

# AI ASSISTED CODING LAB

## ASSIGNMENT-11

ENROLLMENT NO:2503A51L10

BATCH NO: 19

NAME: K.Praneeth

### TASK DESCRIPTION 1:

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

#### Instructions:

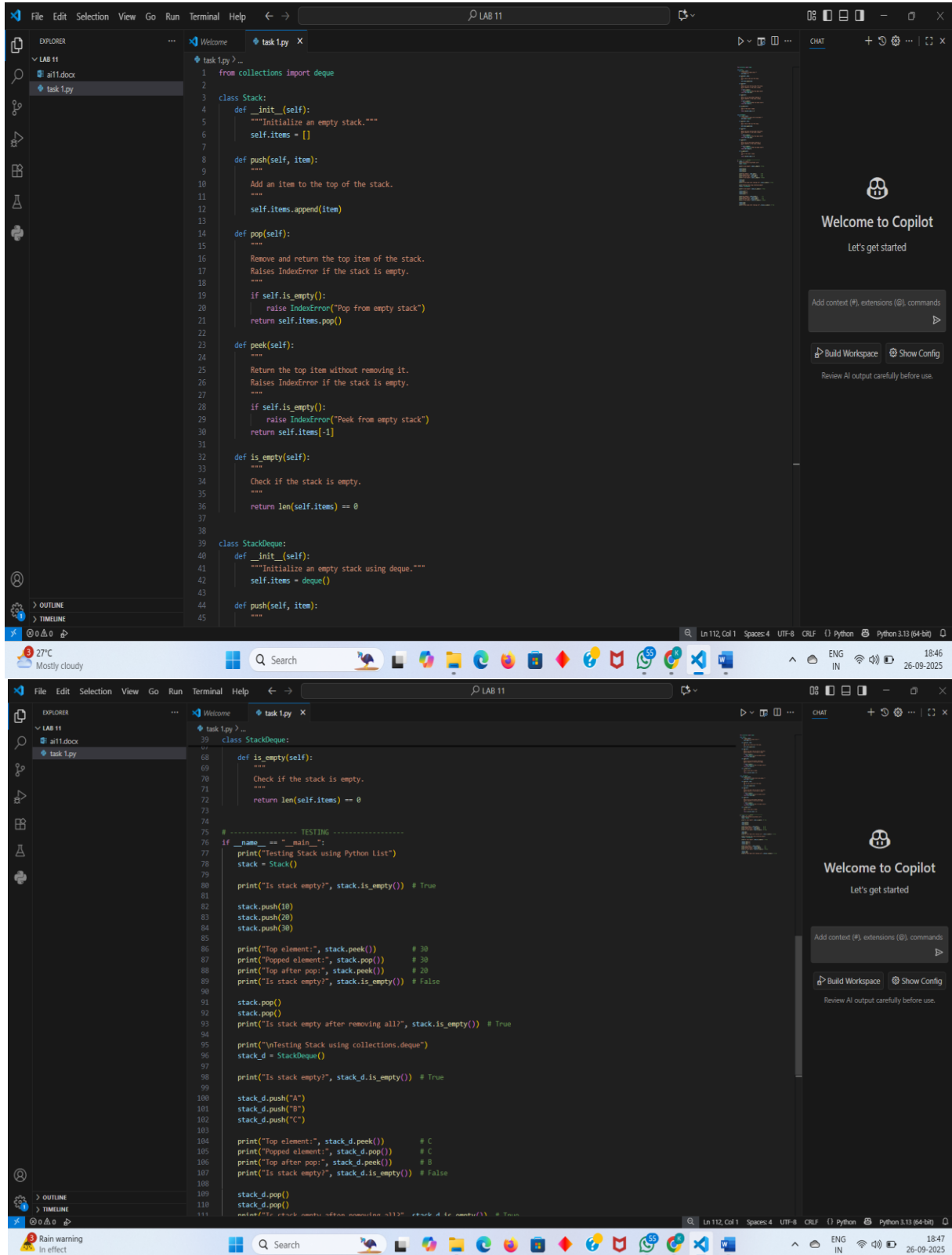
- Ask AI to generate code skeleton with docstrings.
- Test stack operations using sample data.
- Request AI to suggest optimizations or alternative implementations (e.g., using collections. Deque).

### PROMPT 1:

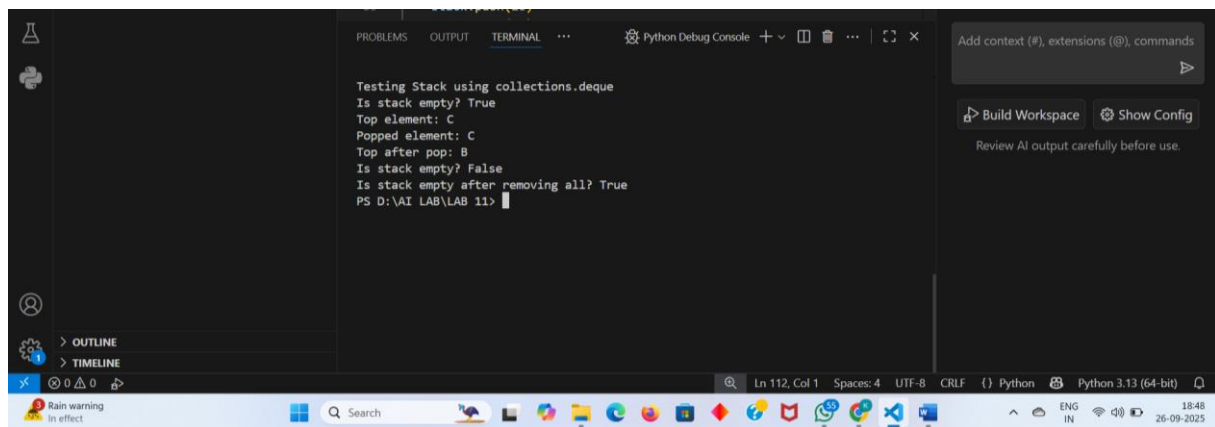
Generate a simple Stack class in Python that includes four basic operations: push () to add an item, pop () to remove the top item, peek () to view the top item without removing it, and is empty () to check if the stack has no elements. First, I'd like the AI to generate a clean code skeleton with helpful comments or docstrings explaining each method. Then, I want to test the stack using example data to make sure all the

operations work correctly. Finally, I'd like the AI to suggest better ways to implement the stack, such as using Python's collections. Deque, and explain why that might be more efficient or readable.

