

# Tree -

🕒 Created time	@March 15, 2023 7:40 PM
🕒 Last edited time	@March 16, 2023 7:57 PM
⚙️ Status	Completed
☰ URL	

**Definition** - Trees are used to represent hierarchical relationships between elements such as in file systems where directories contain files and sub-directories. Trees are also used in computer science algorithms such as binary search trees and heaps

## Some real-world examples of a tree structure -

1. Family Tree: One of the most common examples of a tree data structure is a family tree, which represents relationships between different generations of a family, including grandparents, parents, children, siblings, etc.
2. File System: Another example of a tree structure we use daily is a file system. In a file system, directories or folders are structured as a tree
3. Organization Structure: An organization's structure is another example of a hierarchy that can be represented using trees
4. Databases: Data structures such as trees, heaps, and hash tables are used in databases to store and retrieve data efficiently.

## Terminology

We can categorize all the terms in two for a better understanding

### Technical Terms:

- Tree
- Node
- Degree
- Edge
- Leaf node

- Internal node
- Root node
- Subtree
- Level
- Height
- Depth
- Forest

**Relational Terms:**

- Ancestor
- Descendant
- Parent
- Child
- Sibling
- Grandparent
- Grandchild

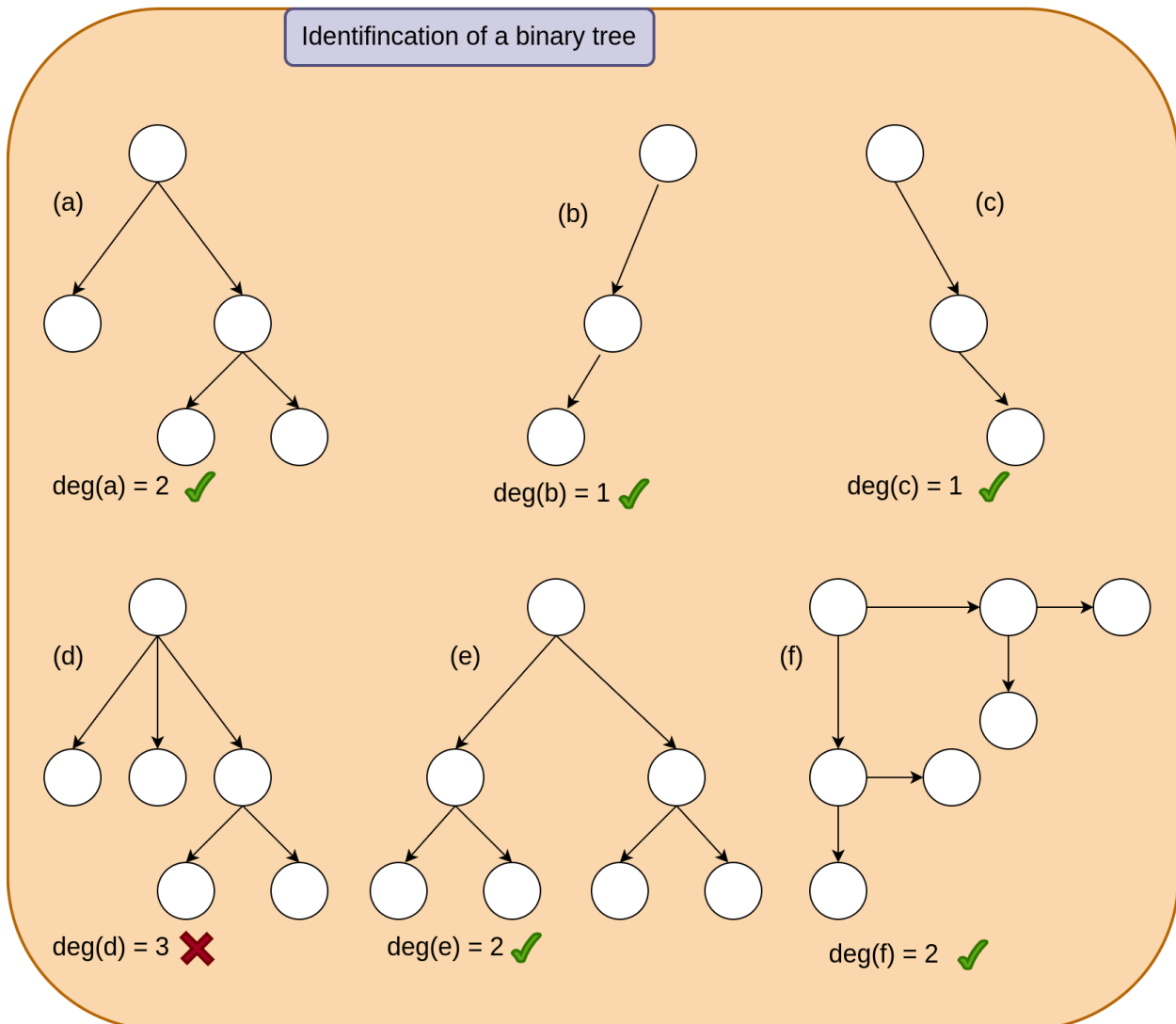
1. Tree(alt definition) - A tree is a collection of **nodes**/vertices and **edges**.



