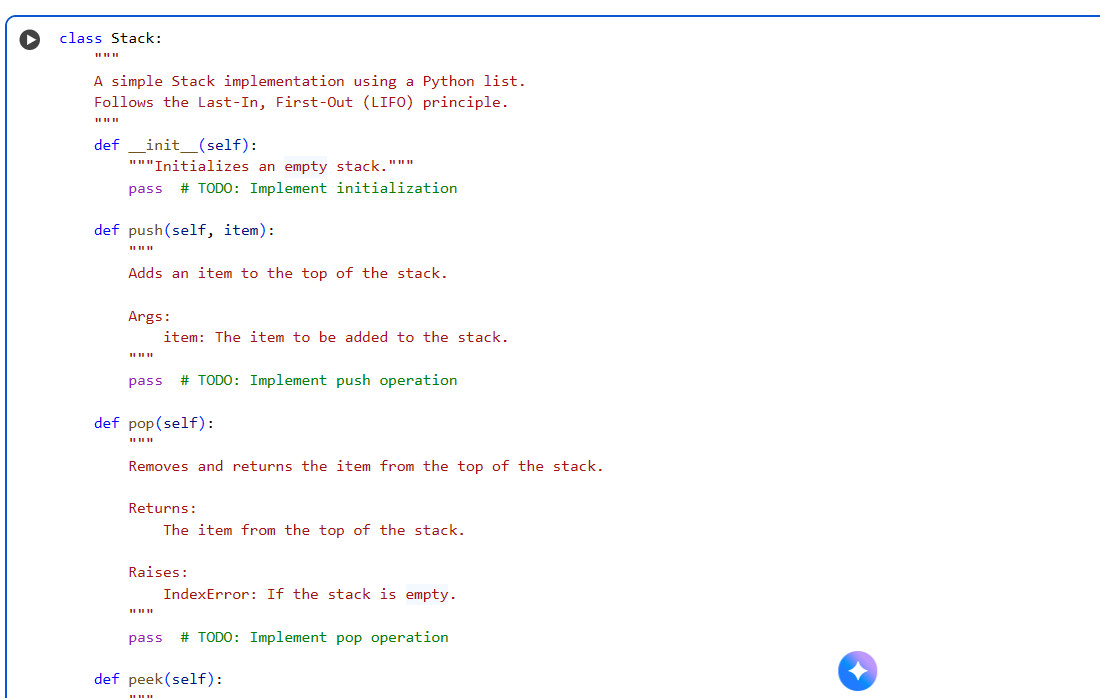
**ASSIGNMENT-11**

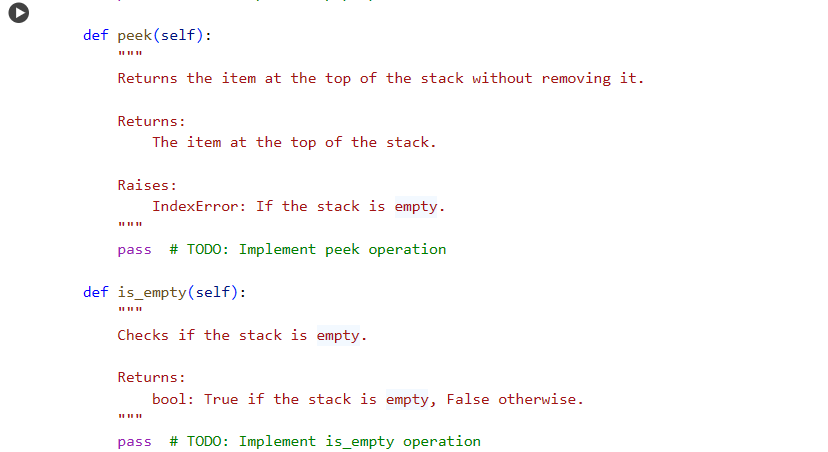
**NAME: G.SAIRAJ**

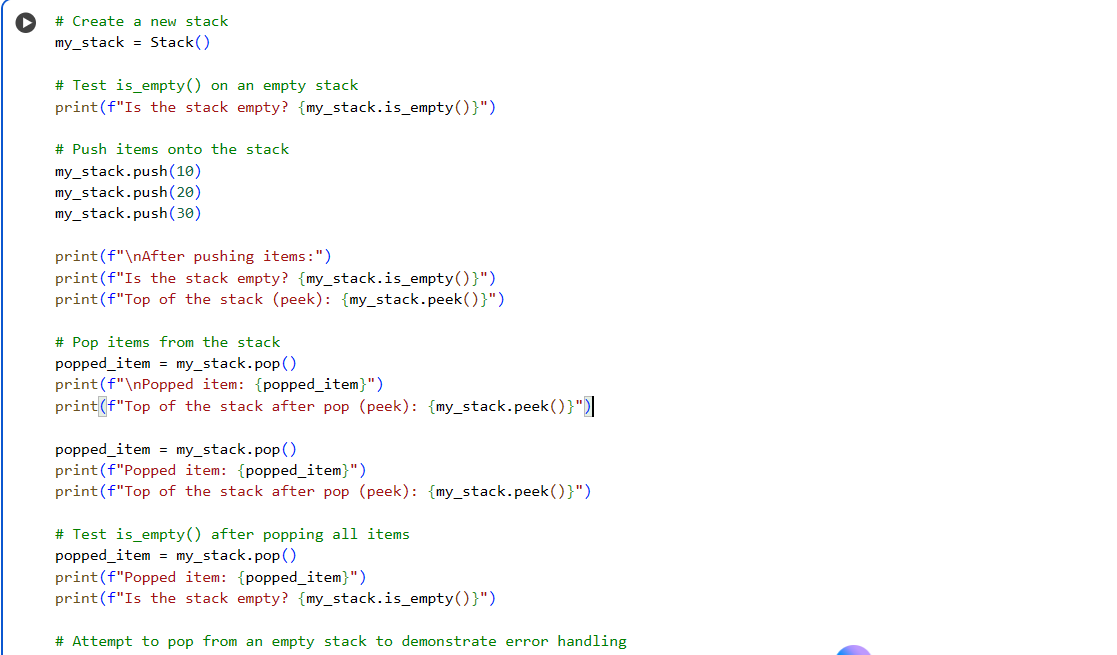
**HT NO:**2403A52149 **BATCH:** 24BTCAIAIB06

