Data Structures & Algorithms

Sparsh technologies | Dream company :)

Programs and explaination from different websites

Mishra, Siddharth

2014

Contents

[Trees & Graphs 2](#_Toc395256722)

[Binary Search Tree (BST) 2](#_Toc395256723)

[Searching a key 2](#_Toc395256724)

[Insertion of a key 3](#_Toc395256725)

[Time Complexity 4](#_Toc395256726)

[Deletion from BST 4](#_Toc395256727)

[Time Complexity 8](#_Toc395256728)

[Questions 8](#_Toc395256729)

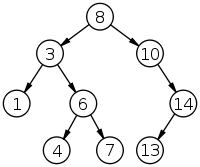
# Trees & Graphs

# Binary Search Tree (BST)

The following is definition of Binary Search Tree (BST) according to [WikiPedia](http://en.wikipedia.org/wiki/Binary_search_tree" \t "_blank)

Binary Search Tree, is a node-based binary tree data structure which 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.
* The left and right subtree each must also be a binary search tree.  
  There must be no duplicate nodes.

[](http://d18khu5s3lkxd9.cloudfront.net/wp-content/uploads/2014/01/200px-Binary_search_tree.svg_.png)

The above properties of Binary Search Tree provide an ordering among keys so that the operations like search, minimum and maximum can be done fast. If there is no ordering, then we may have to compare every key to search a given key.

## Searching a key

To search a given key in Bianry Search Tree, we first compare it with root, if the key is present at root, we return root. If key is greater than root’s key, we recur for right subtree of root node. Otherwise we recur for left subtree.

|  |
| --- |
| // C function to search a given key in a given BST  struct node\* search(struct node\* root, int key)  {      // Base Cases: root is null or key is present at root      if (root == NULL || root->key == key)         return root;        // Key is greater than root's key      if (root->key < key)         return search(root->right, key);        // Key is smaller than root's key      return search(root->left, key);  } |

## Insertion of a key

A new key is always inserted at leaf. We start searching a key from root till we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.

100 100

/ \ Insert 40 / \

20 500 ---------> 20 500

/ \ / \

10 30 10 30

\

40

|  |
| --- |
| // C program to demonstrate insert operation in binary search tree  #include<stdio.h>  #include<stdlib.h>    struct node  {      int key;      struct node \*left, \*right;  };    // A utility function to create a new BST node  struct node \*newNode(int item)  {      struct node \*temp =  (struct node \*)malloc(sizeof(struct node));      temp->key = item;      temp->left = temp->right = NULL;      return temp;  }    // A utility function to do inorder traversal of BST  void inorder(struct node \*root)  {      if (root != NULL)      {          inorder(root->left);          printf("%d ", root->key);          inorder(root->right);      }  }    /\* A utility function to insert a new node with given key in BST \*/  struct node\* insert(struct node\* node, int key)  {      /\* If the tree is empty, return a new node \*/      if (node == NULL) return newNode(key);        /\* Otherwise, recur down the tree \*/      if (key < node->key)          node->left  = insert(node->left, key);      else          node->right = insert(node->right, key);        /\* return the (unchanged) node pointer \*/      return node;  }    // Driver Program to test above functions  int main()  {      /\* Let us create following BST                50             /     \            30      70           /  \    /  \         20   40  60   80 \*/      struct node \*root = NULL;      root = insert(root, 50);      insert(root, 30);      insert(root, 20);      insert(root, 40);      insert(root, 70);      insert(root, 60);      insert(root, 80);        // print inoder traversal of the BST      inorder(root);        return 0;  } |

Output:

20 30 40 50 60 70 80

### Time Complexity

The worst case time complexity of search and insert operations is O(h) where h is height of Binary Search Tree. In worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of search and insert operation may become O(n).

## Deletion from BST

We have discussed [BST search and insert operations](http://geeksquiz.com/binary-search-tree-set-1-search-and-insertion/). In this post, delete operation is discussed. When we delete a node, there possibilities arise.

