التكنولوجيا التطبيقية
APPLIED TECHNOLOGY
وزارة التربية والتعليم والتعليم الفني

MINISTRY OF EDUCATION AND TECHNICAL EDUCATION

مدارس we للتكنولوجيا التطبيقية

Data structure and Algorithm
Unit 14
الصف الثاني

IT Department
Web Programming

Data structure and Algorithm
STUDENT GUIDE

2st

2024 - 2025

التكنولوجيا التطبيقية
APPLIED TECHNOLOGY
وزارة التربية والتعليم والتعليم الفني

Data structure and Algorithm
Unit 14
الصف الثاني

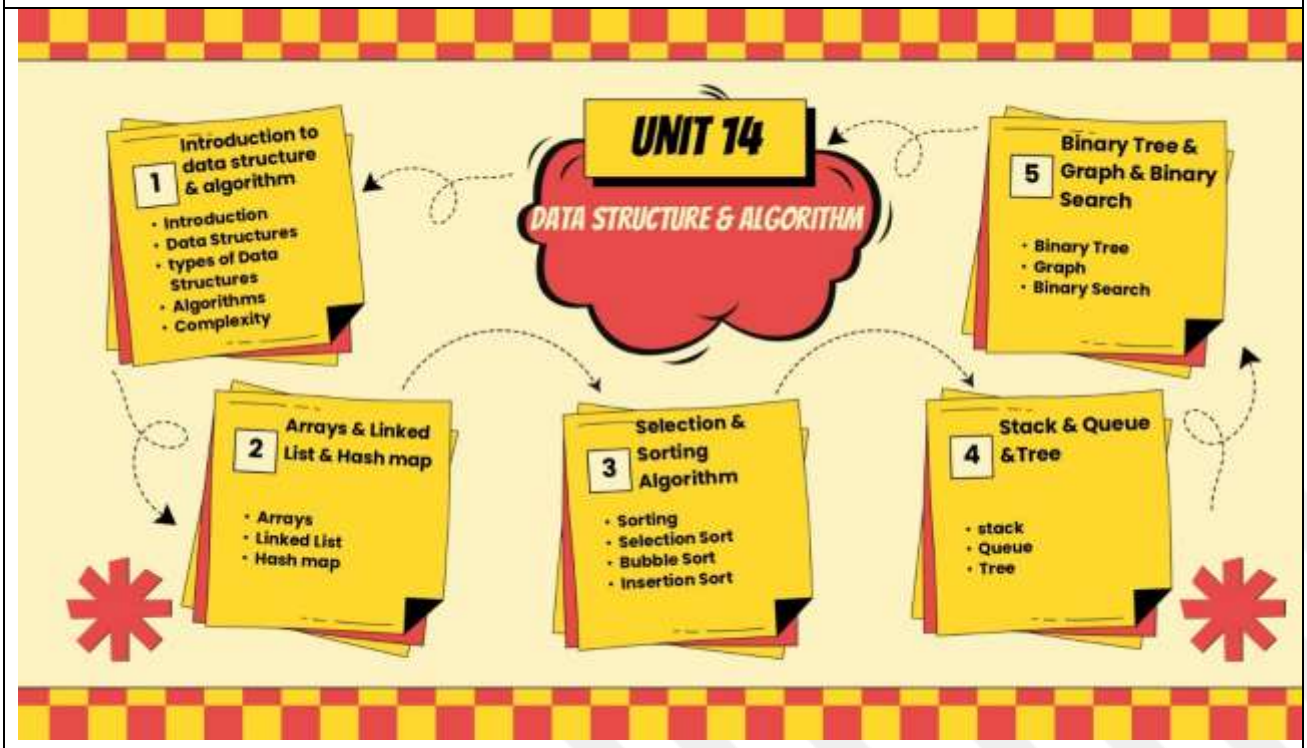
Unit	14
Name	Data structures and algorithms
Goals / Outcomes	➢ **Remembering** 1. Define key concepts in data structures and algorithms, such as arrays, linked lists, hash maps, stacks, queues, trees, graphs, and sorting algorithms. 2. Identify different types of data structures and their purposes. 3. Recall basic sorting techniques like selection sort and bubble sort. 4. Memorize the key operations (e.g., insertion, deletion, traversal) for various data structures. ➢ **Understanding** 1. Explain the role of data structures in organizing and managing data efficiently. 2. Describe the differences and use cases for arrays, linked lists, and hash maps. 3. Understand the logic behind sorting algorithms and how they reorder elements. 4. Illustrate the concept of stacks and queues and their real-life applications. 5. Interpret how binary trees and graphs are structured and how binary search operates in sorted data. ➢ **Applying** 1. Implement arrays, linked lists, hash maps, stacks, queues, and trees in code. 2. Write and execute sorting algorithms like bubble sort and insertion sort. 3. Develop programs to search for data efficiently using binary search. 4. Apply graphs to solve problems such as finding connections or shortest paths. 5. Use data structures in solving practical problems, like organizing data in a program. ➢ **Analyzing** 1. Compare the efficiency of different data structures for specific tasks (e.g., when to use a linked list versus an array).

<table>
<tr><td></td><td colspan="2">2. Analyze the performance of sorting algorithms based on time complexity.
3. Evaluate the suitability of binary trees versus graphs for representing hierarchical versus networked data.
4. Break down the steps of binary search and analyze its efficiency compared to linear search

➢ <u>Evaluating</u>
1. Assess the efficiency and effectiveness of various data structures and algorithms in solving real-world problems.
2. Evaluate trade-offs between time and space complexity for different solutions.
3. Judge the appropriateness of sorting techniques for specific data sets.
4. Critique the design of a data structure or algorithm implementation for potential improvements.

➢ <u>Creating</u>
1. Design and implement custom data structures and algorithms to solve complex problems.
2. Create a real-world application that utilizes a combination of arrays, hash maps, and trees.
3. Develop a visualization or simulation of sorting algorithms or graph traversal methods.
4. Build a program to represent and process hierarchical or networked data using binary trees or graphs.</td></tr>
<tr><td rowspan="3">Knowledge</td><td>Code</td><td>Description</td></tr>
<tr><td>TPK23</td><td>Data structures</td></tr>
<tr><td>TPK24</td><td>Algorithms</td></tr>
<tr><td rowspan="6">Skill</td><td>Code</td><td>Description</td></tr>
<tr><td>TPC6.1</td><td>Using data structures for code optimization and problem-solving</td></tr>
<tr><td>TPC6.2</td><td>Solving problems using algorithms</td></tr>
<tr><td>TPC6.3</td><td>Create classes with custom methods, including initializers and decorated properties</td></tr>
<tr><td>TPC6.4</td><td>Analyze object-based design patterns</td></tr>
<tr><td>TPC6.5</td><td>Handle and produce errors (builtin or custom) to process or signal failure</td></tr>
</table>

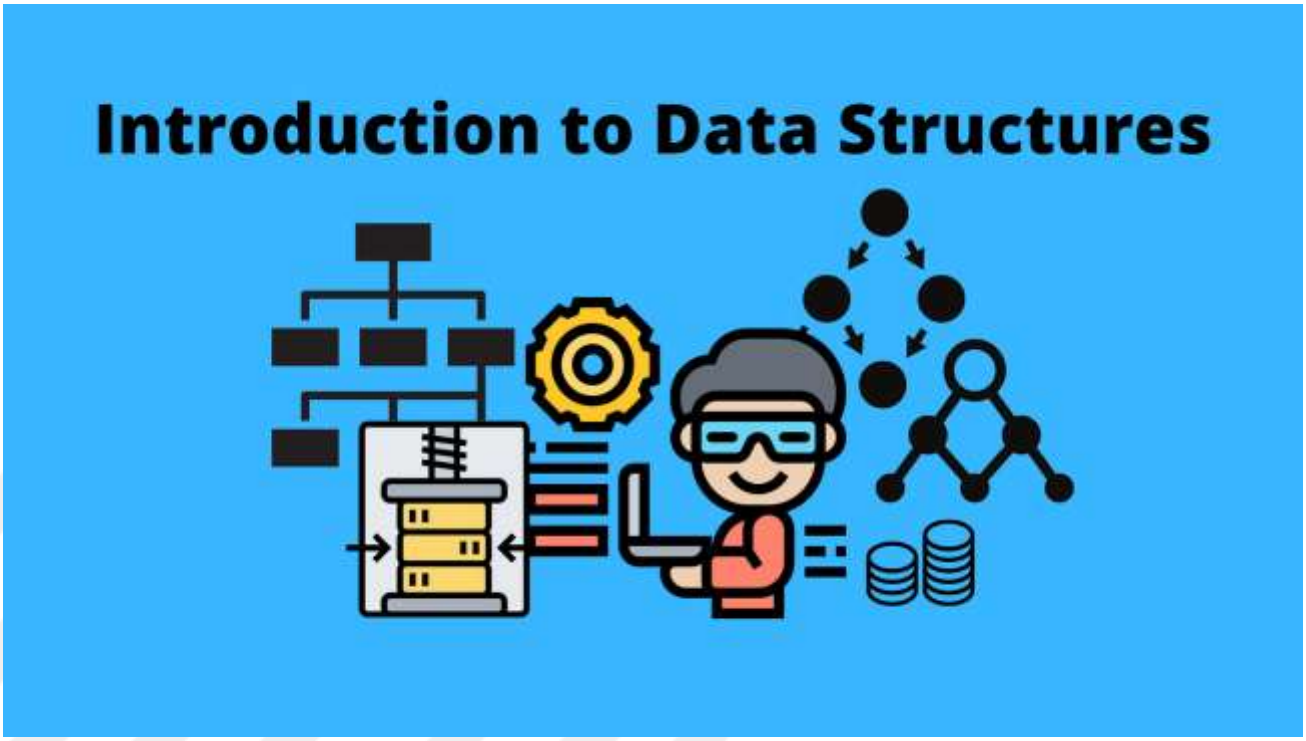
Unit Preface



Lesson	**1**
Name	**Introduction to data structure and algorithm**
Goals / Outcomes	**By the end of this lesson, students should be able to:** ➢ **<u>Remembering</u>** 1. Define what data structures and algorithms are. 2. List the types of data structures (e.g., arrays, linked lists, stacks, queues, trees, graphs). 3. Recall the meaning of complexity (time complexity and space complexity). 4. Identify examples of algorithms and their purposes ➢ **<u>Understanding</u>** 1. Explain the importance of data structures and algorithms in programming. 2. Describe the characteristics and purposes of different types of data structures. 3. Understand how complexity impacts the efficiency of algorithms. 4. Interpret the difference between time complexity and space complexity. ➢ **<u>Applying</u>** 1. Classify problems that can be solved using specific data structures or algorithms. 2. Apply basic complexity analysis to simple algorithms (e.g., counting the number of operations). 3. Demonstrate how an algorithm processes data in a step-by-step manner. ➢ **<u>Analyzing</u>** 1. Compare different types of data structures and their use cases. 2. Analyze the efficiency of simple algorithms by evaluating their time and space complexity.

	3. Break down the steps of an algorithm to determine its overall purpose and functionality. ➢ **<u>Evaluating</u>** 1. Assess the suitability of specific data structures for solving particular problems. 2. Evaluate the trade-offs between using different data structures in terms of performance and memory usage. 3. Critique the efficiency of algorithms based on their complexity. ➢ **<u>Creating</u>** 1. Design a basic algorithm to solve a small problem (sorting a list or searching for an element). 2. Develop an example that highlights the importance of choosing the right data structure for a given task. 3. Create visual representations of data structures to help explain their functionality and organization.	
Knowledge	**Code**	**Description**
	TPK23	Data structures
	TPK24	Algorithms
Skill	**Code**	**Description**
	TPC6.1	Using data structures for code optimization and problem-solving
	TPC6.2	Solving problems using algorithms
	TPC6.3	Create classes with custom methods, including initializers and decorated properties
	TPC6.4	Analyze object-based design patterns
	TPC6.5	Handle and produce errors (builtin or custom) to process or signal failure

Lesson 1: Introduction to data structure and algorithm



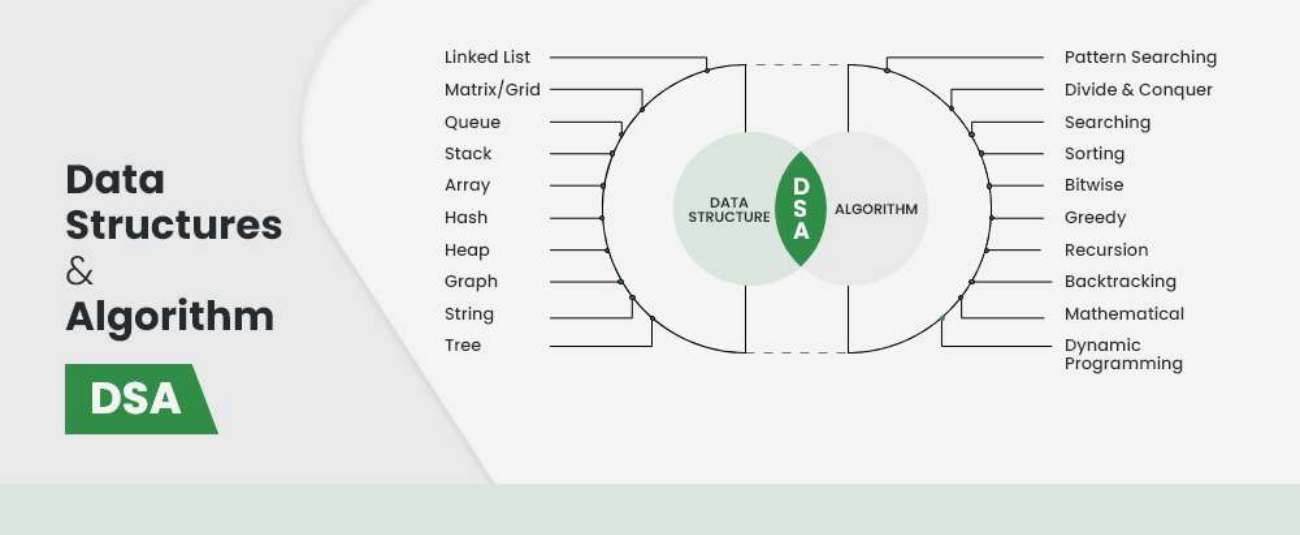
IN THIS LESSON WE LEARN:

- **Introduction**
- **Data Structures**
- **types of Data Structures**
- **Algorithms**
- **Complexity**

Section 1 Introduction

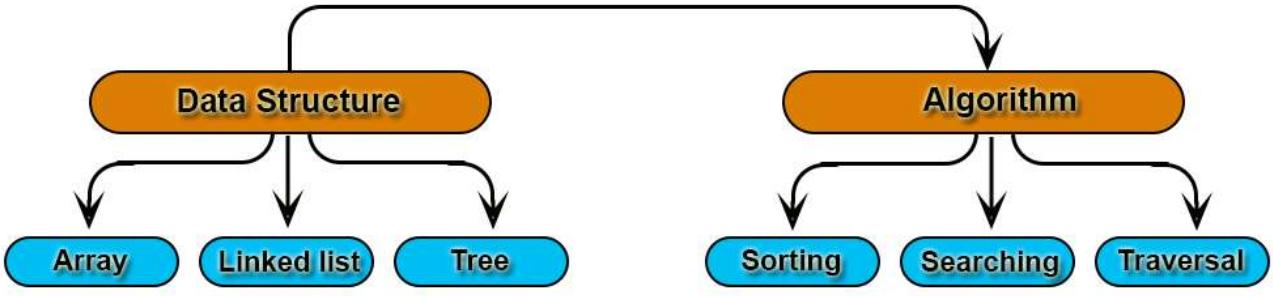
1-1 Data Structures and Algorithms (DSA) Introduction

Data structures and algorithms (DSA) are two important aspects of any programming language. Every programming language has its own data structures and different types of algorithms to handle these data structures.



Data Structures are used to organize and store data to use it in an effective way when performing data operations.

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.



Understanding and utilizing DSA is especially important when optimization is crucial, like in game development, live video apps, and other areas where even a one-second delay can impact performance.

Section 2 Data Structures

2-1 What are Data Structures?

Data Structure is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage

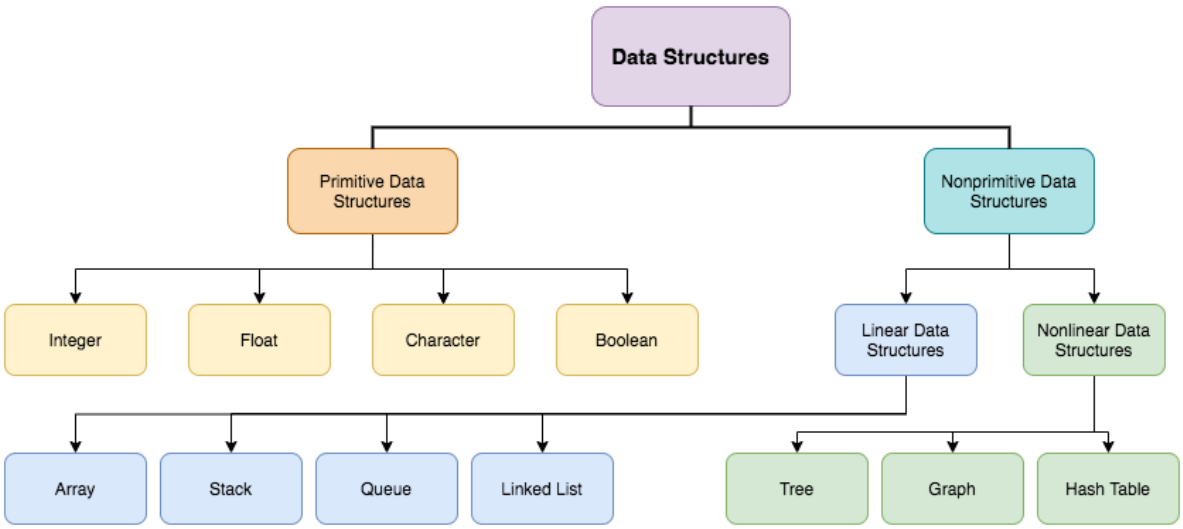
For example, we have some data which has, player's **name** "Mohamed" and **age** 26. Here "Mohamed" is of **String** data type and 26 is of **integer** data type.

We can organize this data as a record like **Player** record, which will have both player's name and age in it. Now we can collect and store player's records in a file or database as a data structure. **For example**: "Ramy" 30, "Shady" 31, "Ehab" 33

2-2 types of Data Structures

As we have discussed above, anything that can store data can be called as a data structure, hence Integer, Float, Boolean, Char etc., all are data structures. They are known as **Primitive Data Structures**.

Then we also have some complex Data Structures, which are used to store large and connected data. Some examples of **Abstract Data Structure** are :



Common data structures include:

- **Arrays**: Fixed-size collections of elements of the same type.
- **Linked Lists**: Collections of elements where each element points to the next.
- **Stacks and Queues**: Collections that follow specific order rules (LIFO for stacks, FIFO for queues).
- **Trees**: Hierarchical structures with nodes connected by edges.
- **Graphs**: Collections of nodes connected by edges, used to represent networks.
- **Hash Tables**: Structures that map keys to values for efficient lookup.

2-3 Choosing the appropriate data structure

Choosing the appropriate data structure depends on several factors, including:

• **Data type:** Are they numbers, texts, or objects?

• **Access operations:** Do you need random access to data or sequential access?

• **Insertion and deletion operations:** Do you need to insert or delete items frequently?

• **Data size:** The amount of data you will store.

Note: understanding data structures is essential for any programmer, as it helps build more efficient and effective programs.

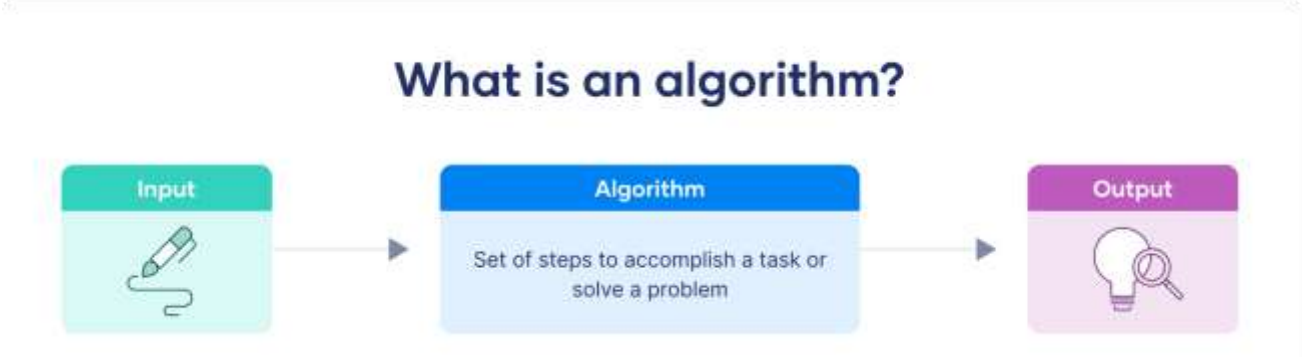
Practical example:
Suppose you want to build an application for a social network. You might use:

• **Array:** to store the list of friends for each user.

• **Tree:** to represent the structure of relationships between users.

