GeeksforGeeks
A computer science portal for geeks

Custom Search

COURSES

HIRE WITH US

Binary Search Tree | Set 2 (Delete)

We have discussed BST search and insert operations. In this post, delete operation is discussed. When we delete a node, three possibilities arise.

1) ***Node to be deleted is leaf:*** Simply remove from the tree.

```
          50                            50
       /     \         delete(20)      /   \
      30      70       --------->    30     70
     /  \    /  \                     \    /  \
    20   40  60   80                   40  60   80
```

2) ***Node to be deleted has only one child:*** Copy the child to the node and delete the child

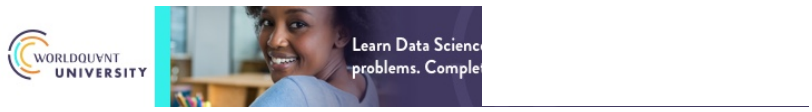
```
          50                            50
       /     \         delete(30)      /   \
      30      70       --------->    40     70
        \    /  \                           /  \
        40  60   80                       60   80
```

3) ***Node to be deleted has two children:*** Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

```
          50                            60
       /     \         delete(50)      /   \
      40      70       --------->    40    70
             /  \                             \
            60   80                            80
```

The important thing to note is, inorder successor is needed only when right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in right child of the node.

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.



C/C++

```
// C program to demonstrate delete operation in binary search tree
#include<stdio.h>
#include<stdlib.h>

struct node
{
    int key;
    struct node *left, *right;
};
```

```
// A utility function to create a new BST node
struct node *newNode(int item)
{
    struct node *temp =  (struct node *)malloc(sizeof(struct node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to do inorder traversal of BST
void inorder(struct node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}

/* A utility function to insert a new node with given key in BST */
struct node* insert(struct node* node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL) return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left  = insert(node->left, key);
    else
        node->right = insert(node->right, key);

    /* return the (unchanged) node pointer */
    return node;
}

/* Given a non-empty binary search tree, return the node with minimum
   key value found in that tree. Note that the entire tree does not
   need to be searched. */
struct node * minValueNode(struct node* node)
{
    struct node* current = node;

    /* loop down to find the leftmost leaf */
    while (current && current->left != NULL)
        current = current->left;

    return current;
}

/* Given a binary search tree and a key, this function deletes the key
   and returns the new root */
struct node* deleteNode(struct node* root, int key)
{
    // base case
    if (root == NULL) return root;

    // If the key to be deleted is smaller than the root's key,
    // then it lies in left subtree
    if (key < root->key)
        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater than the root's key,
    // then it lies in right subtree
    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    // if key is same as root's key, then This is the node
    // to be deleted
    else
    {
        // node with only one child or no child
        if (root->left == NULL)
        {
            struct node *temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL)
        {
            struct node *temp = root->left;
```

```
            free(root);
            return temp;
        }

        // node with two children: Get the inorder successor (smallest
        // in the right subtree)
        struct node* temp = minValueNode(root->right);

        // Copy the inorder successor's content to this node
        root->key = temp->key;

        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}

// Driver Program to test above functions
int main()
{
    /* Let us create following BST
              50
           /     \
          30      70
         /  \    /  \
        20   40  60   80 */
    struct node *root = NULL;
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);

    printf("Inorder traversal of the given tree \n");
    inorder(root);

    printf("\nDelete 20\n");
    root = deleteNode(root, 20);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);

    printf("\nDelete 30\n");
    root = deleteNode(root, 30);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);

    printf("\nDelete 50\n");
    root = deleteNode(root, 50);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);

    return 0;
}
```

