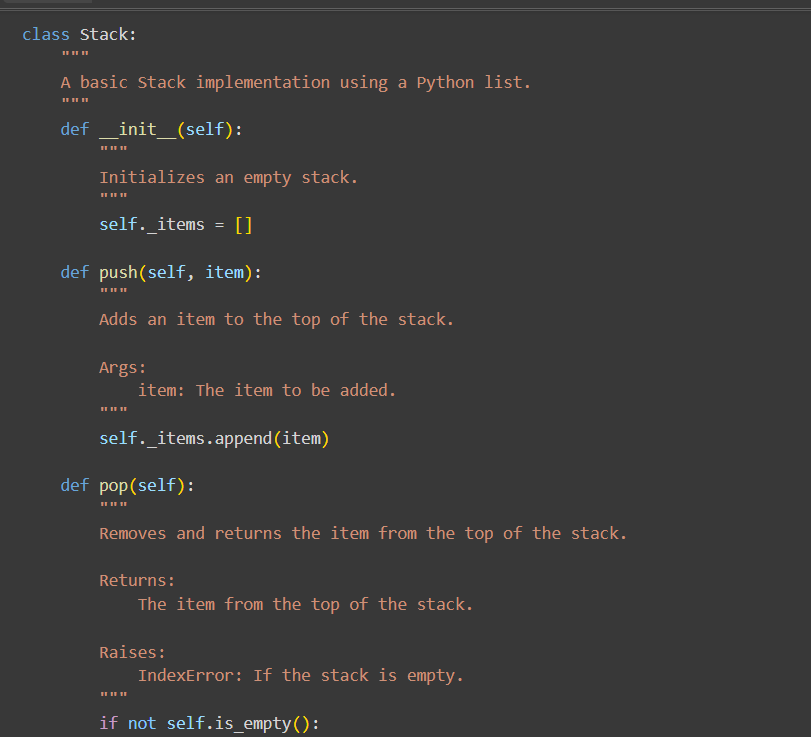
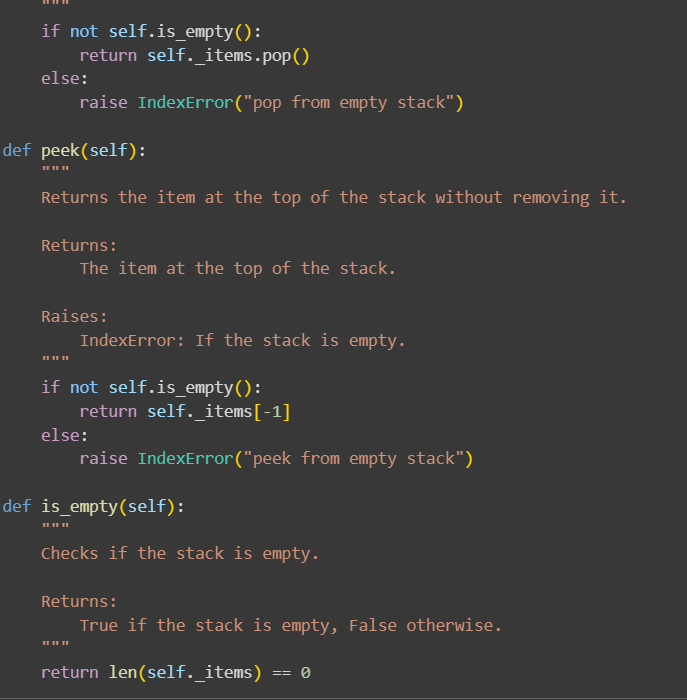
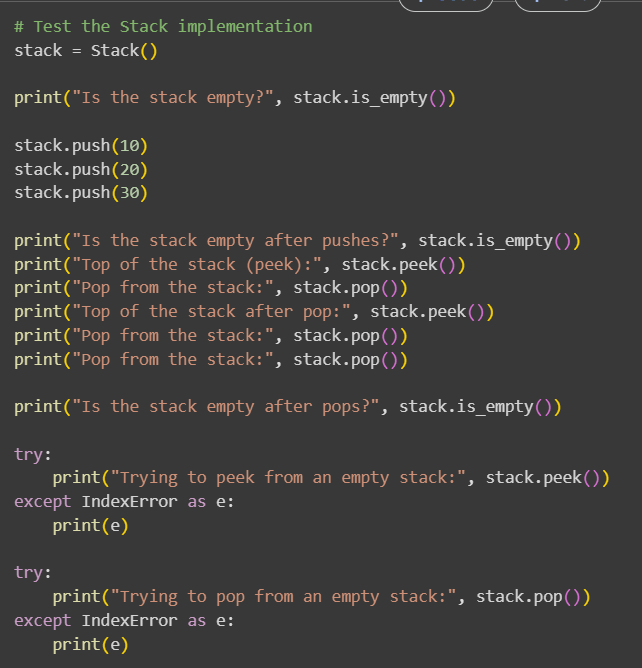
**AI ASSISTED CODING – 11**

**TASK.1**

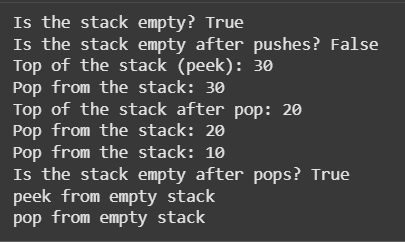
Implementing a Stack (LIFO)  
• Task: Use AI to help implement a Stack class in Python with the  
following operations: push(), pop(), peek(), and is\_empty().  
• Instructions:  
o Ask AI to generate code skeleton with docstrings.  
o Test stack operations using sample data.  
o Request AI to suggest optimizations or alternative  
implementations (e.g., using collections.deque).  
• Expected Output:  
o A working Stack class with proper methods, Google-style  
docstrings, and inline comments for tricky parts.







OUTPUT:



EXPLANATION:

Let's break down the code:

**1. Stack Class (using a Python list):**

* **\_\_init\_\_(self)**: This is the constructor. It initializes an empty list called \_items. This list will be used to store the elements of the stack.
* **push(self, item)**: This method adds an item to the top of the stack. It uses the list's append() method, which adds the item to the end of the list. Since we're using the end of the list as the "top" of the stack, append() effectively pushes the item onto the stack.
* **pop(self)**: This method removes and returns the item from the top of the stack. It first checks if the stack is empty using is\_empty(). If not empty, it uses the list's pop() method. When used without an index, pop() removes and returns the last element of the list, which is exactly what we need for a stack's pop operation. If the stack is empty, it raises an IndexError.
* **peek(self)**: This method returns the item at the top of the stack without removing it. It also checks if the stack is empty. If not empty, it accesses the last element of the list using self.\_items[-1]. This is a standard Python way to access the last element. If the stack is empty, it raises an IndexError.
* **is\_empty(self)**: This method checks if the stack is empty by checking if the length of the \_items list is 0.

**2. Test Code (following the Stack class):**

* This block of code creates an instance of the Stack class and then demonstrates the usage of each method: is\_empty(), push(), peek(), and pop().
* It also includes try...except blocks to show how the IndexError is handled when trying to peek() or pop() from an empty stack.

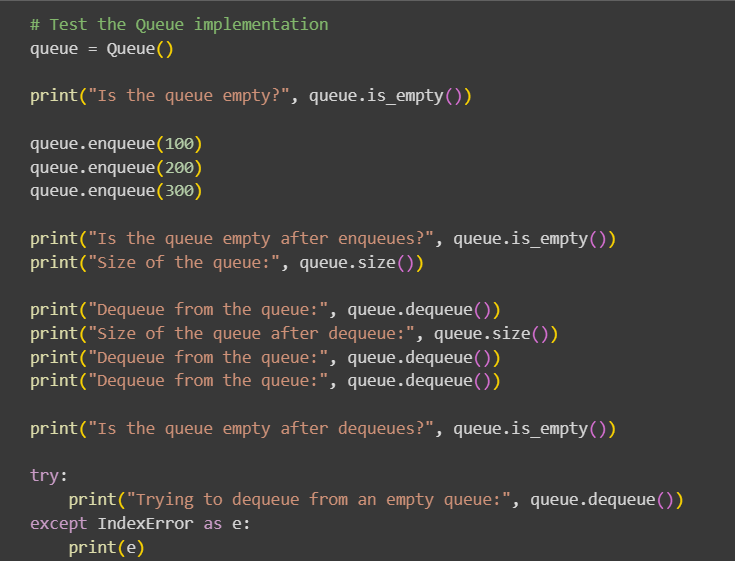
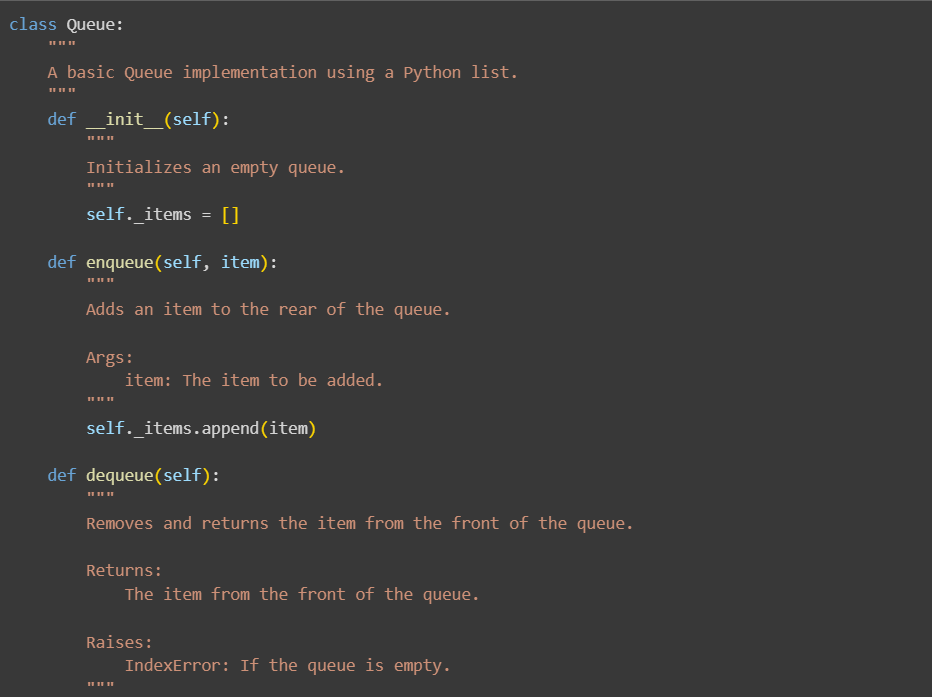
**3. DequeStack Class (using collections.deque):**

* This class provides an alternative implementation of a stack using collections.deque.
* **\_\_init\_\_(self)**: Initializes an empty deque object called \_items.
* **push(self, item)**: Adds an item to the right side of the deque using append().
* **pop(self)**: Removes and returns an item from the right side of the deque using pop().
* **peek(self)**: Returns the item on the right side of the deque without removing it using self.\_items[-1].
* **is\_empty(self)**: Checks if the deque is empty by checking its length.

