves. If you get stuck, refer to the solution tab for the same.

Node with the Largest Data

In a Binary Tree, we must visit every node to figure out the maximum. So the idea is to traverse the given tree and for every node return the maximum of 3 values:

- Node's data.
- Maximum in node's left subtree.
- Maximum in node's right subtree.

Below is the implementation of the above approach.

```
public static int findMaximum(root){
    # Base case
    if (root == null):
        return Integer.MAX_VALUE;

    // Return maximum of 3 values:
    // 1) Root's data
    // 2) Max in Left Subtree
    // 3) Max in right subtree
    int max = root.data;
    int lmax = findMaximum(root.left); //Maximum of left subtree
    int rmax = findMaximum(root.right); //Maximum of right subtree
    if (lmax > max)
        max = lmax;
    if (rmax > max)
        max = rmax;
    return max;
}
```

Construct a binary tree from preorder and inorder traversal

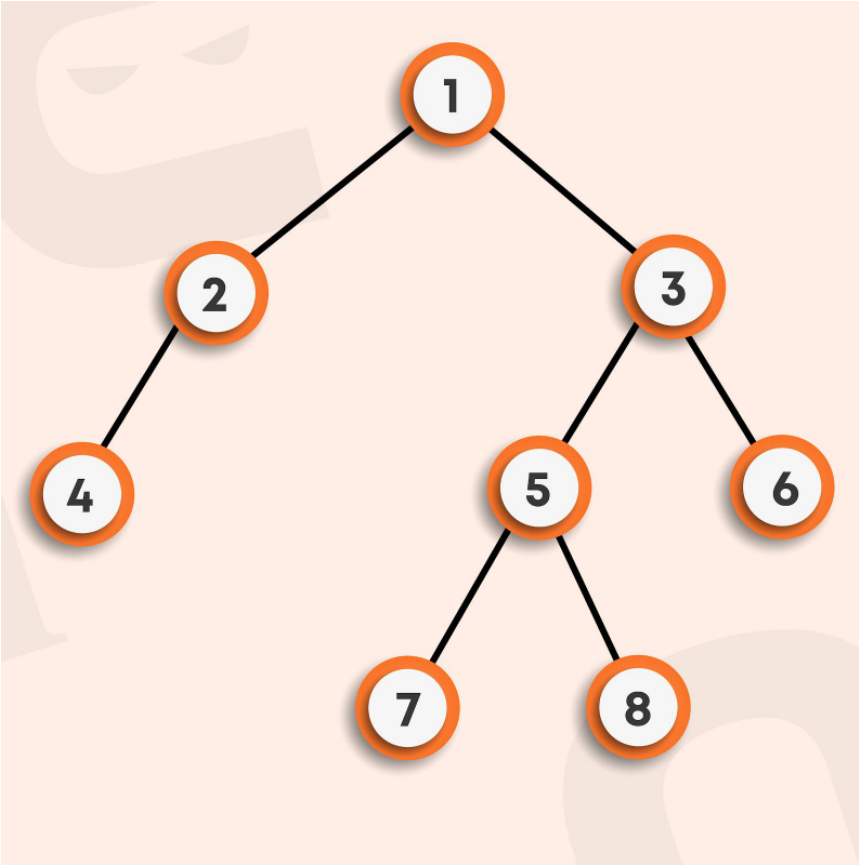
Consider the following example to understand this better.

Input:

Inorder traversal : {4, 2, 1, 7, 5, 8, 3, 6}

Preorder traversal : {1, 2, 4, 3, 5, 7, 8, 6}

Output: Below binary tree...



The idea is to start with the root node, which would be the first item in the preorder sequence and find the boundary of its left and right subtree in the inorder array. Now all keys before the root node in the inorder array become part of the left

subtree, and all the indices after the root node become part of the right subtree. We repeat this recursively for all nodes in the tree and construct the tree in the process.

To illustrate, consider below inorder and preorder sequence-

Inorder: {4, 2, 1, 7, 5, 8, 3, 6}

Preorder: {1, 2, 4, 3, 5, 7, 8, 6}

The root will be the first element in the preorder sequence, i.e. 1. Next, we locate the index of the root node in the inorder sequence. Since 1 is the root node, all nodes before 1 must be included in the left subtree, i.e., {4, 2}, and all the nodes after one must be included in the right subtree, i.e. {7, 5, 8, 3, 6}. Now the problem is reduced to building the left and right subtrees and linking them to the root node.

Left subtree:	**Right subtree:**
Inorder : {4, 2}	**Inorder :** {7, 5, 8, 3, 6}
Preorder : {2, 4}	**Preorder :** {3, 5, 7, 8, 6}

