Level Order Traversal (Breadth First Search) of Binary Tree - GeeksforGeeks

Source: https://www.geeksforgeeks.org/level-order-tree-traversal/

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 Level Order Traversal (Breadth First Search) of Binary Tree Last Updated : 8 Dec, 2025 Level Order Traversal technique is a method to traverse a Tree such that all nodes present in the same level are traversed completely before traversing the next level. Input: Output: [[5], [12, 13], [7, 14, 2], [17, 23, 27, 3, 8, 11]] Explanation: Start with the root - [5] Level 1: [12, 13] Level 2: [7, 14, 2] Level 3: [17, 23, 27, 3, 8, 11] Try it on GfG Practice Table of Content [Approach] Using Recursion - O(n) time and O(n) space [Expected Approach] Using Queue (Iterative) - O(n) time and O(n) space [Approach] Using Recursion - O(n) time and O(n) space The idea is to traverse the tree recursively, starting from the root at level 0. When a node is visited, its value is added to the result array at the index corresponding to its level, and then its left and right children are recursively processed in the same way. This effectively performs a level-order traversal using recursion. C++ #include <iostream> #include <vector> using namespace std ; class Node { public : int data ; Node * left , * right ; // Constructor to initialize a new node Node (int value) { data = value ; left = nullptr ; right = nullptr ; } }; void levelOrderRec (Node * root , int level , vector < vector < int >>& res) { // Base case if (root == nullptr) return ; // Add a new level to the result if needed if (res . size () <= level) res . push_back ({}); // Add current node's data to its corresponding level res [level]. push_back (root -> data); // Recur for left and right children levelOrderRec (root -> left , level + 1 , res); levelOrderRec (root -> right , level + 1 , res); } // Function to perform level order traversal vector < vector < int >> levelOrder (Node * root) { // Stores the result level by level vector < vector < int >> res ; levelOrderRec (root , 0 , res); return res ; } int main () { // 5 // / \ // 12 13 // / \ \ // 7 14 2 // / \ / \ / \ //17 23 27 3 8 11 Node * root = new Node (5); root -> left = new Node (12); root -> right = new Node (13); root -> left -> left = new Node (7); root -> left -> right = new Node (14); root -> right -> right = new Node (2); root -> left -> left -> left = new Node (17); root -> left -> left -> right = new Node (23); root -> left -> right -> left = new Node (27); root -> left -> right -> right = new Node (3); root -> right -> right -> left = new Node (8); root -> right -> right -> right = new Node (11); vector < vector < int >> res = levelOrder (root); for (vector < int > level : res) { for (int val : level) { cout << val << " " ; } cout << endl ; } return 0 ; } Java import java.util.ArrayList ; class Node { int data ; Node left , right ; Node (int value) { data = value ; left = null ; right = null ; } } public class GfG { void levelOrderRec (Node root , int level , ArrayList < ArrayList < Integer >> res) { // Base case if (root == null) return ; // Add a new level to the result if needed if (res . size () <= level) res . add (new ArrayList <> ()); // Add current node's data to its corresponding // level res . get (level). add (root . data); // Recur for left and right children levelOrderRec (root . left , level + 1 , res); levelOrderRec (root . right , level + 1 , res); } // Function to perform level order traversal ArrayList < ArrayList < Integer >> levelOrder (Node root) { // Stores the result level by level ArrayList < ArrayList < Integer >> res = new ArrayList <> (); levelOrderRec (root , 0 , res); return res ; } public static void main (String [] args) { // 5 // / \ // 12 13 // / \ \ // 7 14 2 // / \ / \ / \ //17 23 27 3 8 11 Node root = new Node (5); root . left = new Node (12); root . right = new Node (13); root . left . left = new Node (7); root . left . right = new Node (14); root . right . right = new Node (2); root . left . left . left = new Node (17); root . left . left . right = new Node (23); root . left . right . left = new Node (27); root . left . right . right = new Node (3); root . right . right . left = new Node (8); root . right . right . right = new Node (11); GfG tree = new GfG (); ArrayList < ArrayList < Integer >> res = tree . levelOrder (root); for (ArrayList < Integer > level : res) { for (int val : level) {

