AI ASSIGNMENT:11.4

HTNO:2403A51284

BATCH:12

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

PROMPT:

Generate a **code skeleton** for the Stack class with the following operations:

- push(item)
- pop()
- peek()
- is_empty()

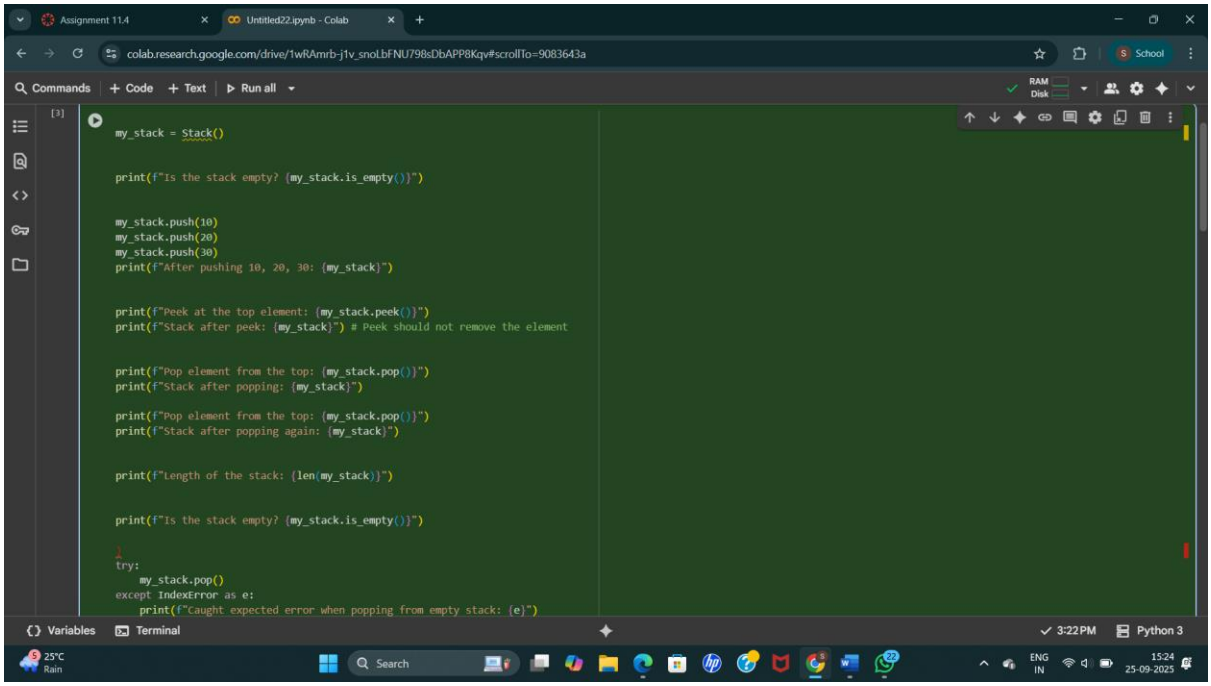
Add **Google-style docstrings** to each method.

Include **inline comments** to explain any tricky parts.

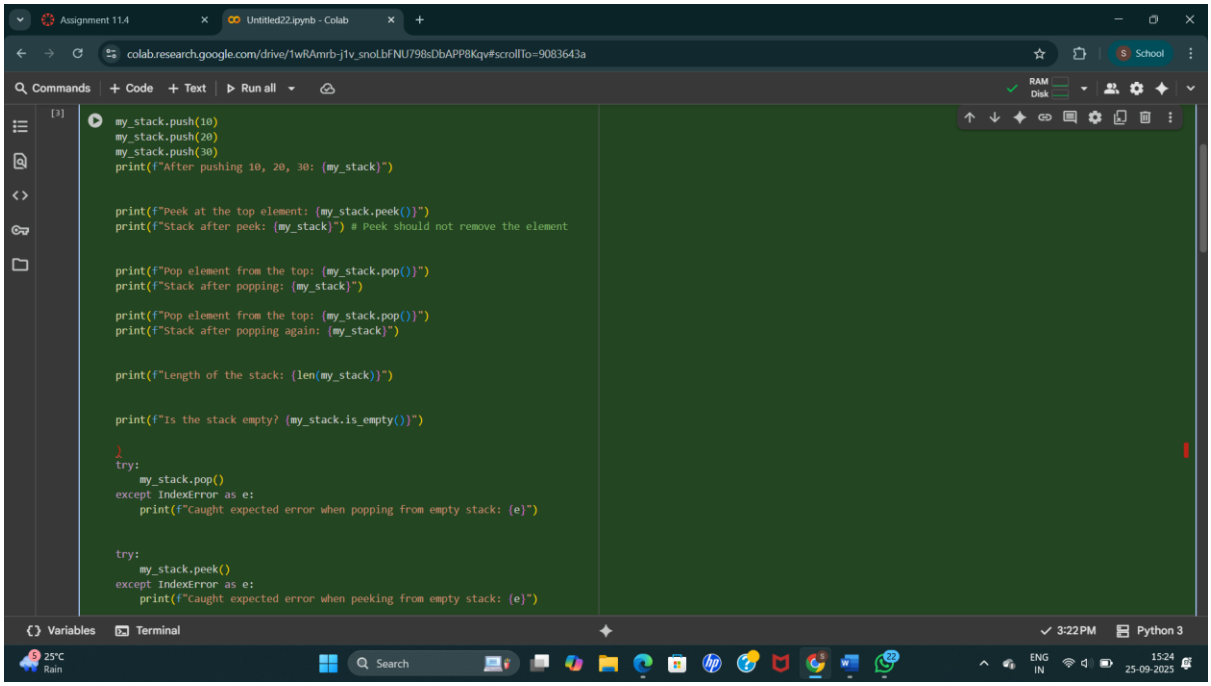
Show me how to **test the stack operations** using sample data (like pushing and popping values).

Suggest possible **optimizations or alternative implementations** (for example, using collections. deque instead of a list).

CODE:

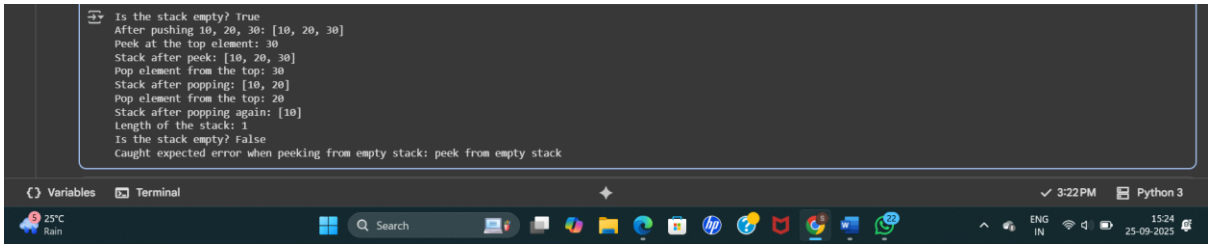


```
my_stack = Stack()


print(f"Is the stack empty? {my_stack.is_empty()}")


my_stack.push(10)
my_stack.push(20)
my_stack.push(30)
print(f"After pushing 10, 20, 30: {my_stack}")


print(f"Peek at the top element: {my_stack.peek()}")
print(f"Stack after peek: {my_stack}") # Peek should not remove the element


print(f"Pop element from the top: {my_stack.pop()}")
print(f"Stack after popping: {my_stack}")

print(f"Pop element from the top: {my_stack.pop()}")
print(f"Stack after popping again: {my_stack}")


print(f"Length of the stack: {len(my_stack)}")


print(f"Is the stack empty? {my_stack.is_empty()}")


try:
    my_stack.pop()
except IndexError as e:
    print(f"Caught expected error when popping from empty stack: {e}")
```



```
my_stack.push(10)
my_stack.push(20)
my_stack.push(30)
print(f"After pushing 10, 20, 30: {my_stack}")


print(f"Peek at the top element: {my_stack.peek()}")
print(f"Stack after peek: {my_stack}") # Peek should not remove the element


print(f"Pop element from the top: {my_stack.pop()}")
print(f"Stack after popping: {my_stack}")

print(f"Pop element from the top: {my_stack.pop()}")
print(f"Stack after popping again: {my_stack}")


print(f"Length of the stack: {len(my_stack)}")


print(f"Is the stack empty? {my_stack.is_empty()}")


try:
    my_stack.pop()
except IndexError as e:
    print(f"Caught expected error when popping from empty stack: {e}")


try:
    my_stack.peek()
except IndexError as e:
    print(f"Caught expected error when peeking from empty stack: {e}")
```

OUTPUT:



```
Is the stack empty? True
After pushing 10, 20, 30: [10, 20, 30]
Peek at the top element: 30
Stack after peek: [10, 20, 30]
Pop element from the top: 30
Stack after popping: [10, 20]
Pop element from the top: 20
Stack after popping again: [10]
Length of the stack: 1
Is the stack empty? False
Caught expected error when peeking from empty stack: peek from empty 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

PROMPT:

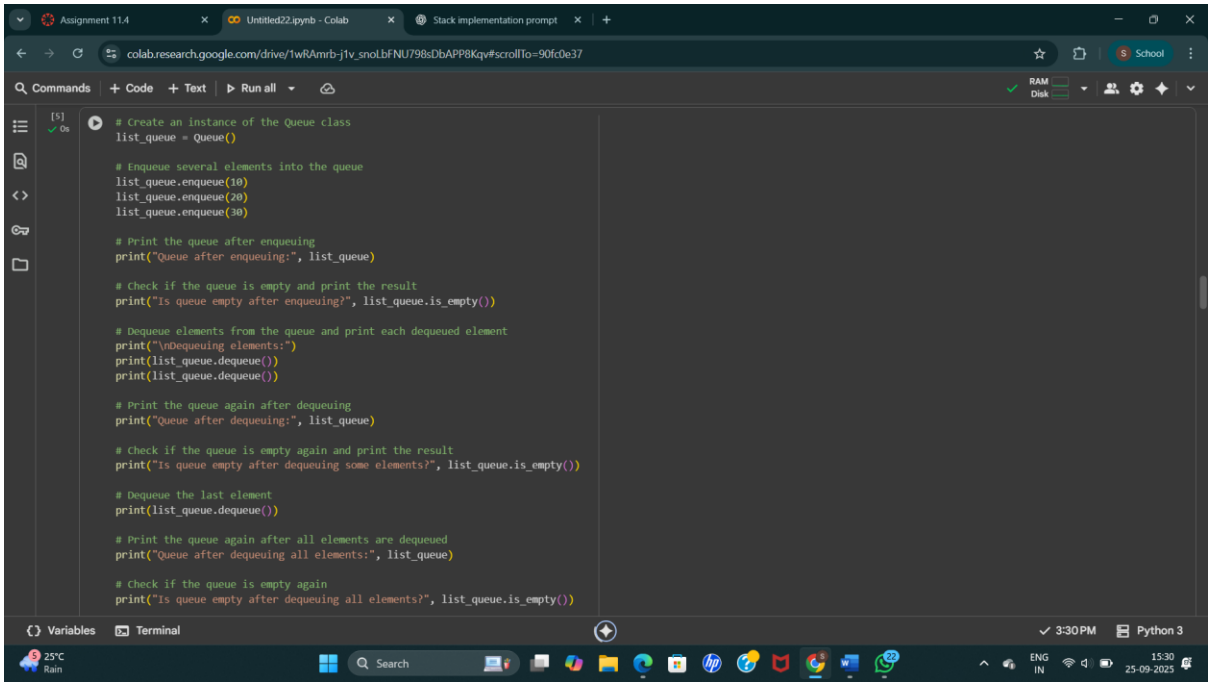
I want to implement a **Queue** in Python with the following operations:

- enqueue(item)
- dequeue()
- is_empty()

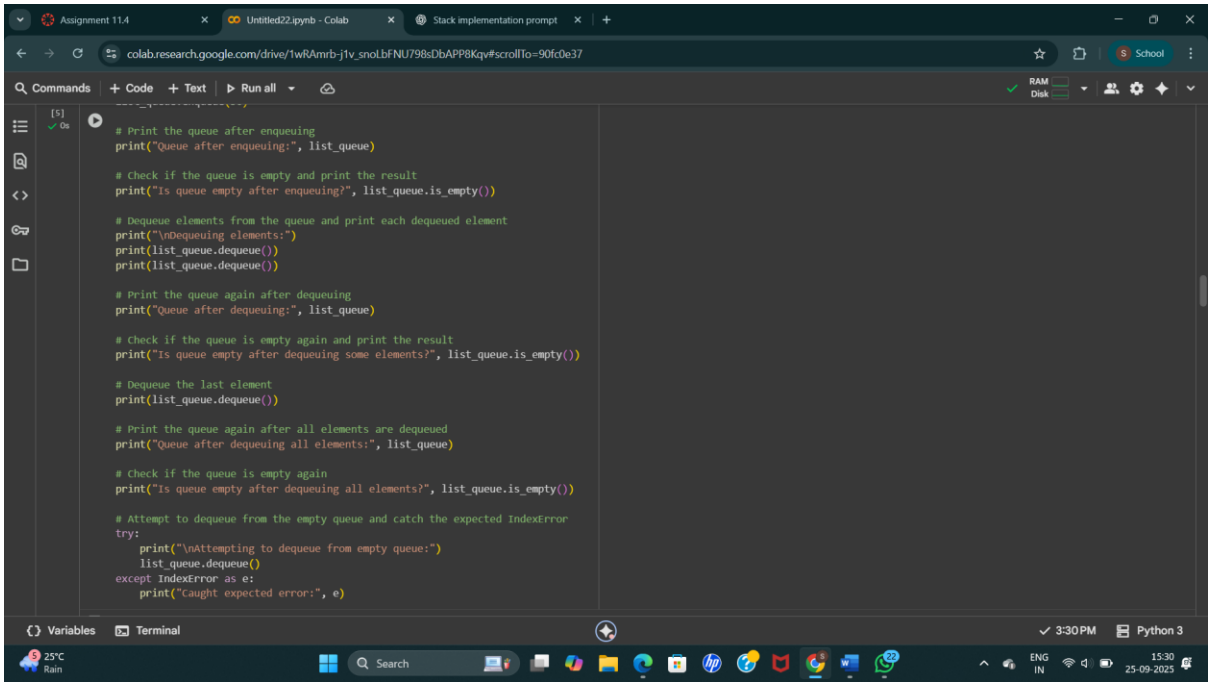
Please do the following:

1. First, implement the Queue using a **Python list**.
2. Then, review the **performance issues** of using lists for queue operations.
3. Suggest and implement a more **efficient version using collections.deque**.
4. Provide a **performance comparison** (time complexity analysis) between both implementations.
5. Add **Google-style docstrings** and inline comments to explain tricky parts.
6. Show me **sample test cases** that demonstrate both versions of the queue in action.

CODE:

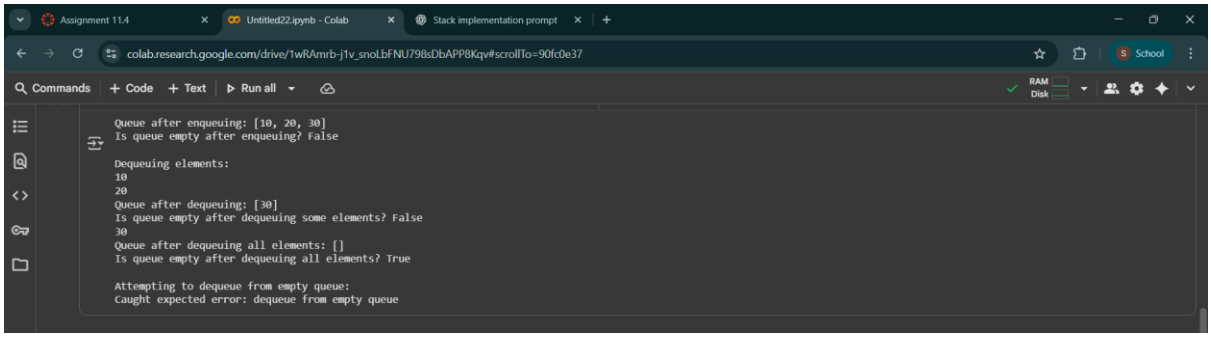


