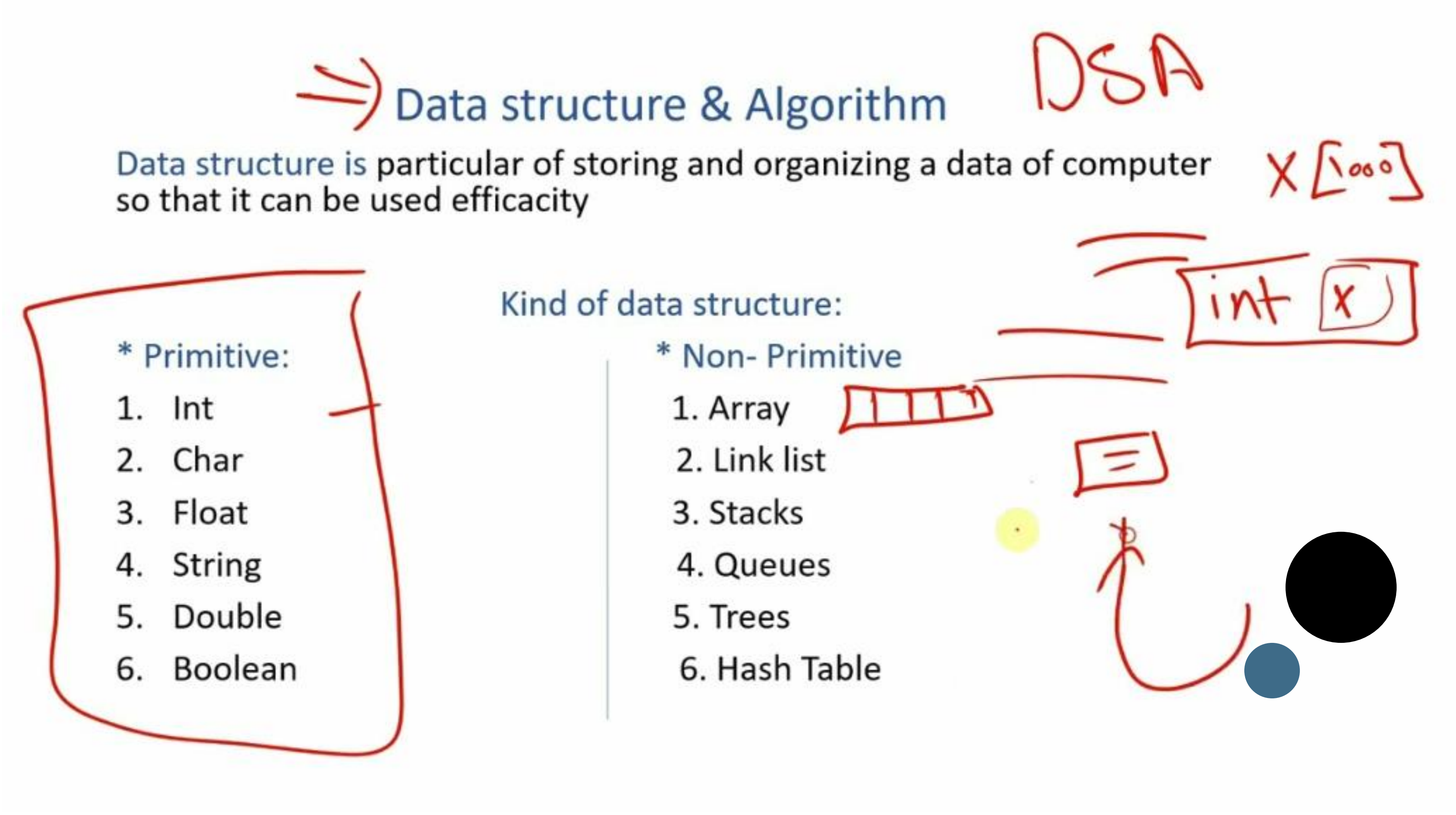
Travel Agency

DATA STRUCTURES AND ALGORITHMS (DSA)

WHAT IS DSA?

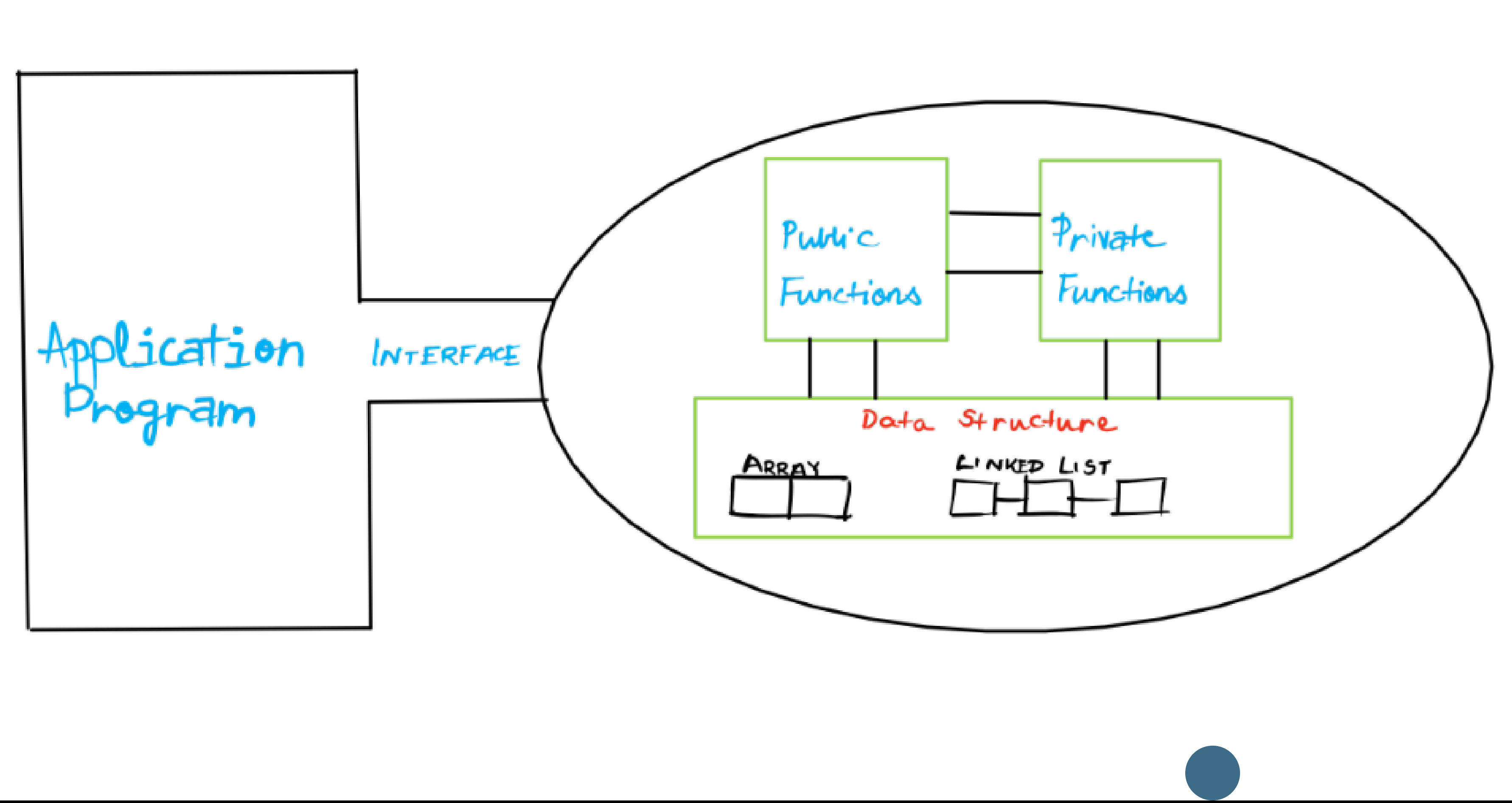
- DSA stands for Data Structures and Algorithms, which are essential components in computer science used to store, organize, and manipulate data efficiently.
- A Data Structure is a way of storing and organizing data, and an Algorithm is a step-by-step procedure for solving a problem.

Read More



WHAT IS AN ADT (ABSTRACT DATA TYPE)?

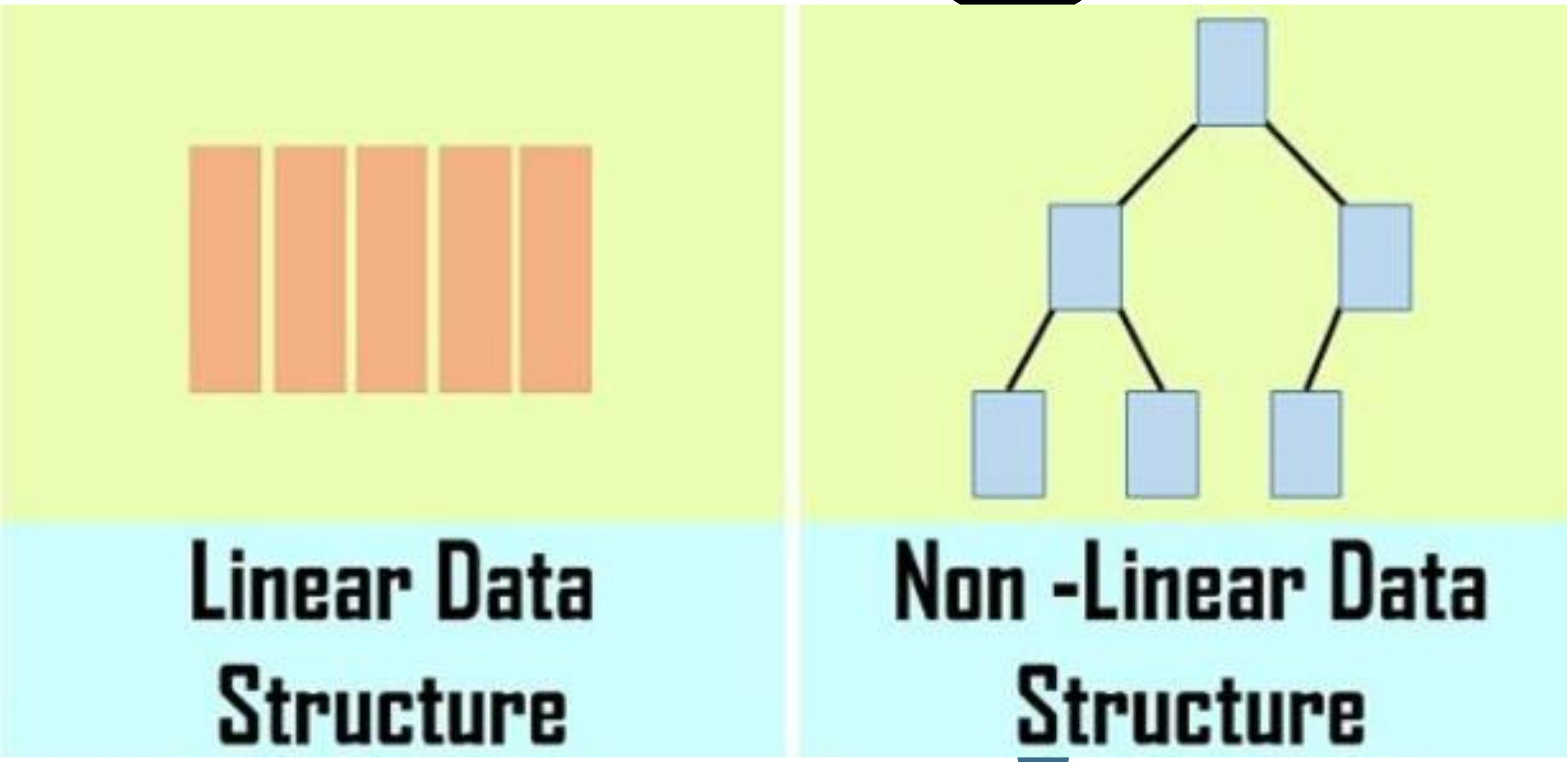
An ADT is a theoretical concept that defines a data structure purely in terms of its behavior (operations and possible outcomes) without specifying how it is implemented.