## CODE GENERATED:



## OUTPUT:

A screenshot of a Python IDE's terminal window. The terminal displays the following text: 'Testing Stack using collections.deque', 'Is stack empty? True', 'Top element: C', 'Popped element: C', 'Top after pop: B', 'Is stack empty? False', 'Is stack empty after removing all? True', and 'PS D:\AI LAB\LAB 11>'. The IDE interface includes a sidebar with 'OUTLINE' and 'TIMELINE' tabs, a top bar with 'PROBLEMS', 'OUTPUT', and 'TERMINAL' tabs, and a bottom status bar showing 'Ln 112, Col 1', 'Spaces: 4', 'UTF-8', 'CRLF', 'Python', and 'Python 3.13 (64-bit)'. The Windows taskbar is visible at the bottom with various application icons and a system clock showing 18:48 on 26-09-2023.

```
Testing Stack using collections.deque
Is stack empty? True
Top element: C
Popped element: C
Top after pop: B
Is stack empty? False
Is stack empty after removing all? True
PS D:\AI LAB\LAB 11>
```

## OBSERVATION:

This assignment effectively demonstrates the use of AI to support structured programming tasks. By implementing a Stack class with core operations—push (), pop (), peek (), and is empty ()—the task reinforces foundational concepts in data structures. The prompt encourages clean code practices through the use of docstrings and method documentation, while also promoting testing with sample data to validate functionality. Additionally, the request for AI-driven suggestions on alternative implementations, such as using collections. Deque, introduces learners to performance considerations and Pythonic design choices. Overall, the assignment balances technical execution with reflective learning, making it a strong

exercise in both coding and code evaluation.

## TASK DESCRIPTION 2:

Implement a **Queue** with `enqueue ()`, `dequeue ()`, and `is empty ()` methods.

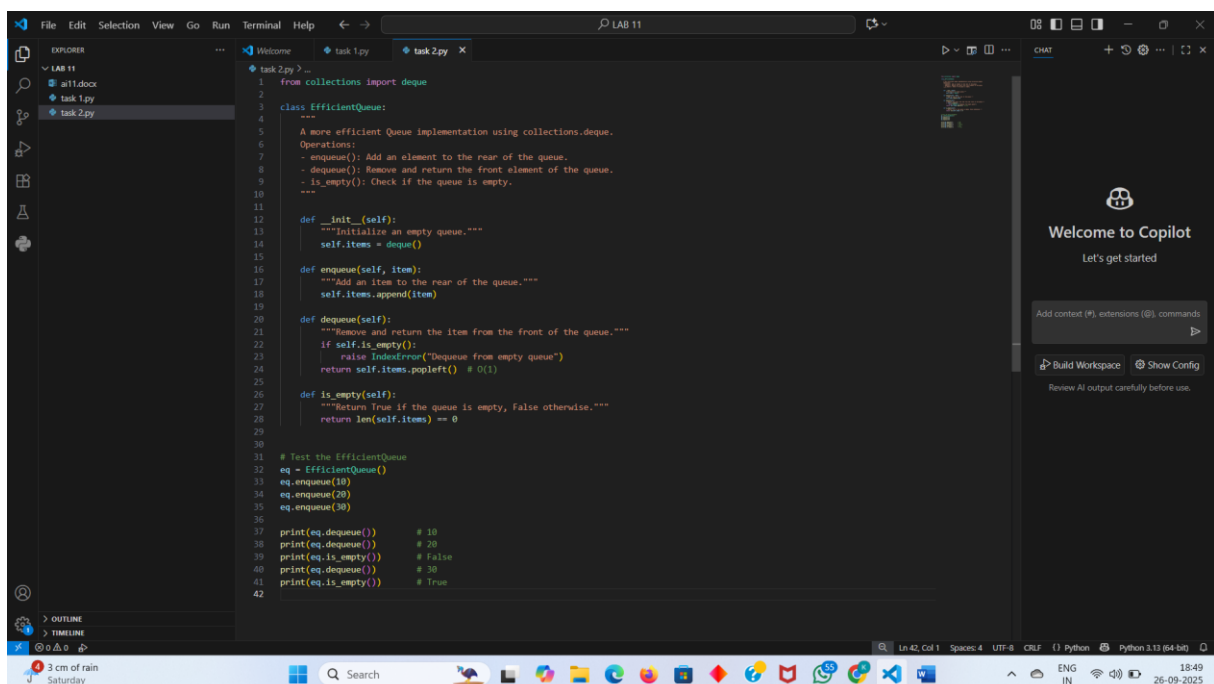
- **Instructions:**

- First, implement using Python lists.
- Then, ask AI to review performance and suggest a more efficient implementation (using collections. Deque).