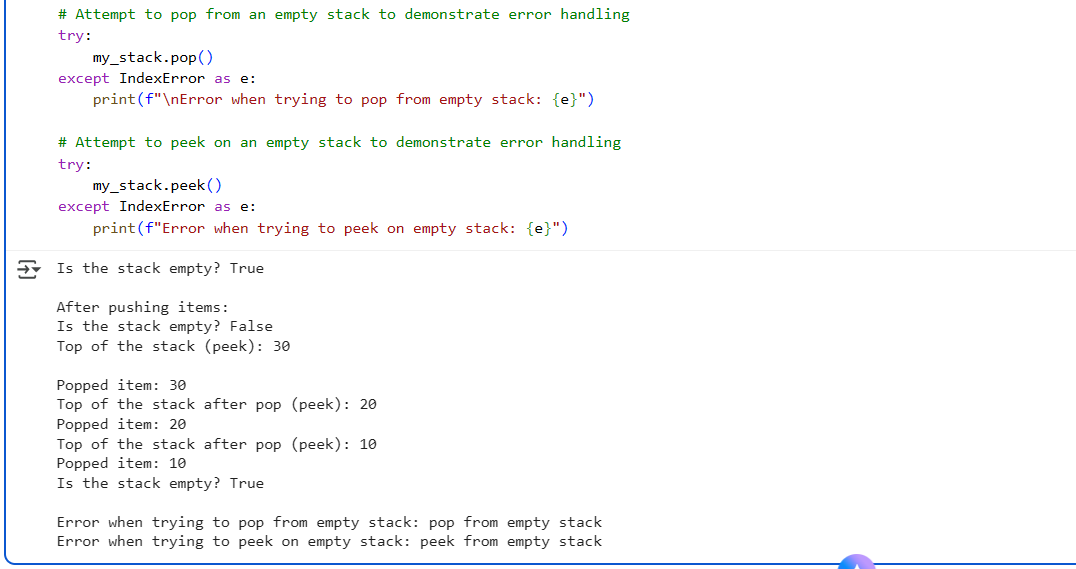
**Task 1:** Implementing a Stack (LIFO)  
• Task: Use AI to help implement a Stack class in Python with the  
following operations: push(), pop(), peek(), and is\_empty().  
• Instructions:  
o Ask AI to generate code skeleton with docstrings.  
o Test stack operations using sample data.  
o Request 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

