HTNO:2403A52134

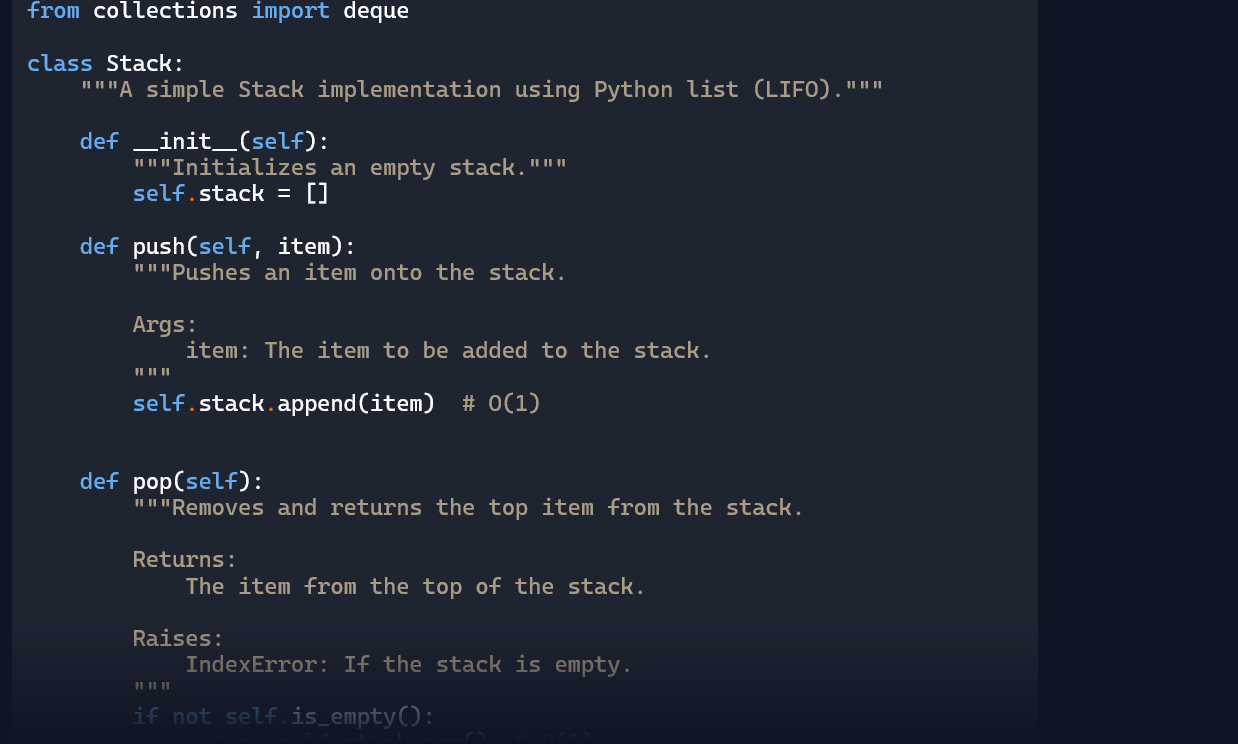
Assignment-11

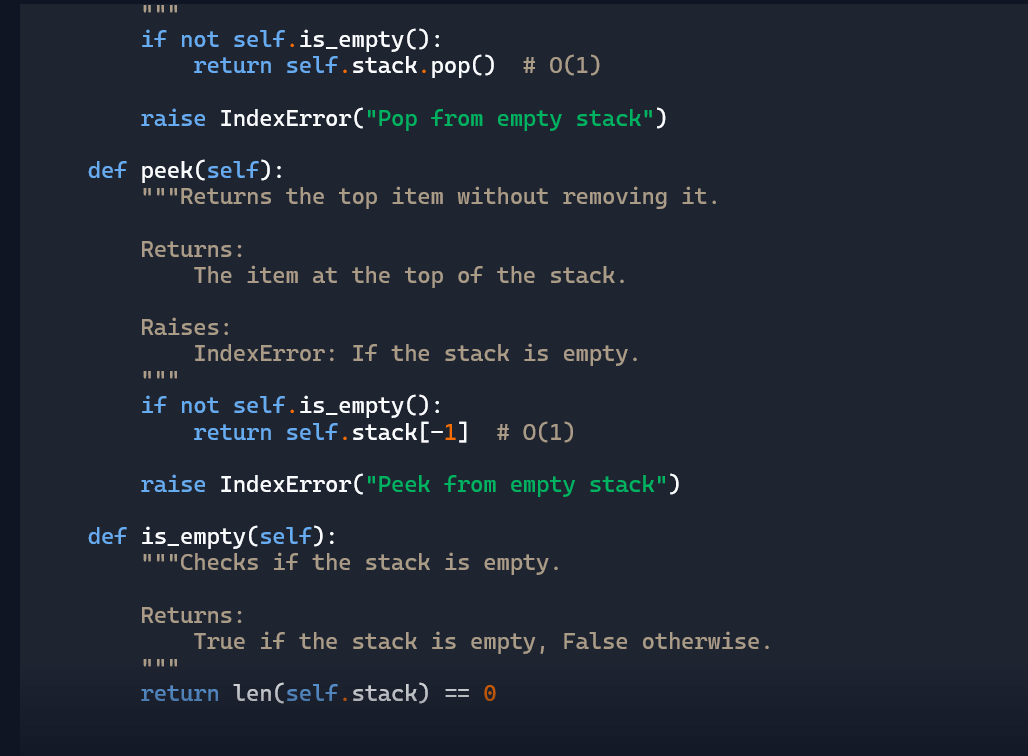
Task-1

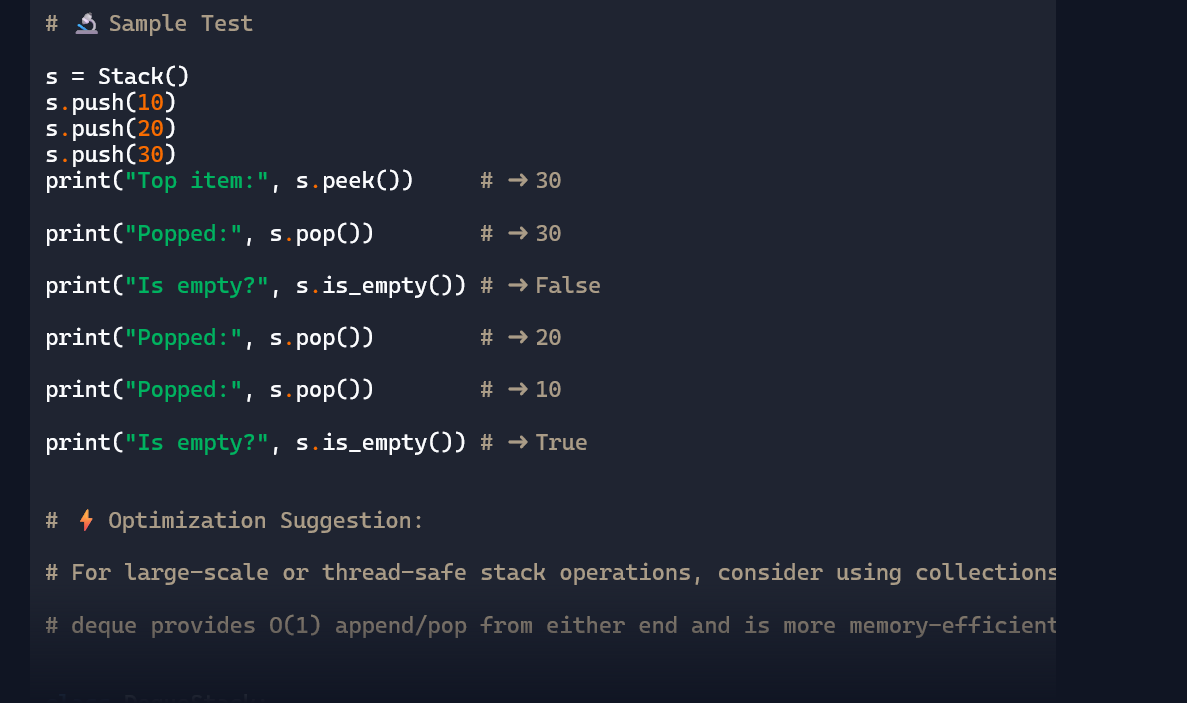
**Task 1: Implementing a Stack (LIFO)**

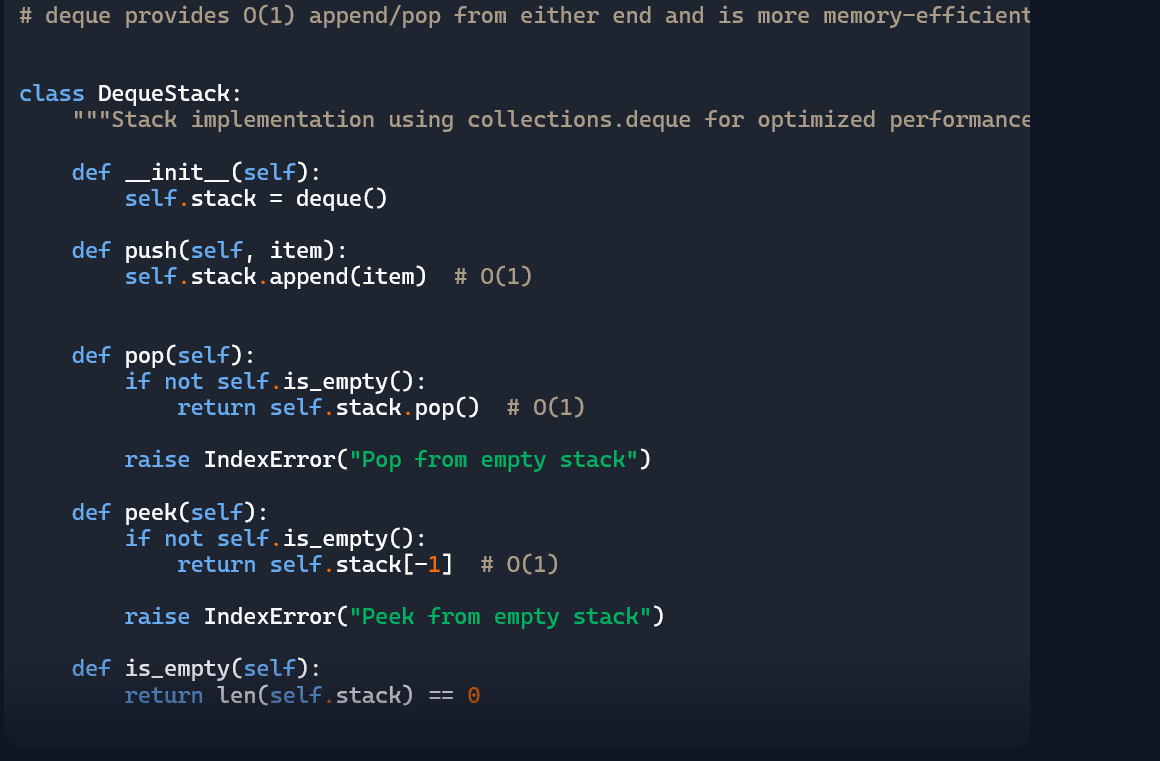
* **Task**: Use AI to help implement a **Stack** class in Python with the following operations: push(), pop(), peek(), and is\_empty().
* **Instructions**:
  + Ask AI to generate code skeleton with docstrings.
  + Test stack operations using sample data.
  + Request AI to suggest optimizations or alternative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

