VIDYA JYOTHI INSTITUTE OF TECHNOLOGY

(An Autonomous Institution)

(Approved by AICTE, New Delhi & Affiliated to JNTUH, Hyderabad)

B.Tech(CSE) II Year / I Semester (R22)

Lecture Notes

Name of the Faculty	**KISHORE K**
Department	**CSE(Data Science)**
Year & Semester	**B.Tech-II & I Sem**
Subject Name	**DATA STRUCTURES**

DEPARTMENT OF CSE (Data Science)

VIDYA JYOTHI INSTITUTE OF TECHNOLOGY

(Approved by AICTE, New Delhi & Affiliated to JNTUH, Hyderabad)

An Autonomous Institution

AZIZ NAGAR, C B POST, HYDERABAD-500075

2025-2026

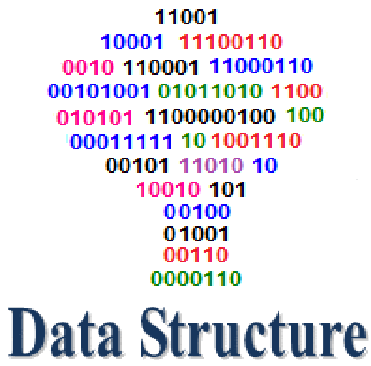
Data Structures **PART-1**

Introduction, Types of data structures, Static and Dynamic representation of data structure and comparison.

Introduction to Data Structures

A **Data Structure** is a way of organizing, storing, and managing data in a computer's memory so that it can be accessed and modified **efficiently**.

The choice of a data structure is crucial because it affects the performance of an algorithm in terms of both time (how fast it runs) and space (how much memory it uses).



The **two main metrics** for evaluating a data structure's performance are:

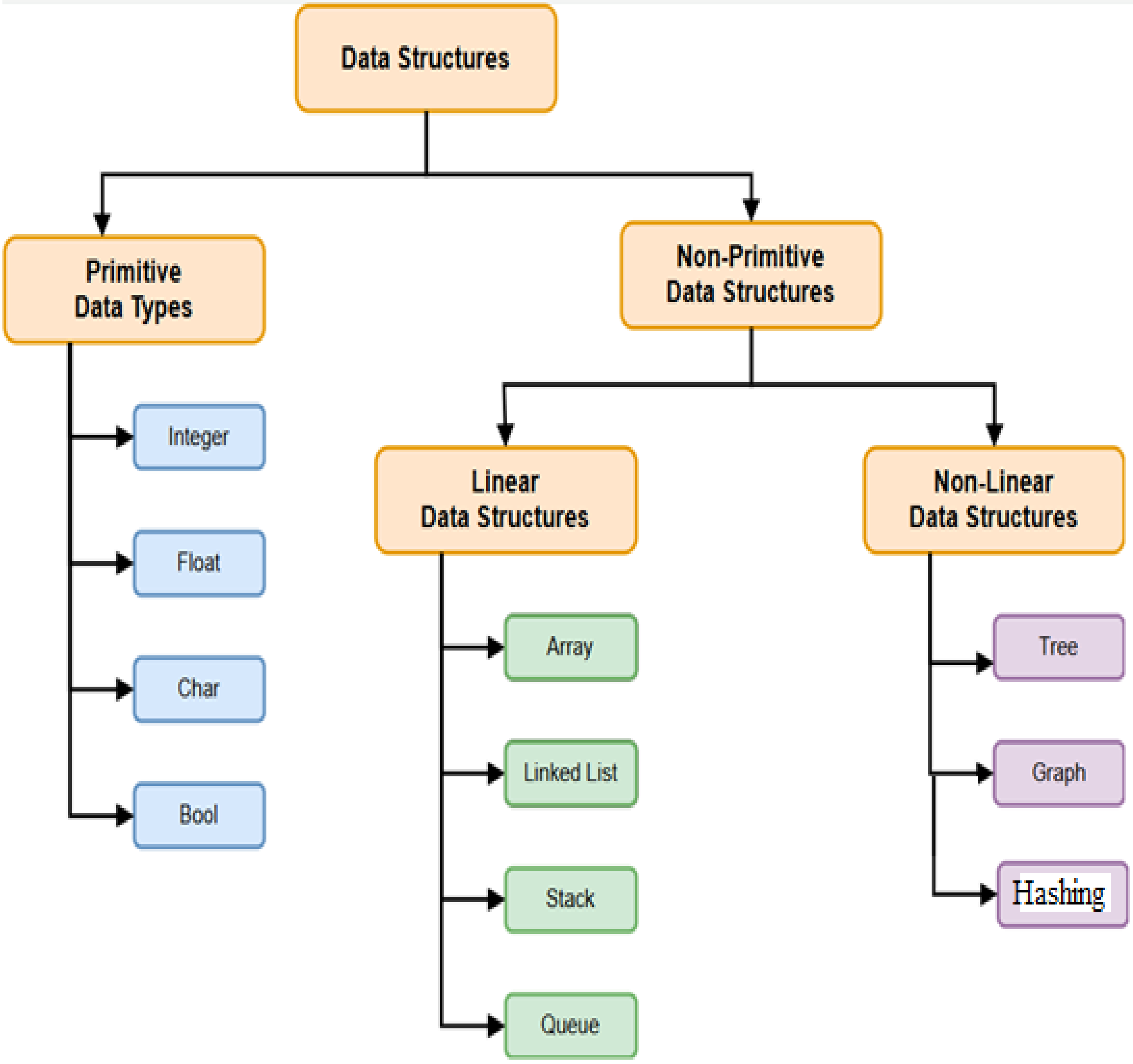
- **Time Complexity:** Measures how the execution time of an algorithm grows with the size of the input data. A good data structure minimizes the time required for operations like searching, insertion, and deletion.
- **Space Complexity:** Measures how the memory usage of an algorithm grows with the size of the input data. A good data structure minimizes the amount of memory needed to store the data.

