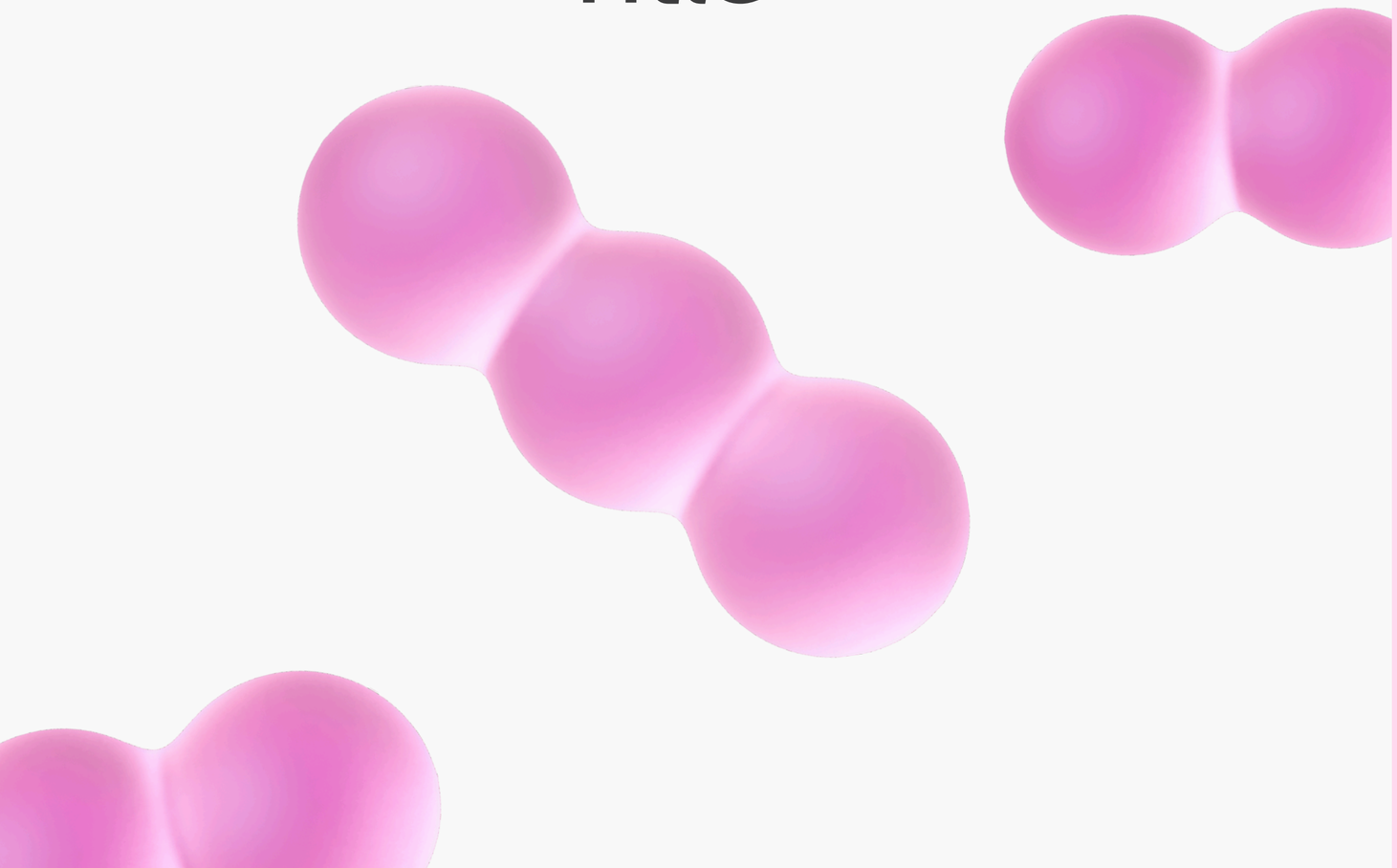
DATA STRUCTURES AND ALGORITHMS (DSA)

Assignment Presentation

PHANNHATLINH_BH00988

Title



DSA and ADT

Understanding Stack ADT

Understanding Queue ADT

Stack vs Queue

Implementing Stack and Queue

Sorting Algorithms - Bubble Sort

Sorting Algorithms - Selection Sort

Complexity Comparison

Shortest Path Algorithms

Dijkstra's Algorithm

Conclusion

What is DSA?

Data Structures and Algorithms (DSA) is a foundational concept in computer science and programming that focuses on organizing and manipulating data efficiently to solve problems.

- Data Structures provide ways to store and organize data (such as arrays, linked lists, stacks, and queues), each with specific capabilities for accessing, inserting, and deleting data.
- Algorithms are step-by-step procedures for solving specific problems, often involving data structures. Examples include searching, sorting, and graph traversal algorithms.