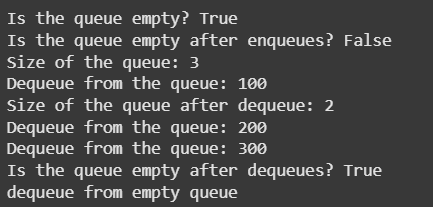
The key difference between the two implementations is the underlying data structure used. While a Python list works, collections.deque is generally more efficient for stack operations (appending and popping from the end) because it's designed for fast appends and pops from both ends, providing O(1) time complexity for these operations.

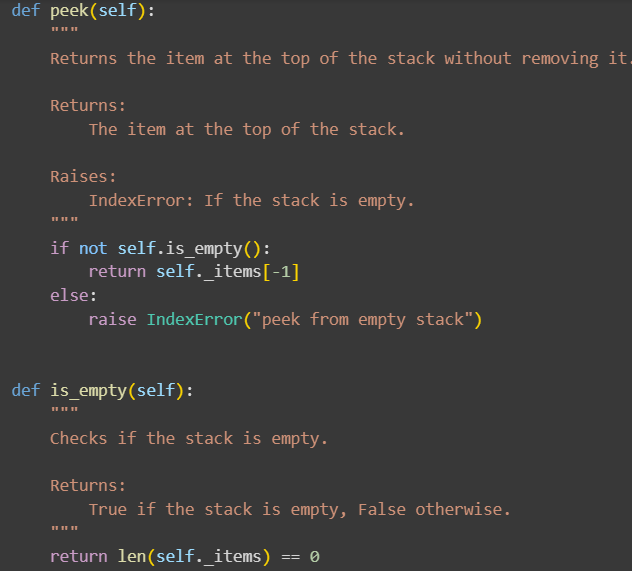
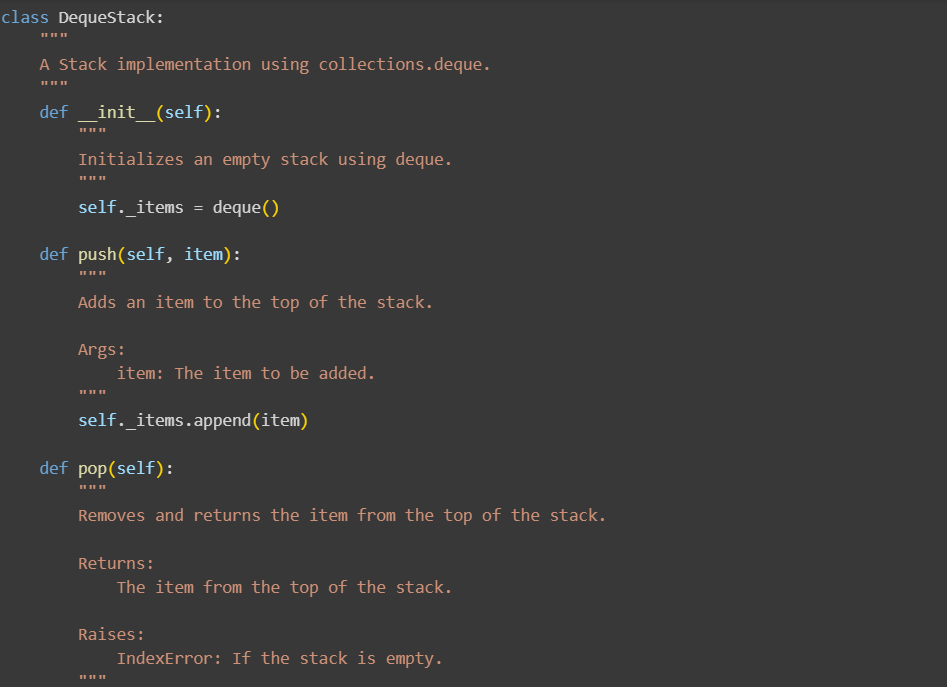
**TASK.2:**

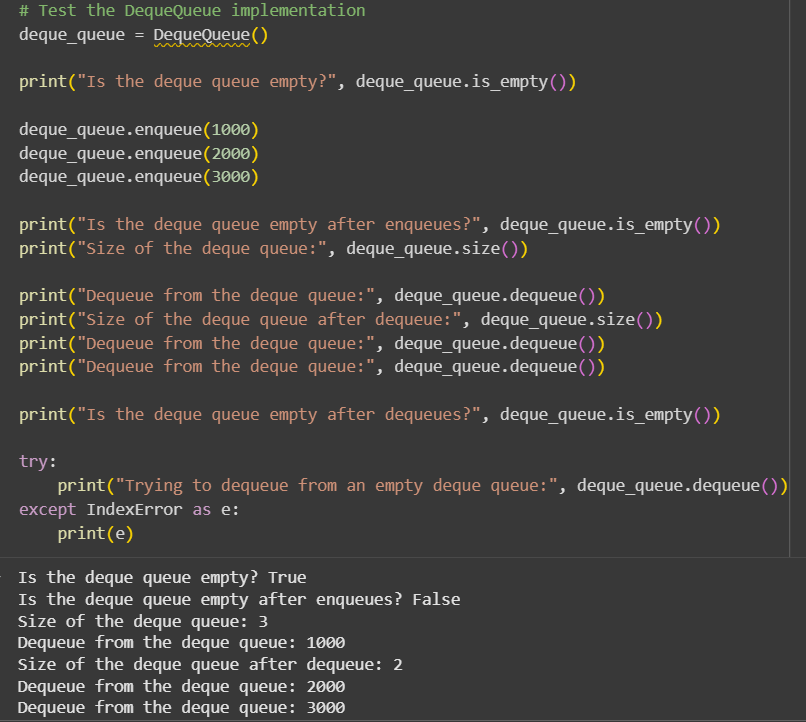
Queue Implementation with Performance Review  
• Task: Implement a Queue with enqueue(), dequeue(), and is\_empty()  
methods.  
• Instructions:  
o First, implement using Python lists.  
o Then, ask AI to review performance and suggest a more  
efficient implementation (using collections.deque)



OUTPUT:







EXPLANATION:

Data Analysis Key Findings

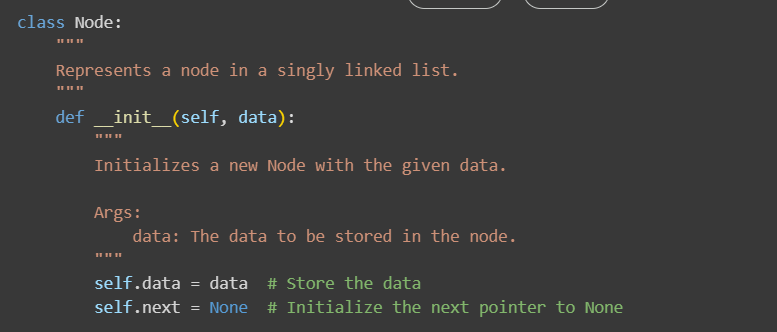
* The QueueList implementation using a Python list has an average time complexity of O(1) for enqueue (append) but a time complexity of O(n) for dequeue (pop(0)) due to the need to shift elements. The is\_empty operation is O(1).
* The QueueDeque implementation using collections.deque achieves a time complexity of O(1) for both enqueue (append) and dequeue (popleft) operations, as well as for is\_empty.
* Testing confirmed that both implementations correctly handle enqueueing, dequeueing in FIFO order, and checking for emptiness.
* Testing also verified that both implementations correctly raise an IndexError when attempting to dequeue from an empty queue.

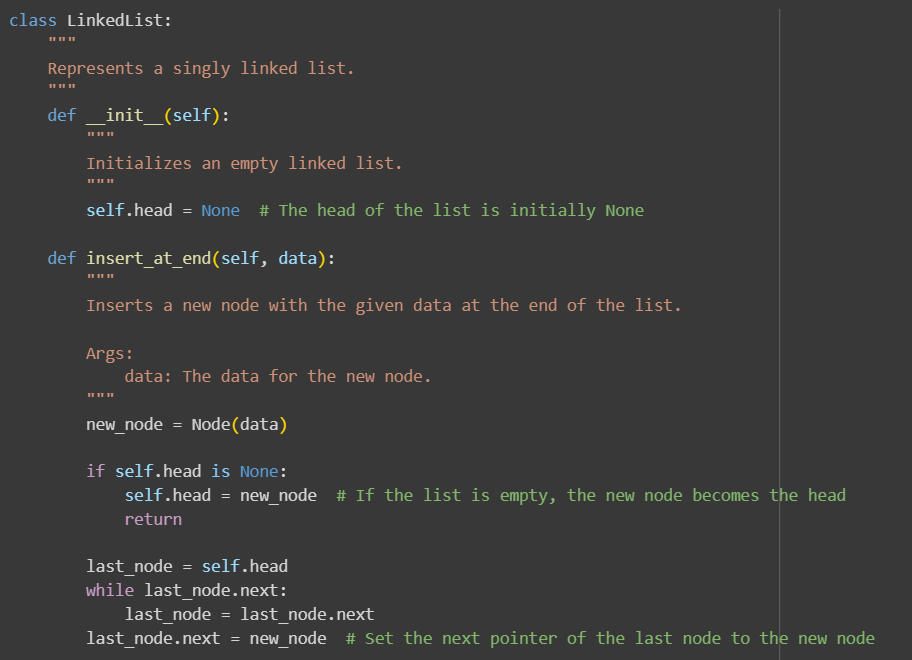
Insights or Next Steps

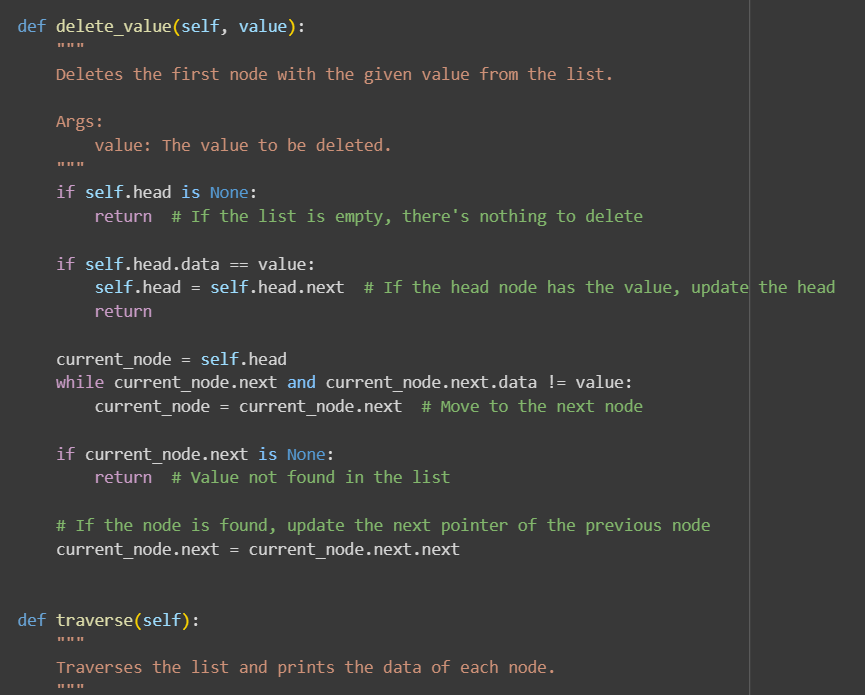
* collections.deque is the significantly more performant and practical choice for implementing a queue in Python due to its O(1) time complexity for both adding and removing elements from either end.
* The list-based implementation is generally unsuitable for performance-critical applications or large datasets due to the O(n) complexity of its dequeue operation.

