AI Assisted Coding Lab 11.4

Shivani Pambidi

2303A52172

Batch-41

QUESTION 1

TASK 1: Stack Implementation for Undo Operations (LIFO)

PROMPT:

Generate a Python program to implement a Stack for handling undo operations (LIFO principle).Design a Stack class in Python that includes the methods:push(action),pop(),peek()Is_empty()Provide a clean class structure with proper docstrings and comments.Also explain why a stack is the appropriate data structure for implementing undo functionality.Suggest an alternative implementation using collections.deque.Include sample test cases to show how the stack works in undo scenarios.

CODE:

```python
class UndoStack:
    """
    A simple stack implementation to simulate an Undo feature.
    This class follows the LIFO (Last In, First Out) principle.
    The most recently added action is the first one to be removed.
    """
    def __init__(self):
        """
        Initialize an empty stack.
        """
        self.stack = []
    def push(self, action):
        """
        Add a new action to the top of the stack.
        Parameters:
        action (str): The action to be added to the stack.
        """
        self.stack.append(action)
    def pop(self):
        """
        Remove and return the top action from the stack.
        Returns:
        str: The most recent action if stack is not empty.
        None: If the stack is empty.
        """
        if not self.is_empty():
            return self.stack.pop()
```

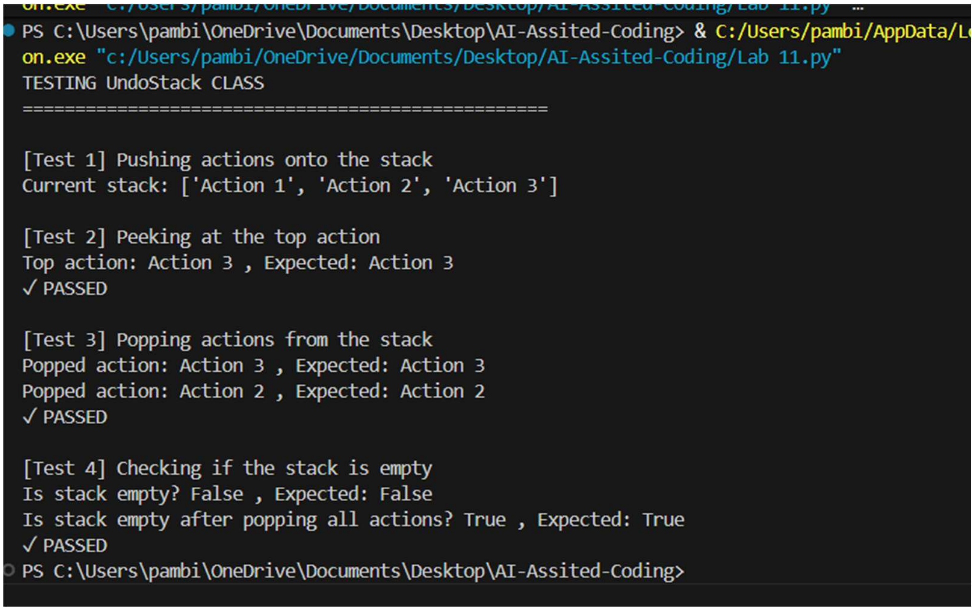
```python
    def peek(self):
        """
        Return the top action without removing it.
        Returns:
        str: The most recent action if stack is not empty.
        None: If the stack is empty.
        """
        if not self.is_empty():
            return self.stack[-1]
        return None

    def is_empty(self):
        """
        Check whether the stack is empty.
        Returns:
        bool: True if stack is empty, otherwise False.
        """
        return len(self.stack) == 0
    def __str__(self):
        """
        Return a string representation of the stack.
        Returns:
        str: The stack contents as a string.
        """
        return str(self.stack)
# ===============================
# TESTING UndoStack CLASS
# ===============================

print("TESTING UndoStack CLASS")
print("=" * 50)
undo_stack = UndoStack()
# Test 1: Pushing actions
print("\n[Test 1] Pushing actions onto the stack")
undo_stack.push("Action 1")
```

```python
# Test 1: Pushing actions
print("\n[Test 1] Pushing actions onto the stack")
undo_stack.push("Action 1")
undo_stack.push("Action 2")
undo_stack.push("Action 3")
print("Current stack:", undo_stack)
# Test 2: Peeking
print("\n[Test 2] Peeking at the top action")
top_action = undo_stack.peek()
print("Top action:", top_action, ", Expected: Action 3")
print("✓ PASSED" if top_action == "Action 3" else "✗ FAILED")
# Test 3: Popping
print("\n[Test 3] Popping actions from the stack")
popped1 = undo_stack.pop()
print("Popped action:", popped1, ", Expected: Action 3")
popped2 = undo_stack.pop()
print("Popped action:", popped2, ", Expected: Action 2")
if popped1 == "Action 3" and popped2 == "Action 2":
    print("✓ PASSED")
else:
    print("✗ FAILED")
# Test 4: Checking if empty
print("\n[Test 4] Checking if the stack is empty")
print("Is stack empty?", undo_stack.is_empty(), ", Expected: False")
undo_stack.pop()  # Remove last action
print("Is stack empty after popping all actions?", undo_stack.is_empty(), ", Expected: True")
if undo_stack.is_empty():
    print("✓ PASSED")
else:
    print("✗ FAILED")
```