Examples include Stack, Queue, List, and Map

Home About Us Trip Package Contact Us

TYPES OF DATA STRUCTURES



Linear Structures: Arrays, Linked Lists, Stacks, Queues

Non-Linear Structures: Trees, Graphs

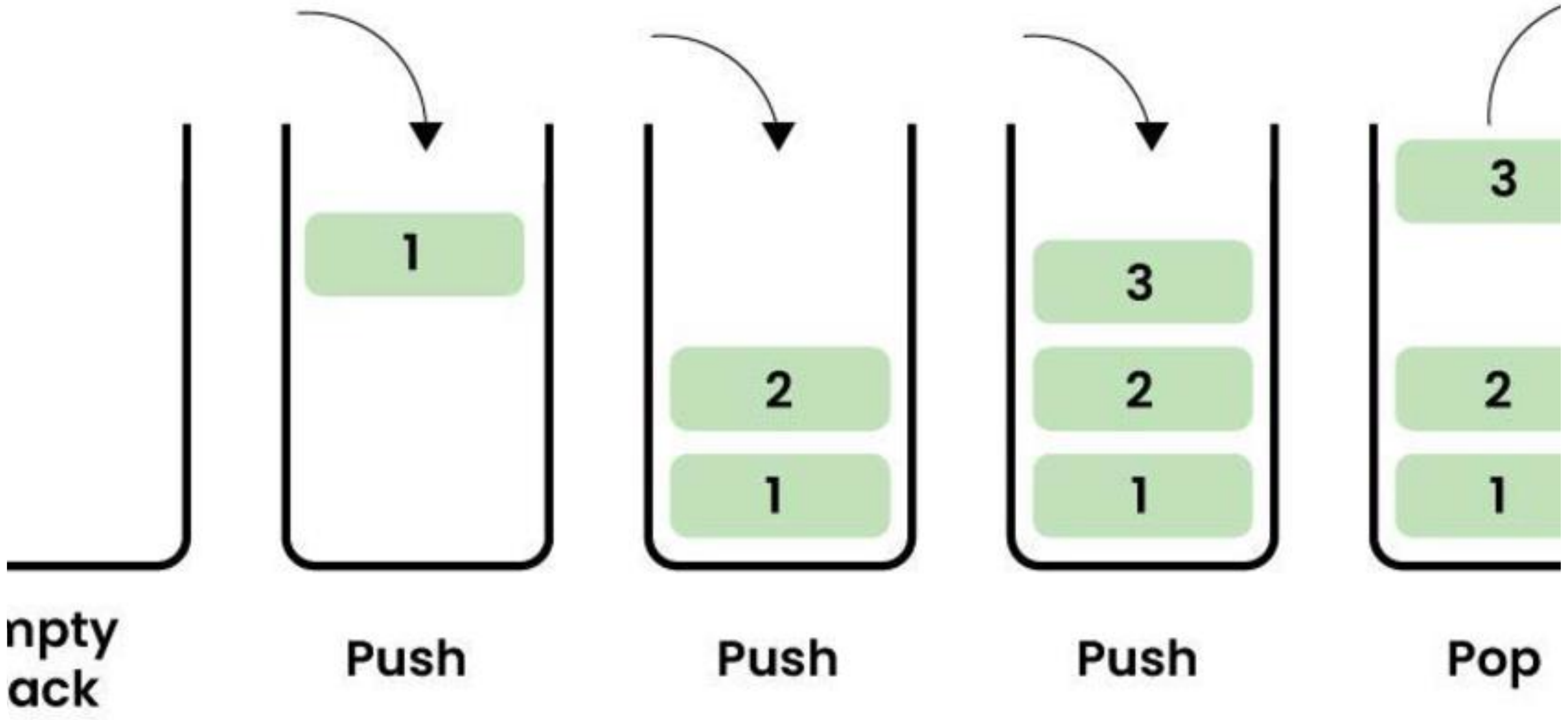
WHAT IS A STACK?

Definition

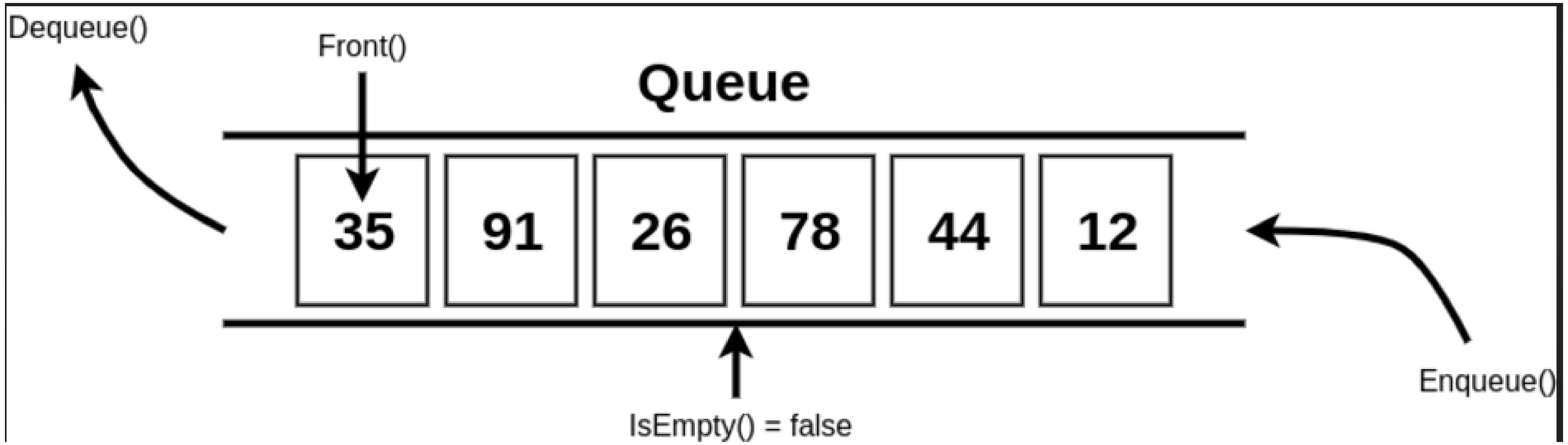
A Stack is a linear data structure that follows the Last In, First Out (LIFO) principle. The last element added is the first to be removed.

Operations

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



WHAT IS A QUEUE?



Definition

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

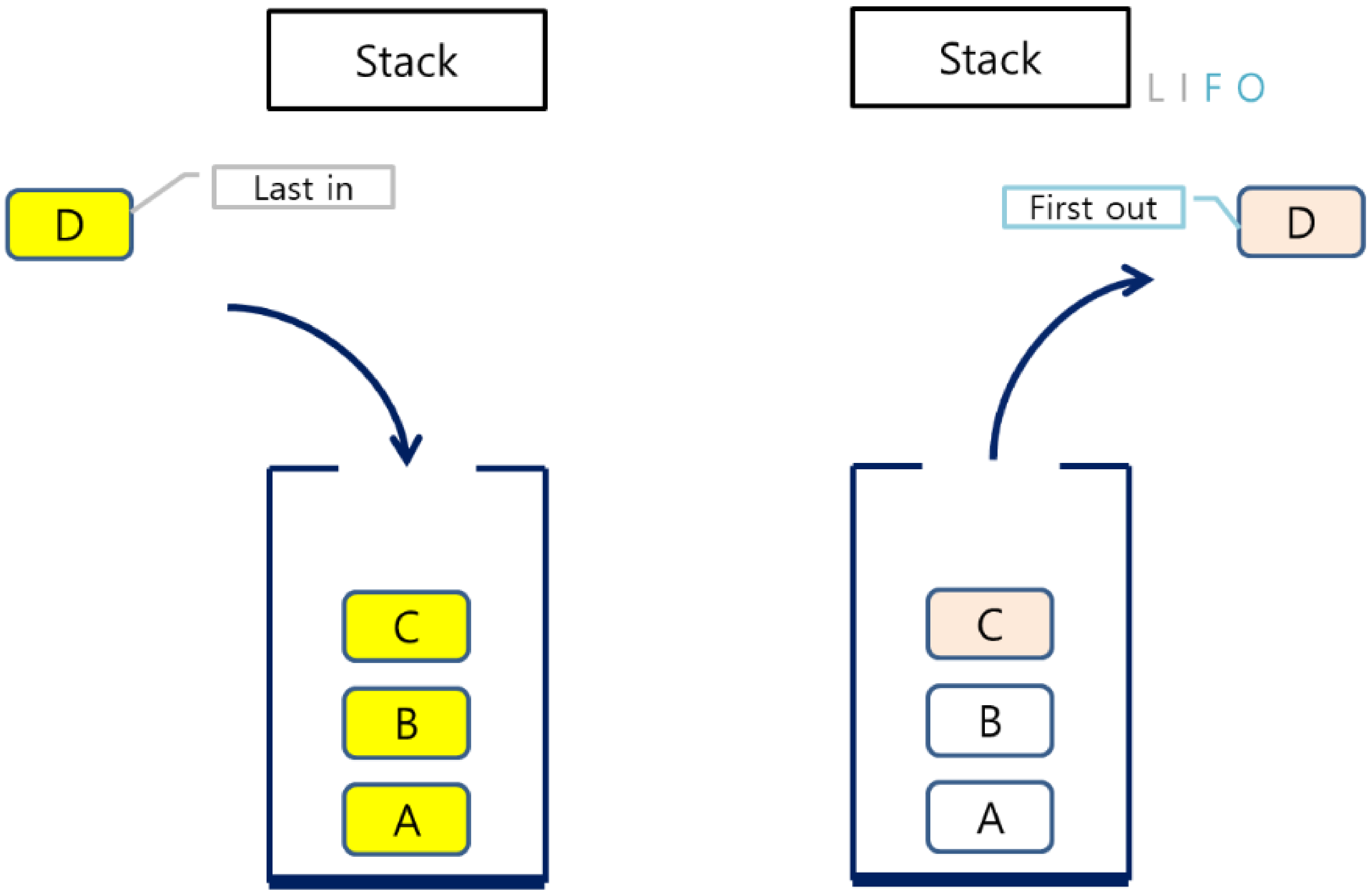
Operations

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

STACK VS. QUEUE

Usage

- Stack: Undo operations, function calls in programming.
- Queue: Scheduling tasks, resource management.

