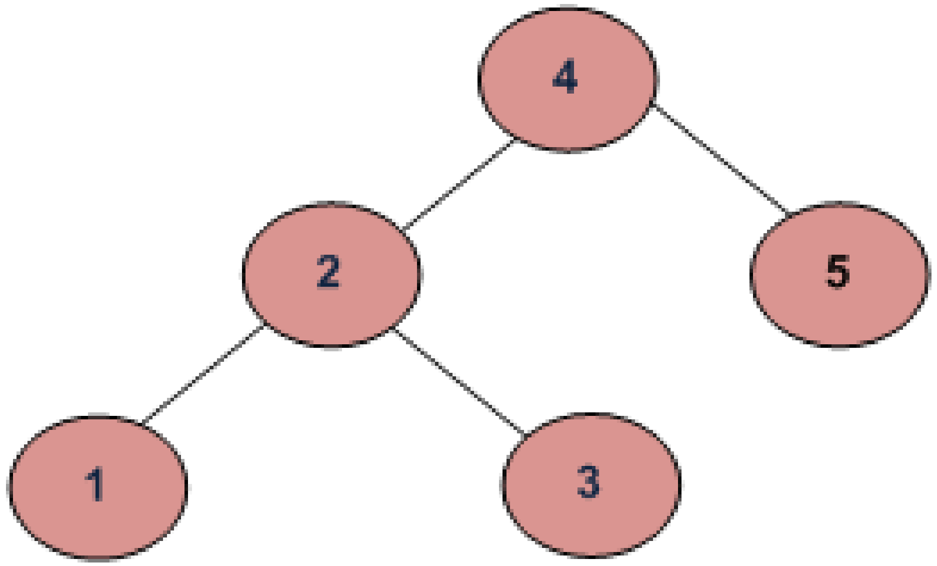
Binary Search Tree - Applications

SWE2016-44

Check if a binary tree is BST or not



Check if a binary tree is BST or not

Properties of BST

1) The left subtree of a node contains nodes with keys less than the node’s key and the right subtree with keys greater than node’s key.

2) The left and right subtree each must also be a binary search tree and there must be no duplicate nodes.

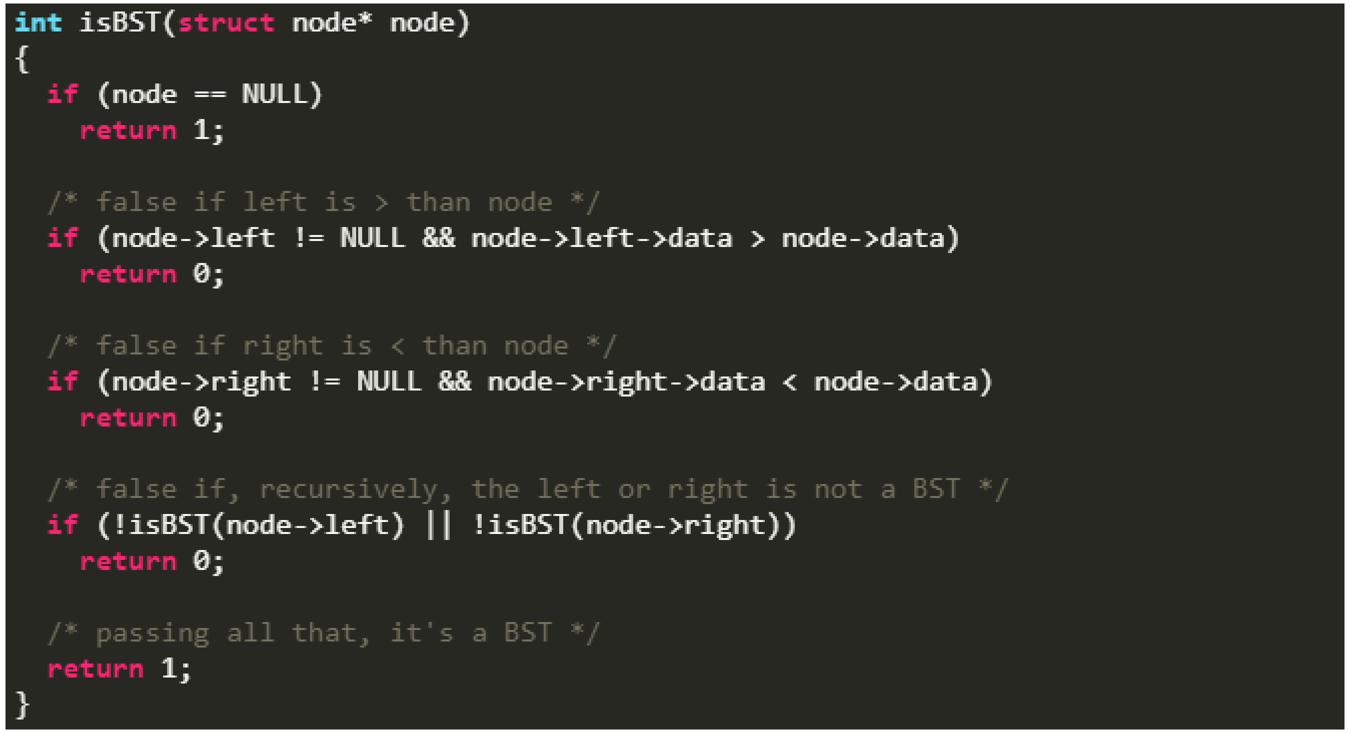
Check if a binary tree is BST or not

Method 1: Simple but Wrong

For each node, check if the left node of it is smaller than the node and right node of it is greater than the node.

Check if a binary tree is BST or not

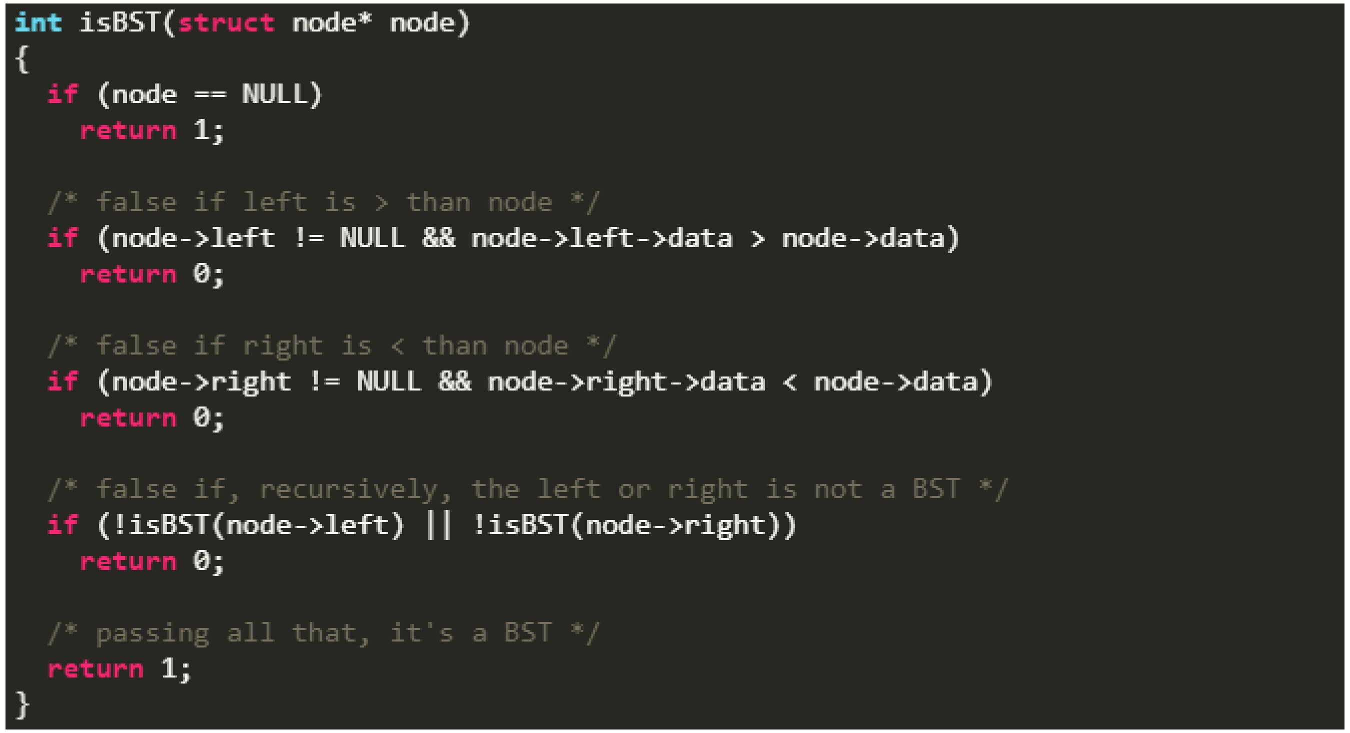
Method 1: Simple but Wrong



```
int isBST(struct node* node)
{
  if (node == NULL)
    return 1;

  /* false if left is > than node */
  if (node->left != NULL && node->left->data > node->data)
    return 0;

  /* false if right is < than node */
  if (node->right != NULL && node->right->data < 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;
}
```

