The Quest for the Enchanted Forest

In a digital realm once thriving with order and harmony, an ancient forest of data structures now lies in disarray. The enchanted forest—home to Binary Trees, Binary Search Trees (BSTs), and Balanced BSTs—has grown wild and unpredictable. To restore balance and unlock its hidden secrets, a band of Code Explorers embarks on a quest. They must implement 12 methods spanning three critical domains:

- **4 Methods for Binary Trees**
- **4 Methods for Binary Search Trees**
- **4 Methods for Balanced Binary Search Trees**

Each method is designed to challenge your algorithmic and data structural prowess, pushing you to explore innovative traversals, order statistics, augmentation, and tree re-balancing techniques. In addition, each method comes with a suite of test cases and detailed explanations to ensure thorough understanding and validation.

The Story

The Digital Forest Awakens

In the heart of an ancient digital realm, a vast forest of tree data structures flourished. These structures—each with their own secrets—had long sustained the kingdom of CodeLand. Over time, however, a mysterious imbalance caused the forest to wither; its branches twisted into chaotic paths, and the once-ordered groves of BSTs became scattered. The elders of CodeLand prophesied that only skilled Code Explorers could restore the lost equilibrium by mastering advanced tree methodologies.

Part I: The Mysterious Binary Trees

Our journey begins in the deep groves of the Binary Tree. These ancient trees hide many secrets in their expansive canopies and shadowed roots. The Code Explorers are tasked with unlocking these mysteries through four advanced methods:

1. Zigzag Traversal (Spiral Order Traversal)

Objective:

Traverse the tree in a mesmerizing spiral, alternating direction at each level. This path mimics the winding trails through dense undergrowth, revealing hidden clearings in alternate orders.

Test Cases & Explanations:

- **Test Case 1: Empty Tree**
  - *Input:* `null`
  - *Expected Output:* `[]`
  - *Explanation:* Validates that the method correctly handles an empty tree without errors.
- **Test Case 2: Single Node Tree**
  - *Input:* A tree consisting of a single node (e.g., Node with value `1`).
  - *Expected Output:* `[[1]]`
  - *Explanation:* Confirms that a minimal tree returns a correct one-level output.
- **Test Case 3: Multi-Level Tree**
  - *Input:*

```
      1
    / \
   2   3
  / \   \
 4   5   6
```

  - *Expected Output:* `[[1], [3, 2], [4, 5, 6]]`
  - *Explanation:* Checks the alternating order—the first level left-to-right, the second level right-to-left, and so on.
- **Test Case 4: Unbalanced Tree**
  - *Input:* A tree where one branch is deeper than the other.
  - *Expected Output:* Depends on structure; ensures zigzag order is maintained regardless of imbalance.
  - *Explanation:* Ensures that the algorithm does not assume a complete tree and handles missing nodes properly.

2. Maximum Width

Objective:
Determine the busiest layer in the forest by measuring the maximum number of nodes at any level.

Test Cases & Explanations:

- **Test Case 1: Empty Tree**
  - *Input:* `null`
  - *Expected Output:* `0`
  - *Explanation:* Verifies that an empty tree returns a width of zero.
- **Test Case 2: Single Node Tree**
  - *Input:* A tree with only the root node (value `1`).
  - *Expected Output:* `1`
  - *Explanation:* Confirms that the width of a minimal tree is counted correctly.
- **Test Case 3: Perfectly Balanced Tree**
  - *Input:*

```
      1
    /   \
   2     3
  / \   / \
 4   5 6   7
```

  - *Expected Output:* `4`
  - *Explanation:* Level 3 has four nodes. This test validates that the method accurately counts nodes on each level.
- **Test Case 4: Unbalanced Tree**
  - *Input:* A tree where one level is wider than others.
  - *Expected Output:* The width of the level with the maximum nodes.
  - *Explanation:* Confirms that the method dynamically detects the widest level even if the tree is skewed.

3. Diameter of the Tree

Objective:
Calculate the diameter of the tree—the length (in terms of number of nodes or edges) of the longest path between any two nodes.

