



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.

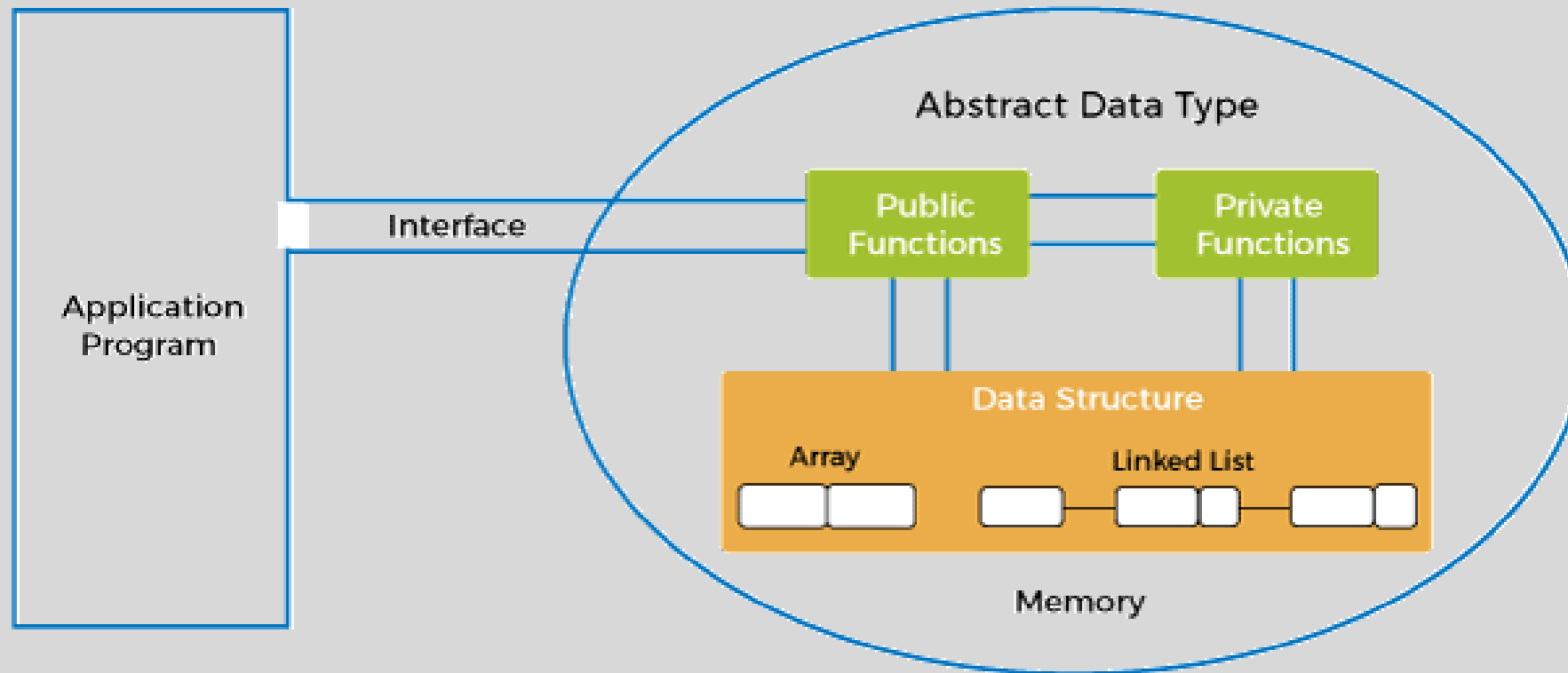
Why DSA is Important:

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.