```
# Create an instance of the Queue class
list_queue = Queue()

# Enqueue several elements into the queue
list_queue.enqueue(10)
list_queue.enqueue(20)
list_queue.enqueue(30)

# Print the queue after enqueuing
print("Queue after enqueuing:", list_queue)

# Check if the queue is empty and print the result
print("Is queue empty after enqueuing?", list_queue.is_empty())

# Dequeue elements from the queue and print each dequeued element
print("\nDequeuing elements:")
print(list_queue.dequeue())
print(list_queue.dequeue())

# Print the queue again after dequeuing
print("Queue after dequeuing:", list_queue)

# Check if the queue is empty again and print the result
print("Is queue empty after dequeuing some elements?", list_queue.is_empty())

# Dequeue the last element
print(list_queue.dequeue())

# Print the queue again after all elements are dequeued
print("Queue after dequeuing all elements:", list_queue)

# Check if the queue is empty again
print("Is queue empty after dequeuing all elements?", list_queue.is_empty())
```



```
# Print the queue after enqueuing
print("Queue after enqueuing:", list_queue)

# Check if the queue is empty and print the result
print("Is queue empty after enqueuing?", list_queue.is_empty())

# Dequeue elements from the queue and print each dequeued element
print("\nDequeuing elements:")
print(list_queue.dequeue())
print(list_queue.dequeue())

# Print the queue again after dequeuing
print("Queue after dequeuing:", list_queue)

# Check if the queue is empty again and print the result
print("Is queue empty after dequeuing some elements?", list_queue.is_empty())

# Dequeue the last element
print(list_queue.dequeue())

# Print the queue again after all elements are dequeued
print("Queue after dequeuing all elements:", list_queue)

# Check if the queue is empty again
print("Is queue empty after dequeuing all elements?", list_queue.is_empty())

# Attempt to dequeue from the empty queue and catch the expected IndexError
try:
    print("\nAttempting to dequeue from empty queue:")
    list_queue.dequeue()
except IndexError as e:
    print("Caught expected error:", e)
```

OUTPUT:



```
Queue after enqueuing: [10, 20, 30]
Is queue empty after enqueuing? False

Dequeuing elements:
10
20
Queue after dequeuing: [30]
Is queue empty after dequeuing some elements? False
30
Queue after dequeuing all elements: []
Is queue empty after dequeuing all elements? True