• **Graph:** to represent the entire social network and the relationships between users.

Section 3 Algorithms



Algorithms are step-by-step procedures or formulas for solving problems. They operate on data structures to perform tasks such as searching, sorting, and manipulating data. Examples include:

- **Sorting Algorithms**: Methods like quicksort, merge sort, and bubble sort that arrange data in a specific order.
- **Search Algorithms**: Techniques like binary search and linear search to find elements in data structures.
- **Graph Algorithms**: Procedures like Dijkstra's and A* for finding the shortest path in a graph.
- **Dynamic Programming**: A method for solving complex problems by breaking them down into simpler subproblems.

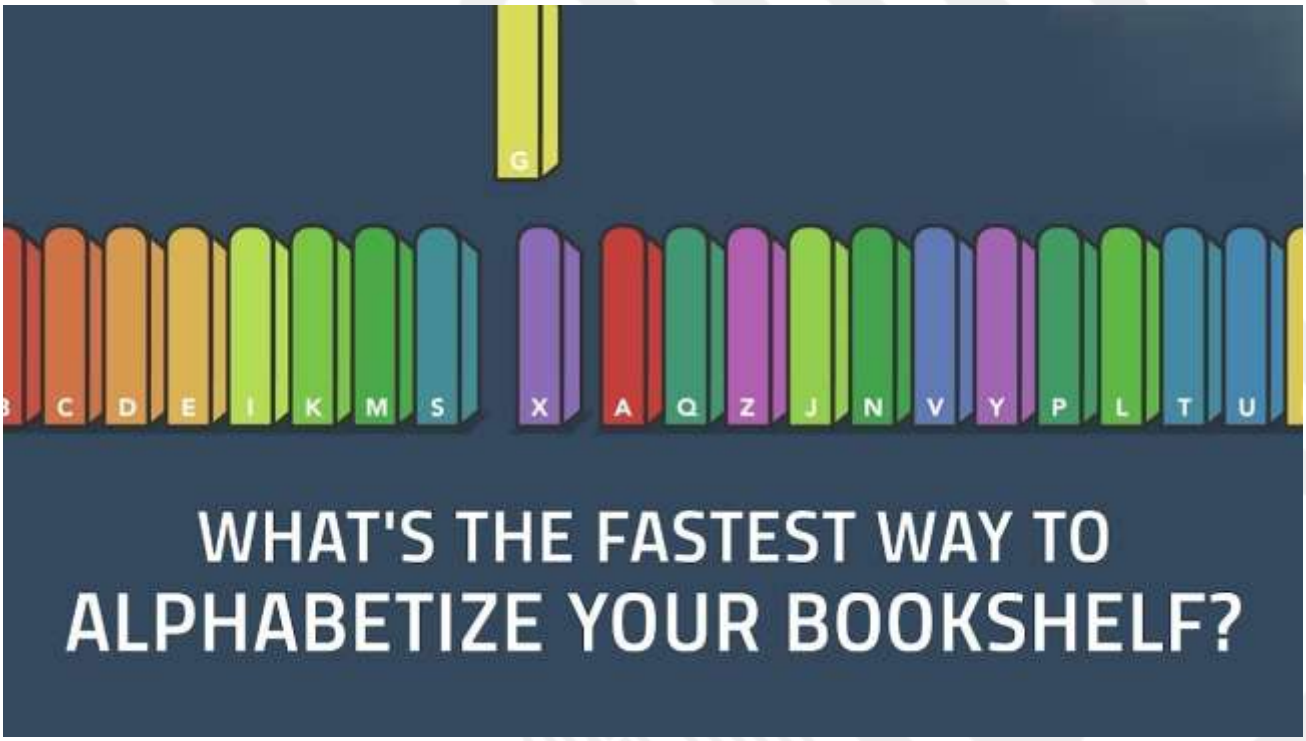
1-2-3 How They Work Together

- **Efficiency**: The choice of data structure can significantly impact the efficiency of an algorithm. For example, using a hash table can make search operations much faster compared to a linked list.
- **Optimization**: Algorithms are designed to optimize operations on data structures. For instance, quicksort is an efficient algorithm for sorting arrays.
- **Problem Solving**: Together, they provide a toolkit for solving a wide range of computational problems. For example, a graph algorithm can use a priority queue (a type of data structure) to efficiently find the shortest path.

In essence, data structures provide the means to manage data, while algorithms provide the methods to process that data. Their synergy is crucial for developing efficient and effective software solutions.

Section 4 Complexity

Imagine you're trying to find a specific book in a library.



Scenario 1: The books are scattered randomly on the shelves.

- You'd have to look at every book one by one to find the one you're looking for. This would take a long time, especially if the library is big.
- This is like an algorithm with **high complexity**. It takes a lot of time to complete.

Scenario 2: The books are organized alphabetically on the shelves.

- You can quickly find the book by looking at the labels and going to the right section.

- This is like an algorithm with **low complexity**. It's much faster than searching randomly.

Complexity is a measure of how fast an algorithm can solve a problem. A low-complexity algorithm is faster than a high-complexity algorithm.

Think about these examples:

- **Finding a friend in a crowded room:** Searching randomly has high complexity, while knowing their location has low complexity.
- **Finding a word in a dictionary:** Using alphabetical order has low complexity, while searching randomly has high complexity.

Important Notes:

- **Complexity depends on the size of the problem.** A fast algorithm for a small problem might be slow for a large problem.
- **Different algorithms can have different complexities for the same problem.**
- **Understanding complexity helps you choose the best algorithm for a given task.**

Let's make it more fun!

Imagine you're playing a guessing game. The person guessing must find a number between 1 and 100.

- If they guess randomly, it might take many tries. (**High complexity**)
- If they use a strategy like dividing the range in half each time, they can find the number much faster. (**Low complexity**)

By understanding complexity, you can create more efficient and effective programs!

Section 5. Terminology

Data Structures – Ways to organize and store data efficiently for performing operations. Examples include arrays, linked lists, trees, graphs, stacks, queues, and hash tables.

Algorithm – A step-by-step procedure or set of instructions designed to perform a specific task or solve a problem.

Primitive Data Structures – Basic data types such as integers, floats, booleans, and characters.

Abstract Data Structures – More complex data structures designed for handling large and connected data, such as linked lists, trees, graphs, and hash tables.

Array – A collection of elements stored in contiguous memory locations, allowing fast access by index.

Linked List – A sequence of nodes where each node points to the next, allowing dynamic memory allocation and efficient insertions/deletions.

Stack – A data structure that follows the **LIFO (Last In, First Out)** principle, where the last inserted element is the first to be removed.

Queue – A data structure that follows the **FIFO (First In, First Out)** principle, where the first inserted element is the first to be removed.

Tree – A hierarchical data structure consisting of nodes, with a root node and child nodes connected by edges.

Graph – A collection of nodes (vertices) connected by edges, used to model relationships between objects.

Hash Table – A data structure that maps keys to values using a hash function for efficient searching and retrieval.

Sorting Algorithms – Methods to arrange data in a particular order, such as quicksort, merge sort, and bubble sort.

Search Algorithms – Techniques to find elements within a data structure, such as linear search and binary search.

Graph Algorithms – Algorithms used to traverse and analyze graphs, such as Dijkstra's algorithm for shortest path finding.

Dynamic Programming – A method for solving complex problems by breaking them down into overlapping subproblems and storing results to avoid redundant computations.

Complexity – A measure of the efficiency of an algorithm in terms of time (time complexity) and space (space complexity) required for execution.

Time Complexity – The amount of time an algorithm takes to complete based on input size.

Space Complexity – The amount of memory an algorithm requires to execute.

Big O Notation – A mathematical notation used to describe the worst-case or upper bound performance of an algorithm, such as $O(n)$, $O(\log n)$, and $O(n^2)$.

Optimization – The process of improving an algorithm's efficiency to reduce execution time or memory usage.

DO I KNOW THIS ALREADY ?

Dear learner: Put "True" in front of the correct statement and "False" in front of the incorrect statement.

1	Data structures are only used for organizing data, not for storing it. ✗	
2	An algorithm is a step-by-step procedure to solve a problem. ✓	
3	Sorting algorithms helps arrange data in a specific order. ✓	
4	Primitive data structures include integers, floats, and characters. ✓	
5	A social network can be represented as a graph data structure. ✓	
6	Complexity measures the speed of an algorithm. ✓	
7	Dynamic programming is a technique to optimize recursive problems. ✓	
8	Big O notation is used to describe algorithm complexity. ✓	
9	An efficient algorithm always has a low time complexity. ✗	
10	Algorithm complexity helps in determining how the performance of an algorithm scales with the size of the input. ✓	

Dear learner: Choose the correct answer.

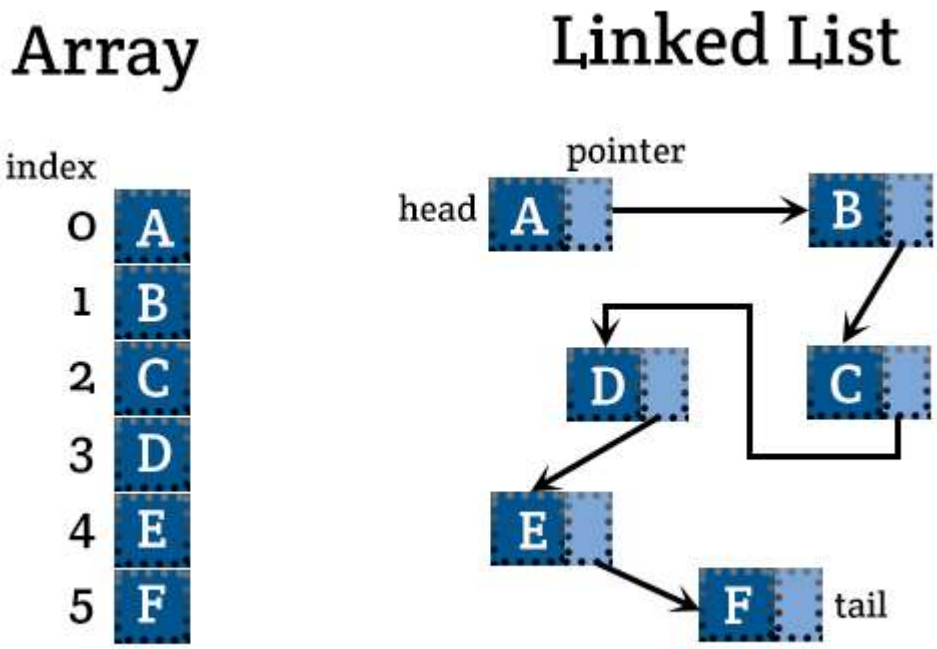
11	**What are data structures used for?** A) To compile programs B) To organize and store data efficiently C) To execute machine code D) To create new programming languages	
12	**What is an algorithm?** A) A programming language B) A step-by-step procedure to solve a problem C) A data storage method D) A type of software	
13	**Why is DSA important in performance-critical applications like game development?** A) It reduces code readability B) It allows faster and more efficient data processing C) It increases the complexity of programs D) It only works in C++	
14	**4. What is a data structure?** A) A way to store and organize data B) A type of algorithm C) A programming language D) A software framework	
15	**Which of the following is NOT a primitive data structure?** A) Integer B) Float C) Graph D) Boolean	
16	**What type of data structure is an array?** A) A collection of elements of different types B) A fixed-size collection of elements of the same type	

	C) A collection of random numbers D) A hierarchical data structure	
17	**Which data structure is best suited for hierarchical data?** A) Stack B) Queue C) Tree D) Array	
18	**When choosing an appropriate data structure, what factor should be considered?** A) The number of functions available in the language B) The data type and operations required C) The name of the programming language D) The color of the interface	
19	**What is an algorithm used for?** A) Organizing data B) Solving problems efficiently C) Deleting data from memory D) Managing operating systems	
20	**What is dynamic programming used for?** A) Solving complex problems by breaking them into smaller subproblems B) Creating new programming languages C) Organizing hierarchical data D) Converting algorithms into data structures	

Lesson	2
Name	**Arrays-Linked List and Hash map**
Goals / Outcomes	**By the end of this lesson, students should be able to:** ➢ **Remembering** 1. Define what arrays, linked lists, and hash maps are. 2. Recall the key characteristics and structure of each data structure. 3. Identify the basic operations performed on arrays, linked lists, and hash maps. ➢ **Understanding** 1. Explain the differences between arrays, linked lists, and hash maps. 2. Describe how data is stored and accessed in each data structure. 3. Understand the advantages and limitations of each data structure in different scenarios. ➢ **Applying** 1. Implement arrays, linked lists, and hash maps in code to solve basic problems. 2. Perform common operations like inserting, deleting, and searching for elements in each data structure. 3. Use hash maps to efficiently store and retrieve key-value pairs. ➢ **Analyzing** 1. Compare the time and space complexity of arrays, linked lists, and hash maps. 2. Analyze the suitability of arrays, linked lists, or hash maps for specific types of problems. 3. Break down how hash functions work in a hash map and how collisions are handled.

	➢ **Evaluating** 1. Evaluate the trade-offs between using arrays, linked lists, or hash maps in terms of performance and memory usage. 2. Assess the effectiveness of hash maps in storing and retrieving data for real-world applications. 3. Critique different implementations of linked lists (e.g., singly vs. doubly linked lists). ➢ **Creating** 1. Design and implement a simple application that utilizes arrays, linked lists, or hash maps to manage data. 2. Create a program that demonstrates the efficiency of hash maps compared to arrays or linked lists for specific tasks. 3. Develop a visualization or simulation to show how linked lists or hash maps work internally.	
Knowledge	**Code**	**Description**
	TPK23	Data structures
	TPK24	Algorithms
Skill	**Code**	**Description**
	TPC6.1	Using data structures for code optimization and problem-solving
	TPC6.2	Solving problems using algorithms
	TPC6.3	Create classes with custom methods, including initializers and decorated properties
	TPC6.4	Analyze object-based design patterns
	TPC6.5	Handle and produce errors (builtin or custom) to process or signal failure

Lesson2: Arrays-Linked List and Hash map



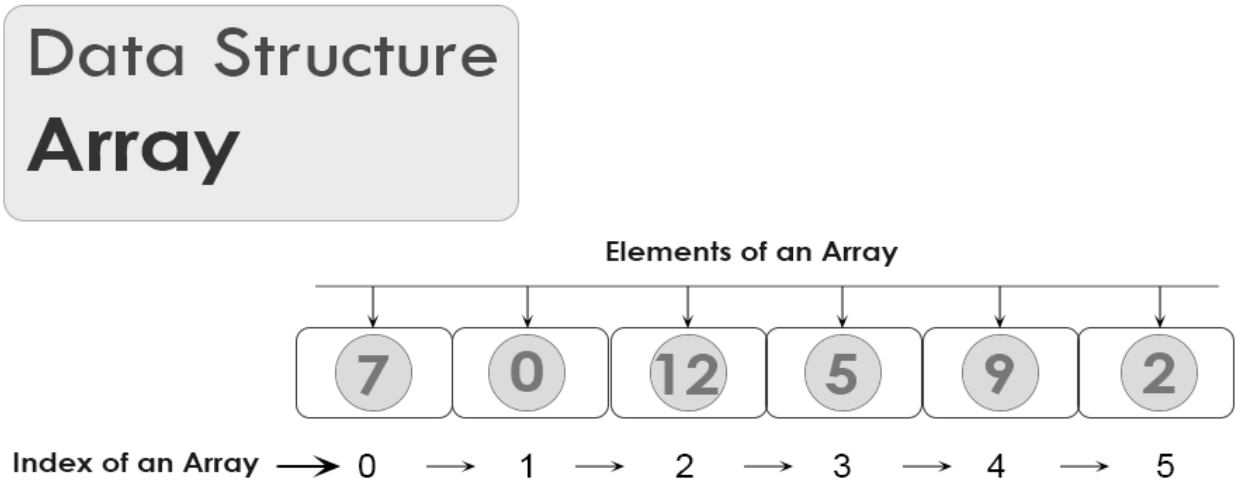
IN THIS LESSON WE LEARN:

- **Arrays**
- **Hash map**
- **Linked List**

Section 1 Arrays Introduction

1-1 Introduction

In programming, we deal not only with numbers and letters, but with collections of such data. These organized collections are what we call data structures. One of the simplest and most common data structures is the array.



1-2 What is an array?

An array is an ordered collection of elements, where each element can be accessed through a unique numerical index starting at zero. Think of an array as a box with numbered drawers, where each element can be placed in a specific drawer.

1-3 Why do we use arrays?

- **Organize data:** Arrays help to organize data in an organized and easy to access way.
- **Recursion**: We can repeat the same process on all the elements of the array using a loop.
- **Sorting and searching:** Arrays make sorting and searching for specific elements easy.

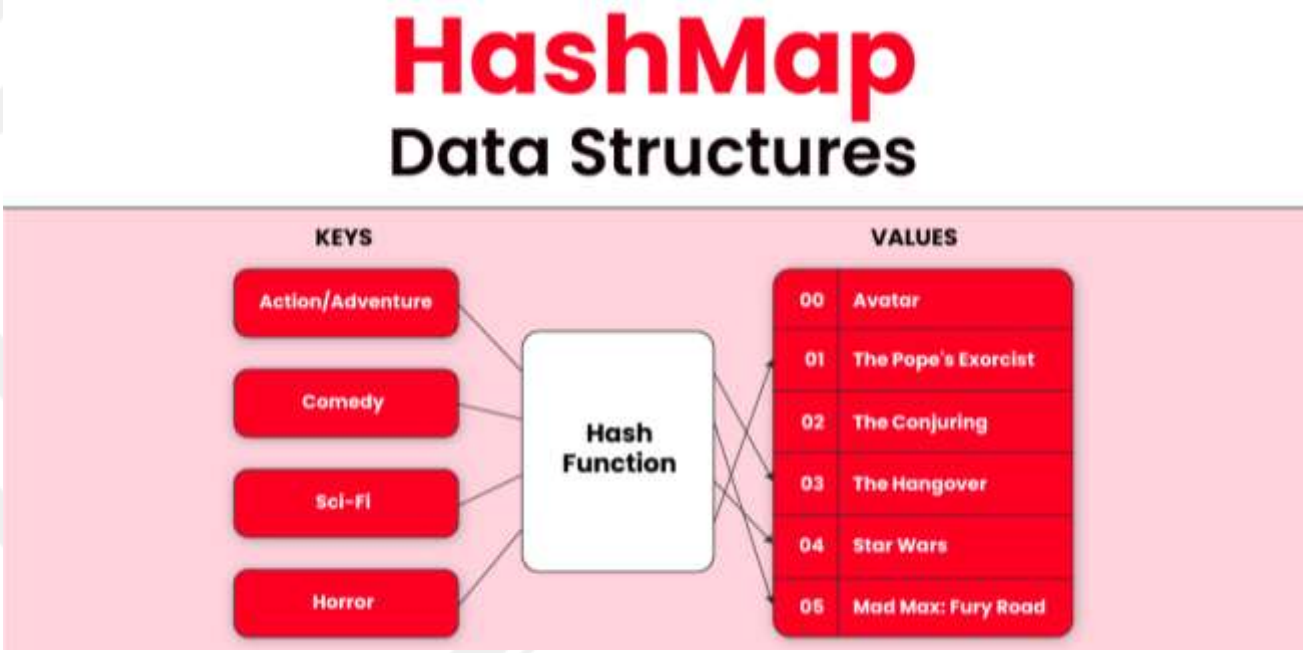
1-4 Important Notes:

- The array index starts from zero. → Zero base indexing
- The length of the array is dynamic and can be changed.
- The array can contain elements of different data types.

1-5 Why are arrays important?

- **Data organization:** Arrays provide an organized way to store data.
- **Fast access:** Elements in an array can be accessed easily and quickly using their pointers.
- **Handling big data:** Arrays can handle large amounts of data efficiently.
- **Basis for building other data structures:** Arrays are the basis for building more complex data structures such as linked lists and trees.

Section 2 Hash map



2-1 What is a Hash Map?

Imagine that you have a big box divided into drawers (or sections). Each drawer has a number or key, and when you put something in the box, you use this number or key to know where it is. This makes it easy to find things quickly!

The idea of a Hash Map is simple:

- **Key:** The way we determine where to put or find the thing.
- **Value:** The thing itself that we want to store.

The Hash Map acts like a magic list that tells you where to find the thing once you know the key.

