NEP NEW SYLLABUS

A Book Of

Data Structure and Algorithms

For M.C.A. (Management) : Semester - I

[Course Code IT 12 : Credit - 03]

As Per New Syllabus of NEP 2020

w.e.f. Academic Year 2024-25

Dr. Manisha Bharambe
M.Sc. (Comp. Sci), M.Phil, Ph.D (Comp. Sci)
Vice Principal and Associate Professor
Computer Science Department
Abasaheb Garware College, Pune - 4

Dr. Jagdish Sangvikar
M.Sc., B.Ed., M.Phil., Ph.D., NET
Department of Computer Science
VPASC College, Baramati

Mr. Vijay T. Patil
M.E. (Computer Engineering)
Head, Department of Computer Engineering
(NBA Accredited)
Vidyalankar Polytechnic
Wadala, Mumbai-37

Supriya G. Sapa
B.E. (CSE), M.Tech (IT)
Asst. Prof Institute of Management Studies
Career Development & Research
Ahmednagar

Price ₹ 270.00

NIRALI PRAKASHAN
ADVANCEMENT OF KNOWLEDGE

N3362

Syllabus ...

1. Arrays/List **[Weightage 15]**

- 1.1 Introduction & Definition of an Array
- 1.2 Memory Allocation & Indexing
- 1.3 Operations on 1-D & 2D Arrays/Lists
- 1.4 Arrays and Their Applications
- 1.5 Sparse Matrices
- 1.6 String manipulation using arrays

2. Linked Lists **[Weightage 20]**

- 2.1 Introduction
- 2.2 Definition of a Linked List
- 2.3 Memory Allocation in a Linked List
- 2.4 Types of Linked Lists
  - 2.4.1 Singly Linked List
  - 2.4.2 Operations on a Singly Linked List
  - 2.4.3 Circular Linked Lists
  - 2.4.4 Operations on a Circular Linked List
  - 2.4.5 Doubly Linked List
  - 2.4.6 Operations on a Doubly Linked List

3. Stacks and Queues **[Weightage 20]**

- 3.1 Introduction and Definition of a Stack
- 3.2 Implementation of a Stack
  - 3.2.1 Implementation of Stacks Using Arrays
  - 3.2.2 Implementation of Stacks Using Linked Lists
- 3.3 Applications of Stacks:
  - 3.3.1 Conversion of an expression (Infix, Prefix, Postfix)
  - 3.3.2 Evaluation of Expression
  - 3.3.3 String Reversal
- 3.4 Introduction and Definition of a Queue
- 3.5 Implementation of a Queue
  - 3.5.1 Implementation of Queues Using Arrays
  - 3.5.2 Implementation of Queues Using Linked Lists
- 3.6 Applications of Queues

4. Tree & Graph [Weightage 25]

- 4.1 Tree Definition, representation
- 4.2 Binary Search Tree and its operations
  - 4.2.1 Tree Traversal
  - 4.2.2 Insertion
  - 4.2.3 Deletion
  - 4.2.4 Search
- 4.3 AVL Tree and its operations
  - 4.3.1 Insertion
  - 4.3.2 Deletion
  - 4.3.3 Rotations
- 4.4 Directed and Undirected Graph
- 4.5 Graph Representations
  - 4.5.1 Adjacency Matrix
  - 4.5.2 Adjacency List
- 4.6 Graph Traversals
  - 4.6.1 BFS
  - 4.6.2 DFS

5. Searching and Sorting [Weightage 20]

- 5.1 Linear Search or Sequential Search
- 5.2 Binary Search
- 5.3 Interpolation Search
- 5.4 Introduction to Sorting
  - 5.5.1 Merge Sort
  - 5.5.2 Quick Sort
  - 5.5.3 Bubble Sort
- 5.5 Heap
  - 5.5.1 Min heap and Max heap
- 5.6 Hashing
  - 5.6.1 Hash Table
  - 5.6.2 Hash Functions

1... Arrays/List

Objectives...

- To study basic concepts of Linear and Non-linear Data Structures.
- To study the basic concepts of Array representation of an Array as ADT.
- To study various Operations on Arrays.
- To study different Applications of Arrays.
- To study memory representation of One-dimensional and Multidimensional Array.
- To study sparse matrices.
- To study about string manipulation using arrays.

1.1 CONCEPT OF LINEAR AND NON-LINEAR DATA ORGANIZATION

- The data is organized within the computer memory in linear or non-linear way. In linear or sequential organization, each element has unique successor. In non-linear organization each element may have one more successor (or predecessors). The Fig. 1.1 shows the data organization.

