Find the largest BST subtree in a given Binary Tree

Given a Binary Tree, write a function that returns the size of the largest subtree which is also a Binary Search Tree (BST). If the complete Binary Tree is BST, then return the size of whole tree.

Examples:

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

Output: 3
The following subtree is the maximum size BST subtree
   2
 /  \
1    3


Input:
       50
     /    \
  30       60
 /  \     /  \
5   20   45    70
              /  \
            65    80
Output: 5
The following subtree is the maximum size BST subtree
      60
     /  \
   45    70
        /  \
      65    80
```

Method 1 (Simple but inefficient)

Start from root and do an inorder traversal of the tree. For each node N, check whether the subtree rooted with N is BST or not. If BST, then return size of the subtree rooted with N. Else, recur down the left and right subtrees and return the maximum of values returned by left and right subtrees.

```
/*
  See http://www.geeksforgeeks.org/archives/632 for implementation of size()

  See Method 3 of http://www.geeksforgeeks.org/archives/3042 for
  implementation of isBST()

  max() returns maximum of two integers
*/
int largestBST(struct node *root)
{
   if (isBST(root))
     return size(root);
   else
    return max(largestBST(root->left), largestBST(root->right));
}
```

Time Complexity: The worst case time complexity of this method will be O(n^2). Consider a skewed tree for worst case analysis.

Method 2 (Tricky and Efficient)

In method 1, we traverse the tree in top down manner and do BST test for every node. If we traverse the tree in bottom up manner, then we can pass information about subtrees to the parent. The passed information can be used by the parent to do BST test (for parent node) only in constant time (or O(1) time). A left subtree need to tell the parent whether it is BST or not and also need to pass maximum value in it. So that we can compare the maximum value with the parent's data to check the BST property. Similarly, the right subtree need to pass the minimum value up the tree. The subtrees need to pass the following information up the tree for the finding the largest BST.
1) Whether the subtree itself is BST or not (In the following code, is_bst_ref is used for this purpose)
2) If the subtree is left subtree of its parent, then maximum value in it. And if it is right subtree then minimum value in it.
3) Size of this subtree if this subtree is BST (In the following code, return value of largestBSTtil() is used for this purpose)

max_ref is used for passing the maximum value up the tree and min_ptr is used for passing minimum value up the tree.

C

```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

/* 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 largestBSTUtil(struct node* node, int *min_ref, int *max_ref,
                             int *max_size_ref, bool *is_bst_ref);

/* Returns size of the largest BST subtree in a Binary Tree
  (efficient version). */
int largestBST(struct node* node)
{
  // Set the initial values for calling largestBSTUtil()
  int min = INT_MAX;  // For minimum value in right subtree
  int max = INT_MIN;  // For maximum value in left subtree

  int max_size = 0;  // For size of the largest BST
  bool is_bst = 0;

  largestBSTUtil(node, &min, &max, &max_size, &is_bst);

  return max_size;
}

/* largestBSTUtil() updates *max_size_ref for the size of the largest BST
   subtree.   Also, if the tree rooted with node is non-empty and a BST,
   then returns size of the tree. Otherwise returns 0.*/
int largestBSTUtil(struct node* node, int *min_ref, int *max_ref,
                            int *max_size_ref, bool *is_bst_ref)
{

  /* Base Case */
  if (node == NULL)
  {
     *is_bst_ref = 1; // An empty tree is BST
     return 0;    // Size of the BST is 0
  }

  int min = INT_MAX;

  /* A flag variable for left subtree property
     i.e., max(root->left) < root->data */
  bool left_flag = false;

  /* A flag variable for right subtree property
     i.e., min(root->right) > root->data */
  bool right_flag = false;

  int ls, rs; // To store sizes of left and right subtrees

  /* Following tasks are done by recursive call for left subtree
    a) Get the maximum value in left subtree (Stored in *max_ref)
    b) Check whether Left Subtree is BST or not (Stored in *is_bst_ref)
    c) Get the size of maximum size BST in left subtree (updates *max_size) */
  *max_ref = INT_MIN;
  ls = largestBSTUtil(node->left, min_ref, max_ref, max_size_ref, is_bst_ref);
  if (*is_bst_ref == 1 && node->data > *max_ref)
     left_flag = true;

  /* Before updating *min_ref, store the min value in left subtree. So that we
     have the correct minimum value for this subtree */
  min = *min_ref;

  /* The following recursive call does similar (similar to left subtree)
    task for right subtree */
  *min_ref =  INT_MAX;
  rs = largestBSTUtil(node->right, min_ref, max_ref, max_size_ref, is_bst_ref);
  if (*is_bst_ref == 1 && node->data < *min_ref)
     right_flag = true;

  // Update min and max values for the parent recursive calls
  if (min < *min_ref)
     *min_ref = min;
  if (node->data < *min_ref) // For leaf nodes
     *min_ref = node->data;
  if (node->data > *max_ref)
     *max_ref = node->data;

  /* If both left and right subtrees are BST. And left and right
     subtree properties hold for this node, then this tree is BST.
     So return the size of this tree */
  if(left_flag && right_flag)
  {
     if (ls + rs + 1 > *max_size_ref)
         *max_size_ref = ls + rs + 1;
     return ls + rs + 1;
  }
  else
  {
    //Since this subtree is not BST, set is_bst flag for parent calls
     *is_bst_ref = 0;
     return 0;
  }
}