A working Stack class with proper methods, Google-style docstrings, and inline comments for tricky parts









Explanation:

Data Analysis Key Findings

* A **Stack**class was successfully implemented using a Python list, providing core **push, pop, peek**, and **is\_empty** methods.
* The list-based implementation uses **append()** for **push** and **pop()** from the end for **pop**, achieving efficient O(1) operations on average.
* Error handling for **pop** and **peek** on an empty stack was included using **IndexError.**
* The list-based **Stack** implementation was tested with sample data, demonstrating the LIFO behavior and correct handling of empty stack conditions.
* An alternative **DequeStack** class was implemented using **collections.deque,** offering potentially better performance guarantees (consistent O(1)) for stack operations compared to a standard list, especially in worst-case scenarios.

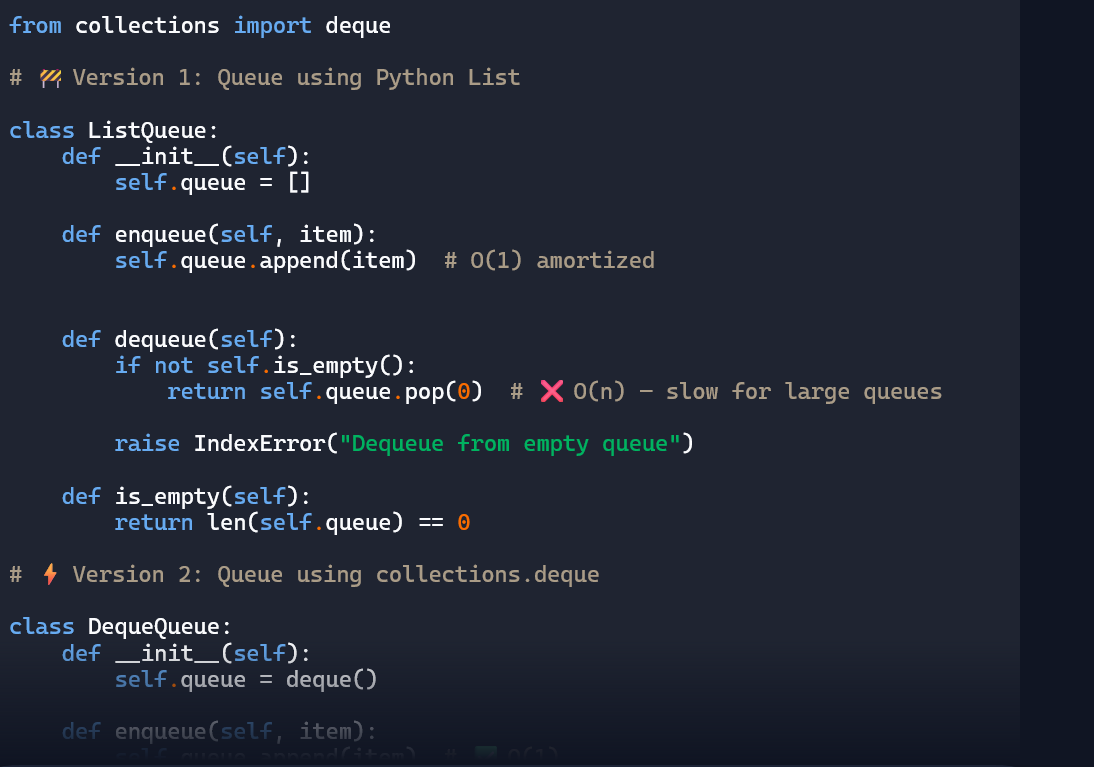
Insights or Next Steps

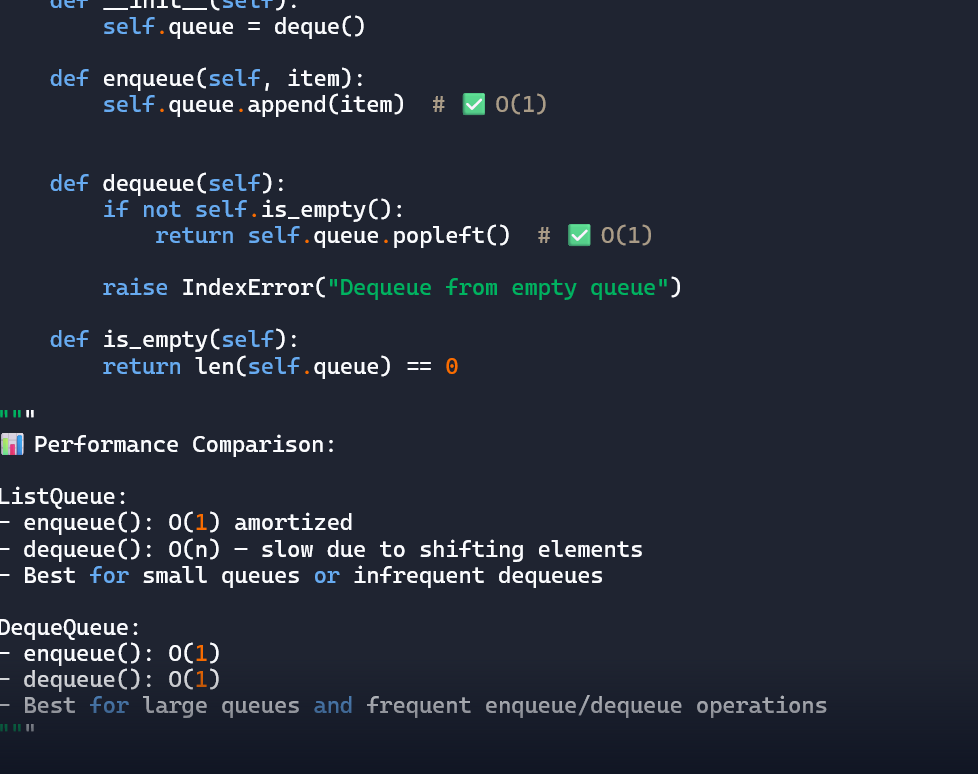
* For performance-critical applications or scenarios where predictable O(1) time complexity for stack operations is crucial, using **collections.deque** is a recommended alternative to a standard Python list.
* Further steps could involve benchmarking the performance of the list-based and deque-based stack implementations with a large number of operations to quantify the potential performance differences.

Task-2