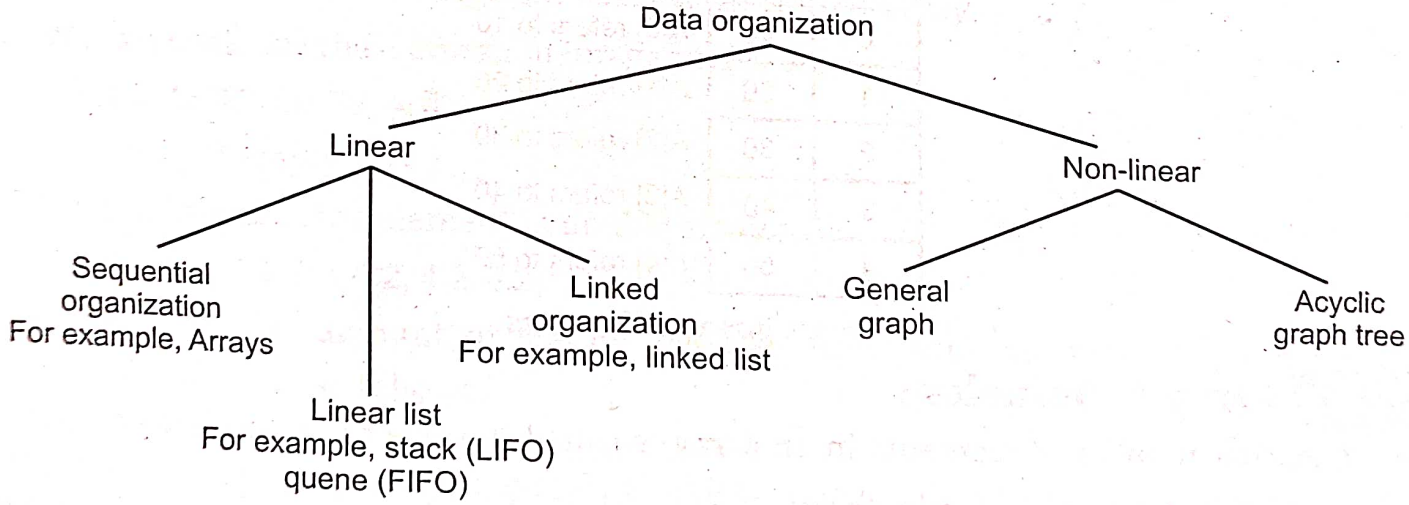


Fig. 1.1: Data Organization

- A data structure is said to be linear if its elements form a sequence or linear list. One way to represent relationship between elements by sequential memory location or other way to represent relationship between elements using pointers and links. For example: Arrays, List, Stacks, Queues etc.

- Non-linear organization is useful when we wish to represent the hierarchical or network relationship between the elements. For example, Trees and Graphs.
- In linear organization the elements are ordered. Direct or Random access to any element is the major advantage of sequential organization.
- Since, the elements are ordered in sequential organization the insertion and deletion of element is expensive. Hence, linked organization is used instead of sequential organization.
- Sequentially mapped data structure is also called as a Random Access data structure which means that the accessing time is independent of the size of the data structure and therefore, requires O(1) time.

1.2 INTRODUCTION AND DEFINITION OF ARRAY

- The array is a consecutive set of memory locations. An array is a set of pairs: index and value. For each index, there is a value associated with that index.

1.2.1 Definition

- An array is a **finite ordered collection of homogeneous data elements** which provides random access to the elements.
- Finite means specific number of elements. Ordered means elements are arrange one by one i.e. first then second and so on and homogeneous means elements are of same type.

  For example, In Python

```
A = [10, 20, 30, 40, 50]
```

- This declaration will create an array A, with five elements. The Fig. 1.2 shows the array of integers with capacity five.

Index	Array A	
0	10	A[0] refers to 10
1	20	A[1] refers to 20
2	30	A[2] refers to 30
3	40	A[3] refers to 40
4	50	A[4] refers to 50

Fig. 1.2: Array of integers

1.2.2 Array Terminology

- **Size:** The number of elements in an array is called the size of array. Size of array is also called as length or dimension, which is always mention in a square bracket[]. Dimension cannot be modified after compilation, hence array are called as static data structure.
- **Type:** Type is data type i.e. type of data to be stored in an array. For example, int, float etc.
- **Base:** The address of the first element of the array in memory is called base address. It varies at runtime and not defines by the programmer.

- **Index:** An element of an array is referenced by a subscript like A[i] for i^{th} element.
- **Range of index:** Range of index is from lower bound (0) to upper bound (n – 1) for dimension 'n'. For example, for A[10] the range of index is 0 to 9.

  The lower bound = Smallest index and Upper bound = Largest index of an array.

1.3 MEMORY ALLOCATION AND INDEXING

1.3.1 Array Initialization

- Arrays can be initialize at the time of declaration.
- **For example,**

  1. `A = [15, 25, 30, 35, 40]`
  2. `B = [5.2, 3.2, 1.2, 6.2, 4.2, 8.2]`

- The values assigned to each of the elements of above arrays:

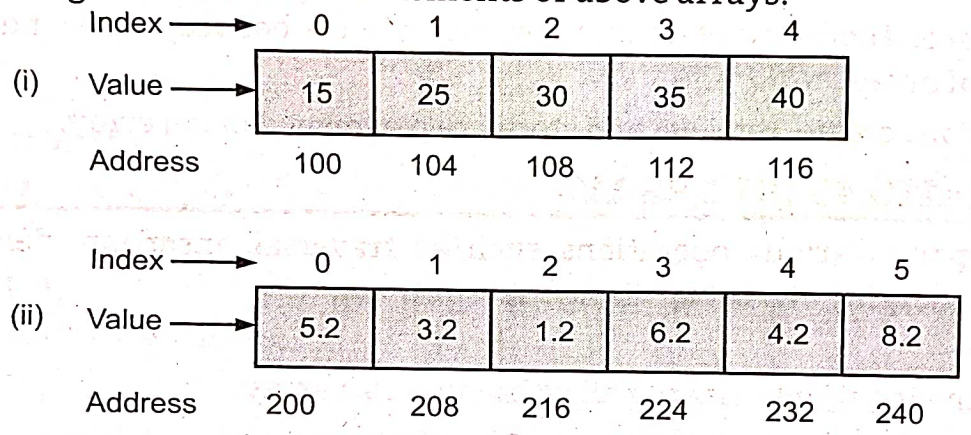


Fig. 1.3: Representation of array

- We can initialize the element in integer array as:

  A = [15, 25, 30, 35, 40]

  A = [ ] // empty array
- We can initialize the element in double array as:

  B = [5.2, 3.2, 1.2, 6.2, 4.2, 8.2]
- In 'Python', the amount of storage depends upon the data type of the array. In 'Python', size are as follows:

  int → 4 bytes (see address field of Fig. 1.3 (i)).

  double → 8 bytes (see address field of Fig. 1.3 (ii)).

