ARRAYS: REPRESENTATION, OPERATIONS, AND APPLICATIONS

Arrays are one of the most fundamental data structures in computer science. They are essential in organizing data so that it can be efficiently accessed and manipulated. An **array** is a collection of elements, typically of the same data type, stored at contiguous memory locations. Arrays allow constant-time access to any element if the index is known, making them highly efficient for certain types of operations.

Representation of Arrays

1. Single-Dimensional Arrays

A **single-dimensional array** is a linear structure where elements are stored in consecutive memory locations. The array has a fixed size, and each element can be accessed using a unique index. The index starts at 0 and increases by 1 for each subsequent element.

For instance, if an array contains 5 elements, each element is stored at a specific memory location that can be directly accessed using the index.

Memory Representation

In memory, arrays are stored in contiguous blocks of memory. For an array A of size n, the memory address of each element

can be calculated using a simple formula based on the base address of the array and the size of each element.

Base Address: The starting address where the array is stored in memory.
Element Size: The number of bytes used to store one element (dependent on the data type).

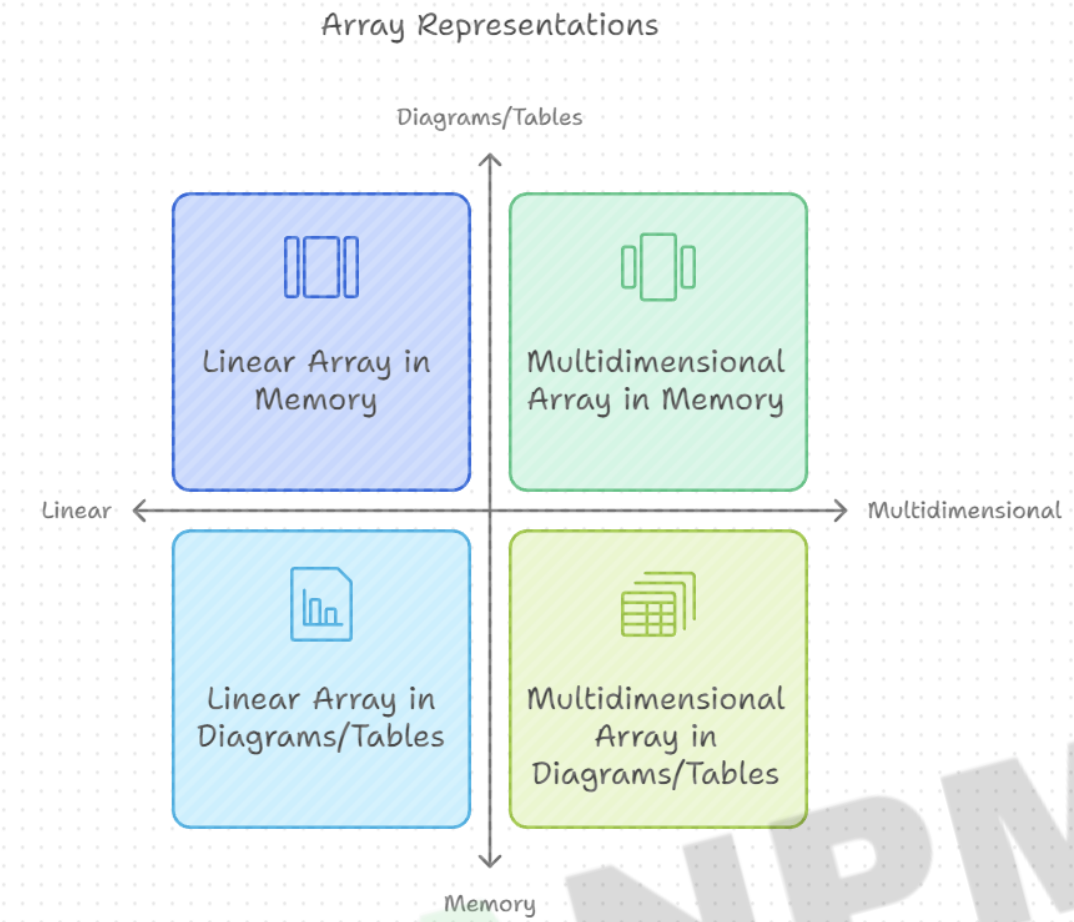
The address of any element A[i] in the array can be calculated using:

- **Address of A[i] = Base Address + (i * Size of Element)**

2. Multi-Dimensional Arrays

A **multi-dimensional array** is essentially an array of arrays. The most commonly used multi-dimensional arrays are **2D arrays**, which can be visualized as a table with rows and columns. A 2D array is used to store data in a matrix form.

For example, in a 2D array B of dimensions m x n, m is the number of rows and n is the number of columns. To access an element, you need two indices: one for the row and one for the column.



Memory Representation of 2D Arrays

In memory, a 2D array can be stored in two ways:

1. **Row-Major Order**: This stores all the elements of a row contiguously in memory. After storing one row, the next row is stored.
2. **Column-Major Order**: This stores all the elements of a column contiguously in memory. After storing one column, the next column is stored.

Address Calculation for Multi-Dimensional Arrays

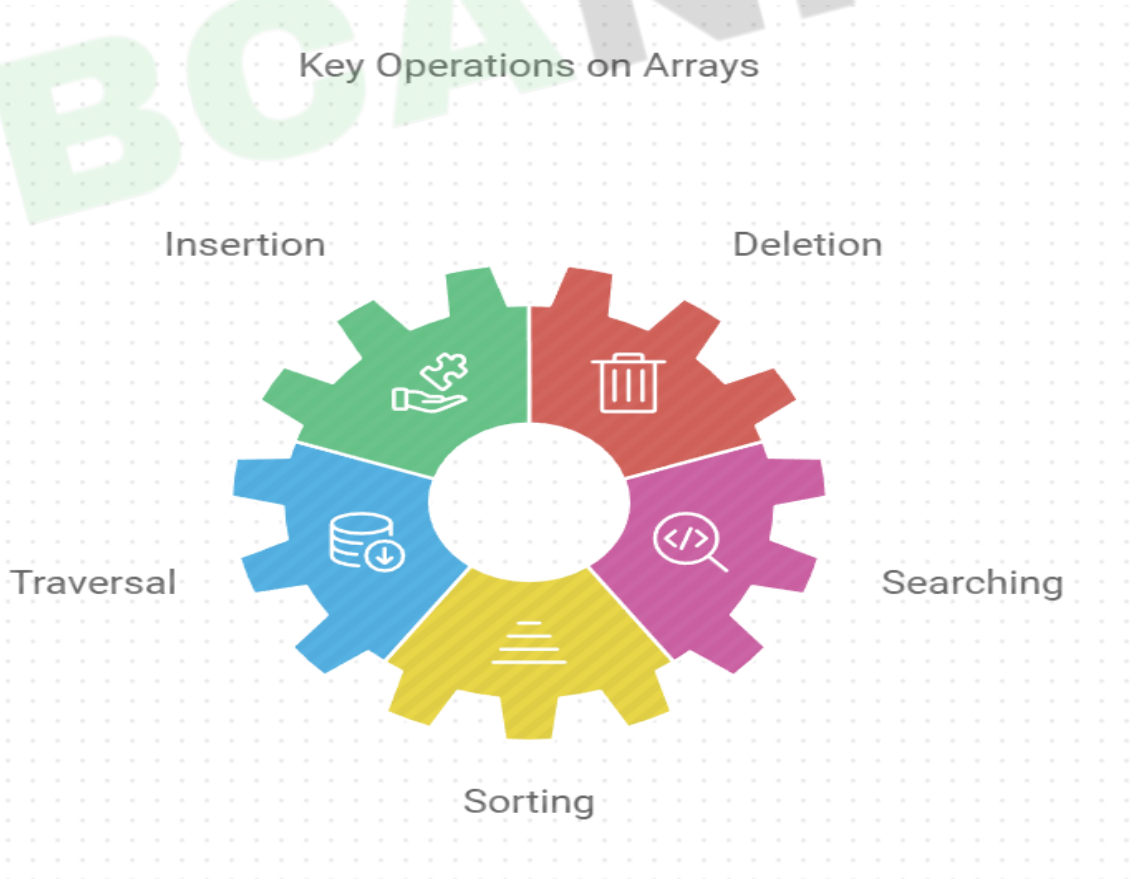
The formula for calculating the address of an element in a 2D array depends on whether the array is stored in row-major or column-major order:

- **Row-Major Order**:
    - Address of B[i][j] = Base Address + [(i * n) + j] * Size of Element Where ‘n’ is the total number of columns.
- **Column-Major Order**:
    - Address of B[i][j] = Base Address + [(j * m) + i] * Size of Element Where ‘m’ is the total number of rows.

For higher-dimensional arrays, similar principles apply, but more indices are involved for each dimension.

Operations on Arrays

Arrays support various operations that allow for effective data manipulation. Here are some common operations performed on arrays:



1. Traversing

Traversal refers to visiting each element of the array. This operation is fundamental for most algorithms, as it allows accessing and processing each element.

2. Insertion

Insertion is the process of adding a new element to the array. However, since arrays are of fixed size, insertion can only be done in an available space or by shifting elements to make room for the new element.

- **At the beginning**: Requires shifting all the elements one position to the right, making space at index 0.
- **At the end**: Can be directly placed at the last available index.
- **In the middle**: Elements from the insertion point must be shifted to make room for the new element.

3. Deletion

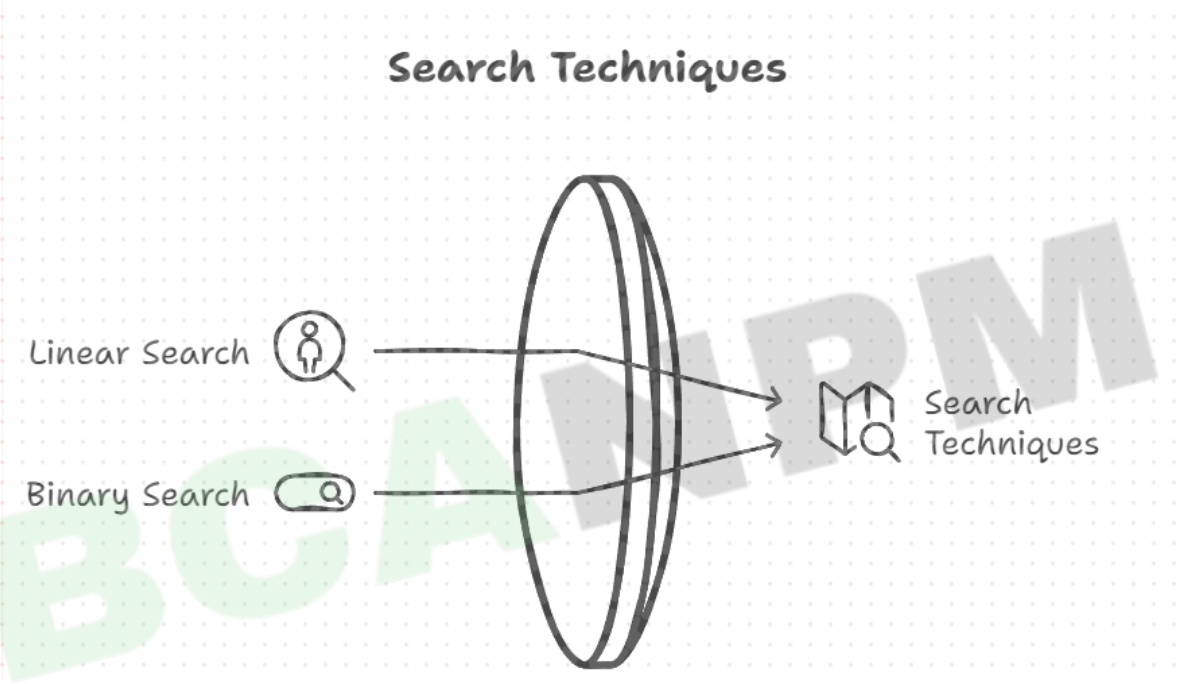
Deletion involves removing an element from the array. After the element is deleted, subsequent elements must be shifted to maintain the structure.

