CSC211 · Algorithms and Data Structures

# 20. BST: Search and Insertion

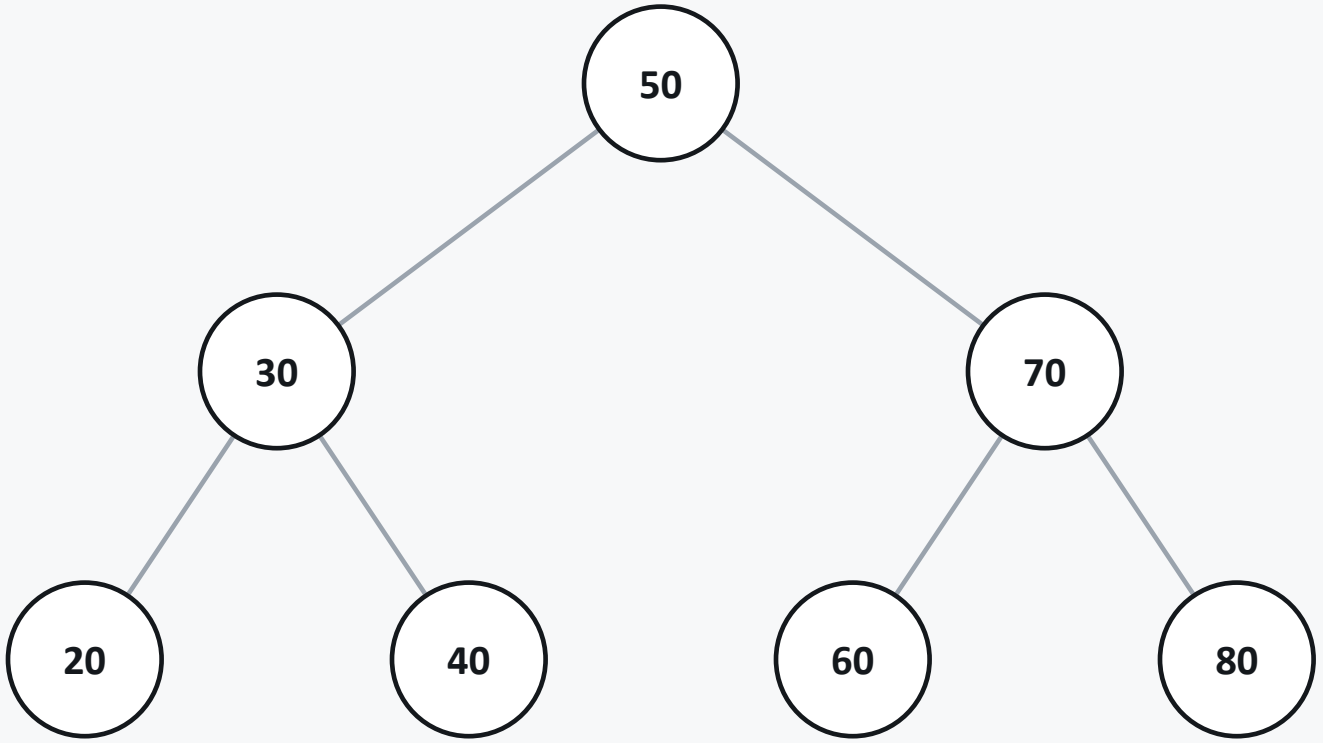
One ordering rule turns a plain binary tree into a fast-searchable structure

COMSATS University Islamabad, Lahore Campus

# Agenda

1. The BST property, and how it differs from a plain binary tree
2. Searching a BST, both recursively and iteratively
3. Inserting into a BST while preserving the property
4. Finding the minimum and maximum values
5. The complexity of every BST operation — and why shape matters
6. Balanced vs. skewed: a real, measured search-speed comparison

# The BST Property




**For every node:**

- **Left subtree** holds only smaller values
- **Right subtree** holds only larger values

30, 20, 40 < 50 < 60, 70, 80 — and the same rule holds recursively at every node (20 < 30 < 40; 60 < 70 < 80), not just the root.

Every value left of 50 is smaller; every value right is larger.

# Binary Tree vs. BST

## Plain Binary Tree

?

Makes no promise about where values sit — shape only, no order.

Search must potentially check both subtrees at every node.

## Binary Search Tree

✓

Adds one ordering constraint on top of the plain binary tree shape.

A single comparison tells you which one subtree the target must be in.

