AI ASSISTED CODING

HALL TICKET :2403A51270

BATCH:12

ASSIGNMENT :11.4

QUESTION:

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:

"Implement a Stack class in Python with methods push(), pop(), peek(), and is\_empty(). Write Google-style docstrings and inline comments for tricky parts. Provide a simple test case that demonstrates pushing, popping, and peeking. Then, suggest an optimization or alternative implementation using collections.deque."

CODE:

class Stack:

    def \_\_init\_\_(self):

        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():

            return None

        return self.\_items[-1]

    def is\_empty(self):

        """Check if the stack is empty."""

        return len(self.\_items) == 0

# Sample usage:

if \_\_name\_\_ == "\_\_main\_\_":

    stack = Stack()

    stack.push(10)

    stack.push(20)

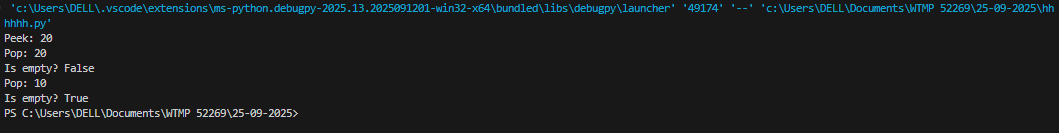
    print("Peek:", stack.peek())

    print("Pop:", stack.pop())

    print("Is empty?", stack.is\_empty())

    print("Pop:", stack.pop())

    print("Is empty?", stack.is\_empty())

OUTPUT: 

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

Prompt:

"Implement a Queue class in Python with methods enqueue(), dequeue(), and is\_empty(). First, implement it using Python lists. Then explain the performance drawbacks. Next, implement an optimized version using collections.deque and provide an AI-generated performance comparison. Include test cases for both versions."

CODE:2.1

class QueueList:

    def \_\_init\_\_(self):

        self.\_items = []

    def enqueue(self, item):

        self.\_items.append(item)

    def dequeue(self):

        if self.is\_empty():

            raise IndexError("Dequeue from empty queue")

        return self.\_items.pop(0)

    def is\_empty(self):

        """Check if the queue is empty."""

        return len(self.\_items) == 0

if \_\_name\_\_ == "\_\_main\_\_":

    q = QueueList()

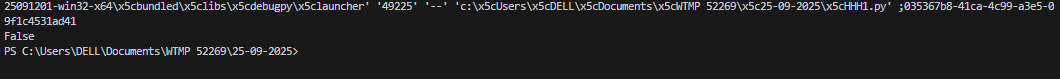
    q.enqueue(1)

    q.enqueue(2)

    print(q.dequeue())

    print(q.is\_empty())

OUTPUT:



CODE 2.2:

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):

        """Check if the queue is empty."""

        return len(self.\_items) == 0

if \_\_name\_\_ == "\_\_main\_\_":

    q = QueueDeque()

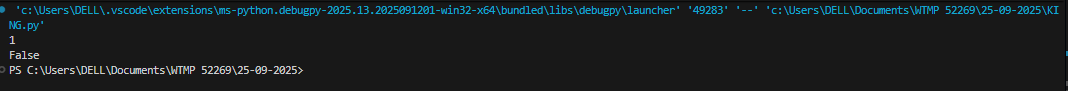
    q.enqueue(1)

    q.enqueue(2)

    print(q.dequeue())  # Output: 1

    print(q.is\_empty())  # Output: False

OUTPUT:



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

Prompt:

"Implement a Singly Linked List in Python with a Node and LinkedList class. Add methods insert\_at\_end(), delete\_value(), and traverse(). Use inline comments to clearly explain pointer updates (since they are tricky). Also, suggest test cases to validate insertions, deletions (head, middle, last node), and traversal."

CODE:

class Node:

    def \_\_init\_\_(self, data):

        self.data = data

        self.next = None

class LinkedList:

    def \_\_init\_\_(self):

        self.head = None

    def insert\_at\_end(self, data):

        new\_node = Node(data)

        if not self.head:

            self.head = new\_node

            return

        current = self.head

        while current.next:

            current = current.next

        current.next = new\_node

    def delete\_value(self, value):

        current = self.head

        prev = None

        while current:

            if current.data == value:

                if prev is None:

                    self.head = current.next

                else:

                    prev.next = current.next

                return True

            prev = current

            current = current.next

        return False

    def traverse(self):

        elements = []

        current = self.head

        while current:

            elements.append(current.data)

            current = current.next

        return elements

if \_\_name\_\_ == "\_\_main\_\_":

    ll = LinkedList()

    ll.insert\_at\_end(10)

    ll.insert\_at\_end(20)