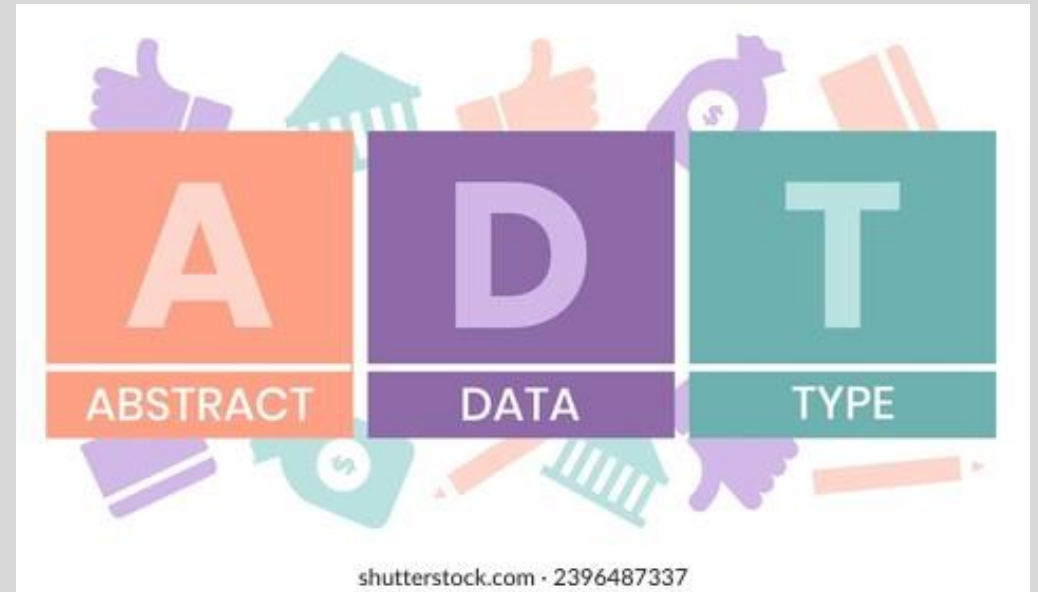
Abstract Data Type



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:

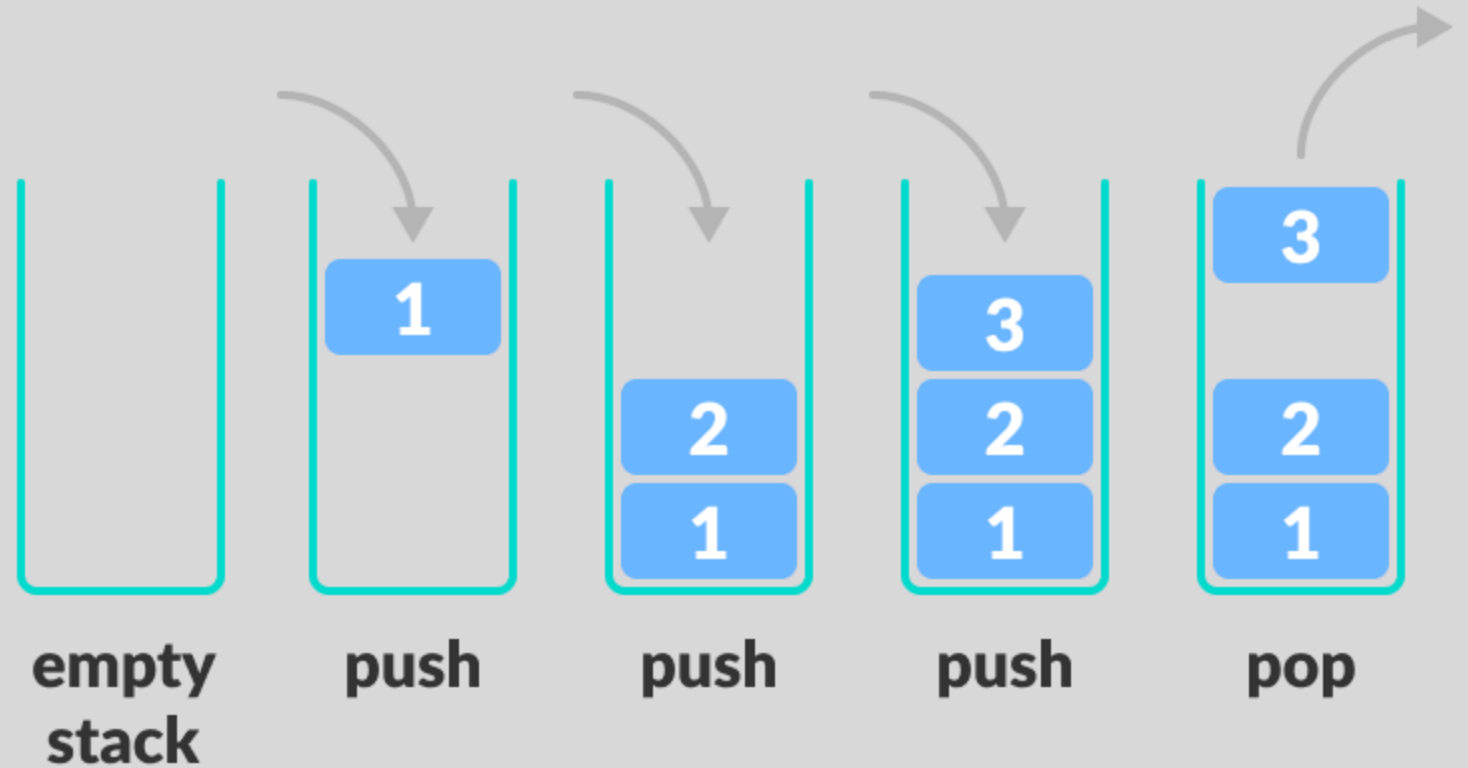
push(): Adds an element to the top.