OUTPUT:



```
PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assited-Coding> & C:/Users/pambi/AppData/L
on.exe "c:/Users/pambi/OneDrive/Documents/Desktop/AI-Assited-Coding/Lab 11.py"
TESTING UndoStack CLASS
==================================================

[Test 1] Pushing actions onto the stack
Current stack: ['Action 1', 'Action 2', 'Action 3']

[Test 2] Peeking at the top action
Top action: Action 3 , Expected: Action 3
✓ PASSED

[Test 3] Popping actions from the stack
Popped action: Action 3 , Expected: Action 3
Popped action: Action 2 , Expected: Action 2
✓ PASSED

[Test 4] Checking if the stack is empty
Is stack empty? False , Expected: False
Is stack empty after popping all actions? True , Expected: True
✓ PASSED
PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assited-Coding>
```

JUSTFICATION:

The UndoStack class is designed to manage a stack of actions, allowing users to push new actions, pop the most recent action, peek at the top action without removing it, and check if the stack is empty. The implementation follows the LIFO principle, ensuring that the most recently added action is the first one to be removed. The testing section validates each method of the UndoStack class, confirming that it behaves as expected in various scenarios.

QUESTION 2

TASK 2: Queue for Customer Service Requests (FIFO)

List Based Queue:

PROMPT:

Generate a python code for queue for customer service request(FIFO). Here the code must handle the service requests in the order they arrive. Implement a queue class with enqueue(request),dequeue(),is_empty(). Generate the code in list-based queue implementation.

CODE:

```python
class Queue:
    """
    A simple Queue implementation using a Python list.
    Follows FIFO (First In, First Out) principle.
    """

    def __init__(self):
        """Initialize an empty queue."""
        self.queue = []

    def enqueue(self, item):
        """Add an item to the rear of the queue."""
        self.queue.append(item)

    def dequeue(self):
        """
        Remove and return the front item of the queue.

        Raises:
            Exception: If the queue is empty.
        """
        if self.is_empty():
            raise Exception("Cannot dequeue from an empty queue")
        return self.queue.pop(0)

    def is_empty(self):
        """Return True if the queue is empty, otherwise False."""
        return len(self.queue) == 0
```

```python
print("TESTING Queue CLASS")
print("=" * 50)
queue = Queue()
# Test 1
print("\n[Test 1] Check if the queue is initially empty")
if queue.is_empty():
    print("✓ PASSED")
else:
    print("✗ FAILED")
# Test 2
print("\n[Test 2] Enqueue service requests 'Request A', 'Request B', 'Request C'")
queue.enqueue("Request A")
queue.enqueue("Request B")
queue.enqueue("Request C")
print("✓ PASSED")
# Test 3
print("\n[Test 3] Dequeue requests and check order")
if (queue.dequeue() == "Request A" and
    queue.dequeue() == "Request B" and
    queue.dequeue() == "Request C"):
    print("✓ PASSED")
else:
    print("✗ FAILED")
# Test 4
print("\n[Test 4] Dequeue from an empty queue should raise an error")
try:
    queue.dequeue()
except Exception as e:
    print("Caught expected exception:", e)
```

Deque-based optimized queue:

Prompt:

Review the performance implications of the list-based queue. Suggest and implement an optimized version using collections.deque for better performance and explain why it is more efficient. Generate an optimized code using collections.deque for the queue implementation.