The **main goal** of using a data structure is to enable efficient operations such as:

- Arranging the data in a specified order (ascending or descending)
- Searching for a specific data item.
- Inserting a new data item.
- Deleting a data item.
- Traversing (processing all items in a specific order).

Types of Data Structures

Data structures are primarily categorized based on their organization.



Primitive Data Structures

These are the most basic building blocks provided by a programming language. They are single-valued, indivisible units of data.

- **Integer (int):** Used to store integer numbers.
- **Float (float, double):** Used to store numbers with a fractional component.
- **Character (char):** Used to store a single character.
- **Boolean (boolean):** Used to store logical values (true or false).

Non-Primitive Data Structures

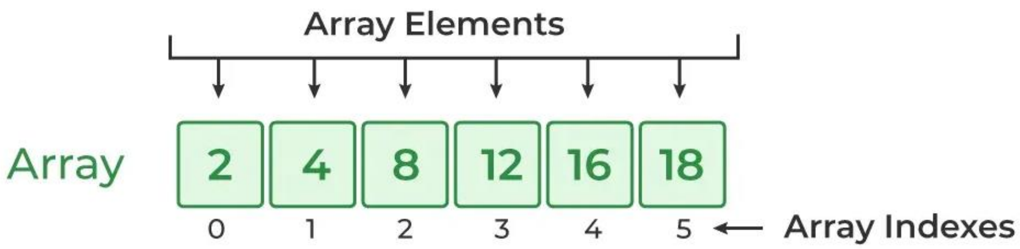
These are more complex data structures created by combining primitive data types. They are designed to hold collections of data. Non-primitive data structures are further sub-divided into two categories.

1. Linear
2. Non-Linear

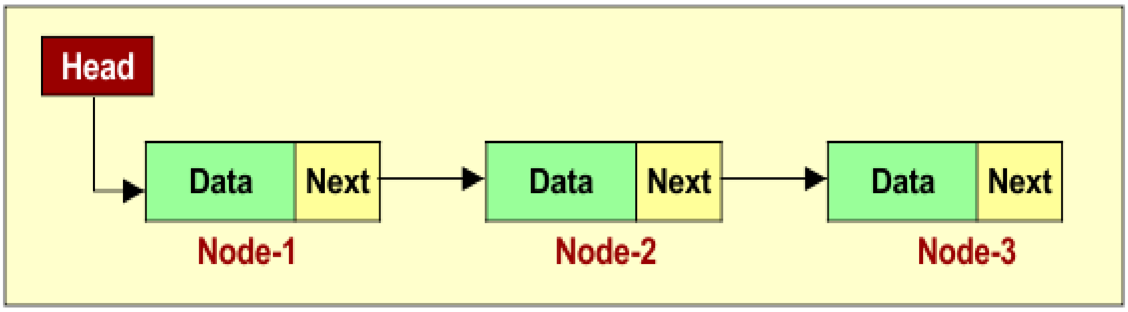
1. Linear Data Structures

Elements are arranged in a sequential, single level. Traversal of the elements is possible in a single run.

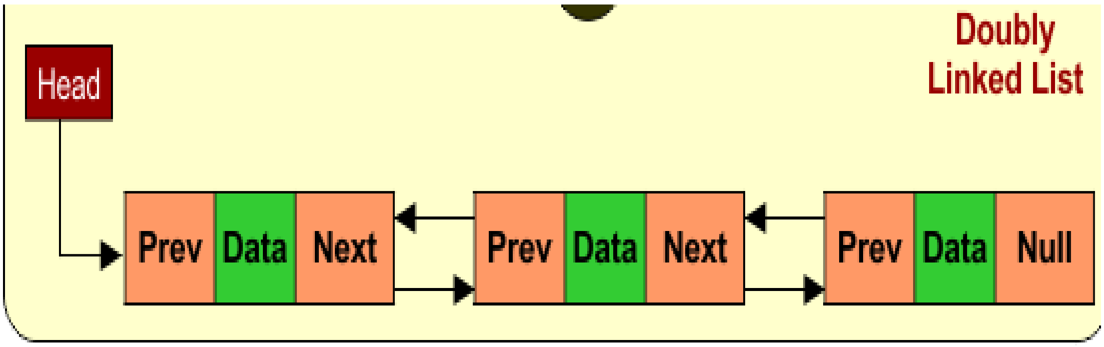
- **Arrays:**
  - **Description:** A collection of items of the **same data type** stored in contiguous memory locations.