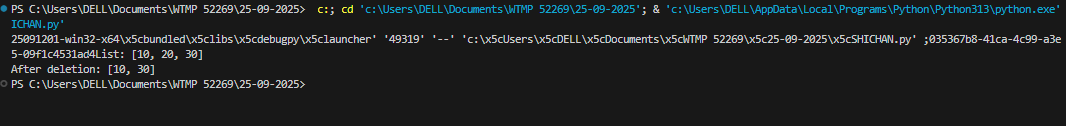
    ll.insert\_at\_end(30)

    print("List:", ll.traverse())

    ll.delete\_value(20)

    print("After deletion:", ll.traverse())

OUTPUT:



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.  
• Expected Output:  
o A BST class with clean implementation, meaningful docstrings,  
and correct traversal output

Prompt:

"Complete a Binary Search Tree implementation in Python. Provide insert(), search(), and inorder\_traversal() methods inside a BST class, with a supporting BSTNode class. Add Google-style docstrings. Show tests using a list of integers, and demonstrate searching for both present and absent elements."

CODE:

class BSTNode:

    def \_\_init\_\_(self, value):

        self.value = value

        self.left = None

        self.right = None

class BinarySearchTree:

    def \_\_init\_\_(self):

        self.root = None

    def insert(self, value):

        if self.root is None:

            self.root = BSTNode(value)

        else:

            self.\_insert\_recursive(self.root, value)

    def \_insert\_recursive(self, node, value):

        if value < node.value:

            if node.left is None:

                node.left = BSTNode(value)

            else:

                self.\_insert\_recursive(node.left, value)

        else:

            if node.right is None:

                node.right = BSTNode(value)

            else:

                self.\_insert\_recursive(node.right, value)

    def search(self, value):

        return self.\_search\_recursive(self.root, value)

    def \_search\_recursive(self, node, value):

        if node is None:

            return False

        if node.value == 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):

        elements = []

        self.\_inorder\_recursive(self.root, elements)

        return elements

    def \_inorder\_recursive(self, node, elements):

        if node:

            self.\_inorder\_recursive(node.left, elements)

            elements.append(node.value)

            self.\_inorder\_recursive(node.right, elements)

if \_\_name\_\_ == "\_\_main\_\_":

    bst = BinarySearchTree()

    for val in [15, 10, 20, 8, 12, 17, 25]:

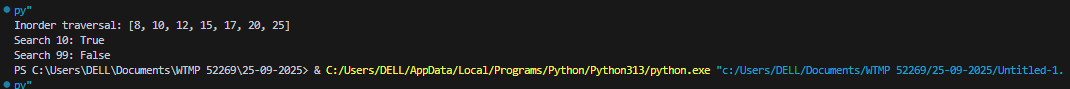
        bst.insert(val)

    print("Inorder traversal:", bst.inorder\_traversal())

    print("Search 10:", bst.search(10))

    print("Search 99:", bst.search(99))

OUTPUT:



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.

Prompt:

"Implement a Graph in Python using an adjacency list (dictionary). Add methods bfs(start), dfs(start) (iterative), and dfs\_recursive(start). Include inline comments explaining the traversal logic step by step. Also, compare recursive vs iterative DFS. Provide a test graph with at least 5–6 nodes and show traversal outputs."

CODE:

from collections import deque

class Graph:

    def \_\_init\_\_(self):

        self.adj\_list = {}

    def add\_edge(self, u, v):

        if u not in self.adj\_list:

            self.adj\_list[u] = []

        if v not in self.adj\_list:

            self.adj\_list[v] = []

        self.adj\_list[u].append(v)

        self.adj\_list[v].append(u)

    def bfs(self, start):

        visited = set()

        queue = deque([start])

        visited.add(start)

        result = []

        while queue:

            vertex = queue.popleft()

            result.append(vertex)

            # Visit all adjacent nodes

            for neighbor in self.adj\_list.get(vertex, []):

                if neighbor not in visited:

                    visited.add(neighbor)

                    queue.append(neighbor)

        return result

    def dfs(self, start):

        visited = set()

        stack = [start]

        result = []

        while stack:

            vertex = stack.pop()

            if vertex not in visited:

                visited.add(vertex)

                result.append(vertex)

                neighbors = self.adj\_list.get(vertex, [])

                for neighbor in reversed(neighbors):

                    if neighbor not in visited:

                        stack.append(neighbor)

        return result

    def dfs\_recursive(self, start):

        visited = set()

        result = []

        def dfs\_helper(v):

            visited.add(v)

            result.append(v)

            for neighbor in self.adj\_list.get(v, []):

                if neighbor not in visited:

                    dfs\_helper(neighbor)

        dfs\_helper(start)

        return result

if \_\_name\_\_ == "\_\_main\_\_":

    graph = Graph()

    edges = [(1, 2), (1, 3), (2, 4), (3, 5), (4, 5)]

    for u, v in edges:

        graph.add\_edge(u, v)

    print("BFS:", graph.bfs(1))

    print("DFS iterative:", graph.dfs(1))

    print("DFS recursive:", graph.dfs\_recursive(1))

OUTPUT:

