HOANGDM-BH00859

What is DSA?

why it is important?

1. Data Structures:
Data structures are specialized formats for organizing, processing, and storing data. They define how data is stored and accessed in memory.

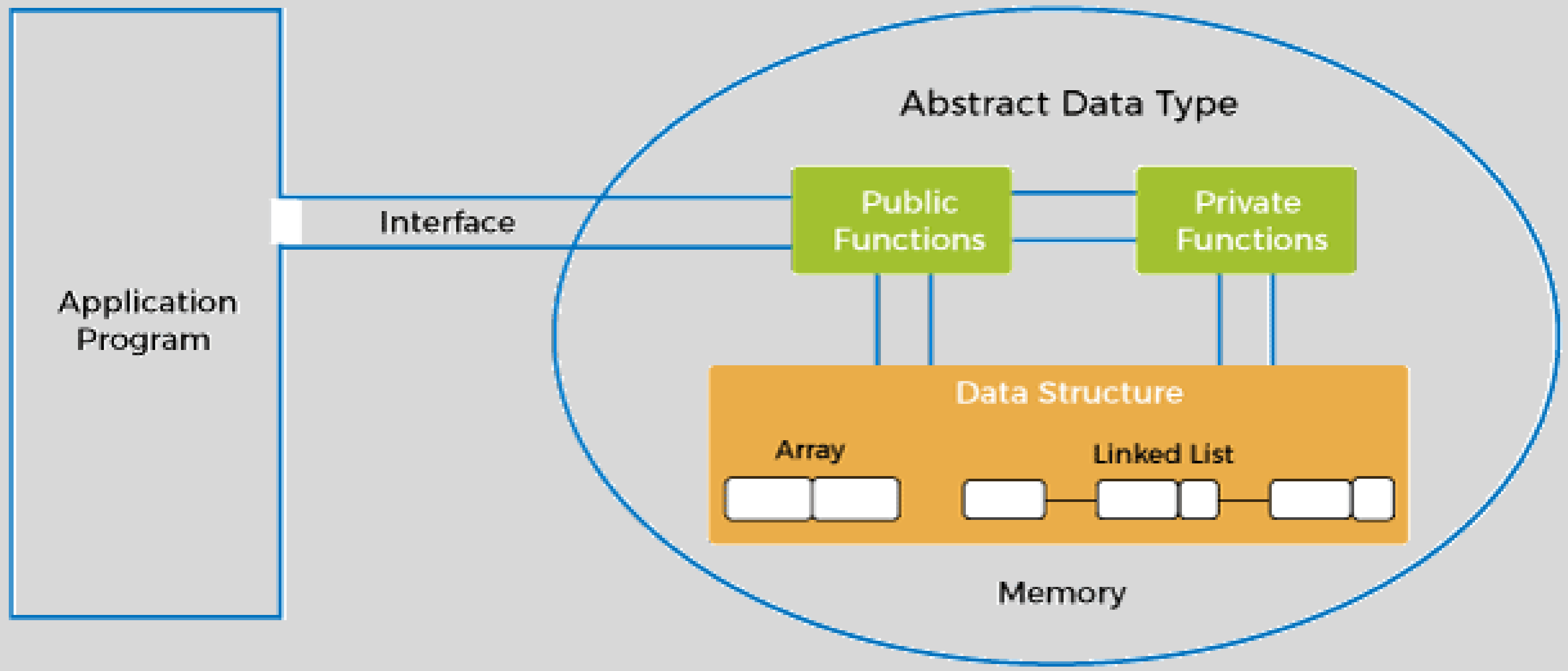
Examples of Data Structures:
Array: A collection of items stored at contiguous memory locations.
Linked List: A series of connected nodes, where each node contains data and a reference to the next node.
Stack: Follows a Last In, First Out (LIFO) principle, like a stack of books.
Queue: Follows a First In, First Out (FIFO) principle, like a line of people.
Hash Table: Stores key-value pairs for quick access using a hash function.
Tree: A hierarchical structure with a root and children, used in databases or file systems (e.g., Binary Tree, Binary Search Tree).
Graph: Consists of vertices (nodes) and edges, used to represent networks like social graphs or routing paths.

2. Algorithms:
An algorithm is a step-by-step procedure or a set of rules to solve a problem or perform a task. Algorithms manipulate data stored in data structures and help in solving computational problems efficiently.

Examples of Algorithms:
Sorting Algorithms:
Bubble Sort: Compares adjacent elements and swaps them if they are in the wrong order.
Merge Sort: Divides the array into halves, sorts them, and merges them back together.
Quick Sort: Uses a pivot element to partition the array and recursively sort partitions.
Searching Algorithms:
Linear Search: Checks each element sequentially until the target is found.
Binary Search: Efficiently searches a sorted array by repeatedly dividing it in half.
Graph Algorithms:
Dijkstra’s Algorithm: Finds the shortest path between nodes in a graph.
Depth-First Search (DFS): Explores as far as possible along a branch before backtracking.
Breadth-First Search (BFS): Explores all neighbors at the present depth before moving deeper.

