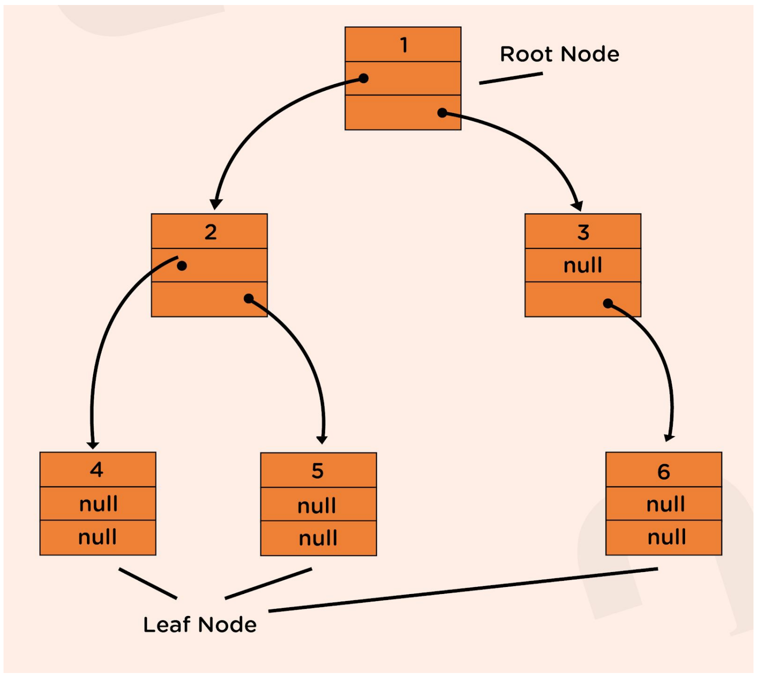
Binary Trees

Introduction

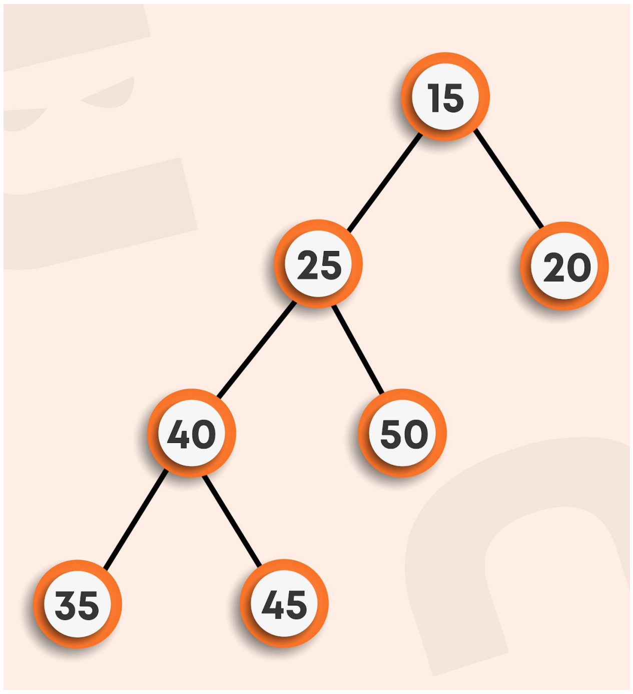
A generic tree with at most two child nodes for each parent node is known as a binary tree.

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. A null pointer represents a binary tree with no elements, i.e., an empty 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**.



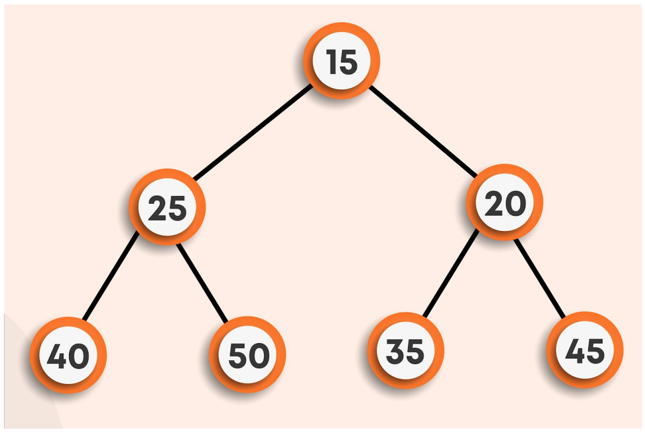
Types of binary trees:

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

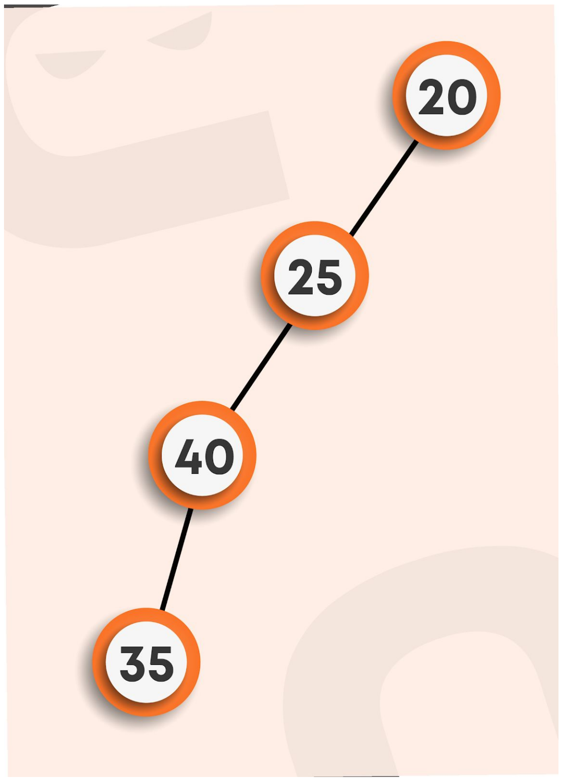


- **Complete binary tree:** A complete binary tree has all the levels filled except for the last level, which has all its nodes as much as to the left.

- **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.

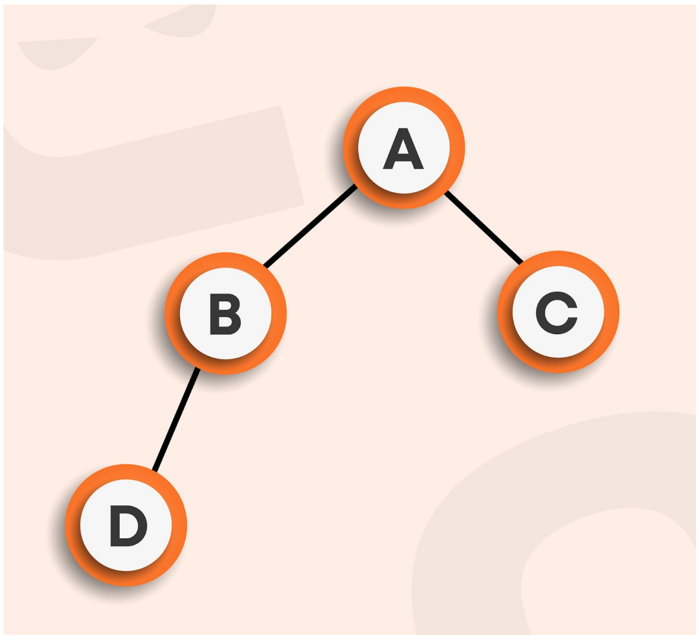


- **A degenerate tree:** In a degenerate tree, each internal node has only one child.



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.