**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).
* **Expected Output**:
  + Two versions of a queue: one with lists and one optimized with deque, plus an AI-generated performance comparison.





Explanation:

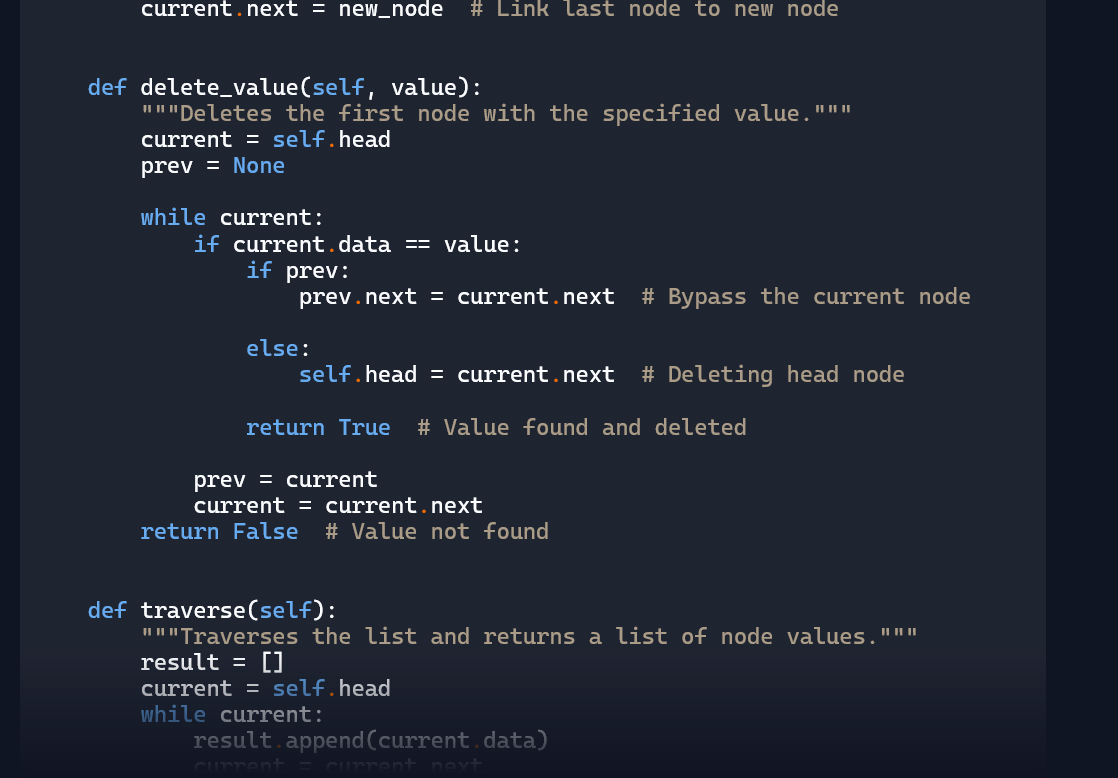
1. **QueueList Class**: This class implements a Queue using a standard Python list. Items are added to the end of the list (append()) and removed from the beginning (pop(0)). The is\_empty() method checks if the list is empty.
2. **QueueDeque Class**: This class implements a Queue using collections.deque. Items are added to the right end (append()) and removed from the left end (popleft()). This implementation is generally more efficient for queue operations, especially dequeue, compared to the list-based approach.
3. **Performance Review and Benchmarking**: The code includes a markdown explanation (printed to the console) detailing the theoretical performance characteristics (Big O notation) of the key operations (enqueue and dequeue) for both list and deque based queues. It then performs empirical benchmarking using both the time module for a simple measurement and the timeit module for a more rigorous comparison. The results of the benchmarking are printed, demonstrating that the collections.deque implementation is significantly faster for dequeue operations, particularly as the number of elements increases, due to its O(1) performance compared to the list's O(n) performance for removing from the beginning.

