Tree Data Structure Mega Cheat Sheet

1. Tree Concepts & Terminology

- **Node**: Individual element of a tree containing a value and references to child nodes.
- **Root**: The topmost node; only entry point.
- **Leaf**: Node with no children.
- **Child/Parent**: Relationships between levels.
- **Sibling**: Nodes with the same parent.
- **Subtree**: A child node and all its descendants.
- **Depth**: Edges from the root to a node.
- **Height**: Edges from a node to the farthest leaf.
- **Edge**: Link between two nodes.

2. Types of Trees

- **Binary Tree**: Each node has ≤2 children (left, right).
- **Binary Search Tree (BST)**: Binary tree; left < root < right for all nodes (ordering property).
- **Balanced Tree**: Subtrees' heights differ by at most 1 (e.g., AVL, Red-Black).
- **Complete Binary Tree**: All levels filled, last level filled left to right.
- **Perfect Binary Tree**: All internal nodes have two children; all leaves at the same level.
- **N-ary Tree**: Any number of children per node; generalized tree.

3. Tree Node Definition (Python Example)

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

4. Tree Traversals

A. Depth-First Search (DFS) Traversals

Preorder (Root, Left, Right):

```
def preorder(root):
    if not root: return []
    return [root.val] + preorder(root.left) + preorder(root.right)
```

Inorder (Left, Root, Right): *(BST property: returns sorted values)*

```
def inorder(root):
    if not root: return []
    return inorder(root.left) + [root.val] + inorder(root.right)
```

Postorder (Left, Right, Root):

```
def postorder(root):
    if not root: return []
    return postorder(root.left) + postorder(root.right) + [root.val]
```

B. Breadth-First Search (BFS: Level Order Traversal)

```
from collections import deque

def level_order(root):
    if not root: return []
    res, queue = [], 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)
        res.append(level)
    return res
```

5. Core Interview Patterns and Solutions

A. Height/Max Depth

```
def max_depth(root):
    if not root: return 0
    return 1 + max(max_depth(root.left), max_depth(root.right))
```

B. Minimum Depth

```
def min_depth(root):
    if not root: return 0
    if not root.left and not root.right: return 1
    min_d = float('inf')
    if root.left: min_d = min(min_d, min_depth(root.left))
    if root.right: min_d = min(min_d, min_depth(root.right))
    return min_d + 1
```

C. Diameter of Binary Tree

```
def diameter(root):
    def dfs(node):
        if not node: return 0
        left = dfs(node.left)
        right = dfs(node.right)
        nonlocal dia
        dia = max(dia, left + right)
        return 1 + max(left, right)
    dia = 0
    dfs(root)
    return dia
```

D. Balanced Binary Tree Check

```
def is_balanced(root):
    def depth(node):
        if not node: return 0
        l, r = depth(node.left), depth(node.right)
        if l == -1 or r == -1 or abs(l - r) > 1:
            return -1
```

```
        return 1 + max(l, r)
    return depth(root) != -1
```

E. Same/Symmetric Tree Checks

```
def is_same(p, q):
    if not p and not q: return True
    if not p or not q or p.val != q.val: return False
    return is_same(p.left, q.left) and is_same(p.right, q.right)

def is_symmetric(root):
    def check(l, r):
        if not l and not r: return True
        if not l or not r or l.val != r.val: return False
        return check(l.left, r.right) and check(l.right, r.left)
    return not root or check(root.left, root.right)
```

F. Invert (Mirror) Binary Tree

```
def invert(root):
    if not root: return None
    root.left, root.right = invert(root.right), invert(root.left)
    return root
```

6. Path Patterns

- **All root-to-leaf paths:**

```
def paths(root):
    res = []
    def dfs(node, path):
        if not node: return
        path.append(node.val)
        if not node.left and not node.right:
            res.append(list(path))
        else:
            dfs(node.left, path)
            dfs(node.right, path)
        path.pop()
```

```
    dfs(root, [])
    return res
```