Test Cases & Explanations:

- **Test Case 1: Empty Tree**
  - *Input:* `null`
  - *Expected Output:* `0` or `-1` (depending on your definition)
  - *Explanation:* Ensures the method handles an empty structure appropriately.
- **Test Case 2: Single Node Tree**
  - *Input:* A single node.
  - *Expected Output:* `1` (if counting nodes) or `0` (if counting edges)
  - *Explanation:* Validates the base scenario where no path exists beyond the single node.
- **Test Case 3: Balanced Tree**
  - *Input:*

```
    1
   / \
  2   3
 / \
4   5
```

  - *Expected Output:* Depending on whether counting nodes or edges, it might be `4` nodes or `3` edges.
  - *Explanation:* Tests standard behavior where the longest path runs from one leaf node, through the root, to another leaf.
- **Test Case 4: Unbalanced Tree**
  - *Input:* A tree where one branch is significantly longer.
  - *Expected Output:* Reflects the true longest path in the tree.
  - *Explanation:* Confirms that the algorithm can handle trees with uneven depths, correctly computing the diameter across disparate lengths.

4. Lowest Common Ancestor (LCA)

Objective:
Trace back the lineage of two distinct paths until they converge at a common ancestor—the sacred origin from which the forest sprang.

Test Cases & Explanations:

- **Test Case 1: Both Nodes are Same**

- *Input:* Tree with a single node (value `1`) and both search values as `1`.
- *Expected Output:* Node with value `1`.
- *Explanation:* Checks that the method correctly returns the node when both target values are the same.

- **Test Case 2: Simple Binary Tree**
  - *Input:*

```
      3
     / \
    5   1
   / \
  6   2
```

    Finding LCA for nodes `6` and `2`.
  - *Expected Output:* `5`
  - *Explanation:* Validates that the method finds the first common ancestor that splits the paths.

- **Test Case 3: One Node is Ancestor of the Other**
  - *Input:* Same as above, finding LCA for nodes `5` and `6`.
  - *Expected Output:* `5`
  - *Explanation:* Ensures the method handles cases when one target is an ancestor of the other.

- **Test Case 4: Nodes in Different Subtrees**
  - *Input:* A larger tree where nodes are in completely separate branches.
  - *Expected Output:* The common ancestor higher up in the tree.
  - *Explanation:* Demonstrates the algorithm's ability to backtrack across divergent branches.

Part II: The Secrets of the Binary Search Trees (BSTs)

Delving deeper into the forest, the Code Explorers discover a mystical grove where every branch is in perfect order—the Binary Search Trees. Hidden treasures lie in wait for those who can decipher the grove's secrets:

5. kth Smallest Element

Objective:
Seek out the rare artifact hidden in numerical order. By identifying the kth smallest element, explorers claim a prized relic of intrinsic value and historical significance.

Test Cases & Explanations:

- **Test Case 1: k Exceeds Tree Size**
  - *Input:* A BST with nodes {2, 3, 5} and k = 5.
  - *Expected Output:* Error or a special value indicating an invalid request.
  - *Explanation:* Checks the method's ability to handle k values that exceed the number of nodes.
- **Test Case 2: Single Node Tree**
  - *Input:* A BST with one node (value `10`) and k = 1.
  - *Expected Output:* `10`
  - *Explanation:* Verifies the algorithm works correctly for minimal input.
- **Test Case 3: Regular BST**
  - *Input:*

    Request: kth smallest where k = 3.
  - *Expected Output:* `4` (given the in-order sequence is [2, 3, 4, 5, 7, 8]).
  - *Explanation:* Tests that the method accurately counts and returns the kth smallest element.
- **Test Case 4: BST with Duplicate Values (if allowed)**
  - *Input:* A BST where duplicates are permitted and sorted in a defined order.
  - *Expected Output:* The kth element as per the duplicate ordering rules.
  - *Explanation:* Ensures the method correctly handles trees with duplicate entries.

6. Inorder Successor

Objective:

Like following the trail of breadcrumbs, find the next precious item in the sacred sequence—an essential step in navigating the ordered maze.