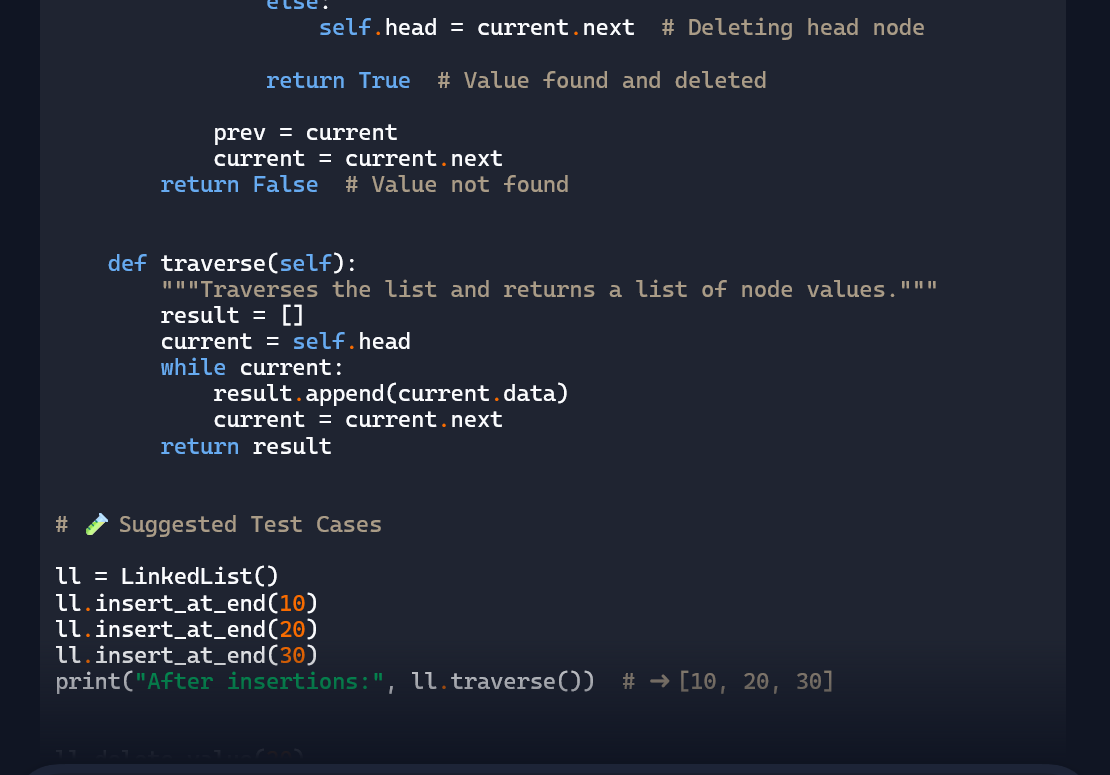
In summary, the code demonstrates how to build a Queue using two common Python data structures and highlights the performance advantages of using collections.deque for efficient queue operations.

