Data Structure

Introduction

- An efficient way of storing and organising data, so that computer can be used efficiently.

- Main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way.

- plays a vital role in enhancing the performance of a software or a program

Basic Terminology

- Data
- Group Items
- Record
- File
- Attribute and Entity
- Field

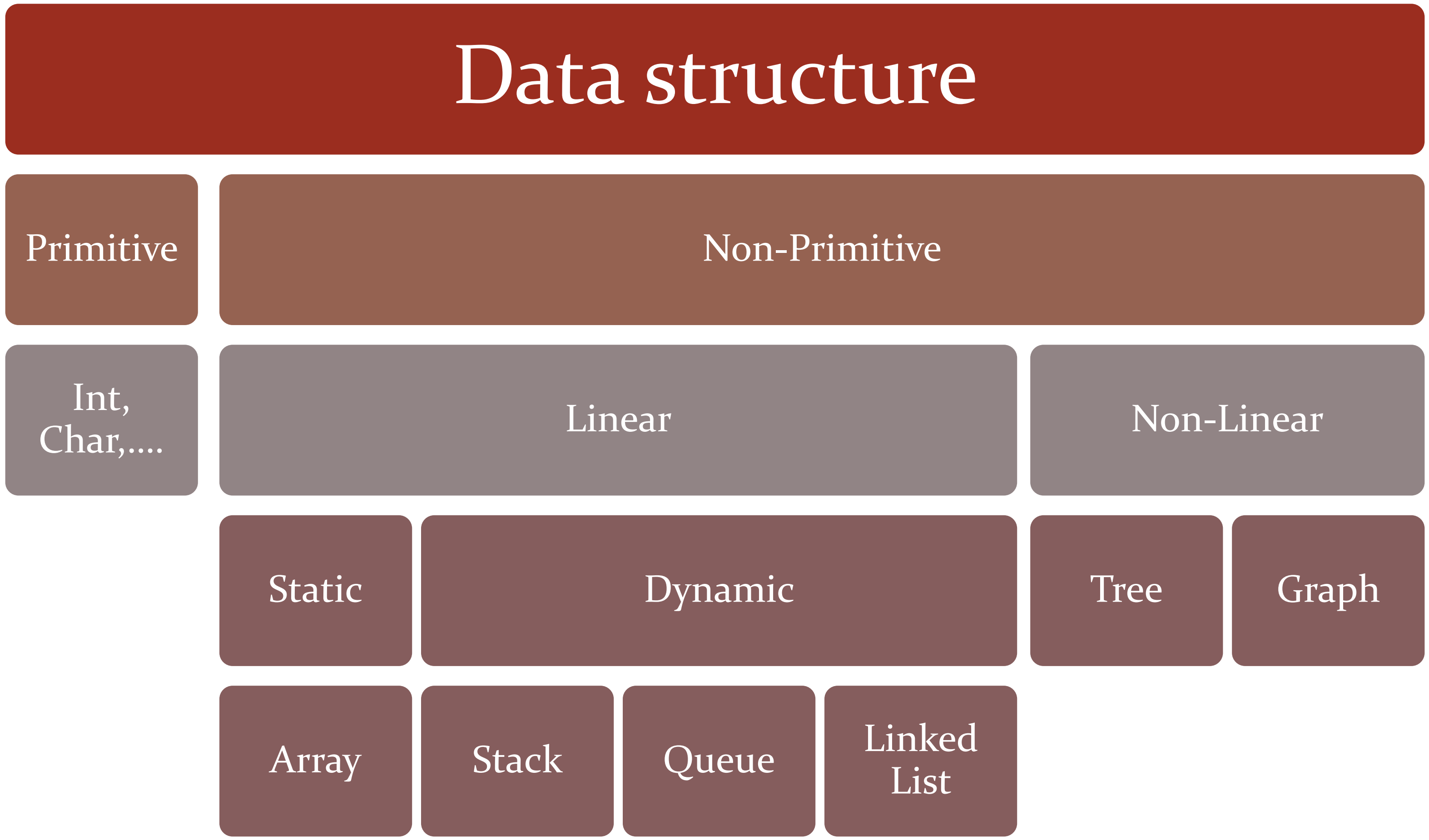
Need of Data Structures

- ✓ **Processor speed**
- ✓ **Data Search**
- ✓ **Multiple requests**

Advantages of Data Structures

- ✓ **Efficiency**
- ✓ **Reusability**
- ✓ **Abstraction**

Data Structure Classification



Operations on Data Structure

- ✓ Insertion
- ✓ Deletion
- ✓ Searching
- ✓ Sorting
- ✓ Traversing
- ✓ Merging

Algorithm

- An algorithm is a set of well-defined instructions in sequence to solve a problem.

