```cpp
struct TreeNode {
      int val;
      TreeNode *left;
      TreeNode *right;
      TreeNode() : val(0), left(nullptr), right(nullptr) {}
      TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
      TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
  };

struct ListNode {
      int val;
      ListNode *next;
      ListNode() : val(0), next(nullptr) {}
      ListNode(int x) : val(x), next(nullptr) {}
      ListNode(int x, ListNode *next) : val(x), next(next) {}
  };
```

95. Unique Binary Search Trees II

```cpp
class Solution {
public:
    vector<TreeNode*> generateTrees(int n) {
        if (n == 0) return {};
        return generate(1, n);
    }

    vector<TreeNode*> generate(int start, int end) {
        vector<TreeNode*> trees;

        if (start > end) {
            trees.push_back(nullptr);
            return trees;
        }

        for (int i = start; i <= end; ++i) {
            vector<TreeNode*> leftSubtrees = generate(start, i - 1);
            vector<TreeNode*> rightSubtrees = generate(i + 1, end);

            for (TreeNode* left : leftSubtrees) {
                for (TreeNode* right : rightSubtrees) {
                    TreeNode* root = new TreeNode(i);
                    root->left = left;
                    root->right = right;
                    trees.push_back(root);
                }
            }
        }
```

```
        return trees;
    }
};
```

96. Unique Binary Search Trees

```
class Solution {
public:
    int numTrees(int n) {
        if(n <= 1) return 1;
        int ans = 0;
        for(int i = 1; i <= n; i++)
            ans += numTrees(i-1) * numTrees(n-i);
        return ans;
    }
};
```

98. Validate Binary Search Tree

```
class Solution {
public:
    bool validate(TreeNode* root, long long lower, long long upper) {
        if(!root) return true;
        if(root->val<=lower||root->val>=upper) return false;
        return validate(root->left,lower,root->val) && validate(root->right,root->val,upper);
    }
    bool isValidBST(TreeNode* root) {
        return validate(root,LLONG_MIN, LLONG_MAX);
    }
};
```

99. Recover Binary Search Tree

```
class Solution {
public:
    TreeNode* first=nullptr;
    TreeNode* second=nullptr;
    TreeNode* prev=nullptr;

    void LNR(TreeNode* root) {
        if(!root) return;
        LNR(root->left);
        if(prev&&prev->val > root->val) {
            if(!first) first=prev;
            second=root;
        }
        prev=root;
        LNR(root->right);
```

```
    }
    void recoverTree(TreeNode* root) {
        LNR(root);
        if(first&&second) swap(first->val,second->val);
    }
};
```

108. Convert Sorted Array to Binary Search Tree

```
class Solution {
public:
    TreeNode* sortedArrayToBST(vector<int>& nums) {
        return buildTree(nums, 0, nums.size() - 1);
    }

    TreeNode* buildTree(const vector<int>& nums, int left, int right) {
        if (left > right) {
            return nullptr;
        }

        int mid = left + (right - left) / 2;

        TreeNode* root = new TreeNode(nums[mid]);

        root->left = buildTree(nums, left, mid - 1);
        root->right = buildTree(nums, mid + 1, right);

        return root;
    }
};
```

109. Convert Sorted List to Binary Search Tree

```
class Solution {
public:
    TreeNode* sortedListToBST(ListNode* head) {
        int size = 0;
        ListNode* temp = head;
        while (temp) {
            size++;
            temp = temp->next;
        }

        return sortedListToBSTHelper(head, 0, size - 1);
    }

private:
    TreeNode* sortedListToBSTHelper(ListNode*& head, int left, int right) {
```

```
        if (left > right) {
            return nullptr;
        }
                int mid = left + (right - left) / 2;
        TreeNode* leftChild = sortedListToBSTHelper(head, left, mid - 1);

        TreeNode* root = new TreeNode(head->val);

        head = head->next;

        root->left = leftChild;
        root->right = sortedListToBSTHelper(head, mid + 1, right);

        return root;
    }
};
```

