**ASIGNEMENT – 11**

*Name – Sandela NavaTej*

*Roll – 2403a54130*

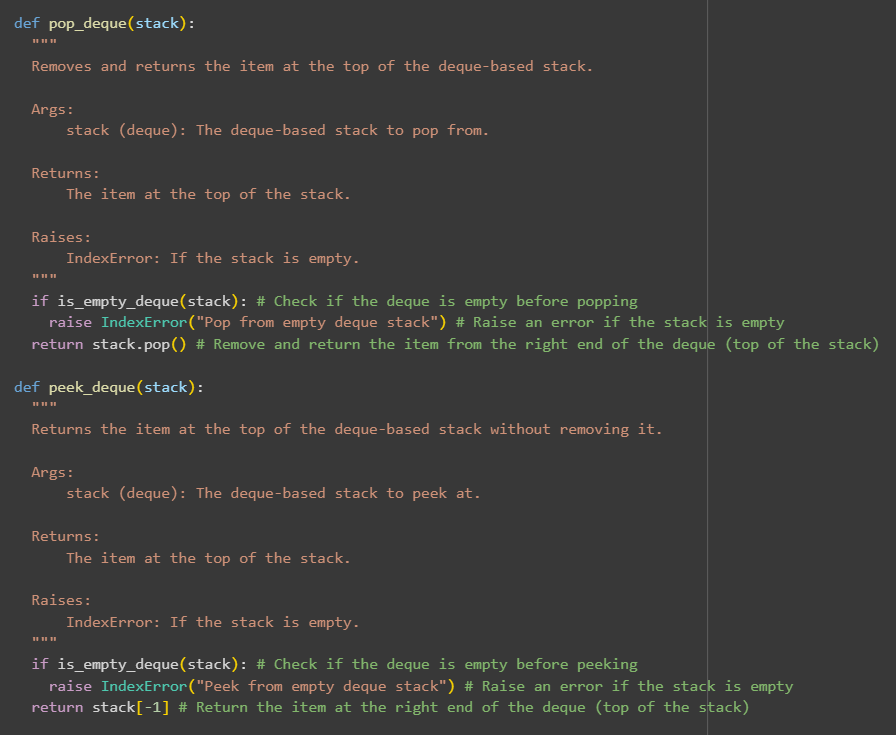
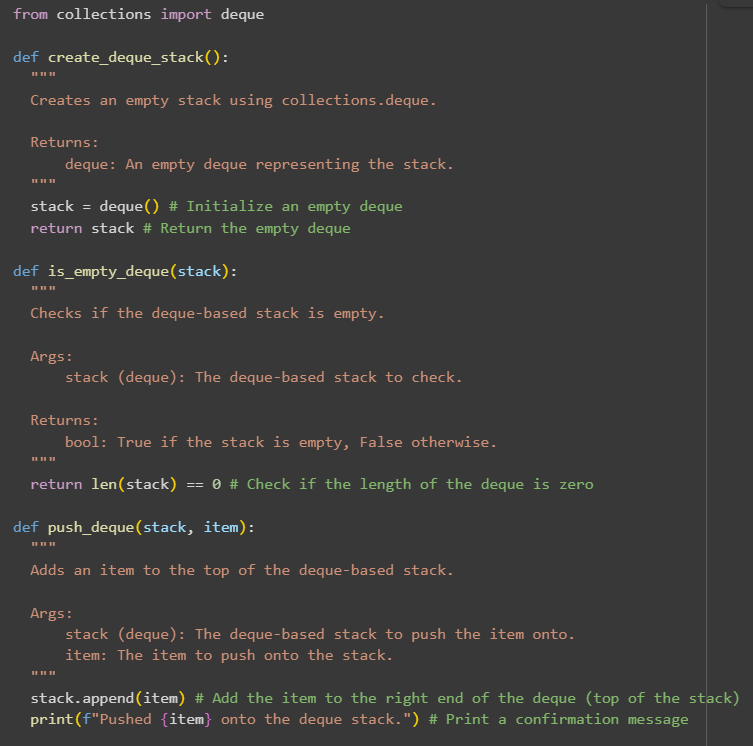
*Batch (DS) – 04*