Qualities of an Algorithm:

- ✓ Input and output should be defined precisely.
- ✓ Each step in the algorithm should be clear and unambiguous.
- ✓ Algorithms should be most effective among many different ways to solve a problem.
- ✓ An algorithm shouldn't include computer code, should be written in such a way that it can be used in different programming languages

Factors of an Algorithm

- Modularity
- Correctness
- Maintainability
- Functionality
- Robustness
- User-friendly
- Simplicity
- Extensibility

Approaches of Algorithm

- **Brute force algorithm:** an exhaustive search algorithm that searches all the possibilities to provide the required solution

- **Divide and conquer:** It allows you to break down the problem into different methods, and valid output is produced for the valid input. This valid output is passed to some other function.

- **Greedy algorithm:** It is an algorithm paradigm that makes an optimal choice on each iteration with the hope of getting the best solution.

- **Dynamic programming:** It breaks down the problem into a sub-problem to find the optimal solution, Stores the result, Reuse the result. Finally, computes the result of the complex program.

- **Branch and Bound Algorithm** This approach divides all the sets of feasible solutions into smaller subsets. These subsets are further evaluated to find the best solution.

- **Randomized Algorithm:** In a randomized algorithm, some random bits are introduced by the algorithm and added in the input to produce the output, which is random in nature. Randomized algorithms are simpler and efficient than the deterministic algorithm.

- **Backtracking:** Backtracking is an algorithmic technique that solves the problem recursively and removes the solution if it does not satisfy the constraints of a problem.

Algorithm Complexity

Performance measures of an Algorithm:

1. **Time complexity:** Time required to complete the execution.

   Falls under three category:

   a. Worst Case

   b. Average Case

   c. Best Case

2. **Space complexity:** Space required to solve a problem and produce an output.

❖ The ideal data structure is a structure that occupies the least possible time to perform all its operation and the memory space.

Asymptotic Notations

Mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value:

- Big oh Notation (O): Upper bound
- Omega Notation (Ω): Lower bound
- Theta Notation (θ): Tight bound

Array

- The collection of homogenous data items stored at contiguous memory locations.

Properties of the Array

- Each element is of same data type and carries a same size i.e. int = 4 bytes.
- Elements of the array are stored at contiguous memory locations where the first element is stored at the smallest memory location.
- Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of data element.

Complexity of Array operations

- **Time Complexity**

Algorithm	Average Case	Worst Case
Access	O(1)	O(1)
Search	O(n)	O(n)
Insertion	O(n)	O(n)
Deletion	O(n)	O(n)

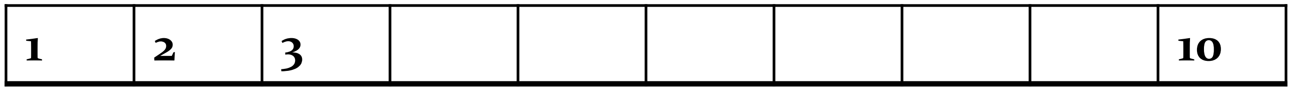
- **Space Complexity :** worst case is **O(n)**.

One Dimensional Array

- A one-dimensional array (or single dimension array) is a type of linear array.
- Accessing its elements involves a single subscript which can either represent a row or column index.

indices starting from zero through nine

```
int A[10];
```



Address of element in 1-D Array

Address of data element,

A[k] = BA(A) + w(k – lower_bound)

Ex:

Given an array int marks[] = {99,67,78,56,88,90,34,85}, calculate the address of marks[4] if the base address = 1000.

Solution:

99	67	78	56	88	90	34	85
1000	1004	1008	10012	**1016**	1020	1024	1028

We know that storing an integer value requires 4 bytes, therefore, its size is 4 bytes.

marks[4] = 1000 + 4(4 – 0)

= 1000 + 4(4) = 1016

Operation on Arrays

- Traversing an array
- Inserting an element in an array
- Searching an element in an array
- Deleting an element from an array
- Merging two arrays
- Sorting an array in ascending or descending order

Insert element in middle of an Array

```
Step 1: [INITIALIZATION] SET I = N
Step 2: Repeat Steps 3 and 4 while I >= POS
Step 3: SET A[I + 1] = A[I]
Step 4: SET I = I – 1
[END OF LOOP]
Step 5: SET N = N + 1
Step 6: SET A[POS] = VAL
Step 7: EXIT
```