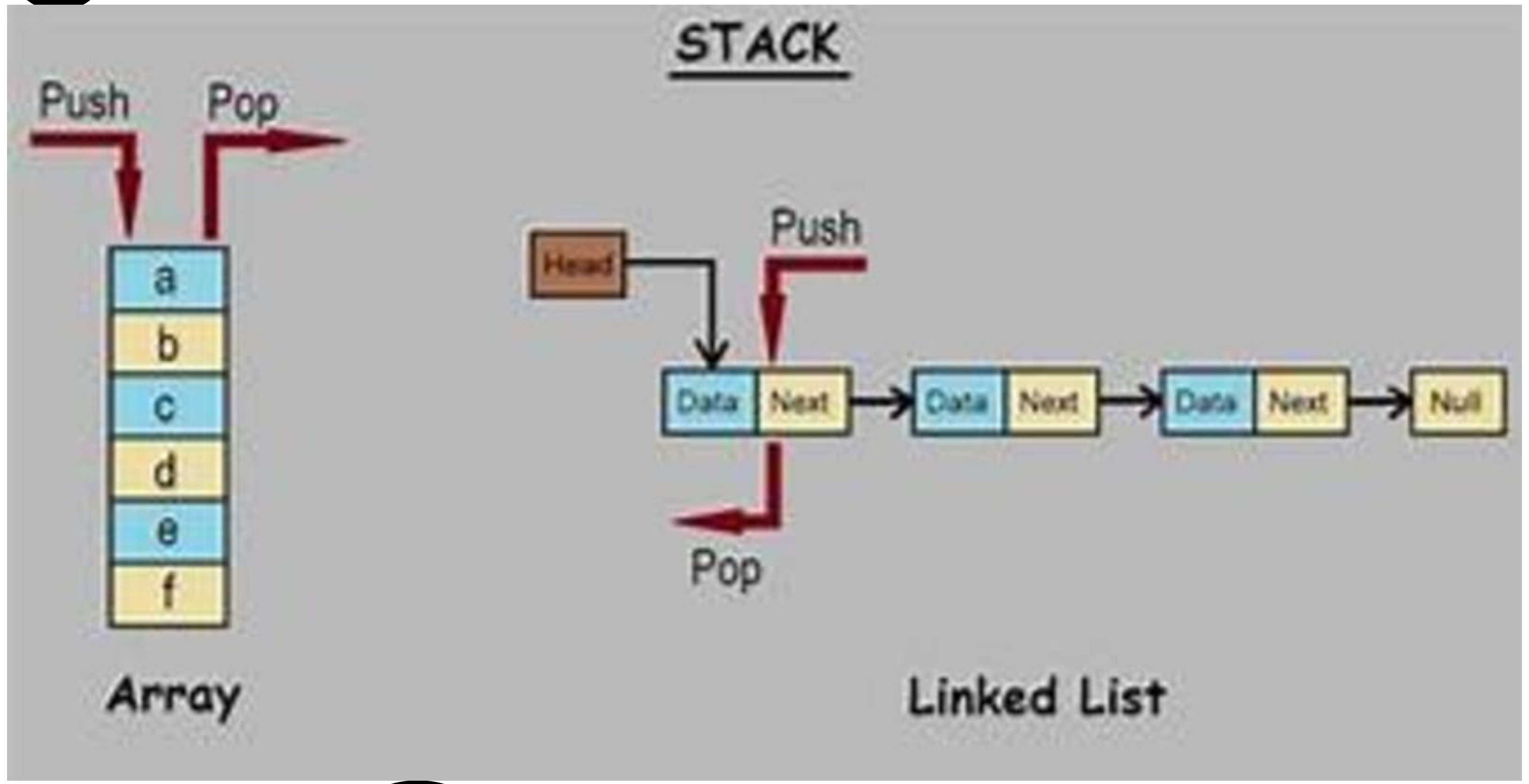


Comparison

Stack is LIFO (Last In First Out), Queue is FIFO (First In First Out).

Home About Us Trip Package Contact Us

HOW TO IMPLEMENT A STACK?



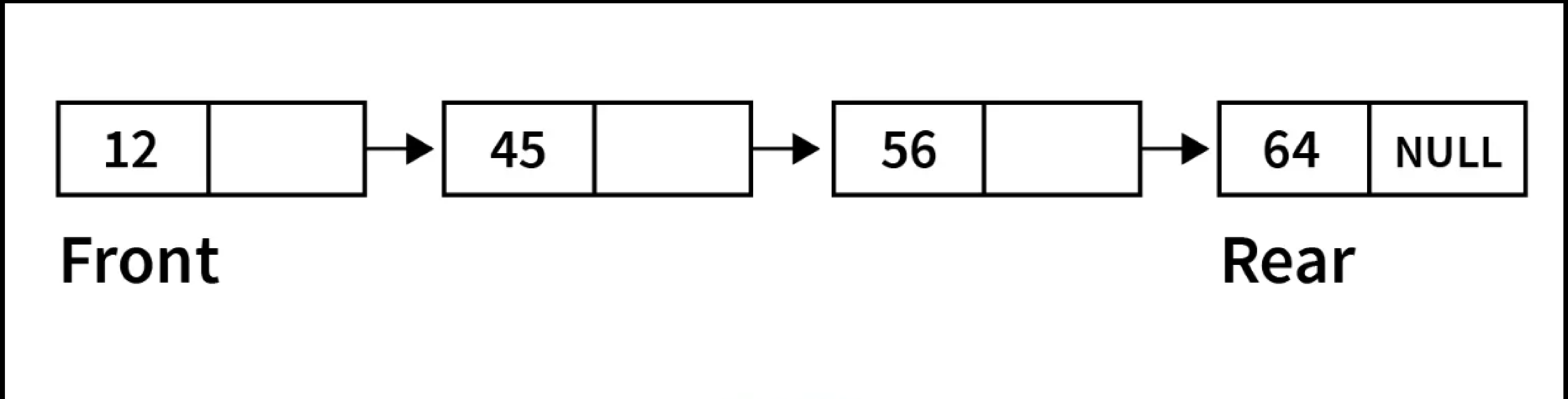
Ways to Implement

Array-based Implementation: Fixed size, simple to use.

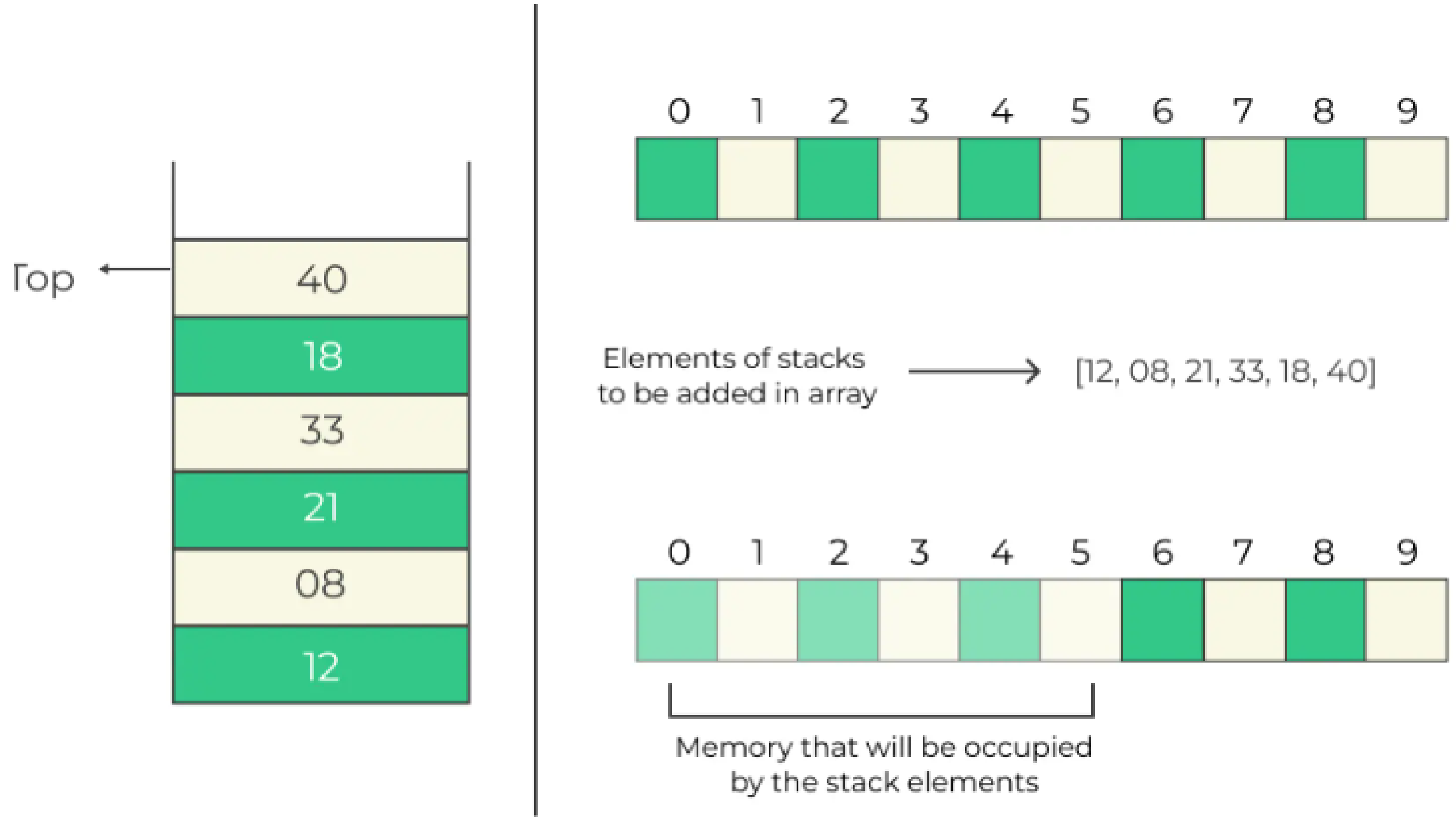
Linked List-based Implementation: Dynamic size, more flexible.

HOW TO IMPLEMENT A QUEUE?

- Array-based Implementation: Fixed size, circular queue to prevent overflow
- Linked List-based Implementation: Dynamic size, more flexible.

