# AI ASSISSTANT CODING LAB-11.4

**Name:M.Shashidhar Batch:41 Rollno:2303a52291**

**Task 1: Stack for Undo Operations (LIFO)**

**Prompt Used**

**Implement a Stack class in Python with push, pop, peek, and is_empty methods. Add proper docstrings, explain why Stack is suitable for undo functionality, and compare list-based and deque-based implementations.**

**Code**

43.py > ...

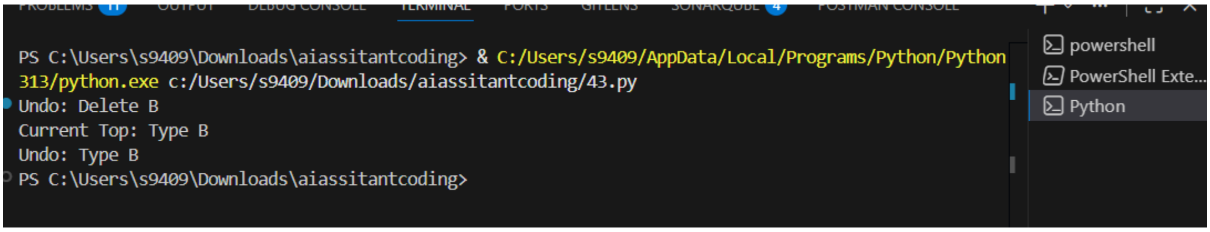
```python
class Stack:
    """
    Stack implementation using Python list.
    Follows LIFO (Last-In-First-Out) principle.
    Suitable for undo operations.
    """

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

    def push(self, action):
        """
        Push an action onto the stack.
        Time Complexity: O(1)
        """
        self.items.append(action)

    def pop(self):
        """
        Remove and return the most recent action.
        Time Complexity: O(1)
        """
        if not self.is_empty():
            return self.items.pop()
```

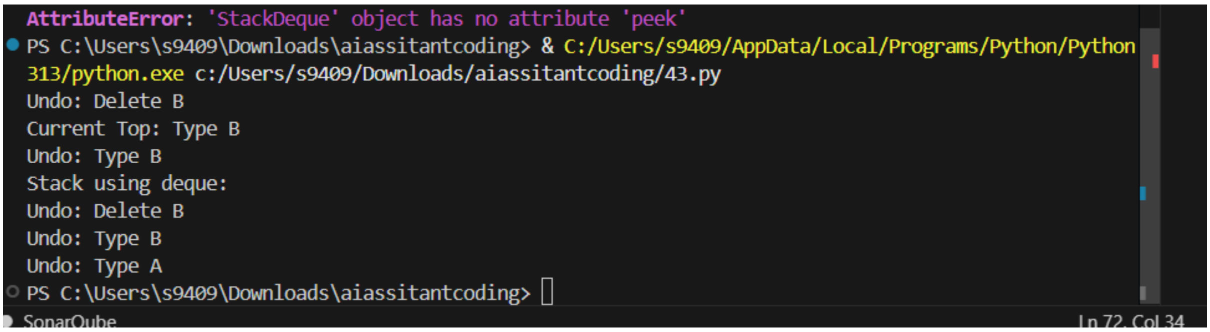
```python
class Stack:
    def pop(self):
            return self.items.pop()
        return None

    def peek(self):
        """
        Return the top element without removing it.
        Time Complexity: O(1)
        """
        if not self.is_empty():
            return self.items[-1]
        return None

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


# Test Case: Simulating Undo Operations

stack = Stack()

stack.push("Type A")
stack.push("Type B")
stack.push("Delete B")

print("Undo:", stack.pop())
print("Current Top:", stack.peek())
print("Undo:", stack.pop())
```

## Optimized

```python
stack.push("Delete B")

print("Undo:", stack.pop())
print("Current Top:", stack.peek())
print("Undo:", stack.pop())
from collections import deque

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

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

    def pop(self):
        if self.items:
            return self.items.pop()
        return None
print("Stack using deque:")
stack_deque = StackDeque()
stack_deque.push("Type A")
stack_deque.push("Type B")
stack_deque.push("Delete B")
```

## Output:




**Justification**

Why Stack is Suitable for Undo

Undo operations must reverse the most recent action first. This matches LIFO behavior:

- Latest action → undone first
- Earlier actions → undone later

Example:

1. Type A
2. Type B
3. Delete B
   Undo → Delete B undone first

Stack provides O(1) push and pop, making it ideal for frequent undo operations.

**Task 2: Queue for Customer Service (FIFO)**

**Prompt Used**

Implement a Queue using list, analyze performance, and optimize using deque.