The algorithm INSERT will be declared as INSERT (A, N, POS, VAL). The arguments are
(a) A, the array in which the element has to be inserted
(b) N, the number of elements in the array
(c) POS, the position at which the element has to be inserted
(d) VAL, the value that has to be inserted

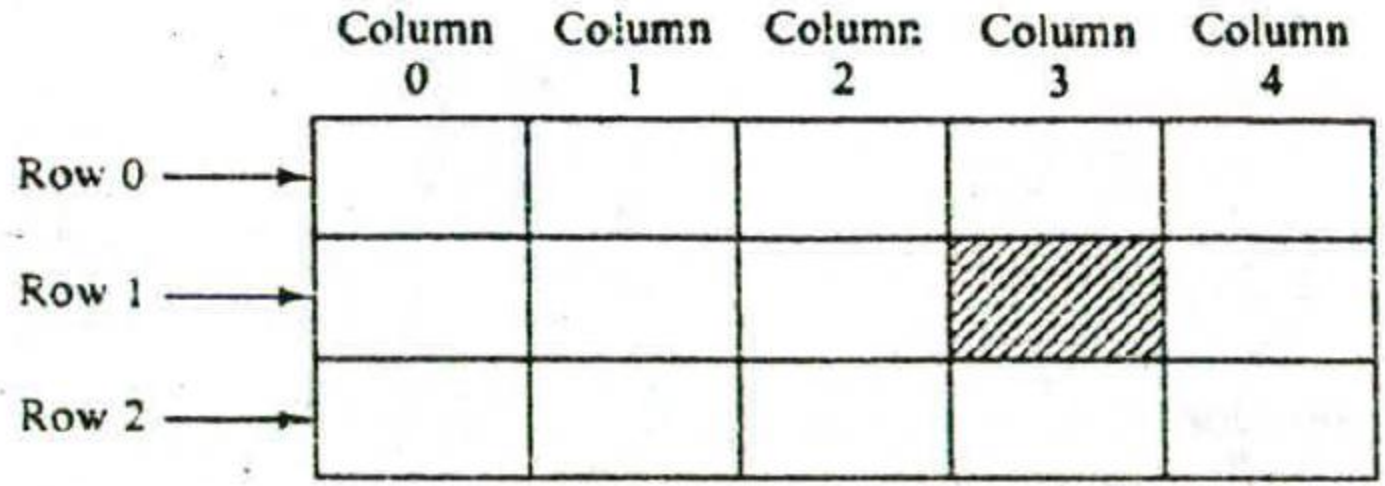
```
Void main()
{int i, n, num, pos, arr[10];
clrscr();
printf("\n Enter the number of elements in the array : ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
printf("\n arr[%d] = ", i);
scanf("%d", &arr[i]);
}
printf("\n Enter the number to be inserted : ");
scanf("%d", &num);
printf("\n Enter the position at which the number has to
     be added : ");
scanf("%d", &pos);
for(i=n–1;i>=pos;i––)
arr[i+1] = arr[i];
arr[pos] = num;
n = n+1;
printf("\n The array after insertion of %d is : ", num);
for(i=0;i<n;i++)
printf("\n arr[%d] = %d", i, arr[i]);
getch(); }
```

Output:

```
Enter the number of
elements in the array : 5
arr[0] = 1
arr[1] = 2
arr[2] = 3
arr[3] = 4
arr[4] = 5
Enter the number to be
inserted : 0
Enter the position at which
the number has to be
added : 3
The array after insertion of
0 is :
arr[0] = 1
arr[1] = 2
arr[2] = 3
arr[3] = 0
arr[4] = 4
arr[5] = 5
```

Two Dimensional Array

- ✓ 2D array can be defined as an array of arrays.

