In this lab, you will implement various methods on binary search trees (BSTs). Assume the BST is defined using the following structure:

```
struct Node {
    int key;
    Node* left;
    Node* right;
    Node(int k) : key(k), left(nullptr), right(nullptr) {}
};
```

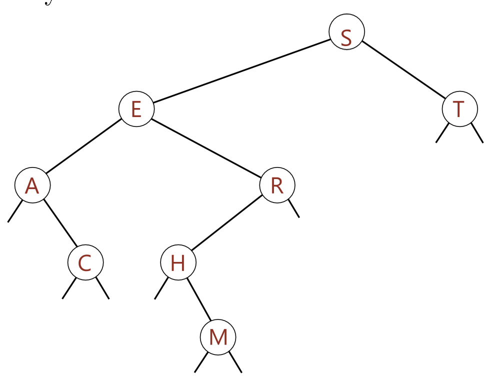
Exercise 1: *Level order traversal of a BST.* . . . .

(a) Given a level-order traversal sequence of a binary search tree, reconstruct the original tree and return its root.

Hint: To construct a BST from level-order of its keys, you can use the following approaches.

1. **[50% credit]** Insert keys into the BST in the order given by the level-order sequence. The resulting tree is the unique BST corresponding to that sequence. The time complexity is $O(n \times h)$, where n is the number of keys and h is the height of the tree. In the worst case, h can be $O(n)$, leading to a time complexity of $O(n^2)$.
2. **[for full credit]** Use a queue to keep track of leaf nodes and their valid key ranges. Each node in a BST is valid only if its key lies within a certain range $[min, max]$ Root starts with range $(-\infty, \infty)$ For each node, we check the next unprocessed key in the level-order sequence:
   - If it fits in the valid range, create the child.
   - Push it into the queue with its updated valid range:
     - Left child: $(min, node.key)$
     - Right child: $(node.key, max)$

   This way, every key is placed once in its correct position without recursive searching. That gives us $O(n)$ time complexity.



level-order traversal: S E T A R C H M

(b) Given a binary tree, return the level order traversal of its nodes' keys. (i.e., from left to right, level by level).

Hint: Use a queue to keep track of nodes at the current level.

Exercise 2: *Recursive methods on binary trees.* . . . .

Given a binary search tree (BST), implement the following recursive methods.

(a) `height()` – max depth of the tree

(b) `sizeOdd()` – number of Nodes with an odd key

(c) `isPerfectlyBalancedH` – at each Node, do left and right subtrees have same height?

(d) `isSemiBalancedI` – is each Node semibalanced? that is, either a leaf or else size(larger child) / size (smaller child) ≤ 2

(e) `sizeAtDepth` – number of nodes at a given depth

(f) `sizeAboveDepth` – number of nodes whose depth is $<$ a given depth

(g) `sizeBelowDepth` – number of nodes whose depth is $>$ a given depth

To get you started, here is an example of `size` method that count the number of nodes in the tree:

```
int size(Node* root) {
    if (root == nullptr) {
        return 0;
    }
    return 1 + size(root->left) + size(root->right);
}
```