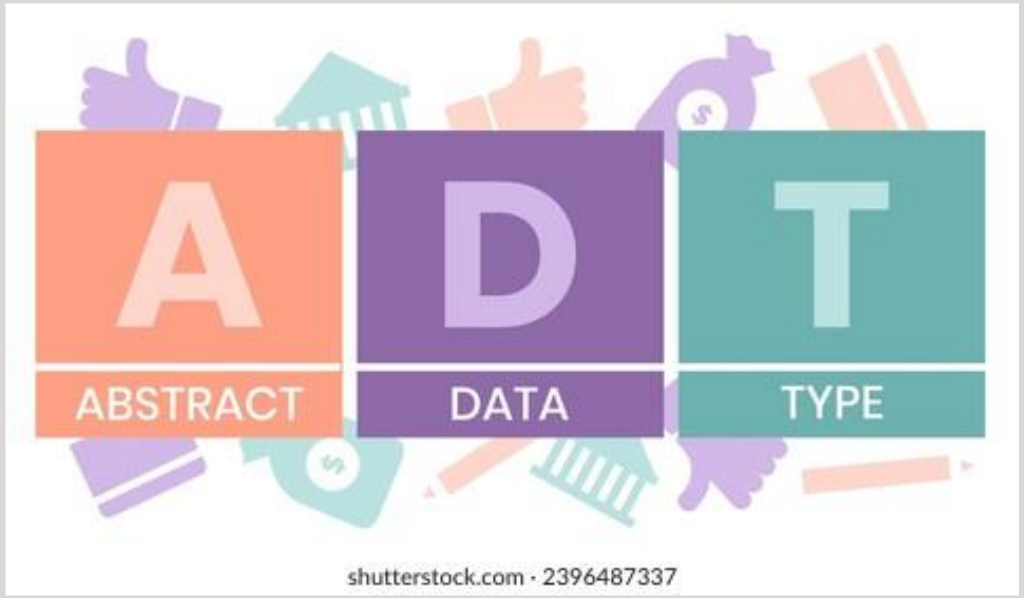
Why DSA is Important:
Efficiency: Choosing the right data structure and algorithm can make a significant difference in the performance of a program. Efficient algorithms minimize the time and space required to process data.
Problem Solving: Many real-world problems can be mapped to problems that can be solved using DSA (e.g., finding the shortest route in GPS navigation, social network analysis, etc.).
Scalability: When working with large datasets, good algorithms and data structures ensure that your program scales well with the input size.

Abstract Data Type



ADT (Abstract Data Type) is a conceptual model that defines a data structure along with the operations that can be performed on it, without specifying how those operations will be implemented. It provides an abstract view of how data is organized and manipulated but hides the underlying implementation details.

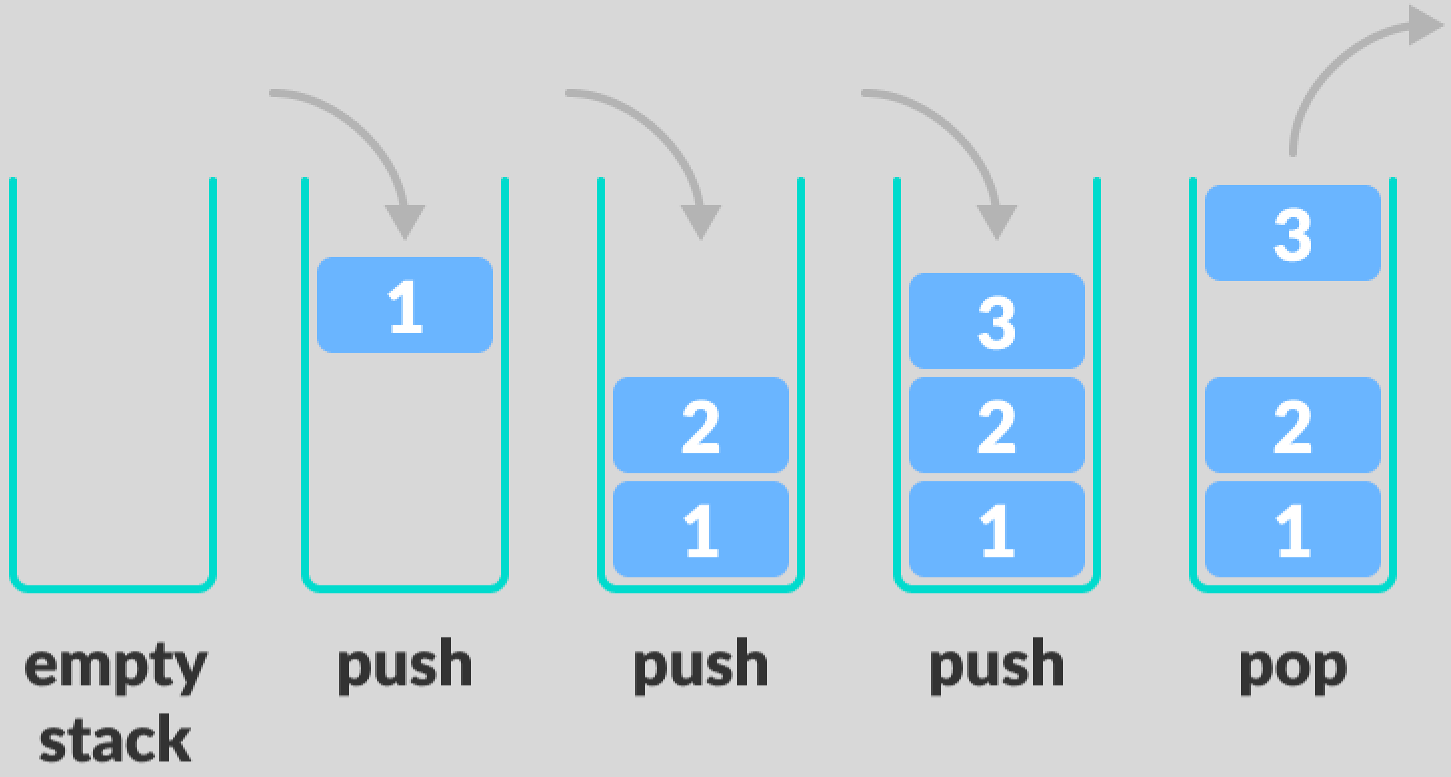
Key Concepts of ADT:
Abstract Representation: An ADT defines what operations are supported and what behaviors the data type should have, but it does not specify how these operations are implemented internally.



