AI ASSISTED CODING
ASSIGNMENT 11.4

S.SIRI
2403A51236
BATCH-11

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:

o Ask AI to generate code skeleton with docstrings.

o Test stack operations using sample data.

o Request AI to suggest optimizations or alternative implementations (e.g., using collections.deque).

• Expected Output:

o A working Stack class with proper methods, Google-style docstrings, and inline comments for tricky parts

Prompt:

TASK: Implement a Stack class with push, pop, peek, is_empty methods.

Include Google-style docstrings, inline comments for tricky parts, and test the stack operations with sample data.

TASK: Rewrite the Stack class using collections.deque instead of a list.

Keep the same methods (push, pop, peek, is_empty) and include docstrings.

Explain why deque might be better than a list in this scenario.

Code and output:

task11.1.py > Stack > push

```python
from collections import deque

class Stack:
    """A simple stack implementation using collections.deque.

    Methods:
        push(item): Add an item to the top of the stack.
        pop(): Remove and return the top item from the stack.
        peek(): Return the top item without removing it.
        is_empty(): Return True if the stack is empty, False otherwise.
    """

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

    def push(self, item):
        """Add an item to the top of the stack.

        Args:
            item: The item to be added.
        """
        self._items.append(item)

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

        Returns:
            The item at the top of the stack.

        Raises:
            IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("pop from empty stack")
        return self._items.pop()

```

Ln 19, Col 1 Spaces: 4 UTF-8

```python
class Stack:
    def peek(self):
        """Return the top item without removing it.

        Returns:
            The item at the top of the stack.

        Raises:
            IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("peek from empty stack")
        return self._items[-1]

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

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

# --- Sample usage and tests ---

if __name__ == "__main__":
    stack = Stack()
    print("Is stack empty?", stack.is_empty())  # True

    stack.push(10)
    stack.push(20)
    stack.push(30)
    print("Peek:", stack.peek())  # 30

    print("Pop:", stack.pop())    # 30
    print("Pop:", stack.pop())    # 20

```

```python

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

    print("Pop:", stack.pop())    # 10

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

    # Uncomment to see error handling:
    # stack.pop()  # Raises IndexError

# Explanation:
# Using collections.deque is generally better than a list for stack operations
# because deque provides O(1) time complexity for append() and pop() operations
# at both ends, while list's pop() from the end is O(1) but can be less efficient
# due to resizing and memory reallocation. Deque is optimized for fast fixed-time
# appends and pops, making it more suitable for stack and queue implementations.
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\siris\New folder (2)> & C:/Users/siris/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/siris/New folder (2)/task11.1.py"

Is stack empty? True
Peek: 30
Pop: 30
Pop: 20
Is stack empty? False
Pop: 10
Is stack empty? True
PS C:\Users\siris\New folder (2)>
```

Task 2: Queue Implementation with Performance Review

• Task: Implement a Queue with enqueue(), dequeue(), and is_empty() methods.

• Instructions:

o First, implement using Python lists.

o Then, ask AI to review performance and suggest a more efficient implementation (using collections.deque).

• Expected Output:

o Two versions of a queue: one with lists and one optimized with deque, plus an AI-generated performance comparison

Code:

task11.2.py > ...

```python
class QueueList:
    """Queue implementation using a Python list."""

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

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

    def dequeue(self):
        """Remove and return the item from the front of the queue."""
        if self.is_empty():
            raise IndexError("Dequeue from empty queue")
        return self.items.pop(0)  # O(n) operation

    def is_empty(self):
        """Check if the queue is empty."""
        return len(self.items) == 0


# Quick test
q1 = QueueList()
q1.enqueue(10)
q1.enqueue(20)
print(q1.dequeue())  # Should print 10
print(q1.is_empty())  # Should be False
```

Prompt:

TASK: Review the QueueList implementation above.

Point out any performance issues with using list.pop(0) for dequeue.

Suggest a more efficient implementation using collections.deque.

Provide code for the optimized version.

