DATA STRUCTURES AND ALGORITHMS

WHAT IS DSA?

DSA stands for Data Structure and Algorithms. It is the foundation of organizing and manipulating data efficiently within software applications.

DSA STAND FOR

DATA STRUCTURES (DS):

These are organized ways to store, manage, and access data. Examples include arrays, linked lists, stacks, queues, trees, and graphs.

ALGORITHMS (A):

These are procedures or sets of instructions used to solve problems or perform operations on data. Examples include sorting, searching, and pathfinding algorithms.

Together, DSA is essential for optimizing the performance and efficiency of applications by reducing time and space complexity.

INTRODUCTION TO ADTS

An Abstract Data Type (ADT) is a theoretical model of a data structure that defines the behavior and operations that can be performed on data without specifying the actual implementation. It includes:

OPERATIONS:

Defined actions, like insert, delete, update, etc.

ENCAPSULATION:

The internal workings are hidden, focusing only on what the ADT does, not how it’s done.

IMPORTANCE

Provides a high-level interface without exposing underlying implementation details.

WHY ADTS MATTER IN SOFTWARE DESIGN

1. Encapsulation: Hiding Implementation and Promoting Modularity
    - Reduced Complexity: Encapsulation hides unnecessary details, reducing cognitive load for developers.
    - Modularity: Code is organized into discrete modules, each with a well-defined purpose. This separation enhances maintenance and debugging by localizing changes to specific modules.
    - Protection: Encapsulation prevents unauthorized access to data, protecting the integrity of the ADT.

WHY ADTS MATTER IN SOFTWARE DESIGN

2. Reusability: Enabling Reusable Components
    - Saves Time and Effort: Well-designed ADTs can be used as-is, saving development resources on new projects.
    - Standardization: Common ADTs (like stacks, queues, and lists) provide a standard way to handle data, making it easier for teams to collaborate and share code.
    - Reliability: Reused components are often well-tested, reducing bugs and improving software quality.

WHY ADTS MATTER IN SOFTWARE DESIGN

3. Flexibility: Allowing Different Implementations Without Changing the Interface
    - Optimized Performance: ADTs allow switching between implementations (e.g., an array-based vs. linked list-based queue) depending on performance needs.
    - Adaptability: As requirements evolve, implementations can be updated without altering the rest of the codebase, minimizing the impact of changes.
    - Scalability: ADTs support scalability by enabling efficient data handling strategies for large datasets, whether through different storage methods, memory management techniques, or optimized algorithms.

COMMON ADTS IN PROGRAMMING

- Stack: LIFO data management.
- Queue: FIFO processing.
- List: Ordered collection with flexible data access.
- Priority Queue: Elements prioritized, not just ordered.

```
Function Call Sequence:
1. main()          --> Current Stack Frame
2. foo(int a)      --> New Stack Frame (contains a, return address)
3. bar(int b)      --> New Stack Frame (contains b, return address)

After bar() completes:
- The bar frame is popped off the stack.

After foo() completes:
- The foo frame is popped off the stack.

Return to main() and resume execution.
+-----------------+
|   Return Addr   |  <-- Top of Stack (for bar)
|       b         |
+-----------------+
|   Return Addr   |  <-- Top of Stack (for foo)
|       a         |
+-----------------+
|   Return Addr   |  <-- Top of Stack (for main)
|     localVar    |
+-----------------+
|      ...        |
|                 |
+-----------------+
```

STACK DATA STRUCTURE

- Definition: A data structure following Last In, First Out (LIFO) principles.
- Analogy: Stack of plates where the last plate placed is the first to be removed.

OPERATIONS OF STACK ADT

- Push: Add an element to the top.
- Pop: Remove an element from the top.
- Peek: View the top element without removing it.
- IsEmpty: Check if the stack is empty.

IMPLEMENTATION OF STACK

- Array-based Stack: Fixed size, limited by array capacity.
- Linked List-based Stack: Dynamic resizing, more memory-efficient.

APPLICATIONS OF STACK

- Function Call Stack: Tracks active functions in recursion.
- Backtracking Algorithms: Useful for parsing expressions, such as balancing parentheses.
- Undo Operations: Implemented in editors and applications.

