Week 4 Worked Companion

Worked solutions

104	102	98	230

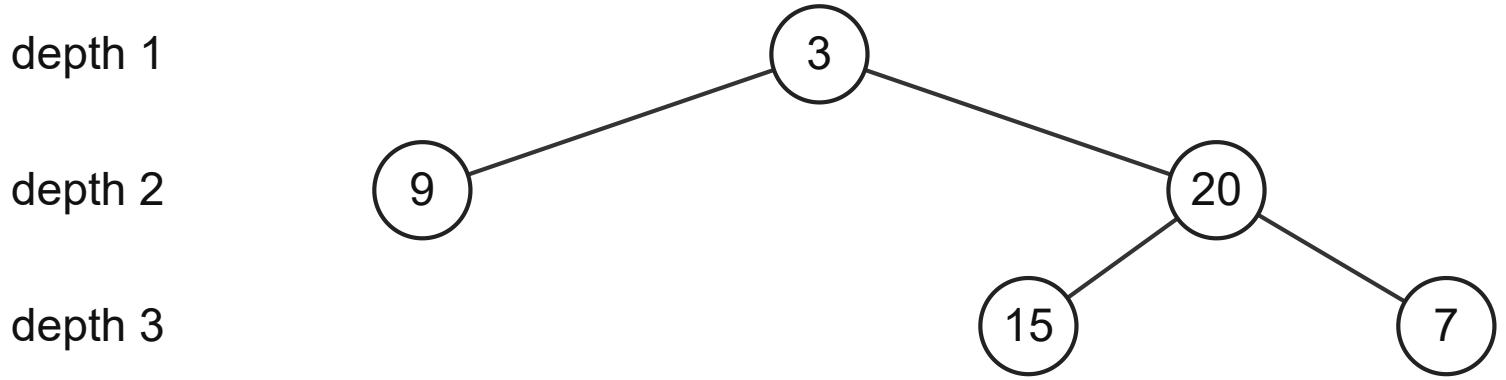
A task tree has at most two subtasks per node. What is its deepest level? We also need a report grouped by level and a check that a search tree preserves its ordering rules.

Each node can branch in two directions. Recursive return values, stacks, queues, and ancestor bounds carry the information needed for these questions.

Tree height, width, and storage

A node stores a value and two child references. Missing children are None. Depth counts nodes from the root; height h is maximum depth, and width w is the largest number of nodes on one level.

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val, self.left, self.right = val, left, right
```



Here n=5, h=3, and w=2. DFS retains unfinished paths in a stack; BFS retains pending levels in a queue. Trees with equal node counts can therefore have different peak storage requirements.

From list notation to a tree

`[4,2,7,None,3]` is level-order notation. Read the left and right child slots of each real node; None means that slot is empty. Do not read child slots for None itself.

Values read	Connection	Nodes awaiting child slots
4	create the root	[4]
2,7	4.left=2, 4.right=7	[2,7]
None,3	2.left=None, 2.right=3	[7,3]

Omitted trailing child slots are None. The path 4→2→3 has depth 3 and the levels are `[[4],[2,7],[3]]`. The depth function receives the connected TreeNode root, rather than this list directly.

In this notation for a sparse tree, children of list position i are not always at 2i+1 and 2i+2. In `[1,None,2,3]`, node 3 is the left child of node 2.

What a recursive depth call returns

```
def depth_recursive(root):
    if root is None:
        return 0
    return 1 + max(depth_recursive(root.left),
                   depth_recursive(root.right))
```

Current node	Left return	Right return	Current return
Each leaf: 9,15,7	0	0	1
20	1 (from 15)	1 (from 7)	2
3	1 (from 9)	2 (from 20)	3

Each call has its own local root variable. A parent remains active while waiting for child results. Every node is visited once, giving O(n) time, but up to h calls wait simultaneously, giving O(h) auxiliary space.

Python limits recursion depth. The following implementation retains depth information in an explicit stack to handle long chains.

104 · Idea and execution trace

LeetCode 104: Maximum Depth of Binary Tree · Easy

Return the maximum root-to-leaf path length measured in nodes.

Depth counts nodes from the root to the current node. Store `(node, depth)` together so depth need not be reconstructed. Each child is one level deeper; the largest encountered depth is the answer.

Tree: root 3; children 9 and 20; children of 20 are 15 and 7. Stack top is on the right.

Popped (value,depth)	Stack after child pushes	best
(3,1)	[(9,2),(20,2)]	1
(20,2)	[(9,2),(15,3),(7,3)]	2
(7,3)	[(9,2),(15,3)]	3
(15,3)	[(9,2)]	3
(9,2)	[]	3

104 · Turning the idea into code