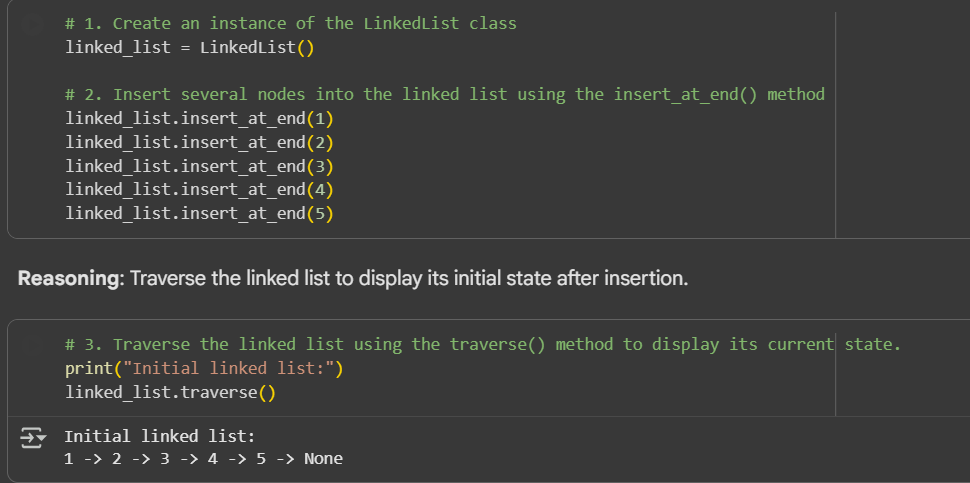
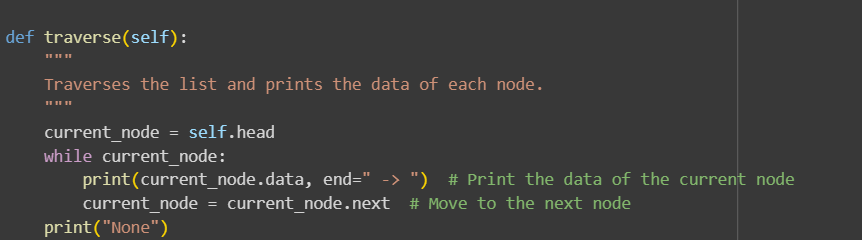
**TASK.3:**

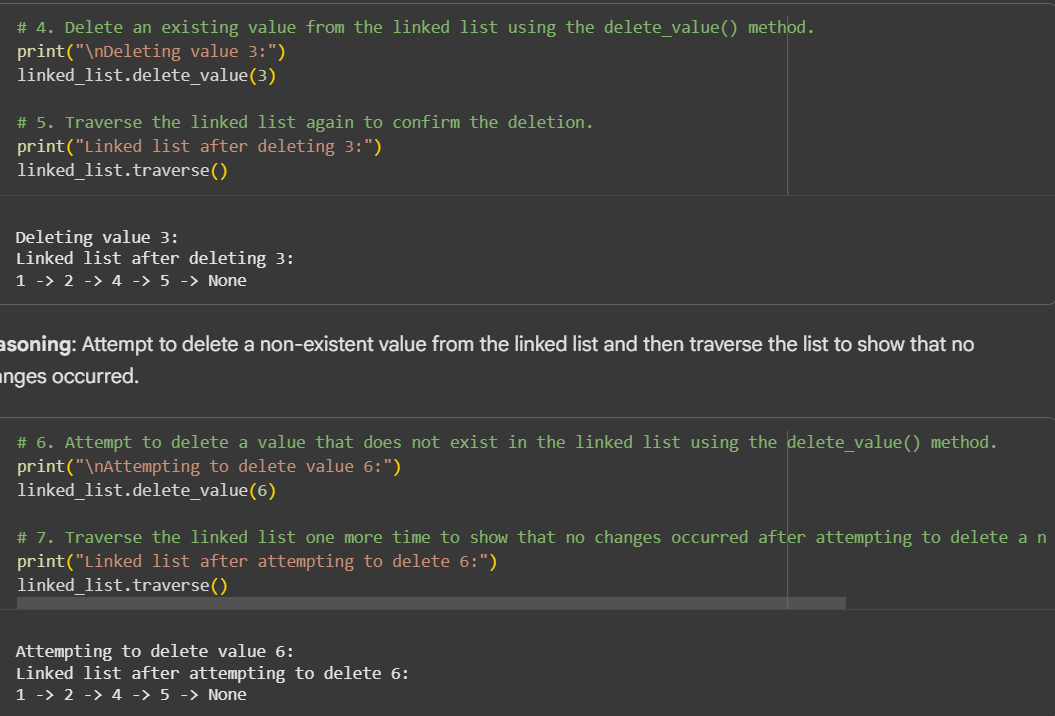
Singly Linked List with Traversal  
• Task: Implement a Singly Linked List with operations:  
insert\_at\_end(), delete\_value(), and traverse().  
• Instructions:  
o Start with a simple class-based implementation (Node,  
LinkedList).  
o Use AI to generate inline comments explaining pointer updates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



