Trees Cheatsheet

Topic Overview

Trees in Java are hierarchical structures with nodes, used for organizing data efficiently. This cheatsheet covers tree-based techniques.

Prerequisites

Linked List

List of Subtopics

- Binary Tree
- Binary Search Tree (BST)
- AVL Tree
- Preorder Traversal
- Inorder Traversal
- Postorder Traversal
- Level Order Traversal
- Lowest Common Ancestor (LCA)
- Tree Height
- Validate BST

Key Concepts Explained

- **Binary Tree**: Each node has at most two children, left and right.
- **BST**: Left child $<$ node $<$ right child, enabling efficient search.
- **AVL Tree**: Self-balancing BST to maintain O(log n) height.

Approaches to Solve Problems with Step-by-Step Algorithms

- **Binary Tree**:
  - **Algorithm**:
    1. Define a node with data, left, and right pointers.
    2. Initialize root as null for an empty tree.
    3. Recursively or iteratively process nodes.
  - **Context**: O(n) for traversal, O(h) for search where h is height.
- **Binary Search Tree (BST)**:
  - **Algorithm**:
    1. Insert by comparing with root, go left if smaller, right if larger.
    2. Search by following the BST property until node or null.
    3. Delete by handling leaf, one child, or two children cases.
  - **Context**: O(log n) average, O(n) worst case.
- **AVL Tree**:
  - **Algorithm**:
    1. Insert as in BST, then check balance factor (height difference).
    2. Perform rotations (left, right, left-right) if imbalance occurs.
    3. Update heights after each operation.
  - **Context**: O(log n) time, O(n) space.
- **Preorder Traversal**:
  - **Algorithm**:
    1. Visit root, recursively traverse left subtree, then right.
  - **Context**: O(n) time, used for copying trees.
- **Inorder Traversal**:
  - **Algorithm**:
    1. Recursively traverse left subtree, visit root, then right.
  - **Context**: O(n) time, sorts BST values.
- **Postorder Traversal**:
  - **Algorithm**:
    1. Recursively traverse left, then right, visit root.
  - **Context**: O(n) time, used for deletion.
- **Level Order Traversal**:

  - **Algorithm**:
    1. Use a queue, enqueue root.
    2. Dequeue node, process, enqueue children.
    3. Repeat until queue is empty.
  - **Context**: O(n) time, O(w) space.

- **Lowest Common Ancestor (LCA)**:
  - **Algorithm**:
    1. If root is one node, return root.
    2. Recursively find LCA in left and right subtrees.
    3. If both nodes in different subtrees, root is LCA.
  - **Context**: O(n) time, O(h) space.

- **Tree Height**:
  - **Algorithm**:
    1. If node is null, return 0.
    2. Recursively get height of left and right subtrees.
    3. Return max(height(left), height(right)) + 1.
  - **Context**: O(n) time, O(h) space.

- **Validate BST**:
  - **Algorithm**:
    1. Use inorder traversal, check if values are strictly increasing.
    2. Alternatively, recursively check if left $<$ root $<$ right.
  - **Context**: O(n) time, O(h) space.

Common LeetCode Problems with Approaches

- **Invert Binary Tree (226)**: Swap left and right children recursively.
- **Validate Binary Search Tree (98)**: Check BST property with bounds.
- **Lowest Common Ancestor of BST (235)**: Use BST property for efficient LCA.
- **Maximum Depth of Binary Tree (104)**: Use height calculation.
-
- **Symmetric Tree (101)**: Check mirror symmetry with recursion.

Time & Space Complexities

- Traversal: O(n)
- Search/Insert/Delete (BST): O(log n) average, O(n) worst
- Space: O(h) for recursion, O(w) for level order

Important Tips & Tricks

- Use recursion for tree traversals.
- Balance trees to maintain log n height.
- Use queues for level-based processing.
- Handle null nodes in recursive calls.
- Optimize space with iterative methods.