Check if leaf traversal of two Binary Trees is same?

Leaf traversal is sequence of leaves traversed from left to right. The problem is to check if leaf traversals of two given Binary Trees are same or not.

Expected time complexity O(n). Expected auxiliary space O(h1 + h2) where h1 and h2 are heights of two Binary Trees.

Examples:

```
Input: Roots of below Binary Trees
         1
        / \
       2   3
      /   / \
     4   6   7

         0
        /  \
       5    8
        \  / \
        4  6  7
Output: same
Leaf order traversal of both trees is 4 6 7

Input: Roots of below Binary Trees
         0
        / \
       1   2
      / \
     8   9

         1
        / \
       4   3
        \ / \
        8 2  9
```

```
Output: Not Same
Leaf traversals of two trees are different.
For first, it is 8 9 2 and for second it is
8 2 9
```

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

A **Simple Solution** is traverse first tree and store leaves from left and right in an array. Then traverse other tree and store leaves in another array. Finally compare two arrays. If both arrays are same, then return true.

The above solution requires O(m+n) extra space where m and n are nodes in first and second tree respectively.

How to check with O(h1 + h2) space?

The idea is use iterative traversal. Traverse both trees simultaneously, look for a leaf node in both trees and compare the found leaves. All leaves must match.

Algorithm:

```
1. Create empty stacks stack1 and stack2
   for iterative traversals of tree1 and tree2

2. insert (root of tree1) in stack1
   insert (root of tree2) in stack2

3. Stores current leaf nodes of tree1 and tree2
temp1 = (root of tree1)
temp2 = (root of tree2)

4. Traverse both trees using stacks
while (stack1 and stack2 parent empty)
{
    // Means excess leaves in one tree
    if (if one of the stacks are empty)
        return false

   // get next leaf node in tree1
   temp1 = stack1.pop()
   while (temp1 is not leaf node)
   {
        push right child to stack1
        push left child to stack1
   }

   // get next leaf node in tree2
   temp2 = stack2.pop()
   while (temp2 is not leaf node)
   {
        push right child to stack2
```

```
        push left child to stack2
    }

    // If leaves do not match return false
    if (temp1 != temp2)
        return false
 }

 5. If all leaves matched, return true
```

Below is Java implementation of above algorithm.

```
// Java program to check if two Leaf Traversal of
// Two Binary Trees is same or not
import java.util.*;
import java.lang.*;
import java.io.*;

// Binary Tree node
class Node
{
   int data;
   Node left, right;
   public Node(int x)
   {
      data = x;
      left = right = null;
   }
   public boolean isLeaf()
   {
      return (left==null && right==null);
   }
}

class LeafOrderTraversal
{
   // Returns true of leaf traversal of two trees is
   // same, else false
   public static boolean isSame(Node root1, Node root2)
   {
      // Create empty stacks.  These stacks are going
      // to be used for iterative traversals.
      Stack<Node> s1 = new Stack<Node>();
      Stack<Node> s2 = new Stack<Node>();

      s1.push(root1);
      s2.push(root2);

      // Loop until either of two stacks is not empty
      while (!s1.empty() || !s2.empty())
      {
         // If one of the stacks is empty means other
         // stack has extra leaves so return false
         if (s1.empty() || s2.empty())
            return false;

         Node temp1 = s1.pop();
         while (temp1!=null && !temp1.isLeaf())
         {
            // Push right and left children of temp1.
            // Note that right child is inserted
```

```
            // before left
            if (temp1.right != null)
               s1.push(temp1. right);
            if (temp1.left != null)
               s1.push(temp1.left);
            temp1 = s1.pop();
         }

         // same for tree2
         Node temp2 = s2.pop();
         while (temp2!=null && !temp2.isLeaf())
         {
            if (temp2.right != null)
               s2.push(temp2.right);
            if (temp2.left != null)
               s2.push(temp2.left);
            temp2 = s2.pop();
         }

         // If one is null and other is not, then
         // return false
         if (temp1==null && temp2!=null)
            return false;
         if (temp1!=null && temp2==null)
            return false;

         // If both are not null and data is not
         // same return false
         if (temp1!=null && temp2!=null)
         {
            if (temp1.data != temp2.data)
               return false;
         }
      }

      // If control reaches this point, all leaves
      // are matched
      return true;
   }

   // Driver program to test
   public static void main(String[] args)
   {
      // Let us create trees in above example 1
      Node root1 = new Node(1);
      root1.left = new Node(2);
      root1.right = new Node(3);
      root1.left.left = new Node(4);
      root1.right.left = new Node(6);
      root1.right.right = new Node(7);

      Node root2 = new Node(0);
      root2.left = new Node(1);
      root2.right = new Node(5);
      root2.left.right = new Node(4);
      root2.right.left = new Node(6);
      root2.right.right = new Node(7);

      if (isSame(root1, root2))
         System.out.println("Same");
      else
         System.out.println("Not Same");
   }
}
```

Run on IDE

Output:

```
Same
```

This article is contributed by **Kumar Gaurav**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

20 Comments Category: Trees

Related Posts:

- Lowest Common Ancestor in a Binary Tree | Set 2 (Using Parent Pointer)
- Construct a Binary Search Tree from given postorder
- BFS vs DFS for Binary Tree
- Maximum difference between node and its ancestor in Binary Tree
- Inorder Non-threaded Binary Tree Traversal without Recursion or Stack
- Closest leaf to a given node in Binary Tree
- Locking and Unlocking of Resources arranged in the form of n-ary Tree
- Find all possible binary trees with given Inorder Traversal