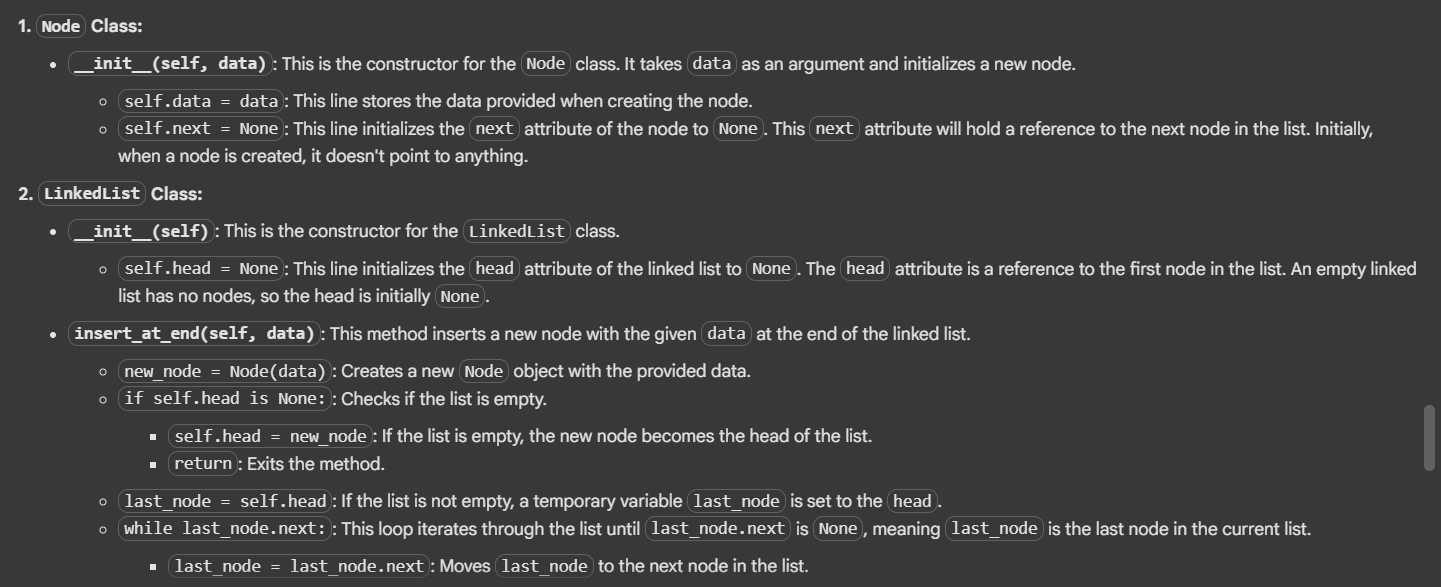


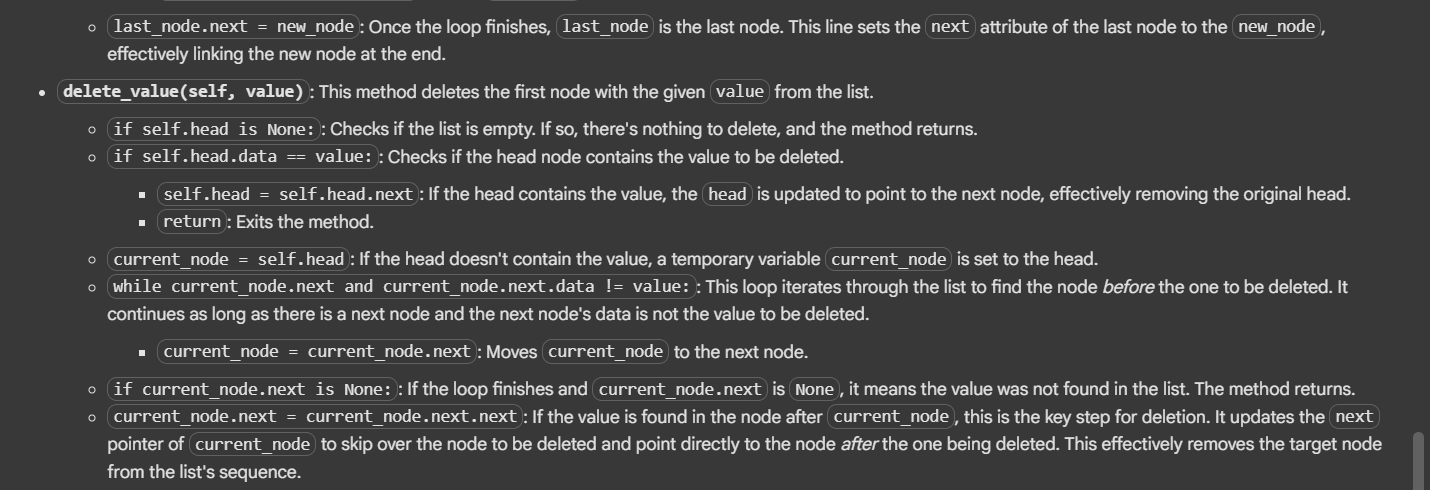


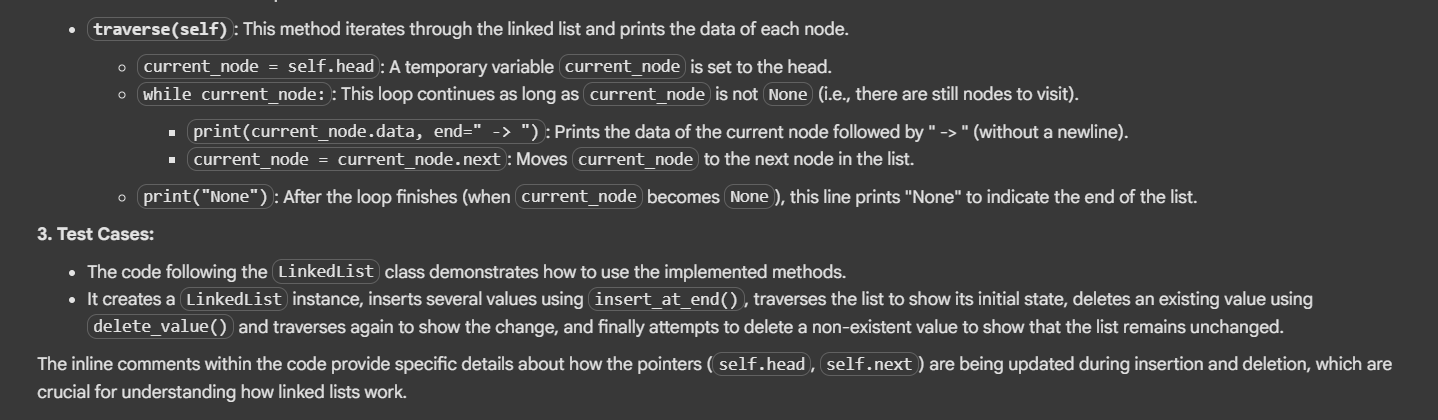




EXPLANATION:



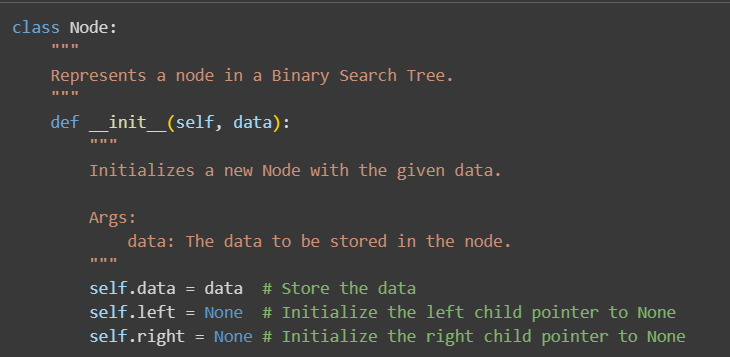
****

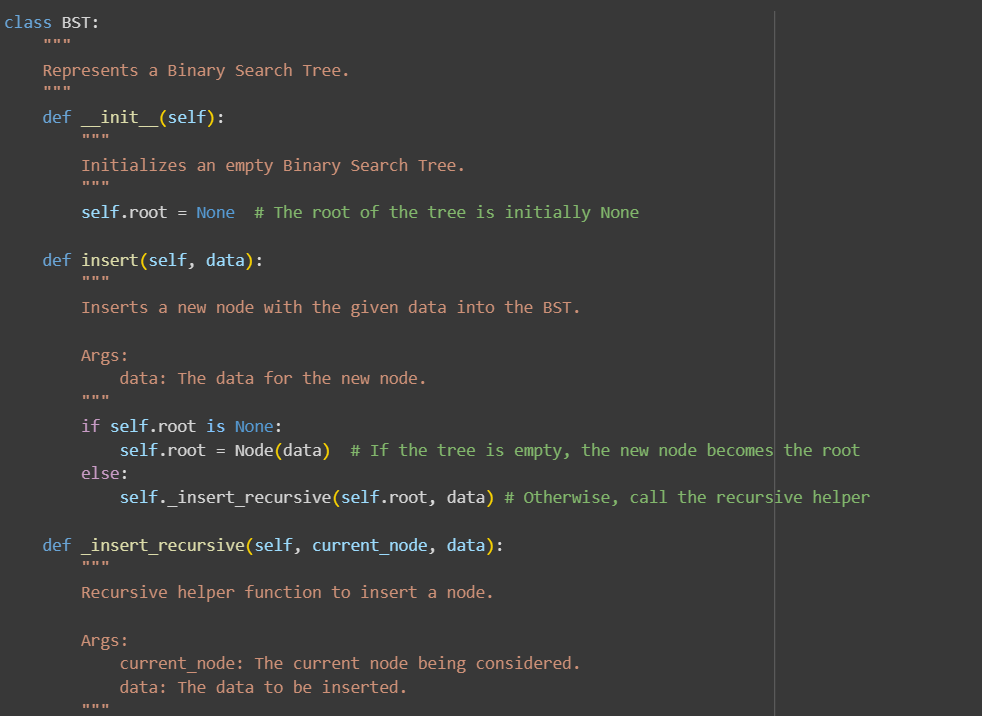


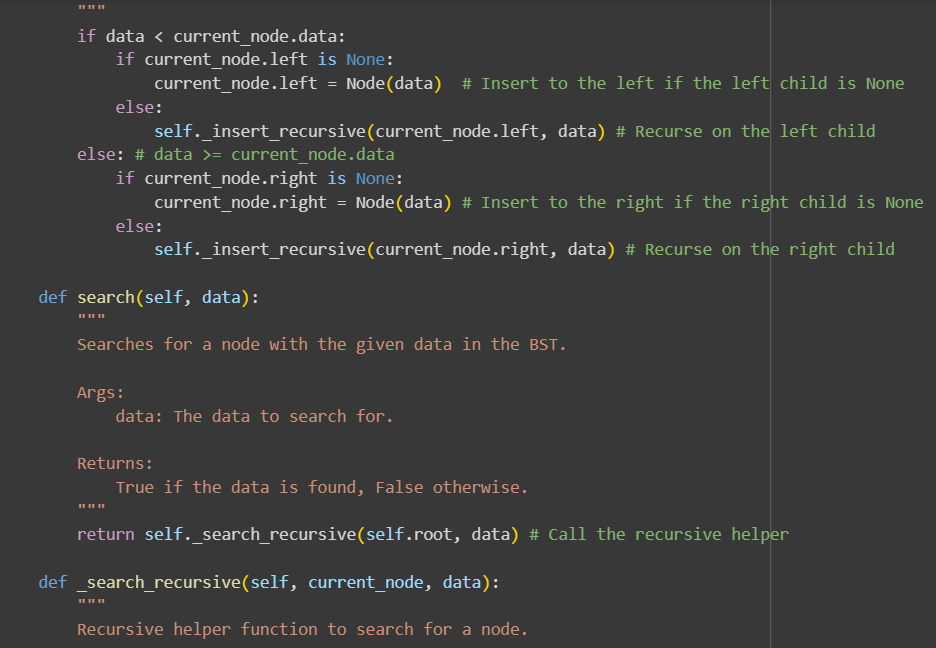
**TASK-4:**

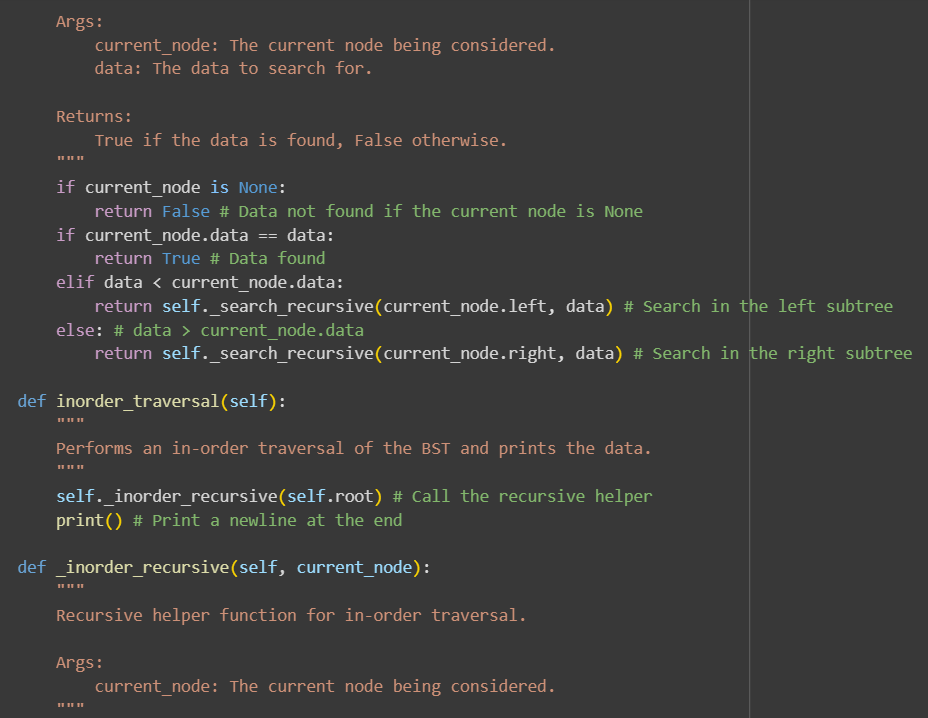
Binary Search Tree (BST)  
• Task: Implement a Binary Search Tree with methods for insert(),  
search(), and inorder\_traversal().  
• Instructions:  
o Provide AI with a partially written Node and BST class.