**1)*Node to be deleted is leaf:*** Simply remove from the tree.

50 50

/ \ delete(20) / \

30 70 ---------> 30 70

/ \ / \ \ / \

20 40 60 80 40 60 80

**2) *Node to be deleted has only one child:*** Copy the child to the node and delete the child

50 50

/ \ delete(30) / \

30 70 ---------> 40 70

\ / \ / \

40 60 80 60 80

**3) *Node to be deleted has two children:***Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

50 60

/ \ delete(50) / \

40 70 ---------> 40 70

/ \ \

60 80 80

The important thing to note is, inorder successor is needed only when right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in right child of the node.

|  |
| --- |
| // C program to demonstrate delete operation in binary search tree  #include<stdio.h>  #include<stdlib.h>    struct node  {      int key;      struct node \*left, \*right;  };    // A utility function to create a new BST node  struct node \*newNode(int item)  {      struct node \*temp =  (struct node \*)malloc(sizeof(struct node));      temp->key = item;      temp->left = temp->right = NULL;      return temp;  }    // A utility function to do inorder traversal of BST  void inorder(struct node \*root)  {      if (root != NULL)      {          inorder(root->left);          printf("%d ", root->key);          inorder(root->right);      }  }    /\* A utility function to insert a new node with given key in BST \*/  struct node\* insert(struct node\* node, int key)  {      /\* If the tree is empty, return a new node \*/      if (node == NULL) return newNode(key);        /\* Otherwise, recur down the tree \*/      if (key < node->key)          node->left  = insert(node->left, key);      else          node->right = insert(node->right, key);        /\* return the (unchanged) node pointer \*/      return node;  }    /\* Given a non-empty binary search tree, return the node with minimum     key value found in that tree. Note that the entire tree does not     need to be searched. \*/  struct node \* minValueNode(struct node\* node)  {      struct node\* current = node;        /\* loop down to find the leftmost leaf \*/      while (current->left != NULL)          current = current->left;        return current;  }    /\* Given a binary search tree and a key, this function deletes the key     and returns the new root \*/  struct node\* deleteNode(struct node\* root, int key)  {      // base case      if (root == NULL) return root;        // If the key to be deleted is smaller than the root's key,      // then it lies in left subtree      if (key < root->key)          root->left = deleteNode(root->left, key);        // If the key to be deleted is greater than the root's key,      // then it lies in right subtree      else if (key > root->key)          root->right = deleteNode(root->right, key);        // if key is same as root's key, then This is the node      // to be deleted      else      {          // node with only one child or no child          if (root->left == NULL)          {              struct node \*temp = root->right;              free(root);              return temp;          }          else if (root->right == NULL)          {              struct node \*temp = root->left;              free(root);              return temp;          }            // node with two children: Get the inorder successor (smallest          // in the right subtree)          struct node\* temp = minValueNode(root->right);            // Copy the inorder successor's content to this node          root->key = temp->key;            // Delete the inorder successor          root->right = deleteNode(root->right, temp->key);      }      return root;  }    // Driver Program to test above functions  int main()  {      /\* Let us create following BST                50             /     \            30      70           /  \    /  \         20   40  60   80 \*/      struct node \*root = NULL;      root = insert(root, 50);      root = insert(root, 30);      root = insert(root, 20);      root = insert(root, 40);      root = insert(root, 70);      root = insert(root, 60);      root = insert(root, 80);        printf("Inorder traversal of the given tree \n");      inorder(root);        printf("\nDelete 20\n");      root = deleteNode(root, 20);      printf("Inorder traversal of the modified tree \n");      inorder(root);        printf("\nDelete 30\n");      root = deleteNode(root, 30);      printf("Inorder traversal of the modified tree \n");      inorder(root);        printf("\nDelete 50\n");      root = deleteNode(root, 50);      printf("Inorder traversal of the modified tree \n");      inorder(root);        return 0;  } |

Output:

Inorder traversal of the given tree

20 30 40 50 60 70 80

Delete 20

Inorder traversal of the modified tree

30 40 50 60 70 80

Delete 30

Inorder traversal of the modified tree

40 50 60 70 80

Delete 50

Inorder traversal of the modified tree

40 60 70 80

### Time Complexity

The worst case time complexity of delete operation is O(h) where h is height of Binary Search Tree. In worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of delete operation may become O(n)

## Questions

|  |
| --- |
| **Question 1** |

What is the worst case time complexity for search, insert and delete operations in a general Binary Search Tree?

|  |  |
| --- | --- |
| A | O(n) for all |
| B | O(Logn) for all |
| C | O(Logn) for search and insert, and O(n) for delete |
| D | O(Logn) for search, and O(n) for insert and delete |

|  |
| --- |
| **Question 2** |

In delete operation of BST, we need inorder successor (or predecessor) of a node when the node to be deleted has both left and right child as non-empty. Which of the following is true about inorder successor needed in delete operation?