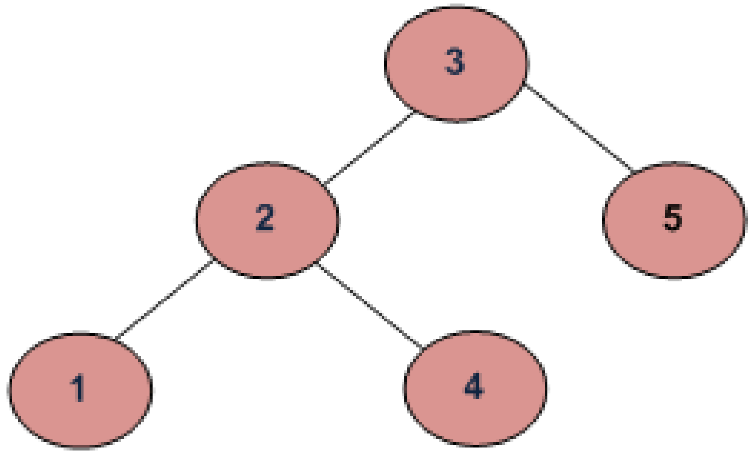
Check if a binary tree is BST or not

Method 1: Simple but Wrong



```
int isBST(struct node* node)
{
  if (node == NULL)
    return 1;

  /* false if left is > than node */
  if (node->left != NULL && node->left->data > node->data)
    return 0;

  /* false if right is < than node */
  if (node->right != NULL && node->right->data < 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;
}
```

→This approach is wrong as this will return true for below binary tree



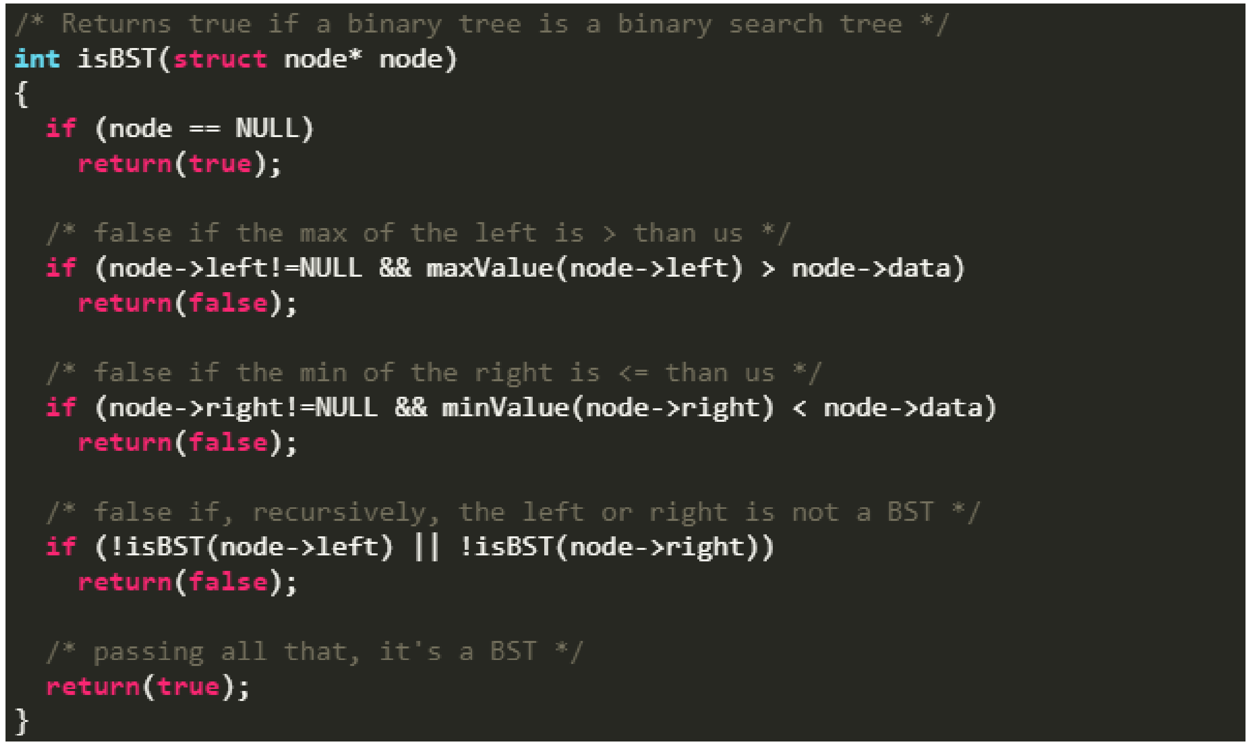
Check if a binary tree is BST or not

Method 2: Correct but not efficient

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.

Check if a binary tree is BST or not

Method 2: Correct but not efficient

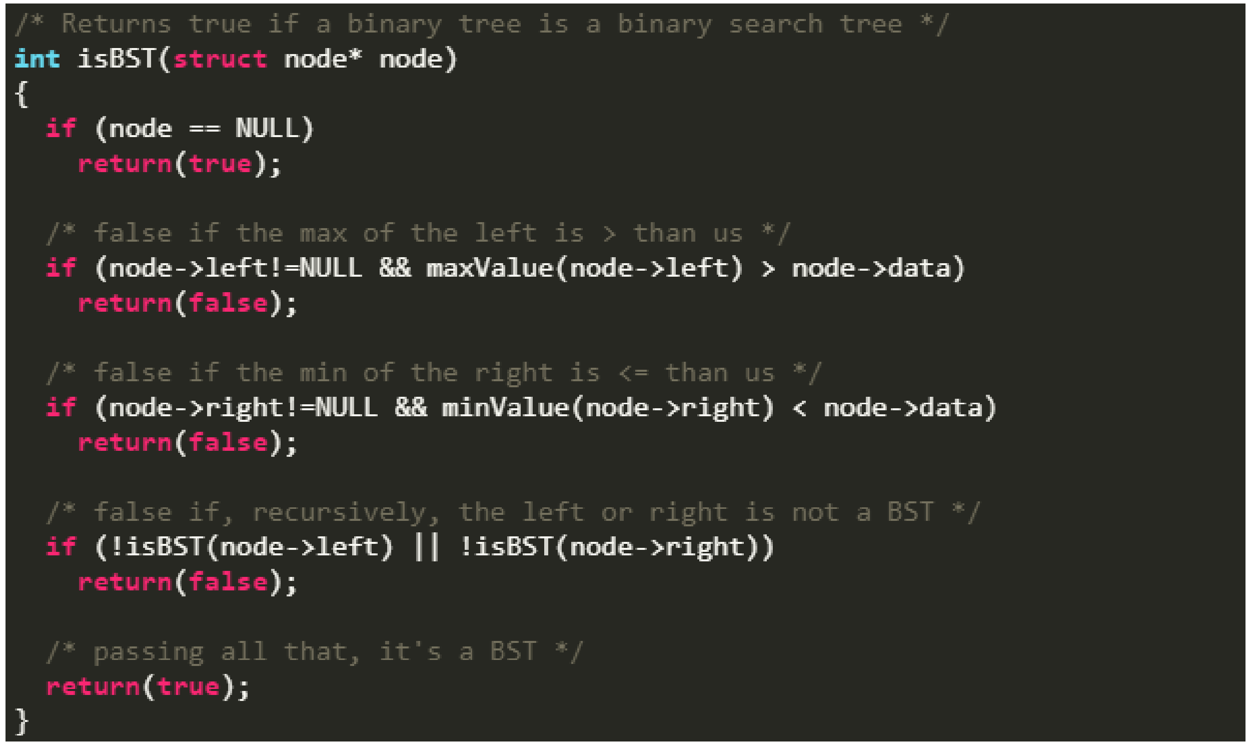


```c
/* Returns true if a binary tree is a binary search tree */
int isBST(struct node* node)
{
  if (node == NULL)
    return(true);

  /* false if the max of the left is > than us */
  if (node->left!=NULL && maxValue(node->left) > node->data)
    return(false);

  /* false if the min of the right is <= than us */
  if (node->right!=NULL && minValue(node->right) < node->data)
    return(false);

  /* false if, recursively, the left or right is not a BST */
  if (!isBST(node->left) || !isBST(node->right))
    return(false);

  /* passing all that, it's a BST */
  return(true);
}
```