**Every BST is a binary tree — the ordering promise is the entire source of a BST's value, since it's what makes fast search possible at all.**

# Searching: One Comparison Picks the Side

**recursive**

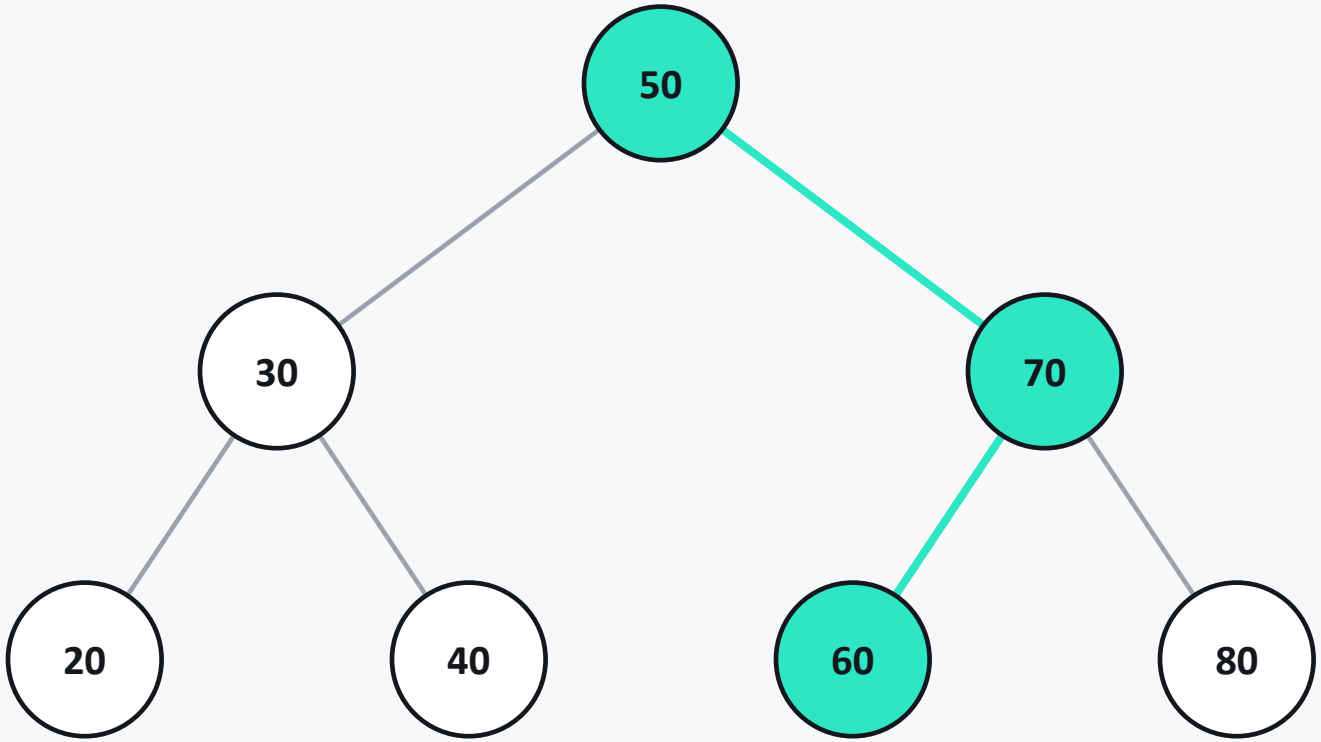
```
bool search(TreeNode* node, int target) {
    if (node == nullptr) return false;
    if (target == node->data) return true;
    if (target < node->data)
        return search(node->left, target);
    return search(node->right, target);
}
```

**iterative**

```
bool search(TreeNode* root, int target) {
    TreeNode* cur = root;
    while (cur != nullptr) {
        if (target == cur->data) return true;
        cur = (target < cur->data)
              ? cur->left : cur->right;
    }
    return false;
}
```

**Both do exactly the same comparisons in exactly the same order.** The iterative version avoids call overhead and can't stack-overflow on a very deep tree — which is why real library implementations are usually iterative.

# Tracing a Search: Find 60




**Comparison trace**

```
60 vs 50  →  60 > 50, go right
60 vs 70  →  60 < 70, go left
60 vs 60  →  match — found!
```

3 comparisons for a 7-node tree — one per level walked, never all 7 nodes.

Because of the BST property, search never checks both subtrees — the comparison at each node tells you the single subtree that must hold the target.

