Binary Tree from Inorder and Preorder traversals - GeeksforGeeks

Source: https://www.geeksforgeeks.org/construct-tree-from-given-inorder-and-preorder-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 Binary Tree from Inorder and Preorder traversals Last Updated : 8 Oct, 2025 Given inorder and preorder traversals of a Binary Tree in array inorder[] and preorder[] respectively, Construct the Binary Tree and return it’s root. Note: All values in inorder[] and preorder[] are distinct. Example: Input: inorder[] = [3, 1, 4, 0, 5, 2], preorder[] = [0, 1, 3, 4, 2, 5] Output: [[0], [1, 2], [3, 4, 5, N]] Explanation: The tree will look like: Try it on GfG Practice Table of Content [Approach 1] Using Pre-order traversal - O(n^2) Time and O(h) Space [Approach 2] Using Pre-order traversal and Hash map - O(n) Time and O(n) Space [Approach 1] Using Pre-order traversal - O(n 2) Time and O(h) Space The idea is to construct the tree using pre-order traversal. Take the first element of the pre-order array and create root node. Find the index of this node in the in-order array. Create the left subtree using the elements present on left side of root node in in-order array. Similarly create the right subtree using the elements present on the right side of the root node in in-order array. C++ #include <iostream> #include <queue> #include <vector> using namespace std ; // Node Structure class Node { public : int data ; Node * left , * right ; Node (int x) { data = x ; left = nullptr ; right = nullptr ; } }; // Function to find the index of an element in the array. int search (vector < int > & inorder , int value , int left , int right) { for (int i = left ; i <= right ; i ++) { if (inorder [i] == value) return i ; } return -1 ; } // Recursive function to build the binary tree. Node * buildTreeRecur (vector < int > & inorder , vector < int > & preorder , int & preIndex , int left , int right) { // For empty inorder array, return null if (left > right) return nullptr ; int rootVal = preorder [preIndex]; preIndex ++ ; Node * root = new Node (rootVal); int index = search (inorder , rootVal , left , right); // Recursively create the left and right subtree. root -> left = buildTreeRecur (inorder , preorder , preIndex , left , index - 1); root -> right = buildTreeRecur (inorder , preorder , preIndex , index + 1 , right); return root ; } // Function to construct tree from its inorder and preorder traversals Node * buildTree (vector < int > & inorder , vector < int > & preorder) { int preIndex = 0 ; Node * root = buildTreeRecur (inorder , preorder , preIndex , 0 , preorder . size () - 1); return root ; } int getHeight (Node * root , int h) { if (root == nullptr) return h - 1 ; return max (getHeight (root -> left , h + 1), getHeight (root -> right , h + 1)); } void levelOrder (Node * root) { queue < pair < Node * , int >> q ; q . push ({ root , 0 }); int lastLevel = 0 ; // function to get the height of tree int height = getHeight (root , 0); // printing the level order of tree while (! q . empty ()) { auto top = q . front (); q . pop (); Node * node = top . first ; int lvl = top . second ; if (lvl > lastLevel) { cout << " \n " ; lastLevel = lvl ; } // all levels are printed if (lvl > height) break ; if (node -> data != -1) cout << node -> data << " " ; // printing null node else cout << "N " ; // null node has no children if (node -> data == -1) continue ; if (node -> left == nullptr) q . push ({ new Node (-1), lvl + 1 }); else q . push ({ node -> left , lvl + 1 }); if (node -> right == nullptr) q . push ({ new Node (-1), lvl + 1 }); else q . push ({ node -> right , lvl + 1 }); } } int main () { vector < int > inorder = { 3 , 1 , 4 , 0 , 5 , 2 }; vector < int > preorder = { 0 , 1 , 3 , 4 , 2 , 5 }; Node * root = buildTree (inorder , preorder); levelOrder (root); return 0 ; } Java import java.util.LinkedList ; import java.util.Queue ; import java.util.List ; import java.util.ArrayList ; // Node Structure class Node { int data ; Node left , right ; Node (int x) { data = x ; left = null ; right = null ; } } class GFG { // Function to find the index of an element in the array. static int search (int [] inorder , int value , int left , int right) { for (int i = left ; i <= right ; i ++) { if (inorder [i] == value) return i ; } return - 1 ; } // Recursive function to build the binary tree. static Node buildTreeRecur (int [] inorder , int [] preorder , int [] preIndex , int left , int right) { // For empty inorder array, return null if (