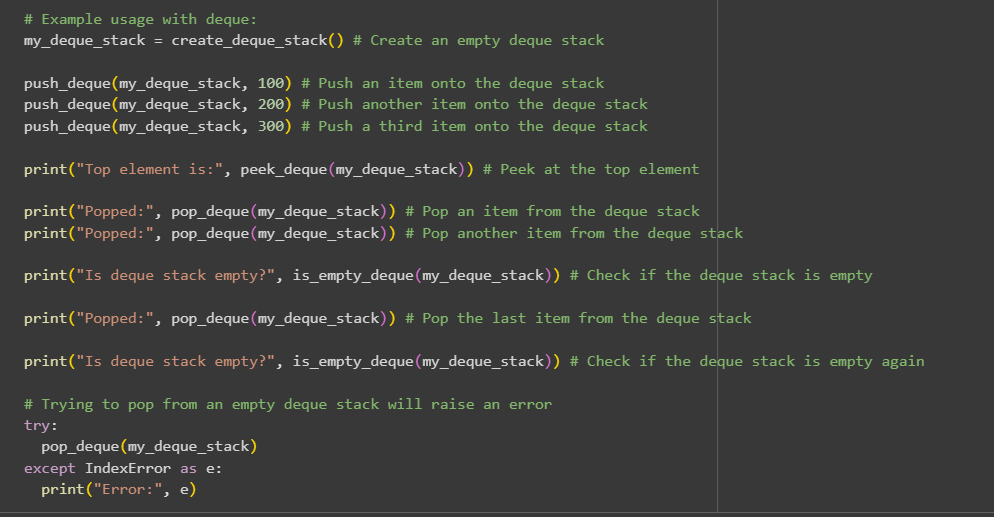
**Task 1: Implementing a Stack (LIFO):**

**Task: Use AI to help implement a Stack class in Python with the following operations: push(), pop(), peek(), and is\_empty().**

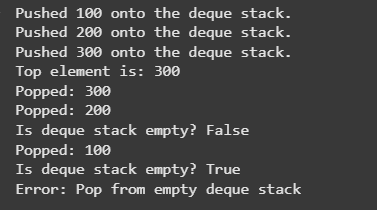
**PROMPT**: Generate a python code for implementing a stack LIFO using functions like pop(),push(),peek(),is\_empty(). Add docstrings to all the functions and add inline comments for every line.

**CODE:**

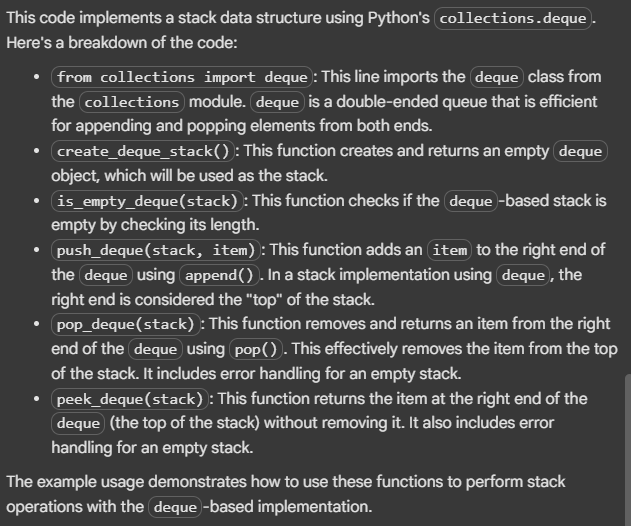




**OUTPUT:**

****

**EXPLANATION:**

****

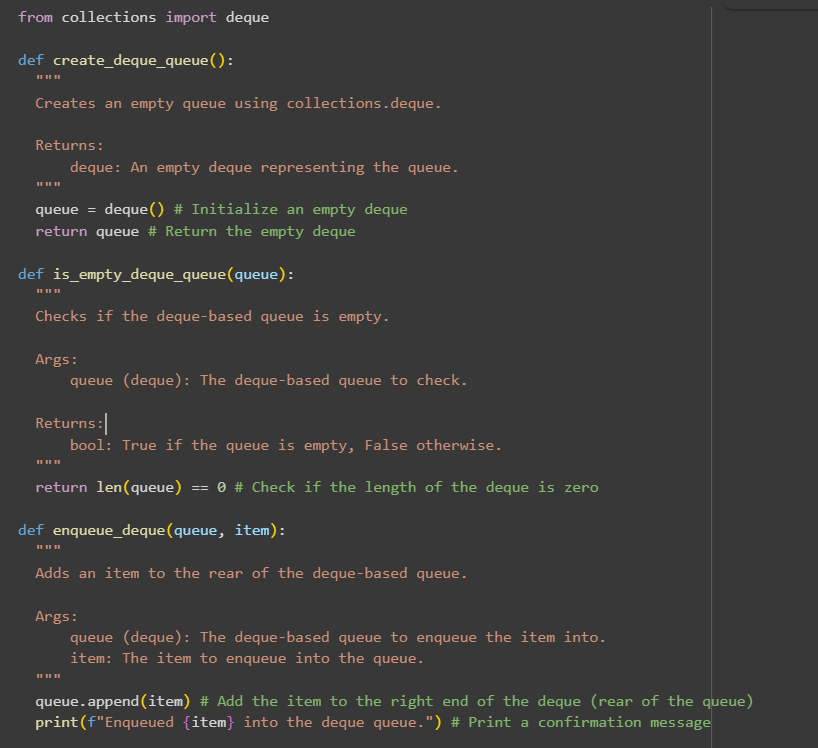
**Task 2: Queue Implementation with Performance Review**