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

20 Comments GeeksforGeeks Login

Recommend Share Sort by Newest

Join the discussion…

LOGAN • 19 days ago
http://www.universeforyour.com...
http://www.universeforyour.com...
http://www.universeforyour.com...
nice...
• Reply • Share ›

Devansh • 25 days ago
Can you please explain how the space complexity is O(h1 + h2)?
For ex: In a full binary tree, in the level before leaves, the number of nodes in stack will be 2^(h-1).
Can you give the detailed proof for this?
• Reply • Share ›

Anujraaj Goel • 2 months ago
Since temp contains elements popped out of the stack and stack at this stage cannot be empty as ensured by the while loop and if conditions above , then doesn't this ensure the fact that temp will never be null, then why do we explicitly need to check (temp!=null) ?
• Reply • Share ›

IIIT Allahabad Anujraaj Goel • 2 months ago
Consider the case when one (or both) of root1 and root2 =null (in main don't add any node..just keep "Node root1 =null ; ").In this case, both "while" and "if" condition will allow entry. Then those (temp != null) conditions will come to your rescue...without them you'll get a run time error..try it out ! Hope that helps :)
1 • Reply • Share ›

Mysterious Mind • 2 months ago
If the modification is allowed, then this can also be done with o(n) using recursion. Connect leaves in both trees. Check both connected leaves are having same values. Remove the links between connected leaves to restore the original trees.
http://code.geeksforgeeks.org/...
• Reply • Share ›

Ambika • 2 months ago
How simple solution need o(m+n) space, since we have to only store leaf nodes in the arrays. And Maximum leaf nodes for n node tree : floor(log(n))+1, then how it needs

arrays). And Maximum leaf nodes for n node tree : floor(log(n)) + 1, then how it needs O(m+n) space

• Reply • Share ›

kumar ➔ Ambika • 2 months ago

Maximum number of leaves in a binary tree is of O(n). See below for explanation.

For full binary tree:

N = total nodes in binary tee

h = height of binary tree

N = 2^0 + 2^1 + 2^2 + ... + 2^(h-1)

N = 2^h - 1

N+1 = 2^h

(N+1)/2 = 2^(h-1)

Number of Leaf nodes = Number of nodes at last level = 2^(h-1) = (N+1)/2 = O(n)

• Reply • Share ›

Shravan • 2 months ago

Can we not just compare temp1 and temp2 to return FALSE instead of comparing each?

// If one is null and other is not, then

// return false

if (temp1==null && temp2!=null)

return false;

if (temp1!=null && temp2==null)

return false;

// If both are not null and data is not

// same return false

if (temp1!=null && temp2!=null)

{

if (temp1.data != temp2.data)

return false;

}

• Reply • Share ›

kumar ➚ Shravan • 2 months ago

if both references temp1 and temp2 points to the same object, then you can use temp1 == temp2. But since in above code both temp1 and temp2 points to different objects, we can't use == operator.

• Reply • Share ›

kumar ➚ Shravan • 2 months ago

in java you can't compare two class objects. They are different instances.
You have to use object1.data = object2.data.

• Reply • Share ›

Shravan ➚ kumar • 2 months ago

oh yeah, forgot, i am a C guy. So isn't the last check enough?

• Reply • Share ›

Vinoth • 2 months ago

can someone please explain why the space complexity is O(h1 + h2)? I understand as we use two stacks, it is h1 +h2. But how come it is O(h) for one stack?

• Reply • Share ›

kumar ➚ Vinoth • 2 months ago

You can try by one simple example for binary tree of height 3. The stack size will never increase the height of tree.

• Reply • Share ›

New_guy • 2 months ago

Can the algorithm be implemented with Queues also?

• Reply • Share ›

kumar ➚ New_guy • 2 months ago

if u use queue, you will be simply doing Level order traversal, and stack size will be O(m) and O(n), so space complexity= O(m+n) , same as simple method first described above.

• Reply • Share ›

Anupam • 2 months ago

How the memory space is O(log n) i.e height of the tree? Since at the worst case which will be when both trees are full binary tree, we will have n/2 nodes in the last level of each tree which will ultimately be pushed to their respective stacks. Space complexity should

be O(n/2 + m/2)

• Reply • Share ›

kumar ➚ Anupam • 2 months ago

In case of skew trees. the height of trees will be m and n.
So space complexity= O(height1+ height2)=O(m+n)

• Reply • Share ›

kumar ➚ Anupam • 2 months ago

In case of full binary tree also, the stack size will never exceed the Height of binary tree. It is just Inorder traversal of tree. And if u look closely in the algorithm we are not saving or keeping leaves in stack. we pop it each time after comparison with other tree's leaf.

• Reply • Share ›

hrs009 • 2 months ago

We can solve this by maintaining two Ordered dicts (in Python) or hash tables (C/C++) in the below fashion:

While traversing inorder for the two trees we need to keep the index of the leaf node i.e., number of nodes traversed so far. If it's a leaf node :
1. Then have a key in the dict with the leaf node.
2. Add the index of the leaf node as a value for the key.

So the dicts for the given problem would look like :

D1 = { 4 : 0, 6 : 3, 7 : 5}, D2 = { 4 : 1, 6 : 3, 7 : 5}

Since, we have taken Ordereddict, it will maintain the order in which the entry was added to the dict.

• Reply • Share ›

kumar ➚ hrs009 • 2 months ago

Yes, But when nodes in the trees are much much larger then you can't use hashing in C/C++. The solution in this article is time(O(n)) as well as memory optimised. (O(log (n)))

• Reply • Share ›