|  |  |
| --- | --- |
| A | Inorder Successor is always a leaf node |
| B | Inorder successor is always either a leaf node or a node with empty left child |
| C | Inorder successor may be an ancestor of the node |
| D | Inorder successor is always either a leaf node or a node with empty right child |

|  |
| --- |
| **Question 3** |

We are given a set of n distinct elements and an unlabeled binary tree with n nodes. In how many ways can we populate the tree with the given set so that it becomes a binary search tree? (GATE CS 2011)

|  |  |
| --- | --- |
| A | 0 |
| B | 1 |
| C | n! |
| D | (1/(n+1)).2nCn |

|  |
| --- |
| **Question 4** |

How many distinct binary search trees can be created out of 4 distinct keys?

|  |  |
| --- | --- |
| A | 4 |
| B | 14 |
| C | 24 |
| D | 42 |

|  |
| --- |
| **Question 5** |

Which of the following traversal outputs the data in sorted order in a BST?

|  |  |
| --- | --- |
| A | Preorder |
| B | Inorder |
| C | Postorder |
| D | Level order |

|  |
| --- |
| **Question 6** |

Suppose the numbers 7, 5, 1, 8, 3, 6, 0, 9, 4, 2 are inserted in that order into an initially empty binary search tree. The binary search tree uses the usual ordering on natural numbers. What is the in-order traversal sequence of the resultant tree?

|  |  |
| --- | --- |
| A | 7 5 1 0 3 2 4 6 8 9 |
| B | 0 2 4 3 1 6 5 9 8 7 |
| C | 0 1 2 3 4 5 6 7 8 9 |
| D | 9 8 6 4 2 3 0 1 5 7 |

|  |
| --- |
| **Question 7** |

The following numbers are inserted into an empty binary search tree in the given order: 10, 1, 3, 5, 15, 12, 16. What is the height of the binary search tree (the height is the maximum distance of a leaf node from the root)? (GATE CS 2004)

|  |  |
| --- | --- |
| A | 2 |
| B | 3 |
| C | 4 |
| D | 6 |

|  |
| --- |
| **Question 8** |

The preorder traversal sequence of a binary search tree is 30, 20, 10, 15, 25, 23, 39, 35, 42. Which one of the following is the postorder traversal sequence of the same tree?

|  |  |
| --- | --- |
| A | 10, 20, 15, 23, 25, 35, 42, 39, 30 |
| B | 15, 10, 25, 23, 20, 42, 35, 39, 30 |
| C | 15, 20, 10, 23, 25, 42, 35, 39, 30 |
| D | 15, 10, 23, 25, 20, 35, 42, 39, 30 |

|  |
| --- |
| **Question 9** |

Consider the following Binary Search Tree

10

/ \

5 20

/ / \

4 15 30

/

11

If we randomly search one of the keys present in above BST, what would be the expected number of comparisons?

|  |  |
| --- | --- |
| A | 2.75 |
| B | 2.25 |
| C | 2.57 |
| D | 3.25 |

|  |
| --- |
| **Question 10** |

Which of the following traversals is sufficient to construct BST from given traversals 1) Inorder 2) Preorder 3) Postorder

|  |  |
| --- | --- |
| A | Any one of the given three traversals is sufficient |
| B | Either 2 or 3 is sufficient |
| C | 2 and 3 |
| D | 1 and 3 |

|  |
| --- |
| **Question 11** |

Consider the following code snippet in C. The function print() receives root of a Binary Search Tree (BST) and a positive integer k as arguments.

|  |
| --- |
| // A BST node  struct node {      int data;      struct node \*left, \*right;  };    int count = 0;    void print(struct node \*root, int k)  {      if (root != NULL && count <= k)      {          print(root->right, k);          count++;          if (count == k)            printf("%d ", root->data);         print(root->left, k);      }  } |

What is the output of print(root, 3) where root represent root of the following BST.

