

AI ASSISTED CODING

NAME:M.ABHINAND REDDY

HALL TNO:2403A51288

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:

Use AI to generate a Python Stack class with push(), pop(), peek(), and is_empty() methods.

- Include Google-style docstrings for each method.
- Add inline comments for tricky logic.
- Test stack operations with sample data.
- Ask AI to suggest optimizations or alternative implementations (e.g., using collections.deque).

The screenshot shows a Google Colab notebook titled 'Untitled30.ipynb'. The code in the first cell is as follows:

```
[2] ✓ 0s
# Create a new stack
my_stack = Stack()

# Check if the stack is empty
print(f"Is the stack empty initially? {my_stack.is_empty()}")

# Push elements onto the stack
my_stack.push(10)
my_stack.push(20)
my_stack.push(30)

# Check the size of the stack
print(f"Size of the stack after pushing elements: {my_stack.size()}")

# Peek at the top element
print(f"Top element of the stack: {my_stack.peak()}")

# Pop elements from the stack
print(f"Popped element: {my_stack.pop()}")
print(f"Popped element: {my_stack.pop()}")

# Check if the stack is empty after popping
print(f"Is the stack empty after popping? {my_stack.is_empty()}")

# Try to pop from an empty stack (this should raise an error)
try:
    my_stack.pop()
except IndexError as e:
    print(f"Caught an error when popping from an empty stack: {e}")
```

The bottom of the notebook shows the 'Variables' tab with an empty list and the 'Terminal' tab which is also empty. The status bar at the bottom indicates '3:23 PM' and 'Python 3'.

The screenshot shows the same Google Colab notebook after execution. The code cell now includes additional error handling for the peek method:

```
[2] ✓ 0s
print(f"Size of the stack after pushing elements: {my_stack.size()}")

# Peek at the top element
print(f"Top element of the stack: {my_stack.peak()}")

# Pop elements from the stack
print(f"Popped element: {my_stack.pop()}")
print(f"Popped element: {my_stack.pop()}")

# Check if the stack is empty after popping
print(f"Is the stack empty after popping? {my_stack.is_empty()}")

# Try to pop from an empty stack (this should raise an error)
try:
    my_stack.pop()
except IndexError as e:
    print(f"Caught an error when popping from an empty stack: {e}")

# Try to peek at an empty stack (this should raise an error)
try:
    my_stack.peak()
except IndexError as e:
    print(f"Caught an error when peeking at an empty stack: {e}")
```

The 'Variables' tab now shows the following output:

```
Is the stack empty initially? True
Size of the stack after pushing elements: 3
Top element of the stack: 30
Popped element: 30
Popped element: 20
Is the stack empty after popping? False
Caught an error when popping from an empty stack: pop from empty stack
```

The 'Terminal' tab is still empty. The status bar at the bottom indicates '3:23 PM' and 'Python 3'.

Task 2 :

