CHAPTER 1
BASIC CONCEPTS OF DATA STRUCTURES

1.1 Basic Concepts

➢ Data v/s Information

Data	Information
- Data means known facts, that can be recorded and have implicit meanings.	- Information means processed or organized data.

- Examples:
  - Student no: *7001*
  - Student name: *Riya*
  - City name: *Ahmedabad*
  - Account Number: *A01*
  - Balance: *5000*
	- Examples:
	  - Percentage: *82.20%*
	  (derived from processing marks)
	  - Run-rate in a cricket match: *6.0 runs/over*
	  (derived from total runs and over)

- Data are raw materials used to derive information. - For example, marks of subjects.	- Information is a product derived from data. - For example, percentage.
- Data convey something less, comparatively. - It is not convenient to represent result as marks of 5 subjects in form of – 80, 78, 85, 86, and 82.	- Information conveys something more, comparatively. - It is convenient to represent result as a percentage in form of 82.20%.
- Data is comparatively less useful.	- Information is comparatively more useful.

➢ Data Structure

- "The way of handling/managing data items, that considers not only the items stored, but also their relationship to each other."
- For example, an Array provides a convenient way to manage collection of data of the same type. So, an Array is considered as a data structure.
- As another example, a Structure provides a convenient way to manage collection of logically related data. So, a Structure is also considered as one kind of data structure.
- Other examples are – String, Stack, Queue, Linked-List, Tree, and Graph.

➢ Data Structure Management

- "The process of managing data structures is referred as data structure management."

1.2 Types of Data Structures

Data structures can be broadly divided into two categories – Primitive data structures, and Non-primitive data structures.

Non-primitive data structures can be further divided into two categories as Linear data structures and Non-linear data structures.

The following figure depicts the classification of data structures.

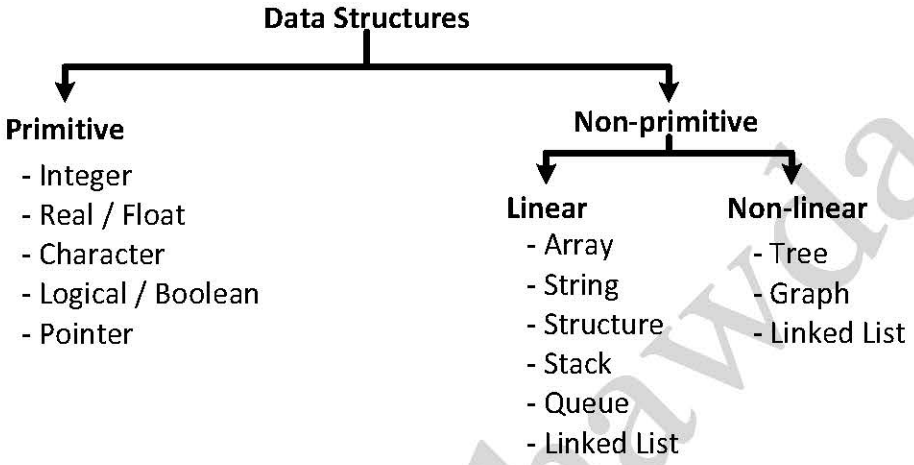


Figure 1: Types of Data Structures

❖ Primitive v/s Non-primitive Data Structures:

	Primitive Data Structure	Non-primitive Data Structure
1.	Basic or Fundamental data structure.	Not a basic data structure.
2.	Can't be divided into further data structures.	Can be divided into further data structures.
3.	Does not depend upon other data structures.	Depends upon other (primitive) data structures.
4.	Can be operated by machine level instructions directly.	Can't be operated by machine level instructions directly.
5.	Examples: - Integer - Real / Float - Character - Logical / Boolean - Pointer	Examples: - Array - String - Structure - Stack - Queue - Linked List - Tree - Graph

❖ Linear v/s Non-linear Data Structures:

	Linear Data Structure	Non-linear Data Structure
1.	Data structures whose elements form a linear sequence.	Data structures whose elements do not form a linear sequence.
2.	Elements can be accessed sequentially (linearly).	Elements cannot be accessed sequentially (linearly).
3.	Elements occupy contiguous memory space.	Elements do not occupy contiguous memory space. They will be scattered in memory.
4.	Examples: - Array - String - Structure - Stack - Queue - Linked List*	Examples: - Tree - Graph - Linked List*

❖ Primitive Data Structures – Examples:

1. **Integer:**
   - Represents whole numerical values.
   - Example: -5, 0, 10
2. **Float / Real:**
   - Represents floating point (real) numerical values.
   - Examples: -10.25, 0.7, 5.43
3. **Character:**
   - Represents characters and strings.
   - Example: 'A', 'a', "bbit", "VVNagar"
4. **Boolean:**
   - Represents Boolean values.
   - Example: true, false.
5. **Pointer:**
   - Represents addresses of other data or variables.

❖ Non-primitive Data Structures – Examples:

1. **Array:**
   - "Sequential collection of data-items of the same type that share a common name."
   - Example: int a [5] → represents array of 5 integer elements.
2. **String:**
   - "Array of characters terminated by Null character."
   - Example: char name[20]
3. **Structure:**
   - "Collection of logically related data items, grouped together under a single name."
   - Example:

```
struct student
{
    int no;
    char name[20];
};
```

4. **Stack:**
   - "A data structure in which items can be inserted and removed from the one end only."
5. **Queue:**
   - "A data structure in which items can be inserted at one end and removed from the other end."
6. **Linked List:**
   - "Collection of nodes in which each node points to its next node in a linear sequence."
7. **Tree:**
   - **"**A data structure that represents hierarchical relationship among individual data-items."
8. **Graph:**
   - "A collection of a set of nodes (vertices) and a set of edges (arcs)."

1.3 Algorithm:

❖ Def:

- "A finite set of **instructions** which accomplish a particular **task**."

❖ Structure of an Algorithm:

- **Name:**
  - Represents the name of an algorithm with input arguments.
- **Comment:**
  - Specifies the task/operation performed by the algorithm.
- **Variables:**
  - Describes variables used in algorithm.
- **Steps:**
  - Specifies step-no, comment for an individual step and instructions to perform that step.

1.4 Algorithmic Notations:

- **Reading an Input:**
  - READ variable-name
  - Example: READ N
- **Displaying an Output:**
  - WRITE (MESSAGE) variable-name
  - Example: WRITE ('Hello World')
- **Assignment Operation:**
  - Use left Arrow.
  - Example: N ← 5, SUM ← SUM + TERM
- **Exit from Algorithm:**
  - EXIT.
- **Arithmetic Operators:** +, –, *, /
- **Relational Operators:** <, ≤, >, ≥, =, <> (≠)
- **Logical Operators:** AND, OR, NOT

- **Control Structures:**

  1. **IF:**

```
IF    condition    THEN
      // code-block
END IF
```

  2. **IF … ELSE:**

```
IF    condition    THEN
      // True code-block
ELSE
      // False code-block
END IF
```

  3. **IF … ELSE IF Ladder:**

```
IF    condition-1  THEN
      // Code-block-I
ELSE IF condition-2     THEN
      // Code-block-II
      :
      :
ELSE
      // Default code-block
END IF
```

  4. **SWITCH … CASE:**

```
SELECT    CASE ( EXPRESSION )
      CASE V1:
            // Code-block-1
      CASE V2:
            // Code-block-2
      DEFAULT:
            // Default code-block
END SELECT
```

5. **WHILE Loop:**

```
WHILE      condition      DO
        // Code-block
END WHILE
```

6. **FOR Loop:**

```
FOR Index := Start TO End BY Increment
DO
        // Code-block
END FOR
```

1.5 Analysis Terms

1. **Time Complexity:**

- "Total amount of **time** required to execute an algorithm (or program)."

2. **Space Complexity:**

- "Total amount of **space** required to execute an algorithm (or program)."

3. **Asymptotic Notation:**

- "A way to describe efficiency and performance of an algorithm."
- It is used to determine which algorithm is better and more efficient.