15

/ \

10 20

/ \ / \

8 12 16 25

|  |  |
| --- | --- |
| A | 10 |
| B | 16 |
| C | 20 |
| D | 20 10 |

|  |
| --- |
| **Question 12** |

Consider the same code as given in above question. What does the function print() do in general? The function print() receives root of a Binary Search Tree (BST) and a positive integer k as arguments.

|  |  |  |
| --- | --- | --- |
| // A BST node  struct node {      int data;      struct node \*left, \*right;  };    int count = 0;    void print(struct node \*root, int k)  {      if (root != NULL && count <= k)      {          print(root->right, k);          count++;          if (count == k)            printf("%d ", root->data);         print(root->left, k);      }  } | | |
| A | Prints the kth smallest element in BST |
| B | Prints the kth largest element in BST |
| C | Prints the leftmost node at level k from root |
| D | Prints the rightmost node at level k from root |

|  |
| --- |
| **Question 13** |

You are given the postorder traversal, P, of a binary search tree on the n elements 1, 2, ..., n. You have to determine the unique binary search tree that has P as its postorder traversal. What is the time complexity of the most efficient algorithm for doing this?

|  |  |
| --- | --- |
| A | O(Logn) |
| B | O(n) |
| C | O(nLogn) |
| D | none of the above, as the tree cannot be uniquely determined. |

|  |
| --- |
| **Question 14** |

Suppose we have a balanced binary search tree T holding n numbers. We are given two numbers L and H and wish to sum up all the numbers in T that lie between L and H. Suppose there are m such numbers in T. If the tightest upper bound on the time to compute the sum is O(nalogb n + mc logd n), the value of a + 10b + 100c + 1000d is \_\_\_\_.

|  |  |
| --- | --- |
| A | 60 |
| B | 110 |
| C | 210 |
| D | 50 |

## Find the node with minimum value in a Binary Search Tree

This is quite simple. Just traverse the node from root to left recursively until left is NULL. The node whose left is NULL is the node with minimum value.



For the above tree, we start with 20, then we move left 8, we keep on moving to left until we see NULL. Since left of 4 is NULL, 4 is the node with minimum value.

|  |
| --- |
| #include <stdio.h>  #include<stdlib.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);  }    /\* Give a binary search tree and a number,  inserts a new node with the given number in  the correct place in the tree. Returns the new  root pointer which the caller should then use  (the standard trick to avoid using reference  parameters). \*/  struct node\* insert(struct node\* node, int data)  {    /\* 1. If the tree is empty, return a new,        single node \*/    if (node == NULL)      return(newNode(data));    else    {      /\* 2. Otherwise, recur down the tree \*/      if (data <= node->data)          node->left  = insert(node->left, data);      else          node->right = insert(node->right, data);        /\* return the (unchanged) node pointer \*/      return node;    }  }    /\* Given a non-empty binary search tree,  return the minimum data value found in that  tree. Note that the entire tree does not need  to be searched. \*/  int minValue(struct node\* node) {    struct node\* current = node;      /\* loop down to find the leftmost leaf \*/    while (current->left != NULL) {      current = current->left;    }    return(current->data);  }    /\* Driver program to test sameTree function\*/  int main()  {    struct node\* root = NULL;    root = insert(root, 4);    insert(root, 2);    insert(root, 1);    insert(root, 3);    insert(root, 6);    insert(root, 5);      printf("\n Minimum value in BST is %d", minValue(root));    getchar();    return 0;  } |

### **Time Complexity**

O(n) Worst case happens for left skewed trees. Similarly we can get the maximum value by recursively traversing the right node of a binary search tree.

## In-order predecessor and successor for a given key in BST

I recently encountered with a question in an interview at e-commerce company. The interviewer asked the following question:

There is BST given with root node with key part as integer only. The structure of each node is as follows:

|  |
| --- |
| struct Node  {      int key;      struct Node \*left, \*right ;  }; |

You need to find the inorder successor and predecessor of a given key. In case the given key is not found in BST, then return the two values within which this key will lie.

Following is the algorithm to reach the desired result. Its a recursive method:

Input: root node, key

output: predecessor node, successor node

1. If root is NULL

then return

2. if key is found then

a. If its left subtree is not null

Then predecessor will be the right most

child of left subtree or left child itself.

b. If its right subtree is not null

The successor will be the left most child

of right subtree or right child itself.