# Insertion: Same Logic, Walk to an Empty Spot

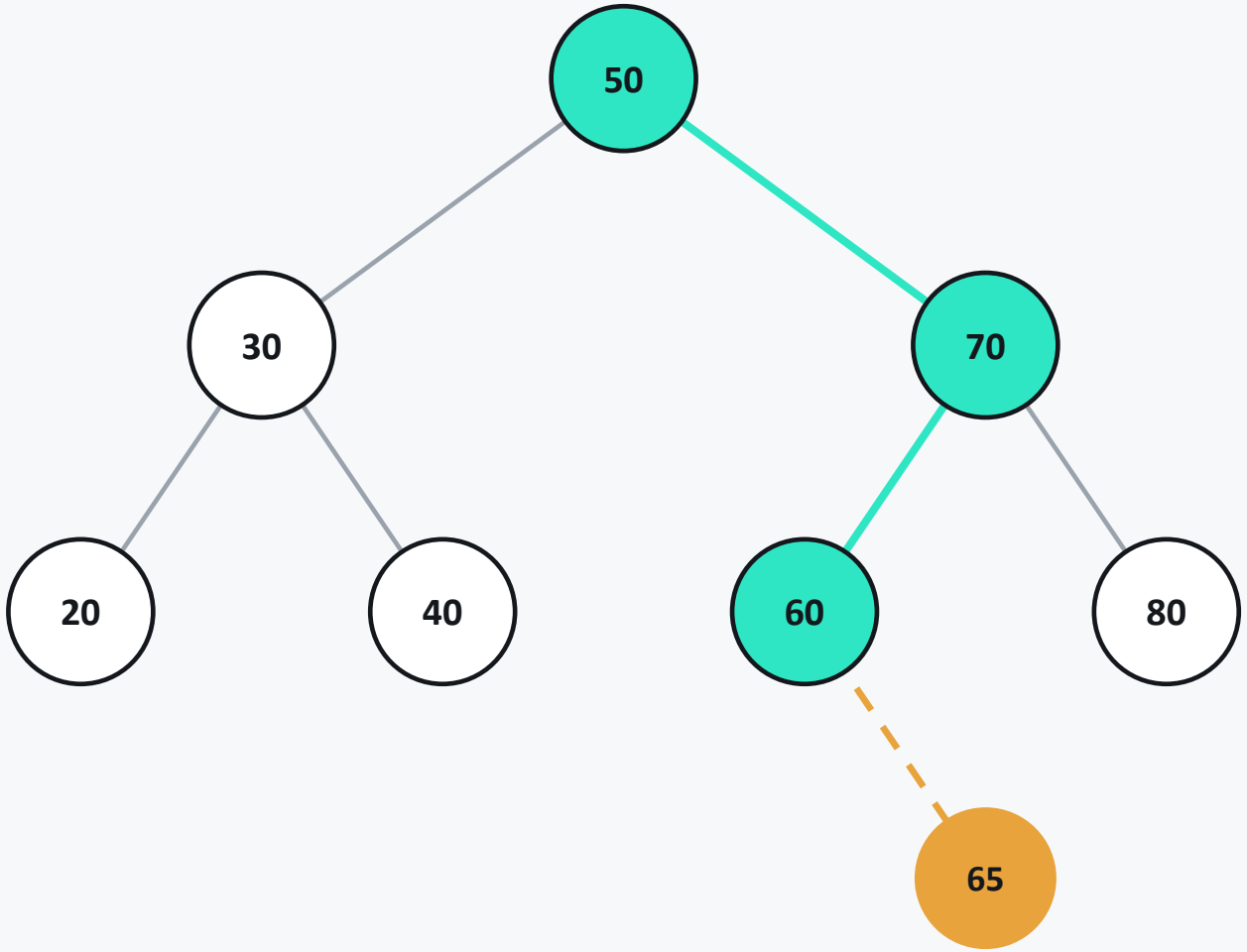
bst.cpp — BST::insertHelper

```
TreeNode* insertHelper(TreeNode* node, int value) {
    if (node == nullptr) return new TreeNode(value);
    if (value < node->data)
        node->left = insertHelper(node->left, value);
    else if (value > node->data)
        node->right = insertHelper(node->right, value);
    return node;
}
```

**Insertion follows the exact same "which side does it belong on?" logic as search,** walking down until it finds an empty spot (nullptr) — then places the new node there.

No duplicates: if value == node->data, the recursion returns node unchanged — nothing is inserted.

