### **✅ What is Data Structure? A way to organize and store data so it can be used efficiently. 📦 Example: Array, Stack, Queue, Linked List, Tree, Graph**

### **✅ What is Algorithm? A step-by-step set of instructions to solve a problem. ⚙️ Example: Sorting numbers, searching a name, finding the shortest path**

### **🔗 Why are they important together? Data Structures = *how* data is stored Algorithms = *how* data is processed**

### **Together, they help build fast and efficient programs.**

### **✅ What is Data Structure? A way to organize and store data so it can be used efficiently. 📦 Example: Array, Stack, Queue, Linked List, Tree, Graph**

### **✅ Real-Life Example:**

### **Imagine you’re managing a library:**

### **You can use an array to store books in order.**

### **Use binary search to quickly find a book.**

### **Use a queue to manage people waiting in line.**

### 

### **Why Do We Use Data Structures and Algorithms (DSA)?**

### 

### 

### 

### 

### **✅ 1. To Solve Problems Efficiently**

### **DSA helps you write code that runs faster and uses less memory**

### **✅ 2. To Handle Large Data**

### **DSA helps you organize and process it easily.**

### **Example: Searching for a name in a contact list of 10 people vs 1 million people.**

### 

### **✅ 3. To Build Better Software**

### **Apps like WhatsApp, Google Maps, Amazon, etc. all use DSA**

### **what are the common type of data structures ?**

### **1. Array**

* A collection of elements stored in **a fixed-size** order.
* Each element is accessed by its **index**.
* 📦 Like boxes in a row: A[0], A[1], A[2]...

**Example:** [10, 20, 30, 40]

### 

### **2. Linked List**

* A chain of elements called **nodes**.
* Each node has **data** and a **link** to the next node.
* Good for inserting/removing elements easily.

**Types:** Singly, Doubly, Circular Linked List

### **3. Stack**

* Follows **LIFO** (Last In, First Out).
* Think of a **stack of plates** — you take the top one first.

**Main operations:**

* push() – add item
* pop() – remove item

### **4. Queue**

* Follows **FIFO** (First In, First Out).
* Like a **line at a ticket counter** – first come, first served.

**Main operations:**

* enqueue() – add item
* dequeue() – remove item

### **5. Hash Table (HashMap)**

* Stores data in **key-value pairs**.
* Very fast for searching.

**Example:** "name" → "Sharik"  
 "age" → 22

### **7. Graph**

* A set of **nodes (vertices)** connected by **edges**.
* Used in social networks, maps, etc.

### **8. Heap**

A **Heap** is a special kind of **Tree** data structure used mainly for quick **access to the largest or smallest element**.

It's used in **priority queues** (like a task manager where urgent tasks come first).

### **✅ Real-World Applications of Data Structures:**

1. **Stack** – Used in **Undo/Redo** features (MS Word, Photoshop)
2. **Queue** – Used in **Job Scheduling** (OS, printers)
3. **Tree** – Used in **DOM** (HTML structure in browsers)
4. **Graph** – Used in **Social Networks** (friends suggestion)
5. **Hash Table** – Used in **Databases, Caching, Indexing**
6. **Linked List** – Used in **Music playlist, Browser history**

**What is Built-in DSA?**

**Array , object , string , set , map**

What is custom dsa ?

**Stack**, **Queue**, or **Linked List**. hash , tree , graph

### **✅ Advantages of DSA:**

* **Efficient data management**
* Helps solve complex problems quickly
* **Optimized for performance**
* **Improves coding skills**

### **❌ Disadvantages of DSA:**

* Takes time to learn and implement
* **Memory-consuming** for large data
* Can be **overcomplicated** for small problems

### **✅ Linear DSA:**

* **Data is stored in a sequence** (one after the other).
* Examples: **Array**, **Stack**, **Queue**.

### **❌ Non-linear DSA:**

* **Data is stored in a hierarchy or network**.
* Examples: **Tree**, **Graph**.

**Linear** = one line (like a list)

**Non-linear** = branching or connected (like a tree or web)

### **✅ What is Algorithm? A step-by-step set of instructions to solve a problem. ⚙️ Example: Sorting numbers, searching a name, finding the shortest path**

### **You can think of it like a recipe for cooking:**

### **It tells you exactly what to do, in what order, to get the final result**

### **🔍 Example (Real-life):**

**Task:** Make a cup of tea  
 **Algorithm:**

1. Boil water
2. Add tea leaves
3. Add sugar and milk
4. Let it boil
5. Strain and serve

### **💻 Example (In programming):**

**Task:** Add two numbers  
 **Algorithm:**

1. Take two inputs
2. Add them
3. Show the result

**❓ Why Do We Use Algorithms?**

✅ To solve problems in a **step-by-step way** ✅ To make programs **fast, efficient, and correct** ✅ To handle **large amounts of data** quickly  
 ✅ To make code **easy to understand and reuse**

### **COMMON TYPE ?**

### **✅ 1. Searching Algorithms**

Used to **find an element** in a list or array.

* **Linear Search**: Checks each element one by one.  
   🔎 *Slow but simple.*
* **Binary Search**: Divides the list in half each time (only works on **sorted** lists).  
   ⚡ *Faster than linear search.*

### **✅ 2. Sorting Algorithms**

Used to **arrange data** in a particular order (ascending or descending).

* **Bubble Sort**: Repeatedly swaps adjacent elements if they are in the wrong order.
* **Selection Sort**: Selects the smallest (or largest) element and puts it in the correct place.
* **Insertion Sort**: Builds the sorted list one item at a time.
* **Merge Sort**: Divides the list, sorts each half, and merges them.
* **Quick Sort**: Picks a "pivot" and sorts elements around it.

### **✅ 3. Recursion Algorithms**

An algorithm that **calls itself** to solve smaller parts of a problem.  
 Example: Factorial, Fibonacci Series.

### **✅ 4. Divide and Conquer**

Divides the problem into smaller sub-problems, solves them, and combines the results.  
 Used in: **Merge Sort**, **Quick Sort**, **Binary Search**

### **✅ 5. Greedy Algorithms**

Makes the **best choice at each step**, hoping to find the global best solution.  
 Used in: Coin Change, Huffman Coding

### **✅ 6. Dynamic Programming (DP)**

Stores results of **sub-problems** to avoid solving the same thing again.  
 Used in: Fibonacci Series, Knapsack Problem

### **✅ 7. Backtracking**

Tries all possibilities and **goes back** when a solution path doesn’t work.  
 Used in: Maze problems, Sudoku solver, N-Queens problem

### **✅ 8. Graph Algorithms**

Used to work with **networks, maps, or connected data**.

* **Breadth-First Search (BFS)**
* **Depth-First Search (DFS)**
* **Dijkstra’s Algorithm** (Shortest path)
* **Kruskal’s and Prim’s Algorithm** (Minimum spanning tree)

**WHAT ARE THE KEY FACTORS to consider in algorithm ?**

### **1. Correctness 2. Time Complexity**

### **3. Space Complexity**

### **4. Simplicity & Readability**

### **✅ Real-World Applications of Algorithms:**

1. **Search Algorithms** – Used in **Search Engines** (Google, Bing)
2. *Path Finding (Dijkstra, A)*\* – Used in **Navigation Systems** (Google Maps)
3. **Sorting Algorithms** – Used in **E-commerce Filters** (price low to high)
4. **Encryption Algorithms** – Used in **Security** (Online payments, OTPs)

### **✅ Advantages of Algorithms:**

* **Faster problem-solving**
* Can handle large datasets efficiently
* **Optimizes tasks** (like sorting, searching)
* Helps in **real-world applications** (search engines, security)

### **❌ Disadvantages of Algorithms:**

* May be **complex to understand** at first
* Some algorithms may not work well with **huge data sets**
* Can be **time-consuming** to develop for specific problems

### 

### **🧠 What is Memory (in computers)?**

### **Memory is where the computer stores data and instructions temporarily or permanently.**

### 

### **🗂️ Two Main Types:**

1. **RAM (Random Access Memory)** –  
    Temporary memory used **while programs run**.  
    ➕ Fast, ➖ Data is lost when power is off.
2. **ROM (Read Only Memory)** –  
    Permanent memory, used to store **startup instructions**

**✅ Why is Memory Important?**

Without memory, the computer can’t **store data** or **run programs**.

### **✅ What is Memory Allocation?**

* **Explanation**: Memory allocation is when the **computer sets aside space** for storing data while the program runs.

**Example**:

**Static memory allocation**: In C, when you define an array with a fixed size:  
  
  
int arr[5]; // Memory for 5 integers is allocated statically.

**Dynamic memory allocation**: In JavaScript, creating an array whose size can change:  
  
let arr = [1, 2, 3]; // Memory is dynamically allocated as needed.

arr.push(4); // Array can grow dynamically.

### **✅ What is Memory Leak?**

* **Explanation**: A memory leak occurs when the **program uses memory but doesn’t release it** after it's no longer needed.

**Example**:

**Memory Leak in JavaScript**:  
  
  
function memoryLeak() {

let obj = { name: "Leaky" };

// Forget to delete or null the object when done

}

memoryLeak();

// 'obj' stays in memory even though it's no longer needed.

### **✅ Common Causes of Memory Leak:**

**Forgetting to release memory**:  
  
  
let obj = { name: "Leak" }; // obj is never cleared from memory

**Circular references** (when two objects reference each other):

let obj1 = {};

let obj2 = { obj1: obj1 };

obj1.obj2 = obj2;

// These objects are never cleaned by garbage collector.

**Detached DOM nodes**:  
  
  
const element = document.createElement('div');

document.body.appendChild(element);

element = null; // Still in memory if event listeners are not removed

### **✅ Symptoms of Memory Leak:**

1. **Increasing memory usage**:  
    As memory keeps increasing without being freed, your app’s RAM usage climbs, making the system slower.
2. **Performance issues**:  
    The program slows down due to using up too much memory.
3. **Crashes**:  
    The program might **crash** or **freeze** due to excessive memory usage.
4. **System slowdowns**:  
    Other applications on your system might also slow down as available memory decreases.

### **✅ Stack vs Heap:**

1. **Stack**:  
   * **Memory is used for function calls** and local variables.
   * Automatically freed when a function ends.

**2 Heap**:

* **Memory is used for dynamic objects** like arrays or objects that need resizing.
* Requires garbage collection in some languages (like JavaScript).

### **✅ Static Memory Allocation vs Dynamic Memory Allocation:**

1. **Static Memory Allocation**:  
   * Memory size is fixed at compile-time.

int arr[5]; // Static allocation of 5 integers

**2 Dynamic Memory Allocation**:

* Memory size is determined at runtime and can change.

let arr = []; // Dynamically allocated

arr.push(1); // Can grow based on input

### **✅ Contiguous Memory Allocation**

* **What it means**: Memory is allocated in a **single block**, all in a row.
* **Example**: Arrays, where all elements are stored next to each other.
* **Advantage**: **Faster** to access because it’s one block.
* **Disadvantage**: Needs a big enough chunk of free memory, or it can’t fit.

### **✅ Non-Contiguous Memory Allocation**

* **What it means**: Memory is allocated in **separate pieces**, not in one block.
* **Example**: Linked lists, where each item points to the next.
* **Advantage**: **Flexible**, doesn’t need a big block of memory.
* **Disadvantage**: **Slower** to access because you need to follow the links.

**Contiguous**: One big block of memory, fast but less flexible.

**Non-Contiguous**: Memory spread out, flexible but slower.

### **✅ Memory Allocation in JavaScript: Types of Memory Allocation**

* **JavaScript** uses **two types of memory allocation**:  
  1. **Stack**: For simple data types like numbers, strings, and booleans, and function calls.
  2. **Heap**: For complex data like objects, arrays, and functions.

### **✅ JavaScript: Stack vs Heap**

1. **Stack**:  
   * **Stores simple variables** (like integers or references to objects).
   * **Automatically freed** when the function ends.
   * **Faster** because it’s organized in a LIFO (Last In, First Out) manner.

