Assignment 11.1

N SOURISH

2303A52360

Batch: 44

Task 1: Stack Implementation

Code:

```python
class Stack:
    """
    A simple implementation of a Stack data structure using a Python list.

    Supports the following operations:
    - push(item): Add an item to the top of the stack
    - pop(): Remove and return the top item
    - peek(): Return the top item without removing it
    - is_empty(): Check whether the stack is empty
    """

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

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

        :param item: The element to be added
        """
        self.items.append(item)

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

        :return: The top element
        :raises IndexError: If the stack is 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.

        :return: The top element
        :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 whether the stack is empty.

        :return: True if stack is empty, False otherwise
        """
        return len(self.items) == 0
# Example usage:
if __name__ == "__main__":
    stack = Stack()
    stack.push(1)
    stack.push(2)
    stack.push(3)

    print(stack.peek())  # Output: 3
    print(stack.pop())   # Output: 3
    print(stack.pop())   # Output: 2
    print(stack.is_empty())  # Output: False
    print(stack.pop())   # Output: 1
    print(stack.is_empty())  # Output: True
```

Output:

```
3
3
2
False
1
True
```

Task 2: Queue Implementation

Code:

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

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

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

    def peek(self):
        if not self.items:
            raise IndexError("peek from empty queue")
        return self.items[0]

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

# Example usage:
if __name__ == "__main__":
    queue = Queue()
    queue.enqueue(1)
    queue.enqueue(2)
    queue.enqueue(3)

    print(queue.peek())  # Output: 1
    print(queue.dequeue())  # Output: 1
    print(queue.dequeue())  # Output: 2
    print(queue.size())  # Output: 1
    print(queue.dequeue())  # Output: 3
    print(queue.size())  # Output: 0
```

Output:

```
1
1
2
1
3
0
```

Task 3: Linked List

Code:

```
class Node:
    """Represents one element in a singly linked list."""

    def __init__(self, data):
        """Create a node with the given value and no next node."""
        self.data = data
        self.next = None


class LinkedList:
    """Singly linked list with insert and display operations."""

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

    def insert(self, data):
        """Insert a new node containing `data` at the end of the list."""
        new_node = Node(data)

        if self.head is None:
            self.head = new_node
            return

        current = self.head
        while current.next is not None:
            current = current.next
        current.next = new_node
```

```
    def display(self):
        """Return a string showing all nodes in order using arrows."""
        values = []
        current = self.head

        while current is not None:
            values.append(str(current.data))
            current = current.next

        return " -> ".join(values) if values else "Empty Linked List"

# Example usage:
if __name__ == "__main__":
    linked_list = LinkedList()
    linked_list.insert(10)
    linked_list.insert(20)
    linked_list.insert(30)

    print(linked_list.display())
```

Output:

```
10 -> 20 -> 30
```

Task 4: Binary Search Tree (BST)

Code:

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None


class BST:
    def __init__(self):
        self.root = None

    def insert(self, value):
        self.root = self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        if node is None:
            return Node(value)

        if value < node.value:
            node.left = self._insert_recursive(node.left, value)
        elif value > node.value:
            node.right = self._insert_recursive(node.right, value)

        return node

    def inorder(self):
        result = []
        self._inorder_recursive(self.root, result)
        return result
```

```
    def _inorder_recursive(self, node, result):
        if node is None:
            return

        self._inorder_recursive(node.left, result)
        result.append(node.value)
        self._inorder_recursive(node.right, result)


if __name__ == "__main__":
    bst = BST()
    for number in [50, 30, 70, 20, 40, 60, 80]:
        bst.insert(number)

    print("In-order traversal:", bst.inorder())
```

Output:

```
In-order traversal: [20, 30, 40, 50, 60, 70, 80]
```

Task 5: Hash Table

Code:

```
class HashTable:
    def __init__(self, size=10):
        """Initialize hash table with fixed-size buckets for chaining."""
        self.size = size
        self.buckets = [[] for _ in range(size)]

    def _hash(self, key):
        """Compute bucket index for a given key."""
        return hash(key) % self.size

    def insert(self, key, value):
        """
        Insert or update a key-value pair.
        Collision handling uses chaining (list of pairs in each bucket).
        """
        index = self._hash(key)
        bucket = self.buckets[index]

        for item in bucket:
            if item[0] == key:
                item[1] = value
                return

        bucket.append([key, value])

    def search(self, key):
        """Return value for key if found, otherwise return None."""
        index = self._hash(key)
        bucket = self.buckets[index]

        for stored_key, stored_value in bucket:
            if stored_key == key:
                return stored_value
        return None
```

```
    def delete(self, key):
        """Delete key-value pair if present and return True; otherwise return False."""
        index = self._hash(key)
        bucket = self.buckets[index]

        for i, (stored_key, _) in enumerate(bucket):
            if stored_key == key:
                del bucket[i]
                return True

        return False


if __name__ == "__main__":
    table = HashTable(size=5)

    table.insert("name", "Sam")
    table.insert("age", 21)
    table.insert("city", "Hyderabad")

    print("Search 'name':", table.search("name"))
    print("Search 'age':", table.search("age"))
    print("Search 'country':", table.search("country"))

    print("Delete 'age':", table.delete("age"))
    print("Search 'age' after delete:", table.search("age"))
```

