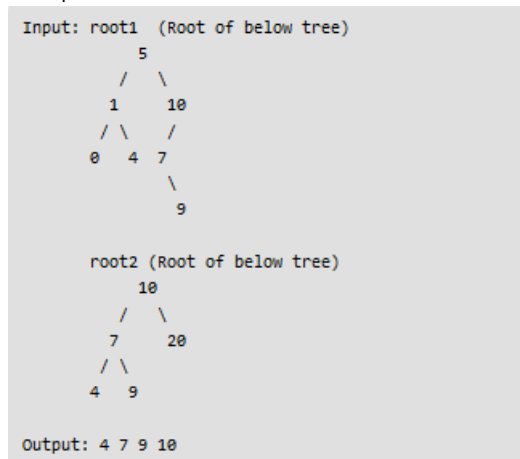


Print Common Nodes in Two Binary Search Trees

Given two Binary Search Trees, find common nodes in them. In other words, find intersection of two BSTs.

Example:



We strongly recommend you to minimize your browser and try this yourself first.

Method 1 (Simple Solution) A simple way is to one by one search every node of first tree in second tree. Time complexity of this solution is $O(m * h)$ where m is number of nodes in first tree and h is height of second tree.

Method 2 (Linear Time) We can find common elements in $O(n)$ time.

- 1) Do inorder traversal of first tree and store the traversal in an auxiliary array `ar1[]`. See `sortedInorder()` [here](#).
- 2) Do inorder traversal of second tree and store the traversal in an auxiliary array `ar2[]`.
- 3) Find intersection of `ar1[]` and `ar2[]`. See [this](#) for details.

Time complexity of this method is $O(m+n)$ where m and n are number of nodes in first and second tree respectively. This solution requires $O(m+n)$ extra space.

Method 3 (Linear Time and limited Extra Space) We can find common elements in $O(n)$ time and $O(h_1 + h_2)$ extra space where h_1 and h_2 are heights of first and second BSTs respectively.

The idea is to use [iterative inorder traversal](#). We use two auxiliary stacks for two BSTs. Since we need to find common elements, whenever we get same element, we print it.

```
// Iterative traversal based method to find common elements
// in two BSTs.
#include<iostream>
#include<stack>
using namespace std;

// A BST node
struct Node
{
    int key;
    struct Node *left, *right;
};

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

// Function to print common elements in given two trees
void printCommon(Node *root1, Node *root2)
```

```

// A utility function to do inorder traversal
void inorder(struct Node *root)
{
    if (root)
    {
        // Create two stacks for two inorder traversals
        stack<Node *> stack1, s1, s2;

        while (1)
        {
            // push the Nodes of first tree in stack s1
            if (root1)
            {
                s1.push(root1);
                root1 = root1->left;
            }

            // push the Nodes of second tree in stack s2
            else if (root2)
            {
                s2.push(root2);
                root2 = root2->left;
            }

            // Both root1 and root2 are NULL here
            else if (!s1.empty() && !s2.empty())
            {
                root1 = s1.top();
                root2 = s2.top();

                // If current keys in two trees are same
                if (root1->key == root2->key)
                {
                    cout << root1->key << " ";
                    s1.pop();
                    s2.pop();

                    // move to the inorder successor
                    root1 = root1->right;
                    root2 = root2->right;
                }

                else if (root1->key < root2->key)
                {
                    // If Node of first tree is smaller, than that of
                    // second tree, then its obvious that the inorder
                    // successors of current Node can have same value
                    // as that of the second tree Node. Thus, we pop
                    // from s2
                    s1.pop();
                    root1 = root1->right;

                    // root2 is set to NULL, because we need
                    // new Nodes of tree 1
                    root2 = NULL;
                }
                else if (root1->key > root2->key)
                {
                    s2.pop();
                    root2 = root2->right;
                    root1 = NULL;
                }
            }

            // Both roots and both stacks are empty
            else break;
        }
    }

    // A utility function to do inorder traversal
    void inorder(struct Node *root)
    {
        if (root)
        {
            inorder(root->left);
            cout<<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 if (key > node->key)
        node->right = insert(node->right, key);

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

// Driver program
int main()
{
    // Create first tree as shown in example
    Node *root1 = NULL;
    root1 = insert(root1, 5);
    root1 = insert(root1, 1);
    root1 = insert(root1, 10);
    root1 = insert(root1, 0);
    root1 = insert(root1, 4);
    root1 = insert(root1, 7);
    root1 = insert(root1, 9);

    // Create second tree as shown in example
    Node *root2 = NULL;
    root2 = insert(root2, 10);
    root2 = insert(root2, 7);
    root2 = insert(root2, 20);
    root2 = insert(root2, 4);
    root2 = insert(root2, 9);

    cout << "Tree 1 : ";
    inorder(root1);
    cout << endl;

    cout << "Tree 2 : ";
    inorder(root2);

    cout << "\nCommon Nodes: ";
    printCommon(root1, root2);

    return 0;
}

```

Output:

```
4 7 9 10
```