EXAMPLE

```
class StudentStack {
    private final int maxSize;
    private final Student[] stackArray;
private int top;
```

```
public boolean isEmpty() {
    return top == -1;
}
```

```
ublic void push(Student student) {
    if (isFull()) {
        System.out.println("Stack is full. Can't add more students.");
    } else {
        stackArray[++top] = student;
        System.out.println("Student added: " + student.name);
    }
}

ublic boolean isFull() {
    return top == maxSize - 1;
}
```

```
public Student peek() {
    if (isEmpty()) {
        System.out.println("Stack is empty. No students to display.");
        return null;
    } else {
        return stackArray[top]; // Return the student at the top without
removing it
    }
}
```

QUEUE DATA STRUCTURE

A data structure following First In, First Out (FIFO) principles.A line at a ticket counter where the first person in line is served first.

```java
class ArrayQueue {
    private int[] queue;
    private int front;
    private int rear;
    private int size;
    private int capacity;

    public ArrayQueue(int capacity) {
        this.capacity = capacity;
        this.queue = new int[capacity];
        this.front = 0;
        this.rear = 0;
        this.size = 0;
    }

    public void enqueue(int element) { /* Implementation */ }
    public int dequeue() { /* Implementation */ }
    public boolean isEmpty() { /* Implementation */ }
    public int size() { /* Implementation */ }
}
```

OPERATIONS OF QUEUE ADT

- Enqueue: Add an element to the rear.
- Dequeue: Remove an element from the front.
- Peek: View the front element without removing it.
- IsEmpty: Check if the queue is empty.

TYPES OF QUEUES

- Simple Queue: Basic FIFO structure.
- Circular Queue: Overcomes fixed size limitation by connecting front and rear.
- Priority Queue: Items dequeued based on priority, not just order.

IMPLEMENTATION OF QUEUE

- Array-based Queue: Efficient, but has a fixed size.
- Linked List-based Queue: Dynamic resizing, supports unlimited size.

APPLICATIONS OF QUEUE

- Task Scheduling: Used in operating systems to manage processes.
- Print Spooling: Manages jobs in the printing process.
- Breadth-First Search (BFS): Graph traversal method uses queues.

EXAMPLE

```java
import java.util.NoSuchElementException;

class ArrayQueue {
    private int[] queue;
    private int front, rear, capacity;

    public ArrayQueue(int size) {
        queue = new int[size];
        capacity = size;
        front = 0;
        rear = -1;
    }

    public void enqueue(int item) {
        if (rear < capacity - 1) {
            queue[++rear] = item; // Add item to the end
        } else {
            System.out.println("Queue is full");
        }
    }

    public void dequeue() {
        if (front <= rear) {
            front++; // Remove item from the front
        } else {
            System.out.println("Queue is empty");
```

```java
class LinkedListQueue {
    private class Node {
        Student student;
        Node next;

        public Node(Student student) {
            this.student = student;
            this.next = null;
        }
    }

    private Node front, rear;

    public LinkedListQueue() {
        front = rear = null;
    }

    public void enqueue(Student student) {
        Node newNode = new Node(student);
        if (rear == null) {
            front = rear = newNode;
        } else {
            rear.next = newNode;
            rear = newNode;
        }
    }

    public Student dequeue() {
        if (front == null) {
            throw new RuntimeException("Queue is empty");
        }
```

COMPARING STACK AND QUEUE

DEFINITION

STACKS

A LIFO (Last In, First Out) data structure

Queue

A FIFO (First In, First Out) data structure

COMPARING STACK AND QUEUE

INSERTION (PUSH/ENQUEUE)

STACKS

Adds elements to the top of the stack

Queue

Adds elements to the rear of the queue

COMPARING STACK AND QUEUE

REMOVAL (POP/DEQUEUE)

STACKS

Removes elements from the top of the stack

Queue

Removes elements from the front of the queue

COMPARING STACK AND QUEUE

USE CASES

STACKS

Function call management, undo operations

Queue

Print spooling, scheduling tasks

COMPARING STACK AND QUEUE

OPERATIONS

STACKS

Push (add), Pop (remove), Peek (top)

Queue

Enqueue (add), Dequeue (remove), Peek (front)

COMPARING STACK AND QUEUE

SOME FUN FACT

STACKS

Backtracking Wizards: Stacks are behind the "undo" function in most applications! Every time you make a change, it’s pushed onto a stack. When you hit "undo," the application pops the latest action, going back in time to the previous state.

Queue

Queues in Real Life: Queues mimic real-life lines everywhere! Think about waiting in line at a coffee shop or customer service – the person in front gets served first, just like how elements are dequeued from a data structure.

SORTING ALGORITHMS

- Organizes data for faster access and management.
- Types: Quick Sort, Merge Sort, Bubble Sort, etc.
- Focus Algorithms: Quick Sort and Merge Sort.

COMPARE

QUICKSORT TIME COMPLEXITY

BEST CASE

: O(n log n) (when the pivot divides the array evenly).

AVERAGE CASE

O(n log n).

WORST CASE

$O(n^2)$ (when the smallest or largest element is always chosen as the pivot).

COMPARE

MERGE SORT TIME COMPLEXITY

BEST CASE	AVERAGE CASE	WORST CASE
: O(n log n)	O(n log n).	O(n).