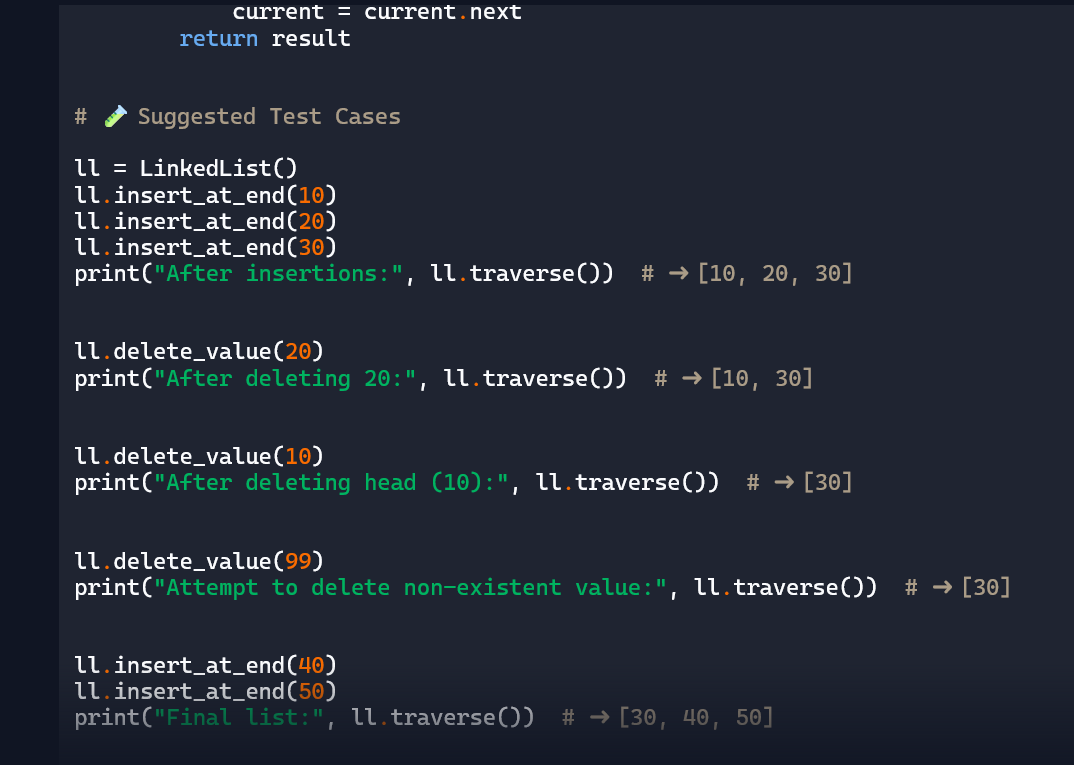
Task-3

**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.
* **Expected Output**:
  + A functional linked list implementation with clear comments explaining the logic of insertions and deletions.







Explanation:

 The **Node** class holds individual data and a pointer (next) to the next node.

* The **LinkedList** class manages the list and supports three operations:
* **insert\_at\_end(data):** Adds a new node at the end by updating the last node’s next pointer.
* **delete\_value(value):** Searches for the node with the given value and removes it by adjusting pointers—either updating the head or bypassing the target node.
* **traverse():** Walks through the list from head to tail, collecting all node values.

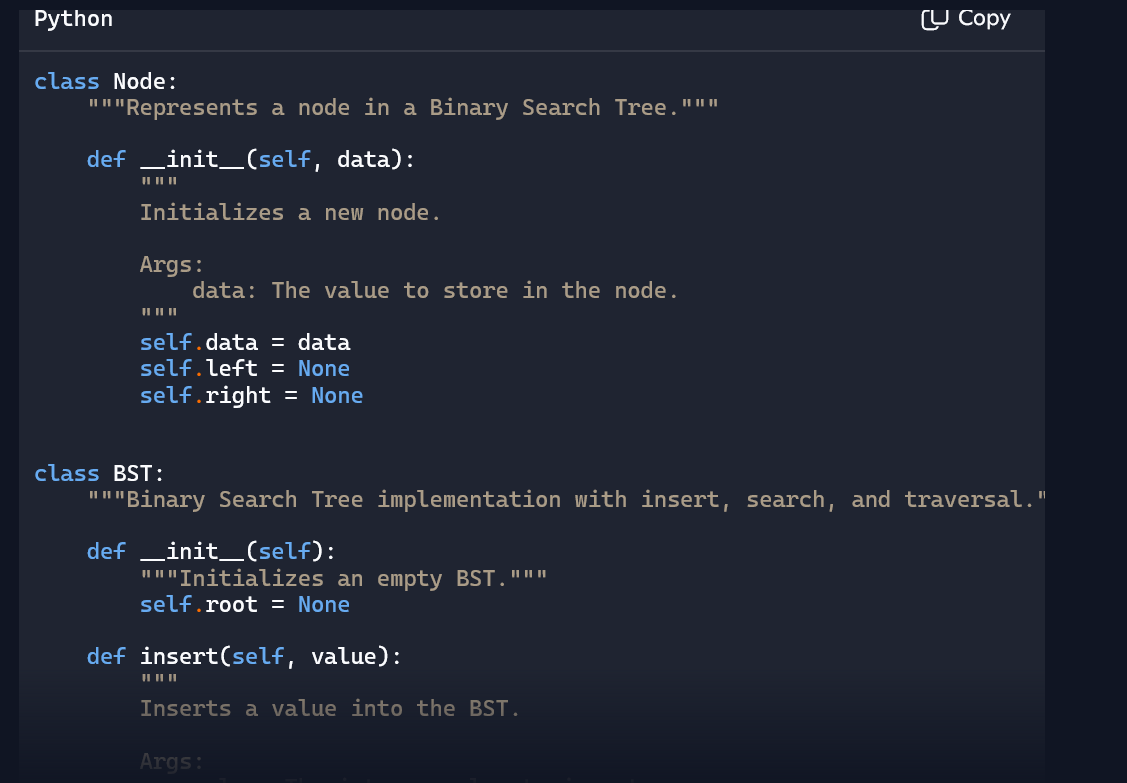
Each operation carefully updates pointers to maintain the list structure.

