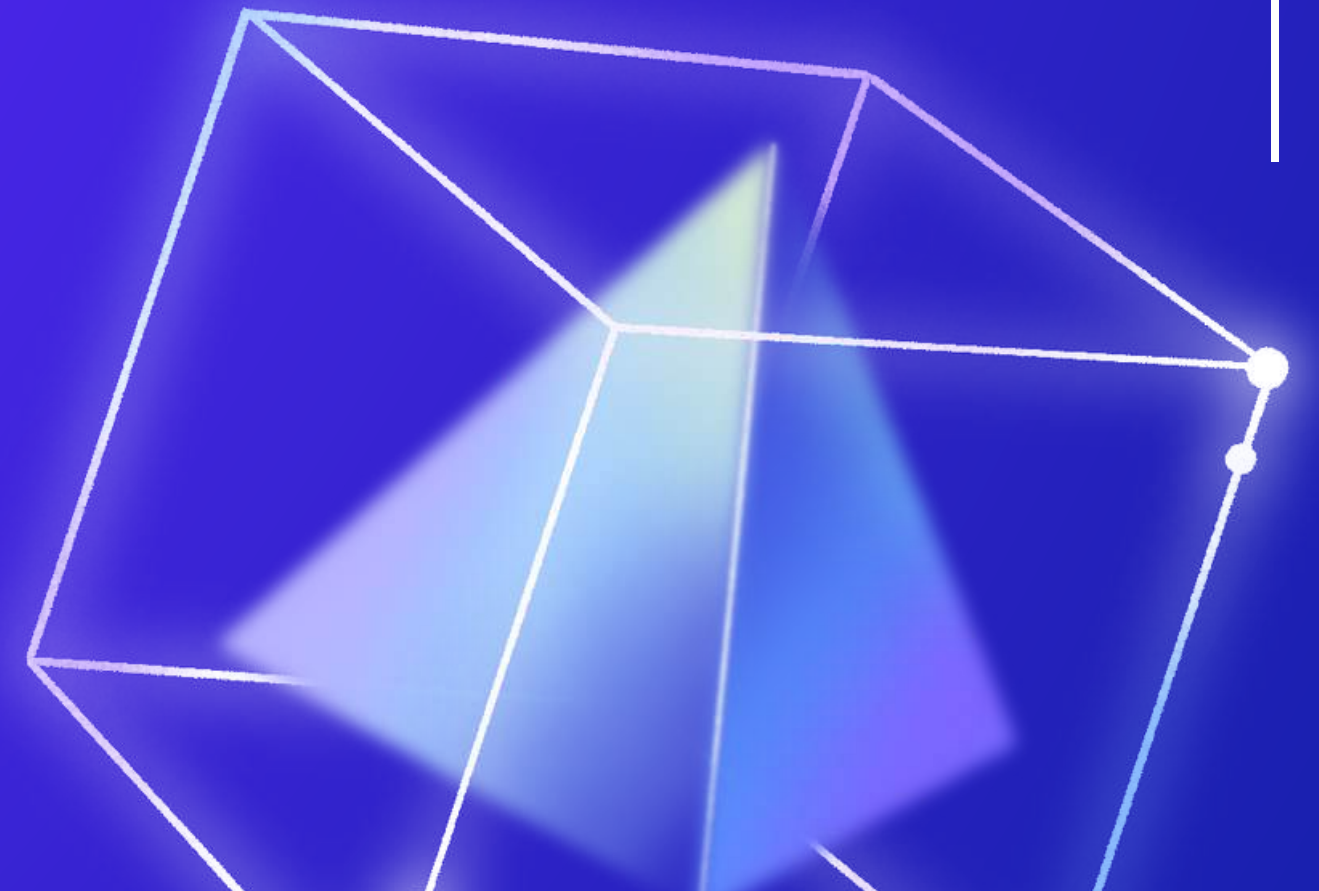




TABLE OF CONTENTS

• Introduction	01
• ADT	03
• Stack	05
• Queue	08
• Compare Stack vs Queue	11
• Sorting compare	12
• Dijkstra compare with Bell-man	18



INTRODUCTION

- **Data Structure Fundamentals:** Data structures are the backbone of software engineering, providing the foundation for how data is stored, accessed, and managed in applications.
- **Why Data Structures Matter:** Efficient data organization is critical to software performance, scalability, and usability. Using the right data structure optimizes processing speed, reduces memory consumption, and enhances user experience.
- **Relevance to Project Goals:** For our Student Management System, appropriate data structures ensure smooth operations like managing student records, handling registration queues, calculating grades, and organizing attendance. Without these, the system risks inefficiencies and potential data inconsistencies.

