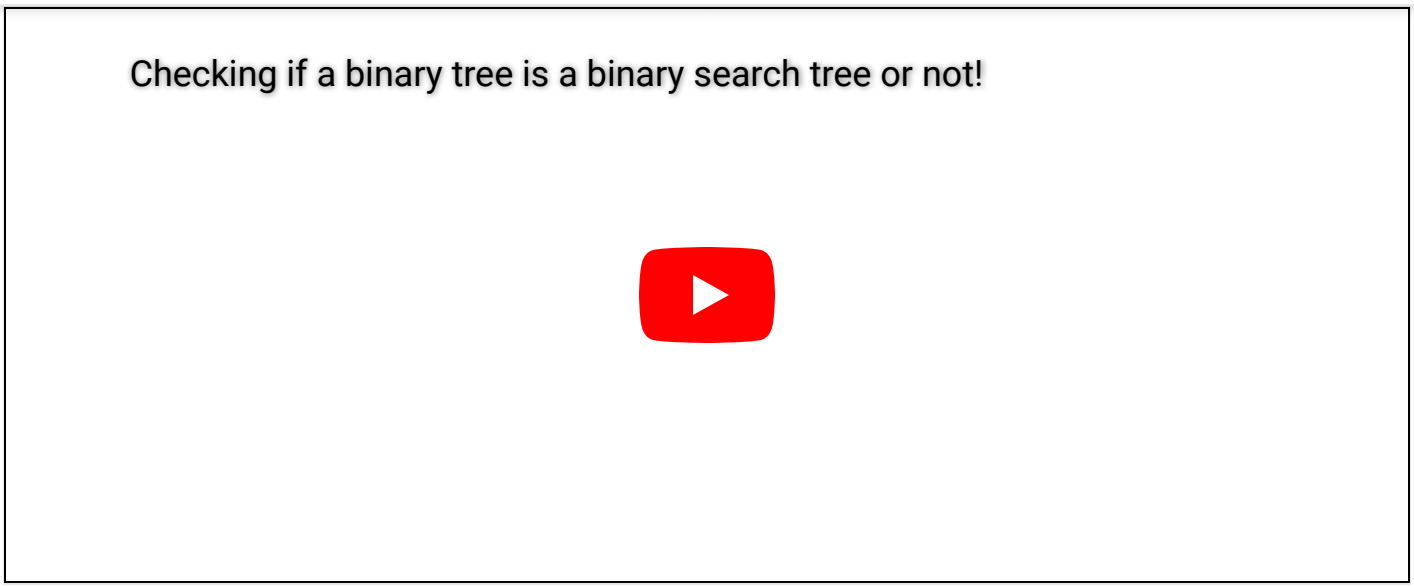


Checking if a binary tree is a binary search tree or not!

In the last lecture, we got to know about one more variety of binary tree, called binary search trees. We discussed all the properties of a binary search tree. We drew a fine line between binary trees and binary search trees. Today, we'll delve into their differences and distinguish a binary search tree from a normal binary tree. And check if a binary tree is a binary search tree or not.

But before that let's briefly revise the properties of a binary search tree once.

Properties of a binary search tree:

1. All nodes of the left subtree are lesser than the node itself.
2. All nodes of the right subtree are greater than the node itself.
3. Left and Right subtrees are also binary trees.
4. There are no duplicate nodes.
5. The InOrder traversal of a binary search tree gives an ascending sorted array.

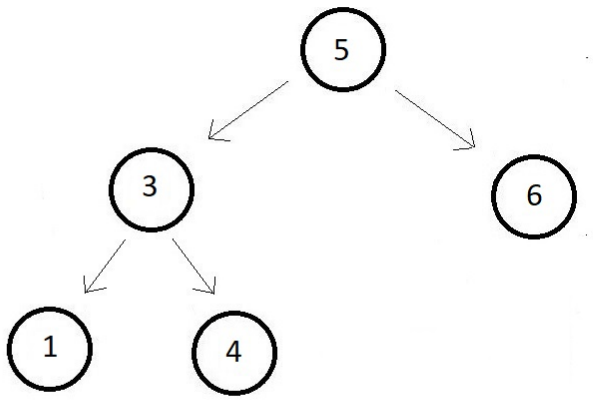
And the last one is of utmost importance to us. And that is what we use the most to check if a binary tree is a binary search tree or not. So, basically, we'll check if a

binary tree is a binary search tree or not by making an InOrder Traversal in the tree and analyzing its order. I have attached the source code below. Follow it as we proceed.

Understanding the code snippet below:

1. Since we'll create a binary search tree at first which is nothing but a binary tree only and we'll use the InOrder traversal function as well, we'll simply copy everything we did in our last programming tutorial where we learned to create the function inOrder. And we did this because we avoided doing repetitions in the course and starting from scratch, creating the whole struct Node and the createNode thing again and all the functions would have made the lecture redundant. And this has also saved us a lot of time.
2. I just hope that you've followed all the way from the start of the Binary Search lecture and not just jumped right here. Make sure you check them all before proceeding.

You should now be able to create Binary Tree yourselves. Create the binary search tree I've illustrated below using the createNode function. And if you are still not sure about creating a binary search tree, follow the previous lectures.



