Serialize and Deserialize a Binary Tree - GeeksforGeeks

Source: https://www.geeksforgeeks.org/serialize-deserialize-binary-tree/

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 Serialize and Deserialize a Binary Tree Last Updated : 4 Oct, 2025 Given the root of a binary tree, Complete the functions: serialize() : Traverse the tree, store node values in an array, and insert -1 wherever a node is null. Returns the array representing the tree. deSerialize() : Read the values of the array arr[] to reconstruct the tree. An element arr[i] == -1 represents a missing (null) child. Note: Multiple nodes can have the same data and the node values are always positive. Table of Content [Approach 1] Using Preorder Traversal (Recursive) - O(n) Time and O(n) Space [Approach 2] Using Level Order Traversal - O(n) Time and O(n) Space [Approach 1] Using Preorder Traversal (Recursive) - O(n) Time and O(n) Space The idea is to use preorder traversal for serialization. While traversing, if a node is not null, store its value else store -1 as a marker. Try it on GfG Practice Serialization : Perform preorder traversal, store node values and insert -1 wherever a node is null. Deserialization : Rebuild the tree by reading values in preorder order. If the value is -1, return null. Otherwise, create a node and recursively build its left and right children. Pseudo-code Idea (Deserialization) If current value = -1 $\rightarrow$ return NULL Else Create a new node with the current value Recursively call to build node->left and node->right Return the node C++ #include <iostream> #include <vector> #include <queue> using namespace std ; // Node Structure class Node { public : int data ; Node * left , * right ; Node (int x) { data = x ; left = nullptr ; right = nullptr ; } }; // Function to store the preorder void serializePreOrder (Node * root , vector < int > & arr) { // Push -1 if root is null. if (root == nullptr) { arr . push_back (-1); return ; } // Push the root into result. arr . push_back (root -> data); serializePreOrder (root -> left , arr); serializePreOrder (root -> right , arr); } // Main function to serialize a tree. vector < int > serialize (Node * root) { vector < int > arr ; serializePreOrder (root , arr); return arr ; } // Function to restore the array from preorder Node * deserializePreOrder (int & i , vector < int > & arr) { // if element is -1 return null if (arr [i] == -1){ i ++ ; return nullptr ; } // Create the root node. Node * root = new Node (arr [i]); i ++ ; // Create the left and right subtree. root -> left = deserializePreOrder (i , arr); root -> right = deserializePreOrder (i , arr); return root ; } Node * deSerialize (vector < int > & arr) { int i = 0 ; return deserializePreOrder (i , arr); } int main () { // 10 // / \ // 20 30 // / \ // 40 60 Node * root = new Node (10); root -> left = new Node (20); root -> right = new Node (30); root -> left -> left = new Node (40); root -> left -> right = new Node (60); vector < int > arr = serialize (root); Node * res = deSerialize (arr); } Java import java.util.ArrayList ; import java.util.Queue ; import java.util.LinkedList ; // Node Structure class Node { int data ; Node left , right ; Node (int x) { data = x ; left = null ; right = null ; } } class GFG { // Function to store the preorder static void serializePreOrder (Node root , ArrayList < Integer > arr) { // Push -1 if root is null. if (root == null) { arr . add (- 1); return ; } // Push the root into result. arr . add (root . data); serializePreOrder (root . left , arr); serializePreOrder (root . right , arr); } // function to serialize a tree. static ArrayList < Integer > serialize (Node root) { ArrayList < Integer > arr = new ArrayList <> (); serializePreOrder (root , arr); return arr ; } // Function to restore the tree from preorder static Node deserializePreOrder (int [] i , ArrayList < Integer > arr) { // if elemnt is -1 return null if (arr . get (i [0]) == - 1) { i [0]++ ; return null ; } // Create the root node. Node root = new Node (arr . get (i [0])); i [0]++ ; // Create the left and right subtree. root . left = deserializePreOrder (i , arr); root . right = deserializePreOrder (i , arr); return root ; } // function to deserialize a tree. static Node deSerialize (ArrayList < Integer > arr) { int [] i = { 0 }; return deserializePreOrder (i , arr); } public static void main (String [] args) { // 10 // / \ // 20 30 // / \ //