Here are some common abstract data types:

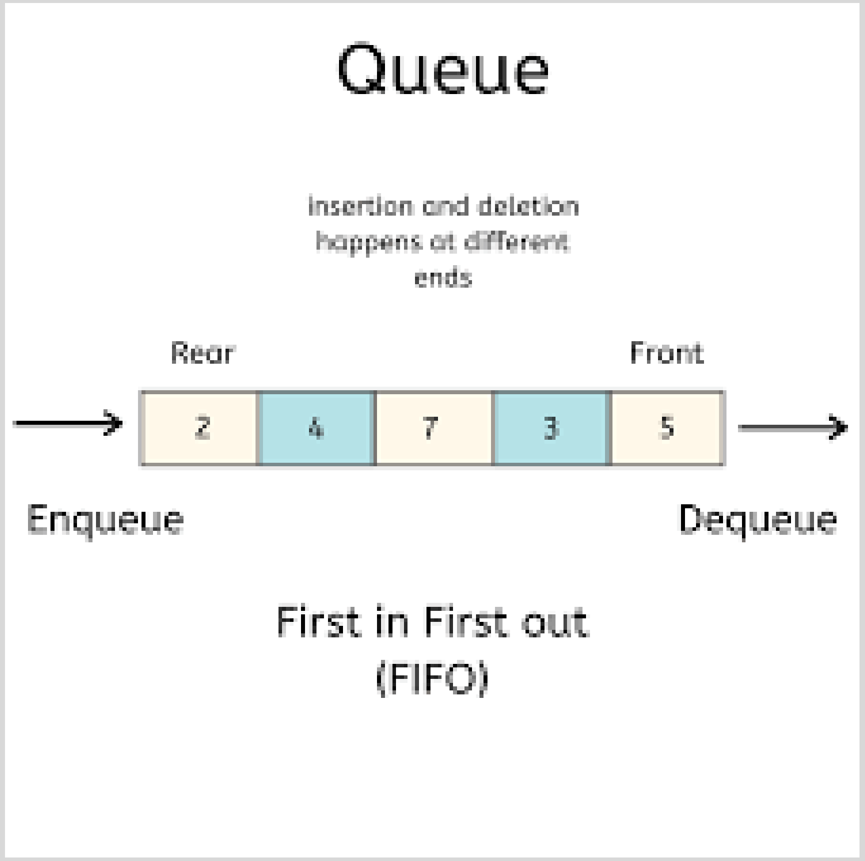
Stack (ADT):

Operations:
push(): Adds an element to the top.
pop(): Removes the top element.
peek(): Views the top element without removing it.
isEmpty(): Checks if the stack is empty.
Summary: Stacks follow the LIFO (Last In, First Out) principle. It doesn't matter whether it's implemented as an array or a linked list; the concept remains the same.



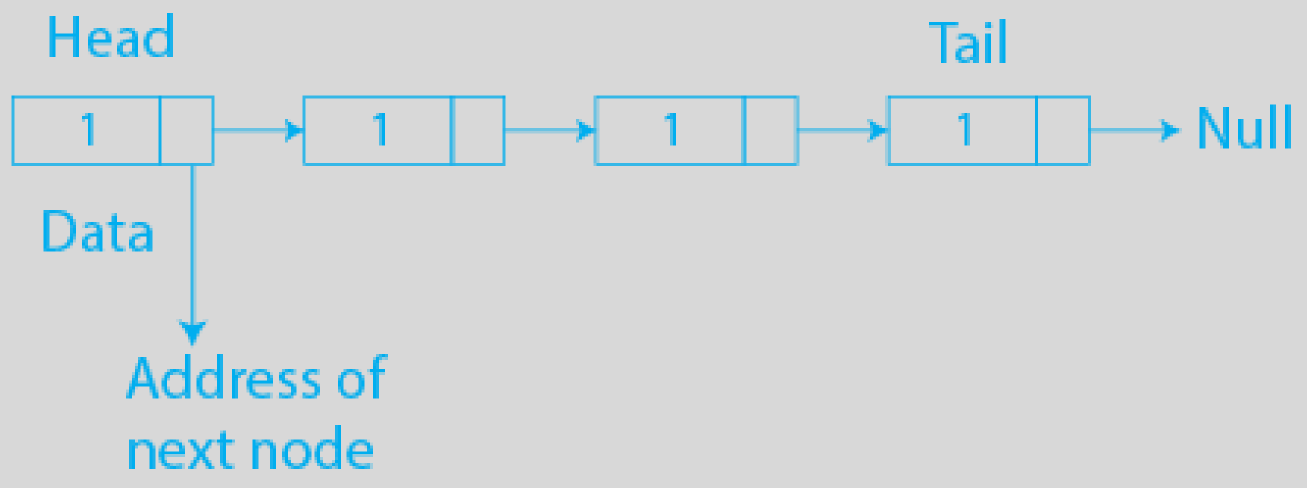
Queue (ADT):

Operations:
enqueue(): Adds an element to the back.
dequeue(): Removes an element from the front.
front(): Views the front element.
isEmpty(): Checks if the queue is empty.
Summary: Queues follow the FIFO (First In, First Out) principle. The implementations may vary, but the concept remains the same.


Queue
insertion and deletion happens at different ends
Rear
Front
2
4
7
3
5
Enqueue
Dequeue
First in First out (FIFO)

List (ADT):

Operations:
insert(): Adds an element to the specified position.
delete(): Removes an element from the specified position.
get(): Retrieves the element at the specified position.
size(): Gets the number of elements in the list.
Summary: Lists allow positional access, but whether they are implemented as arrays, linked lists, or other structures is irrelevant to the user.



Why ADTs are Important:

Abstraction: ADTs promote abstraction by hiding implementation details, allowing users to focus on what operations the data type performs rather than how they are implemented.
Modularity: ADTs allow for modular program design, making it easy to modify the implementation without affecting the rest of the code.
Reusability: Since an ADT specifies an interface, the underlying implementation can be changed or reused in different contexts without modifying the code that uses the ADT.