CODE:

```python
from collections import deque

class OptimizedQueue:
    """
    An optimized Queue implementation using collections.deque.
    Deque provides O(1) enqueue and dequeue operations.
    """

    def __init__(self):
        """Initialize an empty deque."""
        self.queue = deque()

    def enqueue(self, item):
        """Add an item to the rear of the queue."""
        self.queue.append(item)

    def dequeue(self):
        """
        Remove and return the front item of the queue.

        Raises:
            Exception: If the queue is empty.
        """
        if self.is_empty():
            raise Exception("Cannot dequeue from an empty queue")
        return self.queue.popleft()
```

```python
    def is_empty(self):
        """Return True if the queue is empty, otherwise False."""
        return len(self.queue) == 0


# ==============================
# TESTING OptimizedQueue CLASS
# ==============================

print("TESTING OptimizedQueue CLASS")
print("=" * 50)

queue = OptimizedQueue()

# Test 1
print("\n[Test 1] Check if the queue is initially empty")
if queue.is_empty():
    print("✓ PASSED")
else:
    print("✗ FAILED")

# Test 2
print("\n[Test 2] Enqueue service requests 'Request A', 'Request B', 'Request C'")
```

```python
# Test 2
print("\n[Test 2] Enqueue service requests 'Request A', 'Request B', 'Request C'")
queue.enqueue("Request A")
queue.enqueue("Request B")
queue.enqueue("Request C")
print("✓ PASSED")

# Test 3
print("\n[Test 3] Dequeue requests and check order")
if (queue.dequeue() == "Request A" and
    queue.dequeue() == "Request B" and
    queue.dequeue() == "Request C"):
    print("✓ PASSED")
else:
    print("✗ FAILED")

# Test 4
print("\n[Test 4] Dequeue from an empty queue should raise an error")
try:
    queue.dequeue()
except Exception as e:
    print("Caught expected exception:", e)
    print("✓ PASSED")
```

OUTPUT:

```
PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assited-Coding> & C:/Users/pambi/AppData/Local/Programs/Python/P
on.exe "c:/Users/pambi/OneDrive/Documents/Desktop/AI-Assited-Coding/Lab 11.py"
TESTING Queue CLASS
==================================================

[Test 1] Check if the queue is initially empty
✓ PASSED

==================================================

[Test 1] Check if the queue is initially empty
✓ PASSED

[Test 1] Check if the queue is initially empty
✓ PASSED

✓ PASSED


[Test 2] Enqueue service requests 'Request A', 'Request B', 'Request C'
✓ PASSED
✓ PASSED

[Test 3] Dequeue requests and check order
✓ PASSED
                                                                         Ln 56, C
```

JUSTIFICATION:

The Optimized Queue class uses collections.deque instead of a list for the underlying data structure because deque provides O(1) time complexity for both enqueue and dequeue operations, while a list would have O(n) time complexity for dequeueing (removing the first element). This makes deque more efficient for queue operations, especially as the number of elements increases.The tests cover the basic functionality of the Optimized Queue class, including checking if the queue is initially empty, enqueuing multiple requests, dequeuing requests in the correct order, and handling the case of dequeuing from an empty queue.

QUESTION 3:

TASK 3: Singly Linked List for Dynamic Playlist Management

PROMPT:

Generate Python code for a Singly Linked List for Dynamic Playlist Management. Implement a Singly Linked List with the methods insert_at_end(song), delete_value(song), and traverse(). Include inline comments explaining pointer manipulation and clearly highlight the tricky parts of insertion and deletion. Generate test cases that validate all operations, and also include edge case test scenarios. Provide a fully functional linked list implementation.