return

3. If key is smaller then root node

set the successor as root

search recursively into left subtree

else

set the predecessor as root

search recursively into right subtree

Following is C++ implementation of the above algorithm:

|  |
| --- |
| // C++ program to find predecessor and successor in a BST  #include <iostream>  using namespace std;    // BST Node  struct Node  {      int key;      struct Node \*left, \*right;  };    // This function finds predecessor and successor of key in BST.  // It sets pre and suc as predecessor and successor respectively  void findPreSuc(Node\* root, Node\*& pre, Node\*& suc, int key)  {      // Base case      if (root == NULL)  return ;        // If key is present at root      if (root->key == key)      {          // the maximum value in left subtree is predecessor          if (root->left != NULL)          {              Node\* tmp = root->left;              while (tmp->right)                  tmp = tmp->right;              pre = tmp ;          }            // the minimum value in right subtree is successor          if (root->right != NULL)          {              Node\* tmp = root->right ;              while (tmp->left)                  tmp = tmp->left ;              suc = tmp ;          }          return ;      }        // If key is smaller than root's key, go to left subtree      if (root->key > key)      {          suc = root ;          findPreSuc(root->left, pre, suc, key) ;      }      else // go to right subtree      {          pre = root ;          findPreSuc(root->right, pre, suc, key) ;      }  }    // A utility function to create a new BST node  Node \*newNode(int item)  {      Node \*temp =  new Node;      temp->key = item;      temp->left = temp->right = NULL;      return temp;  }    /\* A utility function to insert a new node with given key in BST \*/  Node\* insert(Node\* node, int key)  {      if (node == NULL) return newNode(key);      if (key < node->key)          node->left  = insert(node->left, key);      else          node->right = insert(node->right, key);      return node;  }    // Driver program to test above function  int main()  {      int key = 65;    //Key to be searched in BST       /\* Let us create following BST                50             /     \            30      70           /  \    /  \         20   40  60   80 \*/      Node \*root = NULL;      root = insert(root, 50);      insert(root, 30);      insert(root, 20);      insert(root, 40);      insert(root, 70);      insert(root, 60);      insert(root, 80);          Node\* pre = NULL, \*suc = NULL;        findPreSuc(root, pre, suc, key);      if (pre != NULL)        cout << "Predecessor is " << pre->key << endl;      else        cout << "No Predecessor";        if (suc != NULL)        cout << "Successor is " << suc->key;      else        cout << "No Successor";      return 0;  } |

Output:

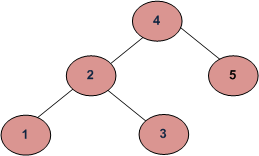
Predecessor is 60

Successor is 70

## 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 which 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.

From the above properties it naturally follows that:  
• Each node (item in the tree) has a distinct key.

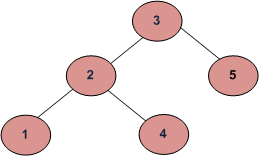


### METHOD 1 (Simple but Wrong)

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

|  |
| --- |
| 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 (and below tree is not a BST because 4 is in left subtree of 3)**



### 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.

|  |
| --- |
| /\* 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

### METHOD 3 (Correct and Efficient)

Method 2 above runs slowly since it traverses over some parts of the tree many times. A better solution looks at each node only once. The trick is to 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.

/\* Returns true if the given tree is a binary search tree

(efficient version). \*/

int isBST(struct 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(struct node\* node, int min, int max)

**Implementation:**

|  |
| --- |
| #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;  };    int isBSTUtil(struct node\* node, int min, int max);    /\* Returns true if the given tree is a binary search tree   (efficient version). \*/  int isBST(struct 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(struct 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  }    /\* 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);  }    /\* Driver program to test above functions\*/  int main()  {    struct node \*root = newNode(4);    root->left        = newNode(2);    root->right       = newNode(5);    root->left->left  = newNode(1);    root->left->right = newNode(3);      if(isBST(root))      printf("Is BST");    else      printf("Not a BST");      getchar();    return 0;  } |

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

### METHOD 4(Using In-Order Traversal)

Thanks to [LJW489](http://www.geeksforgeeks.org/archives/3042/comment-page-1" \l "comment-562)for suggesting this method.  
1) Do In-Order Traversal of the given tree and store the result in a temp array.  
3) Check if the temp array is sorted in ascending order, if it is, then the tree is BST.

Time Complexity: O(n)

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. Thanks to [ygos](http://www.geeksforgeeks.org/archives/3042/comment-page-1" \l "comment-5805)for this space optimization.

|  |
| --- |
| bool isBST(struct node\* root)  {      static struct 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;  } |

The use of static variable can also be avoided by using reference to prev node as a parameter (Similar to [this](http://www.geeksforgeeks.org/archives/17358)post).

**Sources:**  
<http://en.wikipedia.org/wiki/Binary_search_tree>  
<http://cslibrary.stanford.edu/110/BinaryTrees.html>