**2 Heap**:

* **Stores objects, arrays, functions**.
* **Requires garbage collection** to clean up unused memory.
* **Slower** compared to the stack, but more flexible.

### **✅ What is Garbage Collection?**

* **Garbage Collection (GC)** is the **automatic process** by which JavaScript manages memory.
* The **engine** checks for objects that are no longer being used and **removes** them from memory to free up space.

### **✅ What is Mark-and-Sweep?**

* **Mark-and-Sweep** is the most common **garbage collection algorithm** in JavaScript.  
  1. **Mark phase**: The GC **marks** all objects that are reachable from the root (global object, local variables, etc.).
  2. **Sweep phase**: It **frees memory** by removing unmarked objects (those that are no longer accessible).

### **How to Debug Memory Leaks in JavaScript?**

1. **Use the browser’s developer tools** (like Chrome DevTools).  
   * Open **DevTools > Memory** tab to track memory usage over time.
   * Use **Heap Snapshot** or **Allocation instrumentation on timeline** to detect objects not being cleaned up.
2. **Look for Detached DOM nodes**:  
   * Unused DOM nodes that are not removed (e.g., elements still holding references).
   * **Check event listeners**: Sometimes event listeners attached to DOM elements aren't removed, leading to memory leaks.

### 

### **What is Complexity Analysis?**

Complexity analysis is a method used in computer science to measure the efficiency of an algorithm in terms of time and space. It helps us understand how an algorithm performs as the input size increases.

### **Why is Complexity Analysis Important?**

1. **Performance Comparison** – It allows us to compare different algorithms to choose the most efficient one.
2. **Optimization** – Helps in improving an algorithm to reduce execution time or memory usage.
3. **Scalability** – Ensures that an algorithm can handle large inputs without significant slowdowns.

### **How Does Complexity Analysis Work?**

Complexity analysis is divided into two main types:

1. **Time Complexity** – Measures how the execution time of an algorithm changes with input size (**n**).  
   * Example:  
     + A loop running **n** times → **O(n)** (linear complexity)
     + A nested loop running **n × n** times → **O(n²)** (quadratic complexity)
2. **Space Complexity** – Measures the amount of memory an algorithm uses as input size increases.  
   * Example:  
     + Using an extra array of size **n** → **O(n)** space complexity
     + Using only a few variables → **O(1)** (constant space complexity)

### **📌 What is Asymptotic Analysis?**

Asymptotic analysis is a **method** to measure the efficiency of an algorithm as the **input size (n) grows very large**.

🔹 **Why do we use it?**

* Helps compare different algorithms based on **time** and **space**.
* Focuses on **big inputs** (ignoring small variations).
* Predicts how an algorithm behaves for **large-scale problems**.

🔹 **How does it work?**

* We express the **growth rate** of an algorithm in **mathematical notation**.
* We ignore **constant factors** and **lower-order terms**.

## **📌 What is Asymptotic Notation?**

Asymptotic notation describes an algorithm’s **time complexity** using mathematical symbols.

There are **three main types**:  
 1️⃣ **Big-O (O)** → Upper Bound (Worst Case)  
 2️⃣ **Omega (Ω)** → Lower Bound (Best Case)  
 3️⃣ **Theta (Θ)** → Tight Bound (Average Case)

### **What is Big-O Notation?**

**Big-O notation** is a mathematical way to describe how the runtime (execution time) or space (memory usage) of an algorithm **grows** as the input size (**n**) increases. It helps us compare different algorithms and understand their efficiency.

### **Why is Big-O Notation Important?**

1. **Predicts Performance** – Helps us understand how an algorithm behaves with large inputs.
2. **Ignores Hardware Differences** – Focuses on the algorithm's efficiency, not the system running it.
3. **Compares Algorithms** – Helps us choose the best algorithm for a given problem.

### **Common Big-O Notations (Time Complexity)**

| **Big-O Notation** | **Name** | **Example** | **Performance** |
| --- | --- | --- | --- |
| **O(1)** | Constant Time | Accessing an array element arr[i] | ✅ **Fastest** |
| **O(log n)** | Logarithmic Time | Binary Search | 🔥 **Very Fast** |
| **O(n)** | Linear Time | Looping through an array | ⚡ **Average** |
| **O(n log n)** | Linearithmic Time | Merge Sort, Quick Sort (Best/Average case) | 🔥 **Good for sorting** |
| **O(n²)**  **O(n3)cubic** | Quadratic Time | Nested loops (Bubble Sort, Selection Sort) | ❌ **Slow for large n** |
| **O(2ⁿ)** | Exponential Time | Recursion in Fibonacci | ⏳ **Very slow** |
| **O(n!)** | Factorial Time | Traveling Salesman Problem | 🚫 **Worst** |

### **📌 Summary Table: Worst-Case Scenarios**

| **Algorithm** | **Worst-Case Input** | **Big-O Complexity** |
| --- | --- | --- |
| **Linear Search** | **Target at the end / Not found** | **O(n)** |
| **Bubble Sort** | **Reverse sorted array** | **O(n²)** |
| **Binary Search** | **Target not found** | **O(log n)** |
| **Fibonacci Recursion** | **Large n** | **O(2ⁿ)** |
| **Traveling Salesman** | **All cities must be visited** | **O(n!)** |

### **1️⃣ O(log log n) - Double Logarithmic Time**

* **What it means: The number of operations increases really slowly, even slower than O(log n).**

### **2️⃣ O(√n) - Square Root Time**

* **What it means: The number of operations grows with the square root of the input size. So for very large inputs, it’s still manageable.**

### **3️⃣ O(n^k) - Polynomial Time**

* **What it means: The number of operations grows as a polynomial based on the size of the input. As n gets bigger, the operations increase really fast.**

## **🔢 What is an Array?**

**An array is a collection of elements stored in a single variable.  
 All elements are of the same data type (e.g., all numbers or all strings), and they are stored in continuous memory locations.**

## **📌 What is a String?**

**A string is a sequence of characters enclosed in quotes ("", '', or ````).**

**Primitive String: Faster and uses less memory. immuted**

**Object String: It’s like an object and more complex but not commonly used. muted**

### **✅ What is Recursion?**

### **Recursion is when a function calls itself to solve a smaller part of a problem.**

### **🧠 It keeps repeating until it reaches a base condition (stopping point).**

### 

### **🤔 Why Use Recursion?**

* **Makes code shorter and easier to understand, especially for problems that repeat or have subproblems.**

### **🌟 When to Use Recursion:**

* **Problems that can be broken into smaller subproblems  
   👉 Like: Factorial, Fibonacci, Tree, Graph traversal, Tower of Hanoi**

### **🚫 When to Avoid Recursion:**

* **When the problem is simple and can be solved using loops**
* **When memory usage is important – recursion uses more memory (call stack)**

### **Where Do We Use Recursion?**

| **🧠 Use Case** | **✅ Example** | **💬 Why Recursion Helps** |
| --- | --- | --- |
| **📁 File Systems** | **Opening folders inside folders** | **Navigates deep structures easily** |
| **🧮 Math Problems** | **Factorial, Fibonacci, GCD** | **Breaks big problems into smaller ones** |
| **🧩 Data Structures** | **Trees (Binary Tree, BST, etc.)** | **Visit nodes (DFS, traversals)** |
| **🔍 Searching Algorithms** | **Binary Search** | **Keep dividing the search space** |
| **🧠 Backtracking** | **Solving puzzles, mazes, Sudoku** | **Try all options step-by-step** |
| **🔁 Divide and Conquer** | **Merge Sort, Quick Sort** | **Split problem → Solve → Combine** |

### **✅ Pros (Advantages):**

1. **Clean and short code**
2. **Best for tree/graph structures**
3. **Solves divide-and-conquer problems easily**

### **❌ Cons (Disadvantages):**

1. **Uses more memory (because of call stack)**
2. **Can cause stack overflow if too many recursive calls**
3. **Slower than iteration in some cases**

### **📌 Key Characteristics of Recursion:**

1. **Base Case – tells when to stop**
2. **Recursive Case – when the function calls itself**
3. **Stack Usage – function calls are stored in memory until done**
4. **Smaller Inputs – each call solves a smaller version of the problem**

### **✅ 1. What is a Recurrence Relation?**

**A recurrence relation is a formula that describes how to break a problem into smaller parts to solve it recursively.**

**💬 In Simple Words:  
 It tells how a problem can be split into smaller pieces, like solving for n-1 to get the answer for n.**

**Example:  
 To calculate factorial(n), you need to solve for factorial(n-1), then multiply the result.**

**function factorial(n) {**

**if (n == 1) return 1;**

**return n \* factorial(n - 1);**

**}**

### 

### **✅ 2. Types of Recurrence Relations**

**Here are the main types:**

| **Type** | **Example** | **Meaning** |
| --- | --- | --- |
| **Linear** | **T(n) = T(n - 1) + c** | **Solves one smaller subproblem** |
| **Binary/Divide-and-Conquer** | **T(n) = 2T(n / 2) + c** | **Solves two smaller problems of half size** |
| **Multiple Recursive Calls** | **T(n) = T(n - 1) + T(n - 2) + c** | **Like Fibonacci (multiple calls)** |
| **Tail Recursion** | **return example(n - 1, result + n)** | **Recursive call is the last thing** |

### **✅ 3. What are variables/parts in recursion?**

**In recursion, these parts help break down the problem:**

1. **Function Arguments: The inputs the function works with (e.g., n, index).**
2. **Base Case: Tells the function when to stop to avoid endless recursion.**
3. **Recursive Call: The function calls itself with simpler inputs to reduce the problem size.**
4. **Return Type: The type of result the function gives back, such as a number or an array.**

### **✅ 4. Why is return type important in recursion?**

**The return type is important because each recursive step relies on the result of the previous step. Without a valid return value, the function will break or return an error.**

**Example:**

**return n + sum(n - 1)**

**If the function didn’t return anything, the result would be:**

**return n + undefined // ❌ Error!**

### **✅ 5. What is Tail Recursion?**

* **Tail Recursion happens when the recursive call is the last step of the function. This is special because no further work is done after the recursive call.**

**Example of Tail Recursion:**

**function tailSum(n, acc = 0) {**

**if (n === 0) return acc; // Base case**

**return tailSum(n - 1, acc + n); // Tail call (last thing)**

**}**

**Why is it useful?**

* **It’s memory-efficient because it reduces the amount of memory used (in some programming languages).**
* **Some languages optimize tail recursion to reuse memory and prevent stack overflow.**

### **✅ 6. Regular Recursion vs Tail Recursion**

**Regular Recursion: After the recursive call, the function still has more work to do.  
 Example:  
  
function sum(n) {**

**if (n === 0) return 0;**

**return n + sum(n - 1); // More work after the call**

**}**

**Tail Recursion: The function does nothing after the recursive call. It just returns the result right away.  
 Example:  
  
function tailSum(n, acc = 0) {**

**if (n === 0) return acc;**

**return tailSum(n - 1, acc + n); // No more work after the call**

**}**

### **✅ 7. Why is Tail Recursion Useful?**

* **Memory Efficient: Some languages optimize it to use less memory.**
* **Prevents Stack Overflow: Helps avoid crashes due to too many recursive calls.**
* **Optimized in Some Languages: Some languages (like Python, Scheme) optimize tail recursion automatically, but JavaScript does not currently optimize it.**

### **✅ What is Iteration (Iterative Approach)?**

**Iteration means repeating a set of instructions using loops like:**

* **for loop**
* **while loop**
* **do-while loop**

**✅ What is Divide and Conquer Strategy?**

**Divide and Conquer is a strategy where you:**

1. **Divide the problem into smaller sub-problems**
2. **Conquer (solve) each sub-problem recursively**
3. **Combine the results to get the final answer**