```
def max_depth(root):
    if root is None:
        return 0
    stack, best = [(root, 1)], 0
    while stack:
        node, depth = stack.pop()
        best = max(best, depth)
        if node.left:
            stack.append((node.left, depth + 1))
        if node.right:
            stack.append((node.right, depth + 1))
    return best
```

Start the root at depth 1. Update best after popping, then push children at `depth+1`. The right child is pushed last and therefore visited first; this does not change the maximum.

An empty tree has depth 0. The recurrence `1 + max(left depth, right depth)` expresses the same calculation; the explicit stack avoids Python recursion-depth limits on long chains.

104 · Why it works; time and memory

Structural induction proves the recurrence. The explicit stack carries the true root depth to every node and records its maximum.

Every tree node is pushed once by its unique parent. There are n pushes and pops, plus two child checks per node, so time is O(n). The two if statements do not add another size-dependent loop.

For height h, at most one pending sibling branch per level accumulates, giving O(h) stack space, at worst O(n). A one-child chain keeps only one pending entry in this iterative version, whereas recursive calls accumulate to depth h.

Python recursion depth may be below a 10^4-node chain; distinguish the small-tree recursive demonstration from the full-contract iterative version.

```
root=[4,2,7,None,3] → 3
```

102 · Idea and execution trace

LeetCode 102: Binary Tree Level Order Traversal · Medium

Return tree values grouped by depth, left to right.

A first-in, first-out queue visits nodes by depth. Freeze its length at the start of each level so children appended during that level remain for the next one.

Use the preceding 3–9–20–15–7 tree. The queue's left end is removed next.

Queue at level start	Frozen length	Current level	Next queue
[3]	1	[3]	[9,20]
[9,20]	2	[9,20]	[15,7]
[15,7]	2	[15,7]	[]

The result is `[[3],[9,20],[15,7]]`.

At the second level, queue=[9,20] fixes the loop count at two. Removing 9 leaves [20]; removing 20 and adding its children leaves [15,7]. Although the queue is nonempty, those two iterations are complete, so level=[9,20] is stored before the next level starts.

102 · Turning the idea into code

```
from collections import deque

def level_order(root):
    if root is None:
        return []
    queue, result = deque([root]), []
    while queue:
        level = []
        for _ in range(len(queue)):
            node = queue.popleft()
            level.append(node.val)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        result.append(level)
    return result
```

`range(len(queue))` fixes the iteration count when the for loop begins; later appends do not increase it. Allocate a new level list each round and append it to result after that level finishes.

Use `deque.popleft()` for removal at the front. A list's `pop(0)` would shift the remaining references.

102 · Why it works; time and memory

At each outer-loop entry the queue contains one level. Processing its initial size leaves only the next level; left-first insertion preserves order.

The outer loop counts levels, but the inner-loop counts sum to exactly n nodes across all levels. Constant-time queue insertion and removal give O(n) time. Using list.pop(0) on wide levels can instead cause O(n^2) total shifting.

If maximum level width is w, the queue holds the rest of one level plus part of the next, still O(w) entries. Queue workspace is O(w); returned level values occupy O(n). Including output, extra space is O(n).

list.pop(0) shifts the remaining entries; repeated use can reach O(n^2) on wide trees.