4. **Big 'O' (Big Oh)Notation:**

- "A way to describe upper bound of an algorithm's performance."
- Performance can be measured in terms of time complexity or space complexity.

5. **Best case Time Complexity:**

- "Amount of **time** required to execute an algorithm in best case."
- It provides lower bound on the time (minimum time) required to execute an algorithm.

6. **Average case Time Complexity:**

- "Amount of **time** required to execute an algorithm in average case."
- It provides average time required to execute an algorithm.

7. **Worst case Time Complexity:**

- "Amount of **time** required to execute an algorithm in worst case."
- It provides upper bound on the time (maximum time) required to execute an algorithm.

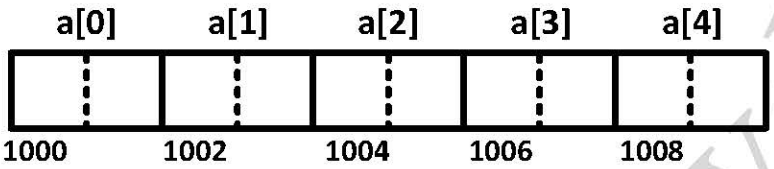
1.6 Row Major and Column Major Arrays

Array is a sequential collection of data-items of the same type that share a common name.

Array can be one dimensional, two dimensional or multi dimensional.

❖ One Dimensional (1-D) Array:

- **Syntax:** data-type array-name [Size];
- **Example:** int a [5];
- **Representation** in memory:



- **Equation** to calculate address of an element:

  Address of a[i] = Starting Address + [i * Required Bytes]

- **Example:**

  Address of a[2] = 1000 + [2 * 2] = 1000 + 4 = **1004**
  (In our example, starting address = 1000; i = 2; and required bytes = 2 as each element of int requires 2 bytes storage space.)

❖ Two Dimensional (2-D) Array:

- **Syntax:** data-type array-name [ROW][COL];

  Where ROW and COL represents no. of rows and no. of columns respectively.

- **Example:** int a [2][3];

  This array contains 6 elements. It has two rows; each row having three columns. This arrangement is depicted in following figure.

	0th Column	1st Column	2nd Column
0th Row	**a[0][0]**	**a[0][1]**	**a[0][2]**
1st Row	**a[1][0]**	**a[1][1]**	**a[1][2]**

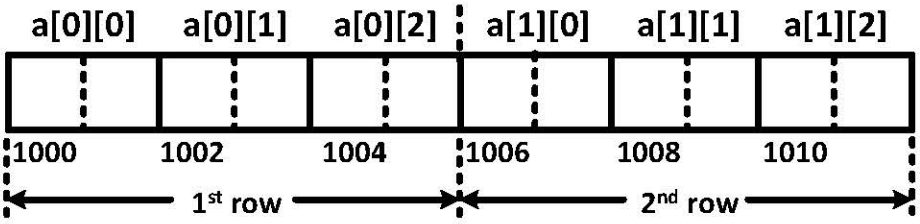
- **Representation** in memory:

  There are two different ways to store two-dimensional arrays in memory:

  i) Row Major Order, and
  ii) Column Major Order.

1. **Row Major Order / Row Major Array:**

- **Storage:** Elements are stored sequentially by **Rows**.
- **Representation** in memory:



- **Equation** to calculate address of an element:

  Address of a[i][j] = Starting Address + [(i * COL + j) * Required Bytes]

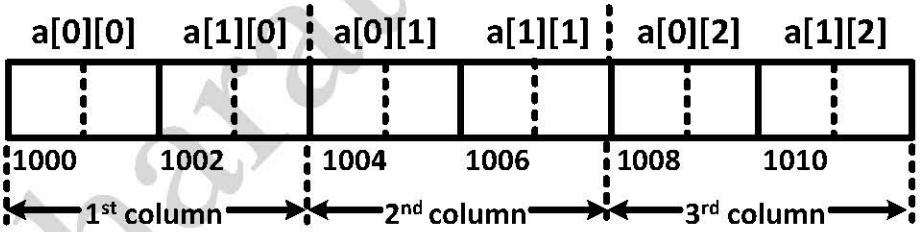
- **Example:**

  Address of a[1][0] = 1000 + [(1 * 3 + 0) * 2] = 1000 + [3 * 2] = **1006**
  (In our example, starting address = 1000; i = 1; j = 0; and required bytes = 2 as each element of int requires 2 bytes storage space.)