Output:

```
Search 'name': sourish
Search 'age': 20
Search 'country': None
Delete 'age': True
Search 'age' after delete: None
```

Task 6: Graph Representation

Code:

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

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

    def add_edge(self, source, destination):
        self.add_vertex(source)
        self.add_vertex(destination)

        if destination not in self.adjacency_list[source]:
            self.adjacency_list[source].append(destination)
        if source not in self.adjacency_list[destination]:
            self.adjacency_list[destination].append(source)

    def display_connections(self):
        for vertex, neighbors in self.adjacency_list.items():
            connected_to = ", ".join(str(neighbor) for neighbor in neighbors)
            print(f"{vertex} -> {connected_to}")


if __name__ == "__main__":
    graph = Graph()

    graph.add_vertex("A")
    graph.add_vertex("B")
    graph.add_vertex("C")

    graph.add_edge("A", "B")
    graph.add_edge("A", "C")
    graph.add_edge("B", "C")

    graph.display_connections()
```

Output:

```
A -> B, C
B -> A, C
C -> A, B
```

Task 7: Priority Queue

Code:

```
import heapq

class PriorityQueue:
    def __init__(self):
        self._heap = []

    def enqueue(self, item, priority):
        heapq.heappush(self._heap, (-priority, item))

    def dequeue(self):
        if not self._heap:
            return None
        _, item = heapq.heappop(self._heap)
        return item

    def display(self):
        ordered = sorted(self._heap)
        for negative_priority, item in ordered:
            print(f"{item} (priority={-negative_priority})")


if __name__ == "__main__":
    pq = PriorityQueue()

    pq.enqueue("Low", 1)
    pq.enqueue("High", 5)
    pq.enqueue("Medium", 3)

    print("Queue contents:")
    pq.display()

    print("Dequeued:", pq.dequeue())
    print("Dequeued:", pq.dequeue())
    print("Dequeued:", pq.dequeue())
    print("Dequeued:", pq.dequeue())
```

Output:

```
Queue contents:
High (priority=5)
Medium (priority=3)
Low (priority=1)
Dequeued: High
Dequeued: Medium
Dequeued: Low
Dequeued: None
```

Task 8: Deque

Code:

```
from collections import deque
class DequeDS:
    """Double-ended queue data structure using collections.deque."""

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

    def insert_front(self, value):
        """Insert a value at the front (left side) of the deque."""
        self.items.appendleft(value)

    def insert_rear(self, value):
        """Insert a value at the rear (right side) of the deque."""
        self.items.append(value)

    def remove_front(self):
        """Remove and return the front value, or None if deque is empty."""
        if not self.items:
            return None
        return self.items.popleft()

    def remove_rear(self):
        """Remove and return the rear value, or None if deque is empty."""
        if not self.items:
            return None
        return self.items.pop()

    def display(self):
        """Return a list representation of current deque elements."""
        return list(self.items)

if __name__ == "__main__":
    dq = DequeDS()

    dq.insert_front(20)
    dq.insert_front(10)
    dq.insert_rear(30)
    dq.insert_rear(40)

    print("Deque after insertions:", dq.display())
    print("Removed from front:", dq.remove_front())
    print("Removed from rear:", dq.remove_rear())
    print("Deque now:", dq.display())
```

Output:

```
Deque after insertions: [10, 20, 30, 40]
Removed from front: 10
Removed from rear: 40
Deque now: [20, 30]
```

Task 9: Real time Application Challenge

Code:

```
class HashTable:
    """Hash table with chaining to support fast insert, search, and delete."""

    def __init__(self, size=101):
        """Initialize bucket array where each bucket stores [key, value] pairs."""
        self.size = size
        self.buckets = [[] for _ in range(size)]

    def _index(self, key):
        """Compute array index for a given key."""
        return hash(key) % self.size

    def insert(self, key, value):
        """Insert a key-value pair, or update the value if key already exists."""
        index = self._index(key)
        bucket = self.buckets[index]

        for pair in bucket:
            if pair[0] == key:
                pair[1] = value
                return

        bucket.append([key, value])

    def search(self, key):
        """Return the value for key if found; otherwise return None."""
        index = self._index(key)
        bucket = self.buckets[index]

        for stored_key, stored_value in bucket:
            if stored_key == key:
                return stored_value
        return None

    def delete(self, key):
        """Delete a key-value pair if present and return True; else return False."""
        index = self._index(key)
        bucket = self.buckets[index]

        for position, (stored_key, _) in enumerate(bucket):
            if stored_key == key:
                del bucket[position]
                return True
        return False
```