DATA STRUCTURE AND ALGORITHM



Data Structure

A data structure is a specialized way of organizing, managing, and storing data for efficient access and modification. Different data structures are optimized for different types of operations, making it crucial to select the right one based on the application requirements.

Algorithm

An algorithm is a finite set of well-defined steps or instructions to solve a particular problem.


Algorithms determine the efficiency of how data is processed and manipulated within data structures. A good algorithm reduces resource usage (time, memory), ensuring fast and reliable software.

WHAT IS AN ABSTRACT DATA TYPE (ADT)?

- An Abstract Data Type (ADT) is a mathematical model for data structures that defines the data and the operations that can be performed on the data, independently from its implementation.
- Abstraction: It focuses on what operations are available without detailing how these operations are implemented, providing a higher-level understanding.
- Key Characteristics of ADTs:
- Encapsulation: ADTs hide implementation details, ensuring that users interact with data through well-defined interfaces.
- Modularity: ADTs make code modular, allowing sections to be updated independently, which is useful in large-scale applications.
- Reliability: They ensure that operations on data are safe and consistent, reducing the risk of errors.



UNDERSTANDING THE STACK ABSTRACT DATA TYPE (ADT)

- What is a Stack?
 - A Stack is a linear data structure that follows the Last In, First Out (LIFO) principle, where the last element added is the first to be removed.
 - Often visualized as a physical stack of items (e.g., a stack of books), where only the top item is accessible for operations like addition or removal.
 - Core Operations of a Stack ADT:
 1. Push: Adds an element to the top of the stack.
 2. Pop: Removes and returns the element from the top of the stack.
 3. Peek/Top: Views the element at the top without removing it.
 4. IsEmpty: Checks if the stack contains any elements.
 5. IsFull (optional in fixed-capacity stacks): Checks if the stack has reached its maximum capacity.
- 

USE CASE

USE CASES OF STACK ADT:

FUNCTION CALL MANAGEMENT: IN PROGRAMMING, STACKS HELP TRACK ACTIVE FUNCTIONS THROUGH STACK FRAMES, PARTICULARLY USEFUL IN RECURSION.

UNDO OPERATIONS: IN APPLICATIONS (E.G., TEXT EDITORS), STACKS STORE PREVIOUS ACTIONS FOR UNDO FUNCTIONALITY.

EXPRESSION EVALUATION: USED IN PARSING EXPRESSIONS, STACKS MANAGE OPERATORS AND OPERANDS TO EVALUATE EXPRESSIONS CORRECTLY.



EXAMPLE

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

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

```
public 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);
    }
}

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


UNDERSTANDING THE QUEUE ABSTRACT DATA TYPE (ADT)

What is a Queue?

A Queue is a linear data structure that follows the First In, First Out (FIFO) principle, where the first element added is the first to be removed.

Think of a queue as a line at a ticket counter—people are served in the same order they arrive.

Core Operations of a Queue ADT:

- Enqueue: Adds an element to the back of the queue.
- Dequeue: Removes and returns the element from the front of the queue.
- Front/Ppeek: Views the element at the front without removing it.
- IsEmpty: Checks if the queue has no elements.
- IsFull (optional in fixed-capacity queues): Checks if the queue has reached its maximum capacity.

Types of Queues:

- Linear Queue: Basic queue where elements are processed in FIFO order.
- Circular Queue: Connects the front and rear to manage overflow efficiently in a fixed-size queue.
- Priority Queue: Elements are processed based on priority, not strictly by order of arrival.
- Deque (Double-Ended Queue): Allows insertion and removal from both ends.

QUEUE IN OUR STUDENT MANAGEMENT SYSTEM APPLICATION:

- IN OUR STUDENT MANAGEMENT SYSTEM, A QUEUE CAN EFFICIENTLY MANAGE TASKS LIKE STUDENT REGISTRATION PROCESSING OR HANDLING SUPPORT REQUESTS.
- EXAMPLE: IF STUDENTS SUBMIT MULTIPLE INQUIRIES, EACH REQUEST CAN BE PLACED IN A QUEUE, ENSURING THAT THE EARLIEST INQUIRIES ARE ADDRESSED FIRST, SUPPORTING FAIR, ORGANIZED SERVICE.



EXAMPLE