## PROMPT 1:

Generate a basic Queue class in Python with three functions: `enqueue ()` to add an item, `dequeue ()` to remove the first item, and `is empty ()` to check if the queue is empty. First, I'll use a regular Python list to build it. Then, I'd like the AI to look at how well this version performs and suggest a better way to write it—like using collections. Deque—and explain why that might be faster or more efficient

## CODE GENERATED:



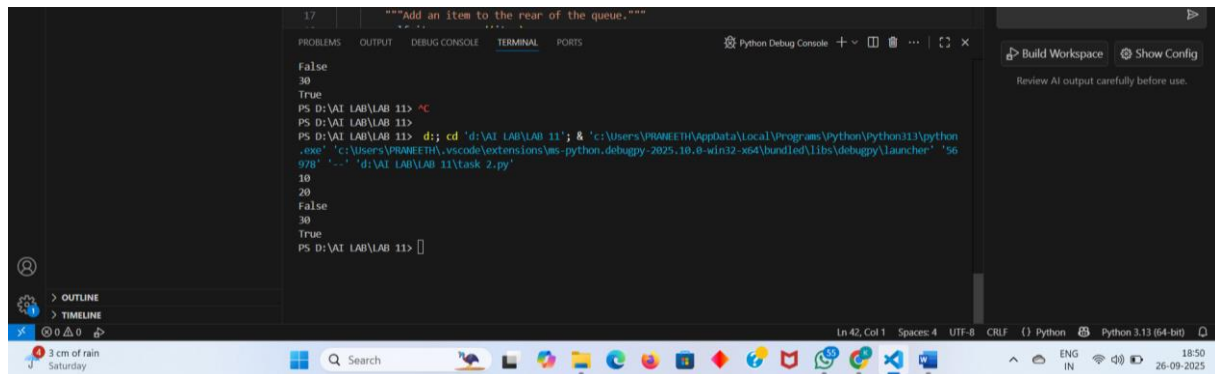
The screenshot shows a VS Code editor window with a file explorer on the left, a terminal at the bottom, and a chat window on the right. The main editor displays a Python file named `task2.py` containing the following code:

```
1 from collections import deque
2
3 class EfficientQueue:
4     """
5     A more efficient Queue implementation using collections.deque.
6     Operations:
7     - enqueue(): Add an element to the rear of the queue.
8     - dequeue(): Remove and return the front element of the queue.
9     - is_empty(): Check if the queue is empty.
10    """
11
12    def __init__(self):
13        """Initialize an empty queue."""
14        self.items = deque()
15
16    def enqueue(self, item):
17        """Add an item to the rear of the queue."""
18        self.items.append(item)
19
20    def dequeue(self):
21        """Remove and return the item from the front of the queue."""
22        if self.is_empty():
23            raise IndexError("Dequeue from empty queue")
24        return self.items.popleft() # O(1)
25
26    def is_empty(self):
27        """Return True if the queue is empty, False otherwise."""
28        return len(self.items) == 0
29
30
31 # Test the EfficientQueue
32 eq = EfficientQueue()
33 eq.enqueue(10)
34 eq.enqueue(20)
35 eq.enqueue(30)
36
37 print(eq.dequeue()) # 10
38 print(eq.dequeue()) # 20
39 print(eq.is_empty()) # False
40 print(eq.dequeue()) # 30
41 print(eq.is_empty()) # True
42
```

The chat window on the right displays the "Welcome to Copilot" message and a "Let's get started" button. The terminal at the bottom shows the output of the code execution:

```
Ln 42, Col 1: Spaces: 4 UTF-8 CRLF Python Python 3.12 (64-bit)
```

## OUTPUT:



```
17 """Add an item to the rear of the queue."""
....
False
30
True
PS D:\VAI LAB\LAB 11> ^C
PS D:\VAI LAB\LAB 11>
PS D:\VAI LAB\LAB 11> d: cd 'd:\VAI LAB\LAB 11'; & 'c:\Users\PRANEETH\AppData\Local\Programs\Python\Python313\python
.exe' 'c:\Users\PRANEETH\vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundle\libs\debugpy\launcher' '56
978' '-' 'd:\VAI LAB\LAB 11\task 2.py'
10
20
False
30
True
PS D:\VAI LAB\LAB 11> 
```

## OBSERVATION:

This assignment provides a practical introduction to queue data structures and highlights the importance of performance-aware coding. By first implementing a Queue class using Python lists, it reinforces the basic logic behind enqueueing and dequeuing operations. However, it also encourages deeper thinking by asking for a performance review, which reveals the limitations of list-based queues—particularly the inefficiency of removing items from the front. The follow-up suggestion to use collections. Deque introduces learners to a more optimized and Pythonic solution, demonstrating how built-in modules can improve both speed and memory usage. Overall, the task blends hands-on coding with thoughtful evaluation, making it a strong exercise in both implementation and improvement.

## TASK DESCRIPTION 3:

Implement a **Singly Linked List** with operations: instated (), delete value (), and traverse ().

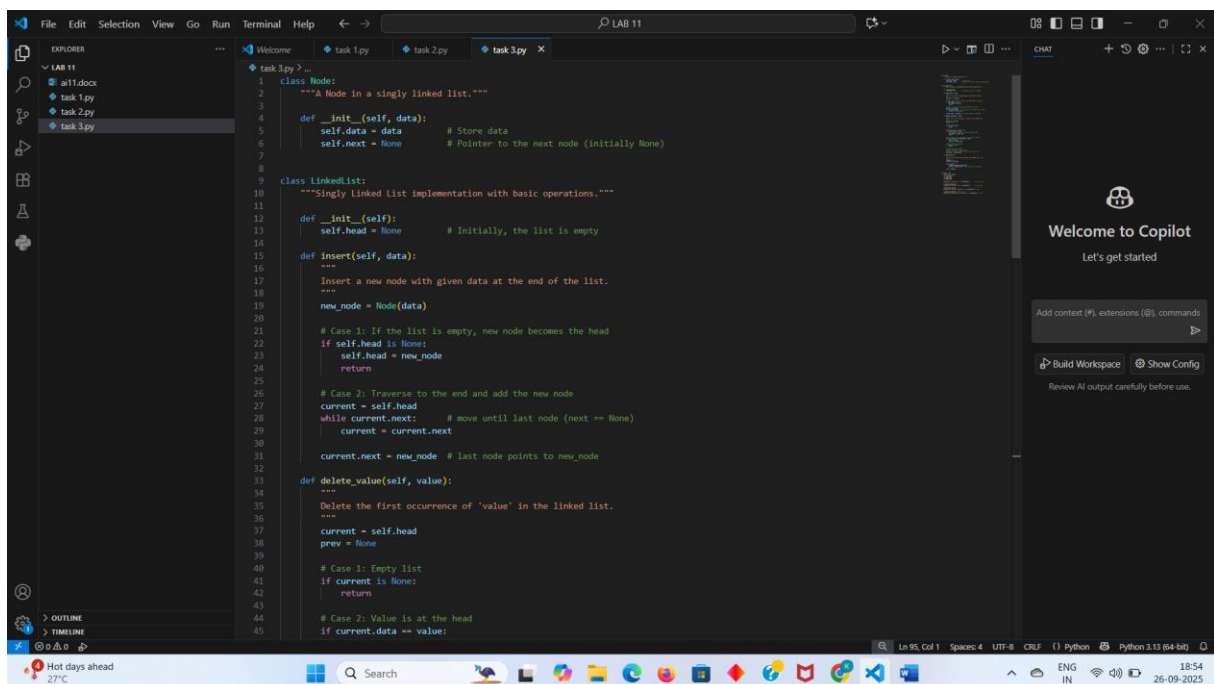
- **Instructions:**
  - Start with a simple class-based implementation (Node, LinkedList).
  - Use AI to generate inline comments explaining pointer updates (which are non-trivial).