Output code:

task11.2.py > ...

```python
from collections import deque


class QueueDeque:
    """Queue implementation using collections.deque."""

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

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

    def dequeue(self):
        """Remove and return the item from the front of the queue."""
        if self.is_empty():
            raise IndexError("Dequeue from empty queue")
        return self.items.popleft()  # O(1) operation

    def is_empty(self):
        """Check if the queue is empty."""
        return len(self.items) == 0


# Quick test
q2 = QueueDeque()
q2.enqueue(10)
q2.enqueue(20)
print(q2.dequeue())  # Should print 10
print(q2.is_empty())  # Should be False
```

Task 3: Singly Linked List with Traversal

• Task: Implement a Singly Linked List with operations:
insert_at_end(), delete_value(), and traverse().

• Instructions:

o Start with a simple class-based implementation (Node, LinkedList).

o Use AI to generate inline comments explaining pointer updates (which are non-trivial).

o Ask AI to suggest test cases to validate all operations.

• Expected Output:

o A functional linked list implementation with clear comments explaining the logic of insertions and deletions.

Prompts:

TASK: Add detailed inline comments explaining pointer changes especially for insert_at_end() and delete_value().

TASK: Suggest comprehensive test cases to validate:

1. Insertion into empty list

2. Insertion into non-empty list

3. Deleting head node
4. Deleting middle node
5. Deleting last node
6. Attempting to delete non-existent value
7. Traversing empty list

Final code and output:

task11.3.py > LinkedList > insert_at_end

```python
class Node:
    """A node of a singly linked list."""

    def __init__(self, data):
        """Initialize a node with data and no next reference."""
        self.data = data
        self.next = None  # Pointer to the next node


class LinkedList:
    """Singly linked list implementation."""

    def __init__(self):
        """Initialize an empty linked list."""
        self.head = None

    def insert_at_end(self, data):
        """Insert a new node with given data at the end of the list."""
        new_node = Node(data)
        # If list is empty, new node becomes head
        if self.head is None:
            self.head = new_node  # Head now points to the new node
            return
        # Otherwise, traverse to the last node
        current = self.head
        while current.next:
            current = current.next  # Move to next node until reaching the end
        # Point the last node's next to the new node
        current.next = new_node  # Last node now points to the new node

    def delete_value(self, value):
        """Delete the first occurrence of 'value' from the list."""
        current = self.head
        prev = None
        # If the head itself holds the value
        if current and current.data == value:
            self.head = current.next  # Move head to next node, old head is removed
```

```python
    def delete_value(self, value):
            self.head = current.next  # Move head to next node, old head is removed
            return True
        # Traverse the list to find the value
        while current and current.data != value:
            prev = current           # Keep track of previous node
            current = current.next   # Move to next node
        # If value not found
        if current is None:
            return False
        # Unlink the node from the linked list
        prev.next = current.next     # Previous node skips the current node
        # Now, current node is removed from the list (no references to it)
        return True

    def traverse(self):
        """Traverse the list and return all elements as a Python list."""
        elements = []
        current = self.head
        while current:
            elements.append(current.data)
            current = current.next
        return elements


# Quick test
ll = LinkedList()
ll.insert_at_end(10)
ll.insert_at_end(20)
ll.insert_at_end(30)
print(ll.traverse())  # [10, 20, 30]
ll.delete_value(20)
print(ll.traverse())  # [10, 30]

def test_linked_list():
    ll = LinkedList()
```

```python

    # 1. Insertion into empty list
    ll.insert_at_end(1)
    assert ll.traverse() == [1], "Failed: Insert into empty list"

    # 2. Insertion into non-empty list
    ll.insert_at_end(2)
    ll.insert_at_end(3)
    assert ll.traverse() == [1, 2, 3], "Failed: Insert into non-empty list"

    # 3. Deleting head node
    assert ll.delete_value(1) is True, "Failed: Delete head node"
    assert ll.traverse() == [2, 3], "Failed: List after deleting head"

    # 4. Deleting middle node
    ll.insert_at_end(4)
    assert ll.delete_value(3) is True, "Failed: Delete middle node"
    assert ll.traverse() == [2, 4], "Failed: List after deleting middle node"

    # 5. Deleting last node
    assert ll.delete_value(4) is True, "Failed: Delete last node"
    assert ll.traverse() == [2], "Failed: List after deleting last node"

    # 6. Deleting from a single-element list
    assert ll.delete_value(2) is True, "Failed: Delete from single-element list"
    assert ll.traverse() == [], "Failed: List should be empty now"

    # 7. Deleting non-existent value
    assert ll.delete_value(10) is False, "Failed: Delete non-existent value"
    assert ll.traverse() == [], "Failed: List should still be empty"

    print("All tests passed!")


test_linked_list()

```

```
PS C:\Users\siris\New folder (2)> & C:/Users/siris/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/siris/New folder (2)/task11.3.py
"
[10, 20, 30]
[10, 30]
All tests passed!
PS C:\Users\siris\New folder (2)>
```

Task 4: Binary Search Tree (BST)

• Task: Implement a Binary Search Tree with methods for insert(), search(), and inorder_traversal().