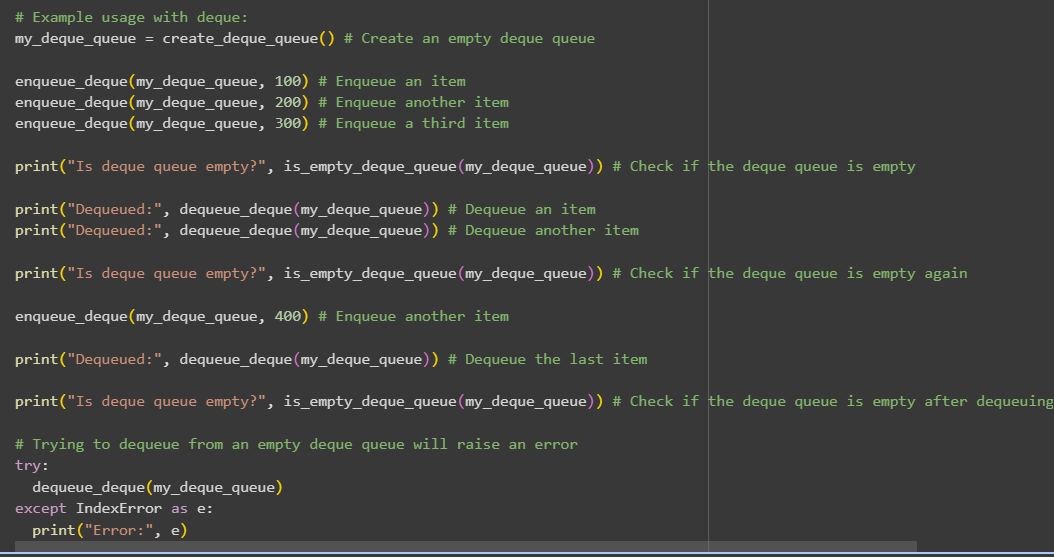
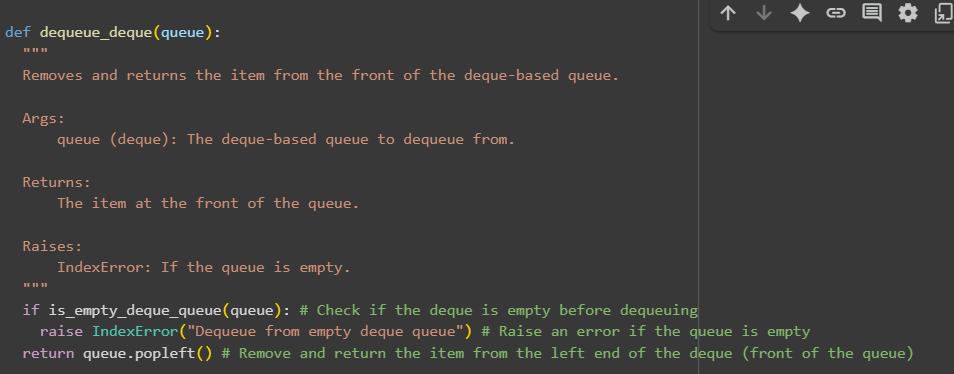
**Task: Implement a Queue with enqueue(), dequeue(), and is\_empty() methods.**

**PROMPT1:** write a python code for FIFO with functions like enqueue(), dequeue and is\_empty methods using python lists

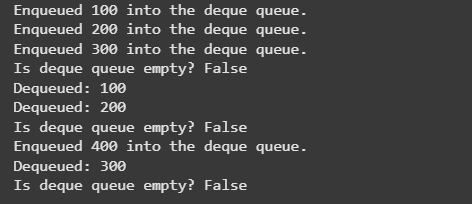
**PROMPT2:** Review the performance and customize write the code using collections.deque