Queue Implementation with Performance Review

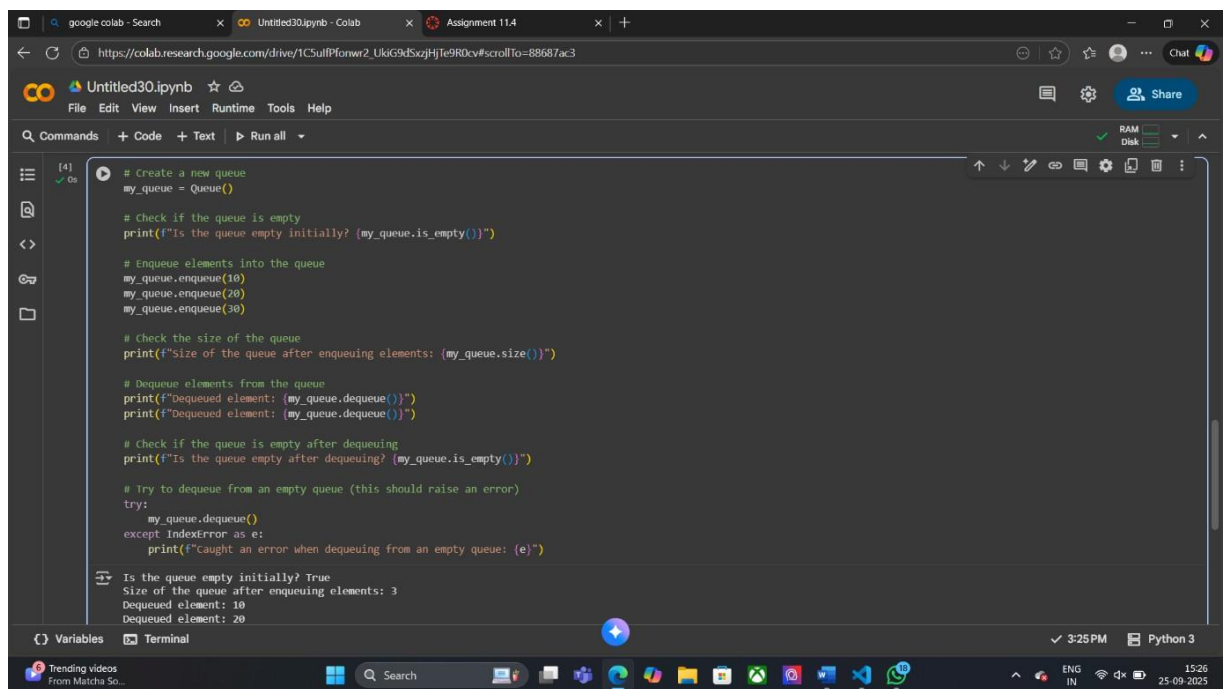
- Task: Implement a Queue with enqueue(), dequeue(), and is_empty() methods.
- Instructions:
 - o First, implement using Python lists.
 - o Then, ask AI to review performance and suggest a more efficient implementation (using collections.deque).
- Expected Output:

o Two versions of a queue: one with lists and one optimized with deque, plus an AI-generated performance comparison

Prompt:

Implement a Python Queue class with enqueue(), dequeue(), and is_empty() methods using lists.