Has path sum (root-to-leaf):

```
def has_path_sum(root, s):
    if not root: return False
    if not root.left and not root.right: return root.val == s
    return (has_path_sum(root.left, s - root.val) or
            has_path_sum(root.right, s - root.val))
```

Maximum Path Sum (any-to-any):

```
def max_path_sum(root):
    def dfs(node):
        nonlocal mx
        if not node: return 0
        l, r = max(dfs(node.left), 0), max(dfs(node.right), 0)
        mx = max(mx, node.val + l + r)
        return node.val + max(l, r)
    mx = float('-inf')
    dfs(root)
    return mx
```

7. Level Patterns

- **Zigzag Level Order:**

```
def zigzag_level_order(root):
    if not root: return []
    result, queue, left = [], deque([root]), True
    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)
        if not left: level.reverse()
```

```
        left = not left
        result.append(level)
    return result
```

- **Right Side View:**

```
def right_view(root):
    if not root: return []
    queue, result = deque([root]), []
    while queue:
        for i in range(len(queue)):
            node = queue.popleft()
            if node.left: queue.append(node.left)
            if node.right: queue.append(node.right)
            if i == 0: result.append(node.val)
    return result
```

8. BST Patterns

- **Validate BST:**

```
def is_valid_bst(root):
    def helper(node, lo, hi):
        if not node: return True
        if not lo < node.val < hi: return False
        return helper(node.left, lo, node.val) and helper(node.right, node.val, hi)
    return helper(root, float('-inf'), float('inf'))
```

- **Search/Insert/Delete:**

```
def search_bst(root, val):
    if not root or root.val == val: return root
    if val < root.val: return search_bst(root.left, val)
    else: return search_bst(root.right, val)

def insert_bst(root, val):
    if not root: return TreeNode(val)
    if val < root.val:
        root.left = insert_bst(root.left, val)
    else:
```

```
        root.right = insert_bst(root.right, val)
    return root

def delete_bst(root, key):
    if not root: return None
    if key < root.val:
        root.left = delete_bst(root.left, key)
    elif key > root.val:
        root.right = delete_bst(root.right, key)
    else:
        if not root.left: return root.right
        if not root.right: return root.left
        t = root.right
        while t.left: t = t.left
        root.val = t.val
        root.right = delete_bst(root.right, root.val)
    return root
```

- **Kth Smallest:**

```
def kth_smallest(root, k):
    def inorder(node):
        if not node: return []
        return inorder(node.left) + [node.val] + inorder(node.right)
    return inorder(root)[k-1]
```

9. Lowest Common Ancestor (LCA)

- **Binary Tree:**

```
def lca(root, p, q):
    if not root or root == p or root == q: return root
    left, right = lca(root.left, p, q), lca(root.right, p, q)
    return root if left and right else left or right
```

- **BST:**

```
def lca_bst(root, p, q):
    while root:
        if p.val < root.val and q.val < root.val:
```

```
            root = root.left
        elif p.val > root.val and q.val > root.val:
            root = root.right
        else:
            return root
```

10. Serialize/Deserialize (Rec/DFS)

```
def serialize(root):
    vals = []
    def preorder(node):
        if not node:
            vals.append('#')
        else:
            vals.append(str(node.val))
            preorder(node.left)
            preorder(node.right)
    preorder(root)
    return ','.join(vals)

def deserialize(data):
    vals = iter(data.split(','))
    def build():
        val = next(vals)
        if val == '#': return None
        node = TreeNode(int(val))
        node.left = build()
        node.right = build()
        return node
    return build()
```

11. General Interview & Practical Tips

- **Always check for `None` roots**
- **Handle both iterative & recursive patterns**
- **Practice constructing/destructing trees from traversals**
- For BFS: use `deque` for efficient queue pops

- **Write helper functions** for modular code (especially in recursion)

12. Resources for Mastery

- Practice: LeetCode ("Tree" tag), GeeksforGeeks, NeetCode patterns
- Visualizations: visualgo.net/en/bst
- Revisit all standard traversal patterns and their iterative/recursive forms
- For BST: Inorder traversal is your friend!
- For DP/advanced trees: Focus on "bottom up" recursion for optimal substructure problems