**AI Assisted Coding ASSIGNMENT** 11.1

**NAME** : Naradala satya sri charan

**COURSE** : AI assited coding

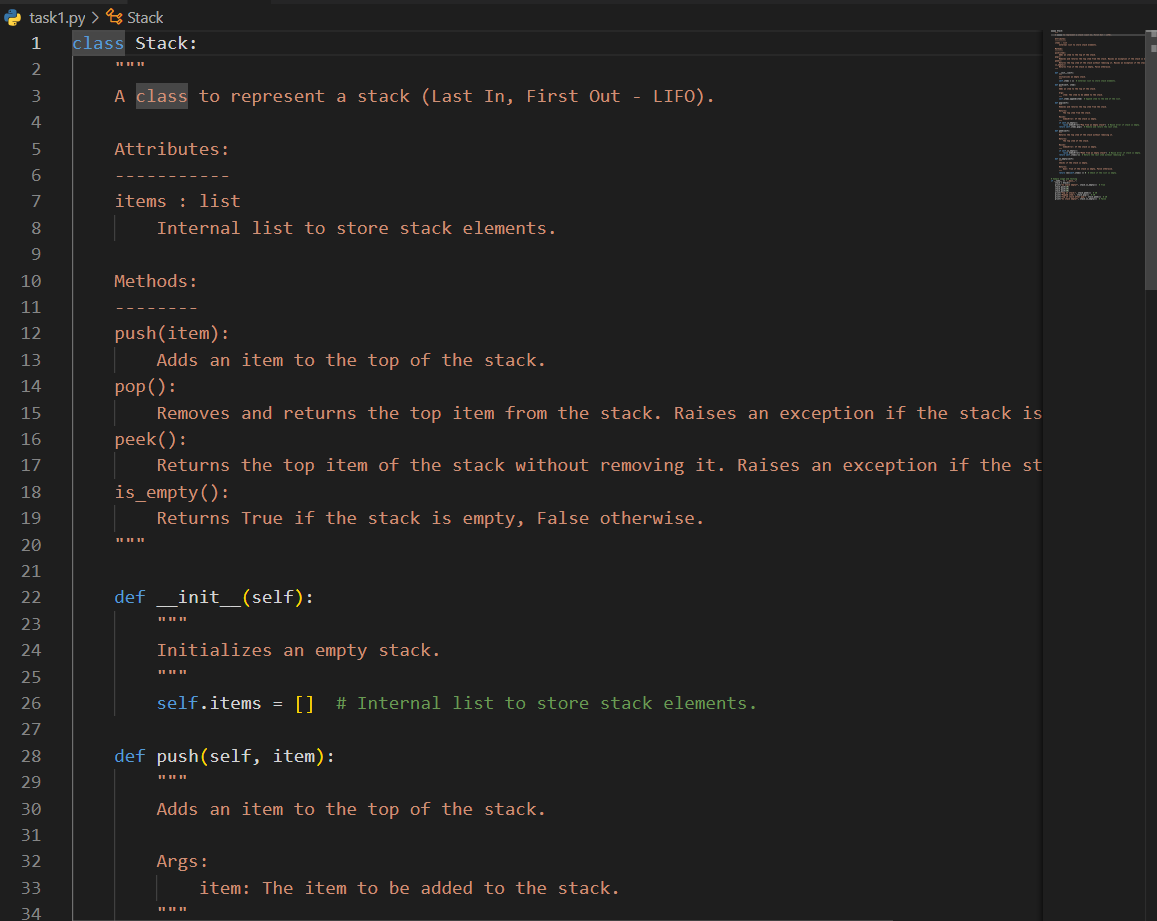
**ROLL NO** : 2503a51l05

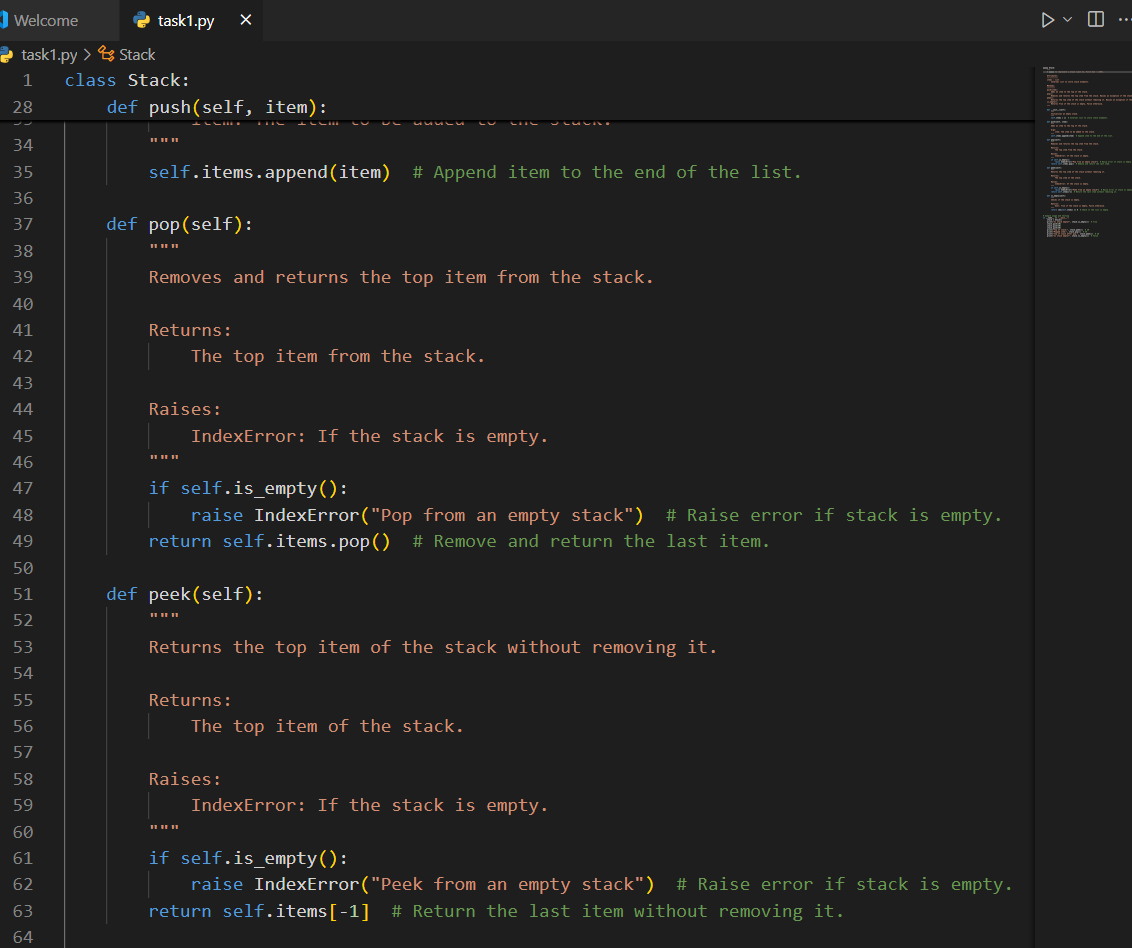
**BATCH** : 19

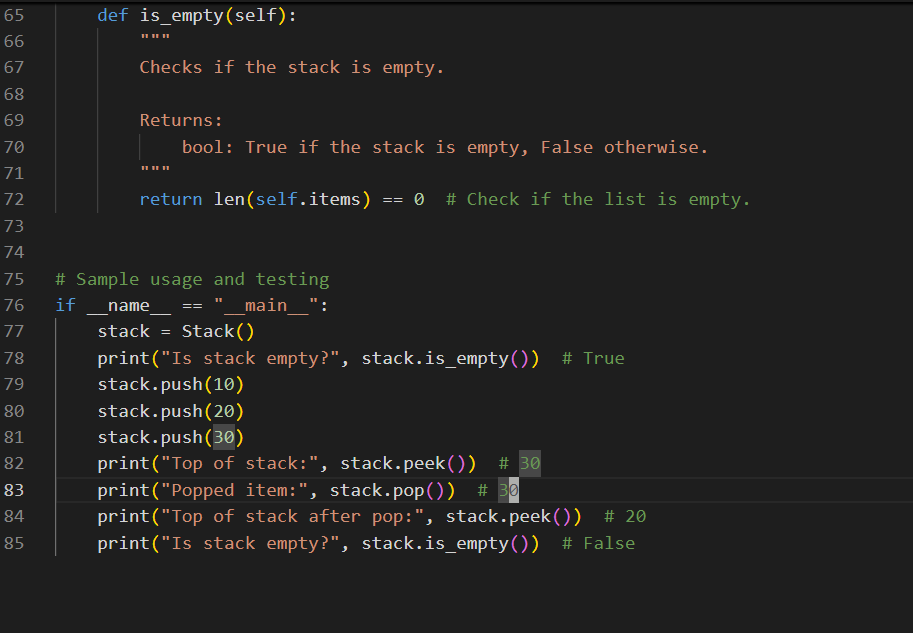
**TASK1** : **:** Implementing a Stack (LIFO)