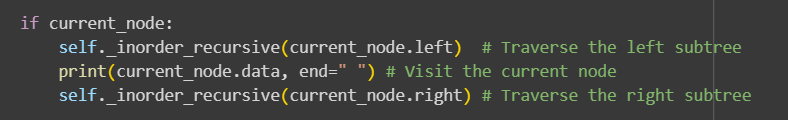
o Ask AI to complete missing methods and add docstrings.  
o Test with a list of integers and compare outputs of search() for  
present vs absent elements.

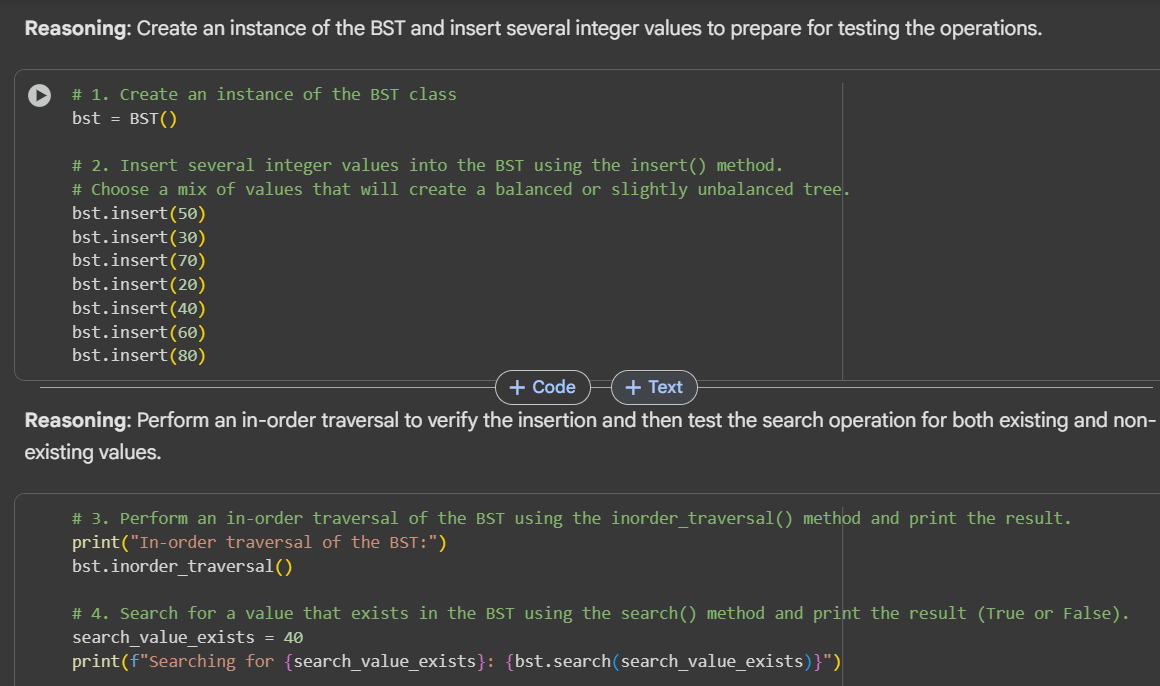


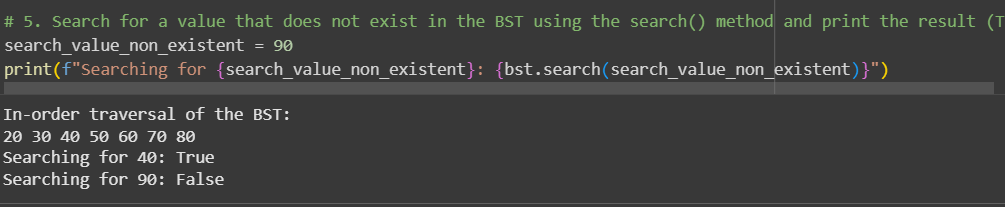












EXPLANATION:

Let's break down the code:

**1. Node Class:**

* **\_\_init\_\_(self, data)**: This is the constructor for the Node class, similar to the one in the linked list but with two child pointers.
  + self.data = data: Stores the data for the node.
  + self.left = None: Initializes the left child pointer to None. This will point to the root of the left subtree.
  + self.right = None: Initializes the right child pointer to None. This will point to the root of the right subtree.

**2. BST Class:**

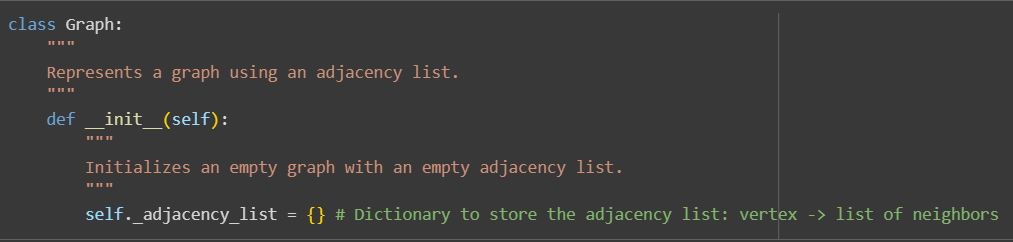
* **\_\_init\_\_(self)**: This is the constructor for the BST class.
  + self.root = None: Initializes the root of the tree to None. An empty tree has no nodes, so the root is initially None.
* **insert(self, data)**: This method inserts a new node with the given data into the BST while maintaining the BST properties (left child data < node data < right child data).
  + if self.root is None:: Checks if the tree is empty.
    - self.root = Node(data): If empty, the new node becomes the root.
  + else:: If not empty, it calls the recursive helper function \_insert\_recursive.
* **\_insert\_recursive(self, current\_node, data)**: This is a private helper function that recursively finds the correct position to insert the new node.
  + if data < current\_node.data:: If the data to be inserted is less than the current node's data, it should go in the left subtree.
    - if current\_node.left is None:: If the left child is None, the new node is inserted here.
    - else:: Otherwise, the recursion continues on the left child (\_insert\_recursive(current\_node.left, data)).
  + else:: If the data is greater than or equal to the current node's data, it should go in the right subtree.
    - if current\_node.right is None:: If the right child is None, the new node is inserted here.
    - else:: Otherwise, the recursion continues on the right child (\_insert\_recursive(current\_node.right, data)).
* **search(self, data)**: This method searches for a node with the given data in the BST. It returns True if found, False otherwise. It calls the recursive helper function \_search\_recursive.
* **\_search\_recursive(self, current\_node, data)**: This is a private helper function that recursively searches for the data.
  + if current\_node is None:: If the current node is None, the data was not found in this branch, so it returns False.
  + if current\_node.data == data:: If the current node's data matches the search data, it returns True (data found).
  + elif data < current\_node.data:: If the search data is less than the current node's data, the search continues recursively in the left subtree (\_search\_recursive(current\_node.left, data)).
  + else:: If the search data is greater than the current node's data, the search continues recursively in the right subtree (\_search\_recursive(current\_node.right, data)).
* **inorder\_traversal(self)**: This method performs an in-order traversal of the BST and prints the data of the nodes. In-order traversal visits the left subtree, then the current node, and then the right subtree. For a BST, this results in printing the data in sorted order. It calls the recursive helper function \_inorder\_recursive.
* **\_inorder\_recursive(self, current\_node)**: This is a private helper function that recursively performs the in-order traversal.
  + if current\_node:: If the current node is not None:
    - self.\_inorder\_recursive(current\_node.left): Recursively traverse the left subtree.
    - print(current\_node.data, end=" "): Visit and print the data of the current node. The end=" " prevents a newline after each number.
    - self.\_inorder\_recursive(current\_node.right): Recursively traverse the right subtree.

