二叉搜索树

1. 对于BST每个节点node,其左子树节点值都比node的值小，右子树的值都比node节点的值大；
2. 对于每个节点node的左右子树都是BST。

BST的中序遍历的结果是升序序列。

```
void traverse(TreeNode *root) {
  printf("%d ", root->val);
  traverse(root->left);
  traverse(root->right);
}
```

查找二叉树的第k小的值

分析：

1. BST的中序遍历是一个递增序列；采用中序遍历的方式查找；
2. 采用全局变量，记录当前元素在二叉树中的位置，向下遍历；当index == k时得到结果。

```
class Solution {
public:
  int kthSmallest(TreeNode *root, int k) {
    rank = 0;
    res  = 0;
    traverse(root, k);

    return res;
  }

private:
  void traverse(TreeNode *root, int k) {
    if (root == nullptr) {
      return;
    }
    traverse(root->left, k);
    rank++;
    if (k == rank) {
      res = root->val;
      return;
    }
    traverse(root->right, k);
  }

  int res;
  int rank;
};
```

BST转为累加树

> 给出二叉 搜索 树的根节点，该树的节点值各不相同，请你将其转换为累加树（Greater Sum Tree），使每个节点 node 的新值等于原树中大于或等于 node.val 的值之和。
>
> 提醒一下，二叉搜索树满足下列约束条件：
>
> 节点的左子树仅包含键 小于 节点键的节点。
> 节点的右子树仅包含键 大于 节点键的节点。
> 左右子树也必须是二叉搜索树。

分析：

1. BST右子树大于左子树；
2. 将中序遍历修改，先遍历右子树，再遍历根节点，最后遍历左子树，这样将二叉树遍历得到一个递减序列；
3. 利用一个计数器，每次加root的值，在赋值给root.

```
class Solution {
public:
  TreeNode *convertBST(TreeNode *root) {
    sum = 0;
    traverse(root);

    return root;
  }

private:
  void traverse(TreeNode *root) {
    if (root == nullptr) {
      return;
    }

    traverse(root->right);
    sum += root->val;
    root->val = sum;
    traverse(root->left);
  }

  int sum;
};
```