**CODE:**

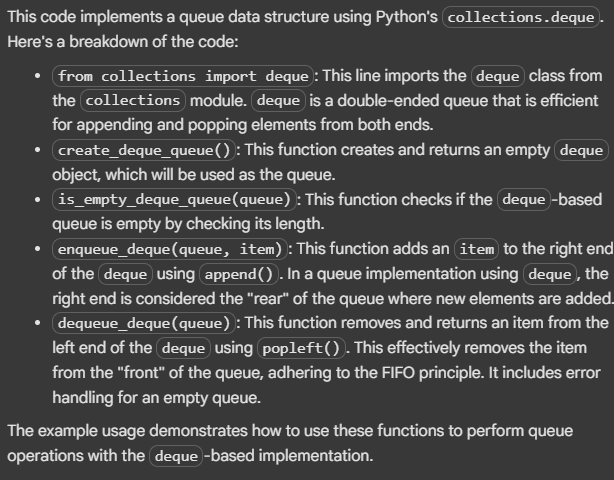




**OUTPUT:**

****

**EXPLANATION:**

****

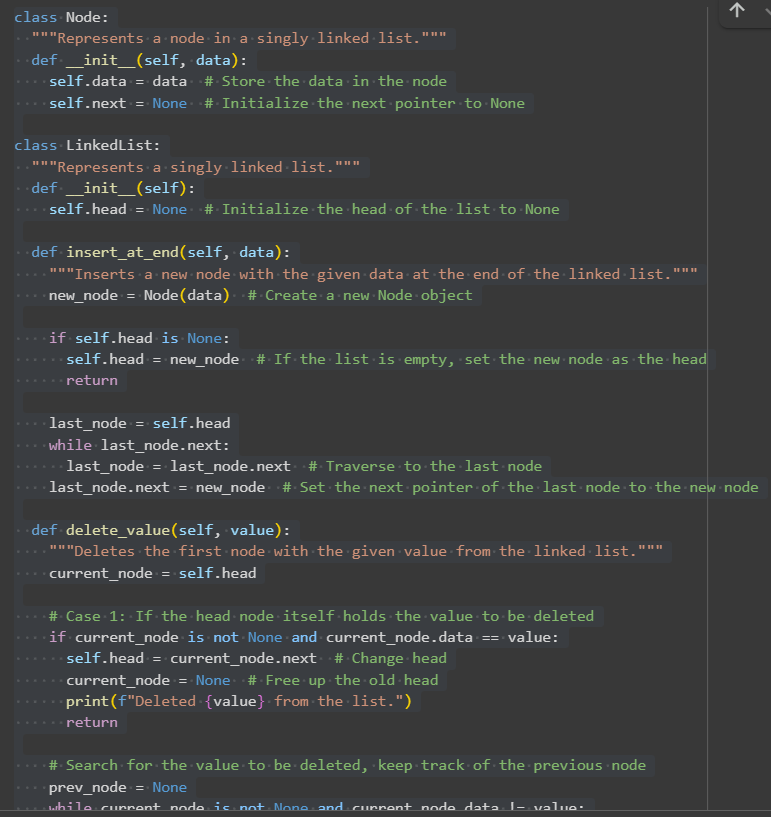
**Task 3: Singly Linked List with Traversal**

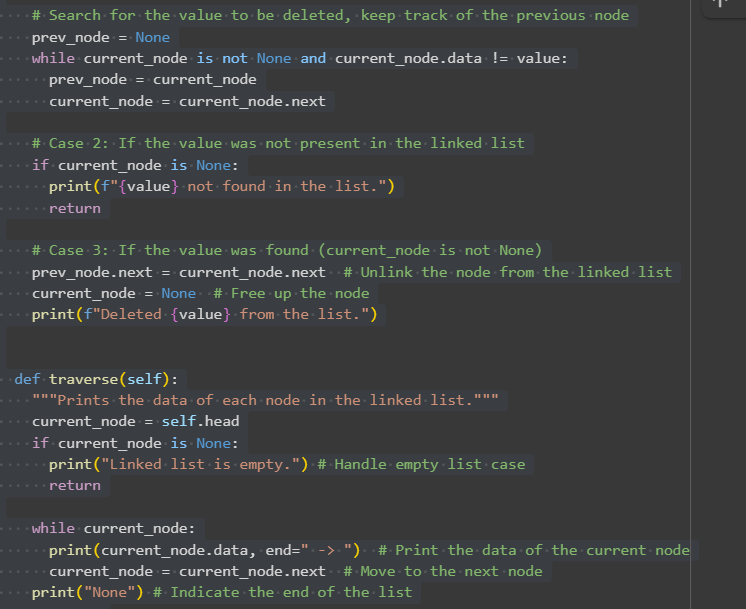
**Task: Implement a Singly Linked List with operations: insert\_at\_end(), delete\_value(), and traverse().**