- Ask AI to review the performance and suggest a more efficient implementation using collections.deque.
- Provide both versions and an AI-generated performance comparison.



```
[4] ✓ Os
# Create a new queue
my_queue = Queue()

# Check if the queue is empty
print(f"Is the queue empty initially? {my_queue.is_empty()}")

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

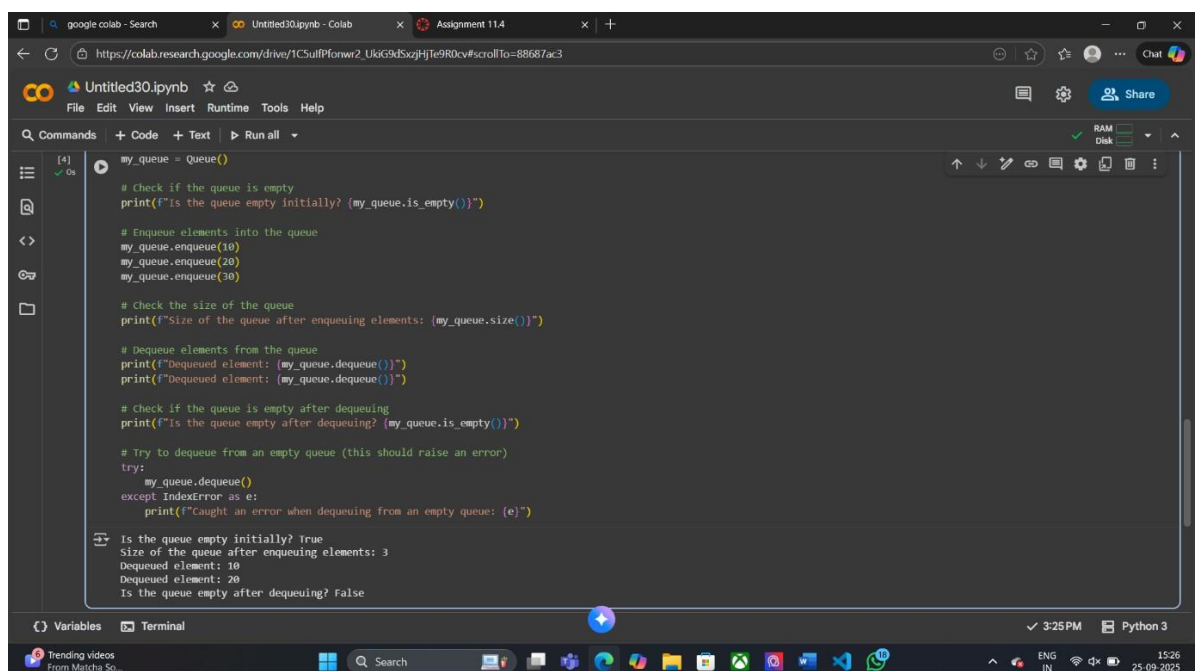
# Check the size of the queue
print(f"Size of the queue after enqueueing elements: {my_queue.size()}")

# Dequeue elements from the queue
print(f"Dequeued element: {my_queue.dequeue()}")
print(f"Dequeued element: {my_queue.dequeue()}")

# Check if the queue is empty after dequeuing
print(f"Is the queue empty after dequeuing? {my_queue.is_empty()}")

# Try to dequeue from an empty queue (this should raise an error)
try:
    my_queue.dequeue()
except IndexError as e:
    print(f"Caught an error when dequeuing from an empty queue: {e}")

Is the queue empty initially? True
Size of the queue after enqueueing elements: 3
Dequeued element: 10
Dequeued element: 20
```



```
[4] ✓ Os
# Create a new queue
my_queue = Queue()

# Check if the queue is empty
print(f"Is the queue empty initially? {my_queue.is_empty()}")

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

# Check the size of the queue
print(f"Size of the queue after enqueueing elements: {my_queue.size()}")

# Dequeue elements from the queue
print(f"Dequeued element: {my_queue.dequeue()}")
print(f"Dequeued element: {my_queue.dequeue()}")

# Check if the queue is empty after dequeuing
print(f"Is the queue empty after dequeuing? {my_queue.is_empty()}")

# Try to dequeue from an empty queue (this should raise an error)
try:
    my_queue.dequeue()
except IndexError as e:
    print(f"Caught an error when dequeuing from an empty queue: {e}")

Is the queue empty initially? True
Size of the queue after enqueueing elements: 3
Dequeued element: 10
Dequeued element: 20
Is the queue empty after dequeuing? False
```

Task 3 :

