**Number System and Data Representation**

Number systems and data representation are fundamental concepts in computer science and data structures. Computers store and process data in various forms using different number systems. Understanding these concepts is crucial for efficient programming, memory management, and algorithm design.

**1. Number System**

A number system defines a way to represent numbers using a set of symbols and rules. In computing, different number systems are used to represent and manipulate data.

**Types of Number Systems**

1. **Decimal Number System (Base-10)**
   * Uses digits: **0, 1, 2, 3, 4, 5, 6, 7, 8, 9**
   * Each digit's position represents a power of 10.
   * Example: 256=2×102+5×101+6×100256 = 2 \times 10^2 + 5 \times 10^1 + 6 \times 10^0256=2×102+5×101+6×100
2. **Binary Number System (Base-2)**
   * Uses digits: **0, 1**
   * Each digit’s position represents a power of 2.
   * Example: 10112=1×23+0×22+1×21+1×20=11101011\_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 11\_{10}10112​=1×23+0×22+1×21+1×20=1110​
3. **Octal Number System (Base-8)**
   * Uses digits: **0, 1, 2, 3, 4, 5, 6, 7**
   * Each digit’s position represents a power of 8.
   * Example: 578=5×81+7×80=471057\_8 = 5 \times 8^1 + 7 \times 8^0 = 47\_{10}578​=5×81+7×80=4710​
4. **Hexadecimal Number System (Base-16)**
   * Uses digits: **0–9, A–F** (where A=10, B=11, ..., F=15)
   * Each digit’s position represents a power of 16.
   * Example: 1A316=1×162+A(10)×161+3×160=419101A3\_{16} = 1 \times 16^2 + A(10) \times 16^1 + 3 \times 16^0 = 419\_{10}1A316​=1×162+A(10)×161+3×160=41910​

**2. Conversion Between Number Systems**

**(a) Decimal to Other Bases**

1. **Decimal to Binary:** Divide the number by 2, store the remainder, and read in reverse.
   * Example: 131013\_{10}1310​ → Binary?
     + 13 ÷ 2 = 6, remainder **1**
     + 6 ÷ 2 = 3, remainder **0**
     + 3 ÷ 2 = 1, remainder **1**
     + 1 ÷ 2 = 0, remainder **1**
     + Answer: **1101₂**
2. **Decimal to Octal:** Divide by 8 and store the remainders.
3. **Decimal to Hexadecimal:** Divide by 16 and store the remainders.

**(b) Binary to Other Bases**

1. **Binary to Decimal:** Multiply each bit by powers of 2 and sum.
2. **Binary to Octal:** Group bits in sets of 3 (right to left) and convert.
3. **Binary to Hexadecimal:** Group bits in sets of 4 (right to left) and convert.

Example:

* Binary 110110121101101\_211011012​ → Octal?
  + Group: **110** **110** **1** (Add 0: 001101101)
  + Convert: 0012=18001\_2 = 1\_80012​=18​, 1012=58101\_2 = 5\_81012​=58​, 1012=58101\_2 = 5\_81012​=58​
  + Answer: **155₈**

**3. Data Representation in Computers**

Computers represent and store different types of data, including numbers, characters, and images, using **bits (binary digits: 0 and 1).**

**(a) Integer Representation**

1. **Unsigned Integer Representation**
   * Stores only positive numbers.
   * Example: An 8-bit representation can store numbers from **0 to 255**.
2. **Signed Integer Representation**
   * Uses **sign-magnitude, one’s complement, or two’s complement** to store negative numbers.
   * **Two’s complement** is most commonly used because it simplifies arithmetic operations.

**Example (8-bit representation of -5 in two’s complement)**

* + Convert 5 to binary: 00000101200000101\_2000001012​
  + Take one’s complement: 11111010211111010\_2111110102​
  + Add 1: 11111011211111011\_2111110112​ → This represents **-5**.

**(b) Floating-Point Representation (For Real Numbers)**

* Follows **IEEE 754 standard**.
* Stores a number using:
  + **Sign bit** (0 for positive, 1 for negative)
  + **Exponent** (adjusted for range)
  + **Mantissa** (fractional part)

Example:  
The decimal number **10.75** in binary:

* 10.7510=1010.11210.75\_{10} = 1010.11\_210.7510​=1010.112​

Using IEEE 754:

* **32-bit representation**: 01000001001110000000000000000000201000001001110000000000000000000\_2010000010011100000000000000000002​

**(c) Character Representation**

1. **ASCII (American Standard Code for Information Interchange)**
   * Uses **7 or 8 bits** per character.
   * Example: ‘A’ = **65** (Binary: 1000001)
2. **Unicode (UTF-8, UTF-16, UTF-32)**
   * Supports multiple languages and symbols.
   * Example: ‘A’ in UTF-8 = **01000001**

**(d) Image, Audio, and Video Representation**

1. **Image Representation**
   * **Bitmap (BMP, PNG, JPEG)**: Uses pixels and color codes.
   * Example: A **24-bit color** image uses **8 bits for Red, Green, and Blue (RGB)**.
2. **Audio Representation**
   * Uses **sampling rate (Hz)** and **bit depth**.
   * Example: A **44.1 kHz, 16-bit stereo** audio file means **44,100 samples per second, 16-bit depth**.
3. **Video Representation**
   * Combination of **frames per second (fps)** and **color encoding**.
   * Example: A **1080p video at 60 fps** contains **1920x1080 pixels per frame, updated 60 times per second**.

**4. Error Detection and Correction**

Since data is stored in binary, errors can occur during transmission. To detect and correct errors, various techniques are used:

**(a) Parity Bit**

* A **single bit** added to ensure even or odd number of 1s.
* Example: **Even parity for 1011 → 10110 (Added 0 for even 1s count).**

**(b) Hamming Code**

* Used for **error detection and correction**.
* Adds multiple **parity bits** to detect and fix errors.

**(c) Checksum and CRC (Cyclic Redundancy Check)**

* Used in **network communication** to verify data integrity.

