**DSA**

## What is DSA?

### ****DSA**** = ****Data Structures + Algorithms****

### ****Data Structures**** = Smart ways to ****store and organize data****

Examples:

* Arrays(store multiple data but same data type and size also fixed)
* Linked Lists ()
* Stacks
* Queues
* Trees
* Graphs
* HashMaps

### ****Algorithms**** = Step-by-step instructions to ****solve problems****

Great! Let’s go through each data structure one by one, with simple and clear theoretical explanations — perfect for interview preparation.

## 🔹 1. ****Array****

### 🔸 Definition:

An array is a **collection of elements** stored in **contiguous memory locations**, where each element can be accessed using its **index**.

### 🔸 Key Concepts:

* **Fixed size**: You must define the size when creating it.
* **Indexing starts at 0**.
* Stores elements of the **same data type**.
* Memory is allocated in a continuous block.

### 🔸 Operations:

* **Access**: O(1)
* **Insert/Delete**: O(n) (because elements may need to be shifted)

### 🔸 Example:

int arr[5] = {10, 20, 30, 40, 50};

cout << arr[2]; // Outputs 30

### 🔸 Real-World Use:

* Storing exam scores, days of the week, etc.

## 🔹 2. ****Linked List****

### 🔸 Definition:

A linked list is a **linear data structure** where each element (node) contains **data** and a **pointer** to the next node.

### 🔸 Key Concepts:

* **Dynamic size**: Grows/shrinks as needed.
* **No contiguous memory** needed.
* Types: **Singly**, **Doubly**, **Circular** Linked Lists.

### 🔸 Operations:

* **Insert/Delete**: O(1) at beginning, O(n) at end
* **Access/Search**: O(n)

### 🔸 Example:

struct Node {

int data;

Node\* next;

};

### 🔸 Real-World Use:

* Implementing stacks/queues, playlist navigation.

## 🔹 3. ****Stack****

### 🔸 Definition:

A stack is a **Last In, First Out (LIFO)** data structure. The last element added is the first one removed.

### 🔸 Key Concepts:

* **push()** – add to top
* **pop()** – remove from top
* **peek()** – view top item
* Can be implemented using array or linked list.

### 🔸 Operations:

* Push/Pop: O(1)

### 🔸 Example:

stack<int> s;

s.push(10);

s.pop();

### 🔸 Real-World Use:

* Undo features, browser history, function calls.

## 🔹 4. ****Queue****

### 🔸 Definition:

A queue is a **First In, First Out (FIFO)** data structure. The first element added is the first removed.

### 🔸 Key Concepts:

* **enqueue()** – add to rear
* **dequeue()** – remove from front
* Types: **Simple Queue**, **Circular Queue**, **Priority Queue**, **Deque**

### 🔸 Operations:

* Enqueue/Dequeue: O(1)

### 🔸 Example:

queue<int> q;

q.push(1);

q.pop();

### 🔸 Real-World Use:

* Print queues, task scheduling, customer service systems.

## 🔹 5. ****Tree****

### 🔸 Definition:

A tree is a **non-linear hierarchical** structure with a **root node** and **child nodes**.

### 🔸 Key Concepts:

* Each node can have 0 or more children.
* **Binary Tree**: Each node has max 2 children.
* **BST (Binary Search Tree)**: Left child < Root < Right child
* **Traversal Methods**: Inorder, Preorder, Postorder

### 🔸 Operations in BST:

* Search, Insert, Delete: O(log n) (if balanced)

### 🔸 Real-World Use:

* File systems, databases (e.g., B-Trees), XML parsers.

## 🔹 6. ****Graph****

### 🔸 Definition:

A graph is a collection of **nodes (vertices)** connected by **edges**. Can be **directed** or **undirected**.