3. Edge - edges are the connection between the pointer and the pointed node. so the number of edges of each node is equal to the number of nodes that the node is pointing to. and the total number of edges in a tree is one less than the total nodes since the topmost node doesn't have any node pointing to it.
4. Degree - degree of a node is the number of children that it has and the degree of a tree is the max number of children each node has or can have.
5. Child nodes - for each node, the nodes it's pointing to are called the children of that node. And all children of that child node are known grandchildren and great-grandchildren recursively and the collection of all the children, grand and great-grandchildren are called **descendants** of that node.
6. Parent node - for each node the node that is pointing it is called the parent node of that child node and just like the child node every node in the hierarchy is called grand and great-grandparents recursively. And the collection of such nodes is called **ancestors**.
7. Sibling nodes - for each node the nodes which have the same parent are called the siblings of that particular node.
8. Leaf/External/Terminal nodes - the nodes that do not have any children or the nodes whose degree is zero are called leaf nodes, those are at the end of the hierarchy
9. Internal/Non-Terminal nodes - every node which is not a leaf node means the nodes which has at least one child are internal nodes.
10. root node - the topmost node in the tree which doesn't has any parent is called the root of that tree,
11. Subtree - since the tree is a recursive data structure each node(including the leaf node) can be visualized as a tree itself and for which the root will be that node.
12. level - level starts from 0(many people start from 1) from the root node and as we go down in a path we increment the level with each node.
13. height - the height of a tree is the max level + 1.
14. depth - depth is calculated from the perspective of a node as to how many edges will require to reach the leaf node of that path.
15. Forest - A collection of trees is called a Forest.

## Binary tree

Trees in which each node can have a max of two nodes are called the Binary tree. so the degree of the tree should be 2. and since we have only two nodes we can give a name to the child as the left child and the right child.






### Observation

we can create some relations/formulae by observing binary trees, these relationships help in analyzing space and time complexity and help understand the data structure better.

For unlabelled nodes -



The number of different trees we can make with n number of nodes.

Number of nodes n	Number of trees
1	
2	
3	

which turn out to be a series called Catalan number which has two formula

i.e. (combination formula)  $\Rightarrow \frac{2^n C_n}{(n+1)}$

and (recursive formula)  $\Rightarrow n \sum_{i=1}^n T(i-1) * T(n-i)$



The max height of a binary tree with n nodes is n-1 so how many trees we can generate with n nodes of max-height

with 1 node = 1

with 2 nodes = 2

with 3 nodes = 4

with 4 nodes = 16

so that looks like  $2^{n-1}$

For labelled nodes -



The number of different trees we can make with n number of nodes.

we know we can create  $\frac{2^n C_n}{(n+1)}$  different trees now in labelled nodes we can create n! variation of each tree so the total nodes formula will look like  $\frac{2^n C_n}{(n+1)} * n!$ .