• Instructions:

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

o Ask AI to complete missing methods and add docstrings.

o Test with a list of integers and compare outputs of search() for present vs absent elements

prompt:

TASK: Complete the insert(), search(), and inorder_traversal() methods

for the BST class. Add Google-style docstrings for each method. Use recursion where appropriate.

Final code :

task11.4.py > ...

```python
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):
        """Insert a value into the BST.

        Args:
            data: The value to insert.
        """
        def _insert(node, data):
            if node is None:
                return Node(data)
            if data < node.data:
                node.left = _insert(node.left, data)
            elif data > node.data:
                node.right = _insert(node.right, data)
            # If data == node.data, do not insert duplicates
            return node

        self.root = _insert(self.root, data)

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

        Args:
            value: The value to search for.

        Returns:
            True if value is found, False otherwise.
```

task11.4.py > ...

```python
class BST:
    def search(self, value):
            True if value is found, False otherwise.
        """
        def _search(node, value):
            if node is None:
                return False
            if value == node.data:
                return True
            elif value < node.data:
                return _search(node.left, value)
            else:
                return _search(node.right, value)

        return _search(self.root, value)

    def inorder_traversal(self):
        """Perform inorder traversal of the BST.

        Returns:
            A list of values in sorted order.
        """
        def _inorder(node, result):
            if node:
                _inorder(node.left, result)
                result.append(node.data)
                _inorder(node.right, result)

        result = []
        _inorder(self.root, result)
        return result

# Example usage and test
if __name__ == "__main__":
    bst = BST()
    for val in [5, 3, 7, 2, 4, 6, 8]:
```

```python
if __name__ == "__main__":
    bst = BST()
    for val in [5, 3, 7, 2, 4, 6, 8]:
        bst.insert(val)
    print("Inorder:", bst.inorder_traversal())  # [2, 3, 4, 5, 6, 7, 8]
    print("Search 4:", bst.search(4))           # True
    print("Search 10:", bst.search(10))         # False
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\siris\New folder (2)> & C:/Users/siris/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/siris/New folder (2)/task11.4.py"
Inorder: [2, 3, 4, 5, 6, 7, 8]
Search 4: True
Search 10: False
PS C:\Users\siris\New folder (2)>
```

Task 5: Graph Representation and BFS/DFS Traversal

• Task: Implement a Graph using an adjacency list, with traversal methods BFS() and DFS().

• Instructions:

o Start with an adjacency list dictionary.

o Ask AI to generate BFS and DFS implementations with inline comments.

o Compare recursive vs iterative DFS if suggested by AI.

• Expected Output:

o A graph implementation with BFS and DFS traversal methods, with AI-generated comments explaining traversal steps.

Prompts:

TASK 1: Create a Graph class using an adjacency list in Python.

Include methods:

- add_vertex(vertex)
- add_edge(v1, v2) (undirected edges)
- bfs(start) to perform Breadth-First Search
- dfs_iterative(start) to perform Depth-First Search iteratively
- dfs_recursive(start) to perform Depth-First Search recursively

Add Google-style docstrings for each method.

Return traversal order as a list in BFS/DFS.

