AI ASSIGNMENT LAB-11

ROLLNO:2403A52397.

NAME: Varshitha.k

Task 1: Implementing a Stack (LIFO)

- **Task**: 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).

```python
class Stack:

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

    def push(self, item):

        self.items.append(item)

    def pop(self):

        if self.is_empty():
            raise IndexError("Pop from empty stack")
        return self.items.pop()

    def peek(self):

        if self.is_empty():
            raise IndexError("Peek from empty stack")
        return self.items[-1]

    def is_empty(self):

        return len(self.items) == 0

    def __str__(self):
        """Return a readable string for the stack."""
        return f"Stack(top → bottom): {list(reversed(self.items))}"


stack = Stack()
print("Is stack empty?", stack.is_empty())

stack.push(10)
stack.push(20)
stack.push(30)
print("After pushing:", stack)

print("Peek:", stack.peek())

print("Pop:", stack.pop())
print("After pop:", stack)
```

```python
stack.push(30)
print("After pushing:", stack)

print("Peek:", stack.peek())

print("Pop:", stack.pop())
print("After pop:", stack)

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


from collections import deque

class StackDeque:
    """Optimized stack using deque for O(1) performance."""
    def __init__(self):
        self.items = deque()

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

    def pop(self):
        if not self.items:
            raise IndexError("Pop from empty stack")
        return self.items.pop()

    def peek(self):
        if not self.items:
            raise IndexError("Peek from empty stack")
        return self.items[-1]

    def is_empty(self):
        return not self.items


print("\n--- Using deque-based Stack ---")
sd = StackDeque()
sd.push('A')
sd.push('B')
sd.push('C')
print("Top element:", sd.peek())
print("Popped:", sd.pop())
print("Is empty?", sd.is_empty())
```

```
Is stack empty? True
After pushing: Stack(top → bottom): [30, 20, 10]
Peek: 30
Pop: 30
After pop: Stack(top → bottom): [20, 10]
Is stack empty? False

--- Using deque-based Stack ---
Top element: C
Popped: C
Is empty? False
```

Task 2: Queue Implementation with Performance Review

- **Task**: 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)

```python
from collections import deque

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

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

    def dequeue(self):
        if self.is_empty():
            raise IndexError("Dequeue from empty queue")
        return self.items.popleft()

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

    def __str__(self):
        return f"Queue(front → rear): {list(self.items)}"


queue = QueueDeque()
print("Is queue empty?", queue.is_empty())

queue.enqueue("A")
queue.enqueue("B")
queue.enqueue("C")
print("After enqueue:", queue)

print("Dequeued:", queue.dequeue())
print("After one dequeue:", queue)

print("Is queue empty?", queue.is_empty())
```

```
Is queue empty? True
After enqueue: Queue(front → rear): ['A', 'B', 'C']
Dequeued: A
After one dequeue: Queue(front → rear): ['B', 'C']
Is queue empty? False
```

Task 3: Singly Linked List with Traversal

- **Task**: Implement a **Singly Linked List** with operations: insert_at_end(), 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

Here are some additional test cases to consider:

- Test `insert_at_end` **on an empty list:** Verify that the new node becomes the head.
- Test `insert_at_end` **on a non-empty list:** Verify that the new node is added at the very end.
- Test `delete_value` **when the value is in the middle of the list:** Ensure the correct node is removed and the links are updated.
- Test `delete_value` **when the value is the only element in the list:** Verify that the list becomes empty.
- Test `delete_value` **when the list has duplicate values:** Confirm that only the first occurrence is deleted.
- Test `delete_value` **multiple times:** Check the behavior after several deletions.
- Test `traverse` **on an empty list:** Ensure it prints "None".
- Test `traverse` **after insertions and deletions:** Verify the correct sequence of elements is printed.

```
class Node:
    """Represents a node in a singly linked list."""
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    """Represents a singly linked list."""
    def __init__(self):
        self.head = None # Initialize head as None for an empty list

    def insert_at_end(self, data):
        """Inserts a new node with the given data at the end of the list."""
        new_node = Node(data)
        if self.head is None:
            self.head = new_node # If list is empty, the new node becomes the head
            return
        last_node = self.head
        while last_node.next:
            last_node = last_node.next # Traverse to the last node
        last_node.next = new_node # Link the last node to the new node

    def delete_value(self, value):
        """Deletes the first node with the given value from the list."""
        if self.head is None:
            return # Return if the list is empty

        if self.head.data == value:
            self.head = self.head.next # If head is the node to delete, update head
            return

        current_node = self.head
        while current_node.next and current_node.next.data != value:
            current_node = current_node.next # Traverse to the node before the one to delete

        if current_node.next:
            current_node.next = current_node.next.next # Skip the node to be deleted by linking the previous node to the next node
```