left > right) return null ; int rootVal = preorder [preIndex [0]] ; preIndex [0]++ ; Node root = new Node (rootVal); int index = search (inorder , rootVal , left , right); // Recursively create the left and right subtree. root . left = buildTreeRecur (inorder , preorder , preIndex , left , index - 1); root . right = buildTreeRecur (inorder , preorder , preIndex , index + 1 , right); return root ; } // Function to construct tree from its inorder and preorder traversals static Node buildTree (int [] inorder , int [] preorder) { int [] preIndex = { 0 }; return buildTreeRecur (inorder , preorder , preIndex , 0 , preorder . length - 1); } static int getHeight (Node root , int h) { if (root == null) return h - 1 ; return Math . max (getHeight (root . left , h + 1), getHeight (root . right , h + 1)); } static void levelOrder (Node root) { Queue < List < Object >> queue = new LinkedList <> (); queue . offer (List . of (root , 0)); int lastLevel = 0 ; // function to get the height of tree int height = getHeight (root , 0); // printing the level order of tree while (! queue . isEmpty ()) { List < Object > top = queue . poll (); Node node = (Node) top . get (0); int lvl = (int) top . get (1); if (lvl > lastLevel) { System . out . println (); lastLevel = lvl ; } // all levels are printed if (lvl > height) break ; // printing null node System . out . print ((node . data == - 1 ? "N" : node . data) + " "); // null node has no children if (node . data == - 1) continue ; if (node . left == null) queue . offer (List . of (new Node (- 1), lvl + 1)); else queue . offer (List . of (node . left , lvl + 1)); if (node . right == null) queue . offer (List . of (new Node (- 1), lvl + 1)); else queue . offer (List . of (node . right , lvl + 1)); } } public static void main (String [] args) { int [] inorder = { 3 , 1 , 4 , 0 , 5 , 2 }; int [] preorder = { 0 , 1 , 3 , 4 , 2 , 5 }; Node root = buildTree (inorder , preorder); levelOrder (root); } } Python from collections import deque # Node Structure class Node : def __init__ (self , x): self . data = x self . left = None self . right = None # Function to find the index of an element in the array def search (inorder , value , left , right): for i in range (left , right + 1): if inorder [i] == value : return i return - 1 # Recursive function to build the binary tree. def buildTreeRecur (inorder , preorder , preIndex , left , right): # For empty inorder array, return null if left > right : return None rootVal = preorder [preIndex [0]] preIndex [0] += 1 root = Node (rootVal) index = search (inorder , rootVal , left , right) # Recursively create the left and right subtree. root . left = buildTreeRecur (inorder , preorder , preIndex , left , index - 1) root . right = buildTreeRecur (inorder , preorder , preIndex , index + 1 , right) return root # Function to construct tree from its inorder and preorder traversals def buildTree (inorder , preorder): preIndex = [0] return buildTreeRecur (inorder , preorder , preIndex , 0 , len (preorder) - 1) def getHeight (root , h): if root is None : return h - 1 return max (getHeight (root . left , h + 1), getHeight (root . right , h + 1)) def levelOrder (root): queue = deque ([[root , 0]]) lastLevel = 0 # function to get the height of tree height = getHeight (root , 0) # printing the level order of tree while queue : node , lvl = queue . popleft () if lvl > lastLevel : print () lastLevel = lvl # all levels are printed if lvl > height : break # printing null node print ("N" if node . data == - 1 else node . data , end = " ") # null node has no children if node . data == - 1 : continue if node . left is None : queue . append ([Node (- 1), lvl + 1]) else : queue . append ([node . left , lvl + 1]) if node . right is None : queue . append ([Node (- 1), lvl + 1]) else : queue . append ([node . right , lvl + 1]) if __name__ == "__main__" : inorder = [3 , 1 , 4 , 0 , 5 , 2] preorder = [0 , 1 , 3 , 4 , 2 , 5] root = buildTree (inorder , preorder) levelOrder (root) C# using System ; using System.Collections.Generic ; // Node Structure class Node { public int data ; public Node left , right ; public Node (int x) { data = x ; left = null ; right = null ; } } class GFG { // Print tree as level order static void printLevelOrder (Node root) { if (root == null) { Console . Write ("N "); return ; } Queue < Node > q = new Queue < Node > (); q . Enqueue (root); int nonNull = 1 ; while (q . Count > 0 && nonNull > 0) { Node curr = q . Dequeue (); if (curr == null) { Console . Write ("N "); continue ; } nonNull -- ; Console . Write (curr . data + " "); q . Enqueue (curr . left); q . Enqueue (curr . right); if (curr . left != null) nonNull ++ ; if (curr . right != null) nonNull ++ ; } } // Function to find the index of an element in the array static int search (int [] inorder , int value , int left , int right) { for (int i = left ; i <= right ; i ++) { if (inorder [i] == value) return i ; } return - 1 ; } // Recursive function to build the binary tree. static Node buildTreeRecur (int [] inorder , int [] preorder , ref int preIndex , int left , int right) { // For empty inorder array, return null if (left > right) return null ; int rootVal = preorder [preIndex]; preIndex ++ ; Node root = new Node (rootVal); int index = search (inorder , rootVal , left , right); // Recursively create the left and right subtree. root . left = buildTreeRecur (inorder , preorder , ref preIndex , left , index - 1); root . right = buildTreeRecur (inorder , preorder , ref preIndex , index + 1 , right); return root ; } // Function to construct tree from its inorder and preorder traversals static Node buildTree (int [] inorder , int [] preorder) { int preIndex = 0 ; return buildTreeRecur (inorder , preorder , ref preIndex , 0 , preorder . Length - 1); } static int getHeight (Node root , int h) { if (root == null) return h - 1 ; return Math . Max (getHeight (root . left , h + 1), getHeight (root . right , h + 1)); } static void levelOrder (Node root) { Queue < (Node , int) > queue = new Queue < (Node , int) > (); queue . Enqueue ((root , 0)); int lastLevel = 0 ; // function to get the height of tree int height = getHeight (root , 0); // printing the level

order of tree while (queue . Count > 0) { var (node , lvl) = queue . Dequeue (); if (lvl > lastLevel) { Console . WriteLine (); lastLevel = lvl ; } // all levels are printed if (lvl > height) break ; // printing null node Console . Write ((node . data == - 1 ? "N" : node . data . ToString ()) + " "); // null node has no children if (node . data == - 1) continue ; if (node . left == null) queue . Enqueue ((new Node (- 1), lvl + 1)); else queue . Enqueue ((node . left , lvl + 1)); if (node . right == null) queue . Enqueue ((new Node (- 1), lvl + 1)); else queue . Enqueue ((node . right , lvl + 1)); } } static void Main (string [] args) { int [] inorder = { 3 , 1 , 4 , 0 , 5 , 2 }; int [] preorder = { 0 , 1 , 3 , 4 , 2 , 5 }; Node root = buildTree (inorder , preorder); levelOrder (root); } } JavaScript // Node Structure class Node { constructor (x) { this . data = x ; this . left = null ; this . right = null ; } } // Queue node class QNode { constructor (data) { this . data = data ; this . next = null ; } } class Queue { constructor () { this . front = null ; this . rear = null ; } isEmpty () { return this . front === null ; } enqueue (x) { let newNode = new QNode (x); if (this . rear === null) { this . front = this . rear = newNode ; return ; } this . rear . next = newNode ; this . rear = newNode ; } dequeue () { if (this . front === null) return null ; let temp = this . front ; this . front = this . front . next ; if (this . front === null) this . rear = null ; return temp . data ; } } // Function to find the index of an element in the array function search (inorder , value , left , right) { for (let i = left ; i <= right ; i ++) { if (inorder [i] === value) return i ; } return - 1 ; } // Recursive function to build the binary tree. function buildTreeRecur (inorder , preorder , preIndex , left , right) { // For empty inorder array, return null if (left > right) return null ; const rootVal = preorder [preIndex [0]]; preIndex [0] ++ ; const root = new Node (rootVal); const index = search (inorder , rootVal , left , right); // Recursively create the left and right subtree. root . left = buildTreeRecur (inorder , preorder , preIndex , left , index - 1); root . right = buildTreeRecur (inorder , preorder , preIndex , index + 1 , right); return root ; } // Function to construct tree from its inorder and preorder traversals function buildTree (inorder , preorder) { const preIndex = [0]; return buildTreeRecur (inorder , preorder , preIndex , 0 , preorder . length - 1); } function getHeight (root , h) { if (root === null) return h - 1 ; return Math . max (getHeight (root . left , h + 1), getHeight (root . right , h + 1)); } function levelOrder (root) { let queue = []; queue . push ([root , 0]); let lastLevel = 0 ; // get the height of tree let height = getHeight (root , 0); // printing the level order of tree while (queue . length > 0) { let [node , lvl] = queue . shift (); if (lvl > lastLevel) { console . log (""); lastLevel = lvl ; } // all levels are printed if (lvl > height) break ; // printing null node process . stdout . write ((node . data === - 1 ? "N" : node . data) + " "); // null node has no children if (node . data === - 1) continue ; if (node . left === null) queue . push ([new Node (- 1), lvl + 1]); else queue . push ([node . left , lvl + 1]); if (node . right === null) queue . push ([new Node (- 1), lvl + 1]); else queue . push ([node . right , lvl + 1]); } } // Driver Code const inorder = [3 , 1 , 4 , 0 , 5 , 2]; const preorder = [0 , 1 , 3 , 4 , 2 , 5]; const root = buildTree (inorder , preorder); levelOrder (root); Output 0 1 2 3 4 5 N [Approach 2] Using Pre-order traversal and Hash map - O(n) Time and O(n) Space The idea is similar to first approach, but instead of linearly searching the in-order array for each node we can use hashing . Map the values of in-order array to its indices. This will reduce the searching complexity from O(n) to O(1). C++ #include <iostream> #include <queue> #include <vector> #include <unordered_map> using namespace std ; // Node Structure class Node { public : int data ; Node * left , * right ; Node (int x) { data = x ; left = nullptr ; right = nullptr ; } }; // Recursive function to build the binary tree. Node * buildTreeRecur (unordered_map < int , int > & mp , vector < int > & preorder , int & preIndex , int left , int right) { // For empty inorder array, return null if (left > right) return nullptr ; int rootVal = preorder [preIndex]; preIndex ++ ; Node * root = new Node (rootVal); int index = mp [rootVal]; // Recursively create the left and right subtree. root -> left = buildTreeRecur (mp , preorder , preIndex , left , index - 1); root -> right = buildTreeRecur (mp , preorder , preIndex , index + 1 , right); return root ; } // Function to construct tree from its inorder and preorder traversals Node * buildTree (vector < int > & inorder , vector < int > & preorder) { // Hash map that stores index of a root element in inorder array unordered_map < int , int > mp ; for (int i = 0 ; i < inorder . size (); i ++) mp [inorder [i]] = i ; int preIndex = 0 ; Node * root = buildTreeRecur (mp , preorder , preIndex , 0 , inorder . size () - 1); return root ; } int getHeight (Node * root , int h) { if (root == nullptr) return h - 1 ; return max (getHeight (root -> left , h + 1), getHeight (root -> right , h + 1)); } void levelOrder (Node * root) { queue < pair < Node * , int >> q ; q . push ({ root , 0 }); int lastLevel = 0 ; // function to get the height of tree int height = getHeight (root , 0); // printing the level order of tree while (! q . empty ()) { auto top = q . front (); q . pop (); Node * node = top . first ; int lvl = top . second ; if (lvl > lastLevel) { cout << " \n " ; lastLevel = lvl ; } // all levels are printed if (lvl > height) break ; if (node -> data != -1) cout << node -> data << " " ; // printing null node else cout << "N " ; // null node has no children if (node -> data == -1) continue ; if (node -> left == nullptr) q . push ({ new Node (-1), lvl + 1 }); else q . push ({ node -> left , lvl + 1 }); if (node -> right == nullptr) q . push ({ new Node (-1), lvl + 1 }); else q . push ({ node -> right , lvl + 1 }); } } int main () { vector < int > inorder =

{ 3 , 1 , 4 , 0 , 5 , 2 }; vector < int > preorder = { 0 , 1 , 3 , 4 , 2 , 5 }; Node * root = buildTree (inorder , preorder); levelOrder (root); return 0 ; } Java import java.util.LinkedList ; import java.util.Queue ; import java.util.List ; import java.util.HashMap ; import java.util.Map ; import java.util.ArrayList ; // Node Structure class Node { int data ; Node left , right ; Node (int x) { data = x ; left = null ; right = null ; } } class GFG { // Recursive function to build the binary tree. static Node buildTreeRecur (Map < Integer , Integer > mp , int [] preorder , int [] preIndex , int left , int right) { // For empty inorder array, return null if (left > right) return null ; int rootVal = preorder [preIndex [0]] ; preIndex [0]++ ; Node root = new Node (rootVal); int index = mp . get (rootVal); // Recursively create the left and right subtree. root . left = buildTreeRecur (mp , preorder , preIndex , left , index - 1); root . right = buildTreeRecur (mp , preorder , preIndex , index + 1 , right); return root ; } // Function to construct