Attempting to dequeue from empty queue:
Caught expected error: dequeue from empty queue
```

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.

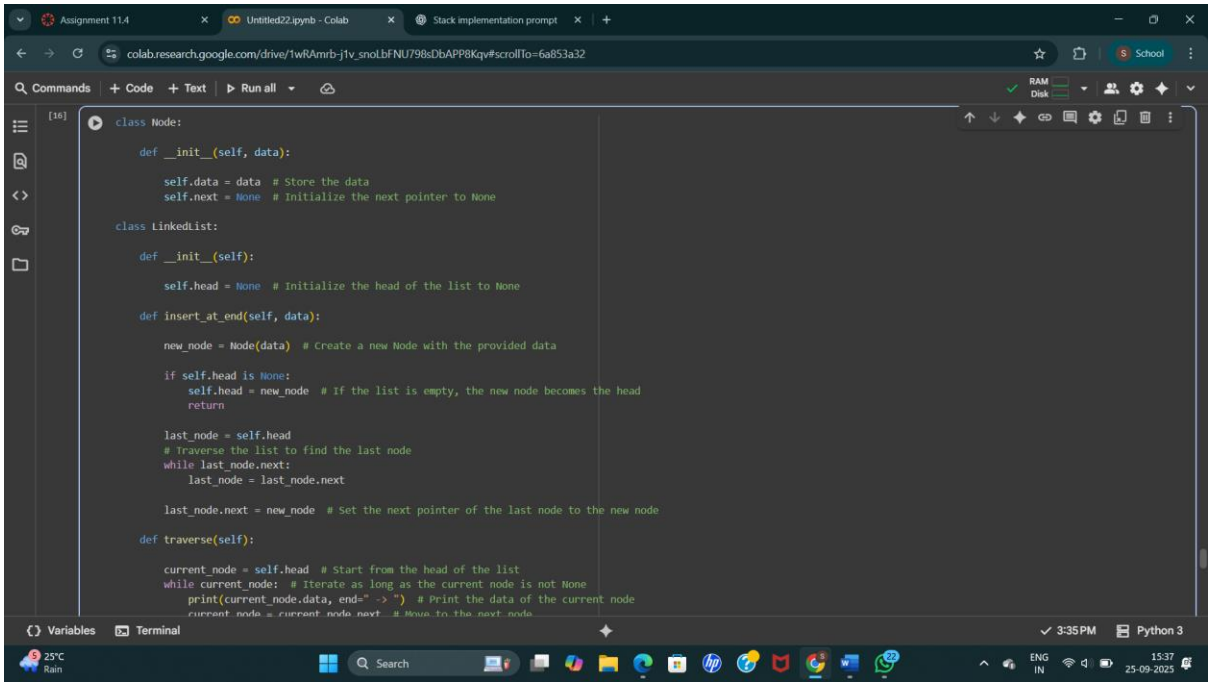
PROMPT:

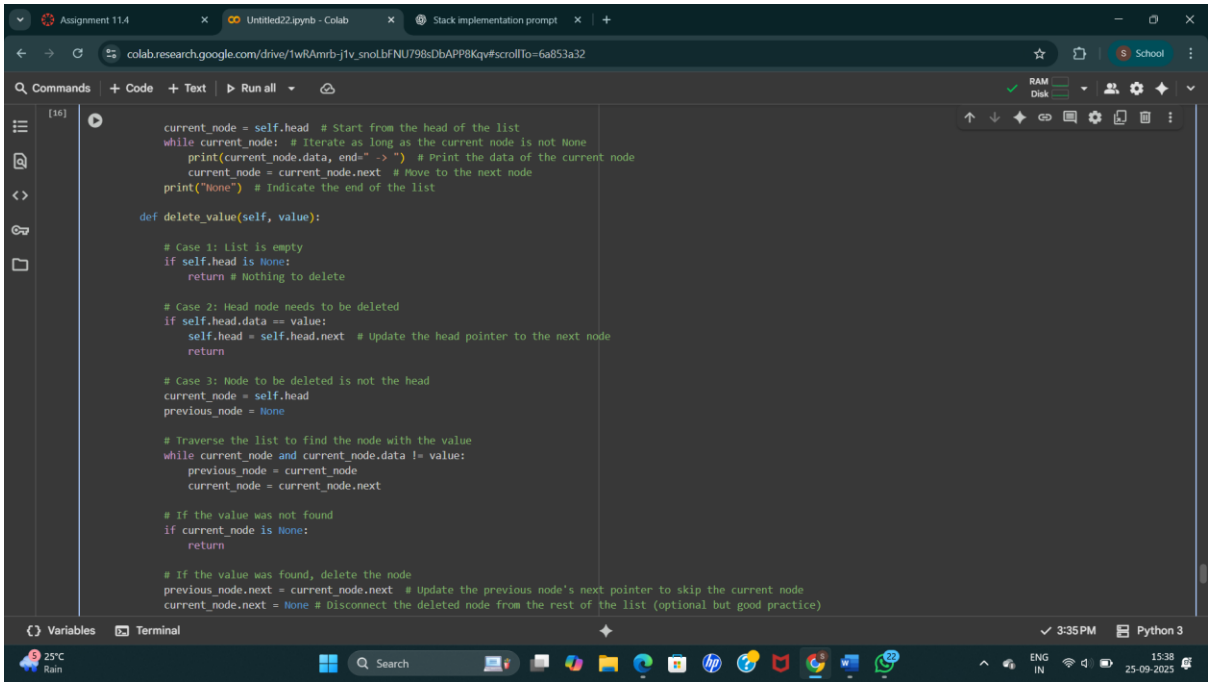
1. Create a **class-based implementation** with Node and LinkedList classes.
2. Implement these methods:
    - insert_at_end(value)
    - delete_value(value)
    - traverse()
3. Add **inline comments** explaining pointer updates clearly (since linked list insertions and deletions can be tricky).
4. Use **Google-style docstrings** for all classes and methods.
5. Suggest **test cases** to validate all operations, including:
    - inserting multiple values
    - deleting a value in the middle, first, and last node
    - attempting to delete a value not in the list
    - traversing an empty list
6. Show how to run these test cases and the expected outputs.

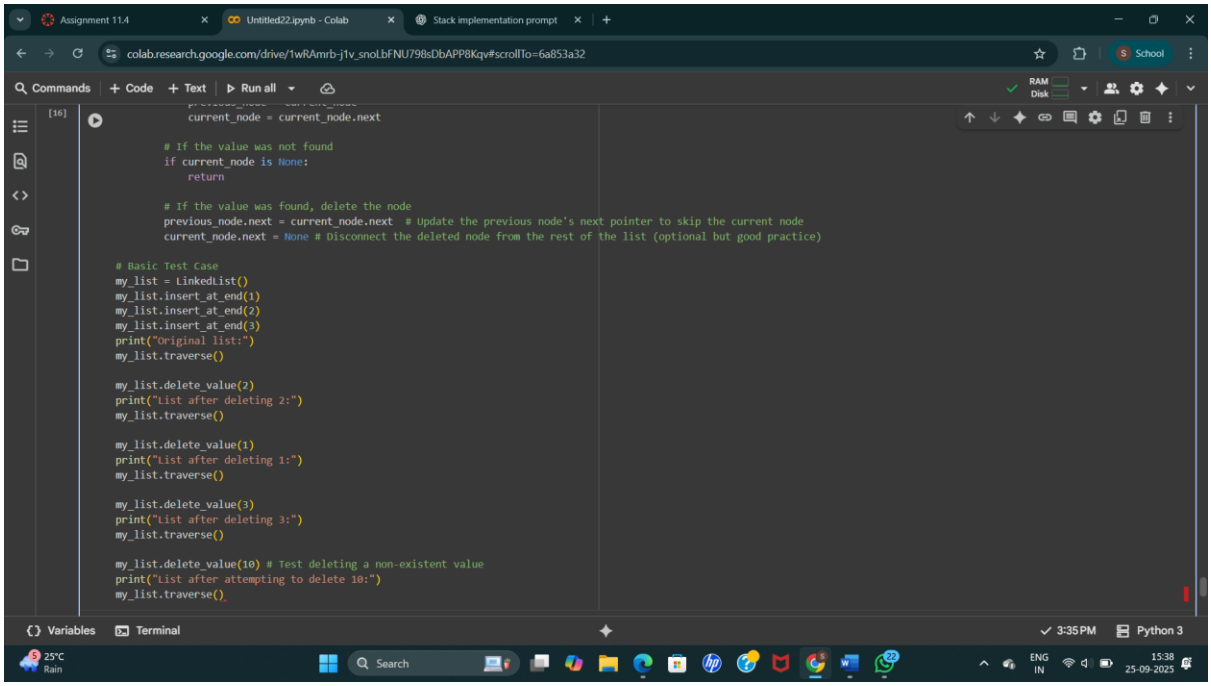
Expected Output:

- A **working singly linked list implementation**
- **Clear comments** for pointer updates
- **Suggested test cases** that cover normal and edge cases

CODE:

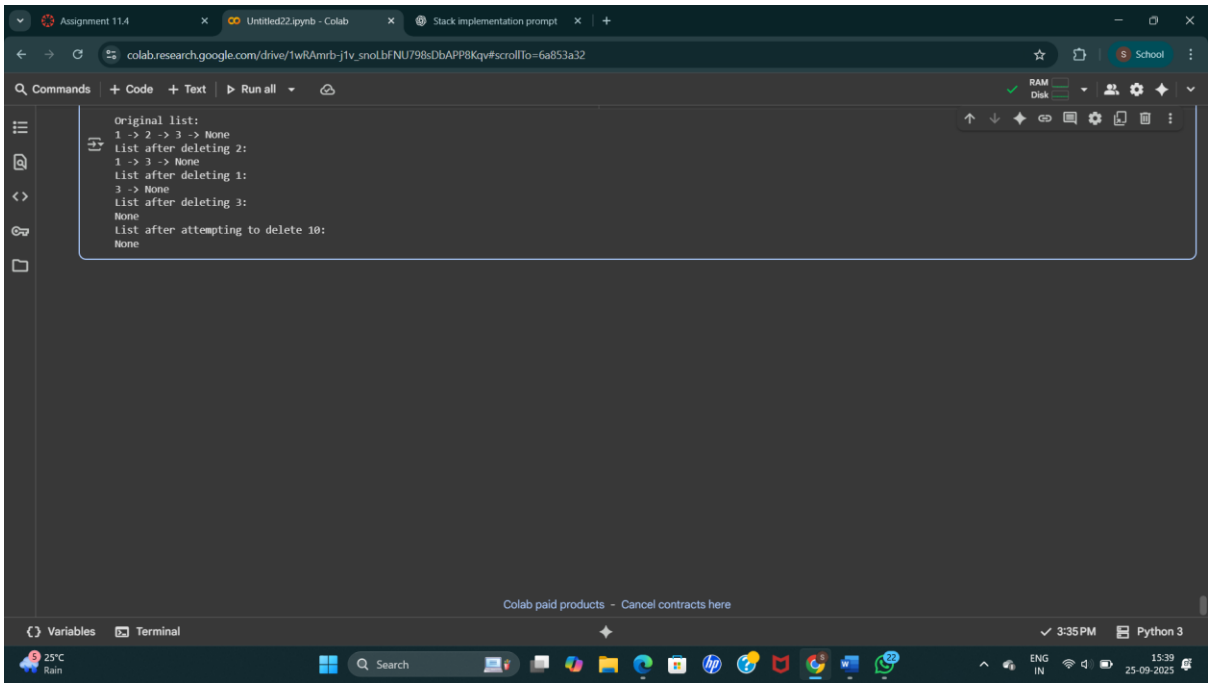

Assignment 11.4
Untitled22.ipynb - Colab
Stack implementation prompt
colab.research.google.com/drive/1wRAmrb-j1v_snoLbFNU798sDbAPP8Kqv#scrollTo=6a853a32
Commands
+ Code
+ Text
Run all
[16]
class Node:
def __init__(self, data):
self.data = data # Store the data
self.next = None # Initialize the next pointer to None
class LinkedList:
def __init__(self):
self.head = None # Initialize the head of the list to None
def insert_at_end(self, data):
new_node = Node(data) # Create a new Node with the provided data
if self.head is None:
self.head = new_node # If the list is empty, the new node becomes the head
return
last_node = self.head
# Traverse the list to find the last node
while last_node.next:
last_node = last_node.next
last_node.next = new_node # Set the next pointer of the last node to the new node
def traverse(self):
current_node = self.head # Start from the head of the list
while current_node: # Iterate as long as the current node is not None
print(current_node.data, end=" -> ") # Print the data of the current node
Variables
Terminal
3:35PM
Python 3
25°C
Rain
Search
ENG
IN
15:37
25-09-2025


Assignment 11.4
Untitled22.ipynb - Colab
Stack implementation prompt
colab.research.google.com/drive/1wRAmrb-j1v_snoLbFNU798sDbAPP8Kqv#scrollTo=6a853a32
Commands
+ Code
+ Text
Run all
[16]
current_node = self.head # Start from the head of the list
while current_node: # Iterate as long as the current node is not None
print(current_node.data, end=" -> ") # Print the data of the current node
current_node = current_node.next # Move to the next node
print("None") # Indicate the end of the list
def delete_value(self, value):
# Case 1: List is empty
if self.head is None:
return # Nothing to delete
# Case 2: Head node needs to be deleted
if self.head.data == value:
self.head = self.head.next # Update the head pointer to the next node
return
# Case 3: Node to be deleted is not the head
current_node = self.head
previous_node = None
# Traverse the list to find the node with the value
while current_node and current_node.data != value:
previous_node = current_node
current_node = current_node.next
# If the value was not found
if current_node is None:
return
# If the value was found, delete the node
previous_node.next = current_node.next # Update the previous node's next pointer to skip the current node
current_node.next = None # Disconnect the deleted node from the rest of the list (optional but good practice)
Variables
Terminal
3:35PM
Python 3
25°C
Rain
Search
ENG
IN
15:38
25-09-2025



