Data Structures and Algorithms

Foundations of Data Structures and Algorithms

Engage and Think

A **shopping website** needs to store and manage thousands of products. At first, finding an item is easy, but as more products are added, **searching takes longer**, and pages load more slowly.

What approaches could help fix this problem?

Learning Objectives

By the end of this lesson, you will be able to:

- Define data structures and data types to manipulate data efficiently
- Understand the practical applications of algorithms to equip you for problem-solving
- Illustrate the categories of data structures to facilitate clear decision-making in programming
- Interpret the time and space complexity of different algorithms to assess their performance

Overview of Data Structures and Algorithms

Data Structures and Algorithms (DSA)

DSA provides a systematic approach to storing, organizing, and processing data to solve problems efficiently.

Just like books are arranged on a shelf with a plan to find a specific one quickly, data structures keep things in order, and algorithms determine the best way to utilize them.

Data Structures and Algorithms (DSA)

It plays a critical role in how modern software is built and functions.
Here is how DSA contributes:

Organizes and manages data using data structures for easy access and modification

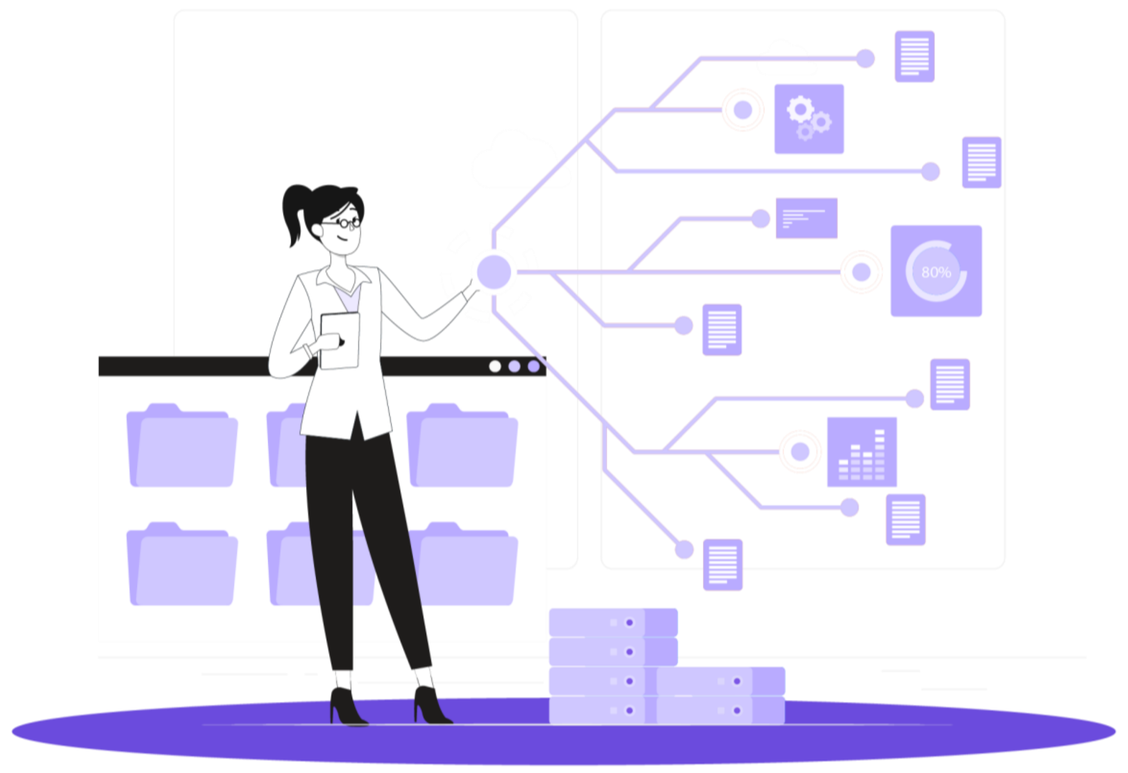
Solves problems efficiently using algorithms that help in making optimal decisions and performing tasks

From social networks to search engines, DSA enables fast, scalable, and intelligent software systems.

Introduction to Data Structures

What Are Data Structures?

They are specialized formats that help organize and manage data in memory, allowing quick and efficient access, modification, and storage.



Choosing the right data structure is key to building scalable, high-performance applications.

Need for Data Structures

Here are some key reasons highlighting the need for data structures:

Optimized operation – Enhancing performance
Example: Google search retrieves results in milliseconds

Memory utilization – Efficient allocation and deallocation
Example: Operating systems effectively manage virtual memory

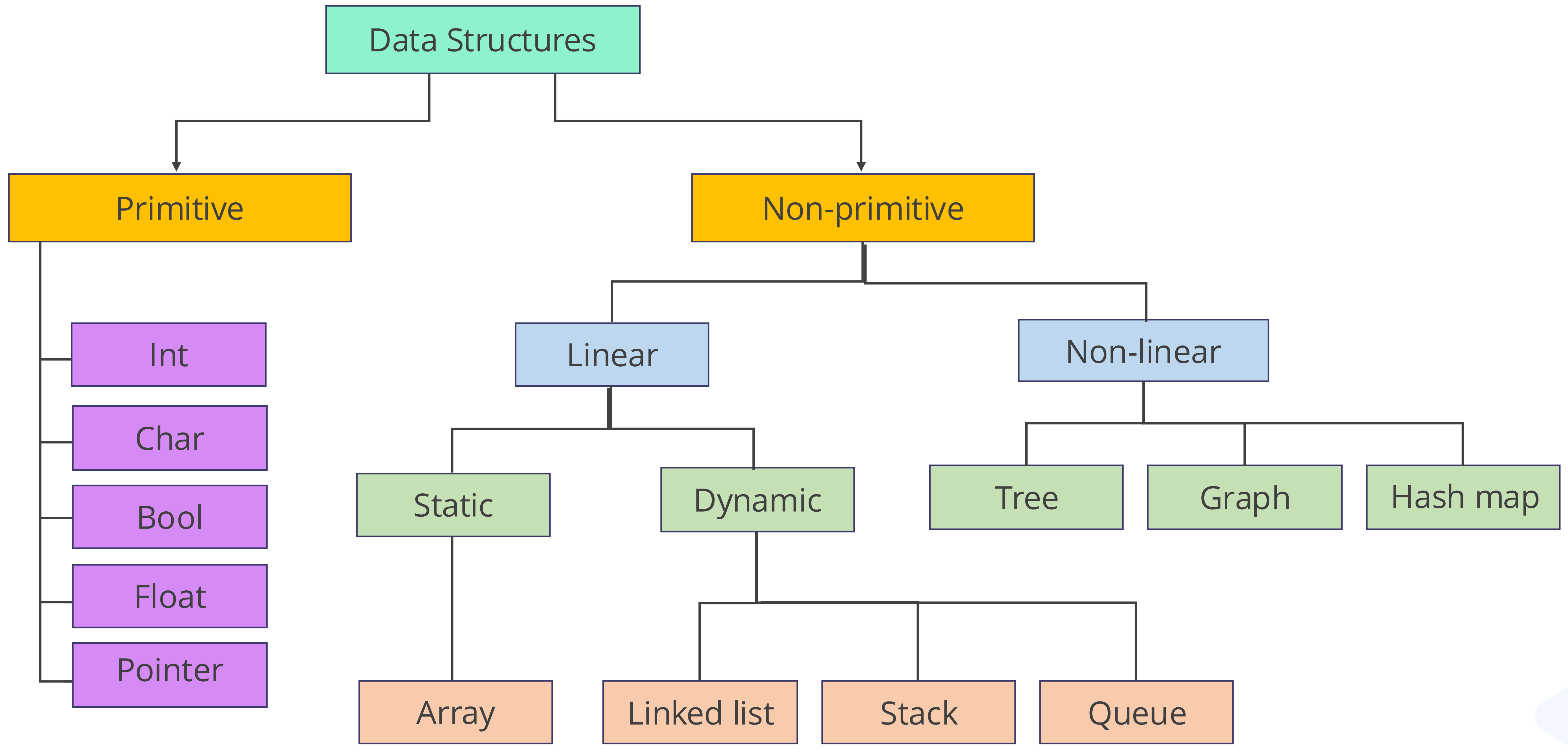
Memory management – Avoiding leaks and optimizing usage
Example: Garbage collection in programming languages

Scalability – Managing large data sets
Example: Social media feeds manage millions of posts smoothly

Reusability – Modular and maintainable code
Example: React Uses trees DS to render UI components

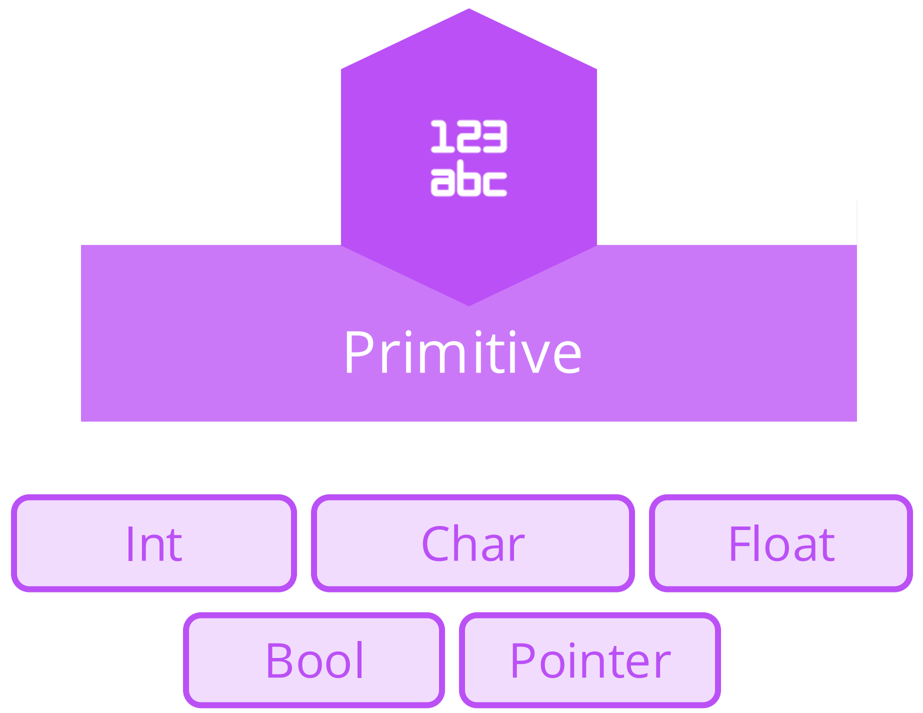
Classification of Data Structures

Data Structures: Categorization



Primitive Data Structures

It is a basic, built-in, or predefined data type that stores simple values, such as integers, characters, strings, or booleans, each holding a single value at a time and remaining unchanged during runtime.

