MSML606 Assignment 3 Report

Ayush Sinha LNU

UID: 121334060

Approach for Each Problem:

Question 1: Construct Binary Tree from BFS Input

Approach:

- We have taken BFS-style input string, split it by the commas and construct a binary tree.
- The first value is assigned as root.
- Queue is utilized to iterate over the values which adds left and right children sequentially.
- “None” values are treated as null nodes.

Assumptions:

- We have assumed that the input would be a valid BFS representation of a binary tree.
- The tree may consist of null nodes i.e. the “None” values.

Question 2: In-Order Traversal

Part 1: Recursive In-Order Traversal

Approach:

- We use a recursive function to traverse the left subtree, then process the root and traverse the right subtree.

Assumptions:

- The input tree is correctly structured.

Part 2: Iterative In-Order Traversal

Approach:

- We use a stack to mimic the recursive call stack.
- The left subtree nodes are pushed onto the stack before processing the root and moving to the right subtree.

Assumptions:

- Here we assume that the input tree is correctly structured.

Question 3: Binary Tree Generation

Part (a): Generating a Random Permutation

Approach:

- Here we create a list of numbers from 1 to N and shuffle it randomly.

Part (b): Generating a Complete Binary Tree

Approach:

- Here we create TreeNode instances for each number.
- Nodes are linked based on their index in a complete binary tree.

Part (c): Generating a Skewed Binary Tree

Approach:

- A tree similar to a linked-list is created with nodes attached either only to the left or to the right.

Question 4: Validate a Binary Search Tree

Approach:

- Here we use a recursive function to validate if the tree satisfies BST properties.
- The left subtree must contain values less than the root and the right subtree must contain values greater than the root.

Assumptions:

- The assumption here would be to consider that the input is a binary tree that may or may not be a BST.

Time Complexity Analysis

Problem	Approach	Time Complexity
Construct Binary Tree	BFS Iteration	O(N)
Recursive In-Order Traversal	Recursion	O(N)
Iterative In-Order Traversal	Stack Based Iteration	O(N)
Random Permutation	Shuffling	O(N)
Complete Binary Tree Construction	Index Based Assignments	O(N)
Skewed Binary Tree Construction	Sequential Node Linking	O(N)
Validate BST	Recursive Bound Checking	O(N)

1. **Constructing Binary Tree (BFS Iteration)**
    - We process N nodes once in a BFS manner.
    - Each node is enqueued and dequeued only once.
    - Time Complexity = O(N).
2. **Recursive In-Order Traversal**
    - We visit each node once, making one recursion call per node.
    - Time Complexity = O(N).
3. **Iterative In-Order Traversal**
    - Each node is pushed into the stack once and popped once.
    - Time Complexity = O(N).
4. **Generating Random Permutation**
    - This swaps the elements in the list in O(N) time.
    - Time Complexity = O(N).
5. **Complete Binary Tree Construction**
    - Creates N nodes and assigns the left/right children based on indices.
    - Each node takes O(1) time.
    - Time Complexity = O(N).
6. **Skewed Binary Tree Construction**
    - Creates N nodes and links them sequentially.
    - Time complexity = O(N).
7. **Validating BST (Recursive Bound Check)**
    - Each node is visited once and each recursive call processes one node.
    - Time Complexity = O(N).

Performance Analysis

Setup

- We measured execution time and memory usage for recursive and iterative in-order on both complete and skewed trees.
- Tests were conducted on trees of sizes 10, 100, 1000, and 5000 with 100 trials per test.

Results

Running Time and Memory Usage Comparison (Tabular Form and Plot):

Tree Type	Method	N=10 (Time Memory)	N=100 (Time Memory)	N=1000 (Time Memory)	N=5000 (Time Memory)
Complete	Recursive	0.001018s, 416 bytes	0.009153s, 1744 bytes	0.106255s, 16144 bytes	0.558826s, 80144 bytes
Complete	Iterative	0.000246s, 192 bytes	0.001877s, 928 bytes	0.012925s, 8896 bytes	0.049470s, 41920 bytes
Skewed	Recursive	0.000861s 192 bytes	0.013819s, 1632 bytes	0.561993s, 16032 bytes	11.559169s, 80032 bytes
Skewed	Iterative	0.0003s, 192 bytes	0.001512s, 1312 bytes	0.008699s, 12480 bytes	0.034079s, 62336 bytes

Observations

- Recursive traversal is efficient for complete trees but suffers from stack overflow in skewed trees when N is large.

