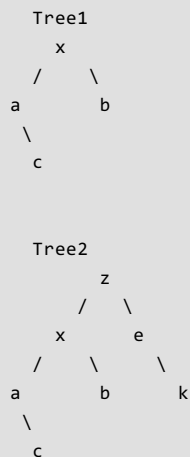


Check if a binary tree is subtree of another binary tree | Set 2

Given two binary trees, check if the first tree is subtree of the second one. A subtree of a tree T is a tree S consisting of a node in T and all of its descendants in T.

The subtree corresponding to the root node is the entire tree; the subtree corresponding to any other node is called a proper subtree.

For example, in the following case, Tree1 is a subtree of Tree2.



We have discussed a $O(n^2)$ solution for this problem. In this post a $O(n)$ solution is discussed. The idea is based on the fact that **inorder and preorder/postorder uniquely identify a binary tree**. Tree S is a subtree of T if both inorder and preorder traversals of S are substrings of inorder and preorder traversals of T respectively.

Following are detailed steps.

- 1) Find inorder and preorder traversals of T, store them in two auxiliary arrays `inT[]` and `preT[]`.
- 2) Find inorder and preorder traversals of S, store them in two auxiliary arrays `inS[]` and `preS[]`.
- 3) If `inS[]` is a subarray of `inT[]` and `preS[]` is a subarray of `preT[]`, then S is a subtree of T. Else not.

We can also use postorder traversal in place of preorder in the above algorithm.

Let us consider the above example

Inorder and Preorder traversals of the big tree are.

`inT[] = {a, c, x, b, z, e, k}`

`preT[] = {z, x, a, c, b, e, k}`

Inorder and Preorder traversals of small tree are

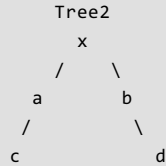
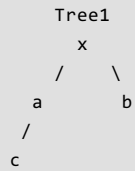
`inS[] = {a, c, x, b}`

`preS[] = {x, a, c, b}`

We can easily figure out that `inS[]` is a subarray of `inT[]` and `preS[]` is a subarray of `preT[]`.

EDIT

The above algorithm doesn't work for cases where a tree is present in another tree, but not as a subtree. Consider the following example.



Inorder and Preorder traversals of the big tree or Tree2 are.

Inorder and Preorder traversals of small tree or Tree1 are

The Tree2 is not a subtree of Tree1, but inS[] and preS[] are subarrays of inT[] and preT[] respectively.

The above algorithm can be extended to handle such cases by adding a special character whenever we encounter NULL in inorder and preorder traversals. Thanks to Shivam Goel for suggesting this extension.

Following is the implementation of above algorithm.

C

```
#include <iostream>
#include <cstring>
using namespace std;
#define MAX 100

// Structure of a tree node
struct Node
{
    char key;
    struct Node *left, *right;
};

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

// A utility function to store inorder traversal of tree rooted
// with root in an array arr[]. Note that i is passed as reference
void storeInorder(Node *root, char arr[], int &i)
{
    if (root == NULL)
    {
        arr[i++] = '$';
        return;
    }
    storeInorder(root->left, arr, i);
    arr[i++] = root->key;
    storeInorder(root->right, arr, i);
}

// A utility function to store preorder traversal of tree rooted
// with root in an array arr[]. Note that i is passed as reference
void storePreOrder(Node *root, char arr[], int &i)
```

```

{
    if (root == NULL)
    {
        arr[i++] = '$';
        return;
    }
    arr[i++] = root->key;
    storePreOrder(root->left, arr, i);
    storePreOrder(root->right, arr, i);
}

/* This function returns true if S is a subtree of T, otherwise false */
bool isSubtree(Node *T, Node *S)
{
    /* base cases */
    if (S == NULL) return true;
    if (T == NULL) return false;

    // Store Inorder traversals of T and S in inT[0..m-1]
    // and inS[0..n-1] respectively
    int m = 0, n = 0;
    char inT[MAX], inS[MAX];
    storeInorder(T, inT, m);
    storeInorder(S, inS, n);
    inT[m] = '\0', inS[n] = '\0';

    // If inS[] is not a substring of preS[], return false
    if (strstr(inT, inS) == NULL)
        return false;

    // Store Preorder traversals of T and S in inT[0..m-1]
    // and inS[0..n-1] respectively
    m = 0, n = 0;
    char preT[MAX], preS[MAX];
    storePreOrder(T, preT, m);
    storePreOrder(S, preS, n);
    preT[m] = '\0', preS[n] = '\0';

    // If inS[] is not a substring of preS[], return false
    // Else return true
    return (strstr(preT, preS) != NULL);
}

// Driver program to test above function
int main()
{
    Node *T = newNode('a');
    T->left = newNode('b');
    T->right = newNode('d');
    T->left->left = newNode('c');
    T->right->right = newNode('e');

    Node *S = newNode('a');
    S->left = newNode('b');
    S->left->left = newNode('c');
    S->right = newNode('d');

    if (isSubtree(T, S))
        cout << "Yes: S is a subtree of T";
    else
        cout << "No: S is NOT a subtree of T";

    return 0;
}