### **🔁 Example Algorithms Using Divide and Conquer:**

* **Merge Sort**
* **Quick Sort**
* **Binary Search**
* **Matrix Multiplication (Strassen's Algorithm)**

### **🤔 Why Use It?**

* **Solves complex problems faster**
* **Helps in breaking down big problems**
* **Improves performance in many cases**

### **💪 Brute Force Algorithm:**

* **Tries all possible solutions**
* **Simple, but can be very slow**

**🧠 Example:  
 To search for an item in a list, brute force checks every element one by one**

### **⚡ Optimized Algorithm:**

* **Uses a better, faster approach**
* **Solves the problem using smart logic**

**🧠 Example:  
 Using Binary Search instead of linear search in a sorted array  
 (Splits array in half each time = much faster)**

## **🧠 What is Virtual Memory?**

**Virtual Memory is like an illusion that makes your computer think it has more RAM than it actually does.**

**💬 In Simple Words:  
 It lets big programs run even if your computer doesn’t have enough physical RAM by using a part of your hard drive as extra memory.**

### **📌 Key Concepts of Virtual Memory (Simplified)**

### **1️⃣ Paging**

* **Memory is cut into small blocks called pages.**
* **If RAM is full, some pages are moved to the hard disk.**
* **💡 Think: Like moving book pages between your desk (RAM) and your bag (hard disk) when there's no space.**

### **2️⃣ Swap Space**

* **A part of your hard disk acts like extra RAM.**
* **It's slower, but it stops your programs from crashing when RAM is full.**

### **3️⃣ Address Translation**

* **Programs use virtual addresses.**
* **The OS changes them into real (physical) addresses in RAM.**
* **💡 Think: Like using nicknames, and the OS finds the real names.**

## **✅ Benefits of Virtual Memory:**

* **➕ Run Larger Programs: Even with low RAM.**
* **➕ Multitasking: Multiple programs can run at once smoothly.**
* **➕ Isolation: Each program thinks it has full memory (more secure).**
* **➕ Efficient Memory Use: Unused data stays on disk, freeing RAM.**

### **✅ Common Operations in Normal Array**

**🧰 These are basic things we do with an array:**

| **Operation** | **Meaning** |
| --- | --- |
| **Access** | **Get value at a specific index → arr[2]** |
| **Insert** | **Add a value at a position** |
| **Delete** | **Remove a value from a position** |
| **Update** | **Change a value → arr[1] = 10** |
| **Traverse** | **Go through all elements (loop)** |
| **Search** | **Find a value (like linear/binary search)** |

### **✅ What is a Dynamic Array?**

**A dynamic array is an array that can grow or shrink automatically during the program.  
 It gives extra space when needed.**

**🧠 In JavaScript:**

**let arr = []; // This is a dynamic array**

### **⚙️ How Resizing Works (Amortized Resizing)**

**When the array gets full, it creates a new bigger array and copies everything into it.**

**🔁 Example:**

1. **You have space for 4 items: [1, 2, 3, 4]**
2. **You try to add 5 → no space!**
3. **It creates a bigger array (usually double the size, like 8)**
4. **Copies [1, 2, 3, 4] → [1, 2, 3, 4, \_, \_, \_, \_]**
5. **Adds 5 → [1, 2, 3, 4, 5, \_, \_, \_]**

### **✅ 1. What is Amortized Analysis?**

**📌 What:  
 Amortized analysis is used to find the average time per operation, over a sequence of operations, even if some are slow.**

**📈 Instead of looking at the worst-case of a single operation, it spreads the cost across many operations.**

**🧠 Why:  
 It gives a more realistic performance when some operations are fast and a few are slow.**

### **✅ 2. What is Amortized Resizing?**

**📌 What:  
 When a dynamic array (like ArrayList) becomes full, it resizes — often doubling in size.**

**⚠️ The resizing takes time, but it doesn't happen every time.**

**✅ So over many insertions, the average time per insert is still O(1) — that's amortized constant time.**

### **✅ 3. What is Dynamic Programming (DP)?**

**📌 What:  
 Dynamic Programming is a method to solve complex problems by breaking them into smaller subproblems and storing their results to avoid recalculating.**

**🧠 Why:  
 It saves time by avoiding repeating the same work.**

**🔁 Common in problems with:**

* **Overlapping subproblems**
* **Optimal substructure**

**💡 Examples: Fibonacci, Knapsack, Longest Common Subsequence**

### **✅ 4. What is Memoization?**

**📌 What:  
 Memoization is a top-down technique in Dynamic Programming.  
 It means storing answers of subproblems in a table (cache) and reusing them instead of recomputing.**

**🧠 Example:**

**let memo = {};**

**function fib(n) {**

**if (n <= 1) return n;**

**if (memo[n]) return memo[n];**

**return memo[n] = fib(n - 1) + fib(n - 2);**

**}**

### **✅ 5. What is Bottom-Up Approach (in DP)?**

**📌 What:  
 It is the opposite of memoization.**

**We solve smaller subproblems first, store the answers, and build up to the final solution.**

**🔁 Uses loops instead of recursion.**

**🧠 Why:**

* **Uses less memory (no call stack)**
* **Faster than recursion in many cases**

### **✅ 6. What is Compacting? (Simple)**

**📌 What:  
 Compacting means rearranging memory to remove empty spaces (gaps) and make it cleaner.**

**💡 Why:  
 It helps free up space and makes memory more efficient.**

**🧠 Where used:  
 Used in garbage collection (like in Java or Python) to fix memory fragmentation.**

### **✅ What is a Linked List?**

**A linked list is a data structure where each element (node) has:**

1. **The data**
2. **A pointer (link) to the next node**

**🧠 It’s like a chain where each link knows where the next one is.**

### **🤔 Why Use Linked Lists?**

* **To easily insert or delete elements**
* **When you don’t know the size in advance**
* **Better than arrays when you want flexible memory use**

### **📂 Types of Linked Lists:**

| **Type** | **Description** |
| --- | --- |
| **Singly Linked List** | **Each node points to the next only** |
| **Doubly Linked List** | **Nodes point to next and previous nodes** |
| **Circular Linked List** | **Last node connects to the first node** |

### **✅ Pros (Advantages):**

1. **Dynamic size (no need to define size in advance)**
2. **Easy insert/delete from beginning/middle**
3. **Efficient memory usage (no wasted space like arrays)**

### **❌ Cons (Disadvantages):**

1. **Slower access (no direct indexing like arrays)**
2. **More memory (because of pointers)**
3. **Harder to debug or reverse**

### **🧠 Applications of Linked List:**

* **📄 Implementing stacks and queues**
* **↔️ Undo/Redo in editors (like MS Word)**
* **🔁 Navigation systems (back and forward in browsers)**
* **🧹 Memory management (Garbage collectors)**
* **🎮 Dynamic game objects (changing sizes)**

### 

### **✅ What is Unicode?**

**📌 Unicode is a global standard that represents almost all characters from all languages.**

* **Supports over 1 million symbols**
* **Covers English, Hindi, Chinese, emojis, math symbols, etc.**

**🧠 ASCII is actually a subset of Unicode.**

### **✅ What is ASCII?**

**📌 ASCII = American Standard Code for Information Interchange  
 👉 It gives a number to each English character.**

| **Character** | **ASCII Code** |
| --- | --- |
| **'A'** | **65** |
| **'a'** | **97** |
| **'0'** | **48** |

**🧠 Used for basic English text and communication between computers.  
 Only supports 128 characters (English letters, numbers, symbols).**

### **✅ What is UTF-8?**

**📌 UTF-8 = Unicode format that can store all characters from all languages.  
 👉 It uses 1 to 4 bytes per character.**

**✅ Supports English, Chinese, Arabic, emojis 😄, etc.  
 ✅ Most used encoding on the internet.  
 ✅ Backward compatible with ASCII.**

### **🆚 ASCII vs UTF-8**

| **Feature** | **ASCII** | **UTF-8** |
| --- | --- | --- |
| **Size** | **1 byte** | **1 to 4 bytes** |
| **Characters** | **128 only** | **Over 1 million** |
| **Language** | **English only** | **All languages + emojis** |
| **Speed** | **Fast & simple** | **Slightly heavier, more flexible** |
| **Use case** | **Old systems, simple files** | **Websites, apps, emojis** |

### **🎯 Final Tip:**

* **Use ASCII for simple English text.**
* **Use UTF-8 for everything else (websites, global content, emojis, etc.)**

**🔹 What is an Escape Sequence?**

**It’s a special way to add characters in a string that you can’t type directly or would confuse the code.**

**✅ You start it with a backslash \ and then add a letter or symbol.**

## **🧠 Think of it like a secret shortcut!**

| **Escape** | **What it does** | **Example** | **Output** |
| --- | --- | --- | --- |
| **\n** | **New line** | **"Hello\nSharik"** | **Hello**  **Sharik** |
| **\t** | **Tab space** | **"Name:\tSharik"** | **Name: Sharik** |
| **\'** | **Single quote** | **'It\'s ok'** | **It's ok** |
| **\"** | **Double quote** | **"She said: \"Hi\""** | **She said: "Hi"** |
| **\\** | **Backslash** | **"C:\\folder\\file"** | **C:\folder\file** |

### **🧩 Why we use it?**

* **To add special characters like new lines or tabs**
* **To fix errors when using quotes**

### **What is Sorting?**

**Sorting is the process of arranging a set of data in a specific order, usually in:**

* **Ascending order (e.g., 1, 2, 3, 4)**
* **Descending order (e.g., 4, 3, 2, 1)**

### **✅ Why is Sorting Important?**

**1️⃣ Faster Search  
 📦 Sorted data = You can use Binary Search (fast).  
 2️⃣ Better Performance  
 ⚙️ Sorting helps other algorithms (like heaps, trees) work faster and smoother.**

**3️⃣ Neat & Organized  
 📊 Makes reports, rankings, and schedules look clean and easy to use.**

### **📌 Example Use Cases for Sorting:**

* **Database: Sorting records by date, name, etc.**
* **E-commerce: Sorting products by price or rating.**
* **Search engines: Sorting search results by relevance or rank.**

### **✅ What is Bubble Sort?**

**Bubble Sort is a simple sorting algorithm that works by:**

**👉 Repeatedly comparing each pair of adjacent elements  
 👉 Swapping them if they are in the wrong order**

**🔁 This keeps going until the entire list is sorted.**

### **🧠 Why is it called Bubble Sort?**

**Because the largest elements "bubble up" to the end of the list after each round.**

### **🔄 How it works (in easy steps):**

**Example:  
 Array = [4, 2, 1, 3]**

* **1st pass:  
   Compare 4 & 2 → swap → [2, 4, 1, 3]  
   Compare 4 & 1 → swap → [2, 1, 4, 3]  
   Compare 4 & 3 → swap → [2, 1, 3, 4]**
* **Keep repeating until fully sorted.**

### **⏱️ Time and Space Complexity:**

| **Case** | **Time Complexity** |
| --- | --- |
| **Best Case** | **O(n) *(already sorted)*** |
| **Average Case** | **O(n²)** |
| **Worst Case** | **O(n²) *(reverse order)*** |

**🧠 Space Complexity: O(1) → uses no extra space**

**👍 Pros:**

* **Super easy to understand and implement**
* **No extra memory needed**
* **Good for small datasets**

**👎 Cons:**

* **Very slow for large datasets**
* **Not used in real-world systems**

**What is apllication ?**

**1️⃣ Educational Tool  
2️⃣ Small Datasets  
3️⃣ Nearly Sorted Data**

### **✅ What is Selection Sort?**

**Selection Sort is a sorting algorithm that:**

**👉 Finds the smallest element from the unsorted part  
 👉 Places it at the beginning of the array**

**This repeats for the next smallest, and so on.**

### **🔄 How it works (simple steps):**

**Example: [4, 3, 1, 2]**

* **Step 1: Find the smallest → 1, swap with first → [1, 3, 4, 2]**
* **Step 2: Find the next smallest → 2, swap → [1, 2, 4, 3]**
* **Step 3: Repeat until sorted → [1, 2, 3, 4]**