TASK 2: Add detailed inline comments to the BFS and DFS methods explaining:

- How the queue (BFS) and stack (DFS) are updated at each step
- How visited nodes are tracked

Also generate sample test code:

- Build a small graph (5–6 nodes), Print the BFS and DFS traversals

Finally, explain in plain text the differences between recursive and iterative DFS:

- Memory usage, Recursion limits , Control over traversal order

Code :

task5.py > Graph > add_edge

```python
class Graph:
    """Graph implementation using an adjacency list."""

    def __init__(self):
        """Initialize an empty adjacency list."""
        self.adj = {}

    def add_vertex(self, vertex):
        """Add a vertex to the graph.

        Args:
            vertex: The vertex to add.
        """
        if vertex not in self.adj:
            self.adj[vertex] = []

    def add_edge(self, v1, v2):
        """Add an undirected edge between v1 and v2.

        Args:
            v1: First vertex.
            v2: Second vertex.
        """
        self.add_vertex(v1)
        self.add_vertex(v2)
        if v2 not in self.adj[v1]:
            self.adj[v1].append(v2)
        if v1 not in self.adj[v2]:
            self.adj[v2].append(v1)

    def bfs(self, start):
        """Perform Breadth-First Search (BFS) starting from 'start'.

        Args:
            start: The starting vertex.

        Returns:
```

task5.py > Graph > add_edge

class Graph:
    def bfs(self, start):

```python
        Returns:
            List of vertices in BFS traversal order.
        """
        visited = set()      # Track visited nodes to avoid revisiting
        queue = [start]      # Queue for BFS; FIFO order
        order = []           # List to store traversal order

        while queue:
            vertex = queue.pop(0)  # Dequeue from front
            if vertex not in visited:
                visited.add(vertex)     # Mark as visited
                order.append(vertex)    # Add to traversal order
                # Enqueue all unvisited neighbors
                for n in self.adj.get(vertex, []):
                    if n not in visited:
                        queue.append(n)
                # At this point, queue contains next layer of nodes to visit
        return order

    def dfs_iterative(self, start):
        """Perform iterative Depth-First Search (DFS) starting from 'start'.

        Args:
            start: The starting vertex.

        Returns:
            List of vertices in DFS traversal order.
        """
        visited = set()      # Track visited nodes
        stack = [start]      # Stack for DFS; LIFO order
        order = []           # List to store traversal order

        while stack:
            vertex = stack.pop()  # Pop from top of stack
            if vertex not in visited:
```

Got a minute? Help us ma

Source: GitHub Copilot Chat

task5.py > Graph > add_edge

```python
class Graph:
    def dfs_iterative(self, start):
            if vertex not in visited:
                visited.add(vertex)    # Mark as visited
                order.append(vertex)   # Add to traversal order
                # Push all unvisited neighbors onto the stack (reverse for correct order)
                for n in reversed(self.adj.get(vertex, [])):
                    if n not in visited:
                        stack.append(n)
                # At this point, stack contains next nodes to explore (deepest first)
        return order

    def dfs_recursive(self, start):
        """Perform recursive Depth-First Search (DFS) starting from 'start'.

        Args:
            start: The starting vertex.

        Returns:
            List of vertices in DFS traversal order.
        """
        order = []
        visited = set()

        def _dfs(v):
            visited.add(v)      # Mark as visited
            order.append(v)     # Add to traversal order
            for neighbor in self.adj.get(v, []):
                if neighbor not in visited:
                    _dfs(neighbor)  # Recursively visit unvisited neighbors

        _dfs(start)
        return order

# --- Sample test code ---

if __name__ == "__main__":
```

```python
    #   F
    g = Graph()
    g.add_edge('A', 'B')
    g.add_edge('A', 'C')
    g.add_edge('B', 'D')
    g.add_edge('C', 'D')
    g.add_edge('C', 'E')
    g.add_edge('E', 'F')

    print("BFS from A:", g.bfs('A'))              # Example: ['A', 'B', 'C', 'D', 'E', 'F']
    print("DFS iterative from A:", g.dfs_iterative('A'))  # Example: ['A', 'C', 'E', 'F', 'D', 'B']
    print("DFS recursive from A:", g.dfs_recursive('A'))  # Example: ['A', 'B', 'D', 'C', 'E', 'F']

# --- Explanation: Recursive vs Iterative DFS ---

"""
Recursive DFS:
- Uses the call stack to remember the path, so memory usage depends on recursion depth (can be up to O(V) for V nodes).
- Limited by Python's recursion limit (default ~1000), so very deep or large graphs may cause a RecursionError.
- Traversal order is determined by the order of neighbors and recursion.

Iterative DFS:
- Uses an explicit stack (Python list) to manage traversal, so not limited by recursion depth.
- More control over traversal order (can reverse neighbors, etc.).
- Safer for large/deep graphs, as it avoids recursion limits.
"""
```