[判定一棵二叉树是否是二叉搜索树](http://blog.csdn.net/ssjhust123/article/details/7771096)

分类： [面试题系列](http://blog.csdn.net/ssjhust123/article/category/1112654) [数据结构与算法](http://blog.csdn.net/ssjhust123/article/category/1190606) [二叉树](http://blog.csdn.net/ssjhust123/article/category/1215817)2012-07-21 12:18 1225人阅读 [评论](http://blog.csdn.net/ssjhust123/article/details/7771096#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/ssjhust123/article/details/7771096#report)

目录[(?)[+]](http://blog.csdn.net/ssjhust123/article/details/7771096)

**问题**

给定一棵二叉树，判定该二叉树是否是二叉搜索树（Binary Search Tree）？

**解法1:暴力搜索**

首先说明一下二叉树和二叉搜索树的区别。二叉树指这样的树结构，它的每个结点的孩子数目最多为2个；二叉搜索树是一种二叉树，但是它有附加的一些约束条件，这些约束条件必须对每个结点都成立：

* 结点node的左子树所有结点的值都小于node的值。
* 结点node的右子树所有结点的值都大于node的值。
* 结点node的左右子树同样都必须是二叉搜索树。

该问题在面试中也许经常问到，考察的是对二叉搜索树定义的理解。初看这个问题，也许会想这样来实现：

假定当前结点值为k。对于二叉树中每个结点，判断其左孩子的值是否小于k，其右孩子的值是否大于k。如果所有结点都满足该条件，则该二叉树是一棵二叉搜索树。

**很不幸的是，这个算法是错误的。考虑下面的二叉树，它符合上面算法的条件，但是它不是一棵二叉搜索树。**

    10

/ \

5 15 -------- binary tree (1)

/ \

6 20

那么，根据二叉搜索树的定义，可以想到一种暴力搜索的方法来判定二叉树是否为二叉搜索树。

**假定当前结点值为k。则对于二叉树中每个结点，其左子树所有结点的值必须都小于k，其右子树所有结点的值都必须大于k。**

暴力搜索算法代码如下，虽然效率不高，但是它确实能够完成工作。该解法最坏情况复杂度为O（n^2)，n为结点数目。（当所有结点都在一边的时候出现最坏情况）

**[cpp]** [view plaincopy](http://blog.csdn.net/ssjhust123/article/details/7771096)

1. /\*判断左子树的结点值是否都小于val\*/
2. **bool** isSubTreeLessThan(BinaryTree \*p, **int** val)
3. {
4. **if** (!p) **return** **true**;
5. **return** (p->data < val &&
6. isSubTreeLessThan(p->left, val) &&
7. isSubTreeLessThan(p->right, val));
8. }
9. /\*判断右子树的结点值是否都大于val\*/
10. **bool** isSubTreeGreaterThan(BinaryTree \*p, **int** val)
11. {
12. **if** (!p) **return** **true**;
13. **return** (p->data > val &&
14. isSubTreeGreaterThan(p->left, val) &&
15. isSubTreeGreaterThan(p->right, val));
16. }
17. /\*判定二叉树是否是二叉搜索树\*/
18. **bool** isBSTBruteForce(BinaryTree \*p)
19. {
20. **if** (!p) **return** **true**;
21. **return** isSubTreeLessThan(p->left, p->data) &&
22. isSubTreeGreaterThan(p->right, p->data) &&
23. isBSTBruteForce(p->left) &&
24. isBSTBruteForce(p->right);
25. }

一个类似的解法是：对于结点node，判断其左子树最大值是否大于node的值，如果是，则该二叉树不是二叉搜索树。如果不是，则接着判断右子树最小值是否小于或等于node的值，如果是，则不是二叉搜索树。如果不是则接着递归判断左右子树是否是二叉搜索树。（代码中的maxValue和minValue函数功能分别是返回二叉树中的最大值和最小值，这里假定二叉树为二叉搜索树，实际返回的不一定是最大值和最小值）

**[cpp]** [view plaincopy](http://blog.csdn.net/ssjhust123/article/details/7771096)

1. **int** isBST(**struct** node\* node)
2. {
3. **if** (node==NULL) **return**(**true**);
4. //如果左子树最大值>=当前node的值，则返回false
5. **if** (node->left!=NULL && maxValue(node->left) >= node->data)
6. **return**(**false**);
7. // 如果右子树最小值<=当前node的值，返回false
8. **if** (node->right!=NULL && minValue(node->right) <= node->data)
9. **return**(**false**);
10. // 如果左子树或者右子树不是BST，返回false
11. **if** (!isBST(node->left) || !isBST(node->right))
12. **return**(**false**);
13. // 通过所有测试，返回true
14. **return**(**true**);
15. }

**解法2：更好的解法**

以前面提到的binary tree（1）为例，当我们从结点10遍历到右结点15时，我们知道右子树结点值肯定都在**10和+INFINITY（无穷大）**之间**。**当我们遍历到结点15的左孩子结点6时，我们知道结点15的左子树结点值都必须在10到15之间。显然，结点6不符合条件，因此它不是一棵二叉搜索树。该算法代码如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/ssjhust123/article/details/7771096)

1. **int** isBST2(**struct** node\* node)
2. {
3. **return**(isBSTUtil(node, INT\_MIN, INT\_MAX));
4. }
5. /\*
6. 给定的二叉树是BST则返回true，且它的值  >min 以及 < max.
7. \*/
8. **int** isBSTUtil(**struct** node\* node, **int** min, **int** max)
9. {
10. **if** (node==NULL) **return**(**true**);
11. // 如果不满足min和max约束，返回false
12. **if** (node->data<=min || node->data>=max) **return**(**false**);
13. // 递归判断左右子树是否满足min和max约束条件
14. **return**
15. isBSTUtil(node->left, min, node->data) &&
16. isBSTUtil(node->right, node->data, max)
17. );
18. }

由于该算法只需要访问每个结点1次，因此时间复杂度为O（n），比解法1效率高很多。

**解法3：中序遍历算法**

因为一棵二叉搜索树的中序遍历后其结点值是从小到大排好序的，所以依此给出下面的解法。该解法时间复杂度也是O（n）。

**[cpp]** [view plaincopy](http://blog.csdn.net/ssjhust123/article/details/7771096)

1. **bool** isBSTInOrder(BinaryTree \*root)
2. {
3. **int** prev = INT\_MIN;
4. **return** isBSTInOrderHelper(root, prev);
5. }
6. /\*该函数判断二叉树p是否是一棵二叉搜索树，且其结点值都大于prev\*/
7. **bool** isBSTInOrderHelper(BinaryTree \*p, **int**& prev)
8. {
9. **if** (!p) **return** **true**;
10. **if** (isBSTInOrderHelper(p->left, prev)) { // 如果左子树是二叉搜索树，且结点值都大于prev
11. **if** (p->data > prev) { //判断当前结点值是否大于prev，因为此时prev已经设置为已经中序遍历过的结点的最大值。
12. prev = p->data;
13. **return** isBSTInOrderHelper(p->right, prev); //若结点值大于prev，则设置prev为当前结点值，并判断右子树是否二叉搜索树且结点值都大于prev。
14. } **else** {
15. **return** **false**;
16. }
17. }
18. **else** {
19. **return** **false**;
20. }
21. }