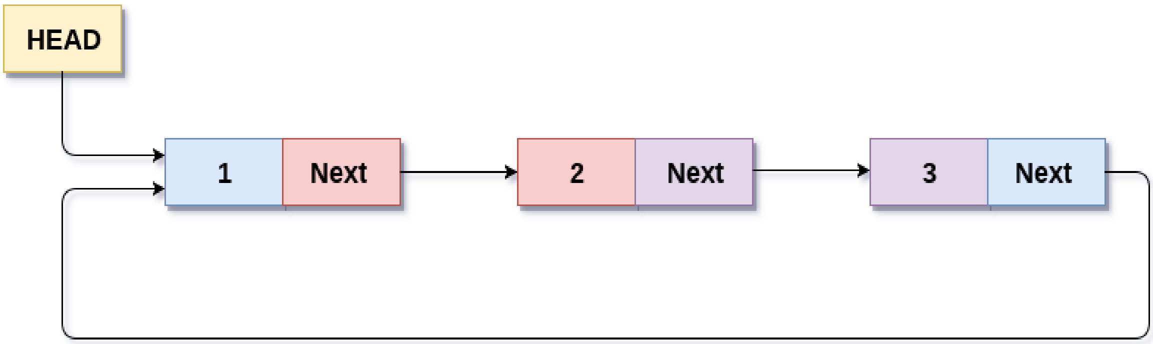


  - **Characteristics:**
    - Fixed size (in C/C++).
    - Elements are accessed directly using an index (e.g., array[i]).
    - Efficient for searching (O(1) access time) and traversal.
    - Inefficient for insertions and deletions in the middle, as it requires shifting elements (O(n)).
- **Linked Lists:**
  - **Description:** A sequence of nodes, where each node contains data and a pointer/reference to the next node in the sequence. The nodes are not necessarily stored in contiguous memory.
  - **Types:**
    - **Singly Linked List:** Each node points to the next.
    - **Doubly Linked List:** Each node has pointers to both the next and previous nodes.
    - **Circular Linked List:** The last node points back to the first.

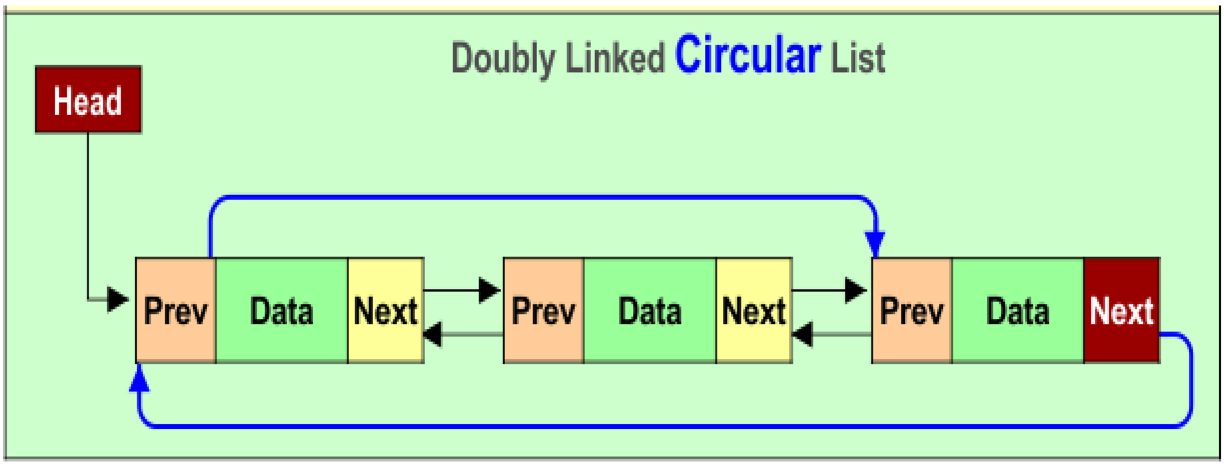


Synatax - Singly Linked List



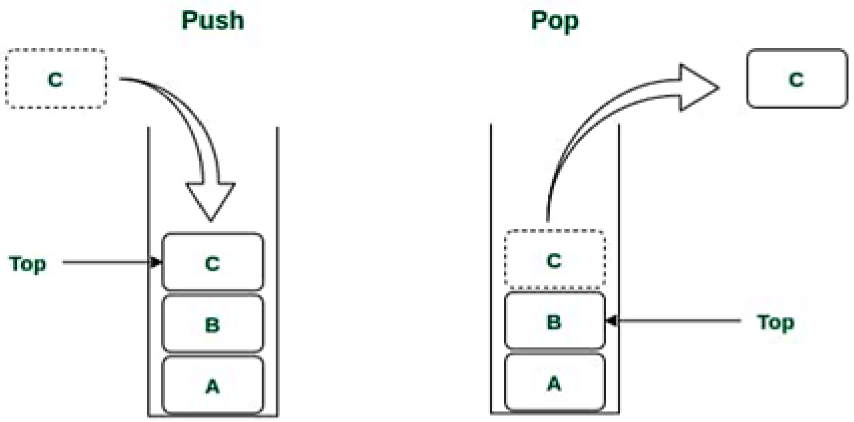


Circular Singly Linked List



- **Characteristics:**
    - Dynamic size, can grow or shrink at runtime.
    - Efficient for insertions and deletions (O(1) once the node is found).
    - Inefficient for random access (O(n) for a specific element).