- Ask AI to suggest test cases to validate all operations.

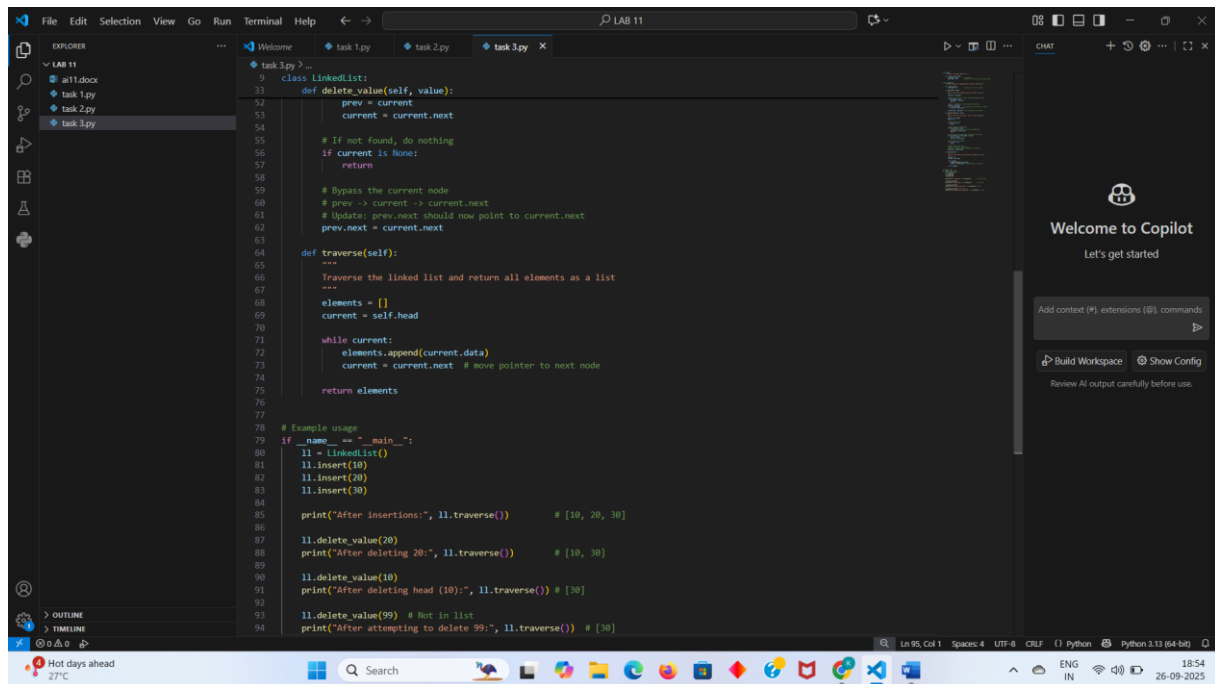
## PROMPT 1:

Generate a Singly Linked List in Python with three main operations: `inserted ()` to add a node at the end, `delete value ()` to remove a node with a specific value, and `traverse ()` to print or return all the elements in order. I'll start with a basic class-based setup using a `Node` class and a `LinkedList` class. I'd like the AI to help write inline comments that explain how pointers are updated during each operation, since that part can be tricky. Finally, I want the AI to suggest test cases to check if all the functions work correctly, including edge cases like deleting from an empty list or removing the last node.

