Inorder predecessor and successor in BST - GeeksforGeeks

Source: https://www.geeksforgeeks.org/inorder-predecessor-successor-given-key-bst/

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 Inorder predecessor and successor in BST Last Updated : 13 Oct, 2025 Given the root of a Binary Search Tree (BST) and an integer key , find the predecessor and successor of the given key. The predecessor of a node is the node with the largest value smaller than the key. The successor of a node is the node with the smallest value greater than the key. If the predecessor does not exist, return NULL for the predecessor. If the successor does not exist, return NULL for the successor. Note: The given key may or may not be present in the BST. Examples: Input: key = 65 Output: [60, 70] Explanation: In given BST the inorder predecessor of 65 is 60 and inorder successor of 65 is 70. Input: key = 8 Output: [4, 9] Explanation: In the given BST the inorder predecessor of 8 is 4 and inorder successor of 8 is 9. Try it on GfG Practice Table of Content [Naive Approach] Maintaining predecessor and successors - O(n) Time and O(1) Space [Expected Approach-1] Using Two Traversals - O(h) Time and O(1) Space [Expected Approach-2] Using Single Traversal - O(h) Time and O(1) Space [Naive Approach] Maintaining predecessor and successors - O(n) Time and O(1) Space We traverse the tree and maintain two values for predecessor and successors and compare the current node's value with both of them. If current predecessor < node.data and node.data < key, update the current predecessor with node's value. If current successor > node.data and node.data > key, update the current successor with node's value. C++ //Driver Code Starts #include <iostream> #include <vector> using namespace std ; // Node Structure class Node { public : int data ; Node * left ; Node * right ; Node (int x) { data = x ; left = right = nullptr ; } }; //Driver Code Ends // traversal to find // predecessor and successor void preorder (Node * root , int key , Node *& pre , Node *& suc) { if (root == nullptr) return ; if (root -> data < key && (pre == nullptr || pre -> data < root -> data)) { pre = root ; } if (root -> data > key && (suc == nullptr || suc -> data > root -> data)) { suc = root ; } preorder (root -> left , key , pre , suc); preorder (root -> right , key , pre , suc); } // return vector with predecessor at // index 0 and successor at index 1 vector < Node *> findPreSuc (Node * root , int key) { Node * pre = nullptr ; Node * suc = nullptr ; preorder (root , key , pre , suc); return { pre , suc }; } //Driver Code Starts int main () { // Create BST: // 50 // / \ // 30 70 // / \ / \ // 20 40 60 80 int key = 65 ; Node * root = new Node (50); root -> left = new Node (30); root -> right = new Node (70); root -> left -> left = new Node (20); root -> left -> right = new Node (40); root -> right -> left = new Node (60); root -> right -> right = new Node (80); vector < Node *> result = findPreSuc (root , key); Node * pre = result [0]; Node * suc = result [1]; cout << (pre ? to_string (pre -> data) : "NULL") << " " ; cout << (suc ? to_string (suc -> data) : "NULL") << endl ; } //Driver Code Ends Java //Driver Code Starts import java.util.ArrayList ; import java.util.Arrays ; // Node Structure class Node { int data ; Node left , right ; Node (int x) { data = x ; left = right = null ; } } class GFG { //Driver Code Ends // return ArrayList with predecessor at index 0 // and successor at index 1 static ArrayList < Node > findPreSuc (Node root , int key) { Node [] pre = new Node [1] ; Node [] suc = new Node [1] ; preorder (root , key , pre , suc); return new ArrayList <> (Arrays . asList (pre [0] , suc [0])); } // traversal to find predecessor and successor static void preorder (Node root , int key , Node [] pre , Node [] suc) { if (root == null) return ; if (root . data < key && (pre [0] == null || pre [0] . data < root . data)) { pre [0] = root ; } if (root . data > key && (suc [0] == null || suc [0] . data > root . data)) { suc [0] = root ; } preorder (root . left , key , pre , suc); preorder (root . right , key , pre , suc); } //Driver Code Starts public static void main (String [] args) { // Create BST: // 50 // / \ // 30 70 // / \ / \ // 20 40 60