- In order to overcome this exception, “sys.setrecursionlimit(10000)” was used which led to efficient handling when N became very large.
- Iterative traversal consistently performs better for skewed trees due stack optimization.
- The memory usage of recursive methods is significantly higher for skewed trees.
- The time consumption for skewed trees in recursive methods is very high.
- Iterative traversal remains stable in both running time and memory usage across all three types.

Asymptotic Analysis

- Recursive traversal has an average-case time complexity of O(N) but requires O(H) memory where H is nothing but tree height.
- Iterative Traversal has an O(N) time complexity and O(N) worst-case space for skewed trees but O(log N) for balanced trees.

Graphical Plots

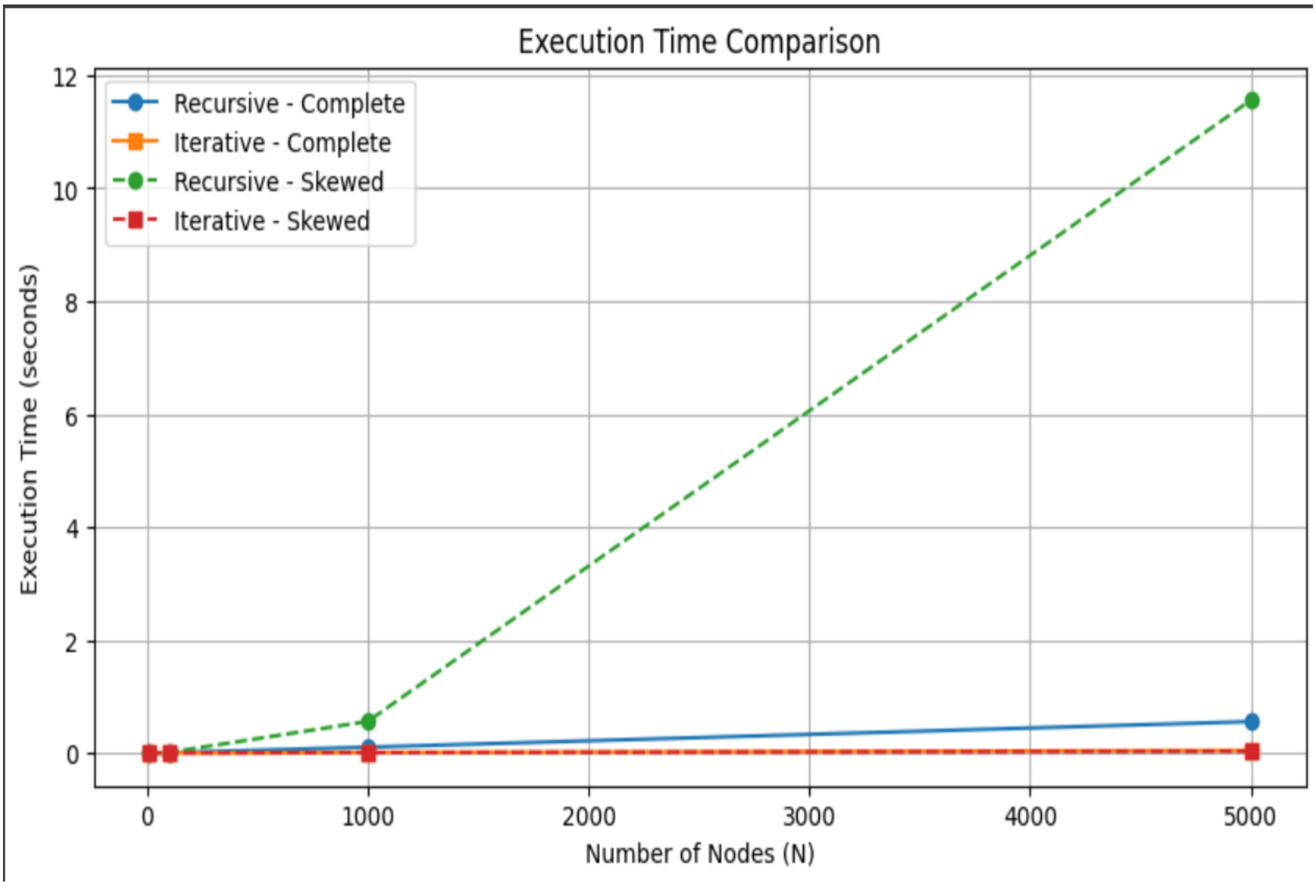
The following plots shows us a comparative analysis of the execution time and memory usage for both the recursive as well as the iterative approaches for complete and skewed binary trees.

