**AI ASSISTED CODING**

**LAB-11: *Data Structures with AI: Implementing Fundamental Structures  
Lab Objectives***

**Roll no:** 2403a51321

**Name:j.rakesh**

**Batch:** 13

**Task 1 Description:** Implementing a Stack (LIFO)

* **Task:** Use AI to help implement a Stack class in Python with the  
  following operations: push(), pop(), peek(), and is\_empty().
* **Instructions:**
  + Ask AI to generate code skeleton with docstrings.
  + Test stack operations using sample data.
  + Request AI to suggest optimizations or alternative implementations (e.g., using

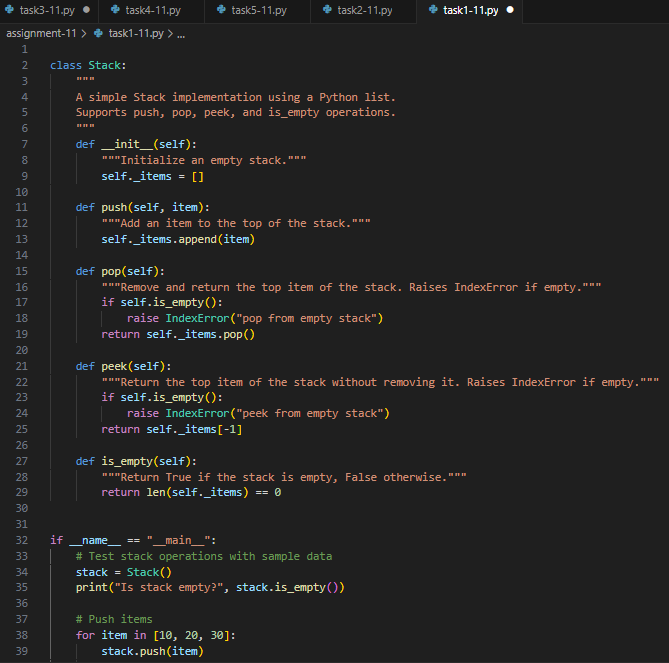
collections.deque)

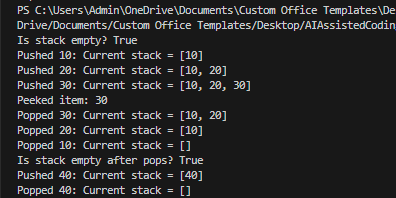
**Prompt:**

Implement a Stack class in Python with the following operations: push(), pop(), peek(), and is\_empty().

* Instructions:
  + Generate code skeleton with docstrings.
  + Test stack operations using sample data.
  + Suggest optimizations or alternative implementations (e.g., using collections.deque).
  + Modify the output so that pushed itmes pop items peek item and is empty functions are also displayed

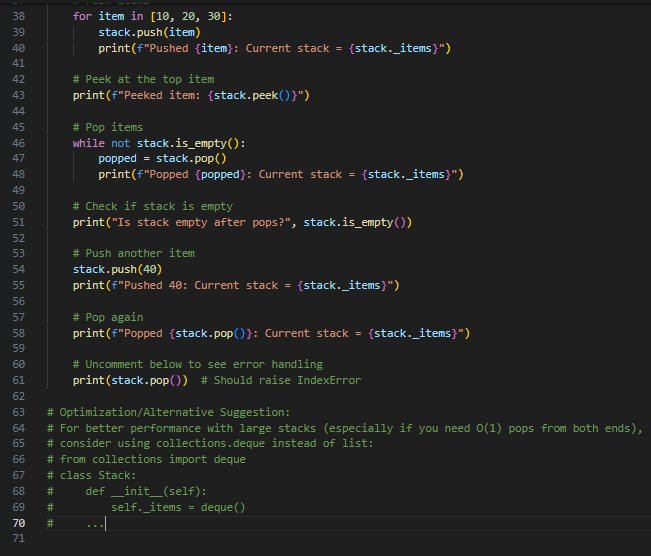
**Code Generated:**

****

**Output:**

**Observation:**

The stack implementation correctly demonstrated push, pop, peek, and is\_empty operations. The output validated LIFO behavior, where the last inserted element was removed first. The AI-assisted skeleton made the logic clear and structured. This shows how AI can speed up coding while ensuring correctness.