* **Task**: Use AI to help implement a **Stack** class in Python with the following operations: push(), pop(), peek(), andis\_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** : I am implementing a Stack (LIFO) class in Python with the following operations: push(), pop(), peek(), and is\_empty()







**CODE :**

class Stack:

"""

A class to represent a stack (Last In, First Out - LIFO).

Attributes:

-----------

items : list

Internal list to store stack elements.

Methods:

--------

push(item):

Adds an item to the top of the stack.

pop():

Removes and returns the top item from the stack. Raises an exception if the stack is empty.

peek():

Returns the top item of the stack without removing it. Raises an exception if the stack is empty.

is\_empty():

Returns True if the stack is empty, False otherwise.

""”

def \_\_init\_\_(self):

"""

Initializes an empty stack.

"""

self.items = [] # Internal list to store stack elements.

def push(self, item):

"""

Adds an item to the top of the stack.

Args:

item: The item to be added to the stack.

"""

self.items.append(item) # Append item to the end of the list.

def pop(self):

"""

Removes and returns the top item from the stack.

Returns:

The top item from the stack.

Raises:

IndexError: If the stack is empty.

"""

if self.is\_empty():

raise IndexError("Pop from an empty stack") # Raise error if stack is empty.

return self.items.pop() # Remove and return the last item.

def peek(self):

"""

Returns the top item of the stack without removing it.

Returns:

The top item of the stack.

Raises:

IndexError: If the stack is empty.

"""

if self.is\_empty():

raise IndexError("Peek from an empty stack") # Raise error if stack is empty.

return self.items[-1] # Return the last item without removing it.

def is\_empty(self):

"""

Checks if the stack is empty.

Returns:

bool: True if the stack is empty, False otherwise.

"""

return len(self.items) == 0 # Check if the list is empty.

# Sample usage and testing

if \_\_name\_\_ == "\_\_main\_\_":

stack = Stack()

print("Is stack empty?", stack.is\_empty()) # True

stack.push(10)

stack.push(20)

stack.push(30)

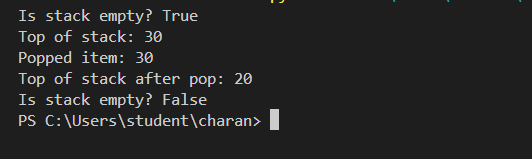
print("Top of stack:", stack.peek()) # 30