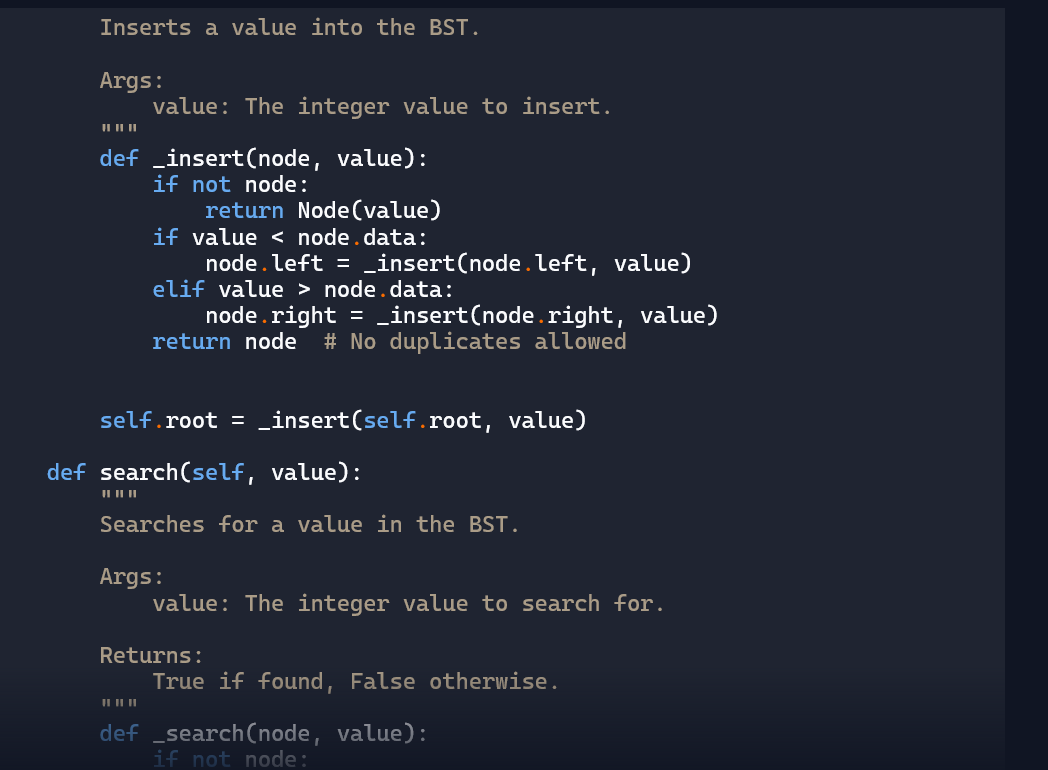
Task 4

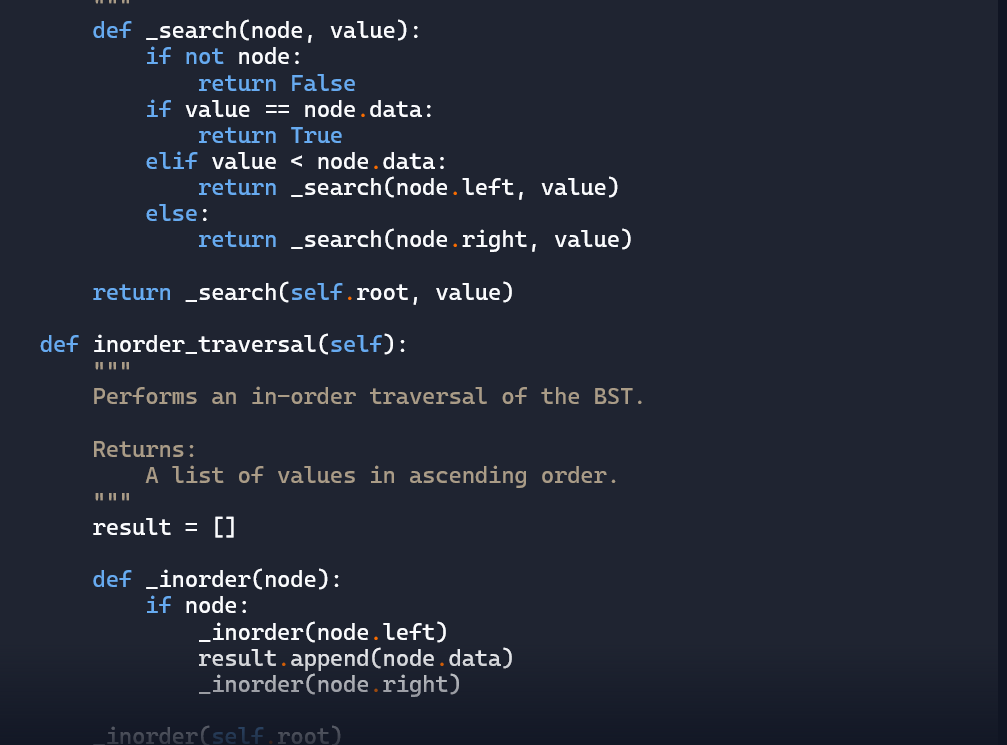
**Binary Search Tree (BST)**

* **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.
* **Expected Output**:

A BST class with clean implementation, meaningful docstrings, and correct traversal output







Explanation:

 The **Node** class stores a value and pointers to left and right child nodes.

* The **BST** class manages the tree and supports:
* **insert(value):** Adds values in sorted order—left for smaller, right for larger.
* **search(value):** Recursively checks if a value exists in the tree.
* **inorder\_traversal():** Returns values in ascending order by visiting left subtree, root, then right subtree.