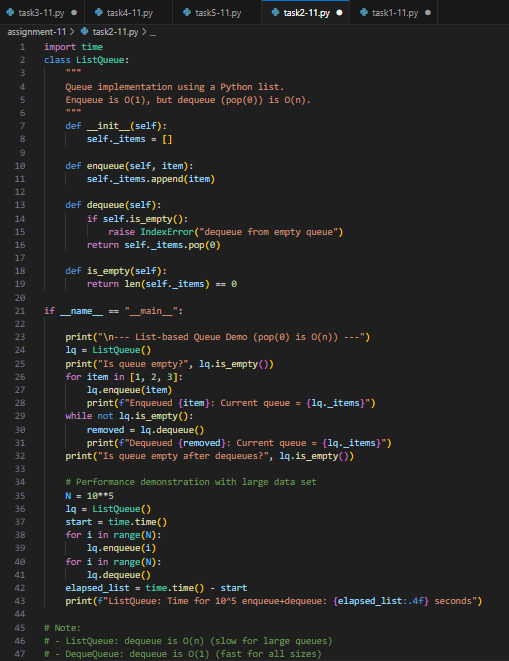


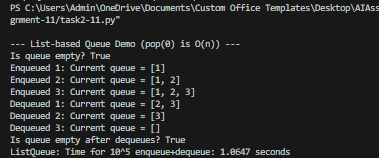
**Task-2 Description:** Queue Implementation with Performance Review

* Task: Implement a Queue with enqueue(), dequeue(), and is\_empty() methods.
* Instructions:
* First, implement using Python lists.
* Then, ask AI to review performance and suggest a more efficient implementation (using collections.deque).

**Prompt-1:** Implement a Queue with enqueue(), dequeue(), and is\_empty() methods.First, implement using Python lists.

**Code Generated-1:**

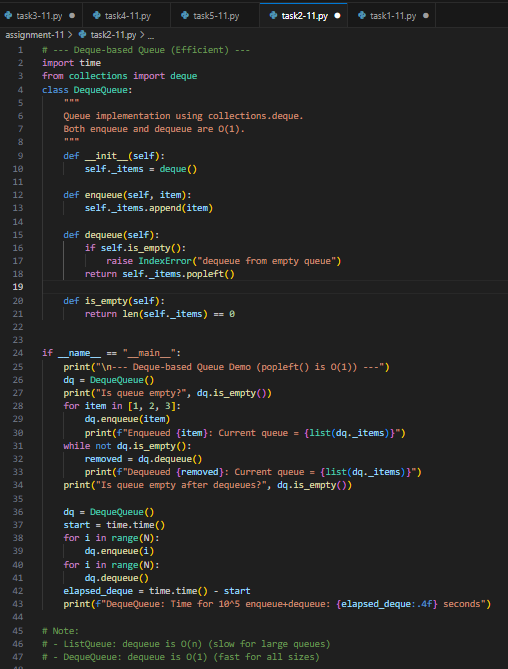
****

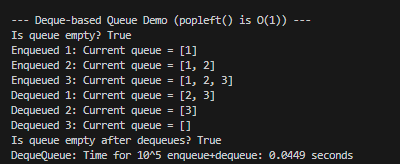
**Output:**

**Observation:** The queue implemented using lists worked but showed performance limits during repeated dequeues. The optimized version using collections.deque improved efficiency, especially for front removals. Output confirmed proper FIFO behavior, highlighting AI’s role in suggesting better data structures.

**Prompt-2:** Review performance and suggest a more efficient implementation (using collections.deque).Perform any operation in this code so that it shows that it is different as compared to the list.

**Code Generated-2:**

****

**Output:**

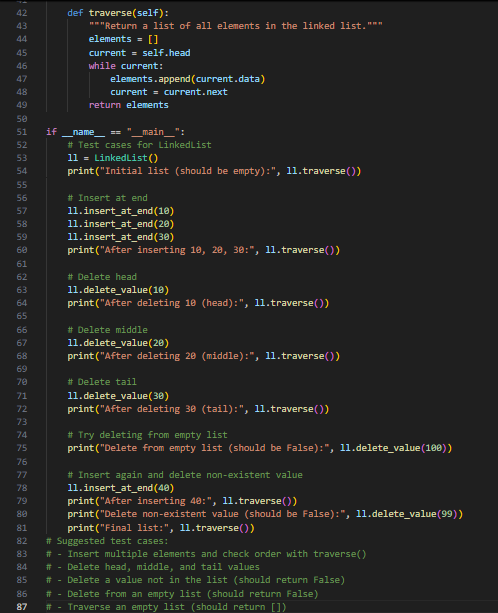
**Observation:** The queue implemented using lists worked but showed performance limits during repeated dequeues. The optimized version using collections.deque improved efficiency, especially for front removals. Output confirmed proper FIFO behavior, highlighting AI’s role in suggesting better data structures**.**