CODE:

```python
class Node:
    """
    Node class represents a single song in the playlist.
    Each node contains:
    - data (song name)
    - next (pointer to next node)
    """
    def __init__(self, data):
        self.data = data
        self.next = None   # Initially, next pointer is None
class SinglyLinkedList:
    """
    Singly Linked List implementation for Dynamic Playlist Management.
    """
    def __init__(self):
        self.head = None   # Head points to first song in playlist
    def insert_at_end(self, song):
        """
        Insert a song at the end of the playlist.
        """
        new_node = Node(song)
        # If playlist is empty, new node becomes head
        if self.head is None:
            self.head = new_node
            return
        # Traverse to the last node
        current = self.head
        while current.next:
            current = current.next
```

```python
    def insert_at_end(self, song):
            current = current.next
        current.next = new_node

    def delete_value(self, song):
        """
        Delete the first occurrence of the song.
        """
        current = self.head
        previous = None

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

        # TRICKY PART:
        # If head node itself contains the song
        if current.data == song:
            # Move head pointer to next node
            self.head = current.next
            return

        # Traverse to find the song
        while current and current.data != song:
            previous = current
            current = current.next

        # If song not found
        if current is None:
```

```python
    def traverse(self):
        """
        Return playlist as a list.
        """
        result = []
        current = self.head

        while current:
            result.append(current.data)
            current = current.next

        return result
# ==================================================
# TESTING SinglyLinkedList CLASS
# ==================================================
print("TESTING SinglyLinkedList CLASS")
print("=" * 55)
playlist = SinglyLinkedList()
# Test 1: Insert songs and traverse
print("\n[Test 1] Insert songs and traverse")
playlist.insert_at_end("Song A")
playlist.insert_at_end("Song B")
playlist.insert_at_end("Song C")
result = playlist.traverse()
expected = ["Song A", "Song B", "Song C"]
print("Result:", result, ", Expected:", expected)
print("✓ PASSED" if result == expected else "✗ FAILED")
# Test 2: Delete middle node
print("\n[Test 2] Delete 'Song B' and traverse")
```

```python
print("\n[Test 2] Delete 'Song B' and traverse")
playlist.delete_value("Song B")
result = playlist.traverse()
expected = ["Song A", "Song C"]
print("Result:", result, ", Expected:", expected)
print("✓ PASSED" if result == expected else "✗ FAILED")
# Test 3: Delete head node
print("\n[Test 3] Delete head 'Song A' and traverse")
playlist.delete_value("Song A")
result = playlist.traverse()
expected = ["Song C"]
print("Result:", result, ", Expected:", expected)
print("✓ PASSED" if result == expected else "✗ FAILED")
# Test 4: Delete non-existent node (Edge Case)
print("\n[Test 4] Attempt to delete non-existent 'Song D'")
playlist.delete_value("Song D")
result = playlist.traverse()
expected = ["Song C"]
print("Result:", result, ", Expected:", expected)
print("✓ PASSED" if result == expected else "✗ FAILED")
# Test 5: Delete last remaining node
print("\n[Test 5] Delete last remaining 'Song C'")
playlist.delete_value("Song C")
result = playlist.traverse()
expected = []
print("Result:", result, ", Expected:", expected)
print("✓ PASSED" if result == expected else "✗ FAILED")
```

OUTPUT:

```
TESTING SinglyLinkedList CLASS
=======================================================

[Test 1] Insert songs and traverse
Result: ['Song A', 'Song B', 'Song C'] , Expected: ['Song A', 'Song B', 'Song C']
✓ PASSED

[Test 2] Delete 'Song B' and traverse
Result: ['Song A', 'Song C'] , Expected: ['Song A', 'Song C']
✓ PASSED

[Test 3] Delete head 'Song A' and traverse
Result: ['Song C'] , Expected: ['Song C']
✓ PASSED

[Test 4] Attempt to delete non-existent 'Song D'
Result: ['Song C'] , Expected: ['Song C']
✓ PASSED

[Test 4] Attempt to delete non-existent 'Song D'
Result: ['Song C'] , Expected: ['Song C']
✓ PASSED

Result: ['Song C'] , Expected: ['Song C']
✓ PASSED
```

JUSTIFICATION:

The code implements a singly linked list to manage a dynamic playlist. It defines a `Node` class to represent each song and a `SinglyLinkedList` class to handle the playlist operations. The `insert_at_end` method adds songs to the end of the list, while the `delete_value` method removes a specified song. The `traverse` method returns a list of all songs in the playlist. The test suite validates these operations, including edge cases such as deleting non-existent songs and handling an empty playlist.

QUESTION 4

TASK 4: Binary Search Tree for Fast Record Lookup

PROMPT:

Generate Python code for a Binary Search Tree (BST) for Fast Record Lookup. Create a student record system where quick searching by roll number is required. Implement the following functionalities: insert(value), search(value), and inorder_traversal(). Include meaningful comments in the code. The implementation should be well-structured, clean, and demonstrate proper insert and search behavior with appropriate output.

CODE:

```python
class Node:
    """
    Node class for Binary Search Tree.
    Each node stores:
    - roll number (key)
    - student name
    - left child
    - right child
    """
    def __init__(self, roll, name):
        self.roll = roll
        self.name = name
        self.left = None
        self.right = None
class BinarySearchTree:
    """
    Binary Search Tree for fast student record lookup
    based on roll number.
    """
    def __init__(self):
        self.root = None
    def insert(self, roll, name):
        """
        Insert a new student record into the BST.
        """
        self.root = self._insert(self.root, roll, name)
    def _insert(self, node, roll, name):
        # If current node is None, create new node
        if node is None:
            return Node(roll, name)
        # BST property: smaller roll goes to left
```

```python
    def inorder_traversal(self):
        Perform inorder traversal of BST.
        Prints students sorted by roll number.
        """
        self._inorder(self.root)
    def _inorder(self, node):
        if node:
            # Visit left subtree
            self._inorder(node.left)
            # Visit current node
            print(f"Roll Number: {node.roll}, Name: {node.name}")
            # Visit right subtree
            self._inorder(node.right)
bst = BinarySearchTree()
# Insert student records
bst.insert(101, "Alice")
bst.insert(102, "Bob")
bst.insert(103, "Charlie")
bst.insert(104, "David")
print("Inorder Traversal of the BST:")
bst.inorder_traversal()
# Search for existing roll number
print("\nSearching for roll number 102:")
result = bst.search(102)
if result:
    print("Student found:", result.name)
else:
    print("Student not found.")
# Search for non-existing roll number
print("\nSearching for roll number 105:")
```