Let's test the implemented Singly Linked List:

```python
# Create a new linked list
my_list = LinkedList()

# Insert elements at the end
my_list.insert_at_end(10)
my_list.insert_at_end(20)
my_list.insert_at_end(30)
my_list.insert_at_end(40)

# Traverse the list
print("Initial list:")
my_list.traverse()

# Delete a value
print("\nDeleting value 20:")
my_list.delete_value(20)
my_list.traverse()

# Delete a non-existent value
print("\nDeleting value 50:")
my_list.delete_value(50)
my_list.traverse()

# Delete the head
print("\nDeleting value 10:")
my_list.delete_value(10)
my_list.traverse()

# Delete the last element
print("\nDeleting value 40:")
my_list.delete_value(40)
my_list.traverse()

# Delete from an empty list
print("\nDeleting from empty list:")
my_list.delete_value(30)
my_list.traverse()
```

```
Initial list:
10 -> 20 -> 30 -> 40 -> None

Deleting value 20:
10 -> 30 -> 40 -> None

Deleting value 50:
10 -> 30 -> 40 -> None

Deleting value 10:
30 -> 40 -> None

Deleting value 40:
30 -> None

Deleting from empty list:
None
```

Task 4: Binary Search Tree (BST)

- **Task**: 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.

```
class Node:

    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None


class BST:

    def __init__(self):
        self.root = None

    def insert(self, data):

        if self.root is None:
            self.root = Node(data)
        else:
            self._insert_recursive(self.root, data)

    def _insert_recursive(self, current_node, data):

        if data < current_node.data:
            if current_node.left is None:
                current_node.left = Node(data)
            else:
                self._insert_recursive(current_node.left, data)
        elif data > current_node.data:
            if current_node.right is None:
                current_node.right = Node(data)
            else:
                self._insert_recursive(current_node.right, data)
        # Duplicate data is ignored

    def search(self, data):

        return self._search_recursive(self.root, data)

    def _search_recursive(self, current_node, data):

        if current_node is None:
            return False
        if data == current_node.data:
            return True
        elif data < current_node.data:
            return self._search_recursive(current_node.left, data)
        else:
            return self._search_recursive(current_node.right, data)

    def inorder_traversal(self):

        self._inorder_recursive(self.root)
        print() # Print a newline at the end for cleaner output

    def _inorder_recursive(self, current_node):

        if current_node:
            self._inorder_recursive(current_node.left)
            print(current_node.data, end=" ")
            self._inorder_recursive(current_node.right)
```

```python
# Create a list of integers
integer_list = [50, 30, 70, 20, 40, 60, 80, 10, 90]

# Instantiate the BST class
bst = BST()

# Insert the integers into the BST
for number in integer_list:
    bst.insert(number)
```

```python
# Test search for a value present in the tree
search_value_present = 40
found_present = bst.search(search_value_present)
print(f"Is {search_value_present} present in the BST? {found_present}")

# Test search for a value absent from the tree
search_value_absent = 99
found_absent = bst.search(search_value_absent)
print(f"Is {search_value_absent} present in the BST? {found_absent}")
```

```
Is 40 present in the BST? True
Is 99 present in the BST? False
```

```python
print("\nInorder traversal of the BST:")
bst.inorder_traversal()
```

```

Inorder traversal of the BST:
10 20 30 40 50 60 70 80 90
```

Task 5: Graph Representation and BFS/DFS Traversal

- **Task**: 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.

```python
        # Check if the start_node exists in the graph
        if start_node not in self.graph:
            print(f"Node {start_node} not found in the graph.")
            return

        # Initialize a set to keep track of visited nodes
        visited = set()

        # Start the recursive DFS traversal
        self._dfs_recursive(start_node, visited)
        print() # Print a newline at the end for cleaner output


    def _dfs_recursive(self, current_node, visited):
        """
        Recursive helper function for Depth-First Search traversal.

        Args:
            current_node: The current node being visited.
            visited: A set of visited nodes.
        """
        # Mark the current node as visited
        visited.add(current_node)

        # Print the current node (order of traversal)
        print(current_node, end=" ")

        # Iterate through the neighbors of the current node
        # Check if current_node exists in the graph before iterating over its neighbors
        if current_node in self.graph:
            for neighbor in self.graph[current_node]:
                # If the neighbor has not been visited
                if neighbor not in visited:
                    # Recursively call DFS on the neighbor
                    self._dfs_recursive(neighbor, visited)
```

st graph and traversals

btask:

ate a sample graph, and then test the `bfs()` and `dfs()` methods to demonstrate their functionality.

asoning: Instantiate the Graph class, add edges to create a sample graph, and then call the bfs() and dfs() methods with a starting de, printing descriptive messages before the output.

```python
# Instantiate the Graph class
graph = Graph()

# Add edges to create a sample graph
graph.add_edge("A", "B")
graph.add_edge("A", "C")
graph.add_edge("B", "D")
graph.add_edge("C", "E")
graph.add_edge("D", "E")
graph.add_edge("D", "F")
graph.add_edge("E", "F")

# Call the bfs() method
print("BFS traversal starting from node A:")
graph.bfs("A")

# Call the dfs() method
print("\nDFS traversal starting from node A:")
graph.dfs("A")
```

```
BFS traversal starting from node A:
A B C D E F

DFS traversal starting from node A:
A B D E C F
```