173. Binary Search Tree Iterator

```
class BSTIterator {
private:
    stack<TreeNode*> stk;

public:
    BSTIterator(TreeNode* root) {
        // Push all the leftmost nodes onto the stack
        while (root != nullptr) {
            stk.push(root);
            root = root->left;
        }
    }

    bool hasNext() {
        return !stk.empty();
    }

    int next() {
        TreeNode* node = stk.top();
        stk.pop();

        TreeNode* rightChild = node->right;
        while (rightChild != nullptr) {
            stk.push(rightChild);
            rightChild = rightChild->left;
        }

        return node->val;
    }
```

```
};
```

230. Kth Smallest Element in a BST

```
class Solution {
private:
    int count = 0;  // Counter to track the number of visited nodes
    int kthValue = -1;  // Store the kth smallest value

public:
    int kthSmallest(TreeNode* root, int k) {
        inorder(root, k);
        return kthValue;
    }

private:
    void inorder(TreeNode* node, int k) {
        if (node == nullptr) {
            return;
        }

        inorder(node->left, k);

        count++;
        if (count == k) {
            kthValue = node->val;
            return;  // Stop once we've found the kth smallest
        }

        inorder(node->right, k);
    }
};
```

235. Lowest Common Ancestor of a Binary Search Tree

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        while (root) {
            if (p->val < root->val && q->val < root->val) {
                root = root->left;
            }
            else if (p->val > root->val && q->val > root->val) {
                root = root->right;
            }
            else {
                return root;
            }
```

```
        }
        return nullptr;
    }
};
```

2476. Closest Nodes Queries in a Binary Search Tree

```
class Solution {
public:
    vector<vector<int>> closestNodes(TreeNode* root, vector<int>& queries) {
        vector<vector<int>> answer;

        for (int q : queries) {
            int mini = findMini(root, q);
            int maxi = findMaxi(root, q);
            answer.push_back({mini, maxi});
        }

        return answer;
    }

private:
    int findMini(TreeNode* root, int target) {
        int mini = -1;
        while (root) {
            if (root->val <= target) {
                mini = root->val;  // Update mini
                root = root->right;  // Move right to find closer match
            } else {
                root = root->left;  // Move left
            }
        }
        return mini;
    }

    int findMaxi(TreeNode* root, int target) {
        int maxi = -1;
        while (root) {
            if (root->val >= target) {
                maxi = root->val;  // Update maxi
                root = root->left;              } else {
                root = root->right;  // Move right
            }
        }
        return maxi;
    }
};
```

1382. Balance a Binary Search Tree

```
class Solution {
public:
    TreeNode* balanceBST(TreeNode* root) {
        vector<int> sortedValues;
        inorderTraversal(root, sortedValues);
        return buildBalancedTree(sortedValues, 0, sortedValues.size() - 1);
    }

private:
    void inorderTraversal(TreeNode* node, vector<int>& values) {
        if (!node) return;
        inorderTraversal(node->left, values);  // Traverse left subtree
        values.push_back(node->val);           // Visit current node
        inorderTraversal(node->right, values); // Traverse right subtree
    }

    TreeNode* buildBalancedTree(const vector<int>& values, int start, int end) {
        if (start > end) return nullptr;

        int mid = start + (end - start) / 2;
        TreeNode* node = new TreeNode(values[mid]);

        node->left = buildBalancedTree(values, start, mid - 1);
        node->right = buildBalancedTree(values, mid + 1, end);

        return node;
    }
};
```

1305. All Elements in Two Binary Search Trees

```
class Solution {
public:
    vector<int> getAllElements(TreeNode* root1, TreeNode* root2) {
        vector<int> sorted1, sorted2, result;

        inorderTraversal(root1, sorted1);
        inorderTraversal(root2, sorted2);

        mergeSortedArrays(sorted1, sorted2, result);