Code

```python
.py > ...
class QueueList:
    """
    Queue implementation using Python list.
    Follows FIFO (First-In-First-Out).
    """

    def __init__(self):
        self.queue = []

    def enqueue(self, request):
        """
        Add request to end of queue.
        Time Complexity: O(1)
        """
        self.queue.append(request)

    def dequeue(self):
        """
        Remove first request.
        Time Complexity: O(n) due to shifting.
        """
        if not self.is_empty():
            return self.queue.pop(0)
        return None

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


# Test Case
q1 = QueueList()
q1.enqueue("Customer1")
q1.enqueue("Customer2")
print("Serving:", q1.dequeue())    Define a constant instead of duplicating this literal "Serving:" 4 times. [+3 locations]
print("Serving:", q1.dequeue())
print("Serving:", q1.dequeue())  # Should return None (empty queue)
print("Is queue empty?", q1.is_empty())
print("Enqueueing Customer3")
q1.enqueue("Customer3")
print("Serving:", q1.dequeue())
print("Is queue empty?", q1.is_empty())
```

Optimized

```python
from collections import deque

class QueueDeque:
    """
    Optimized queue implementation using collections.deque.
    """

    def __init__(self):
        self.queue = deque()

    def enqueue(self, request):
        self.queue.append(request)  # O(1)

    def dequeue(self):
        if not self.is_empty():
            return self.queue.popleft()  # O(1)
        return None

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


# Test Case
q2 = QueueDeque()
q2.enqueue("Customer1")
q2.enqueue("Customer2")
print("Serving:", q2.dequeue())
```

Output:

```
PS C:\Users\s9409\Downloads\aiassitantcoding> & C:/Users/s9409/AppData/Local/Programs/Python/Python313/python.exe c:/Users/s9409/Downloads/aiassitantcoding/44.py
Serving: Customer1
Serving: Customer2
Serving: None
Is queue empty? True
Enqueueing Customer3
Serving: Customer3
Is queue empty? True
PS C:\Users\s9409\Downloads\aiassitantcoding>
```

```
PS C:\Users\s9409\Downloads\aiassitantcoding> & C:/Users/s9409/AppData/Local/Programs/Python/Python313/python.exe c:/Users/s9409/Downloads/aiassitantcoding/44.py
Serving: Customer1
PS C:\Users\s9409\Downloads\aiassitantcoding>
```

Python

**Performance Analysis**

**List-Based Queue**

- enqueue → O(1)
- dequeue → O(n) (shifts elements)

For 10,000 requests, dequeue becomes expensive.

**Deque-Based Queue**

- enqueue → O(1)
- dequeue → O(1)

Deque uses a doubly-linked structure internally, avoiding element shifting.

**Real-World Impact**

In high-volume systems (banking, server queues, chat systems), list-based queues degrade performance significantly. Deque ensures stable O(1) operations.

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()

prompt used

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)

Code

```python
class Node:
    """
    Node class representing each song in playlist.
    """
    def __init__(self, song):
        self.song = song
        self.next = None  # Pointer to next node


class SinglyLinkedList:
    """
    Singly Linked List for playlist management.
    """

    def __init__(self):
        self.head = None

    def insert_at_end(self, song):
        """
        Insert song at end of list.
        """
        new_node = Node(song)

        # Case 1: Empty list
        if self.head is None:
            self.head = new_node
            return

        # Traverse until last node
        current = self.head
        while current.next:
            current = current.next

        # Update last node pointer
        current.next = new_node

    def delete_value(self, song):
        """
        Delete first occurrence of song.
        """
        current = self.head
```

```python
    def delete_value(self, song):
        if current and current.song == song:
            self.head = current.next
            return

        prev = None
        while current and current.song != song:
            prev = current
            current = current.next

        # Case 2: Song not found
        if current is None:
            return

        # Bypass node (pointer update)
        prev.next = current.next

    def traverse(self):
        """
        Traverse and print playlist.
        """
        current = self.head
        while current:
            print(current.song, end=" -> ")
            current = current.next
        print("None")
print("Playlist using Singly Linked List:")
playlist = SinglyLinkedList()
playlist.insert_at_end("Song A")
playlist.insert_at_end("Song B")    Define a constant instead of duplicating this literal "Song B" 4 times. [+3 locations]
playlist.insert_at_end("Song C")
playlist.traverse()
playlist.delete_value("Song B")
playlist.traverse()
print("Playlist using built-in list:")
playlist_list = []
playlist_list.append("Song A")
playlist_list.append("Song B")
print("Playlist:", playlist_list)
playlist_list.remove("Song B")
print("Playlist after deletion:", playlist_list)
```

Output:

```
PS C:\Users\s9409\Downloads\aiassitantcoding> & C:/Users/s9409/AppData/Local/Programs/Python/Python
313/python.exe c:/Users/s9409/Downloads/aiassitantcoding/45.py
Playlist using Singly Linked List:
Song A -> Song B -> Song C -> None
Song A -> Song C -> None
Playlist using built-in list:
Playlist: ['Song A', 'Song B']
Playlist after deletion: ['Song A']
PS C:\Users\s9409\Downloads\aiassitantcoding>
```

**Edge Cases**

- Empty list insertion
- Delete from empty list
- Delete head node
- Delete single node list
- Delete non-existent value

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.