1.3.2 Advantages of Array in Program

1. Easy to manage large amount of information, rather than having separate variable names.
2. Using index field, each element can be easily accessed.
3. Direct access is possible.

1.4 ARRAY AS AN ADT

- Abstract definition of array we can say, array is a set of pairs i.e. index and value. There is one-to-one correspondence between elements and index. The primitive operations on arrays are create, store and retrieve.
  1. **Create:** The create function creates an empty array or new array, which needs information regarding type of data objects and size of array.
  2. **Retrieve:** It takes the input as an array and an index and either return the appropriate value or an error.
  3. **Store:** It is used to enter new index value pairs or data object in array.

  Formally, the array ADT is defined as collection of domain, structure and operations.
  4. **Domain:** Domain is a collection of fixed number of components of same type and a set of index values.
  5. **Structure:** There is one-to-one correspondence between each index and a data object of array.
- The operations create, retrieve and store are performed on an array.

1.5 OPERATIONS ON ARRAYS

- Array supports various operations such as traversal, insertion, deletion, sorting, searching, merging etc. Let us discuss few of them.

1. Traversal:

- Traversal means simply visiting all elements in an array.

```
a = [1, 2, 3]
  print("The array is : ", end=" ")
for i in range(0, 3):
print(a[i], end=" ")
```

Output:

The array is : 1 2 3

2. Insertion:

- Insertion means adding another element to the array. Insertion of the element can be done at particular location or element can be inserted at the end.
- If an element is inserted in the middle of an array, then the insert function is used which has 2 parameters, first is the position of insertion and second parameter is the value

```
a.insert(1, 4)
inserting the integer 4 at index 1
a = [1, 2, 3]
print("Array before insertion : ", end=" ")
```

```
for i in range(0, 3):
  print(a[i], end=" ")
print()
a.insert(1, 4)
print("Array after insertion : ", end=" ")
for i in (a):
  print(i, end=" ")
```

Output:

Array before insertion : 1 2 3

Array after insertion : 1 4 2 3

- If an element is to be inserted at the end, then the append function is used which has 1 parameter, the value to be inserted.

Appending the double value **4.4** to the array.

```
b = [2.5, 3.2, 3.3]
print("Array before insertion : ", end=" ")
for i in range(0, 3):
  print(b[i], end=" ")
print()
b.append(4.4)
print("Array after insertion : ", end=" ")
for i in (b):
  print(i, end=" ")
print()
```

Output:

Array before insertion : 2.5 3.2 3.3

Array after insertion : 2.5 3.2 3.3 4.4

3. **Delete:**

- It deletes a particular element from an array.
- The pop() method removes and returns an element at a specified index.

```
arr = [1, 2, 3, 1, 5]
print("The array is : ", end="")
for i in range(0, 5):
  print(arr[i], end=" ")
print("The popped element is : ", end="")
print(arr.pop(2))
print("The array after popping is : ", end="")
for i in range(0, 4):
  print(arr[i], end=" ")
```

Output:

The array is : 1 2 3 1 5

The popped element is : 3

The array after popping is : 1 2 1 5

- The `remove()` method removes the first occurrence of a specified value from the list

```
arr.remove(1)
print("The array after removing is : ", end="")
for i in range(0, 3):
print(arr[i], end=" ")
```

Output:

The array after removing is : 2 1 5

1.6 ONE-DIMENSIONAL ARRAY

- The simplex form of an array is one-dimensional array. The array is given a name and its elements are referred to by their subscripts or indices.

For examples,

In 'Python', we write the declaration of array of an empty array as,

```
a=[]
```

We can declare the array and initialize each of the elements in the array.

For example,

```
b = [0,0,0,0,0]
```

Will declare an array name b contains 5 values and initialize each of these elements to zero.

An array is represented in the memory by using a sequential mapping i.e. every element is at fixed distance apart. The Fig. 1.4 shows the memory representation of a one-dimensional array.

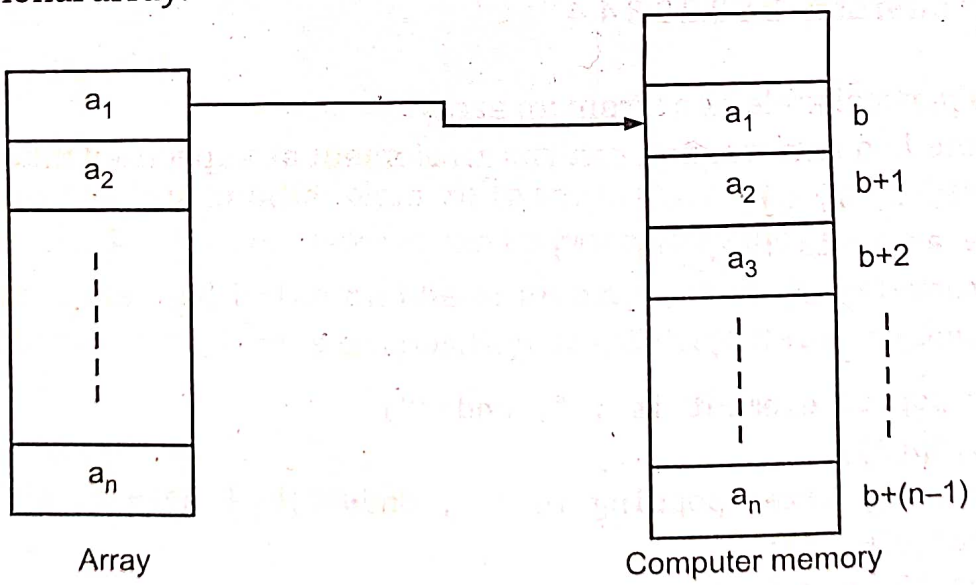


Fig. 1.4: Memory representation of one-dimensional array

Where b is the base address, which is the address of the first element. Using this base address the computer calculates the address of any element of an array by using the following formula:

Address i^{th} element = Base address + (offset of the i^{th} element from base address)

where offset is computed as:

Offset of i^{th} element = (Number of element before i^{th} element) * (size of each element)

If LB is lower bound than offset is,

Offset = (i – LB) * size

Example: If a is an array of 5 elements. Each integer is of 2 bytes and base address is 100, then to get the address of 3rd element we use formula:

address of 3rd element

= 100 + (2 * 2)

= 104

size

offset

Consider the Fig. 1.3 (i)

a [4] = 100 + (4 * size)

(i.e. 5^{th} element) = 100 + 4 × 2

= 108

1.7 TWO-DIMENSIONAL ARRAY

- A two dimensional array is a collection of elements placed in m rows and n columns. A two dimensional (2-D) array is also called Matrix. The Fig. 1.5 shows the representation of 2-D array in memory. The array is A[m][n] containing m rows and n columns.

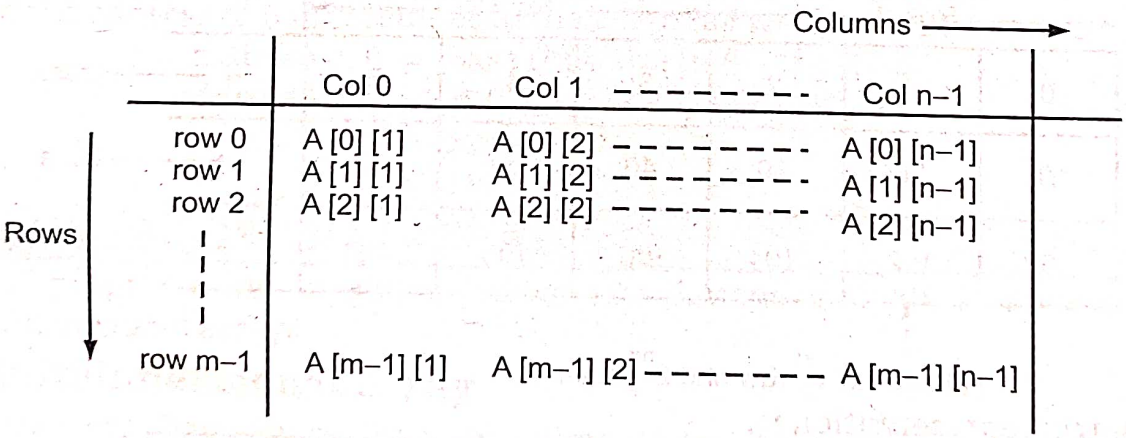


Fig. 1.5: Array A[m] [n]

- A 2-D array is a logical data structure useful in programming.
- A 2-D array differentiates between the logical and physical view of data.

1.7.1 Representation of Two-Dimensional Array in Memory

- Let A be a two dimensional array m × n. The elements in the array A will be stored either column by column i.e. column major representation or row by row i.e. row major representation.

1. Row-major Representation:

- All elements of a matrix or 2-D array get stored in memory in a linear fashion. Hence, 2-D array can be considered as 1-D array in memory. The representation of 2-D array A[2][3] in memory is shown in Fig. 1.6.

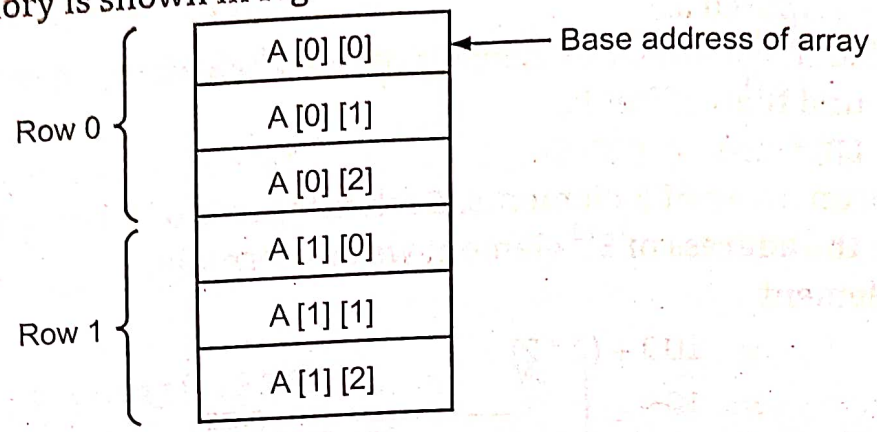


Fig. 1.6: Representation of 2-D array in row-major order

Let us calculate the address of element in 2-D array.

Formula to calculate $(i, j)^{th}$ element of 2-D array of $m \times n$ dimension is:

$$A(i, j) = base(A) + size\,[n(i-1) + (j-1)$$

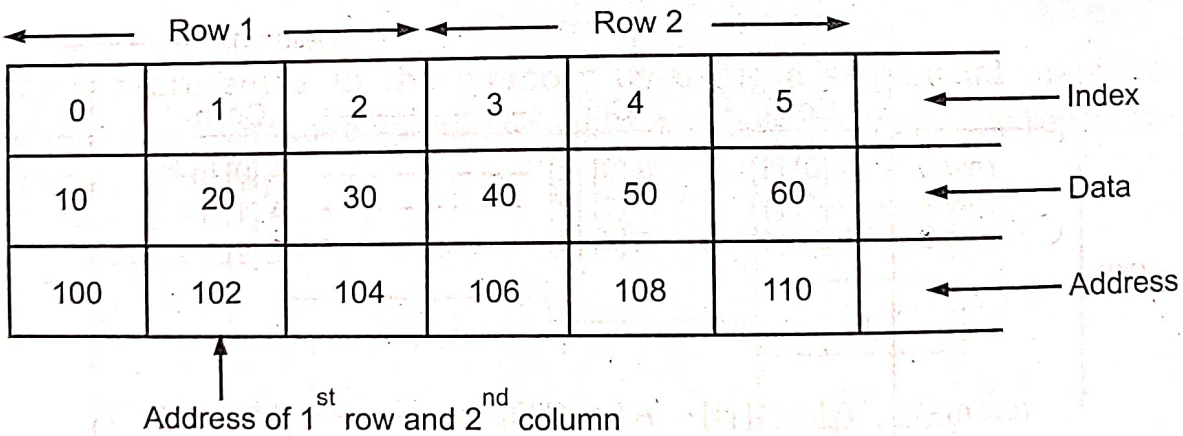
where n = Number of columns

To calculate A(1, 2) address of integer array of size 2×3 with base address 100.

$$address\ (1, 2) = 100 + 2\,[3 \times (1-1) + (2-1)]$$

(Address of 1st row and 2nd column) $= 100 + 2 \times 1$

$= 102$



The matrix representation is,

10	20	30	40	50	60
(1,1)	(1,2)	(1,3)	(2,1)	(2,2)	(2,3)

2. Column-major Representation:

- It is also possible to view a two-dimensional array as one single row of columns and map it sequentially as shown in Fig. 1.7. The Fig. 1.7 shows the 2-D matrix A[2][3] representation in memory.

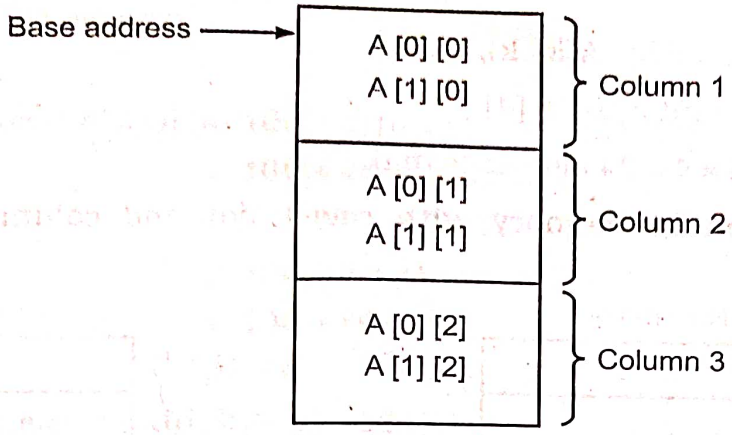
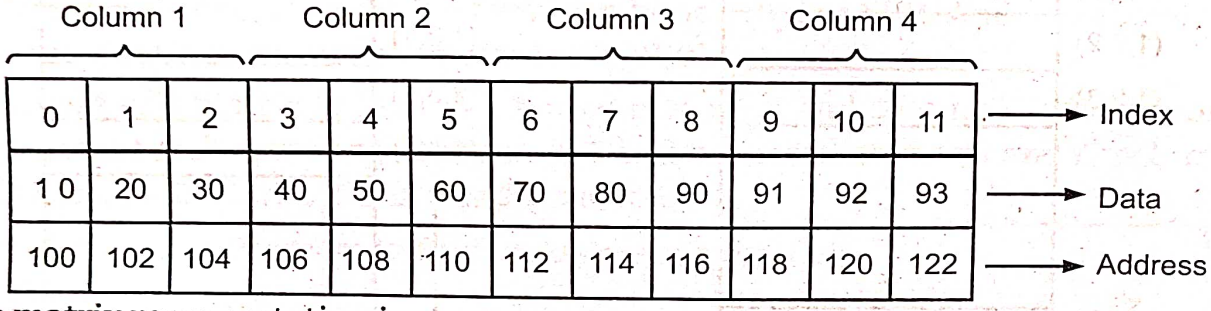
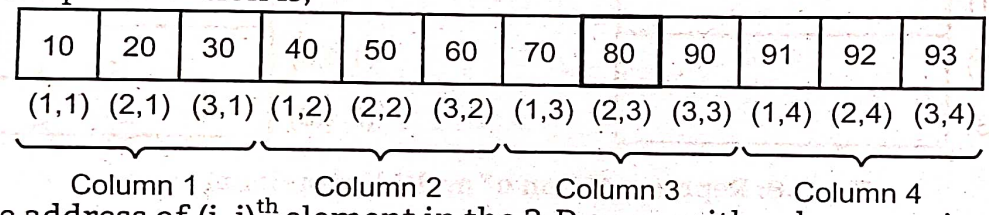


Fig. 1.7: Representation of 2-D array in column major order

For example, Let A be 2-D array of size 3 × 4. The representation of integer array is shown as follows.



The matrix representation is,



To find the address of $(i, j)^{th}$ element in the 2-D array with column major is,

$$\text{address } (i, j) = \text{Base } (A) + \text{size } [m(j-1) + (i-1)$$

$$\text{address } (2, 3) = 100 + 2\,[3\,(3-1) + (2-1)]$$

$$= 100 + 2 \times 7$$

$$= 114$$

1.8 MULTIDIMENSIONAL ARRAY

- Most of programming language allows two dimensional, three dimensional and multidimensional arrays.

1.8.1 Multidimensional Array

- The array can have more than two dimensions. An n-dimensional array A is a collection of $m_1 \times m_2 \times m_3 \ldots \times m_n$ elements. All the elements in the array are referred by subscripts, $k_1, k_2, \ldots k_n$ where,

$$1 \leq k_1 \leq m_1$$

$$1 \leq k_2 \leq m_2$$

.