**CODE AND OUTPUT:**

****

****

****

****

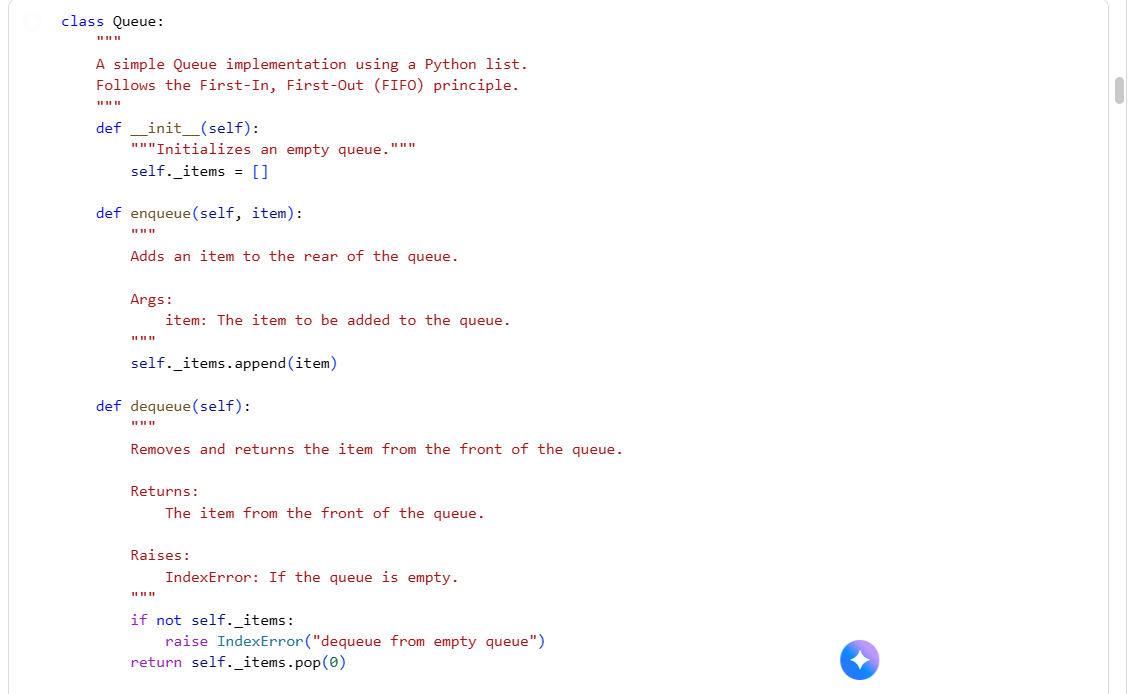
**EXPLANATION:**

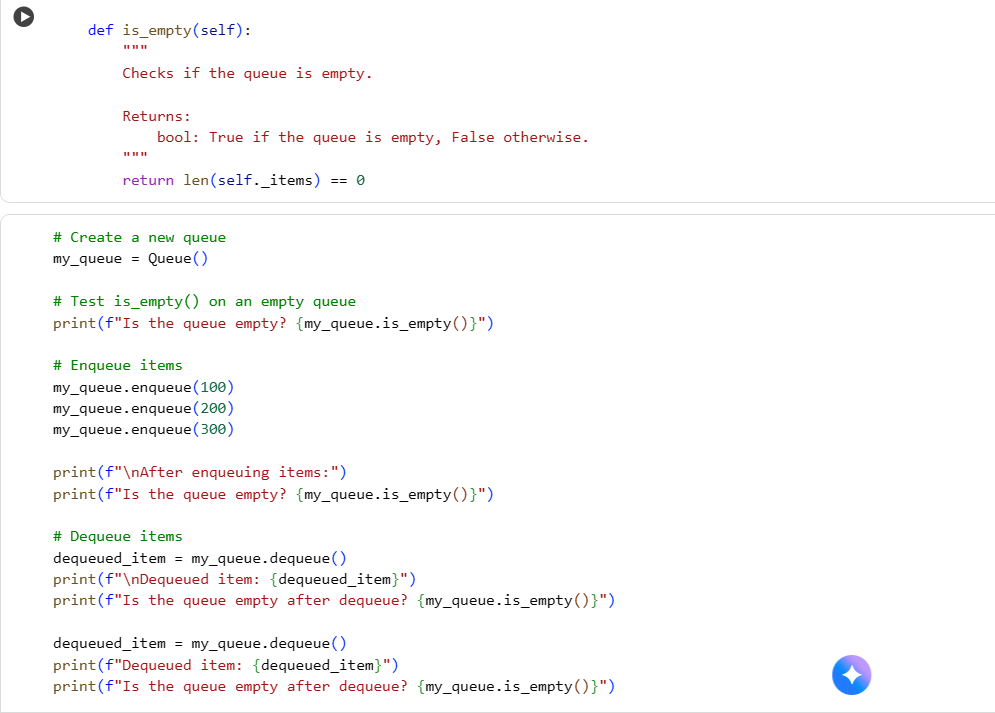
* **\_\_init\_\_(self)**: This is the constructor that initializes an empty stack by creating an empty list called \_items. This list is used to store the elements of the stack.
* **push(self, item)**: This method adds an item to the top of the stack. It uses the append() method of the underlying list, which adds the item to the end of the list. In a stack, the end of the list is considered the top.
* **pop(self)**: This method removes and returns the item from the top of the stack. It first checks if the stack is empty. If it is, it raises an IndexError. Otherwise, it uses the pop() method of the list to remove and return the last element (which is the top of the stack).
* **peek(self)**: This method returns the item at the top of the stack without removing it. Similar to pop(), it checks if the stack is empty and raises an IndexError if it is. Otherwise, it returns the last element of the list (self.\_items[-1]) without removing it.
* **is\_empty(self)**: This method checks if the stack is empty by checking if the length of the \_items list is 0. It returns True if the stack is empty and False otherwise.

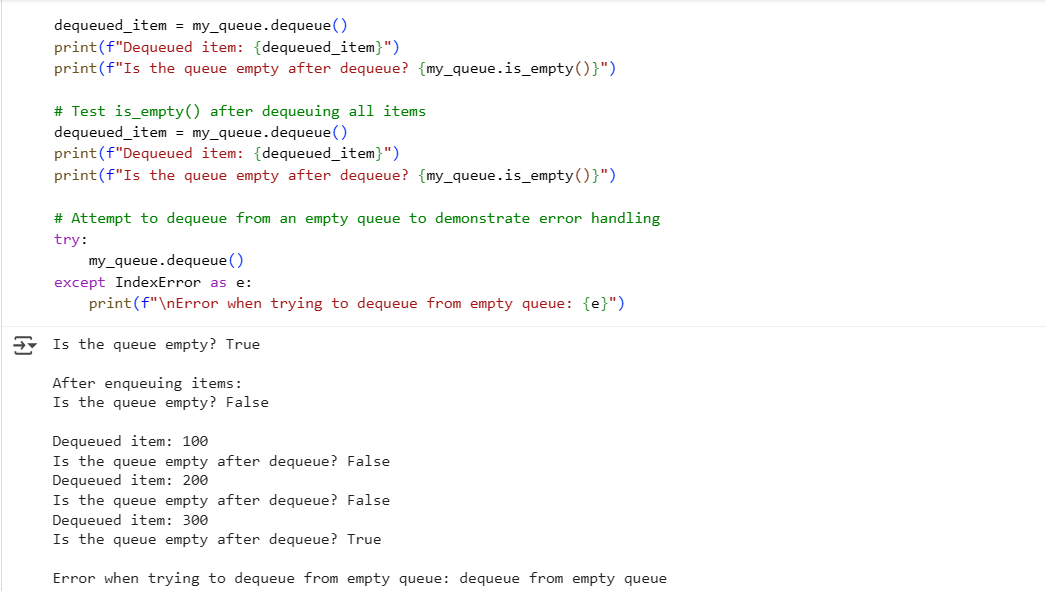
This implementation uses a Python list as the underlying data structure. The append() method for pushing and pop() for removing from the end are efficient operations for a list. However, pop(0) which would be used for the front of a list (like in a queue implementation) is less efficient. For a stack, this list-based approach works well because we are always operating on the end of the list (the "top" of the stack).