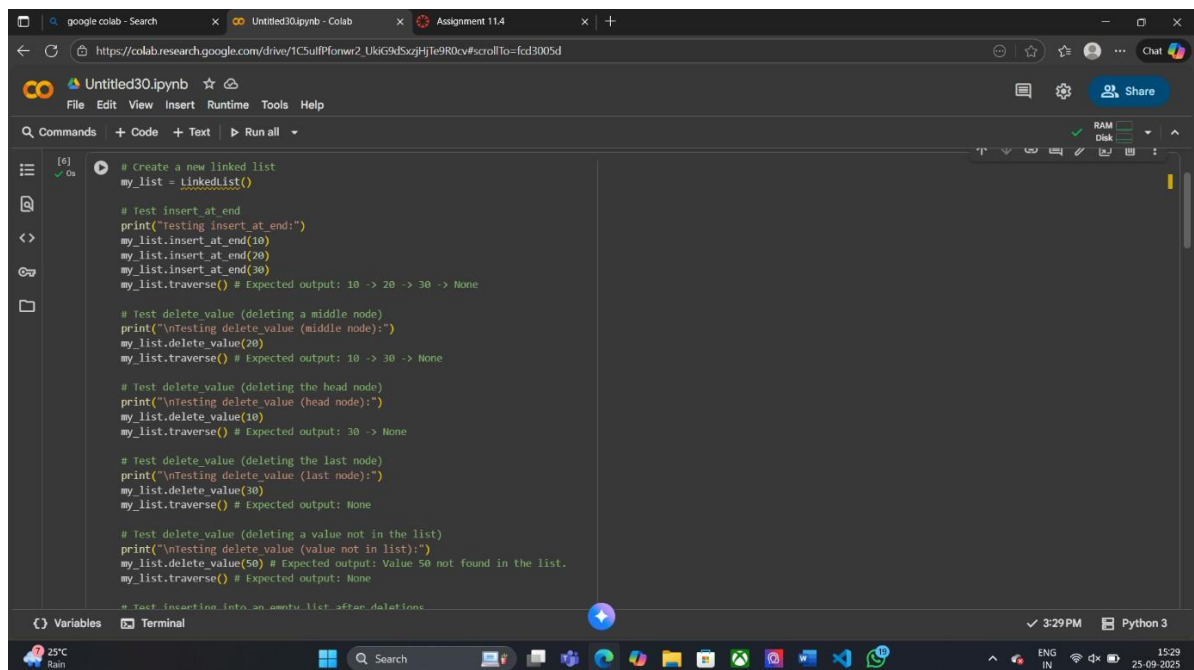
Singly Linked List with Traversal

- 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:

Implement a singly linked list in Python with `insert_at_end()`, `delete_value()`, and `traverse()` methods.

- Use AI to generate inline comments explaining pointer updates.
- Ask AI to suggest test cases to validate all operations.



```
[6] ✓ Us # Create a new linked list
my_list = LinkedList()

# Test insert at end
print("Testing insert at end:")
my_list.insert_at_end(10)
my_list.insert_at_end(20)
my_list.insert_at_end(30)
my_list.traverse() # Expected output: 10 -> 20 -> 30 -> None

# Test delete value (deleting a middle node)
print("\nTesting delete value (middle node):")
my_list.delete_value(20)
my_list.traverse() # Expected output: 10 -> 30 -> None

# Test delete value (deleting the head node)
print("\nTesting delete value (head node):")
my_list.delete_value(10)
my_list.traverse() # Expected output: 30 -> None

# Test delete value (deleting the last node)
print("\nTesting delete value (last node):")
my_list.delete_value(30)
my_list.traverse() # Expected output: None

# Test delete value (deleting a value not in the list)
print("\nTesting delete value (value not in list):")
my_list.delete_value(50) # Expected output: Value 50 not found in the list.
my_list.traverse() # Expected output: None

# Test inserting into an empty list after deletions
```

```
[6] ✓ 0s
print("\nTesting delete_value (last node):")
my_list.delete_value(30)
my_list.traverse() # Expected output: None

# Test delete_value (deleting a value not in the list)
print("\nTesting delete_value (value not in list):")
my_list.delete_value(50) # Expected output: Value 50 not found in the list.
my_list.traverse() # Expected output: None

# Test inserting into an empty list after deletions
print("\nTesting insert_at_end on an empty list:")
my_list.insert_at_end(100)
my_list.traverse() # Expected output: 100 -> None

Testing insert_at_end:
10 -> 20 -> 30None

Testing delete_value (middle node):
10 -> 30None

Testing delete_value (head node):
30None

Testing delete_value (last node):
None

Testing delete_value (value not in list):
Value 50 not found in the list.
None

Testing insert_at_end on an empty list:
100None
```

Task 4 :