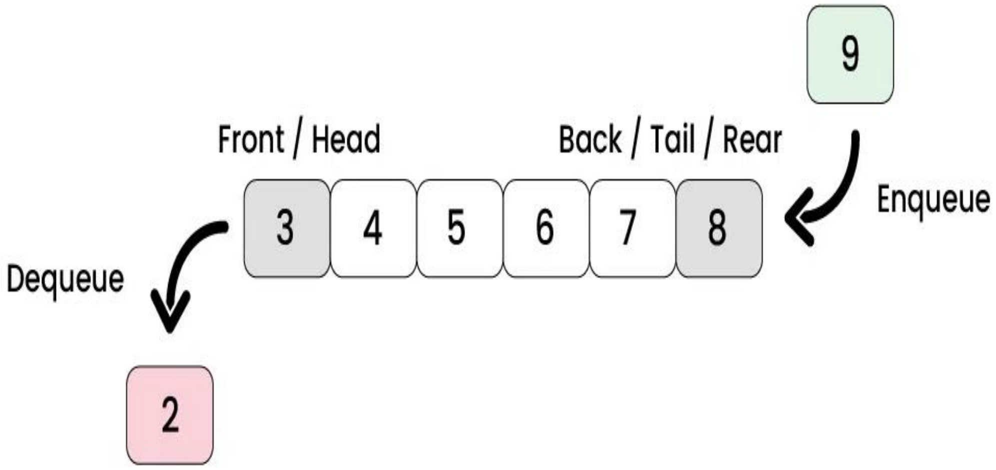
- **Stacks:**
    - **Description:** A LIFO (Last-In, First-Out) data structure.



Stack Data Structure

- **Key Operations:**
    - **Push:** Adds an element to the top of the stack.
    - **Pop:** Removes and returns the element from the top.
- **Applications:** Function call stacks, expression evaluation, undo/redo features.

- **Queues:**
    - **Description:** A FIFO (First-In, First-Out) data structure.