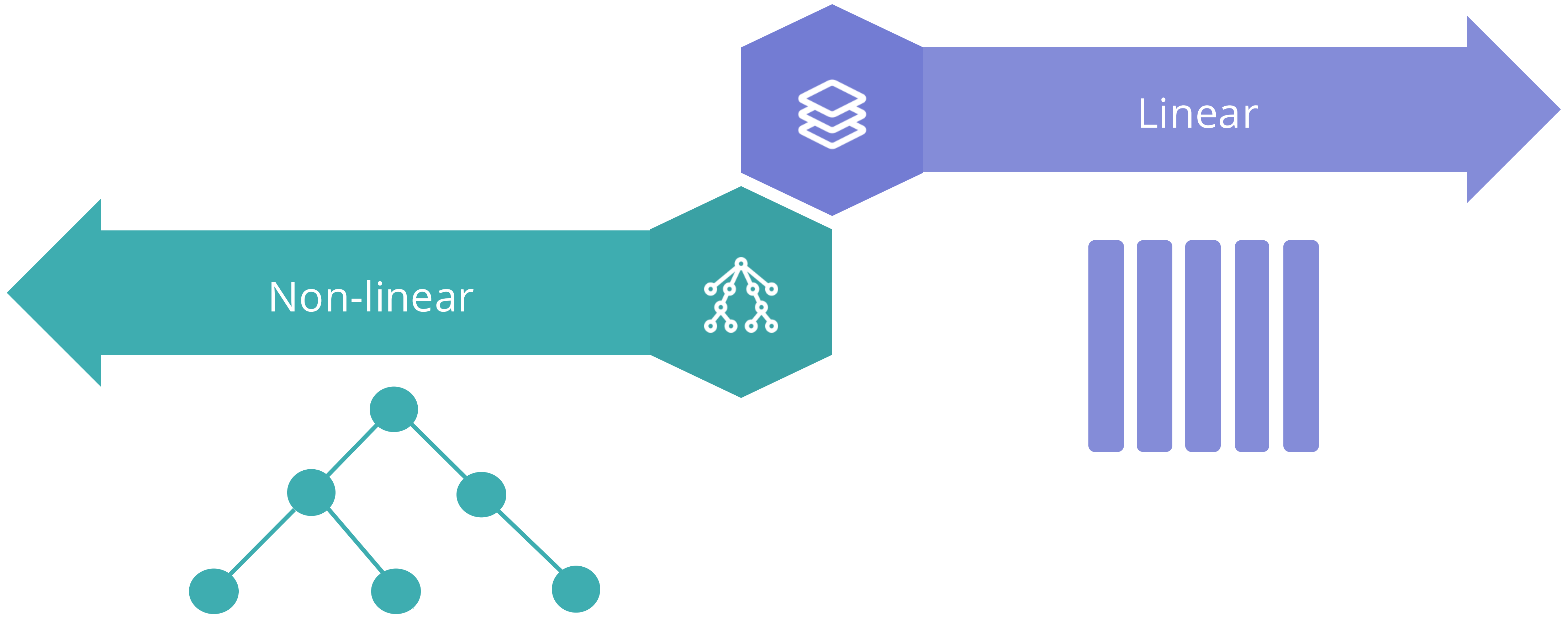


Different programming languages handle these primitive data types differently, but the core concept remains consistent across them.

Non-primitive Data Structures

These are derived from primitive data types and are designed to store multiple values in a structured format, allowing for resizing or modification during runtime.

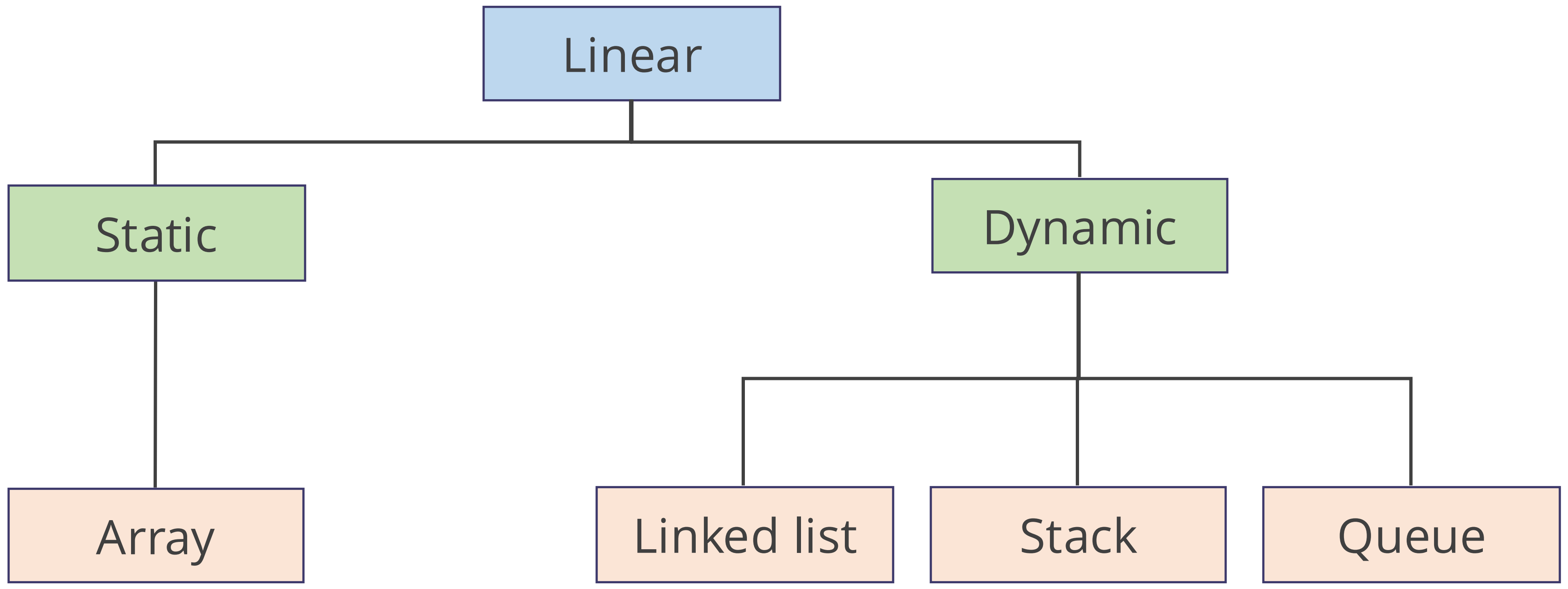
They are broadly classified into two categories based on how they organize and store data:



Understanding Linear Data Structures

What Are Linear Data Structures?

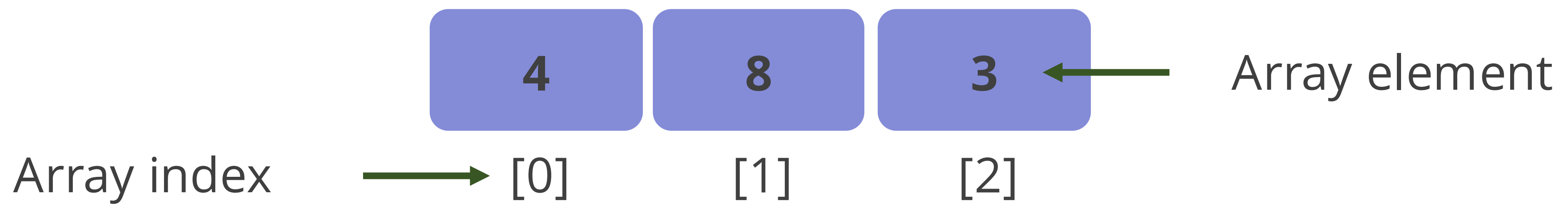
Elements in a linear data structure are arranged in a sequence, where each item is directly connected to the next and previous item. They can be static or dynamic in size.



For example, in to-do list apps, tasks are stored one after another. Behind the scenes, this is often implemented using arrays or linked lists.

Static: Arrays

These are static data structures that store multiple values of the same data type in a fixed-size, contiguous memory location. They allow fast access to elements using index numbers.



What is the value stored at **index [1]**?

Arrays are utilized in various applications, including database storage, image processing, and matrix calculations.

Quick Check

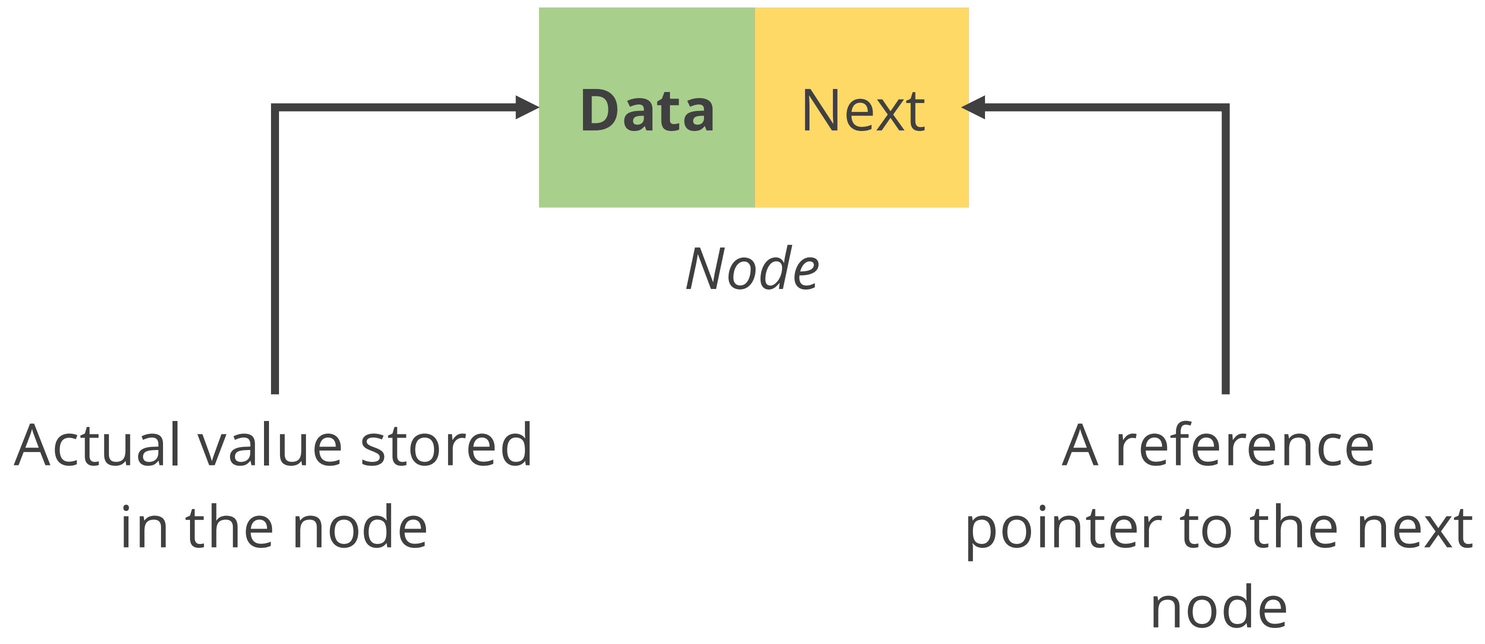
A program requires storing multiple values in a structure where the size is not fixed and can hold different types of data.