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

**CODE AND OUTPUT:**

****

****

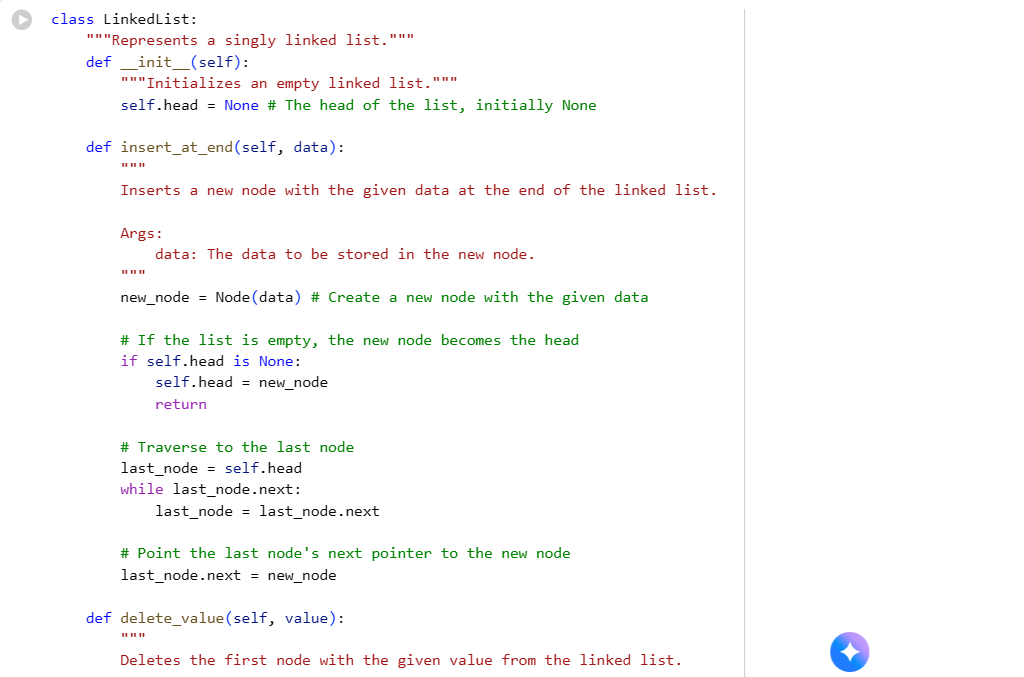
****

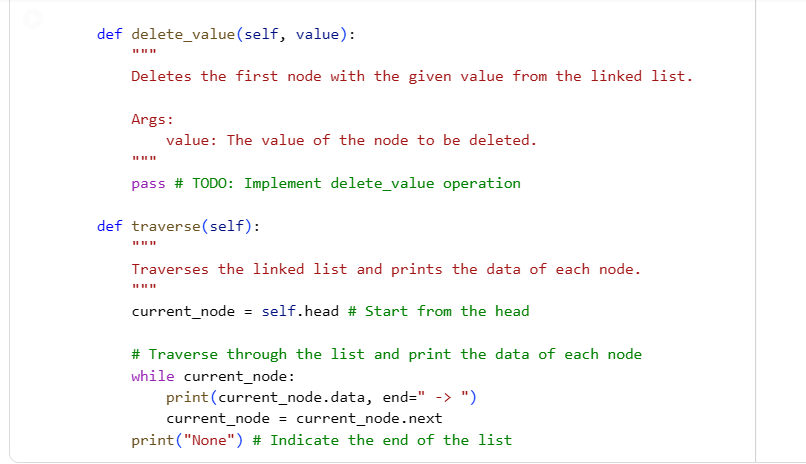
* **EXPLANATION:  
  \_\_init\_\_(self)**: This is the constructor. It initializes an empty queue by creating an empty list called \_items. This list will hold the elements of the queue.
* **enqueue(self, item)**: This method adds an item to the rear of the queue. It uses the append() method of the list, which adds the item to the end of the list. In this list-based queue implementation, the end of the list represents the rear of the queue.
* **dequeue(self)**: This method removes and returns the item from the front of the queue. It first checks if the queue is empty by looking at self.\_items. If the list is empty, it raises an IndexError with the message "dequeue from empty queue". If the queue is not empty, it uses self.\_items.pop(0) to remove and return the element at index 0, which is the first element added to the list and thus the front of the queue.
* **is\_empty(self)**: This method checks if the queue is empty by comparing the length of the \_items list to 0. It returns True if the list is empty (the queue is empty) and False otherwise.

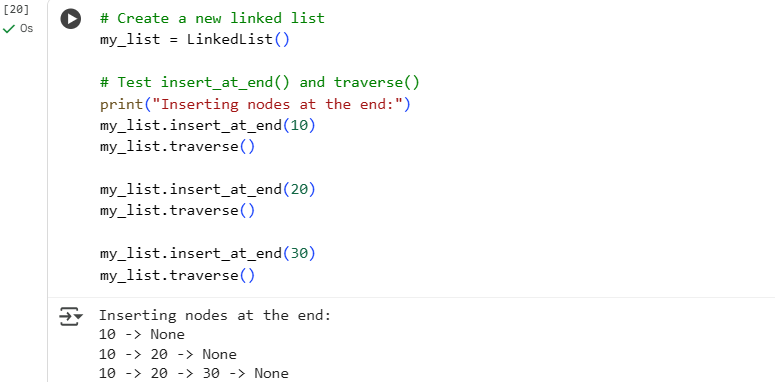
While this implementation is simple, using pop(0) on a Python list can be inefficient for large queues because removing the first element requires shifting all subsequent elements to the left. This is why for performance-critical applications, using collections.deque is often preferred for implementing queues.