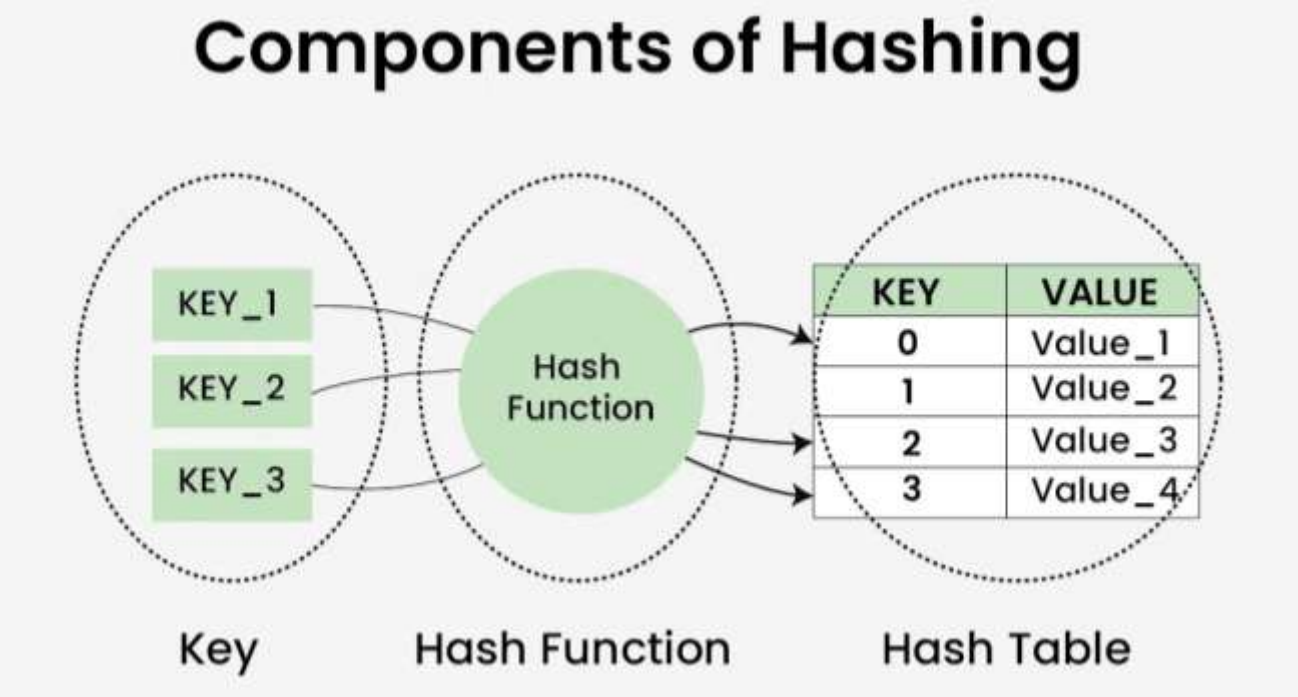
2-2 Example from everyday life:

Imagine that you went to a library with many shelves. Each shelf has a number (such as 1, 2, 3...), and when you want a book, you look at the number written in the book list (such as "Shelf No. 5").

- **Key:** The shelf number.
- **Value:** The book on the shelf.

2-3 How does a Hash Map work?

1. When you want to store something, you give it a key (such as its name or number).
2. A special function (Hash Function) is used to convert the key into a number that indicates where it is stored.
3. When you need the item later, you give the key, and the map tells you where it is immediately.



2-4 Benefits of Hash Map:

1. **Search speed:** You can find anything using just the key.
2. **Organization:** It stores things in an organized way that makes them easy to access.
3. **Flexibility:** You can add or delete data easily.

A simple comparison for students between Hash Map and Array:

Hash Map	Array
Relies on a key to find quickly.	Needs to search through all elements.
Easy to add and remove.	Requires modifying elements when removing.
Best for relational data (key-value).	Best for simple lists.

2-5 Practical activity to illustrate Hash Map:

Tools:

- Small cards.
- A box divided into numbered sections.
- A pen.

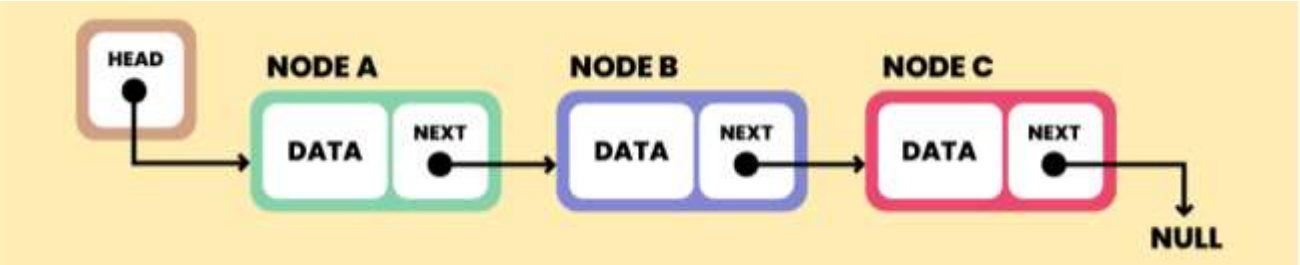
Method:

1. On each card, write a name (e.g. "Ali"), and on the back write a number (e.g. "123456").
2. Place the card in the appropriate section inside the box (choose the section number logically or randomly).
3. Ask the student to find a phone number using the name:

If you say, "Where is Ali's number?", they have to quickly find the card using the name.

Meaning: The key (name) identifies the location of the card that holds the value (number).

Section 3 Linked List



3-1 What is the linked list?

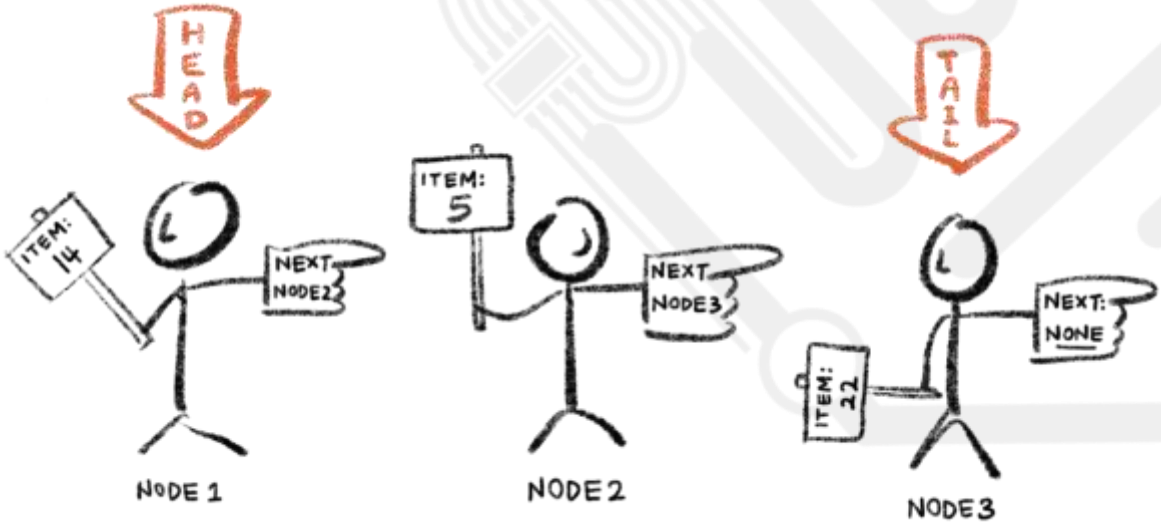
A linked list is a special type of data structure that is used to store data in a way that makes it easy to quickly add or remove data. The main idea is that the data is not stored in adjacent locations like arrays but is linked to each other using links.

3-2 How does a linked list work?

A linked list consists of a collection of "nodes", and each node contains:

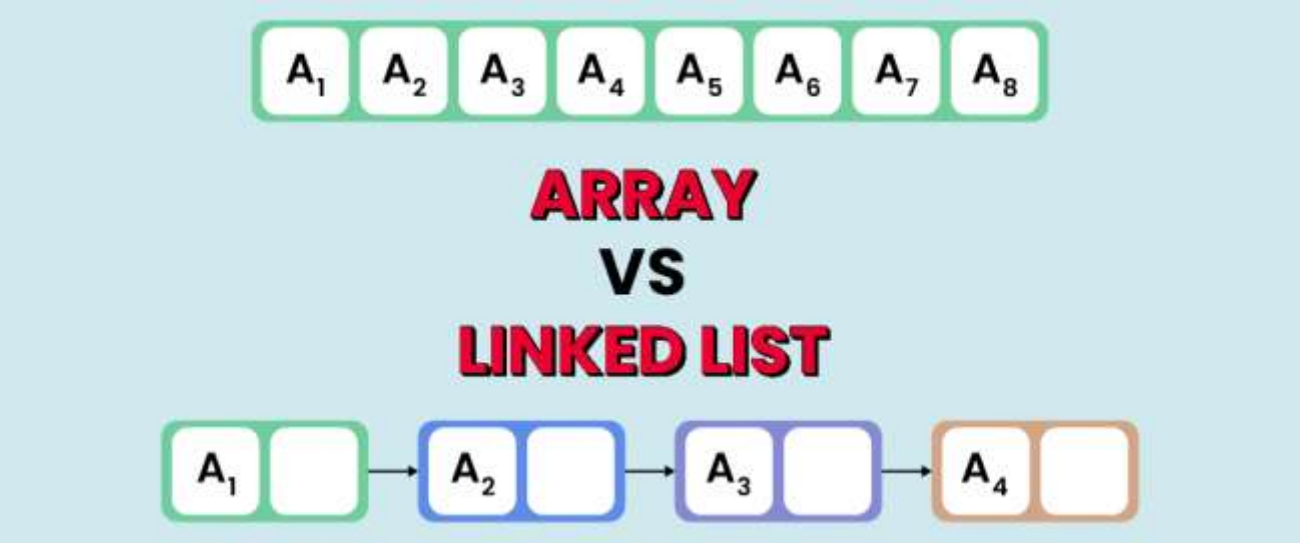
1. Data (such as a name, number, or anything else).
2. A link to the next node in the list.

Each node points to the node that comes after it. The last node is called the null node because it does not point to any other node.



3-3 Comparison between a linked list and an array:

- **An array:** is a collection of data that is stored in adjacent locations in memory.
- **A linked list:** is a collection of nodes, each of which contains data and a link to the next node.

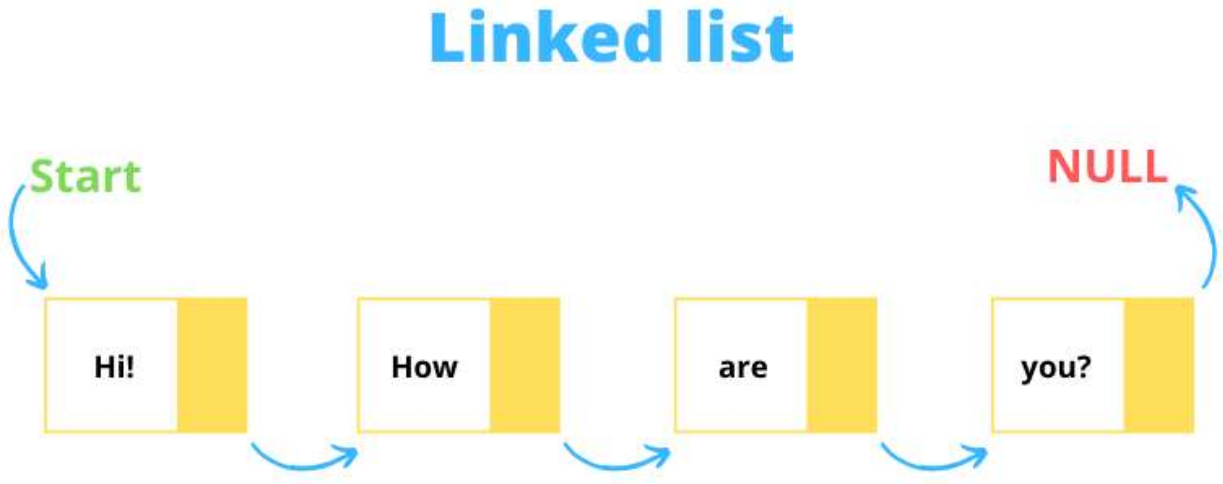


In a linked list, you can easily add or remove elements, whereas in an array, you may need to move the rest of the elements when adding or removing an element.

3-4 What does a node contain?

A node contains two parts:

1. **Data:** It can be any type of data.
2. **Pointer:** A link to the next node in the list.

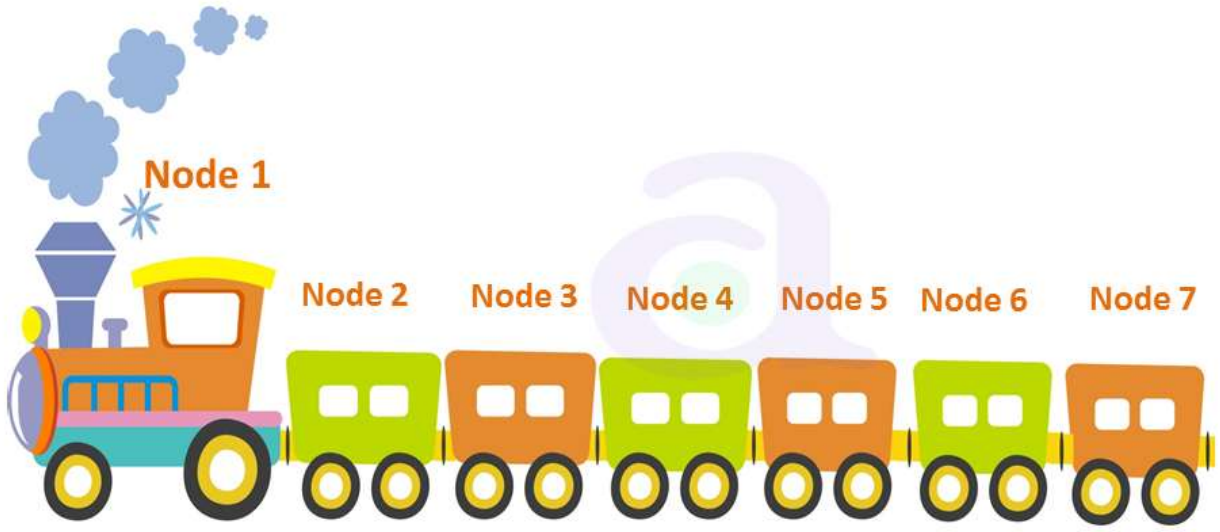


3-5 Benefits of Linked List:

1. **Easy to add:** You can add a new element anywhere without having to sort.
2. **Flexible:** You don't need to know the size in advance, unlike an array.
3. **Easy to add and delete:** We can easily add or delete nodes from anywhere in the list without having to move the rest of the elements. You can remove any element just by modifying the arrow
4. **Flexible memory consumption:** Linked list only uses the memory needed for the nodes that contain data.

3-6 A Train Example:

Think of a linked list like a train, where each carriage (bogie) is connected to the next one by a coupling mechanism. This chain of carriages move forward as the train progresses.



This train analogy is a great way to visualize how linked lists work. The first car is the head of the list, and the last car has no connection to another car, representing the NULL pointer.

3-7 Practical activity to illustrate Linked List:

Tools:

- Small sheets of paper.
- Pen.
- Paper clips.

Method:

1. Write a name on each sheet of paper ("Ali", "Sara", "Ahmed").
2. Connect the sheets of paper with the paper clips.
3. Add an arrow on the sheet of paper that points to the next sheet.
4. Try removing a sheet of paper from the middle and reconnecting the chain.

Meaning:

- The paper represents nodes.
- The paper clip represents a pointer.

Section 4. Terminology

Array: An ordered collection of elements, accessible by a numerical index.

Index: A unique numerical identifier assigned to each element in an array, starting from zero.

Dynamic Length: The ability of an array to grow or shrink in size.

Data Organization: The process of structuring data efficiently within an array.

Sorting & Searching: Techniques used to arrange and locate elements in an array efficiently.

Linked List: A data structure where elements (nodes) are linked together rather than stored in adjacent memory locations.

Node: A fundamental unit in a linked list that contains data and a pointer to the next node.

Pointer: A reference or link to the next node in the sequence.

Null Node: The last node in a linked list that does not point to any other node.

Hash Map: A data structure that stores key-value pairs for fast retrieval.

Key: A unique identifier used to access values in a hash map.

Value: The data stored and associated with a specific key.

Search Speed: The ability to quickly find values using keys instead of scanning an entire list.

Flexibility: The ease of adding and removing data without shifting elements.

DO I KNOW THIS ALREADY ?

Dear learner: Choose the correct answer.

21	**What happens if you try to access an index beyond the array size?** a) It returns a null value b) It causes an error (IndexOutOfBounds) c) It loops back to the first element d) It adds a new element automatically	
22	**Which of the following is NOT a feature of an array?** a) Elements must be of the same data type b) Fixed size (in static arrays) c) Can dynamically grow or shrink d) Provides fast lookup using indices	
23	**What is a disadvantage of linked lists compared to arrays?** a) Cannot store multiple data types b) Cannot be dynamically allocated c) Uses more memory due to extra pointers d) Has a fixed size	
24	**Which of the following is NOT a benefit of using a linked list?** a) Easy to insert new elements b) Easy to delete elements c) Efficient random access d) Flexible memory usage	
25	**What is the primary purpose of a Hash Map?** a) Sorting data b) Storing key-value pairs efficiently c) Managing file systems d) Implementing recursion	
26	**Which of the following statements is TRUE about Hash Maps?** a) They use key-value pairs for quick lookups b) They always maintain a sorted order of keys c) They are slower than linked lists for searching d) They require shifting elements when adding new data	

Essay questions- clear and readable handwriting

27	**Question**	**Why are arrays important in data structures?**
	Answer	
28	**Question**	**How does an array differ from a linked list?**
	Answer	
29	**Question**	**Explain how a linked list works.**
	Answer	
30	**Question**	**What are the benefits of using a Hash Map instead of an array?**
	Answer	

Dear learner: Give The Scientific Term

31	An ordered collection of elements accessible by index.	
32	A data structure consisting of nodes linked together.	
33	A key-value based data structure is used for fast lookups.	
34	A node's reference to the next node in a linked list.	
35	The process of arranging data in a specific order.	

36-40 Match the following with their correct descriptions:

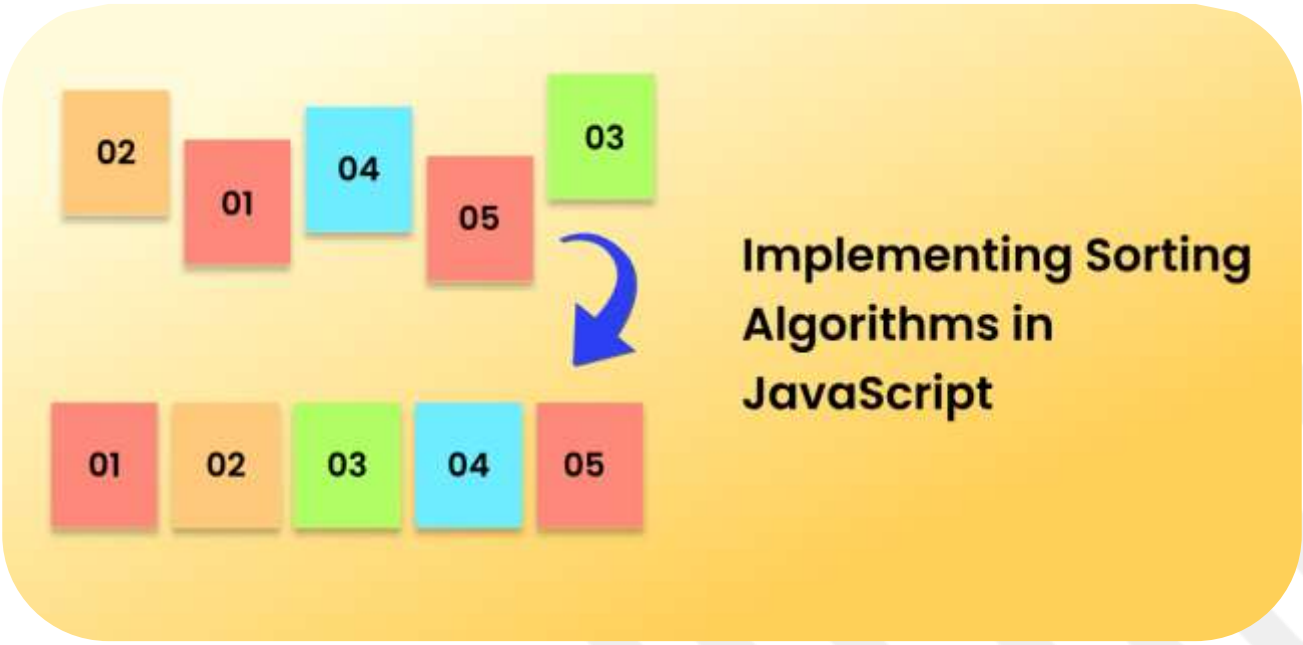
1	Array	A	Reference to the next node in a linked list	
2	Linked List	B	Ordered collection with index-based access	
3	Hash Map	C	Element in a linked list containing data and a pointer	
4	Node	D	Uses key-value pairs for storage	
5	Pointer	E	Data structure with nodes and pointers	

Lesson	**3**

Name	Selection and Sorting Algorithm
Goals / Outcomes	**By the end of this lesson, students should be able to:** ➢ **Remembering** 1. Define what sorting is and why it is important in programming. 2. List common sorting algorithms such as selection sort, bubble sort, and insertion sort. 3. Recall the basic steps involved in each sorting algorithm. ➢ **Understanding** 1. Explain how each sorting algorithm (selection sort, bubble sort, insertion sort) works. 2. Describe the differences between these sorting algorithms in terms of their process and logic. 3. Understand the concept of comparison-based sorting and how elements are arranged in order. ➢ **Applying** 1. Implement selection sort, bubble sort, and insertion sort in code to sort arrays. 2. Use sorting algorithms to arrange data in ascending or descending order in real-world scenarios. 3. Demonstrate how to trace and debug sorting algorithms step-by-step. ➢ **Analyzing** 1. Implement selection sort, bubble sort, and insertion sort in code to sort arrays. 2. Use sorting algorithms to arrange data in ascending or descending order in real-world scenarios. 3. Demonstrate how to trace and debug sorting algorithms step-by-step.

