University Of Cross River State
Department Of Computer Science
First Semester Examinations, 2022/23 Session

CSCS 3105: Object-Oriented Programming

Evergreen Dynamics - Group 26

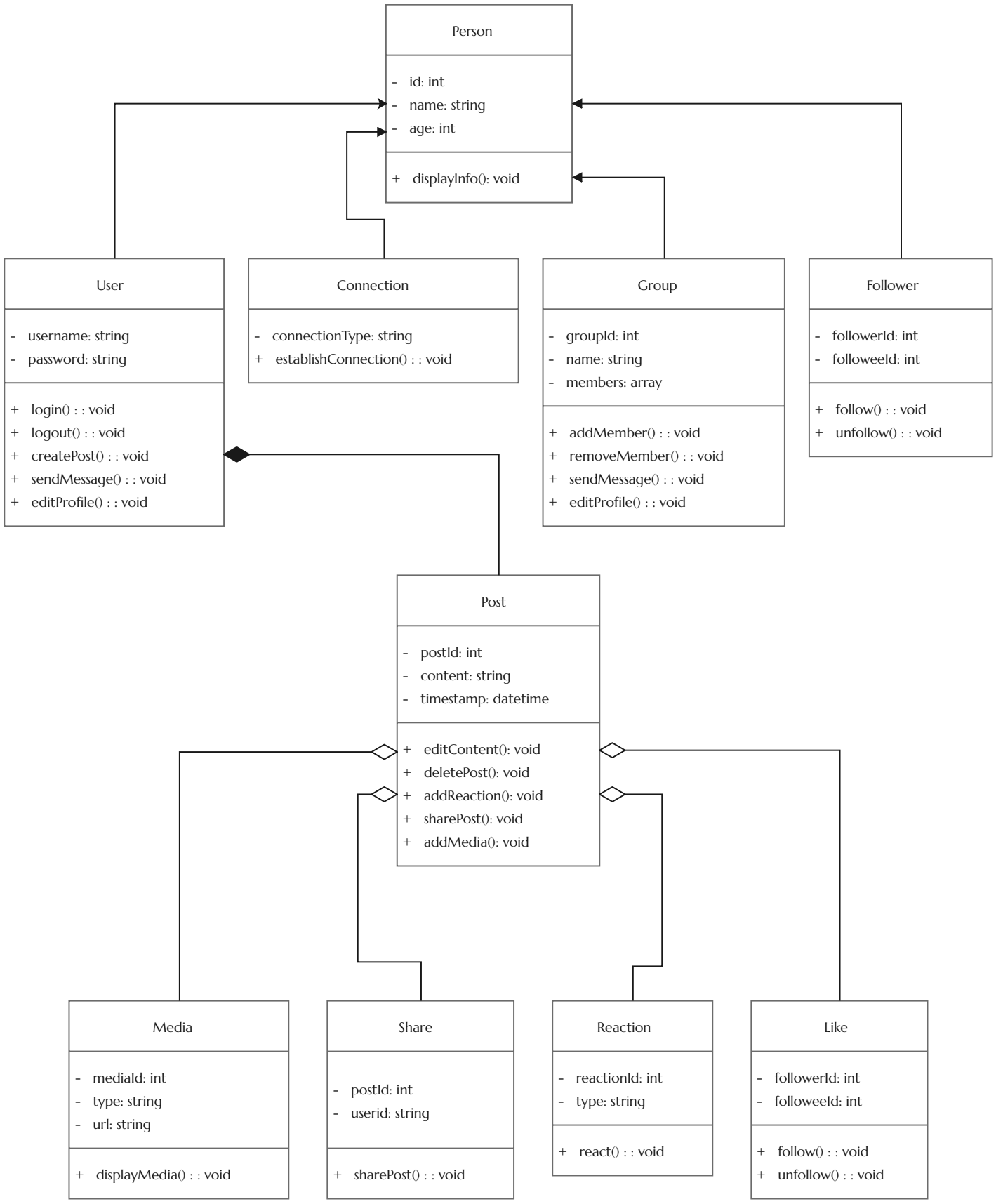
Project 1

(A) UML Class Diagram - Next page

(B) Explanation

S/No.	Concept A	Concept B	Relationship	Explanation
1	Person	User	Inheritance	A User is a specific type of Person, inheriting properties and behaviors of the Person class.
2	Person	Connection	Inheritance	A Connection is a specific type of Person, representing a connection or relationship with another person.
3	Person	Group	Inheritance	A Group is a specific type of Person, representing a group of individuals.
4	Person	Follower	Association	A Person can have multiple followers, establishing an association between them.
5	Post	Media	Composition	A Post may contain Media, forming a composition relationship where Media is a part of a Post.
6	Post	Share	Association	A Post can be shared by multiple users, establishing an association.
7	Post	Reaction	Association	A Post can have multiple reactions from users, forming an association.
8	Post	Like	Association	A Post can be liked by multiple users, forming an association.