### 🔸 Key Concepts:

* Represented using:
  + **Adjacency List**
  + **Adjacency Matrix**
* Types: **Weighted/Unweighted**, **Cyclic/Acyclic**

### 🔸 Operations:

* DFS (Depth-First Search), BFS (Breadth-First Search)

### 🔸 Real-World Use:

* Social networks, Google Maps, networks, recommendation systems.

Great questions! Let's break each of these topics down **simply and clearly** so you can understand them easily for interview purposes.

## 🔹 1. ****Contiguous vs Uncontiguous Memory****

| **Contiguous Memory** | **Uncontiguous Memory** |
| --- | --- |
| Memory is allocated in **one continuous block**. | Memory is allocated in **different locations** (not next to each other). |
| Example: **Arrays** | Example: **Linked Lists** |
| Faster access using index. | Access via pointers, so slightly slower. |

### 🧠 Simple Example:

* Imagine books placed on **one shelf side by side** = contiguous.
* Books scattered on **different shelves** but linked with a note saying "next book is on shelf 3" = uncontiguous.

## 🔹 2. ****Types of Linked Lists****

### 🔸 a) Singly Linked List:

* Each node points to the **next** node only.
* Last node points to NULL.

1 -> 2 -> 3 -> NULL

### 🔸 b) Doubly Linked List:

* Each node points to the **next** and **previous** node.
* Allows two-way traversal.

NULL <- 1 <-> 2 <-> 3 -> NULL

### 🔸 c) Circular Linked List:

* The **last node points back** to the first node.
* Can be singly or doubly circular.

1 -> 2 -> 3 --+

^ |

+------------+

## 🔹 3. ****Types of Queues****

### 🔸 a) Simple Queue (or Linear Queue):

* Follows **FIFO** – first in, first out.
* Insert at rear, delete from front.

Front -> [1, 2, 3] <- Rear

### 🔸 b) Circular Queue:

* The last position connects to the first to make a **circle**.
* Efficient use of memory.

After rear, it wraps around to the front (if free).

### 🔸 c) Priority Queue:

* Elements are dequeued based on **priority**, not just order.
* Highest (or lowest) priority is removed first.

Tasks with higher priority get served first.

## 🔹 4. ****Tree Traversals (Inorder, Preorder, Postorder)****

### 📚 Example Tree:

1

/ \

2 3

### 🔸 a) ****Inorder (Left → Root → Right)****:

* Visit left child, then root, then right child.
* Output: 2 1 3

### 🔸 b) ****Preorder (Root → Left → Right)****:

* Visit root first, then left, then right.
* Output: 1 2 3

### 🔸 c) ****Postorder (Left → Right → Root)****:

* Visit left child, right child, then root.
* Output: 2 3 1

### ✅ Summary Table:

| **Traversal Type** | **Order** | **Example Output** |
| --- | --- | --- |
| Inorder | Left → Root → Right | 2 1 3 |
| Preorder | Root → Left → Right | 1 2 3 |
| Postorder | Left → Right → Root | 2 3 1 |

Examples:

* Searching
* Sorting
* Path finding
* Recursion
* Greedy methods
* Dynamic programming

## Why do we need DSA? (Purpose)

To **write code that is fast, smart, and memory-efficient**.

Without DSA:

* Code is slow
* Takes too much memory
* Doesn’t work well with large data

With DSA:

* Code runs faster
* Saves space
* Easily solves big, complex problems

## Where is DSA used today? (With Examples)

| **Area** | **How DSA is used** |
| --- | --- |
| 🔍 Google Search | Uses **graphs** and **search algorithms** to find best results |
| 📱 Social Media (Facebook, Instagram) | Uses **graphs**, **queues**, **hashmaps** for news feeds, comments, etc. |
| 🛒 E-Commerce (Amazon) | Uses **sorting**, **searching**, **dynamic programming** for filtering, suggestions |
| 🚗 Google Maps | Uses **graph algorithms** like Dijkstra to find shortest path |
| 🎮 Games | Uses **trees**, **backtracking**, and **AI algorithms** |
| 🧠 AI & Machine Learning | Uses **matrices**, **graphs**, and **greedy algorithms** |

## Simple Example:

### Problem:

You want to find a contact name in your phone with 5000 contacts.

