

Common Node in Two BST

Link: [Find Common Nodes in two BSTs | Practice | GeeksforGeeks](#)

MY CODE:

```
class Solution {  
public:  
    // Function to find the nodes that are common in both BST.  
    vector<int> findCommon(Node *r1, Node *r2) {  
        // Your code here  
        stack<Node*> s1;  
        stack<Node*> s2;  
  
        vector<int> ans;  
  
        if(!r1 || !r2)  
            return ans;  
  
        //Approach 1:  
        //create 2 inorder vector for bst1 & bst2  
        //Using 2 pointers, compare, and push common elements into the ans vector.  
  
        //Approach 2:  
        //root+all left elements into the stack. (iterative stack all left elements approach).  
  
        s1.push(r1);  
        s2.push(r2);  
  
        Node* root1 = r1;  
        Node* root2 = r2;
```

```
//All the left side, push into the s1

while(root1){

    s1.push(root1);

    root1 = root1->left;

}

//All the left side, push into the s2

while(root2){

    s2.push(root2);

    root2 = root2->left;

}

while(!s1.empty() && !s2.empty()){

    //s1 == s2 top elements equal

    if(s1.top()->data == s2.top()->data){

        ans.push_back(s1.top()->data);

        root1 = s1.top()->right;

        s1.pop();

        root2 = s2.top()->right;

        s2.pop();

    }

    //s1 > s2

    else if(s1.top()->data > s2.top()->data){

        root2 = s2.top()->right;

        s2.pop();

    }

}
```

```
//s2 > s1

else {
    root1 = s1.top()->right;
    s1.pop();
}

//All the left side, push into the s1
while(root1){
    s1.push(root1);
    root1 = root1->left;
}

//All the left side, push into the s2
while(root2){
    s2.push(root2);
    root2 = root2->left;
}

}
};
```

Error:

Wrong Answer. !!!

Ask Yogi Bot

Possibly your code does not work correctly for multiple test-cases (TCs).

The first test case where your code failed:

Test Cases Passed: 527 /1120

For Input :

```
81 9 265 4 67 214 298 N 6 42 73 88 257 283 313 N 7 11 56 71 75 83 144 224 262 271 297 308  
N N 8 N 23 46 62 70 N 74 77 N 85 141 177 219 246 259 264 269 281 295 N 301 311 N N 18  
41 43 55 60 64 N N N N...
```

Input is too large [Download Full File](#)

Your Code's output is:

```
7 11 12 15 16 17 20 22 23 25 26 29 30 31 32 37 42 43 50 51 52 56 63 64 67 71 74 75 83 85 86  
87 89 90 91 94 96 97 98 99 101 110 115 116 119 122 124 125 128 129 136 137 143 144 145  
149 151 154 156 160 1...
```

Your Output is too large [Download Full File](#)

It's Correct output is:

```
7 11 12 15 16 17 20 22 23 25 26 29 30 31 32 37 42 43 50 51 52 56 63 64 67 71 74 75 83 85 86  
87 89 90 91 94 96 97 98 99 101 110 115 116 119 122 124 125 128 129 136 137 143 144 145  
149 151 154 156 160 1...
```

Correct Output is too large [Download Full File](#)

Geek Tip:

- Solving the sample test case does not guarantee correctness of solution as your code is checked against multiple test cases.
- Avoid using static/global variables in your code.
- Do not print anything unless mentioned in the problem statement. Avoid unnecessary new line characters.

Your iterative inorder traversal code has **two critical bugs** causing it to fail on large test cases:

Bug 1: Wrong variable assignment when popping stacks

cpp

```
if(s1.top()->data == s2.top()->data){  
    ans.push_back(s1.top()->data);  
    root1 = s1.top()->right; //BUG: Should be s1.top()->right after pop  
    s1.pop();  
    root2 = s2.top()->right; //BUG: s2.top() is invalid after s2.pop()!  
    s2.pop();  
}
```

Fix:

cpp

```
if(s1.top()->data == s2.top()->data){  
    ans.push_back(s1.top()->data);  
    s1.pop();  
    root1 = s1.top() ? s1.top()->right : NULL; //Safe access  
    s2.pop();  
    root2 = s2.top() ? s2.top()->right : NULL;  
}
```

Bug 2: root1/root2 get corrupted and never reset

When you do `root1 = s1.top()->right`, you're overwriting the stack iterator variables with right child pointers. Then in the next push loop:

cpp

```
while(root1){ //root1 is now some random right child!
    s1.push(root1);
    root1 = root1->left;
}
```

This causes duplicate/incorrect nodes to be pushed, leading to wrong traversal order and infinite loops or stack overflow on large trees.