Compare different between Stack and Queue

1. Basic Principle:
Stack: Operates on a LIFO (Last In, First Out) principle.
The last element added to the stack is the first one to be removed.
Queue: Operates on a FIFO (First In, First Out) principle.
The first element added to the queue is the first one to be removed.
2. Operations:
Stack Operations:
Push (element): Adds an element to the top of the stack.
Pop (): Removes the top element from the stack.
Peek (): Returns the top element without removing it.
isEmpty (): Checks if the stack is empty.
Queue Operations:
Enqueue (element): Adds an element to the end (rear) of the queue.
Dequeue (): Removes an element from the front of the queue.
Front (): Returns the front element without removing it.
isEmpty (): Checks if the queue is empty.

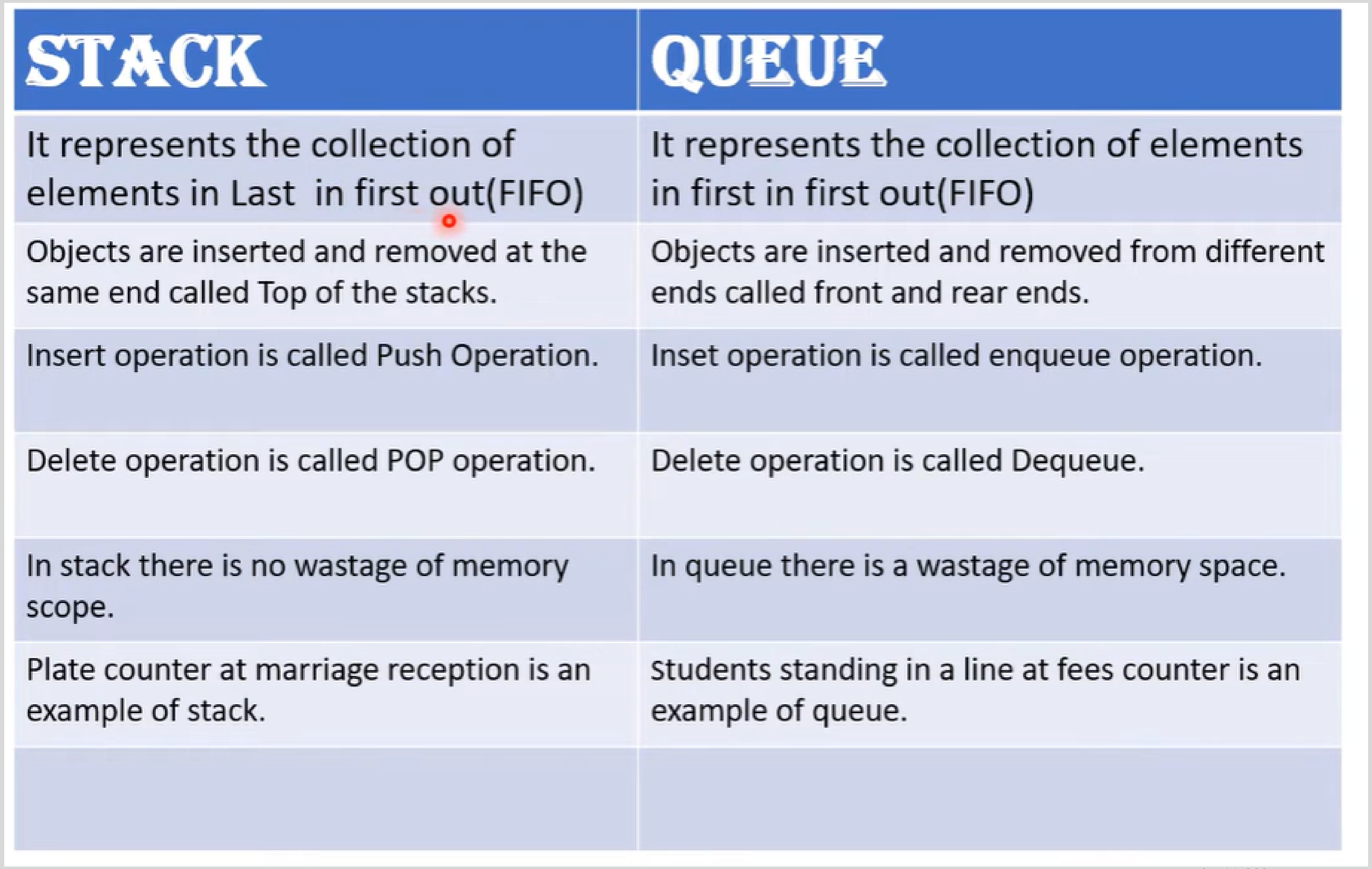
3. Use Cases:
Stack:
Backtracking: Used in algorithms that require undo operations (e.g., solving mazes, browser history, function calls).
Recursive Functions: The call stack used by recursion operates as a stack.
Expression Evaluation: Used in converting and evaluating expressions (infix to postfix conversion).
Queue:
Scheduling: Used in task scheduling (e.g., printer queue, CPU task scheduling).
Breadth-First Search (BFS): Used in BFS for exploring nodes level by level.
Order Processing: Managing processes in real-time systems (e.g., customer service systems, ticket queues).

4. Time Complexity:
Both Stack and Queue operations (insertion, removal, access) generally have the following time complexity:

Push/Pop in Stack: O(1)
Enqueue/Dequeue in Queue: O(1)
The time complexity remains constant regardless of the size of the data structure, as long as the underlying implementation supports efficient access.

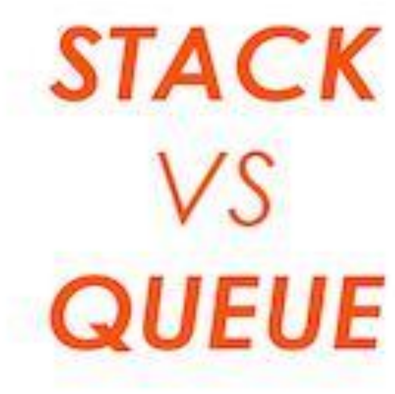
5. Data Structure Type:
Stack: Can be implemented using arrays or linked lists.
Queue: Can also be implemented using arrays or linked lists, though it may involve a circular array for efficient space management.