/* Driver program to test above functions*/
int main()
{
    /* Let us construct the following Tree
          50
       /      \
     10        60
    /  \       /  \
   5   20    55    70
            /     /  \
          45     65    80
  */
  struct node *root = newNode(50);
  root->left        = newNode(10);
  root->right       = newNode(60);
  root->left->left  = newNode(5);
  root->left->right = newNode(20);
  root->right->left  = newNode(55);
  root->right->left->left  = newNode(45);
  root->right->right = newNode(70);
  root->right->right->left = newNode(65);
  root->right->right->right = newNode(80);

  /* The complete tree is not BST as 45 is in right subtree of 50.
     The following subtree is the largest BST
        60
      /  \
    55    70
   /     /  \
 45     65    80
  */
  printf(" Size of the largest BST is %d", largestBST(root));

  getchar();
  return 0;
}
```

Java

```
// Java program to find largest BST subtree in given Binary Tree

class Node {

    int data;
    Node left, right;

    Node(int d) {
        data = d;
        left = right = null;
    }
}

class Value {

    int max_size = 0; // for size of largest BST
    boolean is_bst = false;
    int min = Integer.MAX_VALUE;  // For minimum value in right subtree
    int max = Integer.MIN_VALUE;  // For maximum value in left subtree

}

class BinaryTree {

    static Node root;
    Value val = new Value();

    /* Returns size of the largest BST subtree in a Binary Tree
     (efficient version). */
    int largestBST(Node node) {

        largestBSTUtil(node, val, val, val, val);

        return val.max_size;
    }

    /* largestBSTUtil() updates *max_size_ref for the size of the largest BST
     subtree.   Also, if the tree rooted with node is non-empty and a BST,
     then returns size of the tree. Otherwise returns 0.*/
    int largestBSTUtil(Node node, Value min_ref, Value max_ref,
            Value max_size_ref, Value is_bst_ref) {

        /* Base Case */
        if (node == null) {
            is_bst_ref.is_bst = true; // An empty tree is BST
            return 0;    // Size of the BST is 0
        }

        int min = Integer.MAX_VALUE;

        /* A flag variable for left subtree property
         i.e., max(root->left) < root->data */
        boolean left_flag = false;

        /* A flag variable for right subtree property
         i.e., min(root->right) > root->data */
        boolean right_flag = false;

        int ls, rs; // To store sizes of left and right subtrees

        /* Following tasks are done by recursive call for left subtree
         a) Get the maximum value in left subtree (Stored in *max_ref)
         b) Check whether Left Subtree is BST or not (Stored in *is_bst_ref)
         c) Get the size of maximum size BST in left subtree (updates *max_size) */
        max_ref.max = Integer.MIN_VALUE;
        ls = largestBSTUtil(node.left, min_ref, max_ref, max_size_ref, is_bst_ref);
        if (is_bst_ref.is_bst == true && node.data > max_ref.max) {
            left_flag = true;
        }

        /* Before updating *min_ref, store the min value in left subtree. So that we
         have the correct minimum value for this subtree */
        min = min_ref.min;

        /* The following recursive call does similar (similar to left subtree)
         task for right subtree */
        min_ref.min = Integer.MAX_VALUE;
        rs = largestBSTUtil(node.right, min_ref, max_ref, max_size_ref, is_bst_ref);
        if (is_bst_ref.is_bst == true && node.data < min_ref.min) {
            right_flag = true;
        }

        // Update min and max values for the parent recursive calls
        if (min < min_ref.min) {
            min_ref.min = min;
        }
        if (node.data < min_ref.min) // For leaf nodes
        {
            min_ref.min = node.data;
        }
        if (node.data > max_ref.max) {
            max_ref.max = node.data;
        }

        /* If both left and right subtrees are BST. And left and right
         subtree properties hold for this node, then this tree is BST.
         So return the size of this tree */
        if (left_flag && right_flag) {
            if (ls + rs + 1 > max_size_ref.max_size) {
                max_size_ref.max_size = ls + rs + 1;
            }
            return ls + rs + 1;
        } else {
            //Since this subtree is not BST, set is_bst flag for parent calls
            is_bst_ref.is_bst = false;
            return 0;
        }
    }

    public static void main(String[] args) {
        /* Let us construct the following Tree
                50
             /      \
            10        60
           /  \       /  \
          5   20    55    70
         /     /  \
        45   65    80
         */

        BinaryTree tree = new BinaryTree();
        tree.root = new Node(50);
        tree.root.left = new Node(10);
        tree.root.right = new Node(60);
        tree.root.left.left = new Node(5);
        tree.root.left.right = new Node(20);
        tree.root.right.left = new Node(55);
        tree.root.right.left.left = new Node(45);
        tree.root.right.right = new Node(70);
        tree.root.right.right.left = new Node(65);
        tree.root.right.right.right = new Node(80);

        /* The complete tree is not BST as 45 is in right subtree of 50.
         The following subtree is the largest BST
             60
            /  \
          55    70
          /     /  \
        45     65   80
        */
        System.out.println("Size of largest BST is " + tree.largestBST(root));
    }
}

// This code has been contributed by Mayank Jaiswal
```

Time Complexity: O(n) where n is the number of nodes in the given Binary Tree.