## CODE GENERATED:

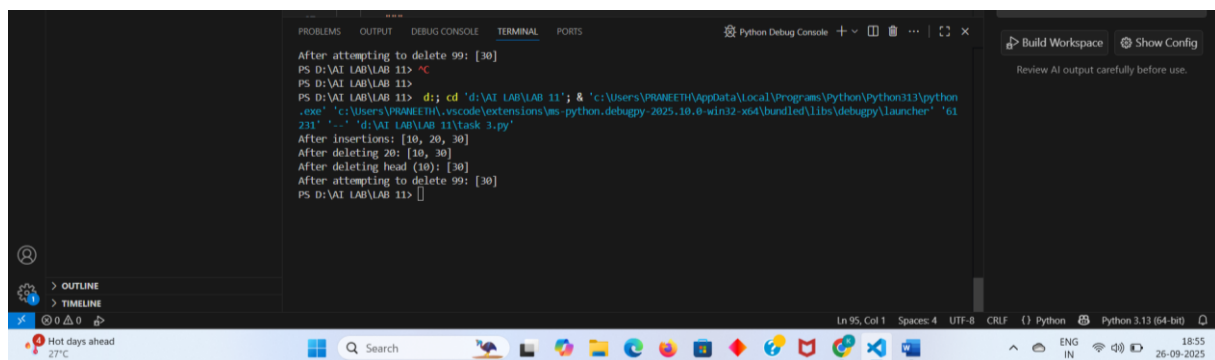


```
1 class Node:
2     """A Node in a singly linked list."""
3
4     def __init__(self, data):
5         self.data = data        # Store data
6         self.next = None       # Pointer to the next node (initially None)
7
8
9 class LinkedList:
10     """Singly Linked List implementation with basic operations."""
11
12     def __init__(self):
13         self.head = None       # Initially, the list is empty
14
15     def insert(self, data):
16         """
17         Insert a new node with given data at the end of the list.
18         """
19         new_node = Node(data)
20
21         # Case 1: If the list is empty, new node becomes the head
22         if self.head is None:
23             self.head = new_node
24             return
25
26         # Case 2: Traverse to the end and add the new node
27         current = self.head
28         while current.next:      # move until last node (next == None)
29             current = current.next
30
31         current.next = new_node # last node points to new_node
32
33     def delete_value(self, value):
34         """
35         Delete the first occurrence of 'value' in the linked list.
36         """
37         current = self.head
38         prev = None
39
40         # Case 1: Empty list
41         if current is None:
42             return
43
44         # Case 2: Value is at the head
45         if current.data == value:
```



```
1 class LinkedList:
2     def __init__(self):
3         self.head = None
4
5     def delete_value(self, value):
6         # If not found, do nothing
7         # If current is None:
8         #     return
9         # Bypass the current node
10        # prev -> current -> current.next
11        # Update: prev.next should now point to current.next
12        prev = current
13        current = current.next
14
15        # If not found, do nothing
16        if current is None:
17            return
18
19        # Bypass the current node
20        # prev -> current -> current.next
21        # Update: prev.next should now point to current.next
22        prev.next = current.next
23
24    def traverse(self):
25        """
26        Traverse the linked list and return all elements as a list
27        """
28        elements = []
29        current = self.head
30
31        while current:
32            elements.append(current.data)
33            current = current.next # move pointer to next node
34
35        return elements
36
37    # Example usage
38    if __name__ == "__main__":
39        ll = LinkedList()
40        ll.insert(10)
41        ll.insert(20)
42        ll.insert(30)
43
44        print("After insertions:", ll.traverse()) # [10, 20, 30]
45
46        ll.delete_value(20)
47        print("After deleting 20:", ll.traverse()) # [10, 30]
48
49        ll.delete_value(10)
50        print("After deleting head (10):", ll.traverse()) # [30]
51
52        ll.delete_value(99) # Not in list
53        print("After attempting to delete 99:", ll.traverse()) # [30]
```

## OUTPUT:



```
After attempting to delete 99: [30]
PS D:\VAI LAB\LAB 11> ^C
PS D:\VAI LAB\LAB 11> d: cd 'd:\VAI LAB\LAB 11'; & 'C:\Users\PRANEETH\AppData\Local\Programs\Python\Python313\python.exe' 'c:\Users\PRANEETH\vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bin\debugpy_launcher' '61231' '...' 'd:\VAI LAB\LAB 11\task 3.py'
After insertions: [10, 20, 30]
After deleting 20: [10, 30]
After deleting head (10): [30]
After attempting to delete 99: [30]
PS D:\VAI LAB\LAB 11> ]
```

## OBSERVATION:

This assignment offers a hands-on approach to understanding how singly linked lists work, especially the role of pointers in dynamic data structures. By implementing operations like instated (), delete value (), and traverse (), it reinforces the concept of node manipulation and sequential access. The use of a class-based design with Node and LinkedList classes encourages clean organization and object-oriented thinking. Asking AI to generate inline comments for pointer updates adds clarity to a commonly misunderstood part of linked list logic, making the code more readable and educational. Additionally, requesting test

cases—especially for edge conditions like deleting from an empty list or removing the last node—promotes thorough validation and defensive programming. Overall, the assignment blends implementation, explanation, and testing in a way that deepens both conceptual understanding and practical coding skills.

## TASK DESCRIPTION 4:

- 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 1:

Generate binary Search Tree in Python and have already started writing the `Node` and `BST` classes. I want help from AI to complete the missing methods: `insert ()` to add values, `search ()` to find a value, and `inorder_traversal ()` to return the sorted order of elements. Please add docstrings to explain what each method does. After the code is complete, I'd like to test it using a list of integers and compare the results of `search ()` for values that are in the tree versus values that are not.

