AI-ASSITED CODING

ASSIGNMENT -11.2

ROLLNO:2403A52084
BATCH:04

TASK:1

PROMPT: Create a Python class Stack using a list to store elements. Implement push, pop, peek, and is_empty methods with proper docstrings. Ensure pop and peek raise exceptions when the stack is empty.

CODE:

```python
class Stack:
    def __init__(self):
        """Initialize an empty stack."""
        self.items = []

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

    def pop(self):
        """Remove and return the top item from the stack. Raise IndexError if empty."""
        if self.is_empty():
            raise IndexError("pop from empty stack")
        return self.items.pop()

    def peek(self):
        """Return the top item without removing it. Raise IndexError if empty."""
        if self.is_empty():
            raise IndexError("peek from empty stack")
        return self.items[-1]

    def is_empty(self):
        """Return True if the stack is empty, False otherwise."""
        return len(self.items) == 0

if __name__ == "__main__":
    s = Stack()
    print("Is empty?", s.is_empty())
    s.push(10)
    s.push(20)
    s.push(30)
    print("Peek:", s.peek())
    print("Pop:", s.pop())
    print("Current stack:", s.items)
    print("Peek after pop:", s.peek())
    print("Current stack:", s.items)
    print("Is empty?", s.is_empty())
```

OUTPUT:

```
PS C:\Users\NIKHITHA\OneDrive\Desktop\AI> python -u "c:\Users\NIKHITHA\OneDrive\Desktop\AI\LAB11.2\task1.py"
Is empty? True
Peek: 30
Pop: 30
Current stack: [10, 20]
Peek after pop: 20
Current stack: [10, 20]
Is empty? False
PS C:\Users\NIKHITHA\OneDrive\Desktop\AI>
```

OBSERVATOIN:

The Stack class in task1.py is a clean and correct implementation of a fundamental data structure. It effectively uses a Python list to achieve the Last-In, First-Out (LIFO) behavior and includes robust error handling by raising an IndexError on an empty stack. The code is well-documented and the example usage block clearly demonstrates its functionality.

TASK02:

PROMPT: Implement a Python class Queue using a list to store elements. Include enqueue, dequeue, peek, and size methods to follow FIFO behavior. Add docstrings to explain each method clearly.

CODE:

```python
class Queue:
    def __init__(self):
        self._items = []

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

    def dequeue(self):
        if self.is_empty():
            raise IndexError("dequeue from an empty queue")
        return self._items.pop(0)

    def peek(self):
        if self.is_empty():
            raise IndexError("peek from an empty queue")
        return self._items[0]

    def is_empty(self):
        return not self._items

    def size(self):
        return len(self._items)

    def __str__(self):
        return str(self._items)
# Example Usage:
if __name__ == '__main__':
    my_queue = Queue()
    print(f"Is the queue empty? {my_queue.is_empty()}")  # Expected: True
    print("Enqueuing 10, 20, 30 into the queue...")
    my_queue.enqueue(10)
    my_queue.enqueue(20)
    my_queue.enqueue(30)
    print(f"Current queue: {my_queue}")  # Expected: [10, 20, 30]
    print(f"Queue size: {my_queue.size()}")  # Expected: 3
    print(f"Peeking at the front item: {my_queue.peek()}")  # Expected: 10
    print(f"Current queue after peek: {my_queue}")  # Expected: [10, 20, 30]
    print(f"Dequeuing item: {my_queue.dequeue()}")  # Expected: 10
    print(f"Current queue after dequeue: {my_queue}")  # Expected: [20, 30]
    print(f"Dequeuing item: {my_queue.dequeue()}")  # Expected: 20
    print(f"Current queue: {my_queue}")  # Expected: [30]
    print(f"Queue size: {my_queue.size()}")  # Expected: 1
```

OUTPUT:

```
                                        > python -u "c:\Users\NIKHITHA\
Is the queue empty? True
Enqueuing 10, 20, 30 into the queue...
Current queue: [10, 20, 30]
Queue size: 3
Peeking at the front item: 10
Current queue after peek: [10, 20, 30]
Dequeuing item: 10
Current queue after dequeue: [20, 30]
Dequeuing item: 20
Current queue: [30]
Queue size: 1
Dequeuing item: 30
Is the queue empty? True
PS C:\Users\NIKHITHA\OneDrive\Desktop\AI>
```

OBSERVATION:

he Queue class you've implemented in task2.py is an excellent, functional example of a First-In, First-Out (FIFO) data structure using a Python list. It correctly adds items to the end (enqueue) and removes them from the beginning (dequeue), perfectly simulating a real-world queue. The implementation is clean, well-documented, and includes a thorough example block that demonstrates its correctness.

This code effectively uses a list to manage the queue's state, with append() for adding items and pop(0) for removing them, which is a classic approach for this data structure.

