Name : K.Laxmi Prasanna

H.no:2303A52310

Batch-42

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:
  - push(action)
  - pop()
  - peek()
  - is_empty()
- Prompt the AI to:
  - Generate a clean class skeleton with **docstrings**
  - Explain why a stack is suitable for undo functionality
  - Suggest an alternative implementation using collections.deque

Expected Outcome

- A working Stack class with:
  - Correct LIFO behavior
  - Clear docstrings and comments
  - Sample test cases simulating undo operations
- A short AI-generated explanation comparing list-based and deque-based stacks

1. Stack Implementation Using Python List

```python
class Stack:
    """
    A simple Stack implementation using a Python list.

    This stack follows the Last-In-First-Out (LIFO) principle.
    It can be used to implement undo functionality in applications
    like text editors.
    """

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

    def push(self, action):
        """
        Push a new action onto the stack.

        :param action: The action to be added to the stack.
        """
        self._items.append(action)

    def pop(self):
        """
        Remove and return the top action from the stack.

        :return: The most recently added action.
        :raises IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("Cannot pop from an empty stack.")
        return self._items.pop()
```

```
    def peek(self):
        """
        Return the top action without removing it.

        :return: The most recent action.
        :raises IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("Cannot peek into an empty stack.")
        return self._items[-1]

    def is_empty(self):
        """
        Check whether the stack is empty.

        :return: True if empty, False otherwise.
        """
        return len(self._items) == 0

    def __len__(self):
        """Return the number of elements in the stack."""
        return len(self._items)
ef simulate_text_editor():
    undo_stack = Stack()

    # Simulating actions
    undo_stack.push("Typed 'Hello'")
    undo_stack.push("Typed ' World'")
    undo_stack.push("Deleted 'o'")
```

```
    print("Current top action:", undo_stack.peek())

    # Undo operations
    print("Undo:", undo_stack.pop())
    print("Undo:", undo_stack.pop())

    print("Is stack empty?", undo_stack.is_empty())
    print("Undo:", undo_stack.pop())

    print("Is stack empty now?", undo_stack.is_empty())


simulate_text_editor()
```

```
PS C:\Users\ramya\OneDrive\Desktop\coding> & C:\Users\ramya\AppData\Local\Microsoft\WindowsApps\python3.11.exe
 c:/Users/ramya/OneDrive/Desktop/coding/factorila.py
Current top action: Deleted 'o'
Undo: Deleted 'o'
Undo: Typed ' World'
Is stack empty? False
Undo: Typed 'Hello'
Is stack empty now? True
PS C:\Users\ramya\OneDrive\Desktop\coding>
```

2. Alternative Implementation Using collections.deque

```python
from collections import deque


class DequeStack:
    """
    Stack implementation using collections.deque.

    Deque provides efficient O(1) append and pop operations
    from both ends.
    """

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

    def push(self, action):
        """Push an action onto the stack."""
        self._items.append(action)

    def pop(self):
        """Remove and return the top action."""
        if self.is_empty():
            raise IndexError("Cannot pop from an empty stack.")
        return self._items.pop()

    def peek(self):
        """Return the top action without removing it."""
        if self.is_empty():
            raise IndexError("Cannot peek into an empty stack.")
        return self._items[-1]

    def is_empty(self):
        """Check whether the stack is empty."""
        return len(self._items) == 0
```

Comparison: List-Based vs Deque-Based Stack

Feature	List-Based Stack	Deque-Based Stack
Implementation	Simple, built-in list	Requires collections module
Push (append)	O(1)	O(1)
Pop (end)	O(1)	O(1)

Feature	List-Based Stack	Deque-Based Stack
Memory Efficiency	Slightly less optimized	More memory-efficient
Recommended For	Most simple stack use cases	High-performance or heavy operations

JUSTIFICATION:

A **stack** is ideal for undo functionality because it follows the **Last-In-First-Out (LIFO)** principle. The most recent action performed by the user must be undone first, which matches how a stack removes elements — the last item pushed is the first item popped.

Using a Python **list** is simple and efficient for stack operations (append() and pop() are O(1)). Alternatively, collections.deque provides similar O(1) performance and is slightly more optimized for heavy usage.

TASK-2:QUEUE FOR CUSTOMER SERVICE REQUESTS

Why a Queue is Suitable

A **queue** follows the **First-In-First-Out (FIFO)** principle.
In a customer support system:

- The **first customer request received** should be handled first.
- New requests go to the end of the line.

