AI ASSIGNMENT - 11

ENROLLMENT NO:2503A51L16

BATCH NO: 19

NAME: K. JASHUVA

TASK 1

TASK DESCRIPTION:

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:

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:

L1T1.PY ×

L1T1.PY

```python
from collections import deque
class Stack:
   def __init__(self):
    """Initialize an empty stack."""
    self.items = []
  def push(self, item):
    """
    Add an item to the top of the stack.
    """
    self.items.append(item)

  def pop(self):
    """
    Remove and return the top item of the stack.
    Raises IndexError if the stack is empty.
    """
    if self.is_empty():
      raise IndexError("Pop from empty stack")
    return self.items.pop()

  def peek(self):
    """
    Return the top item without removing it.
    Raises IndexError if the stack is empty.
    """
    if self.is_empty():
      raise IndexError("Peek from empty stack")
    return self.items[-1]

  def is_empty(self):
    """
    Check if the stack is empty.
    """
    return len(self.items) == 0


class StackDeque:
```

L1T1.PY ×

L1T1.PY

```python
# ---------------- TESTING ----------------
if __name__ == "__main__":
    print("Testing Stack using Python List")
    stack = Stack()

    print("Is stack empty?", stack.is_empty())  # True

    stack.push(10)
    stack.push(20)
    stack.push(30)

    print("Top element:", stack.peek())  # 30
    print("Popped element:", stack.pop())  # 30
    print("Top after pop:", stack.peek())  # 20
    print("Is stack empty?", stack.is_empty())  # False

    stack.pop()
    stack.pop()
    print("Is stack empty after removing all?", stack.is_empty())  # True

    print("\nTesting Stack using collections.deque")
    stack_d = StackDeque()

    print("Is stack empty?", stack_d.is_empty())  # True

    stack_d.push("A")
    stack_d.push("B")
    stack_d.push("C")

    print("Top element:", stack_d.peek())  # C
    print("Popped element:", stack_d.pop())  # C
    print("Top after pop:", stack_d.peek())  # B
    print("Is stack empty?", stack_d.is_empty())  # False

    stack_d.pop()
    stack_d.pop()
    print("Is stack empty after removing all?", stack_d.is_empty())  # True
```

OUTPUT:

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

```
[Running] python -u "c:\Users\Charan\OneDrive\Desktop\python\L1T1.PY"
Testing Stack using Python List
Is stack empty? True
Top element: 30
Popped element: 30
Top after pop: 20
Is stack empty? False
Is stack empty after removing all? True

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
```

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 2

TASK DESCRIPTION:

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:

L1T1.PY ×

L1T1.PY

```python
from collections import deque
class EfficientQueue:
    """
    A more efficient Queue implementation using collections.deque.
    Operations:
    - enqueue(): Add an element to the rear of the queue.
    - dequeue(): Remove and return the front element of the queue.
    - is_empty(): Check if the queue is empty.
    """
    def __init__(self):
        """Initialize an empty queue."""
        self.items = deque()
    def enqueue(self, item):
        """Add an item to the rear of the queue."""
        self.items.append(item)
    def dequeue(self):
        """Remove and return the item from the front of the queue."""
        if self.is_empty():
            raise IndexError("Dequeue from empty queue")
        return self.items.popleft()  # O(1)
    def is_empty(self):
        """Return True if the queue is empty, False otherwise."""
        return len(self.items) == 0
# Test the EfficientQueue
eq = EfficientQueue()
eq.enqueue(10)
eq.enqueue(20)
eq.enqueue(30)
print(eq.dequeue())  # 10
print(eq.dequeue())  # 20
print(eq.is_empty()) # False
print(eq.dequeue())  # 30
print(eq.is_empty()) # True
```

OUTPUT:

```
[Running] python -u "c:\Users\Charan\OneDrive\Desktop\python\L1T1.PY"
10
20
False
30
True
```

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 3

TASK DESCRIPTION :

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 :

