AI ASSISTANT CODING - ASSIGNMENT-11.4

Name: Valaboju Harshavardhanu

Roll No: 2303A52354 **Batch:** 41

Task 1: Stack Implementation for Undo Operations (LIFO)

Prompt used:

#Generate Stack class in Python with the following methods: push() ,pop(), peek(),is_empty() along with proper docstrings .Explain why a stack is suitable for undo functionality ,testcases for undo and also suggest an alternative implementation using collections.deque.

Code:

```
class Stack:
    """
    A simple implementation of a stack data structure.
    """
    def __init__(self):
        """Initialize an empty stack."""
        self.items = []

    def push(self, action):
        """
        Add an action to the top of the stack.
        Args:
            action: The action to be added to the stack.
        """
        self.items.append(action)

    def pop(self):
        """
        Remove and return the action at the top of the stack.
        Returns:
            The action at the top of the stack.
        Raises:
            IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("Stack is empty")
        return self.items.pop()
    def peek(self):
        """
        Return the action at the top of the stack without removing it.
        Returns:
            The action at the top of the stack.
        Raises:
            IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("Stack is empty")
        return self.items[-1]
    def is_empty(self):
        """
        Check if the stack is empty.
        Returns:
            bool: True if the stack is empty, False otherwise.
        """
        return len(self.items) == 0
```

```python
stack = Stack()
stack.push("Action 1")
stack.push("Action 2")
print(stack.peek())  # Output: Action 2
print(stack.pop())   # Output: Action 2
print(stack.peek())  # Output: Action 1
# Alternative implementation using collections.deque
from collections import deque
class StackDeque:
    """
    A stack implementation using collections.deque for efficient appends and pops.
    """
    def __init__(self):
        """Initialize an empty stack."""
        self.items = deque()
    def push(self, action):
        """Add an action to the top of the stack."""
        self.items.append(action)
    def pop(self):
        """Remove and return the action at the top of the stack."""
        if self.is_empty():
            raise IndexError("Stack is empty")
        return self.items.pop()
    def peek(self):
        """Return the action at the top of the stack without removing it."""
        if self.is_empty():
            raise IndexError("Stack is empty")
        return self.items[-1]
    def is_empty(self):
        """Check if the stack is empty."""
        return len(self.items) == 0
```

Output:

```
Action 2
Action 2
Action 1
PS C:\Users\saipr\OneDrive\Desktop\Ai-Assistant>
```

Justification:

A stack is suitable for undo functionality because it follows the Last In, First Out (LIFO) principle. When a user performs an action, it is pushed onto the stack. If the user wants to undo the last action, we can simply pop the top action from the stack and reverse it. This allows us to easily manage and track the sequence of actions.

Task 2: Queue for Customer Service Requests (FIFO)

Prompt used:

#Implement a Queue class for a customer support system where requests are handled in FIFO order using a Python list with enqueue(request), dequeue(), and is_empty(), review its performance, then implement an optimized version using collections.deque, compare time complexity, and justify why deque is better.

Code:

```
class Queue:
    A simple implementation of a queue data structure using a Python list.
    """
    def __init__(self):
        """Initialize an empty queue."""
        self.items = []
    def enqueue(self, request):
        """Add a request to the end of the queue."""
        self.items.append(request)
    def dequeue(self):
        """Remove and return the request at the front of the queue."""
        if self.is_empty():
            raise IndexError("Queue is empty")
        return self.items.pop(0)
    def is_empty(self):
        """Check if the queue is empty."""
        return len(self.items) == 0
# Example usage:
queue = Queue()
queue.enqueue("Request 1")
queue.enqueue("Request 2")
print(queue.dequeue())  # Output: Request 1
# Performance review: The dequeue operation in this implementation has a time complexity of O(
from collections import deque
class QueueDeque:
    """
    A queue implementation using collections.deque for efficient enqueue and dequeue operation
    """
    def __init__(self):
        """Initialize an empty queue."""
        self.items = deque()
    def enqueue(self, request):
        """Add a request to the end of the queue."""
        self.items.append(request)
    def dequeue(self):
        """Remove and return the request at the front of the queue."""
        if self.is_empty():
            raise IndexError("Queue is empty")
        return self.items.popleft()
    def is_empty(self):
        """Check if the queue is empty."""
        return len(self.items) == 0
# Time complexity comparison: In the optimized version using collections.deque, both enqueue a
# Example usage:
queue_deque = QueueDeque()
queue_deque.enqueue("Request 1")
queue_deque.enqueue("Request 2")
print(queue_deque.dequeue())  # Output: Request 1
```

Output:

```
PS C:\Users\saipr\OneDrive\Desktop\Ai-Assistant> &
istant/Assign10-3.py
Request 1
Request 1
PS C:\Users\saipr\OneDrive\Desktop\Ai-Assistant>
```

Justification:

In the list-based queue, enqueue runs in O(1) time but dequeue uses pop(0) which takes O(n) time because all remaining elements must shift left, making it inefficient for large numbers of customer requests, whereas in the deque-based queue both enqueue and dequeue run in O(1) time since deque is implemented as a doubly linked structure that allows efficient insertion and removal from both ends without shifting elements, making deque more scalable and better suited for real-world customer support systems handling high request volumes.

