

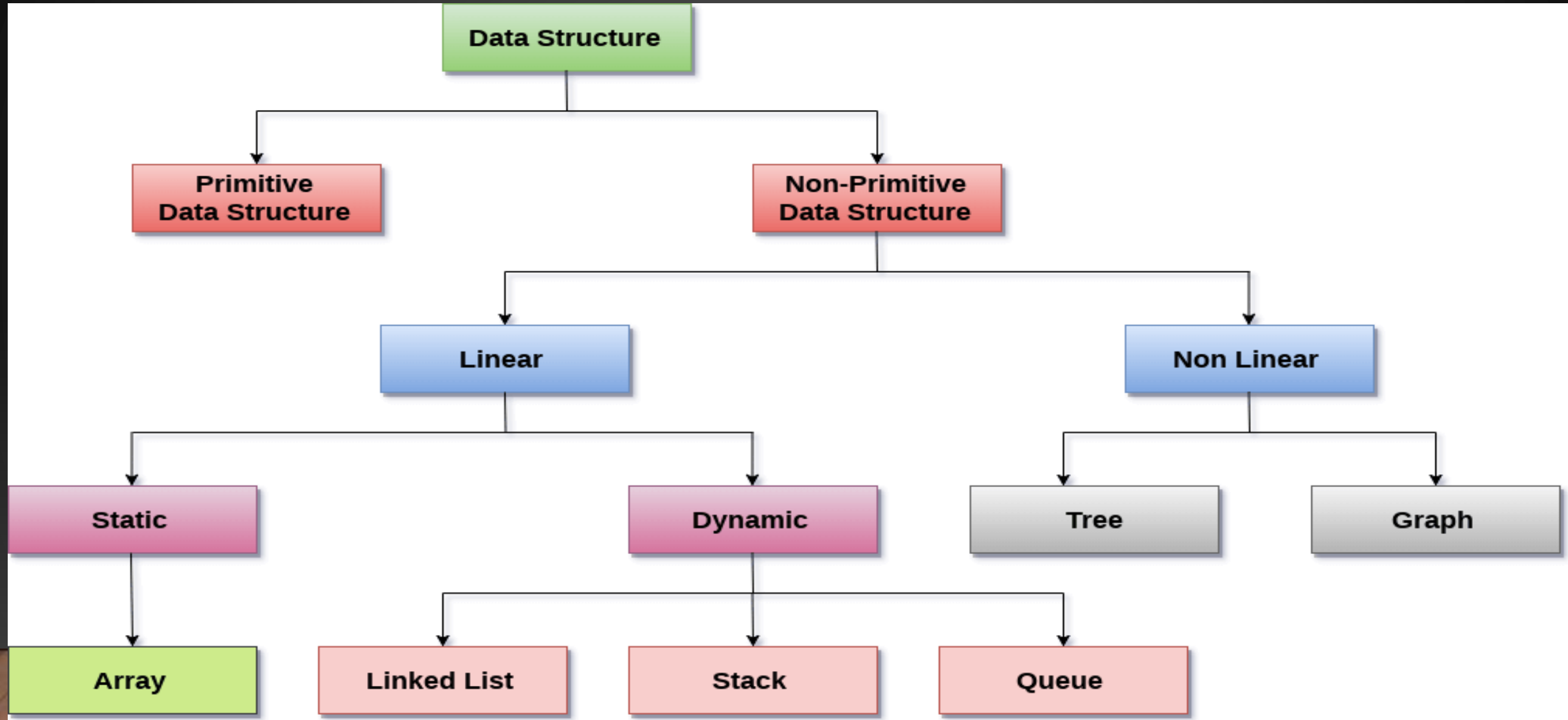
DATA STRUCTURE

INTRODUCTION TO DATA STRUCTURE AND ITS
CHARACTERISTICS ARRAY

UNIT - 1

ELEMENTARY DATA ORGANIZATION

Data Structure Classification :



Linear Data Structures:

A data structure is called linear if all of its elements are arranged in the linear order. In linear data structures, the elements are stored in non-hierarchical way where each element has the successors and predecessors except the first and last element.