	➢ **Evaluating** 1. Implement selection sort, bubble sort, and insertion sort in code to sort arrays. 2. Use sorting algorithms to arrange data in ascending or descending order in real-world scenarios. 3. Demonstrate how to trace and debug sorting algorithms step-by-step. ➢ **Creating** 1. Implement selection sort, bubble sort, and insertion sort in code to sort arrays. 2. Use sorting algorithms to arrange data in ascending or descending order in real-world scenarios. 3. Demonstrate how to trace and debug sorting algorithms step-by-step.	
Knowledge	**Code**	**Description**
	TPK23	Data structures
	TPK24	Algorithms
Skill	**Code**	**Description**
	TPC6.1	Using data structures for code optimization and problem-solving
	TPC6.2	Solving problems using algorithms
	TPC6.3	Create classes with custom methods, including initializers and decorated properties
	TPC6.4	Analyze object-based design patterns
	TPC6.5	Handle and produce errors (builtin or custom) to process or signal failure

Lesson 3: Selection and Sorting Algorithm



IN THIS LESSON WE LEARN:

- **Introduction**
- **Selection Sort**
- **Bubble Sort**
- **Insertion Sort**

Section 1: Introduction

1-1 Do you know how we arrange things around us?

We arrange books on a shelf, we arrange toys in a box, and we even arrange ourselves in a queue! Order is very important in our lives, as it helps us find things easily and quickly.

In the world of computers, there are special ways to arrange things called "algorithms".

In this lesson we will learn together about two types of these algorithms: Selection Algorithm and Sorting Algorithm.

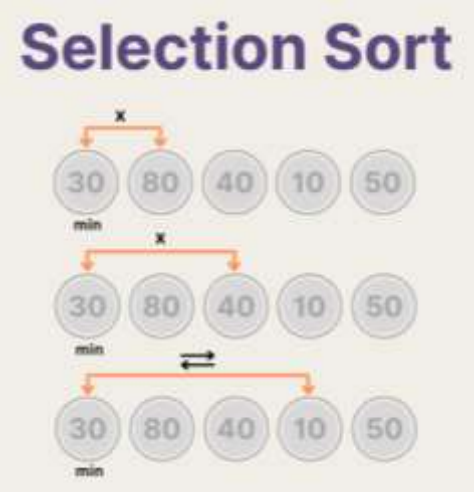
1-2 The Difference between Selection Algorithm and Sorting Algorithm

In the world of programming and data analysis, we use different algorithms to solve problems. Among these algorithms, there are the selection algorithm and the sorting algorithm. They may seem similar, but each has a different purpose and a different way of working. Let us explain the difference between them in a simple way.

We will show the difference through three factors:

- Purpose
- How They Work
- Example

1-2-1 First, Purpose of the Algorithm:

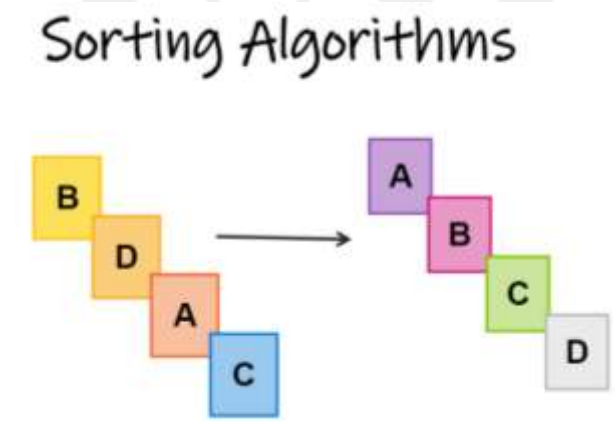


Selection Algorithm:

Purpose: To find a specific element in a collection of elements, such as finding the smallest element, the largest element, or the element at a specific position (the fifth element in a sorted list).

Example: If you have a list of numbers and want to find the smallest number in it.

Sorting Algorithm:



Purpose: To arrange elements in a collection in an organized manner, either from smallest to largest or from largest to smallest.

Example: If you have a list of numbers and want to sort them from smallest to largest.

1-2-2 Second, How They Work:

Selection algorithm:

- It examines the elements one by one to find the desired element.
- It does not care about the order of the other elements; it only finds the element you are looking for.
- **Example:** If you are searching for the smallest number in the list [5, 3, 8, 1, 2], you will check each number until you find that the smallest number is **1**.

Sorting algorithm:

- It arranges **all the elements** in the list in an organized manner.
- It cares about the relationship between each element and the others.
- **Example:** If you want to sort the list [5, 3, 8, 1, 2] from smallest to largest, the result will be [1, 2, 3, 5, 8].

1-2-3 Third, Practical Example for Clarification:

List: [7, 2, 5, 1, 9]

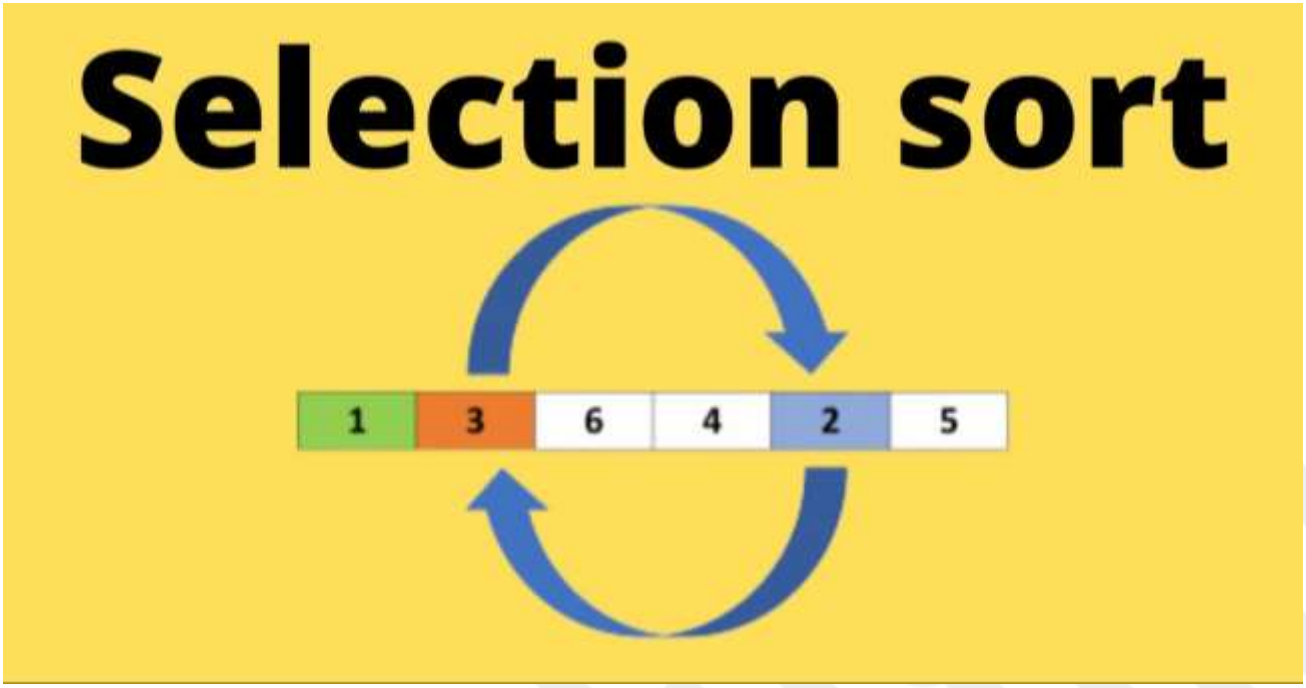
Selection algorithm:

- If you are searching for the **smallest element**, you will check the numbers one by one:
  - 7 (first number, consider it the smallest for now).
  - 2 (smaller than 7, becomes the smallest).
  - 5 (not smaller than 2).
  - 1 (smaller than 2, becomes the smallest).
  - 9 (not smaller than 1).
- Result: The smallest element is **1**.

Sorting algorithm:

- If you want to sort the list from smallest to largest:
  - Find the smallest element (1) and place it at the beginning: [1, 7, 2, 5, 9].
  - Find the smallest element in the remaining list (2) and place it in the second position: [1, 2, 7, 5, 9].
  - Find the smallest element in the remaining list (5) and place it in the third position: [1, 2, 5, 7, 9].
  - Finally, the numbers 7 and 9 remain in their places.
- Result: The sorted list is [1, 2, 5, 7, 9].

Section 2 Selection Sort



1-1 What is Selection Sort?

Selection Sort is a simple and intuitive sorting algorithm. It works by repeatedly selecting the smallest (or largest, depending on the order) element from the unsorted portion of the list and swapping it with the first unsorted element. This process continues until the entire list is sorted.

Working of Selection Sort

1. Set the first element as minimum.

Select first element as minimum

2. Compare minimum with the second element. If the second element is smaller than minimum, assign the second element as minimum.

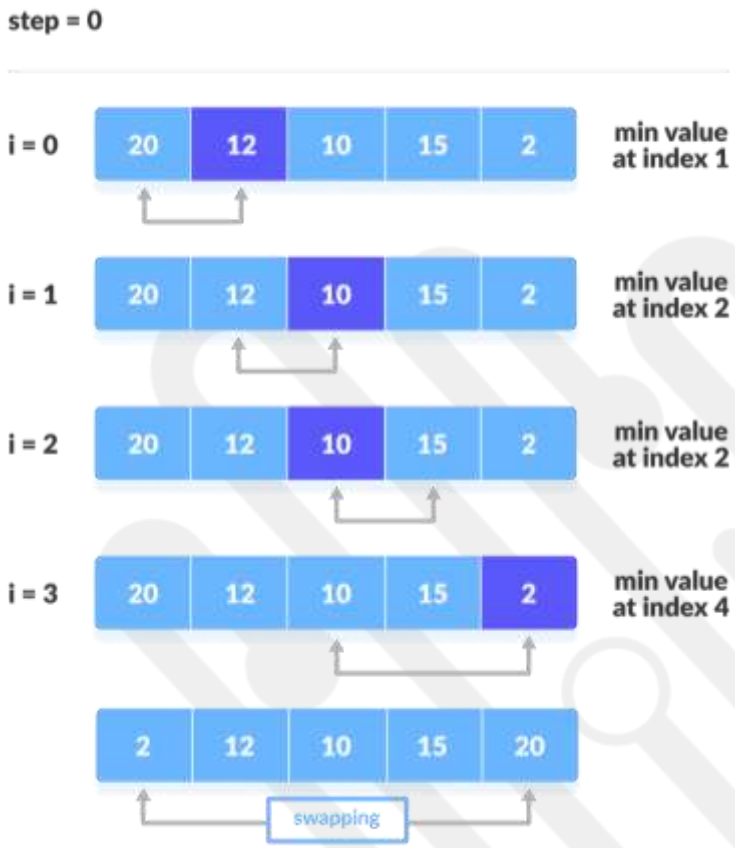
Compare minimum with the third element. Again, if the third element is smaller, then assign minimum to the third element otherwise do nothing. The process goes on until the last element.

Compare minimum with the remaining elements

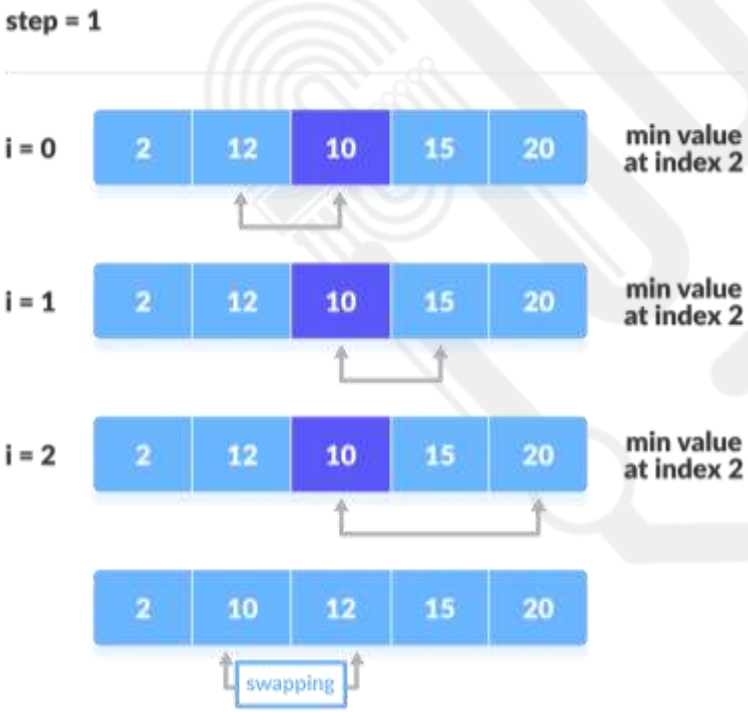
3. After each iteration, minimum is placed in the front of the unsorted List

Swap the first with minimum

4. For each iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions.



The first iteration



The second iteration

The third iteration

The fourth iteration

Practical activity to illustrate the idea:

Tools:

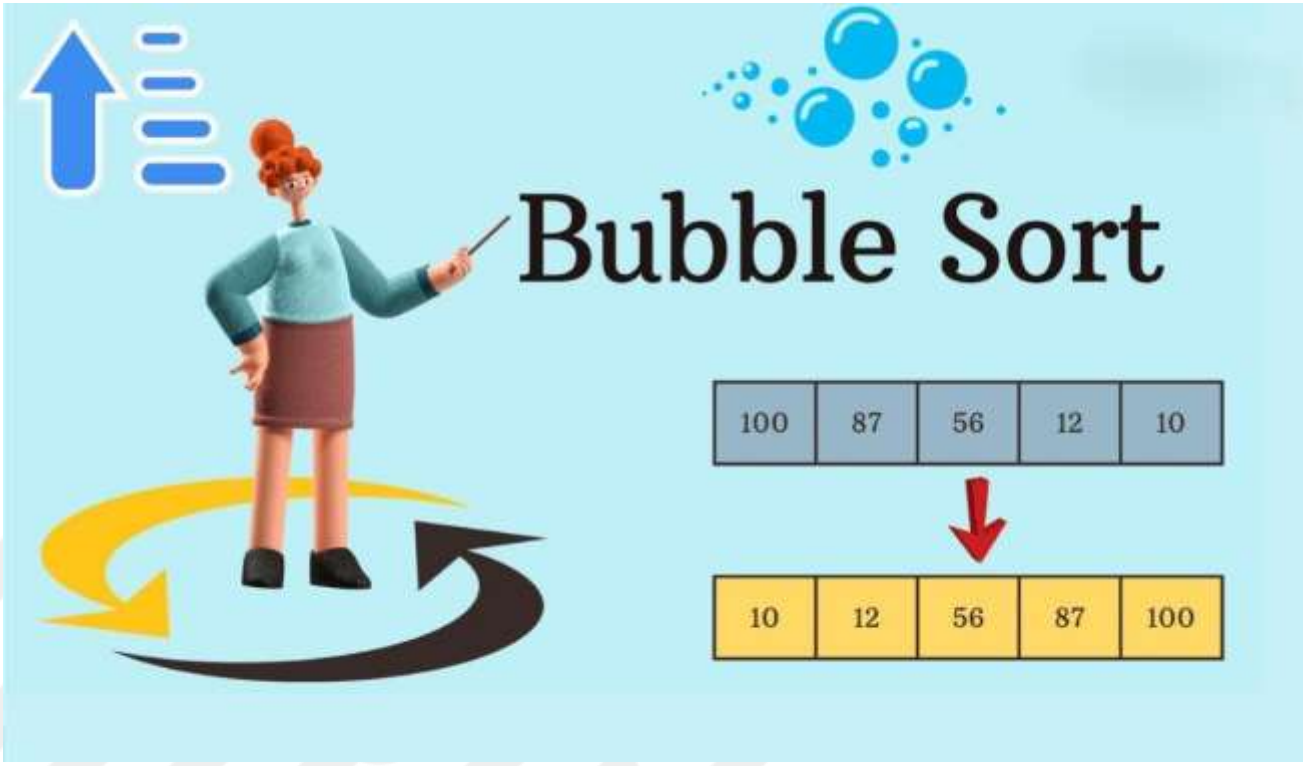
- Small papers with numbers on them.
- A table.

Method:

1. Write random numbers on the papers (5, 3, 8, 2).
2. Ask the child to:
    - Look at all the papers.
    - Choose the smallest number and put it at the beginning.
    - Repeat the process with the rest.

Outcome: Arrange the papers from smallest to largest.

Section 3 Bubble Sort



3-1 What is Bubble Sort?

Bubble Sort is a simple sorting algorithm that repeatedly steps through a list, compares adjacent elements, and swaps them if they are in the wrong order.

This process is repeated until the list is sorted. The algorithm gets its name because smaller elements "bubble" to the top of the list (beginning of the array) like bubbles rising in water.

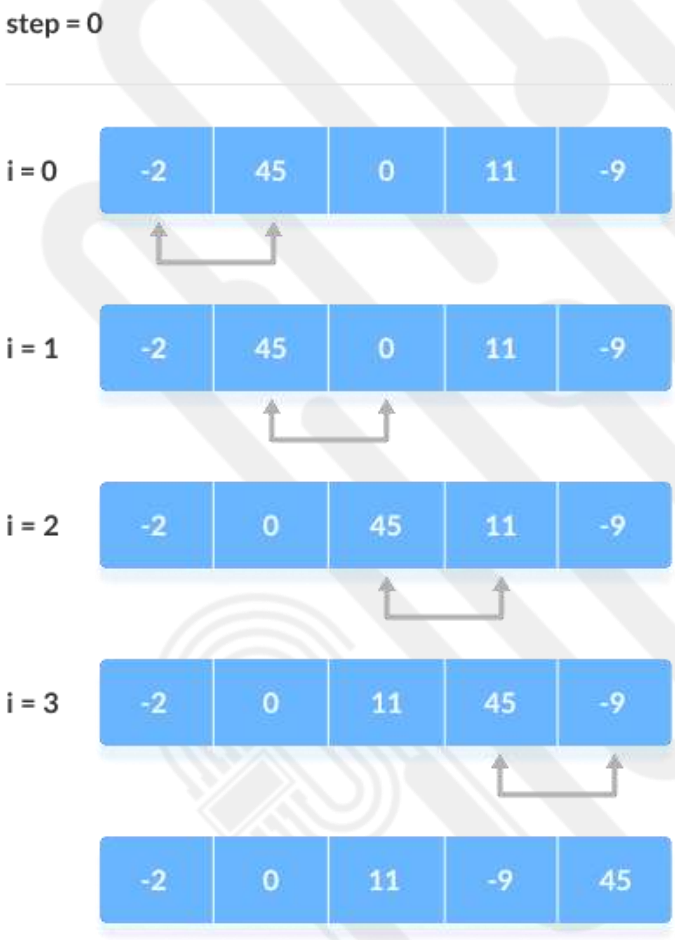
3-2 Working of Bubble Sort

Suppose we are trying to sort the elements in **ascending order**.

1. First Iteration (Compare and Swap)

1. Starting from the first index, compare the first and the second elements.

2. If the first element is greater than the second element, they are swapped.
3. Now, compare the second and the third elements. Swap them if they are not in order.
4. The above process goes on until the last element.

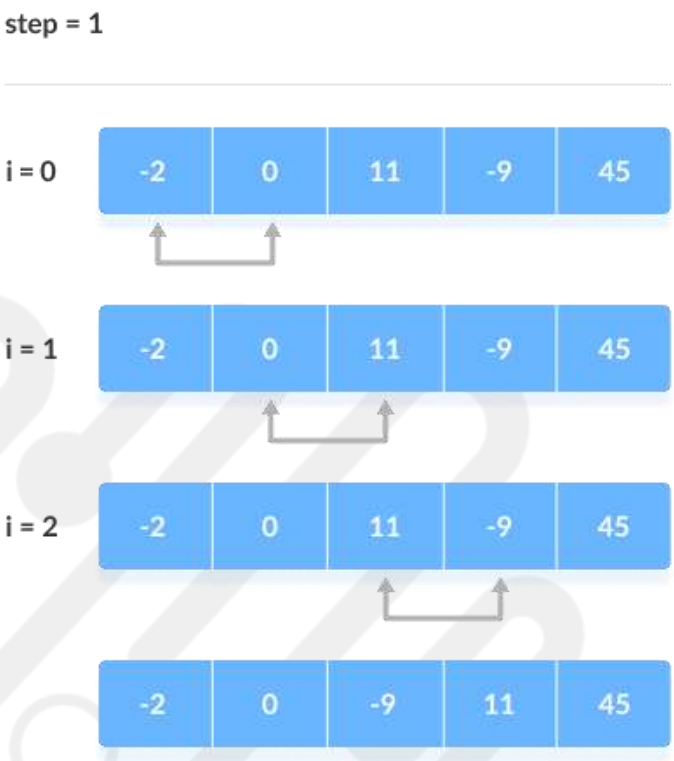


Compare the Adjacent Elements

2. Remaining Iteration

The same process goes on for the remaining iterations.