Task 3: Singly Linked List for Dynamic Playlist Management

Prompt used:

#Design a Singly Linked List for managing a music playlist where songs can be added at the end, removed by name, and displayed in order. The implementation should clearly explain how pointers move during insertion and deletion, highlight tricky parts, and include proper test cases for edge conditions like empty list, single node, and deleting the head.

Code:

```
class Node:
    """
    A node in a singly linked list.
    """
    def __init__(self, song):
        """Initialize a node with a song and a pointer to the next node."""
        self.song = song
        self.next = None
class SinglyLinkedList:
    """
    A singly linked list to manage a music playlist.
    """
    def __init__(self):
        """Initialize an empty linked list."""
        self.head = None
```

```
    def add_song(self, song):
        """Add a song to the end of the playlist."""
        new_node = Node(song)
        if not self.head:
            self.head = new_node
            return
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node
    def remove_song(self, song):
        """Remove a song by name from the playlist."""
        current_node = self.head
        previous_node = None
        while current_node:
            if current_node.song == song:
                if previous_node:
                    previous_node.next = current_node.next
                else:
                    self.head = current_node.next  # Deleting the head node
                return True  # Song found and removed
            previous_node = current_node
            current_node = current_node.next
        return False  # Song not found
    def display_playlist(self):
        """Display the songs in the playlist in order."""
        songs = []
        current_node = self.head
        while current_node:
            songs.append(current_node.song)
            current_node = current_node.next
        return songs
# Test cases
playlist = SinglyLinkedList()
# Test case: Adding songs to the playlist
playlist.add_song("Song 1")
playlist.add_song("Song 2")
print(playlist.display_playlist())  # Output: ['Song 1', 'Song 2']
# Test case: delete middle song
playlist.add_song("Song 3")
playlist.remove_song("Song 2")
print(playlist.display_playlist())  # Output: ['Song 1', 'Song 3']
# Test case: Deleting the head song
playlist.remove_song("Song 1")
print(playlist.display_playlist())  # Output: ['Song 3']
# Test case: Deleting a last song
playlist.remove_song("Song 3")
print(playlist.display_playlist())  # Output: []
# Test case: Deleting from an empty playlist
result = playlist.remove_song("Non-existent Song")
print(result)  # Output: False
```

Output:

```
PS C:\Users\saipr\OneDrive\Desktop\Ai-Assistant> & "C:\Program Files\Python314\pyt
['Song 1', 'Song 2']
['Song 1', 'Song 3']
['Song 1', 'Song 2']
['Song 1', 'Song 3']
['Song 3']
[]
False
PS C:\Users\saipr\OneDrive\Desktop\Ai-Assistant>
```

Justification:

A singly linked list is perfect for a playlist because you can easily add or remove songs without disturbing the entire list. When adding a song, the important part is making sure the last song properly points to the new one. While deleting, extra care is needed if the song is the first one or somewhere in the middle, so the links stay correct. By handling cases like an empty playlist or a single song, we make sure the system works smoothly every time

Task 4: Binary Search Tree for Fast Record Lookup.

Prompt used:

Build a Binary Search Tree (BST) for a student record system where fast searching by roll number is required. Complete the missing methods for insert, search, and inorder traversal, add clear docstrings, and explain how a BST improves search efficiency compared to linear search, including best-case and worst-case performance.

Code:

```python
class TreeNode:
    """
    A node in a binary search tree.
    """
    def __init__(self, roll_number, student_name):
        """Initialize a tree node with a roll number, student name, and pointers to left and right children."""
        self.roll_number = roll_number
        self.student_name = student_name
        self.left = None
        self.right = None
class BinarySearchTree:
    """
    A binary search tree for managing student records.
    """
    def __init__(self):
        """Initialize an empty binary search tree."""
        self.root = None
    def insert(self, roll_number, student_name):
        """Insert a new student record into the BST."""
        new_node = TreeNode(roll_number, student_name)
        if not self.root:
            self.root = new_node
            return
        current_node = self.root
```

```
        while True:
            if roll_number < current_node.roll_number:
                if current_node.left is None:
                    current_node.left = new_node
                    return
                current_node = current_node.left
            else:
                if current_node.right is None:
                    current_node.right = new_node
                    return
                current_node = current_node.right
    def search(self, roll_number):
        """Search for a student record by roll number."""
        current_node = self.root
        while current_node:
            if roll_number == current_node.roll_number:
                return current_node.student_name  # Student found
            elif roll_number < current_node.roll_number:
                current_node = current_node.left
            else:
                current_node = current_node.right
        return None  # Student not found
    def inorder_traversal(self):
        """Perform an inorder traversal of the BST and return a list of student records."""
        records = []
        def traverse(node):
            if node:
                traverse(node.left)
                records.append((node.roll_number, node.student_name))
                traverse(node.right)
        traverse(self.root)
        return records
# Explanation of search efficiency:
# A binary search tree (BST) improves search efficiency compared to linear search because it organizes data
# in a hierarchical structure. In a BST, each node has at most two children, and the left child contains values
# less than the parent node, while the right child contains values greater than the parent node.
# This allows for efficient searching by eliminating half of the remaining nodes at each step.
# Best-case performance occurs when the BST is perfectly balanced, resulting in a search time complexity of O(log n).
# Worst-case performance occurs when the BST is skewed (e.g., all nodes are on
# one side), resulting in a search time complexity of O(n), which is equivalent to linear search.
# However, with proper balancing techniques (e.g., AVL trees, Red-Black trees), we can maintain O(log n) search efficiency
# even in the worst case.
# Example usage:
bst = BinarySearchTree()
bst.insert(101, "Alice")
bst.insert(102, "Bob")
bst.insert(100, "Charlie")
print(bst.search(101))  # Output: Alice
print(bst.search(105))  # Output: None
print(bst.inorder_traversal())  # Output: [(100, 'Charlie'),# (101, 'Alice'), (102, 'Bob')]
```