TASK03:

PROMPT:

Generate a Python implementation of a singly linked list with Node and LinkedList classes. Include insert(data) to add elements and display() to print the list. Add clear docstrings for each method.

CODE:

```python
class Node:
    def __init__(self, data):
        """Initialize node with data and next pointer."""
        self.data = data
        self.next = None
class LinkedList:
    def __init__(self):
        """Initialize an empty linked list."""
        self.head = None
    def insert(self, data):
        """Insert a new node with the given data at the end of the list."""
        new_node = Node(data)
        if not self.head:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node
    def display(self):
        """Print all elements in the linked list."""
        elements = []
        current = self.head
        while current:
            elements.append(str(current.data))
            current = current.next
        print(" -> ".join(elements))
if __name__ == "__main__":
    ll = LinkedList()
    ll.insert(5)
    ll.insert(10)
    ll.insert(15)
    print("Linked List contents:")
    ll.display()
```

OUTPUT:

```
5 -> 10 -> 15
PS C:\Users\NIKHITHA\OneDrive\Desktop\AI> 
```

OBSERVATOIN:

This code defines a singly linked list with nodes that store data and a pointer to the next node. The insert method appends new data to the end, and display prints all elements in order. The __main__ block creates a list and shows its contents.

TASK:04

This code defines a singly linked list with nodes that store data and a pointer to the next node. The insert method appends new data to the end, and display prints all elements in order. The __main__ block creates a list and shows its contents.

CODE:

```python
class Node:
    def __init__(self, data):
        self.left = None
        self.right = None
class BST:
    def __init__(self):
        self.root = None
    def insert(self, data):
        self.root = self._insert_recursive(self.root, data)
    def _insert_recursive(self, node, data):
        if node is None:
            return Node(data)
        if data < node.data:
            node.left = self._insert_recursive(node.left, data)
        else:
            node.right = self._insert_recursive(node.right, data)
        return node
    def in_order_traversal(self):
        self._in_order_recursive(self.root)

    def _in_order_recursive(self, node):
        if node:
            self._in_order_recursive(node.left)
            print(node.data, end=' ')
            self._in_order_recursive(node.right)
# Sample usage
if __name__ == "__main__":
    bst = BST()
    bst.insert(50)
    bst.insert(30)
    bst.insert(70)
    bst.insert(20)
    bst.insert(40)
    bst.insert(60)
    bst.insert(80)
    print("In-order traversal of BST:")
    bst.in_order_traversal()
```

OUTPUT:

```
In-order traversal of BST:
20 30 40 50 60 70 80
PS C:\Users\NIKHITHA\OneDrive\Desktop\AI>
```

OBSERVATOIN:

This code builds a Binary Search Tree (BST) where each node is inserted recursively based on its value. The in_order_traversal method prints the tree's elements in ascending order by visiting left subtree, root, then right subtree.

TASK05:

PROMPT: Create a Python class HashTable using lists and chaining to handle collisions. Implement insert, search, and delete methods with clear docstrings and comments. Ensure keys are hashed and stored efficiently.

CODE:

task5 > HashTable > delete

C:\Users\NIKHITHA\OneDrive\Desktop\AI\LAB11.2

```python
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]
    def _hash(self, key):
        return hash(key) % self.size
    def insert(self, key, value):
        index = self._hash(key)
        for pair in self.table[index]:
            if pair[0] == key:
                pair[1] = value
                return
        self.table[index].append([key, value])
    def search(self, key):
        index = self._hash(key)
        for pair in self.table[index]:
            if pair[0] == key:
                return pair[1]
        return None
    def delete(self, key):
        index = self._hash(key)
        for i, pair in enumerate(self.table[index]):
            if pair[0] == key:
                del self.table[index][i]
                return True
        return False
# Sample usage
if __name__ == "__main__":
    ht = HashTable()
    ht.insert("apple", 100)
    ht.insert("banana", 200)
    ht.insert("orange", 300)

    print("Search 'banana':", ht.search("banana"))  # Output: 200
    ht.delete("banana")
    print("Search 'banana' after deletion:", ht.search("banana"))  # Output: None
```

OUTPUT:

```
Search 'banana': 200
Search 'banana' after deletion: None
PS C:\Users\NIKHITHA\OneDrive\Desktop\AI>
```

OBSERVATOIN:

This code defines a hash table using lists and handles collisions with chaining (linked lists in each bucket). The insert method adds or updates key-value pairs, search retrieves values by key, and delete removes entries—all using a hash function to determine bucket placement.

TASK06:

PROMPT:

Implement a graph using an adjacency list in Python. Define a Graph class with methods to add vertices, add edges, and display connections between nodes. Use dictionaries to store adjacency relationships efficiently.