### **⏱️ Time and Space Complexity:**

| **Case** | **Time Complexity** |
| --- | --- |
| **Best Case** | **O(n²)** |
| **Average Case** | **O(n²)** |
| **Worst Case** | **O(n²)** |

**🧠 Space Complexity: O(1) → No extra space needed**

### **👍 Pros:**

* **Very simple to understand**
* **No extra space needed**
* **Works well when memory is limited**

### **👎 Cons:**

* **Slow for large data (always O(n²))**
* **Not stable (may change order of equal elements)**

### **📌 Applications of Selection Sort:**

**1️⃣ Simple Sorting for Small Data Sets**

**2️⃣ Memory-Constrained Environments**

**3️⃣ Teaching/Illustrative Purposes**

### **✅ What is Insertion Sort?**

**Insertion Sort is a sorting method where we build the final sorted list one item at a time.**

**👉 Think of sorting playing cards in your hand —  
 You pick one card at a time and insert it in the correct place.**

### **🔄 How It Works? (Simple Steps)**

**Example: [5, 3, 4, 1]**

* **Start with second element (3), compare it with 5 → insert before → [3, 5, 4, 1]**
* **Next, take 4 → insert between 3 and 5 → [3, 4, 5, 1]**
* **Take 1 → insert at front → [1, 3, 4, 5]**

**➡️ Keep inserting each element in the correct position like a key in the right place.**

### **⏱️ Time & Space Complexity**

| **Case** | **Time** |
| --- | --- |
| **Best (sorted)** | **O(n)** |
| **Average** | **O(n²)** |
| **Worst** | **O(n²)** |

**🧠 Space Complexity: O(1) (no extra space used)**

### **👍 Pros (Advantages):**

* **✅ Very simple and easy to understand**
* **✅ Works great for small or almost sorted data**
* **✅ No extra memory needed**

### **👎 Cons (Disadvantages):**

* **❌ Slow for large data**
* **❌ Not suitable for big systems or large inputs**

### **Key Applications of Insertion Sort:**

1. **Small Data Sets**
2. **Nearly Sorted Data**
3. **Online Sorting**
4. **Adaptive Algorithms**
5. **Memory-Constrained Systems**

### 

### **✅ What is Quick Sort?**

### **Quick Sort is a fast sorting algorithm that works by:**

### **👉 Choosing a pivot element 👉 Putting smaller elements on the left of the pivot, and bigger ones on the right 👉 Then it recursively sorts the left and right parts**

### 

### **🔄 How It Works (Simple Steps):**

### **Let’s say: [6, 3, 9, 5, 2]**

### **Pick pivot (e.g., 6)**

### **Move smaller numbers to the left: [3, 5, 2]**

### **Bigger numbers to the right: [9]**

### **Repeat the same for both sides**

### 

### **⏱️ Time & Space Complexity:**

| **Case** | **Time Complexity** |
| --- | --- |
| **Best Case** | **O(n log n)** |
| **Average Case** | **O(n log n)** |
| **Worst Case** | **O(n²) *(when pivot is badly chosen)*** |

### **🧠 Space Complexity: O(log n) – due to recursion stack**

### 

### **👍 Pros of Quick Sort:**

### **Super fast in practice**

### **Works well on large datasets**

### **Uses less memory than Merge Sort**

### 

### **👎 Cons of Quick Sort:**

### **Can be slow in worst case (like already sorted array)**

### **Uses recursion, so can cause stack overflow if not handled well**

### 

### **📌 Applications of Quick Sort:**

### **✅ Used in standard libraries (like Python, Java, C++) ✅ Great for large datasets ✅ Used in databases, file systems, and search engines ✅ Good when we need fast sorting and less memory usage**

### 

### **✅ What is Merge Sort?**

**Merge Sort is a Divide and Conquer sorting algorithm.**

**It works by:**

1. **Dividing the array into halves again and again**
2. **Sorting each half**
3. **Merging the sorted halves together**

**🧠 It keeps dividing until each part has only one element, then merges them in order.**

### **🧠 Easy Way to Remember:**

**"Split → Sort → Merge"  
 That’s the whole idea of Merge Sort!**

### **🔄 How it Works (Easy Example)**

**For array [4, 2, 1, 3]:**

* **Divide: [4, 2] and [1, 3]**
* **Divide again: [4] [2] and [1] [3]**
* **Merge and sort: [2, 4] and [1, 3]**
* **Final merge: [1, 2, 3, 4]**

### **⏱️ Time and Space Complexity**

| **Case** | **Time Complexity** |
| --- | --- |
| **Best Case** | **O(n log n)** |
| **Average Case** | **O(n log n)** |
| **Worst Case** | **O(n log n)** |

**📦 Space Complexity: O(n) → Because it uses extra space to merge arrays**

### **✅ Why Use Merge Sort?**

* **It’s very fast and consistent**
* **Good for large datasets**
* **Works well with linked lists too**

### **👍 Pros**

* **Always has O(n log n) time**
* **Stable sort (keeps equal elements in same order)**
* **Works well with large or external data**

### **👎 Cons**

* **Uses extra memory (O(n))**
* **Slightly slower for small arrays**

### 

### 

### **📌 Applications of Merge Sort**

**✅ Sorting large files or data from external sources  
 ✅ Linked lists (because no need for random access)  
 ✅ Used in databases and big data systems**

### **1. In-place vs Non In-place Sorting**

* **In-place:**
  + **Sorts without extra memory.**
  + **Example: Quick Sort.**
* **Non In-place:**
  + **Uses extra memory to sort.**
  + **Example: Merge Sort.**

### **2. Stable vs Unstable Sorting**

* **Stable:**
  + **Keeps the original order of equal elements.**
  + **Example: Merge Sort, Bubble Sort.**
* **Unstable:**
  + **May change the order of equal elements.**
  + **Example: Quick Sort, Selection Sort.**

### **3. Adaptive vs Non-Adaptive Sorting**

* **Adaptive:**
  + **Works faster if the data is partially sorted.**
  + **Example: Insertion Sort, Bubble Sort.**
* **Non-Adaptive:**
  + **Takes the same time, even if the data is nearly sorted.**
  + **Example: Selection Sort, Merge Sort.**

### **✅ 4. Hybrid Sorting Algorithms**

**💡 Hybrid algorithms combine two or more sorting algorithms to get the best performance.**

| **Name** | **What It Combines** |
| --- | --- |
| **Tim Sort** | **Merge Sort + Insertion Sort (used in Python, Java)** |
| **Intro Sort** | **Quick Sort + Heap Sort + Insertion Sort (used in C++)** |
| **Merge-Insertion Sort** | **Merge Sort + Insertion Sort** |

**🔹They switch methods based on data size, pattern, or depth.**

**Abstract Data Type (ADT) means that the data type is defined by a set of operations, and not by its implementation details.**

* **For example:**
  + **Stack has operations like push, pop, and peek, but it doesn't specify how these operations are implemented.**
  + **Queue has operations like enqueue, dequeue, and front, but the implementation can vary (using arrays, linked lists, etc.).**

**Both stack and queue are defined by the operations they support, not by how they are implemented, making them ADTs.**

**✅ What is a Stack?**

**A stack is a linear data structure that works on the principle of:**

**LIFO – Last In, First Out**

**That means:  
 👉 The last item added is the first one to come out.**

### **📌 Real-Life Example:**

* **Stack of plates:  
   You place plates one over another. To take one, you remove the topmost plate first.**

### **📦 Why Use a Stack?**

* **Helps store data temporarily in a way that the most recent item is always on top.**
* **Useful when we need to undo actions, reverse things, or keep track of what came last.**

### **📚 Types of Stack:**

1. **Static Stack**
   * **Fixed size (like using an array)**
   * **Faster but limited memory**
2. **Dynamic Stack**
   * **Grows and shrinks (like using a linked list)**
   * **Flexible size**

### **🧠 Common Operations:**

* **push() → Add item to top**
* **pop() → Remove item from top**
* **peek() or top() → See the top item**
* **isEmpty() → Check if the stack is empty**

### **✅ Where is Stack Used? (Applications)**

* **Undo feature in editors (like Ctrl+Z)**
* **Backtracking (mazes, puzzles)**
* **Function call management (call stack)**
* **Reversing strings or numbers**

### **✅ What is Stack Pointer?**

**A stack pointer is a special pointer (or variable) that keeps track of the top element in the stack.**

**👉 It tells us where the last (top) item is stored in memory.**

### **⚠️ What is Stack Underflow?**

* **Happens when you try to pop (remove) an element from an empty stack.**
* **❌ Nothing to remove → Underflow Error**

**🧠 Example:**

**stack = []**

**stack.pop() // ❌ Underflow**

### **⚠️ What is Stack Overflow?**

* **Happens when you try to push (add) an element but the stack is already full (in fixed size stacks).**
* **❌ No more space → Overflow Error**

**🧠 Example:**

**stack = [1, 2, 3] // size 3**

**stack.push(4) // ❌ Overflow**

### **✅ What is an Array-Based Stack?**

* **A stack built using arrays.**
* **Size is fixed, defined when the stack is created.**

**🧠 Example:**

**js**

**CopyEdit**

**let stack = [ ]; // array as stack**

**stack.push(10);**

**stack.pop();**

### **✅ What is a Linked List-Based Stack?**

* **A stack built using linked list nodes.**
* **Size is not fixed, it grows and shrinks as needed.**

**🧠 How it works:**

* **Each new push() creates a new node**
* **pop() deletes the top node**

### **➕ Flexible, no overflow (until memory runs out)**

### **⚠️ Drawbacks of Stack (General):**

1. **Limited Access:  
    You can only access the top element, not random ones.**
2. **Fixed Size (in arrays):  
    May cause overflow if size is small.**
3. **Recursive Stack Overflow:  
    Too much recursion can overflow the call stack.**

### **✅ Advantages of Stack:**

1. **Simple and Fast:  
    Push and pop are quick – O(1) time.**
2. **Great for Reversal Tasks:  
    Like reversing a string or backtracking.**
3. **Used in Many Systems:  
    Function calls, expression parsing, undo/redo features.**

### **✅ What is a Queue?**

**A Queue is a linear data structure that works on the rule:**

**FIFO – First In, First Out**

**👉 The first element added is the first one removed.  
 Think of it like a line at a ticket counter.**

### **📌 Real-Life Example:**

* **People in a queue at a bus stop:  
   First person enters first, leaves first.**

### **🤔 Why Use a Queue?**

**Because it helps process data in the order it arrives.  
 Useful when things need to wait their turn.**

### **🧠 Main Operations in a Queue:**

| **Operation** | **What it Does** |
| --- | --- |
| **enqueue()** | **Add item to the rear of queue** |
| **dequeue()** | **Remove item from the front** |
| **peek()** | **See the front item** |
| **isEmpty()** | **Check if queue is empty** |

### **📚 Types of Queue with Uses + Examples:**

| **Type** | **What It Is** | **Example Use Case** |
| --- | --- | --- |
| **Simple Queue** | **Basic FIFO queue** | **Printer tasks, Call center lines** |
| **Circular Queue** | **Last position connects to first (reuses space)** | **OS scheduling, Streaming buffer** |
| **Priority Queue** | **Each element has a priority, highest goes first** | **Task management, Emergency room systems** |
| **Deque (Double Ended Queue)** | **Insert/delete from both ends** | **Browser history (back/forward), Palindrome check** |

### **✅ Where is Queue Used? (Applications)**

* **CPU task scheduling**
* **Managing print jobs**
* **Call center handling**
* **Breadth-First Search (BFS) in trees/graphs**
* **Data streaming (like YouTube buffering)**