COMPARE

QUICK SORT SPACE COMPLEXITY

BEST CASE	AVERAGE CASE	WORST CASE
: O(log n)	O(log n).	O(n^2)

COMPARE

MERGE SORT SPACE COMPLEXITY

BEST CASE	AVERAGE CASE	WORST CASE
: O(n)	O(n).	O(n²)

COMPARE

QUICK,MERGE SORT

QUICK SORT

Not stable. Equal elements may not retain their relative order after sorting

MERGE SORT

Stable. Equal elements retain their relative order after sorting.

COMPARE

QUICK,MERGE SORT

QUICK SORT

Quick Sort generally performs better in practice due to its smaller constant factors and better cache performance. It's faster for large datasets

MERGE SORT

Merge Sort guarantees O(n log n) performance, making it more reliable for performance-sensitive applications, particularly when stable sorting is required or when working with linked lists.

SHORTEST PATH ALGORITHMS

DEFINITION

The shortest path algorithms are the ones that focuses on calculating the minimum travelling cost from source node to destination node of a graph in optimal time and space complexities.

SHORTEST PATH ALGORITHMS

TWO KIND OF SPA

DIJKSTRA'S ALGORITHM

Description: Greedy algorithm for finding the shortest path with non-negative weights.

Steps:

1.Set initial distances.

2.Visit nodes with the smallest distance.

3.Update neighboring nodes.

Bellman-Ford Algorithm

Description: Works with graphs containing negative weights and detects negative cycles.

Steps:

1.Set initial distances.

2.Relax edges repeatedly.

3.Detect negative cycles.

SHORTEST PATH ALGORITHMS

COMPLEXITY AND USE CASES

DIJKSTRA'S ALGORITHM

Time Complexity: O((V+E)logV)
Space Complexity: O(V).
Applications: GPS navigation, pathfinding in games.

Bellman-Ford Algorithm

Time Complexity:O(V×E)
Space Complexity: .O(V)
Applications: Currency arbitrage, routing with negative weights.

SHORTEST PATH ALGORITHMS

COMPARE

PERFORMANCE

- Dijkstra’s is faster but requires non-negative weights. Bellman-Ford supports negative weights.

Use Cases

- Dijkstra for positive weights; Bellman-Ford when cycles and negative weights are factors.

EXAMPLE

```java
import java.util.*;

class Dijkstra {
    public Map<String, Integer> dijkstra(Map<String, List<Edge>> graph, String start)
{
        Map<String, Integer> distances = new HashMap<>();
        PriorityQueue<Edge> priorityQueue = new
PriorityQueue<>(Comparator.comparingInt(e -> e.weight));

        for (String vertex : graph.keySet()) {
            distances.put(vertex, Integer.MAX_VALUE);
        }
        distances.put(start, 0);
        priorityQueue.add(new Edge(start, 0));

        while (!priorityQueue.isEmpty()) {
            Edge current = priorityQueue.poll();
            for (Edge edge : graph.get(current.vertex)) {
                int newDist = distances.get(current.vertex) + edge.weight;
                if (newDist < distances.get(edge.vertex)) {
                    distances.put(edge.vertex, newDist);
                    priorityQueue.add(new Edge(edge.vertex, newDist));
                }
            }
        }
        return distances;
    }
}
```

```java
import java.util.Arrays;
import java.util.List;

class BellmanFord {
    private final int V;

    public BellmanFord(int v) {
        this.V = v;
    }

    public void bellmanFord(List<Edge> edges, int source) {
        int[] distances = new int[V];
        Arrays.fill(distances, Integer.MAX_VALUE);
        distances[source] = 0;

        for (int i = 1; i < V; i++) {
            for (Edge edge : edges) {
                if (distances[edge.source] != Integer.MAX_VALUE &&
                        distances[edge.source] + edge.weight <
distances[edge.destination]) {
                    distances[edge.destination] = distances[edge.source] +
edge.weight;
                }
            }
        }

        // Check for negative weight cycles
        for (Edge edge : edges) {
            if (distances[edge.source] != Integer.MAX_VALUE &&
                    distances[edge.source] + edge.weight <
distances[edge.destination]) {
                System.out.println("Graph contains negative weight cycle");
                return;
            }
```

CONCLUSION

- The Power of Abstract Data Types (ADTs):
- ADTs like stacks and queues play a foundational role in structuring data and simplifying complex operations in software development.
- By focusing on what operations are possible rather than how they're implemented, ADTs promote modularity, flexibility, and reusability in software design
- Real-World Relevance and Applications:
- Beyond theory, these data structures and algorithms power everyday technologies—from GPS navigation and data processing to system memory management.
- Understanding their core principles enables developers to solve practical problems with greater efficiency and clarity.