Types of Linear Data Structures are given below:

Arrays:

An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double.

The elements of array share the same variable name but each one carries a different index number known as subscript. The array can be one dimensional, two dimensional or multidimensional.

The individual elements of the array age are:

age[0], age[1], age[2], age[3],..... age[98], age[99].

Linked List:

Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node.

Stack:

Stack is a linear list in which insertion and deletions are allowed only at one end, called top.

A stack is an abstract data type (ADT), can be implemented in most of the programming languages. It is named as stack because it behaves like a real-world stack, for example: - piles of plates or deck of cards etc.

Queue:

Queue is a linear list in which elements can be inserted only at one end called rear and deleted only at the other end called front.

It is an abstract data structure, similar to stack. Queue is opened at both end therefore it follows First-In-First-Out (FIFO) methodology for storing the data items.

Non Linear Data Structures:

This data structure does not form a sequence i.e. each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in sequential structure.

Types of Non Linear Data Structures are given below:

Trees:

Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottommost nodes in the hierarchy are called leaf node while the topmost node is called root node. Each node contains pointers to point adjacent nodes.

Tree data structure is based on the parent-child relationship among the nodes. Each node in the tree can have more than one children except the leaf nodes whereas each node can have atmost one parent except the root node. Trees can be classified into many categories which will be discussed later in this tutorial.

Graphs:

Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from tree in the sense that a graph can have cycle while the tree can not have the one.

Operations on data structure

1) Traversing: Every data structure contains the set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting.

Example: If we need to calculate the average of the marks obtained by a student in 6 different subject, we need to traverse the complete array of marks and calculate the total sum, then we will divide that sum by the number of subjects i.e. 6, in order to find the average.

2) Insertion: Insertion can be defined as the process of adding the elements to the data structure at any location.

If the size of data structure is n then we can only insert $n-1$ data elements into it.

3) Deletion: The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location.

If we try to delete an element from an empty data structure then underflow occurs.

- 4) Searching: The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching, Linear Search and Binary Search.
- 5) Sorting: The process of arranging the data structure in a specific order is known as Sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.
- 6) Merging: When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size $(M+N)$, then this process is called merging.

ARRAYS

- An Array is defined as, an ordered set of similar data items. All the data items of an array are stored in consecutive memory locations.
- The data items of an array are of same type and each data items can be accessed using the same name but different index value.
- An array is a set of pairs, , such that each index has a value associated with it. It can be called as corresponding or a mapping

Ex: < 0 , 25 > list[0]=25
 < 1 , 15 > list[1]=15
 < 2 , 20 > list[2]=20
 < 3 , 17 > list[3]=17
 < 4 , 35 > list[4]=35

Here, list is the name of array. By using, list [0] to list [4] the data items in list can be accessed.

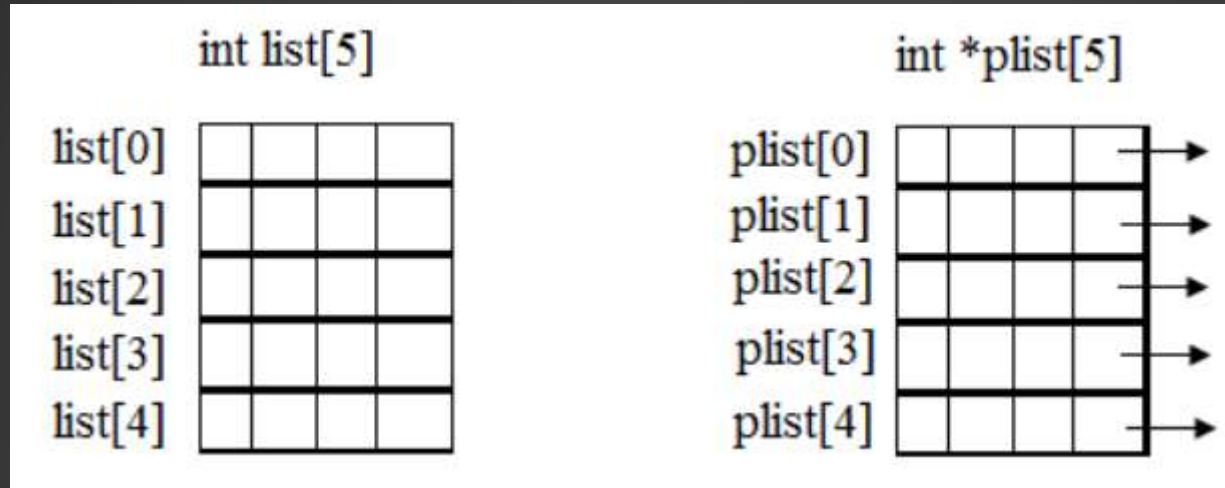
Array in C Declaration:

A one dimensional array in C is declared by adding brackets to the name of a variable.

Ex:

```
int list[5], *plist[5];
```

- The array `list[5]`, defines 5 integers and in C array start at index 0, so `list[0]`, `list[1]`, `list[2]`, `list[3]`, `list[4]` are the names of five array elements which contains an integer value.
- The array `*plist[5]`, defines an array of 5 pointers to integers. Where, `plist[0]`, `plist[1]`, `plist[2]`, `plist[3]`, `plist[4]` are the five array elements which contains a pointer to an integer.



Implementation:

- When the compiler encounters an array declaration, `list[5]`, it allocates five consecutive memory locations. Each memory is enough large to hold a single integer.
- The address of first element of an array is called Base Address. Ex: For `list[5]` the address of `list[0]` is called the base address.

- If the memory address of `list[i]` need to compute by the compiler, then the size of the `int` would get by `sizeof (int)`, then memory address of `list[i]` is as follows:

$$\text{list}[i] = \alpha + i * \text{sizeof}(\text{int})$$

Where, α is base address.

<code>list[3]</code>	$= \alpha + 3 * \text{sizeof}(\text{int})$	<code>list[0]</code>	<code>list[1]</code>	<code>list[2]</code>	<code>list[3]</code>	<code>list[4]</code>
	$= 2000 + 3 * 4$					
<code>list[3]</code>	$= 2012$					
		2000	2004	2008	2012	2016

Example: Array Program

```
#define MAX_SIZE 100
float sum(float [], int);
float input[MAX_SIZE], answer;
void main(void)
{
    int i;
    for( i=0; i<MAX_SIZE; i++)
        input[i]= i;
    answer = sum(input, MAX_SIZE);
    printf("\n The sum is: %f \n",answer);
}

float sum(float list[], int n)
{
    int i;
    float tempsum = 0;
    for(i=0; i<n; i++)
        tempsum = tempsum + list[i];
    return tempsum;
}
```

When **sum** is invoked, **input=&input[0]** is copied into a temporary location and associated with the formal parameter *list*

A function that prints out both the address of the *i*th element of the array and the value found at that address can be written as shown in below program.

```
void printl (int *ptr, int rows)
{
    int i;
    printf(" Address contents \n");
    for(i=0; i<rows; i++)
        printf("% 8u %5d \n", ptr+i, *(ptr+i));
    printf("\n");
}
```

Output:

Address	Content
12244868	0
12344872	1
12344876	2
12344880	3
12344884	4

Sparse Matrix

A matrix can be defined as a two-dimensional array having 'm' columns and 'n' rows representing $m \times n$ matrix. 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.

Why do we need to use a sparse matrix instead of a simple matrix?

We can also use the simple matrix to store the elements in the memory; then why do we need to use the sparse matrix. The following are the advantages of using a sparse matrix:

- **Storage:** As we know, a sparse matrix that contains lesser non-zero elements than zero 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.

Representing a sparse matrix by a 2D array leads to the wastage of lots of memory. The zeroes in the matrix are of no use to store zeroes with non-zero elements. 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.

Sparse Matrix Representation

The non-zero elements can be stored with triples, i.e., rows, columns, and value. The sparse matrix can be represented in the following ways:

Array representation

Linked list representation

Array Representation

The 2d array can be used to represent a sparse matrix in which there are three rows named as:

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

Let's understand the sparse matrix using array representation through an example.