print("Popped item:", stack.pop()) # 30

print("Top of stack after pop:", stack.peek()) # 20

print("Is stack empty?", stack.is\_empty()) # False

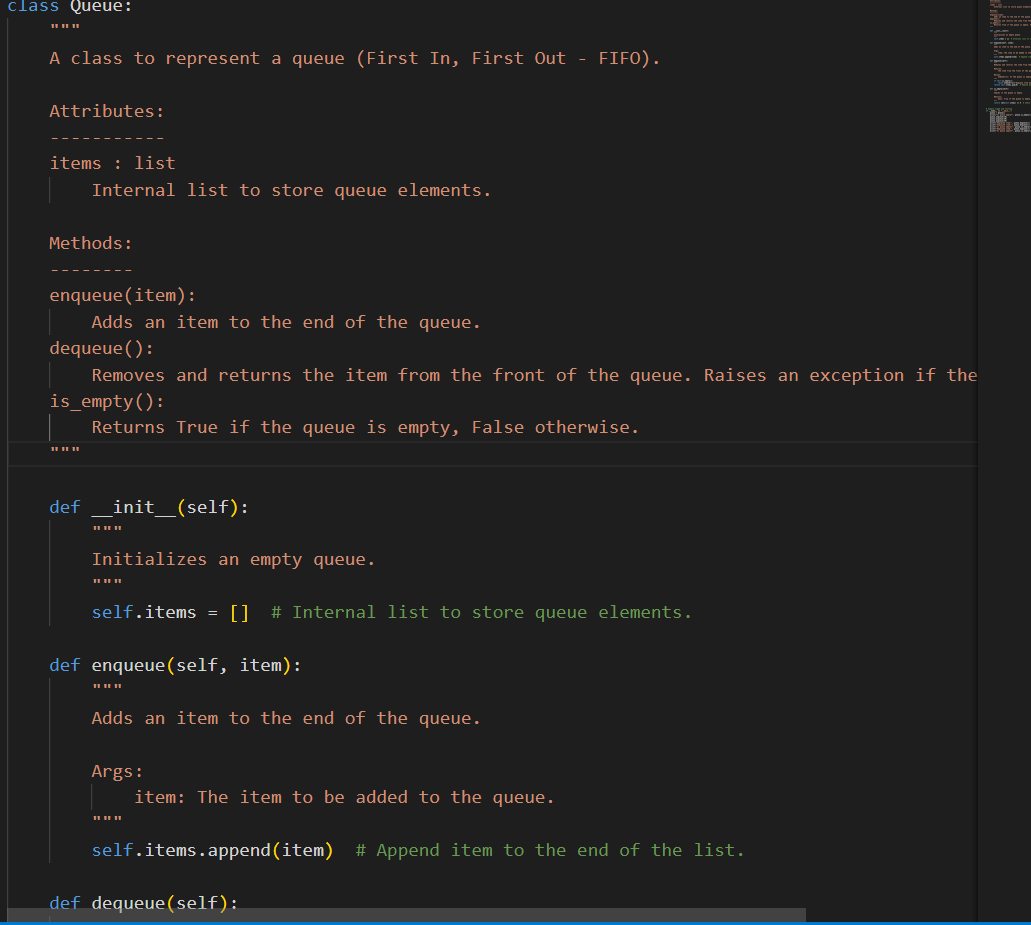
**OUTPUT :**

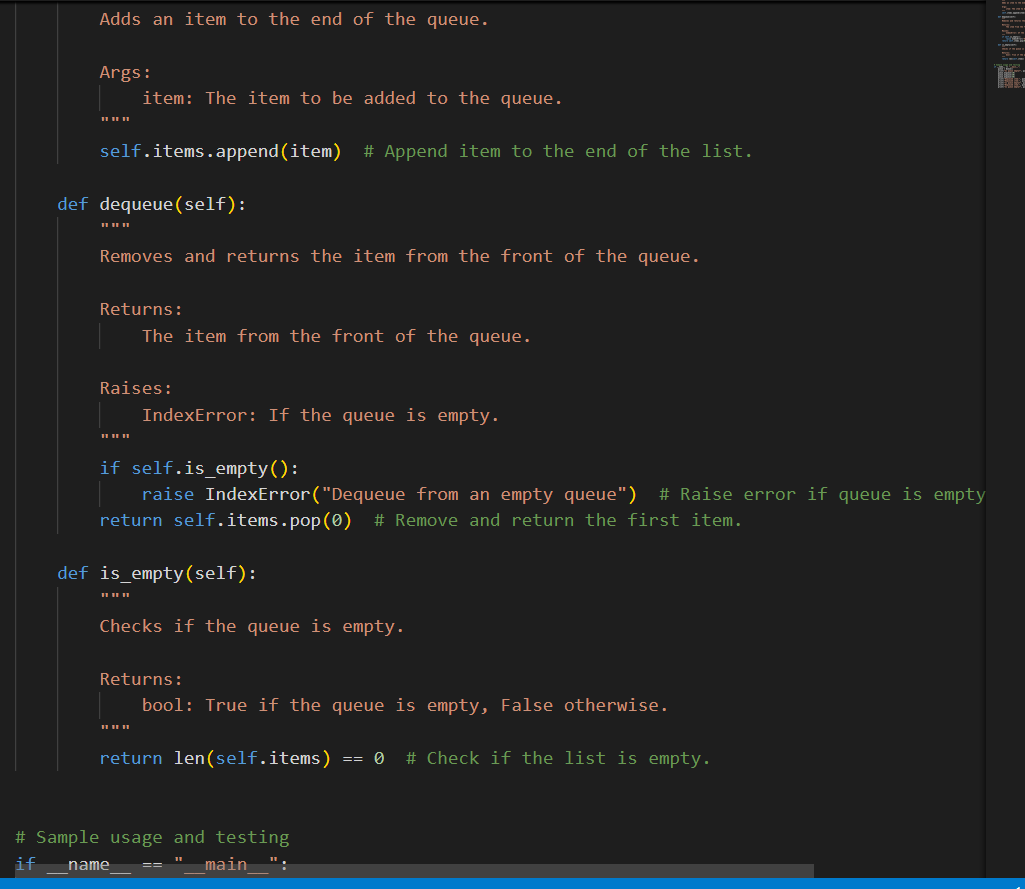
****

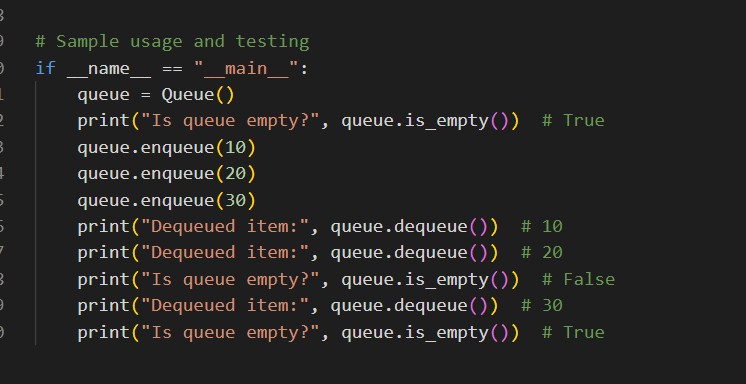
**TASK2: : Queue Implementation with Performance Review**

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

**PROMPT :** A queue can be implemented using a Python list. You can use the append() method to **enqueue** (add an item) and pop(0) to **dequeue** (remove an item from the front).

****

****

****

**CODE :**

class Queue:

"""

A class to represent a queue (First In, First Out - FIFO).

Attributes:

-----------

items : list

Internal list to store queue elements.

Methods:

--------

enqueue(item):

Adds an item to the end of the queue.

dequeue():

Removes and returns the item from the front of the queue. Raises an exception if the queue is empty.

is\_empty():

Returns True if the queue is empty, False otherwise.

"""

def \_\_init\_\_(self):

"""

Initializes an empty queue.

"""

self.items = [] # Internal list to store queue elements.

def enqueue(self, item):

"""

Adds an item to the end of the queue.

Args:

item: The item to be added to the queue.

"""

self.items.append(item) # Append item to the end of the list.

def dequeue(self):

"""

Removes and returns the item from the front of the queue.

Returns:

The item from the front of the queue.

Raises:

IndexError: If the queue is empty.

"""

if self.is\_empty():

raise IndexError("Dequeue from an empty queue") # Raise error if queue is empty.

return self.items.pop(0) # Remove and return the first item.

def is\_empty(self):

"""

Checks if the queue is empty.

Returns:

bool: True if the queue is empty, False otherwise.

"""

return len(self.items) == 0 # Check if the list is empty.

# Sample usage and testing

if \_\_name\_\_ == "\_\_main\_\_":

queue = Queue()

print("Is queue empty?", queue.is\_empty()) # True

queue.enqueue(10)

queue.enqueue(20)

queue.enqueue(30)

print("Dequeued item:", queue.dequeue()) # 10

print("Dequeued item:", queue.dequeue()) # 20

print("Is queue empty?", queue.is\_empty()) # False

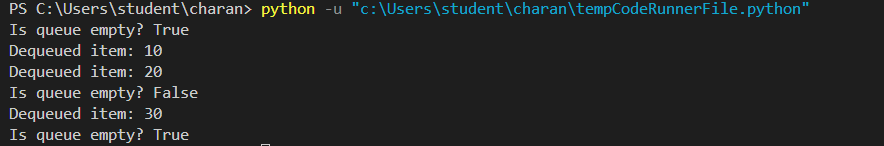
print("Dequeued item:", queue.dequeue()) # 30

print("Is queue empty?", queue.is\_empty()) # True

**OBSERVATION:**

Queues are a **First-In, First-Out (FIFO)** data structure, much like a line at a store where the first person in line is the first one to be served.