tree from its inorder and preorder traversals static Node buildTree (int [] inorder , int [] preorder) { // Hash map that stores index of a root element in inorder array Map < Integer , Integer > mp = new HashMap <> (); for (int i = 0 ; i < inorder . length ; i ++) mp . put (inorder [i] , i); int [] preIndex = { 0 }; return buildTreeRecur (mp , preorder , preIndex , 0 , inorder . length - 1); } static int getHeight (Node root , int h) { if (root == null) return h - 1 ; return Math . max (getHeight (root . left , h + 1), getHeight (root . right , h + 1)); } static void levelOrder (Node root) { Queue < List < Object >> queue = new LinkedList <> (); queue . offer (List . of (root , 0)); int lastLevel = 0 ; // function to get the height of tree int height = getHeight (root , 0); // printing the level order of tree while (! queue . isEmpty ()) { List < Object > top = queue . poll (); Node node = (Node) top . get (0); int lvl = (int) top . get (1); if (lvl > lastLevel) { System . out . println (); lastLevel = lvl ; } // all levels are printed if (lvl > height) break ; // printing null node System . out . print ((node . data == - 1 ? "N" : node . data) + " "); // null node has no children if (node . data == - 1) continue ; if (node . left == null) queue . offer (List . of (new Node (- 1), lvl + 1)); else queue . offer (List . of (node . left , lvl + 1)); if (node . right == null) queue . offer (List . of (new Node (- 1), lvl + 1)); else queue . offer (List . of (node . right , lvl + 1)); } } public static void main (String [] args) { int [] inorder = { 3 , 1 , 4 , 0 , 5 , 2 }; int [] preorder = { 0 , 1 , 3 , 4 , 2 , 5 }; Node root = buildTree (inorder , preorder); levelOrder (root); } } Python from collections import deque # Node Structure class Node : def __init__ (self , x): self . data = x self . left = None self . right = None # Recursive function to build the binary tree. def buildTreeRecur (mp , preorder , preIndex , left , right): # For empty inorder array, return None if left > right : return None rootVal = preorder [preIndex [0]] preIndex [0] += 1 root = Node (rootVal) index = mp [rootVal] # Recursively create the left and right subtree. root . left = buildTreeRecur (mp , preorder , preIndex , left , index - 1) root . right = buildTreeRecur (mp , preorder , preIndex , index + 1 , right) return root # Function to construct tree from its inorder and preorder traversals def buildTree (inorder , preorder): # Hash map that stores index of a root element in inorder array mp = { value : idx for idx , value in enumerate (inorder)} preIndex = [0] return buildTreeRecur (mp , preorder , preIndex , 0 , len (inorder) - 1) def getHeight (root , h): if root is None : return h - 1 return max (getHeight (root . left , h + 1), getHeight (root . right , h + 1)) def levelOrder (root): queue = deque ([[root , 0]]) lastLevel = 0 # function to get the height of tree height = getHeight (root , 0) # printing the level order of tree while queue : node , lvl = queue . popleft () if lvl > lastLevel : print () lastLevel = lvl # all levels are printed if lvl > height : break # printing null node print ("N" if node . data == - 1 else node . data , end = " ") # null node has no children if node . data == - 1 : continue if node . left is None : queue . append ([Node (- 1), lvl + 1]) else : queue . append ([node . left , lvl + 1]) if node . right is None : queue . append ([Node (- 1), lvl + 1]) else : queue . append ([node . right , lvl + 1]) if __name__ == "__main__" : inorder = [3 , 1 , 4 , 0 , 5 , 2] preorder = [0 , 1 , 3 , 4 , 2 , 5] root = buildTree (inorder , preorder) levelOrder (root) C# using System ; using System.Collections.Generic ; // Node Structure class Node { public int data ; public Node left , right ; public Node (int x) { data = x ; left = null ; right = null ; } } class GFG { // Recursive function to build the binary tree. static Node buildTreeRecur (Dictionary < int , int > mp , int [] preorder , ref int preIndex , int left , int right) { // For empty inorder array, return null if (left > right) return null ; int rootVal = preorder [preIndex]; preIndex ++ ; Node root = new Node (rootVal); int index = mp [rootVal]; // Recursively create the left and right subtree. root . left = buildTreeRecur (mp , preorder , ref preIndex , left , index - 1); root . right = buildTreeRecur (mp , preorder , ref preIndex , index + 1 , right); return root ; } // Function to construct tree from its inorder and preorder traversals static Node buildTree (int [] inorder , int [] preorder) { // Hash map that stores index of a root element in inorder array Dictionary < int , int > mp = new Dictionary < int , int > (); for (int i = 0 ; i < inorder . Length ; i ++) mp [inorder [i]] = i ; int preIndex = 0 ; return buildTreeRecur (mp , preorder , ref preIndex , 0 , inorder . Length - 1); } static int getHeight (Node root , int h) { if (root == null) return h - 1 ; return Math . Max (getHeight (root . left , h + 1), getHeight (root . right , h + 1)); } static void levelOrder (Node root) { Queue < (Node , int) > queue =