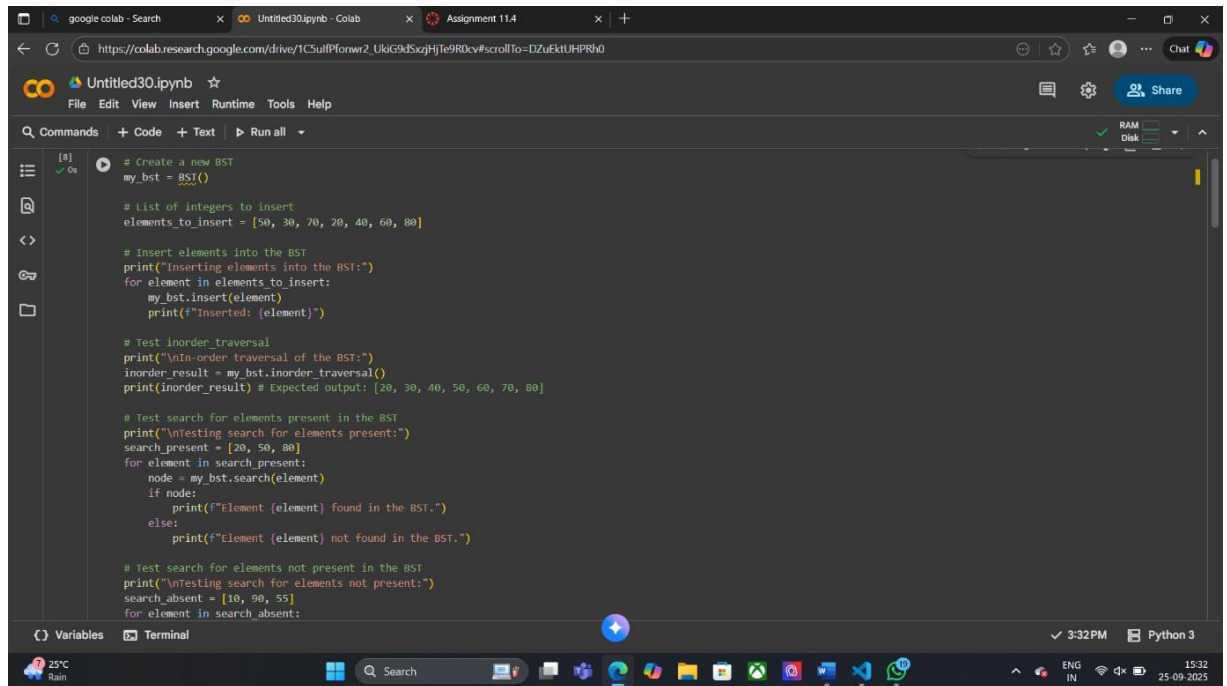
Binary Search Tree (BST)

- 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.

Prompt:

Given a partially written Node and BST class, use AI to help complete the implementation with insert(), search(), and inorder_traversal() methods.

- Ensure code is clear and well-documented.



```
[8] ✓ 0s # Create a new BST
my_bst = BST()

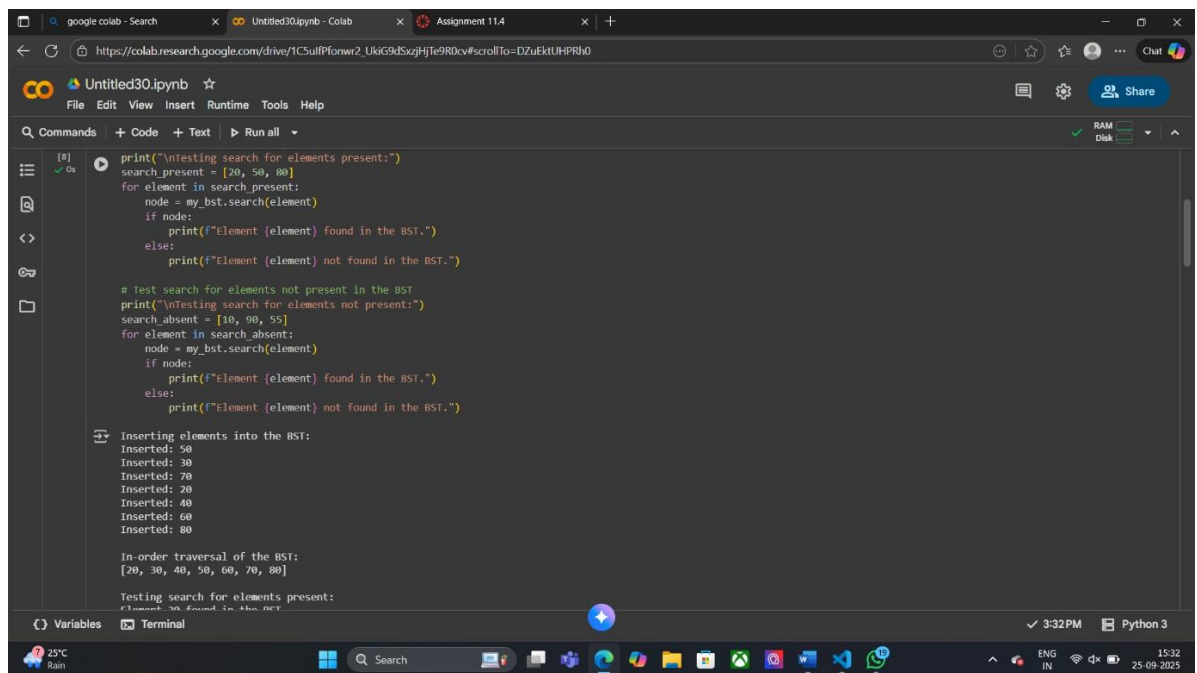
# List of integers to insert
elements_to_insert = [50, 30, 70, 20, 40, 60, 80]

# Insert elements into the BST
print("\nInserting elements into the BST:")
for element in elements_to_insert:
    my_bst.insert(element)
    print(f"Inserted: {element}")

# Test in-order traversal
print("\nIn-order traversal of the BST:")
inorder_result = my_bst.inorder_traversal()
print(inorder_result) # Expected output: [20, 30, 40, 50, 60, 70, 80]

# Test search for elements present in the BST
print("\nTesting search for elements present:")
search_present = [20, 50, 80]
for element in search_present:
    node = my_bst.search(element)
    if node:
        print(f"Element {element} found in the BST.")
    else:
        print(f"Element {element} not found in the BST.")

# Test search for elements not present in the BST
print("\nTesting search for elements not present:")
search_absent = [10, 90, 55]
for element in search_absent:
```



