AI-ASSISSTANT-CODING-LAB-11.4

Name: K . Dinesh

Batch:42

Roll-No:2303A52288

Task 1: Stack Implementation for Undo Operations (LIFO)
Scenario
You are building a text editor where users can undo their recent actions
(typing, deleting, formatting). Each action must be reversed in Last-In-First-
Out (LIFO) order.
Task Description
• Use an AI coding assistant to help implement a Stack class in Python
with the following methods:
o push(action)
o pop()
o peek()
o is_empty()
• Prompt the AI to:
o Generate a clean class skeleton with docstrings
o Explain why a stack is suitable for undo functionality
o Suggest an alternative implementation using collections.deque

CODE:

```python
# TASK 1: Stack for Undo Operations (LIFO)
# ========================================================================
# PROMPT: "Create Stack class with push(), pop(), peek(), is_empty() for undo functionality"

class Stack:
    def __init__(self):
        self.items = []

    def push(self, action):
        self.items.append(action)

    def pop(self):
        if not self.is_empty():
            return self.items.pop()

    def peek(self):
        if not self.is_empty():
            return self.items[-1]

    def is_empty(self):
        return len(self.items) == 0
```

```python
# Test Cases
print("\n" + "="*60)
print("TASK 1: Stack for Undo Operations")
print("="*60)

print("\nTest 1: Basic Stack Operations")
stack = Stack()
print(f"Is empty: {stack.is_empty()}")
stack.push("Type: Hello")
stack.push("Type: World")
stack.push("Format: Bold")
stack.push("Type: !")
stack.push("Format: Italic")
print(f"Stack size: {len(stack.items)}")
print(f"Top action (peek): {stack.peek()}")

print("\nTest 2: Undo Multiple Actions")
print(f"Undo 1: {stack.pop()}")
print(f"Undo 2: {stack.pop()}")
print(f"Undo 3: {stack.pop()}")
print(f"Remaining: {len(stack.items)} actions")
print(f"Current top: {stack.peek()}")

print("\nTest 3: Undo All Actions")
while not stack.is_empty():
    print(f"Undo: {stack.pop()}")
print(f"Stack is empty: {stack.is_empty()}")

print("\nTest 4: Deque-based Stack")
deque_stack = Stack()
for i in range(5):
    deque_stack.push(f"Action {i+1}")
print(f"Added 5 actions, top: {deque_stack.peek()}")
for i in range(3):
    deque_stack.pop()
print(f"After 3 undos, remaining: {len(deque_stack.items)} actions")
```

OUTPUT:

```
===============================================================
TASK 1: Stack for Undo Operations
===============================================================


Test 1: Basic Stack Operations
Is empty: True
Stack size: 5
Top action (peek): Format: Italic

Test 2: Undo Multiple Actions
Undo 1: Format: Italic
Undo 2: Type: !
Undo 3: Format: Bold
Remaining: 2 actions
Current top: Type: World

Test 3: Undo All Actions
Undo: Type: World
Undo: Type: Hello
Stack is empty: True


Test 4: Deque-based Stack
Added 5 actions, top: Action 5
After 3 undos, remaining: 2 actions
```

Justification:
A stack follows the Last In, First Out (LIFO) principle, which is ideal for undo functionality. The most recent action must be undone first. The implemented stack supports push, pop, peek, and empty check operations, ensuring reliable undo behavior. Multiple test cases validate normal usage, multiple undos, and empty stack handling.

Task 2: Queue for Customer Service Requests (FIFO)
Scenario
You are developing a customer support system where service requests must be handled in the order they arrive.
Task Description
• Implement a Queue class with:
o enqueue(request)
o dequeue()
o is_empty()
• First, use a Python list-based approach.