Which type of data should be used?

A) Boolean
B) Integer
C) Float
D) Array

Dynamic: Linked List

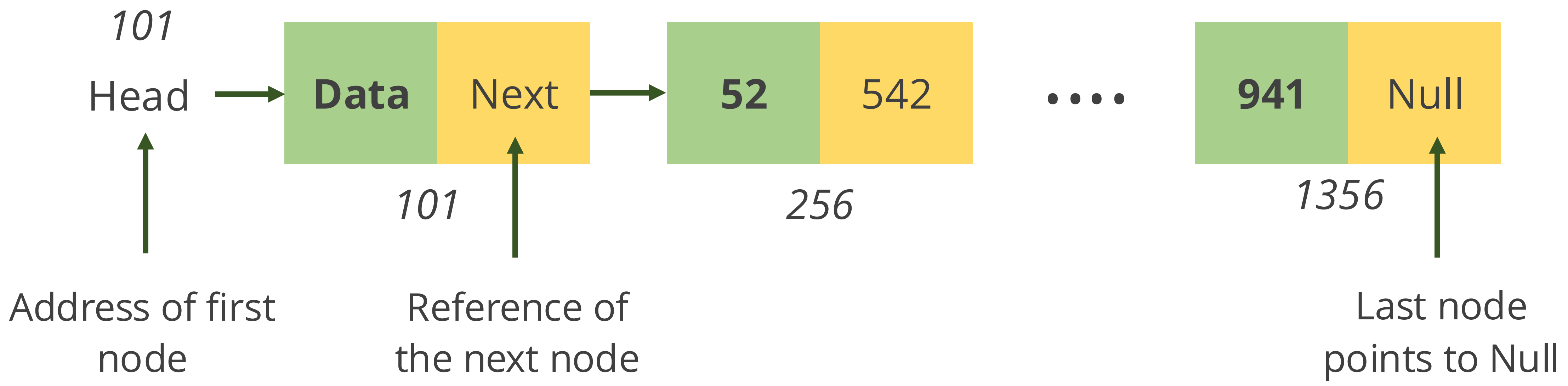
It is a linear data structure in which elements, called nodes, are stored in non-contiguous memory locations. It supports dynamic resizing during runtime.


Data
Next
Node
Actual value stored in the node
A reference pointer to the next node

For example, music playlist apps often use linked lists, each song pointing to the next one in the playlist, allowing for smooth transitions.

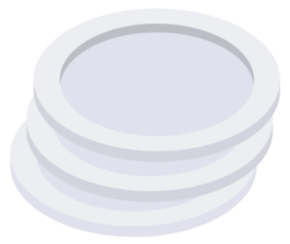
Dynamic: Linked List Example

Here is an example of a linked list with n nodes, starting from the head and ending with the last node pointing to null:

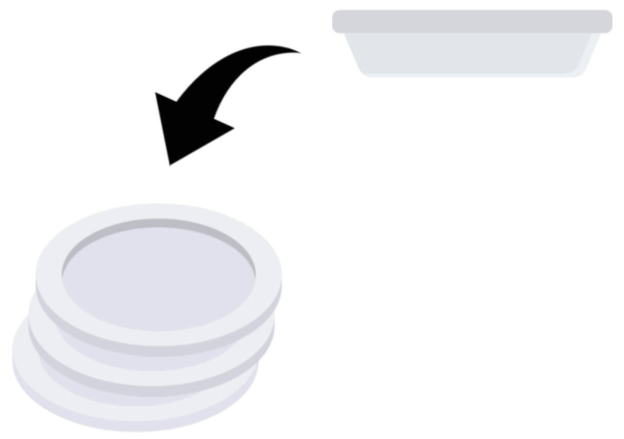


Dynamic: Stack

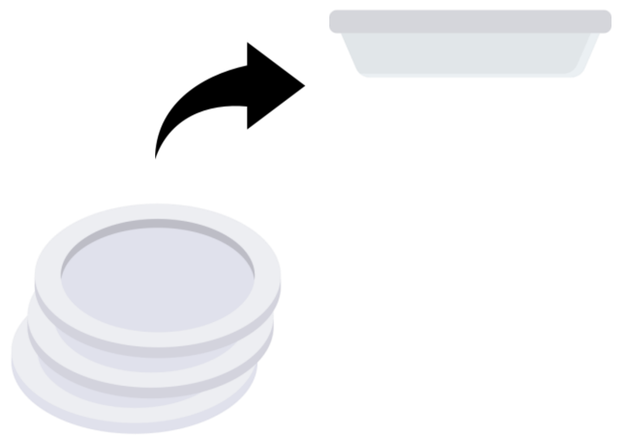
A stack functions similarly to a stack of books or plates. New items are added (pushed) to the top, and the topmost item is removed (popped) first. This follows the Last-In, First-Out (LIFO) principle.



Stack of plates



Adding a plate to the stack

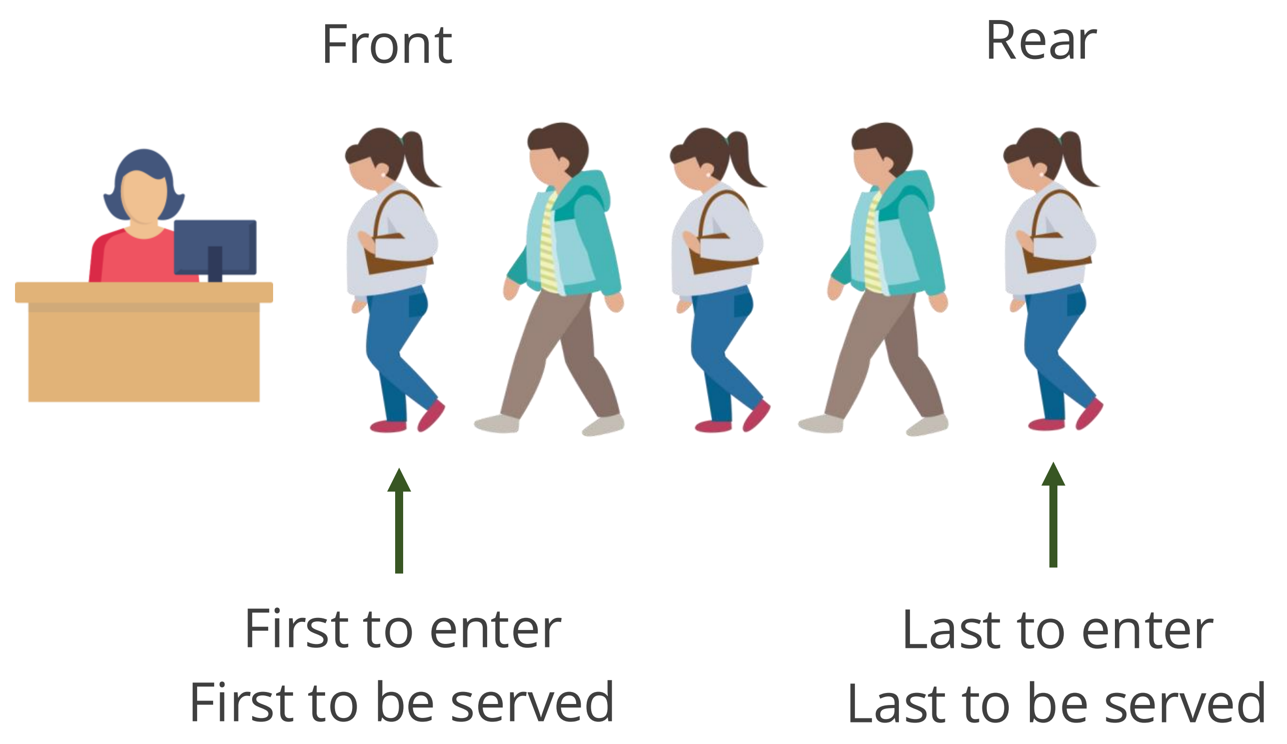


Removing a plate from the stack

For example, the ***Undo button*** in apps uses a stack, and the most recent action is removed first.

Dynamic: Queue

It works like a line of people waiting at a counter. The first person to enter the queue is the first to be served. This principle is known as First-In, First-Out (FIFO).

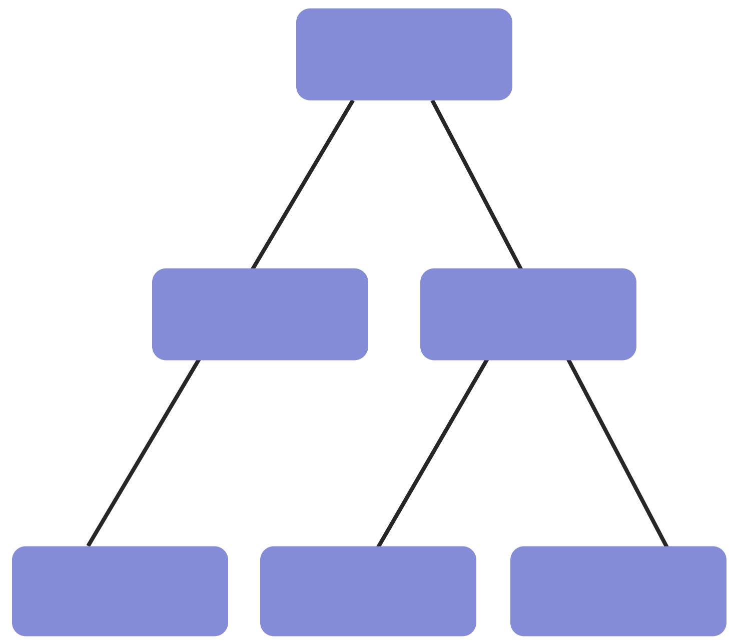


For example, food delivery apps use queues to manage incoming orders. Each new order joins at the end, and restaurants process them in the order they arrived.