.

.

$$1 \leq k_n \leq m_n$$

The array A is denoted by, $A[k_1, k_2, \ldots, k_n]$

Consider 3-D array: int A[2] [3] [4]

The array has $2 \times 3 \times 4 = 24$ elements in it.

- The representation in memory with row-major and column-major is shown in Fig. 1.8.

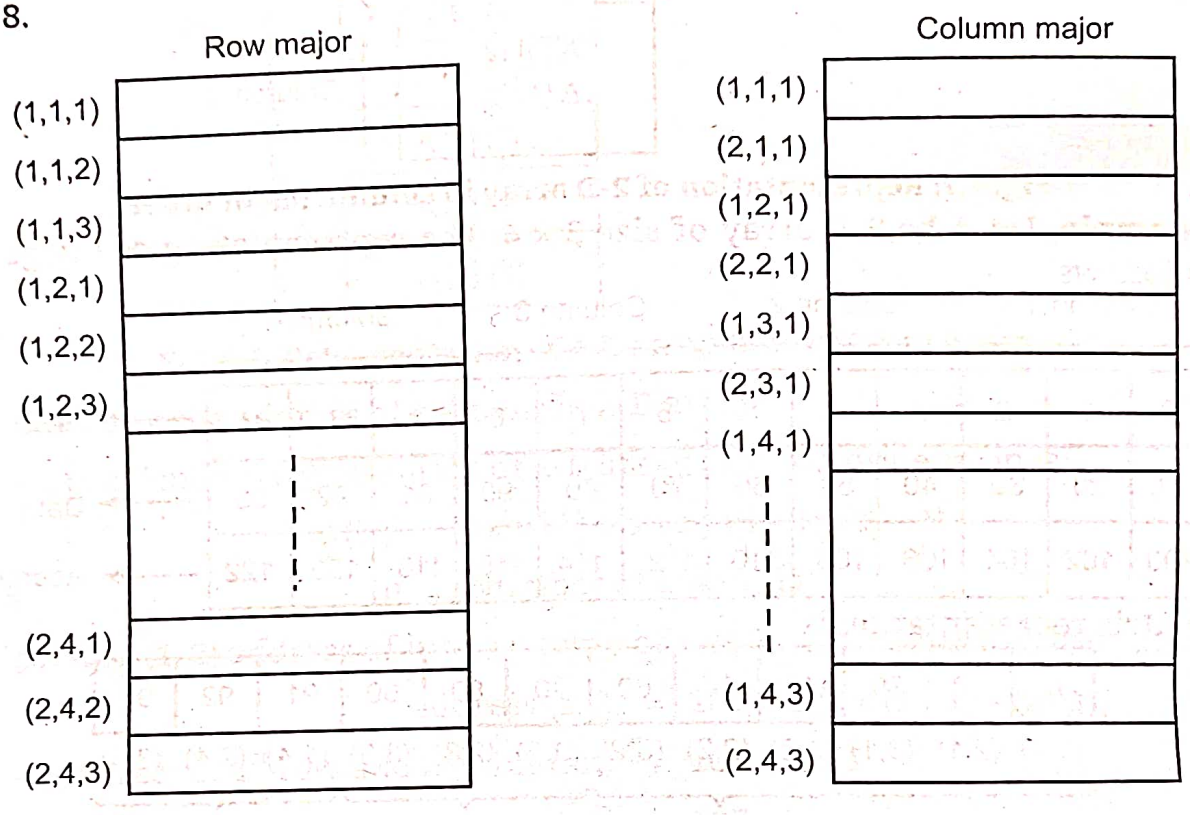


Fig. 1.8: Representation of multidimensional array

For example, consider the declaration of array A[5 ...10, –5 ...10] of integers. Assuming A is stored in a row major order with first element of A is at address 100 and each integer occupying 4 bytes. What would be the lowest byte address of the element A[6, 0]?

Solution: In row major order,

The elements are stored in the following order:

1	2	3	4	5	6	7		20	21	22	→ Index
A[5,-5]	A[5,-4]	A[5,-3]	A[5,-2]	A[5,-1]	A[5,0]	A[5,1]	- - - - - - - -	A[6,-2]	A[6,-1]	A[6,0]	→ Element location
100	104	108	112	116	120	124		176	180	184	→ Address

A[6, 0] is the 22^{nd} element of array. Each integer is of 4 bytes.

address A[i, j] = base(A) + size(n(i – 1) + (j – 1)]

Here, m = 6 and n = 16 → dimension of array.

The location of 22^{nd} element is,

2^{nd} row and 6^{th} column,

$$\therefore \quad \text{address } A[2, 6] = 100 + 4[16 \times ((2-1) + (6-1))]$$
$$= 100 + 4 \times (1 \times 16 + 5)$$
$$= 100 + 4 \times (16 + 5)$$
$$= 100 + 4 \times 21$$
$$= 100 + 84$$
$$= 184$$

$\therefore$ Address of element of index A[6, 0] = 184

- In Python, we can implement a matrix as a nested list (list inside a list). We can treat each element as a row of the matrix. For example X = [[1, 2], [4, 5], [3, 6]] would represent a 3×2 matrix. The first row can be selected as X[0]. And, the element in the first-row first column can be selected as X[0][0].
- Transpose of a matrix is the interchanging of rows and columns. It is denoted as X'. The element at i^{th} row and j^{th} column in X will be placed at j^{th} row and i^{th} column in X'. So if X is a 3×2 matrix, X' will be a 2×3 matrix.