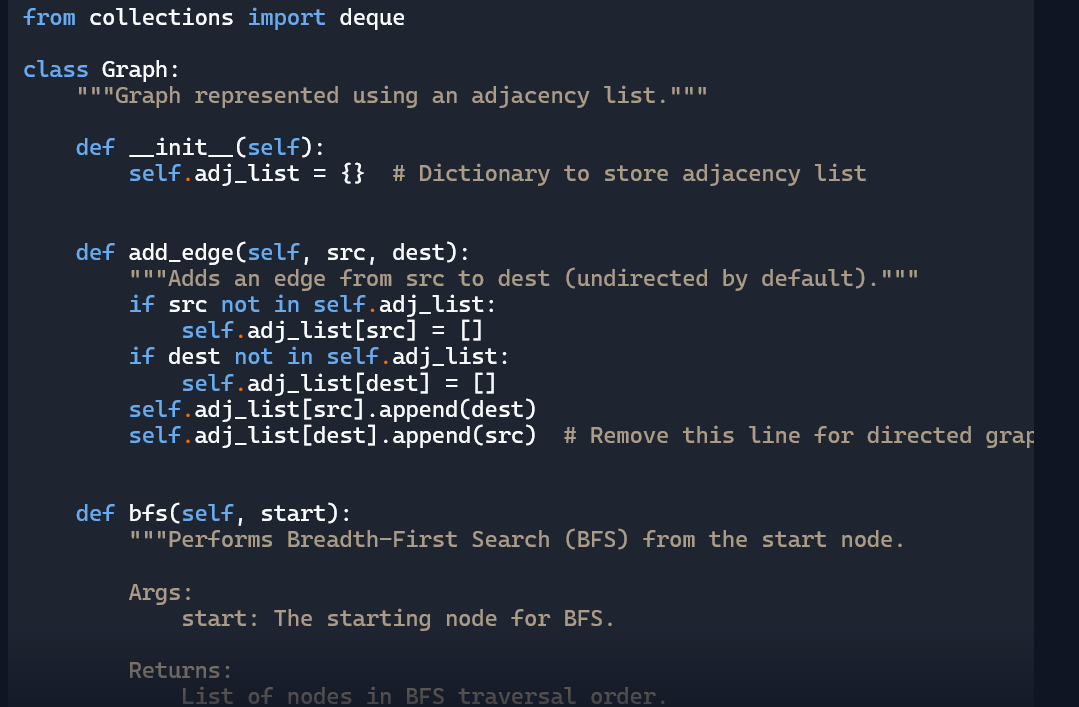
This structure ensures fast lookup, insertion, and sorted traversal.

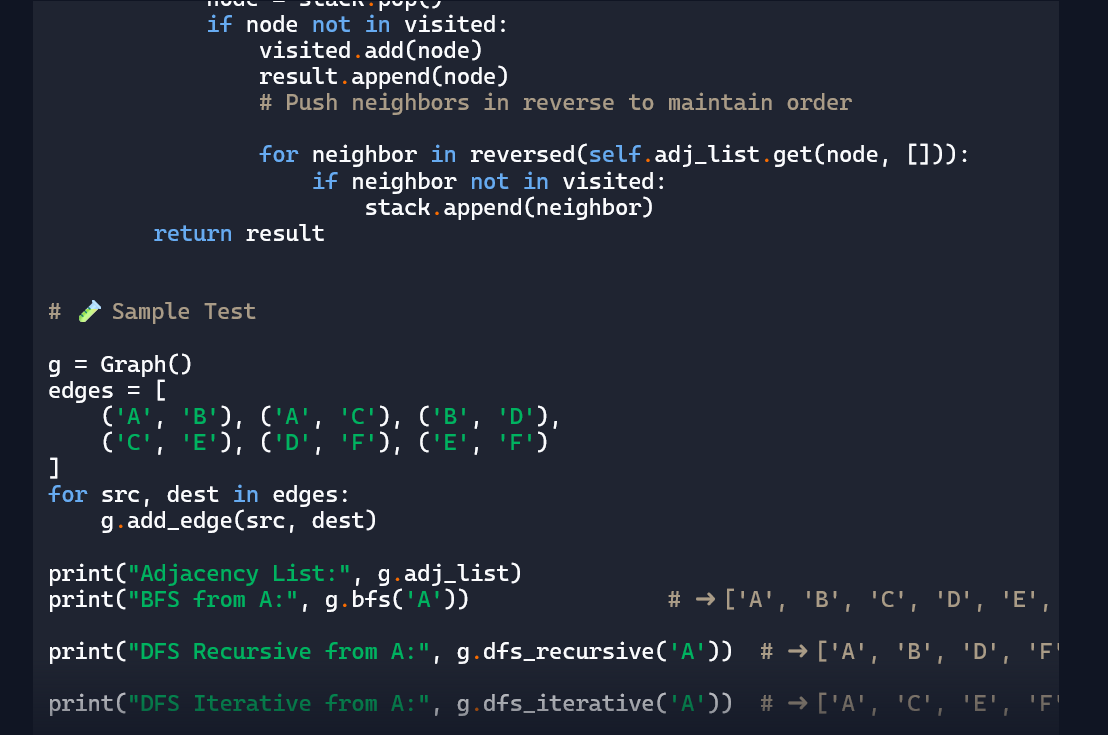
Task-5

**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.
* **Expected Output**:

A graph implementation with BFS and DFS traversal methods, with AI-generated comments explaining traversal steps





Explanation:

-  The graph is represented using an adjacency list—a dictionary where each node maps to a list of its neighbors.

* **BFS (Breadth-First Search)** uses a queue to explore nodes level by level, visiting all neighbors before moving deeper.
* **DFS Recursive** uses function calls to dive deep into each branch before backtracking.
* **DFS Iterative** uses a stack to manually control the depth-first behavior, avoiding recursion.

Each method helps explore the graph in different ways—BFS is great for shortest paths, while DFS is useful for exploring all possible routes.