ASSIGNMENT-11.4

NAME :-P.CHARANKUMAR

ROLL NO :-2303A52138

BATCH -41

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.

Prompt Used:

Build a Stack class for undo operations in a text editor.

Include push, pop, peek, is_empty with docstrings.

Add test cases and an alternative deque-based implementation.

AI-Generated Code

```python
class Stack:
    """A simple Stack implementation for undo operations in a text editor."""

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

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

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

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

        Returns:
            The item at the top of the stack.

        Raises:
            IndexError: If the stack is empty.
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
Assets > #You are helping me build an undo featur.py > ...
class Stack:
    def pop(self):
        Raises:
            IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("Pop from an empty stack")
        return self.stack.pop()

    def peek(self):
        """Return the item at the top of the stack 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 an empty stack")
        return self.stack[-1]

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

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

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

        Returns:
            True if the stack is empty, False otherwise.
        """
        return len(self.stack) == 0
# Test cases for the Stack class
if __name__ == "__main__":
    stack = Stack()
    stack.push("First action")
    stack.push("Second action")
    stack.push("Third action")

    print(stack.peek())  # Output: Third action
    print(stack.pop())   # Output: Third action
    print(stack.peek())  # Output: Second action
    print(stack.is_empty())  # Output: False
    stack.pop()
    stack.pop()
    print(stack.is_empty())  # Output: True
```

BLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

Output:-

```
PS C:\Users\HARSHAVARDHAN\Downloads\Snake Game\Snake Game> & C:/Users/HARSHAVARDHAN/AppData/Local/Programs/Python/Python315/python.exe "c:/Users/HARSHAVARDHAN/Downloads/Snake Game/Snake Game/Assets/#You are helping me build an undo featur.py"
Third action
Third action
Second action
False
True
```

Explanation:

The code defines a `Stack` class that implements a simple stack data structure, which is commonly used for undo operations in text editors. The stack supports four main operations: `push`, `pop`, `peek`, and `is_empty`.

Task 2: Queue for Customer Service Requests (FIFO)

Scenario

You are developing a customer support system where service requests must be handled in the order they arrive.

Prompt Used:

Implement a FIFO queue for customer service requests.

First use a list-based approach, then an optimized deque-based approach.

Include enqueue, dequeue, is_empty, test cases, and performance comparison.

List-based approach

AI-Generated Code

```
class ListQueue:
    def __init__(self):
        self.queue = []

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

    def dequeue(self):
        if not self.is_empty():
            return self.queue.pop(0)
        raise IndexError("Dequeue from an empty queue")

    def is_empty(self):
        return len(self.queue) == 0
# Deque-based approach
from collections import deque
class DequeQueue:
    def __init__(self):
        self.queue = deque()

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

```

```python
class DequeQueue:
    def dequeue(self):
        if not self.is_empty():
            return self.queue.popleft()
        raise IndexError("Dequeue from an empty queue")

    def is_empty(self):
        return len(self.queue) == 0
# Test cases
def test_queue(queue_class):
    queue = queue_class()
    assert queue.is_empty() == True
    queue.enqueue("Request 1")
    queue.enqueue("Request 2")
    queue.enqueue("Request 3")
    assert queue.is_empty() == False
    assert queue.dequeue() == "Request 1"
    assert queue.dequeue() == "Request 2"
    assert queue.dequeue() == "Request 3"
    assert queue.is_empty() == True
if __name__ == "__main__":
    print("Testing ListQueue...")
    test_queue(ListQueue)
```

ROBLEMS 15 OUTPUT DEBUG CONSOLE TERMINAL PORTS

```python
    queue.enqueue("Request 2")
    queue.enqueue("Request 3")
    assert queue.is_empty() == False
    assert queue.dequeue() == "Request 1"
    assert queue.dequeue() == "Request 2"
    assert queue.dequeue() == "Request 3"
    assert queue.is_empty() == True
if __name__ == "__main__":
    print("Testing ListQueue...")
    test_queue(ListQueue)
    print("ListQueue tests passed.")

    print("Testing DequeQueue...")
    test_queue(DequeQueue)
    print("DequeQueue tests passed.")
```

```python
# Performance comparison
import time
def performance_comparison():
    list_queue = ListQueue()
    deque_queue = DequeQueue()
    num_operations = 100000

    # Test ListQueue
    start_time = time.time()
    for i in range(num_operations):
        list_queue.enqueue(f"Request {i}")
    for i in range(num_operations):
        list_queue.dequeue()
    list_time = time.time() - start_time

    # Test DequeQueue
    start_time = time.time()
    for i in range(num_operations):
        deque_queue.enqueue(f"Request {i}")
    for i in range(num_operations):
        deque_queue.dequeue()
    deque_time = time.time() - start_time
```

```python
    list_time = time.time() - start_time

    # Test DequeQueue
    start_time = time.time()
    for i in range(num_operations):
        deque_queue.enqueue(f"Request {i}")
    for i in range(num_operations):
        deque_queue.dequeue()
    deque_time = time.time() - start_time

    print(f"ListQueue time: {list_time:.4f} seconds")
    print(f"DequeQueue time: {deque_time:.4f} seconds")
if __name__ == "__main__":
    performance_comparison()
```