### Without DSA:

Loop through all contacts — takes time.  
Slow if you have thousands of names.

### With DSA:

Use a **binary search** (algorithm) on a **sorted array** (data structure).  
You can find the name in just a few steps!

**Arrays**

## What is Bubble Sort?

**Bubble Sort** is the simplest way to **sort a list** by **repeatedly comparing two numbers and swapping them** if they are in the wrong order.

## How It Works?

### Step-by-step:

1. Start from the **first element**.
2. Compare it with the **next element**.
3. If the **first is bigger**, **swap** them.
4. Move to the next pair and repeat.
5. Keep doing this until the **biggest number moves to the last**.
6. Repeat the process for the remaining list (excluding the last one, which is already sorted).
7. Continue until the list is sorted.

## 📊 Example: Sort this list

**Original list:**  
[5, 2, 4, 1]

### Pass 1:

* Compare 5 and 2 → 5 > 2 → Swap → [2, 5, 4, 1]
* Compare 5 and 4 → 5 > 4 → Swap → [2, 4, 5, 1]
* Compare 5 and 1 → 5 > 1 → Swap → [2, 4, 1, 5] ✅

Biggest (5) is now at the end.

### Pass 2:

* Compare 2 and 4 → No swap
* Compare 4 and 1 → Swap → [2, 1, 4, 5]

### Pass 3:

* Compare 2 and 1 → Swap → [1, 2, 4, 5] ✅

Now the list is sorted!

## What is Selection Sort?

**Selection Sort** is a sorting method where you **select the smallest element** from the unsorted part of the list and **place it at the correct position** (beginning), one by one.

## 🔍 How It Works (Step-by-step):

1. Start with the full list.
2. Find the **smallest element**.
3. Swap it with the **first element**.
4. Then look at the remaining list (excluding the first) and **repeat**:
   * Find the next smallest.
   * Swap it with the second element.
5. Keep doing this until the list is sorted.

## 📊 Example: Sort this list

**Original list:**  
[5, 2, 4, 1]

### Pass 1:

* Smallest = 1
* Swap 1 with 5 → [1, 2, 4, 5]

### Pass 2:

* From [2, 4, 5], smallest = 2 → Already in place → No swap

### Pass 3:

* From [4, 5], smallest = 4 → Already in place → No swap

✅ List is now sorted.

## What is Insertion Sort?

**Insertion Sort** is a simple method to sort elements by taking **one element at a time** and **inserting** it into its **correct position** in the sorted part of the list.

## 🧠 How It Works (Step-by-Step):

Let's say we want to sort this list:  
[5, 3, 4, 1]

We start from index 1 (second element), and:

1. **Pick 3**  
   Compare it with 5 → 3 < 5 → Move 5 → Insert 3 before 5  
   → [3, 5, 4, 1]
2. **Pick 4**  
   Compare with 5 → 4 < 5 → Move 5 → Compare with 3 → 4 > 3 → Insert 4 after 3  
   → [3, 4, 5, 1]
3. **Pick 1**  
   Compare with 5, 4, 3 → all are greater → Move all → Insert 1 at start  
   → [1, 3, 4, 5] ✅ Sorted!

## 🧪 Example Walkthrough:

### Original Array:

[5, 3, 4, 1]

| **Step** | **Key** | **Result** |
| --- | --- | --- |
| 1 | 3 | [3, 5, 4, 1] |
| 2 | 4 | [3, 4, 5, 1] |
| 3 | 1 | [1, 3, 4, 5] ✅ |

## What is Quick Sort?

**Quick Sort** is a **fast** and efficient sorting algorithm that works by **dividing** the array into smaller parts, sorting them, and combining them.