Understanding Non-linear Data Structures

Non-linear Data Structures

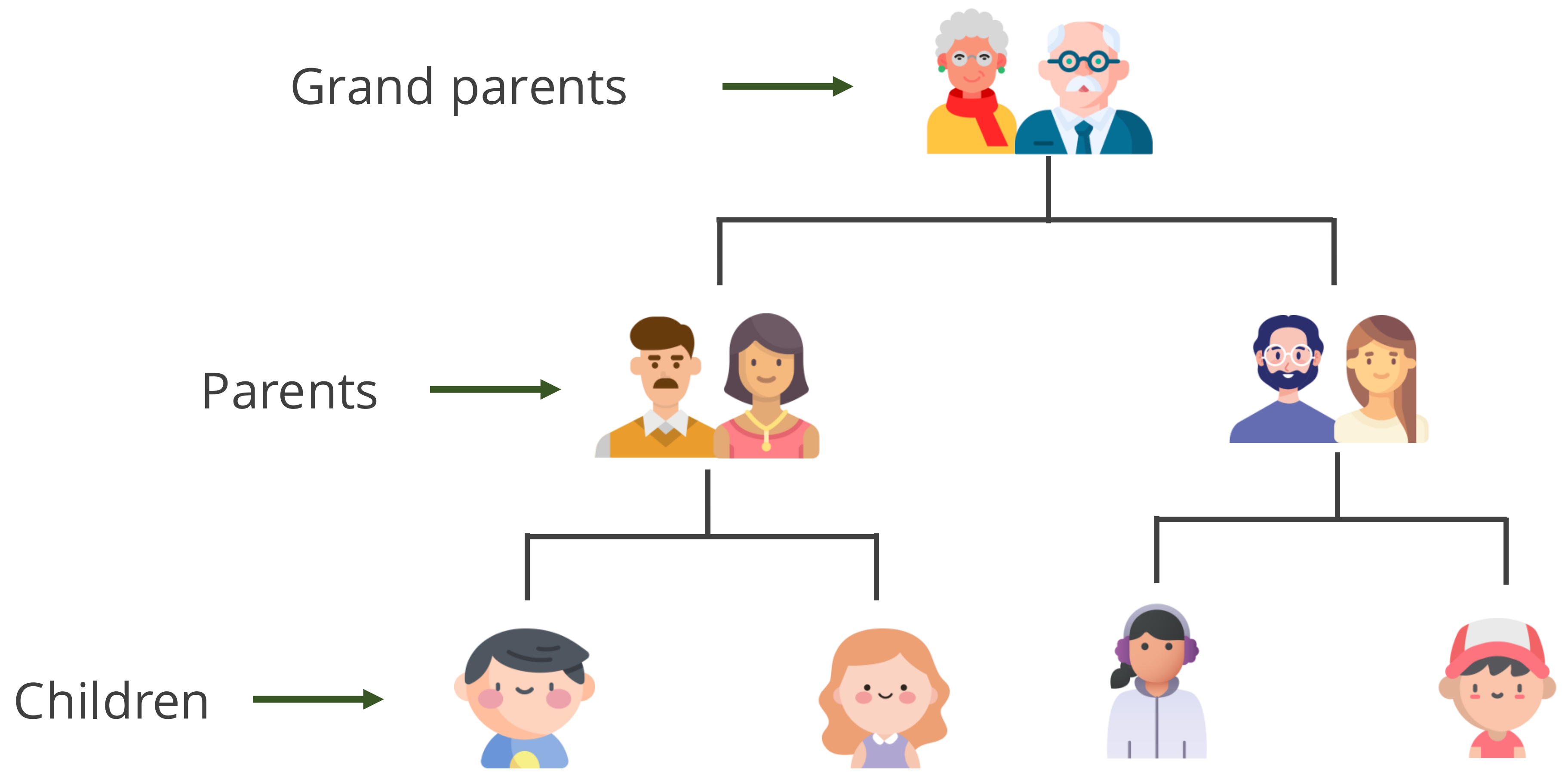
These data structures do not arrange elements sequentially. Instead, they organize elements hierarchically or in a multi-level manner, allowing one element to connect to one or more others.



Non-linear data structures are classified into two types: graphs and trees.

Non-linear Data Structures: Tree

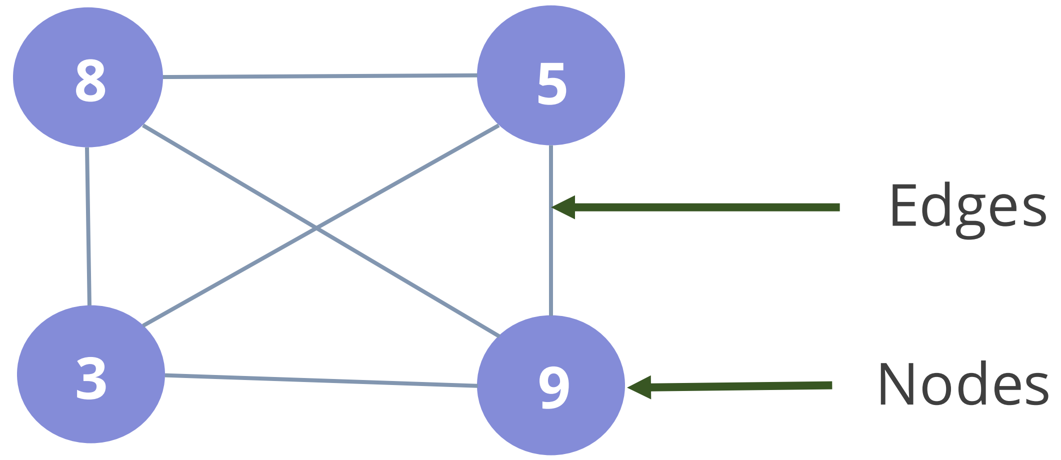
It works like a family tree where each ancestor branches out into multiple descendants. It starts from a root and spreads across different generations.



For example, a computer's file system uses a tree structure. The root folder contains subfolders that further branch into smaller files or folders.

Non-linear Data Structures: Graph

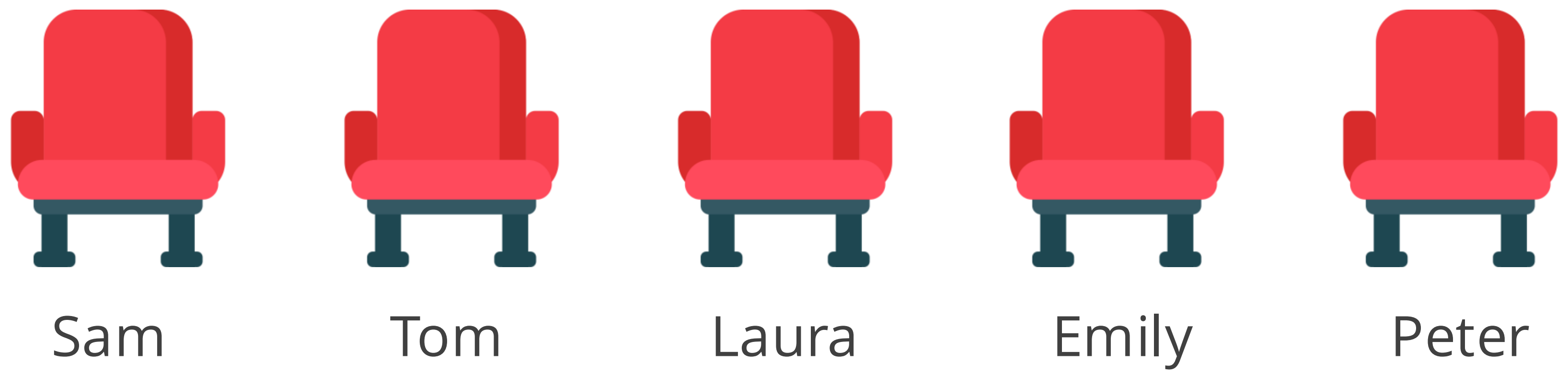
It is a non-linear data structure made of nodes and edges. Nodes (vertices) represent entities, and edges show how they are connected.



For example, **Google Maps** utilizes graphs, where locations are represented as nodes, and paths are represented as edges.

Non-linear Data Structures: HashMap

It works like assigning seats in a theater where each seat number (key) corresponds to a specific person (value). You can quickly find who is sitting in a particular seat without checking the entire row.



For example, a contact list works like a HashMap. A person’s name (key) maps directly to their phone number (value), making retrieval instant.

Comparison of Linear and Non-linear

	Linear	Non-linear
Arrangement	Data elements are arranged sequentially.	Data elements have a hierarchical structure.
Traversal	Can traverse it in a single run	Need multiple runs to traverse
Complexity of time	Simpler in structure and organization	More complex in structure, allowing representation of intricate relationships
Application	Suitable for scenarios where elements follow a specific order or priority	Suitable for scenarios where relationships and connections between elements are crucial

Applications of Data Structures

Data structures help software work faster, more innovatively, and more efficiently behind the scenes.

Searching on Google:

Trees enable the fast retrieval of relevant web results.

Amazon tracks recent views:

Stacks and queues manage and restore recently viewed items.

Friend suggestions on Facebook:

Graphs map user connections and suggest mutual friends.

WhatsApp loads recent chats instantly:

Queues and linked lists manage message order and load recent chats.

