

# Introduction to Tree Data Structure - GeeksforGeeks

**Source:** <https://www.geeksforgeeks.org/introduction-to-tree-data-structure/>

Courses Tutorials Practice Jobs DSA Tutorial Interview Questions Quizzes Must Do Advanced DSA System Design Aptitude Puzzles Interview Corner DSA Python Technical Scripter 2026 Explore DSA Fundamentals Logic Building Problems Analysis of Algorithms Data Structures Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Tutorial Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Segment Tree Binary Indexed Tree or Fenwick Tree Square Root (Sqrt) Decomposition Algorithm Binary Lifting Geometry Interview Preparation Interview Corner GfG160 Practice Problem GeeksforGeeks Practice - Leading Online Coding Platform Problem of The Day - Develop the Habit of Coding DSA Course 90% Refund Introduction to Tree Data Structure Last Updated : 7 Oct, 2025 A tree is a hierarchical data structure used to organize and represent data in a parent-child relationship. It consists of nodes, where the topmost node is called the root, and every other node can have one or more child nodes. Basic Terminologies In Tree Data Structure: Parent Node: A node that is an immediate predecessor of another node. Example: 35 is the parent of 3 and 6. Child Node: A node that is an immediate successor of another node. Example: 3 and 6 are children of 35. Root Node: The topmost node in a tree, which does not have a parent. Example: 15 is the root node. Leaf Node (External Node): Nodes that do not have any children. Example: 1, 10, 12, 5, 7, 7 are leaf nodes. Ancestor: Any node on the path from the root to a given node (excluding the node itself). Example: 15 and 35 are ancestors of 10. Descendant: A node x is a descendant of another node y if y is an ancestor of x. Example: 1, 10, and 6 are descendants of 35. Sibling: Nodes that share the same parent. Example: 1 and 10 are siblings, and 5 and 7 are siblings. Level of a Node: The number of edges in the path from the root to that node. The root node is at level 0. Internal Node: A node with at least one child. Neighbor of a Node: The parent or children of a node. Subtree: A node and all its descendants form a subtree. Why Tree is considered a non-linear data structure? Data in a tree is not stored sequentially (i.e., not in a linear order). Instead, it is organized across multiple levels, forming a hierarchical structure. Because of this arrangement, a tree is classified as a non-linear data structure. Representation of a Node in Tree Data Structure: A tree can be represented using a collection of nodes. Each of the nodes can be represented with the help of class or structs.

```
C++ // Node structure for tree class Node { public : int data ; vector < Node * > children ; Node ( int x ) { data = x ; } } ; Java // Node structure for tree class Node { int data ; List < Node > children ; Node ( int x ) { data = x ; children = new ArrayList <> ( ) ; } } Python # Node structure for tree class Node : def __init__ ( self , x ) : self . data = x self . children = [ ] C# class Node { public int data ; public List < Node > children ; public Node ( int x ) { data = x ; children = new List < Node > ( ) ; } } JavaScript // Node structure for tree class Node { constructor ( x ) { this . data = x ; this . children = [ ] ; } }
```

Importance of Tree Data Structure: Trees are useful for storing data that naturally forms a hierarchy. File systems on computers are structured as trees, with folders containing subfolders and files. The DOM (Document Object Model) of an HTML page is a tree: The `<html>` tag is the root. `<head>` and `<body>` are its children. These tags can have their own child nodes, forming a hierarchical structure. Trees help in efficient data organization and retrieval for hierarchical relationships.

Types of Tree Data Structures : Tree data structures can be classified based on the number of children each node can have.

- Binary Tree:** Each node can have a maximum of two children.
- Full Binary Tree –** every node has either 0 or 2 children.
- Complete Binary Tree –** all levels are fully filled except possibly the last, which is filled from left to right.
- Balanced Binary Tree –** height difference between left and right subtrees of every node is minimal.
- Ternary Tree:** Each node can have at most three children, often labeled as left, middle, and right.
- N-ary Tree (or Generic Tree):** Each node can have any number of children. Each node typically contains: Some data, A list of references to its children (duplicates are not allowed). Unlike linked lists, nodes store references to multiple child nodes. Please refer Types of Trees in Data Structures for details.

Basic Operations Of Tree Data Structure:

- Create –** create a tree in the data structure.
- Insert –** Inserts data in a tree.
- Search –** Searches specific data in a tree to check whether it is present or not.