It uses a technique called **Divide and Conquer**.

## 🎯 Real-Life Analogy:

Imagine you’re organizing books on a shelf by height:

1. Pick one book as a **pivot**.
2. Move **shorter books to the left**, **taller books to the right**.
3. Repeat the same process for the left and right parts.

## 🧠 How Quick Sort Works (Step-by-Step):

Let’s sort:  
[5, 3, 8, 4, 2]

### Step 1: Choose a pivot

Let’s choose the **last element** → pivot = 2

### Step 2: Partition

Move numbers **< 2** to the **left**, **> 2** to the **right** →  
Result after partition: [2, 3, 8, 4, 5] (pivot 2 is in correct place)

Now:

* Left part: [] (empty)
* Right part: [3, 8, 4, 5]  
  → Apply the same process **recursively** to both parts.

### Step 3: Continue recursively

Next for [3, 8, 4, 5]  
Pivot = 5  
Partition: [3, 4] 5 [8]

Now we have:

* [2] [3, 4] [5] [8]

Finally, sorted list:  
✅ [2, 3, 4, 5, 8]

## What is Merge Sort?

**Merge Sort** is a **Divide and Conquer** algorithm that divides the list into smaller parts, sorts each part, and then **merges** them back into a single sorted list.

[6, 3, 8, 2] → [6, 3] and [8, 2]

[6, 3] → [6] and [3]

[8, 2] → [8] and [2]

### Step 2: Merge the sublists (in sorted order)

Now merge while sorting:

* Merge [6] and [3] → [3, 6]
* Merge [8] and [2] → [2, 8]
* Merge [3, 6] and [2, 8] → Final: [2, 3, 6, 8]

✅ Sorted!

## What is Linear Search?

**Linear Search** is a basic method of searching an element in a list or array by **checking each element one by one** from start to end.

## How It Works (Step-by-Step)

Let's say we have this array:  
[10, 25, 30, 45, 60]

And we want to search for 30.

### Steps:

1. Check index 0 → is 10 = 30? ❌
2. Check index 1 → is 25 = 30? ❌
3. Check index 2 → is 30 = 30? ✅ Found at index 2

If the number was not in the list, it would check all elements and return "not found".

array = [10, 25, 30, 45, 60]

target = 30

→ Start from index 0:

10 ❌

25 ❌

30 ✅ → Found at index 2

## What is Binary Search?

**Binary Search** is an efficient method to find a value in a **sorted array** by **repeatedly dividing the search range in half**.

**Real-Life Example:**

Imagine a dictionary 📖 and you're looking for the word **"Mango"**:

* You open the dictionary **in the middle**.
* If you land on **"Pineapple"**, you know "Mango" must be **before**.
* If you land on **"Apple"**, you know it's **after**.
* You keep dividing the remaining pages until you find the word.

## Important Rule:

**Binary Search only works on sorted arrays** (e.g., [2, 4, 6, 8, 10, 12]).

## 🧠 How It Works (Step-by-Step)

Let's say we have this sorted array:  
[10, 20, 30, 40, 50, 60, 70]  
We want to search for: 40

### Steps:

1. **Start with the entire array**  
   low = 0, high = 6 (index range)
2. **Find the middle**:  
   mid = (0 + 6) / 2 = 3 → array[3] = 40
3. Is 40 == 40? ✅ YES → Found at index 3

If it wasn’t equal:

* If the target was **less than** the middle → search left half.
* If the target was **greater than** the middle → search right half.

**Linked List**

Great question! Let’s understand **Linked List** in **simple terms** and cover **all related topics** step by step in one clear explanation.

## 🧩 What is a Linked List?

A **linked list** is a data structure used to store a **collection of elements** (called nodes).  
Unlike arrays, elements are not stored in **continuous memory**.