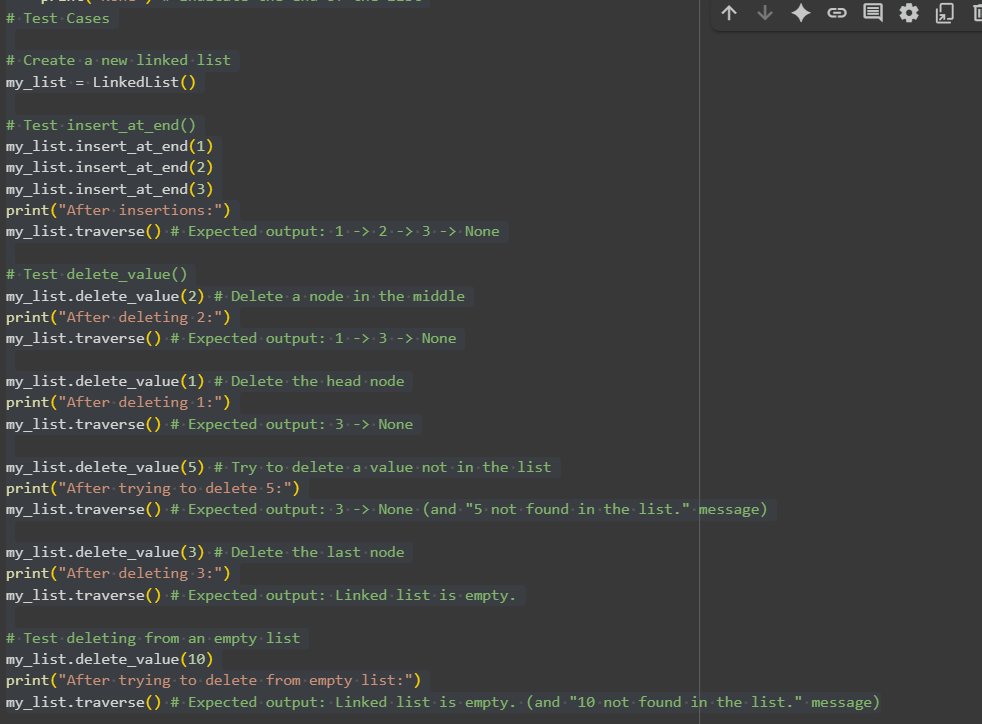
**PROMPT:** Write a python code having singly linked list (Node, LinkedList). having functions like insert\_at\_end(),delete\_value(),and traverse().Write inline comments for every step which are non-trivial

and suggest few test cases.

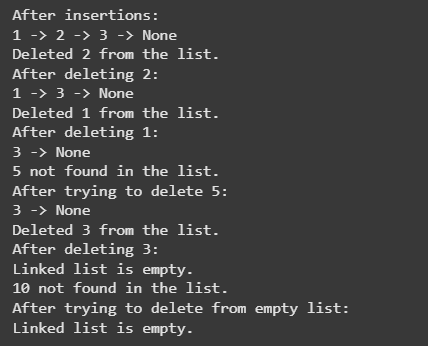
**CODE:**

****

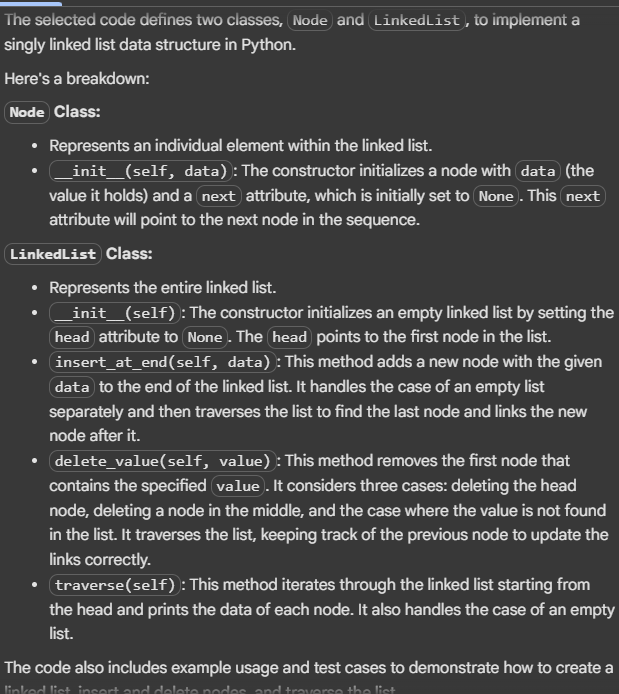
****

****

**OUTPUT:**

****

**EXPLANATION:**

****

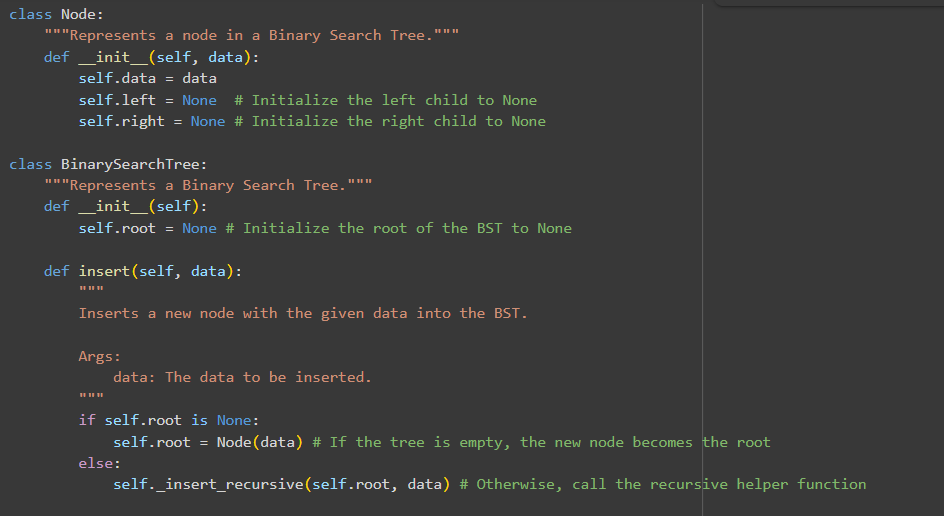
**Task 4: Binary Search Tree (BST)**

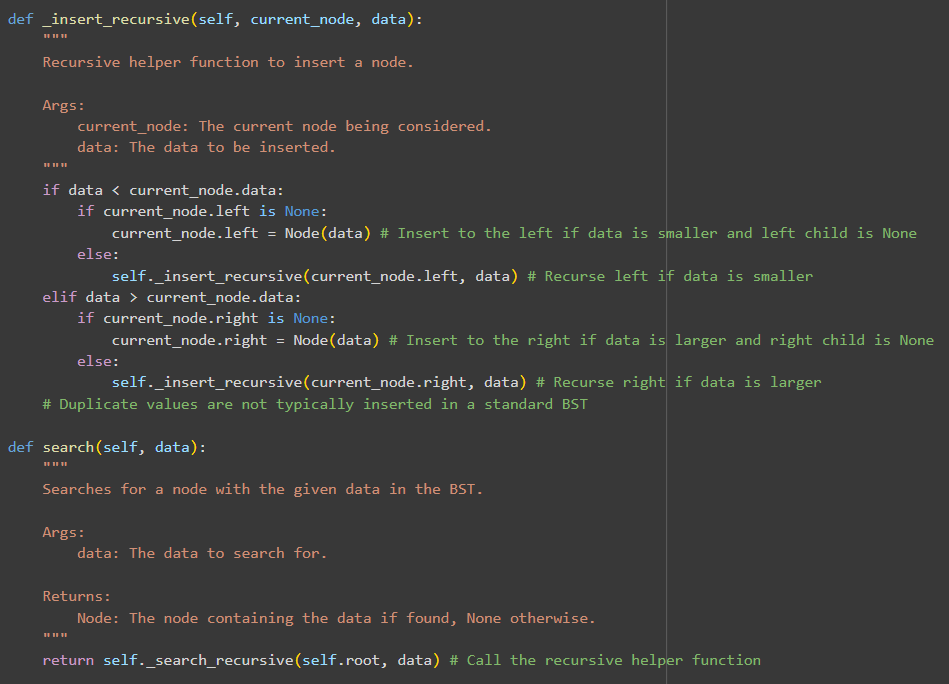
**Task: Implement a Binary Search Tree with methods for insert(), search(), and inorder\_traversal().**

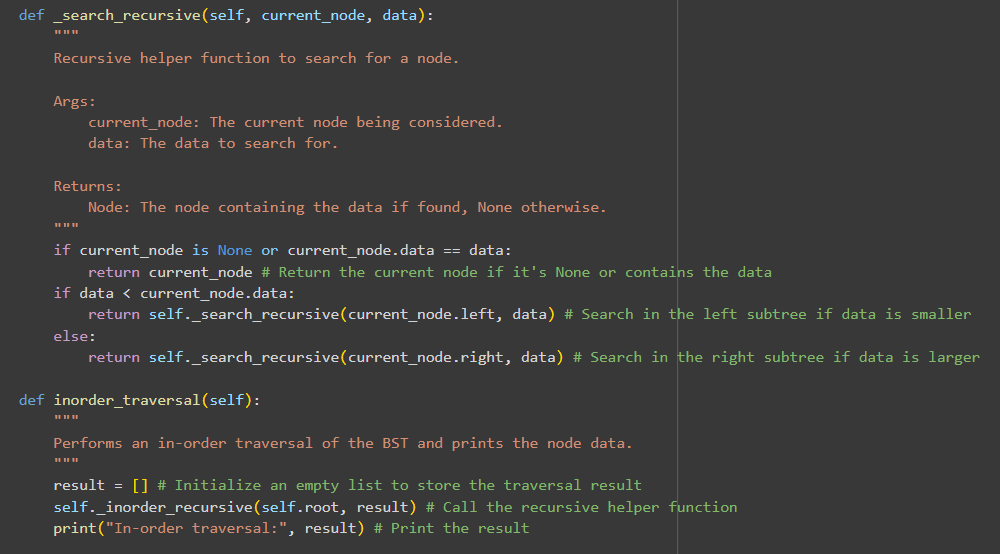
**PROMPT:** Write a python code using Binary Search Tree having fuction methods like insert(),search(), and inorder\_tranversal(). Write function methods which are missed and add doctrings to all