3. Let's now just create a function to check if the InOrder traversal of the binary tree is in ascending order or not.

Creating the function isBST:

4. Create an integer function *isBST* and pass the pointer to the root node of the tree you want to check as the only parameter. Inside the function, check if the pointer is not NULL, as we have been checking the whole time, and this is also

considered as the base case for the recursion to stop. If it is NULL, we would simply return 1 since an empty tree is always a binary search tree. Else, this is a complex yet understandable part. You should follow what I am saying.

5. Create a static struct Node pointer *prev* initialised with NULL. This maintains the pointer to the parent node. And since the root node doesn't have any parent, we have initialized it with NULL and made it static.
6. Now, see if the left subtree is a Binary Search Tree or not, by calling it recursively. If it is not a BST, return 0 here itself. Else, see if the *prev* is not NULL otherwise this is the root node of the whole tree and we won't check this condition. If the *prev* node is not NULL and the current node, which is the root node of the current subtree, is smaller than or equal to the *prev* node, then we would return 0. Since this violates the increasing orderliness.
7. If it still passes all the if conditions we have structured above, we will store the current node in the *prev* and move it recursively to the right subtree. And this is nothing but a modified version of the InOrder Traversal.

```
int isBST(struct  node* root){
    static struct node *prev = NULL;
    if(root!=NULL){
        if(!isBST(root->left)){
            return 0;
        }
        if(prev!=NULL && root->data <= prev->data){
            return 0;
        }
        prev = root;
        return isBST(root->right);
    }
    else{
        return 1;
    }
}
```

Code Snippet 1: Creating the isBST function

I suggest ignoring the recursion and do not trace how the function isBST works if the function isBST is at all confusing. Just note that we have done nothing but the InOrder traversal of the tree, and we have checked if the previous value is smaller than the current value, that's it.

Note: Static variables are used when we don't want our value for that variable to change every time that function is called.

Here is the whole source code:

```
#include<stdio.h>
#include<malloc.h>

struct node{
    int data;
    struct node* left;
    struct node* right;
};

struct node* createNode(int data){
    struct node *n; // creating a node pointer
    n = (struct node *) malloc(sizeof(struct node)); // Allocating m
    n->data = data; // Setting the data
    n->left = NULL; // Setting the left and right children to NULL
    n->right = NULL; // Setting the left and right children to NULL
    return n; // Finally returning the created node
}

void preOrder(struct  node* root){
    if(root!=NULL){
        printf("%d ", root->data);
        preOrder(root->left);
        preOrder(root->right);
    }
}
```

```
void postOrder(struct  node* root){
    if(root!=NULL){
        postOrder(root->left);
        postOrder(root->right);
        printf("%d ", root->data);
    }
}

void inOrder(struct  node* root){
    if(root!=NULL){
        inOrder(root->left);
        printf("%d ", root->data);
        inOrder(root->right);
    }
}

int isBST(struct  node* root){
    static struct node *prev = NULL;
    if(root!=NULL){
        if(!isBST(root->left)){
            return 0;
        }
        if(prev!=NULL && root->data <= prev->data){
            return 0;
        }
        prev = root;
        return isBST(root->right);
    }
    else{
        return 1;
    }
}

int main(){
```

```
    // Constructing the root node - Using Function (Recommended)
    struct node *p = createNode(5);
    struct node *p1 = createNode(3);
    struct node *p2 = createNode(6);
    struct node *p3 = createNode(1);
    struct node *p4 = createNode(4);
    // Finally The tree looks like this:
    //      5
    //     / \
    //    3   6
    //   / \
    //  1   4

    // Linking the root node with left and right children
    p->left = p1;
    p->right = p2;
    p1->left = p3;
    p1->right = p4;

    // preOrder(p);
    // printf("\n");
    // postOrder(p);
    // printf("\n");
    inOrder(p);
    printf("\n");
    // printf("%d", isBST(p));
    if(isBST(p)){
        printf("This is a bst" );
    }
    else{
        printf("This is not a bst");
    }
    return 0;
}
```

Code Snippet 2: Implementing the isBST function

Let's now check if our function actually works, and it reverts whether our binary tree is a BST or not. We would also like to print the InOrder traversal of the tree.

```
 inOrder(p);
printf("\n");
if(isBST(p)){
    printf("This is a bst" );
}
else{
    printf("This is not a bst");
}
```

Code Snippet 3: Using the isBST function

And the output we received was:

```
1 3 4 5 6
This is a bst
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Figure 1: Output of the above code

And yes, it says, our function is a Binary Search Tree and that was what we expected. The InOrder traversal is also in ascending order. So, things are pretty accurate and consistent.

So, this was all we had to do to automate the checking of a binary tree for a binary search tree. Go through the functions again, and keep in mind the things I've told you not to worry about. Focus on the algorithm, and the modifications we made in the InOrder function. And that would be all. It is now time to move on to our next topic which entails our first operation on Binary Search Trees.

I appreciate your support throughout. I hope you enjoyed the tutorial. If you genuinely appreciate my work, please let your friends know about this course too. If you haven't checked out the whole playlist yet, move on to codewithharry.com or

my YouTube channel to access it. See you all in the next tutorial where we'll learn about the most important operation we use binary search trees for, the **search operation.** Till then keep coding.

Previous Next

CodeWithHarry