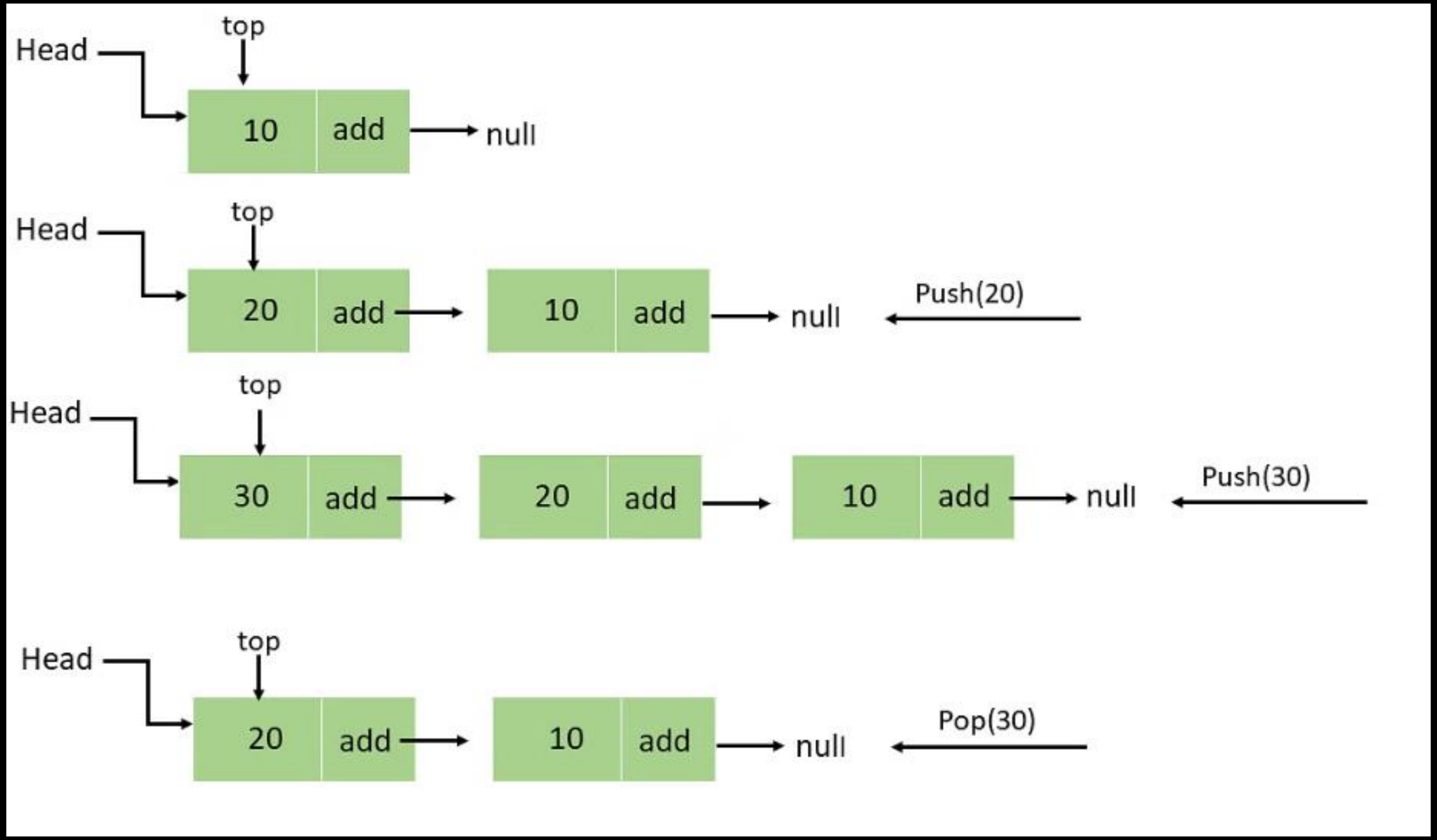


ARRAY IMPLEMENTATION OF STACK



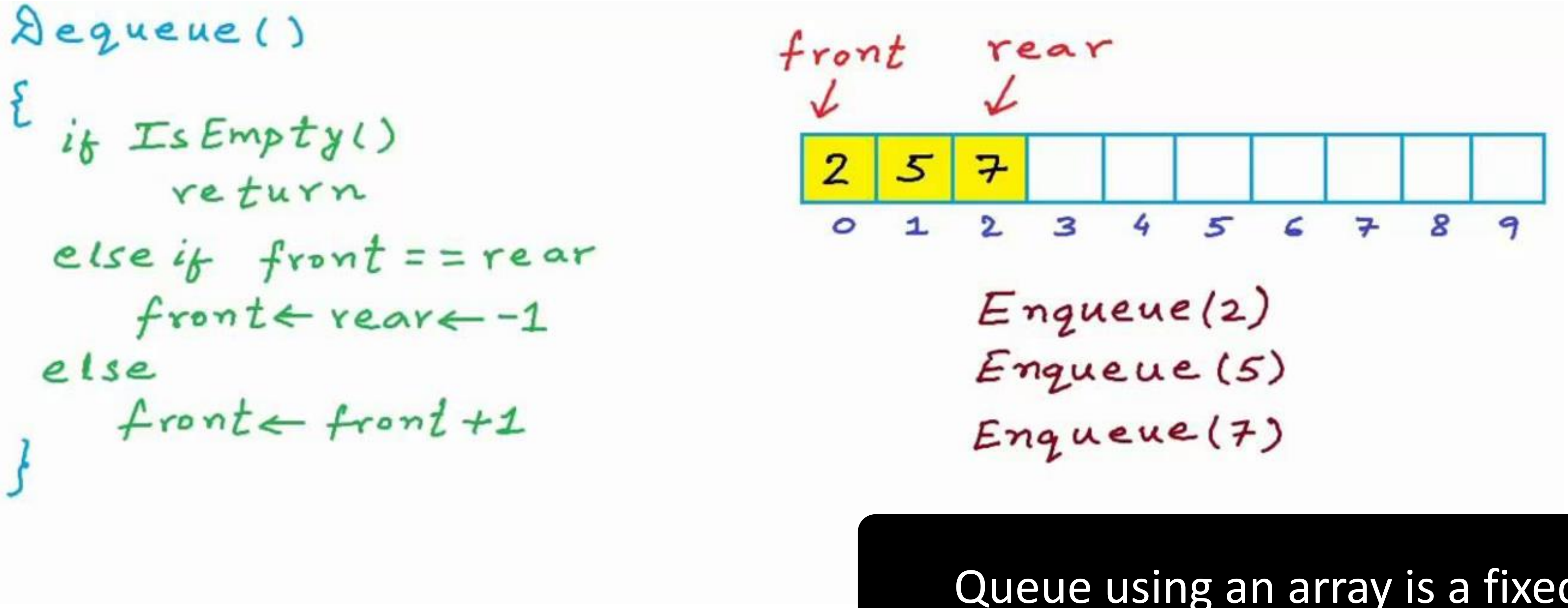
- Stack using an array is simple to implement. However, the size of the stack is fixed and cannot grow beyond its defined capacity.

LINKED LIST IMPLEMENTATION OF STACK



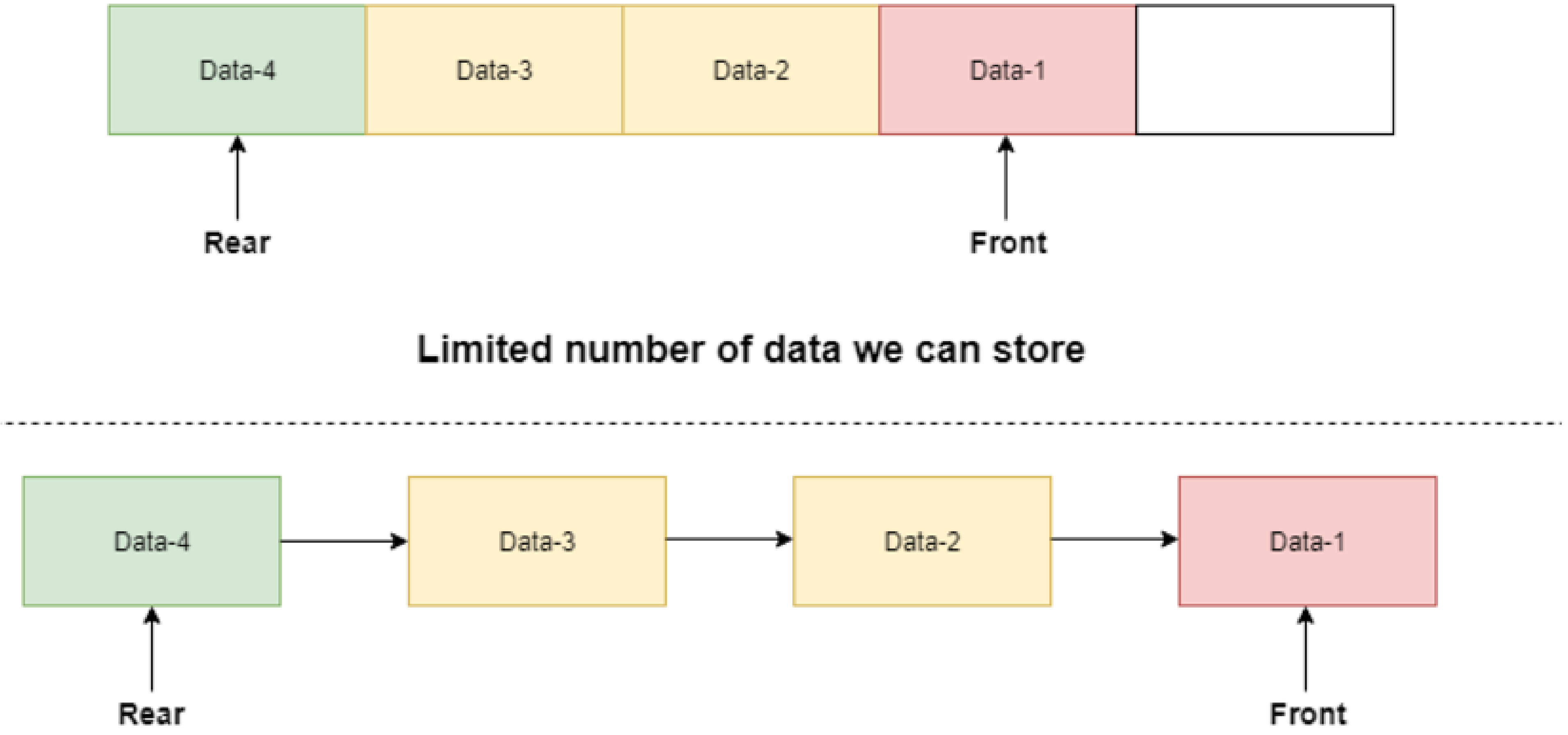
Stack using a linked list is more flexible and dynamic in size, as it grows or shrinks as needed.

ARRAY IMPLEMENTATION OF QUEUE



Queue using an array is a fixed-size data structure. It is implemented using a circular array to efficiently manage the front and rear.

LINKED LIST IMPLEMENTATION OF QUEUE

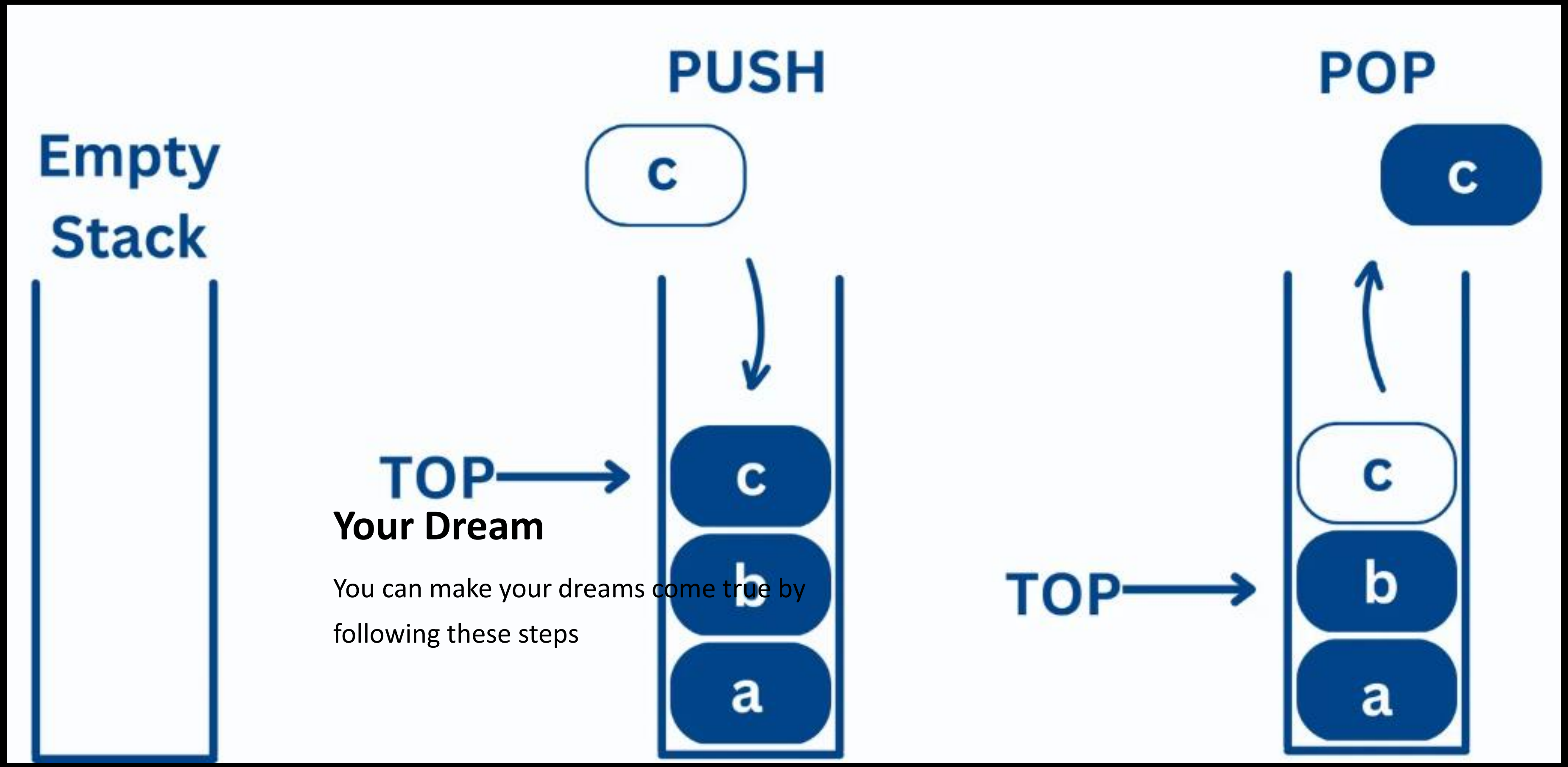


Queue using a linked list is dynamic and allows efficient insertion and removal of elements from both ends.

STACK APPLICATIONS

Use Cases

- Undo/Redo functionality in applications.
- Function call management (e.g., recursion).
- Syntax parsing in compilers.