This ensures fairness and ordered processing.

List-Based Queue Implementation

```python
class ListQueue:
    """
    A simple Queue implementation using a Python list.

    This queue follows the First-In-First-Out (FIFO) principle.
    Suitable for basic customer service request handling.
    """

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

    def enqueue(self, request):
        """
        Add a new request to the end of the queue.

        :param request: The customer request to add.
        """
        self._items.append(request)

    def dequeue(self):
        """
        Remove and return the first request in the queue.

        :return: The oldest request.
        :raises IndexError: If the queue is empty.
        """
        if self.is_empty():
            raise IndexError("Cannot dequeue from an empty queue.")
        return self._items.pop(0)   # Removes first element

    def is_empty(self):
        """Return True if the queue is empty."""
        return len(self._items) == 0

    def __len__(self):
        """Return number of items in queue."""
        return len(self._items)
def simulate_customer_support():
    queue = ListQueue()

    queue.enqueue("Request 1: Password Reset")
    queue.enqueue("Request 2: Account Locked")
```

```
        queue.enqueue("Request 2: Account Locked")
        queue.enqueue("Request 3: Billing Issue")

        print("Processing:", queue.dequeue())
        print("Processing:", queue.dequeue())
        print("Processing:", queue.dequeue())

        print("Queue Empty?", queue.is_empty())


simulate_customer_support()
```

```
PS C:\Users\ramya\OneDrive\Desktop\coding> & C:\Users\ramya\AppData\
  c:/Users/ramya/OneDrive/Desktop/coding/factorila.py
Processing: Request 1: Password Reset
Processing: Request 2: Account Locked
Processing: Request 3: Billing Issue
Queue Empty? True
PS C:\Users\ramya\OneDrive\Desktop\coding>
```

Performance Review of List-Based Queue

Problem:

pop(0) has **O(n)** time complexity.

Why?
Because removing the first element shifts all remaining elements one position to the left.

If there are 10,000 requests:

- Every dequeue operation shifts 9,999 elements.
- This becomes slow in large systems.

Real-World Impact:

In a real customer service system:

- High traffic = thousands of requests
- Frequent dequeue operations
- Performance degradation over time

Optimized Queue Using collections.deque

factorila.py > ...

```python
from collections import deque


class DequeQueue:
    """
    Optimized Queue implementation using collections.deque.

    Deque provides efficient O(1) enqueue and dequeue
    operations from both ends.
    """

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

    def enqueue(self, request):
        """Add a request to the end of the queue."""
        self._items.append(request)

    def dequeue(self):
        """Remove and return the oldest request."""
        if self.is_empty():
            raise IndexError("Cannot dequeue from an empty queue.")
        return self._items.popleft()   # O(1) operation

    def is_empty(self):
        """Return True if queue is empty."""
        return len(self._items) == 0

    def __len__(self):
        """Return number of items in queue."""
        return len(self._items)

```

Performance Comparison

Operation	List-Based Queue	Deque-Based Queue
Enqueue	O(1)	O(1)
Dequeue	O(n)	O(1)
Memory Efficiency	Moderate	Optimized
Suitable For	Small systems	Large-scale systems

Why deque Performs Better

collections.deque is implemented as a doubly linked list (not a dynamic array like list).

This means:

- **No shifting of elements**
- **append() → O(1)**
- **popleft() → O(1)**
- **Efficient for both stack and queue operations**

A queue is appropriate for customer service requests because it follows the First-In-First-Out (FIFO) principle. The first request received should be handled first to ensure fairness and proper order of service.

A list-based queue works for small systems, but pop(0) takes O(n) time because all remaining elements must shift after removal. In high-traffic systems, this causes performance slowdowns.

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:
  - insert_at_end(song)
  - delete_value(song)
  - traverse()
- Use AI to:
  - Add inline comments explaining pointer manipulation
  - Highlight tricky parts of insertion and deletion
  - Suggest edge case test scenarios (empty list, single node, deletion at head)

Expected Outcome

- A fully functional linked list implementation
- Clear AI-generated comments explaining node traversal and pointer updates
- Test cases validating all operations

🎵 Why Singly Linked List for Playlist?