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

**CODE AND OUTPUT:**

****

****

****

**EXPLANATION:**

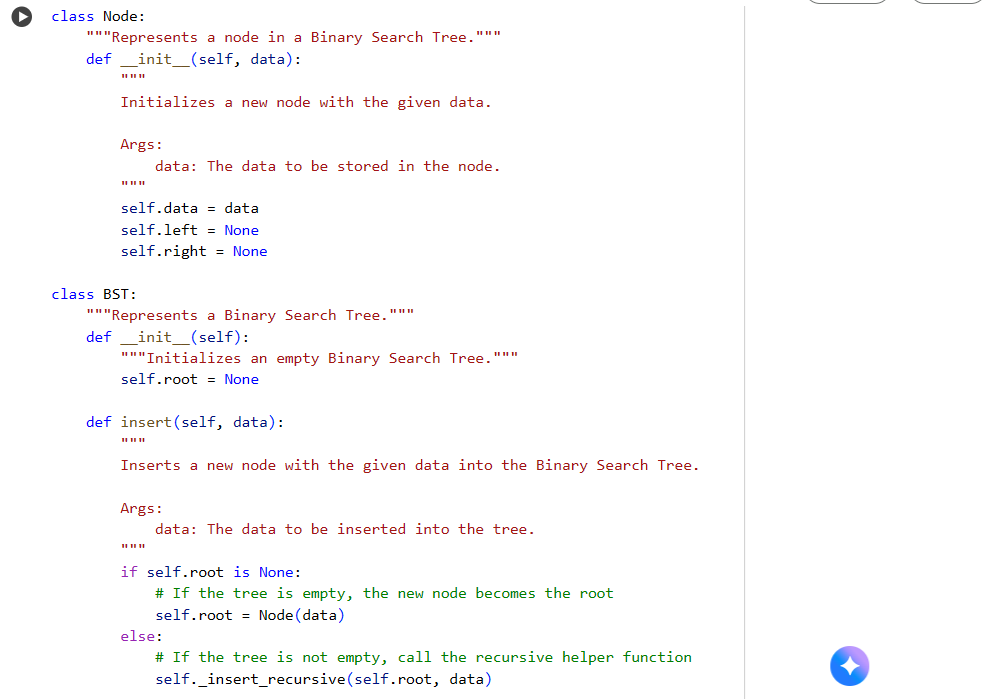
* **\_\_init\_\_(self)**: This is the constructor for the LinkedList class. It initializes an empty linked list by setting the head attribute to None. The head attribute is a reference to the first node in the list.
* **insert\_at\_end(self, data)**: This method adds a new node containing the given data to the end of the linked list.
  + It first creates a new\_node instance with the provided data.
  + If the list is empty (self.head is None), the new\_node becomes the head of the list.
  + If the list is not empty, it traverses the list starting from the head until it reaches the last node (the node whose next attribute is None).
  + Finally, it sets the next attribute of the last node to the new\_node, effectively linking the new node to the end of the list.
* **delete\_value(self, value)**: This method is currently a placeholder (pass). It is intended to delete the first node in the list that contains the specified value. This method would require traversing the list to find the node to delete and updating the pointers of the preceding and succeeding nodes.
* **traverse(self)**: This method iterates through the linked list starting from the head.
  + It uses a current\_node variable, initialized to self.head.
  + It enters a while loop that continues as long as current\_node is not None.
  + Inside the loop, it prints the data of the current\_node followed by " -> ".
  + It then moves to the next node by updating current\_node to current\_node.next.
  + After the loop finishes (meaning it has reached the end of the list), it prints "None" to indicate the end of the linked list.

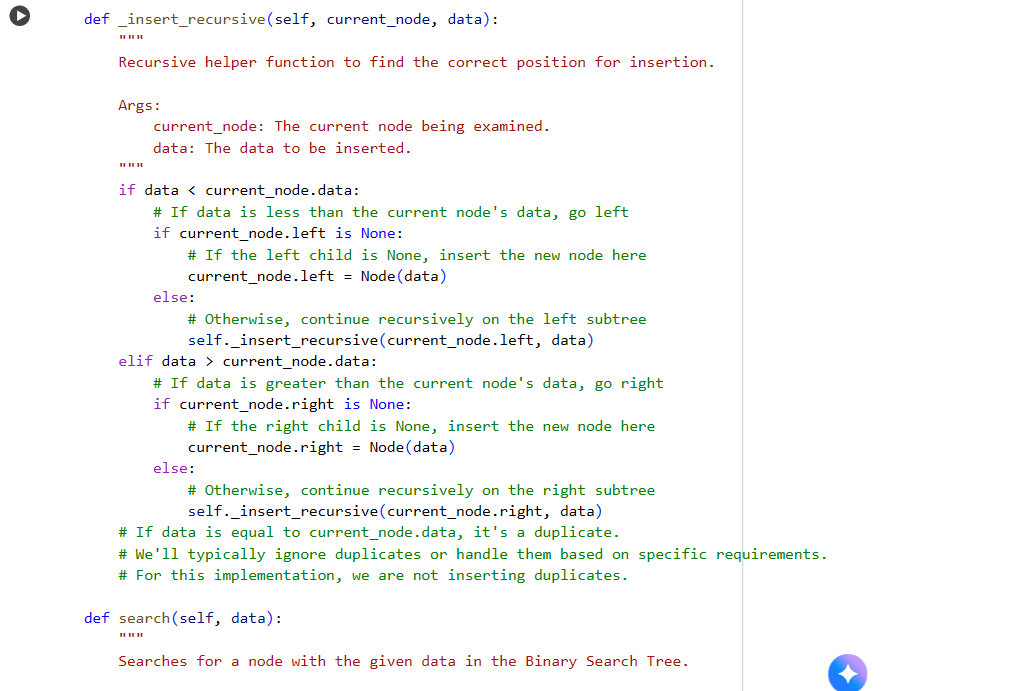
This implementation provides the basic structure of a singly linked list and allows for adding elements to the end and printing the list's contents. The delete\_value method is still pending implementation.

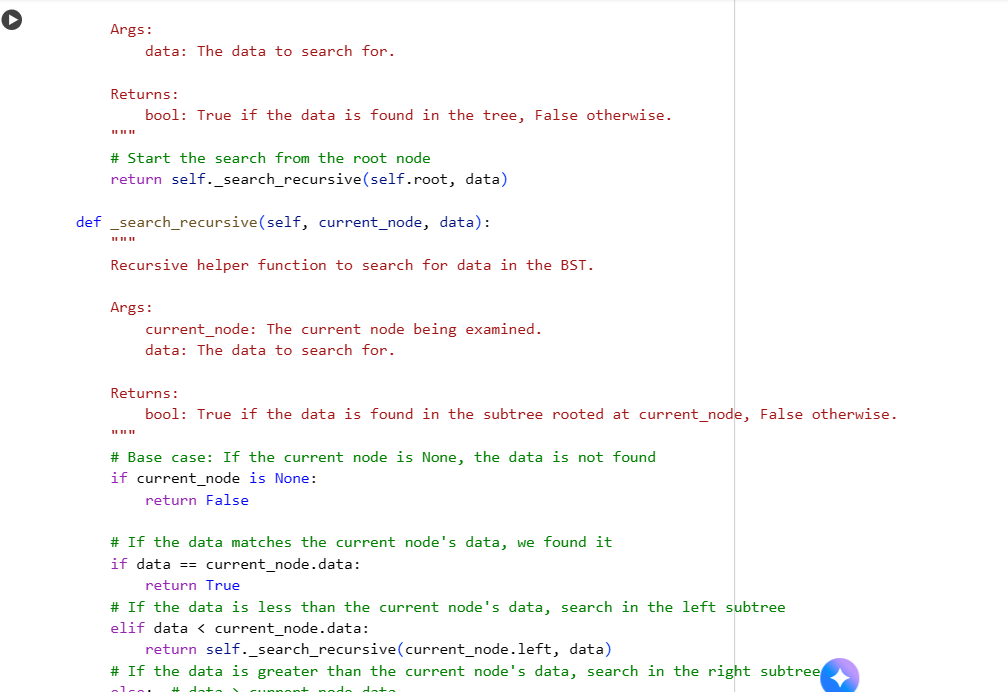
**Task 4:** Binary Search Tree (BST)  
• Task: Implement a Binary Search Tree with methods for insert(),  
search(), and inorder\_traversal().  
• Instructions:  
o Provide AI with a partially written Node and BST class.

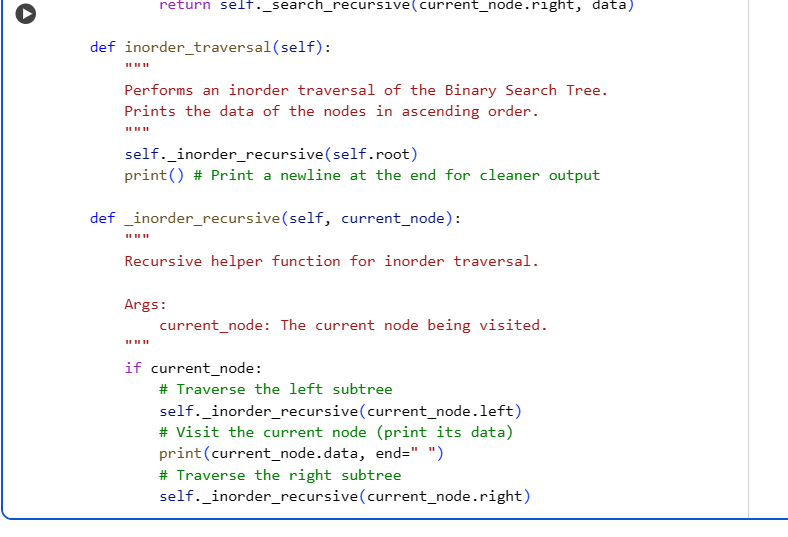
o Ask AI to complete missing methods and add docstrings.  
o Test with a list of integers and compare outputs of search() for  
present vs absent elements.  
• Expected Output:  
o A BST class with clean implementation, meaningful docstrings,  
and correct traversal output