**Conclusion**

Understanding number systems and data representation is crucial for computer programming, networking, and system design. These concepts form the foundation for operations like data storage, processing, and transmission in modern computers.

4o

**Fundamentals of Algorithms**

Algorithms are the foundation of computer science and data structures. They define a step-by-step approach to solving problems efficiently. Understanding algorithms is essential for optimizing performance, reducing computational complexity, and improving problem-solving skills.

**1. What is an Algorithm?**

An **algorithm** is a finite set of well-defined instructions to solve a specific problem. It takes some input, processes it, and produces an output.

**Characteristics of a Good Algorithm**

A well-designed algorithm should have the following properties:

1. **Input:** It should take zero or more inputs.
2. **Output:** It should produce at least one output.
3. **Definiteness:** Every step should be clearly defined.
4. **Finiteness:** It should have a finite number of steps.
5. **Effectiveness:** Each step should be simple and achievable.

**Example of an Algorithm**

**Algorithm to Find the Sum of Two Numbers**

1. Start
2. Take two numbers as input
3. Add the two numbers
4. Display the result
5. Stop

This is a simple algorithm that performs addition.

**2. Algorithm Representation**

Algorithms can be represented in different ways:

**(a) Natural Language**

* Written in plain English.
* Example:  
  **Step 1:** Start  
  **Step 2:** Take input A and B  
  **Step 3:** Compute sum = A + B  
  **Step 4:** Print sum  
  **Step 5:** Stop

**(b) Flowchart**

* A **graphical representation** of an algorithm using symbols like rectangles, diamonds, and arrows.

**(c) Pseudocode**

* A **high-level description** of an algorithm that resembles programming syntax.
* Example:

plaintext

CopyEdit

Algorithm Sum(A, B)

Begin

Sum ← A + B

Print Sum

End

**3. Types of Algorithms**

Different algorithms are used for different problem-solving techniques:

**(a) Brute Force Algorithm**

* Tries all possible solutions.
* Example: **Linear Search** (searches each element one by one).

**(b) Divide and Conquer Algorithm**

* Splits the problem into smaller parts, solves them independently, and combines results.
* Example: **Merge Sort, Quick Sort**

**(c) Greedy Algorithm**

* Makes the best possible choice at each step.
* Example: **Dijkstra’s Algorithm (shortest path finding)**

**(d) Dynamic Programming**

* Solves problems by breaking them into overlapping subproblems and storing results.
* Example: **Fibonacci Series using Memoization**

**(e) Backtracking**

* Tries all solutions and backtracks when an invalid path is found.
* Example: **N-Queens Problem, Sudoku Solver**

**(f) Recursion-Based Algorithm**

* Calls itself to solve subproblems.
* Example: **Factorial Calculation (n! = n × (n-1)!)**

**4. Algorithm Performance Analysis**

To evaluate an algorithm, we analyze:

**(a) Time Complexity**

* The amount of time an algorithm takes to execute.
* Expressed using **Big-O Notation**.
* Example:
  + Linear Search: **O(n)**
  + Binary Search: **O(log n)**
  + Bubble Sort: **O(n²)**
  + Merge Sort: **O(n log n)**

**(b) Space Complexity**

* The amount of memory an algorithm uses.
* Includes:
  1. **Fixed Part**: Instructions and constants.
  2. **Variable Part**: Memory for variables, recursion stack, etc.

**5. Important Algorithmic Problems**

Some common algorithmic problems include:

* **Sorting Algorithms** (Bubble Sort, Quick Sort, Merge Sort)
* **Searching Algorithms** (Linear Search, Binary Search)
* **Graph Algorithms** (Dijkstra’s Algorithm, BFS, DFS)
* **Tree Algorithms** (Tree Traversals: Inorder, Preorder, Postorder)

**6. Algorithm Design Techniques**

To design efficient algorithms, different techniques are used:

1. **Incremental Approach** – Solve the problem step by step.
2. **Decomposition Approach** – Break into smaller problems.
3. **Recursive Approach** – Solve using self-calling functions.

**Conclusion**

Algorithms are essential for solving computational problems efficiently. Understanding different algorithm types, performance analysis, and design techniques helps in writing optimized code and improving programming skills.

**Data Types: Primitive, Non-Primitive, and Abstract Data Types (ADT)**

Data types define the kind of data that variables can hold and the operations that can be performed on them. In data structures and programming, data types are classified into **Primitive**, **Non-Primitive**, and **Abstract Data Types (ADT)**.

**1. Primitive Data Types**

Primitive data types are the **basic building blocks** of data representation in programming languages. They store single values and are pre-defined by the language.

**Types of Primitive Data Types**

1. **Integer (int)**
   * Stores whole numbers (positive, negative, or zero).
   * Example: int x = 10;
   * Memory Usage: 2 or 4 bytes (depends on system).
2. **Floating-Point (float, double)**
   * Stores real numbers (numbers with decimal points).
   * Example: float pi = 3.14;
   * Memory Usage:
     + float (4 bytes, single precision)
     + double (8 bytes, double precision)
3. **Character (char)**
   * Stores a single character.
   * Example: char grade = 'A';
   * Memory Usage: 1 byte (ASCII) or 2 bytes (Unicode).
4. **Boolean (bool)**
   * Stores only two values: true or false.
   * Example: bool isAvailable = true;
   * Memory Usage: 1 bit or 1 byte (depends on the language).
5. **Void**
   * Represents "no value" or "empty."
   * Used for functions that do not return a value.
   * Example: void display();

**2. Non-Primitive Data Types**

Non-primitive data types are more complex and are derived from primitive data types. They can hold multiple values and allow structured data storage.

**Types of Non-Primitive Data Types**

1. **Arrays**
   * A collection of elements of the same data type.
   * Elements are stored in **contiguous memory locations**.
   * Example:

c

CopyEdit

int numbers[5] = {1, 2, 3, 4, 5};

* + Memory Usage: Depends on the number of elements.