### 📌 Each node contains:

1. **Data** – the value (like a number or name)
2. **Pointer (or link)** – that points to the **next node** in the list

## 🔗 Visual Example:

[10 | next] → [20 | next] → [30 | NULL]

Each box is a **node**, and the next points to the next node in the list.

## ❓ Why Use a Linked List?

* Arrays have **fixed size**; linked lists can **grow or shrink easily**.
* Inserting or deleting elements in arrays is slow; linked lists make it easier.
* Useful in dynamic memory situations.

## 🧠 Types of Linked Lists

### 1. ****Singly Linked List****

* Each node points to the **next** node.
* Last node points to **NULL**.

[1 | next] → [2 | next] → [3 | NULL]

### 2. ****Doubly Linked List****

* Each node has two pointers: **prev** and **next**
* Can move in **both directions**.

NULL ← [1] ↔ [2] ↔ [3] → NULL

### 3. ****Circular Linked List****

* The **last node** points back to the **first node**.

[1] → [2] → [3] → [1] (back to start)

## 🔧 Basic Operations

### ✅ 1. ****Traversal (Display)****

* Move from one node to the next until NULL is reached.

### ➕ 2. ****Insertion****

* At beginning
* At end
* At a specific position

### ➖ 3. ****Deletion****

* Delete first node
* Delete last node
* Delete a specific node (by value or position)

### 🔍 4. ****Search****

* Traverse the list and compare each node's value.

## 🧱 Structure of a Node in C++

struct Node {

int data; // data part

Node\* next; // pointer to next node

};

## 🔁 Example: Creating and Printing a Singly Linked List

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* next;

};

int main() {

// Creating nodes

Node\* head = new Node();

Node\* second = new Node();

Node\* third = new Node();

// Assign data

head->data = 1;

head->next = second;

second->data = 2;

second->next = third;

third->data = 3;

third->next = NULL;

// Print the list

Node\* temp = head;

while (temp != NULL) {

cout << temp->data << " ";

temp = temp->next;

}

return 0;

}

**Output**:

1 2 3

## 🧾 Summary Table

| **Concept** | **Meaning** |
| --- | --- |
| 🔗 Linked List | List of nodes connected by pointers |
| 📘 Node Structure | data + pointer |
| 🔄 Types | Singly, Doubly, Circular |
| ⚙️ Operations | Insert, Delete, Traverse, Search |
| 💡 Benefit | Dynamic size, easy insert/delete |
| ❗ Limitation | No direct access like arrays (arr[i]) |

**Stack/Queue**

## 📚 What is a ****Stack****?

A **stack** is a data structure that follows the rule:

### 🧠 LIFO – Last In, First Out

This means **the last element added** is the **first one to come out**.

### 🎯 Real-Life Example of Stack:

* A **stack of plates**: You can only take the **top plate**.
* Pushing = placing a plate
* Popping = removing the top plate

### 🔧 Stack Basic Operations

| **Operation** | **Meaning** |
| --- | --- |
| push() | Add an item on top |
| pop() | Remove the top item |
| peek() or top() | Look at the top item |
| isEmpty() | Check if the stack is empty |

### 📦 How Stack Works in Memory

Top → [30]

[20]

[10]

Bottom

### 🔁 Stack in C++ (Using STL)

#include <iostream>

#include <stack>

using namespace std;

int main() {

stack<int> s;

s.push(10);

s.push(20);

s.push(30);

cout << "Top: " << s.top() << endl; // 30

s.pop();

cout << "Top after pop: " << s.top() << endl; // 20

}

## 📚 What is a ****Queue****?

A **queue** is a data structure that follows the rule:

### 🧠 FIFO – First In, First Out

This means the **first element added** is the **first one to be removed**.

### 🎯 Real-Life Example of Queue:

* A **line at a ticket counter**: First person in line gets served first.