Traversal :

- Depth-First-Search Traversal
- Breadth-First-Search Traversal

```
C++ #include <iostream>
#include <vector>
using namespace std ; // Node structure for tree class Node { public : int data ; vector
```

```

< Node *> children ; Node ( int x ) { data = x ; } ; // Function to add a child to a node void addChild ( Node * parent , Node * child ) { parent -> children . push_back ( child ); } // Function to print parents of each node void printParents ( Node * node , Node * parent ) { if ( parent == nullptr ) cout << node -> data << " -> NULL" << endl ; else cout << node -> data << " -> " << parent -> data << endl ; for ( auto child : node -> children ) printParents ( child , node ); } // Function to print children of each node void printChildren ( Node * node ) { cout << node -> data << " -> " ; for ( auto child : node -> children ) cout << child -> data << " " ; cout << endl ; for ( auto child : node -> children ) printChildren ( child ); } // Function to print leaf nodes void printLeafNodes ( Node * node ) { if ( node -> children . empty () ) { cout << node -> data << " " ; return ; } for ( auto child : node -> children ) printLeafNodes ( child ); } // Function to print degrees of each node void printDegrees ( Node * node , Node * parent ) { int degree = node -> children . size (); if ( parent != nullptr ) degree ++ ; cout << node -> data << " -> " << degree << endl ; for ( auto child : node -> children ) printDegrees ( child , node ); } int main () { // Creating nodes Node * root = new Node ( 1 ); Node * n2 = new Node ( 2 ); Node * n3 = new Node ( 3 ); Node * n4 = new Node ( 4 ); Node * n5 = new Node ( 5 ); // Constructing tree addChild ( root , n2 ); addChild ( root , n3 ); addChild ( n2 , n4 ); addChild ( n2 , n5 ); cout << "Parents of each node:" << endl ; printParents ( root , nullptr ); cout << "Children of each node:" << endl ; printChildren ( root ); cout << "Leaf nodes: " ; printLeafNodes ( root ); cout << endl ; cout << "Degrees of nodes:" << endl ; printDegrees ( root , nullptr ); return 0 ; } Java import java.util.ArrayList ; import java.util.List ; // Node structure for tree class Node { int data ; List < Node > children ; Node ( int x ) { data = x ; children = new ArrayList <> (); } } class GFG { // Function to add a child to a node static void addChild ( Node parent , Node child ) { parent . children . add ( child ); } // Function to print parents of each node static void printParents ( Node node , Node parent ) { if ( parent == null ) System . out . println ( node . data + " -> NULL" ); else System . out . println ( node . data + " -> " + parent . data ); for ( Node child : node . children ) printParents ( child , node ); } // Function to print children of each node static void printChildren ( Node node ) { System . out . print ( node . data + " -> " ); for ( Node child : node . children ) System . out . print ( child . data + " " ); System . out . println (); for ( Node child : node . children ) printChildren ( child ); } // Function to print leaf nodes static void printLeafNodes ( Node node ) { if ( node . children . isEmpty () ) { System . out . print ( node . data + " " ); return ; } for ( Node child : node . children ) printLeafNodes ( child ); } // Function to print degrees of each node static void printDegrees ( Node node , Node parent ) { int degree = node . children . size (); if ( parent != null ) degree ++ ; System . out . println ( node . data + " -> " + degree ); for ( Node child : node . children ) printDegrees ( child , node ); } public static void main ( String [] args ) { // Creating nodes Node root = new Node ( 1 ); Node n2 = new Node ( 2 ); Node n3 = new Node ( 3 ); Node n4 = new Node ( 4 ); Node n5 = new Node ( 5 ); // Constructing tree addChild ( root , n2 ); addChild ( root , n3 ); addChild ( n2 , n4 ); addChild ( n2 , n5 ); System . out . println ( "Parents of each node:" ); printParents ( root , null ); System . out . println ( "Children of each node:" ); printChildren ( root ); System . out . print ( "Leaf nodes: " ); printLeafNodes ( root ); System . out . println (); System . out . println ( "Degrees of nodes:" ); printDegrees ( root , null ); } } Python import sys # Node structure for tree class Node : def __init__ ( self , x ): self . data = x self . children = [] # Function to add a child to a node def addChild ( parent , child ): parent . children . append ( child ) # Function to print parents of each node def printParents ( node , parent ): if parent is None : print ( str ( node . data ) + " -> NULL" ) else : print ( str ( node . data ) + " -> " + str ( parent . data )) for child in node . children : printParents ( child , node ) # Function to print children of each node def printChildren ( node ): children _str = " " . join ( str ( child . data ) for child in node . children ) print ( str ( node . data ) + " -> " + children _str ) for child in node . children : printChildren ( child ) # Function to print leaf nodes def printLeafNodes ( node ): if not node . children : sys . stdout . write ( str ( node . data ) + " " ) return for child in node . children : printLeafNodes ( child ) # Function to print degrees of each node def printDegrees ( node , parent ): degree = len ( node . children ) if parent is not None : degree += 1 print ( str ( node . data ) + " -> " + str ( degree )) for child in node . children : printDegrees ( child , node ) # Main execution if __name__ == "__main__" : # Creating nodes root = Node ( 1 ) n2 = Node ( 2 ) n3 = Node ( 3 ) n4 = Node ( 4 ) n5 = Node ( 5 ) # Constructing tree addChild ( root , n2 ) addChild ( root , n3 ) addChild ( n2 , n4 ) addChild ( n2 , n5 ) print ( "Parents of each node:" ) printParents ( root , None ) print ( "Children of each node:" ) printChildren ( root ) print ( "Leaf nodes: " ) printLeafNodes ( root ) print ( "\n " ) # Newline after leaf nodes print ( "Degrees of nodes:" ) printDegrees ( root , None ) C# using System ; using System.Collections.Generic ; class Node { public int data ; public List < Node > children ; public Node ( int x ) { data = x ; children = new List < Node > (); } } class GFG { // Function to add a child to a node public static void addChild ( Node parent , Node child ) { parent . children . Add ( child ); } // Function to print parents of each node public static void printParents ( Node node , Node parent ) { if