Java

```
// Java program to demonstrate delete operation in binary search tree
class BinarySearchTree
{
    /* Class containing left and right child of current node and key value*/
    class Node
    {
        int key;
        Node left, right;

        public Node(int item)
        {
            key = item;
            left = right = null;
        }
    }

    // Root of BST
    Node root;

    // Constructor
    BinarySearchTree()
```

```
{
    root = null;
}

// This method mainly calls deleteRec()
void deleteKey(int key)
{
    root = deleteRec(root, key);
}

/* A recursive function to insert a new key in BST */
Node deleteRec(Node root, int key)
{
    /* Base Case: If the tree is empty */
    if (root == null)  return root;

    /* Otherwise, recur down the tree */
    if (key < root.key)
        root.left = deleteRec(root.left, key);
    else if (key > root.key)
        root.right = deleteRec(root.right, key);

    // if key is same as root's key, then This is the node
    // to be deleted
    else
    {
        // node with only one child or no child
        if (root.left == null)
            return root.right;
        else if (root.right == null)
            return root.left;

        // node with two children: Get the inorder successor (smallest
        // in the right subtree)
        root.key = minValue(root.right);

        // Delete the inorder successor
        root.right = deleteRec(root.right, root.key);
    }

    return root;
}

int minValue(Node root)
{
    int minv = root.key;
    while (root.left != null)
    {
        minv = root.left.key;
        root = root.left;
    }
    return minv;
}

// This method mainly calls insertRec()
void insert(int key)
{
    root = insertRec(root, key);
}

/* A recursive function to insert a new key in BST */
Node insertRec(Node root, int key)
{

    /* If the tree is empty, return a new node */
    if (root == null)
    {
        root = new Node(key);
        return root;
    }

    /* Otherwise, recur down the tree */
    if (key < root.key)
        root.left = insertRec(root.left, key);
    else if (key > root.key)
        root.right = insertRec(root.right, key);

    /* return the (unchanged) node pointer */
    return root;
}
```

```
    // This method mainly calls InorderRec()
    void inorder()
    {
        inorderRec(root);
    }

    // A utility function to do inorder traversal of BST
    void inorderRec(Node root)
    {
        if (root != null)
        {
            inorderRec(root.left);
            System.out.print(root.key + " ");
            inorderRec(root.right);
        }
    }

    // Driver Program to test above functions
    public static void main(String[] args)
    {
        BinarySearchTree tree = new BinarySearchTree();

        /* Let us create following BST
              50
           /     \
          30      70
         /  \    /  \
        20   40  60   80 */
        tree.insert(50);
        tree.insert(30);
        tree.insert(20);
        tree.insert(40);
        tree.insert(70);
        tree.insert(60);
        tree.insert(80);

        System.out.println("Inorder traversal of the given tree");
        tree.inorder();

        System.out.println("\nDelete 20");
        tree.deleteKey(20);
        System.out.println("Inorder traversal of the modified tree");
        tree.inorder();

        System.out.println("\nDelete 30");
        tree.deleteKey(30);
        System.out.println("Inorder traversal of the modified tree");
        tree.inorder();

        System.out.println("\nDelete 50");
        tree.deleteKey(50);
        System.out.println("Inorder traversal of the modified tree");
        tree.inorder();
    }
}
```

Python

```
# Python program to demonstrate delete operation
# in binary search tree

# A Binary Tree Node
class Node:

    # Constructor to create a new node
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None


# A utility function to do inorder traversal of BST
def inorder(root):
    if root is not None:
        inorder(root.left)
        print root.key,
        inorder(root.right)
```

```
# A utility function to insert a new node with given key in BST
def insert( node, key):

    # If the tree is empty, return a new node
    if node is None:
        return Node(key)

    # Otherwise recur down the tree
    if key < node.key:
        node.left = insert(node.left, key)
    else:
        node.right = insert(node.right, key)

    # return the (unchanged) node pointer
    return node

# Given a non-empty binary search tree, return the node
# with minum key value found in that tree. Note that the
# entire tree does not need to be searched
def minValueNode( node):
    current = node

    # loop down to find the leftmost leaf
    while(current.left is not None):
        current = current.left

    return current

# Given a binary search tree and a key, this function
# delete the key and returns the new root
def deleteNode(root, key):

    # Base Case
    if root is None:
        return root

    # If the key to be deleted is smaller than the root's
    # key then it lies in  left subtree
    if key < root.key:
        root.left = deleteNode(root.left, key)

    # If the kye to be delete is greater than the root's key
    # then it lies in right subtree
    elif(key > root.key):
        root.right = deleteNode(root.right, key)

    # If key is same as root's key, then this is the node
    # to be deleted
    else:

        # Node with only one child or no child
        if root.left is None :
            temp = root.right
            root = None
            return temp

        elif root.right is None :
            temp = root.left
            root = None
            return temp

        # Node with two children: Get the inorder successor
        # (smallest in the right subtree)
        temp = minValueNode(root.right)

        # Copy the inorder successor's content to this node
        root.key = temp.key

        # Delete the inorder successor
        root.right = deleteNode(root.right , temp.key)


    return root

# Driver program to test above functions
""" Let us create following BST
              50
           /     \
          30      70
         /  \    /  \
```

```
        20   40  60   80 """

root = None
root = insert(root, 50)
root = insert(root, 30)
root = insert(root, 20)
root = insert(root, 40)
root = insert(root, 70)
root = insert(root, 60)
root = insert(root, 80)

print "Inorder traversal of the given tree"
inorder(root)

print "\nDelete 20"
root = deleteNode(root, 20)
print "Inorder traversal of the modified tree"
inorder(root)

print "\nDelete 30"
root = deleteNode(root, 30)
print "Inorder traversal of the modified tree"
inorder(root)

print "\nDelete 50"
root = deleteNode(root, 50)
print "Inorder traversal of the modified tree"
inorder(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

C#