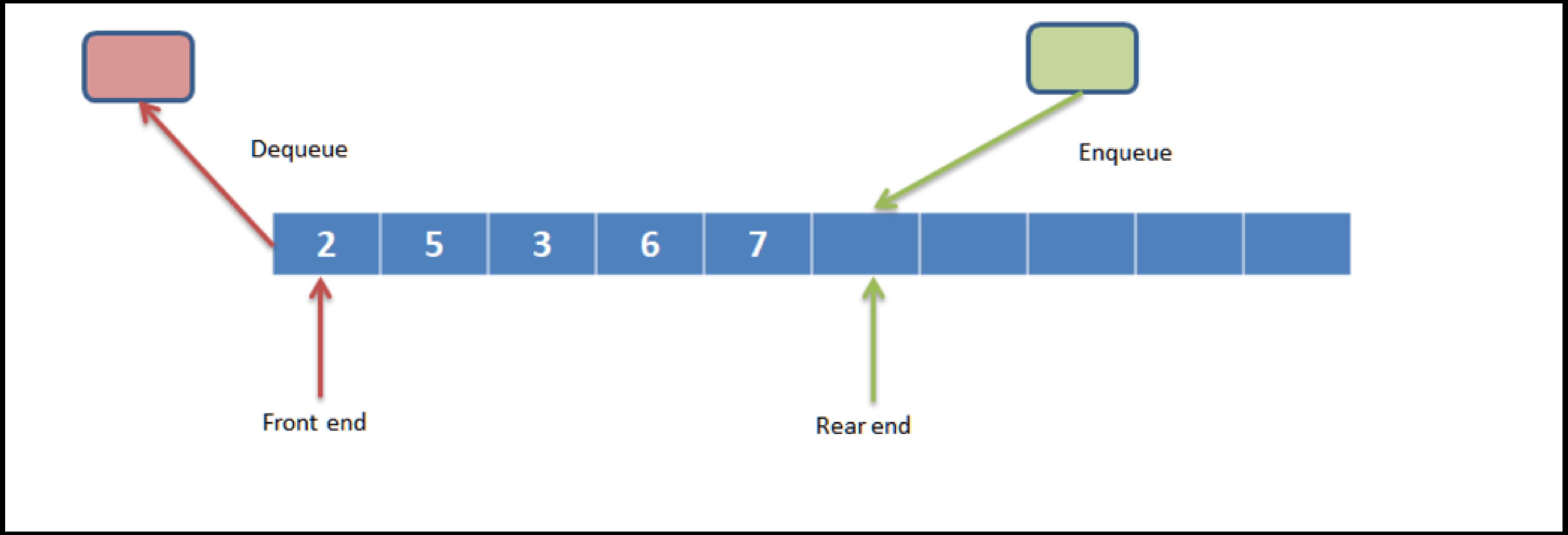
Your Dream

You can make your dreams come true by following these steps

QUEUE APPLICATIONS

Use Cases:

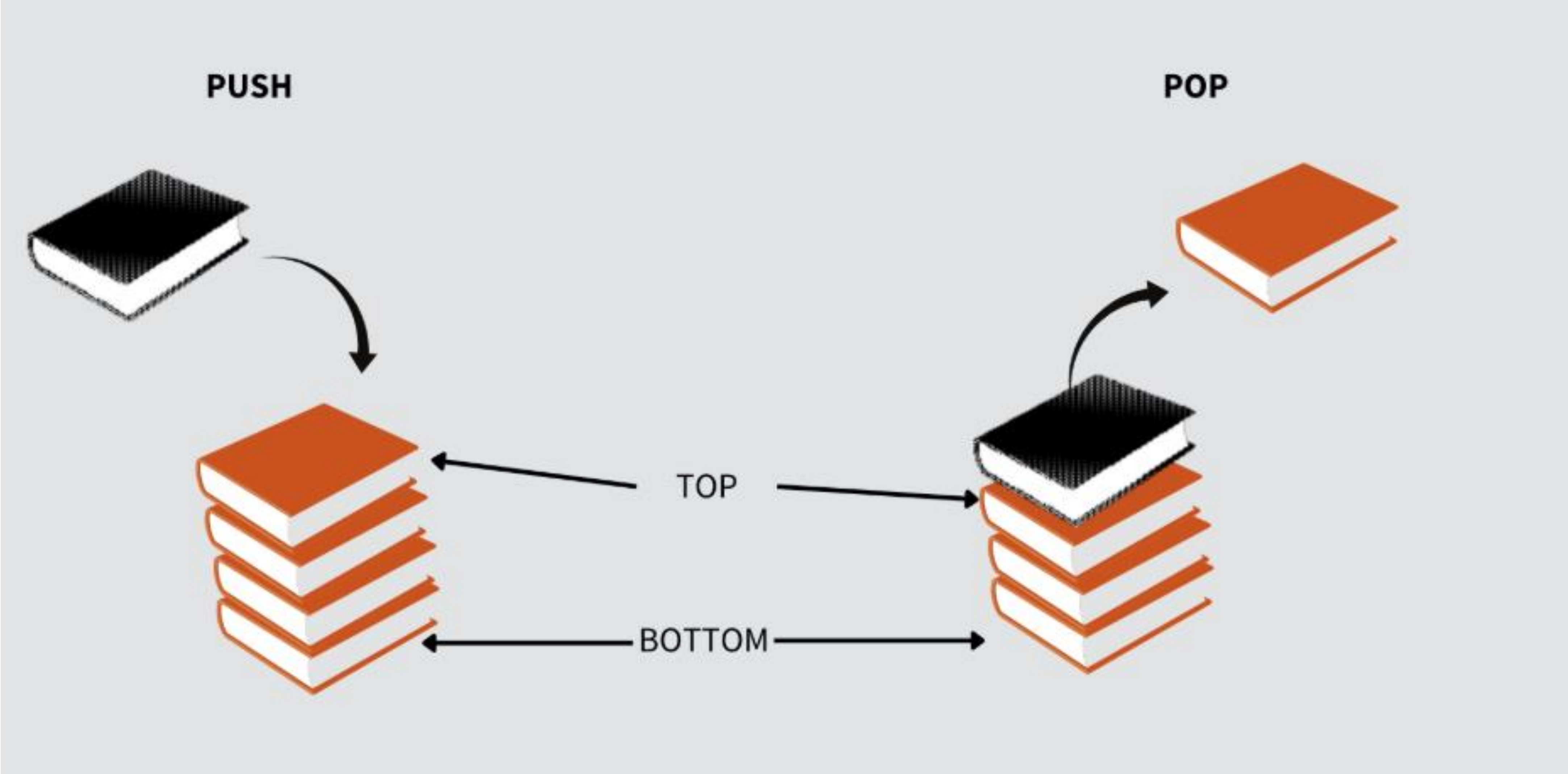
- Task scheduling (e.g., printer queues, process scheduling).
- Breadth-first search (BFS) in graph traversal.
- Handling requests in web servers.



ADVANTAGES OF USING STACK AND QUEUE

Advantages:

- Stack: Simple to implement, efficient for recursion and managing function calls.
- Queue: Ideal for handling tasks in order, maintains sequence integrity.



SORTING ALGORITHMS OVERVIEW

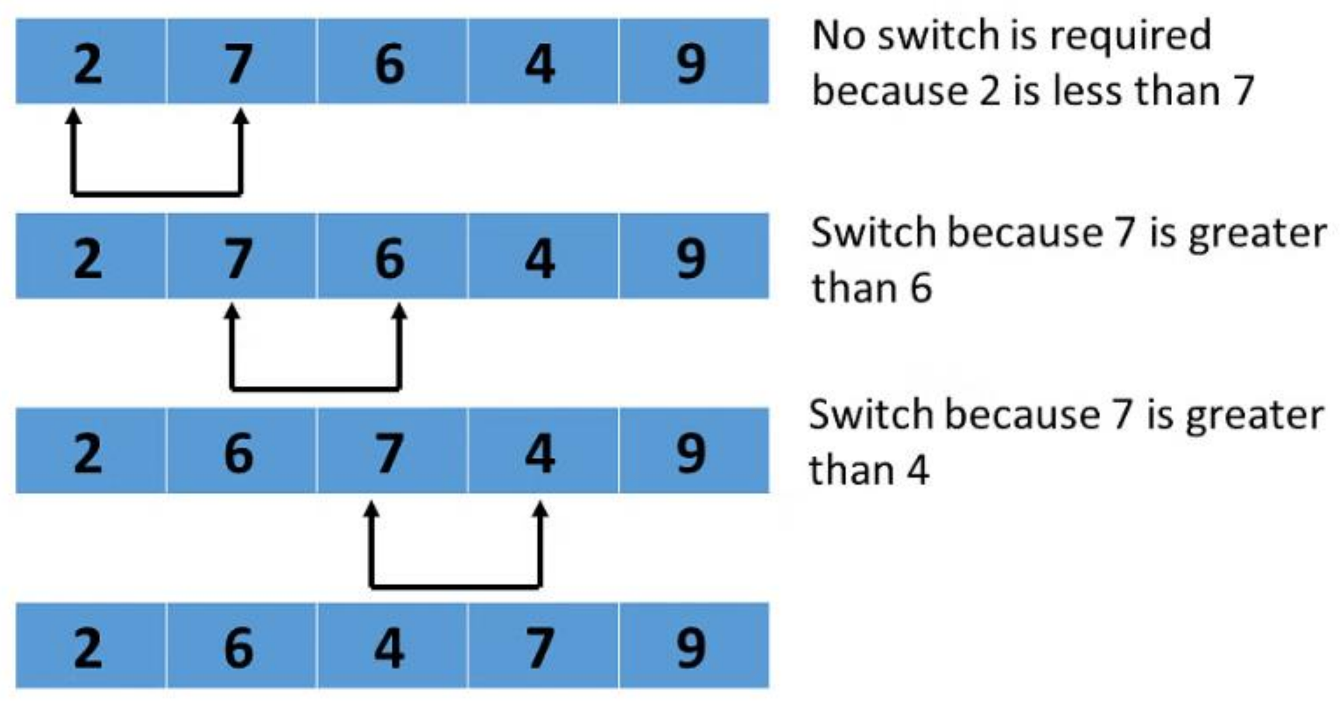
- Sorting is a fundamental operation in DSA, organizing data into a specific order.
- Common use cases: searching, data analysis, and optimizing storage.
- Sorting algorithms work by comparing and rearranging data elements.

Home About Us Trip Package Contact Us

SELECTED SORTING ALGORITHMS

Algorithm 1: Bubble Sort
- A simple comparison-based algorithm.
- Swaps adjacent elements if they are in the wrong order.
- Easy to implement but inefficient for large datasets.

Algorithm 2: Quick Sort
- A divide-and-conquer algorithm.
- Picks a "pivot" and partitions the array around it.
- More efficient, especially for large datasets.



Home About Us Trip Package Contact Us Studio Shodwe

Pricing Plan

TIME AND SPACE COMPLEXITY

Algorithm	Time Complexity (Best)	Time Complexity (Worst)	Space Complexity
Bubble Sort	$O(n)$	$O(n^2)$	$O(1)$
Quick Sort	$O(n \log n)$	$O(n^2)$	$O(\log n)$

Bubble Sort is better for small datasets but inefficient for large ones

Quick Sort is faster for large datasets but requires extra space for recursion.

Home About Us Trip Package Contact Us

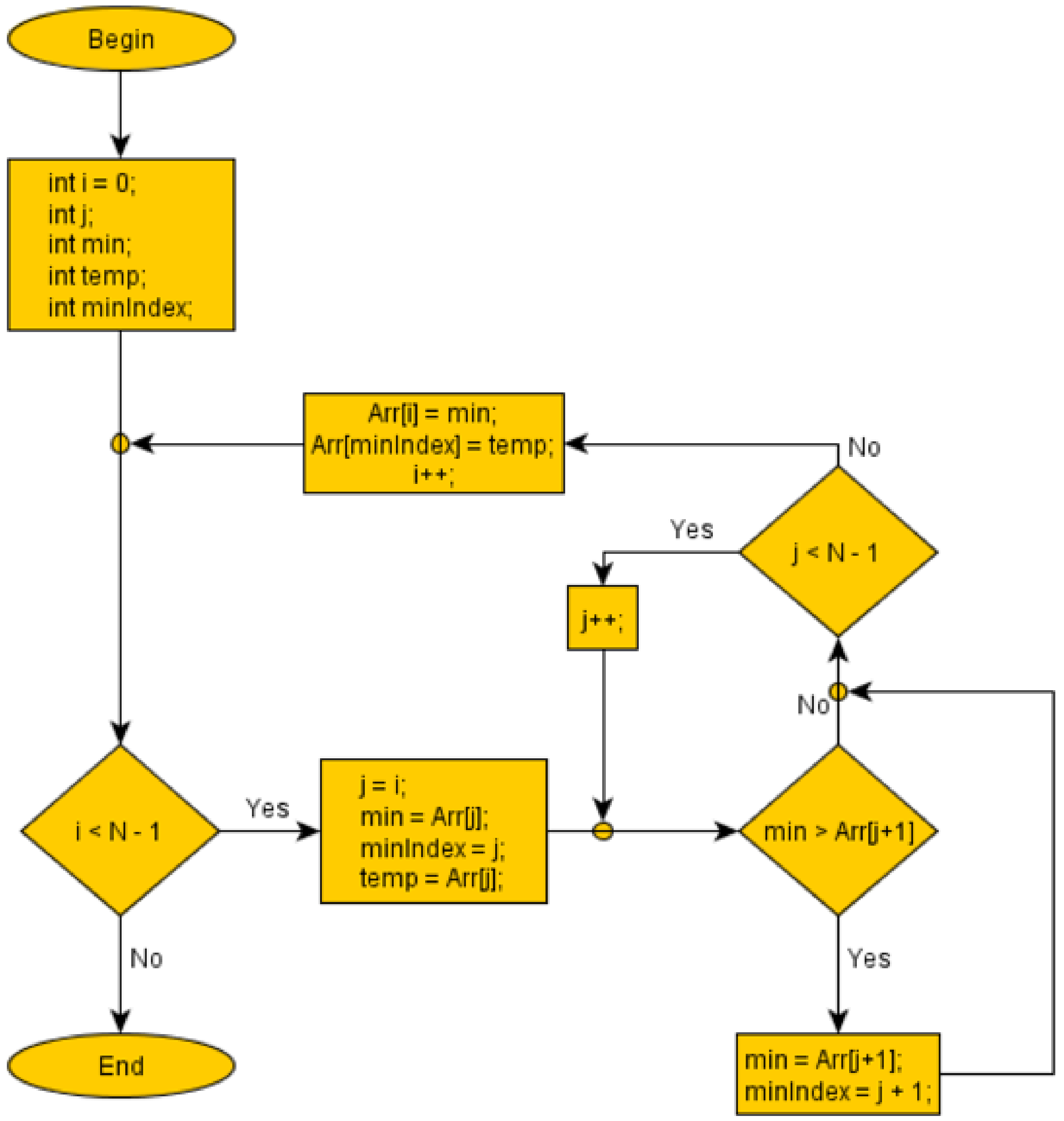
WHEN TO USE EACH ALGORITHM

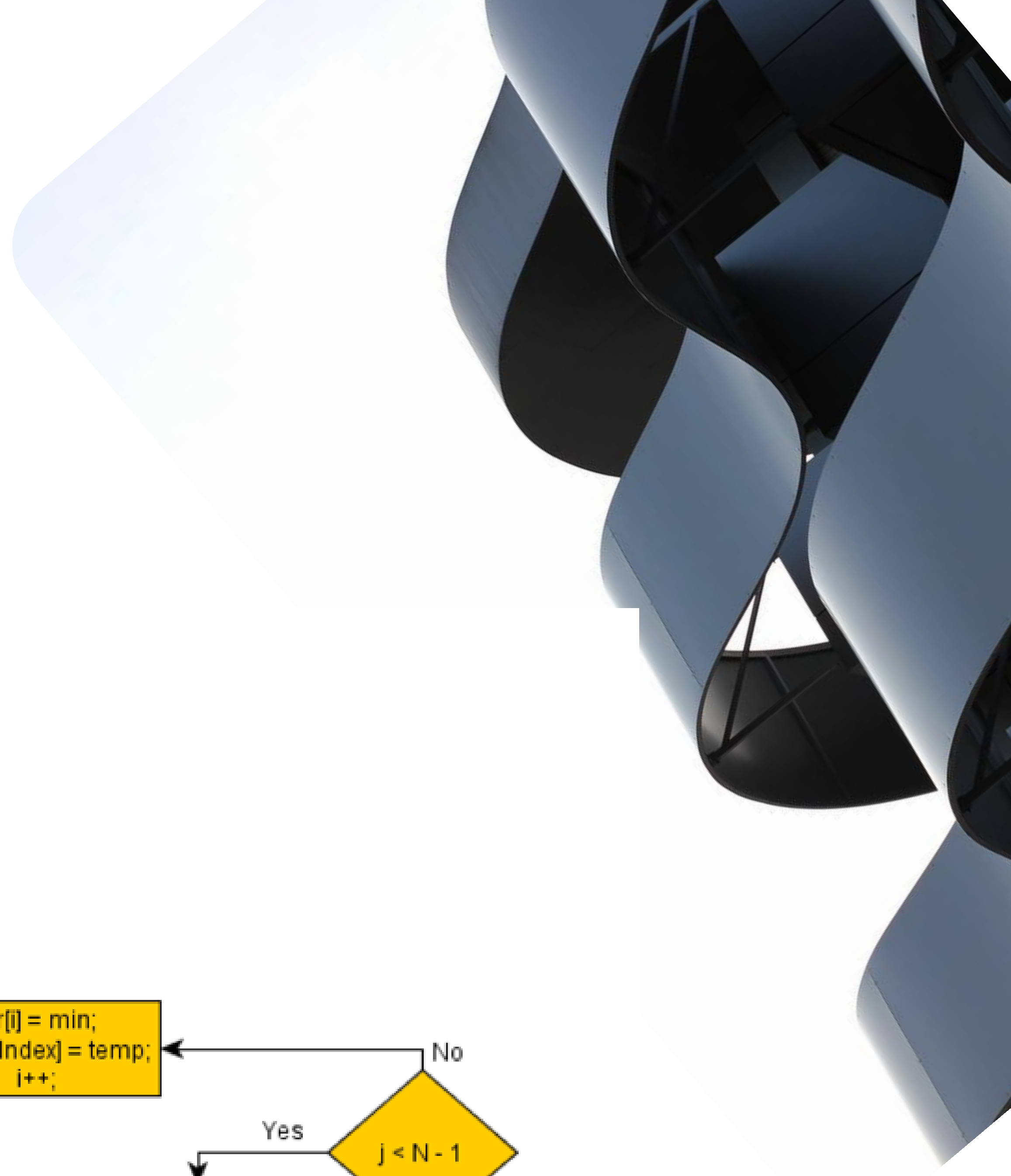
Bubble Sort:

- Educational purposes and small datasets.
- Easy to implement and understand.

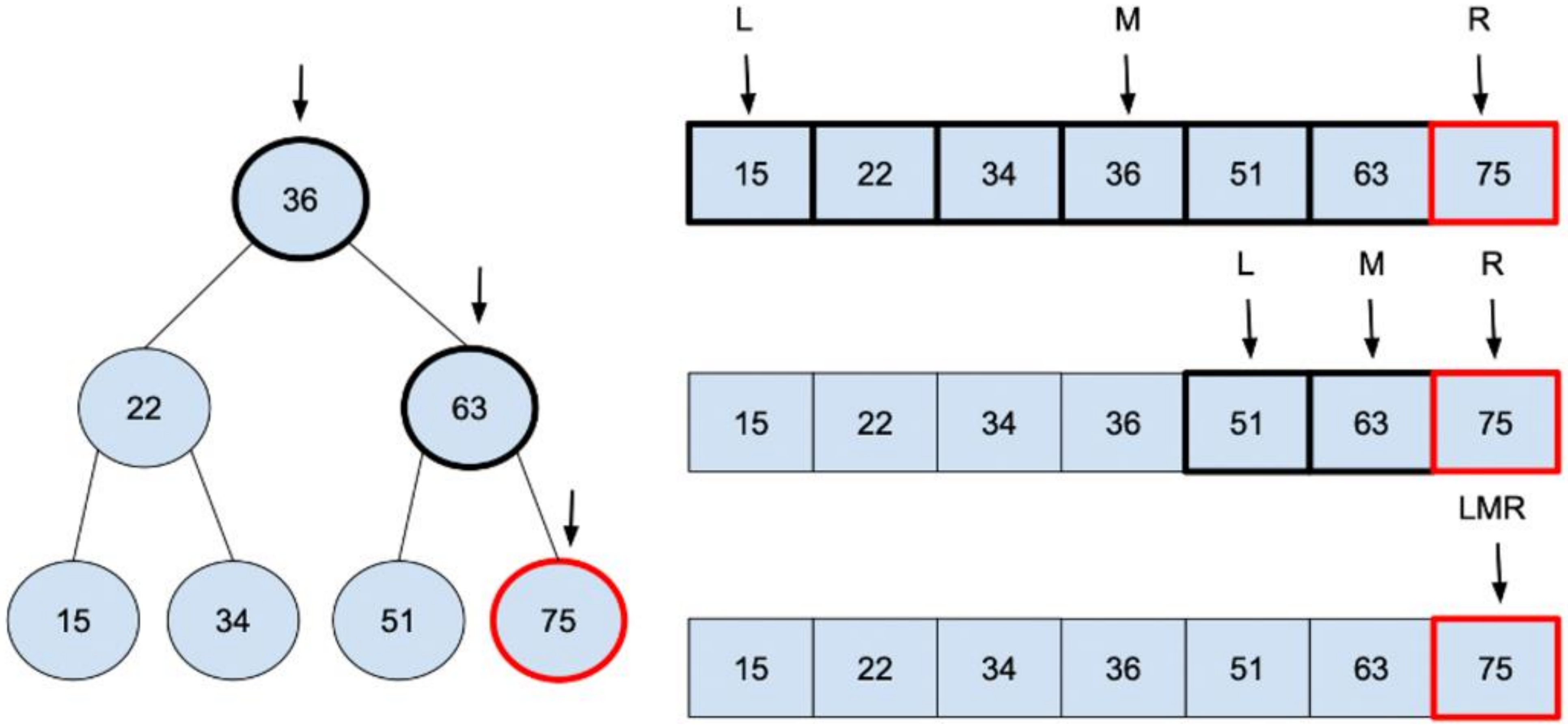
Quick Sort:

- Large datasets where performance is critical.
- Often used in real-world applications like database management.





SORTING IN THE CONTEXT OF DSA AND ADTS



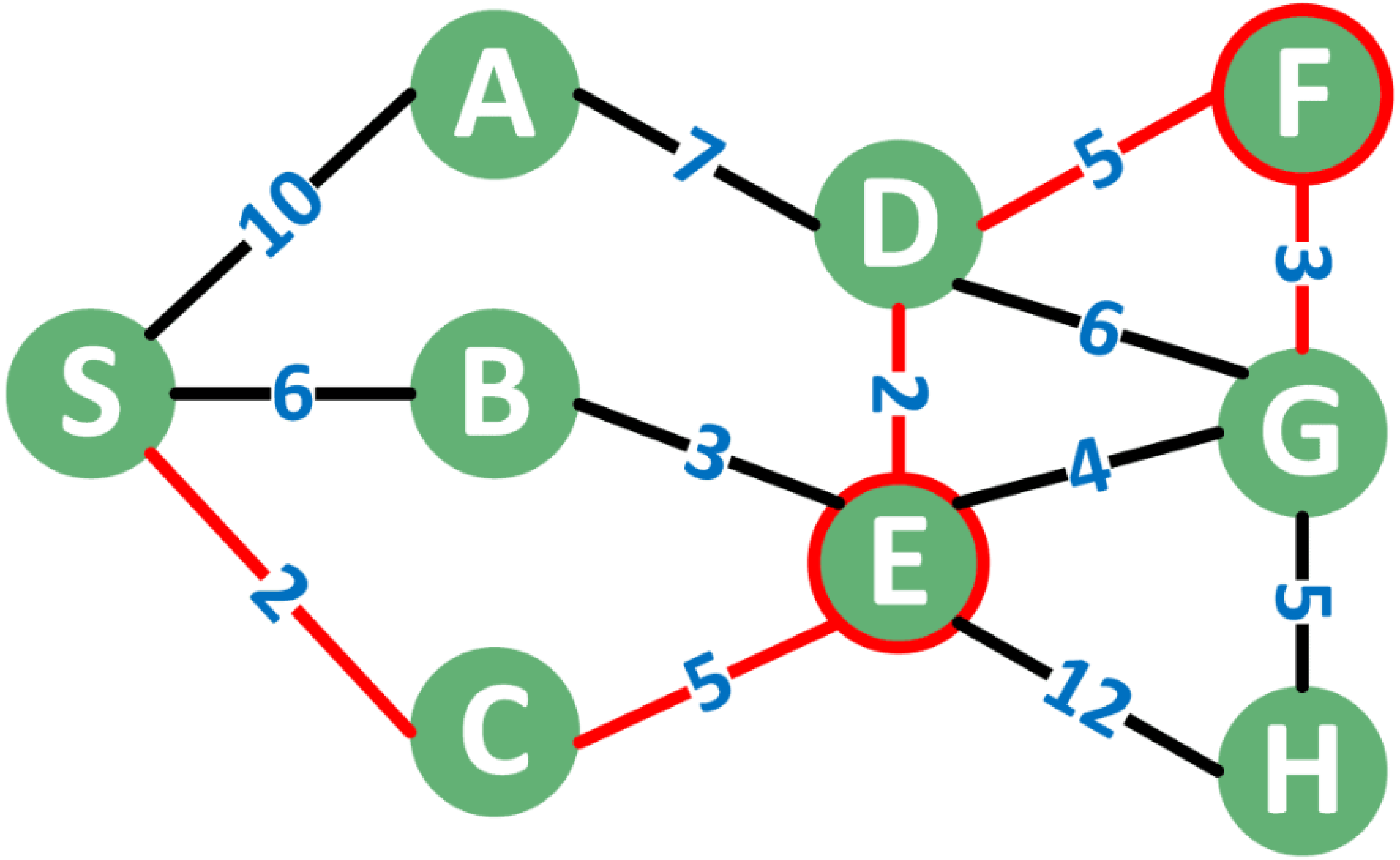
Sorting enhances the efficiency of data structures like Stacks and Queues.