### 🔧 Queue Basic Operations

| **Operation** | **Meaning** |
| --- | --- |
| enqueue() | Add an item to the **rear** |
| dequeue() | Remove an item from the **front** |
| front() | See the front item |
| isEmpty() | Check if the queue is empty |

### 📦 How Queue Works in Memory

Front → [10] [20] [30] ← Rear

### 🔁 Queue in C++ (Using STL)

#include <iostream>

#include <queue>

using namespace std;

int main() {

queue<int> q;

q.push(10); // enqueue

q.push(20);

q.push(30);

cout << "Front: " << q.front() << endl; // 10

q.pop(); // dequeue

cout << "Front after pop: " << q.front() << endl; // 20

}

## 🔄 Difference Between Stack and Queue

| **Feature** | **Stack** | **Queue** |
| --- | --- | --- |
| Order | LIFO (Last In First Out) | FIFO (First In First Out) |
| Insert at | Top | Rear |
| Remove from | Top | Front |
| Real-life example | Stack of plates | Waiting line at a bank |

## 📌 Where Are They Used?

### ✅ Stack is used in:

* Undo operations (like Ctrl+Z)
* Function call stack
* Expression evaluation (e.g., postfix)

### ✅ Queue is used in:

* Printer queues
* CPU task scheduling
* Order processing

**Trees**

## 🌳 What is a Tree (in Data Structures)?

A **Tree** is a **non-linear data structure** that represents data in a **hierarchical (parent-child) format**.

Unlike arrays, linked lists, stacks, or queues (which are linear), **trees branch out**.

### 📌 Real-Life Examples of Trees:

* A **family tree**
* A **folder** system on a computer
* **Company hierarchy** (CEO → Manager → Employees)

## 🔧 Basic Terminology

| **Term** | **Meaning** |
| --- | --- |
| **Node** | Each element in a tree |
| **Root** | The **top node** of the tree |
| **Child** | A node that comes from another node |
| **Parent** | The node that has child nodes |
| **Leaf** | A node that **has no children** |
| **Edge** | A connection between two nodes |
| **Subtree** | A smaller tree formed from a node and its children |
| **Height** | Number of levels (depth) from root to the lowest leaf |
| **Level** | Distance from the root (root is level 0, its children are level 1, etc) |

## 🌳 Structure of a Tree

1 ← Root

/ \

2 3 ← Level 1

/ \

4 5 ← Level 2

## ✅ Types of Trees

### 1. ****Binary Tree****

* Each node has **at most 2 children**

1

/ \

2 3

### 2. ****Binary Search Tree (BST)****

* Left child < Parent < Right child
* Easy for **searching and sorting**

### 3. ****Balanced Tree (e.g., AVL Tree)****

* Keeps height balanced to avoid long chains.

### 4. ****Complete Binary Tree****

* All levels are filled, and last level is filled **left to right**

### 5. ****Full Binary Tree****

* Every node has 0 or 2 children

## 🔁 Tree Traversal (Visiting Nodes)

Three common ways to traverse a tree:

| **Type** | **Order** |
| --- | --- |
| Inorder | Left → Root → Right |
| Preorder | Root → Left → Right |
| Postorder | Left → Right → Root |

### 🔁 Example (Inorder)

1

/ \

2 3

Inorder: 2 1 3

## 🌱 Tree Node in C++

struct Node {

int data;

Node\* left;

Node\* right;

};

## 🌲 Simple C++ Example of a Tree

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* left;

Node\* right;

};

Node\* createNode(int value) {

Node\* newNode = new Node();

newNode->data = value;

newNode->left = newNode->right = NULL;

return newNode;

}

int main() {

Node\* root = createNode(1);

root->left = createNode(2);

root->right = createNode(3);

cout << "Root: " << root->data << endl;

cout << "Left Child: " << root->left->data << endl;

cout << "Right Child: " << root->right->data << endl;

return 0;

}

## 💡 Where Trees Are Used?

* **Databases** (like B-Trees for indexing)
* **File systems**
* **AI** (decision trees)
* **Compilers** (syntax trees)
* **Search engines**
* **Routers and networks**

## 📌 Summary

| **Concept** | **Explanation** |
| --- | --- |
| Tree | Hierarchical data structure |
| Node | Element in a tree |
| Root | First/top node |
| Binary Tree | Max 2 children per node |
| Traversal | Visiting nodes (Inorder, Pre, Post) |
| Used In | Databases, file systems, AI, etc. |

**Graphs**

## 🌐 What is a Graph?

A **Graph** is a **non-linear data structure** that consists of a set of **nodes (vertices)** connected by **edges**.