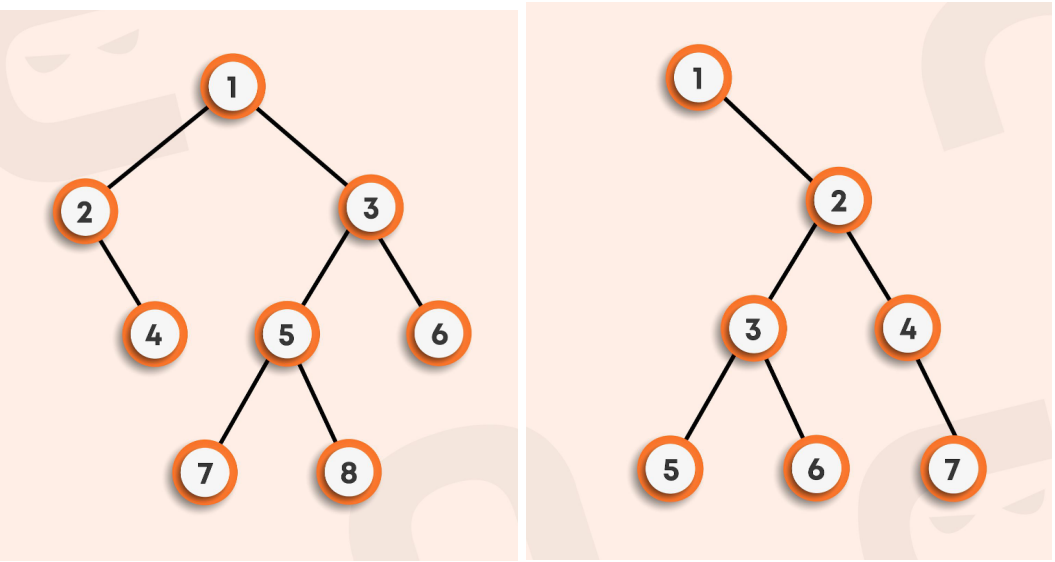
Using the above explanation you can easily create logic and code this. Refer solution tab for solution code for this problem.

Now, try to construct the binary tree when inorder and postorder traversals are given...

The diameter of a binary tree

The diameter of a tree (sometimes called the width) is the number of nodes on the longest path between two child nodes. The diameter of the binary tree may pass through the root (not necessary).

For example, the Below figure shows two binary trees having diameters 6 and 5, respectively (nodes highlighted in blue color). The diameter of the binary tree shown on the left side passes through the root node while on the right side, it doesn't.



There are three possible paths of the diameter:

1. The diameter could be the sum of the left height and the right height.
2. It could be the left subtree's diameter.
3. It could be the right subtree's diameter.

We will pick the one with the maximum value.

Now let's check the code for this...

```
public int height(BinaryTreeNode<Integer> root) {//Func for height of tree
      if (root == null) {
            return 0;
      }
      return 1 + Math.max(height(root.left), height(root.right));
}

public int diameter(BinaryTreeNode<Integer> root) {//calculates diameter
      if (root == null) {            // Base case
            return 0;
      }

      int option1 = height(root.left) + height(root.right); // Option 1
      int option2 = diameter(root.left);                    // Option 2
      int option3 = diameter(root->right);                  // Option 3
      return Math.max(option1, Math.max(option2, option3)); //returns max
}
```

The time complexity for the above approach:

- Height function traverses each node once; hence time complexity will be O(n).
- Option2 and Option3 also traverse on each node, but for each node, we are calculating the height of the tree considering that node as the root node, which makes time complexity equal to O(n*h). (worst case with skewed trees, i.e., a type of binary tree in which all the nodes have only either one child or no child.) Here, h is the height of the tree, which could be $O(n^2)$.

This could be reduced if the height and diameter are obtained simultaneously, which could prevent extra n traversals for each node. To achieve this, move towards the other sections...

The Diameter of a Binary tree: Better Approach

In the previous approach, for each node, we were finding the height and diameter independently, which was increasing the time complexity. In this approach, we will find height and diameter for each node at the same time, i.e., we will store the height and diameter using a pair class where the **first** pointer will be storing the height of that node and the **second** pointer will be storing the diameter. Here, also we will be using recursion.

Let's focus on the **base case**: For a null tree, height and diameter both are equal to 0. Hence, pair class will store both of its values as zero.

Now, moving to **Hypothesis**: We will get the height and diameter for both left and right subtrees, which could be directly used.

Finally, the **induction step**: Using the result of the Hypothesis, we will find the height and diameter of the current node:

Height = max(leftHeight, rightHeight)

Diameter = max(leftHeight + rightHeight, leftDiameter, rightDiameter)

Now we will create a Pair class which will help us do this problem in better complexity.

```
class Pair {
    int first;
    int second;
    public Pair(int first, int second){
        this.first = first;
        this.second = second;
    }
}
```

To access this pair class, we will use **.first** and **.second** pointers.

Follow the code below along with the comments to get a better grip on it...

```
public static Pair heightDiameter(BinaryTreeNode<Integer> root) {
// pair class return-type function
      if (root == null) {              // Base case
            Pair p = new Pair(0, 0);
            // p.first = 0;
            // p.second = 0;
            return p;
      }
      // Recursive calls over left and right subtree
      Pair leftAns = heightDiameter(root.left);
      Pair rightAns = heightDiameter(root.right);
      // Hypothesis step
      // Left diameter, Left height
      int ld = leftAns.second;
      int lh = leftAns.first;
      // Right diameter, Right height
      int rd = rightAns.second;
      int rh = rightAns.first;

      // Induction step
      int height = 1 + Math.max(lh, rh);     // height of current root node
      int diameter = Math.max(lh + rh, Math.max(ld, rd));//diameter of
                                                        // current root node
      Pair p;                  // Pair class for current root node
      p.first = height;
      p.second = diameter;
      return p;
}
```

Now, talking about the time complexity of this method, it can be observed that we are just traversing each node once while making recursive calls and rest all other operations are performed in constant time, hence the time complexity of this program is O(n), where n is the number of nodes.