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

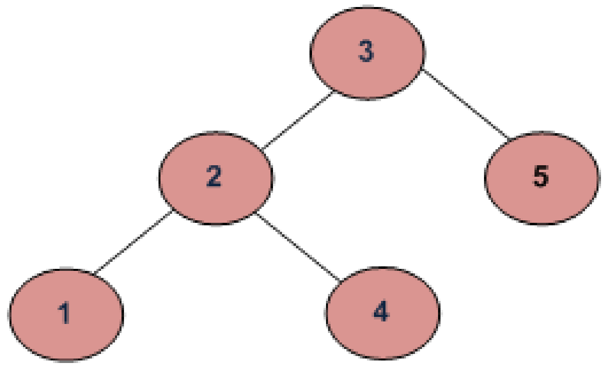
Check if a binary tree is BST or not

Method 2: Correct but not efficient



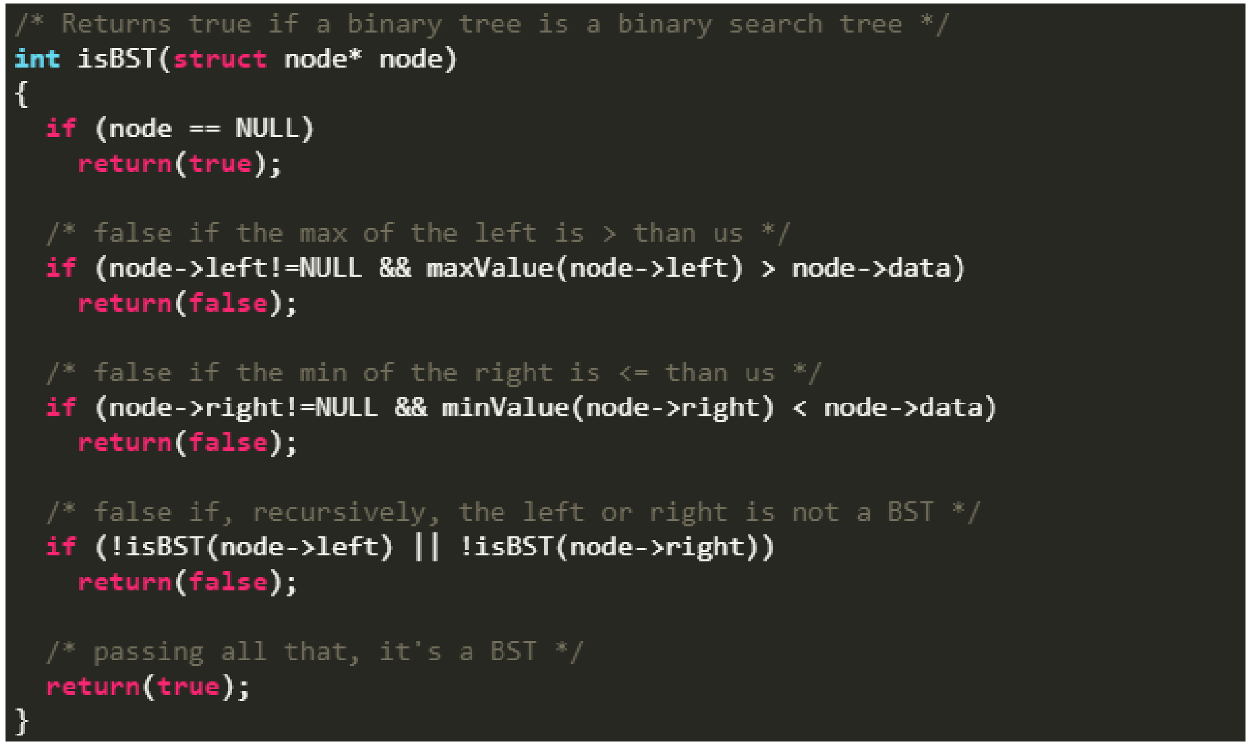
```
/* Returns true if a binary tree is a binary search tree */
int isBST(struct node* node)
{
  if (node == NULL)
    return(true);

  /* false if the max of the left is > than us */
  if (node->left!=NULL && maxValue(node->left) > node->data)
    return(false);

  /* false if the min of the right is <= than us */
  if (node->right!=NULL && minValue(node->right) < node->data)
    return(false);

  /* false if, recursively, the left or right is not a BST */
  if (!isBST(node->left) || !isBST(node->right))
    return(false);

  /* passing all that, it's a BST */
  return(true);
}
```

→ **Avoid the following:**



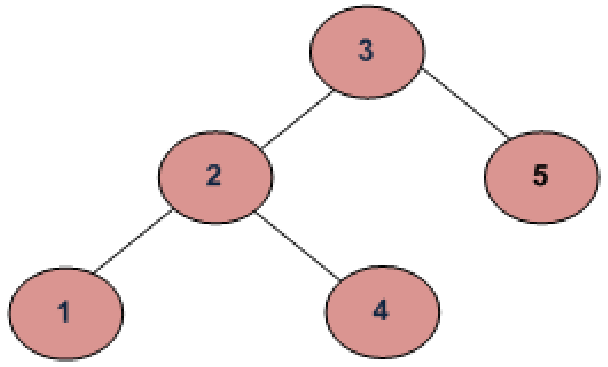
Check if a binary tree is BST or not

Method 2: Correct but not efficient



```
/* Returns true if a binary tree is a binary search tree */
int isBST(struct node* node)
{
  if (node == NULL)
    return(true);

  /* false if the max of the left is > than us */
  if (node->left!=NULL && maxValue(node->left) > node->data)
    return(false);

  /* false if the min of the right is <= than us */
  if (node->right!=NULL && minValue(node->right) < node->data)
    return(false);

  /* false if, recursively, the left or right is not a BST */
  if (!isBST(node->left) || !isBST(node->right))
    return(false);

  /* passing all that, it's a BST */
  return(true);
}
```

→ **Avoid the following:**



→ **It traverses over some parts of the tree many times**

→ **Time Complexity: O(n*n)**

Check if a binary tree is BST or not

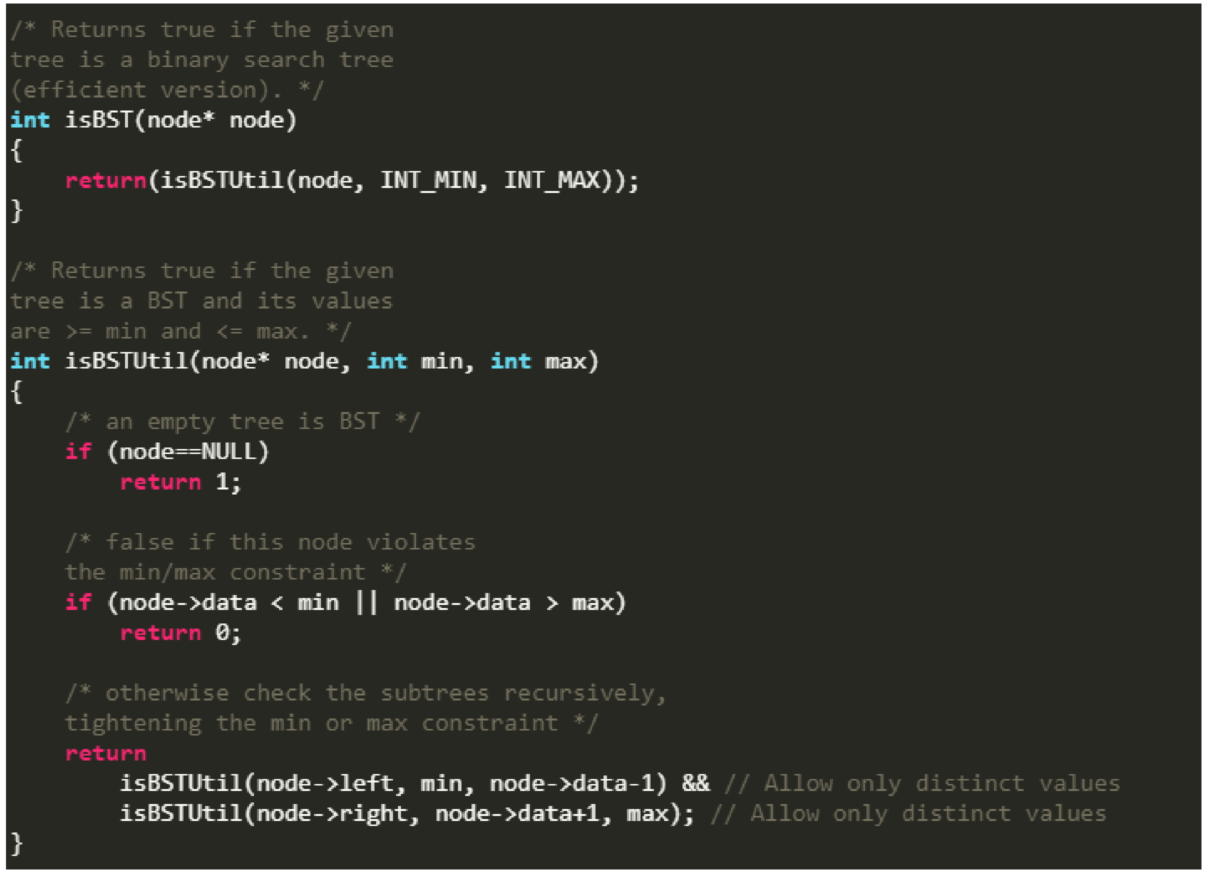
Method 3: Correct but Efficient

Write a utility helper function isBSTUtil(struct node* node, int min, int max) that traverses down the tree keeping track of the narrowing min and max allowed values as it goes, looking at each node only once.

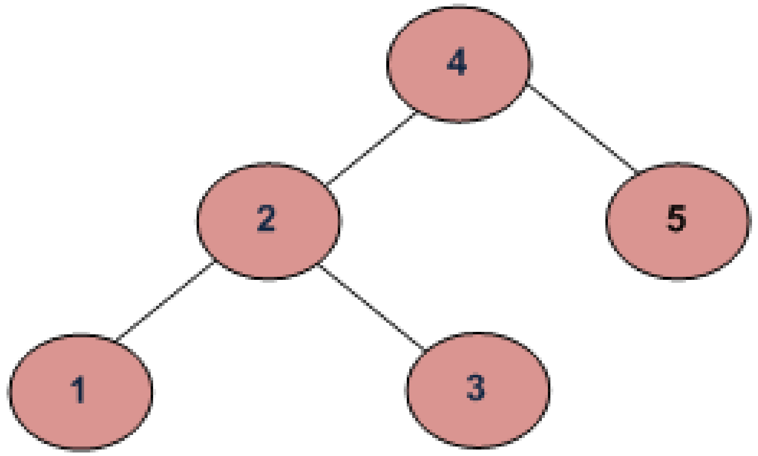
The initial values for min and max should be INT_MIN and INT_MAX — they narrow from there.