6. Variation:
Stack:
Double Stack: A stack with two ends.
Queue:
Circular Queue: A queue where the rear and front are connected to make efficient use of memory.
Priority Queue: A queue where elements are dequeued based on priority rather than insertion order.
Deque (Double-ended Queue): Allows insertion and removal from both ends.



STACK	QUEUE
It represents the collection of elements in Last in first out(FIFO)	It represents the collection of elements in first in first out(FIFO)
Objects are inserted and removed at the same end called Top of the stacks.	Objects are inserted and removed from different ends called front and rear ends.
Insert operation is called Push Operation.	Inset operation is called enqueue operation.
Delete operation is called POP operation.	Delete operation is called Dequeue.
In stack there is no wastage of memory scope.	In queue there is a wastage of memory space.
Plate counter at marriage reception is an example of stack.	Students standing in a line at fees counter is an example of queue.

How many ways are there to implement Stack and Queue?



1. Stack Implementations:
a. Array-Based Stack:
How it works: An array is used to store elements, and a pointer (top) keeps track of the index of the last element pushed onto the stack.

Key Operations:

Push: Adds an element to the top of the array.
Pop: Removes the element from the top and decreases the pointer.
Peek: Returns the element at the top without removing it.
b. Linked List-Based Stack:
How it works: A linked list is used, where each node contains a value and a pointer to the next node. The top is represented by the head of the linked list.

Key Operations:
Push: Adds a new node at the head (top).
Pop: Removes the node at the head (top) and reassigns the pointer to the next node.
Peek: Returns the value at the head.

c. Dynamic Array-Based Stack:
How it works: Similar to an array-based stack, but the size of the array grows or shrinks dynamically as elements are pushed or popped.

Key Operations:

Push: If the array is full, double its size and push the element.
Pop: Shrinks the array if it becomes under-utilized.

2. Queue Implementations:

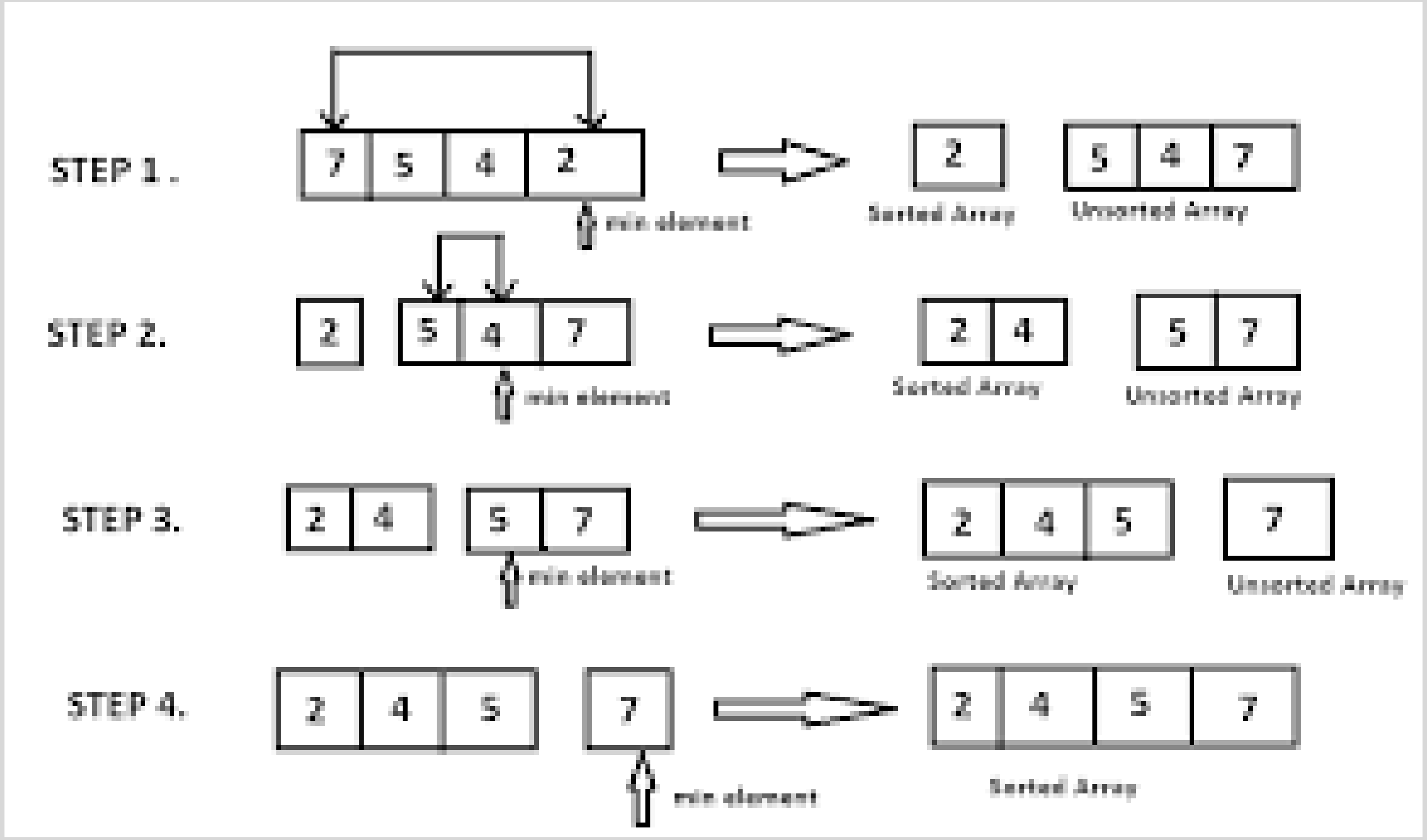
a. Array-Based Queue:
How it works: Elements are stored in a contiguous array. Two pointers, front and rear, track the positions of the first and last elements respectively.