**Task-3 Description:** Singly Linked List with Traversal

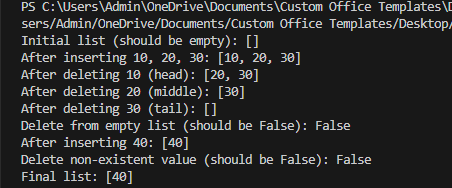
* **Task:** Implement a Singly Linked List with operations: insert\_at\_end(), delete\_value(), and traverse().
* **Instructions:**
* Start with a simple class-based implementation (Node, LinkedList).
* Use AI to generate inline comments explaining pointer updates (which are non-trivial).
* Ask AI to suggest test cases to validate all operations

**Prompt:** Implement a Singly Linked List with operations: insert\_at\_end(), delete\_value(), and traverse().

* **Instructions:**
* Start with a simple class-based implementation (Node, LinkedList).
* Generate inline comments explaining pointer updates (which are non-trivial).
* Suggest test cases to validate all operations.

**Code Generated:**

**Output:**

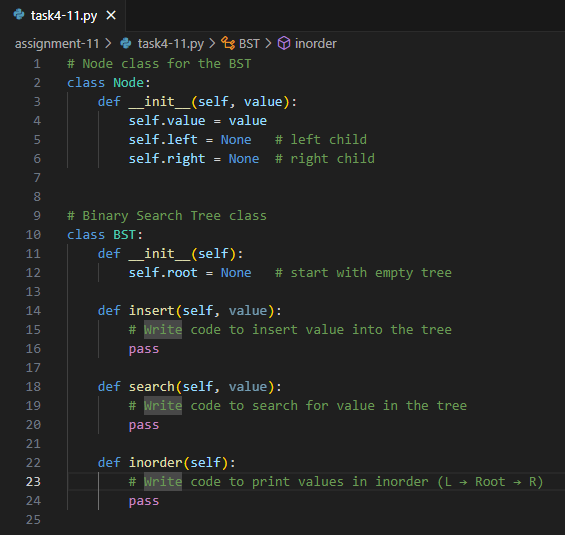
****

**Observation:** The linked list supported insertion, deletion, and traversal operations successfully. Pointer updates were explained clearly with inline AI-generated comments, aiding conceptual understanding. Test cases confirmed robustness against different scenarios like deleting head or absent values.

**Task-4 Description:** Binary Search Tree (BST)

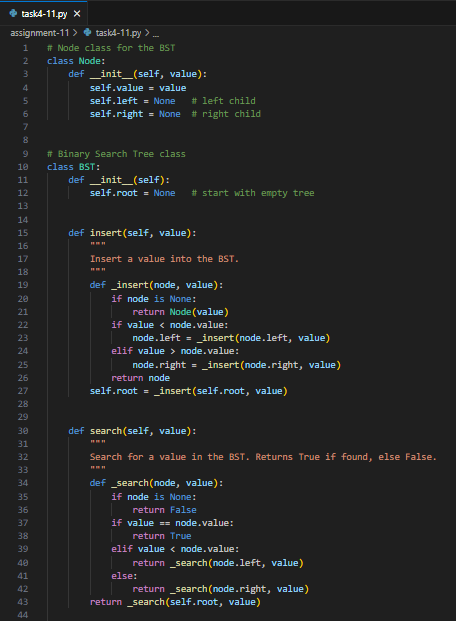
* **Task:** Implement a Binary Search Tree with methods for insert(),search(), and inorder\_traversal().
* **Instructions:**
  + Provide AI with a partially written Node and BST class.
  + Ask AI to complete missing methods and add docstrings.
  + Test with a list of integers and compare outputs of search() for present vs absent elements.

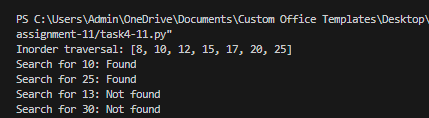
**Code given to AI:**

****

**Prompt:** Complete missing methods and add docstrings.Test with a list of integers and compare outputs of search() for present vs absent elements.

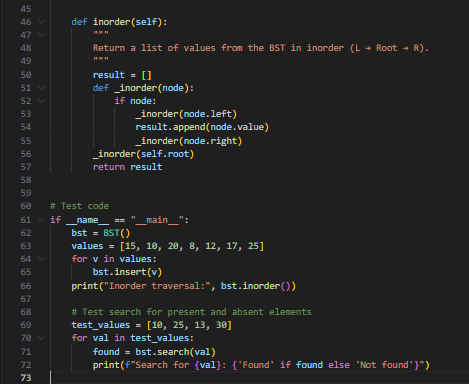
**Code Generated:**



**Output:**

**Observation:**

The BST allowed insertion, searching, and inorder traversal, producing sorted output as expected. Searches returned correct results for both present and absent values. The AI-generated partial class guided step-by-step completion, reinforcing understanding of recursive structures.