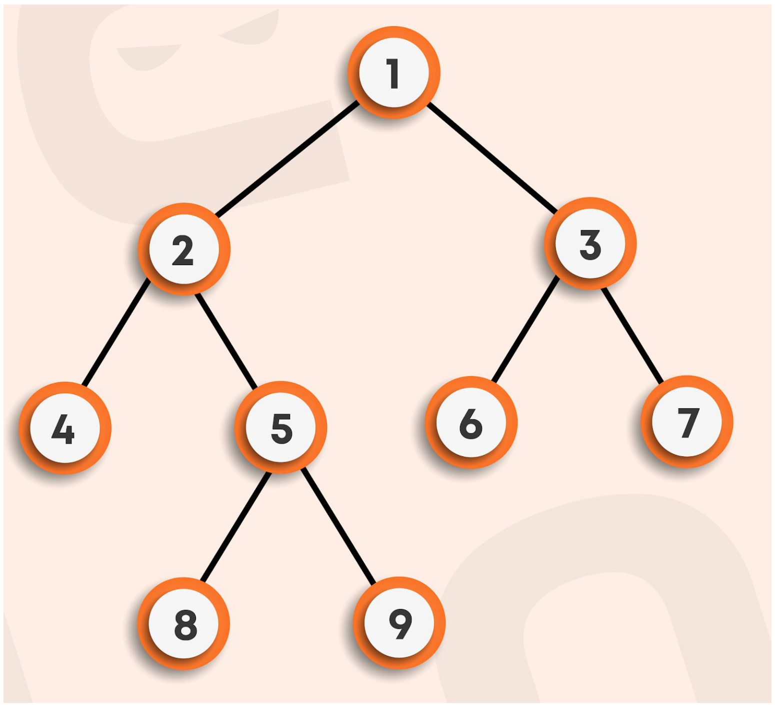
Binary tree representation:

Binary trees can be represented in two ways:

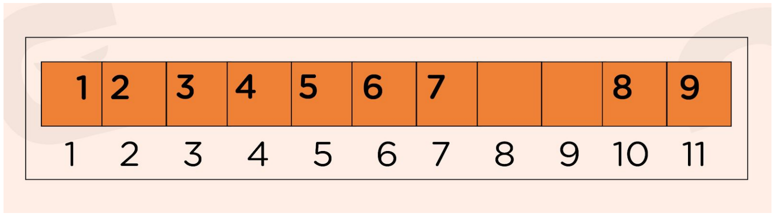
- **Sequential representation:** This is the most straightforward technique to store a tree data structure. An array is used to store the tree nodes. The number of nodes in a tree defines the size of the array. The root node of the tree is held at the first index in the array.

  In general, if a node is stored at the ith location, then its left and right child is kept at 2i and 2i+1 locations, respectively.

Consider the following binary tree:



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



1	2	3	4	5	6	7			8	9
1	2	3	4	5	6	7	8	9	10	11

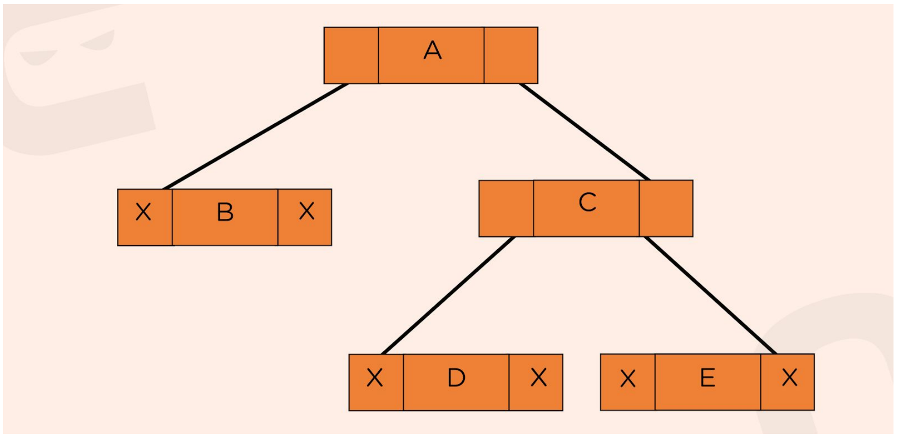
Here, we see that the left and right child of each node is stored at locations 2*(node_position) and 2*(node_position)+1, respectively.

<u>For Example</u>: 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.

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:

- Left pointer
- Data part
- Right pointer

The left pointer has a pointer to the left child of the node; the right pointer has a pointer to the right child of the node whereas the data part contains the actual data of the node. 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 binary tree class. Like TreeNode class, here also we will be creating a separate file with **.h extension** and then use it wherever necessary. (Here the name of the file will be: **BinaryTreeNode.h**)

```
template <typename T>
```

```
class BinaryTreeNode {
        public:
        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
                left = NULL;                    // initializes left and right pointers to NULL
                right = NULL;
        }
        // Destructor
        ~BinaryTreeNode() {
                delete left;                    // Deletes the left pointer
                delete right;                   // Deletes the right pointer
        }                                       // As it ends, deletes the node itself
};
```