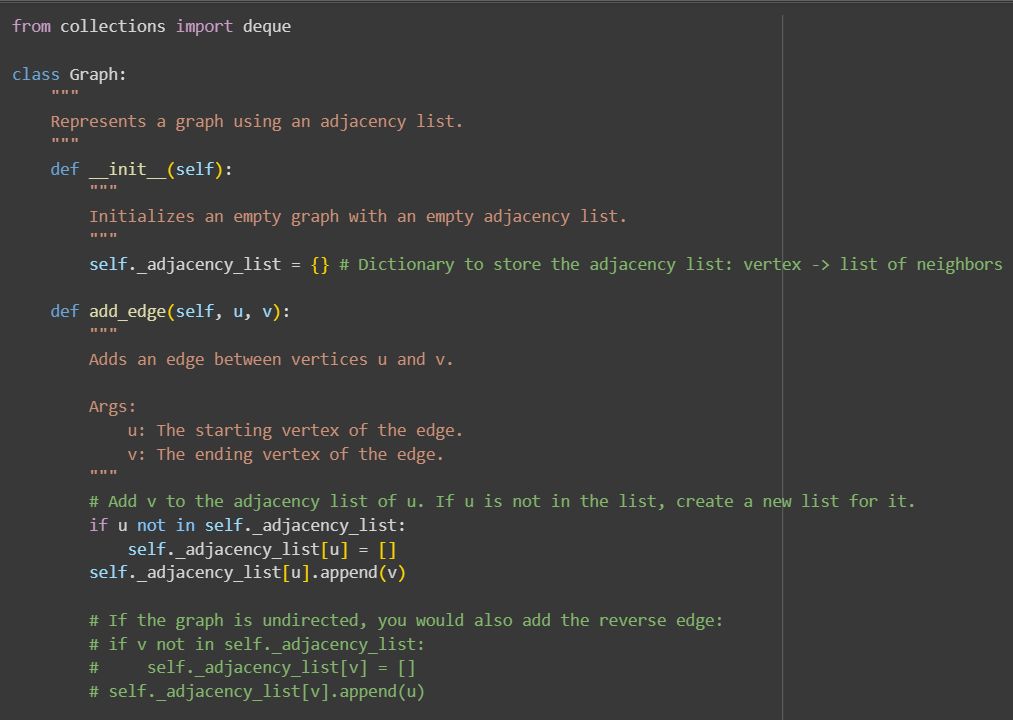
**3. Test Cases:**

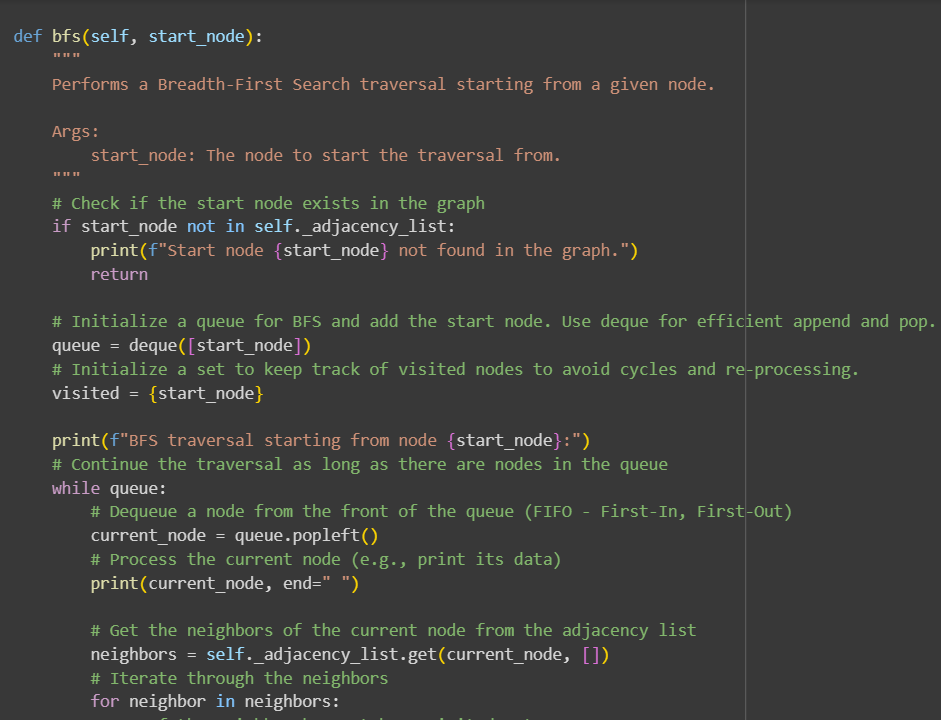
* The code following the BST class demonstrates how to use the implemented methods.
* It creates a BST instance, inserts several integer values (chosen to create a somewhat balanced tree), performs an in-order traversal to show the sorted output, and then tests the search() method with both an existing value and a non-existent value.

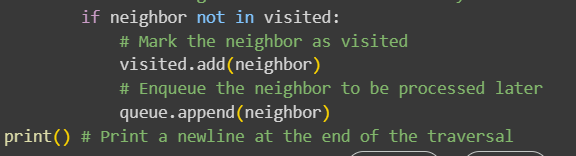
**TASK-5:**

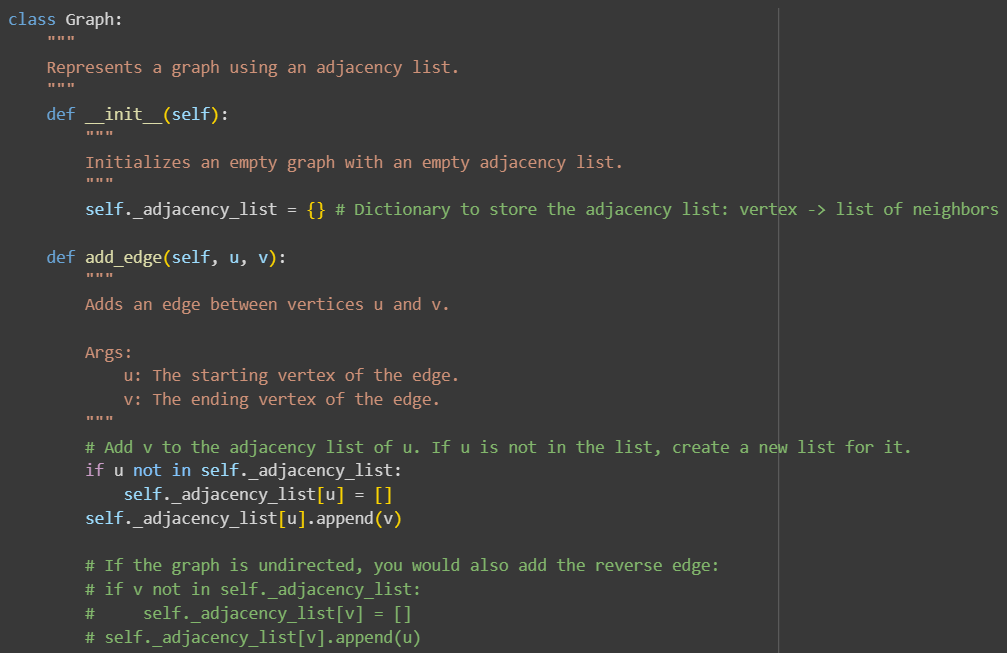
Graph Representation and BFS/DFS Traversal  
• Task: Implement a Graph using an adjacency list, with traversal  
methods BFS() and DFS().  
• Instructions:  
o Start with an adjacency list dictionary.  
o Ask AI to generate BFS and DFS implementations with inline  
comments.  
o Compare recursive vs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