Quick Check

A logistics company manages a warehouse network connecting cities. The system must:

- Find the shortest delivery route between cities
- Track deliveries in order as trucks depart

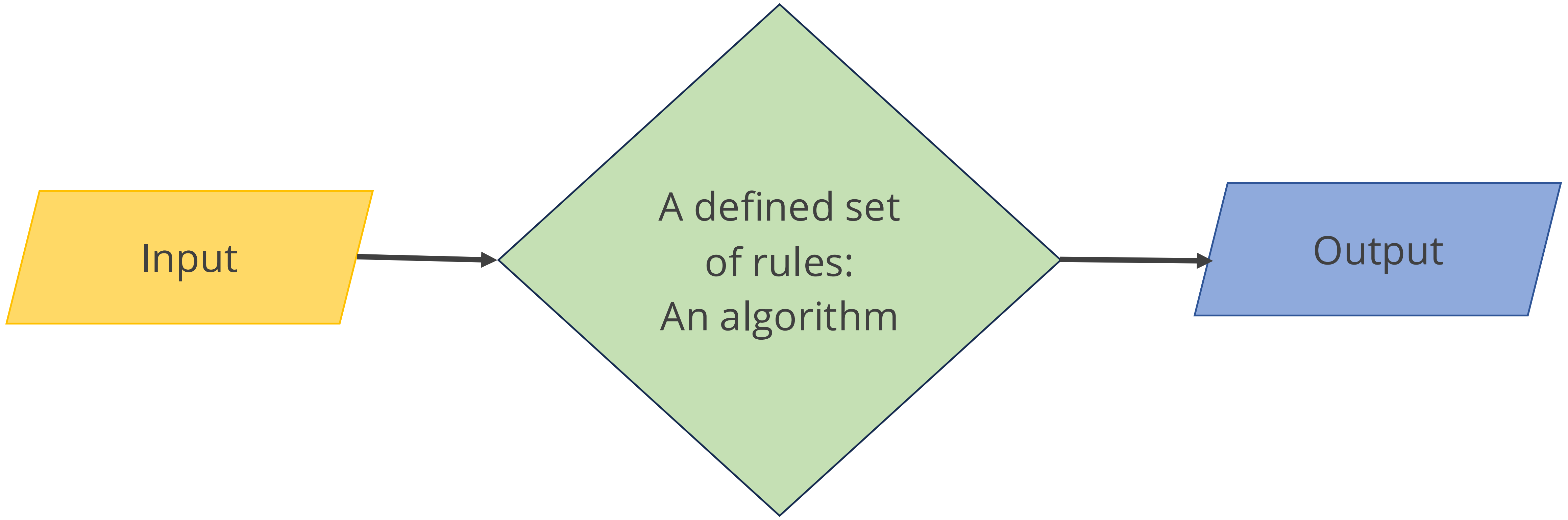
Which combination of data structures will best support this?

A) Array and queue
B) Graph and queue
C) Tree and queue
D) Graph and linked list

Introduction to Algorithms

What Is an Algorithm?

It is a sequence of well-defined instructions that systematically solve a problem or complete a task. It processes input and produces the desired output efficiently.



For example, a spell-check algorithm solves the problem of detecting and correcting typing errors in documents and messages.

Why Do We Use Algorithms?

Scenario: Two friends, Alice and Ben, want to choose a book from a large library.

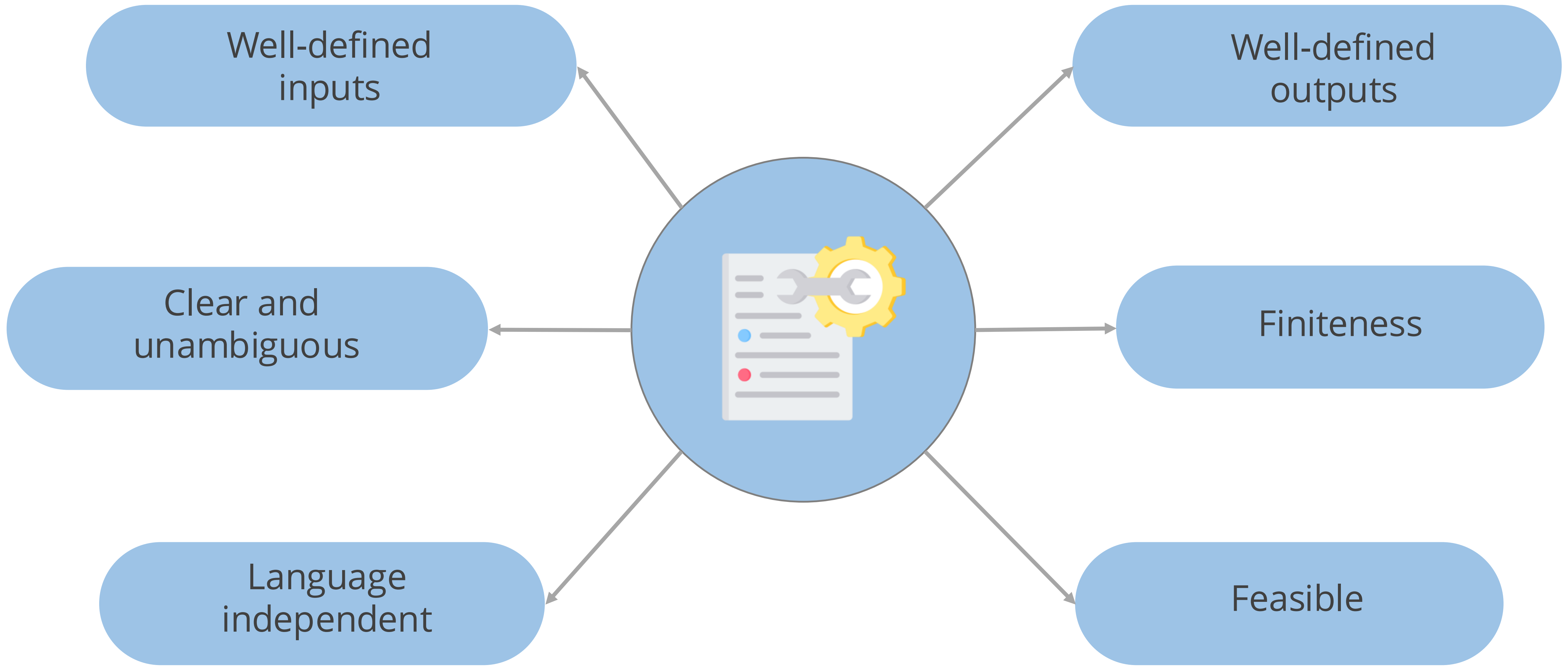
- Alice randomly browses shelves and picks books without a clear process.
- She spends hours deciding and still feels unsure.

- Ben follows a step-by-step process: filter genres, check top-rated, and read synopses.
- He quickly picks a book he enjoys

Algorithms simplify decision-making by following structured steps and reducing uncertainty.

Characteristics of Algorithms

Not every set of programming instructions qualifies as an algorithm. Only those that meet specific characteristics can be considered an algorithm. Here are the key characteristics:



Types of Algorithms

Brute Force	Tries all possible solutions until the correct one is found. Example: Checking all possible combinations in a password cracker.
Searching	Finds the position of an element in a dataset. Example: Binary Search, Linear Search.
Sorting	Arranges elements in a particular order (ascending or descending). Example: Bubble Sort, Merge Sort, Quick Sort.
Divide and conquer	Breaks the problem into smaller sub-problems, solves each recursively, and combines results. Example: Merge Sort, Quick Sort.

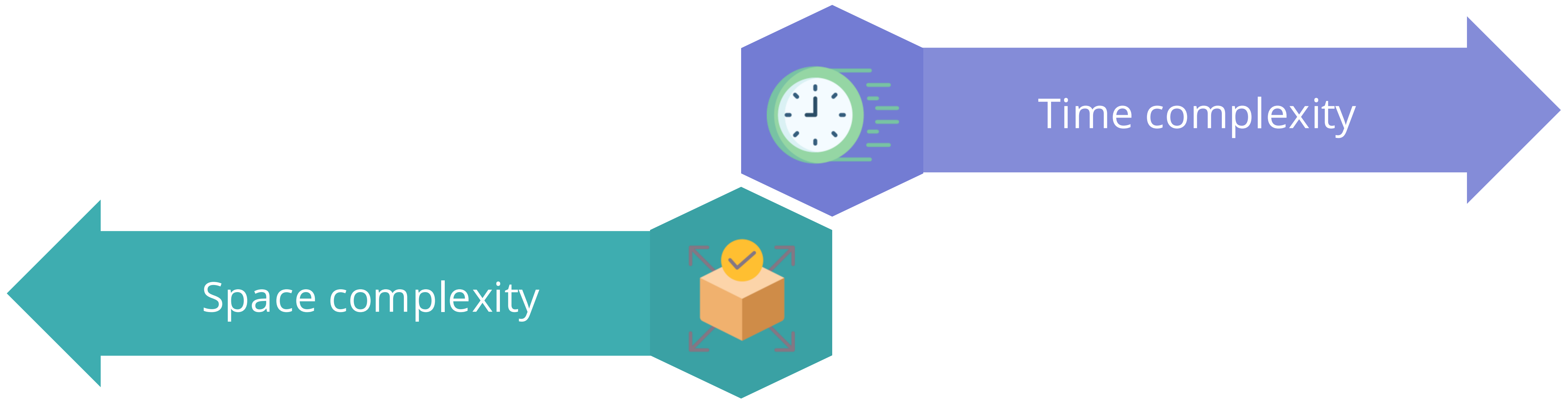
Types of Algorithms

Backtracking	Tries different possible solutions and abandons paths that lead to failure. Example: N-Queens problem, Sudoku solver.
Greedy	Makes the best choice at each step to achieve an optimal global solution. Example: Dijkstra’s algorithm for shortest path.
Dynamic programming	Solves problems by breaking them down into overlapping sub-problems and storing results to avoid re-computation. Example: Fibonacci sequence, Knapsack problem.

Introduction to Algorithm Complexities

Complexities of an Algorithm

It measures how efficiently an algorithm uses resources like time and memory during execution.