80 int key = 65 ; Node root = new Node (50); root . left = new Node (30); root . right = new Node (70); root . left . left = new Node (20); root . left . right = new Node (40); root . right . left = new Node (60); root . right . right = new Node (80); ArrayList < Node > result = findPreSuc (root , key); Node pre = result . get (0); Node suc = result . get (1); System . out . print ((pre == null) ? "NULL " : (pre . data + " ")); System . out . print ((suc == null) ? "NULL " : (suc . data + " ")); } } //Driver Code Ends Python #Driver Code Starts # Node Structure class Node : def __init__ (self , x): self . data = x self . left = None self . right = None #Driver Code Ends # traversal to find predecessor and successor def preorder (root , key , pre , suc): if root is None : return if root . data < key and (pre [0] is None or pre [0] . data < root . data): pre [0] = root if root . data > key and (suc [0] is None or suc [0] . data > root . data): suc [0] = root preorder (root . left , key , pre , suc) preorder (root . right , key , pre , suc) # return list with predecessor at index 0 and successor at index 1 def findPreSuc (root , key): pre = [None] suc = [None] preorder (root , key , pre , suc) return [pre [0], suc [0]] #Driver Code Starts if __name__ == "__main__" : # Create BST: # 50 # / \ # 30 70 # / \ / \ # 20 40 60 80 key = 65 root = Node (50) root . left = Node (30) root . right = Node (70) root . left . left = Node (20) root . left . right = Node (40) root . right . left = Node (60) root . right . right = Node (80) result = findPreSuc (root , key) pre = result [0] suc = result [1] print ((str (pre . data) if pre else "NULL"), (str (suc . data) if suc else "NULL")) #Driver Code Ends C# //Driver Code Starts using System ; using System.Collections.Generic ; // Node Structure class Node { public int data ; public Node left , right ; public Node (int x) { data = x ; left = right = null ; } } class GFG { //Driver Code Ends // return List with predecessor at index 0 // and successor at index 1 static List < Node > findPreSuc (Node root , int key) { Node [] pre = new Node [1]; Node [] suc = new Node [1]; preorder (root , key , pre , suc); return new List < Node > { pre [0], suc [0] }; } // traversal to find // predecessor and successor static void preorder (Node root , int key , Node [] pre , Node [] suc) { if (root == null) return ; if (root . data < key && (pre [0] == null || pre [0]. data < root . data)) { pre [0] = root ; } if (root . data > key && (suc [0] == null || suc [0]. data > root . data)) { suc [0] = root ; } preorder (root . left , key , pre , suc); preorder (root . right , key , pre , suc); } //Driver Code Starts public static void Main () { // Create BST: // 50 // / \ // 30 70 // / \ / \ // 20 40 60 80 int key = 65 ; Node root = new Node (50); root . left = new Node (30); root . right = new Node (70); root . left . left = new Node (20); root . left . right = new Node (40); root . right . left = new Node (60); root . right . right = new Node (80); List < Node > result = findPreSuc (root , key); Node pre = result [0]; Node suc = result [1]; Console . Write ((pre == null ? "NULL " : pre . data + " ")); Console . Write ((suc == null ? "NULL " : suc . data + " ")); } } //Driver Code Ends JavaScript //Driver Code Starts // Node Structure class Node { constructor (x) { this . data = x ; this . left = null ; this . right = null ; } } //Driver Code Ends // traversal to find predecessor and successor function preorder (root , key , pre , suc) { if (root === null) return ; if (root . data < key && (pre [0] === null || pre [0]. data < root . data)) { pre [0] = root ; } if (root . data > key && (suc [0] === null || suc [0]. data > root . data)) { suc [0] = root ; } preorder (root . left , key , pre , suc); preorder (root . right , key , pre , suc); } // return array with predecessor at index 0 and successor at index 1 function findPreSuc (root , key) { let pre = [null]; let suc = [null]; preorder (root , key , pre , suc); return [pre [0], suc [0]]; } //Driver Code Starts // Driver code // Create BST: // 50 // / \ // 30 70 // / \ / \ // 20 40 60 80 let key = 65 ; let root = new Node (50); root . left = new Node (30); root . right = new Node (70); root . left . left = new Node (20); root . left . right = new Node (40); root . right . left = new Node (60); root . right . right = new Node (80); let result = findPreSuc (root , key); let pre = result [0]; let suc = result [1]; console . log ((pre ? pre . data : "NULL") + " " + (suc ? suc . data : "NULL")); //Driver Code Ends Output 60 70 [Expected Approach-1] Using Two Traversals - O(h) Time and O(1) Space We traverse the tree twice — once to find the predecessor and once to find the successor. For predecessor , if root < key, and then root becomes the predecessor and we search for larger value. if root >= key, then as predecessor < key, therefore we search in left subtree. For successor, if root <= key, then as successor > key, therefore we search in right subtree. if root > key, and then root becomes the successor and we search for smaller value. C++ //Driver Code Starts #include <vector> #include <iostream> using namespace std ; // Node Structure class Node { public : int data ; Node * left ; Node * right ; Node (int val) { data = val ; left = right = nullptr ; } }; //Driver Code Ends // Finding predecessor of key Node * findPredecessor (Node * root , int key) { Node * predecessor = NULL ; while (root) { if (key > root -> data) { // potential predecessor predecessor = root ; // look for larger predecessors root = root -> right ; } else { root = root -> left ; } } return predecessor ; } // Finding successor of key Node * findSuccessor (Node * root , int key) { Node * successor = NULL ; while (root) { if (key < root -> data) { // potential successor successor = root ; // look for smaller successor root = root -> left ; } else { root = root -> right ; } } return successor ; } // return vector with predecessor at index 0 // and successor at index 1 vector < Node *> findPreSuc (Node *

```
root , int key ) { Node * pre = findPredecessor ( root , key ); Node * suc = findSuccessor ( root , key ); return { pre , suc }; } //Driver Code Starts int main () { // Create BST: // 50 // / \ // 30 70 // / \ / \ // 20 40 60 80 Node * root = new Node ( 50 ); root -> left = new Node ( 30 ); root -> right = new Node ( 70 ); root -> left -> left = new Node ( 20 ); root -> left -> right = new Node ( 40 ); root -> right -> left = new Node ( 60 ); root -> right -> right = new Node ( 80 ); int key = 65 ; vector < Node *> result = findPreSuc ( root , key ); Node * pre = result [ 0 ]; Node * suc = result [ 1 ]; cout << ( pre ? to_string ( pre -> data ) : string ( "NULL" )) << " " << ( suc ? to_string ( suc -> data ) : string ( "NULL" )); return 0 ; } //Driver Code Ends Java //Driver Code Starts import java.util.ArrayList ; import java.util.Arrays ; // Node Structure class Node { int data ; Node left , right ; Node ( int x ) { data = x ; left = right = null ; } } class GFG { //Driver Code Ends // return ArrayList with predecessor at index 0 // and successor at index 1 static ArrayList < Node > findPreSuc ( Node root , int key ) { Node predecessor = findPredecessor ( root , key ); Node successor = findSuccessor ( root , key ); return new ArrayList <> ( Arrays . asList ( predecessor , successor )); } // Finding predecessor of key static Node findPredecessor ( Node root , int key ) { Node predecessor = null ; while ( root != null ) { if ( key > root . data ) { // potential predecessor predecessor = root ; // look for larger predecessors root = root . right ; } else { root = root . left ; } } return predecessor ; } // Finding successor of key static Node findSuccessor ( Node root , int key ) { Node successor = null ; while ( root != null ) { if ( key < root . data ) { // potential successor successor = root ; // look for smaller successors root = root . left ; } else { root = root . right ; } } return successor ; } //Driver Code Starts public static void main ( String [] args ) { // Create BST: // 50 // / \ // 30 70 // / \ / \ // 20 40 60 80 int key = 65 ; Node root = new Node ( 50 ); root . left = new Node ( 30 ); root . right = new Node ( 70 ); root . left . left = new Node ( 20 ); root . left . right = new Node ( 40 ); root . right . left = new Node ( 60 ); root . right . right = new Node ( 80 ); ArrayList < Node > result = findPreSuc ( root , key ); Node pre = result . get ( 0 ); Node suc = result . get ( 1 ); System . out . print (( pre != null ? pre . data : "NULL" ) + " " ); System . out . print ( suc != null ? suc . data : "NULL" ); } } //Driver