After each iteration, the largest element among the unsorted elements is placed at the end.

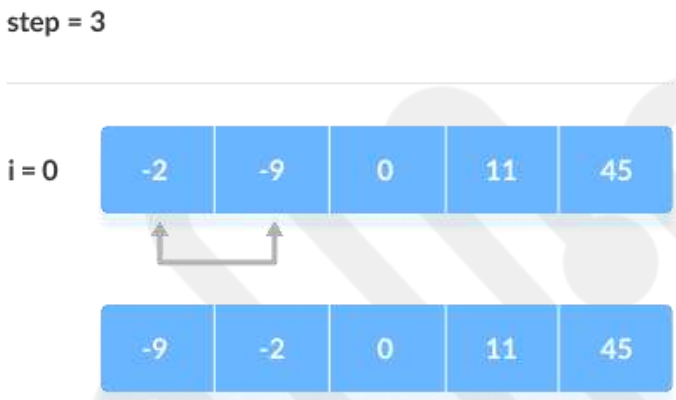


Put the largest element at the end

In each iteration, the comparison takes place up to the last unsorted element.

Compare the adjacent elements

The array is sorted when all the unsorted elements are placed at their correct positions.



The array is sorted if all elements are kept in the right order

3-3 Bubble Sort Activity:

Materials:

A set of cards with numbers ([7, 4, 5, 2]).

Method:

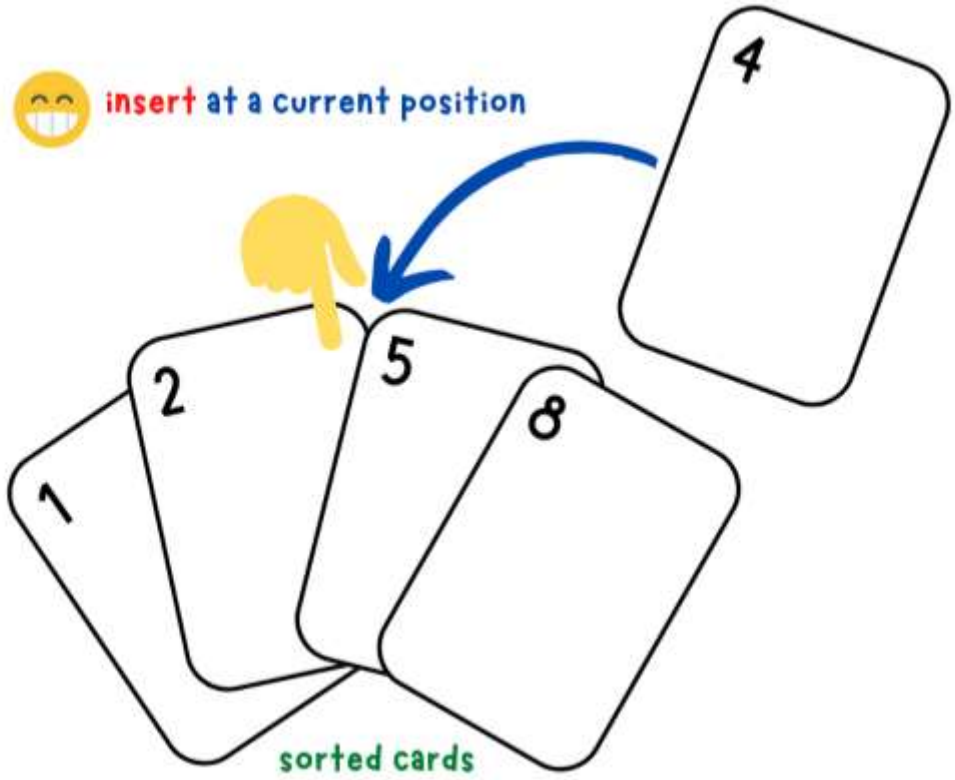
1. Compare two adjacent balls.
2. If the first one is larger, swap their positions.
3. Repeat the process until the cards are in order.

Section 4 Insertion Sort

4-1 What is Insertion Sort?

Imagine that you collect a deck of cards in your hand, and you add a new card in the right place right away so that the cards stay sorted.

That's Insertion Sort: you take a new item and put it in its right place in the sorted part of the list.



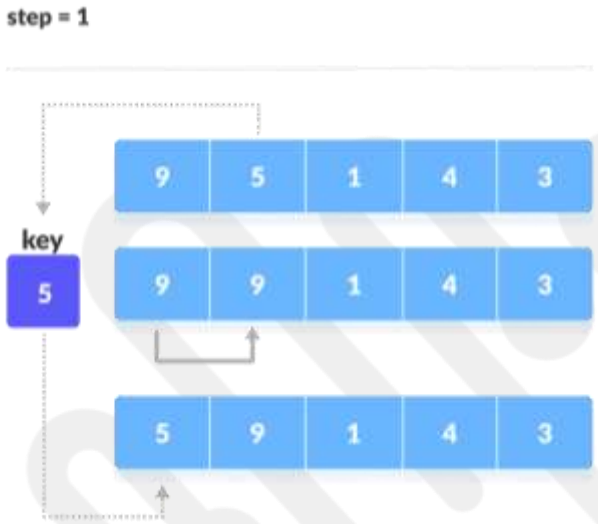
4-2 Working of Insertion Sort

Suppose we need to sort the following array.

Initial array

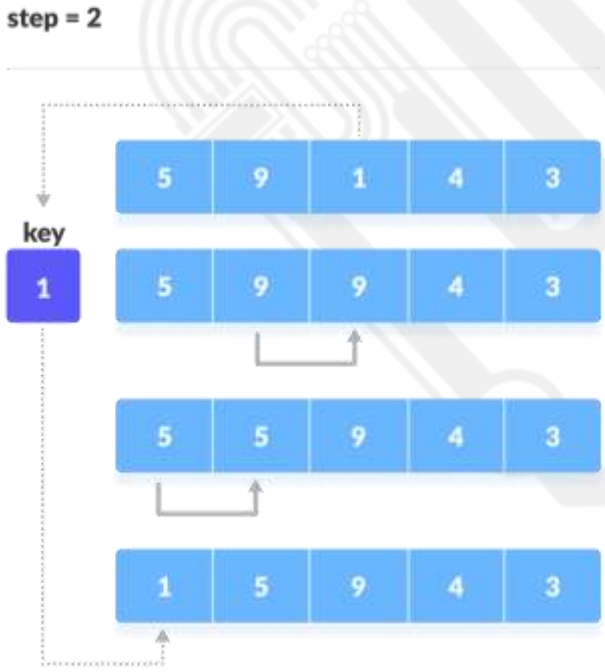
1. The first element in the array is assumed to be sorted. Take the second element and store it separately in key.

Compare key with the first element. If the first element is greater than key, then key is placed in front of the first element.



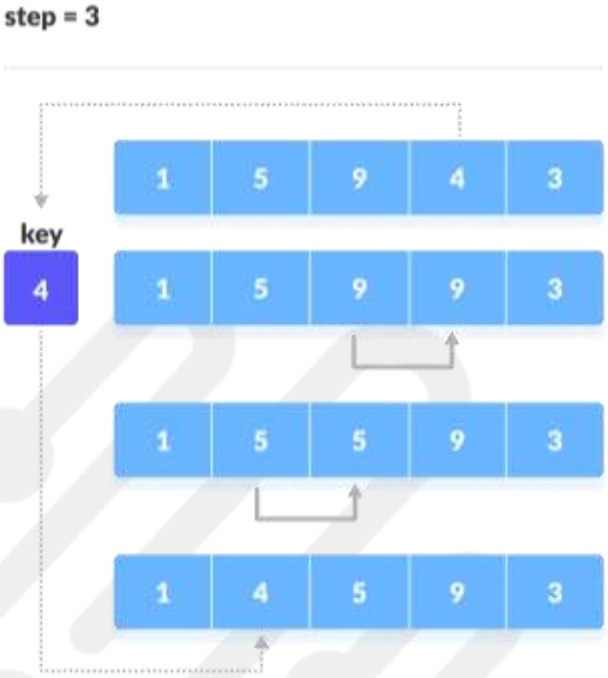
If the first element is greater than key, then key is placed in front of the first element.

2. Now, the first two elements are sorted.
   Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.

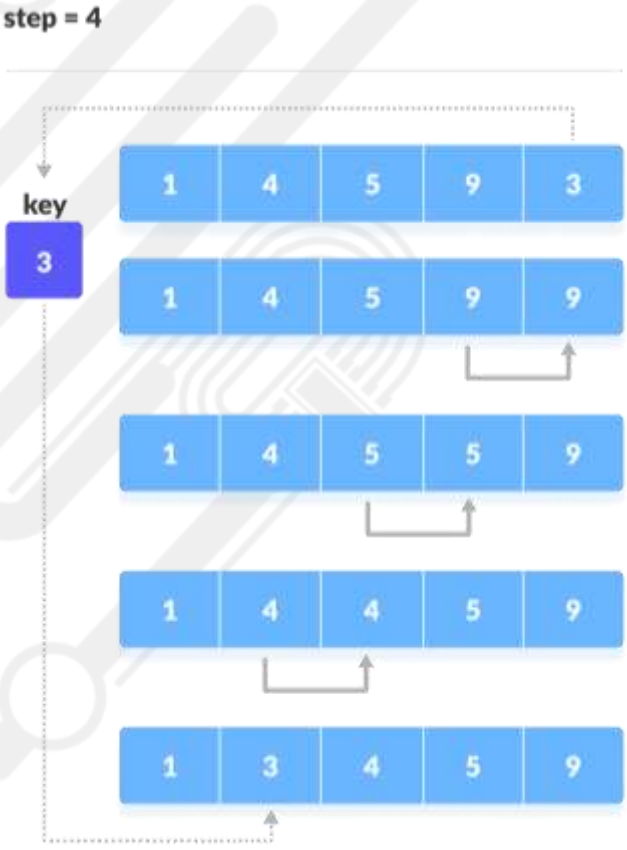


Place 1 at the beginning

3. Similarly, place every unsorted element at its correct position.



Place 4 behind 1



Place 3 behind 1 and the array is sorted

Section 5 Terminology

- **Algorithm:** A set of well-defined instructions for solving a problem or performing a task.
- **Sorting:** Arranging a collection of items in a specific order (e.g., ascending or descending).
- **Selection:** Choosing a specific item from a collection based on a certain criterion (e.g., smallest, largest).
- **Selection Sort:** A sorting algorithm that repeatedly finds the minimum element from the unsorted part and puts it at the beginning.
- **Bubble Sort:** A sorting algorithm that repeatedly compares adjacent elements and swaps them if they are in the wrong order.
- **Insertion Sort:** A sorting algorithm that builds the final sorted array one item at a time by repeatedly taking an element from the input and inserting it into the correct position in the already sorted part.

DO I KNOW THIS ALREADY ?

Dear learner: Give The Scientific Term

41	The algorithm that repeatedly compares and swaps adjacent elements is called _____.	
42	The sorting algorithm that repeatedly selects the smallest remaining element is _____.	
43	The algorithm where you take a new item and put it in its right place in the sorted part of the list is _____.	
44	A set of steps to solve a problem is called an _____.	
45	Arranging elements in a specific order is the purpose of a _____ algorithm.	

46	Finding a specific element in a collection is the goal of a _____ algorithm.	

Essay questions- clear and readable handwriting

47	**Question**	**What is the main difference between a selection algorithm and a sorting algorithm?**
	Answer	
48	**Question**	**Describe how bubble sort works in simple terms.**
	Answer	
49	**Question**	**What is the key idea behind insertion sort?**
	Answer	
50	**Question**	**How does selection sort find the minimum element in each iteration?**
	Answer	
51	**Question**	**Why is bubble sort called "bubble sort"?**
	Answer	
52	**Question**	**What is the practical activity used to illustrate selection sort?**
	Answer	Arranging numbered papers.
53	**Question**	**What is the purpose of the swap operation in sorting algorithms?**
	Answer	To exchange the positions of two elements.

54 Match the following with their correct descriptions:

1	Bubble Sort	**A**	Selects the smallest remaining element	
2	Selection Sort	**B**	Inserts elements into their correct position	
3	Insertion Sort	**C**	Compares and swaps adjacent elements	
4	Sorting Algorithm	**D**	Finds a specific element	
5	Selection Algorithm	**E**	Arranges elements in order	

Complete the Sentence Questions:

55	_____ algorithms arrange elements in a specific order.	
56	_____ algorithms are used to find a specific element.	
57	Bubble sort repeatedly compares and _____ adjacent elements.	
58	Selection sort places the smallest remaining element in its _____ position.	
59	Insertion sort inserts each element into its _____ position in the sorted part of the list.	
60	Bubble sort gets its name from the way smaller elements _____ to the top.	

Lesson	4
Name	**Stack-Queue and Tree**
Goals / Outcomes	**By the end of this lesson, students should be able to:** ➢ **Remembering** 1. Implement selection sort, bubble sort, and insertion sort in code to sort arrays. 2. Use sorting algorithms to arrange data in ascending or descending order in real-world scenarios. 3. Demonstrate how to trace and debug sorting algorithms step-by-step. ➢ **Understanding** 1. Implement selection sort, bubble sort, and insertion sort in code to sort arrays. 2. Use sorting algorithms to arrange data in ascending or descending order in real-world scenarios. 3. Demonstrate how to trace and debug sorting algorithms step-by-step. ➢ **Applying** 1. Implement selection sort, bubble sort, and insertion sort in code to sort arrays. 2. Use sorting algorithms to arrange data in ascending or descending order in real-world scenarios. 3. Demonstrate how to trace and debug sorting algorithms step-by-step. ➢ **Analyzing** 1. Implement selection sort, bubble sort, and insertion sort in code to sort arrays. 2. Use sorting algorithms to arrange data in ascending or descending order in real-world scenarios. 3. Demonstrate how to trace and debug sorting algorithms step-by-step. ➢ **Evaluating** 4. Implement selection sort, bubble sort, and insertion sort in code to sort arrays. 5. Use sorting algorithms to arrange data in ascending or descending order in real-world scenarios. 6. Demonstrate how to trace and debug sorting algorithms step-by-step. ➢ **Creating** 4. Implement selection sort, bubble sort, and insertion sort in code to sort arrays.

	5. Use sorting algorithms to arrange data in ascending or descending order in real-world scenarios. 6. Demonstrate how to trace and debug sorting algorithms step-by-step.	
Knowledge	**Code**	**Description**
	TPK23	Data structures
	TPK24	Algorithms
Skill	**Code**	**Description**
	TPC6.1	Using data structures for code optimization and problem-solving
	TPC6.2	Solving problems using algorithms
	TPC6.3	Create classes with custom methods, including initializers and decorated properties
	TPC6.4	Analyze object-based design patterns
	TPC6.5	Handle and produce errors (builtin or custom) to process or signal failure

Lesson 4: Stack-Queue and Tree

IN THIS LESSON WE LEARN:

- **stack**
- **Queue**
- **Tree**

Section 1 stack

1-1 What is a stack?

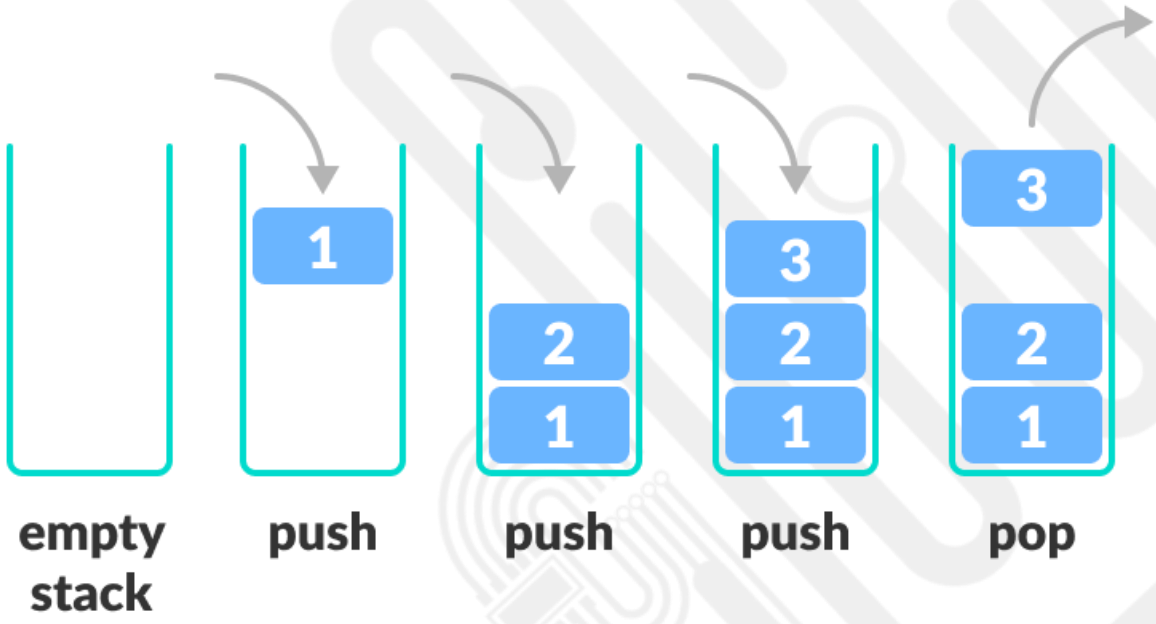
A stack is like a pile of things, like a set of plates on top of each other.

When you add a new plate, you put it on top.

When you want to take a plate, you take the one on top first.

This rule is called: LIFO (Last In, First Out)

The last thing you add is the first thing you will take.



A stack is a linear data structure that follows the principle of **Last In First Out (LIFO)**. This means the last element inserted inside the stack is removed first.

You can think of the stack data structure as the pile of plates on top of another.

Stack representation similar to a pile of plate

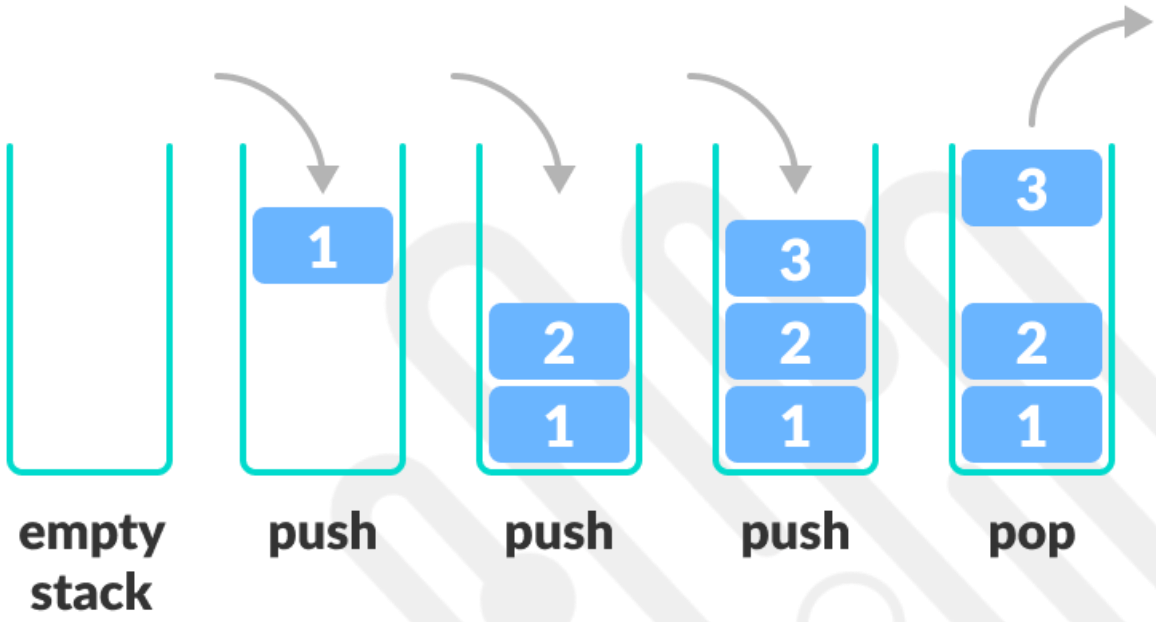
Here, you can:

- Put a new plate on top
- Remove the top plate

And, if you want the plate at the bottom, you must first remove all the plates on top. This is exactly how the stack data structure works.

1-2 LIFO Principle of Stack

In programming terms, putting an item on top of the stack is called **push** and removing an item is called **pop**.



Stack Push and Pop Operations

In the above image, although item **3** was kept last, it was removed first. This is exactly how the **LIFO (Last In First Out) Principle** works.

We can implement a stack in any programming language like C, C++, Java, Python or C#, but the specification is pretty much the same.