Take input and print recursively

Let's first check on printing the binary tree recursively. Follow the comments in the code below...

```
void printTree(BinaryTreeNode<int>* root) {
        if (root == NULL) {             // Base case
                return;
        }
        cout << root->data << ":";  // printing the data at root node
        if (root->left != NULL) {      // checking if left not NULL, then print it's data also
                cout << "L" << root->left->data;
        }

        if (root->right != NULL) { // checking if right not NULL, then print it's data also
                cout << "R" << root->right->data;
        }
        cout << endl;
        printTree(root->left);    // Now recursively, call on the left and right subtrees
        printTree(root->right);
}
```

Now, let’s check the input function: (We will be following the level-wise order for taking input and -1 denotes the NULL pointer or simply, it means that a pointer over the same place is a NULL pointer.

```
BinaryTreeNode<int>* takeInput() {
        int rootData;
        cout << "Enter data" << endl;
        cin >> rootData;                        // taking data as input
        if (rootData == -1) {                   // if the data is -1, means NULL pointer
                return NULL;
        }
       // Dynamically create the root Node which calls constructor of the same class
        BinaryTreeNode<int>* root = new BinaryTreeNode<int>(rootData);
       // Recursively calling over left subtree
        BinaryTreeNode<int>* leftChild = takeInput();
       // Recursively calling over right subtree
        BinaryTreeNode<int>* rightChild = takeInput();
        root->left = leftChild;     // now allotting left and right childs to the root node
        root->right = rightChild;
        return root;
}
```

Taking input iteratively

We have already discussed a recursive approach for taking input in a binary tree in a level order fashion. Now, let’s discuss the iterative procedure for the same using queue as we did in Generic trees. Follow the code along with the comments...

```
BinaryTreeNode<int>* takeInputLevelWise() {
        int rootData;
        cout << "Enter root data" << endl;
        cin >> rootData;                          // Taking node's data as input
        if (rootData == -1) {                     // As -1 refers to NULL pointer
                return NULL;
        }
        // Dynamic allocation of root node
        BinaryTreeNode<int>* root = new BinaryTreeNode<int>(rootData);
        // Using queue to level wise traversal for iterative approach
        queue<BinaryTreeNode<int>*> pendingNodes;
        pendingNodes.push(root);              // root node pushed into the queue
        while (pendingNodes.size() != 0) {     // process continued until the size of queue ≠ 0
                BinaryTreeNode<int>* front = pendingNodes.front();//front of queue is stored
                pendingNodes.pop();
                cout << "Enter left child of " << front->data << endl;
                int leftChildData;
                cin >> leftChildData;       // Left child of current node is taken input
                if (leftChildData != -1) {    // If the value of left child is not -1 that is, not NULL
                        BinaryTreeNode<int>* child = new BinaryTreeNode<int>(leftChildData);
                        front->left = child;             // Value assigned to left part and then pushed
                        pendingNodes.push(child);   // to the queue
                }
                // Similar work is done for right subtree
                cout << "Enter right child of " << front->data << endl;
                int rightChildData;
                cin >> rightChildData;
                if (rightChildData != -1) {
                        BinaryTreeNode<int>* child = new BinaryTreeNode<int>(rightChildData);
                        front->right = child;
                        pendingNodes.push(child);
                }
        }
        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. Kindly follow the comments in the upcoming code...

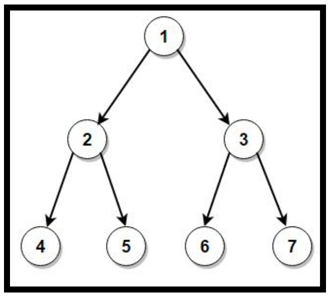
```
int numNodes(BinaryTreeNode<int>* root) {
        if (root == NULL) {                    // Condition to check if the node is not NULL
                return 0;                       // counted as zero if so
        }
        return 1 + numNodes(root->left) + numNodes(root->right);   // recursive 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:

- **Level order traversal**: Moving on each level from left to right direction
- **Preorder traversal** : ROOT -> LEFT -> RIGHT
- **Postorder traversal** : LEFT -> RIGHT-> ROOT
- **Inorder traversal** : LEFT -> ROOT -> RIGHT

Some examples of the above-stated traversal methods:



- **Level order traversal:** 1, 2, 3, 4, 5, 6, 7
- **Preorder traversal:** 1, 2, 4, 5, 3, 6, 7
- **Post order 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:

```
void inorder(BinaryTreeNode<int>* 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
        cout << root->data << " ";                 // Now printed root's data
        inorder(root->right);       // Finally a recursive call made over right subtree
}
```

Now, from this inorder traversal code, try to code preorder and postorder traversal yourselves. For the answer, refer to the solution tab for the same.