Code Ends Python #Driver Code Starts # Node Structure class Node : def __init__ ( self , x ): self . data = x self . left = None self . right = None #Driver Code Ends # Finding predecessor of key def findPredecessor ( root , key ): predecessor = None while root : if key > root . data : # potential predecessor predecessor = root ; # look for larger predecessors root = root . right else : root = root . left return predecessor # Finding successor of key def findSuccessor ( root , key ): successor = None while root : if key < root . data : # potential successor successor = root # look for smaller successor root = root . left else : root = root . right return successor # return list with predecessor at index 0 # and successor at index 1 def findPreSuc ( root , key ): return [ findPredecessor ( root , key ), findSuccessor ( root , key )] #Driver Code Starts if __name__ == '__main__' : # Create BST: # 50 # / \ # 30 70 # / \ / \ # 20 40 60 80 root = Node ( 50 ) root . left = Node ( 30 ) root . right = Node ( 70 ) root . left . left = Node ( 20 ) root . left . right = Node ( 40 ) root . right . left = Node ( 60 ) root . right . right = Node ( 80 ) key = 65 pre , suc = findPreSuc ( root , key ) print (( pre . data if pre else "NULL" ), ( suc . data if suc else "NULL" )) #Driver Code Ends C# //Driver Code Starts using System ; using System.Collections.Generic ; // Node Structure class Node { public int data ; public Node left , right ; public Node ( int x ) { data = x ; left = right = null ; } } class GFG { //Driver Code Ends // Finding predecessor of key static Node findPredecessor ( Node root , int key ) { Node predecessor = null ; while ( root != null ) { if ( key > root . data ) { // potential predecessor predecessor = root ; // look for larger predecessors root = root . right ; } else { root = root . left ; } } return predecessor ; } // Finding successor of key static Node findSuccessor ( Node root , int key ) { Node successor = null ; while ( root != null ) { if ( key < root . data ) { // potential successor successor = root ; // look for smaller successors root = root . left ; } else { root = root . right ; } } return successor ; } // return List with predecessor at index 0 // and successor at index 1 static List < Node > findPreSuc ( Node root , int key ) { Node pre = findPredecessor ( root , key ); Node suc = findSuccessor ( root , key ); return new List < Node > { pre , suc }; } //Driver Code Starts static void Main () { // Create BST: // 50 // / \ // 30 70 // / \ / \ // 20 40 60 80 Node root = new Node ( 50 ); root . left = new Node ( 30 ); root . right = new Node ( 70 ); root . left . left = new Node ( 20 ); root . left . right = new Node ( 40 ); root . right . left = new Node ( 60 ); root . right . right = new Node ( 80 ); int key = 65 ; List < Node > result = findPreSuc ( root , key ); Node pre = result [ 0 ]; Node suc = result [ 1 ]; Console . WriteLine ( $"{(pre != null ? pre.data.ToString() : " NULL ")} " + $"{(suc != null ? suc.data.ToString() : " NULL ")}" ); } } //Driver Code Ends JavaScript //Driver Code Starts // Node Structure class Node { constructor ( x ) { this . data = x ; this . left = this . right = null ; } } //Driver Code Ends // Finding predecessor of key function findPredecessor ( root , key ) { let predecessor = null ; while ( root ) { if ( key > root . data ) { // potential predecessor predecessor = root ; // look for larger predecessors root = root . right ; } else { root = root . left ; } } return predecessor ; } // Finding successor of key function findSuccessor ( root , key ) { let successor = null ; while ( root ) { if (
```

key < root . data) { // potential successor successor = root ; // look for smaller successor root = root . left ; } else { root = root . right ; } } return successor ; } // return array with predecessor at index 0 // and successor at index 1 function findPreSuc (root , key) { return [findPredecessor (root , key), findSuccessor (root , key)]; } //Driver Code Starts // Driver Code // Create BST: // 50 // / \ // 30 70 // / \ / \ // 20 40 60 80 let root = new Node (50); root . left = new Node (30); root . right = new Node (70); root . left . left = new Node (20); root . left . right = new Node (40); root . right . left = new Node (60); root . right . right = new Node (80); let