For example:

- Sorting helps organize elements in a Queue for optimized scheduling.
- Sorting ensures efficient management of call stacks in recursive algorithms.

UNDERSTANDING SHORTEST PATH ALGORITHMS

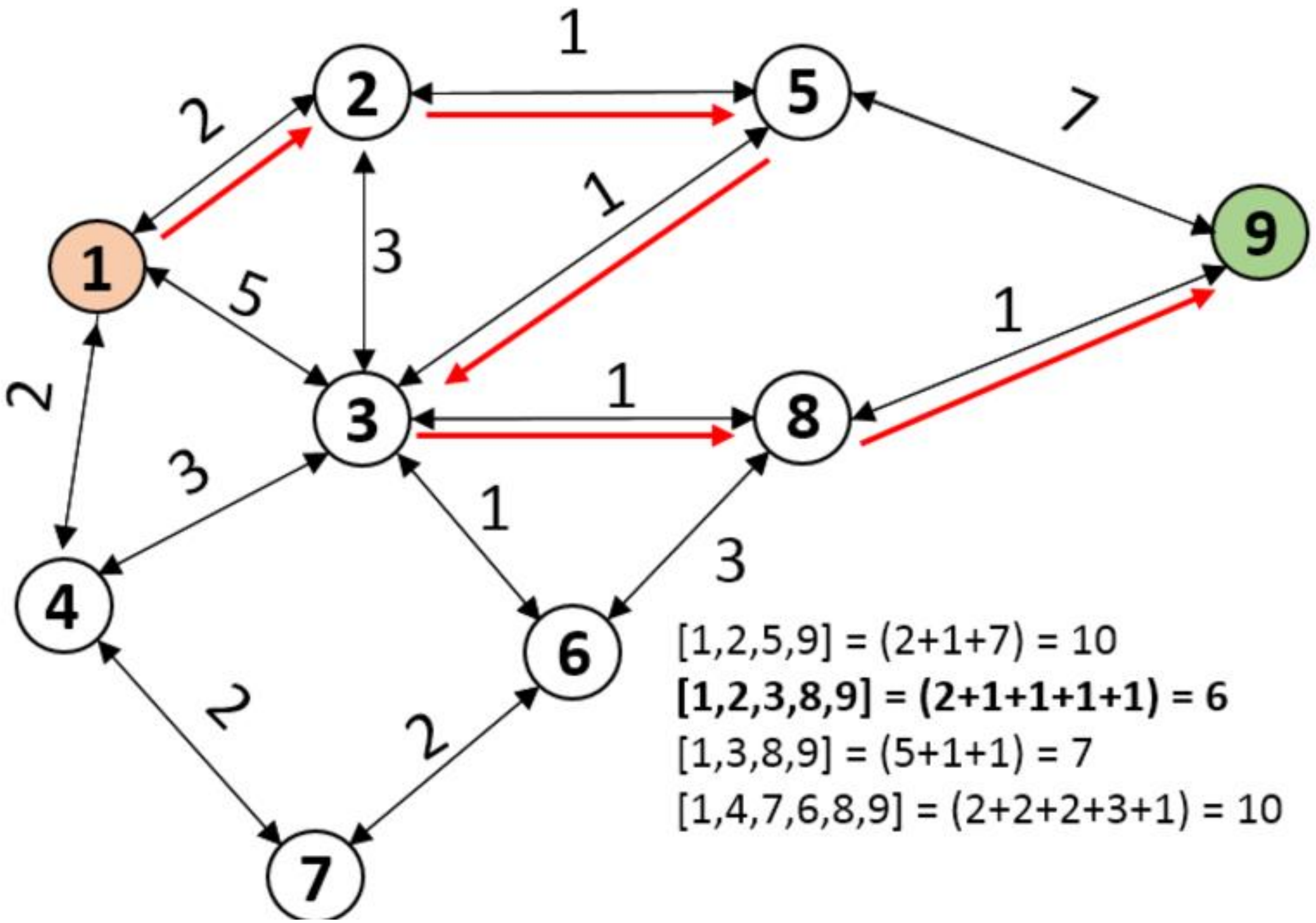
Shortest path algorithms are used to find the minimum distance between two nodes in a graph.



Applications:

- Network routing
- Transportation planning
- Game AI
- Common algorithms: Dijkstra's, Bellman-Ford, and A*.

DIJKSTRA'S ALGORITHM OVERVIEW



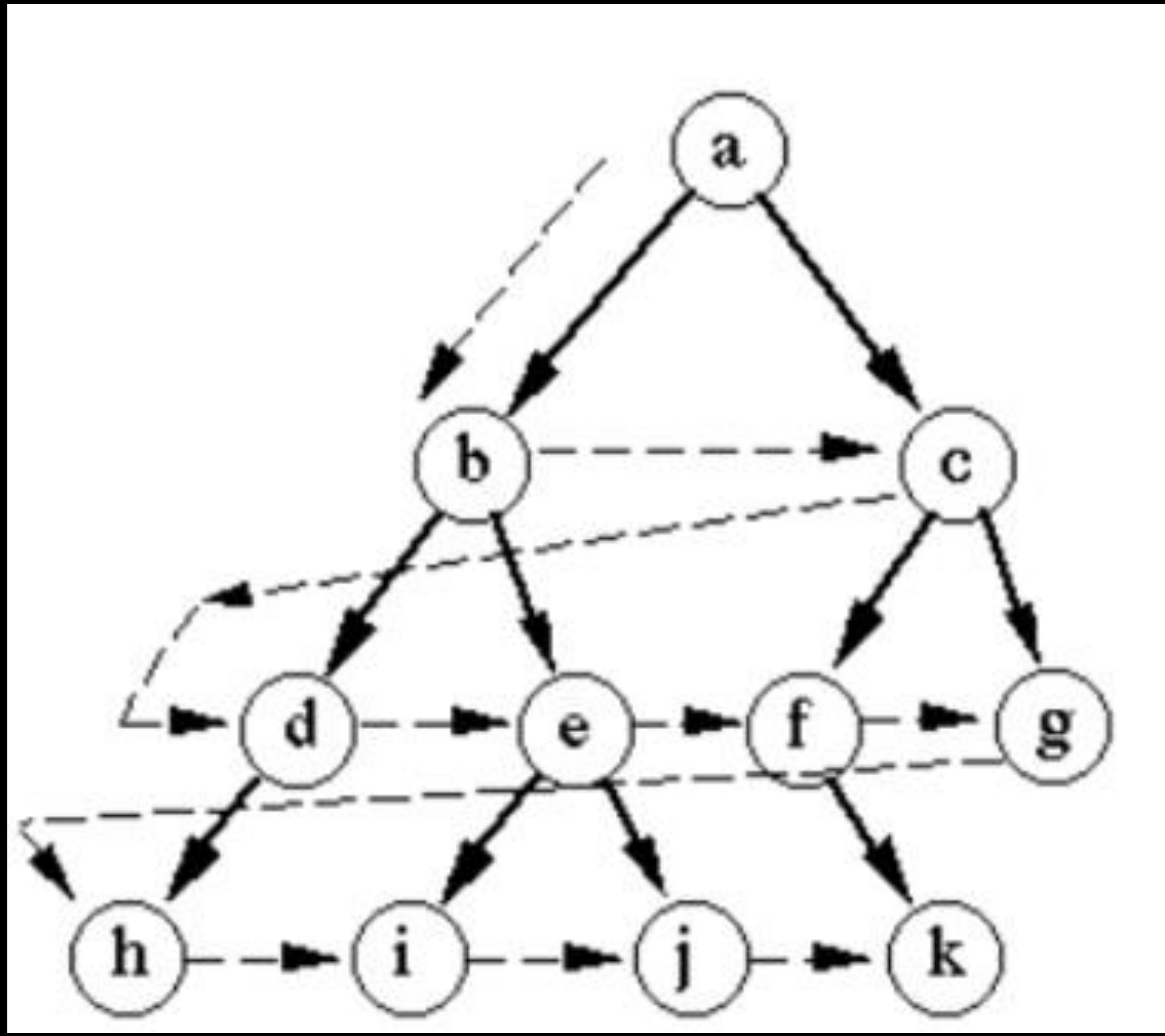
- Finds the shortest path from a source node to all other nodes in a weighted graph.
- Assumes non-negative weights.
- Steps:

1. Assign an initial distance of infinity to all nodes (0 for the source).
2. Mark all nodes as unvisited.
3. Update distances for neighbors of the current node.
4. Select the unvisited node with the smallest distance.
5. Repeat until all nodes are visited.

Home About Us Trip Package Contact Us

Studio Shodwe

COMPLEXITY OF DIJKSTRA'S ALGORITHM



- Time Complexity:
- Using a priority queue: O((V+E)logV)O((V + E) \log V)O((V+E)logV), where VVV = vertices, EEE = edges.
- Without a priority queue: O(V2)O(V^2)O(V2).
- Space Complexity: O(V)O(V)O(V) for storing distances and visited nodes.