1-3 Basic Operations of Stack

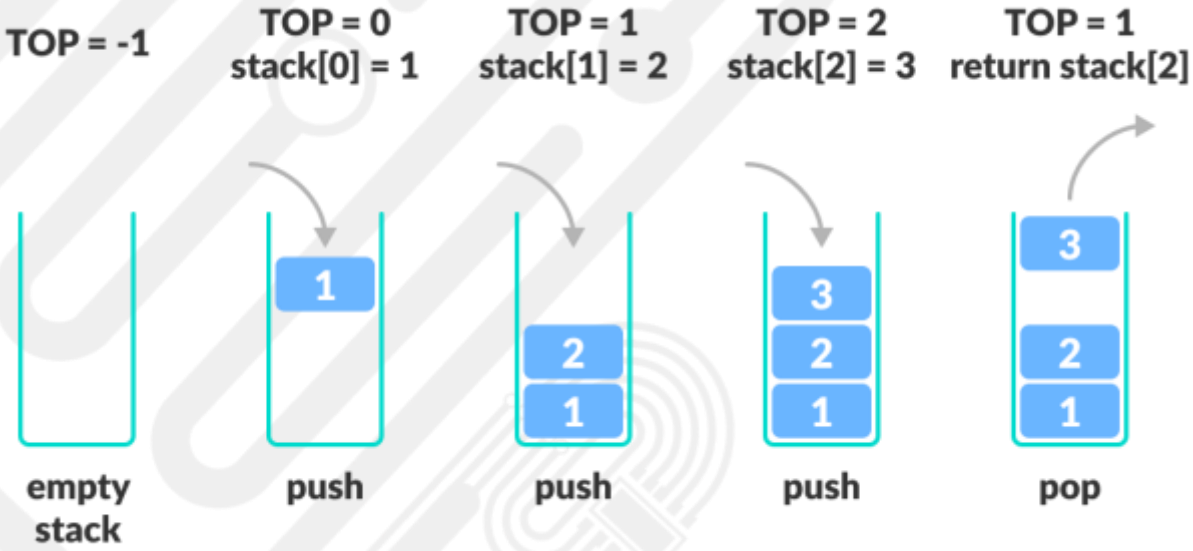
There are some basic operations that allow us to perform different actions on a stack.

- **Push**: Add an element to the top of a stack
- **Pop**: Remove an element from the top of a stack
- **IsEmpty**: Check if the stack is empty
- **IsFull**: Check if the stack is full
- **Peek**: Get the value of the top element without removing it

1-4 Working of Stack Data Structure

The operations work as follows:

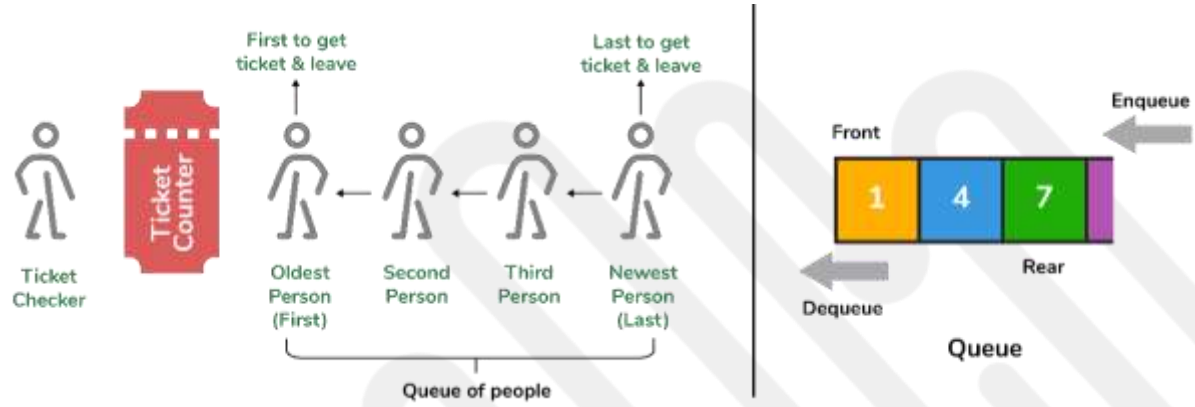
1. A pointer called TOP is used to keep track of the top element in the stack.
2. When initializing the stack, we set its value to -1 so that we can check if the stack is empty by comparing TOP == -1.
3. On pushing an element, we increase the value of TOP and place the new element in the position pointed to by TOP.
4. On popping an element, we return the element pointed to by TOP and reduce its value.
5. Before pushing, we check if the stack is already full
6. Before popping, we check if the stack is already empty



Working of Stack Data Structure

Section 2 Queue

Imagine you are standing in a queue to buy something. The first person in the queue is the first person to be served.



Rules:

FIFO (First In, First Out): The first thing in is the first thing out.

2-1 Queue Data Structure

A queue is a useful data structure in programming. It is similar to the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets the ticket.

Queue follows the **First In First Out (FIFO)** rule - the item that goes in first is the item that comes out first.

FIFO Representation of Queue

In the above image, since 1 was kept in the queue before 2, it is the first to be removed from the queue as well. It follows the **FIFO** rule.

In programming terms, putting items in the queue is called **enqueue**, and removing items from the queue is called **dequeue**.

We can implement the queue in any programming language like C, C++, Java, Python or C#, but the specification is pretty much the same.

2-2 Basic Operations of Queue

A queue is an object (an abstract data structure - ADT) that allows the following operations:

- **Enqueue**: Add an element to the end of the queue
- **Dequeue**: Remove an element from the front of the queue
- **IsEmpty**: Check if the queue is empty
- **IsFull**: Check if the queue is full
- **Peek**: Get the value of the front of the queue without removing it

2-3 Working of Queue

Queue operations work as follows:

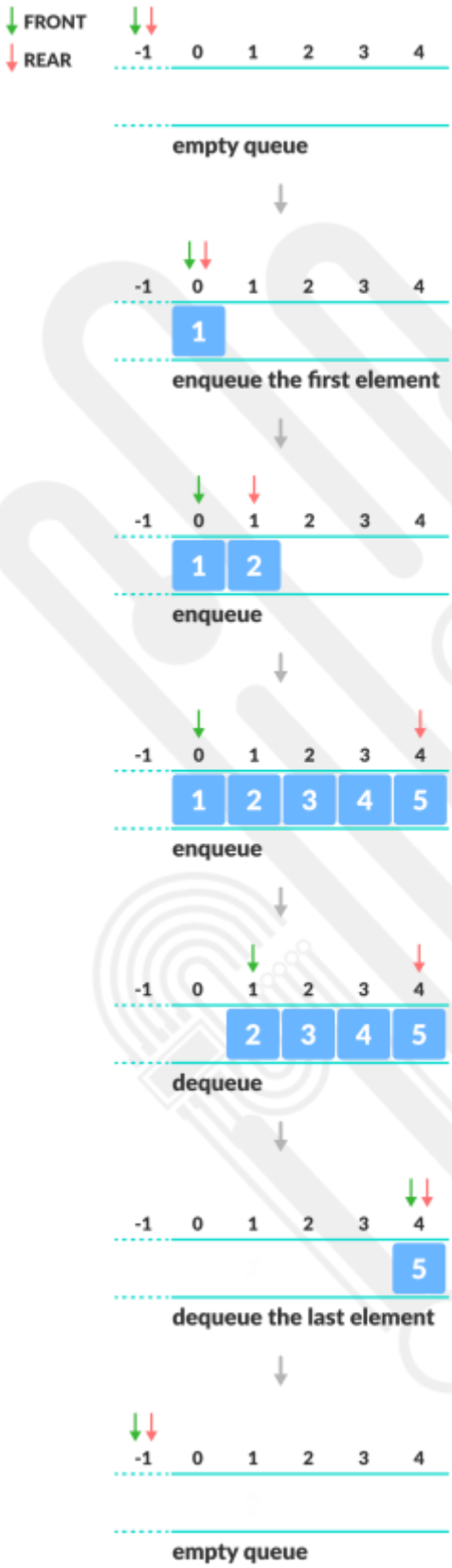
- two pointers FRONT and REAR
- FRONT track the first element of the queue
- REAR track the last element of the queue
- initially, set value of FRONT and REAR to -1

Enqueue Operation

- check if the queue is full
- for the first element, set the value of FRONT to 0
- increase the REAR index by 1
- add the new element in the position pointed to by REAR

Dequeue Operation

- check if the queue is empty
- return the value pointed by FRONT
- increase the FRONT index by 1
- for the last element, reset the values of FRONT and REAR to -1

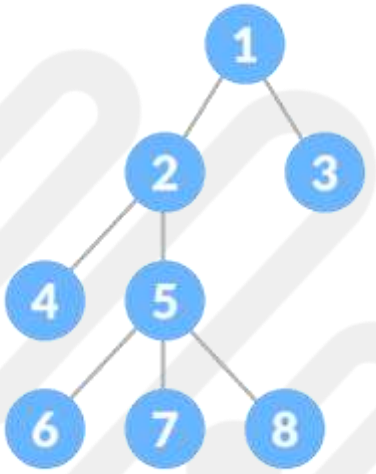


Enqueue and Dequeue Operations

Section 3 Tree

Tree Data Structure

A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges.



A Tree

Why Tree Data Structure?

Other data structures such as arrays, linked list, stack, and queue are linear data structures that store data sequentially. To perform any operation in a linear data structure, the time complexity increases with the increase in the data size. But it is not acceptable in today's computational world.

Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure.

Tree Terminologies

Node

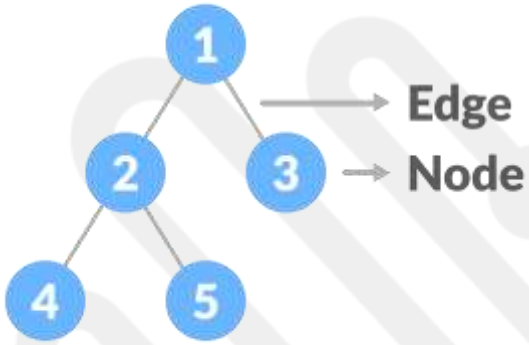
A node is an entity that contains a key or value and pointers to its child nodes.

The last nodes of each path are called **leaf nodes or external nodes** that do not contain a link/pointer to child nodes.

The node having at least a child node is called an **internal node**.

Edge

It is the link between any two nodes.



Nodes and edges of a tree

Root

It is the topmost node of a tree.

Height of a Node

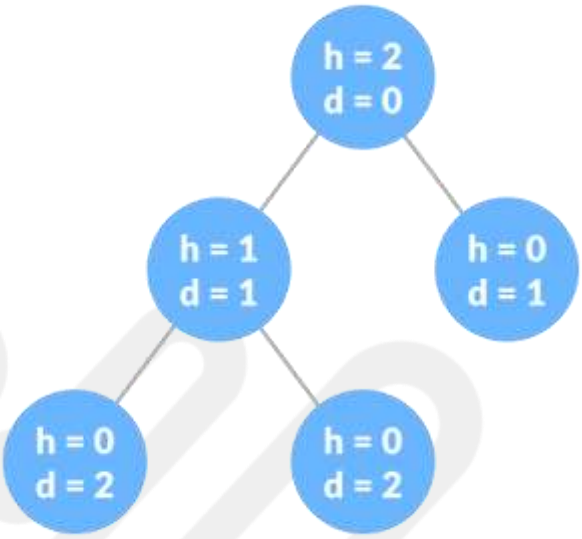
The height of a node is the number of edges from the node to the deepest leaf (ie. the longest path from the node to a leaf node).

Depth of a Node

The depth of a node is the number of edges from the root to the node.

Height of a Tree

The height of a Tree is the height of the root node or the depth of the deepest node.



Height and depth of each node in a tree

Degree of a Node

The degree of a node is the total number of branches of that node.

Forest

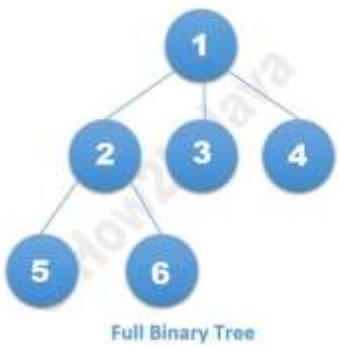
A collection of disjoint trees is called a forest.

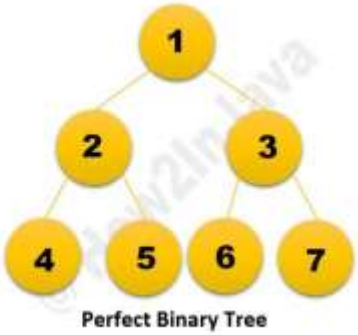
Creating a forest from a tree

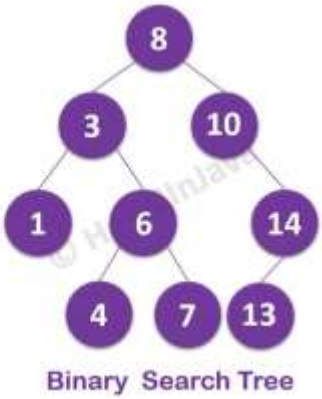
You can create a forest by cutting the root of a tree.

Types of Tree

- Binary Tree
- Binary Search Tree
- AVL Tree
- B-Tree







Section 4 Comparison between Stack, Queue and Tree

Feature	**Stack**	**Queue**	**Tree**
Arrangement	LIFO	FIFO	Hierarchical structure.
Usage	Backtracking (such as backtracking in texts).	Queue (like reservation).	Organize data (eg family tree).

Section 5 Terminology

Stack-Related Terms:

- **Stack:**
  A linear data structure that follows the Last In First Out (LIFO) principle, meaning the last element added is the first one to be removed.
- **LIFO (Last In First Out):**
  The rule that the most recently added item is the first one to be removed.
- **Push:**
  The operation of adding an element to the top of the stack.
- **Pop:**
  The operation of removing the element from the top of the stack.
- **Peek:**
  The operation of retrieving the top element of the stack without removing it.
- **IsEmpty:**
  A check to determine if the stack has no elements.
- **IsFull:**
  A check to determine if the stack has reached its capacity (in case of a fixed-size stack).
- **TOP Pointer:**
  A pointer that indicates the current top element in the stack. It is typically initialized to -1 to indicate an empty stack.

Queue-Related Terms:

- **Queue:**
  A linear data structure that follows the First In First Out (FIFO)

principle, meaning the first element added is the first one to be removed.

- **FIFO (First In First Out):**
  The rule that the first element added is the first one to be removed.
- **Enqueue:**
  The operation of adding an element to the end (rear) of the queue.
- **Dequeue:**
  The operation of removing an element from the front of the queue.
- **Peek:**
  The operation of viewing the front element of the queue without removing it.
- **IsEmpty:**
  A check to determine if the queue has no elements.
- **IsFull:**
  A check to determine if the queue has reached its capacity (in case of a fixed-size queue).
- **FRONT and REAR Pointers:**
  - **FRONT: Tracks the first element of the queue.**
  - **REAR: Tracks the last element of the queue.**

Both are often initialized to -1 to indicate that the queue is empty.

Tree-Related Terms:

- **Tree:**
  A nonlinear hierarchical data structure consisting of nodes connected by edges.

- **Node:**
  An individual element of a tree that contains a value (or key) and may have pointers to child nodes.
- **Edge:**
  The link or connection between two nodes in a tree.
- **Root:**
  The topmost node of a tree, which serves as the starting point for any tree traversal.
- **Leaf Node (External Node):**
  A node that does not have any child nodes.
- **Internal Node:**
  A node that has at least one child node.
- **Height of a Node:**
  The number of edges on the longest downward path between that node and a leaf. Alternatively, the height of a tree is the height of its root node.
- **Depth of a Node:**
  The number of edges from the root node to the particular node.
- **Degree of a Node:**
  The number of children (branches) a node has.
- **Forest:**
  A collection of disjoint trees, which can be created, for example, by cutting the root of a tree.

DO I KNOW THIS ALREADY ?

Dear learner: Give The Scientific Term

61	The data structure that operates on the principle of Last In First Out is called a _____.	Stack
62	In a stack, the operation to add an element is known as _____.	push
63	The operation used to remove the top element from a stack is called _____.	pop()
64	The principle that describes the order of removal in a stack is _____.	LIFO
65	The pointer that indicates the current top element in a stack is known as _____.	TOP
66	The data structure where the first element added is the first element removed is called a _____.	Queue
67	The process of adding an element to the end of a queue is known as _____.	Enqueue
68	Removing an element from the front of a queue is called _____.	Dequeue
69	The pointer that tracks the front of the queue is called _____.	Front
70	In queue operations, the pointer that indicates the last element is called _____.	REAR
71	A hierarchical data structure composed of nodes connected by edges is known as a _____.	TREE
72	The topmost node in a tree is referred to as the _____.	
73	A node in a tree that has no children is called a _____.	
74	The number of edges on the longest path from a node to a leaf is known as the node's _____.	

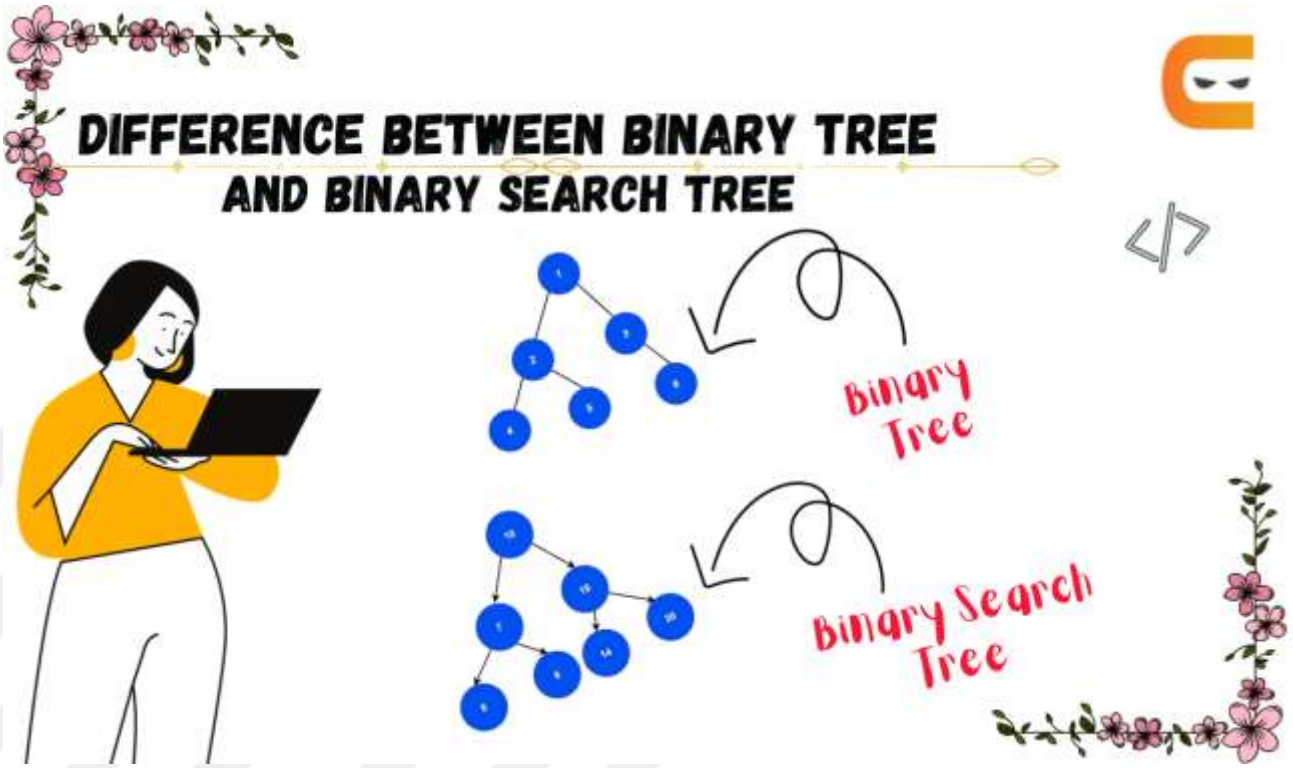
75	The number of edges from the root to a specific node is called the node's _____.	
76	A tree data structure where each node has at most two children is called a _____ tree.	
77	A specialized tree that maintains balance, ensuring that the height difference between subtrees is minimal, is known as an _____ Tree.	
78	In the context of trees, a _____ is a collection of disjoint trees.	
79	The connection between two nodes in a tree is known as an _____.	
80	The process of viewing the top element of a stack without removing it is called _____.	

Lesson	**5**
Name	**Binary Tree-Graph and Binary Search**
Goals / Outcomes	**By the end of this lesson, students should be able to:** ➢ **Remembering** 1. Define what a binary tree, graph, and binary search are. 2. Identify the key components of a binary tree (e.g., root, nodes, leaves) and a graph (e.g., vertices, edges). 3. Recall the steps involved in the binary search algorithm. 4. Recognize use cases for binary trees, graphs, and binary search in real-world applications. ➢ **Understanding** 1. Explain how data is organized in a binary tree and how traversal works. 2. Describe the structure of a graph and differentiate between directed and undirected graphs. 3. Understand the process of binary search and why it is efficient for sorted datasets. 4. Illustrate the differences between tree-based and graph-based data structures.. ➢ **Applying** 1. Implement a binary tree and perform common operations such as insertion, deletion, and traversal. 2. Create and manipulate a graph using adjacency lists or matrices. 3. Write a program to perform binary search on a sorted list of elements. 4. Solve problems involving hierarchical data (binary tree) and networked data (graph). ➢ **Analyzing** 1. Compare the time and space complexity of binary tree operations, graph traversals, and binary search.

