AVL Tree Specifications List

- 1. Self-Balancing (AVL)

1.1 Height Tracking
- **Purpose**: Store and update the height of each node to enable balance calculations
- **Assumptions**:
  - Each "AVLNode" has an integer "height" field initialized to 1 on creation
  - Heights of child pointers are correct before update
- **Inputs**:
  - A pointer to the node being updated during insert or delete operations
- **Outputs**:
  - Updated "height" value on the node ("1 + max(height(left), height(right))")
- **State Changes:**
  - Modifies the "height" field of the node
- **Cases and Expected Behavior:**
  - New Node: height set to 1
  - Leaf Promotion: when a leaf gains or loses children, height updates reflect new subtree height
  - Null Child: children pointers that are null are treated as height 0

1.2 Balance Factor
- **Purpose**: Compute the difference in heights ("height(left) - height(right)") to detect imbalance
- **Assumptions**:
  - Node heights are up-to-date
- **Inputs**:
  - A pointer to the node whose balance is being checked
- **Outputs**:
  - Integer balance factor
- **State Changes:**
  - None (read-only).
- **Cases and Expected Behavior:**
  - Balanced Node: returns 0
  - Left-Heavy: returns > +1 triggers rotations
  - Right-Heavy: returns < -1 triggers rotations
  - Null Node: returns 0.

1.3 Rotations
- **Purpose**: Rebalance subtrees when imbalance is detected
- **Assumptions**:
  - The subtree root is the first unbalanced node encountered
  - Children pointers and heights are valid.
- **Inputs**:

  - Pointer to unbalanced subtree root
- **Outputs**:
  - New subtree root pointer after rotation
- **State Changes:**
  - Modifies child pointers and updates affected node heights
- Cases and Expected Behavior:
  - LL Case:
    - Condition: balance > +1 and pos < node->left->pos
    - Action: single right rotation.
  - RR Case:
    - Condition: balance < -1 and pos > node->right->pos
    - Action: single left rotation.
  - LR Case:
    - Condition: balance > +1 and pos > node->left->pos
    - Action: left rotation on left child, then right rotation
  - RL Case:
    - Condition: balance < -1 and pos < node->right->pos
    - Action: right rotation on right child, then left rotation

- <u>2. Deletion with Rebalancing</u>

- **Purpose**: Remove a node by position and restore AVL balance
- **Assumptions**:
  - The tree follows BST invariants
- **Inputs**:
  - position to delete
- **Outputs**:
  - Updated tree with the node removed (if present)
- **State Changes:**
  - Alters tree structure: removes or replaces a node; updates heights; may rotate
- **Cases and Expected Behavior:**
  - Delete Leaf: removes node, parent updates height, then balances
  - Delete Node with One Child: child takes place of removed node
  - Delete Node with Two Children:
    - Find in-order successor ("minValueNode"), copy its pos, and delete successor
  - Delete Non-Existent pos: no change, returns original subtree
  - Post-Deletion Balancing: apply the same rotation logic as insertion

- <u>3. Search ("contains")</u>

- **Purpose**: Check presence of a pos in the tree (BST property search)
- **Assumptions**:
  - Tree nodes are properly ordered (left < parent < right)
- **Inputs**:

- position to search for
- **Outputs**:
  - Boolean indicating presence ("true" if found, "false" otherwise)
- **State Changes**
  - None
- **Cases and Expected Behavior:**
  - position Present: returns "true" after descending left/right
  - position Absent: returns "false" if a null child is reached
  - Empty Tree: returns "false"

- 4. Traversals

4.1 In-Order ("printInOrder")
- **Purpose**: Output position in ascending sorted order
- **Assumptions**:
  - Tree is a valid BST
- **Inputs**:
  - None (uses internal root pointer)
- **Outputs**:
  - Prints position to "stdout" separated by spaces
- **State Changes:**
  - None
- **Cases and Expected Behavior:**
  - Empty Tree: prints nothing, followed by newline
  - Single Node: prints that single value
  - General Case: left subtree, root, then right subtree

4.2 Pre-Order ("printPreOrder")
- **Purpose**: Output positions in pre-order (root, left, right) for debugging
- **Assumptions**:
  - Tree is a valid BST
- **Inputs**:
  - None (uses internal root pointer)
- **Outputs**:
  - Prints positions to "stdout" separated by spaces
- **State Changes:**
  - None
- **Cases and Expected Behavior:**
  - Provides root-first ordering, useful for serialization or debugging

- 5. Utility

5.1 Demo in "main.cpp"
- **Purpose**: Example usage: insert first 25 Fibonacci numbers and print in-order.

- **Assumptions**:
  - Fibonacci sequence fits in "int"
- **Inputs**:
  - None
- **Outputs**:
  - Printed sequence of in-order positions to console
- **State Changes:**
  - Populates tree, then read-only traversal
- **Cases and Expected Behavior:**
  - Successful Run: displays ascending Fibonacci numbers
  - Overflow (beyond "int"): undefined behavior; assumptions state 25th fib fits

5.2 Unit Tests in "test_avl_tree.cpp"
- Purpose: Automate validation of core features
- Assumptions:
  - Assertions are enabled; "assert" aborts on failure
- Inputs:
  - No external inputs; tests hardcode scenarios
- Outputs:
  - Console messages on test pass; program aborts on failure
- State Changes:
  - Each test uses a fresh "AVLTree" instance
- Cases and Expected Behavior:
  - Insert & Contains: verify present/absent positions
  - In-Order Traversal: visual check of sorted output
  - Delete Leaf & Two-Child Node: validate structure and search correctness
  - Failure Path: any failed assumption causes immediate abort