### **✅ Advantages of Queue:**

1. **Orderly Processing (FIFO):  
    Ensures tasks are handled in the exact order they arrive.**
2. **Efficient in Scheduling:  
    Used in CPU, printer, or customer service to manage waiting tasks fairly.**
3. **Simple Operations:  
    Only need to work with front and rear, making implementation straightforward.**
4. **Useful in Real-Time Systems:  
    Like live video streaming, buffering, or handling real-time requests.**

### **❌ Drawbacks of Queue:**

1. **Limited Access:  
    You can only access the first item (front). No random access like arrays.**
2. **Overflow (in fixed-size queues):  
    If the queue is full, and space isn’t reused (in simple queue), it causes waste of space.**
3. **Slower in Linked List Queues:  
    Slightly more memory is used in dynamic queues due to node pointers.**
4. **No Prioritization (in simple queue):  
    All tasks treated equally — can't prioritize important tasks (unless using a priority queue).**

### **✅ What is a Hash Table?**

**A Hash Table is a data structure that stores data in key-value pairs.**

**It uses a special function called a hash function to quickly find where to store or get data.**

**👉 It allows very fast access, like almost O(1) time in best cases!**

### 

### **📌 Real-Life Example:**

* **Dictionary / Phonebook:  
   You search for a word (key) and get its meaning (value).**
* **Login system:  
   Username (key) → Password or User Data (value)**

### **🤔 Why Use a Hash Table?**

**Because it makes searching, inserting, and deleting very fast — much faster than arrays or lists.**

**It’s perfect when:**

* **You want to store and find data using a unique key**
* **You care about speed**

### **🧠 Key Operations in a Hash Table:**

| **Operation** | **Description** |
| --- | --- |
| **insert(key, value)** | **Store a value using a unique key** |
| **get(key)** | **Retrieve the value using its key** |
| **delete(key)** | **Remove the key-value pair** |
| **contains(key)** | **Check if a key exists** |

### **✅ Where is Hash Table Used? (Applications)**

* **Storing user info (login systems)**
* **Caching (storing recently used data)**
* **Counting items (word frequency, etc.)**
* **Compilers, Databases, Blockchain**

**🧠 Example:  
 Searching for a phone number using a name → fast in a hash table, slow in an array.**

**🟢 Advantages of Hash Tables:**

1. **Fast Lookup (O(1))**
   * **Quickly find values using their key, no need to search through all items.**
2. **Efficient Insertion & Deletion**
   * **Add or remove items quickly without shifting elements.**
3. **Key-Value Mapping**
   * **Easily store data as key-value pairs (e.g., name → phone number).**
4. **Flexible Keys**
   * **Keys can be numbers, strings, etc.**
5. **Dynamic Resizing**
   * **Grows automatically to handle more data when needed.**

**🔴 Disadvantages of Hash Tables:**

1. **Collisions**
   * **Two different keys might map to the same location, requiring special handling.**
2. **No Order**
   * **Items are not stored in any specific order.**
3. **Space Overhead**
   * **Takes up more memory compared to simpler structures like arrays.**
4. **Complex Hash Functions**
   * **A bad hash function can make performance worse.**
5. **Not Ideal for Range Queries**
   * **Can't easily find items within a range (like "all items between A and D").**

### **💥 What is Collision in Hashing?**

**A collision happens when two different keys are mapped to the same index by the hash function.**

**🧠 Example:**

**text**

**CopyEdit**

**hash("apple") → index 2**

**hash("banana") → index 2 ← collision!**

**This means both keys are trying to store data at the same place, which is not allowed.**

### **🎯 Why Collisions Happen?**

**Because hash tables have limited size, and many keys might be mapped to the same index, especially if the hash function isn't perfect.**

### **💡 How to Handle Collisions?**

**Two common methods:**

1. **Chaining – Store multiple values at the same index using a list.**
2. **Open Addressing – Find another empty spot nearby (like linear probing).**

**🌀 What is Clustering?**

**Clustering happens when many items in a hash table are stored near each other, making it harder and slower to insert or search for new items.**

**📌 Types of Clustering:**

1. **Primary Clustering**
   * **Happens when large groups of items form together in one area, usually because of linear probing (searching for the next available spot).**
2. **Secondary Clustering**
   * **This occurs when different keys with the same hash values follow the same path to find their spot, causing them to cluster together.**

**🤔 Example of Clustering:**

**Imagine the hash table looks like this:**

| **Index** | **4** | **5** | **6** | **7** |
| --- | --- | --- | --- | --- |
| **Data** | **X** | **X** | **X** | **(New data tries to go here)** |

**Now, when you try to add new items, they have to move through this crowded area (from index 4 to 7), slowing everything down.**

### **✅ What is a Hash Function?**

**Special function A hash function takes a key (like a string or number) and converts it into a fixed index (number).  
 That index tells the hash table where to store or find the value.**

**🧠 Example:**

**js**

**CopyEdit**

**hash("sharik") → 3**

**// So the value is stored at index 3 in the table**

### **⭐ Properties of a Good Hash Function:**

1. **Deterministic – Same key always gives the same index**
2. **Uniform Distribution – Spreads keys evenly to avoid clustering**
3. **Fast to Compute – Must work quickly even for large inputs**
4. **Minimize Collisions – Tries to avoid two keys mapping to same index**
5. **Handle All Types – Works for strings, numbers, etc.**

### **🎯 What is Perfect Hashing?**

**A perfect hash function is one that causes no collisions.**

* **Every key gets a unique index**
* **Works when the key set is known in advance**

**🧠 Example: Useful in compilers or static lookup tables.**

### **🔧 What is ASCII Hashing?**

**ASCII Hashing uses the ASCII values of characters in the key to create the hash.**

**🧠 Example:**

**js**

**CopyEdit**

**hash("abc") = 97 + 98 + 99 = 294**

* **Simple, but may cause many collisions if not modified.**

### **🏹 What is Robin Hood Hashing?**

**Robin Hood Hashing is a collision resolution strategy in open addressing.**

**📌 How it works:**

* **If a new item’s “home” slot is taken, it compares distances:  
    
    
   "If I’ve traveled more than you to get here, I take your spot!"**

**It balances the hash table by spreading out items fairly.**

**🧠 Use case: Helps reduce variance in search time.**

### **🔑 1. What are Common Hash method?**

**Hash functions turn a key into a number (index). Some popular ones:**

| **Hash Function** | **Description** |
| --- | --- |
| **Division Method** | **index = key % table\_size – Simple and fast** |
| **Multiplication Method** | **Uses fractions and multiplication to spread keys** |
| **ASCII/Character Sum** | **Adds up ASCII values of characters in a string** |
| **DJB2 / SHA-256 / MD5** | **Used in real-world applications for better distribution** |

### **Key Considerations in a Hash Table:**

1. **Good Hash Function**
   * **Why it matters: Helps avoid collisions and ensures even distribution of values.**
2. **Size of Table**
   * **Why it matters: A prime number size gives better value distribution.**
3. **Collision Handling**
   * **Why it matters: Solves issues when two values land on the same spot (using methods like chaining or open addressing).**
4. **Load Factor**
   * **Why it matters: The higher the load factor, the slower the performance (more items compared to table size).**
5. **Resizing**
   * **Why it matters: Automatically grows the table when it gets too full for better performance.**

### **🔁 3. What is Dynamic Restructuring?**

**Dynamic Restructuring = Changing the structure of the hash table when needed.**

**🧠 Why?**

* **If too many collisions → performance drops**
* **If table is full → we resize and rehash**

**📌 Rehashing means:**

**Recalculating indexes for all keys with a new table size.**

**This keeps performance fast even with more data.**

### **🌱 4. What is WeakSet and WeakMap? (JavaScript)**

**These are special data structures in JavaScript that hold weak references to objects.**

#### **🔹 WeakSet:**

* **Like a Set, but only stores objects**
* **Doesn’t prevent objects from being garbage collected**

**js**

**CopyEdit**

**let ws = new WeakSet();**

**ws.add({name: "Sharik"});**

#### **🔹 WeakMap:**

* **Like a Map, but keys must be objects**
* **Values can be anything**
* **Helps prevent memory leaks**

**let wm = new WeakMap();**

**let obj = {id: 1};**

**wm.set(obj, "Sharik’s Data");**

**🔐 These are used when you want temporary storage without affecting memory too much.**

### **Difference Between Hash Keys and Array Keys:**

| **Feature** | **Hash Keys** | **Array Keys** |
| --- | --- | --- |
| **Type of Key** | **Can be strings, numbers, or objects** | **Usually integers (0, 1, 2, 3...)** |
| **Access Method** | **Accessed directly via key (key → value)** | **Accessed by index (position in array)** |
| **Order** | **No guaranteed order** | **Elements are stored in order** |
| **Duplicates** | **No duplicate keys (values can be same)** | **Allows duplicate values at different indices** |
| **Usage** | **Used in hash tables for fast lookups** | **Used in arrays or lists for ordered data** |

**🧠 Example:**

* **Hash Table: "name" → "Sharik", "age" → 25**
* **Array: ["apple", "banana", "cherry"]**

### 

### 

### 

### **What is Secure Hash Algorithm (SHA)?**

**SHA is a set of cryptographic hash functions that turn data into a fixed-size hash. It’s designed to be secure, ensuring small changes in the input create large differences in the hash output.**

**🧠 Key Features of SHA:**

* **Fixed Length Output: Output is always the same size, no matter the input size.**
* **Deterministic: Same input → same output.**
* **Fast Computation: Quickly computed for any data size.**
* **Pre-image Resistance: Hard to figure out the original input from the hash.**
* **Collision Resistance: Very hard to find two different inputs that give the same hash.**
* **Avalanche Effect: Small input changes result in very different output.**

### **🔄 What are the Variants of SHA?**

**The SHA family includes several different hash algorithms with different output sizes:**

1. **SHA-0: Old, not widely used, and insecure.**
2. **SHA-1: Produces a 160-bit hash value, but no longer considered secure for cryptographic use due to vulnerabilities.**
3. **SHA-2: Currently the most widely used, producing hash values of 224, 256, 384, or 512 bits (SHA-224, SHA-256, SHA-384, SHA-512).**
4. **SHA-3: The latest standard, which uses a different construction and provides stronger security.**

### **🔐 What are the Uses of SHA?**

**SHA is widely used for:**

* **Password storage: Hashing passwords before storing them for security.**
* **Digital signatures: Ensuring data integrity by hashing the message and signing it.**
* **Data integrity checks: Verifying the integrity of files and messages (e.g., checksums).**
* **Blockchain: In cryptocurrency like Bitcoin, SHA-256 is used to create blocks.**

**What is load factor ?**

**"Load factor tells how full the hash table is.**

**If it's too full, it causes more collisions and slows things down.**

**We solve this by rehashing (making the table bigger).**

**A good load factor is around 0. 7."**

### **🔹 What is a Linear Data Structure?**

**📌 \*\*Elements are arranged in a sequence (one after another).**

#### **✅ Examples:**

* **Array**
* **Linked List**
* **Stack**
* **Queue**

#### **🧠 Key Points:**

* **One-to-one relationship (next-next-next…)**
* **Easy to traverse (from start to end)**
* **Memory is used in order**

### **🔸 What is a Non-Linear Data Structure?**

**📌 \*\*Elements are arranged in a hierarchical or branching way.**

#### **✅ Examples:**

* **Tree**
* **Graph**
* **Heap**
* **Trie**

#### **🧠 Key Points:**

* **One-to-many relationships (parent → children)**
* **Not easy to traverse in a straight line**
* **Used when relationships are complex (e.g., maps, networks, file systems)**

### **✅ What is a Tree (in DSA)?**

**A tree is a hierarchical data structure made of nodes connected like branches.  
 It starts from a root node, and each node can have child nodes.**