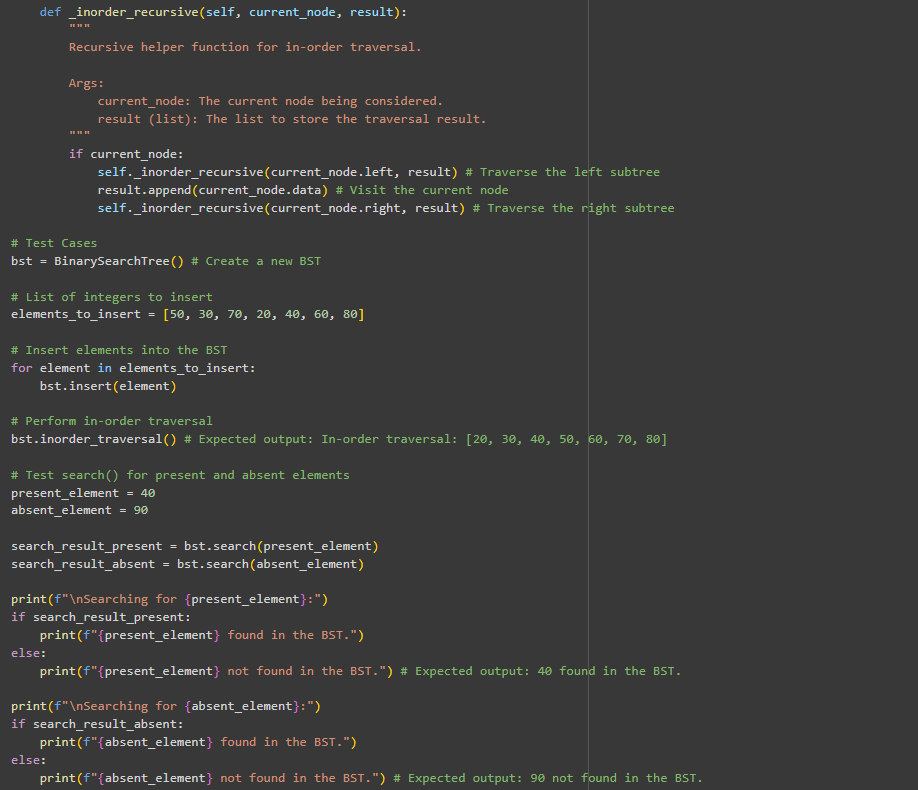
functions.Finally Test with a list of integers and compare outputs of search() for present vs absent elements.