```
void enqueue(int element) {  
    if (!isFull()) {  
        queue[rear++] = element;  
    }  
}
```

```
int dequeue() {  
    if (!isEmpty()) {  
        return queue[front++];  
    }  
}
```


COMPARE STACK VS QUEUE



Stack (LIFO)
Order
Last In, First Out



Queue (FIFO)
Order
First In, First Out



Stack (LIFO)
Use Cases
Function calls, parsing



Queue (FIFO)
Use Cases
Scheduling, task queues

SORTING ALGORITHMS

Understanding Sorting Algorithms

What is Sorting?

- Sorting is the process of arranging data in a specific order, typically in ascending or descending order.
- Sorting is fundamental in organizing data for efficient searching, analysis, and presentation.

Why are Sorting Algorithms Important?

- Data Organization: Sorted data is easier to manage and analyze.
- Performance Optimization: Sorting algorithms can reduce search time significantly in large datasets.
- Application in Real-World Scenarios: Sorting is used in various systems, from databases and search engines to applications needing data prioritization.

WHAT IS QUICK SORT?

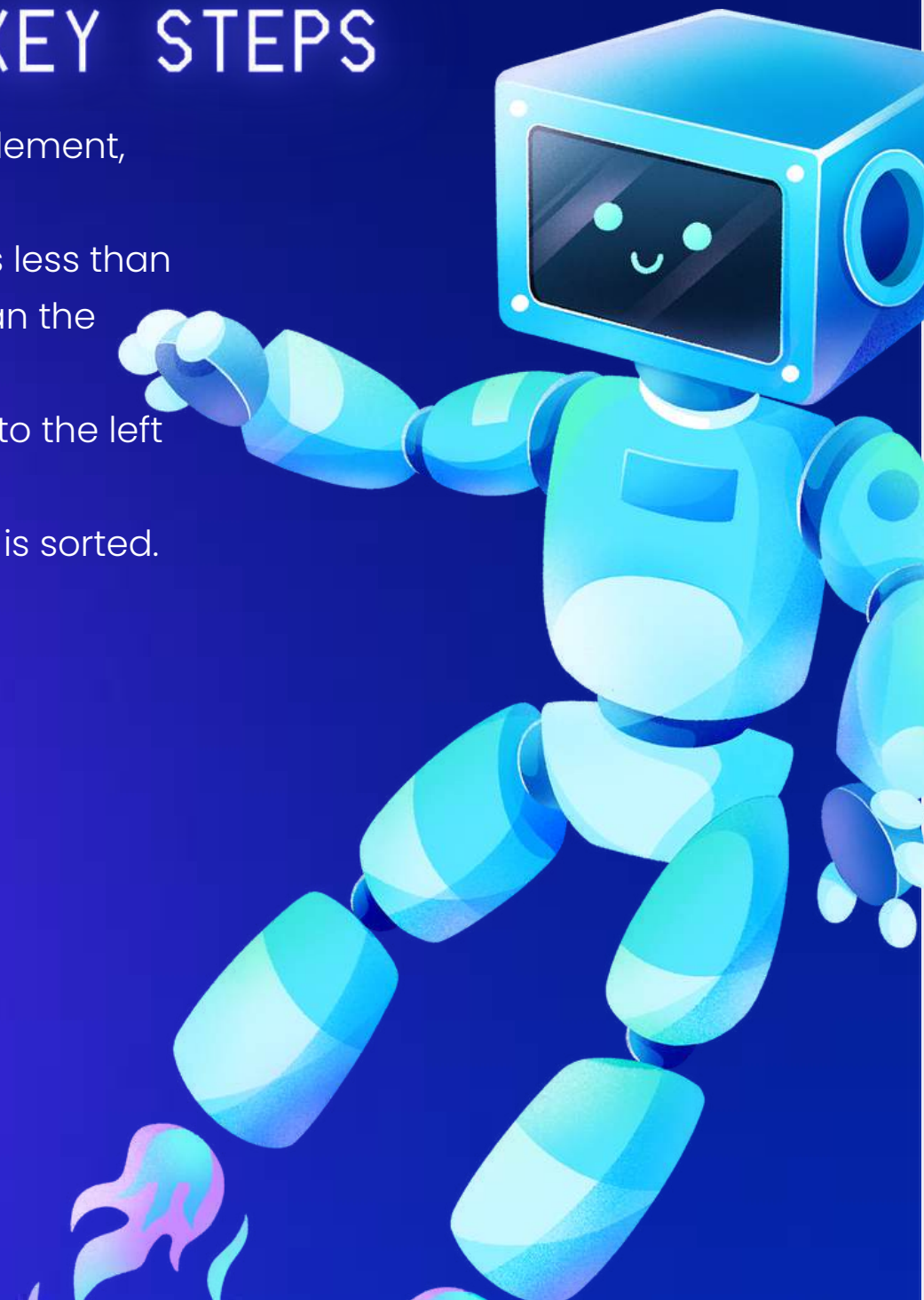
Quick Sort is a divide-and-conquer algorithm that sorts data by partitioning the array into subarrays. A pivot element is chosen, and elements are arranged so that values less than the pivot come before it, and values greater come after it.

HOW QUICK SORT WORKS: KEY STEPS

- Choose a Pivot: Typically, the pivot is the middle element, the first element, or a random element.
- Partitioning: Rearrange the array so that elements less than the pivot are on the left, and elements greater than the pivot are on the right.
- Recursively Sort Subarrays: Apply the same steps to the left and right subarrays around the pivot.
- Combine: Repeat recursively until the entire array is sorted.

QUICK SORT STABILITY

Quick Sort is not a stable sort, meaning it doesn't maintain the relative order of equal elements.



WHAT IS MERGE SORT?

Merge Sort is a divide-and-conquer algorithm that splits the data into halves, recursively sorts each half, and then merges them back together.

This approach is particularly useful for maintaining the order of equal elements, making it a stable sorting algorithm.

How Merge Sort Works: Key Steps

1. Divide: Recursively split the array in half until each subarray contains a single element.
2. Conquer: Sort each pair of subarrays.
3. Merge: Combine the sorted subarrays to produce sorted arrays, working up the recursion stack until the entire array is merged and sorted.

QUICK SORT VS. MERGE SORT: KEY DIFFERENCES

Quick Sort:

Advantages:

- Efficient for large, unsorted lists of student data.
- Memory-friendly, given its in-place sorting, which minimizes extra space requirements.

Disadvantages:

- Unstable: If two students have the same scores, their relative order might not be preserved.
- Worst-case scenario can be inefficient with nearly sorted or reverse-sorted data.

QUICK SORT VS. MERGE SORT: KEY DIFFERENCES

Merge Sort:

- Advantages:
 - Stable: Maintains relative order of students with the same score, ideal for ordered sorting.
 - Predictable $O(n \log n)$ performance for any input, making it more reliable for consistently sorted results.
- Disadvantages:
 - Higher memory usage: Requires extra space for merging, which may impact system memory in cases with very large data volumes.

COMPARE TABLE

Aspect	Quick Sort	Merge Sort
Algorithm Type	Divide-and-Conquer, In-Place	Divide-and-Conquer, Not In-Place
Best & Average Time	$O(n \log n)$	$O(n \log n)$
Worst Case Time	$O(n^2)$ (if pivot selection is poor)	$O(n \log n)$ (always)
Space Complexity	$O(\log n)$ (stack for recursion)	$O(n)$ (additional array for merging)
Stability	Generally Unstable	Stable
Memory Usage	Low (in-place)	Higher (requires additional memory)
Suitability	Best with random, unsorted, or partially sorted data; poor for nearly sorted data	Consistent for all types of data; handles large and nearly sorted lists efficiently

OVERVIEW OF SHORTEST PATH ALGORITHMS

Definition: Shortest Path Algorithms are designed to find the shortest path or minimum distance between nodes in a graph. These algorithms play a crucial role in various applications, such as navigation systems, network routing, and resource optimization.

Importance:

- Real-World Applications: Used in GPS navigation, flight routing, telecommunications, and logistics for efficient pathfinding and cost minimization.
- Graph Theory: Fundamental concepts in computer science and mathematics, allowing for the analysis of connectivity and distances in networks.

Key Concepts:

1. Graph Representation:

- A graph consists of vertices (nodes) and edges (connections). We can represent graphs using:
 - Adjacency Matrix: A 2D array indicating the presence (or weight) of edges between nodes.
 - Adjacency List: A list where each vertex has a corresponding list of edges.

2. Weight:

- The weight of an edge represents the cost or distance to travel from one node to another. We can have both positive and negative weights, influencing which algorithm to use.

3. Types of Graphs:

- Directed Graphs: Edges have a direction, indicating the flow from one vertex to another.
- Undirected Graphs: Edges have no direction, and traversal can occur in both ways.
- Weighted Graphs: Edges have weights representing distances or costs.

INTRODUCTION TO DIJKSTRA'S ALGORITHM

Introduction to Dijkstra's Algorithm

Definition: Dijkstra's Algorithm is a popular graph search algorithm that finds the shortest path from a single source node to all other nodes in a weighted graph with non-negative edge weights. Named after its creator, Edsger W. Dijkstra, the algorithm is foundational in computer science, particularly in the fields of networking and geographical information systems.

Key Features:

- Greedy Approach: Dijkstra's Algorithm operates on the principle of making the locally optimal choice at each step with the hope of finding a global optimum.
- Non-negative Weights: The algorithm assumes that all edge weights (costs) are non-negative, which allows it to effectively determine the shortest paths without the risk of getting trapped in negative cycles.



HOW DIJKSTRA'S ALGORITHM WORKS:

Initialization:

- Start by assigning a tentative distance value to every node: set it to zero for the initial node and infinity for all other nodes.
- Create a set of unvisited nodes containing all nodes in the graph.


Set the Initial Node:

- Mark the initial node as current and examine its neighbors. Calculate their tentative distances through the current node.
- Update Tentative Distances:
- For each neighboring node, if the calculated tentative distance from the current node is less than the previously recorded distance, update it. This ensures we always have the shortest known distance.

Mark the Current Node as Visited:

- Once all neighbors of the current node have been evaluated, mark the current node as visited. A visited node will not be checked again.
- Select the Next Current Node:
- From the unvisited nodes, select the node with the smallest tentative distance and set it as the new current node. Repeat the process until all nodes are visited or the shortest path to the target node is found.

Terminate:

- The algorithm terminates when all nodes have been visited or when the smallest tentative distance among the unvisited nodes is infinity, indicating that remaining nodes are inaccessible from the initial node.
- 

INTRODUCTION TO BELLMAN-FORD ALGORITHM

Definition: The Bellman-Ford algorithm is a fundamental algorithm in graph theory that computes the shortest paths from a single source vertex to all other vertices in a weighted graph. Unlike Dijkstra's algorithm, the Bellman-Ford algorithm can handle graphs with negative edge weights, making it particularly useful in scenarios where costs can decrease.

Key Features:

- **Handles Negative Weights:** One of the primary advantages of the Bellman-Ford algorithm is its ability to accommodate graphs with negative edge weights, which is crucial for applications involving toll roads or cost reductions.
- **Detects Negative Cycles:** The algorithm can also detect negative cycles in a graph—situations where a cycle can reduce the total cost indefinitely, which is important for ensuring the validity of computed paths.



HOW BELLMAN-FORD ALGORITHM WORKS:

Initialization:

- Similar to Dijkstra's algorithm, begin by assigning a tentative distance value to every node: set the distance to zero for the source node and infinity for all other nodes.


Relaxation Process:

- For each vertex, iterate through all edges in the graph and update the distance to each vertex if a shorter path is found. This is done for a total of $V-1$ iterations (where V is the number of vertices), ensuring that the shortest path is found for all vertices.

Negative Cycle Check:

- After the $V-1$ iterations, perform one additional iteration through all edges. If any distance can still be updated, a negative cycle exists, and the algorithm can report this situation.

Final Output:

- If no negative cycles are detected, the algorithm will output the shortest distances from the source vertex to all other vertices.
- 

COMPARE TABLE

Feature	Dijkstra's Algorithm	Bellman-Ford Algorithm
Weight Handling	Non-negative weights only	Handles negative weights
Cycle Detection	Cannot detect negative cycles	Can detect negative cycles
Time Complexity	$O((V + E) \log V)$	$O(V * E)$
Algorithm Type	Greedy	Dynamic programming
Use Cases	Efficient for shortest paths in non-negative graphs	Suitable for graphs with negative weights
Performance	Generally faster in practice	Slower, especially with dense graphs