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:

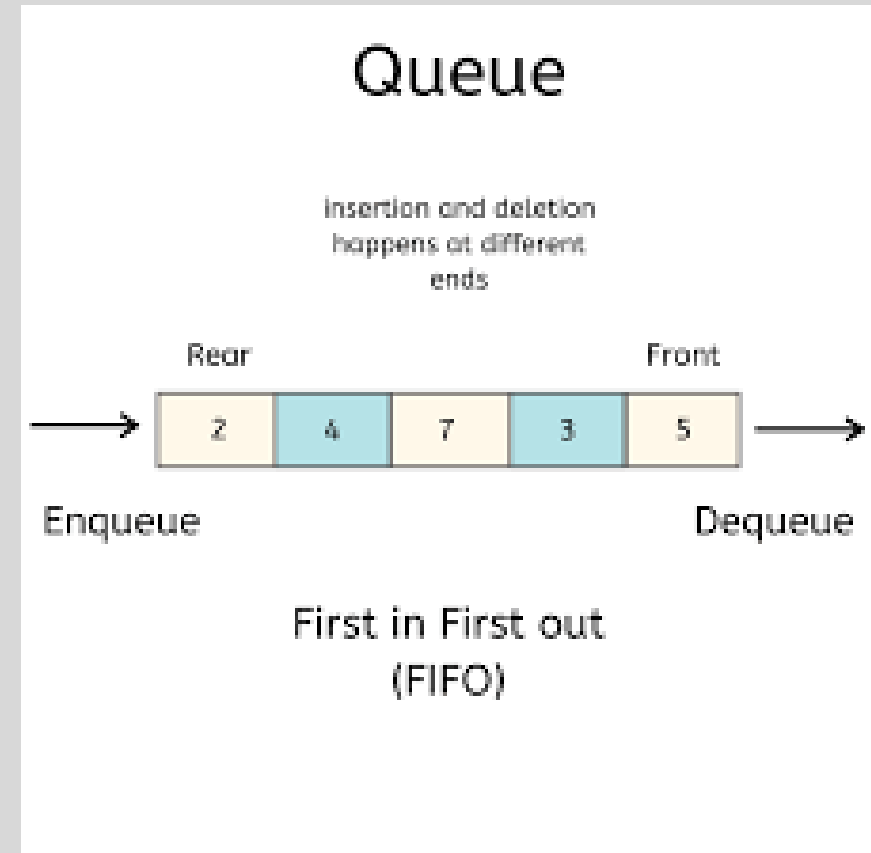
enqueue(): Adds an element to the back.

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.



List (ADT):

Operations:

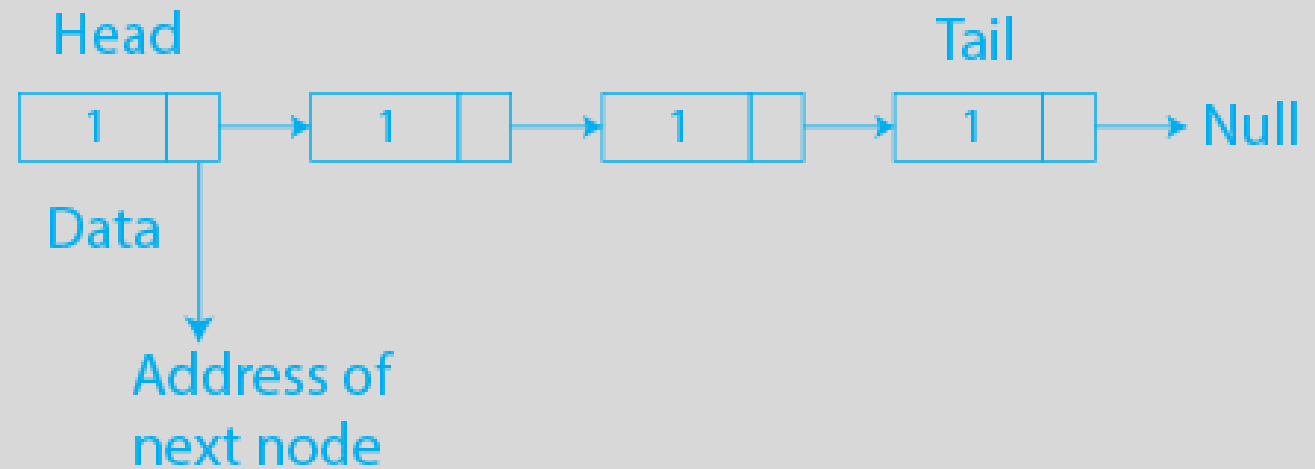
insert(): Adds an element to the specified position.

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

It represents the collection of elements in Last in first out(LIFO)

Objects are inserted and removed at the same end called Top of the stacks.

Insert operation is called Push Operation.

Delete operation is called POP operation.

In stack there is no wastage of memory scope.

Plate counter at marriage reception is an example of stack.

QUEUE

It represents the collection of elements in first in first out(FIFO)

Objects are inserted and removed from different ends called front and rear ends.

Inset operation is called enqueue operation.

Delete operation is called Dequeue.

In queue there is a wastage of memory space.

Students standing in a line at fees counter is an example of queue.

How many ways are there to implement Stack and Queue?

STACK VS QUEUE

Stack



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.

