ASSESSMENT - 04

Data Structures and Design Patterns

SECTION 1 - MCQ

1. Which of these best describes an array?
   B. Container of objects of similar types
   (An array is a data structure consisting of a collection of elements of the same type)[1].
2. In a stack, if a user tries to remove an element from an empty stack it is called _____
   a) Underflow
   (Stack underflow occurs when trying to pop from an empty stack)[2].
3. A linear list of elements in which deletion can be done from one end (front) and insertion can take place only at the other end (rear) is known as _____
   A. Queue
   (Queue is FIFO: insertion at rear, deletion at front)[3].
4. Which of the following sorting algorithms can be used to sort a random linked list with minimum time complexity?
   D. Merge Sort
   (Merge sort is best suited for linked lists due to less random memory access)[4].
5. What is a dequeue?
   A. A queue with insert/delete defined for both front and rear ends of the queue
   (Deque allows insertion and deletion at both ends)[5].
6. What is the order of a matrix?
   A. number of rows X number of columns
   (Order of a matrix is m × n where m = rows and n = columns)[6].
7. In a max-heap, element with the greatest key is always in the which node?
   C. root node
   (The root node contains the greatest element in a max-heap)[7].
8. In a simple graph, the number of edges is equal to twice the sum of the degrees of the vertices.
   B. False
   (The sum of degrees of vertices is twice the number of edges, not the other way around).

9. Which of the following sorting algorithms is of Divide and Conquer Type?
   C. Quick Sort
   (Quick Sort is a divide and conquer algorithm).
10. Which of following algorithms scans the list by swapping the entries whenever pair of adjacent keys are out of desired orders?
    D. Bubble Sort
    (Bubble sort swaps adjacent elements if out of order).
11. Which of the below is not a valid classification of design pattern?
    D. Java patterns
    (Creational, Structural, and Behavioural are valid design pattern categories, Java patterns is not).
12. Which design pattern provides a single class which provides simplified methods required by client and delegates call to those methods?
    C. Facade pattern
    (Facade pattern provides a simplified interface to a complex system).
13. Which design pattern ensures that only one object of particular class gets created?
    A. Singleton pattern
    (Singleton pattern restricts instantiation to one object).
14. Which design pattern suggests multiple classes through which request is passed and multiple but only relevant classes carry out operations on the request?
    B. Chain of responsibility pattern
    (Request is passed along a chain until handled).
15. Which design pattern represents a way to access all the objects in a collection?
    A. Iterator pattern
    (Iterator pattern provides sequential access to elements).
16. Is design pattern a logical concept.
    True
    (Design patterns are conceptual solutions).
17. Which of the following is correct about Abstract Factory design pattern.
    D. All the Above
    (It is a creational pattern, works around a super-factory creating other factories, and uses interfaces to create related objects)[general knowledge].
18. Which of the following describes the Adapter pattern correctly?
    C. This pattern works as a bridge between two incompatible interfaces.
    (Adapter converts one interface to another).
19. Which of the following describes the Flyweight pattern correctly?
    C. This pattern is primarily used to reduce the number of objects created and to

decrease memory footprint and increase performance.
(Flyweight reduces memory usage by sharing objects).

20. Which of the following describes the Memento pattern correctly?
C. This pattern is used to restore state of an object to a previous state.
(Memento stores and restores object state).

SECTION 2 - Predict the output

1. Predict the output for the following snippet of Code.

```
Push(1);

Pop();

Push(2);

Push(3);

Pop();

Push(4);

Pop();

Pop();

Push(5);
```

2.

```
public Object delete_key()
{
   if(count == 0)
   {
      System.out.println("Q is empty");
      System.exit(0);
   }
   else
   {
      Node cur = head.getNext();
      Node dup = cur.getNext();
```

```
        Object e = cur.getEle();
        head.setNext(dup);
        count--;
        return e;
    }
}
```

3.

Consider the following doubly linked list: head-1-2-3-4-5-tail. What will be the list after performing the given sequence of operations?