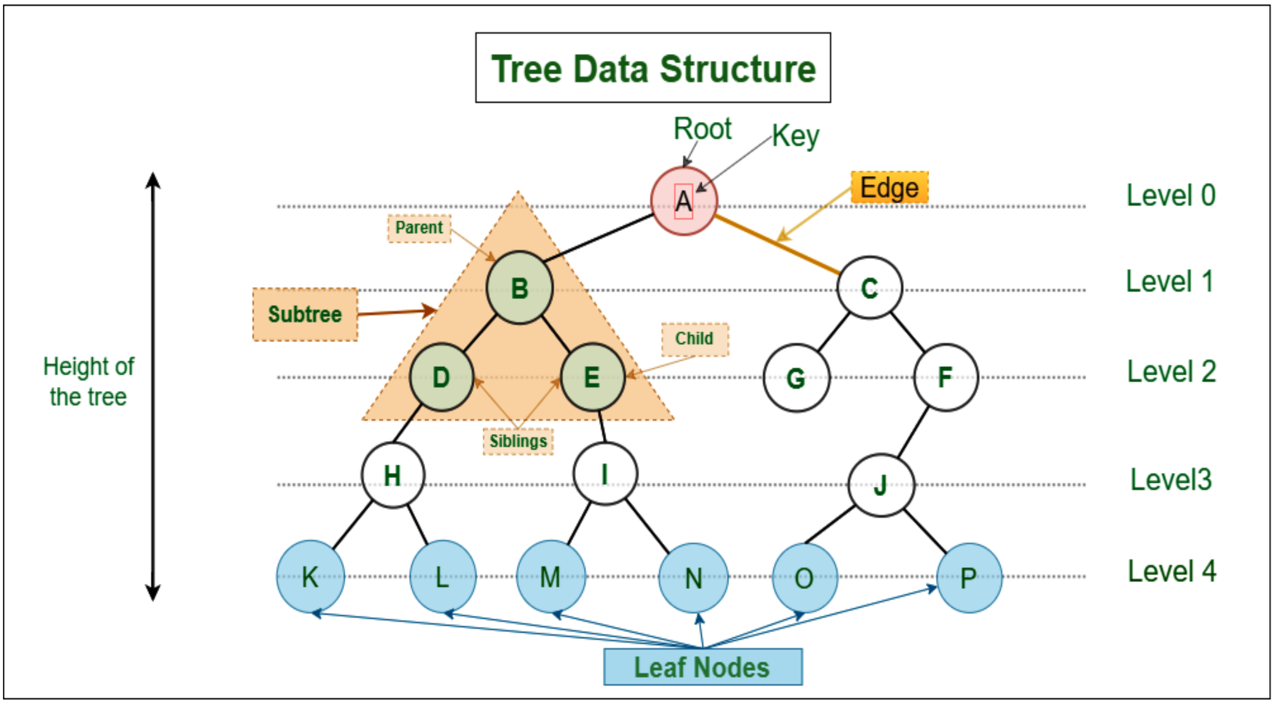


- **Key Operations:**
  - **Enqueue:** Adds an element to the rear of the queue.
  - **Dequeue:** Removes and returns the element from the front.
- **Applications:** Task scheduling, resource management, bread-first search (BFS) in graphs.

2. Non-Linear Data Structures

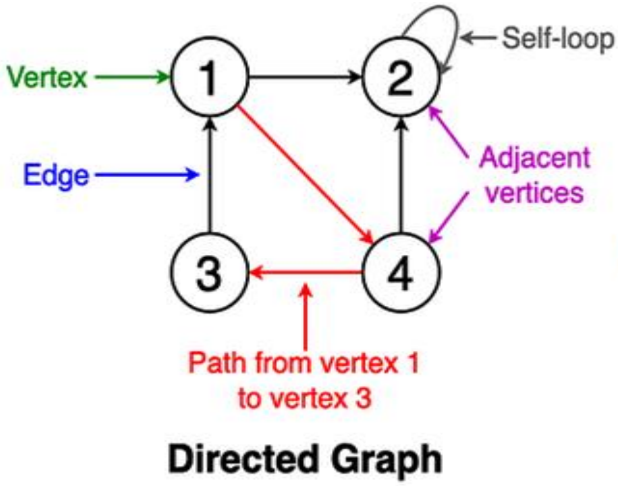
Elements are not arranged in a sequential fashion. An element can be connected to multiple other elements, representing more complex relationships.

- **Trees:**
  - **Description:** A hierarchical data structure with a root node and sub-trees connected by edges.

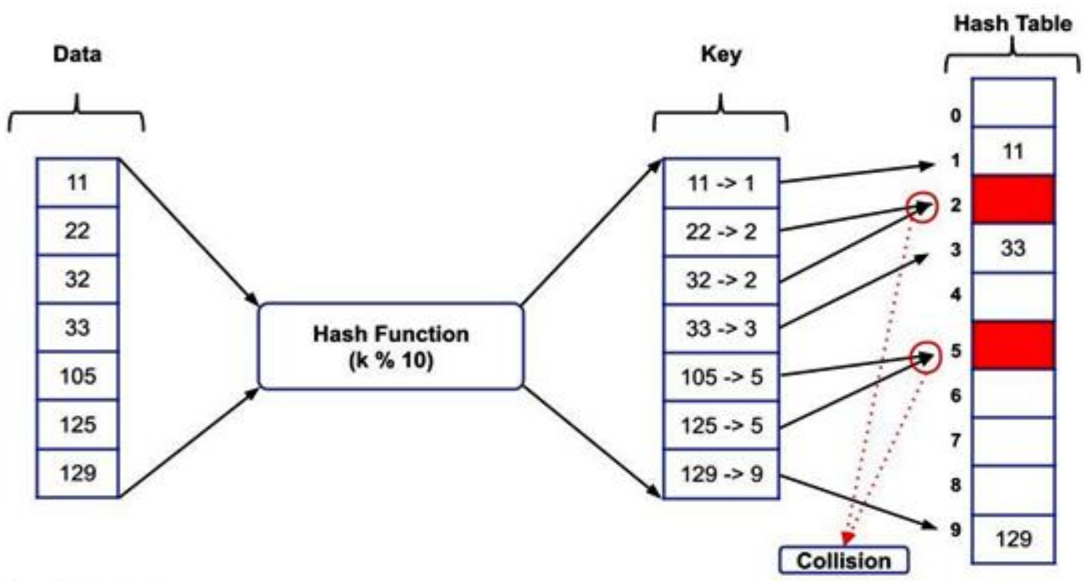


  - **Types:**
    - **Binary Tree:** Each node has at most two children.
    - **Binary Search Tree (BST):** A special type of binary tree where the left child's value is less than the parent's, and the right child's value is greater. This allows for efficient searching (O(log n)).
    - **AVL Tree, Red-Black Tree:** Self-balancing trees that guarantee logarithmic time for insertions and deletions.
  - **Applications:** Database indexing, file systems, syntax parsing.

- **Graphs:**
  - **Description:** A collection of vertices (nodes) and edges (connections). Graphs can represent relationships between any two objects.



  - **Types:**
    - **Directed vs. Undirected:** Edges can have a direction or not.
    - **Weighted vs. Unweighted:** Edges can have a weight or cost associated with them.
  - **Applications:** Social networks, GPS navigation systems, network routing.
- **Hash Tables:**
  - **Description:** A data structure that stores data in key-value pairs. It uses a **hash function** to compute an index in an array (or bucket) from a given key.



