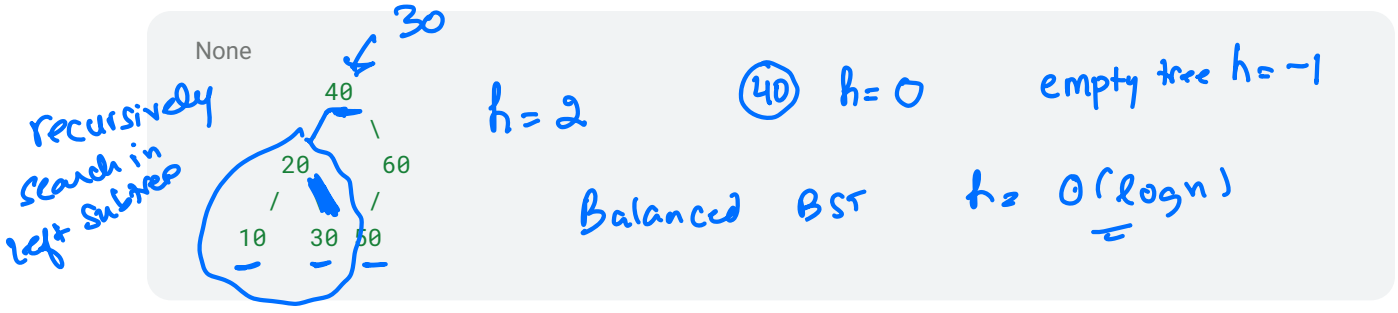
Lecture 7: Review of BST and Big O analysis

Definition: The **height** of a tree is the length of the longest path (number of edges) from the root to a leaf node.
A single-node tree has height **0**. An empty tree has height **-1**.

Definition: A balanced BST is a BST whose height = O(log n), where n is the number of keys in the BST. Here we are thinking of height as a function of n, so h(n) = O(log n)

Problem 1: Big-O Analysis of BST Operations

Setup: Put this BST on the board:



Q: What is the height of this tree?

Part A: Trace search

- Walk through searching for **key 30**. Write the path followed from root to 30.

40 → 20 → 30 → found it!

- How is the running time of search related to the number of nodes visited?

$n_{visited}$: number of nodes in the path to search.
$T(n) = O(n_{visited})$

- How is the number of nodes visited related to height of the BST(h)?

$\max(n_{visited}) = h + 1$

$T(n) = O(h+1) = O(h)$

Part B: Connecting height to Big-O

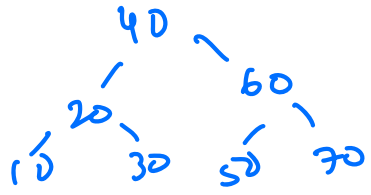
Q: What is the worst-case running time of search in terms of h (height)?

O(h)

Q: For a BST with n nodes, what are the best and worst possible heights?

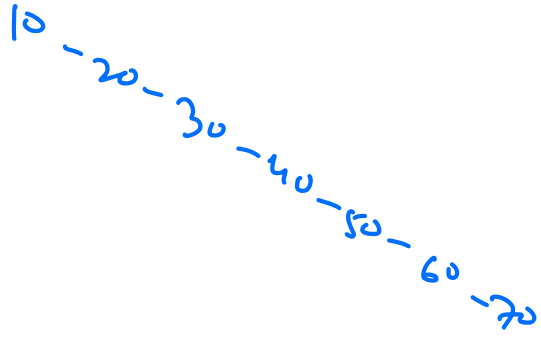
Draw best case and worst case with keys 10, 20, 30, 40, 50, 60, 70

Best case - balanced tree e.g.



T(n) = O(log n)

Worst case skewed tree. e.g.



T(n) = O(n)

Takeaway: BST operations (search, insert, min, max) are all O(h). The height depends on insertion order. Worst case is O(n), best case is O(log n).

Problem 2: Writing a Recursive BST Function

Problem: Check if a binary tree is a valid BST

Approach 1: Check immediate children (naive)

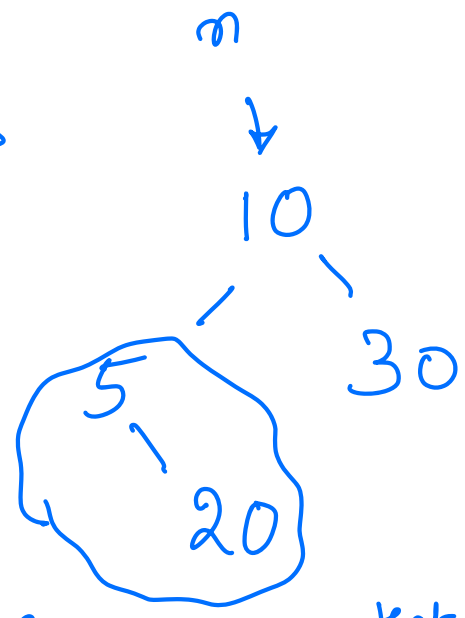
First instinct: At each node, just check that left child < node < right child.

```
None
isBST(node):
    if node is null, return true
    if left child exists and left->data >= node->data, return false
    if right child exists and right->data <= node->data, return false
    return isBST(left) AND isBST(right)
```

Does this work? Why or Why not?

empty tree is a BST? Yes

One node tree is a BST? Yes



Counter example that shows flaw in algorithm for approach 1.

Approach 2: Apply the full definition (bottom-up)

```
None
isBST(node):
    if node is null, return true
O(n)  if max key in left subtree > node, return false
+ O(n)  if min key in right subtree < node, return false
    return isBST(left) AND isBST(right)
```

In class we noted that we cannot apply the BST max algo since tree may not be a BST

Defn of BST:

for any node n

keys($T_L(n)$) < key(n) < keys($T_R(n)$)

Running time of preorder traversal → O(n)
finding the min & max at each node adds O(n) per node traversed. So overall T(n) = $O(n^2)$

Other approaches: ③ Write the keys into a vector using an inorder traversal. Take a second pass through the vector to check whether keys are monotonically increasing (sorted smallest to largest). If yes, tree is a BST otherwise tree is not a BST.

Time complexity = $O(n)$ + $O(n)$ = $O(n)$
(Inorder) (Pass through vector)

Space complexity = $O(n)$ (Needs extra space for vector)

④ Yet another approach: Modify an inorder traversal to check for a valid BST in a single pass. Need to keep track of pointer to prev node in an inorder traversal & check that value of current node > value of previous node in an inorder traversal

Time complexity $T(n) = O(n)$

Space complexity $S(n) = O(1)$

Next Approach: Flip the direction — push constraints down (top-down)

What extra parameters do I need? A `min` and `max` defining the allowed range.

How do they change? Trace the ranges on a valid BST:

Which traversal? Preorder

Key insight: when recursing left, update max to be value of node, when recursing right, update min to be value of node.

Write the code

```C/C++
class bst {
 private:
    struct Node {
        int data;
        Node* left;
        Node* right;
    };
    Node* root;
    //Add helper here


 public:
    bool isBST() const;
};
```

Link to LeetCode problem:
https://leetcode.com/problems/validate-binary-search-tree/description/

Helper needs `(node, min, max)`. Check node against range, recurse with tighter bounds:

```C/C++
bool bst::isBST() const {
   //TO DO



}
```

Trace to verify

Problem: Return the k-th smallest key in the BST (1-indexed).

We need nodes in sorted order to know the k-th position → **inorder**. We pass a counter by reference that increments each time we visit a node. Once it hits k, we're done.

```C/C++
void findKthSmallest(Node* r, int k, int& count, int& result) const {
    if (!r) return;

    findKthSmallest(r->left, k, count, result); // left first (smaller
keys)

    count++;                          // visit current node
    if (count == k) {
        result = r->data;
        return;
    }

    findKthSmallest(r->right, k, count, result); // right (larger keys)
}
```

Takeaway: Ask yourself — does the answer at this node depend on what's **above** it (constraints from ancestors → preorder) or what's **below** it (results from children → postorder)?