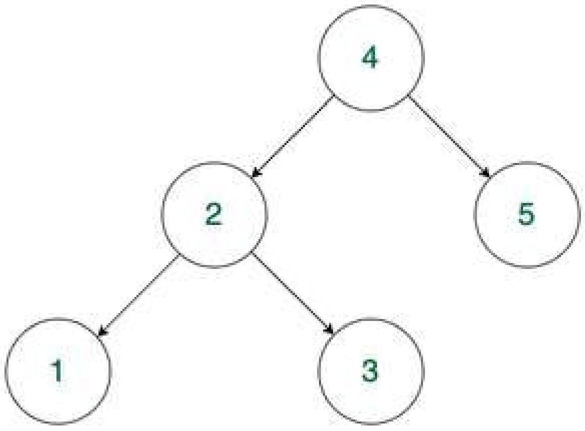
A program to check if a Binary Tree is BST or not

A Binary Search Tree (BST) is a node-based binary tree data structure that has the following properties.

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.
- Each node (item in the tree) has a distinct key.



Binary Search Tree

Naive Approach:

*The idea is to for each node, check if **max** value in **left subtree** is smaller than the node and **min** value in **right subtree** greater than the node.*

Follow the below steps to solve the problem:

- If the current node is **null** then return **true**
- If the value of the left child of the node is greater than or equal to the current node then return **false**
- If the value of the right child of the node is less than or equal to the current node then return **false**
- If the left subtree or the right subtree is not a BST then return **false**
- Else return **true**

Below is the implementation of the above approach:

C++

```
#include <bits/stdc++.h>

using namespace std;


/* A binary tree node has data, pointer to left child

and a pointer to right child */

struct node {

    int data;

    struct node* left;

    struct node* right;

};


/* Helper function that allocates a new node with the

given data and NULL left and right pointers. */

struct node* newNode(int data)

{

    struct node* node

        = (struct node*)malloc(sizeof(struct node));

    node->data = data;

    node->left = NULL;

    node->right = NULL;


    return (node);

}


int maxValue(struct node* node)

{

    if (node == NULL) {
        return INT16_MIN;
    }
    int value = node->data;
    int leftMax = maxValue(node->left);
    int rightMax = maxValue(node->right);

    return max(value, max(leftMax, rightMax));
}

int minValue(struct node* node)
{
    if (node == NULL) {
        return INT16_MAX;
    }
    int value = node->data;
    int leftMax = minValue(node->left);
    int rightMax = minValue(node->right);

    return min(value, min(leftMax, rightMax));
}

/* Returns true if a binary tree is a binary search tree */
int isBST(struct node* node)
{
    if (node == NULL)
        return 1;
    /* false if the max of the left is > than us */

    if (node->left != NULL

        && maxValue(node->left) >= node->data)

        return 0;


    /* false if the min of the right is <= than us */

    if (node->right != NULL

        && minValue(node->right) <= node->data)

        return 0;


    /* false if, recursively, the left or right is not a BST

     */

    if (!isBST(node->left) || !isBST(node->right))

        return 0;


    /* passing all that, it's a BST */

    return 1;

}


/* Driver code*/

int main()

{

    struct node* root = newNode(4);

    root->left = newNode(2);

    root->right = newNode(5);

    // root->right->left = newNode(7);

    root->left->left = newNode(1);
    root->left->right = newNode(3);


    // Function call
    if (isBST(root))
        printf("Is BST");
    else
        printf("Not a BST");


    return 0;
}
```

Output

```
Is BST
```

Note: It is assumed that you have helper functions minValue() and maxValue() that return the min or max int value from a non-empty tree

Time Complexity: O(N^2), As we visit every node just once and our helper method also takes O(N) time, so overall time complexity becomes O(N) * O(N) = O(N^2)

Auxiliary Space: O(H), Where H is the height of the binary tree, and the extra space is used due to the function call stack.

Check BST using specified range of minimum and maximum values of nodes:

*The **isBSTUtil()** function is a recursive helper function that checks whether a subtree (rooted at a given node) is a BST within the specified range of minimum (**min**) and maximum (**max**) values. If any node violates this range, the function returns **false**; otherwise, it continues checking the left and right subtrees.*

Below is the implementation of the above approach:

C++

```
#include <bits/stdc++.h>
using namespace std;
```

```
/* A binary tree node has data,

pointer to left child and

a pointer to right child */

class node {

public:

    int data;

    node* left;

    node* right;



    /* Constructor that allocates

    a new node with the given data

    and NULL left and right pointers. */

    node(int data)

    {

        this->data = data;

        this->left = NULL;

        this->right = NULL;

    }

};



int isBSTUtil(node* node, int min, int max);



/* Returns true if the given

tree is a binary search tree

(efficient version). */

int isBST(node* node)
```

```
{
    return (isBSTUtil(node, INT_MIN, INT_MAX));
}


/* Returns true if the given
tree is a BST and its values
are >= min and <= max. */
int isBSTUtil(node* node, int min, int max)
{
    /* an empty tree is BST */
    if (node == NULL)
        return 1;

    /* false if this node violates
    the min/max constraint */
    if (node->data < min || node->data > max)
        return 0;

    /* otherwise check the subtrees recursively,
    tightening the min or max constraint */
    return isBSTUtil(node->left, min, node->data - 1)

           && // Allow only distinct values
           isBSTUtil(node->right, node->data + 1,
                     max); // Allow only distinct values
}
```

```
/* Driver code*/
int main()
{
    node* root = new node(4);
    root->left = new node(2);
    root->right = new node(5);
    root->left->left = new node(1);
    root->left->right = new node(3);

      // Function call
    if (isBST(root))
        cout << "Is BST";
    else
        cout << "Not a BST";

    return 0;
}
```