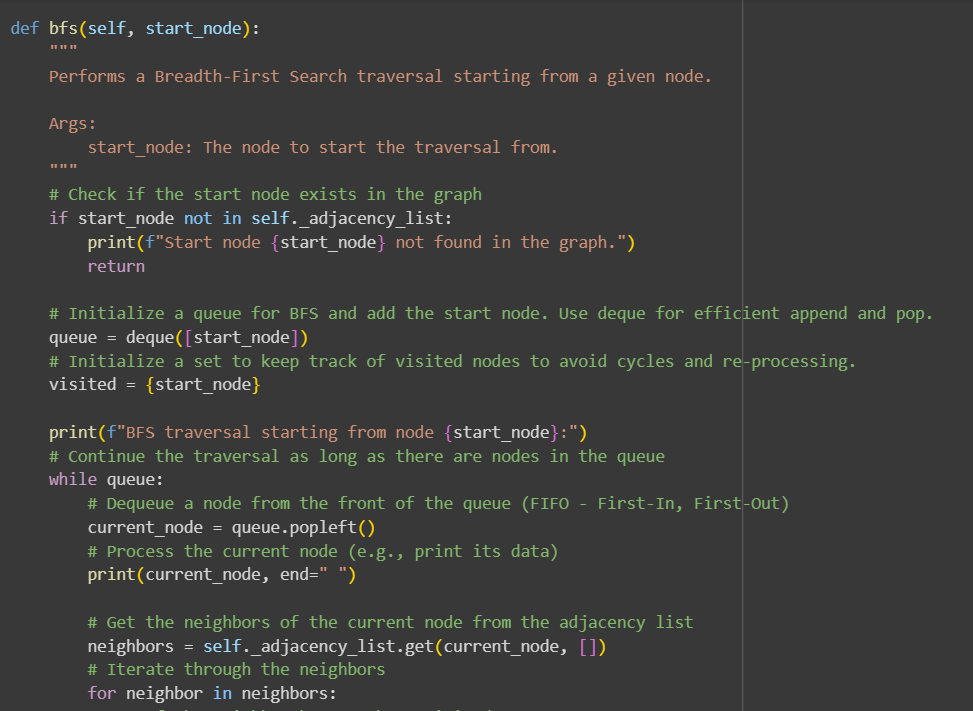


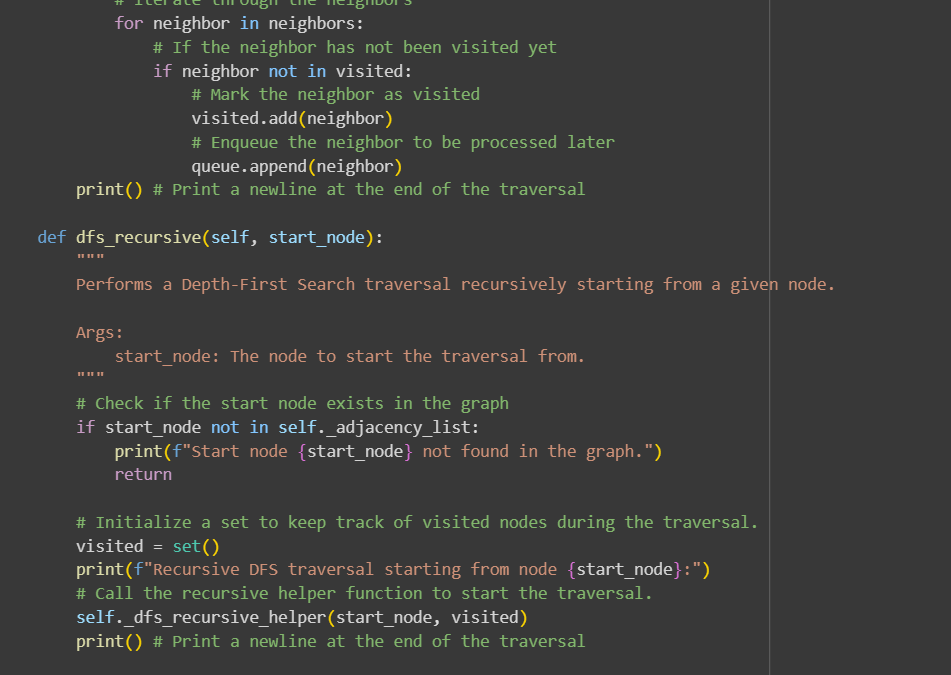


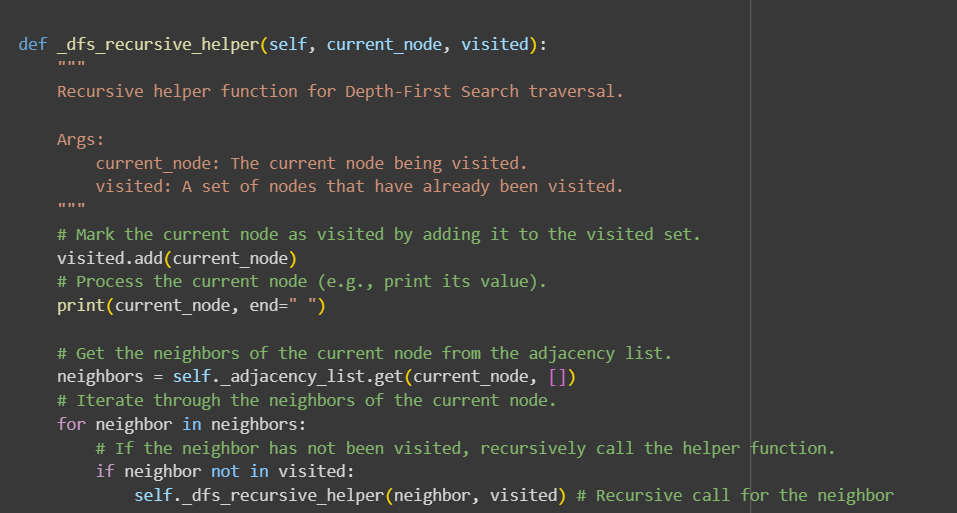


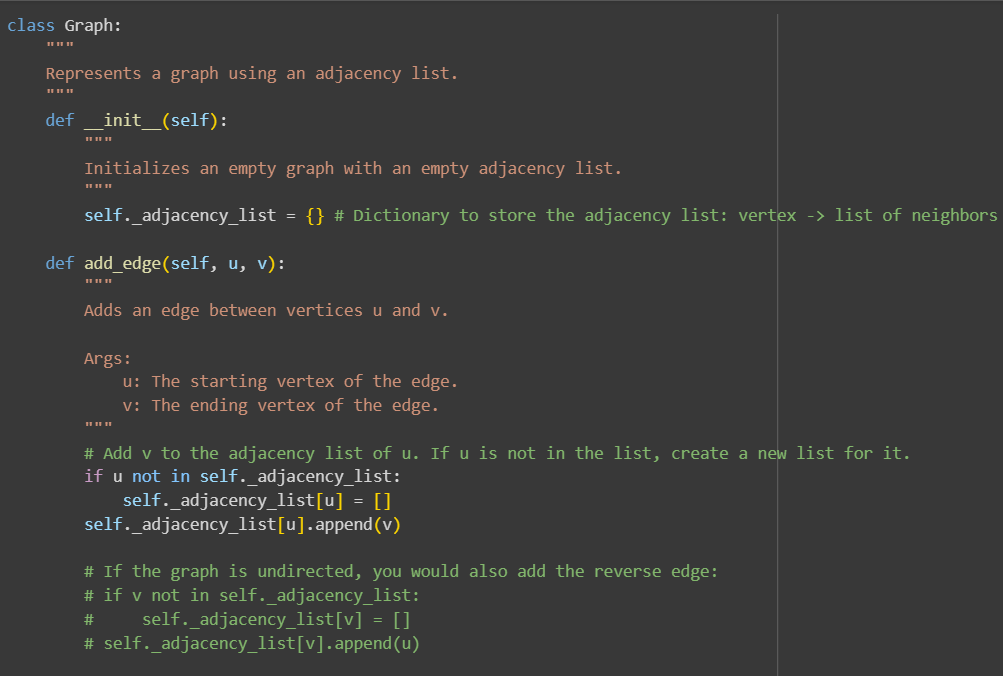


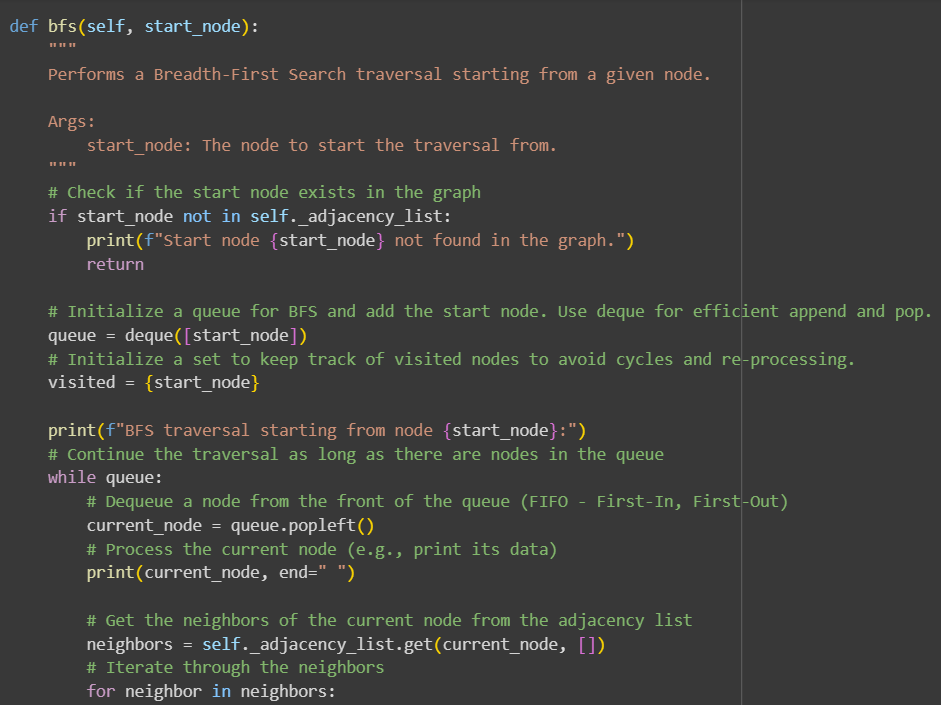


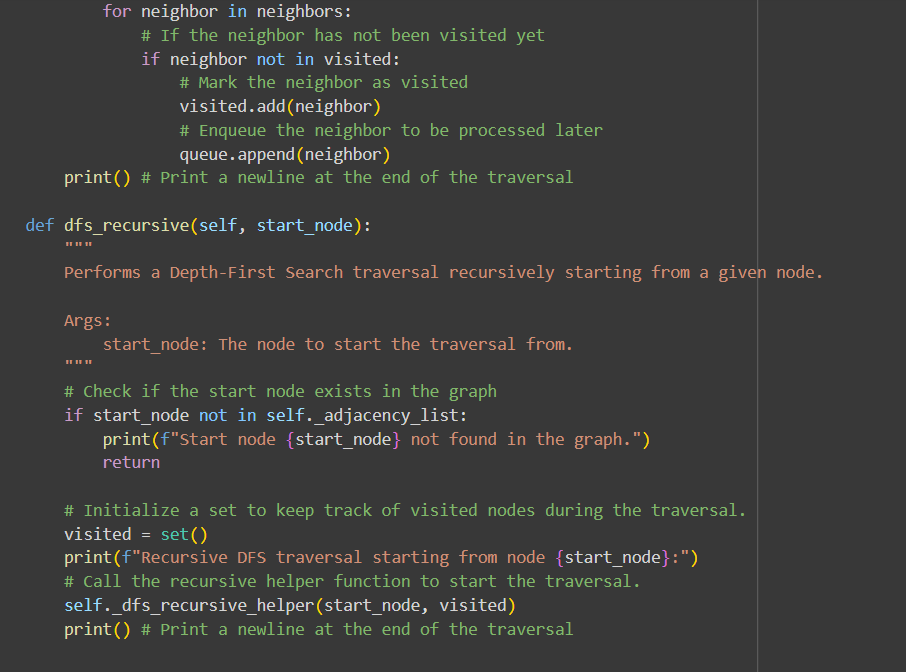


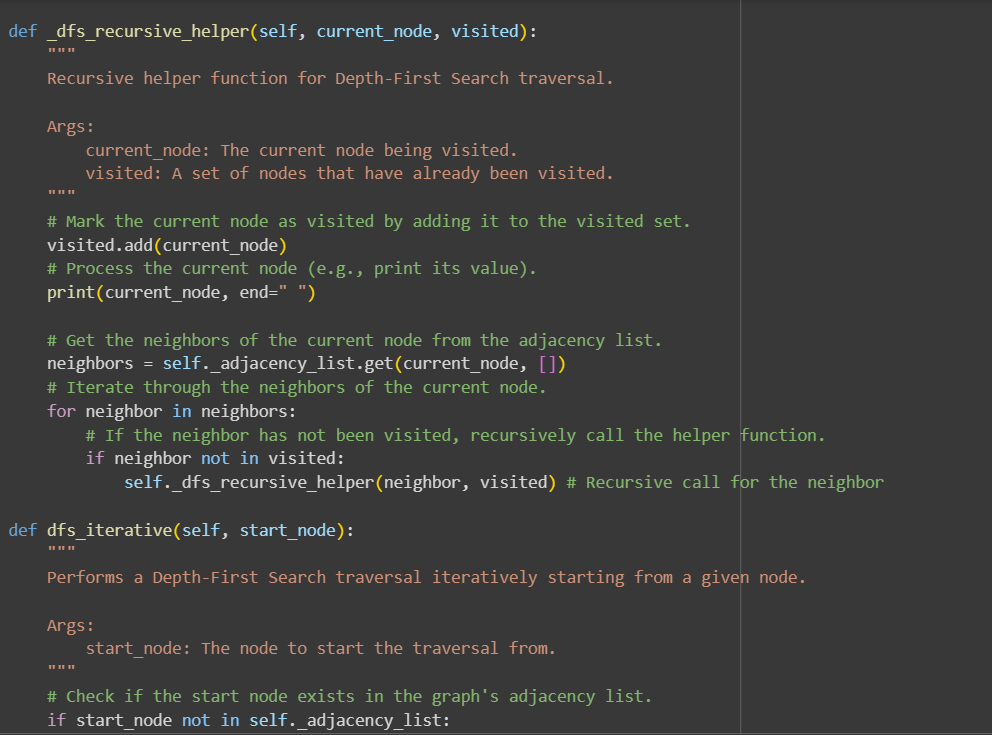


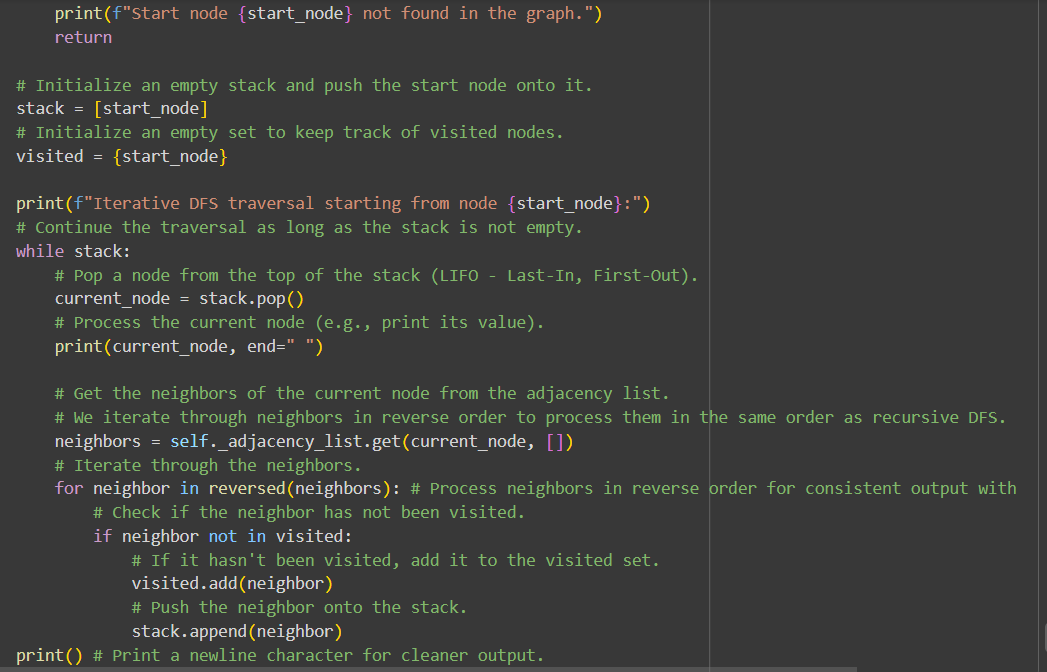


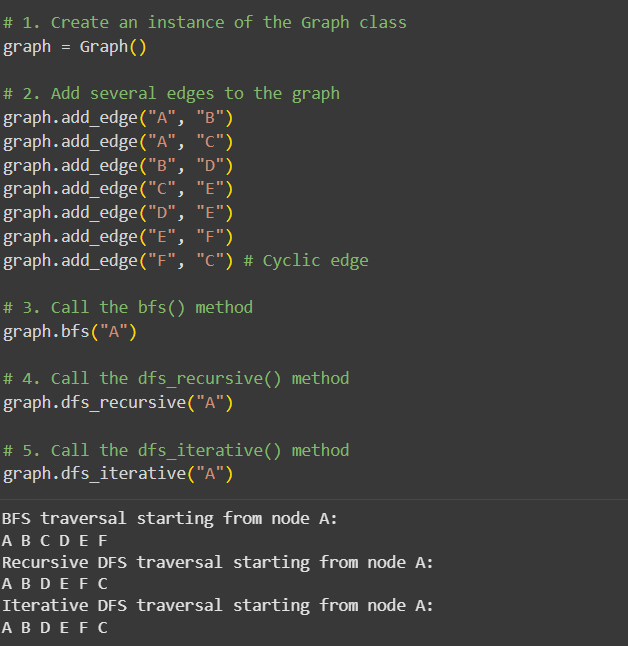


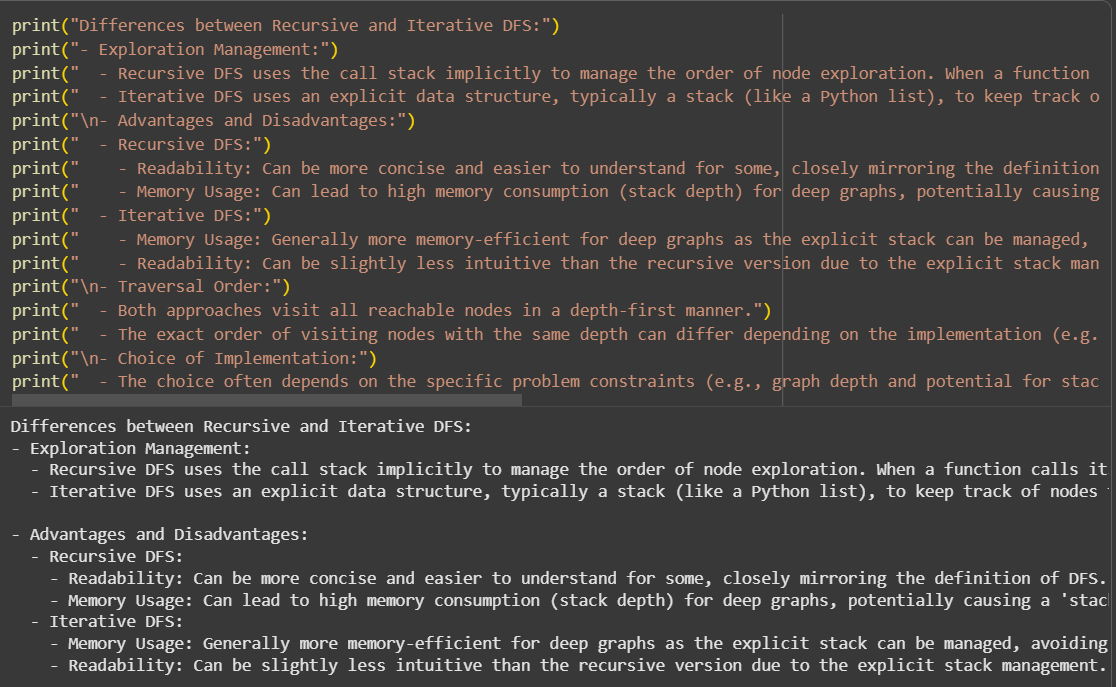


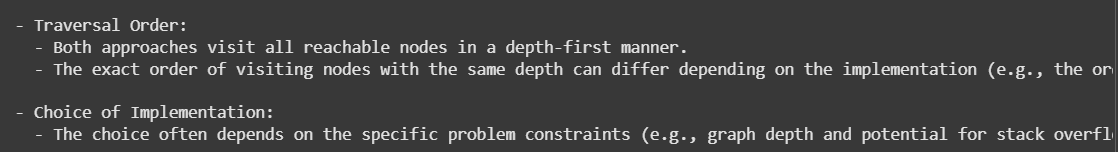












EXPLANATION:

Let's break down the code:

**1. Graph Class:**

* **\_\_init\_\_(self)**: This is the constructor for the Graph class.
  + self.\_adjacency\_list = {}: This line initializes an empty dictionary called \_adjacency\_list. This dictionary will store the graph's structure. The keys of the dictionary represent the vertices (nodes) in the graph, and the values are lists of vertices that are adjacent to the key vertex (i.e., connected by an edge).
* **add\_edge(self, u, v)**: This method adds a directed edge from vertex u to vertex v.
  + if u not in self.\_adjacency\_list:: Checks if vertex u is already a key in the adjacency list.
    - self.\_adjacency\_list[u] = []: If u is not in the list, a new empty list is created for it.
  + self.\_adjacency\_list[u].append(v): Appends vertex v to the list of neighbors for vertex u.
  + The commented-out section shows how you would add the reverse edge (v to u) if the graph were undirected.

**2. bfs(self, start\_node):**

* This method performs a Breadth-First Search traversal starting from start\_node. BFS explores the graph layer by layer.
  + if start\_node not in self.\_adjacency\_list:: Checks if the start node exists in the graph.
  + queue = deque([start\_node]): Initializes a deque (double-ended queue) and adds the start\_node. A queue is used in BFS to process nodes in a FIFO (First-In, First-Out) manner.
  + visited = {start\_node}: Initializes a set to keep track of visited nodes to prevent infinite loops in graphs with cycles and avoid processing nodes multiple times. The start\_node is marked as visited.
  + while queue:: The loop continues as long as there are nodes in the queue.
    - current\_node = queue.popleft(): Removes the node from the front of the queue.
    - print(current\_node, end=" "): Processes (prints) the current node.
    - neighbors = self.\_adjacency\_list.get(current\_node, []): Gets the list of neighbors for the current\_node. get() is used with a default empty list ([]) in case the node has no neighbors defined in the adjacency list.
    - for neighbor in neighbors:: Iterates through the neighbors.
      * if neighbor not in visited:: Checks if the neighbor has been visited.
        + visited.add(neighbor): If not visited, it's marked as visited.
        + queue.append(neighbor): The unvisited neighbor is added to the end of the queue to be processed later.

**3. dfs\_recursive(self, start\_node):**

* This method performs a Depth-First Search traversal recursively starting from start\_node. DFS explores as far as possible along each branch before backtracking.
  + if start\_node not in self.\_adjacency\_list:: Checks if the start node exists.
  + visited = set(): Initializes a set to keep track of visited nodes during the recursive traversal.
