# Binary Search Trees

## Data Structures and Algorithms Tutorial

IIITH - S26

Ordered Trees, Search, Insert, Delete

# Agenda

1. Overview
2. Operations
3. Practice Problems
4. Homework
5. Summary

# Overview

# What is a BST?

Definition

- Binary tree with ordering property
- Left subtree: all values < node's value
- Right subtree: all values > node's value

Key Property

- Inorder traversal of BST = sorted order!

Why BST?

- Efficient search: $O(h)$ where $h$ is height
- Balanced BST: $O(\log n)$ for search, insert, delete
- Dynamic: can grow/shrink unlike arrays

# Operations

# Search in BST

Main Idea

- Compare target with current node
- If equal: found!
- If smaller: search left
- If larger: search right
- If NULL: not found

Intuition

- At each step, eliminate half the remaining tree
- Like binary search on array, but on tree structure
- Worst case: $O(h)$, best case $O(\log n)$ for balanced

# Search: Implementation

```
Node* search(Node* root, int key) {
    if (root == NULL || root->data == key)
        return root;

    if (key < root->data)
        return search(root->left, key);

    return search(root->right, key);
}
```

Time: $O(h)$ | Space: $O(h)$ for recursion

- For balanced BST: $O(\log n)$
- For skewed BST: $O(n)$

# Insert in BST

Main Idea

- Find the correct position (like search)
- When we reach NULL, insert new node there
- Always insert at leaf level!

Algorithm

1. If tree is empty, return new node
2. If key < current: go left, insert in left subtree
3. If key > current: go right, insert in right subtree
4. Return (unchanged) root

Duplicates: Usually not allowed (or go to right by convention)

# Insert: Implementation

```
Node* insert(Node* root, int key) {
    if (root == NULL) return createNode(key);

    if (key < root->data)
        root->left = insert(root->left, key);
    else if (key > root->data)
        root->right = insert(root->right, key);

    return root;
}
```

Time: $O(h)$ | Space: $O(h)$

- Same complexity as search (follows same path)

# Practice Problems

# Problem 1: Search in BST

Problem Statement Given BST root and a value, find node with that value. Return subtree rooted at that node (or NULL if not found).

Intuition

- This is literally BST search!
- We're just returning the node instead of true/false
- Follow the same logic: compare and go left or right

Edge Cases

- Empty tree
- Value not in tree
- Value is root

## Solution

```
Node* searchBST(Node* root, int val) {
    if (root == NULL) return NULL;
    if (root->data == val) return root;

    if (val < root->data)
        return searchBST(root->left, val);
    else
        return searchBST(root->right, val);
}
```

Time: $O(h)$ | Space: $O(h)$

# Problem 2: Validate BST

Problem Statement Check if given binary tree is a valid BST.

Intuition

- At each node, we need to know the valid range
- Root: can be any value (-infinity, +infinity)
- Go left: max becomes current node's value
- Go right: min becomes current node's value

Example

- At root 8: range is (-inf, +inf)
- Go left to 3: range becomes (-inf, 8)
- Go right to 10: range becomes (8, +inf)

## Solution

```
int isValid(Node* root, long min, long max) {
    if (root == NULL) return 1;
    if (root->data <= min || root->data >= max) return 0;
    return isValid(root->left, min, root->data) &&
           isValid(root->right, root->data, max);
}
```

Time: $O(n)$ | Space: $O(h)$

- Make sure the initial call is isValid(root, LONG_MIN, LONG_MAX) so as to handle edge cases :D
- Must visit every node (could be anywhere)

# Problem 3: LCA in BST

Problem Statement Find Lowest Common Ancestor of two nodes in a BST.

Intuition

- LCA is the node where paths to p and q diverge
- In BST, we can use ordering property!
- If both p and q are smaller: LCA is in left subtree
- If both p and q are larger: LCA is in right subtree
- Otherwise: current node is the LCA (one left, one right)

## Solution

```
Node* LCA(Node* root, int p, int q) {
    if (root == NULL) return NULL;
    if (p < root->data && q < root->data)
        return LCA(root->left, p, q);

    if (p > root->data && q > root->data)
        return LCA(root->right, p, q);

    return root;
}
```

Time: $O(h)$ | Space: $O(h)$

- Faster than regular binary tree LCA -> We only traverse one path instead of both subtrees.

# Homework

# Homework: Delete in BST

Problem Statement Delete a node with given key from BST. Return root of modified tree.

Three Cases

1. Leaf node: Just delete it (return NULL to parent)
2. One child: Replace node with its child
3. Two children: Find inorder successor (or predecessor), replace, delete successor

Hint

- Inorder successor = smallest node in right subtree
- Go right once, then keep going left

# Summary

# Summary: BST Operations

| Operation | Time | Space |
|---|---|---|
| Search | $O(h)$ | $O(h)$ |
| Insert | $O(h)$ | $O(h)$ |
| Delete | $O(h)$ | $O(h)$ |
| Validate | $O(n)$ | $O(h)$ |

Where $h$ = height. For balanced BST: $h = O(\log n)$

# When to Use BST?

Use BST when:

- Data frequently changes (insert/delete)
- Need ordered data with range queries
- Want to find closest/smallest/larger elements

Remember

- BST property: left < node < right
- Inorder = sorted order
- Balance matters! Unbalanced BST = linked list
- For guaranteed $O(\log n)$, use balanced variants (AVL, Red-Black)

# Questions?

Thank you!