```
// C# program to demonstrate delete
// operation in binary search tree
using System;

public class BinarySearchTree
{
    /* Class containing left and right
    child of current node and key value*/
    class Node
    {
        public int key;
        public Node left, right;

        public Node(int item)
        {
            key = item;
            left = right = null;
        }
    }

    // Root of BST
    Node root;

    // Constructor
    BinarySearchTree()
    {
        root = null;
    }

    // This method mainly calls deleteRec()
    void deleteKey(int key)
    {
        root = deleteRec(root, key);
    }

    /* A recursive function to insert a new key in BST */
    Node deleteRec(Node root, int key)
    {
        /* Base Case: If the tree is empty */
        if (root == null) return root;

        /* Otherwise, recur down the tree */
        if (key < root.key)
            root.left = deleteRec(root.left, key);
        else if (key > root.key)
            root.right = deleteRec(root.right, key);
```

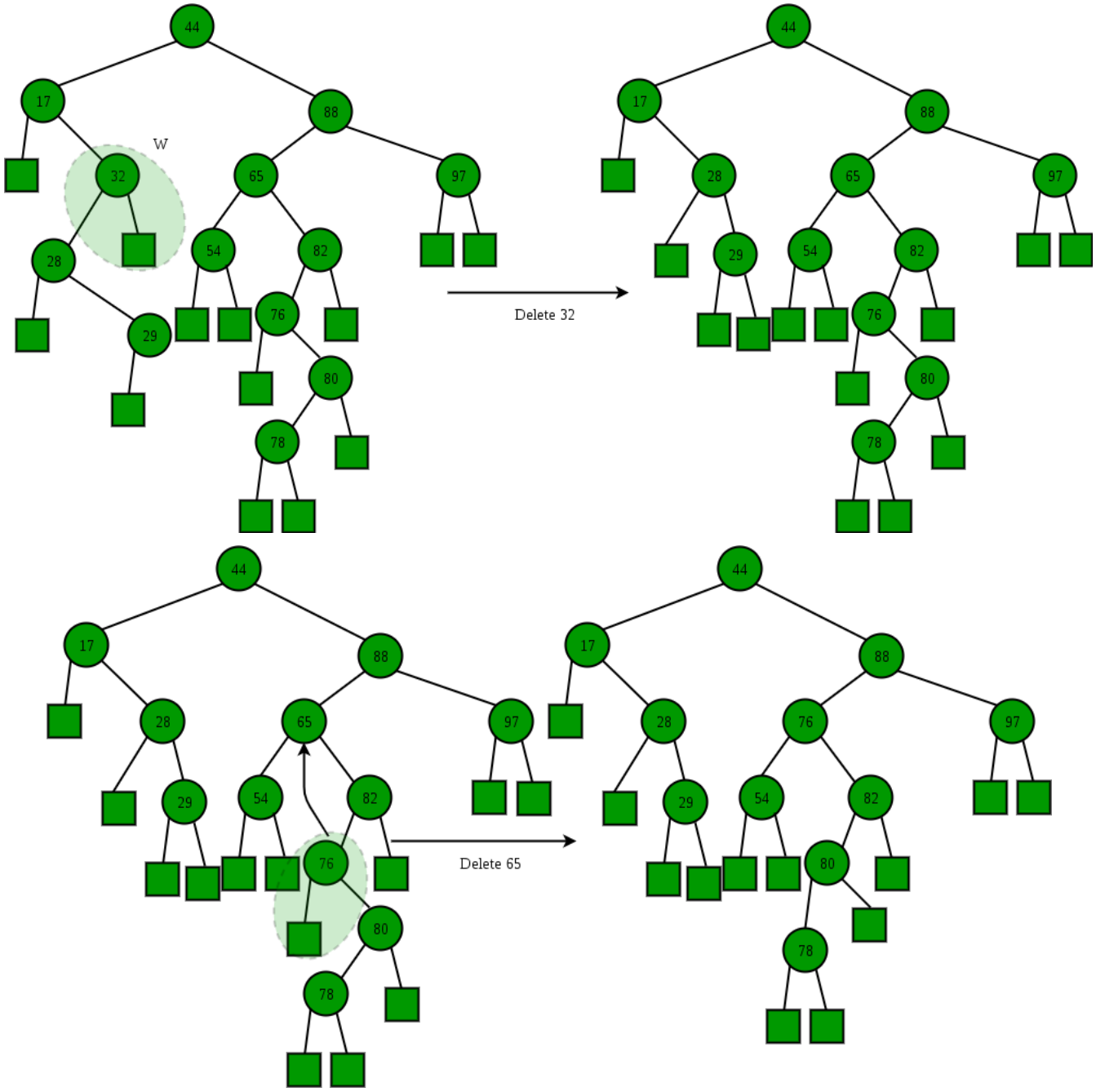
```
        // if key is same as root's key, then This is the node
        // to be deleted
        else
        {
            // node with only one child or no child
            if (root.left == null)
                return root.right;
            else if (root.right == null)
                return root.left;