	2. Analyze the efficiency of binary search versus linear search for different datasets. 3. Examine the suitability of graphs versus binary trees for solving specific problems. 4. Break down the traversal algorithms (e.g., BFS, DFS) used in graphs. ➢ **<u>Evaluating</u>** 1. Evaluate the performance of binary trees and graphs in real-world applications such as databases and navigation systems. 2. Assess the efficiency of binary search for large-scale datasets. 3. Critique different implementations of graphs (e.g., adjacency list vs. matrix) for their impact on performance. 4. Judge the balance of a binary tree and its effect on traversal efficiency ➢ **<u>Creating</u>** 1. Design a program to visualize a binary tree structure and its traversal methods. 2. Create a graph-based solution to model a network or a relationship system. 3. Develop a practical application that uses binary search for fast lookups (e.g., finding a word in a dictionary). 4. Build an interactive tool to demonstrate the differences between binary tree traversals and graph searches (BFS, DFS).	
Knowledge	**Code**	**Description**
	TPK23	Data structures
	TPK24	Algorithms
Skill	**Code**	**Description**
	TPC6.1	Using data structures for code optimization and problem-solving

	TPC6.2	Solving problems using algorithms
	TPC6.3	Create classes with custom methods, including initializers and decorated properties
	TPC6.4	Analyze object-based design patterns
	TPC6.5	Handle and produce errors (builtin or custom) to process or signal failure

Lesson 5: Binary Tree-Graph and Binary Search

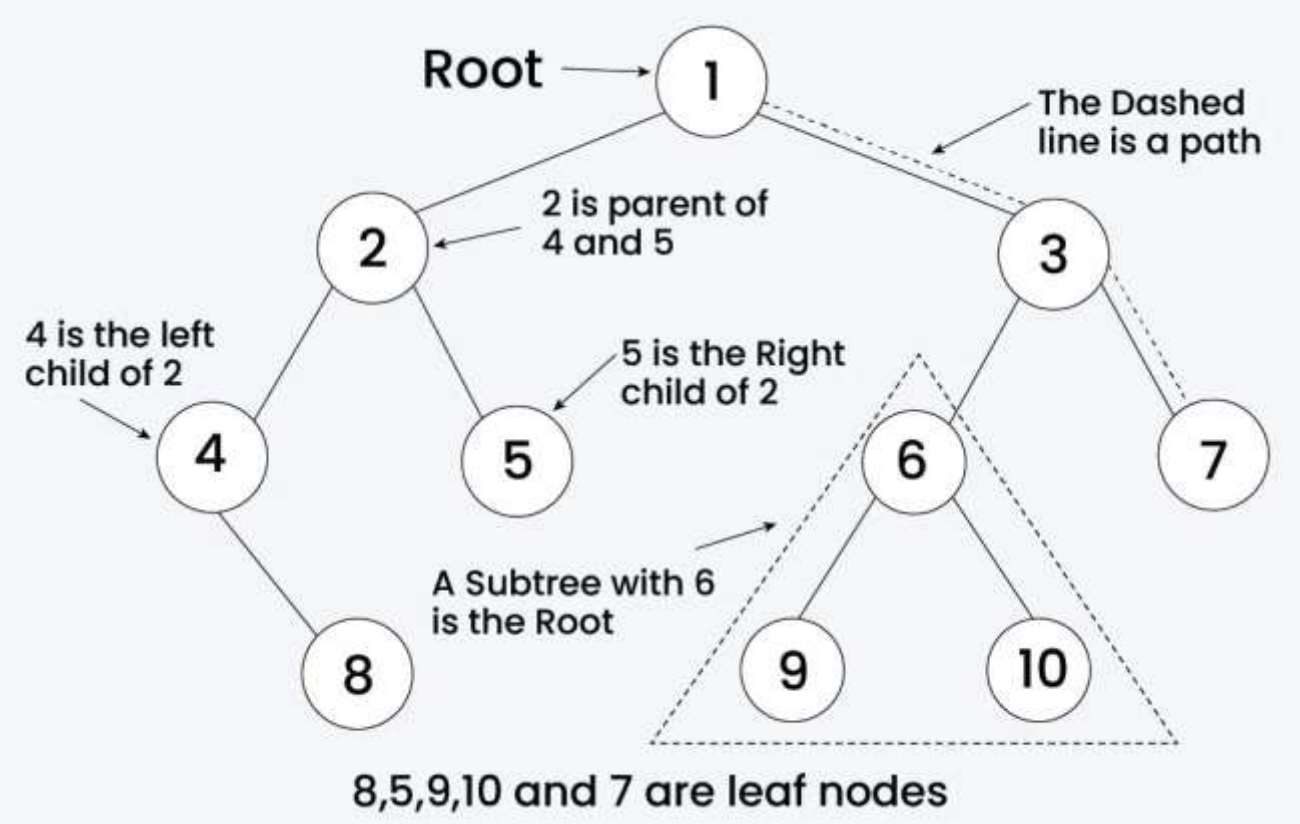


IN THIS LESSON WE LEARN:

- **Binary Tree**
- **Graph**
- **Binary Search**

Section 1: Binary Tree

A binary tree is a fun and simple way to organize information, just like a family tree or a flowchart! It's called "binary" because each "node" (or point) in the tree can have up to **two branches** (or children)



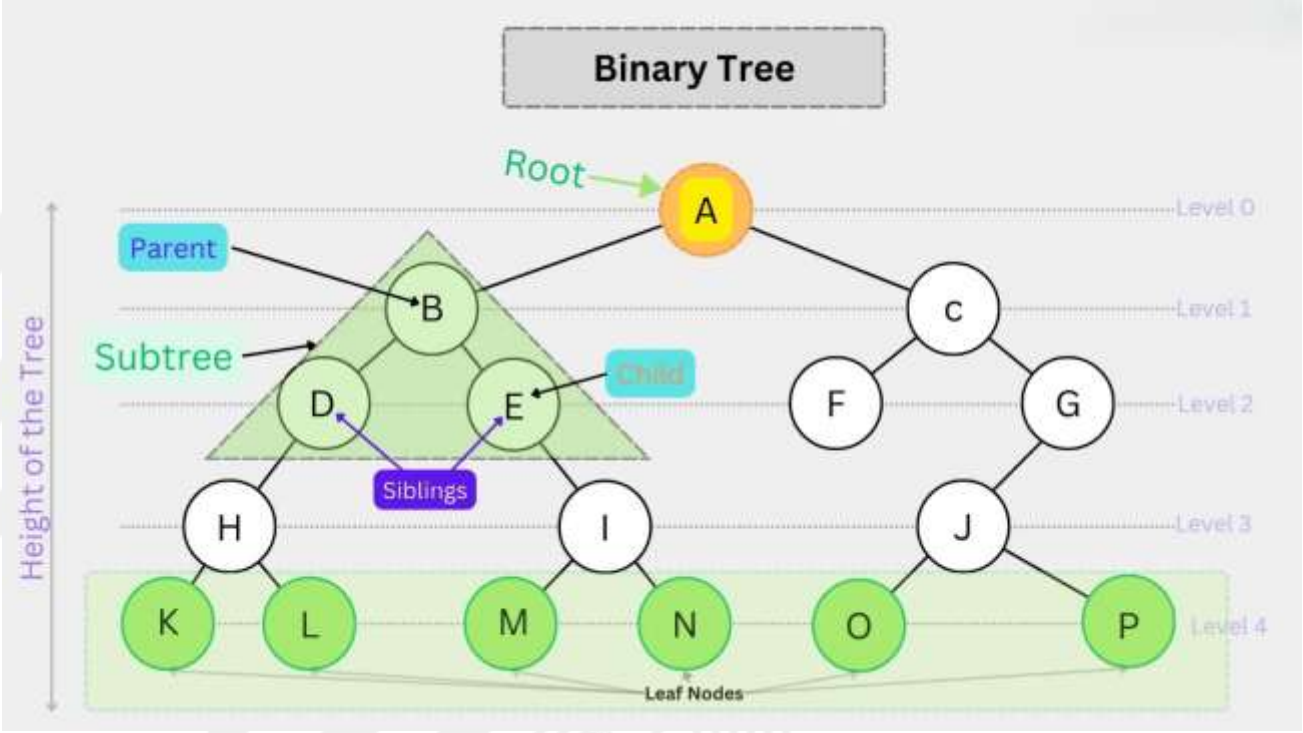
1-1 What is a Binary Tree?

Imagine a tree with branches, but instead of leaves, each branch has a **question** or a **choice**. For example:

- Start at the top of the tree (this is called the **root**).
- The root asks a question, like, "Is it an animal?"
- If the answer is **YES**, you go to the left branch.
- If the answer is **NO**, you go to the right branch.
- Each branch leads to another question or a final answer.

A binary tree is a data structure that **consists of:**

- **Node:** Each individual in the tree.
- **Root:** The great-grandfather (or great-grandmother).
- **Branches:** The relationships between individuals (father and children).
- **Leaves:** The individuals who do not have children.



1-2 Why do we use them?

- It's like a **decision-making game** where you answer questions to find the right answer.
- It helps organize information in a clear and logical way.
- Computers use binary trees to solve problems quickly!

1-3 Example: Animal Guessing Game

Let's make a simple binary tree to guess an animal:

1. **Root Question**: Does it live in water?

- **Left Branch (YES)**: Is it a fish?
  - **Left Branch (YES)**: It’s a fish!
  - **Right Branch (NO)**: Is it a whale?
    - **Left Branch (YES)**: It’s a whale!
    - **Right Branch (NO)**: It’s a dolphin!
- **Right Branch (NO)**: Does it have wings?
  - **Left Branch (YES)**: Is it a bird?
    - **Left Branch (YES)**: It’s a bird!
    - **Right Branch (NO)**: It’s a bat!
  - **Right Branch (NO)**: Does it have four legs?
    - **Left Branch (YES)**: It’s a dog!
    - **Right Branch (NO)**: It’s a snake!

1-4 Practical activity:

Draw Your Own Binary Tree!

Grab a piece of paper and draw your own binary tree. Start with a question at the top, then add two branches for "YES" and "NO." Keep adding questions until you reach the final answers. You can make it about animals, sports, food, or anything you like!

Section 2: Graph

2-1 What is a graph?

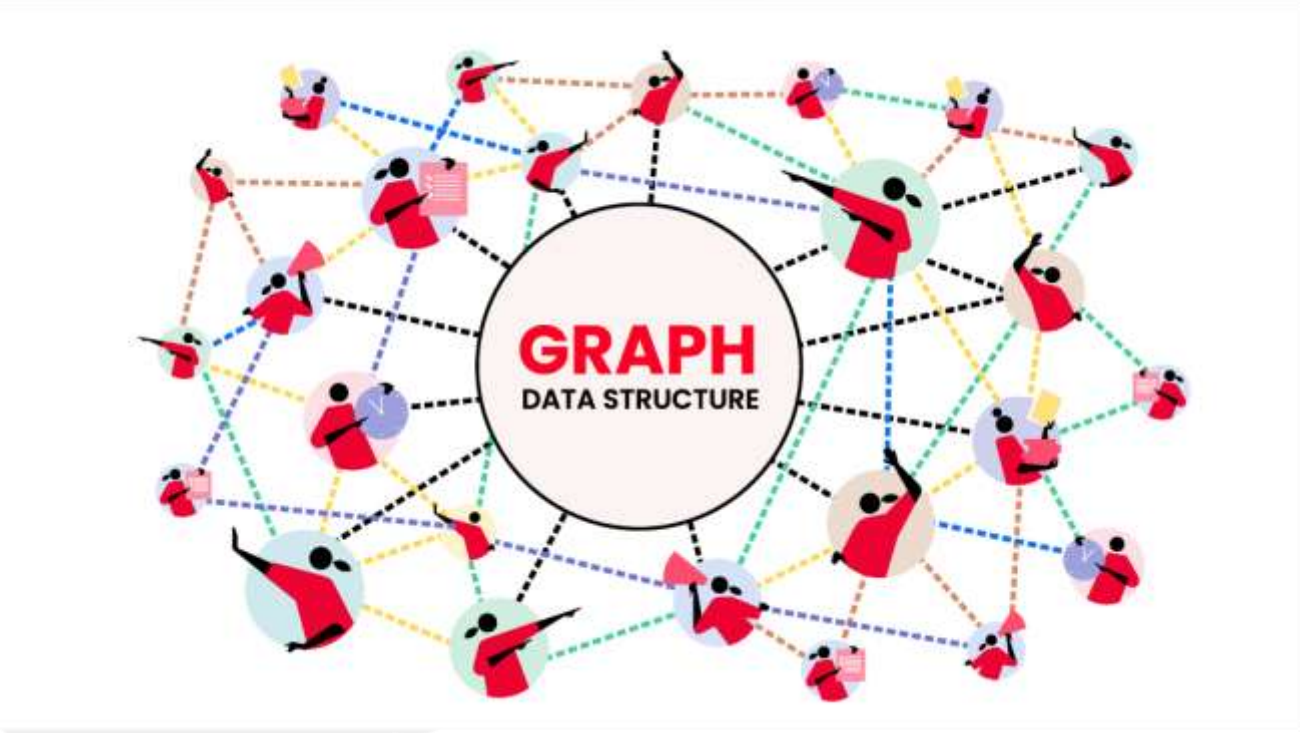
A **graph** is a way to show how things are connected. It's like a map of relationships between different objects or points. Graphs are super cool because they can be used to solve puzzles, plan routes, or even understand how friends are connected on social media!

Imagine a map of a city. Every place in the city is a "node," and the roads connecting the places are "edges." A graph is a collection of nodes and the edges connecting them.

A graph is made up of:

A node is every place in the city.

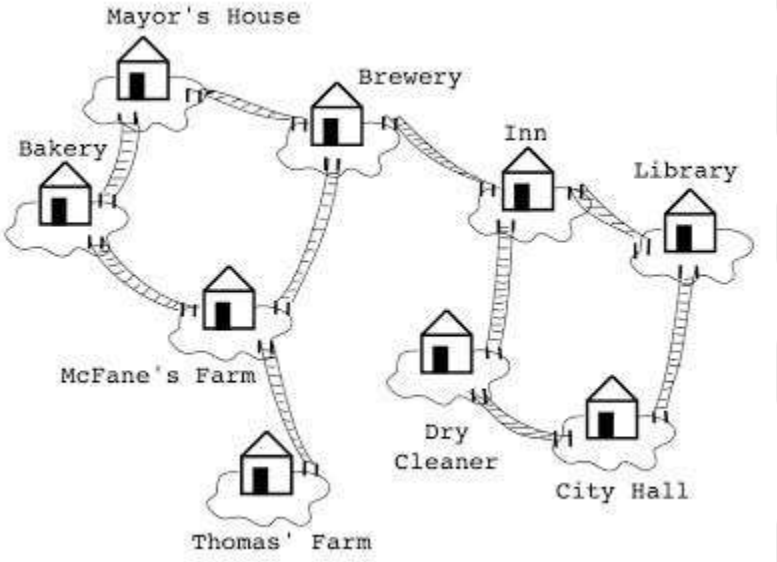
An edge is a road connecting two places.



2-2 Why do we use them?

We use graphs to represent relationships between things, such as:

- **Bus and train schedules.**
- **Road Maps**: Cities are nodes, and roads are edges or Road network between two cities.
- **Social Networks**: People are nodes, and friendships are edges.
- **Computer Networks**: Computers are nodes, and connections (like Wi-Fi or cables) are edges.



2-4 Activity:

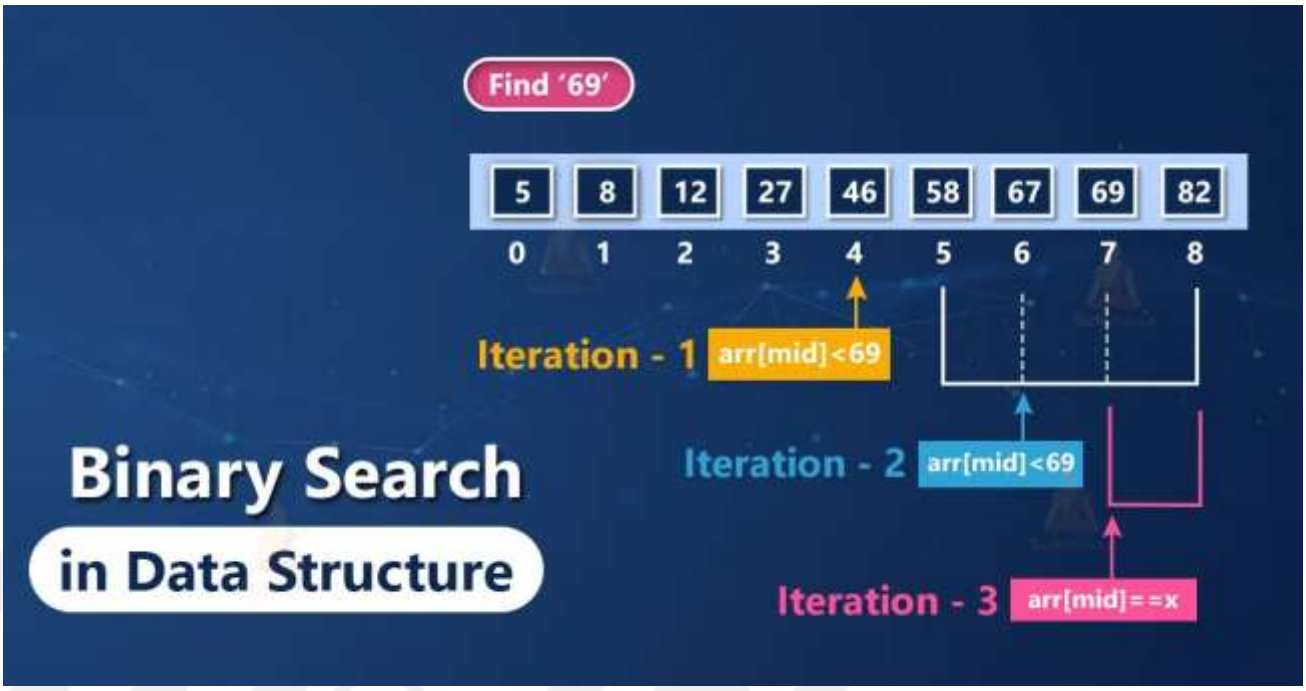
Fun Activity: Make Your Own Graph!

1. Draw circles (nodes) to represent people, places, or things.
2. Draw lines (edges) to connect them and show relationships.
3. Add labels to explain what the connections mean.

For example:

- Draw a graph of your family and show who is friends with whom.
- Draw a graph of your favorite places in your neighborhood and how they're connected by roads.

Section 3: Binary Search



3-1 What is binary search?

Binary Search is a super-smart way to find something quickly in a sorted list. It's like playing a guessing game where you cut the possibilities in half every time you make a guess. Let me explain it in a simple way!

Imagine you have a large textbook, and you want to find a specific word. Instead of searching from the first page to the end, you can use a faster method called binary search.

3-2 Binary Search Working

Binary Search Algorithm can be implemented in two ways which are discussed below.

1. Iterative Method
2. Recursive Method

The general steps for both methods are discussed below.

1. The array in which searching is to be performed is:

Initial array

Let x = 4 be the element to be searched.

2. Set two pointers low and high at the lowest and the highest positions respectively.

Setting pointers

3. Find the middle position mid of the array ie. mid = (low + high)/2 and arr[mid] = 6.

Mid element

4. If x == arr[mid], then return mid. Else, compare the element to be searched with arr[mid].

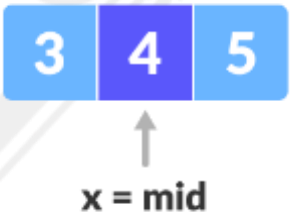
5. If x > arr[mid], compare x with the middle element of the elements on the right side of arr[mid]. This is done by setting low to low = mid + 1.
6. Else, compare x with the middle element of the elements on the left side of arr[mid]. This is done by setting high to high = mid - 1.

Finding mid element

7. Repeat steps 3 to 6 until low meets high.

Mid element

8. x = 4 is found.



Found

3-3 Why is Binary Search Awesome?

- It's **super-fast**! Instead of checking every single item, you cut the list in half each time.
- It only works on **sorted lists**, so make sure your list is in order first.

3-4 Fun Activity: Play the Guessing Game!

You can play a binary search game with a friend:

1. Think of a number between 1 and 100.
2. Your friend guesses the middle number (50).
3. If the guess is too high, you say "lower." If it's too low, you say "higher."
4. Keep guessing the middle number of the new range until you find the right number!

Section 4 Summary:

Binary tree: A structure like a family tree.

Graph: A network of connected points.

Binary search: An efficient way to search through ordered lists.

Section 5 Terminology

Section 1: Binary Tree

1. **Binary Tree**: A data structure where each node has up to two branches (or children).
2. **Node**: A point in the binary tree that contains data or a question.
3. **Root**: The topmost node in a binary tree (the starting point).
4. **Branches**: The connections between nodes (left and right children).
5. **Leaves**: Nodes that do not have any children (the end points of the tree).
6. **Decision-Making Game**: A way to use a binary tree to answer questions and reach a final answer.

Section 2: Graph

1. **Graph**: A way to show connections between objects or points.
2. **Node (or Vertex)**: A point in the graph that represents an object (e.g., a person, city, or computer).
3. **Edge (or Line)**: A connection between two nodes that shows a relationship.
4. **Undirected Graph**: A graph where edges have no direction (e.g., friendships).
5. **Directed Graph**: A graph where edges have a direction (e.g., one-way roads).
6. **Social Networks**: A real-life example of a graph where people are nodes and friendships are edges.
7. **Road Maps**: A real-life example of a graph where cities are nodes and roads are edges.
8. **Computer Networks**: A real-life example of a graph where computers are nodes and connections (like Wi-Fi) are edges.