Prompt used :

Complete missing methods,Add meaningful docstrings, Explain how BST improves search efficiency over linear search

Code

```python
class Node:
    """
    Node of Binary Search Tree.
    """
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None


class BST:
    """
    Binary Search Tree implementation.
    """

    def __init__(self):
        self.root = None

    def insert(self, value):
        """
        Insert value while maintaining BST property.
        """

        def _insert(node, value):
            if node is None:
                return Node(value)

            if value < node.value:
                node.left = _insert(node.left, value)
            else:
                node.right = _insert(node.right, value)
            return node
```

```python
            self.root = _insert(self.root, value)

    def search(self, value):
        """
        Search value in BST.
        """
        current = self.root

        while current:
            if value == current.value:
                return True
            elif value < current.value:
                current = current.left
            else:
                current = current.right

        return False

    def inorder_traversal(self):
        """
        Print values in sorted order.
        """

        def _inorder(node):
            if node:
                _inorder(node.left)
                print(node.value, end=" ")
                _inorder(node.right)

        _inorder(self.root)
print("BST Example:")
bst = BST()
bst.insert(5)
bst.insert(3)
bst.insert(7)
print("Inorder Traversal:", end=" ")
bst.inorder_traversal()
print("\nSearch for 3:", bst.search(3))
print("Search for 10:", bst.search(10))
```

Output:

```
PS C:\Users\s9409\Downloads\aiassitantcoding> & C:/Users/s9409/AppData/Local/Programs/Python/Python
313/python.exe c:/Users/s9409/Downloads/aiassitantcoding/46.py
BST Example:
Inorder Traversal: 3 5 7
Search for 3: True
Search for 10: False
PS C:\Users\s9409\Downloads\aiassitantcoding>
```

**Why BST Improves Search Efficiency**

| **Approach** | **Time Complexity** |
|---|---|
| Linear Search | O(n) |
| BST (Average) | O(log n) |
| BST (Worst - Skewed) | O(n) |

BST reduces search space by half at each step in balanced cases.

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

prompt used:

Add inline comments explaining traversal steps, Compare recursive and iterative DFS approaches,Suggest practical use cases for BFS vs DFS

```python
from collections import deque

class Graph:
    """
    Graph using adjacency list representation.
    """

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

    def add_edge(self, u, v):
        """
        Add undirected edge between u and v.
        """
        self.graph.setdefault(u, []).append(v)
        self.graph.setdefault(v, []).append(u)

    def bfs(self, start):
        """
        Breadth-First Search traversal.
        Uses queue.
        """
        visited = set()
        queue = deque([start])
        visited.add(start)

        while queue:
            vertex = queue.popleft()
            print(vertex, end=" ")

            for neighbor in self.graph.get(vertex, []):
```

```python
class Graph:
    def bfs(self, start):
                if neighbor not in visited:
                    visited.add(neighbor)
                    queue.append(neighbor)

    def dfs(self, start, visited=None):
        """
        Depth-First Search traversal (recursive).
        """
        if visited is None:
            visited = set()

        visited.add(start)
        print(start, end=" ")

        for neighbor in self.graph.get(start, []):
            if neighbor not in visited:
                self.dfs(neighbor, visited)
print("Graph Example:")
g = Graph()
g.add_edge(1, 2)
g.add_edge(1, 3)
g.add_edge(2, 4)
print("BFS starting from vertex 1:")
g.bfs(1)
print("\nDFS starting from vertex 1:")
g.dfs(1)
print()
```

Output:

```
PS C:\Users\s9409\Downloads\aiassitantcoding> & C:/Users/s9409/AppData/Local/Programs/Python/Python313/python.exe c:/Users/s9409/Downloads/aiassitantcoding/47.py
Graph Example:
BFS starting from vertex 1:
1 2 3 4
DFS starting from vertex 1:
1 2 4 3
PS C:\Users\s9409\Downloads\aiassitantcoding>
```

**BFS vs DFS Comparison**

| Feature | BFS | DFS |
|---|---|---|
| Structure | Queue | Stack/Recursion |
| Traversal | Level-by-level | Depth-first |
| Shortest Path | Yes (unweighted) | No |
| Memory | Higher (wide graphs) | Lower (deep graphs) |

**Practical Use Cases**

BFS:

- Shortest path

- Social network degree search
- Broadcasting systems

DFS:

- Cycle detection
- Path existence
- Topological