Generate a Singly Linked List in Python with three main operations: instated () 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:

L1T1.PY ×

L1T1.PY

```python
class Node:
    """A Node in a singly linked list."""

    def __init__(self, data):
        self.data = data       # Store data
        self.next = None       # Pointer to the next node (initially None)


class LinkedList:
    """Singly Linked List implementation with basic operations."""

    def __init__(self):
        self.head = None  # Initially, the list is empty

    def insert(self, data):
        """
        Insert a new node with given data at the end of the list.
        """
        new_node = Node(data)

        # Case 1: If the list is empty, new node becomes the head
        if self.head is None:
            self.head = new_node
            return

        # Case 2: Traverse to the end and add the new node
        current = self.head
        while current.next:  # move until last node (next == None)
            current = current.next

        current.next = new_node  # last node points to new_node

    def delete_value(self, value):
        """
        Delete the first occurrence of 'value' in the linked list.
        """
        current = self.head
```

L1T1.PY

L1T1.PY

```python
class LinkedList:
    def delete_value(self, value):
        prev = None

        # Case 1: Empty list
        if current is None:
            return

        # Case 2: Value is at the head
        if current.data == value:
            # Update head pointer to skip the deleted node
            self.head = current.next
            return

        # Case 3: Search for the value in the rest of the list
        while current and current.data != value:
            prev = current
            current = current.next

        # If not found, do nothing
        if current is None:
            return

        # Bypass the current node
        # prev -> current -> current.next
        # Update: prev.next should now point to current.next
        prev.next = current.next

    def traverse(self):
        """
        Traverse the linked list and return all elements as a list
        """
        elements = []
        current = self.head

        while current:
```

L1T1.PY

```python
class LinkedList:
    def traverse(self):
        while current:
            elements.append(current.data)
            current = current.next  # move pointer to next node

        return elements


# Example usage
if __name__ == "__main__":
    ll = LinkedList()
    ll.insert(10)
    ll.insert(20)
    ll.insert(30)

    print("After insertions:", ll.traverse())  # [10, 20, 30]

    ll.delete_value(20)
    print("After deleting 20:", ll.traverse())  # [10, 30]

    ll.delete_value(10)
    print("After deleting head (10):", ll.traverse())  # [30]

    ll.delete_value(99)  # Not in list
    print("After attempting to delete 99:", ll.traverse())  # [30]
```

OUTPUT:

```
[Running] python -u "c:\Users\Charan\OneDrive\Desktop\python\L1T1.PY"
After insertions: [10, 20, 30]
After deleting 20: [10, 30]
After deleting head (10): [30]
After attempting to delete 99: [30]
```

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 4

TASK DESCRIPTION :

- 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 :

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:

L1T1.PY ×

L1T1.PY

```python
class Node:
    """
    A Node in the Binary Search Tree (BST).

    Attributes
    ----------
    key : int
        The value stored in the node.
    left : Node, optional
        Pointer to the left child node.
    right : Node, optional
        Pointer to the right child node.
    """
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None


class BinarySearchTree:
    """
    A Binary Search Tree (BST) implementation.

    Methods
    -------
    insert(key):
        Inserts a new key into the BST.
    search(key):
        Searches for a key in the BST. Returns True if found, else False.
    inorder_traversal():
        Returns the inorder traversal (sorted order) of the BST.
    """

    def __init__(self):
        self.root = None

    def insert(self, key):
        """Insert a new key into the BST."""
        if self.root is None:
            self.root = Node(key)
        else:
            self._insert_recursive(self.root, key)