In simple terms, a graph is used to represent relationships between **pairs of objects**. These objects can be anything, like people, cities, computers, etc., and the connections between them are represented by **edges**.

### 📌 Real-Life Examples of Graphs:

* **Social Network**: Each person is a node, and friendships are edges.
* **City Map**: Cities are nodes, and roads are edges connecting them.
* **Web Pages**: Pages are nodes, and links between them are edges.

## 🔧 Key Terminologies in Graphs

| **Term** | **Meaning** |
| --- | --- |
| **Vertex (Node)** | A single point or object in the graph (e.g., a person, city) |
| **Edge** | A connection between two vertices (e.g., a friendship, road) |
| **Degree of a vertex** | The number of edges connected to a vertex |
| **Path** | A sequence of vertices connected by edges |
| **Cycle** | A path that starts and ends at the same vertex |
| **Connected Graph** | A graph where there is a path between any two vertices |
| **Disconnected Graph** | A graph with at least two vertices without a direct path |

## 🌍 Types of Graphs

### 1. ****Directed Graph (Digraph)****

* **Edges** have a direction (pointing from one vertex to another).
* **Example**: A one-way street between two cities.

A → B

### 2. ****Undirected Graph****

* **Edges** have no direction (the connection is mutual).
* **Example**: A friendship between two people.

A — B

### 3. ****Weighted Graph****

* **Edges** have weights (values) associated with them, representing costs or distances.
* **Example**: Road distances between cities.

A —(5)— B

### 4. ****Unweighted Graph****

* **Edges** do not have weights; it's just a connection between two vertices.

## 🔁 Graph Representations

### 1. ****Adjacency Matrix****

A 2D array is used where matrix[i][j] is 1 if there is an edge from vertex i to vertex j, else 0.

Example for undirected graph:

A — B

| |

C — D

Adjacency Matrix:

A B C D

A [0, 1, 1, 0]

B [1, 0, 0, 1]

C [1, 0, 0, 1]

D [0, 1, 1, 0]

### 2. ****Adjacency List****

Each vertex stores a **list** of its **neighbors** (connected vertices).

Example for the same graph:

A — B

| |

C — D

Adjacency List:

A -> [B, C]

B -> [A, D]

C -> [A, D]

D -> [B, C]

## 📌 Graph Traversal Methods

### 1. ****Depth First Search (DFS)****

* **Explores as deep as possible** before backtracking.
* Uses a **stack** (can be recursive).
* **Order**: Start from the root, visit a vertex, then its adjacent vertices, and continue until no further adjacent vertices.

**Example**: If you start at vertex A, the order could be A → B → D → C.

### 2. ****Breadth First Search (BFS)****

* **Explores level by level**, visiting all neighbors first.
* Uses a **queue**.
* **Order**: Start from the root, visit all adjacent vertices, then move to the next level.