**A ← root**

**/ \**

**B C ← children**

**/ \**

**D E ← more children**

### **📌 Real-Life Examples:**

1. **Family Tree – Shows parents, children, and generations.**
2. **Folders in a Computer – A main folder contains subfolders and files.**
3. **Company Org Chart – CEO at top → Managers → Employees.**
4. **Game Menus – Main menu → Submenus → Options.**

### **🤔 Why Use Trees?**

* **To represent hierarchical relationships.**
* **To organize data in a way that’s fast to search, insert, or delete.**
* **Useful in cases where order matters (like file systems or decision-making).**

### **✅ Features of a Tree**

| **Feature** | **Description** |
| --- | --- |
| **🔗 Hierarchical Structure** | **Tree represents data in a parent-child relationship.** |
| **🌱 Root Node** | **The topmost node (starting point) of the tree.** |
| **🌿 Leaves** | **Nodes with no children (end of a path).** |
| **➕ Nodes** | **Each element in the tree is called a node.** |
| **🔁 Recursive Structure** | **Each subtree is also a tree (self-similar).** |
| **➖ No Loops** | **A tree is acyclic—no cycles or backward paths.** |
| **📏 Levels** | **Distance from the root to a node (root is level 0).** |
| **🔍 Depth/Height** | **Depth: node’s level from root; Height: longest path from root to leaf.** |

### **🔧 How It Works (Basic Terms):**

* **Root: Topmost node (starting point).**
* **Child: Node connected below another node.**
* **Parent: Node above a child.**
* **Leaf: Node with no children.**
* **Edge: Connection between two nodes.**
* **Subtree: Part of the tree under a node.**

### 

### **📱 Applications of Trees:**

| **Application** | **Use** |
| --- | --- |
| **📁 File System** | **Folders/files structure** |
| **🗂 Databases** | **Indexing (e.g., B-Trees)** |
| **🌐 HTML DOM** | **Web page structure** |
| **🧠 AI** | **Decision trees** |
| **🔍 Searching** | **Binary Search Trees (BST)** |
| **🔤 Spell Check** | **Tries (Prefix Trees)** |

### 

### 

### **🌳 Basic Tree Terminologies**

| **Term** | **Meaning** |
| --- | --- |
| **Node** | **Each item in a tree (can hold data).** |
| **Root** | **The topmost node (starting point of the tree).** |
| **Parent** | **A node that has children.** |
| **Child** | **A node that comes from a parent.** |
| **Leaf** | **A node with no children.** |
| **Edge** | **The line or link between two nodes.** |
| **Subtree** | **A smaller tree formed under any node.** |
| **Siblings** | **Nodes that share the same parent.** |
| **Level** | **Depth of a node from the root (Root is Level 0).** |
| **Height of Node** | **Number of edges from that node to the deepest leaf.** |
| **Height of Tree** | **Height of the root node (longest path from root to any leaf).** |
| **Depth of Node** | **Number of edges from root to that node.** |
| **Degree of Node**  **N nodes** | **Number of children a node has.**  **N-1 edges** |
| **Internal Node** | **A node that is not a leaf (has at least one child).** |
| **External Node** | **A leaf node (no children).** |
| **Ancestor** | **Any node above the current node in the path to root.** |
| **Descendant** | **Nodes that come below a given node in the tree.** |

**LEVEL :**

**A ← Level 0 (Root)**

**/ \**

**B C ← Level 1**

**/ \ \**

**D E F ← Level 2**

### **✅ Why Do We Use Different Types of Binary Trees?**

**We use different types because each one is designed to solve a specific problem — like fast searching, saving memory, or handling sorted data. Choosing the right type helps make programs faster and more efficient.**

### **🌳 Types of Binary Trees (What, Why, and Real-Life Example)**

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

* **What: Just a tree where each node has up to 2 children.**
* **Why: When no specific structure is needed.**
* **Example: Expression trees in calculators (e.g., (3 + 5) \* 2)**

#### **2. Full Binary Tree**

* **What: Every node has 0 or 2 children (no single child).**
* **Why: Makes structure predictable.**
* **Example: Family tree models where each person has two parents.**

#### **3. Complete Binary Tree**

* **What: All levels are fully filled, except maybe the last (filled left to right).**
* **Why: Used in heaps for efficient priority operations.**
* **Example: Task manager apps where you need quick access to highest priority tasks.**

#### **4. Perfect Binary Tree**

* **What: All nodes have 2 children and all leaves are at the same level.**
* **Why: Symmetric and easy to balance.**
* **Example: Tournament brackets where every player competes evenly.**

#### **5. Degenerate (Pathological) Tree**

* **What: Every node has only one child (like a linked list).**
* **Why: Not efficient, but good for worst-case examples.**
* **Example: Poorly built contact list, where every new name is added at the end.**

#### **6. Balanced Binary Tree**

* **What: Height of left and right subtree differ by max 1.**
* **Why: Keeps operations fast (O(log n)).**
* **Example: Autocomplete system where typing is fast even with many words.**

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

* **What: Left child < parent < right child.**
* **Why: For fast searching, insertion, and deletion.**
* **Example: Phone contact search system (names sorted A–Z).**

#### **8. AVL Tree**

* **What: A self-balancing BST (difference of heights ≤ 1).**
* **Why: Keeps things balanced always → fast operations.**
* **Example: Database indexing, where data needs to be fetched quickly.**

#### **9. Red-Black Tree**

* **What: A balanced BST using red/black color rules.**
* **Why: Slightly less strict than AVL, easier to maintain.**
* **Example: Used in libraries like Java's TreeMap or C++ STL map.**

#### **10. Trie (Prefix Tree)**

* **What: Each node stores letters, used for words/prefixes.**
* **Why: Fast prefix search.**
* **Example: Google Search Autocomplete (“ap” → “apple”, “april”, etc.)**

### **🌳 What is a Binary Tree? Or Full /proper/strict Binary Tree ?**

**A Binary Tree is a tree data structure where each node has at most 2 children — called the left child and right child.**

### **🔢 Max and Min Nodes in a Binary Tree**

**Let’s say height = h (starting from 0).**

#### **✅ Maximum Nodes**

* **A binary tree can have at most 2^level nodes at each level.**
* **So for height h, the total =  
   2⁰ + 2¹ + 2² + ... + 2^h = 2^(h+1) - 1**

**📌 Formula:  
 👉 Max nodes = 2^(h+1) - 1**

**📍 Example:  
 If h = 2:  
 → 2^(2+1) - 1 = 2³ - 1 = 8 - 1 = 7 nodes  
 ✔️ That’s a fully filled tree with 3 levels (0, 1, 2).**

#### **✅ Minimum Nodes**

* **Happens when each node has only 1 child (like a linked list). 📌 Formula:  
   👉 Min nodes = h + 1**

**📍 Example:  
 If h = 2:  
 → 2 + 1 = 3 nodes  
 ✔️ Just 1 node per level (linked list shape).**

### **📏 Min and Max Height of a Binary Tree**

**Let’s say you know the number of nodes = n.**

#### **✅ Minimum Height**

* **Happens in a perfectly balanced tree.**
* **You keep doubling nodes each level (like a pyramid). 📌 Formula:  
   👉 Min height = log₂(n) (rounded down)**

**Log 👉 "How many times do you divide n by 2 until you reach 1?"This is called log base 2, written as log₂(n).**

### **✅ Example 1: log₂(8)**

* **8 ÷ 2 = 4**
* **4 ÷ 2 = 2**
* **2 ÷ 2 = 1**

**You divided 3 times, so:  
 log₂(8) = 3**

**📍 Example:  
 If n = 15:  
 → log₂(15) ≈ 3.9 → Rounded down = 3**

**✔️ So with 15 nodes, the shortest perfect binary tree has height 3.**

#### **✅ Maximum Height**

* **Happens when the tree becomes like a linked list (worst case). 📌 Formula:  
   👉 Max height = n - 1**

**📍 Example:  
 If n = 4:  
 → Max height = 4 - 1 = 3**

**✔️ Just 1 node per level, all down one side.**

### **🌳 Types of Trees Based on Nodes**

**These are categorized by how many children each node has:**

| **Type** | **Description** |
| --- | --- |
| **Binary Tree** | **Each node has at most 2 children (left and right).** |
| **Ternary Tree** | **Each node can have up to 3 children.** |
| **N-ary Tree** | **Each node can have up to N children.** |
| **Leaf Node** | **A node with no children.** |
| **Internal Node** | **A node with at least one child.** |

### **🧱 Types of Trees Based on Structure**

**These define how the tree is organized or shaped:**

| **Type** | **Description** |
| --- | --- |
| **Full Binary Tree** | **Every node has 0 or 2 children (no node with only 1 child).** |
| **Complete Binary Tree** | **All levels are completely filled, except possibly the last (filled left to right).** |
| **Perfect Binary Tree** | **All internal nodes have 2 children and all leaves are at the same level.** |
| **Degenerate Tree** | **Each parent has only one child, behaves like a linked list.** |
| **Balanced Tree** | **The height difference between left and right subtrees is not more than 1.** |
| **Skewed Tree** | **All nodes are on one side—either left-skewed or right-skewed.** |

### **🌳 What is BST (Binary Search Tree)?**

**A Binary Search Tree is a special type of binary tree where:**

* **Left child < Parent**
* **Right child > Parent**

**✅ Why use BST? To search, insert, and delete data quickly (better than a normal array or linked list if balanced).**

**✅ How it works? Data is arranged in sorted order so you can skip half the tree each time (like binary search).**

**🧠 Example:**

**markdown**

**Copy code**

**10**

**/ \**

**5 15**

**Here, 5 is < 10 and 15 is > 10 — BST rule followed!**

### **🚶‍♂️ What is BFT (Breadth-First Traversal)?**

**BFT = Level Order Traversal → Visit nodes level by level (top to bottom, left to right).**

**✅ Uses of BFT:**

* **Find shortest path in graphs/trees**
* **Helpful in AI, network routing, and file system searches**

### **⚖️ Balanced vs Unbalanced Tree**

| **Feature** | **Balanced Tree** | **Unbalanced Tree** |
| --- | --- | --- |
| **Shape** | **Heights of left/right subtrees ≈ equal** | **One side is much deeper** |
| **Speed** | **Fast operations (O(log n))** | **Slower (can become like a linked list)** |
| **Looks like** | **Proper triangle/tree shape** | **Tall and thin** |

**🧠 Balanced trees = better performance!**

### **✅ Properties of BST**

* **Each node has 0, 1 or 2 children**
* **Left subtree < root < right subtree**
* **No duplicates (in most cases)**
* **Fast search if the tree is balanced**
* **In-order traversal gives sorted output**

### **BST Operations (Keys Only):**

* **Insert – Add value using BST rules (left <, right >).**
* **Search – Check if a value exists.**
* **Delete – Remove node (3 cases: leaf, 1 child, 2 children).**

### **🌳 Tree Traversal Techniques:**

* **In-order (L → Root → R)  
   👉 Use: Gets nodes in sorted order (for BST).  
   📌 Example: [1, 2, 3, 4, 5]**
* **Pre-order (Root → L → R)  
   👉 Use: Used to copy or recreate the tree.  
   📌 Saves structure → good for serialization.**
* **Post-order (L → R → Root)  
   👉 Use: Used to delete the tree.  
   📌 Deletes children before parent (safe memory release).**
* **Level-order (BFS)  
   👉 Use: Breadth-First Traversal (go level by level).  
   📌 Good for printing tree layer by layer (like a queue).**
* **Find Min – Go leftmost**
* **Find Max – Go rightmost**

### **🔁 What is Traversal?**

**Traversal means visiting all nodes in a tree in a specific order.**

### 

### 

### 

### **🔎 DFS (Depth-First Search)**

**Goes deep first, exploring as far as possible along each branch.**

**Types:**

* **In-order (Left → Root → Right) → ✅ Sorted output (BST)**
* **Pre-order (Root → Left → Right) → ✅ Used to copy tree**
* **Post-order (Left → Right → Root) → ✅ Used to delete/free tree**