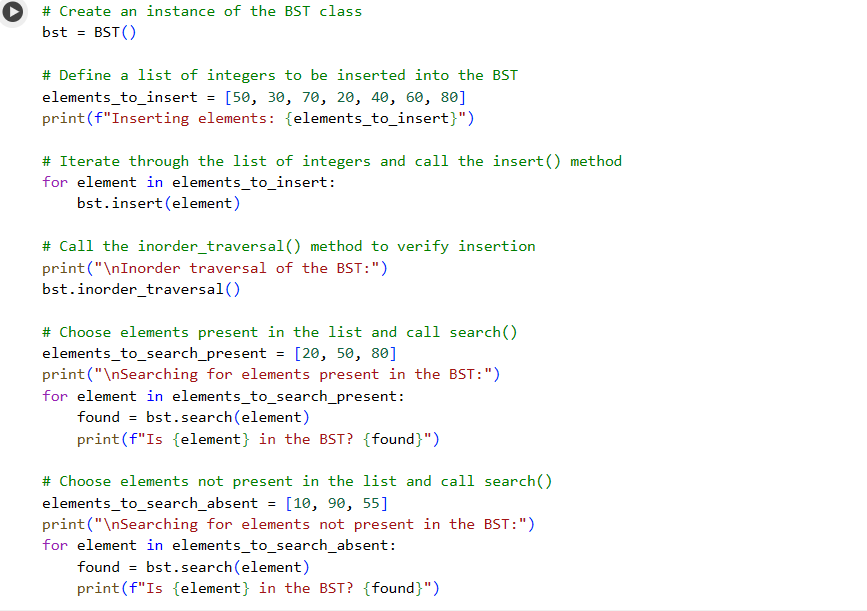
**CODE AND OUPUT:**

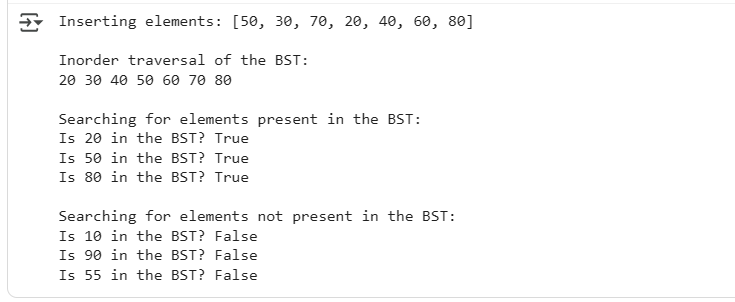
****









**EXPLANATION:**

**Node Class:**

* **\_\_init\_\_(self, data)**: This constructor initializes a new node with the given data. It sets the left and right attributes to None, indicating that the node initially has no children.