• Then, ask AI to:

o Review the performance implications of list-based queues

o Suggest and implement an optimized version using collections.deque

o Explain why deque performs better for queues

CODE:

```python
# ========================================================================
# TASK 2: Queue for Customer Service (FIFO)
# ========================================================================
# PROMPT: "Create Queue with enqueue(), dequeue(), is_empty(). Compare list vs deque performance"

class ListQueue:
    def __init__(self):
        self.items = []

    def enqueue(self, item):
        self.items.append(item)

    def dequeue(self):  # O(n) - inefficient
        if not self.is_empty():
            return self.items.pop(0)

    def is_empty(self):
        return len(self.items) == 0


class DequeQueue:
    def __init__(self):
        self.items = deque()

    def enqueue(self, item):
        self.items.append(item)

    def dequeue(self):  # O(1) - efficient
        if not self.is_empty():
            return self.items.popleft()

    def is_empty(self):
        return len(self.items) == 0
```

```python
# Test Cases
print("\n" + "="*60)
print("TASK 2: Queue for Customer Service")
print("="*60)


print("\nTest 1: List-based Queue (O(n) dequeue)")
list_q = ListQueue()
list_q.enqueue("Ticket #101: Password Reset")
list_q.enqueue("Ticket #102: Billing Issue")
list_q.enqueue("Ticket #103: Feature Request")
list_q.enqueue("Ticket #104: Bug Report")
print(f"Queue size: {len(list_q.items)}")
print(f"Process: {list_q.dequeue()}")
print(f"Process: {list_q.dequeue()}")
print(f"Remaining: {len(list_q.items)} tickets")


print("\nTest 2: Deque-based Queue (O(1) dequeue - Optimized)")
queue = DequeQueue()
tickets = [
    "Ticket #201: Account Locked",
    "Ticket #202: Refund Request",
    "Ticket #203: Technical Support",
    "Ticket #204: Installation Help",
    "Ticket #205: Data Recovery"
]
for ticket in tickets:
    queue.enqueue(ticket)
    print(f"Enqueued: {ticket}")

print(f"\nQueue size: {len(queue.items)}")
print("\nProcessing tickets in FIFO order:")
while not queue.is_empty():
    print(f"  Processing: {queue.dequeue()}")
```

```python
print("\nTest 3: Edge Cases")
queue2 = DequeQueue()
print(f"Empty queue: {queue2.is_empty()}")
queue2.enqueue("Single Ticket")
print(f"Added one ticket, is empty: {queue2.is_empty()}")
print(f"Dequeue: {queue2.dequeue()}")
print(f"Now empty: {queue2.is_empty()}")
```

OUTPUT:

```
================================================================
TASK 2: Queue for Customer Service
================================================================


Test 1: List-based Queue (O(n) dequeue)
Queue size: 4
Process: Ticket #101: Password Reset
Process: Ticket #102: Billing Issue
Remaining: 2 tickets

Test 2: Deque-based Queue (O(1) dequeue - Optimized)
Enqueued: Ticket #201: Account Locked
Enqueued: Ticket #202: Refund Request
Enqueued: Ticket #203: Technical Support
Enqueued: Ticket #204: Installation Help
Enqueued: Ticket #205: Data Recovery

Queue size: 5

Processing tickets in FIFO order:
  Processing: Ticket #201: Account Locked
  Processing: Ticket #202: Refund Request
  Processing: Ticket #203: Technical Support
  Processing: Ticket #204: Installation Help
  Processing: Ticket #205: Data Recovery

Test 3: Edge Cases
Empty queue: True
Added one ticket, is empty: False
Dequeue: Single Ticket
Now empty: True
```

Justification:
Customer service systems require First In, First Out (FIFO) processing to maintain fairness. This task compares a list-based queue (O(n) dequeue) with a deque-based queue (O(1) dequeue). The comparison justifies why deque is more efficient for real-world queue systems where performance matters.

Task 3: Singly Linked List for Dynamic Playlist Management
Scenario
You are designing a music playlist feature where songs can be added or

removed dynamically while maintaining order.