**CODE:**

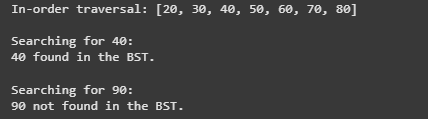


****

****

****

**OUTPUT:**

****

**EXPLANATION:**

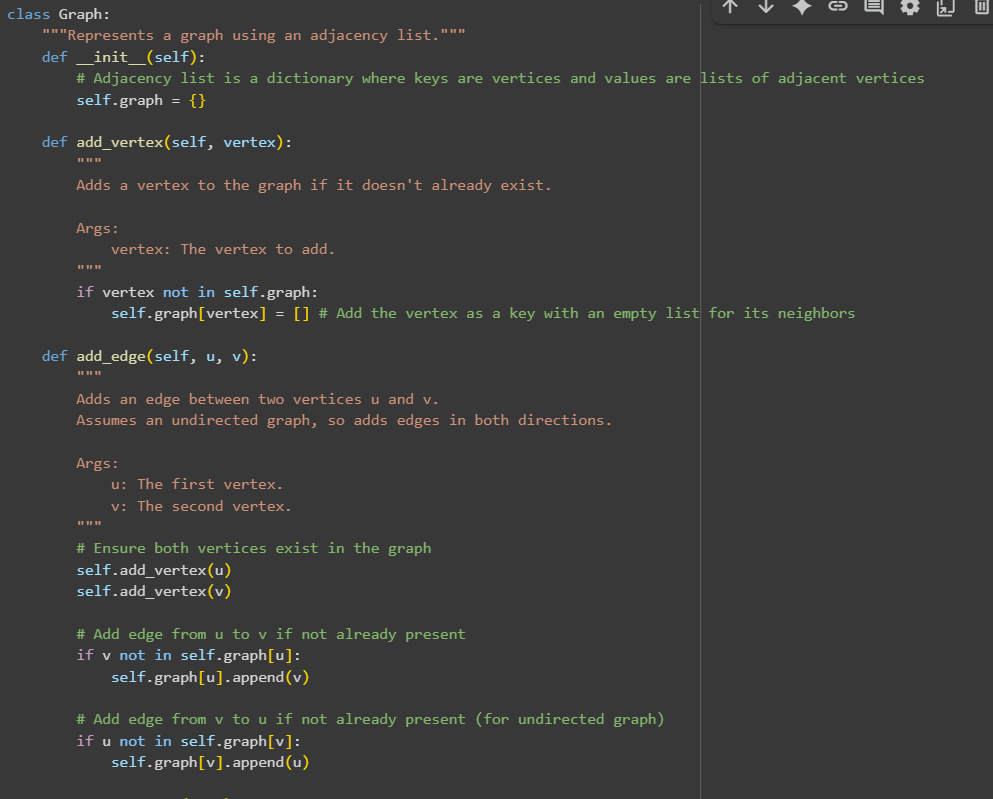
****

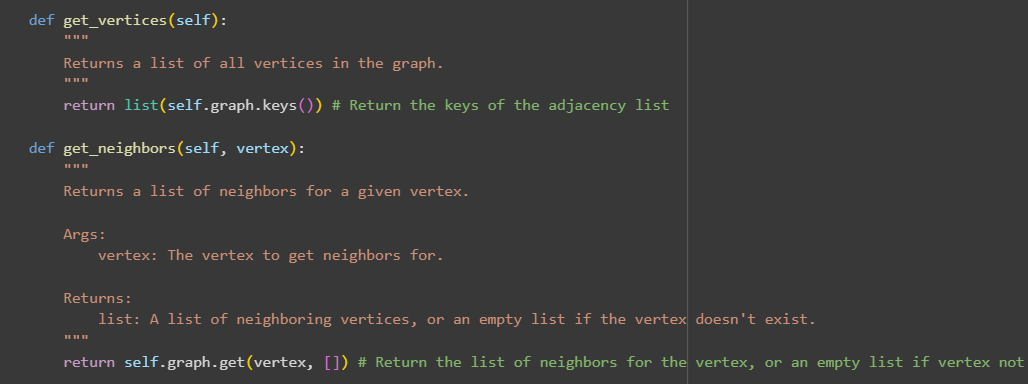
**Task 5: Graph Representation and BFS/DFS Traversal**

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

**PROMPT:** Write a python code to impliment a graph using an adjancency list having traversal methods like BFS(),DFS(). Add inline comments to every line.

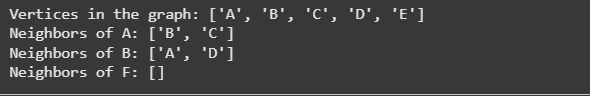
**CODE:**

****

****

****

**OUTPUT:**

****

**EXPLANATION:**

This code defines a Graph class that represents a graph using an adjacency list. Here's a breakdown of the code:

* Graph Class:
  + \_\_init\_\_(self): The constructor initializes the graph with an empty dictionary self.graph. This dictionary serves as the adjacency list, where keys will be vertices and the corresponding values will be lists of their adjacent vertices.
  + add\_vertex(self, vertex): This method adds a vertex to the graph if it doesn't already exist as a key in the self.graph dictionary. If it's a new vertex, it's added as a key with an empty list as its value to store its future neighbors.
  + add\_edge(self, u, v): This method adds an edge between two vertices u and v. It first ensures that both u and v exist in the graph by calling add\_vertex() for each. Then, it adds v to the list of neighbors for u and u to the list of neighbors for v. This implementation assumes an undirected graph, hence adding the edge in both directions.
  + get\_vertices(self): This method returns a list of all the vertices currently in the graph by extracting the keys from the self.graph dictionary.
  + get\_neighbors(self, vertex): This method returns the list of neighbors for a given vertex. It uses the .get() method on the self.graph dictionary, which safely returns an empty list if the vertex is not found in the graph, preventing a KeyError.
* Example Usage:
  + An instance of the Graph class named g is created.
  + Several vertices ('A', 'B', 'C', 'D', 'E') are added to the graph using the add\_vertex() method.
  + Edges are added between vertices using the add\_edge() method, defining the connections in the graph.
  + Finally, the code demonstrates how to use get\_vertices() to see all the vertices and get\_neighbors() to see the connections for specific vertices, including handling a vertex that doesn't exist ('F').