- **Application:** Programming languages like C, C++, Java use Row Major Arrays.

2. **Column Major Order / Column Major Array:**

- **Storage:** Elements are stored sequentially by **Columns**.
- **Representation** in memory:



- **Equation** to calculate address of an element:

  Address of a[i][j] = Starting Address + [(j * ROW + i) * Required Bytes]

- **Example:**

  Address of a[1][0] = 1000 + [(0 * 2 + 1) * 2] = 1000 + [1 * 2] = **1002**
  (In our example, starting address = 1000; i = 1; j = 0; and required bytes = 2 as each element of int requires 2 bytes storage space.)

- **Application:** Programming languages like MATLAB use Column Major Arrays.

1.7 Overview of various Array Operations

Various operations that can be performed on array are given below.

- **Traversal:** Visit individual elements of an Array and process them.
- **Insertion:** Add / Insert a new element into an Array.
- **Deletion:** Remove / Delete an element from an Array.
- **Searching:** Find the location of a particular element from an Array.
- **Sorting:** Sort the elements of an Array in specific order.
- **Merging:** Combine two arrays into single Array.
- **Reversing:** Reverse the elements of an Array.

1.8 Searching an Element into Array

- "Searching is a process of finding out location of given data element."
- Main techniques for searching an element into array:
  - ❖ **Linear / Sequential Search:**
  - ❖ **Binary Search:**

    These techniques are described below.

1.8.1 Linear / Sequential Search

- **Concept:**
  - Sequentially compare the given element to be searched with each element in an array one by one from start to end.
  - This process continues until the element is found or an array ends.
  - If the element is found, the position of the element is returned.
  - If the element is not found, the error element, such as negative index, is returned.
- **Example:**
  - (Explain working with example as covered in class.)
- **Advantage:**
  - Easy to understand and implement.
  - Applicable to array as well as linked list.
  - Data need not to be sorted; can be applied to unsorted data also.
- **Disadvantage:**
  - Not efficient; Search time is more.

- **Function:**

```
int    search_seq  ( int  arr [ ], int  x )
{
        int  i;
        // Search element sequentially from start to end...
        for  ( i = 0;  i < MAX ; i  + +)
        {
                if ( arr [ i ] == x )
                {
                        return  i;     // element found...
                }
        }
        return  -1 ;  // element not found...
}
```

- **Algorithm:**
  - (Prepare on your own.)

1.8.2 Binary Search

- **Concept:**
  - Binary search can be applied only on the sorted data.
  - Compute the **middle** index, and get the element stored at middle position.
  - Compare element to be searched with middle element.
  - If element is found, return **middle** position as an index.
  - If element is not found, divide the list in upper half and lower half.
  - If element to be searched is less than the middle element, continue search process in upper half.
  - If element to be searched is greater than the middle element, continue search process in lower half.
  - This process continues until the element is found, or an array has no element to compare.

- **Example:**
  - (Explain working with example as covered in class.)
- **Advantage:**
  - Very efficient.
- **Disadvantage:**
  - Data must be sorted; cannot be applied to unsorted data.
  - Applicable to array only; cannot be applied on linked list.
- **Function:**

```
int    search_bin  ( int  arr [ ], int   x )
{
        int  mid, low, high;
        // Initialization…
        low = 0;
        high = MAX – 1;

        // Search element…
        while ( low <= high )
        {
                mid = ( low + high ) / 2;
                if ( arr [ mid ] < x)
                        low = mid + 1;
                else if ( arr [ mid ] > x )
                        high = mid - 1;
                else
                        return mid;  // element found…
        }
        return  -1 ;  // element not found…
}
```

- **Algorithm:**

  (Prepare on your own.)

● ● ●