题目：

Given a binary search tree, write a function `kthSmallest` to find the **k**th smallest element in it.

Note:

You may assume k is always valid, 1 ≤ k ≤ BST's total elements.

Follow up:

What if the BST is modified (insert/delete operations) often and you need to find the kth smallest frequently? How would you optimize the kthSmallest routine?

Hint:

1. Try to utilize the property of a BST.
2. What if you could modify the BST node's structure?
3. The optimal runtime complexity is O(height of BST).

1.时间：O（N）；空间：O（1）

```
class Solution {
public:
    int kthSmallest(TreeNode* root, int k) {
        if (root == nullptr) return -1;
        int count = 0, result = -1;
        inOrder(root, count, k, result);
        return result;
    }
private:
```

```
    bool inOrder(TreeNode* root, int& count, int k, int& result){
        if (root == nullptr) return false;
        if (inOrder(root->left, count, k, result)) return true;
        if (count == k - 1){
            result = root->val;
            return true;
        }
        count++;
        return inOrder(root->right, count, k, result);
    }
};
```

2.时间：O（N）；空间：O（N）

```
class Solution {
public:
    int calcTreeSize(TreeNode* root){
        if (root == NULL)
            return 0;
        return 1+calcTreeSize(root->left) + calcTreeSize(root->right);
    }
    int kthSmallest(TreeNode* root, int k) {
        if (root == NULL)
            return 0;
```

```
            int leftSize = calcTreeSize(root->left);
            if (k == leftSize+1){
                return root->val;
            }else if (leftSize >= k){
                return kthSmallest(root->left,k);
            }else{
                return kthSmallest(root->right, k-leftSize-1);
            }
        }
};
```

3.时间：O（N）；空间：O（N） ->非递归

```
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};
class Solution {
public:
    int kthSmallest(TreeNode* root, int k) {
        if (root == nullptr) return -1;
        std::stack<TreeNode*> stack;
```

```
        TreeNode* curNode = root;
        while (!stack.empty() || curNode != nullptr){
            if (curNode != nullptr){
                stack.push(curNode);
                curNode = curNode->left;
            } else{
                curNode = stack.top();
                stack.pop();
                if (--k == 0) return curNode->val;
                curNode = curNode->right;
            }
        }
        return -1;
    }
};
```