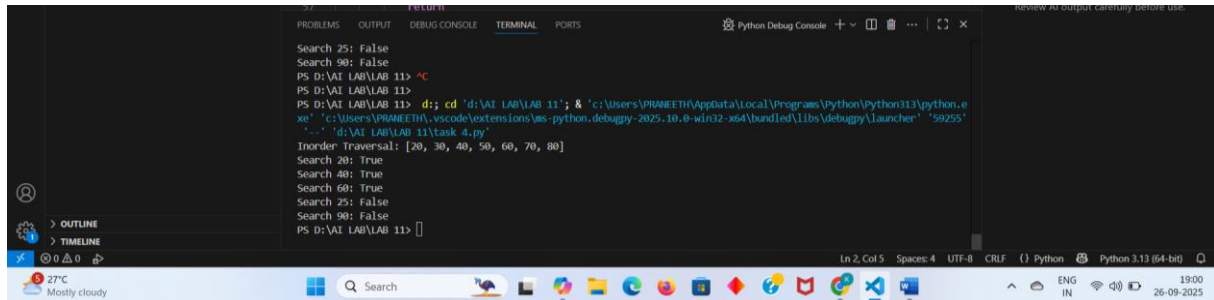
## CODE GENERATED:



```
1 class Node:
2
3
4 def __init__(self, key):
5     self.key = key
6     self.left = None
7     self.right = None
8
9 class BinarySearchTree:
10
11
12 def __init__(self):
13     self.root = None
14
15 def insert(self, key):
16     """Insert a new key into the BST."""
17     if self.root is None:
18         self.root = Node(key)
19     else:
20         self._insert_recursive(self.root, key)
21
22 def _insert_recursive(self, current, key):
23     if key < current.key:
24         if current.left is None:
25             current.left = Node(key)
26         else:
27             self._insert_recursive(current.left, key)
28     elif key > current.key:
29         if current.right is None:
30             current.right = Node(key)
31         else:
32             self._insert_recursive(current.right, key)
33     # If key == current.key, do nothing (no duplicates allowed)
34
35 def search(self, key):
36     """Return True if key is in the BST, else False."""
37     return self._search_recursive(self.root, key)
38
39 def _search_recursive(self, current, key):
40     if current is None:
41         return False
42     elif key == current.key:
43         return True
44     elif key < current.key:
45         return self._search_recursive(current.left, key)
```

```
38
39 def _search_recursive(self, current, key):
40     if current is None:
41         return False
42     elif key == current.key:
43         return True
44     elif key < current.key:
45         return self._search_recursive(current.left, key)
46     else:
47         return self._search_recursive(current.right, key)
48
49 def inorder(self):
50     """Return the Inorder traversal (sorted order) of the BST as a list."""
51     result = []
52     self._inorder_recursive(self.root, result)
53     return result
54
55 def _inorder_recursive(self, current, result):
56     if current is None:
57         return
58     self._inorder_recursive(current.left, result)
59     result.append(current.key)
60     self._inorder_recursive(current.right, result)
61
62 # ----- TESTING -----
63 if __name__ == "__main__":
64     bst = BinarySearchTree()
65     numbers = [50, 30, 70, 20, 40, 60, 80]
66
67     # Insert numbers into BST
68     for num in numbers:
69         bst.insert(num)
70
71     # Test inorder traversal (should be sorted list)
72     print("Inorder Traversal:", bst.inorder())
73
74     # Test search for present elements
75     print("Search 20:", bst.search(20)) # Expected: True
76     print("Search 40:", bst.search(40)) # Expected: True
77     print("Search 60:", bst.search(60)) # Expected: True
78
79     # Test search for absent elements
80     print("Search 25:", bst.search(25)) # Expected: False
81     print("Search 90:", bst.search(90)) # Expected: False
```

## OUTPUT:



```
Search 25: False
Search 90: False
PS D:\VAI LAB\LAB 11> ^C
PS D:\VAI LAB\LAB 11> d; cd 'd:\VAI LAB\LAB 11'; & 'c:\Users\PRAMEETH\AppData\Local\Programs\Python\Python313\python.exe' 'c:\Users\PRAMEETH\vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundle\libs\debugpy\launcher' '59255' '...' 'd:\VAI LAB\LAB 11\task 4.py'
Inorder Traversal: [20, 30, 40, 50, 60, 70, 80]
Search 20: True
Search 40: True
Search 60: True
Search 25: False
Search 90: False
PS D:\VAI LAB\LAB 11> []
```

## OBSERVATION:

This assignment offers a solid introduction to recursive data structures and algorithmic thinking through the implementation of a Binary Search Tree (BST). By starting with a partially written Node and BST class, it encourages learners to focus on completing core methods—insert (), search (), and inorder\_traversal ()—which are fundamental to understanding tree-based logic. The use of AI to fill in missing code and generate docstrings promotes clarity and reinforces best practices in documentation. Testing the tree with a list of integers and comparing search results for present versus absent values adds a layer of validation and encourages attention to edge cases. Overall, the assignment blends hands-on coding, conceptual understanding, and reflective testing, making it a well-rounded exercise in both implementation and evaluation.

## TASK DESCRIPTION 5:

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

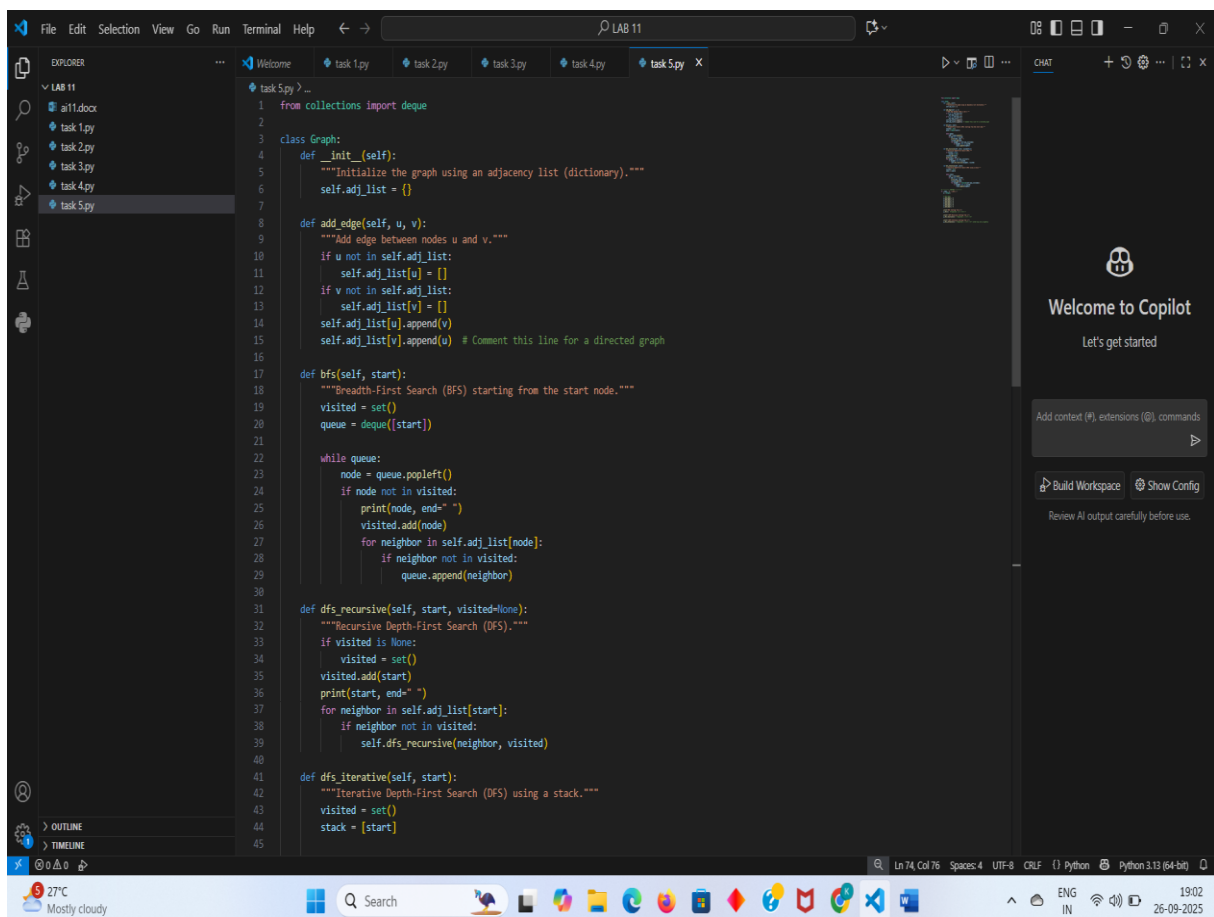
- **Instructions:**
  - Start with an adjacency list dictionary.
  - Ask AI to generate BFS and DFS implementations with inline comments.