Output:-

```
PS C:\Users\HARSHAVARDHAN\Downloads\Snake Game\Snake Game> & C:/Users/HARSHAVARDHAN/AppData/Local/Pro
grams/Python/Python315/python.exe "c:/Users/HARSHAVARDHAN/Downloads/Snake Game/Snake Game/Assets/# Im
plement a FIFO queue for customer se.py"
Testing ListQueue...
ListQueue tests passed.
Testing DequeQueue...
DequeQueue tests passed.
ListQueue time: 9.1778 seconds
DequeQueue time: 0.0194 seconds
```

Explanation:

he code implements two versions of a FIFO (First-In-First-Out) queue for customer service requests: one using a list and the other using a deque from the collections module.

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.

Prompt Used:

Implement a singly linked list for a music playlist.

Support insert at end, delete by value, and traversal.

Add comments explaining pointer manipulation and edge cases.

AI-Generated Code

ssets > # Implement a singly linked list for a m.py > MusicPlaylist > insert_at_end

```
class MusicPlaylist:
    def delete_by_value(self, title):
                    previous.next = current.next  # Bypass the current node
                else:  # If it's the head node
                    self.head = current.next  # Move the head to the next node
                return  # Exit after deleting the song
            previous = current  # Move the previous pointer to the current node
            current = current.next  # Move to the next node
    def traverse(self):
        current = self.head  # Start from the head of the playlist
        while current:  # Traverse through the playlist
            print(current.title)  # Print the title of the current song
            current = current.next  # Move to the next node
# Example usage:
playlist = MusicPlaylist()
playlist.insert_at_end("Song A")
playlist.insert_at_end("Song B")
playlist.insert_at_end("Song C")
print("Playlist after inserting songs:")
playlist.traverse()
playlist.delete_by_value("Song B")
print("\nPlaylist after deleting 'Song B':")
playlist.traverse()
```

Output:-

```
PS C:\Users\HARSHAVARDHAN\Downloads\Snake Game\Snake Game> & C:/Users/HARSHAVARDHAN/AppData/Local/Programs/Python/Python315/python.exe "c:/Users/HARSHAVARDHAN/Downloads/Snake Game/Snake Game/Assets/# Implement a singly linked list for a m.py"
Playlist after inserting songs:
Song A
Song B
Song C

Playlist after deleting 'Song B':
Song A
Song C
PS C:\Users\HARSHAVARDHAN\Downloads\Snake Game\Snake Game>
```

Explaination:

The code defines a singly linked list to represent a music playlist. Each song is represented as a node (SongNode) that contains the song title and a pointer to the next song. The MusicPlaylist class provides methods to insert songs at the end of the playlist, delete songs by their title, and traverse the playlist to display all songs. The code also includes comments explaining the pointer manipulation and edge cases, such as handling an empty playlist or deleting the head node.

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.

Prompt Used:

Implement a Binary Search Tree (BST) for a student record system.

Support insert(value), search(value), and inorder_traversal().

Start from a partially written Node and BST class.

Complete the missing methods using recursion.

Maintain BST property: left < root < right.

Add meaningful docstrings and inline comments explaining left/right pointer updates.

Explain how BST improves search efficiency compared to linear search.

Include test cases for insertion, searching, and inorder traversal.

Discuss best-case, average-case, and worst-case time complexity.