```
            current_node = current_node.next

        # If the value was not found
        if current_node is None:
            return

        # If the value was found, delete the node
        previous_node.next = current_node.next  # Update the previous node's next pointer to skip the current node
        current_node.next = None # Disconnect the deleted node from the rest of the list (optional but good practice)

# Basic Test Case
my_list = LinkedList()
my_list.insert_at_end(1)
my_list.insert_at_end(2)
my_list.insert_at_end(3)
print("Original list:")
my_list.traverse()

my_list.delete_value(2)
print("List after deleting 2:")
my_list.traverse()

my_list.delete_value(1)
print("List after deleting 1:")
my_list.traverse()

my_list.delete_value(3)
print("List after deleting 3:")
my_list.traverse()

my_list.delete_value(10) # Test deleting a non-existent value
print("List after attempting to delete 10:")
my_list.traverse()
```

OUTPUT:



```
Original list:
1 -> 2 -> 3 -> None
List after deleting 2:
1 -> 3 -> None
List after deleting 1:
3 -> None
List after deleting 3:
None
List after attempting to delete 10:
None
```

Task 4: Binary Search Tree (BST)

• Task: Implement a Binary Search Tree with methods for insert(), search(), and in order_ 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

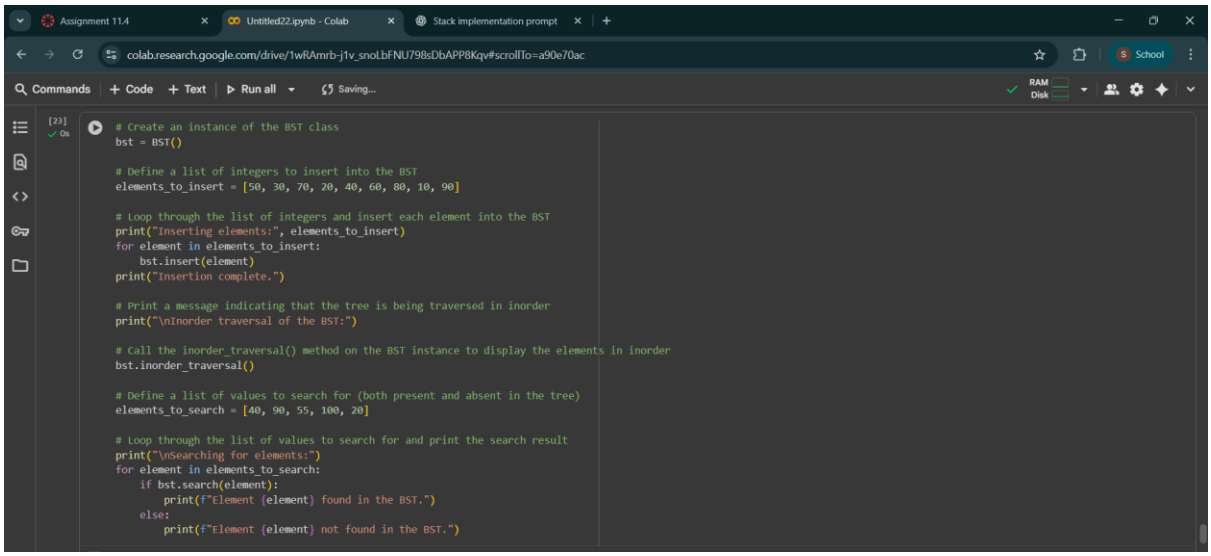
PROMPT:

1. Complete the missing method
    - insert(value)
    - search(value)
    - in order _traversal ()
