Tree & Trie

//Tree struct define

```
struct TreeNode{
     int val;
     TreeNode* left;
     TreeNode* right;
     TreeNode(int x): val(x), left(nullptr), right(nullptr){}
};
```

`tree traversal`:

```
vector<int> preorde(TreeNode* root) {
     vector<int> res;
     traversal(root, res);
     return res;
}

//preorde
void traversal(TreeNode* root, vector<int>& res) {
      if (root == nullptr) return;
      res.push_back(root->val);
      traversal(root->left, res);
      traversal(root->right, res);
}

class Solution {
public:
    vector<int> preorderTraversal(TreeNode* root) {
        stack<TreeNode*> st;
        vector<int> result;
        if (root == NULL) return result;
        st.push(root);
        while (!st.empty()) {
            TreeNode* node = st.top();                       // 中
            st.pop();
            result.push_back(node->val);
            if (node->right) st.push(node->right);           // 右（空节点不入栈）
            if (node->left) st.push(node->left);             // 左（空节点不入栈）
        }
        return result;
    }
};

//inorder
void traversal(TreeNode* cur, vector<int>& vec) {
    if (cur == NULL) return;
    traversal(cur->left, vec);  // 左
    vec.push_back(cur->val);    // 中
    traversal(cur->right, vec); // 右
}
```

```
//postorder
void traversal(TreeNode* cur, vector<int>& vec) {
     if (cur == nullptr) return;
     traversal(cur->left, vec);
     traversal(cur->right, vec);
     vec.push_back(cur->val);
}
```

102. Binary Tree Level Order Traversal (BFS)

```
#include<vector>
#include<queue>
#include<iostream>
using namespace std;

vector<vector<int>> levelOrder(TreeNode* root) {
      vector<vector<int>> res;
      if (root == nullptr) return res;
      queue<TreeNode*> qe;
      qe.push(root);
      while (!qe.empty()) {
           vector<int> level;
           int n = qe.size(); //level size
           for (int i = qe.size(); i > 0; i--) {
                TreeNode* cur = qe.front();
                qe.pop();
                level.push_back(cur->val);
                if (cur->left) qe.push(cur->left);
                if (cur->right) qe.push(cur->right);
            }
           res.push_back(level);
        }
        return res;
}
```

98. Valid BST

```
bool isValidBST(TreeNode* root) {
        //node in the range
       // O(N); preorde traversal

        return valid(root, NULL, NULL);
    }

    bool valid(TreeNode* cur, TreeNode* maxNode, TreeNode* minNode) {
        if (cur == nullptr)
            return true;
        if (minNode && cur->val <= minNode->val || maxNode&& cur->val >= maxNode->val) return false;

        return valid(cur->left, cur, minNode) && valid(cur->right, maxNode, cur);

    }
```

226. Invert Binary Tree

```
TreeNode* invertBTree(TreeNode* root) {
//postorder
       if (root) {
           invertBTree(root->left);
           invertBTree(root->right);
           swap(root->left, root->right);
       }
}
```

230. Kth Smallest Element in a BST

```
class Solution {
public:
    int kthSmallest(TreeNode* root, int k) {
        return inorder(root, k);
        }
private:
    int inorder(TreeNode* root, int& k) {
        //inorder
           if (!root) return -1;
           int x = inorder(root->left, k);
           if (k == 0) return x;
           if (--k == 0) return root->val;
           return inorder(root->right,k);
    }
};
```

104.**Maximum Depth of Binary Tree**

```
class Solution {
public:
    int maxDepth(TreeNode* root) {
        if (!root) return 0;
        return max (maxDepth(root->left) + 1, maxDepth(root->right) + 1);

    }
};
```

00. Same Tree

```
class Solution {
public:
    bool isSameTree(TreeNode* p, TreeNode* q) {
        if (!p && !q) return true;
        if (!p && q || p && !q) return false;
        if (p->val != q->val) return false;
        return isSameTree(p->left, q->left) && isSameTree(p->right, q->right);
    }
};
```

572. Subtree of Another Tree

```
class Solution {
public:
    bool isSubtree(TreeNode* root, TreeNode* subRoot) {
        if (root == nullptr) return false;
        if (subRoot == nullptr) return true;
        if (isSameTree(root, subRoot)) return true;
        return isSubtree(root->left, subRoot) || isSubtree(root->right, subRoot);
    }
private:
    bool isSameTree(TreeNode* r, TreeNode* s) {
        if (r == nullptr && s == nullptr) return true;
        if (r ==nullptr || s == nullptr) return false;
        return (r->val == s->val) && (isSameTree(r->left, s->left) && isSameTree(r->right, s->right));
    }
};
```

35. Lowest Common Ancestor of a Binary Search Tree

```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
//Binary search tree condition
if (root->val > p->val && root->val > q->val)
   return lowestCommonAncestor(root->left, p, q);
else if (root->val < p->val && root->val < q->val)
   return lowestCommonAncestor(root->right, p, q);
else
   return root;
}
```