- Compare recursive vs iterative DFS if suggested by AI.

## PROMPT 1:

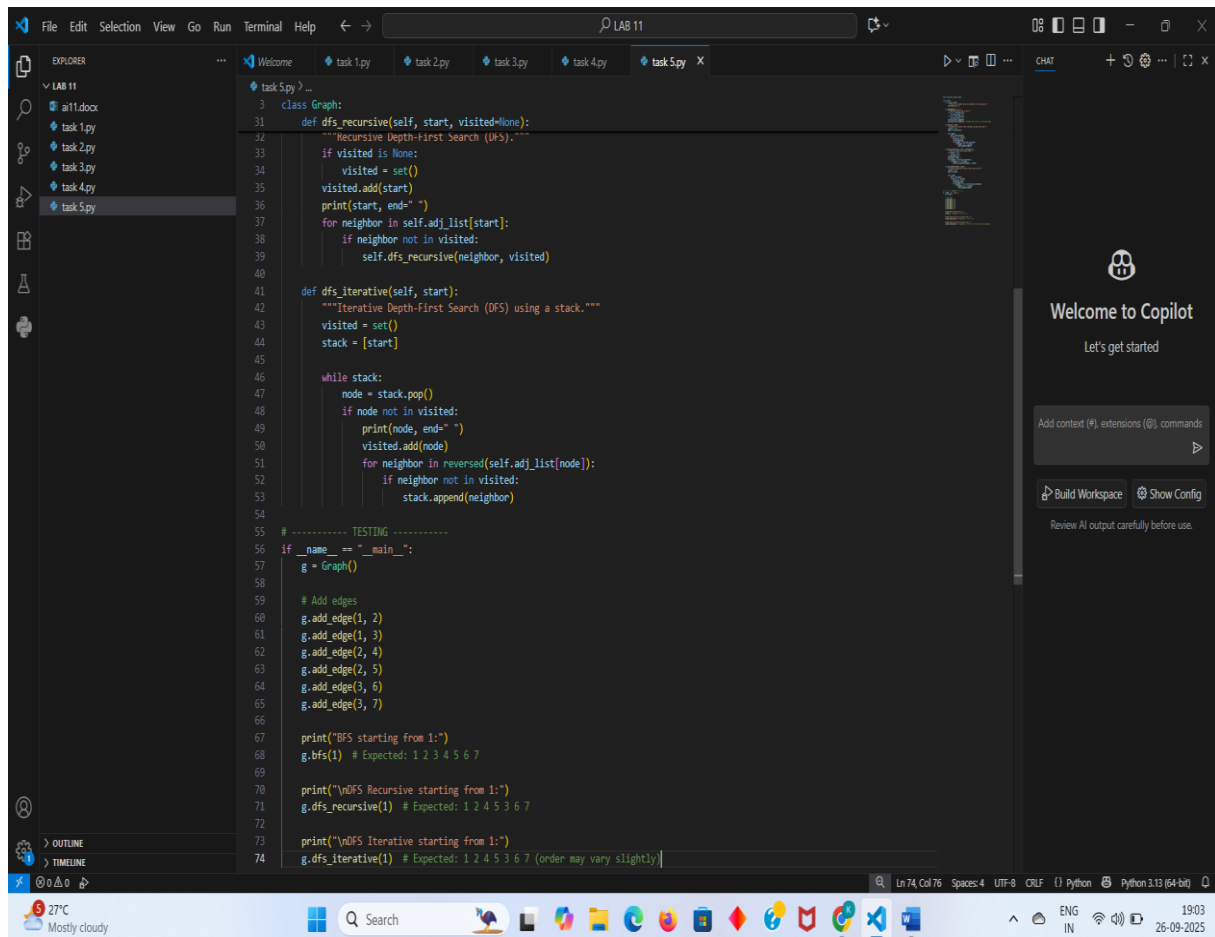
Generate a Graph in Python using an adjacency list represented as a dictionary. I need help from AI to write the traversal methods: BFS () for breadth-first search and DFS () for depth-first search. Please include inline comments that explain how each part of the traversal works, especially how nodes are visited and tracked. If possible, I'd also like to compare recursive and iterative versions of DFS, and understand which one is better in terms of readability or performance.