### **🔹 1. PreOrder Traversal (Root → Left → Right)**

**Steps:**

1. **Visit current node (print root value)**
2. **Traverse left subtree using preOrder**
3. **Traverse right subtree using preOrder**

### **🔹 2. InOrder Traversal (Left → Root → Right)**

**Steps:**

1. **Traverse left subtree using inOrder**
2. **Visit current node (print value)**
3. **Traverse right subtree using inOrder**

### **🔹 3. PostOrder Traversal (Left → Right → Root)**

**Steps:**

1. **Traverse left subtree using postOrder**
2. **Traverse right subtree using postOrder**
3. **Visit current node (print value)**

### **🌐 BFS (Breadth-First Search) / Level-order**

**Visits level by level from top to bottom, left to right.**

**✅ Used in shortest path (unweighted), hierarchy printing.**

### **🔹 4. LevelOrder Traversal (Breadth-First Search)**

**Steps:**

1. **✅ Create an empty queue.**
2. **✅ Enqueue the root node.**
3. **While the queue is not empty:**
   * **🔁 Dequeue the front node.**
   * **📌 Print the node's value.**
   * **📥 Enqueue its left child (if it exists).**
   * **📥 Enqueue its right child (if it exists).**

### **🌳 What is a Balanced Search Tree?**

**A Balanced Search Tree keeps the height small (logarithmic), so that operations like search, insert, and delete stay fast.**

**✅ Goal: Keep the tree evenly balanced so it doesn’t become like a linked list.**

### **💡 Why Balance Matters?**

* **Unbalanced tree = long chains = 🔴 slow search (O(n))**
* **Balanced tree = short height = ✅ fast search (O(log n))**

### 

### **🔑 Examples of Balanced Search Trees:**

| **Tree Type** | **Simple Meaning** | **Balance Property** |
| --- | --- | --- |
| **AVL Tree** | **Self-balancing BST** | **Height of left and right subtrees differs by at most 1** |
| **Red-Black Tree** | **Special BST with red/black rules** | **Keeps height approx. log(n)** |
| **B-Trees** | **Used in databases** | **Keeps data balanced and sorted, good for disk storage** |
| **2-3 Tree** | **Each node can have 2 or 3 children** | **Auto-balances during insert/delete** |

### **🔴 Red-Black Tree vs ✅ AVL Tree**

| **Feature** | **🟥 Red-Black Tree** | **✅ AVL Tree** |
| --- | --- | --- |
| **Balance Strictness** | **Less strict** | **More strict** |
| **Rotation Frequency** | **Fewer rotations** | **More rotations** |
| **Performance (Insert/Delete)** | **Faster for insert/delete** | **Slower for insert/delete** |
| **Performance (Search)** | **Slower than AVL** | **Faster than Red-Black** |
| **Height Guarantee** | **log(n) \* 2** | **log(n)** |
| **Used In** | **Real-world systems like Linux, Java** | **Memory-limited apps, fast searches** |
| **Maintenance Cost** | **Low** | **High** |

### **⏱️ Time Complexities**

| **Operation** | **🌳 Tree** | **🌲 Binary Tree** | **🌴 BST** | **✅ Balanced BST (AVL/Red-Black)** |
| --- | --- | --- | --- | --- |
| **Insert** | **O(n)** | **O(n)** | **O(n)** | **O(log n)** |
| **Search** | **O(n)** | **O(n)** | **O(n)** | **O(log n)** |
| **Delete** | **O(n)** | **O(n)** | **O(n)** | **O(log n)** |
| **Inorder/Preorder/Postorder** | **O(n)** | **O(n)** | **O(n)** | **O(n)** |
| **Min/Max** | **O(n)** | **O(n)** | **O(h)** | **O(log n)** |
| **Kth Smallest/Largest** | **O(n)** | **O(n)** | **O(h)** | **O(log n)** |
| **Successor/Predecessor** | **O(n)** | **O(n)** | **O(h)** | **O(log n)** |

### **⏱️ Space Complexities**

| **Operation** | **🌳 Tree** | **🌲 Binary Tree** | **🌴 BST** | **✅ Balanced BST (AVL/Red-Black)** |
| --- | --- | --- | --- | --- |
| **Insert** | **O(n)** | **O(n)** | **O(n)** | **O(n)** |
| **Search** | **O(n)** | **O(n)** | **O(n)** | **O(n)** |
| **Delete** | **O(n)** | **O(n)** | **O(n)** | **O(n)** |
| **Traversal** | **O(n)** | **O(n)** | **O(n)** | **O(n)** |
| **Height** | **O(n)** | **O(n)** | **O(n)** | **O(log n)** |
| **Balanced Property** | **O(n)** | **O(n)** | **O(n)** | **O(log n)** |

### **✅ Advantages of Binary Search Tree (BST)**

1. **Fast Search – If balanced, operations like search, insert, delete take O(log n) time.**
2. **Sorted Order – In-order traversal gives sorted data easily.**
3. **Dynamic – Efficiently handles dynamic data (unlike arrays which need shifting).**

### **❌ Disadvantages of BST**

1. **Can Become Unbalanced – Worst-case becomes like a linked list (O(n) time).**
2. **Hard to Maintain Balance – Needs special types (like AVL or Red-Black Tree) to stay efficient.**
3. **Slower Than Hash Table – For pure search, hashing is faster (O(1)) than BST (O(log n)).**

### **🔺 What is a Heap?**

**A Heap is a special kind of binary tree used to manage data based on priority.**

**✅ It always follows:**

* **Complete binary tree structure (all levels filled left to right)**
* **Heap Property 2 type:**
  + **Min Heap: Parent is smaller than children**
  + **Max Heap: Parent is larger than children**

### **🧠 Real-Life Example:**

**💼 Job Scheduler (CPU or Task Manager)**

**You have multiple tasks, each with different priorities:**

* **Task A: High Priority**
* **Task B: Medium Priority**
* **Task C: Low Priority**

**The system uses a heap to always pick the highest priority task first — just like a Max Heap!**

### **❓ Why Use a Heap?**

* **To quickly access the smallest or largest element**
* **It’s faster than sorting every time**
* **It saves time and memory in problems where order matters**

### **🔧 Applications of Heap**

1. **✅ Priority Queue (like patient treatment system in hospitals)**
2. **🔁 Heap Sort (sorting algorithm)**
3. **🧭 Dijkstra’s Algorithm (shortest path in maps)**
4. **🌐 Prim’s Algorithm (minimum spanning tree)**
5. **🎯 Finding Kth Largest/Smallest item**
6. **🧩 Merge K sorted arrays**
7. **🗜️ Huffman Coding (data compression)**
8. **⏳ Event Scheduling**
9. **📊 Median Maintenance**

### 

### **A Heap is usually implemented as a complete binary tree stored in an array.**

### **Why an array? Because you can calculate parent and children indices using formulas — no need for pointers like in trees**

### **📌 1. Child/Parent Node Index in Binary Heap (Using Array)**

**In a binary heap stored as an array:**

* **Left Child of node at index i = 2\*i + 1**
* **Right Child = 2\*i + 2**
* **Parent = (i - 1) // 2**

**✅ This makes it easy to navigate a heap stored in an array format.**

### **📌 2. What is Heapify?**

**🧠 Heapify = Process of rearranging nodes to maintain the heap property (min or max).**

* **After insertion or deletion, heap property may break.**
* **So, we call heapify to fix it!**

### **📌 3. Bottom-Up Heapify (Heapify Up)**

**🔼 Used during insertion:**

* **Start from the inserted node (bottom)**
* **Compare with parent**
* **If heap property breaks → swap upward**
* **Continue until it fits the heap**

**✅ Maintains heap from leaf to root**

**🕒 Time: O(log n)**

### **📌 4. Top-Down Heapify (Heapify Down)**

**🔽 Used during deletion:**

* **Start from root**
* **Compare with children**
* **Swap with smaller (min heap) or larger (max heap) child**
* **Continue until it fits the heap**

**✅ Maintains heap from root to leaf**

**🕒 Time: O(log n)**

### **📌 5. What is DEPQ (Double Ended Priority Queue)?**

**🧮 DEPQ = A special structure that lets you:**

* **Access both min and max in constant time**
* **Insert/delete from both ends**

**📦 Can be implemented using:**

* **Two heaps (min-heap + max-heap)**
* **Or special data structures**

**📍Use Cases: Scheduling, simulations, dual-ended comparisons**

### 

### **✅ What is Heap Sort?**

**Heap Sort is a sorting technique that uses a binary heap (usually a max heap) to sort elements.**

### **❓ Why Use Heap Sort?**

* **✅ Time complexity: O(n log n) — efficient!**
* **✅ Works in-place (no extra memory needed)**
* **✅ Good for priority-based sorting**
* **❌ Not stable (doesn’t preserve order of equal elements)**

### **⚙️ How Heap Sort Works (Step-by-Step)**

**We’ll use a Max Heap so the largest element comes to the top:**

**🔁 Recap**

1. **Build Max Heap**
2. **Swap root with last item**
3. **Heapify remaining part**
4. **Repeat until sorted**

**All done in O(n log n) time!**

### **✅ Advantages of Heap:**

1. **Fast access to min/max element in O(1) time**
2. **Efficient insert/delete in O(log n) time**
3. **Useful for priority-based tasks (like job scheduling)**

### **❌ Disadvantages of Heap:**

1. **Not good for searching random elements (O(n) time)**
2. **Not memory efficient like arrays or linked lists**
3. **Not stable for sorting (equal values may lose order)**

**Heap ?**

**Insert, delete: O(log n).**

**Peek: O(1).**

**Heapify/Build Heap: O(n).**

**Search: O(n).**

**Space: O(n).**

### **📚 What is a Trie?**

* **A Trie (pronounced "try") is a special tree-like data structure.**
* **It is used to store and search words very fast, letter by letter.**

**✅ Each node represents one character (letter).**

**✅ Paths down the tree represent complete words.**

### **🔥 Example of Trie:**

**Let's insert these words: "cat", "car", "care"**

**scss**

**CopyEdit**

**(root)**

**└── c**

**└── a**

**├── t (cat)**

**└── r**

**└── e (care)**

**✅ You can see that common letters (c → a) are shared to save space.**

### **🎯 Why Use Trie?**

* **Fast Word Search:  
   Find a word in O(length of word) time.**
* **Prefix Searching:  
   Quickly find words that start with a prefix.  
   (Example: all words starting with "ca")**
* **Efficient Memory Usage:  
   Common prefixes are stored once, not again and again.**

### **🚀 Applications of Trie:**

| **Application** | **Example** |
| --- | --- |
| **🔍 Autocomplete** | **Typing "ca" suggests "cat", "car", "care"** |
| **📚 Spell Checker** | **Finding wrong spelling suggestions** |
| **🛡️ IP Routing (Networking)** | **Fast lookup of IP addresses** |
| **🕵️ Pattern Matching Algorithms** | **Finding patterns in texts or DNA sequences** |
| **📦 Word Games (like Scrabble, Boggle)** | **Checking if a word is valid** |

# **1️⃣ String vs Trie**

| **String** | **Trie** |
| --- | --- |
| **Stores one word or sentence** | **Stores many words sharing letters** |
| **Simple data type ("cat", "dog")** | **Special tree-like structure** |
| **Searching a word takes O(n) time** | **Searching can be faster (especially with many words)** |
| **No memory sharing** | **Shares memory when words have common prefixes** |

**🔵 In short:**

* **String = Just a word or text.**
* **Trie = Many words stored smartly together for fast search.**

# **2️⃣ What is Prefix and Suffix Tree?**

**✅ Prefix Tree (Trie):**

* **A tree where paths represent prefixes of words.**
* **Example: "car", "care", "cat" — all start with "ca".**

**✅ Suffix Tree:**

* **A tree where paths represent suffixes (ending parts) of a word.**
* **Example: For "banana", suffixes are "banana", "anana", "nana", "ana", "na", "a".**