            // node with two children: Get the
            // inorder successor (smallest
            // in the right subtree)
            root.key = minValue(root.right);

            // Delete the inorder successor
            root.right = deleteRec(root.right, root.key);
        }
        return root;
    }

    int minValue(Node root)
    {
        int minv = root.key;
        while (root.left != null)
        {
            minv = root.left.key;
            root = root.left;
        }
        return minv;
    }

    // This method mainly calls insertRec()
    void insert(int key)
    {
        root = insertRec(root, key);
    }

    /* A recursive function to insert a new key in BST */
    Node insertRec(Node root, int key)
    {

        /* If the tree is empty, return a new node */
        if (root == null)
        {
            root = new Node(key);
            return root;
        }

        /* Otherwise, recur down the tree */
        if (key < root.key)
            root.left = insertRec(root.left, key);
        else if (key > root.key)
            root.right = insertRec(root.right, key);

        /* return the (unchanged) node pointer */
        return root;
    }

    // This method mainly calls InorderRec()
    void inorder()
    {
        inorderRec(root);
    }

    // A utility function to do inorder traversal of BST
    void inorderRec(Node root)
    {
        if (root != null)
        {
            inorderRec(root.left);
            Console.Write(root.key + " ");
            inorderRec(root.right);
        }
    }

    // Driver code
    public static void Main(String[] args)
    {
        BinarySearchTree tree = new BinarySearchTree();
```

```
        /* Let us create following BST
            50
        / \
        30 70
        / \ / \
        20 40 60 80 */
        tree.insert(50);
        tree.insert(30);
        tree.insert(20);
        tree.insert(40);
        tree.insert(70);
        tree.insert(60);
        tree.insert(80);

        Console.WriteLine("Inorder traversal of the given tree");
        tree.inorder();

        Console.WriteLine("\nDelete 20");
        tree.deleteKey(20);
        Console.WriteLine("Inorder traversal of the modified tree");
        tree.inorder();

        Console.WriteLine("\nDelete 30");
        tree.deleteKey(30);
        Console.WriteLine("Inorder traversal of the modified tree");
        tree.inorder();

        Console.WriteLine("\nDelete 50");
        tree.deleteKey(50);
        Console.WriteLine("Inorder traversal of the modified tree");
        tree.inorder();
    }
}

// This code has been contributed
// by PrinciRaj1992
```

Output:

```
Inorder traversal of the given tree
20 30 40 50 60 70 80
Delete 20
Inorder traversal of the modified tree
30 40 50 60 70 80
Delete 30
Inorder traversal of the modified tree
40 50 60 70 80
Delete 50
Inorder traversal of the modified tree
40 60 70 80
```

Illustration:



Time Complexity: The worst case time complexity of delete operation is O(h) where h is height of Binary Search Tree. In worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of delete operation may become O(n)

Optimization to above code for two children case :

In the above recursive code, we recursively call delete() for successor. We can avoid recursive call by keeping track of parent node of successor so that we can simply remove the successor by making child of parent as NULL. We know that successor would always be a leaf node.