## CODE GENERATED:



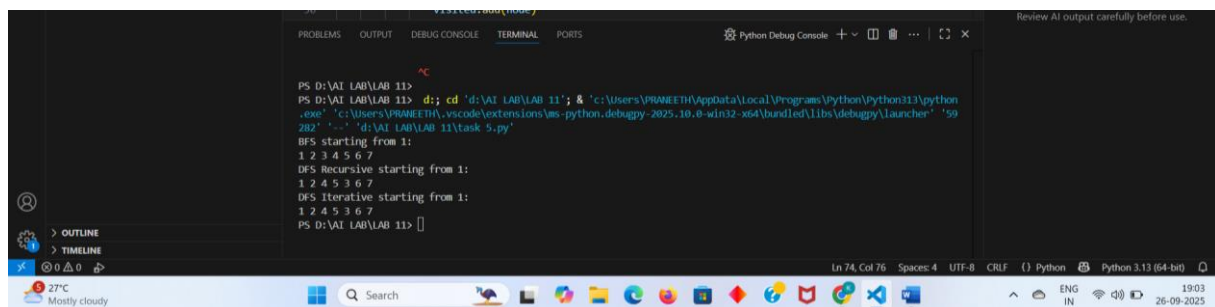
```
1 from collections import deque
2
3 class Graph:
4     def __init__(self):
5         """Initialize the graph using an adjacency list (dictionary)."""
6         self.adj_list = {}
7
8     def add_edge(self, u, v):
9         """Add edge between nodes u and v."""
10        if u not in self.adj_list:
11            self.adj_list[u] = []
12        if v not in self.adj_list:
13            self.adj_list[v] = []
14        self.adj_list[u].append(v)
15        self.adj_list[v].append(u) # Comment this line for a directed graph
16
17    def bfs(self, start):
18        """Breadth-First Search (BFS) starting from the start node."""
19        visited = set()
20        queue = deque([start])
21
22        while queue:
23            node = queue.popleft()
24            if node not in visited:
25                print(node, end=" ")
26                visited.add(node)
27                for neighbor in self.adj_list[node]:
28                    if neighbor not in visited:
29                        queue.append(neighbor)
30
31    def dfs_recursive(self, start, visited=None):
32        """Recursive Depth-First Search (DFS)."""
33        if visited is None:
34            visited = set()
35        visited.add(start)
36        print(start, end=" ")
37        for neighbor in self.adj_list[start]:
38            if neighbor not in visited:
39                self.dfs_recursive(neighbor, visited)
40
41    def dfs_iterative(self, start):
42        """Iterative Depth-First Search (DFS) using a stack."""
43        visited = set()
44        stack = [start]
```

The screenshot shows a Visual Studio Code editor window with a Python file named 'task 5.py'. The code defines a 'Graph' class with an adjacency list. It includes methods for adding edges, breadth-first search (BFS), and two versions of depth-first search (DFS): recursive and iterative. The recursive DFS method uses a helper function to track visited nodes, while the iterative DFS method uses a stack. The editor interface includes a file explorer on the left, a terminal at the bottom, and a Copilot chat window on the right. The system tray at the bottom shows the date and time as 19:02 on 26-09-2025.



```
3 class Graph:
31 def dfs_recursive(self, start, visited=None):
32     """Recursive Depth-First Search (DFS)"""
33     if visited is None:
34         visited = set()
35     visited.add(start)
36     print(start, end=" ")
37     for neighbor in self.adj_list[start]:
38         if neighbor not in visited:
39             self.dfs_recursive(neighbor, visited)
40
41 def dfs_iterative(self, start):
42     """Iterative Depth-First Search (DFS) using a stack"""
43     visited = set()
44     stack = [start]
45
46     while stack:
47         node = stack.pop()
48         if node not in visited:
49             print(node, end=" ")
50             visited.add(node)
51             for neighbor in reversed(self.adj_list[node]):
52                 if neighbor not in visited:
53                     stack.append(neighbor)
54
55 # ----- TESTING -----
56 if __name__ == "__main__":
57     g = Graph()
58
59     # Add edges
60     g.add_edge(1, 2)
61     g.add_edge(1, 3)
62     g.add_edge(2, 4)
63     g.add_edge(2, 5)
64     g.add_edge(3, 6)
65     g.add_edge(3, 7)
66
67     print("BFS starting from 1:")
68     g.bfs(1) # Expected: 1 2 3 4 5 6 7
69
70     print("\nDFS Recursive starting from 1:")
71     g.dfs_recursive(1) # Expected: 1 2 4 5 3 6 7
72
73     print("\nDFS Iterative starting from 1:")
74     g.dfs_iterative(1) # Expected: 1 2 4 5 3 6 7 (order may vary slightly)
```

## OUTPUT:



```
PS D:\VAI LAB\LAB 11>
PS D:\VAI LAB\LAB 11> cd 'd:\VAI LAB\LAB 11'; & 'c:\Users\PRANEETH\AppData\Local\Programs\Python\Python313\python.exe' 'c:\Users\PRANEETH\vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\libs\debugpy\launcher' '59282' '-.' 'd:\VAI LAB\LAB 11\task 5.py'
BFS starting from 1:
1 2 3 4 5 6 7
DFS Recursive starting from 1:
1 2 4 5 3 6 7
DFS Iterative starting from 1:
1 2 4 5 3 6 7
PS D:\VAI LAB\LAB 11>
```

## OBSERVATION:

This assignment provides a strong foundation for understanding graph traversal techniques using Python. By implementing a graph with an adjacency list, learners engage with a widely used and memory-efficient representation of graph structures. The inclusion of both BFS (Breadth-First Search) and DFS (Depth-First Search) encourages exploration of different traversal strategies, each with distinct use cases and behaviours. Requesting inline comments for each step of the traversal helps clarify

how nodes are visited, queued, or stacked—making the logic more transparent and easier to follow. The comparison between recursive and iterative DFS adds a valuable layer of analysis, prompting reflection on performance trade-offs and code readability. Overall, this assignment blends implementation, explanation, and evaluation, making it a well-rounded exercise in algorithmic thinking and practical coding.