Corrected code:

cpp

```
vector<int> findCommon(Node *r1, Node *r2) {
    stack<Node*> s1, s2;
    vector<int> ans;

    if(!r1 || !r2) return ans;

    // Push all left path for both trees
    Node* curr1 = r1;
    while(curr1){
        s1.push(curr1);
        curr1 = curr1->left;
    }

    Node* curr2 = r2;
    while(curr2){
        s2.push(curr2);
        curr2 = curr2->left;
    }
```

```
while(!s1.empty() && !s2.empty()) {

    Node* top1 = s1.top();

    Node* top2 = s2.top();

    if(top1->data == top2->data) {

        ans.push_back(top1->data);

        s1.pop();

        s2.pop();

        // Push left children of right subtrees

        Node* next1 = top1->right;

        while(next1){

            s1.push(next1);

            next1 = next1->left;

        }

        Node* next2 = top2->right;

        while(next2){

            s2.push(next2);

            next2 = next2->left;

        }

    }

    else if(top1->data < top2->data) {

        s1.pop();

        Node* next1 = top1->right;

        while(next1){
```

```

    s1.push(next1);

    next1 = next1->left;

}

}

else { // top1->data > top2->data

    s2.pop();

    Node* next2 = top2->right;

    while(next2){

        s2.push(next2);

        next2 = next2->left;

    }

}

return ans;

}

```

Key fixes:

1. Store `top1 = s1.top()`, `top2 = s2.top()` **before** popping
2. Use separate `curr1`, `curr2`, `next1`, `next2` variables for iteration
3. Push left chain of right children **after** popping, not overwriting stack iterators

This correctly simulates merged inorder traversal of both BSTs.[wikipedia](#)

1. https://en.wikipedia.org/wiki/Binary_search_tree
2. <https://www.programiz.com/dsa/binary-search-tree>
3. <https://www.csd.uwo.ca/courses/CS1027b/notes/CS1027-015-BinarySearchTrees-W12.pdf>
4. <https://algs4.cs.princeton.edu/32bst/>
5. https://www.w3schools.com/dsa/dsa_data_binarysearchtrees.php
6. <https://www.cs.cmu.edu/afs/cs/academic/class/15210-f14/www/lectures/bst.pdf>

7. <https://www.geeksforgeeks.org/dsa/binary-search-tree-set-1-search-and-insertion/>
8. <https://www.geeksforgeeks.org/dsa/binary-search-tree-data-structure/>
9. <https://seneca-ictoer.github.io/data-structures-and-algorithms/J-Binary-Search-Trees/intro-to-bst>
10. https://www.dgp.toronto.edu/public_user/JamesStewart/378notes/15bst/

okay that worked well, now again please help me understand this using an example and in simpler terms, why this worked and my didn't?

what I did was before popping, directly storing it's right child and then after checking for all it's left children

and what you did is, storing the element, popping it, accessing it's right and then checking for it's left children?

how is it different? let me know using an example?

Perplexity ans:

The difference boils down to when and how you access the right child while doing the inorder traversal using a stack.

Your approach:

- You directly assign `root1 = s1.top()->right` **before** popping the node from the stack.
- Then you pop and proceed to push the left children of `root1`.
- But now `root1` is pointing to the right child of a node still on the stack.
- This leads to pushing nodes not in the correct inorder sequence and duplicates because the node itself is still on the stack.

Correct approach:

- You first **store** `top1 = s1.top()`.
- Then you **pop** the stack (removing the node).

- Then you access `top1->right` and push its left chain.
- This respects the inorder logic: visit left subtree, then node, then right subtree.

Example to clarify

Suppose a subtree rooted at node 10 with left child 5 and right child 15:

Stack before popping: [10]

- Your way: you peek 10, do `root1 = 10->right = 15` before pop
- Then pop 10 off stack
- Push left chain of root1 (which is 15 and its left children) → pushed 15 immediately, even though 10 wasn't popped before assigning right
- Next iteration you still have 10 presumably on stack or some confusion arises

Correct way:

- Peek 10 as `top1`
- Pop 10 off stack
- Then push left chain starting from `top1->right = 15`
- Now the stack correctly advances to right child 15 **after** completely processing node 10

This separation ensures the node itself is popped before we move to processing its right subtree, preserving correct inorder traversal semantics and preventing duplicate visits or wrong order.

In simpler terms:

- **You accessed right child while node was still "waiting" on stack.**
- **Correct approach removes the node first, then proceeds to right child.**

This order is critical to simulate recursion stack behavior exactly in iterative traversal.