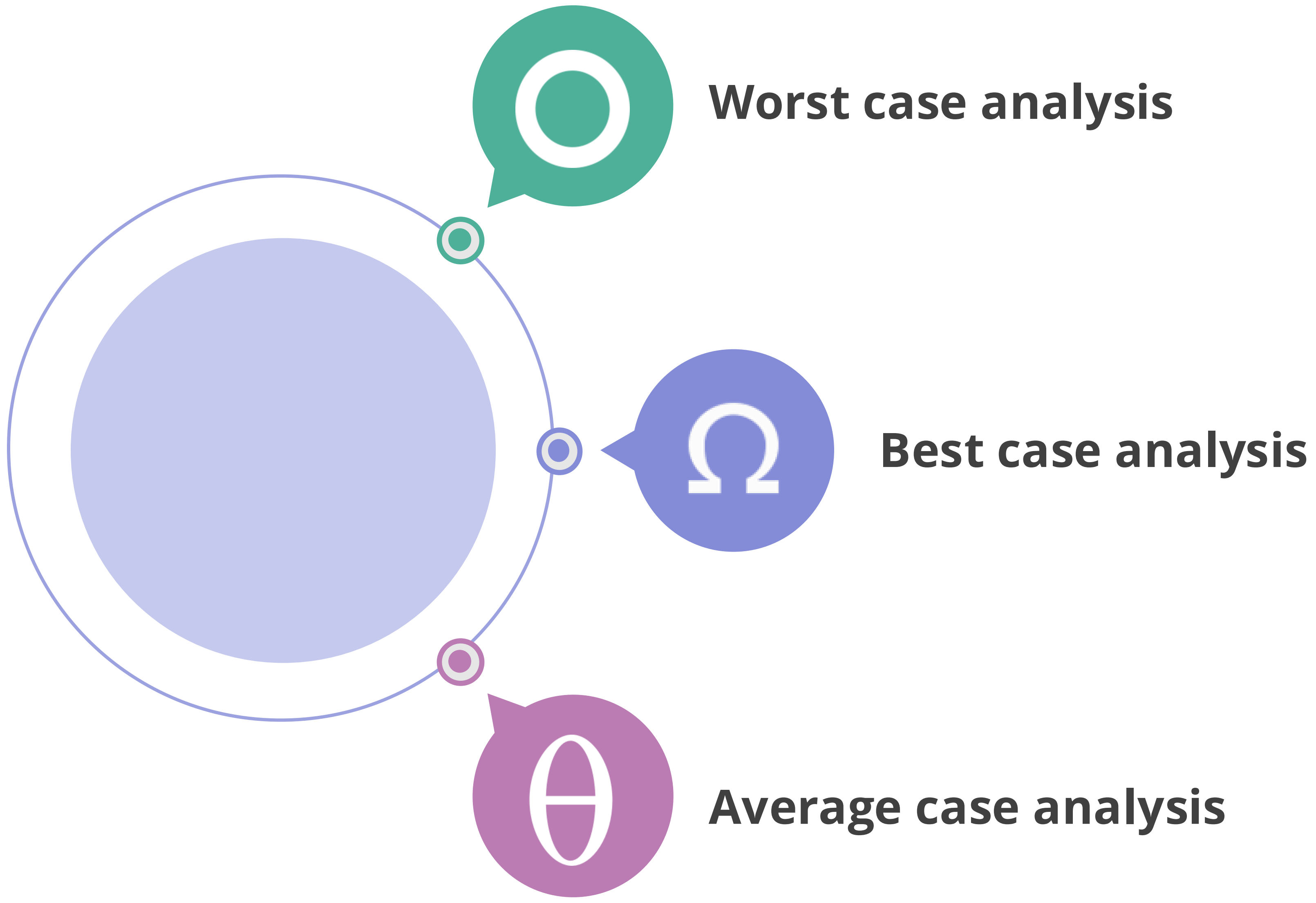
Measuring complexity helps in choosing the most optimal algorithm for a task.

Time Complexity

It describes how quickly or slowly an algorithm executes as the input size increases. It helps in comparing the efficiency of different solutions.

Analysis of Algorithms

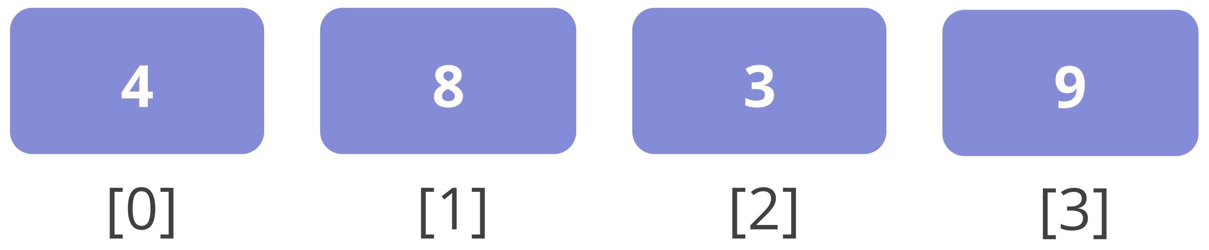
When analyzing algorithms, three different scenarios are commonly considered to evaluate their efficiency:



Worst Case Analysis

It represents the maximum time an algorithm can take for the most unfavorable input.

For example, when searching for an element, if it is absent or last, the search takes the longest time.

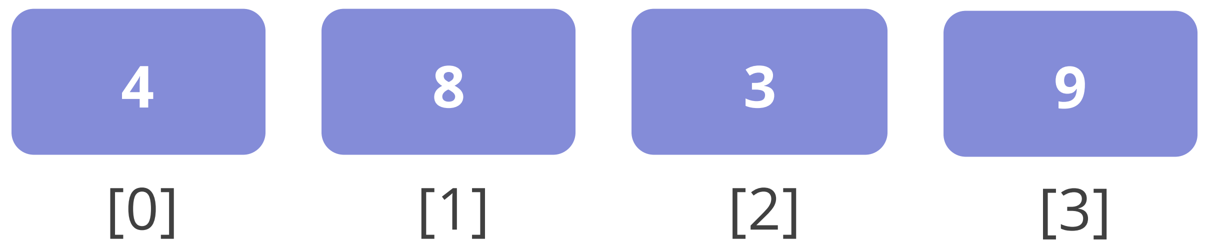


*Searching **17** (absent) or **9** (last) requires checking all **4 elements**.*

Best Case Analysis

It represents the fastest time an algorithm takes, when the input is in the most favorable position.

For example, if the number is present at the beginning, the search takes the least time.

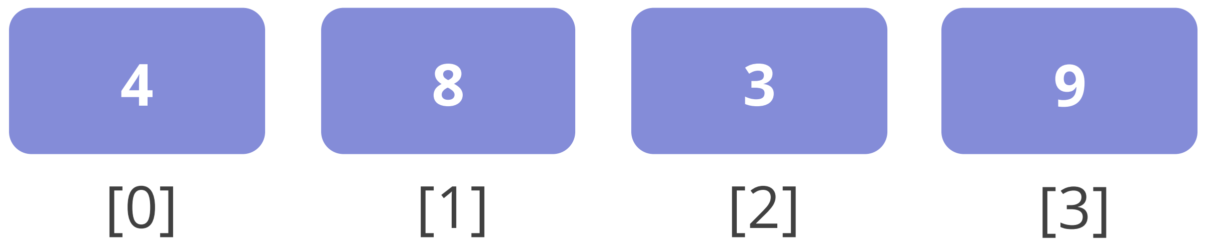


*Searching **4** (first position)*
*completes in **1 comparison**.*

Average Case Analysis

It shows the expected time an algorithm takes, assuming the input could be in any random position.

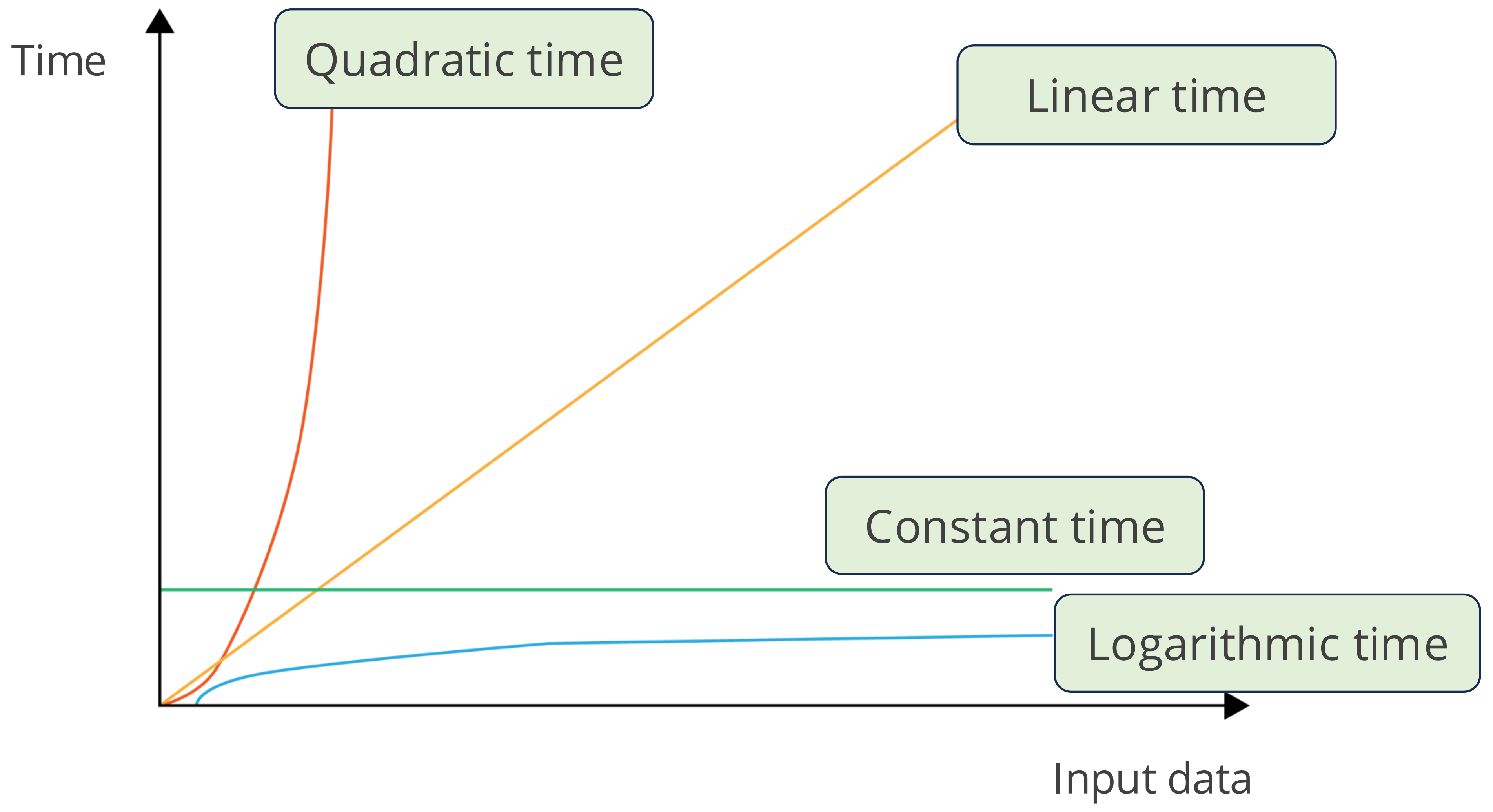
For example, if we search for a number that exists in the list, it could be at any random position.