- **From the beginning**: Remove the element at index 0 and shift the rest to the left.
- **From the end**: Simply reduce the size of the array.
- **From the middle**: Remove the desired element and shift subsequent elements leftward.

4. Searching

Searching is finding the index of a specific element. There are two primary search techniques:

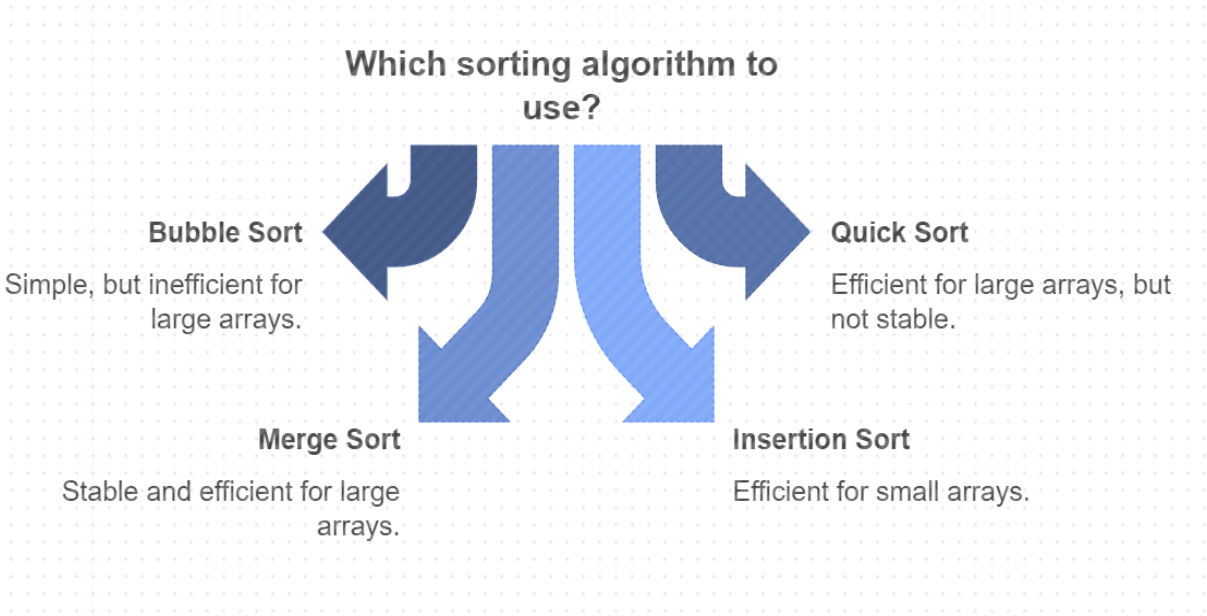
- **Linear Search**: Sequentially check each element until the desired one is found.
- **Binary Search**: Efficient for sorted arrays. This method splits the array into halves and discards one half at each step.



5. Sorting

Sorting involves rearranging the array elements in a specific order, typically in ascending or descending order. Common sorting algorithms include:

- **Bubble Sort**
- **Selection Sort**
- **Quick Sort**
- **Merge Sort**



Vectors

A **vector** is a dynamic array that can change its size automatically when elements are added or removed. Vectors can grow or shrink in size dynamically, unlike regular arrays that have a fixed size. They are more flexible but can have a slight performance overhead due to resizing operations.