Check if a binary tree is BST or not

Method 3: Correct but Efficient

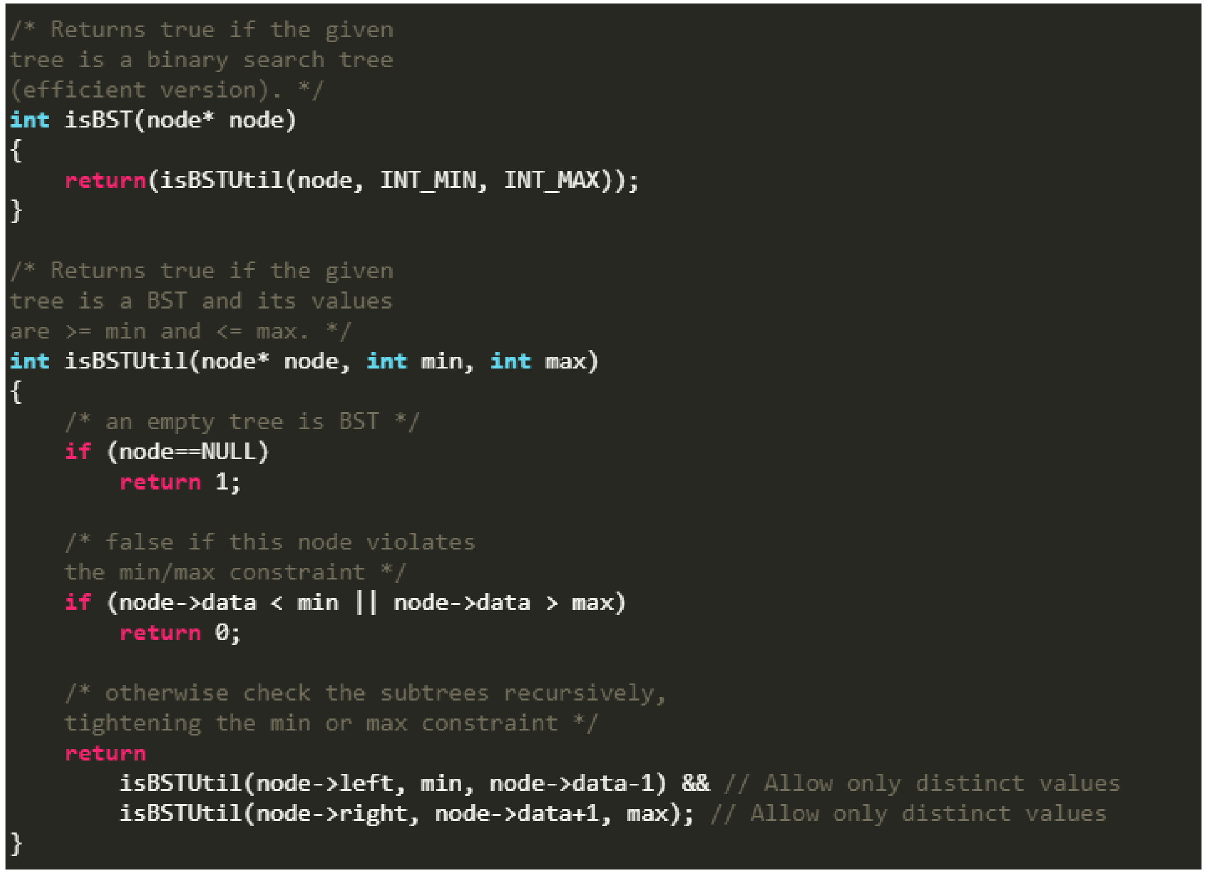


```
/* Returns true if the given
tree is a binary search tree
(efficient version). */
int isBST(node* node)
{
    return(isBSTUtil(node, INT_MIN, INT_MAX));
}

/* 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
}
```



Check if a binary tree is BST or not

Method 3: Correct but Efficient



```
/* 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
}
```

- Time Complexity: O(n)
- Auxiliary Space : O(1) if Function Call Stack size is not considered, otherwise O(n)

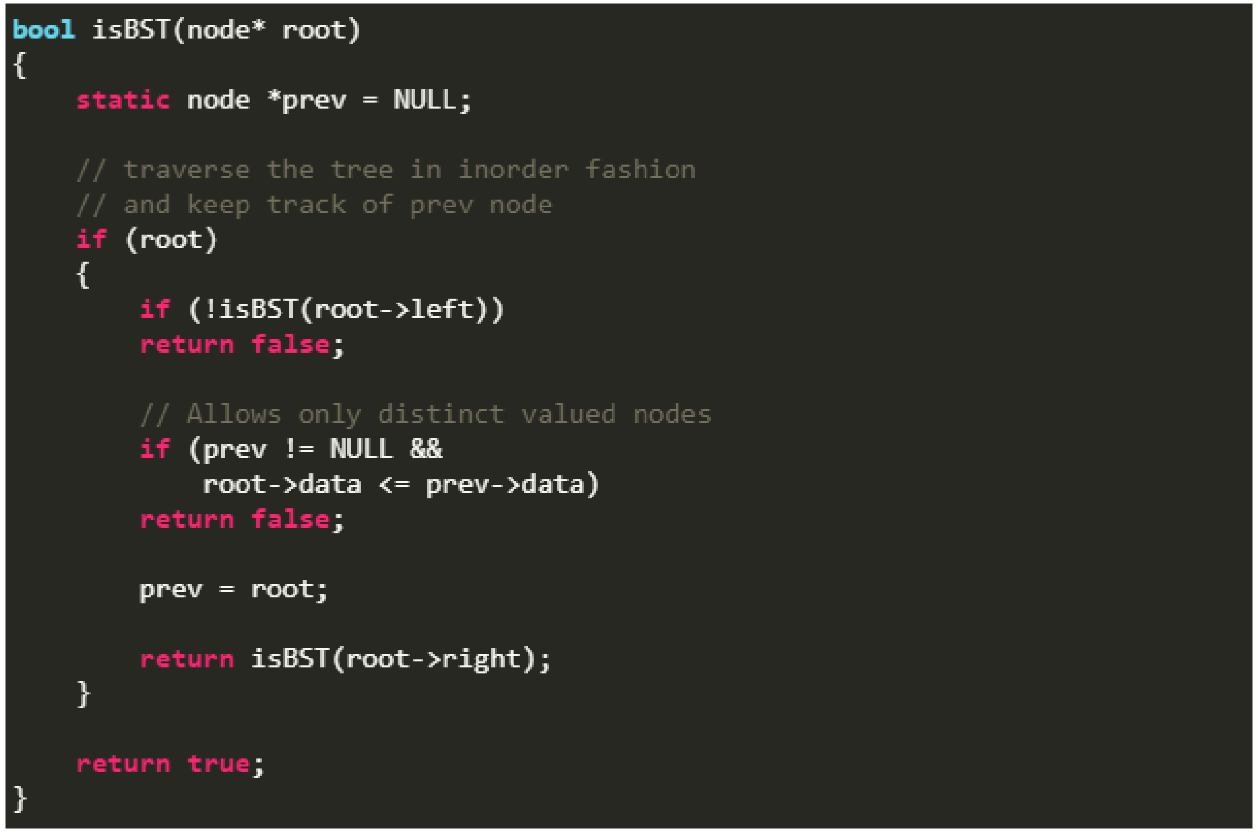
Check if a binary tree is BST or not

Method 4: Using Inorder Traversal

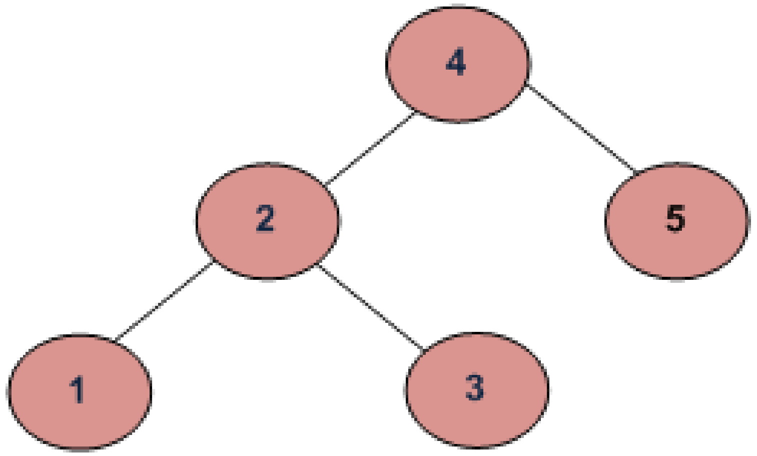
1) **Do In-Order Traversal of the given tree and store the result in a temp array.**
2) **Check if the temp array is sorted in ascending order, if it is, then the tree is BST.**