A Singly Linked List is suitable because:

- **Songs can be added dynamically without resizing (like arrays).**
- **Deletions are efficient when the node is known.**
- **Order of songs is maintained.**
- **Memory is allocated dynamically.**

Unlike lists, linked lists do not require shifting elements during deletion.

```python
class Node:
    """
    Represents a single song in the playlist.
    Each node stores the song name and a pointer to the next node.
    """
    def __init__(self, song):
        self.song = song
        self.next = None  # Pointer to the next node


class SinglyLinkedList:
    """
    Singly Linked List implementation for managing a music playlist.
    """

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

    def insert_at_end(self, song):
        """
        Insert a new song at the end of the playlist.
        """
        new_node = Node(song)

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

        # Case 2: Traverse to the last node
        current = self.head
        while current.next:  # Move until last node
            current = current.next

        # Link last node to new node
        current.next = new_node
```

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

        # Case 1: Empty list
        if current is None:
            print("Playlist is empty.")
            return

        # Case 2: Deletion at head
        if current.song == song:
            self.head = current.next  # Move head pointer forward
            return

        # Case 3: Deletion in middle or end
        while current and current.song != song:
            previous = current        # Keep track of previous node
            current = current.next    # Move forward

        # If song not found
        if current is None:
            print("Song not found.")
            return

        # Bypass the node to delete it
        previous.next = current.next

    def traverse(self):
        """
        Traverse and print all songs in the playlist.
        """
        current = self.head
```

```
        if current is None:
            print("Playlist is empty.")
            return

        while current:
            print(current.song, end=" -> ")
            current = current.next
        print("None")
ef test_playlist():
    playlist = SinglyLinkedList()

    # Edge Case 1: Empty list traversal
    playlist.traverse()

    # Insert songs
    playlist.insert_at_end("Song A")
    playlist.insert_at_end("Song B")
    playlist.insert_at_end("Song C")

    print("After insertion:")
    playlist.traverse()

    # Edge Case 2: Delete head
    playlist.delete_value("Song A")
    print("After deleting head (Song A):")
    playlist.traverse()

    # Delete middle
    playlist.delete_value("Song B")
    print("After deleting Song B:")
    playlist.traverse()

    # Edge Case 3: Delete last remaining node
    playlist.delete_value("Song C")
```

```python
def test_playlist():
    # Delete middle
    playlist.delete_value("Song B")
    print("After deleting Song B:")
    playlist.traverse()

    # Edge Case 3: Delete last remaining node
    playlist.delete_value("Song C")
    print("After deleting Song C:")
    playlist.traverse()

    # Edge Case 4: Delete from empty list
    playlist.delete_value("Song X")


test_playlist()
```

Output:

```
PS C:\Users\ramya\OneDrive\Desktop\coding> & C:\Users\ramya\AppData\Local\Microsoft\WindowsApps\python3.11.exe
  c:/Users/ramya/OneDrive/Desktop/coding/factorila.py
Playlist is empty.
After insertion:
Song A -> Song B -> Song C -> None
After deleting head (Song A):
Song B -> Song C -> None
After deleting Song B:
Song C -> None
After deleting Song C:
Playlist is empty.
Playlist is empty.
PS C:\Users\ramya\OneDrive\Desktop\coding>
```

JUSTIFICATION:

A **Singly Linked List** is suitable for dynamic playlist management because songs can be added or removed without shifting other elements, unlike arrays or lists. Each song (node) points to the next, making insertion at the end and deletion efficient through pointer updates.

Task 4: Binary Search Tree for Fast Record Lookup

Why Use a BST for Student Records?

A Binary Search Tree (BST) allows fast searching based on roll numbers because:

- **Left subtree contains values smaller than the node.**
- **Right subtree contains values greater than the node.**
- **This ordered structure reduces search space at every step.**

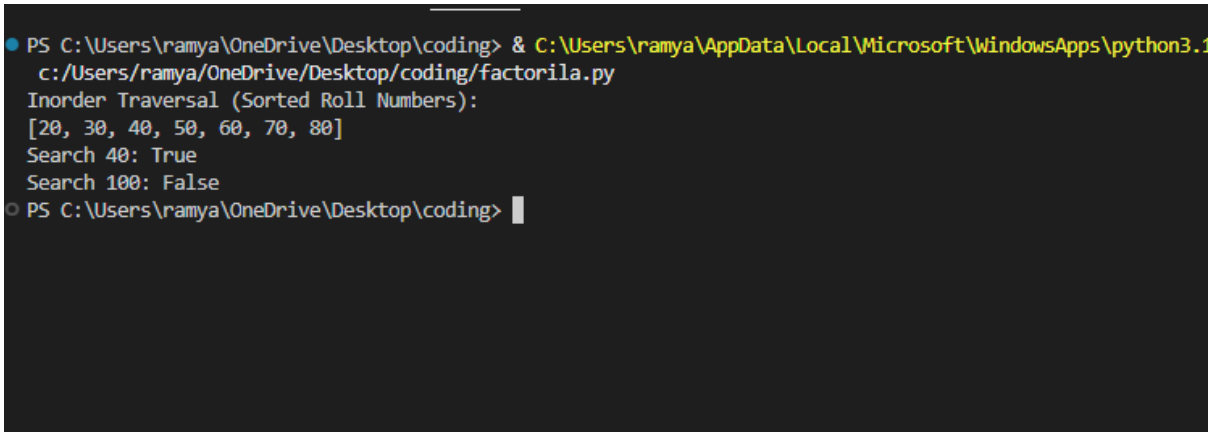
Instead of checking every record (like linear search), BST eliminates half the remaining records at each comparison (in ideal cases).