```

```

( parent == null ) Console . WriteLine ( node . data + " -> NULL" ); else Console . WriteLine ( node .
data + " -> " + parent . data ); foreach ( var child in node . children ) printParents ( child , node ); } // Function to print children of each node public static void printChildren ( Node node ) { Console . Write (
node . data + " -> " ); foreach ( var child in node . children ) Console . Write ( child . data + " " ); Console .
WriteLine (); foreach ( var child in node . children ) printChildren ( child ); } // Function to print leaf nodes public static void printLeafNodes ( Node node ) { if ( node . children . Count == 0 ) { Console .
Write ( node . data + " " ); return ; } foreach ( var child in node . children ) printLeafNodes ( child ); } // Function to print degrees of each node public static void printDegrees ( Node node , Node parent ) { int
degree = node . children . Count ; if ( parent != null ) degree ++ ; Console . WriteLine ( node . data + " ->
" + degree ); foreach ( var child in node . children ) printDegrees ( child , node ); } static void Main (
string [] args ) { // Creating nodes Node root = new Node ( 1 ); Node n2 = new Node ( 2 ); Node n3 =
new Node ( 3 ); Node n4 = new Node ( 4 ); Node n5 = new Node ( 5 ); // Constructing tree addChild ( root ,
n2 ); addChild ( root , n3 ); addChild ( n2 , n4 ); addChild ( n2 , n5 ); Console . WriteLine (
"Parents of each node:" ); printParents ( root , null ); Console . WriteLine ( "Children of each node:" );
printChildren ( root ); Console . Write ( "Leaf nodes: " ); printLeafNodes ( root ); Console . WriteLine ();
Console . WriteLine ( "Degrees of nodes:" ); printDegrees ( root , null ); } } JavaScript // Node structure for tree class Node { constructor ( x ) { this . data = x ; this . children = [] ; } } // Function to add a child to a node function addChild ( parent , child ) { parent . children . push ( child ); } // Function to print parents of each node function printParents ( node , parent ) { if ( parent === null ) console . log ( node . data + " -> root" );
else console . log ( node . data + " -> " + parent . data ); for ( let child of node . children )
printParents ( child , node ); } // Function to print children of each node function printChildren ( node ) {
let str = node . data + " -> " ; for ( let child of node . children ) str += child . data + " " ; console . log ( str );
for ( let child of node . children ) printChildren ( child ); } // Function to print leaf nodes function
printLeafNodes ( node ) { if ( node . children . length === 0 ) { process . stdout . write ( node . data + " " );
return ; } for ( let child of node . children ) printLeafNodes ( child ); } // Function to print degrees of each node function printDegrees ( node , parent ) { let degree = node . children . length ; if ( parent !== null )
degree ++ ; console . log ( node . data + " -> " + degree ); for ( let child of node . children )
printDegrees ( child , node ); } // Driver code let root = new Node ( 1 ); let n2 = new Node ( 2 ); let n3 =
new Node ( 3 ); let n4 = new Node ( 4 ); let n5 = new Node ( 5 ); // Constructing tree addChild ( root , n2 );
addChild ( root , n3 ); addChild ( n2 , n4 ); addChild ( n2 , n5 ); console . log ( "Parents of each node:" );
printParents ( root , null ); console . log ( "Children of each node:" ); printChildren ( root ); process .
stdout . write ( "Leaf nodes: " ); printLeafNodes ( root ); console . log (); console . log ( "Degrees of nodes:" );
printDegrees ( root , null ); Output Parents of each node: 1 -> NULL 2 -> 1 4 -> 2 5 -> 2 3 -> 1
Children of each node: 1 -> 2 3 2 -> 4 5 4 -> 5 -> 3 -> Leaf nodes: 4 5 3 Degrees of nodes: 1 -> 2 2 -> 3
4 -> 1 5 -> 1 3 -> 1 Properties of Tree Data Structure: Number of edges : An edge is the connection between two nodes. A tree with N nodes will always have N - 1 edges. There is exactly one path from any node to any other node in the tree. Depth of a node: The depth of a node is the length of the path from the root to that node. Each edge in the path adds 1 unit to the length. Equivalently, it is the number of edges from the root to the node. Height of the tree: The height of the tree is the length of the longest path from the root to any leaf node. Degree of a node: The degree of a node is the number of subtrees attached to it (i.e., the number of children it has). A leaf node has a degree of 0. The degree of the tree is the maximum degree among all nodes in the tree. Comment Article Tags: Article Tags: Tree Data Structures DSA JavaScript-DSA

```