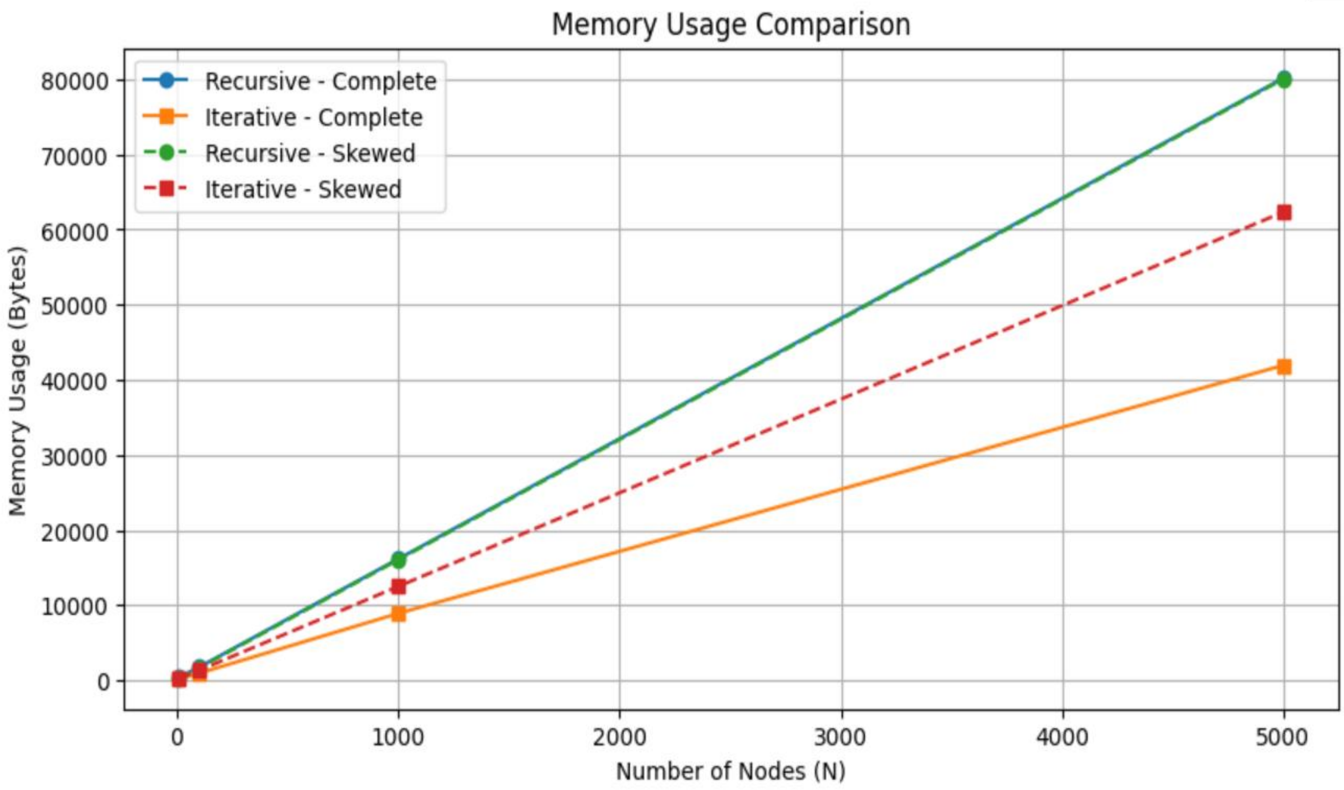
1. Execution Time Plot

- This graph represents the execution time for recursive and iterative methods.
- We can see that recursion performs better for complete balanced trees but performs poorly for the skewed trees.
- Iterative methods perform consistently for both types of trees.
- These plots were created using Google Colab.

2. Memory Usage Plot

- This plot shows the memory usage of recursive and iterative approaches.
- Recursive approach for both the skewed and complete binary trees consumes a significant amount of memory.
- Iterative methods on the other hand shows a better performance in memory usage.
- These plots were created using Google Colab.





Conclusion

- Iterative in-order traversal is more reliable for large, skewed trees due to controlled stack depth.
- Recursive methods are intuitive, but they are at risk of stack overflowing.
- Complete trees balance recursion well, while skewed trees highlight its limitations.