Determine if a Binary Tree is a Binary Search Tree (BST)

September 14, 2010 by 1337c0d3r 52 Replies

> Write a function isBST(BinaryTree *node) to verify if a given binary tree is a Binary Search Tree (BST) or not.

First, you must understand the difference between *Binary Tree* and *Binary Search Tree (BST)*. Binary tree is a tree data structure in which each node has at most two child nodes. A binary search tree (BST) is based on binary tree, but with the following additional 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.

This question is a very good interview question, because it *really* tests your understanding of the definition of BST. Most people will fall to this first trap, and gives the following algorithm:

> Assume that the current node's value is *k*. Then for each node, check if the left node's value is less than *k* and the right node's value is greater than *k*. If all of the nodes satisfy this property, then it is a BST.

It sounds correct and convincing, but look at this counter example below: A sample tree which we name it as binary tree (1).

```
  10
   /  \
  5    15       -------- binary tree (1)
     /  \
    6    20
```

It's obvious that this is not a valid BST, since (6) could never be on the right of (10).

Based on BST's definition, we can then easily devise a brute-force solution:

> Assume that the current node's value is *k*. Then for each node, check if all nodes of left subtree contain values that are less than *k*. Also check if all nodes of right subtree contain values that are greater than *k*. If all of the nodes satisfy this property, then it must be a BST.

Below is the brute force code (though inefficient, but works):

```
bool isSubTreeLessThan(BinaryTree *p, int val) {
  if (!p) return true;
  return (p->data < val &&
          isSubTreeLessThan(p->left, val) &&
```

```
          isSubTreeLessThan(p->right, val));
}

bool isSubTreeGreaterThan(BinaryTree *p, int val) {
  if (!p) return true;
  return (p->data > val &&
          isSubTreeGreaterThan(p->left, val) &&
          isSubTreeGreaterThan(p->right, val));
}

bool isBSTBruteForce(BinaryTree *p) {
  if (!p) return true;
  return isSubTreeLessThan(p->left, p->data) &&
         isSubTreeGreaterThan(p->right, p->data) &&
         isBSTBruteForce(p->left) &&
         isBSTBruteForce(p->right);
}
```

Solution:

Here is the much better solution. Instead of examining all nodes of both subtrees in each pass, we only need to examine two nodes in each pass.

Refer back to the binary tree (1) above. As we traverse down the tree from node (10) to right node (15), we know for sure that the right node's value fall between **10** and **+INFINITY**. Then, as we traverse further down from node (15) to left node (6), we know for sure that the left node's value fall between **10** and **15**. And since (6) does not satisfy the above requirement, we can quickly determine it is not a valid BST. All we need to do is to pass down the low and high limits from node to node! Once we figure out this algorithm, it is easy to code.

```
bool isBSTHelper(BinaryTree *p, int low, int high) {
  if (!p) return true;
  if (low < p->data && p->data < high)
    return isBSTHelper(p->left, low, p->data) &&
           isBSTHelper(p->right, p->data, high);
  else
    return false;
}

bool isBST(BinaryTree *root) {
  // INT_MIN and INT_MAX are defined in C++'s <climits> library
  return isBSTHelper(root, INT_MIN, INT_MAX);
}
```

This algorithm runs in O(N) time, where N is the number of nodes of the binary tree. It also uses O(1) space (neglecting the stack space used by calling function recursively).

Alternative Solution:

Another solution is to do an in-order traversal of the binary tree, and verify that the previous value (can be passed into the recursive function as reference) is less than the current value. This works because when you do an in-order traversal on a BST, the elements must be strictly in increasing order. This method also runs in O(N) time and O(1) space.

```
bool isBSTInOrderHelper(BinaryTree *p, int& prev) {
  if (!p) return true;
  return (isBSTInOrderHelper(p->left, prev) &&
          (p->data > prev) && (prev = p->data) &&
          isBSTInOrderHelper(p->right, prev));
```

```
}

bool isBSTInOrder(BinaryTree *root) {
  int prev = INT_MIN;
  return isBSTInOrderHelper(root, prev);
}
```

EDIT: (Bug fix)

An id han6 from the MITBBS forum pointed that the above code has a bug. When one of the node's value is 0, the function would return false straight away, even though it is a valid BST. Why?

Below is the corrected code:

```
bool isBSTInOrderHelper(BinaryTree *p, int& prev) {
  if (!p) return true;
  if (isBSTInOrderHelper(p->left, prev)) {
    if (p->data > prev) {
      prev = p->data;
      return isBSTInOrderHelper(p->right, prev);
    } else {
      return false;
    }
  }
  else {
    return false;
  }
}
```

Additional Exercise:

We know that the brute force solution is inefficient. But how does it compare to the better solutions? In other words, what is the run time complexity of the brute force solution? Try to estimate as best as you can, and then find the correct answer by proving it using Math. Does your estimate fares well with the correct answer? Why?

(Hint: The run time complexity for the brute force approach is NOT exponential.)