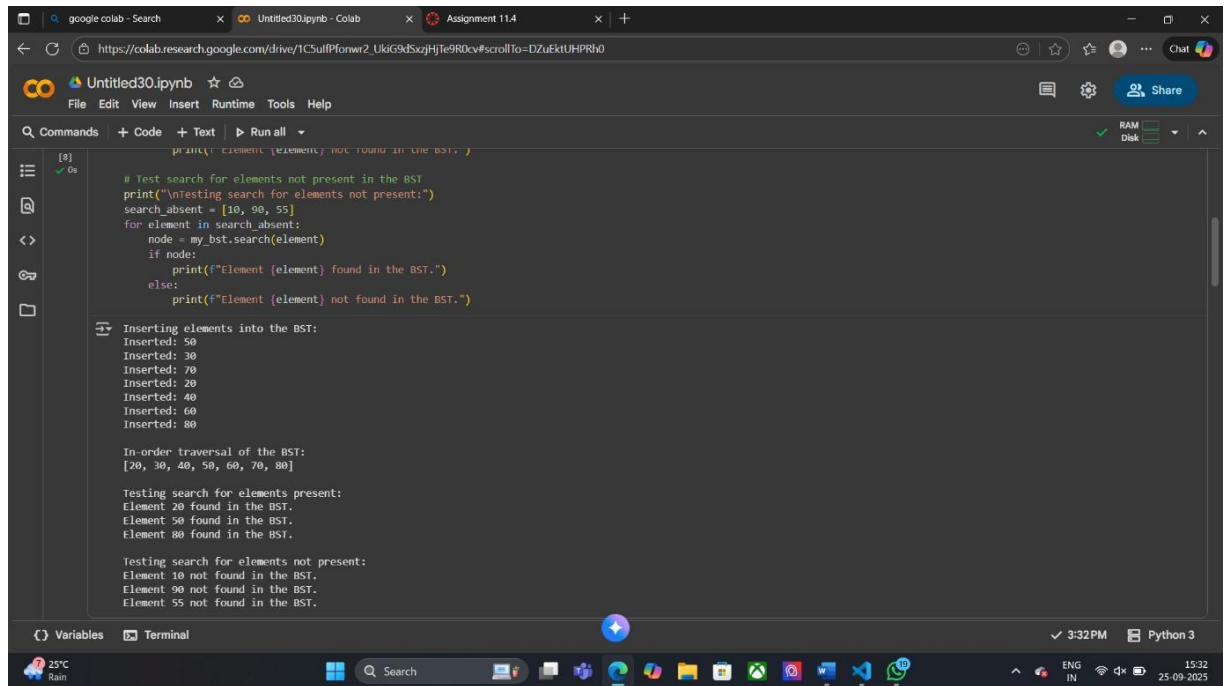
```
[8] ✓ 0s print("\nTesting search for elements present:")
search_present = [20, 50, 80]
for element in search_present:
    node = my_bst.search(element)
    if node:
        print(f"Element {element} found in the BST.")
    else:
        print(f"Element {element} not found in the BST.")

# Test search for elements not present in the BST
print("\nTesting search for elements not present:")
search_absent = [10, 90, 55]
for element in search_absent:
    node = my_bst.search(element)
    if node:
        print(f"Element {element} found in the BST.")
    else:
        print(f"Element {element} not found in the BST.")

Inserting elements into the BST:
Inserted: 50
Inserted: 30
Inserted: 70
Inserted: 20
Inserted: 40
Inserted: 60
Inserted: 80

In-order traversal of the BST:
[20, 30, 40, 50, 60, 70, 80]

Testing search for elements present:
Element 20 found in the BST.
```



```
[8] ✓ 0s

print(f"Element {element} not found in the BST.")

# Test search for elements not present in the BST
print("\nTesting search for elements not present:")
search_absent = [10, 90, 55]
for element in search_absent:
    node = my_bst.search(element)
    if node:
        print(f"Element {element} found in the BST.")
    else:
        print(f"Element {element} not found in the BST.")

Inserting elements into the BST:
Inserted: 50
Inserted: 30
Inserted: 70
Inserted: 20
Inserted: 40
Inserted: 60
Inserted: 80

In-order traversal of the BST:
[20, 30, 40, 50, 60, 70, 80]

Testing search for elements present:
Element 20 found in the BST.
Element 50 found in the BST.
Element 80 found in the BST.

Testing search for elements not present:
Element 10 not found in the BST.
Element 90 not found in the BST.
Element 55 not found in the BST.
```