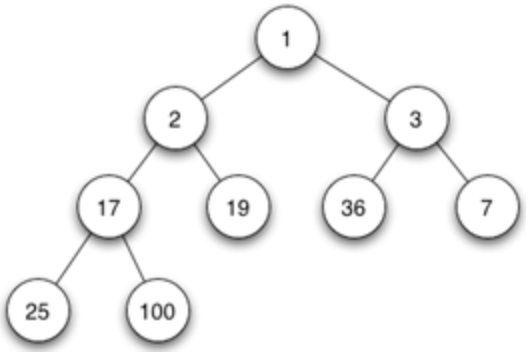
```
    Node temp = new Node(6,head,head.getNext());
    Node temp1 = new Node(0,tail.getPrev(),tail);
    head.setNext(temp);
    temp.getNext().setPrev(temp);
    tail.setPrev(temp1);
    temp1.getPrev().setNext(temp1);
```

4. After performing these set of operations, what does the final list look contain?

```
InsertFront(10);
InsertFront(20);
InsertRear(30);
DeleteFront();
InsertRear(40);
InsertRear(10);
DeleteRear();
InsertRear(15);
display();
```

5.

If we implement heap as min-heap, deleting root node (value 1)from the heap. What would be the value of root node after the second iteration if leaf node (value 100) is chosen to replace the root at start.



Section III [Descriptive Questions]: 5 Marks Each

1. What is a Data Structure? Provide a brief Overview of Data Structures.

Data is a collection of facts and figures. It is a set of values of specific format that refers to a single set of items. A data is an information that is stored or transmitted.

A Data Structure is a storage that is used to store and organize data. It is a way to arrange data so that it can be accessed easy and efficiently. A data structure is not only used for organizing the data but also used for processing, storing and retrieving of data and information. There are two types of data structured - 1) Primitive 2) Non primitive

A) Primitive data structures are basic types provided by programming languages like int, float, char, and Boolean. They store single values. Primitive types are typically stored in stack memory, making access fast and efficient. They support basic arithmetic and logical operations and have fixed sizes. Primitive types are used for simple data representation, while non-primitive types organize and manage complex data.

B) Non-primitive data structures are complex and user-defined, such as arrays, lists, stacks, and trees. They are categorized into 2 types - Linear and Non - linear. Linear data structures include Arrays, List, Stack, Queues. Non-Linear includes Graphs and trees. Non- primitive types are generally stored in heap memory, allowing dynamic size and flexibility. They can store multiple values, often of different types, and are mutable.

They have special inbuilt methods for data access and manipulation. Non-primitive data structures enable advanced operations and are essential for handling large or dynamic datasets.

2. Explain Searching algorithms and Sorting Algorithms with suitable sample illustrations

Searching algorithms:
1) Linear Search - Linear search is defined as a sequential search algorithm that starts at one end and goes until the element is found, or the iteration continues till the end of the data structure and terminates.

```
Linear_Search(a, key):
    for i from 0 to length(a) - 1:
        if a[i] == key:
            return i
    return -1
```

2) Binary search - Works on sorted array, if not the array has to be sorted first. The array is divided into equal half's on each iteration. Each part has a top element, mid element and bottom element. The target value is compared with the mid element id present return. Else If the target is smaller then, bottom is updated to mid-1 and if target is greater then, top is updated to mid+1

```
Binary_Search(array, target)
    top = 0
    bottom = length(a) - 1
    while top <= bottom
        mid = (top + bottom) / 2
        if a[mid] == target
            return mid
        else if a[mid] < target
            top = mid + 1
```

```
        else
            bottom = mid - 1
    return -1
```

Sorting Algorithm:

1) Bubble Sort - Bubble sort is a simple sorting algorithm that repeatedly steps through a list, compares adjacent elements, and swaps them if they are in the wrong order. This process continues until no swaps are needed, meaning the list is sorted. The name comes from how larger elements "bubble" to the top (end) of the list with each pass. O(n2) time complexity.

```
procedure BubbleSort(A)
    n = length(A)
    repeat
        swapped = false
        for i = 1 to n - 1
            if A[i - 1] > A[i] then
                swap A[i - 1] and A[i]
                swapped = true
            end if
        end for
        n = n - 1
    until not swapped
```

2) Insertion sort - Insertion sort is a simple sorting algorithm that builds the sorted list one element at a time. It works by taking each element from the unsorted portion and inserting it into its correct position in the sorted portion by comparing it with elements to its left and shifting those larger than it to the right. This process repeats until all elements are sorted. O(n2) time complexity

```
InsertionSort(a)
    for i = 1 to length(a) - 1
        key = a[i]
```

```
    j = i - 1
    while j >= 0 and a[j] > key
        a[j + 1] = a[j]
        j = j - 1
    a[j + 1] = key
```