- **Characteristics:**
  - Provides very fast average-case time complexity for searching, insertion, and deletion (O(1)).
  - The worst-case complexity can be O(n) due to collisions (multiple keys mapping to the same index).
- **Applications:** Symbol tables in compilers, database indexing, caches.

Static and Dynamic representation of Data Structure

Data Structures are ways of organizing and storing data in a computer's memory so that it can be used efficiently. One of the fundamental distinctions in data structures is whether their representation is **Static** or **Dynamic**.

Static Representation of Data Structures

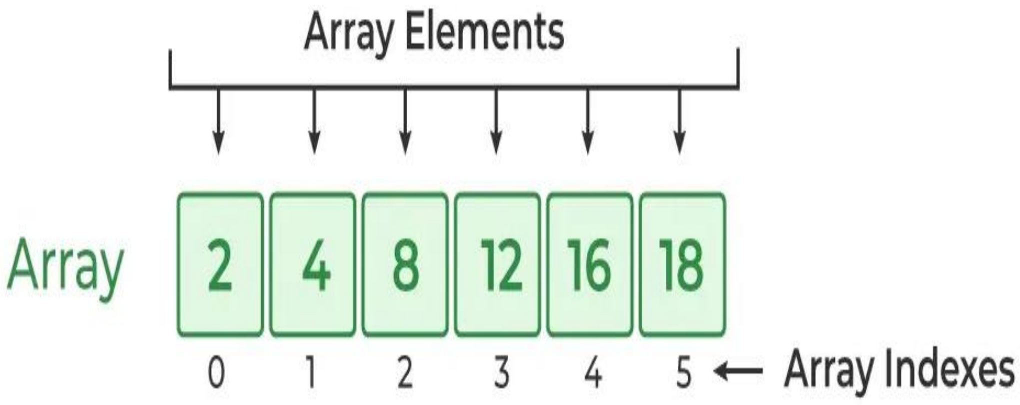
A data structure with a static representation has a **fixed size** that is determined at **compile time**. Once the memory is allocated for the structure, its size cannot be changed during the program's execution.

Key Characteristics:

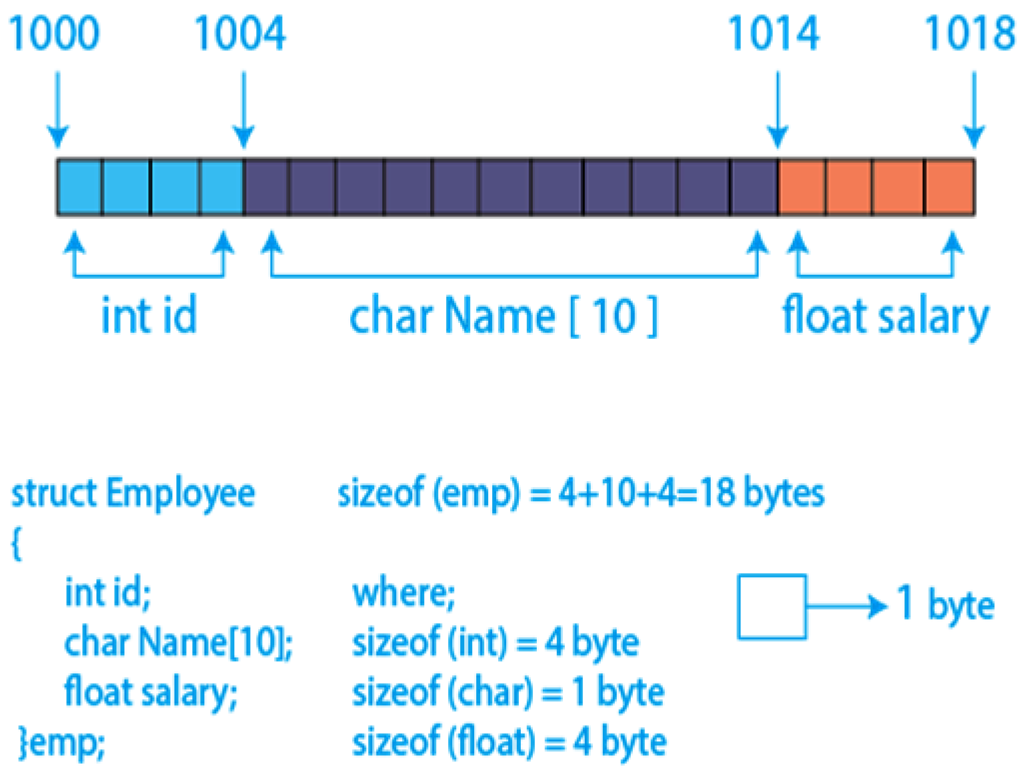
- **Fixed Size:** The maximum number of elements the structure can hold must be specified in advance.
- **Compile-Time Allocation:** Memory is typically allocated on the **stack**, a region of memory used for local variables and function calls.
- **Memory Management:** Memory is automatically allocated and deallocated by the compiler. There is no manual management by the programmer.
- **Contiguous Memory:** Elements are stored in a single, contiguous block of memory. This allows for fast, direct access to any element using an index.
- **Simpler Implementation:** Static data structures are generally easier to implement and use because of their fixed nature.