2. Add **Google-style docstrings** for each method.
3. Add **inline comments** to explain key logic (like recursive insertions, traversal order, etc.).
4. Show me how to **test the BST** with a list of integers (e.g., [50, 30, 70, 20, 40, 60, 80]).
5. Demonstrate how search() behaves for both **present** and **absent** values.

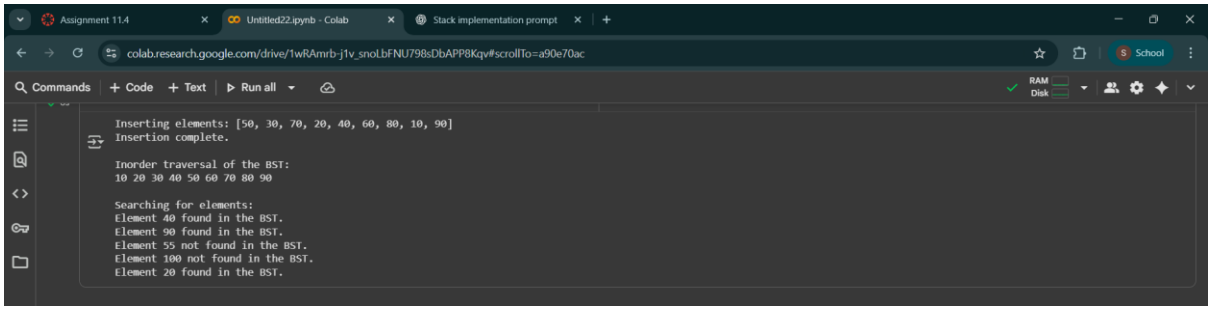
Expected Output:

- A **working BST class** with clear docstrings.
- **Correct inorder traversal output** (sorted order).
- **Search results** for existing and non-existing elements.

CODE:



OUTPUT:



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.

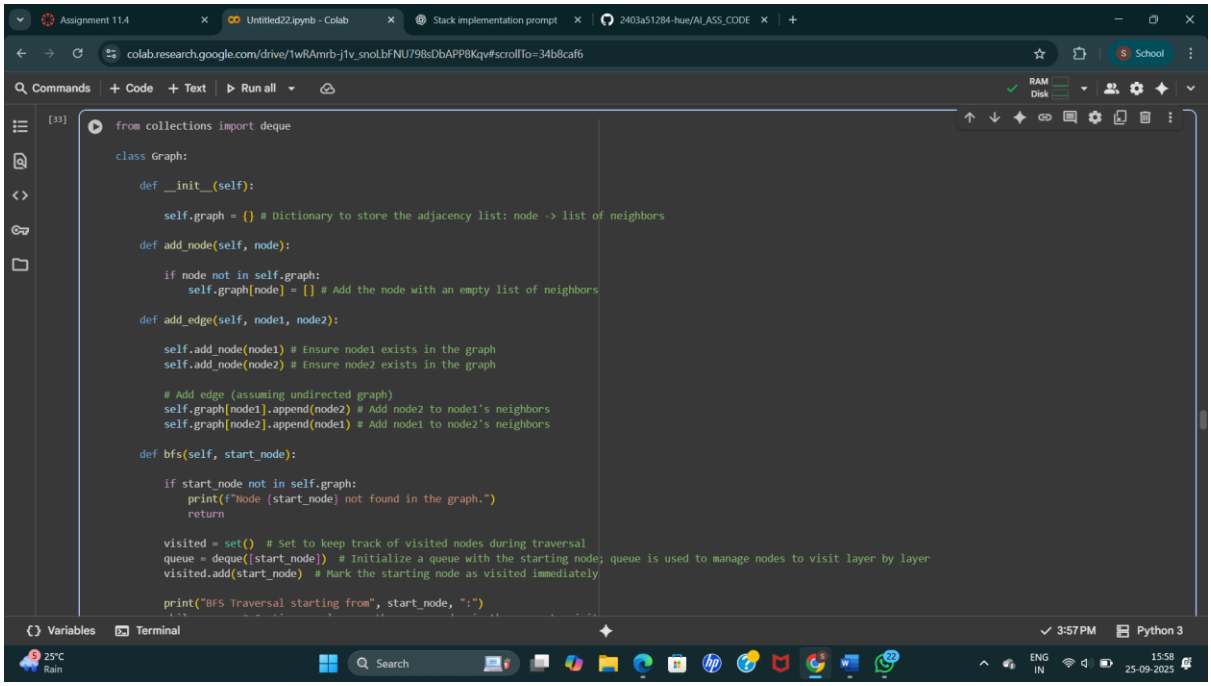
PROMPT:

1. Implement methods for:
    - BFS(start_vertex)
    - DFS(start_vertex)
2. Add **inline comments** explaining each traversal step (queue usage in BFS, stack/recursion in DFS).
3. Provide both **iterative DFS** (using stack) and **recursive DFS** versions, and compare them.
4. Add **Google-style docstrings** to each method.
5. Show sample tests with a small graph (e.g., undirected graph with vertices A, B, C, D, E) and demonstrate BFS/DFS results starting from a given node.

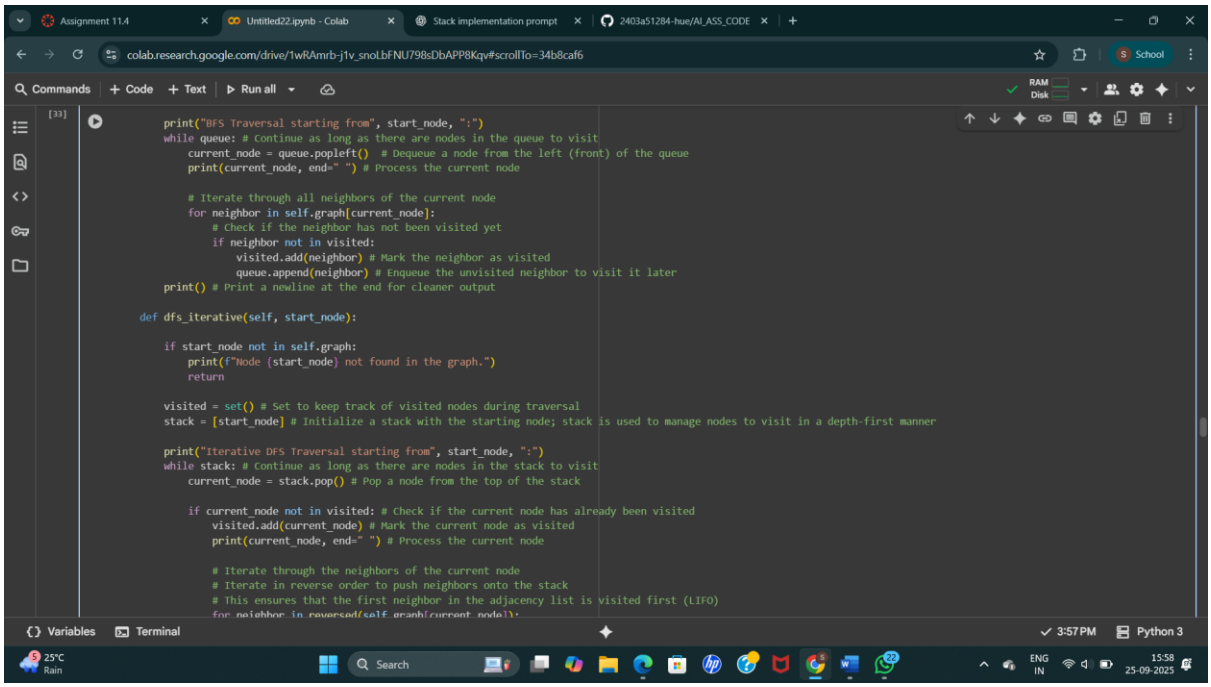
Expected Output:

- A graph implementation with BFS and DFS methods.
- Clear inline comments explaining traversal logic.
- Example runs showing traversal order.

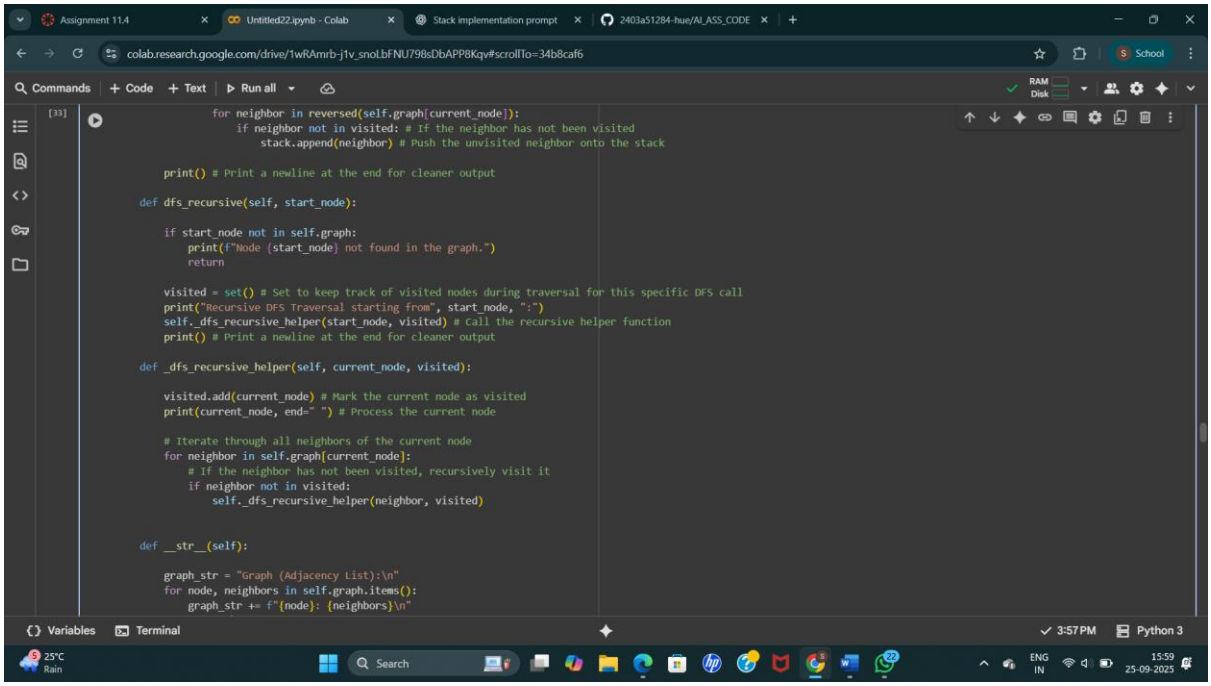
CODE:



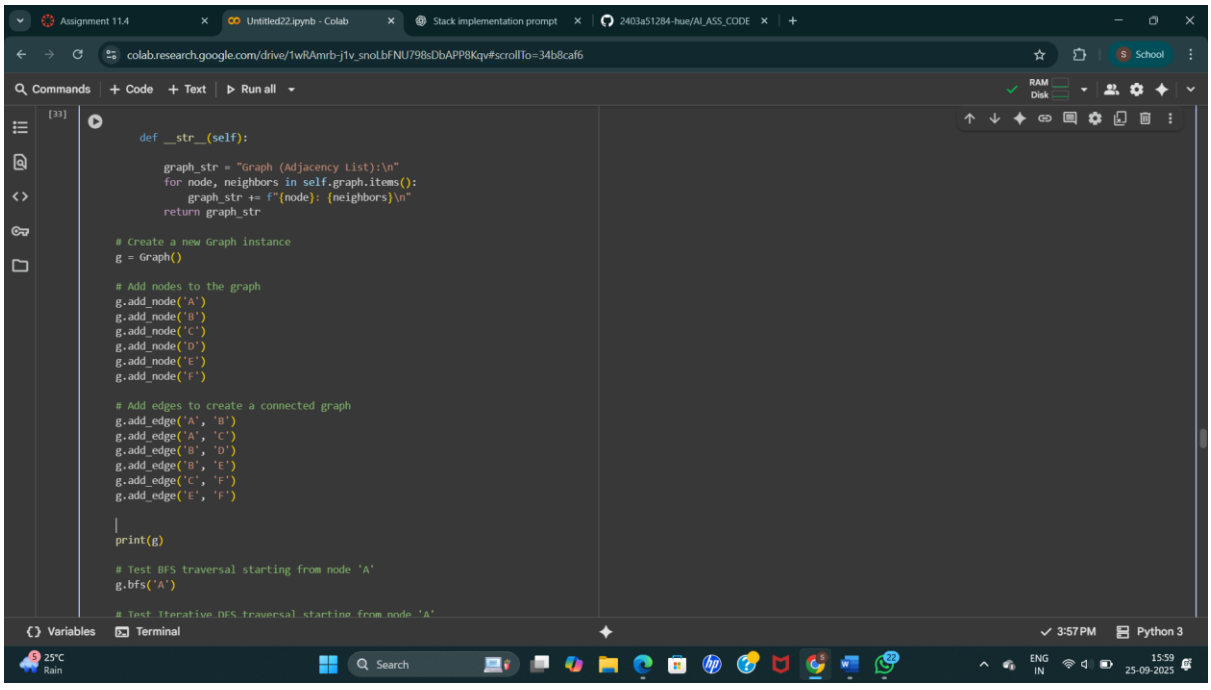
```
[33]
from collections import deque