*Searching for **8** takes **2 comparison**, whereas searching **3** takes **3 comparisons**.*

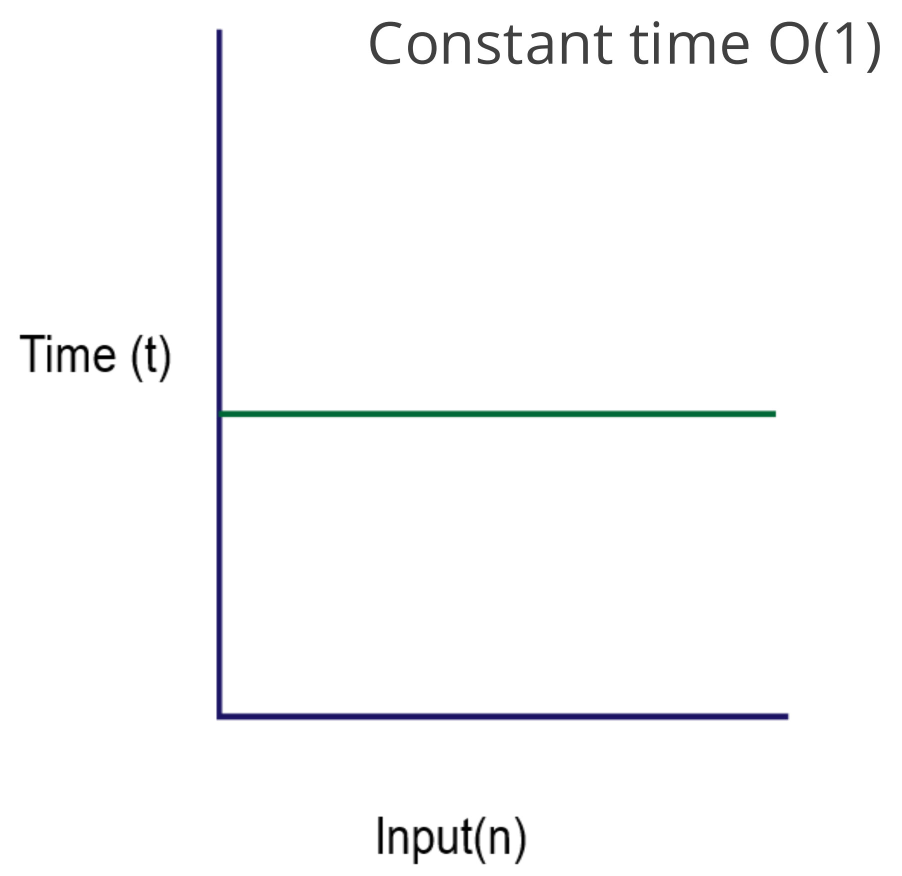
Types of Time Complexity

Different types of time complexity help compare how efficiently algorithms perform as input size increases.



Constant Time Complexity

It is often denoted as O(1), which means that the execution time of an algorithm remains constant, regardless of the size of the input data.



It is best for operations that always take a fixed number of steps.

Accessing an element in an array by its index takes constant time.

Constant Time Complexity: Calculation

Some examples of constant time complexity in JavaScript are:

Accessing element in an array

```
const myArray = [1, 2, 3, 4, 5];
const element = myArray[2];
// Accessing element at index 2

// O(1) operation: Direct access by index
```

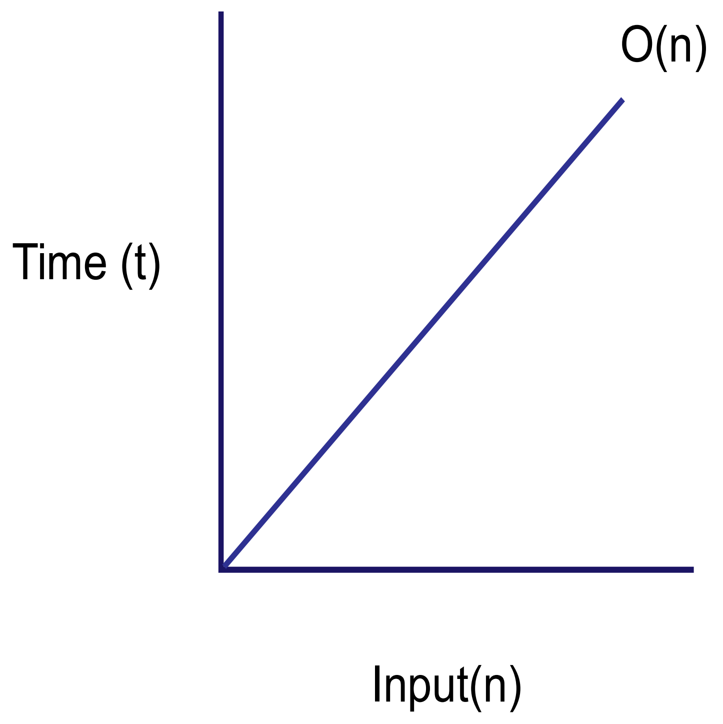
Addition of two elements

```
const a = 5;
const b = 10;
const sum = a + b;  // O(1) operation
```

Linear Time Complexity

It refers to algorithms whose execution time grows linearly with the size of the input data.

Linear time O(n)



If n = 10, max 10 checks

If n = 100, max 100 checks

O(n) complexity means performance drops as input size increases.

Linear Time Complexity: Calculation

Calculate the linear time complexity for:

Accessing array elements

```
const myArray = [1, 2, 3, 4, 5];

for (let i = 0; i < myArray.length; i++)
{
  // Linear time complexity operation for
each element
  console.log(myArray[i]);
}
```

The loop runs n times
(where n is the size of myArray).

Each operation inside the loop takes
constant time → O(1).

Since the loop executes n times, the total
complexity is n * O(1) → O(n).

Linear Time Complexity: Calculation

Calculate the linear time complexity for:

Linear search

```
function linearSearch(array, target) {
  for (let i = 0; i < n; i++) {
    if (array[i] === target) {
      return i; // Element found
    }
  }
  return -1; // Element not found
}
```

The loop checks each element one by one until it finds the target or reaches the end.

In the best case, the target is found at the first position → O(1).

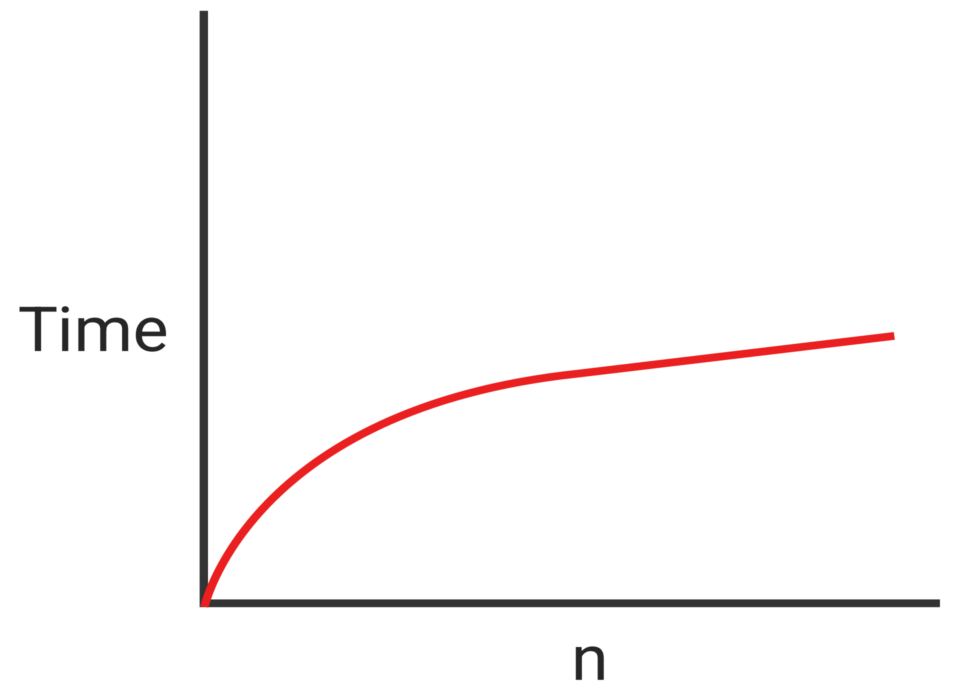
In the worst case, the target is at the last position or not present and requires n searches.

Therefore, the time complexity is n * O(1) → O(n).

Logarithmic Time Complexity

It means the algorithm reduces the problem size at each step, so the time taken grows very slowly even as the input size increases.

Logarithmic time O(log N)



If N = 10, the algorithm may perform $\log_2(10) \approx 3.3$ operations.

If N = 1,000,000, it may perform $\log_2(1,000,000) \approx 20$ operations.

O(log N) algorithms scale efficiently, handling large datasets with minimal performance degradation.

Logarithmic Time Complexity: Calculation

Logarithmic time complexity using binary search:

```
function binarySearch(sortedArray, target) {
  let low = 0;
  let n = sortedArray.length;
  let high = n - 1;

  while (low <= high) {
    const mid = Math.floor((low + high) / 2);
    if (sortedArray[mid] === target) {
      return mid; // Element found
    } else if (sortedArray[mid] < target) {
      low = mid + 1; // Search in the right half
    } else {
      high = mid - 1; // Search in the left half
    }
  }

  return -1; // Element not found
```

It starts with n elements.

Each step cuts the search space in half→ from n to n/2, then n/4, and so on.

This continues until only 1 element remains→ So, $n \div 2^k = 1$.

Taking $\log_2$ on both sides gives: $k = \log_2 (n)$.

This results in $O(\log n)$ time complexity.

Note: k represents the number of steps (iterations) the algorithm takes to reduce the input size from n to 1.

Quick Check

Two different algorithms solve the same problem. Algorithm A runs in O(N) time, and Algorithm B runs in O(log N) time.