```python
class Node:
    """
    Represents a single node in the Binary Search Tree.

    Each node contains:
    - value: The roll number (or record key)
    - left: Reference to left child (values smaller than this node)
    - right: Reference to right child (values greater than this node)
    """
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None


class BST:
    """
    Binary Search Tree implementation for fast student record lookup.
    """

    def __init__(self):
        """Initialize an empty BST."""
        self.root = None

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

    def _insert_recursive(self, node, value):
        """
        Helper method to recursively insert a value
        at the correct position based on BST property.
        """
        if value < node.value:
            # Go to left subtree
```

```
            if node.left is None:
                node.left = Node(value)
            else:
                self._insert_recursive(node.left, value)
        elif value > node.value:
            # Go to right subtree
            if node.right is None:
                node.right = Node(value)
            else:
                self._insert_recursive(node.right, value)
        # If value == node.value, ignore (no duplicates allowed)

    def search(self, value):
        """
        Search for a value in the BST.

        Returns True if found, otherwise False.
        """
        return self._search_recursive(self.root, value)

    def _search_recursive(self, node, value):
        """
        Recursive helper method for searching.
        """
        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):
        """
        Perform inorder traversal.
```

```
        Returns elements in sorted order.
        """
        elements = []
        self._inorder_recursive(self.root, elements)
        return elements

    def _inorder_recursive(self, node, elements):
        """
        Recursive helper for inorder traversal.
        Left -> Root -> Right
        """
        if node:
            self._inorder_recursive(node.left, elements)
            elements.append(node.value)
            self._inorder_recursive(node.right, elements)
def test_bst():
    bst = BST()

    # Insert student roll numbers
    bst.insert(50)
    bst.insert(30)
    bst.insert(70)
    bst.insert(20)
    bst.insert(40)
    bst.insert(60)
    bst.insert(80)

    print("Inorder Traversal (Sorted Roll Numbers):")
    print(bst.inorder_traversal())

    print("Search 40:", bst.search(40))  # True
    print("Search 100:", bst.search(100))  # False


test_bst()
```

Output:



Why BST Improves Search Efficiency

- **Linear Search (List)**
  - **Checks every element one by one.**
  - **Time Complexity: O(n)**

If there are 10,000 student records, worst case → 10,000 comparisons.

Binary Search Tree

Because of ordering:

- **Compare with root.**
- **If smaller → go left.**
- **If larger → go right.**
- **Each step eliminates half the tree (in balanced case).**

Performance Analysis

Case	Time Complexity
Best Case (Balanced Tree)	O(log n)
Average Case	O(log n)
Worst Case (Skewed Tree)	O(n)

✅ **Best Case (Balanced)**

If tree is balanced:

- **Height ≈ $\log_2(n)$**
- **Very fast lookup.**

Example:
10,000 records → ~14 comparisons only.

❌ **Worst Case (Skewed Tree)**

If values inserted in sorted order:

- **Tree becomes like a linked list.**
- **Search becomes O(n).**

Justification:

A Binary Search Tree (BST) is suitable for student record lookup because it stores roll numbers in a sorted hierarchical structure. This allows the system to eliminate half of the remaining records at each comparison, making search operations faster than linear search**.**

Task 5: Graph Traversal for Social Network Connections

Why Use a Graph?

A social network naturally forms a graph:

- **Each user → Vertex (Node)**
- **Each friendship → Edge (Connection)**

We use an adjacency list representation because:

- **Efficient for sparse graphs (like social networks)**
- **Easy to store and traverse connections**

factorila.py > ...