1. **Strings**
   * A sequence of characters.
   * In C, it is represented as an array of characters ending with a **null (\0) character**.
   * Example:

c

CopyEdit

char name[] = "Hello";

* + Memory Usage: Number of characters + 1 (for \0).

1. **Structures (struct)**
   * A collection of variables of different types.
   * Example:

c

CopyEdit

struct Student {

char name[20];

int age;

float marks;

};

* + Memory Usage: Sum of all member variables.

1. **Unions**
   * Similar to a structure, but all members share the **same memory location**.
   * Example:

c

CopyEdit

union Data {

int i;

float f;

char str[20];

};

* + Memory Usage: Size of the largest member.

1. **Enumerations (enum)**
   * A user-defined data type that assigns names to integer values.
   * Example:

c

CopyEdit

enum Color {RED, GREEN, BLUE};

* + Memory Usage: 4 bytes (default integer size).

1. **Pointers**
   * Stores the memory address of another variable.
   * Example:

c

CopyEdit

int x = 10;

int \*ptr = &x;

* + Memory Usage: 4 bytes (for 32-bit systems) or 8 bytes (for 64-bit systems).

**3. Abstract Data Type (ADT)**

An **Abstract Data Type (ADT)** is a **mathematical model** for data structures that defines the **behavior** of the data without specifying the implementation.

**Characteristics of ADT**

1. **Encapsulation**: Hides implementation details.
2. **Abstraction**: Defines only essential operations.
3. **Reusability**: Can be implemented in different ways.

**Examples of ADT**

1. **List ADT**
   * A collection of ordered elements.
   * Operations: Insertion, Deletion, Searching, Traversing.
2. **Stack ADT**
   * Follows **LIFO (Last In, First Out)** principle.
   * Operations:
     + push() – Adds an element.
     + pop() – Removes an element.
     + peek() – Returns top element.
3. **Queue ADT**
   * Follows **FIFO (First In, First Out)** principle.
   * Operations:
     + enqueue() – Adds an element.
     + dequeue() – Removes an element.
4. **Deque ADT (Double-Ended Queue)**
   * Elements can be added or removed from both ends.
5. **Priority Queue ADT**
   * Each element has a priority.
   * The element with the highest priority is removed first.
6. **Graph ADT**
   * Consists of **nodes (vertices)** and **edges (connections)**.
   * Operations:
     + addEdge()
     + removeEdge()
     + traverseGraph()

**Comparison of Data Types**

| **Category** | **Examples** | **Usage** |
| --- | --- | --- |
| **Primitive** | int, float, char, bool | Stores single values |
| **Non-Primitive** | array, struct, pointer | Stores multiple values, complex structures |
| **Abstract Data Type (ADT)** | Stack, Queue, Graph | Defines behavior without implementation |

**Conclusion**

Understanding data types is essential for efficient memory management and program execution. Primitive types handle simple values, non-primitive types manage complex data structures, and Abstract Data Types (ADTs) define logical models for data handling.

**Classification of Data Structure: Linear and Non-Linear Data Structures**

Data structures are essential for organizing, managing, and storing data efficiently. They can be broadly classified into **Linear Data Structures** and **Non-Linear Data Structures** based on how data elements are arranged and accessed.

**1. What is a Data Structure?**

A **data structure** is a specialized format for organizing and storing data so that it can be efficiently accessed and modified. The choice of data structure impacts the efficiency of algorithms used for different operations like searching, sorting, and traversing.

**2. Classification of Data Structures**

Data structures are classified into two main types:

1. **Linear Data Structures**
2. **Non-Linear Data Structures**

**I. Linear Data Structures**

Linear data structures store data **sequentially**, meaning each element has a unique **predecessor and successor** (except for the first and last elements). The elements are arranged in a single level, making traversal straightforward.

**Characteristics of Linear Data Structures:**

✔ Data elements are stored in a **sequential** manner.  
✔ Simple to implement and easy to traverse.  
✔ Requires **contiguous** memory allocation (in some cases).  
✔ The time complexity for access depends on the structure (e.g., **O(1)** for arrays, **O(n)** for linked lists).

**Types of Linear Data Structures**

**1. Arrays**

An **array** is a collection of elements of the same data type stored in contiguous memory locations.  
✔ **Fixed size** (Static Memory Allocation).  
✔ **Accessed using an index** (0-based or 1-based depending on the language).  
✔ **Efficient for random access** (O(1) time complexity).  
✔ **Insertion and deletion** are costly (O(n) in the worst case).

**Example:**

c

CopyEdit

int arr[5] = {10, 20, 30, 40, 50}; // Array of 5 integers

**2. Linked List**

A **linked list** is a linear data structure where each element (node) is connected using **pointers**. It does not require contiguous memory allocation.

✔ **Dynamic size** (Can grow or shrink at runtime).  
✔ **Efficient insertions and deletions** (O(1) for head insertions).  
✔ **Slower access time** compared to arrays (O(n) for searching).

**Types of Linked Lists:**

* **Singly Linked List** (Each node has a pointer to the next node).
* **Doubly Linked List** (Each node has pointers to both previous and next nodes).
* **Circular Linked List** (Last node points back to the first node).

**Example of Singly Linked List Node:**

c

CopyEdit

struct Node {

int data;

struct Node\* next;

};

**3. Stack**

A **stack** follows the **LIFO (Last In, First Out)** principle. Elements can only be added or removed from the **top** of the stack.

✔ Operations:

* **Push()** – Adds an element to the top.
* **Pop()** – Removes the top element.
* **Peek()** – Returns the top element without removing it.

✔ **Used in:**

* Function calls (Recursion)
* Undo/Redo operations
* Expression evaluation (Postfix, Prefix)

**Example of Stack Operations:**

c

CopyEdit

push(10);

push(20);

pop(); // Removes 20 (Last element added)

**4. Queue**

A **queue** follows the **FIFO (First In, First Out)** principle. The element inserted first is removed first.

✔ Operations:

* **Enqueue()** – Adds an element at the rear.
* **Dequeue()** – Removes an element from the front.
* **Peek()** – Returns the front element.