key = 65 ; let [pre , suc] = findPreSuc (root , key); console . log ((pre ? pre . data : "NULL") + " " + (suc ? suc . data : "NULL")); //Driver Code Ends Output 60 70 [Expected Approach-2] Using Single Traversal - O(h) Time and O(1) Space The idea is to traverse the BST once to find both the predecessor and successor of a given key. While traversing, if the current node’s value is less than the key, it could be the predecessor. To find a larger value that is still smaller than the key, we move to the right subtree. Any potential successor must be in the right subtree if it exists. If the current node’s value is greater than the key, it could be the successor. To find a smaller value that is still greater than the key, we move to the left subtree. Any potential predecessor must be in the left subtree if it exists. If the current node’s value is equal to the key, the predecessor is the maximum value in its left subtree, and the successor is the minimum value in its right subtree. C++ //Driver Code Starts #include <vector> #include <iostream> using namespace std ; // Node Structure class Node { public : int data ; Node * left ; Node * right ; Node (int val) { data = val ; left = right = nullptr ; } }; //Driver Code Ends Node * rightMost (Node * node) { while (node -> right) node = node -> right ; return node ; } Node * leftMost (Node * node) { while (node -> left) node = node -> left ; return node ; } // return vector with predecessor at index 0 // and successor at index 1 vector < Node *> findPreSuc (Node * root , int key) { Node * pre = NULL ; Node * suc = NULL ; Node * curr = root ; while (curr) { if (curr -> data < key) { pre = curr ; // look for predecessor with greater value curr = curr -> right ; } else if (curr -> data > key) { suc = curr ; // look for successor with smaller value curr = curr -> left ; } else { if (curr -> left) pre = rightMost (curr -> left); if (curr -> right) suc = leftMost (curr -> right); break ; } } return { pre , suc }; } //Driver Code Starts int main () { int key = 65 ; // Create BST: // 50 // / \ // 30 70 // / \ / \ // 20 40 60 80 Node * root = new Node (50); root -> left = new Node (30); root -> right = new Node (70); root -> left -> left = new Node (20); root -> left -> right = new Node (40); root -> right -> left = new Node (60); root -> right -> right = new Node (80); vector < Node *> result = findPreSuc (root , key); Node * pre = result [0]; Node * suc = result [1]; cout << (pre ? to_string (pre -> data) : "NULL") << " " ; cout << (suc ? to_string (suc -> data) : "NULL"); } //Driver Code Ends Java //Driver Code Starts import java.util.ArrayList ; // Node Structure class Node { int data ; Node left , right ; Node (int x) { data = x ; left = right = null ; } } class GFG { //Driver Code Ends static Node rightMost (Node node) { while (node . right != null) { node = node . right ; } return node ; } static Node leftMost (Node node) { while (node . left != null) { node = node . left ; } return node ; } // return ArrayList with predecessor at index 0 // and successor at index 1 static ArrayList < Node > findPreSuc (Node root , int key) { Node pre = null , suc = null ; Node curr = root ; while (curr != null) { if (curr . data < key) { pre = curr ; // look for predecessor with greater value curr = curr . right ; } else if (curr . data > key) { suc = curr ; // look for successor with smaller value curr = curr . left ; } else { if (curr . left != null) pre = rightMost (curr . left); if (curr . right != null) suc = leftMost (curr . right); break ; } } ArrayList < Node > result = new ArrayList <> (); result . add (pre); result . add (suc); return result ; } //Driver Code Starts public static void main (String [] args) { int key = 65 ; // Create BST: // 50 // / \ // 30 70 // / \ / \ // 20 40 60 80 Node root = new Node (50); root . left = new Node (30); root . right = new Node (70); root . left . left = new Node (20); root . left . right = new Node (40); root . right . left = new Node (60); root . right . right = new Node (80); ArrayList < Node > result = findPreSuc (root , key); Node pre = result . get (0); Node suc = result . get (1); System . out . print ((pre == null) ? "NULL " : (pre . data + " ")); System . out . print ((suc == null) ? "NULL " : (suc . data + " ")); } } //Driver Code Ends