Examples:

- **Arrays:** A classic example. When you declare int arr[6]; Its size cannot be changed later.



- **C-style structs:** While a struct itself can be a fixed size, it is a static representation when its members are of fixed size and are allocated on the stack.



Dynamic Representation of Data Structures

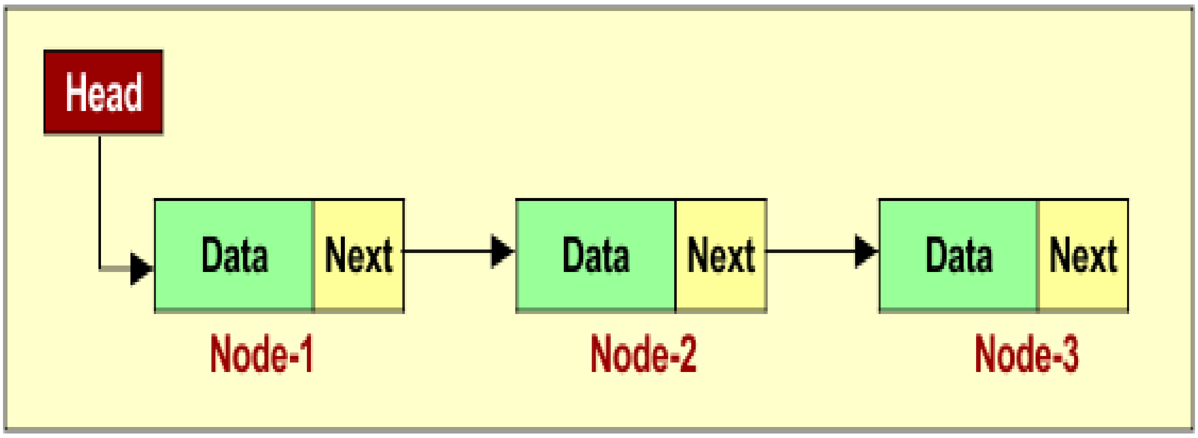
A data structure with a dynamic representation has a **variable size** that can be changed during **runtime**. It can **grow** or **shrink** as needed to accommodate the data.

Key Characteristics:

- **Variable Size:** The size of the structure is not fixed and can be altered while the program is running.
- **Runtime Allocation:** Memory is allocated on the **heap**, a separate region of memory managed by the operating system. This allows for flexible allocation and deallocation.
- **Manual Memory Management:** The Programmer is responsible for allocating and deallocating memory (e.g., using malloc(), calloc() and free() in C). Improper management can lead to **memory leaks**.
- **Non-Contiguous Memory (often):** Elements are not always stored in a single contiguous block. They are often linked together using pointers, which can be scattered across different memory locations.
- **Complex Implementation:** Dynamic Data Structures are more complex to implement due to the need for managing pointers and memory allocation.

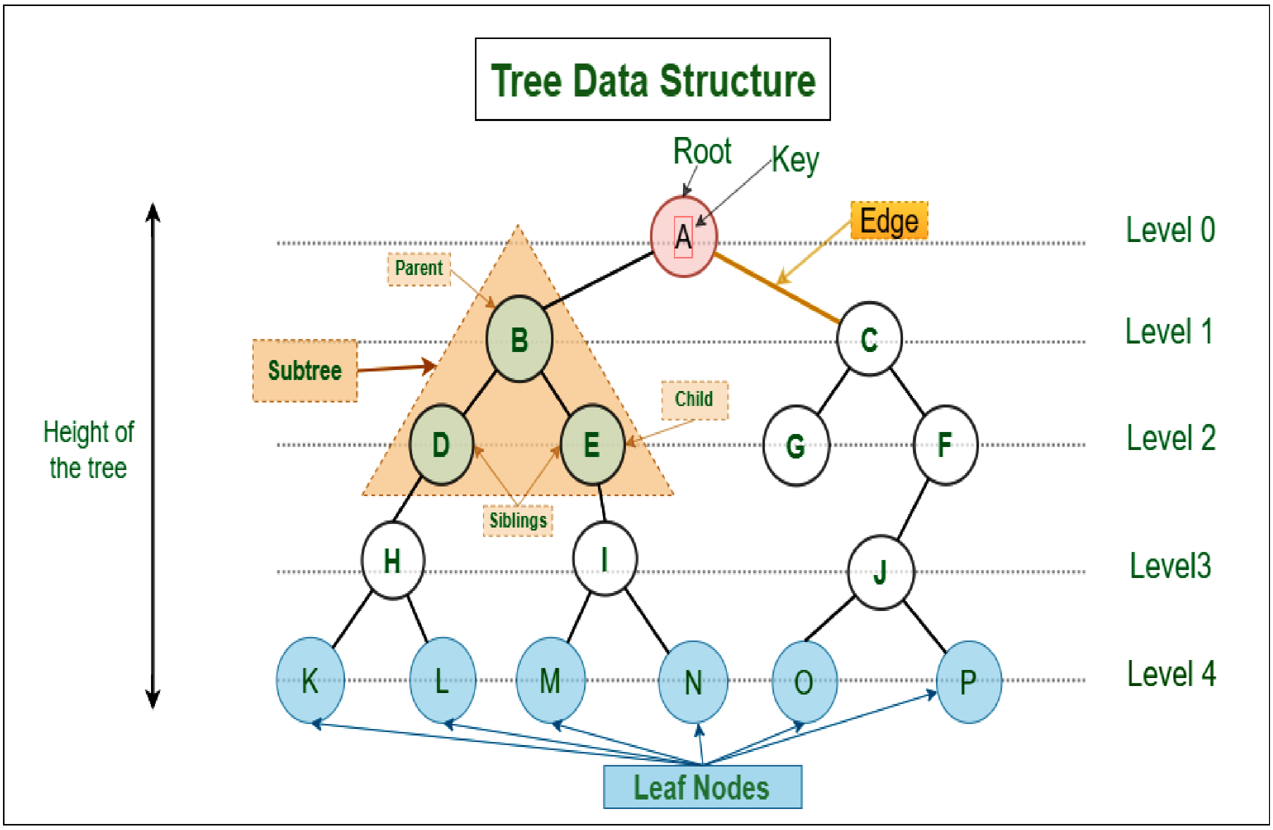
Examples:

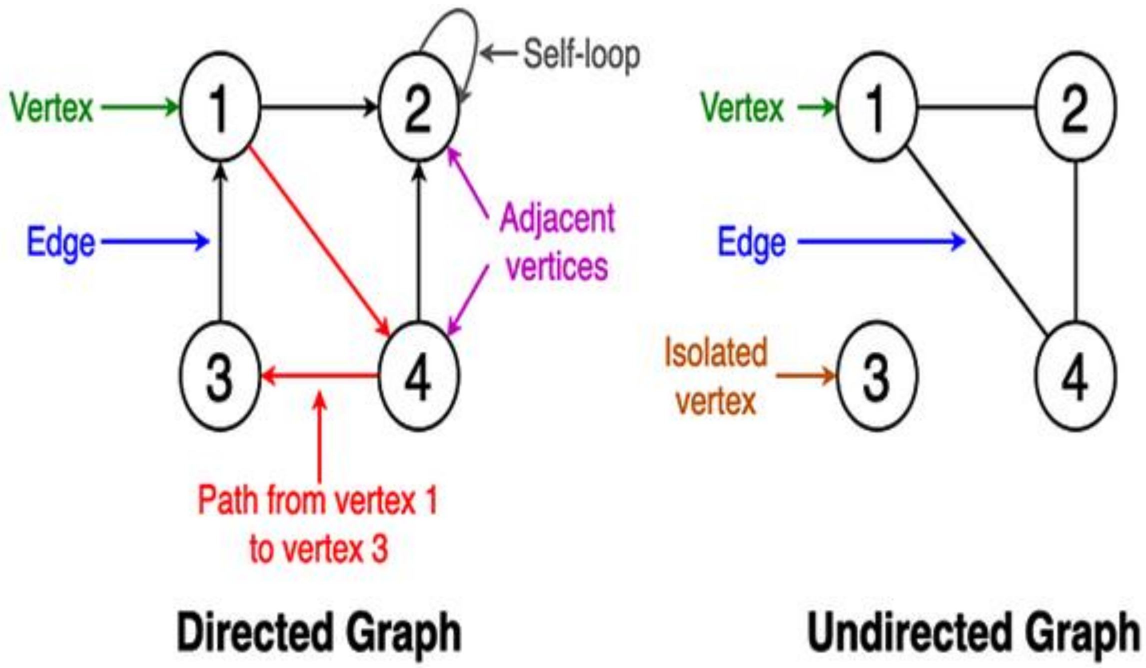
- **Linked Lists:** Each element (node) stores the data and a pointer to the next element. Nodes can be added or removed dynamically.



Synatax - Singly Linked List

- **Trees and Graphs:** These are complex, non-linear data structures where nodes are connected by pointers, allowing for flexible growth and change.





Comparison Table

Feature	Static Data Structure	Dynamic Data Structure
Size	Fixed; determined at compile time.	Variable; can change during runtime.
Memory Allocation	Compile-time (stack).	Run-time (heap) (using using malloc, calloc).
Memory Management	Automatic; handled by the compiler.	Manual; handled by the programmer.
Memory Utilization	Inefficient if the allocated size is much larger than the actual data.	Efficient; memory is allocated as needed, reducing waste.
Access Speed	Generally faster due to contiguous memory and direct indexing (O (1)).	Can be slower due to non-contiguous memory, requiring traversal (e.g., linked lists) or additional overhead for pointer management.
Ease of Use	Simpler to implement and use.	More complex to implement and manage.
Flexibility	Less flexible; size cannot be changed.	Highly flexible; can grow or shrink to fit data requirements.
Common Examples	Arrays, Struct	Linked Lists, Trees, Graphs, Hash Tables