  + self.\_dfs\_recursive\_helper(start\_node, visited): Calls the recursive helper function to start the traversal.
* **\_dfs\_recursive\_helper(self, current\_node, visited):** This is the private recursive helper function for DFS.
  + visited.add(current\_node): Marks the current node as visited.
  + print(current\_node, end=" "): Processes (prints) the current node.
  + neighbors = self.\_adjacency\_list.get(current\_node, []): Gets the neighbors.
  + for neighbor in neighbors:: Iterates through the neighbors.
    - if neighbor not in visited:: If a neighbor hasn't been visited:
      * self.\_dfs\_recursive\_helper(neighbor, visited): Recursively calls the helper function for the unvisited neighbor. This is where the "depth-first" exploration happens as the function calls go deeper into the graph branches.

**4. dfs\_iterative(self, start\_node):**

* This method performs a Depth-First Search traversal iteratively using a stack.
  + if start\_node not in self.\_adjacency\_list:: Checks if the start node exists.
  + stack = [start\_node]: Initializes a Python list to act as a stack and pushes the start\_node onto it.
  + visited = {start\_node}: Initializes a set to keep track of visited nodes. The start\_node is marked as visited.
  + while stack:: The loop continues as long as there are nodes in the stack.
    - current\_node = stack.pop(): Removes the node from the top of the stack (LIFO - Last-In, First-Out).
    - print(current\_node, end=" "): Processes (prints) the current node.
    - neighbors = self.\_adjacency\_list.get(current\_node, []): Gets the neighbors.
    - for neighbor in reversed(neighbors):: Iterates through the neighbors in reverse order. This is done to ensure that when neighbors are pushed onto the stack, they are popped and processed in the same order as they would be in the recursive DFS (assuming neighbors are added to the adjacency list in a consistent order).
      * if neighbor not in visited:: If a neighbor hasn't been visited:
        + visited.add(neighbor): Marks the neighbor as visited.
        + stack.append(neighbor): Pushes the unvisited neighbor onto the stack.

**5. Test Cases:**

* This code block creates a Graph instance, adds several edges to form a sample directed graph with a cycle (F to C), and then calls bfs(), dfs\_recursive(), and dfs\_iterative() starting from node "A" to demonstrate the traversal outputs.

**6. Comparison between Recursive and Iterative DFS:**

* The final markdown cell provides a discussion comparing the two DFS implementations.
  + **Exploration Management:** Explains how recursive DFS uses the implicit call stack, while iterative DFS uses an explicit stack.
  + **Advantages and Disadvantages:** Discusses readability (recursive can be more concise) and memory usage (iterative can be more memory-efficient for deep graphs to avoid stack overflow).
  + **Traversal Order:** Notes that both visit all reachable nodes depth-first, but the exact order at the same depth can differ based on implementation details.
  + **Choice of Implementation:** Mentions that the choice depends on factors like graph depth, performance needs, and coding style.

This code provides a solid foundation for working with graphs and understanding the fundamental BFS and DFS traversal algorithms.