Applications of Arrays

Arrays play a significant role in various real-world applications due to their efficient data organization and access. Here are some common applications:

1. Matrix Multiplication

Arrays, especially 2D arrays, are frequently used to represent matrices. Matrix multiplication is a fundamental operation in many fields, including computer graphics, physics simulations, and machine learning.

Matrix multiplication involves taking the dot product of rows and columns from two matrices to produce a new matrix. If A

is an m x n matrix and B is an n x p matrix, their product will be an m x p matrix.

2. Sparse Polynomials

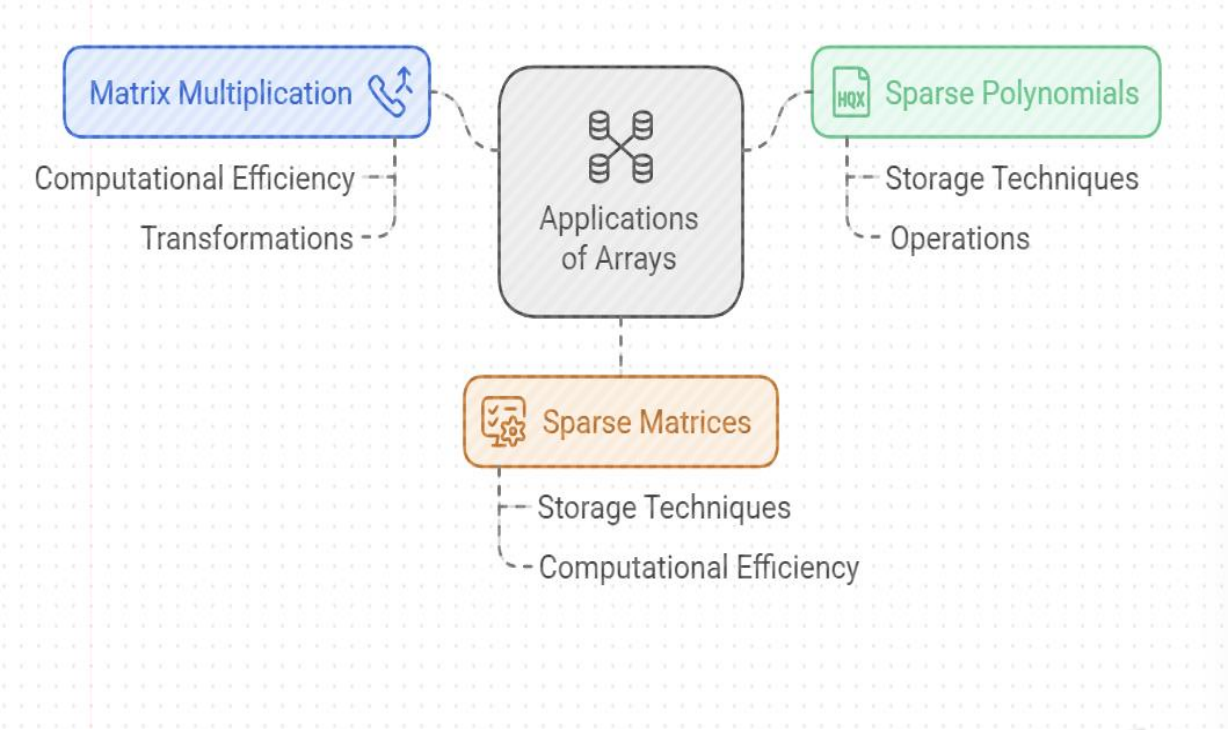
A **sparse polynomial** is a polynomial where most of the coefficients are zero. Instead of using arrays that store every coefficient, sparse polynomials use arrays that store only the non-zero coefficients and their corresponding exponents.