(A) UML Class Diagram



Project 2

Data structures used in programming can be broadly categorized into several types:

Arrays:
Arrays are a fundamental data structure consisting of a collection of elements stored at contiguous memory locations. They allow for constant-time access to elements by index.

Lists:
Lists are dynamic data structures that allow for the storage of elements of varying types and sizes. They include linked lists, which use pointers to connect elements, and resizable arrays, which dynamically adjust their size as elements are added or removed.

Stacks:
Stacks are a type of linear data structure that follows the Last In, First Out (LIFO) principle, where elements are inserted and removed from the same end. They are commonly used in algorithms involving recursion, parsing, and expression evaluation.

Queues:
Queues are another type of linear data structure that follows the First In, First Out (FIFO) principle, where elements are inserted at the rear and removed from the front. They are useful for modeling scenarios such as task scheduling and breadth-first search algorithms.

Trees:
Trees are hierarchical data structures composed of nodes connected by edges. They include binary trees, binary search trees, AVL trees, and red-black trees, among others. Trees are widely used for representing hierarchical relationships and searching efficiently.

Graphs:
Graphs are non-linear data structures consisting of vertices (nodes) and edges that connect them. They are used to model complex relationships between objects, such as social networks, transportation networks, and dependency graphs.

Hash tables:
Hash tables are data structures that store key-value pairs, allowing for constant-time insertion, deletion, and lookup of elements. They use a hash function to map keys to indices in an array, providing efficient access to data.

Sets:
Sets are collections of unique elements that do not allow duplicates. They are

useful for performing mathematical operations such as union, intersection, and difference.

Maps (or dictionaries):
Maps are data structures that store key-value pairs, where each key is associated with a value. They provide efficient lookup and retrieval of values based on keys.

Heaps:
Heaps are specialized tree-based data structures that satisfy the heap property, where the value of each node is greater than or equal to (or less than or equal to) the values of its children. They are commonly used for implementing priority queues and heap sort algorithms.

Below are the implementations of all the data structures mentioned, separated by paragraphs and commented for clarity.
Written in Python

```python
# Arrays
arr = [1, 2, 3, 4, 5]

# Lists
my_list = [1, 2, 3, 4, 5]
my_list.append(6)  # Add element to end of list
my_list.insert(0, 0)  # Insert element at index 0
my_list.pop()  # Remove and return last element

# Stacks
class Stack:
    def __init__(self):
        self.items = []

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

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
```

```
        else:
            return None

    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        else:
            return None

    def is_empty(self):
        return len(self.items) == 0

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

# Example usage:
stack = Stack()
stack.push(1)
stack.push(2)
stack.push(3)
print(stack.pop())  # Output: 3
print(stack.peek())  # Output: 2
print(stack.size())  # Output: 2

# Queues
from collections import deque

queue = deque()
queue.append(1)  # enqueue
queue.append(2)
queue.append(3)
print(queue.popleft())  # dequeue - Output: 1
print(queue[0])  # Peek front element - Output: 2

# Trees
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

# Example of creating a binary tree
root = TreeNode(1)
root.left = TreeNode(2)
```

```
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)

# Graphs
class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
        self.graph[u].append(v)
        self.graph[v].append(u)

# Example usage:
g = Graph()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 2)
g.add_edge(2, 3)

# Hash tables
hash_table = {}

# Insert key-value pair
hash_table["key1"] = "value1"
hash_table["key2"] = "value2"

# Retrieve value by key
print(hash_table["key1"])  # Output: value1

# Sets
my_set = {1, 2, 3, 4, 5}
my_set.add(6)  # Add element to set
my_set.remove(3)  # Remove element from set

# Heaps
import heapq

heap = []
heapq.heappush(heap, 1)
```

```
heapq.heappush(heap, 2)
heapq.heappush(heap, 3)
print(heapq.heappop(heap))  # Output: 1
```

Conclusion:
Data structures are fundamental building blocks in computer science and programming. They provide a way to organize and store data efficiently, enabling faster access, insertion, deletion, and manipulation of data. Understanding data structures is essential for developing efficient algorithms and writing effective programs.

Different data structures have different strengths and weaknesses, making them suitable for different use cases. For example, arrays are efficient for random access but may be inefficient for inserting or deleting elements in the middle. Linked lists, on the other hand, excel at insertion and deletion but may have slower access times.

By choosing the appropriate data structure for a given problem, developers can optimize the performance of their programs and improve code readability and maintainability. Additionally, understanding data structures allows developers to better understand and analyze algorithms and problem-solving techniques.