Check if a binary tree is BST or not

Method 4: Using Inorder Traversal

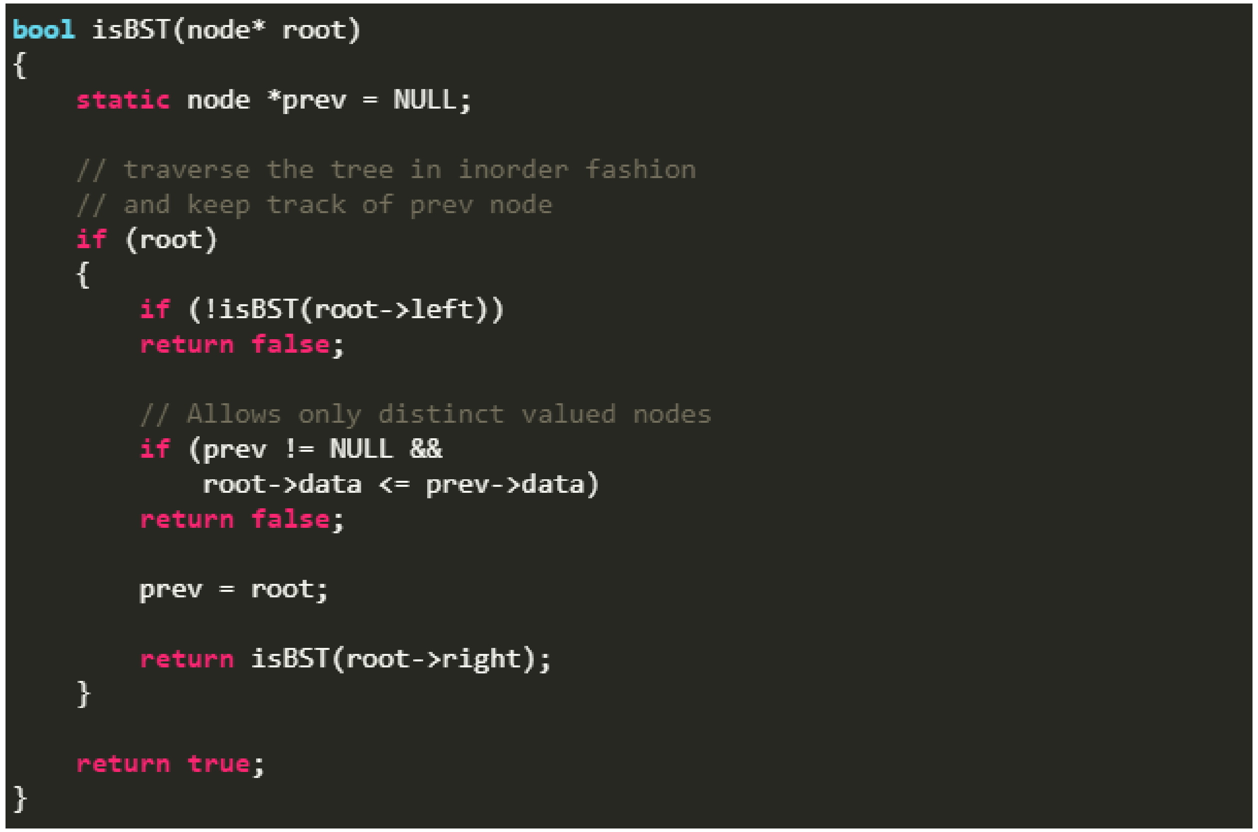


```
bool isBST(node* root)
{
    static node *prev = NULL;

    // traverse the tree in inorder fashion
    // and keep track of prev node
    if (root)
    {
        if (!isBST(root->left))
        return false;

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

        prev = root;

        return isBST(root->right);
    }

    return true;
}
```



Check if a binary tree is BST or not

Method 4: Using Inorder Traversal



```
bool isBST(node* root)
{
    static node *prev = NULL;

    // traverse the tree in inorder fashion
    // and keep track of prev node
    if (root)
    {
        if (!isBST(root->left))
        return false;

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

        prev = root;

        return isBST(root->right);
    }

    return true;
}
```

- **Time Complexity: O(n)**
- **Auxiliary Space : O(n)**

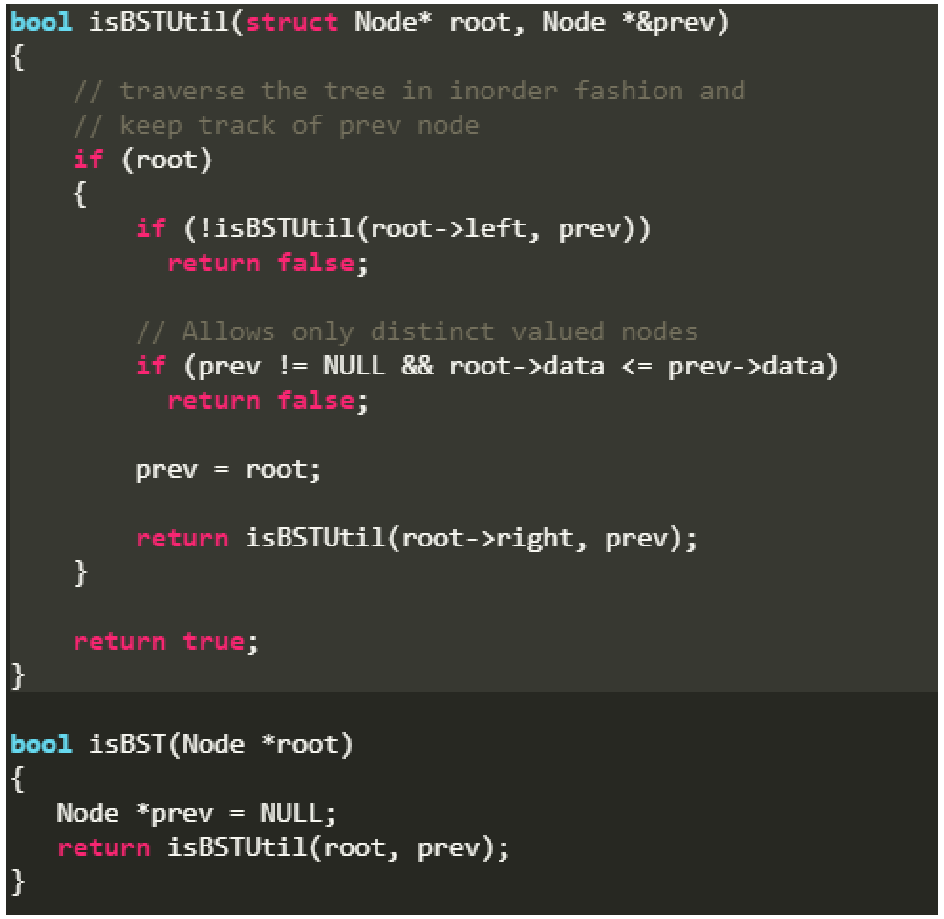
Check if a binary tree is BST or not

Method 4: Using Inorder Traversal (Modified)

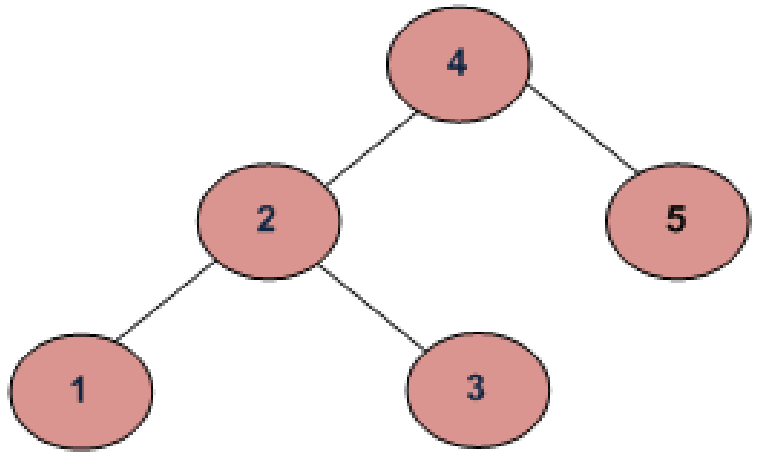
We can avoid the use of Auxiliary Array. While doing In-Order traversal, we can keep track of previously visited node. If the value of the currently visited node is less than the previous value, then tree is not BST.

Check if a binary tree is BST or not

Method 4: Using Inorder Traversal (Modified)

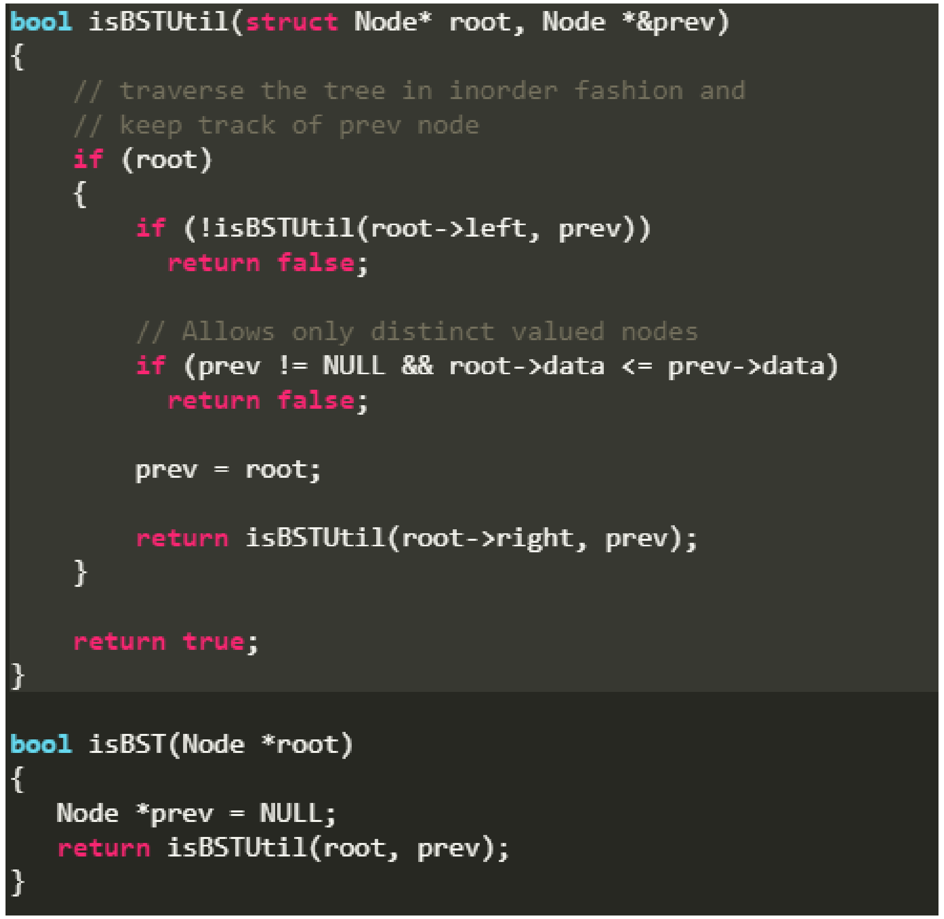


```
bool isBSTUtil(struct Node* root, Node *&prev)
{
    // traverse the tree in inorder fashion and
    // keep track of prev node
    if (root)
    {
        if (!isBSTUtil(root->left, prev))
          return false;

        // 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);
}
```