✔ **Used in:**

* CPU Scheduling
* Printer Spooling
* Data Buffering

**Example of Queue Operations:**

c

CopyEdit

enqueue(10);

enqueue(20);

dequeue(); // Removes 10 (First element added)

✔ **Variations of Queue:**

* **Circular Queue** – The last position connects back to the first position.
* **Priority Queue** – Elements are dequeued based on priority rather than FIFO.
* **Deque (Double-Ended Queue)** – Insertion and deletion can happen at both ends.

**II. Non-Linear Data Structures**

Non-linear data structures do **not store data sequentially**. Instead, they follow a hierarchical or interconnected pattern, making them suitable for complex relationships between elements.

**Characteristics of Non-Linear Data Structures:**

✔ Elements are arranged in **hierarchical or interconnected** structures.  
✔ **More complex** than linear structures.  
✔ Efficient in handling **multi-dimensional relationships**.  
✔ Suitable for **graph-based and hierarchical problems**.

**Types of Non-Linear Data Structures**

**1. Trees**

A **tree** is a hierarchical data structure consisting of **nodes** connected by edges. It has a **root node** and multiple child nodes.

✔ **Important Terms:**

* **Root**: The first node of the tree.
* **Parent and Child**: Nodes connected directly.
* **Leaf Node**: A node with no children.

✔ **Used in:**

* File systems
* Database indexing
* Artificial Intelligence (Decision Trees)

✔ **Types of Trees:**

* **Binary Tree** – Each node has at most two children.
* **Binary Search Tree (BST)** – Left child contains smaller values, right child contains larger values.
* **Heap** – A complete binary tree used in priority queue implementations.
* **Trie** – Used in dictionary and auto-complete operations.

**Example of Binary Search Tree (BST):**

plaintext

CopyEdit

50

/ \

30 70

/ \ / \

20 40 60 80

**2. Graphs**

A **graph** is a non-linear data structure consisting of **vertices (nodes)** and **edges (connections)**.

✔ **Graph Representation:**

* **Adjacency Matrix** – Uses a 2D array.
* **Adjacency List** – Uses lists of neighbors.

✔ **Types of Graphs:**

* **Directed Graph (Digraph)** – Edges have a direction.
* **Undirected Graph** – Edges do not have a direction.
* **Weighted Graph** – Each edge has a weight (cost).

✔ **Used in:**

* Social Networks (Facebook, LinkedIn)
* Shortest Path Problems (Google Maps, Dijkstra's Algorithm)

**Example of Graph Representation:**

plaintext

CopyEdit

(A)---(B)

| |

(C)---(D)

**Comparison: Linear vs. Non-Linear Data Structures**

| **Feature** | **Linear Data Structure** | **Non-Linear Data Structure** |
| --- | --- | --- |
| **Storage** | Sequential (one level) | Hierarchical or connected |
| **Traversal** | Simple and direct | Complex traversal (DFS, BFS) |
| **Memory Usage** | Efficient for small data | Can be memory-intensive |
| **Examples** | Arrays, Linked Lists, Stacks, Queues | Trees, Graphs |

**Conclusion**

The choice between **linear** and **non-linear** data structures depends on the **problem requirements**. Linear structures like **arrays and linked lists** are simple and efficient for ordered data, whereas non-linear structures like **trees and graphs** are essential for complex relationships and hierarchical storage. Understanding their differences helps in choosing the best structure for a given application.

**Arrays and Its Types**

An **array** is one of the most fundamental and widely used **data structures** in programming. It provides a way to store multiple values of the same data type in a contiguous memory location.

**1. What is an Array?**

An **array** is a collection of elements of the same data type, stored in **contiguous memory locations** and accessible using an **index**.

**Characteristics of an Array:**

✔ **Fixed Size** – The size of the array is defined at the time of declaration.  
✔ **Same Data Type** – All elements must be of the same data type.  
✔ **Indexed Access** – Elements are accessed using an index (starting from 0 in most programming languages).  
✔ **Efficient for Iteration** – Arrays allow fast access and traversal using loops.

**Syntax of Array Declaration (C Language)**

c

CopyEdit

data\_type array\_name[size];

**Example:**

c

CopyEdit

int arr[5]; // Declaring an array of size 5

**2. Types of Arrays**

Arrays can be classified into the following types:

**(A) One-Dimensional (1D) Array**

A **one-dimensional array** is the simplest form, where elements are stored in a single row.

✔ **Syntax:**

c

CopyEdit

data\_type array\_name[size];

✔ **Example:**

c

CopyEdit

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

✔ **Accessing Elements:**

c

CopyEdit

printf("%d", numbers[2]); // Output: 30 (Index starts from 0)

✔ **Operations on 1D Array:**

* **Insertion** – Adding an element.
* **Deletion** – Removing an element.
* **Traversal** – Accessing all elements one by one.
* **Searching** – Finding an element (Linear Search, Binary Search).

**(B) Two-Dimensional (2D) Array (Matrix)**

A **two-dimensional array** is used to store data in a **tabular (row & column) form**.

✔ **Syntax:**

c

CopyEdit

data\_type array\_name[rows][columns];

✔ **Example:**

c

CopyEdit

int matrix[3][3] = {

{1, 2, 3},

{4, 5, 6},

{7, 8, 9}

};

✔ **Accessing Elements:**

c

CopyEdit

printf("%d", matrix[1][2]); // Output: 6 (Row index 1, Column index 2)

✔ **Operations on 2D Array:**

* **Matrix Addition, Subtraction, Multiplication**
* **Searching an Element**
* **Traversing the Matrix**

**(C) Multi-Dimensional Array (3D and Higher)**

A **multi-dimensional array** is an extension of 2D arrays, allowing more levels of data organization.