AI-Generated Code:

sets > # Implement a Binary Search Tree (BST) f.py > BinarySearchTree > insert

```python
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None  # Pointer to the left child
        self.right = None  # Pointer to the right child
class BinarySearchTree:
    def __init__(self):
        self.root = None  # Initialize the root of the BST
    def insert(self, value):
        """Insert a new value into the BST while maintaining the BST property."""
        if self.root is None:
            self.root = TreeNode(value)  # If tree is empty, set root to new node
        else:
            self._insert_recursive(self.root, value)  # Otherwise, insert recursively
    def _insert_recursive(self, node, value):
        """Helper method to insert a value recursively."""
        if value < node.value:
            if node.left is None:
                node.left = TreeNode(value)  # Insert as left child
            else:
                self._insert_recursive(node.left, value)  # Recur left
        else:
            if node.right is None:
```

ROBLEMS 64 OUTPUT DEBUG CONSOLE TERMINAL PORTS

ssets > # Implement a Binary Search Tree (BST) f.py > BinarySearchTree > insert

```python
class BinarySearchTree:
    def _insert_recursive(self, node, value):
            else:
                (method) def _search_recursive(
                    self: Self@BinarySearchTree,           nsert as right child
                    node: Any,
                    value: Any                             , value)  # Recur right
    def         ) -> (Any | bool)
                                                           rn True if found, else False."""
                Helper method to search for a value recursively.  value)
    def _search_recursive(self, node, value):
        """Helper method to search for a value recursively."""
        if node is None:
            return False  # Value not found
        if node.value == value:
            return True  # Value found
        elif value < node.value:
            return self._search_recursive(node.left, value)  # Search left subtree
        else:
            return self._search_recursive(node.right, value)  # Search right subtree
    def inorder_traversal(self):
        """Perform an inorder traversal of the BST and return a list of values."""
        result = []
```

ROBLEMS 64 OUTPUT DEBUG CONSOLE TERMINAL PORTS

```python
        result = []
        self._inorder_recursive(self.root, result)
        return result
    def _inorder_recursive(self, node, result):
        """Helper method to perform inorder traversal recursively."""
        if node is not None:
            self._inorder_recursive(node.left, result)  # Traverse left subtree
            result.append(node.value)  # Visit current node
            self._inorder_recursive(node.right, result)  # Traverse right subtree
# Test cases
if __name__ == "__main__":
    bst = BinarySearchTree()
    bst.insert(10)
    bst.insert(5)
    bst.insert(15)
    bst.insert(3)
    bst.insert(7)
    bst.insert(12)
    bst.insert(18)

    # Test search
```

PROBLEMS 64 OUTPUT DEBUG CONSOLE TERMINAL PORTS

ustomer se.py 9+ · # Implement a singly linked list for a m.py 9+ · # Implement a Binary Search Tree (BST) f.py

sets > # Implement a Binary Search Tree (BST) f.py > BinarySearchTree > insert

```python
    bst.insert(5)
    bst.insert(15)
    bst.insert(3)
    bst.insert(7)
    bst.insert(12)
    bst.insert(18)

    # Test search
    print(bst.search(7))  # Output: True
    print(bst.search(4))  # Output: False

    # Test inorder traversal
    print(bst.inorder_traversal())  # Output: [3, 5, 7, 10, 12, 15, 18]
```

Output:

```
PS C:\Users\HARSHAVARDHAN\Downloads\Snake Game\Snake Game> & C:/Users/HARSHAVARDHAN/AppData/Local/Programs/Python/Python315/python.exe "c:/Users/HARSHAVARDHAN/Downloads/Snake Game/Snake Game/Assets/# Implement a Binary Search Tree (BST) f.py"
True
False
[3, 5, 7, 10, 12, 15, 18]
PS C:\Users\HARSHAVARDHAN\Downloads\Snake Game\Snake Game>
```

Explanation:

The code defines a Binary Search Tree (BST) data structure for managing student records. The TreeNode class represents each node in the tree, containing a value and pointers to the left and right children. The BinarySearchTree class provides methods for inserting values, searching for values, and performing an inorder traversal. The insert method adds new values while maintaining the BST property, ensuring that left children are less than the parent node and right children are greater. The search method checks for the presence of a value in the tree, and the inorder_traversal method returns a sorted list of values in the tree. The test cases demonstrate the functionality of the BST by inserting values, searching for existing and non-existing values, and performing an inorder traversal to display the sorted order of the values. The BST improves search efficiency compared to linear search by allowing it to skip large portions of the tree based on comparisons, resulting in an average-case time complexity of O(log n) for balanced trees, while linear search has a time complexity of O(n). However, in the worst case (e.g., when the tree is skewed), the time complexity can degrade to O(n).

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.

Prompt Used:

Implement a graph to model social network connections using an adjacency list.

Each user is a node, and friendships are edges.

Implement graph traversal algorithms:

- Breadth-First Search (BFS) to find nearby connections

- Depth-First Search (DFS) to explore deep connection paths

Add inline comments explaining each step of BFS and DFS traversal.