**OUTPUT:**

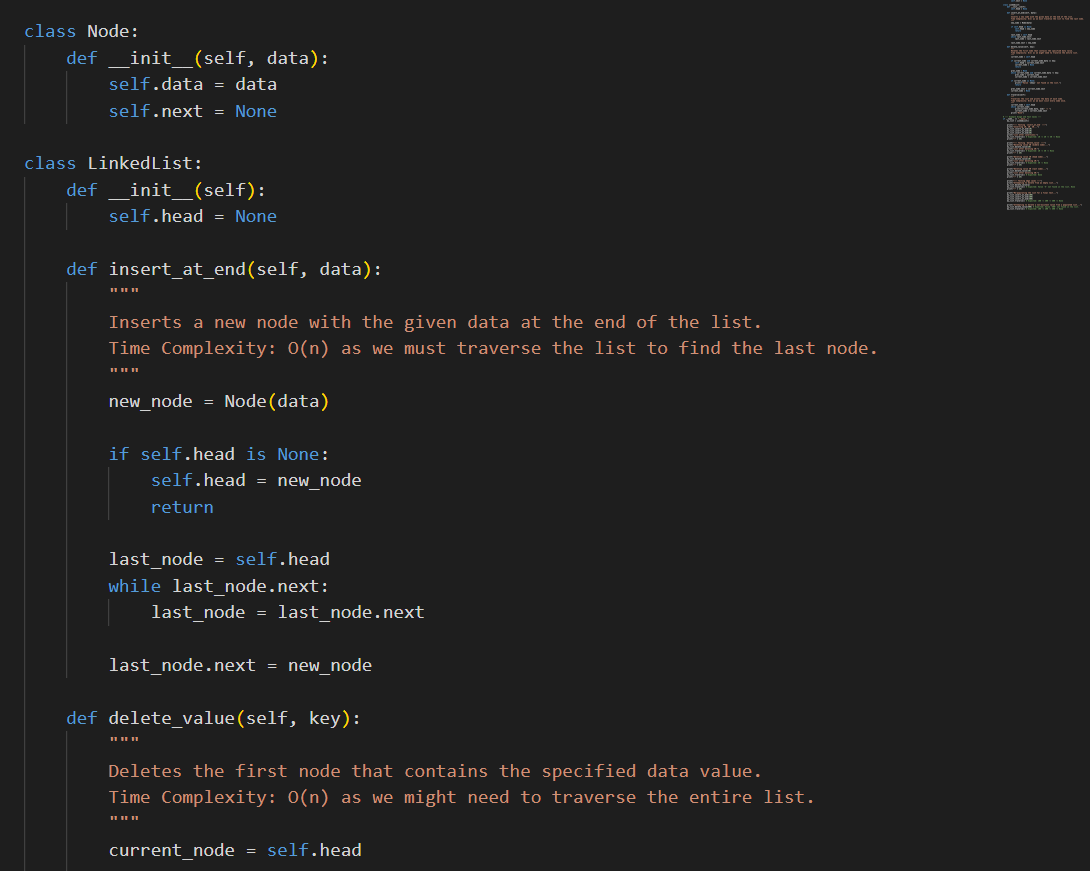
****

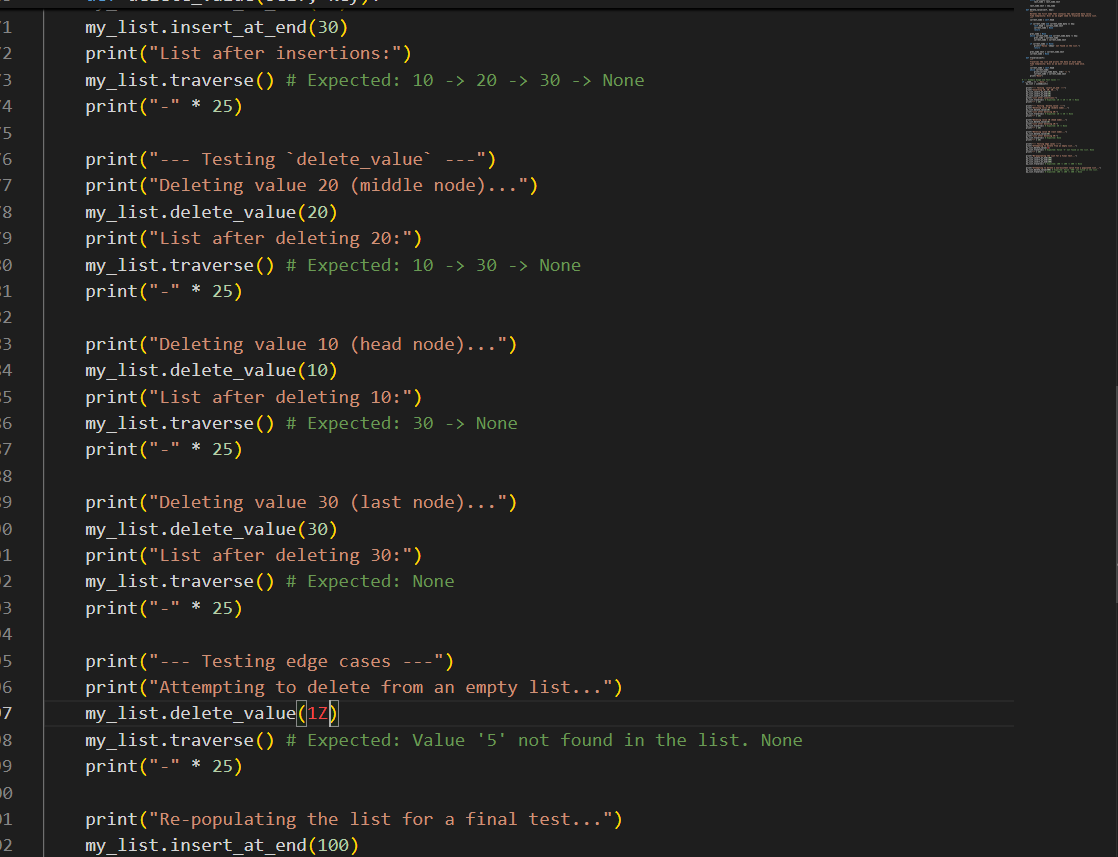
**TASK3 : Singly Linked List with Traversal**

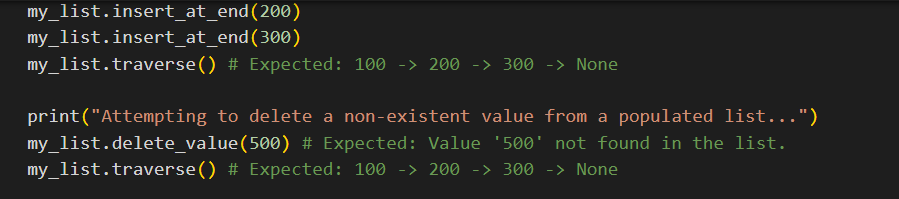
* **Task**: Implement a **Singly Linked List** with operations: insert\_at\_end(), delete\_value(), and traverse().
* **Instructions**:
  + Start with a simple class-based implementation (Node, LinkedList).
  + Use AI to generate inline comments explaining pointer updates (which are non-trivial).

Ask AI to suggest test cases to validate all operations

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

****

****

****

**CODE:**

class Node:

def \_\_init\_\_(self, data):

self.data = data

self.next = None

class LinkedList:

def \_\_init\_\_(self):

self.head = None

def insert\_at\_end(self, data):

"""

Inserts a new node with the given data at the end of the list.

Time Complexity: O(n) as we must traverse the list to find the last node.

"""

new\_node = Node(data)

if self.head is None:

self.head = new\_node

return

last\_node = self.head

while last\_node.next:

last\_node = last\_node.next

last\_node.next = new\_node

def delete\_value(self, key):

"""

Deletes the first node that contains the specified data value.

Time Complexity: O(n) as we might need to traverse the entire list.

"""

current\_node = self.head

if current\_node and current\_node.data == key:

self.head = current\_node.next

current\_node = None

return

prev\_node = None

while current\_node and current\_node.data != key:

prev\_node = current\_node

current\_node = current\_node.next

if current\_node is None:

print(f"Value '{key}' not found in the list.")

return

prev\_node.next = current\_node.next

current\_node = None

def traverse(self):

"""

Traverses the list and prints the data of each node.

Time Complexity: O(n) as we must visit every node once.

"""

current\_node = self.head

while current\_node:

print(current\_node.data, end=" -> ")

current\_node = current\_node.next

print("None")

# --- Example Usage and Test Cases ---

if \_\_name\_\_ == "\_\_main\_\_":

my\_list = LinkedList()

print("--- Testing `insert\_at\_end` ---")

print("Inserting 10, 20, 30...")

my\_list.insert\_at\_end(10)

my\_list.insert\_at\_end(20)

my\_list.insert\_at\_end(30)

print("List after insertions:")

my\_list.traverse() # Expected: 10 -> 20 -> 30 -> None

print("-" \* 25)

print("--- Testing `delete\_value` ---")

print("Deleting value 20 (middle node)...")

my\_list.delete\_value(20)

print("List after deleting 20:")

my\_list.traverse() # Expected: 10 -> 30 -> None

print("-" \* 25)

print("Deleting value 10 (head node)...")

my\_list.delete\_value(10)

print("List after deleting 10:")

my\_list.traverse() # Expected: 30 -> None

print("-" \* 25)

print("Deleting value 30 (last node)...")

my\_list.delete\_value(30)

print("List after deleting 30:")

my\_list.traverse() # Expected: None

print("-" \* 25)

print("--- Testing edge cases ---")

print("Attempting to delete from an empty list...")

my\_list.delete\_value(1Z)

my\_list.traverse() # Expected: Value '5' not found in the list. None

print("-" \* 25)

print("Re-populating the list for a final test...")

my\_list.insert\_at\_end(100)

my\_list.insert\_at\_end(200)

my\_list.insert\_at\_end(300)

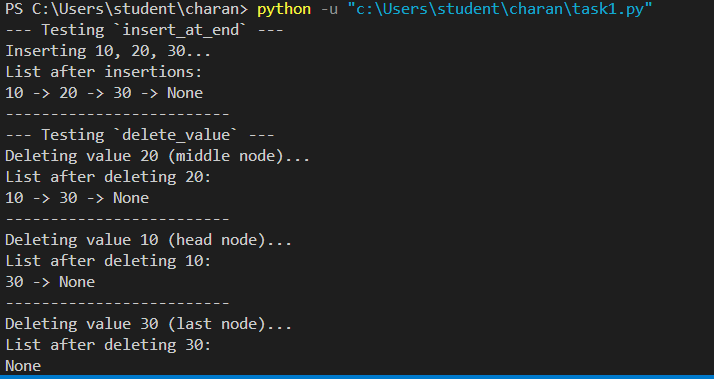
my\_list.traverse() # Expected: 100 -> 200 -> 300 -> None

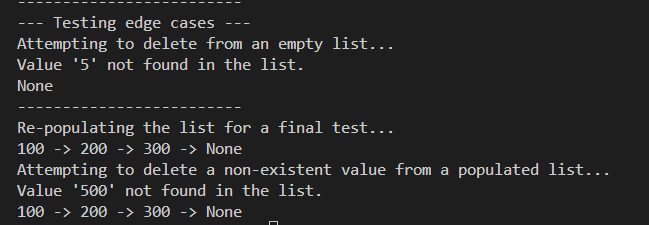
print("Attempting to delete a non-existent value from a populated list...")

my\_list.delete\_value(500) # Expected: Value '500' not found in the list.

my\_list.traverse() # Expected: 100 -> 200 -> 300 -> None

**OUTPUT:**

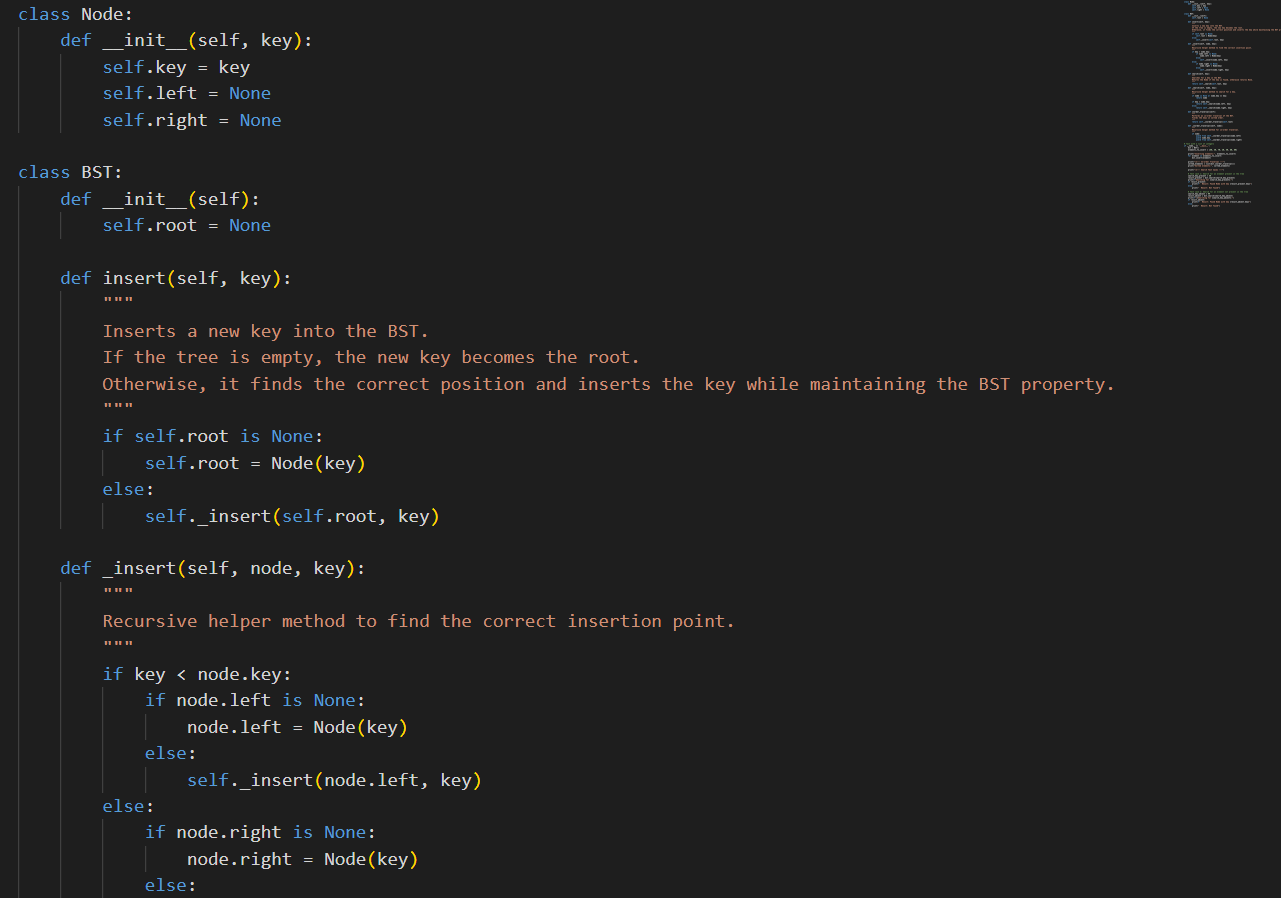
****

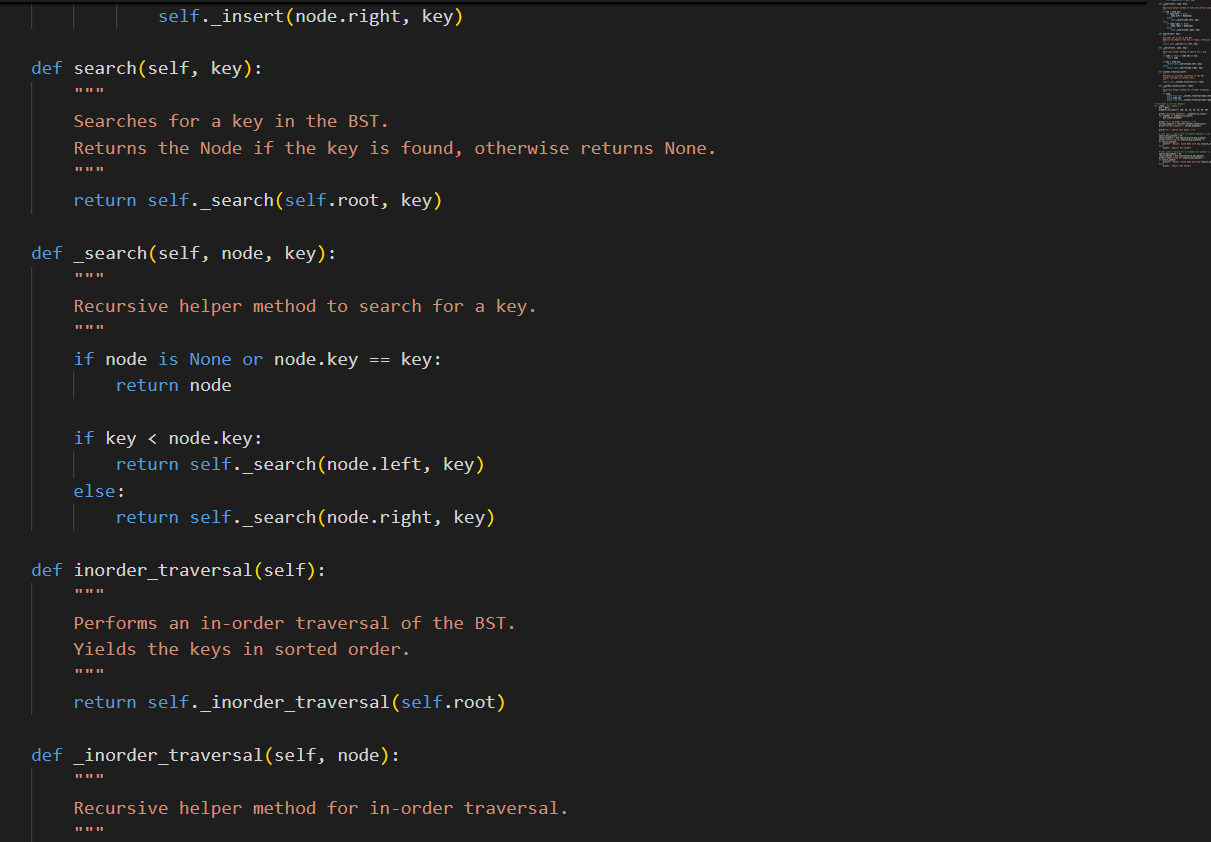
****

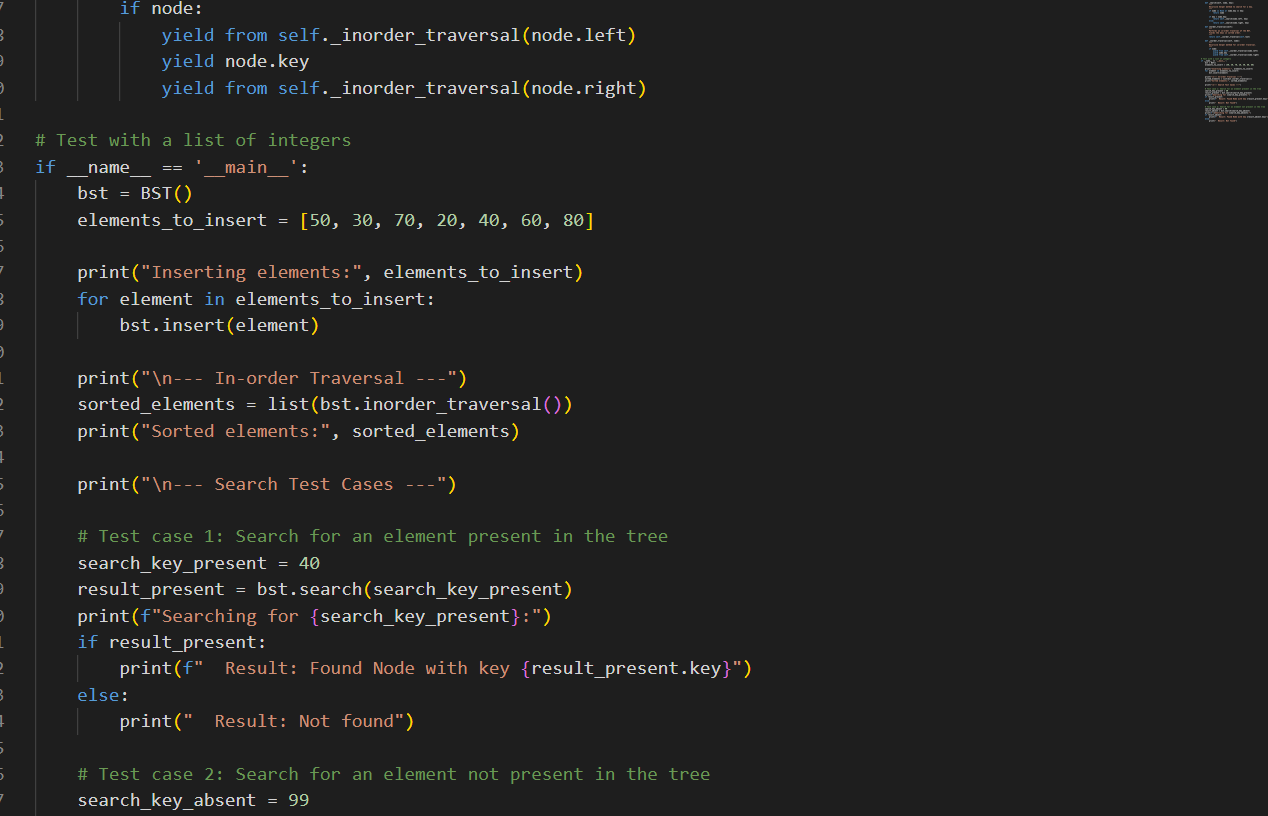
**OBSERVATION:** singly linked list with three core operations: insert\_at\_end, delete\_value, and traverse , The provided test cases cover a wide range of scenarios, including inserting into an empty and non-empty list, deleting the head, middle, and tail nodes, and handling attempts to delete non-existent values. The output clearly validates that all methods function as expected by showing the state of the list after each operation. The code's structure is clean and well-commented, making the logic of pointer manipulation easy to follow.

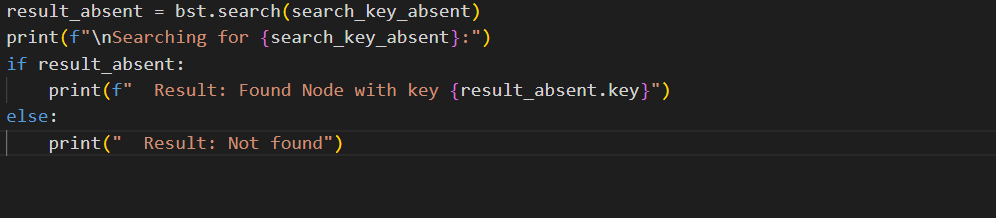
**Task 4: Binary Search Tree (BST)**

* **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.

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

****

****

****

**CODE :**

class Node:

def \_\_init\_\_(self, key):

self.key = key

self.left = None

self.right = None

class BST:

def \_\_init\_\_(self):

self.root = None

def insert(self, key):

"""

Inserts a new key into the BST.

If the tree is empty, the new key becomes the root.

Otherwise, it finds the correct position and inserts the key while maintaining the BST property.

"""

if self.root is None:

self.root = Node(key)

else:

self.\_insert(self.root, key)

def \_insert(self, node, key):

"""

Recursive helper method to find the correct insertion point.

"""

if key < node.key:

if node.left is None:

node.left = Node(key)

else:

self.\_insert(node.left, key)

else:

if node.right is None:

node.right = Node(key)

else:

self.\_insert(node.right, key)

def search(self, key):

"""

Searches for a key in the BST.

Returns the Node if the key is found, otherwise returns None.

"""

return self.\_search(self.root, key)

def \_search(self, node, key):

"""

Recursive helper method to search for a key.

"""

if node is None or node.key == key:

return node

if key < node.key:

return self.\_search(node.left, key)

else:

return self.\_search(node.right, key)

def inorder\_traversal(self):

"""

Performs an in-order traversal of the BST.

Yields the keys in sorted order.

"""

return self.\_inorder\_traversal(self.root)

def \_inorder\_traversal(self, node):

"""

Recursive helper method for in-order traversal.

"""

if node:

yield from self.\_inorder\_traversal(node.left)

yield node.key

yield from self.\_inorder\_traversal(node.right)

# Test with a list of integers

if \_\_name\_\_ == '\_\_main\_\_':

bst = BST()

elements\_to\_insert = [50, 30, 70, 20, 40, 60, 80]

print("Inserting elements:", elements\_to\_insert)

for element in elements\_to\_insert:

bst.insert(element)

print("\n--- In-order Traversal ---")

sorted\_elements = list(bst.inorder\_traversal())

print("Sorted elements:", sorted\_elements)

print("\n--- Search Test Cases ---")

# Test case 1: Search for an element present in the tree

search\_key\_present = 40

result\_present = bst.search(search\_key\_present)

print(f"Searching for {search\_key\_present}:")

if result\_present:

print(f" Result: Found Node with key {result\_present.key}")

else:

print(" Result: Not found")

# Test case 2: Search for an element not present in the tree

search\_key\_absent = 99

result\_absent = bst.search(search\_key\_absent)

print(f"\nSearching for {search\_key\_absent}:")

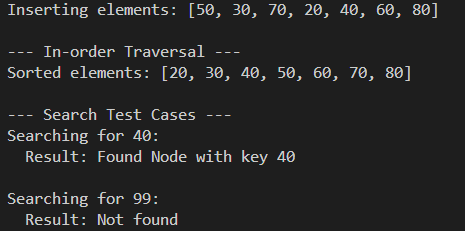
if result\_absent:

print(f" Result: Found Node with key {result\_absent.key}")

else:

print(" Result: Not found")

**OUTPUT:**

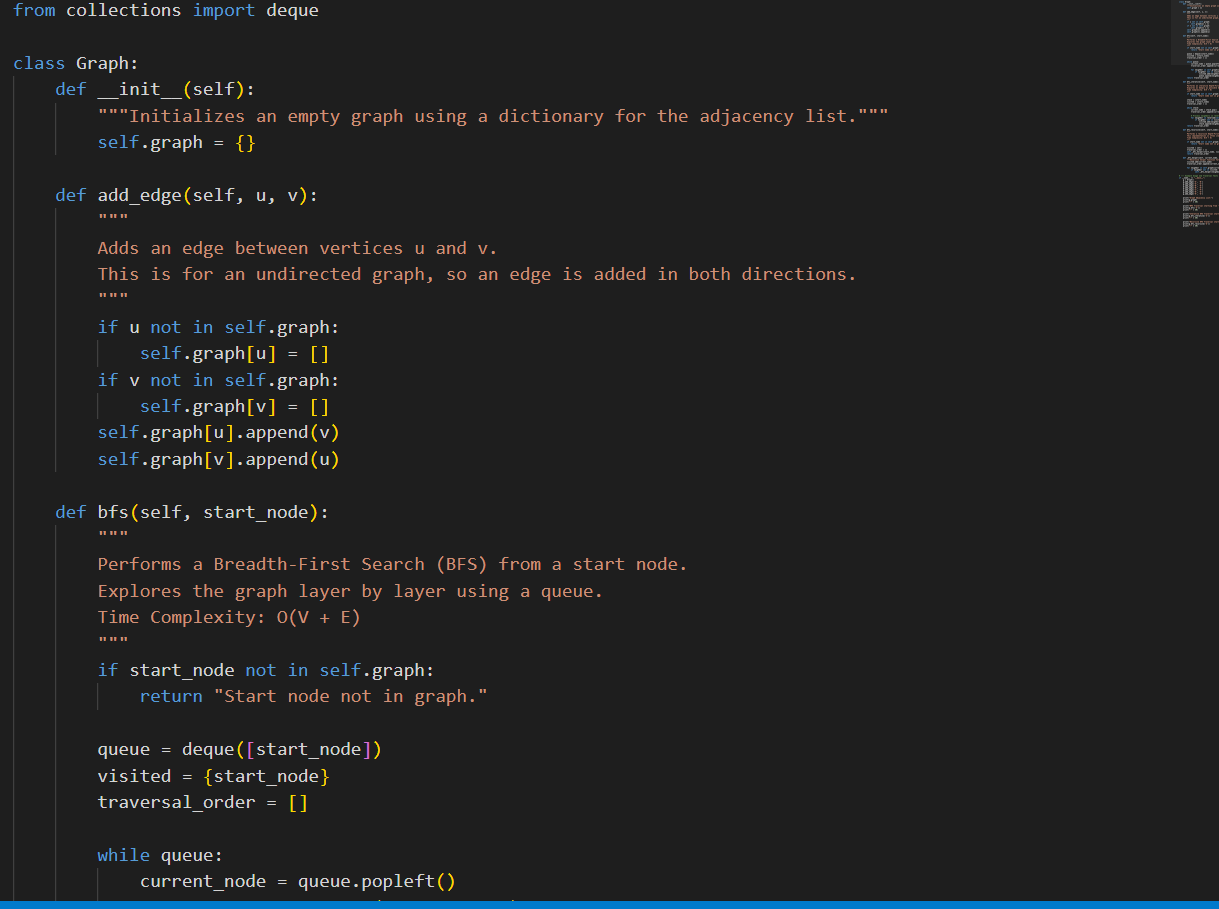
****

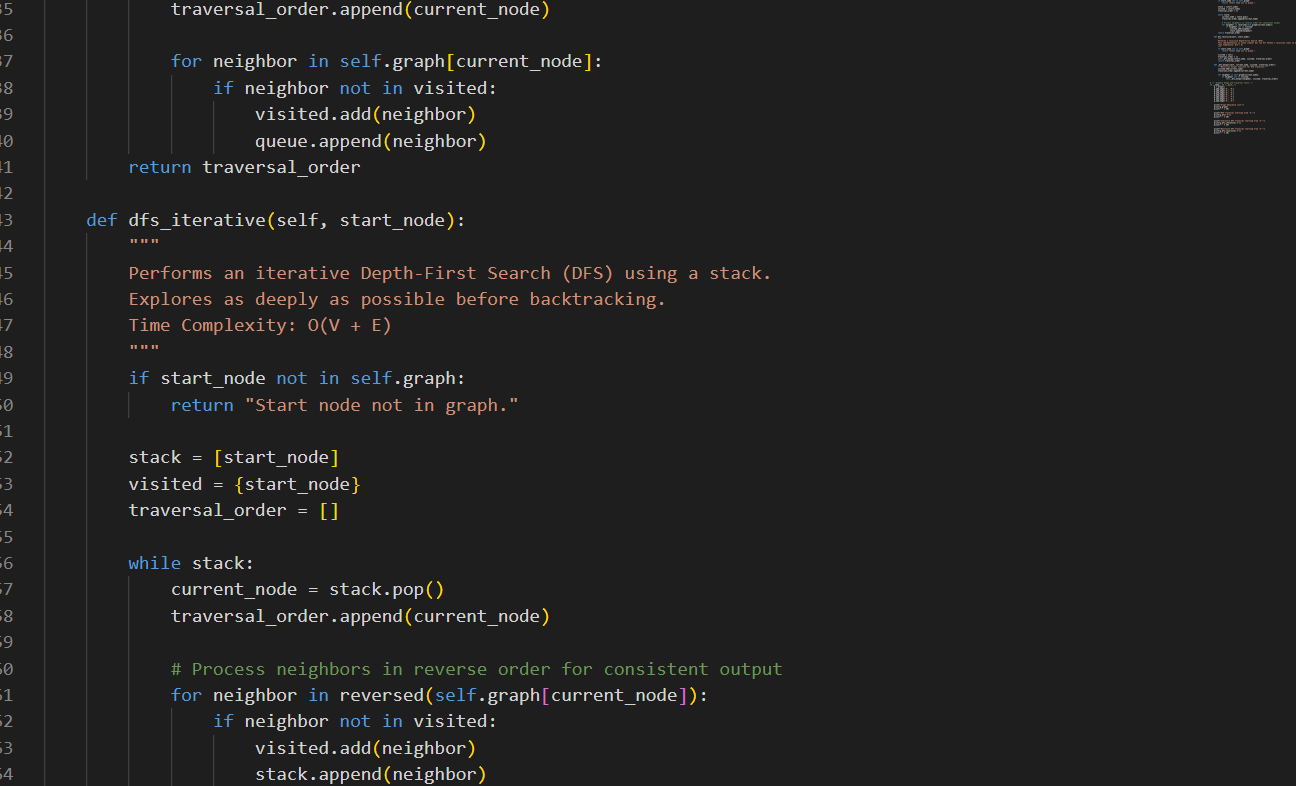
**OBSERVATION:** he code correctly defines the BST's structure and behavior, and the test cases effectively demonstrate its core functionality, including insertion, sorted traversal, and searching for both present and absent elements.