```python
class EventRegistrationSystem:
    """Manage event participants with quick add, search, and removal by student ID."""

    def __init__(self):
        """Initialize registration store."""
        self.registrations = HashTable()

    def register_participant(self, student_id, student_name):
        """Add or update a participant record."""
        self.registrations.insert(student_id, student_name)

    def find_participant(self, student_id):
        """Return participant name for given student ID, or None if absent."""
        return self.registrations.search(student_id)

    def remove_participant(self, student_id):
        """Remove participant by student ID and return True/False based on result."""
        return self.registrations.delete(student_id)

    def display_participants(self):
        """Print participants sorted by student ID."""
        participant_list = sorted(self.registrations.items(), key=lambda item: item[0])
        if not participant_list:
            print("No participants registered.")
            return

        print("Registered Participants:")
        for student_id, student_name in participant_list:
            print(f"{student_id}: {student_name}")


if __name__ == "__main__":
    system = EventRegistrationSystem()

    system.register_participant("S101", "Aarav")
    system.register_participant("S102", "Diya")
    system.register_participant("S103", "Rahul")

    system.display_participants()
    print("\nFind S102:", system.find_participant("S102"))
    print("Remove S101:", system.remove_participant("S101"))
    print("Find S101:", system.find_participant("S101"))
    print()
    system.display_participants()
```

Output:

```
Registered Participants:
S101: Aarav
S102: Diya
S103: Rahul

Find S102: Diya
Remove S101: True
Find S101: None

Registered Participants:
S102: Diya
S103: Rahul
```

Task 10: Smart E-Commerce Platform

Code:

```
class ProductHashTable:
    """Hash table with chaining for product records keyed by product ID."""

    def __init__(self, size=101):
        """Create bucket array; each bucket is a list of [key, value] pairs."""
        self.size = size
        self.buckets = [[] for _ in range(size)]

    def _hash(self, product_id):
        """Return bucket index for a given product ID."""
        return hash(product_id) % self.size

    def insert(self, product_id, details):
        """Insert new product or update existing product details."""
        index = self._hash(product_id)
        bucket = self.buckets[index]

        for pair in bucket:
            if pair[0] == product_id:
                pair[1] = details
                return

        bucket.append([product_id, details])

    def search(self, product_id):
        """Find product details by ID and return None if not present."""
        index = self._hash(product_id)
        bucket = self.buckets[index]

        for stored_id, details in bucket:
            if stored_id == product_id:
                return details
        return None

    def delete(self, product_id):
        """Delete product by ID and return True if removed, else False."""
        index = self._hash(product_id)
        bucket = self.buckets[index]

        for i, (stored_id, _) in enumerate(bucket):
            if stored_id == product_id:
                del bucket[i]
                return True
        return False
```

```python
class ProductSearchEngine:
    """Fast product search system powered by a hash table."""

    def __init__(self):
        """Initialize underlying storage for product data."""
        self.products = ProductHashTable()

    def add_product(self, product_id, name, price, stock):
        """Add or update a product using product ID as key."""
        self.products.insert(
            product_id,
            {
                "name": name,
                "price": price,
                "stock": stock,
            },
        )

    def find_product(self, product_id):
        """Return product details for a given product ID."""
        return self.products.search(product_id)

    def remove_product(self, product_id):
        """Remove a product from the index by product ID."""
        return self.products.delete(product_id)

    def show_product(self, product_id):
        """Print product details in a user-friendly format."""
        product = self.find_product(product_id)
        if product is None:
            print(f"Product {product_id} not found.")
            return

        print(
            f"{product_id} -> Name: {product['name']}, "
            f"Price: ₹{product['price']}, Stock: {product['stock']}"
        )
```

```python
if __name__ == "__main__":
    engine = ProductSearchEngine()

    # Add products into the search engine.
    engine.add_product("P1001", "Wireless Mouse", 799, 45)
    engine.add_product("P1002", "Mechanical Keyboard", 2499, 30)
    engine.add_product("P1003", "USB-C Charger", 1199, 60)

    # Fast lookup by product ID.
    print("Search P1002:")
    engine.show_product("P1002")

    # Delete and verify removal.
    print("\nRemoving P1001:", engine.remove_product("P1001"))
    print("Search P1001 after deletion:")
    engine.show_product("P1001")
```

Output:

```
Search P1002:
P1002 -> Name: Mechanical Keyboard, Price: ₹2499, Stock: 30

Removing P1001: True
Search P1001 after deletion:
Product P1001 not found.
```