```

Java

```

// Java program to check if binary tree is subtree of another binary tree
class Node {

```

```

char data;
Node left, right;

Node(char item) {
    data = item;
    left = right = null;
}
}

class Passing {

    int i;
    int m = 0;
    int n = 0;
}

class BinaryTree {

    static Node root;
    Passing p = new Passing();

    String strstr(String haystack, String needle) {
        if (haystack == null || needle == null) {
            return null;
        }
        int hLength = haystack.length();
        int nLength = needle.length();
        if (hLength < nLength) {
            return null;
        }
        if (nLength == 0) {
            return haystack;
        }
        for (int i = 0; i <= hLength - nLength; i++) {
            if (haystack.charAt(i) == needle.charAt(0)) {
                int j = 0;
                for (; j < nLength; j++) {
                    if (haystack.charAt(i + j) != needle.charAt(j)) {
                        break;
                    }
                }
                if (j == nLength) {
                    return haystack.substring(i);
                }
            }
        }
        return null;
    }

    // A utility function to store inorder traversal of tree rooted
    // with root in an array arr[]. Note that i is passed as reference
    void storeInorder(Node node, char arr[], Passing i) {
        if (node == null) {
            arr[i.i++] = '$';
            return;
        }
        storeInorder(node.left, arr, i);
        arr[i.i++] = node.data;
        storeInorder(node.right, arr, i);
    }

    // A utility function to store preorder traversal of tree rooted
    // with root in an array arr[]. Note that i is passed as reference
    void storePreOrder(Node node, char arr[], Passing i) {
        if (node == null) {
            arr[i.i++] = '$';
            return;
        }
        arr[i.i++] = node.data;
        storePreOrder(node.left, arr, i);
        storePreOrder(node.right, arr, i);
    }
}

```

```

/* This function returns true if S is a subtree of T, otherwise false */
boolean isSubtree(Node T, Node S) {
    /* base cases */
    if (S == null) {
        return true;
    }
    if (T == null) {
        return false;
    }

    // Store Inorder traversals of T and S in inT[0..m-1]
    // and inS[0..n-1] respectively
    char inT[] = new char[100];
    String op1 = String.valueOf(inT);
    char inS[] = new char[100];
    String op2 = String.valueOf(inS);
    storeInorder(T, inT, p);
    storeInorder(S, inS, p);
    inT[p.m] = '\0';
    inS[p.m] = '\0';

    // If inS[] is not a substring of preS[], return false
    if (strstr(op1, op2) != null) {
        return false;
    }

    // Store Preorder traversals of T and S in inT[0..m-1]
    // and inS[0..n-1] respectively
    p.m = 0;
    p.n = 0;
    char preT[] = new char[100];
    char preS[] = new char[100];
    String op3 = String.valueOf(preT);
    String op4 = String.valueOf(preS);
    storePreOrder(T, preT, p);
    storePreOrder(S, preS, p);
    preT[p.m] = '\0';
    preS[p.n] = '\0';

    // If inS[] is not a substring of preS[], return false
    // Else return true
    return (strstr(op3, op4) != null);
}

//Driver program to test above functions
public static void main(String args[]) {
    BinaryTree tree = new BinaryTree();
    Node T = new Node('a');
    T.left = new Node('b');
    T.right = new Node('d');
    T.left.left = new Node('c');
    T.right.right = new Node('e');

    Node S = new Node('a');
    S.left = new Node('b');
    S.right = new Node('d');
    S.left.left = new Node('c');

    if (tree.isSubtree(T, S)) {
        System.out.println("Yes , S is a subtree of T");
    } else {
        System.out.println("No, S is not a subtree of T");
    }
}

}

// This code is contributed by Mayank Jaiswal

```

Output:

No: S is NOT a subtree of T

Time Complexity: Inorder and Preorder traversals of Binary Tree take $O(n)$ time. The function `strstr()` can also be implemented in $O(n)$ time using [KMP string matching algorithm](#).

Auxiliary Space: $O(n)$