CODE:

```python
class Graph:
    def __init__(self):
        self.adj_list = {}

    def add_vertex(self, vertex):
        if vertex not in self.adj_list:
            self.adj_list[vertex] = []

    def add_edge(self, src, dest):
        if src not in self.adj_list:
            self.add_vertex(src)
        if dest not in self.adj_list:
            self.add_vertex(dest)
        self.adj_list[src].append(dest)
        self.adj_list[dest].append(src)  # For undirected graph

    def display(self):
        for vertex in self.adj_list:
            print(f"{vertex} -> {', '.join(self.adj_list[vertex])}")

# Sample usage
g = Graph()
g.add_edge("A", "B")
g.add_edge("A", "C")
g.add_edge("B", "D")
g.display()
```

OUTPUT:

```
PS C:\Users\NIKHITHA\OneDrive\Desktop\AI> python -u "c:\Users\NIKHITHA\OneDrive\Desktop\AI\LAB11.2\task6"
A -> B, C
B -> A, D
C -> A
D -> B
PS C:\Users\NIKHITHA\OneDrive\Desktop\AI>
```

OBSEVATION:

The Graph class uses a dictionary to represent an adjacency list, where each key is a vertex and its value is a list of connected vertices. add_edge ensures both vertices exist and adds a bidirectional link, while display prints all connections clearly.

TASK07

PROMPT:

Implement a priority queue in Python using the heapq module. Create a PriorityQueue class with methods to enqueue items with priority, dequeue the highest priority item, and display the current queue state. Use a min-heap for efficient priority management.

CODE:

```python
import heapq

class PriorityQueue:
    def __init__(self):
        self.heap = []
        self.count = 0  # To handle items with same priority

    def enqueue(self, item, priority):
        heapq.heappush(self.heap, (priority, self.count, item))
        self.count += 1

    def dequeue(self):
        if self.heap:
            return heapq.heappop(self.heap)[2]
        return None

    def display(self):
        print("Queue state:")
        for priority, _, item in sorted(self.heap):
            print(f"{item} (Priority: {priority})")

# Sample usage
pq = PriorityQueue()
pq.enqueue("Task A", 3)
pq.enqueue("Task B", 1)
pq.enqueue("Task C", 2)
pq.display()
print("Dequeued:", pq.dequeue())
pq.display()
```

OUTPUT:

```
Queue state:
Task B (Priority: 1)
Task C (Priority: 2)
Task A (Priority: 3)
Dequeued: Task B
Queue state:
Task C (Priority: 2)
Task A (Priority: 3)
PS C:\Users\NIKHITHA\OneDrive\Desktop\AI>
```

OBSERVATION :

The PriorityQueue class uses a min-heap to store items with their priorities, ensuring efficient retrieval of the highest-priority item. The enqueue method adds items with a tie-breaker count, dequeue removes the item with the lowest priority value, and display shows the queue in sorted order.

TASK08:

PROMPT:

Implement a double-ended queue (deque) using Python's collections.deque. Create a DequeDS class with methods to insert and remove elements from both ends, and include docstrings to explain each method's functionality.

CODE:

```python
from collections import deque

class DequeDS:
    def __init__(self):
        """Initialize an empty deque."""
        self.deque = deque()

    def insert_front(self, item):
        """Insert an item at the front of the deque."""
        self.deque.appendleft(item)

    def insert_rear(self, item):
        """Insert an item at the rear of the deque."""
        self.deque.append(item)

    def remove_front(self):
        """Remove and return the item from the front of the deque."""
        if self.deque:
            return self.deque.popleft()
        return None

    def remove_rear(self):
        """Remove and return the item from the rear of the deque."""
        if self.deque:
            return self.deque.pop()
        return None

    def display(self):
        """Display the current state of the deque."""
        print("Deque contents:", list(self.deque))

# Sample usage
dq = DequeDS()
dq.insert_front(10)
dq.insert_rear(20)
dq.insert_front(5)
dq.display()
print("Removed from rear:", dq.remove_rear())
dq.display()
```

OUTPUT:

```
Deque contents: [5, 10, 20]
Removed from rear: 20
Deque contents: [5, 10]
PS C:\Users\NIKHITHA\OneDrive\Desktop\AI>
```

TASK09:

PROMPT:

Generate a markdown table comparing common data structures like stack, queue, linked list, and others. Include key operations (insert, delete, search) and their time complexities to highlight performance differences.