**Example**: If you start at vertex A, the order could be A → B → C → D.

## 📌 Where are Graphs Used?

Graphs are widely used in many real-world applications:

* **Social Networks** (to model relationships)
* **Navigation Systems** (finding shortest paths between cities)
* **Recommendation Systems** (like YouTube or Amazon recommendations)
* **Networking** (routing data packets)
* **Web Crawling** (search engines use graphs to find web pages)

## 🧾 Summary of Graph Concepts

| **Concept** | **Description** |
| --- | --- |
| **Graph** | A collection of vertices connected by edges |
| **Vertex** | A node in a graph |
| **Edge** | A connection between two vertices |
| **Directed Graph** | Edges have direction |
| **Undirected Graph** | Edges have no direction |
| **BFS** | Explores level by level using a queue |
| **DFS** | Explores deep first using a stack |
| **Representation** | Adjacency List and Adjacency Matrix |
| **Applications** | Social Networks, Web Crawling, Path Finding, etc. |

**Algorithms**

## 🔹 1. What is ****Dijkstra's Algorithm****?

### 📌 Purpose:

To find the **shortest path** from a starting point (called the **source**) to **all other nodes** in a **weighted graph** (with non-negative weights).

### 🧠 Why We Need It:

* To find the **fastest** or **least expensive** way to travel, send data, or move through a network.
* Used in **Google Maps**, **network routing**, etc.

### 🚗 Simple Example:

Imagine you’re in city A and want to find the shortest distance to other cities (B, C, D). Roads have different lengths (weights). Dijkstra helps find the **shortest route** from A to every other city.

### 📋 How it works (in simple steps):

1. Start at the source node (distance = 0).
2. Set all other nodes to distance = infinity.
3. Visit the **nearest unvisited node**.
4. Update the distance to its neighbors **if a shorter path is found**.
5. Repeat until all nodes are visited.

## 🔹 2. What is ****Bellman-Ford Algorithm****?

### 📌 Purpose:

Also used to find **shortest paths** from a source to all other nodes — but **works with negative weights** too.

### 🧠 Why We Need It:

* Dijkstra **does not** work with **negative edge weights**.
* Bellman-Ford can **detect negative weight cycles**, which Dijkstra can't.

### 🏁 Example:

Suppose you’re getting rewards or discounts on some roads (negative cost). Bellman-Ford finds the shortest path **even if some roads give you rewards**.

### 📋 How it works:

1. Start at the source.
2. Repeat (V - 1) times:
   * For every edge, check if the distance to the destination can be shortened.
3. In the end, check for **negative cycles**.

## 🧠 Summary: Dijkstra vs Bellman-Ford

| **Feature** | **Dijkstra** | **Bellman-Ford** |
| --- | --- | --- |
| Handles Negative Weights | ❌ No | ✅ Yes |
| Time Complexity | Faster (O(V²) or O(E log V)) | Slower (O(V × E)) |
| Detects Negative Cycle | ❌ No | ✅ Yes |
| Best for | Maps, Networks (no -ve weights) | Graphs with negative weights |

## 🌲 3. What is a ****Spanning Tree****?

### 📌 Definition:

A **spanning tree** is a **subgraph** of a graph that:

* Includes **all the vertices**
* Has **no cycles**
* Uses **minimum number of edges** to connect all nodes

For a graph with N nodes, a spanning tree has exactly N - 1 edges.

### 🧠 Why We Need It:

To **connect all points** with **minimum cost** and **no loops**.

### 📦 Real-Life Example:

Suppose you are laying internet cables between houses. You want to connect all houses (nodes) with **minimum wire** and **no loops** — that’s what a spanning tree does.

## 🛠️ Types of Spanning Tree Algorithms:

1. **Prim’s Algorithm** – builds the tree starting from one node.
2. **Kruskal’s Algorithm** – adds the shortest edge that doesn’t form a cycle.

### 🚀 Summary of All Three:

| **Concept** | **Purpose** | **Handles Negative Edges** | **Use Cases** |
| --- | --- | --- | --- |
| **Dijkstra** | Find shortest path | ❌ No | GPS, routing (positive weights) |
| **Bellman-Ford** | Find shortest path + detects -ve cycle | ✅ Yes | Financial graphs, risky paths |
| **Spanning Tree** | Connect all nodes with minimum edges | Not for shortest paths | Network design, wiring |