Here are some more relations and formulas between the nodes and the height of a binary tree:

1. The minimum number of nodes in a binary tree of height h is h+1.
2. The maximum number of nodes in a binary tree of height h is  $2^{(h+1)} - 1$ .
3. The height of a binary tree with n nodes is at most  $\log_2(n+1) - 1$ .
4. The maximum number of nodes at depth d in a binary tree of height h is  $2^{(h-d)}$ .
5. The minimum height of a binary tree with n nodes is  $\text{ceil}(\log_2(n+1)) - 1$ , where  $\text{ceil}(x)$  is the smallest integer greater than or equal to x.
6. The maximum height of a binary tree with n nodes is n-1.
7. If a binary tree of height h has exactly k leaf nodes, then the minimum number of nodes it can have is  $k + (k-1)/h$ .
8. If a binary tree of height h has exactly k leaf nodes, then the maximum number of nodes it can have is  $2k - 1$ .
9. If a binary tree of height h has exactly k leaf nodes, then the average number of nodes it can have is  $(2k - 1 + (2^h - k))/2$ .



Here are some relations and formulas between the internal and external nodes of a binary tree:

1. The number of external nodes in a binary tree with  $n$  nodes is  $n+1$ .
2. The number of internal nodes in a binary tree with  $n$  nodes is  $n-1$ .
3. The total number of nodes in a binary tree with  $i$  internal nodes and  $e$  external nodes is  $i+e$ .
4. The maximum number of external nodes in a binary tree with  $i$  internal nodes is  $i+1$ .
5. The minimum number of external nodes in a binary tree with  $i$  internal nodes is 1 when the binary tree is a linear chain.
6. The maximum number of internal nodes in a binary tree with  $e$  external nodes is  $e-1$ .
7. The minimum number of internal nodes in a binary tree with  $e$  external nodes is 0 when the binary tree has only one node.
8. The total number of nodes in a full binary tree with height  $h$  is  $2^{(h+1)} - 1$ , and half of them are external nodes and half of them are internal nodes.
9. The total number of nodes in a complete binary tree with height  $h$  and  $n$  internal nodes is  $2n+1$ .

## Types of Binary Tree

### Complete/Strict BT

all levels are completely filled except the last level and in the last level, all the nodes should be on the left side of the tree. or we can say if the height of a tree is ' $h$ ' then until ' $h-1$ ' every node will have 0 or 2 children. and in the last level, all nodes should be left skewed.

1. In a strict binary tree of height  $h$ , the number of nodes is  $2^{(h+1)} - 1$ .
2. In a strict binary tree of  $n$  nodes, the height is  $\text{floor}(\log_2(n))+1$ .
3. In a strict binary tree, the maximum height is  $n-1$ , where  $n$  is the number of nodes.



4. In a strict binary tree of height  $h$ , the maximum number of nodes is  $2^{(h+1)} - 1$ .
5. In a strict binary tree of height  $h$ , the number of external nodes (i.e., leaves) is  $2^h$ .
6. In a strict binary tree of height  $h$ , the number of internal nodes is  $2^h - 1$ .
7. In a strict binary tree of height  $h$ , the minimum number of nodes is  $2^{(h+1)} - 1$ . This occurs when the tree is a complete binary tree.

## Full BT

every node will have either 0 or 2 children. so a full binary tree is always a complete binary tree

1. In a full binary tree of height  $h$ , the number of nodes is  $2^{(h+1)} - 1$ .
2. In a full binary tree of  $n$  nodes, the height is  $\text{floor}(\log_2(n+1))$ .
3. In a full binary tree of height  $h$ , the minimum number of nodes is  $2^h$ , and it occurs when all levels, except possibly the last, are completely filled.
4. In a full binary tree of  $n$  nodes, the maximum height is  $\log_2(n+1) - 1$ .
5. In a full binary tree, the maximum number of nodes at any given level  $i$  is  $2^i$ .
6. In a full binary tree of height  $h$ , the number of external nodes (i.e., leaves) is at most  $2^h$  and at least  $2^{(h-1)}$ .
7. In a full binary tree, the number of internal nodes is at most  $2^h - 1$  and at least  $2^{(h-1)} - 1$ .