CODE:

```python
class DataStructureComparison:
    def __init__(self):
        self.table = [
            ["Data Structure", "Insert (Avg)", "Delete (Avg)", "Search (Avg)", "Notes"],
            ["Stack", "O(1)", "O(1)", "O(n)", "LIFO; uses push/pop"],
            ["Queue", "O(1)", "O(1)", "O(n)", "FIFO; uses enqueue/dequeue"],
            ["Deque", "O(1)", "O(1)", "O(n)", "Insert/remove from both ends"],
            ["Linked List", "O(1)", "O(1)", "O(n)", "Sequential access"],
            ["Doubly Linked List", "O(1)", "O(1)", "O(n)", "Bi-directional traversal"],
            ["Array", "O(1)", "O(n)", "O(n)", "Fixed size; fast indexing"],
            ["Dynamic Array", "O(1)*", "O(n)", "O(n)", "Resizes automatically"],
            ["Hash Table", "O(1)", "O(1)", "O(1)", "Fast lookup; may have collisions"],
            ["Binary Search Tree", "O(log n)", "O(log n)", "O(log n)", "Requires balancing"],
            ["Heap (Min/Max)", "O(log n)", "O(log n)", "O(n)", "Used in priority queues"]
        ]

    def display(self):
        for row in self.table:
            print("{:<22} {:<15} {:<15} {:<15} {}".format(*row))

# Sample usage
ds = DataStructureComparison()
ds.display()
```

OUTPUT:

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

Queue                  O(1)            O(1)            O(n)            FIFO; uses enqueue/dequeue
Deque                  O(1)            O(1)            O(n)            Insert/remove from both ends
Linked List            O(1)            O(1)            O(n)            Sequential access
Doubly Linked List     O(1)            O(1)            O(n)            Bi-directional traversal
Array                  O(1)            O(n)            O(n)            Fixed size; fast indexing
Dynamic Array          O(1)*           O(n)            O(n)            Resizes automatically
Hash Table             O(1)            O(1)            O(1)            Fast lookup; may have collision
s
Binary Search Tree     O(log n)        O(log n)        O(log n)        Requires balancing
Heap (Min/Max)         O(log n)        O(log n)        O(n)            Used in priority queues
PS C:\Users\NIKHITHA\OneDrive\Desktop\AI>
```

TASK10:

PROMPT:

To develop a Campus Resource Management System, each feature should be paired with the most suitable data structure based on its operational needs. For **Student Attendance Tracking**, a **Hash Table** is ideal as it allows constant-time access to student records using unique IDs, making logging and retrieval efficient. The **Event Registration System** benefits from a **Linked List**, which supports dynamic participant addition and quick removal without shifting elements. For **Library Book Borrowing**, a **Binary Search Tree (BST)** is appropriate since it enables ordered storage and fast lookup of books by title or ID, along with efficient due date tracking. The **Bus Scheduling System** is best modeled using a **Graph**, which naturally represents routes and connections between stops, allowing traversal and optimization. Lastly, the **Cafeteria Order Queue** should use a **Queue**, ensuring students are served in the order they arrive (FIFO). Among these, implementing the **Event Registration System** using a linked list is straightforward and practical. The linked list allows flexible participant management, and with AI-assisted code generation, we can build a Python program that supports adding, removing, and displaying participants. This implementation will include docstrings, inline comments, and assert-based test cases to validate functionality. A report will compile the prompt, code, test results, and analysis, ensuring clarity and completeness for academic submission.

CODE:

```python
class ParticipantNode:
    """Represents a participant in the event."""
    def __init__(self, name):
        self.name = name
        self.next = None

class EventRegistration:
    """Manages event participants using a singly linked list."""
    def __init__(self):
        self.head = None

    def register(self, name):
        """Add a participant to the end of the list."""
        new_node = ParticipantNode(name)
        if not self.head:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node

    def remove(self, name):
        """Remove a participant by name."""
        current = self.head
        prev = None
        while current:
            if current.name == name:
                if prev:
                    prev.next = current.next
                else:
                    self.head = current.next
                return True
            prev = current
            current = current.next
        return False

    def search(self, name):
        """Check if a participant is registered."""
        current = self.head
        while current:
            if current.name == name:
```

```python
    def display(self):
        """Display all registered participants."""
        participants = []
        current = self.head
        while current:
            participants.append(current.name)
            current = current.next
        print("Registered Participants:", participants)
        return participants

# ------------------ ✅ Test Cases ------------------

def test_event_registration():
    event = EventRegistration()
    event.register("Alice")
    event.register("Bob")
    event.register("Charlie")

    # Test search
    assert event.search("Bob") == True
    assert event.search("Daisy") == False

    # Test remove
    assert event.remove("Bob") == True
    assert event.remove("Daisy") == False

    # Test display
    assert event.display() == ["Alice", "Charlie"]

test_event_registration()
print("All test cases passed.")
```

OUTPUT:

```
PS C:\Users\NIKHITHA\OneDrive\Desktop\AI> python -u "c:\Users\NIKHITHA\OneDrive\Desktop\AI\LAB11.2\task10
Registered Participants: ['Alice', 'Charlie']
All test cases passed.
PS C:\Users\NIKHITHA\OneDrive\Desktop\AI> 
```

Ln 58, Col 1   Space