This helps optimize memory usage and computational efficiency, especially for very large polynomials with a high degree but relatively few non-zero terms.

3. Sparse Matrices

In mathematics, a **sparse matrix** is one in which most elements are zero. For large matrices, storing every element (including zeros) can waste memory. Sparse matrix representation techniques use arrays that store only non-zero elements along with their row and column indices.

Sparse matrices are common in fields such as network theory, machine learning, and scientific computing. Operations on sparse matrices include addition, multiplication, and transposition, which are all optimized using sparse representations.

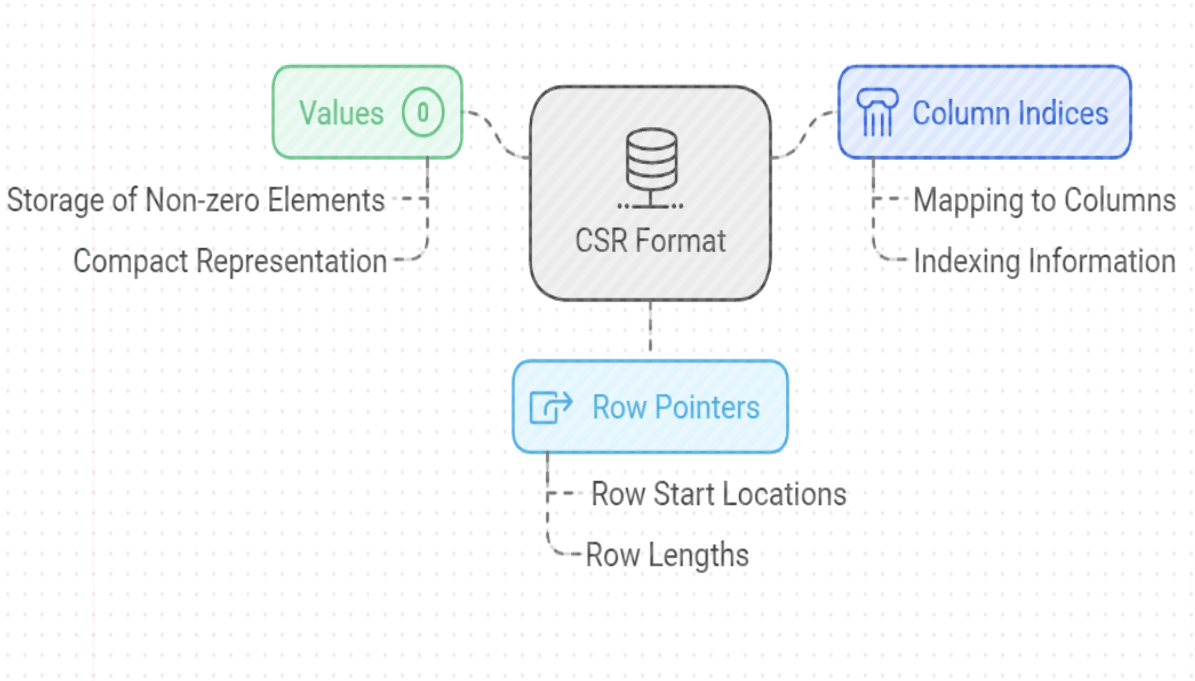


Array Operations for Sparse Matrix Representation

Sparse matrices can be stored in memory using arrays in an efficient manner. For example, one popular format is the **Compressed Sparse Row (CSR)** format, which uses three arrays:

- **Array of non-zero values**: Stores only the non-zero elements of the matrix.
- **Array of column indices**: Stores the column index corresponding to each non-zero element.
- **Array of row pointers**: Stores the index in the non-zero values array where each row starts.

Using these arrays, operations such as matrix addition and multiplication can be performed more efficiently on sparse matrices.


Values
0
Storage of Non-zero Elements
Compact Representation
CSR Format
Column Indices
Mapping to Columns
Indexing Information
Row Pointers
Row Start Locations
Row Lengths