## Perfect BT

when all the leaf nodes are at the same level.



## Balanced BT

when then the height is  $\leq \log(n)$  where  $n$  is the number of nodes.



## Degenerate Tree

when the height of the tree == the number of nodes(essentially this is a linked list).



## Traversal of Binary Tree

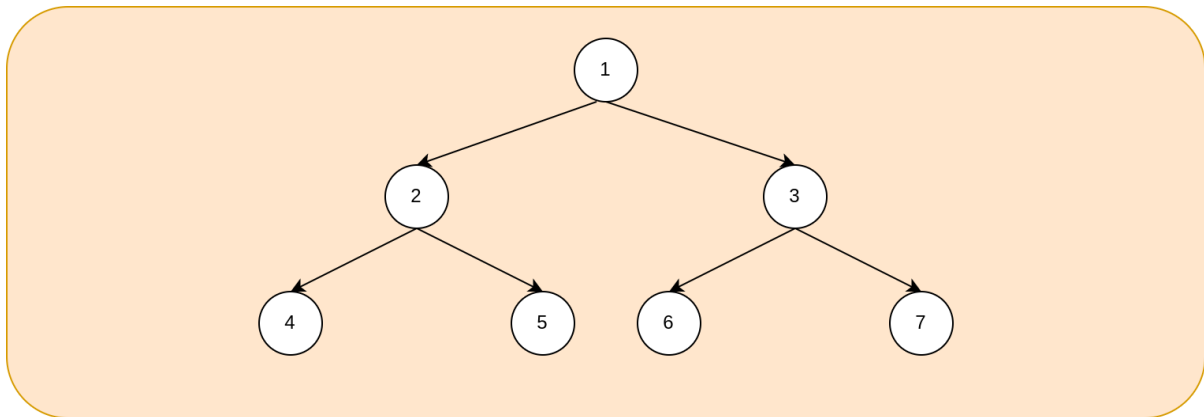
In order to solve any problem with the tree we will require to traverse the tree, there are two techniques i.e. depth-first- search (DFS). and the breadth-first search (BFS)

the output will look like this = 1 2 3 4 5 6 7

### DFS -

there are three types of DFS technique known as in-order, pre-order and post-order which is based on which order we are gonna traverse the tree. in in-order we go to the left-most subtree until we can't go more left after we go to the right-most sub tree when we find the most left then right we apply Left-Root-Right in each node as we were doing for the subtree, and similarly for Pre-order and post-order too.

traversing a tree in these orders we will get



- inorder: inorder(Left) → visit(node) → inorder(Right) = 4 2 5 1 6 3 7
- preorder: visit(node) → preorder(Left) → preorder(Right) = 1 2 4 5 3 6 7
- postorder: postorder(Left) → postorder(Right) → visit(node) = 4 5 2 6 7 3 1

## Pre-order traversal

1. Check if the current node is null.
2. If not, visit the current node and print its value.
3. Recursively traverse the left subtree by calling the pre-order function on the left child of the current node.
4. Recursively traverse the right sub-tree by calling the pre-order function on the right child of the current node.

```
// This method performs a preorder traversal of a binary tree
public static void preOrder(Node root){
    if(root != null){
        // Process the current node
        System.out.println(root.data);
        // Traverse the left subtree recursively
        preOrder(root.left);
        // Traverse the right subtree recursively
        preOrder(root.right);
    }
}
```

For example, consider a binary tree with root node A, left child B, and right child C. The pre-order traversal would be A -> B -> C.

1. Visit A and print its value.
2. Call the pre-order function on B.
3. Visit B and print its value.
4. Call the pre-order function on C.
5. Visit C and print its value.

The final output would be: A -> B -> C

## Post-order traversal

1. Check if the current node is null.
2. Recursively traverse the left subtree by calling the post-order function on the left child of the current node.
3. Recursively traverse the right subtree by calling the post-order function on the right child of the current node.
4. If not, visit the current node and print its value.

```
public static void postOrder(Node root){
    if(root != null){
        // Traverse the left subtree recursively
        postOrder(root.left);
        // Traverse the right subtree recursively
        postOrder(root.right);
        // Process the current node
        System.out.println(root.data);
    }
}
```

For example, consider a binary tree with root node A, left child B, and right child C. The post-order traversal would be B -> C -> A.