Python #Driver Code Starts # Node Structure class Node : def __init__ (self , x): self . data = x self . left = None self . right = None #Driver Code Ends def rightMost (node): while node . right : node = node . right return node def leftMost (node): while node . left : node = node . left return node # return list with predecessor at index 0 # and successor at index 1 def findPreSuc (root , key): pre , suc = None , None curr = root while curr : if curr . data < key : pre = curr # look for predecessor with greater value curr = curr . right elif curr . data > key : suc = curr # look for successor with smaller value curr = curr . left else : if curr . left : pre = rightMost (curr . left) if curr . right : suc = leftMost (curr . right) break return [pre , suc] #Driver Code Starts if __name__ == "__main__" : key = 65 # Create BST: # 50 # / \ # 30 70 # / \ / \ # 20 40 60 80 root = Node (50) root . left = Node (30) root . right = Node (70) root . left . left = Node (20) root . left . right = Node (40) root . right . left = Node (60) root . right . right = Node (80) pre , suc = findPreSuc (root ,

key) print ((pre . data if pre else "NULL"), (suc . data if suc else "NULL")) #Driver Code Ends C# //Driver Code Starts using System ; using System.Collections.Generic ; // Node Structure class Node { public int data ; public Node left , right ; public Node (int x) { data = x ; left = right = null ; } } class GfG { //Driver Code Ends static Node RightMost (Node node) { while (node . right != null) node = node . right ; return node ; } static Node LeftMost (Node node) { while (node . left != null) node = node . left ; return node ; } // return List with predecessor at index 0 // and successor at index 1 public static List < Node > FindPreSuc (Node root , int key) { Node pre = null , suc = null ; Node curr = root ; while (curr != null) { if (curr . data < key) { pre = curr ; // look for predecessor with greater value curr = curr . right ; } else if (curr . data > key) { suc = curr ; // look for successor with smaller value curr = curr . left ; } else { if (curr . left != null) pre = RightMost (curr . left); if (curr . right != null) suc = LeftMost (curr . right); break ; } } return new List < Node > { pre , suc }; } //Driver Code Starts public static void Main () { int key = 65 ; // Create BST: // 50 // / \ // 30 70 // / \ / \ // 20 40 60 80 Node root = new Node (50); root . left = new Node (30); root . right = new Node (70); root . left . left = new Node (20); root . left . right = new Node (40); root . right . left = new Node (60); root . right . right = new Node (80); List < Node > result = FindPreSuc (root , key); Node pre = result [0]; Node suc = result [1]; Console . Write ((pre == null ? "NULL " : pre . data + " ")); Console . Write ((suc == null ? "NULL" : suc . data . ToString ())); } } //Driver Code Ends Javascript //Driver Code Starts // Node Structure class Node { constructor (x) { this . data = x ; this . left = null ; this . right = null ; } } //Driver Code Ends function leftMost (node) { while (node . left) node = node . left ; return node ; } function rightMost (node) { while (node . right) node = node . right ; return node ; } // return array with predecessor at index 0 // and successor at index 1 function findPreSuc (root , key) { let pre = null , suc = null ; let curr = root ; while (curr) { if (curr . data < key) { pre = curr ; // look for predecessor with greater value curr = curr . right ; } else if (curr . data > key) { suc = curr ; // look for successor with smaller value curr = curr . left ; } else { if (curr . left) pre = rightMost (curr . left); if (curr . right) suc = leftMost (curr . right); break ; } } return [pre , suc]; } //Driver Code Starts // Driver code // Create BST: // 50 // / \ // 30 70 // / \ / \ // 20 40 60 80 let key = 65 ; let root = new Node (50); root . left = new Node (30); root . right = new Node (70); root . left . left = new Node (20); root . left . right = new Node (40); root . right . left = new Node (60); root . right . right = new Node (80); let [pre , suc] = findPreSuc (root , key); console . log ((pre ? pre . data : "NULL") + " " + (suc ? suc . data : "NULL")); //Driver Code Ends Output 60 70 Comment Article Tags: Article Tags: Tree Binary Search Tree Advanced Data Structure DSA