✔ **Syntax:**

c

CopyEdit

data\_type array\_name[depth][rows][columns];

✔ **Example:**

c

CopyEdit

int cube[2][3][3] = {

{ {1,2,3}, {4,5,6}, {7,8,9} },

{ {10,11,12}, {13,14,15}, {16,17,18} }

};

✔ **Accessing Elements:**

c

CopyEdit

printf("%d", cube[1][2][1]); // Output: 17

✔ **Uses:**

* 3D games, Graphics Processing
* Scientific computations

**(D) Dynamic Arrays**

A **dynamic array** allows resizing at runtime (unlike static arrays).

✔ **Implemented using pointers and memory allocation (malloc, calloc in C)**

✔ **Example (C language using malloc)**

c

CopyEdit

int \*arr;

arr = (int\*)malloc(5 \* sizeof(int)); // Allocates memory for 5 integers

✔ **Uses:**

* When the array size is not known in advance.
* When dynamic memory management is required.

**3. Advantages & Disadvantages of Arrays**

**Advantages:**

✔ **Fast Access** – Direct access using index.  
✔ **Efficient Traversal** – Can be easily looped through.  
✔ **Memory Efficiency** – No extra overhead like linked lists.

**Disadvantages:**

✖ **Fixed Size** – Cannot resize after declaration (except dynamic arrays).  
✖ **Insertion & Deletion Costly** – Requires shifting elements.  
✖ **Wasted Memory** – If allocated size is more than required.

**4. Applications of Arrays**

✔ **Storing and processing data** (marks of students, temperatures, etc.)  
✔ **Mathematical computations** (matrices, statistics, etc.)  
✔ **Used in various algorithms** (sorting, searching, dynamic programming)  
✔ **Image processing** (storing pixel values)

**Conclusion**

Arrays are a fundamental data structure that allows efficient storage and access of elements. They come in different forms, such as **1D, 2D, multi-dimensional, and dynamic arrays**, each with unique applications. Understanding arrays is essential for mastering data structures and algorithms.

**Memory Allocation and Address Calculation of Array**

Arrays are stored in **contiguous memory locations**, meaning elements are placed next to each other in memory. Understanding how memory allocation and address calculation work is crucial for efficient programming and memory management.

**1. Memory Allocation of Arrays**

**(A) Static Memory Allocation (Compile-Time Allocation)**

* The memory for the array is allocated **before the program runs**.
* The size of the array is **fixed at compile time**.
* It is stored in the **stack** section of memory.

✔ **Example (C Language):**

c

CopyEdit

int arr[5]; // Allocates memory for 5 integers (5 × 4 bytes = 20 bytes)

✔ **Characteristics:**  
✔ Fast access  
✔ No manual deallocation needed  
✖ Wastes memory if size is too large  
✖ Cannot resize at runtime

**(B) Dynamic Memory Allocation (Run-Time Allocation)**

* The memory for the array is allocated **during program execution**.
* The size of the array can be **modified at runtime**.
* It is stored in the **heap** section of memory.

✔ **Example (C using malloc):**

c

CopyEdit

int \*arr;

arr = (int\*) malloc(5 \* sizeof(int)); // Allocates memory for 5 integers

✔ **Characteristics:**  
✔ Flexible size  
✔ Efficient memory usage  
✖ Requires manual deallocation (free())  
✖ Slightly slower than static allocation

✔ **Deallocating Memory:**

c

CopyEdit

free(arr); // Releases allocated memory

**2. Address Calculation of Array Elements**

Every element in an array has a **memory address**, which can be calculated using **pointer arithmetic**.

**(A) Address Calculation Formula**

If the **base address** (starting address of the array) is B and each element takes w bytes, then the address of the i-th element is:

**For One-Dimensional (1D) Arrays**

**Address of arr[i]** = Base Address + (i × w)  
✔ i = index of the element  
✔ w = size of each element (in bytes)

✔ **Example:**

c

CopyEdit

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

✔ Assuming the **base address** of arr[0] is 1000 and int takes 4 bytes:

| **Index (i)** | **Element** | **Address Formula** | **Actual Address** |
| --- | --- | --- | --- |
| 0 | 10 | 1000 + (0×4) | 1000 |
| 1 | 20 | 1000 + (1×4) | 1004 |
| 2 | 30 | 1000 + (2×4) | 1008 |
| 3 | 40 | 1000 + (3×4) | 1012 |
| 4 | 50 | 1000 + (4×4) | 1016 |

✔ **Accessing Addresses in C:**

c

CopyEdit

printf("%p", &arr[2]); // Output: Address of arr[2]

**(B) Address Calculation for Two-Dimensional (2D) Arrays**

For a **2D array**, memory is allocated in a **row-major order** (one row after another).

✔ **Formula for Address Calculation in Row-Major Order:**  
**Address of arr[i][j]** = Base Address + [(i × Total\_Columns) + j] × w

✔ **Example:**

c

CopyEdit

int matrix[3][3] = {

{1, 2, 3},

{4, 5, 6},

{7, 8, 9}

};

✔ Assuming **base address** = 2000, int size = 4 bytes:

| **Row (i)** | **Column (j)** | **Element** | **Address Formula** | **Actual Address** |
| --- | --- | --- | --- | --- |
| 0 | 0 | 1 | 2000 + [(0×3) + 0]×4 | 2000 |
| 0 | 1 | 2 | 2000 + [(0×3) + 1]×4 | 2004 |
| 0 | 2 | 3 | 2000 + [(0×3) + 2]×4 | 2008 |
| 1 | 0 | 4 | 2000 + [(1×3) + 0]×4 | 2012 |
| 1 | 1 | 5 | 2000 + [(1×3) + 1]×4 | 2016 |
| 1 | 2 | 6 | 2000 + [(1×3) + 2]×4 | 2020 |

✔ **Accessing Addresses in C:**

c

CopyEdit

printf("%p", &matrix[1][2]); // Output: Address of matrix[1][2]

**3. Pointer Representation of Arrays**

**(A) Pointer Notation for 1D Arrays**

An array name itself acts as a pointer to the first element.