```
// C++ program to implement optimized delete in BST.
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int key;
    struct Node *left, *right;
};

// A utility function to create a new BST node
Node* newNode(int item)
{
    Node* temp = new Node;
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to do inorder traversal of BST
void inorder(Node* root)
{
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}

/* A utility function to insert a new node with given key in BST */
Node* insert(Node* node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL)
        return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);

    /* return the (unchanged) node pointer */
    return node;
}

/* Given a binary search tree and a key, this function deletes the key
   and returns the new root */
Node* deleteNode(Node* root, int k)
{
    // Base case
    if (root == NULL)
        return root;
```

```
    // Recursive calls for ancestors of
    // node to be deleted
    if (root->key > k) {
        root->left = deleteNode(root->left, k);
        return root;
    }
    else if (root->key < k) {
        root->right = deleteNode(root->right, k);
        return root;
    }

    // We reach here when root is the node
    // to be deleted.

    // If one of the children is empty
    if (root->left == NULL) {
        Node* temp = root->right;
        delete root;
        return temp;
    }
    else if (root->right == NULL) {
        Node* temp = root->left;
        delete root;
        return temp;
    }

    // If both children exist
    else {

        Node* succParent = root->right;

        // Find successor
        Node *succ = root->right;
        while (succ->left != NULL) {
            succParent = succ;
            succ = succ->left;
        }

        // Delete successor.  Since successor
        // is always left child of its parent
        // we can safely make successor's right
        // right child as left of its parent.
        succParent->left = succ->right;

        // Copy Successor Data to root
        root->key = succ->key;

        // Delete Successor and return root
        delete succ;
        return root;
    }
}

// Driver Program to test above functions
int main()
{
    /* Let us create following BST
              50
           /     \
          30      70
         /  \    /  \
        20   40  60   80 */
    Node* root = NULL;
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);

    printf("Inorder traversal of the given tree \n");
    inorder(root);

    printf("\nDelete 20\n");
    root = deleteNode(root, 20);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);

    printf("\nDelete 30\n");
```

```
    root = deleteNode(root, 30);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);

    printf("\nDelete 50\n");
    root = deleteNode(root, 50);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);

    return 0;
}
```

Output:

```
Inorder traversal of the given tree
20 30 40 50 60 70 80
Delete 20
Inorder traversal of the modified tree
30 40 50 60 70 80
Delete 30
Inorder traversal of the modified tree
40 50 60 70 80
Delete 50
Inorder traversal of the modified tree
40 60 70 80
```

Thanks to wolffgang010 for suggesting above optimization.

Related Links:

- Binary Search Tree Introduction, Search and Insert/a>
- Quiz on Binary Search Tree
- Coding practice on BST
- All Articles on BST

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Recommended Posts:

Binary Search Tree | Set 1 (Search and Insertion)

Count the Number of Binary Search Trees present in a Binary Tree

Binary Tree to Binary Search Tree Conversion using STL set

Binary Tree to Binary Search Tree Conversion

Floor in Binary Search Tree (BST)

Sum of all the levels in a Binary Search Tree

Make Binary Search Tree

Optimal Binary Search Tree | DP-24

Number of pairs with a given sum in a Binary Search Tree

How to handle duplicates in Binary Search Tree?

Inorder Successor in Binary Search Tree

Print all odd nodes of Binary Search Tree

Construct a Binary Search Tree from given postorder

Print Binary Search Tree in Min Max Fashion

Threaded Binary Search Tree | Deletion

Improved By : Manoj Kumar 20, wolffgang010, princiraj1992, Sarvesh Ranjan

Article Tags : Binary Search Tree | Accolite | Amazon | Qualcomm | Samsung

Practice Tags : Amazon | Accolite | Samsung | Qualcomm | Binary Search Tree

33

To-do Done

2.9

Based on **200** vote(s)

Feedback/ Suggest Improvement | Notes | Improve Article

Please write to us at contribute@geeksforgeeks.org to report any issue with the above content.

Writing code in comment? Please use ide.geeksforgeeks.org, generate link and share the link here.

Load Comments

A computer science portal for geeks

5th Floor, A-118,
Sector-136, Noida, Uttar Pradesh - 201305
feedback@geeksforgeeks.org

COMPANY
About Us
Careers
Privacy Policy
Contact Us

LEARN
Algorithms
Data Structures
Languages
CS Subjects
Video Tutorials

PRACTICE
Courses
Company-wise
Topic-wise
How to begin?

CONTRIBUTE
Write an Article
Write Interview Experience
Internships
Videos

@geeksforgeeks, Some rights reserved