JAVA CODE FOR DIJKSTRA’S ALGORITHM (SETUP AND GRAPH REPRESENTATION)

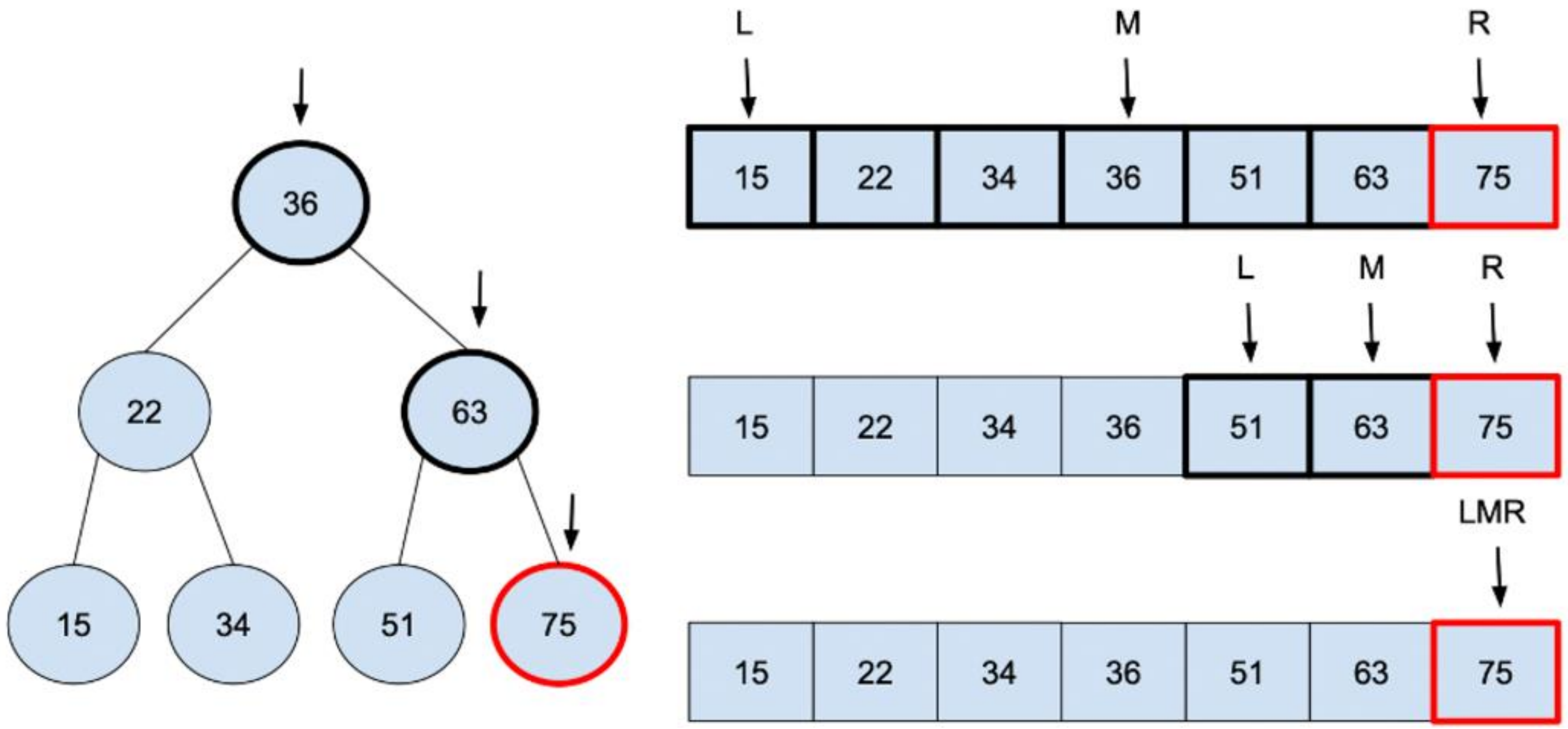
```
import java.util.*;

class Node implements Comparable<Node> {
    int vertex;
    int weight;

    Node(int vertex, int weight) {
        this.vertex = vertex;
        this.weight = weight;
    }

    @Override
    public int compareTo(Node other) {
        return Integer.compare(this.weight, other.weight);
    }
}
```

BELLMAN-FORD ALGORITHM OVERVIEW



- Finds the shortest path from a source node to all other nodes.
- Handles graphs with negative weights.
- Steps:

1. Initialize distances to infinity (0 for the source).
2. Relax edges repeatedly for V−1V - 1V−1 iterations (where VVV is the number of vertices).
3. Check for negative-weight cycles.

Java Code for Dijkstra's Algorithm (Main Logic)

```
public static int[] dijkstra(List<List<Node>> graph, int source) {
    int n = graph.size();
    int[] distances = new int[n];
    Arrays.fill(distances, Integer.MAX_VALUE);
    distances[source] = 0;

    PriorityQueue<Node> pq = new PriorityQueue<>();
    pq.add(new Node(source, 0));

    while (!pq.isEmpty()) {
        Node current = pq.poll();
        for (Node neighbor : graph.get(current.vertex)) {
            int newDist = distances[current.vertex] + neighbor.weight;
            if (newDist < distances[neighbor.vertex]) {
                distances[neighbor.vertex] = newDist;
                pq.add(new Node(neighbor.vertex, newDist));
            }
        }
    }
    return distances;
}
```

EXAMPLE GRAPH AND OUTPUT

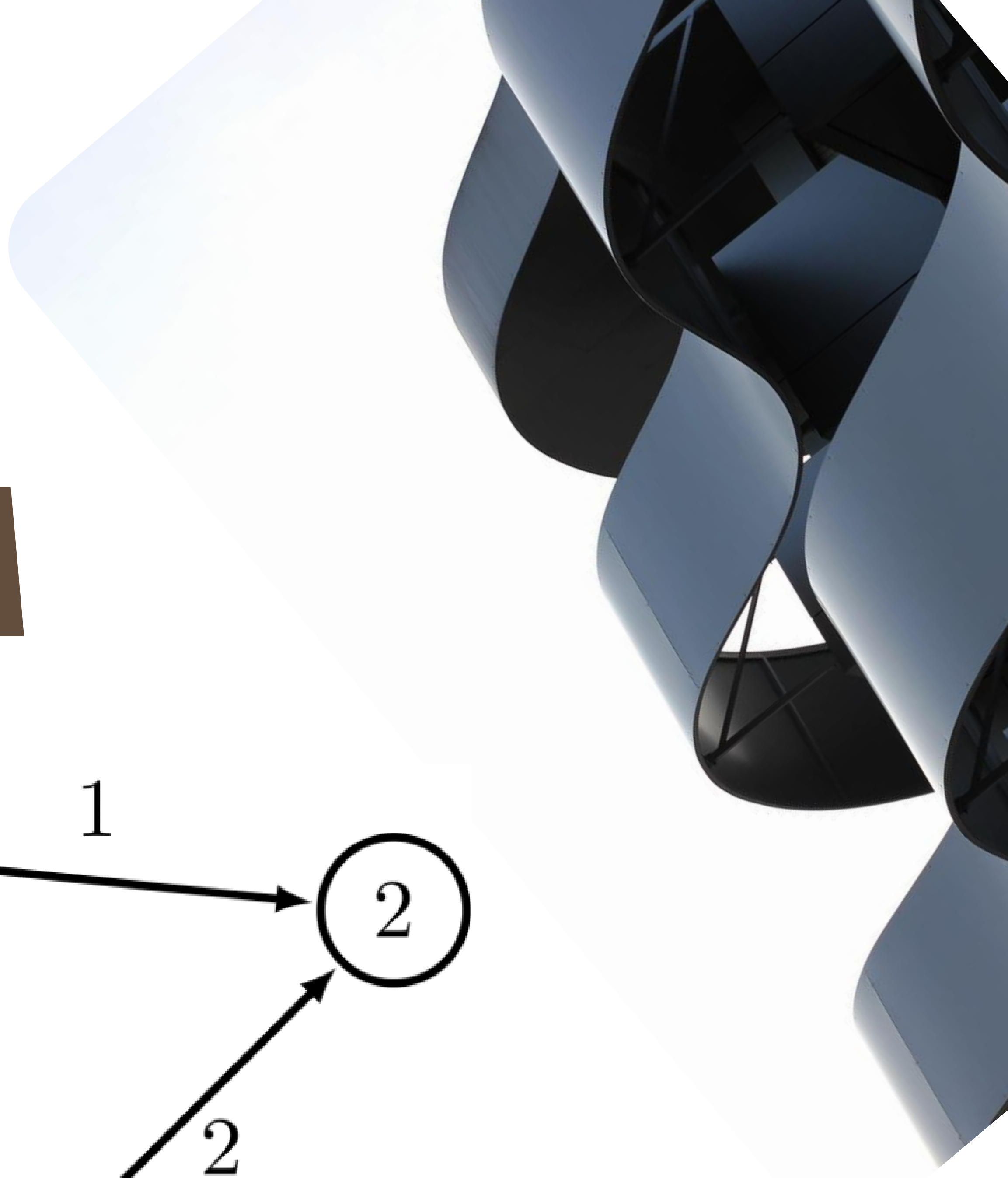
```java
public static void main(String[] args) {
    int n = 5; // Number of nodes
    List<List<Node>> graph = new ArrayList<>();
    for (int i = 0; i < n; i++) graph.add(new ArrayList<>());

    // Adding edges
    graph.get(0).add(new Node(1, 2));
    graph.get(0).add(new Node(2, 4));
    graph.get(1).add(new Node(2, 1));
    graph.get(1).add(new Node(3, 7));
    graph.get(2).add(new Node(3, 3));
    graph.get(3).add(new Node(4, 1));

    int[] distances = dijkstra(graph, 0);
    System.out.println(Arrays.toString(distances)); // Output distances
}
```

Output Example:

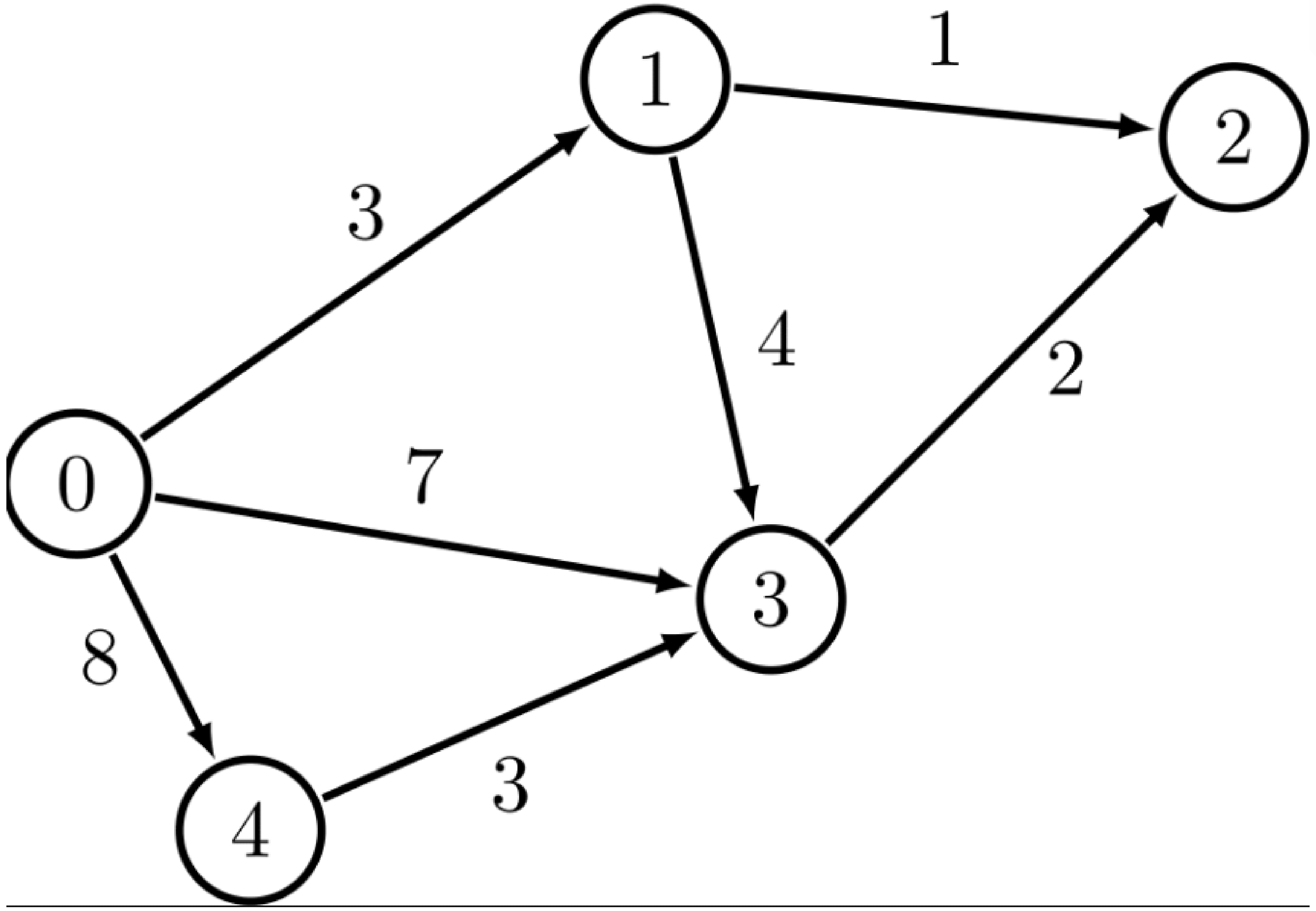
From source node 0: [0, 2, 3, 6, 7].



PROS AND CONS OF DIJKSTRA’S ALGORITHM

Advantages:

- Efficient for graphs with non-negative weights.
- Simple to implement with clear logic.
- Adaptable to various real-world problems.
- Limitations:
- Cannot handle negative weights.
- Computationally intensive for dense graphs.



THANK YOU!