Section 3: Binary Search

1. **Binary Search**: A fast way to find an item in a sorted list by repeatedly dividing the list in half.
2. **Sorted List**: A list where items are arranged in order (e.g., from smallest to largest).
3. **Middle Number**: The number in the center of the list, used as a reference point in binary search.
4. **Iterative Method**: A way to perform binary search using loops.
5. **Recursive Method**: A way to perform binary search by calling the function within itself.
6. **Pointers (Low and High)**: Markers used to track the range of the list being searched.
7. **Mid**: The middle position in the current range of the list, calculated as (low + high) / 2.

Section 4: Summary

1. **Binary Tree**: A structure like a family tree with nodes and branches.
2. **Graph**: A network of connected points (nodes and edges).
3. **Binary Search**: An efficient way to search through ordered lists by dividing them in half.

DO I KNOW THIS ALREADY ?

Dear learner: Choose the correct answer.

81	**What is the top node in a binary tree called?** a) Leaf b) Root c) Branch d) Edge	

82	**How many children can each node have in a binary tree?** a) 1 b) 2 c) 3 d) Unlimited	
83	**What is a graph made up of?** a) Nodes and edges b) Roots and leaves c) Questions and answers d) Numbers and letters	
84	**What does binary search do to the list each time it makes a guess?** a) Doubles the list b) Cuts the list in half c) Reverses the list d) Sorts the list	
85	**Which of the following is NOT a use of graphs?** a) Road maps b) Social networks c) Sorting numbers d) Computer networks	
86	**What is the middle number in binary search called?** a) Root b) Mid c) Leaf d) Edge	
87	**In a binary tree, what does the left branch represent?** a) YES b) NO	

	c) Maybe d) Stop	
88	**What is the term for nodes in a graph that have no connections?** a) Roots b) Leaves c) Isolated nodes d) Branches	
89	**Which of the following is required for binary search to work?** a) Unsorted list b) Sorted list c) Random list d) Empty list	
90	**What is the term for the connections between nodes in a graph?** a) Roots b) Edges c) Leaves d) Branches	
91	**What is the term for the final nodes in a binary tree that have no children?** a) Roots b) Leaves c) Branches d) Edges	
92	**What is the first step in binary search?** a) Guess the first number b) Guess the last number c) Guess the middle number d) Sort the list	
93	**Which of the following is an example of a graph?**	

	a) Family tree b) Road map c) Binary tree d) Guessing game	
94	**What is the term for the starting point in a binary tree?** a) Leaf b) Root c) Edge d) Node	
95	**What is the term for the process of dividing the list in binary search?** a) Splitting b) Cutting c) Halving d) Sorting	

Essay questions- clear and readable handwriting

96	**Question**	**How many children can each node have in a binary tree?**
	Answer	
97	**Question**	**What are the connections between nodes in a graph called?**
	Answer	
98	**Question**	**What is the first step in binary search?**
	Answer	
99	**Question**	**What is the term for nodes in a binary tree that have no children?**
	Answer	
100	**Question**	**What is required for binary search to work?**
	Answer	

Unit Terminologies

Data Structures – Ways to organize and store data efficiently for performing operations. Examples include arrays, linked lists, trees, graphs, stacks, queues, and hash tables.

Algorithm – A step-by-step procedure or set of instructions designed to perform a specific task or solve a problem.

Primitive Data Structures – Basic data types such as integers, floats, booleans, and characters.

Abstract Data Structures – More complex data structures designed for handling large and connected data, such as linked lists, trees, graphs, and hash tables.

Array – A collection of elements stored in contiguous memory locations, allowing fast access by index.

Linked List – A sequence of nodes where each node points to the next, allowing dynamic memory allocation and efficient insertions/deletions.

Stack – A data structure that follows the **LIFO (Last In, First Out)** principle, where the last inserted element is the first to be removed.

Queue – A data structure that follows the **FIFO (First In, First Out)** principle, where the first inserted element is the first to be removed.

Tree – A hierarchical data structure consisting of nodes, with a root node and child nodes connected by edges.

Graph – A collection of nodes (vertices) connected by edges, used to model relationships between objects.

Hash Table – A data structure that maps keys to values using a hash function for efficient searching and retrieval.

Sorting Algorithms – Methods to arrange data in a particular order, such as quicksort, merge sort, and bubble sort.

Search Algorithms – Techniques to find elements within a data structure, such as linear search and binary search.

Graph Algorithms – Algorithms used to traverse and analyze graphs, such as Dijkstra's algorithm for shortest path finding.

Dynamic Programming – A method for solving complex problems by breaking them down into overlapping subproblems and storing results to avoid redundant computations.

Complexity – A measure of the efficiency of an algorithm in terms of time (time complexity) and space (space complexity) required for execution.

Time Complexity – The amount of time an algorithm takes to complete based on input size.

Space Complexity – The amount of memory an algorithm requires to execute.

Big O Notation – A mathematical notation used to describe the worst-case or upper bound performance of an algorithm, such as O(n), O(log n), and $O(n^2)$.

Optimization – The process of improving an algorithm's efficiency to reduce execution time or memory usage.

Array: An ordered collection of elements, accessible by a numerical index.

Index: A unique numerical identifier assigned to each element in an array, starting from zero.

Dynamic Length: The ability of an array to grow or shrink in size.

Data Organization: The process of structuring data efficiently within an array.

Sorting & Searching: Techniques used to arrange and locate elements in an array efficiently.

Linked List: A data structure where elements (nodes) are linked together rather than stored in adjacent memory locations.

Node: A fundamental unit in a linked list that contains data and a pointer to the next node.

Pointer: A reference or link to the next node in the sequence.

Null Node: The last node in a linked list that does not point to any other node.

Hash Map: A data structure that stores key-value pairs for fast retrieval.

Key: A unique identifier used to access values in a hash map.

Value: The data stored and associated with a specific key.

Search Speed: The ability to quickly find values using keys instead of scanning an entire list.

Flexibility: The ease of adding and removing data without shifting elements.

Algorithm: A set of well-defined instructions for solving a problem or performing a task.

Sorting: Arranging a collection of items in a specific order (e.g., ascending or descending).

Selection: Choosing a specific item from a collection based on a certain criterion (e.g., smallest, largest).

Selection Sort: A sorting algorithm that repeatedly finds the minimum element from the unsorted part and puts it at the beginning.

Bubble Sort: A sorting algorithm that repeatedly compares adjacent elements and swaps them if they are in the wrong order.

Insertion Sort: A sorting algorithm that builds the final sorted array one item at a time by repeatedly taking an element from the input and inserting it into the correct position in the already sorted part.

Stack:
A linear data structure that follows the Last In First Out (LIFO) principle, meaning the last element added is the first one to be removed.

LIFO (Last In First Out):
The rule that the most recently added item is the first one to be removed.

Push:
The operation of adding an element to the top of the stack.

Pop:
The operation of removing the element from the top of the stack.

Peek:
The operation of retrieving the top element of the stack without removing it.

IsEmpty:
A check to determine if the stack has no elements.

IsFull:
A check to determine if the stack has reached its capacity (in case of a fixed-size stack).

TOP Pointer:
A pointer that indicates the current top element in the stack. It is typically initialized to -1 to indicate an empty stack.

Queue:
A linear data structure that follows the First In First Out (FIFO) principle, meaning the first element added is the first one to be removed.

FIFO (First In First Out):
The rule that the first element added is the first one to be removed.

Enqueue:
The operation of adding an element to the end (rear) of the queue.

Dequeue:
The operation of removing an element from the front of the queue.

Peek:
The operation of viewing the front element of the queue without removing it.

IsEmpty:
A check to determine if the queue has no elements.

IsFull:
A check to determine if the queue has reached its capacity (in case of a fixed-size queue).

FRONT and REAR Pointers:

FRONT: Tracks the first element of the queue.

REAR: Tracks the last element of the queue.

Both are often initialized to -1 to indicate that the queue is empty**.**

Tree:
A nonlinear hierarchical data structure consisting of nodes connected by edges.

Node:
An individual element of a tree that contains a value (or key) and may have pointers to child nodes.

Edge:
The link or connection between two nodes in a tree.

Root:
The topmost node of a tree, which serves as the starting point for any tree traversal.

Leaf Node (External Node):
A node that does not have any child nodes.

Internal Node:
A node that has at least one child node.

Height of a Node:
The number of edges on the longest downward path between that node and a leaf. Alternatively, the height of a tree is the height of its root node.

Depth of a Node:
The number of edges from the root node to the particular node.

Degree of a Node:
The number of children (branches) a node has.

Forest:
A collection of disjoint trees, which can be created, for example, by cutting the root of a tree.

Binary Tree: A data structure where each node has up to two branches (or children).

Node: A point in the binary tree that contains data or a question.

Root: The topmost node in a binary tree (the starting point).

Branches: The connections between nodes (left and right children).

Leaves: Nodes that do not have any children (the end points of the tree).

Decision-Making Game: A way to use a binary tree to answer questions and reach a final answer.

Graph: A way to show connections between objects or points.

Node (or Vertex): A point in the graph that represents an object (e.g., a person, city, or computer).

Edge (or Line): A connection between two nodes that shows a relationship.

Undirected Graph: A graph where edges have no direction (e.g., friendships).

Directed Graph: A graph where edges have a direction (e.g., one-way roads).

Social Networks: A real-life example of a graph where people are nodes and friendships are edges.

Road Maps: A real-life example of a graph where cities are nodes and roads are edges.

Computer Networks: A real-life example of a graph where computers are nodes and connections (like Wi-Fi) are edges.

Binary Search: A fast way to find an item in a sorted list by repeatedly dividing the list in half.

Sorted List: A list where items are arranged in order (e.g., from smallest to largest).

Middle Number: The number in the center of the list, used as a reference point in binary search.

Iterative Method: A way to perform binary search using loops.

Recursive Method: A way to perform binary search by calling the function within itself.

Pointers (Low and High): Markers used to track the range of the list being searched.

Mid: The middle position in the current range of the list, calculated as (low + high) / 2.

Binary Tree: A structure like a family tree with nodes and branches.

Graph: A network of connected points (nodes and edges).

Binary Search: An efficient way to search through ordered lists by dividing them in half.

DO I KNOW THIS ALREADY ?

Dear learner: Put "True" in front of the correct statement and "False" in front of the incorrect statement.

1	Data structures are only used for organizing data, not for storing it.	False

2	An algorithm is a step-by-step procedure to solve a problem.	True
3	Sorting algorithms helps arrange data in a specific order.	True
4	Primitive data structures include integers, floats, and characters.	True
5	A social network can be represented as a graph data structure.	True
6	Complexity measures the speed of an algorithm.	True
7	Dynamic programming is a technique to optimize recursive problems.	True
8	Big O notation is used to describe algorithm complexity.	True
9	An efficient algorithm always has a low time complexity.	False
10	Algorithm complexity helps in determining how the performance of an algorithm scales with the size of the input.	True

Dear learner: Choose the correct answer.

11	**What are data structures used for?** A) To compile programs B) To organize and store data efficiently C) To execute machine code D) To create new programming languages	B
12	**What is an algorithm?** A) A programming language B) A step-by-step procedure to solve a problem C) A data storage method D) A type of software	B
13	**Why is DSA important in performance-critical applications like game development?**	B

	A) It reduces code readability B) It allows faster and more efficient data processing C) It increases the complexity of programs D) It only works in C++	
14	**4. What is a data structure?** A) A way to store and organize data B) A type of algorithm C) A programming language D) A software framework	A
15	**Which of the following is NOT a primitive data structure?** A) Integer B) Float C) Graph D) Boolean	C
16	**What type of data structure is an array?** A) A collection of elements of different types B) A fixed-size collection of elements of the same type C) A collection of random numbers D) A hierarchical data structure	B
17	**Which data structure is best suited for hierarchical data?** A) Stack B) Queue C) Tree D) Array	c
18	**When choosing an appropriate data structure, what factor should be considered?** A) The number of functions available in the language B) The data type and operations required	B

	C) The name of the programming language D) The color of the interface	
19	**What is an algorithm used for?** A) Organizing data B) Solving problems efficiently C) Deleting data from memory D) Managing operating systems	B
20	**What is dynamic programming used for?** A) Solving complex problems by breaking them into smaller subproblems B) Creating new programming languages C) Organizing hierarchical data D) Converting algorithms into data structures	A

Dear learner: Choose the correct answer.

21	**What happens if you try to access an index beyond the array size?** a) It returns a null value b) It causes an error (IndexOutOfBounds) c) It loops back to the first element d) It adds a new element automatically	b
22	**Which of the following is NOT a feature of an array?** a) Elements must be of the same data type b) Fixed size (in static arrays) c) Can dynamically grow or shrink d) Provides fast lookup using indices	c
23	**What is a disadvantage of linked lists compared to arrays?** a) Cannot store multiple data types b) Cannot be dynamically allocated c) Uses more memory due to extra pointers d) Has a fixed size	c
24	**Which of the following is NOT a benefit of using a linked list?** a) Easy to insert new elements b) Easy to delete elements c) Efficient random access	C

	d) Flexible memory usage	
25	**What is the primary purpose of a Hash Map?** a) Sorting data b) Storing key-value pairs efficiently c) Managing file systems d) Implementing recursion	b
26	**Which of the following statements is TRUE about Hash Maps?** a) They use key-value pairs for quick lookups b) They always maintain a sorted order of keys c) They are slower than linked lists for searching d) They require shifting elements when adding new data	a

Essay questions- clear and readable handwriting

27	**Question**	**Why are arrays important in data structures?**
	Answer	Arrays provide efficient data organization, fast element access, and serve as a foundation for more complex structures like linked lists and trees.
28	**Question**	**How does an array differ from a linked list?**
	Answer	Arrays store data in contiguous memory locations, whereas linked lists store data in nodes connected by pointers.
29	**Question**	**Explain how a linked list works.**
	Answer	A linked list consists of nodes, each containing data and a pointer to the next node. The last node points to null.
30	**Question**	**What are the benefits of using a Hash Map instead of an array?**
	Answer	Hash Maps provide fast search speed using keys, better organization, and easy addition/removal of data.

Dear learner: Give The Scientific Term

31	An ordered collection of elements accessible by index.	**Array**
32	A data structure consisting of nodes linked together.	**Linked List**

33	A key-value based data structure is used for fast lookups.	**Hash Map**
34	A node's reference to the next node in a linked list.	**Pointer**
35	The process of arranging data in a specific order.	**Sorting**

36-40 Match the following with their correct descriptions:

1	Array	A	Reference to the next node in a linked list	1 For B
2	Linked List	B	Ordered collection with index-based access	2 For E
3	Hash Map	C	Element in a linked list containing data and a pointer	3 For D
4	Node	D	Uses key-value pairs for storage	4 For C
5	Pointer	E	Data structure with nodes and pointers	5 For A

Dear learner: Give The Scientific Term

41	The algorithm that repeatedly compares and swaps adjacent elements is called _____.	**Bubble Sort**
42	The sorting algorithm that repeatedly selects the smallest remaining element is _____.	**Selection Sort**
43	The algorithm where you take a new item and put it in its right place in the sorted part of the list is _____.	**Insertion Sort**
44	A set of steps to solve a problem is called an _____.	**Algorithm**
45	Arranging elements in a specific order is the purpose of a _____ algorithm.	**Sorting**
46	Finding a specific element in a collection is the goal of a _____ algorithm.	**Selection**

Essay questions- clear and readable handwriting

47	**Question**	**What is the main difference between a selection algorithm and a sorting algorithm?**
	Answer	• Selection algorithms find a specific element • Sorting algorithms arrange all elements.
48	**Question**	**Describe how bubble sort works in simple terms.**
	Answer	It repeatedly compares adjacent elements and swaps them if they are in the wrong order.
49	**Question**	**What is the key idea behind insertion sort?**
	Answer	To take each element and insert it into its correct position in the already sorted part of the list.
50	**Question**	**How does selection sort find the minimum element in each iteration?**
	Answer	By comparing it with all other elements in the unsorted portion.
51	**Question**	**Why is bubble sort called "bubble sort"?**
	Answer	Because smaller elements "bubble" to the top of the list.
52	**Question**	**What is the practical activity used to illustrate selection sort?**
	Answer	Arranging numbered papers.
53	**Question**	**What is the purpose of the swap operation in sorting algorithms?**
	Answer	To exchange the positions of two elements.

54 <u>Match the following with their correct descriptions:</u>

1	Bubble Sort	**A**	Selects the smallest remaining element	1 For C
2	Selection Sort	**B**	Inserts elements into their correct position	2 For A
3	Insertion Sort	**C**	Compares and swaps adjacent elements	3 For B
4	Sorting Algorithm	**D**	Finds a specific element	4 For E
5	Selection Algorithm	**E**	Arranges elements in order	5 For D

<u>Complete the Sentence Questions:</u>

55	_____ algorithms arrange elements in a specific order.	**Sorting**
56	_____ algorithms are used to find a specific element.	**Selection**
57	Bubble sort repeatedly compares and _____ adjacent elements.	**Swaps**
58	Selection sort places the smallest remaining element in its _____ position.	**Correct**
59	Insertion sort inserts each element into its _____ position in the sorted part of the list.	**Correct**
60	Bubble sort gets its name from the way smaller elements _____ to the top.	**Bubble**

Dear learner: Give The Scientific Term

61	The data structure that operates on the principle of Last In First Out is called a _____.	**Stack**
62	In a stack, the operation to add an element is known as _____.	**Push**
63	The operation used to remove the top element from a stack is called _____.	**Pop**
64	The principle that describes the order of removal in a stack is _____.	**LIFO**
65	The pointer that indicates the current top element in a stack is known as _____.	**TOP**
66	The data structure where the first element added is the first element removed is called a _____.	**Queue**
67	The process of adding an element to the end of a queue is known as _____.	**Enqueue**
68	Removing an element from the front of a queue is called _____.	**Dequeue**
69	The pointer that tracks the front of the queue is called _____.	**FRONT**

70	In queue operations, the pointer that indicates the last element is called _____.	**REAR**
71	A hierarchical data structure composed of nodes connected by edges is known as a _____.	**Tree**
72	The topmost node in a tree is referred to as the _____.	**Root**
73	A node in a tree that has no children is called a _____.	**Leaf**
74	The number of edges on the longest path from a node to a leaf is known as the node's _____.	**Height**
75	The number of edges from the root to a specific node is called the node's _____.	**Depth**
76	A tree data structure where each node has at most two children is called a _____ tree.	**Binary**
77	A specialized tree that maintains balance, ensuring that the height difference between subtrees is minimal, is known as an _____ Tree.	**AVL**
78	In the context of trees, a _____ is a collection of disjoint trees.	**Forest**
79	The connection between two nodes in a tree is known as an _____.	**Edge**
80	The process of viewing the top element of a stack without removing it is called _____.	**Peek**

Dear learner: Choose the correct answer.

81	**What is the top node in a binary tree called?** a) Leaf b) Root c) Branch d) Edge	B
82	**How many children can each node have in a binary tree?** a) 1 b) 2	B

	c) 3 d) Unlimited	
83	**What is a graph made up of?** a) Nodes and edges b) Roots and leaves c) Questions and answers d) Numbers and letters	A
84	**What does binary search do to the list each time it makes a guess?** a) Doubles the list b) Cuts the list in half c) Reverses the list d) Sorts the list	B
85	**Which of the following is NOT a use of graphs?** a) Road maps b) Social networks c) Sorting numbers d) Computer networks	C
86	**What is the middle number in binary search called?** a) Root b) Mid c) Leaf d) Edge	B
87	**In a binary tree, what does the left branch represent?** a) YES b) NO c) Maybe d) Stop	A
88	**What is the term for nodes in a graph that have no connections?**	C

	a) Roots b) Leaves c) Isolated nodes d) Branches	
89	**Which of the following is required for binary search to work?** a) Unsorted list b) Sorted list c) Random list d) Empty list	B
90	**What is the term for the connections between nodes in a graph?** a) Roots b) Edges c) Leaves d) Branches	B
91	**What is the term for the final nodes in a binary tree that have no children?** a) Roots b) Leaves c) Branches d) Edges	B
92	**What is the first step in binary search?** a) Guess the first number b) Guess the last number c) Guess the middle number d) Sort the list	C
93	**Which of the following is an example of a graph?** a) Family tree b) Road map c) Binary tree d) Guessing game	B

94	**What is the term for the starting point in a binary tree?** a) Leaf b) Root c) Edge d) Node	B
95	**What is the term for the process of dividing the list in binary search?** a) Splitting b) Cutting c) Halving d) Sorting	C

Essay questions- clear and readable handwriting

96	**Question**	**How many children can each node have in a binary tree?**
	Answer	Two
97	**Question**	**What are the connections between nodes in a graph called?**
	Answer	Edges
98	**Question**	**What is the first step in binary search?**
	Answer	Guess the middle number
99	**Question**	**What is the term for nodes in a binary tree that have no children?**
	Answer	Leaves
100	**Question**	**What is required for binary search to work?**
	Answer	A sorted list