```
root=[4,2,7,None,3] → [[4],[2,7],[3]]
```

98 · Idea and execution trace

LeetCode 98: Validate Binary Search Tree · Medium

Decide whether all descendants satisfy strict BST ordering.

Comparing only parent and child misses constraints inherited from ancestors. With root 5, every value in the right subtree must exceed 5. Pass an allowed interval `(lower, upper)` to summarize all ancestral constraints.

Check a tree with root 5, right child 8, and 8's left child 4.

Node value	Allowed interval	Test result
5	(-∞,∞)	Pass
8	(5,∞)	Pass
4	(5,8)	5 < 4 is false → False

Although 4 is smaller than its parent 8, it cannot lie to the right of root 5.

98 · Turning the idea into code

```
def valid_bst(root):
    stack = [(root, float("-inf"), float("inf"))]
    while stack:
        node, lower, upper = stack.pop()
        if node is None:
            continue
        if not lower < node.val < upper:
            return False
        stack.append((node.right, node.val, upper))
        stack.append((node.left, lower, node.val))
    return True
```

For a right child, raise lower to the current value and retain upper. For a left child, lower upper to the current value and retain lower.

The strict test `lower < node.val < upper` implements a BST contract that excludes duplicates. Skip None entries because they are not nodes.

98 · Why it works; time and memory

The interval is the intersection of all ancestor constraints. Tightening it preserves that meaning; any violation invalidates the whole tree.

A valid tree requires processing n nodes and a fixed number of empty-child entries per node. Two bound comparisons and stack operations per node give O(n) time. Early failure can shorten a run but does not change the worst-case bound.

Each stack entry contains a node reference and two bounds, a constant-size record. At most O(h) entries remain pending, so auxiliary space is O(h), at worst O(n). Adding bounds does not make each record grow with the tree.

The example’s 6 is below its parent 10 but lies right of root 8; parent-only checks miss it.

```
root=[8,3,10,None,None,6,12] → False
```

230 · Idea and execution trace

LeetCode 230: Kth Smallest Element in a BST · Medium

Return the kth smallest BST value.

In a BST, visiting the left subtree, node, then right subtree yields increasing values. Stop at the kth visit rather than collecting and sorting every value.

Root 3, left child 1 with right child 2, and right child 4. k=2; stack top is on the right.

Action	stack	Remaining k	Visited values
Push left path 3,1	[3,1]	2	None
Pop 1; move to its right, 2	[3]	1	1
Push 2	[3,2]	1	1
Pop 2	[3]	0	1,2 → return 2

230 · Turning the idea into code

```
def kth_smallest(root, k):
    stack, node = [], root
    while stack or node:
        while node:
            stack.append(node)
            node = node.left
        node = stack.pop()
        k -= 1
        if k == 0:
            return node.val
        node = node.right
    raise ValueError("k exceeds tree size")
```

The inner loop saves an unvisited left path. A node is visited when popped, so decrement k at that moment. Then repeat the leftmost descent in its right subtree.

The stack retains ancestors whose right portions remain to be processed. The input must be a valid BST with k between 1 and the node count.

230 · Why it works; time and memory

Every left value is smaller and every right value larger. Induction makes inorder sorted, so the kth visit gives the answer.

The initial left descent pushes at most h nodes. The algorithm pops k nodes; total pushes can be charged to popped nodes plus the at most h entries still pending. Time is O(h+k), becoming O(n) for k=n.

At most h ancestors occupy the stack, so auxiliary space is O(h). No full traversal array is stored. A balanced tree has h=O(log n), but a skew tree can have h=O(n).

If materializing inorder, index at k-1. That alternative uses O(n) auxiliary space and time.

```
root=[5,2,8,1,3], k=3 → 3
```

Midpoint checks: answers and reasoning

1. `[4,2,7,None,3]`: depth 3, levels `[[4],[2,7],[3]]`. The path 4,2,3 contains three nodes.
2. `[8,3,10,None,None,6,12]`: invalid. The value 6 is in root 8's right subtree and must exceed 8, even though it is smaller than parent 10.
3. BST `[5,2,8,1,3]`: k=1,3,5 gives 1,3,8. Inorder visits `1,2,3,5,8`.

An empty tree returns depth 0 and levels `[]`. A single node has depth 1. Replacing a strict bound with an inclusive bound incorrectly accepts duplicate BST keys.

Exercises and sources

LeetCode · 104, 102, 98, 230 · Worked solutions · Python ZIP

```
python demo_07.py
python demo_08.py
python verify.py
```

After implementing the student file, check it with `python verify.py --student`.

Erik Demaine, Srini Devadas. MIT 6.006, Introduction to Algorithms (2011).