Task Description

• Implement a Singly Linked List with:

o insert_at_end(song)

o delete_value(song)

o traverse()

• Use AI to:

o Add inline comments explaining pointer manipulation

o Highlight tricky parts of insertion and deletion

o Suggest edge case test scenarios (empty list, single node, deletion at head)

CODE:

```python
# TASK 3: Linked List for Playlist
# ==========================================================================
# PROMPT: "Create Singly Linked List with insert_at_end(), delete_value(), traverse()"

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def insert_at_end(self, song):
        new_node = Node(song)
        if self.head is None:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node

    def delete_value(self, song):
        if self.head is None:
            return False

        if self.head.data == song:
            self.head = self.head.next
            return True

    def traverse(self):
        songs = []
        current = self.head
        while current:
            songs.append(current.data)
            current = current.next
        return songs
```

```python
# Test Cases
print("\n" + "="*60)
print("TASK 3: Linked List for Playlist")
print("="*60)

print("\nTest 1: Building Playlist")
playlist = LinkedList()
songs = [
    "Bohemian Rhapsody - Queen",
    "Stairway to Heaven - Led Zeppelin",
    "Hotel California - Eagles",
    "Imagine - John Lennon",
    "Sweet Child O' Mine - Guns N' Roses"
]
for song in songs:
    playlist.insert_at_end(song)
    print(f"Added: {song}")

print("\nTest 2: Display Full Playlist")
current_playlist = playlist.traverse()
for i, song in enumerate(current_playlist, 1):
    print(f"  {i}. {song}")

print("\nTest 3: Delete from Middle")
playlist.delete_value("Hotel California - Eagles")
print("Deleted: Hotel California - Eagles")
print(f"Playlist: {playlist.traverse()}")

print("\nTest 4: Delete from Head")
playlist.delete_value("Bohemian Rhapsody - Queen")
print("Deleted: Bohemian Rhapsody - Queen (head)")
print(f"Playlist: {playlist.traverse()}")
```

OUTPUT:

```
============================================================
TASK 3: Linked List for Playlist
============================================================

Test 1: Building Playlist
Added: Bohemian Rhapsody - Queen
Added: Stairway to Heaven - Led Zeppelin
Added: Hotel California - Eagles
Added: Imagine - John Lennon
Added: Sweet Child O' Mine - Guns N' Roses

Test 2: Display Full Playlist
  1. Bohemian Rhapsody - Queen
  2. Stairway to Heaven - Led Zeppelin
  3. Hotel California - Eagles
  4. Imagine - John Lennon
  5. Sweet Child O' Mine - Guns N' Roses

Test 3: Delete from Middle
Deleted: Hotel California - Eagles
Playlist: ['Bohemian Rhapsody - Queen', 'Stairway to Heaven - Led Zeppelin', 'Imagine - John Lennon', "Sweet Child O' Mine - Guns N' 
es"]

Test 4: Delete from Head
Deleted: Bohemian Rhapsody - Queen (head)
Playlist: ['Stairway to Heaven - Led Zeppelin', 'Imagine - John Lennon', "Sweet Child O' Mine - Guns N' Roses"]

Test 5: Delete from End
Deleted: Sweet Child O' Mine (end)
Playlist: ['Stairway to Heaven - Led Zeppelin', 'Imagine - John Lennon']
```

```
Test 6: Try to Delete Non-existent Song
Delete 'Thriller': False

Test 7: Final Playlist
  1. Stairway to Heaven - Led Zeppelin
  2. Imagine - John Lennon
```

Justification:

A singly linked list allows dynamic insertion and deletion without shifting elements, making it suitable for playlists. Songs can be added or removed from any position efficiently. The test cases justify correct handling of head, middle, end, and non-existent deletions.

Task 4: Binary Search Tree for Fast Record Lookup

Scenario

You are building a student record system where quick searching by roll number is required.

Task Description

• Implement a Binary Search Tree (BST) with:

o insert(value)

o search(value)

o inorder_traversal()

• Provide AI with a partially written Node and BST class.

• Ask AI to:

o Complete missing methods

o Add meaningful docstrings

o Explain how BST improves search efficiency over linear search

CODE:

```
# ==========================================================================
# TASK 4: Binary Search Tree for Student Records
# ==========================================================================
# PROMPT: "Create BST with insert(), search(), inorder_traversal() for fast lookup"

class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if self.root is None:
            self.root = TreeNode(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        if value < node.value:
            if node.left is None:
                node.left = TreeNode(value)
            else:
                self._insert_recursive(node.left, value)
        elif value > node.value:
            if node.right is None:
                node.right = TreeNode(value)
            else:
                self._insert_recursive(node.right, value)
```

```python
    def _search_recursive(self, node, value):
        if node is None:
            return False
        if value == node.value:
            return True
        elif value < node.value:
            return self._search_recursive(node.left, value)
        else:
            return self._search_recursive(node.right, value)

    def inorder_traversal(self):
        result = []
        self._inorder_recursive(self.root, result)
        return result

    def _inorder_recursive(self, node, result):
        if node:
            self._inorder_recursive(node.left, result)
            result.append(node.value)
            self._inorder_recursive(node.right, result)

# Test Cases
print("\n" + "="*60)
print("TASK 4: BST for Student Records")
print("="*60)

print("\nTest 1: Inserting Student Roll Numbers")
bst = BinarySearchTree()
roll_numbers = [50, 30, 70, 20, 40, 60, 80, 10, 25, 35, 65, 90]
for roll in roll_numbers:
    bst.insert(roll)
    print(f"Inserted Roll #{roll}")

print(f"\nTotal students: {len(roll_numbers)}")

print("\nTest 2: Searching Existing Records")
search_existing = [50, 30, 70, 10, 35, 65, 90]
print("Searching for students:")
for roll in search_existing:
    found = bst.search(roll)
    print(f"  Roll #{roll}: {'Found ✓' if found else 'Not Found X'}")

print("\nTest 3: Searching Non-existent Records")
search_missing = [5, 45, 55, 75, 100]
print("Searching for non-existent rolls:")
for roll in search_missing:
    found = bst.search(roll)
    print(f"  Roll #{roll}: {'Found ✓' if found else 'Not Found X'}")

print("\nTest 4: Inorder Traversal (Sorted Order)")
sorted_rolls = bst.inorder_traversal()
print(f"All rolls in sorted order:")
print(f"  {sorted_rolls}")
```

OUTPUT:

```
TASK 4: BST for Student Records
==============================================================

Test 1: Inserting Student Roll Numbers
Inserted Roll #50
Inserted Roll #30
Inserted Roll #70
Inserted Roll #20
Inserted Roll #40
Inserted Roll #60
Inserted Roll #80
Inserted Roll #10
Inserted Roll #25
Inserted Roll #35
Inserted Roll #65
Inserted Roll #90

Total students: 12

Test 2: Searching Existing Records
Searching for students:
  Roll #50: Found ✓
  Roll #30: Found ✓
  Roll #70: Found ✓
  Roll #10: Found ✓
  Roll #35: Found ✓
  Roll #65: Found ✓
  Roll #90: Found ✓

 Test 3: Searching Non-existent Records
 Searching for non-existent rolls:
   Roll #5: Not Found X
   Roll #45: Not Found X
   Roll #55: Not Found X
   Roll #75: Not Found X
   Roll #100: Not Found X

 Test 4: Inorder Traversal (Sorted Order)
 All rolls in sorted order:
   [10, 20, 25, 30, 35, 40, 50, 60, 65, 70, 80, 90]

 Test 5: BST Efficiency
   Total students: 12
   Linear search max checks: 12
   BST average checks: 4 (log₂ n)
   Efficiency gain: 3.0x faster
```

Justification:

A Binary Search Tree enables fast searching and sorted data retrieval. Student roll numbers are stored efficiently, reducing search complexity from O(n) to O(log n). Inorder traversal confirms sorted order, and efficiency comparison proves the advantage over linear search.

Task 5: Graph Traversal for Social Network Connections

Scenario

You are modeling a social network, where users are connected to friends, and you want to explore connections.

Task Description

• Represent the network using a graph (adjacency list).

• Use AI to implement:

o Breadth-First Search (BFS) to find nearby connections

o Depth-First Search (DFS) to explore deep connection paths

• Ask AI to:

o Add inline comments explaining traversal steps

o Compare recursive and iterative DFS approaches

o Suggest practical use cases for BFS vs DFS

CODE:

```python
# TASK 5: Graph Traversal for Social Network
# =============================================================================
# PROMPT: "Create Graph with BFS and DFS (iterative & recursive) for social connections"

class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
        self.graph[u].append(v)
        self.graph[v].append(u)

    def bfs(self, start):
        visited = set()
        queue = deque([start])
        result = []

        while queue:
            node = queue.popleft()
            if node not in visited:
                visited.add(node)
                result.append(node)
                for neighbor in self.graph.get(node, []):
                    if neighbor not in visited:
                        queue.append(neighbor)
        return result
```

```python
    def dfs_iterative(self, start):
        visited = set()
        stack = [start]
        result = []

        while stack:
            node = stack.pop()
            if node not in visited:
                visited.add(node)
                result.append(node)
                for neighbor in reversed(self.graph.get(node, [])):
                    if neighbor not in visited:
                        stack.append(neighbor)
        return result

    def dfs_recursive(self, start, visited=None, result=None):
        if visited is None:
            visited = set()
            result = []

        visited.add(start)
        result.append(start)

        for neighbor in self.graph.get(start, []):
            if neighbor not in visited:
                self.dfs_recursive(neighbor, visited, result)

        return result

# Test Cases
print("\n" + "="*60)
print("TASK 5: Graph Traversal for Social Network")
print("="*60)

print("\nTest 1: Building Social Network")
g = Graph()
connections = [
    ("Alice", "Bob"),
    ("Alice", "Charlie"),
    ("Alice", "David"),
    ("Bob", "Emma"),
    ("Bob", "Frank"),
    ("Charlie", "George"),
    ("David", "Henry"),
    ("Emma", "Ivan"),
    ("Frank", "Ivan"),
    ("George", "Henry")
]

for user1, user2 in connections:
    g.add_edge(user1, user2)
    print(f"Connected: {user1} ↔ {user2}")

print("\nNetwork Structure:")
for user, friends in sorted(g.graph.items()):
    print(f"  {user}: {', '.join(friends)}")

print("\nTest 2: BFS Traversal (Level by Level)")
bfs_result = g.bfs('Alice')
print(f"BFS from Alice: {' → '.join(bfs_result)}")
print("Purpose: Find shortest path, nearby connections")
```

OUTPUT:

```
============================================================
TASK 5: Graph Traversal for Social Network
============================================================

Test 1: Building Social Network
Connected: Alice ↔ Bob
Connected: Alice ↔ Charlie
Connected: Alice ↔ David
Connected: Bob ↔ Emma
Connected: Bob ↔ Frank
Connected: Charlie ↔ George
Connected: David ↔ Henry
Connected: Emma ↔ Ivan
Connected: Frank ↔ Ivan
Connected: George ↔ Henry

Network Structure:
  Alice: Bob, Charlie, David
  Bob: Alice, Emma, Frank
  Charlie: Alice, George
  David: Alice, Henry
  Emma: Bob, Ivan
  Frank: Bob, Ivan
  George: Charlie, Henry
  Henry: David, George
  Ivan: Emma, Frank

Test 2: BFS Traversal (Level by Level)
BFS from Alice: Alice → Bob → Charlie → David → Emma → Frank → George → Henry → Ivan
Purpose: Find shortest path, nearby connections

 Test 3: DFS Iterative (Deep Exploration)
 DFS Iterative: Alice → Bob → Emma → Ivan → Frank → Charlie → George → Henry → David
 Purpose: Explore deep connections, detect connectivity

 Test 4: DFS Recursive (Natural Backtracking)
 DFS Recursive: Alice → Bob → Emma → Ivan → Frank → Charlie → George → Henry → David
 Purpose: Same as iterative but cleaner code

 Test 5: Traversal from Different Starting Points
 BFS from Bob: Bob → Alice → Emma → Frank → Charlie → David → Ivan → George → Henry
 BFS from Emma: Emma → Bob → Ivan → Alice → Frank → Charlie → David → George → Henry
 BFS from George: George → Charlie → Henry → Alice → David → Bob → Emma → Frank → Ivan

 Test 6: Compare BFS vs DFS Order
 BFS visits nearby friends first: ['Alice', 'Bob', 'Charlie', 'David']
 DFS goes deep in one path first: ['Alice', 'Bob', 'Emma', 'Ivan']
 BFS = Shortest path | DFS = Any path
```

Justification:
Graphs naturally represent social networks where users are connected bidirectionally.
- BFS is used to find nearest connections and shortest paths.
- DFS explores deep relationships and connectivity.
  Both iterative and recursive DFS implementations demonstrate flexibility and correctness across different traversal strategies.