3) Selection sort - Selection sort is a simple sorting algorithm that divides the list into a sorted and unsorted part. It repeatedly selects the smallest (or largest) element from the unsorted part and swaps it with the first element of the unsorted part, growing the sorted portion one element at a time. O9n2) time complexity.

```
SelectionSort(a)
    n = length(a)
    for i = 0 to n - 2
        minIndex = i
        for j = i + 1 to n - 1
            if a[j] < a[minIndex]
                minIndex = j
        if minIndex != i
            swap a[i] and a[minIndex]
```

3. What are Creational Design Patterns used for? Discuss the types of it and provide sample illustrations for any 2 types of your choice.

Creational Design Patterns provide solutions to object creation problems in software design. They abstract the instantiation process, making a system independent of how its objects are created, composed, and represented. This promotes flexibility, reusability, and maintainability.

1) Singleton Pattern - Ensures that a class has only one instance throughout the application and provides a global point of access to it.

Working - The class has a private static variable holding the single instance, a private constructor to prevent external instantiation, and a public static method (e.g., getInstance()) to return the instance.
Use case: Managing shared resources like database connections or logging.

2) Factory Pattern - Defines an interface for creating objects but lets subclasses decide which class to instantiate. This promotes loose coupling by hiding the object creation logic from the client.
Working - The client calls a factory method with parameters, and the factory returns the appropriate object based on input or context.
Use case: Creating objects of different types that share a common interface, such as different types of documents, shapes, or vehicles.

3) Builder Pattern - Separates the construction of a complex object from its representation so that the same construction process can create different representations.
Working - Uses a director class to control the building steps, while builder classes implement the actual construction. The client gets the final product without worrying about the building process.
Use case: Creating complex objects like a customizable house, meal, or computer configuration.

4) Prototype Pattern - Creates new objects by cloning an existing object (prototype) instead of instantiating new ones. This is efficient when object creation is expensive.
Working - The prototype class implements a clone method that returns a copy of itself. The client requests a clone instead of creating a new object.
Use case: When creating many similar objects or when object creation involves costly operations.

4. What are Structural Design Patterns used for? Discuss the types of it and provide sample Illustrations for any 2 types of your choice.

Structural Design Patterns focus on how classes and objects are composed to form larger structures while keeping these structures flexible and efficient. They help manage relationships between entities, enabling easier maintenance, extension, and reuse of code.

Types of Structural Design Patterns:

1) Proxy Pattern - Provides a placeholder or surrogate for another object to control access to it. It implements the same interface as the original object and can add additional behavior like lazy loading, access control, or logging before or after forwarding requests to the real object.

Example: A virtual proxy for an image that loads the actual image data only when required, saving memory and improving performance.

```
interface Image {
    void display();
}

class RealImage implements Image {
    private String filename;

    public RealImage(String filename) {
        this.filename = filename;
        loadFromDisk();
    }

    private void loadFromDisk() {
        System.out.println("Loading " + filename);
    }
```

```
    public void display() {
        System.out.println("Displaying " + filename);
    }
}

class ProxyImage implements Image {
    private RealImage realImage;
    private String filename;

    public ProxyImage(String filename) {
        this.filename = filename;
    }

    public void display() {
        if (realImage == null) {
            realImage = new RealImage(filename);
        }
        realImage.display();
    }
}

public class ProxyDemo {
    public static void main(String[] args) {
        Image image = new ProxyImage("photo.jpg");
        image.display();
        image.display();
    }
}
```

2) Decorator Pattern - Allows adding new behaviors or responsibilities to objects dynamically without altering their structure.