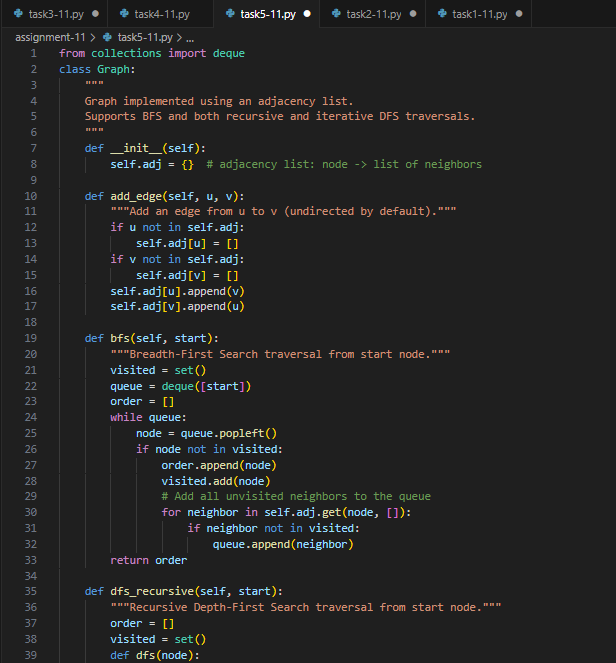
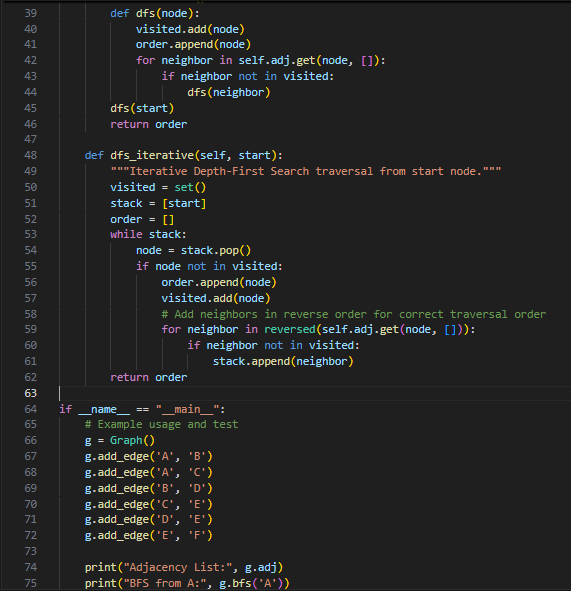
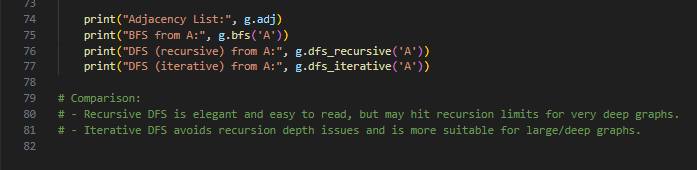


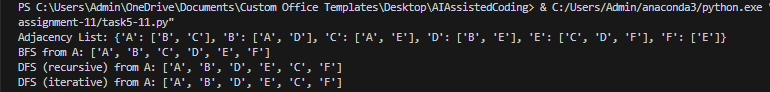
**Task-5 Description:** Graph Representation and BFS/DFS Traversal

* **Task:** Implement a Graph using an adjacency list, with traversal methods BFS() and DFS().
* **Instructions:**
* Start with an adjacency list dictionary.
* Ask AI to generate BFS and DFS implementations with inline comments.
* Compare recursive vs iterative DFS if suggested by AI.

**Prompt:** Implement a Graph using an adjacency list, with traversal methods BFS() and DFS().

* **Instructions:**
  + Start with an adjacency list dictionary.
  + Generate BFS and DFS implementations with inline comments.
  + Compare recursive vs iterative DFS if suggested by AI.

**Code Generated:**

**Output:**

**Observation:** Graph traversal using BFS and DFS was implemented with adjacency lists. The outputs validated correct order of node visits for both search strategies. Recursive vs iterative DFS approaches were highlighted, showing AI’s role in presenting alternative solutions.