You can visualize this process by stepping through each push/pop on small example trees to see the stack contents and traversal order.[enjoyalgorithms+1](#)

1. <https://www.enjoyalgorithms.com/blog/iterative-binary-tree-traversals-using-stack/>
2. <https://www.geeksforgeeks.org/dsa/binary-search-tree-traversal-inorder-preorder-post-order/>

3. <https://stackoverflow.com/questions/15375581/fixing-my-implementation-of-inorder-tree-traversal-algorithm-with-a-stack>
4. <https://stackoverflow.com/questions/13272527/implement-a-stack-using-a-bst>
5. <https://www.geeksforgeeks.org/dsa/inorder-traversal-of-binary-tree/>
6. https://www.w3schools.com/dsa/dsa_algo_binarytrees_inorder.php
7. <https://www.freecodecamp.org/news/binary-search-tree-traversal-inorder-preorder-post-order-for-bst/>
8. https://www.reddit.com/r/learnprogramming/comments/31iwn4/javahow_to_traverse_a_binary_search_tree_inorder/
9. <https://www.youtube.com/watch?v=lxTGsVXjwvM>
10. <https://www.youtube.com/watch?v=ZNYieAcf8eA>

but then why it worked for first 527 testcases and then after only it failed, if the approach had issues, why it worked for few?

I got the difference, I was accessing the right child even before popping, and then popping from the stack,

but then what difference does it make,

I was directly storing right child and popping element then after,

while you were storing element first then popping, and then accessing right child.

this I understand, but what is the need for this?

and what difference does it make? and why it failed after 527 testcases only.

Your approach worked for some test cases because in simpler or smaller BSTs, the order of operations (accessing right child before popping vs. popping then accessing) might not have caused inconsistent states or duplicates visibly due to lesser branching and stack size. The error manifests when the BST is large or more complex, exposing how accessing the right child before removing the current node from the stack disrupts the correct inorder sequence.

Why this difference matters — simplified understanding:

- **Stack's role:** It mimics the call stack in recursion, remembering nodes to process later after fully processing their left subtree.
- **Popping first:** When you **pop the node first**, you mark that the current node has been completely processed for its left subtree, and now you're moving to its right subtree.
- **Accessing right child before popping:** Means you treat the right child as if you had already finished processing the current node, but actually, you still have the current node in the stack. This breaks the controlled sequencing enforced by the stack, causing duplicate visits or wrong order pushing.
- **Early success in some cases:** In smaller or simpler trees, you might not notice because right children don't cause extra pushing or duplicates immediately, or the right subtree may be empty or simple enough. But on larger inputs, this misstep accumulates, leading to wrong traversal, infinite loops, or stack overload.

Example analogy:

Imagine you have a stack of books to read (nodes). Their left subtree is reading the intro chapters (pushing left children), the current book is "processing the node", and the right subtree is reading the closing chapters (pushing right children).

- Your approach tries **to read closing chapters (right child) before actually finishing the current book (popping the node)**, which confuses the reading order.
- Correct approach: **finish the current book (pop node) before starting the closing chapters (push right child)**, keeps reading order consistent.

This exact sequencing ensures the stack always represents the proper "call stack" state as if you were doing recursion, producing the correct final output.

So, your method works sometimes but fails on more complex cases due to this subtle but critical ordering problem.[enjoyalgorithms+2](#)

1. <https://www.enjoyalgorithms.com/blog/iterative-binary-tree-traversals-using-stack/>
2. https://www.reddit.com/r/algorithms/comments/9c6we2/is_a bst_inorder_iterative_walk_possible_without/
3. <https://www.enjoyalgorithms.com/blog/binary-tree-traversals-preorder-inorder-postorder/>
4. <https://www.geeksforgeeks.org/java/implementing-inorder-preorder-postorder-using-stack-in-java/>

5. <https://www.geeksforgeeks.org/dsa/binary-search-tree-traversal-inorder-preorder-post-order/>
6. <https://stackoverflow.com/questions/13272527/implement-a-stack-using-a-bst>
7. <https://stackoverflow.com/questions/15375581/fixing-my-implementation-of-inorder-tree-traversal-algorithm-with-a-stack>
8. <https://www.freecodecamp.org/news/binary-search-tree-traversal-inorder-preorder-post-order-for-bst/>
9. https://en.wikipedia.org/wiki/Tree_traversal
10. https://www.w3schools.com/dsa/dsa_algo_binarytrees_inorder.php

Minimal Changes Made

1. **Fixed dangling pointer:** Capture top1->right and top2->right **before** any pop() calls
2. **Set correct next pointers:** Assign next1 = top1->right and next2 = top2->right in **all** branches (equal, s1>s2, s2>s1)
3. **Added missing return ans** at the end
4. **Minor cleanup:** Commas in declarations, removed comments for brevity

Key insight: Always get right child **before** popping, and set nextX in every case so the correct subtree gets explored.