# Tracing an Insert: Add 65

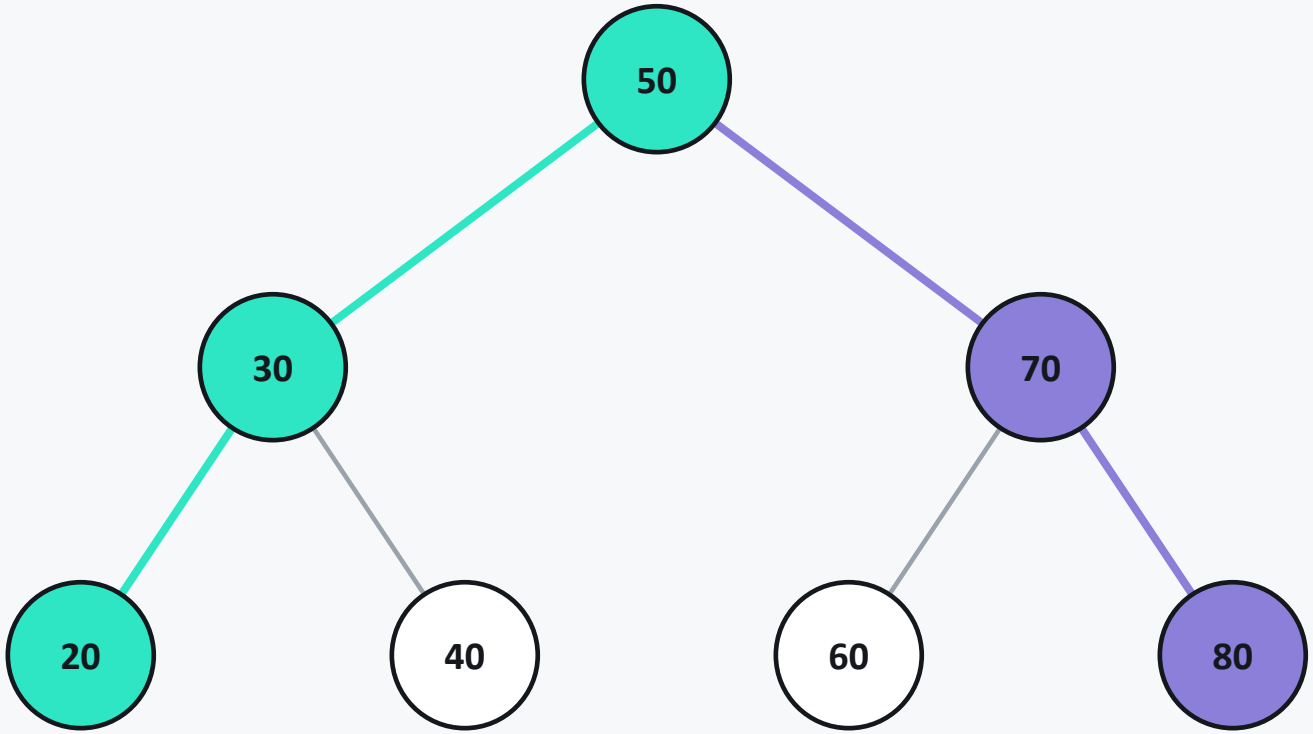



65 is a new right child of 60 (dashed = newly inserted)

**Comparison trace**

```
65 vs 50 → right
65 vs 70 → left
65 vs 60 → right
nullptr → insert here
```

# Minimum and Maximum: No Comparisons Needed




```
findMin() / findMax()
while (cur->left) cur = cur->left;
return cur->data;  // minimum
while (cur->right) cur = cur->right;
return cur->data;  // maximum
```

The BST property gives a direct shortcut: no need to check every value, just follow one pointer chain to the end.

teal = follow left to minimum · purple = follow right to maximum

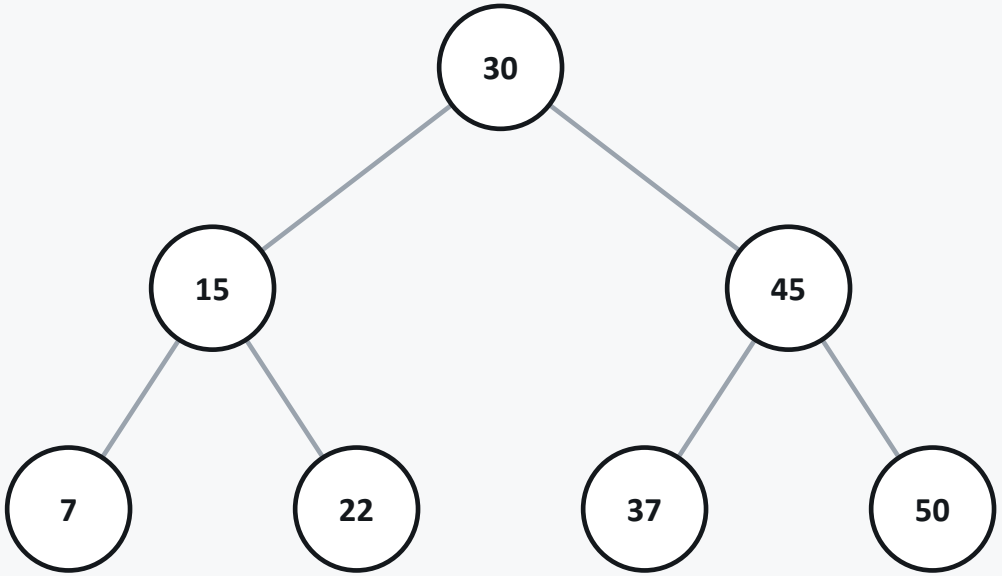
# Complexity: Everything Depends on Height