It wraps the original object inside a decorator class that implements the same interface and adds extra functionality.
Example: In a text editor, decorators add features like bold, italic, or underline formatting to text objects without changing their core implementation.

```
interface Drink {
    String getName();
    double getPrice();
}

class Tea implements Drink {
    public String getName() {
        return "Tea";
    }
    public double getPrice() {
        return 10.0;
    }
}

abstract class DrinkDecorator implements Drink {
    protected Drink drink;

    public DrinkDecorator(Drink drink) {
        this.drink = drink;
    }

    public String getName() {
        return drink.getName();
    }
    public double getPrice() {
        return drink.getPrice();
```

```
    }
}

class Lemon extends DrinkDecorator {
    public Lemon(Drink drink) {
        super(drink);
    }

    public String getName() {
        return drink.getName() + " + Lemon";
    }
    public double getPrice() {
        return drink.getPrice() + 2.0;
    }
}

public class DecoratorExample {
    public static void main(String[] args) {
        Drink tea = new Tea();
        System.out.println(tea.getName() + " costs " + tea.getPrice());

        Drink teaWithLemon = new Lemon(tea);
        System.out.println(teaWithLemon.getName() + " costs " +
teaWithLemon.getPrice());
    }
}
```

3) Adapter Pattern - Converts the interface of a class into another interface clients expect, allowing incompatible interfaces to work together.

4) Bridge Pattern - Separates an abstraction from its implementation so the two can vary independently.
5) Composite Pattern - Composes objects into tree structures to represent part-whole hierarchies, letting clients treat individual objects and compositions uniformly.
6) Facade Pattern - Provides a simplified interface to a complex subsystem.
7) Flyweight Pattern - Reduces memory usage by sharing common parts of object state among multiple objects.

5. What are Behavioral Design Patterns used for? Discuss the types of it and provide sample illustrations for any 2 types of your choice.

Behavioral Design Patterns focus on how objects interact and communicate with each other, defining clear responsibilities and improving flexibility in the flow of control and data between objects.
Types of Behavioral patterns include - Chain of responsibility, command, interpreter, Memento, mediator, state, template, Visitor and so on.
1) Observer Pattern - Defines a subscription mechanism to notify multiple objects (observers) about changes in another object (subject). Observers register with the subject. When the subject changes, it notifies all registered observers.

```
interface Observer {
    void update(String message);
}
class Subject {
    private List<Observer> observers = new ArrayList<>();
    private String state;

    public void attach(Observer o) {
        observers.add(o);
    }

    public void setState(String state) {
```

```
        this.state = state;
        notifyAllObservers();
    }

    private void notifyAllObservers() {
        for (Observer o : observers) {
            o.update(state);
        }
    }
}

class ConcreteObserver implements Observer {
    private String name;

    public ConcreteObserver(String name) {
        this.name = name;
    }

    public void update(String message) {
        System.out.println(name + " received update: " + message);
    }
}

public class ObserverDemo {
    public static void main(String[] args) {
        Subject subject = new Subject();

        Observer obs1 = new ConcreteObserver("Observer1");
        Observer obs2 = new ConcreteObserver("Observer2");

        subject.attach(obs1);
```

```
        subject.attach(obs2);

        subject.setState("New State");
    }
}
```

2. Strategy Pattern - Defines a family of algorithms, encapsulates each one, and makes them interchangeable at runtime. The context uses a strategy interface to call the algorithm, and the concrete strategies implement different algorithms.

```
interface Strategy {
    int doOperation(int a, int b);
}

class Add implements Strategy {
    public int doOperation(int a, int b) {
        return a + b;
    }
}

class Multiply implements Strategy {
    public int doOperation(int a, int b) {
        return a * b;
    }
}

class Context {
    private Strategy strategy;

    public Context(Strategy strategy) {
        this.strategy = strategy;
    }
```

```
    public int executeStrategy(int a, int b) {
        return strategy.doOperation(a, b);
    }
}

public class StrategyDemo {
    public static void main(String[] args) {
        Context context = new Context(new Add());
        System.out.println("Add: " + context.executeStrategy(5, 3));

        context = new Context(new Multiply());
        System.out.println("Multiply: " + context.executeStrategy(5, 3));
    }
}
```

3) Chain of Responsibility - Allows a request to pass through a chain of handlers. Each handler decides either to process the request or pass it to the next handler, promoting loose coupling between sender and receiver.

4) Iterator - Provides a way to access elements of a collection sequentially without exposing its underlying representation.

5) Mediator - Defines an object that encapsulates how a set of objects interact, promoting loose coupling by preventing objects from referring to each other explicitly.

6) State - Allows an object to alter its behavior when its internal state changes, appearing to change its class dynamically.