Importance of DSA:

Efficiency: Enables efficient use of resources (time and memory), which is crucial in handling large datasets.

Problem Solving: Helps in designing solutions for complex tasks and real-world applications.

Scalability: Good data structures and algorithms ensure that programs can handle growing data needs without performance issues.

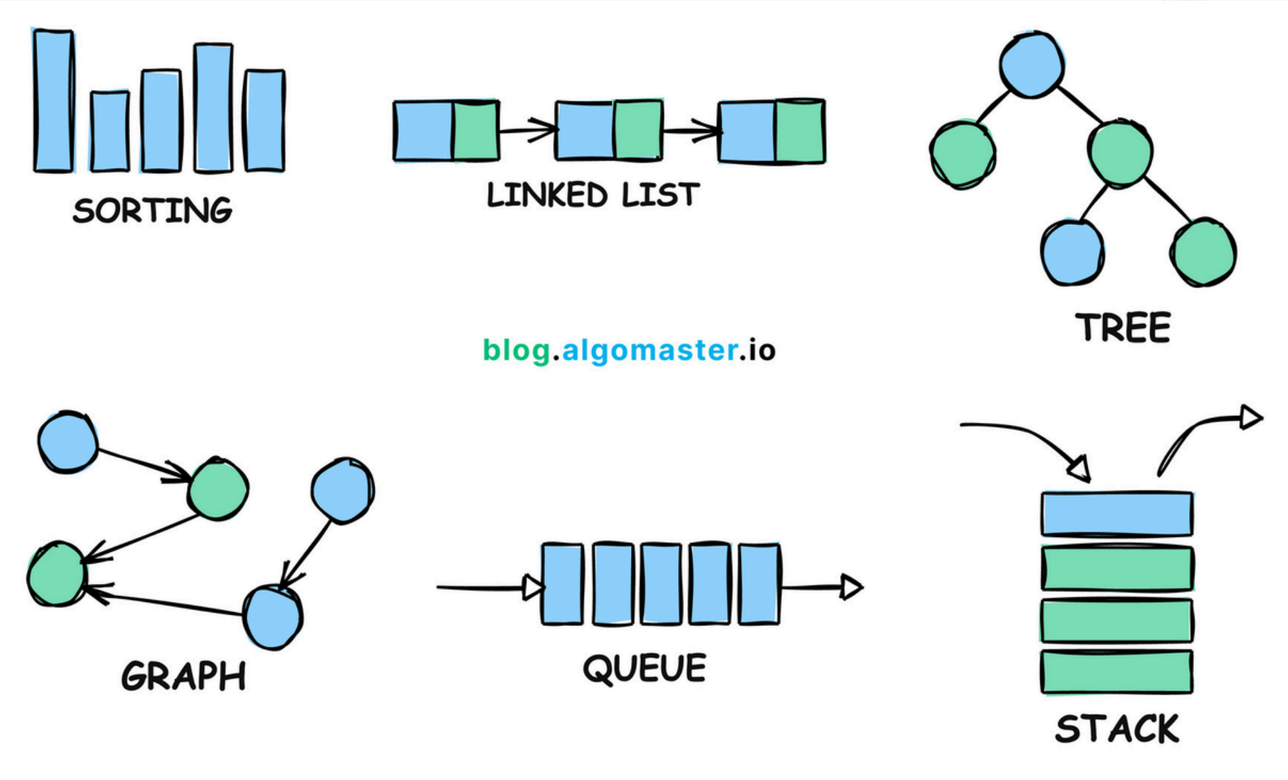
What is ADT?

Abstract Data Type (ADT) is a model for data structures that defines the behavior of the data (i.e., operations that can be performed on the data) without specifying the actual implementation details.

Key Points about ADT:
Encapsulation: ADTs encapsulate data and operations, hiding the specifics of how data is stored and managed.
Flexibility: Developers can change the underlying data structure without affecting code that relies on the ADT, enabling more adaptable and maintainable code.

Common Examples of ADTs

- Stack: Follows the Last In, First Out (LIFO) principle, with operations like push, pop, and peek.
- Queue: Follows the First In, First Out (FIFO) principle, supporting operations like enqueue and dequeue.
- List: A collection that supports ordered storage of elements, with operations such as insert, remove, and access.
- Tree: A hierarchical structure with nodes, allowing for operations like insert, delete, and traversal methods (preorder, inorder, postorder).