new Queue < (Node , int) > (); queue . Enqueue ((root , 0)); int lastLevel = 0 ; // function to get the height of tree int height = getHeight (root , 0); // printing the level order of tree while (queue . Count > 0) { var (node , lvl) = queue . Dequeue (); if (lvl > lastLevel) { Console . WriteLine (); lastLevel = lvl ; } // all levels are printed if (lvl > height) break ; // printing null node Console . Write ((node . data == - 1 ? "N" : node . data . ToString ()) + " "); // null node has no children if (node . data == - 1) continue ; if (node . left == null) queue . Enqueue ((new Node (- 1), lvl + 1)); else queue . Enqueue ((node . left , lvl + 1)); if (node . right == null) queue . Enqueue ((new Node (- 1), lvl + 1)); else queue . Enqueue ((node . right , lvl + 1)); } } public static void Main (string [] args) { int [] inorder = { 3 , 1 , 4 , 0 , 5 , 2 }; int [] preorder = { 0 , 1 , 3 , 4 , 2 , 5 }; Node root = buildTree (inorder , preorder); levelOrder (root); } } JavaScript // Node Structure class Node { constructor (x) { this . data = x ; this . left = null ; this . right = null ; } } // Queue node class QNode { constructor (data) { this . data = data ; this . next = null ; } } class Queue { constructor () { this . front = null ; this . rear = null ; } isEmpty () { return this . front === null ; } enqueue (x) { let newNode = new QNode (x); if (this . rear === null) { this . front = this . rear = newNode ; return ; } this . rear . next = newNode ; this . rear = newNode ; } dequeue () { if (this . front === null) return null ; let temp = this . front ; this . front = this . front . next ; if (this . front === null) this . rear = null ; return temp . data ; } } // Recursive function to build the binary tree. function buildTreeRecur (mp , preorder , preIndex , left , right) { // For empty inorder array, return null if (left > right) return null ; const rootVal = preorder [preIndex [0]]; preIndex [0] ++ ; const root = new Node (rootVal); const index = mp [rootVal]; // Recursively create the left and right subtree. root . left = buildTreeRecur (mp , preorder , preIndex , left , index - 1); root . right = buildTreeRecur (mp , preorder , preIndex , index + 1 , right); return root ; } // Function to construct tree from its inorder and preorder traversals function buildTree (inorder , preorder) { // Hash map that stores index of a root element in inorder array const mp = {}; inorder . forEach ((val , idx) => { mp [val] = idx ; }); const preIndex = [0]; return buildTreeRecur (mp , preorder , preIndex , 0 , inorder . length - 1); } function getHeight (root , h) { if (root === null) return h - 1 ; return Math . max (getHeight (root . left , h + 1), getHeight (root . right , h + 1)); } function levelOrder (root) { let queue = []; queue . push ([root , 0]); let lastLevel = 0 ; // get the height of tree let height = getHeight (root , 0); // printing the level order of tree while (queue . length > 0) { let [node , lvl] = queue . shift (); if (lvl > lastLevel) { console . log (""); lastLevel = lvl ; } // all levels are printed if (lvl > height) break ; // printing null node process . stdout . write ((node . data === - 1 ? "N" : node . data) + " "); // null node has no children if (node . data === - 1) continue ; if (node . left === null) queue . push ([new Node (- 1), lvl + 1]); else queue . push ([node . left , lvl + 1]); if (node . right === null) queue . push ([new Node (- 1), lvl + 1]); else queue . push ([node . right , lvl + 1]); } } // Driver Code const inorder = [3 , 1 , 4 , 0 , 5 , 2]; const preorder = [0 , 1 , 3 , 4 , 2 , 5]; const root = buildTree (inorder , preorder); levelOrder (root); Output 0 1 2 3 4 5 N Related articles: Construct a Binary Tree from Postorder and Inorder Comment Article Tags: Article Tags: Tree DSA Inorder Traversal Preorder Traversal Microsoft Amazon Accolite tree-traversal cpp-unordered_map + 5 More