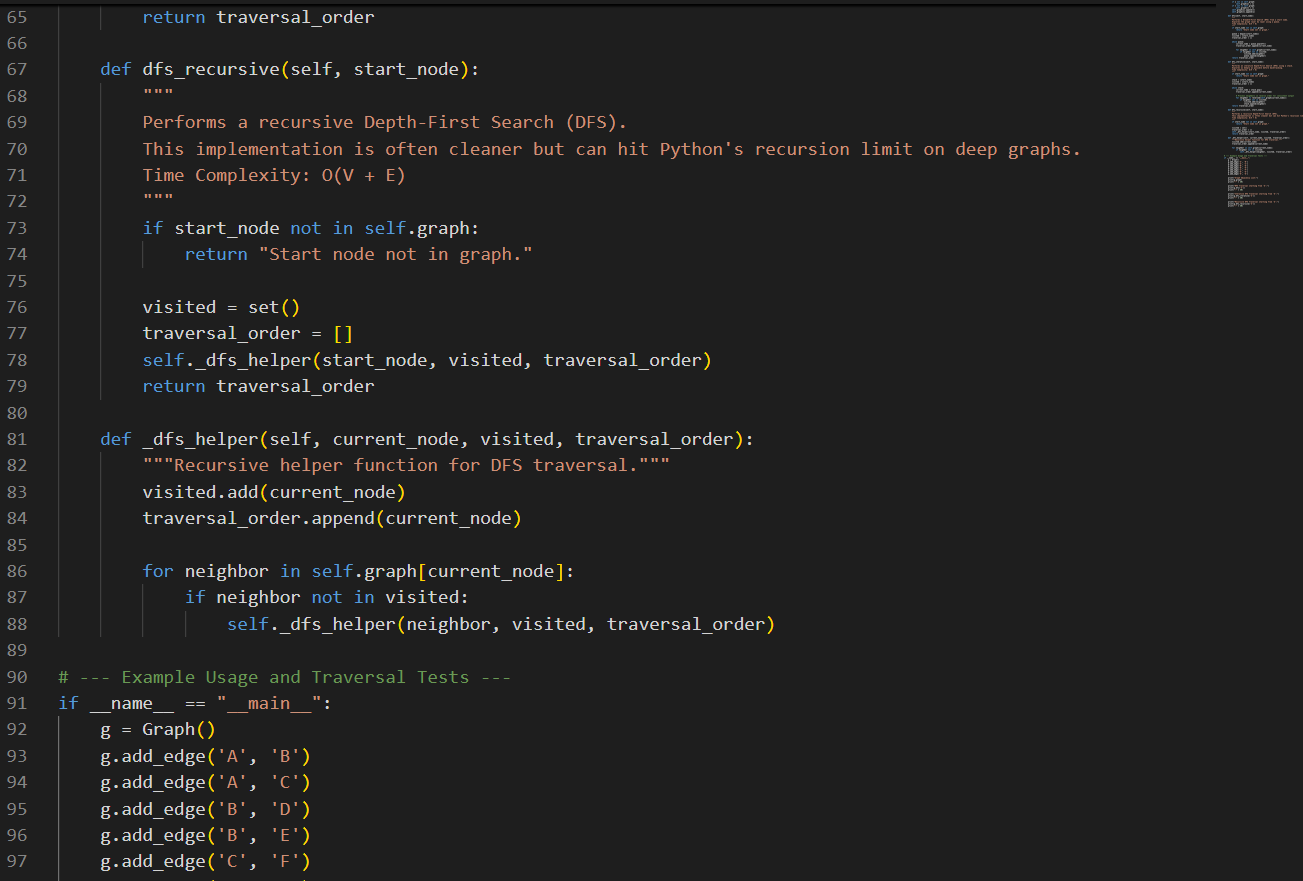
**TASK 5: Graph Representation and BFS/DFS Traversal**

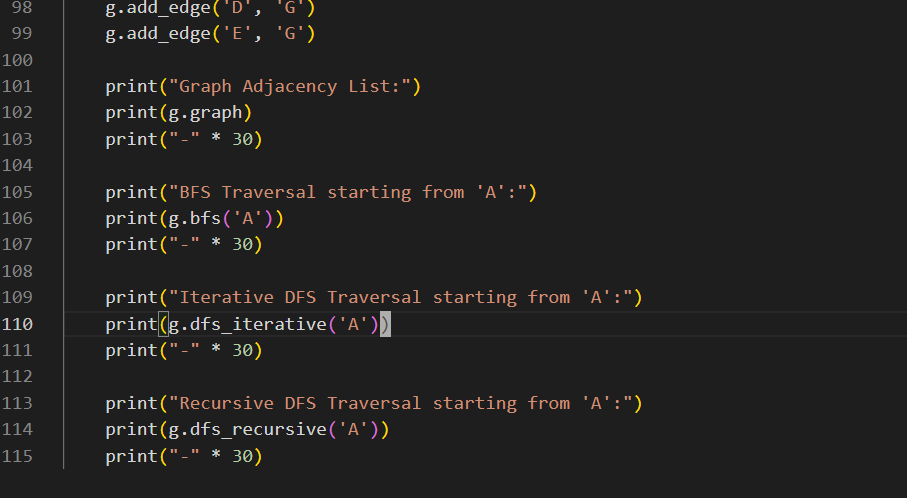
* **Task**: Implement a **Graph** using an adjacency list, with traversal methods BFS() and DFS().
* **Instructions**:
  + Start with an adjacency list dictionary.
  + Ask AI to generate BFS and DFS implementations with inline comments.
  + Compare recursive vs iterative DFS if suggested by AI.

**PROMPT :** Binary Search Tree with methods for insert(), search(), and inorder\_traversal().

****

****

****

****

**CODE :**

**from collections import deque**

class Graph:

def \_\_init\_\_(self):

"""Initializes an empty graph using a dictionary for the adjacency list."""

self.graph = {}

def add\_edge(self, u, v):

"""

Adds an edge between vertices u and v.

This is for an undirected graph, so an edge is added in both directions.

"""

if u not in self.graph:

self.graph[u] = []

if v not in self.graph:

self.graph[v] = []

self.graph[u].append(v)

self.graph[v].append(u)

def bfs(self, start\_node):

"""

Performs a Breadth-First Search (BFS) from a start node.

Explores the graph layer by layer using a queue.

Time Complexity: O(V + E)

"""

if start\_node not in self.graph:

return "Start node not in graph."

queue = deque([start\_node])

visited = {start\_node}

traversal\_order = []

while queue:

current\_node = queue.popleft()

traversal\_order.append(current\_node)

for neighbor in self.graph[current\_node]:

if neighbor not in visited:

visited.add(neighbor)

queue.append(neighbor)

return traversal\_order

def dfs\_iterative(self, start\_node):

"""

Performs an iterative Depth-First Search (DFS) using a stack.

Explores as deeply as possible before backtracking.

Time Complexity: O(V + E)

"""

if start\_node not in self.graph:

return "Start node not in graph."

stack = [start\_node]

visited = {start\_node}

traversal\_order = []

while stack:

current\_node = stack.pop()

traversal\_order.append(current\_node)

# Process neighbors in reverse order for consistent output

for neighbor in reversed(self.graph[current\_node]):

if neighbor not in visited:

visited.add(neighbor)

stack.append(neighbor)

return traversal\_order

def dfs\_recursive(self, start\_node):

"""

Performs a recursive Depth-First Search (DFS).

This implementation is often cleaner but can hit Python's recursion limit on deep graphs.

Time Complexity: O(V + E)

"""

if start\_node not in self.graph:

return "Start node not in graph."

visited = set()

traversal\_order = []

self.\_dfs\_helper(start\_node, visited, traversal\_order)

return traversal\_order

def \_dfs\_helper(self, current\_node, visited, traversal\_order):

"""Recursive helper function for DFS traversal."""

visited.add(current\_node)

traversal\_order.append(current\_node)

for neighbor in self.graph[current\_node]:

if neighbor not in visited:

self.\_dfs\_helper(neighbor, visited, traversal\_order)

# --- Example Usage and Traversal Tests ---

if \_\_name\_\_ == "\_\_main\_\_":

g = Graph()

g.add\_edge('A', 'B')

g.add\_edge('A', 'C')

g.add\_edge('B', 'D')

g.add\_edge('B', 'E')

g.add\_edge('C', 'F')

g.add\_edge('D', 'G')

g.add\_edge('E', 'G')

print("Graph Adjacency List:")

print(g.graph)

print("-" \* 30)

print("BFS Traversal starting from 'A':")

print(g.bfs('A'))

print("-" \* 30)

print("Iterative DFS Traversal starting from 'A':")

print(g.dfs\_iterative('A'))

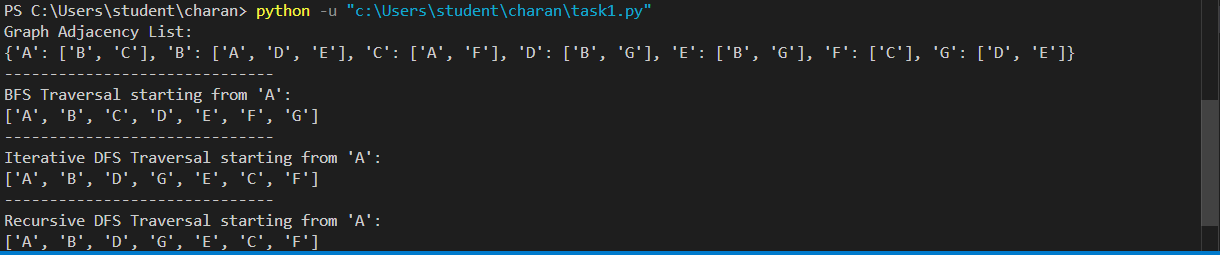
print("-" \* 30)

print("Recursive DFS Traversal starting from 'A':")

print(g.dfs\_recursive('A'))

print("-" \* 30)

**OUTPUT:**

****