| Operation | Balanced BST | Skewed BST (worst case) |
|---|---|---|
| Search | O(log n) | O(n) |
| Insert | O(log n) | O(n) |
| Find min / max | O(log n) | O(n) |

**A BST's speed depends entirely on its shape**

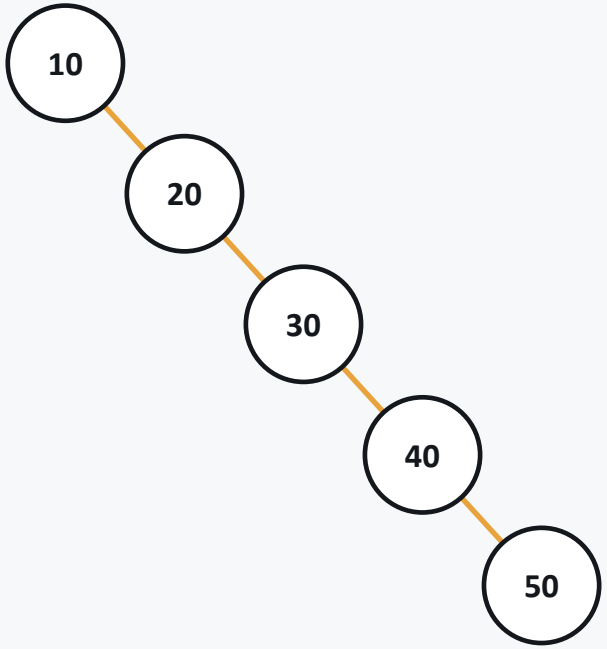
Insert 10, 20, 30, 40, 50 in already-sorted order and every node ends up with only a right child — a completely skewed tree, structurally identical to a linked list, with search degrading to O(n). Nothing guarantees balance; Lecture 22's AVL tree actively rebalances after every insertion to fix this.

# Same-ish Values, Wildly Different Shapes




Balanced order: 30, 15, 45, 7, 22, 37, 50

**2**
height, 7 nodes




Sorted-order insertion: 10, 20, 30, 40, 50

**4**
height, only 5 nodes

7 nodes fit in height 2 for the balanced tree; the skewed tree needs height 4 for only 5 — “linked list wearing a tree's clothing.”

# Measured, Not Guessed: Search Speed

bst_timing.cpp (excerpt)

```
// Balanced: insert the MIDDLE of the range first,
// recursively — keeps height ~ log2(n).
insertBalanced(root, values, 0, N-1);

// Skewed: insert already-sorted values in order.
for (int v : values) skewedRoot =
    insertNode(skewedRoot, v);

// Then: search for every value, 20 passes, N = 6000.
```

**Search verdict**

balanced tree

**FASTER**

over 120,000 total searches
(6,000 values × 20 passes, both trees)

All 6,000 values found in both trees — correctness is identical.
Shape alone makes the difference.

**A skewed BST with n nodes has height n−1,** so searching it degrades to the same O(n) walk as a linked list — the BST property only constrains order, never shape.

# Applications of BST

### Fast, ordered lookup tables

Anywhere you need both fast search and sorted retrieval (via in-order traversal, Lecture 19).

### Sets and maps

C++'s std::set and std::map are typically backed by a self-balancing BST.

### Range queries

“Find all values between X and Y” can skip entire subtrees outside the range.

# Key Takeaways

- The BST property — left subtree smaller, right subtree larger, at every node — is the one rule that makes fast search possible.
- Search and insertion both work by repeatedly choosing left or right based on a single comparison, walking one path from the root.
- Minimum and maximum are found directly by following left or right pointers to the end — no need to check every value.
- Insertion order, not just the set of values, determines shape: already-sorted input produces a skewed tree, height n-1.
- A BST is O(log n) only if it stays roughly balanced — a skewed BST degrades to O(n), exactly what Lecture 22's AVL tree exists to prevent.