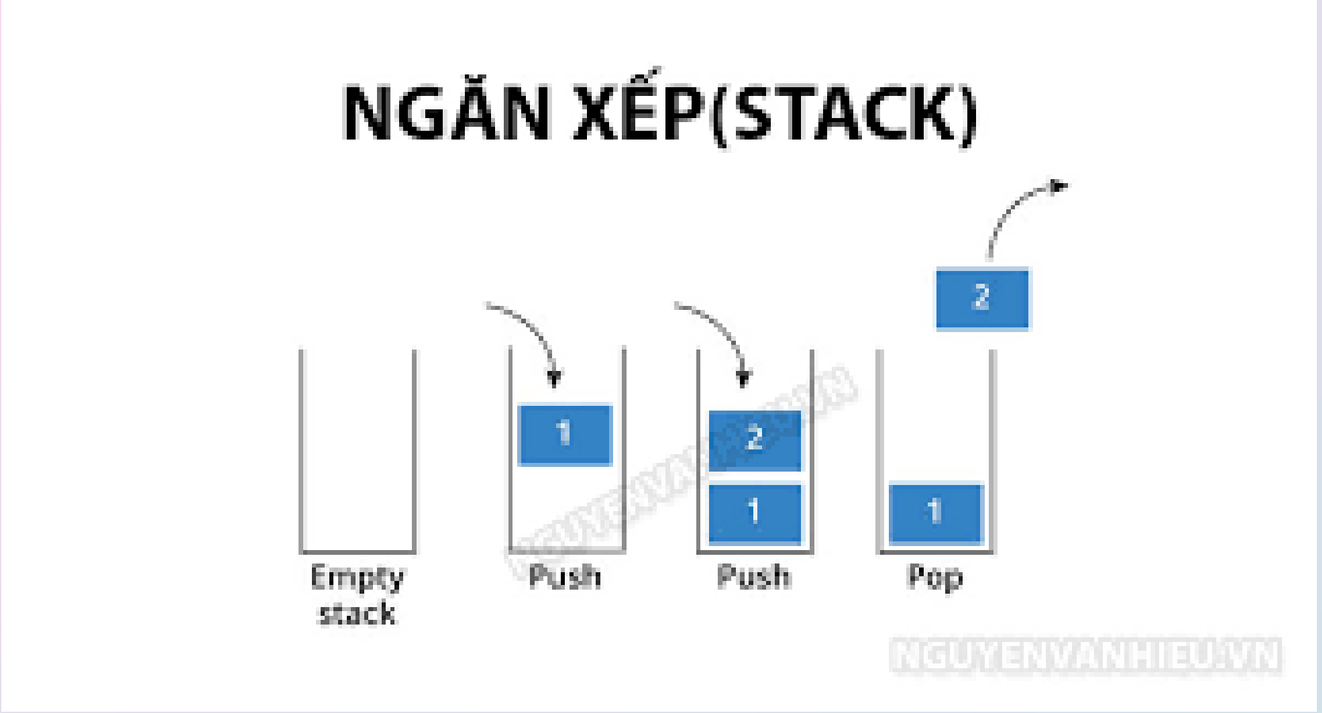
UNDERSTANDING STACK ADT

Stack ADT Overview

A stack is a linear data structure where elements are stored in the LIFO (Last In First Out) principle where the last element inserted would be the first element to be deleted. A stack is an Abstract Data Type (ADT), that is popularly used in most programming languages.

Key Operations in Stack

- Push(element): Adds element to the top.
- Pop(): Removes and returns the top element.
- Peek(): Returns the top element without removing it.
- isEmpty(): Checks if the stack has no elements.



- Push Operation :
- Push(1) : Number 1 is added to the stack. It becomes the top element.
- Push(2) : Number 2 is added to the top of 1. Now 2 is the top element.
- Push(3) : Number 3 is added to the top of 2. Now 3 is the top element.
- Each push operation adds a new element to the top of the stack, following the "Last In, First Out" (LIFO) principle.
- Pop Operation :
- Pop() : The top element, 3, is removed from the stack. After popping, 2 becomes the top element.
- Additional stack operations (not shown in the image but related):
-
- Peek() : This operation will return the current top element (in this case 2) without removing it.
- isEmpty() : Checks if the stack is empty. In this case, it will return false because there are still elements on the stack.

Understanding Queue ADT

A queue is an Abstract Data Type (ADT) that operates on a First In, First Out (FIFO) basis.

Key Operations in Queue

enqueue(): Adds an item to the end of the queue.
dequeue(): Removes the item at the front of the queue.
peek(): Views the item at the front without removing it.
isEmpty(): Checks if the queue is empty.