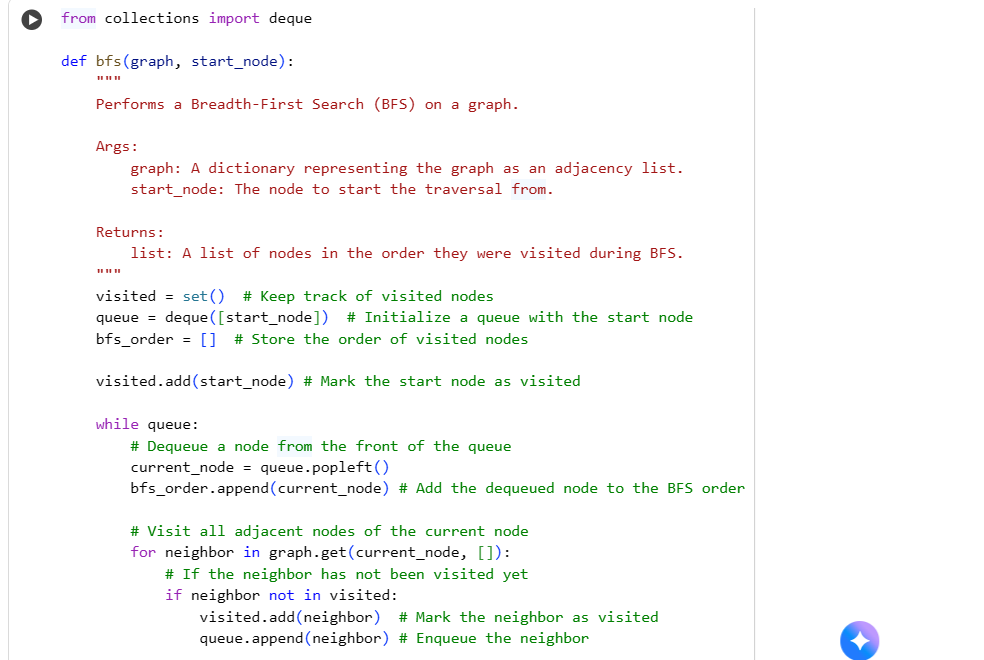
**BST Class:**

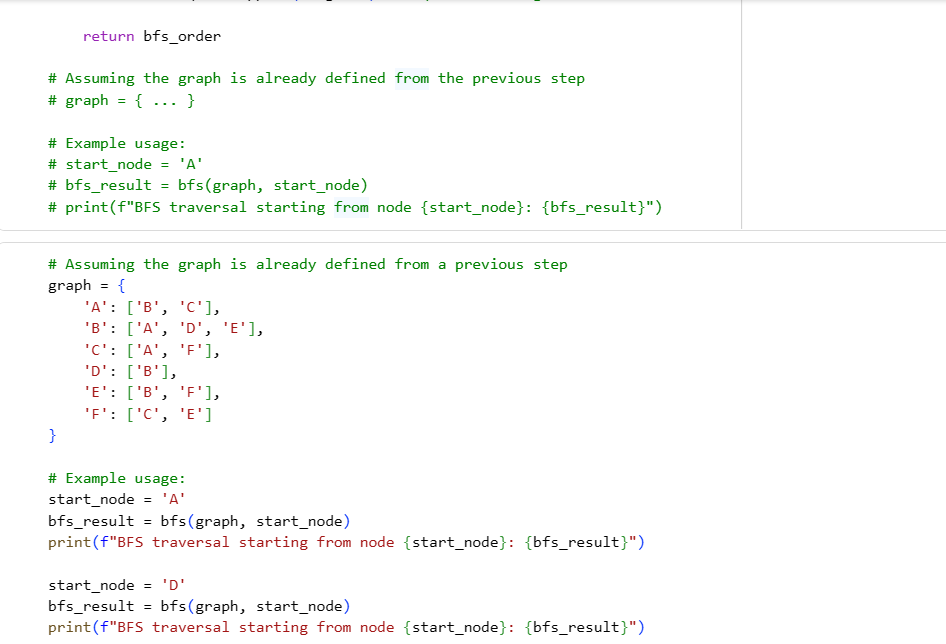
* **\_\_init\_\_(self)**: This constructor initializes an empty Binary Search Tree by setting the root attribute to None. The root attribute is a reference to the top-most node in the tree.
* **insert(self, data)**: This method inserts a new node with the given data into the BST.
  + If the tree is empty (self.root is None), the new node becomes the root.
  + Otherwise, it calls a recursive helper function \_insert\_recursive to find the correct position for the new node.
* **\_insert\_recursive(self, current\_node, data)**: This is a recursive helper function used by insert(). It traverses the tree to find where to insert the new node based on the BST properties (smaller data goes to the left, larger data goes to the right).
  + If the data is less than the current\_node's data, it goes to the left subtree. If the left child is None, the new node is inserted there. Otherwise, it recursively calls itself on the left child.
  + If the data is greater than the current\_node's data, it goes to the right subtree, following a similar logic.
  + The code also includes a comment indicating that duplicates are not inserted in this implementation.
* **search(self, data)**: This method searches for a node with the given data in the BST. It starts the search from the root by calling the recursive helper function \_search\_recursive.
* **\_search\_recursive(self, current\_node, data)**: This is a recursive helper function for searching.
  + The base case is when current\_node is None, meaning the data is not found, and it returns False.
  + If the current\_node's data matches the search data, it returns True.
  + If the search data is less than the current\_node's data, it recursively searches in the left subtree.
  + If the search data is greater than the current\_node's data, it recursively searches in the right subtree.
* **inorder\_traversal(self)**: This method performs an inorder traversal of the BST, which visits nodes in ascending order. It calls a recursive helper function \_inorder\_recursive starting from the root.
* **\_inorder\_recursive(self, current\_node)**: This is a recursive helper function for inorder traversal.
  + It checks if the current\_node is not None.
  + It recursively traverses the left subtree.
  + It then visits the current\_node (in this case, it prints the node's data).
  + Finally, it recursively traverses the right subtree.
  + After the initial call completes, it prints a newline for cleaner output.

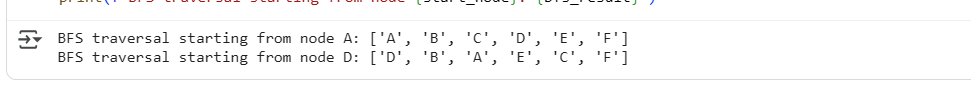
This code provides a functional implementation of a Binary Search Tree with the core operations for managing and navigating the tree structure.

**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.  
• Expected Output:  
o A graph implementation with BFS and DFS traversal methods,  
with AI-generated comments explaining traversal steps.

**CODE AND OUTPUT:**

****

****

****

**EXPLANATION:**

* **from collections import deque**: This line imports the deque class from the collections module. deque (double-ended queue) is used here because it provides efficient append and pop operations from both ends, which is ideal for implementing a queue needed for BFS.
* **def bfs(graph, start\_node):**: This defines the bfs function that takes two arguments:
  + graph: A dictionary representing the graph where keys are nodes and values are lists of their adjacent nodes (the adjacency list).
  + start\_node: The node from which the BFS traversal should begin.
* **visited = set()**: Initializes an empty set called visited to keep track of the nodes that have already been visited during the traversal. This prevents infinite loops in graphs with cycles.
* **queue = deque([start\_node])**: Initializes a deque (which acts as our queue) and adds the start\_node to it. The BFS algorithm starts by exploring the starting node.
* **bfs\_order = []**: Initializes an empty list called bfs\_order to store the nodes in the order they are visited during the BFS traversal.
* **visited.add(start\_node)**: Adds the start\_node to the visited set to mark it as visited.
* **while queue:**: This loop continues as long as there are nodes in the queue.
* **current\_node = queue.popleft()**: Removes and returns the node from the left side (the front) of the queue. This is the node currently being processed.
* **bfs\_order.append(current\_node)**: Appends the current\_node to the bfs\_order list, recording the order of traversal.
* **for neighbor in graph.get(current\_node, []):**: This loop iterates through the neighbors of the current\_node. graph.get(current\_node, []) safely retrieves the list of neighbors, returning an empty list if the current\_node is not found in the graph dictionary.
* **if neighbor not in visited:**: Checks if the neighbor has already been visited.
* **visited.add(neighbor)**: If the neighbor has not been visited, it is added to the visited set.
* **queue.append(neighbor)**: The unvisited neighbor is added to the right side (the rear) of the queue to be visited later.
* **return bfs\_order**: After the loop finishes (when the queue is empty and all reachable nodes have been visited), the function returns the bfs\_order list, which contains the nodes in the order they were visited during the BFS traversal.

This function effectively explores the graph level by level, starting from the start\_node, ensuring that all nodes at a certain distance from the start node are visited before moving on to nodes at a greater distance.