Binary Search Tree:-

1️⃣ Introduction to Binary Search Tree (TUF – Easy)

Problem Description:

- BST ek tree data structure hai jisme:
    1. Left subtree me **sab node root se chhote** hote hain
    2. Right subtree me **sab node root se bade** hote hain
- Duplicate nodes **usually allowed nahi hote**.
- BST ke operations fast hote hain: Search, Insert, Delete → `O(log n)` on average.

Approach / Concept:

- Har node ke left/right subtree me BST property follow hoti hai
- Recursive ya iterative traversal se operations perform hote hain.

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

Key Notes / Tips:

- Inorder traversal → sorted array
- BST se Binary Tree me easily convert kar sakte ho aur vice-versa.

2️⃣ Search in a Binary Search Tree (LeetCode – Easy)

Problem Description:

- Given a BST and a value `val`, check if the value exists.
- Return node if exists, else return NULL.

Approach:

- Compare val with root:
    - `val == root.val` → found
    - `val < root.val` → search left
    - `val > root.val` → search right

```
TreeNode* searchBST(TreeNode* root, int val) {
    if(!root) return nullptr;
    if(root->val == val) return root;
    if(val < root->val) return searchBST(root->left, val);
    return searchBST(root->right, val);
}
```

Key Notes:

- Recursive or iterative dono approach work karte hain
- Time Complexity: `O(h)` → height of tree

3️⃣ Find Min / Max in BST (HackerRank – Basic)

Problem Description:

- Find **smallest** and **largest** value in BST.

Approach:

- Min → keep going left till NULL
- Max → keep going right till NULL

```
int findMin(TreeNode* root) {
    while(root->left) root = root->left;
    return root->val;
}

int findMax(TreeNode* root) {
    while(root->right) root = root->right;
```

```
    return root→val;
}
```

Key Notes:

- Always traverse left for min, right for max
- O(h) time

4️⃣ Ceil in a BST (TUF – Medium)

Problem Description:

- Ceil of `X` = **smallest element ≥ X** in BST.

Approach:

- If root→val < X → ceil in right subtree
- Else → ceil could be root or in left subtree

```
int ceilBST(TreeNode* root, int X) {
    if(!root) return -1;
    if(root→val == X) return X;
    if(root→val < X) return ceilBST(root→right, X);
    int left = ceilBST(root→left, X);
    return (left >= X) ? left : root→val;
}
```

Key Notes:

- Recursive or iterative
- Mirror logic for **Floor**

5️⃣ Floor in a BST (TUF – Medium)

Problem Description:

- Floor of `X` = **largest element ≤ X** in BST.

Approach:

- If root→val > X → floor in left subtree
- Else → floor could be root or in right subtree

```
int floorBST(TreeNode* root, int X) {
    if(!root) return -1;
    if(root→val == X) return X;
    if(root→val > X) return floorBST(root→left, X);
    int right = floorBST(root→right, X);
    return (right <= X && right != -1) ? right : root→val;
}
```

Key Notes:

- Ceil and Floor ka logic symmetric hai

6️⃣ Insert a Given Node in BST (LeetCode – Medium)

Problem Description:

- Given a BST and a value `val`, insert it into BST.
- Return the **root** after insertion.
- BST property must be maintained.

Approach:

- Compare val with current node:
    - `val < root.val` → insert in left subtree
    - `val > root.val` → insert in right subtree
- Recursion continues till NULL node is reached → create new node

```
TreeNode* insertBST(TreeNode* root, int val) {
    if(!root) return new TreeNode(val);
    if(val < root→val) root→left = insertBST(root→left, val);
    else root→right = insertBST(root→right, val);
```

```
    return root;
}
```

Iterative Version (Optional):

```
TreeNode* insertBSTIter(TreeNode* root, int val) {
    TreeNode* node = new TreeNode(val);
    if(!root) return node;
    TreeNode* curr = root;
    TreeNode* parent = nullptr;
    while(curr) {
        parent = curr;
        if(val < curr->val) curr = curr->left;
        else curr = curr->right;
    }
    if(val < parent->val) parent->left = node;
    else parent->right = node;
    return root;
}
```

Key Notes:

- Recursive is clean and short
- Iterative avoids recursion stack overhead
- Time Complexity: O(h)

7️⃣ Delete a Node in BST (LeetCode – Medium)

Problem Description:

- Delete node with value `key` from BST.
- Return the new root.

Approach:

1. Find node to delete (recursive search).
2. Three cases:

- **Leaf Node:** Just delete it
- **One Child:** Replace node with child
- **Two Children:**
    - Find **inorder successor** (smallest in right subtree)
    - Copy successor's value to current node
    - Delete successor node recursively

```
TreeNode* deleteNode(TreeNode* root, int key) {
    if(!root) return nullptr;

    if(key < root→val) root→left = deleteNode(root→left, key);
    else if(key > root→val) root→right = deleteNode(root→right, key);
    else {
        // Node found
        if(!root→left) return root→right;
        if(!root→right) return root→left;

        // Two children
        TreeNode* succ = root→right;
        while(succ→left) succ = succ→left;
        root→val = succ→val;
        root→right = deleteNode(root→right, succ→val);
    }
    return root;
}
```

Key Notes:

- **Inorder successor** ensures BST property after deletion
- Time Complexity: O(h)
- Edge cases: Deleting root, leaf, or node with one child

8️⃣ Find K-th smallest/largest element in BST (LeetCode – Medium)

Problem Description:

- Find K-th smallest or largest element in BST

Approach:

- Inorder traversal → sorted array
- K-th smallest = k-th element in inorder
- K-th largest = reverse inorder → k-th element

```
void inorder(TreeNode* root, vector<int>& nums) {
    if(!root) return;
    inorder(root->left, nums);
    nums.push_back(root->val);
    inorder(root->right, nums);
}

int kthSmallest(TreeNode* root, int k) {
    vector<int> nums;
    inorder(root, nums);
    return nums[k-1];
}

int kthLargest(TreeNode* root, int k) {
    vector<int> nums;
    inorder(root, nums);
    return nums[nums.size()-k];
}
```

9️⃣ Check if a tree is BST or BT (LeetCode – Medium)

Problem Description:

- Verify if binary tree satisfies BST property

Approach:

- Recursive bounds check (min/max)

```
bool isBST(TreeNode* root, long minVal = LONG_MIN, long maxVal = LONG_M
AX) {
    if(!root) return true;
    if(root→val <= minVal || root→val >= maxVal) return false;
    return isBST(root→left, minVal, root→val) &&
           isBST(root→right, root→val, maxVal);
}
```

🔟 LCA in BST (LeetCode – Medium)

Problem Description:

- Lowest Common Ancestor of two nodes `p` and `q`

Approach:

- BST property → move left/right based on `p` and `q` values

```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode
* q) {
    if(!root) return nullptr;
    if(p→val < root→val && q→val < root→val) return lowestCommonAncestor(r
oot→left, p, q);
    if(p→val > root→val && q→val > root→val) return lowestCommonAncestor(r
oot→right, p, q);
    return root;
}
```

1️⃣ Construct a BST from Preorder Traversal (LeetCode – Medium)

Problem Description:

- Build BST from preorder array

Approach:

- Recursive + bounds method

```
TreeNode* constructBST(vector<int>& preorder, int& idx, int minVal, int maxVa
l) {
    if(idx >= preorder.size()) return nullptr;
    int val = preorder[idx];
    if(val < minVal || val > maxVal) return nullptr;

    TreeNode* root = new TreeNode(val);
    idx++;
    root->left = constructBST(preorder, idx, minVal, val);
    root->right = constructBST(preorder, idx, val, maxVal);
    return root;
}

TreeNode* bstFromPreorder(vector<int>& preorder) {
    int idx = 0;
    return constructBST(preorder, idx, INT_MIN, INT_MAX);
}
```

1️⃣2️⃣ Inorder Successor / Predecessor in BST (LeetCode – Medium)

Problem Description:

- Successor → smallest value > node
- Predecessor → largest value < node

Approach:

- Successor: Go right → leftmost node
- Predecessor: Go left → rightmost node

```
TreeNode* inorderSuccessor(TreeNode* root, TreeNode* p) {
    TreeNode* succ = nullptr;
```

```
    while(root) {
        if(p→val < root→val) { succ = root; root = root→left; }
        else root = root→right;
    }
    return succ;
}
```

1️⃣3️⃣ Merge 2 BSTs (LeetCode – Medium)

Problem Description:

- Merge two BSTs into one BST

Approach:

1. Inorder traversal both → sorted arrays
2. Merge arrays → sorted array
3. Build BST from sorted array

```
TreeNode* buildBST(vector<int>& nums, int l, int r) {
    if(l > r) return nullptr;
    int mid = l + (r-l)/2;
    TreeNode* root = new TreeNode(nums[mid]);
    root→left = buildBST(nums, l, mid-1);
    root→right = buildBST(nums, mid+1, r);
    return root;
}
```

1️⃣4️⃣ Two Sum in BST (LeetCode – Easy)

Problem Description:

- Check if BST has pair with sum K

Approach:

- Use HashSet / Inorder + Two Pointer

```
bool findTarget(TreeNode* root, int k) {
    vector<int> nums;
    inorder(root, nums);
    int i=0, j=nums.size()-1;
    while(i<j){
        int sum = nums[i]+nums[j];
        if(sum==k) return true;
        else if(sum<k) i++;
        else j--;
    }
    return false;
}
```

1️⃣5️⃣ Recover BST (LeetCode – Medium)

Problem Description:

- Two nodes swapped, fix BST

Approach:

- Inorder traversal → find two misplaced nodes → swap values

```
TreeNode* first = nullptr, *second = nullptr, *prev = nullptr;

void recover(TreeNode* root){
    if(!root) return;
    recover(root→left);
    if(prev && root→val < prev→val){
        if(!first) first = prev;
        second = root;
    }
    prev = root;
    recover(root→right);
}
```

```
void recoverTree(TreeNode* root){
    recover(root);
    swap(first→val, second→val);
}
```

1️⃣6️⃣ Largest BST in Binary Tree

Problem Description:

- Find largest BST in a binary tree (not necessarily BST)

Approach:

- Use **post-order traversal**
- Track subtree size, min, max, isBST
- Update largest BST size

```
struct Info{
    int size;
    int minVal;
    int maxVal;
    bool isBST;
};

Info largestBST(TreeNode* root, int &maxSize){
    if(!root) return {0, INT_MAX, INT_MIN, true};
    auto l = largestBST(root→left, maxSize);
    auto r = largestBST(root→right, maxSize);
    Info curr;
    curr.isBST = l.isBST && r.isBST && root→val > l.maxVal && root→val < r.min
Val;
    if(curr.isBST){
        curr.size = l.size + r.size + 1;
        curr.minVal = min(root→val, l.minVal);
        curr.maxVal = max(root→val, r.maxVal);
        maxSize = max(maxSize, curr.size);
```

```
    } else curr.size = 0;
    return curr;
}
```