Construct a binary tree from preorder and inorder traversal

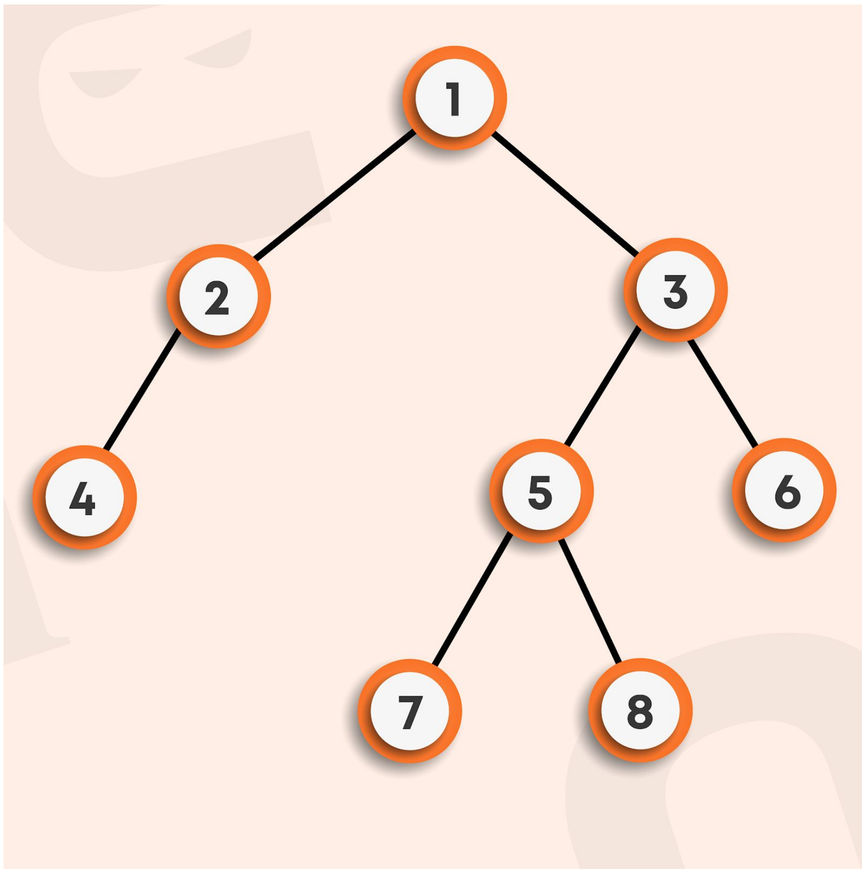
Consider the following example to understand this better.

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}

Follow the above approach until the complete tree is constructed. Now let us look at the code for this problem:

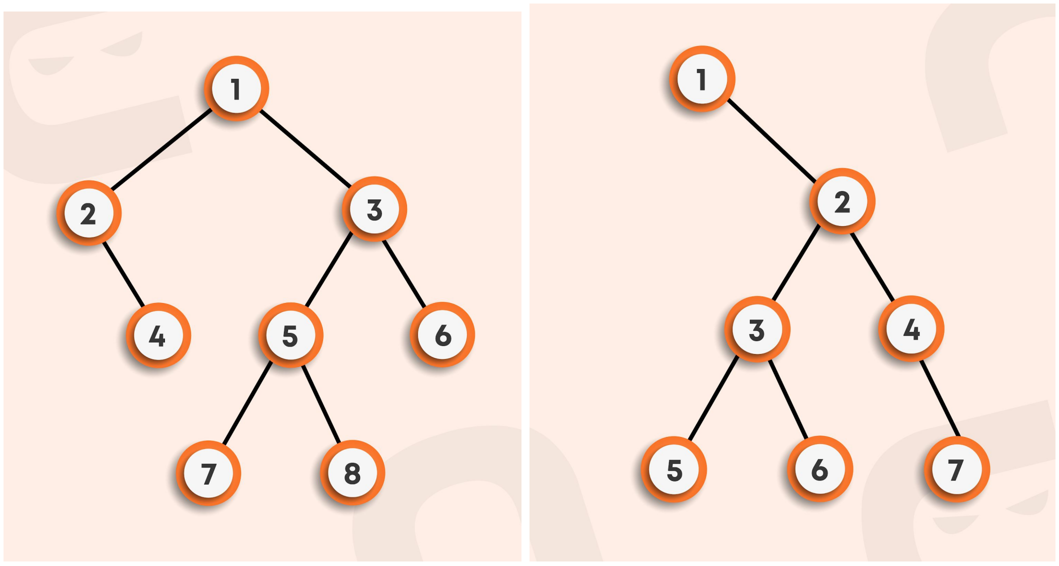
```
BinaryTreeNode<int>* buildTreeHelper(int* in, int* pre, int inS, int inE, int preS, int preE) {
        if (inS > inE) {                        // Base case
                return NULL;
        }

        int rootData = pre[preS];   // Root's data will be first element of the preorder array
        int rootIndex = -1;         // initialised root's index to -1 and searched for it's value
        for (int i = inS; i <= inE; i++) {      // in inorder list
                if (in[i] == rootData) {
                        rootIndex = i;
                        break;
                }
        }
        // Initializing the left subtree's indices for recursive call
        int lInS = inS;
        int lInE = rootIndex - 1;
        int lPreS = preS + 1;
        int lPreE = lInE - lInS + lPreS;
        // Initializing the right subtree's indices for recursive call
        int rPreS = lPreE + 1;
        int rPreE = preE;
        int rInS = rootIndex + 1;
        int rInE = inE;
        // Recursive calls follows
        BinaryTreeNode<int>*  root = new BinaryTreeNode<int>(rootData);
        root->left = buildTreeHelper(in, pre, lInS, lInE, lPreS, lPreE);   // over left subtree
        root->right = buildTreeHelper(in, pre, rInS, rInE, rPreS, rPreE); // over right subtree
        return root;    // finally returned the root
}
BinaryTreeNode<int>* buildTree(int* in, int* pre, int size) {    // this is the function called
 // from the main() with inorder and preorder traversals in the form of arrays and their
// size which is obviously same for both
        return buildTreeHelper(in, pre, 0, size - 1, 0, size - 1);  // These arguments are of the
   // form (inorder_array, preorder_array, inorder_start, inorder_end, preorder_start,
    // preorder_end) in the helper function for the same written above.
}
```

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...

```
int height(BinaryTreeNode<int>* root) {   // Function to calculate height of tree
        if (root == NULL) {
                return 0;
        }
        return 1 + max(height(root->left), height(root->right));
}

int diameter(BinaryTreeNode<int>* root) {    // Function for calculating 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 max(option1, max(option2, option3));   // returns the maximum value
}
```

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)

Note: C++ provides an in-built pair class, which prevents us from creating one of our own. The Syntax for using pair class:

```
pair<datatype1, datatype2> name_of_pair_class;
```

For example: To create a pair class of int, int with a name p, follow the syntax below:

pair<int, int> p;

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...

```
pair<int, int> heightDiameter(BinaryTreeNode<int>* root) {
                                          // pair class return-type function
      if (root == NULL) {           // Base case
            pair<int, int> p;
            p.first = 0;
            p.second = 0;
            return p;
      }
      // Recursive calls over left and right subtree
      pair<int, int> leftAns = heightDiameter(root->left);
      pair<int,int> 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 + max(lh, rh);                    // height of current root node
      int diameter = max(lh + rh, max(ld, rd));   // diameter of current root node
      pair<int, int> 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.

Practice problems:

- https://www.hackerrank.com/challenges/tree-top-view/problem
- https://www.codechef.com/problems/BTREEKK
- https://www.spoj.com/problems/TREEVERSE/
- https://www.hackerearth.com/practice/data-structures/trees/binary-and-nary-trees/practice-problems/approximate/largest-cycle-in-a-tree-9113b3ab/