```python
from collections import deque

class Graph:
    """
    Graph implementation using adjacency list.
    Each user maps to a list of connected friends.
    """

    def __init__(self):
        self.adj_list = {}

    def add_user(self, user):
        """Add a new user to the network."""
        if user not in self.adj_list:
            self.adj_list[user] = []

    def add_connection(self, user1, user2):
        """Create a bidirectional friendship."""
        self.adj_list[user1].append(user2)
        self.adj_list[user2].append(user1)

    # ---------------- BFS ----------------
    def bfs(self, start):
        """
        Breadth-First Search traversal.
        Explores level by level (nearby friends first).
        """
        visited = set()                 # Track visited users
        queue = deque([start])          # Queue for BFS (FIFO)
        visited.add(start)

        result = []

        while queue:
            user = queue.popleft()      # Remove from front
            result.append(user)
```

```
            # Visit all unvisited neighbors
            for friend in self.adj_list[user]:
                if friend not in visited:
                    visited.add(friend)
                    queue.append(friend)

        return result

    # ---------------- DFS (Recursive) ----------------
    def dfs_recursive(self, start, visited=None, result=None):
        """
        Depth-First Search using recursion.
        Explores as deep as possible before backtracking.
        """
        if visited is None:
            visited = set()
            result = []

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

        # Visit each unvisited neighbor deeply
        for friend in self.adj_list[start]:
            if friend not in visited:
                self.dfs_recursive(friend, visited, result)

        return result

    # ---------------- DFS (Iterative) ----------------
    def dfs_iterative(self, start):
        """
        Depth-First Search using a stack (iterative approach).
        """
```

```
        visited = set()
        stack = [start]     # Stack for DFS (LIFO)
        result = []

        while stack:
            user = stack.pop()   # Remove last element
            if user not in visited:
                visited.add(user)
                result.append(user)

                # Add neighbors to stack
                # Reverse to maintain similar order as recursive
                for friend in reversed(self.adj_list[user]):
                    if friend not in visited:
                        stack.append(friend)

        return result
def test_social_network():
    network = Graph()

    users = ["A", "B", "C", "D", "E"]
    for user in users:
        network.add_user(user)

    network.add_connection("A", "B")
    network.add_connection("A", "C")
    network.add_connection("B", "D")
    network.add_connection("C", "E")

    print("BFS from A:", network.bfs("A"))
    print("DFS Recursive from A:", network.dfs_recursive("A"))
    print("DFS Iterative from A:", network.dfs_iterative("A"))
```

Output:

```
PS C:\Users\ramya\OneDrive\Desktop\coding> & C:\Users\ramya\AppData\Local\Microsoft\WindowsApps\python3.11.exe
 c:/Users/ramya/OneDrive/Desktop/coding/factorila.py
BFS from A: ['A', 'B', 'C', 'D', 'E']
DFS Recursive from A: ['A', 'B', 'D', 'C', 'E']
DFS Iterative from A: ['A', 'B', 'D', 'C', 'E']
PS C:\Users\ramya\OneDrive\Desktop\coding>
```

Traversal Logic Explained

- **BFS (Breadth-First Search)**

- **Uses a queue (FIFO).**
- **Visits:**
  - **First → direct friends**
  - **Then → friends of friends**
- **Explores level by level.**

🔹 **DFS (Depth-First Search)**

- **Uses a stack (LIFO) or recursion.**
- **Goes deep along one path before backtracking.**
- **Explores branch fully before moving to next.**

- # Recursive vs Iterative DFS

I. Feature	Recursive DFS	Iterative DFS
II. Uses	System call stack	Explicit stack
III. Code Simplicity	Cleaner and shorter	More control
IV. Risk	Stack overflow (deep graph)	Safer for large graphs
V. Memory Control	Less direct control	Full control

Practical Use Cases

✅ **When to Use BFS**

- Finding **shortest path** in unweighted graph
- Finding nearby friends (1st, 2nd level connections)
- Social network friend suggestions

✅ **When to Use DFS**

- Exploring all possible connection paths
- Detecting cycles
- Finding connected components
- Path existence checking

Justification:

A graph is suitable for modeling a social network because users can be represented as nodes and friendships as edges. This structure naturally represents connections between people.

BFS is useful for finding nearby connections (like mutual friends or shortest path), while DFS is useful for exploring deep connection

paths and checking overall connectivity. Together, they help efficiently analyze relationships within the network.