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:

Implement a Python Graph class using an adjacency list (dictionary).

- Add methods for BFS(start) and DFS(start) traversal.
- Ask AI to generate both BFS and DFS implementations with inline comments explaining each traversal step.
- If suggested by AI, compare recursive and iterative DFS approaches.
- Ensure the code is clear, with comments that explain the logic of each traversal


```
[15] ✓ 0s # Create a graph instance
graph = Graph()

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

# Add edges to create a connected graph
graph.add_edge('A', 'B')
graph.add_edge('A', 'C')
graph.add_edge('B', 'D')
graph.add_edge('B', 'E')
graph.add_edge('C', 'F')
graph.add_edge('E', 'F') # Adding an edge to make it more interesting

print("BFS Traversal (starting from A):")
graph.bfs('A') # Call the bfs method

print("\nDFS Recursive Traversal (starting from A):")
graph.dfs_recursive('A') # Call the dfs_recursive method

print("\nDFS Iterative Traversal (starting from A):")
graph.dfs_iterative('A') # Call the dfs_iterative method

# Test with a non-existent node
print("\nTesting with non-existent node:")
graph.bfs('Z')
```

```
[15] ✓ 0s graph.add_edge('B', 'E')
graph.add_edge('C', 'F')
graph.add_edge('E', 'F') # Adding an edge to make it more interesting

print("BFS Traversal (starting from A):")
graph.bfs('A') # Call the bfs method

print("\nDFS Recursive Traversal (starting from A):")
graph.dfs_recursive('A') # Call the dfs_recursive method

print("\nDFS Iterative Traversal (starting from A):")
graph.dfs_iterative('A') # Call the dfs_iterative method

# Test with a non-existent node
print("\nTesting with non-existent node:")
graph.bfs('Z')
graph.dfs_recursive('Z')
graph.dfs_iterative('Z')
```

BFS Traversal (starting from A):
A B C D E F

DFS Recursive Traversal (starting from A):
A B D E F C

DFS Iterative Traversal (starting from A):
A B D E F C

Testing with non-existent node:
Node Z not found in the graph.
Node Z not found in the graph.
Node Z not found in the graph.