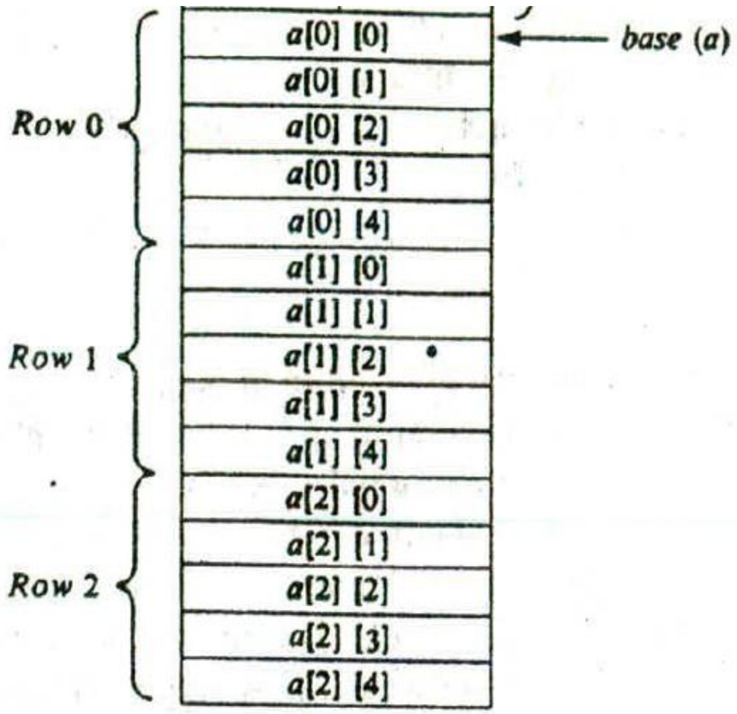


int A[3][5]

```
#include <stdio.h>
void main ()
{
    int arr[3][3],i,j;
    for (i=0;i<3;i++)
    {
        for (j=0;j<3;j++)
        {
            printf("Enter a[%d][%d]: ",i,j);

            scanf("%d",&arr[i][j]);
        }
    }
    printf("\n printing the elements ....\n");

    for(i=0;i<3;i++)
    {
        printf("\n");
        for (j=0;j<3;j++)
        {
            printf("%d\t",arr[i][j]);
        }
    }
}
```

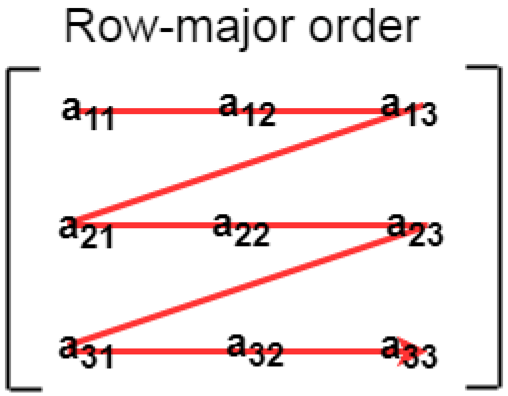


Mapping 2D array to 1D Array

- There are two main techniques of storing 2D array elements into memory

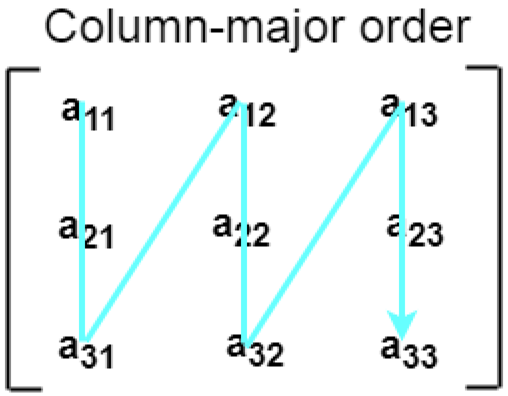
- **Row Major ordering**

Address(A[I][J]) = BA+ w{N (I – 1) + (J – 1)}



- **Column Major ordering**

Address(A[I][J]) = BA+ w{M (J – 1) + (I – 1)}



W= bytes required

M= no. of rows

N= no. of columns

Address of element in 2-D Array

Consider a 20 x 5 two-dimensional array marks which has its base address = 1000 and the size of an element = 4 Now compute the address of the element, marks[18][4] assuming that the elements are stored in row major order.

Solution

Address(A[I][J]) = BA + w{N (I – 1) + (J – 1)}

Address(marks[18][4])

= 1000 + 4 {5(18 – 1) + (4 – 1)}

= 1000 + 4 {5(17) + 3}

= 1000 + 4(88)

= 1000 + 352 = 1352

Operations on 2-D Array

- ***Transpose*** : Transpose of an m x n matrix A is given as a n x m matrix B, where Bi,j = Aj,i.
- ***Sum :*** The elements of two matrices can be added by writing: Ci,j = Ai,j + Bi,j
- ***Difference:*** The elements of two matrices can be subtracted by writing: Ci,j = Ai,j – Bi,j
- ***Product:*** Two matrices can be multiplied with each other if the number of columns in the first matrix is equal to the number of rows in the second matrix.

```
int main()
{
int arr[2][2] = {12, 34, 56,32};
int i, j;
for(i=0;i<2;i++)
{
printf("\n");
for(j=0;j<2;j++)
printf("%d\t", arr[i][j]);
}
return 0;
}
```