Test Cases & Explanations:

- **Test Case 1: Node with Right Subtree**
  - *Input:* BST where the target node has a right child (e.g., Node `10` with right subtree).
  - *Expected Output:* The leftmost node in the right subtree.
  - *Explanation:* Verifies that the method correctly navigates the right branch when available.
- **Test Case 2: Node without Right Subtree**
  - *Input:* BST where target node does not have a right child.
  - *Expected Output:* The nearest ancestor for which the node lies in the left subtree.
  - *Explanation:* Confirms proper upward traversal to find the inorder successor.
- **Test Case 3: Maximum Element in BST**
  - *Input:* Target node is the maximum element in the tree.
  - *Expected Output:* `null` (or equivalent indication that no successor exists).
  - *Explanation:* Tests the edge case where the target has no successor.
- **Test Case 4: Single Node Tree**
  - *Input:* A tree with only one node.
  - *Expected Output:* `null`
  - *Explanation:* Minimal scenario ensuring that the algorithm handles solitary nodes appropriately.

7. Merge Two BSTs

Objective:
Unite two disparate groves into one harmonious sanctuary. By merging BSTs, the Code Explorers restore order, cultivating a single, thriving forest.

Test Cases & Explanations:

- **Test Case 1: One Empty Tree**
  - *Input:* One BST is empty while the other is non-empty.
  - *Expected Output:* The non-empty BST as-is.

    - *Explanation:* Validates correct handling of merging when one tree lacks nodes.

- **Test Case 2: Both Trees are Non-empty**
    - *Input:* Two BSTs with distinct, non-overlapping values.
    - *Expected Output:* A merged BST containing all values in sorted order.
    - *Explanation:* Checks that the merge operation properly combines and maintains BST properties.

- **Test Case 3: Overlapping Values (if allowed)**
    - *Input:* Two BSTs that may contain duplicate values.
    - *Expected Output:* A merged BST that correctly integrates the duplicates following your chosen ordering rules.
    - *Explanation:* Ensures that merging handles duplicates correctly if permitted by your design.

- **Test Case 4: Larger Trees**
    - *Input:* Two sizeable BSTs to test performance and correctness.
    - *Expected Output:* A balanced merged tree with nodes in strictly increasing order.
    - *Explanation:* Tests efficiency and validates that the method scales for larger datasets.

8. Find Closest Value

Objective:
Wield an enchanted compass to pinpoint the relic whose power is most akin to the mystical target value—ensuring that every treasure is discovered with precision.

Test Cases & Explanations:

- **Test Case 1: Exact Match Present**
    - *Input:* A BST with target value exactly matching a node.
    - *Expected Output:* That exact value.
    - *Explanation:* Validates that the algorithm immediately recognizes an exact match.

- **Test Case 2: Target Between Two Values**
    - *Input:* A BST where the target lies between two nodes' values.
    - *Expected Output:* The closest node value by difference.
    - *Explanation:* Confirms the method's ability to compute the minimal absolute difference

correctly.

- **Test Case 3: Target Less Than Minimum Value**
  - *Input:* A BST where the target is smaller than the smallest node value.
  - *Expected Output:* The smallest node value.
  - *Explanation:* Validates that the search works correctly at the lower bound.
- **Test Case 4: Target Greater Than Maximum Value**
  - *Input:* A BST where the target is larger than the largest node value.
  - *Expected Output:* The largest node value.
  - *Explanation:* Ensures the algorithm correctly identifies the boundary condition at the upper limit.

Part III: Restoring Harmony in the Balanced Forest

At the heart of the digital realm lies the Balanced BST—a tree of perfection whose equilibrium is the key to true restoration. Only through advanced augmentation and clever manipulation can the forest's harmony be reinstated:

9. Split BST

Objective:
With the wisdom of division, split a mighty tree into two balanced groves. One resulting BST must contain all nodes with values less than a given pivot, and the other holds nodes greater than or equal to the pivot.

Test Cases & Explanations:

- **Test Case 1: Empty Tree**
  - *Input:* An empty tree with any pivot value.
  - *Expected Output:* Two empty trees.
  - *Explanation:* Checks that splitting an empty tree gracefully returns two empty structures.
- **Test Case 2: Single Node Tree**
  - *Input:* A tree with one node and a pivot that is less than, equal to, or greater than the node's value.
  - *Expected Output:* Depending on the pivot: one BST contains the node, the other remains empty.

- *Explanation:* Tests basic split logic on minimal input.