System . out . print (val + " "); } System . out . println (); } } } Python class Node : def __init__ (self , value): self . data = value self . left = None self . right = None def levelOrderRec (root , level , res): # Base case if root is None : return # Add a new level to the result if needed if len (res) <= level : res . append ([]) # Add current node's data to its corresponding level res [level] . append (root . data) # Recur for left and right children level_order_rec (root . left , level + 1 , res) level_order_rec (root . right , level + 1 , res) # Function to perform level order traversal def levelOrder (root): # Stores the result level by level res = [] level_order_rec (root , 0 , res) return res if __name__ == '__main__' : # 5 # / \ # 12 13 # / \ \ # 7 14 2 # / \ / \ / \ #17 23 27 3 8 11 root = Node (5) root . left = Node (12) root . right = Node (13) root . left . left = Node (7) root . left . right = Node (14) root . right . right = Node (2) root . left . left . left = Node (17) root . left . left . right = Node (23) root . left . right . left = Node (27) root . left . right . right = Node (3) root . right . right . left = Node (8) root . right . right . right = Node (11) res = level_order (root) for level in res : print (' ' . join (map (str , level))) C# using System ; using System.Collections.Generic ; class Node { public int data ; public Node left , right ; // Constructor to initialize a new node public Node (int value) { data = value ; left = null ; right = null ; } } class GfG { static void levelOrderRec (Node root , int level , List < List < int >> res) { // Base case if (root == null) return ; // Add a new level to the result if needed if (res . Count <= level) res . Add (new List < int > ()); // Add current node's data to its corresponding // level res [level]. Add (root . data); // Recur for left and right children levelOrderRec (root . left , level + 1 , res); levelOrderRec (root . right , level + 1 , res); } // Function to perform level order traversal static List < List < int >> levelOrder (Node root) { // Stores the result level by level List < List < int >> res = new List < List < int >> (); levelOrderRec (root , 0 , res); return res ; } static void Main () { // 5 // / \ // 12 13 // / \ \ // 7 14 2 // / \ / \ / \ //17 23 27 3 8 11 Node root = new Node (5); root . left = new Node (12); root . right = new Node (13); root . left . left = new Node (7); root . left . right = new Node (14); root . right . right = new Node (2); root . left . left . left = new Node (17); root . left . left . right = new Node (23); root . left . right . left = new Node (27); root . left . right . right = new Node (3); root . right . right . left = new Node (8); root . right . right . right = new Node (11); List < List < int > > res = levelOrder (root); // Print level by level foreach (var level in res) { foreach (int val in level) { Console . Write (val + " "); } Console . WriteLine (); } } } JavaScript class Node { constructor (value) { this . data = value ; this . left = null ; this . right = null ; } } function levelOrderRec (root , level , res) { // Base case if (root === null) return ; // Add a new level to the result if needed if (res . length <= level) res . push ([]); // Add current node's data to its corresponding level res [level]. push (root . data); // Recur for left and right children levelOrderRec (root . left , level + 1 , res); levelOrderRec (root . right , level + 1 , res); } // Function to perform level order traversal function levelOrder (root) { // Stores the result level by level const res = []; levelOrderRec (root , 0 , res); return res ; } // Driver Code // 5 // / \ // 12 13 // / \ \ // 7 14 2 // / \ / \ / \ //17 23 27 3 8 11 const root = new Node (5); root . left = new Node (12); root . right = new Node (13); root . left . left = new Node (7); root . left . right = new Node (14); root . right . right = new Node (2); root . left . left . left = new Node (17); root . left . left . right = new Node (23); root . left . right . left = new Node (27); root . left . right . right = new Node (3); root . right . right . left = new Node (8); root . right . right . right = new Node (11); const res = levelOrder (root); for (const level of res) { console . log (level . join (' ')); } Output 5 12 13 7 14 2 17 23 27 3 8 11 [Expected Approach] Using Queue (Iterative) - O(n) time and O(n) space The idea is to use a queue to traverse the tree level by level. Start by adding the root to the queue. Then, repeatedly remove a node from the queue, store its value in the result, and add its left and right children to the queue. Continue this process until the queue is empty. C++ #include <iostream> #include <queue> #include <vector> using namespace std ; class Node { public : int data ; Node * left , * right ; // Constructor to initialize a new node Node (int value) { data = value ; left = nullptr ; right = nullptr ; } }; // Iterative method to perform level order traversal vector < vector < int >> levelOrder (Node * root) { if (root == nullptr) return {}; // Create an empty queue for level order traversal queue < Node *> q ; vector < vector < int >> res ; // Enqueue Root q . push (root); int currLevel = 0 ; while (! q . empty ()) { int len = q . size (); res . push_back ({}); for (int i = 0 ; i < len ; i ++) { // Add front of queue and remove it from queue Node * node = q . front (); q . pop (); res [currLevel]. push_back (node -> data); // Enqueue left child if (node -> left != nullptr) q . push (node -> left); // Enqueue right child if (node -> right != nullptr) q . push (node -> right); } currLevel ++ ; } return res ; } int main () { // 5 // / \ // 12 13 // / \ \ // 7 14 2 // / \ / \ / \ //17 23 27 3 8 11 Node * root = new Node (5); root -> left = new Node (12); root -> right = new Node (13); root -> left -> left = new Node (7); root -> left -> right = new Node (14); root -> right -> right = new Node (2); root -> left -> left -> left = new Node (17); root -> left -> left -> right = new Node (23); root -> left -> right -> left = new Node (27); root -> left -> right -> right = new Node (3); root -> right -> right -> left = new Node (8); root -> right -> right -> right = new Node (11); vector < vector <

int >> res = levelOrder (root); for (vector < int > level : res) { for (int val : level) { cout << val << " " ; } cout << endl ; } return 0 ; } Java import java.util.ArrayList ; import java.util.LinkedList ; import java.util.Queue ; class Node { int data ; Node left , right ; Node (int value) { data = value ; left = null ; right = null ; } } // Iterative method to perform level order traversal public class GfG { public static ArrayList < ArrayList < Integer >> levelOrder (Node root) { if (root == null) return new ArrayList <> (); // Create an empty queue for level order traversal Queue < Node > q = new LinkedList <> (); ArrayList < ArrayList < Integer >> res = new ArrayList <> (); // Enqueue Root q . offer (root); int currLevel = 0 ; while (! q . isEmpty ()) { int len = q . size (); res . add (new ArrayList <> ()); for (int i = 0 ; i < len ; i ++) { // Add front of queue and remove it from // queue Node node = q . poll (); res . get (currLevel). add (node . data); // Enqueue left child if (node . left != null) q . offer (node . left); // Enqueue right child if (node . right != null) q . offer (node . right); } currLevel ++ ; } return res ; } public static void main (String [] args) { // 5 // / \ // 12 13 // / \ \ // 7 14 2 // / \ / \ / \ //17 23 27 3 8 11 Node root = new Node (5); root . left = new Node (12); root . right = new Node (13); root . left . left = new Node (7); root . left . right = new Node (14); root . right . right = new Node (2); root . left . left . left = new Node (17); root . left . left . right = new Node (23); root . left . right . left = new Node (27); root . left . right . right = new Node (3); root . right . right . left = new Node (8); root . right . right . right = new Node (11); // Perform level order traversal and get the result ArrayList < ArrayList < Integer >> res = levelOrder (root); for (ArrayList < Integer > level : res) { for (int val : level) { System . out . print (val + " "); } System . out . println (); } } } Python class Node : def __init__ (self , value): self . data = value self . left = None self . right = None # Iterative method to perform level order traversal def levelOrder (root): if root is None : return [] # Create an empty queue for level order traversal q = [] res = [] # Enqueue Root q . append (root) curr_level = 0 while q : len_q = len (q) res . append ([]) for _ in range (len_q): # Add front of queue and remove it from queue node = q . pop (0) res [curr_level] . append (node . data) # Enqueue left child if node . left is not None : q . append (node . left) # Enqueue right child if node . right is not None : q . append (node . right) curr_level += 1 return res if __name__ == '__main__' : # 5 # / \ # 12 13 # / \ \ # 7 14 2 # / \ / \ / \ #17 23 2 3 8 11 root = Node (5) root . left = Node (12) root . right = Node (13) root . left . left = Node (7) root . left . right = Node (14) root . right . right = Node (2) root . left . left . left = Node (17) root . left . left . right = Node (23) root . left . right . left = Node (27) root . left . right . right = Node (3) root . right . right . left = Node (8) root . right . right . right = Node (11) # Perform level order traversal and get the result res = levelOrder (root) for level in res : for val in level : print (val , end = ' ') print () C# using System ; using System.Collections.Generic ; class Node { public int data ; public Node left , right ; // Constructor to initialize a new node public Node (int value) { data = value ; left = null ; right = null ; } } class GfG { // Iterative method to perform level order traversal static List < List < int >> levelOrder (Node root) { if (root == null) return new List < List < int >> (); // Create an empty queue for level order traversal Queue < Node > q = new Queue < Node > (); List < List < int >> res = new List < List < int >> (); // Enqueue Root q . Enqueue (root); int currLevel = 0 ; while (q . Count > 0) { int len = q . Count ; res . Add (new List < int > ()); for (int i = 0 ; i < len ; i ++) { // Add front of queue and remove it from queue Node node = q . Dequeue (); res [currLevel]. Add (node . data); // Enqueue left child if (node . left != null) q . Enqueue (node . left); // Enqueue right child if (node . right != null) q . Enqueue (node . right); } currLevel ++ ; } return res ; } static void Main () { // 5 // / \ // 12 13 // / \ \ // 7 14 2 // / \ / \ / \ //17 23 27 3 8 11 Node root = new Node (5); root . left = new Node (12); root . right = new Node (13); root . left . left = new Node (7); root . left . right = new Node (14); root . right . right = new Node (2); root . left . left . left = new Node (17); root . left . left . right = new Node (23); root . left . right . left = new Node (27); root . left . right . right = new Node (3); root . right . right . left = new Node (8); root . right . right . right = new Node (11); List < List < int >> res = levelOrder (root); foreach (List < int > level in res) { foreach (int val in level) { Console . Write (val + " "); } Console . WriteLine (); } } } JavaScript class Node { constructor (value) { this . data = value ; this . left = null ; this . right = null ; } } // Iterative method to perform level order traversal function levelOrder (root) { if (root === null) return []; // Create an empty queue for level order traversal let q = []; let res = []; // Enqueue Root q . push (root); let currLevel = 0 ; while (q . length > 0) { let len = q . length ; res . push ([]); for (let i = 0 ; i < len ; i ++) { // Add front of queue and remove it from queue let node = q . shift (); res [currLevel]. push (node . data); // Enqueue left child if (node . left !== null) q . push (node . left); // Enqueue right child if (node . right !== null) q . push (node . right); } currLevel ++ ; } return res ; } //Driver Code // 5 // / \ // 12 13 // / \ \ // 7 14 2 // / \ / \ / \ //17 23 27 3 8 11 const root = new Node (5); root . left = new Node (12); root . right = new Node (13); root . left . left = new Node (7); root . left . right = new Node (14); root . right . right = new Node (2); root . left . left . left = new Node (17); root . left . left . right = new Node (23); root . left . right . left = new Node (27); root . left . right . right = new Node

(3); root . right . right . left = new Node (8); root . right . right . right = new Node (11); // Perform level order traversal and get the result const res = levelOrder (root); for (const level of res) { console . log (level . join (' ')); } Output 5 12 13 7 14 2 17 23 27 3 8 11 Comment Article Tags: Article Tags: Tree Queue DSA Microsoft Amazon Morgan Stanley Flipkart Qualcomm Samsung D-E-Shaw Cisco Payu Ola Cabs tree-level-order + 10 More