Check if a binary tree is BST or not

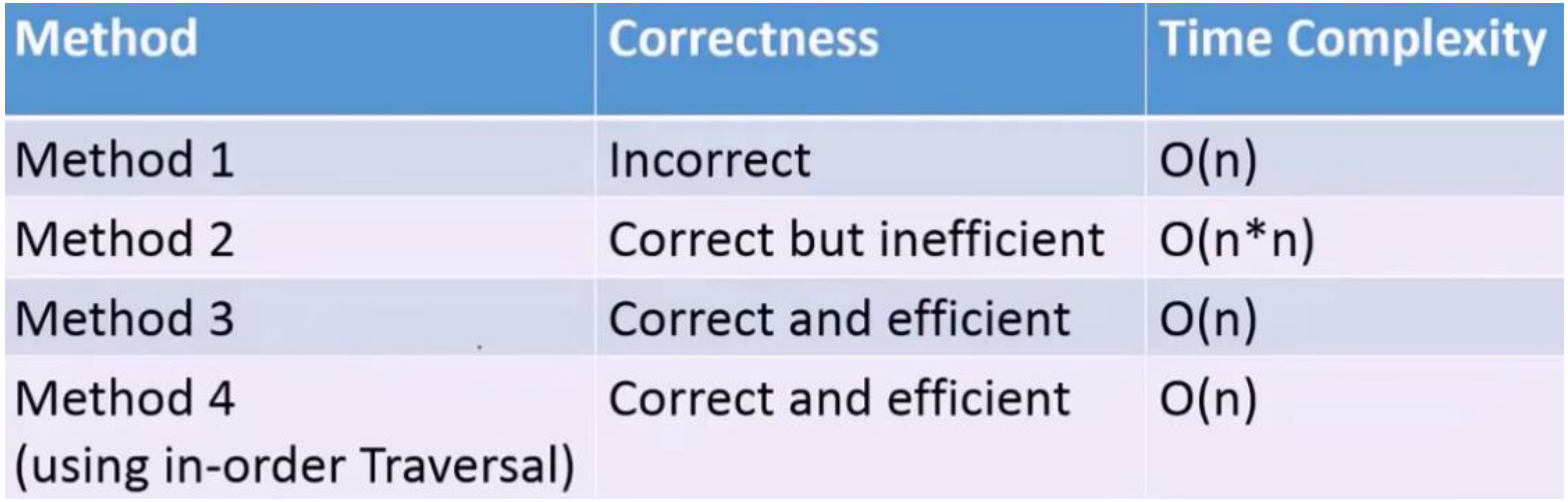
Method 4: Using Inorder Traversal (Modified)



```
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);
}
```

- **Time Complexity: O(n)**
- **Auxiliary Space : O(1)**

Check if a binary tree is BST or not



Method	Correctness	Time Complexity
Method 1	Incorrect	O(n)
Method 2	Correct but inefficient	O(n*n)
Method 3	Correct and efficient	O(n)
Method 4 (using in-order Traversal)	Correct and efficient	O(n)

Sorted Array to Balanced BST

Given a sorted array. Write a function that creates a Balanced Binary Search Tree using array elements.

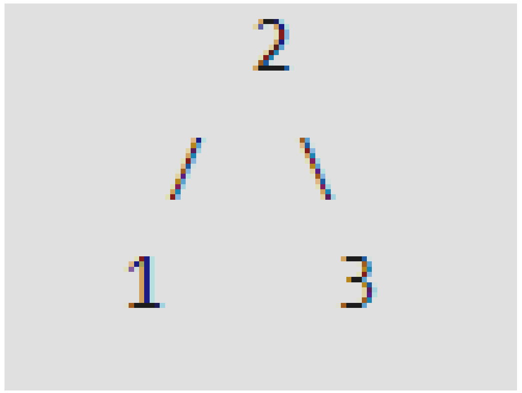
Sorted Array to Balanced BST

Given a sorted array. Write a function that creates a Balanced Binary Search Tree using array elements.

Example 1:

Input array: {1, 2, 3}

Output Balanced BST



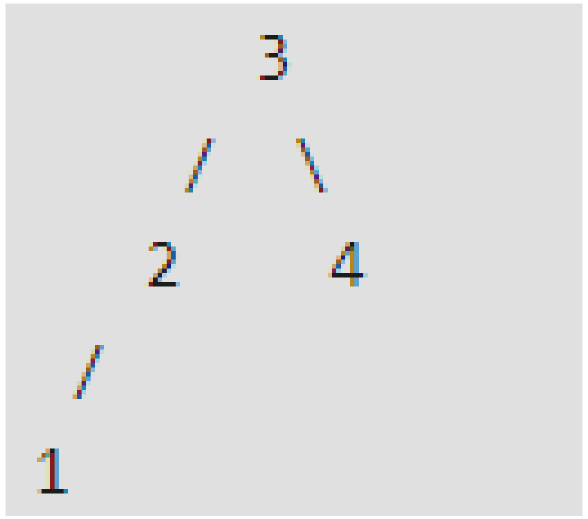
Sorted Array to Balanced BST

Given a sorted array. Write a function that creates a Balanced Binary Search Tree using array elements.

Example 2:

Input array: {1, 2, 3, 4}

Output Balanced BST



Sorted Array to Balanced BST

Given a sorted array. Write a function that creates a Balanced Binary Search Tree using array elements.

Solution:

1. **Get the middle of the array and make it root**

Sorted Array to Balanced BST

Given a sorted array. Write a function that creates a Balanced Binary Search Tree using array elements.

Solution:

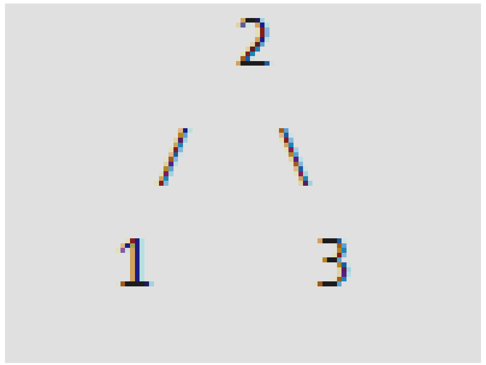
1. **Get the middle of the array and make it root**
2. **Recursively do the same for left and right halves**
   - **Get the middle of left half and make it left child of the root created in step 1**
   - **Get the middle of right half and make it right child of the root created in step 1**

Sorted Array to Balanced BST

Given a sorted array. Write a function that creates a Balanced Binary Search Tree using array elements.

Consider Example 1:

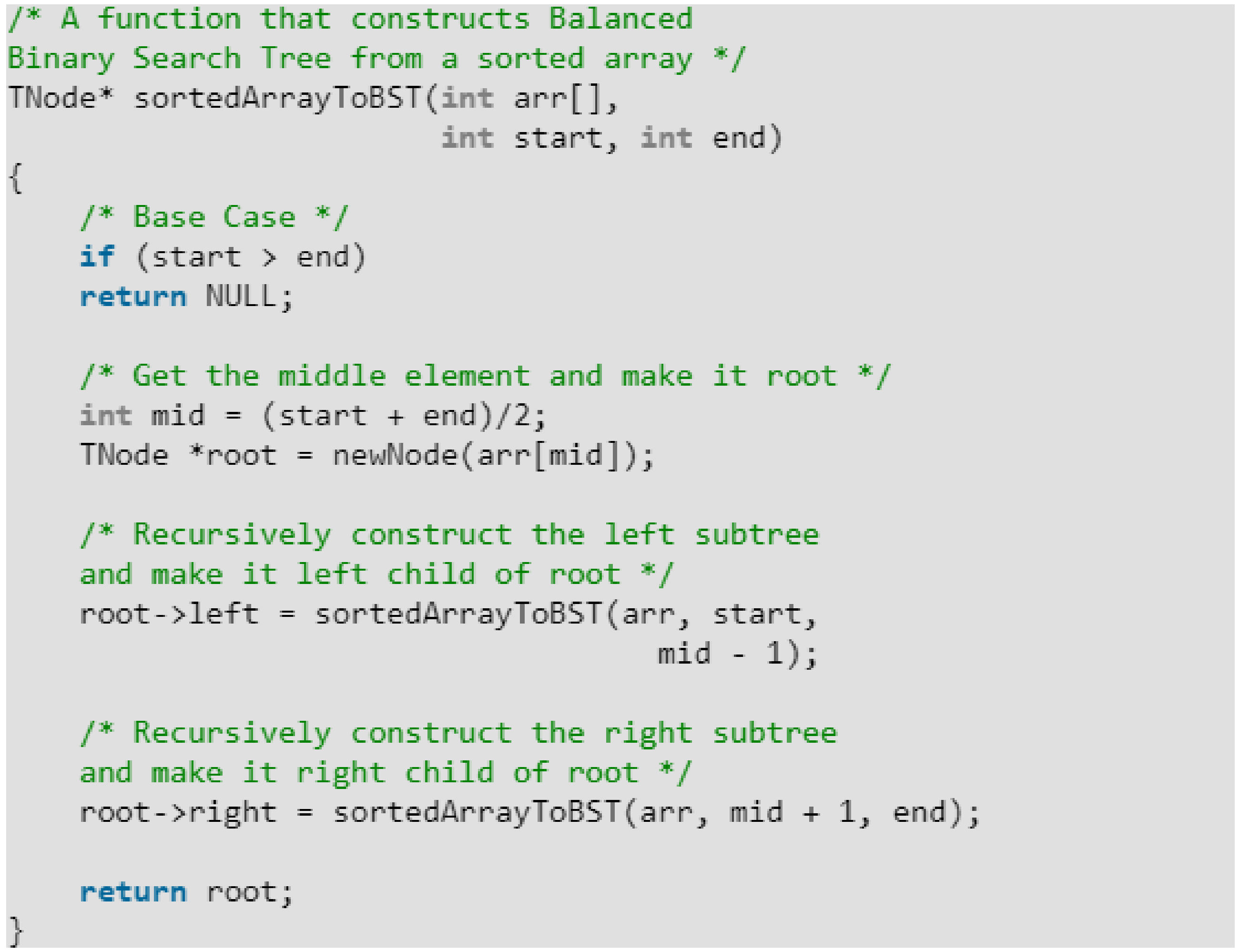
Input array: {1, 2, 3}



Here, the middle element of the sorted array is 2

This is made the root of the required BST

Sorted Array to Balanced BST



```
/* A function that constructs Balanced
Binary Search Tree from a sorted array */
TNode* sortedArrayToBST(int arr[],
                        int start, int end)
{
    /* Base Case */
    if (start > end)
    return NULL;

    /* Get the middle element and make it root */
    int mid = (start + end)/2;
    TNode *root = newNode(arr[mid]);

    /* Recursively construct the left subtree
    and make it left child of root */
    root->left = sortedArrayToBST(arr, start,
                                    mid - 1);

    /* Recursively construct the right subtree
    and make it right child of root */
    root->right = sortedArrayToBST(arr, mid + 1, end);

    return root;
}
```

Sorted Array to Balanced BST

Time Complexity

Sorted Array to Balanced BST

Time Complexity

Following is the recurrence relation for the function:

$T(n) = 2T(n/2) + C$

- **$T(n)$: Time taken for an array of size n**
- **C: Constant (Finding middle of array and linking root to left and right subtrees take constant time)**

Sorted Array to Balanced BST

Time Complexity

Following is the recurrence relation for the function:

$T(n) = 2T(n/2) + C$

- **T(n): Time taken for an array of size n**
- **C: Constant (Finding middle of array and linking root to left and right subtrees take constant time)**

Solving it using Master’s theorem, we get the time complexity as O(n).

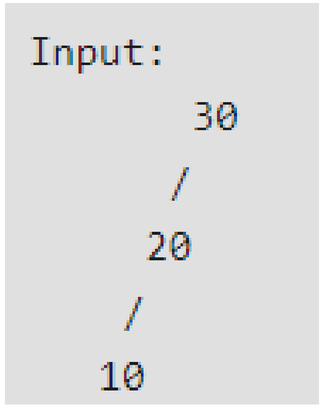
Convert a normal BST to Balanced BST

Given a BST (Binary Search Tree) that may be unbalanced, convert it into a balanced BST that has minimum possible height.

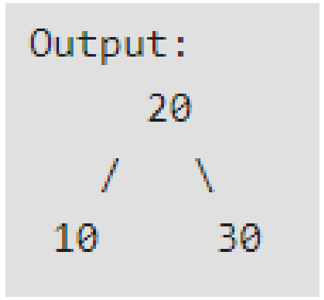
Convert a normal BST to Balanced BST

Given a BST (Binary Search Tree) that may be unbalanced, convert it into a balanced BST that has minimum possible height.

Example 1:



```
Input:
         30
        /
      20
     /
   10
```

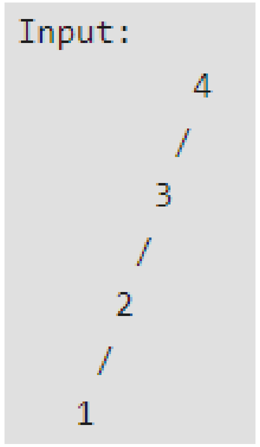


```
Output:
     20
   /   \
 10     30
```

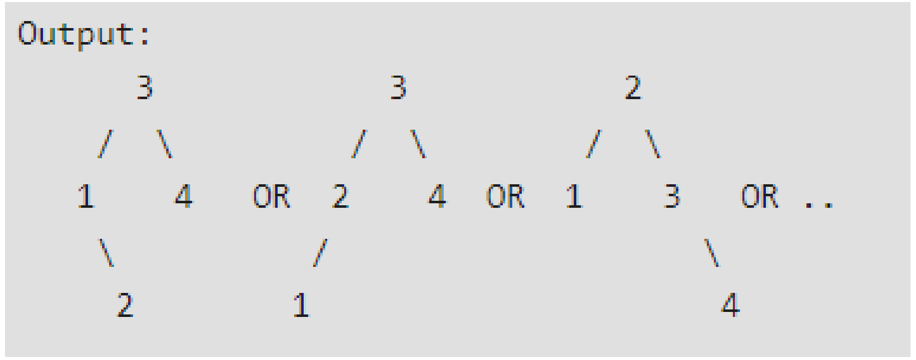
Convert a normal BST to Balanced BST

Given a BST (Binary Search Tree) that may be unbalanced, convert it into a balanced BST that has minimum possible height.

Example 2:



```
Input:
         4
        /
        3
      /
     2
    /
   1
```

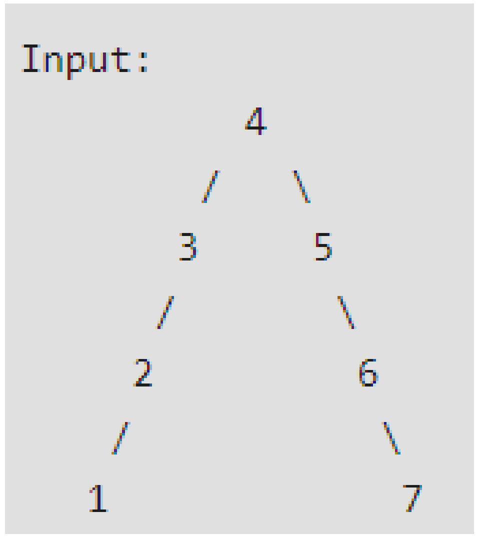


```
Output:
      3            3           2
    /  \         /  \        /  \
   1    4   OR   2    4  OR  1    3   OR ..
    \          /                   \
     2        1                     4
```

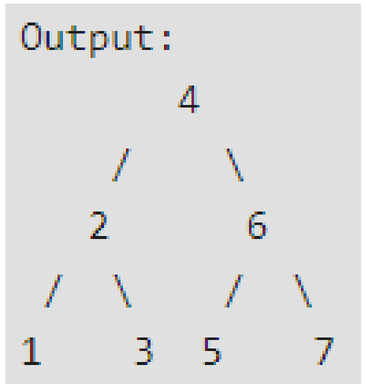
Convert a normal BST to Balanced BST

Given a BST (Binary Search Tree) that may be unbalanced, convert it into a balanced BST that has minimum possible height.

Example 3:



```
Input:
          4
        /   \
       3     5
      /       \
     2         6
    /           \
   1             7
```



```
Output:
       4
     /   \
    2     6
   / \   / \
  1   3 5   7
```

Convert a normal BST to Balanced BST

- An Efficient Solution can construct balanced BST in O(n) time with minimum possible height. Below are steps.

Convert a normal BST to Balanced BST

- **An Efficient Solution can construct balanced BST in O(n) time with minimum possible height. Below are steps.**

  1. **Traverse given BST in inorder and store result in an array. This step takes O(n) time. This array would be sorted as inorder traversal of BST always produces sorted sequence.**

Convert a normal BST to Balanced BST

- **An Efficient Solution can construct balanced BST in O(n) time with minimum possible height. Below are steps.**

1. **Traverse given BST in inorder and store result in an array. This step takes O(n) time. This array would be sorted as inorder traversal of BST always produces sorted sequence.**

2. **Build a balanced BST from the above created sorted array using the recursive approach. This step also takes O(n) time as we traverse every element exactly once and processing an element takes O(1) time.**

Merge Two Balanced Binary Search Trees

You are given two balanced binary search trees. Write a function that merges the two given balanced BSTs into a balanced binary search tree. Let there be m elements in first tree and n elements in the other tree. Your merge function should take O(m+n) time.