Program 1.1: Python Program to find transpose of a matrix.

```
A =[ [1, 1, 1, 1],
     [2, 2, 2, 2],
     [3, 3, 3, 3],
     [4, 4, 4, 4]]
B =[[0,0,0,0],
     [0,0,0,0],
     [0,0,0,0],
     [0,0,0,0]]
for i in range(len(A)):
    for j in range(len(A[0])):
        B[j][i] = A[i][j]
for i in B:
    print(i)
```

Output:

The transpose of the given martix is

[1, 2, 3, 4]

[1, 2, 3, 4]

[1, 2, 3, 4]

[1, 2, 3, 4]

1.9 PROS AND CONS OF ARRAYS

1.9.1 Characteristics

- An array is a finite ordered collection of homogeneous data elements.
- Successive elements of list are stored at a fixed distance apart.
- Defined as set of pairs, index and value.
- Random access to any element is provided.
- Insertion and Deletion of elements in between requires data movement.
- Provides static allocation, which means space allocation done once before execution, can not be changed during execution.

1.9.2 Advantages

1. Arrays permit efficient random access in constant time O(1).
2. Most appropriate for storing a fixed amount of data.
3. Also most appropriate for data which will be accessed often and also accessed in an unpredictable fashion.
4. Arrays are among the most compact data structures; if we store 100 integers in an array, it takes only as much space as the 100 integers, and no more, (like linked list in which each data element has additional link field).
5. Has popularity in applications like searching, hash tables, matrix operations, sorting etc.
6. Wherever, there is a direct mapping between the elements and there position, arrays are the most suitable data structures.
7. Ordered lists such as polynomials are most efficiently handled using arrays.

1.9.3 Disadvantages

1. Arrays provide static memory management. Hence, during execution the size can neither be grown nor be shrink.
2. There is a solution to handle the problem as to declare array of some arbitrarily maximum size. This leads to two other problems.
   (a) In future if user need to exceed this limit, can not be handled.
   (b) Higher the maximum size, the more is the memory wastage. Because very often the many locations remain unused, but still allocated (reserved) for the program. This leads to poor utilization of memory space.
3. Static allocation is problem associated with implementation in many programming languages except few such as JAVA.
4. Inserting and deleting an element needs lot of data movement.
5. Inefficient for the applications, which very often need insert and delete in between.
6. Space inefficient for large objects and large quantity.

7. A drawback of simplicity of arrays is the possibility of referencing a non-existent element by using an index outside the valid range. This is known as **exceeding the array bounds**. The result is a program working with incorrect data. At the worst, the whole system can crash. In 'C', the powerful syntax is unfortunately prone to this kind of error. In some languages have built-in bounds checking, and will refuse to index an array outside of its permitted range.

1.9.4 Applications

- Although useful in their own right, arrays also form the basis for several more complex data structures, such as heaps, hash tables and can be used to represent strings, stacks and queues.
- All these applications benefit from the compactness and direct access benefits of arrays.
- Matrix handling and matrix operations.
- Indexing and sorting keys.
- In some applications where the data is the same or is missing for most values of the indices, or for large ranges of indices, space is saved by not storing an array at all, also called as Sparse Matrix representation. This has an associative array with integer keys. There are many specialized data structures specifically for applications include, address translation table and routing tables.

1.10 SPARSE MATRICES

- Sparse matrices are those matrices that have the majority of their elements equal to zero. In other words, the sparse matrix can be defined as the matrix that has a greater number of zero elements than the non-zero elements.

Advantages of using the sparse matrix:

- **Storage:** We know that a sparse matrix contains lesser non-zero elements than zero elements, so less memory can be used to store elements. It evaluates only the non-zero elements.
- **Computing time:** In the case of searching in sparse matrix, we need to traverse only the non-zero elements rather than traversing all the sparse matrix elements. It saves computing time by logically designing a data structure traversing non-zero elements.

1.10.1 Array representation of the sparse matrix

- Representing a sparse matrix by a 2D array leads to the wastage of lots of memory. This is because zeroes in the matrix are of no use, so storing zeroes with non-zero elements is wastage of memory. To avoid such wastage, we can store only non-zero elements. If we store only non-zero elements, it reduces the traversal time and the storage space.
- In 2D array representation of sparse matrix, there are three fields used that are named as:

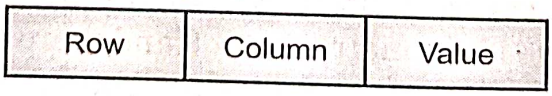


Fig. 1.9: Representation of Sparse Matrix

- **Row:** It is the index of a row where a non-zero element is located in the matrix.
- **Column:** It is the index of the column where a non-zero element is located in the matrix.
- **Value:** It is the value of the non-zero element that is located at the index (row, column).

- Consider the sparse matrix,

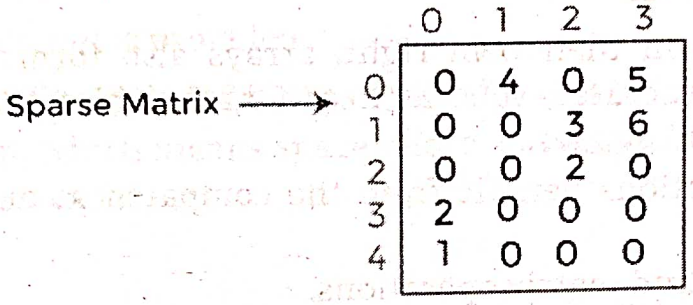


Fig. 1.10: Representation of Sparse Matrix

- In the Fig. 1.10, we can observe a 5 × 4 sparse matrix containing 7 non-zero elements and 13 zero elements. The above matrix occupies 5 × 4 = 20 memory space. Increasing the size of matrix will increase the wastage space.
- The tabular representation of the above matrix is given in table:

Table 1.1: Tabular representation of Sparse Matrix

Row	Column	Value
0	1	4
0	3	5
1	2	3
1	3	6
2	2	2
3	0	2
4	0	1
5	4	7

- In the table, first column represents the rows, the second column represents the columns, and the third column represents the non-zero value. The first row of the table represents the triplets. The first triplet represents that the value 4 is stored at 0^{th} row and 1^{st} column. Similarly, the second triplet represents that the value 5 is stored at the 0^{th} row and 3^{rd} column. In a similar manner, all triplets represent the stored location of the non-zero elements in the matrix.
- The size of the table depends upon the total number of non-zero elements in the given sparse matrix. Table 1.1 occupies 8 × 3 = 24 memory space which is more than the space occupied by the sparse matrix. So, what's the benefit of using the sparse matrix? Consider the case if the matrix is 8 × 8 and there are only 8 non-zero elements in the matrix, then the space occupied by the sparse matrix would be 8 × 8 = 64, whereas the space occupied by the table represented using triplets would be 8 × 3 = 24.

Program 1.2: Python program to convert a matrix to sparse matrix.

```
# function display a matrix
def displayMatrix(matrix):

    for row in matrix:
        for element in row:
            print(element, end =" ")
        print()

# function to convert the matrix into a sparse matrix
def convertToSparseMatrix(matrix):

    # creating an empty sparse  matrix list
    sparseMatrix =[]

    # searching values greater  than zero
    for i in range(len(matrix)):
        for j in range(len(matrix[0])):
            if matrix[i][j] != 0 :

                # creating a temporary sublist
                temp = []

                # appending row value, column value and element into the
                sublist
                temp.append(i)
                temp.append(j)
                temp.append(matrix[i][j])

                # appending the sublist into the sparse matrix list
                sparseMatrix.append(temp)

    # displaying the sparse matrix
    print("\nSparse Matrix: ")
    displayMatrix(sparseMatrix)

# Driver's code
# initializing a normal matrix
normalMatrix =[[1, 0, 0, 0],
               [0, 2, 0, 0],
               [0, 0, 3, 0],
               [0, 0, 0, 4],
               [5, 0, 0, 0]]

# displaying the matrix
displayMatrix(normalMatrix)

# converting the matrix to sparse  and displayMatrix
convertToSparseMatrix(normalMatrix)
```

Output:

```
1 0 0 0
0 2 0 0
0 0 3 0
0 0 0 4
5 0 0 0
```

Sparse Matrix:

```
0 0 1
1 1 2
2 2 3
3 3 4
4 0 5
```

1.11 STRING MANIPULATION USING ARRAYS

- String arrays are a data structure used to store multiple strings in memory. This allows us to easily manipulate, access, and modify the individual elements of the array.

For example,

```
my_city = "New York"
print(type(my_city))
#Single quotes have exactly the same use as double quotes
my_city = 'New York'
print(type(my_city))

#Setting the variable type explicitly
my_city = str("New York")
print(type(my_city))

word1 = 'New '
word2 = 'York'
print(word1 + word2)
print(my_city[1])
```

Output:

```
<class 'str'>
<class 'str'>
<class 'str'>
New York
e
```

For example,

```
names = [
  "Asha",
  "Pooja",
  "Ankita",
  "Anvi",
  "Avni"
]
print(names[4])
```

Output:

Avni

For example,

```
bookshelf = []
bookshelf.append("Mechanical Engineer")
bookshelf.append("Computer Engineer")
bookshelf.append("Electronics Engineer")
print(bookshelf[0])
print(bookshelf[1])
```

Output:

Mechanical Engineer
Computer Engineer

For example,

```
#Remove specified string from an array
bookshelf.remove('Computer Engineer')
```

Output:

Mechanical Engineer
Electronics Engineer

For example,

```
# Pop string from an array
bookshelf.pop(1)
```

Output:

Mechanical Engineer

Summary

- An array is a **finite ordered collection of homogeneous data elements** which provides random access to the elements.
- Array supports various operations such as traversal, insertion, deletion, sorting, searching, merging.
- A one-dimensional array is of fixed size sequence of elements of the same type.
- Array elements can be access using index.
- A two-dimensional array is a representation of a table with rows and columns.

- A 2-D array is representing in memory with row-major or column-major order.
- The sparse matrix can be defined as the matrix that has a greater number of zero elements than the non-zero elements.
- String Manipulation is a class of problems where a user is asked to process a given string and use/change its data.

Check Your Understanding

I. Multiple Choice Questions:

1. Which of the following expressions access the $(i, j)^{th}$ of a $m \times n$ matrix stored in column major form?
   (a) $n \times (i-1) + j$ (b) $m \times (j-1) + i$
   (c) $m \times (n-j) + j$ (d) $n \times (m-i) + j$
2. The smallest element of an array's index is called its _____
   (a) Lower bound (b) Upper bound
   (c) Range (d) Extraction
3. While passing an array as an actual argument, the function call must have
   (a) the array name with empty brackets.
   (b) the array name with its size.
   (c) the array name alone.
   (d) none of the above.
4. In a two-dimensional array with four rows, which is the row with the highest addresses in memory?
   (a) first (b) second
   (c) third (d) fourth

Answers

1. (b)	2. (a)	3. (c)	4. (a)

Practice Questions

Q.I Answer the following questions in short.

1. Define an array?
2. How are arrays declared?
3. How are two-dimensional array elements stored in the memory?

Q.II Answer the following questions.

1. What are the differences between linear and non-linear data structures? Explain with appropriate examples.
2. What is meant by linear data structure? Explain any one of the linear data structure in detail.
3. Derive a formula for address calculation of n-dimensional array. Assume a row major implementation of storage for integer array offset in bytes.
4. Why is there a need for data structures 'array'?
5. How is an element in an array different from an element in a record?