40 60 Node root = new Node (10); root . left = new Node (20); root . right = new Node (30); root . left . left = new Node (40); root . left . right = new Node (60); ArrayList < Integer > arr = serialize (root); Node res = deSerialize (arr); } } Python from collections import deque # Node Structure class Node : def __init__ (self , x): self . data = x self . left = None self . right = None # Function to store preorder def serializePreOrder (root , arr): # Push -1 if root is null. if root is None : arr . append (- 1) return # Push the root into result. arr . append (root . data) serializePreOrder (root . left , arr) serializePreOrder (root . right , arr) # function to serialize a tree. def serialize (root): arr = [] serializePreOrder (root , arr) return arr # Function to restore the tree from preorder def deserializePreOrder (i , arr): # if element is -1 return null if arr [i [0]] == - 1 : i [0] += 1 return None # Create the root node. root = Node (arr [i [0]]) i [0] += 1 # Create the left and right subtree. root . left = deserializePreOrder (i , arr) root . right = deserializePreOrder (i , arr) return root # function to deserialize a tree. def deSerialize (arr): i = [0] return deserializePreOrder (i , arr) if __name__ == "__main__" : # 10 # / \ # 20 30 # / \ # 40 60 root = Node (10) root . left = Node (20) root . right = Node (30) root . left . left = Node (40) root . left . right = Node (60) arr = serialize (root) res = deSerialize (arr) 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 { // Function to store the preorder static void serializePreOrder (Node root , List < int > arr) { // Push -1 if root is null. if (root == null) { arr . Add (- 1); return ; } // Push the root into result. arr . Add (root . data); serializePreOrder (root . left , arr); serializePreOrder (root . right , arr); } // function to serialize a tree. static List < int > serialize (Node root) { List < int > arr = new List < int > (); serializePreOrder (root , arr); return arr ; } // Function to restore the tree from preorder static Node deserializePreOrder (ref int i , List < int > arr) { // if element is -1 return null if (arr [i] == - 1) { i ++ ; return null ; } // Create the root node. Node root = new Node (arr [i]); i ++ ; // Create the left and right subtree. root . left = deserializePreOrder (ref i , arr); root . right = deserializePreOrder (ref i , arr); return root ; } // function to deserialize a tree. static Node deSerialize (List < int > arr) { int i = 0 ; return deserializePreOrder (ref i , arr); } static void Main () { // 10 // / \ // 20 30 // / \ // 40 60 Node root = new Node (10); root . left = new Node (20); root . right = new Node (30); root . left . left = new Node (40); root . left . right = new Node (60); List < int > arr = serialize (root); // Variable assigned but never used #pragma warning disable CS0219 Node res = deSerialize (arr); #pragma warning restore CS0219 } } JavaScript // Node Structure class Node { constructor (x) { this . data = x ; this . left = null ; this . right = null ; } } // Function to store the preorder function serializePreOrder (root , arr) { // Push -1 if root is null. if (root === null) { arr . push (- 1); return ; } // Push the root into result. arr . push (root . data); serializePreOrder (root . left , arr); serializePreOrder (root . right , arr); } // function to serialize a tree. function serialize (root) { const arr = []; serializePreOrder (root , arr); return arr ; } // Function to restore the tree from preorder function deserializePreOrder (i , arr) { // if element is -1 return null if (arr [i [0]] === - 1) { i [0] ++ ; return null ; } // Create the root node. const root = new Node (arr [i [0]]); i [0] ++ ; // Create the left and right subtree. root . left = deserializePreOrder (i , arr); root . right = deserializePreOrder (i , arr); return root ; } // function to deserialize a tree. function deserialize (arr) { const i = [0]; return deserializePreOrder (i , arr); } // Driver Code // 10 // / \ // 20 30 // / \ // 40 60 const root = new Node (10); root . left = new Node (20); root . right = new Node (30); root . left . left = new Node (40); root . left . right = new Node (60); const arr = serialize (root); const res = deserialize (arr); [Approach 2] Using Level Order Traversal - O(n) Time and O(n) Space The idea is to use level-order traversal for serialization. Push the root into a queue and traverse the tree level by level. For each node, if it exists, store its value and push its left and right children into the queue; if it’s null, store -1 to mark null. For deserialization, create the root from the first element and push it into a queue. Then, for each element in the array, pop a node from the queue and link its left and right children if the element is -1, the child is null; otherwise, create the node and push it into the queue. This ensures the tree structure is fully preserved. C++ #include <iostream> #include <vector> #include <queue> using namespace std ; // Node Structure class Node { public : int data ; Node * left , * right ; Node (int x) { data = x ; left = nullptr ; right = nullptr ; } }; // function to serialize a tree. vector < int > serialize (Node * root) { vector < int > arr ; queue < Node *> q ; q . push (root); while (! q . empty ()) { Node * curr = q . front (); q . pop (); // If curr node is null, // append -1 to result. if (curr == nullptr) { arr . push_back (-1); continue ; } // else push its value into // result and push left and right // child nodes into queue. arr . push_back (curr -> data); q . push (curr -> left); q . push (curr -> right); } return arr ; } // function to deserialize a tree. Node * deSerialize (vector < int > & arr) { if (arr [0] == -1) return nullptr ; // create root node and push // it into queue Node * root = new Node (arr [0]); queue < Node *> q ; q . push (root); int i = 1 ; while (! q . empty ()) { Node * curr = q . front (); q . pop (); // If left node is not null if (arr [i] != -1) { Node * left

= new Node (arr [i]); curr -> left = left ; q . push (left); } i ++ ; // If right node is not null if (arr [i] != -1) { Node * right = new Node (arr [i]); curr -> right = right ; q . push (right); } i ++ ; } return root ; } int main () { // Create a hard coded tree // 10 // / \ // 20 30 // / \ // 40 60 Node * root = new Node (10); root -> left = new Node (20); root -> right = new Node (30); root -> left -> left = new Node (40); root -> left -> right = new Node (60); vector < int > arr = serialize (root); Node * res = deSerialize (arr); } Java import java.util.ArrayList ; import java.util.LinkedList ; import java.util.Queue ; // Node Structure class Node { int data ; Node left , right ; Node (int x) { data = x ; left = null ; right = null ; } } class GFG { // Main function to serialize a tree. static ArrayList < Integer > serialize (Node root) { ArrayList < Integer > arr = new ArrayList <> (); Queue < Node > q = new LinkedList <> (); q . add (root); while (! q . isEmpty ()) { Node curr = q . poll (); // If curr node is null, // append -1 to result. if (curr == null) { arr . add (- 1); continue ; } // else push its value into // result and push left and right // child nodes into queue. arr . add (curr . data); q . add (curr . left); q . add (curr . right); } return arr ; } // Main function to deserialize a tree. static Node deSerialize (ArrayList < Integer > arr) { // base case if (arr . get (0) == - 1) return null ; // create root node and push // it into queue Node root = new Node (arr . get (0)); Queue < Node > q = new LinkedList <> (); q . add (root); int i = 1 ; while (! q . isEmpty ()) { Node curr = q . poll (); // If left node is not null if (arr . get (i) != - 1) { Node left = new Node (arr . get (i)); curr . left = left ; q . add (left); } i ++ ; // If right node is not null if (arr . get (i) != - 1) { Node right = new Node (arr . get (i)); curr . right = right ; q . add (right); } i ++ ; } return root ; } public static void main (String [] args) { // 10 // / \ // 20 30 // / \ // 40 60 Node root = new Node (10); root . left = new Node (20); root . right = new Node (30); root . left . left = new Node (40); root . left . right = new Node (60); ArrayList < Integer > arr = serialize (root); Node res = deSerialize (arr); } } Python from collections import deque # Node Structure class Node : def __init__ (self , x): self . data = x self . left = None self . right = None # function to serialize a tree. def serialize (root): arr = [] q = deque ([root]) while q : curr = q . popleft () # If curr node is null, # append -1 to result. if not curr : arr . append (- 1) continue # else push its value into # result and push left and right # child nodes into queue. arr . append (curr . data) q . append (curr . left) q . append (curr . right) return arr # function to deserialize a tree. def deSerialize (arr): # base case if arr [0] == - 1 : return None # create root node and push # it into queue root = Node (arr [0]) q = deque ([root]) i = 1 while q : curr = q . popleft () # If left node is not null if arr [i] != - 1 : left = Node (arr [i]) curr . left = left q . append (left) i += 1 # If right node is not null if arr [i] != - 1 : right = Node (arr [i]) curr . right = right q . append (right) i += 1 return root if __name__ == "__main__" : # 10 # / \ # 20 30 # / \ # 40 60 root = Node (10) root . left = Node (20) root . right = Node (30) root . left . left = Node (40) root . left . right = Node (60) arr = serialize (root) res = deSerialize (arr) 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 { // function to serialize a tree. static List < int > serialize (Node root) { List < int > arr = new List < int > (); Queue < Node > q = new Queue < Node > (); q . Enqueue (root); while (q . Count > 0) { Node curr = q . Dequeue (); // If curr node is null, // append -1 to result. if (curr == null) { arr . Add (- 1); continue ; } // else push its value into // result and push left and right // child nodes into queue. arr . Add (curr . data); q . Enqueue (curr . left); q . Enqueue (curr . right); } return arr ; } // function to deserialize a tree. static Node deSerialize (List < int > arr) { // base case if (arr [0] == - 1) return null ; // create root node and push // it into queue Node root = new Node (arr [0]); Queue < Node > q = new Queue < Node > (); q . Enqueue (root); int i = 1 ; while (q . Count > 0) { Node curr = q . Dequeue (); // If left node is not null if (arr [i] != - 1) { Node left = new Node (arr [i]); curr . left = left ; q . Enqueue (left); } i ++ ; // If right node is not null if (arr [i] != - 1) { Node right = new Node (arr [i]); curr . right = right ; q . Enqueue (right); } i ++ ; } return root ; } static void Main (string [] args) { // 10 // / \ // 20 30 // / \ // 40 60 Node root = new Node (10); root . left = new Node (20); root . right = new Node (30); root . left . left = new Node (40); root . left . right = new Node (60); List < int > arr = serialize (root); // Variable assigned but never used #pragma warning disable CS0219 Node res = deSerialize (arr); #pragma warning restore CS0219 } } JavaScript // Queue node class QNode { constructor (data) { this . data = data ; this . next = null ; } } // Custom Queue implementation 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 ; } } // Node Structure class Node { constructor (x) { this . data = x ; this . left = null ; this . right = null ; } } // function to serialize a tree. function serialize (root) { const arr = []; const q = new Queue (); q . enqueue (root); while (! q . isEmpty ()) { const curr = q . dequeue (); // If curr node is null, // append -1 to result. if (curr

=== null) { arr . push (- 1); continue ; } // else push its value into result arr . push (curr . data); // enqueue children q . enqueue (curr . left); q . enqueue (curr . right); } return arr ; } // function to deserialize a tree. function deserialize (arr) { if (arr [0] === - 1) return null ; const root = new Node (arr [0]); const q = new Queue (); q . enqueue (root); let i = 1 ; while (! q . isEmpty ()) { const curr = q . dequeue (); // Left child if (arr [i] !== - 1) { const left = new Node (arr [i]); curr . left = left ; q . enqueue (left); } i ++ ; // Right child if (arr [i] !== - 1) { const right = new Node (arr [i]); curr . right = right ; q . enqueue (right); } i ++ ; } return root ; } // Driver Code // 10 // / \ // 20 30 // / \ // 40 60 const root = new Node (10); root . left = new Node (20); root . right = new Node (30); root . left . left = new Node (40); root . left . right = new Node (60); const arr = serialize (root); const res = deserialize (arr); Comment Article Tags: Article Tags: Tree DSA Microsoft Amazon Adobe Yahoo Flipkart Paytm Accolite InMobi MAQ Software Linkedin Quikr + 9 More