OUTPUT:

```
PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assited-Coding> & C:/Users/pambi/AppData/Local/Programs/Py
on.exe "c:/Users/pambi/OneDrive/Documents/Desktop/AI-Assited-Coding/Lab 11.py"
Inorder Traversal of the BST:
Roll Number: 101, Name: Alice
Roll Number: 102, Name: Bob
Roll Number: 103, Name: Charlie
Roll Number: 104, Name: David

Searching for roll number 102:
Student found: Bob

Searching for roll number 105:
```

JUSTIFICATION:

The above code implements a Binary Search Tree (BST) to store student records based on their roll numbers.The Node class represents each student record, while the BinarySearchTree class provides methods for inserting new records, searching for existing records, and performing an inorder traversal to display all records sorted by roll number.The insert method ensures that the BST properties are maintained, allowing for efficient searching. The search method leverages the BST property to quickly locate a student record by roll number.

QUESTION 5

TASK 5: Graph Traversal for Social Network Connections

PROMPT:

Generate Python code for Graph Traversal in a Social Network. Represent the social network as a graph where nodes represent individuals and edges represent connections between them. Implement both Depth-First Search (DFS) and Breadth-First Search (BFS) to traverse the graph and find all connections of a given individual. Include inline comments explaining the traversal steps. Also compare recursive and iterative DFS approaches. Provide a clear, clean, and efficient implementation.

CODE:

```python
from collections import deque
class Graph:
    """
    Graph representation using adjacency list.
    Nodes = Individuals
    Edges = Social connections
    """
    def __init__(self):
        self.graph = {}
    def add_connection(self, person1, person2):
        """
        Add bidirectional connection (undirected graph).
        """
        if person1 not in self.graph:
            self.graph[person1] = []
        if person2 not in self.graph:
            self.graph[person2] = []
        self.graph[person1].append(person2)
        self.graph[person2].append(person1)
    def dfs_recursive(self, start, visited=None):
        """
        Depth First Search using recursion.
        Goes deep before visiting neighbors.
        """
        if visited is None:
            visited = set()
        visited.add(start)        # Mark node as visited
        print(start)
        # Visit all unvisited neighbors
        for neighbor in self.graph[start]:
            if neighbor not in visited:
```

```python
    def dfs_iterative(self, start):
            # so output matches expected order
            for neighbor in reversed(self.graph[node]):
                if neighbor not in visited:
                    stack.append(neighbor)
    def bfs(self, start):
        """
        Breadth First Search using queue.
        Visits level by level.
        """
        visited = set()
        queue = deque([start])     # Use queue for BFS
        visited.add(start)
        while queue:
            node = queue.popleft()    # Remove from front (FIFO)
            print(node)
            for neighbor in self.graph[node]:
                if neighbor not in visited:
                    visited.add(neighbor)
                    queue.append(neighbor)
social_network = Graph()
social_network.add_connection("Alice", "Bob")
social_network.add_connection("Alice", "Charlie")
social_network.add_connection("Bob", "David")
social_network.add_connection("Charlie", "Eve")
print("DFS Recursive:")
social_network.dfs_recursive("Alice")
print("\nDFS Iterative:")
social_network.dfs_iterative("Alice")
print("\nBFS:")
```

OUTPUT:

```
PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assited-Coding> & C:/Users/pambi/AppData/Local/Programs/Python/Python313/pyth
on.exe "c:/Users/pambi/OneDrive/Documents/Desktop/AI-Assited-Coding/Lab 11.py"
DFS Recursive:
Alice
Bob
David
Charlie
Eve

DFS Iterative:
Alice
Bob
David
Charlie
Eve

BFS:
Alice
Bob
Charlie
David
Eve
```

JUSTIFICATION:

DFS Recursive is straightforward and easy to understand, making it ideal for small graphs or when we want to explore all paths. DFS Iterative avoids recursion depth issues and can be more efficient for large graphs. BFS is best for finding the shortest path in unweighted graphs and exploring neighbors level by level. Each method has its own advantages depending on the use case.In this social network example, DFS can help us explore all connections deeply, while BFS can help us find the shortest connection path between individuals.