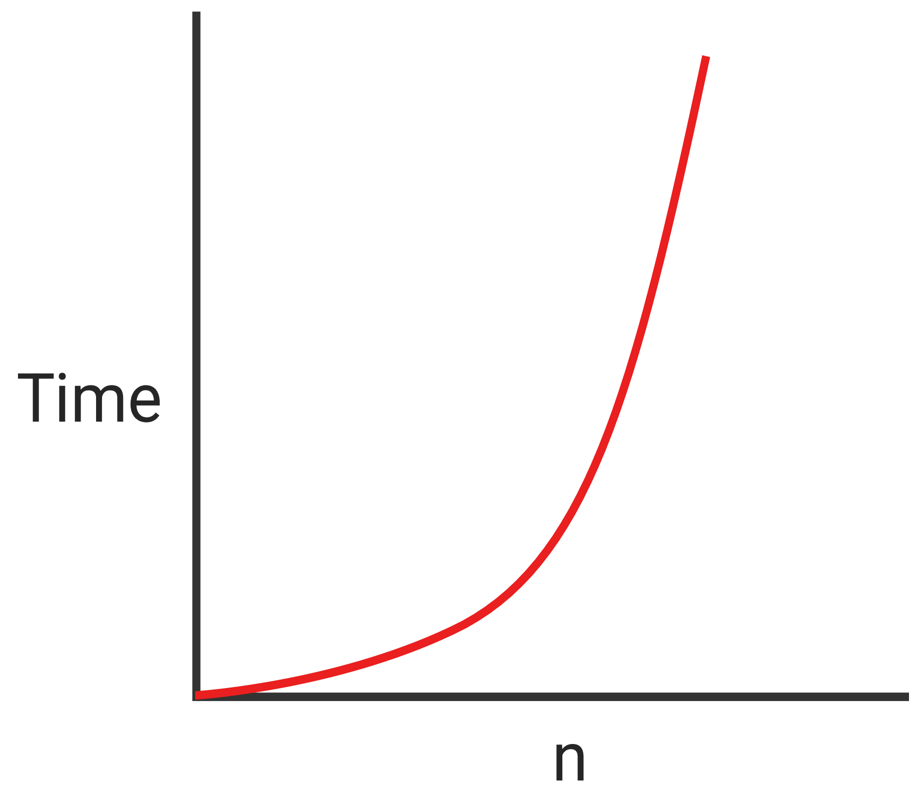
As the input size grows significantly, what is likely to happen?

A) Algorithm A will perform better for all inputs.
B) Algorithm B will scale better because its execution time grows much slower.
C) Both algorithms will always take the same time to execute.
D) Algorithm A will always be faster for smaller inputs.

Quadratic Time Complexity

It means the execution time of an algorithm grows proportionally to the square of the input size (N). As the input increases, the number of operations increases quadratically.

Quadratic time O(N^2)



If N = 10, the algorithm may perform $10^2 = 100$ operations.

If N = 100, it may perform $100^2 = 10,000$ operations.

When an algorithm uses **nested loops**, it often results in **O(n^2) complexity**.

Quadratic Time Complexity: Calculation

Quadratic time complexity using counting pairs in an array:

```
function countPairs(arr) {
    let count = 0;
    for (let i = 0; i < n; i++) {
        for (let j = i + 1; j < n; j++) {
            count++; // Counting each pair
        }
    }
    return count;
}
```

Outer loop runs n times.

Inner loop runs (n - i) times for each i.

Total operations = sum of 1 to (n - 1) → n(n - 1)/2

Ignoring constants, this simplifies to $O(n^2)$.

Quadratic Time Complexity: Calculation

Quadratic time complexity using nested loop:

```
function quadraticExample(array) {
  for (let i = 0; i< n; i++) {
    for (let j = 0; j < n; j++) {
      // Quadratic time complexity operation
      console.log(array[i], array[j]);
    }
  }
}
```

Outer loop runs n times (from i = 0 to i < n).

For each iteration of i, the inner loop also runs n times.

So, total operations = $n \times n = n^2$

Ignoring constants, this results in a time complexity of $O(n^2)$.

Quick Check

A developer notices that an algorithm, which previously ran fine with 1,000 inputs, suddenly becomes extremely slow when processing 10,000 inputs.

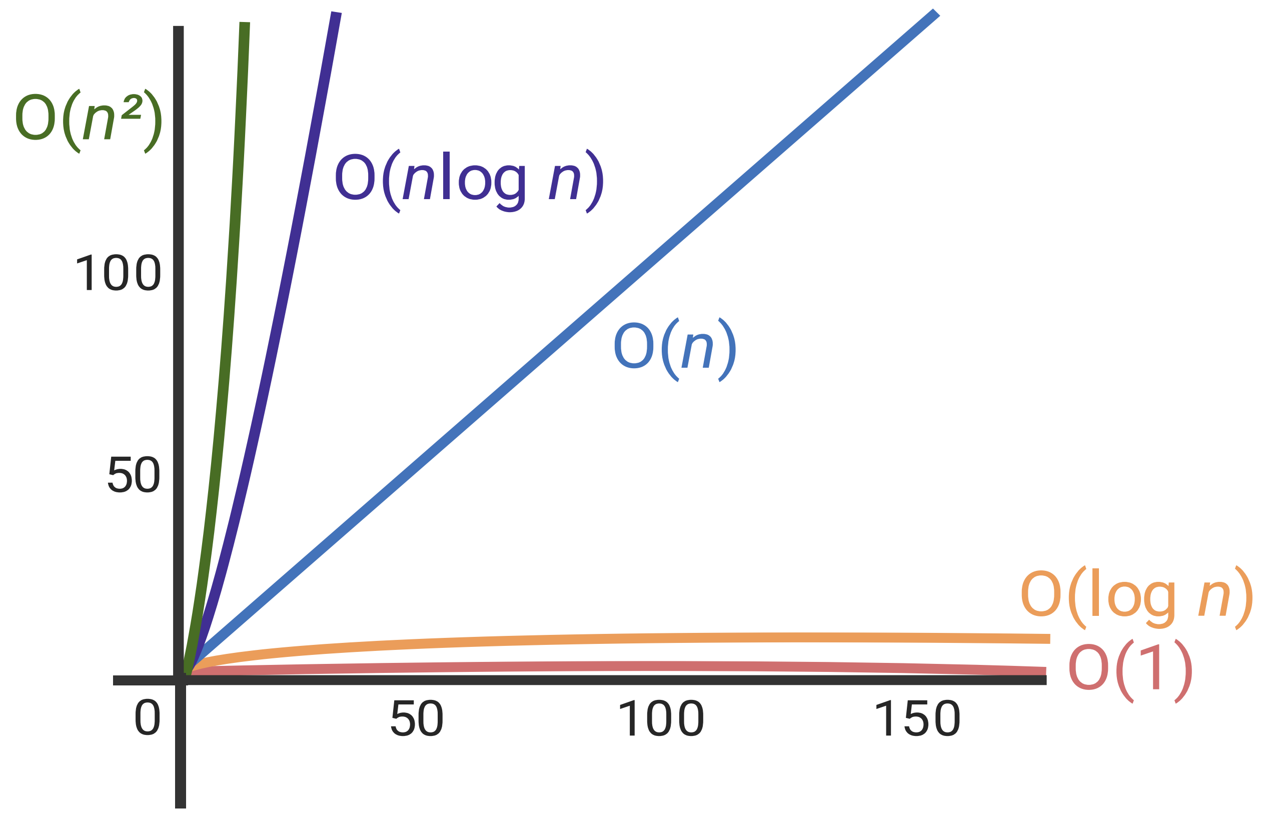
What could be the most likely reason?

A) The algorithm's time complexity is higher than O(N), possibly $O(N^2)$ or worse.
B) The algorithm is running in constant time O(1), so input size should not matter.
C) The programming language used is inefficient.
D) The computer's memory is full.

Space Complexity of an Algorithm

Space Complexity

It refers to the amount of memory an algorithm requires to run. It includes the space used by variables and data structures.



Types of Space Complexity

Constant space O(1)	Algorithms with constant space complexity use a fixed amount of memory, regardless of the input size.
Linear space O(n)	It indicates that the amount of memory used by the algorithm grows linearly with the size of the input.
Logarithmic space O(log n)	Logarithmic space complexity increases their memory usage by a logarithmic factor as the input size grows.

Types of Space Complexity

Polynomial space $O(n^k)$	The memory grows as a power of the input size, for example, $O(n^2)$ or $O(n^3)$, depending on the algorithm.
Exponential space $O(2^n)$	The memory used by the algorithm doubles (or grows exponentially) with each additional element in the input.
Quasilinear space $O(n \log n)$	Quasilinear space complexity is a combination of linear and logarithmic growth.

Space Complexity: Calculation

Here is an example to calculate linear space complexity in JavaScript are:

```
function exampleArray(n) {
  let arr = []; // O(1) space for the array
reference
  for (let i = 0; i < n; i++) {
    arr.push(i); // O(n) space for the
elements in the array
  }
}
```

arr is initialized once → takes O(1) space for reference.

n elements are pushed into the array→ takes O(n) space to store those elements.

No extra space is used beyond this array.

Total space complexity is O(n).

Space Complexity: Calculation

Some examples to calculate space complexity in JavaScript are:

```
function exampleDataStructure(n) {
  let stack = []; // O(1) space for the stack
reference
  for (let i = 0; i < n; i++) {
    stack.push(i); // O(n) space for the
elements in the stack
  }
}
```

Stack is initialized once → O(1) space for the reference.

n elements are pushed → O(n) space to store the elements.

No extra structures or memory are used.

Total space complexity is O(n).

Space Complexity: Calculation

Some examples to calculate space complexity in JavaScript are:

```
function generateSubsets(arr) {
  if (arr.length === 0) {
    return [[]]; // Base case: 1 subset
(empty set)
  }

  let firstElement = arr[0];
  let restSubsets =
generateSubsets(arr.slice(1)); // Recursive
call on rest

  let newSubsets = restSubsets.map(subset =>
[firstElement, ...subset]);

  return [...restSubsets, ...newSubsets];
}
```

Empty set returns [[]] → O(1) space

Add element 2 → [[], [2]] → O(2) space

Add element 1 → [[], [2], [1], [1, 2]] → $O(2^2)$ space

Subsets double with each element → O(2^n)

Key Takeaways

- A data structure is a way of organizing and storing data to perform operations efficiently.
- Data structures are broadly categorized into primitive and non-primitive.
- An algorithm is a finite set of well-defined instructions that systematically solves a problem or completes a task.
- Time complexity is a measure of the amount of time an algorithm takes to complete based on the input size.
- Space complexity is a measure of the amount of memory an algorithm requires relative to the input size.

Thank You