✔ **Example:**

c

CopyEdit

int arr[3] = {10, 20, 30};

printf("%p", arr); // Address of arr[0]

printf("%p", arr + 1); // Address of arr[1]

✔ **Dereferencing Elements Using Pointers:**

c

CopyEdit

printf("%d", \*(arr + 1)); // Output: 20

**(B) Pointer Notation for 2D Arrays**

A 2D array is internally stored as a **1D array** in memory.

✔ **Example:**

c

CopyEdit

int matrix[2][2] = {{1, 2}, {3, 4}};

printf("%d", \*(\*(matrix + 1) + 1)); // Output: 4

✔ Explanation:

* matrix + 1 moves to the **second row**.
* \*(matrix + 1) gives the **base address of the second row**.
* \*(matrix + 1) + 1 moves to the **second column**.
* \*(\*(matrix + 1) + 1) retrieves the **value at that position** (4).

**4. Difference Between Static and Dynamic Arrays**

| **Feature** | **Static Array** | **Dynamic Array** |
| --- | --- | --- |
| **Memory Allocation** | Compile-time (Stack) | Run-time (Heap) |
| **Size** | Fixed | Flexible |
| **Efficiency** | Fast (Direct Access) | Slower (Heap Allocation) |
| **Deallocation** | Automatic | Manual (free()) |
| **Example** | int arr[10]; | int \*arr = (int\*)malloc(10 \* sizeof(int)); |

**5. Conclusion**

Understanding **memory allocation and address calculation** is essential for optimizing performance and managing memory efficiently. **Static arrays** are fast but rigid, whereas **dynamic arrays** are flexible but require manual memory management. **Address calculation** formulas help in understanding how elements are stored in memory and accessed efficiently.

**Sparse Array**

A **sparse array** (or **sparse matrix**) is a type of array in which **most of the elements are zero (or some default value)**. Instead of storing all elements (including zeros), we store only the **non-zero elements along with their positions**, which reduces memory usage.

**1. Characteristics of Sparse Arrays**

✔ **Large Size** – Sparse arrays are typically used when dealing with large datasets.  
✔ **Mostly Zeros** – The majority of the elements in the array are zero (or default values).  
✔ **Memory Efficient** – Instead of storing unnecessary zeroes, only non-zero elements and their positions are stored.  
✔ **Efficient Computation** – Operations like addition, multiplication, and searching become optimized when implemented efficiently.

**2. Example of a Sparse Array**

Consider a **5×5 matrix** where most elements are zero:

[0030000005070000000090020]\begin{bmatrix} 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5 \\ 0 & 7 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 9 & 0 & 0 & 2 & 0 \end{bmatrix}​00009​00700​30000​00002​05000​​

Here, only **5 elements** are non-zero:

* (0,2) → 3
* (1,4) → 5
* (2,1) → 7
* (4,0) → 9
* (4,3) → 2

Instead of storing all **25 elements**, we store only **non-zero values with their positions**.

**3. Representation of Sparse Array**

Sparse arrays can be stored efficiently using different representations:

**(A) Triplet Representation (Row, Column, Value)**

We store only **non-zero elements** along with their **row and column indices**.

| **Row** | **Column** | **Value** |
| --- | --- | --- |
| 0 | 2 | 3 |
| 1 | 4 | 5 |
| 2 | 1 | 7 |
| 4 | 0 | 9 |
| 4 | 3 | 2 |

✔ **Memory Required:** Instead of storing a 5×5 = 25 matrix, we store only 5×3 = 15 values.

✔ **Example Code in C:**

c

CopyEdit

#include <stdio.h>

#define MAX 5 // Maximum non-zero elements

// Function to print sparse matrix in triplet form

void printSparseMatrix(int sparse[MAX][3], int n) {

printf("Row\tColumn\tValue\n");

for (int i = 0; i < n; i++)

printf("%d\t%d\t%d\n", sparse[i][0], sparse[i][1], sparse[i][2]);

}

int main() {

// Sparse matrix in triplet form

int sparse[MAX][3] = {

{0, 2, 3},

{1, 4, 5},

{2, 1, 7},

{4, 0, 9},

{4, 3, 2}

};

// Printing sparse matrix

printSparseMatrix(sparse, MAX);

return 0;

}

**(B) Linked List Representation**

Instead of using arrays, we store **each non-zero element as a node** in a linked list.

✔ **Structure of a Node (C Language)**:

c

CopyEdit

struct Node {

int row, col, value;

struct Node \*next;

};

✔ **Advantages of Linked List Representation**:

* Memory-efficient (no wasted space for zero elements).
* Can be dynamically updated (insert or delete non-zero elements).
* Useful for very large matrices.

**(C) Compressed Sparse Row (CSR) Representation**

* Used in **scientific computing** and **machine learning**.
* Stores:
  1. **Values array** – Non-zero elements.
  2. **Column indices array** – Column numbers of non-zero elements.
  3. **Row pointer array** – Position of first non-zero element in each row.

✔ **Example:**  
For the given **5×5** matrix, we store:

| **Values** | **3, 5, 7, 9, 2** |
| --- | --- |
| Column Indices | 2, 4, 1, 0, 3 |
| Row Pointer | 0, 1, 2, 3, 3, 5 |

✔ **Memory Required:** O(non-zero elements)

✔ **Usage:**

* Solving large **linear equations** (scientific computing).
* **Machine learning** (storing sparse feature vectors).
* **Graph representation** (Adjacency Matrix in graphs).

**4. Operations on Sparse Arrays**

✔ **(A) Addition of Sparse Arrays**

* If two matrices have a **non-zero element at the same position**, we **add their values**.
* If a non-zero element is **present in only one matrix**, it is directly included in the result.

✔ **Example in Triplet Form:**

c

CopyEdit

// Given matrices A and B

A = (0,2,3), (1,4,5), (2,1,7)

B = (0,2,2), (2,1,3), (4,3,2)

// Addition result:

Result = (0,2,5), (1,4,5), (2,1,10), (4,3,2)

✔ **(B) Transpose of a Sparse Array**

* Swaps **rows and columns** of the sparse matrix.
* Useful in **graph algorithms** and **image processing**.