Output:

```
PS C:\Users\saipr\OneDrive\Desktop\Ai-Assistant> & "C:\Program Files\Python314\python.exe" c:/Users/saipr/OneDrive/Desktop/Ai-Assistant/Assign10-3.py
Alice
None
[(100, 'Charlie'), (101, 'Alice'), (102, 'Bob')]
PS C:\Users\saipr\OneDrive\Desktop\Ai-Assistant>
```

Justification:

A Binary Search Tree improves search efficiency by dividing the data at each step, reducing the search time to O(log n) in the best (balanced) case compared to O(n) in linear search.However, in the worst case when the tree becomes unbalanced, the search time can degrade to O(n), similar to linear search.

Task 5: Graph Traversal for Social Network Connections

Prompt used:

#Create a social network graph using an adjacency list and implement BFS to find nearby friends and DFS to explore deeper connections, with clear comments and explanation of when to use each.

Code:

```python
class Graph:
    """
    A simple implementation of a graph using an adjacency list.
    """
    def __init__(self):
        """Initialize an empty graph."""
        self.adjacency_list = {}
    def add_edge(self, person1, person2):
        """Add a bidirectional edge between two people in the graph."""
        if person1 not in self.adjacency_list:
            self.adjacency_list[person1] = []
        if person2 not in self.adjacency_list:
            self.adjacency_list[person2] = []
        self.adjacency_list[person1].append(person2)
        self.adjacency_list[person2].append(person1)
    def bfs(self, start):
        """Perform Breadth-First Search (BFS) to find nearby friends."""
        visited = set()
        queue = [start]
        nearby_friends = []
        while queue:
            current_person = queue.pop(0)
            if current_person not in visited:
                visited.add(current_person)
                nearby_friends.append(current_person)
                # Add neighbors to the queue
                for neighbor in self.adjacency_list.get(current_person, []):
                    if neighbor not in visited:
                        queue.append(neighbor)
        return nearby_friends
    def dfs(self, start, visited=None):
        """Perform Depth-First Search (DFS) to explore deeper connections."""
        if visited is None:
            visited = set()
        visited.add(start)
        print(start)  # Process the current person (e.g., print their name)
        for neighbor in self.adjacency_list.get(start, []):
            if neighbor not in visited:
                self.dfs(neighbor, visited)
# Explanation of when to use BFS and DFS:
# BFS is ideal for finding nearby friends because it explores the graph level by level, starting from
# the given node and visiting all its neighbors before moving on to the neighbors' neighbors. This makes
# it suitable for scenarios where we want to find all friends within a certain distance (.
# DFS, on the other hand, is better for exploring deeper connections because it goes as far
# as possible along a branch before backtracking. This is useful for scenarios where we want to
# explore the entire network of connections or find specific paths between nodes.
# Example usage:
graph = Graph()
graph.add_edge("Alice", "Bob")
graph.add_edge("Alice", "Charlie")
graph.add_edge("Bob", "David")
graph.add_edge("Charlie", "Eve")
print(graph.bfs("Alice"))  # Output: ['Alice', 'Bob', 'Charlie', 'David', 'Eve']
print("DFS starting from Alice:")
graph.dfs("Alice")  # Output: Alice, Bob, David, Charlie, Eve (order may vary)
```

Output:

```
PS C:\Users\saipr\OneDrive\Desktop\Ai-Assistant> & "C:\Program Files\Python314\python.exe" c:/Users/saipr/OneDrive/Desktop/Ai-Assistant/Assign10-3.py
['Alice', 'Bob', 'Charlie', 'David', 'Eve']
DFS starting from Alice:
Alice
Bob
David
Charlie
Eve
PS C:\Users\saipr\OneDrive\Desktop\Ai-Assistant>
```

Justification:

BFS is useful when we want to find nearby friends or the shortest connection between users because it explores level by level. DFS is better when we want to explore deep relationship chains or analyse the overall structure of the network .Recursive DFS is simpler to write, while iterative DFS is safer for large networks as it avoids recursion limits.