Key Operations:
Enqueue: Adds an element at the rear.
Dequeue: Removes an element from the front.
Front: Returns the element at the front without removing it.

b. Circular Array-Based Queue (Ring Buffer):
How it works: A modification of the array-based queue where the array wraps around itself to form a circular buffer. The rear pointer wraps around to the beginning of the array when it reaches the end.
Key Operations:
Enqueue: Adds an element at the rear, wrapping around if necessary.
Dequeue: Removes an element from the front, wrapping around if necessary.
c. Linked List-Based Queue:
How it works: A singly linked list is used where each node contains a value and a pointer to the next node. The front and rear pointers track the first and last nodes in the list.
Key Operations:
Enqueue: Adds a node at the rear.
Dequeue: Removes the node from the front.
Front: Returns the value at the front.
d. Double-Ended Queue (Deque):
How it works: A deque allows elements to be added and removed from both the front and rear. It can be implemented using arrays or linked lists.
Key Operations:
Add Front: Adds an element at the front.
Add Rear: Adds an element at the rear.
Remove Front: Removes an element from the front.
Remove Rear: Removes an element from the rear.

Sorting Algorithms

1.Selection Sort



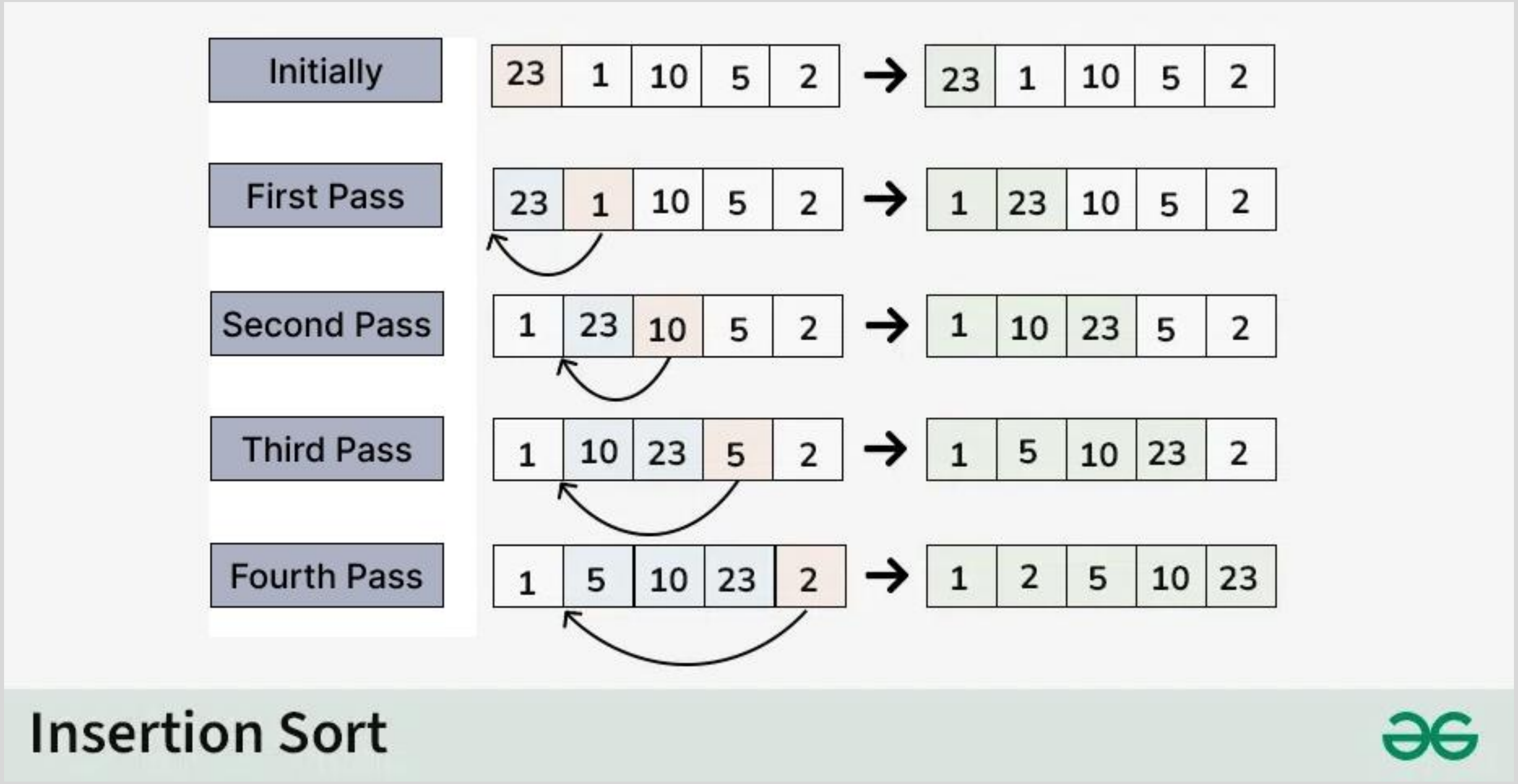
Description: Selection Sort sorts by finding the smallest (or largest, depending on the sort order) element in the list and moving it to the top. It then repeats with the rest of the list, finding the next smallest element and moving it to the next position. This process continues until the list is completely sorted.

Time Complexity:
1. Best Case: $O(n2)$O(n 2)
2. Average Case: $O(n2)$O(n 2)
3. Worst Case: $O(n2)$O(n 2)

Features:

1. Not a stable sorting algorithm, meaning it can change the order of elements with equal values.
2. Requires fewer calculations and swaps than some other algorithms, but is not as fast.
3. Suitable for small lists or when memory is limited, as no additional memory is needed.