Merge Two Balanced Binary Search Trees

Method 1: Insert elements of first tree to second

Take all elements of first BST one by one, and insert them into the second BST

Merge Two Balanced Binary Search Trees

Method 1: Insert elements of first tree to second

Take all elements of first BST one by one, and insert them into the second BST

→ **Inserting an element to a self balancing BST takes Log n time where n is size of the BST.**

Merge Two Balanced Binary Search Trees

Method 1: Insert elements of first tree to second

Take all elements of first BST one by one, and insert them into the second BST

- → **Inserting an element to a self balancing BST takes Log n time where n is size of the BST.**
- → **So time complexity of this method is Log(n) + Log(n+1) … Log(m+n-1)**

Merge Two Balanced Binary Search Trees

Method 1: Insert elements of first tree to second

Take all elements of first BST one by one, and insert them into the second BST

- → **Inserting an element to a self balancing BST takes Log n time where n is size of the BST.**
- → **So time complexity of this method is Log(n) + Log(n+1) … Log(m+n-1)**
- → **The value of this expression will be between mLog(n) and mLog(m+n-1)**

Merge Two Balanced Binary Search Trees

Method 1: Insert elements of first tree to second

Take all elements of first BST one by one, and insert them into the second BST

- → **Inserting an element to a self balancing BST takes Log n time where n is size of the BST.**
- → **So time complexity of this method is Log(n) + Log(n+1) … Log(m+n-1)**
- → **The value of this expression will be between mLog(n) and mLog(m+n-1)**

We can pick the smaller tree as first tree.

Merge Two Balanced Binary Search Trees

Method 2: Merge Inorder Traversals

Merge Two Balanced Binary Search Trees

Method 2: Merge Inorder Traversals

1) **Do inorder traversal of first tree and store the traversal in one temp array arr1[]. This step takes O(m) time.**

Merge Two Balanced Binary Search Trees

Method 2: Merge Inorder Traversals

1) **Do inorder traversal of first tree and store the traversal in one temp array arr1[]. This step takes O(m) time.**
2) **Do inorder traversal of second tree and store the traversal in another temp array arr2[]. This step takes O(n) time.**

Merge Two Balanced Binary Search Trees

Method 2: Merge Inorder Traversals

1) **Do inorder traversal of first tree and store the traversal in one temp array arr1[]. This step takes O(m) time.**
2) **Do inorder traversal of second tree and store the traversal in another temp array arr2[]. This step takes O(n) time.**
3) **The arrays created in step 1 and 2 are sorted arrays. Merge the two sorted arrays into one array of size m + n. This step takes O(m+n) time.**

Merge Two Balanced Binary Search Trees

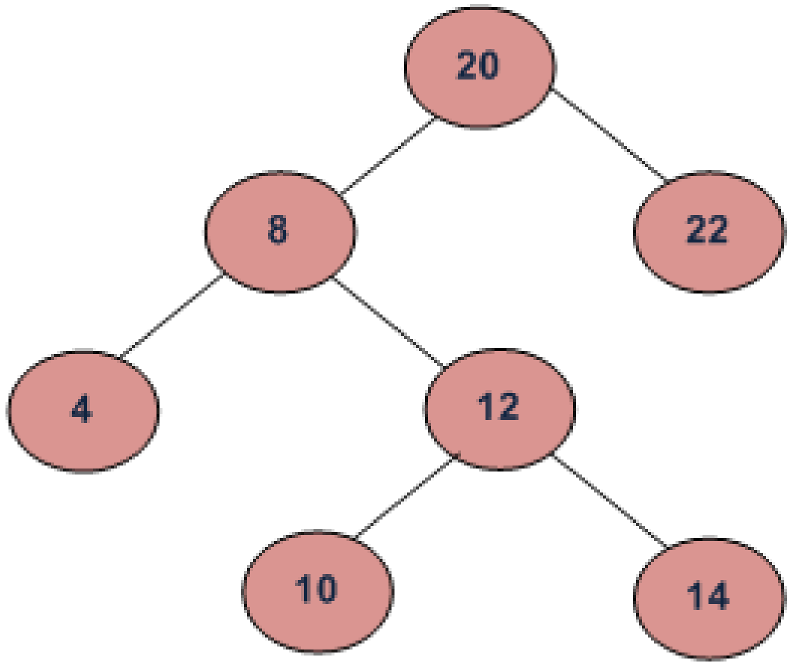
Method 2: Merge Inorder Traversals

1) **Do inorder traversal of first tree and store the traversal in one temp array arr1[]. This step takes O(m) time.**
2) **Do inorder traversal of second tree and store the traversal in another temp array arr2[]. This step takes O(n) time.**
3) **The arrays created in step 1 and 2 are sorted arrays. Merge the two sorted arrays into one array of size m + n. This step takes O(m+n) time.**
4) **Construct a balanced tree from the merged array using the technique discussed in this post. This step takes O(m+n) time.**

Find k-th smallest element in BST

Given root of binary search tree and K as input, find K-th smallest element in BST.

For example, in the following BST, if k = 3, then output should be 10, and if k = 5, then output should be 14.



Find k-th smallest element in BST

Given root of binary search tree and K as input, find K-th smallest element in BST.

Method 1: Using Inorder Traversal

Inorder traversal of BST retrieves elements of tree in the sorted order.

Find k-th smallest element in BST

Given root of binary search tree and K as input, find K-th smallest element in BST.

Method 1: Using Inorder Traversal

Inorder traversal of BST retrieves elements of tree in the sorted order.

Time complexity: O(n) where n is total nodes in tree

Find k-th smallest element in BST

Given root of binary search tree and K as input, find K-th smallest element in BST.

Method 2: Augmented Tree Data Structure

Find k-th smallest element in BST

Given root of binary search tree and K as input, find K-th smallest element in BST.

Method 2: Augmented Tree Data Structure

The idea is to maintain rank of each node. We can keep track of elements in a subtree of any node while building the tree. Since we need K-th smallest element, we can maintain number of elements of left subtree in every node.

Find k-th smallest element in BST

Given root of binary search tree and K as input, find K-th smallest element in BST.

Method 2: Augmented Tree Data Structure

Assume that the root is having N nodes in its left subtree.

- **If K = N + 1, root is K-th node.**

Find k-th smallest element in BST

Given root of binary search tree and K as input, find K-th smallest element in BST.

Method 2: Augmented Tree Data Structure

Assume that the root is having N nodes in its left subtree.

- **If K = N + 1, root is K-th node.**
- **If K < N, we will continue our search (recursion) for the Kth smallest element in the left subtree of root.**

Find k-th smallest element in BST

Given root of binary search tree and K as input, find K-th smallest element in BST.

Method 2: Augmented Tree Data Structure

Assume that the root is having N nodes in its left subtree.

- **If K = N + 1, root is K-th node.**
- **If K < N, we will continue our search (recursion) for the Kth smallest element in the left subtree of root.**
- **If K > N + 1, we continue our search in the right subtree for the (K – N – 1)-th smallest element.**

Find k-th smallest element in BST

Given root of binary search tree and K as input, find K-th smallest element in BST.

Method 2: Augmented Tree Data Structure

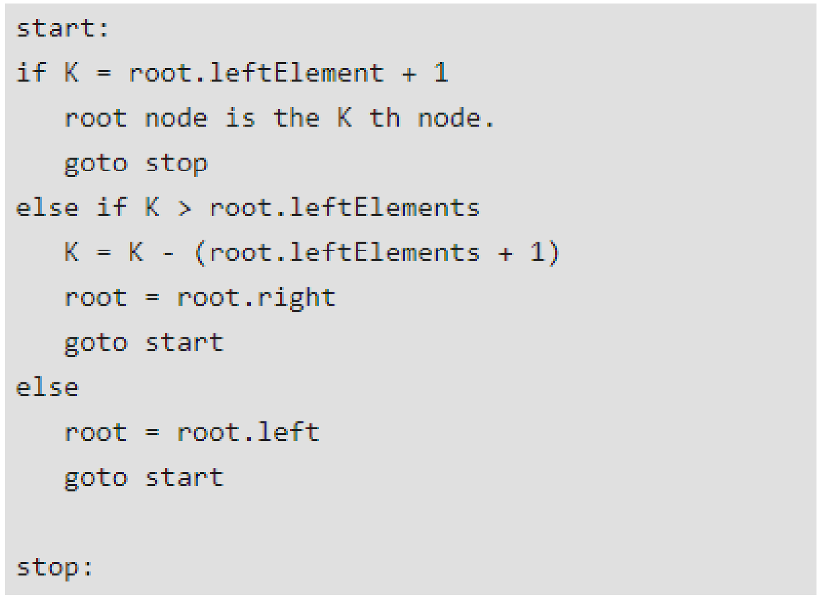
Assume that the root is having N nodes in its left subtree.

- **If K = N + 1, root is K-th node.**
- **If K < N, we will continue our search (recursion) for the Kth smallest element in the left subtree of root.**
- **If K > N + 1, we continue our search in the right subtree for the (K – N – 1)-th smallest element.**
- **Note that we need the count of elements in left subtree only.**

Find k-th smallest element in BST

Given root of binary search tree and K as input, find K-th smallest element in BST.

Method 2: Augmented Tree Data Structure



```
start:
if K = root.leftElement + 1
   root node is the K th node.
   goto stop
else if K > root.leftElements
   K = K - (root.leftElements + 1)
   root = root.right
   goto start
else
   root = root.left
   goto start

stop:
```

Find k-th smallest element in BST

Given root of binary search tree and K as input, find K-th smallest element in BST.

Method 2: Augmented Tree Data Structure

Time complexity: O(h) where h is height of tree.

Reference

- https://www.geeksforgeeks.org