Output

```
12 34
56 32
```

stores the marks of five students in three subjects and display the highest marks in each subject

```
int main()
{    int marks[5][3], i, j, max_marks;
for(i=0; i<5; i++)
{  printf("\n Enter the marks obtained by student
     %d",i+1);
for(j=0; j<3; j++)
{ printf("\n marks[%d][%d] = ", i, j);
scanf("%d", &marks[i][j]);
} }
for(j=0; j<3; j++)
{
max_marks = 0;
for(i=0; i<5; i++)
{
if(marks[i][j]>max_marks)
max_marks = marks[i][j];
}
printf("\n The highest marks obtained in the subject
     %d = %d", j+1, max_marks);
}
getch();
return 0; }
```

Output

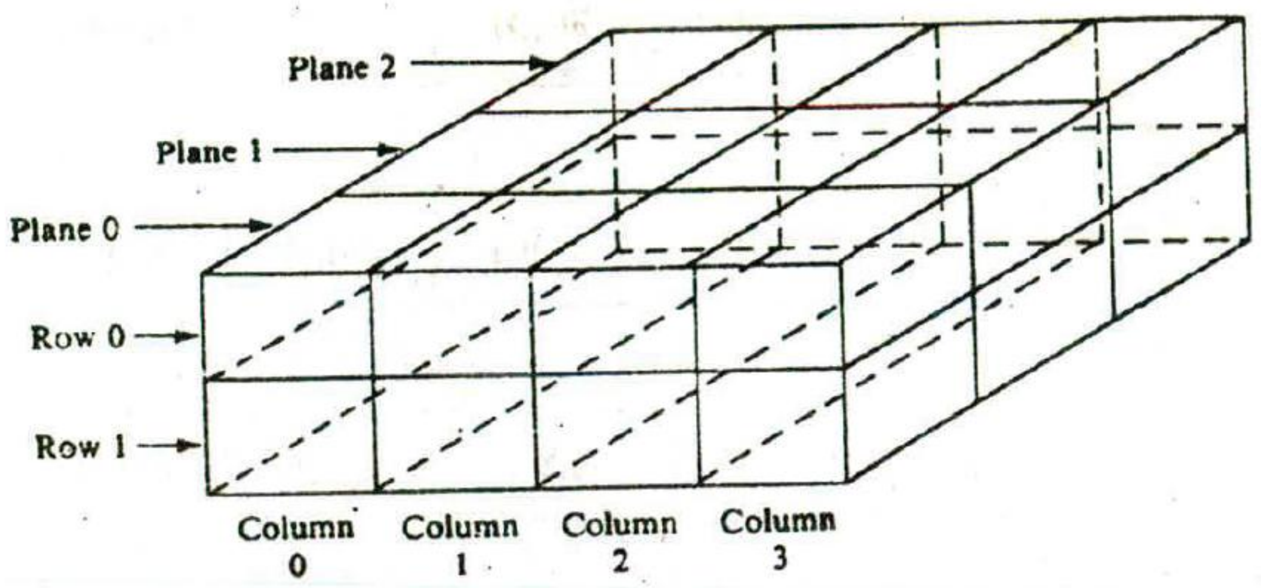
```
Enter the marks obtained by student 1
marks[0][0] = 89
marks[0][1] = 76
marks[0][2] = 100
Enter the marks obtained by student 2
marks[1][0] = 99
marks[1][1] = 90
marks[1][2] = 89
Enter the marks obtained by student 3
marks[2][0] = 67
marks[2][1] = 76
marks[2][2] = 56
Enter the marks obtained by student 4
marks[3][0] = 88
marks[3][1] = 77
marks[3][2] = 66
Enter the marks obtained by student 5
marks[4][0] = 67
marks[4][1] = 78
marks[4][2] = 89


The highest marks obtained in the subject 1 = 99
The highest marks obtained in the subject 2 = 90
The highest marks obtained in the subject 3 = 100
```

N-D Array

Array with n size of row, column and spaces.

int a[2][3]........[6];


Plane 2
Plane 1
Plane 0
Row 0
Row 1
Column 0
Column 1
Column 2
Column 3