        return result;
    }

private:
    void inorderTraversal(TreeNode* node, vector<int>& result) {
```

```
        if (!node) return;
        inorderTraversal(node->left, result); // Traverse left subtree
        result.push_back(node->val);         // Visit current node
        inorderTraversal(node->right, result); // Traverse right subtree
    }

    void mergeSortedArrays(const vector<int>& arr1, const vector<int>& arr2, vector<int>&
result) {
        int i = 0, j = 0;

        while (i < arr1.size() && j < arr2.size()) {
            if (arr1[i] <= arr2[j]) {
                result.push_back(arr1[i++]);
            } else {
                result.push_back(arr2[j++]);
            }
        }

        while (i < arr1.size()) {
            result.push_back(arr1[i++]);
        }

        while (j < arr2.size()) {
            result.push_back(arr2[j++]);
        }
    }
};
```

1038. Binary Search Tree to Greater Sum Tree

```
class Solution {
public:
    TreeNode* bstToGst(TreeNode* root) {
        int sum = 0;
        reverseInOrder(root, sum);
        return root;
    }

private:
    void reverseInOrder(TreeNode* node, int& sum) {
        if (!node) return;

        reverseInOrder(node->right, sum);

        sum += node->val;
        node->val = sum;

        reverseInOrder(node->left, sum);
```

```
    }
};
```

1008. Construct Binary Search Tree from Preorder Traversal

```
class Solution {
public:
    TreeNode* bstFromPreorder(vector<int>& preorder) {
        int index = 0;
        return buildTree(preorder, index, INT_MIN, INT_MAX);
    }

private:
    TreeNode* buildTree(vector<int>& preorder, int& index, int lower, int upper) {
        if (index >= preorder.size() || preorder[index] < lower || preorder[index] > upper) {
            return nullptr;
        }

        int val = preorder[index++];
        TreeNode* root = new TreeNode(val);

        root->left = buildTree(preorder, index, lower, val);

        root->right = buildTree(preorder, index, val, upper);

        return root;
    }
};
```

669. Trim a Binary Search Tree

```
class Solution {
public:
    TreeNode* trimBST(TreeNode* root, int low, int high) {
        if (!root) return nullptr;

        if (root->val < low) {
            return trimBST(root->right, low, high);
        }

        if (root->val > high) {
            return trimBST(root->left, low, high);
        }

        root->left = trimBST(root->left, low, high);
        root->right = trimBST(root->right, low, high);

        return root;
```

```
    }
};
```

538. Convert BST to Greater Tree

```
class Solution {
public:
    int sum = 0;

    TreeNode* convertBST(TreeNode* root) {
        if (!root) return nullptr;

        convertBST(root->right);

        sum += root->val;
        root->val = sum;

        // Process the left subtree
        convertBST(root->left);

        return root;
    }
};
```

449. Serialize and Deserialize BST

```
class Codec {
public:    void preorder(TreeNode* root, ostringstream& out) {
        if (!root) return;
        out << root->val << " ";
        preorder(root->left, out);
        preorder(root->right, out);
    }

    string serialize(TreeNode* root) {
        ostringstream out;
        preorder(root, out);
        return out.str();
    }

    TreeNode* buildTree(queue<int>& preorder, int min, int max) {
        if (preorder.empty() || preorder.front() < min || preorder.front() > max) {
            return nullptr;
        }
        int val = preorder.front();
        preorder.pop();
        TreeNode* root = new TreeNode(val);
        root->left = buildTree(preorder, min, val);  // Left subtree
```

```
        root->right = buildTree(preorder, val, max); // Right subtree
        return root;
    }
    TreeNode* deserialize(string data) {
        if (data.empty()) return nullptr;

        istringstream in(data);
        queue<int> preorder;
        int val;
        while (in >> val) {
            preorder.push(val);
        }

        return buildTree(preorder, INT_MIN, INT_MAX);
    }
};
```