✔ **(C) Multiplication of Sparse Arrays**

* Used in **image processing, physics simulations**.

✔ **(D) Searching an Element**

* Instead of checking all N×M elements, we only search non-zero elements (O(non-zero elements) time complexity).

**5. Advantages & Disadvantages of Sparse Arrays**

**✔ Advantages:**

✅ **Memory Efficient** – Saves space by not storing zero values.  
✅ **Fast Computation** – Operations (addition, multiplication) are faster due to fewer elements.  
✅ **Used in Large Datasets** – Found in machine learning, scientific computing.

**✖ Disadvantages:**

❌ **Complex Implementation** – Requires special data structures.  
❌ **Indexing Complexity** – Accessing an element is slower than regular arrays.  
❌ **Modification Overhead** – Inserting/deleting elements is harder than simple arrays.

**6. Applications of Sparse Arrays**

✔ **Graph Representation** – Adjacency matrices in large graphs.  
✔ **Machine Learning** – Storing sparse feature vectors.  
✔ **Image Processing** – Storing large image pixels with mostly empty regions.  
✔ **Computational Physics** – Solving large matrix equations.  
✔ **Search Engines** – Storing large text data efficiently.

**7. Conclusion**

A **sparse array** efficiently stores and processes data where most elements are **zero**. It is widely used in **scientific computing, machine learning, and graph algorithms**. Different representations like **triplet form, linked list, and compressed row format (CSR)** optimize performance and memory usage.

**Linked List: Types & Operations**

A **linked list** is a **linear data structure** where elements (nodes) are stored **dynamically in memory** and linked using pointers. Unlike arrays, linked lists are **not stored in contiguous memory locations**, making insertion and deletion operations more efficient.

**1. Structure of a Linked List Node**

Each node in a linked list consists of:

1. **Data** – Stores the actual value.
2. **Pointer (Next)** – Stores the address of the next node.

✔ **Basic Structure in C:**

c

CopyEdit

struct Node {

int data;

struct Node\* next;

};

**2. Types of Linked Lists**

**(A) Singly Linked List (SLL)**

✔ Each node has **only one pointer (next)** pointing to the next node.  
✔ Operations like **insertion, deletion, and traversal** are efficient.

✔ **Example:**

cpp

CopyEdit

10 → 20 → 30 → 40 → NULL

✔ **Structure in C:**

c

CopyEdit

struct Node {

int data;

struct Node\* next;

};

✔ **Advantages:**  
✅ Less memory usage (only one pointer per node).  
✅ Efficient insertion/deletion at the beginning.

✔ **Disadvantages:**  
❌ Cannot traverse backward (one-way direction).  
❌ Deletion in the middle requires traversal from the start.

**(B) Doubly Linked List (DLL)**

✔ Each node has **two pointers**:

* **next** (points to the next node)
* **prev** (points to the previous node)

✔ **Example:**

cpp

CopyEdit

NULL ← 10 ↔ 20 ↔ 30 ↔ 40 → NULL

✔ **Structure in C:**

c

CopyEdit

struct Node {

int data;

struct Node\* next;

struct Node\* prev;

};

✔ **Advantages:**  
✅ Can traverse **forward and backward**.  
✅ More efficient deletion (does not require full traversal).

✔ **Disadvantages:**  
❌ More memory consumption (extra pointer).  
❌ More complex to implement.

**(C) Circular Linked List (CLL)**

✔ The **last node points to the first node**, forming a **loop**.  
✔ Can be **singly or doubly linked**.

✔ **Example (Singly Circular Linked List):**

css

CopyEdit

10 → 20 → 30 → 40 ↻ (back to 10)

✔ **Advantages:**  
✅ No null pointers (can keep traversing).  
✅ Good for applications requiring **continuous cyclic traversal** (e.g., round-robin scheduling).

✔ **Disadvantages:**  
❌ Requires extra logic to stop traversal.  
❌ More complex than singly linked lists.

**(D) Circular Doubly Linked List (CDLL)**

✔ Combines features of **Doubly and Circular Linked List**.  
✔ The **last node points to the first**, and the **first node’s prev points to the last**.

✔ **Example:**

pgsql

CopyEdit

NULL ← 10 ↔ 20 ↔ 30 ↔ 40 ↻ (back to 10)

✔ **Advantages:**  
✅ Can traverse **both directions indefinitely**.  
✅ No need for a special condition for the end.

✔ **Disadvantages:**  
❌ More memory overhead (extra pointer).

**3. Operations on Linked Lists**

**(A) Insert Operation in Linked List**

**1. Insertion at the Beginning**

✔ Create a new node and make it the **head**.  
✔ Update the next pointer of the new node to point to the **previous head**.

✔ **Code (Singly Linked List):**

c

CopyEdit

void insertAtBeginning(struct Node\*\* head, int value) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = value;

newNode->next = \*head;

\*head = newNode;

}

✔ **Time Complexity:** O(1)

**2. Insertion at the End**

✔ Traverse to the last node and link the new node.

✔ **Code (Singly Linked List):**

c

CopyEdit

void insertAtEnd(struct Node\*\* head, int value) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

struct Node\* temp = \*head;

newNode->data = value;

newNode->next = NULL;

if (\*head == NULL) {

\*head = newNode;

return;

}

while (temp->next != NULL)

temp = temp->next;

temp->next = newNode;

}

✔ **Time Complexity:** O(n)

**3. Insertion at a Specific Position**

✔ Traverse to the desired position and insert the node.

✔ **Code (Singly Linked List):**

c

CopyEdit

void insertAtPosition(struct Node\*\* head, int value, int pos) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

struct Node\* temp = \*head;

newNode->data = value;

if (pos == 1) {

newNode->next = \*head;

\*head = newNode;

return;

}

for (int i = 1; i < pos - 1 && temp != NULL; i++)

temp = temp->next;

newNode->next = temp->next;

temp->next = newNode;

}

✔ **Time Complexity:** O(n)

**(B) Delete Operation in Linked List**

**1. Delete at the Beginning**

✔ Move head pointer to the next node and free memory.