1. Call the post-order function on B.
2. Call post-order function on C.
3. Visit A and print its value.

4. Visit B and print its value.
5. Visit C and print its value.

The final output would be: B -> C -> A

## In-order traversal

1. Check if the current node is null.
2. Recursively traverse the left subtree by calling the in-order function on the left child of the current node.
3. If not, visit the current node and print its value.
4. Recursively traverse the right subtree by calling the in-order function on the right child of the current node.

```
public static void inOrder(Node root){  
    if(root != null){  
        // Traverse the left subtree recursively  
        inOrder(root.left);  
        // Process the current node  
        System.out.println(root.data);  
        // Traverse the right subtree recursively  
        inOrder(root.right);  
    }  
}
```

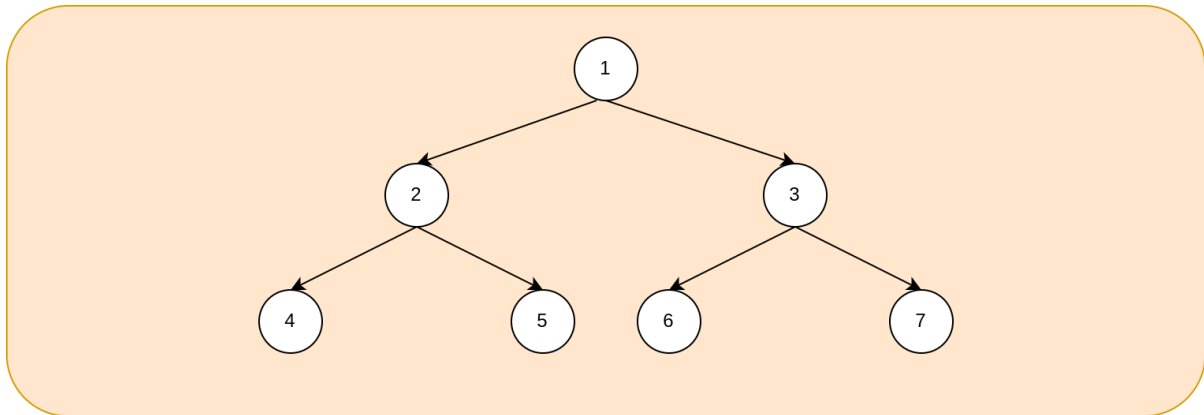
For example, consider a binary tree with root node A, left child B, and right child C. The in-order traversal would be B -> A -> C.

1. Call in-order function on B.
2. Visit A and print its value.
3. Call in-order function on C.
4. Visit B and print its value.
5. Visit C and print its value.

The final output would be: B -> A -> C

## BFS -

in the breadth-first search we go to each 'level' and print every node like in the tree:



## Level-order Traversal

```
public List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> answer = new ArrayList<>();
    if(root == null) return answer;
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    while(!queue.isEmpty()){
        int sizeOfCurrentQueue = queue.size();
        List<Integer> list = new ArrayList<>();
        for(int i = 0; i < sizeOfCurrentQueue; i++){
            if (queue.peek() != null && queue.peek().left != null) queue.offer(queue.p
            eek().left);
            if (queue.peek() != null && queue.peek().right != null) queue.offer(queue.
            peek().right);
            list.add(Objects.requireNonNull(queue.poll()).val);
        }
        answer.add(list);
    }
    return answer;
}
```

For level order, we're gonna use an iterative approach, because that will be more intuitive for level order.

since in recursive many things are automatically done by recursion we have to do it manually here we will create a queue to store the references for traversal.

1. create a queue and add the first(root) node.
2. visit every node in the queue waiting to be printed.
3. and as we print the node add their right and left nodes if they are not null so that in the next iteration we have a new list of nodes to print.

4. and if the queue becomes empty means during the last iteration we couldn't find any non-null node on the right or left side of any node in the queue so the loop will end here and we have our answer.