```

```python
    def _search_recursive(self, current, key):
        """Helper method to recursively search for a key."""
        if current is None:
            return False
        if current.key == key:
            return True
        elif key < current.key:
            return self._search_recursive(current.left, key)
        else:
            return self._search_recursive(current.right, key)

    def inorder_traversal(self):
        """Return the inorder traversal (sorted order) of the BST as a list."""
        result = []
        self._inorder_recursive(self.root, result)
        return result

    def _inorder_recursive(self, current, result):
        """Helper method for inorder traversal."""
        if current:
            self._inorder_recursive(current.left, result)
            result.append(current.key)
            self._inorder_recursive(current.right, result)


# -------- TESTING --------
if __name__ == "__main__":
    bst = BinarySearchTree()
    numbers = [50, 30, 70, 20, 40, 60, 80]

    # Insert numbers into BST
    for num in numbers:
        bst.insert(num)

    # Test inorder traversal (should be sorted list)
    print("Inorder Traversal:", bst.inorder_traversal())

    # Test search for present elements
    print("Search 40:", bst.search(40))  # Expected: True
    print("Search 70:", bst.search(70))  # Expected: True

    # Test search for absent elements
    print("Search 25:", bst.search(25))  # Expected: False
    print("Search 90:", bst.search(90))  # Expected: False
```

OUTPUT:

```
[Running] python -u "c:\Users\Charan\OneDrive\Desktop\python\L1T1.PY
Inorder Traversal: [20, 30, 40, 50, 60, 70, 80]
Search 40: True
Search 70: True
Search 25: False
Search 90: False
```

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 5

TASK DESCRIPTION:

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:

L1T1.PY

L1T1.PY

```python
from collections import deque

class Graph:
    def __init__(self):
        """Initialize the graph using an adjacency list (dictionary)."""
        self.adj_list = {}

    def add_edge(self, u, v):
        """Add an edge to the graph (undirected by default)."""
        if u not in self.adj_list:
            self.adj_list[u] = []
        if v not in self.adj_list:
            self.adj_list[v] = []
        self.adj_list[u].append(v)
        self.adj_list[v].append(u)  # comment this line if you want directed graph

    def bfs(self, start):
        """Breadth-First Search (BFS) starting from a node."""
        visited = set()            # keep track of visited nodes
        queue = deque([start])     # queue for BFS
        visited.add(start)

        while queue:
            node = queue.popleft()  # dequeue front element
            print(node, end=" ")    # process node (here just print)

            # visit all unvisited neighbors
            for neighbor in self.adj_list[node]:
                if neighbor not in visited:
                    visited.add(neighbor)
                    queue.append(neighbor)

    def dfs_recursive(self, start, visited=None):
        """Recursive Depth-First Search (DFS)."""
        if visited is None:
            visited = set()

        visited.add(start)
        print(start, end=" ")   # process node

        # explore unvisited neighbors recursively
        for neighbor in self.adj_list[start]:
            if neighbor not in visited:
                self.dfs_recursive(neighbor, visited)
```

```python
    def dfs_iterative(self, start):
        """Iterative Depth-First Search (DFS) using a stack."""
        visited = set()
        stack = [start]

        while stack:
            node = stack.pop()      # pop last element (LIFO)
            if node not in visited:
                print(node, end=" ")   # process node
                visited.add(node)

                # add neighbors to stack (reverse for consistent order)
                for neighbor in reversed(self.adj_list[node]):
                    if neighbor not in visited:
                        stack.append(neighbor)


# --------------------------
# Example Usage
# --------------------------
if __name__ == "__main__":
    # Create graph instance
    g = Graph()

    # Add edges
    g.add_edge(1, 2)
    g.add_edge(1, 3)
    g.add_edge(2, 4)
    g.add_edge(2, 5)
    g.add_edge(3, 6)
    g.add_edge(3, 7)

    print("BFS starting from 1:")
    g.bfs(1)   # Expected: 1 2 3 4 5 6 7

    print("\n\nDFS Recursive starting from 1:")
    g.dfs_recursive(1)  # Expected: 1 2 4 5 3 6 7

    print("\n\nDFS Iterative starting from 1:")
    g.dfs_iterative(1)  # Expected: 1 2 4 5 3 6 7 (order may vary slightly)
```

OUTPUT:

```
PS C:\Users\Charan\OneDrive\Desktop\python> PYTHON L1T1.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
```

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.