✔ **Code:**

c

CopyEdit

void deleteAtBeginning(struct Node\*\* head) {

if (\*head == NULL) return;

struct Node\* temp = \*head;

\*head = temp->next;

free(temp);

}

✔ **Time Complexity:** O(1)

**2. Delete at the End**

✔ Traverse to the last node, update the second last node’s next to NULL.

✔ **Code:**

c

CopyEdit

void deleteAtEnd(struct Node\*\* head) {

if (\*head == NULL) return;

struct Node\* temp = \*head;

struct Node\* prev = NULL;

while (temp->next != NULL) {

prev = temp;

temp = temp->next;

}

prev->next = NULL;

free(temp);

}

✔ **Time Complexity:** O(n)

**3. Delete at a Specific Position**

✔ Traverse to the position and update links accordingly.

✔ **Code:**

c

CopyEdit

void deleteAtPosition(struct Node\*\* head, int pos) {

if (\*head == NULL) return;

struct Node\* temp = \*head;

if (pos == 1) {

\*head = temp->next;

free(temp);

return;

}

struct Node\* prev = NULL;

for (int i = 1; temp != NULL && i < pos; i++) {

prev = temp;

temp = temp->next;

}

prev->next = temp->next;

free(temp);

}

✔ **Time Complexity:** O(n)

**(C) Traverse (Display) a Linked List**

✔ Start from the head and print each node.

✔ **Code:**

c

CopyEdit

void traverse(struct Node\* head) {

struct Node\* temp = head;

while (temp != NULL) {

printf("%d -> ", temp->data);

temp = temp->next;

}

printf("NULL\n");

}

✔ **Time Complexity:** O(n)

**4. Conclusion**

✔ **Linked lists** provide **efficient insertion and deletion** but require extra memory for pointers.  
✔ Different types (**Singly, Doubly, Circular**) offer various benefits.  
✔ Operations like **Insert, Delete, and Traverse** are fundamental for linked list manipulation.

**Introduction and Application of Stack and Queue**

**1. Introduction to Stack and Queue**

Stack and Queue are **linear data structures** used for data storage and retrieval. Both follow different rules for inserting and removing elements, making them useful in various applications like memory management, job scheduling, and expression evaluation.

**2. Stack**

**2.1 What is a Stack?**

A **stack** is a **Last In, First Out (LIFO)** data structure, meaning the last element inserted is the first to be removed.

✔ **Example:**  
Imagine a stack of plates; you always add and remove plates from the top.

✔ **Basic Stack Operations:**

* **Push:** Insert an element on the top of the stack.
* **Pop:** Remove the top element from the stack.
* **Peek (Top):** Get the value of the top element without removing it.
* **isEmpty:** Check if the stack is empty.
* **isFull:** Check if the stack is full (in case of an array-based implementation).

✔ **Stack Representation:**

markdown

CopyEdit

Stack (Top = 4)

--------------

| 40 | <- Top

| 30 |

| 20 |

| 10 |

--------------

**2.2 Applications of Stack**

✔ **1. Function Call Stack (Recursion):**

* Stores function calls in programming languages.
* Example: Recursive calls in C use a stack.

✔ **2. Undo-Redo in Editors:**

* Pressing **Ctrl+Z** (undo) in MS Word uses a stack.

✔ **3. Backtracking (Maze, Game AI, DFS in Graphs):**

* Used in solving mazes, puzzles, and graph traversal algorithms.

✔ **4. Expression Evaluation & Conversion:**

* Used in converting **infix** to **postfix** expressions.
* Example: Evaluating **(3 + 4) \* 5** using stacks.

✔ **5. Memory Management:**

* CPU uses a stack for storing function calls and local variables.

**3. Queue**

**3.1 What is a Queue?**

A **queue** is a **First In, First Out (FIFO)** data structure, meaning the first element inserted is the first to be removed.

✔ **Example:**  
Imagine a queue at a bank; the first person in line gets served first.

✔ **Basic Queue Operations:**

* **Enqueue:** Insert an element at the rear of the queue.
* **Dequeue:** Remove an element from the front of the queue.
* **Front (Peek):** Get the front element without removing it.
* **isEmpty:** Check if the queue is empty.
* **isFull:** Check if the queue is full (in case of an array-based queue).

✔ **Queue Representation:**

rust

CopyEdit

Front -> 10 | 20 | 30 | 40 <- Rear

**3.2 Applications of Queue**

✔ **1. Job Scheduling (CPU Scheduling, Printer Queue):**

* **First come, first served** scheduling in operating systems.
* Printing jobs are handled using a queue.

✔ **2. Data Buffering (IO Buffers, Disk Scheduling):**

* Used in streaming video and audio (Netflix, YouTube).

✔ **3. Process Management in OS (Ready Queue, IO Queue):**

* Operating systems use queues for handling processes.

✔ **4. Network Packet Scheduling:**

* Routers manage network traffic using queues.

✔ **5. BFS (Breadth-First Search) in Graphs:**

* Used for traversing graphs and finding shortest paths.

**4. Differences Between Stack and Queue**

| **Feature** | **Stack** | **Queue** |
| --- | --- | --- |
| **Data Structure Type** | LIFO (Last In, First Out) | FIFO (First In, First Out) |
| **Insertion** | Push (at the top) | Enqueue (at the rear) |
| **Deletion** | Pop (from the top) | Dequeue (from the front) |
| **Access** | Only the top element can be accessed | Only the front element can be accessed |
| **Example** | Function calls, expression evaluation | CPU scheduling, BFS in graphs |

**5. Conclusion**

✔ **Stack** follows **LIFO**, useful for function calls, undo-redo, and backtracking.  
✔ **Queue** follows **FIFO**, used in scheduling, buffering, and BFS.  
✔ Both are fundamental **linear data structures** with real-world applications.