2. Insertion Sort



Description: Insertion Sort sorts by building a sorted sublist at the beginning of the array. For each element in the array, it "inserts" that element into the appropriate position in the sorted sublist, ensuring that the previous elements are all sorted.

Time Complexity:

1. Best Case: $O(n)$O(n) – when the list is sorted.
2. Average Case: $O(n2)$O(n 2)
3. Worst Case: $O(n2)$O(n 2)

Features:

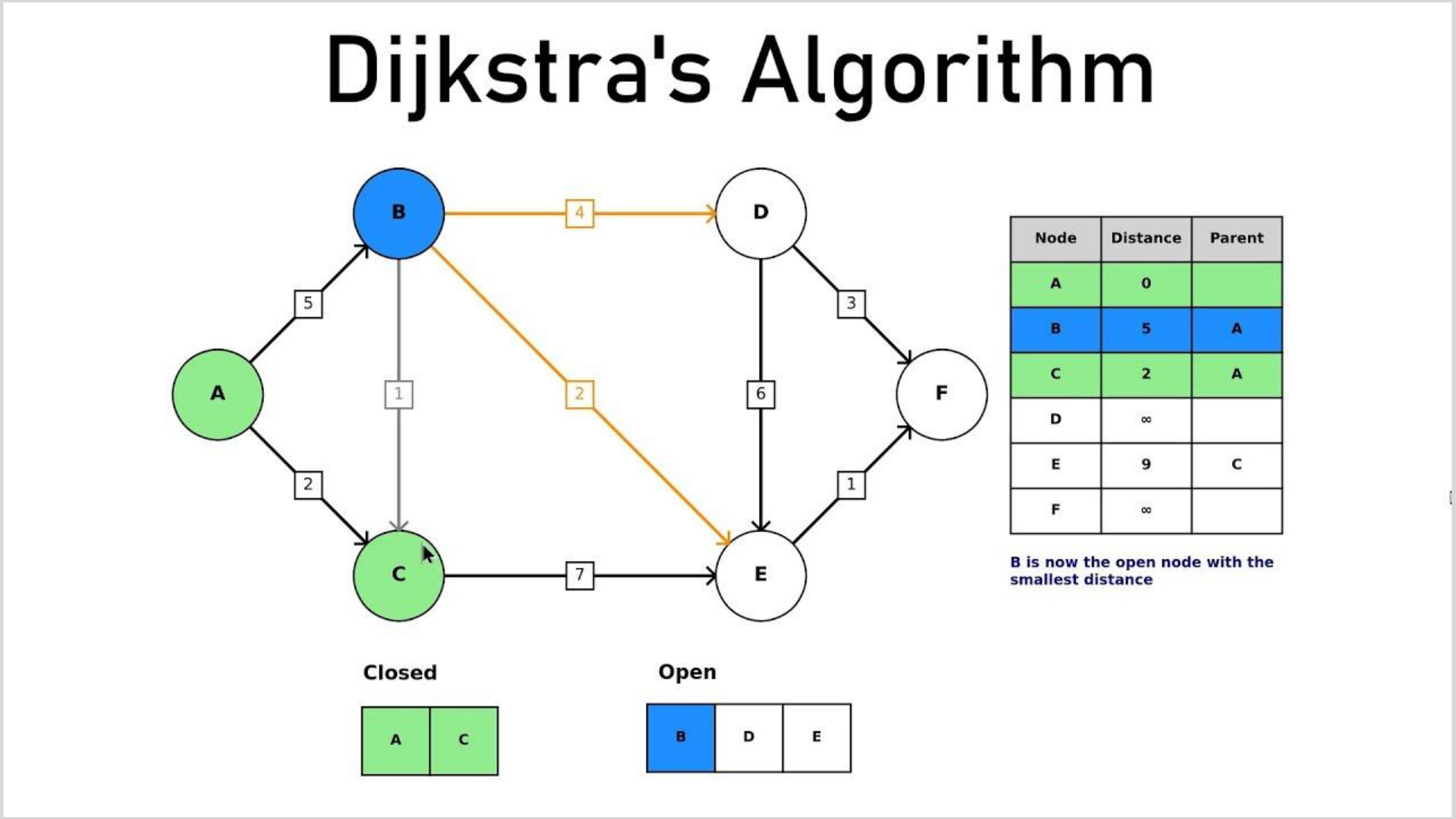
1. It is a stable sorting algorithm, meaning it does not change the order of elements with equal values.
2. It is efficient for small lists or lists that are almost sorted.
3. It performs well when you need to sort directly in the array without requiring additional memory.

Compare Selection Sort and Insertion Sort

Attribute	Selection Sort	Insertion Sort
Best Case Complexity	O(n2)	O(n)
Average Case Complexity	O(n2)	O(n2)
Worst Case Complexity	O(n2)	O(n2)
Stability	Unstable	Stable
Number of Swaps	Fewer than Insertion Sort	More if the list is unsorted
Applications	Suitable for small lists	Good for small or nearly sorted lists
Implementation Complexity	Easy	Easy
Performance on Sorted Data	No improvement	Improved (best case O(n))

Shortest Path Algorithms.

1.Dijkstra's Algorithm



Node	Distance	Parent
A	0	
B	5	A
C	2	A
D	∞	
E	9	C
F	∞	

B is now the open node with the smallest distance

Description: Dijkstra’s algorithm is a greedy algorithm that finds the shortest path from a single source node to all other nodes in a weighted, directed graph with non-negative edge weights. It operates by selecting the node with the smallest known distance from the source node and then updating the distances to its neighboring nodes.

How it Works:

Initialize the distance of the source node to 0 and all other nodes to infinity (∞).

Mark all nodes as unvisited.

