**Report for MSML606 HW3:  
Submitted by: Sriniketh Shankar (121113580)**

**1. Approach and Assumptions**

**Problem 1: Constructing a Binary Tree from BFS**

* **Approach**:  
  I split the input string on commas, then create the root node from the first token. Using a queue, I assign left and right children to each node in level-order. If the token is "None", the child pointer is not created.
* **Assumptions**:
  + The input string has no duplicates.
  + Missing children are represented by "None".

**Problem 2: In-Order Traversals (Recursive and Iterative)**

* **Recursive**:  
  A straightforward traversal that visits the left subtree, then the current node, then the right subtree.
* **Iterative**:  
  Uses a stack to simulate recursion. Repeatedly push left children, pop the top node, and process the right child afterward.
* **Assumptions**:
  + The tree could be incomplete or skewed.
  + Both traversals should produce the same node order.

**Problem 4: Validating a Binary Search Tree (BST)**

* **Approach**:  
  I use a recursive helper that checks whether each node’s value is between a lower and upper bound. These bounds get updated as I traverse left and right.
* **Assumptions**:
  + The BFS string used here can represent any tree, not necessarily a BST.
  + The function should return True only if all BST properties hold.

**2. Time Complexity of Each Method**

1. **Constructing the Tree** (Problem 1):
   * **Time**: O(n), where n is the number of nodes, because each node is processed once in a level-order manner.
   * **Space**: O(n) for the queue and the tree nodes.
2. **In-Order Traversals** (Problem 2):
   * **Recursive**:
     + **Time**: O(n), visiting each node exactly once.
     + **Space**: O(h) for the call stack, where h is the tree height (worst-case O(n) in a skewed tree).
   * **Iterative**:
     + **Time**: O(n).
     + **Space**: O(h) for the stack, similarly worst-case O(n).
3. **BST Validation** (Problem 4):
   * **Time**: O(n) because each node is checked exactly once.
   * **Space**: O(h) in recursion stack usage.

**3. Experimental Comparison for Problem 3**

I generated two types of trees for varying n (10, 100, 1000, 5000):

1. **Complete Binary Trees**: Height ≈ log n.
2. **Skewed Binary Trees** (all-left): Height =n.

For both **recursive** and **iterative** in-order traversals, I measured:

* **Running Time** (milliseconds) using time.perf\_counter(), averaged over multiple runs.
* **Peak Memory** (KB) using tracemalloc.

**3.1 Sample Results Table**

**A screenshot of a computer

AI-generated content may be incorrect.**

A graph of different colored lines

AI-generated content may be incorrect.

**3.2 Observations and Trends**

1. **Time Complexity**:  
   Both methods show linear growth in running time, consistent with O(n). However, the recursive approach can fail for large skewed trees because the call stack becomes too deep.
2. **Memory Usage**:
   * For **complete** trees, memory usage remains relatively low because the height is only O(log n).
   * For **skewed** trees, the iterative approach still uses O(n) space, but avoids recursion limits. The recursive version can exceed Python’s recursion depth around 1000 nodes, hence the “∞” in the table.
3. **Asymptotic Analysis**:
   * In both balanced and skewed trees, the theoretical time complexity for in-order traversal (recursive or iterative) is O(n)O(n).
   * Space complexity is O(h)O(h). For complete trees, h≈ log n, while for skewed trees, h=h = n. The recursive method hits practical limitations due to Python’s recursion cap, while the iterative method handles large skewed trees more gracefully.

**Conclusion**

* **Problems 1 & 2**: Verified correctness of tree construction and in-order traversal via BFS-based tests.
* **Problem 3**: Demonstrated that recursive and iterative traversals are both O(n)O(n), but recursion can fail for large skewed trees. The iterative approach avoids stack-depth issues.
* **Problem 4**: Confirmed BST validation is correct by comparing against expected “True/False” results.

Overall, the experiments illustrate how, despite having the same asymptotic complexity, recursion may not be practical for very deep (skewed) trees in Python. The iterative method is more robust in those scenarios, which is reflected in the timing and memory measurements.