class Graph:

    def __init__(self):

        self.graph = {} # Dictionary to store the adjacency list: node -> list of neighbors

    def add_node(self, node):

        if node not in self.graph:
            self.graph[node] = [] # Add the node with an empty list of neighbors

    def add_edge(self, node1, node2):

        self.add_node(node1) # Ensure node1 exists in the graph
        self.add_node(node2) # Ensure node2 exists in the graph

        # Add edge (assuming undirected graph)
        self.graph[node1].append(node2) # Add node2 to node1's neighbors
        self.graph[node2].append(node1) # Add node1 to node2's neighbors

    def bfs(self, start_node):

        if start_node not in self.graph:
            print(f"Node {start_node} not found in the graph.")
            return

        visited = set()  # Set to keep track of visited nodes during traversal
        queue = deque([start_node])  # Initialize a queue with the starting node; queue is used to manage nodes to visit layer by layer
        visited.add(start_node)  # Mark the starting node as visited immediately

        print("BFS Traversal starting from", start_node, ":")
```



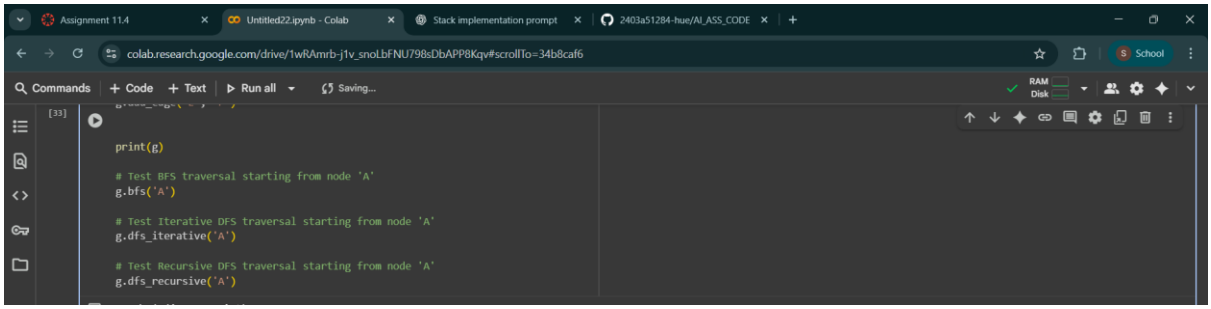
```
[33]
        print("BFS Traversal starting from", start_node, ":")
        while queue: # Continue as long as there are nodes in the queue to visit
            current_node = queue.popleft()  # Dequeue a node from the left (front) of the queue
            print(current_node, end=" ") # Process the current node

            # Iterate through all neighbors of the current node
            for neighbor in self.graph[current_node]:
                # Check if the neighbor has not been visited yet
                if neighbor not in visited:
                    visited.add(neighbor) # Mark the neighbor as visited
                    queue.append(neighbor) # Enqueue the unvisited neighbor to visit it later
        print() # Print a newline at the end for cleaner output

    def dfs_iterative(self, start_node):

        if start_node not in self.graph:
            print(f"Node {start_node} not found in the graph.")
            return

        visited = set() # Set to keep track of visited nodes during traversal
        stack = [start_node] # Initialize a stack with the starting node; stack is used to manage nodes to visit in a depth-first manner

        print("Iterative DFS Traversal starting from", start_node, ":")
        while stack: # Continue as long as there are nodes in the stack to visit
            current_node = stack.pop() # Pop a node from the top of the stack

            if current_node not in visited: # Check if the current node has already been visited
                visited.add(current_node) # Mark the current node as visited
                print(current_node, end=" ") # Process the current node

                # Iterate through the neighbors of the current node
                # Iterate in reverse order to push neighbors onto the stack
                # This ensures that the first neighbor in the adjacency list is visited first (LIFO)
                for neighbor in reversed(self.graph[current_node]):
```



```
                for neighbor in reversed(self.graph[current_node]):
                    if neighbor not in visited: # If the neighbor has not been visited
                        stack.append(neighbor) # Push the unvisited neighbor onto the stack

        print() # Print a newline at the end for cleaner output

    def dfs_recursive(self, start_node):

        if start_node not in self.graph:
            print(f"Node {start_node} not found in the graph.")
            return

        visited = set() # Set to keep track of visited nodes during traversal for this specific DFS call
        print("Recursive DFS Traversal starting from", start_node, ":")
        self._dfs_recursive_helper(start_node, visited) # Call the recursive helper function
        print() # Print a newline at the end for cleaner output

    def _dfs_recursive_helper(self, current_node, visited):

        visited.add(current_node) # Mark the current node as visited
        print(current_node, end=" ") # Process the current node

        # Iterate through all neighbors of the current node
        for neighbor in self.graph[current_node]:
            # If the neighbor has not been visited, recursively visit it
            if neighbor not in visited:
                self._dfs_recursive_helper(neighbor, visited)


    def __str__(self):

        graph_str = "Graph (Adjacency List):\n"
        for node, neighbors in self.graph.items():
            graph_str += f"{node}: {neighbors}\n"
```



```
    def __str__(self):

        graph_str = "Graph (Adjacency List):\n"
        for node, neighbors in self.graph.items():
            graph_str += f"{node}: {neighbors}\n"
        return graph_str

# Create a new Graph instance
g = Graph()

# Add nodes to the graph
g.add_node('A')
g.add_node('B')
g.add_node('C')
g.add_node('D')
g.add_node('E')
g.add_node('F')

# Add edges to create a connected 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('E', 'F')

print(g)

# Test BFS traversal starting from node 'A'
g.bfs('A')

# Test Iterative DFS traversal starting from node 'A'
```



```
print(g)

# Test BFS traversal starting from node 'A'
g.bfs('A')

# Test Iterative DFS traversal starting from node 'A'
g.dfs_iterative('A')

# Test Recursive DFS traversal starting from node 'A'
g.dfs_recursive('A')
```

OUTPUT:

```
Graph (Adjacency List):
A: ['B', 'C']
B: ['A', 'D', 'E']
C: ['A', 'F']
D: ['B']
E: ['B', 'F']
F: ['C', 'E']

BFS Traversal starting from A :
A B C D E F
Iterative DFS Traversal starting from A :
A B D E F C
Recursive DFS Traversal starting from A :
A B D E F C
```