Sparse matrix →

	0	1	2	3
0	0	4	0	5
1	0	0	3	6
2	0	0	2	0
3	2	3	0	0
4	0	0	0	0

Table Structure

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

In C programming, sparse arrays and triangular matrices can be efficiently represented using vectors to store only the non-zero elements. Here's how you can represent upper and lower triangular matrices using vector representations in C.

1. Upper Triangular Matrix

An upper triangular matrix is a matrix where all the elements below the main diagonal are zero. This means we only need to store the elements on or above the diagonal.

If we have an $n \times n$ upper triangular matrix, the number of non-zero elements is $n(n + 1) / 2$. We can store these non-zero elements in a 1D array (vector).

Example in C:

```

#include <stdio.h>
#include <stdlib.h>
// Function to map 2D index (i, j) to 1D index in
the vector
int upper_triangular_index(int i, int j, int n) {
    if (i <= j) {
        return (i * n - i * (i - 1) / 2 + (j - i));
    }
    return -1; // Invalid access for lower part of the
matrix
}
void print_upper_matrix(int *vector, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i <= j)
                printf("%d ",
vector[upper_triangular_index(i, j, n)]);
            else
                printf("0 ");
        }
        printf("\n");
    }
}

```

```

int main() {
    int n = 4; // Size of the matrix (n x n)
    int size = n * (n + 1) / 2; // Size of the vector
    int *vector = (int *)malloc(size * sizeof(int));

    // Filling vector with sample data (for example, value
= row + column)
    for (int i = 0; i < size; i++) {
        vector[i] = i + 1; // Assign values to the vector
    }

    printf("Upper Triangular Matrix:\n");
    print_upper_matrix(vector, n);

    free(vector);
    return 0;
}

```

Explanation:

- We store only the upper triangular part in a vector.
- The function `upper_triangular_index` maps the 2D matrix indices (i, j) to a 1D index in the vector.
- The matrix is printed by checking if an element is part of the upper triangle.

Output:

Upper Triangular Matrix:

```
1 2 3 4
0 5 6 7
0 0 8 9
0 0 0 10
```

2. Lower Triangular Matrix

A lower triangular matrix is a matrix where all the elements above the main diagonal are zero. We only need to store the elements on or below the diagonal.

If we have an $n \times n$ lower triangular matrix, the number of non-zero elements is also $n(n + 1) / 2$. We can store these non-zero elements in a 1D array (vector).

Example in C:

```
#include <stdio.h>
#include <stdlib.h>

// Function to map 2D index (i, j) to 1D index in the vector
int lower_triangular_index(int i, int j, int n) {
    if (i >= j) {
        return (i * (i + 1) / 2 + j);
    }
    return -1; // Invalid access for upper part of the matrix
}

void print_lower_matrix(int *vector, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i >= j)
                printf("%d ", vector[lower_triangular_index(i, j, n)]);
            else
                printf("0 ");
        }
        printf("\n");
    }
}
```

```
int main() {
    int n = 4; // Size of the matrix (n x n)
    int size = n * (n + 1) / 2; // Size of the vector
    int *vector = (int *)malloc(size * sizeof(int));

    // Filling vector with sample data (for example,
    // value = row + column)
    for (int i = 0; i < size; i++) {
        vector[i] = i + 1; // Assign values to the vector
    }

    printf("Lower Triangular Matrix:\n");
    print_lower_matrix(vector, n);

    free(vector);
    return 0;
}
```

Explanation:

- We store only the lower triangular part in a vector.
- The function `lower_triangular_index` maps the 2D matrix indices (i, j) to a 1D index in the vector.
- The matrix is printed by checking if an element is part of the lower triangle.

Output:

Lower Triangular Matrix:

1 0 0 0

2 3 0 0

4 5 6 0

7 8 9 10

Summary:

Upper Triangular Matrix: Stores elements on or above the diagonal, mapped to a vector.

Lower Triangular Matrix: Stores elements on or below the diagonal, mapped to a vector.

This vector representation saves memory by not storing zero elements.

UNIT – 1
COMPLETED