For the current node, examine its unvisited neighbors. Calculate their tentative distances from the source by summing the distance to the current node and the weight of the edge connecting the two.

If the tentative distance of a neighbor is less than the previously recorded distance, update the shortest distance for that neighbor.

Mark the current node as visited and move to the unvisited node with the smallest distance. Repeat this process until all nodes have been visited or the smallest tentative distance among the unvisited nodes is infinity.

Complexity:

Time Complexity: $O((V+E)\log V)$ for a graph with V vertices and E edges using a min-heap priority queue.

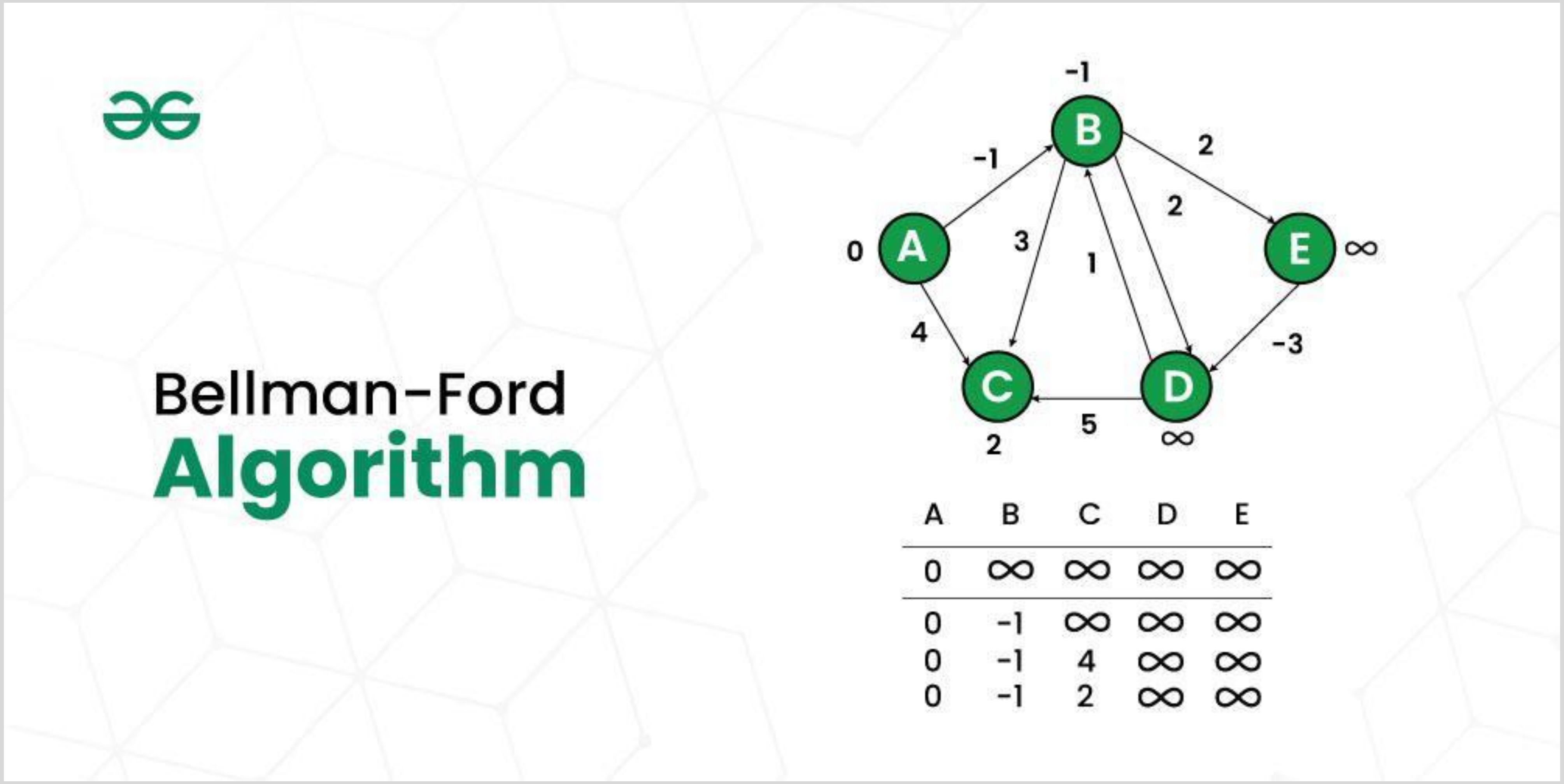
Space Complexity: $O(V)$, for storing distances and the priority queue.

Limitations:

Requires all edge weights to be non-negative; it doesn’t work correctly with graphs that have negative edge weights.

Efficient for dense graphs, but the performance can decrease for graphs with a large number of vertices.

2. Bellman-Ford Algorithm



A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞

Description: The Bellman-Ford algorithm is more versatile than Dijkstra’s algorithm and can handle graphs with negative edge weights. It is designed to find the shortest path from a single source node to all other nodes in a weighted, directed graph, even when there are negative weights. However, it doesn’t work with graphs containing negative weight cycles (cycles whose total weight is negative), as the shortest path can become infinitely short.

How it Works:

1. Initialize the distance of the source node to 0 and all other nodes to infinity (∞).
2. Relax all edges in the graph $V-1$ times (where VVV is the number of vertices). For each edge, update the distance to the target vertex if the distance to the source vertex plus the edge weight is less than the known distance to the target vertex.
3. Perform an additional pass to check for negative weight cycles. If any distance can still be updated, then a negative weight cycle exists, indicating that the graph doesn’t have a shortest path for certain nodes.

Complexity:

- **Time Complexity:** $O(V \times E)$, where V is the number of vertices and E is the number of edges.
- **Space Complexity:** $O(V)$, for storing distances.

Limitations:

- Slower than Dijkstra’s algorithm for graphs without negative weights.
- May have a high time complexity on large graphs, making it less efficient for dense networks.