Explain how queues are used in BFS and stacks/recursion are used in DFS.

Implement DFS using:

- Recursive approach

- Iterative approach (using an explicit stack)

Compare recursive vs iterative DFS in terms of memory and use cases.

Explain practical real-world use cases of BFS vs DFS in social networks.

Include sample test cases demonstrating:

- BFS traversal from a given user

- DFS traversal from a given user

Keep the code clean, readable, and beginner-friendly.

AI-Generated Code

```python
# Keep the code clean, readable, and beginner-friendly.
class Graph:
    def __init__(self):
        self.adjacency_list = {}

    def add_user(self, user):
        if user not in self.adjacency_list:
            self.adjacency_list[user] = []

    def add_friendship(self, user1, user2):
        self.add_user(user1)
        self.add_user(user2)
        self.adjacency_list[user1].append(user2)
        self.adjacency_list[user2].append(user1)

    def bfs(self, start_user):
        visited = set()  # To keep track of visited nodes
        queue = [start_user]  # Queue for BFS
        bfs_result = []

        while queue:
            current_user = queue.pop(0)  # Dequeue the first user
            if current_user not in visited:
                visited.add(current_user)  # Mark the user as visited
                bfs_result.append(current_user)  # Add to result
```

```python
                bfs_result.append(current_user)  # Add to result

                # Enqueue all unvisited friends of the current user
                for friend in self.adjacency_list[current_user]:
                    if friend not in visited:
                        queue.append(friend)

        return bfs_result

    def dfs_recursive(self, start_user, visited=None):
        if visited is None:
            visited = set()  # To keep track of visited nodes
        visited.add(start_user)  # Mark the user as visited
        dfs_result = [start_user]  # Add to result

        # Recursively visit all unvisited friends of the current user
        for friend in self.adjacency_list[start_user]:
            if friend not in visited:
                dfs_result.extend(self.dfs_recursive(friend, visited))

        return dfs_result

    def dfs_iterative(self, start_user):
```

```python
    def dfs_iterative(self, start_user):
        visited = set()  # To keep track of visited nodes
        stack = [start_user]  # Stack for DFS
        dfs_result = []

        while stack:
            current_user = stack.pop()  # Pop the last user from the stack
            if current_user not in visited:
                visited.add(current_user)  # Mark the user as visited
                dfs_result.append(current_user)  # Add to result

                # Push all unvisited friends of the current user onto the stack
                for friend in reversed(self.adjacency_list[current_user]):
                    if friend not in visited:
                        stack.append(friend)

        return dfs_result
# Sample test cases
if __name__ == "__main__":
    graph = Graph()
    graph.add_friendship("Alice", "Bob")
    graph.add_friendship("Alice", "Charlie")
    graph.add_friendship("Bob", "David")
```

```python
f __name__ == "__main__":
    graph = Graph()
    graph.add_friendship("Alice", "Bob")
    graph.add_friendship("Alice", "Charlie")
    graph.add_friendship("Bob", "David")
    graph.add_friendship("Charlie", "Eve")
    graph.add_friendship("David", "Frank")

    print("BFS from Alice:", graph.bfs("Alice"))  # Output: ['Alice', 'Bob', 'Charlie', 'David'
    print("DFS (recursive) from Alice:", graph.dfs_recursive("Alice"))  # Output: ['Alice', 'Bo
    print("DFS (iterative) from Alice:", graph.dfs_iterative("Alice"))  # Output: ['Alice', 'Ch
```

OUTPUT:

```
PS C:\Users\HARSHAVARDHAN\Downloads\Snake Game\Snake Game> & C:/Users/HARSHAVARDHAN/AppData/Local/Pro
grams/Python/Python315/python.exe "c:/Users/HARSHAVARDHAN/Downloads/Snake Game/Snake Game/Assets/# Im
plement a graph to model social netw.py"
BFS from Alice: ['Alice', 'Bob', 'Charlie', 'David', 'Eve', 'Frank']
DFS (recursive) from Alice: ['Alice', 'Bob', 'David', 'Frank', 'Charlie', 'Eve']
DFS (iterative) from Alice: ['Alice', 'Bob', 'David', 'Frank', 'Charlie', 'Eve']
PS C:\Users\HARSHAVARDHAN\Downloads\Snake Game\Snake Game>
```

Explanation:

The code defines a Graph class to model social network connections using an adjacency list. Each user is represented as a node, and friendships are represented as edges between nodes. The Graph class provides methods to add users, add friendships, and perform breadth-first search (BFS) and depth-first search (DFS) traversals.