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**.

Problem 1: Big-O Analysis of BST Operations (~12 min)

Setup: Put this BST on the board:

```None
        40
       /   \
     20     60
    /  \   /
  10   30 50
```

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

Part B: Connecting height to Big-O

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

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

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

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?

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

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

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:

```None
       40
     /    \
    20     60
   / \     /
  10 30  50
```

Which traversal?

Write the code

```C/C++
class bst {
 private:
    struct Node {
        int data;
        Node* left;
        Node* right;
    };
    Node* root;
    bool isBST(Node* r, int min, int max) const;
 public:
    bool isBST() const;
};
```

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

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




}

bool bst::isBST(Node* r, int min, int max) const {














}
```

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)?