- **Test Case 3: Balanced Tree**
  - *Input:* A balanced BST with several nodes and a pivot in the middle of the value range.
  - *Expected Output:* Two BSTs that are individually balanced and preserve BST properties.
  - *Explanation:* Ensures that the method splits correctly without disturbing the ordering.

- **Test Case 4: All Nodes on One Side**
  - *Input:* A BST where all node values are either less than or greater than the pivot.
  - *Expected Output:* One BST is identical to the original and the other is empty.
  - *Explanation:* Validates handling of edge cases where one sub-tree is entirely empty after the split.

10. Join BSTs

Objective:
Reunite two perfectly balanced groves into one extensive, enchanted forest. Assume that every element in the first BST is less than every element in the second BST.

Test Cases & Explanations:

- **Test Case 1: One Empty Tree**
  - *Input:* One balanced BST is empty, the other is non-empty.
  - *Expected Output:* The non-empty BST remains unchanged.
  - *Explanation:* Checks that joining with an empty tree returns the non-empty tree.

- **Test Case 2: Two Non-empty Trees**
  - *Input:* Two balanced BSTs with non-overlapping ranges.
  - *Expected Output:* A single balanced BST that maintains the in-order sequence.
  - *Explanation:* Validates that the join operation correctly merges both trees into a balanced result.

- **Test Case 3: Large Trees**
  - *Input:* Two large BSTs to test efficiency and correctness.
  - *Expected Output:* A combined BST that remains balanced with all nodes in order.
  - *Explanation:* Ensures that performance holds up when merging larger datasets.

- **Test Case 4: Boundary Conditions**
    - *Input:* Trees where the maximum value of the first tree is immediately less than the minimum of the second.
    - *Expected Output:* A seamless merge with no ordering conflicts.
    - *Explanation:* Tests the correctness when the trees are at their numeric boundaries.

11. Find Median

Objective:
Discover the central pivot of the forest—the median value that symbolizes absolute equilibrium. This requires that each node tracks the size of its subtree to efficiently compute the median.

Test Cases & Explanations:

- **Test Case 1: Empty Tree**
    - *Input:* `null` or empty tree.
    - *Expected Output:* An error or a designated value indicating an empty structure.
    - *Explanation:* Checks handling of edge cases where no median exists.
- **Test Case 2: Single Node Tree**
    - *Input:* A tree with one node.
    - *Expected Output:* The value of that node.
    - *Explanation:* Confirms correct behavior for the simplest non-empty tree.
- **Test Case 3: Even Number of Nodes**
    - *Input:* A BST with an even count of nodes.
    - *Expected Output:* Depending on your implementation (lower median, average, etc.).
    - *Explanation:* Validates that the method gracefully handles even-numbered collections.
- **Test Case 4: Odd Number of Nodes**
    - *Input:* A BST with an odd count of nodes.
    - *Expected Output:* The exact middle element in the in-order sequence.
    - *Explanation:* Ensures that the median is correctly identified in a straightforward scenario.

12. Range Sum Query

Objective:
Accumulate the latent energies of tree artifacts within a set range. Calculate the sum of all node values within an inclusive range [low, high] using augmented subtree sum data.

Test Cases & Explanations:

- **Test Case 1: Empty Tree**
  - *Input:* An empty tree and any range values.
  - *Expected Output:* `0`
  - *Explanation:* Validates that an empty tree contributes no sum.
- **Test Case 2: Single Node Tree**
  - *Input:* A tree with one node. For example, node value `10` with range [5, 15].
  - *Expected Output:* `10`
  - *Explanation:* Confirms that the method correctly sums when only one node exists.
- **Test Case 3: Range Excludes All Nodes**
  - *Input:* A BST where the provided range does not include any node values (e.g., range [50, 100] for a tree with all values below 50).
  - *Expected Output:* `0`
  - *Explanation:* Ensures the query returns zero when no nodes fall within the specified range.
- **Test Case 4: Mixed Range**
  - *Input:* A balanced BST with multiple nodes, and a range that includes some but not all nodes.
  - *Expected Output:* The sum of node values that are within the range.
  - *Explanation:* Verifies that the method efficiently computes the range sum using augmentation, even for partially overlapping ranges.

Happy coding, and may the algorithms be ever in your favor!