**🔵 In short:**

* **Prefix Tree → Focus on starting parts of words.**
* **Suffix Tree → Focus on ending parts of words.**

# 

# 

# **3️⃣ What is a Terminator Character?**

**✅ A special symbol added at the end of a word in a Trie.**

**✅ It tells the Trie, "This is the end of the word."  
 (Otherwise, the Trie won’t know where a word stops.)**

**Example:**

* **"cat" would be stored as c → a → t → \* (where \* is terminator)**

**Common terminator: \* or $ symbol.**

**🔵 In short:**

* **Terminator character = Marks end of word inside a Trie.**

### **1️⃣ What is a Compressed Trie?**

* **A Compressed Trie (or Radix Tree) is a type of Trie where common parts of words are stored together in one node, instead of storing each character separately.**

#### **Example:**

**For the words:**

* **"cat"**
* **"car"**
* **"dog"**

**In a normal Trie (before compression), you would store them like this:**

**scss**

**CopyEdit**

**(root)**

**└── c**

**├── a**

**│ ├── t (cat)**

**│ └── r (car)**

**└── d**

**└── o**

**└── g (dog)**

**In a Compressed Trie, it will combine common parts and store them in fewer nodes:**

**scss**

**CopyEdit**

**(root)**

**└── c**

**├── at (cat)**

**└── ar (car)**

**└── dog (dog)**

### **2️⃣ What is a Radix Tree?**

* **A Radix Tree is the same as a Compressed Trie. The only difference is that in a Radix Tree, each node stores multiple characters (a prefix) instead of just one character.**

#### **Example:**

**For the words:**

* **"apple"**
* **"appl"**
* **"apex"**

**In a normal Trie (before compression), you'd store each character in a separate node:**

**scss**

**CopyEdit**

**(root)**

**└── a**

**└── p**

**└── p**

**└── l**

**├── e (apple)**

**└── (end of word) (appl)**

**└── x (apex)**

**In a Radix Tree, you store the common prefix "ap" in one node:**

**scss**

**CopyEdit**

**(root)**

**└── ap**

**├── ple (apple)**

**├── pl (appl)**

**└── ex (apex)**

### **✅ Advantages of Trie:**

1. **Fast Search:**
   * **Searching for a word is quick (O(m) time), where m is the length of the word.**
2. **Efficient Prefix Matching:**
   * **Easily find words that start with a given prefix.**
3. **Memory Efficiency (for shared prefixes):**
   * **Tries share common parts (prefixes) between words, reducing memory usage.**

### **❌ Disadvantages of Trie:**

1. **High Space Complexity:**
   * **Tries can take up a lot of memory if there are many words with few common prefixes.**
2. **Complex to Implement:**
   * **Tries are harder to implement compared to simpler data structures like arrays.**
3. **Inefficient for Small Datasets:**
   * **For storing just a few words, Tries might use more memory than needed.**

### **⏱️ Time Complexity:**

* **Search/Insert a Word: Takes time based on the length of the word.**
  + **O(m), where m is the number of characters in the word.**

### **💾 Space Complexity:**

* **Takes up space based on the number of words and the average length of the words.**
  + **O(N \* m), where N is the number of words and m is the average length.**

### **What is a Graph?**

* **A graph is a non-linear data structure made up of:**
  + **Vertices (Nodes): These are the points or elements in the graph (like cities in a network, people in a social network, etc.).**
  + **Edges (Arcs): These are the connections or relationships between the vertices (like roads between cities, or friends between people).**

### **Key Terms:**

* **Vertex (Node): A single point or element in the graph (e.g., a city, person, or location).**
* **Edge (Arc): A connection between two vertices, showing how they are related or connected.**
* **Degree: The number of edges connected to a vertex. It shows how many connections that vertex has.**

### **Example:**

* **Imagine a graph where cities are vertices and roads are edges. The degree of a city (vertex) would be the number of roads (edges) connected to it.**

### **Why Use Graphs?**

**Graphs are useful when you need to represent relationships or connections between different elements. They help model complex systems and structures where the connections between elements are important.**

**Here are the 4 key applications of graphs:**

1. **Social Networks**
2. **Navigation Systems**
3. **Web Page Links**
4. **Recommendation Systems**

### **Adjacency Matrix:**

* **A 2D array where rows and columns represent vertices.**
* **If there's an edge between two vertices, the corresponding position is 1; if no edge, it's 0.**

#### **Example:**

**For a graph with 3 vertices (A, B, C) and edges (A-B, B-C):**

**css**

**CopyEdit**

**A B C**

**A [0, 1, 0]**

**B [1, 0, 1]**

**C [0, 1, 0]**

* **A-B (1 means edge exists between A and B).**
* **B-C (1 means edge exists between B and C).**

### **Adjacency List:**

* **A list for each vertex, where each list contains the vertices it's connected to.**

#### **Example:**

**For the same graph:**

**css**

**CopyEdit**

**A → [B]**

**B → [A, C]**

**C → [B]**

* **A is connected to B.**
* **B is connected to A and C.**
* **C is connected to B.**

**What are type of graph?**

### **1️⃣ Directed Graph (Unidirectional):**

* **Edges have a direction**. An edge from A to B doesn't mean there's an edge from B to A.

#### **Example:**

* **A → B** (There is an edge from A to B, but not necessarily the other way around).

### **2️⃣ Undirected Graph (Bidirectional):**

* **Edges don't have a direction**. If there's an edge between A and B, you can travel both ways.

#### **Example:**

* **A - B** (You can go from A to B and from B to A).

### **3️⃣ Cyclic Graph:**

* A graph that has at least one **cycle**, meaning there's a path that goes from a vertex and eventually returns to it.

#### **Example:**

* **A → B → C → A** (A cycle exists because you can go from A to B to C and back to A).

### **4️⃣ Disconnected Graph:**

* A graph is **disconnected** if there are vertices that **don't have any path** connecting them.

#### **Example:**

* **A → B**, but **C** is not connected to either A or B. There’s no path between C and others.

### **5️⃣ Weighted Graph:**

* Each edge has a **weight** (a value) that could represent cost, distance, or time.

#### **Example:**

* **A --(5)-- B** (The edge between A and B has a weight of 5, like a distance or cost).

### **6️⃣ Unweighted Graph:**

* All edges are treated as having the **same weight** or no weight at all.

#### **Example:**

* **A - B** (Each edge is equal, with no specific weight).

### **7️⃣ Bipartite Graph:**

* The vertices can be divided into two sets, where every edge connects a vertex from one set to the other. **No edges exist within the same set**.

#### **Example:**

* Set 1: **A, B**
* Set 2: **C, D**
* Edges: **A - C**, **B - D** (Edges only connect vertices from one set to the other).

### **Graph Traversal:?**

* **BFS (Breadth-First Search)**:  
  + Explores the graph **level by level** starting from a source vertex.
  + It visits all neighbors first, then their neighbors, and so on.
  + **Uses a queue** to keep track of nodes to explore.
  + **Application**: Finding the shortest path in unweighted graphs.

#### **Example:**

Start at **A**, visit all neighbors (**B**, **C**), then move on to **B’s** and **C’s** neighbors.

* **DFS (Depth-First Search)**:  
  + Explores a graph by going as **deep as possible** into a branch before backtracking.
  + **Uses a stack or recursion** to manage traversal.
  + **Application**: Detecting cycles or connected components in a graph.

#### **Example:**

Start at **A**, go deep into one branch (e.g., **B** → **C**), then backtrack when needed.

### **Acyclic Graph:?**

* A **graph without cycles** (no path that starts and ends at the same vertex).
* **No loops** are allowed in the graph.

#### **Example:**

**A → B → C** (No way back to A from C).

### **Complete Graph:?**

* Every vertex is **directly connected** to every other vertex.
* **All possible edges** exist in the graph.

#### **Example:**

For 3 vertices (A, B, C), edges: **A-B**, **A-C**, **B-C**.

### **Graph Indexing:?**

* **Graph indexing** refers to assigning **unique identifiers** (indices) to vertices (nodes) in a graph.
* Helps with **efficient storage and retrieval** of graph-related information.

#### **Example:**

Each vertex in a graph is assigned a unique ID to make operations faster.

### **Cycle Detection:?**

* **Cycle detection** is the process of finding if a graph contains **at least one cycle** (a path that starts and ends at the same vertex).
* Important to **avoid infinite loops** or **deadlocks** in various applications.

#### **Example:**

* In DFS or BFS, if a node is revisited that was already in the current path, a **cycle** is detected.

Time and space?

* **Adjacency Matrix:**
  + **Time Complexity**: **O(1)** for checking an edge, **O(V^2)** for traversal.
  + **Space Complexity**: **O(V^2)**.
* **Adjacency List: v means size as increase**
  + **Time Complexity**: **O(V + E)** for traversal.
  + **Space Complexity**: **O(V + E)**.

**1. Spanning Tree**

* A **spanning tree** is a **subset** of a graph that includes all the nodes but just enough edges to connect them.
* It’s like connecting all cities in a country using the fewest roads, without forming any loops.
* **Example:** If you have a graph of 5 cities (nodes), a spanning tree would connect all cities with 4 roads (edges).

### **2. Minimum Spanning Tree (MST)**

* A **minimum spanning tree** is a **spanning tree** where the total weight (or cost) of the edges is as **small as possible**.
* We aim to connect all nodes with the least cost, so if roads have costs, MST would find the cheapest set of roads to connect all cities.
* **Example:** If each road has a cost, MST will choose the roads with the least total cost to connect all cities.

### **3. Prim’s Algorithm**

* **Prim’s algorithm** is a way to find the **Minimum Spanning Tree (MST)**.
* It starts with any node and keeps adding the shortest edge that connects a new node to the existing tree, until all nodes are connected.
* **Example:** If you start in the middle of a map and keep choosing the cheapest road that connects to a new city, that’s Prim’s algorithm.

### **4. Shortest Path Algorithm**

* A **shortest path algorithm** finds the **quickest route** between two nodes in a graph, based on the weights (distances or costs) of the edges.
* **Dijkstra's Algorithm** and **Bellman-Ford Algorithm** are examples.
* **Example:** Finding the shortest route from your home to school, avoiding longer or costlier roads.

### **5. Acyclic Travel**

* **Acyclic travel** refers to a way of moving through a graph or network that **does not form any cycles** (loops). In simple terms, you don’t go in circles.
* **Example:** A one-way street system in a city where there are no loops, so you can't return to where you started by following the roads.

### **6. Topological Sorting**

* **Topological sorting** is the process of arranging the nodes of a **Directed Acyclic Graph (DAG)** in a **linear order**, where each node comes before any nodes it points to.
* It's useful for tasks that need to be done in a specific order, like scheduling jobs or tasks.
* **Example:** In a project, you need to finish task A before task B. Topological sorting helps you figure out the order.

### **7. Graph vs Tree**

* **Graph**: A graph can have cycles (loops), and nodes can be connected by more than one edge. It can be **directed** (edges have directions) or **undirected**.
* **Tree**: A tree is a **special type of graph** where there are no cycles, and it is always connected.
* **Example:** A family tree (tree) vs. a social network (graph).

### **8. Tree vs Trie**

* **Tree**: A tree is a hierarchical structure where each node can have many children, but each node has **one parent** (except the root).
* **Trie**: A **trie** is a **type of tree** used for storing strings, where each node represents a character of the string.
* **Example:** A tree could represent an organization’s structure, while a trie is useful for **word search** (like in auto-complete).

### **9. Degree of a Node in a Graph**

* The **degree** of a node is the **number of edges** connected to that node.  
  + For an **undirected graph**, it’s simply the count of edges.
  + For a **directed graph**, you have **in-degree** (edges coming in) and **out-degree** (edges going out).
* **Example:** In a social network, the degree of a person (node) is the number of friends (edges) they have.