Comparison: Stack vs. Queue

- Differences Between Stack and Queue

Aspect	Stack	Queue
Principle	Last In, First Out (**LIFO**)	First In, First Out (**FIFO**)
Main Operations	`Push` (add to top)	`Enqueue` (add to rear)
	`Pop` (remove from top)	`Dequeue` (remove from front)
Access Pattern	Accesses elements from the top	Accesses elements from the front
Use Cases	- Function call management	- Task scheduling
	- Expression evaluation	- Real-time processing
	- Undo/Redo operations	- Breadth-First Search (BFS)
Insertion Point	Top only	Rear only
Deletion Point	Top only	Front only
Example	Stack of books, browser back button	Line at a checkout, print job requests

Stack and queue are basic data structures that serve different purposes based on their distinct characteristics and operations. A stack follows the LIFO principle and is used for backtracking, managing function calls, and evaluating expressions. A queue follows the FIFO principle and is used for task scheduling, resource management, and breadth-first search algorithms.

Sorting Algorithms - Bubble Sort

Introduction to Sorting

- What is Sorting?

Sorting algorithms are used to rearrange an array or list of elements in order.

Review the list from start to finish,
- check each adjacent pair,
- if the order of the two elements in the pair is incorrect, swap their positions,
- continue this process until the end of the list, swapping any pairs that are out of order.
- After reviewing the entire list, the position at the end of the list is the correct position.
- Repeat the process with the new list consisting of n-1 elements, repeating the steps above until there are no more elements to review

Sorting Algorithms - Selection Sort

Definition: Selection Sort is a comparison-based sorting algorithm that repeatedly finds the minimum element from the unsorted part of the array and places it at the beginning of the sorted portion.
Process: The algorithm divides the array into a sorted and an unsorted region. Each pass selects the smallest element from the unsorted region and swaps it with the leftmost unsorted element, expanding the sorted portion by one element.

Selection Sort (Quick Explanation)

1. Start with an unsorted array.
2. Find the minimum element in the unsorted portion of the array.
3. Swap it with the first element of the unsorted portion.
4. Move to the next position and repeat steps 2-3 until the array is sorted.

Example: For the array 14 33 27 10 35 19 42 44:

- Pass 1: Find minimum 10, swap with 14 → 10 33 27 14 35 19 42 44
- Pass 2: Find minimum 14, swap with 33 → 10 14 27 33 35 19 42 44
- Continue until sorted: 10 14 19 27 33 35 42 44.

Final Sorted Array: 10 14 19 27 33 35 42 44

Complexity Comparison - Bubble Sort vs Selection Sort

Comparison Table

Aspect	Selection Sort	Bubble Sort
Complexity	$O(n^2)$ for all cases	$O(n^2)$ worst ca
Swaps	Fewer swaps (only when minimum is out of place)	Many swaps (every time adjacent elements are wrong)
Passes Required	n-1 passes	n-1 passes, but may exit early if sorted
Best For	Small datasets	Small or mostly sorted datasets

Buble Sort

Process: Repeatedly compares adjacent elements and swaps them if they are in the wrong order. This process "bubbles" larger elements to the end with each pass.
Swaps: Generally more swaps, as each adjacent element pair is compared and possibly swapped in every pass.
Best Use: Very small or mostly sorted datasets; simple but inefficient for large arrays

Selection Sort

Process: Finds the smallest (or largest) element in the unsorted portion of the array and moves it to the sorted portion. This repeats until all elements are sorted.
Swaps: Fewer swaps compared to Bubble Sort (only one per pass if necessary).
Best Use: Small datasets or when minimizing swaps is important.

Shortest Path Algorithms

What are the Shortest Path Algorithms?

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.

2 types of shortest path algorithms.

Dijkstra's Algorithm

Bellman-Ford Algorithm

Dijkstra's Algorithm

- Dijkstra's Algorithm is a greedy algorithm used to find the shortest path from a starting node to all other nodes in a graph with non-negative edge weights.
- It’s often used in network routing and GPS navigation systems to determine the shortest route.

How It Works:

1. Initialize Distances: Set the starting node's distance to 0 (in this case, Node 0) and all other nodes' distances to infinity.
2. Select the Node with the Smallest Distance: Begin from the starting node (Node 0), and visit each of its neighbors.
3. Update Distances: For each neighbor, if the calculated path is shorter, update the distance.
4. Mark as Visited: Once a node’s shortest path is found, mark it as visited and do not reprocess it.
5. Repeat Steps until all nodes are processed.

Bellman-Ford Algorithm

- The Bellman-Ford Algorithm is a dynamic programming algorithm that can find the shortest path from a single source to all other nodes, even in graphs with negative edge weights.
- Commonly used in networking and financial models where paths might have both gains and losses.

How It Works:

1. Initialize distances: Set the starting node's distance to 0 and all others to infinity.
2. Relax edges: For each edge in the graph, update the distance to the destination node if a shorter path is found. Repeat this process for V - 1 passes (where VVV is the number of vertices).
3. Detect negative cycles: After V - 1 passes, perform one more pass. If any distance can still be updated, a negative-weight cycle exists.

Thank you for listening!