Output

```
Is BST
```

Time Complexity: O(N), Where N is the number of nodes in the tree
Auxiliary Space: O(1), if Function Call Stack size is not considered, otherwise O(H) where H is the height of the tree
----

Check BST using Inorder Traversal:
*The idea is to use **Inorder traversal** of a binary search tree generates output, sorted in ascending order. So generate inorder traversal of the given binary tree and check if the values are sorted or not*
Follow the below steps to solve the problem:

- Do In-Order Traversal of the given tree and store the result in a **temp** array.
- Check if the **temp** array is sorted in ascending order, if it is, then the tree is BST.

Note: We can avoid the use of an Auxiliary Array. While doing In-Order traversal, we can keep track of previously visited nodes. If the value of the currently visited node is less than the previous value, then the tree is not BST.
Below is the implementation of the above approach:
C++

```
// C++ program to check if a given tree is BST.
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to
left child and a pointer to right child */
struct Node {
    int data;
    struct Node *left, *right;

    Node(int data)
    {
        this->data = data;
        left = right = NULL;
    }
};

bool isBSTUtil(struct Node* root, Node*& prev)
{
    // traverse the tree in inorder fashion and
    // keep track of prev node
    if (root) {
        if (!isBSTUtil(root->left, prev))
            return false;

        // Allows only distinct valued nodes
        if (prev != NULL && root->data <= prev->data)
            return false;

        prev = root;

        return isBSTUtil(root->right, prev);
    }

    return true;
}

bool isBST(Node* root)
{
    Node* prev = NULL;
    return isBSTUtil(root, prev);
}

/* Driver code*/
int main()
{
    struct Node* root = new Node(3);
    root->left = new Node(2);
    root->right = new Node(5);
    root->left->left = new Node(1);
    root->left->right = new Node(4);

    // Function call
    if (isBST(root))
        cout << "Is BST";
    else
        cout << "Not a BST";

    return 0;
}
```

Output

```
Not a BST
```

Time Complexity: O(N), Where N is the number of nodes in the tree
Auxiliary Space: O(H), Here H is the height of the tree and the extra space is used due to the function call stack.

Check BST using Morris Traversal:

Follow the steps to implement the approach:

- While the current node is not null, do the following:
    - If the current node does not have a left child, then process the node and move to its right child.
    - If the current node has a left child, then find its inorder predecessor (i.e., the rightmost node in its left subtree) and check if it is less than the current node's value.
        - if the predecessor's right child is null, then set it to point to the current node and move to the current node's left child.

            - If the predecessor’s right child is already pointing to the current node, then reset it to null (to restore the original binary tree structure), process the current node, and move to its right child.
        - Repeat steps b and c until the current node is null.
    - If the traversal completes without finding any violation of the BST property, then the binary tree is a valid BST

Below is the implementation of the above approach:

C++

```
// C++ code to implement the morris traversal approach
#include <iostream>
using namespace std;

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

bool isValidBST(TreeNode* root) {
    TreeNode* curr = root;
    TreeNode* prev = NULL;

    while (curr != NULL) {
        if (curr->left == NULL) { // case 1: no left child
            // process the current node
            if (prev != NULL && prev->val >= curr->val)
```

```
                return false;
            prev = curr;
            curr = curr->right;
        }
        else { // case 2: has a left child
            // find the inorder predecessor
            TreeNode* pred = curr->left;
            while (pred->right != NULL && pred->right != curr)
                pred = pred->right;

            if (pred->right == NULL) { // make threaded link
                pred->right = curr;
                curr = curr->left;
            }
            else { // remove threaded link
                pred->right = NULL;
                // process the current node
                if (prev != NULL && prev->val >= curr->val)
                    return false;
                prev = curr;
                curr = curr->right;
            }
        }
    }

    return true; // binary tree is a valid BST
}
```

```cpp
// Driver Code
int main() {
    // create the binary tree from the example
    TreeNode* root = new TreeNode(4);
    root->left = new TreeNode(2);
    root->right = new TreeNode(5);
    root->left->left = new TreeNode(1);
    root->left->right = new TreeNode(3);

    // check if the binary tree is a valid BST
    if (isValidBST(root))
        cout << "The binary tree is a valid BST." << endl;
    else
        cout << "The binary tree is not a valid BST." << endl;

    return 0;
}
```

Output

```
The binary tree is a valid BST.
```

Time Complexity: O(N), Where N is the number of nodes in the tree.
Auxiliary Space: O(1) , Because we are not using any addition data structure or recursive call stack.